Boundary-Scan Tutorial

See the ASSET homepage on the World Wide Web at
http://www.asset-intertech.com

Table of Contents

Table of Figures

Introduction

In this tutorial, you will learn the basic elements of boundary-scan architecture — where it came from, what problem it solves, and the implications on the design of an integrated-circuit device. This tutorial also provides an overview of the data standards applicable to the boundary-scan architecture and an overview of the software tools available to perform boundary-scan-based tests.

The core reference is the standard:

> IEEE Standard 1149.1-1990 “Test Access Port and Boundary-Scan Architecture,” available from the IEEE, 445 Hoes Lane, PO Box 1331, Piscataway, New Jersey 08855-1331, USA.

The standard was revised in 1993 and again in 1994. You can also obtain a copy of the standard via the WWW on the IEEE home page at: http://standards.ieee.org/catalog.

The 1993 revision to the standard, referred to as “1149.1a,” contained many clarifications, corrections, and minor enhancements. Two new instructions were introduced in 1149.1a and these are described in this tutorial.

The 1994 supplement contains a description of the Boundary-Scan Description Language (BSDL).

For further, more recent publications on the boundary-scan architecture, see the Bibliography at the end of this tutorial.

Chapter 1: The Motivation for Boundary-Scan Architecture

Since the mid-1970s, the structural testing of loaded printed circuit boards (PCBs) has relied very heavily on the use of the so-called in-circuit “bed-of-nails” technique (Figure 1). This method of testing makes use of a fixture containing a bed-of-nails to access individual devices on the board through test lands laid into the copper interconnect, or other convenient contact points. Testing then proceeds in two phases: the power-off tests followed by power-on tests. Power-off tests check the integrity of the physical contact between nail and the on-board access point. They then carry out open and shorts tests based on impedance measurements.

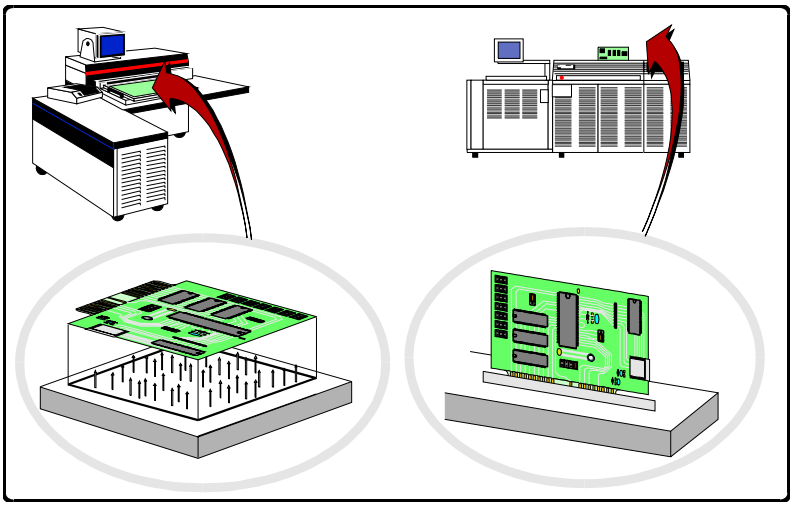


Figure 1: ICT vs. Functional Test

Power-on tests apply stimulus to a chosen device on a board, with an accompanying measurement of the response from that device. Other devices that are electrically connected to the device-under-test are usually placed into a safe state (a process called “guarding”). In this way, the

tester is able to check the presence, orientation, and bonding of the device-under-test in place on the board.

Fundamentally, the in-circuit bed-of-nails technique relies on physical access to all devices on a board. For plated-through-hole technology, the access is usually gained by adding test lands into the interconnects on the “B” side of the board — that is, the solder side of the board. The advent of onserted devices (surface mount) meant that manufacturers began to place components on both sides of the board — the “A” side and the “B” side. The smaller pitch between the leads of surface-mount components caused a corresponding decrease in the physical distance between the interconnects. This had serious impact on the ability to place a nail accurately onto a target test land. The whole question of access was further compounded by the development of multi-layer boards.

Such was the situation in the mid-1980s when a group of concerned test engineers in a number of European electronics systems companies got together to examine the problem and its possible solutions. The group of people called themselves the Joint European Test Action Group (JETAG). Their preferred method of solution was based on the concept of a serial shift register around the boundary of the device — hence the name “boundary scan.” Later, the group was joined by representatives from North American companies and the ‘E’ for “European” was dropped from the title of the organization leaving it Joint Test Action Group (JTAG). This was the organization that finally converted their ideas into an international standard.

Chapter 2: The Principle of Boundary-Scan Architecture

Each primary input signal and primary output signal is supplemented with a multi-purpose memory element called a boundary-scan cell. Cells on device primary inputs are referred to as “input cells;” cells on primary outputs are referred to as “output cells.” “Input” and “output” is relative to the core logic of the device. (Later, we will see that it is more convenient to reference the terms “input” and “output” to the interconnect between two or more devices.) See Figure 2.

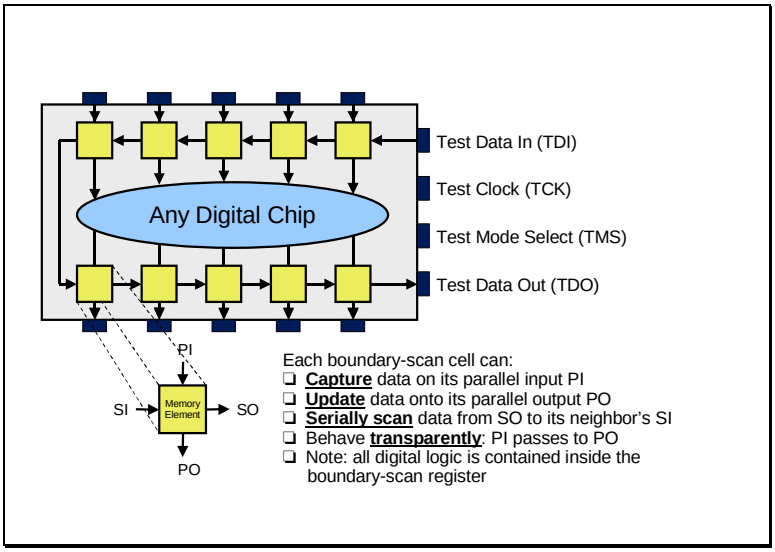


Figure 2: Principle of Boundary-Scan Architecture

The collection of boundary-scan cells is configured into a parallel-in, parallel-out shift register. A parallel load operation, called a “capture” operation, causes signal values on device input pins to be loaded into input cells and, signal values passing from the core logic to device output pins to be loaded into output cells. A parallel unload operation —

called an “update” operation — causes signal values already present in the output scan cells to be passed out through the device output pins. Signal values already present in the input scan cells will be passed into the core logic.

Data can also be shifted around the shift register, in serial mode, starting from a dedicated device input pin called “Test Data In” (TDI) and terminating at a dedicated device output pin called “Test Data Out” (TDO). The test clock, TCK, is fed in via yet another dedicated device input pin and the mode of operation is controlled by a dedicated “Test Mode Select” (TMS) serial control signal.

Using the Scan Path

At the device level, the boundary-scan elements contribute nothing to the functionality of the core logic. In fact, the boundary-scan path is independent of the function of the device. The value of the scan path is at the board level as shown in Figure 3.

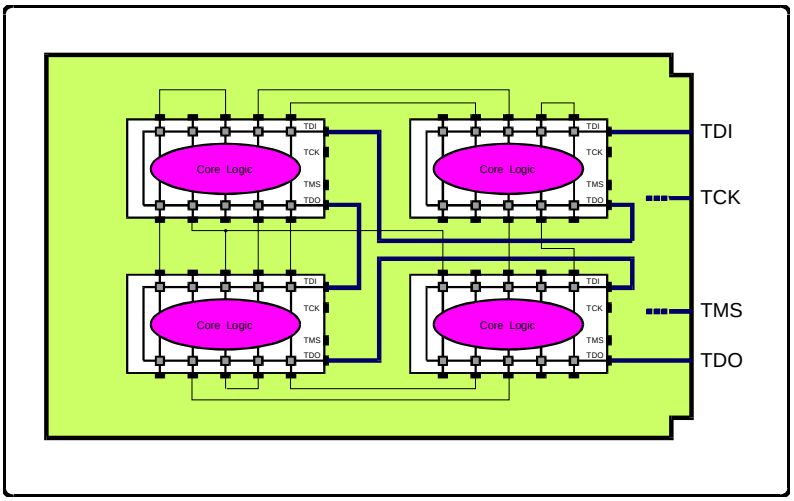


Figure 3: Using the Boundary-Scan Path

Figure 3 shows a board containing four boundary-scan devices. Notice that there is an edge-connector input called TDI connected to the TDI of the first device. TDO from the first device is connected to TDI of the second device, and so on, creating a global scan path terminating at the edge connector output called TDO. TCK is connected in parallel to each device TCK input, TMS works similarly.

In this way, particular tests can be applied to the device interconnects via the global scan path — by loading the stimulus values into the appropriate device-output scan cells via the edge connector TDI (shift-in operation), applying the stimulus (update operation), capturing the responses at device-input scan cells (capture operation), and shifting the response values out to the edge connector TDO (shift-out operation).

Essentially, boundary-scan cells can be thought of as “virtual nails.”

Figure 4 shows a basic universal boundary-scan cell. It has four modes of operation: normal, update, capture, and serial shift. The memory element is shown to be a simple D-type flip-flop with front-end and back-end multiplexing of data. (As with all circuits in this tutorial, it is important to note that the circuit shown in Figure 4 is only an example of how the requirement defined in the Standard could be realized. The IEEE 1149.1 Standard does not mandate the design of the circuit, only its functional specification.)

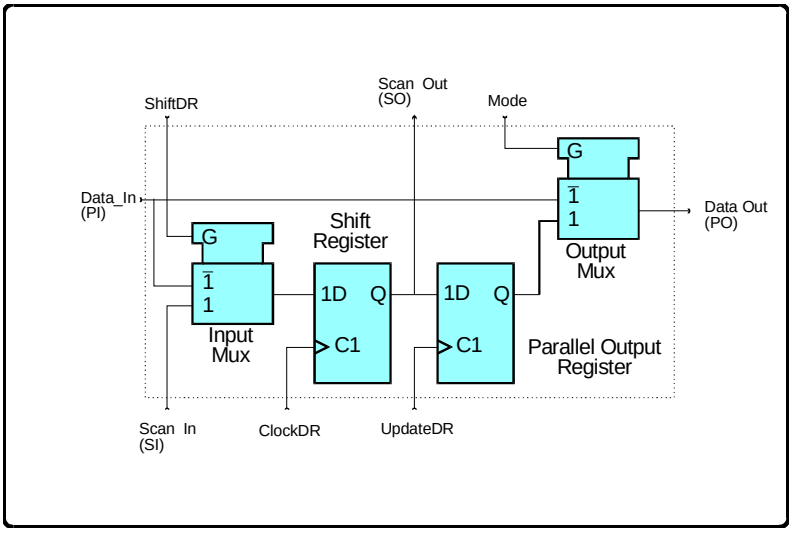
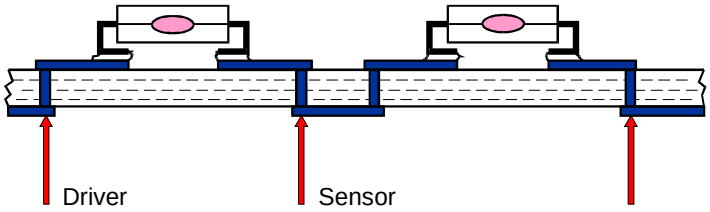


Figure 4: Basic Boundary-Scan Cell

During normal mode, Data_In is passed straight through to Data_Out. During update mode, the content of the output register is passed through to Data_Out. During capture mode, the Data_In signal is routed to the shift register and the value is captured by the next ClockDR. During shift mode, the Scan_Out of one register flip-flop is passed to the Scan_In of the next via a hard-wired path. Note that both capture and shift operations do not interfere with the normal passing of data from the parallel-in terminal to the parallel-out terminal. This allows the capture of operational values “on the fly” and the movement of these values for inspection without interference. This application of the boundary-scan architecture has tremendous potential for real-time monitoring of the operational status of a system — a sort of electronic camera taking snapshots — and is one reason why TCK is kept separate from any system clocks.

The use of boundary-scan cells to test the presence, orientation, and bonding of devices in place was the original motivation for inclusion in a device. The use of scan cells as a means of applying tests to individual devices is not the major application of boundary-scan architecture. Consider the reason for boundary-scan architecture in the first place.

The prime function of the bed-of-nails in-circuit tester was to test for manufacturing defects, such as missing devices, damaged devices, open and short circuits, misaligned devices, and wrong devices. See Figure 5.



Defects covered:

nail - plated-through hole - interconnect - solder - leg - bond wire - device - bond wire - leg - solder - interconnect - plated-through hole - nail

Figure 5: Bed-of-Nails Fault Coverage

In-circuit testers were not intended to prove the overall functionality of the devices. It was assumed that devices had already been tested for functionality when they existed only as devices (i.e., prior to assembly on the board). Unfortunately, in-circuit test techniques had to make use of device functionality in order to test the interconnect structure — hence the rather large libraries of merchant device functions and the problems caused by increasing use of ASICs.

Given that boundary-scan architecture was seen as an alternative way of testing for the presence of manufacturing defects, we should question what these defects are, what causes them, and where they occur.

An examination of the root cause for defects shows them to be caused by any one of three “shock waves”: electrical shock (e.g., electrostatic discharge), mechanical shock

(e.g., clumsy handling), or thermal shock (e.g., hot spots caused by the solder operation). A defect, if it occurs, is likely present either in the periphery of the device (leg, bond wire, driver amplifier), in the solder, or in the interconnect between devices. It is very unusual to find damage to the core logic without there being some associated damage to the periphery of the device.

In this respect, the boundary-scan cells are precisely where we want them — at the beginning and ends of the core function of the device (see Figure 6)

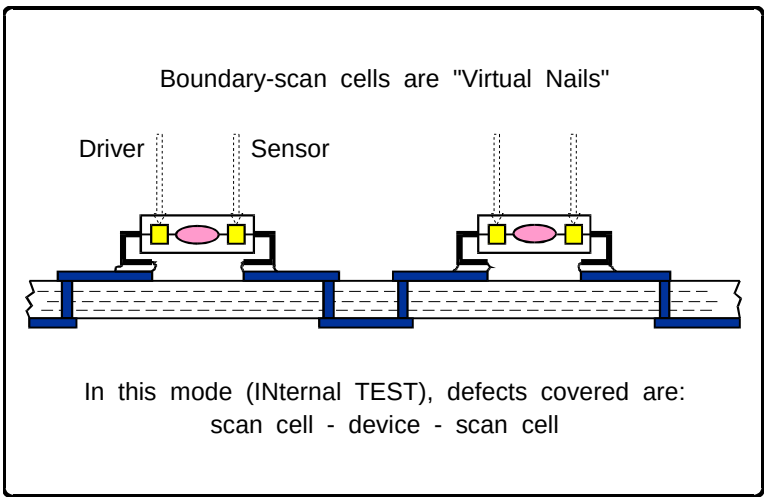
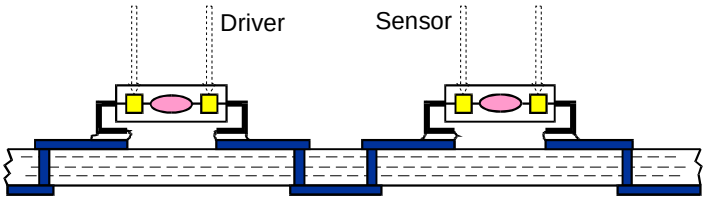


Figure 6: Boundary-Scan Fault Coverage (*Intest*)

and at the beginning and end of interconnect paths (see Figure 7).



In this mode (EXternal TEST), defects covered are:
scan cell - driver - bond wire - leg - solder - interconnect - solder - leg - bond wire - driver - scan cell

Figure 7: Boundary-Scan Fault Coverage (*Extest*)

Using the boundary-scan cells to test the core functionality is called “internal test,” shortened to *Intest*. Using the boundary-scan cells to test the interconnect structure between two devices is called “external test,” shortened to *Extest*. The use of the cells for *Extest* is the major application of boundary-scan architecture, searching for opens and shorts plus damage to the periphery of the device. *Intest* is only really used for very limited testing of the core functionality (i.e., an existence test — “are you there, are you alive?”) to identify defects such as devices missing, incorrectly oriented, or misalignment.

Chapter 3: IEEE 1149.1 Device Architecture

After nearly five year's discussion, the JTAG organization finally proposed the architecture shown in Figure 8.

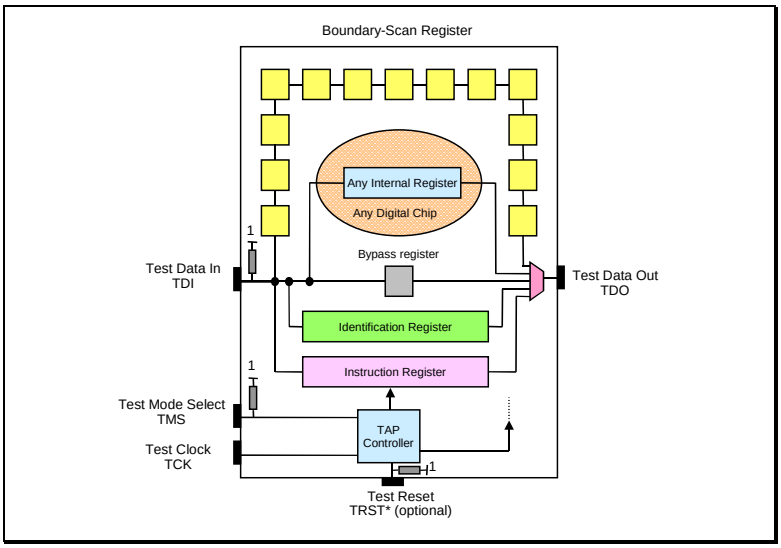


Figure 8: IEEE 1149.1 Chip Architecture

Figure 8 shows the following elements:

- A set of four dedicated test pins — Test Data In (TDI), Test Mode Select (TMS), Test Clock (TCK), Test Data Out (TDO) — and one optional test pin Test Reset (TRST*). These pins are collectively referred to as the Test Access Port (TAP).
- A boundary-scan cell on each device primary input and primary output pin, connected internally to form a serial boundary-scan register (Boundary Scan).
- A finite-state machine TAP controller with inputs TCK, TMS, and TRST*.
- An n-bit ($n \geq 2$) Instruction Register (IR), holding the current instruction.

- A 1-bit bypass register (Bypass).
- An optional 32-bit Identification Register (Ident) capable of being loaded with a permanent device identification code.

At any time, only one register can be connected from TDI to TDO (e.g., IR, Bypass, Boundary-scan, Ident, or even some appropriate register internal to the core logic). The selected register is identified by the decoded output of the IR. Certain instructions are mandatory, such as *Extest* (boundary-scan register selected), whereas others are optional, such as the *Idcode* instruction (Ident register selected).

Let's take a closer look at each part of this architecture.

The Instruction Register

An Instruction Register (IR) has a shift section that can be connected to TDI and TDO, and a hold section, holding the current instruction as shown in Figure 9.

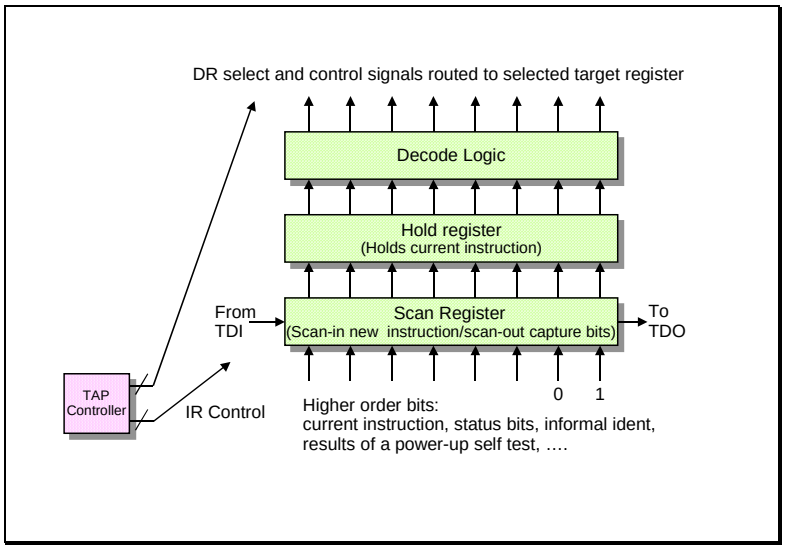


Figure 9: The Instruction Register

There may be some decoding logic between the two sections depending on the width of the register and number of different instructions. The control signals to the IR originate from the TAP controller and either cause a shift-in, shift-out through the IR shift section, or cause the contents of the shift section to be passed across to the hold section (update operation). It is also possible to load (capture) certain hard-wired values into the shift section of the IR. The IR must be at least two-bits long (to allow coding of the three mandatory instructions — *Bypass*, *Sample/Preload*, *Extest* — but the maximum length of the IR is not defined. In capture mode, the two least significant bits must capture a 01 pattern (see Figure 9). The values captured into higher-order bits are not defined. One possible use of these higher order bits is to capture an informal identification code if the 32-bit Ident register is not implemented. In practice, the only mandated bits for IR capture is the 01 pattern. We will return to the value of capturing this pattern later in this tutorial.

The Instructions

The IEEE 1149.1 Standard describes three mandatory instructions: *Bypass*, *Sample/Preload*, and *Extest*.

The *Bypass* instruction must be assigned an all-1s code and when executed, causes the Bypass register to be placed between the TDI and TDO pins. By definition, the initialized state of the hold section of the IR should contain the *Bypass* instruction unless the optional Identification Register (Ident) has been implemented, in which case, the *Idcode* instruction should be present in the hold section.

The *Sample/Preload* instruction selects the boundary-scan register when executed. The instruction sets up the boundary-scan cells either to sample (capture) values moving in to the device or to preload known values into the output boundary-scan cells prior to some follow-on operation. The code for the *Sample/Preload* instruction is not defined.

The *Extest* instruction selects the boundary-scan register when executed, preparatory to interconnect testing. The code for *Extest* is defined as the all-0s code.

The IEEE 1149.1 Standard defines a number of optional instructions (instructions that do not need to be implemented, but which have a prescribed operation when they are used). Examples of optional instructions include:

Intest, the instruction that selects the boundary-scan register preparatory to applying tests to the core logic of the device.

Idcode, the instruction to select the Identification Register between TDI and TDO, preparatory to loading the *Idcode* code and reading it out through TDO. Note that if the *Idcode* instruction is loaded and there is no Identification Register present on the device, then the *Idcode* instruction must be interpreted as if it were the *Bypass* instruction.

Runbist, the instruction to initiate an internal self-test routine and to place the pass/fail result register between TDI and TDO.

Two new instructions introduced in the 1993 revision, 1149.1a, were *Clamp* and *Highz*. *Clamp* is an instruction that drives preset values onto the outputs of devices (established initially with the *Sample/Preload* instruction) and then selects the Bypass register between TDI and TDO (unlike the *Sample/Preload* instruction). *Clamp* would be used to set up safe “guarding” values on the outputs of certain devices in order to avoid bus contention problems, for example.

Highz is similar to *Clamp*, but it leaves the device output pins in a high-impedance state. *Highz* also selects the Bypass register between TDI and TDO.

With the exception of *Extest* and *Bypass*, the codes for all instructions are undefined. Given the need for three mandatory instructions, the minimum length of the IR is two bits. The maximum length is undefined. Any instruction can have more than one code and all unused codes are interpreted as *Bypass*. Note that the designer may use certain codes to implement “private” instructions — that is, instructions whose functions are not made public. In these circumstances, the designer must state that these codes are private so that the user can avoid loading the codes.

Using the Instruction Register (IR)

Before proceeding with a description of other parts of the architecture, we will first examine how to load the IR and decode its contents. Consider the board circuit shown in Figure 10.

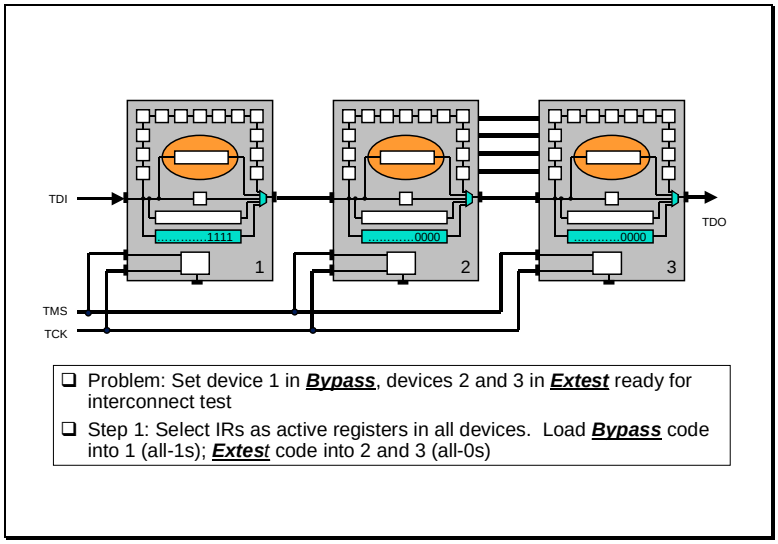


Figure 10: Using the Instruction Register — Step 1

Assume that what we want to do is to place Chip 1 into bypass mode (to shorten the time it takes to get test stimulus to follow-on devices) and place chips 2 and 3 into Extest mode preparatory to setting up tests to check the interconnect between Chips 2 and 3. This set-up requires loading the *Bypass* instruction (all-1s) into the IR of chip 1, and the *Extest* instruction (all-0s) into the IRs of Chips 2 and 3.

Step 1 is to connect the IRs of all three devices between their respective TDI and TDO pins. This is achieved by a special sequence of values on the serial control line TMS going to each TAP controller. Note that the TMS (and TCK) lines are connected to all devices in parallel. Any sequence of values on TMS will be interpreted in the same way by each TAP controller. Later, we will see the precise TMS sequence to select the IR between TDI and TDO. For now, we will assume that such a sequence exists.

Step 2 is to load the appropriate instructions into the various IRs via the global connection of IRs. If we assume simple two-bit IRs per device, this operation amounts to a serial load of the sequence 110000 into the edge-connector TDI to place 00 in the IRs of Chips 2 and 3, and 11 in the IR of Chip 1. The IRs are now set up with the correct instructions loaded in their shift sections.

Step 3, shown in Figure 11, is to continue with values on TMS to cause each TAP controller to issue the control-signal values to transfer the values in the shift sections of the IRs to the hold sections where they become the current instruction. This is the Update operation. At this point, the various instructions are obeyed — that is, Chip 1 deselects the IR and selects the Bypass register between TDI and TDO (*Bypass* instruction), and Chips 2 and 3 deselect their IRs and select their boundary-scan registers between TDI and TDO (*Extest* instruction). The devices are now set up ready for Extest operation.

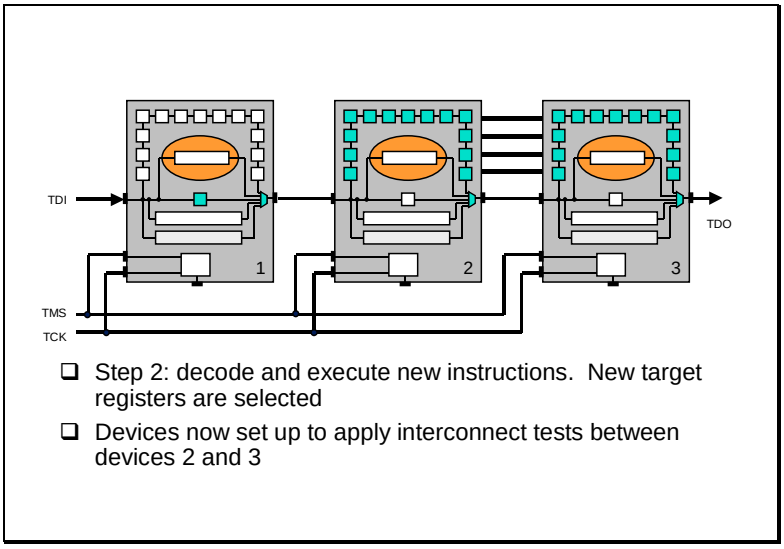


Figure 11: Using the Instruction Register — Step 3

Use of the "Capture 01" Mode

Previously we discussed the capture of the fixed 01 pattern into the least two significant positions of the Instruction Register. Normally, we would think only of "shift and update" operations for the IR. The question arises — what is the use of the "capture 01" pattern?

To answer this question, we need to think about the use of boundary-scan architecture at the board level. Consider again the circuit in Figure 10.

Previously, we saw how to set up a test environment preparatory to carrying out interconnect tests. To do this, we made use of the test infrastructure (i.e., the on-chip boundary-scan features plus the board-level TMS and TCK connections and the chip-to-chip TDO-to-TDI interconnects). It is important to know that this infrastructure is fault-free before making use of it. In other words, we must first "test the tester" before using the tester to test other parts of the board. This is the purpose of the IR capture 01 operation.

Essentially, what happens is as follows:

Step 1: Apply the sequence to TMS, which causes each device to place the IR between TDI and TDO. At this stage, there is a serial shift register that starts at the board TDI input and ends at the board TDO output and which is made up of the various IRs in the devices — an IR chain.

Step 2: Apply an additional sequence to TMS to cause each IR to capture the hardwired 01 into the least two significant positions of the IR. Higher-order bits capture what they are set up to capture. These values are not mandated by the Standard. The captured 01 values constitute a checkerboard "flush" test for the serial IR chain.

Step 3: Clock the captured values out of the IR chain to the board's TDO output.

If the sequence TDO: 10…10…10… emerges, then we can be reasonably sure of the following facts:

- The TMS control signal is properly connected from the board's TMS input to the TMS inputs of every device.
- The TCK control signal is properly connected from the board's TCK input to the TCK inputs of every device.
- The TDO from one device is properly connected to the TDI of its logical neighbor.
- Each internal TAP controller is at least capable of responding correctly to the sequences on TMS that cause the IR both to capture and to shift.

It is usual to feed the inverse values 10 into the board TDI input so as to know when to terminate the shift-out phase (Step 3). These bits are called the "sentinel" bits. They have an added benefit as they help to remove a possible cause of incorrect diagnosis if there is a TDI-to-TDO short circuit on one of the devices.

Steps 1 to 3 represent a minimum integrity test for the boundary-scan infrastructure. Additional tests can be included. For example: load and execute the *Bypass* instruction into all devices to show that the bypass registers are functioning correctly; load an instruction (e.g., *Extest*) to select the boundary-scan register and pass a flush test through the register to check the integrity of the boundary-scan cells. The question that is raised is why do all these additional integrity tests? If our purpose is just to test for manufacturing defects on the test infrastructure, the IR checkerboard test is probably sufficient. All additional integrity tests deal with testing the functionality of the IEEE 1149.1 features on the devices. We could argue that this is more a chip test requirement, not a board test requirement (in fact, the same argument used earlier to explain why the *Intest* instruction is not mandatory).

Most test engineers run the extra integrity tests as time permits. These tests provide additional confidence that the test infrastructure is healthy before using it to test other parts of the board.

The Test Access Port (TAP)

We return now to the TAP and its controller (Figure 12). The TAP consists of four mandatory terminals plus one optional terminal.

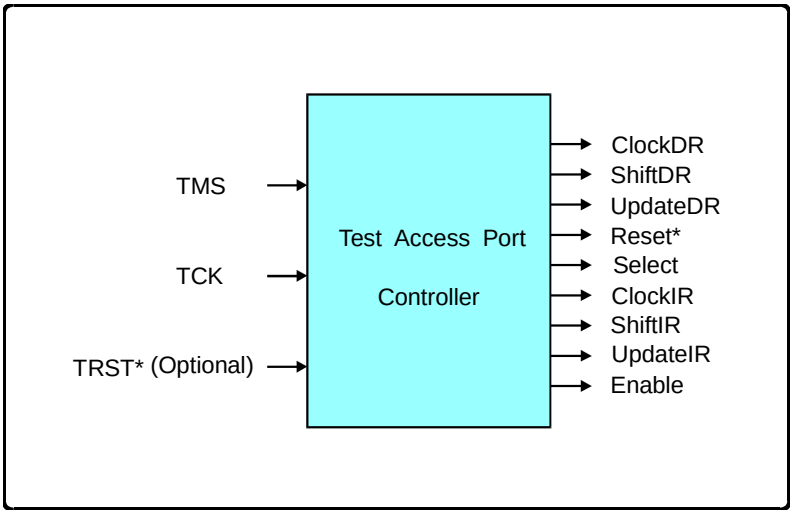


Figure 12: TAP Controller Global View

The mandatory terminals are:

- Test Data In (TDI): serial test data in with a default value of 1.
- Test Data Out (TDO): serial test data out with a default value of Z and only active during a shift operation.
- Test Mode Select (TMS): serial input control signal with a default value of 1.
- Test Clock (TCK): dedicated test clock, any convenient frequency.

The optional terminal is:

- Test Reset (TRST*): asynchronous TAP controller reset with default value of 1 and active low.

TMS and TCK (and the optional TRST*) go to a finite-state machine controller, which produces the various control signals. These signals include dedicated signals to the IR (ClockIR, ShiftIR, UpdateIR) and generic signals to all data registers (ClockDR, ShiftDR, UpdateDR). The data register that actually responds is the one enabled by the conditional

control signals generated at the parallel outputs of the IR, according to the particular instruction. Additionally, there are generic Select, Reset, and Enable signals.

Figure 13 shows the state table for the TAP controller. The value on the state transition arcs is the value of TMS. A state transition occurs on the positive edge of TCK and output values change on the negative edge of TCK.

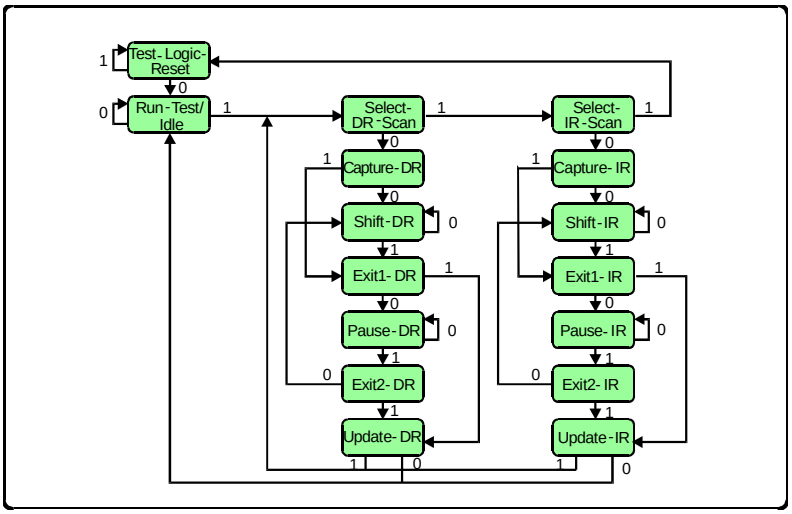


Figure 13: TAP Controller State Table Diagram

The TAP controller initializes in the *Test-Logic-Reset* state ("Asleep" state). While TMS remains a 1 (the default value), the state remains unchanged. Pulling TMS low causes a transition to the *Run-Test/Idle* state ("Awake, and do nothing" state). Normally, we want to move to the *Select-IR-Scan* state ready to load and execute a new instruction.

An additional one-one sequence on TMS will achieve this. From here, we can move through the various *Capture-IR*, *Shift-IR*, and *Update-IR* states as required. The last operation is the *Update-IR* operation and, at this point, the instruction loaded into the shift section of the IR is transferred to the hold section to become the current instruction. This causes the IR to be deselected as the register connected between TDI and TDO and the data

register identified by the current instruction to be selected as the new target register between TDI and TDO (e.g., if the instruction is *Bypass*, the Bypass register is the selected data register). From now on, we can manipulate the target data register with the generic *Capture-DR*, *Shift-DR*, and *Update-DR* control signals.

Note that there is no master reset to the TAP controller if the optional TRST* is not implemented. The TAP controller is mandated to power up in the *Test-Logic Reset* state. If there is a need to re-initialize the controller, it can be done by holding TMS high and clocking TCK up to a maximum of five clocks. In general, TMS = 0 holds the current state whereas TMS = 1 causes a state transition. The reader is invited to verify that from any start state, five TCKs is sufficient to return the controller to the *Test-Logic-Reset* state, given that TMS remains at logic 1.

Each of the main branches of the state table contains additional *Exit* and *Pause* states. The *Exit1* state allows a transition from the shift operation to *Update*. It also allows the controller to be placed in a *Pause* state. This might be necessary if, for example, all devices have their boundary-scan registers selected as the data registers and an external tester pin channel is either loading or unloading test data (e.g., as in the use of *Extest* to test interconnect structures). If the length of the chained boundary-scan registers is longer than the memory associated with the tester channel then it will become necessary to update or unload the content of the channel memory before resuming the shift operation through the boundary-scan path. The *Pause* state enables this action and *Exit2* state allows a return to the shift operation.

In general, a TAP controller requires four state flip-flops and another four flip-flops to hold the values of certain output signals. The additional next-state decoder and output decoder logic adds another 20 to 40 gates.

The Bypass Register

Figure 14 shows a typical design for a Bypass register. It is a 1-bit register, selected by the *Bypass* instruction and provides a basic shift function. There is no parallel output (which means that the *Update-DR* control has no effect on the register), but there is a defined effect with the *Capture-DR* control — the register captures a hard-wired value of 0. We will shortly explain the value of this.

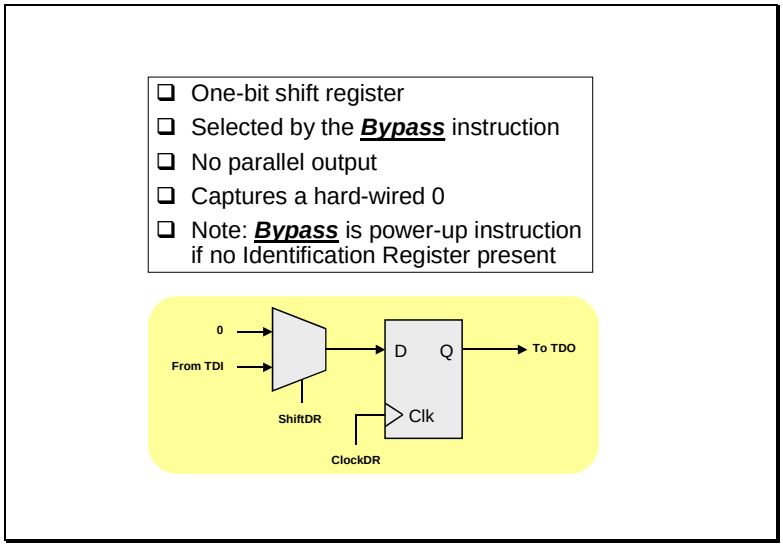


Figure 14: The Bypass Register

The Identification Register

The optional Identification (Ident) register is a 32-bit register with capture and shift modes of operation (Figure 15). The captured 32 bits identify the device through the following fields:

- Bit 0 (least significant bit) is always 1.
- Bits 1 - 11 identify the manufacturer of the device using a compact form of the JEDEC identification code.
- Bits 12 - 27 provide a 16-bit free format part number field.

- Bits 28 - 31 provide a 4-bit free format field to specify up to 16 different versions of the same basic device.

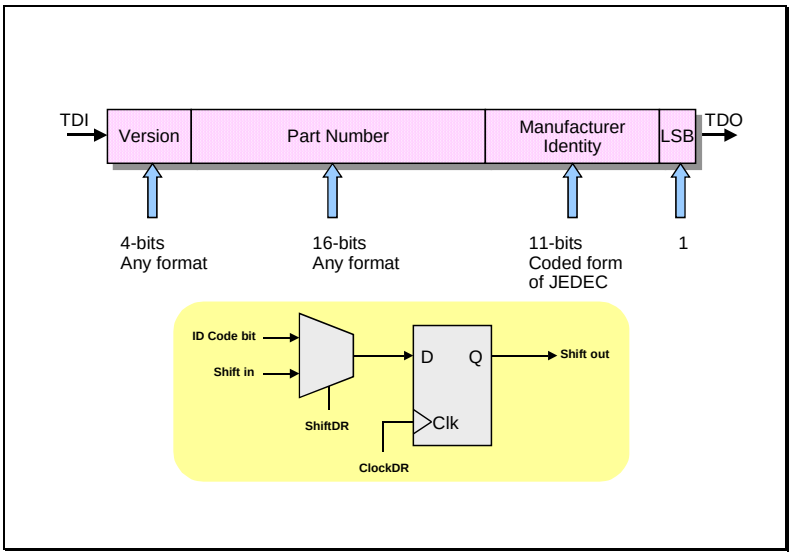


Figure 15: Device Identification Code Structure

Once captured, the 32-bit identification code can be shifted out through TDO for inspection. Figure 15 also shows a possible implementation of one cell in the 32-bit register.

We will now investigate why the least significant bit (lsb) of the Ident register is a 1 and why the Bypass register captures a hard-wired value of 0.

Use of the lsb = 1 Feature

Consider the following field servicing scenario. A customer's computer system has broken down. The cause is suspected to be a hardware fault on a particular board. There are many variations of the board and the service engineer needs to identify the board type and the component versions. All the engineer knows is that there are boundary-scan components on the board and the location of the primary (edge-connector) TDI, TDO, TMS, TCK ports plus

Power and Ground. The following procedure identifies the boundary-scan components on the board and whether or not they have Ident registers.

Step 1: Power up the board and sequence values on TMS to enter the *Select DR-Scan* state. By default, the instruction loaded into the hold stage of every boundary-scan device on power-up must be *Idcode* if the device contains an *Ident* register, or *Bypass* if the device does not contain an *Ident* register. This is mandated by the standard. This is shown in Figure 16.

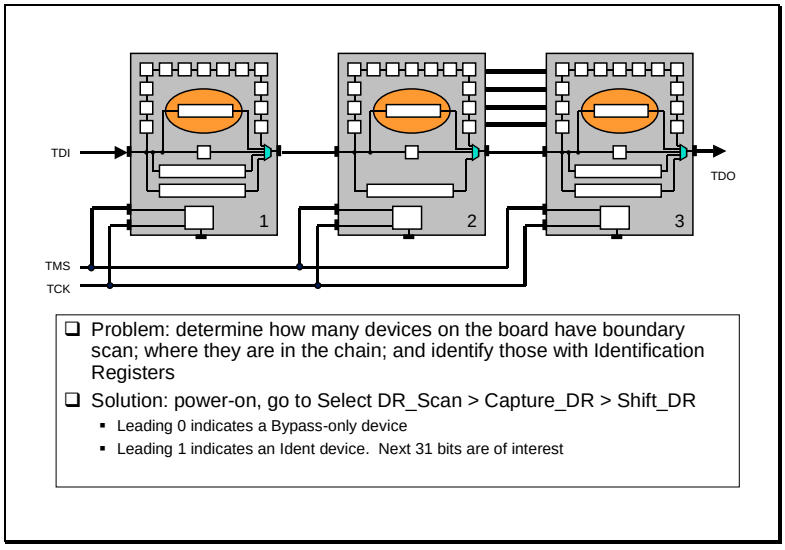


Figure 16: Use of the lsb = 1 Feature — Step 1

Step 2: Capture the hard-wired values (*Capture-DR*) in the default selected *Bypass* or *Ident* register.

Step 3: Shift (*Shift-DR*) the captured values out through the primary TDO output. See Figure 17. A leading 0 identifies a device without an *Ident* register. A leading 1 identifies a device with an *Ident* register, in which case the next 31 bits are of interest.

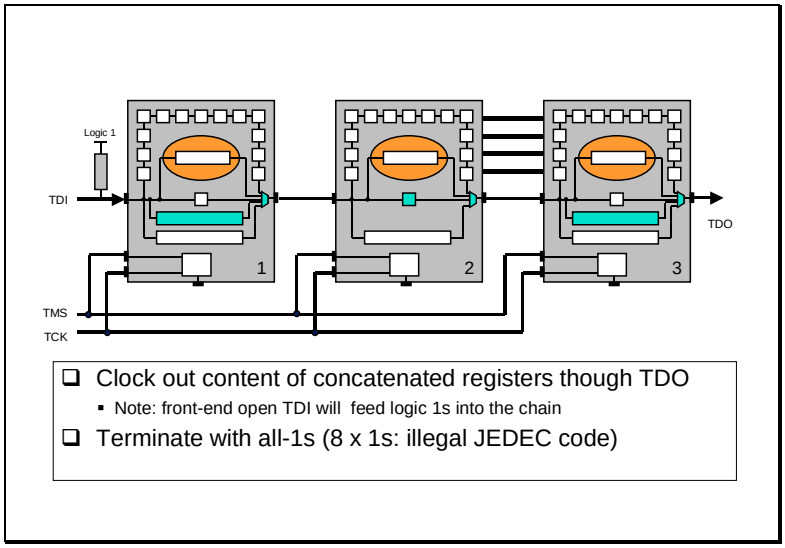


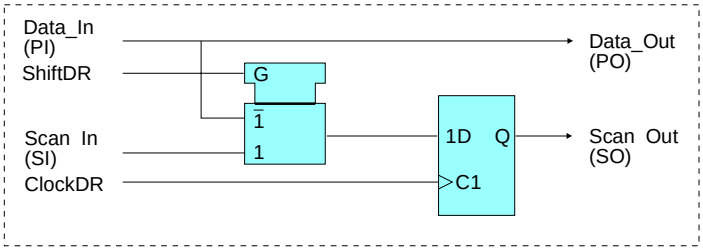
Figure 17: Use of the lsb = 1 Feature — Step 3

In the situation of a true "blind" interrogation (i.e., one in which it is not known how many devices on the board have IEEE 1149.1 features), the process can be terminated by feeding in an illegal sequence through the primary TDI and waiting for this sequence to appear at the primary TDO. Such a sequence is seven consecutive 1s in bits 1-7 of the manufacturer identity field. The JEDEC coding system avoids this sequence. It is usual to add a further 0 to this sequence just in case the primary TDI is stuck-at-1. See Figure 17.

Boundary-Scan Register

We are now ready to take a more detailed look at the boundary-scan cells. Boundary-scan cells are placed on the device signal input ports, output ports, and on the control lines of bidirectional (I/O) ports and tristate (0, 1, Z) ports. These cells are linked together to form the boundary-scan register. The order of linking is determined by the physical adjacency of the pins and/or by other layout constraints.

The boundary-scan register is selected by the *Extest*, *Sample/Preload*, and *Intest* instructions.
There are many different designs for boundary-scan cells. Figure 18 shows a simple design capable only of capture and shift operations. Such a cell could be used on device inputs that are especially sensitive to extra loading on the Data_In signal e.g., a system clock. (Note: the three mandatory instructions do not require an update operation on the input scan cells.)



Only Capture and Shift: No update

Note: Mandatory instructions do not require Update on input scan cells.

Used on sensitive device inputs (e.g., System Clock)

Figure 18: Basic Boundary-Scan Cell (Input)

Figure 19 shows a more universal design for a boundary-scan cell: it is capable of all three operations of capture, shift, and update, and is suitable as a cell on the device inputs or outputs. This design has separate flip-flops for shift and hold functions. Data can be shifted through the boundary-scan shift path without interfering with the value in the hold section (which could be routed to the data-out port through the output multiplexer).

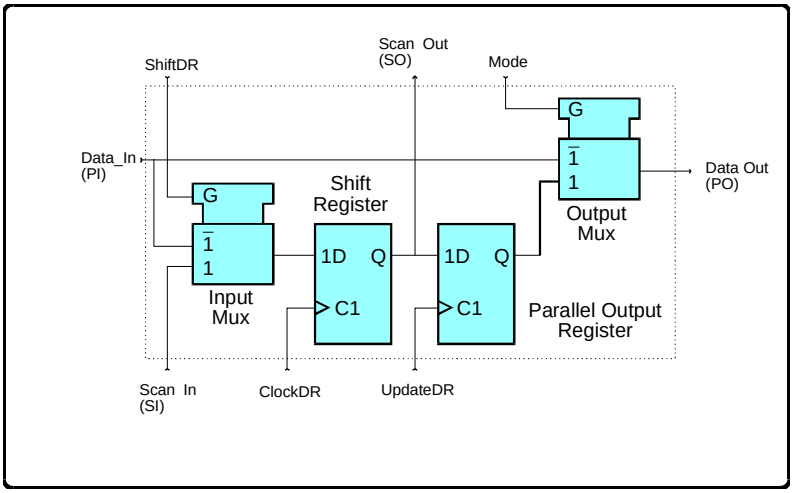


Figure 19: Basic Boundary-Scan Cell (Input/Output)

Figure 20 shows why a hold section might be required. Assume that the three outputs from the boundary-scan device are control signals to the Chip-Select (CS) controls of three RAM devices. In the normal course of events, only one RAM is selected to talk to the data bus. This means that most combinations of the three CS signals are illegal.

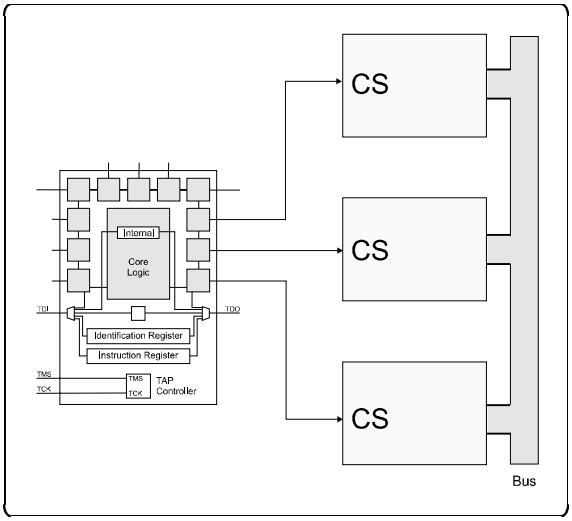


Figure 20: A Reason for the Hold State

It would be impossible to guard against illegal sequences if we were passing data along the boundary-scan path without the hold element and the output multiplexer was open to the shifting values. If the multiplexer was open to the values generated by the core logic, we may still have a problem if we are not exercising tight control over the status of the core logic. A simple solution is to include the hold section and to use the Clamp instruction to load safe values into the hold sections. Then, pass these values out through the output multiplexer.

Providing Boundary-Scan Cells

Primarily, boundary-scan cells must be provided on all device input and output signal pins, with the exception of Power and Ground. Note that there must be no circuitry between the pin and the boundary-scan cell with the exception of driver amplifiers or other forms of analog circuitry.

In the case of pin fan-in, boundary-scan cells should be provided on each primary input to the core logic. In this way, each input can be set up with an independent value. This provides the maximum flexibility for *Intest*.

Similarly, for the case of pin fan-out: if each output pin has a boundary-scan cell, then so *Extest* is able to set different and independent values.

Where there are tristate output pins, then there must be a boundary-scan cell on the status control signal into the output driver amplifier. Figure 21 shows a simple example of a tristate output pin.

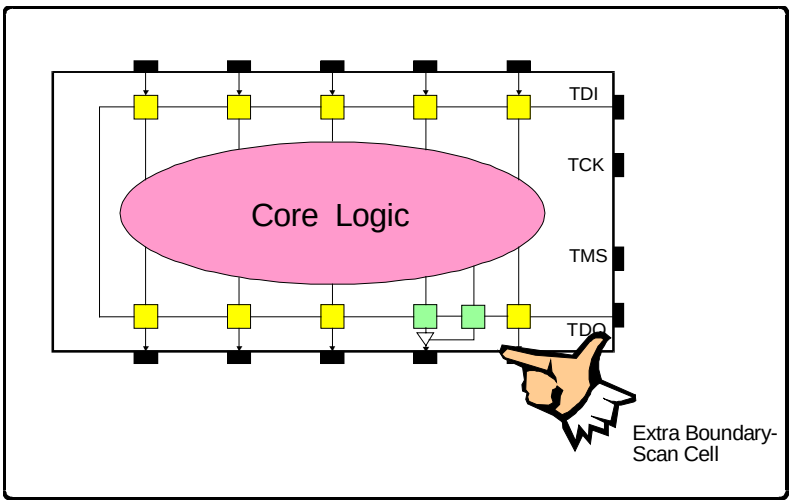


Figure 21: Control of Tristate Outputs

Figure 22 shows the set up for a bidirectional I/O pin. Here, we see that three boundary-scan cells are required: one on the input side, one on the output side, and one to allow control of the I/O status.

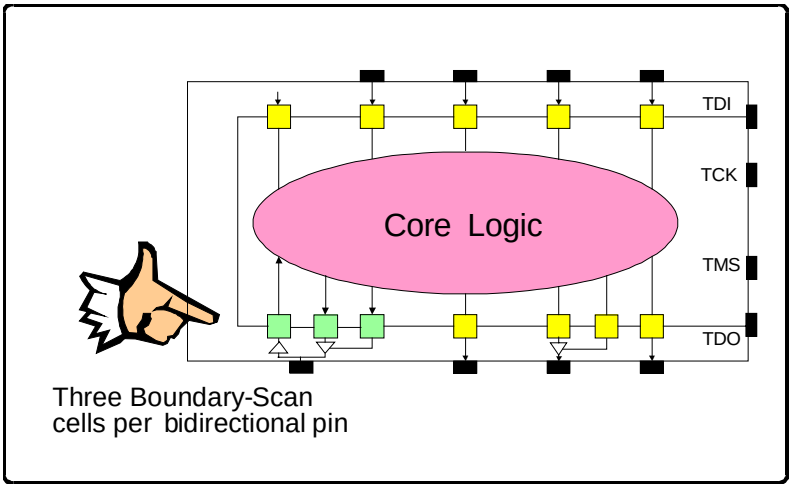


Figure 22: Bidirectional Input/Output Pins

Accessing Other Core-Logic Registers

The IEEE 1149.1 architecture does allow the definition and use of “private” instructions to access any suitable internal shift registers. An example could be an instruction *InScan* to allow access to an internal scan path register via the TDI-TDO route.

Another important optional instruction is *RunBist*. Because of the growing importance of self-test structures, the behavior of *RunBist* is defined in the standard. The self-test routine must be self-initializing (i.e., no external seed values are allowed), and the execution of *RunBist* essentially targets a self-test result register between TDI and TDO. Once the self-test routine is initiated, the TAP controller is held in its *Run-Test/Idle* state for the duration of the test. The self-test clock can either be TCK or some other suitable and available clock.

At the end of the self-test cycle, the targeted register holds the pass/fail result. It is important that this value is not changed by any subsequent pulses on TCK. In this way, parallel self-tests of different lengths on different devices on the same board can be carried out. When the final (i.e., the longest in run time) self-test is complete, all results can be clocked out along the register path made up of the linked individual result registers.

Chapter 4: Application at the Board Level

General Strategy

As a complement to this tutorial, we will look briefly at the three major stages of board-test strategy for a board populated by IEEE 1149.1-compliant devices (a “pure” boundary-scan board).

A general-purpose, three-step strategy for testing a pure boundary-scan board is:

Step 1. Carry out a boundary-scan infrastructure test by using either the blind interrogation technique described earlier (pages 29-30) or through the *Capture-IR/Shift-IR* operations to load and shift the built-in checkerboard values. Further optional infrastructure tests can be carried out if time permits.

Step 2. Use the *Extest* instruction to apply stimulus and capture responses across the interconnect structures between the devices on the board.

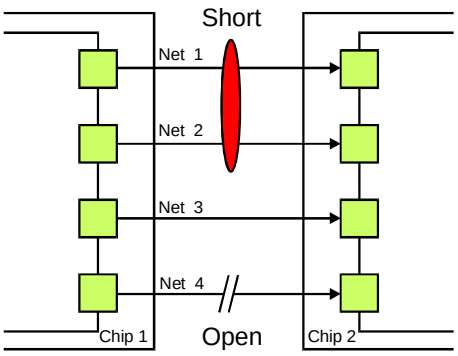
This is the major application of boundary-scan architecture and we will return to the basic algorithms later in this tutorial.

Step 3. Carry out either a limited “existence” test on the individual devices (using *Intest*) or initiate device self-test routines (using *RunBist*).

At the end of Step 3, we have “tested the tester” (Step 1); tested the regions most susceptible to assembly damage caused by electrical, mechanical, or thermal shock (Step 2); and tested that the right devices are in their correct positions on the board (Step 3).

Interconnect Test Example

Consider the simple four-net interconnect structure shown in Figure 23. Assume both devices are IEEE 1149.1 compliant and the left-hand device drives values into the right-hand device. Assume further that there is an unwanted short-circuit defect between Nets 1 and 2, and an unwanted open-circuit defect along Net 4. How can we test for such defects?



Problem: How to test for the open and short faults?

Figure 23: Interconnect Testing Example

Figure 24 shows a solution. The short circuit (assumed to behave logically like a wired-AND gate) is detected by applying unequal logic values (i.e., logic 1 on Net 1, logic 0 on Net 2) from Chip 1 to Chip 2. The wired-AND behavior causes Chip 2 to receive two logic 0s, allowing identification of the defect.

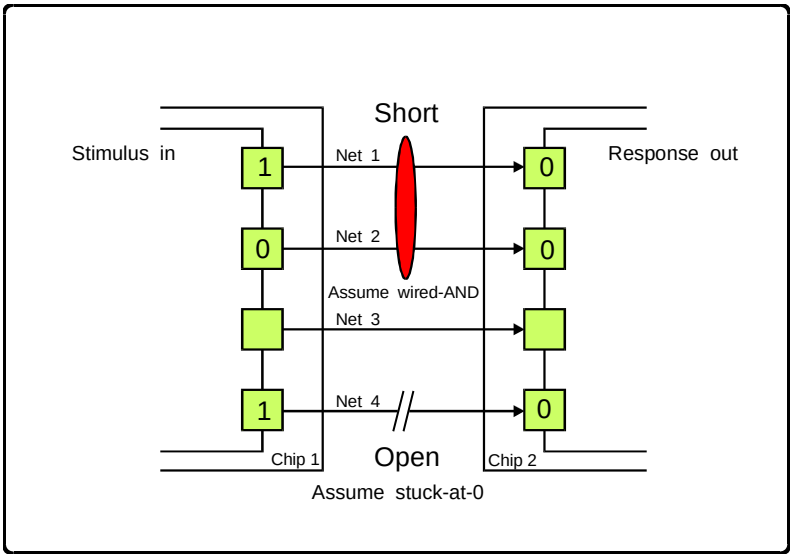


Figure 24: Interconnect Testing Solution

Similarly, if the open-circuit behaves like a stuck-at-0 fault, the defect is detected by applying a logic 1 from Chip 1 on Net 4 and observing that Chip 2 captured a logic 0.

A question arises — can we devise a general-purpose algorithm for creating a series of tests capable of detecting any 2-net short circuit (of either a wired-AND or a wired-OR nature) and any single-net open circuit (causing either a stuck-at-1 or a stuck-at-0 fault)?

This question was answered in 1974 in connection with a similar requirement for testing ribbon cables (Kautz, IEEE Trans. Computers, 1974, pp. 358-363). Consider Figure 25.

This diagram shows three consecutive tests applied to Nets 1 to 4. The first test is the vertical pattern 1110; the second is 0101; and the third is 1001. Think about the patterns “horizontally”; that is, the sequence 101 applied to Net 1, and so on. We can consider 101 to be a binary code assigned to Net 1. Similarly, the three tests define other horizontal codes for Nets 2, 3, and 4. Kautz showed that a sufficient condition to detect any pair of short-circuited nets

was that the "horizontal" codes must be unique for each net[1]. This means that the total number of bits in each code (the number of tests) is given by *ceil* [$\log_2$(N)], where N is the number of nets and *ceil* means ceiling (the upper integer value of the logarithm). This is illustrated in Figure 25.

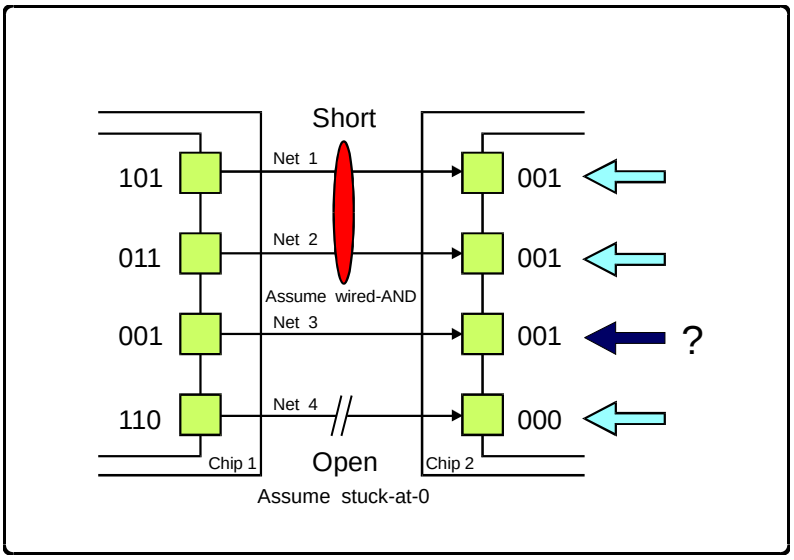


Figure 25: Detecting the Fault

In Figure 25, each horizontal stimulus code constructed from the three vertical tests is different. The response codes on nets 1 and 2 are incorrect because of the short circuit between these two nets.

At this point, we can ask, why use a three-bit code? With four nets, *ceil* [$\log_2$ (N)] is 2 and each net could be assigned a unique two-bit code. This is true, but the additional requirement to cover single stuck-at-1 and stuck-at-0 faults precludes the all-1 and all-0 codes. A stuck-at-1 fault would never be detected if the input code is all 1s: similarly for the stuck-at-0 fault and the all-0 code. In effect, the all-1s and all-0s become forbidden codes.

[1] If each net has a unique code, at some point any two nets have complementary stimulus values assigned. This is a necessary and sufficient condition to detect a short circuit of type wired-AND or wired-OR.

This means that the total number of bits in each code to satisfy the uniqueness property and to exclude the two forbidden codes is given by *ceil* [$\log_2$ (N+2)] where the "+2" represents the two "virtual" nets with the all-0 and all-1 code assignments. This results in a three-bit code for the four nets in Figure 25.

Now consider the effect of applying these codes to the four-net infrastructure. The response codes on Nets 1 and 2 are different to their respective input stimulus codes, but they are both the same code (001). From this information, we deduce:

1. there is a short-circuit fault between Nets 1 and 2
2. the short-circuit is a "wired-AND" type

Unfortunately, this diagnosis may not be fully correct. Net 3, which is not short-circuited, was tested by the code 001. This code is the same as the faulty response code, and although net 3 response is correct in terms of being the same as the stimulus code, it could be 001 because net 3 is also part of the short circuit problem (i.e., nets 1, 2 and 3 could all be shorted together).

This diagnostic ambiguity is an example of the aliasing syndrome of short-circuit faults. There are ways of overcoming this syndrome (and other syndromes), but the solutions are beyond the scope of this tutorial. One additional test to reduce the ambiguity is 0011 (see Figure 26). Basically, the fourth test splits the known short circuit pair (net 1, net 2) from the possible short-circuit candidate (net 3).

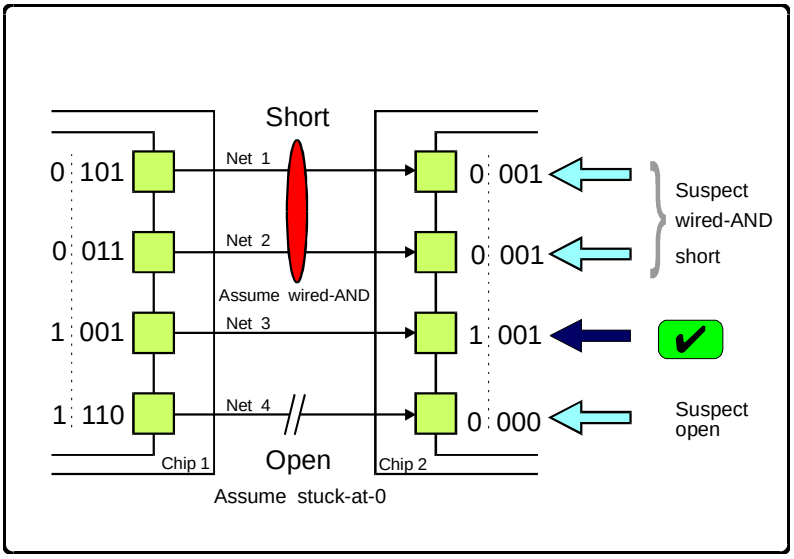


Figure 26: Locating the Fault

To conclude this example, notice that the s-a-0 open circuit on net 4 is detected and located cleanly by the all-0 response code. This code is one of the two forbidden codes and cannot be aliased to any other code associated with a defect-free interconnect.

Practical Aspects of Using Boundary-Scan Technology

Handling Non-Boundary-Scan Clusters

In reality, boards are populated with both boundary-scan (BScan) and non-boundary-scan (non-BScan) devices. The question arises, “what can we do to test the presence, orientation and bonding of the non-boundary-scan devices?” The answer to the question depends, in part, on the degree of controllability and observability afforded to the non-BScan devices through the boundary-scan registers of the BScan devices.

Figure 27 shows a “cluster” of three non-BScan devices surrounded by three BScan devices. The boundary-scan

registers in U1, U2, U3 can be used to drive test-pattern stimuli into the non-BScan cluster, and to observe the cluster responses but the difficulty will be to control and observe the truly buried nets inside the cluster (e.g., between U4 and U5).

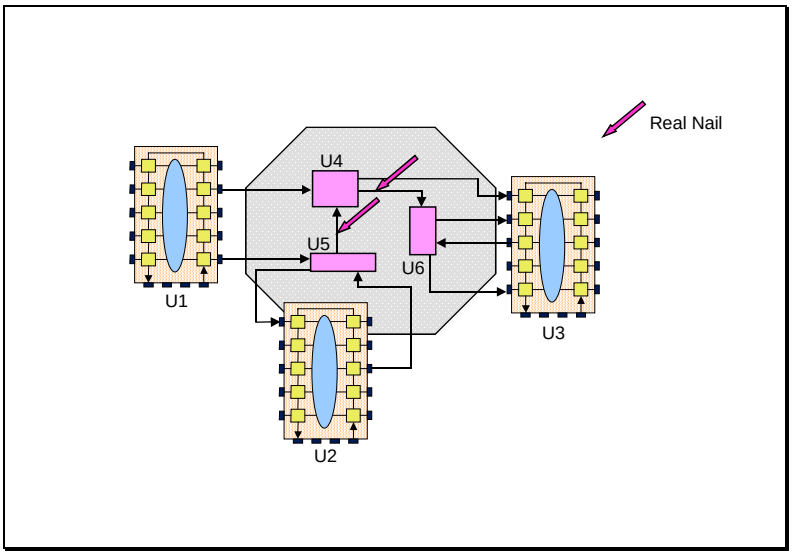


Figure 27. Handling Non-BScan Clusters

Given that we are not testing the full functionality of the non-BScan devices — only their presence, orientation and bonding — one solution is to develop a suitable set of tests for the non-BScan cluster that are applied from the boundary-scan driver cells and which drive signal values along the buried nets, targeted on opens and shorts. The responses are propagated out to the boundary-scan receiver cells.

For clusters of relatively simple non-BScan devices, generating these tests may not be too difficult. For clusters of complex non-BScan devices, generating the tests may become very difficult and there are no automatic pattern-generator tools to help the board test programmer.

Consequently, an alternative solution is to make use of real nails to access the buried nets, as shown in the diagram. Clearly, these nets have to be brought to the surface of the board (to allow physical probing) and the cost of test will increase (because of the extra cost of the bed-of-nails fixture), but this may be the only way to solve the problem. A solution that combines the virtual access of boundary scan and the real access of a bed-of-nails system is generally known as a *Limited Access* solution.

Access to RAM Arrays

Many boards contain arrays of Random Access Memory (RAM) devices (see Figure 28). RAMs are not usually equipped with boundary scan and so they too present manufacturing-defect testing challenges. In a way, an array of RAMs is a special case of a cluster of non-BScan devices.

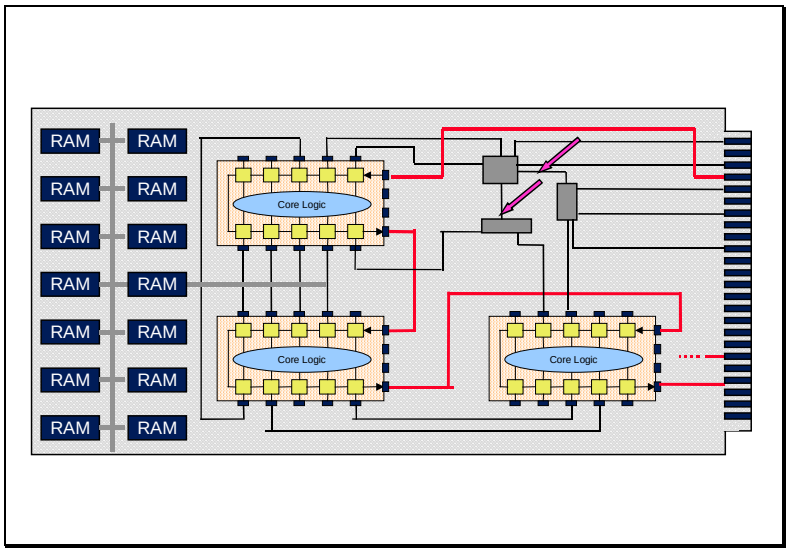


Figure 28. Testing a RAM Array Via Boundary Scan

Boards that contain RAMs typically also contain a programmable device, such as a microprocessor. The usual practice is to use the microprocessor to test the presence,

orientation and bonding of the RAM devices (i.e., the microprocessor becomes an on-board tester). This is OK as long as the microprocessor exists on the board. If there is no such device, then the RAMs can be tested for manufacturing defects through the boundary-scan registers of BScan devices as long as the BScan devices have boundary-scan register access to the control, data and address ports of the RAMs. Test times will be slow but the number of tests are not that great given that the purpose of the tests is to identify any opens or shorts on the RAM pins. Suitable tests can be derived from the classical walking-1/walking-0 patterns or from the *ceil* [$\log_2$ (N+2)] patterns described earlier.

Other Issues of BScan-to-Non-BScan Interfacing

Figure 29 illustrates some of the other issues of interfacing between BScan and non-BScan devices.

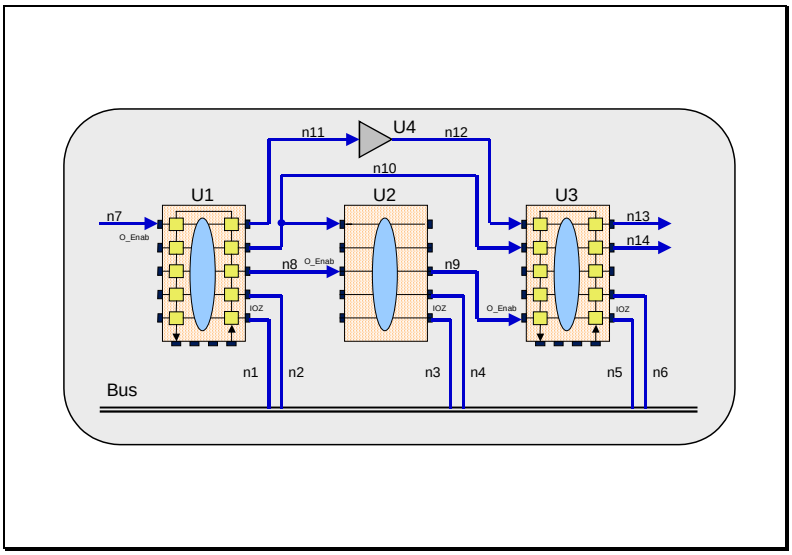


Figure 29. BScan-to-non-BScan Interface

Consider what happens when we try to set up interconnect tests between U1's bidirectional pins (marked IOZ on nets n1, n2) and U3's bidirectional pins (marked IOZ on nets n5,

n6) via the bus. First, we have to determine the exact nature of the boundary-scan cells on U1 and U3 IOZ pins. One set has to be set up as drivers and the other set as receivers. Assume we specify U1's pins to be the drivers and U3's pins to be the receivers. The interconnect test-pattern generator will compute tests from U1 to U3 based on the standard algorithm.

To set U1's pins into driver mode, we need to control net n7 (U1 ***O_Enab***) to the appropriate value. n7 is directly controllable so this will not be a problem. Now consider the ***O_Enab*** pin of U3. The value on this pin needs to be set to the appropriate level to make U3's bidirectional pins behave as receivers. The control for U3 ***O_Enab*** comes from the non-BScan device U2, along net n9. n9 is not directly controllable so we have a problem of trying to find out what to do on the input side of U2 to set U3's ***O_Enab*** to the correct value. If the inputs to U2 can be controlled by a BScan device (e.g., by the boundary-scan register of U1), then we can set fixed values in U1's output scan cells to hold U2 inputs to set U2's output values to the values required by U3's ***O_Enab*** input. The values held in U1's output scan cells are known as *constraints*, overriding any other values that might be generated by the interconnect test-pattern generator. Basically, the requirement for a constraint generates a mask that ensures that a particular output driver scan cell is always updated with the same constraint value.

Now return to the U1-to-U3 interconnect tests. The board-level netlist will identify U2 as another device with access to the bus. Before tests can be applied between U1 and U3, we first have to know the nature of the pins of U2 that are connected to the bus. Are they inputs only (I), outputs only (O), outputs with a high-Z state OZ), or full bidirectionals (IOZ)? Eventually, we might need to know the input-output nature of every pin on this non-BScan device. This data, sometimes called *characteristic* data, is easily created but absolutely necessary if we are to avoid potentially dangerous situations during interconnect test. For example, if U2's pins are IOZ and they are in their output-drive state,

then tests between U1 and U3 can cause damage to U2 through back-driving (bus contention). As a result, we need to set yet another *constraint* value into the boundary-scan cell in U1 that controls the value on U2's ***O_Enab*** pin, along net n8.

Now consider net n10. This net connects between the two BScan devices, U1 and U3, and so is a candidate for interconnect testing. Note however that the net also connects to the non-BScan device, U2. Again, we need to know the nature of the U2 pin: is it an input or an output? If it is an input, then there is no problem with driving between U1 and U3. If it is an OZ pin, then, again, we would need to set it into its high-Z safe state before applying the interconnect test on n10.

Finally, consider the connections n11 and n12 between U1 and U3 via U4. This appears to be a BScan-to-non-BScan-to-BScan series of connections and so is not amenable to interconnect testing between U1 and U3. But, we note that U4 has a very special logical property: it is transparent to digital signals. If we knew about this property, we could basically ignore its presence and treat n11 and n12 as a single connection between U1 and U3, thereby increasing defect coverage. In general, identifying transparent devices (e.g., series resistors, non-inverting line drivers) or devices with simple transparent modes (e.g., multiplexers), will enhance the defect coverage. In the case of a multiplexer, we need to control the control signals to select a particular input to pass through to the output. Constraint values can be used to achieve this.

The bottom line on all this is that most of the time spent in preparing a board-level test program is spent on the BScan-to-non-BScan interface: identifying and solving potential problems, as discussed above. The more boundary-scan devices there are on the board, as a percentage of all the digital devices on the board, the easier it becomes.

Assembling the Final Test Program

Figure 30 summarizes the major stages of assembling a final test program.

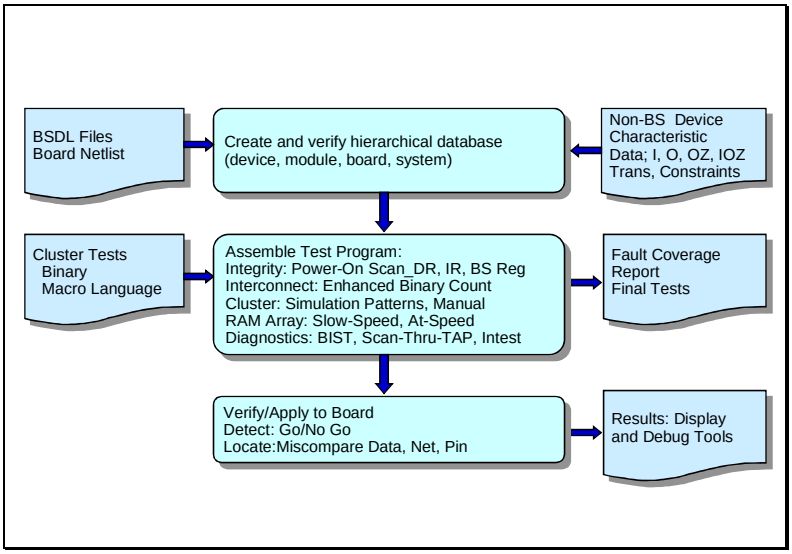


Figure 30. Assembling a Test Program: Tool Flow

First, the device BSDL files (see later) and board netlist data is used to compile a database. non-BScan characteristic data is also assembled ready to be used by the various pattern generators. The test program itself is composed of several segments:

- Board-level test infrastructure integrity test: device TDO-to-TDI interconnects, distribution of TMS, TCK and TRST*, if present. Typically, these tests use both a DR-Scan cycle and an IR-Scan cycle. The former is an application of the blind interrogation test whereas the latter uses the 01 captured into the Instruction Register, as described earlier.
- Full Enhanced Binary Count tests between all boundary-scan interconnects, setting non-BScan devices into safe

states and/or using non-BScan outputs to assert control over BScan devices where necessary.

- Tests to be applied to non-BScan clusters via a combination of BScan devices, real nails (if available), and the normal board edge-connector signals. These tests may be input in a simple one/zero format, or by using a higher-level test language, such as a *macro* language or C++.
- Tests to be applied to on-board RAM devices, either via an on-board microprocessor or via the boundary-scan registers of the BScan devices.

Diagnostics applied to production boards may then make use of internal design-for-test structures such as internal scan (often called *Scan-Thru-TAP*), Built-In Self Test or simply through the *InTest* Instruction, if available. The final test results are displayed to the user through an interface which allows line-by-line real-time debug, or by means of a graphical display of applied stimulus and captured test waveforms.

Tester Hardware

Modern low-cost board testers for boards populated with boundary-scan devices are based on a Personal Computer (see Figure 31). The drive/sense capability of the PC is enhanced through a controller card fitted either into an expansion slot (PC-AT, PCI or VXI) or into a PC Card slot, connecting to the board-under-test via a signal interface pod. TCK speeds are generally in the region of 10 MHz to 25 MHz, but can be higher. Additional driver/sensors are often available to provide direct control and observe on selected edge-connector positions (e.g., control a board Master Reset signal). The stimulus/response patterns themselves, along with the correct value-changes on TMS, are stored in RAM devices mounted on the controller card. These devices form a hardware buffer to hold applied stimulus values and collect actual response values for comparison with the expected values. Overall, the test-preparation and test-application software in the PC is controlled under Windows 98/2000 or Windows NT.

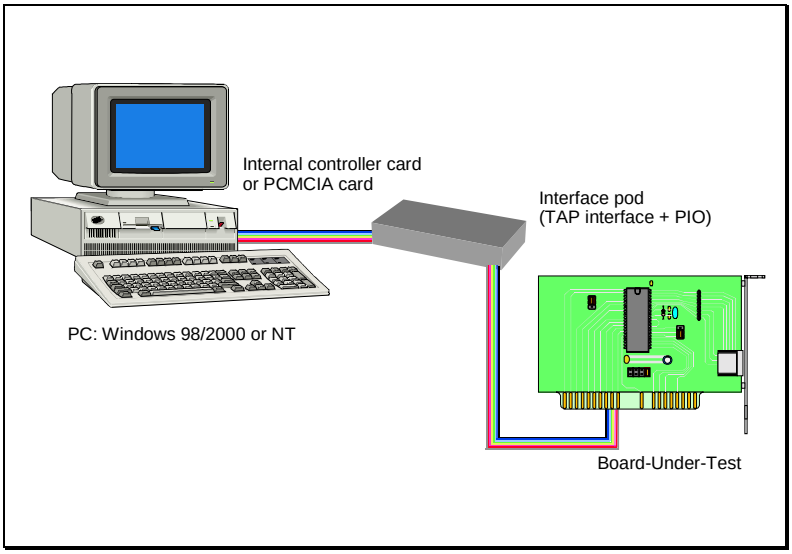


Figure 31. Tester Hardware

Such board testers are low-cost, compared to traditional in-circuit testers, and very portable, opening up the possibility to make use of the test program in other test requirements on the boards (e.g., in multi-board system integration and debug, and in field service).

Chapter 5: Related Standards

Several data formats have emerged to make IEEE 1149.1 successful and well-supported by tools. This chapter discusses the most widely accepted data formats that support IEEE 1149.1 — BSDL, HSDL, and SVF.

Boundary-Scan Description Language (BSDL)

This section discusses the most popular data format for describing how IEEE 1149.1 was implemented in a device — BSDL, or Boundary-Scan Description Language.

What Is BSDL?

Since 1990 when the IEEE 1149.1 standard was approved, implementation of the standard has accelerated. As more people became aware of and used the standard, the need for a standard method for describing IEEE 1149.1-compatible devices was recognized. The IEEE 1149.1 working group established a subcommittee to develop a device description language to address this need.

The subcommittee has since developed and approved an industry standard language called Boundary-Scan Description Language (BSDL). BSDL is a subset of VHDL (VHSIC Hardware Description Language) that describes how IEEE 1149.1 is implemented in a device and how it operates. BSDL captures the essential features of any IEEE 1149.1 implementation. BSDL was approved in 1994 as IEEE Std.1149.1b.

One of the major uses of BSDL is as an enabler for the development of tools to automate the testing process based on IEEE 1149.1. Tools developed to support the standard can control the TAP (Test Access Port) if they know how the boundary-scan architecture was implemented in the device. Tools can also control the I/O pins of the device. BSDL provides a standard machine and human-readable data format for describing how IEEE 1149.1 is implemented in a device.

How BSDL is Used

Many IEEE 1149.1 tools on the market support BSDL as a data input format. These tools offer different capabilities to persons implementing IEEE 1149.1 into their designs including board interconnect Automatic Test Pattern Generation (ATPG) and Automatic Test Equipment (ATE).

When you use tools that support BSDL, you can often obtain BSDL from your semiconductor vendor. This can result in significant time and cost savings.

Teradyne estimates that to create in-circuit test patterns for a leading microprocessor normally can require as much as seven weeks time:

- One week to study the device
- Four weeks to develop in-circuit test patterns
- Two weeks to verify the patterns on ATE

If the microprocessor supports IEEE 1149.1, and the BSDL is supplied by the vendor, the time to develop in-circuit test patterns is less than two hours using today's tools.

Elements of BSDL

A BSDL description for a device consists of the following elements:

- Entity descriptions
- Generic parameter
- Logical port description
- Use statements
- Pin mapping(s)
- Scan port identification
- Instruction Register description
- Register access description
- Boundary Register description

Entity Descriptions — The entity statement names the entity, such as the device name (e.g., SN74ABT8245). An entity description begins with an entity statement and terminates with an end statement.

```
entity XYZ is
  {statements to describe the entity go here}
 end XYZ
```

Generic Parameter — A generic parameter is a parameter that may come from outside the entity, or it may be defaulted, such as a package type (e.g., “DW”).

```
generic (PHYSICAL_PIN_MAP : string := “DW”);
```

Logical Port Description — The port description gives logical names to the I/O pins (system and TAP pins), and denotes their nature such as input, output, bidirectional, and so on.

```
port (OE:in bit;
    Y:out bit_vector(1 to 3);
    A:in bit_vector(1 to 3);
    GND, VCC, NC:linkage bit;
    TDO:out bit;
    TMS, TDI, TCK:in bit);
```

Use Statements — The use statement refers to external definitions found in packages and package bodies.

```
use STD_1149_1_1994.all;
```

Pin Mapping(s) — The pin mapping provides a mapping of logical signals onto the physical pins of a particular device package.

```
attribute PIN_MAP of XYZ : entity is
PHYSICAL_PIN_MAP;
constant DW:PIN_MAP_STRING:=
 “OE:1, Y:(2,3,4), A:(5,6,7), GND:8, VCC:9, “&
    “TDO:10, TDI:11, TMS:12, TCK:13, NC:14”;
```

Scan Port Identification — The scan port identification statements define the device's TAP.

```
attribute TAP_SCAN_IN of TDI : signal is TRUE;
attribute TAP_SCAN_OUT of TDO : signal is TRUE;
attribute TAP_SCAN_MODE of TMS : signal is TRUE;
attribute TAP_SCAN_CLOCK of TCK : signal is (50.0e6,
   BOTH);
```

Instruction Register Description — The Instruction Register description identifies the device-dependent characteristics of the Instruction Register.

```
attribute INSTRUCTION_LENGTH of XYZ : entity is 2;
attribute INSTRUCTION_OPCODE of XYZ : entity is
 “BYPASS (11), “&
 “EXTEST (00), “&
 “SAMPLE (10) “;
attribute INSTRUCTION_CAPTURE of XYZ : entity is
   “01”;
```

Register Access Description — The register access defines which register is placed between TDI and TDO for each instruction.

```
attribute REGISTER_ACCESS of XYZ : entity is
 “BOUNDARY (EXTEST, SAMPLE), “&
 “BYPASS (BYPASS) “;
```

Boundary Register Description — The Boundary Register description contains a list of boundary-scan cells, along with information regarding the cell type and associated control.

```
attribute BOUNDARY_LENGTH of XYZ : entity is 7;
attribute BOUNDARY_REGISTER of XYZ : entity is
 “0 (BC_1, Y(1), output3, X, 6, 0, Z), “&
 “1 (BC_1, Y(2), output3, X, 6, 0, Z), “&
 “2 (BC_1, Y(3), output3, X, 6, 0, Z), “&
 “3 (BC_1, A(1), input, X), “&
 “4 (BC_1, A(2), input, X), “&
 “5 (BC_1, A(3), input, X), “&
 “6 (BC_1, OE, input, X), “&
 “6 (BC_1, *, control, 0)”;
```

Hierarchical Scan Description Language (HSDL)

This section discusses a data format for describing how IEEE 1149.1 was implemented at the board or system level — HSDL, or Hierarchical Scan Description Language.

What Is HSDL?

Texas Instruments developed the Hierarchical Scan Description Language (HSDL) to complement BSDL, using the same subset of VHDL statements as BSDL. ASSET InterTech, Inc. is the contact point for maintaining the HSDL standard and is directly responsible for additions or changes to the standard.

HSDL picks up where BSDL stops to describe additional attributes of IEEE 1149.1 devices and how IEEE 1149.1 devices are connected at the board and system level.

HSDL uses the BSDL entity and package in new ways. Entities in HSDL are used to describe modules as well as devices. A module is any level of architecture above the device level, including boards, multichip modules, backplanes, subsystems, and systems. In addition, HSDL provides two new packages used to indicate that an entity is an HSDL device or module.

BSDL is well suited for describing how IEEE 1149.1 is implemented in a device, but stops there. HSDL provides a method for describing how IEEE 1149.1 devices are connected at the board, module, and system levels. HSDL serves three needs not addressed by BSDL.

- Description of the test bus interconnections of IEEE 1149.1 at the board or module level
- Description of boards with dynamic and reconfigurable architectures
- Ease-of-use and risk reduction improvement during interactive design debug and verification

In this way, BSDL and HSDL can be used together to obtain a full description of the unit under test (UUT). In addition, a basic device-level BSDL file can be augmented with appropriate HSDL statements to ease its use for interactive design debug of the UUT.

HSDL Module Statements

HSDL module statements use much of the same syntax as BSDL. New statements have been added to describe the members and scan paths of the module and to simplify interactive use.

- Entity descriptions
- Generic parameter
- Logical port description
- Use statements
- [Optional module descriptions]
- [Optional port description(s)]
- Pin mapping(s)
- Scan port identification
- [Optional member description(s)]
- [Optional bus description(s)]
- Path description
- [Optional member connections]
- [Optional constraint description(s)]
- [Optional design warning]

Entity Descriptions — The entity statement names the entity, such as the module name (e.g., BOARD). An entity description begins with an entity statement and terminates with an end statement.

```
entity BOARD is
 {statements to describe the entity go here}
end BOARD;
```

Generic Parameter — A generic parameter may come from outside the entity or it may be defaulted, such as a package type (e.g., “UNDEFINED”).

```
generic (PHYSICAL_PIN_MAP : string := (“UNDEFINED”)
```

Logical Port Description — The port description gives logical names to the I/O pins (system and TAP pins), and denotes their nature such as input, output, bidirectional, and so on.

```
port (TDI:in bit;
    TDO:out bit;
    TMS:in bit;
    TCK:in bit);
```

Use Statements — The use statement refers to external definitions found in packages and package bodies.

```
use STD_1149_1_1994.all;
use HSDL_module.all;
```

Pin Mapping(s) — The pin mapping provides a mapping of logical signals onto the physical pins of a particular entity.

```
attribute PIN_MAP of BOARD : entity is
PHYSICAL_PIN_MAP;
 constant PINOUT1 : PIN_MAP_STRING :=
 “TDI:1, TDO:2, TMS:3, TCK:4, GND:5”;
```

Scan Port Identification — The scan port identification statements define the entity's TAP.

```
attribute TAP_SCAN_IN of TDI : signal is TRUE;
attribute TAP_SCAN_OUT of TDO : signal is TRUE;
attribute TAP_SCAN_MODE of TMS : signal is TRUE;
attribute TAP_SCAN_CLOCK of TCK : signal is (5.0e6,
      LOW);
```

Members Description (Optional) — Members represent devices or other modules that are on the module. Usually members represent components, but some boards may contain scannable daughtercards, card slots, or other sub-assemblies that require modules to describe them.

```
attribute MEMBERS of BOARD : entity is
 “U1 (XYZ1, DW),”&
 “U2 (XYZ2, DW), “;
```

Bus Composition (Optional) — Buses in an HSDL module can be built of module buses, member module buses, member device buses, and member device test registers.

```
attribute BUS_COMPOSITION of BOARD : entity is
 “bus1[4] (U1.Boundary[3,0]), “&
 “bus2[4] (U2.Boundary[3,0]), “;
```

Path Description — Module paths are intended to describe the netlist of TAP signals (scan paths) on the board.

```
 constant boardpath1 : STATIC_PATH :=
  “U1, U2”;
 end BOARD;
```

For a complete specification of the HSDL language contact ASSET InterTech or your local ASSET representative.

Serial Vector Format (SVF)

What Is SVF?

Serial Vector Format, commonly referred to as SVF, was jointly developed by Texas Instruments and Teradyne in 1991. ASSET InterTech, Inc. is the contact point for maintaining the SVF standard and is directly responsible for additions or changes to the standard.

SVF is a standard ASCII format for expressing test patterns that represent the stimulus, expected response, and mask data for IEEE 1149.1-based tests. The need for SVF arose from the desire to have vendor-independent IEEE 1149.1 test patterns that are transportable across a wide selection of simulation software and test equipment — from design verification through field diagnostics.

Boundary-scan test execution is controlled by the sequencing of TAP signals on the pins of the devices. Each device's behavior is determined solely by the states of its TAP pins. Boundary-scan tools must maintain knowledge of the sequences required to exert certain behaviors within a

device and where that device is located down the serial scan path.

SVF controls the IEEE 1149.1 test bus using commands that transition the TAP from one steady state to another. Rather than describe the explicit state of the IEEE 1149.1 bus on every TCK cycle, SVF describes it in terms of transactions conducted between stable states. For instance, the process of scanning in an instruction is described merely in terms of the data involved and the desired stable state to enter after the scan has been completed.

The states such as Capture, Shift, and Update are inferred rather than explicitly represented. The data to be scanned in, expected data out, and compare mask are all grouped in an easily understandable manner. A command is provided to support deterministic navigation of TAP states where required.

In addition to supporting a higher-level depiction of scan operations, SVF also supports combined serial and parallel operations. This allows SVF to accommodate ATE environments where some stimulus/response is handled via parallel I/O, and serial signals are accessed via an IEEE 1149.1-control environment.

SVF also supports the concept of scan offsets. Offsets allow a test to be applied to a component or cluster of logic embedded in the middle of a scan path. For example, assume a device exists in multiple instances on a board. Serially applied tests were generated by the designer and are available in SVF format. To reuse this test, it is necessary to put all other devices on the scan path into bypass mode. The IEEE 1149.1 test controller must therefore comprehend the number of Instruction Register bits before and after the target device. Once in bypass, the devices introduce Data Register bits before and after the target device.

SVF allows a header and trailer to be defined once, which maintains the Instruction Register and Data Registers of the non-targeted devices in the desired bypass state. No

modifications are required to the SVF for the device. If the same test was targeted towards another device downstream in the scan path, this would be accommodated by changing the headers and trailers.

The offset approach is capable of installing any Instruction and Data Register stimulus, provided these values are constant for the entire process of applying the SVF device sequence.

SVF Structure

The SVF file is defined as an ASCII file that consists of a set of SVF statements. Statements are terminated by a semicolon (;) and may continue for more than one line. The maximum number of ASCII characters per line is 256. SVF is not case sensitive, and comments can be inserted into an SVF file after an exclamation point (!) or a pair of slashes (//).

Each statement consists of a command and parameters associated with that specific command. Commands can be grouped into three types: state commands, offset commands, and parallel commands.

State Commands

State commands are used to specify how the test sequences traverse the IEEE 1149.1 TAP state machine. The following state commands are supported:

- SDR — Scan Data Register
- SIR — Scan Instruction Register
- ENDDR — Define end state of DR scan
- ENDIR — Define end state of IR scan
- RUNTEST — Enter *Run-Test/Idle* state
- STATE — Go to specified stable state
- TRST — Drive TRST line to the designated level

SDR performs an IEEE 1149.1 Data Register scan. SIR performs an IEEE 1149.1 Instruction Register scan. ENDDR and ENDIR establish a default state for the bus following any Data Register scan or Instruction Register scan, respectively. RUNTEST goes to *Run-Test/Idle* state for a

specific number of TCKs. For each of the above commands, a default path through the state machine is used. Each of these commands also terminates in a stable, nonscannable state.

STATE places the bus in a designated IEEE 1149.1 stable state. TRST activates or deactivates the optional test reset signal of the IEEE 1149.1 bus.

Offset Commands

Offset commands allow a series of SVF commands to be targeted towards a contiguous series of points in the scan path. Examples would be a sequence for executing self-test on a device, or a cluster test where all devices involved in the cluster test are grouped together. The following offset commands are supported:

- HDR — Header data for data bits
- HIR — Header data for instruction bits
- TDR — Trailer data for data bits
- TIR — Trailer data for instruction bits

HDR specifies a particular pattern of data bits to be padded onto the front of every data scan. HIR specifies the same for the front of every Instruction Register scan. These patterns need only be specified once and are included on each scan unless changed by a subsequent HDR, HIR, TDR, or TIR command.

Parallel Commands

Parallel commands are used to map and apply the following commands:

- PIO — Specifies a parallel test pattern
- PIOMAP — Designates the mapping of bits in the PIO command to logical pin names

Parallel commands allow SVF to combine serial and parallel sequences. PIOMAP commands are used by parallel I/O controllers to map data bits in the command into parallel I/O channels using the ASCII logical pin name as a reference. The PIO command specifies the execution of a parallel

pattern application/sample. SVF does not specify any other properties of parallel I/O such as drive, levels, or skew.

Default State Transitions

SVF uses names for the TAP states that are similar to the IEEE 1149.1 TAP state names. Following is a list of SVF equivalent names for the TAP states.

IEEE 1149.1 TAP State Name [SVF TAP State Name]

- *Test-Logic-Reset* [RESET]
- *Run-Test/Idle* [IDLE]
- *Select-DR-Scan* [DRSELECT]
- *Capture-DR* [DRCAPTURE]
- *Shift-DR* [DRSHIFT]
- *Pause-DR* [DRPAUSE]
- *Exit1-DR* [DREXIT1]
- *Exit2-DR* [DREXIT2]
- *Update-DR* [DRUPDATE]
- *Select-IR-Scan* [IRSELECT]
- *Capture-IR* [IRCAPTURE]
- *Shift-IR* [IRSHIFT]
- *Pause-IR* [IRPAUSE]
- *Exit1-IR* [IREXIT1]
- *Exit2-IR* [IREXIT2]
- *Update-IR* [IRUPDATE]

The following list identifies sample default paths taken when transitioning from one state to a specified new state. For example, if the current state is RESET and you select DRPAUSE as the end state, the TAP moves from RESET through IDLE, DRSELECT, DRCAPTURE, DREXIT1 to DRPAUSE. You only have to specify the current and end states and not each intermediate step.

Stable State Path Examples

Current State	End State	State Path
RESET	RESET	RESET
RESET	IDLE	RESET
		IDLE
RESET	DRPAUSE	RESET
		IDLE
		DRSELECT
		DRCAPTURE
		DREXIT1
		DRPAUSE
RESET	IRPAUSE	RESET
		IDLE
		DRSELECT
		IRSELECT
		IRCAPTURE
		IREXIT1
		IRPAUSE

SVF Example

The following is an example SVF file:

```
!  Begin Test Program
!  Disable Test Reset line
   TRST OFF;
!  Initialize UUT
   STATE RESET;
!  End IR scans in DRPAUSE
   ENDIR DRPAUSE;
!  24 bit IR header
   HIR 24 TDI (FFFFFF);
!  3 bit DR header
   HDR 3 TDI (7) TDO (7) MASK (0);
!  16 bit IR trailer
   TIR 16 TDI (FFFF);
!  2 bit DR trailer
   TDR 2 TDI (3);
!  8 bit IR scan, load BIST seed
   SDR 16 TDI (ABCD);
!  RUNBIST for 95 TCK Clocks
   RUNTEST 95 TCK ENDSTATE IRPAUSE
!  16 bit DR scan, check BIST status
   SDR 16 TDI (0000) TDO (1234) MASK (FFFF);
!  Enter Test-Logic-Reset
   STATE RESET;
! End Test Program
```

The test begins by deasserting TRST. The DRPAUSE state is established as the default end state for instruction scans and data scans. Twenty-four bits of header and sixteen bits of trailer data are specified for Instruction Register scans. No status bits are checked. Three bits of header data and two bits of trailer data are specified for Data Register scans.

In this example, a single device in the middle of the scan is targeted. Notice from the 24-bit IR header (3x8-bit IR) and the 3-bit DR header (3x1-bit DR) that the targeted device has three devices before it in the scan path. From the 16-bit IR trailer (2x8-bit IR) and the 2-bit DR trailer (2x2-bit DR), the targeted device has two devices following it in the scan path. After the header and trailer offsets are established, all subsequent scans are the concatenation of the header, scan data, and trailer bits. The targeted device supports BIST, which is initialized by scanning a hex ABCD into the selected Data Register. The BIST in the targeted device is executed by entering the *Run-Test/Idle* state for 95-clock cycles. Next, the BIST result is scanned out and the status bits compared against a deterministic value to determine pass/fail.

Chapter 6: Boundary-Scan Tools

To complete this tutorial, we will turn our attention to what software tools are required in order to use boundary-scan technology for interconnect testing and other design debug and diagnostic operations on devices, boards, and systems.

Product Life Cycle Issues

Reaping the full value from your boundary-scan investment requires the use of a toolset that meets your testing and debug needs during the entire product life cycle. Because of the common and simple fixturing requirements of boundary-scan designs, you can now use a common toolset during all phases of the product life cycle. However, the toolset should also offer features to meet your specific needs for each phase of the product life cycle.

The toolset you choose should meet the needs for major phases of the product life cycle, including design debug, manufacturing test, and field test and repair. In addition, tools used during the manufacturing test process should also meet the needs for its four subprocesses: vector creation, test program creation, test program execution, and diagnosis. A discussion of the objectives of each process follows.

Design Debug

Design Debug is the process of taking an unknown product and ensuring that it is functioning properly. Often, the product in question is one of a limited number of products built in order to prove out the functional design of the system; these are called **prototype products** or **prototypes**. Even though the goal of this process is to determine if the prototype system functions as expected, the design engineer must first identify and repair any structural problems caused by incorrect physical construction of the product, e.g., solder globs that short two adjacent pins on a device. In this sense, the design engineer must first perform the manufacturing test process in order to complete the

debugging process. Completion of the structural and functional testing involves performing the test creation process.

Manufacturing Test

The goals of this process are to determine if any errors were made in the manufacturing process of the UUT and if the unit being tested functions as specified and verified during the design debug process.

Vector Creation

The focus of this subprocess is the creation and verification of the test vectors required to meet the test objectives for the current project. The tests that can be developed fall into two major categories: structural or functional. The goal of structural tests is identifying structural problems caused by incorrect physical construction of the product, e.g., solder globs that short two adjacent pins on a device. Functional tests attempt to verify that the product functions as expected under specified stimulus. In order to do functional tests, the product must usually be free from any structural defects. During the test vector verification stage of this subprocess, a known product should be used in order to detect any issues with the test vectors themselves.

Test Program Development

The goal of this subprocess is to provide an executable software program, including test vectors, to apply the appropriate boundary-scan tests and, in the case of a test failure, determine what action should be taken with respect to the failed product. This software program is called a **test program**. Once available, the test program is installed on the test machines on the manufacturing floor and executed by the *test operator* on products as they pass through the manufacturing line.

There is a wide range of capabilities that might be placed into a test program. At one end of the scale, the test program may simply be a batch file that sequentially executes the same test on each product without requiring

any interaction with the operator. In this case, the test program may only provide textual information to the operator on the results of the test application as a PASS/FAIL message with instructions to remove the bad product from the manufacturing line. At the other end of the scale, the test program may involve a sophisticated graphical user interface, which requires significant decision making on the test operator's part to complete the test and provide significant diagnostic information to the test operator as to what is wrong with the product being tested.

Another consideration in this subprocess is the need to have structural and functional tests to complete testing of the unit. A concern arises because sometimes the test operator must use multiple test tools each tuned for a particular test type.

Test Program Execution

This subprocess involves the actual execution of the appropriate test vectors on products as they move through the manufacturing line. This subprocess also involves determining what action to take when a product fails a specific test. This test execution and diagnosis is controlled by the test program. The person who executes this subprocess is called the **test operator**.

Since the goal of the manufacturing line is to keep products moving at a specified pace, full analysis or repair of failed products is not done at this time. Most often failed products are removed from the line, tagged as being defective, and attached with some type of information, that can be used to further diagnosis and repair of the product at a later time.

Diagnosis

This subprocess has two goals: 1) determine why a specific product failed a specific test and, 2) if possible, effect the necessary repairs to that product. During the normal manufacturing process failed units are diagnosed in order to effect sufficient repairs to allow the units to become part of manufacturing output.

In the preferred case, the diagnostic engineer first examines the test results from the test failure that occurred to determine that the product is faulty. After this examination, if the defect cannot be determined, the diagnostic engineer would like to rerun the same test to determine if the failure is repeatable in the current environment. If the defect is still not determinable, the diagnostic engineer will want to execute additional tests, either ones previously created or others created during diagnosis to try and debug the product.

In many ways, this process is similar to the design debug process, except that the diagnostic engineer knows that the board has at least one defect and can have some information to pinpoint where that defect is.

Different from the design debug process, this diagnostic engineer almost certainly does not have any depth of knowledge of the product at hand and does not have access to the type of computer-aided design information or other data available in earlier processes.

Field Test and Repair

The goal of this process is to, as quickly as possible, determine what product or part of a product in end-customer use is faulty and replace the faulty unit. In this way, this process is similar to the first part of the diagnosis process, except here the engineer may be dealing with the test and diagnosis of a much more complicated system involving many individual boards or subsystems.

As in the diagnosis process, the field test engineer will want to run the test program for a product to determine what is wrong. And, may want to run additional tests or interactive applications in order to further isolate the defective unit. Also, there is a desire for this testing to be done without any or minimal human intervention. In this case, the product's operating system automatically, or under human direction, runs the required tests and reports back appropriate diagnostic information.

A key point about this process is that it always occurs in an environment that is not directly under the control of the company that produced the product. This means that the people and tools used during this process must be flexible and must be available at the end-customer's site.

Boundary-Scan Tools Requirements

A well-developed implementation of boundary-scan architecture in combination with the right boundary-scan software tools can provide major benefits over more traditional methods such as logic analyzers, oscilloscopes, and in-circuit testers for many test and design debug tasks.

These benefits include:

- Easily handle complex system configurations which include daughtercards, multichip modules (MCMs), single inline memory modules (SIMMs), or other modules that are added to the main board
- Test systems which are configurable where the system composition changes based on end-customer demands
- Access and control device registers, buses, and pins
- Easily access Built-In Self-Test (BIST) capabilities present in devices in the system
- Integrate testing of non-scannable devices and memories
- Integrate a boundary-scan test suite with other test tools and test executives through industry-standard programming interfaces
- Better match price and performance by providing the toolset running on multiple platforms
- Reuse test suites at higher levels of integration and through different phases of the product life-cycle
- Embed tests into the system for on-line testing and diagnostics while in the field
- Complete the required manufacturing defect detection and diagnosis

In the following discussion, we will examine each process of the product life-cycle and what boundary-scan tools are required to gain the most from your boundary-scan investment.

Design Debug

With the inclusion of boundary-scan architecture into a design, the design team has a real opportunity (for the first time) to perform deterministic structural defect analysis on their prototype boards and systems as done in the manufacturing environment. To support this type of analysis, the boundary-scan tools must present the same kinds of tools as traditionally found in the manufacturing test environment. A more complete description of these capabilities is included in the Manufacturing Test section, but in general these are the capabilities required are:

- Vector creation tools for:
  - scan path and interconnect testing
  - non-scan clusters of logic surrounded by boundary-scan devices
  - memory testing
  - conversion of chip-level parallel tests for application in a serial environment
  - easily creating other custom tests for the UUT

- Diagnostics capabilities for interconnect testing with resolution to at least the net-level and preferably to the pin-level and other diagnostics for analyzing results from serial vector application

In addition to assisting with manufacturing defect analysis of prototypes, boundary-scan tools can provide the design engineer with many capabilities to assist functional debug of the prototype design. Boundary-scan based interactive design tools allow the design engineer to access and control boundary-scan device registers and pins as an adjunct to other functional tester access. With this ability, the designer

can ensure that correct values are driven to critical components, drive specific values onto a device, or gain access to internal device registers that might provide clues to functional errors. These design debug tools fall into two general capabilities: scan analysis and debugging.

Scan analysis tools allow you to apply test vectors to the unit under test, capture responses, and view those responses in state table or digital waveform displays. These tools also support concepts common to logic analyzers such as triggering and sequences that allow you to control when and how much response data to collect for analysis. With these tools, you can view a large number of vectors and analyze the hardware's response. Comparisons can be made between expected and actual values automatically speeding debug time.

Debugging tools provide an interactive interface for control and observation of the IEEE 1149.1 architecture. Features includes:

- Graphical view of the design hierarchy
- Ability to edit scan data at the register and pin level
- Data manipulation via user-defined symbolics or via binary, decimal, or hexadecimal data input
- Register grouping based on a design's functionality
- One-button interface to apply changes made to the instructions and data values
- Single-step application of pre-existing tests or serial vectors
- Interactive recording to create a test from an interactive sequence of debug steps

Manufacturing Test

Vector Creation

Vector creation tools provide a means to create and verify five basic types of tests: scan path integrity, interconnect,

cluster, memory, and custom. A brief discussion of each of these types of vector creation follows.

Scan path integrity tests involve verifying that the four-wire connection for the boundary-scan test bus does not have faults on it. The tool should provide an automated means of creating the vector sequence required to verify this. Diagnostics will pinpoint the device and signal which is faulty.

Interconnect tests are the same as in the traditional manufacturing test environment and provide the ability to detect and isolate common stuck-at and open/shorts on device interconnect. The tool should provide an automated means of creating these vectors taking as input your device-level boundary-scan descriptions, a description of how the boundary-scan devices are arranged in the scan chain, and CAE netlist information in common formats for the non-test device interconnections. The tools also should provide a means to easily ignore series devices in the design as a means of improving diagnostics later and provide an easy means of setting control values on certain pins that may not be changed during the test, e.g., a program pin on an FPGA or PLD. Textual outputs of fault coverage and vector responses are also required.

Cluster tests are generated to test either a single device or cluster of non-boundary-scan devices surrounded by boundary-scan devices. This capability allows you to extend vectors created in the CAE environment into your boundary-scan testing. The cluster test tool provides a means of translating those parallel CAE vectors into a serial format for easy application in your boundary-scan system. This tool should provide a means of automatically setting the values for boundary-scan control cells to control the operation of bidirectional and tristate pins during vector application and response acquisition.

Memory test creation involves the automatic generation of the vectors required to test address, data, and control lines for memory devices adjacent to boundary-scan devices.

This tool should use the boundary-scan description of the system, and specific information on the type and size of the memory device to create vectors for application.

Custom tests involve those tests you might want to create which are particular to your design. For this type of test creation the vector creation tool needs to provide an easy-to-use and simple programming language that allows full access and control to the registers and pins of boundary-scan devices. Using this programming language you should be able to easily tailor vectors to verify an array of static functional or structural problems with your design, including execution of BIST capabilities of a device.

Test Program Creation

Once the vectors required for the scan-based manufacturing test have been created, they must be assembled into a test program for delivery to the manufacturing floor.

Test program creation takes into account all of the varying needs you have in the manufacturing environment to provide you the functions necessary to develop custom test suites, integrate test suites with other test tools and executives, or build an entire manufacturing test capability. The environment should include a simple means of creating a test program based on industry standard test executives and provide industry-standard programming environments such as C and C++ for more complex test program creation.

The programming environment should provide access to the boundary-scan-accessible registers and pins through natural programming methods for custom-test suite development. It also should allow reuse of previously created test programs and vectors to speed the test development process.

Finally, with a high-level language basis, the programming environment enables you to:

- Reuse diagnostic reporting or other error routines that have been developed previously
- Share data with other test instruments
- Integrate boundary-scan-based tests into commercial test executives
- Quickly produce a customized user-interface for your test program based on Windows® technology

Test Program Execution

Once test programs are complete, you will need an effective means of deploying your boundary-scan based tests in a manufacturing environment. For this task, boundary-scan solutions should include PC- and VXI- based test application systems and the ability to integrate boundary-scan controlled parallel I/O modules. These solutions provide help in creating a manufacturing test environment to fully use boundary-scan testing.

Diagnosis

When failures are discovered in the manufacturing line, the boundary-scan tools provide several levels of diagnostics. These include text-based analysis of serial vectors results, net-level diagnostics for interconnect and cluster tests, and pin-level diagnostics for interconnect tests.

The net-level diagnostics must provide isolation down to the failing net, but may not detect the actual pin with the fault. This is often sufficient for many faults and provides sufficient data for fixing or proceeding with other tests.

Pin-level diagnostics must provide detailed fault diagnostics for various stuck-at conditions, bridging faults, open and bad bidirectional cells, and other opens and shorts. With this detailed information, you can find and fix the faulty component.

Field Test and Repair

For field test and repair, boundary-scan tools allow the extension of debug and diagnostics capabilities through employment of portable computing solutions such as access through a parallel printer port or PCMCIA card. In addition, tests can be embedded into the unit under test for self-test purposes. This involves the inclusion in the design of a test bus controller device and use of controller-specific “C” code to direct the application of vectors, acquisition of responses, and diagnostics. Although diagnostics are often limited to go/no-go, this provides a powerful alternative to lower the cost of testing by eliminating the expense of on-site visits for determining which unit must be replaced or repaired.

Chapter 7: Conclusion

Widespread adoption of IEEE 1149.1 Standard for boundary-scan architecture reflects an industry-wide need to simplify the complex problem of testing boards and systems for a range of manufacturing defects and performing other design debug tasks. This standard provides a unique opportunity to simplify the design debug and test processes by enabling a simple and standard means of automatically creating and applying tests at the device, board, and system levels. Several companies have responded with boundary-scan-based software tools that take advantage of the access and control provided by boundary-scan architecture to ease the testing process.

In this tutorial, we have discussed the motivation for the standard, the architecture of an IEEE 1149.1-compliant device, and presented a simple introduction to the use of the IEEE 1149.1 features at the board level — both to detect and locate manufacturing defects. We have reviewed applicable data standards and discussed the issues associated with choosing boundary-scan tools. For further details on boundary-scan — at the device level, board level, or system level — see the references listed in the Bibliography.

Bibliography

1. K. Parker, “The Boundary-Scan Handbook: Analog and Digital,” Kluwer Academic Press, 1998, Second Edition (Very good chapters on BSDL and on DFT guidelines)

2. H. Bleeker et al., “Boundary-Scan Test: A Practical Approach,” Kluwer Academic Publishers, 1993 (The “Philips” approach, including a chapter targeted on managers)

Reference

1. IEEE 1149.1 Working Group, http://grouper.ieee.org/groups.

2. BSDL/IEEE 1149.1 verification service, maintained by Agilent Technologies. See http://www.agilent.com/see/bsdl_service

3. Latest issues of: IEEE ITC Proceedings; Journal of Electrical Test: Theory and Application (Kluwer Academic Press); IEEE Design & Test of Computers