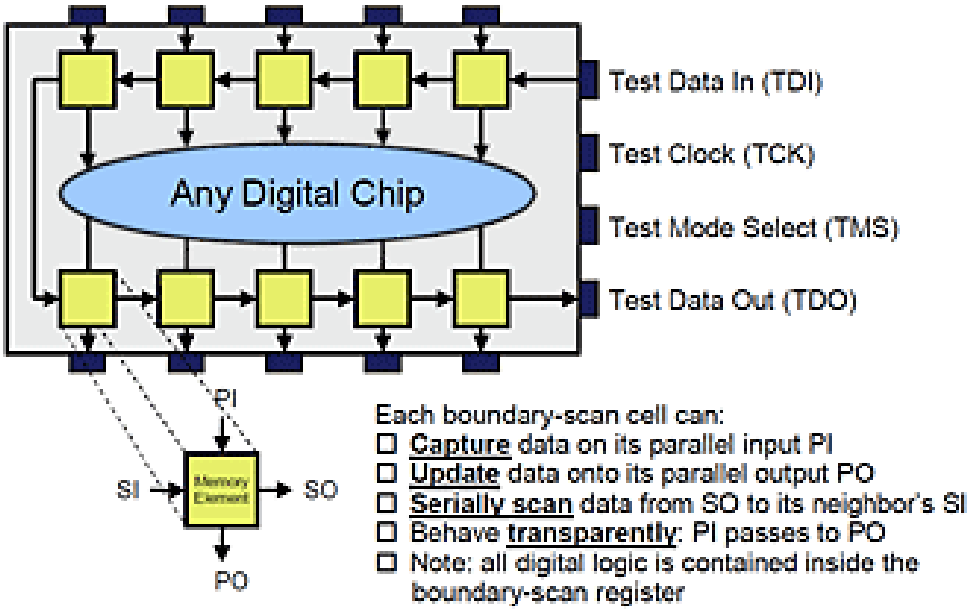
IEEE 1149.1 JTAG
and
Boundary Scan
Tutorial



Second Edition

by Dr. Ben Bennetts,
Adam Ley
and Ben Bales

ASSET

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 problems it solves, and its 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 2001 version of the Standard:

> IEEE Standard 1149.1-2001 "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 initially created in 1990 and revised in 1993, 1994 and 2001. You can obtain a copy of the standard via: http://standards.ieee.org/

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 landing sites or pads laid into the copper interconnect, or other convenient contact points. Testing then proceeds in two phases: power-off tests followed by power-on tests. Power-off tests check the integrity of the physical contact between a nail and the on-board access point. Open and shorts tests are then carried out based on impedance measurements.

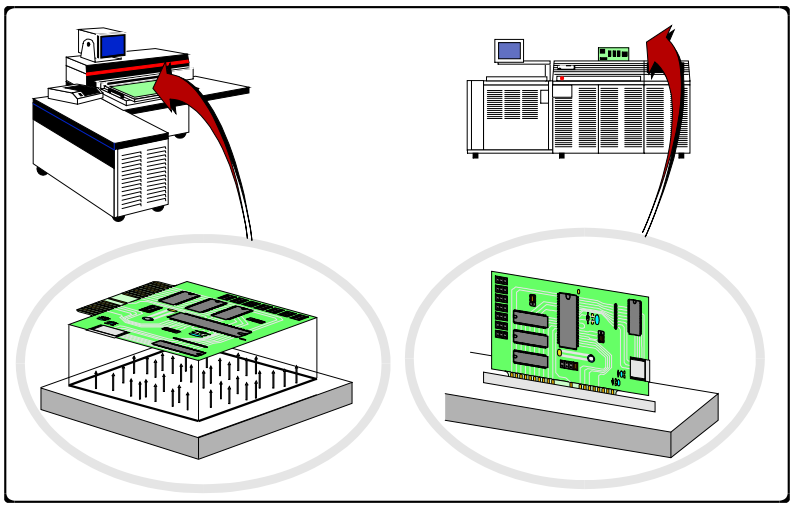


Figure 1: ICT versus Functional Test

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 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 landing sites or “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. This group of people called itself 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 its 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 a device's 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.

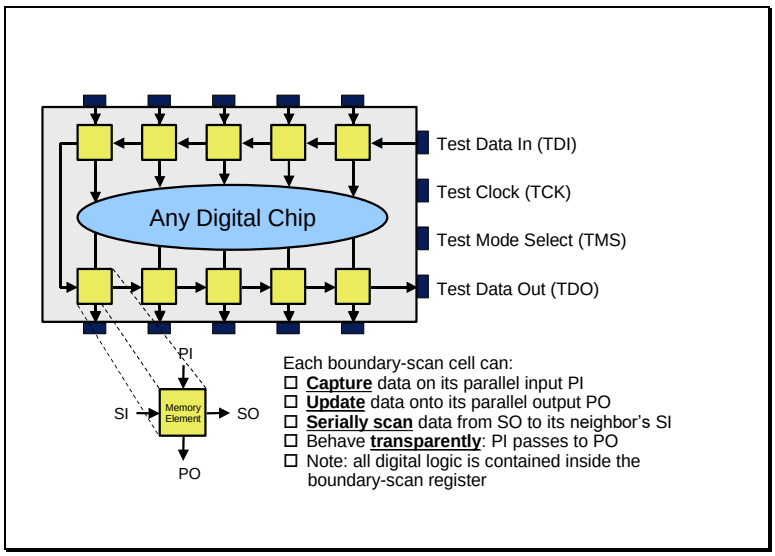


Figure 2: Principle of Boundary-Scan Architecture

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. Depending on the nature of the input scan cells, 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. As shown in Figure 3, the value of the scan path is at the board level.

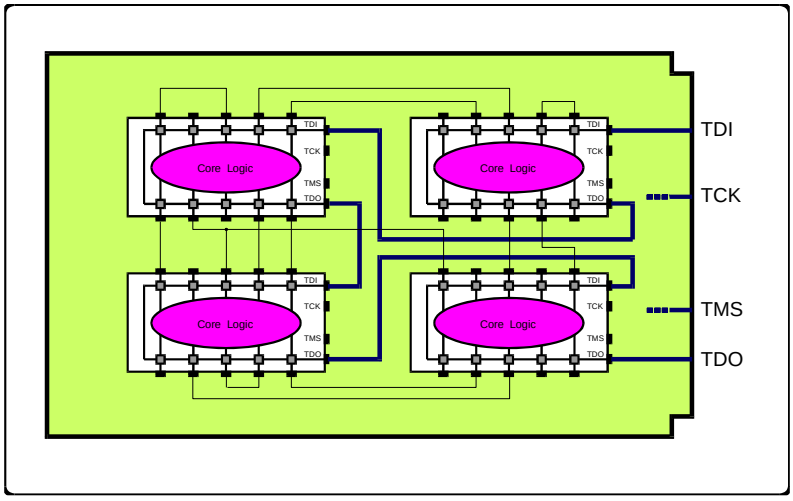


Figure 3: Using the Boundary-Scan Path

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's 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.)

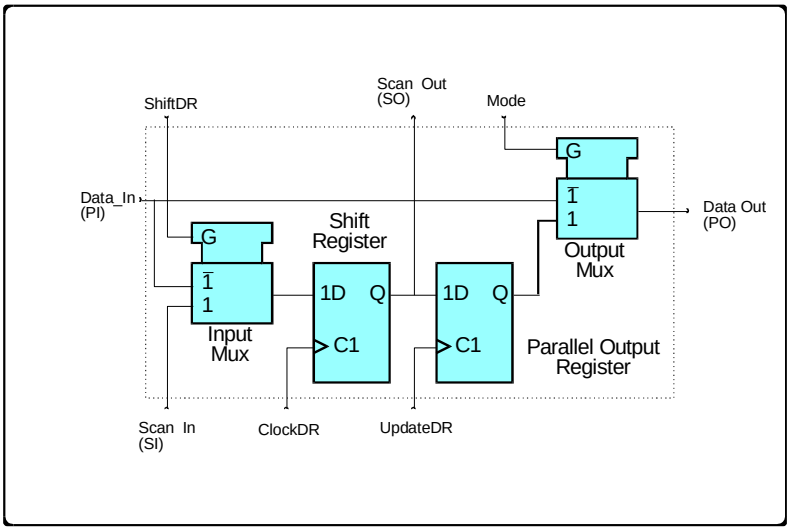


Figure 4: Basic Boundary-Scan Cell

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 with functional modes of operation. 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 on a circuit board 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 reasons 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.

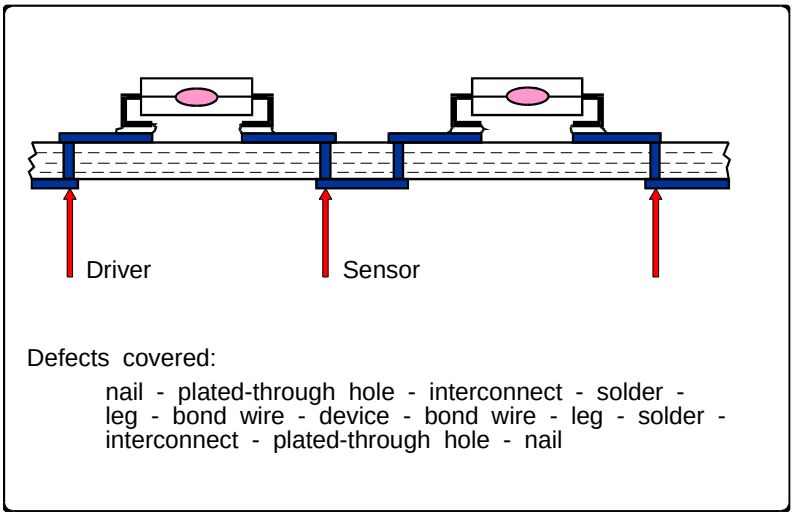


Figure 5: Bed-of-Nails Fault Coverage

In-circuit testers were not intended to prove the overall functionality of the devices on-board. It was assumed that devices had already been tested for functionality 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 the increasing use of custom designs such as ASICs and CPLDs.

Given that the 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 some sort of 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).

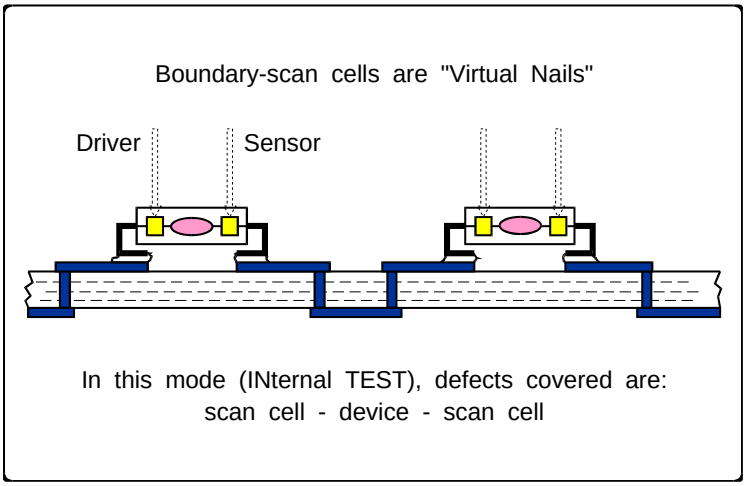


Figure 6: Boundary-Scan Fault Coverage (*Intest*)

And, more importantly, boundary scan cells are located at the beginning and end of interconnect paths because this is the region most likely to be damaged during board assembly (see Figure 7).

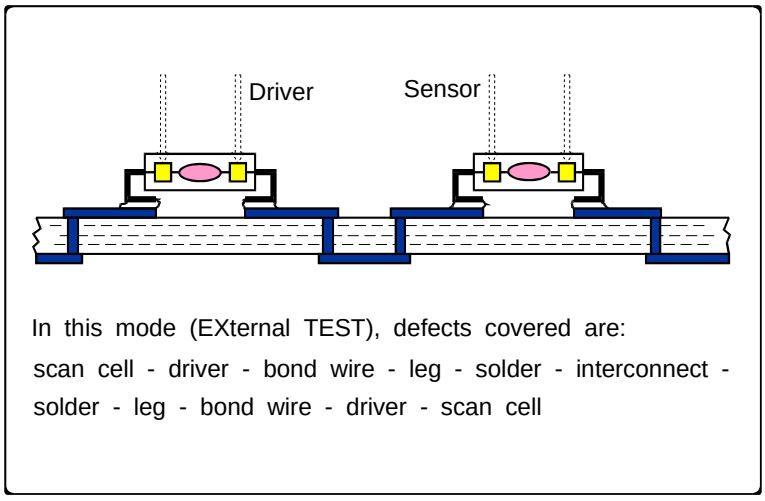


Figure 7: Boundary-Scan Fault Coverage (*Extest*)

Using boundary-scan cells to test a device's core functionality is called "internal test" or simply *Intest*. Using the boundary-scan cells to test the interconnect structure between two devices is called "external test" or simply *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?") or to identify defects such as devices missing, incorrectly oriented, or misalignment.

Chapter 3: IEEE 1149.1 Device Architecture

After nearly five years of discussion, the JTAG organization finally proposed the architecture shown in Figure 8.

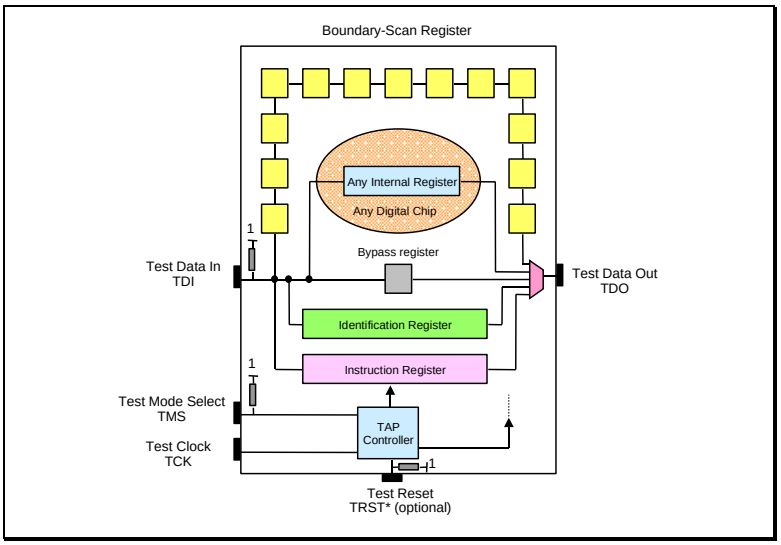


Figure 8: IEEE 1149.1 Chip Architecture

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 the device primary input and primary output pins of a device, 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.

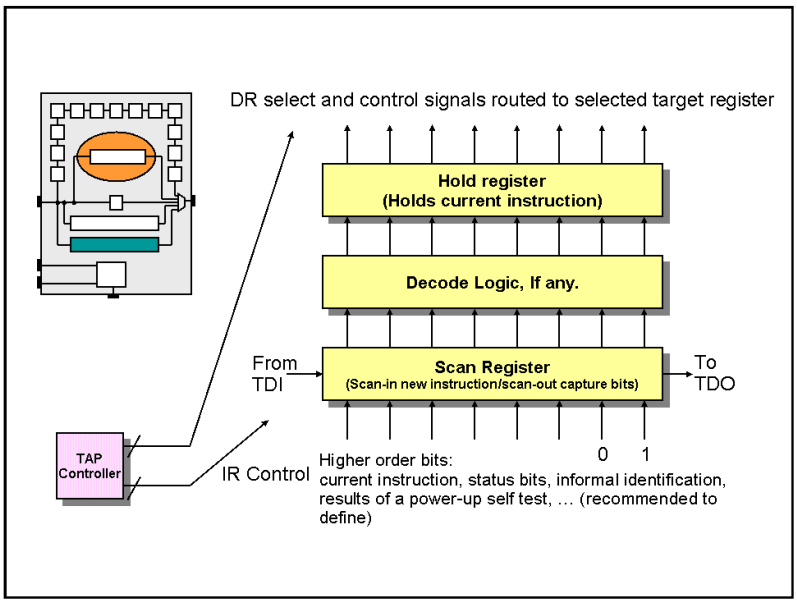


Figure 9: The Instruction Register

There may be some decoding logic between the two sections depending on the width of the register and the 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 four 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 bit pattern 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 1149.1-2001 version of the Standard describes four mandatory instructions: *Bypass, Sample, Preload,* and *Extest.*

The *Bypass* instruction must be assigned an all-1s code. When executed, the Bypass instruction 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 code unless the optional Identification Register (Ident) has been implemented, in which case, the *Idcode* instruction code should be present in the hold section.

The *Sample and Preload* instructions both select the boundary-scan register when executed. The *Sample* instruction sets up the boundary-scan cells to sample (capture) values moving into the device. The *Preload* instruction is used to preload known values into the output boundary-scan cells prior to some follow-on operation. The codes for the *Sample* and *Preload* instructions are not defined.

The *Extest* instruction selects the boundary-scan register preparatory to interconnect testing. Prior to the 2001 version of the Standard, the code for *Extest* was defined to be the all-0s code. Since 2001, the code is no longer defined.

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 that selects the Identification Register between TDI and TDO preparatory to loading the internal hard-wired Idcode values and reading them 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.

The *Runbist* instruction initiates an internal self-test routine and places the pass/fail result register between TDI and TDO.

Two new instructions were introduced in the 1993 revision, 1149.1a. These were *Clamp* and *Highz*. *Clamp* is an instruction that drives preset scan-cell values onto the outputs of devices (established initially with the Preload instruction) and then selects the Bypass register between TDI and TDO. 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 the device is designed such that all outputs can be placed in either a high-Z mode (3-state outputs) or input receive mode (for bidirectional scan cells). *HighZ* establishes these values on the output pins but leaves the Bypass register as the selected register Note that the Enable control signal to do this is supplied directly from the IR upon execution of the *HighZ* instruction. Preload is not used.

With the exception of Bypass, the codes for all instructions are undefined. Given the need for four 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.

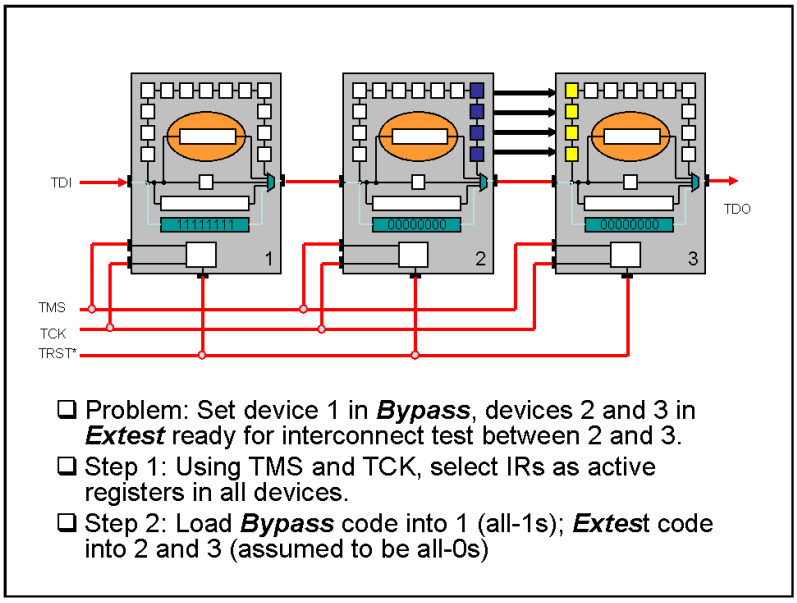


Figure 10: Using the Instruction Register — Step 1

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 devices farther down the scan chain) 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 (assumed to be all-0s for Chips 2 and 3) into the IRs of Chips 2 and 3.

Step 1 connects 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 loads 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 into their shift sections.

Step 3, shown in Figure 11, places further 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 and ready for the Extest operation.

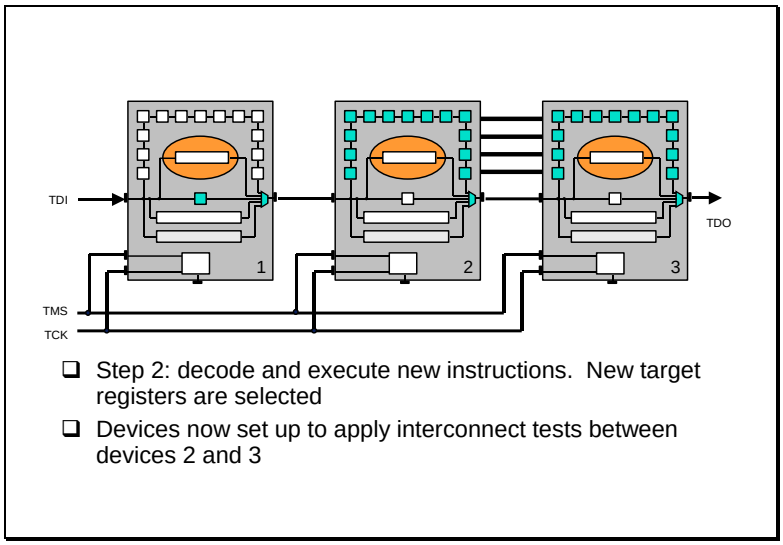


Figure 11: Using the Instruction Register — Step 3

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, the following happens:

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 shift register. 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 while clocking in he instruction code sequence 110000.

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 to determine 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 performed. 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 boundary-scan register to check the integrity of the boundary-scan cells. The question that is raised is why should all these additional integrity tests be done? 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 could be raised 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.

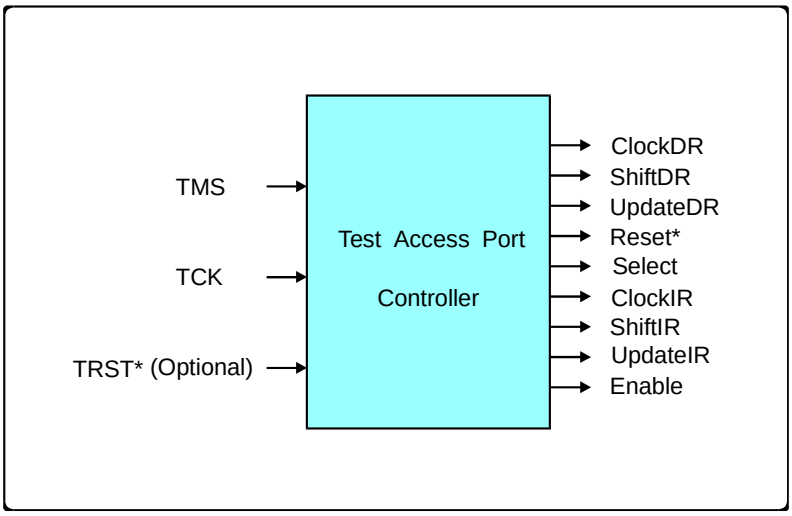


Figure 12: TAP Controller Global View

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 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.

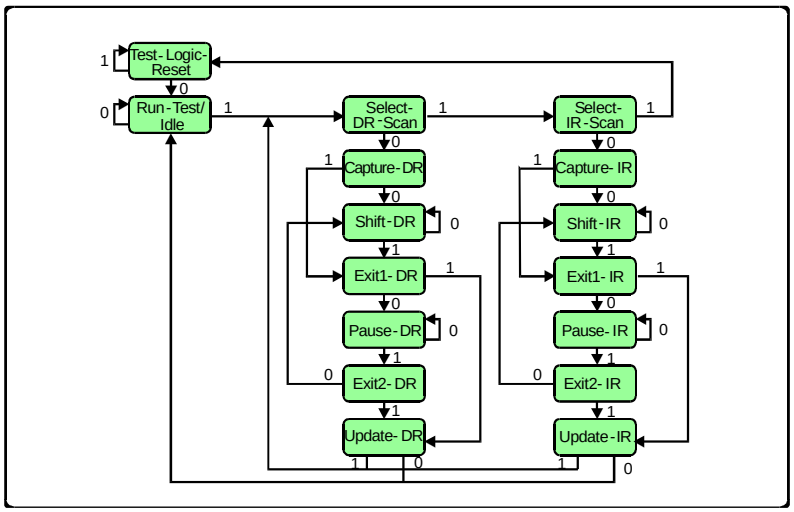


Figure 13: TAP Controller State Table Diagram

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 be 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 the *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 it 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 benefit of this.

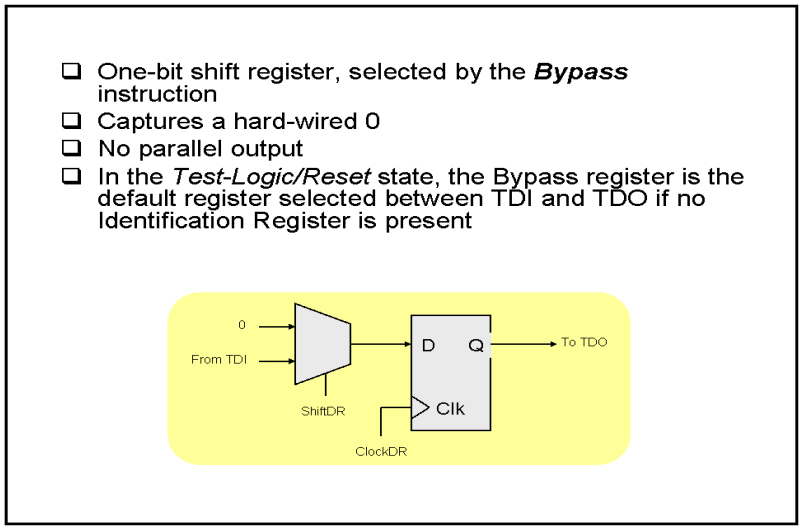


Figure 14: The Bypass Register

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.

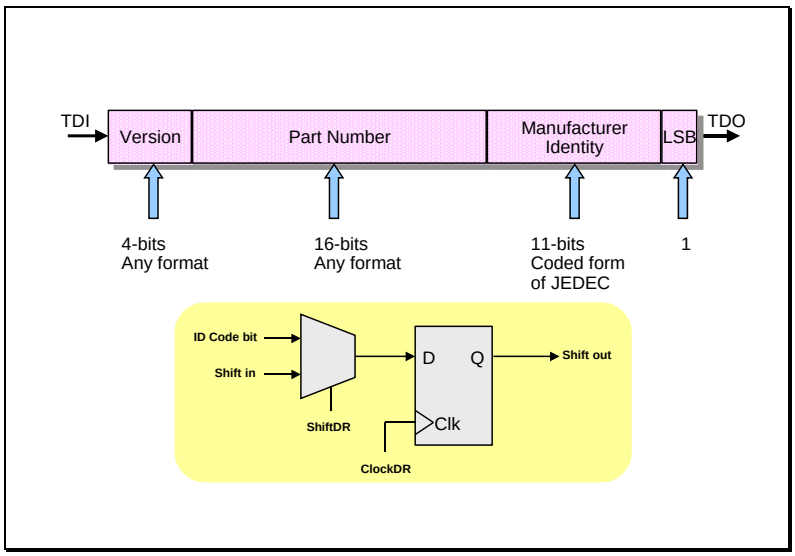


Figure 15: Device Identification Code Structure

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 service scenario. A customer's computer system has broken down. A hardware fault on a particular board is suspected as the cause. 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 locations 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.

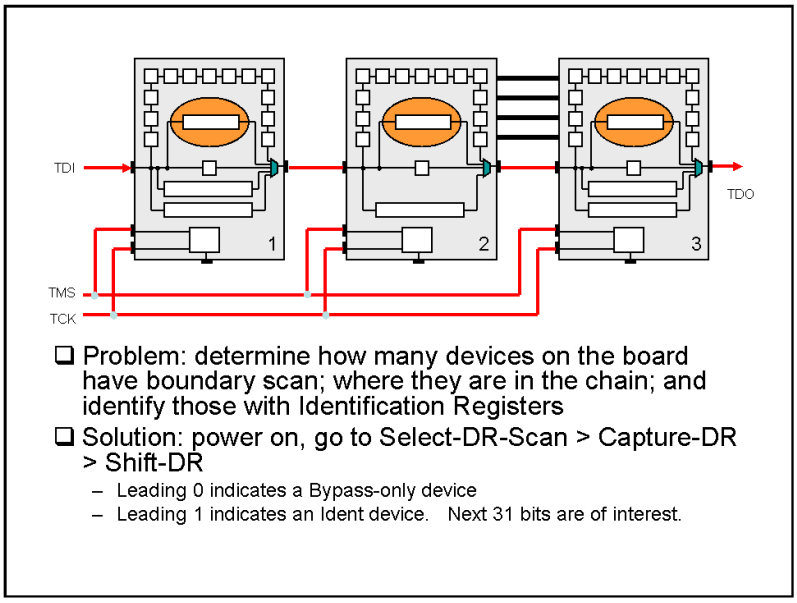


Figure 16: Use of the lsb = 1 Feature — Step 1

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.

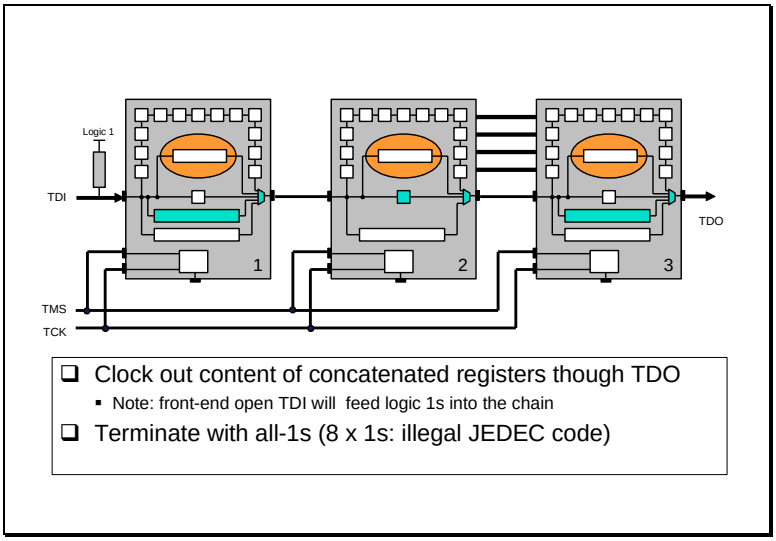


Figure 17: Use of the lsb = 1 Feature — Step 3

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 another 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: none of the four mandatory instructions require an update operation on the input scan cells.)

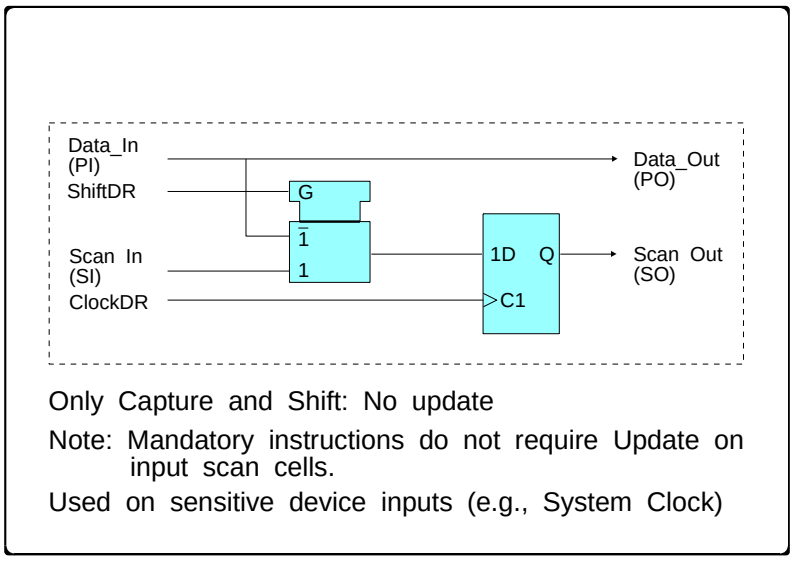


Figure 18: Basic Boundary-Scan Cell (Input)

Figure 19 shows a more universal design for a boundary-scan cell. This design is capable of all three operations, capture, shift, and update, and it can serve as either an input or an output cell on a device signal pin. 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).

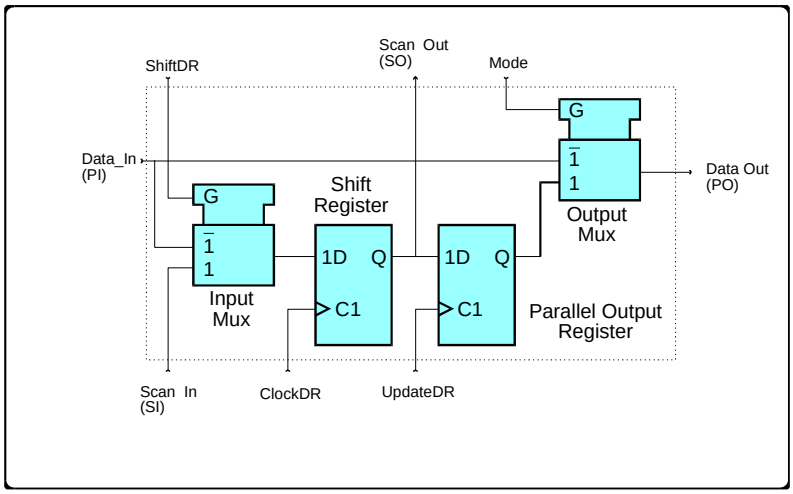


Figure 19: Basic Boundary-Scan Cell (Input/Output)

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.

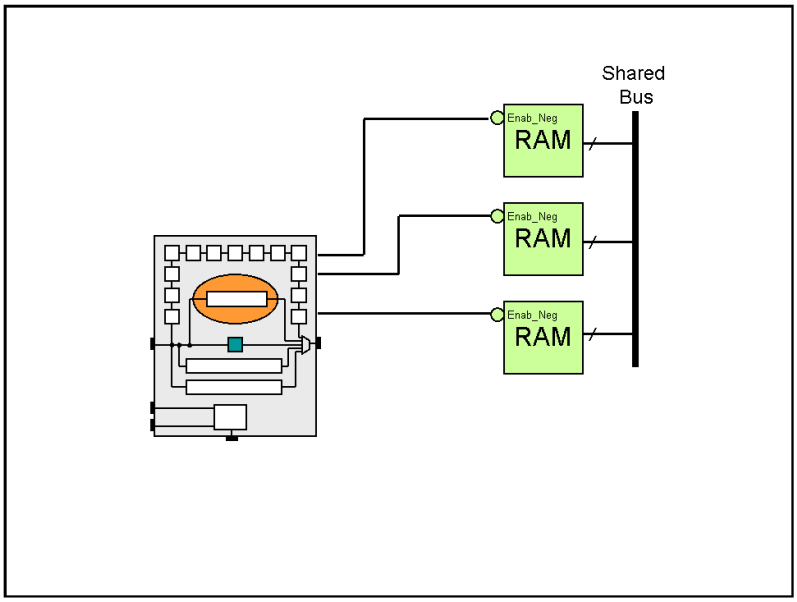


Figure 20: A Reason for the Hold State

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 *Preload* instruction to load safe values into the hold sections. Then, use the *Clamp* instruction to 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 with pin fan-out, if each output pin has a boundary-scan cell, then *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.

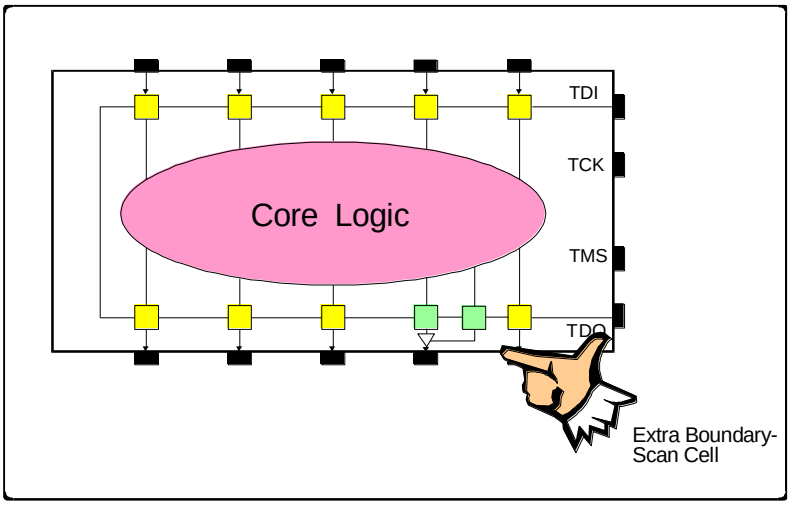


Figure 21: Control of Tristate Outputs

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.

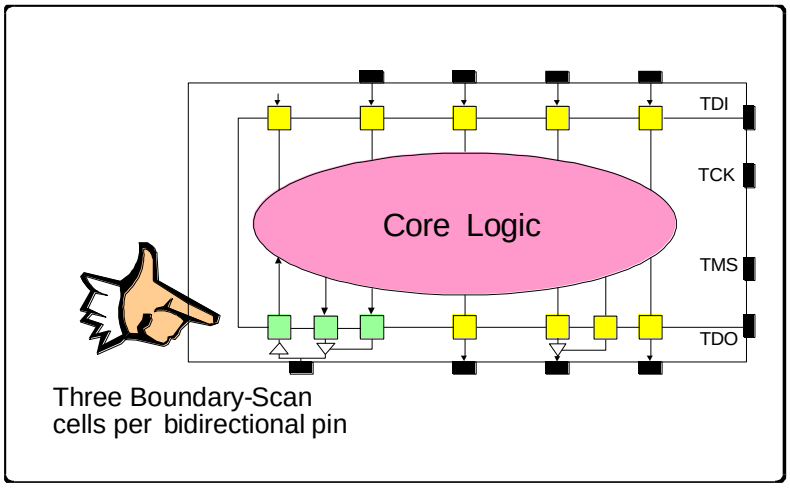


Figure 22: Bidirectional Input/Output Pins

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 device-internal 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, more usually, a much faster 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.

The 2013 Version

In 2013, a major revision to the 1149.1 standard was ratified. Importantly, devices compliant with previous versions of the 1149.1 standard are compliant with the 2013 version. The 2013 version offers optional additions. For more comprehensive details, refer directly to the IEEE 1149.1-2013 standard. [2] The changes to the 1149.1 standard are summarized below:

- A test mode persistence controller that can keep the test mode active even if the active instruction does not force test mode. Three new instructions (CLAMP_HOLD, TMP_STATUS, AND CLAMP_RELEASE) support this controller.
- A new, optional instruction (ECIDCODE). This instruction supplements IDCODE and USERCODE and queries the Electronic Chip Identification value used to identify individual integrated circuits.
- A component initialization mechanism that allows more options in initializing components for test. Three new instructions (INIT_SETUP, INIT_SETUP_CLAMP, and INIT_RUN) support this mechanism.
- A new test data register (reset_select) that allows TAP-controlled reset functions for components. A new instruction (IC_RESET) supports this register.
- A recommended, but optional, TAP to test data register interface.

- Rules to support observe-only boundary-scan register cells. These can capture signal values or fault conditions on target pins.
- Rules to support excludable boundary-scan register segments.
- Updates to the Boundary Scan Description Language definitions.
- Codification of Procedural Description Language (PDL). PDL is used to document the procedural and data requirements for some of the new instructions.

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 the 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?

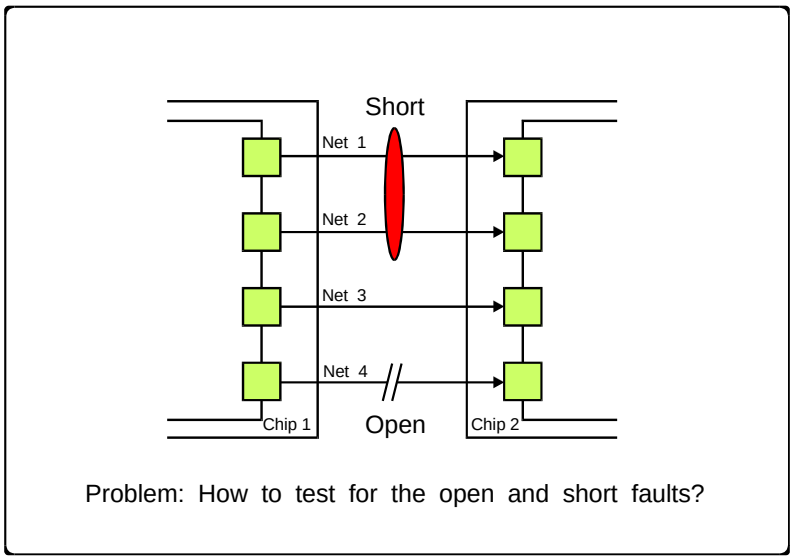


Figure 23: Interconnect Testing Example

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.

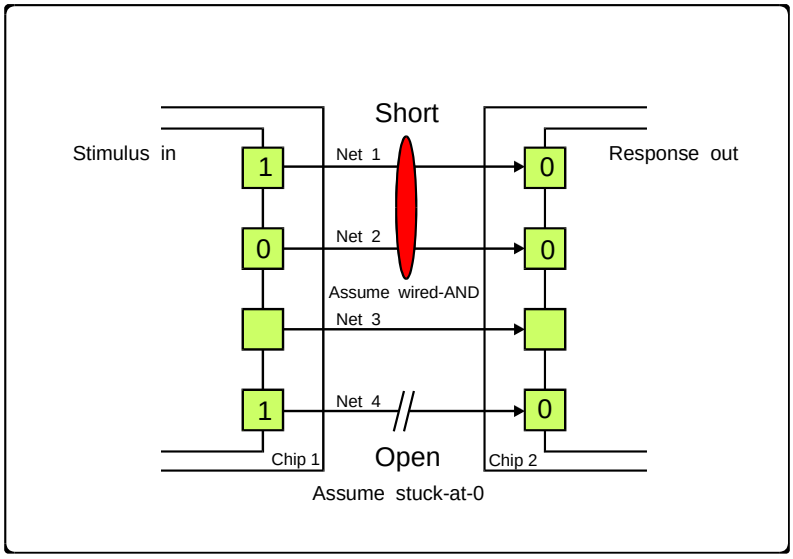


Figure 24: Interconnect Testing Solution

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* $[\log_2(N)]$,

[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.

where N is the number of nets and *ceil* means ceiling (the upper integer value of the logarithm). This is illustrated in Figure 25.

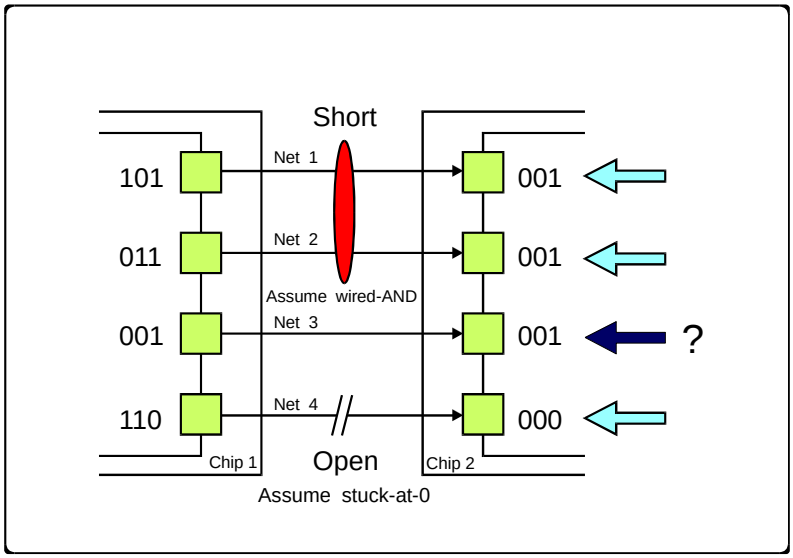


Figure 25: Detecting the Fault

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.

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 the net 3 response is correct in terms of it 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).

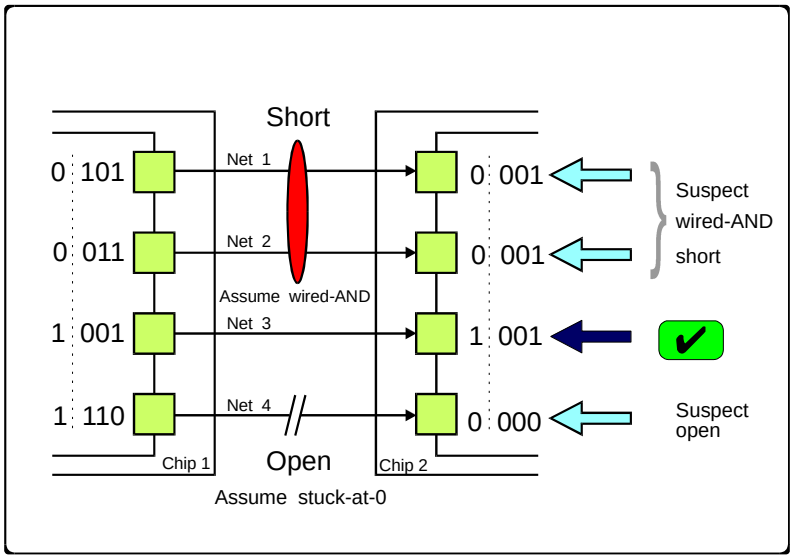


Figure 26: Locating the Fault

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 and non-boundary-scan 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 the non-boundary scan devices through the boundary-scan registers of boundary scan devices.

Figure 27 shows a "cluster" of three non-boundary scan devices surrounded by three boundary scan devices. The boundary-scan registers in U1, U2, U3 can be used to drive test-pattern stimuli into the non-boundary scan cluster, and to observe the cluster responses, but it will be difficult to control and observe the truly buried nets inside the cluster (e.g., between U4 and U5).

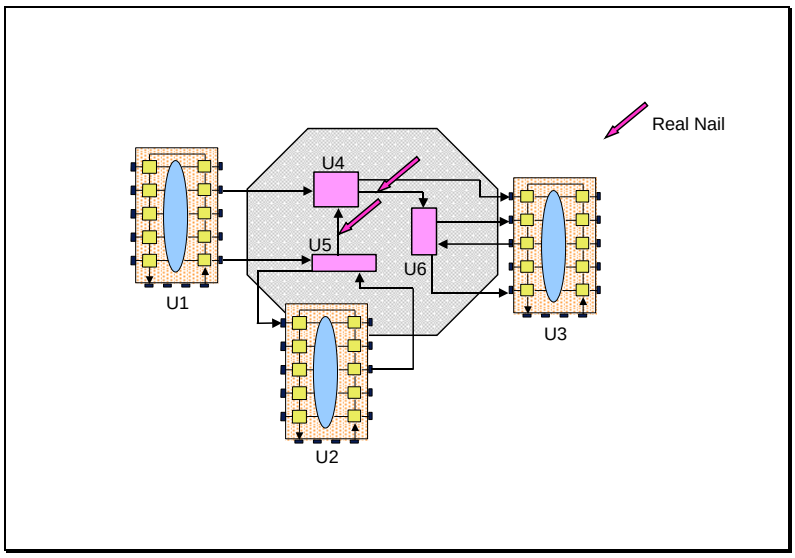


Figure 27: Handling Non-Boundary Scan Clusters

Given that we are not testing the full functionality of the non-boundary scan devices — only their presence, orientation and bonding — one solution is to develop a suitable set of tests for the non-boundary scan 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-boundary scan devices, generating these tests may not be too difficult. For clusters of complex non-boundary scan 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 (i.e. physical probes) 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 (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-boundary scan devices.

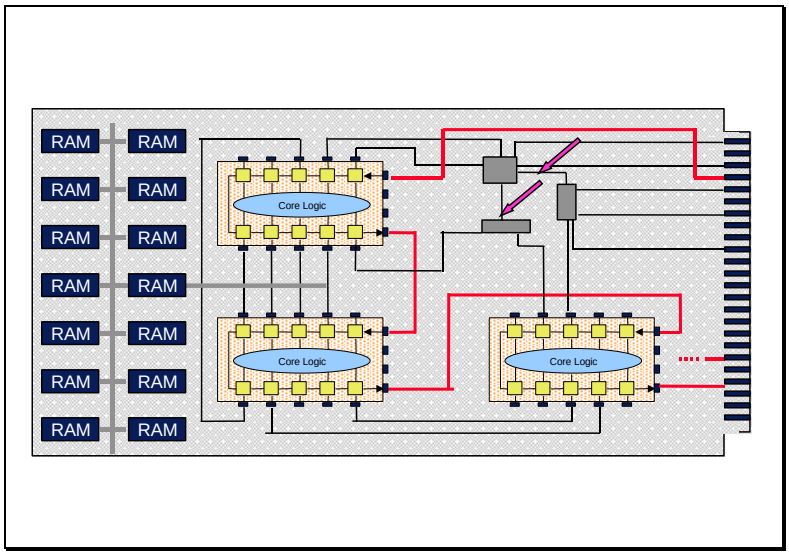


Figure 28: Testing a RAM Array Via Boundary Scan

Boards that contain RAMs typically also contain 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 acceptable as long as there is a microprocessor on the board. If there is no such device, then the RAMs can be tested for manufacturing defects through the boundary-scan registers of boundary scan devices as long as these registers have access to the control, data and address ports of the RAMs. Test times will be slow but the number of tests will not be significant since 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 Boundary Scan-to-Non-Boundary Scan Interfacing

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

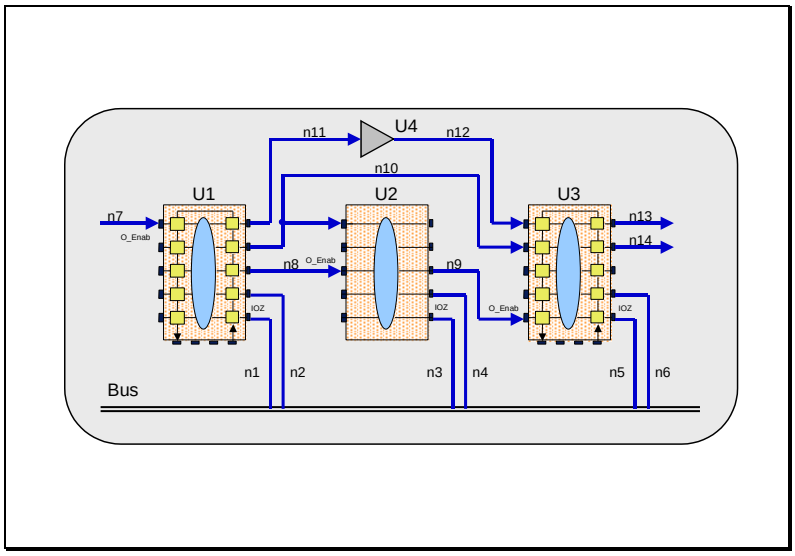


Figure 29: Boundary Scan-to-non-Boundary Scan Interface

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-boundary scan 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 boundary scan 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 which 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-boundary scan device. This data, which is sometimes called *characteristic* or *cluster-model* data, is easily created and 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.

Next, let's consider net n10. This net connects between the two boundary scan devices, U1 and U3, and, as a result,is a candidate for interconnect testing. Note however that the net also connects to the non-boundary-scan 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 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 boundary-scan-to-non-boundary-scan-to-boundary-scan 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 boundary-scan-to-non-boundary-scan 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 is to prepare a board-level test program.

Assembling the Final Test Program

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

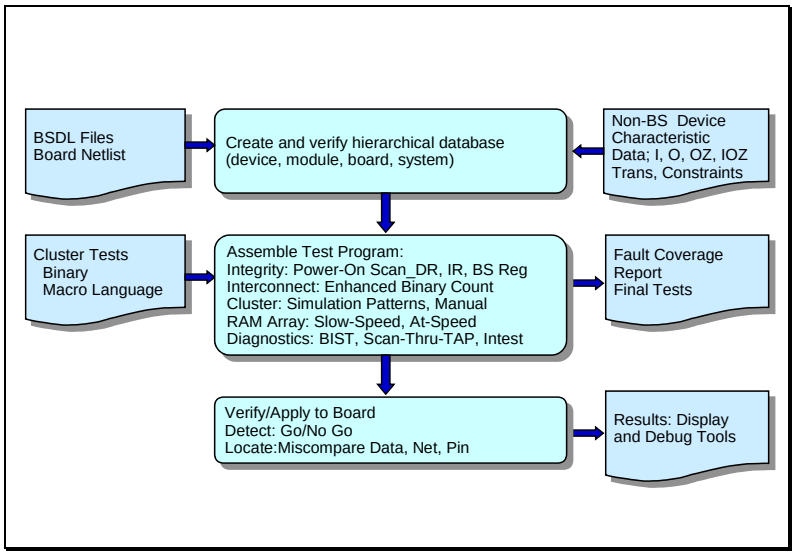


Figure 30: Assembling a Test Program - Tool Flow

First, the device BSDL files (more on this later) and board netlist data is used to compile a database. Non-boundary-scan characteristic or model data is also assembled so that it can be used by the various pattern generators. The test program itself is composed of several segments:

- Board-level infrastructure integrity test, which tests the following: 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. This test sets non-boundary-scan devices into safe states and/or uses non-boundary-scan outputs to assert control over boundary-scan devices where necessary.

- Tests to be applied to non-boundary-scan clusters via a combination of boundary-scan 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 boundary-scan 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 USB socket. The PC is connected 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 or monitor a flash memory device's Ready/Busy 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 the Windows OS.

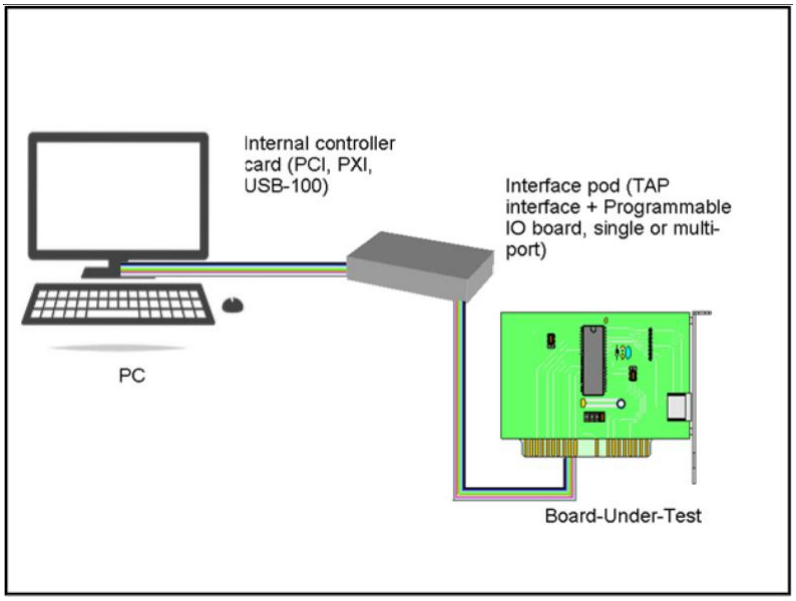


Figure 31: Tester Hardware

Such board testers are low-cost, compared to traditional in-circuit testers. In addition, PC-based testers can be 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 Data Formats

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 and STAPL.

Boundary-Scan Description Language (BSDL)

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

What Is 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 in their designs, including Automatic Test Pattern Generation (ATPG) for interconnect tests and Automatic Test Equipment (ATE).

When you use tools that support BSDL, you should obtain BSDL files for boundary-scan devices from your semiconductor vendor. This results 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?

Originally, Texas Instruments (TI) developed the Hierarchical Scan Description Language (HSDL) to complement BSDL using the same subset of VHDL statements as BSDL. TI transferred HSDL ownership to ASSET Intertech, INC. in 1995, which is now 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. HSDL describes 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/System in Packages, 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 which are on the scan path into bypass mode. To accomplish this, the IEEE 1149.1 test controller must 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 at 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 containing 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, non-scannable 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 target 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 gives several examples of the default paths that are 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, 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 de-asserting 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.

Standard Test And Programming Language, STAPL

What is STAPL?

The Standard Test And Programming language, STAPL, was developed in the late 1990s to overcome the limitations of SVF when used to program on-board CPLDs and FPGAs via their respective 1149.1 interfaces. The language was created by taking a proprietary Altera programming language called JAM

(not an acronym) and wrapping a BASIC program control flow around it. The resulting language was then submitted to the JEDEC (formerly known as Joint Electron Design Engineering Council) Solid State Technology Association and finally approved as JEDEC's JESD 71-1999 STAPL Standard, available for free from www.jedec.org/download/search/jesd71.pdf

Basic Structure of a STAPL program

A STAPL file contains four main sections:

1. NOTE statements containing information-only text strings.
2. CPLD ACTION statements, such as bulk or sector erase, program the device, verify programmed content, add program security, check IDCODE values, discharge high-voltage charge pumps, etc.
3. PROCEDURE blocks and DATA blocks, containing STAPL executable processing and computation statements and associated data. PROCEDURE operation statements are based on but extended from SVF statements. Here are some examples:

IRSCAN (instead of SVF's SIR),
DRSCAN (instead of SVF's SDR)
FREQUENCY (to change TCK frequency),
POSTDR (modify the behaviour of a DRSCAN operation)

PROCEDURE flow control statements are also available. Here are some examples:

```
IF <Condition> THEN,
GOTO,
FOR loops,
```

PROCEDURES are identified inside ACTIONS.

1. A CRC statement containing the Cyclic Redundancy Code for the complete file to verify that the file has not been corrupted.

STAPL Composers, Players and Sessions

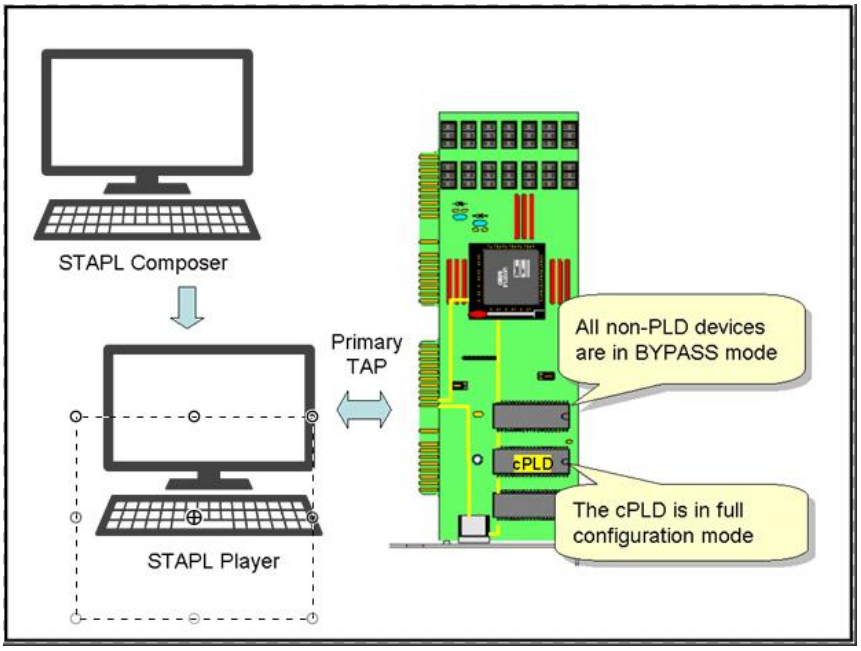


Figure 32: Programming a CPLD Through the Scan Chain

A STAPL program is *composed* using a Composer and then *played* using a Player. A STAPL *session* occurs when a STAPL program is executed

A STAPL Composer can be a free-standing utility or part of an integrated program-preparation-and-delivery system. The only requirement is that the final program is compliant with the JEEC STAPL Standard and that it contains the following:

- All mandatory NOTE fields
- At least one ACTION statement
- One or more PROCEDURE and DATA blocks
- One CRC statement.

STAPL players must support both interpreted and compiled STAPL programs and must posses the following characteristics:

- Execute a STAPL file
- Process user-specified ACTIONS and PROCEDURES

- Check the CRC signature.
- Extract information from NOTE fields
- Correctly access the 1149.1 signals at the primary TAP
- Access other special control signals e.g. signals outside of the control of 1149.1
- Be able to create real-time delays
- Report results

STAPL Program Example

The following simple example of a STAPL program reads the ID code from an 1149.1 CPLD.

```
' Lines with a leading apostrophe character are Comment lines
' This program reads the ID code of a CPLD 32 times
' The expected code is 00FDEC01
' The IDCODE instruction code is a 9-bit code 001101000

' The following are the mandatory NOTE statements
NOTE "CREATOR" "STAPL Composer v2.3;
NOTE "DEVICE" "Test32MC";
NOTE "DATE" 2007/01/23";
NOTE "STAPL_VERSION" "JEDS00-A";
NOTE "ALG_VERSION" "3";
NOTE "STACK_DEPTH" "2";
NOTE "MAX_FREQ" "10000000";    ' 10MHz
NOTE "TARGET" "1";
NOTE "IDCODE" "00FDEC01";

' Beginning of the executable part of the file.
' Define an ACTION for the file.
PROCEDURE DO_READ_IDCODE;

' Declare variables for data arrays.
BOOLEAN capture_data 32;
BOOLEAN idcode_instr[9] = #001101000;
BOOLEAN all_ones[32] = #FFFFFFFF;
INTEGER i;

' Initialize the device to the Test-Logic Reset state
STATE RESET;

' Load IDCODE instruction code.
IRSCAN 9, idcode_instr[8..0];

' Capture IDCODE values 32 times and shift out.
' Check that least-significant bit is always a logic-1.
' This acts as a crude connection and signal integrity test.
DRSCAN 32, all_ones[31..0], CAPTURE capture_data[31..0];

' Display captured value on console.
EXPORT "IDCODE", capture_data[31..0];
ENDPROC;

' Finally, check file CRC signature
```

```
CRC 7358;
```

STAPL: final comments

Since its inception in 1999, the acceptance of STAPL by the industry has been *slow but steady*. Its acceptance was boosted by the development of the IEEE 1532 In-System Configuration Standard in 2000 and, more recently, the SJTAG working group has seriously considered STAPL as the language for describing system-level JTAG operations in a multi-board environment. But, it remains to be seen if all PLD vendors will support it and if the SJTAG working group will be able to extend it to suit the purposes of that initiative.

Chapter 6: IEEE 1532 In-Circuit Configuration Standard

In this chapter, concepts concerning the in-system configuration of programmable logic devices on boards and the approach taken in the 1532 Standard are presented.

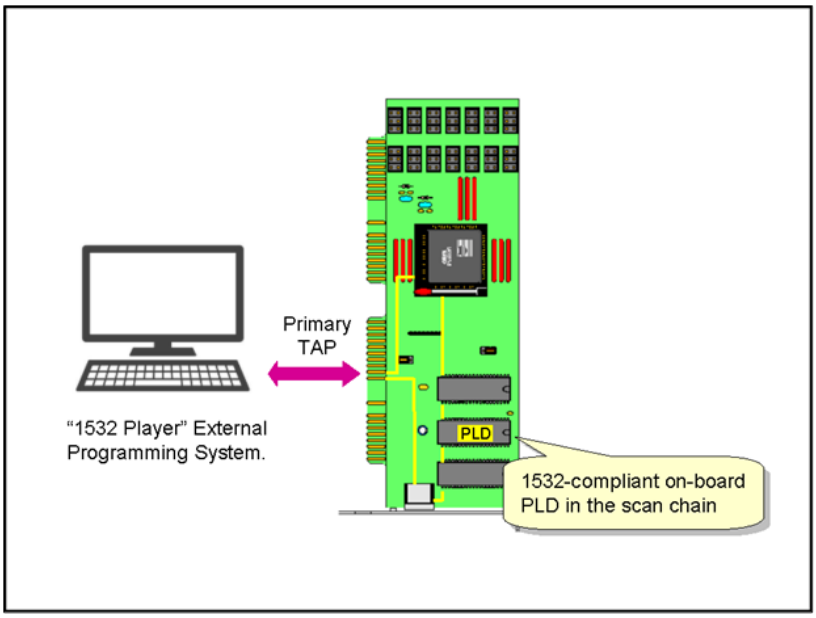


Figure 33: Background on In-System Configuration

Development of the IEEE 1532 Standard

In the early 90s when boundary-scan was first established, the Programmable Logic Device (PLD) vendors were among the first to adopt this new technology. They did so for in-system configuration purposes – that is, the programming of on-board PLDs *in situ*. The PLD vendors did this by adding

1149.1 features to their devices and then placing the devices in the board-level scan path. They quickly realized that 1149.1 access to the internal data and address registers was an ideal way to load configuration data into a PLD after the device had been assembled onto the board. The problem was that each vendor developed a proprietary method for defining the program data and controlling the configuration process. This resulted in devices that had to be individually programmed. Attempts at data-format standardization were made, including using ASSET's Serial Vector Format and Altera's JAM with its JEDEC derivative, STAPL (see Chapter 5), but the result was that boards containing multiple PLDs from different vendors still required that each PLD be individually programmed. This became a limiting factor in board manufacturing.

PLD Programming Environment

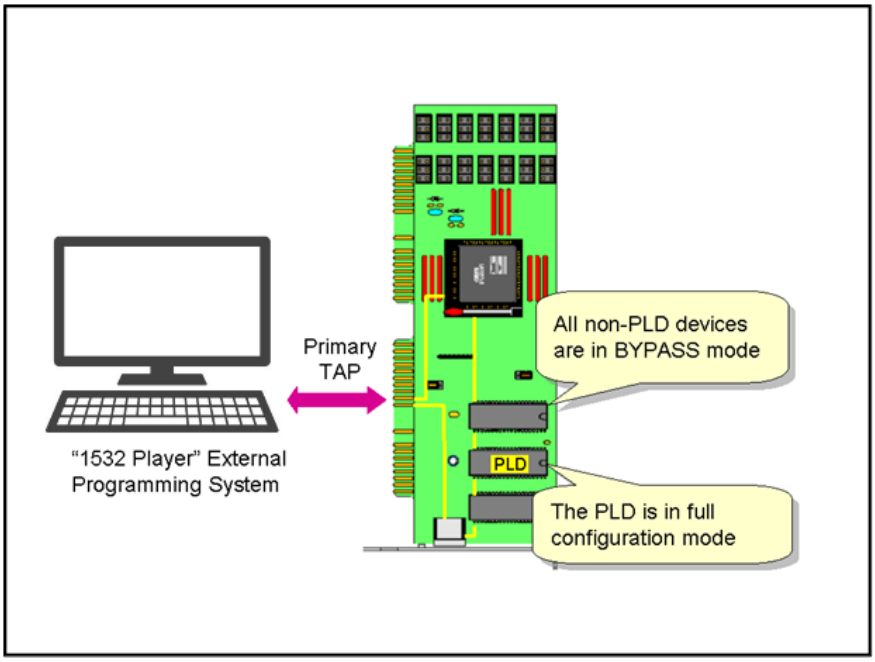


Figure 34: Programming the PLD Through the Scan Chain

Here is a typical in-system configuration programming environment.

Basically, all the non-PLD boundary-scan devices are placed in *BYPASS* mode but the PLD is in full programming configuration mode. Through special 1532 instructions, the program data register and program address register are both placed between the PLD's TDI and TDO pins. In this way, an external

programming system, such as ASSET's *ScanWorks*®, can access the PLD for programming purposes through the board's primary Test Access Port.

Before we get to the detail of 1532, let's take a look at some of the reasons for wanting to configure PLDs after they have been mounted on a board:

- Inventory management is simplified.
- The need for off-line programming stations is either reduced or removed completely.
- Rapid prototype configuration and especially re-configuration increases design flexibility and response-to-change requests.
- The need for on-board sockets is removed. Such sockets are often a cause of pin damage or interconnect failures.
- Similarly, the risk of damage caused by mechanical handling and electro-static discharge is also reduced.
- And finally, the ability to carry out system and field-service program upgrades assists system debug.

PLD Programming Formats and Languages

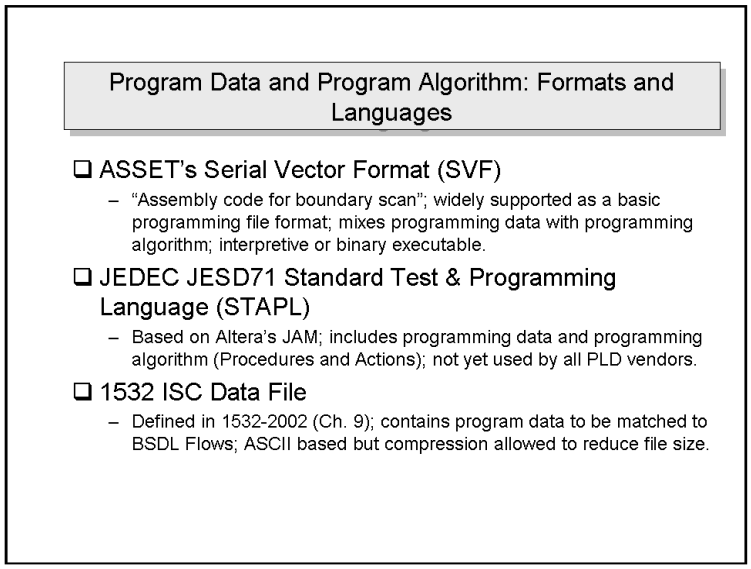


Figure 35: PLD Programming Formats and Languages

PLD formats and languages for describing program data and program algorithms should be clarified at this point. The figure shows three of the more well-known formats and languages, but there are others, which are usually proprietary technology developed by PLD vendors. Of the three mentioned above, Serial Vector Format, although basic, has become a baseline generic format supported by virtually all PLD and ATE vendors. STAPL, the JEDEC ***Standard Test And Programming Language***, is a more-recent attempt to produce a vendor-neutral programming language. It is based on Altera's JAM language and SVF. SVF and STAPL are described in more detail in Chapter 5.

The 2002 version of 1532 defined its own data format using ASCII to define data that matched the programming ***Flows*** built into the extended BSDL file. More on this later.

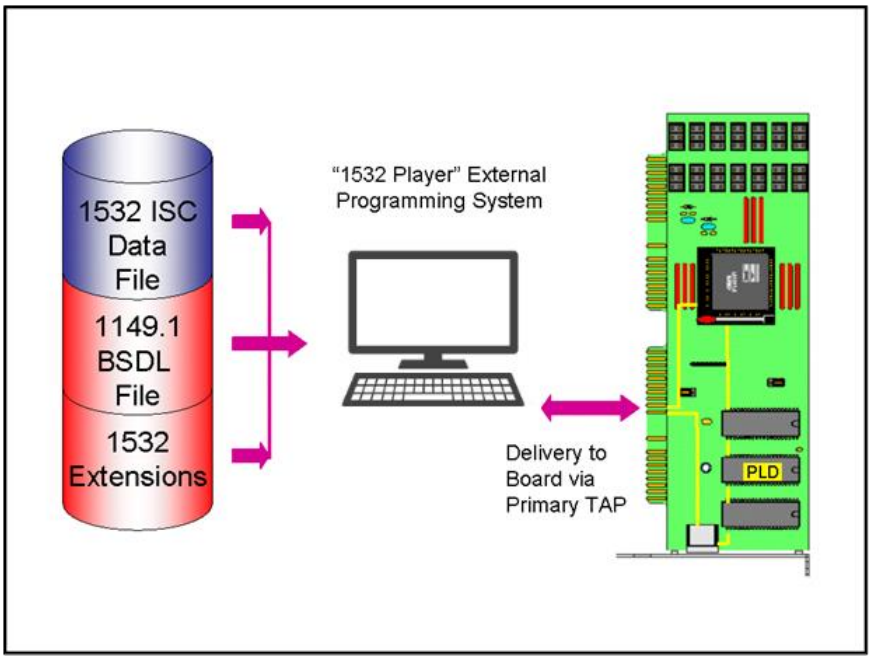


Figure 36: Getting Programming Data to the PLD

In terms of delivery of the program data and program algorithms to the on-board PLD via the scan chain, this is achieved by a so-called "1532 Player" program-delivery-and-verification-system, such as ASSET's *ScanWorks*®. In addition to the usual device BSDL files and board-level netlist, the Player will require the 1532 in-system configuration data file, plus the BSDL file for the on-board PLD, or PLDs, and the 1532 BSDL extensions. More information on these extensions will be presented later.

Actual delivery to the on-board PLD is via the primary board TAP.

IEEE 1532 In-System Configuration Standard

The IEEE 1532 Working Group was formed in 1996. The mission of the group was "to define, document and promote the use of a standardized process and methodology for implementing programming capabilities within programmable ICs utilizing and compatible with the IEEE 1149.1 communication protocol …" Basically, to configure or reconfigure, read back, verify or erase PLDs after they have been assembled on a board. The 1532 Standard defines several new internal registers to assist configuration programming, along with new mandatory and optional instructions compatible with the 1149.1 physical and logical protocols.

Please note that In-System Configuration, ISC, is also known as In-System Programming, ISP, but because ISP is a trademark of Lattice its use is not approved by the IEEE Standards Authority.

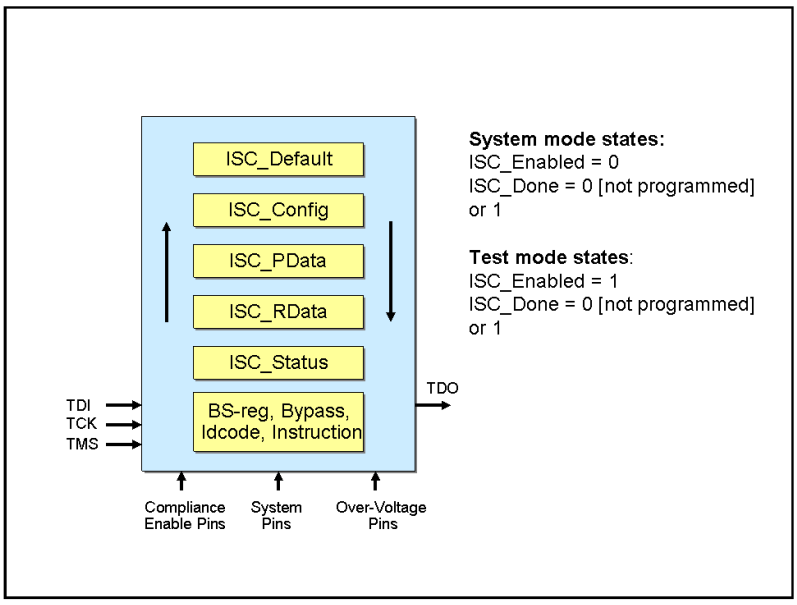


Figure 37: IEEE 1532 - General Architecture

Here we see the general architecture of a 1532 device. In addition to the standard 1149.1 ports, registers and TAP controller, a number of new registers are defined together with two new internal control signals: ***ISC_Enabled*** and ***ISC_Done***. ***ISC_Enabled*** indicates that the device is either ready for a programming activity or not. ***ISC_Done*** indicates that a program has either been successfully written into the device or not.

Now, to return to the new registers.

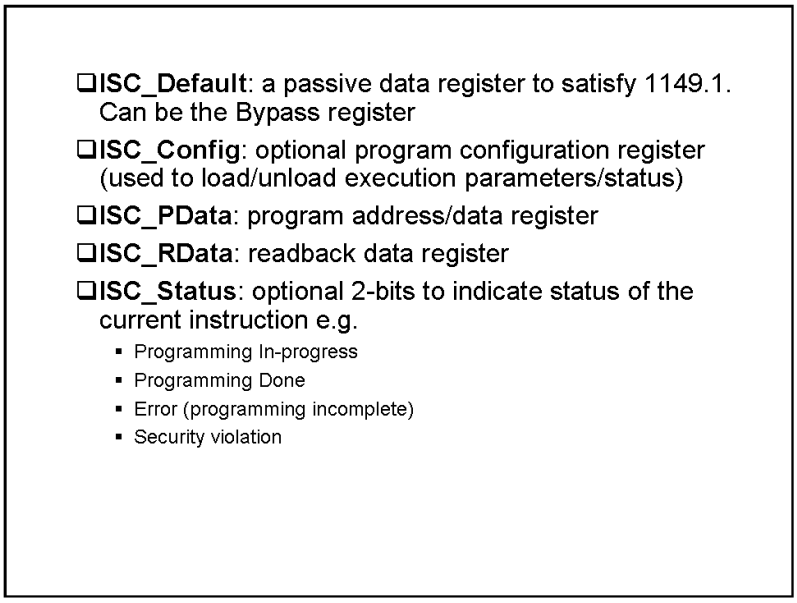


Figure 38: IEEE 1532 - New Registers

Some of the new registers, and their function, are listed above, but the following offers more detail.

- ***ISC_Default*** is used by any in-system configuration instruction that does not need to shift-in or shift-out data to control or monitor an operation. An example could be a bulk erase operation. However, it is a requirement of 1149.1 that ***a*** register is placed between TDI and TDO by any instruction. The typical solution in 1532 is to use the Bypass register as I***SC_Default.***
- ***ISC_Config*** is an optional program configuration register used for pre-setting internal configuration data. Values captured or loaded determine the results or parameters of a programming environment. An example could be to select a specific part of the array for programming.
- ***ISC_PData*** is the program address and/or data register.
- ***ISC_RData*** is the readback data register.
- ***ISC_Status*** is an optional 2-bit register that indicate status of the current instruction, as shown in the Figure.

- ***ISC_Sector***: used to identify programming sectors in the memory array
- ***ISC_Info***: contains current state and factory-set information
- ***ISC_Data***: allows access to the data portion of an address/data pair.
- ***ISC_Address***: allows access to the address portion of an address/data pair
- ***ISC_Inc***: allows control of address modification during programming
- Compliance enable pins: statically conditioned before 1149.1 facilities become compliant
- Fixed system pins: IO function not determined by configuration data e.g. Microcontroller + PLD on the same device

Figure 39: IEEE 1532 - Other New Registers (Optional)

Other optional registers defined in the Standard are listed in this Figure.

Accessing Program Data and Address Registers

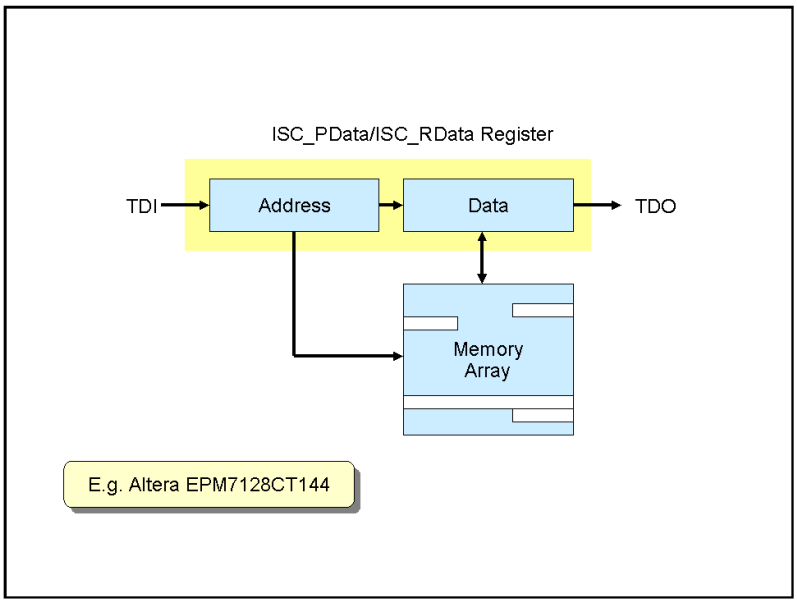


Figure 40: Basic Program Memory Array Access

This figure shows one of the ways by which the internal data and address registers can be configured during programming operations that write or read data into or from specific addressed locations. Note that the 1532 Standard does not mandate the architecture of the memory array access circuitry.

The diagram shows a basic scheme whereby ***Address*** and ***Data*** are concatenated to form one accessible register. The memory array holds the configuration data that determines the programmed function of the device.

IEEE 1532 Instructions

We'll now take a look at the various mandatory and optional instructions defined in the 1532 Standard.

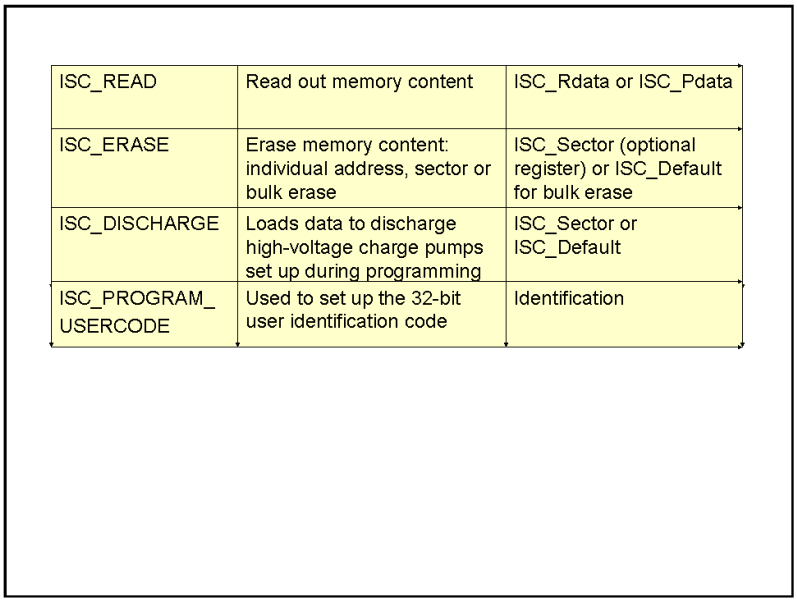
EXTEST, BYPASS, PRELOAD, SAMPLE	Normal 1149.1 instructions	Boundary Scan or Bypass
IDCODE, USERCODE	Now mandatory for 1532	Identification
ISC_ENABLE	Enables programming environment. Sets ISC_Enable signal	ISC_Config or ISC_Default
ISC_DISABLE	Disables programming environment. Resets ISC_Enable signal	ISC_Config or ISC_Default
ISC_PROGRAM	Accesses program address + data registers	ISC_PData or ISC_Default
ISC_NOOP	Used to place device into a passive state even though device is in ISC_Accessed modal state e.g. used for programming multiple devices in parallel	ISC_Default

Figure 41: Mandatory Instructions

First, a 1532-compliant device must also be fully compliant with 1149.1. This means that the four mandatory 1149.1 instructions are also mandatory for 1532.

Second, the two optional 1149.1 instructions, IDCODE and USERCODE, become mandatory for 1532.

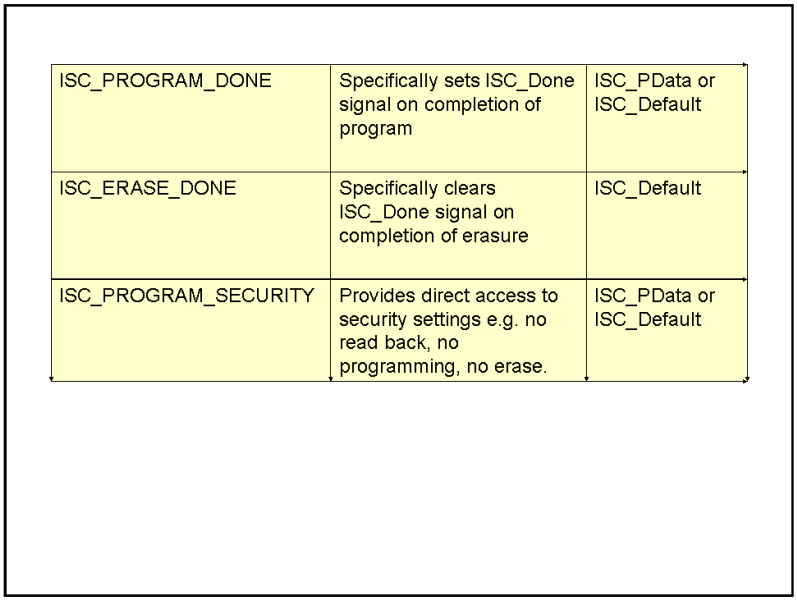
The remaining four new instructions – ISC_ENABLE, ISC_DISABLE, ISC_PROGRAM and ISC_NOOP – are the basic instructions required to enable or disable the programming mode and to carry out a programming action.



ISC_READ	Read out memory content	ISC_Rdata or ISC_Pdata
ISC_ERASE	Erase memory content: individual address, sector or bulk erase	ISC_Sector (optional register) or ISC_Default for bulk erase
ISC_DISCHARGE	Loads data to discharge high-voltage charge pumps set up during programming	ISC_Sector or ISC_Default
ISC_PROGRAM_ USERCODE	Used to set up the 32-bit user identification code	Identification

Figure 42: Optional Programming Instructions

This figure defines optional instructions to facilitate the programming process depending on the existence, or otherwise, of the associated feature in the PLD.



ISC_PROGRAM_DONE	Specifically sets ISC_Done signal on completion of program	ISC_PData or ISC_Default
ISC_ERASE_DONE	Specifically clears ISC_Done signal on completion of erasure	ISC_Default
ISC_PROGRAM_SECURITY	Provides direct access to security settings e.g. no read back, no programming, no erase.	ISC_PData or ISC_Default

Figure 43: Optional Program Control and Security Instructions

Similarly, this figure defines more optional instructions – this time to facilitate program control and also to set program security parameters.

ISC_READ_INFO	Allows read access to a non-volatile ISC data register containing package info e.g. mask ID, foundry, date manufactured, lot number, etc	ISC_Info
ISC_DATA_SHIFT	Facilitate direct loading of program data and memory readout	ISC_Data (may be part of ISC_Pdata reg.)
ISC_ADDRESS_SHIFT	Allows data set up by ISC_DATA_SHIFT to be loaded into a separate Address register	ISC_Address (may be part of ISC_Pdata reg.)
ISC_INCREMENT	Triggers an internal increment to the content of the ISC_Address register	ISC_Inc or ISC_Default
ISC_SETUP	Loading or updating configuration data to modify/adjust the behaviour of other ISC instructions	ISC_Config

Figure 44: Optional Address and Data Access Instructions

These optional instructions allow access to additional hard-wired internal data and to refine the address and data-access operations.

As you see, 1532 is very rich in its instruction set and, of course, additional private instructions can still be developed to accommodate a variety of special features available in commercial PLDs.

Flows, Procedures and Actions

Turning to the Boundary Scan Description Language requirements, there is an extensions mechanism in BSDL that will allow the special attributes of 1532 features to be defined. This is a straight-forward process and does not merit any special comment, but one thing that has been added to 1532 BSDL is the ability to define programming ***Flows, Procedures*** and ***Actions***. These **are** worth further elaboration.

- An in-system configuration ***Flow*** is a logical sequence of in-system configuration instructions that perform a basic programming function. Examples could be ***program array, read array, erase***, ***check ID***, and others. ***Flow*** also specifies how programming data is to be provided to a programming process. Some examples are hard-coded data, address data, R/W data.
- An in-system configuration ***Procedure*** is an allowed sequence of ***Flows.*** For example, ***Procedure Erase*** could be based on the flows ***check ID*** followed by ***erase***.

- An in-system configuration ***Action*** is an allowed sequence of ***Procedures***, such as a complete programming cycle.

This hierarchical method for defining programming operations is illustrated in the next figure.

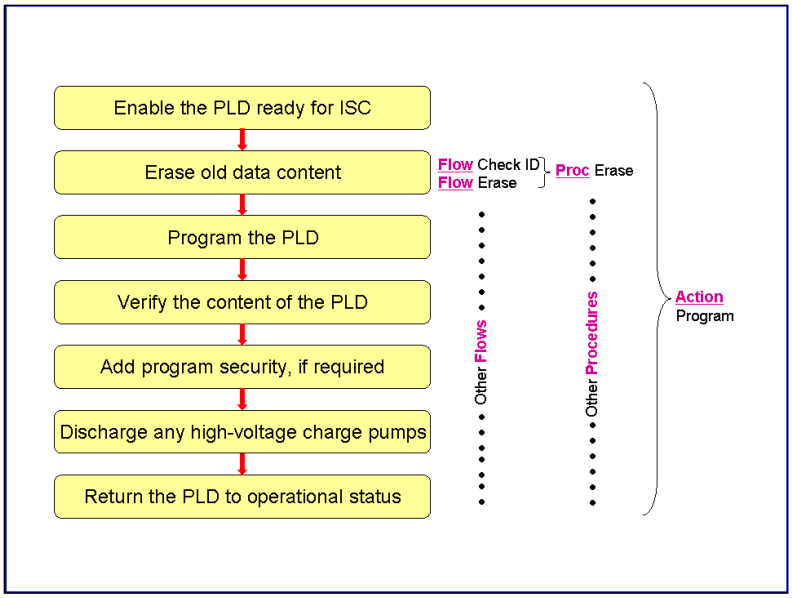


Figure 45: Top-Level ISC Programming Flows, Procedures and Actions

The diagram shows a possible sequence of operations to program a 1532-compliant device. The ***Flow, Procedure*** and ***Action*** examples just mentioned are also shown.

Conclusions

To conclude, IEEE1532 is intended to support three different types of programming environments:

- Programming a free-standing single device using a device programmer.
- In-system configuration of a single device on a printed-circuit assembly.
- Concurrent in-system configuration of multiple devices on a printed-circuit assembly.

In reality, the last two environments are where 1532 really comes into its own.

Adoption of the 1532 Standard has been “slow but steady” and companies such as ASSET InterTech now provide programming support for boards containing 1532-compliant devices.

To Probe Further …

- IEEE 1532 ***In-System Configuration of programmable devices,*** 2002, http://standards.ieee.org/catalog/
- Neil Jacobson, ***The In-System Configuration Handbook***, Kluwer, 2003
- Ken Parker ***Boundary-Scan Handbook 3rd Edition***, Springer, 2003, Chapter 9
- Neil Jacobson, Proc. IEEE International Test Conference 2000, Paper 32.1

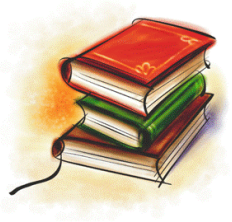


Figure 46: IEEE 1532 - To Probe Further

Chapter 7: The IEEE 1149.6 Standard

The IEEE 1149.6-2003 Standard for *Boundary-Scan Testing of Advanced Digital Networks* was approved in March 2003 and was created to solve test problems associated with DC-coupled and AC-coupled single-ended and differential signal interconnects. This chapter briefly describes these problems and summarizes the approach taken by this new standard.

What's The Problem?

In terms of "what's the problem?", the answer is that testing AC-coupled interconnects is both different from and more difficult than testing DC-coupled interconnects, especially if the interconnects are differential. But many modern data buses now make use of AC-coupled differential interconnects so a closer look at the bus structures is warranted. Nowadays, there are a variety of high-speed bus styles, such as the following:

- S-ATA 1.0 (1.5 Gbits/s Serial Advanced Technology Attachment for storage interface communication)
- PCIe (Peripheral Component Interconnect Express)

- HyperTransport
- Infiniband-embedded
- FibreChannel
- Serial RapidIO
- XAUI
- GigaBit Ethernet
- And several others

All of these buses share one salient feature: they are intended for gigabits-per-second data transmission and, in some cases, for device interconnects operating at different output-voltage levels. It is this latter feature that leads to the AC-coupled requirement.

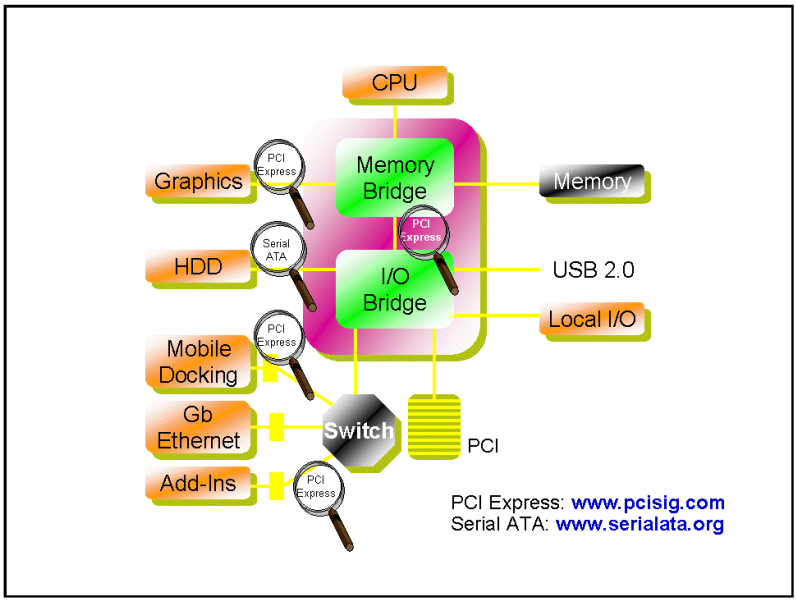


Figure 47: High-Speed Serial Buses - Application to PC Motherboard

Just to show the proliferation of these high-speed buses, the figure shows a PC motherboard making use of two of the more popular buses: PCI-Express and the Serial Advanced Technology Attachment bus.

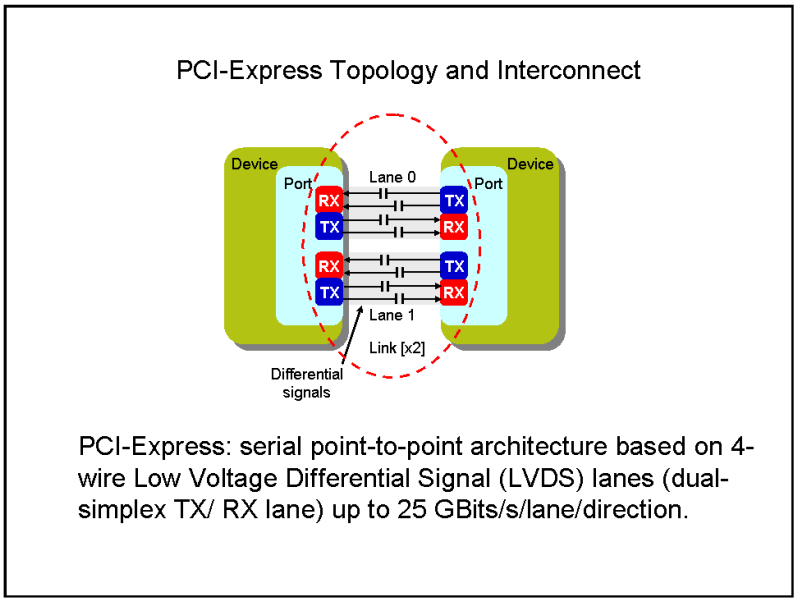


Figure 48: PCI-Express Lane Architecture

Digging a little deeper into the architecture of a PCI-Express system, here we see two 4-wire Transmit/Receive lanes. Looking specifically at Lane 0, we see the two AC-coupled Low Voltage Differential Signal, LVDS, interconnect pairs between the two devices as required by the PCI-Express Standard.

DC and AC-coupled Low-Voltage Differential Signals

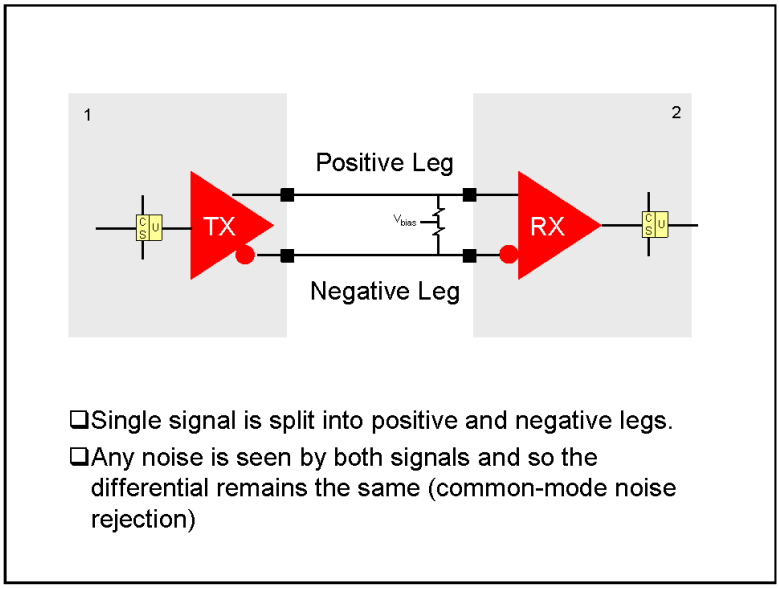


Figure 49: Differential DC Coupling

Here is a typical DC-coupled differential pair. The signal coming from the transmitter driver is split into two legs: a positive leg and a negative leg. The terminating resistor at the receiver develops a positive or negative voltage that is interpreted by the receiver amplifier as a logic-1 or logic-0 depending on the polarity of the voltage.

In this diagram we also see standard 1149.1 boundary-scan cells upstream of the transmitter and downstream of the receiver but note that these cells do not give us independent control of each leg of the interconnect. As a result, defect coverage would be limited to a very small subset of the actual possible opens and shorts that can occur on the differential pair. A key objective of 1149.6 is to improve significantly the defect coverage on differential interconnects.

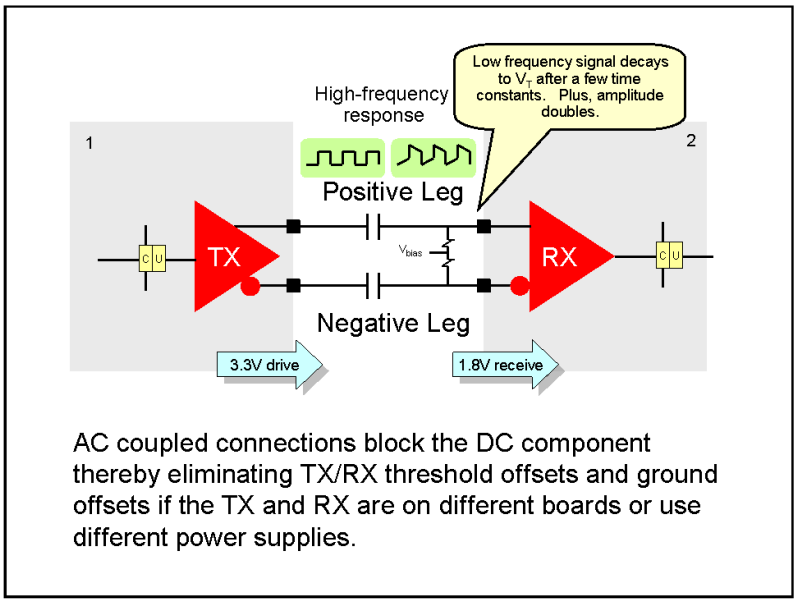


Figure 50: Differential AC Coupling

If the signal drive-voltage from device 1 on the left is different from the signal-receive voltage of device 2 on the right, then we will need to block the DC component of the signal with a series capacitor. This creates an AC-coupled connection. Now, we are truly prevented from using regular 1149.1 boundary-scan cells. 1149.1 assumes a DC connection from the drive scan-cell to the receive scan-cell.

It was this basic limitation of 1149.1 that led to the creation of the 1149.6 standard.

SERializer-DESerializer, SERDES, Structures

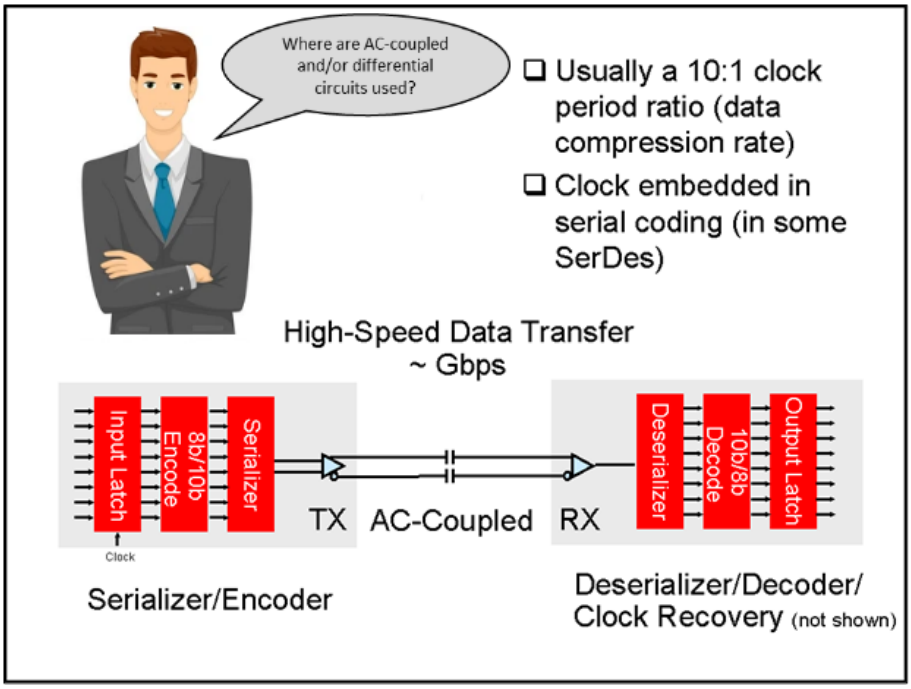


Figure 51: AC-Coupled SERDES Interconnects

Before we get to the 1149.6 solution, let us ask the question "where do we see AC-coupled differential interconnects?"

Mostly, these high-speed interconnects occur between SERializer-DESerializer devices : the so-called SERDES designs. Inside the SERDES devices, data is handled at a lower frequency such as 200 Megabits-per-second and placed in multiple parallel registers. Before transmission, all the data is collected into a serializing register and transmitted at a much-higher frequency. To keep the numbers simple, if we had ten equal-length parallel registers all working at 200 Megabits-per-second and we collect all the data into a single serializer register before transmission, we would have to send at ten times the internal 200 Megabits-per-second speed – that is, at 2 Gigabits-per-second.

At the far end, we would collect all the data into a single receive register and break it back down to blocks of data to go into multiple parallel registers where we can process at the original frequency of 200 Megabits-per-second.

The point here is that the processing speed is lower than the transmission speed and we really need to check the integrity of the transmission path as well as what is going on inside the devices.

Where Can Defects Occur?

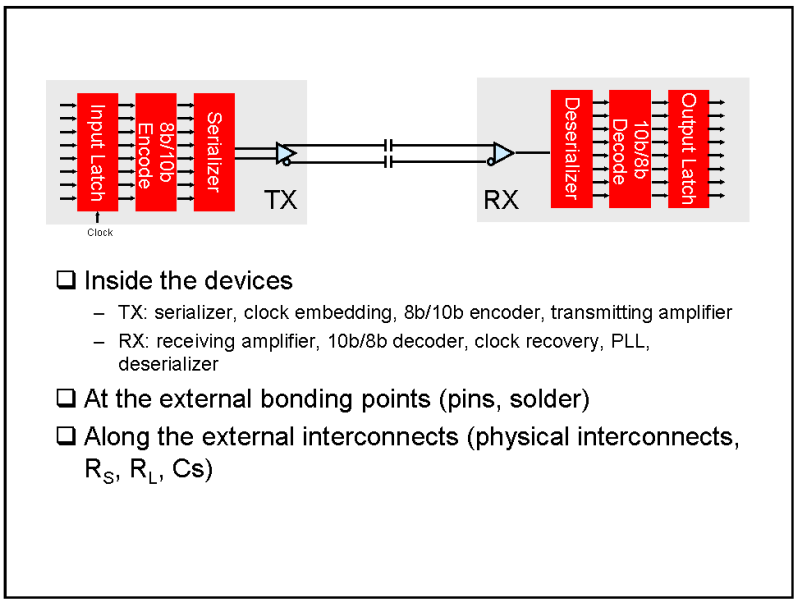


Figure 52: Where Can Defects Occur?

Looking at where defects can occur, there are three main places:

- Inside the devices,
- At the external bonding points, and
- Along the external interconnects themselves.

1149.6 addresses opens and shorts at the external bonding points and along the external interconnects. At this point, care must be taken. Some differential systems are designed to be fault tolerant. That is, if there is an open on one of the two legs, for example, the receiver will still produce a stable logic value, usually logic-1. Similarly, a direct short between the positive leg and the negative leg will also produce a stable result. Such defects may or may not be detected by low-frequency testing of the differential interconnect and may dictate a higher-frequency test strategy. As we will see, 1149.6 is not a high-frequency test strategy and although there are solutions to the need for high-speed testing on differential connections, these solutions are outside the scope of this tutorial.

Options for AC-Coupling Test

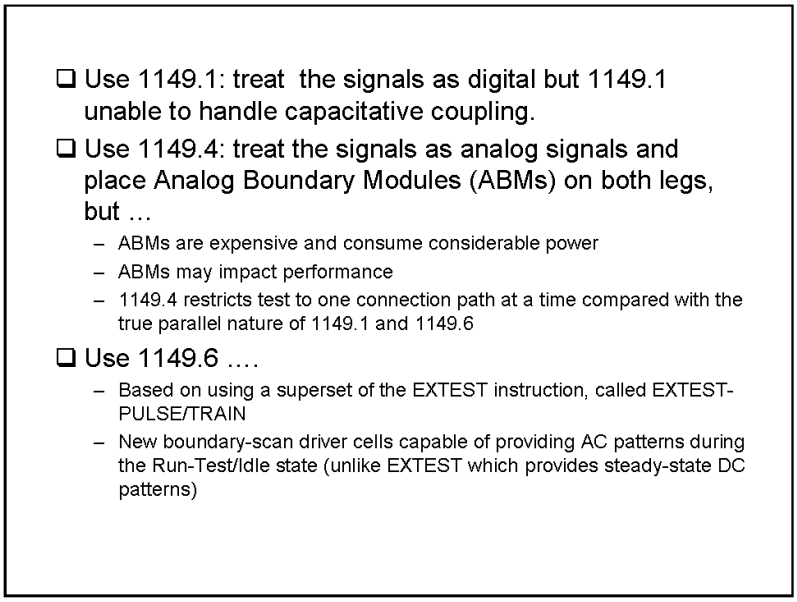


Figure 53: Options for Testing AC-Coupled Interconnects

So, what are the options for testing DC and AC-coupled differential interconnects?

Clearly, 1149.1 will not work for AC-coupled differential interconnects. It may work for DC-coupled differential interconnects but the defect coverage would be low as pointed out earlier.

We could look at a solution based on the use of 1149.4 Analog Boundary Modules attached to the individual legs of the interconnects but the biggest objection to this solution would probably be the potential impact on mission-mode performance at gigabits-per-second speeds of operation. In all likelihood, no one has tried using 1149.4 ABMs in this context.

So that leads us to the 1149.6 solution. In fact, there are other solutions as well, such as, some form of device-internal built-in self-test that drives stimulus and senses and checks responses between devices on a board, such as Intel's recently announced IBIST technique. In this tutorial however, I will focus only on the 1149.6 solution.

Another note of caution. 1149.6 was approved in March 2003. This is relatively recent, and, like any new standard, there may be initial problems with its adoption. To get the latest information on this standard, contact the 1149.6 experts at ASSET InterTech.

IEEE 1149.6 Basic Architecture

The objective of IEEE 1149.6 is to solve test problems associated with *Advanced I/O (AIO)* connections. These are defined as connections based on AC-coupled 2-wire differential signaling. But, the approach must also work with DC-coupled pairs. Testing an AC-coupled line requires a time-varying signal as opposed to a static time-invariant signal. As we will see, this is the main feature that differentiates an 1149.6 solution from an 1149.1 solution.

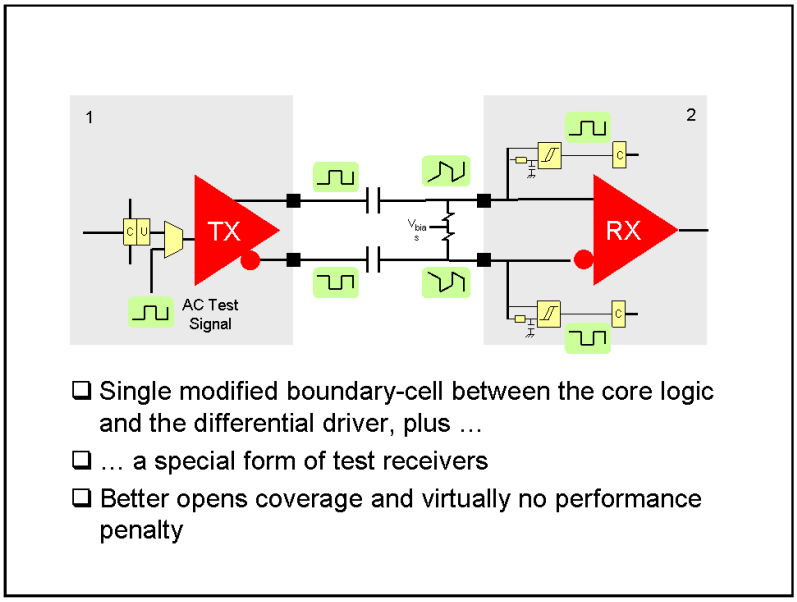


Figure 54: Interconnect Test: IEEE 1149.6 Solution

The new 1149.6 Standard calls for a modified boundary-scan cell upstream of the differential driver. This cell is capable of delivering a single pulse or a train of pulses to the individual legs of the interconnect pair. The standard also calls for a special form of test receiver on each receiving leg. These receivers are capable of determining whether the AC-signal coming in is rising or falling. A rising edge indicates that this leg is positive with respect to the other leg, whereas a falling edge indicates that this leg is negative with respect to the other leg. In this way, the incoming AC-signal is digitized, and the result is captured in normal 1149.1 boundary-scan cells placed at the outputs of the test receivers. Under normal circumstances, the values captured in the two-receiver scan-cells should always be complementary.

The bottom line is that the 1149.6 Standard affords much better fault coverage, particularly of opens, and there is no impact on mission-mode performance.

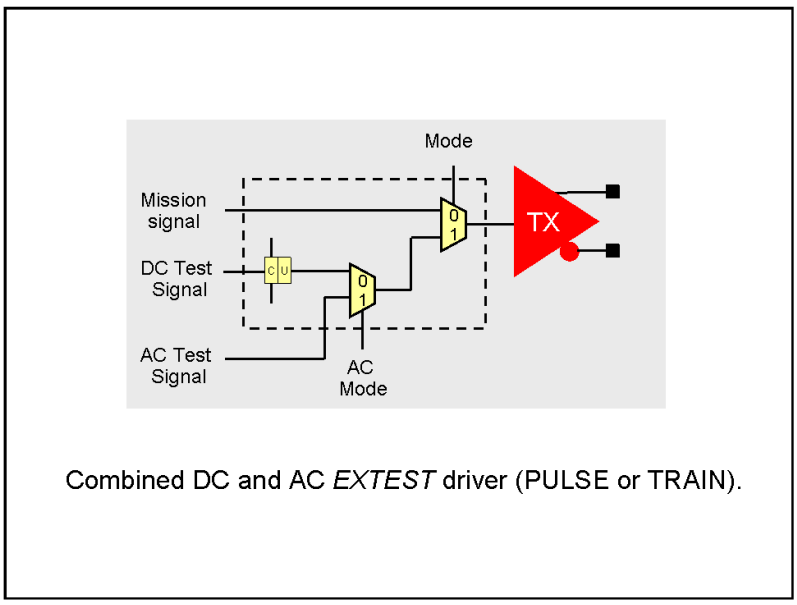


Figure 55: Modified TX Boundary-Scan Cell

Here we see more detail of the modified driver-boundary-scan cell. Essentially, multiplexers are used to route a single AC pulse or train of pulses into the transmitter driver and away to the differential signal legs. Note that the actual signal into the driver can be either the AC test-signal, or a DC test-signal coming from the boundary-scan cell, or the mission-mode signal. In this way, such a boundary-scan cell will support standard 1149.1 ***EXTEST*** operations as well as the new 1149.6 requirement to drive a pulse. This is an important consideration for a board that contains a mix of 1149.1 and 1149.6 devices.

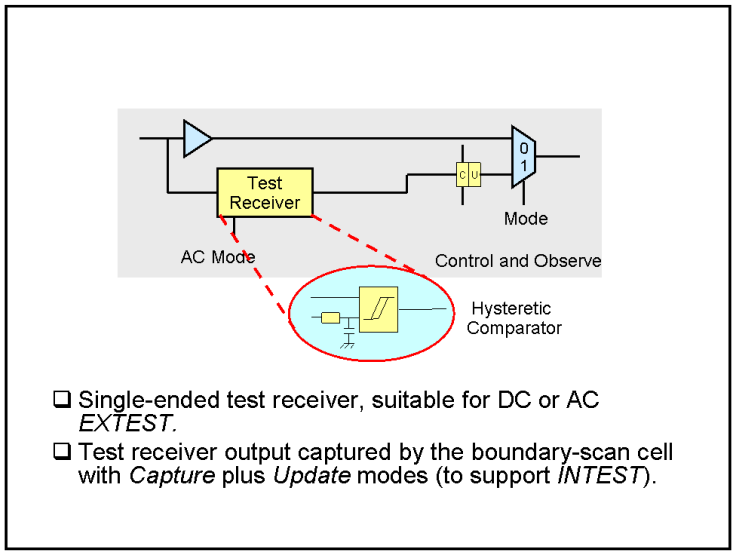


Figure 56: 1149.6 Test Receiver

At the receiver end, the AC signal is fed into what is called a *hysteretic comparator*, which compares the received signal with a delayed version of itself in order to determine the rising or falling nature of the signal. The output – a logic 1 if rising; a logic 0 if falling – is passed to the downstream boundary-scan cell where it can be captured and scanned out in the normal way.

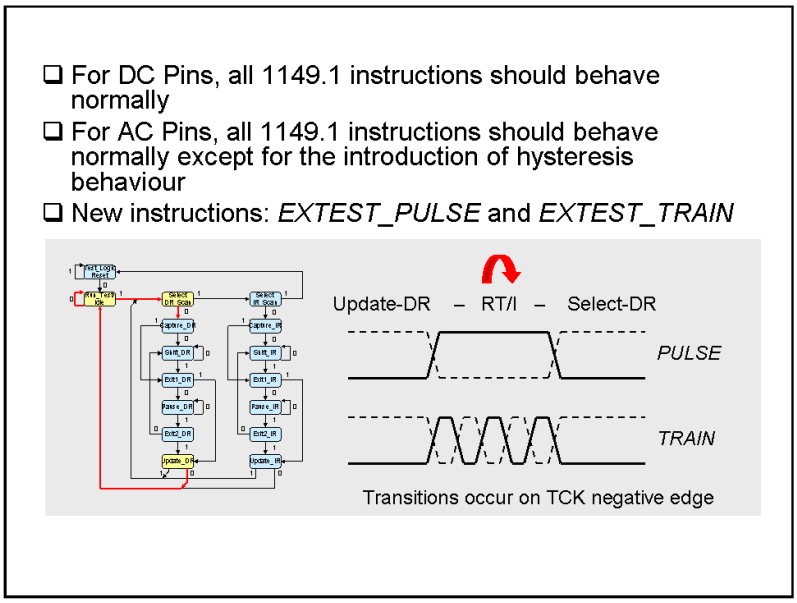


Figure 57: New 1149.6 Instructions

The 1149.6 Standard calls for two new mandatory instructions to support the new architectural features: ***EXTEST_PULSE*** and ***EXTEST_TRAIN***

As the names suggest, ***EXTEST_PULSE*** generates a single pulse by entering and exiting the *Run-Test/Idle* state of the 1149.1 TAP controller; whereas ***EXTEST_TRAIN*** generates a stream of pulses while in the *Run-Test/Idle* state. In case you are curious, the BSDL file for an 1149.6 device will specify the minimum number of pulses and the maximum time period allowed for pulse generation in the *Run-Test/Idle* state. In this way, designers can control the number of pulses that are generated by controlling the frequency of TCK.

Conclusions

The 1149.6 Standard was created and approved within a two-year period. This is very fast for an IEEE standard and essentially demonstrates that there was a real test issue associated with AC-coupled differential interconnects and a willingness to produce a general solution which was not necessarily the

same as any earlier home-spun solutions. Already, companies such as Agilent Technologies, Cisco, National Semiconductors and PLX Technologies have 1149.6-compliant devices and ASSET InterTech has updated its *ScanWorks*® product to accommodate such devices. In addition, several EDA test-synthesis companies have announced 1149.6 insertion tools so the adoption of the standard is off to a good start.

To Probe Further …

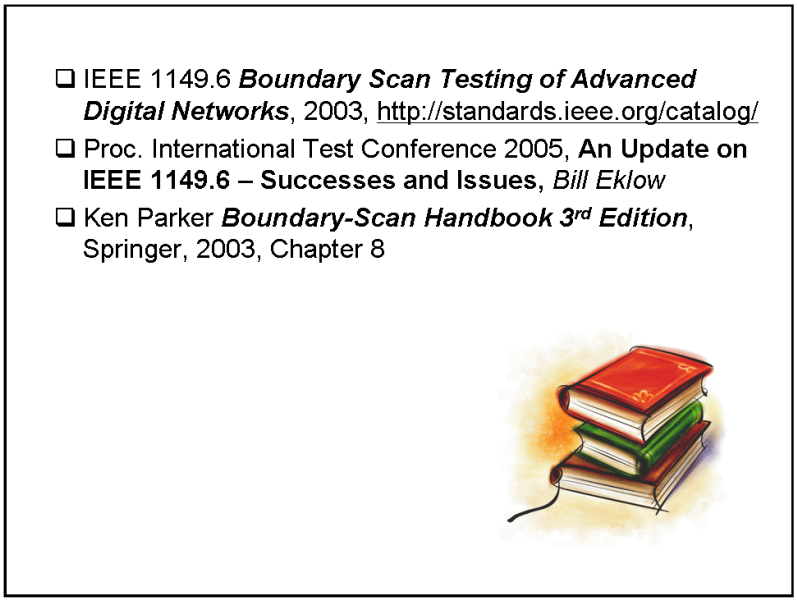


Figure 58: IEEE 1149.6 - To Probe Further

Chapter 8: DFT Boundary-Scan Guidelines for Devices and Boards

This chapter discusses the reasons for Design-For-Test (DFT) guidelines and summarize the more important rules to follow. At the end of the chapter, information will be provided on where to find a more detailed list of DFT guidelines.

Why Do We Need DFT Guidelines?

We should start with the basic question – why do we need DFT guidelines if the on-board digital devices already contain boundary scan?

The answer is that it is easy to design the board such that the board test engineer is not able to take advantage of all the boundary-scan features, or that certain optional boundary-scan features that could

have been designed in are missing. DFT guidelines are intended to act as a checklist for both chip-level designers responsible for inserting boundary scan inside their devices and board-level designers, both electrical and layout, who are responsible for making sure all the boundary-scan features are not only usable but efficiently usable.

Chip-Level DFT Guidelines

Let's explore further, starting with the chip-level designers.

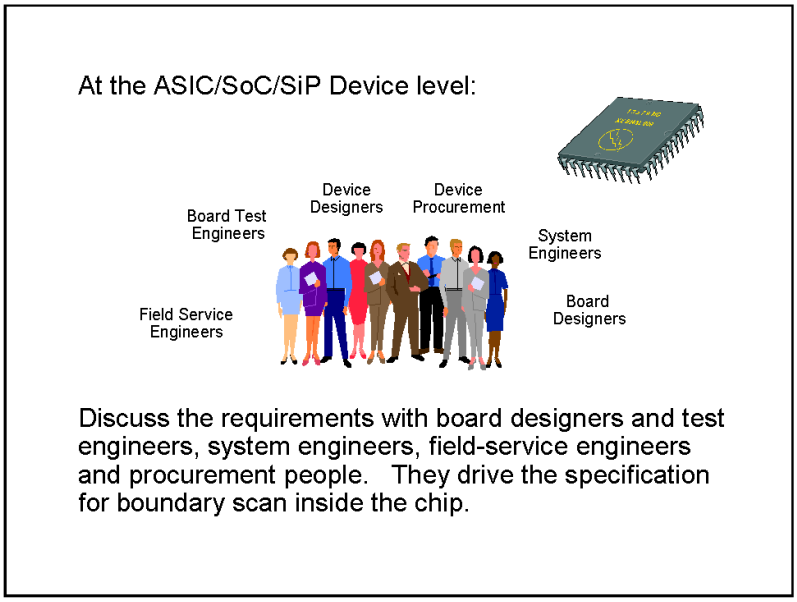


Figure 59: ASIC/SoC/SiP Chip-Level DFT Guidelines

First and foremost, we should ask "Who drives the specification for boundary-scan features inside new Application-Specific ICs (ASICs), System-on-Chip (SoC) and System-in-Package (SiP) devices?" The answer is "all those who will benefit from the use of boundary scan" – that is, prototype board debug engineers, board test programmers, system integrators, and field service repair engineers. It's also worth adding device procurement people to this group. This makes such people conscious of the need to buy 1149.1-compliant devices. Consequently, the correct way to specify the 1149.1 features in a new ASIC, SoC or SiP is for all these people to sit down and decide the product life-time strategy for testing the boards. Each engineer will have a different view of the requirements specification and, collectively, the group can try to come up with a consensus view of the overall requirement, but note that obtaining a consensus view can be difficult!

Once specified, the requirement can be presented to the chip designer in the form of a BSDL file. Some 1149.1 chip-level boundary-scan synthesis programs can accept BSDL as an input specification. Check with your EDA vendor.

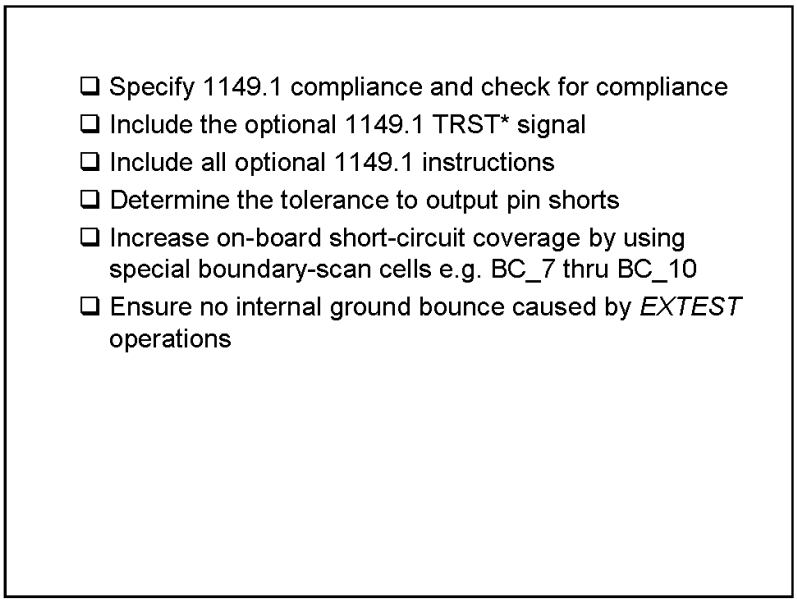


Figure 60: Chip-Level DFT Guidelines: Some Examples

In terms of the chip-level guidelines themselves, the list above illustrates the sort of things that should be considered by those responsible for adding boundary scan logic to devices.

Working through the list:

Specify 1149.1 compliance and check for compliance.

It is very important that boundary-scan devices not only comply to the 1149.1 Standard but that they have been checked by a reputable compliance checker such as ASSET's **BSDL Validation Services**.

Include the optional 1149.1 TRST* signal.

This signal is optional but adding it to the device simplifies board initialization.

Include all optional 1149.1 instructions.

There are six optional instructions. They all have value at board test. Check the requirements with the board test engineers.

Determine the tolerance to output pin shorts.

It is ironic that the only way to use boundary-scan to find pin-to-pin shorts at the board level is by deliberately creating contention on two driver scan cells i.e. causing one driver to drive a logic 1 and the other to drive a logic 0. If the short is present, then we assume that it is either a strong-1 weak-0 (Wired-OR) short or weak-1 strong-0 (Wired-AND) short. Such a test places a stress on the output drivers since one output will dominate, forcing the other to its opposite logic value and possibly damaging the output drive amplifier of the weaker signal. You will need to determine the vulnerability of the output drivers to such damage.

Increase on-board short-circuit coverage by using special self-sensing boundary-scan cells, such as BC_7s thru BC-10s

Replacing BC_1s with self-sensing boundary-scan cells (BC_7s thru BC_10s) will increase short-circuit detection between boundary-scan and non-boundary-scan devices on a board.

Ensure no internal ground bounce caused by *EXTEST* operations.

The 1149.1 Standard mandates no ground bounce inside the device caused by any boundary-scan activity and so the device should be OK but it is better to check to be sure.

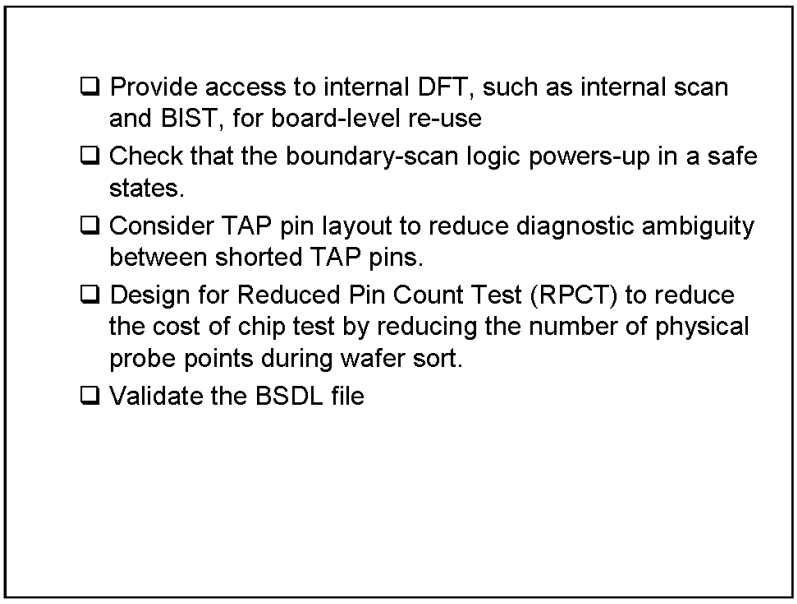


Figure 61: Chip-Level DFT Guidelines - Some More Examples

This figure contains yet more chip-level guidelines. Space does not permit all the detail, but these guidelines are included here for completeness.

Board-Level DFT Guidelines

Let's move on to the board-level guidelines.

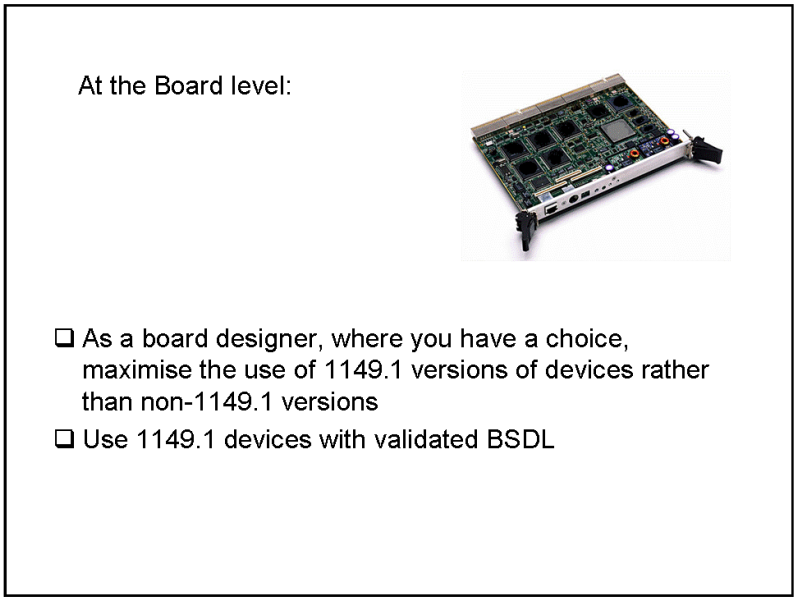


Figure 62: Board-Level DFT Guidelines

First, as a board designer you should maximize the use of 1149.1-compliant devices. The more boundary-scan register access you have, the more fault-coverage will be obtained, both between boundary-scan and boundary-scan devices and between boundary-scan and non-boundary-scan devices.

You should also make sure that the device BSDL files have been validated and that all non-compliant features, if they exist, are documented in the *Design_Warning* section of the BSDL files.

Compliance-enable pins are also available on certain CPLDs. We will discuss what to do with these pins later.

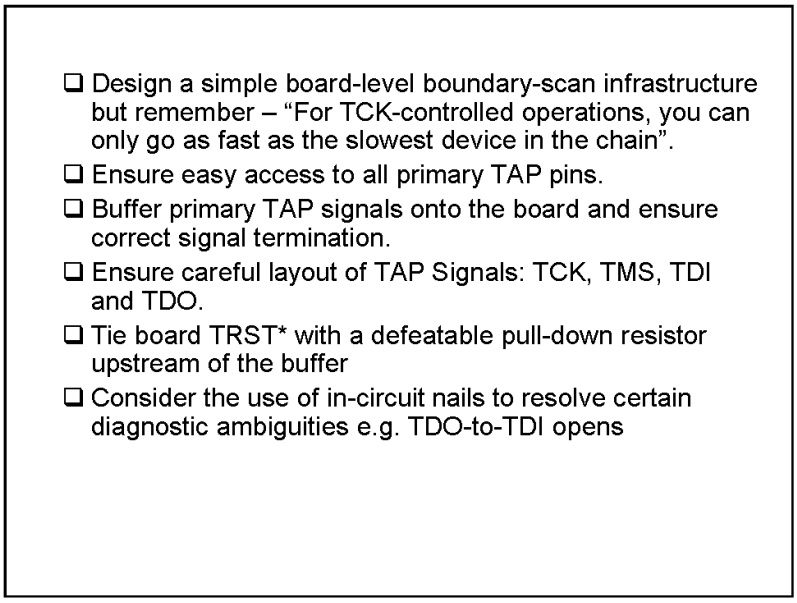


Figure 63: Board-Level DFT Guidelines - Some Examples

Here is additional information on each of the guidelines listed in the figure above.

Design a simple board-level boundary-scan infrastructure.

The distribution of TMS and TCK can be simple i.e. broadcast simultaneously to every boundary-scan device, or complex – multiple TMS controls for example. Also, a single chain of devices can be designed to be dynamically reconfigurable into a number of smaller isolated chains to aid diagnosis. And, remember: "You can only go as fast as the slowest device in the chain". The basic rule for infrastructure design is "keep it simple" unless you can see a reason to design a more complex chain of boundary-scan devices.

Ensure easy access to all primary TAP pins.

Essentially, make sure that all primary TAP signals on the board – TDI-in, TDO-out, TMS, TCK and TRST* (if present) – are easily accessed by the tester, such as from the edge connector or via a plug and socket arrangement.

Buffer primary TAP signals onto the board.

Primary TAP signals require buffering onto the board. Use simple line-driver buffer devices. This guideline also includes advice on primary TAP signal terminations, e.g. 10KΩ pull-ups on TDI-In, TMS and TCK upstream of their buffers, and proper termination on TDO-Out.

Ensure careful layout of TAP Signals: TCK, TMS, TDI and TDO.

TCK and TMS especially need special care during board layout. They are master boundary-scan control signals and on big boards they can be connected to many different devices. So, it is important to remove any signal skew.

Tie board TRST* with a defeatable pull-down resistor upstream of the buffer.

Once the board is operating normally, all boundary-scan logic inside on-board devices should be in a passive state – that is, the *Test-Logic Reset* state. Tying TRST* low ensures this for all devices containing the optional TRST* control pin.

Consider the use of in-circuit nails to resolve certain diagnostic ambiguities like TDO-to-TDI opens.

Boundary scan does not eliminate the need for physical probes. Quite the contrary, physical probes from an in-circuit or flying probe tester, for example, can be used intelligently to supplement the control and observable characteristics of the non-boundary-scan devices on the board. More on this later.

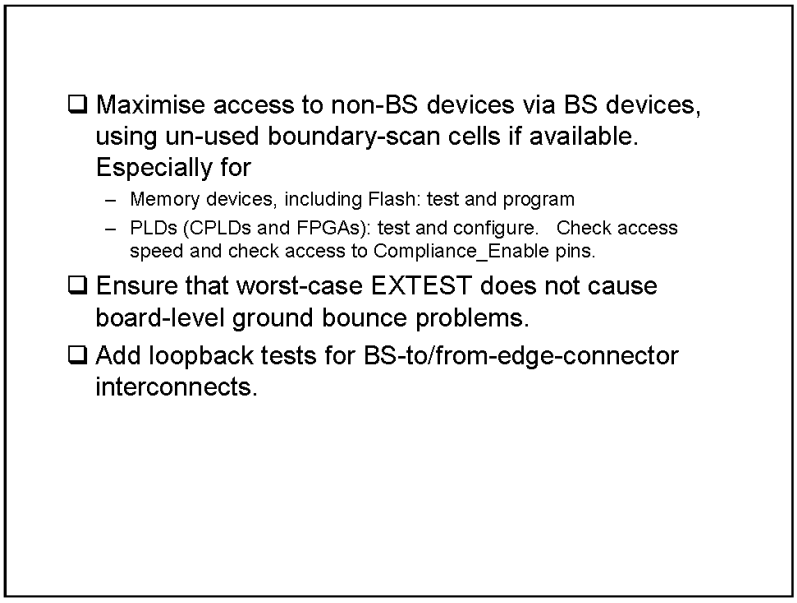


Figure 64: Board-Level DFT Guidelines - Some More Examples

Here are some more guidelines, for completeness. Note the first item in the list above for dealing with access to RAMs, Flash, CPLDs and FPGAs for both test and in-system configuration. Using boundary scan to configure CPLDs and FPGAs once these devices are mounted on a board is a major new application of boundary scan. Similarly, loading data into on-board flash devices is also relatively new. Generally, these applications are known as *In-System Configuration* or *In-System Programming*.: see the chapter on the IEEE 1532 *In-System Configuration* Standard for more detail.

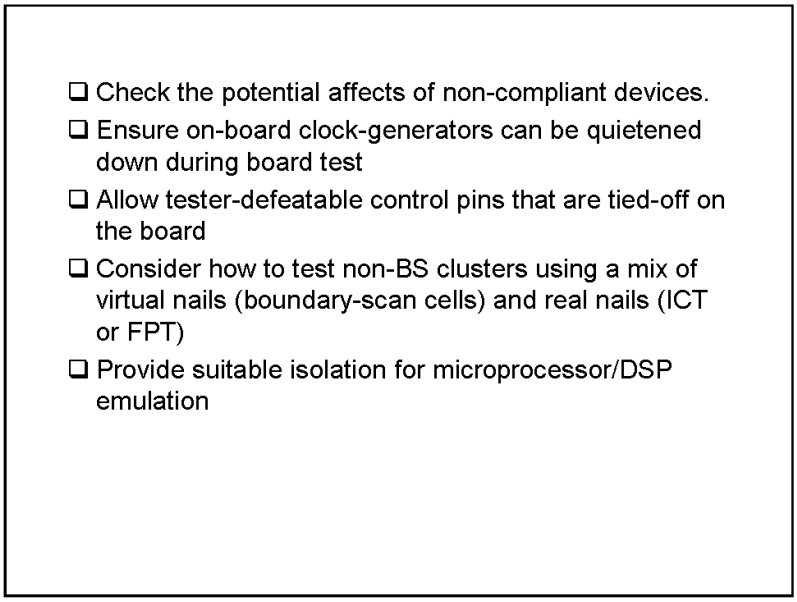


Figure 65: Board-Level DFT Guidelines - Yet More Examples

This should complete the list of board-level DFT guidelines. Of special note here is the fourth item reinforcing earlier comments on combining the virtual access of boundary scan with the physical access, if available, of in-circuit or flying-probe test.

Chapter 9: Boundary-Scan Tools

To complete this tutorial, we will discuss the software tools that are required 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 the fixturing requirements of boundary-scan designs are fairly simple and consistent from one design to the next, you can now use the same toolset during all phases of the product life cycle. However, the toolset must offer features to meet your specific needs during each phase of the product's life cycle.

The toolset you choose should meet the needs of all major phases in 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 of 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 examines a product and ensures 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. To complete structural and functional testing requires creating tests.

Manufacturing Test

The goals of this process are to determine if any errors were made in the manufacturing process and if the unit under test (UUT) 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 that are 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 to identify 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-good 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. For example, a PASS/FAIL message could be provided 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 for structural and functional tests in order to complete the testing of the unit. A concern arises because sometimes the test operator must use several different test tools, each of which is 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. 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 diagnose and repair 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, affect the necessary repairs to that product. During the normal manufacturing process failed units are diagnosed in order to affect 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 to determine that the product is faulty. After this examination, if the defect cannot be determined, the diagnostic engineer usually re-runs 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 typically execute additional previously created tests or tests specifically created during diagnosis to 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. In addition, the diagnostic engineer may have information to help pinpoint where that defect is.

Unlike the design debug process, the 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 quickly determine what product or part of a product in end-customer use is faulty and replace it. 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. Moreover, he may want to run additional tests or interactive applications in

order to further isolate the defective unit. Also, this testing should be done with no or only 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 accessed Built-In Self-Test (BIST) capabilities present in devices in the system
- Integrated 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
- Price commensurate with performance because the toolset runs 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 prototype boards and systems in a manufacturing environment. To support this type of analysis, the boundary-scan tools must have the same kinds of tools as traditionally found in a manufacturing test environment. A more complete description of these capabilities is included in the Manufacturing Test section, but, in general, the following capabilities are required:

- 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 in common with 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 include the following:

- 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, accepting as inputs 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. As

well, the tool should 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 is achieved through Component Actions. 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 of the registers and pins on 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 the execution of BIST capabilities in 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 for developing custom test suites, integrating test suites with other test tools and executives, or building 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-based board test application systems that support multiple possible hardware interfaces such as USB, PCIe, ethernet and PXI with the ability to integrate boundary-scan controlled parallel I/O modules. These solutions assist with 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 application of debug and diagnostics capabilities on portable computing platforms, such as the parallel printer port or PCMCIA card on a laptop computer. 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 10: Recent Developments

Since the first publication of this booklet in 2000, there have been some significant developments in the world of boundary scan technology. This chapter takes a look at these developments.

IEEE 1687 (IJTAG) Initiative

For some years now, enterprising device designers have been using the 1149.1 Test-Access Port (TAP) and Instruction Register system to access and make use of device-internal registers for device design-debug and test purposes. Early examples include access to the control registers of memory-BIST engines and logic-BIST seed setup and signature registers. More-recently, we have seen the development of TAP access to embedded signal-conditioning instrumentation prior to display on an external logic-analyzer or oscilloscope.

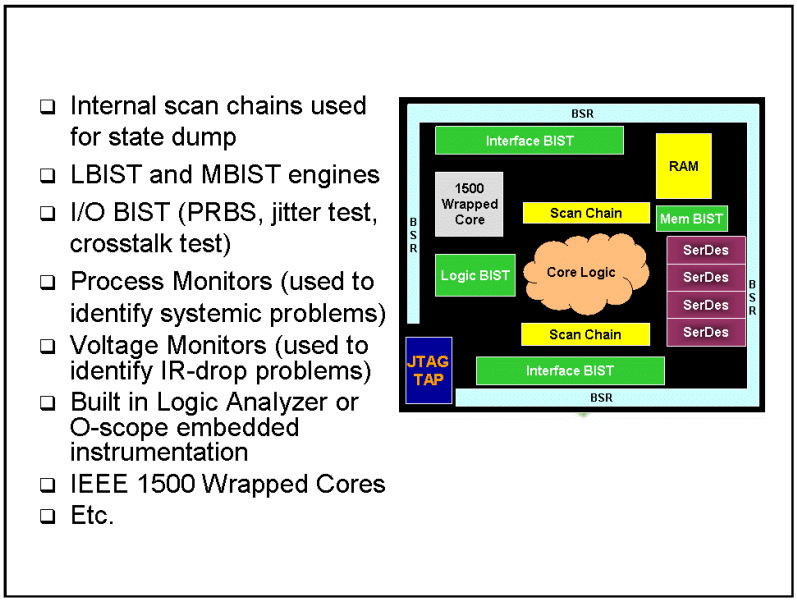


Figure 66: TAP Access to Embedded Instruments

Many other examples abound and in 2004 at the IEEE International Test Conference, several interested parties came together to discuss the need for a new standardization effort. The result was the Internal JTAG (IJTAG) initiative. Basically, the intent of IJTAG is to standardize the way an external system could communicate with any form of internal design debug and test instruments via the 1149.1 TAP. Here, the word *instrument* is used to mean any on-chip circuit for test, debug, diagnosis, monitoring, characterization, configuration, or functional use.

The IJTAG working group is now on its way to creating a new IEEE standard, 1687. The permission to do this was approved by the IEEE Standards Authority in March 2006. To quote from the proposal form:

"This Standard will develop a methodology for access to embedded test and debug features, (but not the *design of the* features themselves) via the IEEE 1149.1 Test Access Port (TAP) and additional signals that may be required. The elements of the methodology include a description language for the characteristics of the features and for communication with the features, and requirements for interfacing to the features"

In simpler words:

1687 enables a solution to access, configure, control and gather results from embedded on-chip instruments (e.g. Internal Scan, BIST engines, IEEE 1500 Wrappers, Test-Data Compressors, Design Debug Logic, etc.) using an 1149.1 TAP Controller as the main gateway to and from the chip. 1687 will

define the instrument access path, the instrument access protocol, and maybe the instrument interface port – but 1687 will not dictate, modify, or define the instrument.

At the time of this writing, the 1687 Working group was finalizing a definition of a standard hardware interface between the top-level 1149.1 TAP and a range of lower-level instruments. To find out the latest status, visit http://grouper.ieee.org/groups/1687/.

System JTAG (SJTAG) Initiative

Since the early 1990s, companies such as Lucent Technologies (formerly AT&T) and Ericsson have been looking to extend the use of single-board boundary-scan infrastructures into the multi-board (system) domain. Initially, this was to verify that:

- each board is present, is the correct board and is correctly inserted into the motherboard socket, and
- each board-to-board interconnect on the motherboard system backplane is validated i.e. a continuity test to detect any unwanted opens or shorts.

But, eventually, the system companies realized that there was much more that could be done with system-level JTAG, such as:

- in-system configuration and re-configuration of on-board Programmable Logic Devices such as CPLDs, FPGAs and Flash devices;
- re-use of device-internal design-debug and test instrumentation to support diagnosis down to the smallest replaceable unit, especially in field-service and repair depot environments;
- access to embedded board-level BIST routines;
- as a way of reducing the number of No-Trouble-Found boards shipped back to a repair depot.

In support of these requirements, various forms of system-level boundary-scan architectures were proposed – ring (daisy chain), star (radial) and multi-drop – and both National Semiconductor and Texas Instruments produced special system JTAG support devices – ScanBridge and Linking Addressable Scan Port respectively. Tool vendors, such as ASSET InterTech, extended their tools to accommodate the new system-level applications of boundary scan.

Unfortunately, there is no common standard for describing the architecture of the system-level boundary-scan infrastructure. Moreover, there is no JTAG Test Command and Data Language capable of covering the following basic requirements:

- **Represent** embedded test data (e.g. vectors) efficiently
- **Write** to, **Read** from and **Manage** test data stored on the board
- **Run** an embedded test
- **Configure** and **Validate** an on-board PLD
- **Capture** the result of a test and **Compare** with the expected result
- **Log** test execution details (time, date, result, etc)
- **Send** specific test reports and service logs to an internal or external Test Manager.

The purpose of SJTAG is to investigate solutions to these questions.

At the time of this writing, the leadership of SJTAG has just changed hands and attention is focused on the wording of a new standard request to the IEEE, defining the various use scenarios, determining the necessary primitives for the control of SJTAG applications via board-level primary TAPs and identifying the commands to define the test steps and associated test application levels.

Boundary Scan and its Relationship with other Test Techniques

As the adoption of boundary scan has grown, questions have been asked about its relationship with other more-established test techniques, notably any form of In-Circuit Test (ICT) based on physical nail access and microprocessor emulation techniques.

Considering nail-based techniques, what became clear is that ICT was and still is one of the main test techniques used by the Electronic Manufacturing Services (EMS) companies and that such companies were very keen to combine the power of boundary-scan test with in-circuit test. Boundary scan had found its main application in prototype board debug in laboratories where access to an ICT was not possible. In contrast, production volume test in EMS companies was mostly based on ICT techniques. Consequently, the boundary-scan vendors, such as ASSET InterTech, and EMS companies looked at the synergistic relationship between these two test technologies. This study was largely driven by the systems company customers of the EMSs and resulted in an integration of the two techniques. A test program developed

solely for use on a boundary-scan tester was integrated into an ICT environment where it could be enhanced by the additional physical access, especially into the non-boundary-scan clusters on the board.

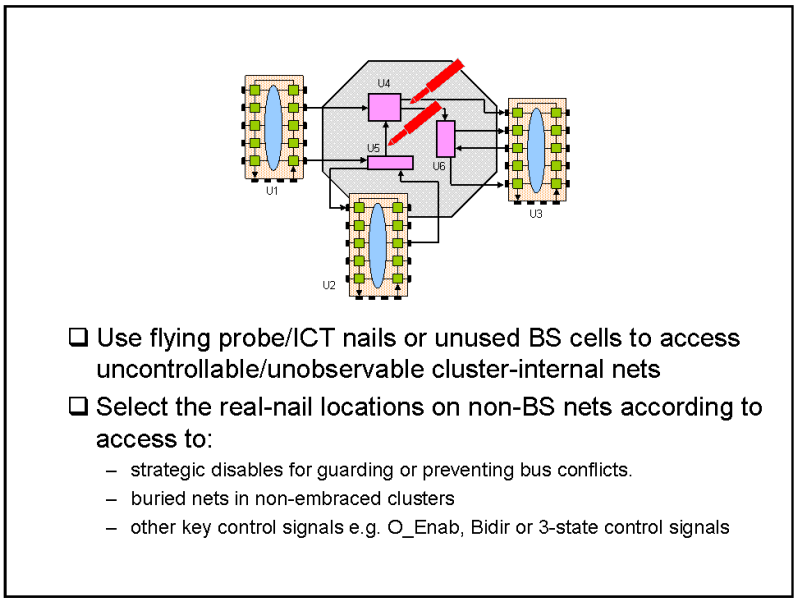


Figure 67: Accessing a Non-Boundary-Scan Cluster Using Nails

But, it is not as easy as it sounds. Just wheeling a boundary-scan tester up against an ICT will not work. There are hardware and software integration issues to work through. Successful commercial solutions are truly integrated in all respects; namely, a single common controller card, common database and DFT guidelines that take advantage of both test technologies.

Turning to emulation, we should first define the type and purpose of board test via emulation.

Traditional CPU emulators replace the board's processor and take control over all processor buses. Read/write access to all parts of the board's memory and I/O are available via the emulator. However, CPU emulators are not viable for higher speed processors (>20-30MHz). So, CPU manufacturers added debug commands to allow their CPUs to be controlled in the same way as an emulator, typically via the 1149.1 TAP.

CPU debug ports use the standard 1149.1 TAP plus a number of additional control lines, including Reset, Power, etc. Testers that use debug ports to control CPUs are still described as CPU Emulators for historical reasons.

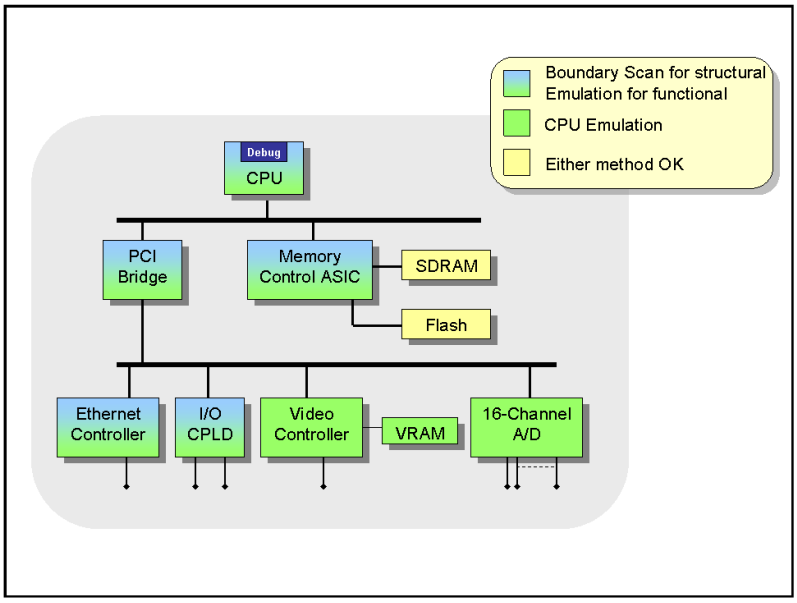


Figure 68: Boundary Scan and Emulation Dual-Approach Testing

CPU emulation is traditionally used for functional checkout of boards but it can also be used for structural checkout. In the diagram above, assume that the CPU, PCI Bridge, Memory Control ASIC, Ethernet Controller and IO CPLD all have boundary scan included in their design. Assume that all the other devices do not have boundary scan. In a traditional boundary-scan only set of structural tests, it would be difficult to test the non-boundary-scan devices using the boundary-scan features on the board. These devices would have to be considered as clusters and we could try to test them for presence, orientation and bonding defects (a structural test) using the boundary-scan registers of the boundary-scan devices. But, the cluster models would be complex and there would be no guarantee of success.

An alternative, therefore, is to use the external CPU Emulator to access the non-boundary-scan devices and carry out simple structural tests.

In summary, we should use boundary scan where possible as vectors and diagnostics can be automatically generated and diagnostics are more specific. But, we can use CPU emulation for testing non-boundary scan parts. Test generation will be semi-automated, and diagnostic resolutions will be to functional misbehavior rather than to a component or interconnect level. In addition, CPU emulation can provide functional test coverage at full speed to both boundary scan and non-boundary scan parts. Memory testing is possible with either method, but only interconnect testing is possible with boundary scan.

This approach has been proven in the industry such as ASSET InterTech's ScanWorks® product Processor-based Test Control (PCT).

Other New Standard Developments

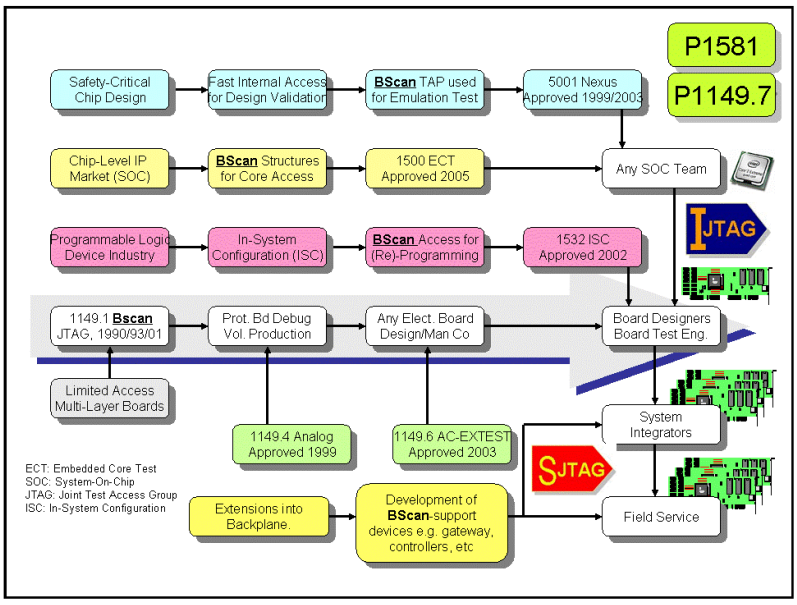


Figure 69: The Explosive Growth of Boundary-Scan Technology

Figure 71 summarizes the explosive growth of boundary-scan technology since the publication of the first version of the IEEE 1149.1 standard in 1990. As we see, boundary-scan has not only expanded at the single board level (IEEE 1149.4-1999, 1532-2002 and 1149.6-2003) but it has also moved backward into core-based designs (IEEE 1500-2005 and 5001-2003 and now 1687/IJTAG) and forward into multi-board system designs (under investigation by the SJTAG Working Group). Additionally, two new standards, IEEE 1581 and 1149.7, are underway. 1581 is targeted at making non-boundary-scan clusters easier to test from a boundary-scan register in an 1149.1-compliant device and 1149.7, which is also known as compact JTAG (cJTAG) is targeted at reducing the standard 4-wire 1149.1 interface down to just two wires.

To read more on all these activities, visit the following web sites.

IEEE 1149.1-2013, https://standards.ieee.org/ieee/1149.1/4484/

IEEE 1149.4, https://standards.ieee.org/ieee/1149.4/4022

IEEE 1149.6 https://standards.ieee.org/ieee/1149.6/4706/

IEEE 1149.7, cJTAG: https://standards.ieee.org/ieee/1149.7/3936/

IEEE 1149.10, https://standards.ieee.org/ieee/1149.10/5786/

IEEE 1500, https://standards.ieee.org/ieee/1500/2238/

IEEE 1532, https://standards.ieee.org/ieee/1532/3366/

IEEE 1581, https://standards.ieee.org/ieee/1581/4212/

IEEE 1687, IJTAG, https://standards.ieee.org/ieee/1687/3931/

IEEE 1838, https://standards.ieee.org/ieee/1838/5073/

IEEE 5001, Nexus, http://www.nexus5001.org/

SJTAG, http://sjtag.org/ copies of approved IEEE Standards can be obtained from:

http://standards.ieee.org/ or http://www.techstreet.com/info/ieee.html

http://www.techstreet.com/info/ieee.html or http://global.ihs.com/

http://global.ihs.com/

Chapter 11: 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," Springer Press, 2003, Third Edition (Contains chapters on BSDL, DFT guidelines, IEEE 1532, IEEE 1149.6)
2. *IEEE Standard for Test Access Port and Boundary-Scan Architecture,* IEEE Standard 1149.1-2013, 2013

Reference

1. BSDL/IEEE 1149.1 verification service, maintained by ASSET InterTech. See http://www.asset-intertech.com/
2. Latest issues of: IEEE International Test Conference Proceedings and IEEE Design & Test of Computers magazine