

## Introduction

This chapter provides an overview of the tools available in the Quartus® II software and the Nios® II Embedded Design Suite (EDS) that you can use to verify and bring up your embedded system.

This chapter covers the following topics:

- Verification Methods
- Board Bring-up
- System Verification

## Verification Methods

Embedded systems can be difficult to debug because they have limited memory and I/O and consist of a mixture of hardware and software components. Altera® provides the following tools and strategies to help you overcome these difficulties:

- FS2 Console
- System Console
- SignalTap II Embedded Logic Analyzer
- External Instrumentation
- Stimuli Generation

## Prerequisites

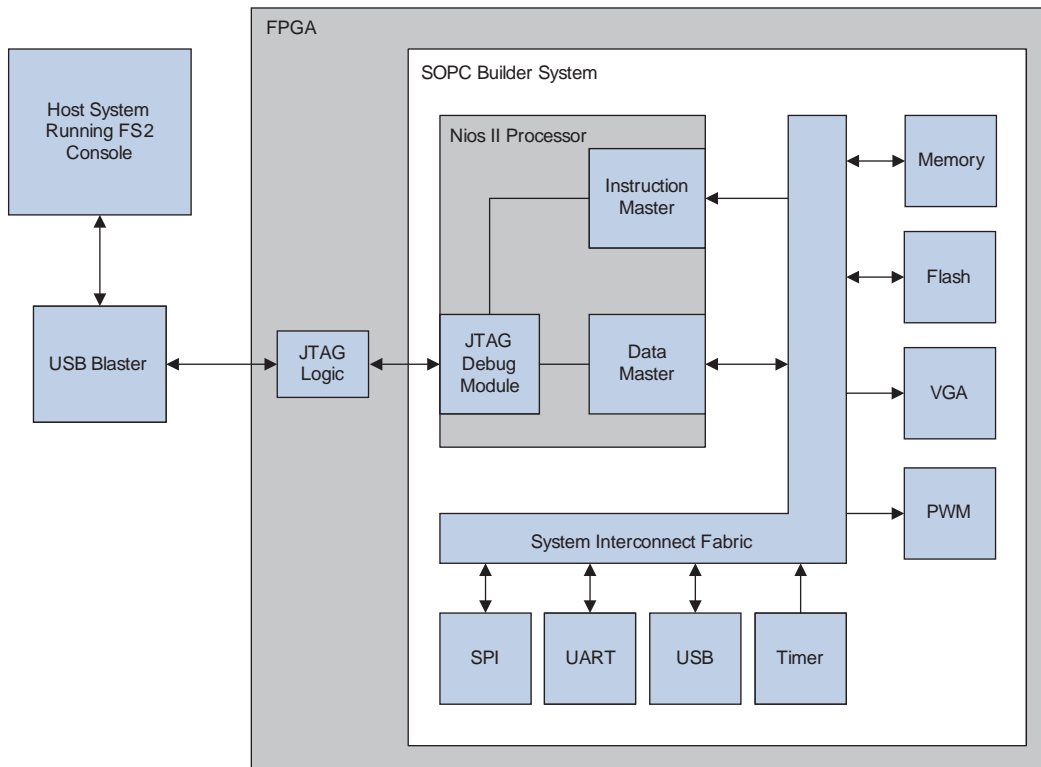
To make effective use of this chapter, you should be familiar with the following topics:

- Defining and generating Nios II hardware systems with SOPC Builder
- Compiling Nios II hardware systems with the Quartus II development software

## FS2 Console

The FS2 console, developed by First Silicon Solutions (FS2) extends the verification functionality of the Nios II processor. The FS2 console communicates with the Nios II JTAG debug module that is available for all three variants of the Nios II processor. FS2 optionally uses an external system analyzer hardware module that creates additional trace support to the Nios II JTAG debug module. Figure 9-1 illustrates the connectivity between an FS2 console and an SOPC Builder system.

**Figure 9-1. FS2 Console Communication Path**



The Nios II JTAG debug module uses the Nios II data master port to communicate with components that contain Avalon<sup>®</sup> Memory-Mapped (Avalon-MM) slave ports. Although the Nios II JTAG debug module is tightly integrated with the Nios II processor, it does not rely on any additional support being provided by the processor. As a result, you can use the Nios II JTAG debug module and the FS2 console to verify a system without having to write software.



The FS2 console does not support host machines running the Linux operating system, and is not compatible with the Software Build Tools from Eclipse.

### SOPC Builder Test Integration

Even if you do not intend to include a Nios II processor or an SOPC Builder system in your final design, you can still include a Nios II processor during the debug phase to take advantage of the embedded tools that Altera provides. The Nios II processor contains a data master which you can use to perform read and write accesses to your hardware blocks.

To include a JTAG debug module in your system follow these steps:

1. On the **System Contents** tab, double-click the **Nios II Processor** component.
2. In the Nios II Processor wizard, click the **JTAG Debug Module** tab.
3. Make sure **Level 1** is selected.

The JTAG debug module is required for communication between your system and the FS2 console.

## Capabilities of the FS2 Console

You can launch the FS2 console from the Nios II IDE or from the Nios II command shell, but not from the Software Build Tools for Eclipse. After the FS2 console is open, you have access to the command line and scripting capabilities of the software. The command line within the FS2 console is sufficient for lightweight debugging. To access help for FS2, simply type `help` for a list of available commands. The help system is hierarchical. When you type `help`, the help system lists the top-level command hierarchy. You can refine your help searching by typing `help <command_name>` to learn more about the commands available. For example, if you type `help memory` The FS2 console displays a list all of the commands to access memory, including: **addr**, **asm**, **byte**, **compare**, **copy**, **dasm**, **dump**, and so on.

Using the FS2 console you can query the FPGA to determine if there are any Nios II debug modules present. The FS2 console can access a Nios II debug module anywhere on the JTAG chain. Because your design may have multiple Nios II debug modules, you can specify the debug module you prefer.

The Nios II processor has a 32-bit data master. Using the FS2 console, you can perform either byte (`byte`), half word (`half`), or word (`word`) accesses to any Avalon-MM slave port.

The FS2 console supports the Tcl/Tk scripting language. Scripting memory accesses is particularly useful if you have many hardware blocks to test or need to instrument regression testing. A Tcl/Tk reference guide is integrated into the FS2 console help menu.

### sld info Command

The `sld info` command lists the JTAG chains that are available on your board. For the chain that it is being used to access your board, this command provides identifying information for the JTAG cable (`hw`), FPGA device or devices (`device`) and debug modules (`node`). [Figure 9-2](#) shows typical output from this command. In this example, communication occurs over the second JTAG chain, Hw 1: USB-Blaster [USB-0]. There is a single FPGA in this JTAG chain, device 1: EP2C35, and there is a single debug module on this chain, node 0: owner: First Silicon Solutions.

You must specify these components to the FS2 Console using the `config` command. For the JTAG chain illustrated in [Figure 9-2](#), you must type the following three commands:

- `config sldHW 1`—selects the second programming cable
- `config sldDev 1`—selects the second device on the JTAG chain

- `config sldNode 0`—selects the first debug module in the FPGA

**Figure 9-2. sld info Command**

```

13> sld info
aji hw 0: ByteBlaster [LPT1]
      Can't Lock hardware[0]'s device!
aji hw 1: USB-Blaster [USB-0]
      aji device 1: EP2C35
      aji node 0: owner: First Silicon Solutions; ver: 3; Id: 22
      aji node -: owner: Altera Corporation; ver: 1; Id: 80; Instance: 0
14> |

```

You can update this configuration information to communicate over a different JTAG cable to a different device and debug module. For example, to communicate over the fourth programming cable to the third device using the second Nios II debug module, you would type the following commands:

- `config sldHW 3`
- `config sldDev 2`
- `config sldNode 1`

When you first bring up the FS2 Console, it is initialized to communicate over the first cable, to the first device and using the debug module in the first FPGA. However, if you update this information, your changes are persistent.

After you compile your design with the Quartus II software, the JTAG debug interface file (`.jdi`) in your project directory includes the debug module instance numbers. If your design includes two SOPC Builder systems in a single FPGA, the debug module instance numbers may change when you recompile. Each debug module is referenced by its full name and level of hierarchy in the design. The debug module number is stored as `sld_instance_index`. For example, if a debug module is assigned to three, the `.jdi` file includes the following setting:

```
<parameter name="sld_instance_index" type="dec" value="3"/>
```

This is the value you use when setting `sldNode`.

### FS2 Examples

The following procedure writes 0x5A followed by 0xA5 to an 8-bit hardware block:


1. Download the hardware image file SRAM object file (`.sof`).
2. In the Nios II command shell, type `nios2-console` to start the FS2 console.
3. In the FS2 console, type `openport sld` to establish communication with a remote debugger.
4. Type `halt` to stop the Nios II processor.
5. Type `byte 0x00810880 0x5A` to write 0x5A to memory location 0x00810880.
6. Type `byte 0x00810880 0xA5` to write 0xA5 to memory location 0x00810880.

**Example 9–1** writes a repeating pattern to an address range. Before trying this example, check the Base (address) column for a memory device in your SOPC Builder system, so that you write and read valid locations. In this example, an on-chip memory has a base address of 0x02100000.

**Example 9–1. Writing a Repeating Pattern to an Address Range**

```
# write a repeating pattern of 0x5a5a5a5a to an address range
word 0x02100000..0x021000FF 0x5a5a5a5a
# read back the data from the address range
dump 0x02100000..0x021000FF word
```

All the commands sent to the JTAG debug module from the FS2 console use the JTAG interface of the FPGA. JTAG is a relatively slow communication medium. You cannot rely on an FS2 console to stress test your memory interfaces. Refer to “**Board Bring-up**” on page 9–10 for strategies to stress test memory.

 To learn more about the FS2 console refer to the First Silicon Solutions website at [www.fs2.com](http://www.fs2.com). The Nios II Embedded Design Suite (EDS) installation also includes documentation for the FS2 console. You can find this documentation at `<Nios II EDS install path>/bin/fs2/doc`.

## System Console

You can use the System Console to perform low-level debugging of an SOPC Builder system. You access the System Console functionality in command line mode. You can work interactively or run a Tcl script. The System Console prints responses to your commands in the terminal window. To facilitate debugging with the System Console, you can include one of the four SOPC Builder components with interfaces that the System Console can use to send commands and receive data. **Table 9–1** lists these components.

**Table 9–1. SOPC Builder Components for Communication with the System Console (Note 1)**

Component Name	Debugs Components with the Following Interface Types
Nios® II processor with JTAG debug enabled	Components that include an Avalon-MM slave interface. The JTAG debug module can also control the Nios II processor for debug functionality, including starting, stopping, and stepping the processor.
JTAG to Avalon master bridge	Components that include an Avalon-MM slave interface
Avalon Streaming (Avalon-ST) JTAG Interface	Components that include an Avalon-ST interface
JTAG UART	The JTAG UART is an Avalon-MM slave device that can be used in conjunction with the System Console to send and receive byte streams.

**Note to Table 9–1:**

(1) The System Console can also send and receive byte streams from any SLD node, whether it is instantiated in an SOPC Builder component provided by Altera, a custom component, or part of your Quartus II project. However, this approach requires detailed knowledge of the JTAG commands.

The System Console allows you to perform any of the following tasks:

- Access memory and peripherals
- Start or stop a Nios II processor
- Access a Nios II processor register set and step through software

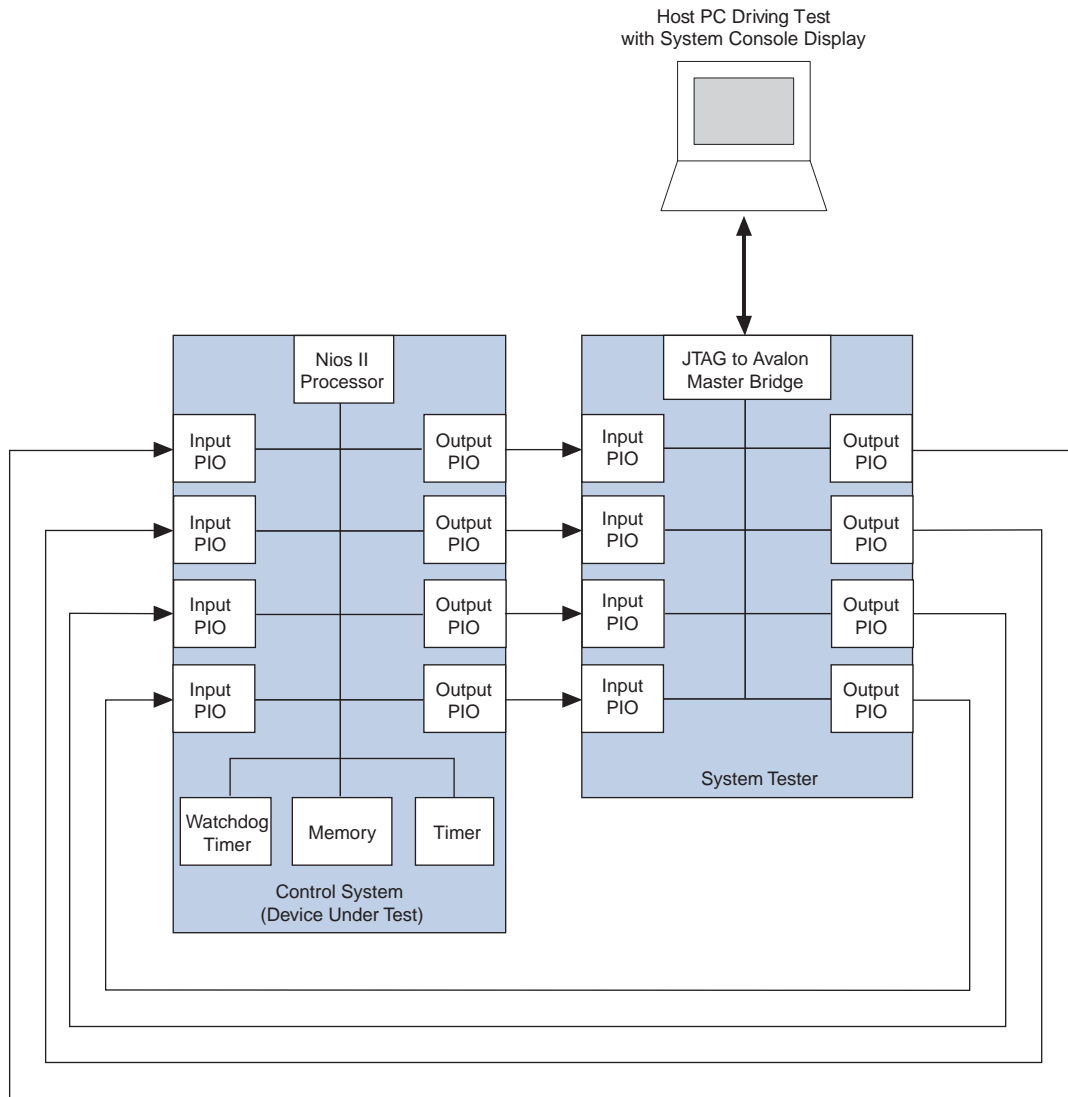
- Verify JTAG connectivity
- Access the reset signal
- Sample the system clock

Using the System Console you can test your own custom components in real hardware without creating a testbench or writing test code for the Nios II processor. By coding a Tcl script to access a component with an Avalon-MM slave port, you create a testbench that abstracts the Avalon-MM master accesses to a higher level. You can use this strategy to quickly test components, I/O, or entire memory-mapped systems.

Embedded control systems typically include inputs such as sensors, outputs such as actuators, and a processor that determines the outputs based on input values. You can test your embedded control system in isolation by creating an additional system to exercise the embedded system in hardware. This approach allows you to perform

automated testing of hardware-in-the-loop (HIL) by using the System Console to drive the inputs into the system and measure the outputs. This approach has the advantage of allowing you to test your embedded system without modifying the design. Figure 9-3 illustrates HIL testing using the System Console.

**Figure 9-3. Hardware-in-the-Loop Testing Using the System Console**



 To learn more about the System Console refer to the [System Console User Guide](#).

## SignalTap II Embedded Logic Analyzer


The SignalTap® II embedded logic analyzer is available in the Quartus II software. It reuses the JTAG pins of the FPGA and has a low Quartus II fitter priority, allowing it to be non-intrusive. Because this logic analyzer is integrated in your design automatically, it takes synchronized measurements without the undesirable side effects of output pin capacitance or I/O delay. The SignalTap II embedded logic analyzer also supports Tcl scripting so that you can automate data capture, duplicating the functionality that external logic analyzers provide.

This logic analyzer can operate while other JTAG components, including the Nios II JTAG debug module and JTAG UART, are in use, allowing you to perform co-verification. You can use the plug-in support available with the SignalTap II embedded logic analyzer to enhance your debug capability with any of the following:

- Instruction address triggering
- Non-processor related triggering
- Software disassembly
- Instruction display (in hexadecimal or symbolic format)

You can also use this logic analyzer to capture data from your embedded system for analysis by the MATLAB software from Mathworks. The MATLAB software receives the data using the JTAG connection and can perform post processing analysis. Using looping structures, you can perform multiple data capture cycles automatically in the MATLAB software, instead of manually controlling the logic analyzer using the Quartus II design software.

Because the SignalTap II embedded logic analyzer uses the FPGA's JTAG connection, continuous data triggering may result in lost samples. For example, if you capture data continuously at 100 MHz, you should not expect all of your samples to be displayed in the logic analyzer GUI. The logic analyzer buffers the data at 100 MHz; however, if the JTAG interface becomes saturated, samples are lost.

 To learn more about SignalTap II embedded logic analyzer and co-verification, refer to the following documentation: *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook* and *AN323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems*.

## External Instrumentation

If your design does not have enough on-chip memory to store trace buffers, you can use an external logic analyzer for debugging. External instrumentation is also necessary if you require any of the following:

- Data collection with pin loading
- Complex triggers
- Asynchronous data capture

Altera provides procedures to connect external verification devices such as oscilloscopes, logic analyzers, and protocol analyzers to your FPGA.


## SignalProbe

The SignalProbe incremental routing feature allows you to route signals to output pins of the FPGA without affecting the existing fit of a design to a significant degree. You can use SignalProbe to investigate internal device signals without rewriting your HDL code to pass them up through multiple layers of the design hierarchy to a pin. Creating such revisions manually is time-consuming and error-prone.

Altera recommends SignalProbe when there are enough pins to route internal signals out of the FPGA for verification. If FPGA pins are not available, you have the following three alternatives:

- Reduce the number of pins used by the design to make more pins available to SignalProbe
- Use the SignalTap II embedded logic analyzer
- Use the Logic Analyzer Interface

Revising your design to increase the number of pins available for verification purposes requires design changes and can impact the design schedule. Using the SignalTap II embedded logic analyzer is a viable solution if you do not require continuous sampling at a high rate. The SignalTap II embedded logic analyzer does not require any additional pins to be routed; however, you must have enough unallocated logic and memory resources in your design to incorporate it. If neither of these approaches is viable, you can use the logic analyzer interface.


 To learn more about SignalProbe, refer to the *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*.

## Logic Analyzer Interface

The Quartus II Logic Analyzer Interface is a JTAG programmable method of driving multiple time-domain multiplexed signals to pins for external verification. Because the Logical Analyzer Interface multiplexes pins, it minimizes the pincount requirement. Groups of signals are assigned to a bank. Using JTAG as a communication channel, you can switch between banks.

You should use this approach when SignalTap II embedded logic analyzer is insufficient for your verification needs. Some external logic analyzer manufacturers support the Logic Analyzer Interface. These logic analyzers have various amounts of support. The most important feature is the ability to let the measurement tools cycle through the signal banks automatically.

The ability to cycle through signal banks is not limited to logic analyzers. You can use it for any external measurement tool. Some developers use low speed indicators, for example LEDs, for verification. You can use the Logic Analyzer interface to map many banks of signals to a small number of verification LEDs. You may wish to leave this form of verification in your final design so that your product is capable of creating low-level error codes after deployment.

 To learn more about the Quartus II Logic Analyzer Interface, refer to the *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*.

## Stimuli Generation

To effectively test your system you must maximize your test coverage with as few stimuli as possible. To maximize your test coverage you should use a combination of static and randomly generated data. The static data contains a fixed set of inputs that you can use to test the standard functionality and corner cases of your system.

Random tests are generated at run time, but must be accessible when failures occur so that you can analyze the failure case. Random test generation is particularly effective after static testing has identified the majority of issues with the basic functionality of your design. The test cases created may uncover unanticipated issues. Whenever randomly generated test inputs uncover issues with your system, you should add those cases to your static test data set for future testing.

Creating random data for use as inputs to your system can be challenging because pseudo random number generators (PRNG) tends to repeat patterns. Choose a different seed each time you initialize the PRNG for your random test generator. The random number generator creates the same data sequence if it is seeded with the same value.

Seed generation is an advanced topic and is not covered in detail in this document. The following recommendations on creating effective seed values should help you avoid repeating data values:

- Use a random noise measurement. One way to do this is by reading the analog output value of an A/D converter.
- Use multiple asynchronous counters in combination to create seed values.
- Use a timer value as the seed (that is, the number of seconds from a fixed point in time).

Using a combination of seed generation techniques can lead to more random behavior. When generating random sequences, it is important to understand the distribution of the random data generated. Some generators create linear sequences in which the distribution is evenly spread across the random number domain. Others create non-linear sequences that may not provide the test coverage you require. Before you begin using a random number generator to verify your system, examine the data created for a few sequences. Doing so helps you understand the patterns created and avoid using an inappropriate set of inputs.


## Board Bring-up

You can minimize board bring-up time by adopting a systematic strategy. First, break the task down into manageable pieces. Verify the design in segments, not as a whole, beginning with peripheral testing.

## Peripheral Testing

The first step in the board bring-up process is peripheral testing. Add one interface at a time to your design. After a peripheral passes the tests you have created for it, you should remove it from the test design. Designers typically leave the peripherals that pass testing in their design as they move on to test other peripherals. Sometimes this is necessary; however, it should be avoided when possible because multiple peripherals can create instability due to noise or crosstalk. By testing peripherals in a system individually, you can isolate the issues in your design to a particular interface.

A common failure in any system is involves memory. The most problematic memory devices operate at high speeds, which can result in timing failures. High performance memory also requires many board traces to transfer data, address, and control signals, which cause failures if not routed properly. You can use the Nios II processor to verify your memory devices using verification software or a debugger such as the FS2 console. The Nios II processor is not capable of stress testing your memory but it can be used to detect memory address and data line issues.

 For more information on debugging refer to the *Debugging Nios II Designs* chapter in the *Embedded Design Handbook*.

### Data Trace Failure

If your board fabrication facility does not perform bare board testing, you must perform these tests. To detect data trace failures on your memory interface you should use a pattern typically referred to as “walking ones.” The walking ones pattern shifts a logical 1 through all of the data traces between the FPGA and the memory device. The pattern can be increasing or decreasing; the important factor is that only one data signal is 1 at any given time. The increasing version of this pattern is as follows: 1, 2, 4, 8, 16, and so on.

Using this pattern you can detect a few issues with the data traces such as short or open circuit signals. A signal is short circuited when it is accidentally connected to another signal. A signal is open circuited when it is accidentally left unconnected. Open circuits can have a random signal behavior unless a pull-up or pull-down resistor is connected to the trace. If a pull-up or pull-down resistor is used, the signal drives a 0 or 1; however, the resistor is weak relative to a signal being driven by the test, so that test value overrides the pull-up or pull-down resistor.

To avoid mixing potential address and data trace issues in the same test, test only one address location at a time. To perform the test, write the test value out to memory, and then read it back. After verifying that the two values are equal, proceed to testing the next value in the pattern. If the verification stage detects a variation between the written and read values, a bit failure has occurred. [Table 9-2](#) provides an example of the process used to find a data trace failure. It makes the simplifying assumption that sequential data bits are routed consecutively on the PCB.

**Table 9-2. Walking Ones Example (Part 1 of 2)**

Written Value	Read Value	Failure Detected
00000001	00000001	No failure detected
00000010	00000000	Error, most likely the second data bit, D[1] stuck low or shorted to ground
00000100	00000100	No failure detected, confirmed D[1] is stuck low or shorted to another trace that is not listed in this table.

**Table 9-2. Walking Ones Example (Part 2 of 2)**

00001000	00001000	No failure detected
00010000	00010000	No failure detected
00100000	01100000	Error, most likely D[6] and D[5] short circuited
01000000	01100000	Error, confirmed that D[6] and D[5] are short circuited
10000000	10000000	No failure detected

### Address Trace Failure

The address trace test is similar to the walking ones test used for data with one exception. For this test you must write to all the test locations before reading back the data. Using address locations that are powers of two, you can quickly verify all the address traces of your circuit board.

The address trace test detects the aliasing effects that short or open circuits can have on your memory interface. For this reason it is important to write to each location with a different data value so that you can detect the address aliasing. You can use increasing numbers such as 1, 2, 3, 4, and so on while you verify the address traces in your system. [Table 9-3](#) shows how to use powers of two in the process of finding an address trace failure:

**Table 9-3. Powers of Two Example**

Address	Written Value	Read Value	Failure Detected
00000000	1	1	No failure detected
00000001	2	2	No failure detected
00000010	3	1	Error, the second address bit, A[1], is stuck low
00000100	4	4	No failure detected
00001000	5	5	No failure detected
00010000	6	6	No failure detected
00100000	7	6	Error, A[5] and A[4] are short circuited
01000000	8	8	No failure detected
10000000	9	9	No failure detected

### Device Isolation

Using device isolation techniques, you can disable features of devices on your PCB that cause your design to fail. Typically designers use device isolation for early revisions of the PCB, and then remove these capabilities before shipping the product.

Most designs use crystal oscillators or other discrete components to create clock signals for the digital logic. If the clock signal is distorted by noise or jitter, failures may occur. To guard against distorted clocks, you can route alternative clock pins to your FPGA. If you include SMA connectors on your board, you can use an external clock generator to create a clean clock signal. Having an alternative clock source is very useful when debugging clock-related issues.

Sometimes the noise generated by a particular device on your board can cause problems with other devices or interfaces. Having the ability to reduce the noise levels of selected components can help you determine the device that is causing issues in your design. The simplest way to isolate a noisy component is to remove the power source for the device in question. For devices that have a limited number of power pins, if you include 0 ohm resistors in the path between the power source and the pin. You can cut off power to the device by removing the resistor. This strategy is typically not possible with larger devices that contain multiple power source pins connecting directly to a board power plane.

Instead of removing the power source from a noisy device, you can often put the device into a reset state by driving the reset pin to an active state. Another option is to simply not exercise the device so that it remains idle.

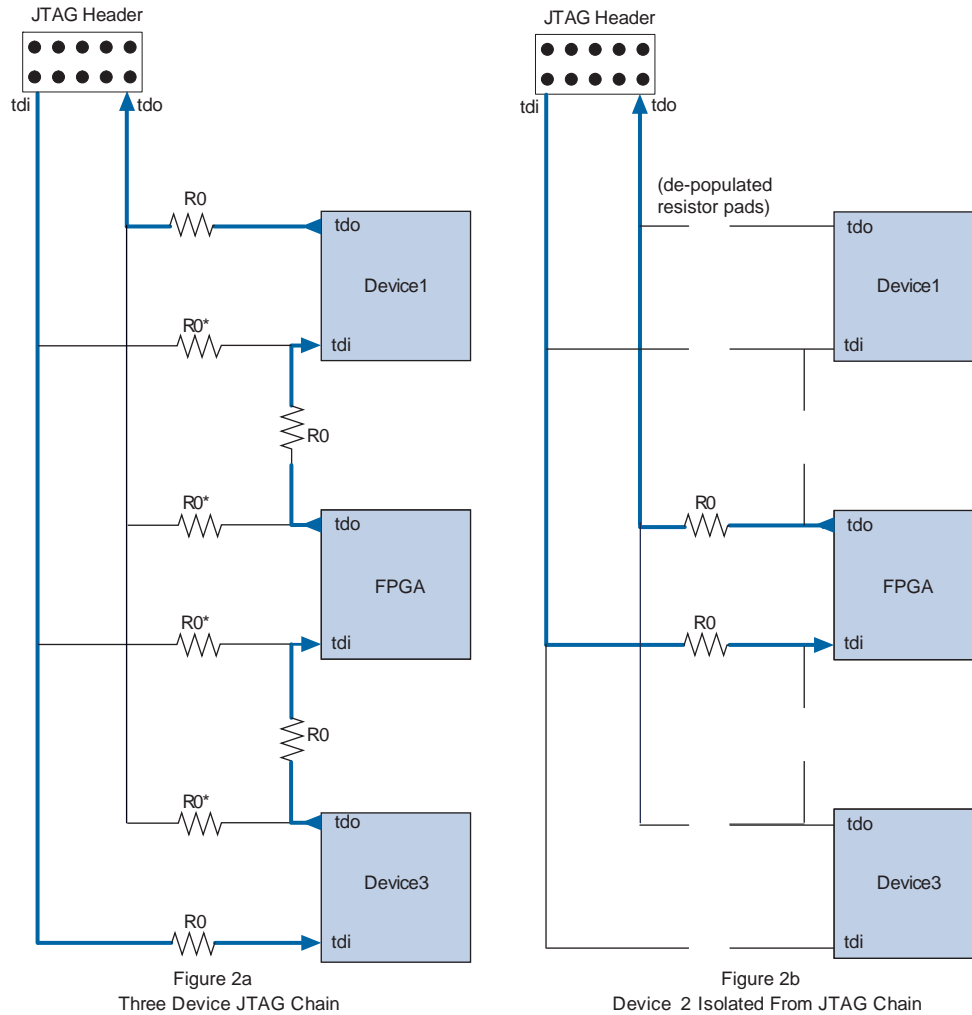
A noisy power supply or ground plane can create signal integrity issues. With the typical voltage swing of digital devices frequently below a single volt, the power supply noise margin of devices on the PCB can be as little as 0.2 volts. Power supply noise can cause digital logic to fail. For this reason it is important to be able to isolate the power supplies on your board. You can isolate your power supply by using fuses that are removed so that a stable external power supply can be substituted temporarily in your design.

## JTAG

FPGAs use the JTAG interface for programming, communication, and verification. Designers frequently connect several components, including FPGAs, discrete processors, and memory devices, communicating with them through a single JTAG chain. Sometimes the JTAG signal is distorted by electrical noise, causing a communication failure for the entire group of devices. To guarantee a stable connection, you must isolate the FPGA under test from the other devices in the same JTAG chain.


Figure 9-4a illustrates a JTAG chain with three devices. The tdi and tdo signals include 0 ohm resistors between each device. By removing the appropriate resistors, it is possible to isolate a single device in the chain as Figure 9-4b illustrates. This technique allows you to isolate one device while using a single JTAG chain.

**Figure 9-4. JTAG Isolation**



R0 Zero Ohm Resistor  
 R0\* Zero Ohm Resistor Stuff Option  
 — Serial Data Flow

tdi = test data in  
 tdo = test data out

 To learn more about JTAG refer to [AN39: IEEE 1149.1\(JTAG\) Boundary-Scan Testing in Altera Devices](#).

## Board Testing

You should convert the simulations you run to verify your intellectual property (IP) before fabrication to test vectors that you can then run on the hardware to verify that the simulation and hardware versions exhibit the same behavior. Manufacturing can also use these tests as part of a regularly scheduled quality assurance test. Because the tests are run by engineers in other organizations they must be documented and easy to run.

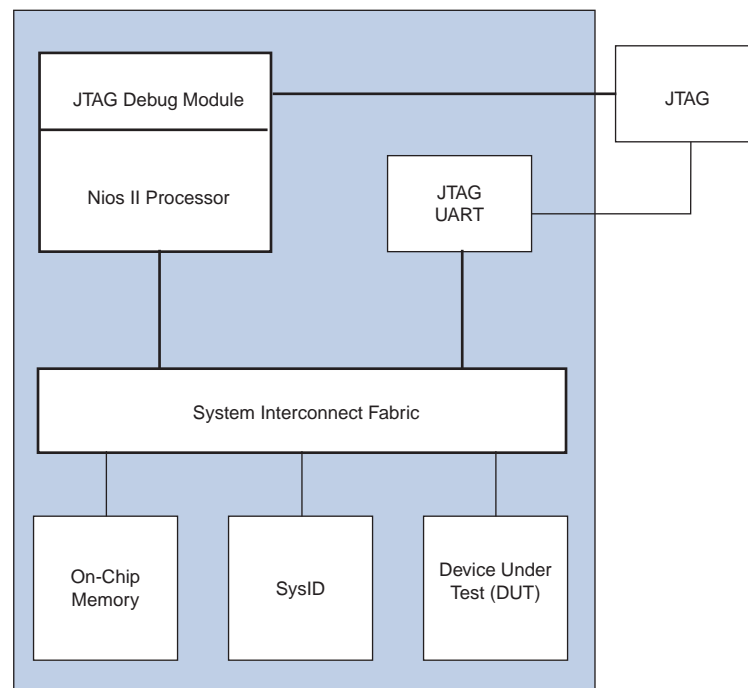
## Minimal Test System

Whether you are creating your first embedded system in a FPGA, or are debugging a complex issue, you should always begin with a minimal system. To minimize the probability of signal integrity issues, reduce the pincount of your system to the absolute minimal number of required pins. In an embedded design that includes the Nios II processor, the minimal pincount might be clock and reset signals. Such a system might include the following the following components:

- Nios II processor (with a level 1 debug core)
- On-chip memory
- JTAG UART
- System ID core

Using these four components you can create a functioning embedded system including debug and terminal access. To simplify your debug process, you should use a Nios II processor that does not contain a data cache. The Nios II/e and Nios II/s cores do not include data caches. The Nios II/f core can also be configured without a data cache. Figure 9-5 illustrates a minimal system. In this system, you have to route only the clock pin and reset pins, because the JTAG signals are automatically connected by the Quartus II software.

**Figure 9-5. Simple Test System**



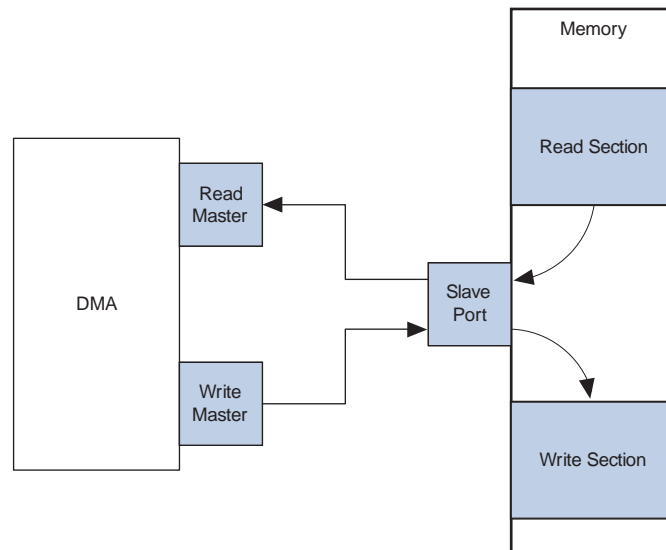
You can use the Nios II JTAG debug module to download software to the processor. Before testing any additional interfaces you should execute a small program that prints a message to the terminal to verify that your minimal system is functioning properly.

After you verify that the simple test system functions properly, archive the design. This design provides a stable starting point to which to add additional components as verification proceeds. In this system, you can use any of the following for testing:

- A Nios II processor
- A Nios II JTAG debug module and FS2 console
- The SignalTap II embedded logic analyzer
- An external logic interface
- SignalProbe
- A direct memory access (DMA) engine
- In-system updating of memory and constants

The Nios II processor is not capable of stress testing high speed memory devices. Altera recommends that you use a DMA engine to stress test memories. A stress test should access memory as frequently as possible, performing continuous reads or writes. Typically, the most problematic access sequence for high-speed memory involves the bus turnaround between read and write accesses. You can test these cases by connecting the DMA read and write masters to the same memory and transferring the contents from one location to another, as shown in [Figure 9-6](#).

**Figure 9-6. Using a DMA to Stress Test Memory Devices**



By modifying the arbitration share values for each master to memory connection, you can control the sequence. To alternate reads and writes, you can use an arbitration share of one for each DMA master port. To perform two reads followed by two writes, use an arbitration value of two for each DMA master port. To create more complicated access sequences you can create a custom master or use the Nios II C2H Compiler to create hardware used for testing.

 To learn more about the topics covered in this section refer to the following documentation:

- [Nios II Hardware Development Tutorial](#)
- [Quartus II Verification Methods](#) web page of the Altera website
- [DMA Controller Core](#) chapter in volume 5 of the *Quartus II Handbook*
- [In-System Updating of Memory and Constants](#) chapter in volume 3 of the *Quartus II Handbook*

## System Verification

System verification is the final step of system design. This section focuses on common mistakes designers make during system verification and methods for correcting and avoiding them. It includes the following topics:

- [Designing with Verification in Mind](#)

- Accelerating Verification
- Using Software to Verify Hardware
- Environmental Testing

## Designing with Verification in Mind

As you design, you should focus on both the development tasks and the verification strategy. Doing so results in a design that is easier to verify. If you create large, complicated blocks of logic and wait until the HDL code is complete before testing, you spend more time verifying your design than if you verify it one section at a time.

Consider leaving in verification code after the individual sections of your design are working. If you remove too much verification logic it becomes very difficult to reintroduce it at a later time if needed. If you discover an issue during system integration, you may need to revisit some of the smaller block designs. If you modify one of the smaller blocks, you must re-test it to verify that you have not created additional issues.

Designing with verification in mind is not limited to leaving verification hooks in your design. Reserving enough hardware resources to perform proper verification is also important. The following recommendations can help you avoid running out of hardware resources:

- Design and verify using a larger pin-compatible FPGA.
- Reserve hardware resources for verification in the design plan.
- Design the logic so that optional features can be removed to free up verification resources.

Finally, schedule a nightly regression test of your design to increase your test coverage between hardware or software compilations.

## Accelerating Verification

Altera recommends the verification flow illustrated in [Figure 9-7](#). Verify each component as it is developed. By minimizing the amount of logic being verified, you can reduce the time it takes to compile and simulate your design. Consequently, you minimize the iteration time to correct design issues.

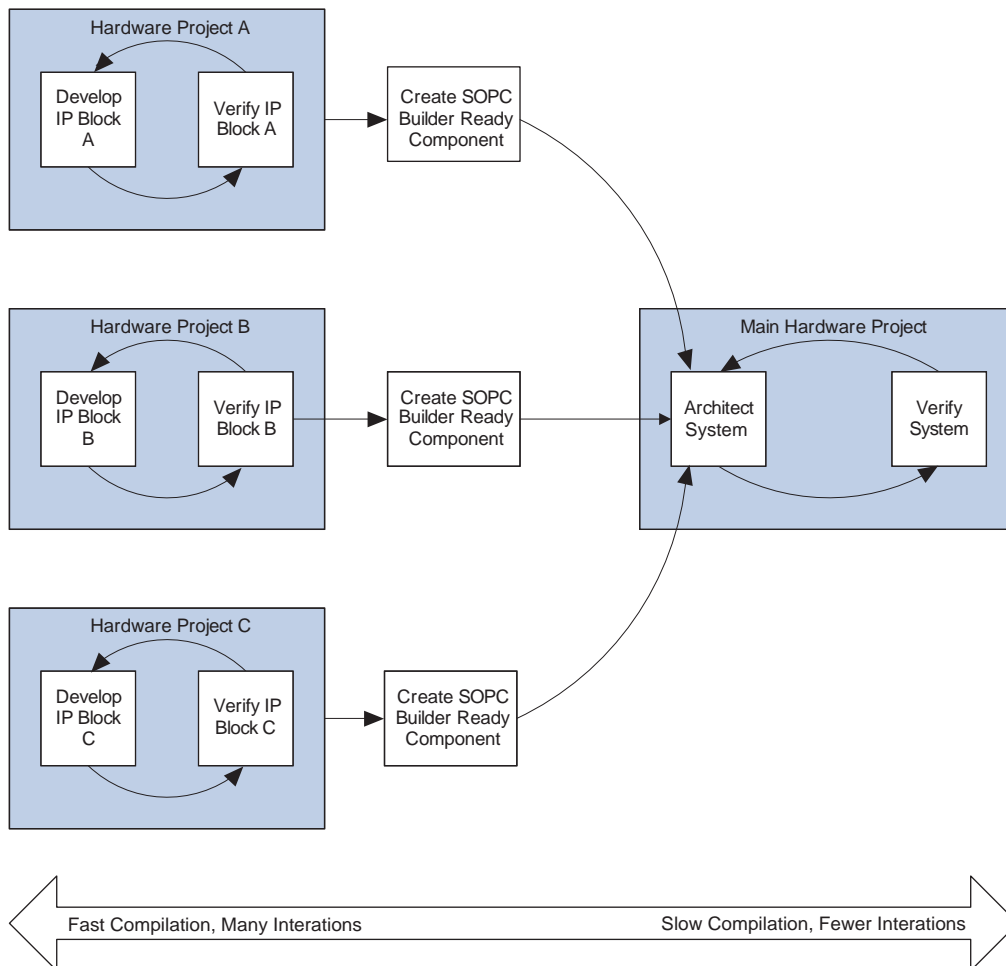
After the individual components are verified, you can integrate them in an SOPC Builder system. The integrated system must include an Avalon-MM or Avalon Streaming (Avalon-ST) port. Using the component editor available from SOPC Builder, you add an Avalon-MM interface to your existing component and integrate it in your system.


After your system is created in SOPC Builder, you can continue the verification process of the system as a whole. Typically, the verification process has the following two steps:

1. Generate then simulate
2. Generate, compile, and then verify in hardware

The first step provides easier access to the signals in your system. When the simulation is functioning properly, you can move the verification to hardware. Because the hardware is orders of magnitude faster than the simulation, running test vectors on the actual hardware saves time.

**Figure 9-7. IP Verification and Integration Flow**



 To learn more about component editor and system integration, refer to the following documentation:

- The *Component Editor* chapter in the *SOPC Builder User Guide*
- The *SOPC Builder Component Development Walkthrough* chapter in the *SOPC Builder User Guide*
- The *Avalon Interface Specifications*

## Using Software to Verify Hardware

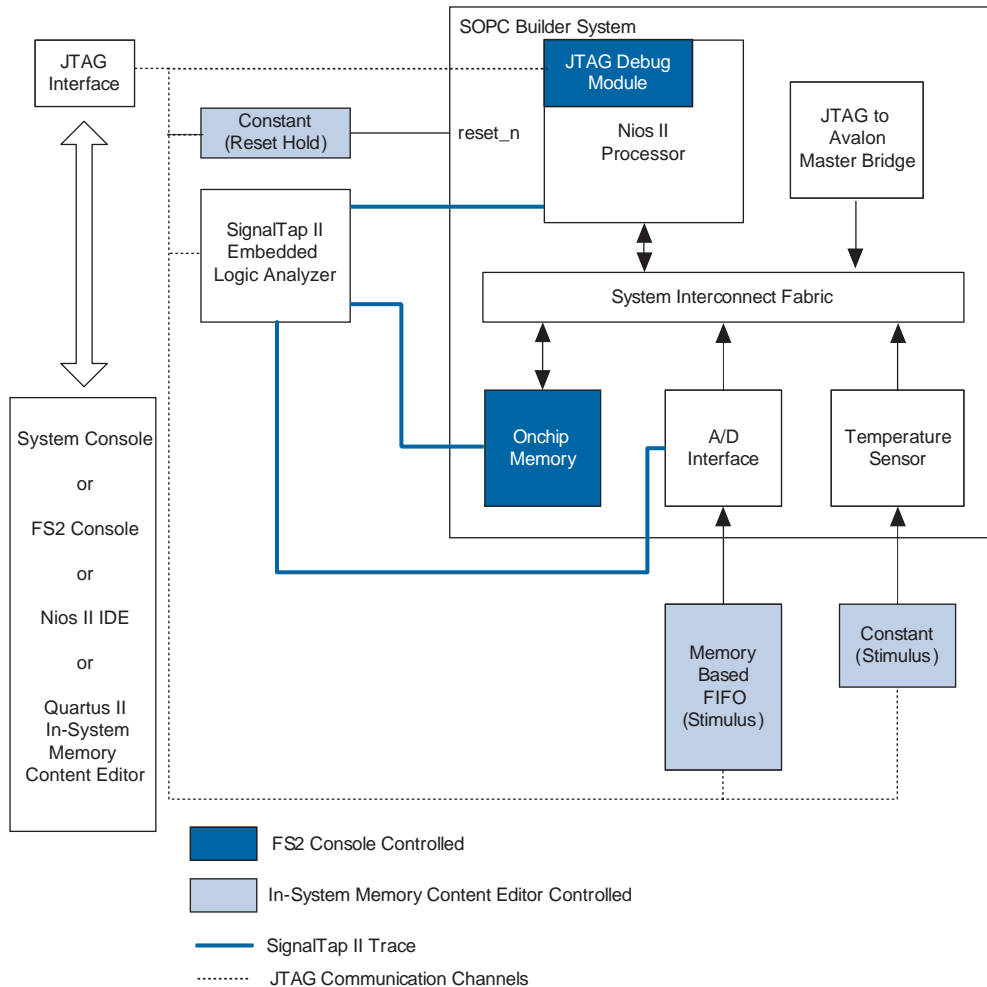
Many hardware developers use test benches and test harnesses to verify their logic in simulations. These strategies can be very time consuming. Instead of relying on simulations for all your verification tasks, you can test your logic using software or scripts, as [Figure 9-8](#) illustrates.

This system uses the JTAG interface to access components connected to the system interconnect fabric and to create stimuli for the system. If you use the JTAG server provided by the Quartus II programmer, this system can also be located on a network and accessed remotely. You can download software to the Nios II processor using the Nios II IDE. You can also use the Nios II JTAG debug core to transmit files to and from your embedded system using the host file system. Using the System Console you can access components in your system and also run scripts for automated testing purposes.

Using the Quartus II In-System Memory Content Editor, you can create stimuli for the two components that control external peripherals. You can also use the In-System Memory Content Editor to place the embedded system in reset while new stimulus values are sent to the system. The In-System Memory Editor supports Tcl scripting, which you can use to automate the verification process. This approach is similar to using the FS2 console to control logic in your system. However, unlike the FS2 console, you can use the In-System Memory Content Editor to access hardware that is not memory-mapped. All of the verification techniques described in this chapter can be scripted, allowing many test cycles to be executed without user interaction.

To learn more about using the host file system refer to the *Host File System* software example design available with the Nios II EDS. The *Developing Nios II Software* chapter of the *Embedded Design Handbook* also includes a significant amount of information about the system file system.

**Figure 9–8. Script Controlled Verification**



To learn more about the verification and scripting abilities outlined in the example above, refer to the following documentation:

- [Basic Quartus II Software Tcl Scripting](#) training course web page of the Altera website
- [Quartus II Scripting Reference Manual](#)

## Environmental Testing

The last stage of verification is end-user environment testing. Most verification is performed under ideal conditions. The following conditions in the end user's environment can cause the system to fail:

- Voltage variation
- Vibration
- Temperature variation
- Electrical noise

Because it is difficult to predict all the applications for a particular product, you should create a list of operational specifications before designing the product. You should verify these specifications before shipping or selling the product. The key issue with environmental testing is the difficulty associated with obtaining measurements while the test is underway. For example, it can be difficult to measure signals with an external logic analyzer while your product is undergoing vibration testing.

While choosing methods to test your hardware design during the early verification stages, you should also consider how to adapt them for environmental testing. If you believe your product is susceptible to vibration problems, you should choose sturdy instrumentation methods when testing memory interfaces. Alternatively, if you believe your product may be susceptible to electrical noise, then you should choose a highly reliable interface for debug purposes.

While performing early verification of your design, you can also begin end-environment testing. Doing so helps you detect potential flaws in early in the design process. For example, if you wish to test temperature variations, you can use a heat gun on the product while you are testing. If you wish to perform voltage variation testing, isolate the power supply in your system and vary the voltage using an external power supply. Using these verification techniques, you can avoid late design changes due to failures during environmental testing.

## Document Revision History

Table 9-4 shows the revision history for this document.

**Table 9-4. Document Revision History**

Date	Version	Changes
July 2011	1.3	<ul style="list-style-type: none"> <li>■ Updated references.</li> </ul>
November 2008	1.2	<ul style="list-style-type: none"> <li>■ In the FS2 Console section, added <code>sld info</code> command and an example that writes and reads a range of memory addresses.</li> <li>■ Added introductory discussion to the System Console.</li> <li>■ Added JTAG to Avalon Master Bridge to <a href="#">Figure 9-8</a>.</li> </ul>
June 2008	1.1	Corrected Table of Contents.
March 2008	1.0	Initial release.