Virtual JTAG (sld_virtual_jtag)

Megafunction User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

Software Version: 8.1
Document Version: 2.0
Document Date: © December 2008

UG-SLDVRTL-2.0

Contents

1. About This Megafunction

Device Family Support

The virtual JTAG (SLD_VIRTUAL_JTAG) megafunction supports the following target Altera® device families:

- Arria® series
- Stratix® series
- Cyclone® series
- HardCopy® ASICs
- MAX® II series
- APEX™ II, APEX 20KE, APEX 20KC

Introduction

The virtual JTAG megafunction provides access to the JTAG input pins and all of the control signals from the JTAG controller on your device. It is one feature in the on-chip debugging tools portfolio.

The on-chip debugging tool suite is a powerful set of tools enabling real time verification of a design.

Each feature in the on-chip debugging tool set leverages on-chip resources to get real time visibility to the logic under test. During runtime, each tool shares the JTAG connection to transmit collected test data to the Quartus® II software for analysis. The tool set consists of a set of GUIs, megafunction intellectual property (IP) cores, and Tcl application programming interfaces (APIs). The GUIs provide for the configuration of test signals and the visualization of data captured during debugging. The Tcl scripting interface provides for automation during runtime.

Table 1–1 shown describes the available tools in the on-chip debugging tool suite.

Table 1–1. Available Tools in the On-Chip Debugging Tool Suite (Part 1 of 2)

Tool	Description	Typical Circumstances of Use
SignalTap® II Embedded Logic Analyzer	This embedded logic analyzer uses FPGA resources to sample tests nodes and outputs the information to the Quartus II software for display and analysis.	You have spare on-chip memory and you want functional verification of your design running in hardware.
SignalProbe	This tool incrementally routes internal signals to I/O pins while preserving the results from your last place-and-route procedure.	You have spare I/O pins and you would like to check the operation of a small set of control pins using either an external logic analyzer or an oscilloscope.

Table 1–1. Available Tools in the On-Chip Debugging Tool Suite (Part 2 of 2)

Tool	Description	Typical Circumstances of Use
Logic Analyzer Interface (LAI)	This tool multiplexes a larger set of signals to a smaller number of spare I/O pins. LAI allows you to select which signals are switched onto the I/O pins over a JTAG connection.	You have limited on-chip memory, and have a large set of internal data buses that you would like to verify using an external logic analyzer. Logic analyzer vendors, such as Tektronics and Agilent, provide integration with the tool to improve the usability.
In-System Memory Content Editor	This tool displays and allows you to edit on-chip memory.	You would like to view and edit the contents of either the instruction cache or data cache of a Nios® II processor application.
In-System Sources and Probes	This feature provides an easy way to drive and sample logic values to and from internal nodes using the JTAG interface.	You want to prototype a front panel with virtual buttons for your FPGA design.
Virtual JTAG Interface	This megafunction opens up the JTAG interface so that you can develop your own custom applications.	You want to generate a large set of test vectors and send them to your device over the JTAG port to functionally verify your design running in hardware.

The virtual JTAG megafunction IP gives you direct access to the JTAG control signals routed to the FPGA core logic, which gives you a fine granularity of control over the JTAG resource. This opens up the JTAG resource as a general-purpose serial communication interface. A complete Tcl API is available for sending and receiving transactions into your device during runtime. Because the JTAG pins are readily accessible during runtime, this megafunction can be an easy way to customize a JTAG scan chain internal to the device, which can be used to create debugging applications. Examples of debugging applications can include the following scenarios:

- Induce trigger conditions evaluated by a SignalTap II Embedded Logic Analyzer by exercising test signals connected to the SignalTap II Embedded Logic Analyzer instance.
- Use as a replacement for a front panel interface during the prototyping phase of the design
- Insert test vectors for exercising the design under test

For more information about the Quartus II software on-chip debugging tool suite, refer to *Section V: In-System Design Debugging* in the *Quartus II Handbook*.

The following section provides background information with an overview of the JTAG protocol and the system-level debugging (SLD) infrastructure used in Altera devices. The SLD infrastructure is an extension of the JTAG protocol for use with Altera-specific applications and user applications, such as the SignalTap II Embedded Logic Analyzer. This section serves as a high-level introduction, with the appropriate level of information needed to use the virtual JTAG megafunction properly.

For more information about the JTAG protocol, refer to *AN 39: IEEE 1149.1 (JTAG) Boundary-Scan Testing in Altera Devices*.

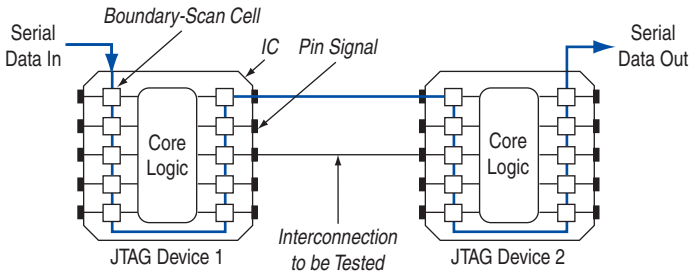
Subsequent sections describe how to use the virtual JTAG megafunction and provide a few application examples to help get you started.

The JTAG Protocol

The original intent of the JTAG protocol (standardized as IEEE 1149.1) was to simplify PCB interconnectivity testing during the manufacturing stage. As access to integrated circuit (IC) pins became more limited due to tighter lead spacing and FPGA packages, testing through traditional probing techniques, such as "Bed-of-nails" test fixtures, became infeasible. The JTAG protocol alleviates the need for physical access to IC pins via a shift register chain placed near the I/O ring. This set of registers near the I/O ring, also known as boundary scan cells (BSCs), sample and force values out onto the I/O pins. The BSCs from JTAG-compliant ICs are daisy-chained into a long serial-shift chain and driven via a serial interface.

During boundary scan testing, software shifts out test data over the serial interface to the BSCs of select ICs. This test data forces a known pattern to the pins connected to the affected BSCs. If the adjacent IC at the other end of the PCB trace is JTAG-compliant, the BSC of the adjacent IC can sample the test pattern and feed the BSCs back to the software for analysis. Figure 1–1 illustrates the concept of boundary-scan testing.

Figure 1–1. IEEE Std. 1149.1 Boundary-Scan Testing



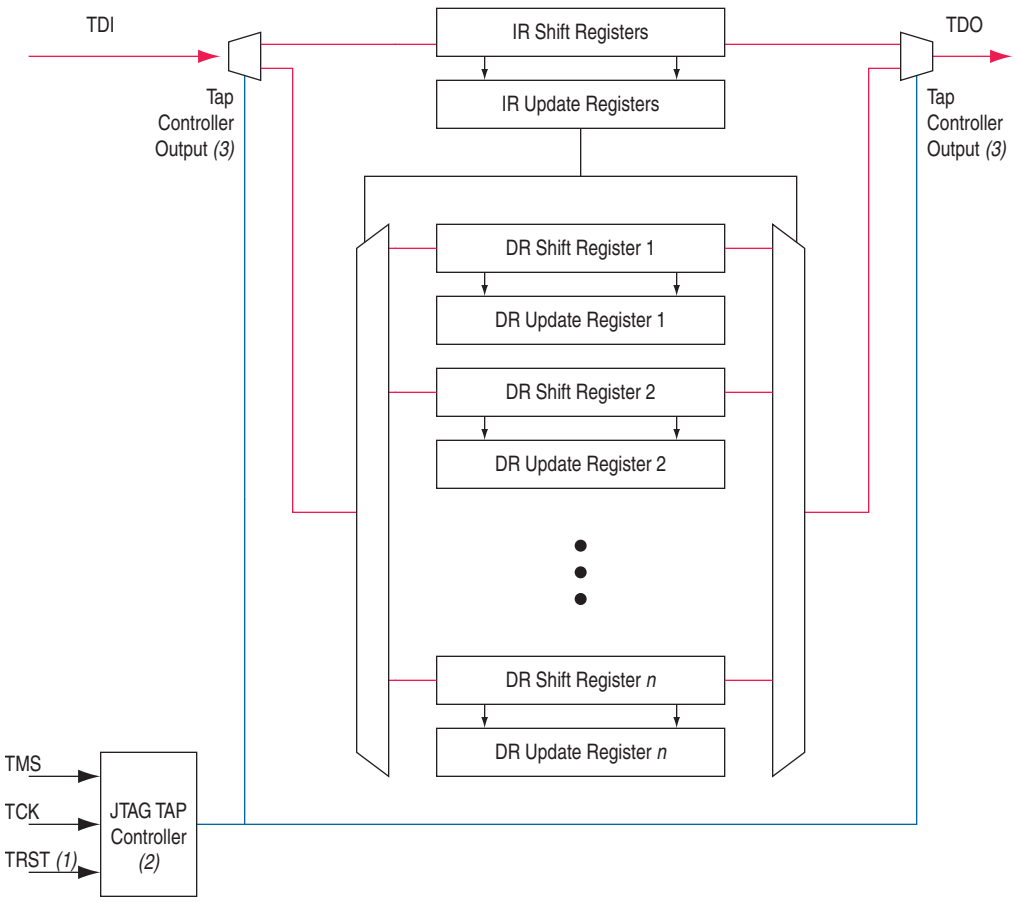
Because the JTAG interface can shift in any information to the device and is available on all Altera devices, it lends itself well to being a low footprint, general purpose communication interface. In addition to boundary scan applications, Altera devices use the JTAG port for other applications, such as device configuration and all of the on-chip debugging features available in the Quartus II software.

The basic architecture of the JTAG circuitry consists of the following components:

- A set of Data Registers (DRs)
- An Instruction Register (IR)
- A state machine to arbitrate data (known as the Test Access Port (TAP) controller)
- A four- or five-pin serial interface, consisting of the following pins:
  - Test data in (`TDI`), used to shift data into the IR and DR shift register chains
  - Test data out (`TDO`), used to shift data out of the IR and DR shift register chains
  - Test mode select (`TMS`), used as an input into the TAP controller
  - `TCK`, used as the clock source for the JTAG circuitry
  - `TRST` resets the TAP controller. This is an optional input pin defined by the 1149.1 standard. (The `TRST` pin is not present in the Cyclone device family.)

Figure 1–2 shows a functional model of the JTAG circuitry.

Figure 1–2. Functional Model of the JTAG Circuitry



Notes to Figure 1–2:

(1) The TRST pin is an optional pin in the 1149.1 Standard. It is not available in Cyclone devices.

(2) The TAP controller is a hard controller; that is, not created using programmable resources.

(3) The major function of the TAP controller is to route test data between the IR and DR register chains. A complete description of the control functions is given in the section describing the TAP controller state machine.

The bank of DRs is the primary data path of the JTAG circuitry. It carries the payload data for all JTAG transactions. Each DR chain is dedicated to serving a specific function. Boundary scan cells form the primary DR chain. The other DR chains are used for identification, bypassing the IC during boundary scan tests, or a custom set of register chains with functions defined by the IC vendor. Altera uses two of the DR chains with user-defined IP that requires the JTAG chain as a communication resource, such as the on-chip debugging applications. The Virtual JTAG megafunction, in particular, allows you to extend the two DR chains to a user-defined custom application.

The instruction register is used to select the bank of Data Registers to which the TDI and TDO must connect. It functions as an address register for the bank of Data Registers. Each IR instruction maps to a specific DR chain.

All shift registers that are a part of the JTAG circuitry (IR and DR register chains) are composed of two kinds of registers:

- **Shift registers**—Capture new serial shift input from the TDI pin
- **Parallel hold registers**—Connect to each shift register to hold the current input in place while any shifting is done; the parallel hold registers ensure stability in the output while new data is being shifted

The TAP controller is a state machine with a set of control signals that routes TDI data between the Instruction Register and the bank of DR chains, controls the start and stop of any shift transactions, and controls the data flow between the parallel hold registers and the shift registers of the Instruction Register and the Data Register. The TAP controller is controlled by the TMS pin.

Figure 1–3 shows the TAP controller state machine. A description of each of the states is provided in Table 1–1.

Figure 1–3. JTAG TAP Controller State Machine

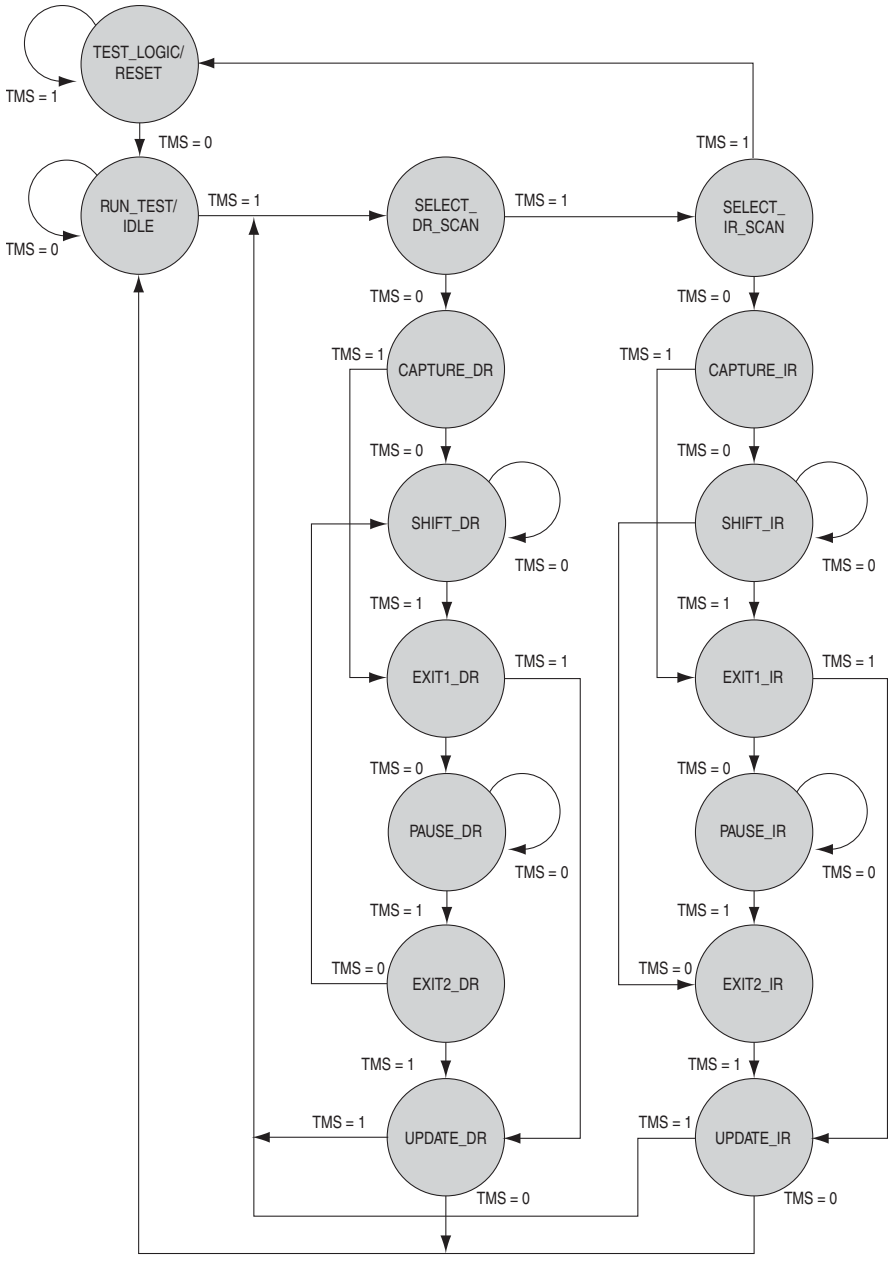


Table 1–2. Functional Description for the TAP Controller States (Part 1 of 2)

TAP Controller State	Functional Description
Test-Logic-Reset	The test logic of the JTAG scan chain is disabled.
Run-Test/Idle	This is a hold state. Once entered, the controller will remain in this state as long as TMS is held low.

Table 1–2. Functional Description for the TAP Controller States (Part 2 of 2)

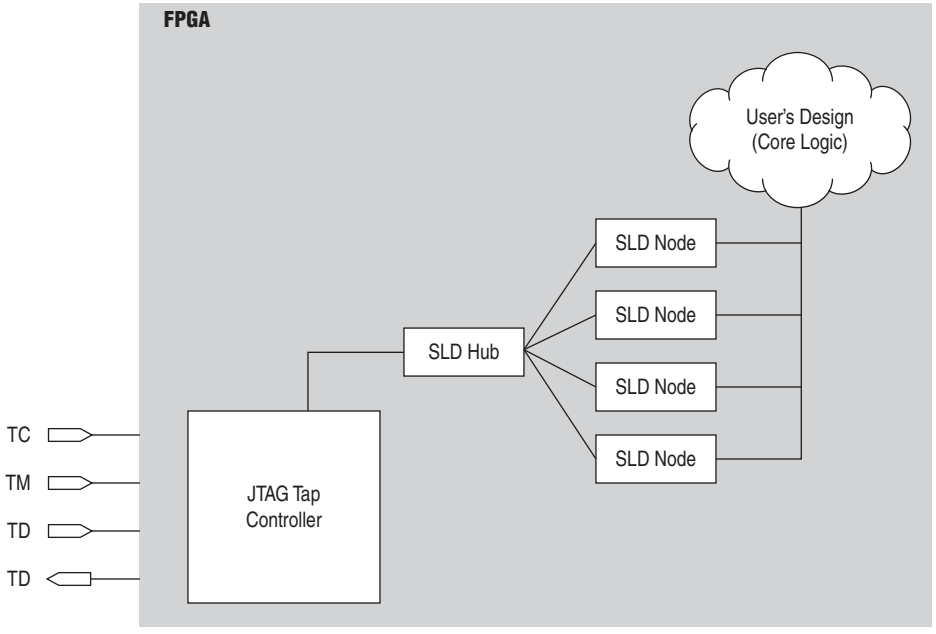
TAP Controller State	Functional Description
Select DR-Scan/Select IR Scan	These are temporary controller states. A decision is made here whether to enter the DR states or the IR states.
Capture DR/Capture IR	These states enable a parallel load of the shift registers from the hold registers on the rising edge of TCK.
Shift DR/Shift IR	These states enable shifting of the DR and IR chains.
Exit1 DR/Exit1 IR	Temporary hold states. A decision is made in these states to either advance to the **Update** states or the **Pause** states.
Pause DR/Pause IR	This controller state allows shifting of the Instruction Register and Data Register to be temporarily halted.
Exit2 DR/Exit2 IR	Temporary hold states. A decision is made in these states to advance to the **Update** states.
Update DR/Update IR	These states enable a parallel load of the hold registers from the shift registers. Update happens on the falling edge of TCK.

System-Level Debugging Infrastructure

All on-chip debugging tools that require the JTAG resources share two Data Register chain paths. USER1 and USER0 instructions select these two DR register chain paths. These datapaths are an extension of the JTAG circuitry for use with the programmable logic elements in Altera devices.

Because the JTAG resource is shared among multiple on-chip applications, an arbitration scheme must define how the USER0 and USER1 scan chains are allocated between the different user applications. The system-level debugging (SLD) infrastructure defines the signaling convention and the arbitration logic for all programmable logic applications using a JTAG resource. Figure 1–4 shows the SLD infrastructure architecture.

Figure 1–4. System Level Debugging Infrastructure Functional Model



In the presence of an application that requires the JTAG resource, the Quartus II software automatically implements the SLD infrastructure to handle the arbitration of the JTAG resource. The communication interface between JTAG and any IP cores is transparent to the end designer. All components of the SLD infrastructure, except for the JTAG TAP controller, are built using programmable logic resources.

The SLD infrastructure mimics the IR/DR paradigm defined by the JTAG protocol. Each application implements an Instruction Register, and a set of Data Registers that operate similarly to the Instruction Register and Data Registers in the JTAG standard. Note that the Instruction Register and the Data Register banks implemented by each application are a subset of the USER1 and USER0 Data Register chains. The SLD infrastructure consists of three subsystems: the JTAG TAP controller (described in the previous section), the SLD hub, and the SLD nodes.

The SLD hub acts as the arbiter that routes the `TDI` pin connection between each SLD node. It is a state machine that mirrors the JTAG TAP controller state machine.

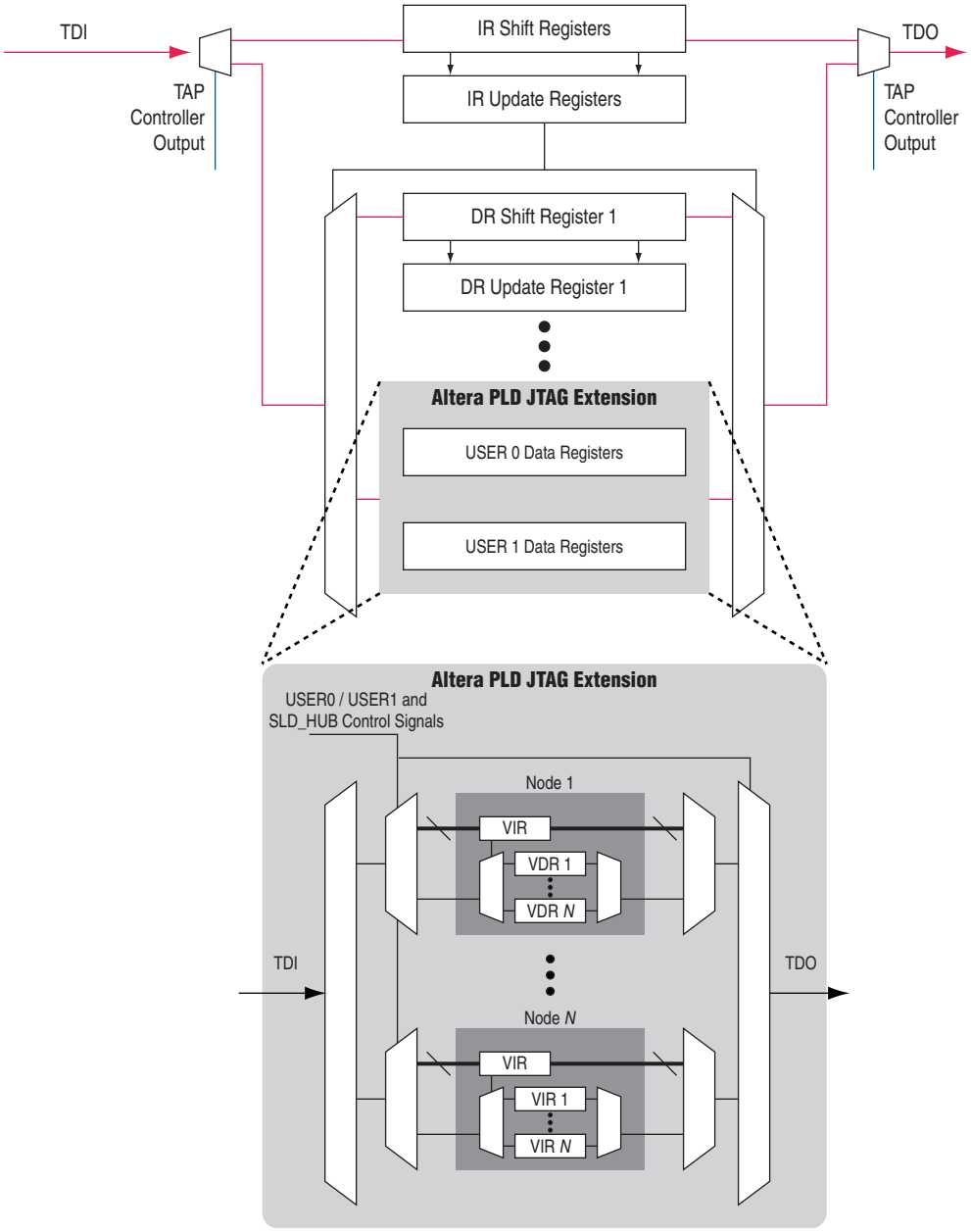
The SLD nodes in Figure 1–4 represent the communication channels for the end applications. Each instance of IP requiring a JTAG communication resource (such as SignalTap II Embedded Logic Analyzer) would have its own communication channel in the form of a SLD node interface to the SLD hub. Each SLD node instance has its own Instruction Register and its own bank of DR chains. Up to 255 SLD nodes can be instantiated, depending on resources available in your device.

Together, the sld_hub and the SLD nodes form a virtual JTAG scan chain within the JTAG protocol. It is virtual in the sense that both the Instruction Register and DR transactions for each SLD node instance are encapsulated within a standard DR scan shift of the JTAG protocol.

The Instruction Register and Data Registers for the SLD nodes are a subset of the USER1 and USER0 Data Registers. Because the SLD Node IR/DR register set is not directly part of the IR/DR register set of the JTAG protocol, the SLD node Instruction Register and Data Register chains are known as Virtual IR (VIR) and Virtual DR (VDR) chains.

Figure 1–5 shows the transaction model of the SLD infrastructure.

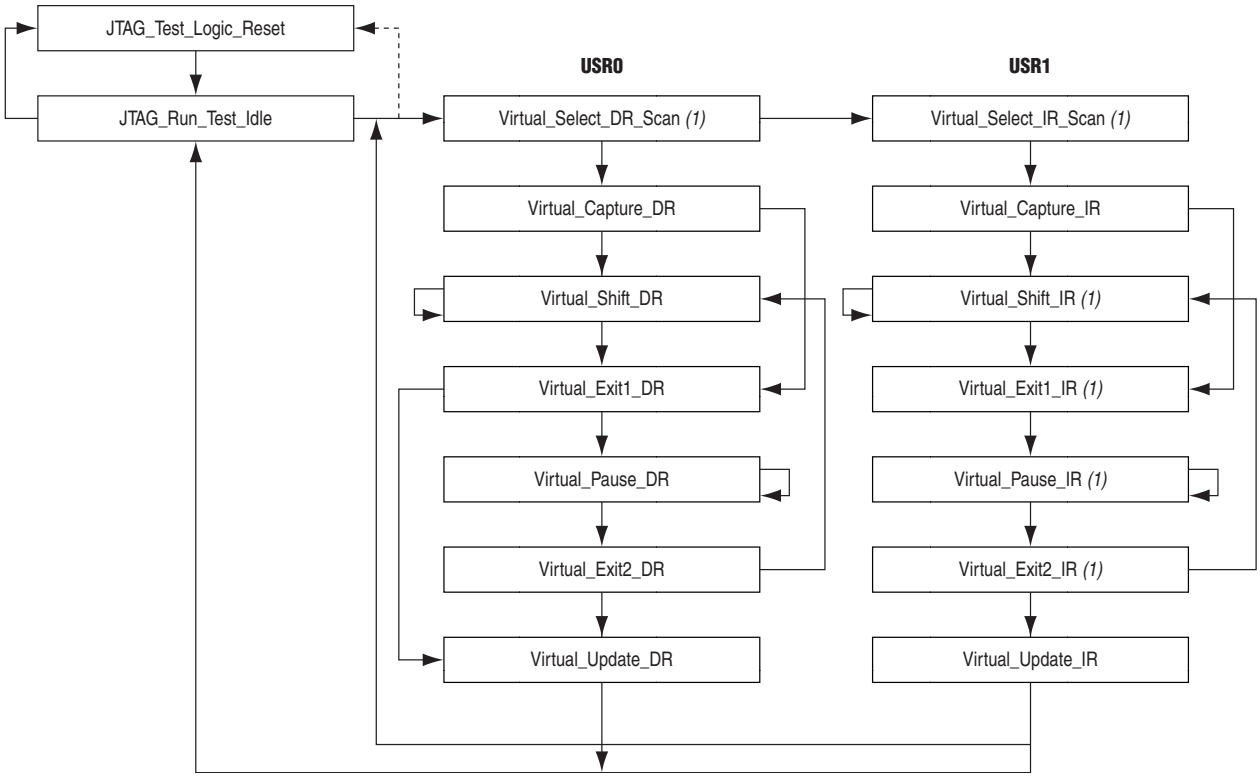
Figure 1–5. Extension of the JTAG Protocol for PLD Applications



The SLD hub decodes TMS independently from the hard JTAG TAP controller state machine and implements an equivalent state machine (called the "SLD hub finite state machine") for the internal JTAG path. The SLD hub performs a similar function for the VIR and VDR chains that the TAP controller performs for the JTAG IR and DR chains. It enables an SLD node as the active path for the TDI pin, selects the TDI data between the VIR and VDR registers, controls the start and stop of any shift transactions, and controls the data flow between the parallel hold registers and the parallel shift registers of the VIR and VDR.

Because all shifts to VIR and VDR are encapsulated within a DR shift transaction, an additional control signal is necessary to select between the VIR and VDR data paths. The SLD hub uses the USER1 command to select the VIR data path and the USER0 command to select the VDR data path. The SLD hub finite state machine is shown in Figure 1–6.

Figure 1–6. sld_hub Finite State Machine



Note to Figure 1–6:

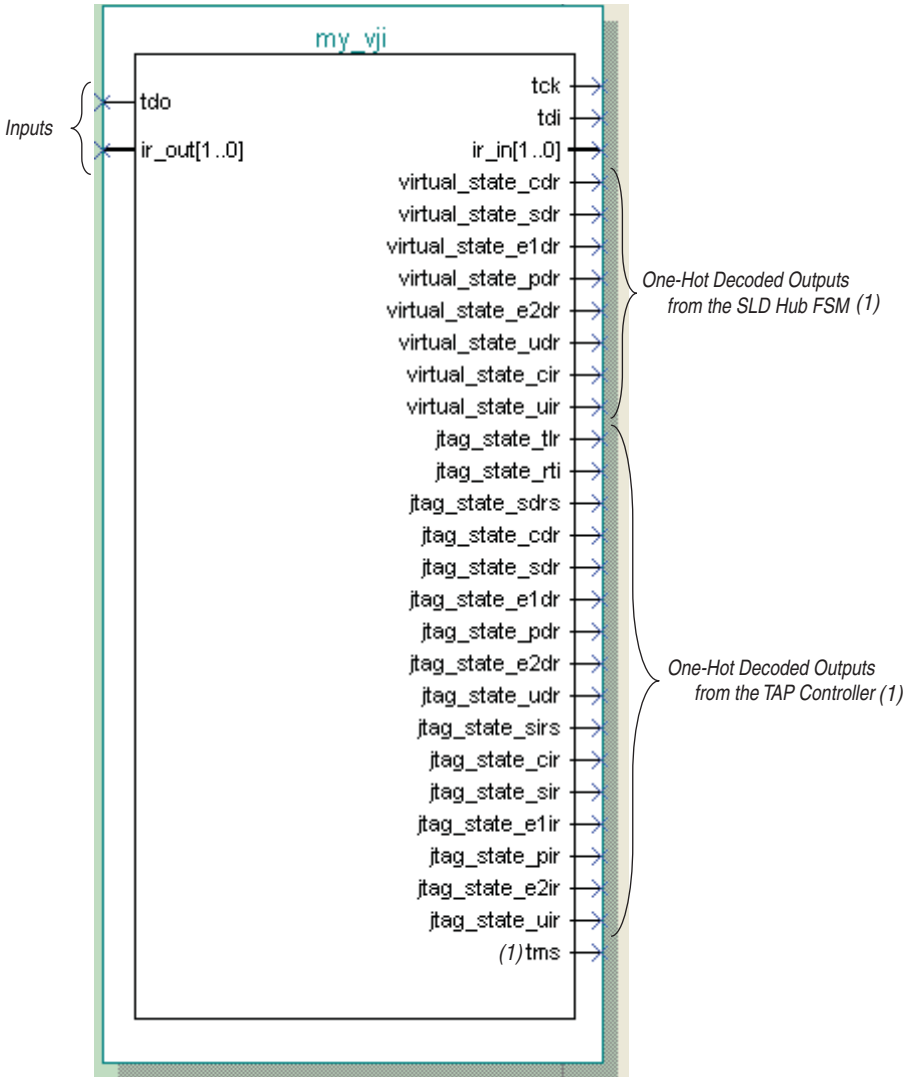
(1) There is no direct state signal available to be used for application design.

This state information, including a bank of enable signals, is forwarded to each of the SLD nodes. The SLD nodes perform the updates to the VIR and VDR according to the control states provided by the sld_hub. The SLD nodes are responsible for maintaining continuity between the TDI and TDO pins.

Description of the Virtual JTAG Interface (VJI)

The Virtual JTAG Interface (VJI) megafunction implements a SLD node interface. It provides a communication interface to the JTAG port. This megafunction exposes control signals that are part of the SLD hub; namely, JTAG port signals, all FSM controller states of the TAP controller, and the SLD hub finite state machine (FSM). Additionally, each instance of the Virtual JTAG megafunctions house the virtual Instruction Register for the SLD node. Instantiation of this megafunction automatically infers the SLD infrastructure. One SLD node is added for each instantiation of the Virtual JTAG megafunction. Refer to Figure 1–7.

Figure 1–7. Input and Output Ports of the Virtual JTAG Megafunction



Notes to Figure 1–7:

(1) The JTAG TAP controller outputs and TMS signals are used for informational purposes only. These signals can be exposed using the option **Create primitive JTAG state signal ports** on page 3 of the MegaWizard® Plug-In Manager.

The Virtual JTAG megafunction provides a port interface that mirrors the actual JTAG ports. The interface contains the JTAG port pins, a one-hot decoded output of all JTAG states, and a one-hot decoded output of all the virtual JTAG states. Virtual JTAG states are the states decoded from the SLD hub finite state machine. The `ir_in` and `ir_out` ports are the parallel input and output to and from the VIR. The VIR ports are used to select the active VDR datapath.

The JTAG states and `TMS` output ports are provided for debugging purposes only. Only the virtual JTAG, `TDI`, `TDO`, and the `IR` signals are functional elements of the megafunction. When configuring this megafunction using the MegaWizard Plug-In Manager, you can hide `TMS` and the decoded JTAG states.

Design Flow

Designing with the Virtual JTAG megafunction includes the following steps:

1. Configuring the Virtual JTAG megafunction with the desired Instruction Register length and instantiating the megafunction
2. Building glue logic for interfacing with your application
3. Communicating with the Virtual JTAG instance during runtime

In addition to the JTAG datapath and control signals, the Virtual JTAG megafunction encompasses the VIR. The Instruction Register size is configured in the MegaWizard Plug-In Manager. The Instruction Register port on the Virtual JTAG megafunction is the parallel output of the VIR. Any updated VIR information can be read off of this port after the `virtual_state_uir` signal is asserted.

For more information about these SLD hub finite state machine controller states, refer to the Quartus II Help.

After instantiating the megafunction, you must create the VDR chains that interface with your application. To do this, use the virtual instruction output to determine which VDR chain is the active datapath. You must create the following:

- Decode logic for the VIR
- VDR chains to which each VIR maps
- Interface logic between your VDR chains and your application logic

Your glue logic uses the decoded one-hot outputs from the megafunction to determine when to shift and when to update the VDR that you created. Your application logic interfaces with the VDR chains during any one of the non-shift virtual JTAG states.

For example, your application logic can parallel read an updated value that was shifted in from the JTAG port after the `virtual_state_uir` signal is asserted. If you load in a value to be shifted out of the JTAG port, you would do so when the `virtual_state_cdr` signal is asserted. Finally, if you enable the shift register to clock out information to `TDO`, you would do so during the assertion of `virtual_state_sdr`.

Maintaining TDI-to-TDO connectivity is important. Ensure that all possible instruction codes map to an active register chain to maintain connectivity in the TDI-to-TDO datapath. Altera recommends including a bypass register as the active register for all unmapped IR values.

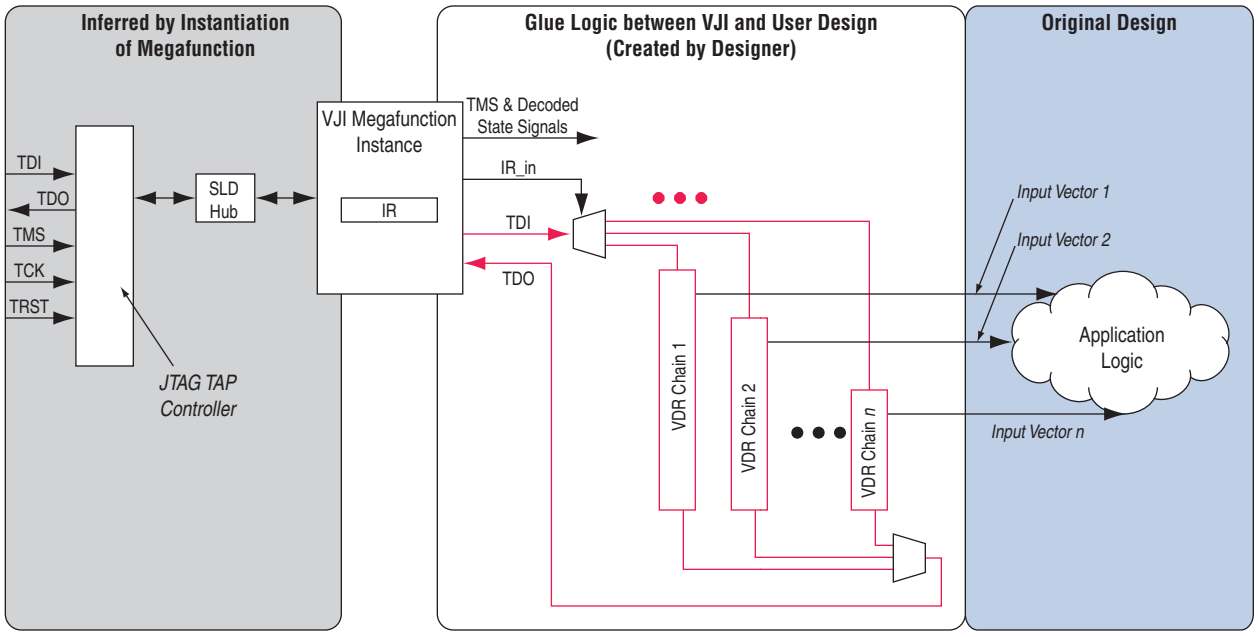
Note that TCK (a maximum 10-MHz clock, if using an Altera programming cable) provides the clock for the entire SLD infrastructure. Be sure to follow best practices for proper clock domain crossing between the JTAG clock domain and the rest of your application logic to avoid metastability issues. The decoded virtual JTAG state signals can help determine a stable output in the VIR and VDR chains.

After compiling and downloading your design into the device, you can perform shift operations directly to the VIR and VDR chains using the Tcl commands from the quartus_stp executable and an Altera programming cable (for example, a USB-Blaster™, a MasterBlaster™, or a ByteBlaster™ II cable). The quartus_stp executable is a command-line executable that contains Tcl commands for all on-chip debug features available in the Quartus II software design suite.

The section "Run-Time Communication with the Virtual JTAG Megafunction" on page 1–14 provides additional details about the specific Tcl commands that interface to this megafunction.

Figure 1–8 shows a block diagram of the components of a design containing one instance of the Virtual JTAG megafunction.

Figure 1–8. Block Diagram of a Design with a Single Virtual JTAG Instantiation *(Note 1)*, *(2)*, *(3)*



Notes to Figure 1–8:

(1) TDI-to-TDO datapath for the virtual JTAG chain, shown in red, consists of a bank of DR registers. Input to the application logic is the parallel output of the VDR chains.

(2) Decoded state signals used to signal start and stop of shift transactions and signals when the VDR output is ready.

(3) The IR_out port, not shown, is an optional input port you can use to parallel load the VIR from the FPGA core logic.

Complete application examples are provided in the section "Instantiating the Virtual JTAG Megafunction in Your Design" on page 2–3 and in the sections that describe "Design Example 1" on page 2–9 and "Design Example 2" on page 2–12.

Simulation Model

The virtual JTAG megafunction provides a functional simulation model. The simulation model provides stimuli that mimic VIR and VDR shifts. You configure the stimuli using the MegaWizard Plug-In Manager. The detailed steps for configuring the simulation model are provided in the section "Using the MegaWizard Plug-In Manager" on page 2–1.

This simulation model is used for functional verification only. The operation of the SLD hub and the SLD node-to-hub interface is not provided in this simulation model.

Run-Time Communication with the Virtual JTAG Megafunction

The Tcl API for the Virtual JTAG megafunction consists of a set of commands for accessing the VIR and VDR of each virtual JTAG instance.

These commands contain the underlying drivers for accessing an Altera programming cable and for issuing shift transactions to each VIR and VDR. Table 1–3 provides all of the Tcl commands in the `quartus_stp` executable that can be used with the Virtual JTAG megafunction.

Details about the underlying bit transactions can be found in Appendix A, SLD_NODE Discovery and Enumeration, and Appendix B, Capturing the Virtual IR Instruction Register. This information is intended for designs that use a custom controller to drive the JTAG chain.

Table 1–3. Tcl Commands Used with the Virtual JTAG Megafunction (Part 1 of 2)

Command	Arguments	Description

`Device_virtual_ir_shift`	`-instance_index <instance_index>`
	`-ir_value <numeric_ir_value>`
	`-no_captured_ir_value` *(1)*
	`-show_equivalent_device_ir_dr_shift` *(1)*
		Perform an IR shift operation to the virtual JTAG instance specified by the `instance_index`. Note that `ir_value` takes a numerical argument.

`Device_virtual_dr_shift`	`-instance_index <instance_index>`
	`-dr_value <dr_value>`
	`-length <data_register_length>`
	`-no_captured_dr_value` *(1)*
	`-show_equivalent_device_ir_dr_shift`
	`-value_in_hex` *(1)*
		Perform a DR shift operation to the virtual JTAG instance

`Get_hardware_names`	NONE	Queries for all available programming cables
`Open_device`	`-device_name <device_name>` `-hardware_name <hardware_name>`	Selects the active device on the JTAG chain
`Close_device`	NONE	Ends communication with the active JTAG device

Table 1–3. Tcl Commands Used with the Virtual JTAG Megafunction (Part 2 of 2)

Command	Arguments	Description
`Device_lock`	`-timeout <timeout>`	Obtains exclusive communication to the JTAG chain
`Device_unlock`	NONE	Releases `device_lock`
`Device_ir_shift`	`-ir_value <ir_value>` `-no_captured_ir_value`	Performs a IR shift operation

`Device_dr_shift`	`-dr_value <dr_value>`
	`-length <data register length>`
	`-no_captured_dr_value`
	`-value_in_hex`
		Performs a DR shift operation

Note to Table 1–3:

(1) This argument is optional.

For detailed information about each of these commands, refer to the *Quartus II Scripting Reference Manual* or the Quartus II Help.

Each instantiation of the Virtual JTAG megafunction includes an instance index. All instances are sequentially numbered and are automatically provided by the Quartus II software. The instance index starts at instance index `0`. The VIR and VDR shift commands described in Table 1–3 decode the instance index and provide an address to the SLD hub for each megafunction instance. You can override the default index provided by the Quartus II software during configuration of the megafunction.

Central to virtual JTAG megafunction are the `device_virtual_ir_shift` and `device_virtual_dr_shift` commands. These two commands perform the shift operation to each VIR/VDR and provide the address to the SLD hub for the active JTAG datapath.

Each `device_virtual_ir_shift` command issues a USER1 instruction to the JTAG Instruction Register followed by a DR shift containing the VIR value provided by the `ir_value` argument prepended by address bits to target the correct SLD node instance.

Use the `-no_captured_ir_value` argument if you do not care about shifting out the contents of the current VIR value. Enabling this option speeds up the VIR shift transaction by eliminating a command cycle within the underlying transaction.

Similarly, each `device_virtual_dr_shift` command issues a USER0 instruction to the JTAG Instruction Register followed by a DR shift containing the VDR value provided by the `dr_value` argument. These commands return the underlying JTAG transactions with the `show_equivalent_device_ir_dr_shift` option set.

The `device_virtual_ir_shift` takes the `ir_value` argument as a numeric value. The `device_virtual_dr_shift` takes the `dr_value` argument by either a binary string or a hexadecimal string. Do not use numeric values for the `device_virtual_dr_shift`.

A simple DR shift operation through a virtual JTAG chain using an Altera download cable consists of the following steps:

1. Query for the Altera programming cable and select the active cable.
2. Target the desired device in the JTAG chain.
3. Obtain a device lock for exclusive communication to the device.
4. Perform a VIR shift.
5. Perform a VDR shift.
6. Release exclusive link with the device with the `device_unlock` command.
7. Close communication with the device with the `close_device` command.

The *Quartus II Scripting Reference Manual*,"Design Example 1" on page 2–9 and "Design Example 2" on page 2–12 include example Tcl scripts for communicating with Virtual JTAG megafunction instances.

Run-Time Communication without Using an Altera Programming Cable

The Virtual JTAG megafunction Tcl API requires an Altera programming cable. Designs that use a custom controller to drive the JTAG chain directly must issue the correct JTAG IR/DR transactions to target the Virtual JTAG megafunction instances. The address values and register length information for each Virtual JTAG megafunction instance are provided in the compilation reports.

Figure 1–9 shows the compilation report for a sample Virtual JTAG Megafunction Instance and Table 1–4 describes each of the columns in the Virtual JTAG Settings compilation report.

Figure 1–9. Compilation Report for Virtual JTAG Megafunction Instances

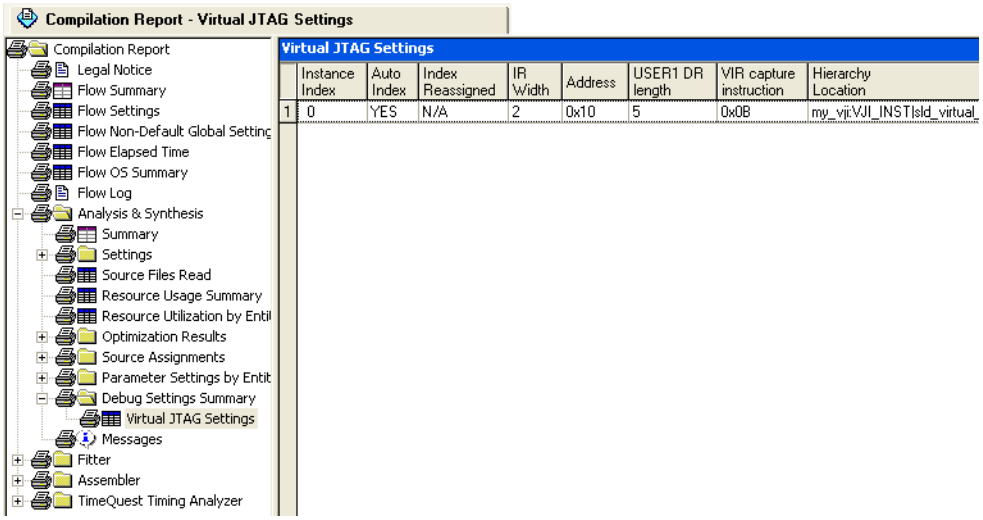


Table 1–4. Virtual JTAG Settings Description

Setting	Description
Instance Index	Instance index of the virtual JTAG megafunction. Assigned at compile time.
Auto Index	Details whether the index was auto-assigned.
Index Re-Assigned	Details whether the index was user-assigned.
IR Width	Length of the Virtual IR register for this megafunction instance; defined in the MegaWizard Plug-In Manager.
Address	The address value assigned to the megafunction by the compiler.
USER1 DR Length	The length of the USER1 DR register. The USER1 DR register encapsulates the VIR for all SLD nodes.
VIR Capture Instruction	Instruction value to capture the VIR of this megafunction instance.

The Tcl API provides a way to return the JTAG IR/DR transactions by using the `show_equivalent_device_ir_dr_shift` argument with the `device_virtual_ir_shift` and `device_virtual_dr_shift` commands. The following examples use returned values of a virtual IR/DR shift to illustrate the format of the underlying transactions.

Refer to the two separate examples in this user guide: "Design Example 1" on page 2–9, which is a Virtual IR/DR shift with the `-no_captured_ir_value` argument, and "Design Example 2" on page 2–12, a Virtual IR/DR shift without the `-no_captured_ir_value` argument.

To use the Tcl API to query for the bit pattern in your design, use the `-show_equivalent_device_ir_dr_shift` argument with the `device_virtual_ir_shift` and `device_virtual_dr_shift` commands.

Both examples are from the same design, with a single Virtual JTAG instance. The VIR length for the reference Virtual JTAG instance is configured to 3 bits in length.

Virtual IR/DR Shift Transaction without Returning Captured IR/DR Values

The Tcl commands in Example 1–1 and Example 1–2 show a VIR/VDR transaction with `no_captured_value` option set. The commands return the underlying JTAG shift transactions that occur. Figure 1–10 shows the bit values and fields associated with the VIR/VDR scans.

Example 1–1. Virtual IR Shift with the no_captured_value Option

```
device_virtual_ir_shift -instance_index 0 -ir_value 1 \
-no_captured_ir_value -show_equivalent_device_ir_dr_shift ↵
```

Returns:

```
Info: Equivalent device ir and dr shift commands
Info: device_ir_shift -ir_value 14
Info: device_dr_shift -length 5 -dr_value 11 -value_in_hex
```

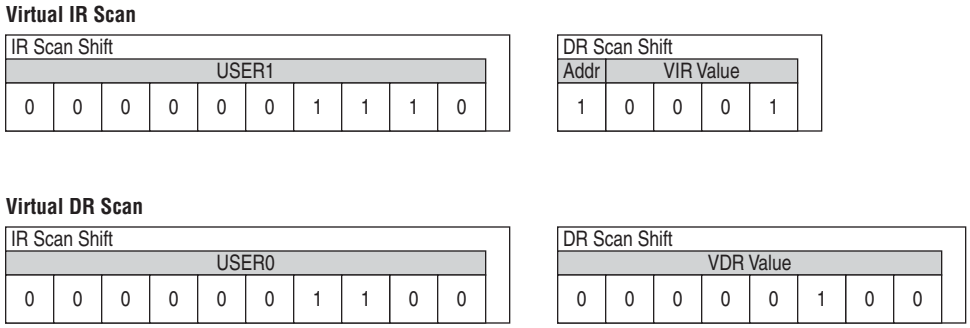
Example 1–2. Virtual DR Shift with the no_captured_value Option

```
device_virtual_dr_shift -instance_index 0 -length 8 -dr_value \
04 -value_in_hex -no_captured_dr_value \
-show_equivalent_device_ir_dr_shift ↵
```

Returns:

```
Info: Equivalent device ir and dr shift commands
Info: device_ir_shift -ir_value 12
Info: device_dr_shift -length 8 -dr_value 04 -value_in_hex
```

Figure 1–10. Equivalent Bit Pattern Shifted into Device by VIR/VDR Shift Commands *(Note 1)*, *(2)*, *(3)*



Notes to Figure 1–10:

(1) The Instruction Register length for all Altera FPGAs and CPLDs is 10 bits long.
(2) The USER1 value is 0x0E and USER0 value is 0x0C for all Altera FPGAs and CPLDs.
(3) The Address bits contained in the DR scan shift of a VIR scan are determined by the Quartus II software.

VIR shifts consist of a USER1 (0x0E) IR shift followed by a DR shift to the virtual Instruction Register. The length and value of the DR shift is dependent on the number of SLD nodes in your design. This value consists of address bits to the SLD node instance concatenated with the desired value of the virtual Instruction Register. The addressing scheme is determined by the Quartus II software during design compilation.

The VIR value field in Figure 1–10 is four bits long, even though the VIR length is configured to be three bits long. All USER1 DR chains must be of uniform length. The length of the VIR value field length is determined by length of the longest VIR register for all SLD nodes instantiated in the design. Because the SLD hub VIR is four bits long, the minimum length for the VIR value field for all SLD nodes in the design is at least four bits in length. The Quartus II Tcl API automatically sizes the shift transaction to the correct length. The length of the VIR register is provided in the Virtual JTAG settings compilation report. If you are driving the JTAG interface with a custom controller, you must pay attention to size of the USER1 DR chain through the information Virtual JTAG Settings Table, the values returned by the Tcl API, or via the information provided in Appendix A, SLD_NODE Discovery and Enumeration.

The VDR shifts consist of a USER0 0x0C IR shift followed by a DR shift to the virtual Data Register. The DR Scan shift consists of the value passed by the `-dr_value` argument.

Virtual IR/DR Shift Transaction that Captures Current VIR/VDR Values

The Tcl commands in Example 1–3 and Example 1–4 show examples in which the `no_captured_value` option is not set in the Virtual IR/DR shift commands and the underlying JTAG shift commands associated with each. In the VIR shift command, the command returns two `device_dr_shift` commands. Figure 1–11 further details the transaction.

Example 1–3. Virtual IR Shift

```
device_virtual_ir_shift -instance_index 0 -ir_value 1 \
-no_captured_ir_value -show_equivalent_device_ir_dr_shift
```

Returns:

```
Info: Equivalent device ir and dr shift commands
Info: device_ir_shift -ir_value 14
Info: device_dr_shift –length 5 –dr_value 0B –value_in_hex
Info: device_dr_shift -length 5 -dr_value 11 -value_in_hex
```

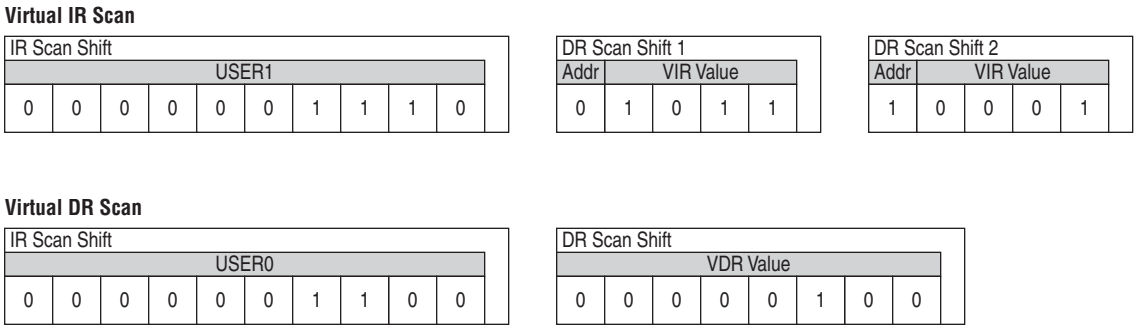
Example 1–4. Virtual DR Shift

```
device_virtual_dr_shift -instance_index 0 -length 8 -dr_value \
04 -value_in_hex -show_equivalent_device_ir_dr_shift
```

Returns:

```
Info: Equivalent device ir and dr shift commands
Info: device_ir_shift -ir_value 12
Info: device_dr_shift -length 8 -dr_value 04 -value_in_hex
```

Figure 1–11. Equivalent Bit Pattern Shifted into Device by VIR/VDR Shift Commands with Captured IR Values *(Note 1), (2)*



Notes to Figure 1–11:

(1) DR Scan Shift 1 targets the SLD hub VIR to force a captured value from Virtual JTAG instance 1. This command is known as the `VIR_CAPTURE` command.

(2) DR Scan Shift 2 targets the VIR of Virtual JTAG instance 1.

DR Scan Shift 1 is the `VIR_CAPTURE` command (shown in Figure 1–11). It targets the VIR of the sld_hub. This command is an address cycle to select the active VIR chain to shift after `jtag_state_cir` is asserted. The `HUB_FORCE_IR` capture must be issued whenever you capture the VIR from a target SLD node that is different than the current active node. The precise definition of the `VIR_CAPTURE` instruction and the `HUB_FORCE_IR` instructions can be found in Appendix B, Capturing the Virtual IR Instruction Register.

Appendix A, SLD_NODE Discovery and Enumeration, describes in detail the discovery and enumeration process for all the SLD Nodes within a design. This information can be used to determine the address scheme for each of the SLD nodes dynamically within the SLD infrastructure.

If you are using an embedded processor as a controller for the JTAG chain and your Virtual JTAG megafunction instances, consider using the JAM Standard Test and Programming Language (STAPL). JAM STAPL is an industry-standard flow-control-based language that supports JTAG communication transactions. JAM STAPL is open source, with software downloads and source code available from the Altera website (www.altera.com).

For more information about JAM STAPL and using JAM STAPL with an embedded processor, refer to the following pages on www.altera.com:

- ISP & the Jam STAPL
- Embedded Programming With Jam STAPL

Reset Considerations when Using a Custom JTAG Controller

The SLD hub decodes TMS independently to determine the JTAG controller state. Under normal operation, the SLD hub mirrors all of the JTAG TAP controller states accurately. The JTAG pins (TCK, TMS, TDI, and TDO) are accessible to the core programmable logic; however, the JTAG TAP controller outputs are not visible to the core programmable logic. In addition, the hard JTAG TAP controller does not use any reset signals as inputs from the core programmable logic. This can cause two situations in which control states of the SLD hub and the JTAG TAP controller are not in lock-step:

- An assertion of the device wide global reset signal (DEV_CLRn)
- An assertion of the TRST signal, if available on the device

DEV_CLRn resets the SLD hub but does not reset the hard TAP controller block. The TAP controller is meant to be decoupled from USER mode device operation to run boundary scan operations. Asserting the global reset signal does not disrupt boundary-scan test (BST) operation.

Conversely, the TRST signal, if available, resets the JTAG TAP controller but does not reset the SLD hub. The TRST signal does not route into the core programmable logic of the PLD.

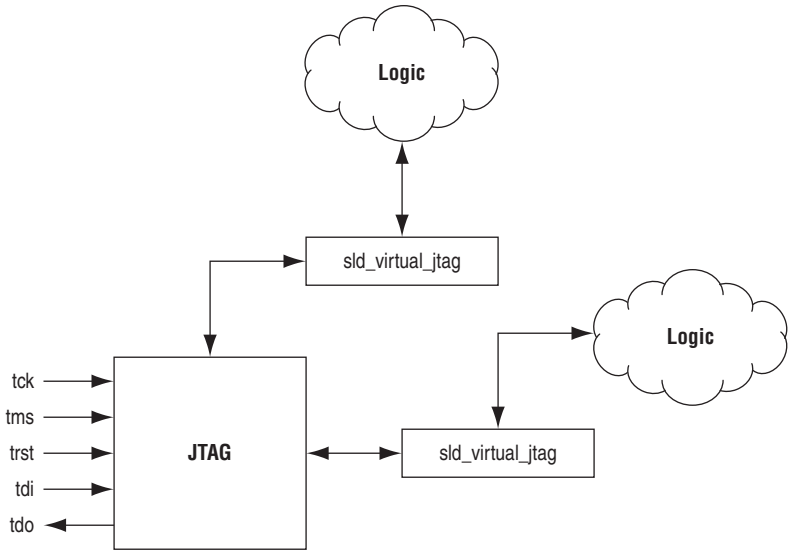
To guarantee that the states of the JTAG TAP controller and the SLD hub state machine are properly synchronized, TMS should be held high for at least five clock cycles after any intentional reset of either the TAP controller or the system logic. Both the JTAG TAP controller and the sld_hub controller are guaranteed to be in the Test Logic Reset state after five clock cycles of TMS held high.

Applications

Single or multiple instances of the Virtual JTAG megafunction can be instantiated in your HDL code. During synthesis, the Quartus II software assigns unique IDs to each instance, so each can be accessed individually. You can instantiate up to 128 instances of the Virtual JTAG megafunction.

Figure 1–12 shows a typical application in a design with multiple instances of the Virtual JTAG megafunction.

Figure 1–12. Application Example



The SLD hub automatically arbitrates between multiple applications that share a single JTAG resource. As such, the Virtual JTAG megafunction can be used in tandem with other on-chip debugging applications (such as the SignalTap II Embedded Logic Analyzer), to increase debugging visibility. The Virtual JTAG megafunction can be used to provide simple stimulus patterns to solicit a response from the design under test during run-time.

The Virtual JTAG megafunction can be used in many applications, including the following circumstances:

- To diagnose, sample, and update the values of internal parts of your logic. With this megafunction, you can easily sample and update the values of the internal counters and state machines in your hardware device.
- You can build your own custom software debugging IP using the Tcl commands listed above to debug your hardware. This IP communicates with the instances of the Virtual JTAG megafunction inside your design.
- You can instrument your design to achieve virtual inputs and outputs in your design.
- If you are building a debugging solution for a system in which a microprocessor controls the JTAG chain, the SignalTap II Embedded Logic Analyzer cannot be used because the JTAG control has to be with the microprocessor. By learning the low-level controls for the JTAG port from the Tcl commands, you can program microprocessors to communicate with the Virtual JTAG megafunction inside the device core.

System and Software Requirements

The instructions in this section require the following hardware and software:

- The Quartus® II software beginning with version 6.0
- An Altera® download cable, such as a USB-Blaster™ cable

The download cable is required to communicate with the Virtual JTAG megafunction from a host running the `quartus_stp` executable.

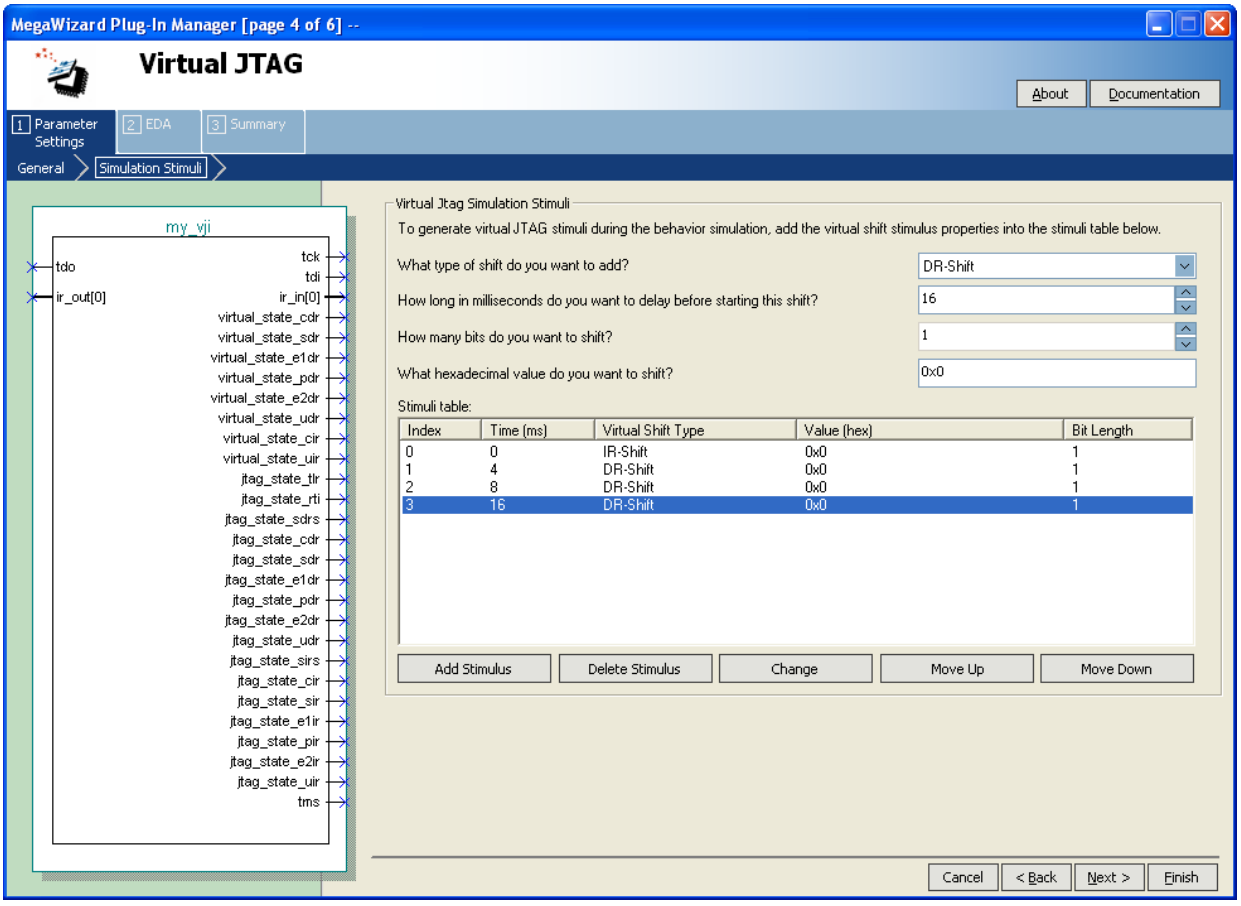
Using the MegaWizard Plug-In Manager

To create the Virtual JTAG megafunction in your design, you must use the MegaWizard® Plug-In Manager within the Quartus II software. Perform the following steps to generate the megafunction:

1. On the Tools menu, click **MegaWizard Plug-In Manager**. The **MegaWizard Plug-In Manager** dialog box appears.
2. Select **Create a new custom megafunction variation**.
3. Click **Next**. Page 2a of the MegaWizard Plug-In Manager appears.
4. In the list of megafunctions, click the "+" icon to expand the **JTAG-accessible Extensions** folder. Click **Virtual JTAG**.
5. Select the device family you are using.
6. Select the type of output file you want to create: Verilog HDL, VHDL, or AHDL.
7. Specify the name of the output file and its location.
8. Click **Next**. Page 3 of the MegaWizard Plug-In Manager appears.
9. Select the width (number of bits) of your Instruction Register.
10. Assign a unique ID to the instance of your Virtual JTAG megafunction. The wizard can assign an ID automatically (recommended), or you can enter one manually.

11. Click **Next**. Page 4 of the MegaWizard Plug-In Manager appears (Figure 2–1).

Figure 2–1. MegaWizard Plug-In Manager, Page 4



12. Page 4 defines the stimuli that are used during the simulation of your megafunction. A stimulus is either a Data Register shift (DR shift) or an Instruction Register shift (IR shift). Each stimulus requires a time at which that shift occurs, the number of bits you want to shift in or out, and the data value you want to shift in during a shift-in operation. You can add multiple stimuli by clicking the **Add Stimulus** button.

    The stimuli specified on Page 4 of the wizard are written to the variation file. If you want to change a stimulus after creating the megafunction, you can either edit the variation file or create the megafunction again with a new stimulus. The wizard provides an easy way to generate your stimuli. If you do not want to generate the stimuli, you can skip this step. However, the stimuli are necessary if you are performing simulation of your design.

13. Click **Next**. Page 5 of the MegaWizard Plug-In Manager appears. In this example, the page shows that you need the **altera_mf** library to simulate the Virtual JTAG megafunction in your design.

    There is no input required from you on this page.

14. Click **Next**. Page 6 of the MegaWizard Plug-In Manager appears.

15. Select any other files you need in addition to the megafunction variation file and the megafunction black box file.
16. Click **Finish** to create the Virtual JTAG megafunction. This action creates the files you need in your project.

The output from the MegaWizard Plug-In Manager is a variation file. Example output files for the Virtual JTAG megafunction variation file are provided in Appendix B, Capturing the Virtual IR Instruction Register.

Instantiating the Virtual JTAG Megafunction in Your Design

To properly connect the Virtual JTAG megafunction in your design, you should follow these basic connection rules:

- The `tck` output from the Virtual JTAG megafunction is the clock used for shifting the data in and out on the `TDI` and `TDO` pins.
- The `TMS` output of the Virtual JTAG megafunction reflects the `TMS` input to the main JTAG circuit.
- The `ir_in` output port of the Virtual JTAG megafunction is the parallel output of the contents that get shifted into the virtual IR of the Virtual JTAG instance. This port is used for decoding logic to select the active virtual DR chain.

You can use the following Verilog HDL template as a guide for instantiating and connecting various signals of the megafunctions in your design.

The purpose of instantiating a Virtual JTAG instance in this example is to load `my_counter` through the JTAG port using a software application built with Tcl commands and the `quartus_stp` executable. In this design, the Virtual JTAG instance is called `my_vji`. Whenever a Virtual JTAG megafunction is instantiated in a design, three logic blocks are usually needed: a decode logic block, a `TDO` logic block, and a Data Register block. Example 2–1 combines the Virtual JTAG instance, the decode logic, the `TDO` logic and the Data Register blocks.

Example 2–1. (Part 1 of 2)

```
module        counter (clock, my_counter);
input         clock;
output [3:0]  my_counter;
reg [3:0]     my_counter;
always @ (posedge clock)
   if (load && e1dr) // decode logic: used to load the counter my_counter
       my_counter <= tmp_reg;
   else
       my_counter <= my_counter + 1;
// Signals and registers declared for VJI instance
wire tck, tdi;
reg  tdo;
wire cdr, eldr, e2dr, pdr, sdr, udr, uir, cir;
wire [1:0] ir_in;
```

Example 2–1. (Part 2 of 2)

```
// Instantiation of VJI
my_vji  VJI_INST(
      .tdo (tdo),
      .tck (tck),
      .tdi (tdi),
      .tms(),
      .ir_in(ir_in),
      .ir_out(),
      .virtual_state_cdr (cdr),
      .virtual_state_e1dr(e1dr),
      .virtual_state_e2dr(e2dr),
      .virtual_state_pdr (pdr),
      .virtual_state_sdr (sdr),
      .virtual_state_udr (udr),
      .virtual_state_uir (uir),
      .virtual_state_cir (cir)
);
// Declaration of data register
reg [3:0] tmp_reg;
// Deocde Logic Block
// Making some decode logic from ir_in output port of VJI
wire load = ir_in[1] && ~ir_in[0];
// Bypass used to maintain the scan chain continuity for
// tdi and tdo ports

bypass_reg <= tdi;
// Data Register Block
always @ (posedge tck)
   if ( load && sdr )
      tmp_reg <= {tdi, tmp_reg[3:1]};
// tdo Logic Block
always @ (tmp_reg[0] or bypass_reg)
   if(load)
      tdo <= tmp_reg[0]
   else
      tdo <= bypass_reg;
endmodule
```

The decode logic is produced by defining a wire `load` to be active high whenever the IR of the Virtual JTAG megafunction is `01`. The IR scan shift is used to load the data into the IR of the Virtual JTAG megafunction. The `ir_in` output port reflects the IR contents.

The Data Register logic contains a 4-bit shift register named `tmp_reg`. The `always` blocks shown for the Data Register logic also contain the decode logic consisting of the `load` and `sdr` signals. The `sdr` signal is the output of the Virtual JTAG megafunction that is asserted high during a DR scan shift operation. The time during which the `sdr` output is asserted high is the time in which the data on `tdi` is valid. During that time period, the data is shifted into the `tmp_reg` shift register. Therefore, `tmp_reg` gets the data from the Virtual JTAG megafunction on the `tdi` output port during a DR scan operation.

There is a 1-bit register named `bypass_reg` whose output is connected with `tdo` logic to maintain the continuity of the scan chain during idle or IR scan shift operation of the Virtual JTAG megafunction. The `tdo` logic consists of outputs coming from `tmp_reg` and `bypass_reg` and connecting to the `tdo` input of the Virtual JTAG megafunction. The `tdo` logic passes the data from `tmp_reg` to the Virtual JTAG megafunction during DR scan shift operations.

The `always` block of a 4-bit counter also consists of some decode logic. This decode logic uses the `load` signal and `e1dr` output signal of the Virtual JTAG megafunction to load the counter with the contents of `tmp_reg`. The Virtual JTAG output signal `e1dr` is asserted high during a DR scan shift operation when all the data is completely shifted into the `tmp_reg` and `sdr` has been de-asserted. In addition to `sdr` and `e1dr`, there are other outputs from the Virtual JTAG megafunction that are asserted high to show various states of the TAP controller and internal states of the Virtual JTAG megafunction. All of these signals can be used to perform different logic operations as needed in your design. Figure 1–7 on page 1–11 shows all of the input and output ports of the Virtual JTAG megafunction.

Simulation Support

Virtual JTAG interface operations can be simulated using all Altera-supported simulators. The simulation support is for DR and IR scan shift operations. For simulation purposes, a behavioral simulation model of the megafunction is provided in both VHDL and Verilog HDL in the **altera_mf** libraries. The I/O structure of the model is the same as the megafunction.

In its implementation, the Virtual JTAG megafunction connects to your design on one side and to the JTAG port through the JTAG hub on the other side. However, a simulation model connects only to your design. There is no simulation model for the JTAG circuit. Therefore, no stimuli can be provided from the JTAG ports of the device to imitate the scan shift operations of the Virtual JTAG megafunction in simulation.

The scan operations in simulation are realized using the simulation model. The simulation model consists of a signal generator, a model of the SLD hub, and the Virtual JTAG model. The stimuli defined in the wizard are passed as parameters to this simulation model from the variation file. The simulation parameters are listed in Table 2–1. The signal generator then produces the necessary signals for Virtual JTAG megafunction outputs such as `tck`, `tdi`, `tms`, and so forth.

The model is parameterized to allow the simulation of an unlimited number of Virtual JTAG instances. The parameter `sld_sim_action` defines the strings used for IR and DR scan shifts. Each Virtual JTAG's variation file passes these parameters to the Virtual JTAG component. The Virtual JTAG's variation file can always be edited for generating different stimuli, though the preferred way to specify stimuli for DR and IR scan shifts is to use the MegaWizard Plug-In Manager.

Table 2–1. Description of Simulation Parameters (Part 1 of 2)

Parameter	Comments
SLD_SIM_N_SCAN	Specifies the number of shifts in the simulation model.

Table 2–1. Description of Simulation Parameters (Part 2 of 2)

Parameter	Comments
SLD_SIM_TOTAL_LENGTH	The total number of bits to be shifted in either an IR shift or a DR shift. This value should be equal to the sum of all the `length` values specified in the SLD_SIM_ACTION string.
SLD_SIM_ACTION	The string has the following format: ((time,type,value,length), (time,type,value,length), ... (time,type,value,length)) where: ■ **time**—A 32-bit value in milliseconds that represents the start time of the shift relative to the completion of the previous shift. ■ **type**—A 4-bit value that determines whether the shift is a DR shift or an IR shift. ■ **value**—The data associated with the shift. For IR shifts, it is a 32-bit value. For DR shifts, the length is determined by **length**. ■ **length**—A 32-bit value that specifies the length of the data being shifted. This value should be equal to SLD_NODE_IR_WIDTH; otherwise, the value field may be padded or truncated. 0 is invalid. Entries are in hexadecimal format.

Simulation has the following limitations:

- Scan shifts (IR or DR) must be at least 1 ms apart in simulation time.
- Only behavioral or functional level simulation support is present for this megafunction. There is no gate level or timing level simulation support.
- For behavioral simulation, the stimuli tell the signal generator model in the Virtual JTAG model to generate the sequence of signals needed to produce the necessary outputs for `tck`, `tms`, `tdi`, and so forth. You cannot provide the stimulus at the JTAG pins of the device.
- The `tck` clock period used in simulation is 10 MHz with a 50% duty cycle. In hardware, the period of the `tck` clock cycle may vary.
- In a real system, each instance of the Virtual JTAG megafunction works independently. In simulation, multiple instances can work at the same time. For example, if you define a scan shift for Virtual JTAG instance number 1 to happen at 3 ms and a scan shift for Virtual JTAG instance number 2 to happen at the same time, the simulation works correctly.

If you are using the ModelSim-Altera simulator, the **altera_mf.v** and **altera_mf.vhd** libraries are provided in precompiled form with the simulator.

The inputs and outputs of the Virtual JTAG megafunction during a typical IR scan shift operation are shown in Figure 2–2.

Figure 2–2. IR Shift Waveform

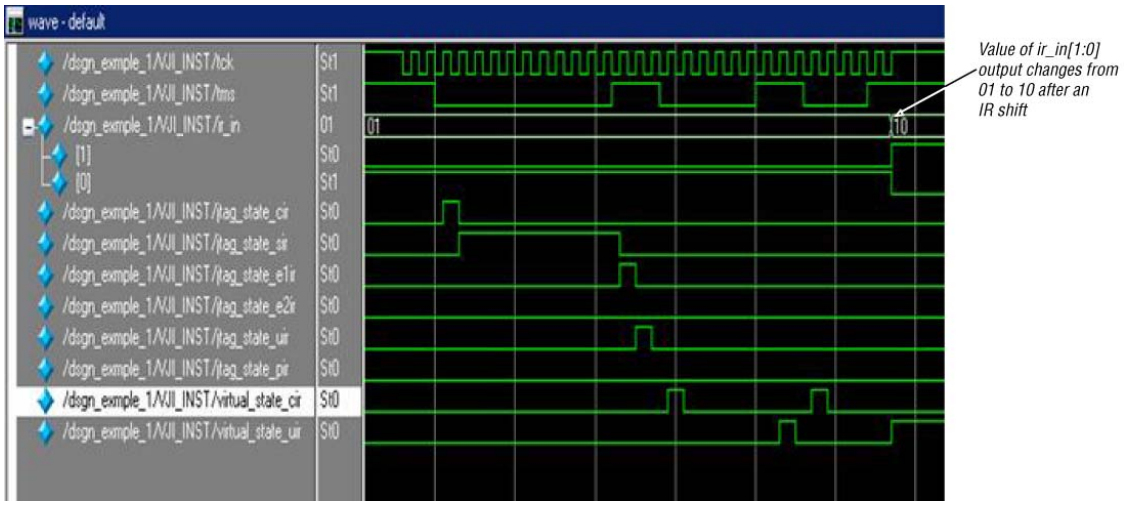
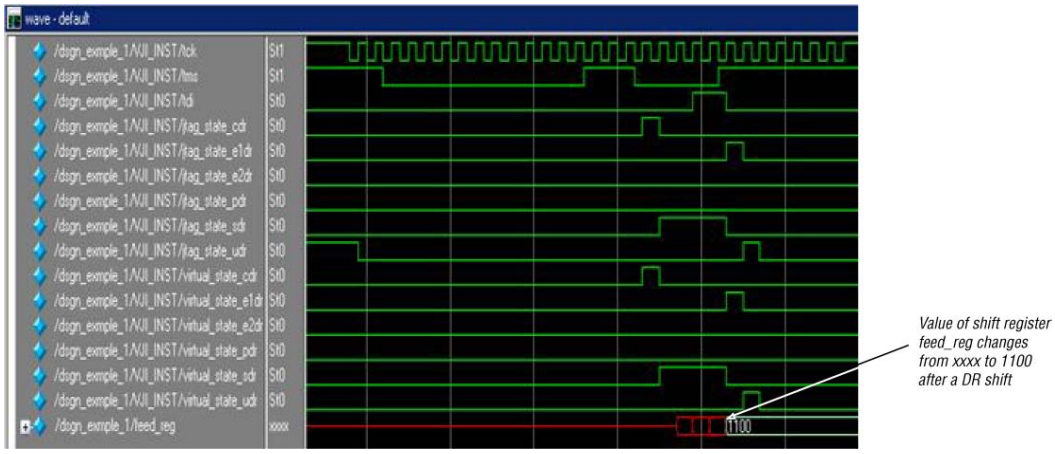


Figure 2–3 shows the inputs and outputs of the Virtual JTAG megafunction during a typical DR scan shift operation.

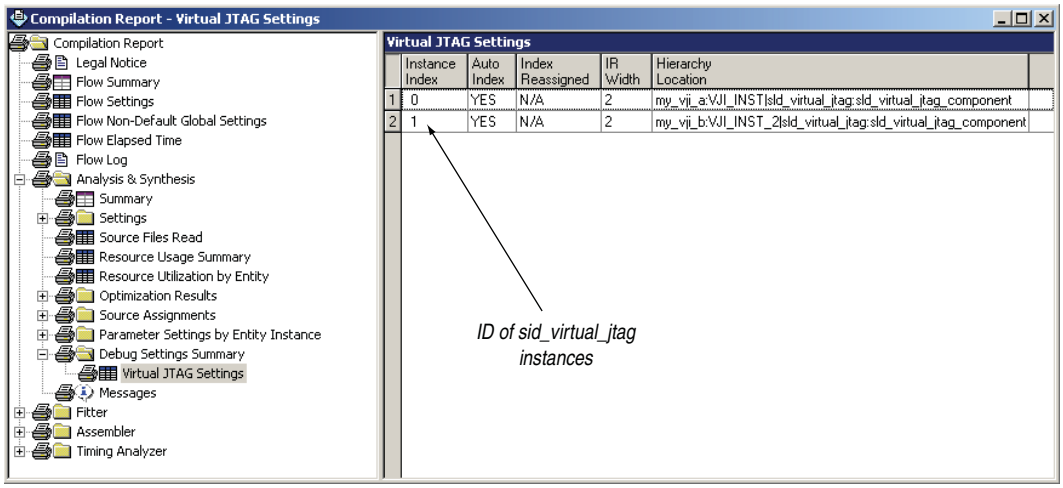
Figure 2–3. DR Shift Waveform



Compiling the Design

You can instantiate a maximum of 128 instances of the Virtual JTAG megafunction. After compilation, each instance has a unique ID, as shown on the **Virtual JTAG Settings** page of the Analysis & Synthesis section of the Compilation Report (Figure 2–4).

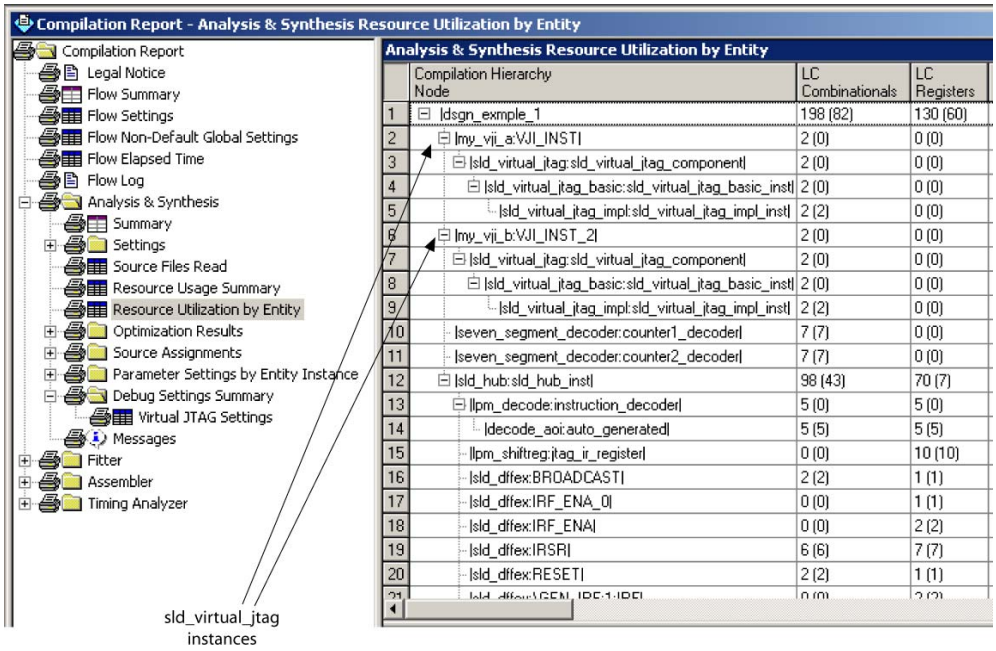
Figure 2–4. IDs of Virtual JTAG Instances



These unique IDs are necessary for Quartus II Tcl API to properly address each instance of the megafunction.

The addition of Virtual JTAG megafunctions uses logic resources in your design. The Fitter Resource Section in the Compilation Report shows the logic resource utilization (Figure 2–5).

Figure 2–5. Logic Resources Utilized



For more information about compiling designs with the Quartus II software or compilation flows, refer to *Volume 2: Design Implementation and Optimization*, and *Volume 3: Verification* of the *Quartus II Handbook*.

Third-Party Synthesis Support

In addition to the variation file, the MegaWizard Plug-In Manager creates a black box file for the Virtual JTAG megafunction you created. For example, if you create a **my_vji.v** file, a **my_vji_bb.v** file is also created. In third-party synthesis, you include this black box file along with your design files to synthesize your project. A VQM file is usually produced by third-party synthesis tools. This VQM netlist and the Virtual JTAG megafunction's variation files are input to the Quartus II software for further compilation.

Design Example 1

The Tcl API that ships with the Virtual JTAG megafunction makes it an ideal solution for developing command-line scripts that can be used to either update data values or toggle control bits at run time. This visibility into the FPGA can help expedite debug closure during the prototyping phase of the design, especially when external equipment is not available to provide a stimulus. This design example demonstrates the use of the Virtual JTAG megafunction and a command-line script to dynamically modify the contents of a DCFIFO at runtime.

The files for this design example are located in the User Guide page in the Literature section of the Altera website (www.altera.com). The design files are targeted to a Cyclone III starter kit.

This design example consists of a Quartus II project file that implements a DCFIFO and a command-line script that is used to modify the contents of the FIFO at runtime.

The RTL consists of a single instantiation of the Virtual JTAG megafunction to communicate with the JTAG circuitry. Both read and write ports of the DCFIFO are clocked at 50 MHz. A SignalTap II Embedded Logic Analyzer instance taps the data output bus of the DCFIFO to read burst transactions from the DCFIFO. The following sections discuss the RTL implementation and the runtime control of the DCFIFO using the Tcl API.

Write Logic

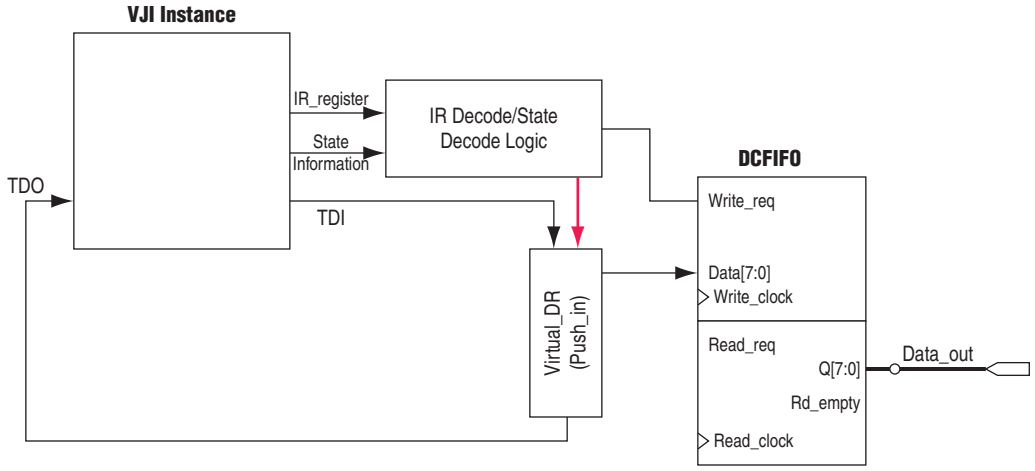
Figure 2–6 describes the implementation for the write side logic for this design example. The RTL uses a single instance of the Virtual JTAG megafunction to decode both the instructions for the write side and read side logic. The IR register is three bits wide, with the three instructions decoded in the RTL, as shown in Table 2–2.

Table 2–2. Instruction Register Values Used in Design Example 1

Instruction Register Value	Function
PUSH	Instruction to write a single value to the write side logic of the DCFIFO
POP	Instruction to read a single value from the read side logic of the DCFIFO
FLUSH	Instruction to perform a burst read transaction from the FIFO until empty.

The IR decode logic shifts the `Push_in` virtual DR chain when the PUSH instruction is on the IR port and `virtual_state_sdr` is asserted. A write enable pulse, synchronized to the `write_clock`, asserts after the `virtual_state_udr` signal goes high. The `virtual_state_udr` signal guarantees stability from the virtual DR chain.

Figure 2–6. Write Side Logic for DCFIFO Design Example

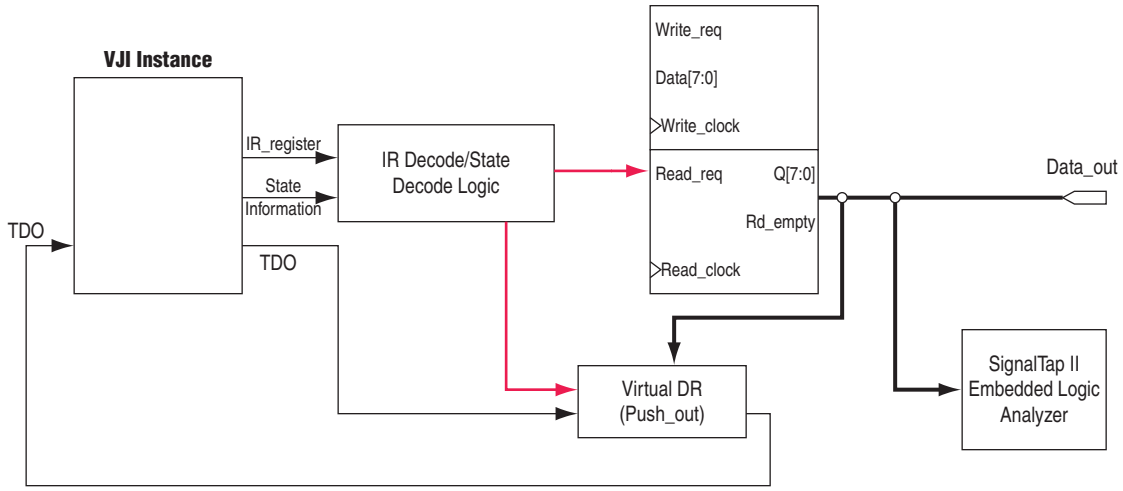


Read Logic

Figure 2–7 describes the implementation of the read side logic for this design example. There are two runtime instructions that this example implements for reading contents out of the FIFO. The IR decode logic selects the `Push_out` virtual DR chain and generates a single read pulse to the read logic when the POP instruction is active. The `Push_out` DR chain is parallel loaded upon the assertion of `virtual_state_cdr` and shifted out to TDO upon the assertion of `virtual_state_sdr`.

When the FLUSH instruction is shifted into the Virtual JTAG instance, the IR decode logic asserts the `read_req` line until the FIFO is empty. The bypass register is selected when the `FLUSH` instruction is active to maintain TDI-to-TDO connectivity.

Figure 2–7. Read Side Logic for DCFIFO Design Example



Runtime Communication

The Tcl script, **dc_fifo_vji.tcl**, contains three procedures—each corresponding to one of the virtual JTAG instructions. Table 2–3 describes each of the procedures.

Table 2–3. Example 2–1 Run-Time Communication Tcl Procedures

Procedure	Description
push [value]	IR shift the PUSH instruction, followed by a DR shift of the value argument. Value must be an integer less than 256.
pop	IR shift the POP instruction, followed by a DR shift of 8 bits.
flushfifo	IR shift the FLUSH instruction.

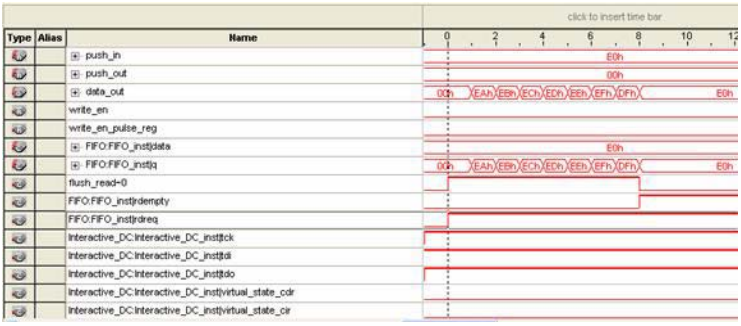
Figure 2–8 shows runtime execution of eight values pushed into the DCFIFO and a `flushfifo` command. Figure 2–9 shows a SignalTap II Embedded Logic Analyzer capture triggering on a flush operation.

Figure 2–8. Runtime Execution, Example 1

```
C:\Virtual_JTAG\example1\DC_FIFO_VJI_restored>quartus_stp -s
Info: ******************************************************************
Info: Running Quartus II SignalTap II
    Info: Version 8.0 Build 215 05/29/2008 SJ Full Version
    Info: Copyright (C) 1991-2008 Altera Corporation. All rights reserved.
    Info: Your use of Altera Corporation's design tools, logic functions
    Info: and other software and tools, and its AMPP partner logic
    Info: functions, and any output files from any of the foregoing
    Info: (including device programming or simulation files), and any
    Info: associated documentation or information are expressly subject
    Info: to the terms and conditions of the Altera Program License
    Info: Subscription Agreement, Altera MegaCore Function License
    Info: Agreement, or other applicable license agreement, including,
    Info: without limitation, that your use is for the sole purpose of
    Info: programming logic devices manufactured by Altera and sold by
    Info: Altera or its authorized distributors.  Please refer to the
    Info: applicable agreement for further details.
    Info: Processing started: Fri Jul 25 17:40:25 2008
Info: ******************************************************************
Info: The Quartus II Shell supports all TCL commands in addition
Info: to Quartus II Tcl commands. All unrecognized commands are
Info: assumed to be external and are run using Tcl's "exec"
Info: command.
Info: - Type "exit" to exit.
Info: - Type "help" to view a list of Quartus II Tcl packages.
Info: - Type "help <package name>" to view a list of Tcl commands
Info:   available for the specified Quartus II Tcl package.
Info: - Type "help -tcl" to get an overview on Quartus II Tcl usages.
Info: ******************************************************************

tcl> source dc_fifo_vji.tcl
tcl> push 234
11101010
tcl> push 235
11101011
tcl> push 236
11101100
tcl> push 237
11101101
tcl> push 238
11101110
tcl> push 239
11101111
tcl> push 223
11011111
tcl> push 224
11100000
tcl> flushfifo
```

Figure 2–9. SignalTap II Embedded Logic Analyzer Capture Triggering on a Flush Operation

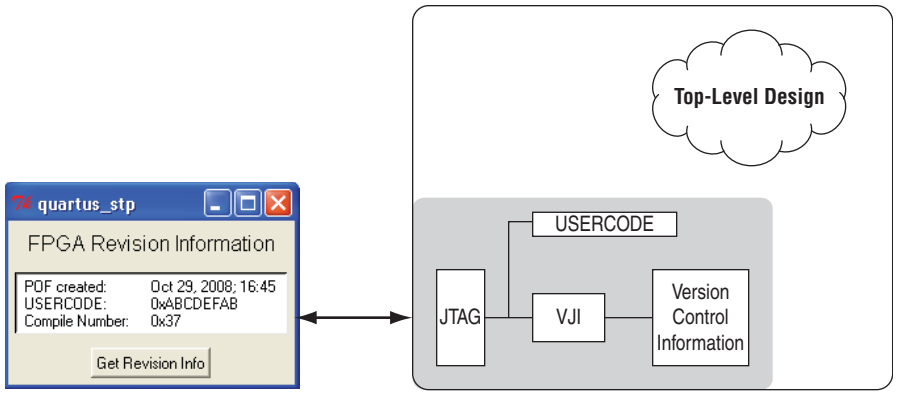


Design Example 2

Because the Quartus II software ships with an installation of Tcl/Tk, you can use the Tk package to build a custom GUI to interact with your design. In many cases, the JTAG port is a convenient interface to use, since it is present in most designs for debug purposes. By leveraging Tk and the virtual JTAG interface, you perform rapid prototyping such as creating virtual front panels or creating simple software applications.

This second example demonstrates a simple example where a GUI is used to offload revision information that is hardwired into a design. The GUI offloads the time that the design was compiled, the USERCODE from the device, and compile number that tracks the number of compile iterations that have been performed. Figure 2–10 shows the organization of the design.

Figure 2–10. Version Control Design Example Using the Virtual JTAG Megafunction

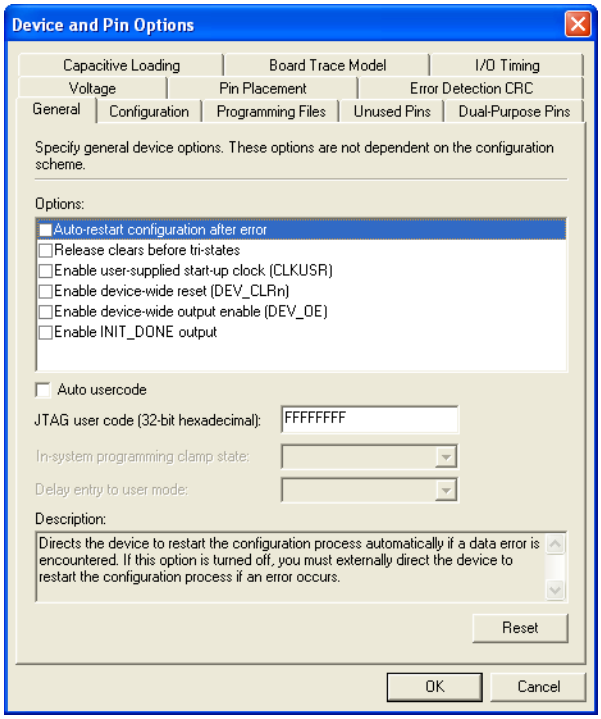


A Tcl script creates and updates the verilog file containing the hard-coded version control information every time the project goes through a full compile. The Tcl script is executed automatically by adding the following assignment to the project's **.qsf** file.

The USERCODE value shifted out by this design example is a user-configurable 32-bit JTAG register. This value is configured in the Quartus II software using the **Device and Pin Options** dialog box. To configure this setting, perform the following steps:

1. On the Assignment menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click **Device**. The **Device** dialog box appears.
3. Click the **Device and Pin Options** button. The **Device and Pin Options** dialog box appears.
4. On the **General** tab, the **JTAG user code** appears about halfway down. Type the user code in 32-bit hexadecimal format (Figure 2–11).
5. Click OK.

Figure 2–11. Configuring the JTAG User Code



A separate script generates the GUI and is executed with the `quartus_stp` command line executable. During runtime, the GUI queries the device for the version information and formats it for display within the message box.

This design example uses the Tcl example scripts on the Altera website for generating an automatic version control number. The example section has additional examples for generating version information for an FPGA, and can be used for customizing this design example to suit your needs. Tcl design examples on the Altera website can be found on the following web page at www.altera.com:

- Quartus II Tcl Example: Automatic Version Number

The files for these design examples are located on the User Guide page in the Literature section of the Altera website (www.altera.com). The design files are targeted to a Cyclone III starter kit.

Conclusion

The Virtual JTAG megafunction gives you the ability to create your own software solution for monitoring, updating, and debugging your designs through the JTAG port without using any I/O pins on the device. It allows you to instrument your design for efficient, fast, and productive debugging solutions. These debugging solutions can be part of an evaluation test where you use other logic analyzers to

debug your design or as part of a production test where you do not have a host running an embedded logic analyzer. In addition to helping in debugging, the Virtual JTAG megafunction can be used to provide a single channel or multiple serial channels through the JTAG port of the device. These serial channels can be used in many applications to capture data or to force data to various parts of your logic.

Appendix A. SLD_NODE Discovery and Enumeration

This appendix describes the transactions necessary to enumerate all Virtual JTAG megafunction instances from your design at runtime. This process can be used by an custom JTAG controller to discover all available Virtual JTAG instances within a design.

All SLD nodes and the virtual JTAG registers that they contain are targeted by two Instruction Register values, USER0 and USER1. These values are shown in Table A–1.

Table A–1. USER1 and USER2 Instruction Values

Instruction	Binary Pattern
USER0	00 0000 1100
USER1	00 0000 1110

The USER1 instruction targets the virtual IR of either the sld_hub or a SLD node. That is, when the USER1 instruction is issued to the device, the subsequent DR scans target a specific virtual IR chain based on an address field contained within the DR scan. Table A–2 shows how the virtual IR, the DR target of the USER1 instruction, is interpreted.

Table A–2. USER1 DR

$m+n-1$ ... m	$m-1$ ... 0
ADDR [($n-1$)..0]	VIR_VALUE [($m-1$)..0]

The ADDR bits act as address values to signal the active SLD node that the virtual IR shift targets. The ADDR field is n bits in length, where n bits must be long enough to encode all SLD nodes within the design (Equation A–1).

Equation A–1.

$$n = \text{CEIL}(\log_2(\text{Number of SLD\_nodes} + 1))$$

The SLD hub is always 0 in the address map (Equation A–2).

Equation A–2.

$$\text{ADDR}[(n-1)..0] = 0$$

The **VIR_VALUE** in Table A–2 is the virtual IR value for the target SLD Node. The width of this field is m bits in length, where m is the length of the largest VIR for all of the SLD nodes in the design. All SLD nodes with VIR lengths of fewer than m bits must pad the **VIR_VALUE** field with zeros up to a length of m.

Discovery and enumeration of the SLD instances within a design requires interrogation of the sld_hub to determine the dimensions of the USER1 DR (m and n) and associating each SLD instance, specifically the Virtual JTAG megafunction instances, with an address value contained within the ADDR bits of the USER1 DR.

The discovery and enumeration process consists of the following steps:

1. Interrogate the SLD hub with the HUB_INFO instruction.
2. Shift out the 32-bit HUB IP Configuration Register to determine the number of SLD nodes in the design and the dimensions of the USER1 DR.
3. Associate the Virtual JTAG instance index to a ADDR value by shifting out the 32-bit SLD node info register for each SLD node in the design.

Issuing the HUB_INFO Instruction

The SLD hub contains the HUB IP Configuration Register and SLD_NODE_INFO register for each SLD node in the design. The HUB IP configuration register provides information needed to determine the dimensions of the USER1 DR chain. The SLD_NODE_INFO register is used to determine the address mapping for Virtual JTAG instance in your design. This register set is shifted out by issuing the HUB_INFO instruction. Both the ADDR bits for the SLD hub and the HUB_INFO instruction is 0 × 0.

Because *m* and *n* are unknown at this point, the DR register (ADDR bits + VIR_VALUE) must be filled with zeros. Shifting a sequence of 64 zeroes into the USER1 DR is sufficient to cover the most conservative case for *m* and *n*.

HUB IP Configuration Register

When the USER1/HUB_INFO instruction sequence is issued, the USER0 instruction must be applied to enable the target register of the HUB_INFO instruction. The HUB IP configuration register is shifted out using eight four-bit nibble scans of the DR register. Each four-bit scan must pass through the UPDATE_DR state before the next four-bit scan. The 8 scans are assembled into a 32-bit value with the definitions shown in Table A–3.

Table A–3. Hub IP Configuration Register

$Nibble_7$	$Nibble_6$	$Nibble_5$	$Nibble_4$	$Nibble_3$	$Nibble_2$	$Nibble_1$	$Nibble_0$
31	27	26	19	18	8	7	0
HUB IP version		*N*		ALTERA_MFG_ID (0 × 06E)		SUM (*m*, *n*)	

The dimensions of the USER1 DR chain can be determined from the SUM (*m*, *n*) and *N* (number of nodes in the design). Equation A–3 shows the values of *m* and *n*.

Equation A–3.

$$n = \text{CEIL}(\log_2(N+1))$$

$$m = \text{SUM}(m, n) - n$$

SLD_NODE Info Register

Because the number of SLD nodes is now known, the Nodes on the hub can be enumerated by repeating the 8 four-bit nibble scans, once for each Node, to yield the SLD_NODE_INFO register of each Node. The DR nibble shifts are a continuation of the HUB_INFO DR shift used to shift out the Hub IP Configuration register.

The order of the Nodes as they are shifted out determines the ADDR values for the Nodes, beginning with ADDR[*n* – 1..0] = 1, for the first Node SLD_NODE_INFO shifted out, up to and including ADDR[*n* – 1..0] = *N*, for the last node on the hub.

Table A–4. SLD_NODE_INFO register

31 27	26 19	18 8	7 0
Node Version	NODE ID	NODE MFG_ID	NODE INST ID

Table A–5 summarizes the function of each field.

Table A–5. Needs title

Field	Function
Node Version	Identifies the version of the SLD node
Node ID	Identifies the type of NODE IP (0x8 for the Virtual JTAG megafunction)
Node MFG_ID	SLD Node Manufacturer ID (0x6E for Virtual JTAG megafunction)
NODE_INST_ID	Used to distinguish multiple instances of the same IP. Corresponds to the instance index assigned in the MegaWizard Plug-In Manager.

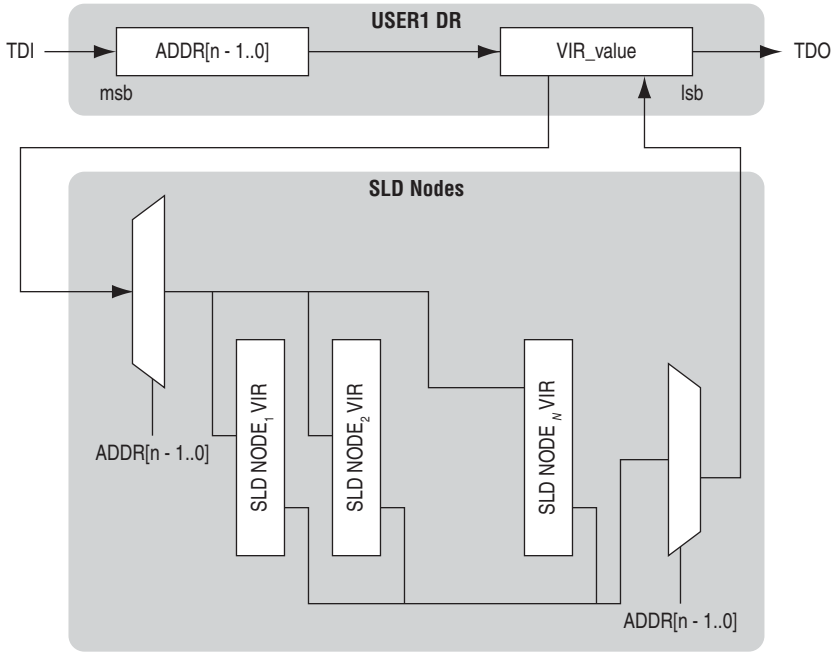
You can identify each Virtual JTAG instance within the design by decoding the NODE ID and NODE_INST_ID fields. The Virtual JTAG megafunction uses a NODE ID of 8. The NODE INST ID corresponds to the instance index that you configured within the Megawizard Plug-In Manager. The ADDR bits for each Virtual JTAG node is then determined by matching each Virtual JTAG instance to the sequence number in which the SLD_NODE_INFO register is shifted out.

Appendix B. Capturing the Virtual IR Instruction Register

In applications that contain multiple SLD nodes, capturing the value of the VIR may require issuing an instruction to the SLD hub to target a SLD node. This appendix describes the method to query for a VIR using the VIR_CAPTURE instruction.

This appendix describes the instruction to return the VIR value of a particular SLD node. Each SLD NODE VIR register acts as a parallel hold rank register to the USER1 DR chain. The sld_hub uses the ADDR[$n-1..0$] bits prepended to the VIR shift value to target the correct SLD NODE VIR register. After the SLD_state_machine asserts virtual_update_IR, the active SLD node latches VIR_VALUE of the USER1 DR register. Figure B–1 shows a functional model of the interaction of the USER1 DR register and the SLD node VIR.

Figure B–1. Functional Model Interaction between USER1 DR CHAIN and SLD Node VIRs



The ADDR bits target the selection muxes in Figure B–1 after the sld_hub FSM has exited the virtual_update_IR state. Upon the next USER1 DR transaction, the USER1 DR chain will latch the VIR of the last active SLD_NODE to shift out of TDO. Thus, if you need to capture the VIR of an SLD node that is different than the one addressed in the previous shift cycle, you must issue the VIR_CAPTURE instruction. The VIR_CAPTURE instruction to the sld_hub acts as an address cycle to force an update to the muxes in Figure B–1.

To form the VIR_CAPTURE instruction, use the instruction format shown in Equation B–1:

Equation B–1.

$$\text{VIR\_CAPTURE} = \text{ZERO}[(m-4)..0] \,\#\#\, \text{ADDR}[(n-1)..0] \,\#\#\, 011$$

Notes to Equation B–1:

In this equation, the following variables are:

(1) **ZERO[]** is an array of zeros

(2) **##** is the concatenation operator.

(3) *m* is the width of the VIR_VALUE field

(4) *n* is the width of the ADDR bit. Both *m* and *n* are defined in Appendix A, SLD_NODE Discovery and Enumeration.

Revision History

The following table shows the revision history for this user guide.

Document Revision History

Date and Version	Changes Made	Summary of Changes
December 2008 v.2.0	■ Expanded description of the system-level debugging (SLD) infrastructure ■ Added two new design examples ■ Updated instructions for using the Tcl API to query for the bit transactions ■ Included two new appendices that describe the enumerations and discovery process for use with a custom JTAG controller	Major re-write.
June 2006 v.1.0	Initial release	—

Referenced Documents

This user guide references the following documents:

- *AN 39: IEEE 1149.1 (JTAG) Boundary-Scan Testing in Altera Devices*
- *Quartus II Scripting Reference Manual*
- *Section V: In-System Design Debugging* in volume 3 of the *Quartus II Handbook*
- *Volume 2: Design Implementation and Optimization* of the *Quartus II Handbook*
- *Volume 3: Verification* of the *Quartus II Handbook*
- The following pages on www.altera.com:
  - ISP & the Jam STAPL
  - Embedded Programming With Jam STAPL

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact *(Note 1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com

Contact *(Note 1)*	Contact Method	Address
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, dialog box options, software utility names, and other GUI labels. For example, **\qdesigns** directory, **d:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicates document titles. For example, *AN 519: Stratix IV Design Guidelines*.
Italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, *<file name>* and *<project name>*.**pof** file.
Initial Capital Letters	Indicates keyboard keys and menu names. For example, Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. Active-low signals are denoted by suffix n. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
1., 2., 3., and a., b., c., and so on.	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The angled arrow instructs you to press Enter.
	The feet direct you to more information about a particular topic.