Altera Virtual JTAG Interface

D. W. Hawkins (dwh@ovro.caltech.edu)
Revision: 1.5

November 1, 2011

Contents

1 Introduction

Altera provides a suite of System-Level Debugging (SLD) Intellectual Property (IP) cores to aid in FPGA debugging. These cores provide access to the programmable logic fabric of the FPGA via the FPGA JTAG interface. The suite of tools is described in the Quartus II Handbook, Volume 3, Section 4, *System Debugging Tools* [3]. The suite of tool includes;

- System Console
- Transceiver Toolkit
- Signal Tap II Logic Analyzer
- Signal Probe
- Logic Analyzer Interface (LAI)
- In-System Sources and Probes
- In-System Memory Content Editor
- Virtual JTAG Interface

Chapter 13 of the handbook provides an overview of these tools, while Chapters 14 through 20 describe all but the Virtual JTAG component in detail. These debug tools consists of a hardware component, and Tcl procedures for interfacing to the hardware. The hardware consists of user-defined instances of debug cores, along with a JTAG hub component that coordinates access. The presence of the hub component is transparent to the user (it is abstracted by the Tcl procedures).

A typical application of these cores is the development of new components. For example, consider the development of a new Altera Avalon bus slave component. The component can be tested by creating an SOPC Builder system design containing an JTAG-to-Avalon Master (or JTAG-to-Avalon Streaming) interface and the component. Transactions to the component can then be generated using System Console Tcl procedures. A Signal Tap II logic analyzer instance could be added to the design to capture the Avalon bus transactions.

The Virtual JTAG Interface (VJI) component provides the lowest-level JTAG access of all the debug cores. Developing custom debug components using this interface provides the maximum flexibility to a user, eg., a user can define their own JTAG-to-Avalon interfaces and include features not available in the Altera provided core. The Virtual JTAG component is described in;

- The Virtual JTAG Megafunction User Guide [1]
- Quartus II help. From the Contents, see;
  - *Using Altera Megafunctions→Megafunctions/LPM*,
    JTAG-accessible Extensions MegaWizards and MegaFunctions
  - *Devices and Adapters→API Functions for Tcl*,
    `::quartus::jtag 1.0`

The Virtual JTAG User's Guide contains the majority of information required to use the Virtual JTAG interface. However, the Altera documentation has errors, and the simulation models are not useful for automated component testing.

This document provides an analysis of the Altera Virtual JTAG documentation, simulation, and synthesis results. It provides a clear description of how the interface works, and provides a simulation model that can be used for automated component testing. A general-purpose Virtual JTAG component is developed that allows custom device identification. Example designs are provided.

2 Virtual JTAG Component Analysis

This section analyzes Altera Virtual JTAG component instantiation, simulation, and synthesis. The section starts by describing Virtual JTAG component instantiation. The simulation results for that component show numerous problems with the Altera simulation model. Tcl procedures for performing JTAG accesses are then described, and used to generate hardware transaction sequences. The hardware transaction sequences are captured using an external logic analyzer and a SignalTap II logic analyzer. The traces are used to show the problems with the Altera simulation model, and are used in Section 3 to develop a functionally correct simulation model. The section ends with a description of the Altera hub and node instruction encoding, and the effect on that encoding when designs contain multiple SLD components.

2.1 Component Instantiation

The Virtual JTAG core can be instantiated using the Quartus II MegaWizard (*Tools→MegaWizard Plug-In Manager*) or by direct instantiation (see the Quartus II help for the template). An Altera DE2 board Virtual JTAG test configuration was created as follows;

- Using the procedure described in Appendix B, a new project for the DE2 board was created. The project folder was named `sld_vjtag_to_gpio`, with top-level component `de2.vhd`.

  The `sld_vjtag_to_gpio` top-level design, `de2.vhd`, contains an `sld_virtual_jtag` instance connected to the DE2 board LEDs, hexadecimal displays, switches, and the GPIO (for external logic analyzer transaction capture). This section describes how the Virtual JTAG instance was instantiated; see the source code for details on the connections to the DE2 board I/O.

- A Virtual JTAG instance was created as follows;

  1. A new Quartus project was created in a temporary work folder (the project targeted the Cyclone II on the DE2 board).
  2. The MegaWizard was used to create a Virtual JTAG instance named `vji`, with the settings shown in Figure 1, i.e.,
     - Virtual instruction register width of 3.
     - Enable the primitive JTAG state signal ports.
  3. No other MegaWizard defaults were changed.
  4. Click 'Finish' to have the `vji.vhd` instance file created.

- The `sld_virtual_jtag` instance from `vji.vhd` was then copied-and-pasted into `de2.vhd` (do not delete the Quartus work folder, as the MegaWizard will be used to create simulation stimulus in the next section).

- The library definition for the `sld_virtual_jtag` component was added to `de2.vhd`, i.e.,

```
library altera_mf;
use altera_mf.altera_mf_components.all;
```

- Internal signals were attached to all of the `sld_virtual_jtag` ports, and those signals were connected to the `gpio_a` and `gpio_b` ports (the DE2 board 2×20 100-mil headers), so that the signals can be probed with an external logic analyzer. (See the `de2.vhd` source for the JTAG signal to GPIO assignments).

- The Virtual instruction register bus from the Virtual JTAG component was connected to GPIO bits and to the LEDs (so that Tcl commands can change the LED state).

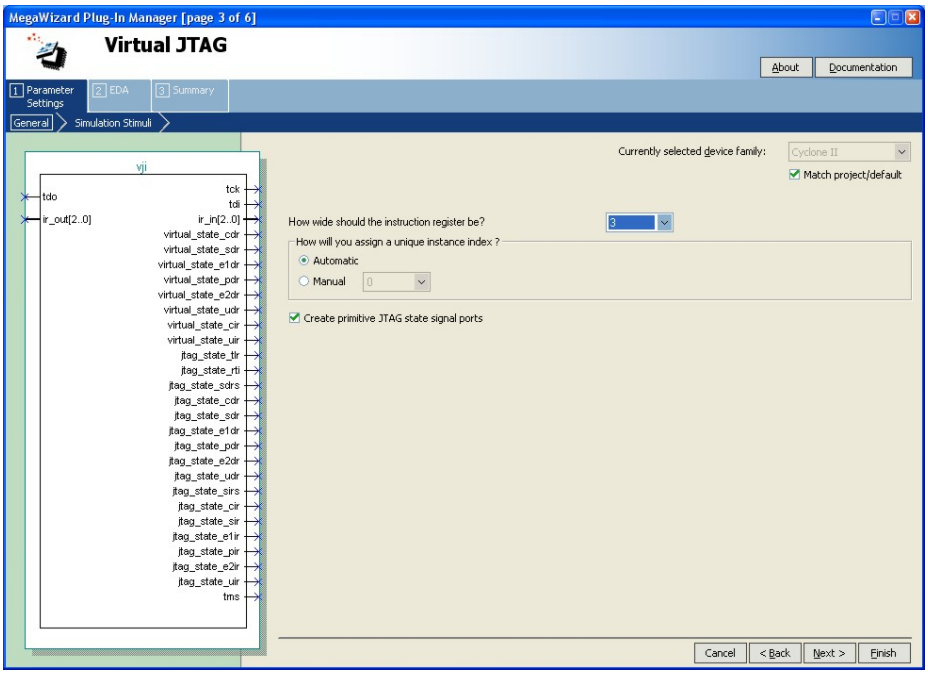


Figure 1: Altera Virtual JTAG component configuration.

- The Virtual instruction register bus to the Virtual JTAG component was connected to the DE2 switches (so that Tcl commands can read the switch state).
- The JTAG TDO signal was configured to toggle on the rising-edge of the JTAG TCK signal.
- The design was synthesized.

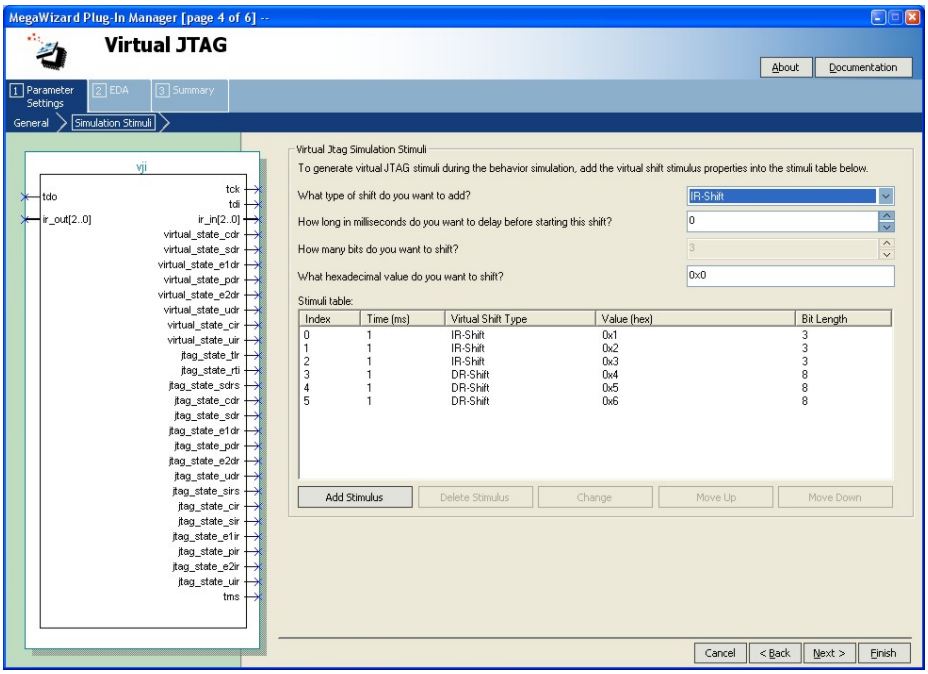


Figure 2: Altera Virtual JTAG simulation stimulus configuration.

2.2 Altera Simulation Model Analysis

The Virtual JTAG component has simulation support, however, the waveforms the simulation model generates are incorrect, making the simulation feature useless. Even if simulation worked, the transaction support is so basic, that it would be barely useful; the model supports the generation of statically-defined, write-only, Virtual IR-shift and Virtual DR-shift operations using generics[1]. The fact that the stimulus is write-only makes writing a self-verifying testbench impossible, as there is no read-back method. This section describes how to create stimulus and contains figures showing the simulation IR-shift and DR-shift operations. The simulation waveforms are annotated to show the model problems, relative to the hardware measurements performed in the next section.

Figure 2 shows the MegaWizard simulation stimulus page; use the MegaWizard to edit the previously generated instance of the Virtual JTAG component and enter the values as shown in the figure, then regenerate the VHDL files (by clicking 'Finish'). The generics in the original instance (copied to the `de2.vhd` file) were;

```
sld_sim_action          => "",
sld_sim_n_scan          => 0,
sld_sim_total_length    => 0,
```

The addition of the stimulus changes the `vji.vhd` generics to;

```
sld_sim_action => "((1,1,1,3),(1,1,2,3),(1,1,3,3),(1,2,4,8),(1,2,5,8),(1,2,6,8))",
sld_sim_n_scan => 6,
sld_sim_total_length => 33,
```

The stimulus format is; delay between transactions (in units of 1ms), operation (1 = IR-shift, 2 = DR-shift), IR/DR value, and IR/DR bit-width.

[1]The stimulus could conceivably be changed during run-time using multiple runs of Modelsim's `vsim` with different generics each run, however, this is not a standard way to construct a self-verifying testbench.

The Virtual JTAG instance created in `vji.vhd` contains the stimulus via the generics, so the design can be simulated directly without wrapping the component in a testbench (the input ports `tdo` and `ir_in[2:0]` ports will be undriven so the Modelsim simulation will show these signals as unknown/undriven). Figures 3 through 7 show time segments from the simulation. The figures show; the JTAG waveforms, the JTAG TAP one-hot state machine encoded signals, the Virtual JTAG one-hot state machine encoded signals, and the Virtual Instruction Register parallel input/output ports. Comments on the start-up and Virtual IR-shift sequences are;

- Figure 3 shows the simulation waveforms after power-on. The waveforms show a JTAG TAP reset sequence (`tms` high for six `tck` rising-edges), followed by a JTAG TAP instruction phase, containing 10-bits of zeros. There is no Altera device 10-bit JTAG instruction code of `00_0000_0000b = 000h`, so this sequence is not useful.

  The stimulus leaves the JTAG TAP state-machine in the Update-IR state (the one-hot JTAG state `jtag_state_uir` remains asserted at the end of the simulation segment).

- Figure 4 shows the simulation waveforms for the first Virtual IR-shift stimulus, where the stimulus value for the IR shift was 1. The stimulus should consist of three JTAG TAP shift sequences;

  – JTAG TAP IR-shift of 10-bits `USER1 = 00_0000_1110b = 00Eh`
  – JTAG TAP DR-shift of 5-bits `VIR_CAPTURE = 0_1011b = 0Bh`
  – JTAG TAP DR-shift of 5-bits `VIR_USER = 1_0001b = 11h`

  where the last two sequences are 5-bit Virtual IR-shift codes (Section 2.6 discusses the bit encoding of the Virtual IR-shift codes). The three JTAG TAP shift sequences are present in the simulation, however, the TDI activity is incorrect, and the last DR-shift phase (the Virtual IR-shift user instruction phase, `VIR_USER`) is missing a shift phase.

  The stimulus leaves the JTAG TAP state-machine in the Update-DR state and the Virtual JTAG state-machine in the Update-IR state (the one-hot JTAG state `jtag_state_udr` and Virtual JTAG state `vjtag_state_uir` remain asserted at the end of the simulation segment).

  The Virtual JTAG instruction register updates to `1` when the Virtual JTAG state machine enters the Update-IR state.

- Figure 5 shows the simulation waveforms for the second Virtual IR-shift stimulus, where the stimulus value for the IR shift was 2. Since the JTAG TAP instruction `USER1` was shifted in during the first Virtual IR-shift stimulus, this instruction phase is not required for the second stimulus, so the second stimulus waveforms consists of the last two shift sequences seen in the waveforms of the first stimulus, i.e., `VIR_CAPTURE` followed by `VIR_USER`. As with the first Virtual IR-shift sequence, the `VIR_USER` sequence is missing a shift phase.

  The Virtual JTAG instruction register updates to `2` when the Virtual JTAG state machine enters the Update-IR state.

- The third Virtual IR-shift stimulus[2] results in waveforms similar to the second Virtual IR-shift stimulus, with the Virtual JTAG instruction register updating to `3` at the end of the stimulus sequence (prior to the Virtual DR-shift stimulus).

[2]Not shown in this document. Run the Modelsim simulation to see it.

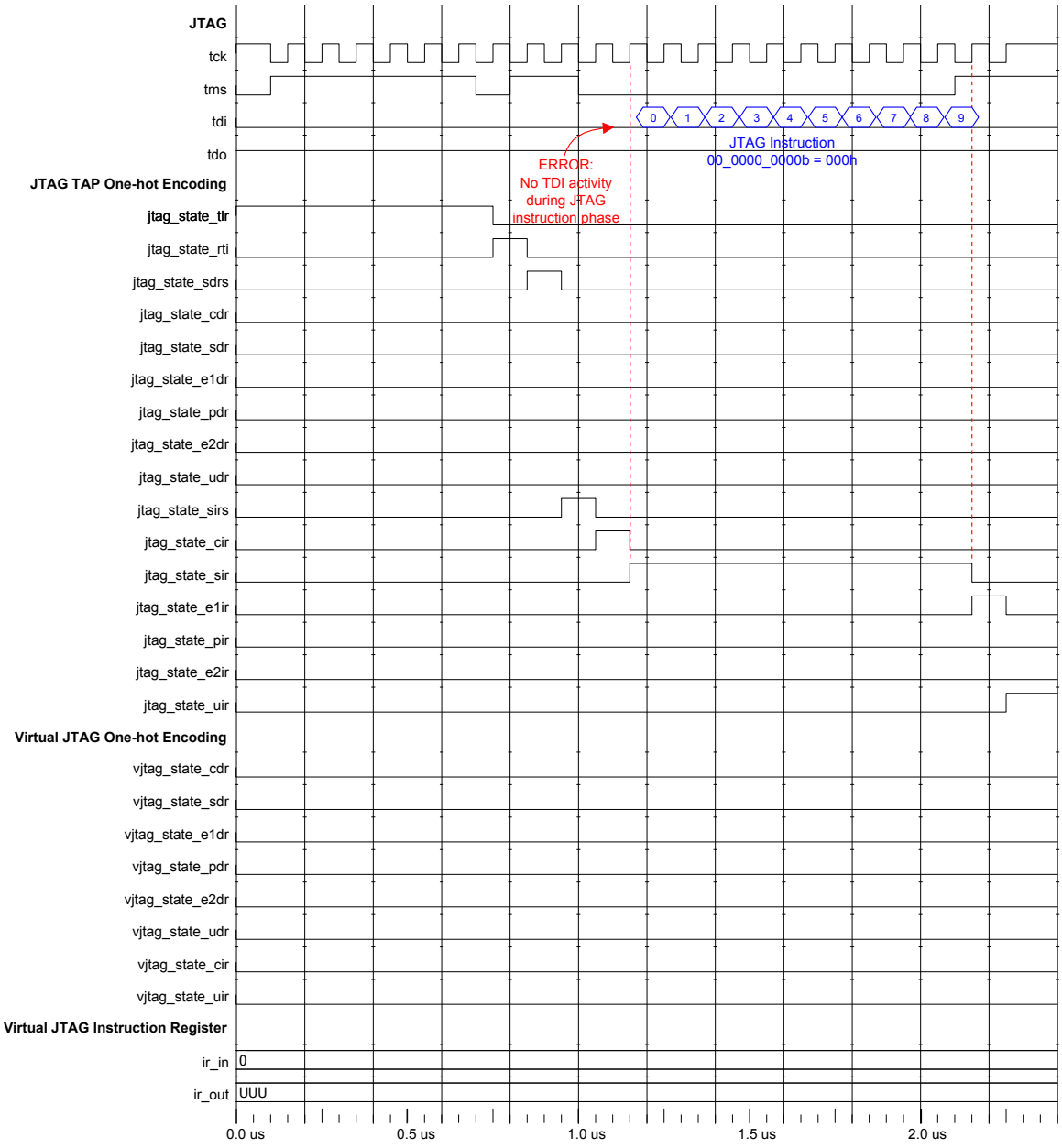


Figure 3: Altera Virtual JTAG simulation; power-on sequence.

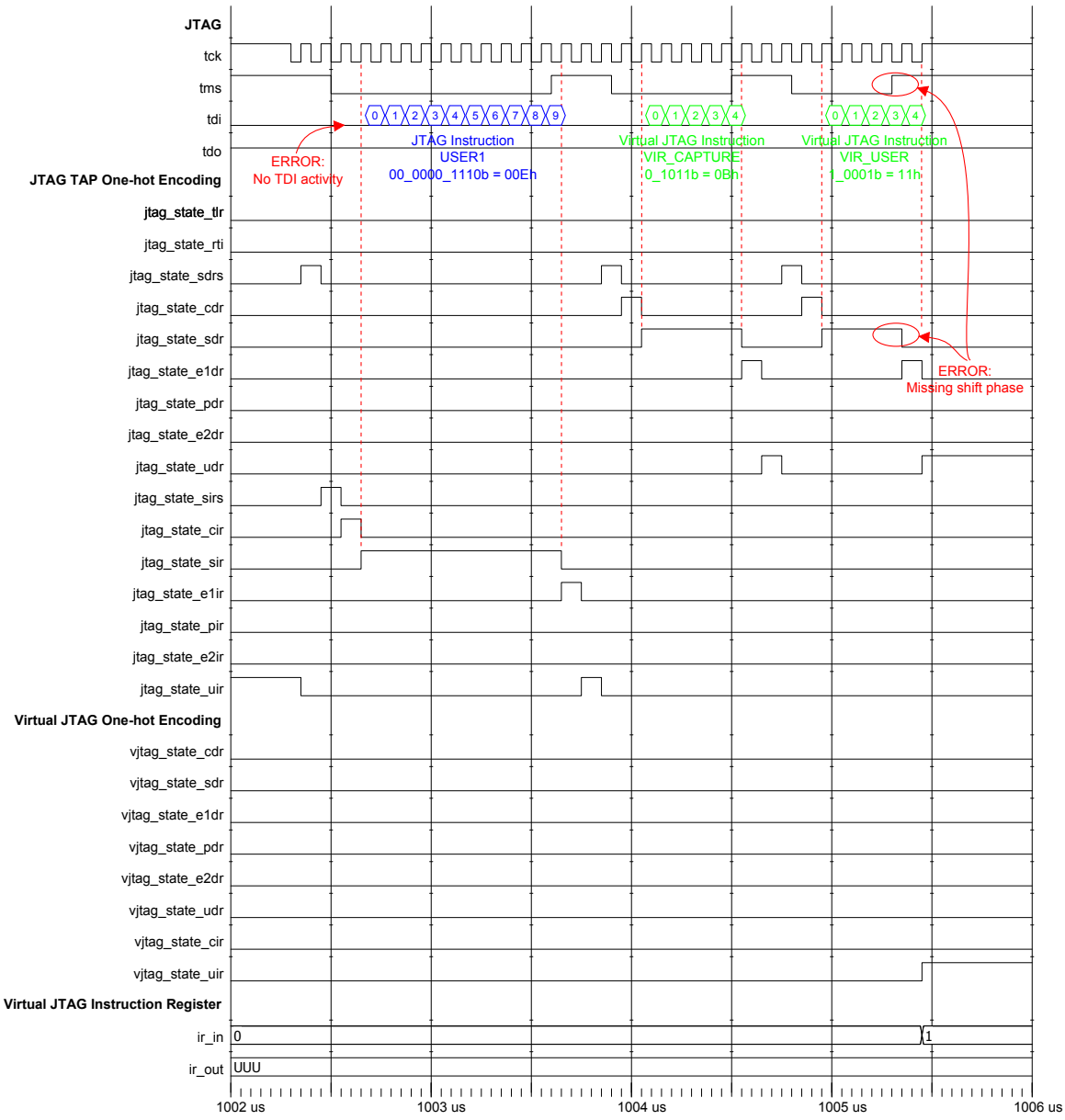


Figure 4: Altera Virtual JTAG simulation; first IR-shift sequence (stimulus value of 1).

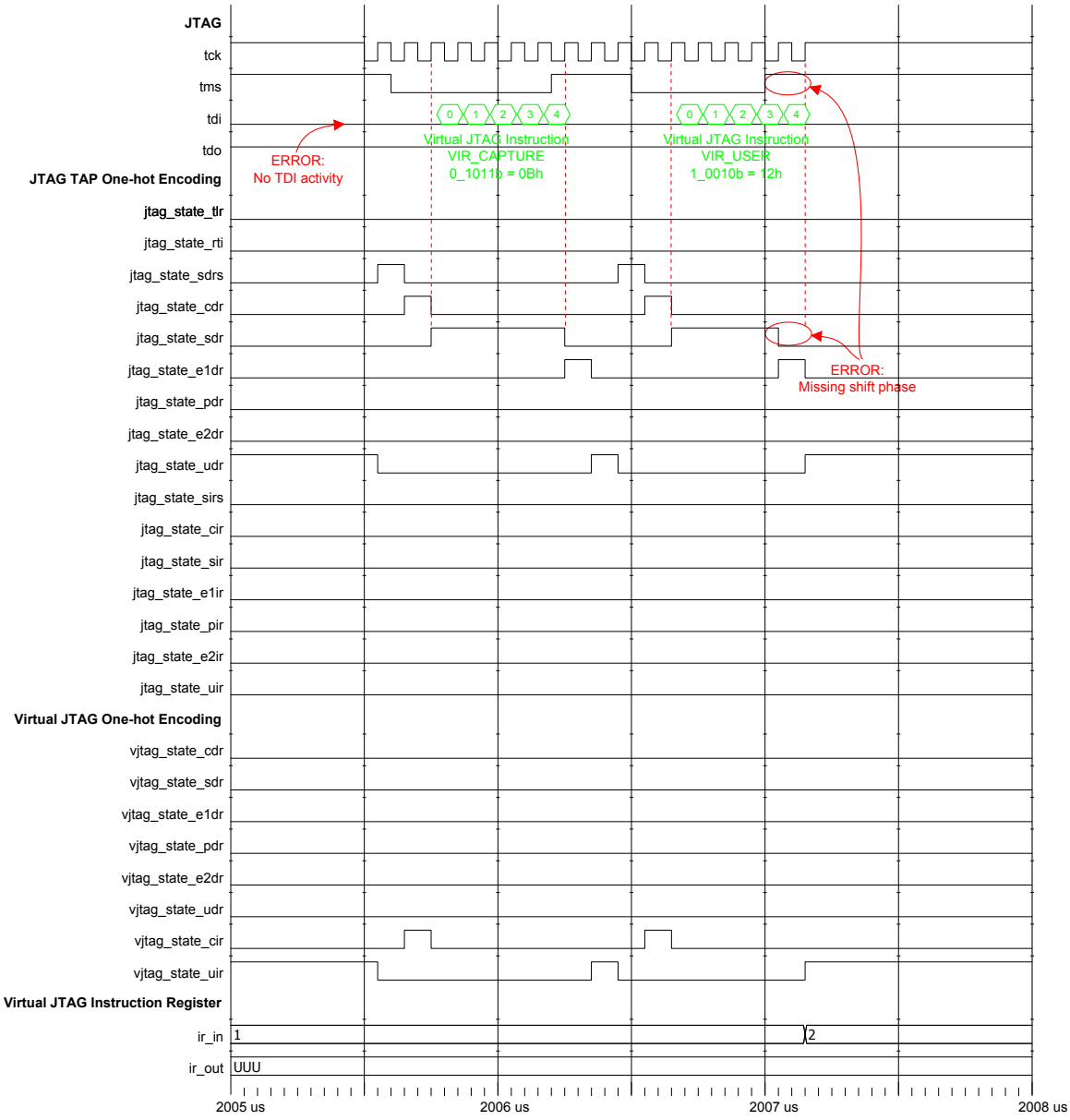


Figure 5: Altera Virtual JTAG simulation; second IR-shift sequence (stimulus value of 2).

Comments on the Virtual DR-shift stimulus are;

- Figure 6 shows the simulation waveforms for the first Virtual DR-shift stimulus, where the stimulus value for the 8-bit Virtual DR-shift was 4. The stimulus should consists of two JTAG TAP shift sequences;
  - JTAG TAP IR-shift of 10-bits `USER0 = 00_0000_1100b = 00Ch`
  - JTAG TAP DR-shift of 8-bits `VDR_USER = 0000_0100b = 04h`

  The two JTAG TAP shift sequences are present in the simulation, however, there are a couple of problems; there is an extra shift clock during the instruction phase, and the TDI activity is incorrect. The TDI signal has no activity during the instruction phase, and during the DR-shift phase, `tdi` asserts during the wrong bit position (based on the other DR-shift stimulus, it appears that the DR-shift value is being shifted out one clock late).

  The stimulus leaves the JTAG TAP and Virtual JTAG state-machines in the Update-DR state (the one-hot states `jtag_state_udr` and `vjtag_state_udr` remain asserted at the end of the simulation segment).

- Figure 7 shows the simulation waveforms for the second Virtual DR-shift stimulus, where the stimulus value for the 8-bit Virtual DR-shift was 5. Since the JTAG TAP instruction `USER0` was shifted in during the first Virtual DR-shift stimulus, this instruction phase is not required for the second stimulus, so the second stimulus waveforms consists of the last shift sequence seen in the waveforms of the first stimulus, i.e., `VDR_USER`. As with the first Virtual DR-shift stimulus, the Virtual DR-shift value, `VDR_USER = 05h`, is shifted out one clock late.

- The third Virtual DR-shift stimulus[3] results in waveforms similar to the second Virtual DR-shift stimulus, with the Virtual DR-shift value, `VDR_USER = 06h`, shifted out one clock late

- The simulation ends with the JTAG TAP being reset (`tms` high for six `tck` rising-edges).

[3] Not shown in this document. Run the Modelsim simulation to see it.

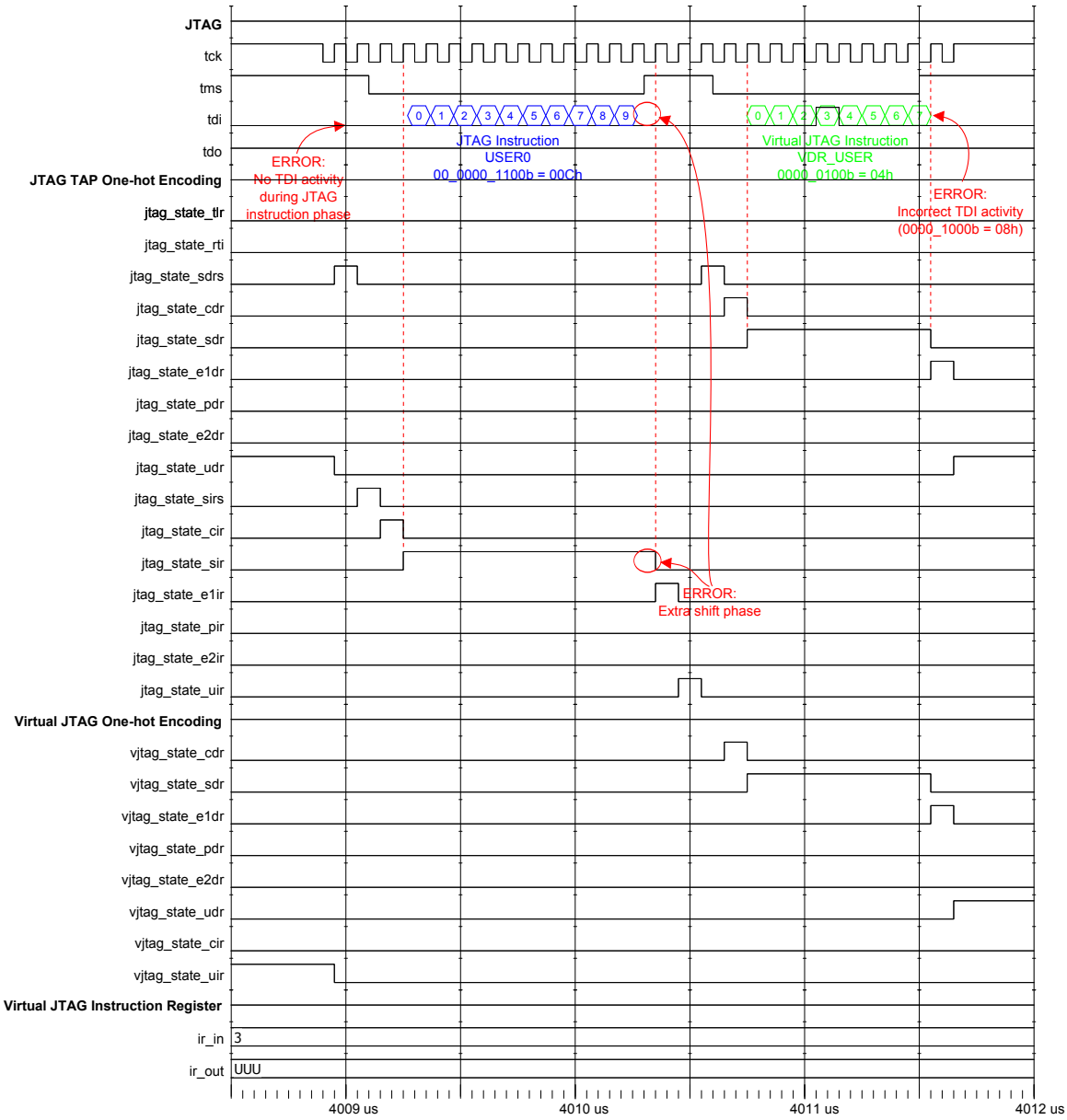


Figure 6: Altera Virtual JTAG simulation; first DR-shift sequence (stimulus value of 4).

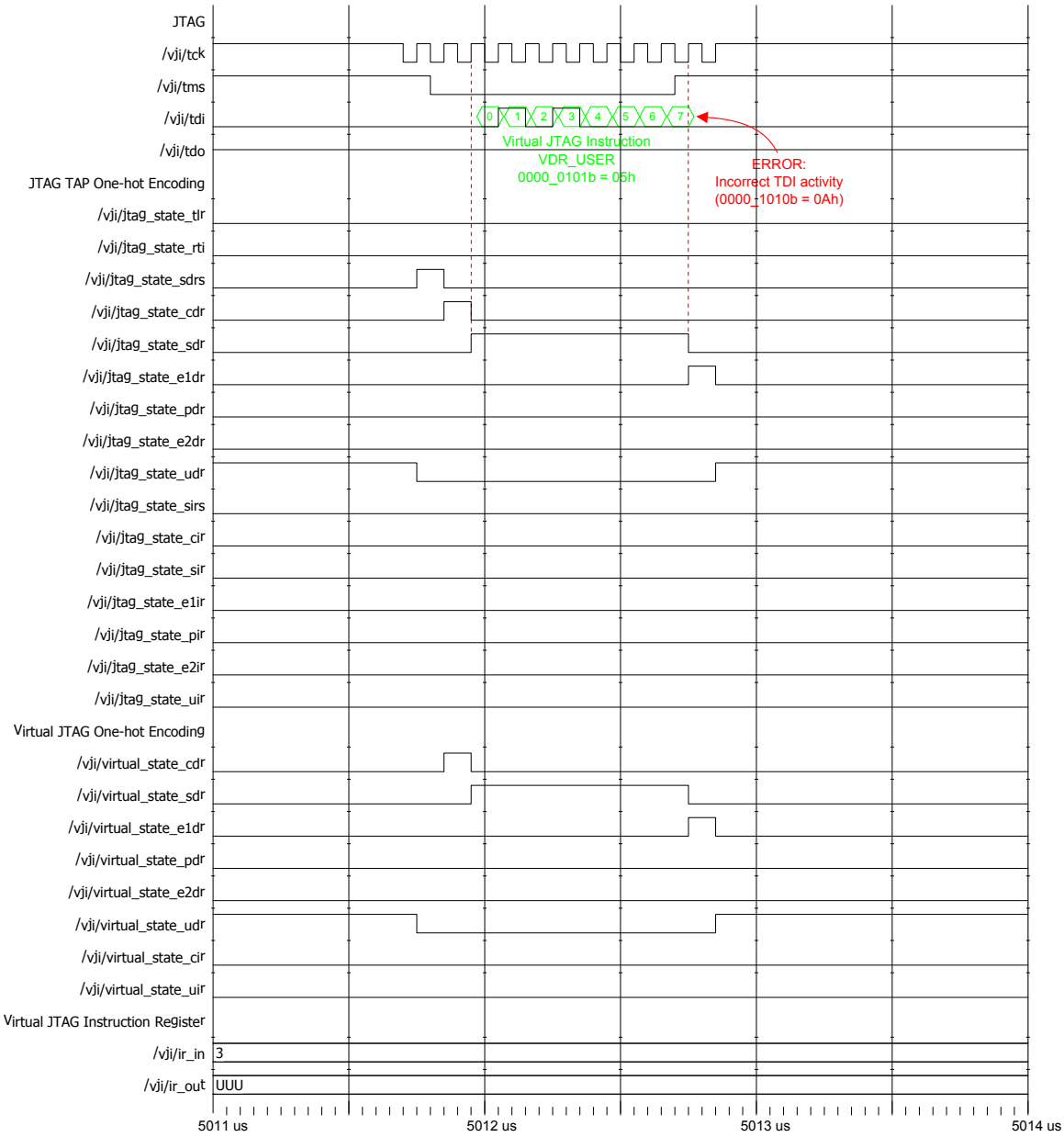


Figure 7: Altera Virtual JTAG simulation; second DR-shift sequence (stimulus value of 5).

2.3 Altera Tcl procedures for JTAG access

Virtual JTAG communications are performed using procedures from the `::quartus::jtag 1.0` Tcl package. Table 1 summarizes the Tcl procedures available in the package. Detailed information on the Tcl procedures are provided in the Quartus II help (*Contents→Devices and Adapters→API Functions for Tcl*), in the `quartus_stp` Tcl console (use `-help` or `-long_help` as an option to the Tcl procedure), and in the Quartus II scripting manual [2]. Figure 8 shows the open/close sequence for a single USB controller connected to a single device JTAG chain, eg., the Altera DE2 board with embedded USB-Blaster and Cyclone II device. The Quartus help recommends holding the JTAG lock for the minimum time possible to allow other applications to access the device.

The JTAG IR-shift and DR-shift procedures require some caution with their use, the following subtleties need to be observed;

- The `device_ir_shift` and `device_virtual_ir_shift` procedure `-ir_value` argument is a Tcl numeric value, eg., the JTAG TAP `USER1` instruction can be issued as decimal 14, hexadecimal 0xE, or octal 016 (Tcl commands `expr 0xE` and `expr 0x16` both return 14).
- The `device_dr_shift` and `device_virtual_dr_shift` procedure `-dr_value` argument is a binary string by default. When the `-value_in_hex` option is used, the `-dr_value` argument is a hexadecimal string *without* a leading `0x` prefix. The length of the binary or hexadecimal string must be sufficient to match the number of bits indicated by the `-length` argument.
- If a JTAG or Virtual JTAG shift command is issued with the option `-no_captured_ir_value` or `-no_captured_dr_value`, *no activity* will occur on the JTAG interface until a command without this option or a `device_unlock` command is issued.

An additional subtlety comes from Tcl; Figure 9 demonstrates that Tcl supports 32-bit signed integers. Arguments to the Tcl procedures for JTAG and Virtual JTAG access reflect this limitation; `-ir_value` is an integer, whereas `-dr_value` is either a binary or hexadecimal string (allowing for larger than 32-bit values). Since the argument to a Virtual IR-shift procedure is limited to 32-bits, the `sld_virtual_jtag` component IR width, `SLD_IR_WIDTH`, is also limited to 32-bits. At the hardware level, a 32-bit Virtual IR-shift instruction has address bits prepended to the instruction, and is issued as a JTAG DR-shift instruction (see Section 2.6 for details on the instruction encoding).

The sequence of JTAG TAP transactions that are issued for a Virtual JTAG transaction depends on the previously issued transactions; this can be seen in the simulations in Section 2.2, and in the logic analyzer traces in Sections 2.4 and 2.5. The sequence of JTAG TAP transactions that are issued for a Virtual JTAG transaction can be reported by the Virtual JTAG Tcl procedures by adding the option `-show_equivalent_device_ir_dr_shift`. Figures 10 and 11 show the Virtual IR-shift and DR-shift sequences that result in different JTAG transactions. These commands are used in the next sections to generate JTAG transactions that are captured via logic analyzer.

Table 1: Quartus JTAG (`::quartus::jtag 1.0`) Tcl procedures.

Command	Description
`get_hardware_names`	List the JTAG hardware, eg. USB-Blaster.
`get_device_names`	List the devices on a JTAG bus.
`open_device`	Access a device on a JTAG bus.
`close_device`	End access to a device.
`device_lock`	Lock the JTAG interface while accessing a device.
`device_unlock`	Release the JTAG interface lock.
`device_run_test_idle`	Move the JTAG TAP to run-test/idle.
`device_dr_shift`	Issue a JTAG TAP DR-shift sequence.
`device_ir_shift`	Issue a JTAG TAP IR-shift sequence.
`device_virtual_dr_shift`	Issue a Virtual JTAG DR-shift sequence.
`device_virtual_ir_shift`	Issue a Virtual JTAG IR-shift sequence.

```
proc jtag_open {} {
    # Get the list of JTAG controllers
    set hardware_names [get_hardware_names]

    # Select the first JTAG controller
    set hardware_name [lindex $hardware_names 0]

    # Get the list of FPGAs in the JTAG chain
    set device_names [get_device_names\
        -hardware_name $hardware_name]

    # Select the first FPGA
    set device_name [lindex $device_names 0]

    puts "\nJTAG: $hardware_name, FPGA: $device_name"
    open_device -hardware_name $hardware_name\
        -device_name $device_name
}

proc jtag_close {} {
    close_device
}
```

Figure 8: Tcl procedures for open and close.

```
proc print_ints {width} {
    for {set i 0} {$i < $width} {incr i} {
        set val [expr {1<<$i}];
        puts "[format "%2d: %9X %12d" $i $val $val]"
    }
}
```

(a)

```
tcl> print_ints 35
 0:         1            1
 1:         2            2
 2:         4            4
 3:         8            8
 4:        10           16
 5:        20           32
 6:        40           64
 7:        80          128
 8:       100          256
 9:       200          512
10:       400         1024
11:       800         2048
12:      1000         4096
13:      2000         8192
14:      4000        16384
15:      8000        32768
16:     10000        65536
17:     20000       131072
18:     40000       262144
19:     80000       524288
20:    100000      1048576
21:    200000      2097152
22:    400000      4194304
23:    800000      8388608
24:   1000000     16777216
25:   2000000     33554432
26:   4000000     67108864
27:   8000000    134217728
28:  10000000    268435456
29:  20000000    536870912
30:  40000000   1073741824
31:  80000000  -2147483648
32:         0            0
33:         0            0
34:         0            0
tcl>
```

(b)

Figure 9: Tcl 32-bit integer support; (a) Tcl procedure `print_ints` to print integer values with widths of 1 to `width` bits, and (b) conversion values for widths of 1 to 35-bits. The output values indicate that Tcl supports signed 32-bit integers.

```
tcl> jtag_open
JTAG: USB-Blaster [USB-0], FPGA: @1:  EP2C35 (0x020B40DD)
tcl> device_lock -timeout 10000
tcl> device_virtual_ir_shift -instance_index 0 -ir_value 1
-show_equivalent_device_ir_dr_shift
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
5
tcl> device_virtual_ir_shift -instance_index 0 -ir_value 2
-show_equivalent_device_ir_dr_shift
Info:  Equivalent device ir and dr shift commands
    Info:  device_dr_shift -length 5 -dr_value 0B -value_in_hex
    Info:  device_dr_shift -length 5 -dr_value 12 -value_in_hex
5
tcl> device_virtual_ir_shift -instance_index 0 -ir_value 3
-show_equivalent_device_ir_dr_shift -no_captured_ir_value
Info:  Equivalent device ir and dr shift commands
    Info:  device_dr_shift -length 5 -dr_value 12 -value_in_hex
tcl> device_unlock
tcl> jtag_close
tcl>
```

Figure 10: Virtual IR-shift Tcl procedures (using `quartus_stp`).

```
tcl> jtag_open
JTAG: USB-Blaster [USB-0], FPGA: @1:  EP2C35 (0x020B40DD)
tcl> device_lock -timeout 10000
tcl> device_virtual_dr_shift -instance_index 0 -length 8 -dr_value 04
-value_in_hex -show_equivalent_device_ir_dr_shift
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
AA
tcl> device_virtual_dr_shift -instance_index 0 -length 8 -dr_value 05
-value_in_hex -show_equivalent_device_ir_dr_shift
Info:  Equivalent device ir and dr shift commands
    Info:  device_dr_shift -length 8 -dr_value 05 -value_in_hex
AA
tcl> device_virtual_dr_shift -instance_index 0 -length 8 -dr_value 05
-value_in_hex -show_equivalent_device_ir_dr_shift -no_captured_dr_value
Info:  Equivalent device ir and dr shift commands
    Info:  device_dr_shift -length 8 -dr_value 06 -value_in_hex
tcl> device_unlock
tcl> jtag_close
tcl>
```

Figure 11: Virtual DR-shift Tcl procedures (using `quartus_stp`).

2.4 External logic analyzer traces

Virtual JTAG communications are performed using Tcl procedures from the `::quartus::jtag 1.0` Tcl package issued from the `quartus_stp` application. Section 2.3 discusses the Tcl procedures. Figures 10 and 11 show the Virtual IR-shift and DR-shift sequences that result in different JTAG transactions. Figures 12 through 16 logic analyzer traces for the IR-shift and DR-shift sequences captured from the Altera DE2 board[4]. The logic analyzer sequences are;

- Figure 12 shows an initial Virtual IR-shift sequence (after accessing the device), with a user instruction value of 1, i.e., the first `device_virtual_ir_shift` command in Figure 10.

  The captured sequence is slightly different than the sequence expected. There is an erroneously formed JTAG `BYPASS` instruction issued prior to the JTAG `USER1` instruction. Alternatively, this sequence could be a JTAG instruction register length detection sequence, eg., a single zero followed by ones is shifted into the JTAG IR register and the control software detects when the zero appears at the TDO output (the `BYPASS` command just happens to be the all-ones instruction). The other minor issue with the captured sequence is that the `TDI` signal shows a glitch during the `USER1` instruction sequence.

- Figure 13 shows a Virtual IR-shift sequence, with an instruction value of 2, following a Virtual IR-shift sequence i.e., the second `device_virtual_ir_shift` command in Figure 10.

- Figure 14 shows a Virtual IR-shift sequence, with an instruction value of 3 and the option `-no_captured_ir_value` following a Virtual IR-shift sequence, i.e., the third `device_virtual_ir_shift` command in Figure 10.

- Figure 15 shows the first Virtual DR-shift sequence, with a DR value of 0xAA, i.e., the first `device_virtual_dr_shift` command in Figure 11 (with the data value changed to 0xAA so that the TDI signal would toggle each data bit).

  During the JTAG `USER0` instruction sequence, the `TDI` signal shows a glitch (similar to the glitch seen during the Virtual IR-shift `USER1` instruction sequence).

  The traces show a logical error in the hardware; there are two Virtual JTAG one-hot state assertions, i.e., both capture-DR (`VS_CDR`) and capture-IR (`VS_CIR`) assert during the same clock period.

- Figure 16 second Virtual DR-shift sequence, with a DR value of 0xAA, i.e., the second `device_virtual_dr_shift` command in Figure 11 (with the data value changed to 0xAA so that the TDI signal would toggle each data bit).

The logic analyzer captures show that the JTAG sequences issued by each Tcl procedure are consistent with the commands reported by the option `-show_equivalent_device_ir_dr_shift`. The logic analyzer traces also provide the state of the JTAG TAP and Virtual JTAG state-machines after each command is issued (the states observed are consistent with the simulation states). The logic analyzer waveforms confirm the simulation model errors identified in Section 2.2.

2.5 SignalTap II logic analyzer traces

Figures 17 and 18 show Virtual IR-shift and Virtual DR-shift sequences captured using a SignalTap II logic analyzer instance in the DE2 design. The traces are very similar to those captured using the external logic anayzer. The slight difference, is that the presence of the SignalTap II instances changes the Virtual Instruction encoding as discussed in Section 2.6. The captured traces show the TDI glitch, and erroneous one-hot state assertion observed with the external logic analyzer.

[4]The `TDO` signal was configured to toggle at each `TCK` rising-edge.

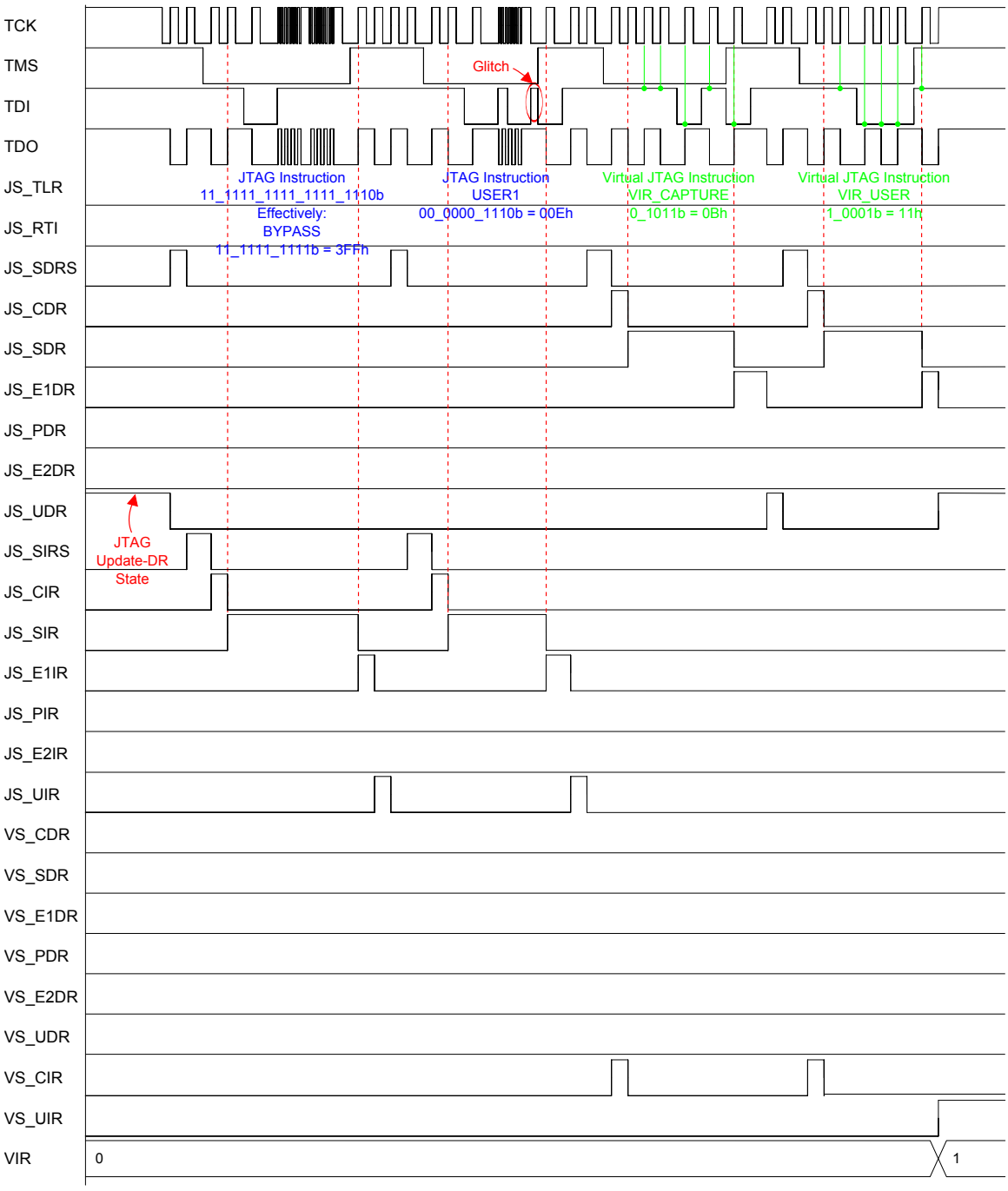


Figure 12: Altera Virtual JTAG logic analyzer trace; initial Virtual IR-shift sequence (after accessing the device). The sequence consists of JTAG `BYPASS` and `USER1` IR-shift sequences followed by the JTAG DR-shift sequences.

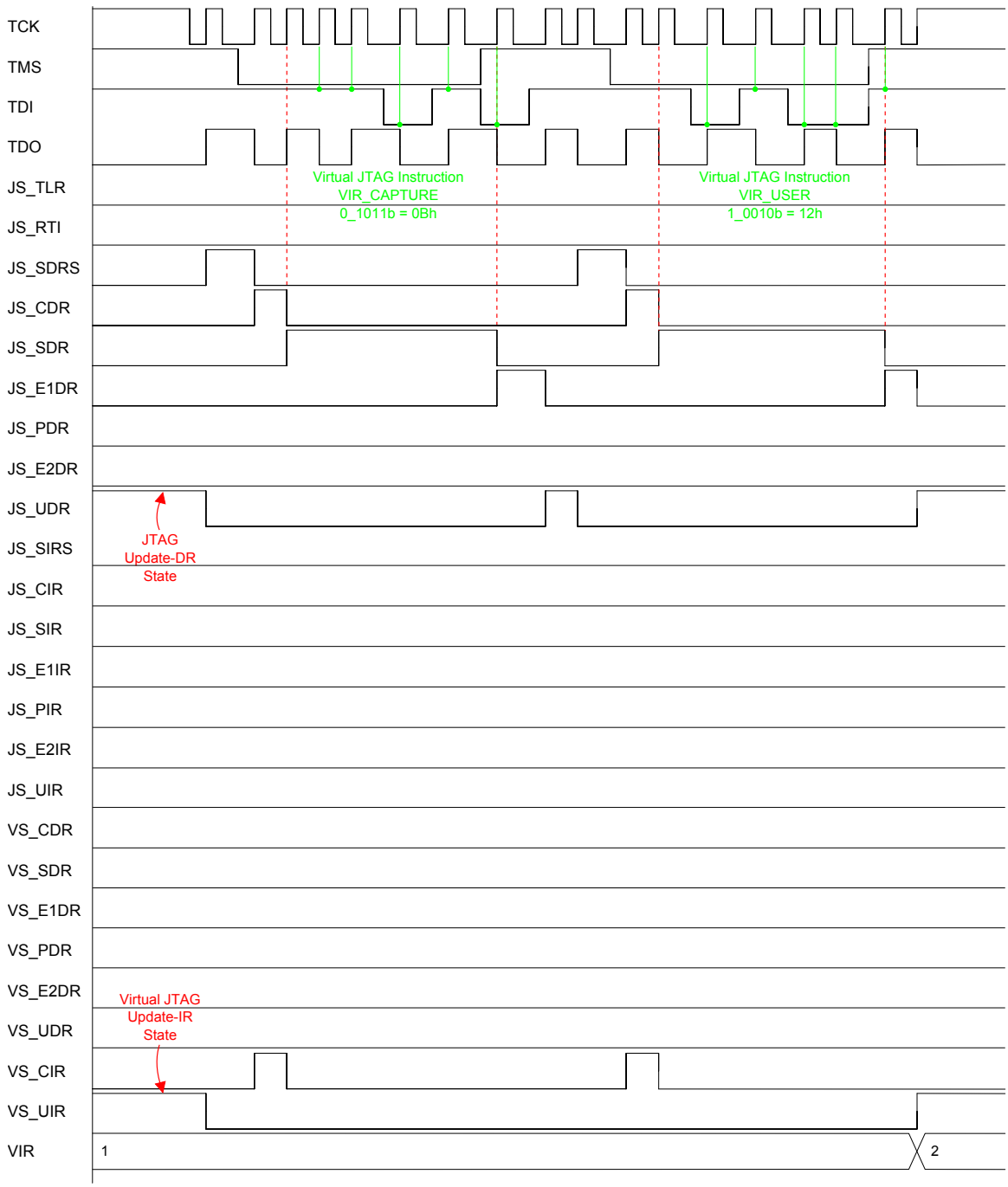


Figure 13: Altera Virtual JTAG logic analyzer trace; Virtual IR-shift sequence following a Virtual IR-shift sequence (the JTAG `USER1` sequence is not issued).

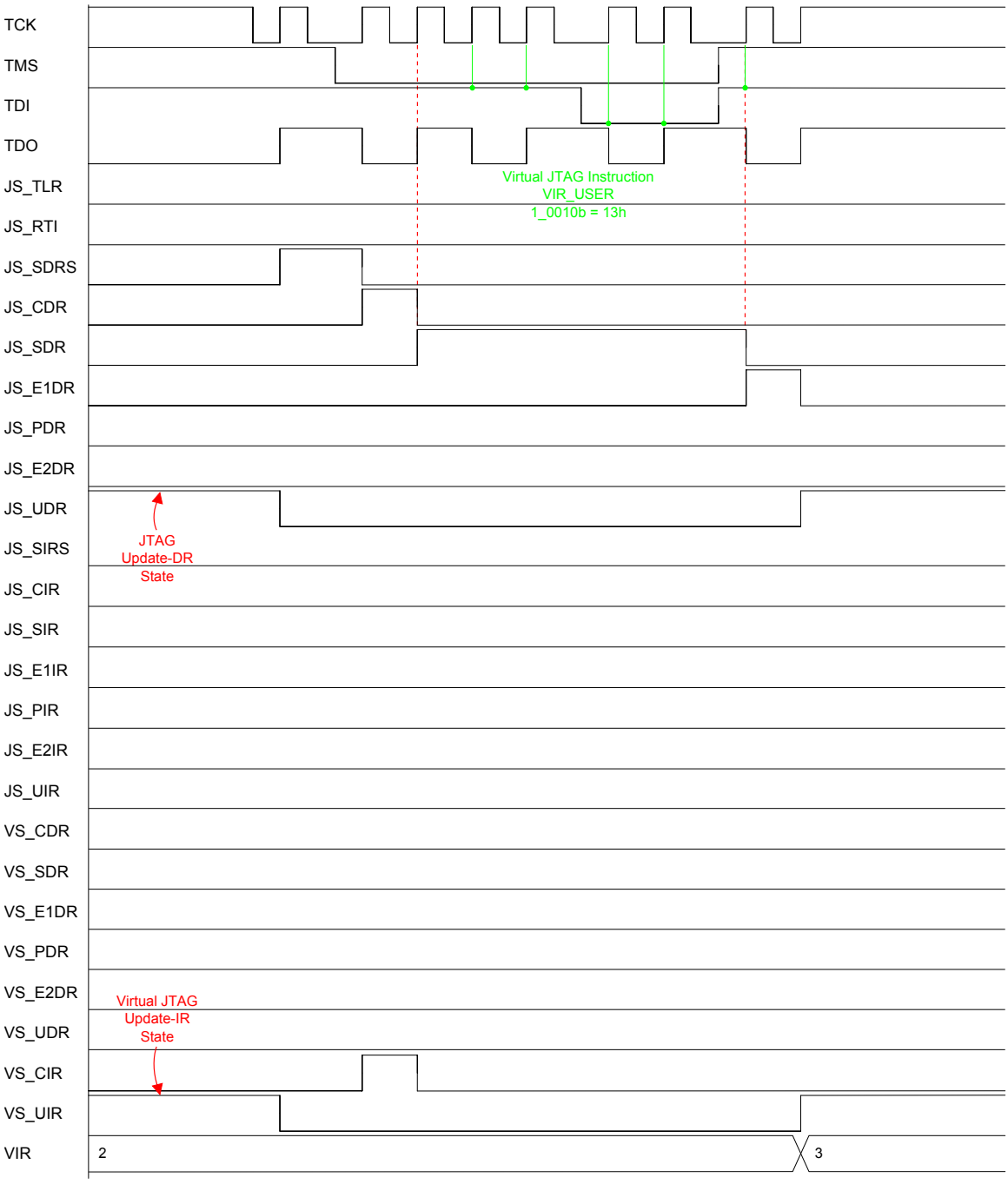


Figure 14: Altera Virtual JTAG logic analyzer trace; Virtual IR-shift sequence with the option `-no_captured_ir_value` following a Virtual IR-shift sequence (the JTAG `USER1` and `VIR_CAPTURE` sequences are not issued).

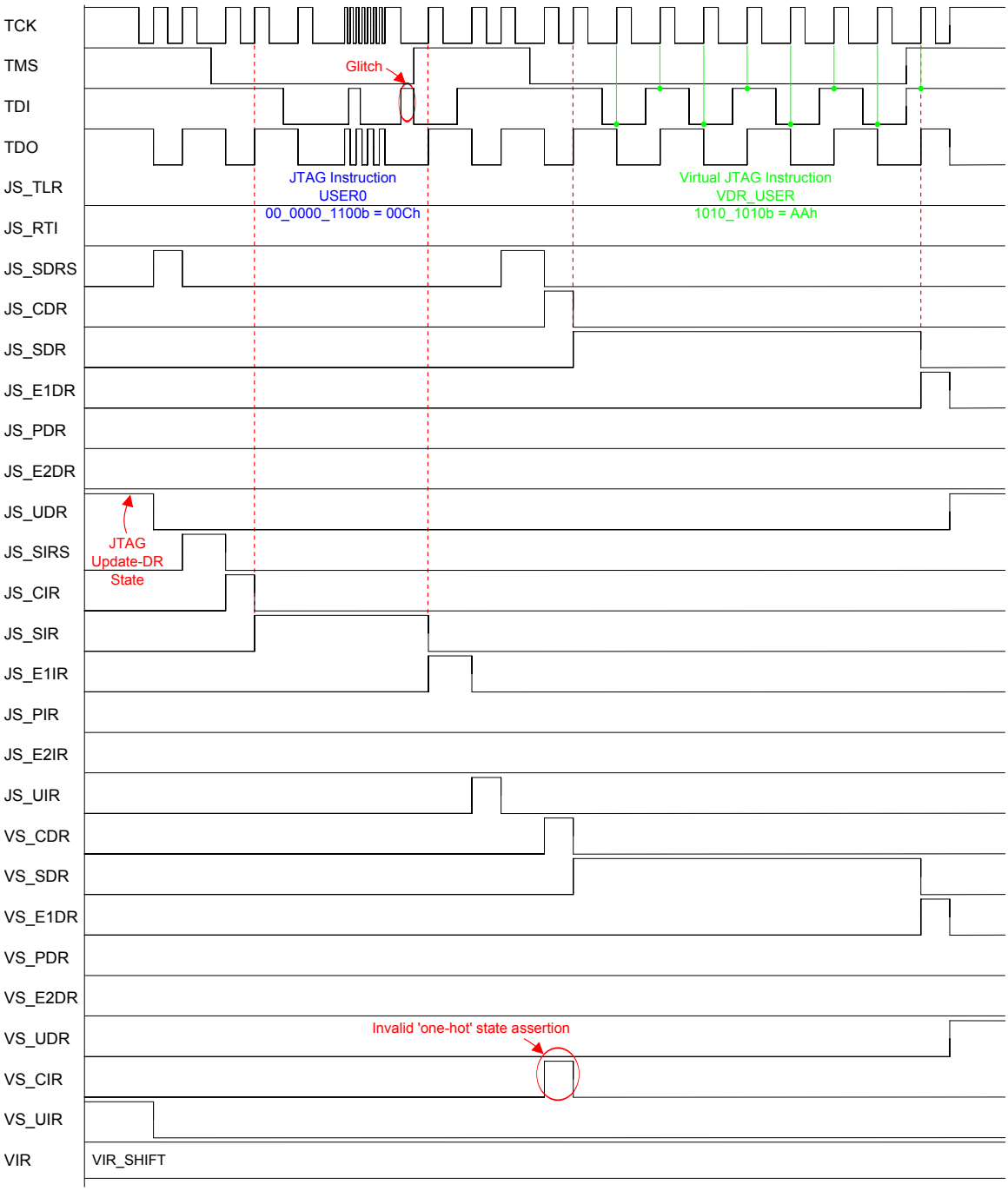


Figure 15: Altera Virtual JTAG logic analyzer trace; Virtual DR-shift sequence following a IR-shift sequence. The sequence consists of JTAG `USER0` IR-shift sequence followed by a JTAG DR-shift sequence. Note the invalid pulse on `virtual_state_cir`; this indicates a logical error in the Altera component.

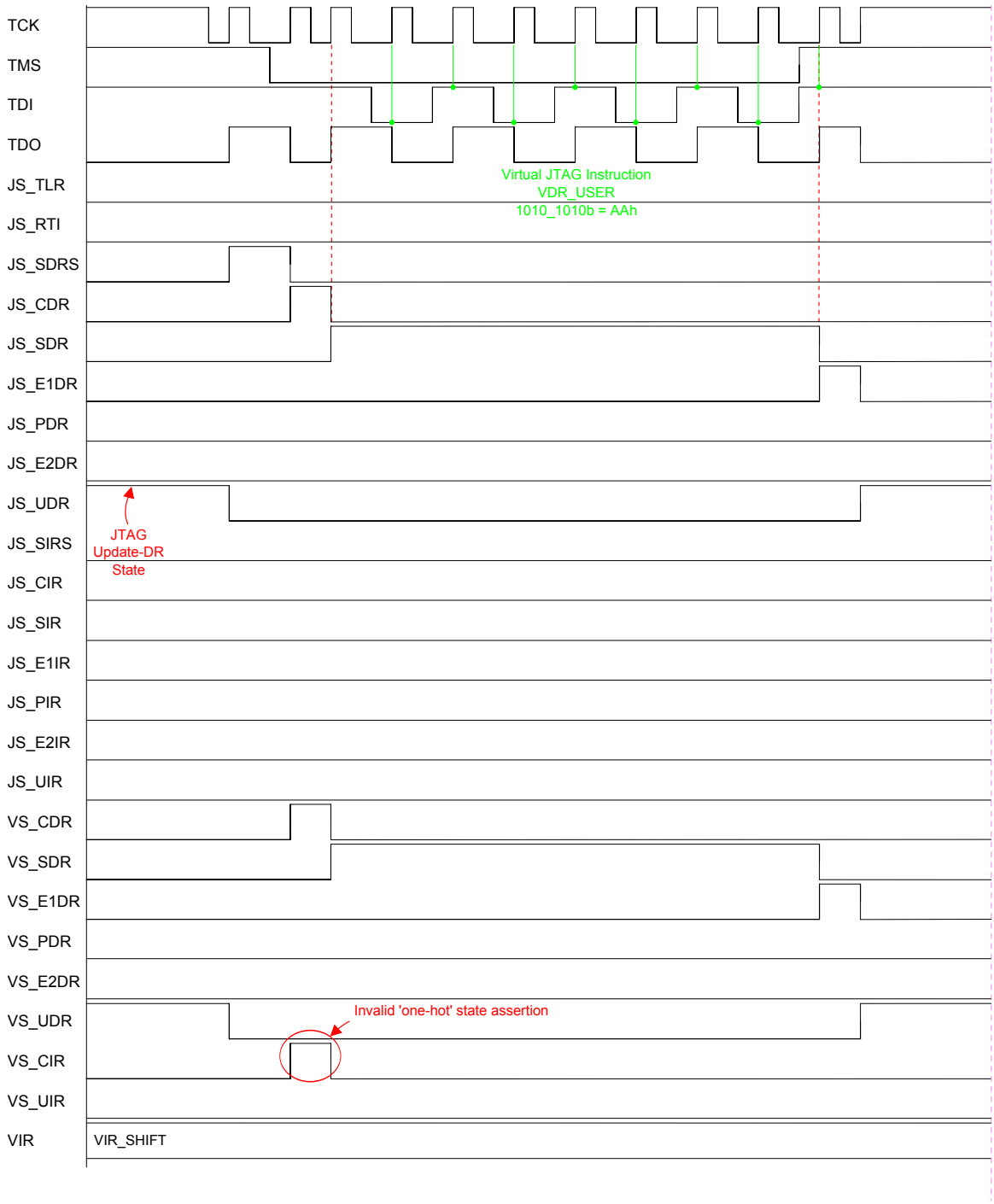


Figure 16: Altera Virtual JTAG logic analyzer trace; Virtual DR-shift sequence following a Virtual DR-shift sequence (the JTAG `USER0` sequence is not issued). Note the invalid pulse on `virtual_state_cir`; this indicates a logical error in the Altera component.

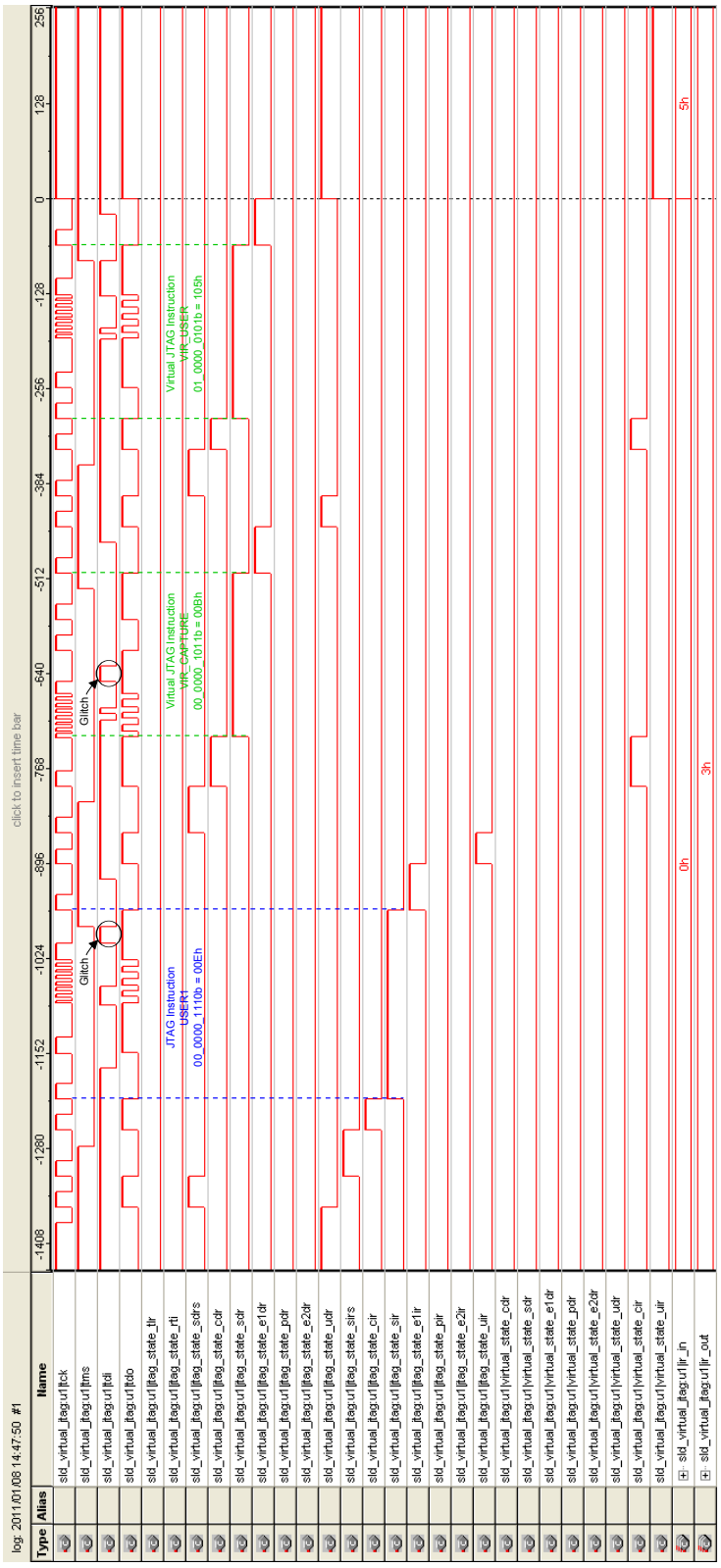


Figure 17: Altera Virtual JTAG SignalTap II logic analyzer trace; initial Virtual IR-shift sequence. The sequence consists of the JTAG USER1 IR-shift sequence followed by the JTAG DR-shift sequences.

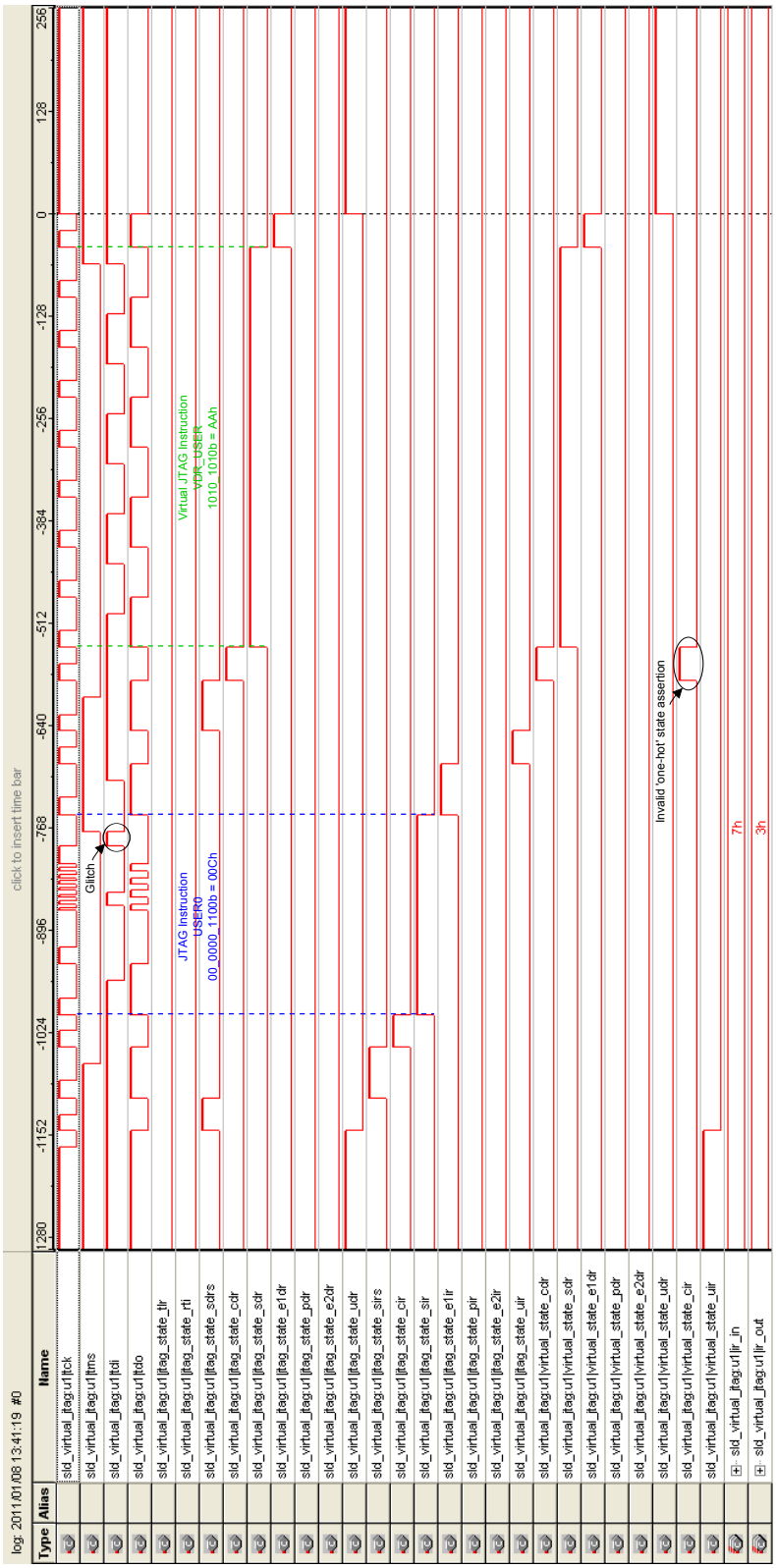


Figure 18: Altera Virtual JTAG SignalTap II logic analyzer trace; Virtual DR-shift sequence following a IR-shift sequence. The sequence consists of the JTAG `USER0` IR-shift sequence followed by a JTAG DR-shift sequence. Note the invalid pulse on `virtual_state_cir`; this indicates a logical error in the Altera component.

2.6 JTAG hub/node identification and instruction encoding

The Appendices in the Virtual JTAG Megafunction User Guide [1] provide low-level details on the JTAG hub and device node identification and instruction encoding. This section provides details on how to use these instructions, and provides a correction relative to the user guide.

The Altera JTAG debug devices are coordinated via the JTAG hub (reported in the Quartus hierarchy window as the `sld_hub`). The properties of the hub and the nodes attached to the hub can be queried using the Virtual Instruction Register (VIR) `HUB_INFO` instruction. Figure 19 shows the VIR instruction encoding, while Figures 20 and 21 show the hub and node information that is accessed using the `HUB_INFO` instruction.

The VIR is used for both hub instructions and user instructions. The VIR width (also known as the USER1 DR length) depends on the width of the hub instructions and the width of the user instructions. The width of the hub instructions depends on the number of nodes in the design. The width of the user instructions depends on the number of nodes in the design, and the *maximum* width of the user-defined field of a user instruction, i.e., the maximum value of the `SLD_IR_WIDTH` parameter used on the `sld_virtual_jtag` instances in a design, and the maximum width of the instruction field required by any other SLD JTAG components in the design, eg., SignalTap II logic analyzer instances and SOPC Builder components such as the JTAG-to-Avalon Bridge and JTAG UART.

The width of the VIR n-bit hub/node address field shown in Figure 19 is

$$n = \text{ceil}\left\{\log_2(N+1)\right\} \tag{1}$$

where N is the number of nodes in the design. The plus one in this equation accounts for the fact that the hub uses an address (address zero).

The width of the VIR m-bit instruction value field shown in Figure 19 is

$$m = \max(n+3, \texttt{MAX\_SLD\_IR\_WIDTH}) \tag{2}$$

where $n+3$ is the field width required to encode the VIR capture instruction, and `MAX_SLD_IR_WIDTH` is the field width required to encode the maximum width of instructions used by the SLD components in the design (instructions with less than m-bits are padded with zeros before being issued).

The width m can be read from the hardware using the VIR `HUB_INFO` instruction. However, encoding the `HUB_INFO` instruction to read the width m requires knowledge of m! This conflict is resolved by the fact that the encoding for the hub address and the `HUB_INFO` instruction is all zeros, so the VIR register can be loaded with zeros up to the maximum possible VIR width. The maximum total VIR width $(m+n)$ can be determined as follows;

- The maximum number of nodes is 255, i.e., $n = 8$.
- The maximum user-defined `SLD_IR_WIDTH` is 32-bits, i.e., $m = 32$.

hence the maximum VIR width (for a design containing only `sld_virtual_jtag` components) is 40-bits. Altera recommends shifting 64-bits of zeros (pA-2 [1]).

Once the `HUB_INFO` instruction has been loaded into the VIR, the 32-bit *Hub IP Configuration Register* can be shifted out by performing eight 4-bit (nibble) DR-shift operations. Figure 20 shows the fields within this register. The Virtual JTAG User's Guide incorrectly defines the first field as the total VIR register width `SUM(m,n)`, whereas this field actually returns the VIR register m-width. The VIR field width n can be determined from the `HUB_INFO` *number of nodes* field, and the width m is read directly. From that point on, VIR instructions (including `HUB_INFO`) can be encoded and issued with the correct number of bits. The `HUB_INFO` *number of nodes* field indicates how many 32-bit `SLD_NODE_INFO` register values can be read following the 32-bit `HUB_INFO` register. These 32-bit registers are read using the same 4-bit (nibble) DR-shift sequence as the `HUB_INFO` register. Figure 21 shows the register format and shows some example node register values.

HUB_INFO Instruction

m+n-1 m	m-1 3	2 0
n-bit Hub Address (all zeros)	Zero Padding	000b

CAPTURE Instruction

m+n-1 m	m-1	n+2 3	2 0
n-bit Hub Address (all zeros)	Zero Padding	n-bit Node Address	011b

User VIR Instruction

m+n-1 m	m-1 0
n-bit Node Address	m-bit User VIR value

Figure 19: JTAG Virtual Instruction Register (VIR) Encoding.

$Nibble_7$	$Nibble_6$	$Nibble_5$	$Nibble_4$	$Nibble_3$	$Nibble_2$	$Nibble_1$	$Nibble_0$
31 27	26 19		18 8			7 0	
Hub IP Version	N (number of nodes)		Altera Manufacturer ID (110 = 6Eh)			VIR m-width	

Figure 20: JTAG Hub IP Configuration Register Format.

$Nibble_7$	$Nibble_6$	$Nibble_5$	$Nibble_4$	$Nibble_3$	$Nibble_2$	$Nibble_1$	$Nibble_0$
31 27	26 19		18 8			7 0	
Node Version	Node ID		Node Manufacturer ID (110 = 6Eh)			Node Instance ID	

(a)

Virtual JTAG

0	8	110 = 6Eh	0

Signal Tap II Logic Analyzer

6	0	110 = 6Eh	0

JTAG-to-Avalon Bridge

0	132 = 84h	110 = 6Eh	0

(b)

Figure 21: JTAG Node Information; (a) Register Format and (b) Example node information (for single node instances).

The equations for the Virtual Instruction Register (VIR) width parameters n and m, given in (1) and (2), were confirmed using a DE2 board design that instantiates multiple `sld_virtual_jtag` components. The design, `sld_vjtag_multiple`, is parameterized with two generics;

- `INSTANCES`

  The number of `sld_virtual_jtag` instances.

- `SLD_IR_WIDTH`

  The width of the user-defined component of the VIR register
  (with all instances having the same width).

Table 2 shows the results of multiple synthesis runs; the table contains the value calculated for n, and the two possible values for the total VIR register width, $(n + m)$. The table can be used to predict the VIR instruction width to expect for a design containing `sld_virtual_jtag` components. The predicted width is determined by first selecting the row consistent with the number of nodes in the design, and then the column consistent with the (maximum) `SLD_IR_WIDTH` used within the design. If the text entry at the intersection of the row and column values is black, then the VIR register width is dominated by the capture instruction, so the width is given by the blue capture instruction width. If however, the entry text is red, then the VIR register width is dominated by the user instruction width and is given by the red text.

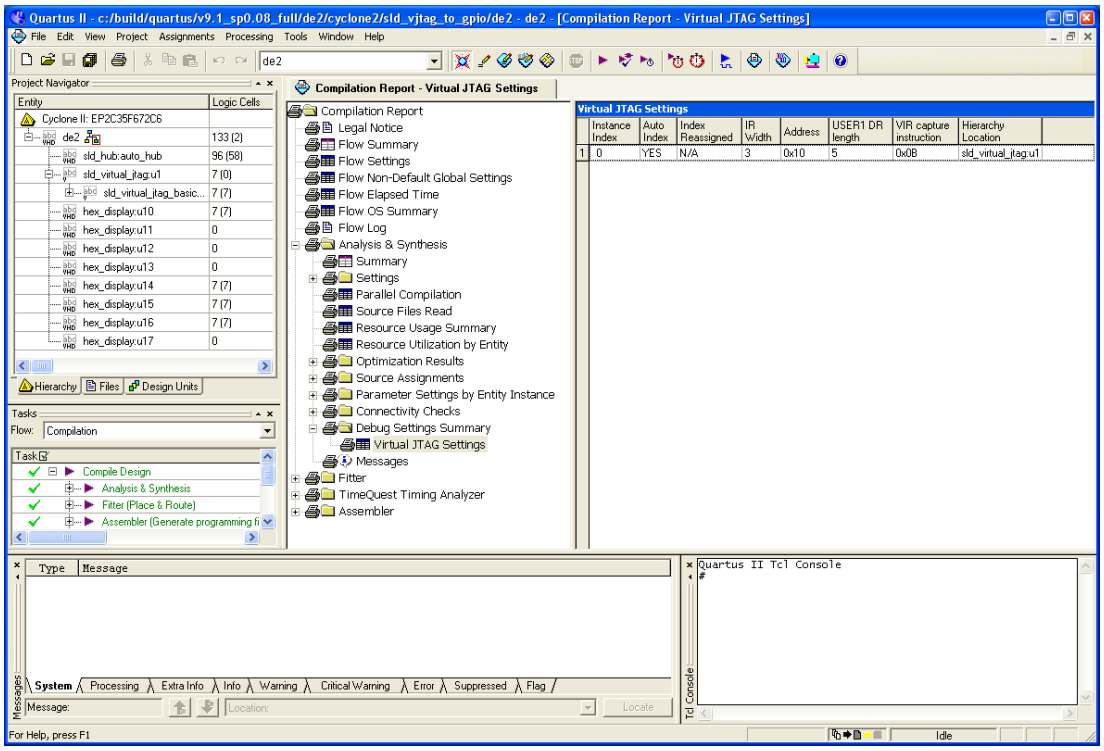
The VIR width parameters for a design are reported in multiple locations in Quartus II;

- The parameters for the `sld_virtual_jtag` component instances are reported in the Virtual JTAG settings compilation report; *Analysis & Synthesis*, *Debug Settings Summary*, *Virtual JTAG Settings*.
  - The USER1 DR length in this report is the total VIR width $(m + n)$.
  - The capture instruction encodings for each instance are reported.
  - The address value reported for each instance are essentially a user instruction with the user instruction value (the `SLD_IR_WIDTH` bits) set to zero.
- The SLD hub parameters can be obtained by hovering the mouse over the `sld_hub` component in the Quartus II hierarchy window after place-and-route of a design.
  - `n_nodes` is the number of nodes in the design (N)
  - `n_sel_bits` is the number of select (address) bits (n)
  - `n_node_ir_bits` is the numbers of bits in the instruction field (m)
  - `node_info` is a binary value equivalent to the concatenation of the 32-bit `SLD_NODE_INFO` values for each node in the design.
- SLD component parameters are reported in the `.jdi` file for a project, eg. `de2.jdi`.
  - The file has the 32-bit `SLD_NODE_INFO` values in hexadecimal.
  - The file contains node details for all of the SLD components in a design. For example, the DE2 board design `sld_vjtag_custom` can be configured to contain an `sld_virtual_jtag` component, a custom `sld_virtual_jtag_basic` component, a JTAG-to-Avalon bridge, and a SignalTap II logic analyzer instance. The Quartus compilation reports contain scattered details on each of these components. The `de2.jdi` file contains the component instance names, and JTAG identification details.

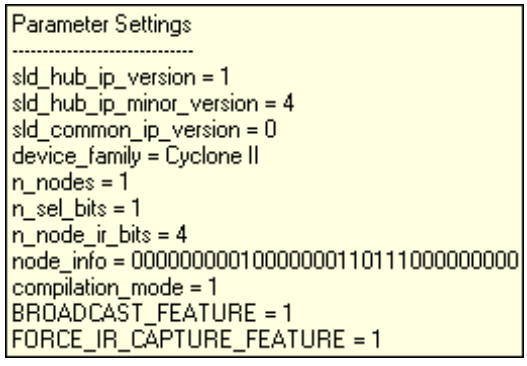
Figure 22 shows the Virtual JTAG settings report and the SLD hub tooltip for the `sld_vjtag_to_gpio` project, while Figure 23 shows the report for the `sld_vjtag_multiple` project.

Table 2: Virtual Instruction Register (VIR) width parameter exploration.

Number of nodes, N	VIR hub/node address width, n	VIR register width required for;											
		Capture $(2n+3)$	User instruction $(n+m)$ for $m =$ `SLD_IR_WIDTH` values of;										
			1	2	3	4	5	6	7	8	9	10	11	12
1	1	5	2	3	4	5	6	7	8	9	10	11	12	13
2-3	2	7	3	4	5	6	7	8	9	10	11	12	13	14
4-7	3	9	4	5	6	7	8	9	10	11	12	13	14	15
8-15	4	11	5	6	7	8	9	10	11	12	13	14	15	16
16-31	5	13	6	7	8	9	10	11	12	13	14	15	16	17
32-63	6	15	7	8	9	10	11	12	13	14	15	16	17	18
64-127	7	17	8	9	10	11	12	13	14	15	16	17	18	19
128-255	8	19	9	10	11	12	13	14	15	16	17	18	19	20

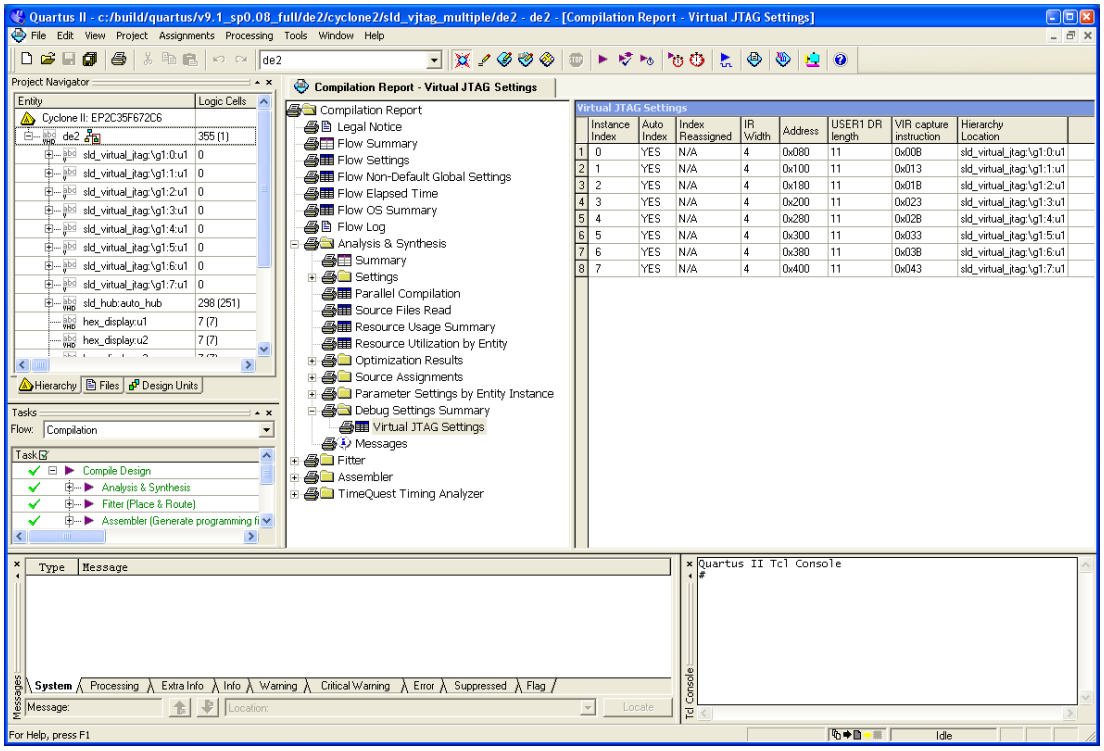


(a)

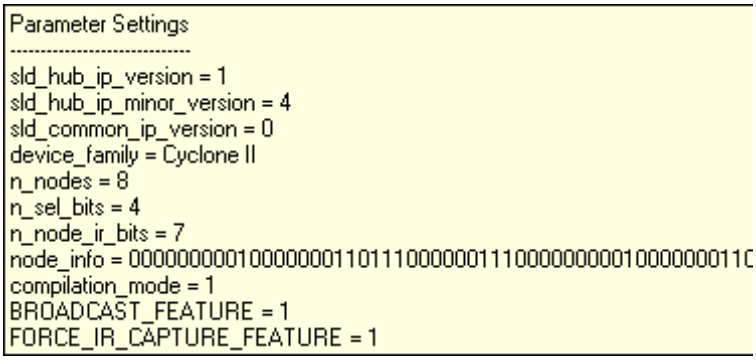


(b)

Figure 22: Quartus II Virtual JTAG Node Information for a single instance design; (a) Virtual JTAG settings compilation report, and (b) SLD hub tooltip information.



(a)



(b)

Figure 23: Quartus II Virtual JTAG Node Information for a multiple instance design; (a) Virtual JTAG settings compilation report, and (b) SLD hub tooltip information.

Table 3: `sld_vjtag_to_gpio` Tcl procedures.

Command	Description
`jtag_open`	Access JTAG on the DE2 board.
`jtag_close`	End JTAG access.
`read_idcode`	Read the Cyclone II JTAG IDCODE, i.e., 020B40DDh.
`read_usercode`	Read the user-defined JTAG USERCODE, eg., 12345678h.
`pulse_nconfig`	Reset the FPGA configuration.
`print_hub_info`	Print the Virtual JTAG `HUB_INFO`.
`print_node_info`	Print the Virtual JTAG `SLD_NODE_INFO`.
`jtag_vir`	Virtual IR-shift.
`jtag_vdr`	Virtual DR-shift.

The Tcl Virtual JTAG procedures use the instance index to access (address) each component instance; the reports in Figures 22 and 23 shows the instance index for each `sld_virtual_jtag` component in the design. The reports also show the instance capture instruction, address, and USER1 DR length (VIR width).

Figure 19 shows the encoding for Virtual instructions. Table 4 shows example encodings. The `SLD_IR_WIDTH` value selected for these examples was such that the VIR width would be determined by the capture instruction width, i.e., in Table 2 the blue values set the width, eg., `SLD_IR_WIDTH` values of 1, 2, 3, or 4 can be used to generate the example encodings in Table 4. The hexadecimal values for the capture instruction and node address reported in Figures 22 and 23 correspond to the $N = 1$ and $N = 8$ values in Table 4.

The examples in Table 4 show the encoding for the capture instruction and the user address. The capture instruction for each node consists of the n-bit hub address (zero), followed by the n-bit node address, and then the 3-bit capture code 011b. Since the VIR width for these examples was determined by the capture instruction width, no zero padding is required (see Figure 19). The user address in Table 4 consists of the n-bit node address, followed by zeros. A practical use of this address is in the formation of user instructions; the first n-bits is the node address, and the last `SLD_IR_WIDTH` bits encode the user instruction, with any unused bits between the user instruction and node address being set to zero.

The JTAG properties of a design can be queried using Tcl procedures. Table 3 shows the Tcl procedures implemented in `sld_vjtag_to_gpio/scripts/vjtag_cmds.tcl`. The JTAG `HUB_INFO` and `SLD_NODE_INFO` commands were issued to the designs shown in Tables 2 and 4 to confirm the table values.

Table 4: VIR instruction encoding examples (for interpreting Quartus reported values).

Number of nodes, N	VIR hub/node address width, n	VIR width, $(m+n)$	Node (instance) Number	Capture Instruction		Node (instance) Address	
				(binary)	(hex)	(binary)	(hex)
1	1	5	1	0_1_011b	0Bh	1_0_000b	10h
2	2	7	1	00_01_011b	0Bh	01_00_000b	20h
			2	00_10_011b	13h	10_00_000b	40h
4	3	9	1	000_001_011b	00Bh	001_000_000b	040h
			2	000_010_011b	013h	010_000_000b	080h
			3	000_011_011b	01Bh	011_000_000b	0C0h
			4	000_100_011b	023h	100_000_000b	100h
8	4	11	1	0000_0001_011b	00Bh	0001_0000_000b	080h
			2	0000_0010_011b	013h	0010_0000_000b	100h
			3	0000_0011_011b	01Bh	0011_0000_000b	180h
			4	0000_0100_011b	023h	0100_0000_000b	200h
			5	0000_0101_011b	02Bh	0101_0000_000b	280h
			6	0000_0110_011b	033h	0110_0000_000b	300h
			7	0000_0111_011b	03Bh	0111_0000_000b	380h
			8	0000_1000_011b	043h	1000_0000_000b	400h

Appendix A of the Virtual JTAG User Guide [1] incorrectly states that that `HUB_INFO` command returns the total VIR width. Figure 20 shows the instruction *actually* returns the VIR m-width. The error in the documentation was confirmed in hardware using the `HUB_INFO` command. For example, the `sld_vjtag_multiple` design with 4 instances and an `SLD_IR_WIDTH` of 8, has `HUB_INFO`;

```
tcl> print_hub_info
         Hub info: 0x08206E08
      VIR m-width: 8
  Manufacturer ID: 0x6E
  Number of nodes: 4
       IP Version: 1
```

Table 2 indicates that the total VIR width is determined by the user instruction and is 11-bits, and $n = 3$, so $m = 11 - 3 = 8$, and this is what the `HUB_INFO` instruction returns. As another example, a design with 16 instances and an `SLD_IR_WIDTH` of 5, has `HUB_INFO`;

```
tcl> print_hub_info
         Hub info: 0x08806E08
      VIR m-width: 8
  Manufacturer ID: 0x6E
  Number of nodes: 16
       IP Version: 1
```

Table 2 indicates that the total VIR width is determined by the capture instruction and is 13-bits, and $n = 5$, so $m = 13 - 5 = 8$, and again this is what the `HUB_INFO` instruction returns.

2.7 Multiple SLD debug component testing

This document has described the Virtual JTAG component `sld_virtual_jtag` instruction encoding properties. The JTAG properties of other SLD components are not well documented by Altera. To investigate the effect other SLD components have on VIR encoding, the design `sld_vjtag_custom` was created. The design can be configured to contain multiple SLD components;

- One or two instances of `sld_virtual_jtag`.
- One or two instances of `sld_virtual_jtag_basic`.
- An instance of `sld_virtual_jtag` and `sld_virtual_jtag_basic`.
- An SOPC system containing a JTAG-to-Avalon bridge.
- A SignalTap II logic analyzer instance.

Table 5: Multiple SLD component VIR encoding results.

Example Number	SignalTap Enabled	SOPC Enabled	Number of Nodes, N	Virtual Instruction Register		
				n	m	$(m+n)$
1 or 2	×	×	1	1	4	5
	×	✓	2	2	5	7
	✓	×	2	2	8	10
	✓	✓	3	2	8	10
3, 4, or 5	×	×	2	2	5	7
	×	✓	3	2	5	7
	✓	×	3	2	8	10
	✓	✓	4	2	8	11

The component `sld_virtual_jtag_basic` is the undocumented component used to implement the `sld_virtual_jtag` component (it can be seen in the hierarchy display in Figure 22) and is the basis of the JTAG PHY used in the JTAG-to-Avalon bridges. The `sld_virtual_jtag_basic` component allows you to specify a manufacturer ID code, a device type ID code, and a version number. This information can then be reported using the `SLD_NODE_INFO` instruction. Unfortunately, once you change the codes from the Altera values, the Virtual JTAG Tcl procedures no longer work, i.e., you can no longer use `device_virtual_ir_shift` and `device_virtual_dr_shift`, but must construct commands using the JTAG Tcl instructions `device_ir_shift` and `device_dr_shift`, which means that user Tcl code has to deal with low-level VIR encoding. Section 4 describes a general-purpose component based on the `sld_virtual_jtag` component that provides identification registers, while allowing the use of the Virtual JTAG Tcl procedures.

Table 5 shows synthesis results from the `sld_vjtag_custom` design. The results are displayed in two blocks; when the `sld_vjtag_custom` generic `EXAMPLE` is set to 1 or 2, the design contains a single `sld_virtual_jtag` or `sld_virtual_jtag_basic` component, and when the generic is set to 3, 4, or 5, the design contains two components. The four results for each of the two blocks indicate the change in VIR parameters as either the SOPC system or SignalTap II instance is, or both are, enabled in the design. The SOPC system contains a JTAG-to-Avalon bridge. The bridge is implemented using an `sld_virtual_jtag_basic` component with an `SLD_IR_WIDTH` of 2; which is small enough that its instruction width is capture instruction width dominated. Table 5 shows that when the design contains the Virtual JTAG instance(s) and the SOPC system, the VIR width changes to accommodate the change in node address bits. When the SignalTap II instance is added, the VIR width increases to accommodate the (undocumented) instruction width required by the SignalTap II instance, which in this case is an effective `SLD_IR_WIDTH` of 8-bits.

Tcl scripts that access Virtual JTAG components must interrogate the hardware before issuing instructions, since the presence of a SignalTap II logic analyzer instance will affect the instruction encoding, and analyzer instances can be added to a design independently of the design source, eg., VHDL. This is a key argument for designing JTAG components around the `sld_virtual_jtag` component, so that the Virtual JTAG Tcl commands can be used (since they take care of hub interrogation).

3 Bus Functional Model (BFM) Simulation

This section describes the design of a functionally correct simulation model, i.e., the model generates transaction waveforms that match logic analyzer traces shown in Sections 2.4 and 2.5, with the exception of not reproducing the logical error present in the Altera IP core, shown in Figures 15, 16 and 18. The JTAG simulation infrastructure consists of;

- A bus functional model (BFM).

  The BFM consists of a VHDL server component, and a VHDL package containing client/server communications procedures.

  Testbenches constructed using the BFM consist of a testcase generator, the BFM server, and the device, or devices, under test (a design containing one or more of the Virtual JTAG simulation models). The testcase generator (the client) communicates with the BFM server using VHDL procedures analogous to the Virtual JTAG Tcl procedures. Procedure parameters are passed between the client and server using a VHDL resolved signal (read the BFM package source for details). The server receives client requests and generates JTAG transactions on the server output signals TCK, TMS, and TDI. The server outputs route to the inputs on each Virtual JTAG model instance. The Virtual JTAG model TDO outputs route back to the BFM server. The BFM server selects the appropriate TDO input based on the instance index indicated by the client (analogous to the way the instance index argument is used by the Tcl procedures).

  The BFM server is analogous to the Altera `sld_hub`.

- A Virtual JTAG model.

  The top-level component `altera_sld_virtual_jtag` can be used to replace the Altera Virtual JTAG component. The new component contains *external* JTAG ports that are used only during simulation (to connect to the BFM server). The additional port definitions are contained within an Altera synthesis directive[5], so that during synthesis the ports are not interpreted by Quartus. Within the component, the design instantiates either the simulation component `altera_sld_virtual_jtag_model` or the synthesis component `sld_virtual_jtag`. Altera synthesis directives are again used[6], so that Quartus only sees the synthesis model.

The `altera_sld_virtual_jtag_model` component contains an `altera_jtag_tap` component that tracks the state of the JTAG transaction, and an instruction register for determining when the USER1 and USER0 instructions have been issued. The component *combinatorially* generates the virtual JTAG one-hot outputs using the instruction encoding and the JTAG state. The logic analyzer traces show that the VIR register does not change when a capture instruction is issued, only when the user instruction is issued, to the model reproduces this sequence.

The simulation model architecture does not match the hardware architecture; a simulation design containing multiple instances of the simulation model will contain redundant copies of the 10-bit JTAG instruction register for USER0/USER1 detection. However, containing this repeated logic within the simulation model makes the simulation infrastructure simpler, as the connection between the BFM server and the models reduces to the JTAG signals, rather than the JTAG signals, USER0 and USER1 decode signals, and Virtual Instruction Register buses (one for each instance). The compilation reports for the single and multiple instance designs in Figures 22 and 23 show that the `sld_virtual_jtag` components consume few or zero logic resources, whereas the `sld_hub` consumes an increasing amount of logic (96 logic cells, then 298 logic cells). This implies that the `sld_hub` is where all the Virtual JTAG logic resides, and that the `sld_virtual_jtag` component is simply the user-visible interface to the hub logic.

[5] `altera translate_off` and `altera translate_on`.

[6] `synthesis read_comments_as_HDL on` and `synthesis read_comments_as_HDL off`.

The simulation model does not implement the `HUB_INFO` and `SLD_NODE_INFO` registers. The logic is not difficult to model, however, the testbench needs the generics that these instructions encode, so its just as easy for the client/server procedures to use the generics directly (otherwise the client procedures would have to first interrogate the hub for the information). The simulation models use generics to construct VIR encode and decode logic that matches the hardware implementation.

Figure 24 shows the simulation model Virtual IR-shift sequence. Comparison of the figure to the logic analyzer traces in Figures 12 and 17, shows that the simulation model generates the correct JTAG sequence. Contrast this to the incorrect simulation results produced by the Altera model in Figure 4.

Figure 25 shows the simulation model Virtual DR-shift sequence. Comparison of the figure to the logic analyzer traces in Figures 15 and 18, shows that the simulation model generates the correct JTAG sequence. Contrast this to the incorrect simulation results produced by the Altera model in Figure 6.

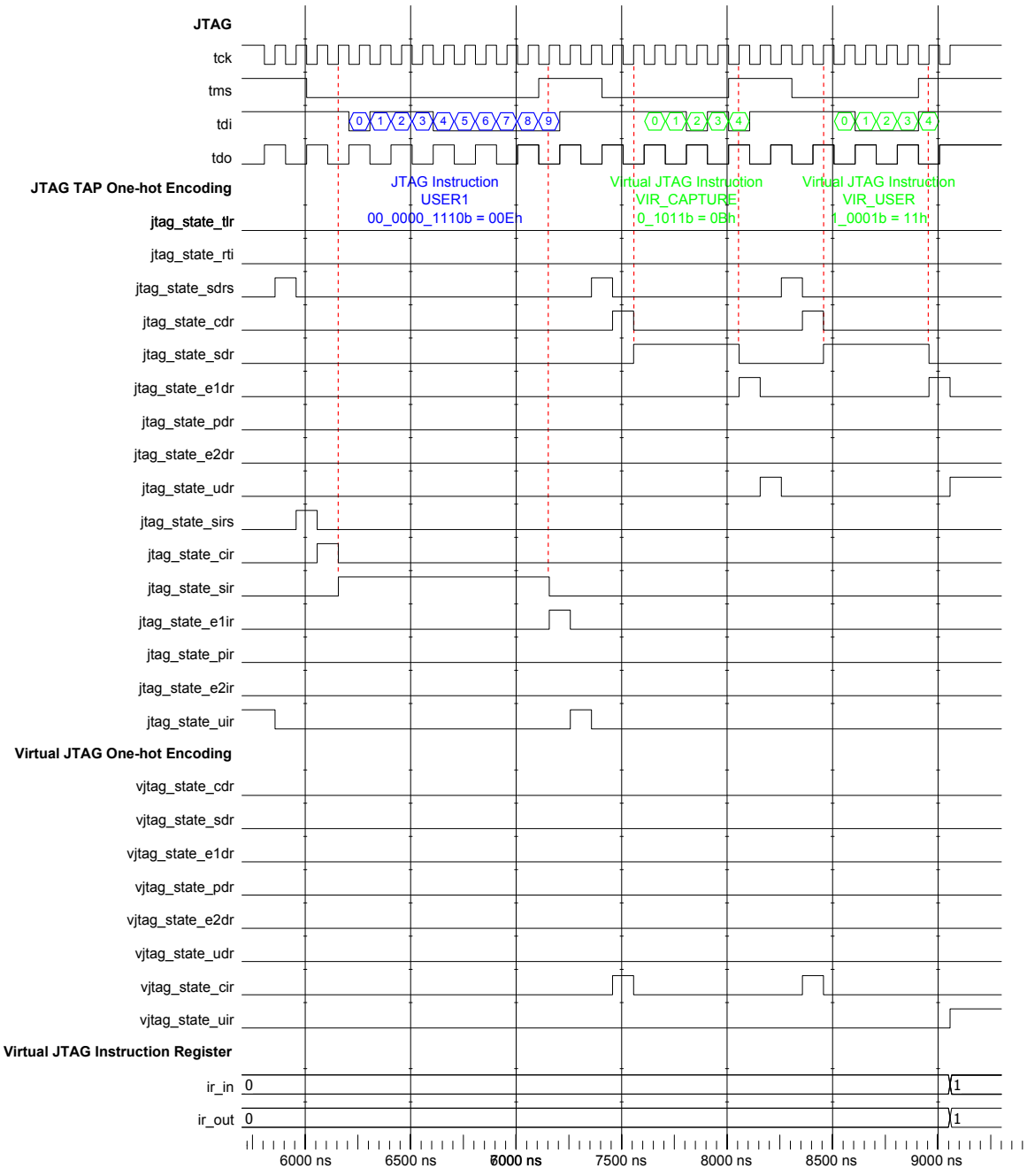


Figure 24: Virtual JTAG simulation; Virtual IR-shift sequence.

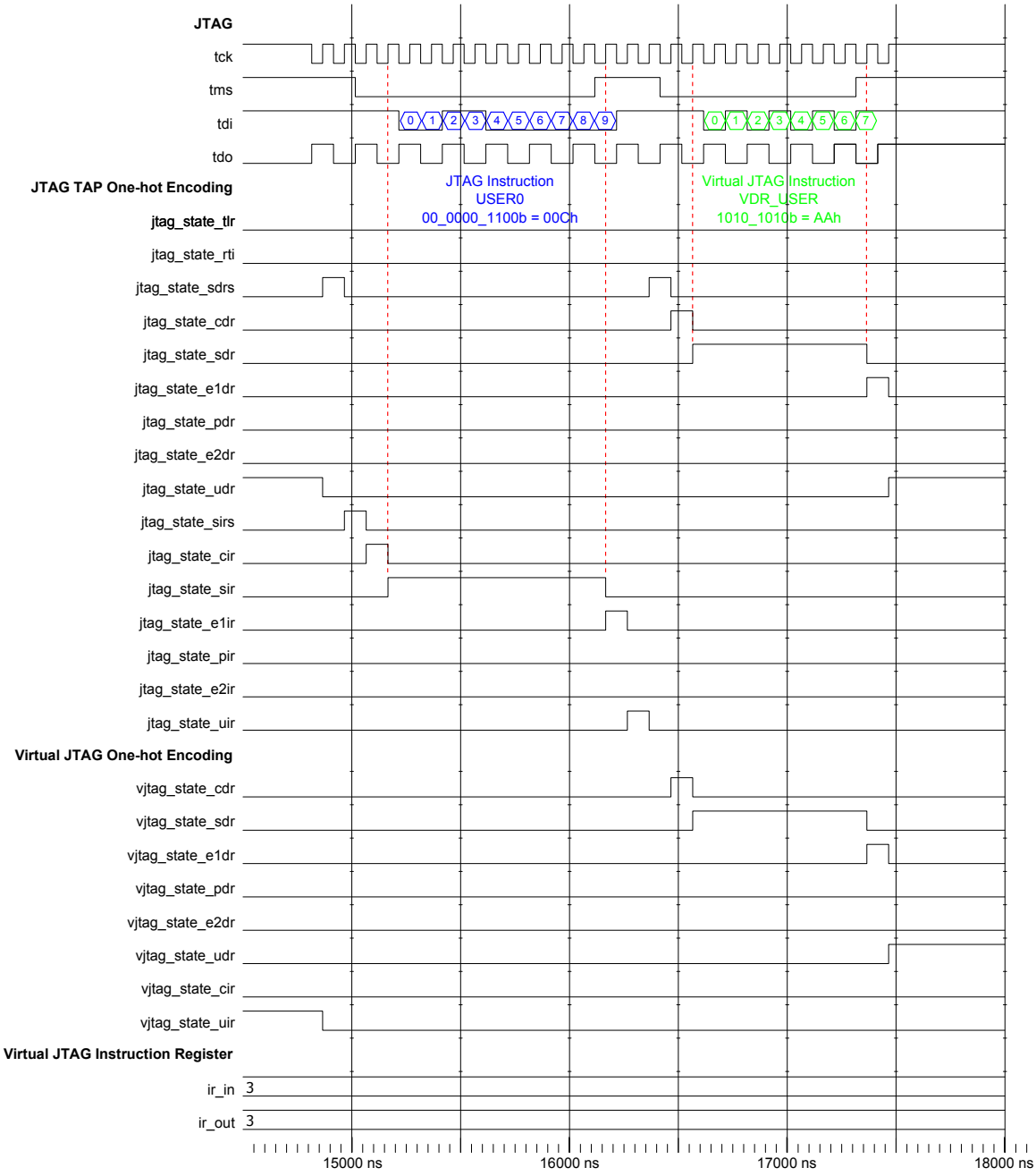


Figure 25: Virtual JTAG simulation; Virtual DR-shift sequence.

Table 6: General-purpose identifiable Virtual JTAG component, `altera_vjtag`, generics.

Generic	Description
User Interface Parameters:	
`UIR_WIDTH`	User instruction width.
Identification Parameters:	
`VID`	16-bit Vendor ID (VID) code.
`PID`	16-bit Product ID (PID) code.
`VERSION`	16-bit Version.
`NODE_INDEX`	The node index (reported by `SLD_NODE_INFO`).
Simulation Parameters:	
`MAX_SLD_IR_WIDTH`	Maximum value of `SLD_IR_WIDTH`.
`NUMBER_OF_NODES`	Number of Virtual JTAG nodes.

4 General-purpose identifiable Virtual JTAG component

Custom components based on the `sld_virtual_jtag` component require some form of identification to uniquely identify each instance. The identification information can then be used by Tcl scripts provided in a general-purpose Tcl package that can be reused between designs. For example, consider a user-defined JTAG-to-Avalon master bridge component. That component would be supplied along with a Tcl package containing procedures for performing Avalon bus read and write transactions. The `SLD_NODE_INFO` command can be used to identify the index of an `sld_virtual_jtag` instance within a design, but it can not provide information on what that component is used to implement. A design-specific Tcl script could be written that provides Avalon read and write transaction to say instance index zero, however, this does not allow for design reuse (other than by copying the Tcl script). The issue of component identification becomes more difficult when a design is created with SOPC Builder, because when the `sld_virtual_jtag` components use automatic instance indexing, the component indexes change depending on the order of components in the generated SOPC system (an ordering that the user can not easily control).

Altera gets around this identification issue by using the `sld_virtual_jtag_basic` component to implement the `sld_virtual_jtag` and JTAG-to-Avalon bridge components with unique identification information. The Altera Tcl procedures then check that identification information before generating component-specific JTAG transactions. A custom user component could be built using the `sld_virtual_jtag_basic` component, however, as discussed in Section 2.7, if you change the Virtual JTAG component identification code, the Virtual JTAG Tcl procedures no longer work. The Virtual JTAG Tcl procedures hide the complexities of virtual instruction encoding from the user, and it would be useful to preserve this functionality (rather than re-implement it using the low-level JTAG Tcl IR-shift and DR-shift procedures). Hence, a custom Virtual JTAG component should be implemented using the `sld_virtual_jtag` component.

Figure 26 shows a block diagram of the `altera_vjtag` component which implements a general-purpose identifiable Virtual JTAG component. The figure caption describes the internal implementation of the component (the VHDL source code can be reviewed for more details). Table 6 describes

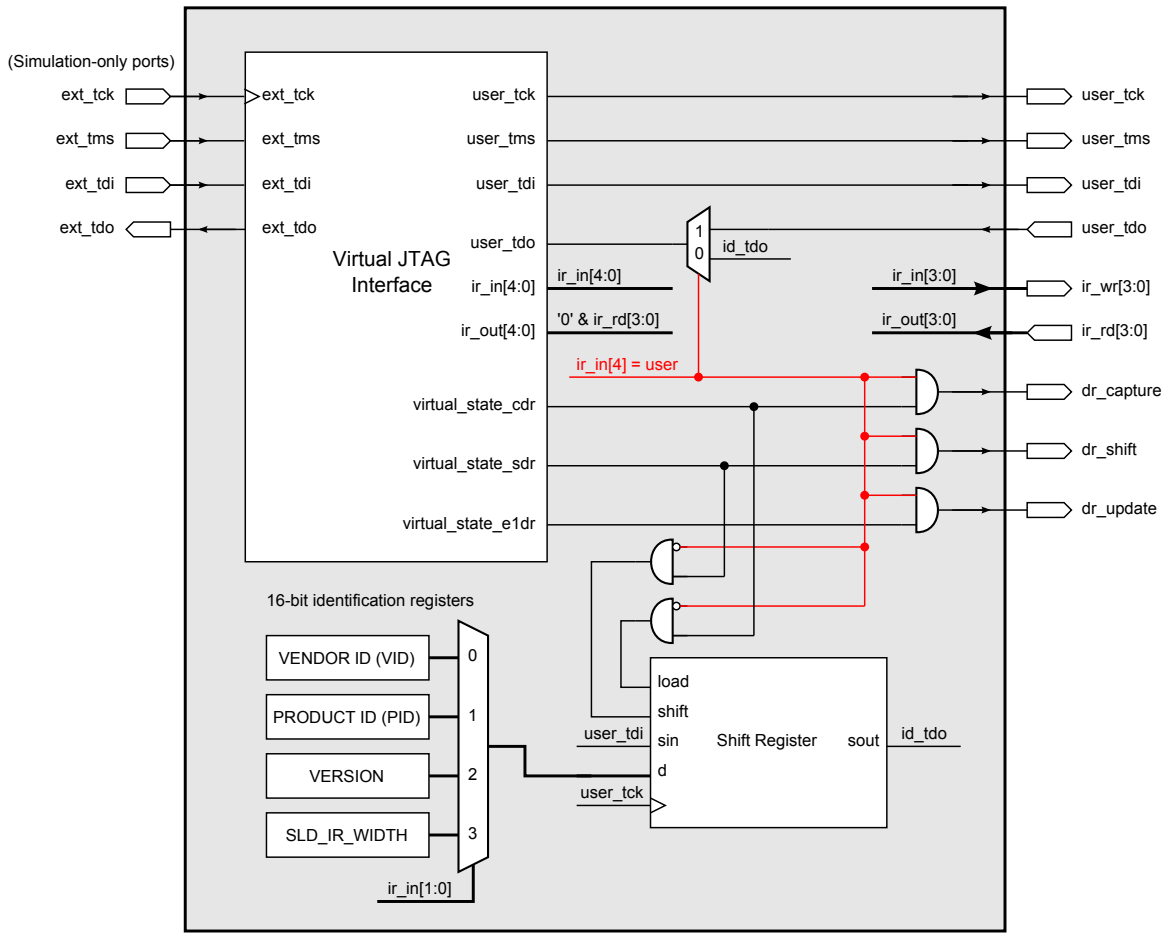


Figure 26: General-purpose identifiable Virtual JTAG component, `altera_vjtag`. The diagram shows a component instance with a user-defined instruction width, `UIR_WIDTH`, of 4. The internal implementation instantiates a Virtual JTAG component with an `SLD_IR_WIDTH` of 5. The most significant bit of the instruction register, `ir_in[4]`, is defined to be the `user` mode select bit; when `user` is high the user TDO, `user_tdo`, drives the JTAG TDO signal, but when `user` is low the identification TDO, `id_tdo`, drives the JTAG TDO signal, with `ir_in[1:0]` selecting the identification register. The internal shift-register load and shift control signals only assert when the corresponding virtual state signal asserts and `user` is low. The external shift-register controls `dr_capture`, `dr_shift`, and `dr_update` only assert when the corresponding virtual state signal asserts and `user` is high.

the generics used to configure the component.

Auto-detection of components based on the `altera_vjtag` component proceeds as follows;

1. For each instance of the `sld_virtual_jtag` component identified by `SLD_NODE_INFO`, issue a virtual user instruction of zero. This sets the `ir_in[]` port to all zeros, i.e., `user` low, so that the identification registers are selected in the component JTAG chain.
2. Read the 16-bit Vendor ID using a 16-bit Virtual DR-shift.
3. If the Vendor ID code is recognized, then issue virtual user instructions of 1, 2, and 3, and use 16-bit Virtual DR-shifts to read the other identification registers.
4. Select the user registers by setting the `user` instruction bit high. The position of this bit is determined from the value read from the `SLD_IR_WIDTH` identification register, i.e., `user = ir_in[SLD_IR_WIDTH-1]`.
5. Perform further device specific register accesses.
6. Enable the use of component-specific Tcl procedures with this instance index.

A key aspect of the identification encoding scheme is to use the `user` low value to select the identification registers. The position of the `user` bit within the instruction register is not known until the identification registers are read. The VIR m-width from the `HUB_INFO` instruction can not be used for this purpose, since that value represent the *maximum* value of `SLD_IR_WIDTH` used in all component instances within a design, not the specific instance of the `altera_vjtag` component.

Another subtle aspect of the `altera_vjtag` component is the use of the virtual state signals; the use of the capture and shift states is fairly obvious, but the use of `virtual_state_e1dr` to generate the update pulse is less so. The reason this signal is used, rather than the `virtual_state_udr` signal, is that `virtual_state_e1dr` is pulsed for one JTAG clock period following the DR-shift sequence, whereas `virtual_state_udr` goes high, and stays high because once the JTAG TAP controller reaches that state, the external JTAG controller stops generating the JTAG clock, eg., see the JTAG and Virtual JTAG state assertions in Figure 25.

The testbench `altera_vjtag_tb` instantitates the testcase `altera_vjtag_tc`, the JTAG BFM server, and an instance of the `altera_vjtag` component with its user instruction register looped-back on itself, and user TDO fed by a signal that toggles every falling edge of TCK. The testcase generator demonstrates the component identification sequence, and demonstrates the use of VHDL assertions for verification. This testbench can be used as the basis for any custom `altera_vjtag` component. Section 5 contains additional example applications of the `altera_vjtag` component.

TODO:

- Tcl package for the core commands for this component, eg., `package require vjtag`. Create a version of `print_identification` that loops over all nodes within a design. Create an example test design containing identifiable and non-identifiable `sld_virtual_jtag` components.
- Get the `altera_vjtag`-based JTAG-to-Avalon bridge working and documented.

5 Design Examples

This section contains synthesizeable Virtual JTAG example designs.

Mention the DE2 board `sld_virtual_jtag` based designs, and the DE2 `altera_vjtag` design.

Clock-domain crossing example.

Provide a couple of basic examples, and then example traces from a custom JTAG-to-Avalon master. Then just refer the reader to my Avalon document.

Tcl GUI to control the DE2 board? Maybe save that for use with the JTAG-to-Avalon bridge.

How fast is TCK on the DE2 board? How about on the BeMicro?

Actually, what examples did I try on the BeMicro? Did I check them into CVS? Didn't I use the VHDL keep signals attributes in the example design?

External TCK/TMS/TDI/TDO signal capture - route to GPIO pins on something (S4GXDK) and trace VIR sequences.

5.1 User Guide Examples

Review the user guide examples; DCFIFO and a read-only timestamp or ID - right? The example was lame in that you'd want that register to be read via JTAG or onboard processor, so really the interface should be an Avalon Master.

Create a simulation version of the examples? Did they include clock domain synchronizers?

5.2 LED control and switch status

The DE2 project connects the Virtual instruction output port to the red and green LEDs, and the Virtual instruction input port to the switches. The LEDs on the board can be illuminated, and the switch state read-back, via the sequence;

```
tcl> jtag_open
JTAG: USB-Blaster [USB-0], FPGA: @1: EP2C35 (0x020B40DD)
tcl> device_lock -timeout 10000
tcl> device_virtual_ir_shift -instance_index 0 -ir_value 7
5
tcl> device_unlock
tcl> jtag_close
```

where since the Virtual instruction width on the JTAG instance was set to 3-bits, three of the red and green LEDs are illuminated by this instruction. The 3-bit switch state read-back was 5, i.e., two switches were on and one was off. The `de2.vhd` file contains a `VIR_WIDTH` parameter that can be set from 1 to 32-bits (per the limits on this value imposed by the Virtual JTAG component). The LEDs and switches will be wired according to the `VIR_WIDTH` (until all LEDs and switches have been connected).

5.3 Control and status register

Simple JTAG component applications.

Using an `sld_virtual_jtag` component without additional logic; connect `ir_in` to outputs, eg. LEDs, and `ir_out` to inputs, eg., switches. Or if you want write/read bits connect `ir_in` bits to the LED, and connect it to `ir_out` for read-back.

`de2/cyclone2/sld_vjtag_to_gpio/` already shows an example of this approach.

5.4 Read/write registers block

Draw a nice version of my read/write registers block figure. Point out that the write-enable decoding disappears when the registers become read-only, and that is how the `altera_vjtag` component identification registers works - create a figure for the read-only registers.

SignalTap II trace of register accesses?

Timing analysis; really needs to use an FPGA clock. Eg. synchronize the `virtual_jtag_uir` to the FPGA clock via a pulse synchronizer, use that synchronizer to enable an address register, and register the output of the registers block. Then you can apply a timing constraint on the FPGA clock domain and cut paths to the JTAG interface.

5.5 Clock and reset checking

Connect the clock or reset for a design in the TDO path and perform a Virtual DR-shift operation of say 32-bits. If the value read back is all zeros or ones, then the signal is at a static level, otherwise its toggling.

5.6 Clock-domain crossing

`de2/altera_vjtag/` just uses a single clock domain, that of JTAG. Extend this example to be able to read the state of the reset control (one of the keys) and to read the clock (`clk_50MHz`), or use my pulse-count logic, eg. toggle a signal using JTAG, and see if the counter enabled by that signal increments. If it does, and the count is fairly large (consistent with the expected clock frequency) then there is a clock present. Also try the simple 'read the clock value' approach to detect if its toggling.

5.7 Address and data bus interface

`vjtag_to_registers_bridge` Like my VJTAG component.

Encode `ADDR_WIDTH`, `BYTEEN_WIDTH`, `DATA_WIDTH` in some read-only registers, then provide a control/status register (or at least locations for them for the test examples), eg. a 32-bit read/write control register and a 32-bit read-only status register. I'd then use the same scheme for my Avalon bridge (but restrict bus parameter values to those in the Avalon specification, eg. 8-bit increments). The control register will need a bit to indicate whether the transaction is a read or write. Add another control bit that determines the addressing scheme; fixed address or automatic address increment. The address control bit will allow you to write a start address (which will load a counter), and then write data to increasing addresses, and similar for reads (saves on JTAG activity). I'm not sure if I have enough JTAG clocks to make this work, so perhaps only implement it for the bridge that uses two clocks. Now that I have a simulation model, I can figure out whether this will work with just the JTAG clock.

Refer to the Avalon bridges, but stick them in my Avalon document: `vjtag_to_avalon_mm_bridge`, `vjtag_to_avalon_st_bridge`, JTAG-to-Avalon Bridges done right, i.e., you can simulate with them. The Altera JTAG-to-Avalon bridge can only be simulated using Verilog PLI and the system console, and that interface is unsupported and barely documented (the Altera Wiki has instructions on what to do).

5.8 Dual-port RAM test

Assuming the RAM clock is faster than the JTAG clock, then the fact that when the RAM read data takes a couple of clocks before it is valid is generally acceptable. Eg. RAM address, byte-enable, asserted, and then by the time the read command is decoded from the JTAG interface, the read data has been valid for ages. However, the RAM clock could be made so slow that this is violated.

When accessing a device that can take a while to response, some form of read-data-ready status register bit is required.

`de2/cyclone2/vjtag_ram_example/`

5.9 Dual-clock FIFO test

Same deal as with RAM, but the JTAG address space is slit to decode to the FIFO input or output. The FIFO status signals can be routed to a read-only status register location. The FIFO can be operated in dual clock mode, and as long as those clocks are faster then the JTAG clock, the JTAG interface will work fine. You can write to the FIFO while monitoring the used value and the FIFO status outputs, and the drain the FIFO. Write a simulation and synthesis test.

`de2/cyclone2/vjtag_fifo_example/`

A Altera Tool Versions

Quartus 9.1, Quartus 10.1, Modelsim SE 6.5b (full version), Modelsim Altera-Edition (use both the versions expected to be used with Quartus 9.1 and 10.1).

Tcl scripts create version specific build directories so all these tools can be installed at the same time, and all the build examples can exist at the same time.

B Source Code Description

VHDL and Tcl source.

Describe the use of the synthesis scripts for building a board design.

Describe the use of a simulation script for setting up procedures that run a simulation.

List the project names and folders - a table perhaps with a description of each project?

Describe how to create a new DE2 project, by copying the DE2 `basic` design.

- New project folders were created in the DE2 board project directory, i.e.,

`$VHDL/boards/de2/cyclone2/sld_virtual_jtag/`	Project folder
`$VHDL/boards/de2/cyclone2/sld_virtual_jtag/src/`	VHDL source
`$VHDL/boards/de2/cyclone2/sld_virtual_jtag/test/`	VHDL testbench
`$VHDL/boards/de2/cyclone2/sld_virtual_jtag/scripts/`	Scripts

- The top-level VHDL file and synthesis script were copied from the `basic` design, i.e.,

  `$VHDL/boards/de2/cyclone2/sld_virtual_jtag/src/de2.vhd`
  `$VHDL/boards/de2/cyclone2/sld_virtual_jtag/scripts/synth.tcl`.

The `basic` design contains a VHDL entity with ports for all the pins used on the DE2 board, along with a basic architecture body that connects switches to hexadecimal displays and deasserts control signals to external devices.

C JTAG Timing Constraints (TimeQuest SDC)

Check the Cyclone II and II databooks, I'm sure that JTAG signals are supposed to be clocked like SPI, i.e., data changes on the falling-edge of the clock, and is captured on the rising-edge. Comment that the logic analyzer traces show this is not the case. Screen shots showing the SDC timing constraints for JTAG. What is the timing margin on the signals?

Create a generic JTAG script. How can the script determine if the JTAG signals have been used? You should be able to look for signals and then if they are found in the design, apply a constraint.

Point out that Altera's SDC examples for JTAG are not consistent with the falling-edge clocking.

How about their STAPL documentation, how does that say the JTAG signals should be clocked? Look at my Byteblaster and COBRA example code. What did DaveM do in the CARMA code?

References

[1] Altera Corporation. Virtual JTAG Megafunction User's Guide (version 2.0), December 2008. http://www.altera.com/.

[2] Altera Corporation. Quartus II Scripting Reference Manual (version 9.1), 2009. http://www.altera.com/.

[3] Altera Corporation. Quartus II Handbook (version 10.0), 2010. http://www.altera.com/.