Altera JTAG-to-Avalon Analysis

D. W. Hawkins (dwh@ovro.caltech.edu)

February 8, 2012

Contents

1 Introduction **3**

2 JTAG-to-Avalon Protocols **5**
2.1 JTAG-to-Avalon-ST 5
2.2 JTAG-to-Avalon-MM 7
2.3 Altera Documentation 10

3 Software **11**
3.1 SystemConsole 11
3.2 Command-line tools (`quartus_stp`) 11

4 Hardware Tests **12**
4.1 JTAG Node 14
4.2 JTAG-to-Avalon-ST 18
4.3 JTAG-to-Avalon-MM 26

A Project synthesis and simulation **38**
A.1 Synthesis 38
A.2 Simulation 39

B JTAG-to-Avalon source, software, and hardware bugs **40**

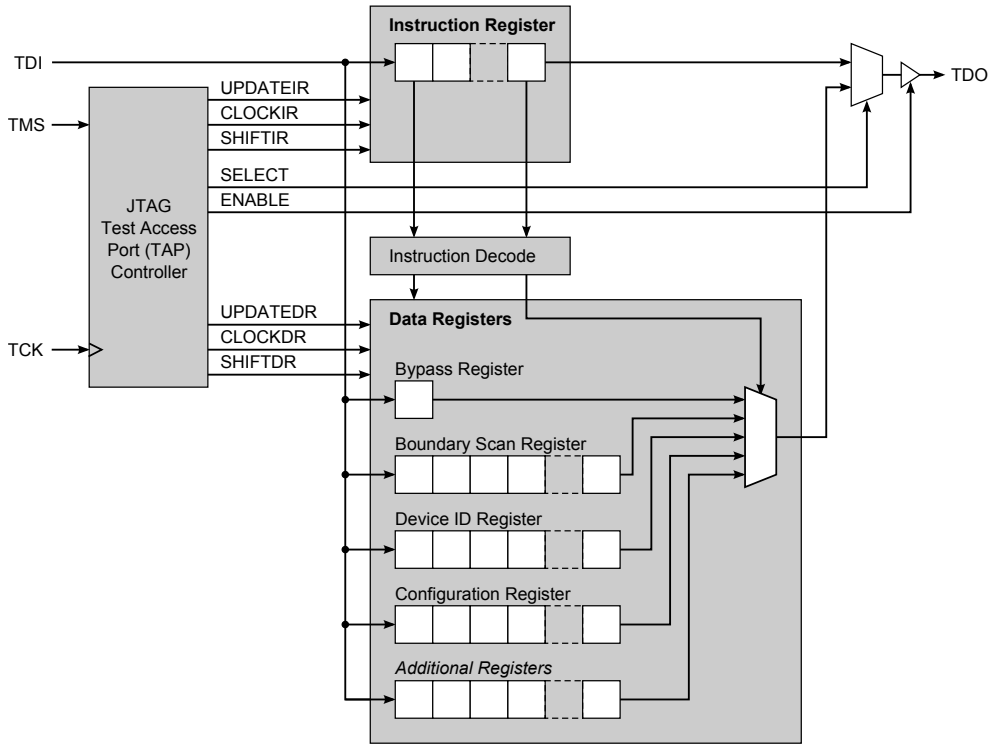

Figure 1: Altera JTAG (IEEE 1149.1) Interface. JTAG transactions consist of writing to the instruction register followed by writing to or reading from the data register.

1 Introduction

Efficient development and debugging of Altera FPGA based designs requires a detailed knowledge of the JTAG interface. The Altera FPGA JTAG interface has several uses;

- The Quartus II programmer uses JTAG for FPGA configuration.
- The SignalTap II logic analyzer interface uses JTAG to communicate with the SignalTap II logic embedded in the FPGA.
- The NIOS II debug and JTAG-UART communicate with the NIOS II IDE via JTAG.
- The Qsys and SOPC Builder JTAG-to-Avalon-MM bridge component uses JTAG to generate Avalon-MM master transactions.
- The SLD Virtual JTAG component [4] can be used to create custom components accessible via JTAG.

The JTAG interface is a *synchronous* serial interface consisting of the JTAG clock (TCK), mode select (TMS), serial data in (TDI), and serial data out (TDO). Figure 1 shows a diagram of the Altera JTAG interface [1], while Figure 2 shows the JTAG TAP controller finite state machine (FSM).

This document is the *missing manual* for Altera's JTAG-to-Avalon interfaces, it provides simulation details and hardware measurements.

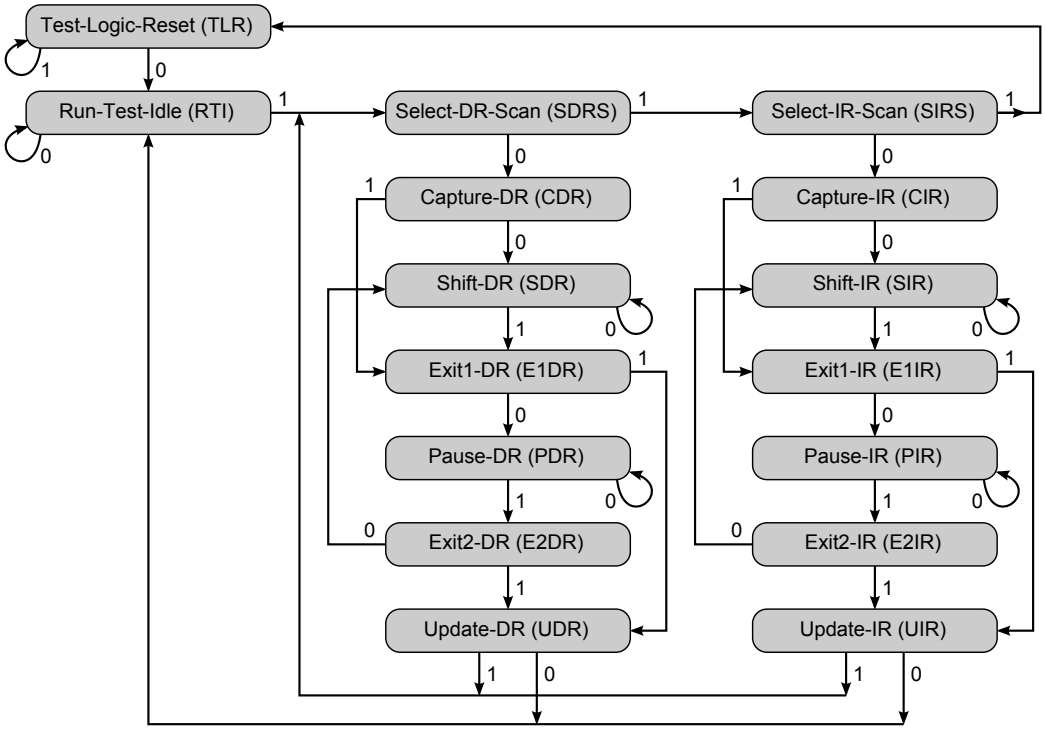

Figure 2: JTAG (IEEE 1149.1) Test Access Port (TAP) Controller. State transitions are controlled by the JTAG clock (TCK) and mode select (TMS) control.

2 JTAG-to-Avalon Protocols

The Altera Avalon Specification [5, 6] defines two interfaces;

- Avalon Streaming (Avalon-ST) Interface
- Avalon Memory-Mapped (Avalon-MM) Interface

The JTAG-to-Avalon-MM master consists of a JTAG-to-Avalon-ST interface that converts JTAG serial transactions into byte streams in and out of the design, and bytes-to-packets conversion logic that decodes and encodes a binary protocol transported over the byte streams. The binary protocol encodes whether to perform an Avalon-MM read or write transaction, and the response for each transaction type.

Access from the JTAG interface to the FPGA fabric is coordinated by the *System-Level Debug* (SLD) JTAG hub. This hub is automatically generated during synthesis (the component appears the Quartus II hierarchy display). The JTAG-to-Avalon components are based on the Altera *SLD Virtual JTAG* Component [4] (they are actually based on a variation of this component that allows a manufacturer ID, component ID and version to be specified). The Virtual JTAG component uses two JTAG instructions to create a Virtual Instruction Register (VIR) and a Virtual Data Register (VDR) that exist in the FPGA fabric. The JTAG-to-Avalon components use the VIR for selecting the component operating mode, and use the VDR path for mode-specific data registers.

2.1 JTAG-to-Avalon-ST

The JTAG-to-Avalon-ST bridge uses the Virtual JTAG interface to stream bytes into and out of the FPGA fabric, and to access control and status registers. Table 1 shows the VIR modes and the register bits.

In *DATA* mode, the JTAG-to-Avalon-ST bridge generates byte-streams from the host-to-device and from the device-to-host. Each byte-stream starts with a 16-bit header, and is followed by encoded data. Table 2 shows the headers (see Section 4 for encoding details), while Table 3 shows the data protocol. The *IDLE* code indicates *no data* and can appear in the host-to-device or device-to-host byte-streams. The *IDLE* code is used when the Avalon-ST byte-stream is unidirectional, eg., when the host is sending data to the device and the device has no response data, the device sends *IDLE* codes, and when the host needs to receive a device response and has no data to send, the host sends *IDLE* codes.

The JTAG-to-Avalon-ST interface is designed to transport binary byte-streams, where every possible byte value can be transmitted. Because the *IDLE* code uses a byte value, an *ESCAPE* code is needed. The *ESCAPE* code is used as an indicator within the byte-stream that a protocol code is being transmitted as data. The binary data value could simply be transmitted following the *ESCAPE* code, however, Altera's protocol uses a data value that is the logical exclusive or (XOR) of the data value with `0x20` (this data masking technique is also used in RFC1662 [10]). For example, a JTAG-to-Avalon-ST transaction containing the *IDLE* code as data will encode the byte as `0x4D 0x6A`, while a transaction containing the *ESCAPE* code as data will encode the byte as `0x4D 0x6D`.

The data headers, and *IDLE* and *ESCAPE* codes appear in the byte-streams on the *JTAG side* of the JTAG-to-Avalon-ST bridge. The byte-streams on the *Avalon-ST side* of the JTAG-to-Avalon-ST bridge contain *only* the data bytes. Section 4 shows simulation and logic analyzer traces of the byte-streams on the JTAG-to-Avalon-ST interfaces.

Table 1: JTAG-to-Avalon-ST JTAG 3-bit Virtual Instruction Register (VIR) modes.

Code	Mode	Description
`0`	*DATA*	Data byte-stream to and from user logic.
`1`	*LOOPBACK*	Loopback the JTAG serial data (via a 1-bit bypass register).
`2`	*DEBUG*	3-bit (read-only) Debug status register: [0]: Avalon-ST reset state [1]: Avalon-ST clock divided-by-2 state [2]: Avalon-ST clock state
`3`	*INFO*	11-bit (read-only) Component identification (Verilog parameter values). [3:0]: floor($\log_2$(`UPSTREAM_FIFO_SIZE`)) [7:4]: floor($\log_2$(`DOWNSTREAM_FIFO_SIZE`)) [10:8]: `PURPOSE` (0 = JTAG Avalon-ST, 1 = Avalon-MM)
`4`	*CONTROL*	9-bit (read/write) Offset and reset-request control. [7:0]: Offset [9]: Reset request

Table 2: JTAG-to-Avalon-ST DATA mode 16-bit headers.

Data direction	Bits	Description
To device	[9:0]	Scan length.
	[12:10]	Read data length.
	[15:13]	Write data length.
From device	[0]	Read data is available.

Table 3: JTAG-to-Avalon-ST DATA mode protocol codes.

Code	Name	Description
`0x4A`	*IDLE*	Inserted into the data stream when there is no data to send.
`0x4D`	*ESCAPE*	Inserted into the data stream when the data to send is a protocol code, followed by the data XORed with `0x20`.

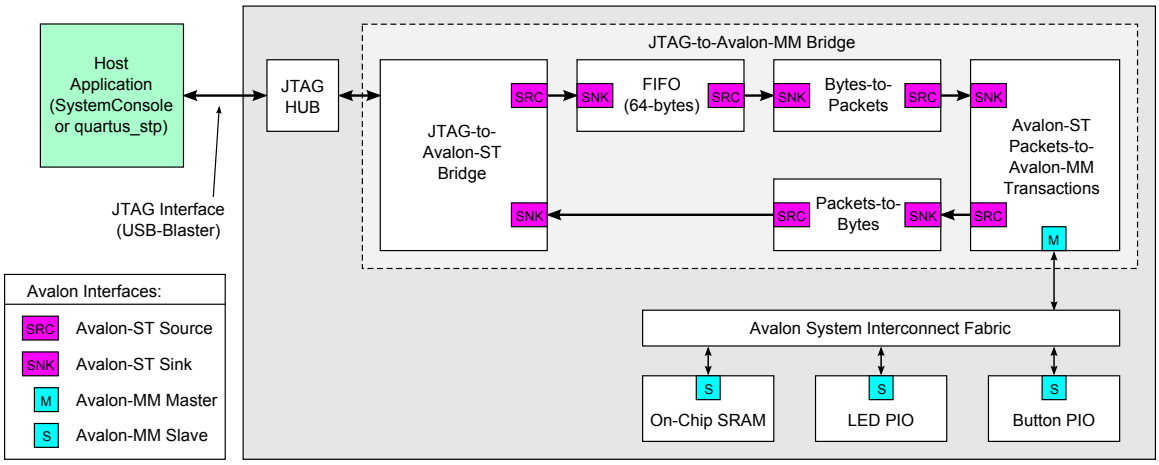

Figure 3: JTAG-to-Avalon-MM Bridge.

2.2 JTAG-to-Avalon-MM

The JTAG-to-Avalon-MM bridge provides an interface for generating Avalon-MM master read or write transactions via JTAG. The bridge is implemented via a series of interconnected components and protocols. Figure 3 shows the components within the JTAG-to-Avalon-MM bridge, and how the bridge connects to an Avalon-MM system. The host generates an Avalon-MM master transaction by first encoding a Avalon-MM command into a command byte-stream, and then issuing the command to the JTAG-to-Avalon-MM bridge via JTAG. The *JTAG-to-Avalon-ST bridge* transports the encoded Avalon-ST command byte-stream (crossing from the JTAG clock domain to the Avalon-ST clock domain), the *bytes-to-packets component* converts the command byte-stream into Avalon-ST command packet (which is a byte-stream with extra Avalon-ST control signals), the *packets-to-transactions* component converts the Avalon-ST packet into an Avalon-MM transaction, issues an Avalon-MM master transaction, and encodes the response as an Avalon-ST response packet, the *packets-to-bytes* component converts the Avalon-ST response packet to an encoded Avalon-ST response byte-stream, which is then read by the host via the *JTAG-to-Avalon-ST bridge* (after the byte-stream has crossed from the Avalon-ST clock domain to the JTAG clock domain). The host decodes the response bytes to determine the completion of an Avalon-MM write, or to access the data from an Avalon-MM read.

The Avalon-ST packets-to-transactions packet format for read and write transactions is shown in Tables 4 and Table 5. The transaction codes used in those packets are shown in Table 6. The transaction packets are encoded to and from byte-streams via the bytes-to-packets/packets-to-bytes protocol codes in Table 7. The bytes-to-packets/packets-to-bytes protocol defines a channel number, start and end of packets, and an escape code. These codes are consumed by the bytes-to-packets core and produced by the packets-to-bytes core. These cores convert between Avalon-ST *byte* streams and Avalon-ST *packet* streams. The packets-to-transactions core converts the Avalon-ST packet stream into an Avalon-MM transaction, and generates a response Avalon-ST packet stream, which is subsequently encoded as an Avalon-ST byte-stream.

The JTAG-to-Avalon-MM bridge is typically used for system debug and initial board bring-up. The bridge can be used to generate Avalon-MM single-read and single-write transactions. The bridge can not be used to generate Avalon-MM bursts or to generate back-to-back transactions, however, the bridge could be used to program another Avalon-MM master capable of generating bursts and back-to-back transactions.

Table 4: JTAG-to-Avalon-MM Packets-to-Transaction Read Packet.

Byte	Field	Description
Command		
`0`	Transaction code	See Table 6
`1`	Reserved	Reserved for future use (use `0x00`).
`[3:2]`	Size	Transaction size in bytes (16-bit big-endian format).
`[7:4]`	Address	Avalon-MM address (32-bit big-endian format).
Response		
`[3:0]`	Data	Avalon-MM data (32-bit little-endian format).

Table 5: JTAG-to-Avalon-MM Packets-to-Transaction Write Packet.

Byte	Field	Description
Command		
`0`	Transaction code	See Table 6
`1`	Reserved	Reserved for future use (use `0x00`).
`[3:2]`	Size	Transaction size in bytes (16-bit big-endian format).
`[7:4]`	Address	Avalon-MM address (32-bit big-endian format).
`[n:8]`	Data	Avalon-MM data (32-bit little-endian format).
Response		
`0`	Transaction code	Transaction code with the MSB inverted.
`1`	Reserved	Reserved for future use (use `0x00`).
`[3:2]`	Size	Total number of bytes written (16-bit big-endian format).

Table 6: JTAG-to-Avalon-MM Packets-to-Transaction Transaction Codes.

Code	Avalon-MM Transaction	Description
Transaction		
`0x00`	Write, non-incrementing	Write transaction(s) to a fixed address.
`0x04`	Write, incrementing	Write transaction(s) starting at the given address.
`0x10`	Read, non-incrementing	Read transaction(s) to a fixed address.
`0x14`	Read, incrementing	Read transaction(s) starting at the given address.
`0x7F`	No transaction	No transaction is generated on the Avalon-MM interface. Used for testing the packets interface.

Table 7: JTAG-to-Avalon-MM Bytes-to-Packets/Packets-to-Bytes Protocol Codes.

Code	Name	Description
`0x7A`	*SOP*	Start of packet.
`0x7B`	*EOP*	End of packet.
`0x7C`	*CHANNEL*	Channel number.
`0x7D`	*ESCAPE*	Inserted into the data stream when the data to send is a protocol code, followed by the data XORed with `0x20`.

2.3 Altera Documentation

The JTAG-to-Avalon-ST, bytes-to-packets, packets-to-bytes, and JTAG-to-Avalon-MM components are (partially) documented in the Embedded IP Users Guide [7];

- *Chapter 11: Avalon-ST Serial Peripheral Interface Core* has the documentation for an SPI controller. This is not used in the JTAG-to-Avalon-MM design, but it is the only place in the IP guide that the special character codes for the JTAG interface are discussed. Page 11-2 has the special codes, i.e., `0x4A` (*IDLE*), and `0x4D` (*ESCAPE*), and that the escape mask (XOR) code is `0x20`.

- *Chapter 18: SPI Slave/JTAG to Avalon Master Bridge Cores* describes the JTAG-to-Avalon-MM master component. Figure 18-3 contains an example of a transaction byte stream, however, the channel code and start-of-packet are reversed from what is observed in hardware (via SignalTap II traces). There is no description of the figure in this chapter (hence the origin of the `0x4D` escape code is not obvious).

- *Chapter 20: Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores* describes the byte stream to transaction command packets components. Page 28-2 describes the special codes used by the packets cores, i.e., `0x7A` (start-of-packet, *SOP*), `0x7B` (end-of-packet, *EOP*), `0x7C` (*CHANNEL*), `0x7D` (*ESCAPE*), and that the escape mask (XOR) code is `0x20`. Figure 20-3 has an example of a transaction byte stream, and this time the channel code and start-of-packet are in the order observed in hardware.

- *Chapter 21: Avalon Packets to Transactions Converter Core* describes the logic that generates an Avalon-MM transaction based on a command packet, and then sends a response packet back via the byte stream. Table 21-2 and 21-3 have the details required to form transaction packets for Avalon-MM read and write operations. Table 21-2 has the response packet format for write, but there is no documentation for the read response packet (it was determined using SignalTap II and the simulation testbench).

- *Chapter 32: Avalon-ST JTAG Interface Core* describes the JTAG to byte stream component. There are no comments regarding the protocol codes used for idle and escape, or the escape XOR mask. This information used to be in this chapter; in the July 2010 version of the guide, this component was described in Chapter 31, and there was a table with the special character codes, and a description of how the codes are used, along with the XOR mask for escaped data.

3 Software

Altera provides two software interfaces for accessing JTAG devices; SystemConsole and `quartus_stp`. Altera provides procedures for accessing the JTAG-to-Avalon components using SystemConsole, but does not provide procedures for use with `quartus_stp`. The code associated with this document contains a Tcl package that provides `quartus_stp` support procedures for accessing JTAG-to-Avalon components.

3.1 SystemConsole

The *SystemConsole* services and procedures for accessing JTAG components are documented in the *Quartus II Handbook, Volume 3: Verification, Chapter 10: Analyzing and Debugging Designs with the System Console* [8]. The services of interest are;

- `sld`: provides low-level access to the Virtual JTAG instruction and data registers.
- `jtag_debug`: provides procedures to access to the status and control registers in the JTAG-to-Avalon-ST bridge.
- `bytestream`: provides procedures to send and receive byte-streams via the data mode of the JTAG-to-Avalon-ST bridge.
- `master`: provides procedures to generate Avalon-MM read and write transactions using the JTAG-to-Avalon-MM bridge.

Using the protocols provided in Section 2, the `master` service procedures can be re-implemented using the `bytestream` service procedures, and all service procedures can be implemented using the `sld` service procedures.

3.2 Command-line tools (`quartus_stp`)

The `quartus_stp` procedures for accessing JTAG components are documented in the *Virtual JTAG Megafunction User Guide* [4]. The procedures of interest are;

- `device_ir_shift` and `device_dr_shift`: provide low-level access to the JTAG instruction and data registers.
- `device_virtual_ir_shift` and `device_virtual_dr_shift`: provide low-level access to the Virtual JTAG instruction and data registers.

The Virtual JTAG functions in `quartus_stp` are essentially identical to those implemented by the `sld` service (with the procedure names changed). The `quartus_stp` Virtual JTAG procedures can be implemented using the low-level JTAG procedures. Understanding the implementation of the JTAG-to-Avalon-MM protocol via the low-level JTAG commands allows the development of custom software to access Avalon-MM components via JTAG.

4 Hardware Tests

The source code to the JTAG-to-Avalon-ST and JTAG-to-Avalon-MM components is provided as part of the Quartus II software installation as shown in Figure 4. Host software accesses the JTAG-to-Avalon-ST byte-streams and generates JTAG-to-Avalon-MM transactions via Tcl procedures provided by the *SystemConsole* user interface. The source code for the SystemConsole procedures is not provided in the Quartus II software installation.

The hardware tests in this section determine how SystemConsole utilizes the JTAG-to-Avalon protocols. Each test instantiates a JTAG-to-Avalon component and a SignalTap II logic analyzer instance, issues a SystemConsole command to the hardware, and captures a logic analyzer trace of the JTAG transaction. The logic analyzer traces are compared to the protocols in Section 2 to determine the *software implementation* of the protocols. The JTAG transactions are then reproduced in simulation. The result of this analysis is that the JTAG-to-Avalon-MM bridge can be used in simulation to generate Avalon-MM transactions (Altera do not officially provide such support).

The JTAG-to-Avalon-ST and JTAG-to-Avalon-MM components are typically instantiated in Qsys [2] or in SOPC Builder [3]. The hardware designs in this section instantiate the JTAG components directly. Refer to the simulation and synthesis scripts for details.

The hardware tests were performed using an Arrow BeMicro-SDK USB stick. The USB stick contains a USB-Blaster interface and a Cyclone IV FPGA. The hardware tests make minimal use of the board I/O, using only the clock, reset, 8 LEDs, and 2 DIP switches. The hardware tests can be ported to other development boards.

- `QUARTUS_ROOTDIR`

  Environment variable created by the Quartus II installer.

- `$QUARTUS_ROOTDIR/eda/sim_lib/altera_mf.v`

  Altera Megafunction Verilog file containing the `sld_virtual_jtag` source.

- `$QUARTUS_ROOTDIR/eda/sim_lib/altera_mf.vhd`

  Altera Megafunction VHDL file containing the `sld_virtual_jtag` source.

- `$QUARTUS_ROOTDIR/eda/sim_lib/altera_mf_components.vhd`

  Altera Megafunction VHDL file containing the `sld_virtual_jtag` and `sld_virtual_jtag_basic` component definitions.

- `QUARTUS_SOPC_IP = $QUARTUS_ROOTDIR/../ip/altera/sopc_builder_ip`

  Quartus SOPC Builder IP directory.

- `$QUARTUS_SOPC_IP/altera_avalon_jtag_phy/altera_jtag_sld_node.v`

  The JTAG node.

  For *synthesis* this component instantiates an `sld_virtual_jtag_basic` component, which is an `sld_virtual_jtag` component with additional parameters to set the manufacturer ID (110 = 0x6E), type ID (132 = 0x84), and version (1). There is no source for the Verilog or VHDL component, just a VHDL component definition.

  For *simulation* this component provide a series of Verilog tasks that are used to generate JTAG transactions.

- `$QUARTUS_SOPC_IP/altera_avalon_jtag_phy/altera_jtag_streaming.v`

  JTAG-to-Avalon-ST protocol component, including the 3-bit JTAG instruction register (IR) values (see Table **??**), and byte stream header and data format details (not exactly easy reading, but enough detail to write simulation stimulus to aid in understanding).

- `$QUARTUS_SOPC_IP/altera_avalon_jtag_phy/altera_avalon_st_jtag_interface.v`

  JTAG-to-Avalon-ST component; instantiates the JTAG node and streaming components.

- `$QUARTUS_SOPC_IP/altera_avalon_st_bytes_to_packets/`

  `$QUARTUS_SOPC_IP/altera_avalon_st_packets_to_bytes/`

  Avalon-ST bytes-to-packets and packets-to-bytes components which implement the byte encoding and decoding protocol in Table 7.

- `$QUARTUS_SOPC_IP/altera_avalon_packets_to_master/altera_avalon_packets_to_master.v`

  Avalon-ST packets to Avalon-MM master transactions component, which implements the transaction codes in Table 6, the write packet protocol in Table 5, and the read packet protocol in Table 4.

- `$QUARTUS_SOPC_IP/altera_jtag_avalon_master/altera_jtag_avalon_master.v`

  JTAG-to-Avalon-MM component; instantiates the JTAG node, streaming, bytes-to-packets, packets-to-bytes, and packets-to-transactions components.

Figure 4: Quartus II JTAG-to-Avalon source.

Table 8: JTAG node (`jtag_node`) project source.

Folder	File	Description
`scripts`	`synth.tc`	Quartus II synthesis script
	`bemicro_sdk.stp`	SignalTap II setup
	`bemicro_sdk.sdc`	Timing constraints
	`sim.tc`	Modelsim simulation script
	`jtag_node_tb.do`	Modelsim wave window setup
	`jtag_cmds_sc.tcl`	SystemConsole commands
`src`	`bemicro_sdk.sv`	Top-level synthesis source
`test`	`jtag_node_tb.sv`	Simulation testbench

Table 9: JTAG node (`jtag_node`) project Virtual Instruction Register (VIR) decode.

VIR[2:0]	Access	Description
0	Write-only	Write to the 8-bit data register
1	Read-only	Read from the 8-bit data register
2	Read-only	Read the state of the 2 DIP switches
3-7	Read-only	Unused. Reads return zero

4.1 JTAG Node

The JTAG node source, `altera_jtag_sld_node.v` (see Figure 4 for the source location), provides the basic JTAG-to-FPGA interface logic. The component selects between the Virtual JTAG component for *synthesis* and a set of Verilog tasks for *simulation*. The hardware tests on the JTAG node determine the JTAG transaction sequences. These sequences are reproduced (as closely as possible) via the JTAG node simulation tasks. The JTAG node simulation model is then used to develop the JTAG-to-Avalon-ST and JTAG-to-Avalon-MM simulation models.

The JTAG node project source layout is shown in Table 8. Appendix A contains build instructions. The top-level synthesis source contains the JTAG node, an 8-bit JTAG data shift-register, and the registers shown in Table 9. Write and read access to the 8-bit data register uses two different VIR codes, due to the fact that JTAG transactions are *simultaneously* both write (host-to-device data) and read (device-to-host data) transactions; when VIR = 0, the device transmits zero over JTAG (and the host ignores it), and when VIR = 1, the host transmits zero over JTAG (and the device ignores it). This simple design is representative of how all Virtual JTAG designs use the Virtual Instruction Register for decoding different Virtual Data Register data paths.

The JTAG node design uses the LEDs for user feedback as follows;

LED[3:0]	Connects to the 4-LSBs of the 8-bit data register
LED[6:4]	The 3-bit VIR value
LED[7]	Blinks at about 1Hz to show the design is running

The Tcl procedure `jtag_sld_vir_count` generates an incrementing count on the 3 VIR LEDs (and reads the 2-bit switch state via VIR out), while the procedure `jtag_sld_data_count` generates a count on the 4 data LEDs (and reads the 8-bit register back and reads the 2-bit switch state). The value written to the LED and the values read back are printed to the console. The LED count increments at about once per second, so the DIP switches can be changed and the value printed to the console seen to change.

Figures 5 and 6 show SignalTap II logic analyzer traces from the two basic Virtual JTAG transactions; a Virtual shift-IR sequence and a Virtual shift-DR sequence. The figures show the JTAG one-hot states as well as the Virtual JTAG one-hot states. The Virtual sequences are constructed from a JTAG shift-IR sequence to load the `USER1` or `USER0` JTAG instruction codes, and then a JTAG shift-DR sequence to access the Virtual IR or Virtual DR path. The Virtual IR sequence in Figure 5 was generated by the Tcl command `jtag_sld_data_read`, which sets IR = 1 (per Table 9), while the Virtual DR sequence in Figure 6 was generated by the Tcl command `jtag_sld_data_write 0x55`, which sets IR = 0 (per Table 9) prior to the trace capture, and then serially shifts in the 8-bit data value `0x55` (overwriting the previous register value of `0xAA`). For more detail on the Virtual JTAG interface see [9] (that document explains the encoding of the 10-bit VIR code in Figure 5).

The JTAG node source, `altera_jtag_sld_node.v` provides a minimal set of Verilog tasks for simulation. The project testbench, `jtag_node_tb.sv`, contains test sequences showing how to use the JTAG node Verilog tasks to update the Virtual Instruction Register to generate the VIR codes shown in Table 1, and how to generate Virtual shift-DR sequences for a single-byte and for multiple-bytes. The simulation *does not* generate waveforms identical to the hardware tests (due to the limitations of the Verilog tasks), but the functionality is sufficient to simulate designs containing the JTAG node.

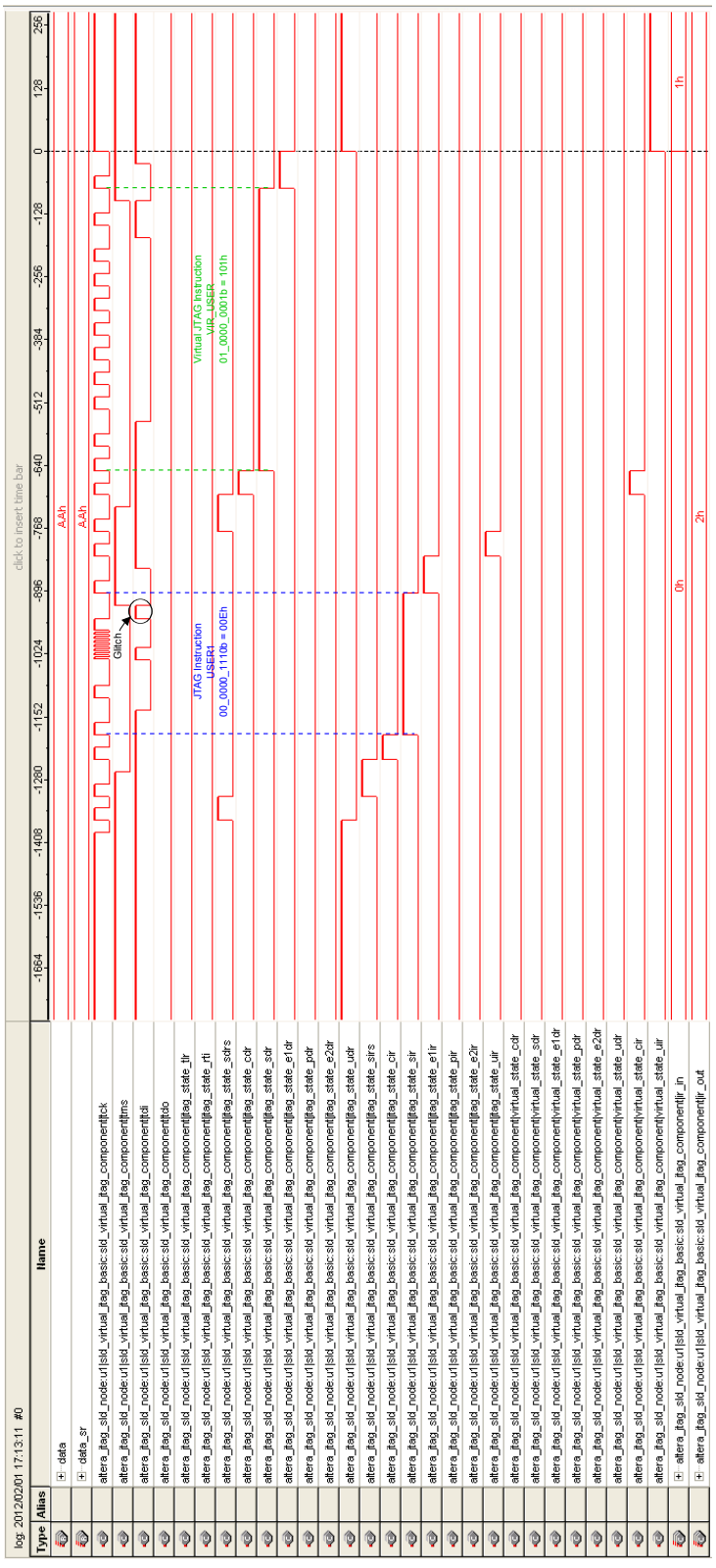

Figure 5: JTAG node Virtual Instruction Register SignalTap II logic analyzer trace. The sequence consists of the JTAG `USER1` IR-shift sequence followed by a JTAG DR-shift sequence. The JTAG shift-DR sequence corresponds to the Virtual JTAG shift-IR sequence. In this example, the sequence sets the Virtual Instruction to `ir_in[2:0] = 1`.

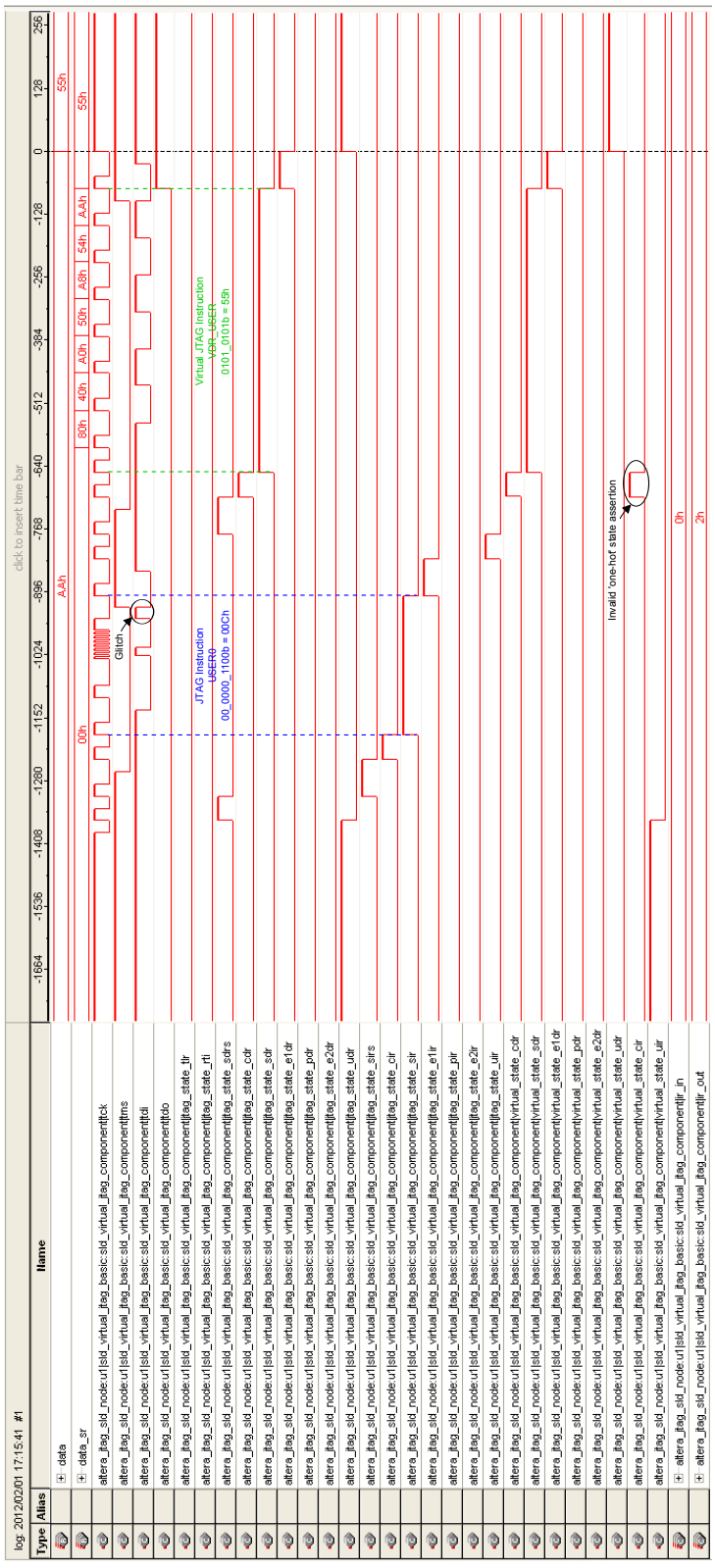

Figure 6: JTAG node Virtual Data Register SignalTap II logic analyzer trace. The sequence consists of the JTAG `USER0` IR-shift sequence followed by a JTAG DR-shift sequence. The JTAG shift-DR sequence corresponds to the Virtual JTAG shift-DR sequence. In this example, the sequence sets the 8-bit data register to `data[7:0] = 0x55`.

Table 10: JTAG-to-Avalon-ST (`jtag_to_avalon_st`) project source.

Folder	File	Description
`scripts`	`synth.tc`	Quartus II synthesis script
	`bemicro_sdk.stp`	SignalTap II setup
	`bemicro_sdk.sdc`	Timing constraints
	`sim.tc`	Modelsim simulation script
	`jtag_to_avalon_st_tb.do`	Modelsim wave window setup
	`jtag_cmds_sc.tcl`	SystemConsole commands
`src`	`bemicro_sdk.sv`	Top-level synthesis source
`test`	`jtag_to_avalon_st_tb.sv`	Simulation testbench

4.2 JTAG-to-Avalon-ST

The JTAG-to-Avalon-ST source, `altera_avalon_st_jtag_interface.v` (see Figure 4 for the source location), provides an interface for the host to stream bytes into and out of the FPGA fabric, and to access control and status registers. The JTAG-to-Avalon-ST protocol is described in Section 2.1, and the SystemConsole procedures that implement the protocol are referenced in Section 3.1.

The JTAG-to-Avalon-ST project source layout is shown in Table 10. Appendix A contains build instructions. The top-level synthesis source contains the JTAG-to-Avalon-ST bridge, an 8-bit data register (which is loaded each time valid data is received on the host-to-device Avalon-ST byte-stream), a write data counter (to track writes to the data register), and a synthesis parameter to control whether the Avalon-ST interface is looped-back (allowing the host to read-back the data it writes) or whether host-to-device data is just dropped (and no device-to-host data is ever generated). The JTAG-to-Avalon-ST design uses the LEDs for user feedback as follows;

LED[5:0]	Connects to the 6-LSBs of the 8-bit data register
LED[6]	The Avalon-ST `resetrequest` output
LED[7]	Blinks at about 1Hz to show the design is running

The SystemConsole Tcl procedures in `jtag_cmds_sc.tcl` allow the data register (and LEDs) to be written and read, allow the transmission of strings and blocks of binary data over the byte-stream interface, control of the `resetrequest` output, and implementation of some of the `bytestream` service commands via the low-level `sld` service commands. See the Tcl script for details.

The JTAG-to-Avalon-ST source file, `altera_jtag_streaming.v`, requires the following modifications to enable SignalTap II probing (to disable logic elimination and net renaming);

```
wire write_data_byte_aligned          /* synthesis keep */;
reg [15:0] header_in  = 'b0           /* synthesis noprune */;
wire [7:0] idle_inserter_source_data  /* synthesis keep */;
wire [7:0] idle_inserter_source_data;
wire data_available                   /* synthesis keep */;
```

Figure 7 shows a SignalTap II logic analyzer trace of the transmission of the string "hello". The string was transmitted by the `jtag_bytestream_string` procedure in the script `jtag_cmds_sc.tcl`. The procedure is implemented using the SystemConsole `bytestream` service, `bytestream_send`

command. The trace, and others like it, show how the `bytestream` service utilizes the JTAG-to-Avalon-ST protocol.

Figure 8 shows a SignalTap II logic analyzer trace of the transmission of the string "HIJKLMN". This string is special in that it contains the *IDLE* (`0x4A`) and *ESCAPE* (`0x4D`) codes as data. The logic analyzer trace shows how the data is encoded over the JTAG byte-stream and that the data on the Avalon-ST interface matches the bytes in the transmitted string.

Figures 9, 10, 11, and 12, show a 1kB transfer of binary data over the JTAG-to-Avalon-ST interface. The binary data consists of a start-of-packet (SOP) code (`0x11`), alternating data bytes `0x55` and `0xAA`, and an end-of-packet (EOP) code (`0x22`). The format of the binary data was chosen to provide unique SOP and EOP codes to trigger SignalTap II, and avoided using any of the JTAG-to-Avalon-ST protocol codes (a data payload of an incrementing count would have to avoid the protocol codes). The figures show a subtle implementation detail of the SystemConsole `bytestream` service, `bytestream_send` procedure; for each 1024-byte transfer only 1022-bytes of data are transmitted, with the final two bytes replaced with the *IDLE* code (see Figure 10). The final two bytes of data are then transmitted in a subsequent data packet (see Figure 11). Figure 12 shows that the 1kB of data could have been transferred in a single 1kB transaction.

The JTAG-to-Avalon-ST SystemConsole `bytestream` procedures were tested using the binary data for various lengths, yielding the following observations;

- The JTAG-to-Avalon-ST header is always `0xFC03` which indicates that the scan length, write length, and read length are all 1024-bytes (0x400).
- The 1024-byte length header is used for all transfers, including transfers of greater than 1024-bytes; the SystemConsole procedure divides the transfer into multiple 1024-byte transactions.

The JTAG-to-Avalon-ST protocol was implemented using the SystemConsole `sld` service for an arbitrary length binary data stream. The JTAG-to-Avalon-ST header was calculated as the next 256-byte increment over the binary data length, with *IDLE* codes used as padding. Signal Tap II traces of various lengths confirmed the operation of the procedures.

The `sld` based procedures also provided a mechanism to exercise the data available field of the JTAG-to-Avalon-ST read data header in Table 2. The data available signal, `data_available`, can be seen asserted at the end of the transaction in Figure 12. The 16-bit read data header following this transaction had the read data available bit set.

The project simulation testbench, `jtag_to_avalon_st_tb.sv`, uses the Verilog tasks in the JTAG SLD node to access the JTAG-to-Avalon-ST bridge control and status registers, and to generate data byte-stream transactions. The testbench contains stimulus sequences that test different JTAG-to-Avalon-ST headers, eg., read-only and write-only operations (modes that are not exercised by any of the Altera procedures), and stimulus that investigates what happens to the bridge interface logic when the JTAG interface does not send the amount of data indicated by the header, i.e., short transactions.

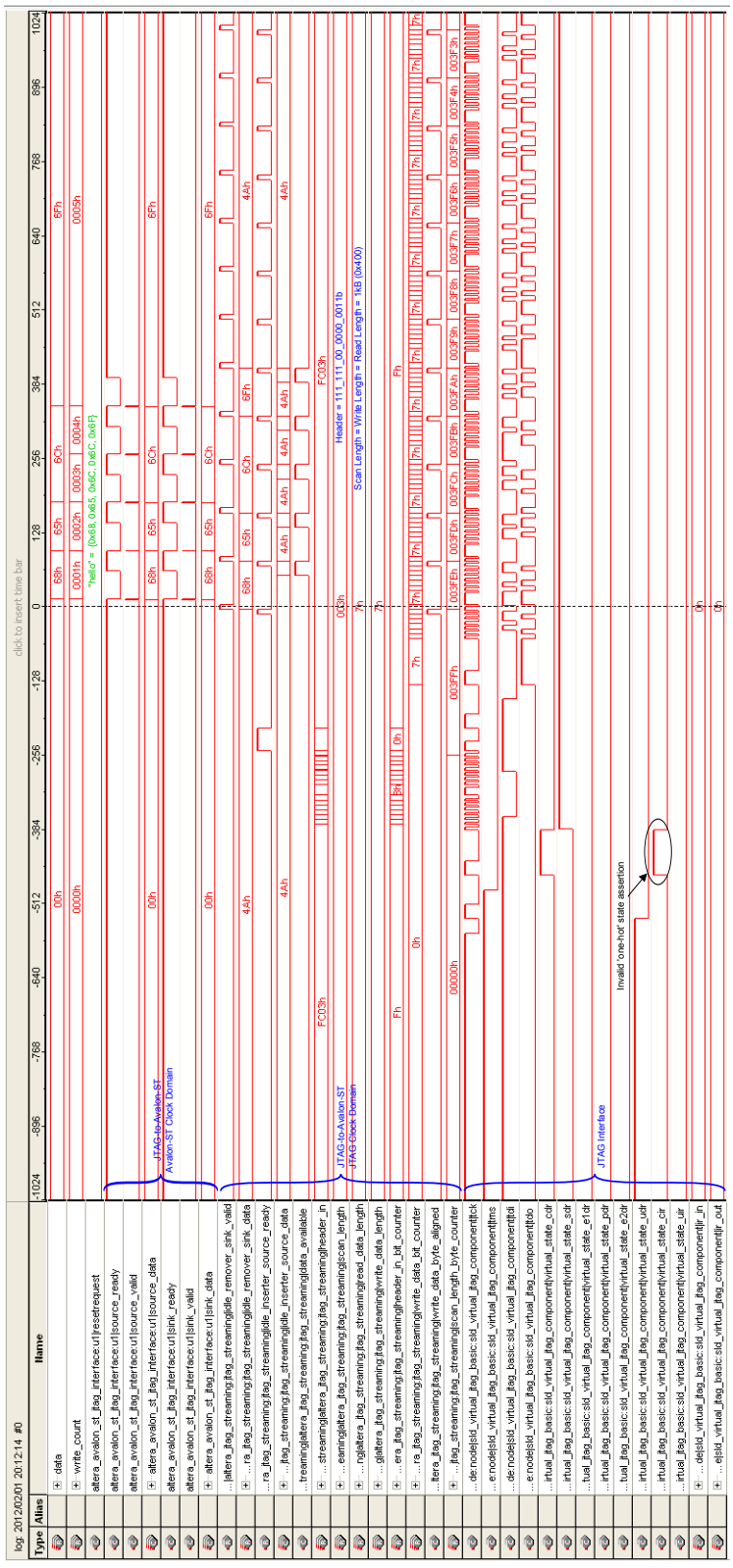

Figure 7: JTAG-to-Avalon-ST loopback of the string "hello". The 16-bit header indicates that the SystemConsole `bytestream_send` command generates a 1kB packet. The packet data consists of the characters in the string "hello", followed by the *IDLE* code (`0x4A`).

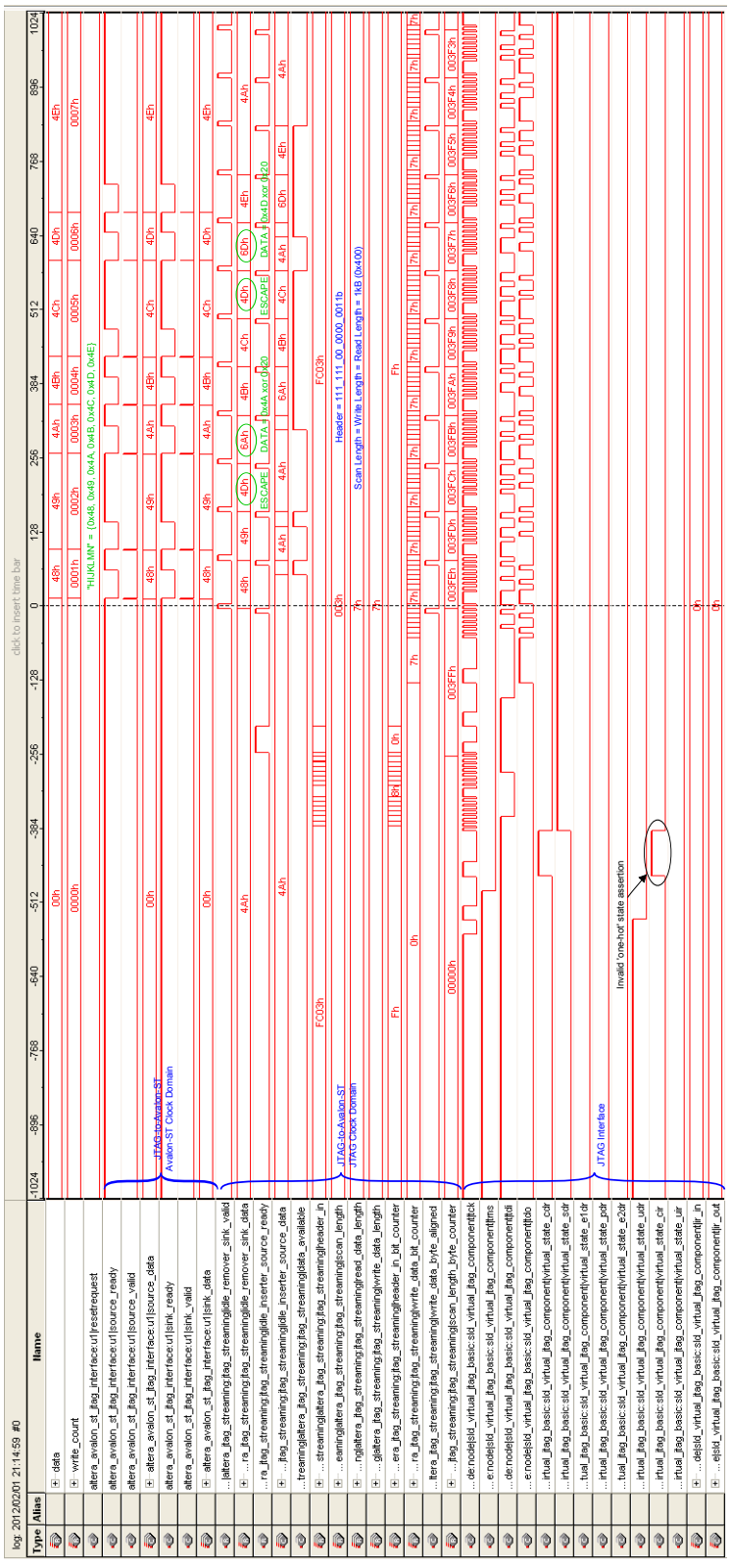

Figure 8: JTAG-to-Avalon-ST loopback of the string "HIJKLMN". This string is special in that it contains the *IDLE* and *ESCAPE* character codes as *data*. The logic analyzer trace shows how the data is encoded in the JTAG byte-stream as an *ESCAPE* code followed by the byte XORed with 0x20.

Figure 9: JTAG-to-Avalon-ST transfer of a 1kB of binary data; logic analyzer trace at the start-of-packet.

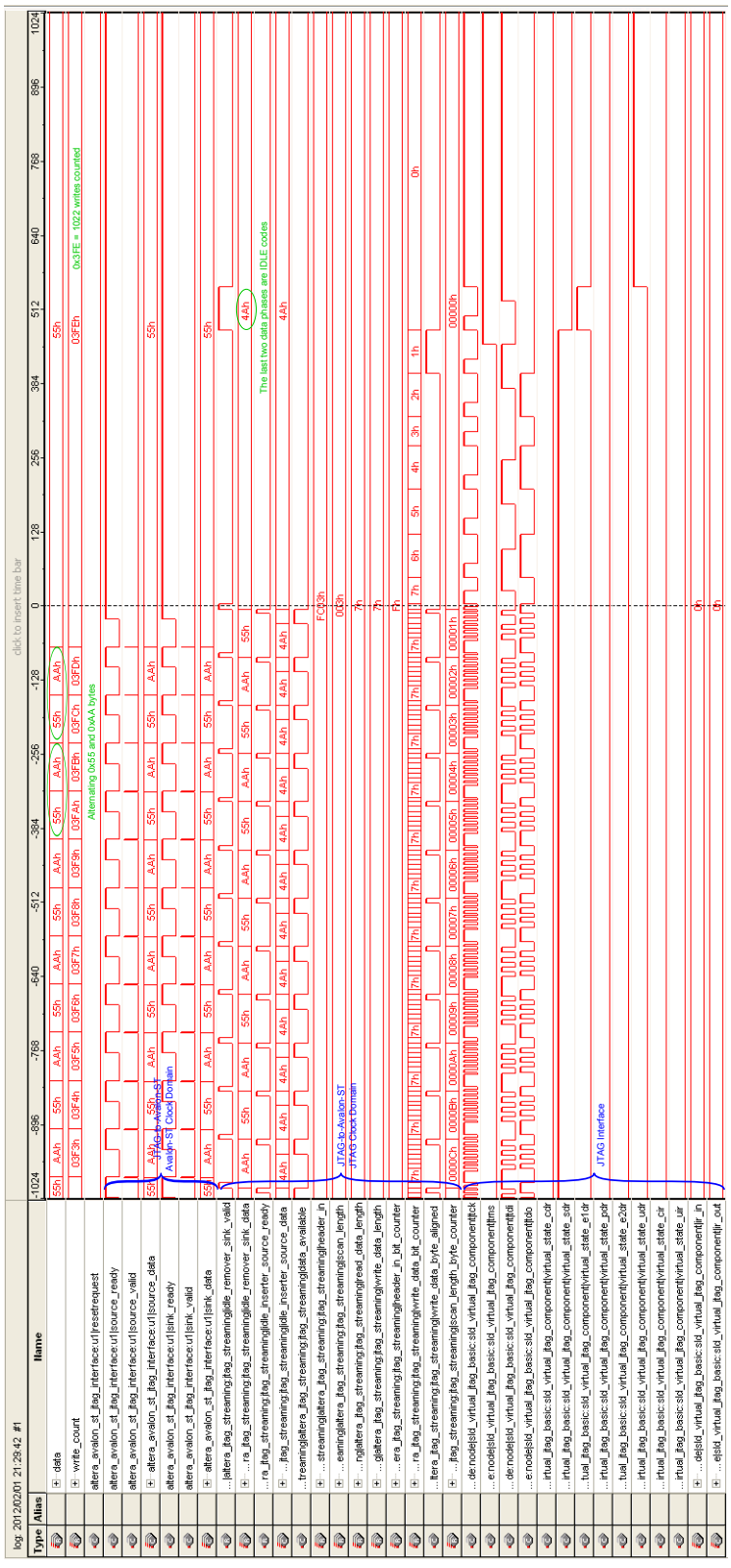

Figure 10: JTAG-to-Avalon-ST transfer of a 1kB of binary data; logic analyzer trace at the end of the 1kB scan length. The SystemConsole `bytestream_send` procedure inserts two *IDLE* codes at the end of the packet, and sends the last two bytes in another transfer.

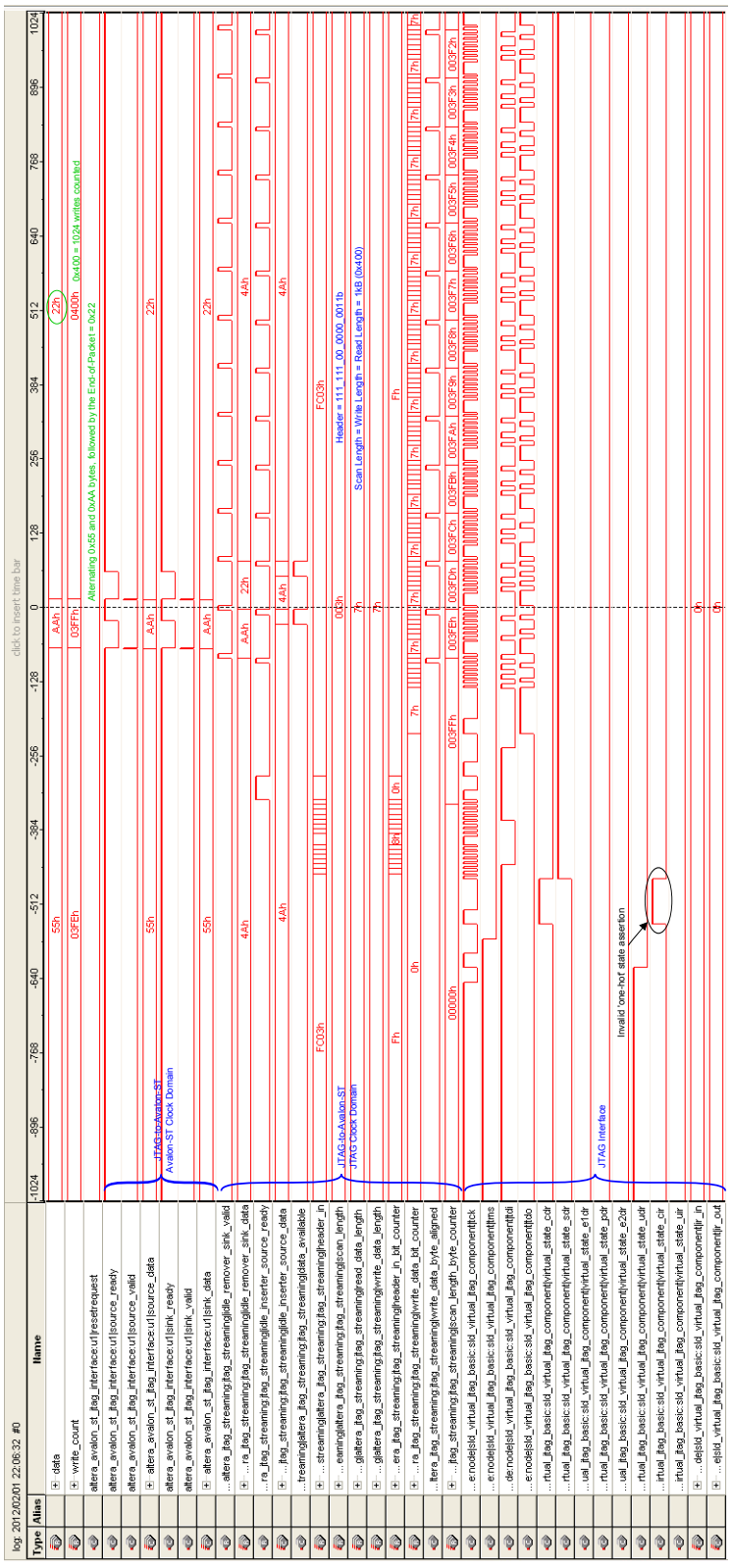

Figure 11: JTAG-to-Avalon-ST transfer of a 1kB of binary data; logic analyzer trace at the end-of-packet. The SystemConsole `bytestream_send` procedure inserts two *IDLE* codes at the end of the packet, and sends the last two bytes in another transfer.

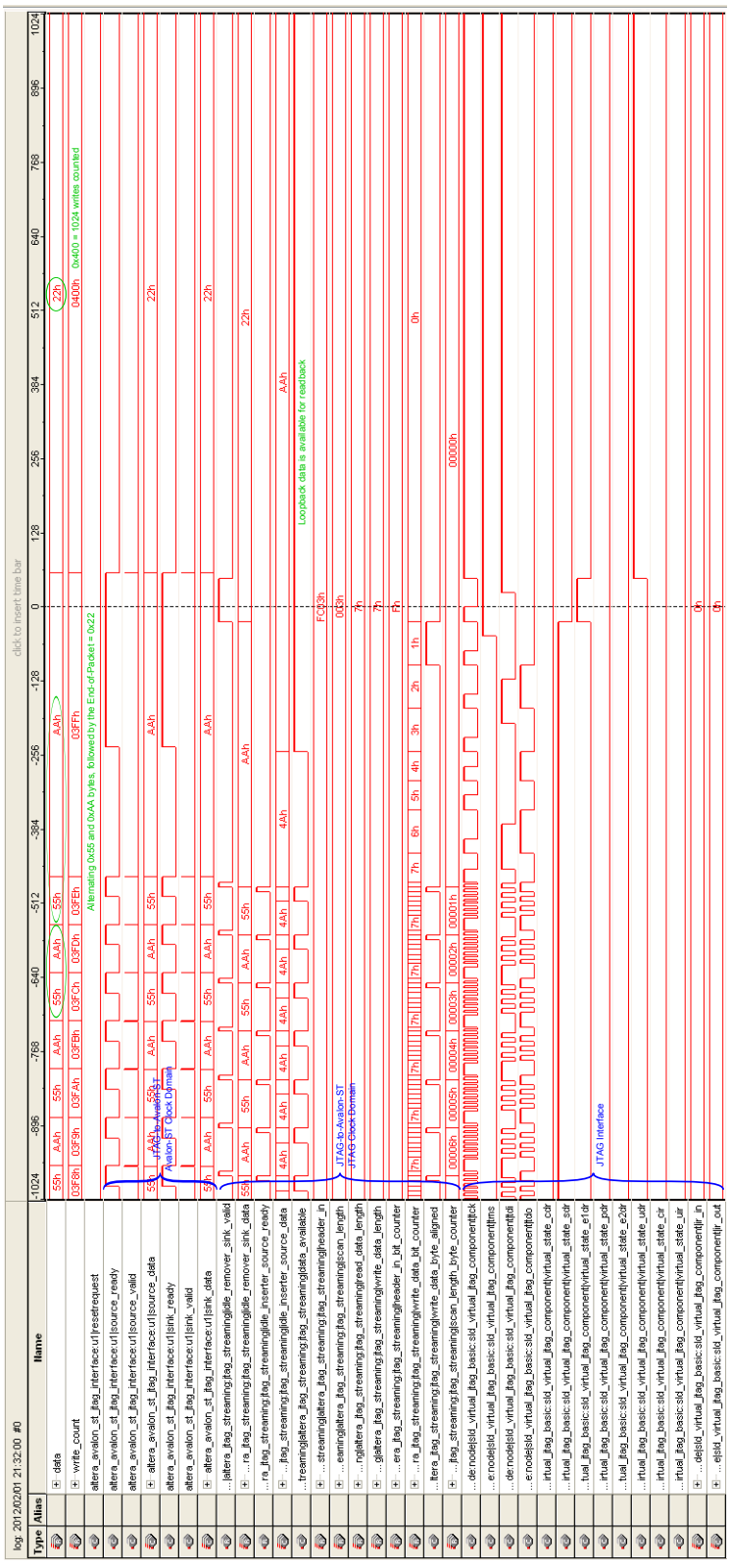

Figure 12: JTAG-to-Avalon-ST transfer of a 1kB of binary data; logic analyzer trace at the end-of-packet. The SystemConsole `bytestream_send` procedure was implemented using the low-level `sld` service procedures to show that the full 1kB data transfer can be performed using a single 1kB transfer. Note that the `data_available` signal remains asserted at the end of the transaction, as Avalon-ST loopback data is ready to be read by the host.

Table 11: JTAG-to-Avalon-MM (`jtag_to_avalon_mm`) project source.

Folder	File	Description
`scripts`	`synth.tc`	Quartus II synthesis script
	`bemicro_sdk.stp`	SignalTap II setup
	`bemicro_sdk.sdc`	Timing constraints
	`sim.tc`	Modelsim simulation script
	`jtag_to_avalon_mm_tb.do`	Modelsim wave window setup
	`jtag_cmds_sc.tcl`	SystemConsole commands
`src`	`bemicro_sdk.sv`	Top-level synthesis source
`test`	`jtag_to_avalon_mm_tb.sv`	Simulation testbench

4.3 JTAG-to-Avalon-MM

The JTAG-to-Avalon-MM source, `altera_jtag_avalon_master.v` (see Figure 4 for the source location), provides an interface for the host to generate Avalon-MM master transactions (and to access control and status registers of the underlying JTAG-to-Avalon-ST component). The JTAG-to-Avalon-MM protocol is described in Section 2.2, and the SystemConsole procedures that implement the protocol are referenced in Section 3.1.

The JTAG-to-Avalon-MM project source layout is shown in Table 11. Appendix A contains build instructions. The top-level synthesis source contains the JTAG-to-Avalon-MM bridge, and a 32-bit data register located at 32-bit aligned address `0x11223344`. The register address makes it easier to see the big-endian ordering of the address bytes in the JTAG-to-Avalon-ST byte-streams. The JTAG-to-Avalon-MM design uses the LEDs for user feedback as follows;

LED[5:0]	Connects to the 6-LSBs of the 32-bit data register
LED[6]	The Avalon-ST `resetrequest` output
LED[7]	Blinks at about 1Hz to show the design is running

The SystemConsole Tcl procedures in `jtag_cmds_sc.tcl` allow the data register (and LEDs) to be written and read, control of the `resetrequest` output, and implementation of some of the `master` service commands via the `bytestream` service commands. See the Tcl script for details.

The SystemConsole `master` service provides the Tcl procedures shown in Table 12 (Quartus II Handbook, Volume 3, Chapter 10 [8]). The following tests were performed to investigate the JTAG transactions generated by the `master` service procedures;

- 32-bit write access via `master_write_32`.

  Figures 13 and 14 show a SignalTap II logic analyzer trace of a 32-bit write transaction for (address, data) = (`0x11223344`, `0x55667788`). The figures are annotated to show the encoded transaction bytes.

- 32-bit read access via `master_read_32`.

  Figures 15 and 16 show a logic analyzer trace of a 32-bit read transaction for (address, data) = (`0x11223344`, `0x55667788`). The figures are annotated to show the encoded transaction bytes.

- 32-bit write/read access via `master_write/read_memory`.

Figures 17 and 18 show a logic analyzer trace of multiple 32-bit write and read transactions. The figures are annotated to show the encoded transaction bytes. The key difference between `master_write/read_memory` and `master_write/read_32` procedures is that the JTAG-to-Avalon-ST encoded byte-streams are used more efficiently for the `master_write/read_memory` procedures.

- Protocol codes as data.

  Figure 19 shows a logic analyzer trace of a 32-bit write for (address, data) = (`0x11223344`, `0x7D7C7B7A`), i.e., the write data contains the bytes-to-packets protocol codes *as data*. The logic analyzer trace shows the *ESCAPE* bytes and XORed data in the JTAG and Avalon-ST byte-streams. Figure 20 shows a logic analyzer trace of a 16-bit write for (address, data) = (`0x11223344`, `0x4D4A`), i.e., the write data contains the Avalon-ST byte-stream protocol codes *as data*. The logic analyzer trace shows the *ESCAPE* bytes and XORed data in the JTAG byte-stream.

- Address alignment.

  The `master_read/write_16/32` procedures enforce 16-bit and 32-bit address alignment at the software interface, i.e., the procedures generate error messages if the address argument is unaligned.

  The `master_read/write_memory` procedures accept an arbitrary address. For unaligned accesses the software generates a mixture of 8-bit, 16-bit, and 32-bit Avalon-MM transactions (depending on the alignment of the start address and the length of the data).

- Number of Avalon-MM transactions per JTAG transaction.

  The `master_read/write_8/16/32` procedures generate a JTAG transaction *per data value*. Each JTAG transaction consists of a 256-byte transfer; the JTAG-to-Avalon-ST header is `0xFC00 = 111_111_00_0000_0000b` which indicates that the scan length, write length, and read length are all 256-bytes. The JTAG transfer rate for multiple words is *slow*.

  The `master_read/write_memory` procedures generate a JTAG transaction for *all data values* if the address is aligned and the data length is consistent, eg., a 32-bit aligned address and a data length that is a multiple of four bytes. If the address is not aligned, or the length is not a multiple of four bytes, then several JTAG transactions will be generated. The `master_read/write_memory` procedures should be used when transferring large amounts of data over the JTAG interface, eg., filling or reading RAM contents.

- JTAG data transfer rate.

  The performance over JTAG is determined by a combination of hardware and software. The USB-Blaster determines the performance at the hardware layer. The USB-Blaster performs parallel-to-serial conversion of host-to-device bytes from USB to the JTAG TDI input, and then serial-to-parallel conversion of the JTAG TDI output to bytes for transfer of device-to-host bytes over USB. The USB-Blaster operates in two modes; bit-mode and byte-mode; bit-mode is used when manipulating the JTAG TAP machine states, and byte-mode is used when shifting data. The two modes can be see in the SignalTap II traces; at the beginning of the traces the USB-Blaster uses bit-mode and then switches to byte-mode to transfer the bulk of the byte-stream. Note how the JTAG clock (TCK) period is longer in bit-mode than in byte-mode.

  The SignalTap II traces show that when the USB-Blaster is operating in byte-mode, each byte is transfered using 8 clocks at 6MHz clock period plus about two 6MHz periods of dead-time between each byte (where TCK is high), i.e., about 10 6MHz periods per byte, or 6MHz/(10 $\times$ 1024) = 586kB/s.

The measured performance of transfers from the software layer (SystemConsole) for 4kB (0x1000), 8kB (0x2000), and 16kB (0x4000) were;

- `master_write_memory`: between 180kB/s and 280kB/s
- `master_read_memory`: between 80kB/s and 140kB/s

- JTAG-to-Avalon-ST header.

  The `master_read/write_8/16/32` procedures generate a JTAG transaction for each data value, so the encoded byte-stream for the host-to-device command and device-to-host response is small, and hence the smallest JTAG-to-Avalon-ST transaction can be used. The smallest JTAG-to-Avalon-ST transaction is 256-bytes. The header value for a 256-byte transfer is `0xFC00` = `111_111_00_0000_0000b`, which indicates that the write and read scan lengths are equal to the scan length, and the scan length is 256-bytes.

  The `master_read/write_memory` commands generate a JTAG transaction containing as much data as possible (to maximize the transfer rate). The header for the *write* transaction uses the next greater scan length needed to fit the encoded data (in increments of 256-bytes). The header always has the 6 MSBs set, to indicate that the write and read data length is set to the same length as the scan length. The header for *read* transactions is slightly different, in that the read command can be encoded in less than 256-bytes, so the write scan length (the 3-MSBs) changes to 001b (256-bytes) once the amount of read data exceeds 256-bytes, i.e., read transfers of up to 256-bytes use a header of `0xFC00`, then the header changes to `0x3C01` = `001_111_00_0000_0001b`, which indicates a write (command) scan length of 256-bytes, a read scan length equal to the scan length, and a scan length of 512-bytes. For longer read transactions, the header changes to `0x3C02` for 768-bytes, `0x3C03` for 1024-bytes, etc. (where the number of bytes in the scan is the number of encoded bytes, not the number of bytes passed via the Tcl procedure argument).

  Even though the write (command) scan length is less than the read scan length, the host still has to send data down the JTAG interface (since JTAG transactions are both a write and a read). When the write scan counter (in the JTAG-to-Avalon-ST bridge) terminates, the hardware ignores the data on TDI and inserts JTAG *IDLE* codes automatically. SignalTap II traces show TDI going low when the write counter terminates, indicating that the System-Console command is sending `0x00` data bytes. This logic is overly complicated and redundant; the host has to send JTAG data bytes anyway, so it may as well send *IDLE* codes, with the header representing just the scan length (there is no need for the header to distinguish between the read, write, and scan length). Actually, the requirement for the JTAG-to-Avalon-ST communications to occur in blocks of 256-bytes is unnecessary, but that is the subject of another document.

The SystemConsole `master` service `master_read/write_32` and `master_read/write_memory` procedures were re-implemented using the `bytestream` service procedures. Signal Tap II traces of various lengths confirmed the operation of the procedures.

The project simulation testbench, `jtag_to_avalon_mm_tb.sv`, uses the Verilog tasks in the JTAG SLD node to access the JTAG-to-Avalon-ST bridge control and status registers, and to generate various JTAG-to-Avalon-MM master transactions. This testbench can be used as the basis for simulation testbenches for testing Qsys and SOPC Systems containing the JTAG-to-Avalon-MM component.

Table 12: SystemConsole JTAG-to-Avalon-MM (`master`) service procedures.

Procedure	Arguments	Description
`master_write_8`	`<handle> <addr> <list of 8-bit values>`	Write the list of 8-bit values, starting from the specified address, using 8-bit accesses.
`master_write_16`	`<handle> <addr> <list of 16-bit values>`	Write the list of 16-bit values, starting from the specified 16-bit aligned address, using 16-bit accesses.
`master_write_32`	`<handle> <addr> <list of 32-bit values>`	Write the list of 32-bit values, starting from the specified 32-bit aligned address, using 32-bit accesses.
`master_write_memory`	`<handle> <addr> <list of 8-bit bytes>`	Write the list of bytes, starting from the specified address.
`master_read_8`	`<handle> <addr> <number of 8-bit values to read>`	Read the requested number of 8-bit values, starting from the specified address, using 8-bit accesses.
`master_read_16`	`<handle> <addr> <number of 16-bit values to read>`	Read the requested number of 16-bit values, starting from the specified 16-bit aligned address, using 16-bit accesses.
`master_read_32`	`<handle> <addr> <number of 32-bit values to read>`	ead the requested number of 32-bit values, starting from the specified 32-bit aligned address, using 32-bit accesses.
`master_read_memory`	`<handle> <addr> <number of bytes to read>`	ead the requested number of bytes, starting from the specified address.

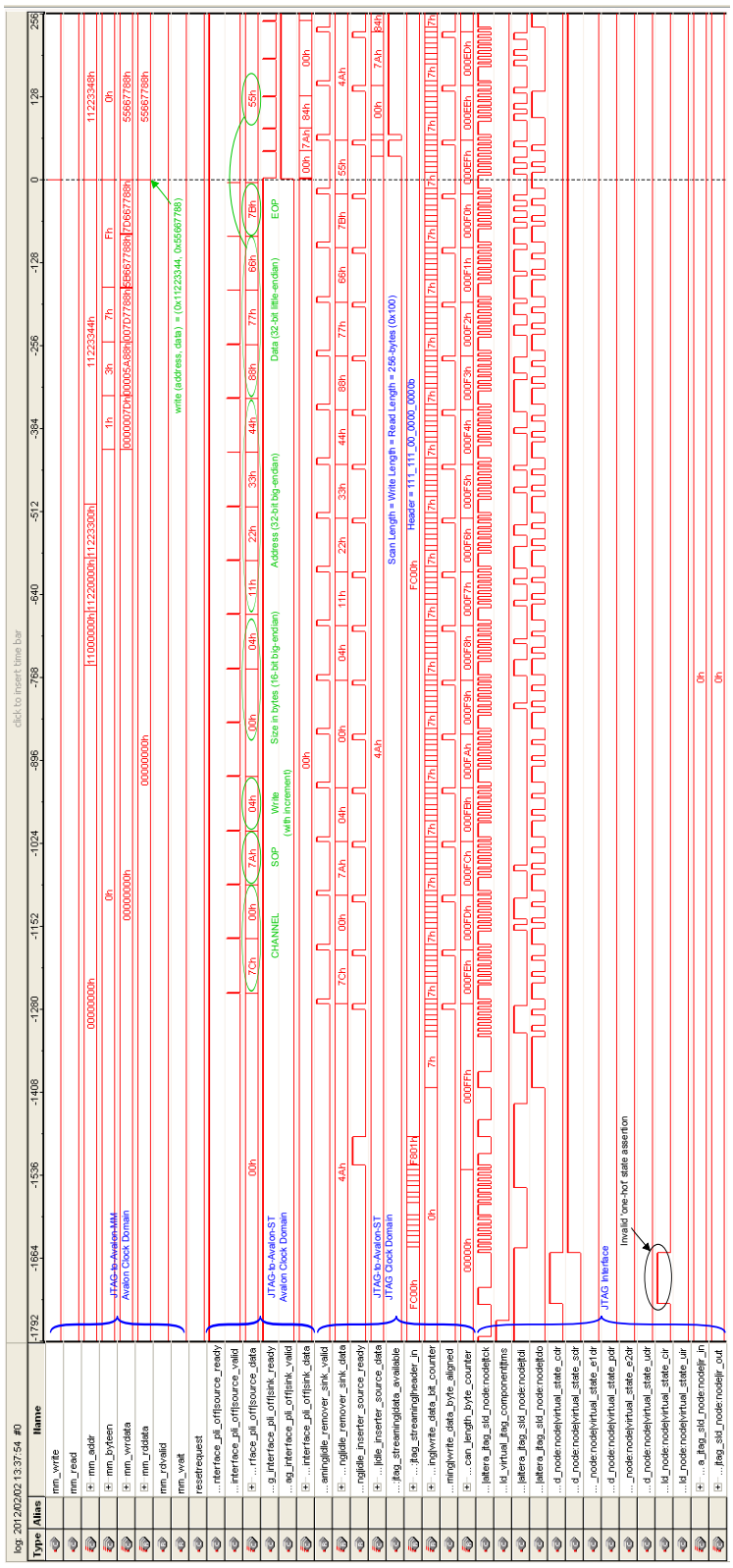

Figure 13: JTAG-to-Avalon-MM 32-bit master write (byte-stream start); (address, data) = (0x11223344, 0x55667788). The Avalon-ST *command* byte-stream contains the Avalon-MM write command bytes in Table 5, with the *write*, *incrementing* transaction code from Table 6, and the packet protocol codes from Table 7.

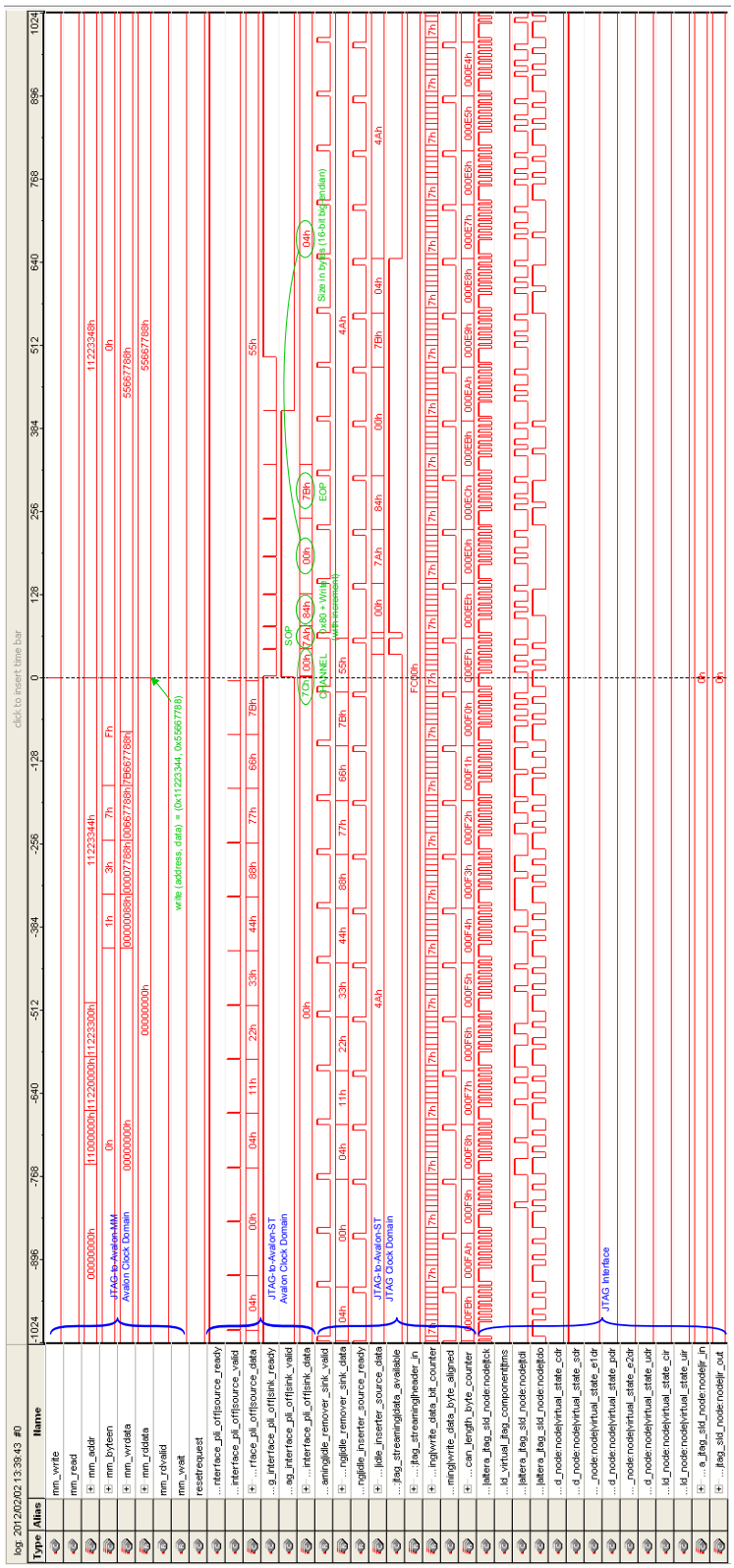

Figure 14: JTAG-to-Avalon-MM 32-bit master write (byte-stream end); (address, data) = (0x11223344, 0x55667788). The Avalon-ST *response* byte-stream contains the Avalon-MM write response bytes in Table 5, with the *write, incrementing* transaction code from Table 6 (with the MSB set), and the packet protocol codes from Table 7.

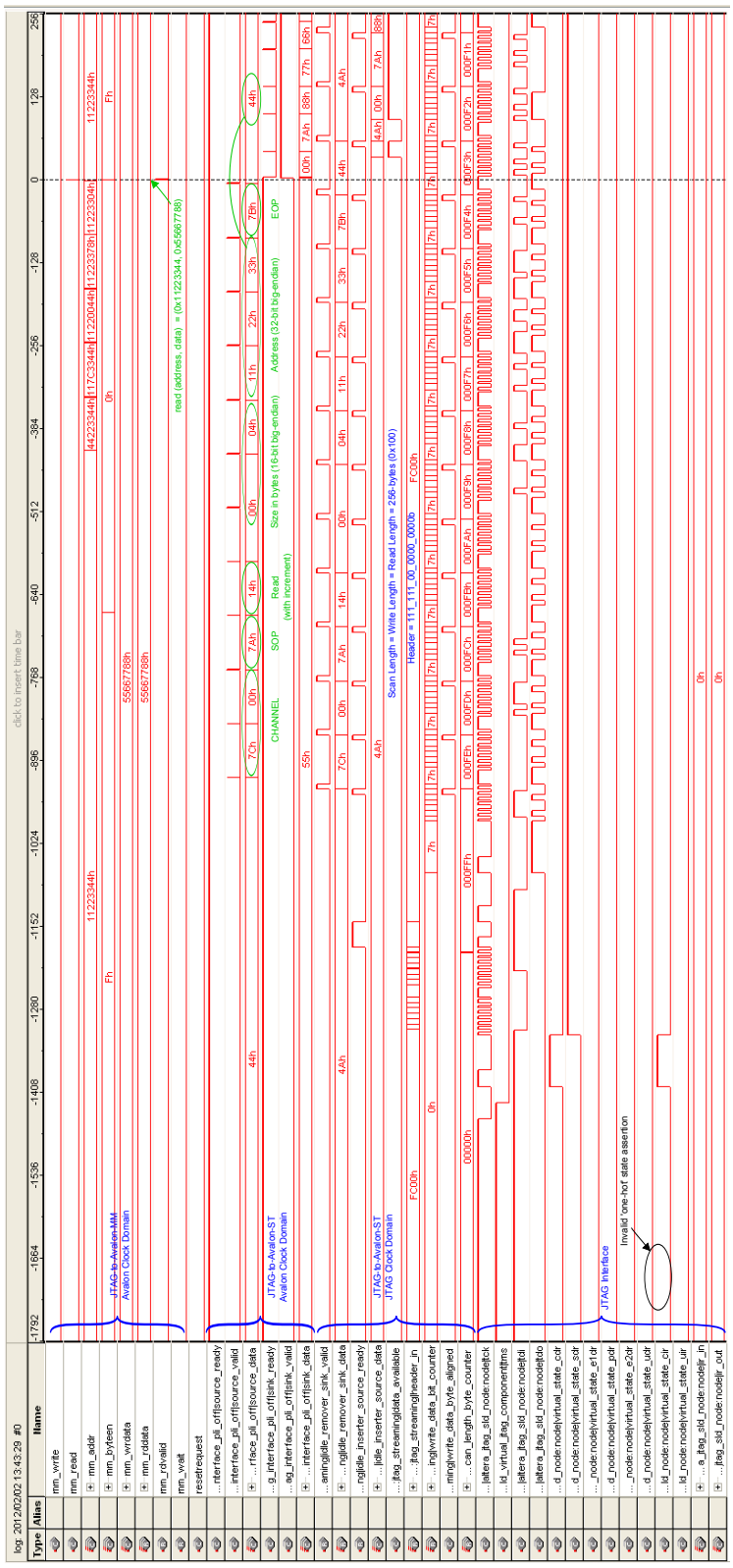

Figure 15: JTAG-to-Avalon-MM 32-bit master read (byte-stream start); (address, data) = (0x11223344, 0x55667788). The Avalon-ST *command* byte-stream contains the Avalon-MM read command bytes in Table 4, with the *read*, *incrementing* transaction code from Table 6, and the packet protocol codes from Table 7.

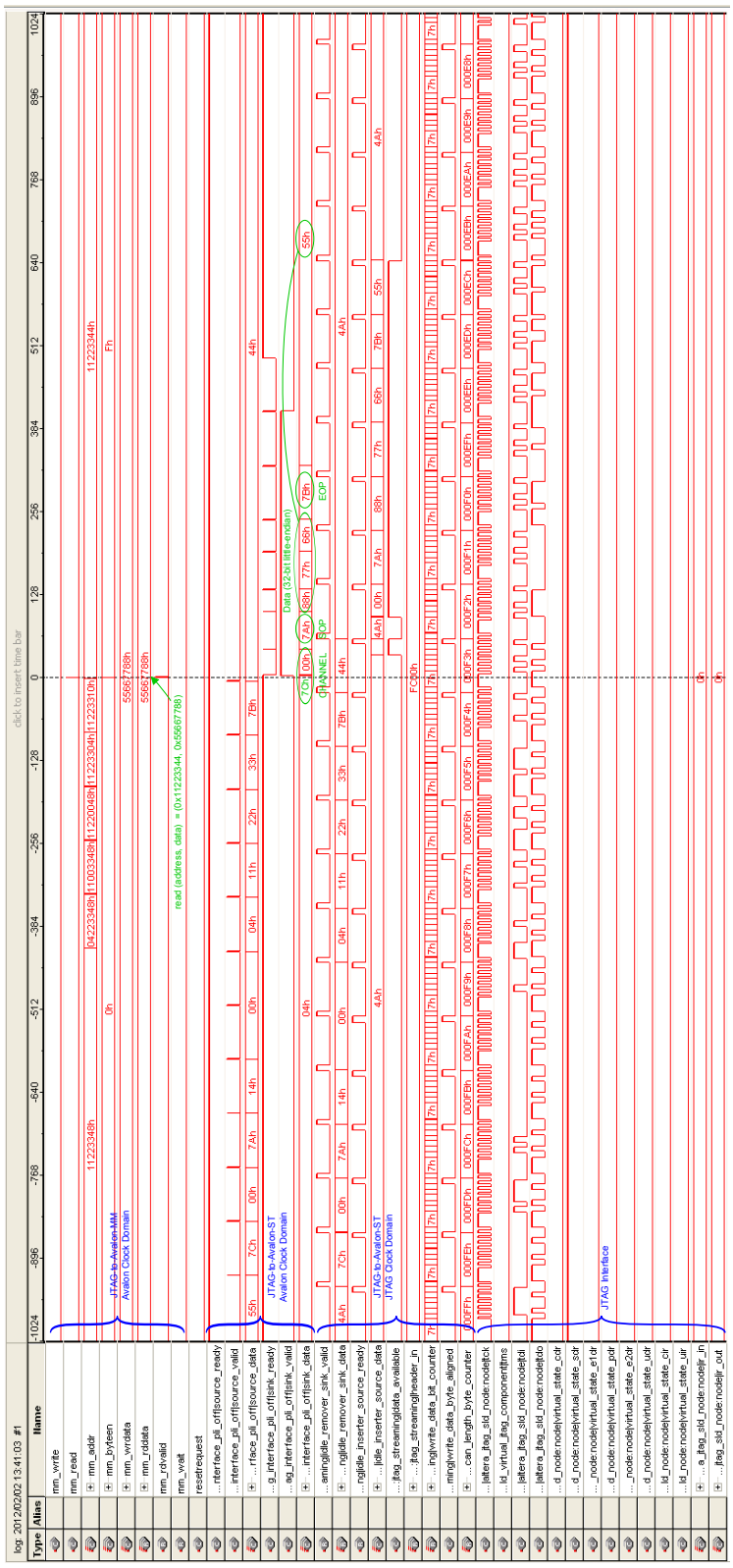

Figure 16: JTAG-to-Avalon-MM 32-bit master read (byte-stream end); (address, data) = (0x11223344, 0x55667788). The Avalon-ST *response* byte-stream contains the Avalon-MM read response bytes in Table 4, with the *read*, *incrementing* transaction code from Table 6 (with the MSB set), and the packet protocol codes from Table 7.

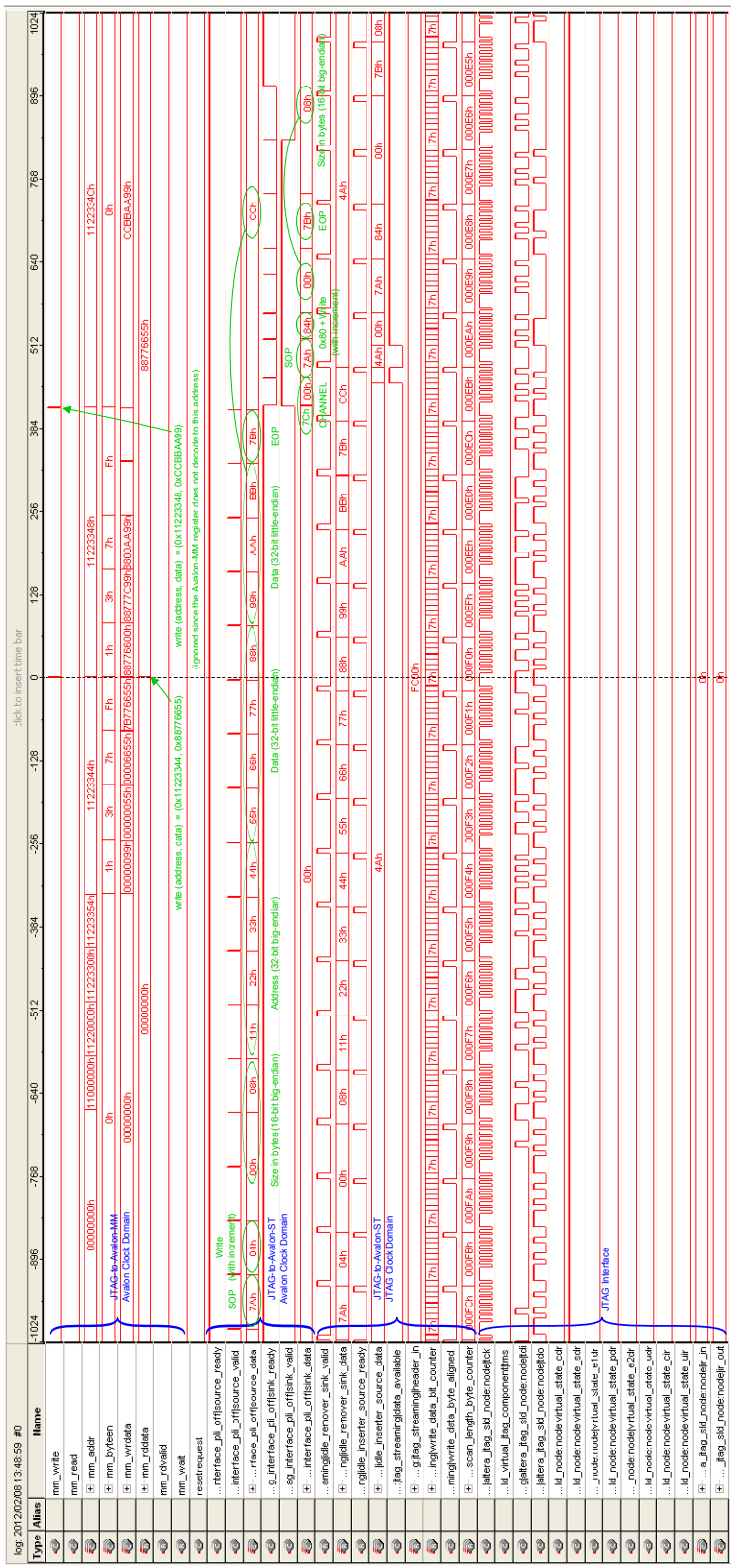

Figure 17: JTAG-to-Avalon-MM master write memory (two 32-bit writes); (address, data) = (0x11223344, {0x55, 0x66, 0x77 0x88, 0x99, 0xAA, 0xBB, 0xCC}). The Avalon-ST *command* byte-stream contains the Avalon-MM write command bytes in Table 5, with the *write, incrementing* transaction code from Table 6, and the packet protocol codes from Table 7.

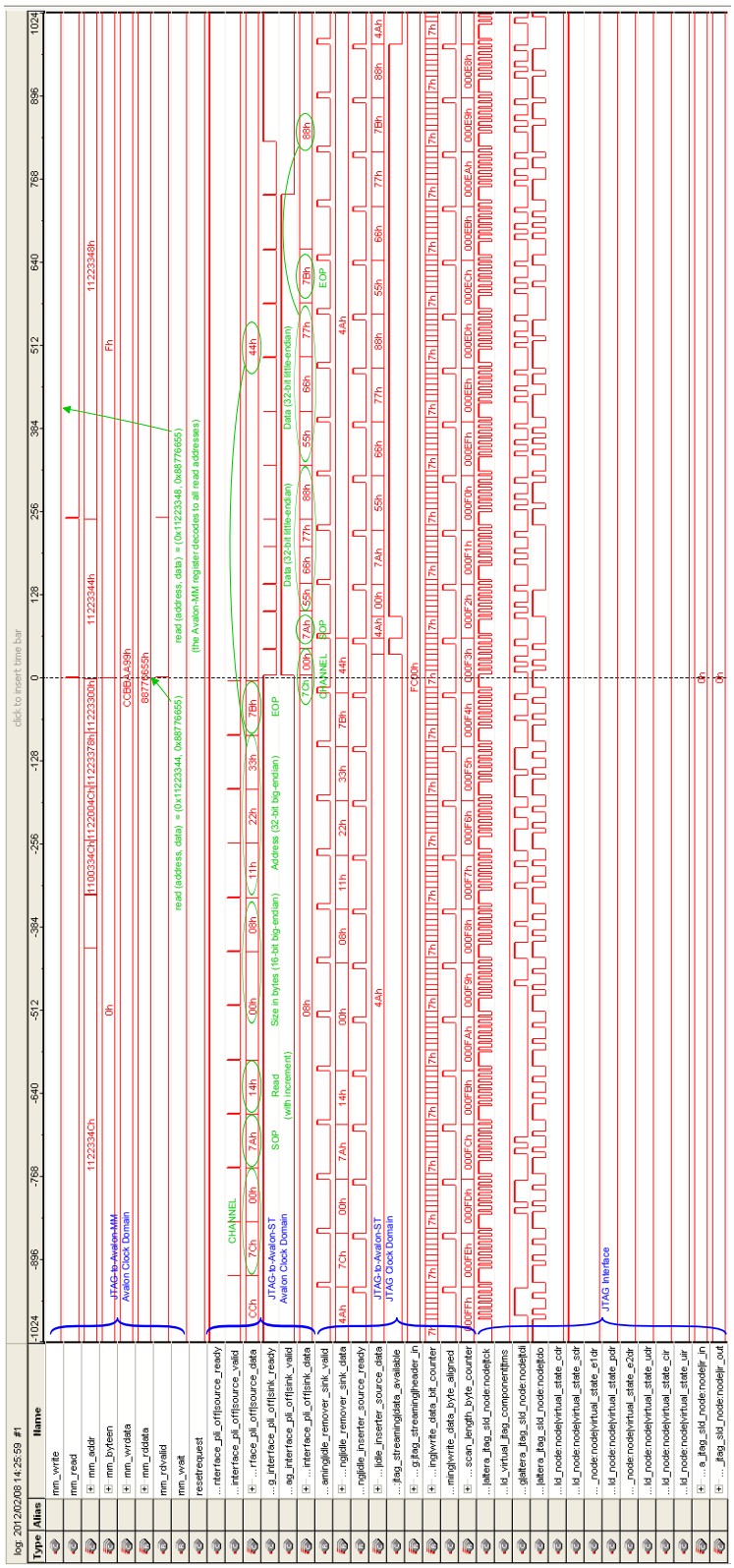

Figure 18: JTAG-to-Avalon-MM 32-bit master read memory (two 32-bit reads); (address, data) = (0x11223344, {0x55, 0x66, 0x77 0x88, 0x55, 0x66, 0x77 0x88}). The Avalon-ST *command* byte-stream contains the Avalon-MM read command bytes in Table 4, with the *read, incrementing* transaction code from Table 6, and the packet protocol codes from Table 7.

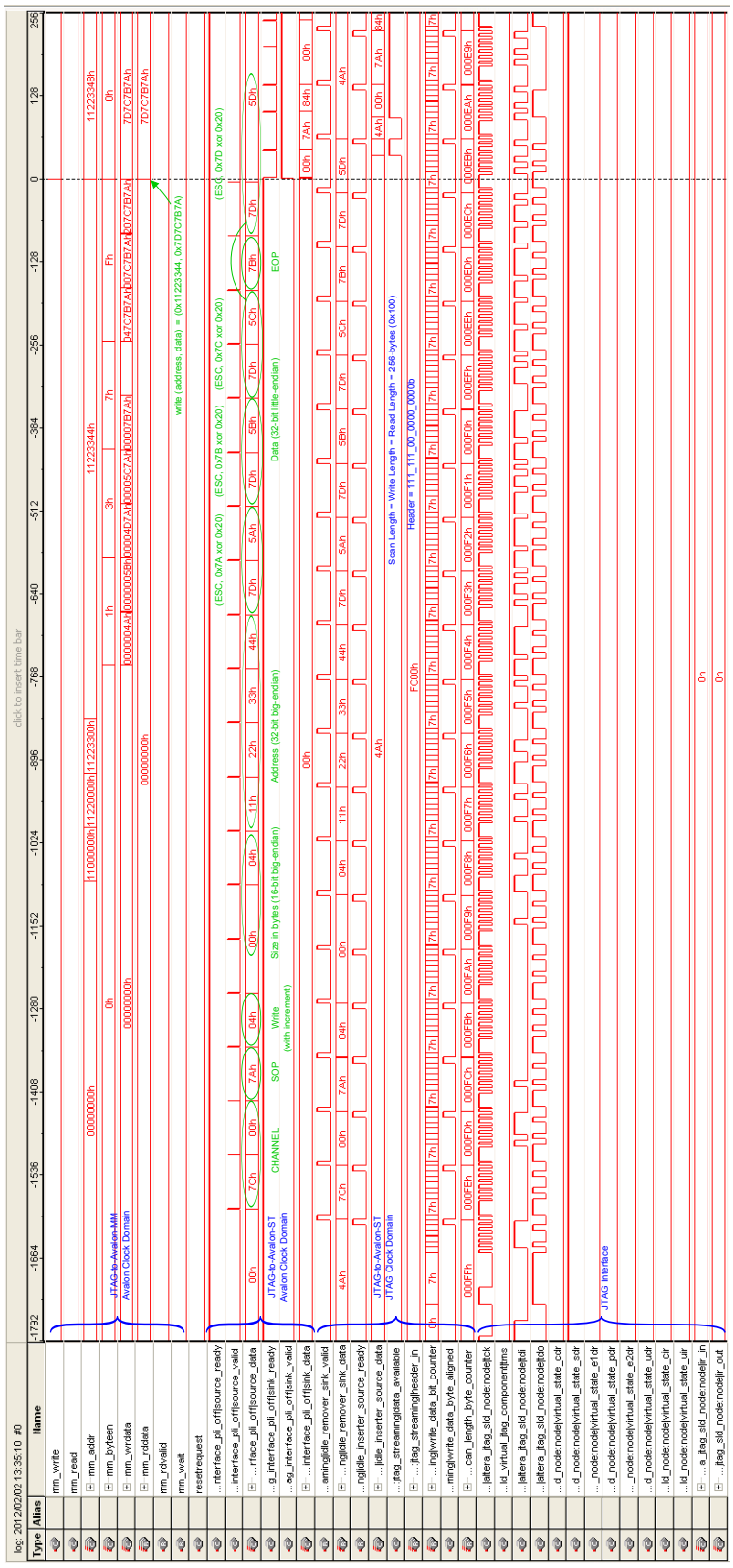

Figure 19: JTAG-to-Avalon-MM 32-bit master write (byte-stream start); (address, data) = (0x11223344, 0x7D7C7B7A). The Avalon-MM write data contains all of the packet protocol codes in Table 7 *as data*. The logic analyzer trace shows the *ESCAPE* code and the XOR modified data in both the JTAG and Avalon-ST byte-streams.

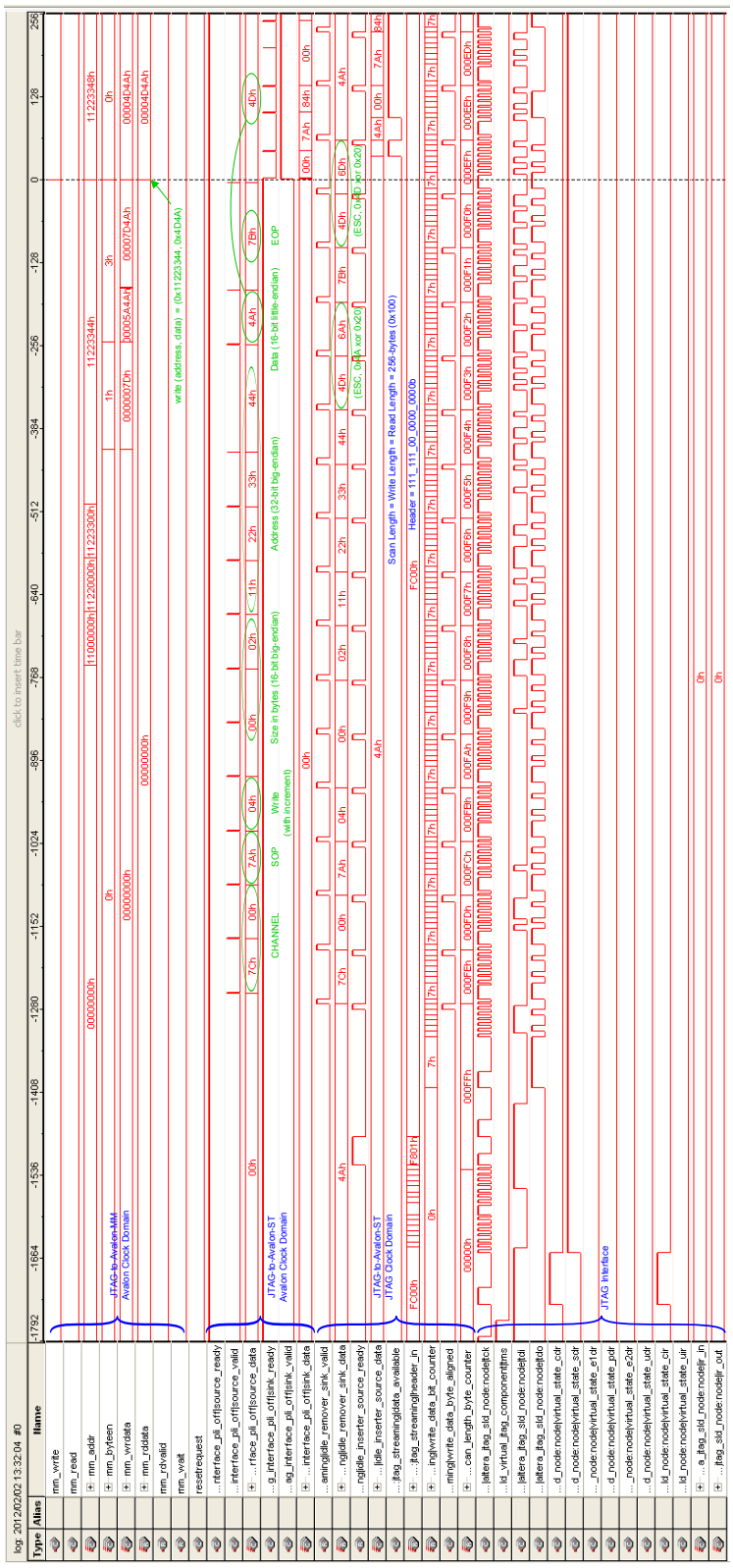

Figure 20: JTAG-to-Avalon-MM 32-bit master write (byte-stream start); (address, data) = (0x11223344, 0x4D4A). The Avalon-MM write data contains all of the Avalon-ST protocol codes in Table 3 *as data*. The logic analyzer trace shows the *ESCAPE* code and the XOR modified data in the JTAG byte-stream.

Table 13: JTAG-to-Avalon-MM (`jtag_to_avalon_mm`) project source.

Quartus II	Modelsim
10.1 and 10.1sp1 (Full) 11.0 and 11.0sp1 (Full) 11.1 and 11.1sp1 (Full)	6.6c (ASE) 6.6d (ASE) 10.0c (ASE)

A Project synthesis and simulation

This appendix describes how to synthesize and simulate each of the projects described in this document. Table 13 shows the tool versions each of the scripts has been tested with. The tools were tested under Windows XP, Windows 7, and Linux (Ubuntu 11.10 and Centos 6.2).

A.1 Synthesis

1. Start Quartus II.
2. If the Tcl console is not visible, add it to the GUI via *View→Utility Windows→Tcl Console*.
3. Click on the Tcl Console command line.
4. Change directory to the top-level of the project source, eg.,

   ```
   cd {C:\altera_jtag_to_avalon_analysis\hdl\jtag_node}
   ```

   where the path was copied from Windows Explorer into the console. The path is converted to a Tcl list using the parentheses `{` and `}`, ensuring the windows path separators are handled correctly.
5. Run the synthesis script

   ```
   source scripts/synth.tcl
   ```

   Table 14 shows the synthesis results for the three projects discussed in this document.
6. (Optional) Enable the SignalTap II trace; *Assignments→Settings*, select *SignalTap II Logic Analyzer*, and check the checkbox (the `bemicro_sdk.stp` file is already setup).
7. Re-synthesize the design by clicking the GUI 'play' button.
8. Download the design to the BeMicro-SDK; LED8 will start to flash.
9. Start SystemConsole; *Tools→Transceiver Toolkit*
10. Load the custom Tcl procedures created for the project

    ```
    source ../scripts/jtag_cmds_sc.tcl
    ```

    (the SystemConsole shell starts in the Quartus work directory, which is located in the project directory, so a relative path can be used to access the scripts directory).

Table 14: JTAG project synthesis results (for Quartus 11.0sp1 full-edition).

Project	Resource Usage			
	JTAG Interface	SLD Hub	User Logic	Total
`jtag_node`	1 LC	99 LCs	49 LCs	149 LCs
`jtag_to_avalon_st`	426 LCs	99 LCs	51 LCs	576 LCs
`jtag_to_avalon_mm`	848 LCs + 512-bits RAM	99 LCs	113 LCs	1060 LCs + 512-bits RAM

11. Start the SignalTap II logic analyzer.
12. Use the procedures in `jtag_cmds_sc.tcl` to generate transactions and trace them using SignalTap II.

A.2 Simulation

1. Start Modelsim.
2. Change directory to the top-level of the project source, eg.,

   ```
   cd {C:\altera_jtag_to_avalon_analysis\hdl\jtag_node}
   ```

   where the path was copied from Windows Explorer into the console. The path is converted to a Tcl list using the parentheses `{` and `}`, ensuring the windows path separators are handled correctly.
3. Run the simulation script

   ```
   source scripts/sim.tcl
   ```

   The simulation script creates a Tcl procedure with the same name as the testbench, eg., `jtag_node_tb`.
4. Run the testbench procedure. The procedure will add signals to the wave window and the Tcl console will output messages from the simulation.

B JTAG-to-Avalon source, software, and hardware bugs

The following are a list of bugs in the Quartus II JTAG-to-Avalon software and source code;

1. In the JTAG-to-Avalon-MM source file

   `altera_jtag_avalon_master/altera_jtag_avalon_master_pli_off.v`

   the parameter `DOWNSTREAM_FIFO_SIZE` is passed as 6, whereas it should be passed as the FIFO size (64-bytes). The JTAG-to-Avalon-ST source file

   `altera_avalon_jtag_phy/altera_jtag_streaming.v`

   calculates the parameter `DOWNSTREAM_ENCODED_SIZE` as the floor($\log_2$(`DOWNSTREAM_FIFO_SIZE`)), and then makes that value available via the *INFO* register. Hardware tests show the downstream FIFO register bits are read-back as floor($\log_2(6)$) = 2, rather than the expected 6.

2. The Verilog simulation tasks in the source file

   `altera_avalon_jtag_phy/altera_jtag_sld_node.v`

   Use a continuously running JTAG clock, TCK. This does not reflect the hardware implementation, and hides a subtle bug with respect to the use of the Virtual JTAG one-hot state signals `virtual_state_e1dr` and `virtual_state_udr`.

3. The JTAG control registers in

   `altera_avalon_jtag_phy/altera_jtag_streaming.v`

   Incorrectly use the Virtual JTAG one-hot state `virtual_state_udr` to update the registers. The logic *should* be using the `virtual_state_e1dr` one-hot state as the enable control. The `virtual_state_e1dr` signal pulses for a single clock after the JTAG state machine exits the shift-DR state (`virtual_state_sdr`). The update-DR state (`virtual_state_udr`) is used to indicate when data register have been updated, it should not be used to update the registers. This causes an issue with the `resetrequest` control, in that you can write to the register and it has no effect until the next JTAG transaction. This occurs due to the fact that the JTAG TAP ends in the `virtual_state_udr` state, and there are no TCK edges to *load* the registers until the *next* JTAG transaction.

   This error is not obvious in the simulation due to the fact that a free-running TCK is used. This generates a clock in the `virtual_state_udr` state causing the resetrequest output to update. This sequence is not reflected in the hardware, where TCK is stopped until the next JTAG transaction.

4. The SystemControl `bytestream_send` command only ever sends 1022-bytes (`0x3FE` bytes) per 1024-bytes (`0x400` bytes) transaction (where the transaction length is determined by the 16-bit byte-stream header). Perhaps the authors of the SystemConsole code incorrectly assumed that the 2-byte header was included in the length?

5. The JTAG-to-Avalon-MM SystemConsole `master` service, `master_write/read_8/16/32` procedures generate 256-byte JTAG-to-Avalon-ST transactions for each Avalon-MM transaction, eg., a call to `master_write_32` with a list of 32-bit data values generates multiple JTAG transactions to perform the 32-bit Avalon-MM bus writes. The overhead of encoding multiple 32-bit transaction results in a significant loss in performance over the JTAG interface.

   The JTAG-to-Avalon-MM SystemConsole `master` service, `master_write/read_memory` procedures can be used to generate multiple 32-bit Avalon-MM bus transactions from a single JTAG-to-Avalon-ST transaction.

   The Quartus II handbook should document the performance difference between the SystemConsole `master` procedures.

6. The SignalTap II traces in Figures 5 and 6 show glitches on the JTAG interface. This implicates the USB-Blaster as the source.

7. The SignalTap II trace in Figure 6 show an invalid Virtual JTAG one-hot state assertion (`virtual_state_cir` during a Virtual shift-DR sequence). This error must be in the implementation of the Virtual JTAG logic (the source of which is not provided by Altera).

8. In versions of Quartus II prior to version 11.1sp1, SystemConsole does not support the Tcl `fileevent` procedure. This procedure is *required* to implement a Tcl server that can support multiple clients. This type of server is *necessary* to provide JTAG-to-Avalon-MM access to software that is not written using SystemConsole.

References

[1] Altera Corporation. JTAG Configuration.
(http://www.altera.com/support/devices/configuration/schemes/jtag/cfg-jtag.html).

[2] Altera Corporation. Qsys Documentation.
(http://www.altera.com/support/software/system/qsys/sof-qsys-index.html).

[3] Altera Corporation. SOPC Builder Documentation.
(http://www.altera.com/literature/lit-sop.jsp).

[4] Altera Corporation. Virtual JTAG Megafunction User Guide (version 2.0), December 2008.
(ug_virtualjtag.pdf).

[5] Altera Corporation. Avalon Interface Specifications (version 1.3, for SOPC Systems), August 2010. (mnl_avalon_spec_1_3.pdf).

[6] Altera Corporation. Avalon Interface Specifications (version 2.0, for Qsys Systems), May 2011.
(mnl_avalon_spec.pdf).

[7] Altera Corporation. Embedded Peripherals IP User Guide (version 11.0), June 2011.
(ug_embedded_ip.pdf).

[8] Altera Corporation. Quartus II Handbook (version 11.1), November 2011.
(quartusii_handbook.pdf).

[9] D. W. Hawkins. Altera Virtual JTAG Analysis, November 2011.
(vjtag.pdf).

[10] W. Simpson. PPP in HDLC-like Framing, July 1994.
(http://tools.ietf.org/html/rfc1662).