FTDI USB-to-Serial/Parallel Interface Design

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

February 5, 2013

Contents

1 Introduction

Future Technologies Devices International (FTDI), http://www.ftdichip.com/, manufactures a range of USB-to-Serial (UART) and USB-to-Parallel (FIFO) devices. These devices provide a simple and cost-effective method for implementing a USB interface to a Field Programmable Gate Array (FPGA). The FTDI devices encapsulate the complexity of the USB interface, and provide a simple serial or parallel interface to the FPGA.

This document provides details on interfacing FTDI devices to Altera FPGAs, including;

- VHDL source for;
  - Simulation Bus Functional Models (BFMs).
  - An FTDI FIFO mode to Avalon-ST bridge.
  - An FTDI FIFO mode to Avalon-MM bridge.
  - A UART to Avalon-ST bridge.
  - A UART to Avalon-MM bridge.
- Software interfacing using FTDI's direct access programming interface (Windows and Linux).
- Hardware resource and performance measurements.

Table 1 contains a list of FTDI devices that can use these bridges to interface to an FPGA. The FT232B USB-to-serial and FT245B USB-to-FIFO are the earlier generation FTDI devices. The FT230X and FT240X are newer generation devices with internal EEPROM and internal oscillators. The part numbers for the devices do not always indicate the device features, eg., the F232B is a serial device, the FT232H, FT2232D, and FT2232H support both serial and parallel modes, but the FT4232H supports only serial mode. The USB Full-Speed devices can achieve asynchronous FIFO mode data rates of 1MB/s. The USB High-Speed devices can achieve asynchronous FIFO mode data rates of 8MB/s, and synchronous FIFO mode data rates of 40MB/s.

Development and testing was performed using the following hardware;

- FTDI UM245R FT245R development module.
- FTDI UM232H FT232H development module.
- FTDI Morph-IC; Altera ACEX 1K FPGA plus FT2232C dual-channel USB-to-Serial/Parallel adapter.
- Altera MAX II PCI development kit; Altera MAX II CPLD plus an FT245BM USB-to-Parallel adapter.
- Arrow BeMicro; Altera Cyclone III plus FT2232C dual-channel USB-to-Serial/Parallel adapter.
- Arrow BeMicro-SDK; Altera USB-Blaster plus Cyclone IV FPGA.

Table 1: FTDI USB-to-Serial/Parallel Interface Devices.

Part Number	Number of Serial Channels	Number of FIFO Channels		FIFO Mode Write Polarity	EEPROM	Reference
		Asynchronous Mode	Synchronous Mode			
USB Full-Speed (12Mbps) Devices						
FT245B		1		Active-High	External	[3]
FT245R		1		Active-High	Internal	[6]
FT232B	1				External	[8]
FT232R	1				Internal	[5]
FT2232D	2	2		Active-High	External	[4]
FT230X	1				Internal	[10]
FT240X		1		Active-High	Internal	[12]
USB High-Speed (480Mbps) Devices						
FT232H	1	1	1	Active-Low	External	[11]
FT2232H	2	2	1	Active-Low	External	[9]
FT4232H	4				External	[7]

2 FTDI Interface Modes

The FTDI devices in Table 1 have two primary interface modes; serial/UART mode or parallel/FIFO mode. The parallel/FIFO mode has three different operating modes; asynchronous FIFO mode with active-high write-enable (on the USB low-speed devices) or active-low write-enable (on the USB high-speed devices), and synchronous FIFO mode (only available on the USB high-speed devices).

2.1 Serial/UART mode

The USB-to-Serial device serial interface implements a standard serial port or asynchronous UART, i.e., the logic-level side of an RS232 interface. The timing of the USB-to-Serial UART interface is programmable, with user defined baud-rate (bit-rate), number of data bits, number of stop bits, and parity. Read the datasheets referenced in Table 1 for details.

A typical application of a USB-to-Serial device with an FPGA, is to implement an FPGA processor with a UART, which connects to the USB-to-Serial transmit and receive signals. Software running on the FPGA processor then processes the serial byte-stream, eg., presents the user with a serial console. Section 5.3 shows an alternative, hardware-centric solution, where the serial byte-stream is processed directly by an FPGA state machine.

2.2 Asynchronous FIFO mode

Table 2 and Figure 1 show the timing parameters for asynchronous FIFO mode with *active-high* write-enable for USB full-speed devices (eg., see the FT245R datasheet pp13-14 [6]). Table 3 and Figure 3 show the timing parameters for asynchronous FIFO mode with *active-low* write-enable for USB full-speed devices (eg., see the FT232H datasheet p29 [11]).

The FPGA interface to either asynchronous FIFO mode must meet the minimum timing requirements shown in the tables and figures. Figures 1 and 2 show the uncertainty in the FTDI device output signal timing in red, and the uncertainty in the FPGA output (FTDI input) signal timing in blue. Because the FTDI outputs are *asynchronous*, the FPGA must use synchronizers to avoid issues created by metastability. Because of the uncertainty in the FPGA output timing, additional timing margin must be provided (by the FPGA logic) to ensure that the read and write pulse minimum timing is guaranteed to be met. FPGA logic can meet the FTDI read and write pulse *minimum* timing requirements by asserting those signals for longer than the minimum requirement (at least as long as the FTDI requirement plus FPGA output clock-to-output uncertainty).

The main advantage of the asynchronous FIFO mode with regards to FPGA interfacing, is that any FPGA can meet the timing requirements, i.e., both low-cost and high-end FPGAs can be used.

2.3 Synchronous FIFO mode

Table 4 and Figure 3 show the timing parameters for synchronous FIFO mode (eg., see the FT232H datasheet pp27-28 [11]). Synchronous FIFO mode is only supported on high-speed devices.

The FPGA interface to synchronous FIFO mode must meet the timing requirements shown in Table 4 and Figure 3. In synchronous mode, the FTDI device outputs a 60MHz (16.67ns) clock. The FTDI outputs have a maximum clock-to-output delay of 9.0ns, resulting in 7.67ns of available setup time for the FPGA inputs. The FTDI inputs have a minimum clock-to-output delay of 0ns, resulting in 0ns of available hold time for the FPGA inputs. The FTDI inputs have a minimum setup time of 7.5ns, which constrains the FPGA outputs to have a maximum clock-to-output delay of 9.17ns. The FTDI inputs have a minimum hold time of 0ns, which constrains the FPGA outputs to have a minimum clock-to-output delay of 0ns (which is easily met). These timing constraints are applied to Altera FPGAs using a Quartus TimeQuest SDC constraints file.

The main issue when interfacing an FPGA to synchronous FIFO mode are; the FPGA *must* meet the 60MHz interface timing requirements (there is no option to change the FTDI output clock frequency). There is another caveat with synchronous mode; the FTDI device only generates the 60MHz output clock once *software* has configured the device to operate in *synchronous* FIFO mode. The FTDI configuration EEPROM does not have an option to select synchronous mode at power-on, the EEPROM must be configured to select asynchronous FIFO mode, and then *software* must be used to select synchronous mode. If stand-alone operation of the FTDI+FPGA board is desired, then an on-board clock is required. If that on-board clock is used for the FPGA control logic, then the FTDI interface bridge requires clock-domain crossing logic. Additional logic would also be recommended to hold the FTDI-side logic in reset until the FTDI 60MHz clock was detected. This logic would ensure that the bridge logic was correctly enabled and disabled. For example, consider the case of a self-powered FPGA board, if the user removes and replugs the USB cable, the FTDI 60MHz clock will be disabled until software enables synchronous FIFO mode. An FPGA PLL could be used to implement this reset logic, by using the FTDI clock as the PLL source, and using the PLL locked output as the reset source (the bridge logic could either be clocked by the FTDI clock directly or by the PLL output).

Table 2: FTDI asynchronous FIFO mode timing parameters for full-speed devices.

Parameter	Minimum (ns)	Maximum (ns)	Description
Read			
t_{RD}	50		RD# active pulse width
t_{RD2RXF}	0	25	RD# inactive to RXF# inactive
t_{RXF}	80		RXF# inactive after read cycle
t_{DQV}	20	50	RD# active to valid data
t_{DQX}	0		RD# inactive to invalid data
t_{RD2RD}	130		RD# to RD# pre-charge time
Write			
t_{WR}	50		WR active pulse width
t_{WR2TXE}	5	25	WR inactive to TXE# inactive
t_{TXE}	80		TXE# inactive after write cycle
t_{SU}	20		Data setup time before WR falling edge
t_{H}	0		Data hold time after WR falling edge
t_{WR2WR}	80		WR to WR pre-charge time

Table 3: FTDI asynchronous FIFO mode timing parameters for high-speed devices.

Parameter	Minimum (ns)	Maximum (ns)	Description
Read			
t_{RXF2RD}	0		RXF# active to RD# active
t_{RD}	30		RD# active pulse width
t_{RD2RXF}	1	14	RD# inactive to RXF# inactive
t_{RXF}	49		RXF# inactive after read cycle
t_{DQV}	1	14	RD# active to valid data
t_{DQX}	1		RD# inactive to invalid data
Write			
t_{TXE2WR}	0		TXE# active to WR# active
t_{WR}	30		WR# active pulse width
t_{WR2TXE}	1	14	WR# active to TXE# inactive
t_{TXE}	49		TXE# inactive after write cycle
t_{SU}	5		Data setup time before WR# falling edge
t_{H}	5		Data hold time after WR# falling edge

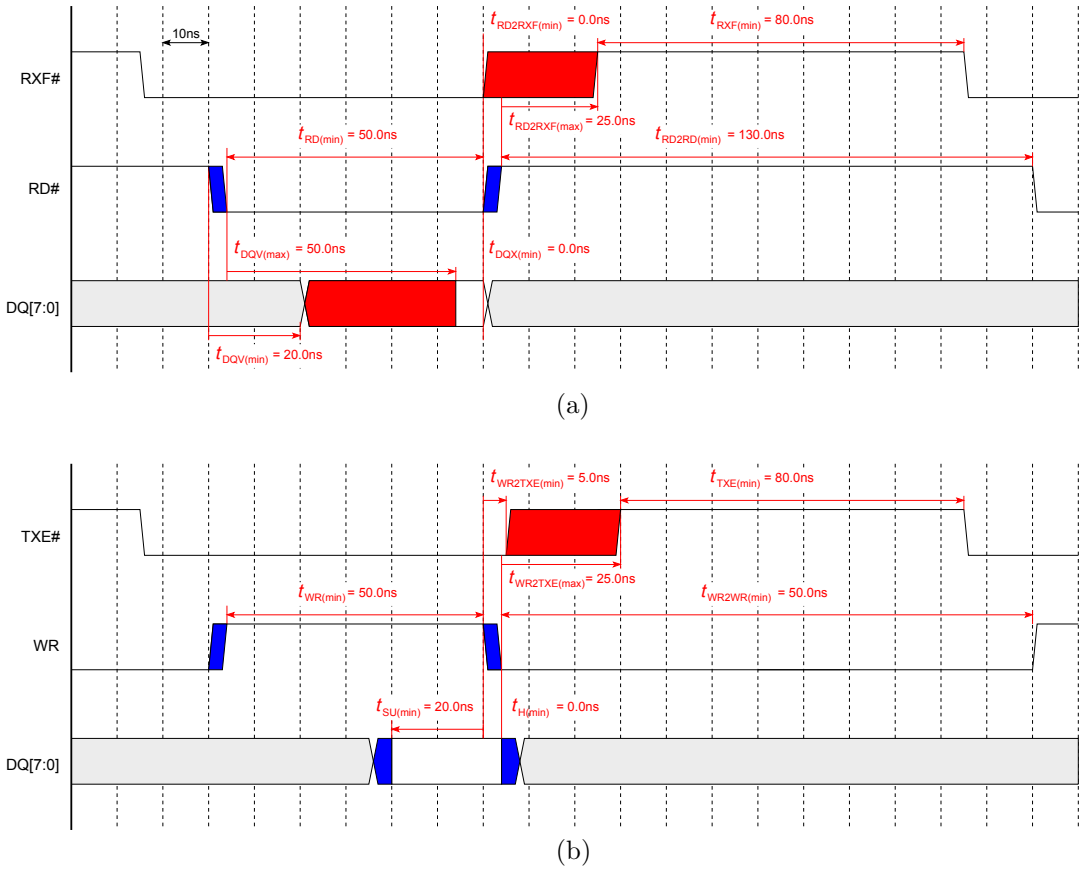
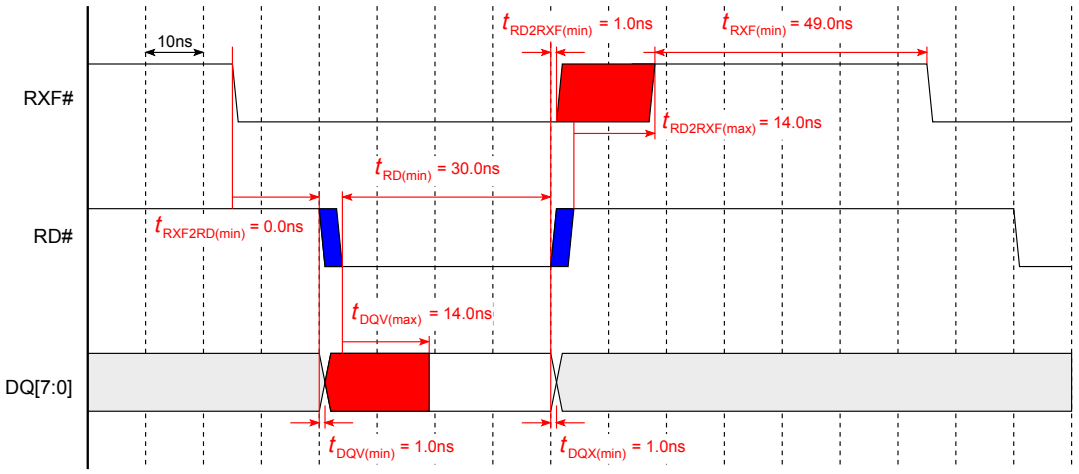
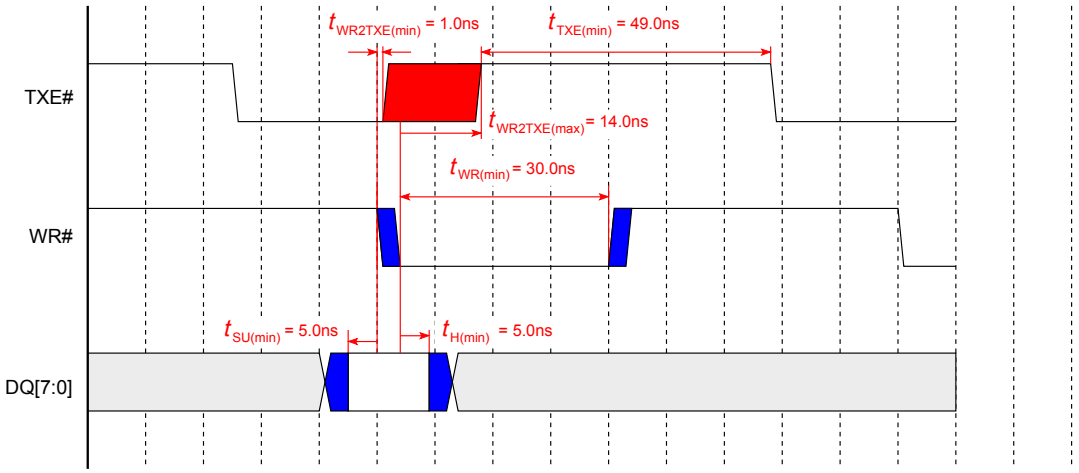


Figure 1: FTDI asynchronous FIFO mode timing for full-speed devices; (a) read and (b) write.



(a)

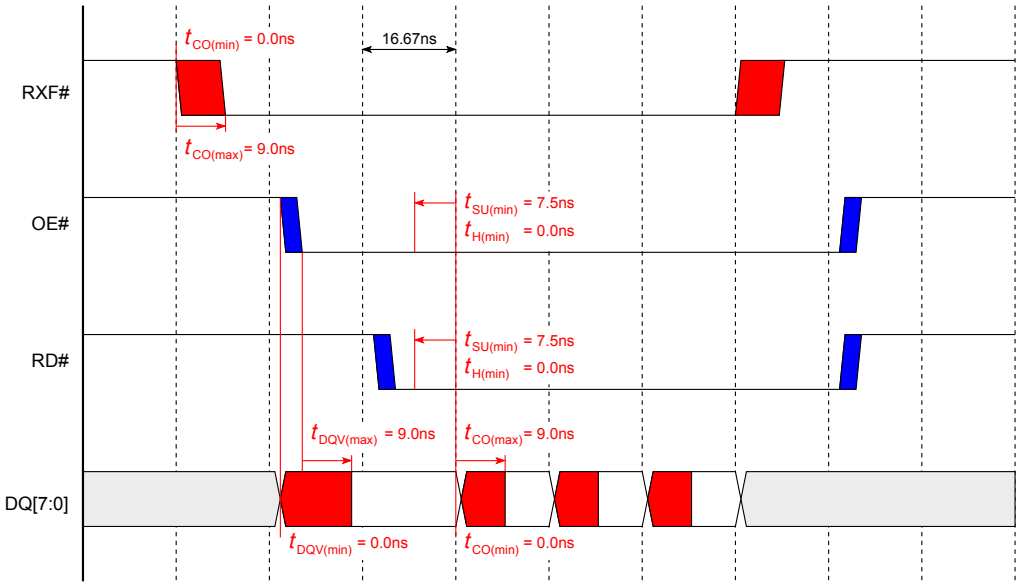


(b)

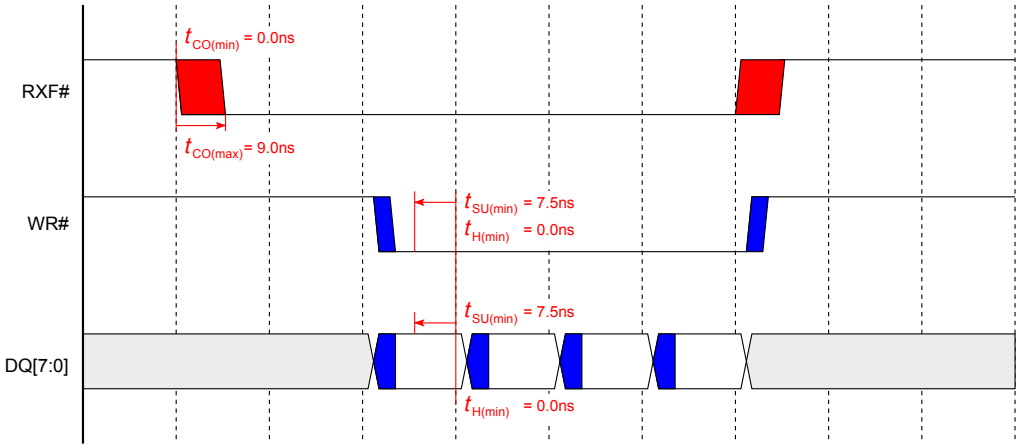
Figure 2: FTDI asynchronous FIFO mode timing for high-speed devices; (a) read and (b) write.

Table 4: FTDI synchronous FIFO mode timing parameters.

Parameter	Minimum (ns)	Maximum (ns)	Description
t_{SU}	7.5		Input setup time
t_{H}	0		Input hold time
t_{CO}	0	9	Output clock-to-output delay
t_{DQV}	0	9	Output-enable to data valid



(a)



(b)

Figure 3: FTDI synchronous FIFO mode timing; (a) read and (b) write.

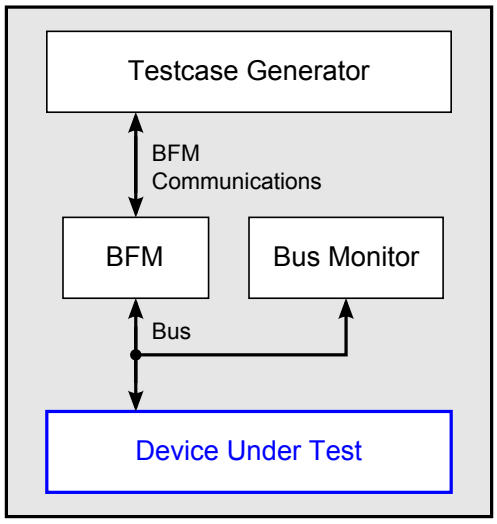


Figure 4: Generic testbench architecture.

3 Bus Functional Models

Figure 4 shows a block diagram of a generic testbench. The testbench consists of the *simulation-only* testcase generator, bus functional model (BFM), and bus monitor, and the *synthesizeable* device under test (DUT).

The BFM provides an abstraction that allows the testcase generator to perform transactions with the device under test, without regards to how those commands are physically implemented at the bus (hardware) level. The BFM is a server-like process that accepts BFM commands from the testcase generator and converts the commands into bus transactions that adhere to the bus protocol. A bus monitor checks timing and adherence to the bus protocol.

BFM communications between the testcase generator and server is implemented using a VHDL resolved signal containing a record. The VHDL record describes the operation the BFM server processor is to perform, and contains data to, or from, the server process. A BFM package typically provides high-level commands for use by the testcase generator, eg., read and write commands with address and data arguments. The BFM package procedure implementation consists of packing the high-level procedure input arguments into the BFM record, performing an interlocked-handshake to pass the record to the server, an interlocked-handshake to receive any response, and unpacking of high-level procedure output arguments. For additional details see the BFM source code, and Bergeron [2].

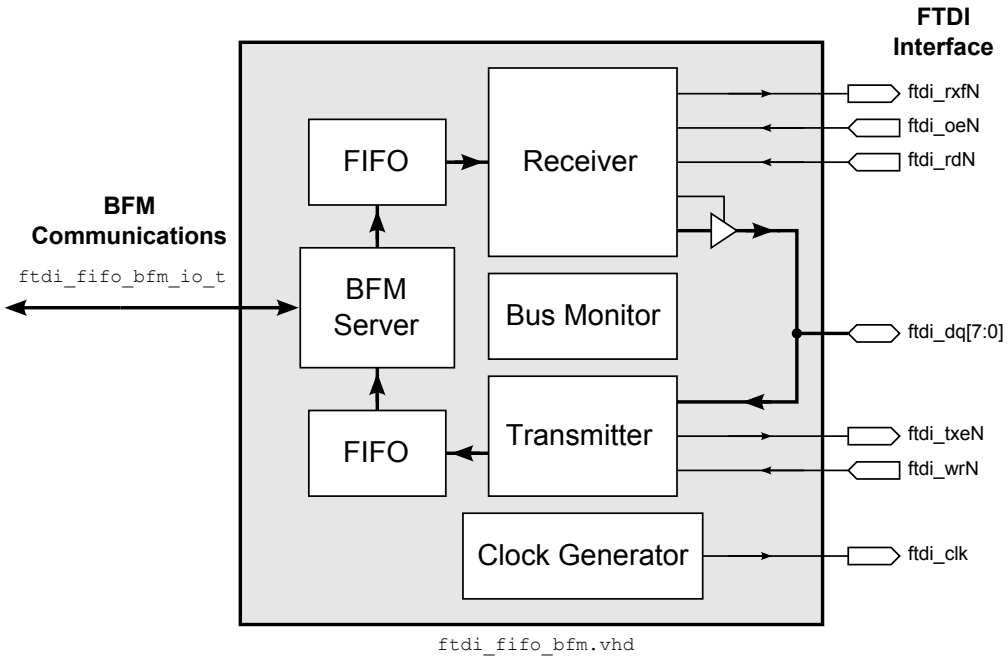


Figure 5: FTDI BFM architecture.

3.1 FTDI FIFO Mode BFM

Figure 5 shows a block diagram of the FTDI FIFO mode BFM. The BFM can be configured for full-speed (active-high write-enable) or high-speed (active-low write-enable) USB device mode, and for asynchronous FIFO or synchronous FIFO mode using VHDL generics. The BFM contains a server process that communicates with the testcase generator. Data from the testcase generator is written by the server to the receiver FIFO, and the receiver process generates the FTDI bus transaction. The transmitter process manages data transfers from the device under test, into the transmitter FIFO. When the testcase generator performs a read transaction, the server process waits for data in the transmitter FIFO, and then sends it to the testcase generator. The FIFOs in the BFM allow it to perform asynchronous transactions, analogous to the FIFOs in the FTDI device.

In asynchronous FIFO mode, the receiver and transmitter processes generate asynchronous output signal transitions delayed in accordance with the data sheet parameters, and the bus monitor checks the timing of the input signals from the device under test. Assertions terminate the simulation if any input signal data sheet timing violations are detected.

In synchronous FIFO mode, the receiver and transmitter processes generate signal transitions with a nominal clock-to-output delay of 1ns, and there is no bus monitoring. The timing analysis of the synchronous interface is performed by an Altera TimeQuest analysis script after synthesis.

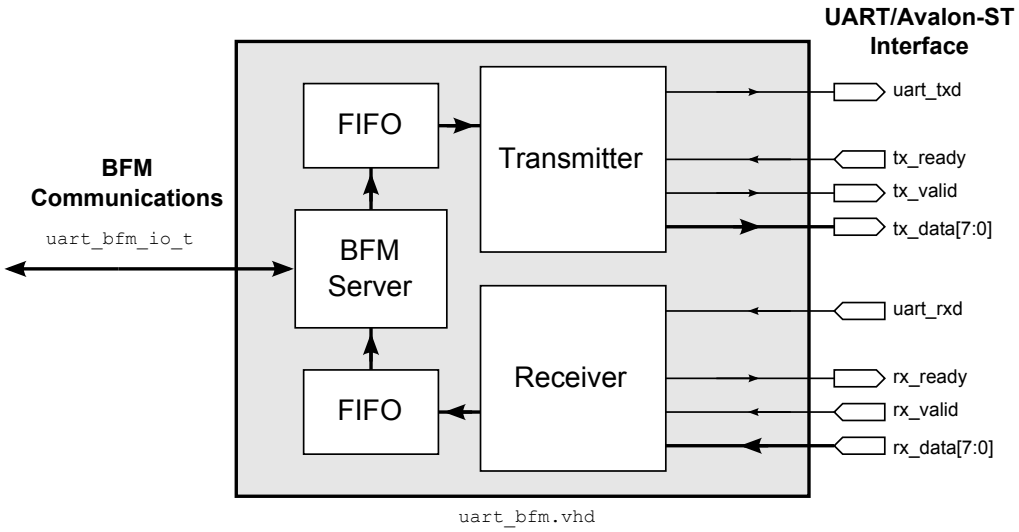


Figure 6: UART/Avalon-ST BFM architecture.

3.2 UART/Avalon-ST BFM

Figure 6 shows a block diagram of the UART/Avalon-ST BFM. The BFM can independently configure the transmitter and receiver interfaces for serial UART or parallel Avalon-ST mode. The dual-mode interfaces make the BFM useful for testing serial, serial-to-parallel, parallel-to-serial, and parallel components.

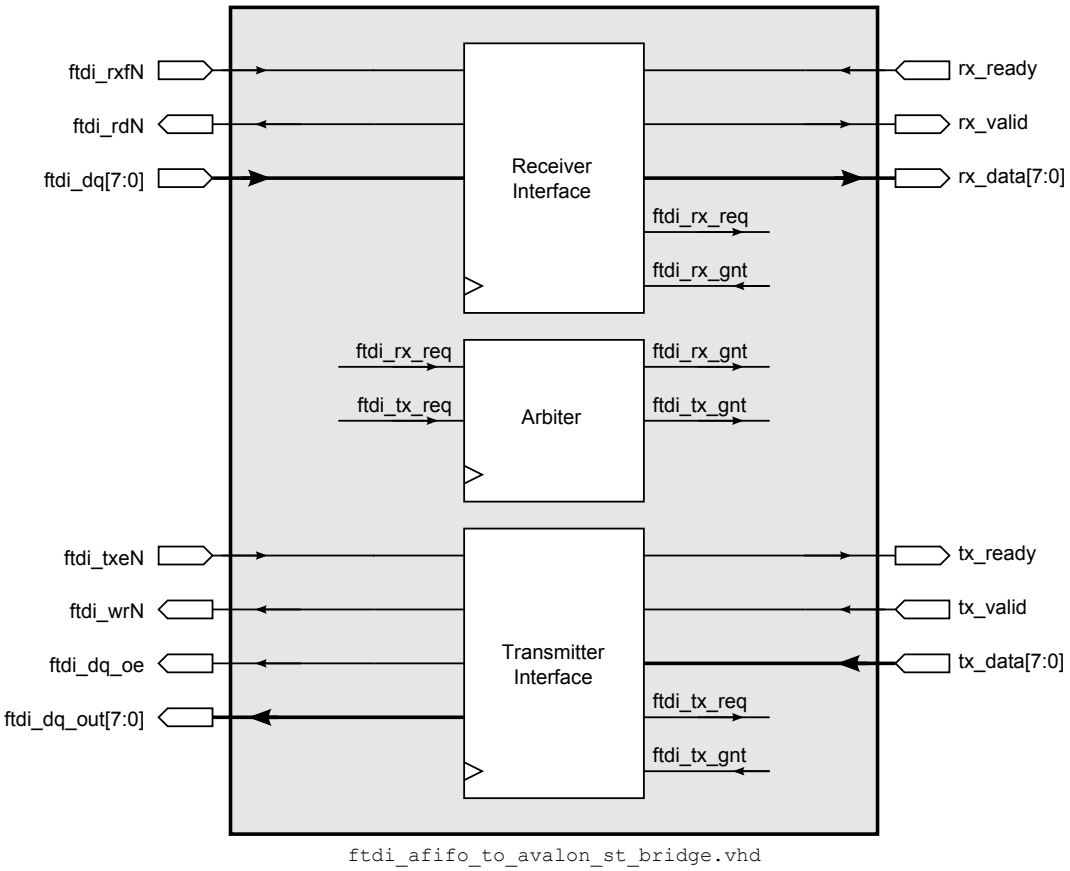


Figure 7: FTDI asynchronous FIFO to Avalon-ST bridge diagram.

4 Avalon-ST Bridges

The Altera *Avalon Interface Specifications* [1] defines protocols for implementing streaming (Avalon-ST) and memory-mapped (Avalon-MM) interfaces. This section describes components that convert FTDI serial and parallel byte-streams into Avalon-ST byte-streams.

4.1 Asynchronous FIFO to Avalon-ST Bridge

The FTDI *asynchronous* FIFO to Avalon-ST bridge is a general purpose interface that synchronizes the FTDI asynchronous FIFO host-to-device (receiver) and device-to-host (transmitter) byte-streams into synchronous Avalon-ST source and sink byte-streams. Figure 7 shows a block diagram of the bridge component. VHDL generics are used to configure the bridge for use with either full-speed or high-speed devices (which configures the write-enable, `ftdi_wrN`, to be active-high or active-low), and to configure the read and write timing parameters.

Figure 7 shows that the bridge design is partitioned into two main interfaces; host-to-device (receiver) and device-to-host (transmitter). Both interfaces share the common 8-bit FTDI data bus, `ftdi_dq[7:0]`, so an arbiter is used to grant access to the bus on a first-come-first-served basis. Figure 8 shows the bus arbiter finite state machine (FSM) algorithmic state machine (ASM) chart.

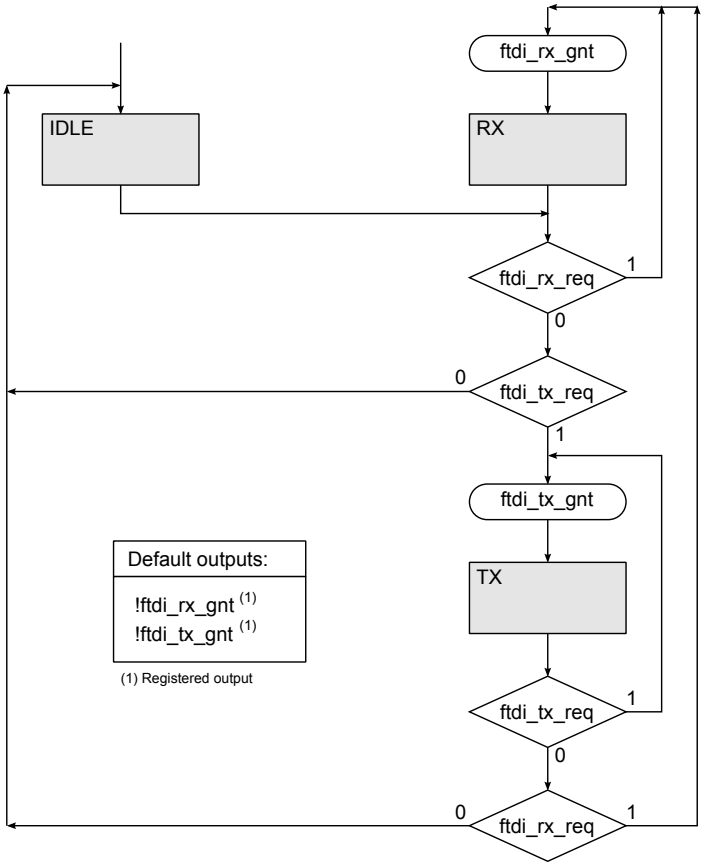


Figure 8: FTDI bus arbiter FSM.

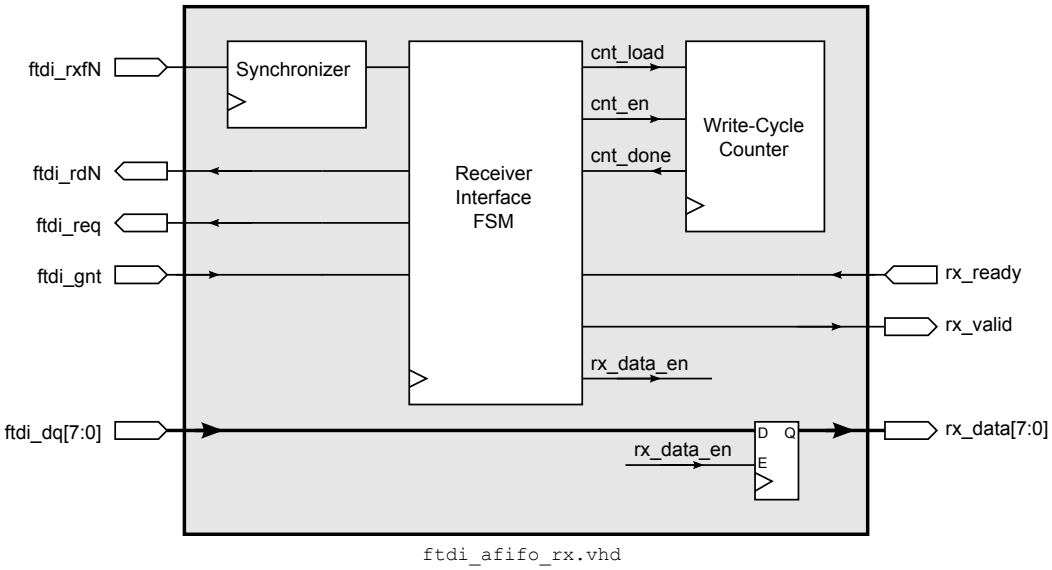


Figure 9: FTDI asynchronous FIFO to Avalon-ST bridge receiver interface diagram.

Receiver (Read) Interface

Figure 9 shows a block diagram of the bridge receiver interface. Figure 10 shows the receiver timing sequence implemented by the finite state machine (FSM) in Figure 11.

The receiver (read) sequence in Figure 10 proceeds as follows;

- The FTDI asynchronous output `ftdi_rxfN` asserts low to indicate there is data in the receive FIFO. The FPGA synchronizer output signal `ftdi_rxfN_sync` routes to the `ftdi_rxfN` input of the FSM, i.e., the signal `ftdi_rxfN` in the ASM chart is the *synchronized* signal `ftdi_rxfN_sync` shown in the receiver datapath section of the timing diagram.
- Assertion of the synchronized `ftdi_rxfN` causes the FSM to assert the FTDI bus request signal, `ftdi_req`, and wait until the grant signal is asserted.
- Assertion of the grant signal, `ftdi_gnt`, causes the FSM to load the read-cycle counter. The read-cycle counter load value is determined by the clock frequency and read-cycle time generics.
- The FSM transitions to the `READ` state, where it asserts `ftdi_rdN` and the counter enable control, until the counter done (carry-out) signal asserts.
- When the counter done signal asserts, the FSM asserts the receive data enable control, `rx_data_en`, to capture the data in the `rx_data[7:0]` register.
- The FSM then transitions to the `WAIT` state, where it waits for the read transaction to complete on the FTDI bus, as indicated by the synchronized version of `ftdi_rxfN` deasserting.
- The FSM then transitions to the `READY` state, where it asserts the Avalon-ST source `rx_valid` output until `rx_ready` is asserted.
- The FSM then transitions to the `IDLE` state.

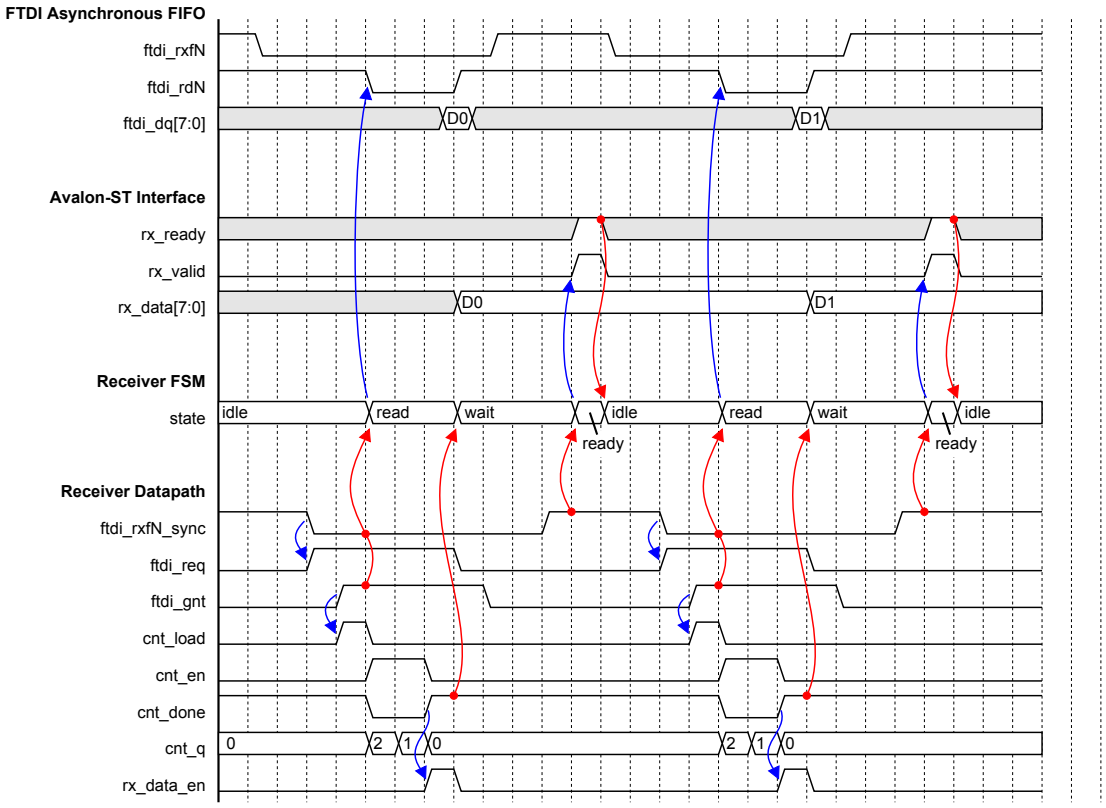


Figure 10: FTDI asynchronous FIFO receiver (host-to-device) interface timing.

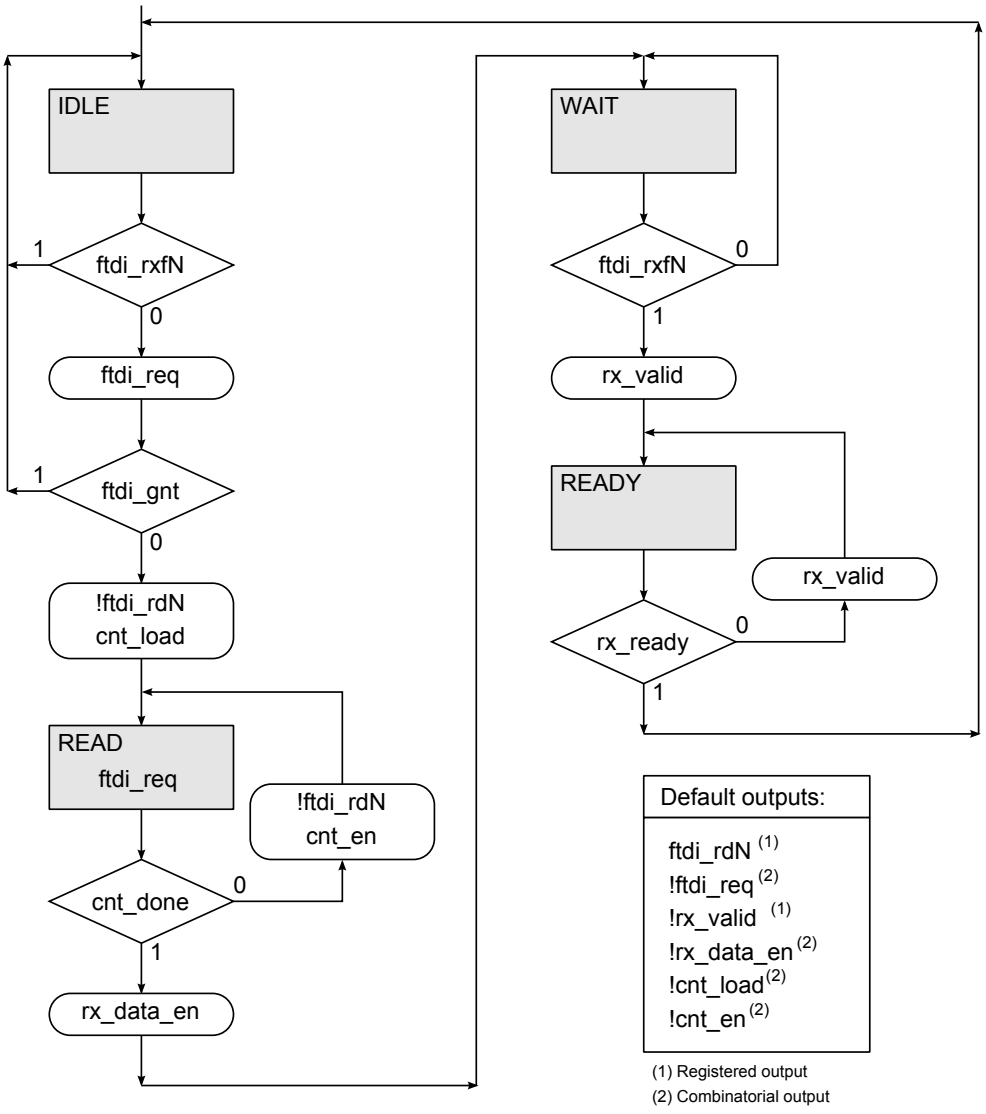


Figure 11: FTDI asynchronous FIFO receiver (host-to-device) interface FSM.

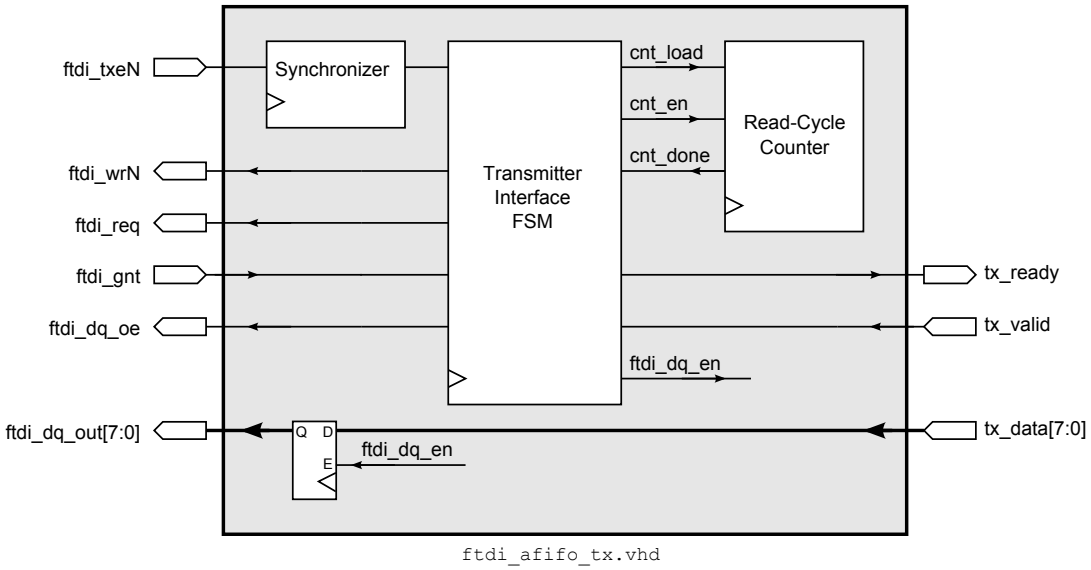


Figure 12: FTDI asynchronous FIFO to Avalon-ST bridge transmitter interface diagram. The write-enable output `ftdi_wrN` is active-high for full-speed devices, and active-low for high-speed devices.

Transmitter (Write) Interface

Figure 12 shows a block diagram of the bridge transmitter interface. The transmitter interface write-enable control is active-high for full-speed USB devices, and active-low for USB high-speed devices. Figures 13 and 15 show the full-speed device write timing and FSM, while Figures 14 and 16 show the high-speed device write timing and FSM.

The transmitter (write) sequence in Figures 13 and 14 proceeds as follows;

- In the `IDLE` state, the transmitter FSM asserts the Avalon-ST sink `tx_ready` signal to indicate that it is ready for data.
- When the Avalon-ST sink `rx_valid` signal asserts to indicate valid data, the FSM asserts the transmit data enable control, `ftdi_dq_en`, to capture the data in the `ftdi_dq_out[7:0]` register, and deasserts the `tx_ready` signal (so that no further Avalon-ST data is accepted).
- The FSM checks the state of the synchronized version of the the FTDI asynchronous output `ftdi_txeN`. If the signal is deasserted, indicating there is space in the transmit FIFO, the FTDI bus request signal, `ftdi_req`, is asserted.
- The FSM transitions to the `READY` state, where when `ftdi_txeN` is deasserted, the FSM asserts the FTDI bus request until the bus grant signal is asserted.
- When `ftdi_txeN` is deasserted and `ftdi_gnt` is asserted, the FSM loads the write-setup count, and transitions to the `SETUP` state.
- In the `SETUP` state, the FSM drives data onto the bus, asserts the counter enable control, and for full-speed devices asserts the active-high write-enable until the counter done (carry-out) signal asserts.
- The FSM then loads the write-hold count, and transitions to the `HOLD` state.

- In the `HOLD` state, the FSM drives data onto the bus, asserts the counter enable control, and for high-speed devices asserts the active-low write-enable until the counter done (carry-out) signal asserts.
- The FSM then transitions to the `WAIT` state, where it waits for the write transaction to complete on the FTDI bus, as indicated by the synchronized version of `ftdi_txeN` deasserting.
- The FSM then transitions to the `IDLE` state, where it asserts `tx_ready`.

The full-speed device write timing in Figure 13 generates an active-high write-cycle pulse of 3 clock periods using a write-setup count of 3 clock periods (a down-counter load value of 2) and a write-hold count of 1 clock period (a down-counter load value of 0). The high-speed device write timing in Figure 14 generates an active-low write-cycle pulse of 3 clock periods using a write-setup count of 1 clock period (a down-counter load value of 0) and a write-hold count of 3 clock periods (a down-counter load value of 3). The write-setup and write-hold counter values are calculated from the clock frequency, write-setup time, and write-hold time generics. The full- and high-speed FSMs are implemented using a single VHDL FSM that uses a generic to determine the active level and assertion timing of the write-enable output.

Figures 13 and 14 shows the write-cycle time as 3 clock periods for both the full- and high- speed timing diagrams. When instantiating the FTDI-to-Avalon-ST bridge, the write-cycle time of the bridge is determined; for full-speed devices by the write-setup generic, and for high-speed devices by the write-hold generic.

State Machine Design

The transmitter (write) interface control FSM shown in Figures 13, 14, 15, and 16 contain the following assumptions;

- The interface clock frequency is sufficient that the synchronized version of `ftdi_txeN` generates a synchronous high pulse.
- The read and write timing generics are set to meet, but not greatly exceed, the timing requirements of the FTDI device.

  If the write-cycle time (set via the write-setup and write-hold generics) is set too large, the pulse on the synchronized version of `ftdi_txeN` can occur while the write counter is still running.

If either of these conditions are not met, then the high pulse on `ftdi_txeN` can be missed, and the FSM will be stuck in the wait state (waiting for `ftdi_txeN` to be high). A testbench would exhibit this failure mechanism.

Designs that require a clock period that is large relative to the `ftdi_txeN` pulse time should use a different form of synchronization for this signal. For example, the `ftdi_txeN` pulse can be synchronized to the FSM clock domain using a pulse synchronizer [13], the pulse synchronizer output can set a flag (the flag represents the fact that `ftdi_txeN` went high), which the FSM exit logic would check is set before exiting. The FSM would clear the flag while in the idle state, and transition from idle only when the flag is clear.

Simulation

The FTDI asynchronous FIFO to Avalon-ST bridge source code contains testbenches that reproduce the timing shown in this section. Figures containing Modelsim simulation waveforms are not included in this document, as the simulations have much more detail than can be adequately reproduced. Section 7.1 contains SignalTap II logic analyzer traces from the BeMicro-SDK interfaced to an FTDI full-speed UM245R module and high-speed UM232H module configured in asynchronous FIFO mode.

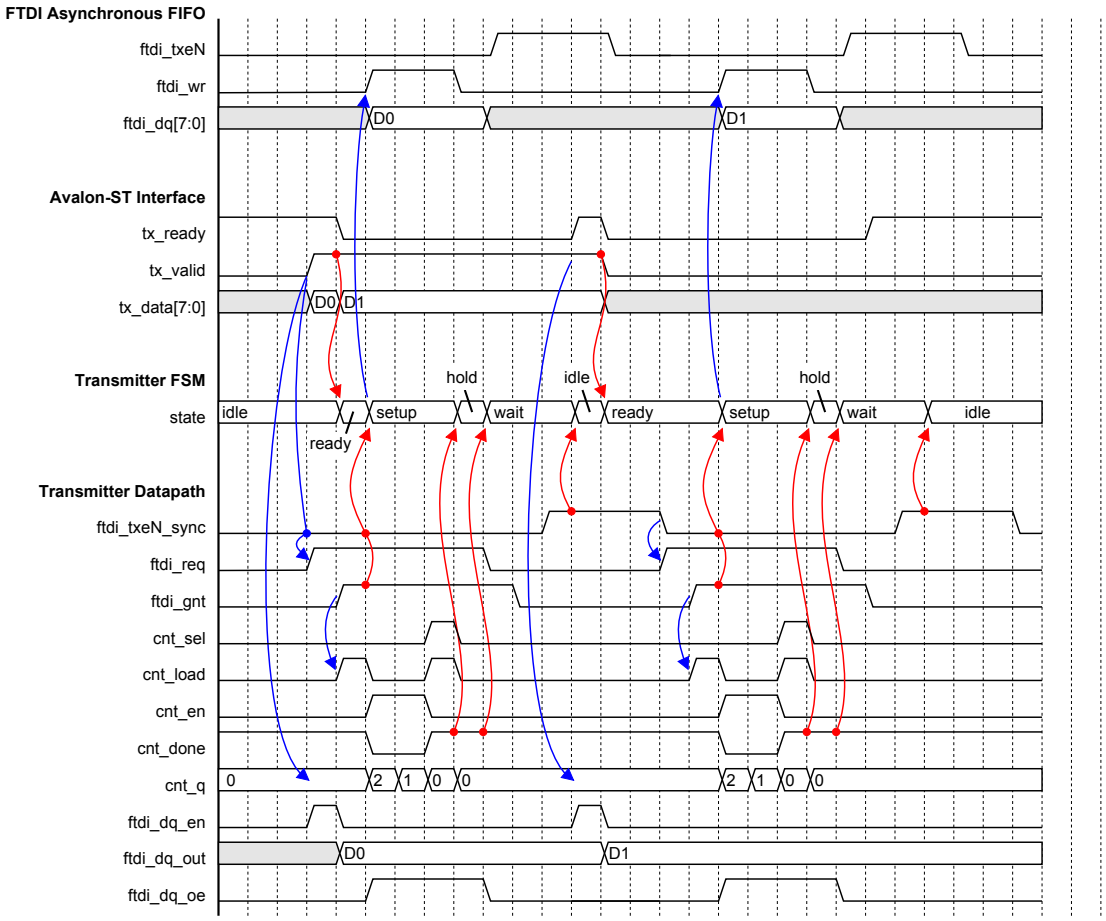


Figure 13: FTDI asynchronous FIFO transmitter (device-to-host) interface timing for full-speed devices (active-high write-enable).

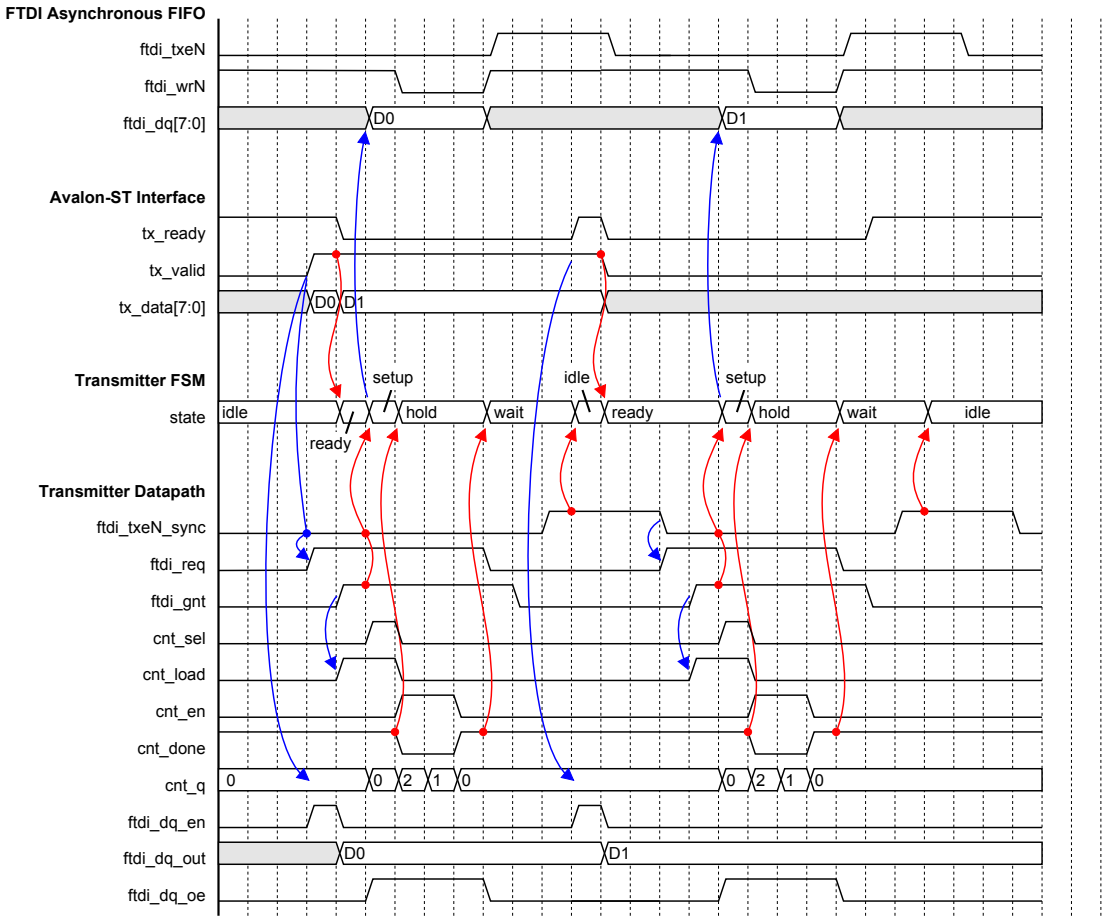


Figure 14: FTDI asynchronous FIFO transmitter (device-to-host) interface timing for high-speed devices (active-low write-enable).

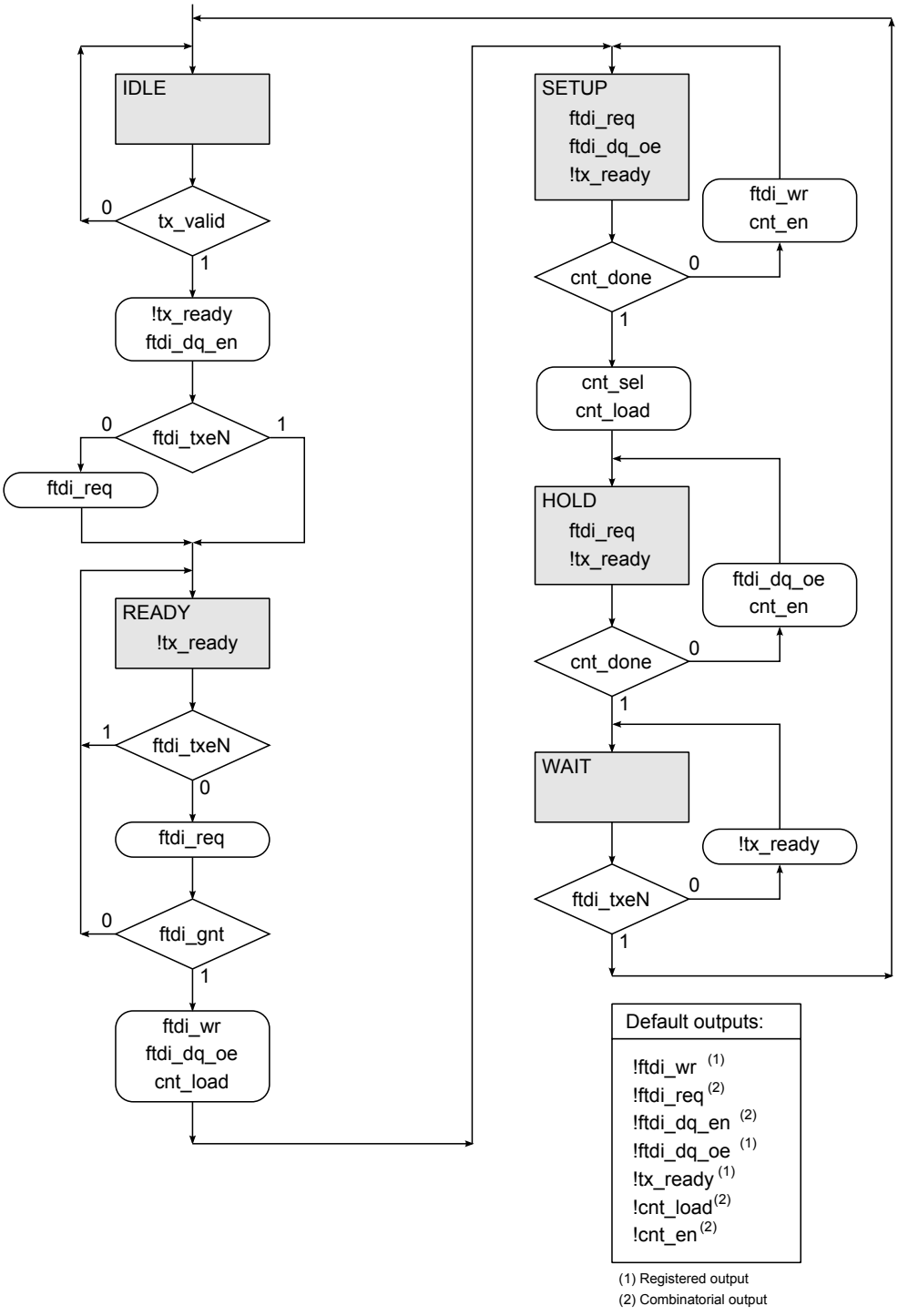


Figure 15: FTDI asynchronous FIFO transmitter (device-to-host) interface FSM for full-speed devices (active-high write-enable).

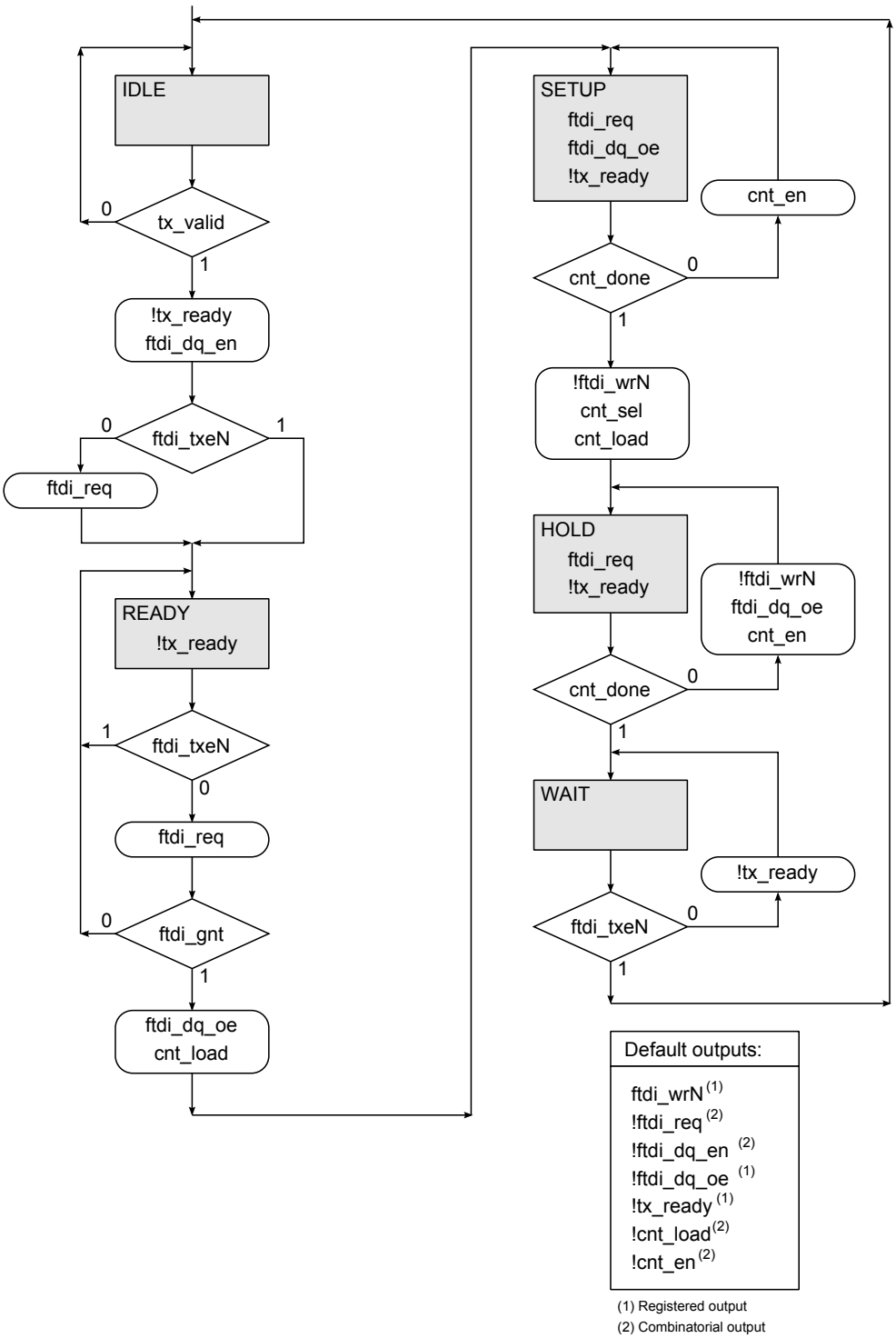


Figure 16: FTDI asynchronous FIFO transmitter (device-to-host) interface FSM for high-speed devices (active-low write-enable).

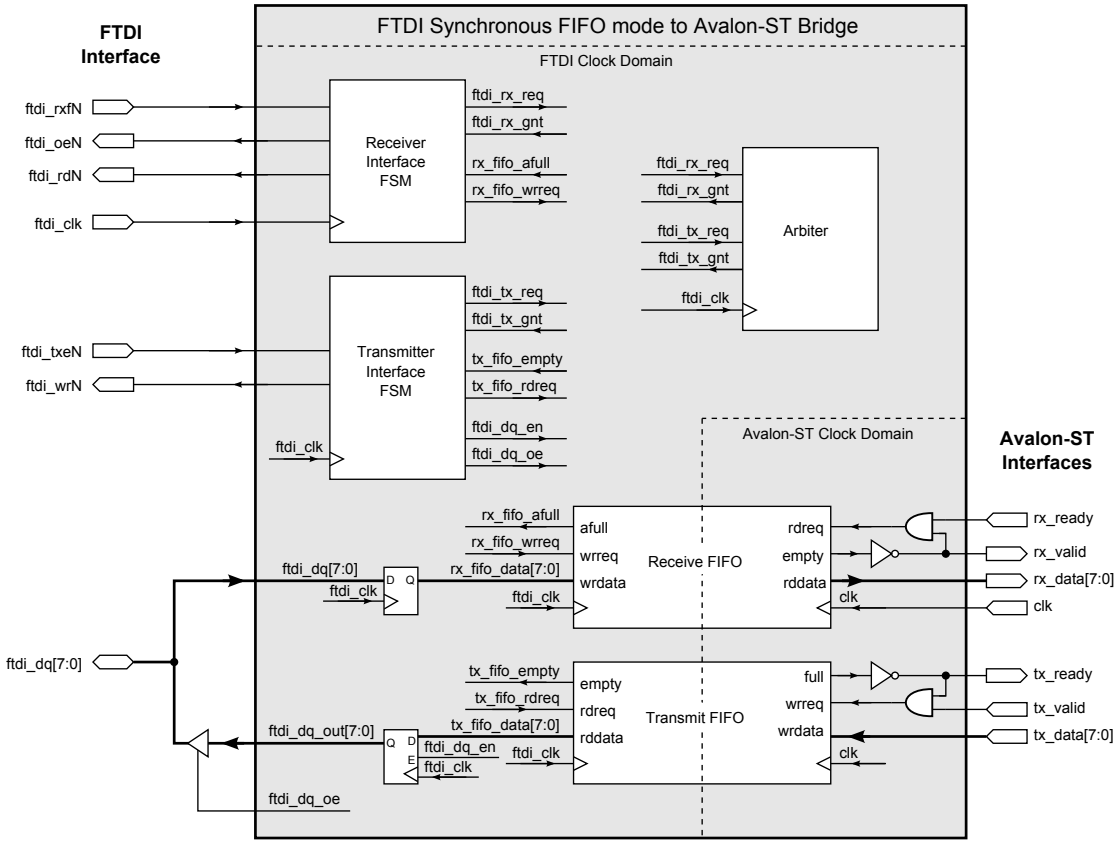


Figure 17: FTDI synchronous FIFO to Avalon-ST bridge diagram.

4.2 Synchronous FIFO to Avalon-ST Bridge

The FTDI *synchronous* FIFO to Avalon-ST bridge is a general purpose interface that converts the FTDI synchronous FIFO host-to-device (receiver) and device-to-host (transmitter) byte-streams into Avalon-ST source and sink byte-streams. Figure 17 shows a block diagram of the bridge component. In synchronous mode, the FTDI device generates a 60MHz clock. The FPGA logic can use that clock or can use an alternative clock source. VHDL generics are used to configure the bridge for single-clock or dual-clock mode, and to configure the depth of the receive and transmit FIFOs.

Figure 17 shows that the bridge design is partitioned into two main interfaces; host-to-device (receiver) and device-to-host (transmitter). As with the asynchronous mode bridge, both interfaces share the common 8-bit FTDI data bus, so a bus arbiter is used.

Receiver (Read) Interface

Figure 17 shows that the receiver interface consists of an FSM, a single- or dual-clock FIFO, and an Avalon-ST source (implemented using combinatorial logic on the FIFO interface). Figure 18 shows receiver timing sequences implemented by the finite state machine (FSM) in Figure 19.

The receiver (read) sequences in Figure 18 proceed as follows;

- The FTDI synchronous output `ftdi_rxfN` asserts low to indicate there is data in the FTDI device receive FIFO. The FSM checks that there is space in the FPGA receive FIFO and if the FIFO is not almost-full, it asserts the FTDI bus arbiter request signal, `ftdi_req`, and waits until the grant signal is asserted.

- Assertion of the grant signal, `ftdi_gnt`, causes the FSM to assert the FTDI output enable signal, `ftdi_oeN`, and transition to the `START` state.

- The `START` state provides a clock period where the FTDI data drivers are enabled. The driver output enable delay is not a clock-to-output delay, but is a delay relative to the input signal `ftdi_oeN`, i.e., the delay to valid data on the FTDI bus is the sum of the FPGA clock-to-output delay for the `ftdi_oeN` output plus the FTDI output-enable to valid data delay. The `START` state ensures that the data is valid on the bus at the end of the first clock period in the `READ` state.

- Data transfers occur on the FTDI bus during clock periods when both `ftdi_rxfN` and `ftdi_rdN` are low. In the `READ` state, the FSM asserts both `ftdi_oeN` and `ftdi_rdN`, and monitors the state of `ftdi_rxfN`. For each clock period that `ftdi_rxfN` is low, data is written to the receive FIFO.

- The FSM remains in the read state as long as `ftdi_rxfN` is asserted and the receive FIFO is not almost-full. If either of these conditions are not true, the FSM transitions to the `END` state.

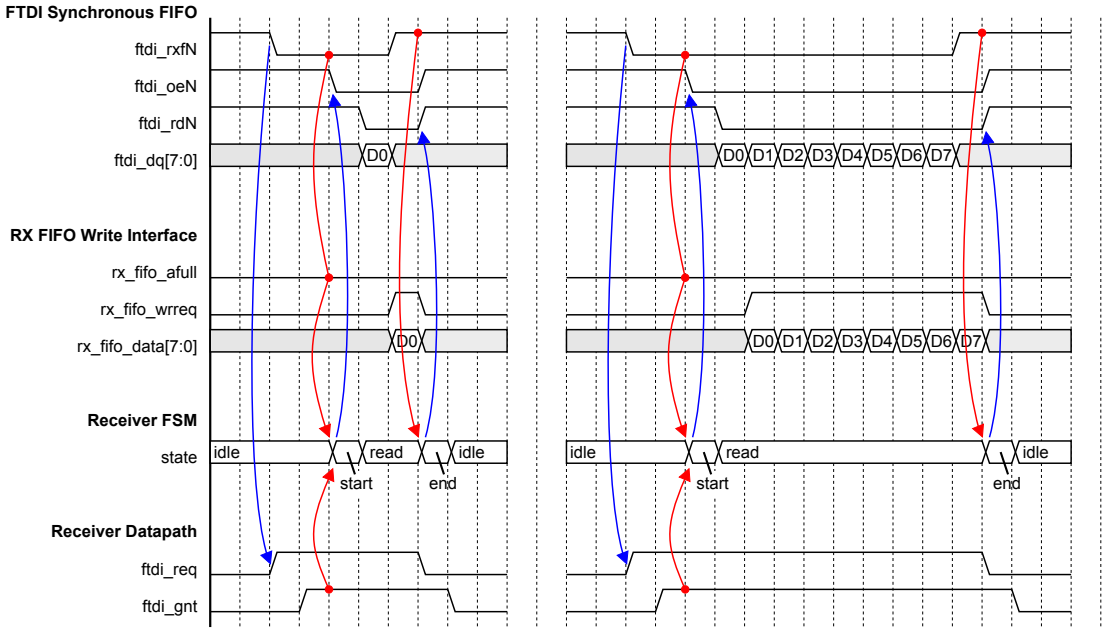
  Figure 18(a) shows a single read. When the FSM transitions to the read state, `ftdi_rxfN` is observed asserted only for the first clock period. The single phase of valid receive data is written to the receive FIFO. In the second clock period in the read state, `ftdi_rxfN` is observed deasserted, so the FSM transitions to the `END` state.

  Figure 18(b) shows a burst read. The timing sequence is essentially identical to that of a single-read, with additional read state data phases with `ftdi_rxfN` asserted indicating valid received data.

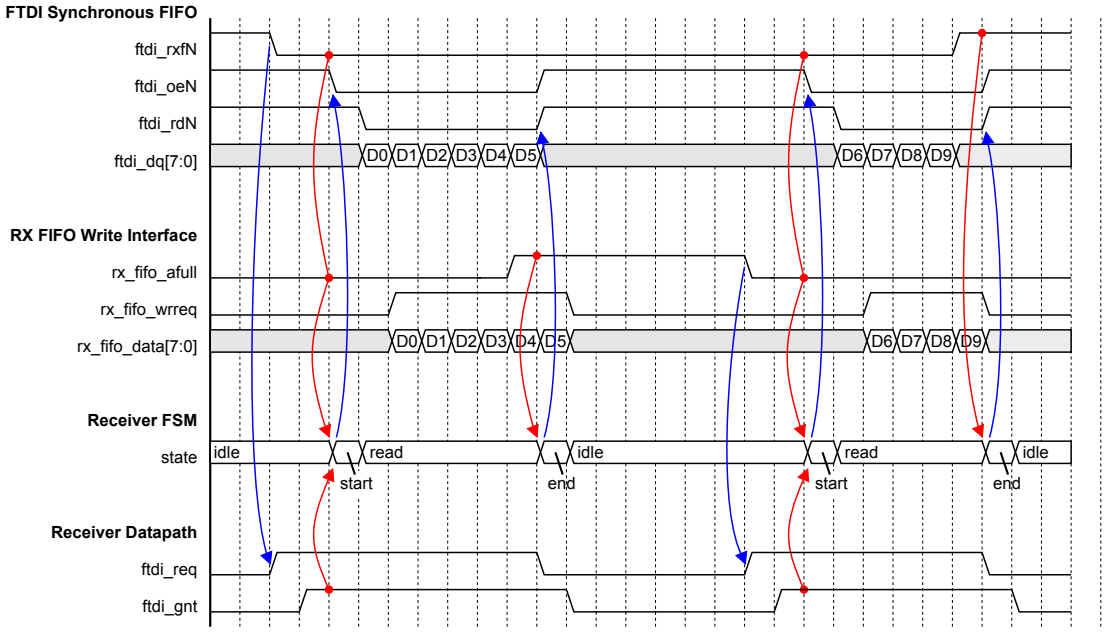
  Figure 18(c) shows a burst read with FPGA-side wait-states. During the burst transaction, the receive FIFO almost-full signal asserts. The FIFO depth at which the almost-full signal asserts is based on the receive data-path pipeline latency; the figure shows that when almost-full asserts, two data phases must be written to the FIFO, hence the almost-full depth is at least two below the FIFO size. When the FIFO almost-full signal asserts, the FSM transitions to the `END` state, and then `IDLE`. This sequence implements the FPGA-side wait-states, since the FSM will not transition from `IDLE` until there is space in the receive FIFO.

- The `END` state provides a single clock period delay which allows the arbiter grant signal, `ftdi_gnt`, a clock period to deassert in response to the FSM deasserting the arbiter request signal, `ftdi_req`. This ensures that the FSM does not erroneously transition from the `IDLE` state based on `ftdi_gnt` being asserted from a previous transaction.

- The FSM then transitions to the `IDLE` state.



(a) Single read (b) Burst read



(c) Burst read with FPGA-side wait-states

Figure 18: FTDI synchronous FIFO receiver (host-to-device) interface timing.

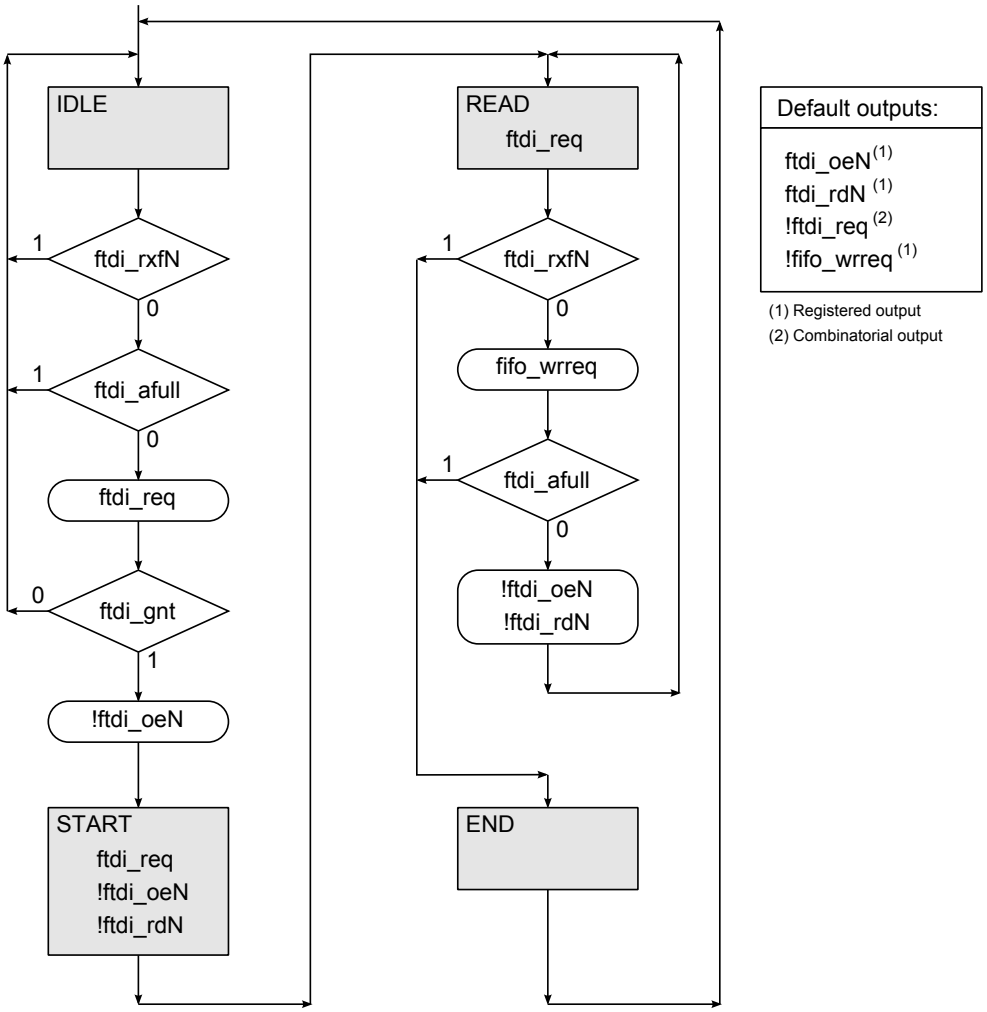


Figure 19: FTDI synchronous FIFO receiver (host-to-device) interface FSM.

Transmitter (Write) Interface

Figure 17 shows that the transmitter interface consists of an FSM, a single- or dual-clock FIFO, and an Avalon-ST sink (implemented using combinatorial logic on the FIFO interface). Figures 20 and 21 show transmitter timing sequences implemented by the finite state machine (FSM) in Figure 22.

The transmitter (write) sequences in Figures 20 and 21 proceed as follows;

- In the `IDLE` state, the transmitter FSM monitors the transmitter FIFO empty signal to determine when there is data in the transmit FIFO.
- When the transmit FIFO empty signal deasserts, the FSM asserts the FTDI bus arbiter request signal, and waits for assertion of the grant signal.
- Assertion of the grant signal causes the FSM to assert the FIFO read-request, `tx_fifo_rdreq`, and to assert the output data register enable control, `ftdi_dq_en`, and transition to the `WRITE` state.
- Data transfers occur on the FTDI bus during clock periods when both `ftdi_txeN` and `ftdi_wrN` are low. In the `WRITE` state, the FSM asserts `ftdi_wrN`, and monitors the state of `ftdi_txeN`. For each clock period that `ftdi_txeN` is low, data is successfully transferred to the FTDI device.
- The FSM remains in the write state as long as `ftdi_txeN` is deasserted and the transmit FIFO has data. If either of these conditions are not true, the FSM transitions to either the `END` or `WAIT` states, as follows.

  Figure 20(a) shows a single write. When the FSM transitions to the write state, `ftdi_txeN` is observed deasserted at the end of the first clock period, indicating that the write data was accepted by the FTDI device. The FSM then transitions to the `END` state.

  Figure 20(b) shows a burst write. During the burst, `ftdi_txeN` is observed deasserted at the end of each write clock period, indicating that each write data phase was accepted by the FTDI device. The FSM then transitions to the `END` state.

  Figure 21(a) shows a burst write with FTDI-side wait-states. During the burst transaction, `ftdi_txeN` deasserts indicating that the last write data phase *was not* accepted by the FTDI device. This means that the transmit data-path must re-attempt the write; the transmit data-path includes the output data register, `ftdi_dq_out[7:0]`, to handle this situation. The output data register holds the write data until it is accepted by the FTDI device (`ftdi_dq_en` is combinatorially generated based on `ftdi_txeN`).

  When `ftdi_txeN` deasserts during the write state, the FSM transitions to the `WAIT` state. This state is necessary as the transition from `WAIT` to `WRITE` is different than the transition from `IDLE` to `WRITE`, i.e., in the FSM in Figure 22, the `WAIT` state logic does not check the FIFO empty flag or generate a FIFO read-request on the transition to the `WRITE` state, since the output data register already contains valid write data.

  Figure 21(a) shows that once `ftdi_txeN` asserts to indicate the FTDI device is ready for more write data, the FSM re-arbitrates for the FTDI bus, transitions to the `WRITE` state, and asserts `ftdi_wrN` to re-attempt the write of the last data phase. Note that during the transition from `WAIT` to `WRITE` the transmit FIFO is *not* read from, since the output data register already contains valid write data.

  The three burst sequences in Figure 21 show that the state of the transmit FIFO is of no consequence to the wait-state logic. The wait-state sequence is solely determined by the state of `ftdi_txeN`.

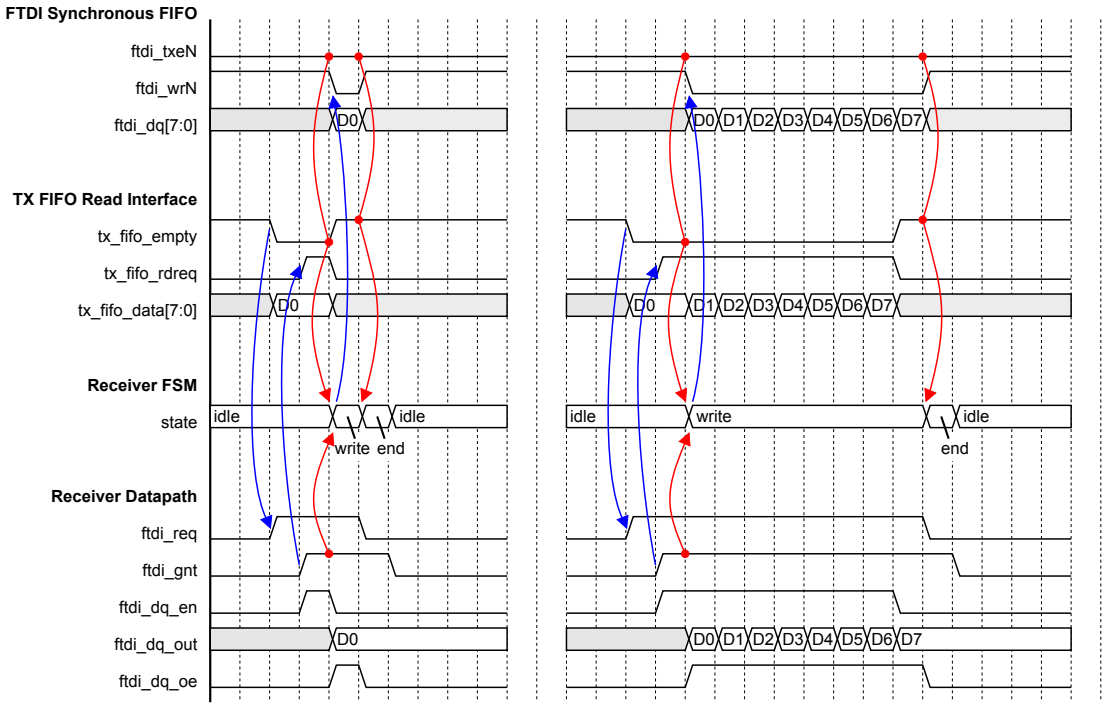
Figure 21(b) shows `ftdi_txeN` deasserting during the same clock as the FIFO emptying, i.e., the last data read from the FIFO was not accepted by the FTDI device. The timing sequence shows how the write is re-attempted once `ftdi_txeN` asserts.

Figure 21(c) shows `ftdi_txeN` deasserting during the same clock as the FIFO emptying, i.e., the last data read from the FIFO was not accepted by the FTDI device. However, relative to Figure 21(b), during the wait-states, more data arrives in the transmit FIFO resulting in the FIFO empty signal deasserting. The state of the FIFO empty signal makes no difference to the transition from the `WAIT` to `WRITE` states, it only affects what happens once the FSM returns to the `WRITE` state, eg., in Figure 21(c) the write burst continues with additional data phases.

- The `END` state provides a single clock period delay which allows the arbiter grant signal, `ftdi_gnt`, a clock period to deassert in response to the FSM deasserting the arbiter request signal, `ftdi_req`. This ensures that the FSM does not erroneously transition from the `IDLE` state based on `ftdi_gnt` being asserted from a previous transaction.
- The FSM then transitions to the `IDLE` state.

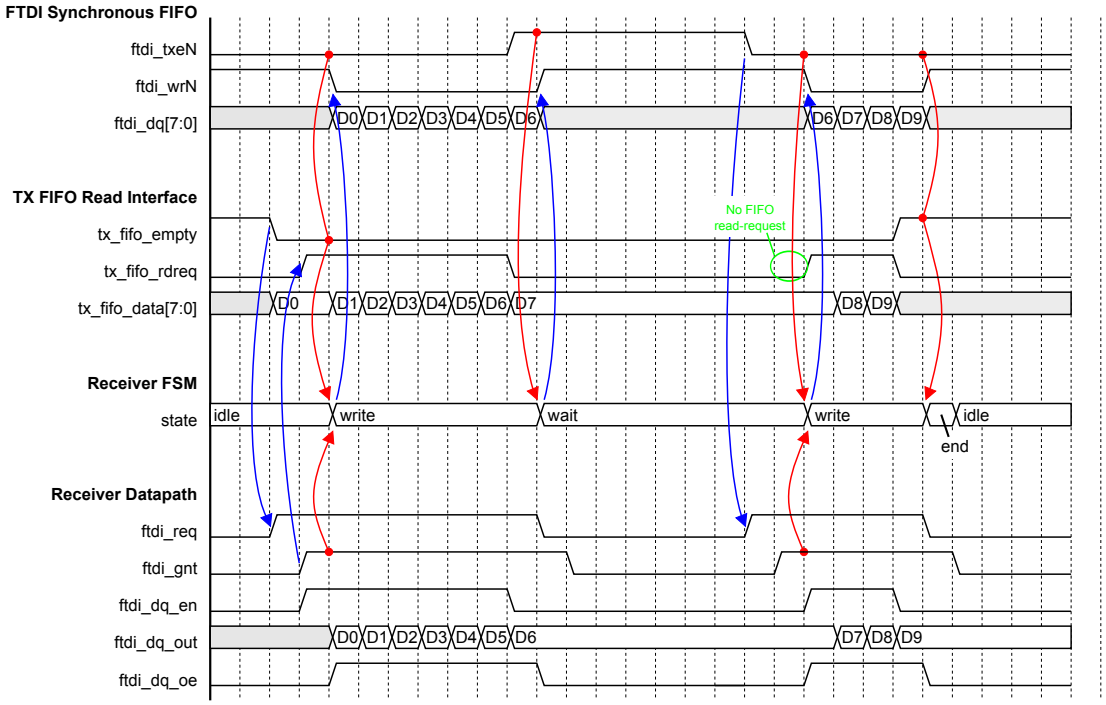
Simulation

The FTDI synchronous FIFO to Avalon-ST bridge source code contains testbenches that reproduce the timing shown in this section. Figures containing Modelsim simulation waveforms are not included in this document, as the simulations have much more detail than can be adequately reproduced. Section 7.1.4 contains SignalTap II logic analyzer traces from the BeMicro-SDK interfaced to an FTDI high-speed UM232H module configured in synchronous FIFO mode.

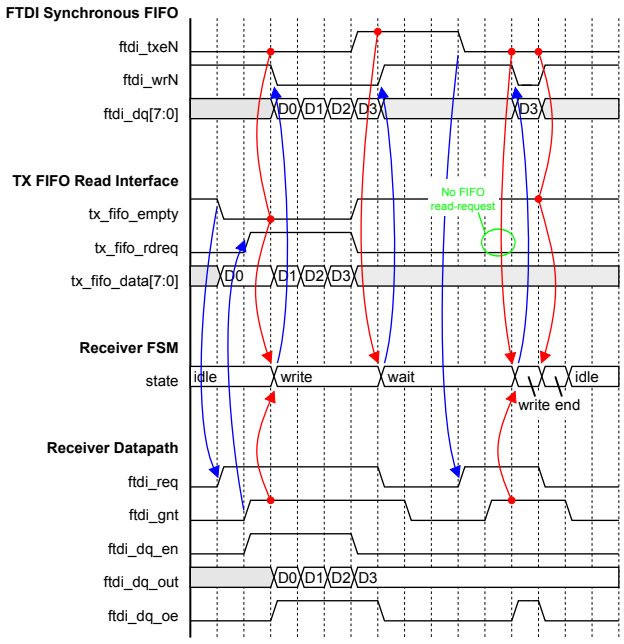


(a) Single write (b) Burst write

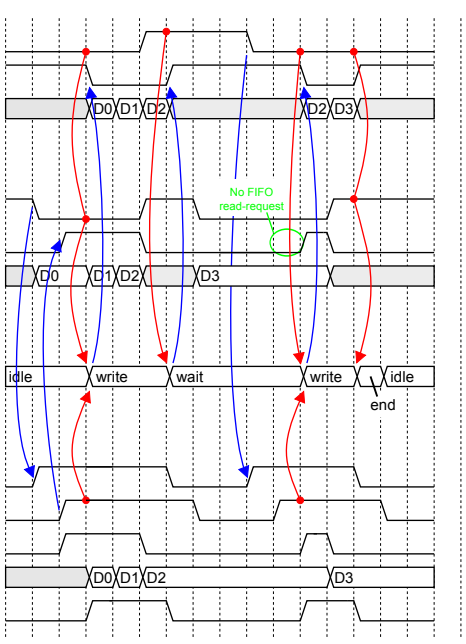
Figure 20: FTDI synchronous FIFO transmitter (device-to-host) interface timing.



(a) Wait-states while FPGA has data



(b) Wait-states on last data phase



(c) More data arrives during wait-states

Figure 21: FTDI synchronous FIFO transmitter (device-to-host) interface timing for burst-writes with FTDI-side wait-states.

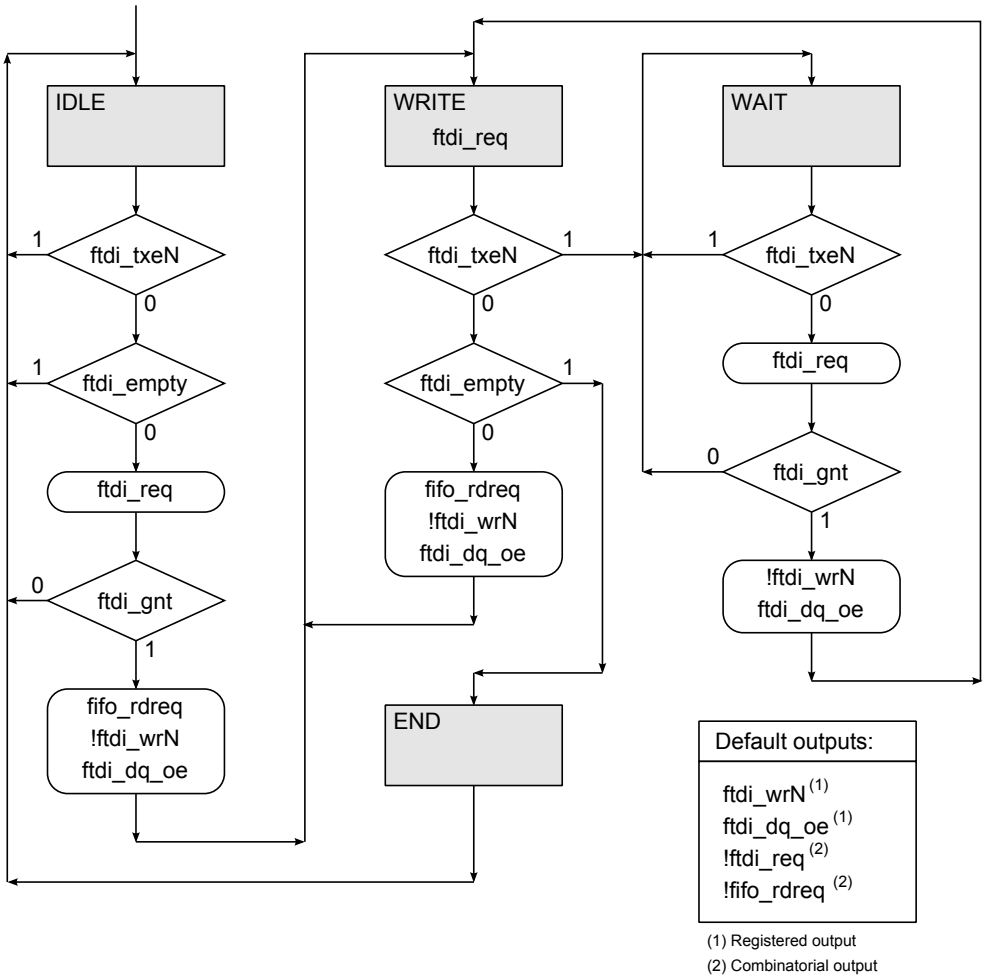


Figure 22: FTDI synchronous FIFO transmitter (device-to-host) interface FSM.

4.3 UART to Avalon-ST Bridge

The UART to Avalon-ST bridge is a general purpose interface that converts asynchronous UART serial byte-streams into Avalon-ST source and sink parallel byte-streams. Figure 23 shows the two components that make up the bridge. The receiver and transmitter data paths are completely independent, so the bridge does not require an arbiter.

Receiver Interface

Figure 23(a) shows a block diagram of the receiver interface. The receiver interface consists of an 8-bit shift-register, a baud-rate counter, a bit counter, and a control FSM. The receiver control FSM is shown in Figure 24.

Reception of a character proceeds as follows;

- The UART receive signal is synchronized to the FPGA clock, and the synchronized signal routes to the FSM and shift-register inputs.
- In the `IDLE` state, the FSM waits for the UART start bit. When the start bit is detected, the baud-rate counter is loaded with a count equal to half the bit period, and the FSM transitions to the `START` state.

  The UART baud-rate (bit-period) is determined by generics, and is fixed at the time of synthesis.
- The `START` state is used to check for the UART serial data start bit. The baud-rate counter is loaded with half the bit period count, so that the logic level of the receive signal can be checked at the *center* of the bit period. If the receive signal is low, the FSM loads the baud-rate counter with a full bit period, loads the bit counter, and transitions to the `BUSY` state. If the receive signal is high, then the FSM transitions to the `IDLE` state.
- In the `BUSY` state, the FSM enables the baud-rate counter for each bit in the 8-bit serial byte. Once the bit counter indicates that all bits have been received, the FSM transitions to the `STOP` state.
- The `STOP` state is used to check for the UART serial data stop bit. The stop bit is checked at the center of the bit period. If the receive signal is high, then the FSM asserts the Avalon-ST data valid signal, `tx_valid`, and transitions to the `READY` state. If the receive signal is low, then the FSM transitions to the `IDLE` state.
- The `READY` state is used to check the Avalon-ST ready handshake, `tx_ready`. When the Avalon-ST sink device acknowledges receipt of the UART byte, the FSM deasserts the valid signal and transitions to the `IDLE` state.

A design objective for the UART-to-Avalon-ST bridge was that it require minimal logic. If the receiver FSM detects an error in data framing (stop and start bits) the data is ignored. The UART receiver does not implement flow control. The data output to the Avalon-ST interface is the content of the shift-register. It is assumed that the Avalon-ST sink receiving the data will acknowledge the data within half a bit period, i.e., before the FSM needs to detect the next start bit.

The lack of flow control is not generally a problem. In a typically application, the FPGA clock is many times faster than the baud-rate, eg., the BeMicro has a 16MHz oscillator and an FT2232C with a serial interface to the second channel, with the serial interface configured for 115200bps, there are 139 16MHz clock periods per UART bit period.

The Avalon-ST processing time can be increased from half a bit period, to 10 bit periods, with the addition of a parallel output data register. With the addition of this register, the FSM would be modified to load the byte into the parallel data register and set the `tx_valid` bit at the end of the `STOP` state. The `tx_valid` bit would then be cleared when `tx_ready` asserts.

Transmitter Interface

Figure 23(b) shows a block diagram of the transmitter interface. The transmitter interface consists of a 9-bit shift-register, a baud-rate counter, a bit counter, and a control FSM. The transmitter control FSM is shown in Figure 25.

Transmission of a character proceeds as follows;

- In the `IDLE` state, the transmitter FSM asserts the Avalon-ST ready signal, `tx_ready`, and waits for valid data. When `tx_valid` asserts, the FSM deasserts the ready signal, loads the baud-rate counter, the shift-register, and transitions to the `START` state.

  The shift-register is loaded with the 8-bit data and a zero in the LSB. The zero is used to generate the start bit. During shift-register shifts, a one is shifted into the MSB. The one is used to generate the stop bit.

  The UART buad-rate (bit-period) is determined by generics, and is fixed at the time of synthesis.

- The `START` state is used to generate the UART serial data start bit. Once the bit-counter expires, the FSM reloads the the bit-counter, loads the bit counter, enables the shift-register, and transitions to the `BUSY` state.

- In the `BUSY` state, the FSM enables the baud-rate counter and shift-register for each bit in the 8-bit serial byte. Once the bit counter indicates that all bits have been transmitted, the FSM transitions to the `STOP` state.

- The `STOP` state is used to generate the UART serial data stop bit. Once the bit-counter expires, the FSM transitions to the `IDLE` state, and reasserts the Avalon-ST ready signal.

The transmitter interface implements flow control, i.e., each byte received on the Avalon-ST interface is serialized before the next byte is acknowledged.

Baud-rate Calculations

The generics `CLK_FREQUENCY` and `BAUD_RATE` are used to configure the receiver and transmitter baud-rate counter load values as follows;

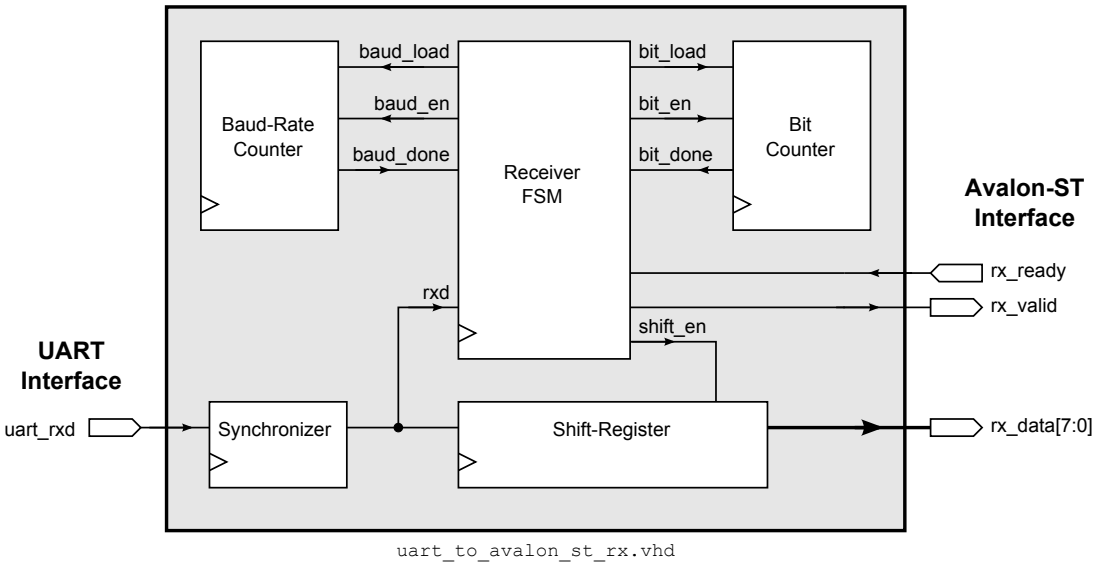
- **Receiver**

```
BAUD_COUNT = integer(floor(CLK_FREQUENCY/BAUD_RATE)) + 1
```

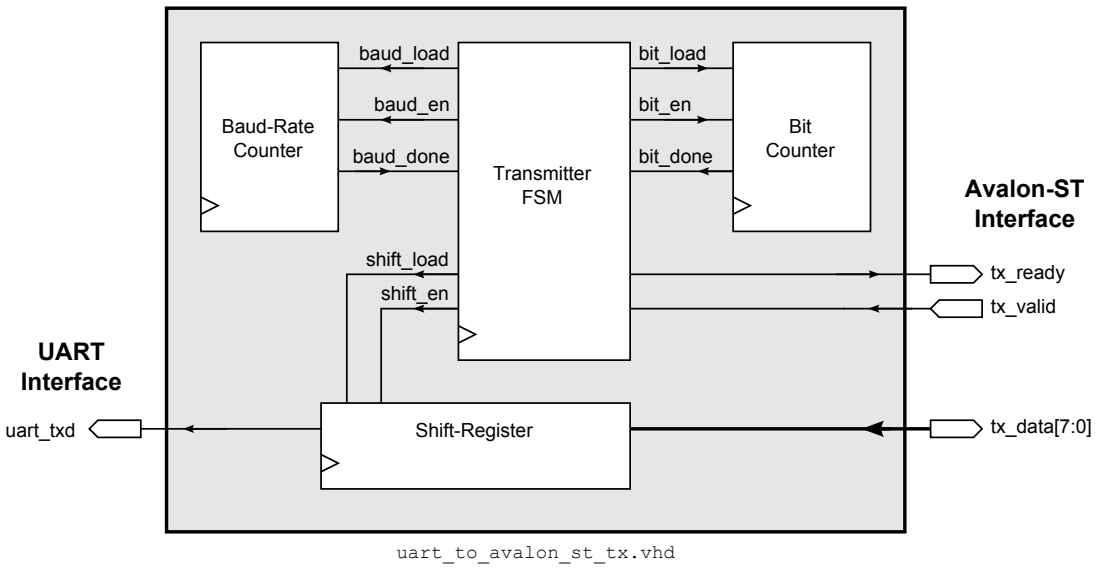
- **Transmitter**

```
BAUD_COUNT = integer(floor(CLK_FREQUENCY/BAUD_RATE))
```

The receiver bit period count is one clock longer than the transmitter, so that the transmitter baud-rate is slightly faster than the receiver baud-rate. This ensures that in an Avalon-ST loopback configuration, the transmitter Avalon-ST sink interface is always ready for receiver Avalon-ST source data, and there is never any need for wait-states (flow control) on the Avalon-ST interface. The hardware example in Section 7.2 contains SignalTap II logic analyzer traces that show the Avalon-ST interface waveforms.



(a) Receiver interface



(b) Transmitter interface

Figure 23: UART to Avalon-ST bridge receiver and transmitter interfaces.

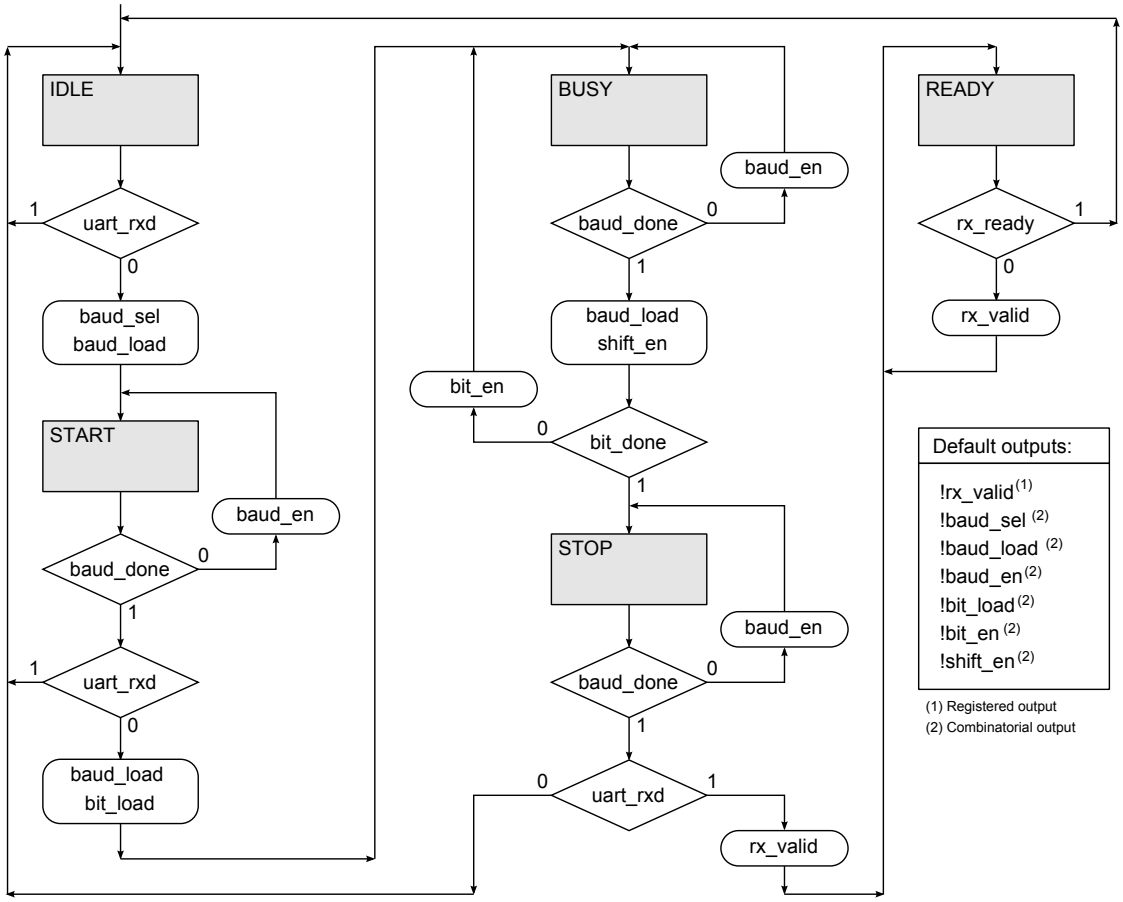


Figure 24: UART to Avalon-ST bridge receiver FSM.

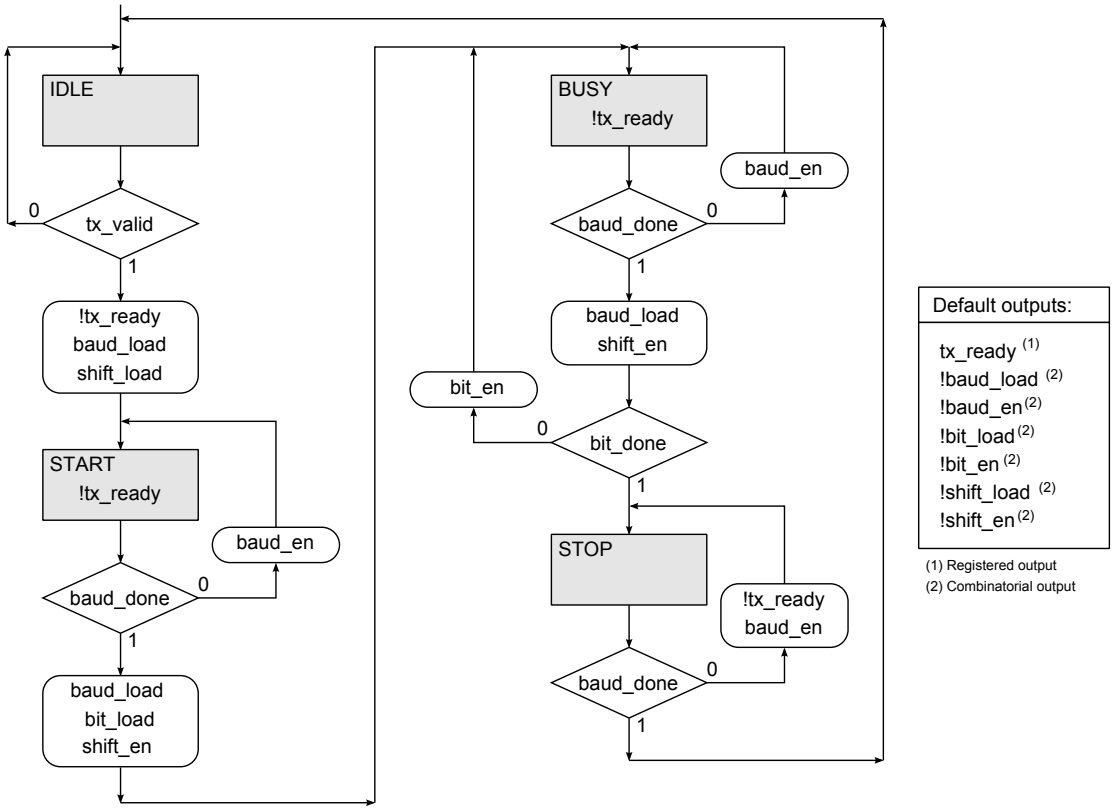


Figure 25: UART to Avalon-ST bridge transmitter FSM.

5 Avalon-MM Bridges

The Altera *Avalon Interface Specifications* [1] defines protocols for implementing streaming (Avalon-ST) and memory-mapped (Avalon-MM) interfaces. This section describes components that map the FTDI serial and parallel interfaces into Avalon-MM interfaces.

5.1 Avalon-ST to Avalon-MM Bridge

5.1.1 ASCII Mode Protocol

5.1.2 Binary Mode Protocol

5.2 FTDI FIFO mode to Avalon-MM Bridge

The FTDI FIFO mode to Avalon-MM consists of the FTDI FIFO mode to Avalon-ST bridge connected to the Avalon-ST to Avalon-MM bridge. Generics on the bridge determine the FIFO FIFO mode (asynchronous full- or high-speed, or synchronous mode), and the Avalon-ST byte-stream protocol.

5.3 UART to Avalon-MM Bridge

The UART to Avalon-MM consists of the UART to Avalon-ST bridge connected to the Avalon-ST to Avalon-MM bridge. Generics on the bridge determine the Avalon-ST byte-stream protocol.

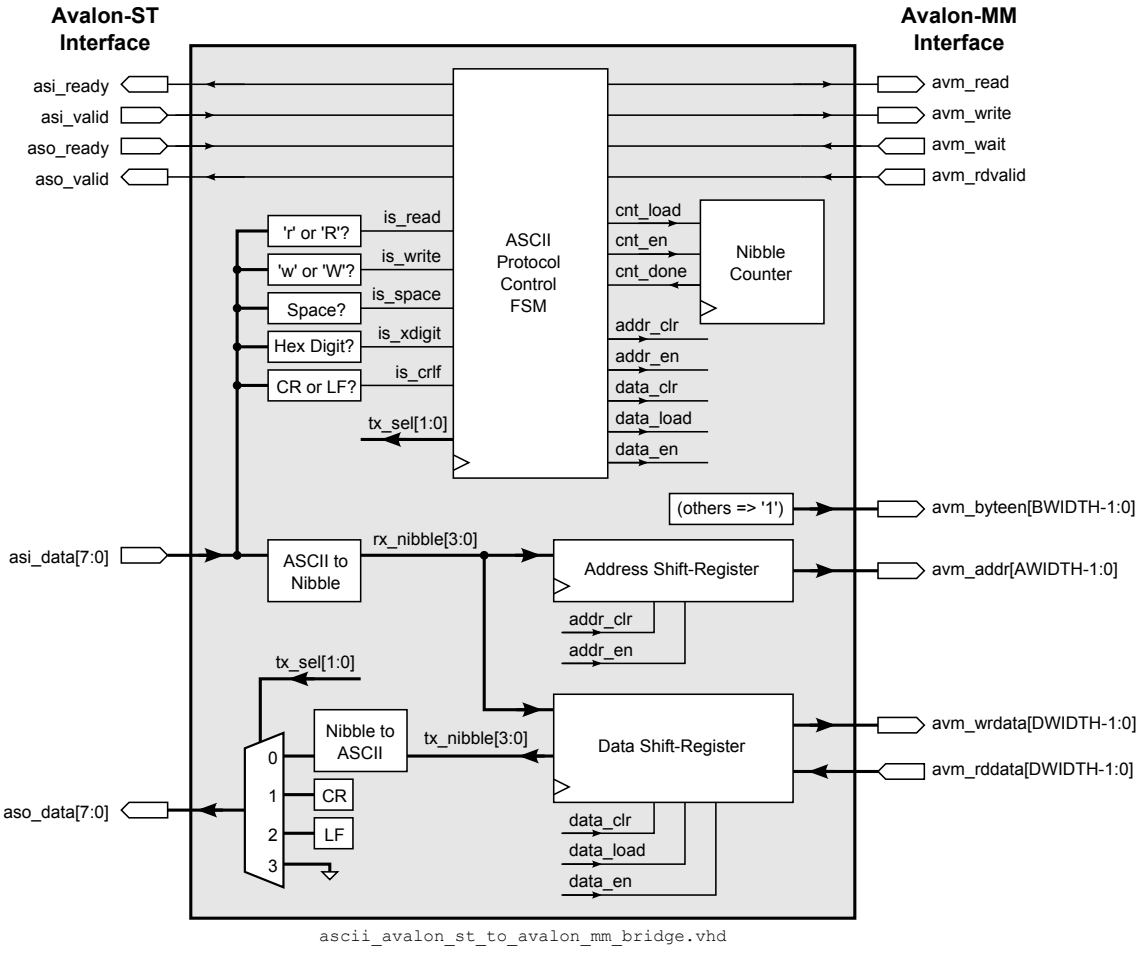


Figure 26: Avalon-ST to Avalon-MM Bridge (ASCII protocol) block diagram.

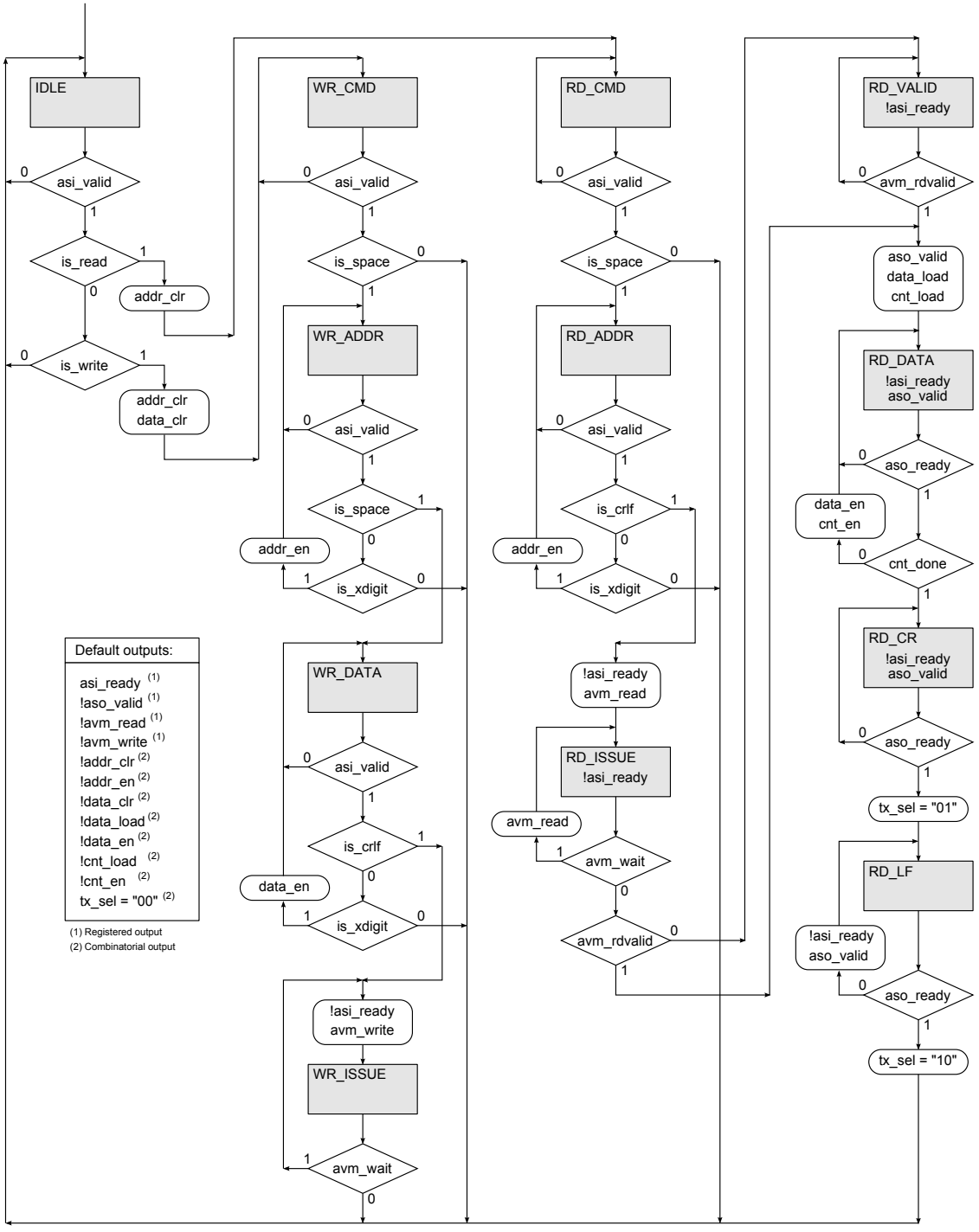
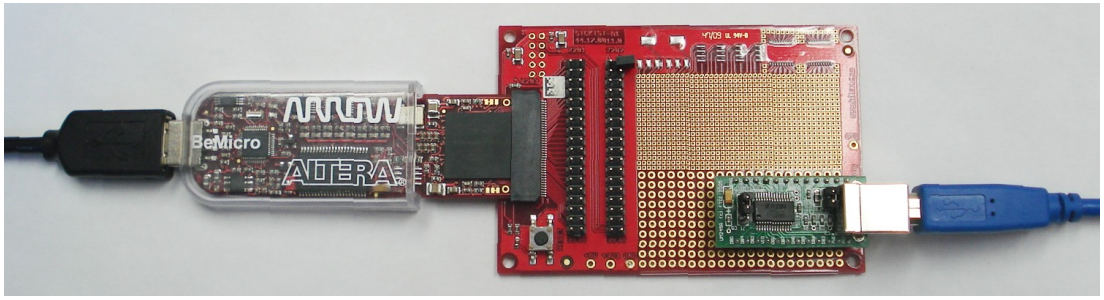
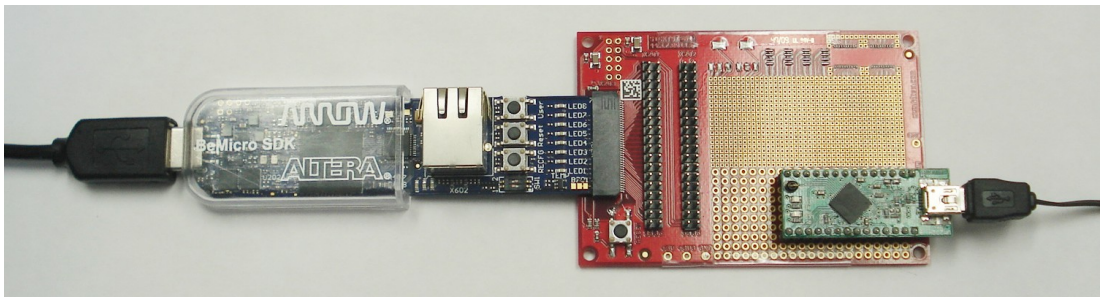


Figure 27: Avalon-ST to Avalon-MM Bridge (ASCII protocol) FSM.

6 Direct Access Programming Interface



(a) Arrow BeMicro (Cyclone III) interfaced to an FTDI UM245R Module



(a) Arrow BeMicro-SDK (Cyclone IV) interfaced to an FTDI UM232H Module

Figure 28: FPGA and FTDI development hardware.

7 Example Designs

This section describes example hardware designs containing the FTDI to Avalon bridge components. Hardware implementations are an important part of component design and verification, as synthesis of a real-world design requires the component VHDL, along with synthesis and timing constraints files. Synthesis of the bridge components into several different hardware targets (different FPGA families and generations) provides real-world resource requirements, and timing margin. A hardware implementation also allows system-level performance measurements, eg., USB data transfer performance from an application (program) running on the USB host computer.

The results in this section were obtained using the hardware shown in Figure 28. The Arrow *BeMicro* (Cyclone III) and Arrow *BeMicro-SDK* (Cyclone IV) FPGA boards were used primarily for two reasons; Cyclone series FPGAs support Altera's SignalTap II logic analyzer interface, and the boards were easily interfaced to FTDI's UM245R and UM232H modules (via BeMicro prototyping boards). Additional hardware used to verify the bridge components were the FTDI *Morph-IC* (which contains an Altera ACEX 1K) and the Altera *MAX II PCI Development Kit*. The ACEX and MAX II devices do not support the SignalTap II logic analyzer, so hardware verification was performed using an oscilloscope and external logic analyzer.

7.1 FTDI FIFO to Avalon-ST Bridge Loopback

Designs were created for the BeMicro and BeMicro-SDK boards to test the FTDI asynchronous and synchronous FIFO to Avalon-ST bridge. The example designs consist of;

- A top-level design file specific to the BeMicro or BeMicro-SDK.
- An `ftdi_fifo_to_avalon_st_bridge` component. This component is a wrapper around the asynchronous and synchronous FIFO bridges, with a generic to select the operating mode.
- Avalon-ST loopback logic. The loopback logic is controlled by a generic that selects either; a direct loopback of the bridge Avalon-ST source and sink interfaces, or an Avalon-ST block FIFO is inserted between the bridge Avalon-ST source and sink interfaces.

  The *Avalon-ST block FIFO*, `avalon_st_block_fifo`, contains a single-clock FIFO and a control FSM that sinks Avalon-ST data until the FIFO is full, and then sources Avalon-ST data until the FIFO is empty. The *block* transfer nature of the Avalon-ST interfaces allows FTDI burst transfers to be traced using the SignalTap II logic analyzer.

The following sections present results for the asynchronous FIFO mode bridge implemented in the BeMicro-SDK with the UM245R and UM232H FTDI modules.

7.1.1 TimeQuest Timing Analysis

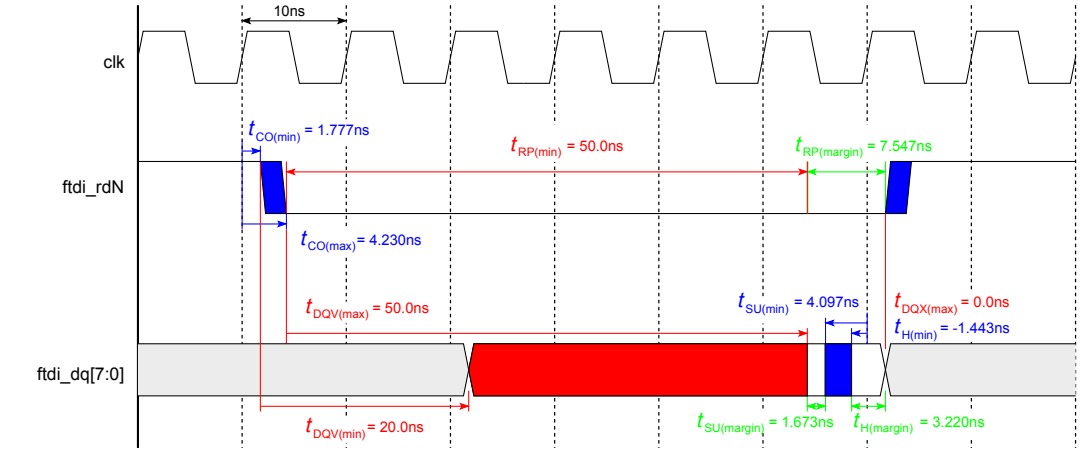
Synthesis of the example design requires an Altera TimeQuest constraints script to constrain the FTDI interface. TimeQuest reports timing margin (also known as timing slack). A TimeQuest analysis script was written that extracts the individual delay components so that input setup and hold, and output clock-to-output delay minimum and maximum values *with respect to the FPGA clock and I/O pins* could be analyzed. Once a design passes timing analysis (has adequate timing margin), hardware tests can be performed.

Figures 29 and 30 show the BeMicro-SDK timing results for asynchronous FIFO mode interfacing to the UM245R full-speed and UM232H high-speed USB modules. The timing margins reported by TimeQuest are the green values shown in the figures. The blue parameters were extracted by using a TimeQuest post-processing script, while the red parameters are the FTDI device data sheet timing parameters per Tables 2 and 3. The synthesis scripts for the BeMicro-SDK design include a script that creates a Quartus report window that displays the timing parameters and timing margins. The timing parameters shown in the figures are highly reproducible; Quartus 11.1sp1 produces the same results each time the design is synthesized.

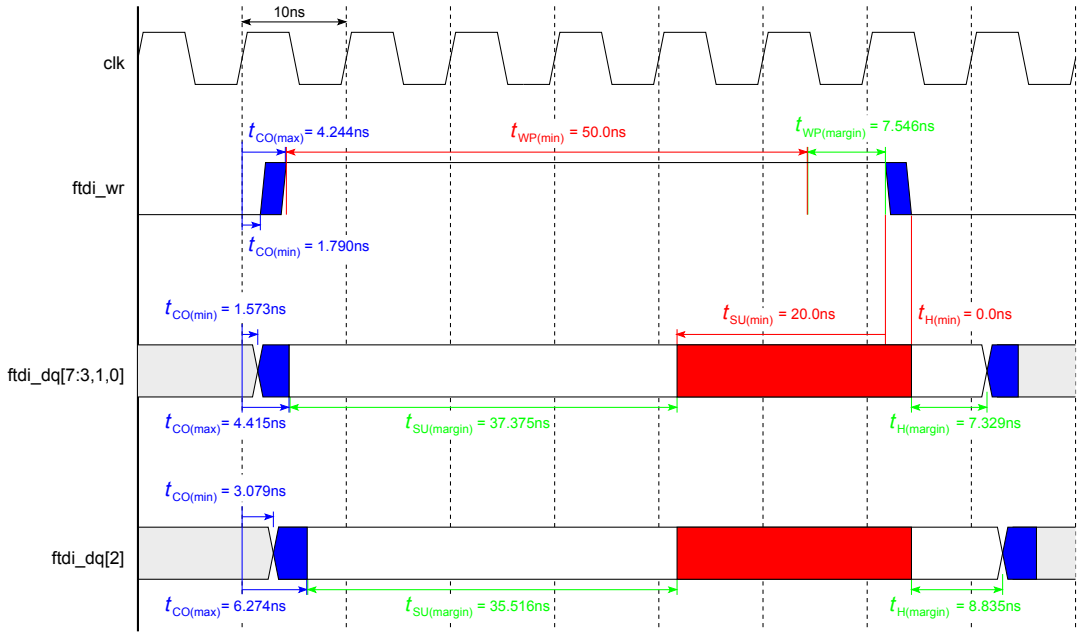
The data bus clock-to-output timing in Figures 29(b) and 30(b) show an interesting *potential* timing issue; the data bus bit `ftdi_dq[2]` has a much longer clock-to-output delay than the other data bus bits (or output control signals). The Quartus pin planner showed the problem; `ftdi_dq[2]` connects to an FPGA pin that is a dual-purpose VREF pin. VREF pins have additional capacitance, which results in a larger clock-to-output delay. When designing a high-performance interface, where a minimal clock-to-output delay is critical to meeting the timing budget, it is important to *avoid* using dual-purpose VREF pins for outputs or bidirectional signals. This example design could have moved the `ftdi_dq[2]` to another pin on the 80-pin expansion connector, however, the assignment was kept, as it helps demonstrate this potential timing issue.

Figure 30(b) shows that the `ftdi_dq[2]` has very little timing margin. In fact, the design fails to meet timing (by 0.149ns) if the data path register shown in Figure 12 and the associated output-enable register are not placed in the I/O element (IOE)[1]. An alternative solution to using the IOE is to increase the write-setup time generic so that the FSM drives the write-data for two clocks prior to the falling-edge of the write signal.

[1] By setting the `FAST_OUTPUT_REGISTER` and `FAST_OUTPUT_ENABLE_REGISTER` synthesis constraints to `ON`.

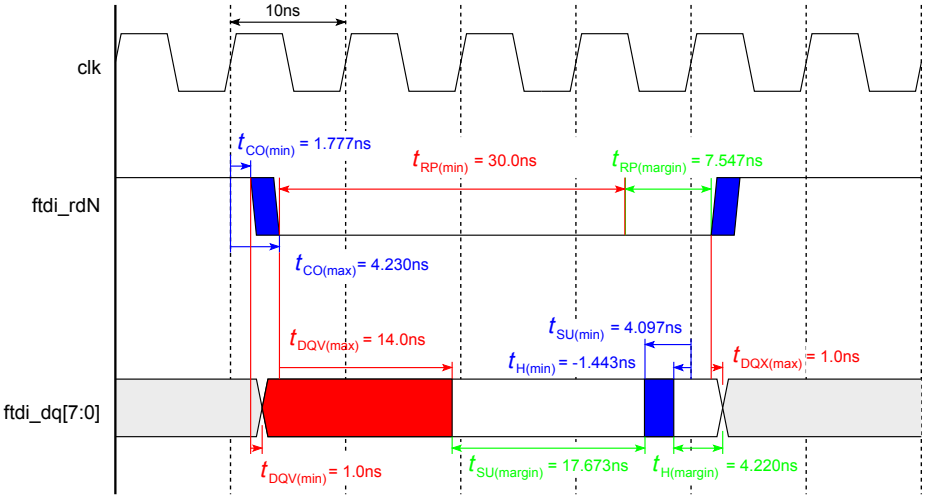


(a) Receiver (read) timing.

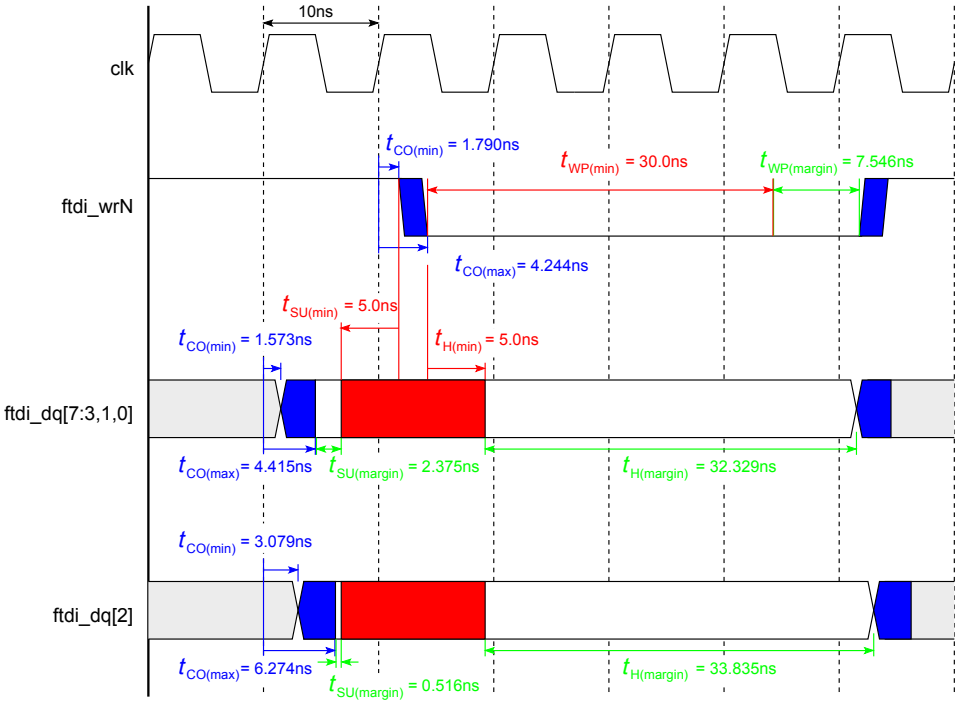


(b) Transmitter (write) timing.

Figure 29: BeMicro-SDK plus UM245R asynchronous FIFO mode timing.



(a) Receiver (read) timing.



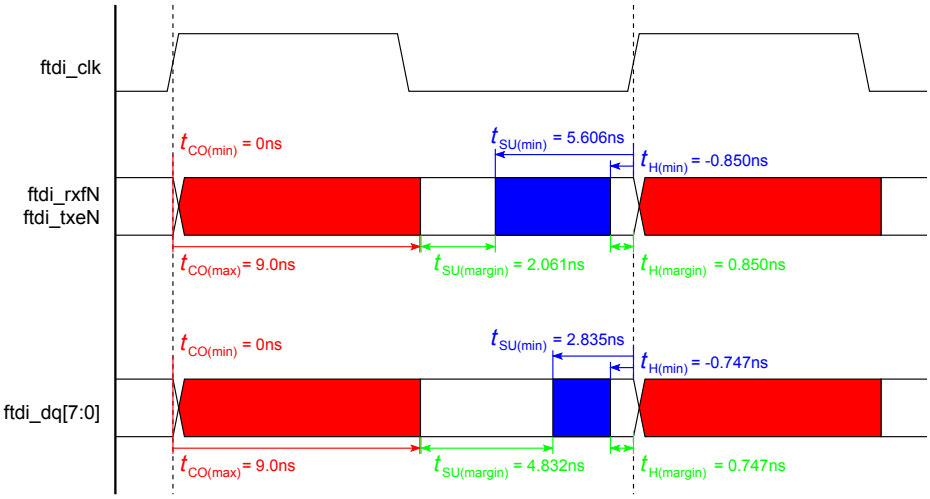
(b) Transmitter (write) timing.

Figure 30: BeMicro-SDK plus UM232H asynchronous FIFO mode timing.

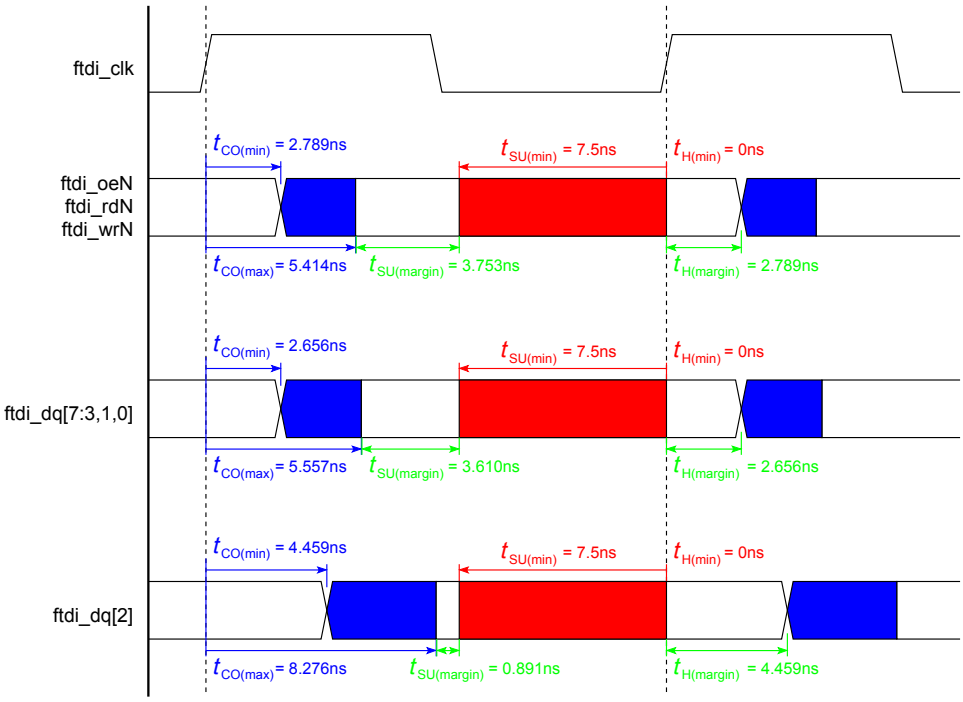
Figure 31 shows the BeMicro-SDK timing results for synchronous FIFO mode interfacing to the UM232H high-speed USB module. For synchronous FPGA interfaces, the timing parameters of interest are; the FPGA input setup-and-hold requirements, and the FPGA output clock-to-output delays. These parameters are not provided in FPGA handbooks; Altera expects the end-user to use TimeQuest to obtain this information.

Figure 31(a) shows the FPGA input timing; the figure groups the control inputs and the data bus inputs and provides the worst-case timing for each group. The control signal input setup-and-hold times have a wider variation than the data bus, due to the fact that the control signals route to multiple destination registers via *combinatorial* paths, whereas the data bus is registered by input registers. The TimeQuest *Report Timing* GUI can be used to show the timing path for each of the input paths from the pins to the input registers; the GUI shows that the input path registers are all located in the FPGA fabric, not in the I/O Elements (IOEs). The design was re-synthesized with an input IOE constraint, however, the constraint was (quietly) ignored, most likely as the IOE constraint conflicts with meeting the input timing constraints.

Figure 31(b) shows the FPGA output timing; the figure groups the control outputs and the data bus outputs and provides the worst-case timing for each group. The data bus bit `ftdi_dq[2]` separated from the other data bus bits, to show the effect of the additional VREF pin capacitance. The control and data bus output clock-to-output delay variation (excluding the VREF bit) is very similar. The extra capacitance of the VREF pin causes the `ftdi_dq[2]` bit to have a larger clock-to-output delay and consequently smaller timing margin. The output timing in Figure 31(b) was obtained using the IOE output registers. A synthesis constraint was *not* required to force the use of these registers, TimeQuest used the IOE registers so that the output timing requirements could be met. The design was re-synthesized with an IOE output register constraint that disallowed the use of the IOE registers. The design failed to meet the output timing requirements; the extra delay on the `ftdi_dq[2]` output caused the signal to fail to meet the FTDI input setup requirement by 0.548ns. If the VREF pin was *not* used as an FTDI bus output, the output timing requirement would have been met.



(a) FTDI output (FPGA input) timing.



(b) FTDI input (FPGA output) timing.

Figure 31: BeMicro-SDK plus UM232H synchronous FIFO mode timing.

7.1.2 UM245R Asynchronous FIFO mode

Figure 32 shows SignalTap II logic analyzer traces for the BeMicro-SDK interfaced to the FTDI UM245R USB Full-Speed device (with active-high write-enable);

- Figure 32(a) Avalon-ST loopback.

  In this example, the bridge Avalon-ST source and sink interfaces are directly connected; note how the traces for `rx_valid` and `rx_ready`, are identical to `tx_valid` and `tx_ready`. The received data, `rx_data[7:0]`, is inverted to create the transmitted data, `tx_data[7:0]`, as this makes it easier to see when the FPGA drives the FTDI data bus.

  The effect of the Avalon-ST loopback can be seen on the FTDI bus traces, where each read from the FTDI bus is immediately followed by a write to the bus.

  The back-to-back reads and writes occur every 500ns, i.e., the read byte-stream is 2MB/s and the write byte-stream is also 2MB/s.

- Figure 32(b) Avalon-ST block FIFO loopback.

  In this example, the bridge Avalon-ST source interface fills the block FIFO, and the Avalon-ST sink interface drains the block FIFO. The block FIFO is configured for a FIFO depth of 8-bytes.

  The presence of the Avalon-ST block FIFO between the bridge Avalon-ST interfaces causes the FTDI interface to issue 8 reads followed by 8 writes. This sequence allows the timing of back-to-back reads (with no writes) and back-to-back writes (with no reads) to be observed.

  The back-to-back reads and writes occur every 500ns, i.e., the read byte-stream is 2MB/s and the write byte-stream is also 2MB/s. The observed data transfer rates are consistent with the data sheet claim of 1MB/s.

  Note that because the individual phases of the back-to-back read followed by back-to-back write traces in Figure 32(b) did not change relative to the read-write timing in Figure 32(a), that the total time in the back-to-back traces in Figure 32(b) is *twice* that of Figure 32(a).

- FTDI device timing parameters.

  Table 2 contains the UM245R timing parameters. Figure 32 shows that the RXF# and TXE# high times observed in hardware are significantly larger than the data sheet minimum values, i.e., 380ns high times versus the data sheet 80ns minimum.

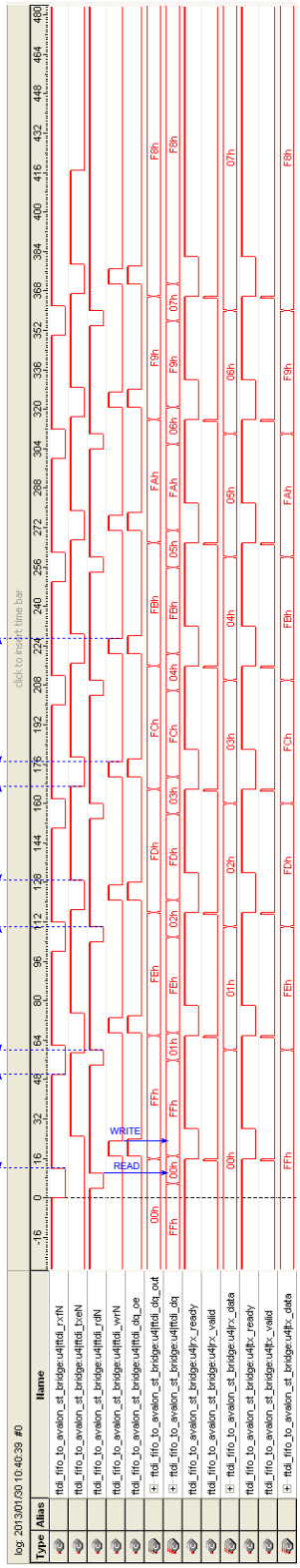
  Hardware tests with the Morph-IC FT2232C showed an RXF# high time of 290ns and a TXE# high time of 490ns. Hardware tests with the MAX II PCI Kit FT245BM showed an RXF# high time of 280ns and a TXE# high time of 520ns.

  The discrepancy between the data sheet RXF# and TXE# high times versus the hardware observations does not affect the bridge design. In fact, it has a beneficial side-effect; because the RXF# high time is much higher than the data sheet value, the read-to-read pre-charge time can never be violated, so there is no need for the bridge receiver FSM to enforce the read-to-read pre-charge time.

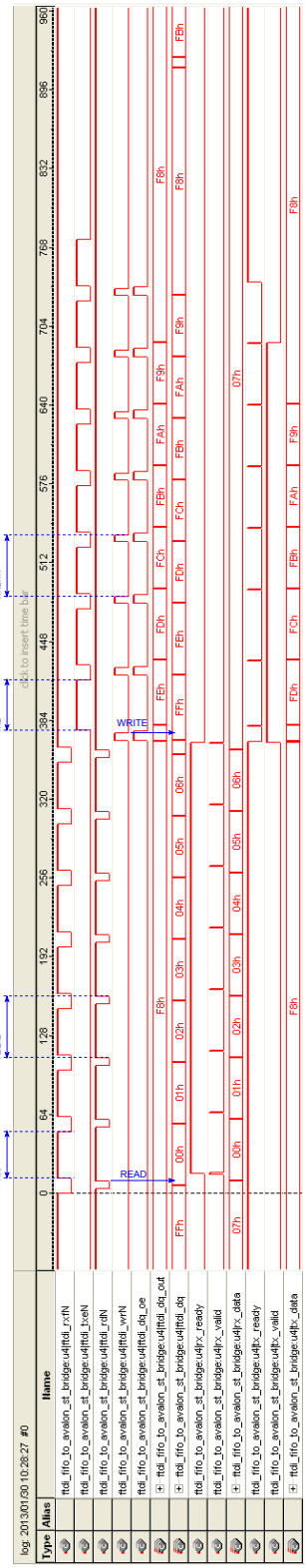
- Cyclone IV Resources.

  The `ftdi_afifo_to_avalon_st_bridge` requires 43 logic cells;

  - `ftdi_arbiter` (3 LCs)
  - `ftdi_afifo_rx` (17 LCs)
  - `ftdi_afifo_tx` (23 LCs)



(a) 8-bytes Avalon-ST loopback



(b) 8-bytes Avalon-ST *block FIFO* loopback

Figure 32: BeMicro-SDK plus UM245R asynchronous FIFO to Avalon-ST bridge loopback SignalTap II logic analyzer traces.

7.1.3 UM232H Asynchronous FIFO mode

Figure 33 shows SignalTap II logic analyzer traces for the BeMicro-SDK interfaced to the FTDI UM232H USB High-Speed device (with active-low write-enable);

- Figure 33(a) Avalon-ST loopback.

  In this example, the bridge Avalon-ST source and sink interfaces are directly connected; note how the traces for `rx_valid` and `rx_ready`, are identical to `tx_valid` and `tx_ready`. The received data, `rx_data[7:0]`, is inverted to create the transmitted data, `tx_data[7:0]`, as this makes it easier to see when the FPGA drives the FTDI data bus.

  The effect of the Avalon-ST loopback can be seen on the FTDI bus traces, where each read from the FTDI bus is immediately followed by a write to the bus.

  The back-to-back reads and writes occur every 170ns, i.e., the read byte-stream is 6MB/s and the write byte-stream is also 6MB/s, or three times faster than the FT245R module.

- Figure 33(b) Avalon-ST block FIFO loopback.

  In this example, the bridge Avalon-ST source interface fills the block FIFO, and the Avalon-ST sink interface drains the block FIFO. The block FIFO is configured for a FIFO depth of 8-bytes.

  The presence of the Avalon-ST block FIFO between the bridge Avalon-ST interfaces causes the FTDI interface to issue 8 reads followed by 8 writes. This sequence allows the timing of back-to-back reads (with no writes) and back-to-back writes (with no reads) to be observed.

  The back-to-back reads occur every 140ns or 7MB/s, while the back-to-back reads occur every 100ns or 10MB/s. The observed data transfer rates are consistent with the data sheet claim of 8MB/s.

  The two SignalTap II traces in Figure 33(a) and 33(b) have the same time-span. The Avalon-ST loopback transfers occur slightly faster than the block FIFO loopback transfers.

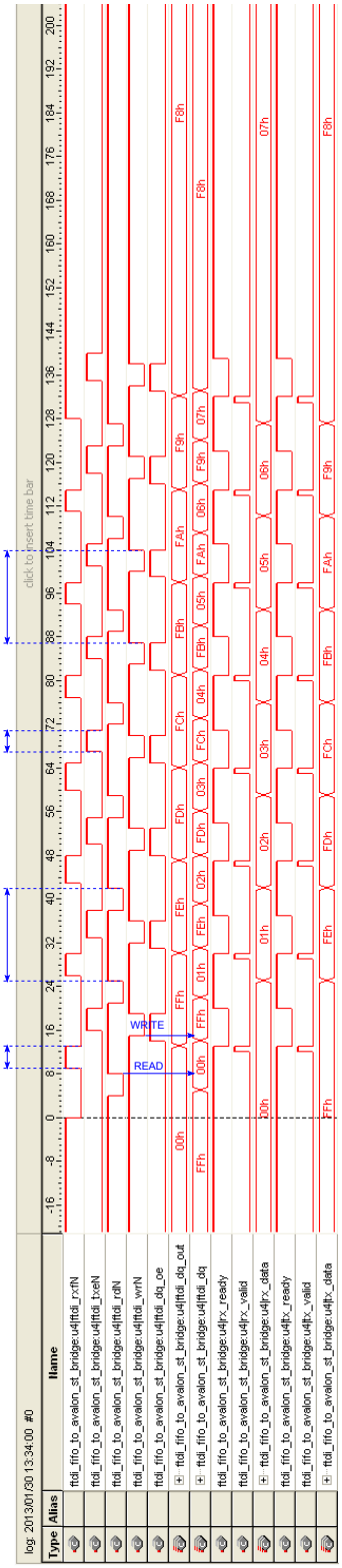
- FTDI device timing parameters.

  Table 3 contains the UM232H timing parameters. Figure 33 shows that the RXF# and TXE# high times observed in hardware are consistent with the data sheet values. The SignalTap II trace pulses are either 40ns or 50ns wide (sampled with a 100MHz or 10ns period clock).

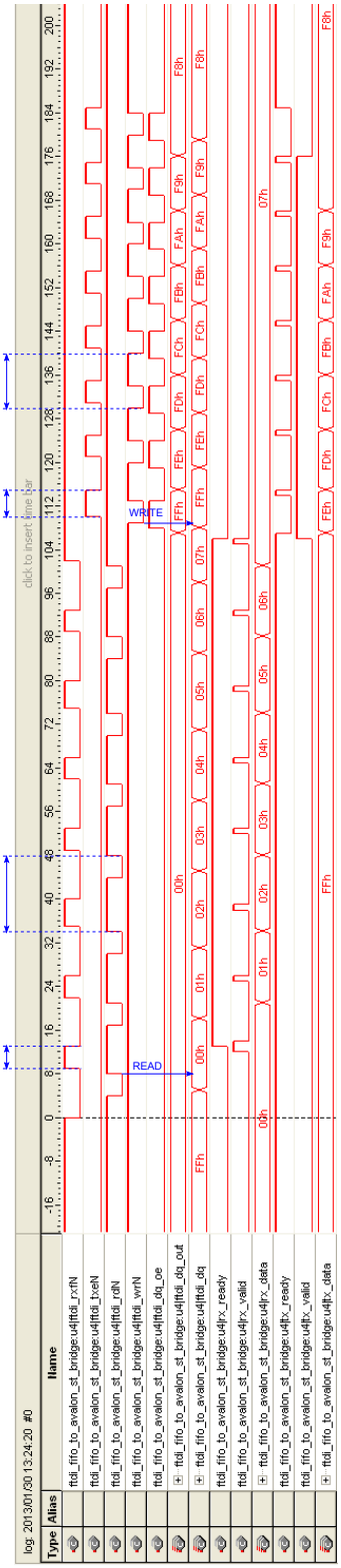
- Cyclone IV Resources.

  The `ftdi_afifo_to_avalon_st_bridge` requires 41 logic cells;

  - `ftdi_arbiter` (3 LCs)
  - `ftdi_afifo_rx` (16 LCs)
  - `ftdi_afifo_tx` (22 LCs)



(a) 8-bytes Avalon-ST loopback



(b) 8-bytes Avalon-ST *block FIFO* loopback

Figure 33: BeMicro-SDK plus UM232H asynchronous FIFO to Avalon-ST bridge loopback SignalTap II logic analyzer traces.

7.1.4 UM232H Synchronous FIFO mode

Figure 34 shows SignalTap II logic analyzer traces for the BeMicro-SDK interfaced to the FTDI UM232H USB High-Speed device;

- Figure 34 Avalon-ST loopback.

  In both of these examples, the bridge Avalon-ST source and sink interfaces are directly connected; note how the traces for `rx_valid` and `rx_ready`, are identical to `tx_valid` and `tx_ready`. The received data, `rx_data[7:0]`, is inverted to create the transmitted data, `tx_data[7:0]`, as this makes it easier to see when the FPGA drives the FTDI data bus. The bridge is operated in single-clock mode so that SignalTap II could probe both the FTDI and Avalon-ST sides of the bridge using the 60MHz FTDI clock.

  The synchronous mode bridge contains FIFOs in both the receive and transmit data paths, so the transactions inherently occur in bursts.

- Figure 34(a) single transaction.

  The timing sequence in this figure reproduces the single read timing in Figure 18(a) and the single write timing in Figure 20(a).

- Figure 33(b) burst transaction.

  The timing sequence in this figure reproduces the 8-byte burst read timing in Figure 18(b) and the burst write timing in Figure 20(b).

- Burst-mode performance.

  Burst transactions of 1MB (0x100000), 16MB (0x1000000), and 256MB (0x10000000) were tested. With a bridge FIFO size of 1024-bytes, the SignalTap II traces showed data transfers occurring in blocks of 512-bytes, which corresponds to the size of the USB Bulk data transfers (p1 [11]). Figure 35 shows a 768-byte burst with bridge FIFO depths of 256-bytes and 128-bytes. The burst transfers occur in blocks equal to the the sum of the receive and transmit FIFO depths.

  The read and write unidirectional transfer data rate during bursts is 60MB/s (1-byte per 60MHz clock period). The observed data transfer rates are consistent with the data sheet claim of 40MB/s.

- Cyclone IV Resources.

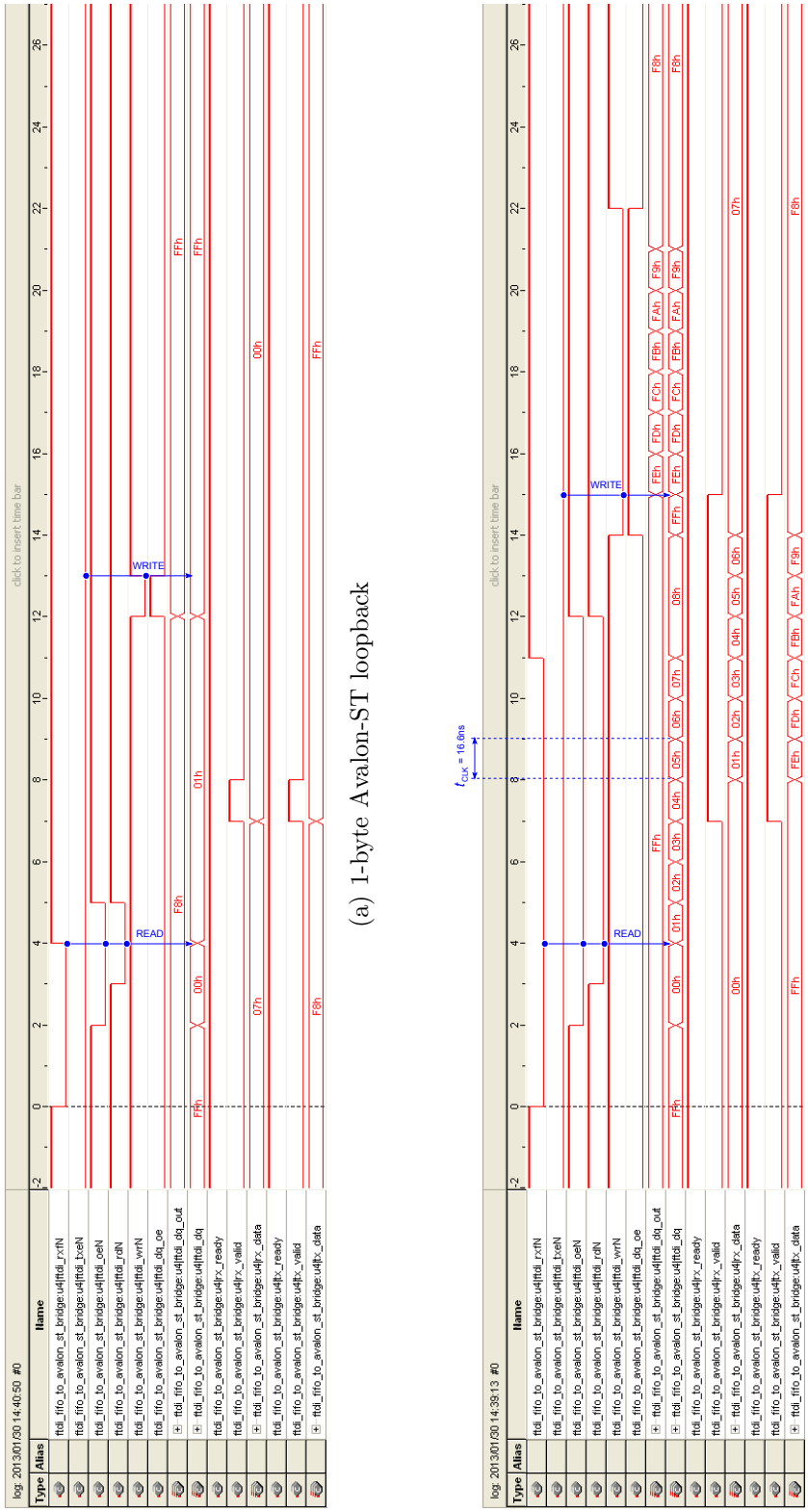
  In single-clock mode the `ftdi_sfifo_to_avalon_st_bridge` requires 163 logic cells and two M9K memory blocks;

  - `scfifo` (128 LCs + 2 M9K)
  - `ftdi_sfifo_to_fifo_interface` (35 LCs)

  In dual-clock mode the `ftdi_sfifo_to_avalon_st_bridge` requires 329 logic cells and two M9K memory blocks;

  - `dcfifo` (293 LCs + 2 M9K)
  - `ftdi_sfifo_to_fifo_interface` (36 LCs)

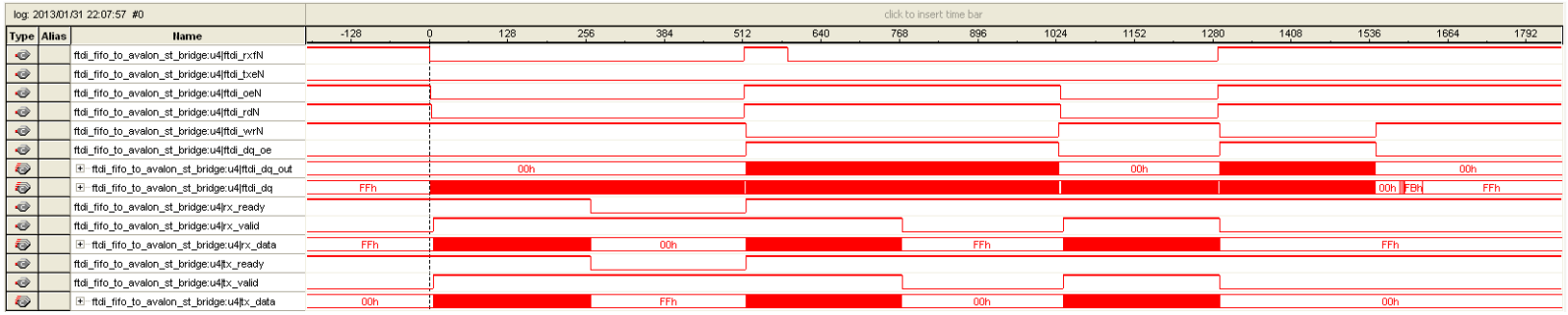
  The Cyclone IV FPGA contains only one memory block type, the M9K (9kbit blocks). To fully utilitize the memory block, the FIFO depths were configured for 1024-bytes (8192-bits). The logic cell resource requirements for the bridge can be reduced by reducing the FIFO depth, eg., reducing the FIFO depths to 32-bytes reduces the logic cell requirements of the single-clock bridge to 118 LCs and the dual-clock bridge to 196 LCs.



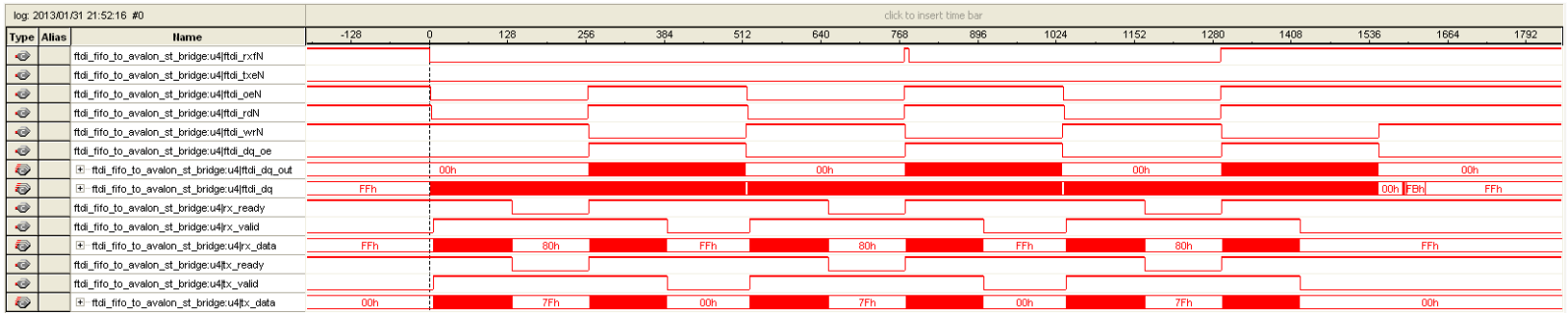
(a) 1-byte Avalon-ST loopback

(b) 8-bytes Avalon-ST loopback

Figure 34: BeMicro-SDK plus UM232H synchronous FIFO to Avalon-ST bridge loopback SignalTap II logic analyzer traces.



(a) Bridge FIFO depth of 256-bytes



(a) Bridge FIFO depth of 128-bytes

Figure 35: BeMicro-SDK plus UM232H synchronous FIFO to Avalon-ST bridge loopback SignalTap II logic analyzer traces for 758-byte bursts. The bursts occur in blocks of 512-bytes or 256-bytes.

7.2 UART to Avalon-ST Bridge Loopback

The BeMicro shown in Figure 28(a) uses an FTDI FT2232C dual-channel USB-to-Serial/Parallel converter to implement an *Arrow USB-Blaster* on channel 1, and a UART on channel 2 (with only the transmitter and receiver signals connected to the FPGA). The BeMicro was configured with a design consisting of;

- The UART to Avalon-ST bridge, `uart_to_avalon_st_bridge`, with the Avalon-ST interfaces connected, i.e., looped back.
- An 8-bit data register that is enabled for each Avalon-ST transaction, i.e., each time ready and valid are simultaneously asserted.
- The 8-bit data register drives the on-board LEDs (inversion is required to light the LEDs with the received binary code).
- A PLL configured to generate a 3.1MHz clock. This clock frequency was selected based on SignalTap II capture of 4k samples, i.e., the clock frequency (sample rate) was adjusted until the traces showed the desired samples.

Figure 36 shows SignalTap II logic analyzer traces for the BeMicro example design. The traces show the reception and transmission of the string `w 12 3456`. This string corresponds to an Avalon-ST to Avalon-MM bridge write command, i.e., write to address `12` the value `3456`. The zoomed traces at the bottom of the figure show how the transmitter Avalon-ST interface is ready before the receiver has valid transmitter data, i.e., there is never any need for Avalon-ST wait-states.

Arrow USB-Blaster and UART Interaction

During the capture of the SignalTap II traces in Figure 36 the BeMicro FT2232C dual-channel device needs to be accessed by both the SignalTap II logic analyzer and a terminal program. Each application needs exclusive access to the FT2232C, so the following sequence was needed;

1. Configure the BeMicro using the Quartus programmer.
2. Start the SignalTap II logic analyzer and initiate capture.
3. Start the terminal program, eg., TeraTerm (Windows XP).
4. Use the terminal program to send the string.
5. Close the terminal program (or at least disconnect from the serial port).
6. SignalTap II will then use the Arrow USB-Interface to download the traces.

If the terminal program does not close the serial port, then the SignalTap II trace acquisition never completes.

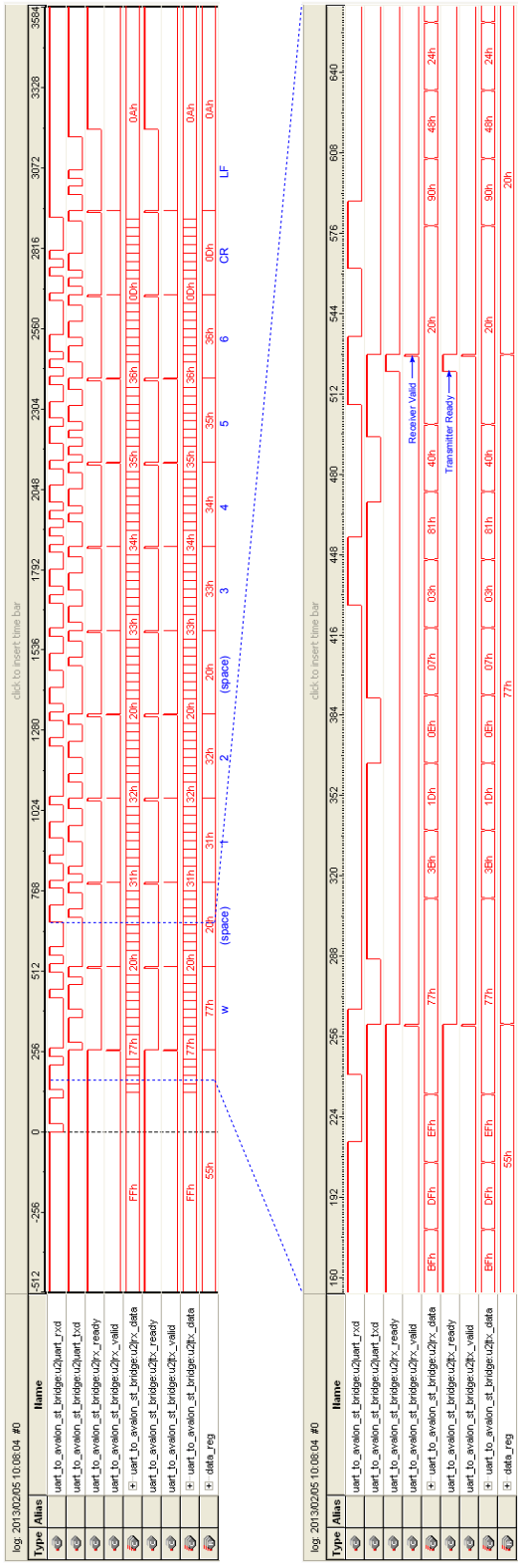


Figure 36: BeMicro FT2232C UART to Avalon-ST bridge loopback SignalTap II logic analyzer traces for the string `w 12 3456`.

7.3 Avalon-MM GPIO

Table 5: BeMicro and BeMicro-SDK FTDI Interface I/O.

FTDI Signal	Expansion Connector
Asynchronous FIFO mode	
`ftdi_dq[7:0]`	GPIO(11:4)
`ftdi_rxfN`	GPIO(12)
`ftdi_txeN`	GPIO(13)
`ftdi_rdN`	GPIO(14)
`ftdi_wrN`	GPIO(15)
Synchronous FIFO mode (BeMicro-SDK only)	
`ftdi_clk`	GPIN(1)
`ftdi_oeN`	GPIO(16)

A Hardware Configuration

A.1 BeMicro/BeMicro-SDK 80-pin Expansion Header Pinouts

Table 5 shows the FTDI to BeMicro and BeMicro-SDK board 80-pin expansion connector signal assignments. Tables 6 and 7 show the mapping between the BeMicro and BeMicro-SDK 80-pin expansion connector and the X201 and X202 headers on the prototype board. The FTDI modules are connected to the X201 and X202 headers via wires soldered between the modules and headers (the wiring is located on the under-side of the prototype boards in Figure 28). Tables 6 and 7 were constructed from the BeMicro User Manual (there is no schematic available) and the BeMicro-SDK schematic.

There are several important differences between the BeMicro and BeMicro-SDK 80-pin expansion header pin assignments;

- The BeMicro connects 64 general purpose I/Os to the GPIO bus in the tables. There is no *global input* assignment, so there is no option to route a clock from the expansion header to an FPGA clock input pin. Because of the lack of any clock pins on the header, the BeMicro was only used for FTDI asynchronous FIFO mode tests.

- The BeMicro-SDK uses the first four pins of the BeMicro GPIO bus for global input signals, so those four pins were renamed GPIN for the BeMicro-SDK.

  The GPIN signals route to FPGA clock input pins, allowing the BeMicro-SDK to be used for FTDI synchronous FIFO mode tests.

The pin assignments in Table 5 allow the UM245R module to be tested in asynchronous FIFO mode with both the BeMicro and BeMicro-SDK.

Table 6: BeMicro and BeMicro-SDK 80-pin edge connector pin assignments.

Pin Number	BeMicro		BeMicro-SDK		Breakout Pin Number
	Signal	Pin	Signal	Pin	
1	3.3V		3.3V		
2	3.3V		3.3V		
3	RST#	T12	RST#	T3	X201.35
4	PWR#		PWR#		X202.33
5	GPIO(0)	R12	GPIN(0)	T9	X201.34
6	GPIO(1)	T10	GPIN(1)	T8	X202.32
7	GPIO(2)	T13	GPIN(2)	R9	X201.33
8	GPIO(3)	R10	GPIN(3)	R8	X202.31
9	GPIO(4)	R13	GPIO(4)	R3	X201.32
10	GND		GND		
11	GPIO(5)	T14	GPIO(5)	P8	X201.31
12	GPIO(6)	T11	GPIO(6)	P6	X202.30
13	GPIO(7)	R14	GPIO(7)	P1	X201.30
14	GPIO(8)	R11	GPIO(8)	N8	X202.29
15	GPIO(9)	T15	GPIO(9)	M8	X201.29
16	GPIO(10)	N11	GPIO(10)	M7	X202.28
17	GPIO(11)	R16	GPIO(11)	P2	X201.28
18	GPIO(12)	N14	GPIO(12)	L8	X202.27
19	GPIO(13)	P14	GPIO(13)	N6	X201.27
20	GPIO(14)	N12	GPIO(14)	L7	X202.26
21	GND		GND		
22	GND		GND		
23	GPIO(15)	P16	GPIO(15)	R1	X201.26
24	GPIO(16)	M11	GPIO(16)	M6	X202.25
25	GPIO(17)	P15	GPIO(17)	N5	X201.25
26	GPIO(18)	L11	GPIO(18)	T2	X202.24
27	GPIO(19)	N16	GPIO(19)	K1	X201.24
28	GPIO(20)	L13	GPIO(20)	G5	X202.23
29	GPIO(21)	N15	GPIO(21)	K2	X201.23
30	GPIO(22)	L14	GPIO(22)	F3	X202.22
31	GPIO(23)	L16	GPIO(23)	P11	X201.22
32	GND		GND		
33	GND		GND		
34	GPIO(24)	L15	GPIO(24)	N16	X202.21
35	GPIO(25)	K15	GPIO(25)	L2	X201.21
36	GPIO(26)	K12	GPIO(26)	L1	X202.20
37	GPIO(27)	K16	GPIO(27)	P16	X201.20
38	GPIO(28)	J15	GPIO(28)	R16	X202.19
39	GPIO(29)	J16	GPIO(29)	P3	X201.19
40	GPIO(30)	J14	GPIO(30)	N3	X202.18

Table 7: BeMicro and BeMicro-SDK 80-pin edge connector pin assignments.

Pin Number	BeMicro		BeMicro-SDK		Breakout Pin Number
	Signal	Pin	Signal	Pin	
41	GPIO(31)	H16	GPIO(31)	N1	X201.18
42	GPIO(32)	J13	GPIO(32)	N2	X202.17
43	GPIO(33)	H15	GPIO(33)	L4	X201.17
44	GND		GND		
45	GPIO(34)	G16	GPIO(34)	L3	X201.16
46	GPIO(35)	J12	GPIO(35)	J2	X202.16
47	GPIO(36)	G15	GPIO(36)	J1	X201.15
48	GPIO(37)	G11	GPIO(37)	T11	X202.15
49	GPIO(38)	F16	GPIO(38)	T10	X201.14
50	GPIO(39)	F14	GPIO(39)	N11	X202.14
51	GPIO(40)	F15	GPIO(40)	G2	X201.13
52	GPIO(41)	F13	GPIO(41)	P14	X202.13
53	GND		GND		
54	GND		GND		
55	GPIO(42)	D15	GPIO(42)	G1	X201.12
56	GPIO(43)	E11	GPIO(43)	N12	X202.12
57	GPIO(44)	D16	GPIO(44)	N14	X201.11
58	GPIO(45)	E10	GPIO(45)	L14	X202.11
59	GPIO(46)	C15	GPIO(46)	F1	X201.10
60	GPIO(47)	D14	GPIO(47)	L15	X202.10
61	GPIO(48)	C16	GPIO(48)	L16	X201.9
62	GPIO(49)	D12	GPIO(49)	K15	X202.9
63	GPIO(50)	C14	GPIO(50)	F2	X201.8
64	GPIO(51)	A13	GPIO(51)	J14	X202.8
65	GPIO(52)	B16	GPIO(52)	J13	X201.7
66	GPIO(53)	C11	GPIO(53)	T7	X202.7
67	GPIO(54)	A15	GPIO(54)	M10	X201.6
68	GPIO(55)	C9	GPIO(55)		X202.6
69	GPIO(56)	B14	GPIO(56)		X201.5
70	GPIO(57)	B11	GPIO(57)		X202.5
71	GPIO(58)	A14	GPIO(58)		X201.4
72	GPIO(59)	A11	GPIO(59)		X202.4
73	GPIO(60)	B13	GPIO(60)		X201.3
74	GPIO(61)	B10	GPIO(61)		X202.3
75	GPIO(62)	B12	GPIO(62)		X201.2
76	GND		GND		
77	GPIO(63)	A12	GPIO(63)		X201.1
78	EXP	A10	EXP	N9	
79	5V		5V		
80	5V		5V		

Table 8: Morph-IC and MAX II PCI Kit FTDI Interface I/O.

FTDI Signal	Morph-IC I/O		MAX II PCI Kit I/O	
	Net Name	Connector	Net Name	Connector
ftdi_dq[0]	BD0	J2.3	SC_0	J4.3
ftdi_dq[1]	BD1	J2.4	SC_1	J4.4
ftdi_dq[2]	BD2	J2.2	SC_2	J4.5
ftdi_dq[3]	BD3	J2.1	SC_3	J4.6
ftdi_dq[4]	BD4	J1.3	SC_4	J4.7
ftdi_dq[5]	BD5	J1.4	SC_5	J4.8
ftdi_dq[6]	BD6	J1.5	SC_6	J4.9
ftdi_dq[7]	BD7	J1.6	SC_7	J4.10
ftdi_rxfN	BC0	J1.8	SC_8	J4.11
ftdi_txeN	BC1	J1.10	SC_9	J4.12
ftdi_rdN	BC2	J1.7	SC_10	J4.13
ftdi_wr	BC3	J1.1	SC_11	J4.14

A.2 Morph-IC and MAX II PCI Development Kit

Table 8 shows the connections used on the Morph-IC and MAX II PCI Development Kit when probing the FTDI bus using an Agilent 1680A logic analyzer. Both boards contain full-speed USB devices with active-high write-enable. The Morph-IC contains an FT2232C, while the MAX II Kit contains an FT245BM. The logic analyzer traces were not included in this document, as the BeMicro and BeMicro-SDK SignalTap II traces provide the same information. The connections in Table 8 for the MAX II PCI Development Kit, are located on the largest of the three connectors that make up the Santa Cruz header.

References

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

[2] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000.

[3] FTDI. FT245BL: USB FIFO IC (version 1.7). Datasheet, 2005. (DS_FT245BL.pdf).

[4] FTDI. FT2232D: Dual USB-to-Serial UART/FIFO IC (version 2.05). Datasheet, 2010. (DS_FT2232D.pdf).

[5] FTDI. FT232R: USB UART IC (version 2.10). Datasheet, 2010. (DS_FT232R.pdf).

[6] FTDI. FT245R: USB FIFO IC (version 2.12). Datasheet, 2010. (DS_FT245R.pdf).

[7] FTDI. FT4232DH: Quad High-Speed USB-to-Multipurpose UART/MPSSE IC (version 2.2). Datasheet, 2010. (DS_FT4232H.pdf).

[8] FTDI. FT232BL/BQ: USB UART IC (version 2.2). Datasheet, 2011. (DS_FT232BL_BQ.pdf).

[9] FTDI. FT2232DH: Dual High-Speed USB-to-Multipurpose UART/FIFO IC (version 2.21). Datasheet, 2012. (DS_FT2232H.pdf).

[10] FTDI. FT230X: USB-to-Basic UART IC (version 1.1). Datasheet, 2012. (DS_FT230X.pdf).

[11] FTDI. FT232DH: Single channel High-Speed USB-to-Multipurpose UART/FIFO IC (version 1.8). Datasheet, 2012. (DS_FT232H.pdf).

[12] FTDI. FT240X: USB 8-bit FIFO IC (version 1.2). Datasheet, 2012. (DS_FT240X.pdf).

[13] M. Stein. Crossing the Abyss: Asynchronous Signals in a Synchronous World. EDN Magazine Article, July 2003. (17561-310388.pdf).