Avalon Interface Specifications

Subscribe

Send Feedback

MNL-AVABUSREF
2014.06.30

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

Contents

Introduction to the Avalon Interface Specifications 1

Subscribe Send Feedback

Avalon® interfaces simplify system design by allowing you to easily connect components in an Altera® FPGA. The Avalon interface family defines interfaces appropriate for streaming high-speed data, reading and writing registers and memory, and controlling off-chip devices. These standard interfaces are designed into the components available in Qsys. You can also use these standardized interfaces in your custom components. By using these standard interfaces, you enhance the interoperability of your designs.

This specification defines all of the Avalon interfaces. After reading it, you should understand which interfaces are appropriate for your components and which signal roles to use for particular behaviors. This specification defines the following seven interface roles:

- Avalon Streaming Interface (Avalon-ST)—an interface that supports the unidirectional flow of data, including multiplexed streams, packets, and DSP data.
- Avalon Memory Mapped Interface (Avalon-MM)—an address-based read/write interface typical of master–slave connections.
- Avalon Conduit Interface— an interface type that accommodates individual signals or groups of signals that do not fit into any of the other Avalon types. You can connect conduit interfaces inside a Qsys system. Or, you can export them to make connections to other modules in the design or to FPGA pins.
- Avalon Tri-State Conduit Interface (Avalon-TC) —an interface to support connections to off-chip peripherals. Multiple peripherals can share pins through signal multiplexing, reducing the pin count of the FPGA and the number of traces on the PCB.
- Avalon Interrupt Interface—an interface that allows components to signal events to other components.
- Avalon Clock Interface—an interface that drives or receives clocks. All Avalon interfaces are synchronous.
- Avalon Reset Interface—an interface that provides reset connectivity.

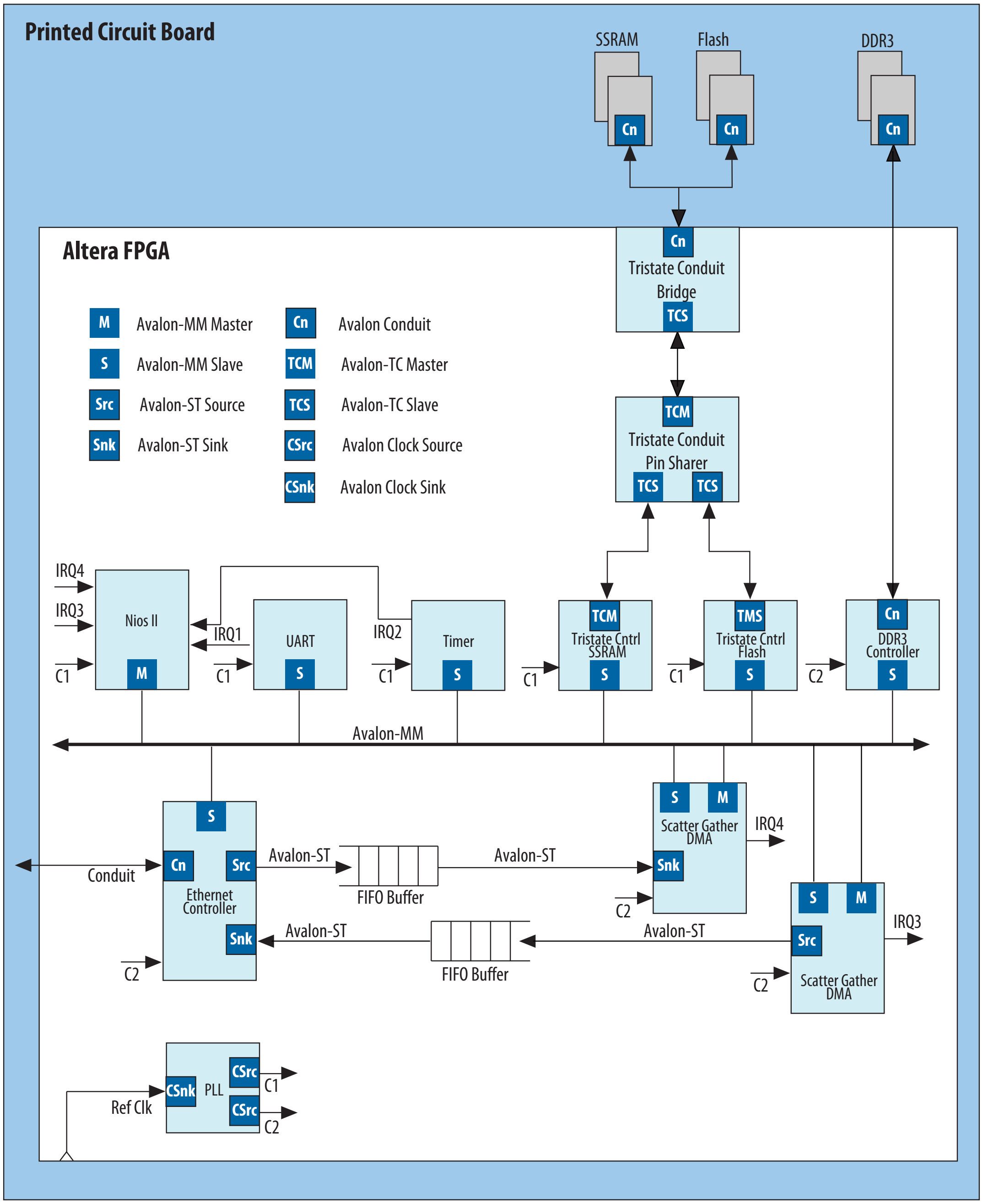
A single component can include any number of these interfaces and can also include multiple instances of the same interface type. For example, in the first figure below, the Ethernet Controller includes the following six different interface types:

- Avalon-MM
- Avalon-ST
- Avalon Conduit
- Avalon-TC
- Avalon Interrupt
- Avalon Clock.

Note: Avalon interfaces are an open standard. No license or royalty is required to develop and sell products that use, or are based on Avalon interfaces.

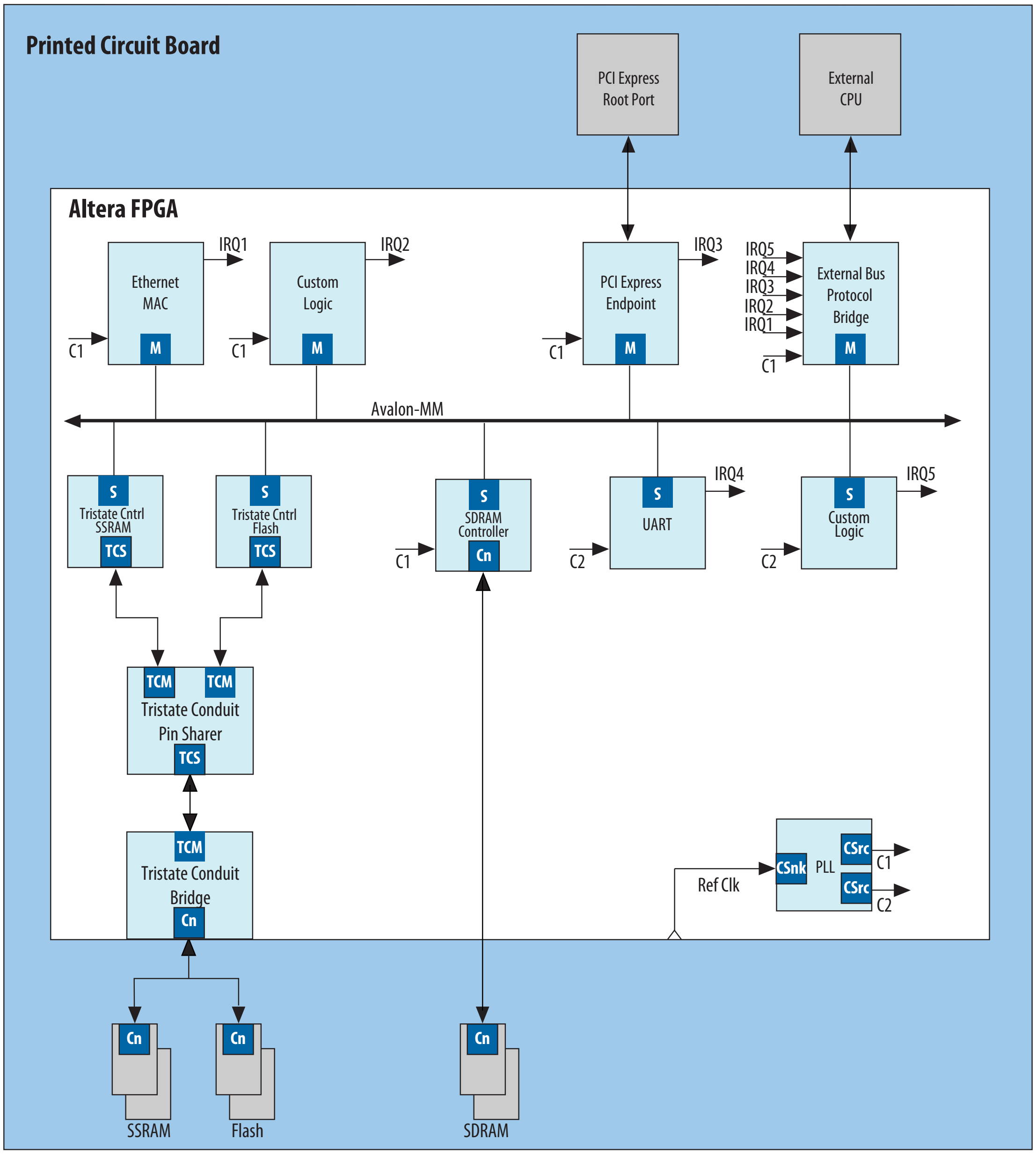
The following figures illustrate the use of the Avalon interfaces in system designs.

Figure 1-1: Avalon Interfaces in a System Design with Scatter Gather DMA Controller and Nios II Processor



In this figure, the Nios® II processor accesses the control and status registers of on-chip components using an Avalon-MM interface. The scatter gather DMAs send and receive data using Avalon-ST interfaces. Four components include interrupt interfaces serviced by software running on the Nios II processor. A PLL accepts a clock via an Avalon Clock Sink interface and provides two clock sources. Two components include Avalon-TC interfaces to access off-chip memories. Finally, the DDR3 controller accesses external DDR3 memory using an Avalon Conduit interface.

Figure 1-2: Avalon Interfaces in a System Design with PCI Express Endpoint and External Processor



In the previous figure, an external processor accesses the control and status registers of on-chip components via an external bus bridge with an Avalon-MM interface. The PCI Express Root Port controls devices on the printed circuit board and the other components of the FPGA by driving an on-chip PCI Express Endpoint with an Avalon-MM master interface. An external processor handles interrupts from five components. A PLL accepts a reference clock via a Avalon Clock sink interface and provides two clock sources. The flash and SRAM memories use an Avalon-TC interface to share FPGA pins. Finally, an SDRAM controller accesses an external SDRAM memory using an Avalon Conduit interface.

1.1 Avalon Properties and Parameters

Avalon interfaces use properties to describe their behavior. For example, the `maxChannel` property of Avalon-ST interfaces allows you to specify the number of channels supported by the interface. The `clockRate` property of the Avalon Clock interface provides the frequency of a clock signal. The specification for each interface type defines all of its properties and specifies the default values.

1.2 Signal Roles

Each of the Avalon interfaces defines a number of signal roles and their behavior. Many signal roles are optional. You have the flexibility to select only the signal roles necessary to implement the required functionality. For example, the Avalon-MM interface includes optional `beginbursttransfer` and `burstcount` signal roles for use in components that support bursting. The Avalon-ST interface includes the optional `startofpacket` and `endofpacket` signal roles for interfaces that support packets.

With the exception of Avalon Conduit interfaces, each interface may include only one signal of each signal role. Active-low signals are permitted for many signal roles. Active-high signals are generally used in this document.

1.3 Interface Timing

Subsequent chapters of this document include timing information that describes transfers for individual interface types. There is no guaranteed performance for any of these interfaces. Actual performance depends on many factors, including component design and system implementation.

Most Avalon interfaces must not be edge sensitive to signals other than the clock and reset. Other signals may transition multiple times before they stabilize. The exact timing of signals between clock edges varies depending upon the characteristics of the selected Altera device. This specification does not specify electrical characteristics. Refer to the appropriate device documentation for electrical specifications.

1.4 Related Documents

For more information on related topics in the following documents and design examples, refer to the following documents:

Related Information

- **Creating a System with Qsys.**
  For an overview of the Qsys system integration tool
- **Creating Qsys Components**
  For information about creating Qsys components, composed components, and dynamic file generation
- **Optimizing Qsys System Performance.**
  For information about system design, including: hierarchy, concurrency, pipelining, throughput, reducing logic utilization and power consumption

- **Component Interface Tcl Reference**
  For information about defining Qsys components and a reference for component Tool Command Language (Tcl
- **Qsys System Design Components**
  For information about system design components available in the IP Catalog
- **Qsys Tutorial Design Example**
  For tutorial that builds a memory test system using components with Avalon interfaces

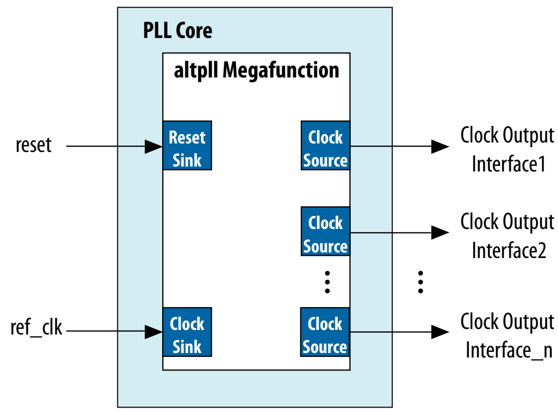
Avalon Clock and Reset Interfaces 2

Subscribe Send Feedback

Avalon Clock interfaces define the clock or clocks used by a component. Components can have clock inputs, clock outputs, or both. A phase locked loop (PLL) is an example of a component that has both a clock input and clock outputs.

The following figure is a simplified illustration showing the most important inputs and outputs of a PLL component.

Figure 2-1: PLL Core Clock Outputs and Inputs



2.1 Clock Sink Signal Roles

A clock sink provides a timing reference for other interfaces and internal logic.

Table 2-1: Clock Input Signal Roles

Signal Role	Width	Direction	Required	Description
`clk`	1	Input	Yes	A clock signal. Provides synchronization for internal logic and for other interfaces.

2.2 Clock Sink Properties

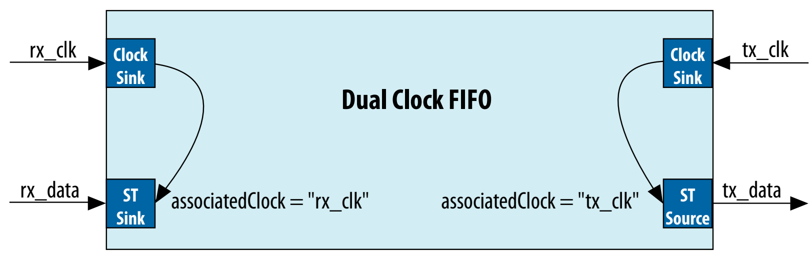
Table 2-2: Clock Sink Properties

Name	Default Value	Legal Values	Description
`clockRate`	0	$0–2^{32}–1$	Indicates the frequency in Hz of the clock sink interface. If 0, the clock rate is not significant.

2.3 Associated Clock Interfaces

All synchronous interfaces have an `associatedClock` property that specifies which clock input on the component is used as a synchronization reference for the interface. This property is illustrated in the following figure.

Figure 2-2: associatedClock Property



2.4 Clock Source Signal Roles

An Avalon Clock source interface drives a clock signal out of a component.

Table 2-3: Clock Source Signal Roles

Signal Role	Width	Direction	Required	Description
`clk`	1	Output	Yes	An output clock signal.

2.5 Clock Source Properties

Table 2-4: Clock Source Properties

Name	Default Value	Legal Values	Description
associatedDirect-Clock	N/A	a clock name	The name of the clock input that directly drive this clock output, if any.
`clockRate`	0	$0–2^{32}–1$	Indicates the frequency in Hz at which the clock output is driven.

Name	Default Value	Legal Values	Description
`clockRateKnown`	false	true, false	Indicates whether or not the clock frequency is known. If the clock frequency is known, this information can be used to customize other components in the system.

2.6 Reset Sink

Table 2-5: Reset Input Signal Roles

The `reset_req` signal is an optional signal that you can use to prevent memory content corruption by performing reset handshake prior to processor reset.

Signal Role	Width	Direction	Required	Description
`reset reset_n`	1	Input	Yes	Resets the internal logic of an interface or component to a user-defined state. Synchronous to the clock input in the associated clock interface.
`reset_req`	1	input	Optional	Early indication of reset signal. When asserted the component is expected to prepare itself to be reset.

2.7 Reset Sink Interface Properties

Table 2-6: Reset Input Signal Roles

Name	Default Value	Legal Values	Description
`associated-Clock`	N/A	a clock name	The name of a clock to which this interface synchronized. Required if the value of `synchronousEdges` is `DEASSERT` or `BOTH`.

`synchronous-Edges`	`DEASSERT`	`NONE`
		`DEASSERT`
		`BOTH`
			Indicates the type of synchronization the reset input requires. The following values are defined:
			• `NONE`–no synchronization is required because the component includes logic for internal synchronization of the reset signal.
			• `DEASSERT`–the reset assertion is asynchronous and deassertion is synchronous.
			`BOTH`–reset assertion and deassertion are synchronous.

2.8 Associated Reset Interfaces

All synchronous interfaces have an `associatedReset` property that specifies which reset signal resets the interface logic.

2.9 Reset Source

Table 2-7: Reset Output Signal Roles

The `reset_req` signal is an optional signal that you can use to prevent memory content corruption by performing reset handshake prior to processor reset.

Signal Role	Width	Direction	Required	Description
`reset reset_n`	1	Output	Yes	Resets the internal logic of an interface or component to a user-defined state.
`reset_req`	1	Output	Optional	Enables reset request generation, which is an early signal that is asserted before reset assertion. Once asserted, this cannot be deasserted until the reset is completed.

2.10 Reset Source Interface Properties

Table 2-8: Reset Interface Properties

Name	Default Value	Legal Values	Description
`associatedClock`	N/A	a clock name	The name of a clock to which this interface synchronized. Required if the value of `synchronousEdges` is `DEASSERT` or `BOTH`.
`associatedDirec-tReset`	N/A	a reset name	The name of the reset input that directly drives this reset source through a one-to-one link.
`associate-dResetSinks`	N/A	a reset name	Specifies reset inputs which will eventually cause a reset source to assert reset. For example, a reset synchronizer `OR`s a number of reset inputs to generate a reset output.
`synchronousEdges`	`DEASSERT`	`NONE` `DEASSERT` `BOTH`	indicates the type of synchronization the reset input requires. The following values are defined: • `NONE`–no synchronization is required because the component includes logic for internal synchronization of the reset signal. • `DEASSERT`–the reset assertion is asynchronous and deassertion is synchronous. • `BOTH`–reset assertion and deassertion are synchronous.

Avalon Memory-Mapped Interfaces 3

Subscribe Send Feedback

3.1 Introduction to Avalon Memory-Mapped Interfaces

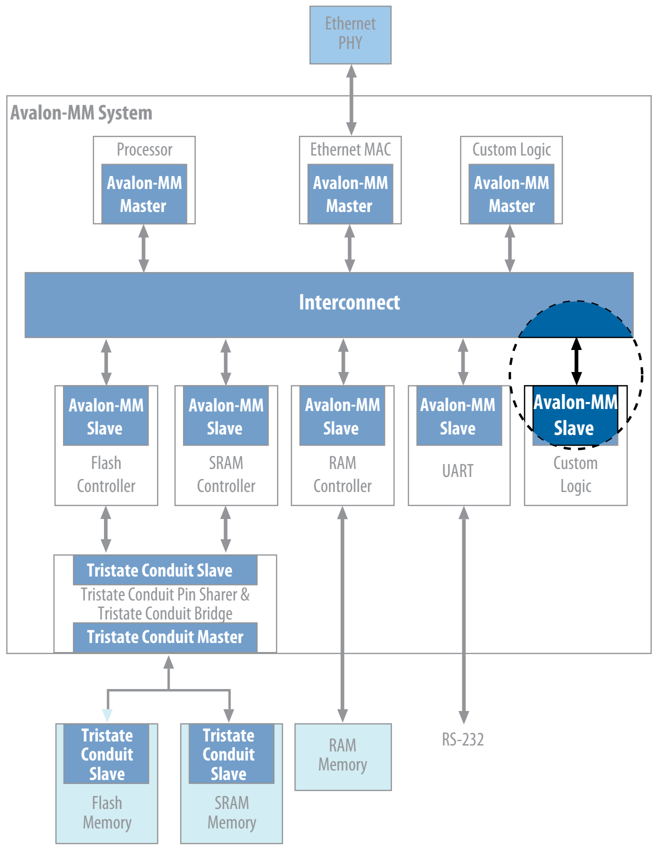
You can use Avalon Memory-Mapped (Avalon-MM) interfaces to implement read and write interfaces for master and slave components. The following are examples of component that typically include memory-mapped interfaces:

- Microprocessors
- Memories
- UARTs
- DMAs
- Timers

Avalon-MM interfaces range from simple to complex. For example, SRAM interfaces that have fixed-cycle read and write transfers have simple Avalon-MM interfaces. Pipelined interfaces capable of burst transfers are complex.

Figure 3-1: Focus on Avalon-MM Slave Transfers

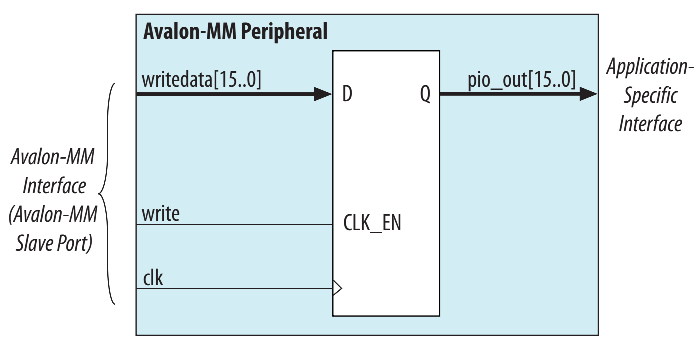
The following figure shows a typical system, highlighting the Avalon-MM slave interface connection to the interconnect fabric.



Avalon-MM components typically include only the signals required for the component logic.

Figure 3-2: Example Slave Component

The 16-bit general-purpose I/O peripheral shown in the following figure only responds to write requests. This component includes only the slave signals required for write transfers.



Each signal in an Avalon-MM slave corresponds to exactly one Avalon-MM signal role. An Avalon-MM port can use only one instance of each signal role.

3.2 Signals

The following table lists the signal roles that constitute the Avalon-MM interface. The signal roles allow you to create masters that use bursts for reads and writes. You can increase the throughput of your system by initiating reads with multiple pipelined slave peripherals. In responding to reads, when a slave peripheral has valid data it asserts `readdatavalid` . The interconnect enables the connection between the master and slave pair.

This specification does not require all signals to exist in an Avalon-MM interface. In fact, there is no one signal that is always required. The minimum requirements are `readdata` for a read-only interface or `writedata` and `write` for a write-only interface.

Table 3-1: Avalon-MM Signals

All Avalon signals are active high. Avalon signals that can also be asserted low list `_n` versions of the signal in the **Signal role** column.

Signal Role	Width	Direction	Description
			Fundamental Signals
`address`	1 - 64	Master → Slave	Masters: By default, the `address` signal represents a byte address. The value of the address must be aligned to the data width. To write to specific bytes within a data word, the master must use the `byteenable` signal. Refer to the `addressUnits` interface property for word addressing. Slaves: By default, the interconnect translates the byte address into a word address in the slave’s address space. Each slave access is for a word of data from the perspective of the slave. For example, `address`= 0 selects the first word of the slave.Address 1 selects the second word of the slave. Refer to the `addressUnits` interface property for byte addressing.
`begintransfer`	1	Master → Slave	Asserted by the interconnect for the first cycle of each transfer regardless of `waitrequest` and other signals. The `begintransfer` signal is optional. A slave can always internally calculate the start of the next transaction from other signals. **Note:** Altera recommends that you do not use this signal. This signal exists to support legacy memory controllers.

Signal Role	Width	Direction	Description
byteenable byteenable_n	2, 4, 8, 16, 32, 64, 128	Master → Slave	Enables specific byte lane(s) during transfers on ports of width greater than 8 bits. Each bit in byteenable corresponds to a byte in writedata and readdata. The master bit <n> of byteenable indicates whether byte <n> is being written to. During writes, byteenables specify which bytes are being written to. Other bytes should be ignored by the slave. During reads, byteenables indicate which bytes the master is reading. Slaves that simply return readdata with no side effects are free to ignore byteenables during reads. If an interface does not have a byteenable signal, the transfer proceeds as if all byteenables are asserted. When more than one bit of the byteenable signal is asserted, all asserted lanes are adjacent. The number of adjacent lines must be a power of 2. The specified bytes must be aligned on an address boundary for the size of the data. For example, the following values are legal for a 32-bit slave: • 1111 writes full 32 bits • 0011 writes lower 2 bytes • 1100 writes upper 2 bytes • 0001 writes byte 0 only • 0010 writes byte 1 only • 0100 writes byte 2 only • 1000 writes byte 3 only Altera strongly recommends that you use the byteenable signal in components that will be used in systems with different word sizes. Using byte enables avoids unintended side effects in systems that include width adapters.
debugaccess	1	Master → Slave	When asserted, allows internal memories that are normally write-protected to be written. For example, on-chip ROM memories can only be written when debugaccess is asserted.
read read_n	1	Master → Slave	Asserted to indicate a read transfer. If present, readdata is required.
readdata	8,16, 32, 64, 128, 256, 512, 1024	Slave → Master	The readdata driven from the slave to the master in response to a read transfer.

Signal Role	Width	Direction	Description
`write` `write_n`	1	Master → Slave	Asserted to indicate a `write` transfer. If present, `writedata` is required.
`writedata`	8,16, 32, 64, 128, 256, 512, 1024	Master → Slave	Data for write transfers. The width must be the same as the width of `readdata` if both are present.
Wait-State Signals			
`lock`	1	Master → Slave	`lock` ensures that once a master wins arbitration, it maintains access to the slave for multiple transactions. It is asserted coincident with the first `read` or `write` of a locked sequence of transactions. It is deasserted on the final transaction of a locked sequence of transactions. `lock` assertion does not guarantee that arbitration will be won. After the lock-asserting master has been granted, it retains grant until it is deasserted. A master equipped with `lock` cannot be a burst master. Arbitration priority values for lock-equipped masters are ignored. `lock` is particularly useful for read-modify-write (RMW) operations. The typical read-modify-write operation includes the following steps: **1.** Master A reads 32-bit data that has multiple bit fields. **2.** Master A changes one bit field and writes the 32-bit data back. `lock` prevents master B from performing a write between Master A's read and write. `lock` also ensures that master A does not overwrite master B's changes.

Signal Role	Width	Direction	Description
`waitrequest` `waitrequest_n`	1	Slave → Master	Asserted by the slave when it is unable to respond to a `read` or `write` request. Forces the master to wait until the interconnect is ready to proceed with the transfer. At the start of all transfers, a master initiates the transfer and waits until `waitrequest` is deasserted. A master must make no assumption about the assertion state of `waitrequest` when the master is idle: `waitrequest` may be high or low, depending on system properties. When `waitrequest` is asserted, master control signals to the slave remain constant with the exception of `begintransfer`., and `beginbursttransfer`. For a timing diagram illustrating the `begintransfer` signal, refer to **Figure 3-3**. For a timing diagram illustrating the `beginbursttransfer` signal, refer to **Figure 3-7**. An Avalon-MM slave may assert `waitrequest` during idle cycles. An Avalon-MM master may initiate a transaction when `waitrequest` is asserted and wait for that signal to be deasserted. To avoid system lockup, a slave device should assert `waitrequest` when in reset.
Pipeline Signals			
`readdatavalid` `readdatavalid_n`	1	Slave → Master	Used for variable-latency, pipelined `read` transfers. When asserted, indicates that the `readdata` signal contains valid data. A slave with `readdatavalid` must assert this signal for one cycle for each `read` access received. There must be at least one cycle of latency between acceptance of the `read` and assertion of `readdatavalid`. For an timing diagram illustrating the `readdatavalid`signal, refer to **Figure 3-5**. A slave may assert `readdatavalid` to transfer data to the master independently of whether or not the slave is stalling a new command with `waitrequest`. Required if the master supports pipelined reads. Bursting masters with read functionality must include the `readdatavalid` signal.
Burst Signals			

Signal Role	Width	Direction	Description
`burstcount`	1 – 11	Master → Slave	Used by bursting masters to indicate the number of transfers in each burst. The value of the maximum `burstcount` parameter must be a power of 2. A burstcount port of width <n> can encode a max burst of size $2^{(<n>-1)}$. For example, a 4-bit `burstcount` signal can support a maximum burst count of 8. The minimum `burstcount` is 1. The `constantBurst` property controls the timing of the `burstcount` signal. Bursting masters with read functionality must include the `readdatavalid` signal. For bursting masters and slaves, the following restriction applies to the width of the address: `<address_w> >= <burstcount_w> + floor(log2 (<symbols_per_word_of_interface>))`
`beginbursttransfer`	1	Interconnect → Slave	Asserted for the first cycle of a burst to indicate when a burst transfer is starting. This signal is deasserted after one cycle regardless of the value of `waitrequest`. The interconnect fabric automatically generates this signal for slaves when requested. For a timing diagram illustrating `beginbursttransfer`, refer to **Figure 3-7**. `beginbursttransfer` is optional. A slave can always internally calculate the start of the next write burst transaction by counting data transfers. Altera recommends that you **do not** use this signal. This signal exists to support legacy memory controllers.

3.3 Interface Properties

Table 3-2: Avalon-MM Interface Properties

Name	Default Value	Legal Values	Description
`addressUnits`	Master - symbols Slave - words	words, symbols	Specifies the unit for addresses. Byte or word addressing are available. A symbol is typically a byte.

Name	Default Value	Legal Values	Description
`burstCountUnits`	words	words, symbols	This property specifies the units for the burstcount signal. For symbols, the `burstcount` value is interpreted as the number of symbols (bytes) in the burst. For words, the `burstcount` value is interpreted as the number of data-width transfers in the burst.
`burstOnBurstBoundariesOnly`	false	true, false	If true, burst transfers presented to this interface begin at addresses which are multiples of the burst size in bytes.
`constantBurstBehavior`	Master - false Slave - false	true, false	Masters: When true, declares that the master holds address and burstcount constant throughout a burst transaction. When false (default), declares that the master holds address and burstcount constant only for the first beat of a burst. Slaves: When true, declares that the slave expects address and burstcount to be held constant throughout a burst. When false (default), declares that the slave samples address and burstcount only on the first beat of a burst.
`holdTime`(1)	0	0 – 1000 cycles	Specifies time in `timingUnits` between the deassertion of `write` and the deassertion of `chipselect`, `address`, and `data`. (Only applies to write transactions.)
`linewrapBursts`	false	true, false	Some memory devices implement a wrapping burst instead of an incrementing burst. When a wrapping burst reaches a burst boundary, the address wraps back to the previous burst boundary. Only the low-order bits are required for address counting. For example, a wrapping burst to addres 0xC with burst boundaries every 32 bytes across a 32-bit interface writes to the following addresses: • 0xC • 0x10 • 0x14 • 0x18 • 0x1C • 0x0 • 0x4 • 0x8

Name	Default Value	Legal Values	Description
`maximumPendingReadTransactions` (1)	1(2)	1 – 64	Slaves: This parameter is the maximum number of pending reads that the slave can queue. For a timing diagram that illustrates this property, refer to Figure 3-5. Do not set this parameter to 0. (For backwards compatibility, the software supports a parameter setting of 0. However, you should not use this setting in new designs) . Masters: This property is the maximum number of outstanding read transactions that the master can generate. Do not set this parameter to 0. (For backwards compatibility, the software supports a parameter setting of 0. However, you should not use this setting in new designs) .
`readLatency`(1)	0	0 – 63	Read latency for fixed-latency Avalon-MM slaves. Not used on interfaces that include the `readdatavalid` signal. For a timing diagram that uses a fixed latency read, refer to Figure 3-6.
`readWaitTime`(1)	1	0 – 1000 cycles	For interfaces that don't use the `waitrequest` signal. `readWaitTime` indicates the number of cycles or nanoseconds before the slave accepts a read command. The timing is as if the slave asserted `waitrequest` for `readWaitTime` cycles.
`setupTime`(1)	0	0 – 1000 cycles	Specifies time in `timingUnits` between the assertion of `chipselect, address`, and `data` and assertion of `read` or `write`.
`timingUnits`(1)	cycles	cycles, nanoseconds	Specifies the units for `setupTime`, `holdTime`, `writeWaitTime` and `readWaitTime`. Use cycles for synchronous devices and nanoseconds for asynchronous devices. Almost all Avalon-MM slave devices are synchronous. An Avalon-MM slave that reads and writes an off-chip bidirectional port is asynchronous. That off-chip device might have a fixed settling time for bus turnaround.

Name	Default Value	Legal Values	Description
writeWaitTime(1)	0	0 – 1000 Cycles	For interfaces that do not use the waitrequest signal, writeWaitTime indicates the number of cycles or nanoseconds before a slave accepts a write. The timing is as if the slave asserted waitrequest for writeWaitTime cycles or nanoseconds. For a timing diagram that illustrates the use of writeWaitTime, refer to **Figure 3-4**.
Interface Relationship Properties			
associatedClock	N/A	N/A	Name of the clock interface to which this Avalon-MM interface is synchronous.
associatedReset	N/A	N/A	Name of the reset interface to which this Avalon-MM interface is synchronous.
bridgesToMaster	0	Avalon-MM Master on the Same Component	An Avalon-MM bridge consists of a slave and a master, and has the property that an access to the slave requesting a particular byte or bytes will cause the same byte or bytes to be requested by the master. The Avalon-MM Pipeline Bridge in the Qsys component library implements this functionality.

Notes:

1. Although this property characterizes a slave device, masters can declare this property to enable direct connections between matching master and slave interfaces.
2. If a component accepts more read transfers than the value indicated here, the internal pending read FIFO may overflow with unpredictable results, including the loss of readdata, routing of readdata to the wrong master interface, or system lockup.

3.4 Timing

The Avalon-MM interface is synchronous. Each Avalon-MM port is synchronized to an associated clock interface. Signals may be combinational if they are driven from the outputs of registers that are synchronous to the clock signal. This specification does not dictate how or when signals transition between clock edges. Timing diagrams are devoid of fine-grained timing information.

3.5 Transfers

This section defines two basic concepts before introducing the transfer types:

- Transfer—A transfer is a read or write operation of a word or symbol of data. Transfers occur between an Avalon-MM port and the interconnect. Avalon-MM transfers words ranging in size from 8–1024 bits. Transfers take one or more clock cycles to complete.

  Both masters and slaves are part of a transfer. The Avalon-MM master initiates the transfer and the Avalon-MM slave responds to it.

- Master-slave pair—This term refers to the master port and slave port involved in a transfer. During a transfer, the master port control and data signals pass through the interconnect fabric and interact with the slave port.

3.5.1 Typical Read and Write Transfers

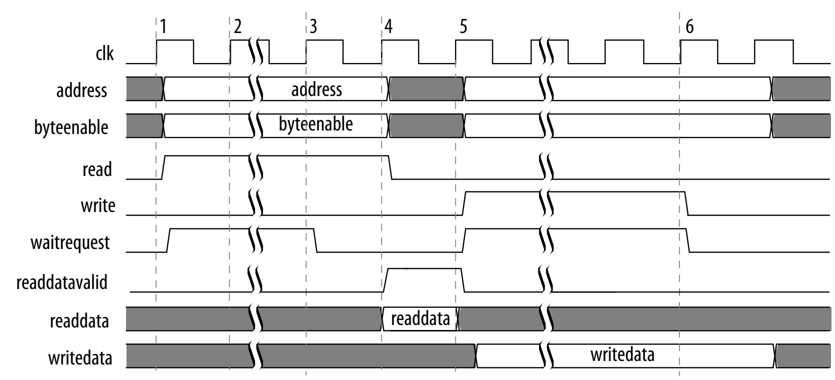
This section describes a typical Avalon-MM interface that supports read and write transfers with slave-controlled `waitrequest`. The slave can stall the interconnect for as many cycles as required by asserting the `waitrequest` signal. If a slave uses `waitrequest` for either read or write transfers, it must use `waitrequest` for both.

A slave typically receives `address`, `byteenable`, `read` or `write`, and `writedata` after the rising edge of the clock. A slave asserts `waitrequest` before the rising clock edge to hold off transfers. When the slave asserts `waitrequest`, the transfer is delayed. And, the address and control signals are held constant. Transfers complete on the rising edge of the first `clk` after the slave port deasserts `waitrequest`.

There is no limit on how long a slave port can stall. Therefore, you must ensure that a slave port does not assert `waitrequest` indefinitely. The following figure shows `read` and `write` transfers using `waitrequest`. In this example, the master and slave both have a `readdatavalid` signal.

Note: `waitrequest` can be decoupled from the `read` and `write` request signals. `waitrequest` may be asserted during idle cycles. An Avalon-MM master may initiate a transaction when `waitrequest` is asserted and wait for that signal to be deasserted. Decoupling `waitrequest` from `read` and `write` requests may improve system timing. Decoupling eliminates a combinational loop including the `read`, `write`, and `waitrequest` signals.

Figure 3-3: Read and Write Transfers with Waitrequest



The numbers in this timing diagram, mark the following transitions:

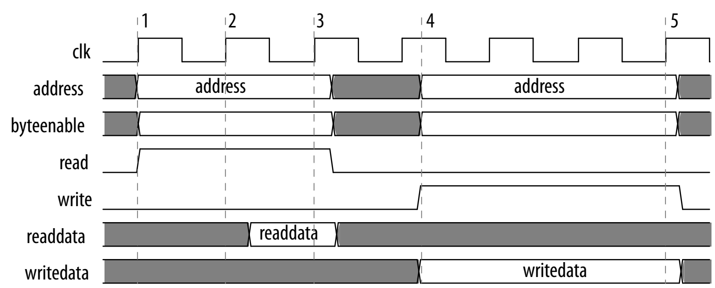
1. `address`, `read`, and `begintransfer` are asserted after the rising edge of `clk`. `waitrequest` is asserted stalling the transfer.
2. `waitrequest` is sampled. Because `waitrequest` is asserted, the cycle becomes a wait-state. `address`, `read`, `write`, and `byteenable` remain constant.
3. The slave deasserts `waitrequest`.
4. `readdata` and deasserted `waitrequest` are sampled, completing the transfer.
5. `address`, `writedata`, `byteenable`, `begintransfer`, and `write` signals are asserted. The slave responds by asserting `waitrequest` stalling the transfer.
6. The slave deasserts `waitrequest` and captures write data ending the transfer.

3.5.2 Read and Write Transfers with Fixed Wait-States

A slave can specify fixed wait-states using the `readWaitTime` and `writeWaitTime` properties. Using fixed wait-states is an alternative to using `waitrequest` to stall a transfer. The address and control signals (`byteenable`, `read`, and `write`) are held constant for the duration of the transfer. Setting `readWaitTime` or `writeWaitTime` to <*n*> is equivalent to asserting `waitrequest` for <*n*> cycles per transfer.

In the following figure, the slave has a `writeWaitTime = 2` and `readWaitTime = 1`.

Figure 3-4: Read and Write Transfer with Fixed Wait-States at the Slave Interface



The numbers in this timing diagram, mark the following transitions:

1. The master asserts `address` and `read` on the rising edge of clk.
2. The next rising edge of `clk` marks the end of the first and only wait-state cycle. The `readWaitTime` is 1.
3. The slave captures `readdata` on the rising edge of `clk`. The read transfer ends.
4. `writedata`, `address`, `byteenable`, and `write` signals are available to the slave.
5. Because `writeWaitTime` is 2, the transfer terminates after completing. The data and control signals are held constant until this time.

Transfers with a single wait-state are commonly used for multicycle off-chip peripherals. The peripheral captures address and control signals on the rising edge of `clk`. The peripheral has one full cycle to return data. Components with zero wait-states are allowed. However, components with zero wait-states may decrease the achievable frequency. Zero wait-states requires the component to generate the response in the same cycle as the request.

3.5.3 Pipelined Transfers

Avalon-MM pipelined read transfers increase the throughput for synchronous slave devices that require several cycles to return data for the first access. Such devices can can typically return one data value per cycle for some time thereafter. New pipelined read transfers can start before `readdata` for the previous transfers is returned. Write transfers cannot be pipelined.

A pipelined read transfer has an address phase and a data phase. A master initiates a transfer by presenting the address during the address phase. A slave port fulfills the transfer by delivering the data during the data phase. The address phase for a new transfer (or multiple transfers) can begin before the data phase of a previous transfer completes. The delay is called pipeline latency. The pipeline latency is the duration from the end of the address phase to the beginning of the data phase.

Transfer timing for wait-states and pipeline latency have the following key differences:

- Wait-states—Wait-states determine the length of the address phase. Wait-states limit the maximum throughput of a port. If a slave requires one wait-state to respond to a transfer request, the port requires at least two clock cycles per transfer.
- Pipeline Latency—Pipeline latency determines the time until data is returned independently of the address phase. A pipelined slave port with no wait-states can sustain one transfer per cycle. However, it may require several cycles of latency to return the first unit of data.

Wait-states and pipelined reads can be supported concurrently. Pipeline latency can be either fixed or variable.

3.5.3.1 Pipelined Read Transfer with Variable Latency

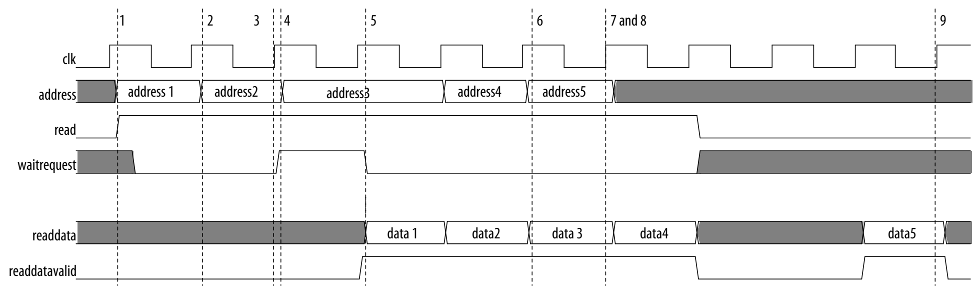
After capturing address and control signals, an Avalon-MM pipelined slave takes one or more cycles to produce data. A pipelined slave port may have multiple pending read transfers at any given time. Variable-latency pipelined read transfers require one additional signal, `readdatavalid`. `readdatavalid` indicates when read data is valid. Variable-latency pipelined read transfer also include the same set of signals as non-pipelined read transfers. Slave peripherals that use `readdatavalid` are considered pipelined with variable latency. The `readdata` and `readdatavalid` signals can be asserted the cycle after the read cycle is asserted, at the earliest.

The slave port must return `readdata` in the same order that it accepted the addresses. Pipelined slave ports with variable latency must use `waitrequest`. The slave can assert `waitrequest` to stall transfers to maintain an acceptable number of pending transfers. A slave may assert `readdatavalid` to transfer data to the master independently of whether or not the slave is stalling a new command with `waitrequest`.

Note: The maximum number of pending transfers is a property of the slave interface. The interconnect fabric builds logic to route `readdata` to requesting masters using this number. The slave interface, not the interconnect fabric, must track the number of pending reads. The slave must assert `waitrequest` to prevent the number of pending reads from exceeding the maximum number.

Figure 3-5: Pipelined Read Transfers with Variable Latency

The following figure shows several slave read transfers. The slave is pipelined with variable latency. In this figure, the slave can accept a maximum of two pending transfers. The slave uses `waitrequest` to avoid overrunning this maximum.



The numbers in this timing diagram, mark the following transitions:

1. The master asserts `address` and `read`, initiating a read transfer.
2. The slave captures `addr1`.
3. The slave captures `addr2`.
4. The slave asserts `waitrequest` because it has accepted a maximum of two pending reads, causing the third transfer to stall.
5. The slave provides data1, the response to addr1. It deassertes `waitrequest`.
6. The slave drives `readdatavalid` and valid `readdata` in response to the third read transfer. The interconnect captures `data2`.
7. The interconnect captures data from transfer 3. The slave captures `addr4` at the same.
8. The slave captures `addr5`. The interconnect captures `data4`.
9. The slave drives `data5` with `readdatavalid` completing the data phase for the final pending read transfer.

If the slave cannot handle a write transfer while it is processing pending read transfers, the slave must assert its `waitrequest` and stall the write operation until the pending read transfers have completed. The Avalon-MM specification does not define the value of `readdata` in the event that a slave accepts a write transfer to the same address as a currently pending read transfer. Pipelined slaves with variable latency must support `waitrequest`.

3.5.3.2 Pipelined Read Transfers with Fixed Latency

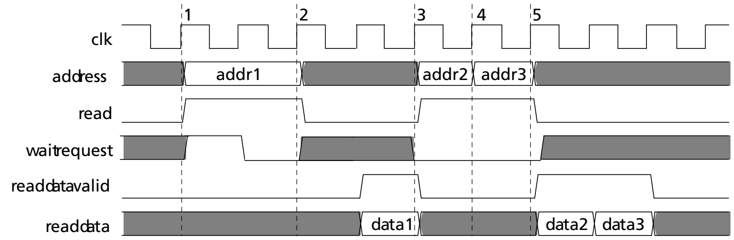
The address phase for fixed latency read transfers is identical to the variable latency case. After the address phase, a pipelined slave port with fixed read latency takes a fixed number of clock cycles to return valid `readdata`. The `readWaitTime` property specifies the number of clock cycles to return valid `readdata`. The interconnect captures `readdata` on the appropriate rising clock edge, ending the data phase.

During the address phase, the slave port asserts `waitrequest` to hold off the transfer. Or, the slave port specifies the `readWaitTime` for a fixed number of wait states. The address phase ends on the next rising edge of `clk` after wait states, if any.

During the data phase, the slave drives `readdata` after a fixed latency. For a read latency of <*n*>, the slave port must present valid `readdata` on the <*nth*> rising edge of `clk` after the end of the address phase.

Figure 3-6: Pipelined Read Transfer with Fixed Latency of Two Cycles

The following figure shows multiple data transfers between a master and a pipelined slave port. The slave drives `waitrequest` to stall transfers. and has a fixed read latency of 2 cycles.



The numbers in this timing diagram, mark the following transitions:

1. A master initiates a read transfer by asserting `read` and `addr1`. The slave asserts `waitrequest` to hold off the transfer for one cycle.
2. The slave deasserts `waitrequest` and captures `addr1` at the rising edge of `clk`. The address phase ends here.
3. The slave presents valid `readdata` after 2 cycles, ending the transfer.
4. `addr2` and `read` are asserted for a new read transfer.
5. The master initiates a third read transfer during the next cycle, before the data from the prior transfer is returned.

3.5.4 Burst Transfers

A burst executes multiple transfers as a unit, rather than treating every word independently. Bursts may increase throughput for slave ports that achieve greater efficiency when handling multiple word at a time, such as SDRAM. The net effect of bursting is to lock the arbitration for the duration of the burst. A bursting Avalon-MM interface that supports both reads and writes, must support both read and write bursts.

Bursting Avalon-MM interfaces include a `burstcount` output signal. If a slave has a `burstcount` input, it is considered burst capable.

The `burstcount` signal behaves as follows:

- At the start of a burst, `burstcount` presents the number of sequential transfers in the burst.
- For width <*n*> of `burstcount`, the maximum burst length is $2^{(<n>-1}$).The minimum legal burst length is one.

To support slave read bursts, a slave must also support:

- Wait states with the `waitrequest` signal.
- Pipelined transfers with variable latency with the `readdatavalid` signal.

At the start of a burst, the slave sees the `address` and a burst length value on `burstcount`. For a burst with an address of <*a*> and a `burstcount` value of <*b*>, the slave must perform <*b*> consecutive transfers starting at address <*a*>. The burst completes after the slave receives (write) or returns (read) the <b^{th}> word of data. The bursting slave must capture `address` and `burstcount` only once for each burst. The slave logic must infer the address for all but the first transfers in the burst. A slave can also use the input signal `beginbursttransfer`, which the interconnect asserts on the first cycle of each burst.

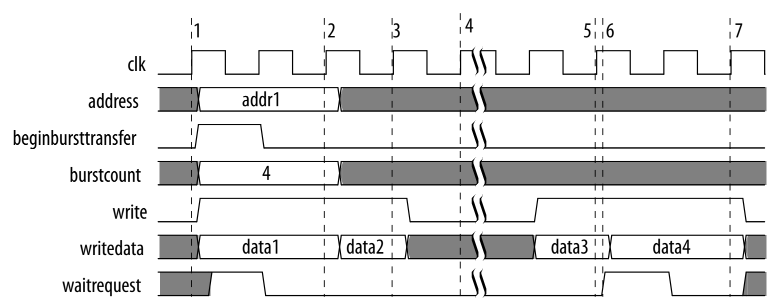
3.5.4.1 Write Bursts

These rules apply when a write burst begins with `burstcount` greater than one:

- When a `burstcount` of <*n*> is presented at the beginning of the burst, the slave must accept <*n*> successive units of `writedata` to complete the burst. Arbitration between the master-slave pair is locked until the burst completes. This lock guarantees that data arrives in order from the master port initiating the burst.
- The slave must only capture `writedata` when `write` is asserted. During the burst, the master can desassert `write` indicating that `writedata` is invalid. Deasserting `write` does not terminate the burst. It delays it. When a burst is delayed, no other masters can access the slave, reducing the transfer efficiency.
- The `constantBurstBehavior` property controls the behavior of the burst signals. When `constantBurst-Behavior` is true for a master, it indicates that the master holds `address` and `burstcount` stable throughout a burst. When `constantBurstBehavior` is false, it indicates that the master holds `address` and `burstcount` stable only for the first transaction of a burst. When true for a slave, `constantBurstBe-havior` declares that the slave expects `address` and `burstcount` to be held stable throughout a burst. When `constantBurstBehavior` is false, it indicates that the slave samples `address` and `burstcount` only on the first transaction of a burst.
- The slave can delay a transfer by asserting `waitrequest` forcing `writedata`, `write`, and `byteenable` to be held constant.
- The functionality of the `byteenable` signal is the same for bursting and non-bursting slaves. For a 32-bit master burst-writing to a 64-bit slave, starting at byte address 4, the first write transfer seen by the slave is at its address 0, with `byteenable = 8b'11110000`. The `byteenables` can change for different words of the burst.
- The `byteenable` signals do not all have to be asserted. A burst master writing partial words can use the `byteenable` signal to identify the data being written.

Figure 3-7: Write Burst with constantBurstBehavior Set to False for Master and Slave

The following figure demonstrates a slave write burst of length 4. In this example, the slave port asserts `waitrequest` twice delaying the burst.



The numbers in this timing diagram, mark the following transitions:

1. The master asserts `address`, `burstcount`, `write`, and drives the first unit of `writedata`. The slave immediately asserts `waitrequest`, indicating that it is not ready to proceed with the transfer.
2. `waitrequest` is low. The slave captures `addr1`, `burstcount`, and the first unit of `writedata` . On subsequent cycles of the transfer, `address` and `burstcount` are ignored.
3. The slave port captures the second unit of data at the rising edge of `clk`.
4. The burst is paused while `write` is deasserted.
5. The slave captures the third unit of data at the rising edge of `clk`.
6. The slave asserts `waitrequest`. In response, all outputs are held constant through another clock cycle.
7. The slave captures the last unit of data on this rising edge of `clk`. The slave write burst ends.

In the figure above, the `beginbursttransfer` signal is asserted for the first clock cycle of a burst and is deasserted on the next clock cycle. Even if the slave asserts `waitrequest`, the `beginbursttransfer` signal is only asserted for the first clock cycle.

For information about Avalon-MM properties, refer to **Table 3-2**.

Related Information

Interface Properties on page 3-7

3.5.4.2 Read Bursts

Read bursts are similar to pipelined read transfers with variable latency. A read burst has distinct address and data phases. `readdatavalid` indicates when the slave is presenting valid `readdata`. Unlike pipelined read transfers, a single read burst address results in multiple data transfers.

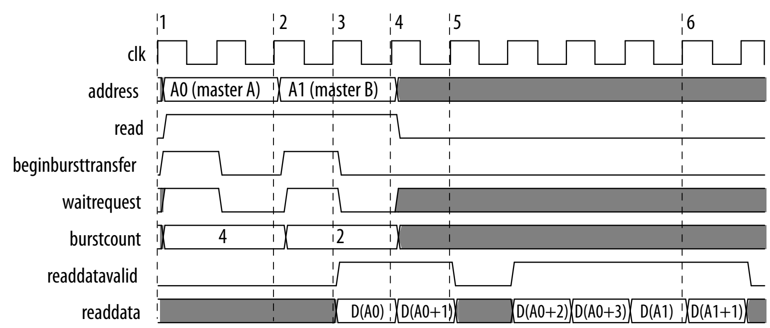
These rules apply to read bursts:

- When `burstcount` is <*n*>, the slave must return <*n*> words of `readdata` to complete the burst.
- The slave presents each word by providing `readdata` and asserting `readdatavalid` for a cycle. Deassertion of `readdatavalid` delays but does not terminate the burst data phase.
- The `byteenables` presented with a read burst command apply to all cycles of the burst. A `byteenable` value of 1 means that the least significant byte is being read across all of the read cycles.

Note: Altera recommends that burst capable slaves not have read side effects. (This specification does not guarantee how many bytes will be read from the slave in order to satisfy a request.)

Figure 3-8: Read Burst

The following figure illustrates a system with two bursting masters accessing a slave. Note that Master B can drive a read request before the data has returned for Master A.



The numbers in this timing diagram, mark the following transitions:

1. Master A asserts `address` (`A0`), `burstcount`, and `read` after the rising edge of `clk`. The slave asserts `waitrequest`, causing all inputs except `beginbursttransfer` to be held constant through another clock cycle.
2. The slave captures `A0` and `burstcount` at this rising edge of `clk`. A new transfer could start on the next cycle.
3. Master B drives `address` (`A1`), `burstcount`, and `read`. The slave asserts `waitrequest`, causing all inputs except `beginbursttransfer` to be held constant. The slave could have returned read data from the first read request at this time, at the earliest.
4. The slave presents valid `readdata` and asserts `readdatavalid`, transferring the first word of data for master A.
5. The second word for master A is transferred. The slave deasserts `readdatavalid` pausing the read burst. The slave port can keep `readdatavalid` deasserted for an arbitrary number of clock cycles.
6. The first word for master B is returned.

3.5.4.3 Line–Wrapped Bursts

Processors with data or instruction caches gain efficiency by using line-wrapped bursts. When a processor requests data that is not in the cache, the cache controller reads enough data from the memory to fill the entire cache line. For a processor with a cache line size of 64 bytes, a cache miss causes 64 bytes to be read from memory. If the processor reads from address 0xC when the cache miss occurred, then an incrementing addressing burst cache controller could issue a burst at address 0, resulting in data from read addresses 0x0, 0x4, 0x8, 0xC, 0x10, 0x14, 0x18, and 0x1C. The requested data is not available until the fourth read. With wrapping bursts, the address order is 0xC, 0x10, 0x14, 0x18, 0x1C, 0x0, 0x4, and 0x8. The requested data is returned first.

3.6 Address Alignment

For systems in which master and slave data widths differ, the interconnect manages address alignment issues. The Avalon-MM interface resolves data width differences, so that any master port can communicate with any slave port, regardless of the respective data widths. The interconnect only supports aligned accesses. A master can only issue addresses that are a multiple of its data width. A master can write partial words by deasserting some `byteenables`. For example, a burst of size 2 at address 0 would have 4'b1100 for the `byteenables`.

3.7 Avalon-MM Slave Addressing

Dynamic bus sizing dynamically manages data during transfers between master-slave pairs of differing data widths. Slave data are aligned in contiguous bytes in the master address space.

If the master is wider than the slave, data bytes in the master address space map to multiple locations in the slave address space. For example, a 32-bit master read from a 16-bit slave port results in two read transfers on the slave side. The reads are to consecutive addresses.

If the master is narrower than the slave, then the interconnect manages the slave byte lanes. During master read transfers, the interconnect presents only the appropriate byte lanes of slave data to the narrower master. During master write transfers, the interconnect automatically asserts the `byteenable` signals to write data only to the specified slave byte lanes.

Slaves must have a data width of 8, 16, 32, 64, 128, 256, 512 or 1024 bits. The following table shows how slave data of various widths is aligned within a 32-bit master performing full-word accesses. In this table, `OFFSET[N]` refers to a slave word size offset into the slave address space.

Table 3-3: Dynamic Bus Sizing Master-to-Slave Address Mapping

Master Byte Address (1)	Access	32-Bit Master Data		
		When Accessing an 8-Bit Slave Port	When Accessing a 16-Bit Slave Port	When Accessing a 64-Bit Slave Port
0x00	1	OFFSET[0] $_{7..0}$	OFFSET[0] $_{15..0}$ (2)	OFFSET[0] $_{31..0}$
	2	OFFSET[1] $_{7..0}$	OFFSET[1] $_{15..0}$	—
	3	OFFSET[2] $_{7..0}$	—	—
	4	OFFSET[3] $_{7..0}$	—	—
0x04	1	OFFSET[4] $_{7..0}$	OFFSET[2] $_{15..0}$	OFFSET[0] $_{63..32}$
	2	OFFSET[5] $_{7..0}$	OFFSET[3] $_{15..0}$	—
	3	OFFSET[6] $_{7..0}$	—	—
	4	OFFSET[7] $_{7..0}$	—	—
0x08	1	OFFSET[8] $_{7..0}$	OFFSET[4] $_{15..0}$	OFFSET[1] $_{31..0}$
	2	OFFSET[9] $_{7..0}$	OFFSET[5] $_{15..0}$	—
	3	OFFSET[10] $_{7..0}$	—	—
	4	OFFSET[11] $_{7..0}$	—	—
0x0C	1	OFFSET[12] $_{7..0}$	OFFSET[6] $_{15..0}$	OFFSET[1] $_{63..32}$
	2	OFFSET[13] $_{7..0}$	OFFSET[7] $_{15..0}$	—
	3	OFFSET[14] $_{7..0}$	—	—
	4	OFFSET[15] $_{7..0}$	—	—
...			...	...

Notes:

1. Although the master is issuing byte addresses, it is accessing full 32-bit words.
2. For all slave entries, [<*n*>] is the word offset and the subscript values are the bits in the word.

Avalon Interrupt Interfaces 4

Subscribe Send Feedback

Avalon Interrupt interfaces allow slave components to signal events to master components. For example, a DMA controller can interrupt a processor when it has completed a DMA transfer.

4.1 Interrupt Sender

An interrupt sender drives a single interrupt signal to an interrupt receiver. The timing of the `irq` signal must be synchronous to the rising edge of its associated clock. `irq` has no relationship to any transfer on any other interface. `irq` must be asserted until acknowledged on the associated Avalon-MM slave interface.

Interrupts are component specific. The receiver typically determines the appropriate response by reading an interrupt status register from an Avalon-MM slave interface.

4.1.1 Interrupt Sender Signal Roles

Table 4-1: Interrupt Sender Signal Roles

Signal Role	Width	Direction	Required	Description
`irq` `irq_n`	1	Output	Yes	Interrupt Request. A slave asserts `irq` when it needs service.

4.1.2 Interrupt Sender Properties

Table 4-2: Interrupt Sender Properties

Property Name	Default Value	Legal Values	Description
`associatedAddressablePoint`	N/A	Name of Avalon-MM slave on this component.	The name of the Avalon-MM slave interface that provides access to the registers to service the interrupt.
`associatedClock`	N/A	Name of a clock interface on this component.	The name of the clock interface to which this interrupt sender is synchronous. The sender and receiver may have different values for this property.

Property Name	Default Value	Legal Values	Description
`associated Reset`	N/A	Name of a reset interface on this component.	The name of the reset interface to which this interrupt sender is synchronous.

4.2 Interrupt Receiver

An interrupt receiver interface receives interrupts from interrupt sender interfaces. Components with Avalon-MM master interfaces can include an interrupt receiver to detect interrupts asserted by slave components with interrupt sender interfaces. The interrupt receiver accepts interrupt requests from each interrupt sender as a separate bit.

4.2.1 Interrupt Receiver Signal Roles

Table 4-3: Interrupt Receiver Signal Roles

Signal Role	Width	Direction	Required	Description
`irq`	1–32	Input	Yes	`irq` is an <*n*>-bit vector, where each bit corresponds directly to one IRQ sender, with no inherent assumption of priority.

4.2.2 Interrupt Receiver Properties

Table 4-4: Interrupt Receiver Properties

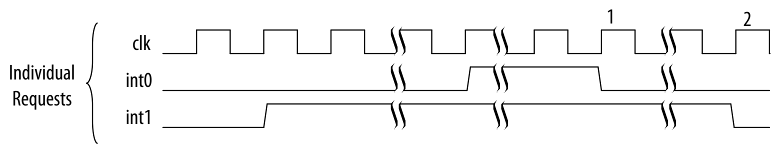
Property Name	Default Value	Legal Values	Description
`associatedAddress-able Point`	N/A	Name of Avalon-MM master interface	The name of the Avalon-MM master interface used to service interrupts received on this interface.
`associatedClock`	N/A	Name of an Avalon Clock interface	The name of the Avalon Clock interface to which this interrupt receiver is synchronous. The sender and receiver may have different values for this property.
`associatedReset`	N/A	Name of an Avalon Reset interface	The name of the reset interface to which this interrupt receiver is synchronous.
`irqScheme`	`individualRequests`	`individualRequests`	Each interrupt sender interface asserts its `irq` signal to request service.

4.2.3 Interrupt Timing

The Avalon-MM master services the priority 0 interrupt before the priority 1 interrupt.

Figure 4-1: Interrupt Timing for Individual Request and Priority Encoded Interrupts

In the following figure, the interrupt receiver interface services `int0` at time 1. It then services `int1` at time 2.

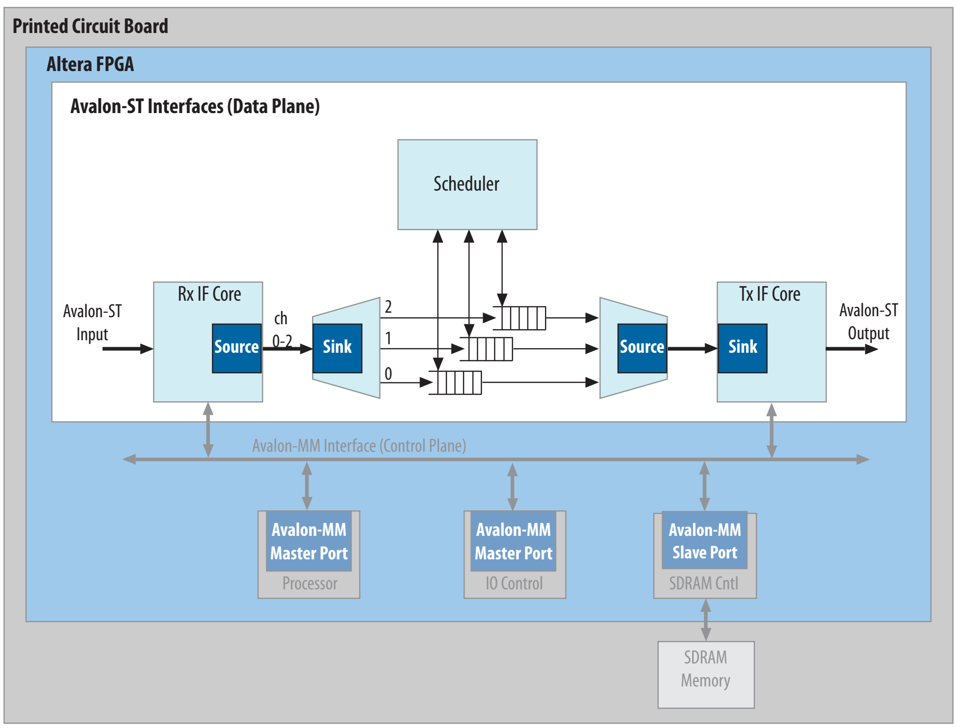


Avalon Streaming Interfaces 5

Subscribe Send Feedback

You can use Avalon Streaming (Avalon-ST) interfaces for components that drive high bandwidth, low latency, unidirectional data. Typical applications include multiplexed streams, packets, and DSP data. The Avalon-ST interface signals can describe traditional streaming interfaces supporting a single stream of data without knowledge of channels or packet boundaries. The interface can also support more complex protocols capable of burst and packet transfers with packets interleaved across multiple channels.

Figure 5-1: Avalon-ST Interface - Typical Application of the Avalon-ST Interface



All Avalon-ST source and sink interfaces are not necessarily interoperable. However, if two interfaces provide compatible functions for the same application space, adapters are available to allow them to interoperate.

Avalon-ST interfaces support datapaths requiring the following features:

- Low latency, high throughput point-to-point data transfer
- Multiple channel support with flexible packet interleaving
- Sideband signaling of channel, error, and start and end of packet delineation

- Support for data bursting
- Automatic interface adaptation

5.1 Terms and Concepts

The Avalon-ST interface protocol defines the following terms and concepts:

- Avalon Streaming System—An Avalon Streaming system contains one or more Avalon-ST connections that transfer data from a source interface to a sink interface. The system shown above consists of Avalon-ST interfaces to transfer data from the system input to output. Avalon-MM control and status register interfaces provide for software control.
- Avalon Streaming Components—A typical system using Avalon-ST interfaces combines multiple functional modules, called components. The system designer configures the components and connects them together to implement a system.
- Source and Sink Interfaces and Connections—When two components are connected, the data flows from the source interface to the sink interface. The combination of a source interface connected to a sink interface is referred to as a connection.
- Backpressure—Backpressure allows a sink to signal a source to stop sending data. Support for backpressure is optional. The sink uses backpressure to stop the flow of data for the following reasons:
  - When its FIFOs are full
  - When there is congestion on its output port.
- Transfers and Ready Cycles—A transfer results in data and control propagation from a source interface to a sink interface. For data interfaces, a ready cycle is a cycle during which the sink can accept a transfer.
- Symbol—A symbol is the smallest unit of data. For most packet interfaces, a symbol is a byte. One or more symbols make up the single unit of data transferred in a cycle.
- Channel—A channel is a physical or logical path or link through which information passes between two ports.
- Beat—A beat is a single cycle transfer between a source and sync interface made up of one or more symbols.
- Packet—A packet is an aggregation of data and control signals that is transmitted together. A packet may contain a header to help routers and other network devices direct the packet to the correct destination. The packet format is defined by the application, not this specification. Avalon-ST packets can be variable in length and can be interleaved across a connection. With an Avalon-ST interfaces, the use of packets is optional.

5.2 Avalon-ST Interface Signals

Each signal in an Avalon-ST source or sink interface corresponds to one Avalon-ST signal role. An Avalon-ST interface may contain only one instance of each signal role. All Avalon-ST signal roles apply to both sources and sinks and have the same meaning for both.

Table 5-1: Avalon-ST Interface Signals

In the following table, all signal roles are active high.

Signal Role	Width	Direction	Description
Fundamental Signals			

Signal Role	Width	Direction	Description
`channel`	1 – 128	Source → Sink	The `channel` number for data being transferred on the current cycle. If an interface supports the channel signal, it must also define the `maxChannel` parameter.
`data`	1 – 4,096	Source → Sink	The `data` signal from the source to the sink, typically carries the bulk of the information being transferred. The contents and format of the `data` signal is further defined by parameters.
`error`	1 – 256	Source → Sink	A bit mask used to mark errors affecting the data being transferred in the current cycle. A single bit in `error` is used for each of the errors recognized by the component, as defined by the `errorDescriptor` property.
`ready`	1	Sink → Source	Asserted high to indicate that the sink can accept data. `ready` is asserted by the sink on cycle <n> to mark cycle <*n* + `readyLatency`> as a ready cycle. The source may only assert `valid` and transfer data during `ready` cycles. Sources without a `ready` input cannot be backpressured. Sinks without a `ready` output never need to backpressure.
`valid`	1	Source → Sink	Asserted by the source to qualify all other source to sink signals. The sink samples data other source-to-sink signals on ready cycles where `valid` is asserted. All other cycles are ignored. Sources without a `valid` output implicitly provide valid data on every cycle that they are not being backpressured. Sinks without a `valid` input expect valid data on every cycle that they are not backpressuring.
			Packet Transfer Signals
`empty`	1 – 8	Source → Sink	Indicates the number of symbols that are empty during cycles that contain the end of a packet. The empty signal is not used on interfaces where there is one symbol per beat. If `endofpacket` is not asserted, this signal is not interpreted.
`endofpacket`	1	Source → Sink	Asserted by the source to mark the end of a packet.
`startofpacket`	1	Source → Sink	Asserted by the source to mark the beginning of a packet.

5.3 Signal Sequencing and Timing

5.3.1 Synchronous Interface

All transfers of an Avalon-ST connection occur synchronous to the rising edge of the associated clock signal. All outputs from a source interface to a sink interface, including the `data`, `channel`, and `error` signals, must be registered on the rising edge of clock. Inputs to a sink interface do not have to be registered. Registering signals at the source facilitates high frequency operation while eliminating back-to-back registers with no intervening logic.

5.3.2 Clock Enables

Avalon-ST components typically do not include a clock enable input. The Avalon-ST signaling itself is sufficient to determine the cycles that a component should and should not be enabled. Avalon-ST compliant components may have a clock enable input for their internal logic. But, they must ensure that the timing of the interface control signals still adheres to the protocol.

5.4 Avalon-ST Interface Properties

Table 5-2: Avalon-ST Interface Properties

Property Name	Default Value	Legal Values	Description
`symbolsPerBeat`	1	1 – 32	The number of symbols that are transferred on every valid cycle.
`associatedClock`	1	Clock interface	The name of the Avalon Clock interface to which this Avalon-ST interface is synchronous.
`associatedReset`	1	Reset interface	The name of the Avalon Reset interface to which this Avalon-ST interface is synchronous.
`beatsPerCycle`	8		Specifies the number of beats that are transferred in a single cycle.
`dataBitsPerSymbol`	8	1 – 512	Defines the number of bits per symbol. For example, byte-oriented interfaces have 8-bit symbols. This value is not restricted to be a power of 2.
`emptyWithinPacket`	false	true, false	When true, identifies invalid data between the `startofpacket` and `endofpacket` signals.
`errorDescriptor`	0	List of strings	A list of words that describe the error associated with each bit of the error signal. The length of the list must be the same as the number of bits in the error signal. The first word in the list applies to the highest order bit. For example, “`crc, overflow`" means that bit[1] of `error` indicates a CRC error. Bit[0] indicates an overflow error.
`firstSymbolInHighOrderBits`	true	true, false	When true, the first-order symbol is driven to the most significant bits of the data interface. The highest-order symbol is labeled `D0` in this specification. When this property is set to false, the first symbol appears on the low bits. D0 appears at `data[7:0]`.
`maxChannel`	0	0 – 255	The maximum number of channels that a data interface can support.

Property Name	Default Value	Legal Values	Description
`readyLatency`	0	0 – 8	Defines the relationship between assertion and deassertion of `ready` and cycles which are considered to be available for data transfer. The `readyLatency` is defined separately for each interface.

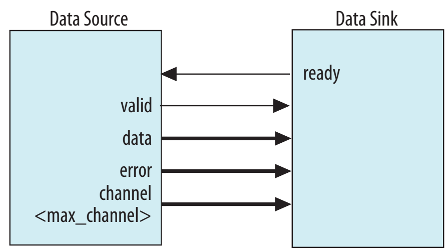
5.5 Typical Data Transfers

This section defines the transfer of data from a source interface to a sink interface. In all cases, the data source and the data sink must comply with the specification. It is not the responsibility of the data sink to detect source protocol errors.

5.6 Signal Details

The following figure shows the signals that are typically included in an Avalon-ST interface. As this figure indicates, a typical Avalon-ST source interface drives the `valid`, `data`, `error`, and `channel` signals to the sink. The sink can apply backpressure using the `ready` signal.

Figure 5-2: Typical Avalon-ST Interface Signals



Here are more details about these signals:

- `ready`—On interfaces supporting backpressure, the sink asserts `ready` to mark the cycles where transfers may take place. If `ready` is asserted on cycle <n>, cycle <*n* + `readyLatency`> is considered a ready cycle.
- `valid`—The `valid` signal qualifies valid data on any cycle where data is being transferred from the source to the sink. On each valid cycle the `data` signal and other source to sink signals are sampled by the sink.
- `data`—The `data` signal typically carries the bulk of the information being transferred from the source to the sink. The data signal consists of one or more symbols being transferred on every clock cycle. The `dataBitsPerSymbol` parameter defines how the data signal is divided into symbols.
- `error`—Errors are signaled with the `error` signal, where each bit in `error` corresponds to a possible error condition. A value of 0 on any cycle indicates the data on that cycle is error-free. The action that a component takes when an error is detected is not defined by this specification.

- `channel`—The optional `channel` signal is driven by the source to indicate the channel to which the data belongs. The meaning of `channel` for a given interface depends on the application. Some applications use `channel` as a port number indication. Other applications use `channel` as a page number or timeslot indication. When the `channel` signal is used, all of the data transferred in each active cycle belongs to the same channel. The source may change to a different channel on successive active cycles.

  An interface that uses the `channel` signal must define the `maxChannel` parameter to indicate the maximum channel number. If the number of channels an interface supports changes dynamically, `maxChannel` is the maximum number the interface can support.

5.7 Data Layout

Figure 5-3: Data Symbols

The following figure shows a 64-bit data signal with `dataBitsPerSymbol=16`. Symbol 0 is the most significant symbol.

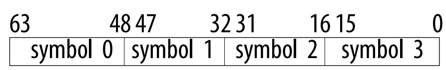
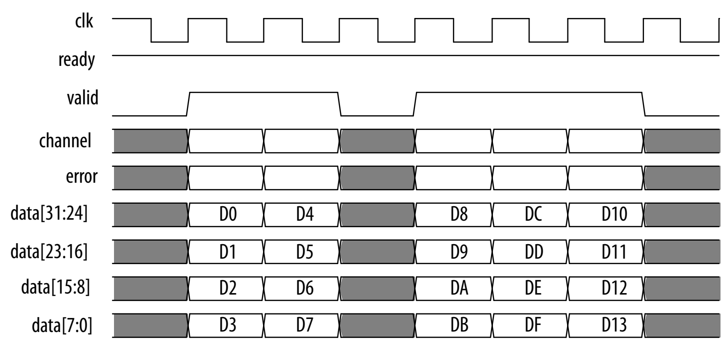


Figure 5-4: Layout of Data

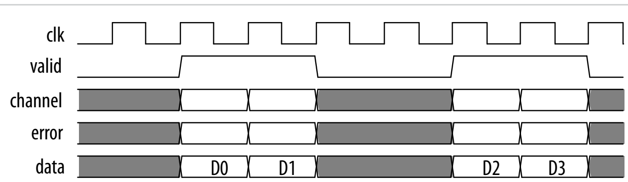
The timing diagram in the following figure shows a 32-bit example where `dataBitsPerSymbol=8`.



5.8 Data Transfer without Backpressure

The data transfer without backpressure is the most basic of Avalon-ST data transfers. On any given clock cycle, the source interface drives the `data` and the optional `channel` and `error` signals, and asserts `valid`. The sink interface samples these signals on the rising edge of the reference clock if `valid` is asserted.

Figure 5-5: Data Transfer without Backpressure



5.9 Data Transfer with Backpressure

The sink asserts `ready` for a single clock cycle to indicate it is ready for an active cycle. Cycles during which the sink is ready for data are called ready cycles. During a ready cycle, the source may assert `valid` and provide data to the sink. If it has no data to send, it deasserts `valid` and can drive `data` to any value.

Each interface that supports backpressure defines the `readyLatency` parameter to indicate the number of cycles from the time that ready is asserted until valid data can be driven. If the `readyLatency` is nonzero, cycle <*n* + *readyLatency*> is a ready cycle if `ready` is asserted on cycle <*n*>. Any interface that includes the `ready` signal and defines the `readyLatency` parameter supports backpressure.

When `readyLatency = 0`, data is transferred only when `ready` and `valid` are asserted on the same cycle. In this mode, the source does not receive the sink's ready signal before it begins sending valid data. The source provides the data and asserts `valid` whenever it can. The source waits for the sink to capture the data and assert `ready`. The source can change the data it is providing at any time. The sink only captures input data from the source when `ready` and `valid` are both asserted.

When `readyLatency >= 1`, the sink asserts `ready` before the `ready` cycle itself. The source can respond during the appropriate cycle by asserting `valid`. It may not assert `valid` during a cycle that is not a `ready` cycle.

Figure 5-6: Avalon-ST Interface with readyLatency = 4

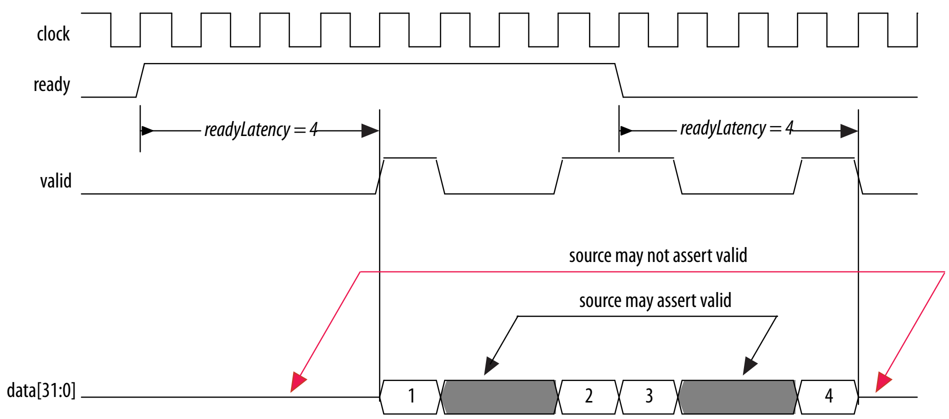


Figure 5-7: Transfer with Backpressure, readyLatency=0

The following figure illustrates these events:

1. The source provides data and asserts `valid` on cycle 1, even though the sink is not ready.
2. The source waits until cycle 2, when the sink does assert `ready`, before moving onto the next data cycle.
3. In cycle 3, the source drives data on the same cycle and the sink is ready to receive it. The transfer occurs immediately.
4. In cycle 4, the sink asserts `ready`, but the source does not drive valid data.

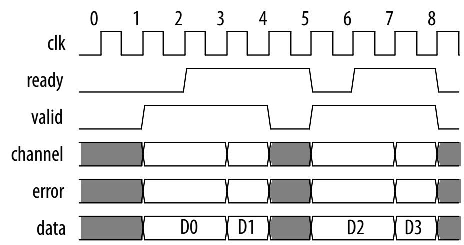


Figure 5-8: Transfer with Backpressure, readyLatency=1

The following figures show data transfers with `readyLatency=1` and `readyLatency=2`, respectively. In both these cases, `ready` is asserted before the ready cycle, and the source responds 1 or 2 cycles later by providing data and asserting `valid`. When `readyLatency` is not 0, the source must deassert `valid` on non-ready cycles.

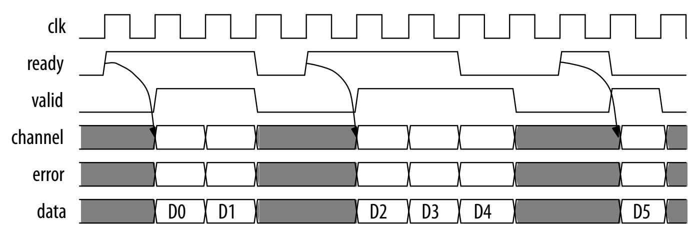
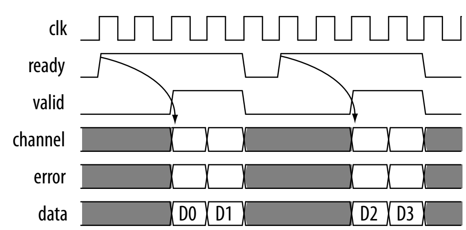


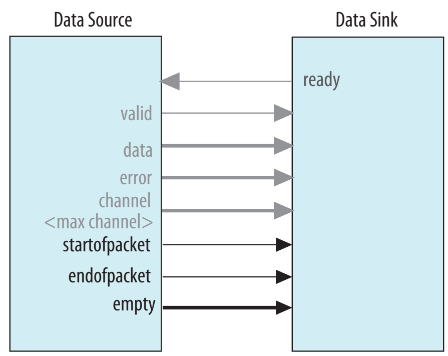
Figure 5-9: Transfer with Backpressure, readyLatency=2



5.10 Packet Data Transfers

The packet transfer property adds support for transferring packets from a source interface to a sink interface. Three additional signals are defined to implement the packet transfer. Both the source and sink interfaces must include these additional signals to support packets. No automatic adaptation creates connections between source and sink interfaces with and without packet support.

Figure 5-10: Avalon-ST Packet Interface Signals



5.11 Signal Details

- `startofpacket`—All interfaces supporting packet transfers require the `startofpacket` signal. `startofpacket` marks the active cycle containing the start of the packet. This signal is only interpreted when `valid` is asserted.
- `endofpacket`—All interfaces supporting packet transfers require the `endofpacket` signal. `endofpacket` marks the active cycle containing the end of the packet. This signal is only interpreted when `valid` is asserted. `startofpacket` and `endofpacket` can be asserted in the same cycle. No idle cycles are required between packets. The `startofpacket` signal can follow immediately after the previous `endofpacket` signal.
- `empty`—The optional `empty` signal indicates the number of symbols that are empty the `endofpacket` cycle. The sink only checks the value of the `empty` during active cycles that have `endofpacket` asserted. The empty symbols are always the last symbols in `data`, those carried by the low-order bits when `firstSymbolInHighOrderBits` = true. The `empty` signal is required on all packet interfaces whose `data` signal carries more than one symbol of data and have a variable length packet format. The size of the `empty` signal in bits is $\log_2$`(<symbols per cycle>)`.

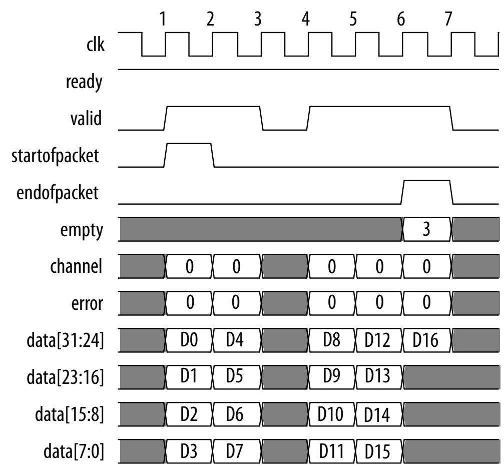
5.12 Protocol Details

Packet data transfer follows the same protocol as the typical data transfer with the addition of the `startofpacket`, `endofpacket`, and `empty`.

Figure 5-11: Packet Transfer

The following figure illustrates the transfer of a 17-byte packet from a source interface to a sink interface, where `readyLatency`=0. It illustrates the following events:

1. Data transfer occurs on cycles 1, 2, 4, 5, and 6, when both `ready` and `valid` are asserted.
2. During cycle 1, `startofpacket` is asserted. The first 4 bytes of packet are transferred.
3. During cycle 6, `endofpacket` is asserted. `empty` has a value of 3. This value indicates that this is the end of the packet and that 3 of the 4 symbols are empty. In cycle 6, the high-order byte, data[31:24] drives valid data.



Avalon Conduit Interfaces 6

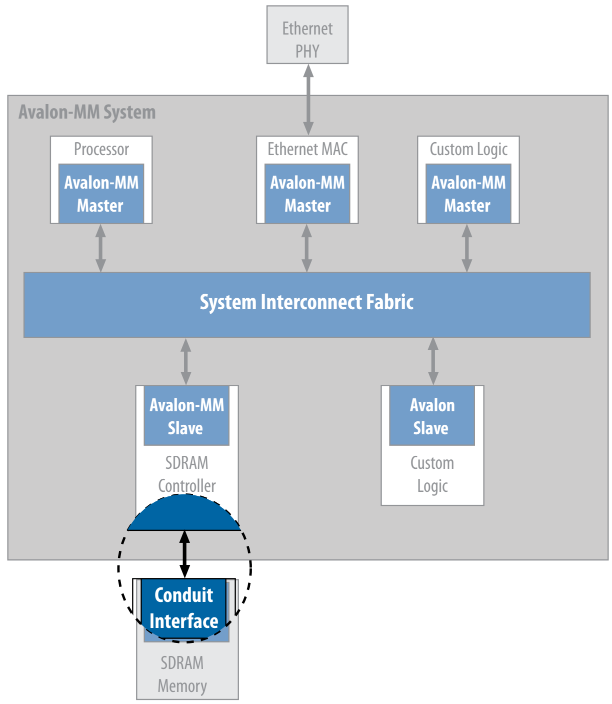
Subscribe Send Feedback

Avalon Conduit interfaces group an arbitrary collection of signals. You can specify any role for conduit signals. However, when you connect conduits, the roles and widths must match and the directions must be opposite. An Avalon Conduit interface can include input, output, and bidirectional signals. A module can have multiple Avalon Conduit interfaces to provide a logical signal grouping.

Note: If possible, you should use the standard Avalon-MM or Avalon-ST interfaces instead of creating an Avalon Conduit interface. Qsys provides validation and checking for these interfaces. It cannot provide validation for Avalon Conduit interfaces. If the signals in your conduit change clock domains between the endpoints, Qsys cannot check or adapt to that.

Signals that interface to the SDRAM, such as address, data and control signals, form an Avalon Conduit interface.

Figure 6-1: Focus on the Conduit Interface



6.1 Conduit Signals

Table 6-1: Conduit Signal Roles

Signal Role	Width	Direction	Description
<any>	<*n*>	In, out, or bidirectional	A conduit interface consists of one or more input, output, or bidirectional signals of arbitrary width. Conduits can have any user-specified role. You can connect compatible Conduit interfaces inside a Qsys system provided the roles and widths match and the directions are opposite.

6.2 Conduit Properties

There are no properties for conduit interfaces.

Avalon Tristate Conduit Interface

7

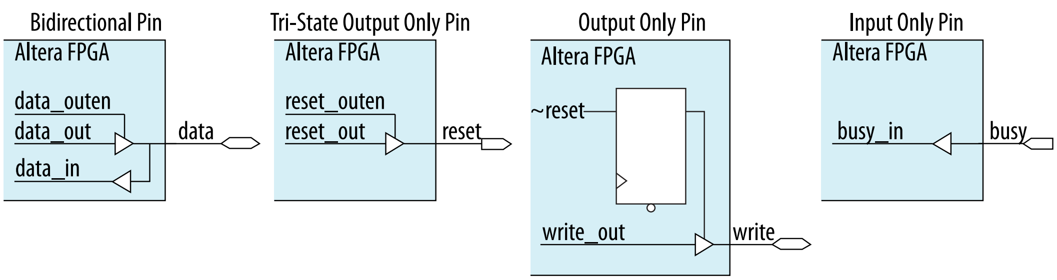
Subscribe Send Feedback

The Avalon Tristate Conduit Interface (Avalon-TC) is a point-to-point interface designed for on-chip controllers that drive off-chip components. This interface allows data, address, and control pins to be shared across multiple tristate devices. Sharing conserves pins in systems that have multiple external memory devices.

The Avalon-TC interface restricts the more general Avalon Conduit Interface in two ways:

- The Avalon-TC requires `request` and `grant` signals. These signals enable bus arbitration when multiple Tristate Conduit Masters (TCM) are requesting access to a shared bus.
- The pin type of a signal must be specified using suffixes appended to a signal's role. The three suffixes are: `_out`, `_in`, and `_outen`. Matching role prefixes identify signals are share the same I/O Pin. The following illustrates the naming conventions for Avalon-TC shared pins.

Figure 7-1: Shared Pin Types



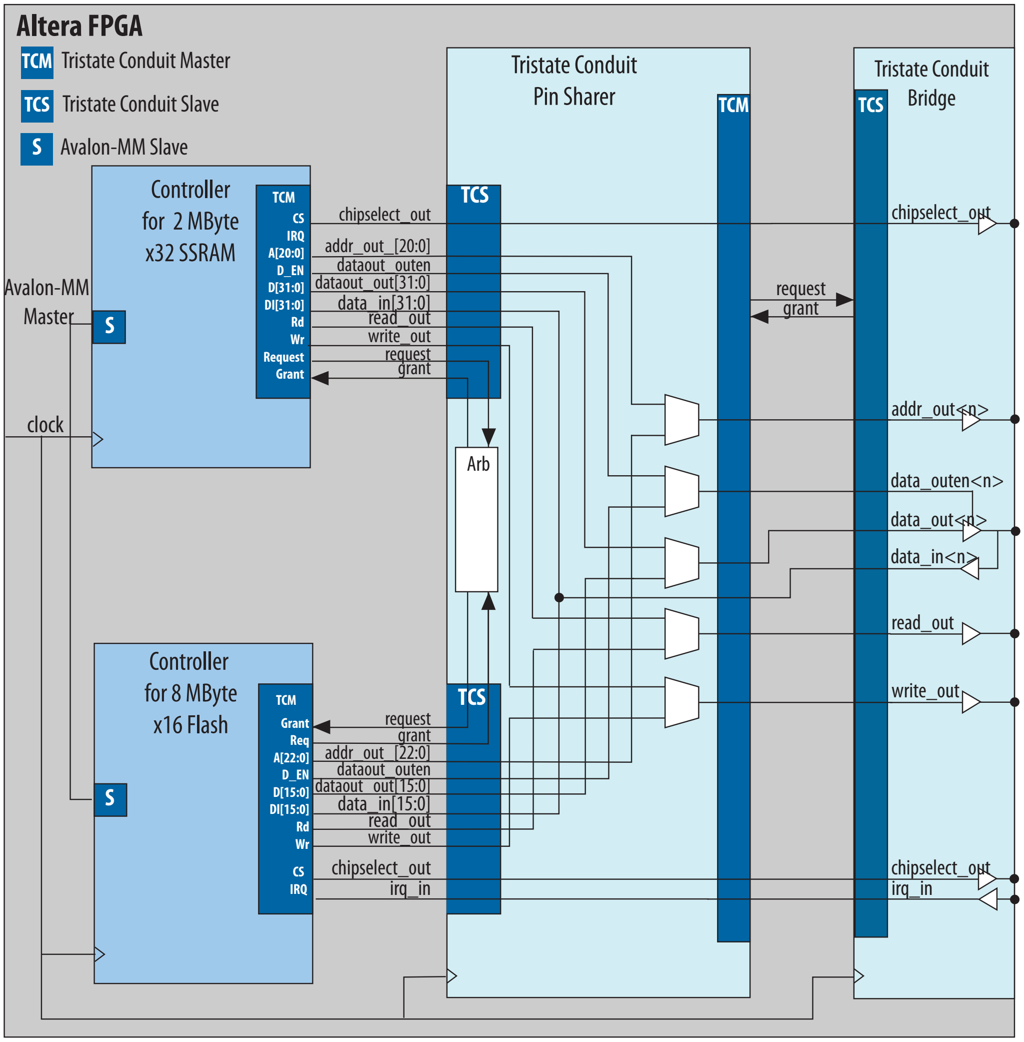
The next figure illustrates pin sharing using Avalon-TC interfaces. This figure illustrates the following points.

- The Tristate Conduit Pins Sharer includes separate Tristate Conduit Slave Interfaces for each Tristate Conduit Master. Each master and slave pair has its own `request` and `grant` signals.
- The Tristate Conduit Pin Sharer identifies signals with identical roles as tristate signals that share the same FPGA pin. In this example, the following signals are shared: `addr_out`, `data_out`, `data_in`, `read_out`, and `write_out`.
- The Tristate Conduit Pin Sharer drives a single bus including all of the shared signals to the Tristate Conduit Bridge. If the widths of shared signals differ, the Tristate Conduit Pin Sharer aligns them on their 0^{th} bit. It drives the higher-order pins to 0 whenever the smaller signal has control of the bus.
- Signals that are not shared propagate directly through the Tristate Conduit Pin Sharer. In this example, the following signals are not shared: `chipselect0_out`, `irq0_out`, `chipselect1_out`, and `irq1_out`.

- All Avalon-TC interfaces connected to the same Tristate Conduit Pin Sharer must be in the same clock domain.

Figure 7-2: Tristate Conduit Interfaces

The following illustrates the typical use of Avalon-TC Master and Slave interfaces and signal naming.



For more information about the Generic Tristate Controller and Tristate Conduit Pin Sharer, refer to the *Avalon Tristate Conduit Components User Guide*.

Related Information

[Avalon Tristate Conduit Components User Guide]

7.1 Tristate Conduit Signals

The following table lists the signal defined for the Avalon-TC interface. All Avalon-TC signals apply to both masters and slaves and have the same meaning for both

Table 7-1: Tristate Conduit Interface Signal Roles

Signal Role	Width	Direction	Required	Description
`request`	1	Master → Slave	Yes	The meaning of `request` depends on the state of the `grant` signal, as the following rules dictate. When `request` is asserted and `grant` is deasserted, `request` is requesting access for the current cycle. When `request` is asserted and `grant` is asserted, `request` is requesting access for the next cycle. Consequently, `request` should be deasserted on the final cycle of an access. The `request` is deasserted in the last cycle of a bus access. It can be reasserted immediately following the final cycle of a transfer. This protocol makes both rearbitration and continuous bus access possible if no other masters are requesting access. Once asserted, `request` must remain asserted until granted. Consequently, the shortest bus access is 2 cycles. Refer to *Tristate Conduit Arbitration Timing* for an example of arbitration timing.
`grant`	1	Slave → Master	Yes	When asserted, indicates that a tristate conduit master has been granted access to perform transactions. `grant` is asserted in response to the `request` signal. It remains asserted until 1 cycle following the deassertion of `request`. The design of the Avalon-TC Interface does not allow a default Avalon-TC master to be granted when no masters are requesting.
`<name>_in`	1 – 1,024	Slave → Master	No	The input signal of a logical tristate signal.
`<name>_out`	1 – 1,024	Master → Slave	No	The output signal of a logical tristate signal.
`<name>_outen`	1	Master → Slave	No	The output enable for a logical tristate signal.

7.2 Tristate Conduit Properties

There are no special properties for the Avalon-TC Interface.

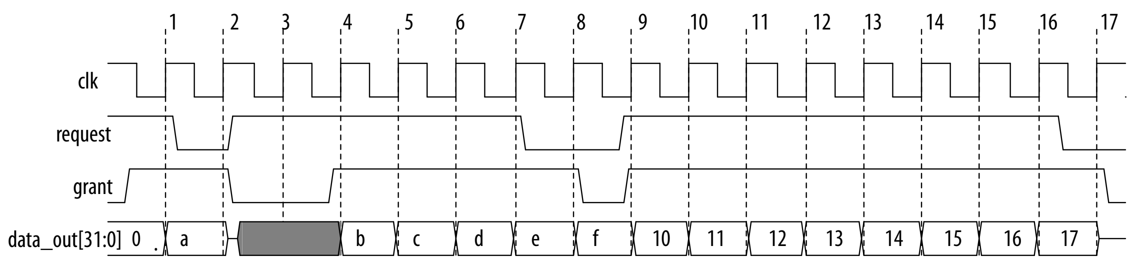
7.3 Tristate Conduit Timing

The following illustrates arbitration timing for the Tristate Conduit Pin Sharer. Note that a device can drive or receive valid data in the granted cycle.

Figure 7-3: Tristate Conduit Arbitration Timing

This figure shows the following sequence of events:

1. In cycle one, the tristate conduit master asserts grant. The granted slave drives valid data in cycles one and two.
2. In cycle 4, the tristate conduit master asserts grant. The granted slave drives valid data in cycles 4–7.
3. In cycle 8, the tristate conduit master asserts grant. The granted slave drives valid data in cycles 8–16.
4. Cycle 3 is the only cycle that does not contain valid data.



Additional Information 8

Subscribe Send Feedback

Table 8-1: Document Revision History

Date	Version	Changes

June 2014	14.0	• Updated the Avalon-MM Signals table, `begintransfer`, `readdatavalid`, and `readdatavalid_n`.
		• Updated the Read and Write Transfers with Waitrequest figure:
		  • Moved deassertion of write to cycle 6.
		  • Moved assertion of `readdatavalid` and `readdata` to cycle 4.
		• Updated the Pipelined Read Transfers with Variable Latency figure:
		  • Moved assertion of `data1` to just after cycle 5, and assertion of `data2` to cycle 6.
		  • Moved assertion of `readdatavalid` to match `data1` and `data2`.

April 2014	13.01	Corrected *Read and Write Transfers with Waitrequest* In *Avalon Memory-Mapped Interfaces* chapter .
May 2013	13.0	Made the following changes: • Minor updates to *Avalon Memory-Mapped Interfaces*. • Minor updates to *Avalon Streaming Interfaces*. • Updated *Avalon Conduit Interfaces* to describe the signal roles supported by Avalon conduit interfaces. • Updated *Shared Pin Types* figure in the *Avalon Tristate Conduit Interface* chapter.
May 2011	11.0	Initial release of the *Avalon Interface Specifications* supported by Qsys.

8.1 How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	
Product literature	Website	www.altera.com/literature
Non-technical support (General)	Email	
(Software Licensing)	Email	

Note to Table:

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

8.2 Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	Indicates variables. For example, n + 1. Variable names are enclosed in angle brackets (< >). For example, <*file name*> and <*project name*>**.pof** file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. The suffix `n` denotes an active-low signal. For example, `reset_n`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
r	An angled arrow instructs you to press the Enter key.

Visual Cue	Meaning
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
1	The hand points to information that requires special attention.
h	A question mark directs you to a software help system with related information.
f	The feet direct you to another document or website with related information.
c	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
w	A warning calls attention to a condition or possible situation that can cause you injury.