Avalon Interface Specifications

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

MNL-AVABUSREF-2.1

Document last updated for Altera Complete Design Suite version: 13.0
Document publication date: May 2013

Feedback Subscribe

Contents

Chapter 1. Introduction

1.1. Avalon Properties and Parameters 1–4
1.2. Signal Roles 1–4
1.3. Interface Timing 1–4
1.4. Related Documents 1–5

Chapter 2. Avalon Clock and Reset Interfaces

2.1. Clock Sink 2–1
2.1.1. Clock Sink Signal Roles 2–1
2.1.2. Clock Sink Properties 2–1
2.1.3. Associated Clock Interfaces 2–2
2.2. Clock Source 2–2
2.2.1. Clock Source Signal Roles 2–2
2.2.2. Clock Source Properties 2–2
2.3. Reset Interfaces 2–3
2.3.1. Reset Sink 2–3
2.3.2. Reset Sink Interface Properties 2–3
2.3.3. Associated Reset Interfaces 2–3
2.3.4. Reset Source 2–3
2.3.5. Reset Source Interface Properties 2–4

Chapter 3. Avalon Memory-Mapped Interfaces

3.1. Introduction 3–1
3.2. Signals 3–2
3.3. Interface Properties 3–5
3.4. Timing 3–7
3.5. Transfers 3–8
3.5.1. Typical Read and Write Transfers 3–8
3.5.2. Read and Write Transfers with Fixed Wait-States 3–9
3.5.3. Pipelined Transfers 3–10
3.5.3.1. Pipelined Read Transfer with Variable Latency 3–11
3.5.3.2. Pipelined Read Transfers with Fixed Latency 3–12
3.5.4. Burst Transfers 3–12
3.5.4.1. Write Bursts 3–13
3.5.4.2. Read Bursts 3–14
3.5.4.3. Line–Wrapped Bursts 3–15
3.6. Address Alignment 3–15
3.7. Avalon-MM Slave Addressing 3–16

Chapter 4. Avalon Interrupt Interfaces

4.1. Interrupt Sender 4–1
4.1.1. Interrupt Sender Signal Roles 4–1
4.1.2. Interrupt Sender Properties 4–1
4.2. Interrupt Receiver 4–2
4.2.1. Interrupt Receiver Signal Roles 4–2
4.2.2. Interrupt Receiver Properties 4–2
4.2.3. Interrupt Timing 4–3

Chapter 5. Avalon Streaming Interfaces
5.1. Features . . . 5–2
5.2. Terms and Concepts . . . 5–2
5.3. Avalon-ST Interface Signals . . . 5–3
5.4. Signal Sequencing and Timing . . . 5–4
5.4.1. Synchronous Interface . . . 5–4
5.4.2. Clock Enables . . . 5–4
5.5. Avalon-ST Interface Properties . . . 5–4
5.6. Typical Data Transfers . . . 5–5
5.7. Signal Details . . . 5–5
5.8. Data Layout . . . 5–6
5.9. Data Transfer without Backpressure . . . 5–7
5.10. Data Transfer with Backpressure . . . 5–7
5.11. Packet Data Transfers . . . 5–9
5.12. Signal Details . . . 5–10
5.13. Protocol Details . . . 5–10

Chapter 6. Avalon Conduit Interfaces
6.1. Signals . . . 6–2
6.2. Properties . . . 6–2

Chapter 7. Avalon Tristate Conduit Interface
7.1. Tristate Conduit Signals . . . 7–3
7.2. Tristate Conduit Properties . . . 7–3
7.3. Tristate Conduit Timing . . . 7–3

Additional Information
Document Revision History . . . Info–1
How to Contact Altera . . . Info–1
Typographic Conventions . . . Info–2

1. Introduction

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 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 Figure 1–1, the Ethernet Controller includes six different interface types: Avalon-MM, Avalon-ST, Avalon Conduit, Avalon-TC, Avalon Interrupt, and Avalon Clock.

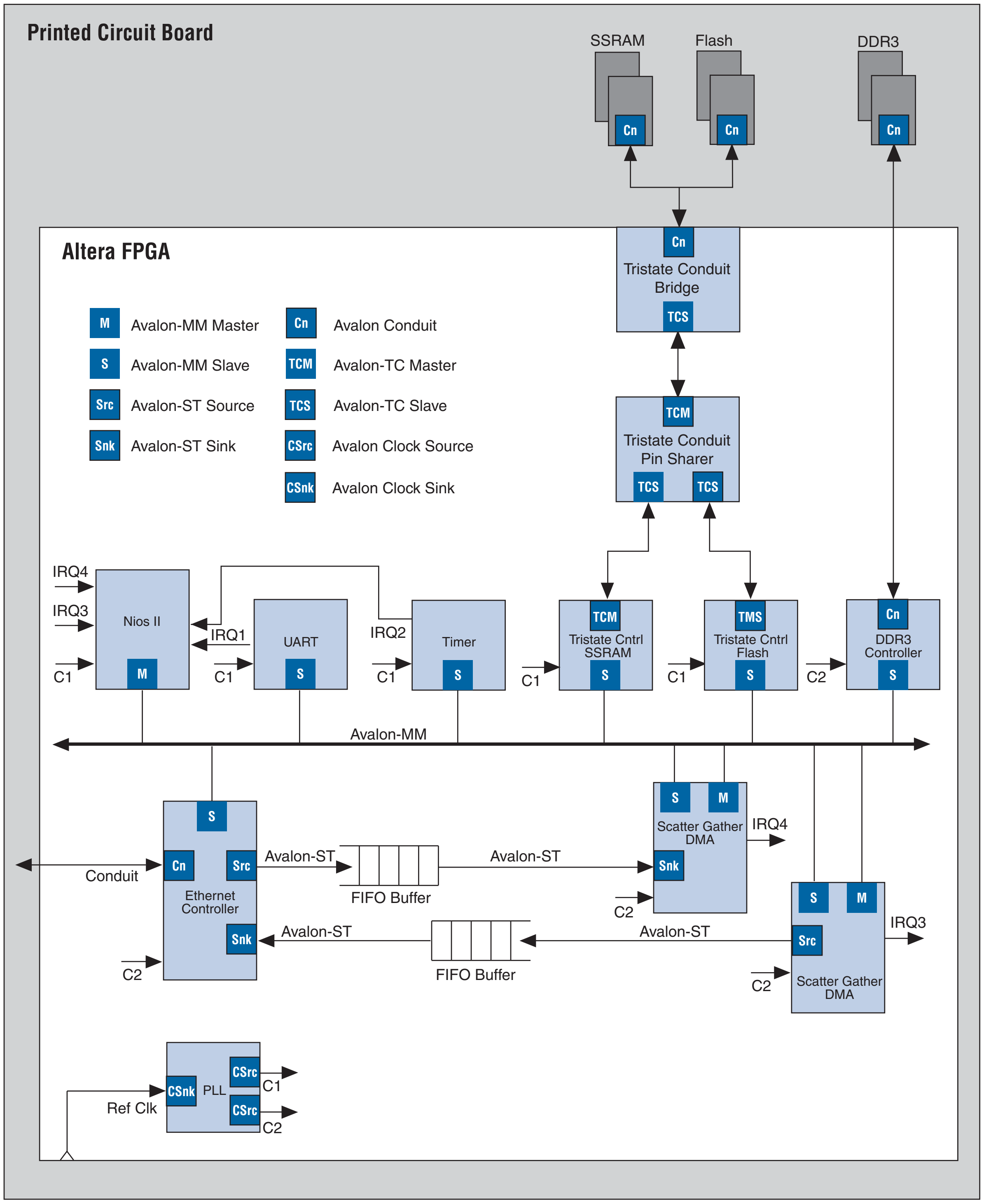
Avalon interfaces are an open standard. No license or royalty is required to develop and sell products that use, or are based on Avalon interfaces.

This specification describes the behavior of the Avalon interfaces supported in Qsys. It supersedes version 1.3 of the *Avalon Interface Specifications* which describes the behavior of Avalon interfaces supported in SOPC Builder.

For more information about the differences between Avalon interfaces supported in Qsys and SOPC Builder, refer to *AN 632: SOPC Builder to Qsys Migration Guidelines*.

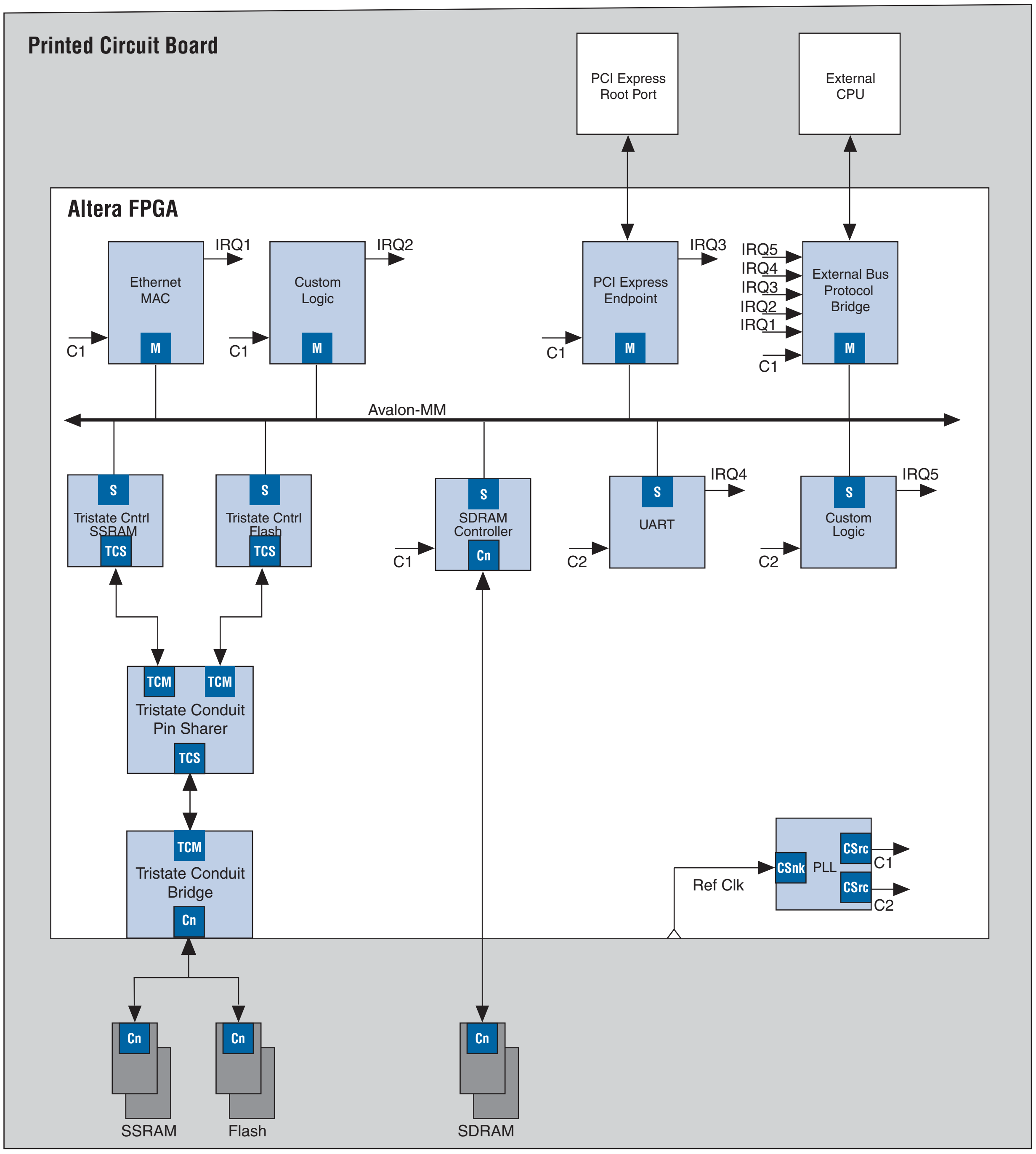
Figure 1–1 and Figure 1–2 illustrate the use of the Avalon interfaces in system designs.

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



In Figure 1–1, 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 Figure 1–2, 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, allowing component designers 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 because 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

You can find additional information on related topics in the following documents and design examples:

- *System Design with Qsys* in volume 1 of the *Quartus II Handbook*.

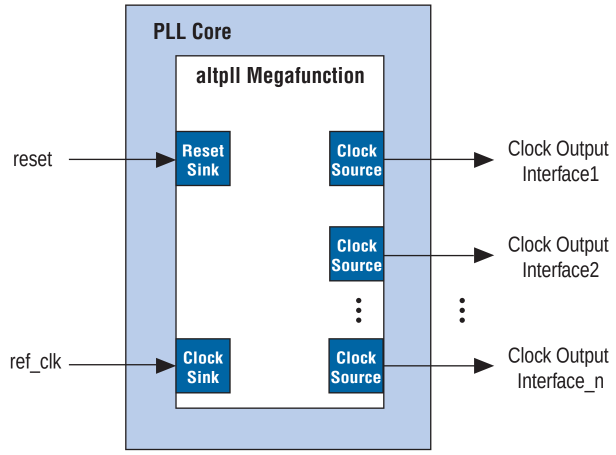
  This section includes the following chapters:

  - *Creating a System with Qsys*—provides an overview of the Qsys system integration tool, including an introduction to hierarchical system design.
  - *Creating Qsys Components*—introduces Qsys components and the Qsys component library. It also provides an overview of the Qsys component editor which you can use to define custom components.
  - *Qsys Interconnect*—discusses the Qsys interconnect, a high-bandwidth structure for connecting components that use Avalon interfaces.
  - *Component Interface Tcl Reference*—describes an alternative method for defining Qsys components by declaring their properties and behaviors in a Hardware Component Description File (**_hw.tcl**). This chapter also provides a reference for the Tool Command Language (Tcl) commands that describe Qsys components.
- *AN 632: SOPC Builder to Qsys Migration Guidelines*—discusses issues and guidelines for migrating designs from SOPC Builder to Qsys.
- Qsys Tutorial Design Example—introduces you to system development in Qsys. It builds a memory test system using components with Avalon interfaces to verify a memory subsystem.

2. Avalon Clock and Reset Interfaces

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. Figure 2–1 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

A typical component includes a clock sink input to provide a timing reference for other interfaces and internal logic.

2.1.1. Clock Sink Signal Roles

Table 2–1 lists the clock input signals.

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.

2.1.2. Clock Sink Properties

Table 2–2 lists the properties of clock inputs.

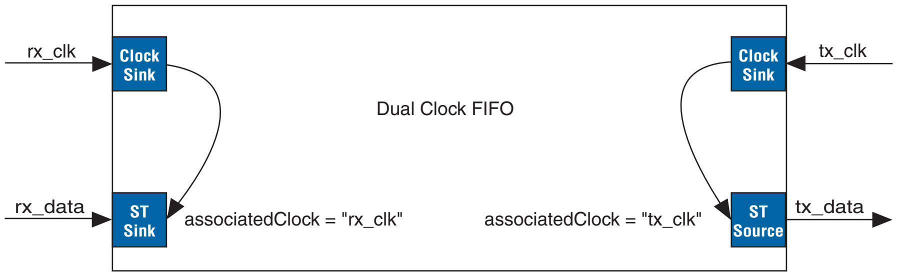
Table 2–2. Clock Sink Properties

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.

2.1.3. 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 Figure 2–2.

Figure 2–2. associatedClock Property



2.2. Clock Source

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

2.2.1. Clock Source Signal Roles

Table 2–3 lists the clock source signals.

Table 2–3. Clock Source Signal Roles

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

2.2.2. Clock Source Properties

Table 2–4 lists the properties of clock outputs.

Table 2–4. Clock Source Properties

Name	Default Value	Legal Values	Description
`associatedDirectClock`	—	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.
`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.

2.3. Reset Interfaces

Avalon Reset interfaces provide both soft and hard reset functionality. Soft reset logic typically reinitializes registers and memories without powering down the device. Hard reset logic initializes the device after power-on.

The following sections describe the properties and signal roles for reset interfaces.

2.3.1. Reset Sink

Table 2–5 lists the reset input signals.

Table 2–5. Reset Input Signal Roles

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.

2.3.2. Reset Sink Interface Properties

Table 2–6 lists the properties of resets.

Table 2–6. Reset Interface Properties

Name	Default Value	Legal Values	Description
`associatedClock`	—	a clock name	The name of a clock to which this interface synchronized. Required if the value of `synchronousEdges` is `DEASSERT` or `BOTH`.

`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.

2.3.3. Associated Reset Interfaces

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

2.3.4. Reset Source

Table 2–7 lists the reset input signals.

Table 2–7. Reset Output Signal Roles

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.

2.3.5. Reset Source Interface Properties

Table 2–6 lists the properties of resets.

Table 2–8. Reset Interface Properties

Name	Default Value	Legal Values	Description
`associatedClock`	—	a clock name	The name of a clock to which this interface synchronized. Required if the value of `synchronousEdges` is `DEASSERT` or `BOTH`.
`associatedDirectReset`	—	a reset name	The name of the reset input that directly drives this reset source through a one-to-one link.
`associatedResetSinks`	—	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.

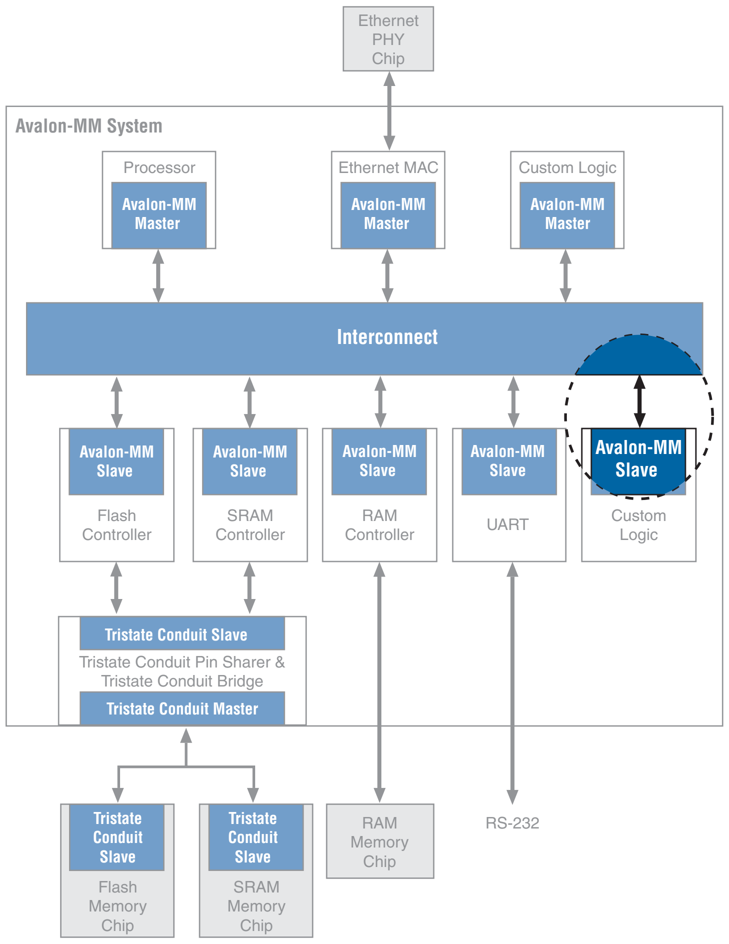
3. Avalon Memory-Mapped Interfaces

3.1. Introduction

Avalon Memory-Mapped (Avalon-MM) interfaces are used for read and write interfaces on master and slave components in a memory-mapped system. These components include microprocessors, memories, UARTs, DMAs, and timers which have master and slave interfaces connected by an interconnect fabric. Avalon-MM interfaces can describe a wide variety of component interfaces, from SRAM interfaces which support simple, fixed-cycle read and write transfers to more complex, pipelined interfaces capable of burst transfers.

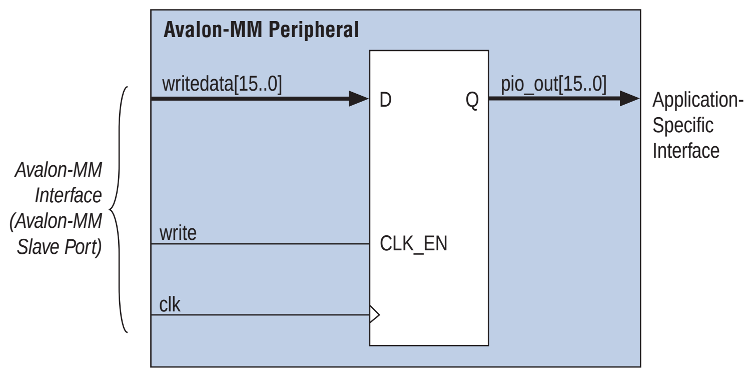
Figure 3–1 shows a typical system, highlighting the Avalon-MM slave interface connection to the interconnect fabric.

Figure 3–1. Focus on Avalon-MM Slave Transfers



Avalon-MM components typically include only the signals required for the component logic. The 16-bit general-purpose I/O peripheral shown in Figure 3–2 only responds to write requests, therefore it includes only the slave signals required for write transfers.

Figure 3–2. Example Slave Component


Avalon-MM Peripheral
writedata[15..0]
D
Q
pio_out[15..0]
Application-Specific Interface
Avalon-MM Interface (Avalon-MM Slave Port)
write
CLK_EN
clk

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

Table 3–1 lists the signal roles that constitute the Avalon-MM interface. The signal roles available for Avalon-MM interfaces allow you to create masters that use bursts for both 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` and 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 *(1)* (Part 1 of 4)

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 so that 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 and address 1 selects the second word of the slave. Refer to the `addressUnits` interface property for byte addressing.

Table 3–1. Avalon-MM Signals *(1)* **(Part 2 of 4)**

Signal Role	Width	Direction	Description
`begintransfer`	1	Master → Slave	Asserted by the interconnect for the first cycle of each transfer regardless of `waitrequest` and other signals. If you do not include this signal in your Avalon-MM master interface, Qsys automatically generates this signal for you.
`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 indicates 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, and 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. Doing so 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.
`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.

Table 3–1. Avalon-MM Signals *(1)* (Part 3 of 4)

Signal Role	Width	Direction	Description
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, and is deasserted on the final transaction of a locked sequence of transactions. `lock` assertion does not guarantee that arbitration will be won, but 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 operations, where master A reads 32-bit data that has multiple bit fields, changes one field, and writes the 32-bit data back. If lock is not used, a master B could perform a write between Master A's read and write and master A's write would overwrite master B's changes.
`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`, as illustrated by Figure 3–3 on page 3–9, and `beginbursttransfer`, as illustrated by Figure 3–7 on page 3–14. 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. Asserted by the slave to indicate that the `readdata` signal contains valid data in response to a previous `read` request. A slave with `readdatavalid` must assert this signal for one cycle for each `read` access it has received. There must be at least one cycle of latency between acceptance of the `read` and assertion of `readdatavalid`. Figure 3–5 on page 3–11 illustrates the `readdatavalid` signal. Required if the master supports pipelined reads. Bursting masters with read functionality must include the `readdatavalid` signal.

Table 3–1. Avalon-MM Signals *(1)* (Part 4 of 4)

Signal Role	Width	Direction	Description
Burst Signals			
`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, so 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 timing of the `burstcount` signal is controlled by the `constantBurst` property. 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(log`$_2$ *(<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`. Refer to Figure 3–7 on page 3–14 for an example of its use. The interconnect fabric automatically generates this signal for slaves when requested. `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.

Notes to Table 3–1:

(1) 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.

3.3. Interface Properties

Table 3–2 describes the Avalon-MM interface properties.

Table 3–2. Avalon-MM Interface Properties (Part 1 of 3)

Name	Default Value	Legal Values	Description
`addressUnits`	Master - symbols Slave - words	words, symbols	Specifies the unit for addresses, that is, whether to use byte or word addressing (a symbol is typically a byte).
`burstCountUnits`	words	words, symbols	This property specifies the units for the burstcount signal. symbols means that the `burstcount` value is interpreted as the number of symbols (bytes) in the burst. word means that 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 are guaranteed to begin at addresses which are multiples of the burst size in bytes.

Table 3–2. Avalon-MM Interface Properties (Part 2 of 3)

Name	Default Value	Legal Values	Description
`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. The difference between the two is that with a wrapping burst, when the address reaches a burst boundary, the address wraps back to the previous burst boundary such that only the low order bits are required for address counting. For example, a wrapping burst with burst boundaries every 32 bytes across a 32-bit interface to address 0xC would write to addresses 0xC, 0x10, 0x14, 0x18, 0x1C, 0x0, 0x4, and 0x8.
`maximumPendingReadTransactions` *(1)*	1 *(2)*	1 – 64	Slaves: this parameter is the maximum number of pending reads that the slave can queue. Refer to Figure 3–5 on page 3–11 for a timing diagram that uses this property. 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. Refer to Figure 3–6 on page 3–12 for a timing diagram that uses this property.
`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.

Table 3–2. Avalon-MM Interface Properties (Part 3 of 3)

Name	Default Value	Legal Values	Description
`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 (depending on the `timingUnits` parameter) for asynchronous devices. Almost all Avalon-MM slave devices are synchronous. One example of a device that requires asynchronous timing is an Avalon-MM slave that reads and writes an off-chip bidirectional port. That off-chip device might have a fixed settling time for bus turnaround.
`writeWaitTime` *(1)*	0	0 – 1000 Cycles	For interfaces that do not use the `waitrequest` signal, `writeWaitTime` indicates the number of cycles or nanoseconds (depending on the `timingUnits` parameter) before a slave accepts a write. The timing is as if the slave asserted `waitrequest` for `writeWaitTime` cycles or nanoseconds. Refer to Figure 3–4 on page 3–10 for a timing diagram that uses this property.
Interface Relationship Properties			
`associatedClock`	—	—	Name of the clock interface to which this Avalon-MM interface is synchronous.
`associatedReset`	—	—	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 to Table 3–2:

(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 document does not dictate how or when signals transition between clock edges and 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, 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's control and data signals pass through the interconnect fabric and interact with the slave port.

3.5.1. Typical Read and Write Transfers

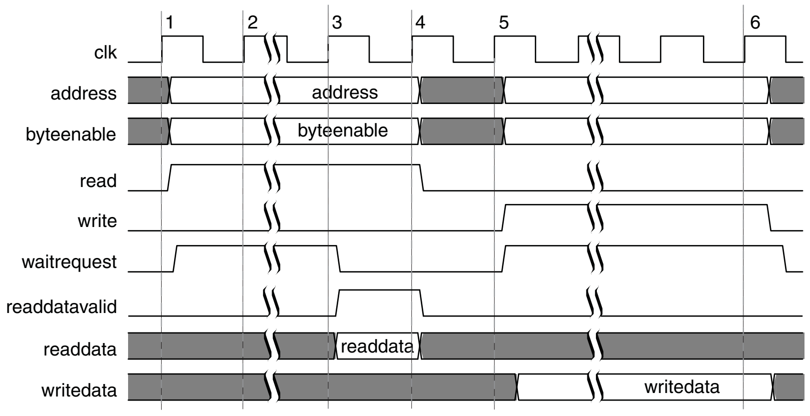
This section describes a typical Avalon-MM interface that supports read and write transfers with slave-controlled `waitrequest`. The slave can stall the interconnect for as many cycles as required by asserting the `waitrequest` signal. If a slave uses `waitrequest` for either read or write transfers,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

If a slave receives `address`, `byteenable`, `read` or `write`, and `writedata` after the rising edge of the clock, the slave port must assert `waitrequest` before the next rising clock edge to hold off the 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. Figure 3–3 shows `read` and `write` transfers using `waitrequest` for a system in which the master and slave both have a `readdatavalid` signal.

waitrequest can be decoupled from the read and write request signals so that it 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 by eliminating a combinational loop including the read, write, and waitrequest signals.

Figure 3–3. Read and Write Transfers with Waitrequest



Notes to Figure 3–3:

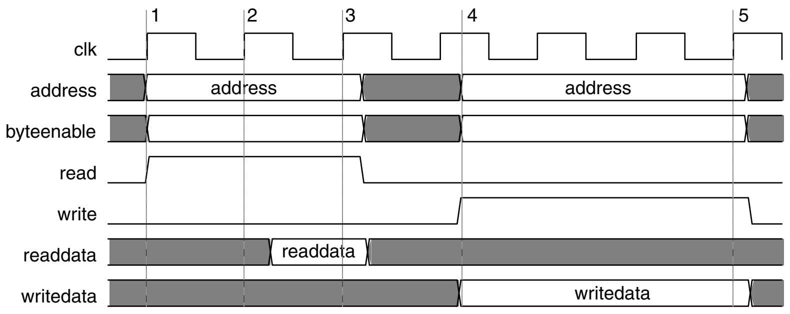
(1) address, read, and begintransfer are asserted after the rising edge of clk. waitrequest is asserted stalling the transfer.
(2) waitrequest is sampled. Because waitrequest is asserted, the cycle becomes a wait-state, and address, read, write, and byteenable remain constant.
(3) The slave presents valid readdata, asserts readdatavalid, and 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 captures writedata and deasserts waitrequest, ending the transfer.

3.5.2. Read and Write Transfers with Fixed Wait-States

Instead of using waitrequest to hold off a transfer, a slave can specify fixed wait-states using the readWaitTime and writeWaitTime properties. The address and control signals (byteenable, read, and write) are held constant for the duration of the transfer. The read/write timing with readWaitTime/writeWaitTime set to <*n*> is exactly the same as asserting waitrequest for <*n*> cycles per transfer.

Figure 3–4 shows an example slave read and write transfers with `writeWaitTime = 2` and `readWaitTime = 1`.

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



Notes to Figure 3–4:

(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 because the `readWaitTime` is 1.

(3) The slave captures `readdata` on the rising edge of `clk`, and 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 can capture address and control signals on the rising edge of `clk`, and has one full cycle to return data. Components with zero wait-states are allowed, but may decrease achievable frequency because they 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, but can return one data value per cycle for some time thereafter. New pipelined read transfers can be started before `readdata` for the previous transfers is returned. Write transfers cannot be pipelined.

A pipelined read transfer is divided into two phases: 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*, which is the duration from the end of the address phase to the beginning of the data phase.

The key differences between how wait-states and pipeline latency affect transfer timing is as follows:

- *Wait-states*—Wait-states determine the length of the address phase, and limit the maximum throughput of a port. If a slave requires one wait-state to respond to a transfer request, then 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, even though it may require several cycles of latency to return the first unit of data.

Wait-states and pipelined reads can be supported concurrently, and pipeline latency can be either fixed or variable, as discussed in the following sections.

3.5.3.1. Pipelined Read Transfer with Variable Latency

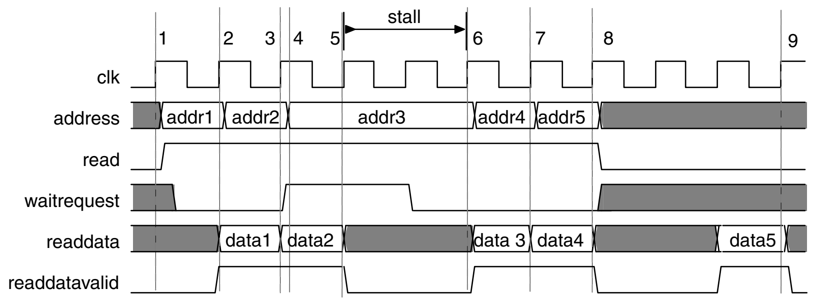
An Avalon-MM pipelined slave takes one or more cycles to produce data after address and control signals have been captured. A pipelined slave port may have multiple pending read transfers at any given time. Variable-latency pipelined read transfers use the same set of signals as non-pipelined read transfers, with one additional signal, `readdatavalid`. 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 the number of pending transfers at an acceptable level.

☞ The maximum number of pending transfers is a property of the slave interface. The interconnect fabric builds logic which routes `readdata` to the requesting masters, parameterized by this maximum number. It is the responsibility of the slave interface, not the interconnect fabric, to keep the number of pending reads from exceeding the stated maximum by asserting `waitrequest`.

Figure 3–5 shows several slave read transfers between a master and a pipelined slave with variable latency. In this example, the slave can accept a maximum of two pending transfers and uses `waitrequest` to prevent overrunning this maximum.

Figure 3–5. Pipelined Read Transfers with Variable Latency



Notes to Figure 3–5:

(1) The master asserts `address` and `read`, initiating a read transfer.
(2) The slave captures `addr1`, and immediately provides the response `data1` and asserts `readdatavalid`.
(3) The slave captures `addr2` and immediately provides the response `data2` and asserts `readdatavalid`. The interconnect captures `data1`.
(4) The slave asserts `waitrequest` for two cycles causing the third transfer to be stalled.
(5) The interconnect captures `data2`.
(6) The slave drives `readdatavalid` and valid `readdata` in response to the third read transfer.
(7) The data from transfer 3 is captured by the interconnect at the same time that the slave captures `addr4`.
(8) The slave captures `addr5`. The interconnect captures `data4`.
(9) `data5` is presented 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

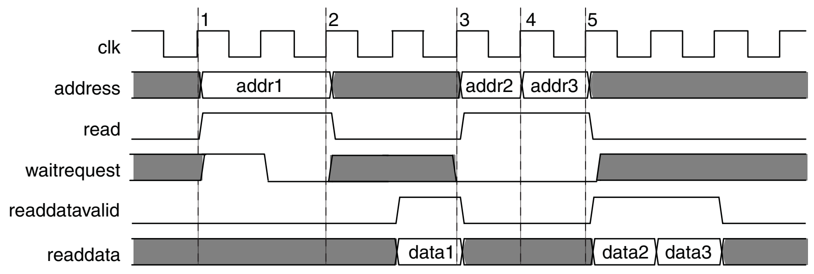
The address phase for fixed latency read transfers is identical to the variable latency case. After the address phase, a pipelined slave port with fixed read latency takes a fixed number of clock cycles to return valid `readdata`, as indicated by the `readWaitTime` property. The interconnect captures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

During the address phase, the slave port can assert `waitrequest` to hold off the transfer or can specify `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. If the slave has 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 shows multiple data transfers between a master and a pipelined slave port that uses `waitrequest` and has a fixed read latency of 2 cycles.

Figure 3–6. Pipelined Read Transfer with Fixed Latency of Two Cycles



Notes to Figure 3–6:

(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. If a Avalon-MM interface includes both read and write functionality and supports bursting, it must support both burst reads and burst writes.

To support bursts, an Avalon-MM interface includes 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.

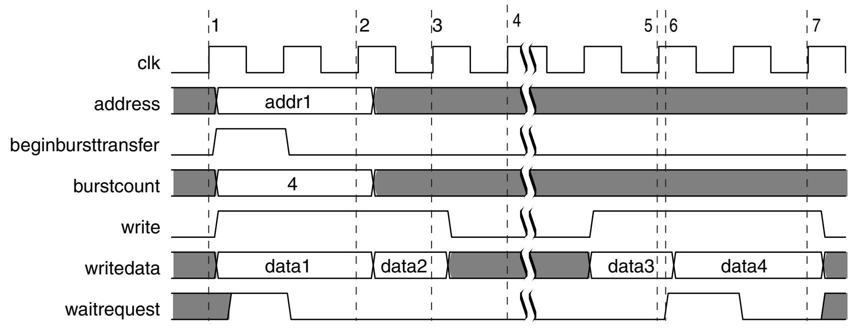
3.5.4.1. Write Bursts

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, guaranteeing that data arrives, in order, from the master port that initiated the burst.
- The slave must only capture `writedata` when `write` is asserted. During the burst, `write` can be deasserted to indicate that it is not presenting valid `writedata`. Deasserting `write` does not terminate the burst; it only 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 true for a master, declares that the master holds `address` and `burstcount` stable throughout a burst; when false, declares that the master holds `address` and `burstcount` stable only for the first transaction of a burst. When true for a slave, declares that the slave expects `address` and `burstcount` to be held stable throughout a burst; when false, declares that the slave samples `address` and `burstcount` only on the first transaction of a burst. (Refer to "Avalon-MM Interface Properties" on page 3–5.)
- The slave can delay a transfer by asserting `waitrequest` which forces `writedata`, `write`, and `byteenable` to be held constant, as usual.
- 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 demonstrates a slave write burst of length 4. In this example, the slave port asserts `waitrequest` twice delaying the burst.

Figure 3–7. Write Burst with constantBurstBehavior Set to False for Master and Slave



Notes to Figure 3–7:

(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 Figure 3–7, 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 asserted only for the first clock cycle.

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, and `readdatavalid` indicates when the slave is presenting valid `readdata`. The difference is that a single read burst address results in multiple data transfers.

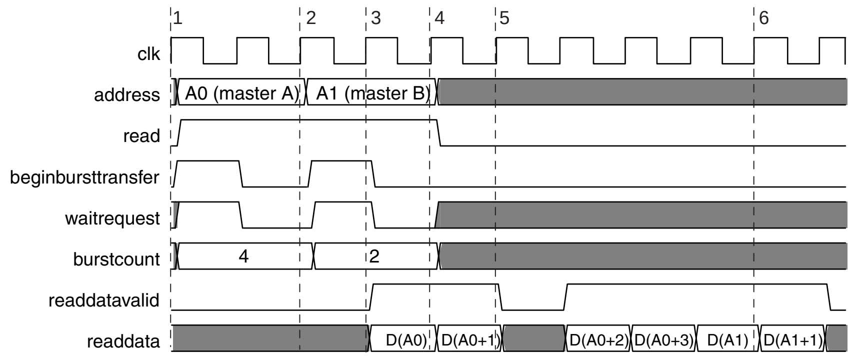
These rules apply to read bursts:

- 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.

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 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.

Figure 3–8. Read Burst



Notes to Figure 3–8:

(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, and the data 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 data that the processor requested is not available until the fourth read. With wrapping bursts, the address order is 0xC, 0x10, 0x14, 0x18, 0x1C, 0x0, 0x4, and 0x8 such that the data that the processor requested 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 the following pattern for the `byteenables`: 1100.)

3.7. Avalon-MM Slave Addressing

Dynamic bus sizing refers to a service provided by the interconnect that dynamically manages data during transfers between master-slave pairs of differing data widths, such that all 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, when a 32-bit master port performs a full 32-bit read transfer from a 16-bit slave port, the interconnect executes two read transfers on the slave side on consecutive addresses, and presents 32-bits of slave data back to the master port.

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. Table 3–3 shows how slave data of various widths is aligned within a 32-bit master when the master is performing full-word accesses. In Table 3–3, `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 to Table 3–3:

(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.

4. Avalon Interrupt Interfaces

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, but has no relationship to any transfer on any other interface. `irq` must be asserted until the interrupt has been acknowledged on the associated Avalon-MM slave interface. The interrupt receiver typically determines how to respond to the event by reading an interrupt status register from an Avalon-MM slave interface. The mechanism used to acknowledge an interrupt is component specific.

4.1.1. Interrupt Sender Signal Roles

Table 4–1 lists the interrupt 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 to be serviced.

4.1.2. Interrupt Sender Properties

Table 4–2 lists the properties associated with interrupt senders.

Table 4–2. Interrupt Sender Properties

Property Name	Default Value	Legal Values	Description
`associatedAddressablePoint`	—	Name of Avalon-MM slave on this component.	The name of the Avalon-MM slave interface that provides access to the registers that should be accessed to service the interrupt.
`associatedClock`	—	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.
`associated Reset`	—	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 an Avalon-MM master interface 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 lists the 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 lists the properties associated with interrupt receivers.

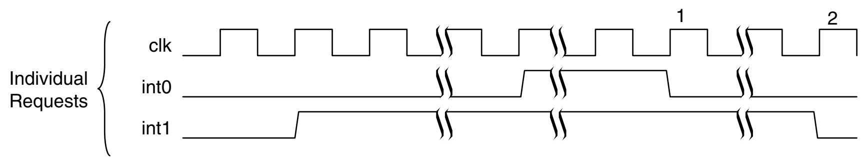
Table 4–4. Interrupt Receiver Properties

Property Name	Default Value	Legal Values	Description
`associatedAddressablePoint`	—	Name of Avalon-MM master interface	The name of the Avalon-MM master interface used to service interrupts received on this interface.
`associatedClock`	—	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`	—	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

Figure 4–1 illustrates 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



Notes to Figure 4–1:

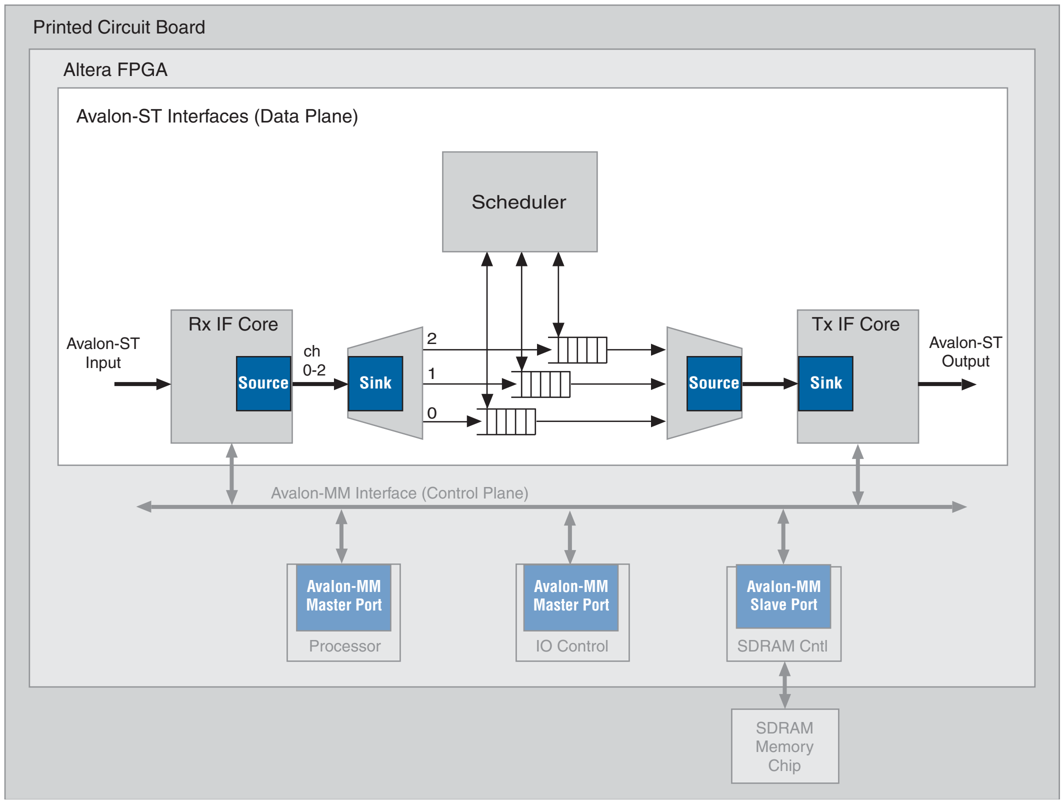
(1) Interrupt 0 serviced.

(2) Interrupt 1 serviced.

5. Avalon Streaming Interfaces

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 illustrates a typical application of the Avalon-ST interface.

Figure 5–1. Avalon-ST Interface - Typical Application



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.

5.1. Features

The following list highlights some of the prominent features of the Avalon-ST interface:

- 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

5.2. Terms and Concepts

This section defines terms and concepts used in the Avalon-ST interface protocol.

- *Avalon Streaming System*—An Avalon Streaming system is a system that contains one or more Avalon-ST connections that transfer data from a source interface to a sink interface. The system shown in Figure 5–1 consists of Avalon-ST interfaces to transfer data from the system input to output and Avalon-MM control and status register interfaces to allow 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 is a mechanism by which a sink can signal to a source to stop sending data. The sink typically uses backpressure to stop the flow of data when its FIFOs are full or when there is congestion on its output port. Support for backpressure is optional.
- *Transfers and Ready Cycles*—A transfer is an operation that 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 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.

5.3. 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 lists the signal roles that comprise an Avalon-ST data interface.

Table 5–1. Avalon-ST Interface Signals

Signal Role	Width	Direction	Description
Fundamental Signals			
`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, during which the source may assert `valid` and transfer data. Sources without a `ready` input cannot be backpressured, and 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. On ready cycles where `valid` is asserted, the data bus and other source to sink signals are sampled by the sink, and on other cycles are ignored. Sources without a `valid` output implicitly provide valid data on every cycle that they are not being backpressured, and 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.

All signal roles listed in Table 5–1 are active high.

5.4. Signal Sequencing and Timing

This section provides information related to timing and sequencing of Avalon-ST interfaces.

5.4.1. 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 provides for high frequency operation while eliminating back-to-back registers with no intervening logic.

5.4.2. Clock Enables

Avalon-ST components typically do not include a clock enable input, because 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 take care to ensure that the timing of the interface control signals still adheres to the protocol.

5.5. Avalon-ST Interface Properties

Table 5–2 lists the properties that characterize an Avalon-ST interface.

Table 5–2. Avalon-ST Interface Properties (Part 1 of 2)

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.
`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.
`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, and 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, and 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, that is, D0 appears at `data[7:0]`.

Table 5–2. Avalon-ST Interface Properties (Part 2 of 2)

Property Name	Default Value	Legal Values	Description
`maxChannel`	0	0 – 255	The maximum number of channels that a data interface can support.
`readyLatency`	0	0 – 8	Defines the relationship between assertion and deassertion of the ready signal, and cycles which are considered to be `ready` for data transfer, separately for each interface.

5.6. Typical Data Transfers

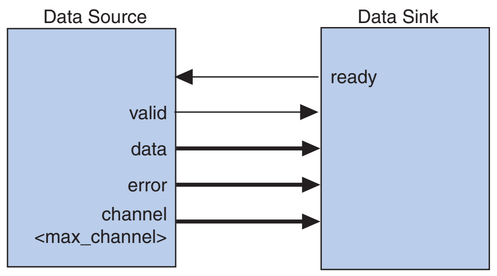
This section defines the transfer of data from a source interface to a sink interface. In all cases, the data source and the data sink must comply with the specification. It is not the responsibility of the data sink to detect source protocol errors.

5.7. Signal Details

This section describes the basic Avalon-ST protocol that all data transfers must follow. It also highlights the flexibility available in choosing Avalon-ST signals to meet the needs of a particular component and makes recommendations about the signals that should be used.

Figure 5–1 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



The following paragraphs provide more details about these signals:

- `ready`—On interfaces supporting backpressure, the sink asserts `ready` to mark the cycles where transfers may take place. Data interfaces that support backpressure must define the `readyLatency` parameter so that 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 active cycle the `data` signal and other source to sink signals are sampled by the sink.

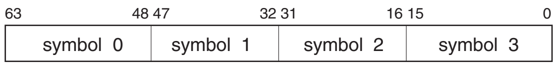
- `data`—The `data` signal typically carries the bulk of the information being transferred from the source to the sink, and consists of one or more symbols being transferred on every clock cycle. The `dataBitsPerSymbol` parameter defines how the data signal is divided into symbols.
- `error`—Errors are 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, while 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 that the interface supports varies while the component is operating, `maxChannel` is the maximum channel number that the interface can support.

5.8. Data Layout

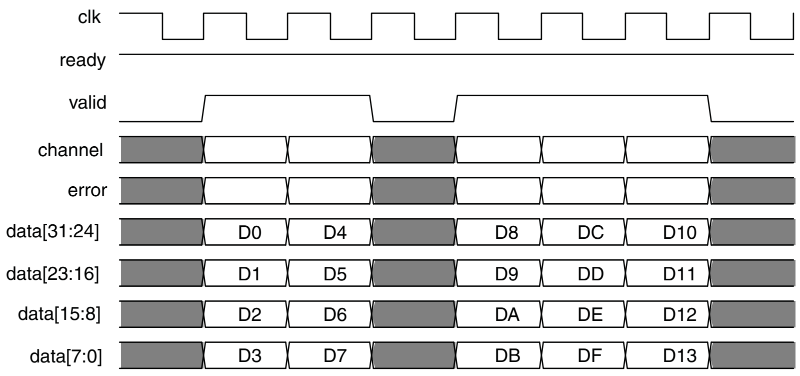
Figure 5–3 shows a 64-bit data signal with `dataBitsPerSymbol`=16. Symbol 0 is the most significant symbol.

Figure 5–3. Data Symbols



The timing diagram in Figure 5–4, provides a 32-bit example where `dataBitsPerSymbol`=8.

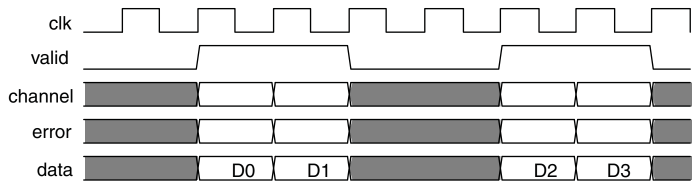
Figure 5–4. Layout of Data



5.9. 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 shows an example of data transfer without backpressure.

Figure 5–5. Data Transfer without Backpressure



5.10. Data Transfer with Backpressure

The sink indicates to the source that it is ready for an active cycle by asserting `ready` for a single clock 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 `readyLatency` has a nonzero value, the interface considers cycle <*`n + readyLatency`*> to be 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, which is called the ready cycle. In this mode of operation, 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 and 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 illustrates an Avalon-ST interface where `readyLatency = 4`.

Figure 5–6. Avalon-ST Interface with readyLatency = 4

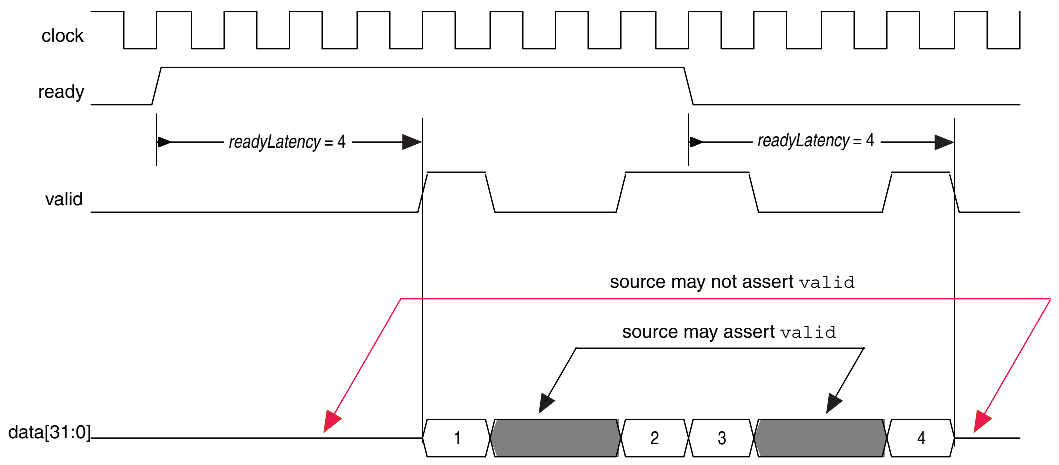


Figure 5–7 illustrates a transfer with backpressure and `readyLatency`=0. The source provides data and asserts `valid` on cycle 1, even though the sink is not ready. The source waits until cycle two, when the sink does assert `ready`, before moving onto the next data cycle. In cycle 3, the source drives data on the same cycle and the sink is ready to receive it; the transfer happens immediately. In cycle 4, the sink asserts `ready`, but the source does not drive valid data.

Figure 5–7. Transfer with Backpressure, readyLatency=0

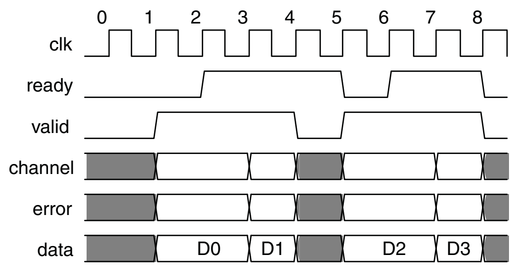


Figure 5–8 and Figure 5–9 show data transfers with `readyLatency`=1 and `readyLatency`=2, respectively. In both these cases, `ready` is asserted before the ready cycle, and the source responds 1 or 2 cycles later by providing data and asserting `valid`. When `readyLatency` is not 0, the source must deassert `valid` on non-ready cycles.

Figure 5–8. Transfer with Backpressure, readyLatency=1

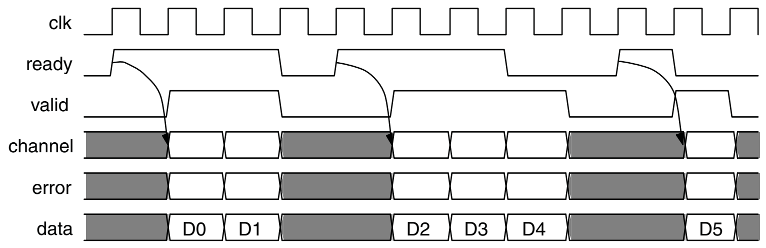
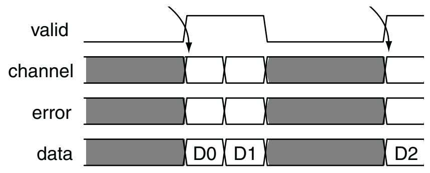


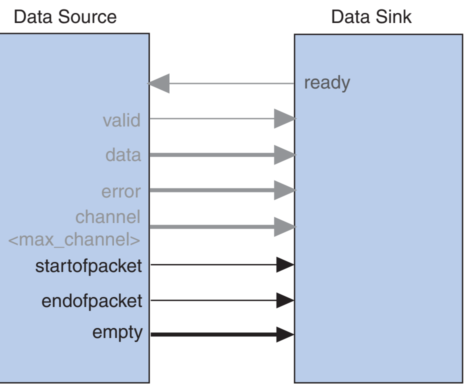
Figure 5–9. Transfer with Backpressure, readyLatency=2



5.11. 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 is provided to create connections between source and sink interfaces with and without packet support.

Figure 5–10. Avalon-ST Packet Interface Signals



5.12. Signal Details

The following paragraphs provide more details about these three signals:

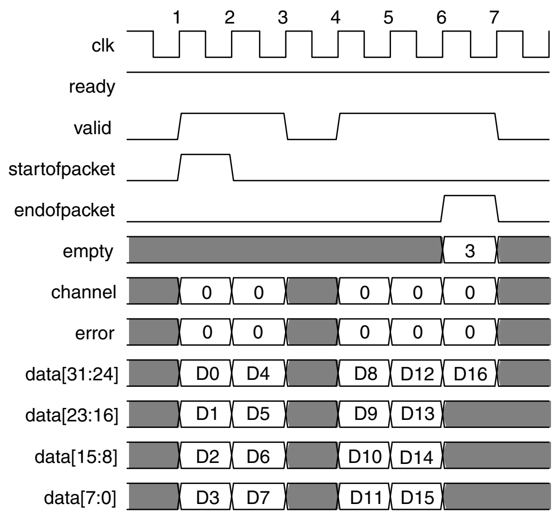
- `startofpacket`—The `startofpacket` signal is required by all interfaces supporting packet transfers and marks the active cycle containing the start of the packet. This signal is only interpreted when `valid` is asserted.
- `endofpacket`—The `endofpacket` signal is required by all interfaces supporting packet transfer and 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, so that the `startofpacket` signal can follow immediately after the previous `endofpacket` signal.
- `empty`—The optional `empty` signal indicates the number of symbols that are empty during the cycles that mark the end of a packet. 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*>`)`.

5.13. Protocol Details

Packet data transfer follows the same protocol as the typical data transfer described in "Typical Data Transfers" on page 5–5, with the addition of the `startofpacket`, `endofpacket`, and `empty`.

Figure 5–11 illustrates the transfer of a 17-byte packet from a source interface to a sink interface, where `readyLatency=0`. Data transfer occurs on cycles 1, 2, 4, 5, and 6, when both `ready` and `valid` are asserted. During cycle 1, `startofpacket` is asserted, and the first 4 bytes of packet are transferred. During cycle 6, `endofpacket` is asserted, and `empty` has a value of 3, indicating 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.

Figure 5–11. Packet Transfer



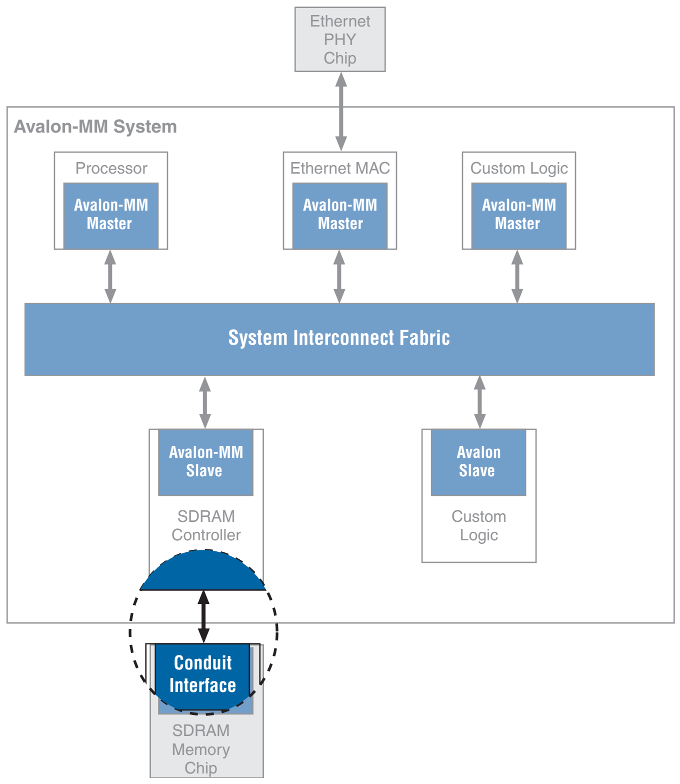
6. Avalon Conduit Interfaces

Avalon Conduit interfaces are used to group together an arbitrary collection of signals. You can specify any role for these 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.

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. For example, if the signals in your conduit change clock domains between the endpoints, Qsys cannot check or adapt to that.

As shown in Figure 6–1, 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. Signals

Table 6–1 lists the conduit signal roles.

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. Properties

There are no properties for conduit interfaces.

7. Avalon Tristate Conduit Interface

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 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. Figure 7–1 illustrates the naming conventions for Avalon-TC shared pins.

Figure 7–1. Shared Pin Types

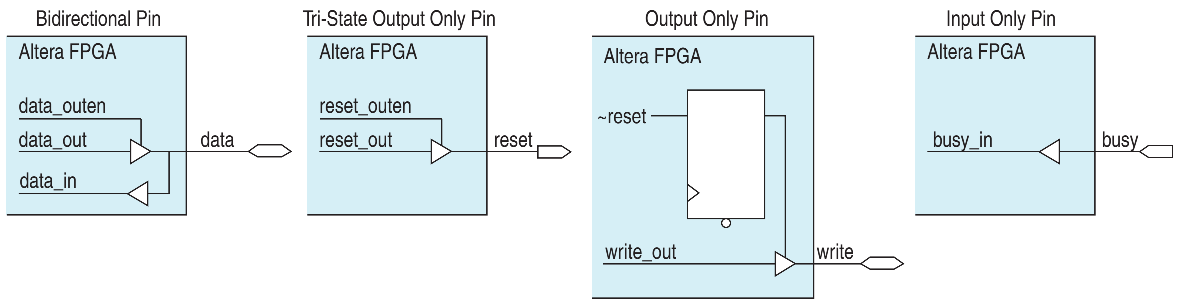
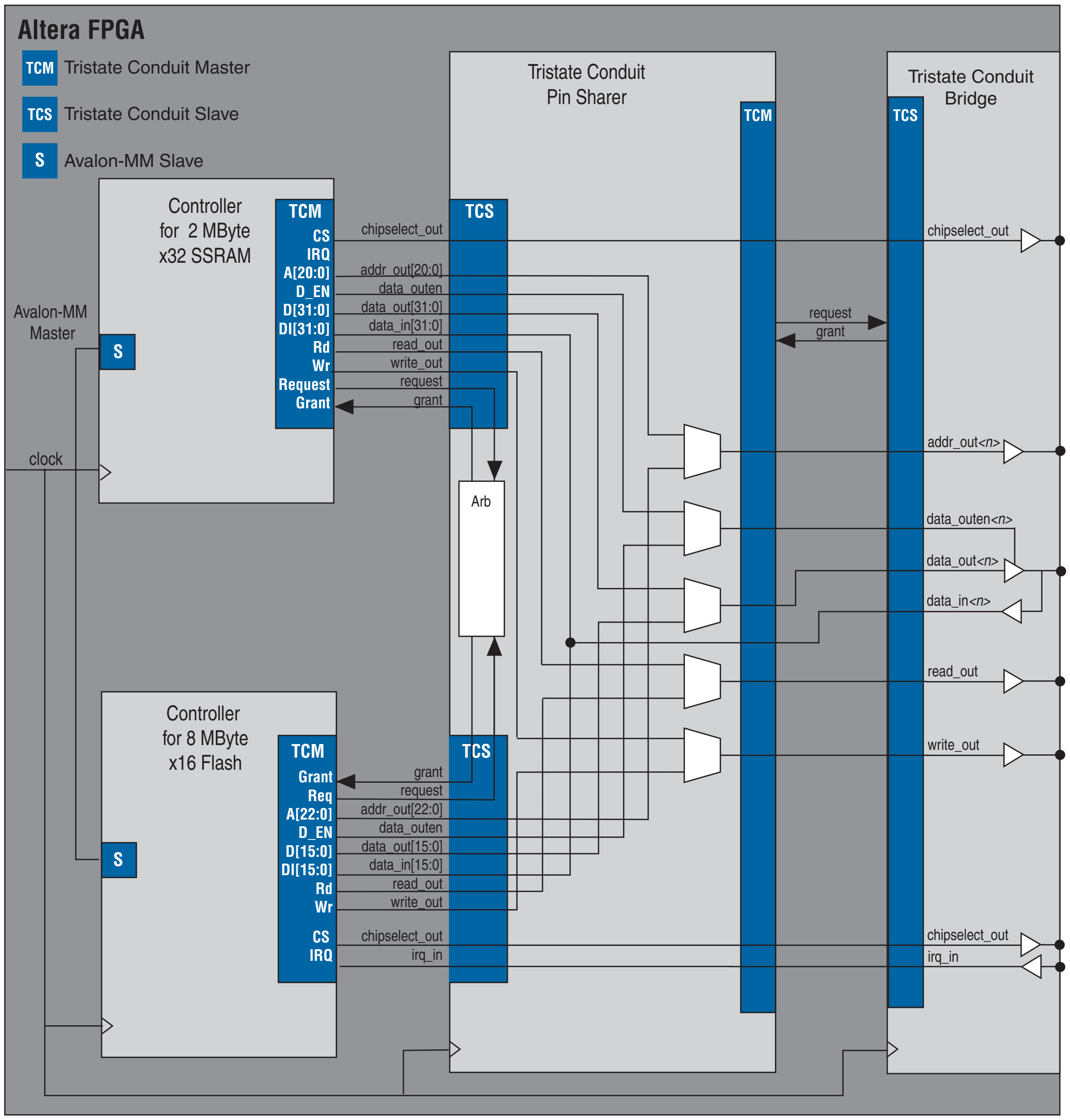


Figure 7–2 illustrates pin sharing using Avalon-TC interfaces. This figure illustrates the following points.

- The Tristate Conduit Pins Sharer includes separate Tristate Conduit Slave Interface 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 pin on the FPGA. 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 0th bit and 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 illustrates the typical use of Avalon-TC Master and Slave interfaces and signal naming.

Figure 7–2. Tristate Conduit Interfaces



For more information about the Generic Tristate Controller and Tristate Conduit Pin Sharer, refer to the *Avalon Tristate Conduit Components User Guide* and the *Qsys Interconnect* chapter in volume 1 of the *Quartus II Handbook*.

7.1. Tristate Conduit Signals

Table 7–1 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. 1. When `request` is asserted and `grant` is deasserted, `request` is requesting access for the current cycle. 2. 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. Because `request` is deasserted in the last cycle of a bus access, it can be reasserted immediately following the final cycle of a transfer, making 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 Figure 7–3 on page 7–4 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 and 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.

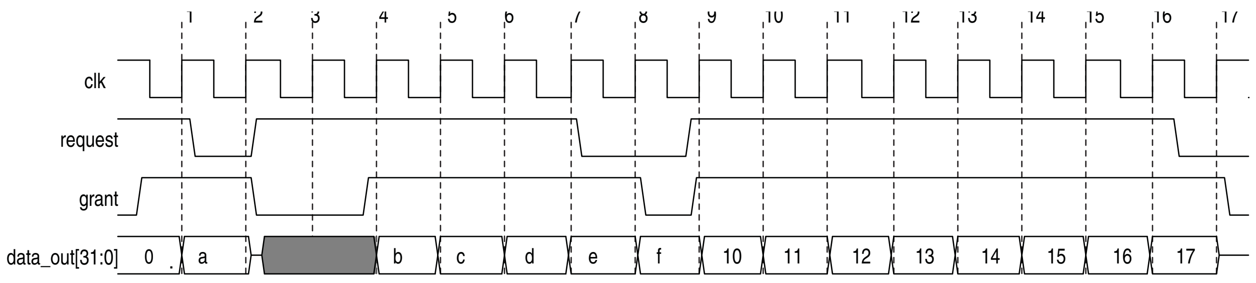
7.3. Tristate Conduit Timing

Figure 7–3 illustrates arbitration timing for the Tristate Conduit Pin Sharer. As this figure illustrates, a device can drive or receive valid data in the granted cycle. Figure 7–3 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.

Figure 7–3. Arbitration Timing



Additional Information

This chapter provides additional information about the document and Altera.

Document Revision History

The following table shows the revision history for this document.

Date	Version	Changes
May 2013	13.0	Minor updates to Chapter 3, Avalon Memory-Mapped Interfaces: Minor updates to Chapter 5, Avalon Streaming Interfaces: Updated Chapter 6, Avalon Conduit Interfaces to describe the signal roles supported by Avalon conduit interfaces. Updated Figure 7–1 on page 7–1.
May 2011	11.0	Initial release of the *Avalon Interface Specifications* supported by Qsys.

How to Contact Altera

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

Contact *(1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note to Table:

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

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`).
↵	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
☞	The hand points to information that requires special attention.
?	A question mark directs you to a software help system with related information.
👣	The feet direct you to another document or website with related information.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
⚠ WARNING	A warning calls attention to a condition or possible situation that can cause you injury.
✉	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.