Avalon Memory-Mapped

Interface Specification

MNL-AVABUSREF-3.3

I.S. EN ISO 9001

Table of Contents

This specification provides comprehensive specification for the Avalon® Memory-Mapped Interface Specification, version 3.3.

How to Contact Altera

For 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
Altera literature services	Email	literature@altera.com
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

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: f_{MAX}, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."

Visual Cue	Meaning
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix n, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the VHDL keyword `BEGIN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
⚠ CAUTION	The caution indicates required information that needs special consideration and understanding and should be read prior to starting or continuing with the procedure or process.
⚡ WARNING	The warning indicates information that should be read prior to starting or continuing the procedure or processes
↵	The angled arrow indicates you should press the Enter key.
👣	The feet direct you to more information on a particular topic.

1. Introduction

The Avalon Memory-Mapped (Avalon-MM) interface specification is designed to accommodate peripheral development for the system-on-a-programmable-chip (SOPC) environment. The specification provides peripheral designers with a basis for describing the address-based read/write interface found on master and slave peripherals, such as microprocessors, memory, UART, timer, etc.

The specification defines transfers between a peripheral and an interconnect structure. The specification's interconnect strategy allows system designers to connect any master-type peripheral to any slave-type peripheral, without *a priori* knowledge of either the master or slave interface. The Avalon-MM interface specification describes a configurable interconnect strategy that allows a peripheral designer to limit the signal types needed to support the specific type(s) of transfers desired.

The Avalon-MM interface defines:

- A set of signal types
- The behavior of these signals
- The types of transfers supported by these signals

For example, the Avalon-MM interface can be used to describe a traditional peripheral interface, such as SRAM, that supports only simple, fixed-cycle read/write transfers. On the other hand, the Avalon-MM interface can also be used to describe a more complex pipelined interface capable of burst transfers.

1.1. Features

Some of the prominent features of the Avalon-MM interface are:

- *Separate Address, Data and Control Lines* – Provides the simplest interface to on-chip logic. By using dedicated address and data paths, Avalon-MM peripherals do not need to decode data and address cycles.
- *Up to 1024-bit Data Width* – Supports data paths up to 1024 bits. The Avalon-MM interface supports arbitrary data widths, including widths that are not an even power of two.

- *Synchronous Operation* – Provides an interface optimized for synchronous, on-chip peripherals. Synchronous operation simplifies the timing behavior of the Avalon-MM interface, and facilitates integration with high-speed peripherals.
- *Dynamic Bus Sizing* – Handles the details of transferring data between peripherals with different data widths. Avalon-MM peripherals with differing data widths can interface easily with no special design considerations.
- *Simplicity* – Provides an easy-to-understand interface protocol with a short learning curve.
- *Low resource utilization* – Provides an interface architecture that conserves on-chip logic resources.
- *High performance* – Provides performance up to one-transfer-per-clock.

The Avalon-MM interface is an open standard. No license is required to produce and distribute custom peripherals using the Avalon-MM interface.

1.2. Terms and Concepts

This section defines terms and concepts upon which the Avalon-MM interface specification is based.

1.2.1. Avalon-MM Peripherals and System Interconnect Fabric

A typical system based on the Avalon-MM interface combines multiple functional modules, called *Avalon-MM peripherals*. *System interconnect fabric* is on-chip interconnect logic that connects the Avalon-MM peripherals together, forming a larger system. Figure 1 shows an example Avalon system with multiple Avalon-MM peripherals connected via system interconnect fabric.

Figure 1. Example Avalon-MM System
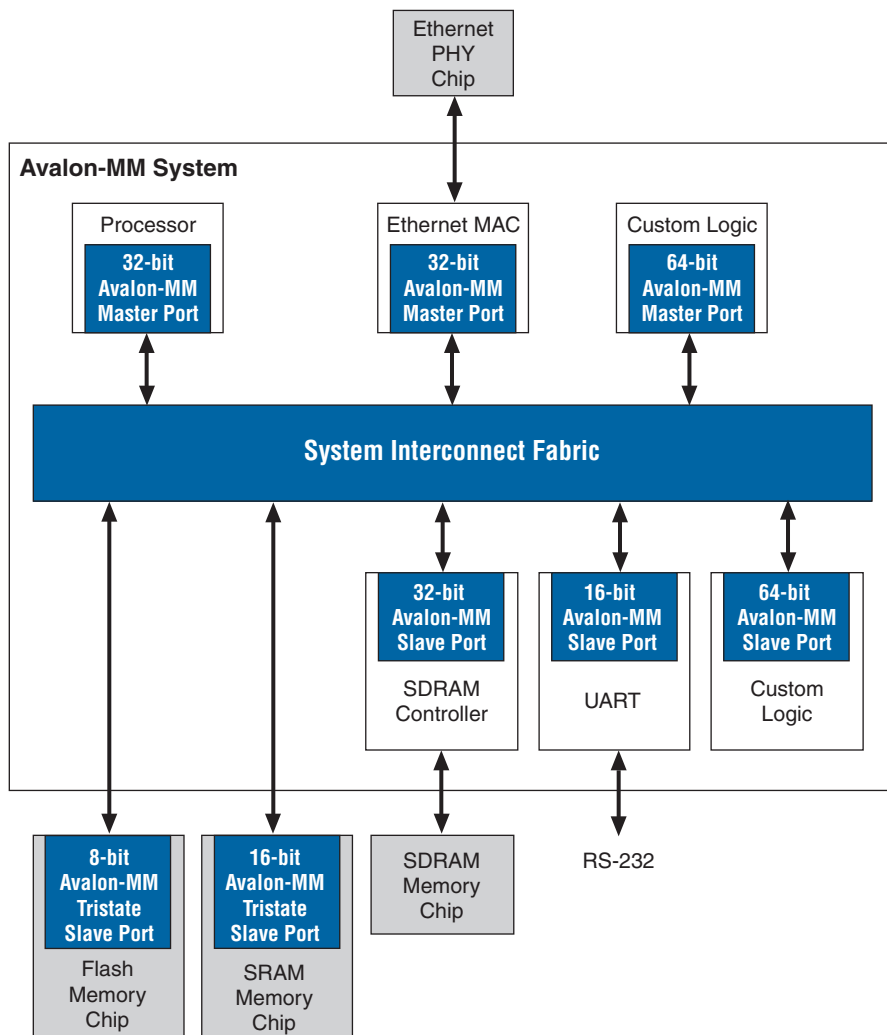
The Avalon-MM interface defines the point of connection between Avalon-MM peripherals and the system interconnect fabric. This document focuses on the interfaces between the peripheral and the fabric. The Avalon-MM interface specification also defines the behavior of system interconnect fabric at the interface level, but does not specify the internal implementation.

1.2.2. Avalon-MM Signal Types: A Configurable Interface

The Avalon-MM interface defines a set of signal types (chip select, read enable, write enable, address, data, etc.) that describe the address-based read/write interfaces found on typical master and slave modules. An Avalon-MM peripheral uses exactly the signals required to interface to the peripheral's core logic, and excludes signals that would add unnecessary overhead. "Avalon-MM Slave Port Signals" on page 15 for the complete list of Avalon-MM signal types.

This configurability is one of the key differences between the Avalon-MM interface and traditional bus interfaces. Avalon-MM peripherals can use a small set of signals to support simple transfer types, or use more signals to support complex transfer types. For example, a ROM interface may require only address, read-data and select signals, while a high-speed memory controller may require additional signals to support pipelined bursts.

The Avalon-MM signal types provide a superset of several other bus interfaces. For example, the pins on most discrete SRAM, ROM and flash chips can be mapped to Avalon-MM signal types, allowing Avalon-MM systems to interface directly to these chips. Similarly, most Wishbone interface signals can be mapped to Avalon-MM signal types, making it easy to include Wishbone cores into Avalon-MM systems.

1.2.3. Master Ports and Slave Ports

An *Avalon-MM port* is a group of Avalon-MM signals used collectively as a single interface. The role of an Avalon-MM port is categorized as either slave or master. A *master port* is the collection of Avalon-MM signal types used to initiate transfers. A *slave port* is the collection of Avalon-MM signal types used to respond to transfer requests.

Avalon-MM master and slave ports do not connect together directly. Instead, Avalon-MM ports connect to system interconnect fabric and the system interconnect fabric translates signals between master ports and slave ports, as shown in Figure 1 on page 11. During a transfer, the signals exchanged between a master port and the system interconnect fabric might be very different than the signals that the system interconnect fabric uses to communicate with the target slave port. For this reason, when discussing Avalon-MM transfers it is important to distinguish which port is the focus, master or slave.

1.2.4. Avalon-MM Peripherals

An Avalon-MM peripheral is a logical device—either on-chip or off-chip—that performs some system-level task, and communicates with other peripherals through one or more Avalon-MM port(s). A peripheral

can have any combination of Avalon-MM ports: One slave port, one master port, multiple slave ports, multiple master ports, or a combination of master and slave ports.

1.2.5. Transfer

A transfer is a read or write operation of a unit of data, transmitted between an Avalon-MM port and the system interconnect fabric. Avalon-MM transfers transmit up to 1024 bits at a time, and take one or more clock cycles to complete. After a transfer completes, the Avalon-MM port is available for another transaction on the next clock.

Avalon-MM transfers are separated into two fundamental categories: master and slave. Avalon-MM maser ports initiate master transfers to the system interconnect fabric. Avalon-MM slave ports respond to slave transfer requests from the system interconnect fabric. The perspective of a transfer is always with respect to the Avalon-MM port: Master ports only perform master transfers, and slave ports only perform slave transfers.

1.2.6. Master-Slave Pair

A master-slave pair refers to a master port and a slave port connected via the system interconnect fabric during a data transfer. During a transfer, the master port's control and data signals pass through the system interconnect fabric and interact with the slave port.

1.2.7. Cycle

A cycle is a basic unit of one clock period, which is defined from rising-edge to rising-edge of the clock associated with the particular port. The shortest duration of an Avalon-MM transfer is one cycle.

2. Avalon-MM Signals

This section defines the signals used by the Avalon-MM interface. The Avalon-MM interface specification defines the possible types of signals that an Avalon-MM peripheral can use, such as `address`, `data`, `chipselect`, etc. An Avalon-MM peripheral design can include any signal type, depending on the requirements for the interface to the peripheral logic.

The Avalon-MM interface specification defines the behavior of the Avalon-MM signal types. Each signal in an Avalon-MM master or slave port corresponds to exactly one Avalon-MM signal type. An Avalon-MM port can use only one instance of each signal type.

Avalon-MM signal types are classified as either slave signals or master signals, depending on whether the Avalon-MM port is a master or slave. Certain signal types exist in both master and slave port interfaces, but their behavior is different, depending on the port type.

For example, consider the 16-bit output-only general-purpose I/O peripheral shown in Figure 2. This simple Avalon-MM peripheral needs only to respond to transfer requests to receive data. Therefore, it uses only Avalon-MM slave signals for write transfers, and no signals for read transfers.

Figure 2. : Example Slave Peripheral
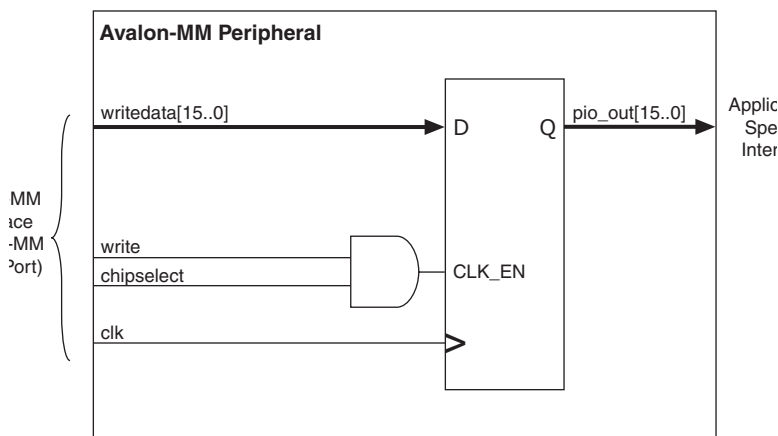
An Avalon-MM peripheral can also include custom, application-specific signals that are not associated with an Avalon-MM interface, such as the `pio_out` signal shown in Figure 2. Application-specific signals connect to logic outside the Avalon-MM system and do not directly interface to system interconnect fabric.

2.1. Complete List of Signal Types

Table 1 on page 15 lists the signal types that comprise the Avalon-MM interface for slave ports. Table 2 on page 19 lists the signal types that comprise the Avalon-MM interface for master ports. For each available signal type the tables provide:

■   The signal type name

■ The possible widths of the signal
■ The direction of the signal from the perspective of the peripheral
■ Whether or not the signal type is required on an Avalon-MM port
■ A brief description of the purpose and function of the signal type, and any special usage requirements

Table 1 and Table 2 categorize each signal by the transfer property that uses the signal. For details, refer to Transfer Properties on page 26.

Table 1. Avalon-MM Slave Port Signals				
Signal Type	**Width**	**Direction**	**Required**	**Description**
Fundamental Signals				
clk	1	In	No	Synchronization clock for the Avalon-MM slave interface. All signals are synchronous to clk. Asynchronous slave ports can omit clk.
chipselect	1	In	No	Chip-select signal to the slave port. The slave port ignores all other Avalon-MM signal inputs unless chipselect is asserted.
address	1-32	In	No	Address lines from the system interconnect fabric to the slave port. Specifies a word offset into the slave address space.
read	1	In	No	Read-request signal to the slave port. Not required if the slave port never outputs data. If used, readdata or data must also be used.
readdata	1-1024 (1)(2)	Out	No	Data lines to the system interconnect fabric for read transfers. Not required if the slave port never outputs data. If used, data cannot be used.

Table 1. Avalon-MM Slave Port Signals				
Signal Type	**Width**	**Direction**	**Required**	**Description**
Fundamental Signals				
write	1	In	No	Write-request signal to the slave port. Not required if the slave port never receives data from a master. If used, `writedata` or `data` must also be used, and `writebyteenable` cannot be used.
writedata	1-1024 (1)(2)	In	No	Data lines from the system interconnect fabric for write transfers. Not required if the slave port never receives data. If used, `write` or `writebyteenable` must also be used, and `data` cannot be used.
byteenable	2,4,8, 16, 32, 64, 128	In	No	Byte-enable signals to enable specific byte lane(s) during transfers on ports of width greater than 8 bits. If used, `writedata` must also be used, and `writebyteenable` cannot be used.
writebyteenable	2,4,8,16, 32, 64, 128	In	No	Equivalent to the logical AND of the `byteenable` and `write` signals. If used, `writedata` must also be used. `write` and `byteenable` cannot be used.
begintransfer	1	In	No	Asserted during the first cycle of every transfer. Usage is peripheral-specific.
Wait-State Signals				

Table 1. Avalon-MM Slave Port Signals				
Signal Type	**Width**	**Direction**	**Required**	**Description**
Fundamental Signals				
waitrequest	1	Out	No	Used to stall the system interconnect fabric when the slave port is not able to respond immediately.
Pipeline Signals				
readdatavalid	1	Out	No	Used for pipelined read transfers with variable latency. Marks the rising clock edge when the slave asserts valid readdata.
Burst Signals				
burstcount	2-32	In	No	Used for burst transfers. Indicates the number of transfers in a burst. When used, waitrequest must also be used.
beginbursttransfer	1	In	No	Asserted for the first cycle of a burst to indicate when a burst transfer is starting. Usage is peripheral-specific.
Flow Control Signals				
readyfordata	1	Out	No	Used for transfers with flow control. Indicates that the peripheral is ready for a write transfer.
dataavailable	1	Out	No	Used for transfers with flow control. Indicates that the peripheral is ready for a read transfer.
endofpacket	1	Out	No	Used for transfers with flow control. Indicates an end-of-packet condition to the system interconnect fabric. Implementation is peripheral specific.

Table 1. Avalon-MM Slave Port Signals				
Signal Type	**Width**	**Direction**	**Required**	**Description**
Fundamental Signals				
Tristate Signals				
data	1-1024 (1)	Bi-directional	No	Bidirectional read and write data for tristate slave ports. If used, `readdata` and `writedata` cannot be used.
outputenable	1	In	No	Output-enable signal for the `data` lines. When deasserted, tristate slave port must not drive its `data` lines. If used, `data` must also be used.
Other Signals				
irq	1	Out	No	Interrupt request. A slave port asserts `irq` when it needs to be serviced by a master.
reset	1	In	No	Peripheral reset signal. When asserted, slave peripheral must enter a deterministic reset state.
resetrequest	1	Out	No	Allows the peripheral to reset the entire Avalon-MM system. The result is immediate.
Notes to Table 1: (A) If the slave port uses dynamic bus sizing, this signal's width must be a power of two. (B) If a slave port uses both `readdata` and `writedata`, the width of both signals must be equal.				

The Avalon-MM interface specification does not mandate the presence of any particular signal in an Avalon-MM slave port.

Table 2. Avalon-MM Master Port Signals				
Signal Type	**Width**	**Direction**	**Required**	**Description**
Fundamental Signals				
clk	1	In	Yes	Synchronization clock for the Avalon-MM slave interface. All signals are synchronous to `clk`.
waitrequest	1	In	Yes	Forces the master port to wait until the system interconnect fabric is ready to proceed with the transfer.
address	1-32	Out	Yes	Address lines from the master port to the system interconnect fabric. The `address` signal represents a byte address. However, the master port must assert `address` on word boundaries only.
read	1	Out	No	Read request signal from master port. Not required if master port never performs read transfers. If used, `readdata` or `data` must also be used.
readdata	8,16,32,64, 128, 256, 512, 1024 (1)	In	No	Data lines from the system interconnect fabric for read transfers. Not required if the master port never performs read transfers. If used, `read` must also be used, and `data` cannot be used.
write	1	Out	No	Write request signal from master port. Not required if the master port never performs write transfers. If used, `writedata` or `data` must also be used.

Table 2. Avalon-MM Master Port Signals				
Signal Type	Width	Direction	Required	Description
Fundamental Signals				
writedata	8,16,32,64, 128, 256, 512, 1024 (1)	Out	No	Data lines to the system interconnect fabric for write transfers. Not required if the master port never performs write transfers. If used, `write must` also be used, and data cannot be used.
byteenable	2,4,8, 16, 32, 64, 128	Out	No	Byte-enable signals to enable specific byte lane(s) during transfers on ports of width greater than 8 bits. The master port must assert all `byteenable` lines during read transfers.
Pipeline Signals				
readdatavalid	1	In	No	Used for pipelined read transfers with latency. Indicates that valid data from the system interconnect fabric is present on the `readdata` lines. Required if the master is pipelined.
flush	1	Out	No	Used for pipelined read transfers. The master port asserts `flush` to clear any pending transfers in the pipeline.
Burst Signals				
burstcount	2-32	Out	No	Used for burst transfers. Indicates the number of transfers in a burst.
Flow Control Signals				

Table 2. Avalon-MM Master Port Signals				
Signal Type	**Width**	**Direction**	**Required**	**Description**
Fundamental Signals				
endofpacket	1	In	No	Used for transfers with flow control. Indicates an end-of-packet condition from the system interconnect fabric. Implementation is peripheral specific.
Tristate Signals				
data	8,16,32,64, 128, 256, 512, 1024			Bidirectional read and write data for tristate master ports. If used, readdata and writedata cannot be used.
Other Signals				
irq	1, 32	In	No	Indicates when one or more slave ports have requested an interrupt. If irq is a 32-bit vector, each line corresponds directly to the irq signal on a slave port, with no inherent assumption of priority. If irq is one bit wide, it is the logical OR of all slave irq signals, and the interrupt priority is encoded on irqnumber.
irqnumber	6	In	No	Indicates the interrupt priority of a slave port asserting its interrupt request. Lower value means higher priority. Used only when the irq signal is one bit wide.
reset	1	In	No	Global reset signal. Implementation is peripheral specific.

Table 2. Avalon-MM Master Port Signals				
Signal Type	**Width**	**Direction**	**Required**	**Description**
Fundamental Signals				
resetrequest	1	Out	No	Allows the peripheral to reset the entire Avalon-MM system. The result is immediate.

Note to Table 2:
(A)   If a master port uses both `readdata` and `writedata`, the width of both signals must be equal.

The Avalon-MM interface specification only mandates the existence of three signals on an Avalon-MM master port: `clk, address, and waitrequest`.

2.2. Signal Polarity

The signal types listed in Table 1 and Table 2 are active high. The Avalon-MM interface also offers the negated version of each signal type, indicated by appending `_n` to the signal type name (e.g., `irq_n`, `read_n`). This is useful for interfacing to off-chip peripherals that use active-low logic.

2.3. Signal Naming Conventions

The Avalon-MM interface specification does not dictate a naming convention for the signals that appear on Avalon-MM peripherals. A signal in an Avalon-MM port can be named the same as its signal type, or it can be named differently to comply with a system-wide naming convention. For example, an Avalon-MM peripheral may have an Avalon-MM slave port with an input signal named `clock_100mhz` of type `clk`.

In the discussion of Avalon-MM transfers in this document, the signal names are the same as the signal type, but this naming convention is not part of the Avalon-MM interface specification.

2.4. Signal Sequencing and Timing

This section describes issues related to timing and sequencing of Avalon-MM signals.

2.4.1. Synchronous Interface

The Avalon-MM interface is a synchronous protocol. Each Avalon-MM port is synchronized to a clock provided by the system interconnect fabric. All transfers occur synchronous to the system interconnect fabric clock. All transfers start on a rising clock edge.

A synchronous interface does not necessarily mean that all Avalon-MM signals are registered. Signals may be combinatorial, based on the outputs of registers that are synchronous to the system interconnect fabric clock. Therefore, an Avalon-MM peripheral must not be edge-sensitive to any Avalon-MM signal besides clk. As with any synchronous design, Avalon-MM peripherals must act only in response to signals that are stable at the rising edge of clk, and must produce stable output signals at the rising edge of clk. The Avalon-MM interface specification does not dictate how or when signals transition between clock edges. For this reason, the system interconnect fabric timing diagrams in this document are devoid of explicit timing information.

2.4.2. Interfacing to Asynchronous Peripherals

It is possible to interface asynchronous peripherals, such as off-chip memory devices, to the system interconnect fabric, but there are a few design considerations. Due to the synchronous operation of the system interconnect fabric, Avalon-MM signals toggle only at intervals equal to the period of the Avalon-MM interface clock. Furthermore, if asynchronous signals are connected directly to system interconnect fabric inputs, the designer must make sure that the signals are stable at the rising edge of clk.

2.4.3. Performance

There is no fixed or maximum performance of the Avalon-MM interface. The interface is synchronous and can be driven at any frequency provided by the system interconnect fabric. The maximum performance is dependent on peripheral design and system implementation.

2.4.4. Electrical Characteristics

The Avalon-MM interface specification does not specify any electrical or physical characteristics traditionally required by shared bus implementations.

2.5. Transfer Properties

Different Avalon-MM ports have different transfer capabilities, because not all Avalon-MM master or slave ports use the same signal types. The Avalon-MM interface specification defines a set of properties that transfers can exhibit. A specific Avalon-MM master or slave port may support one or more of these properties, depending on the peripheral design. The transfer properties supported by an Avalon-MM peripheral are determined at design time, and do not change from transfer-to-transfer.

The Avalon-MM interface specification defines the following transfer properties that Avalon-MM ports can support:

■ Wait-states: Fixed or variable (slave only)
■ Pipeline: Fixed or variable latency
■ Setup and hold time (slave only)
■ Burst
■ Tristate

Each transfer property is discussed in detail in Slave Transfers on page 27 and Master Transfers on page 42.

The basis for all Avalon-MM transfers is the fundamental read or fundamental write transfer. The fundamental transfer is a transfer that does not exhibit any of the properties listed above. It provides a reference point for describing how each transfer property affects the port and the behavior of the Avalon-MM signals. Using a specific property has one or all of the following effects:

■ Changes the behavior of certain signal types
■ Requires the use of one or more signal types to implement the property

Avalon-MM ports can support multiple properties simultaneously. For example, a particular Avalon-MM slave port might support pipelined transfers with variable wait-states. Some properties cannot be used in conjunction with other properties; such restrictions are noted in the discussion of each transfer property.

The master and slave ports in a master-slave pair can have different transfer properties. The system interconnect fabric communicates with each port using the port's specified properties, and translates properties from master port to slave port when necessary. In this way, Avalon-MM peripherals can be designed independently of the properties of the rest of the peripherals in the system.

3. Slave Transfers

This section defines the behavior of Avalon-MM slave transfers between a slave port and the system interconnect fabric. The interface between the system interconnect fabric and the slave port is the exclusive focus of this section, as shown in Figure 3.

Figure 3. Focus on Avalon-MM Slave Transfers
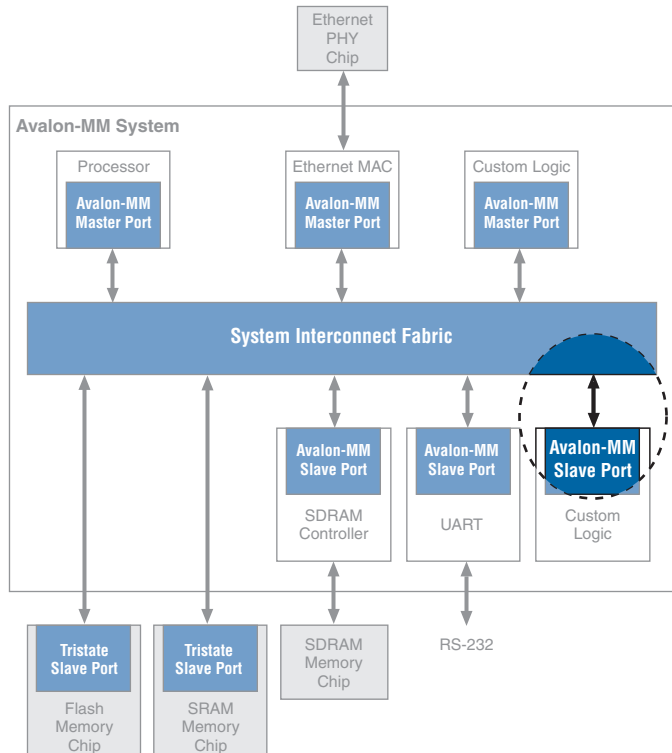
3.1. Slave Signal Details

This section describes noteworthy signal behavior that is true for all slave transfers. This section also highlights the flexibility a designer has in choosing Avalon-MM signals to meet the needs of a particular peripheral.

When a transfer is not occurring, the system interconnect fabric ignores all transfer-related output signals from the slave port. For exceptions, refer to Non-Transfer Related Signals on page 84.

3.1.1. address

The `address` signal for Avalon-MM slave ports is word addressable, specifying a word offset into the slave port's address space. Each slave address value accesses a full unit of data, based on the width of the slave port's `readdata` and/or `writedata` signals.

3.1.2. readdata and writedata

`readdata` and `writedata` are slave signals that carry the data associated with a transfer. A slave port can use one, none, or both of these signals. The width of these signals can be from 1 to 1024 bits wide. Slave ports must have data width of 8, 16, 32, 64, 128, 256, 512 or 1024. If a slave port uses both `readdata` and `writedata`, the widths must be equal for both signals.

3.1.3. chipselect, read, and write

The `chipselect`, `read`, and `write` signals are 1-bit inputs to the slave port that indicate when a new read or write transfer begins. These signals have different behavior, depending on what combination of these signals a slave port uses:

■ Ports with `chipselect` – If a port uses `chipselect`, the port must accept a transfer whenever the system interconnect fabric asserts `chipselect`, and ignore cycles when `chipselect` is deasserted. The system interconnect fabric always asserts `chipselect` in combination with `read` or `write`.
For a slave port with `chipselect`, the behavior of `read` and `write` depends on the following:

   ● If the port uses either the `read` or `write` signal alone, then the signal has additional significance. In this case, `read` also means `write_n` (i.e. not write), and `write` also means `read_n` (i.e. not read).
   ● If the port uses `chipselect`, and both `read` and `write`, then `chipselect` simply acts as a qualifier for the `read` and `write` signals. The slave port ignores any cycles while the system interconnect fabric is not asserting `chipselect`, regardless of the status of `read` or `write`.

■ Ports without `chipselect` – If a slave port does not use `chipselect`, then it uses `read` and/or `write` alone to determine when a new transfer begins. The system interconnect fabric asserts either `read` or `write` to initiate a transfer. The system interconnect fabric deasserts both signals to indicate an idle cycle. The system interconnect fabric never asserts both signals simultaneously.

The timing diagrams of transfers below demonstrate each transfer as an isolated event, while under realistic circumstances transfers can occur in succession. For example, after one read transfer terminates, `chipselect` and `read` might remain asserted if another transfer with this slave port follows on the next cycle.

When `chipselect` is deasserted, a slave port can ignore all input signals, except for `reset`.

3.1.4. byteenable and writebyteenable

The `byteenable` signal is a vector signal with one line for every byte lane in `writedata`. During write transfers to a slave port greater than 8 bits wide, the system interconnect fabric asserts the `byteenable` signal to specify which byte lane(s) to write. During read transfers to a slave port greater than 8 bits wide, the system interconnect fabric might assert different `byteenable` lines, indicating which specific bytes the requesting master will use. The slave port can return valid data on just the requested byte lane(s) or on all byte lanes.

When more than one byte lane is asserted, all asserted lanes are guaranteed to be adjacent. The number of adjacent lines must be a power of two, and the specified bytes must be aligned on an address boundary for the size of the data.

Table 3 shows all possible cases of `byteenable` during write transfers for a 32-bit slave port.

Table 3. Byte-Enable Example for a 32-Bit Slave Port	
Byteenable [3..0]	**Write Action**
1111	Write full 32-bits
0011	Writes lower 2 bytes
1100	Writes upper 2 bytes
0001	Write byte 0 only
0010	Write byte 1 only
0100	Write byte 2 only
1000	Write byte 3 only

For example, in the case of a 32-bit port, the valid `byteenable` combinations are: 0001, 0010, 0100, 1000, 0011, 1100, 1111. The following combinations are not valid: 0000, 0101, 0110, 0111, 1001, 1010, 1011, 1101, 1110.

The `writebyteenable` signal is the logical AND of the `write` and `byteenable` signals. A slave port can use `writebyteenable` instead of the separate `write` and `byteenable` signals to determine when and which byte(s) to write.

3.1.5. begintransfer

The `begintransfer` input signal can be used by any slave port, and provides an easy-to-understand indicator that a new slave transfer has been initiated. The system interconnect fabric asserts the `begintransfer` signal during the first cycle of each slave transfer. Usage is peripheral-specific. For example, a peripheral's core logic may use `begintransfer` to determine exactly when an Avalon-MM slave transfer begins, because the address, `readenable`, `writeenable,` and chipselect signals do not necessarily change at the start of each data transfer.

3.2. Slave Read Transfers

This section defines and demonstrates various Avalon-MM slave read transfers.

3.2.1. Fundamental Slave Read Transfer

The fundamental slave read transfer is the reference point for all other Avalon-MM slave read transfers. It is a read transfer absent any of the transfer properties allowed by the Avalon-MM specification.

The fundamental slave read transfer is initiated by the system interconnect fabric, and transfers one unit of data, the full width of the peripheral's data port, from the slave port to the system interconnect fabric. The transfer completes in a single clock cycle.

Figure 4 shows an example of the fundamental read transfer. The transfer starts on a rising clock edge, and the read transfer completes on the next rising clock edge. On the first rising edge of `clk`, the system interconnect fabric passes the address, `byteenable,` and `read` signals to the slave port. The system interconnect fabric decodes address internally, and drives the `chipselect` signal to the slave port. Once `chipselect` is asserted, the slave port drives `readdata` as soon as it is available. The system interconnect fabric captures `readdata` on the next rising edge of `clk`. For the transfer to complete in a single cycle, the slave port must immediately output the addressed content to the system interconnect fabric before the next rising edge of `clk`.

Figure 4. Fundamental Slave Read Transfer
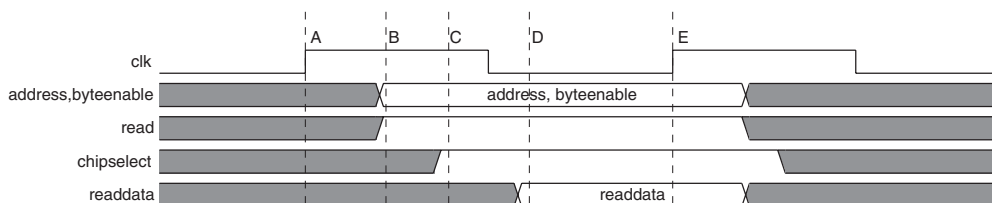
Notes to Figure 4:

(A)   First cycle starts on the rising edge of `clk`.

(B)   `address` and `read`from system interconnect fabric to slave port are valid

(C)   System interconnect fabric decodes `address` and asserts valid `chipselect`.

(D)   Slave port returns valid data on `readdata` during the first cycle.

(E)   System interconnect fabric captures `readdata` on the next rising edge of `clk`, and the read transfer ends. The next cycle begins here, and could be the start of another transfer.

The fundamental read transfer is appropriate only for asynchronous slave peripherals, such as asynchronous memory chips. The slave peripheral must return data immediately whenever it is selected and/or the address changes. The `readdata` signals must be valid and stable before the next rising clock edge.

Synchronous peripherals that register their Avalon-MM input or output signals must use wait-state and/or pipeline properties. On-chip Avalon-MM peripherals typically use a synchronous, registered interface that requires at least one clock cycle (i.e., one wait-state) to capture `address`.

3.2.2. Wait-States

Wait-states extend the read transfer, and give a slave port one or more clock cycles to capture `address` and/or return valid `readdata`. Wait-states affect the transfer throughput to a slave port. For example, a sustained sequence of transfers with zero wait-states can achieve a maximum of one transfer per clock cycle. With one wait-state, the maximum throughput is one transfer per two clock cycles.

There are two kinds of wait-states for slave read transfers: fixed and variable.

3.2.2.1. Slave Read Transfer with Fixed Wait-States
The set of slave signals used for a slave read transfer with fixed wait-states is identical the set used for the fundamental read transfer. The difference is the number of cycles after `chipselect` is asserted until the

slave port must present valid `readdata`. For example, with one fixed wait-state specified, the system interconnect fabric presents a valid `address` and asserts `chipselect`, but waits for one clock cycle before capturing the `readdata signal`. The system interconnect fabric asserts the address and control signals (`chipselect`, `byteenable`, `read`, *etc.)* for the duration of the transfer.

Figure 5 shows an example slave read transfer with one wait-state. The system interconnect fabric presents `address`, `byteenable`, `read and chipselect during` the first cycle. Because of the wait-state, the peripheral does not have to present `readdata` within the first cycle; the first cycle is the first (and only) wait-state. During the second cycle, the slave port presents its `readdata` to the system interconnect fabric. On the third and final rising clock edge, the system interconnect fabric captures `readdata` from the slave port, and completes the transfer.

Figure 5. Slave Read Transfer with One Fixed Wait-State
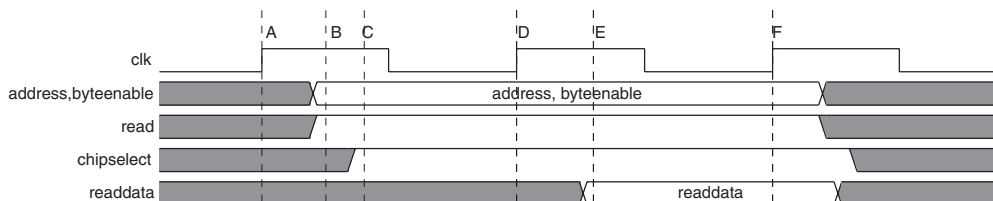
Notes to Figure 5:
(A)   First cycle starts on the rising edge of `clk`.
(B)   Signals `address` and `read` from system interconnect fabric to slave are valid.
(C)   System interconnect fabric decodes `address` and asserts `chipselect`.
(D)   Rising edge of `clk` marks the end of the first and only wait-state cycle. Slave port captures `address`, `byteenable`, `read` and `chipselect` on this rising edge of `clk`.
(E)   Slave port presents valid `readdata` during the second cycle.
(F)   System interconnect fabric captures `readdata` on the rising edge of `clk`, and the read transfer ends. The next cycle begins here and could be the start of another transfer.

Read transfers with a single wait-state are commonly used for synchronous, on-chip peripherals. The peripheral can capture address and control signals on the rising edge of `clk`, and then has one full cycle to present data back to the system interconnect fabric.

Figure 6 shows a read transfer with multiple fixed wait-states. This example uses two wait-states. This case is essentially the same as Figure 5, except that the system interconnect fabric now waits for more than one cycle before capturing `readdata` from the slave port.

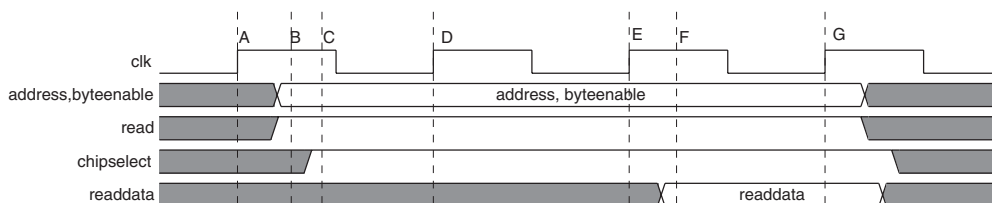*Figure 6. Slave Read Transfer with Multiple Fixed Wait-States*

Notes to Figure 6:

(A)   First cycle starts on the rising edge of `clk`.

(B)   Signals `address` and `read` from system interconnect fabric to slave are valid

(C)   Avalon-MM interface decodes address then asserts `chipselect`.

(D)   Rising edge of `clk` marks the end of the first wait-state cycle. Slave port registers `address`, `read` and `chipselect` on this rising edge of `clk`.

(E)   Rising edge of `clk` marks the end of the second (and last) wait-state.

(F)   Slave port presents valid `readdata` sometime during the third cycle.

(G)   System interconnect fabric captures `readdata` on the rising edge of `clk`, and the read transfer ends. The next cycle begins here and could be the start of another transfer.

3.2.2.2. Slave Read Transfer with Variable Wait-States

Variable wait-states allow a slave port to stall the system interconnect fabric for as many cycles as required to present data. A slave port with this transfer property can take a variable amount of time to present data to the system interconnect fabric. Using variable wait-states requires the Avalon-MM slave port to include the output signal `waitrequest`.

Figure 7 shows a slave read transfer with variable wait-states. The system interconnect fabric presents `address, byteenable, read` and chipselect during the first cycle, exactly like the start of a fundamental read transfer. The slave port must assert `waitrequest` within the first cycle to extend the read transfer. When asserted, `waitrequest` stalls the system interconnect fabric, causing it to hold address and control signals constant, and preventing it from capturing `readdata`. The system interconnect fabric will capture `readdata` on the next rising edge of `clk` after the slave port deasserts `waitrequest`, and the transfer terminates.

The system interconnect fabric does not have a time-out feature to limit how long a slave port can stall. While the system interconnect fabric is stalled, somewhere in the Avalon-MM system there is a master port that is also stalled. Therefore, peripheral designers must ensure that a slave port does not assert `waitrequest` indefinitely and thereby permanently stall a master peripheral.

Figure 7. Slave Read Transfer with Variable Wait-States
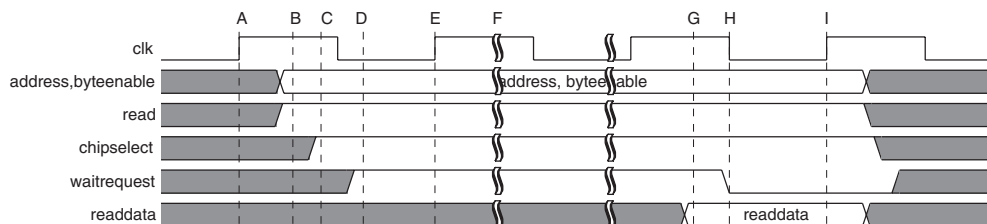
Notes to Figure 7:

(A) First cycle starts on the rising edge of `clk`.
(B) System interconnect fabric asserts `address` and `read` signals.
(C) System interconnect fabric decodes `address` then asserts `chipselect`.
(D) Slave port asserts `waitrequest` before the next rising edge of `clk`.
(E) System interconnect fabric samples `waitrequest` at the rising edge of `clk`; `waitrequest` is asserted, and therefore `readdata` is not captured on this clock edge.
(F) With `waitrequest` asserted throughout, an undefined number of cycles elapse.
(G) Slave port presents valid `readdata`.
(H) Slave port deasserts `waitrequest`.
(I) System interconnect fabric captures `readdata` on the next rising edge of `clk`, and the read transfer ends here. The next cycle begins here and could be the start of another transfer.

3.2.2.3. Restrictions

The following restrictions apply to ports that use wait-states:

■ If a port that uses variable wait-states is capable of both read and write transfers, the port must use variable wait-states for both read and write transfers.

■ If variable wait-states are specified, the slave port cannot also use setup and hold properties. In almost all cases, a peripheral that can generate the `waitrequest` signal will be on-chip and synchronous, making setup- and hold-time considerations unnecessary.

3.2.3. Setup Time

Some peripherals, most commonly asynchronous, off-chip devices, require `address` and `chipselect` signals to be stable for a period of time before the `read` signal is asserted. Avalon-MM transfers with setup time accommodate for such setup time requirements. The signals used for a read transfer with setup time are identical to those used for the fundamental read transfer. The difference is in the timing of signals only.

A nonzero setup time of means that, after the system interconnect fabric asserts `address` and `chipselect` to the slave port, there is a delay of *N* cycles before it asserts `read`. The total number of cycles to complete the

transfer depends on setup and wait-state cycles. For example, a slave port with 2 cycles of setup time and 3 cycles of wait-states will take 6 cycles to complete the transfer: 2 setup cycles, plus 3 wait-state cycles, plus 1 cycle to capture data.

Figure 8 shows a slave read transfers with one cycle of setup and one fixed wait-state.

Figure 8. Slave Read Transfer with Setup Time and Fixed Wait-State
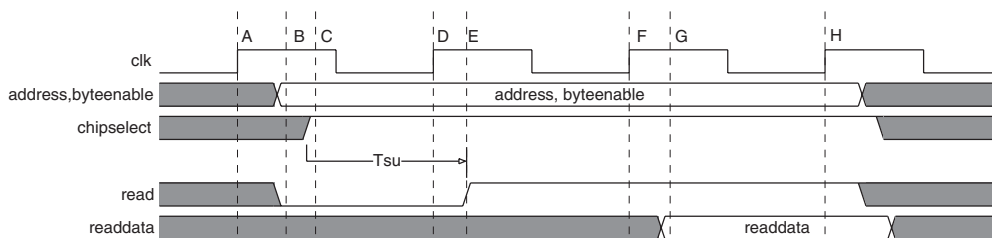
Notes to Figure 8:

(A) Transfer starts on the rising edge of `clk`. The first (and only) cycle of setup time begins here.
(B) System interconnect fabric asserts valid `address` and `byteenable`, but keeps `read`
(C) System interconnect fabric decodes `address` and asserts `chipselect`.
(D) Rising edge of `clk` defines the end of the setup–time cycle (Tsu), and the start of the wait-state cycle.
(E) System interconnect fabric asserts `read`.
(F) Rising edge of `clk` marks the end of the wait-state cycle.
(G) Slave port presents valid `readdata`.
(H) System interconnect fabric captures `readdata` at the rising edge of `clk`, and the transfer ends here. The next cycle begins here and could be the start of another transfer.

3.2.3.1. Restrictions

The following restrictions apply to slave ports that use setup time:

- If a slave port is capable of both a read and write transfer, and setup time is specified, then the same setup time is applied to both read and write transfers.
- Setup time cannot be used if the slave port uses variable wait-states.

3.2.4. Hold Time

By definition Avalon-MM slave read-transfers do not use hold time.

3.2.5. Pipeline, Burst, and Tristate Properties

For details on the Avalon-MM pipeline, burst, or tristate properties for slave transfers, refer to the respective sections devoted to each transfer property:

■ Section "Pipelined Transfers" on page 47.
■ Section "Tristate Transfers" on page 62.
■ Section "Burst Transfers" on page 72.

3.3. Slave Write Transfers

This section defines and demonstrates the Avalon-MM slave write transfers.

3.3.1. Fundamental Slave Write Transfer

The fundamental slave write transfer is the reference point for all other Avalon-MM slave write transfers. It is a write transfer absent any of the transfer properties allowed by the Avalon-MM specification.

The fundamental slave write transfer is initiated by the system interconnect fabric, and transfers one unit of data from the system interconnect fabric to the slave port. The transfer completes in a single clock cycle.

The `byteenable` signal provides a mechanism for the slave port to write only to specific bytes within `writedata`, if `writedata` is greater than one byte wide. If present, `byteenable` specifies which byte lane(s) to write. If the slave port does not use `byteenable`, all byte lanes are permanently enabled during write transfers.

Figure 9 shows the fundamental slave write transfer. The system interconnect fabric presents `address`, `writedata`, `byteenable`, and `write`. The system interconnect fabric decodes `address` internally, and drives the `chipselect` signal to the slave port. The slave port captures the address, data and control on the next rising clock edge, and the write transfer terminates immediately.

Figure 9. Fundamental Slave Write Transfer
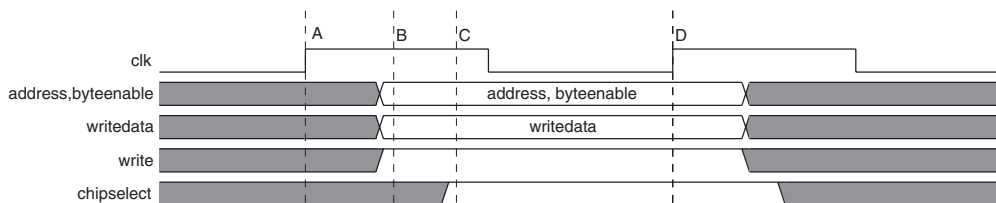
Notes to *Figure 9*:

(A) First cycle starts on the rising edge of `clk`.

(B) The system interconnect fabric asserts valid `writedata`, `address`,`byteenable`and `write`

(C) System interconnect fabric decodes `address` and asserts valid `chipselect` to slave.

(D) Slave port captures `writedata`, `address`, `write`, `byteenable` and `chipselect` on the rising edge of `clk`, and the transfer terminates. The next cycle begins here, and could be the start of another transfer.

When `chipselect` is deasserted, the slave port must ignore all other input signals, and the system interconnect fabric ignores any output signals from the slave port. A low-to-high edge on `chipselect` cannot be used as a trigger to start a write transfer, because such an edge is not guaranteed to occur.

The fundamental write transfer is generally appropriate for synchronous, on-chip peripherals that can capture data in a single clock cycle. Peripherals that cannot capture data in one clock cycle must use wait-states.

3.3.2. Wait-States

Wait-states extend the write transfer, and give a slave port one or more clock cycles to capture `address` and `writedata`. Wait-states affect the transfer throughput to a slave port. For example, a sustained sequence of transfers with zero wait-states achieves one transfer per clock cycle. With one wait-state, the maximum throughput is one transfer per two clock cycles.

There are two kinds of wait-states for slave write transfers: fixed and variable.

3.3.2.1. Slave Write Transfer with Fixed Wait-States

The slave port signals used for a write transfer with fixed wait-states are identical to those used for a fundamental write transfer. The difference is in how long the system interconnect module asserts the address, data and control signals. For example, with one fixed wait-state specified, the

system interconnect fabric waits for one additional clock cycle before deasserting the address, data and control signals. The system interconnect fabric asserts the address, data and control signals (`chipselect`, `byteenable`, `write`, *etc.)* for the duration of the transfer.

Write transfers with wait-states are typically used for peripherals that cannot capture data from the system interconnect fabric in a single cycle. In this transfer mode, the system interconnect fabric presents `address`, `writedata`, `byteenable`, `write` and `chipselect` during the first cycle, exactly like the start of a fundamental write transfer. During the wait-state(s), these signals are held constant. The slave port captures data from the system interconnect fabric within the fixed number of wait-states. The transfer then terminates, and the system interconnect fabric deasserts all signals at the same time. Figure 10 shows an example of a slave write transfer with one wait-state.

Figure 10. Slave Write Transfer with One Fixed Wait-State
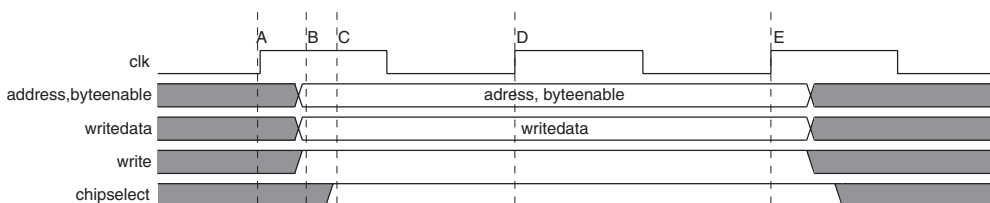
Notes to Figure 10:
(A)   First cycle starts on the rising edge of `clk`.
(B)   Signals `writedata`, `address`, `byteenable`, and `write` signals from system interconnect fabric are valid.
(C)   System interconnect fabric decodes `address` and asserts `chipselect`.
(D)   First wait-state cycle ends at the rising edge of `clk`. All signals from system interconnect fabric remain constant.
(E)   Slave port captures `writedata`, `address`, `byteenable`, `write`, and `chipselect` on or before the rising edge of `clk`, and the write transfer terminates. The next cycle begins here and could be the start of another transfer.

3.3.2.2. Slave Write Transfer with Variable Wait-States

Variable wait-states allow a target peripheral to stall the system interconnect fabric for as many cycles as required to capture `writedata`. This feature is useful for peripherals that require a variable number of cycles to capture the write data. Using variable wait-states requires the Avalon-MM slave port to include the output signal `waitrequest`.

Figure 11 shows an example of a slave write transfer with a variable wait-state. The system interconnect fabric presents `address`, `writedata`, `byteenable`, `write` and `chipselect` during the first cycle, exactly like the start of a fundamental write transfer. If the slave port needs extra

time to capture the data, it must assert `waitrequest` before the next rising clock edge. When asserted, `waitrequest` stalls the system interconnect fabric, and forces it to hold `address`, `writedata`, `byteenable`, `write` and `chipselect` constant. After the slave port deasserts `waitrequest`, the transfer terminates on the next rising clock edge.

The system interconnect fabric does not have a time-out feature to limit how long a slave port can stall. While the system interconnect fabric is stalled, somewhere in the Avalon-MM system there is a master port that is also stalled. Therefore, peripheral designers must ensure that a slave port does not assert `waitrequest` indefinitely and thereby permanently stall a master peripheral.

Figure 11. Slave Write Transfer with Variable Wait-States
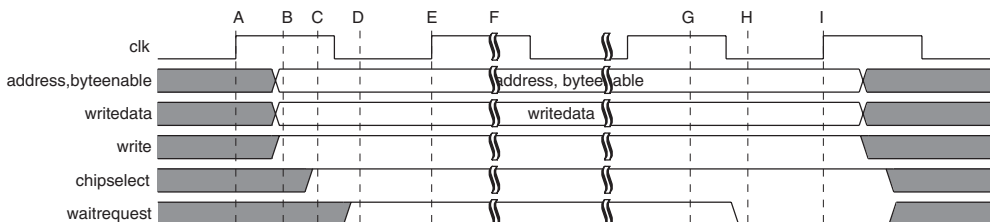
Notes to Figure 11:
(A)   First cycle starts on the rising edge of `clk`.
(B)   (Signals `address`, `writedata`, `byteenable` and `write` signals from system interconnect fabric to slave are valid.
(C)   System interconnect fabric decodes `address`, then asserts `chipselect`.
(D)   Peripheral asserts `waitrequest` before the next rising edge of `clk`.
(E)   System interconnect fabric samples `waitrequest` at the rising edge of `clk`. If `waitrequest` is asserted, the cycle becomes a wait-state, and `address`, `writedata`, `byteenable`, `write` and `chipselect` remain constant.
(F)   With `waitrequest` asserted throughout, an unlimited number of cycles elapse.
(G)   Eventually the slave port captures `writedata`.
(H)   Slave port deasserts `waitrequest`.
(I)   The write transfer ends on the next rising edge of `clk`. The next cycle could be the start of another transfer.

3.3.2.3. Restrictions
The following restriction apply to ports that use wait-states:

■   If a port that uses variable wait-states is capable of both read and write transfers, the port must use variable wait-states for both read and write transfers.

■ If variable wait-states are specified, the slave port cannot also use setup and hold properties. In almost all cases, a peripheral that can generate the `waitrequest` signal will be on-chip and synchronous, making setup- and hold-time considerations unnecessary.

3.3.3. Slave Write Transfer with Setup and Hold Times

Setup and hold time are generally used for off-chip peripherals that require `address`, `byteenable`, `writedata`, and chipselect to remain stable for some amount of time before and/or after the `write` pulse. The signals used for a write transfer with setup and hold times are identical to those used for a fundamental write transfer. The difference is in the timing of signals only.

A nonzero setup time of means that, after the system interconnect fabric asserts `address`, `byteenable`, `writedata` and `chipselect` to the slave port, there is a delay of M cycles before it asserts `write`. Likewise, a nonzero hold time of N means that, after `write` is deasserted, `address`, `byteenable`, `writedata` and `chipselect` remain constant for N more cycles. The total number of cycles to complete the transfer depends on setup, wait-state and hold cycles. For example, a slave port with 2 cycles of setup time, 3 cycles of wait-states, two 2 cycles of hold time will take 8 cycles to complete the transfer: 2 setup cycles plus 3 wait-state cycles plus 2 hold cycles plus 1 cycle to capture data.

A slave port does not have to use both setup and hold time at the same time; the Avalon-MM interface supports transfers with only setup time, only hold time, or both. Figure 9 shows a write transfer with both a setup and a hold time requirement.

Figure 12. Slave Write Transfer with Setup and Hold Times
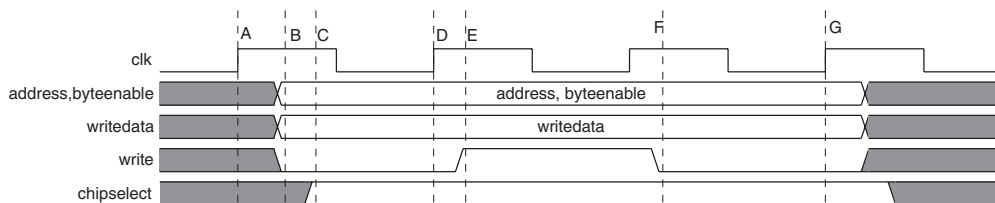
Notes to Figure 12:
(A) First cycle starts on the rising edge of `clk`.
(B) System interconnect fabric asserts `address`, `byteenable` and `writedata` signals from system interconnect fabric, but keeps `write` deasserted.
(C) System interconnect fabric decodes `address` and asserts `chipselect`.
(D) Rising edge of `clk` marks the end of the setup cycle.
(E) System interconnect fabric asserts `write`.
(F) System interconnect fabric deasserts `write` after the next rising edge of `clk`. Signals `address`, `byteenable`, `writedata` and `chipselect` remain constant as the hold-time cycle begins.
(G) System interconnect fabric deasserts `address`, `byteenable`, `writedata` and `chipselect` on the next rising edge of `clk` and the write transfer terminates.

3.3.3.1. Restrictions
The following restrictions apply to ports that use setup and/or hold time:

■   If the port is capable of both read and write transfers, the same setup time applies to both read and write transfers.
■   The port cannot also use variable wait-states.

3.3.4. Pipeline, Burst and Tristate Properties

For details on the Avalon-MM pipeline, burst, and tristate properties for slave ports, see the respective sections devoted to each transfer property:

■   Section "Pipelined Transfers" on page 47.
■   Section "Tristate Transfers" on page 62.
■   Section "Burst Transfers" on page 72.

4. Master Transfers

This section defines the behavior of Avalon-MM master transfers between a master port and the system interconnect fabric. The interface between the system interconnect fabric and the master port is the exclusive focus of this section, as shown in Figure 13.

Figure 13. Focus of Avalon-MM Master Transfers
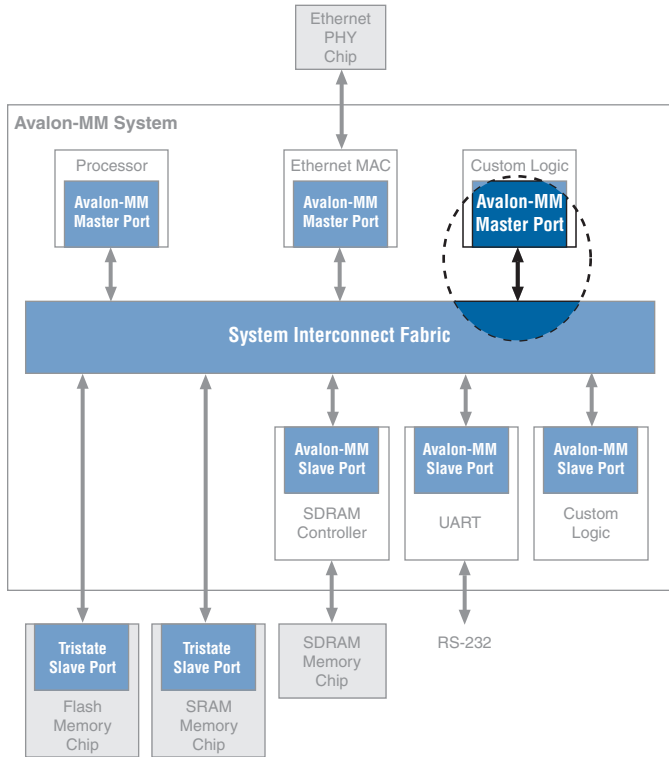
4.1. Master Signal Details

This section describes noteworthy signal behavior that is true for all master transfers.

When a transfer is not occurring, the system interconnect fabric ignores all transfer-related output signals from the master port, and the master port can ignore all transfer-related signals from the system interconnect fabric. For exceptions, see "Non-Transfer Related Signals" on page 81.

4.1.1. waitrequest

The `waitrequest` signal is a master port input that indicates that the system interconnect fabric is not ready to proceed with a transfer. There is one golden rule that applies to all master transfers: Obey the `waitrequest` signal.

At the start of all transfers, a master port asserts the appropriate signals to initiate the transfer, and then waits until the system interconnect fabric deasserts `waitrequest`.

The system interconnect fabric deasserts `waitrequest` when not performing a transfer with a master port.

4.1.2. address

Master addresses represent byte addresses, regardless of the data-width of the master port. A master port can assert only addresses aligned to word boundaries, based on the master port's data width. For example, a 32-bit master port can assert only addresses aligned to 4-byte boundaries, such as 0x00, 0x04, 0x08, 0x0C, etc. In this case, the system interconnect fabric ignores the lower two bits of `address`. To write to a specific byte within a data word, the master port must use the `byteenable` signal.

4.1.3. readdata and writedata

The `readdata` and `writedata` signals carry the data associated with a transfer. A master port can use one, none, or both of these signals. These signals must be 8, 16, 32, 64, 128, 256, 512 or 1024 bits wide. If a master port uses both `readdata` and `writedata`, the widths must be equal for both signals.

4.1.4. read and write

The `read` and `write` signals are 1-bit outputs from the master port to indicate when it is about to start a new read or write transfer.

The timing diagrams of transfers below demonstrate each transfer as an isolated event, but under realistic circumstances transfers can occur in succession. For example, after the master port terminates a read transfer, it can continue asserting `read` to assert another read transfer on the next cycle.

4.1.5. byteenable

The byteenable signal is a vector signal with one line for every byte lane in writedata. During write transfers, a master port greater than 8 bits wide can assert the byteenable signal to specify which byte lane(s) to write. During read transfers, a master port greater than 8 bits wide can assert the byteenable signal to specify which byte lane(s) to read; only the specified lanes of readdata or data are guaranteed to be valid.

When more than one byte lane is asserted, all asserted lanes must be adjacent. The number of adjacent lines must be a power of two, and the specified bytes must be aligned on an address boundary for the size of the data.

Table 4 shows some example cases of byteenable during write transfers for a 32-bit master port.

Table 4. Byte-Enable Example for a 32-Bit Slave Port	
Byteenable [3..0]	**Write Action**
1111	Write full 32-bits
0011	Writes lower 2 bytes
1100	Writes upper 2 bytes
0001	Write byte 0 only
0100	Write byte 2 only

For example, in the case of a 32-bit port, the valid byteenable combinations are: 0001, 0010, 0100, 1000, 0011, 1100, 1111. The following combinations are not valid: 0000, 0101, 0110, 0111, 1001, 1010, 1011, 1101, 1110.

4.2. Fundamental Master Read Transfers

The fundamental master read transfer is the reference point for all other Avalon-MM master read transfers. It is a read transfer absent any of the transfer properties allowed by the Avalon-MM specification.

The fundamental master read transfer is initiated by the master peripheral, and transfers one unit of data from the system interconnect fabric to the master port. In the fastest possible case, the transfer terminates in one cycle. If readdata is not ready, the system interconnect fabric asserts waitrequest and stalls the master port until it can present the data. The transfer terminates when the system interconnect fabric deasserts waitrequest, and the master port captures readdata.

If the system interconnect fabric asserts `waitrequest` for *N* cycles, then the total transfer takes (+ 1) cycles. The system interconnect fabric does not offer a time-out feature to the master port; the master port must stall for as long as `waitrequest` remains asserted.

A master port can use the `byteenable` signal to indicate that it only requires data for specific byte lanes, if `readdata` is more than one byte wide. If a master port does not use the `byteenable` signal, the transfer proceeds as if all byte enable lines are asserted.

The master read transfer starts on the rising edge of `clk`. During the first cycle, the master port asserts the `address`, `byteenable`, and `read` signals. If the system interconnect fabric cannot present `readdata` within the first cycle, it asserts `waitrequest` before the next rising edge of `clk`. When `waitrequest` is asserted at the rising edge of `clk`, the master port must hold all outputs constant through the next cycle. After `waitrequest` is deasserted, the master port captures `readdata` on the next rising edge of `clk`, and deasserts `address` and `read`. If not all `byteenable` are asserted, only the specified lanes of `readdata` are guaranteed to be valid. The master may initiate another transfer immediately on the next cycle.

Figure 14 shows the fundamental master read transfer. In Figure 14 `waitrequest` is never asserted by the system interconnect fabric, and the read transfer ends in one cycle.

Figure 14. Fundamental Master Read Transfer with No Wait-States

Notes to Figure 14:
(A) First cycle starts on the rising edge of `clk`.
(B) Master port asserts valid `address`, `byteenable` and `read`.
(C) Valid `readdata` returns from the system interconnect fabric during first cycle.
(D) Master port captures `readdata` on the next rising edge of `clk` and deasserts all its outputs. The read transfer ends here and the next cycle could be the start of another transfer.

Figure 15 shows the case of the system interconnect fabric asserting `waitrequest` for multiple cycles.

Figure 15. Master Read Transfer with Wait-States

Notes to Figure 15:

(A)   First cycle starts on the rising edge of `clk`.

(B)   Master asserts valid `address`, `byteenable` and `read`.

(C)   System interconnect fabric asserts `waitrequest` before the next rising edge of `clk`.

(D)   Master port accepts `waitrequest` at the rising edge of `clk`. This cycle becomes a wait-state.

(E)   As long as `waitrequest` is asserted, master port holds all outputs constant.

(F)   Valid `readdata` returns from system interconnect fabric.

(G)   System interconnect fabric deasserts `waitrequest`.

(H)   Master port captures `readdata` on the next rising edge of `clk` and deasserts all outputs. The read transfer ends here, and the next cycle could be the start of another transfer.

4.3. Fundamental Master Write Transfers

The fundamental master write transfer is the reference point for all other Avalon-MM master write transfers. It is a write transfer absent any of the transfer properties allowed by the Avalon-MM specification.

The fundamental master write transfer is initiated by the master peripheral and transfers one unit of data from the master port to the system interconnect fabric. If the system interconnect fabric cannot immediately capture the data, it asserts the `waitrequest signal` and stalls the master. In the fastest possible case, the system interconnect fabric does not assert `waitrequest`, and the transfer terminates in one cycle.
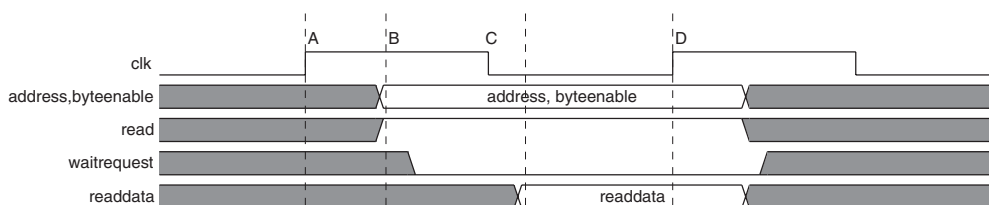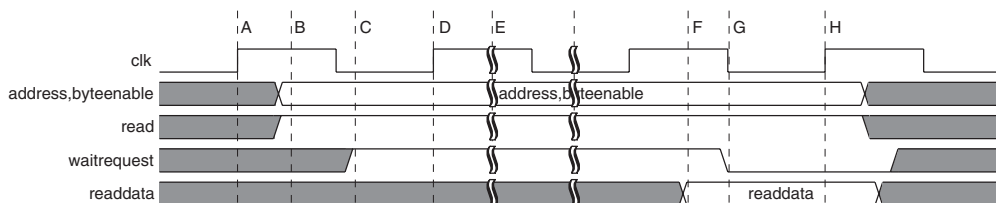
If the system interconnect fabric asserts `waitrequest for N` cycles, then the total transfer takes (+ 1) cycles. The system interconnect fabric does not offer a time-out feature to the master port; the master port must stall for as long as `waitrequest` remains asserted.

A master port can use the `byteenable` signal to write to individual bytes in `writedata`, if `writedata` is more than one byte wide. If a master port does not use the `byteenable` signal, the system interconnect fabric permanently enables all byte lanes for all write transfers from this master port.

The master write transfer starts on the rising edge of `clk`. Immediately after the first rising edge of `clk`, the master asserts the `address`, `byteenable`, `writedata` and `write` signals. If the system interconnect fabric cannot capture `writedata` within the first cycle, it asserts `waitrequest` before the next rising edge of `clk`. When `waitrequest` is asserted at the rising edge of `clk`, the master port must hold all outputs constant through the next cycle. After `waitrequest` is deasserted, the master port deasserts `address`, `byteenable`, `writedata` and `write` on the next rising edge of `clk`. The master may initiate another transfer immediately on the next cycle.

Figure 16 shows an example of a fundamental master write transfer. In this example, the system interconnect fabric does not assert `waitrequest` and the transfer terminates in one cycle.

Figure 16. Fundamental Master Write Transfer with No Wait-States
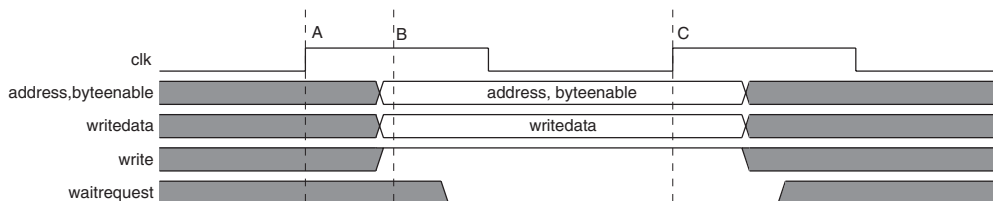
Notes to Figure 16:
(A)   Write transfer starts on the rising edge of `clk`.
(B)   Master asserts valid `address`, `byteenable`, `writedata`, and `write`.
(C)   `waitrequest` is not asserted at the rising edge of `clk`, so write transfer terminates. Another transfer could follow on the next cycle.

Figure 17 shows an example in which `waitrequest` is asserted by the system interconnect fabric for two cycles. The entire write transfer takes three cycles.

Figure 17. Master Write Transfer with Wait-States
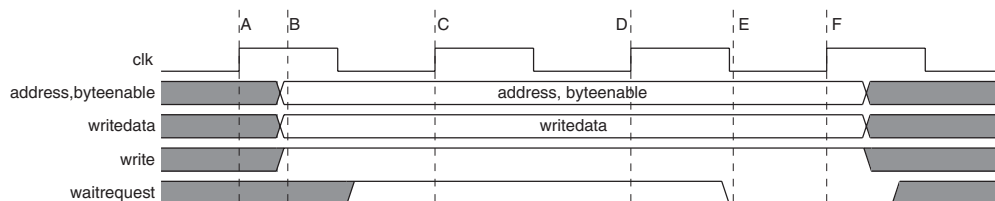
Notes to Figure 17:

(A) First cycle starts on the rising edge of `clk`.

(B) Master asserts valid `address`, `writedata` and `write`.

(C) `waitrequest` is asserted at the rising edge of `clk`, so this cycle becomes the first wait-state. Master holds all outputs constant.

(D) `waitrequest` is asserted at the rising edge of `clk` again, so this becomes the second wait-state. Master holds all outputs constant.

(E) System interconnect fabric deasserts `waitrequest`.

(F) `waitrequest` is not asserted at the rising edge of `clk`, so master deasserts all outputs, and the write transfer terminates. Another read or write transfer may follow on the next cycle.

4.4. Wait-State, Setup Time, and Hold Time Properties

By definition all Avalon-MM master transfers use the `waitrequest` signal to accept an unspecified number of wait-states when required by the system interconnect fabric. In this sense, all Avalon-MM master ports compulsorily support variable wait-states. Master ports do not support fixed wait-states.

By definition Avalon-MM master transfers do not use setup time or hold time. If a target slave peripheral has setup- and/or hold-time properties, the system interconnect fabric manages the translation of signal timing appropriately for the master-slave pair.

4.5. Pipeline, Burst, and Tristate Properties

For details on pipeline, burst, and tristate properties for master ports, see the respective sections devoted to each transfer property:

- Section "Pipelined Transfers" on page 47
- Section "Tristate Transfers" on page 62
- Section "Burst Transfers" on page 72

5. Pipelined Transfers

Avalon-MM pipelined read transfers increase the bandwidth for synchronous slave peripherals that require several cycles to return data for the first access, but can return data every cycle thereafter. Using pipelined read transfers, a port can begin a new transfer before `readdata` for the previous transfer returns. There are only pipelined read transfers; Avalon-MM write transfers do not benefit from pipelined functionality.

The duration of a pipelined read transfer is divided into two distinct phases: Address phase and data phase. A master port initiates a transfer (i.e. fills the pipeline) 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. This delay gives rise to *pipeline latency*, which is the duration from the end of the address phase to the end of the data phase, in other words, the duration of the data phase.

The duration of the address phase (i.e., the number of clock cycles required to capture the address) determines a port's throughput; a longer address phase diminishes throughput. The duration of the data phase reflects only how long it takes for the first unit of data to return. This is the key difference between how wait-states and pipeline latency affect transfer timing:

- *Wait-states*—Wait-states determine the length of the address phase, and limit the maximum throughput of a port. For example, if a slave port requires one wait-state to respond to a transfer request, then the port requires at least two clock cycles per transfer. An Avalon-MM slave port with no wait-states can accept a new transfer on every clock cycle.
- *Pipeline Latency*—Pipeline latency determines the length of the data phase, independently of the address phase. For example, 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.

The pipeline latency can be either fixed or variable, as discussed in the following sections.

5.1. Slave Pipelined Read Transfer with Fixed Latency

An Avalon-MM pipelined slave port takes one or more cycles to produce data after address and control signals have been captured from the system interconnect fabric. After the slave port captures the address, the system interconnect fabric may immediately initiate a new transfer, even before valid `readdata` has returned from the previous transfer. As a

result, a pipelined slave port might have multiple transfers pending at any given time. The set of slave signals used for pipelined transfers with fixed latency is identical to the set used for the fundamental read transfer. The difference is in the signal timing of the address and data phases.

The timing and sequence of signals during the address phase is identical to the fundamental transfer, except for the `readdata` signal. During the address phase, the slave port can use wait-states. The address phase ends on the next rising edge of `clk` after wait-states (if any) finish. The slave port must capture `address` by the last rising clock edge of the address phase. The slave port does not assert `readdata` for this transfer during the address phase. Immediately after the address phase completes, the system interconnect fabric can initiate a new transfer.

During the data phase, the peripheral processes the address over multiple clock cycles and then produces `readdata` after a fixed latency. If the peripheral has a read latency of the slave port must present valid `readdata` on the Nth rising edge of `clk` after the end of the address phase. The data phase, and the whole transfer, ends N cycles after the address phase, on the rising edge of `clk`. For example, if the slave port has a read latency of 1, the slave port presents valid `readdata` on the next (i.e., the first) rising edge of `clk` after capturing address.

Figure 18 shows multiple data transfers between the system interconnect fabric and a slave pipelined port that uses variable wait-states and has a fixed read latency of two cycles.

Figure 18. Slave Pipelined Read Transfer with Fixed Latency
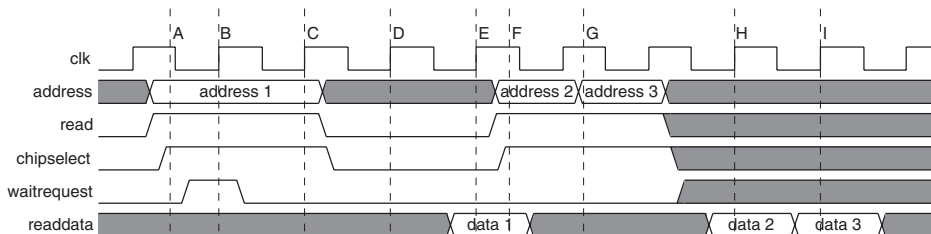
Notes to Figure 18:

(A) System interconnect fabric initiates a read transfer by presenting `chipselect`, `read and address` for the address phase of the new transfer.

(B) The slave port has asserted `waitrequest` so the previous cycle becomes a wait-state. The system interconnect fabric holds `chipselect`, `redound address` constant.

(C) The slave port deasserts `waitrequest` and captures `address` at the rising edge of `clk`. The address phase ends and the data phase starts here.

(D) First latency cycle ends this rising edge of `clk`.

(E) Second latency cycle ends on rising edge of `clk`. The slave data port presents valid `readdata`, and the transfer ends here. This edge of `clk` also marks the beginning of a new read transfer.

(F) System interconnect fabric asserts `address`, `read` and `chipselect` for the new read transfer.

(G) System interconnect fabric issues another read transfer during the next cycle, before the data from the prior transfer returns.

(H) System interconnect fabric captures `readdata` after two latency cycles.

(I) System interconnect fabric captures `readdata` after two latency cycles.

5.2. Slave Pipelined Read Transfer with Variable Latency

Pipelined slave read transfers with variable latency allow a slave port to return valid `readdata after` a variable number of latency cycles. Slave ports with variable latency use an additional signal `readdatavalid` to mark when the slave port presents valid data to the system interconnect fabric. Using the one-bit output signal `readdatavalid` defines a slave port to be pipelined with variable latency.

The address phase is identical to the pipelined slave read transfer with fixed latency. After the address phase, a pipelined slave port with variable read latency can take an arbitrary number of clock cycles to return valid `readdata`. When the peripheral is ready to return valid data, it asserts `readdata` and `readdatavalid` simultaneously and holds the signals until the next rising edge of `clk`. The system interconnect fabric captures `readdicting readdatavalid on` this clock edge, and the data phase (and the whole transfer) ends.

The slave port must return `readdata` in the same order that it accepted the addresses. Pipelined slave ports with variable latency must return `readdata` at least one clock cycle after the address phase ends.

Pipelined slave ports with variable latency commonly use variable wait-states. In practice a pipelined slave port can handle only a finite number of pending transfers. The slave port can assert the `waitrequest` signal to stall new transfers until it reduces the number of pending transfers.

☞ The maximum number of pending transfers is parameterizeable. For example, when a custom memory controller is instantiated using the SOPC Builder MegaWizard configuration wizard you set the number of pending read transactions. This parameter sets the size of a FIFO that records information about pending read transfers. If this FIFO is not large enough to hold all information about all pending read transfers, data will be lost. Figure 19 illustrates a system with a Nios II processor whose instruction and data masters are connected to an off-chip DDR2 memory. Each master is connected to an additional off-chip memory. The instruction master grant FIFO and data master grant FIFO record the grants made by the slave arbiter. The readdata that the master receives is the AND of the `readdatavalid` and `grant` signals.

Figure 19. SOPC Builder System with Variable Latency Pipelined Read Transfers
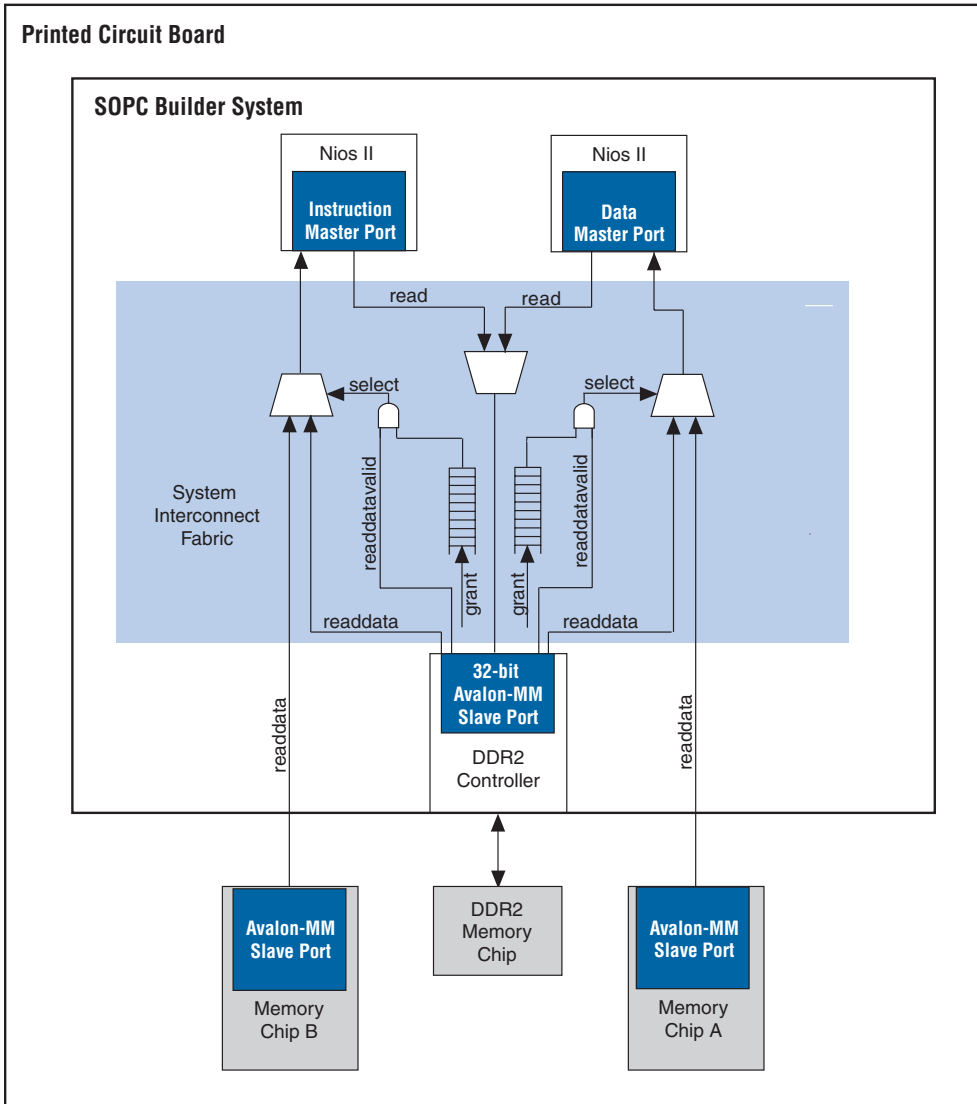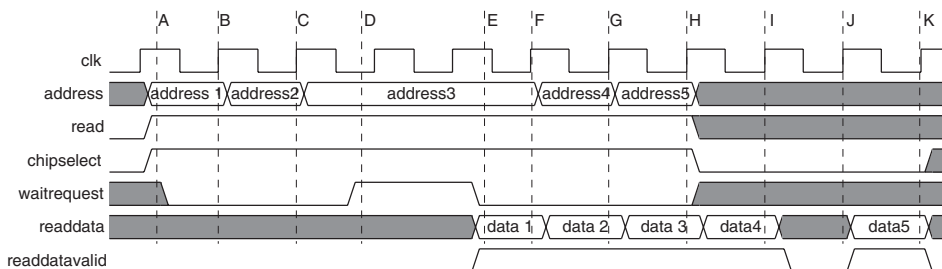
Figure 20 shows several slave read transfers between the system interconnect fabric and a pipelined slave port with variable latency. In this example, the slave port can only accept a maximum of 2 pending transfers, and it uses variable wait-states to prevent overrunning this maximum.

Figure 20. Slave Pipelined Read Transfers with Variable Latency

Notes to Figure 20:
(A) The system interconnect fabric asserts `address`, `read`, and `chipselect`, initiating a read transfer. Assume that there are no pending transfers at this point.
(B) The slave port is not asserting `waitrequest` and therefore captures address1 on this rising edge of `clk`.
(C) The slave peripheral is not asserting `waitrequest` and therefore captures address2 on this rising edge of `clk`.
(D) The slave port has reached its maximum number of allowed pending transfers, and does not have valid data to return. The peripheral asserts `waitrequest` before the next rising edge of `clk`, causing the system interconnect fabric to continue asserting `address`, `read`, and `chipselect`. The peripheral asserts `waitrequest` through two cycles until it can return data for the first pending transfer.
(E) The peripheral drives valid `readdata` (data1) and asserts `readdatavalid`, completing the data phase for the first pending transfer. The peripheral deasserts `waitrequest` because it can accept another pending transfer on the next rising edge of `clk`.
(F) The system interconnect fabric captures data1 on this rising edge of `clk`. The slave peripheral captures address3 on this rising edge of `clk`.
(G) The system interconnect fabric captures data2 on this rising edge of `clk`, because the slave port is asserting `readdatavalid` (Note that data1 and data2 required 4 cycles of latency to return). The system interconnect fabric asserts `address`, `read`, and `chipselect`, and the peripheral captures address4.
(H) The system interconnect fabric captures data3 on this rising edge of `clk`, because the slave port is asserting `readdatavalid`. (Note that data3 required 2 cycles of latency to return.) The system interconnect fabric is asserting `address`, `read`, and `chipselect`, and the peripheral captures address5.
(I) The system interconnect fabric captures data4 on this rising edge of `clk`, because the slave port is asserting `readdatavalid`. The system interconnect fabric deasserts `chipselect`, ending the sequence of read transfers.
(J) The system interconnect fabric does not capture data on this edge of `clk` because the slave port has deasserted `readdatavalid`.
(K) The system interconnect fabric captures `data5` on this rising edge of `clk`, completing the data phase for the final pending read transfer.

The system interconnect fabric can initiate a slave write transfer even while the slave peripheral is processing one or more pending read transfers. If the peripheral cannot handle a write transfer while it is

processing pending read transfers, the slave port 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 port accepts a write transfer to the same address as a currently pending read transfer. The result of the pending read transfer is peripheral-dependent. Peripheral designers must specify the behavior of their logic under this circumstance, or explicitly leave the behavior undefined.

5.2.1. Restrictions

The following restrictions apply to pipelined slave ports:

- Pipelined slave ports with variable latency cannot use the fixed wait-state property. Variable wait-states are supported.
- Pipelined slave ports cannot use the setup and hold time properties.
- Pipelined slave ports with variable latency cannot have the tristate property.

5.3. Master Pipelined Read Transfer

A pipelined master peripheral can initiate a new read transfer before it receives valid data from a previous transfer. Using the one-bit input signal `readdatavalid` defines a master port to be pipelined. The system interconnect fabric asserts `readdatavalid` to the master port to indicate that the `readdata` signal is presenting valid data.

The timing and sequence of signals during the address phase is identical to that of the fundamental Avalon-MM master read transfer, except for the `readdata` signal. The master port must present `read` and `address` and (n present) `byteenable`, and must hold these signals constant as long as its `waitrequest` input is asserted. The address phase ends on the first rising edge of `clk` that `waitrequest` is not asserted. Immediately after the address phase completes, the master port can initiate another read or write transfer.

For pipelined transfers, `readdata` does not necessarily return immediately after the address phase. Valid `readdata` returns sometime later when the system interconnect fabric asserts `readdatavalid`. The system interconnect fabric always returns valid `readdata` in the same order as requested by the master port. There is no time limit on when the system interconnect fabric asserts `readdatavalid`. Pipelined master ports can have an arbitrary number of read transfers pending at any given time. The maximum number is determined by the peripheral design.

Pipelined master ports can optionally use the `flush` signal, which is provided for cases in which a master peripheral determines that it does not need the data for all currently pending transfers. For example, flushing the pipeline is a common requirement for pipelined CPUs that prefetch instructions before knowing if the instructions are valid or not. When the master port asserts `flush` on the rising edge of `clk`, `readdatavalid` is deasserted until the next new read transfer's data is valid on the `readdata` port. The master port can initiate a new read transfer during the same clock cycle that `flush` is asserted. In this case, the data corresponding to this transfer becomes the next valid data to return on `readdata`.

Figure 21 shows several pipelined master read transfers between a master port and the system interconnect fabric. In this example, there is no pattern to why and when the system interconnect fabric asserts `waitrequest` and `readdatavalid`. The purpose is to demonstrate that the master port must respond appropriately to both `waitrequest` and `readdatavalid`, no matter why or when they are asserted. In this example, the second-to-last transfer is flushed using the `flush` signal. However, the unwanted data might have appeared on `readdata` if the latency for that transfer was shorter.

Figure 21. Master Pipelined Read Transfer
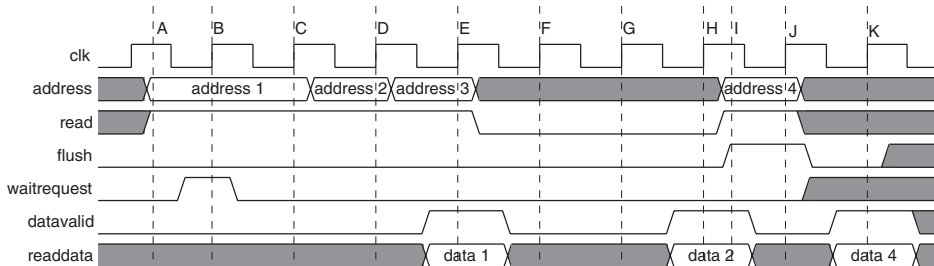
Notes to Figure 21:

(A)   Master initiates a read transfer by presenting `address` and `read` for the address phase of the new transfer.

(B)   System interconnect fabric is asserting `waitrequest`, so the master port waits and asserts `address` and `read` for another cycle.

(C)   The system interconnect fabric deasserts `waitrequest`, and captures `address` at the next rising edge of `clk`. `readdatavalid` is not asserted, so master does not capture `readdata`.

(D)   The system interconnect fabric captures a new address at the rising edge of `clk`. `readdatavalid` is not asserted, so master does not capture `readdata`.

(E)   The system interconnect fabric captures a new address at the rising edge of `clk` (making a total of three pending transfers). `readdatavalid` is asserted, so the master captures valid `readdata` (data 1).

(F)   `readdatavalid` is not asserted, so master does not capture `readdata`.

(G)   `readdatavalid` is not asserted, so master does not capture `readdata`.

(H)   `readdatavalid` is asserted, so master captures valid `readdata` (data 2).

(I)   Master presents `address` and `read` for a new read transfer.

(J)   `readdatavalid` is not asserted, so master does not capture `readdata`. Master asserts `flush`, causing the system interconnect fabric to flushes the pending transfer (address 3). System interconnect fabric captures the new address.

(K)   `readdatavalid` is asserted, so master captures valid `readdata` (data 4). At this point, no transfers are pending.

6. Flow Control

Avalon-MM flow control signals provide a mechanism for a slave port to regulate incoming transfers from a master port, so that a transfer only begins when the slave port indicates that it has valid data or is ready to receive data. The flow control signals provide the following benefits:

■   Simplifies logic design, because the master port does not have to repeatedly poll the slave port to determine whether it is ready to transfer data.

■   Reduces overhead bandwidth, because the slave transfer begins only when the slave port is ready.

■   Allows a slave port to control the flow of data to/from an "unintelligent" master port that unconditionally and continuously initiates transfers.

On the slave side, flow control signals let a slave port declare its readiness for a transfer before a transfer occurs. On the master side, a master port with flow control agrees to trust the slave flow control signals, and wait until the slave port is ready to proceed with a transfer.

For flow control to work, both ports in the master-slave pair must use flow control. If one or both of the ports does not use flow control, then the transfer proceeds as if neither port had it. For example, if a master port does not use flow control, then a slave port's flow control signals will not defer the master transfer.

6.1. Restrictions

Flow control signals cannot be used with Avalon-MM tristate ports.

6.2. Slave Transfers with Flow Control

To use flow control, a slave port can use one or more of the following signals: readyfordata, dataavailable, and endofpacket. A slave port with flow control is defined as a slave port that uses one or more of these signals. The flow control property does not affect the sequencing or timing of other signals.

6.2.1. Flow Control Signals

This section describes the slave signals used for flow control.

6.2.1.1. readyfordata and dataavailable
A slave port indicates that it is ready to accept a write transfer by asserting readyfordata; deasserting readyfordata means that writing will cause data overflow. A slave port indicates that it is ready to produce data for a read transfer by asserting dataavailable; deasserting dataavailable means that reading will cause data underflow.

In a master-slave pair that uses flow control, after a master port initiates a transfer, the system interconnect fabric initiates a transfer with the target slave port only if the readyfordata or dataavailable signals indicate that the slave port it is ready for the transfer. While the slave port is not ready, the system interconnect fabric forces the master port to wait.

Deasserting either signal does not prevent the system interconnect fabric from initiating a transfer from a master port that does not use flow control. For this reason, a slave port must always be ready for a transfer to start, regardless of the status of readyfordata and dataavailable.

6.2.1.2. endofpacket

During any transfer, a slave port with flow control can assert the endofpacket signal, which is passed through the system interconnect fabric to the master port. The interpretation of the endofpacket signal is dependent on the peripheral design, and the peripheral design must specify how a master port should respond to endofpacket. For example, endofpacket can be used as a packet delineator, to mark the boundary where packets start and end within a longer stream of data. Alternately, endofpacket can indicate that the master port should stop the current sequence of transfers. Depending on the peripheral design, the slave port can assert endofpacket for a single clock cycle, or it can assert endofpacket indefinitely until a master port explicitly resets the slave logic. The master port might not use the endofpacket signal, so the slave logic must be able to continue even if a master port does not detect endofpacket.

6.2.2. Slave Read Transfers with Flow Control

Slave read transfers with flow control can use either of the signals dataavailable and endofpacket.

A slave port can assert dataavailable at any time. While dataavailable is asserted, a new transfer from a master port with flow control can begin on the next rising edge of clk. A slave port can deassert dataavailable only at the end of a read transfer. The signal is immediately valid for successive transfers that might follow. If the slave port uses the endofpacket signal, it must assert endofpacket on the same clock edge that it asserts valid readdata.

Figure 22 shows an example of slave read transfers to a slave port using flow control. In this example, assume that a master port with flow control initiates a sequence of transfers while the slave port has dataavailable asserted, and the master port continues initiating read transfers in immediate succession. At some point during the sequence, the slave port deasserts dataavailable, causing the system interconnect fabric to stop initiating transfers. Later, the slave port asserts dataavailable again, and the system interconnect fabric continues the sequence of slave read transfers.

Figure 22. Slave Read Transfer with Flow Control
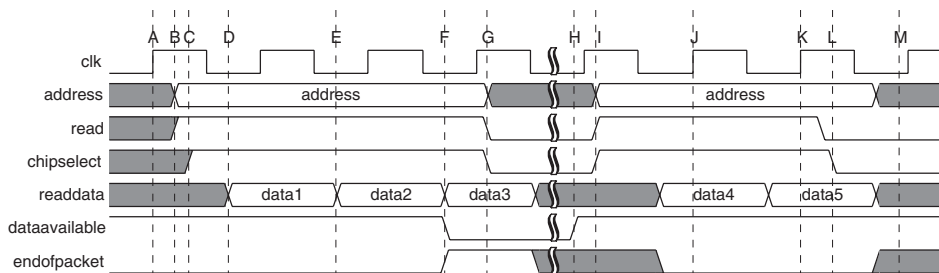
Notes to Figure 22:

(A)	The transfer begins on the rising edge of `clk`.

(B)	System interconnect fabric asserts `address` and `read`.

(C)	System interconnect fabric decodes `address`, and asserts `chipselect`.

(D)	Slave port asserts valid `readdata`. The system interconnect fabric captures `readdata` on the next rising edge of `clk`.

(E)	For each cycle that `chipselect` and `read` remain asserted, the slave port produces valid `readdata`. (In this example, `address` remains constant, but this is not necessarily the case for all peripheral designs.)

(F)	The slave port asserts `endofpacket` as it asserts valid `readdata`. (In this example, the slave port deasserts `endofpacket` after one cycle, but this is not necessarily the case for all peripheral designs.) The slave port also deasserts `dataavailable`, forcing the system interconnect fabric to postpone subsequent read transfers from the master port with flow control.

(G)	The system interconnect fabric deasserts `address`, `read` and `chipselect` in response to `dataavailable`.

(H)	Some time later, the slave port asserts `dataavailable`.

(I)	In response to `dataavailable` and because the master port is still waiting to transfer data, the system interconnect fabric starts a new transfer, reasserting `address`, `read` and `chipselect`.

(J)	The system interconnect fabric captures data4 on the rising edge of `clk`.

(K)	The slave port asserts valid `readdata` for every cycle that `chipselect` and `read` remain asserted.

(L)	The system interconnect fabric deasserts `read` and `chipselect`, ending the sequence of transfers.

(M)	In this example `dataavailable` remains asserted, meaning that the system interconnect fabric can begin another read transfer at any time.

In this example, data is read from a constant slave address that presents new data for each transfer, which is common for I/O peripherals. In this example, the slave port asserts `endofpacket` on the last unit of data before it deasserts `dataavailable`. This is not a requirement; `endofpacket` has no inherent relationship to `dataavailable` nor to how the master peripheral responds. The sequence of transfers finishes while the slave port is asserting `dataavailable`, which means that the master port, not the slave port, has chosen to end the sequence of transfers.

6.2.3. Slave Write Transfer with Flow Control

Slave write transfers with flow control can use either of the signals `readyfordata` and `endofpacket`.

A slave port can assert `readyfordata` from low-to-high at any time. While `readyfordata is` asserted, a new transfer from a master port with flow control can begin on the next rising edge of `clk`. A slave port must deassert `readyfordata` from high-to-low at the end of a write transfer, such that the signal is immediately valid for transfers that might follow. If the slave port uses the `endofpacket` signal, it must assert `endofpacket` on the same clock edge that it captures `writedata`.

Figure 23 shows an example of slave write transfers to a slave port with flow control. In this example, assume that a master port with flow control initiates a sequence of transfers while the slave port has `readyfordata` asserted, and the master port continues initiating write transfers in immediate succession. At some point during the sequence, the slave port deasserts `readyfordata`, causing the system interconnect fabric to stop initiating transfers from the master port. Later, the slave port asserts `readyfordata` again, and the system interconnect fabric continues the sequence of slave write transfers. In this example, data is written to a constant slave address, which is common for I/O peripherals.

Figure 23. Slave Write Transfer with Flow Control
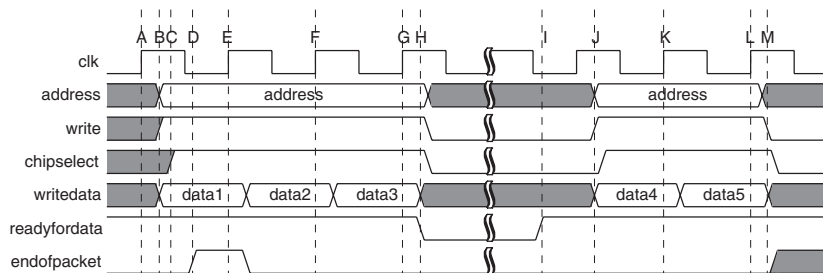
Notes to Figure 23:

(A) The transfer starts on the rising edge of `clk`.

(B) System interconnect fabric asserts `address`, `write` and `writedata`.

(C) System interconnect fabric decodes `address`, and asserts `chipselect`.

(D) Slave port asserts `endofpacket` before the last rising edge of `clk` for the current transfer. In this example, the slave deasserts `endofpacket` after one cycle, but this is not a requirement.

(E) The slave port captures `writedata` on the rising edge of `clk`. The system interconnect fabric captures `endofpacket`.

(F) For each cycle that `chipselect` and `write` remain asserted, the system interconnect fabric produces a valid `writedata`, which the slave port captures on the rising edge of `clk`. In this example, address is held constant, but this may not be the case for all peripheral designs.

(G) For each cycle that `chipselect` and `write` remain asserted, the system interconnect fabric produces a valid `writedata`, which the slave port captures on the rising edge of `clk`. In this example, address is held constant, but this may not be the case for all peripheral designs.

(H) The slave port deasserts `readyfordata`, forcing the system interconnect fabric to postpone any subsequent writes from the master port. The system interconnect fabric deasserts `address`, `write`,`chipselect` and `writedata` in response to `readyfordata`.

(I) Some time later, the slave port asserts `readyfordata` again.

(J) In response to `readyfordata`, the system interconnect fabric starts another transfer by reasserting `address`,`write`, `chipselect` and `writedata`, because the master port is still waiting to transfer data.

(K) The slave port captures `writedata` on risings edge of `clk` when `write` and `chipselect` are asserted.

(L) The slave port captures `writedata` on risings edge of `clk` when `write` and `chipselect` are asserted.

(M) The system interconnect fabric deasserts `write` and `chipselect`, ending the sequence of transfers.

Figure 23 shows the slave port asserting `endofpacket` during the sequence of write transfers. The interpretation is dependent on the design of the master and slave peripherals; `endofpacket` has no inherent relationship to `readyfordata` nor to how the master peripheral responds. The sequence of transfers finishes with the system interconnect fabric deasserting `chipselect and write` while `readyfordata` is still asserted, meaning that the master port, not the slave port, has chosen to end the sequence of transfers.

6.3. Master Transfers with Flow Control

Flow control does not change the timing or sequencing of signals on the master port. Flow control does not require any additional master signals. A master port can use flow control for either read or write transfers, or for both. A master port with flow control can optionally use the input signal endofpacket.

Flow control affects the master port's waitrequest signal, but it does not change how the master port responds to waitrequest. Flow control only adds to the conditions for which the system interconnect fabric asserts a master port's waitrequest. In a master-slave pair with flow control, after the master port initiates a transfer, if the slave port is not ready to accept the transfer, the system interconnect fabric asserts waitrequest. If the target slave port does not use flow control, the transfer proceeds the same as if neither port has flow control.

If endofpacket is used, it serves as a status flag for the current transfer. If both master and slave port use endofpacket, the signal is passed directly from the slave port to the master port. For write transfers, the master port captures endofpacket while asserting valid writedata; for read transfers, it captures endofpacket on the same clock as it captures valid readdata. The interpretation of the endofpacket signal is dependent on the slave logic. For example, endofpacket can be used as a packet delineator, to mark the boundary where packets start and end within a longer stream of data. Alternately, endofpacket can indicate that the master port should stop the current sequence of transfers.

Figure 24 shows an example of a master port performing read and write transfers using flow control. Both waitrequest and endofpacket are asserted at some point during the transfers.

Figure 24. Master Read and Write Transfers with Flow Control
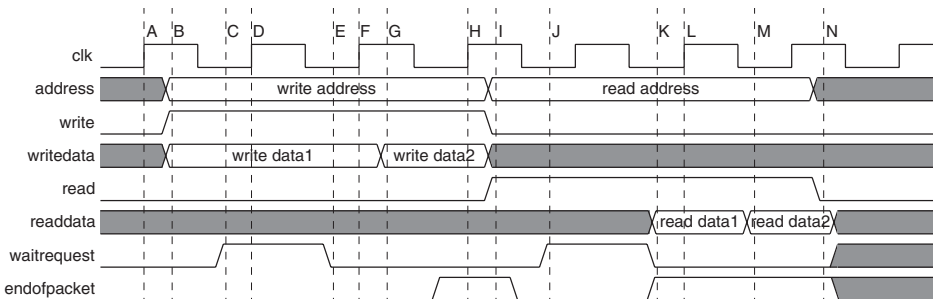
Notes to Figure 24:

(A)   First write transfer starts on the rising edge of `clk`.

(B)   Master port asserts `address`, `write` and valid `writedata`.

(C)   System interconnect fabric asserts `waitrequest` before the next rising edge of `clk`, forcing the master port to wait. The reason might be because the target slave port's flow control signals are not allowing a transfer. Whatever the reason, the master port obeys `waitrequest` because it has to.

(D)   `waitrequest` is asserted at the rising edge of `clk`, so the master port holds `address`, `write` and `writedata` constant.

(E)   System interconnect fabric deasserts `waitrequest`.

(F)   System interconnect fabric captures `writedata` on the rising edge of `clk`.

(G)   Master port keeps `address` and `write` asserted and asserts a new `writedata`. `address` does not necessarily have to remain constant, depending on the peripheral design.

(H)   If necessary, master port captures `endofpacket` on the last rising edge of `clk` of the current transfer. Master port terminates write transfer by deasserting `address`, `write` and `writedata`.

(I)   Master port immediately begins a read transfer during the next cycle by asserting `read` and a valid `address`.

(J)   System interconnect fabric asserts `waitrequest` to indicate that it cannot return valid data on the next rising edge of `clk`. The reason might be because the target slave port's flow control signals are not allowing a transfer. Whatever the reason, the master port obeys `waitrequest` because it has to.

(K)   Eventually the system interconnect fabric deasserts `waitrequest` and presents valid `readdata`. In this example the system interconnect fabric asserts `endofpacket`.

(L)   Master port captures `readdata` and `endofpacket` on the rising edge of `clk`.

(M)   Master port keeps `address` and `read` asserted for another read transfer; the system interconnect fabric presents valid `readdata`.

(N)   Master port deasserts `read` and `address`, and the transfer terminates.

7. Tristate Transfers

The Avalon-MM tristate property allows Avalon-based systems to connect directly to off-chip devices, such as memory chips or an external processor. Using the tristate property, it is possible to define an Avalon-MM port that matches the behavior of many standard memory or processor bus interfaces. If a subset of Avalon-MM signals can describe a chip's interface, then *de facto* that chip possesses an Avalon-MM tristate port. The system interconnect fabric can interface to such a chip using Avalon-MM tristate transfers.

7.1. Tristate Slave Transfers

Avalon-MM tristate slave ports allow the system interconnect fabric to interface to off-chip devices that share address and data bus lines on the physical printed circuit board (PCB). Avalon-MM tristate slave ports can be used to connect the system interconnect fabric to both synchronous and asynchronous memory chips, such as ROM, flash memory, SRAM, SSRAM, and ZBT RAM.

Tristate slave ports use the bidirectional signal `data`, rather than the separate, unidirectional signals `readdata` and `writedata`. The `data` signal is tristatable, which enables multiple tristate peripherals to connect to the data bus without causing signal contention.

The port must also use the `outputenable` signal. A port cannot use `data` in addition to `readdata` or `writedata`. All other Avalon-MM signals behave the same.

It is common for Avalon-MM slave ports to use negative polarity signals such as `read_n`, `chipselect_n`, and `outputenable_n` to be consistent with typical memory chip conventions.

7.1.1. Restrictions

The following restrictions apply to Avalon-MM slave tristate ports:

- Avalon-MM slave tristate ports cannot be pipelined with variable latency. Pipelined tristate ports with fixed latency are supported.
- Altera slave tristate ports cannot use flow control signals.
- Avalon-MM slave tristate ports cannot support bursts.

7.1.2. data Behavior

During write transfers the system interconnect fabric drives the `data` lines to present data to the slave device. During read transfers the slave device drives the `data` lines, and the system interconnect fabric captures the `data` signals.

When `outputenable` is asserted, the tristate slave port must drive its `data` lines. When system interconnect fabric deasserts `outputenable`, the Avalon-MM tristate slave port must tristate its `data` lines. If it does not, signal contention might occur, potentially damaging one or both of the connected devices. For details, refer to outputenable and read Behavior on page66.

7.1.3. address Behavior

For Avalon-MM tristate slave ports, the `address` signal represents a byte address. This is different behavior than non-tristate slave ports, which use word addresses. For tristate slave ports, the `address` signal can be shared among multiple off-chip devices, and these devices might have differing data widths. If the Avalon-MM tristate slave port data width is greater than one byte, then it is necessary to correctly map the address signals from the system interconnect fabric to the address lines on the slave device.

Table 5 specifies which Avalon-MM `address` line corresponds to A0 (the least-significant address line on the external device) for all possible data widths.

Table 5. Connecting External Device AO to Avalon-MM address	
Data Width	**AO connects to**
1-8	address[0]
9-16	address[1]
17-32	address[2]
33-64	address[3]
65-128	address[4]
129-256	address[5]
257-512	address[6]
513-1024	address[7]

For example, when connecting the system interconnect fabric to a 32-bit memory chip using an Avalon-MM tristate slave interface, the two least-significant bits of the Avalon-MM `address` signal do not connect to the address lines on the memory chip. Avalon-MM `address[2]` connects to the device's A0 pin, `address[3]` connects to the A1 pin, and so forth.

7.1.4. outputenable and read Behavior

The system interconnect fabric asserts the `outputenable` signal during read transfers only. When a port's `outputenable` is deasserted, the `data` lines may be active with signals for a write transfer, or with signals from some other peripheral that shares the `data` lines. Therefore, it is critical for the slave peripheral to tristate its `data` lines any time `outputenable` is deasserted.

The behavior of `outputenable` is different depending on whether the Avalon-MM tristate port is pipelined:

■ For Avalon-MM tristate ports without pipelining, the `outputenable` signal and the `read` signal are identical. Therefore, the Avalon-MM signal `read_n` can connect directly to both an external device's output enable pin (e.g. OEn) and read-enable pin (e.g. READn).

■ For Avalon-MM tristate ports with pipelining, the system interconnect fabric asserts `read` during the address phase only, and deasserts it through the data phase. Later, the switch fabric asserts `outputenable` before the final rising clock edge of the transfer, causing the peripheral device to drive its data pins. The system interconnect fabric deasserts `outputenable` when there are no pending read transfers.

7.1.5. write_n and writebyteenable Behavior

Some memory devices have a combined R/Wn pin (i.e., read when high, write when low). The Avalon-MM signal `write_n` behaves in this manner, and can be connected to a R/Wn pin. `write_n` is only asserted during write transfers, and remains deasserted (i.e., in read mode) at all other times. In this case, the Avalon-MM `outputenable_n` signal connects to the output enable pin (e.g. OEn) on the external device, and the Avalon-MM `write_n` signal connects to the R/Wn pin.

Some synchronous memory devices use individual write-enable signals for each byte lane (e.g., BWn1, BWn2, BWn3, and BWn4). The Avalon-MM port `writebyteenable` is the logical AND of the `write` and `byteenable` signals, and can be connected directly to such BWn pins.

Figure 25 shows an example of the connections between the system interconnect fabric and a typical asynchronous 32-bit 1Mbyte memory chip. This chip has an 18-bit address and four byte-enable lanes. Note that the lower two bits of the 20-bit Avalon-MM `address` signal specify a byte address, and therefore do not connect to the chip's address lines. In this example, the Avalon-MM `read_n` signal connects to the OEn pin on the memory, and the Avalon-MM `write_n` signal connects to the R/Wn pin on the memory chip.

Figure 25. Connection to Asynchronous Memory Chip
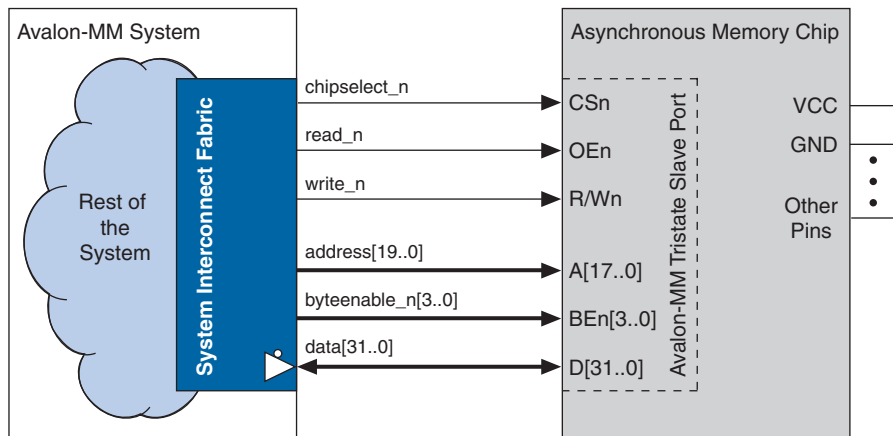
7.1.6. chipselect and Chipselect-Through-Read-Latency Property

For typical memory chips, the Avalon-MM `chipselect_n` signal connects directly to the chip select or chip enable pin (e.g., CSn or CEn) on the external device.

Some synchronous memory chips (which use pipelined transfers with fixed read latency) require a chip select signal to be asserted only during the address phase, while other chips require the chip select to be asserted until the entire transfer completes. The Avalon-MM tristate slave interface supports both cases, using the chipselect-through-read-latency property.

The port must declare which `chipselect` timing it will support:

■   When a port uses the chipselect-through-read-latency `property`, the system interconnect fabric asserts `chipselect` throughout both the address and data phases of the read transfer. In this case, `chipselect` mirrors the `outputenable` signal.
■   When a port does not use the chipselect-through-read-latency property, the system interconnect fabric asserts `chipselect` only during the address phase. In this case, `chipselect` mirrors the `read` signal.

7.1.7. Interfacing to Asynchronous Off-Chip Memory

When connecting system interconnect fabric signals directly to asynchronous off-chip memories with an Avalon-MM tristate slave port, the `clk` signal is not needed. Instead, pulses on the `chipselect`, `read` and/or `write` signals synchronize the transfer, typically using setup and hold time.

All output signals from the system interconnect fabric are glitch-free throughout the transfer.

7.1.8. Interfacing to Synchronous Off-Chip Memory

Avalon-MM tristate slave ports can write data to off-chip synchronous memory devices, such as SSRAM and ZBT RAM. For example, the hold time property can be used to keep `data` asserted several clock cycles after `write` is deasserted.

Continuous back-to-back pipelined read transfers and continuous back-to-back write transfers are supported. However, the system interconnect fabric waits for any pending pipelined read transfers to complete before initiating a new write transfer. This prevents possible signal contention on the `data` lines due to latent read data colliding with write data. As a result, the Avalon-MM tristate port might not achieve the maximum possible bandwidth when performing back-to-back read-write transfer sequences.

Figure 26 shows an example of the connections between the system interconnect fabric and a synchronous 32-bit, 1Mbyte memory chip. In this example, the Avalon-MM tristate slave port is pipelined to accommodate the synchronous memory. Therefore, the port uses separate `read_n` and `outputenable_n` signals. The chip in this example uses the `writebyteenable` signal for its four byte lanes. This chip has an 18-bit address. Note that the lower two bits of the 20-bit Avalon-MM `address` signal specify a byte address, and therefore do not connect to the chip's address lines.

Figure 26. Connection to Synchronous Memory Chip
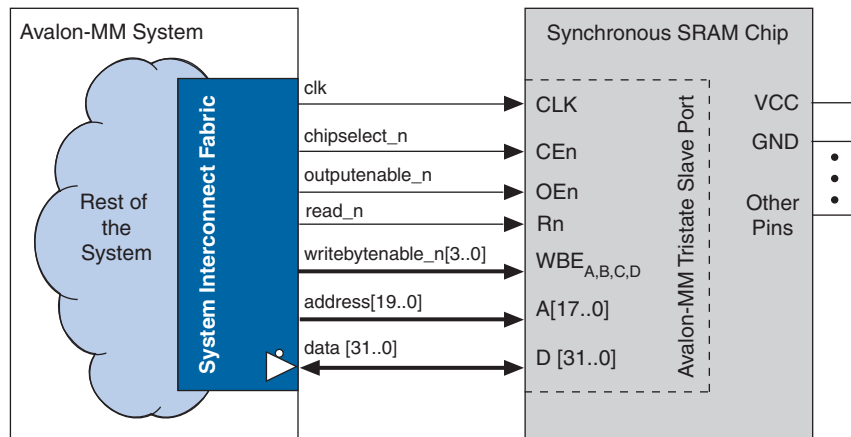
7.1.9. Examples

This section provides examples of various configurations of Avalon-MM tristate slave ports.

7.1.9.1. Tristate Slave Read Transfers to Asynchronous Memory
This example demonstrates an Avalon-MM tristate slave port configuration that is suitable for off-chip, asynchronous RAM or ROM chips. In this case, the tristate slave port typically does not use the clk signal, because the memory chip does not need it. However, the system interconnect fabric is always synchronous, and it toggles and captures signals only at integer multiples of the period of clk.

Figure 27 shows an Avalon-MM tristate slave read transfer. This port uses the following Avalon-MM properties:

- Fixed setup time of one cycle
- Fixed wait-states of one cycle
- No pipelining

The diagram shows the tristate behavior for one peripheral's data path. However, the data lines could be active at any time due to the transfer activity of a different peripheral sharing the data and address signals. write_n is shown here for reference; it is deasserted (i.e., read mode) throughout the transfer. This example uses active-low logic for read_n,

chipselect_n and write_n. This example shows the case for an asynchronous memory chip interface; clk is shown for timing reference only.

Figure 27. Tristate Slave Read Transfer with Setup Time and Wait-States
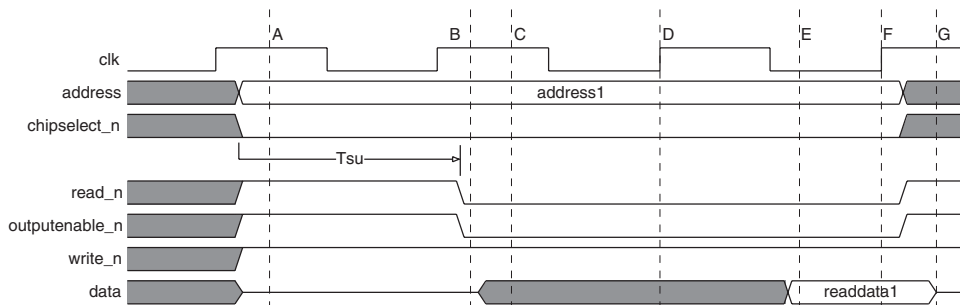
Notes to Figure 27:

(A) The system interconnect fabric drives address and asserts chipselect_n.
(B) After one cycle of setup delay, the system interconnect fabric asserts read_n and outputenable_n.
(C) The slave port drives data in response to outputenable_n.data might not be valid at this point. In this example, it is undefined.
(D) The system interconnect fabric keeps address asserted through one cycle of wait-state.
(E) The slave port drives valid data some time before the final rising clock edge of the transfer.
(F) The system interconnect fabric captures data at this rising edge of clk, and the transfer ends.
(G) The slave port tristates data in response to outputenable_n which is now deasserted.

7.1.9.2. Tristate Pipelined Slave Read Transfers

The pipelined Avalon-MM tristate slave read transfer is suitable for connecting to off-chip synchronous memory devices, such as SSRAM and ZBT SRAM.

Figure 28 shows a pipelined Avalon-MM tristate slave read transfer. This port uses the following Avalon-MM properties:

- Fixed pipeline latency of 2 clock cycles.
- Uses the chipselect-through-read-latency property
- Signals outputenable_n, chipselect_n, read_n and write_n are active low to reflect the conventions used by most external memory devices.

The diagram shows the tristate behavior for one peripheral's data path. However, the data lines could be active at any time due to the transfer activity of a different peripheral sharing the data and address signals. write_n is shown here for reference; it is deasserted (i.e., read mode) throughout the transfer.

Figure 28. Pipelined Tristate Slave Read Transfers
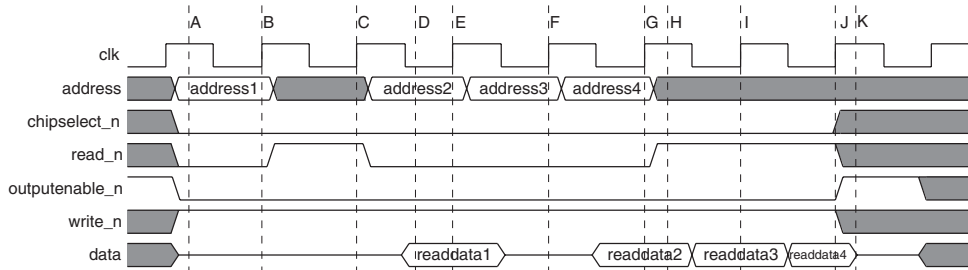
Notes to Figure 28:

(A) The system interconnect fabric asserts `chipselect_n`, `address`, and `read_n`, initiating a read transfer. At this time `outputenable_n` is also asserted, so the slave device is free to drive the `data` lines at any time. In this example, the device does not drive `data` immediately, and the lines remain tristated.

(B) The slave device captures `address` and `read_n` on this rising edge of `clk`. The data phase begins, and the slave device must produce valid data two clock cycles later.

(C) `read_n` is deasserted on this rising edge of `clk`, inserting an idle cycle. `chipselect_n` remains asserted because of the chipselect-through-read-latency property, i.e., `chipselect` must remain asserted until all pending read transfers have completed.

(D) The slave device drives valid `data` (readdata1) at some point before the final rising clock edge of the data phase.

(E) The system interconnect fabric captures readdata1 at this rising edge of `clk`. The system interconnect fabric asserts `chipselect_n`, `address`, and `read_n`, initiating transfer 2.

(F) The system interconnect fabric asserts `chipselect_n`, `address`, and `read_n` at this rising edge of `clk`, initiating transfer 3. The data lines are undefined because of the previous idle cycle. Because `outputenable_n` is asserted, the slave device could be driving the data lines. In this example, the device does not drive `data`, and the lines are tristated.

(G) The system interconnect fabric captures readdata2 at the rising edge of `clk`. The system interconnect fabric asserts `chipselect_n`, `address`, and `read_n` at this rising edge of `clk`, initiating transfer 4.

(H) The system interconnect fabric deasserts `read_n` ending the sequence of read transfers. `chipselect` remains asserted until all pending read transfers have completed.

(I) The system interconnect fabric captures readdata3 at this rising edge of `clk`.

(J) The system interconnect fabric captures readdata4 at this rising edge of `clk`.

(K) There are no more pending transfers, and the system interconnect fabric deasserts `chipselect` and `outputenable_n`, which forces the slave device to tristate its `data` lines.

7.1.9.3. Tristate Slave Write Transfers to Asynchronous Memory

Figure 29 shows an Avalon-MM tristate slave write transfer using setup and hold time. This port uses the following Avalon-MM properties:

■ Setup time of one clock cycle
■ Zero wait-states
■ Hold time of one clock cycle

`outputenable_n` is deasserted throughout the write transfer, and the peripheral must never drive the `data` lines throughout the write transfer. `clk` is shown for timing reference only.

Figure 29. Tristate Slave Write Transfer
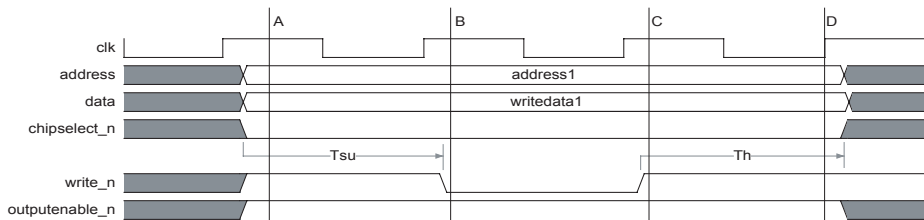
Notes to Figure 29:

(A) System interconnect fabric drives `address`, valid `data`, and asserts `chipselect_n`.
(B) After one cycle of setup delay, the system interconnect fabric asserts `write_n` for one cycle (i.e., no wait-states).
(C) System interconnect fabric deasserts `write_n`, but keeps `address` and `data` asserted for one cycle of hold time.
(D) The write transfer completes on this rising edge of `clk`.

7.2. Tristate Master Transfers

Avalon-MM tristate master ports allow the system interconnect fabric to interface to off-chip master peripherals with bidirectional data ports, such as the data bus on an external processor. Tristate master ports use the bidirectional signal `data`, rather than the separate, unidirectional signals `readdata` and `writedata`.

A port cannot use `data` in addition to `readdata` or `writedata`. All other Avalon-MM master signals behave the same. Unlike Avalon-MM tristate slave ports, Avalon-MM master ports can not share the `data` or `address` lines on the PCB with other tristate master ports.

During write transfers the master port drives the `data` lines to present data to the system interconnect fabric. During read transfers the system interconnect fabric drives the `data` lines, and the master port captures the `data` signals.

7.2.1. Restrictions

The following restrictions apply to Avalon-MM master tristate ports:

■ Avalon-MM master tristate ports cannot be pipelined.
■ Altera master tristate ports cannot use flow control signals.
■ Avalon-MM master tristate ports cannot support bursts.

7.2.2. Example

Figure 30 demonstrates a tristate master port performing write and read transfers.

Figure 30. Tristate Master Port – Write and Read Transfers
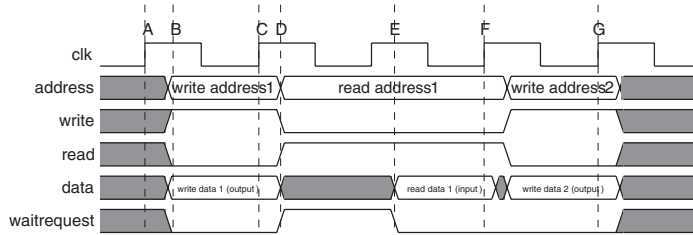
Notes to Figure 30:

(A) The master port initiates a write transfer on this rising edge of `clk`.
(B) The master port asserts `address` and `write`. For the write transfers, the master port drives the `data` lines.
(C) The system interconnect fabric captures write data on this edge of `clk`. The master port initiates a new read transfer in this cycle, asserting `address` and `read`.
(D) The system interconnect fabric asserts `waitrequest`. In response, the master port holds all signals constant through the cycle.
(E) Later the system interconnect fabric drives valid read data on the `data` lines and deasserts `waitrequest`.
(F) The master port captures `data` on this edge of `clk`. The master port initiates a new write transfer in this cycle.
(G) System interconnect fabric captures `data` on this edge of `clk`, ending the write transfer.

8. Burst Transfers

The Avalon-MM interface includes a burst transfer property. A burst executes multiple transfers as a unit, rather than treat every unit of data as an independent transfer. Bursts maximize the throughput for slave ports that achieve the greatest efficiency when handling multiple units of data from one master port at a time.

A burst guarantees that a master port is granted uninterrupted access to a target slave port for the duration of the burst. Once a burst begins between a master-slave pair, the system interconnect fabric does not allow any other master port to access the slave port until the burst completes.

An Avalon-MM master or slave port supports bursts by including the signal `burstcount`. The following characteristics describe `burstcount` for master and slave ports:

■ The `burstcount` signal must be between 2 and 32 bits wide.

- At the start of a burst, `burstcount` presents an encoded value indicating how many sequential transfers are in the current burst.
- The minimum `burstcount` value is one.
- A transfer with `burstcount` of one is equivalent to a single, non-burst transfer.
- For width N of `burstcount`, the maximum burst length is 2^{N-1}. In this case, the most-significant bit of `burstcount` is one, and all other bits are zero.

Avalon-MM bursts do not guarantee that a master or slave port will sustain one transfer per cycle during the burst. Bursts guarantee that arbitration between the master-slave pair is locked throughout a burst; the burst can take an unspecified amount of time, depending on the peripheral logic associated with the master and slave ports.

8.1. Restrictions

The following restrictions apply to ports that support bursts:

- To support master read bursts, a master port must also support pipelined transfers.
  As a result, the master port cannot also use the tristate property, which is disallowed for pipelined master ports.

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

  - Variable wait-states, i.e., it must include the `waitrequest` signal.
  As a result, the port cannot also use setup and hold time, which is disallowed for ports that use variable wait-states.

  - Pipelined transfers with variable latency, i.e., it must include the `readdatavalid` signal.
  As a result, the slave port cannot also use the tristate property, which is disallowed for pipelined ports with variable latency.

8.2. Master Burst

For an Avalon-MM master port, `burstcount` is an output signal. In addition to `burstcount`, burst transfers affect the behavior of the signals `address`, `read`, `readdata`, `readdatavalid`, `write`, `writedata` and `byteenable`.

At the start of a burst, a master port asserts a valid `address` and a burst length value on `burstcount`. The master port presents only one address value for each burst; the addresses for all transfers in the burst are inferred automatically by the system interconnect fabric.

When a master port starts a burst with an address of *A* and a `burstcount` value of *B*, it is committing to *B* consecutive transfers starting at address *A*. The burst does not complete until the master port transfers *B* units of data. A master port cannot abort the burst or give a new address without first exhausting remaining transfers in the current burst.

8.2.1. Master Write Bursts

The start of a write burst is similar to the start of a fundamental master write transfer. The master port asserts `address`, `writedata`, `write`, and `byteenable` (if present) in addition to `burstcount`. If the system interconnect fabric is not ready to continue, it asserts `waitrequest` before the next rising edge of `clk`. Eventually, the system interconnect fabric deasserts `waitrequest`, and captures `address` and `burstcount` on the next rising edge of `clk`. The system interconnect fabric also captures the first unit of `writedata` on this edge of `clk`. The master port must assert constant values on `address`, `byteenable`, and `burstcount` throughout the write burst.

The `address` and `burstcount` signals define the behavior of the rest of the burst. The following rules apply when a master port starts a transfer with burstcount greater than one:

- If the master port specifies `burstcount` of *N*, then the master port must assert `write` and present new `writedata` on *N* rising edges of `clk` to complete the burst. Arbitration between the master-slave pair is locked until the master port completes the burst.
- The master port can delay a transfer by deasserting `write` on a rising edge of `clk`, which prevents the system interconnect fabric from capturing `writedata` for the current cycle.
- The system interconnect fabric can delay a transfer by asserting `waitrequest`, which forces the master port to hold `writedata` and `write` constant through an additional cycle.
- The master port must assert all `byteenable` lines throughout the burst.

Figure 31 demonstrates an example of a master write burst of length 4. In this example, the system interconnect fabric asserts `waitrequest` two times when it cannot capture `writedata`, which delays the burst. The master port also deasserts `write` when it cannot produce a new `writedata` value, which also delays the burst.

Figure 31. Master Write Burst
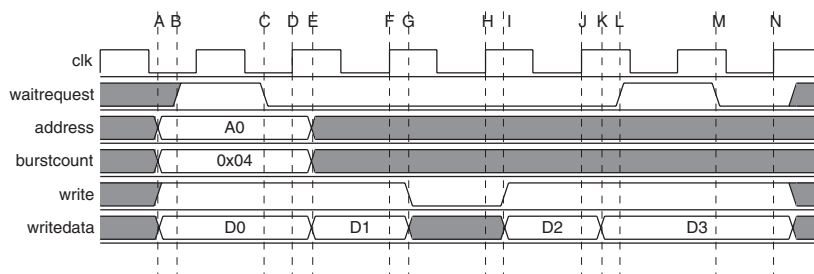
Notes to Figure 31:

(A) Master port asserts `address`, `burstcount`, `write`, and the first unit of `writedata`. In this example, the `burstcount` value is 4.

(B) System interconnect fabric asserts `waitrequest`, indicating that it is not ready to proceed with the burst. In response, the master port holds all outputs constant.

(C) System interconnect fabric deasserts `waitrequest`.

(D) System interconnect fabric captures `address`, `burstcount`, `write`, and the first unit of `writedata` (D0) at the rising edge of `clk`.

(E) Master port deasserts `address` and `burstcount`, which are ignored through the remainder of the burst. Master port presents next unit of `writedata` (D1).

(F) System interconnect fabric captures next unit of `writedata` (D1) at the rising edge of `clk`.

(G) Master port deasserts `write`, indicating that it does not have valid `writedata` for this clock cycle.

(H) The `write` signal is deasserted, so system interconnect fabric does not capture `writedata` on this edge of `clk`.

(I) Master port presents valid `writedata` (D2) and asserts `write` again.

(J) System interconnect captures `writedata` (D2) on this rising edge of `clk`.

(K) Master port presents last unit of `writedata` (D3).

(L) System interconnect fabric asserts `waitrequest`, causing the master port to hold all outputs constant through one clock cycle.

(M) System interconnect fabric deasserts `waitrequest`.

(N) System interconnect captures last unit of `writedata` (D3) on this rising edge of `clk`. The master write burst ends.

8.2.2. Master Read Bursts

Master read bursts are similar to master pipelined read transfers with latency. A master read burst has distinct address and data phases, and uses the `readdatavalid` signal to indicate when the master port must capture `readdata`. The difference is that a single burst address phase corresponds to multiple data phases.

The start of a master read burst is similar to the start of a pipelined master read transfer. The master port asserts `address` and `read` in addition to `burstcount`. If the system interconnect fabric is not ready to continue, it asserts `waitrequest` before the next rising edge of `clk`. Eventually, the

system interconnect fabric deasserts `waitrequest`, and captures `address` and `burstcount` on the next rising edge of `clk`. This is the end of the address phase. Multiple data phases follow.

The following rules apply when a master port starts a read burst with `burstcount` greater than one:

■   If the master port specifies `burstcount` of *N*, then the system interconnect fabric is guaranteed to assert `readdatavalid` on *N* rising edges of `clk` to complete the burst. Arbitration between the master-slave pair is locked until the system interconnect fabric returns all data for the burst.

■   The master port must capture `readdata` whenever the system interconnect fabric asserts `readdatavalid`. Each value of `readdata` is valid for only this one clock cycle.

■   The master port must assert all `byteenable` lines throughout the burst.

Figure 32 demonstrates a master read burst of length 4.

Figure 32. Master Read Burst
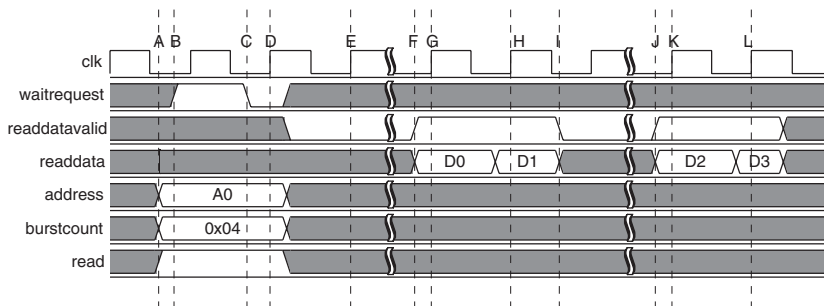
Notes to Figure 32:

(A) Master port asserts `address`, `burstcount` and `read`. In this example, the `burstcount` value is 4.

(B) System interconnect fabric asserts `waitrequest`, indicating that it is not ready to proceed with the burst. In response, the master port holds all outputs constant.

(C) System interconnect fabric deasserts `waitrequest`.

(D) System interconnect fabric captures `address` and `burstcount` at the rising edge of `clk`. The master port could begin a new transfer or burst on this rising edge of `clk` (which is not shown in this example).

(E) This is the earliest clock edge at which the system interconnect fabric could return valid `readdata`. In this example, the system interconnect fabric is not asserting `readdatavalid`, so the master port does not capture `readdata`.

(F) Some later time, the system interconnect fabric presents valid `readdata`, and asserts `readdatavalid`.

(G) Master port captures first unit of `readdata` (D0) on this rising edge of

(H) Master port captures next unit of `readdata` (D1) on this rising edge of `clk`.

(I) System interconnect fabric does not have valid `readdata`, and so it deasserts `readdatavalid`. The system interconnect fabric can keep `readdatavalid` deasserted for an arbitrary number of clock cycles.

(J) Some time later, the system interconnect fabric presents valid `readdata`, and asserts `readdatavalid` again.

(K) Master port captures next unit of `readdata` (D2) on this rising edge of `clk`.

(L) Master port captures last unit of `readdata` (D3) on this rising edge of `clk`. The master read burst ends.

8.3. Slave Bursts

For an Avalon-MM slave port, `burstcount` is an input signal. In addition to `burstcount`, bursts affect the behavior of the `address`, `read`, `readdata`, `readdatavalid`, `write`, `writedata` and `byteenable` signals. A slave port can also use the input signal `beginbursttransfer`, which the system interconnect fabric asserts for the first cycle of each burst.

At the start of a burst, the system interconnect fabric asserts a valid `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 port handles the *B*th unit of data.

The slave port captures `address` only once for each burst. The burst starts at this address, and the peripheral logic infers the address for all remaining transfers in the burst. The inferred addresses depend on whether the slave port uses native address alignment or dynamic bus sizing:

■ For native address alignment, the address remains constant. For example, a burst write with an `address` of 0x1000 and a `burstcount` value of 0x0A transfers 10 units of data to the constant address 0x1000.

■ For dynamic bus sizing, the slave address increments by one for each unit of data. For example, a burst write with an `address` of 0x1000 and a `burstcount` value of 0x04 transfers 4 units of data to slave addresses 0x1000, 0x1001, 0x1002 and 0x1003.

For further details, refer to "Address Alignment" on page 83.

8.3.1. Slave Write Bursts

The start of a slave write burst is similar to the start of a fundamental slave write transfer. The system interconnect fabric asserts `chipselect`, `address`, `byteenable`, `writedata`, and `write`, in addition to `burstcount`. If the slave port is not ready to continue with the transfer, it asserts `waitrequest` before the next rising edge of `clk`. Eventually, the slave port deasserts `waitrequest`, and captures `address` and `burstcount` on the next rising edge of `clk`. The slave port also captures the first unit of `writedata` on this edge of `clk`. This is the only time that the slave port can capture valid `burstcount` and `address` values.

The following rules apply when a slave write burst begins with `burstcount` greater than one:

■ If the system interconnect fabric specifies `burstcount` of *N*, then the slave port 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 port must only capture writedata when `write` is asserted. For the second unit of data or later, the system interconnect fabric can deassert `write` at any rising edge of `clk` to indicate that it is not presenting valid `writedata`. This does not terminate the burst; it only delays the burst until the system interconnect fabric asserts `write` again.

■ The `chipselect` signal mirrors `write`. If/when the system interconnect fabric deasserts `write`, it also deasserts `chipselect`.

- The slave port can delay a transfer by asserting `waitrequest` at a rising edge of `clk`, which forces the system interconnect fabric to hold `writedata`, `write`, and `byteenable` constant through an additional cycle.
- The system interconnect fabric asserts all `byteenable` lines throughout the burst.

Figure 33 demonstrates a slave write burst of length 4. In this example, the slave port asserts `waitrequest` two times when it cannot capture `writedata`, which delays the burst. The system interconnect fabric also deasserts `write` when it cannot produce a new `writedata` value, which also delays the burst.

Figure 33. Slave Write Burst
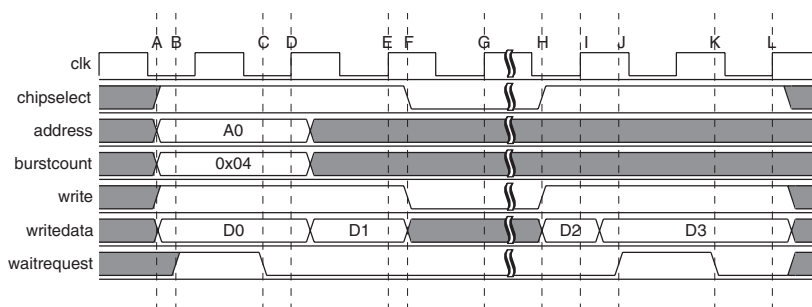
Notes to Figure 33:
(A) System interconnect fabric asserts `chipselect`, `address`, `burstcount`, `write`, and the first unit of `writedata`. In this example, the `burstcount` value is 4.
(B) The slave port asserts `waitrequest`, indicating that it is not ready to proceed with the burst. In response, the system interconnect fabric holds all outputs constant.
(C) Slave port deasserts `waitrequest`.
(D) Slave port captures `address`, `burstcount`, `write`, and the first unit of `writedata` (D0) at the rising edge of `clk`. This is the only time the slave port captures `address` and
(E) Slave port captures the next unit of `writedata` (D1) at the rising edge of `clk`.
(F) System interconnect fabric deasserts `write`, indicating that it does not have valid `writedata` for this clock cycle.
(G) Slave port does not capture `writedata` on this clock edge, because `write` is deasserted.
(H) Some time later, the system interconnect fabric asserts `write` and `writedata` again.
(I) Slave port captures next unit of `writedata` (D2) at the rising edge of `clk`.
(J) The slave port asserts `waitrequest`. In response, the system interconnect fabric holds all outputs constant though another clock cycle.
(K) Slave port deasserts `waitrequest`.
(L) Slave port captures last unit of `writedata` (D3) on this rising edge of `clk`. The slave write burst ends.

8.3.2. Slave Read Bursts

Slave read bursts are similar to slave pipelined read transfers with variable latency. A read burst has distinct address and data phases, and the slave port uses the readdatavalid signal to indicate when it is presenting valid readdata. The difference is that a single burst address phase corresponds to multiple data phases.

At the start of a slave read burst the system interconnect fabric asserts chipselect, address, and read in addition to burstcount. If the slave port is not ready to continue, it asserts waitrequest before the next rising edge of clk. Eventually, the slave port deasserts waitrequest, and captures address and burstcount on the next rising edge of clk. This is the end of the data phase. Multiple data phases follow.

The following rules apply when a slave read burst begins with burstcount greater than one:

■ If the system interconnect fabric specifies burstcount of *N*, then the slave port must produce *N* successive units of readdata to complete the burst. Arbitration between the master-slave pair is locked until the burst completes.

■ The slave port presents each unit of data by asserting valid readdata and asserting readdatavalid for one rising edge of clk. Deasserting readdatavalid does not terminate the burst; it only delays the burst until the slave port asserts readdatavalid again.

■ The system interconnect fabric asserts all byteenable lines throughout the burst.

Figure 34 demonstrates a slave read burst.

Figure 34. Slave Read Burst
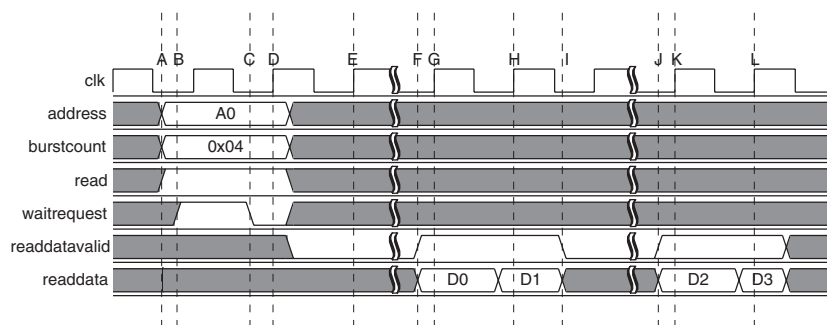
Notes to Figure 34:

(A) System interconnect fabric asserts `address`, `burstcount` and `read`. In this example, the `burstcount` value is 4.

(B) In this example, the slave port asserts `waitrequest`, indicating that it is not ready to proceed with the burst. In response, the system interconnect fabric holds all outputs constant though another clock cycle.

(C) Slave port deasserts `waitrequest`.

(D) Slave port captures `address` and `burstcount` at this rising edge of `clk`. The system interconnect fabric could begin a new transfer on this rising edge of `clk` (which is not shown in this example).

(E) This is the earliest clock edge at which the slave port could return valid `readdata`. In this example, the slave port is not asserting `readdatavalid`, so the system interconnect fabric does not capture `readdata` on this edge of `clk`.

(F) Some later time, the slave port presents valid `readdata`, and asserts `readdatavalid`.

(G) System interconnect fabric captures the first unit of `readdata` (D0) on this rising edge of `clk`.

(H) System interconnect fabric captures next unit of `readdata` (D1) on this rising edge of `clk`.

(I) Slave port does not have valid `readdata`, and so it deasserts `readdatavalid`. The slave port can keep `readdatavalid` deasserted for an arbitrary number of clock cycles.

(J) Some time later, the slave port presents valid `readdata`, and asserts `readdatavalid` again.

(K) System interconnect fabric captures next unit of `readdata` (D2) on this rising edge of `clk`.

(L) System interconnect fabric captures last unit of `readdata` (D3) on this rising edge of `clk`. The slave read burst ends.

9. Non-Transfer Related Signals

The Avalon-MM interface provides control signals with system-level functionality, such as interrupt requests and reset request. These signals are not directly related to individual data transfers.

9.1. Interrupt Request Signals

The Avalon-MM interrupt request signals allow a slave port to assert an interrupt request (IRQ), indicating that it needs to be serviced by a master port. The system interconnect fabric propagates IRQ signals between slave and master ports in a system.

9.1.1. Slave Interrupt Signal: irq

A slave port can include the `irq` output signal that acts as a flag indicating the peripheral logic needs to be serviced by a master port. The slave port can assert `irq` at any time; the timing of the irq signal has no relationship to any transfer. The peripheral logic must assert `irq` continuously until a master port explicitly resets the interrupt request.

9.1.2. Master Interrupt Signals: irq and irqnumber

A master port can include the signals `irq` and `irqnumber`, which let the master port detect and respond to the IRQ status of slave ports in the system. The Avalon-MM interface supports two methods to calculate the IRQ of highest priority: software priority calculation and hardware priority calculation.

9.1.2.1. Software Priority
A master port including a 32-bit `irq` signal defines itself to use software IRQ priority calculation. In this case, the master port does not include `irqnumber`. In the software priority configuration, the system interconnect fabric passes IRQs from up to 32 slaves directly to the master port, without making any assumptions about IRQ priority. Zero to 32 bits of `irq` might be asserted at any given time, indicating the IRQ status of the connected slave ports. In the event that multiple bits are asserted simultaneously, the master logic (presumably under software control) determines which IRQ has highest priority, and responds appropriately. Unused bits of `irq` are permanently disabled.

9.1.2.2. Hardware Priority
A master port including a 1-bit `irq` signal and the `irqnumber` signal defines itself to use hardware IRQ priority calculation. The system interconnect fabric asserts `irq` to the master port, signifying that one or more slave ports have generated an IRQ. The switch fabric simultaneously asserts the 6-bit `irqnumber` signal, indicating the encoded value of the pending IRQ with highest priority.

Using hardware priority, the master port can detect up to 64 slave IRQ signals. The system interconnect fabric (i.e. hardware logic) identifies the IRQ of highest priority and passes only that IRQ number to the master port on `irqnumber`. Lower `irqnumber` values indicate higher priority, with zero being the highest priority. When a higher-priority IRQ is pending, IRQs of lesser priority are undetectable by the master.

9.2. Reset Control Signals

The Avalon-MM interface provides signals that let the system interconnect fabric reset peripherals, and let peripherals reset the system.

9.2.1. reset Signal

Avalon-MM master and slave ports can use the reset input signal. Whenever the system interconnect fabric asserts reset, the peripheral logic must reset itself to a defined initial state.

The system interconnect fabric can assert reset at any time, regardless of whether a transfer is in progress. The reset pulse is guaranteed to be greater than the period of clk.

9.2.2. resetrequest Signal

Avalon-MM master and slave ports can use the resetrequest signal to reset the entire Avalon-MM system. resetrequest is useful for functions like watchdog timers, which —if not serviced within a guaranteed amount of time— can reset the entire system. Asserting resetrequest causes the system interconnect fabric to assert reset on other peripherals in the system.

10. Address Alignment

For systems in which master and slave data widths differ, the system needs to manage address alignment issues. This situation is not specific to Avalon-MM systems. The Avalon-MM interface abstracts data width differences, so that any master port can communicate with any slave port, regardless of the respective data widths.

In this section, *native address boundaries* refer to word addresses determined by the width of master data. For example, for a master port with 8-bit data width, the native address boundaries fall on addresses 0x01, 0x02, 0x03, 0x04, etc.; for a master port with 32-bit data width, the native address boundaries fall on addresses 0x00, 0x04, 0x08, 0x0C, etc.

If all master and slave ports in a system have the same data widths, then all units of slave data are aligned on native address boundaries in the master address space. However, if master and slave port data widths differ, there are two possible address alignments. The Avalon-MM address alignment property defines how slave data is aligned in a master port's address space.

Each Avalon-MM slave port declares its address alignment property to be one of the following:

- Native address alignment
- Dynamic bus sizing

The address alignment property defines what services the system interconnect fabric must provide to properly transport data between master and slave ports. In general, memory peripherals, such as an

SDRAM controller, use dynamic bus sizing. Peripherals use native address alignment if the Avalon-MM slave port is an interface to a register file that provides access to internal peripheral logic, such as a serial I/O peripheral.

The address alignment property affects only master ports; it defines where units of slave data appear in a master port's address space. Address alignment has no affect on the behavior of a slave port. For both master and slave ports, address alignment does not affect the signals used or the signal sequencing during transfers.

10.1.Native Address Alignment

When a master port addresses a slave port with the native address alignment property, all slave data are aligned on native master address boundaries.

When a master port reads from a narrower slave port, the slave data bits map to the lower bits of the master data, and the upper master data bits are padded with zero. During write transfers, the upper bits are ignored. For example, if a 16-bit master port reads an 8-bit slave port, the `readdata` signal is of the form 0x00*XX*, where *XX* represents valid data.

A master port cannot access a slave port with a wider data width that uses native address alignment.

Table 6 shows how master addresses correspond to slave addresses using native address alignment. In Table 6, BASE refers to the slave port's base address in the master address space.

Table 6. Native Address Alignment Master-to-Slave Address Mapping

Master Address					Corresponds to Slave Address
128-Bit Master Data	**64-Bit Master Data**	**32-Bit Master Data**	**16-Bit Master Data**	**8-Bit Master Data**	
BASE + 0x00	BASE + 0x00	BASE + 0x00	BASE + 0x00	BASE + 0x00	0
BASE + 0x10	BASE + 0x08	BASE + 0x04	BASE + 0x02	BASE + 0x01	1
BASE + 0x20	BASE + 0x10	BASE + 0x08	BASE + 0x04	BASE + 0x02	2
BASE + 0x30	BASE + 0x18	BASE + 0x0C	BASE + 0x06	BASE + 0x03	3
BASE + 0x40	BASE + 0x20	BASE + 0x10	BASE + 0x08	BASE + 0x04	4
BASE + 0x50	BASE + 0x28	BASE + 0x14	BASE + 0x0A	BASE + 0x05	5
...	...	...	...	...	...

10.2. Dynamic Bus Sizing

Dynamic bus sizing refers to a service provided by the system interconnect fabric that dynamically manages data during transfers between master-slave pairs of differing data widths. When a master port addresses a slave port with the dynamic bus sizing property, all slave data are aligned in contiguous bytes in the master address space.

If the master port is wider than the slave port, then the upper master data bytes correspond to the next location(s) in the slave address space. For example, when a 32-bit master port performs a read transfer from a 16-bit slave port with dynamic bus sizing, the system interconnect fabric executes two read transfers on the slave side, and presents 32-bits of slave data back to the master port.

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

Slave ports using dynamic bus sizing must have a data width of 8, 16, 32, 64, 128, 256, 512 or 1024. Table 7 shows how slave data of various widths is aligned within a 32-bit master. In Table 7, `OFFSET[N]` refers to an offset into the slave address space.

	32-Bit Master Data	
Master Address	**When Accessing a 16-Bit Slave Port**	**When Accessing a 64-Bit Slave Port**
`0x00`	`OFFSET[1]`$_{15..0}$`:OFFSET[0]`$_{15..0}$	`OFFSET[0]`$_{31..0}$
`0x04`	`OFFSET[3]`$_{15..0}$`:OFFSET[2]`$_{15..0}$	`OFFSET[0]`$_{63..32}$
`0x08`	`OFFSET[5]`$_{15..0}$`:OFFSET[4]`$_{15..0}$	`OFFSET[1]`$_{31..0}$
`0x0C`	`OFFSET[7]`$_{15..0}$`:OFFSET[6]`$_{15..0}$	`OFFSET[1]`$_{63..32}$
`...`	`...`	`...`

Table 7. Dynamic Bus Sizing Master-to-Slave Address Mapping

Table 8. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
May 2007, version 3.3	● Added Figure 19 which explains use of FIFO to store pending read transactions.	Clarified pending read transactions.
November 2006, version 3.2	**New Features**: ● The maximum data width increased to 1024 bits. The maximum width of the `byteenable` signal also increased to 128 bits to accommodate wider data. ● `byteenable` signal can now be asserted during read transfers. See sections "Slave Signal Details" on page 25, and "Master Signal Details" on page 40. For example, in the case of a 32-bit port, the valid `byteenable` combinations are: 0001, 0010, 0100, 1000, 0011, 1100, 1111. The following combinations are not valid: 0000, 0101, 0110, 0111, 1001, 1010, 1011, 1101, 1110. **Fundamental Master Read Transfers. Clarification and Corrections:** ● Added restrictions to the behavior of the `byteenable` signal during master read and write bursts. A master port must assert all `byteenable` lines during burst transfers. See "Master Burst" on page 73. ● Added restrictions to the behavior of the `byteenable` signal during slave read and write bursts. The system interconnect fabric guarantees that all `byteenable` lines are asserted during slave burst transfers. See "Slave Bursts" on page 77." ● Added restrictions to the behavior of the `address`, `burstcount`, and `byteenable` signal during master bursts. The master port must hold these signals constant throughout the burst. See "Master Burst" on page 73. ● Added restrictions to the usage of the byteenable signal. When more than one byte lane is asserted, all asserted lanes must be adjacent. The number of adjacent lines must be a power of two, and the specified bytes must be aligned on an address boundary for the size of the data. See "byteenable and writebyteenable" on page 27 and "byteenable" on page 42."	—

Table 8. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2006, version 3.2	**Nomenclature Changes**: ● Changed the name of the document from "Avalon Interface Specification" to "Avalon Memory-Mapped Interface Specification." ● Renamed "Avalon interface" to "Avalon Memory-Mapped interface" or "Avalon-MM," to accommodate the existence of the new Avalon Streaming Interface. For details, refer to the *Avalon Streaming Interface Specification*. ● Renamed "Avalon switch fabric" to "system interconnect fabric."	
May 2005, version 3.1	**New Features**: (1) Burst transfer support (2) Master tristate support (3) `writebyteenable` signal (4) Data width up to 128 bits Clarification and Corrections: (1) Clarified significance of least-significant master address bits: Master addresses are always byte addresses. (2) Clarified behavior for different combinations of `chipselect`, `read`, and `write` signals on slave ports. Nomenclature Changes: (1) Renamed prior "streaming" transfer property to "flow control" (2) Renamed prior "peripheral-controlled waitstates" to "variable waitstates."	
September 2004, version 3.0	Corrected Avalon Tristate Slave Port Signals table and changed title to Avalon Interface Specification Reference Manual.	
July 2003	Corrected timing diagrams.	
May 2003, version 2.1	Minor edits and additions.	
January 2003, version 2.0	Revised the "Avalon Read Transfer with Latency" and "Avalon Interface to Off-Chip Devices" sections	
July 2002, version 1.2	Minor edits and additions. Replaced Excalibur logo on cover with Altera logo – version 1.2.	
April 2002, version 1.1	Updated PDF – version 1.1	
January 2002, version 1.0	Initial PDF – version 1.0	