Avalon Bus Specification

Reference Manual

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com

Document Version: 2.3
Document Date: July 2003

I.S. EN ISO 9001

MNL-AVABUSREF-2.3

About this Manual

This manual provides comprehensive information about the Altera® Avalon™ Bus.

Table 1 shows the reference manual revision history.

Table 1. Reference Manual Revision History

Date	Description
July 2003	Corrected timing diagrams.
May 2003	Minor edits and additions.
January 2003	Revised the “Avalon Read Transfer with Latency” and “Avalon Interface to Off-Chip Devices” sections.
July 2002	Minor edits and additions. Replaced Excalibur logo on cover with Altera logo - version 1.2
April 2002	Updated PDF - version 1.1
January 2002	Initial PDF - version 1.0

How to Find Information

- The Adobe Acrobat Find feature allows you to search the contents of a PDF file. Click the binoculars toolbar icon to open the Find dialog box.
- Bookmarks serve as an additional table of contents.
- Thumbnail icons, which provide miniature previews of each page, provide a link to the pages.
- Numerous links, shown in green text, allow you to jump to related information.

How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at http://www.altera.com.

For technical support on this product, go to http://www.altera.com/mysupport. For additional information about Altera products, consult the sources shown in Table 2.

Table 2. How to Contact Altera

Information Type	USA & Canada	All Other Locations
Technical support	http://www.altera.com/mysupport/	http://www.altera.com/mysupport/
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	(408) 544-7000 *(1)* (7:00 a.m. to 5:00 p.m. Pacific Time)
Product literature	http://www.altera.com	http://www.altera.com
Altera literature services	lit_req@altera.com *(1)*	lit_req@altera.com *(1)*
Non-technical customer service	(800) 767-3753	(408) 544-7000 (7:30 a.m. to 5:30 p.m. Pacific Time)
FTP site	ftp.altera.com	ftp.altera.com

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

Typographic Conventions

This manual uses the typographic conventions shown in Table 3.

Table 3. Conventions

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: $\mathbf{f_{MAX}}$, **\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."
`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., `reset_n`. 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 AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c.,...	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.
↵	The angled arrow indicates you should press the Enter key.
👣	The feet direct you to more information on a particular topic.

Notes:

Table of Contents

Avalon Bus Specification

General Description

The Avalon bus is a simple bus architecture designed for connecting on-chip processors and peripherals together into a system–on–a–programmable chip (SOPC). The Avalon bus is an interface that specifies the port connections between master and slave components, and specifies the timing by which these components communicate.

The principal design goals of the Avalon bus were:

- *Simplicity* – Provide an easy-to-understand protocol with a short learning curve.
- *Optimized resource utilization for bus logic* – Conserve logic elements (LEs) inside the Programmable Logic Device (PLD).
- *Synchronous operation* – Integrate well with other user logic that coexists on the same PLD, while avoiding complex timing analysis issues.

Basic Avalon bus transactions transfer a single byte, half-word, or word (8, 16, or 32 bits) between a master and slave peripheral. After a transfer completes, the bus is immediately available on the next clock for another transaction, either between the same master-slave pair, or between unrelated masters and slaves. The Avalon bus also supports advanced features, such as latency-aware peripherals, streaming peripherals and multiple bus masters. These advanced transfer modes allow multiple units of data to be transferred between peripherals during a single bus transaction.

The Avalon bus supports multiple bus masters. This multi-master architecture provides great flexibility in the construction of SOPC systems, and is amenable to high bandwidth peripherals. For example, a master peripheral may perform Direct Memory Access (DMA) transfers, without requiring a processor in the data path to transfer data from the peripheral to memory.

Avalon masters and slaves interact with each other based on a technique called slave-side arbitration. Slave-side arbitration determines which master gains access to a slave, in the event that multiple masters attempt to access the same slave at the same time. Slave-side arbitration offers two benefits:

1. The details of arbitration are encapsulated inside the Avalon bus. Therefore, the master and slave interfaces are consistent, regardless of the number of masters and slaves on the bus. Each bus master interfaces to the Avalon bus as if it were the only master on the bus.

2. Multiple masters can perform bus transactions simultaneously, as long as they do not access the same slave during the same bus cycle.

Avalon has been designed to accommodate the system–on–a – programmable chip (SOPC) environment. The Avalon bus is an active, on-chip bus architecture, which consists of logic and routing resources inside a PLD. Some principles of the Avalon architecture are:

1. The interface to peripherals is synchronous to the Avalon clock. Therefore, no complex, asynchronous handshaking/acknowledge schemes are necessary. The performance of the Avalon bus (and the overall system) can be measured using standard, synchronous timing analysis techniques.

2. All signals are active LOW or HIGH, which facilitates immediate turn-around of the bus. Multiplexers (not tri-state buffers) inside the Avalon bus determine which signals drive which peripheral. Peripherals are never required to tri-state their outputs, even when the peripheral is deselected.

3. The address, data and control signals use separate, dedicated ports, which simplifies the design of peripherals. A peripheral does not need to decode address and data bus cycles, and does not need to disable its outputs when it is not selected.

Avalon also includes a number of features and conventions to support automatic generation of systems, busses, and peripherals by the SOPC Builder software.

Features Overview

Up to 4GBytes Address Space—Memory and peripherals may be mapped anywhere within the 32-bit address space.

Synchronous Interface—All Avalon signals are synchronized to the Avalon bus clock. This simplifies the relevant timing behavior of the Avalon bus, and facilitates integration with high-speed peripherals.

Separate Address, Data and Control Lines—Separate, dedicated address and data paths provide the easiest interface to on-chip user logic. Peripherals do not need to decode data and address bus cycles.

B*uilt-in Address Decoding*—The Avalon bus automatically generates Chip Select signals for all peripherals, greatly simplifying the design of Avalon peripherals.

Multiple Master Bus Architecture—Multiple master peripherals can reside on the Avalon bus. The Avalon bus automatically generates arbitration logic.

Wizard-based Configuration—Easy-to-use graphical wizards guide the user through Avalon bus configuration (adding peripherals, specifying master/slave relationships, defining the memory map). The Avalon bus architecture is generated automatically based on user input from the wizard interface.

Dynamic Bus Sizing—The Avalon bus automatically handles the details of transferring data between peripherals with mismatched data widths, allowing peripherals of various widths to interface easily.

Terms and Concepts

Many of the terms and concepts relating to SOPC design are entirely new, or substantially different from traditional, off-chip bus architectures. The designer needs to understand this context in order to understand the Avalon bus specification. The following terms and concepts create a conceptual framework upon which the Avalon bus specification is built. They are used throughout this document.

Bus Cycle

A bus cycle is a basic unit of one bus clock period, which is defined from rising-edge to rising-edge of the Avalon master clock. Bus signal timing is referenced to the bus cycle clock.

Bus Transfer

An Avalon bus transfer is a read or write operation of a data object, which may take one or more bus cycles. The transfer sizes supported by the Avalon bus include byte (8-bit), half-word (16-bit) and word (32-bit).

Streaming Transfer

Streaming transfers create an open channel between a streaming master and streaming slave to perform successive data transfers. This channel allows data to flow between the master-slave pair as data becomes available. The master does not have to continuously access status registers in the slave peripheral to determine whether the slave can send or receive data. Streaming transfers maximize throughput between a master-slave pair, while avoiding data overflow or underflow on the slave peripheral. This is especially useful for DMA transfers.

Read Transfer with Latency

Read transfer with latency increase the bandwidth efficiency to synchronous peripherals that require several cycles of latency for the first access, but can return data every bus cycle thereafter. Latent transfers allow a master to issue a read request, move on to an unrelated task, and receive the data later. The unrelated task can be issuing another read transfer, even though data from the previous transfer hasn't yet returned. This is beneficial for instruction fetch operations and DMA transfers, in which access to sequential addresses is the norm. In these cases, the CPU or the DMA master may prefetch expected data, thereby keeping the synchronous memory active and reducing the average access latency.

SOPC Builder Software & Generation of the Avalon Bus

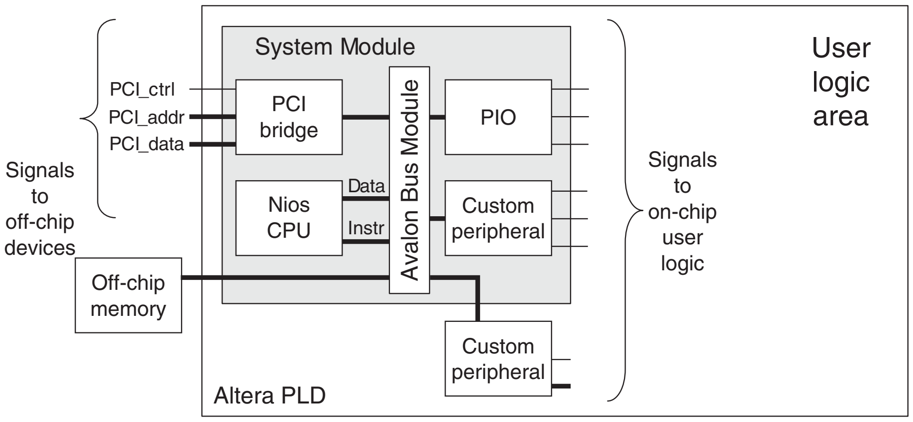
SOPC Builder is a system generation and integration tool developed by Altera. SOPC Builder generates the system module, which is the on-chip circuitry that comprises the Avalon bus, master peripherals, and slave peripherals. SOPC Builder has a graphical user interface for adding master and slave peripherals to the system module, configuring the peripherals, and then configuring the Avalon bus to connect peripherals together. With this information, SOPC Builder automatically creates and connects HDL modules, that implements all or part of the user's PLD design.

See the *SOPC Builder Data Sheet f*or more information about the system module.

System Module

Consider the structure of a user-defined system on a programmable chip, part of which is automatically generated by SOPC Builder. The entire system is implemented on an Altera PLD, as shown inFigure 1.

Figure 1. System Module Integrated with User Logic into an Altera PLD



For purposes of this document, system module refers to the portion of the design that was automatically generated by SOPC Builder. The system module contains at least one Avalon master peripheral and the entire Avalon bus module. The system module usually contains several Avalon slave peripherals, such as UARTs, timers or PIOs. The logic external to the system module may contain custom Avalon peripherals and other custom logic unrelated to the system module.

The system module must be connected to the designer's PLD design. The ports on the system module will vary, depending on which peripherals are included in the system module and which settings were made in SOPC Builder. These ports may include direct connections to the Avalon bus, and user-defined ports to peripherals inside the system module.

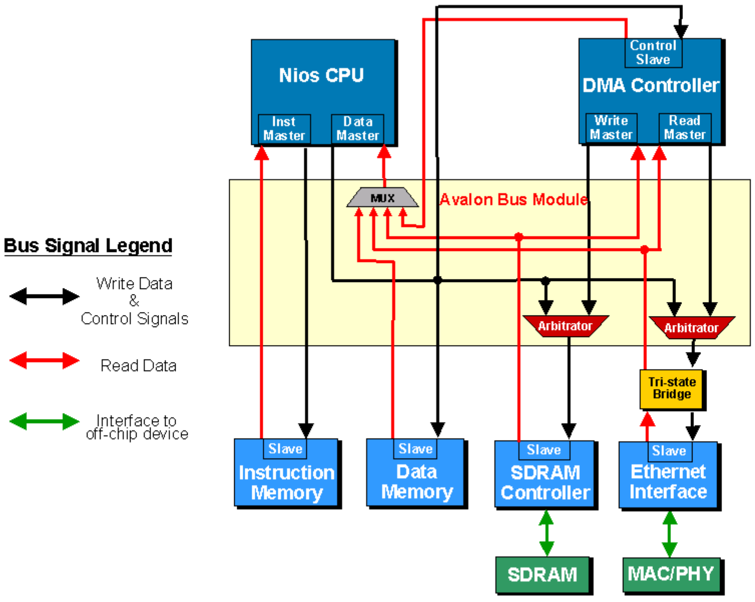
Avalon Bus Module

The Avalon bus module is the backbone of an system module. It is the main path of communication between peripherals components in an SOPC design. The Avalon bus module is the sum of all control, data and address signals and arbitration logic that connect together the peripheral components making up the system module. The Avalon bus module implements a configurable bus architecture, which changes to fit the interconnection needs of the designer's peripherals.

The Avalon bus module is generated automatically by SOPC Builder, so that the system designer is spared the task of connecting the bus and peripherals together. The Avalon bus module is very rarely used as a discrete unit, because the system designer will almost always use SOPC Builder to automate the integration of processors and other Avalon bus peripherals into a system module. The designer's view of the Avalon bus module usually is limited to the specific ports that relate to the connection of custom Avalon peripherals.

The Avalon bus module (an Avalon bus) is a unit of active logic that takes the place of passive, metal bus lines on a physical PCB. (See Figure 2). In this context, the ports of the Avalon bus module could be thought of as the pin connections for all peripheral devices connected to a passive bus. This manual defines only the ports, logical behavior and signal sequencing that comprise the interface to the Avalon bus module. It does not specify any electrical or physical characteristics of a physical bus.

Figure 2. Avalon Bus Module Block Diagram - An Example System



The Avalon bus module provides the following services to Avalon peripherals connected to the bus:

- *Data-Path Multiplexing*—Multiplexers in the Avalon bus module transfer data from the selected slave peripheral to the appropriate master peripheral.

- *Address Decoding*—Address decoding logic produces chip-select signals for each peripheral. This simplifies peripheral design, because individual peripherals do not need to decode the address lines to generate chip-select signals.

- *Wait-State Generation*—Wait-state generation extends bus transfers by one or more bus cycles, for the benefit of peripherals with special synchronization needs. Wait states can be generated to stall a master peripheral in cases when the target slave peripheral cannot respond in a single clock cycle. Wait states can also be generated in cases when read-enable and write-enable signals have setup or hold time requirements.

- *Dynamic Bus Sizing*—Dynamic bus sizing hides the details of interfacing narrow peripherals to a wider Avalon bus, or vice versa. For example, in the case of a 32-bit master read transfer from a 16-bit memory, dynamic bus sizing would automatically execute two slave read transfers to fetch 32 bits of data from the 16-bit memory device. This reduces the logic and/or software complexity in the master peripheral, because the master does not have to worry about the physical nature of the slave peripheral.

- *Interrupt-Priority Assignment*—When one or more slave peripherals generate interrupts, the Avalon bus module passes the (prioritized) interrupts to appropriate master peripherals, along with the appropriate interrupt request (IRQ) number.

- *Latent Transfer Capabilities*—The logic required to perform transfers with latency between master-slave pairs is contained inside the Avalon bus module.

- *Streaming Read and Write Capabilities*—The logic required to allow streaming transfers between master-slave pairs is contained inside the Avalon bus module.

Avalon Peripherals

An Avalon peripheral on the Avalon bus is a logical device—either on-chip or off-chip—that performs some system-level task, and communicates with other system components through the Avalon bus. Peripherals are modular system components, and may be added or removed at design time, depending on the requirements of the system.

Avalon peripherals can be memories and processors, as well as traditional peripheral components, such as a UART, PIO, timer or bus bridge. Any user logic can also be an Avalon peripheral, as long as it provides address, data and control signal interfaces to the Avalon bus as described in this document. A peripheral connects to specific ports on the Avalon bus module allocated for that peripheral. The peripheral may also have user-defined ports in addition to the Avalon address, data and control signals. These signals connect to custom logic external to the system module.

The roles of Avalon peripherals are classified as either a master or slave. A master peripheral is a peripheral that can initiate bus transfers on the Avalon bus. A master peripheral has at least one master port ("Master Port" on page 17) which connects to the Avalon bus module. A master peripheral may also have a slave port ("Slave Port" on page 18), which allows the peripheral to receive bus transfers initiated by other master peripherals on the Avalon bus. A slave peripheral is a peripheral that only accepts bus transfers from the Avalon bus, and cannot initiate bus transfers. Slave peripherals, such as memory devices or UARTs, usually have only one slave port, which connects to the Avalon bus module.

In the SOPC environment, it is important to make the distinction between the following types of peripherals, which may be either Avalon bus masters or slaves.

Peripherals inside the System Module

If SOPC Builder finds a peripheral in a peripheral library, or if the designer specifies the location of a custom peripheral design file, then SOPC Builder automatically connects the peripheral to the Avalon bus module. Such a peripheral is referred to as a peripheral inside the system module, and is treated as a piece of the system module. The details of connecting the address, data and control ports to the Avalon bus module are hidden from the user. Any additional non-Avalon ports on the peripheral are presented to the outside world as ports on the system module. These ports may connect directly to physical device pins, or may connect to the ports of other on-chip modules.

Peripherals outside the System Module

An Avalon bus peripheral can also exist external to the system module. This peripheral is referred to as a peripheral outside the system module. A designer may chose to leave the module outside the system module for several reasons:

- The peripheral may exist physically outside the PLD.
- The peripheral may require some glue logic to connect it to the Avalon bus signals.
- The peripheral design may not be complete at the time the system module is generated.

In this case, the appropriate Avalon bus module signals are presented to the outside world (and to the specific peripheral) as ports on the system module.

Master Port

A master port is the collection of ports on a master peripheral used to initiate transfers on the Avalon bus. The master port connects directly to the Avalon bus module. In practice, a master peripheral may have one or more master ports, as well as a slave port. The interdependence of these master and slave ports is dependent on the peripheral design. However, individual bus transfers on these master or slave ports always conform to this document. Throughout this document, a master transfer refers to an Avalon bus transfer from the perspective of a single master port.

Slave Port

A slave port is the collection of ports on a peripheral to accept Avalon bus transfers from the master port on another Avalon peripheral. The slave port connects directly to the Avalon bus module. Master peripherals may also have a slave port, which allows the peripheral to accept transfers from other masters on the Avalon bus. Throughout this document, a slave transfer refers to an Avalon bus transfer from the perspective of a single slave port.

Master-Slave Pair

A master-slave pair is the combination of a master port and a slave port that are connected via the Avalon bus module. Structurally, these master and slave ports connect to their appropriate ports on the Avalon bus module. Effectively, the master port's control and data signals pass through the Avalon bus module, and interact with the slave port. Connections between master and slave ports (thus creating master-slave pairs) are specified in SOPC Builder.

PTF File & SOPC Builder Parameters & Switches

The configuration of the Avalon bus and peripherals can be specified using the wizard-based SOPC Builder graphical user interface (GUI). Through this GUI the user specifies various parameters and switches, which are then used to generate a system PTF file. The PTF file is a text file that fully defines:

- Parameters that define the structure and/or functionality of the Avalon bus module.
- Parameters for each peripheral that define its structure and/or functionality.
- The master/slave role of each peripheral.
- The ports (such as read enable, read data, write enable, write data) present on each peripheral.
- The arbitration mechanism for each slave port that can be accessed by multiple master ports.

The PTF file is then passed to an HDL generator that creates the actual register transfer level (RTL) description of the system module.

See the *SOPC Builder Data Sheet* and the *SOPC Builder PTF File Reference Manual* for additional information about the system PTF files.

Avalon Bus Transfers

The Avalon bus specification defines the signals and timing required to transfer data between a master port and a slave port via the Avalon bus module. The signals that comprise the interface between the Avalon bus module and the peripheral are different, depending on the type of transfer. Foremost, the interface is different for master transfers and slave transfers, giving rise to the distinct definitions of a slave port and a master port. Furthermore, the exact type and number of signals required will vary, based on assignments made in the system PTF file.

The Avalon bus specification offers a variety of options to tailor the bus signals and timing to the needs of different types of peripherals. Fundamental Avalon bus transfers move a single unit of data per bus transfer between a master-slave pair. The bus transfer can be extended with wait states to accommodate slow peripherals. Streaming transactions along with simultaneous multi-master capabilities accommodate high-bandwidth peripherals. Peripherals can also use a combination of transaction types. The sequencing of signals for all Avalon slave transfers are derived from the fundamental slave read transfer and fundamental slave write transfer. Likewise, the fundamental master read and master write transfers are the basis for all Avalon master transfers.

Master Interface versus Slave Interface

When discussing Avalon bus transfers, it is important to pay attention to which side of the bus is the focus: the master port interface or the slave port interface. The signals output from a master port on the Avalon bus module may be very different from the corresponding signals that are input into the slave port on the target peripheral.

The signal activity on the slave side is always the result of a master peripheral initiating a bus transfer, but the actual slave port input signals do not come directly from the master port. The Avalon bus module relays the signals from the master port, and custom-tailors the signals (e.g., inserts wait states; arbitrates between contending masters) to the needs of the slave peripheral.

For this reason, the discussion of Avalon bus transfers is separated into master transfer types and slave transfer types. Most designers will be interested only in slave transfers, because the custom peripherals they design (if any) will most likely be slave peripherals. In this case, the designer considers only the signaling between the Avalon bus module and the custom peripheral. The discussion of master transfers is only relevant in the event that a designer creates a master peripheral.

Avalon Bus Timing

The Avalon bus is a synchronous bus interface, clocked by a master Avalon bus clock. All bus transfers occur synchronous to the Avalon bus clock. All bus transfers initiate on a rising clock edge, and terminate after valid data is captured on (or before) a subsequent rising clock edge.

A synchronous bus interface does not necessarily mean that all Avalon bus signals are registered. Notably, the Avalon `chipselect` signal is combinatorial, based on the outputs of registers that are synchronous to the Avalon bus clock, `clk`. Therefore, peripherals must not be edge sensitive to Avalon signals, because Avalon signals may transition multiple times before they stabilize. As with any synchronous design, Avalon bus peripherals must function only in response to signals that are stable at the rising edge of `clk`, and output stable signals at the rising edge of `clk`.

It is possible to interface asynchronous peripherals such as off-chip, asynchronous memory to the Avalon bus module, but there are a few design considerations. Due to the synchronous operation of the Avalon bus module, Avalon signals toggle only at intervals equal to the period of the Avalon bus clock. Also, if an asynchronous peripheral's outputs are connected directly to the Avalon bus module, the designer must make sure that the output signals are stable before the rising edge of `clk`.

The Avalon bus specification makes no attempt to dictate how signals transition between clock edges. Toggling signals are triggered by the Avalon bus clock, and that signals must stabilize before the clock edge when they are captured. For this reason, the Avalon bus timing diagrams in this document are devoid of explicit timing information. The exact timing of signals toggling and stabilizing between clock edges will vary, depending upon the characteristics of the Altera PLD selected to implement the system. By the same token, there is no inherent maximum performance of the Avalon bus. After synthesis and place-and-route of the system module for a specific device, the designer must perform standard timing analysis on the system module to determine the maximum speed at which Avalon bus transfers can be performed.

Avalon Bus Signals

Because the Avalon bus is an on-chip bus architecture synthesized from HDL files, special attention must be given to the connections between the Avalon bus module and Avalon peripherals. The situation is very different from a passive, off-chip bus architecture in which all peripherals share access to a pre-defined and constant group of physical metal wires. In the case of the Avalon bus, SOPC Builder must know exactly what Avalon ports are present on each peripheral so that it can connect the peripherals to Avalon bus module. Furthermore, it must know the name of each port and the role of each port. The name and role for each port on an Avalon peripheral is declared in the system PTF file.

The Avalon bus specification does not mandate the existence of any port on an Avalon peripheral. It only defines the possible types of signals (such as address, data, clock) that can exist on a peripheral. Each port on a peripheral is assigned a valid Avalon signal type, which determines the port's role. A port may also be user-defined, in which case SOPC Builder does not connect the port to the Avalon bus module. Fundamentally, the Avalon signal types are classified as either slave port signals or master port signals. Therefore, the signal types used by a peripheral are determined first and foremost by the master/slave role of the port. Each master or slave port may have up to one of each signal type. The set of signal types used by an individual master or slave port is dependent on the design of the peripheral. For example, the design for an output-only PIO slave peripheral would define only ports for write transfers (the output direction), but no ports for read transfers. Such a peripheral also probably would have no use for an Interrupt Request (IRQ) output, even though an IRQ output is an allowed signal type for a slave port.

The Avalon bus specification does not dictate a naming convention for the ports on an Avalon peripheral. The role of each port is well defined, but the name of the port is defined by the peripheral design. The port may be named the same as its signal type, or it may be named differently to comply with a system-wide naming convention. The discussion of Avalon bus transfers in the following sections refers to Avalon signals as, for example, the `readdata` signal or the `irq` signal. The name of the signal type has been used here as the port name, but the actual names given to ports on peripherals in the System Module may be different.

Table 1 shows a partial list of the signal types available to an Avalon slave port as an example. The signal direction is from the perspective of the peripheral. For example, the clock signal `clk` (listed as an input) is an *input* to the slave peripheral, but it is an *output* from the Avalon bus module.

Table 1. Partial List of Avalon Slave Signals

Signal Type	Width	Direction	Required	Description
`clk`	1	in	no	Global clock signal for the system module and Avalon bus module. All bus transactions are synchronous to `clk`. Only asynchronous slave ports can omit `clk`.
`address`	1 - 32	in	no	Address lines from the Avalon bus module.
`read`	1	in	no	Read request signal to slave. Not required if the slave never outputs data to a master. If used, `readdata` must also be used.
`readdata`	1 – 32	out	no	Data lines to the Avalon bus module for read transfers. Not required if the slave never outputs data to a master. If used, `read` signal must also be used.
`write`	1	in	no	Write request signal to slave. Not required if the slave never receives data from a master. If used, `writedata` must also be used.
`writedata`	1 – 32	in	no	Data lines from the Avalon bus module for write transfers. Not required if the slave never receives data from a master. If used, `write` signal must also be used.
`irq`	1	out	no	Interrupt request. Slave asserts `irq` when it needs to be serviced by a master.

The signal types listed in Table 1 are active high. However, the Avalon bus also offers the negated version of each signal type. By appending "_n" to the signal type name (e.g., `irq_n`, `read_n`) in the PTF declaration, the corresponding port is declared active low. This is useful for many off-chip peripherals that use active-low logic.

The Avalon bus signals and their operation is the same, whether a peripheral is implemented inside the system module or outside the system module. In the inside case, SOPC Builder automatically connects the peripheral's master or slave port to the Avalon bus module. In the outside case, the designer must manually connect the master or slave port to the system module. In either case, the Avalon bus signals behave the same.

For additional System Builder and PTF file information see the *SOPC Builder Data Sheet*. and the *SOPC Builder PTF File Reference Manual*.

Simultaneous Multi-Master Avalon Bus Considerations

The Avalon bus accommodates multiple master ports connected to the Avalon bus module. However, no special signals external to the Avalon bus module are used to implement simultaneous multi-master Avalon bus functionality. Slave-side arbitration logic inside the Avalon bus module arbitrates conflicts when multiple master peripherals attempt to access the same slave peripheral at the same time. The arbitration scheme is entirely hidden from Avalon bus peripherals. Therefore, the protocol for Avalon bus transfers—as perceived by master and slave ports—is the same, whether arbitration is used or not.

In other words, slave ports are not aware that multiple masters have simultaneously requested a bus transfer. Likewise, a master peripheral that is forced to wait by the arbitration logic is not aware of the other victorious master. The master port simply sees its wait-request signal asserted, and knows that it must wait until the target slave is ready to proceed with the bus transfer. Hiding the details of arbitration inside the Avalon bus module greatly simplifies peripheral design, because any Avalon peripheral can be used both in single-master and multi-master architectures.

See *AN 184: Simultaneous Multi-Mastering with the Avalon Bus* for more information.

Avalon Slave Transfers

The following sections discuss bus transfers between a slave port and the Avalon bus. From an abstract, system-level viewpoint, master peripherals exchange data with slave peripherals. However, from the viewpoint of a slave peripheral, data is transferred between the peripheral's slave port and the Avalon bus module. In the following discussion of bus transfers with slave ports, it is assumed that a master peripheral somewhere on the Avalon bus has successfully initiated a transfer on the master side of the Avalon bus module. As a result, the Avalon bus module then initiates the transfer with the appropriate slave port. The interface between the Avalon bus module and the slave port is the exclusive focus of this section.

Avalon Signals for Slave Transfers

Table 2 below lists the signal types that interface a peripheral's slave port to the Avalon bus module. The signal direction is from the perspective of the slave port. Not all of the signal types listed in Table 2 will be present on all peripherals, depending on the peripheral design and the ports declared in the PTF file. Table 2 gives a brief description of which signals are required and under what circumstances

Table 2. Avalon Slave Port Signals (Part 1 of 2)

Signal Type	Width	Direction	Required	Description
`clk`	1	in	no	Global clock signal for the system module and Avalon bus module. All bus transactions are synchronous to `clk`. Only asynchronous slave ports can omit `clk`.
`reset`	1	in	no	Global reset signal. Implementation is peripheral-specific.
`chipselect`	1	in	yes	Chip select signal to the slave. The slave port should ignore all other Avalon signal inputs unless `chipselect` is asserted.
`address`	1 - 32	in	no	Address lines from the Avalon bus module.
`begintransfer`	1	in	no	Asserted during the first bus cycle of each new Avalon bus transfer. Usage is peripheral-specific.
`byteenable`	0, 2, 4	in	no	Byte-enable signals to enable specific byte lane(s) during transfers to memories of width greater than 8 bits. Implementation is peripheral-specific.
`read`	1	in	no	Read request signal to slave. Not required if the slave never outputs data to a master. If used, `readdata` must also be used.
`readdata`	1 – 32	out	no	Data lines to the Avalon bus module for read transfers. Not required if the slave never outputs data to a master. If used, `read` signal must also be used.
`write`	1	in	no	Write request signal to slave. Not required if the slave never receives data from a master. If used, `writedata` must also be used.
`writedata`	1 – 32	in	no	Data lines from the Avalon bus module for write transfers. Not required if the slave never receives data from a master. If used, `write` signal must also be used.

Table 2. Avalon Slave Port Signals (Part 2 of 2)

Signal Type	Width	Direction	Required	Description
`readdatavalid`	1	out	no	Used only by slaves with variable latency. Marks the rising clock edge when the slave asserts valid `readdata`.
`waitrequest`	1	out	no	Used to stall the Avalon bus module when slave port is not able to respond immediately.
`readyfordata`	1	out	no	Signal for streaming transfers. Indicates that the streaming slave can receive data.
`dataavailable`	1	out	no	Signal for streaming transfers. Indicates that the streaming slave has data available.
`endofpacket`	1	out	no	Signal for streaming transfers. May be used to indicate an "end of packet" condition to the master port. Implementation is peripheral-specific.
`irq`	1	out	no	Interrupt request. Slave asserts `irq` when it needs to be serviced by a master.
`resetrequest`	1	out	no	A reset signal allowing a peripheral to reset the entire system module.

In the following discussions of Avalon slave transfers, the `read`, `write` and `byteenable` signals are used in their active-low form, which is similar to the traditional convention of using active-low read enable, write enable and byte enable signals. Note the following:

- These signals appear in the form `read_n`, `write_n` and `byteenable_n`.
- Any port of an Avalon signal type may be used with active high or low polarity, based on the port's declaration in the PTF file.

Slave Read Transfers on the Avalon Bus

In the discussions of read transfers below, it is important to realize that under realistic circumstances, bus transfers are not isolated events. They typically happen in continuous succession. For example, a slave read transfer may immediately precede or follow an unrelated write transfer. During the read transfer, the target peripheral's `read_n` and `chipselect` signals are necessarily asserted, as shown in the timing diagrams. However, after the read transfer terminates, `chipselect` and `read_n` may remain asserted if another bus transfer with this slave port follows on the next bus cycle. The timing diagrams below show undefined values on the slave port signals before and after the read transfer. Fundamental slave read transfers have no latency.

Fundamental Slave Read Transfer

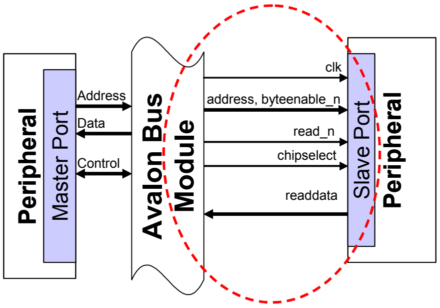
The fundamental slave read transfer is the basis for all Avalon slave read transfers. All other slave read transfer modes use a super set of the fundamental signals, and implement a variation of the fundamental slave read timing. The fundamental slave read transfer is initiated by the Avalon bus module, and transfers one unit of data, the full width of the peripheral's data port, from the slave port to the Avalon bus module. Fundamental slave read transfers have no latency.

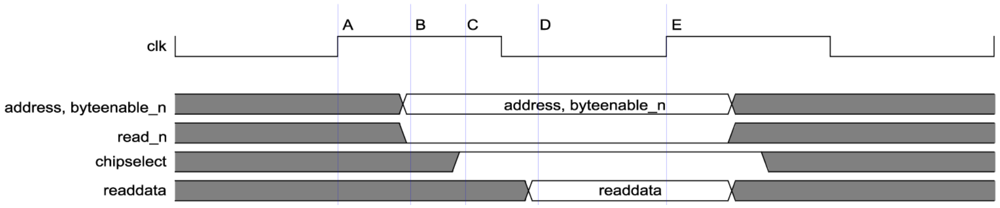
Example 1 shows an example of the fundamental read transfer. In the fundamental Avalon read transfer, the bus transfer starts on a rising clock edge, no wait states are incurred, and the read transfer completes on the next rising clock edge. For the transfer to complete in a single bus cycle, the target peripheral must immediately and asynchronously output the contents of the addressed location to the Avalon bus module.

On the first rising edge of `clk`, the Avalon bus passes the `address`, `byteenable_n`, and `read_n` signals to the target peripheral. The Avalon bus module decodes `address` internally, generates a chip select and drives the combinatorial `chipselect` signal to the slave port. Once `chipselect` is asserted, the slave port drives out its `readdata` as soon as it is available. Finally, the Avalon bus module captures the `readdata` on the next rising edge of the clock.

Example 1. Fundamental Slave Read Transfers

This Example Demonstrates	Relevant PTF Parameters
Read transfer from an asynchronous peripheral	
Zero wait states	Read_Wait_States = “0”
Zero setup	Setup_Time = “0”





Example 1 Time Reference Description

(A) First bus cycle starts on the rising edge of `clk`.
(B) Registered outputs `address` and `read_n` from Avalon bus to slave are valid
(C) Avalon bus decodes `address` & asserts valid `chipselect` to slave.
(D) Slave port returns valid data during the first bus cycle.
(E) Avalon bus captures `readdata` on the next rising edge of `clk`, and the read transfer ends here. The next bus cycle could be the start of another bus transfer.

This fundamental read transfer with zero wait states is appropriate only for truly asynchronous peripherals. The target peripheral must present data to the Avalon bus immediately when the peripheral is selected and/or the address changes. For the transfer to work properly, `readdata`’s output must be valid and stable by the next rising clock edge.

Synchronous peripherals that register the input or output ports cannot use the fundamental slave read transfer with zero wait states. Most on-chip peripherals will use a synchronous interface that requires at least one clock to capture data. This necessitates at least one wait state during the read transfer unless the peripheral is latency aware. See "Avalon Read Transfers with Latency" on page 54 for more information.

The byte enable lines `byteenable_n` may be connected to the peripheral's slave port. Interpretation of `byteenable_n` is peripheral dependent for slave read transfers. In the simplest case, the slave port ignores `byteenable_n`, and always drives all byte lanes whenever `read_n` is asserted. The Avalon bus module captures the full bit width of the `readdata` port every read transfer. Therefore, if an individual byte lane is not enabled during a read transfer, the value returned to the Avalon bus module is undefined, which may or may not affect the master that ultimately receives the data.

When `chipselect` is deasserted, all other input signals should be ignored. The slave port outputs may be driven or left undefined when the slave port is not selected. The `chipselect` signal driven to the target peripheral may be combinatorial, based on registered address values. Furthermore, a low-to-high edge on `chipselect` or a high-to-low edge on `read_n` cannot be used as a start read transfer trigger, because such an edge is not guaranteed.

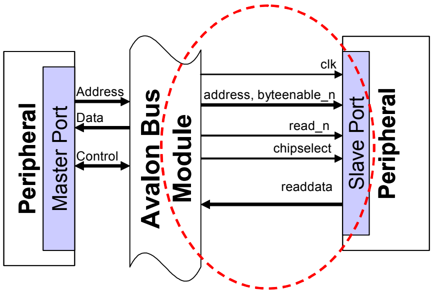
Slave Read Transfer with Fixed Wait States

The ports used for a slave read transfer with fixed wait states are identical to those used for a fundamental read transfer. The difference is in the timing of signals only. Slave read transfers with wait states are useful for peripherals that cannot present data within a single clock cycle. For example, with one fixed wait state specified, the Avalon bus module presents a valid address and control, but waits for one clock cycle before capturing the peripheral's data. Fixed wait states for a peripheral are declared in the PTF file. They are fixed because the Avalon bus module waits a fixed number of bus cycles every read transfer.

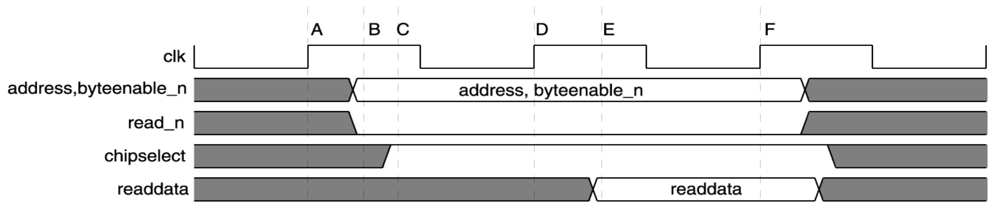
Example 2 shows an example slave read transfer with one wait state. The Avalon bus module presents `address`, `byteenable_n`, `read_n` and `chipselect` during the first bus cycle. Because of the wait state, the peripheral does not have to present `readdata` within the first bus cycle; the first bus cycle is the first (and only) wait state. The slave port may capture address and control signals at any time. On-chip, synchronous peripherals will probably capture address and control on the rising edge of `clk` at the start of the second bus cycle (the end of the wait state). During the second bus cycle, the target peripheral presents its `readdata` to the Avalon bus module. On the third and final rising clock edge, the Avalon bus module captures `readdata` from the slave port, and completes the transfer.

Example 2. Slave Read Transfer with One Fixed Wait State (Part 1 of 2)

This Example Demonstrates	***Relevant PTF Parameters***
Read transfer from a synchronous peripheral	
1 fixed wait state	Read_Wait_States = “1”
No setup time	Setup_Time = “0”



Example 2: Slave Read Transfer with One Fixed Wait State (Part 2 of 2)



Example 2 Time Reference Description

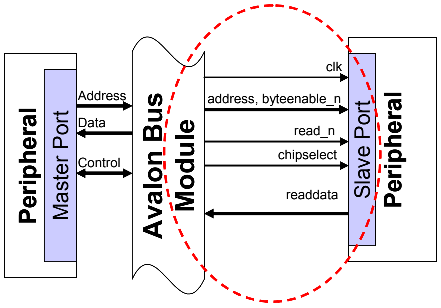
(A) First bus cycle starts on the rising edge of `clk`.
(B) Registered outputs `address` and `read_n` from Avalon bus to slave are valid
(C) Avalon bus decodes `address` & asserts `chipselect`.
(D) Rising edge of `clk` marks the end of the first and only wait-state bus cycle. If the slave port is synchronous, it probably captures `address`, `read_n` & `chipselect` on this rising edge of `clk`.
(E) Peripheral presents valid `readdata` during the second bus cycle.
(F) Avalon bus module captures `readdata` on the rising edge of `clk`, and the read transfer ends here. The next bus cycle could be the start of another bus transfer

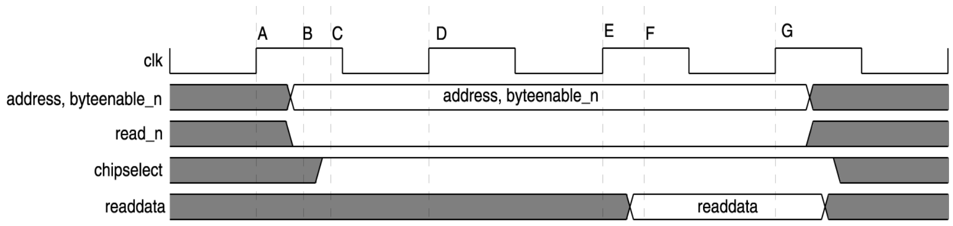
Read transfers with a single wait state are frequently used for synchronous, on-chip peripherals. Sound PLD design methodology dictates that the interface between modules should be synchronized with registers. Adding a wait state makes the transfer more amenable to PLD design, because the peripheral can capture synchronous signals `address`, `byteenable_n`, `read_n` and `chipselect` on the rising edge of `clk` after `chipselect` is asserted. The target peripheral then has at least one full bus cycle to present data back to the Avalon bus module. For higher bandwidth with pipelined read transactions, see "Avalon Read Transfers with Latency" on page 54.

Example 3 on shows a read transfer with multiple fixed wait states. This case is almost identical to Example 2 on page 29, except that the Avalon Bus now waits for more than one bus cycle before sampling the `readdata` from the slave peripheral.

Example 3. Slave Read Transfer with Multiple Fixed Wait States

This Example Demonstrates	*Relevant PTF Parameters*
Read transfer from a synchronous peripheral	
2 fixed wait states	Read_Wait_States = “2”
No setup time	Setup_Time = “0”





Example 3 Time Reference Description

(A) First bus cycle starts on the rising edge of `clk`.
(B) Registered outputs `address` and `read_n` from Avalon bus to slave are valid
(C) Avalon bus decodes `address` then asserts `chipselect`.
(D) Rising edge of `clk` marks the end of the first wait-state bus cycle. If the slave port is synchronous, it probably captures `address`, `read_n` & `chipselect` on this rising edge of `clk`.
(E) Rising edge of `clk` marks the end of the second (and last) wait state.
(F) Peripheral presents valid `readdata` sometime during the third cycle.
(G) Avalon bus module captures `readdata` on the rising edge of `clk`, and the read transfer ends here. The next bus cycle could be the start of another bus transfer.

Slave Read Transfer with Peripheral-Controlled Wait States

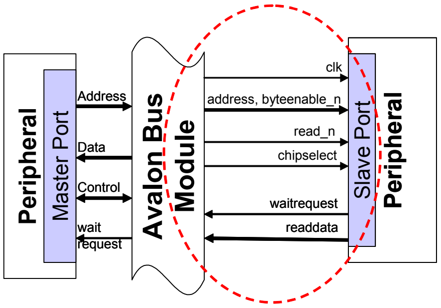
Peripheral-controlled wait states allow a target peripheral to stall the Avalon bus module for as many bus cycles as required to present data. Using this transfer mode, a peripheral can take a variable amount of time to present data to the Avalon bus module.

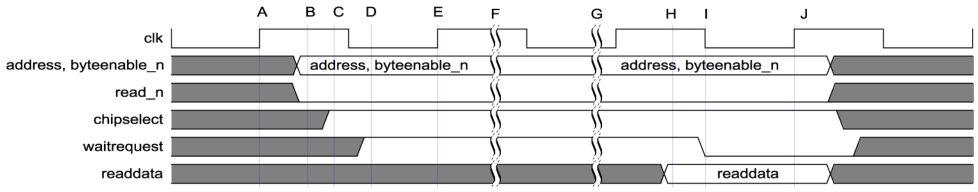
Example 4 on page 33 shows slave read transfer with peripheral-controlled wait states. The peripheral-controlled wait state mode uses the `waitrequest` signal, which is an output from the slave port. After `read_n` is asserted to the slave port, the slave port must return `waitrequest` within the first bus cycle if it wishes to extend the read transfer. When asserted, `waitrequest` stalls the Avalon bus module and prevents it from capturing `readdata`. The Avalon bus module will capture `readdata` on the next rising edge of `clk` after `waitrequest` is deasserted.

The Avalon bus module does not have a time-out feature to limit how long the slave port can stall. When the Avalon bus module is stalled, somewhere in the system module there is a master peripheral that is stalled as well, waiting for the requested data to come back from the addressed slave peripheral. A slave port could permanently “hang” the master port. Therefore, the peripheral designer must ensure that a slave peripheral does not assert `waitrequest` indefinitely.

Example 4. Slave Read Transfer with Peripheral-Controlled Wait States

This Example Demonstrates	*Relevant PTF Parameters*
Read transfer from synchronous peripheral	
More than one peripheral-controlled wait state	Read_Wait_States = "peripheral_controlled"
No setup	Setup_Time =”0”





Example 4 Time Reference Description

(A) First bus cycle starts on the rising edge of `clk`.
(B) Registered outputs `address` and `read_n` from Avalon bus to slave are valid
(C) Avalon bus decodes `address` then asserts `chipselect`.
(D) Slave port asserts `waitrequest` before the next rising edge of `clk`.
(E) Avalon bus module samples `waitrequest` at the rising edge of `clk`. `waitrequest` is asserted. so readdata is not captured on this clock edge.
(F-G) With `waitrequest` asserted throughout, an infinite number of bus cycles elapse.
H) Slave port presents valid `readdata`.
(I) Slave port deasserts `waitrequest`.
(J) Avalon bus module captures `readdata` on the next rising edge of `clk`, and the read transfer ends here. The next bus cycle could be the start of another bus transfer.

When peripheral-controlled wait states are specified, the following restrictions apply to other bus transfer modes. These restrictions apply only to transfers with this specific slave port, not to any other peripheral connected to the Avalon bus module.

If peripheral-controlled wait states are specified, setup and hold wait states cannot be used. In almost all cases, a peripheral that can generate the `waitrequest` signal will be on-chip and synchronous causing setup and hold time considerations unnecessary.

Slave Read Transfer with Setup Time

The Avalon bus module automatically accommodates setup time requirements for each slave port, based on declarations made in the PTF file. The master peripheral that initiates the read transfer does not need to consider the setup and hold requirements of each slave port. The ports used for a read transfer with setup time are identical to those used for a fundamental read transfer. The difference is in the timing of signals only.

Setup time is generally used for off-chip peripherals that require `address` and `chipselect` signals to be stable for a period of time before the read enable signal is asserted. A nonzero setup time of N means that, after `address`, `byteenable_n` and `chipselect` signals are presented to the slave port, there is a delay of N bus cycles before `read_n` is asserted. Note that `chipselect` is not affected by the setup time. If the peripheral requires a setup time for both `read_n` and `chipselect`, then the designer must manually add the appropriate logic (one AND gate) to the interface.

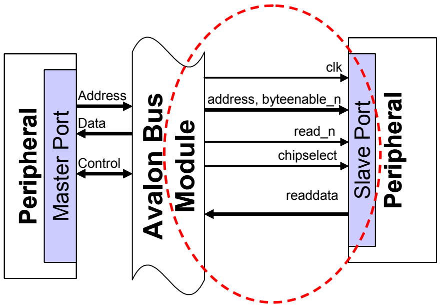
The total number of bus cycles to complete the bus transfer depends on setup and wait-state bus cycles. For example, a peripheral with `Setup_Time="2"` and `Read_Wait_States="3"` will take 6 bus cycles to complete the transfer:

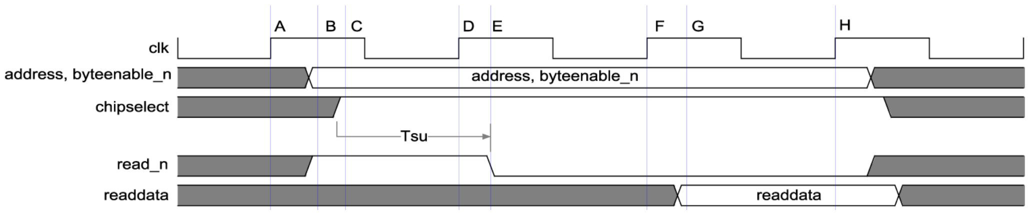
- 2 setup bus cycles plus
- 3 wait-state bus cycles plus
- 1 bus cycle to capture data

Example 5 shows a slave read transfers with one bus cycle of setup and one fixed wait state.

Example 5. Slave Read Transfer with Setup Time

This Example Demonstrates	*Relevant PTF Parameters*
Read transfer from synchronous peripheral	
1 bus cycle of setup time	Setup_Time = “1”
1 fixed wait state	Read_Wait_States = “1”





Example 5 Time Reference Description

(A) First bus cycle on the rising edge of `clk`.
(B) Registered output `address` and `byteenable_n` from the Avalon bus module are valid. `read_n` remains deasserted.
(C) Avalon bus module decodes `address` then asserts `chipselect`.
(D) Rising edge of `clk` defines the end of the setup–time bus cycle (Tsu), and the start of the wait-state bus cycle.
(E) Avalon bus module asserts `read_n`
(F) Rising edge of `clk` marks the end of the wait-state bus cycle.
(G) Peripheral presents valid `readdata`.
(H) Avalon bus module captures `readdata` at the rising edge of `clk`, and the read transfer ends here. The next bus cycle could be the start of another bus transfer.

When setup time is specified for a peripheral on the Avalon bus, the following restrictions apply to other bus transfer modes. These restrictions apply only to this slave port, not to other peripherals connected to the Avalon bus module.

If a peripheral is capable of both read and write bus transfers, 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 peripheral-controlled wait states.

Slave Write Transfers on the Avalon Bus

In the discussions of write transfers below, it is important to realize that under realistic circumstances, bus transfers are not isolated events. For example, a write transfer may immediately precede or follow an unrelated read transfer. During the write bus transfer, the target peripheral’s `chipselect` and `write_n` signals are necessarily asserted, as shown in the timing diagrams. However, after the write transfer terminates, `chipselect` and `write_n` may remain asserted if another transfer with this slave port follows on the next bus cycle. Therefore, the timing diagrams below show unknown values on the slave port signals before and after the write transfer.

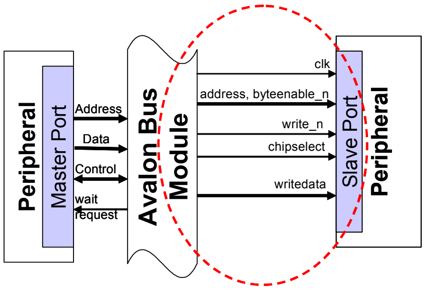
Fundamental Slave Write Transfer

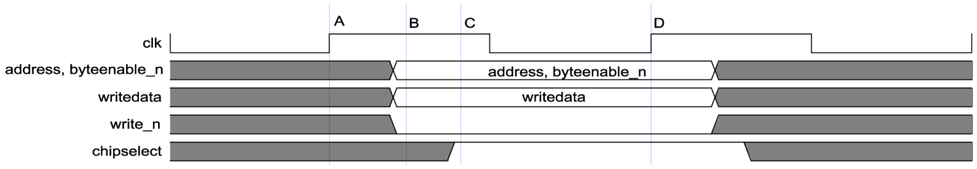
The fundamental slave write transfer is the basis for all Avalon write transfers. All other slave write transfer modes use a super set of the fundamental signals, and implements a variation of the fundamental timing. The fundamental slave write transfer is initiated by the Avalon bus module, and transfers one unit of data from the Avalon bus module to the slave port. Fundamental slave write transfers have no latency.

Example 6 shows the fundamental slave write transfer. There are zero wait states, and no setup-time or hold-time wait states. The Avalon bus module presents `address`, `writedata`, `byteenable_n`, and `write_n`, and then asserts `chipselect`. The slave port captures the address, data and control on the next rising clock edge, and the write transfer terminates immediately. The entire transfer takes only one bus cycle. The slave peripheral may then take additional clock cycles to actually process the write data after the transfer terminates. If the peripheral cannot sustain consecutive write transfers on every bus cycle, then additional design considerations are required to generate wait states.

Example 6. Fundamental Slave Write Transfer

This Example Demonstrates	*Relevant PTF Parameters*
A single write transfer to a synchronous peripheral	
No fixed wait state	Write_Wait_States = “0”
No setup time	Setup_Time = “0”
No hold time	Hold_Time = “0”





Example 6 Time Reference Description

(A) Write transfer starts on the rising edge of `clk`.
(B) Registered `writedata`, `address`, `byteenable_n` and `write_n` signals from the Avalon bus module are valid.
(C) Avalon bus module decodes `address` and asserts valid `chipselect` to slave.
(D) Avalon bus module captures `writedata`, `address`, `write_n`, byteenable and `chipselect` on the rising edge of `clk`, and the transfer terminates. Another read or write transfer may follow on the next bus cycle.

The fundamental write transfer is only appropriate for synchronous peripherals, which includes many on-chip peripherals, such as PIOs and timers for the Nios® processor. The timing for a fundamental write transfer is not appropriate for asynchronous peripherals, because all output signals including `write_n` and `chipselect` are all deasserted at the same time. This would cause a race condition in, for example, an off-chip asynchronous memory. For such a memory, the Avalon bus module provides several hold time options, which are discussed in subsequent sections.

The byte enable lines `byteenable_n` may be connected to the peripheral's slave port, and may be used to write a specific byte lane when `writedata` is wider than one byte wide. `byteenable_n` is a bus with one bit for every byte lane in `writedata`. `byteenable_n` is usually necessary for slave write transfers to off-chip, 16-bit or 32-bit memory devices that are word addressable. When writing a single byte of data, `address` specifies only an appropriate word or half-word address, while `byteenable_n` specifies exactly which byte(s) to write. Some example cases of `byteenable_n` are specified below in Table 3, assuming the slave port is a 32-bit external memory.

Table 3. Byte Enable Usage for 32-bit Slave

byteenable_n[3:0]	Write action
0000	Write full 32-bits
1100	Write lower 2 bytes
0011	Write upper 2 bytes
1110	Write byte 0 only
1011	Write byte 2 only

When `chipselect` is deasserted, all slave port input signals should be ignored. The slave port's outputs may be driven or left undefined when the slave port is not selected. Note that the `chipselect` signal from the Avalon bus module may be combinatorial, and therefore may glitch, based on transitions on the `address` port. Furthermore, a low-to-high edge on `chipselect` or a high-to-low edge on `write_n` cannot be used as a start write transfer trigger, because such an edge is not guaranteed to be clean. If this is not taken into consideration, the slave port will interpret erroneous write operations into unknown locations specified by an undefined `address`.

Slave Write Transfer with Fixed Wait States

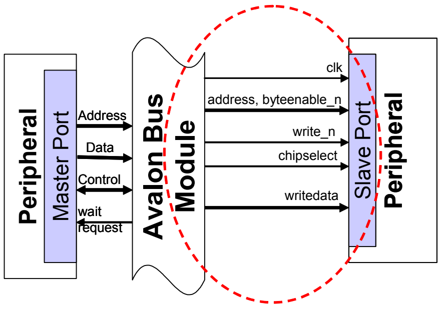
The ports used for a write transfer with fixed wait states are identical to those used for a fundamental write transfer. The only difference is in the timing of signals. For example, with one fixed wait state specified, the Avalon bus module waits for one additional clock cycle before deasserting the address, data and control signals. Wait states are specified by declarations made in the PTF file. They are fixed because the Avalon bus module inserts the same number of wait states for every bus transfer.

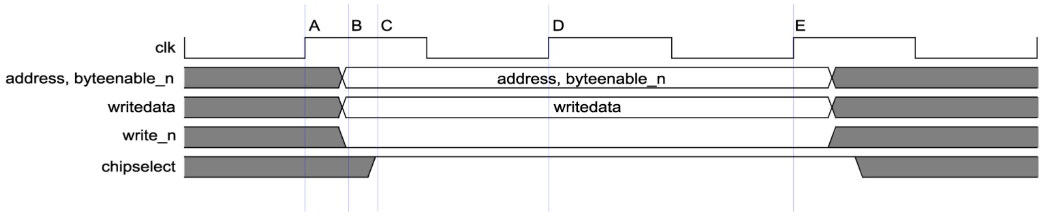
Write transfers with wait states are typically used for peripherals that cannot capture data from the Avalon bus module in a single bus cycle. In this transfer mode, the Avalon bus module presents `address`, `writedata`, `byteenable_n`, `write_n` and `chipselect` during the first bus cycle, exactly like the start of a fundamental write transfer. During the wait states, these signals are held constant. The slave port eventually captures data from the Avalon bus module within the fixed number or wait states. The transfer then terminates, and the Avalon bus module deasserts all signals at the same time.

Example 7 shows an example of a slave write transfer with one wait state.

Example 7. Slave Write Transfer with One Fixed Wait State

This Example Demonstrates	*Relevant PTF Parameters*
Write transfer with wait states to a synchronous slave peripheral	
One fixed wait state	Write_Wait_States = “1”
No setup time	Setup_Time = “0”
No hold time	Hold_Time = “0”





Example 7 Time Reference Description

(A) Write transfer cycle starts on the rising edge of `clk`.
(B) Registered `writedata`, `address`, `byteenable_n`, and `write_n` signals from Avalon bus module are valid.
(C) Avalon bus module decodes `address` and asserts valid `chipselect` to slave.
(D) First (and only) wait state bus cycle ends at the rising edge of `clk`. All signals from Avalon bus module remain constant.
(E) Peripheral captures `writedata`, `address`, `byteenable_n`, `write_n`, and `chipselect` on or before the rising edge of `clk`, and the write transfer terminates.

Slave Write Transfer with Peripheral-Controlled Wait States

Peripheral-controlled wait states allow a target peripheral to stall the Avalon bus module for as many bus cycles as required to capture `writedata`. This feature is useful for peripherals that may require an indefinite number of bus cycles to capture the write data, depending on conditions that vary from transfer to transfer.

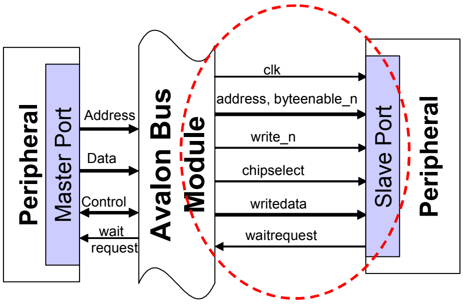
The peripheral-controlled wait state mode uses the `waitrequest` signal, which is an output from the slave port. The Avalon bus module presents `address`, `writedata`, `byteenable_n`, `write_n` and `chipselect` during the first bus cycle, exactly like the start of a fundamental write transfer. If the slave port needs extra time to capture the data, then it must assert `waitrequest` before the next rising clock edge. When asserted, `waitrequest` stalls the Avalon bus module, and forces it to hold `address`, `writedata`, `byteenable_n`, `write_n` and `chipselect` constant. After the slave deasserts `waitrequest`, the bus transfer terminates on the next rising clock edge.

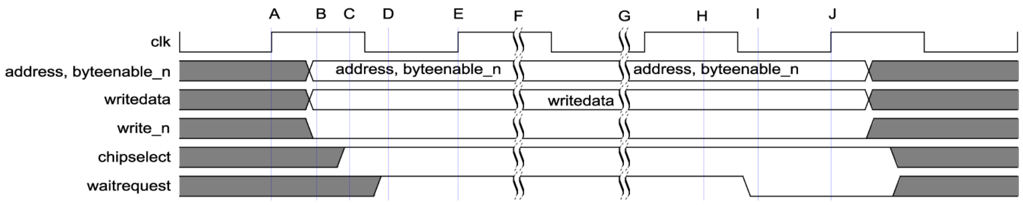
The Avalon bus module does not have a time-out feature to limit how long the slave peripheral can stall. When the Avalon bus module is stalled, somewhere in the system module there is a master peripheral that is stalled as well, waiting for the slave port to capture the write data. A slave peripheral could permanently hang a master peripheral. Therefore, the peripheral designer must ensure that a slave port does not assert `waitrequest` indefinitely.

Example 8 shows an example of a slave write transfer with a peripheral-controlled wait state.

Example 8. Slave Write Transfer with Peripheral-Controlled Wait States

This Example Demonstrates	***Relevant PTF Parameters***
Write transfer to synchronous peripheral	
More than one peripheral-controlled wait state	Write_Wait_States = "peripheral_controlled"
No setup time	Setup_Time ="0"
No hold time	Hold_Time = "0"





Example 8 Time Reference Description

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

(B) Registered outputs `address`, `writedata`, `byteenable_n` and `write_n` signals from Avalon bus module to slave are valid.

(C) Avalon bus module decodes `address`, then asserts `chipselect`.

(D) Peripheral asserts `waitrequest` before the next rising edge of `clk`.

(E) Avalon bus module samples `waitrequest` at the rising edge of `clk`. If `waitrequest` is asserted, the bus cycle becomes a wait state, and `address`, `writedata`, `byteenable_n`, `write_n` and `chipselect` remain constant.

(F-G) With `waitrequest` asserted throughout, an arbitrary unlimited number of bus cycles elapse.

(H) Eventually the slave port captures `writedata`.

(I) Slave port deasserts `waitrequest`.

(J) The write transfer ends on the next rising edge of `clk`. The next bus cycle could be the start of another bus transfer.

When peripheral-controlled wait states are specified, the following restrictions apply to other bus transfer modes. These restrictions apply only to this slave port, not to other slave ports connected to the Avalon bus module.

If peripheral-controlled wait states are specified, setup and hold wait states cannot be used. In almost all cases, a peripheral that can generate the `waitrequest` signal will be on-chip and synchronous that causes setup and hold time considerations unnecessary.

Slave Write Transfer with Setup and Hold Time

The Avalon bus module automatically accommodates setup and hold time requirements for each slave port, based on declarations made in the PTF file. The master peripheral that initiates the write transfer does not need to consider the setup and hold requirements of each slave port. The ports used for a write transfer with setup and hold time are identical to those used for a fundamental write transfer. The difference is in the timing of signals only.

Setup and hold time are generally used for off-chip peripherals that require `address`, `byteenable_n`, `writedata`, and `chipselect` to remain stable for some amount of time before and/or after the `write_n` pulse. A nonzero setup time of M means that, after `address`, `byteenable_n`, `writedata` and `chipselect` signals are presented to the slave peripheral, there is a delay of M bus cycles before `write_n` is asserted. Likewise, a nonzero hold time of N means that, after `write_n` is deasserted, `address`, `byteenable_n`, `writedata` and `chipselect` remain constant for N more bus cycles. Note that `chipselect` is not affected by the setup or hold time. If the peripheral requires a setup or hold time for both `write_n` and `chipselect`, then the designer must manually add the appropriate logic (one AND gate) to the slave port interface.

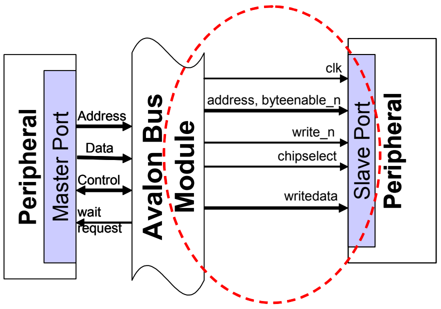
The total number of bus cycles to complete the bus transfer depends on setup, wait-state and hold bus cycles. For example, a peripheral with `Setup_Time = "2"` and `Write_Wait_States = "3"` and `Hold_Time = "2"` will take 8 bus cycles to complete the transfer:

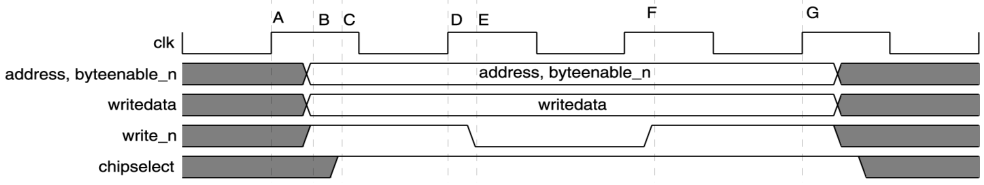
- 2 setup bus cycles plus
- 3 wait-state bus cycles plus
- 2 hold bus cycles plus
- 1 bus cycle to capture data

A slave port does not have to use both setup and hold time at the same time; transfers with only setup or only hold time are acceptable. Example 9 shows a write transfer with both a setup and a hold time requirement.

Example 9. Slave Write Transfer with Setup & Hold Times

This Example Demonstrates	*Relevant PTF Parameters*
Write transfer to synchronous peripheral	
No fixed wait state	Write_Wait_States = "0"
1 bus cycle of setup time	Setup_Time = "1"
1 bus cycle of hold time	Hold_Time = "1"





Example 9 Time Reference Description

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

(B) Registered outputs `address`, `byteenable_n` and `writedata` signals from Avalon bus module are valid, `write_n` remains deasserted.

(C) Avalon bus module decodes `address`, then asserts `chipselect`.

(D) Rising edge of `clk` marks the end of the setup bus cycle.

(E) Avalon bus module asserts `write_n`.

(F) Avalon bus module deasserts `write_n` after the next rising edge of `clk`, `address, byteenable_n`, `writedata` and `chipselect` remain constant as the hold-time bus cycle begins.

(G) Avalon bus module deasserts `address`, `byteenable_n`, `writedata` and `chipselect` on the next rising edge of `clk` and the write transfer terminates.

When setup and/or hold time is specified for a slave port on the Avalon bus, the following restrictions apply to other bus transfer modes. These restrictions apply only to this slave port, not to other peripherals connected to the Avalon bus module.

If a setup time is specified for a slave peripheral, then the same setup time is applied to both read transfers and write transfers. Setup and hold time cannot be used if the slave port uses peripheral-controlled wait states.

Avalon Master Transfers

The following sections discuss bus transfers between a master port and the Avalon bus. From an abstract, system-level viewpoint, master peripherals exchange data with slave peripherals. However, from the viewpoint of a master peripheral, data is transferred between the peripheral's master port and the Avalon bus module only. If the master peripheral *does not* access a defined address in an existing slave peripheral with a slave port connected to the Avalon bus module, an undefined behavior will result. However, the existence of the slave peripheral does not affect the master port interface to the Avalon bus module. It is the Avalon bus module that accepts a transfer from the master port. The Avalon bus module—not the master port—then initiates a slave transfer with the appropriate slave port, and terminates the slave transfer. Therefore, in the following discussions the interface between the master port and the Avalon bus module is the exclusive focus of this discussion.

Compared to the numerous Avalon slave transfer modes, master transfer modes are few and simple. The following discussions assume that the Avalon master peripheral is a synchronous, on-chip module, which is almost always true for Avalon master peripherals. This eliminates the need to consider the myriad requirements of interfacing to off-chip devices. In the event that the master peripheral must reside off-chip—especially in the case that the master address and/or data lines share a tri-state bus—an on-chip bridge module is required to relay the off-chip master's signals to an on-chip Avalon master port.

There is essentially one golden rule of master transactions: Assert all signals to initiate the bus transfer, and then wait until the Avalon bus module deasserts `waitrequest`. With this one rule and the fundamental slave read and write transfers in mind, the master port interface is readily understood.

It is important to realize that under realistic circumstances, bus transfers are not isolated events. They typically happen in continuous succession. For example, a master port may initiate a read transfer from a slave port immediately before or after a write transfer to an unrelated peripheral. During the read bus transfer, the master port's read enable signal is necessarily asserted. However, after the read transfer terminates, the read enable may remain asserted if another read transfer will be initiated on the next bus cycle.

Avalon Signals for Master Transfers

Table 4 below lists the signal names that interface a peripheral's master port to the Avalon bus module and gives a brief description of which ports are required and under what circumstances. Not all of the signals listed in Table 4 are present on all peripherals, depending on the peripheral design and the ports declared in the peripheral's PTF file.

Table 4. Avalon Master Port Signals (Part 1 of 2)

Signal Type	Width	Direction	Required	Description
`clk`	1	in	yes	Global clock signal for the system module and Avalon bus module. All bus transactions are synchronous to `clk`.
`reset`	1	in	no	Global reset signal. Implementation is peripheral-specific.
`address`	1 - 32	out	yes	`Address` lines from the Avalon bus module. All Avalon masters are required to drive a byte address on their `address` output port.
`byteenable`	0, 2, 4	out	no	Byte-enable signals to enable specific byte lane(s) during transfers to memories of width greater than 8 bits. Implementation is peripheral-specific.
`read`	1	out	no	Read request signal from master port. Not required if master never performs read transfers. If used, `readdata` must also be used.
`readdata`	8, 16, 32	in	no	Data lines from the Avalon bus module for read transfers. Not required if the master never performs read transfers. If used, `read` must also be used.
`write`	1	out	no	Write request signal from master port. Not required if the master never performs write transfers. If used, `writedata` must also be used.

Table 4. Avalon Master Port Signals (Part 2 of 2)

Signal Type	Width	Direction	Required	Description
`writedata`	8, 16, 32	out	no	Data lines to the Avalon bus module for write transfers. Not required if the master never performs write transfers. If used, `write` must also be used.
`waitrequest`	1	in	yes	Forces the master port to wait until the Avalon bus module is ready to proceed with the transfer.
`irq`	1	in	no	Interrupt request has been flagged by one or more slave ports.
`irqnumber`	6	in	no	The interrupt priority of the interrupting slave port. Lower value has higher priority.
`endofpacket`	1	in	no	Signal for streaming transfers. May be used to indicate an end of packet condition from the slave to the master port. Implementation is peripheral-specific.
`readdatavalid`	1	in	no	Signal for read transfers with latency and is for a master only. Indicates that valid data from a slave port is present on the `readdata` lines. Required if the master is latency-aware.
`flush`	1	out	no	Signal for read transfers with latency. Master can clear any pending latent read transfers by asserting `flush`.

In the following discussions of Avalon master transfers, the read, write and byteenable signals are used in their active-low form, which is similar to the traditional convention of using active-low read enable, write enable and byte enable signals. Note the following:

- These signals appear in the form `read_n`, `write_n` and `byteenable_n`.
- Any port of an Avalon signal type may be used with active high or low polarity, based on the port's declaration in the PTF file.

Fundamental Master Read Transfers on the Avalon Bus

In the fundamental master read transfer, the master initiates the bus transfer on a rising clock edge by presenting valid address and read request signals to the Avalon bus module. Ideally, the read data returns from the Avalon bus module before the next rising clock edge, and the read transfer terminates in one bus cycle. If the read data is not ready by the next rising clock edge, the Avalon bus module asserts a wait request and stalls the master port until data has been fetched from the addressed slave port. The fundamental master read transfer has no latency.

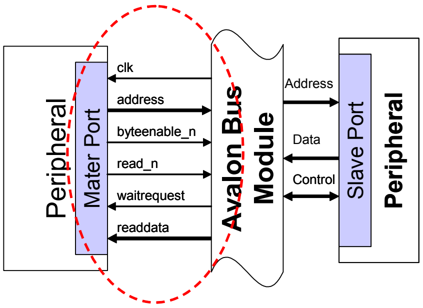
☞ See "Advanced Avalon Bus Transfers" on page 54 for master read transfer with latency and streaming master transfer information.

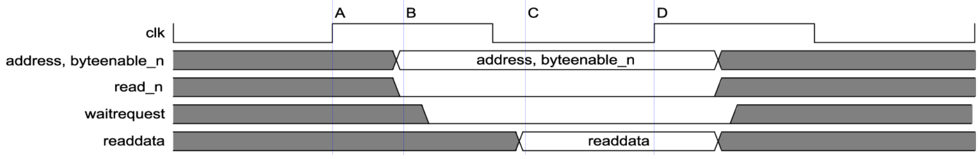
The master read transfer starts on the rising edge of `clk`. Immediately after the first rising edge of `clk`, the master asserts the `address` and `read_n` signals. If the Avalon bus module cannot present `readdata` within the first bus cycle, it asserts `waitrequest` before the next rising edge of `clk`. If the master sees `waitrequest` asserted on the rising edge of `clk`, then it waits. The master must hold all outputs constant until the next rising clock edge after `waitrequest` is deasserted. After `waitrequest` is deasserted, the master port then captures `readdata` on the next rising edge of `clk`, and deasserts `address` and `read_n`. The master may initiate another transfer immediately during the next bus cycle.

Example 10 shows which `waitrequest` is never asserted by the Avalon bus module. The read transfer ends in one bus cycle.

☞ Even though `waitrequest` is never asserted, it is still an active signal in the fundamental master read transfer.

Example 10. Master Read Transfer with No Wait State





Example 10 Time Reference Description

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

(B) Master port asserts valid `address`, `byteenable_n` and `read_n`.

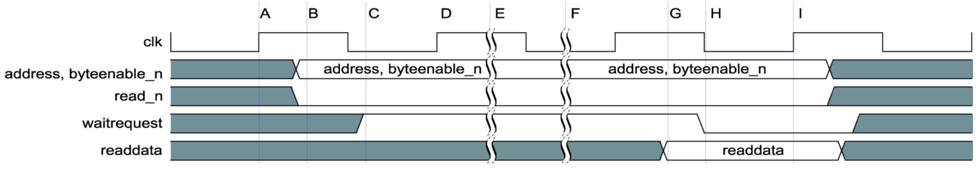
(C) Valid `readdata` returns from Avalon bus module during first bus cycle.

(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 bus cycle could be the start of another bus transfer.

A fundamental read transfer with zero wait states is generally only achievable when the addressed slave peripheral is asynchronous with no latency.

Example 11 shows when `waitrequest` is asserted by the Avalon bus module for an indefinite number of bus cycles. If N is the number of bus cycles that the Avalon bus module asserts `waitrequest`, then the total bus transfer will take (N + 1) bus cycles.

Example 11. Master Read Transfer with Wait States



Example 11 Time Reference Description

(A) First bus cycle starts on the rising edge of `clk`.
(B) Master asserts valid `address`, `byteenable_n` and `read_n`.
(C) Avalon bus module asserts `waitrequest` before the next rising edge of `clk`.
(D) Master port sees `waitrequest` at the rising edge of `clk`. This bus cycle becomes a wait state.
(E-F) As long as `waitrequest` is asserted, master holds all outputs constant.
(G) Valid `readdata` returns from Avalon bus module.
(H) Avalon bus module deasserts `waitrequest`.
(I) Master port captures `readdata` on the next rising edge of `clk` and deasserts all outputs. The read transfer ends here, and the next bus cycle could be the start of another bus transfer.

The Avalon bus module does not offer a time-out feature to the master port. The master port must stall for as long as `waitrequest` remains asserted.

If the master port uses the `byteenable_n` signal, all `byteenable_n` lines must be asserted during master read transfers. A master port can use `byteenable_n` to specify individual byte lanes during master write transfers to wide peripherals, but `byteenable_n` is not used for master read transfers and must be asserted.

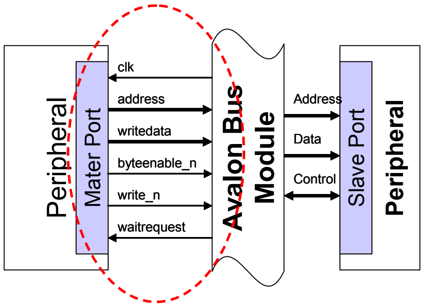
Fundamental Master Write Transfer on the Avalon Bus

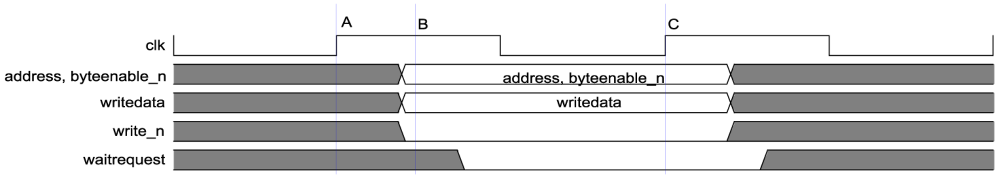
The fundamental master write transfer is used for almost all write transfers to a peripheral with no latency. The master initiates the bus transfer on a rising clock edge, by presenting address, data, and write request signals. Ideally, the target peripheral captures the data on the next rising clock edge, and the write transfer terminates in one bus cycle. If the target peripheral's slave port cannot capture data during the first bus cycle, the Avalon bus module stalls the master port until the slave port captures the data.

The master write transfer starts on the rising edge of `clk`. Immediately after the first rising edge of `clk`, the master asserts the `address`, `writedata` and `write_n` signals. If the data cannot be captured by the next rising clock edge, the Avalon bus module asserts `waitrequest` during the first bus cycle. The master must keep `address`, `writedata` and `write_n` asserted constantly until the next rising clock edge after `waitrequest` is deasserted. After `waitrequest` is deasserted, the master port deasserts `address`, `readdata` and `read_n` on the next rising edge of `clk`. The master may initiate another master transfer during the next bus cycle.

Example 12 shows an example of a fundamental master write transfer. In this example, the Avalon bus module does not assert `waitrequest` and the transfer terminates in one bus cycle.

Example 12. Fundamental Master Write Transfer





Example 12 Time Reference Description

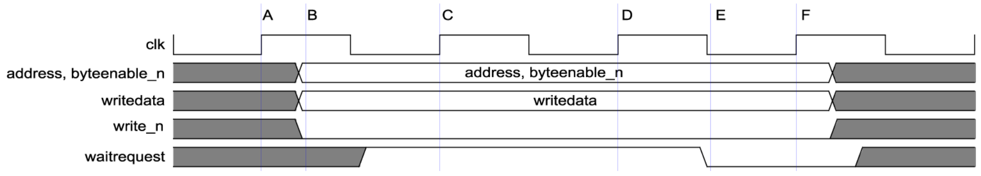
(A) Write transfer starts on the rising edge of `clk`.

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

(C) `waitrequest` is not asserted at the rising edge of `clk`, so write transfer terminates. Another read or write transfer may follow on the next bus cycle.

A write transfer with zero wait states is generally only achievable when the target peripheral is a synchronous peripheral with no latency. Example 13 shows an example in which `waitrequest` is asserted by the Avalon bus module for two bus cycles. The entire write transfer takes three bus cycles.

Example 13. Fundamental Master Write Transfer with Two Wait Requests



Example 13 Time Reference Description

(A) Write transfer on the rising edge of `clk`.
(B) Master asserts valid `address`, data and `write_n`.
(C) `waitrequest` is asserted at the rising edge of `clk`, so this bus 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) Avalon bus module deasserts `waitrequest`.
(F) `waitrequest` is not asserted a 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 bus cycle.

A master port may use the byte enable signal `byteenable_n` to write to specific byte lanes. When present, `byteenable_n` is a bus of 2- or 4-bits wide with one bit for every byte lane in `writedata`. `byteenable_n` is usually necessary for write transfers to off-chip 16-bit or 32-bit memory devices that are word addressable. Some example cases of 32-bit master `byteenable_n` usage are specified in Table 5.

Table 5. Byte Enable Usage for 32-bit Master

byteenable_n[3:0]	Write action
0000	Write full 32-bits
1100	Write lower 2 bytes
0011	Write upper 2 bytes
1110	Write byte 0 only
1011	Write byte 2 only

To write a single byte, the master port should present the byte address rounded down to the nearest master word size and then assert the `byteenable_n` [byte address - master_word_address] signal to the `byteenable` pin. If a master port does not have a `byteenable` pin, the Avalon bus module permanently enables all byte lanes for all write transfers from this master port.

As an example, a 32-bit master writing a byte to address 0xE would assert 0xC on its address and assert `byteenable`[3]. A 64-bit master would have address 0x8 and assert `byteenable`[7] to write a byte to the same location.

Advanced Avalon Bus Transfers

This section describes advanced Avalon bus transfers including bus transfers with latency, streaming transfers, and Avalon bus control signals.

Avalon Read Transfers with Latency

Avalon read transfers with latency increase the bandwidth for synchronous peripherals that require several cycles of latency for the first access, but can return data every bus cycle thereafter. There is no Avalon write transfer with latency, because Avalon write transfers do not require an acknowledge signal to return from the slave port. Latent transfers allow a master to issue a read request, move on to an unrelated task, and receive the data later. This process is often referred to as “posted reads.” The unrelated task could be issuing another read transfer, even though data from the first transfer has not returned yet. This scenario is useful for CPU instruction fetch transfers and DMA read transfers. In these cases, the CPU or the DMA master may pre-fetch expected data, thereby keeping the synchronous memory active and reducing the average access time.

The duration of a read transfer with latency can be divided into two distinct phases, the address phase and the data phase. The key to transfers with latency is the decoupling of the transfer’s address and data phases by providing an extra control signal `readdatavalid` to indicate that valid data has returned from the `readdata` port.

The logic that controls the address and data phases is like two semi-independent ports. An address port initiates the transfer during the address phase, and a data port fulfills the transfer by delivering the data during the data phase. The address and data ports operate independently, except the data port only returns data requested by the address port. Issuing address and control signals to initiate a transfer and capturing the resultant data are conducted independently, and possibly simultaneously. After the Avalon bus module captures an address, the master is free to perform other operations, including issuing more read transfers on its master port. Later (or immediately) during the data phase, the slave port returns valid `readdata` and the appropriate master's `readdatavalid` is asserted.

Latency and wait states are different, but both can occur during a single transfer.

- *Wait states*—A slave peripheral's wait states determine the length of the address phase (i.e., how many clock cycles are required to capture the address), which determines the maximum throughput. For example, if a slave port requires one wait state to present valid data, then at best, the port can complete only one transfer per two clock cycles.

- *Latency*—Latency determines the length of the data phase (i.e., how many clock cycles required for valid data to return), but does not affect the address phase. For example, when accessing a latent slave port with no wait states, a latency-aware master can issue a new read transfer on every clock cycle. The master port can maintain maximum throughput, even though it may wait a few clock cycles of latency for the first valid `readdata` to return.

☞ SOPC Builder automatically generates an Avalon bus module that seamlessly connects any combination of latent or non-latent masters with latent or non-latent slaves.

Master and slave peripherals do not need to know the latency capabilities of the peripherals with which they communicate; the Avalon bus module makes transfers with latency work whenever possible. Therefore, designers can create general-purpose latency-aware master and slave peripherals without *a priori* knowledge of the overall system architecture or latency.

Slave Read Transfer with Fixed Latency

A slave port with fixed latency must be declared with a nonzero `Read_Latency` assignment in the system PTF file. An Avalon slave port with nonzero latency takes one or more bus cycles to produce data after address and control signals have been captured from the Avalon bus module. After the slave port captures the address, the Avalon bus module may immediately initiate a new transfer, even before valid `readdata` has returned from the previous transfer. Recall that non-latent Avalon slave transfers never terminate until the slave has presented valid `readdata` to the Avalon bus module. Therefore, non-latent slaves can have only one pending transfer at a time. Slave ports with nonzero read latency may have multiple transfers pending at any given time.

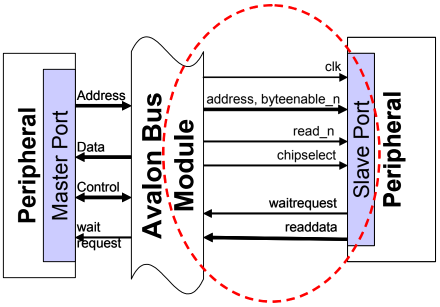
The slave read transfer with latency has two distinct phases: the address phase and the data phase. The timing and sequence of signals during the address phase is identical to that of non-latent Avalon bus transfers, except for the `readdata` signal. During the address phase, the slave port may use setup time and wait states, including peripheral-controlled wait states. After any setup and/or wait states, the slave port must capture address by the last rising clock edge of the transfer. Recall that for read transfers with no latency, valid `readdata` is always asserted on this last rising edge of `clk`. For transfers with latency, `readdata` is not asserted during the address phase. Immediately after the address phase completes, the Avalon bus module 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's read latency is *N*, the slave port must present valid `readdata` on the *Nth* rising edge of `clk` after the edge at which `address` was captured. This latency is fixed; the slave port is absolutely obliged to assert valid `readdata` *N* bus cycles after it captures `address`. For example, if the slave port has a read latency of 1 (i.e., the PTF file declares `Read_Latency = 1`), the slave port presents valid data on the next (i.e., the first) rising edge of `clk` after capturing `address`. The data phase and the bus transfer end after the slave presents `readdata`.

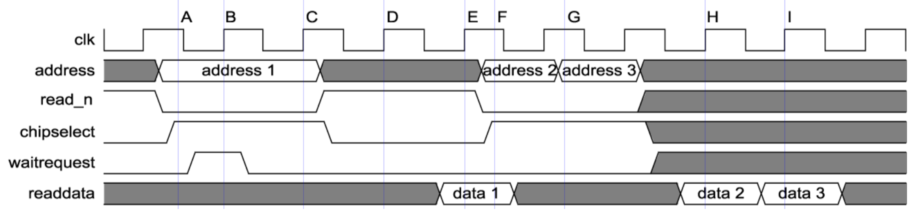
Example 14 shows several data transfers between the Avalon bus module and a latent slave port with a PTF assignment of `Read_Latency = 2`. This slave port uses peripheral-controlled wait states. Slave read transfers with nonzero latency are not guaranteed to have sequential address locations. For example, if there are multiple masters in the system, the slave port does not have control (nor awareness) of the order the Avalon bus module grants access to the masters. A slave port with nonzero read latency can be accessed by master ports that are not latency-aware, and this case does not require any special design considerations. The Avalon bus module accommodates this case by simply forcing the master port to wait until the slave port returns valid data for each transfer. This situation limits the specific master-slave pair to performing a single transfer at a time.

Example 14. Slave Read Transfer with Latency (Part 1 of 2)

This Example Demonstrates	***Relevant PTF Parameters***
Two bus cycles of latency	Read_Latency = “2”
Peripheral-controlled wait states	Read_Wait_States = “peripheral_controlled”



Example 14: Slave Read Transfer with Latency (Part 2 of 2)



Example 14 Time Reference Description

(A) Avalon bus module initiates a read transfer by presenting `chipselect`, `read_n` and `address` for the address phase of the new transfer.
(B) The slave port has asserted `waitrequest` so the previous bus cycle becomes a wait state. The Avalon bus module holds `chipselect`, `read_n` and `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) Avalon bus module asserts `address`, `read_n` and `chipselect` for the next read transfer.
(G) Avalon bus module issues another read transfer during the next bus cycle, before the data from the last transfer returns.
(H) Avalon bus module captures `readdata` after two latency cycles.
(I) Avalon bus module captures `readdata` after two latency cycles.

Slave Read Transfer with Variable Latency

The Avalon interface allows for slave ports that return valid `readdata` after a variable number of clock cycles. Slave read transfers with variable latency are similar to slave read transfers with fixed latency, and the same concept of address phase and data phase applies. Slave ports with variable latency use an additional signal `readdatavalid` to mark when valid data is presented to the Avalon bus module. Slave ports that use the one-bit output signal `readdatavalid` have variable latency by definition.

Slave ports with variable latency *must not* declare a nonzero `Read_Latency` assignment in the system PTF file. A nonzero `Read_Latency` assignment would declare the port to have fixed latency. Slave ports with variable latency also must declare a nonzero `Maximum_Pending_Read_Transactions` assignment in the system PTF file. A *pending read transfer* is a transfer in progress that has completed the address phase (i.e., the slave port has captured the address from the Avalon bus module), for which the corresponding `readdata` result has not yet returned. Slave ports can only accept up to a fixed, predeclared number of pending transactions, as defined by the `Maximum_Pending_Read_Transactions` assignment.

After the address phase, a slave peripheral 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 for one clock cycle. The data phase and the entire transfer end on the next rising clock edge, at which time the Avalon bus module captures `readdata` and `readdatavalid`. The slave port must return `readdata` in the order that it accepted the addresses.

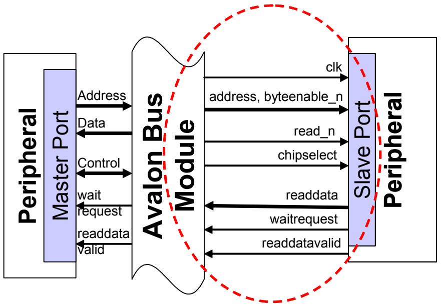
Slave ports with variable latency must return `readdata` at least one clock cycle after capturing `address` from the Avalon bus module. In other words, slave ports with variable latency cannot present `readdata` asynchronously, to be captured on the next clock edge immediately after the Avalon bus module asserts `address`. This timing would be identical to a slave read transfer with no wait states and no latency.

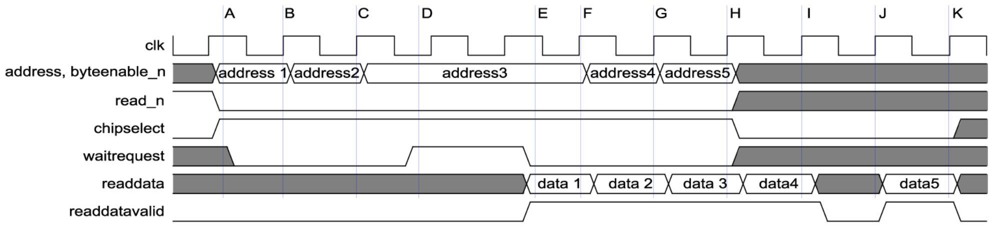
If a slave port can reach its declared maximum number of pending transfers, peripheral-controlled wait states are required. The slave peripheral must assert its `waitrequest` output and stall any new read transfers if it is already processing the declared maximum number of pending read transactions. Typically, the peripheral logic designer does not need to keep track of the number of pending read transactions explicitly. The appropriate behavior for stalling arises if the slave peripheral simply asserts `waitrequest` when its internal read-request buffer or FIFO is full. The designer must, however, inspect the slave peripheral's logic and determine the largest number of pending transactions it will process at one time. This number determines the value of the slave's `Maximum_Pending_Read_Transactions` assignment.

Example 15 on page 60 shows several slave read transfers between the Avalon bus module and a slave port with variable latency and a PTF assignment of `Maximum_Pending_Read_Transactions` = 2. This slave port uses peripheral-controlled wait states. Slave read transfers with variable latency are not guaranteed to have sequential address locations. For example, if there are multiple masters in the system, the slave port does not have control (nor awareness) of the order the Avalon bus module grants access to the masters. A slave port with variable read latency can be accessed by master ports that are not latency-aware, and this case does not require special design considerations. The Avalon bus module accommodates this case by simply forcing the master port to wait until the slave port returns valid data for each transfer. This situation limits the specific master-slave pair to performing a single transfer at a time.

Example 15. Slave Read Transfers with Variable Latency (Part 1 of 2)

This Example Demonstrates	*Relevant PTF Parameters*
Variable latency	Read_Latency = 0
Maximum of 2 pending read transfers	Maximum_Pending_Read_Transactions = “2”
Peripheral-controlled wait states	Read_Wait_States = “peripheral_controlled”





Example 15 Time Reference Description (Part 1 of 2)

(A) The Avalon bus module asserts `address`, `read_n`, and `chipselect`, initiating a read transfer. Assume that the peripheral has no pending transfers at this point.

(B) The slave peripheral 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 Avalon bus module to continue asserting `address`, `read_n`, and `chipselect`. The peripheral asserts `waitrequest` through two bus 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`. The peripheral is not obliged to deassert `waitrequest` just because there are fewer than the maximum number of transfers pending. A peripheral can assert `waitrequest` to stall any transfers.

(F) The Avalon bus module captures `data1` on this rising edge of `clk`. The slave peripheral captures `address3` on this rising edge of `clk`.

(G) The Avalon bus module captures `data2` on this rising edge of `clk`, because the slave peripheral is asserting `readdatavalid`. (`data2` required 4 clock cycles of latency to return.) The Avalon bus module asserts `address`, `read_n`, and `chipselect`, and the peripheral captures `address4`.

Example 15 Time Reference Description (Part 1 of 2)

(H) The Avalon bus module captures `data3` on this rising edge of `clk`, because the slave peripheral is asserting `readdatavalid`. (Note that `data3` required 2 cycles of latency to return.) The Avalon bus module is asserting `address`, `read_n`, and `chipselect`, and the peripheral captures `address5`.

(I) The Avalon bus module captures `data4` on this rising edge of `clk`, because the slave peripheral is asserting `readdatavalid`. The Avalon bus module has deasserted `chipselect`, ending the sequence of read transfers.

(J) The Avalon bus module does not capture data on this edge of `clk` because the slave peripheral has deasserted `readdatavalid`.

(K) The Avalon bus module captures `data5` on this rising edge of `clk`, completing the data phase for the final pending read transfer.

Slave ports *must* return valid data for every transfer that is initiated; a slave cannot "cancel" a transfer once it is initiated. Further, the peripheral cannot refuse a transfer. Once the Avalon bus module initiates the slave read transfer by asserting `address` and `read_n`, the slave port must return valid data and complete the transfer. Slave ports only have control over *when* they return data. The slave peripheral can stall the Avalon bus module for as long as necessary to capture the new address. Additionally, latency-aware slave peripherals can take an arbitrary number of clock cycles to produce valid `readdata`.

The Avalon bus module can initiate a slave write transfer even if the slave peripheral is processing one or more pending read transfers. The slave is still responsible for returning data for all pending read transfers. If the slave peripheral cannot handle a write transfer while it is processing pending read transfers, the slave port can assert its `waitrequest` and stall the write operation until the pending read transfers have completed.

If 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. The slave port could return the data at the address *prior* to the write operation, or the slave port could return the data at the address *after* the write operation. Designers of Avalon slave ports with latency must specify the behavior of their logic under this circumstance. "The outcome is undefined," is an acceptable specification.

When variable latency is specified for a slave port, the following restrictions apply to other bus transfer modes:

- Slave ports with variable latency cannot be used with fixed wait states.
- Slave ports of bus type `"avalon_tristate"` cannot have variable read latency.

- Slave ports of bus type "`avalon_tristate`" cannot have a `readdatavalid` output port.

☞ These restrictions apply only to this slave port, not to other peripherals connected to the Avalon bus module.

Master Read Transfer with Latency

A master peripheral that uses the one-bit input signal `readdatavalid` is latency-aware by definition. A latency-aware master peripheral can initiate a new master read transfer before it receives valid data from a previous transfer. Recall that Avalon master read transfers without latency never terminate until the master has captured data from the Avalon bus module. Therefore, non-latent masters can have only one pending transfer at a time. Latency-aware master ports may have an arbitrary number of read transfers pending at any given time.

The latency-aware master read transfer has two distinct phases: the address phase and the data phase. The timing and sequence of signals during the address phase is identical to that of non-latent Avalon transfers, except for the `readdata` signal. The master address port must present `read` and (if necessary) `address` and `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. Recall that for read transfers with no latency, valid `readdata` is always available on this last rising edge of `clk`. For transfers with nonzero latency, `readdata` may not be returned immediately after the address phase. Valid `readdata` is returned sometime later when the Avalon bus module asserts `readdatavalid`. Immediately after the address phase completes, the master address port can initiate another read or write transfer.

There are two rules for latency-aware Avalon master ports:

- Once you initiate a transaction, heed the `waitrequest` signal.
- For every read transfer that is initiated, the Avalon bus module asserts `readdatavalid` for one clock cycle only. The master port must capture valid `readdata` on the same rising edge of `clk` when `readdatavalid` is asserted.

The Avalon bus module always returns valid `readdata` in the order that it was requested by the master. At any time while the master address port actively issues addresses, the master data port may be capturing valid data from the current or a previous transfer, or may be waiting for the Avalon bus module to present valid data. The Avalon bus module asserts `readdatavalid` when it presents valid `readdata`. The master data port must capture `readdata` on the rising clock edge that `readdatavalid` is asserted. This clock cycle is the only time that `readdata` is guaranteed to be valid. Therefore, if a condition arises in which the master cannot immediately process incoming `readdata`, the master peripheral must have a FIFO buffer connected to the `readdata` input port to guarantee that data from the Avalon bus module is not lost.

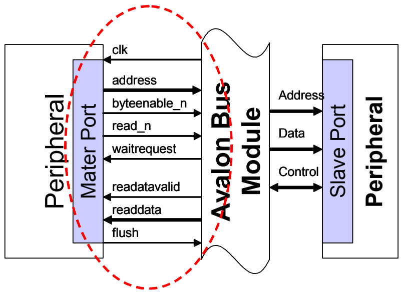
All latency-aware Avalon master ports have variable latency. The number of clock cycles of latency is not fixed for a latency-aware master port. The Avalon bus module makes no guarantees about when valid `readdata` will return, only that data will return in the order it was requested. Therefore, the master data port must be designed to accept an arbitrary number of latency cycles, regardless of the fixed latency of any target slave ports, including zero-latency transfers. The Avalon bus module may introduce latency cycles beyond those cycles required by a slave port with latency. For example, extra latency is usually introduced when a peripheral is accessed through an Avalon tri-state bridge. The tri-state bridge includes internal registers that introduce latency, while improving system $\mathbf{f_{MAX}}$ and simplifying the connection to off-chip devices.

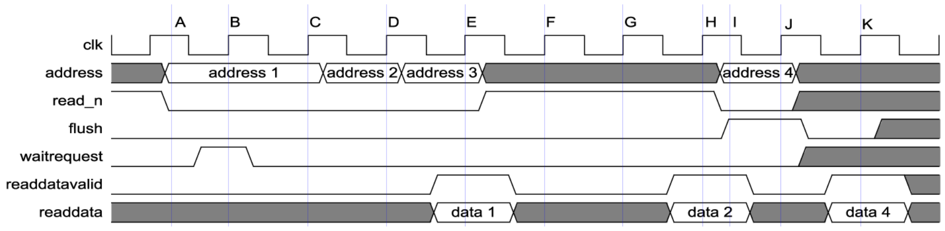
A latency-aware master can access a non-latent slave port without special design considerations. From the master port's perspective, the latency is zero. `readdata` is presented immediately on the rising edge of `clk` that the address phase ends, which is the same as a Avalon bus transfer with no latency. There are no special design requirements for simultaneous multi-master transfers. If multiple masters coexist in a system and issue read transfers to latent and/or non-latent slave ports, the Avalon bus module appropriately performs arbitration, and guarantees that each master receives its requested data in order. This behavior includes the case in which a master port issues `address` to one slave port, while capturing `readdata` from a different slave port.

There may be cases in which the master peripheral determines that it does not need the data from a transfer it has already issued. In such cases, the master port can use the `flush` signal to clear any pending read transfers. For example, a latency-aware master port for fetching CPU instructions may issue several read transfers to pre-fetch instructions, but if a branch instruction is encountered, all pending pre-fetched instructions become irrelevant. The master's data port can assert `flush` on a rising edge of `clk` to clear all pending transfers. `readdatavalid` is deasserted until the next new read transfer's data is ready on the `readdata` port. The Avalon bus module can capture a new value on `address` at the same time that `flush` is asserted. The data corresponding to this address becomes the next valid data to return on `readdata`.

Example 16 shows data transfers with latency between the Avalon bus module and a latency-aware master port. There is no pattern to why and when `waitrequest` and `readdatavalid` are asserted in this example; however, the example shows that no matter what the timing, the master port must respond appropriately to both `waitrequest` and `readdatavalid`. In this example, the second-to-last transfer is flushed using the `flush` signal. However, the unwanted data could have appeared on `readdata` if the latency for that transfer was shorter.

Example 16. Master Read Transfer with Latency





Example 16 Time Reference Description

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

(B) Avalon bus module has asserted `waitrequest` so the master port waits and holds `address` and `read_n` constant for another bus cycle.

(C) `waitrequest` is not asserted, so the Avalon bus module captures `address` at the rising edge of `clk`. `readdatavalid` is not asserted, so master does not capture `readdata`.

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

(E) The Avalon bus module 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`.

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

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

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

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

(J) `readdatavalid` is not asserted, so master does not capture `readdata`. `flush` is asserted, so Avalon bus module flushes the pending transfer. Avalon bus module captures the new `address`.

(K) `readdatavalid` is asserted, so master captures valid `readdata`. No more transfers are pending.

Streaming Transfer

Streaming transfers create an open channel between a streaming master and streaming slave to perform successive data transfers. This channel allows data to flow between the master-slave pair as data becomes available, without requiring the master to continuously access status registers in the slave peripheral to determine whether the slave can send or receive data. Streaming transfers maximize throughput between a master-slave pair, while avoiding data overflow or underflow on the slave peripheral.

In the streaming transfer mode, simple flow control signals are presented from slave to master, such that whenever the slave has new data (or can accept new data), the Avalon bus module automatically transfers the data. The streaming transfer mode eliminates the bandwidth overhead required for the master to check status registers, because the master does not need to access and compare slave status registers for each transfer. This reduces the design complexity of master peripherals with limited intelligence, such as DMA controllers, which may have only simple flow control signals and a counter to transfer data between a slave peripheral and incremental locations in a slave memory.

Streaming Slave Transfers

The slave interface for streaming peripherals introduces three signals in addition to those used for fundamental slave transfers: `readyfordata`, `dataavailable`, and `endofpacket`. A streaming slave port is defined as a slave port that uses one or more of these signals.The slave indicates that it is ready to accept a write transfer from the Avalon bus module by asserting `readyfordata`. The slave indicates that it can produce data for a read transfer from the Avalon bus module by asserting `dataavailable`. When deasserted, these signals force the Avalon bus module (and also the streaming master port that initiated the transfer) to wait until the slave is ready to proceed.

This behavior in which the Avalon bus module initiates a transfer only when `dataavailable` or `readyfordata` is asserted applies only to the case of a transfer between a streaming master port and a streaming slave port. A transfer from a non-streaming master port may be issued to a slave port at any time, regardless if the slave port is streaming or not. For example, the Avalon bus module may issue a slave transfer from a non-streaming master (CPU) to a streaming slave port, even while another transfer from a streaming master (DMA controller) is waiting because `dataavailable` is deasserted.

During any transfer, a streaming slave port can assert the `endofpacket` signal, which is passed through the Avalon bus module to the master peripheral so that it can respond. The interpretation of the `endofpacket` signal is dependent on the design, and the master peripheral must be aware of how to respond appropriately. `endofpacket` does not guarantee that the Avalon bus module will stop the stream of transfers to the slave port. For example, `endofpacket` may be used as a packet delineator, so the master peripheral knows where packets start and end in a longer stream of data. Alternately, `endofpacket` could be designed to interrupt the stream of transfers, and force the master to come back at a later time to continue any further read or write transfers.

Streaming Slave Read Transfer

A streaming slave peripheral indicates that it is can accept a read transfer by asserting `dataavailable`. The Avalon bus module will never initiate a read transfer when `dataavailable` is deasserted. When `dataavailable` is asserted, the Avalon bus module can start a read transfer by asserting `chipselect` at a rising edge of `clk`, similar to any other Avalon read transfer. The timing and sequencing of the `read_n`, `byteenable_n` and `readdata` signals follow the same order as a normal slave read transfer. Based on declarations in the system PTF file, the transfer may use setup time and/or wait states, including peripheral-controlled wait states.

After a transfer terminates, if the peripheral cannot produce more data for subsequent read transfers, it must deassert `dataavailable` so that the Avalon bus module does not attempt to initiate another read transfer on the next rising edge of `clk`. When the peripheral deasserts `dataavailable`, the Avalon bus module is forced to deassert `chipselect`, `read_n`, `address` and `byteenable_n` to this slave port. Therefore, the Avalon bus module cannot begin another read transfer with this slave port until the peripheral asserts `dataavailable` again. If a streaming master port initiates a read transfer (or continues to initiate consecutive read transfers) while the slave port's `dataavailable` is deasserted, the master port is simply forced to wait until the slave port can transfer data again.

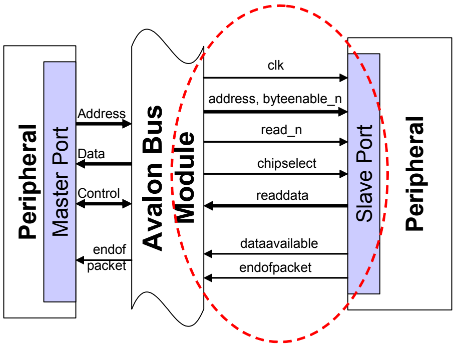
The function of endofpacket is not specified in the Avalon bus specification; the signal is simply passed through the Avalon bus module to the master port. The following guidelines are not part of the Avalon bus specification, but may help ensure that the master port can capture endofpacket from the slave port. The slave port should assert endofpacket at the same time as it asserts valid readdata, so that the master can capture endofpacket together with readdata. The slave port may deassert endofpacket for each transfer, or the peripheral may assert endofpacket indefinitely and wait for a master to reset it.

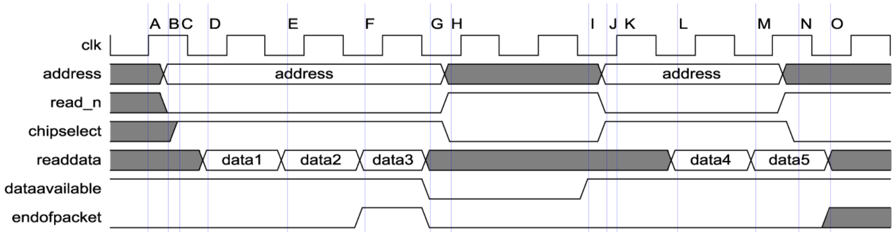
Example 17 on page 69 shows a streaming slave read transfer. In this example, assume that an Avalon streaming master peripheral initiates a sequence of streaming transfers, starting while the slave port has dataavailable asserted. Furthermore, assume that the master continues initiating read transfers in immediate succession. At some time during the sequence, the slave port deasserts dataavailable, forcing the Avalon bus module (and the master port) to wait. Later the slave port asserts dataavailable again, and the Avalon bus module continues the sequence of slave read transfers. In this example, note that data is read from a constant slave address that presents new data on each transfer.

This is common operation for a register-controlled peripheral, such as a UART or SPI. Example 17 on page 69 shows the slave port asserting 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 with the Avalon bus module deasserting chipselect and read_n while dataavailable is still asserted, meaning that the master port, not the slave, has chosen to end the sequence of transfers.

Example 17. Streaming Slave Read Transfer (Part 1 of 2)

This Example Demonstrates	*Relevant PTF Parameters*
Slave port accepting streaming read transfer	
No fixed wait state	Read_Wait_States = “0”
No setup time	Setup_Time = “0”





Example 17 Time Reference Description (Part 1 of 2)

(A) First bus cycle on the rising edge of `clk`.
(B) Registered outputs `address` and `read_n` from the Avalon bus to slave are valid.
(C) Avalon bus module decodes `address`, then asserts `chipselect`.
(D) Slave port asserts valid `readdata` before the next rising edge of `clk`. The Avalon bus module captures `readdata` on the next rising edge of `clk`.
(E) For each bus cycle that `chipselect` and `read_n` remain asserted, the slave port produces valid `readdata`. (In this example, `address` remains constant, but this may not be the case for all peripheral designs).
(F) The slave port may asset `endofpacket` at any time while it asserts valid `readdata`. (In this example, the slave deasserts `endofpacket` after one bus cycle, but this may be different depending on the peripheral designs.
(G) The streaming slave deasserts `dataavailable`, forcing the Avalon bus module to postpone any subsequent streaming reads. Note that `read_n` and `chipselect` are still asserted, indicating that the streaming master port is still waiting for the transfer to terminate.
(H) The Avalon bus module deasserts `address, read_n` and `chipselect` in response to `dataavailable`.
(I) At some point later, the slave port asserts `dataavailable`.

Example 17 Time Reference Description (Part 2 of 2)

(J) In response to `dataavailable`, the Avalon bus module reasserts `address`, `read_n` and `chipselect`. (If there were no pending streaming transfer, these signals would remain undefined).
(K) A new streaming read transfer begins on the rising edge of `clk`.
(L-M)The slave port asserts valid `readdata` before the rising edge of `clk` for every bus cycle that `chipselect` and `read_n` remain asserted.
(N) The Avalon bus module deasserts `read_n` and `chipselect`, indicating that for now – there are no pending streaming transfers
(O) In this example `dataavailable` remains asserted, indicating that another streaming transfer may begin at any later bus cycle.

Streaming Slave Write Transfer

A streaming slave peripheral indicates that it is can accept a write transfer by asserting `readyfordata`. The Avalon bus module will never initiate a write transfer when `readyfordata` is deasserted. When `readyfordata` is asserted, the Avalon bus module can start a write transfer by asserting `chipselect` and `address` at a rising edge of `clk`, similar to any other Avalon read transfer. The timing and sequencing of the `write_n`, `byteenable_n` and `writedata` signals follow the same order as a normal slave read transfer. Based on declarations in the system PTF file, the transfer may use setup time, hold time and/or wait states, including peripheral-controlled wait states.

After a transfer terminates, if the peripheral cannot capture more data on subsequent write transfers, it must deassert `readyfordata` so that the Avalon bus module does not initiate another write transfer on the next rising edge of `clk`. When the peripheral deasserts `readyfordata`, the Avalon bus module is forced to deassert `chipselect`, `write_n`, `address` and `byteenable_n` to this slave port. Therefore, the Avalon bus module cannot begin another write transfer with this slave port until the peripheral asserts `readyfordata` again. If a streaming master port initiates a write transfer (or continues to initiate consecutive write transfers) while the slave port's `readyfordata` is deasserted, the master port is simply forced to wait until the slave port can capture data again.

The function of `endofpacket` is not specified in the Avalon bus specification; the signal is simply passed through the Avalon bus module to the master port. The following guidelines are not part of the Avalon bus specification, but may help ensure that the master port can capture `endofpacket` from the slave port. The slave port should assert `endofpacket` as soon as possible after it captures `writedata` from the Avalon bus module. The slave port must assert `endofpacket` before `chipselect` is deasserted in order for the master port to capture `endofpacket` during the same bus transfer. The slave port may deassert `endofpacket` for each transfer, or the peripheral may assert `endofpacket` indefinitely and wait for a master to reset it. If the streaming slave peripheral requires hold time (a very rare case), then `endofpacket` should be held valid until the Avalon bus module deasserts `chipselect` (even after `write_n` is deasserted).

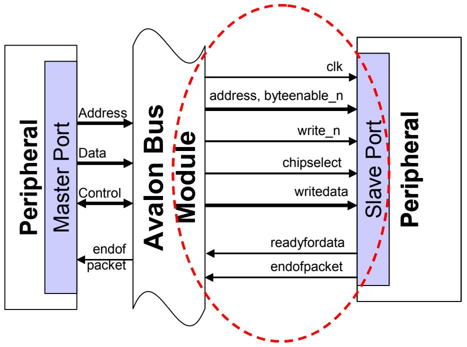
Figure 18 shows an example of a streaming slave write transfer. In this example, assume that an Avalon streaming master peripheral initiates a sequence of streaming transfers, starting while the slave port has `readyfordata` asserted. Furthermore, assume that the master continues initiating write transfers in immediate succession. At some time during the sequence, the slave port deasserts `readyfordata`, forcing the Avalon bus module (and the master port) to wait. Later the slave port asserts `readyfordata` again, and the Avalon bus module continues the sequence of slave write transfers.

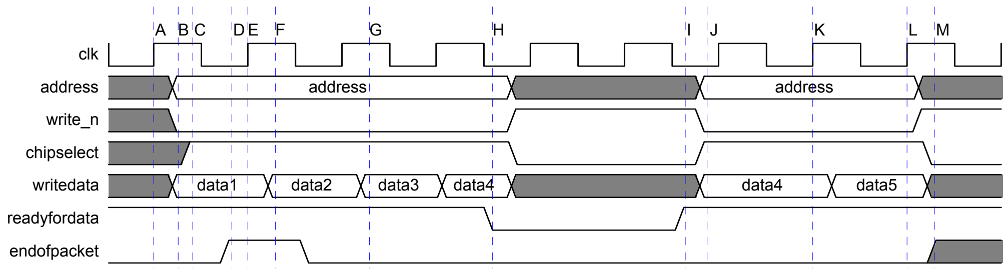
☞ In this example, data is written to a constant slave address. This is common operation for a register-controlled peripheral, such as a UART or SPI.

Example 18 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 Avalon bus module deasserting `chipselect` and `write_n` while `readyfordata` is still asserted, meaning that the master port, not the slave, has chosen to end the sequence of transfers.

Example 18. Streaming Slave Write Transfer (Part 1 of 2)

This Example Demonstrates	*Relevant PTF Parameters*
Slave port accepting streaming write transfer	
No fixed wait state	Write_Wait_States = “0”
No setup time	Setup_Time = “0”
No hold time	Hold_Time = “0”





Example 18 *Time Reference Description (Part 1 of 2)*

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

(B) Registered outputs `address`, `write_n` and `writedata` from Avalon bus to slave are valid.

(C) Avalon bus module decodes `address`, then asserts `chipselect`.

(D) If necessary, the slave asserts `endofpacket` before the last rising edge of `clk` for the current bus transfer. In this example, the slave deasserts `endofpacket` after one bus cycle, but this may be different depending on the peripheral design.

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

(F-G) For each bus cycle that `chipselect` and `write_n` remain asserted, the Avalon bus module produces a valid `writedata`, and the slave port must capture on the following rising edge of `clk`. In this example, `address` is held constant, but this may not be the case for all peripheral designs.

(H) The streaming slave deasserts `readyfordata`, forcing the Avalon bus module to postpone any subsequent streaming writes. Note that `write_n`, `chipselect` and `writedata` are still asserted, indicating that the streaming master port is still waiting for the transfer to terminate. In response, the Avalon bus module deasserts `address`, `write_n`, `chipselect` and `writedata`.

Example 18 ***Time Reference Description (Part 2 of 2***

(I) At some point later, the slave port asserts `readyfordata` again.

(J) In response to `readyfordata`, the Avalon bus module reasserts `address`, `write_n`, `chipselect` and `writedata`. Note that if there were no pending streaming transfer, these signals would remain undefined. A new streaming write transfer begins on the next rising edge of `clk`.

(K-L) The slave port captures `writedata` on the rising edge of `clk`. For each bus cycle that `chipselect` and `write_n` remain asserted, the Avalon bus module presents valid `writedata`.

(M) The Avalon bus module deasserts `write_n` and `chipselect`, indicating that for now, there are no pending streaming transfers. In this example `readyfordata` remains asserted, indicating that another streaming transfer may begin at any later bus cycle.

Streaming Master Transfers

The interface for streaming master peripherals is almost identical to the interface used for normal Avalon master transfers. The streaming master interface introduces only one extra signal, `endofpacket`, which may or may not be necessary depending on the peripheral design. The timing and sequencing of the `write_n`, `read_n`, `address`, `writedata`, `readdata`, `byteenable_n` and other signals follow the same order as a normal master transfer. A streaming master is defined as a master port that has the "Do_Stream_Reads" or "Do_Stream_Writes" or both parameters declared in the PTF file.

If the Avalon bus module requires the master to wait at any time, the Avalon bus module asserts the `waitrequest` signal, and the master port must obey. There are several reasons why the master may have to wait. For example, another master may be accessing the target slave port; the slave port may be requesting wait states; the streaming slave port may not be able to present or accept new data; and so on. The cause of `waitrequest` does not concern the master port, because in any event, the master port cannot abort a transfer once it has started. The master port must only abide by `waitrequest`. Logic inside the Avalon bus module hides the details from the master port, which simplifies the peripheral design for streaming master peripherals.

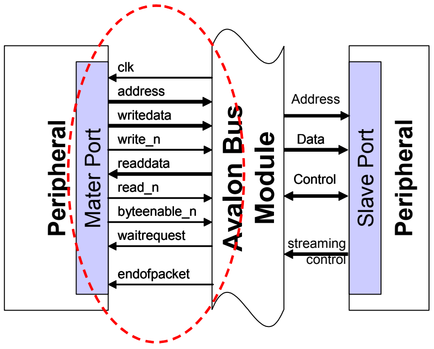
If present, the `endofpacket` signal is passed from the slave port to the master port during each transfer. The master port captures `endofpacket` on the last rising clock edge of the transfer, for both master read and write transfers. The interpretation of the `endofpacket` signal is dependent on the peripheral design. For example, `endofpacket` may be used as a packet delineator, so the master peripheral knows where packets start and end in a longer stream of data. Alternately, based on the value of `endofpacket`, the master peripheral could be designed to determine whether or not to initiate another transfer.

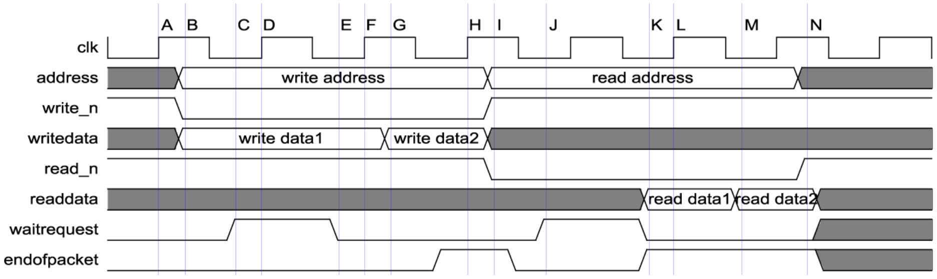
The Avalon bus module does not offer a time-out feature to the master port, regardless if the peripheral is streaming or not. The master port must stall for as long as `waitrequest` remains asserted, and there is no way to abort the transfer. Therefore, if the master needs a method to conditionally transfer data to or from the slave only when the slave is ready, the master-slave pair must employ some convention that uses either the `endofpacket` signal or a status register inside the slave peripheral, or both.

The function of `endofpacket` is not specified in the Avalon bus specification; the signal is simply passed from the slave port to the master port through the Avalon bus module. The following guidelines are not part of the Avalon bus specification, but may help ensure that the master port can capture `endofpacket` from the slave port at a well-defined time. The master port may capture the `endofpacket` signal on the last rising edge of `clk` for the current transfer. This is the clock edge for which the Avalon bus module has deasserted `waitrequest`, and the master port is ready to terminate the transfer. When and why the slave port deasserts `endofpacket` depends on the design of the peripheral. However, note that the master port only sees a valid `endofpacket` signal during a streaming transfer while addressing the appropriate slave port.

Example 19 shows an example of a streaming master read followed by a streaming master–write transfer in which both `waitrequest` and `endofpacket` are asserted at some time the during transfer.

Example 19. Streaming Master Read Followed by a Streaming Master Write Transfer





Example 19 Time Reference Description

(A) First bus cycle starts on the rising edge of `clk`.
(B) Master port asserts `address`, `write_n` and valid `writedata`.
(C) Avalon bus module asserts `waitrequest` before the next rising edge of `clk`, forcing the master port to wait.
(D) `waitrequest` is asserted at the rising edge of `clk`, so master port holds `address`, `write_n` and `writedata` constant.
(E) Avalon bus module deasserts `waitrequest`.
(F) Avalon bus module captures `writedata` on the rising edge of `clk`.
(G) Streaming master port keeps `address` and `write_n` asserted and asserts a new `writedata`. Note that `address` does not 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 streaming write transfer by deasserting `address`, `write_n` and `writedata`.
(I) Master port immediately begins a read transfer during the next bus cycle by asserting `read_n` and a valid `address`.
(J) Avalon bus module asserts `waitrequest` to indicate that it cannot return valid data on the next rising edge of `clk`.
(K) Eventually the Avalon bus module deasserts `waitrequest` and presents valid `readdata`. In this example the Avalon bus module asserts `endofpacket`, but interpretation is left to the streaming master peripheral.
(L) Master port captures `readdata` and `endofpacket`, if necessary on the rising edge of `clk`.
(M) Master port keeps `address` and `read_n` asserted for another streaming read transfer, so the Avalon bus module presents valid `readdata`.
(N) Master port deasserts `read_n` and `address`, and the transfer terminates.

Avalon Bus Control Signals

The Avalon bus module provides some control signals with system-level functionality, which, may not be directly related to the functionality of individual data transfers.

Interrupt Request Signal

Most microprocessor systems require interrupt generation and prioritization logic. The system module implements this service for peripherals and processors connected to the Avalon bus module.

Each slave port may use an `irq` output signal, that can be asserted whenever the peripheral wishes to generate an interrupt. The Avalon bus specification does not define why or when `irq` should be asserted. The timing of the `irq` signal has no relationship to any bus transfer, and `irq` may be asserted at any time. In most practical cases, the slave should assert `irq` and keep it asserted until a master port explicitly resets the interrupt request. An interrupt priority is assigned to each slave port that uses `irq`. The IRQ priorities for each slave port are specified in the system PTF file. Lower IRQ values have higher interrupt priority with IRQ 0 having the highest possible priority.

Master ports may use two input signals to handle interrupt requests: `irq` and `irqnumber`. The `irq` output signals from all slave ports in the system module are ORed together and passed to the master port, such that when any slave port generates an interrupt, `irq` on the master port(s) is asserted. Logic inside the Avalon bus module presents the encoded value (0 to 63) of the IRQ with highest priority on the 6-bit `irqnumber` port. If multiple masters use the `irq` and `irqnumber` signals, each master receives the same values on `irq` and `irqnumber`. The Avalon bus specification does not specify when or how the master peripheral(s) in the system module should respond to the `irq` signal. In most practical cases, a master must respond to the IRQ and then manually reset the IRQ in the slave that generated it.

Reset Control Logic

The system module has a single reset input port, that user logic external to the system module can use to reset the system module and any peripherals it contains. This global reset signal is combined with other reset logic inside the system module, and then distributed to all Avalon peripherals that choose to use the signal type `reset`. Each Avalon peripheral may interpret (or ignore) `reset` based on peripheral design requirements.

Inside the system module, three conditions may cause the `reset` signal to be asserted:

- The PLD has been reconfigured—Immediately after the PLD completes configuration, the Avalon bus module detects this state, and asserts the `reset` signal on all Avalon peripherals for at least one clock cycle.
- The global reset input on the system module has been asserted
- An Avalon slave port has asserted its `resetrequest` signal (defined below).

In general, peripherals that perform operations spanning multiple clock cycles should enter a well-defined reset state whenever `reset` is asserted. The timing of the `reset` signal has no relationship to any bus transfer, and `reset` could be asserted at any time.

Slave ports can use the `resetrequest` signal to force the entire system module to reset. `resetrequest` is useful for functions like watchdog timers, that—if not serviced within a guaranteed amount of time— can reset the entire system. Why and when a peripheral should assert `resetrequest` is not defined in the Avalon bus specification. Note that `resetrequest` is not a request signal like an IRQ that can be serviced at some later time. `resetrequest` causes the Avalon bus module to immediately assert the `reset` signal on all Avalon peripherals that use the `reset` signal, and does not allow other Avalon peripherals to finish pending operations before acknowledging the reset.

Begin Transfer Signal

The `begintransfer` input signal on the slave port offers an easy-to-understand indicator that a new Avalon slave transfer has been initiated. Avalon peripherals, by definition, abide by the Avalon bus specification, and must generate and accept Avalon bus signals in the appropriate sequence. It may be difficult for logic not directly related to the Avalon interface (including the designer's own brain) to determine exactly when an Avalon slave transfer begins, because the address, read enable, write enable, and `chipselect` signals do not necessarily change at the start of each transfer. The Avalon bus module asserts the `begintransfer` signal during the first bus cycle of each Avalon slave transfer. Usage is peripheral-specific.

`begintransfer` can be a helpful debugging signal for clarification when simulating Avalon transfers. `begintransfer` can also simplify the design of less intelligent peripheral functionality, such as clear-on-read operations, set–on–write operations, or other operations that does not require the logic to perform all aspects of Avalon transfers.

Avalon Interface to Off-Chip Devices

This section describes the Avalon tristate interface that allows direct connection of off-chip devices to the Avalon bus module via I/O pins on the PLD. The PTF assignment
`Bus_Type = "avalon_tristate"` is used to specify that an off-chip peripheral uses the Avalon tristate interface. Most systems require an interface to some form of off-chip memory device. Off-chip memory devices often share address and data bus lines on the physical printed circuit board (PCB). This necessitates an interface with bidirectional data pins that can be tristated to allow other bus agents to drive the data lines without causing signal contention. The Avalon tristate interface specifies an adequate interface to connect simple off-chip slave devices such as flash, SRAM, and synchronous SRAM (SSRAM) via device I/O pins. Some Avalon transfer modes are not available to off-chip devices.

The scope of the Avalon tristate interface is limited to off-chip slave peripherals. The off-chip slave peripheral can use peripheral-controlled wait states or constant setup, hold, and wait states. The peripheral can use constant latency, but it cannot use variable latency. The Avalon tristate interface does not extend to off-chip master peripherals. Designers can accommodate off-chip masters by creating a user-defined peripheral (implemented on-chip) that acts as a bridge between the Avalon interface and the protocol used by the off-chip device. Such bridges tend to be complex and specific to an application. They are not nearly as ubiquitous as simple external slave memory devices, and therefore the Avalon bus specification does not attempt to cover these cases.

Avalon Tristate Signals for Slave Transfers

Table 6 lists the signal types that interface an off-chip slave device to the Avalon bus module. The signal direction is from the perspective of the slave device. Not all of the signal types listed in Table 6 are necessary on all peripherals, depending on the peripheral design and the ports declared in the peripheral's PTF file. Table 6 gives a brief description of which signals are required and under what circumstances.

Table 6. Avalon Tristate Slave Port Signals

Signal Type	Width	Direction	Required	Description
`clk`	1	in	no	Global clock signal for the SOPC module and Avalon bus module. All bus transactions are synchronous to `clk`. Only asynchronous slave ports can omit `clk`.
`reset`	1	in	no	Global reset signal. Implementation is peripheral-specific.
`chipselect`	1	in	no	Chip select signal to the slave. The slave port should ignore all other Avalon signal inputs unless `chipselect` is asserted.
`address`	1 - 32	in	no	Address lines from the Avalon bus module. `address` always carries a byte-address value.
`data`	8, 16, 32	bidirectional	yes	Data lines to/from the Avalon bus module for write and read transfers. If used, the `read` or `write` signal must also be used. An Avalon tristate peripheral is defined by the presence of the `data` port.
`read`	1	in	no	Read request signal to slave. Not required if the slave never outputs data to a master.
`outputenable`	1	in	no	Off-chip devices can only drive active data when `outputenable` is asserted. Equivalent to `read` signal on slaves with no latency.
`write`	1	in	no	Write request signal to slave. Not required if the slave never receives data from a master.
`byteenable`	1, 2, 3, 4	in	no	Byte-enable signals to enable specific byte lane(s) during transfers to memories of width greater than 8 bits. Implementation is peripheral-specific.
`writebyteenable`	1,2,3,4	in	no	Logical AND of `write` and `byteenable` signals. Useful control signal for certain types of memory peripherals, especially off-chip SSRAM.
`irq`	1	out	no	Interrupt request. Slave asserts `irq` when it needs to be serviced by a master.
`begintransfer`	1	out	no	Asserted during the first bus cycle of each new Avalon bus transfer. Usage is peripheral-specific.

Like the non-tristate Avalon interface, all signal types are available active-low by appending "`_n`", such as `chipselect_n` and `write_n`.

The Avalon tristate interface uses the bidirectional port `data` instead of separate `readdata` and `writedata` ports. During write transfers the `data` port is driven by the Avalon bus module, and the slave device captures `data`. During read transfers, the slave device drives the `data` port, and the Avalon bus module captures `data`. The `data` port is bidirectional, slave peripherals (and the Avalon bus module) must drive the `data` lines only at specific times.

The Avalon tristate interface introduces the notion of shared ports. Sharing ports is useful to reduce the number of I/O pins required to interface the Avalon bus to external devices. The PTF assignment `Is_Shared` is used to declare that a port type be shared. A shared port of a given type can be connected to (shared by) multiple off-chip slave devices. The data port is always shared. For example, when the Avalon bus module `data` is connected to I/O pins on the PLD, these `data` pins may connect to several off-chip peripheral devices. Other ports can be shared optionally, such as `address`, `read`, and `write`. If one or more other peripherals use a port of the same type and the other peripherals' port is also declared to be shared, then SOPC Builder multiplexes the shared ports onto the same device I/O pins. An Avalon tristate slave peripheral must respond to shared signals only at specific times, defined by `chipselect` and `outputenable`.

Peripherals using the Avalon tristate interface must use the `chipselect` port. An off-chip slave peripheral must accept transfers only when its `chipselect` is asserted. `chipselect` is never a shared signal, and a unique `chipselect` signal is driven to each off-chip peripheral.

The Avalon tristate interface introduces the signal type `outputenable` for slave read transfers. To avoid signal contention on the `data` lines, off-chip slave peripherals must drive their `data` output pins only when `outputenable` is asserted. `outputenable` is used mainly for off-chip memory devices with latency, such as SSRAM that drive the `data` lines several clock cycles after a read transfer is initiated.

Avalon Tristate Slave Read Transfer without Latency

The Avalon tristate slave read transfer without latency is most commonly used when interfacing asynchronous off-chip memory devices, such as SRAM and flash, to the Avalon bus module. The signals and timing for the fundamental Avalon tristate slave read transfer is nearly identical to the (non-tristate) fundamental Avalon slave read transfer. The only difference is the behavior of the bidirectional `data` port. The slave device can drive its `data` lines only when `outputenable` is asserted. At all other times, the slave must tristate `data`. Fixed setup and wait states—including peripheral-controlled wait states—are supported, and the timing is identical to the non-tristate cases.

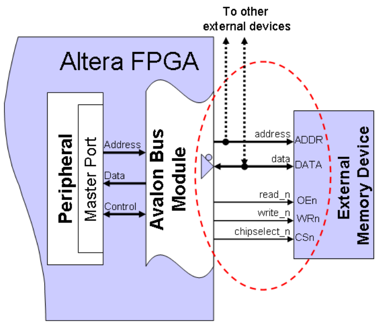
Most board designs connect the Avalon `chipselect_n` signal directly to the chip select pin (e.g., `CSn`) on an external memory device, and connect the Avalon `read_n` signal directly to the output enable pin (e.g., `OEn`). The Avalon signal `outputenable_n` could also be used to drive the output enable pin (e.g., `OEn`) on an external device. However, for slave transfers with no latency, the `outputenable_n` signal is identical to `read_n`.

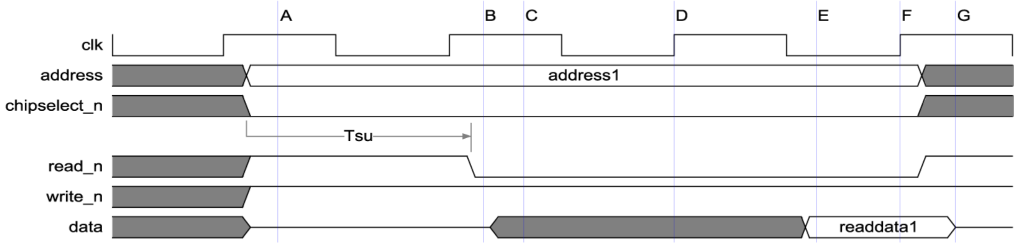
Some memory devices have a combined `R/Wn` (i.e., read when high, write when low) pin. The Avalon tristate `write_n` signal behaves in this manner, and can be connected to a `R/Wn` pin. `write_n` is only asserted during write transfers, and remains deasserted (i.e., “read”) at all other times.

Example 20 shows an Avalon tristate slave read transfer using a fixed setup time and fixed wait states. In this example, the `address` and bidirectional `data` ports are shared. The diagram shows the tristate behavior for a specific peripheral’s `data` port, however, the data lines may be active at any given 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. Active-low logic has been chosen for `read_n`, `chipselect_n` and `write_n` to reflect conventions used by most external memory devices. `clk` is shown for timing reference only.

Example 20. Tristate Slave Read Transfers with Fixed Setup Time & Wait States

This Example Demonstrates	*Relevant PTF Parameters*
Avalon tristate slave read transfer from asynchronous off-chip peripheral	Bus_Type = “avalon_tristate”
1 bus cycle of setup time	Setup_Time = “1”
1 fixed wait state	Read_Wait_States = “1”





Example 20 Time Reference Description:

(A) The Avalon bus module drives `address` and asserts `chipselect_n`.

(B) After one bus cycle of setup delay, the Avalon bus module asserts `read_n` (equivalent to `outputenable_n` in this example).

(C) The slave peripheral drives `data` in response to `read_n`. `data` may or may not be valid at this point. In this example, it is undefined.)

(D) The Avalon bus module keeps `address` asserted through one bus cycle of wait state.

(E) The slave peripheral drives valid data some time before the final rising clock edge of the transfer.

(F) The Avalon bus module captures `data` at this rising edge of `clk`, and the transfer ends.

(G) The slave peripheral tristates its `data` port in response to `read_n` (now deasserted).

Avalon Tristate Slave Read Transfer with Fixed Latency

The Avalon tristate slave read transfer with latency is most commonly used when connecting off-chip synchronous memory devices, such as SSRAM and ZBT SRAM, to the Avalon bus module. Variable latency is not supported for off-chip devices.

The signals and timing for the Avalon tristate slave read transfer with fixed latency is nearly identical to the (non-tristate) Avalon slave read transfer with fixed latency. The only difference is the behavior of the bidirectional `data` port. The slave device can drive its `data` lines only when `outputenable` is asserted. At all other times, `data` must be tristated. Because valid data returns after several cycles of latency, `outputenable` is asserted after the address phase of the transfer. Fixed setup, hold, and wait states are supported, and the timing is identical to the non-tristate cases.

Most board designs connect the Avalon `chipselect_n` signal directly to the chip select or chip enable pin (e.g., `CSn` or `CEn`) on an external memory device. Some synchronous memory devices require a chip select signal to be asserted only during the address phase, while other devices require the chip select to be asserted until the entire transfer completes. The Avalon tristate interface accommodates this variance with the PTF assignment `Active_CS_Through_Read_Latency`. When `Active_CS_Through_Read_Latency` = `1`, `chipselect` is asserted during the entire read transfer. In this case, `chipselect` mirrors the `outputenable` signal. When `Active_CS_Through_Read_Latency` = `0`, `chipselect` is asserted only during the address phase. In this case, `chipselect` mirrors the `read` signal.

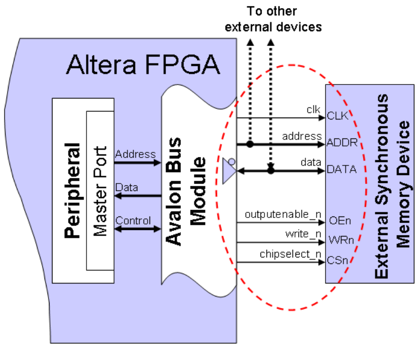
Some synchronous memory devices have a combined `R/Wn` (i.e., read when high, write when low) pin. The Avalon tristate signal `write_n` behaves in this manner, and can be connected directly to a `R/Wn` pin. `write_n` is asserted only during write transfers, and remains deasserted (i.e., in read mode) at all other times.

Unlike Avalon tristate read transfers without latency, the `outputenable` signal is not identical to the `read` signal. For read transfers with latency, `read` is asserted during the address phase, but is deasserted through the data phase. Later, `outputenable` is asserted before the final rising clock edge of the transfer, causing the peripheral device to drive its `data` pins. After one transfer completes, `outputenable` may remain asserted as data returns from further pending read transfers. `outputenable` is deasserted when there are no more pending read transfers. Even after `outputenable` is deasserted, the `data` lines may be active with signals for a write transfer, or with signals intended for 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.

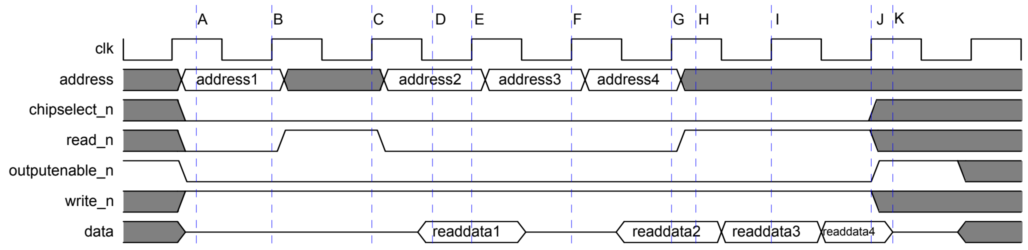
Example 21 shows several Avalon tristate slave read transfers with a fixed latency of 2 clock cycles. In this example, the address and bidirectional data ports are shared. The signals `outputenable_n`, `chipselect_n`, and `write_n` are active low to reflect the conventions used by most external memory devices. `write_n` is used as a `R/Wn` mode select pin and is shown for reference; it remains deasserted throughout the operation.

Example 21. Tristate Slave Read Transfers with Fixed 2-Clock Cycle Latency (Part 1 of 2)

This Example Demonstrates	***Relevant PTF Parameters***
Avalon tristate slave read transfer from synchronous off-chip peripheral	Bus_Type = “avalon_tristate”
Read latency of 2 clock cycles	Read_Latency = “2”
`chipselect` must be asserted throughout the complete read transfer	Active_CS_Through_Read_Latency = “1”
No setup time	Setup_Time = “0”
No wait states	Read_Wait_States = “0”



Example 21. Tristate Slave Read Transfers with Fixed 2-Clock Cycle Latency (Part 2 of 2)



Example 21 Time Reference Description

(A) The Avalon bus module asserts `chipselect_n`, `address`, and `read_n`, initiating read transfer 1. At this time `outputenable_n` is also asserted, i.e., the slave device is free to drive the `data` lines at any time. In this example, the device does not drive `data`, 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 bus cycle. `chipselect` remains asserted because of the PTF setting `Active_CS_Through_Read_Latency = "1"`, 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 Avalon bus module captures `readdata1` at this rising edge of `clk`. The Avalon bus module asserts `chipselect_n`, `address`, and `read_n`, initiating transfer 2.

(F) The Avalon bus module 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 bus 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 remain tristated.

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

(H) The Avalon bus module deasserts `read_n` ending the sequence of read transfers. `chipselect` remains asserted until all pending read transfers have completed.

(I) The Avalon bus module captures `readdata3` at this rising edge of `clk`.

(J) The Avalon bus module captures `readdata4` at this rising edge of `clk`.

(K) There are no more pending transfers, and the Avalon bus module deasserts `outputenable_n`, which forces the slave device to tristate its `data` lines.

Avalon Tristate Slave Write Transfer

The Avalon tristate slave write transfer is used when connecting the Avalon bus module to off-chip writeable memory devices, such as SRAM, SSRAM, and flash. The signals and timing for the fundamental Avalon tristate slave write transfer are nearly identical to the non-tristate fundamental Avalon slave write transfer. The only difference is the usage of the bidirectional `data` port instead of the `writedata` input port. Fixed setup, hold, and wait states are supported, as well as peripheral-controlled wait states, and the timing is identical to the non-tristate cases. Like non-tristate Avalon write transfers, there is no write transfer with latency mode.

Even if a slave peripheral does not perform a transfer, the `data` port (and other shared ports) may be driven by unrelated peripherals at any given time. Therefore, the off-chip device must capture `data` only when `chipselect` is asserted, and the device must never drive `data` during a write transfer.

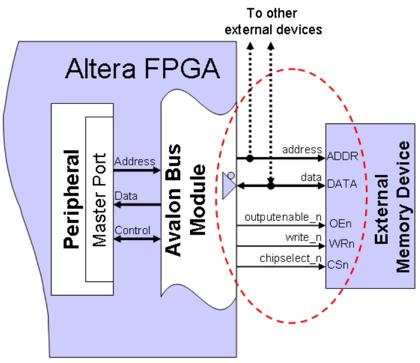
Board designs can connect the Avalon `write_n` signal directly to a write enable pin (e.g., `WEn`). Some synchronous memory devices have a combined `R/Wn` (i.e., read when high, write when low) pin. The Avalon tristate signal `write_n` behaves in this manner, and can be connected directly to a `R/Wn` pin. `write_n` is asserted only during write transfers, and remains deasserted (i.e., read mode) at all other times. In addition, some synchronous memory devices use byte-enable signals during write transfers to specify which byte lanes to write (e.g., `BWn1`, `BWn2`, `BWn3`, and `BWn4`). The Avalon port `writebyteenable` is the logical AND of the `write` and `byteenable` signals, and can be connected directly to such `BWn` pins.

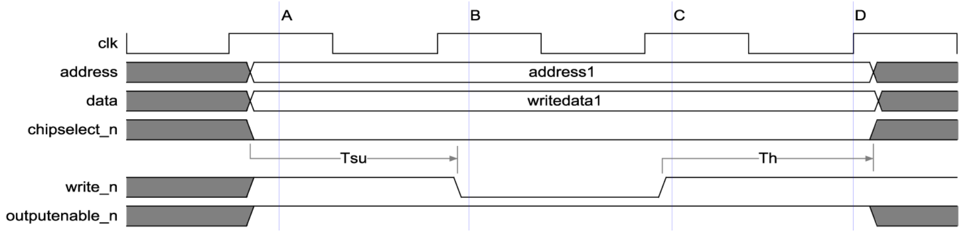
The Avalon tristate interface does not support latent write transfers. However, Avalon tristate slave write transfers can write data successfully to off-chip synchronous memory devices, such as SSRAM and ZBT RAM. For example, hold states can be used to keep `data` asserted several clock cycles after `write` is deasserted. The Avalon bus module waits for any pending read transfers with latency to complete before initiating a new write transfer. This prevents any possible signal contention on the `data` lines due to latent read data colliding with write data. As a result, the Avalon tristate interface may not achieve the maximum possible bandwidth for synchronous memory devices when performing back-to-back read-write transfer sequences. However, the most commonly required high-bandwidth cases of continuous back-to-back read transfers with latency or continuous back-to-back write transfers are supported.

Example 22 on page 87 shows an Avalon tristate slave write transfer using fixed setup time and fixed hold time. In this example, the `address` and bidirectional `data` ports are shared. The signals `write_n`, `chipselect_n`, and `outputenable_n` are active low to reflect conventions used by most external memory devices. `outputenable_n` is shown for reference. `outputenable_n` is deasserted, and the peripheral must never drive its `data` lines throughout the write transfer. `clk` is shown for timing reference only.

Example 22. Avalon Tristate Slave Write Transfer with Fixed Setup & Hold Time

This Example Demonstrates	*Relevant PTF Parameters*
Avalon tristate slave write transfer to asynchronous off-chip peripheral	Bus_Type “avalon_tristate”
1 bus cycle of setup time	Setup_Time = “1”
1 bus cycle of hold time	Hold_Time = “1”





Example 22 Time Reference Description

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

Avalon Bus Address Alignment Options

The following sections describe the Avalon bus address alignment options.

Address Alignment Overview

The Avalon bus module accommodates master and slave peripherals of varying, unmatched data widths. For example, 32-bit master ports can access 8-bit slave ports, and 16-bit master ports can access 32-bit slave ports. Whenever master-slave pairs of unmatched data widths exist together in a system, the issue of address alignment comes up. This situation is not specific to the Avalon bus; the issue arises for all microprocessor systems.

In the discussion of data transfers between master and slave ports of differing data widths, it is necessary to make the distinction of which peripheral has the wider data port. The following discussions describes the master and slave ports in a master-slave pair as being *wide* or *narrow* to indicate that one port has more or fewer data bits than the other.

In the case of a wide master accessing a narrow slave port, the question becomes: What happens to the most significant bits (MSBs) when a wide master reads from (or writes to) a narrow slave? The Avalon bus offers two approaches to handling this situation:

- *Native Address Alignment*—With native address alignment, a single transfer on the master port corresponds to exactly one transfer on the slave port. For example, when a 32-bit master reads from a 16-bit slave port, the Avalon bus module returns a 32-bit unit of data, but only the least significant 16 bits contain valid data from the slave port. The MSBs may be zero or undefined. This is the "master knows best" scenario, typical to many embedded systems. However, the software or hardware design that controls the master port must be aware of the physical data widths and addressing schemes of all relevant slaves. This adds complexity to the design of the master peripheral.

- *Dynamic Bus Sizing*—With dynamic bus sizing, when a wide master reads from a narrow slave port, the slave side of the Avalon bus module performs several read transfers—as many as required to fill the master data width with narrow slave units of data. For example, when a 32-bit master reads from an 8-bit slave, the Avalon bus module returns a 32-bit word filled with four valid bytes of data from the slave. Dynamic bus sizing abstracts the physical details of the slave port, and enables each master to perform data transfers as if the slave were always the same width as the master. Dynamic bus sizing simplifies software design for the master port, by eliminating the need for software to splice together data from a narrow slave peripheral.

☞ In general —memory peripherals use dynamic bus sizing— all other peripherals use native address alignment.

There are also cases in which a narrower (i.e., 16-bit) master port connects to a wider (i.e., 32-bit) slave port. The Avalon bus module also accommodates these cases. The Avalon bus module automatically fetches a full 32-bit word from the slave port, and presents the appropriate half-word to the 16-bit master. The logic required to multiplex the 32-bit data is always integrated into the Avalon bus module in the event that a 16-bit master port connects to a wider slave port.

"Choosing the Address Alignment Assignment for Avalon Peripherals" on page 89 describes the considerations for assigning native or dynamic bus sizing. Later sections describe how a master perceives a slave in both the native and dynamic cases. "Connection to External Devices" on page 100 describes the physical design considerations of how to connect the address and data ports on a peripheral to the system module.

Choosing the Address Alignment Assignment for Avalon Peripherals

Address alignment assignments are declared in the system PTF file. Each slave port connected to the Avalon bus module is assigned an address alignment setting of either native or dynamic. Address alignment assignments are not made for master ports. Masters always receive master-width units of data; the address alignment of the slave port determines how the master perceives this data.

Peripherals used as program or data memory should be assigned dynamic bus sizing. From a system-level perspective, dynamic bus sizing offers three benefits:

- 32-bit and 16-bit processors can use inexpensive 8-bit or 16-bit memory for data and instruction storage. Without dynamic bus sizing, it would be impossible for a processor to execute code from a memory that is narrower than the instruction width.

- The physical width of memory is transparent to the software.

- Software takes fewer instructions and executes faster, because software does not have to perform any read-and-shift operations to patch together wider units of data.

In the event that a microprocessor needs to access only a single byte in memory, the processor can use byte or half-word operations to read or store to the appropriate byte. In most cases, however, the processor wishes to transfer a full-width unit of data. There are few, if any, scenarios in which a processor would benefit from not having dynamic bus sizing.

Native address alignment is appropriate for all other types of slave peripheral. Dynamic bus sizing is not suitable for slave peripherals that are controlled by registers mapped into memory space. The operation of the peripheral is directly affected by read or write transfers to specific control registers. A processor generally accesses a peripheral’s control registers one register at a time. It is desirable for the processor to have complete control over read and write transfers to individual registers, without incidentally accessing unrelated registers in incremental address space.

There may be cases in which a peripheral contains both control registers and memory space. In these rare cases, there are two solutions for assigning the appropriate address alignment. First, in reality, such a peripheral is probably designed specially for a specific embedded system, in which case the designer should design the peripheral to match the master port’s data width. If the slave port is not narrower than the master, address alignment is not an issue. Second, if the peripheral design already exists and cannot be redesigned to match the master width, then interface logic should be designed to incorporate two Avalon slave interfaces. One slave port with native address alignment would then address the peripheral’s register space, and the other slave port with dynamic bus sizing would address the memory space.

Native Address Alignment: 32-Bit Master Port

The following discussion defines how a 32-bit master port perceives data in narrower slave peripherals. Separate discussions are given for slave peripherals of 1 to 8, 9 to 16, and 16 to 32 bits.

Slave Port Between 1 & 8 Bits

In the case of a 32-bit master transferring data to an 8-bit slave peripheral, only the least significant 8 bits of the 32-bit word is valid data, but the unused upper 24 bits also consume address space.

Consider a hypothetical 8-bit slave peripheral connected via the Avalon bus module to a 32-bit master, a Nios processor. The example peripheral has 5 internal 8-bit registers, as shown in Table 7.

Table 7. Example 8-bit Slave Peripheral with Five Registers

Register Internal Address	Register Name
0	aa
1	bb
2	cc
3	dd
4	ee

Suppose this slave peripheral is assigned native address alignment, and it is mapped to some base address "BASE". The result of a 32-bit master reading from this 8-bit peripheral with native alignment is shown in Table 8.

Table 8. 32-Bit Transfer to 8-Bit Peripheral with Native Alignment

Master Address	Master Perceives Data As (1)
BASE+ 0x00	0xuu uu uu aa
BASE+ 0x04	0xuu uu uu bb
BASE+ 0x08	0xuu uu uu cc
BASE+ 0x0C	0xuu uu uu dd
BASE+ 0x10	0xuu uu uu ee

Note to Table 8:
(1) In this table, uu means undefined.

In the event of a read transfer, the valid data from any native-aligned 8-bit (or narrower) peripheral appears in the least significant bits (LSBs) of the 32-bit value presented to the master. The higher-order bits are undefined. In the event of a 32-bit write transfer to a narrow peripheral, data in the upper bits will be ignored. For example, if a 32-bit master writes the value 0xFEDCBA98 to a 4-bit slave, the value written to the slave is 0x8.

Slave Port Between 9 & 16 Bits

In the case of a 32-bit master transferring data to a peripheral between 9 and 16-bits, the least significant bits of the 32-bit word are valid data, but the higher-order bits also consume address space.

Consider a hypothetical 16-bit slave peripheral connected via the Avalon bus module to a 32-bit master, a Nios processor. The example peripheral has 5 internal 16-bit registers, as shown in Table 9.

Table 9. Example 16-Bit Slave Peripheral with Five Registers

Peripheral Internal Address	16-Bit Register Name
0	aaaa
1	bbbb
2	cccc
3	dddd
4	eeee

Suppose this slave peripheral is assigned native address alignment, and it is mapped to some base address BASE. The result of a 32-bit master reading from this 16-bit peripheral with native alignment is shown in Table 10.

Table 10. 32-Bit Transfer to 16-Bit Peripheral with Native Alignment

Master Address	Master Perceives Data As *(1)*
BASE+ 0x00	0x uu uu aaaa
BASE+ 0x04	0x uu uu bbbb
BASE+ 0x08	0x uu uu cccc
BASE+ 0x0C	0x uu uu dddd
BASE+ 0x10	0x uu uu eeee

Note to Table 10:
(1) In this table, uu means undefined.

In the event of a read transfer, the valid data from any native-aligned 16-bit (or narrower) peripheral appears in the LSBs of the 32-bit value presented to the master. The higher-order bits are undefined.

In the event of a 32-bit write transfer to a narrow peripheral, data in the upper bits is ignored. For example, if a 32-bit master writes the value 0xFEDCBA98 to a 12-bit slave, the value written to the slave is 0xA98.

Slave Port Between 17 & 31 Bits

In the case of a 32-bit master transferring data to a native-aligned peripheral between 17 and 31 bits, the least significant bits of the 32-bit word are valid data. If the slave is a full 32-bits wide, then it is not narrower than the master port, and address alignment has no affect on the consideration of the peripheral's address space.

In the event of a read transfer, the valid data from any native-aligned 31-bit (or narrower) peripheral appears in the LSBs of the 32-bit value presented to the master. The higher-order bits are undefined.

In the event of a 32-bit write transfer to a narrow peripheral, data in the upper bits will be ignored. For example, if a 32-bit master writes the value 0xFEDCBA98 to a 24-bit slave, the value written to the slave is 0xDCBA98.

Native Address Alignment: 16-Bit Master Port

The following discussion defines how a 16-bit master port perceives data in narrower slave peripherals. Separate discussions are given for slave peripherals of 1 to 8 and 9 to 16 bits.

Slave Port Between 1 & 8 Bits

In the case of a 16-bit master transferring data to an 8-bit slave peripheral, only the least significant 8 bits of the 16-bit half-word is valid data, but the unused upper 8 bits also consume address space.

Consider again the hypothetical 8-bit slave peripheral (see Table 7 on page 91) connected via the Avalon bus module to a 16-bit master. Suppose this slave peripheral is assigned native address alignment, and it is mapped to some base address "BASE." The result of the 16-bit master reading from this 8-bit peripheral with native alignment is shown in Table 11 on page 94.

Table 11. 16-Bit Transfers to 8-Bit Peripheral with Native Alignment

Master Address	Master Perceives Data As (1)
BASE+ 0x00	0xuu aa
BASE+ 0x02	0xuu bb
BASE+ 0x04	0xuu cc
BASE+ 0x06	0xuu dd
BASE+ 0x08	0xuu ee

Note:
(1) In this table, uu means undefined.

In the event of a read transfer, the valid data from any native-aligned 8-bit (or narrower) peripheral appears in the LSBs of the 16-bit value presented to the master. The higher-order bits are undefined.

In the event of a 16-bit write transfer to a narrow peripheral, data in the upper bits will be ignored. For example, if a 16-bit master writes the value 0xBA98 to a 4-bit slave, the value written to the slave is 0x8.

Slave Port Between 9 & 16 Bits

In the case of a 16-bit master transferring data to a native-aligned slave port between 9 and 15 bits, the least significant bits of the 16-bit half-word are valid data. If the slave port is 16-bits wide, then it is not narrower than the master port, and address alignment has no affect on the consideration of the peripheral's address space.

In the event of a read transfer, the valid data from any native-aligned 15-bit (or narrower) peripheral appears in the least significant bits (LSBs) of the 16-bit value presented to the master. The higher-order bits are undefined. In the event of a 16-bit write transfer to a narrow peripheral, data in the upper bits will be ignored. For example, if a 16-bit master writes the value 0xBA98 to a 12-bit slave, the value written to the slave is 0xA98.

Native Alignment Considerations in Multi-Master System Modules

Multiple master peripherals may connect to the Avalon bus module. The address alignment of the slave ports has little or no effect on the simultaneous multi-master behavior of the Avalon bus.

It may occur to a designer to consider how address space is perceived from the perspective of the multiple masters. Consider the case of two 32-bit master ports that can address a common 16-bit slave port with native address alignment. In the case of master ports of identical widths, both masters perceive the address space identically, and no special considerations are necessary.

Now consider the case of two masters, a 32-bit master and a 16-bit master that both address the hypothetical 16-bit peripheral (see Table 9) mapped at address BASE. In almost all cases with multiple masters of different widths connected to the Avalon bus module, no special considerations are necessary in designing the two masters. However, this case could present a conceptual hurdle for the designer when considering how each master perceives its corresponding address space. The address spaces are shown in Table 12.

Table 12. Masters of Different Width Accessing a 16-Bit Slave Peripheral *Note (1)*

32-Bit Master Address	32-Bit Master Perceives Data	16-Bit Master Address	16-Bit Master Perceives Data
BASE+ 0x00	0x uu uu aaaa	BASE+ 0x00	0xaaaa
BASE+ 0x04	0x uu uu bbbb	BASE+ 0x02	0xbbbb
BASE+ 0x08	0x uu uu cccc	BASE+ 0x04	0xcccc
BASE+ 0x0C	0x uu uu dddd	BASE+ 0x06	0xdddd
BASE+ 0x10	0x uu uu eeee	BASE+ 0x08	0xeeee

Note: to Table 12
(1) In this table, uu means undefined.

At first sight, it may be unsettling to discover that the address space is perceived differently, depending on the width of the master peripheral. For example, the 32-bit master perceives the half-word at BASE+4 to be 0xbbbb, while the 16-bit master perceives the half-word at BASE+4 to be 0xcccc. Discussion of this memory inconsistency is more academic than practical. For several reasons, this inconsistency will not pose problems in real-world systems. The discussion can be broken into two fundamental cases:

- *The slave peripheral is a memory device*—Memory devices should be assigned dynamic, not native, address alignment. Dynamic bus sizing will make the address spaces consistent for all masters accessing the memory.

- *The slave peripheral is controlled by memory-mapped control registers*—In the case of register-controlled peripherals, it is assumed that the software (or hardware logic) that controls the master port has an understanding of how the slave peripheral works. If not, the master simply could not interface to the slave peripheral. Therefore, when there are any special usage requirements (including addressing considerations) for the slave peripheral, the peripheral driver software must be coded to handle these considerations.

In neither case will the address space inconsistency surprise an unsuspecting designer and cause erroneous addressing errors.

Dynamic Bus Sizing

When a wide master port addresses a narrow slave port that is assigned dynamic bus sizing, a single master transfer with the slave port results in multiple slave transfers to gather a full, master-width unit of valid data. As a side effect, this eliminates the presence of unusable or undefined bits in the master's perceived address space. Therefore, software does not have to work around memory that contains unusable bits.

For example, a 32-bit master read transfer from an 8-bit slave memory results in four slave read transfers from the 8-bit memory. The Avalon bus module mediates between the master port and the slave port, so each peripheral sees standard Avalon read transfers. The master port perceives a memory peripheral that—after several wait states—returns a full 32-bits of data every transfer. The slave port perceives four separate read transfers. Logic internal to the Avalon bus module forces the master port to wait during the four slave reads, and then presents the combined 32-bit result to the master port all at once.

The following discussion defines how a wide master port perceives data in a narrower slave peripheral with dynamic bus sizing. This comprises two cases: The case of a 32-bit or 16-bit master port accessing an 8-bit slave port, and the case of a 32-bit master port accessing a 16-bit slave port. Dynamic bus sizing is conceptually identical in the 32-bit master and 16-bit master scenarios, so these scenarios are treated together.

Dynamic bus sizing is used for program data and instruction memory, which tends to come in standard sizes of 8, 16, and 32 bits. Therefore, this discussion focuses solely on these practical cases.

☞ Narrow dynamic writes to peripherals with no byte enables (for example, a Nios processor doing an 8-bit write to an EPXA DPRAM PLD interface) results in undetermined system behavior.

8-bit Slave Port with Dynamic Bus Sizing

Consider an 8-bit slave peripheral like the one shown in Table 7 on page 91, except this time imagine the peripheral has 10 locations, aa to jj. Suppose this slave peripheral is now assigned dynamic bus sizing, and is mapped to some base address BASE. For each 32-bit master transfer, the Avalon bus module performs four slave transfers to four sequential locations on the slave port. Likewise, for each 16-bit master transfer, the Avalon bus module performs two slave transfers to two sequential locations on the slave port. From the perspective of the master and the slave, the nature of the transfers is no different from normal Avalon transfers.

The result of reading from this peripheral with dynamic bus sizing is shown in Table 13.

Table 13. 32-Bit Master Transfer & 16-Bit Transfer with 16-Bit Slave Peripheral with Dynamic Bus Sizing *Note (1)*

32-Bit Master Address	32-Bit Master Perceives Data	16-Bit Master Address	16-Bit Master Perceives Data
BASE+ 0x00	0x dd cc bb aa	BASE+ 0x00	0x bb aa
		BASE+ 0x02	0x dd cc
BASE+ 0x04	0x hh gg ff ee	BASE+ 0x04	0x ff ee
		BASE+ 0x06	0x hh gg
BASE+ 0x08	0x uu uu jj ii	BASE+ 0x08	0x jj ii

Note: to Table 13
(1) In this table, uu means undefined.

In most realistic cases, 8-bit memory peripherals will not end at an uneven word boundary, so an undefined value from a dynamically address-aligned slave is rare.

☞ Note that both the 32-bit master and the 16-bit master perceive their address spaces identically; there is no inconsistency as in the native address alignment case. For example, both masters perceive the half-word at BASE+2 to be 0xddcc.

There is no way for the 32-bit master to read from only one 16-bit location, such as only aa, or only ee. A master read transfer will always cause multiple slave read transfers to sequential addresses in the slave's address space. This is the reason why dynamic bus sizing is poorly suited to register-controlled slave peripherals.

Discussion of transfers with a memory that is narrower than 8 bits may be of questionable practical value, but the functionality of dynamic bus sizing is well defined. For a read transfer, the Avalon bus module captures 8 bits for every slave read transfer. The bits that do not exist on the slave port are undefined. The master peripheral must be aware of how to work around these undefined bits. For a write transfer, the bits in the 16- or 32-bit word that correspond to nonexistent bits in the slave peripheral are simply ignored.

Dynamic bus sizing affects write transfers differently, depending on the size of the data unit to be written. The master port indicates to the Avalon bus module the byte locations it wishes to write, by using the byte enable (`byteenable_n`) outputs. There is one byte enable line for each byte lane in the master data port. According to `byteenable_n`, the Avalon bus module initiates as many slave write transfers as necessary to write the appropriate bytes into the 8-bit slave memory.

16-bit Slave Port with Dynamic Bus Sizing

Consider a 16-bit slave peripheral like the one shown in Table 9 above, with five locations, aaaa to eeee. Suppose this slave peripheral is now assigned dynamic bus sizing, and is mapped to some base address BASE. For each 32-bit master transfer, the Avalon bus module performs two slave transfers to two sequential locations on the slave port. From the perspective of the master and slave ports, the nature of the transfers is no different from normal Avalon transfers. The case of a 16-bit master is no longer relevant, because the master and slave ports match in width. The result of reading from this peripheral with dynamic bus sizing is shown in Table 14.

Table 14. 32-Bit Master Transfer & 16-Bit Masters Transfer with 16-Bit Slave Peripheral with Dynamic Bus Sizing

32-Bit Master Address	32-Bit Master Perceives Data	16-Bit Master Address	16-Bit Master Perceives Data
BASE+ 0x00	0x bbbb aaaa	BASE+ 0x00	0x aaaa
		BASE+ 0x02	0x bbbb
BASE+ 0x04	0x dddd cccc	BASE+ 0x04	0x cccc
		BASE+ 0x06	0x dddd
BASE+ 0x08	0x uuuu eeee	BASE+ 0x08	0x eeee

In this table uuuu means the value is undefined. In most realistic cases, 16-bit memory peripherals will not end at an uneven word boundary, so an undefined value from a dynamically address-aligned slave is rare.

☞ Both the 32-bit master and the 16-bit master perceive their address spaces identically; there is no inconsistency as in the native address alignment case. For example, both masters perceive the half-word at BASE+2 to be 0xbbbb.

There is no way for the 32-bit master to read from only one 16-bit location, such as only aaaa, or only dddd. A master read transfer will always cause multiple slave read transfers to sequential addresses in the slave's address space. This is the reason why dynamic bus sizing is poorly suited to register-controlled slave peripherals.

Discussion of transfers with a memory that is between 8 and 16 bits may be of questionable practical value, but the functionality of dynamic bus sizing is well defined. For a read transfer, the Avalon bus module captures 16 bits for every slave read transfer. The bits that do not exist on the slave port are undefined. The master peripheral must be aware of how to work around these undefined bits. For a write transfer, the bits in the 32-bit word that correspond to nonexistent bits in the slave peripheral are simply ignored.

Dynamic bus sizing affects write transfers differently, depending on the size of the data unit to be written. The master port indicates to the Avalon bus module the byte locations it wishes to write by using the byte enable (`byteenable_n`) outputs. There is one byte enable line for each byte lane in the master data port. According to `byteenable_n`, the Avalon bus module initiates as many slave write transfers as necessary to write the appropriate bytes into the 16-bit slave memory.

32-Bit Slave Port with Dynamic Bus Sizing

If the slave port is 32-bits wide, then its width matches the width of a 32-bit master port, and no addressing considerations are necessary.

In the case that the slave port is between 17 and 31 bits wide, dynamic bus sizing behaves exactly the same as native address alignment

Connection to External Devices

For systems using only peripherals inside the system module, the system designer does not have to consider the details of connecting Avalon peripherals to the Avalon bus. However, most systems require interfaces to off-chip memory devices. The system designer must manually connect peripherals outside the system module (including off-chip devices) to Avalon bus ports. Furthermore, many systems drive Avalon signals off chip via a tri-state bus so that multiple off-chip devices can be addressed through the same physical address and data pins. This connection method is much like a traditional bus architecture, with bus signals routed to physical lines on the PCB. In such systems, it is not always obvious how to connect the address pins of a peripheral outside the system module to an address port on the Avalon bus module. The situation can become especially complicated when using multiple, off-chip peripherals of both dynamic and native address alignment, and with varying bit widths.

In the following discussion, A[0] refers to the least significant address line of the slave device. Every slave device's A[0] pin is not necessarily wired to the least significant line of the Avalon `address` port. Furthermore, the connection depends on the slave's address-alignment option declared in the PTF file. Recall that memory peripherals should always use dynamic bus sizing, and that the Avalon `address` port is byte-addressable. Table 15 lists how to connect A[0] of the off-chip device to the Avalon `address` port.

Table 15. Connecting the Avalon Bus Module to External Devices

Alignment	Master Width	Slave Width	A[0] on Slave Is Connected to Byte Address Bit Number
native	32	32	2
native	32	16	2
native	32	8	2
native	16	32	-- Not Applicable --
native	16	16	1
native	16	8	1
dynamic	32	32	2
dynamic	32	16	1
dynamic	32	8	0
dynamic	16	32	2
dynamic	16	16	1
dynamic	16	8	0

When connecting narrow slave devices to a wider Avalon `data` port, the slave device's least significant data pin should always connect to the least significant bit of the Avalon `data` port.

Notes:

Index

Notes: