

Introduction

The system interconnect fabric for memory-mapped interfaces is a high-bandwidth interconnect structure for connecting components that use the Avalon® Memory-Mapped (Avalon-MM) interface. The system interconnect fabric consumes minimal logic resources, provides greater flexibility, and higher throughput than a typical shared system bus. It is a cross-connect fabric and not a tristated or time domain multiplexed bus. This chapter describes the functions of system interconnect fabric for memory-mapped interfaces and the implementation of those functions.

High-Level Description

The system interconnect fabric is the collection of interconnect and logic resources that connects Avalon-MM master and slaves on components in a system. SOPC Builder generates the system interconnect fabric to match the needs of the components in a system. The system interconnect fabric implements the connection details of a system. It guarantees that signals are routed correctly between master and slaves, as long as the ports adhere to the rules of the *Avalon Interface Specifications*. This chapter provides information on the following topics:

- “Address Decoding” on page 2-4
- “Datapath Multiplexing” on page 2-5
- “Wait State Insertion” on page 2-5
- “Pipelined Read Transfers” on page 2-6
- “Dynamic Bus Sizing and Native Address Alignment” on page 2-7
- “Arbitration for Multimaster Systems” on page 2-9
- “Burst Adapters” on page 2-13
- “Interrupts” on page 2-14
- “Reset Distribution” on page 2-16

 For details about the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

System interconnect fabric for memory-mapped interfaces supports the following items:

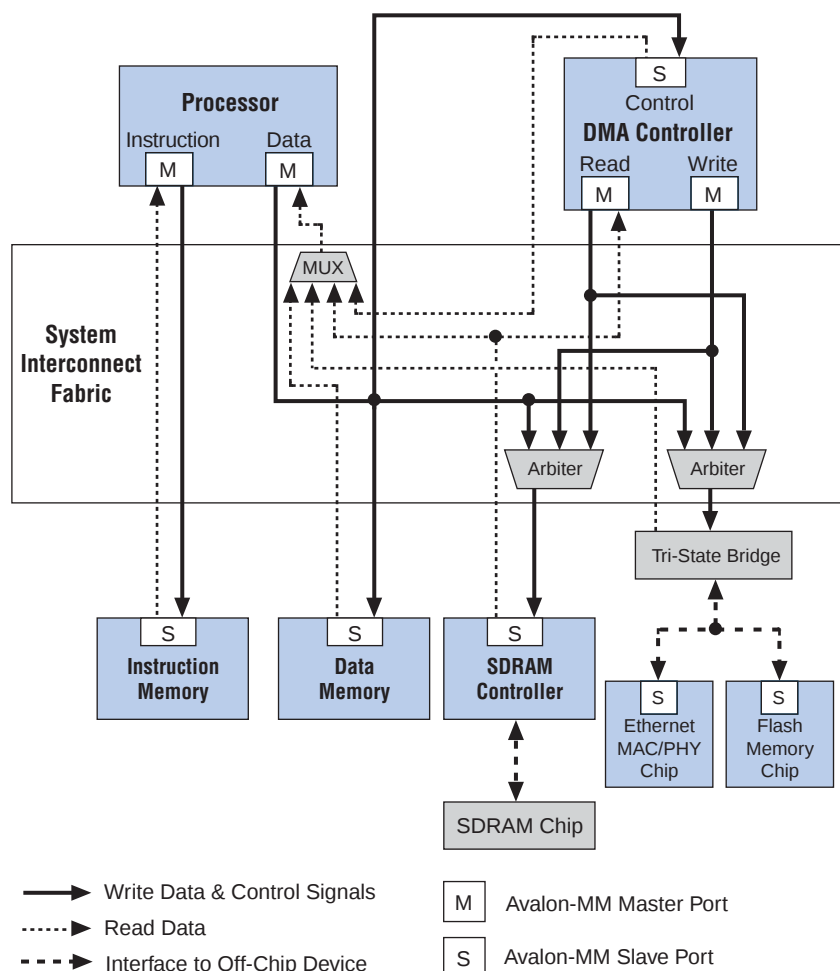
- Any number of master and slave components. The master-to-slave relationship can be one-to-one, one-to-many, many-to-one, or many-to-many.
- On-chip components.
- Interfaces to off-chip devices.
- Master and slaves of different data widths.
- Components operating in different clock domains.
- Components using multiple Avalon-MM ports.

Figure 2-1 shows a simplified diagram of the system interconnect fabric in an example memory-mapped system with multiple masters.



All figures in this chapter are simplified to show only the particular function being discussed. In a complete system, the system interconnect fabric might alter the address, data, and control paths beyond what is shown in any one particular figure.

Figure 2-1. System Interconnect Fabric—Example System



SOPC Builder supports components with multiple Avalon-MM interfaces, such as the processor component shown in Figure 2-1. Because SOPC Builder can create system interconnect fabric to connect components with multiple interfaces, you can create complex interfaces that provide more functionality than a single Avalon-MM interface. For example, you can create a component with two different Avalon-MM slaves, each with an associated interrupt interface.

System interconnect fabric can connect any combination of components, as long as each interface conforms to the *Avalon Interface Specifications*. It can, for example, connect a system comprised of only two components with unidirectional dataflow between them. Avalon-MM interfaces are suitable for random address transactions, such as to memories or embedded peripherals.

Generating system interconnect fabric is SOPC Builder's primary purpose. In most cases, you are not required to modify the generated HDL; however, a basic understanding of how HDL works can help you optimize your system. For example, knowledge of the arbitration algorithm can help designers of multimaster systems minimize the impact of arbitration on the system throughput.

Fundamentals of Implementation

System interconnect fabric for memory-mapped interfaces implements a partial crossbar interconnect structure that provides concurrent paths between master and slaves. System interconnect fabric consists of synchronous logic and routing resources inside the FPGA.

For each component interface, system interconnect fabric manages Avalon-MM transfers, interacting with signals on the connected component. Master and slave interfaces can contain different signals and the system interconnect fabric handle any adaptation necessary between them. In the path between master and slaves, the system interconnect fabric might introduce registers for timing synchronization, finite state machines for event sequencing, or nothing at all, depending on the services required by the specific interfaces.



For more information, refer to the *Avalon Memory-Mapped Design Optimizations* chapter in the *Embedded Design Handbook*.

Functions of System Interconnect Fabric

System interconnect fabric logic provides the following functions:

- "Address Decoding" on page 2-4
- "Datapath Multiplexing" on page 2-5
- "Wait State Insertion" on page 2-5
- "Pipelined Read Transfers" on page 2-6
- "Arbitration for Multimaster Systems" on page 2-9
- "Burst Adapters" on page 2-13
- "Interrupts" on page 2-14
- "Reset Distribution" on page 2-16

The behavior of these functions in a specific SOPC Builder system depends on the design of the components in the system and the settings made in SOPC Builder. The remaining sections of this chapter describe how SOPC Builder implements each function.

Address Decoding

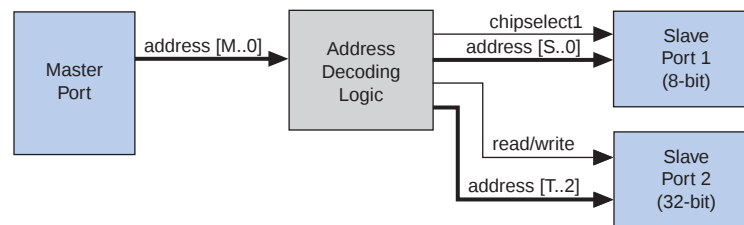
Address decoding logic in the system interconnect fabric forwards an appropriate address and produces a chipselect signal for each slave. Address decoding logic simplifies component design in the following ways:

- The system interconnect fabric selects a slave whenever it is being addressed by a master. Slave components do not need to decode the address to determine when they are selected.
- Slave addresses are properly aligned to the slave interface.
- Changing the system memory map does not involve manually editing HDL.

Figure 2-2 shows a block diagram of the address-decoding logic for one master and two slaves. Separate address-decoding logic is generated for every master in a system.

As Figure 2-2 shows, the address decoding logic handles the difference between the master address width ($\langle M \rangle$) and the individual slave address widths ($\langle S \rangle$ and $\langle T \rangle$). It also maps only the necessary master address bits to access words in each slave's address space.

Figure 2-2. Block Diagram of Address Decoding Logic



In SOPC Builder, the user-configurable aspects of address decoding logic are controlled by the **Base** setting in the list of active components on the **System Contents** tab, as shown in Figure 2-3.

Figure 2-3. Base Settings in SOPC Builder Control Address Decoding

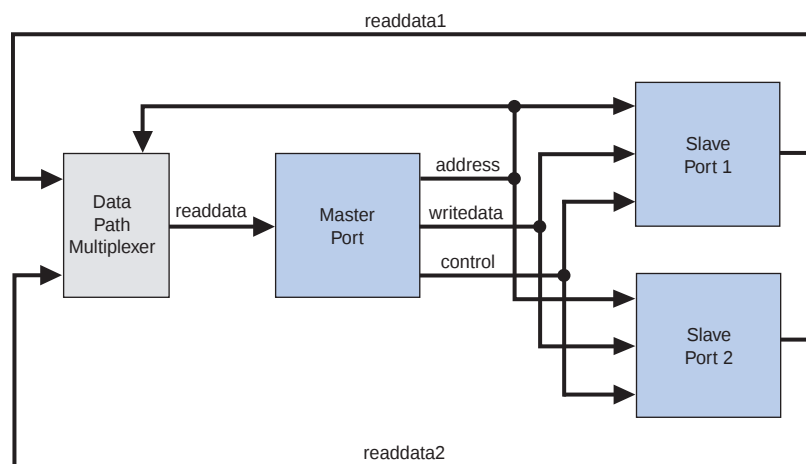
Module Name	Description	Base	End	IRQ
[-] cpu	Nios II Proces...			
instruction_master	Master port			
data_master	Master port			
jtag_debug_mod...	Slave port	0x02120000	0x021207FF	
[+] ext_flash	Flash Memory...	0x00000000	0x007FFFFFFF	
[+] ext_ram	IDT71V416 S...	0x02000000	0x020FFFFFFF	
[+] ext_ram_bus	Avalon Tri-St...			
[+] button_pio	PIO (Parallel I/O)	0x02120860	0x0212086F	2
[+] high_res_timer	Interval timer	0x02120820	0x0212083F	3

Datapath Multiplexing

Datapath multiplexing logic in the system interconnect fabric drives the `writedata` signal from the granted master to the selected slave, and the `readdata` signal from the selected slave back to the requesting master.

Figure 2-4 shows a block diagram of the datapath multiplexing logic for one master and two slaves. SOPC Builder generates separate datapath multiplexing logic for every master in the system.

Figure 2-4. Block Diagram of Datapath Multiplexing Logic

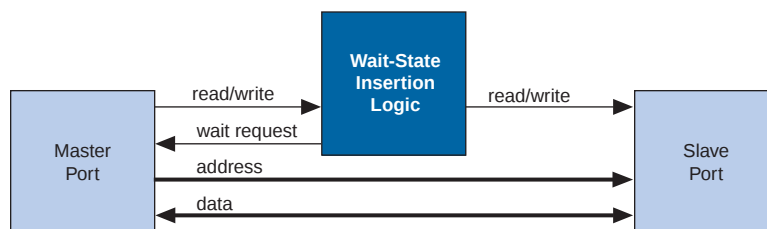


In SOPC Builder, the generation of datapath multiplexing logic is specified using the connections panel on the **System Contents** tab.

Wait State Insertion

Wait states extend the duration of a transfer by one or more cycles. Wait state insertion logic accommodates the timing needs of each slave, and causes the master to wait until the slave can proceed. System interconnect fabric inserts wait states into a transfer when the target slave cannot respond in a single clock cycle. System interconnect fabric also inserts wait states in cases when slave `read_enable` and `write_enable` signals have setup or hold time requirements.

Wait state insertion logic is a small finite-state machine that translates control signal sequencing between the slave side and the master side. Figure 2-5 shows a block diagram of the wait state insertion logic between one master and one slave.

Figure 2–5. Block Diagram of Wait State Insertion Logic

System interconnect fabric can force a master to wait for several reasons in addition to the wait state needs of a slave. For example, arbitration logic in a multimaster system can force a master to wait until it is granted access to a slave.

SOPC Builder generates wait state insertion logic based on the properties of all slaves in the system.

Pipelined Read Transfers

The Avalon-MM interface supports pipelined read transfers, allowing a pipelined master to start multiple read transfers in succession without waiting for the prior transfers to complete. Pipelined transfers allow master-slave pairs to achieve higher throughput, even though the slave requires one or more cycles of latency to return data for each transfer.

SOPC Builder generates system interconnect fabric with pipeline management logic to take advantage of pipelined components wherever possible, based on the pipeline properties of each master-slave pair in the system. Regardless of the pipeline latency of a target slave, SOPC Builder guarantees that read data arrives at each master in the order requested. Because master and slaves often have mismatched pipeline latency, system interconnect fabric often contains logic to reconcile the differences. Many cases of pipeline latency are possible, as shown in [Table 2–1](#).

Table 2–1. Various Cases of Pipeline Latency in a Master-Slave Pair

Master	Slave	Pipeline Management Logic Structure
No pipeline	No pipeline	The system interconnect fabric does not instantiate logic to handle pipeline latency.
No pipeline	Pipelined with fixed or variable latency	The system interconnect fabric forces the master to wait through any slave-side latency cycles. This master-slave pair gains no benefits of pipelining, because the master waits for each transfer to complete before beginning a new transfer. However, while the master is waiting, the slave can accept transfers from a different master.
Pipelined	No pipeline	The system interconnect fabric carries out the transfer as if neither master nor slave were pipelined, causing the master to wait until the slave returns data.
Pipelined	Pipelined with fixed latency	The system interconnect fabric allows the master to capture data at the exact clock cycle when data from the slave is valid. This process enables the master-slave pair to achieve maximum throughput performance.
Pipelined	Pipelined with variable latency	This is the simplest pipelined case, in which the slave asserts a signal when its read data is valid, and the master captures the data. This case enables this master-slave pair to achieve maximum throughput.

SOPC Builder generates logic to handle pipeline latency based on the properties of the master and slaves in the system. When configuring a system in SOPC Builder, there are no settings that directly control the pipeline management logic in the system interconnect fabric.

Dynamic Bus Sizing and Native Address Alignment

SOPC Builder generates system interconnect fabric to accommodate master and slaves with unmatched data widths. Address alignment affects how slave data is aligned in a master's address space, in the case that the master and slave data widths are different. Address alignment is a property of each slave, and can be different for each slave in a system. A slave can declare itself to use one of the following:

- Dynamic bus sizing
- Native address alignment

The following sections explain the implications of the address alignment property slave devices.

Dynamic Bus Sizing

Dynamic bus sizing hides the details of interfacing a narrow component device to a wider master, and vice versa. When an $\langle N \rangle$ -bit master accesses a slave with dynamic bus sizing, the master operates exclusively on full $\langle N \rangle$ -bit words of data, without awareness of the slave data width.



When using dynamic bus sizing, the slave data width in units of bytes must be a power of two.

Dynamic bus sizing provides the following benefits:

- Eliminates the need to create address-alignment hardware manually.
- Reduces design complexity of the master component.
- Enables any master to access any memory device, regardless of the data width.

In the case of dynamic bus sizing, the system interconnect fabric includes a small finite state machine that reconciles the difference between master and slave data widths. The behavior is different depending on whether the master data width is wider or narrower than the slave.

Wider Master

In the case of a wider master, the dynamic bus-sizing logic accepts a single, wide transfer on the master side, and then performs multiple narrow transfers on the slave side. For a data-width ratio of $\langle N \rangle:1$, the dynamic bus-sizing logic generates up to $\langle N \rangle$ slave transfers for each master transfer. The master waits while multiple slave-side transfers complete; the master transfer ends when all slave-side transfers end.

Dynamic bus-sizing logic uses the master-side byte-enable signals to generate appropriate slave transfers. The dynamic bus-sizing logic performs as many slave-side transfers as necessary to write or read the specified byte lanes.

Narrower Master

In the case of a narrower master, one transfer on the master side generates one transfer on the slave side. In this case, multiple master word addresses map to a single offset in the slave memory space. The dynamic bus-sizing logic maps each master address to a subset of byte lanes in the appropriate slave offset. All bytes of the slave memory are accessible in the master address space.

Table 2-2 demonstrates the case of a 32-bit master accessing a 64-bit slave with dynamic bus sizing. In the table, offset refers to the offset into the slave memory space.

Table 2-2. 32-Bit Master View of 64-Bit Slave with Dynamic Bus Sizing

32-bit Address	Data
0x00000000 (word 0)	OFFSET[0] _{31..0}
0x00000004 (word 1)	OFFSET[0] _{63..32}
0x00000008 (word 2)	OFFSET[1] _{31..0}
0x0000000C (word 3)	OFFSET[1] _{63..32}

In the case of a read transfer, the dynamic bus-sizing logic multiplexes the appropriate byte lanes of the slave data to the narrow master. In the case of a write transfer, the dynamic bus-sizing logic uses slave-side byte-enable signals to write only to the appropriate byte lanes.



Altera recommends that you select dynamic bus sizing whenever possible. Dynamic bus sizing offers more flexibility when the master and slave components in your system have different widths.

Native Address Alignment

Table 2-3 demonstrates native address alignment and dynamic bus sizing for a 32-bit master connected to a 16-bit slave (a 2:1 ratio). In this example, the slave is mapped to base address <BASE> in the master's address space. In Table 2-3, OFFSET refers to the offset into the 16-bit slave address space.

Table 2-3. 32-Bit Master View of 16-Bit Slave Data

32-bit Master Address	Data with Native Alignment	Data with Dynamic Bus Sizing
BASE + 0x0 (word 0)	0x0000:OFFSET[0]	OFFSET[1]:OFFSET[0]
BASE + 0x4 (word 1)	0x0000:OFFSET[1]	OFFSET[3]:OFFSET[2]
BASE + 0x8 (word 2)	0x0000:OFFSET[2]	OFFSET[5]:OFFSET[4]
BASE + 0xC (word 3)	0x0000:OFFSET[3]	OFFSET[7]:OFFSET[6]
...	...	...
BASE + 4N (word N)	0x0000:OFFSET[N]	OFFSET[2N+1]:OFFSET[2N]

SOPC Builder generates appropriate address-alignment logic based on the properties of the master and slaves in the system. When configuring a system in SOPC Builder, there are no settings that directly control the address alignment in the system interconnect fabric.

Arbitration for Multimaster Systems

System interconnect fabric supports systems with multiple master components. In a system with multiple masters, such as the system pictured in [Figure 2-1 on page 2-2](#), the system interconnect fabric provides shared access to slaves using a technique called slave-side arbitration. Slave-side arbitration moves the arbitration logic close to the slave, such that the algorithm that determines which master gains access to a specific slave in the event that multiple masters attempt to access the same slave at the same time.

The multimaster architecture used by system interconnect fabric offers the following benefits:

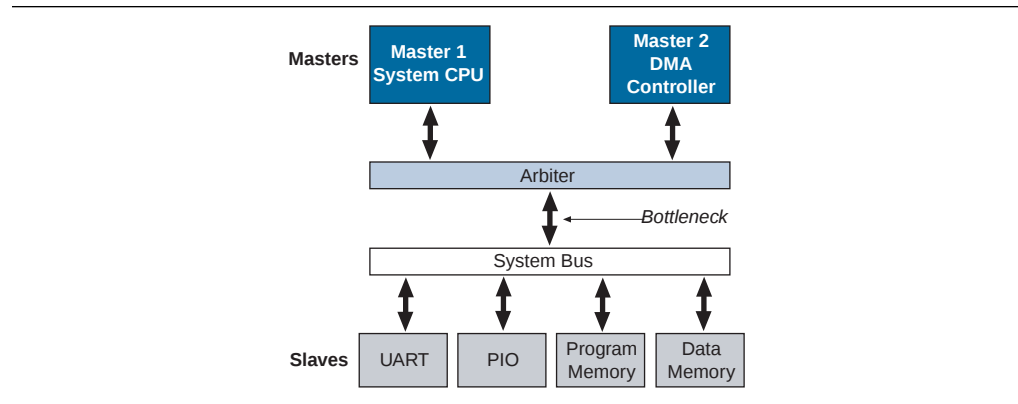
- Eliminates having to create arbitration hardware manually.
- Allows multiple masters to transfer data simultaneously. Unlike traditional host-side arbitration architectures where each master must wait until it is granted access to the shared bus, multiple Avalon-MM masters can simultaneously perform transfers with independent slaves. Arbitration logic stalls a master only when multiple masters attempt to access the same slave during the same cycle.
- Eliminates unnecessary master-slave connections. The connection between a master and a slave exists only if it is specified in SOPC Builder. If a master never initiates transfers to a specific slave, no connection is necessary, and therefore SOPC Builder does not waste logic resources to connect the two ports.
- Provides configurable arbitration settings, and arbitration for each slave is specified independently. For example, you can grant one master more arbitration shares than others, allowing it to gain more access cycles to the slave. The arbitration share settings are defined for each slave independently.
- Simplifies master component design. The details of arbitration are encapsulated inside the system interconnect fabric. Each Avalon-MM master connects to the system interconnect fabric as if it is the only master in the system. As a result, you can reuse a component in single-master and multimaster systems without requiring design changes to the component.

Traditional Shared Bus Architectures

This section discusses the architecture of the system interconnect fabric generated by SOPC Builder for multimaster systems. As a frame of reference for the discussion of multiple masters and arbitration, this section describes traditional bus architectures.

In traditional bus architectures, one or more bus masters and bus slaves connect to a shared bus, consisting of wires on a printed circuit board or on-chip routing. A single arbiter controls the bus (that is, the path between bus masters and bus slaves), so that multiple bus masters do not simultaneously drive the bus. Each bus master requests control of the bus from the arbiter, and the arbiter grants access to a single master at a time. Once a master has control of the bus, the master performs transfers with any bus slave. When multiple masters attempt to access the bus at the same time, the arbiter allocates the bus resources to a single master, forcing all other masters to wait.

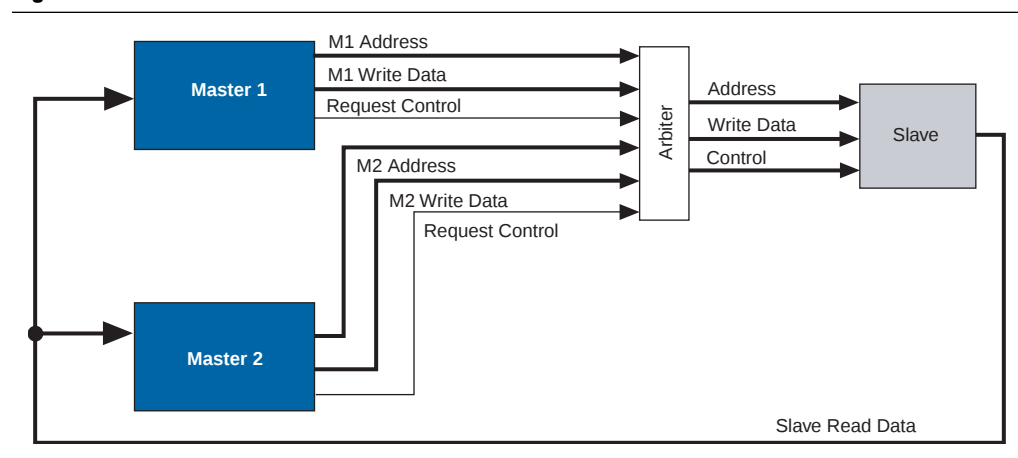
[Figure 2-6](#) illustrates the bus architecture for a traditional processor system. Access to the shared system bus becomes the bottleneck for throughput: only one master has access to the bus at a time, which means that other masters are forced to wait and only one slave can transfer data at a time.

Figure 2-6. Bus Architecture in a Traditional Microprocessor System

Slave-Side Arbitration

The system interconnect fabric uses multimaster architecture to eliminate the bottleneck for access to a shared bus. Multiple masters can be active at the same time, simultaneously transferring data with independent slaves. For example, [Figure 2-1 on page 2-2](#) demonstrates a system with two masters (a CPU and a DMA controller) sharing a slave (an SDRAM controller). Arbitration is performed at the SDRAM slave; the arbiter dictates which master gains access to the slave if both masters initiate a transfer with the slave in the same cycle.

[Figure 2-7](#) focuses on the two masters and the shared slave and shows additional detail of the data, address, and control paths. The arbiter logic multiplexes all address, data, and control signals from a master to a shared slave.

Figure 2-7. Detailed View of Multimaster Connections

Arbiter Details

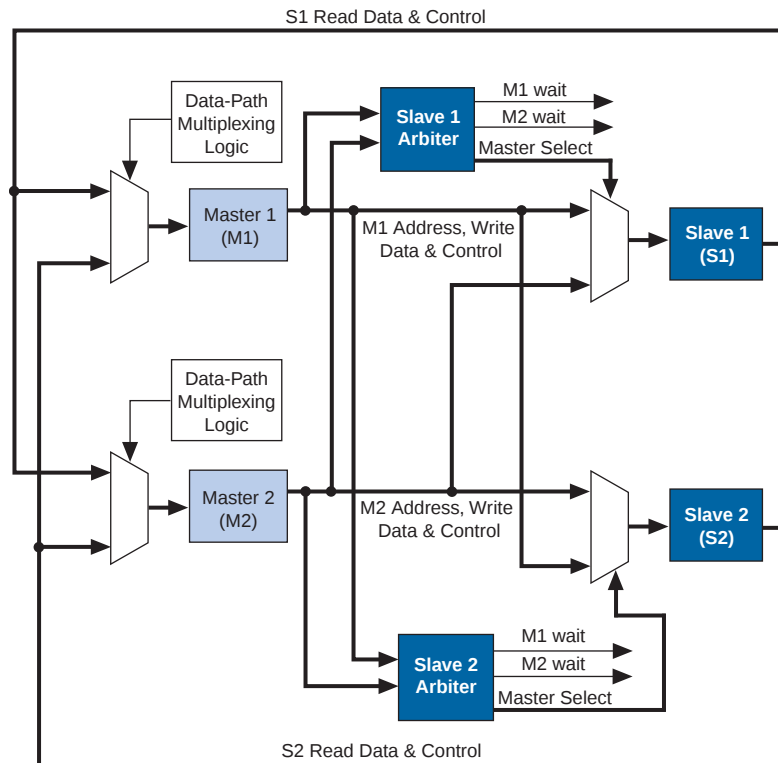
SOPC Builder generates an arbiter for every slave, based on arbitration parameters specified in SOPC Builder. The arbiter logic performs the following functions for its slave:

- Evaluates the address and control signals from each master and determines which master, if any, gains access to the slave next.

- Grants access to the chosen master and forces all other requesting masters to wait.
- Uses multiplexers to connect address, control, and datapaths between the multiple masters and the slave.

Figure 2-8 shows the arbiter logic in an example multimaster system with two masters, each connected to two slaves.

Figure 2-8. Block Diagram of Arbiter Logic



Arbitration Rules

This section describes the rules by which the arbiter grants access to masters when they contend.

Setting Arbitration Parameters in SOPC Builder

You specify the arbitration shares for each master using the connection panel on the **System Contents** tab of SOPC Builder, as shown in Figure 2-9.

Figure 2-9. Arbitration Settings on the System Contents Tab

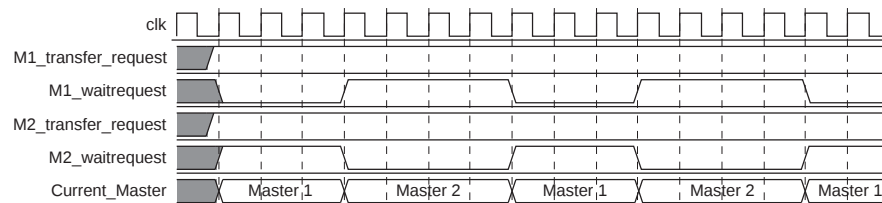
Module Name	Description	Clock
☐ cpu	Nios II Processor - Alte...	clk
☐ instruction_master	Master port	
☐ data_master	Master port	
1 ☐ jtag_debug_module	Slave port	
1 ☒ sys_clk_timer	Interval timer	clk
1 ☒ ext_ram_bus	Avalon Tri-State Bridge	clk
☒ ext_flash	Flash Memory (Commo...	
☒ ext_ram	IDT71V416 SRAM	
1 ☒ epcs_controller	EPCS Serial Flash Cont...	clk
☒ lan91c111	LAN91c111 Interface (...)	
1 ☒ jtag_uart	JTAG UART	clk

The arbitration settings are hidden by default. To see them, on the View menu, click **Show Arbitration**.

Fairness-Based Shares

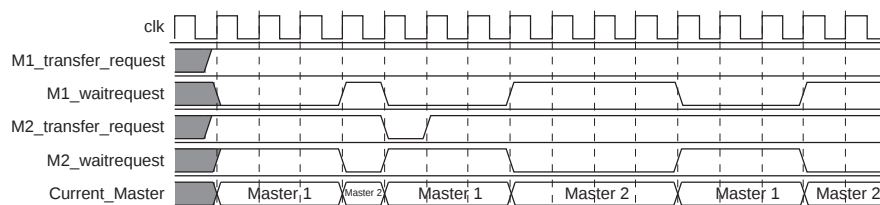
Arbiter logic uses a fairness-based arbitration scheme. In a fairness-based arbitration scheme, each master pair has an integer value of transfer *shares* with respect to a slave. One share represents permission to perform one transfer.

For example, assume that two masters continuously attempt to perform back-to-back transfers to a slave. Master 1 is assigned three shares and Master 2 is assigned four shares. In this case, the arbiter grants Master 1 access for three transfers, then Master 2 for four transfers. This cycle repeats indefinitely. [Figure 2-10](#) demonstrates this case, showing each master's transfer request output, wait request input (which is driven by the arbiter logic), and the current master with control of the slave.

Figure 2-10. Arbitration of Continuous Transfer Requests from Two Masters

If a master stops requesting transfers before it exhausts its shares, it forfeits all its remaining shares, and the arbiter grants access to another requesting master. Refer to [Figure 2-11](#). After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbiter grants access back to Master 1, which gets a replenished supply of shares.

Figure 2-11. Arbitration of Two Masters with a Gap in Transfer Requests



Round-Robin Scheduling

When multiple masters contend for access to a slave, the arbiter grants shares in round-robin order. Round-robin scheduling drives a request interface according to space available and data available credit interfaces. At every slave transfer, only requesting masters are included in the arbitration.

Burst Transfers

Avalon-MM burst transfers grant a master uninterrupted access to a slave for a specified number of transfers. The master specifies the number of transfers when it initiates the burst. Once a burst begins between a master-slave pair, arbiter logic does not allow any other master to access the slave until the burst completes. For burst masters, the size of the burst determines the number of cycles that the master has access to the slave, and the selected arbitration shares have no effect.

Burst Adapters

System interconnect fabric provides burst adaptation logic to accommodate the burst capabilities of each port in the system, including ports that do not support burst transfers. Burst adaptation logic consists of a finite state machine that translates the sequencing of address and control signals between the slave side and the master side.

The maximum burst length for each port is determined by the component design and is independent of other ports in the system. Therefore, a particular master might be capable of initiating a burst longer than a slave's maximum supported burst length. In this case, the burst management logic translates the master burst into smaller slave bursts, or into individual slave transfers if the slave does not support bursts. Until the master completes the burst, the arbiter logic prevents other masters from accessing the target slave.

For example, if a master initiates a burst of 16 transfers to a slave with maximum burst length of 8, the burst adapter logic initiates two bursts of length 8 to the slave. If the master initiates a burst of 14, the burst adapter logic segments the burst transfer into a burst of 8 words followed by a burst of 6 words, because the slave can only handle a maximum burst length of 8. If a master initiates a burst of 16 transfers to a slave that does not support bursts, the burst management logic initiates 16 separate transfers to the slave.



The burst adapter inserts one idle cycle at the start of each burst. System throughput is maximized when burst sizes are as large as possible.

In the case of a non-linewrap burst master connected to a slave with the `linewrapBursts` property set to `TRUE`, it is not always possible to issue the maximum-sized burst to the slave. In cases where a burst would cross a slave burst boundary, the burst adapter must issue the appropriate smaller bursts according to the master request. For example, if a non-linewrap burst master, data width=32 issues a burst of 8 at byte address 0xC to a linewrap burst slave, data width=32, the burst adapter issues a burst read of 5 at byte address 0xC followed by a burst read of size 3 at address 0x20. This example assumes a maximum burst size of 8 for both the master and slave. Table 2-3 provides some examples that show how bursts are handled between master and slaves with and without linewrapping. (Linewrap bursts are common for SDRAM components.) In these examples the following conditions are true:

- The master and slave have the same data width.
- Masters with the `linewrapBursts` property set to `TRUE` must also set `alwaysBurstMaxBurst` to `TRUE` due to a limitation in the burst adapter.

Table 2-4. Burst Behavior for Masters and Slaves with and without Linewrapping

Master Max Burst Size	8								4			
Slave Max Burst Size	8				4				8			
Master Linewrap Bursts	True		False		True		False		True		False	
Slave Linewrap Bursts	T	F	T	F	T	F	T	F	T	F	T	F
Master bursts	8@0	8@0	2@0	2@0	8@0	8@0	2@0	2@0	4@0	4@0	2@0	2@0
Slave receives	8@0	8@0	2@0	2@0	4@0, 4@4	8@0	2@0	2@0	4@0	4@0	2@0	2@0
Master bursts	8@3	8@3	6@3	6@3	8@3	4@0 4@4	6@3	6@3	4@7	4@7	4@7	4@3
Slave Receives	8@3	5@3, 3@0	5@3, 1@8	6@3	1@3, 4@4, 3@0	1@3, 4@4, 3@0	1@3, 4@4, 1@8	4@3, 2@7	1@7, 3@4	1@7, 3@4	1@7, 3@8	4@3



For more information about the `linewrapBursts` property, refer to the *Avalon Memory-Mapped Slave Interfaces* chapter in the *Avalon Interface Specifications*.

Interrupts

In systems where components have interrupt request (IRQ) sender interfaces, the system interconnect fabric includes interrupt controller logic. A separate interrupt controller is generated for each interrupt receiver. The interrupt controller aggregates IRQ signals from all interrupt senders, and maps them to user-specified values on the receiver inputs.



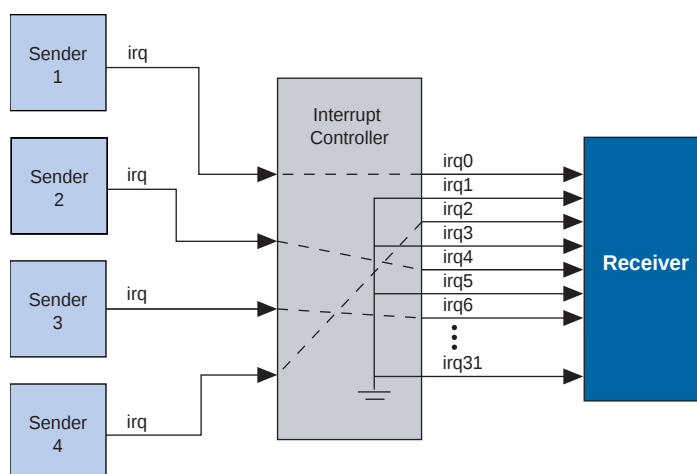
For further information, refer to the *Interrupt Interfaces* chapter in the *Avalon Interface Specifications*.

Individual Requests IRQ Scheme

In the individual requests IRQ scheme, the system interconnect fabric passes IRQs directly from the sender to the receiver, without making any assumptions about IRQ priority. In the event that multiple senders assert their IRQs simultaneously, the receiver logic (presumably under software control) determines which IRQ has highest priority, then responds appropriately.

Using individual requests, the interrupt controller can handle up to 32 IRQ inputs. The interrupt controller generates a 32-bit signal `irq[31:0]` to the receiver, and simply maps slave IRQ signals to the bits of `irq[31:0]`. Any unassigned bits of `irq[31:0]` are disabled. Figure 2-12 shows an example of the interrupt controller mapping the IRQs on four senders to `irq[31:0]` on a receiver.

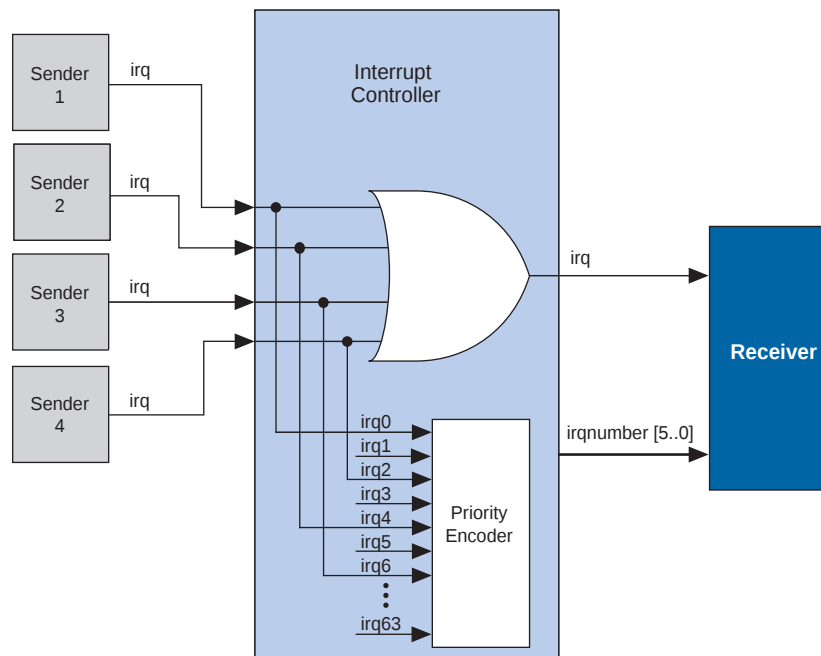
Figure 2-12. IRQ Mapping Using Software Priority



Priority Encoded Interrupt Scheme

In the priority encoded interrupt scheme, in the event that multiple slaves assert their IRQs simultaneously, the system interconnect fabric provides the interrupt receiver with a 1-bit interrupt signal, and the number of the highest priority active interrupt. An IRQ of lesser priority is undetectable until all IRQs of higher priority have been serviced.

Using priority encoded interrupts, the interrupt controller can handle up to 64 slave IRQ signals. The interrupt controller generates a 1-bit `irq` signal to the receiver, signifying that one or more senders have generated an IRQ. The controller also generates a 6-bit `irqnumber` signal, which outputs the encoded value of the highest pending IRQ. See Figure 2-13.

Figure 2-13. IRQ Mapping Using Hardware Priority

Assigning IRQs in SOPC Builder

You specify IRQ settings on the **System Contents** tab of SOPC Builder. After adding all components to the system, you make IRQ settings for all interrupt senders, with respect to each interrupt receiver. For each slave, you can either specify an IRQ number, or specify not to connect the IRQ.

Reset Distribution

SOPC Builder generates the logic used in the system interconnect fabric, which drives the reset pulse to all the logic. The system interconnect fabric distributes the reset signal conditioned for each clock domain. The duration of the reset signal is at least one clock period.

The system interconnect fabric asserts the system-wide reset in the following conditions:

- The global reset input to the SOPC Builder system is asserted.
- Any component asserts its `resetrequest` signal.

The global reset and reset requests are ORed together. This signal is then synchronized to each clock domain associated to an Avalon-MM port, which causes the asynchronous resets to be de-asserted synchronously.

Referenced Documents

This chapter references the following documents:

- [Avalon Interface Specifications](#)
- [Avalon Memory-Mapped Bridges](#) chapter in volume 4 of the *Quartus II Handbook*
- [Avalon Memory-Mapped Design Optimizations](#) chapter in the *Embedded Design Handbook*

Document Revision History

[Table 2-5](#) shows the revision history for this chapter.

Table 2-5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	■ Added table showing the behavior of the burst adapter for master and slaves with and without <code>linewrapBursts</code> set to <code>TRUE</code>.	Clarification of burst behavior.
November 2008, v8.1.0	■ Added discussion of a non-bursting Avalon-MM master connected to a Avalon-MM slave with <code>linewrapBursts = TRUE</code>. Removed discussion on minimum arbitration shares; this feature is no longer supported. ■ Changed page size to 8.5 x 11 inches	Minor update to reflect software changes.
May 2008, v8.0.0	■ Updated references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications. ■ Moved clock-crossing bridge section from this chapter to chapter 11.	The two specifications have been combined into one for all Avalon interfaces.



For previous versions of the *Quartus II Handbook*, refer to the [Quartus II Handbook Archive](#).

