

Introduction

Avalon® switch fabric is a high-bandwidth interconnect structure that consumes minimal logic resources and provides greater flexibility than a typical shared system bus. This chapter describes the functions of Avalon switch fabric and the implementation of those functions.

High-Level Description

Avalon switch fabric is the glue that binds together components in a system based on the Avalon interface.

Avalon switch fabric is the collection of interconnect and logic resources that connects Avalon master and slave ports on components in a system. Avalon switch fabric encapsulates the connection details of a system. Avalon switch fabric guarantees that signals travel correctly between master and slave ports, as long as the ports adhere to the rules of the Avalon interface specification. As a result, system designers can think at a higher level and focus on the parts of a system that add value, rather than worry about the interconnect.



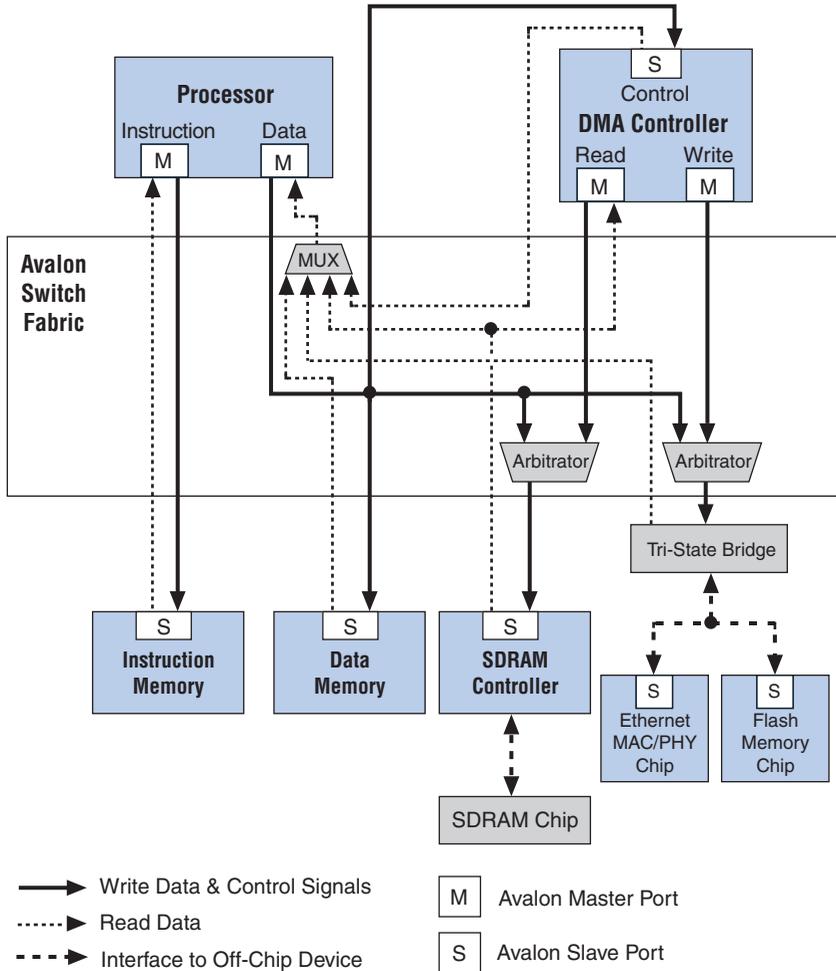
For details on the Avalon interface, refer to the *Avalon Interface Specification* available at www.altera.com. For details on how to use SOPC Builder to create Avalon switch fabric, refer to the *Tour of the SOPC Builder User Interface* chapter in volume 4 of the *Quartus II Handbook*.

Avalon switch fabric supports:

- 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 peripherals
- Components of differing data widths
- Big-endian or little-endian components
- Components operating in different clock domains
- Components using multiple Avalon ports

Figure 3–1 shows a simplified diagram of the Avalon switch fabric in an example system with multiple masters.

Figure 3–1. Avalon Switch Fabric Block Diagram – Example System



Some components in Figure 3–1 use multiple Avalon ports. Because an Avalon component can have multiple Avalon ports, you can use Avalon switch fabric to create *super interfaces* that provide more functionality than a single Avalon port. For example, an Avalon slave port can have only one interrupt-request (IRQ) signal. However, by using three Avalon

slave ports together, you can create a component that generates three separate IRQs. In this case, SOPC Builder generates the Avalon switch fabric to connect all ports.

Generating Avalon switch fabric is SOPC Builder's primary purpose. Because SOPC Builder generates Avalon switch fabric automatically, most users do not interact directly with it or the HDL that describes it. You do not need to know anything about the internal workings of Avalon switch fabric to take advantage of the services it provides. On the other hand, a basic understanding of how it works can help you optimize your components and systems. For example, knowledge of the arbitration mechanism can help designers of multi-master systems minimize the impact of arbitration on the system throughput.

Fundamentals of Avalon Switch Fabric Implementation

Avalon switch fabric uses active logic to implement a switched interconnect structure that provides a dedicated path between master and slave ports. Avalon switch fabric consists of synchronous logic and routing resources inside an FPGA.

At each port interface, Avalon switch fabric manages Avalon transfers, responding to signals from the connected component. The signals that appear on the master port and corresponding slave port during a transfer can be very different, depending on how the Avalon switch fabric transports signals between the master-slave pair. In the path between master and slave ports, the Avalon switch fabric can introduce registers for timing synchronization, finite state machines for event sequencing, or nothing at all, depending on the services required by those ports.

Functions of Avalon Switch Fabric

Avalon switch fabric logic provides the following functions:

- Address Decoding (page 3-4)
- Data-Path Multiplexing (page 3-5)
- Wait-State Insertion (page 3-6)
- Pipelining for High Performance (page 3-7)
- Pipeline Management (page 3-8)
- Endian Conversion (page 3-9)
- Native Address Alignment & Dynamic Bus Sizing (page 3-10)
- Arbitration for Multi-Master Systems (page 3-13)
- Burst Management (page 3-20)
- Clock Domain Crossing (page 3-21)
- Interrupt Controller (page 3-25)
- Reset Distribution (page 3-27)

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 the SOPC Builder GUI. The remaining sections of this chapter describe how SOPC Builder implements each function.

Address Decoding

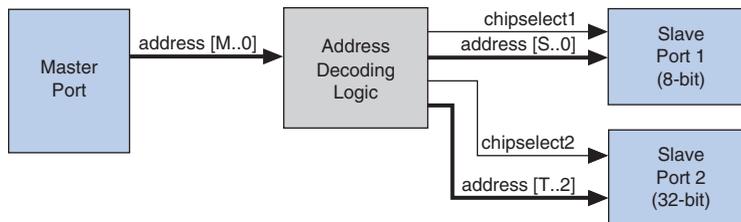
Address decoding logic in the Avalon switch fabric distributes an appropriate address and produces a chipselect signal for each slave port. Address decoding logic simplifies component design in the following ways:

- The Avalon switch fabric selects a slave port whenever it is being addressed by a master. Slave components do not need to decode the address to determine when they are selected.
- Slave port addresses are always properly aligned for the data width of the slave port.
- SOPC Builder automatically generates address decoding logic to implement the memory map specified in the GUI. Therefore, changing the system memory map does not involve manually editing HDL.

Figure 3–2 shows a block diagram of the address-decoding logic for one master and two slave ports. Separate address-decoding logic is generated for every master port in a system.

As shown in Figure 3–2, the address decoding logic handles the difference between the master address width (M) and the individual slave address widths (S & T). It also maps only the necessary master address bits to access words in each slave port's address space.

Figure 3–2. Block Diagram of Address Decoding Logic



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

In the SOPC Builder GUI, the user-configurable aspects of address decoding logic are controlled by the **Base** setting in the list of active components on the **System Contents** page, as shown in [Figure 3-3](#).

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

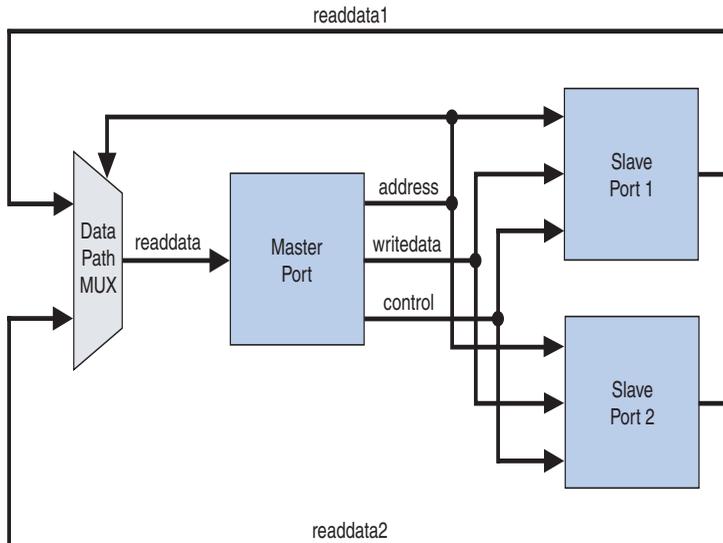
| Module Name | Description | Base | End | IRQ |
|--------------------|--------------------|------------|--------------|--------|
| cpu | Nios II Proces... | | | |
| instruction_master | Master port | | | |
| data_master | Master port | | | IRQ 0 |
| jtag_debug_mod... | Slave port | 0x02120000 | 0x021207FF | IRQ 31 |
| 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 |

Data-Path Multiplexing

Data-path multiplexing logic in the Avalon switch fabric aggregates read-data signals from multiple slave ports during a read transfer, and presents the signals from only the selected slave back to the master port.

[Figure 3-4](#) shows a block diagram of the data-path multiplexing logic for one master and two slave ports. SOPC Builder generates separate data-path multiplexing logic for every master port in the system.

Figure 3-4. Block Diagram of Data-Path Multiplexing Logic



Data-path multiplexing is not necessary in the write-data direction for write transfers. The `writedata` signals are distributed equally to all slave ports, and each slave port ignores `writedata` except for when the address-decoding logic selects that port.

In the SOPC Builder GUI, the generation of data-path multiplexing logic is specified using the connections panel on the **System Contents** page, as shown in [Figure 3–5](#).

Figure 3–5. Connection Panel Settings in SOPC Builder Control Data-Path Multiplexing

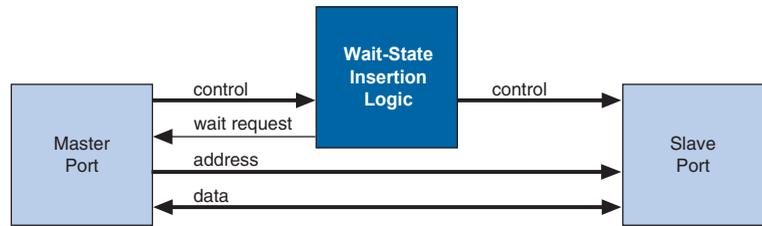
| Use | Module Name | Description | Input Clock | Base |
|--------------------------|------------------|-------------------------------------|-------------|------------|
| <input type="checkbox"/> | cpu | ARM9 Processor - Altera Corporation | ch_05 | |
| <input type="checkbox"/> | instruction_bus | Master port | | 0x0 |
| <input type="checkbox"/> | data_master | Master port | | 0x02200000 |
| <input type="checkbox"/> | cpu_slave_port | Slave port | | |
| <input type="checkbox"/> | ahb_sram_bridge | Avalon Tristate Bridge | ch_05 | |
| <input type="checkbox"/> | ahb_sram | Cypress CY7C130C SDRAM | ch_05 | 0x02000000 |
| <input type="checkbox"/> | ahb_flash_bridge | PC (Parallel) I/O | ch_05 | 0x02220000 |
| <input type="checkbox"/> | ahb_flash | Inter-via Inver | ch_05 | 0x02200020 |
| <input type="checkbox"/> | ahb_uart | UART | ch_05 | 0x02220000 |
| <input type="checkbox"/> | ahb_ethernet | LANE0/1/2 interface (Ethernet) | ch_05 | 0x02210000 |

Wait-State Insertion

Wait states extend the duration of a transfer by one or more cycles for the benefit of components with special synchronization needs. Wait-state insertion logic accommodates the timing needs of each slave port, and coordinates the master port to wait until the slave can proceed. Avalon switch fabric inserts wait states into a transfer when the target slave port cannot respond in a single clock cycle. Avalon switch 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 3–6](#) shows a block diagram of the wait-state insertion logic between one master and one slave.

Figure 3–6. Block Diagram of Wait-State Insertion Logic



Avalon switch fabric can force a master port to wait for several reasons in addition to the wait state needs of a slave port. For example, arbitration logic in a multi-master system can force a master port to wait until it is granted access to a slave port.

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

Pipelining for High Performance

SOPC Builder can pipeline the Avalon switch fabric by inserting stages of registers between master-slave pairs. Adding pipeline registers can increase the f_{MAX} performance of the system and ensure that the critical timing path does not occur inside the Avalon switch fabric.

The pipeline registers introduce one or more clock cycles of latency between master-slave pairs, which creates a trade-off between transfer latency and f_{MAX} performance. The pipeline registers can also increase logic utilization considerably, depending on the complexity of the system. Components that support pipelined Avalon transfers minimize the effects of the pipeline latency. For details on how pipelining for high performance affects pipelined Avalon ports, see section “[Pipeline Management](#)” on page 3–8.



Pipeline registers are most likely to improve performance for the case of many master ports sharing a common slave port. For N masters accessing a slave port, the increased latency is on the order of $(\log_2 N + 1)$.

You specify whether or not to add pipelining for high performance with the clock settings table on the **System Contents** tab in SOPC Builder, shown in [Figure 3–7](#). You can pipeline each clock domain separately by turning on its **Pipeline** check box.

Figure 3–7. Turning On Pipelining for High Performance

| Clock | Source | MHz | Pipeline |
|-----------------|-------------|--------|-------------------------------------|
| clk_85 | External | 85.0 | <input type="checkbox"/> |
| clk_233 | c0 from pll | 233.75 | <input checked="" type="checkbox"/> |
| click to add... | | | <input type="checkbox"/> |

Pipeline Management

The Avalon interface supports pipelined read transfers. A pipelined Avalon port can start multiple read transfers in succession without waiting for the prior transfers to complete. Pipelined transfers allow master-slave pairs to achieve maximum throughput, even though the slave port may require one or more cycles of latency to return data for each transfer.

SOPC Builder generates Avalon switch 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 port, SOPC Builder guarantees that read data arrives at each master port in the order requested. Because master and slave ports often have mismatched pipeline latency, Avalon switch fabric often contains logic to reconcile the differences. Many cases are possible, as shown in [Table 3–1](#).

| Master Port | Slave Port | Pipeline Management Logic Structure |
|-------------|--|---|
| No Pipeline | No Pipeline | The Avalon switch fabric does not instantiate logic to handle pipeline latency. |
| No Pipeline | Pipelined with Fixed or Variable Latency | The Avalon switch fabric forces the master port to wait through any slave-side latency cycles. This master-slave pair gains no benefits of pipelining, because the master port is not pipelined and therefore waits for each transfer to complete before beginning a new transfer. However, while the master port is waiting, the slave port can accept transfers from a different master port. |
| Pipelined | No Pipeline | The Avalon switch fabric carries out the transfer as if neither port were pipelined, forcing the master port to wait until the slave port returns data. |

Table 3–1. Various Cases of Pipeline Latency in a Master-Slave Pair (Part 2 of 2)

| Master Port | Slave Port | Pipeline Management Logic Structure |
|-------------|---------------------------------|--|
| Pipelined | Pipelined with Fixed Latency | The Avalon switch fabric coordinates the master port to capture data at the exact clock cycle when data is valid on the slave port. This case enables this master-slave pair to achieve maximum throughput performance. |
| Pipelined | Pipelined with Variable Latency | This is the simplest pipelined case, in which the slave port asserts a signal when its data is valid, and the master port captures the data. This case enables this master-slave pair to achieve maximum throughput performance. |

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

Endian Conversion

Starting with version 5.1 of the Quartus II software, SOPC Builder supports big endian master ports. Prior to version 5.1, SOPC Builder treated all components as little endian. With version 5.1 and later, an Avalon-based system can contain both big and little endian components.

The endianness of an Avalon port depends on the component design. Endianness affects the order a master port expects individual bytes to be arranged within a larger word. If all master ports in the system use the same endianness, then all master ports' perception of byte addresses is consistent within the system. In this case there is no further endian-related design consideration required.

Avalon switch fabric provides endian-conversion functionality to allow master ports of differing endianness to share memory. The Avalon endian-conversion logic hides the endian difference of master ports when the following conditions are met:

1. The master ports access a common memory slave port.
2. The data width of the master ports are equal.
3. The master ports read and write the memory using only native-width units of data. For example, a 32-bit master port can read and write only 32-bit units of data.
4. The master ports use a common interpretation of the data type.

As an example, consider a three-chip system comprised of a discrete 32-bit CPU chip, an FPGA containing 32-bit coprocessor logic (an SOPC Builder system), and a shared DDR SDRAM chip. Furthermore, suppose that the CPU is big endian, while the FPGA coprocessor system is little endian. In this case, the CPU and the coprocessor can share data in the SDRAM seamlessly without manually accounting for the endianness of data.



The Avalon switch fabric does not guarantee proper byte arrangement for big-endian master ports when accessing peripheral registers via an Avalon slave port.

Native Address Alignment & Dynamic Bus Sizing

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

- Native address alignment
- Dynamic bus sizing

Table 3–2 demonstrates native address alignment and dynamic bus sizing for a 32-bit master port connected to a 16-bit slave port (a 2:1 ratio). In this example, the slave port is mapped to base address 0x0000000 in the master port. In Table 3–2, OFFSET refers to the offset into the 16-bit slave port address space.

| 32-bit Master Address | Data with Native Alignment | Data with Dynamic Bus Sizing |
|------------------------------|-----------------------------------|---|
| 0x00000000 (word 0) | 0x0000 : OFFSET [0] | 0xOFFSET [1] : OFFSET [0] |
| 0x00000004 (word 1) | 0x0000 : OFFSET [1] | 0xOFFSET [3] : OFFSET [2] |
| 0x00000008 (word 2) | 0x0000 : OFFSET [2] | 0xOFFSET [5] : OFFSET [4] |
| 0x0000000C (word 3) | 0x0000 : OFFSET [3] | 0xOFFSET [7] : OFFSET [6] |
| ... | | |
| (word <i>N</i>) | 0x0000 : OFFSET [<i>N</i>] | 0xOFFSET [2 <i>N</i> +1] : OFFSET [2 <i>N</i>] |

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

Native Address Alignment

Slave ports that access address-mapped registers inside the component generally use native address alignment. The defining properties of native address alignment are:

- Each slave offset (that is, word) maps to exactly one master word, regardless of the data width of the ports.
- One transfer on the master port generates exactly one transfer on the slave port.

In the case of native address alignment, Avalon switch fabric maps all slave data bits to the lower bits of the master data, and fills any remaining upper bits with zero. Avalon switch fabric performs simple wire-mapping in the data path, but nothing else.

Native address alignment is only valid if the master data width is equal to or wider than the slave data width. If an N -bit master port is connected to a wider slave with native alignment, then the master port can access only the lower N data bits at each offset in the slave.

Dynamic Bus Sizing

Slave ports that access memory devices generally use dynamic bus sizing. Dynamic bus sizing hides the details of interfacing a narrow memory device to a wider master port, and vice versa. When an N -bit master port accesses a slave port with dynamic bus sizing, the master port operates exclusively on full N -bit words of data, without awareness of the slave data width.



When using dynamic bus sizing, the slave data width 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 port to access any memory device seamlessly, regardless of the data width.

In the case of dynamic bus sizing, the Avalon switch 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 $N:1$, the dynamic bus-sizing logic generates N slave transfers. The master port pays a performance penalty, because it must wait while multiple slave-side transfers complete.

In the case of a read transfer, the Avalon switch fabric merges slave data from multiple read transfers before presenting them to the master port. A read transfer from a wide master port always causes multiple slave-read transfers to sequential addresses in the slave's address space. For example, even if a 32-bit master port needs only one byte from a dynamically-aligned 8-bit memory, a master read transfer generates four slave transfers, and the master port waits until all four transfers complete.

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

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. There is no performance penalty when accessing a wider slave port using dynamic bus sizing.

Table 3-3 demonstrates the case of a 32-bit master port accessing a 64-bit slave port with dynamic bus sizing. In the table, offset refers to the offset into the slave port memory space.

| Table 3-3. 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 port. In the case of a write transfer, the dynamic bus-sizing logic uses the slave-side byte-enable signals to write only to the appropriate byte lanes.

Arbitration for Multi-Master Systems

Avalon switch fabric supports systems with multiple master components. In a system with multiple master ports, such as the system pictured in [Figure 3-1 on page 3-2](#), the Avalon switch fabric provides shared access to slave ports using a technique called slave-side arbitration. Slave-side arbitration determines which master port gains access to a specific slave port in the event that multiple master ports attempt to access the same slave port at the same time.

The multi-master architecture used by Avalon switch fabric offers the following benefits:

- Eliminates the need to create arbitration hardware manually.
- Allows multiple master ports to transfer data simultaneously. Unlike traditional host-side arbitration architectures in which each master must wait until it is granted access to the shared bus, multiple Avalon masters can simultaneously perform transfers with independent slaves. Arbitration logic stalls a master port only when multiple master ports attempt to access the same slave port during the same cycle.
- Eliminates unnecessary master-slave connections. The connection between a master port and a slave port exists only if it is specified in the SOPC Builder GUI. If a master port never initiates transfers to a specific slave port, 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 port is specified independently. For example, you can grant one master port the most access to a particular slave port, while other master ports have more access to other slave ports.
- Simplifies master component design. The details of arbitration are encapsulated inside the switch fabric. Each Avalon master port connects to the Avalon switch fabric like it is the only master port in the system. As a result, you can reuse a component in single-master and multi-master systems without requiring design changes to the component.

This section discusses the architecture of the Avalon switch fabric generated by SOPC Builder for multi-master systems.

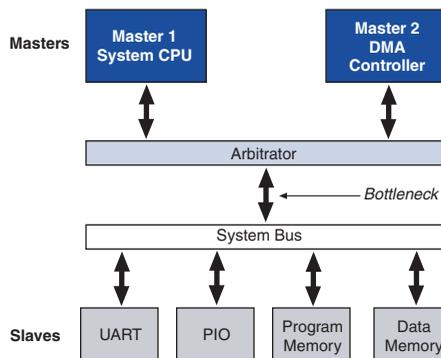
Traditional Shared Bus Architectures

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. A single arbitrator controls the bus (that is, the path between bus masters and bus slaves), so that multiple bus masters do not simultaneously drive the bus and cause electrical contention. Each bus master requests control of the bus from the arbitrator, and the arbitrator grants access to a single master at a time. Once a master has control of the bus, the master performs transfers with a bus slave. If multiple masters attempt to access the bus at the same time, the arbitrator allocates the bus resources to a single master based on fixed arbitration rules, forcing all other masters to wait. For example, the priority arbitration scheme—in which the arbitrator always grants control to the master with the highest priority—is used in many existing bus architectures.

Figure 3–8 illustrates the bus architecture for a traditional processor system. Access to the shared system bus becomes the bottleneck for throughput and utilization performance. Only one master has access to the bus at a time, which means that other masters are forced to wait (diminishing throughput), and only one slave can transfer data at a time (diminishing utilization).

Figure 3–8. Bus Architecture in a Traditional Microprocessor System



Slave-Side Arbitration

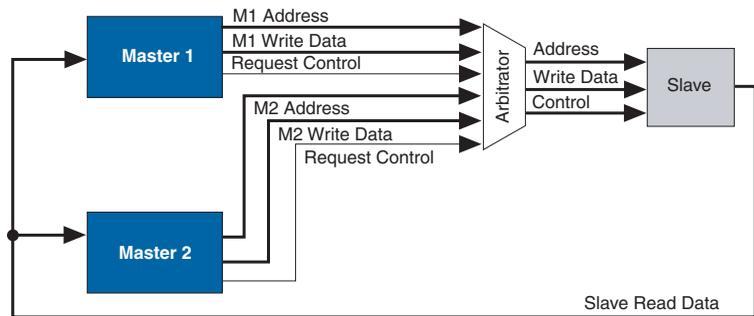
The multi-master architecture used by Avalon switch fabric eliminates the bottleneck for access to a shared bus, because the system does not have shared bus lines. Avalon master-slave pairs are connected by dedicated paths. A master port never waits to access a slave port, unless a different master port attempts to access the same slave port at the same time. As a result, multiple master ports can be active at the same time, simultaneously transferring data with independent slave ports.

A multi-master Avalon system requires arbitration, but only when two masters contend for the same slave port. This arbitration is called slave-side arbitration, because it is implemented at the point where two (or more) master ports connect to a single slave. Master ports contend for individual slave ports, not for a shared bus resource.

For example, [Figure 3–1 on page 3–2](#) demonstrates a system with two master ports (a CPU and a DMA controller) sharing a slave port (an SDRAM controller). Arbitration is performed on the SDRAM slave port; the arbitrator dictates which master port gains access to the slave port if both master ports initiate a transfer with the slave port at the same time.

[Figure 3–9](#) focuses on the two master ports and the shared slave port, and shows additional detail of the data, address, and control paths. The arbitrator logic multiplexes all address, data, and control signals from a master port to a shared slave port.

Figure 3–9. Detailed View of Multi-Master Connections



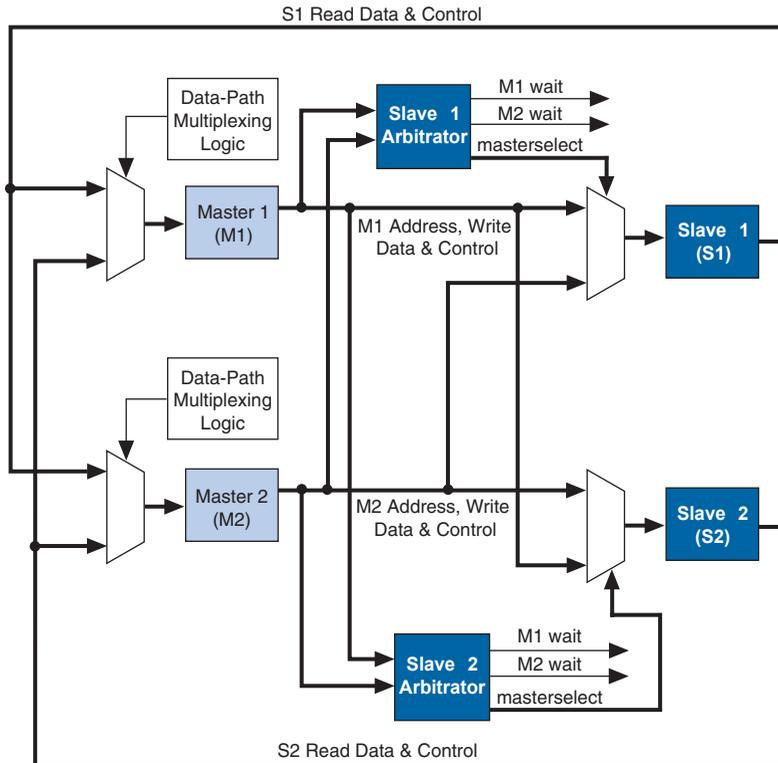
Arbitrator Details

SOPC Builder generates an arbitrator for every slave port connected to multiple master ports, based on arbitration parameters specified in the SOPC Builder GUI. The arbitrator logic performs the following functions for its associated slave port:

- Evaluates the address and control signals from each master port at every clock cycle when a new transfer can begin, and determines which master port, if any, is requesting access to the slave.
- Chooses which master port gains access to the slave next.
- Grants access to the chosen master port (that is, allows it to proceed with the transfer), and forces all other requesting master ports to wait.
- Uses multiplexers to connect address, control, and data paths between the multiple master ports and the slave port. The arbitrator logic guarantees that an appropriate master port (if any) is connected to the slave port.

Figure 3–10 shows the arbitrator logic in an example multi-master system with two master ports, each connected to two slave ports.

Figure 3–10. Block Diagram of Arbitrator Logic



Arbitration Rules

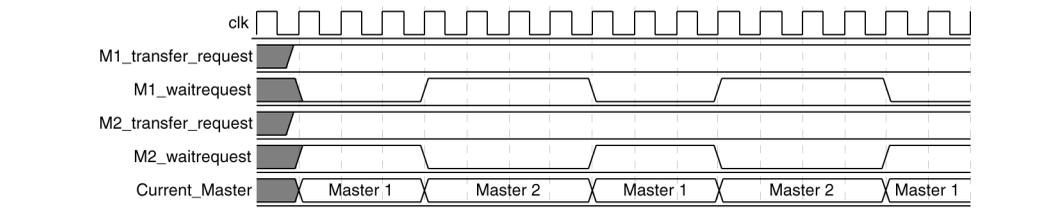
This section describes the rules by which the arbitrator grants access to master ports when they contend.

Fairness-Based Shares

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

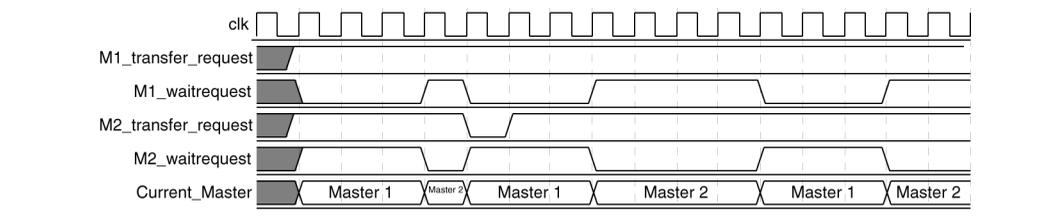
For example, assume that two master ports continuously attempt to perform back-to-back transfers to a slave port. Master 1 is assigned three shares and Master 2 is assigned four shares. In this case, the arbitrator grants Master 1 access for three transfers, then Master 2 for four transfers. This cycle repeats indefinitely. Figure 3–11 demonstrates this case, showing each master port's transfer request output, wait request input (which is driven by the arbitrator logic), and the current master with control of the slave.

Figure 3–11. Arbitration of Continuous Transfer Requests from Two Master Ports



If a master stops requesting transfers before it exhausts its shares, it forfeits all its remaining shares, and the arbitrator grants access to another requesting master. See Figure 3–12. After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbitrator grants access back to Master 1, which gets a replenished supply of shares.

Figure 3–12. Arbitration of Two Masters with a Gap in Transfer Requests



Round-Robin Scheduling

When multiple master ports contend for access to a slave port, the arbitrator grants shares in round-robin order. At every slave transfer, only requesting master ports are included in the round-robin arbitration.

Burst Transfers

Avalon burst transfers grant a master port uninterrupted access to a slave port for a specified number of transfers. The master port specifies the number of transfers when it initiates the burst. Once a burst begins between a master-slave pair, arbitrator logic does not allow any other master port to access the slave port until the burst completes. For further information, see [“Burst Management” on page 3–20](#).

Minimum Share Value

A component design can declare the minimum number of shares in each round-robin cycle, which affects how the arbitrator grants access. For example, if a slave port has a minimum share value of ten, then the arbitrator will grant at least ten shares to any master port when it begins a sequence of transfer requests. The arbitrator might grant more shares, if the master port is assigned more shares in the SOPC Builder GUI.

By declaring a minimum share value of N , a slave port declares that it is more efficient at handling continuous sequential transfers of length N . Accessing the slave port in sequences less than N incurs performance penalties that might prevent the slave port from achieving higher performance. By nature, continuous back-to-back master transfers tend to access sequential addresses. However, there is no requirement that the master port perform transfers to sequential addresses.



Burst transfers provide even higher performance for continuous transfers when they are guaranteed to access sequential addresses. The minimum share value does not apply to slave ports that support bursts; the burst length takes precedence over minimum share value. See [“Burst Management” on page 3–20](#) for information.

Setting Arbitration Parameters in the SOPC Builder GUI

You specify the arbitration shares for each master using the connection panel on the **System Contents** tab of the SOPC Builder GUI, as shown in [Figure 3–13](#).

Figure 3–13. Arbitration Settings on the System Contents Tab

| Module Name | Description | Clock |
|--|---|---|
| <ul style="list-style-type: none"> [-] cpu <ul style="list-style-type: none"> instruction_master data_master 1 jtag_debug_module | Nios II Processor - Alte... Master port Master port Slave port | clk  |
| 1 [-] sys_clk_timer | Interval timer | clk |
| 1 1 [-] ext_ram_bus | Avalon Tri-State Bridge | clk |
| [-] ext_flash | Flash Memory (Commo... |  |
| [-] ext_ram | IDT71V416 SRAM |  |
| 1 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 view them, on the View menu, click **Show Arbitration**.

Burst Management

Avalon switch fabric provides burst management logic to accommodate the burst capabilities of each port in the system, including ports that do not support burst transfers. Burst management logic is 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 port might be capable of initiating a burst longer than a slave port'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 port does not support bursts. Until the master port completes the burst, the Avalon arbitrator logic prevents other master ports from accessing the target slave port.

For example, if a master port initiates a burst of 16 transfers to a slave port with maximum burst length of 8, the burst management logic initiates two bursts of length 8 to the slave port. If a master port initiates a burst of 16 transfers to a slave port that does not support bursts, the burst management logic initiates 16 separate transfers to the slave port.

Clock Domain Crossing

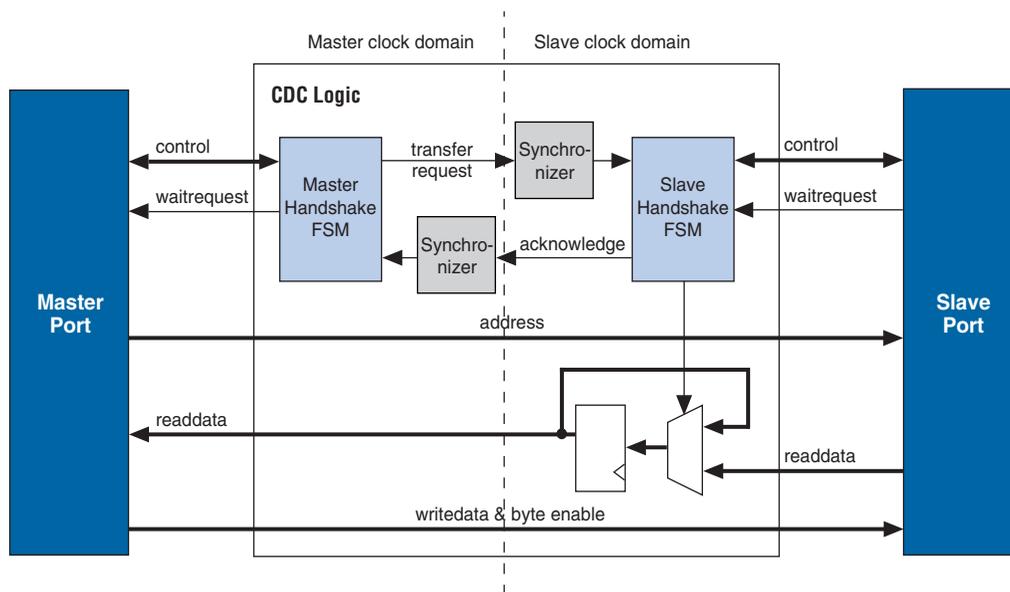
SOPC Builder generates clock-domain crossing (CDC) logic that hides the details of interfacing components operating in asynchronous clock domains. The Avalon switch fabric upholds the Avalon protocol with each port independently, and therefore each Avalon port need only be aware of its own clock domain. The Avalon switch fabric logic propagates transfers across clock domain boundaries transparently to the user.

The CDC logic in Avalon switch fabric provides the following benefits that simplify system design efforts:

- Allows component interfaces to operate at a different clock frequency than system logic.
- Eliminates the need to design CDC hardware manually.
- Each Avalon port operates in only one clock domain, which reduces design complexity of components.
- Enables master ports to access any slave port without awareness of the slave clock domain.
- Allows you to focus performance optimization efforts only on components that require fast clock speed.

Description of Clock Domain-Crossing Logic

The CDC logic consists of two finite state machines (FSM), one in each clock domain, which use a simple hand-shaking protocol to propagate transfer control signals (read request, write request, and the master wait-request signals) across the clock boundary. [Figure 3-14](#) shows a block diagram of the clock domain crossing logic between one master and one slave port.

Figure 3–14. Block Diagram of Clock Domain-Crossing Logic

The Synchronizer blocks in [Figure 3–14](#) use multiple stages of flip-flops to eliminate the propagation of metastable events on the control signals that enter the hand-shake FSMs.

The CDC logic works with any clock ratio. Altera® tests the CDC logic extensively on a variety of system architectures, both in simulation and in hardware, to ensure that the logic functions correctly.

The typical sequence of events for a transfer across the CDC logic is described below:

1. Master port asserts address, data, and control signals.
2. The master handshake FSM captures the control signals, and immediately forces the master port to wait.



The FSM uses only the control signals, not address and data. For example, the master port simply holds the address signal constant until the slave side has safely captured it.

3. Master handshake FSM initiates a transfer request to the slave handshake FSM.

4. The transfer request is synchronized to the slave clock domain.
5. The slave handshake FSM processes the request, performing the requested transfer with the slave port.
6. When the slave transfer completes, the slave handshake FSM sends an acknowledge back to the master handshake FSM.
7. The acknowledge is synchronized back to the master clock domain.
8. The master handshake FSM completes the transaction by releasing the master port from the wait condition.

Transfers proceed as normal on the slave and the master side, without a special protocol to handle crossing clock domains. From the perspective of a slave port, there is nothing different about a transfer initiated by a master port in a different clock domain. From the perspective of a master port, a transfer across clock domains simply takes extra clock cycles. Similar to other transfer delay cases (for example, arbitration delay and/or wait states on the slave side), the Avalon switch fabric simply forces the master port to wait until the transfer terminates. As a result, latency-aware master ports do not benefit from pipelining when performing transfers to a different clock domain.

Location of Clock Domain Crossing Logic

SOPC Builder automatically determines where to insert the CDC logic, based on the system contents and the connections between components. SOPC Builder places CDC logic to maintain the highest transfer rate for all components. SOPC Builder evaluates the need for CDC logic on each slave port independently, and generates CDC logic wherever necessary.

Duration of Transfers Crossing Clock Domains

CDC logic extends the duration of master transfers across clock domain boundaries. In the worst case, each transfer is extended by five master clock cycles and five slave clock cycles. The components of this delay are the following:

- Four additional master clock cycles, due to the master-side clock synchronizer
- Four additional slave clock cycles, due to the slave-side clock synchronizer
- One additional clock in each direction, due to potential metastable events as the control signals cross clock domains

Implementing Multiple Clock Domains in the SOPC Builder GUI

You specify the clock domains used by your system on the **System Contents** tab of the SOPC Builder GUI. You define the input clocks to the system using the clock settings table, shown in [Figure 3–15](#). Clock sources can be driven by external input signals to the system module, or by PLLs inside the system module. Clock domains are differentiated based on the name of the clock. It is possible to create multiple asynchronous clocks with the same frequency.

Figure 3–15. Clock Settings on the System Contents Tab

| Clock | Source | MHz | Pipeline |
|-----------------|-------------|--------|-------------------------------------|
| clk_85 | External | 85.0 | <input type="checkbox"/> |
| clk_233 | c0 from pll | 233.75 | <input checked="" type="checkbox"/> |
| click to add... | | | <input type="checkbox"/> |

After you define the system clocks, you specify which clock drives which components using the table of active components, as shown in [Figure 3–16](#).

Figure 3–16. Assigning Clocks to Components

| Module Name | Description | Clock | Base | End | IRQ |
|----------------------|---------------------------|---------|------------|------------|-----|
| high_res_timer | Interval timer | clk | 0x02120820 | 0x0212083F | 3 |
| seven_seg_pio | PIO (Parallel I/O) | clk | 0x02120890 | 0x0212089F | |
| reconfig_request_pio | PIO (Parallel I/O) | fastclk | 0x021208A0 | 0x021208AF | |
| uart1 | UART (RS-232 serial port) | clk | 0x0212085F | 0x0212085F | 4 |
| sysid | System ID Peripheral | clk | 0x021208B8 | 0x021208BF | |
| sdram | SDRAM Controller | clk | 0x01000000 | 0x01FFFFFF | |
| dma_0 | DMA | fastclk | 0x00800000 | 0x0080001F | 7 |
| read_buffer | On-Chip Memory (RAM ...) | fastclk | 0x00801000 | 0x00801FFF | |
| write_buffer | On-Chip Memory (RAM ...) | fastclk | 0x00802000 | 0x00802FFF | |



For further details, refer to the *Building Systems with Multiple Clock Domains* chapter in volume 4 of the *Quartus II Handbook*.

Interrupt Controller

In systems with one or more slave ports that generate IRQs, the Avalon switch fabric includes interrupt controller logic. A separate interrupt controller is generated for each master port that accepts interrupts. The interrupt controller aggregates IRQ signals from all slave ports, and maps slave IRQ outputs to user-specified values on the master IRQ inputs.

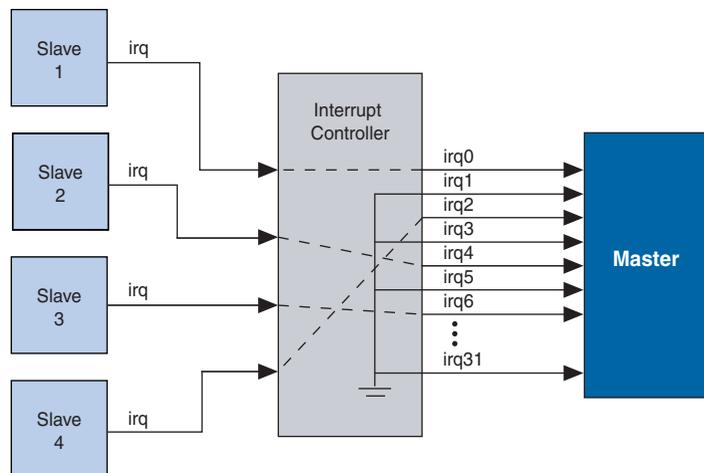
Each slave port optionally produces an IRQ output signal. There are two master signals related to interrupts: `irq` and `irqnumber`. SOPC Builder generates the interrupt controller in one of two configurations, software priority or hardware priority, depending on the interrupt signals present on the master port.

Software Priority

In the software priority configuration, the Avalon switch fabric passes IRQs directly from slave to master port, without making any assumptions about IRQ priority. In the event that multiple slave ports assert their IRQs simultaneously, the master logic (presumably under software control) determines which IRQ has highest priority, then responds appropriately.

Using software priority, the interrupt controller can handle up to 32 slave IRQ inputs. The interrupt controller generates a 32-bit signal `irq[31..0]` to the master port, and simply maps slave IRQ signals to the bits of `irq[31..0]`. Any unassigned bits of `irq[31..0]` are permanently disabled. [Figure 3-17](#) shows an example of the interrupt controller mapping the IRQs on four slave ports to `irq[31..0]` on a master port.

Figure 3-17. IRQ Mapping Using Software Priority

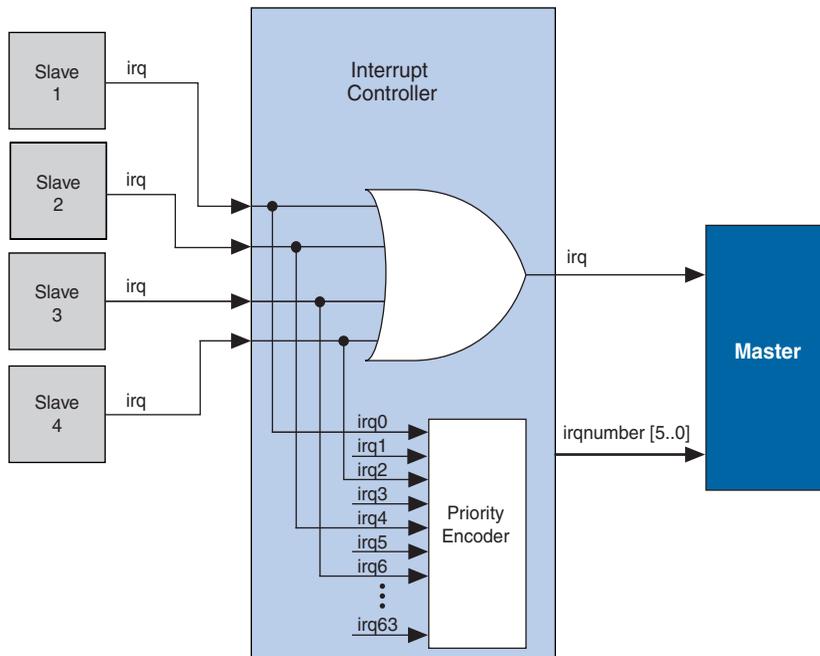


Hardware Priority

In the hardware priority configuration, in the event that multiple slaves assert their IRQs simultaneously, the Avalon switch fabric (i.e. hardware logic) identifies the IRQ of highest priority and passes only that IRQ number to the master port. An IRQ of lesser priority is undetectable until a master port clears all IRQs of higher priority.

Using hardware priority, the interrupt controller can handle up to 64 slave IRQ signals. The interrupt controller generates a 1-bit `irq` signal to the master port, signifying that one or more slave ports 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 3–18](#).

Figure 3–18. IRQ Mapping Using Hardware Priority



Assigning IRQs in the SOPC Builder GUI

You specify IRQ settings on the **System Contents** tab of the SOPC Builder GUI. After adding all components to the system, you make IRQ settings for all slave ports that can generate IRQs, with respect to each master

port. For each slave port, you can either specify an IRQ number, or specify not to connect the IRQ. Figure 3–19 shows the IRQ settings for multiple slave IRQs that drive the master component named `cpu`.

Figure 3–19. Assigning IRQs in the SOPC Builder GUI

| Module Name | Description | Clock | Base | End | IRQ |
|------------------------------|------------------------------|-------|------------|--------------|-----|
| <code>cpu</code> | Nios II Processor - Alter... | clk | 0x02120000 | 0x021207FF | |
| <code>ext_ram_bus</code> | Avalon Tri-State Bridge | clk | | | |
| <code>ext_flash</code> | Flash Memory (Common ... | | 0x00000000 | 0x007FFFFFFF | |
| <code>ext_ram</code> | IDT71V416 SRAM | | 0x02000000 | 0x020FFFFFFF | |
| <code>epcs_controller</code> | EPCS Serial Flash Contr... | clk | 0x02100000 | 0x021007FF | HC |
| <code>lan91c111</code> | LAN91c111 Interface (E... | | 0x02110000 | 0x0211FFFF | 6 |
| <code>sys_clk_timer</code> | Interval timer | clk | 0x02120800 | 0x0212081F | 1 |
| <code>jtag_uart</code> | JTAG UART | clk | 0x021208B0 | 0x021208B7 | 4 |
| <code>button_pio</code> | PIO (Parallel I/O) | clk | 0x02120860 | 0x0212086F | 2 |
| <code>led_pio</code> | PIO (Parallel I/O) | clk | 0x02120870 | 0x0212087F | 1 |
| <code>high_res_timer</code> | Interval timer | clk | 0x02120820 | 0x0212083F | 3 |
| <code>lcd_display</code> | Character LCD (16x2, O... | clk | 0x02120880 | 0x0212088F | |
| <code>epcs_controller</code> | EPCS Serial Flash Contr... | clk | 0x02100000 | 0x021007FF | |

Reset Distribution

The Avalon switch fabric generates and distributes a system-wide reset pulse to all logic in the system module. The switch fabric distributes the reset signal conditioned for each clock domain. The duration of the reset signal is at least one clock period.

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

- The global reset input to the system module is asserted.
- A slave port asserts its `resetrequest` signal.
- The FPGA is reconfigured.

All components must enter a well-defined reset state whenever the Avalon switch fabric asserts the system-wide reset. The timing of the reset signal is asynchronous to the operation of transfers.

