Qsys System Design Components 10

2014.06.30

QII51025 Subscribe Send Feedback

You can use Qsys IP components to create Qsys systems. Qsys interfaces include components appropriate for streaming high-speed data, reading and writing registers and memory, controlling off-chip devices, and transporting data between components.

Qsys supports Avalon®, AMBA® AXI3™ (version 1.0), AMBA AXI4™ (version 2.0), AMBA AXI4-Lite™ (version 2.0), AMBA AXI4-Stream (version 1.0), and AMBA APB™ 3 (version 1.0) interface specifications.

Related Information

- Avalon Interface Specifications
- AMBA Protocol Specifications
- Creating a System with Qsys
- Qsys Interconnect
- Embedded Peripherals IP User Guide

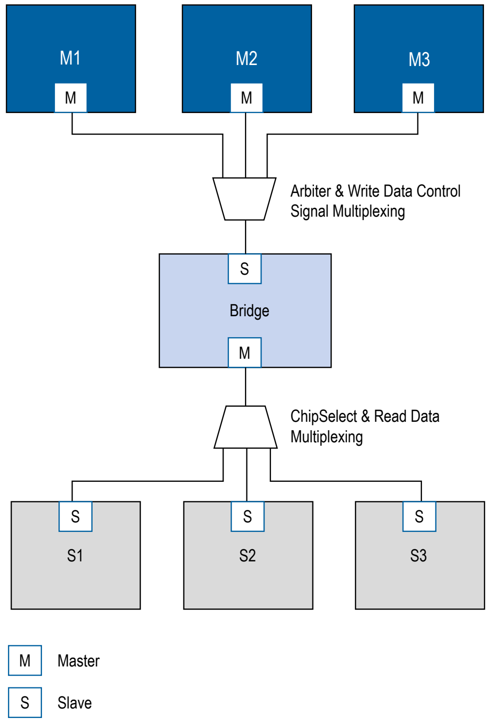
Bridges

Bridges affect the way Qsys transports data between components. You can insert bridges between masters and slave interfaces to control the topology of a Qsys system, which affects the interconnect that Qsys generates. You can also use bridges to separate components into different clock domains to isolate clock domain crossing logic.

A bridge has one slave interface and one master interface. In Qsys, one or more master interfaces from other components connect to the bridge slave. The bridge master connects to one or more slave interfaces on other components.

Figure 10-1: Using a Bridge in a Qsys System

In this example, three masters have logical connections to three slaves, although physically each master connects only to the bridge. Transfers initiated to the slave propagate to the master in the same order in which they are initiated on the slave.

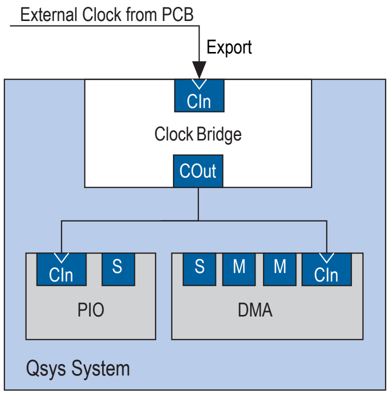


Clock Bridge

The Clock Bridge allows you to connect a clock source to multiple clock input interfaces. You can use the clock bridge to connect a clock source that is outside the Qsys system. You create the connection through an exported interface, and then connect to multiple clock input interfaces.

Clock outputs have the ability to fan-out without the use of a bridge. You require a bridge only when you want a clock from an exported source to connect internally to more than one source.

Figure 10-2: Clock Bridge



Avalon-MM Clock Crossing Bridge

The Avalon-MM Clock Crossing Bridge transfers Avalon-MM commands and responses between different clock domains. You can also use the Avalon-MM Clock Crossing Bridge between AXI masters and slaves of different clock domains.

The Avalon-MM Clock Crossing Bridge uses asynchronous FIFOs to implement clock crossing logic. The bridge parameters control the depth of the command and response FIFOs in both the master and slave clock domains. If the number of active reads exceeds the depth of the response FIFO, the Clock Crossing Bridge stops sending reads.

To maintain throughput for high-performance applications, increase the response FIFO depth from the default minimum depth, which is twice the maximum burst size.

Related Information

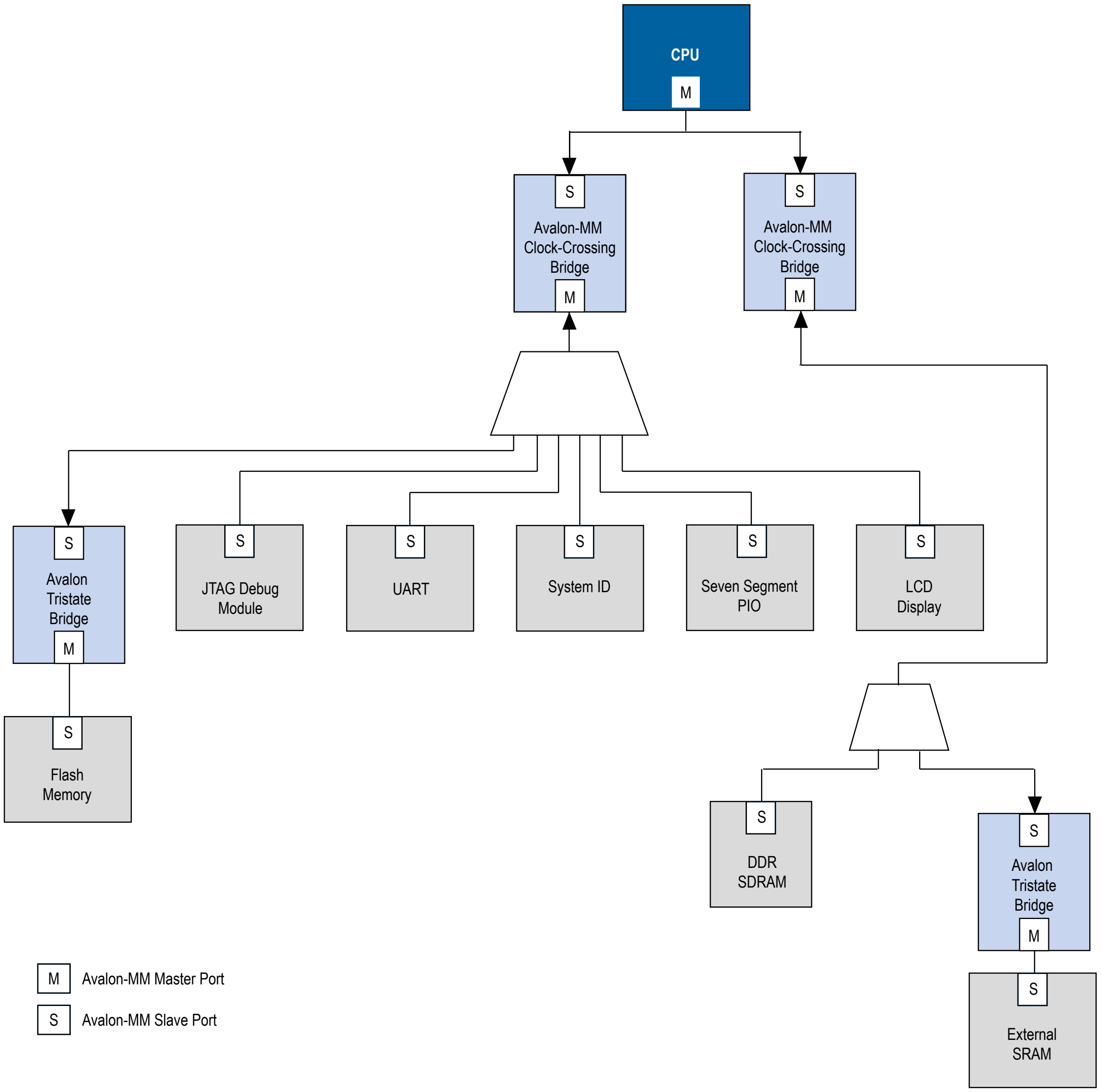
Creating a System with Qsys

Avalon-MM Clock Crossing Bridge Example

In this example, the Avalon-MM Clock Crossing bridges separate slave components into two groups. Low-performance slave components are placed behind a single bridge and are clocked at a low speed. High performance components are placed behind a second bridge and are clocked at a higher speed.

By inserting clock-crossing bridges, you optimize the Qsys interconnect and allow the Quartus® II Fitter to optimize paths that require minimal propagation delay.

Figure 10-3: Avalon-MM Clock Crossing Bridge



Avalon-MM Clock Crossing Bridge Parameters

Table 10-1: Avalon-MM Clock Crossing Bridge Parameters

Parameters	Values	Description
Data width	8, 16, 32, 64, 128, 256,512, 1024 (bits)	Determines the data width of the interfaces on the bridge, and affects the size of both FIFOs. For the highest bandwidth, set **Data width** to be as wide as the widest master that connects to the bridge.

Parameters	Values	Description
Symbol width	1, 2, 4, 8, 16, 32, 64 (bits)	Number of bits per symbol. For example, byte-oriented interfaces have 8-bit symbols.
Address width	1-32 (bits)	The address bits needed to address the downstream slaves.
Use automatically-determined address width	-	The minimum bridge address width that is required to address the downstream slaves.
Maximum burst size	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 (bits)	Determines the maximum length of bursts that the bridge supports.
Command FIFO depth	2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 2048, 4096, 8192, 16384 (bits)	Command (master-to-slave) FIFO depth.
Respond FIFO depth	2, 4, 8,16, 32, 64, 128, 256, 512, 1024 2048, 4096, 8192,16384 (bits)	Response (slave-to-master) FIFO depth.
Master clock domain synchronizer depth	2, 3, 4, 5 (bits)	The number of pipeline stages in the clock crossing logic in the issuing master to target slave direction. Increasing this value leads to a larger meantime between failures (MTBF). You can determine the MTBF for a design by running a TimeQuest timing analysis.
Slave clock domain synchronizer depth	2, 3, 4, 5 (bits)	The number of pipeline stages in the clock crossing logic in the target slave to master direction. Increasing this value leads to a larger meantime between failures (MTBF). You can determine the MTBF for a design by running a TimeQuest timing analysis.

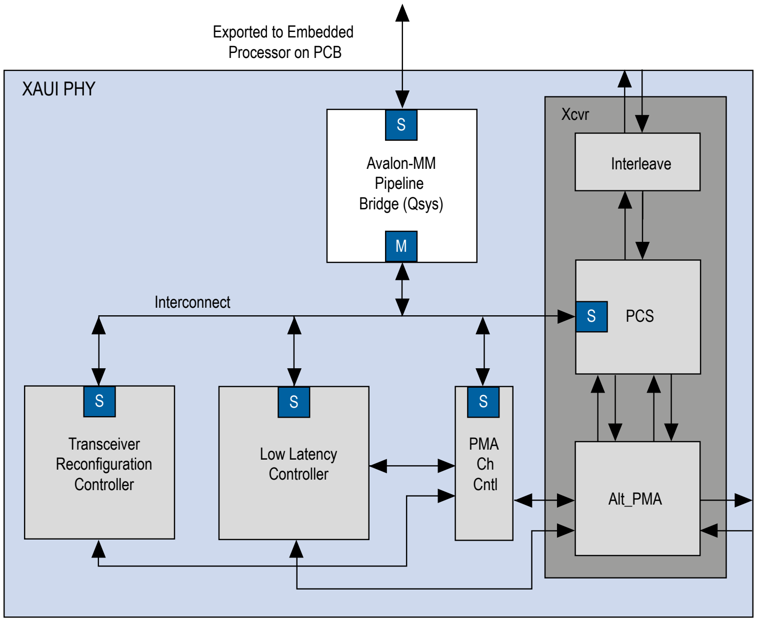
Avalon-MM Pipeline Bridge

The Avalon-MM Pipeline Bridge inserts a register stage in the Avalon-MM command and response paths. It accepts commands on its Avalon-MM slave port and propagates the commands to its Avalon-MM master port. The pipeline bridge provides separate parameters to turn on pipelining in the command and response networks.

You can use the Avalon-MM bridge to export a single Avalon-MM slave interface to use to control multiple Avalon-MM slave devices. The pipelining feature is optional. You can optionally turn off the pipelining feature of this bridge.

Figure 10-4: Avalon-MM Pipeline Bridge in a XAUI PHY Transceiver IP Core

In this example, the bridge transfers commands received on its slave interface to its master port.

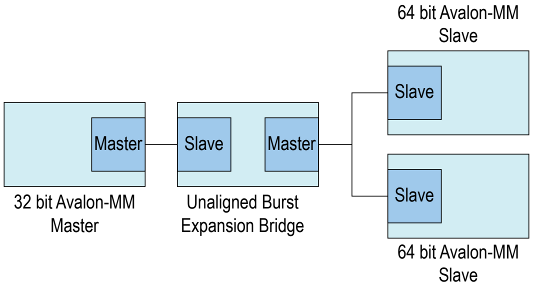


Because the slave interface is exported to the pins of the device, having a single slave port, rather than separate ports for each slave device, reduces the pin count of the FPGA.

Avalon-MM Unaligned Burst Expansion Bridge

The Avalon-MM Unaligned Burst Expansion Bridge aligns read burst transactions from masters connected to its slave interface, to the address space of slaves connected to its master interface. This alignment ensures that all read burst transactions are delivered to the slave as a single transaction.

Figure 10-5: Avalon-MM Unaligned Burst Expansion Bridge



You can use the Avalon Unaligned Burst Expansion Bridge to align read burst transactions from masters that have narrower data widths than the target slaves. Using the bridge for this purpose improves bandwidth utilization for the master-slave pair, and ensures that un-aligned bursts are processed as single transactions rather than multiple transactions.

Note: Do not use the Avalon-MM Unaligned Burst Expansion Bridge if any connected slave has read side effects from reading addresses that are exposed to any connected master's address map. This bridge can cause read side effects due to alignment modification to read burst transaction addresses.

Note: For Qsys 14.0, the Avalon-MM Unaligned Burst Expansion Bridge does not support VHDL simulation.

Related Information

Qsys Interconnect

Using the Avalon-MM Unaligned Burst Expansion Bridge

When a master sends a read burst transaction to a slave, the Avalon-MM Unaligned Burst Expansion Bridge initially determines whether the start address of the read burst transaction is aligned to the slave's memory address space. If the base address is aligned, the bridge does not change the base address. If the base address is not aligned, the bridge aligns the base address to the nearest aligned address that is less than the requested base address.

The Avalon-MM Unaligned Burst Expansion Bridge then determines whether the final word requested by the master is the last word at the slave read burst address. If a single slave address contains multiple words, all of those words must be requested in order for a single read burst transaction to occur.

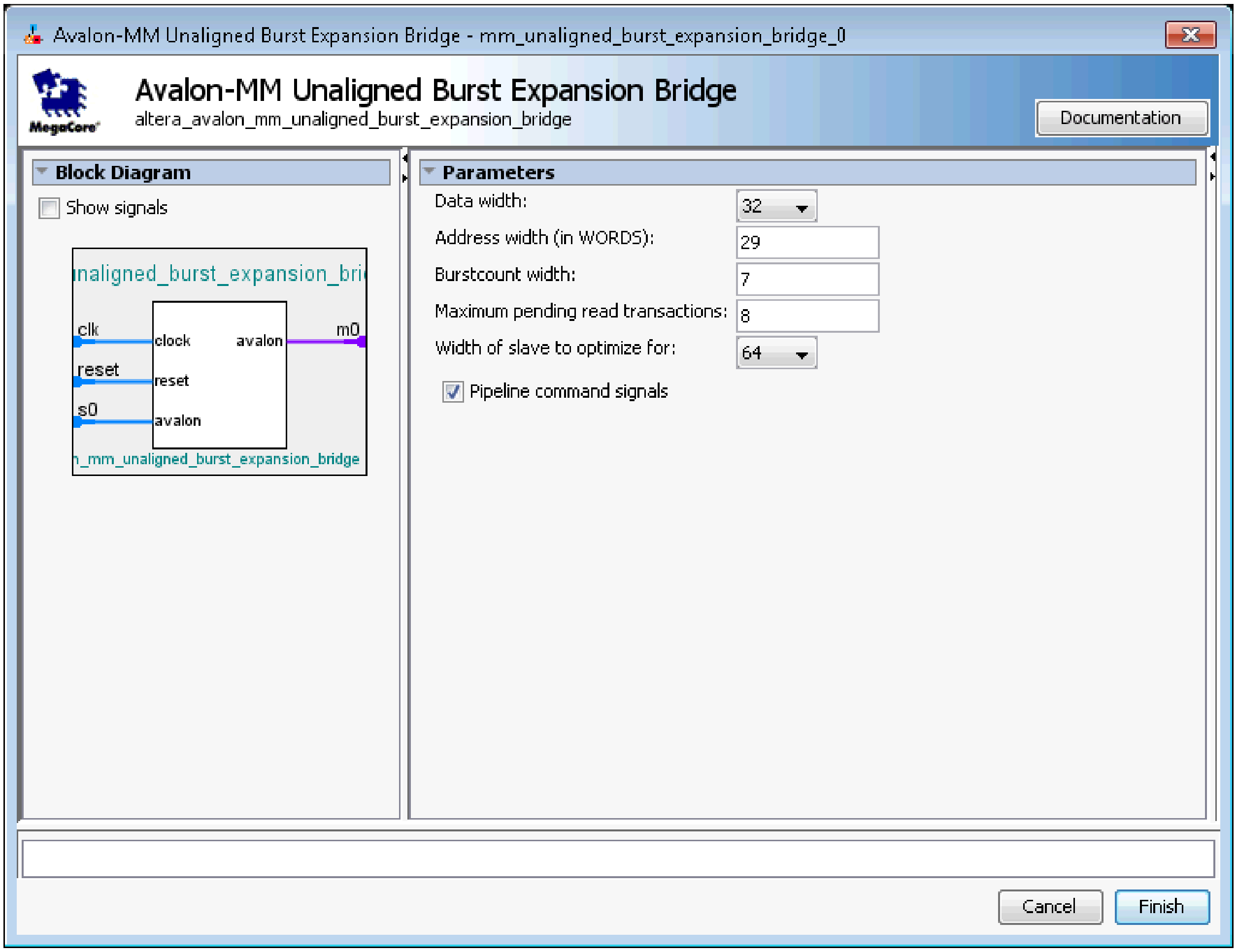
- If the final word requested by the master is the last word at the slave read burst address, the bridge does not modify the burst length of the read burst command to the slave.
- If the final word requested by the master is not the last word at the slave read burst address, the bridge increases the burst length of the read burst command to the slave. The final word requested by the modified read burst command is then the last word at the slave read burst address.

The bridge stores information about each aligned read burst command that it sends to slaves connected to a master interface. When a read response is received on the master interface, the bridge determines if the base address or burst length of the issued read burst command was altered.

If the bridge alters either the base address or the burst length of the issued read burst command, it receives response words that the master did not request. The bridge suppresses words that it receives from the aligned burst response that are not part of the original read burst command from the master.

Avalon-MM Unaligned Burst Expansion Bridge Parameters

Figure 10-6: Avalon-MM Unaligned Burst Expansion Bridge Parameter Editor



Avalon-MM Unaligned Burst Expansion Bridge - mm_unaligned_burst_expansion_bridge_0

Avalon-MM Unaligned Burst Expansion Bridge

altera_avalon_mm_unaligned_burst_expansion_bridge

Data width: 32

Address width (in WORDS): 29

Burstcount width: 7

Maximum pending read transactions: 8

Width of slave to optimize for: 64

☑ Pipeline command signals

Table 10-2: Avalon-MM Unaligned Burst Expansion Bridge Parameters

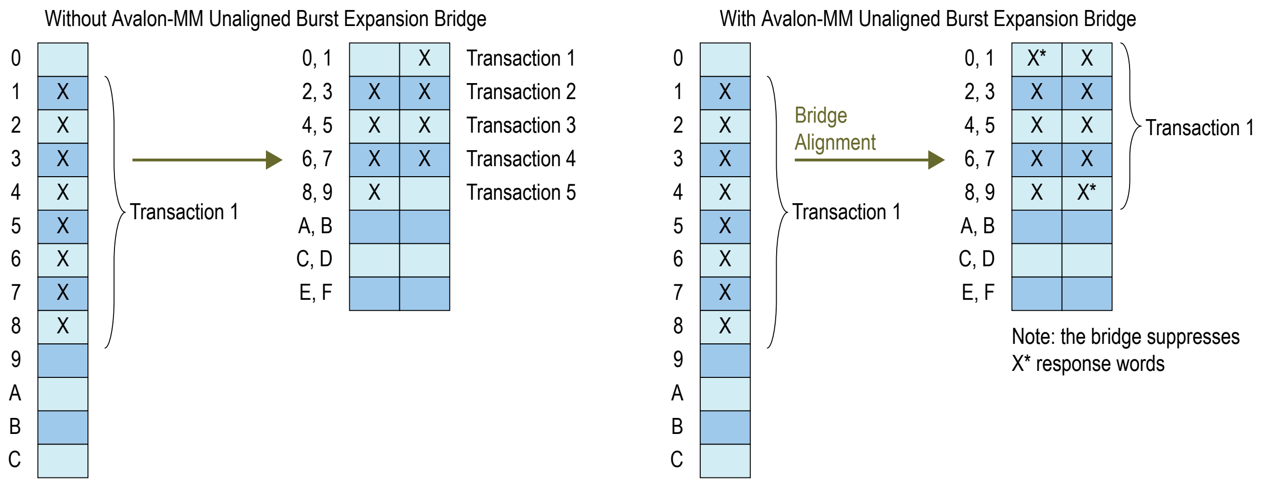
Parameter	Description
Data width	Data width of the master connected to the bridge.
Address width (in WORDS)	The address width of the master connected to the bridge.
Burstcount width	The burstcount signal width of the master connected to the bridge.
Maximum pending read transactions	The maximum pending read transactions interface property of the bridge.
Width of slave to optimize for	The data width of the connected slave. Supported values are: 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096 bits. **Note:** If you connect multiple slaves, all slaves must have the same data width.

Parameter	Description
Pipeline command signals	When turned on, the command path is pipelined, minimizing the bridge's critical path at the expense of increased logic usage and latency.

Avalon-MM Unaligned Burst Expansion Bridge Example

Figure 10-7: Unaligned Burst Expansion Bridge

The example below shows an unaligned read burst command from a master that the Avalon-MM Unaligned Burst Expansion Bridge converts to an aligned request for a connected slave, and the suppression of words due to the aligned read burst command. In this example, a 32-bit master requests an 8-beat burst of 32-bit words from a 64-bit slave with a start address that is not 64-bit aligned.



Because the target slave has a 64-bit data width, address 1 is unaligned in the slave's address space. As a result, several smaller burst transactions are needed to request the data associated with the master's read burst command.

With an Avalon-MM Unaligned Burst Expansion Bridge in place, the bridge issues a new read burst command to the target slave beginning at address 0 with burst length 10, which requests data up to the word stored at address 9.

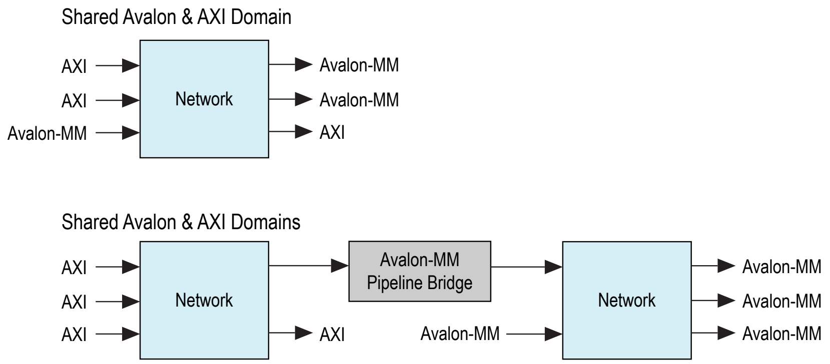
When the bridge receives the word corresponding to address 0, it suppresses it from the master, and then delivers the words corresponding to addresses 1 through 8 to the master. When the bridge receives the word corresponding to address 9, it suppresses that word from the master.

Bridges Between Avalon and AXI Interfaces

When designing a Qsys system, you can make connections between AXI and Avalon interfaces without the use of explicitly-instantiated bridges; the interconnect provides all necessary bridging logic. However, this does not prevent the use of explicit bridges to separate the AXI and Avalon domains.

Figure 10-8: Avalon-MM Pipeline Bridge Between Avalon-MM and AXI Domains

Using an explicit Avalon-MM bridge to separate the AXI and Avalon domains reduces the amount of bridging logic in the interconnect at the expense of concurrency.



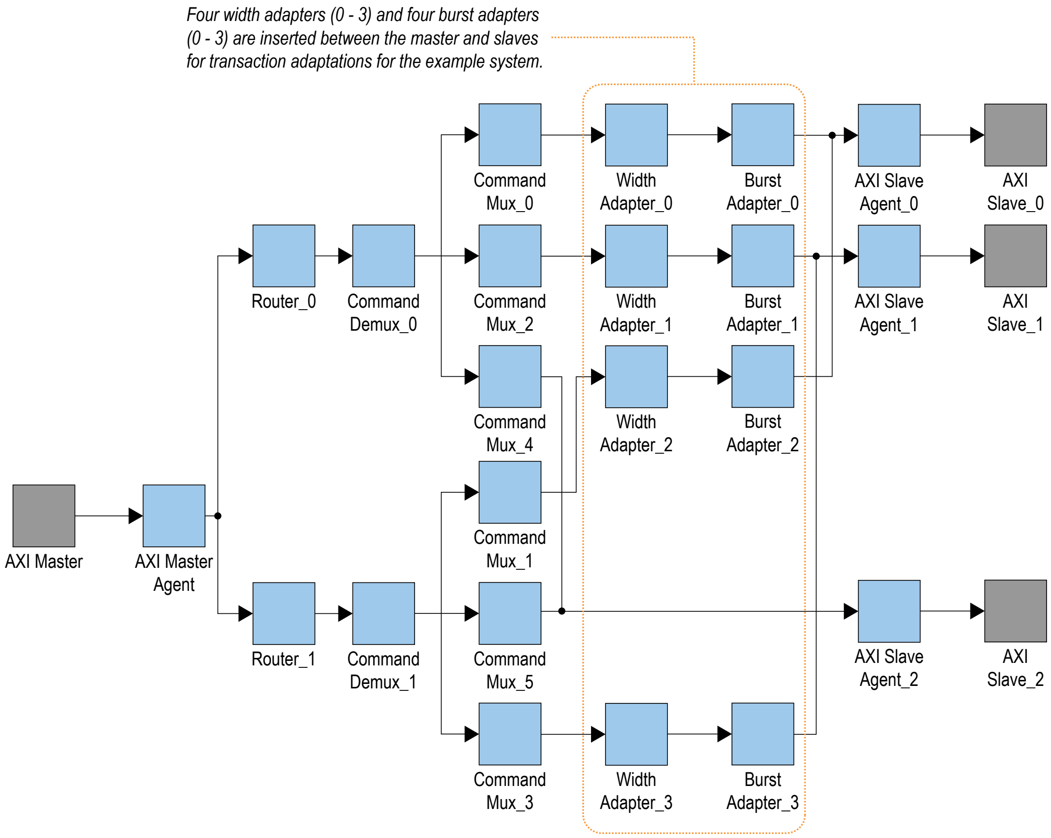
AXI Bridge

With an AXI bridge, you can influence the placement of resource-intensive components, such as the width and burst adapters. Depending on its use, an AXI bridge may reduce throughput and concurrency, in return for higher f_{Max} and less logic.

You can use an AXI bridge to group different parts of your Qsys system. Then, other parts of the system connect to the bridge interface instead of to multiple separate master or slave interfaces. You can also use an AXI bridge to export AXI interfaces from Qsys systems.

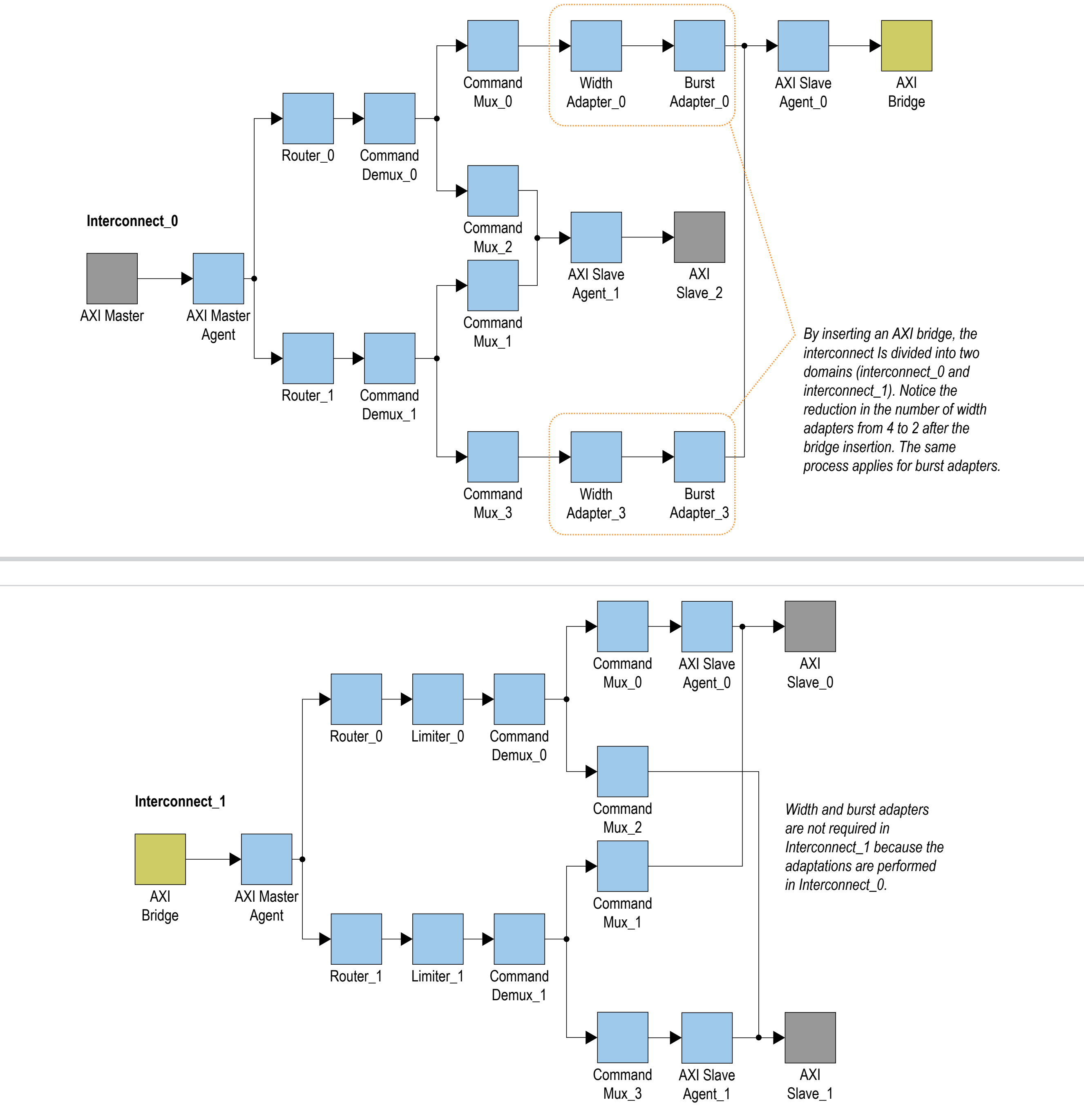
The example below shows a system with a single AXI master and three AXI slaves. It also has various interconnect components, such as routers, demuxes, and muxes. Two of the slaves have a narrower data width than the master; 16-bit slaves versus a 32-bit master. In this system, Qsys interconnect creates four width adapters and four burst adapters to access the two slaves. In this case, you could improve resource usage by adding an AXI bridge. This would result in Qsys having to add only two width adapters and two burst adapters, one pair for the read channels, and another pair for the write channel.

Figure 10-9: AXI Example Without a Bridge: Adding a Bridge Can Reduce the Number of Adapters



The example below shows the same system with an AXI bridge component, and the decrease in the number of width and burst adapters. Qsys creates only two width adapters, and two burst adapters, as compared to the four width adapters and four burst adapters in the previous example. The system includes more components, but the overall system performance improves because there are fewer resource-intensive width and burst adapters.

Figure 10-10: Width and Burst Adapters Added to a System With a Bridge



By inserting an AXI bridge, the interconnect Is divided into two domains (interconnect_0 and interconnect_1). Notice the reduction in the number of width adapters from 4 to 2 after the bridge insertion. The same process applies for burst adapters.

Width and burst adapters are not required in Interconnect_1 because the adaptations are performed in Interconnect_0.

AXI Bridge Signal Types

Based on parameter selections that you make for the AXI Bridge component, Qsys instantiates either the AXI3 or AXI4 master and slave interfaces into the component.

Note: In AXI3, `aw/aruser` accommodates sideband signal usage by hard processor systems (HPS).

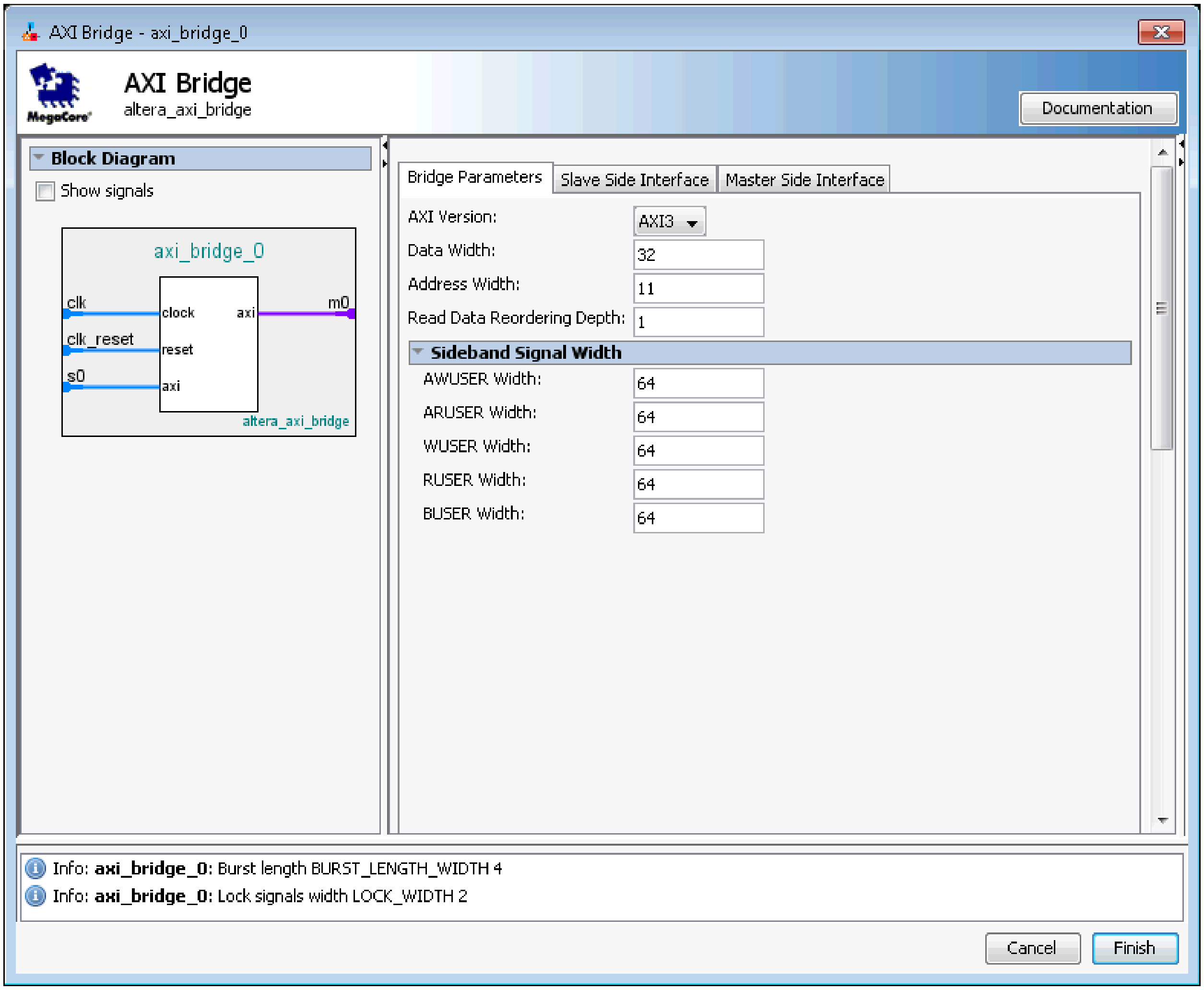
Table 10-3: Sets of Signals for the AXI Bridge Based on the Protocol

Signal Name	AXI3	AXI4
`awid / arid`	yes	yes
`awaddr /araddr`	yes	yes
`awlen / arlen`	yes (4-bit)	yes (8-bit)
`awsize/ arsize`	yes	yes
`awburst /arburst`	yes	yes
`awlock /arlock`	yes	yes (1-bit optional)
`awcache / arcache`	yes (2-bit)	yes (optional)
`awprot / arprot`	yes	yes
`awuser /aruser`	yes	yes
`awvalid / arvalid`	yes	yes
`awready /arready`	yes	yes
`awqos /arqos`	no	yes
`awregion /arregion`	no	yes
`wid`	yes	no (optional)
`wdata / rdata`	yes	yes
`wstrb`	yes	yes
`wlast /rvalid`	yes	yes
`wvalid /rlast`	yes	yes
`wready /rready`	yes	yes
`wuser / ruser`	no	yes
`bid / rid`	yes	yes
`bresp / rresp`	yes	yes (optional)
`bvalid`	yes	yes
`bready`	yes	yes

AXI Bridge Parameters

In the parameter editor, you can customize the parameters for the AXI bridge according to the requirements of your design.

Figure 10-11: AXI Bridge Parameter Editor



AXI Version: AXI3
Data Width: 32
Address Width: 11
Read Data Reordering Depth: 1

Sideband Signal Width

AWUSER Width: 64
ARUSER Width: 64
WUSER Width: 64
RUSER Width: 64
BUSER Width: 64

Info: **axi_bridge_0:** Burst length BURST_LENGTH_WIDTH 4
Info: **axi_bridge_0:** Lock signals width LOCK_WIDTH 2

Table 10-4: AXI Bridge Parameters

Parameter	Type	Range	Description
AXI Version	string	AXI3/ AXI4	Specifies the AXI version and signals that Qsys generates for the slave and master interfaces of the bridge.
Data Width	int	8:1024	Controls the width of the data for the master and slave interfaces.
Address Width	int	1-64 bits	Controls the width of the address for the master and slave interfaces.

Parameter	Type	Range	Description
Read Data Reordering Depth	int	1-16	Controls the multithreading feature and out-of-order responses. If a master issues different thread IDs to different slaves, in order for a slave to view the different thread IDs, you must set the **Read Data Reordering Depth** to 1.
AWUSER Width	int	1-64 bits	Controls the width of the write address channel sideband signals of the master and slave interfaces.
ARUSER Width	int	1-64 bits	Controls the width of the read address channel sideband signals of the master and slave interfaces.
WUSER Width	int	1-64 bits	Controls the width of the write data channel sideband signals of the master and slave interfaces.
RUSER Width	int	1-16 bits	Controls the width of the read data channel sideband signals of the master and slave interfaces.
BUSER Width	int	1-16 bits	Controls the width of the write response channel sideband signals of the master and slave interfaces.

AXI Bridge Slave and Master Interface Parameters

Table 10-5: AXI Bridge Slave and Master Interface Parameters

Parameter	Description
ID Width	Controls the width of the thread ID of the master and slave interfaces.
Write/Read/Combined Acceptance Capability	Controls the depth of the FIFO that Qsys needs in the interconnect agents based on the maximum pending commands that the slave interface accepts.
Write/Read/Combined Issuing Capability	Controls the depth of the FIFO that Qsys needs in the interconnect agents based on the maximum pending commands that the master interface issues. Issuing capability must follow acceptance capability to avoid unnecessary creation of FIFOs in the bridge.

Note: Maximum acceptance/issuing capability is a model-only parameter and does not influence the bridge HDL. The bridge does not backpressure when this limit is reached. Downstream components and/or the interconnect must apply backpreasure.

Address Span Extender

The Address Span Extender creates a windowed bridge and allows memory-mapped master interfaces to access a larger or smaller address map than the width of their address signals allow. With an address span extender, a restricted master can access a broader address range. The address span extender splits the addressable space into multiple separate windows so that the master can access the appropriate part of the memory through the window.

The address span extender does not limit master and slave widths to a 32-bit and 64-bit configuration. You can use the address span extender for other width configurations. The address span extender supports 1-64 bit address windows.

If a processor can address only 2GB of an address span, and your system contains 4GB of memory, the address span extender can provide two 2GB windows in the 4GB memory address space. This issue sometimes occurs with Altera SoC devices. For example, an HPS subsystem in an SoC device can address only 1GB of an address span within the FPGA using the HPS-to-FPGA bridge. The address span extender enables the SoC device to address all of the address space in the FPGA using multiple 1GB windows.

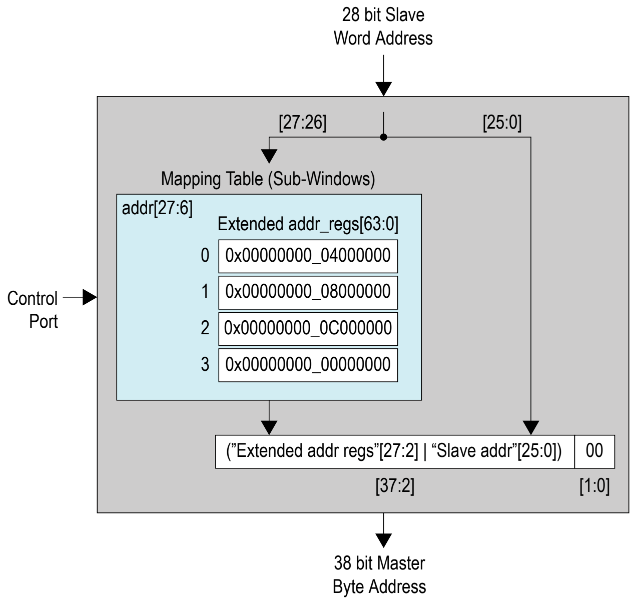
CTRL Register Layout

The control registers consist of a 64-bit register for each window. You write the base address that you want for each window to its corresponding control register. For example, if `CTRL_BASE` is the base address of the address span extender's control register, and there are two windows (0 and 1), then window 0's control register starts at `CTRL_BASE`, and window 1's control register starts at `CTRL_BASE` + 8 (using byte addresses).

Calculating the Address Span Extender Slave Address

The diagram below describes how Qsys calculates the slave address. In this example the address, span extender is configured with a 28-bit address space for slaves. The lower 26 bits (bits 0 to 25 or `[25:0]`) is the offset into a particular window and originate from the address span extender's data port. The upper 2 bits `[27:26]` originate from the control registers.

Figure 10-12: Address Span Extender



Using the Address Span Extender

When you implement the address span extender in Qsys, you must know the amount of address space the master uses (the size of the window), the total size of the addressable space (the number of windows), and how much address space (the size of the window) you want a particular slave to occupy in a master's address map.

This component supports 1 to 64 address windows. Qsys requires an assigned number of registers to hold the upper address bits for each window. In the parameter editor, you must select the number of bits in the expanded address map you want to access (**Expanded Master Byte Address Width**), the number of bits you want the master to see (**Slave Word Address Width**), and the number of sub-windows.

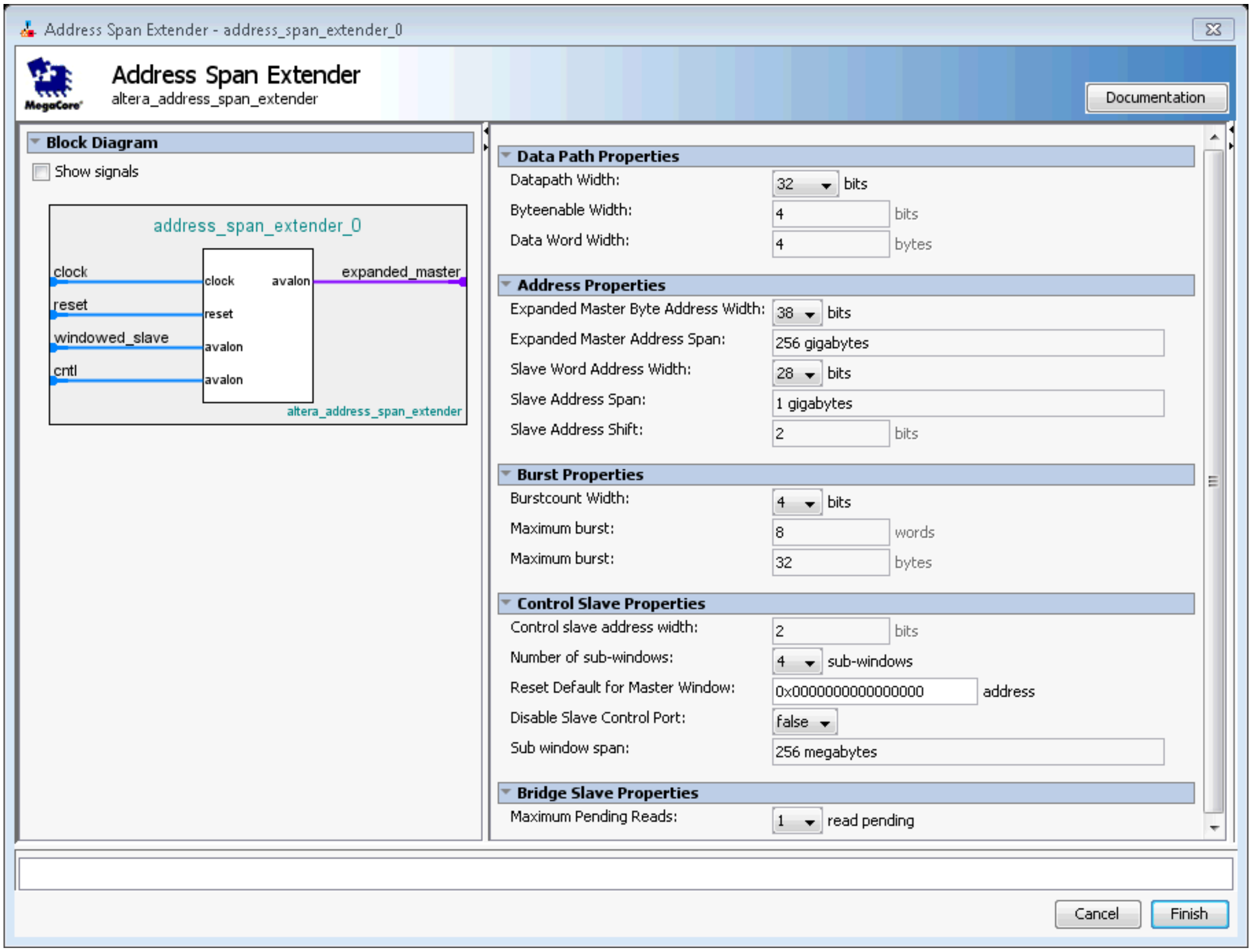
Each sub-window has a 64-bit register set that defines the sub window's upper address, and use only the bits greater than the slave byte address.

- **window 0**—expanded address `[63:0]`
- **window 1**—expanded address `[63:0]`

Qsys uses the upper bits of the slave address to pick which window to use. For example, if you specify 4 windows, then Qsys uses the top 2 bits of the slave address to specify window `[0,1,2,3]`. Therefore having more windows does require the windows to be smaller, for example having 4 windows requires the windows themselves to be 1/4 the size of the slave address space. The total windowed address space is still equal to the original slave address space, but the windows allow access to memory regions in a larger overall address space.

In the parameter editor for the address span extender, you can click **Documentation** to obtain more information about the component.

Figure 10-13: Address Span Extender Parameter Editor



Data Path Properties

Datapath Width: 32 bits
Byteenable Width: 4 bits
Data Word Width: 4 bytes

Address Properties

Expanded Master Byte Address Width: 38 bits
Expanded Master Address Span: 256 gigabytes
Slave Word Address Width: 28 bits
Slave Address Span: 1 gigabytes
Slave Address Shift: 2 bits

Burst Properties

Burstcount Width: 4 bits
Maximum burst: 8 words
Maximum burst: 32 bytes

Control Slave Properties

Control slave address width: 2 bits
Number of sub-windows: 4 sub-windows
Reset Default for Master Window: 0x0000000000000000 address
Disable Slave Control Port: false
Sub window span: 256 megabytes

Bridge Slave Properties

Maximum Pending Reads: 1 read pending

Alternate Options for the Address Span Extender

You can set parameters for the address span extender with an initial fixed address value. Enter an address for the **Reset Default for Master Window** option, and select **True** for the **Disable Slave Control Port** option. This allows the address span extender to function as a fixed, non-programmable component.

Each sub-window is equal in size and stacks sequentially in the windowed slave interface's address space. To control the fixed address bits of a particular sub-window, you can write to the sub-window's register in the register control slave interface. Qsys structures the logic so that Qsys can optimize and remove bits that are not needed.

If **Burstcount Width** is greater than 1, Qsys processes the read burst in a single cycle, and assumes all byteenables are asserted on every cycle.

NIOS II Support

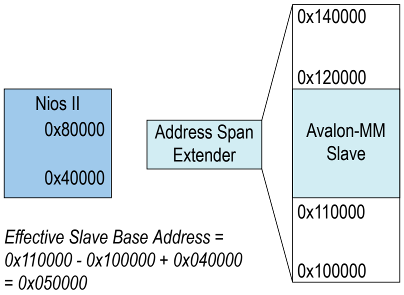
If the address span extender window is fixed, for example, the **Disable Slave Control Port** option is turned on, then the address span extender performs as a bridge. Components on the slave side of the address span extender that are within the window are visible to the NIOS II processor. Components partially within a window appear to NIOS II as if they have a reduced span. For example, a memory partially within a window appears as having a smaller size.

You can also use the address span extender to provide a window for the Nios II processor so that the HPS memory map is visible to NIOS II. In this way it is possible for the Nios II to communicate with HPS peripherals.

In the example below, a NIOS II processor has an address span extender from address `0x40000` to `0x80000`. There is a window within the address span extender starting at `0x100000`. Within the address span extender's address space there is a slave at base address `0x1100000`. The slave appears to NIOS II as being at address:

```
0x110000 - 0x100000 + 0x40000 = 0x050000
```

Figure 10-14: NIOS II Support and the Address Span Extender



If the address span extender window is dynamic. For example, when the **Disable Slave Control Port** option is turned off, the NIOS II processor is unable to see components on the slave side of the address span extender.

AXI Default Slave

An AXI Default Slave provides a predictable error response service for master interfaces that send transactions that attempt to access an undefined memory region. This service guarantees an error response, should a master access a memory region that is not decoded to an instantiated slave. The error response service also helps to avoid unpredictable behavior in your system.

The default slave is an AXI3 component and displays in the IP Catalog as either **AXI Default Slave** or **Error Response Slave**.

AXI protocol requires that if the interconnect cannot successfully decode slave access, it must return the `DECERR` error response. Therefore, the default slave is required in AXI systems where the address space is not fully decoded to slave interfaces.

The default slave behaves like any other component in the system and is bound by translation and adaptation interconnect logic. An increase in resource usage may occur when a default slave connects to masters of different data widths, including Avalon or AXI-Lite masters.

You can connect clock, reset, and IRQ signals to a default slave, as well as AXI3 and AXI4 master interfaces without also instantiating a bridge. When you connect a default slave to a master, the default slave accepts cycles sent from the master, and returns the `DECERR` error response. On the AXI interface, the default slave supports only a read and write acceptance of 1, and does not support write data interleaving. The read and write channels are independent, and responses are returned when simultaneously targeted by a read and write cycle.

There is an optional interface on the default slave that supports CSR accesses for debug. CSR registers log the required information when returning an error response. When turned on, this channel acts as an Avalon interface with read and write channels with a fixed latency of 1.

To enable a slave interface as a default slave for a master interface in your system, you must connect the slave to the master in your Qsys system. You specify a default slave for a master it by turning on the **Default Slave** column option in the **System Contents** tab. A system can contain more than one default slave. Altera recommends instantiating a separate default slave for each AXI master in your system.

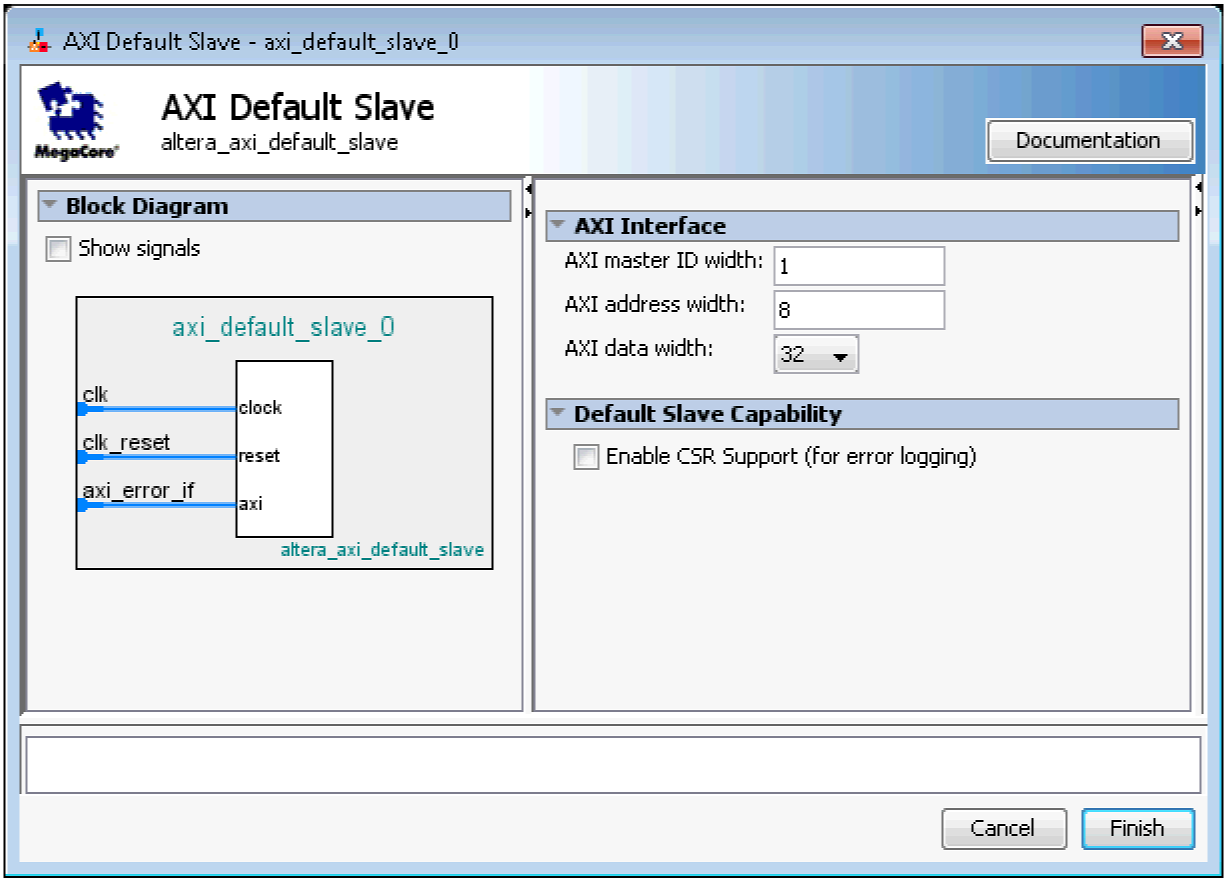
For information about creating secure systems and accessing undefined memory regions, refer to *Creating a System with Qsys* in volume 1 of the *Quartus II Handbook*.

Related Information

Creating a System with Qsys

AXI Default Slave Parameters

Figure 10-15: AXI Default Slave Parameter Editor



AXI Default Slave - axi_default_slave_0

AXI Default Slave
altera_axi_default_slave

Block Diagram

☐ Show signals

AXI Interface

AXI master ID width: 1

AXI address width: 8

AXI data width: 32

Default Slave Capability

☐ Enable CSR Support (for error logging)

Table 10-6: AXI Default Slave Parameters

Parameter	Value	Description
AXI master ID width	1-8 bits	Determines the master ID width for error logging.

Parameter	Value	Description
AXI address width	8-64 bits	Determines the address width for error logging. This value also affects the overall address width of the system, and should not exceed the maximum address width required in the system.
AXI data width	32, 64, or128 bits	Determines the data width for error logging.
Enable CSR Support (for error logging)	On or Off	When turned on, instantiates an Avalon CSR interface for error logging.
CSR Error Log Depth	1-16 bits	Depth of the transaction log, for example, the number of transactions the CSR logs for cycles with errors.
Register Avalon CSR inputs	On or Off	When turned on, controls debug access to the CSR interface.

CSR Registers

When an access violation occurs, and the CSR port is enabled, the AXI Default Slave generates an interrupt and transfers the transaction information into the error log FIFO.

The error log count continues until the n^{th} log, where *n* is the log depth. When Qsys responds to the interrupt bit, it reads the register until the interrupt bit is no longer valid. The interrupt bit is valid as long as there is a valid bit in FIFO. A cleared interrupt bit is not affected by the FIFO status. When Qsys finishes reading the register, the access violation service is ready to receive new access violation requests. If an access violation occurs when FIFO is full, then an overflow bit is set, indicating more than *n* access violations have occurred, and some are not logged.

Qsys exits the access violation service after either the interrupt bit is no longer set, or when it determines that the access violation service has continued for too long.

CSR Interrupt Status Registers

Table 10-7: CSR Interrupt Status Registers

For CSR register maps: `Address = Memory Address Base + Offset`.

Offset	Bit	Attribute	Default	Descripton
`0x00`	`31:4`	R0	0	Reserved.
	3	RW1C	0	**Read Access Violation Interrupt Overflow register** Asserted when a read access causes the Interconnect to return a `DECERR` response, and the buffer log depth is full. Indicates that there is a logging error lost due to an exceeded buffer log depth. Cleared by setting the bit to 1.

Offset	Bit	Attribute	Default	Descripton
	2	RW1C	0	**Write Access Violation Interrupt Overflow register** Asserted when a write access causes the Interconnect to return a `DECERR` response, and the buffer log depth is full. Indicates that there is a logging error lost due to an exceeded buffer log depth. Cleared by setting the bit to 1.
	1	RW1C	0	**Read Access Violation Interrupt register** Asserted when a read access causes the Interconnect to return a `DECERR` response. Cleared by setting the bit to 1. **Note:** Access violation are logged until the bit is cleared.
	0	RW1C	0	**Write Access Violation Interrupt register** Asserted when a write access causes the Interconnect to return a `DECERR` response. Cleared by setting the bit to 1. **Note:** Access violation are logged until the bit is cleared.

CSR Read Access Violation Log

The CSR read access violation log settings are valid only when an associated read interrupt register is set. This set of registers should be read until the valid bit is cleared.

Table 10-8: CSR Read Access Violation Log

Offset	Bit	Attribute	Default	Description
`0x100`	31:13	R0	0	Reserved.
	12:11	R0	0	Indicates the burst type of the initiating cycle that causes the access violation.
	10:7	R0	0	Indicates the burst length of the initiating cycle that causes the access violation.
	6:4	R0	0	Indicates the burst size of the initiating cycle that causes the access violation.
	3:1	R0	0	Indicates the `PROT` of the initiating cycle that causes the access violation.

Offset	Bit	Attribute	Default	Description
	0	RO	0	Read access violation log for the transaction is valid only when this bit is set. This bit is cleared when the interrupt register is cleared.
`0x104`	31:0	RO	0	Master ID for the cycle that causes the access violation.
`0x108`	31:0	RO	0	Read cycle target address for the cycle that causes the access violation (lower 32-bit).
`0x10C`	31:0	RO	0	Read cycle target address for the cycle that causes the access violation (upper 32-bit). Valid only if widest address in system is larger than 32-bits. **Note:** When this register is read, the current read access violation log is recovered from FIFO.

CSR Write Access Violation Log

The CSR write access violation log settings are valid only when an associated read interrupt register is set. This set of registers should be read until the valid bit is cleared.

Table 10-9: CSR Write Access Violation Log

Offset	Bit	Attribute	Default	Description
`0x190`	`31:13`	RO	0	Reserved.
	12:11	RO	0	Indicates the burst type of the initiating cycle that causes the access violation.
	10:7	RO	0	Indicates the burst length of the initiating cycle that causes the access violation.
	6:4	RO	0	Indicates the burst size of the initiating cycle that causes the access violation.
	3:1	RO	0	Indicates the `PROT` of the initiating cycle that causes the access violation.
	0	RO	0	Write access violation log for the transaction is valid only when this bit is set. This bit is cleared when the interrupt register is cleared.
`0x194`	31:0	RO	0	Master ID for the cycle that causes the access violation.
`0x198`	31:0	RO	0	Write target address for the cycle that causes the access violation (lower 32-bit).
`0x19C`	31:0	RO	0	Write target address for the cycle that causes the access violation (upper 32-bit). Valid only if widest address in system is larger than 32-bits.

Offset	Bit	Attribute	Default	Description
`0x1A0`	31:0	RO	0	First 32-bits of the write data for the write cycle that causes the access violation. **Note:** When this register is read, the current write access violation log is recovered from FIFO, when the data width is 32-bits.
`0x1A4`	31:0	RO	0	Bits [`63:32`] of the write data for the write cycle that causes the access violation. Valid only if the data width is greater than 32 -bits.
`0x1A8`	31:0	RO	0	Bits [`95:64`] of the write data for the write cycle that causes the access violation. Valid only if the data width is greater than 64 -bits.
`0x1AC`	31:0	RO	0	The first bits (`127:96`) of the write data for the write cycle that causes the access violation. Valid only if the data width is greater than 64 -bits. **Note:** When this register is read, the current write access violation log is recovered from FIFO.

Designating a Default Slave in the System Contents Tab

You can designate any slave in your Qsys system as the error response default slave. The designated default slave provides an error response service for masters that attempt access to an undefined memory region.

1. In your Qsys system, in the **System Contents** tab, right-click the header and turn on **Show Default Slave Column**.
2. Select the slave that you want to designate as the default slave, and then click the checkbox for the slave in the **Default Slave** column.
3. In the **System Contents** tab, in the **Connections** column, connect the designated default slave to one or more masters.

Tri-State Components

The tri-state interface type allows you to design Qsys subsystems that connect to tri-state devices on your PCB. You can use tri-state components to implement pin sharing, convert between unidirectional and bidirectional signals, and create tri-state controllers for devices whose interfaces can be described using the tri-state signal types.

Figure 10-16: Tri-State Conduit System to Control Off-Chip SRAM and Flash Devices

In this example, there are two generic Tri-State Conduit Controllers. The first is customized to control a flash memory. The second is customized to control an off-chip SSRAM. The Tri-State Conduit Pin Sharer multiplexes between these two controllers, and the Tri-State Conduit Bridge converts between an on-chip encoding of tri-state signals and true bidirectional signals. By default, the Tri-State Conduit Pin Sharer and Tri-State Conduit Bridge present byte addresses. Typically, each address location contains more than one byte of data.

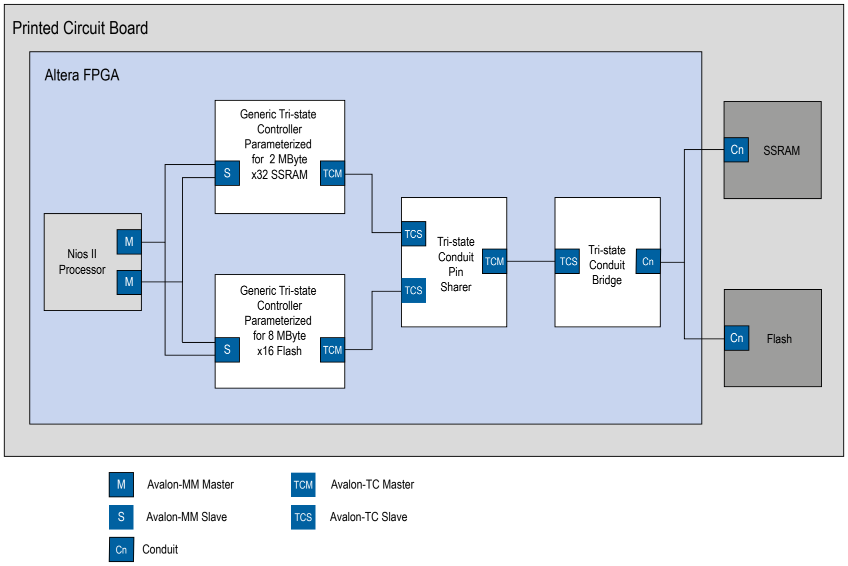
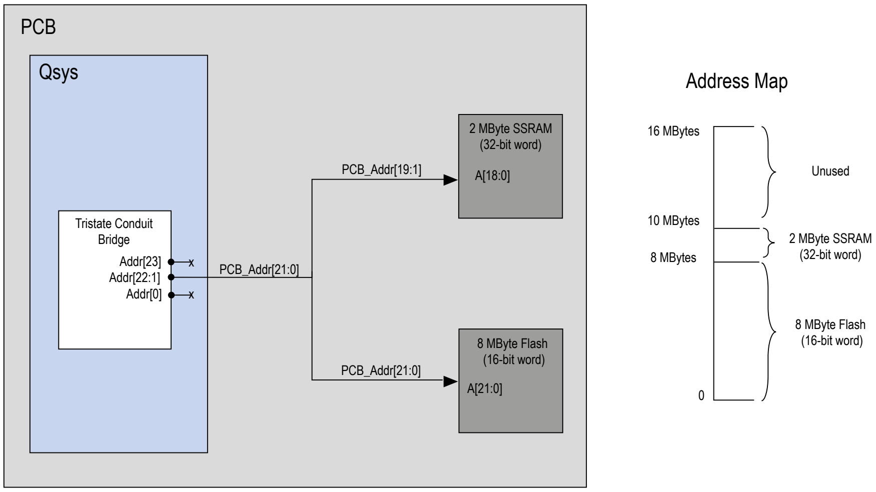


Figure 10-17: Address Connections from Qsys System to PCB

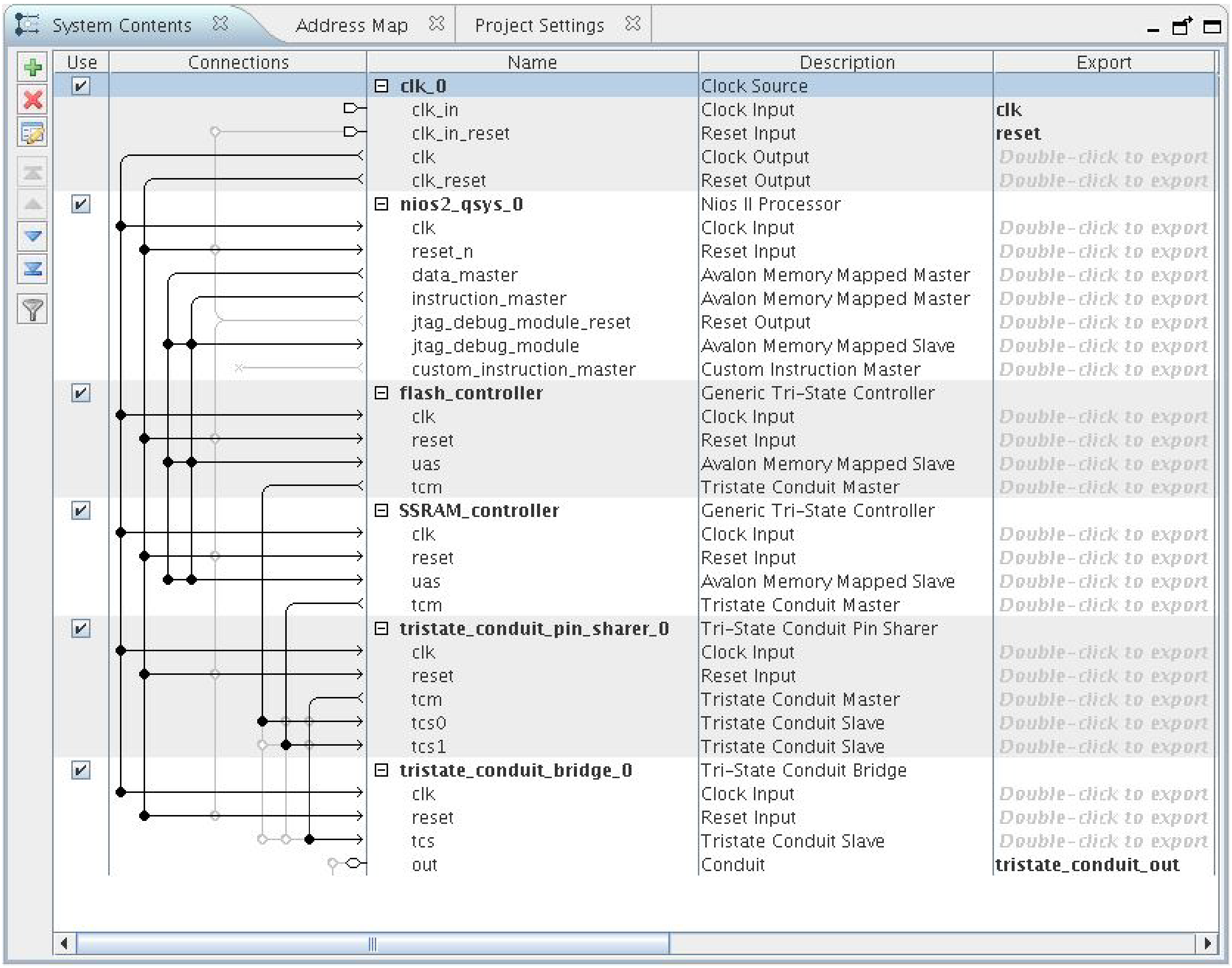
The flash device operates on 16-bit words and must ignore the least-significant bit of the Avalon-MM address, and shows `addr[0]`as not connected. The SSRAM memory operates on 32-bit words and must ignore the two, low-order memory bits. Because neither device requires a byte address, `addr[0]` is not routed on the PCB.

The flash device responds to address range 0 MBytes to 8 MBytes-1. The SSRAM responds to address range 8 MBytes to 10 MBytes-1. The PCB schematic for the PCB connects `addr [21:0]` to `addr [18:0]` of the SSRAM device because the SSRAM responds to 32-bit word address. The 8 MByte flash device accesses 16-bit words; consequently, the schematic does not connect `addr[0]`. The `chipselect` signals select between the two devices.



Note: If you create a custom tri-state conduit master with word aligned addresses, the Tri-state Conduit Pin Sharer does not change or align the address signals.

Figure 10-18: Tri-State Conduit System in Qsys



Use	Connections	Name	Description	Export
☑		⊟ **clk_0**	Clock Source	
		clk_in	Clock Input	**clk**
		clk_in_reset	Reset Input	**reset**
		clk	Clock Output	*Double-click to export*
		clk_reset	Reset Output	*Double-click to export*
☑		⊟ **nios2_qsys_0**	Nios II Processor	
		clk	Clock Input	*Double-click to export*
		reset_n	Reset Input	*Double-click to export*
		data_master	Avalon Memory Mapped Master	*Double-click to export*
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*
		jtag_debug_module_reset	Reset Output	*Double-click to export*
		jtag_debug_module	Avalon Memory Mapped Slave	*Double-click to export*
		custom_instruction_master	Custom Instruction Master	*Double-click to export*
☑		⊟ **flash_controller**	Generic Tri-State Controller	
		clk	Clock Input	*Double-click to export*
		reset	Reset Input	*Double-click to export*
		uas	Avalon Memory Mapped Slave	*Double-click to export*
		tcm	Tristate Conduit Master	*Double-click to export*
☑		⊟ **SSRAM_controller**	Generic Tri-State Controller	
		clk	Clock Input	*Double-click to export*
		reset	Reset Input	*Double-click to export*
		uas	Avalon Memory Mapped Slave	*Double-click to export*
		tcm	Tristate Conduit Master	*Double-click to export*
☑		⊟ **tristate_conduit_pin_sharer_0**	Tri-State Conduit Pin Sharer	
		clk	Clock Input	*Double-click to export*
		reset	Reset Input	*Double-click to export*
		tcm	Tristate Conduit Master	*Double-click to export*
		tcs0	Tristate Conduit Slave	*Double-click to export*
		tcs1	Tristate Conduit Slave	*Double-click to export*
☑		⊟ **tristate_conduit_bridge_0**	Tri-State Conduit Bridge	
		clk	Clock Input	*Double-click to export*
		reset	Reset Input	*Double-click to export*
		tcs	Tristate Conduit Slave	*Double-click to export*
		out	Conduit	**tristate_conduit_out**

Related Information

- Avalon Interface Specifications
- Avalon Tri-State Conduit Components User Guide

Generic Tri-State Controller

The Generic Tri-State Controller provides a template for a controller. You can customize the tri-state controller with various parameters to reflect the behavior of an off-chip device. The following types of parameters are available for the tri-state controller:

- Width of the address and data signals
- Read and write wait times
- Bus-turnaround time
- Data hold time

Note: In calculating delays, the Generic Tri-State Controller chooses the larger of the bus-turnaround time and read latency. Turnaround time is measured from the time that a command is accepted, not from the time that the previous read returned data.

The Generic Tri-State Controller includes the following interfaces:

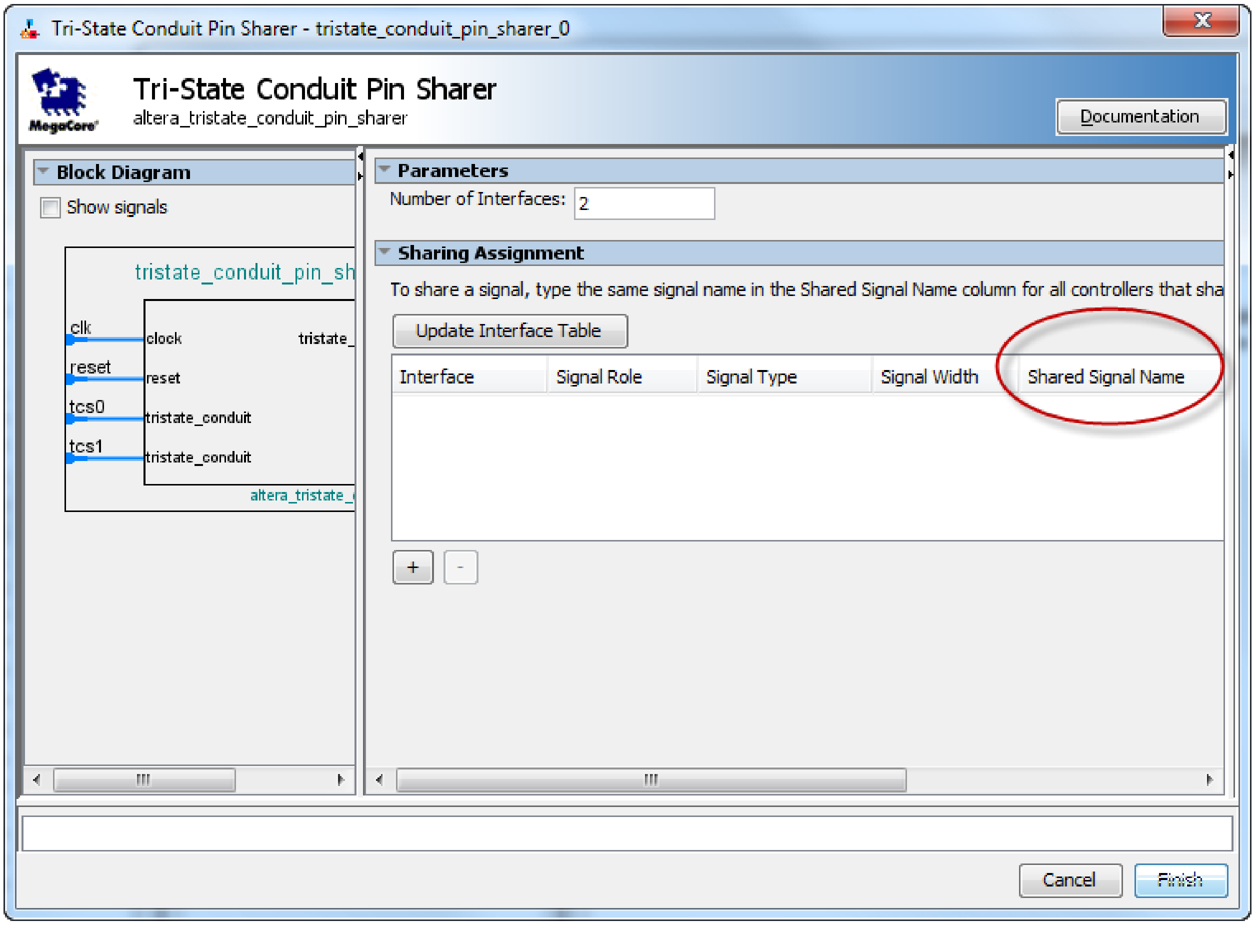
- **Memory-mapped slave interface**—This interface connects to an memory-mapped master, such as a processor.
- **Tristate Conduit Master interface**—Tri-state master interface usually connects to the tri-state conduit slave interface of the tri-state conduit pin sharer.
- **Clock sink**—The component's clock reference. You must connect this interface to a clock source.
- **Reset sink**—This interface connects to a reset source interface.

Tri-State Conduit Pin Sharer

The Tri-state Conduit Pin Sharer multiplexes between the signals of the connected tri-state controllers. You connect all signals from the tri-state controllers to the Tri-state Conduit Pin Sharer and use the parameter editor to specify the signals that are shared.

Figure 10-19: Tri-State Conduit Pin Sharer Parameter Editor

The parameter editor includes a **Shared Signal Name** column. If the widths of shared signals differ, the signals are aligned on their 0th bit and the higher-order pins are driven to 0 whenever the smaller signal has control of the bus. Unshared signals always propagate through the pin sharer. The tri-state conduit pin sharer uses the round-robin arbiter to select between tri-state conduit controllers.



Number of Interfaces: 2

To share a signal, type the same signal name in the Shared Signal Name column for all controllers that sha

Interface	Signal Role	Signal Type	Signal Width	Shared Signal Name

Note: All tri-state conduit components are connected to a pin sharer must be in the same clock domain.

Related Information

Avalon-ST Round Robin Scheduler on page 10-57

Tri-State Conduit Bridge

The Tri-State Conduit Bridge instantiates bidirectional signals for each tri-state signal while passing all other signals straight through the component. The Tri-State Conduit Bridge registers all outgoing and incoming signals, which adds two cycles of latency for a read request. You must account for this additional pipelining when designing a custom controller. During reset, all outputs are placed in a high-impedance state. Outputs are enabled in the first clock cycle after reset is deasserted, and the output signals are then bidirectional.

Test Pattern Generator and Checker Cores

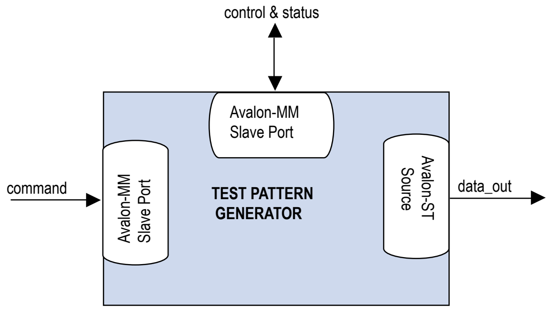
The data generation and monitoring solution for Avalon-ST consists of two components: a test pattern generator core that generates data, and sends it out on an Avalon-ST data interface, and a test pattern checker core that receives the same data and verifies it. Optionally, the data can be formatted as packets, with accompanying `start_of_packet` and `end_of_packet` signals.

The test pattern generator inserts different error conditions, and the test pattern checker reports these error conditions to the control interface, each via an Avalon Memory-Mapped (Avalon-MM) slave. The **Throttle Seed** is the starting value for the throttle control random number generator. Altera recommends a unique value for each instance of the test pattern generator and checker cores in a system.

Test Pattern Generator

Figure 10-20: Test Pattern Generator Core

The test pattern generator core accepts commands to generate data via an Avalon-MM command interface, and drives the generated data to an Avalon-ST data interface. You can parameterize most aspects of the Avalon-ST data interface, such as the number of error bits and data signal width, thus allowing you to test components with different interfaces.



The data pattern is calculated as: *Symbol Value* = *Symbol Position in Packet* `XOR` *Data Error Mask*. Data that is not organized in packets is a single stream with no beginning or end. The test pattern generator has a throttle register that is set via the Avalon-MM control interface. The test pattern generator uses the value of the throttle register in conjunction with a pseudo-random number generator to throttle the data generation rate.

Test Pattern Generator Command Interface

The command interface for the Test Pattern Generator is a 32-bit Avalon-MM write slave that accepts data generation commands. It is connected to a 16-element deep FIFO, thus allowing a master peripheral to drive a number of commands into the test pattern generator.

The command interface maps to the following registers: `cmd_lo` and `cmd_hi`. The command is pushed into the FIFO when the register `cmd_lo` (address 0) is addressed. When the FIFO is full, the command interface asserts the `waitrequest` signal. You can create errors by writing to the register `cmd_hi` (address 1). The errors are cleared when 0 is written to this register, or its respective fields.

Test Pattern Generator Control and Status Interface

The control and status interface of the Test Pattern Generator is a 32-bit Avalon-MM slave that allows you to enable or disable the data generation, as well as set the throttle. This interface also provides generation-time information, such as the number of channels and whether or not data packets are supported.

Test Pattern Generator Output Interface

The output interface of the Test Pattern Generator is an Avalon-ST interface that optionally supports data packets. You can configure the output interface to align with your system requirements. Depending on the incoming stream of commands, the output data may contain interleaved packet fragments for different channels. To keep track of the current symbol's position within each packet, the test pattern generator maintains an internal state for each channel.

You can configure the output interface of the test pattern generator with the following parameters:

- **Number of Channels**—Number of channels that the test pattern generator supports. Valid values are 1 to 256.
- **Data Bits Per Symbol**—Bits per symbol is related to the width of `readdata` and `writedata` signals, which must be a multiple of the bits per symbol.
- **Data Symbols Per Beat**—Number of symbols (words) that are transferred per beat. Valid values are 1 to 256.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Error Signal Width (bits)**—Width of the error signal on the output interface. Valid values are 0 to 31. A value of `0` indicates that the error signal is not in use.

Note: If you change only bits per symbol, and do not change the data width, errors are generated.

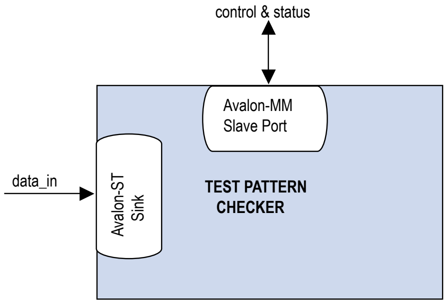
Test Pattern Generator Functional Parameter

The Test Pattern Generator functional parameter allows you to configure the test pattern generator as a whole system.

Test Pattern Checker

Figure 10-21: Test Pattern Checker

The test pattern checker core accepts data via an Avalon-ST interface and verifies it against the same predetermined pattern that the test pattern generator uses to produce the data. The test pattern checker core reports any exceptions to the control interface. You can parameterize most aspects of the test pattern checker's Avalon-ST interface such as the number of error bits and the data signal width. This enables the ability to test components with different interfaces. The test pattern checker has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register controls the rate at which data is accepted.



The test pattern checker detects exceptions and reports them to the control interface via a 32-element deep internal FIFO. Possible exceptions are data error, missing start-of-packet (SOP), missing end-of-packet (EOP), and signaled error.

As each exception occurs, an exception descriptor is pushed into the FIFO. If the same exception occurs more than once consecutively, only one exception descriptor is pushed into the FIFO. All exceptions are ignored when the FIFO is full. Exception descriptors are deleted from the FIFO after they are read by the control and status interface.

Test Pattern Checker Input Interface

The Test Pattern Checker input interface is an Avalon-ST interface that optionally supports data packets. You can configure the input interface to align with your system requirements. Incoming data may contain interleaved packet fragments. To keep track of the current symbol's position, the test pattern checker maintains an internal state for each channel.

Test Pattern Checker Control and Status Interface

The Test Pattern Checker control and status interface is a 32-bit Avalon-MM slave that allows you to enable or disable data acceptance, as well as set the throttle. This interface provides generation-time information, such as the number of channels and whether the test pattern checker supports data packets. The control and status interface also provides information on the exceptions detected by the test pattern checker. The interface obtains this information by reading from the exception FIFO.

Test Pattern Checker Functional Parameter

The Test Pattern Checker functional parameter allows you to configure the test pattern checker as a whole system.

Test Pattern Checker Input Parameters

You can configure the input interface of the test pattern checker using the following parameters:

- **Data Bits Per Symbol**—Bits per symbol is related to the width of `readdata` and `writedata` signals, which must be a multiple of the bits per symbol.
- **Data Symbols Per Beat**—Number of symbols (words) that are transferred per beat. Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not data packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Number of Channels**—Number of channels that the test pattern checker supports. Valid values are 1 to 256.
- **Error Signal Width (bits)**—Width of the `error` signal on the input interface. Valid values are 0 to 31. A value of `0` indicates that the `error` signal in not in use.

Note: If you change only bits per symbol, and do not change the data width, errors are generated.

Software Programming Model for the Test Pattern Generator and Checker Cores

The HAL system library support, software files, and register maps describe the software programming model for the test pattern generator and checker cores.

HAL System Library Support

For Nios II processor users, Altera provides HAL system library drivers that allow you to initialize and access the test pattern generator and checker cores. Altera recommends you to use the provided drivers to access the cores instead of accessing the registers directly.

For Nios II IDE users, copy the provided drivers from the following installation folders to your software application directory:

- ***<IP installation directory>*****/ip/sopc_builder_ip/altera_avalon_data_source/HAL**
- ***<IP installation directory>*****/ip/sopc_builder_ip/altera_avalon_data_sink/HAL**

Note: This instruction does not apply if you use the Nios II command-line tools.

Test Pattern Generator and Test Pattern Checker Core Files

The following files define the low-level access to the hardware, and provide the routines for the HAL device drivers.

Note: Do not modify the test pattern generator or test pattern checker core files.

- Test pattern generator core files:
  - **data_source_regs.h**—Header file that defines the test pattern generator's register maps.
  - **data_source_util.h , data_source_util.c**—Header and source code for the functions and variables required to integrate the driver into the HAL system library.
- Test pattern checker core files:
  - **data_sink_regs.h**—Header file that defines the core's register maps.
  - **data_sink_util.h , data_sink_util.c**—Header and source code for the functions and variables required to integrate the driver into the HAL system library.

Register Maps for the Test Pattern Generator and Test Pattern Checker Cores

Test Pattern Generator Control and Status Registers

Table 10-10: Test Pattern Generator Control and Status Register Map

Shows the offset for the test pattern generator control and status registers. Each register is 32-bits wide.

Offset	Register Name
base + 0	`status`
base + 1	`control`
base + 2	`fill`

Table 10-11: Test Pattern Generator Status Register Bits

Bit(s)	Name	Access	Description
[15:0]	`ID`	RO	A constant value of `0x64`.
[23:16]	`NUMCHANNELS`	RO	The configured number of channels.
[30:24]	`NUMSYMBOLS`	RO	The configured number of symbols per beat.
[31]	`SUPPORTPACKETS`	RO	A value of 1 indicates data packet support.

Table 10-12: Test Pattern Generator Control Register Bits

Bit(s)	Name	Access	Description
[0]	`ENABLE`	RW	Setting this bit to 1 enables the test pattern generator core.
[7:1]	Reserved		
[16:8]	`THROTTLE`	RW	Specifies the throttle value which can be between 0–256, inclusively. The test pattern generator uses this value in conjunction with a pseudo-random number generator to throttle the data generation rate. Setting `THROTTLE` to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0–256 result in a data rate proportional to the throttle value.
[17]	`SOFT RESET`	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
[31:18]	Reserved		

Table 10-13: Test Pattern Generator Fill Register Bits

Bit(s)	Name	Access	Description
[0]	`BUSY`	RO	A value of 1 indicates that data transmission is in progress, or that there is at least one command in the command queue.
[6:1]	Reserved		
[15:7]	`FILL`	RO	The number of commands currently in the command FIFO.
[31:16]	Reserved		

Test Pattern Generator Command Registers

Table 10-14: Test Pattern Generator Command Register Map

Shows the offset for the command registers. Each register is 32-bits wide.

Offset	Register Name
base + 0	`cmd_lo`
base + 1	`cmd_hi`

The `cmd_lo` is pushed into the FIFO only when the `cmd_lo` register is addressed.

Table 10-15: `cmd_lo` Register Bits

Bit(s)	Name	Access	Description
[15:0]	`SIZE`	RW	The segment size in symbols. Except for the last segment in a packet, the size of all segments must be a multiple of the configured number of symbols per beat. If this condition is not met, the test pattern generator core inserts additional symbols to the segment to ensure the condition is fulfilled.
[29:16]	`CHANNEL`	RW	The channel to send the segment on. If the `channel` signal is less than 14 bits wide, the test pattern generator uses the low order bits of this register to drive the signal.
[30]	`SOP`	RW	Set this bit to 1 when sending the first segment in a packet. This bit is ignored when data packets are not supported.
[31]	`EOP`	RW	Set this bit to 1 when sending the last segment in a packet. This bit is ignored when data packets are not supported.

Table 10-16: `cmd_hi` Register Bits

Bit(s)	Name	Access	Description
[15:0]	`SIGNALED ERROR`	RW	Specifies the value to drive the `error` signal. A non-zero value creates a signalled error.
[23:16]	`DATA ERROR`	RW	The output data is `XOR`ed with the contents of this register to create data errors. To stop creating data errors, set this register to 0.
[24]	`SUPPRESS SOP`	RW	Set this bit to 1 to suppress the assertion of the `startofpacket` signal when the first segment in a packet is sent.
[25]	`SUPRESS EOP`	RW	Set this bit to 1 to suppress the assertion of the `endofpacket` signal when the last segment in a packet is sent.

Test Pattern Checker Control and Status Registers

Table 10-17: Test Pattern Checker Control and Status Register Map

Shows the offset for the control and status registers. Each register is 32 bits wide.

Offset	Register Name
base + 0	`status`
base + 1	`control`
base + 2	Reserved
base + 3	
base + 4	
base + 5	`exception_descriptor`
base + 6	`indirect_select`
base + 7	`indirect_count`

Table 10-18: Test Pattern Checker Status Register Bits

Bit(s)	Name	Access	Description
[15:0]	`ID`	RO	Contains a constant value of `0x65`.
[23:16]	`NUMCHANNELS`	RO	The configured number of channels.
[30:24]	`NUMSYMBOLS`	RO	The configured number of symbols per beat.
[31]	`SUPPORTPACKETS`	RO	A value of 1 indicates packet support.

Table 10-19: Test Pattern Checker Control Register Bits

Bit(s)	Name	Access	Description
[0]	`ENABLE`	RW	Setting this bit to 1 enables the test pattern checker.
[7:1]	Reserved		
[16:8]	`THROTTLE`	RW	Specifies the throttle value which can be between 0–256, inclusively. Qsys uses this value in conjunction with a pseudo-random number generator to throttle the data generation rate. Setting `THROTTLE` to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0–256 result in a data rate proportional to the throttle value.
[17]	`SOFT RESET`	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
[31:18]	Reserved		

If there is no exception, reading the `exception_descriptor` register bit register returns 0.

Table 10-20: `exception_descriptor` Register Bits

Bit(s)	Name	Access	Description
[0]	`DATA ERROR`	RO	A value of 1 indicates that an error is detected in the incoming data.
[1]	`MISSINGSOP`	RO	A value of 1 indicates missing start-of-packet.
[2]	`MISSINGEOP`	RO	A value of 1 indicates missing end-of-packet.
[7:3]	Reserved		
[15:8]	`SIGNALLED ERROR`	RO	The value of the `error` signal.
[23:16]	Reserved		
[31:24]	`CHANNEL`	RO	The channel on which the exception was detected.

Table 10-21: `indirect_select` Register Bits

Bit	Bits Name	Access	Description
[7:0]	`INDIRECT CHANNEL`	RW	Specifies the channel number that applies to the `INDIRECT PACKET COUNT`, `INDIRECT SYMBOL COUNT`, and `INDIRECT ERROR COUNT` registers.
[15:8]	Reserved		

Bit	Bits Name	Access	Description
[31:16]	`INDIRECT ERROR`	RO	The number of data errors that occurred on the channel specified by `INDIRECT CHANNEL`.

Table 10-22: `indirect_count` Register Bits

Bit	Bits Name	Access	Description
[15:0]	`INDIRECT PACKET COUNT`	RO	The number of data packets received on the channel specified by `INDIRECT CHANNEL`.
[31:16]	`INDIRECT SYMBOL COUNT`	RO	The number of symbols received on the channel specified by `INDIRECT CHANNEL`.

.

Test Pattern Generator API

The following subsections describe application programming interface (API) for the test pattern generator.

Note: API functions are currently not available from the interrupt service routine (ISR).

data_source_reset()

Table 10-23: data_source_reset()

Information Type	Description
Prototype	`void data_source_reset(alt_u32 base);`
Thread-safe	No
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	`void`
Description	Resets the test pattern generator core including all internal counters and FIFOs. The control and status registers are not reset by this function.

data_source_init()

Table 10-24: data_source_init()

Information Type	Description
Prototype	`int data_source_init(alt_u32 base, alt_u32 command_base);`
Thread-safe	No
Include	<***data_source_util.h*** >
Parameters	`base`—Base address of the control and status slave. `command_base`—Base address of the command slave.
Returns	`1`—Initialization is successful. `0`—Initialization is unsuccessful.
Description	Performs the following operations to initialize the test pattern generator core: • Resets and disables the test pattern generator core. • Sets the maximum throttle. • Clears all inserted errors.

data_source_get_id()

Table 10-25: data_source_get_id()

Information Type	Description
Prototype	`int data_source_get_id(alt_u32 base);`

Information Type	Description
Thread-safe	Yes
Include	<*data_source_util.h* >
Parameters	`base`—Base address of the control and status slave.
Returns	Test pattern generator core identifier.
Description	Retrieves the test pattern generator core's identifier.

data_source_get_supports_packets()

Table 10-26: data_source_get_supports_packets()

Information Type	Description
Prototype	`int data_source_init(alt_u32 base);`
Thread-safe	Yes
Include	<*data_source_util.h* >
Parameters	`base`—Base address of the control and status slave.
Returns	`1`—Data packets are supported. `0`—Data packets are not supported.
Description	Checks if the test pattern generator core supports data packets.

data_source_get_num_channels()

Table 10-27: data_source_get_num_channels()

Description	Description
Prototype	`int data_source_get_num_channels(alt_u32 base);`
Thread-safe	Yes
Include	<*data_source_util.h* >
Parameters	`base`—Base address of the control and status slave.
Returns	Number of channels supported.
Description	Retrieves the number of channels supported by the test pattern generator core.

data_source_get_symbols_per_cycle()

Table 10-28: data_source_get_symbols_per_cycle()

Description	Description
Prototype	`int data_source_get_symbols(alt_u32 base);`
Thread-safe	Yes
Include	<*data_source_util.h* >
Parameters	`base`—Base address of the control and status slave.
Returns	Number of symbols transferred in a beat.
Description	Retrieves the number of symbols transferred by the test pattern generator core in each beat.

data_source_get_enable()

Table 10-29: data_source_get_enable()

Information Type	Description
Prototype	`int data_source_get_enable(alt_u32 base);`
Thread-safe	Yes
Include	<*data_source_util.h* >
Parameters	`base`—Base address of the control and status slave.
Returns	Value of the `ENABLE` bit.
Description	Retrieves the value of the `ENABLE` bit.

data_source_set_enable()

Table 10-30: data_source_set_enable()

Information Type	Description
Prototype	`void data_source_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe	No
Include	<*data_source_util.h* >
Parameters	`base`—Base address of the control and status slave. `value`— `ENABLE` bit set to the value of this parameter.

Information Type	Description
Returns	`void`
Description	Enables or disables the test pattern generator core. When disabled, the test pattern generator core stops data transmission but continues to accept commands and stores them in the FIFO

data_source_get_throttle()

Table 10-31: data_source_get_throttle()

Information Type	Description
Prototype	`int data_source_get_throttle(alt_u32 base);`
Thread-safe	Yes
Include	<*data_source_util.h* >
Parameters	`base`—Base address of the control and status slave.
Returns	Throttle value.
Description	Retrieves the current throttle value.

data_source_set_throttle()

Table 10-32: data_source_set_throttle()

Information Type	Description
Prototype	`void data_source_set_throttle(alt_u32 base, alt_u32 value)` `;`
Thread-safe	No
Include	<*data_source_util.h* >
Parameters	`base`—Base address of the control and status slave. `value`—Throttle value.
Returns	`void`
Description	Sets the throttle value, which can be between 0–256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern generator sends data.

data_source_is_busy()

Table 10-33: data_source_is_busy()

Information Type	Description
Prototype	`int data_source_is_busy(alt_u32 base);`
Thread-safe	Yes
Include	<*data_source_util.h* >
Parameters	`base`—Base address of the control and status slave.
Returns	`1`—Test pattern generator core is busy. `0`—Test pattern generator core is not busy.
Description	Checks if the test pattern generator is busy. The test pattern generator core is busy when it is sending data or has data in the command FIFO to be sent.

data_source_fill_level()

Table 10-34: data_source_fill_level()

Information Type	Description
Prototype	`int data_source_fill_level(alt_u32 base);`
Thread-safe	Yes
Include	<*data_source_util.h* >
Parameters	`base`—Base address of the control and status slave.
Returns	Number of commands in the command FIFO.
Description	Retrieves the number of commands currently in the command FIFO.

data_source_send_data()

Table 10-35: data_source_send_data()

Information Type	Description
Prototype	`int data_source_send_data(alt_u32 cmd_base, alt_u16 channel, alt_u16 size, alt_u32 flags, alt_u16 error, alt_u8 data_error_mask);`
Thread-safe	No
Include	<*data_source_util.h* >

Information Type	Description
Parameters	`cmd_base`—Base address of the command slave. `channel`—Channel to send the data. `size`—Data size. `flags` —Specifies whether to send or suppress SOP and EOP signals. Valid values are `DATA_SOURCE_SEND_SOP`, `DATA_SOURCE_SEND_EOP`, `DATA_SOURCE_SEND_SUPRESS_SOP` and `DATA_SOURCE_SEND_SUPRESS_EOP`. `error`—Value asserted on the `error` signal on the output interface. `data_error_mask`—Parameter and the data are `XOR`ed together to produce erroneous data.
Returns	Returns `1`.
Description	Sends a data fragment to the specified channel. If data packets are supported, applications must ensure consistent usage of SOP and EOP in each channel. Except for the last segment in a packet, the length of each segment is a multiple of the data width. If data packets are not supported, applications must ensure that there are no SOP and EOP indicators in the data. The length of each segment in a packet is a multiple of the data width.

Test Pattern Checker API

The following subsections describe API for the test pattern checker core. The API functions are currently not available from the ISR.

data_sink_exception_is_exception() on page 10-50

data_sink_exception_has_data_error() on page 10-51

data_sink_exception_has_missing_sop() on page 10-51

data_sink_exception_has_missing_eop() on page 10-51

data_sink_exception_signalled_error() on page 10-52

data_sink_exception_channel() on page 10-52

data_sink_reset()

Table 10-36: data_sink_reset()

Information Type	Description
Prototype	`void data_sink_reset(alt_u32 base);`
Thread-safe	No
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	`void`
Description	Resets the test pattern checker core including all internal counters.

data_sink_init()

Table 10-37: data_sink_init()

Information Type	Description
Prototype	`int data_source_init(alt_u32 base);`
Thread-safe	No
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	`1`—Initialization is successful. `0`—Initialization is unsuccessful.
Description	Performs the following operations to initialize the test pattern checker core: • Resets and disables the test pattern checker core. • Sets the throttle to the maximum value.

data_sink_get_id()

Table 10-38: data_sink_get_id()

Information Type	Description
Prototype	`int data_sink_get_id(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Test pattern checker core identifier.
Description	Retrieves the test pattern checker core’s identifier.

data_sink_get_supports_packets()

Table 10-39: data_sink_get_supports_packets()

Information Type	Description
Prototype	`int data_sink_init(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	`1`—Data packets are supported. `0`—Data packets are not supported.
Description	Checks if the test pattern checker core supports data packets.

data_sink_get_num_channels()

Table 10-40: data_sink_get_num_channels()

Information Type	Description
Prototype	`int data_sink_get_num_channels(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Number of channels supported.

Information Type	Description
Description	Retrieves the number of channels supported by the test pattern checker core.

data_sink_get_symbols_per_cycle()

Table 10-41: data_sink_get_symbols_per_cycle()

Information Type	Description
Prototype	`int data_sink_get_symbols(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Number of symbols received in a beat.
Description	Retrieves the number of symbols received by the test pattern checker core in each beat.

data_sink_get_enable()

Table 10-42: data_sink_get_enable()

Information Type	Description
Prototype	`int data_sink_get_enable(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	Value of the `ENABLE` bit.
Description	Retrieves the value of the `ENABLE` bit.

data_sink_set enable()

Table 10-43: data_sink_set enable()

Information Type	Description
Prototype	`void data_sink_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe	No

Information Type	Description
Include	<*data_sink_util.h* >
Parameters	`base`—Base address of the control and status slave. `value`—`ENABLE` bit is set to the value of the parameter.
Returns	`void`
Description	Enables the test pattern checker core.

data_sink_get_throttle()

Table 10-44: data_sink_get_throttle()

Information Type	Description
Prototype	`int data_sink_get_throttle(alt_u32 base);`
Thread-safe	Yes
Include	<*data_sink_util.h* >
Parameters	`base`—Base address of the control and status slave.
Returns	Throttle value.
Description	Retrieves the throttle value.

data_sink_set_throttle()

Table 10-45: data_sink_set_throttle()

Information Type	Description
Prototype	`void data_sink_set_throttle(alt_u32 base, alt_u32 value);`
Thread-safe	No
Include:	<*data_sink_util.h* >
Parameters	`base`—Base address of the control and status slave. `value`—Throttle value.
Returns	`void`
Description	Sets the throttle value, which can be between 0–256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern checker receives data.

data_sink_get_packet_count()

Table 10-46: data_sink_get_packet_count()

Information Type	Description
Prototype	`int data_sink_get_packet_count(alt_u32 base, alt_u32 channel) ;`
Thread-safe	No
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave. `channel`—Channel number.
Returns	Number of data packets received on the channel.
Description	Retrieves the number of data packets received on a channel.

data_sink_get_error_count()

Table 10-47: data_sink_get_error_count()

Information Type	Description
Prototype	`int data_sink_get_error_count(alt_u32 base, alt_u32 channel) ;`
Thread-safe	No
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave. `channel`—Channel number.
Returns	Number of errors received on the channel.
Description	Retrieves the number of errors received on a channel.

data_sink_get_symbol_count()

Table 10-48: data_sink_get_symbol_count()

Information Type	Description
Prototype	`int data_sink_get_symbol_count(alt_u32 base, alt_u32 channel) ;`
Thread-safe	No
Include	<***data_sink_util.h*** >

Information Type	Description
Parameters	`base`—Base address of the control and status slave. `channel`—Channel number.
Returns	Number of symbols received on the channel.
Description	Retrieves the number of symbols received on a channel.

data_sink_get_exception()

Table 10-49: data_sink_get_exception()

Information Type	Description
Prototype	`int data_sink_get_exception(alt_u32 base);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`base`—Base address of the control and status slave.
Returns	First exception descriptor in the exception FIFO. `0`—No exception descriptor found in the exception FIFO.
Description	Retrieves the first exception descriptor in the exception FIFO and pops it off the FIFO.

data_sink_exception_is_exception()

Table 10-50: data_sink_exception_is_exception()

Information Type	Description
Prototype	`int data_sink_exception_is_exception(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor
Returns	`1`—Indicates an exception. `0`—No exception.
Description	Checks if an exception descriptor describes a valid exception.

data_sink_exception_has_data_error()

Table 10-51: data_sink_exception_has_data_error()

Information Type	Description
Prototype	`int data_sink_exception_has_data_error(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor.
Returns	`1`—Data has errors. `0`—No errors.
Description	Checks if an exception indicates erroneous data.

data_sink_exception_has_missing_sop()

Table 10-52: data_sink_exception_has_missing_sop()

Information Type	Description
Prototype	`int data_sink_exception_has_missing_sop(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor.
Returns	`1`—Missing SOP. `0`—Other exception types.
Description	Checks if an exception descriptor indicates missing SOP.

data_sink_exception_has_missing_eop()

Table 10-53: data_sink_exception_has_missing_eop()

Information Type	Description
Prototype	`int data_sink_exception_has_missing_eop(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor.

Information Type	Description
Returns	`1`—Missing EOP. `0`—Other exception types.
Description	Checks if an exception descriptor indicates missing EOP.

data_sink_exception_signalled_error()

Table 10-54: data_sink_exception_signalled_error()

Information Type	Description
Prototype	`int data_sink_exception_signalled_error(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor.
Returns	Signal error value.
Description	Retrieves the value of the signaled error from the exception.

data_sink_exception_channel()

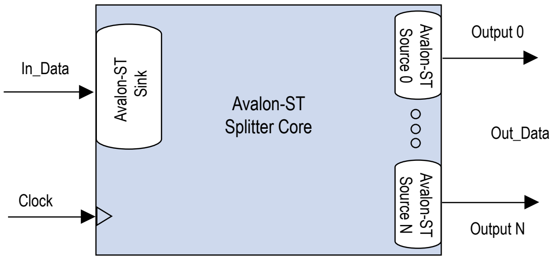
Table 10-55: data_sink_exception_channel()

Information Type	Description
Prototype	`int data_sink_exception_channel(int exception);`
Thread-safe	Yes
Include	<***data_sink_util.h*** >
Parameters	`exception`—Exception descriptor.
Returns	Channel number on which an exception occurred.
Description	Retrieves the channel number on which an exception occurred.

Avalon-ST Splitter Core

Figure 10-22: Avalon-ST Splitter Core

The Avalon-ST Splitter Core allows you to replicate transactions from an Avalon-ST source interface to multiple Avalon-ST sink interfaces. This core supports from 1 to 16 outputs.



The Avalon-ST Splitter core copies input signals from the input interface to the corresponding output signals of each output interface without altering the size or functionality. This includes all signals except for the `ready` signal. The core includes a clock signal to determine the Avalon-ST interface and clock domain where the core resides. Because the splitter core does nor use the `clock` signal internally, latency is not introduced when using this core.

Splitter Core Backpressure

The Avalon-ST Splitter core integrates with backpressure by AND-ing the `ready` signals from the output interfaces and sending the result to the input interface. As a result, if an output interface deasserts the `ready` signal, the input interface receives the deasserted `ready` signal, as well. This functionality ensures that backpressure on the output interfaces is propagated to the input interface.

When the **Qualify Valid Out** parameter is set to 1, the `out_valid` signals on all other output interfaces are gated when backpressure is applied from one output interface. In this case, when any output interface deasserts its `ready` signal, the `out_valid` signals on the other output interfaces are also deasserted.

When the **Qualify Valid Out** parameter is set to 0, the output interfaces have a non-gated `out_valid` signal when backpressure is applied. In this case, when an output interface deasserts its ready signal, the `out_valid` signals on the other output interfaces are not affected.

Because the logic is combinational, the core introduces no latency.

Splitter Core Interfaces

The Avalon-ST Splitter core supports streaming data, with optional packet, channel, and error signals. The core propagates backpressure from any output interface to the input interface.

Table 10-56: Avalon-ST Splitter Core Support

Feature	Support
Backpressure	Ready latency = 0.
Data Width	Configurable.

Feature	Support
Channel	Supported (optional).
Error	Supported (optional).
Packet	Supported (optional).

Splitter Core Parameters

Table 10-57: Avalon-ST Splitter Core Parameters

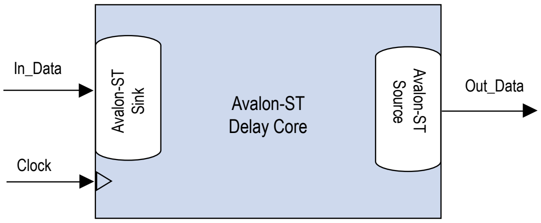
Parameter	Legal Values	Default Value	Description
Number Of Outputs	1 to 16	2	The number of output interfaces. Qsys supports 1 for some systems where no duplicated output is required.
Qualify Valid Out	0 or 1	1	Determines whether the `out_valid` signal is gated or non-gated when backpressure is applied.
Data Width	1–512	8	The width of the data on the Avalon-ST data interfaces.
Bits Per Symbol	1–512	8	The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.
Use Packets	0 or 1	0	Indicates whether or not data packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use Channel	0 or 1	0	The option to enable or disable the channel signal.
Channel Width	0-8	1	The width of the `channel` signal on the data interfaces. This parameter is disabled when **Use Channel** is set to 0.
Max Channels	0-255	1	The maximum number of channels that a data interface can support. This parameter is disabled when **Use Channel** is set to 0.
Use Error	0 or 1	0	The option to enable or disable the error signal.

Parameter	Legal Values	Default Value	Description
Error Width	0–31	1	The width of the `error` signal on the output interfaces. A value of `0` indicates that the splitter core is not using the `error` signal. This parameter is disabled when **Use Error** is set to `0`.

Avalon-ST Delay Core

Figure 10-23: Avalon-ST Delay Core

The Avalon-ST Delay Core provides a solution to delay Avalon-ST transactions by a constant number of clock cycles. This core supports up to 16 clock cycle delays.



The Delay core adds a delay between the input and output interfaces. The core accepts transactions presented on the input interface and reproduces them on the output interface `N` cycles later without changing the transaction.

The input interface delays the input signals by a constant `N` number of clock cycles to the corresponding output signals of the output interface. The **Number Of Delay Clocks** parameter defines the constant `N`, which must be between 0 and 16. The change of the `in_valid` signal is reflected on the `out_valid` signal exactly `N` cycles later.

Delay Core Reset Signal

The Avalon-ST Delay core has a `reset` signal that is synchronous to the `clk` signal. When the core asserts the `reset` signal, the output signals are held at 0. After the `reset` signal is deasserted, the output signals are held at 0 for `N` clock cycles. The delayed values of the input signals are then reflected at the output signals after `N` clock cycles.

Delay Core Interfaces

The Delay core supports streaming data, with optional packet, channel, and error signals. The delay core does not support backpressure.

Table 10-58: Avalon-ST Delay Core Support

Feature	Support
Backpressure	Not supported.
Data Width	Configurable.

Feature	Support
Channel	Supported (optional).
Error	Supported (optional).
Packet	Supported (optional).

Delay Core Parameters

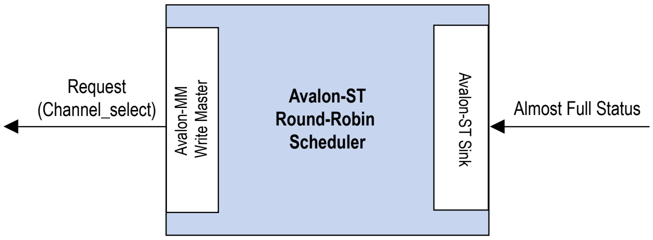
Table 10-59: Avalon-ST Delay Core Parameters

Parameter	Legal Values	Default Value	Description
Number Of Delay Clocks	0 to 16	1	Specifies the delay the core introduces, in clock cycles. Qsys supports 0 for some systems where no delay is required.
Data Width	1–512	8	The width of the data on the Avalon-ST data interfaces.
Bits Per Symbol	1–512	8	The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.
Use Packets	0 or 1	0	Indicates whether or not data packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use Channel	0 or 1	0	The option to enable or disable the channel signal.
Channel Width	0-8	1	The width of the `channel` signal on the data interfaces. This parameter is disabled when **Use Channel** is set to 0.
Max Channels	0-255	1	The maximum number of channels that a data interface can support. This parameter is disabled when **Use Channel** is set to 0.
Use Error	0 or 1	0	The option to enable or disable the error signal.
Error Width	0–31	1	The width of the `error` signal on the output interfaces. A value of 0 indicates that the error signal is not in use. This parameter is disabled when **Use Error** is set to 0.

Avalon-ST Round Robin Scheduler

Figure 10-24: Avalon-ST Round Robin Scheduler

The Avalon-ST Round Robin Scheduler core controls the read operations from a multi-channel Avalon-ST component that buffers data by channels. It reads the almost-full threshold values from the multiple channels in the multi-channel component and issues the read request to the Avalon-ST source according to a round-robin scheduling algorithm.



In a multi-channel component, the component can store data either in the sequence that it comes in (FIFO), or in segments according to the channel. When data is stored in segments according to channels, a scheduler is needed to schedule the read operations.

Almost-Full Status Interface (Round Robin Scheduler)

The Almost-Full Status interface is an Avalon-ST sink interface that collects the almost-full status from the sink components for the channels in the sequence provided.

Table 10-60: Avalon-ST Interface Feature Support

Feature	Property
Backpressure	Not supported
Data Width	Data width = 1; Bits per symbol = 1
Channel	Maximum channel = 32; Channel width = 5
Error	Not supported
Packet	Not supported

Request Interface (Round Robin Scheduler)

The Request Interface is an Avalon-MM write master interface that requests data from a specific channel. The Avalon-ST Round Robin Scheduler cycles through the channels it supports and schedules data to be read.

Round Robin Scheduler Operation

If a particular channel is almost full, the Avalon-ST Round Robin Scheduler does not schedule data to be read from that channel in the source component.

The scheduler only requests 1 beat of data from a channel at each transaction. To request 1 beat of data from channel *n*, the scheduler writes the value `1` to address ($4 \times n$). For example, if the scheduler is requesting data from channel 3, the scheduler writes `1` to address `0xC`. At every clock cycle, the scheduler requests data from the next channel. Therefore, if the scheduler starts requesting from channel 1, at the next clock cycle, it requests from channel 2. The scheduler does not request data from a particular channel if the almost-full status for the channel is asserted. In this case, the scheduler uses one clock cycle without a request transaction.

The Avalon-ST Round Robin Scheduler cannot determine if the requested component is able to service the request transaction. The component asserts `waitrequest` when it cannot accept new requests.

Table 10-61: Avalon-ST Round Robin Scheduler Ports

Signal	Direction	Description
Clock and Reset		
`clk`	In	Clock reference.
`reset_n`	In	Asynchronous active low reset.
Avalon-MM Request Interface		
`request_address (log`$_2$ `Max_Channels-1:0)`	Out	The write address that indicates which channel has the request.
`request_write`	Out	Write enable signal.
`request_writedata`	Out	The amount of data requested from the particular channel. This value is always fixed at 1.
`request_waitrequest`	In	Wait request signal that pauses the scheduler when the slave cannot accept a new request.
Avalon-ST Almost-Full Status Interface		
`almost_full_valid`	In	Indicates that `almost_full_channel` and `almost_full_data` are valid.
`almost_full_channel (Channel_Width-1:0)`	In	Indicates the channel for the current status indication.
`almost_full_data (log`$_2$ `Max_Channels-1:0)`	In	A 1-bit signal that is asserted high to indicate that the channel indicated by `almost_full_channel` is almost full.

Round Robin Scheduler Parameters

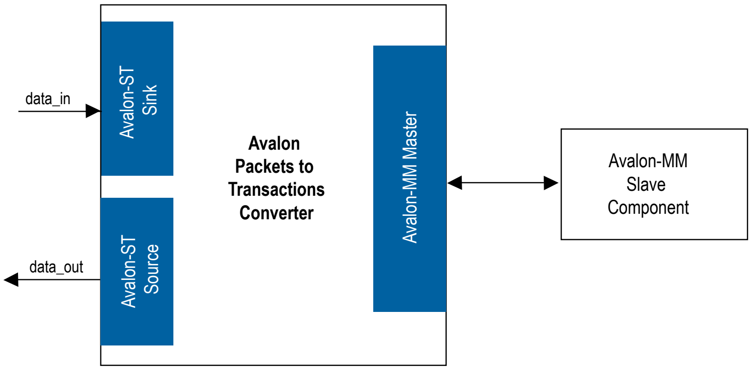
Table 10-62: Avalon-ST Round Robin Scheduler Parameters

Parameters	Values	Description
Number of channels	2–32	Specifies the number of channels the Avalon-ST Round Robin Scheduler supports.
Use almost-full status	0–1	Specifies whether the scheduler uses the almost-full interface. If not, the core requests data from the next channel at the next clock cycle.

Avalon Packets to Transactions Converter

Figure 10-25: Avalon Packets to Transactions Converter Core

The Avalon Packets to Transactions Converter core receives streaming data from upstream components and initiates Avalon-MM transactions. The core then returns Avalon-MM transaction responses to the requesting components.



Note: The SPI Slave to Avalon Master Bridge and JTAG to Avalon Master Bridge are examples of the Packets to Transactions Converter core. For more information, refer to the *Avalon Interface Specifications*.

Related Information

Avalon Interface Specifications

Packets to Transactions Converter Interfaces

Table 10-63: Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Data width = 8 bits; Bits per symbol = 8.

Feature	Property
Channel	Not supported.
Error	Not used.
Packet	Supported.

The Avalon-MM master interface supports read and write transactions. The data width is set to 32 bits, and burst transactions are not supported.

Packets to Transactions Converter Operation

The Packets to Transactions Converter core receives streams of packets on its Avalon-ST sink interface and initiates Avalon-MM transactions. Upon receiving transaction responses from Avalon-MM slaves, the core transforms the responses to packets and returns them to the requesting components via its Avalon-ST source interface. The core does not report Avalon-ST errors.

Packets to Transactions Converter Data Packet Formats

A response packet is returned for every write transaction. The core also returns a response packet if a no transaction (`0x7f`) is received. An invalid transaction code is regarded as a no transaction. For read transactions, the core returns the data read.

The Packets to Transactions Converter core expects incoming data streams to be in the formats shown the table below.

Table 10-64: Data Packet Formats

Byte	Field	Description
Transaction Packet Format		
0	`Transaction code`	Type of transaction.
1	Reserved	Reserved for future use.
[3:2]	`Size`	Transaction size in bytes. For write transactions, the size indicates the size of the `data` field. For read transactions, the size indicates the total number of bytes to read.
[7:4]	`Address`	32-bit address for the transaction.
[n:8]	`Data`	Transaction data; data to be written for write transactions.
Response Packet Format		
0	`Transaction code`	The transaction code with the most significant bit inversed.
1	Reserved	Reserved for future use.
[4:2]	`Size`	Total number of bytes read/written successfully.

Related Information

Packets to Transactions Converter Interfaces on page 10-59

Packets to Transactions Converter Supported Transactions

Table 10-65: Packets to Transactions Converter Supported Transactions

Avalon-MM transactions supported by the Packets to Transactions Converter core.

Transaction Code	Avalon-MM Transaction	Description
`0x00`	Write, non-incrementing address.	Writes data to the address until the total number of bytes written to the same word address equals to the value specified in the `size` field.
`0x04`	Write, incrementing address.	Writes transaction data starting at the current address.
`0x10`	Read, non-incrementing address.	Reads 32 bits of data from the address until the total number of bytes read from the same address equals to the value specified in the `size` field.
`0x14`	Read, incrementing address.	Reads the number of bytes specified in the `size` parameter starting from the current address.
`0x7f`	No transaction.	No transaction is initiated. You can use this transaction type for testing purposes. Although no transaction is initiated on the Avalon-MM interface, the core still returns a response packet for this transaction code.

The Packets to Transactions Converter core can process only a single transaction at a time. The `ready` signal on the core's Avalon-ST sink interface is asserted only when the current transaction is completely processed.

No internal buffer is implemented on the data paths. Data received on the Avalon-ST interface is forwarded directly to the Avalon-MM interface and vice-versa. Asserting the `waitrequest` signal on the Avalon-MM interface backpressures the Avalon-ST sink interface. In the opposite direction, if the Avalon-ST source interface is backpressured, the `read` signal on the Avalon-MM interface is not asserted until the backpressure is alleviated. Backpressuring the Avalon-ST source in the middle of a read could result in data loss. In this cases, the core returns the data that is successfully received.

A transaction is considered complete when the core receives an EOP. For write transactions, the actual data size is expected to be the same as the value of the `size` property. Whether or not both values agree, the core always uses the end of packet (EOP) to determine the end of data.

Packets to Transactions Converter Malformed Packets

The following are examples of malformed packets:

- **Consecutive start of packet (SOP)**—An SOP marks the beginning of a transaction. If an SOP is received in the middle of a transaction, the core drops the current transaction without returning a response packet for the transaction, and initiates a new transaction. This effectively precesses packets without an end of packet (EOP).
- **Unsupported transaction codes**—The core processes unsupported transactions as a no transaction.

Avalon-ST Streaming Pipeline Stage

The Avalon-ST pipeline stage receives data from an Avalon-ST source interface, and outputs the data to an Avalon-ST sink interface. In the absence of back pressure, the Avalon-ST pipeline stage source interface outputs data one cycle after receiving the data on its sink interface.

If the pipeline stage receives back pressure on its source interface, it continues to assert its source interface's current data output. While the pipeline stage is receiving back pressure on its source interface and it receives new data on its sink interface, the pipeline stage internally buffers the new data. It then asserts back pressure on its sink interface.

Once the backpressure is deasserted, the pipeline stage's source interface is de-asserted and the pipeline stage asserts internally buffered data (if present). Additionally, the pipeline stage de-asserts back pressure on its sink interface.

Figure 10-26: Pipeline Stage Simple Register

If the ready signal is not pipelined, the pipeline stage becomes a simple register.

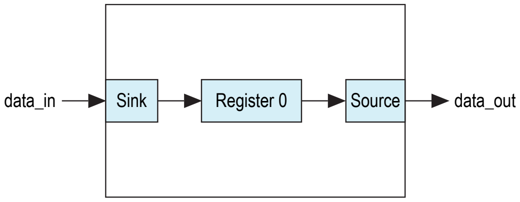
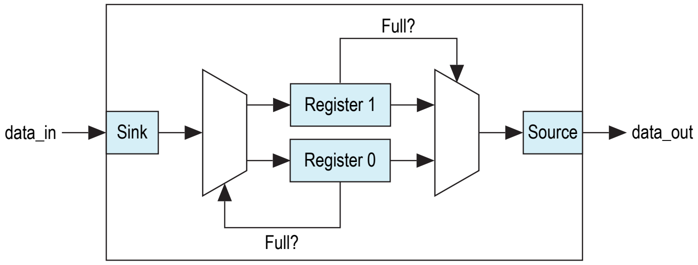


Figure 10-27: Pipeline Stage Holding Register

If the ready signal is pipelined, the pipeline stage must also include a second "holding" register.



Streaming Channel Multiplexer and Demultiplexer Cores

The Avalon-ST channel multiplexer core receives data from various input interfaces and multiplexes the data into a single output interface, using the optional `channel` signal to indicate the origin of the data. The Avalon-ST channel demultiplexer core receives data from a channelized input interface and drives that data to multiple output interfaces, where the output interface is selected by the input `channel` signal.

The multiplexer and demultiplexer cores can transfer data between interfaces on cores that support unidirectional flow of data. The multiplexer and demultiplexer allow you to create multiplexed or demultiplexed data paths without having to write custom HDL code. The multiplexer includes an Avalon-ST Round Robin Scheduler.

Related Information

Avalon-ST Round Robin Scheduler on page 10-57

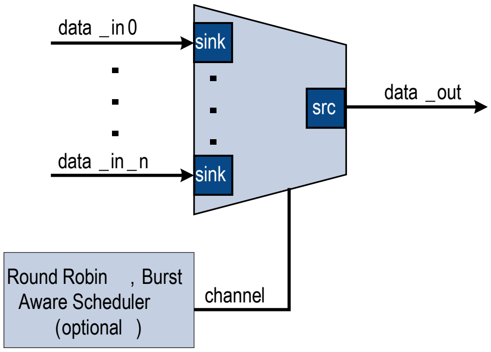
Software Programming Model For the Multiplexer and Demultiplexer Components

The multiplexer and demultiplexer components do not have any user-visible control or status registers. Therefore, Qsys cannot control or configure any aspect of the multiplexer or demultiplexer at run-time. The components cannot generate interrupts.

Avalon-ST Multiplexer

Figure 10-28: Avalon-ST Multiplexer

The Avalon-ST multiplexer takes data from a variety of input data interfaces, and multiplexes the data onto a single output interface. The multiplexer includes a round-robin scheduler that selects from the next input interface that has data. Each input interface has the same width as the output interface, so that the other input interfaces are backpressured when the multiplexer is carrying data from a different input interface.



The multiplexer includes an optional channel signal that enables each input interface to carry channelized data. The output interface channel width is equal to:

(log2 (n-1)) + 1 + w

where `n` is the number of input interfaces, and `w` is the channel width of each input interface. All input interfaces must have the same channel width. These bits are appended to either the most or least significant bits of the output channel signal.

The scheduler processes one input interface at a time, selecting it for transfer. Once an input interface has been selected, data from that input interface is sent until one of the following scenarios occurs:

- The specified number of cycles have elapsed.
- The input interface has no more data to send and the `valid` signal is deasserted on a ready cycle.
- When packets are supported, `endofpacket` is asserted.

Multiplexer Input Interfaces

Each input interface is an Avalon-ST data interface that optionally supports packets. The input interfaces are identical; they have the same symbol and data widths, error widths, and channel widths.

Multiplexer Output Interface

The output interface carries the multiplexed data stream with data from the inputs. The symbol, data, and error widths are the same as the input interfaces.

The width of the `channel` signal is the same as the input interfaces, with the addition of the bits needed to indicate the origin of the data.

You can configure the following parameters for the output interface:

- **Data Bits Per Symbol**—The bits per symbol is related to the width of `readdata` and `writedata` signals, which must be a multiple of the bits per symbol.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**— The number of bits Qsys uses for the channel signal for output interfaces. For example, set this parameter to 1 if you have two input interfaces with no channel, or set this parameter to 2 if you have two input interfaces with a channel width of 1 bit. The input channel can have a width between 0-31 bits.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of `0` means the `error` signal is not in use.

Note: If you change only bits per symbol, and do not change the data width, errors are generated.

Multiplexer Parameters

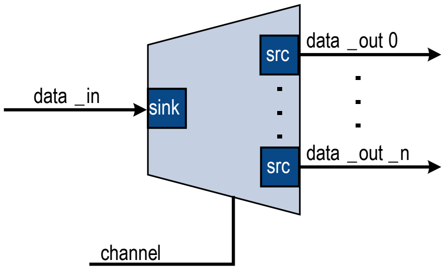
You can configure the following parameters for the multiplexer:

- **Number of Input Ports**—The number of input interfaces that the multiplexer supports. Valid values are 2 to 16.
- **Scheduling Size (Cycles)**—The number of cycles that are sent from a single channel before changing to the next channel.
- **Use Packet Scheduling**—When this parameter is turned on, the multiplexer only switches the selected input interface on packet boundaries. Therefore, packets on the output interface are not interleaved.
- **Use high bits to indicate source port**—When this parameter is turned on, the multiplexer uses the high bits of the output `channel` signal to indicate the origin of the input interface of the data. For example, if the input interfaces have 4-bit channel signals, and the multiplexer has 4 input interfaces, the output interface has a 6-bit channel signal. If this parameter is turned on, bits [5:4] of the output channel signal indicate origin of the input interface of the data, and bits [3:0] are the channel bits that were presented at the input interface.

Avalon-ST Demultiplexer

Figure 10-29: Avalon-ST Demultiplexer

That Avalon-ST demultiplexer takes data from a channelized input data interface and provides that data to multiple output interfaces, where the output interface selected for a particular transfer is specified by the input `channel` signal.



The data is delivered to the output interfaces in the same order it is received at the input interface, regardless of the value of `channel`, `packet`, `frame`, or any other signal. Each of the output interfaces has the same width as the input interface; each output interface is idle when the demultiplexer is driving data to a different output interface. The demultiplexer uses $\log_2$ (`num_output_interfaces`) bits of the `channel` signal to select the output for the data; the remainder of the channel bits are forwarded to the appropriate output interface unchanged.

Demultiplexer Input Interface

Each input interface is an Avalon-ST data interface that optionally supports packets. You can configure the following parameters for the input interface:

- **Data Bits Per Symbol**—The bits per symbol is related to the width of `readdata` and `writedata` signals, which must be a multiple of the bits per symbol.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not data packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**—The number of bits for the `channel` signal for output interfaces. A value of `0` means that output interfaces do not use the optional `channel` signal.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of `0` means the `error` signal is in use.

Note: If you change only bits per symbol, and do not change the data width, errors are generated.

Demultiplexer Output Interface

Each output interface carries data from a subset of channels from the input interface. Each output interface is identical; all have the same symbol and data widths, error widths, and channel widths. The symbol, data, and error widths are the same as the input interface. The width of the `channel` signal is the same as the input interface, without the bits that the demultiplexer uses to select the output interface.

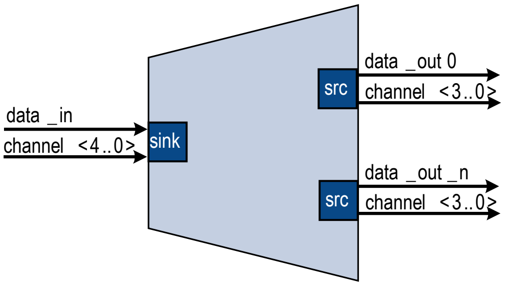
Demultiplexer Parameters

You can configure the following parameters for the demultiplexer:

- **Number of Output Ports**—The number of output interfaces that the multiplexer supports Valid values are 2 to 16.
- **High channel bits select output**—When this option is turned on, the demultiplexing function uses the high bits of the input `channel` signal, and the low order bits are passed to the output. When this option is turned off, the demultiplexing function uses the low order bits, and the high order bits are passed to the output.

Where you place the signals in our design affects the functionality; for example, there is one input interface and two output interfaces. If the low-order bits of the channel signal select the output interfaces, the even channels goes to channel 0, and the odd channels goes to channel 1. If the high-order bits of the channel signal select the output interface, channels 0 to 7 goes to channel 0 and channels 8 to 15 goes to channel 1.

Figure 10-30: Select Bits for the Demultiplexer



Single-Clock and Dual-Clock FIFO Cores

The Avalon-ST Single-Clock and Avalon-ST Dual-Clock FIFO cores are FIFO buffers which operate with a common clock and independent clocks for input and output ports respectively.

Figure 10-31: Avalon-ST Single Clock FIFO Core

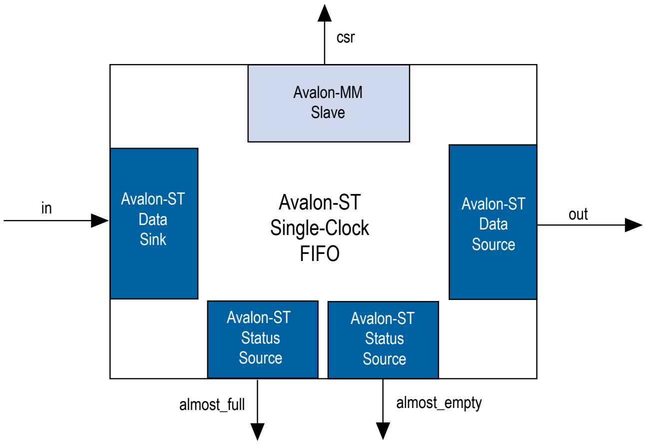
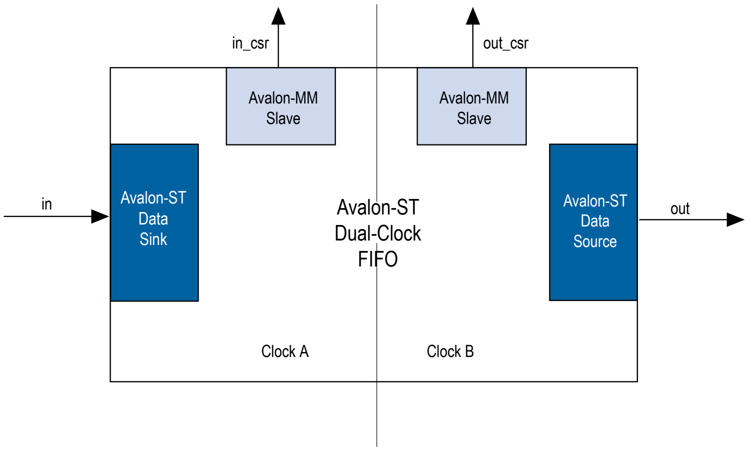


Figure 10-32: Avalon-ST Dual Clock FIFO Core



Interfaces Implemented in FIFO Cores

The following interfaces are implemented in FIFO cores:

Avalon-ST Data Interface on page 10-67

Avalon-MM Control and Status Register Interface on page 10-67

Avalon-ST Status Interface on page 10-68

Avalon-ST Data Interface

Each FIFO core has an Avalon-ST data sink and source interfaces. The data sink and source interfaces in the dual-clock FIFO core are driven by different clocks.

Table 10-66: Avalon-ST Interfaces Properties

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Configurable.
Channel	Supported, up to 255 channels.
Error	Configurable.
Packet	Configurable.

Avalon-MM Control and Status Register Interface

You can configure the single-clock FIFO core to include an optional Avalon-MM interface, and the dual-clock FIFO core to include an Avalon-MM interface in each clock domain. The Avalon-MM interface provides access to 32-bit registers, which allows you to retrieve the FIFO buffer fill level and configure the almost-empty and almost-full thresholds. In the single-clock FIFO core, you can also configure the packet and error handling modes.

Avalon-ST Status Interface

The single-clock FIFO core has two optional Avalon-ST status source interfaces from which you can obtain the FIFO buffer almost-full and almost empty statuses.

FIFO Operating Modes

- **Default mode**—The core accepts incoming data on the `in` interface (Avalon-ST data sink) and forwards it to the `out` interface (Avalon-ST data source). The core asserts the `valid` signal on the Avalon-ST source interface to indicate that data is available at the interface.
- **Store and forward mode**—This mode applies only to the single-clock FIFO core. The core asserts the `valid` signal on the `out` interface only when a full packet of data is available at the interface. In this mode, you can also enable the drop-on-error feature by setting the `drop_on_error` register to 1. When this feature is enabled, the core drops all packets received with the `in_error` signal asserted.
- **Cut-through mode**—This mode applies only to the single-clock FIFO core. The core asserts the `valid` signal on the `out` interface to indicate that data is available for consumption when the number of entries specified in the `cut_through_threshold` register are available in the FIFO buffer.

To use the store and forward or cut-through mode, turn on the **Use store and forward** parameter to include the `csr` interface (Avalon-MM slave). Set the `cut_through_threshold` register to 0 to enable the store and forward mode, and then set the register to any value greater than 0 to enable the cut-through mode. The non-zero value specifies the minimum number of FIFO entries that must be available before the data is ready for consumption. Setting the register to 1 provides you with the default mode.

Fill Level of the FIFO Buffer

You can obtain the fill level of the FIFO buffer via the optional Avalon-MM control and status interface. Turn on the **Use fill level** parameter (**Use sink fill level** and **Use source fill level** in the dual-clock FIFO core) and read the `fill_level` register.

The dual-clock FIFO core has two fill levels. one in each clock domain. Due to the latency of the clock crossing logic, the fill levels reported in the input and output clock domains may be different for any instance. In both cases, the fill level may report badly for the clock domain; that is, the fill level is reported high in the input clock domain, and low in the output clock domain.

The dual-clock FIFO has an output pipeline stage to improve f_{MAX}. This output stage is accounted for when calculating the output fill level, but not when calculating the input fill level. Therefore, the best measure of the amount of data in the FIFO is by the fill level in the output clock domain. The fill level in the input clock domain represents the amount of space available in the FIFO (available space = FIFO depth – input fill level).

Almost-Full and Almost-Empty Thresholds to Prevent Overflow and Underflow

You can use almost-full and almost-empty thresholds as a mechanism to prevent FIFO overflow and underflow. This feature is available only in the single-clock FIFO core. To use the thresholds, turn on the **Use fill level**, **Use almost-full status**, and **Use almost-empty status** parameters. You can access the `almost_full_threshold` and `almost_full_threshold` registers via the csr interface and set the registers to an optimal value for your application.

You can obtain the almost-full and almost-empty statuses from `almost_full` and `almost_empty` interfaces (Avalon-ST status source). The core asserts the `almost_full` signal when the fill level is equal to or higher than the almost-full threshold. Likewise, the core asserts the `almost_empty` signal when the fill level is equal to or lower than the almost-empty threshold.

Single-Clock and Dual-Clock FIFO Core Parameters

Table 10-67: Single-Clock and Dual-Clock FIFO Core Parameters

Parameter	Legal Values	Description
Bits per symbol	1–32	These parameters determine the width of the FIFO. FIFO width = **Bits per symbol** * **Symbols per beat**, where: **Bits per symbol** is the number of bits in a symbol, and **Symbols per beat** is the number of symbols transferred in a beat.
Symbols per beat	1–32	
Error width	0–32	The width of the `error` signal.
FIFO depth	2^n	The FIFO depth. An output pipeline stage is added to the FIFO to increase performance, which increases the FIFO depth by one. <n> = n=1,2,3,4...
Use packets	—	Turn on this parameter to enable data packet support on the Avalon-ST data interfaces.
Channel width	1–32	The width of the `channel` signal.
Avalon-ST Single Clock FIFO Only		
Use fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface.
Avalon-ST Dual Clock FIFO Only		
Use sink fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface in the input clock domain.
Use source fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface in the output clock domain.
Write pointer synchronizer length	2–8	The length of the write pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability while increasing the latency of the core.
Read pointer synchronizer length	2–8	The length of the read pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability.
Use Max Channel	—	Turn on this parameter to specify the maximum channel number.
Max Channel	1–255	Maximum channel number.

Note: For more information on metastability in Altera devices, refer to *Understanding Metastability in FPGAs*. For more information on metastability analysis and synchronization register chains, refer to the *Managing Metastability*.

Related Information

- Understanding Metastability in FPGAs
- Managing Metastability

Avalon-ST Single-Clock FIFO Registers

Table 10-68: Avalon-ST Single-Clock FIFO Registers

The `csr` interface in the Avalon-ST Single Clock FIFO core provides access to registers.

32-Bit Word Offset	Name	Access	Reset	Description
0	`fill_level`	R	0	24-bit FIFO fill level. Bits 24 to 31 are not used.
1	Reserved	—	—	Reserved for future use.
2	`almost_full_threshold`	RW	**FIFO depth**–1	Set this register to a value that indicates the FIFO buffer is getting full.
3	`almost_empty_threshold`	RW	0	Set this register to a value that indicates the FIFO buffer is getting empty.
4	`cut_through_threshold`	RW	0	**0**—Enables store and forward mode. **Greater than 0**—Enables cut-through mode and specifies the minimum of entries in the FIFO buffer before the `valid` signal on the Avalon-ST source interface is asserted. Once the FIFO core starts sending the data to the downstream component, it continues to do so until the end of the packet. This register applies only when the **Use store and forward** parameter is turned on.
5	`drop_on_error`	RW	0	**0**—Disables drop-on error. **1**—Enables drop-on error. This register applies only when the **Use packet** and **Use store and forward** parameters are turned on.

Table 10-69: Register Description for Avalon-ST Dual-Clock FIFO

The `in_csr` and `out_csr` interfaces in the Avalon-ST Dual Clock FIFO core reports the FIFO fill level.

32-Bit Word Offset	Name	Access	Reset Value	Description
0	`fill_level`	R	0	24-bit FIFO fill level. Bits 24 to 31 are not used.

Related Information

- Avalon Interface Specifications
- Avalon Memory-Mapped Design Optimizations

Document Revision History

Table 10-70: Document Revision History

The table below indicates edits made to the *Qsys System Design Components* content since its creation.

Date	Version	Changes
June 2014	14.0.0	• AXI Bridge support. • Address Span Extender updates. • Avalon-MM Unaligned Burst Expansion Bridge support.
November 2013	13.1.0	• Address Span Extender
May 2013	13.0.0	• Added Streaming Pipeline Stage support. • Added AMBA APB support.
November 2012	12.1.0	• Moved relevant content from the *Embedded Peripherals IP User Guide.*

Related Information

Quartus II Handbook Archive