Qsys Interconnect 8

2013.11.4

QII51021 Subscribe Send Feedback

Qsys interconnect is a high-bandwidth structure for connecting components, and that allows you to connect IP cores to other IP cores with various interfaces. Qsys supports standard Avalon®, AMBA® AXI3™ (version 1.0), AMBA AXI4™ (version 2.0), and AMBA APB™ 3 (version 1.0) interfaces.

Related Information

- Avalon Interface Specifications
- AMBA Protocol Specifications
- Creating a System with Qsys
- Creating Qsys Components
- Qsys System Design Components

Memory-Mapped Interfaces

This topic describes the implementation and structure of the Qsys interconnect for memory-mapped interfaces. The content pertains to both Avalon and AXI memory-mapped interfaces, unless noted otherwise.

Memory-mapped transactions between masters and slaves are encapsulated in packets and transmitted on a network that carries the packets between masters and slaves. The command network transports read and write command packets from master interfaces to slave interfaces. The response network transports response packets from slave interfaces to master interfaces.

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

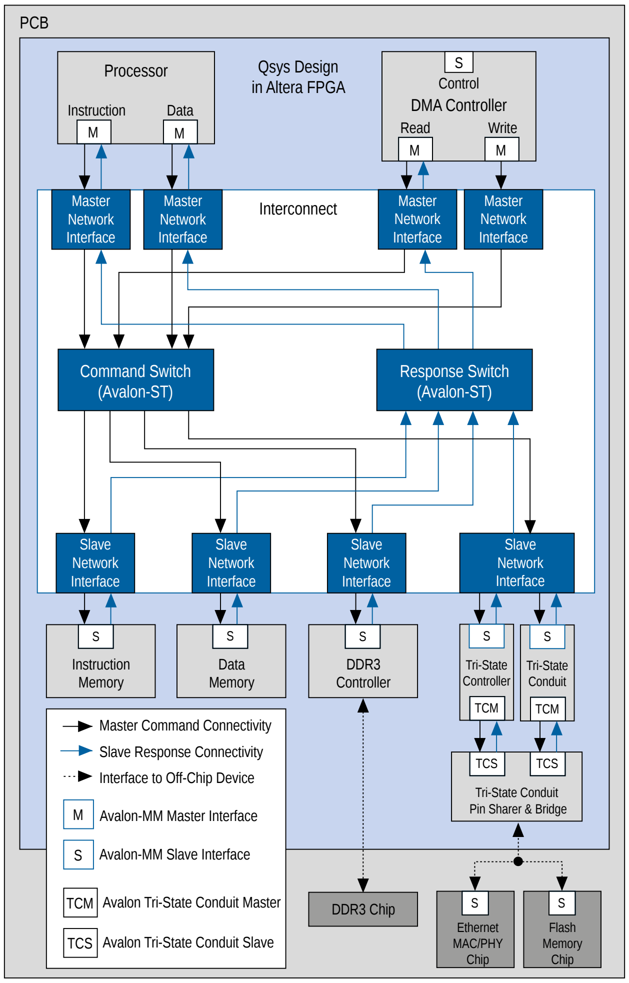
Qsys interconnect supports the following implementation scenarios:

- Any number of components with master and slave interfaces. The master-to-slave relationship can be one-to-one, one-to-many, many-to-one, or many-to-many.
- Master and slaves of different data widths.
- Master and slaves operating in different clock domains.
- IP Components with different interface properties and signals. Qsys adapts the component interfaces so that interfaces with the following differences can be connected:
  - Avalon and AXI interfaces that use active-high and active-low signalling. AXI signals are active high, except for the reset signal.
  - Interfaces with different burst characteristics.
  - Interfaces with different latencies.
  - Interfaces with different data widths.
  - Interfaces with different port signatures.

  Note: AXI3/4 to AXI3/4 interface connections declare a fixed set of signals with variable latency, so there is no need for adapting between active-high/low signalling, burst characteristics, different latencies, or port signatures. Some adaptation is necessary when going to or from Avalon interfaces.

Figure 8-1 illustrates the Qsys interconnect for an Avalon-MM system with multiple masters. In this example, there are two components mastering the system, a processor and a DMA controller, each with two master interfaces. The masters connect through the Qsys interconnect to several slaves in the Qsys system. The blue blocks represent interconnect components. The dark grey boxes indicate items outside of the Qsys system and the Quartus II software design, and show how component interfaces can be exported and connected to external devices.

Figure 8-1: Qsys interconnect—System Example


PCB
Processor
Instruction
Data
M
Qsys Design in Altera FPGA
S
Control
DMA Controller
Read
Write
Master Network Interface
Interconnect
Command Switch (Avalon-ST)
Response Switch (Avalon-ST)
Slave Network Interface
Instruction Memory
Data Memory
DDR3 Controller
Tri-State Controller
Tri-State Conduit
TCM
TCS
Tri-State Conduit Pin Sharer & Bridge
Master Command Connectivity
Slave Response Connectivity
Interface to Off-Chip Device
Avalon-MM Master Interface
Avalon-MM Slave Interface
Avalon Tri-State Conduit Master
Avalon Tri-State Conduit Slave
DDR3 Chip
Ethernet MAC/PHY Chip
Flash Memory Chip

Packet Format for Memory-Mapped Interfaces

The Qsys packet format supports Avalon, AXI, and APB transactions. Memory-mapped transactions between masters and slaves are encapsulated in Qsys packets. For Avalon systems without AXI or APB interfaces, some fields are ignored or removed.

Qsys Packet Format

The fields of the Qsys packet format are variable length to minimize the resources used. However, if the majority of components in a design have a single data width, for example 32-bits, and a single component has a data width of 64-bits, Qsys inserts a width adapter to accommodate 64-bit transfers.

Table 8-1 describes the fields of the Qsys packet that encapsulate the memory-mapped master commands and memory-mapped slave responses.

Table 8-1: Qsys Packet Format

Field	Description
`Address`	Specifies the byte address for the lowest byte in the current cycle. There are no restrictions on address alignment.
`Size`	Encodes the run-time size of the transaction. In conjunction with address, this field describes the segment of the payload that contains valid data for a beat within the packet.
`Address Sideband`	Carries “address” sideband signals. The interconnect passes this field from master to slave. This field is valid for each beat in a packet, even though it is only produced and consumed by an address cycle. Up to 8-bit sideband signals are supported for both read and write address channels.
`Cache`	Carries the AXI cache signals.
`Transaction (Exclusive)`	Indicates whether the transaction has exclusive access.
`Transaction (Posted)`	Used to indicate non-posted writes (writes that require responses).
`Data`	For command packets, carries the data to be written. For read response packets, carries the data that has been read.
`Byteenable`	Specifies which symbols are valid. AXI can issue or accept any byteenable pattern. For compatibility with Avalon, Altera recommends that you use the following legal values for 32-bit data transactions between Avalon masters and slaves: • **1111**—Writes full 32 bits • **0011**—Writes lower 2 bytes • **1100**—Writes upper 2 bytes • **0001**—Writes byte 0 only • **0010**—Writes byte 1 only • **0100**—Writes byte 2 only • **1000**—Writes byte 3 only
`Source_ID`	The ID of the master or slave that initiated the command or response.
`Destination_ID`	The ID of the master or slave to which the command or response is directed.
`Response`	Carries the AXI response signals.
`Thread ID`	Carries the AXI transaction ID values.
`Byte count`	The number of bytes remaining in the transaction, including this beat. Number of bytes requested by the packet.

Field	Description
`Burstwrap`	The burstwrap value specifies the wrapping behavior of the current burst. The burstwrap value is of the form $2^{<n>}$ -1. The following types are defined: • Variable wrap–Variable wrap bursts can wrap at any integer power of 2 value. When the burst reaches the wrap boundary, it wraps back to the previous burst boundary so that only the low order bits are used for addressing. For example, a burst starting at address 0x1C, with a burst wrap boundary of 32 bytes and a burst size of 20 bytes, would write to addresses 0x1C, 0x0, 0x4, 0x8, and 0xC. • For a burst wrap boundary of size <*m*>, `Burstwrap` = <*m*> - 1, or for this case `Burstwrap` = (32 - 1) = 31 which is 2^5 -1. • For AXI masters, the burstwrap boundary value (m) based on the different `AXBURST`: • Burstwrap set to all 1's. For example, for a 6-bit burstwrap, burstwrap is `6'b111111`. • For `WRAP` bursts, burstwrap = AXLEN * size – 1 • For `FIXED` bursts, burstwrap = size – 1 • Sequential–Sequential bursts increment the address for each transfer in the burst. For sequential bursts, the `Burstwrap` field is set to all 1s. For example, with a 6-bit `Burstwrap` field, the value for a sequential burst is 6'b111111 or 63, which is 2^6 - 1. For Avalon masters, Qsys adaptation logic sets a hardwired value for the burstwrap field, according the declared master burst properties. For example, for a master that declares sequential bursting, the burstwrap field is set to ones. Similarly, masters that declare burst have their burstwrap field set to the appropriate constant value. AXI masters choose their burst type at run-time, depending on the value of the `AW` or `ARBURST` signal. The interconnect calculates the burstwrap value at run-time for AXI masters.
`Protection`	Access level protection. When the lowest bit is 0, the packet has normal access. When the lowest bit is 1, the packet has privileged access. For Avalon-MM interfaces, this field maps directly to the privileged access signal, which allows an memory-mapped master to write to an on-chip memory ROM instance. The other bits in this field support AXI secure accesses.
`QoS`	QoS (Quality of Service Signaling) is a 4-bit field that is part of the AXI4 interface that carries QoS information for the packet from the AXI master to the AXI slave. Transactions from AXI3 and Avalon masters have the default value `4'b0000`, that indicates that they are not participating in the QoS scheme. QoS values are dropped for slaves that do not support QoS.
`Data sideband`	Carries data sideband signals for the packet. On a write command, the data sideband directly maps to `WUSER`. On a read response, the data sideband directly maps to `RUSER`. On a write response, the data sideband directly maps to `BUSER`.

Transaction Types for Memory-Mapped Interfaces

Table 8-2 describes the information carried by each bit in the packet format's transaction field.

Table 8-2: Transaction Types for Memory-Mapped Interfaces

Bit	Name	Definition
0	PKT_TRANS_READ	When asserted, indicates a read transaction.
1	PKT_TRANS_COMPRESSED_READ	For read transactions, specifies whether or not the read command can be expressed in a single cycle, that is whether or not it has all `byteenables` asserted on every cycle.
2	PKT_TRANS_WRITE	When asserted, indicates a write transaction.
3	PKT_TRANS_POSTED	When asserted, no response is required.
4	PKT_TRANS_LOCK	When asserted, indicates arbitration is locked. Applies to write packets.

Interconnect Domains

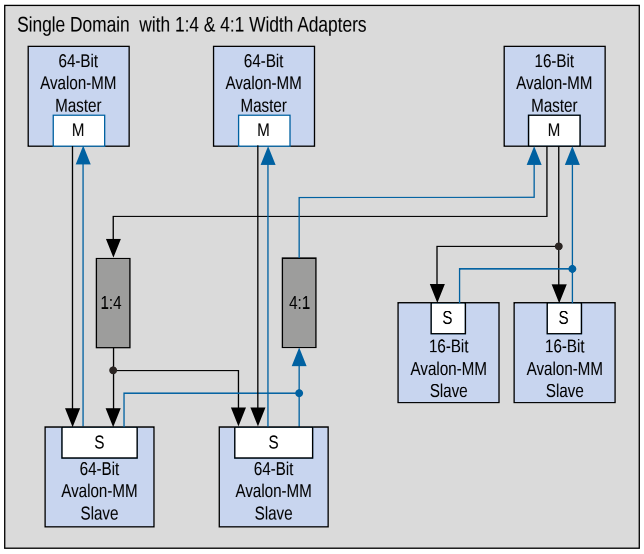
An interconnect domain is a group of connected memory-mapped masters and slaves that share the same interconnect. The components in a single interconnect domain share the same packet format.

Using One Domain with Width Adaptation

When one of the masters in a system connects to all of the slaves, Qsys creates a single domain with two packet formats: one with 64-bit data, and one with 16-bit data. A width adapter manages accesses between the 16-bit master and 64-bit slaves.

Figure 8-2 illustrates a Qsys system that includes two 64-bit masters that access two 64-bit slaves. It also includes one 16-bit master, that accesses two 16-bit slaves and two 64-bit slaves. The 16-bit Avalon master connects through a 1:4 adapter, then a 4:1 adapter to reach its 16-bit slaves.

Figure 8-2: One Domain with 1:4 and 4:1 Width Adapters

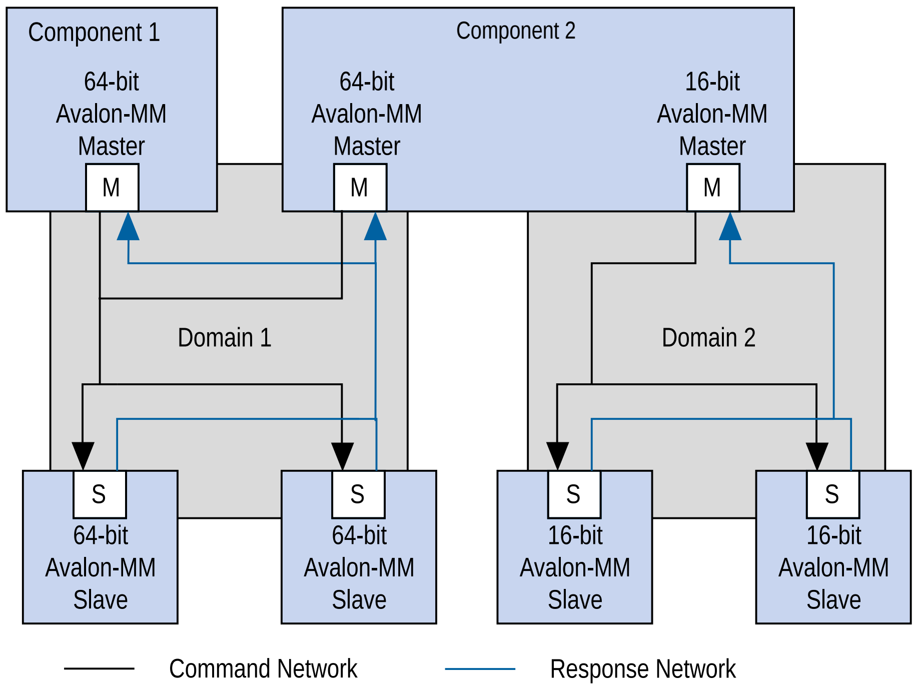


Using Two Separate Domains

In this example, a second domain includes one 16-bit master connected to two 16-bit slaves. Because the interfaces in Domain 1 and Domain 2 do not share any connections, Qsys can optimize the packet format for the two separate domains. In this example, the first domain uses a 64-bit data width and the second domain uses 16-bit data.

Figure 8-3 illustrates the use of two separate domains. The first domain includes two 64-bit masters connected to two 64-bit slaves.

Figure 8-3: Two Separate Domains

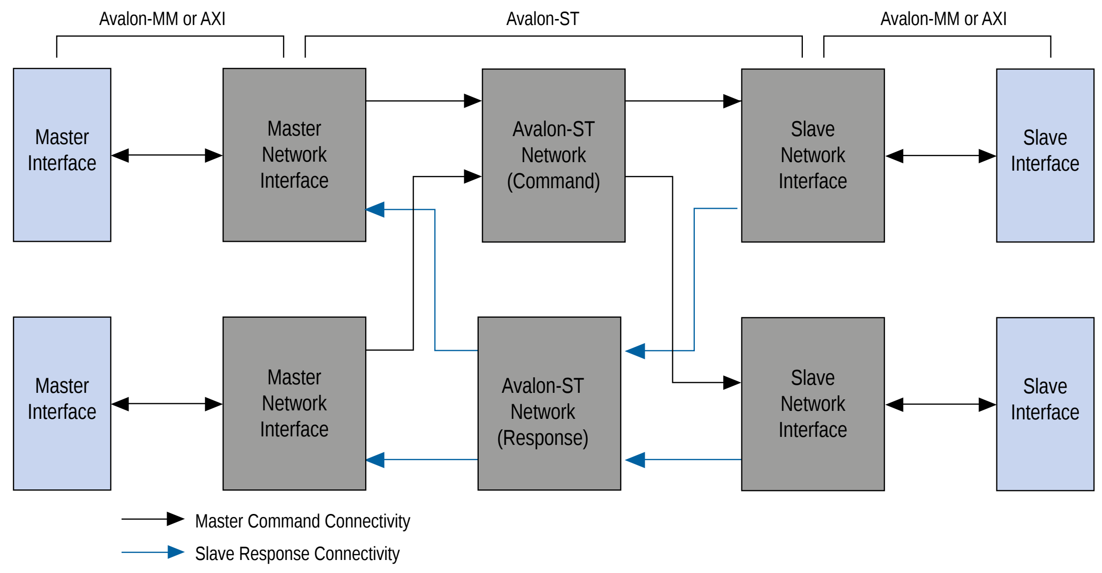


Qsys Transformations

The memory-mapped master and slave components connect to network interface modules that encapsulate the transaction in Avalon-ST packets. The memory-mapped interfaces have no information about the encapsulation or the function of the layer transporting the packets and simply operate in accordance with memory-mapped protocol, using the read and write signals and transfers as defined in the Avalon or AXI interface specification.

Figure 8-4 provides a detailed view of the transformation that occurs when you generate a Qsys system with memory-mapped master and slave components. It shows a Qsys system with memory-mapped master and slave components. The Qsys components that implement the blocks appear shaded in *Master Network Interfaces* and *Slave Network Interfaces*.

Figure 8-4: Qsys Transform from Memory-Mapped Interfaces to Avalon-ST



Related Information

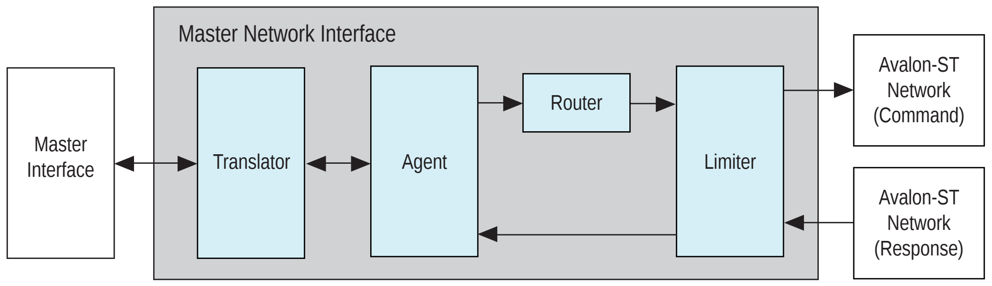
- **Master Network Interfaces** on page 8-9
- **Slave Network Interfaces** on page 8-11

Master Network Interfaces

Avalon network interfaces drive default values for the `QoS` and `BUSER`, `WUSER`, and `RUSER` packet fields in the master agent, and drop the packet fields in the slave agent.

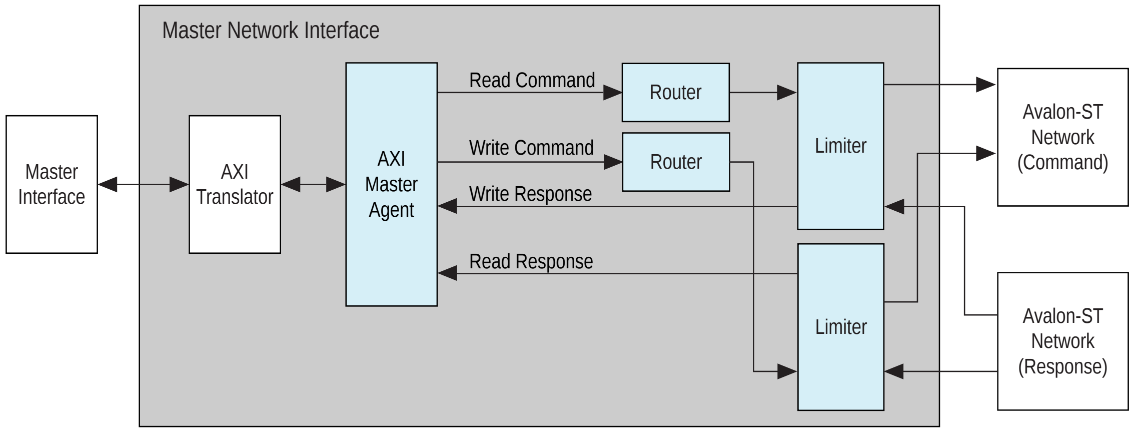
Figure 8-5 shows the Avalon-MM Master network interface.

Figure 8-5: Avalon-MM Master Network Interface



An AXI4 master supports `INCR` bursts up to 256 beats, QoS signals, and Data Sideband signals. **Figure 8-6** shows the AXI Master Network Interface.

Figure 8-6: AXI Master Network Interface



Avalon-MM Master Agent

The Avalon-MM Master Agent translates Avalon-MM master transactions into Qsys command packets and translates the Qsys Avalon-MM slave response packets into Avalon-MM responses.

Avalon-MM Master Translator

The Avalon-MM Master Translator interfaces with an Avalon-MM master component and converts the Avalon-MM master interface to a simpler representation for use in Qsys.

The Avalon-MM Master translator performs the following functions:

- Translates active low signalling to active high signalling
- Inserts wait states to prevent an Avalon-MM master from reading invalid data
- Translates word and symbol addresses
- Translates word and symbol burst counts
- Manages re-timing and re-sequencing bursts
- Removes unnecessary address bits

AXI Master Agent

An AXI Master Agent accepts AXI commands and produces Qsys command packets. It also accepts Qsys response packets and converts those into AXI responses. This component has separate packet channels for read commands, write commands, read responses, and write responses. Avalon master agent drives the `QoS` and BUSER, WUSER, and RUSER packet fields with default values `AXQO` and `b0000`, respectively.

Note: For signal descriptions, refer to *Qsys Packet Format*.

Related Information

Qsys Packet Format on page 8-3

AXI Translator

AXI4 allows some signals to be omitted from interfaces. The translator bridges between these "incomplete" AXI4 interfaces and the "complete" AXI4 interface on the network interfaces.

The AXI translator is inserted for both AXI4 master and slave, and performs the following functions:

- Matches ID widths between the master and slave in 1x1 systems.
- Drives default values as defined in the *AMBA Protocol Specifications* for missing signals.
- Performs lock transaction bit conversion when an AXI3 master connects to an AXI4 slave in 1x1 systems.

Related Information

AMBA Protocol Specifications

APB Master Agent

An APB master agent accepts APB commands and produces or generates Qsys command packets. It also converts Qsys response packets to APB responses.

APB Slave Agent

An APB slave agent issues resulting transaction to the APB interface. It also accepts creates Qsys response packets.

APB Translator

An APB peripheral does not require `pslverr` signals to support additional signals for the APB debug interface.

The APB translator is inserted for both the master and slave and performs the following functions:

- Sets the response value default to `OKAY` if the APB slave does not have a `pslverr` signal.
- Turns on or off additional signals between the APB debug interface, which is used with HPS (Altera SoC's Hard Processor System).

Memory-Mapped Router

The Memory-Mapped Router routes command packets from the master to the slave, and response packets from the slave to the master. For master command packets, the router uses the address to set the `Destination_ID` and Avalon-ST channel. For the slave response packet, the router uses the `Destination_ID` to set the Avalon-ST channel. The demultiplexers use the Avalon-ST channel to route the packet to the correct destination.

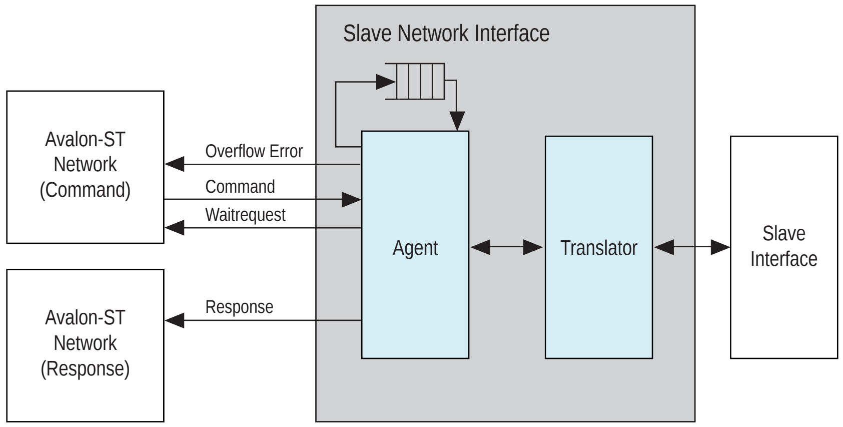
Memory-Mapped Traffic Limiter

The Memory-Mapped Traffic Limiter ensures the responses arrive in order. It prevents any command from being sent if the response could conflict with the response for a command that has already been issued. By guaranteeing in-order responses, the Traffic Limiter simplifies the response network.

Slave Network Interfaces

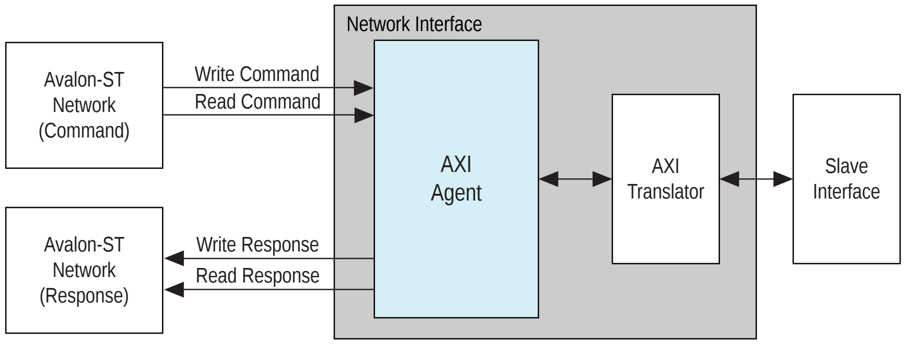
Figure 8-7 shows an Avalon slave network interface.

Figure 8-7: Avalon-MM Slave Network Interface



An AXI4 slave supports up to 256 beat `INCR` bursts, QoS signals, and data sideband signals. **Figure 8-8** shows the AXI slave network interface.

Figure 8-8: AXI Slave Network Interface



Avalon-MM Slave Translator

The Avalon-MM Slave Translator interfaces to an Avalon-MM slave component as the *Avalon-MM Slave Network Interface* figure illustrates. It converts the Avalon-MM slave interface to a simplified representation that the Qsys network can use.

An Avalon-MM Merlin Slave Translator performs the following functions:

- Drives the `begintransfer`, `beginbursttransfer`, and `byteenable` signals.
- Supports Avalon-MM slaves that operate using fixed timing and or slaves that use the `readdatavalid` signal to identify valid data.
- Translates the `read`, `write`, and `chipselect` signals into the representation that the Avalon-ST slave response network uses.
- Converts active low signals to active high signals.
- Translates word and symbol addresses and burstcounts.
- Handles burstcount timing and sequencing.
- Removes unnecessary address bits.

Related Information

Slave Network Interfaces on page 8-11

AXI Translator

AXI4 allows some signals to be omitted from interfaces. The translator bridges between these "incomplete" AXI4 interfaces and the "complete" AXI4 interface on the network interfaces.

The AXI translator is inserted for both AXI4 master and slave, and performs the following functions:

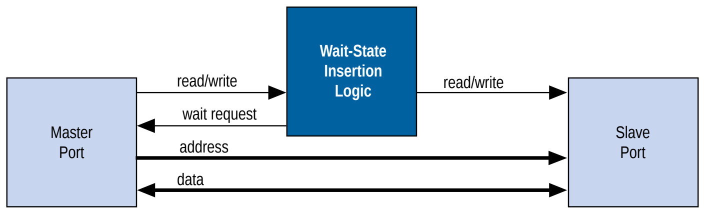
- Matches ID widths between master and slave in 1x1 systems.
- Drives default values as defined in the *AMBA Protocol Specifications* for missing signals.
- Performs lock transaction bit conversion when an AXI3 master connects to an AXI4 slave in 1x1 systems.

Wait State Insertion

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

Wait state insertion logic is a small finite-state machine that translates control signal sequencing between the slave side and the master side. Qsys interconnect can force a master to wait for the wait state needs of a slave. For example, arbitration logic in a multimaster system. Qsys generates wait state insertion logic based on the properties of all slaves in the system.

Figure 8-9: Block Diagram of Wait State Insertion Logic for One Master and One Slave



Avalon-MM Slave Agent

The Avalon-MM Slave Agent accepts command packets and issues the resulting transactions to the Avalon interface. For pipelined slaves, an Avalon-ST FIFO stores information about pending transactions. The size of this FIFO is the maximum number of pending responses that you specify when creating the slave component. The Avalon-MM Slave Agent also `backpressures` the Avalon-MM master command interface when the FIFO is full if the slave component includes the `waitrequest` signal.

AXI Slave Agent

An AXI Slave Agent works similar to a master agent in reverse. The AXI slave Agent accepts Qsys command packets to create AXI commands, and accepts AXI responses to create Qsys response packets. This component has separate packet channels for read commands, write commands, read responses, and write responses.

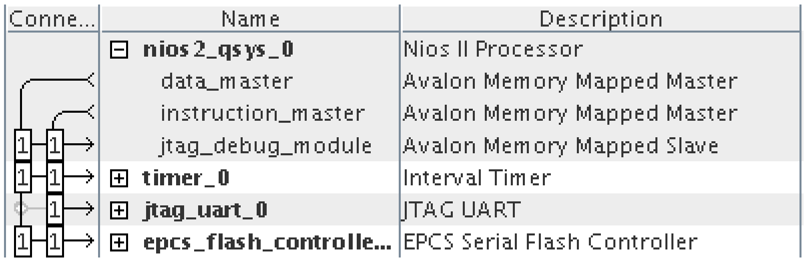
Arbitration

When multiple masters contend for access to a slave, Qsys automatically inserts arbitration logic which grants access in fairness-based, round-robin order.

In a fairness-based arbitration scheme, each master has an integer value of transfer *shares* with respect to a slave. One share represents permission to perform one transfer. The default arbitration scheme is equal share round-robin that grants equal, sequential access to all requesting masters. You can change the arbitration scheme to weighted round-robin by specifying a relative number of arbitration shares to the masters that access a particular slave. AXI slaves have separate arbitration for their independent read and write channels, and the **Arbitration Shares** setting affects both the read and write arbitration. To display arbitration settings, right-click an instance on the **System Contents** tab, and then click **Show Arbitration Shares**.

Figure 8-10 illustrates arbitration shares indicated in the **Connections** column.

Figure 8-10: Arbitration Settings on the System Contents Tab



Conne...	Name	Description
	⊟ **nios2_qsys_0**	Nios II Processor
	data_master	Avalon Memory Mapped Master
	instruction_master	Avalon Memory Mapped Master
1 1	jtag_debug_module	Avalon Memory Mapped Slave
1 1	⊞ **timer_0**	Interval Timer
1	⊞ **jtag_uart_0**	JTAG UART
1 1	⊞ **epcs_flash_controlle...**	EPCS Serial Flash Controller

Arbitration Rules

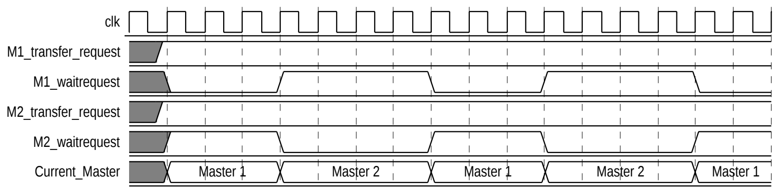
The rules by which the arbiter grants access to masters are described as: fairness-based shares, round-robin scheduling, and burst transfers.

Fairness-Based Shares

In a fairness-based arbitration scheme, each master-to-slave connection provides a transfer share count. This count is a request for the arbiter to grant a specific number of transfers to this master before giving control to a different master. One share represents permission to perform one transfer.

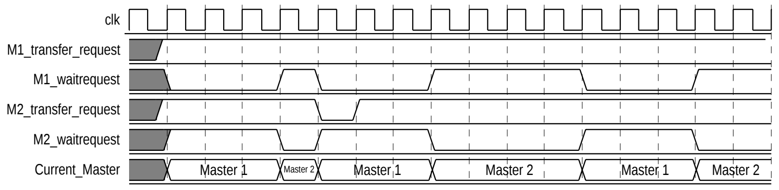
For example, for two masters that continuously attempt to perform back-to-back transfers to a slave, the arbiter grants Master 1 access for three transfers, and Master 2 for four transfers. This cycle repeats indefinitely. Figure 8-11 shows each master's transfer request output, wait request input (which is driven by the arbiter logic), and the current master with control of the slave.

Figure 8-11: Arbitration of Continuous Transfer Requests from Two Masters



If a master stops requesting transfers before it exhausts its shares, it forfeits all of its remaining shares, and the arbiter grants access to another requesting master, as shown in Figure 8-12. After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbiter grants access back to Master 1, which gets a replenished supply of shares.

Figure 8-12: Arbitration of Two Masters with a Gap in Transfer Requests



Round-Robin Scheduling

When multiple masters contend for access to a slave, the arbiter grants shares in Round-Robin order. At every slave transaction, only requesting masters are included in the arbitration.

Burst Transfers

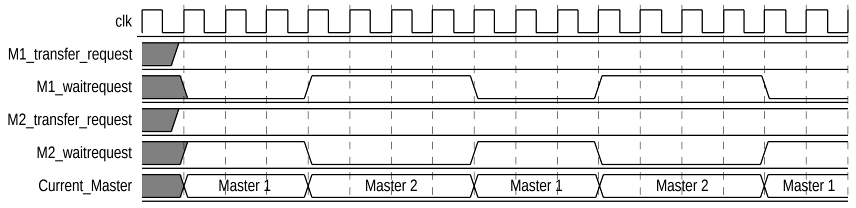
For burst transfer arbitration, there is a one-to-one relationship between a single share and a transaction, so that arbitration shares are respected during burst transactions. For example, for an arbitration share of 3, a master is granted 3 burst transactions. Once a burst begins between a master-slave pair, arbiter logic does not allow any other master to access the slave until the burst completes.

Arbitration Examples

Arbitration can occur with continuous transfer requests from two masters, or with two masters with a gap in a transfer request.

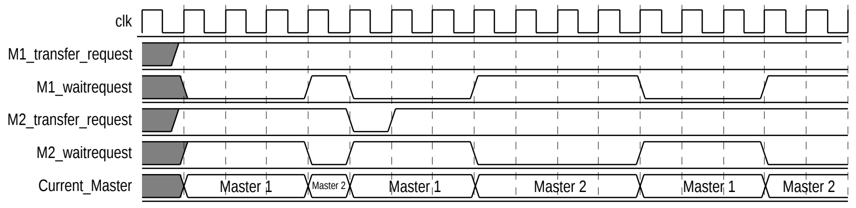
Figure 8-13 illustrates the timing for two Avalon-MM masters continuously accessing a single Avalon-MM slave to perform back-to-back transfers. Master 1 has three shares and Master 2 has four shares.The arbiter grants Master 1 access for three transfers, then Master 2 for four transfers. This cycle repeats indefinitely.

Figure 8-13: Arbitration of Continuous Transfer Requests from Two Masters



If a master stops requesting transfers before it exhausts its shares, it forfeits all of its remaining shares, and the arbiter grants access to another requesting master, as shown in Figure 8-14. After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbiter grants access back to Master 1, which gets three shares.

Figure 8-14: Arbitration of Two Masters with a Gap in Transfer Requests

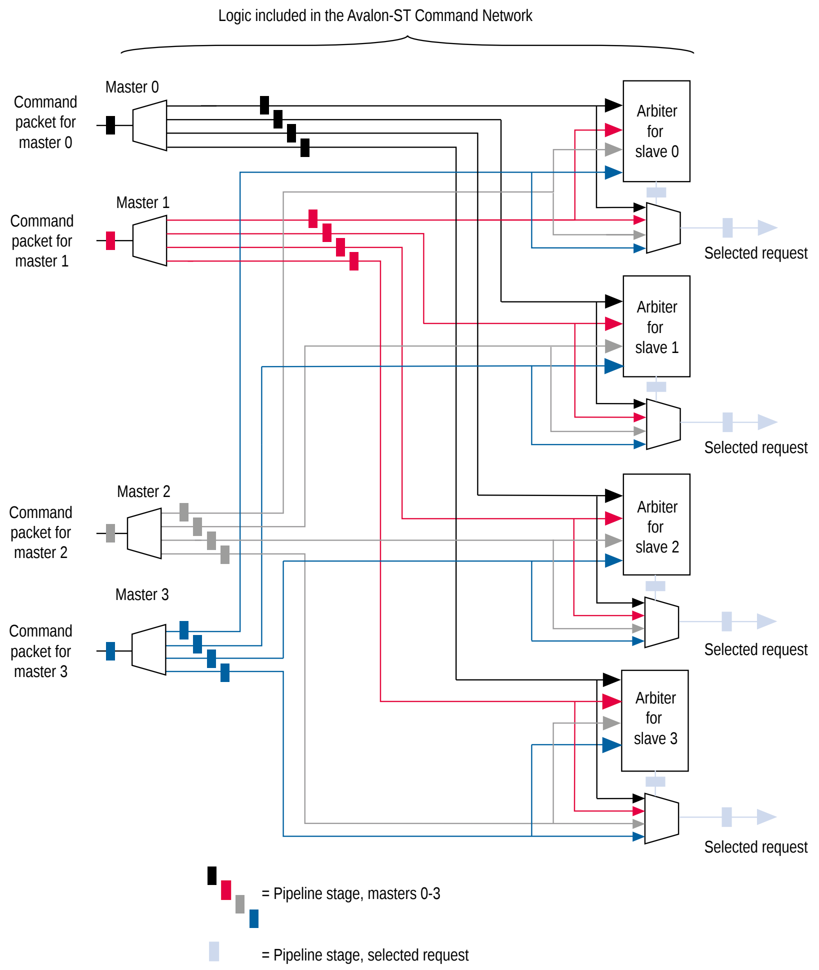


Memory-Mapped Arbiter

The input to the Memory-Mapped Arbiter is the command packet for all masters requesting access to a particular slave. The arbiter outputs the channel number for the selected master. This channel number controls the output of a multiplexer that selects the slave device. The figure below illustrates this logic.

In **Figure 8-15**, four Avalon-MM masters connect to four Avalon-MM slaves. In each cycle, an arbiter positioned in front of each Avalon-MM slave selects among the requesting Avalon-MM masters.

Figure 8-15: Arbitration Logic

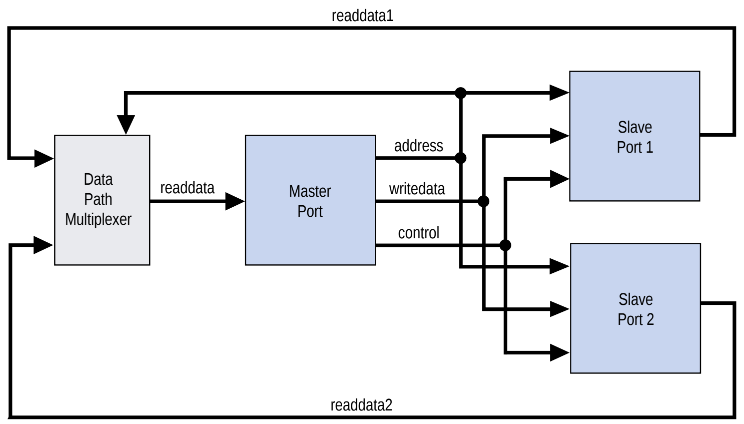

Logic included in the Avalon-ST Command Network
Master 0
Command packet for master 0
Master 1
Command packet for master 1
Master 2
Command packet for master 2
Master 3
Command packet for master 3
Arbiter for slave 0
Arbiter for slave 1
Arbiter for slave 2
Arbiter for slave 3
Selected request
Selected request
Selected request
Selected request
= Pipeline stage, masters 0-3
= Pipeline stage, selected request

If you specified a **Limit interconnect pipeline stages to** parameter greater than zero on the Qsys **Project Settings** tab, the output of the Arbiter is registered. Registering this output reduces the amount of combinational logic between the master and interconnect, increasing the f_{MAX} of the system.

Datapath Multiplexing

Datapath multiplexing logic drives the `writedata` signal from the granted master to the selected slave, and the `readdata` signal from the selected slave back to the requesting master. Qsys generates separate datapath multiplexing logic for every master in the system (`readdata`), and for every slave in the system (`writedata`). Qsys does not generate multiplexing logic if it is not needed.

Figure 8-16: Block Diagram of Datapath Multiplexing Logic for One Master and Two Slaves



Width Adaptation

Qsys width adaptation converts between Avalon memory-mapped master and slaves with different data and byte enable widths, and manages the run-time size requirements of AXI. Width adaptation for AXI to Avalon interfaces is also supported.

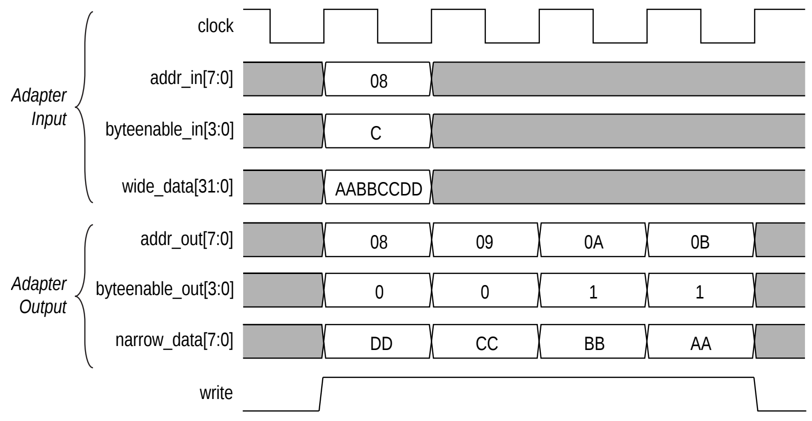
Memory-Mapped Width Adapter

The Memory-Mapped Width Adapter is used in the Avalon-ST domain and operates with information contained in the packet format.

The memory-mapped width adapter accepts packets on its sink interface with one data width and produces output packets on its source interface with a different data width. The ration of the narrow data width must be a power of two, such as 4:1, 8:1, and 16:1. The ratio of the wider data width to the narrower width must be a power of two, such as 4:1, 8:1, and 16:1 These output packets may have a different size if the input size exceeds the output data bus width, or if data packing is enabled.

This adapter assumes that the field ordering of the input and output packets is the same, with the only difference being the width of the data and accompanying byte enable fields. When the width adapter converts from wide data to narrow data, the narrower data is transmitted over several beats. The first output beat contains the lowest addressed segment of the input data and byte enables. Figure 8-17 illustrates the timing for a 4:1 width adapter. When the width adapter converts from narrow data to wide data, each input beat's data and byte enables are copied to the appropriate segment of the wider output data and byte enables signals.

Figure 8-17: Width Adapter Timing for a 4:1 Adapter



AXI Wide to Narrow Adaptation

For all cases of AXI wide to narrow adaptation, read data is repacked to match the original size. Responses are merged, with the following error precedence: `DECERR`, `SLVERR`, `OKAY`, and `EXOKAY`.

Table 8-3 describes burst behavior for AXI wide-to-narrow adaptation.

Table 8-3: Wide to Narrow Adaptation (Downsizing)

Burst Type	Behavior
`Incrementing`	If the transaction size is less than or equal to the output width, the burst is unmodified. Otherwise, it is converted to an incrementing burst with a larger length and size equal to the output width. If the resulting burst is unsuitable for the slave, the burst is converted to multiple sequential bursts of the largest allowable lengths. For example, for a 2:1 downsizing ratio, an `INCR9` burst is converted into `INCR16 + INCR2` bursts. This is true if the maximum burstcount a slave can accept is 16, which is the case for AXI3 slaves. Avalon slaves have a maximum burstcount of 64.
`Wrapping`	If the transaction size is less than or equal to the output width, the burst is unmodified. Otherwise, it is converted to a wrapping burst with a larger length, with a size equal to the output width. If the resulting burst is unsuitable for the slave, the burst is converted to multiple sequential bursts of the largest allowable lengths; respecting wrap boundaries. For example, for a 2:1 downsizing ratio, a `WRAP16` burst is converted into two or three `INCR` bursts, depending on the address.
`Fixed`	If the transaction size is less than or equal to the output width, the burst is unmodified. Otherwise, it is converted into repeated sequential bursts over the same addresses. For example, for a 2:1 downsizing ratio, a FIXED single burst is converted into an `INCR2` burst.

AXI Narrow to Wide Adaptation

Table 8-4 describes burst behavior for AXI narrow-to-wide adaptation.

Table 8-4: Narrow to Wide Adaptation (Upsizing)

Burst Type	Behavior
`Incrementing`	The burst (and its response) passes through unmodified. Data and write strobes are placed in the correct output segment.
`Wrapping`	The burst (and its response) passes through unmodified.
`Fixed`	The burst (and its response) passes through unmodified.

Burst Transfers

Avalon-MM and AXI burst transactions grant a master uninterrupted access to a slave for a specified number of transfers. The master specifies the number of transfers when it initiates the burst. Once a burst begins between a master and slave pair, arbiter logic is locked until the burst completes. For burst masters, the length of the burst is the number of cycles that the master has access to the slave, and the selected arbitration shares have no effect.

Memory-Mapped Burst Adapter

The Qsys interconnect uses the memory-mapped burst adapter to accommodate the burst capabilities of each interface in the system, including interfaces that do not support burst transfers.

The maximum burst length for each interface is a property of the component interface and is independent of other interfaces in the system. Therefore, a particular master might be capable of initiating a burst longer than a slave's maximum supported burst length. In this case, the burst adapter translates the large master burst into smaller bursts, or into individual slave transfers if the slave does not support bursting. Until the master completes the burst, the arbiter logic prevents other masters from accessing the target slave. For example, if a master initiates a burst of 16 transfers to a slave with maximum burst length of 8, the burst adapter initiates 2 bursts of length 8 to the slave.

Note: AXI masters can issue burst types that Avalon cannot accept, for example, fixed bursts. In this case, the burst adapter converts the fixed burst into a sequence of transactions to the same address.

Note: For AXI4 slaves, Qsys allows 256-beat `INCR` bursts, though you must ensure that 256-beat narrow-sized `INCR` bursts are shortened to 16-beat narrow-sized INCR bursts for AXI3 slaves.

Avalon-MM masters always issue addresses that are aligned to the size of the transfer. However, in some cases, when a narrow-to-wide width adaptation is used, the resulting address may be unaligned. In the case of unaligned addresses, the burst adapter issues the maximum possible sized bursts, with appropriate byte enables, to bring the burst-in-progress up to an aligned slave address. Then, it completes the burst on aligned addresses.

The burst adapter supports variable wrap or sequential burst types to accommodate the different properties of memory-mapped masters. Some bursting masters can issue more than one burst type.

Burst adaptation is available for Avalon to Avalon, Avalon to AXI, and AXI to Avalon connections, and AXI to AXI connections. For information about AXI-to-AXI adaptation, refer to *AXI Wide to Narrow Adaptation*

Note: For AXI4 to AXI3 connections, Qsys follows an AXI4 256 burst length to AXI3 16 burst length.

Burst Adaptation: AXI to Avalon on page 8-20

Burst Adaptation: Avalon to AXI on page 8-20

Burst Adaptation: AXI to Avalon

Table 8-5 specifies the behavior when converting between AXI and Avalon burst types.

Table 8-5: Burst Adaptation: AXI to Avalon

Burst Type	Behavior
`Incrementing`	**Sequential Slave** Bursts that exceed `slave_max_burst_length` are converted to multiple sequential bursts of a length less than or equal to the `slave_max_burst_length`. Otherwise, the burst is unconverted. For example, for an Avalon slave with a maximum burst length of 4, an `INCR7` burst is converted to `INCR4 + INCR3`. **Wrapping Slave** Bursts that exceed the `slave_max_burst_length` are converted to multiple sequential bursts of length less than or equal to the `slave_max_burst_length`. Bursts that exceed the wrapping boundary are converted to multiple sequential bursts that respect the slave's wrapping boundary.
`Wrapping`	**Sequential Slave** A WRAP burst is converted to multiple sequential bursts. The sequential bursts are less than or equal to the `max_burst_length` and respect the transaction's wrapping boundary **Wrapping Slave** If the WRAP transaction's boundary matches the slave's boundary, then the burst passes through. Otherwise, the burst is converted to sequential bursts that respect both the transaction and slave wrap boundaries.
`Fixed`	Fixed bursts are converted to sequential bursts of length 1 that repeatedly access the same address.
`Narrow`	All narrow-sized bursts are broken into multiple bursts of length 1.

Burst Adaptation: Avalon to AXI

Table 8-6 specifies the behavior when converting between Avalon and AXI burst types.

Table 8-6: Burst Adaptation: Avalon to AXI

Burst Type	Definition
`Sequential`	Bursts of length greater than16 are converted to multiple `INCR` bursts of a length less than or equal to16. Bursts of length less than or equal to16 are not converted.
`Wrapping`	Only Avalon masters with `alwaysBurstMaxBurst = true` are supported. The WRAP burst is passed through if the length is less than or equal to16. Otherwise, it is converted to two or more `INCR` bursts that respect the transaction's wrap boundary.

Address Decoding

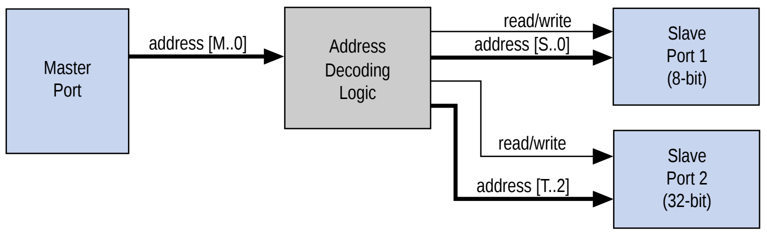
Address decoding logic forwards appropriate addresses to each slave.

Address decoding logic simplifies component design in the following ways:

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

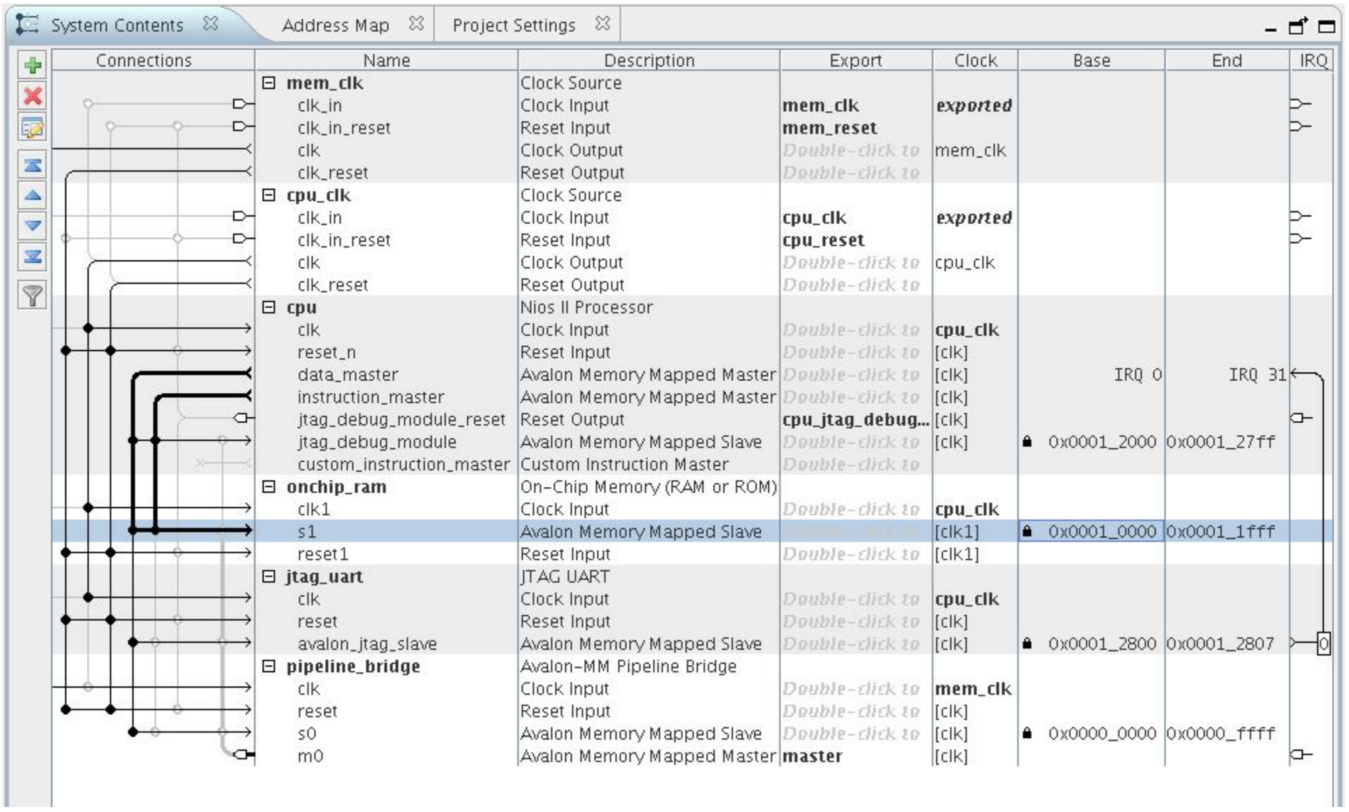
Separate address-decoding logic is generated for every master in a system. The address decoding logic processes the difference between the master address width (<`M`>) and the individual slave address widths (<`S` >) and (<`T` >). The address decoding logic also maps only the necessary master address bits to access words in each slave's address space.

Figure 8-18: Block Diagram of Address Decoding for One Master and Two Slaves



In Qsys, the base addresses are controlled by the **Base** setting of active components on the **System Contents** tab, as shown in Figure 8-19. The base address of a slave component must be a multiple of the address span of the component. This restriction is part of the Qsys interconnect to allow the address decoding logic to be efficient, and to achieve the best possible f_{MAX}.

Figure 8-19: Base Settings in Qsys Address Decoding



Connections	Name	Description	Export	Clock	Base	End	IRQ
	⊟ **mem_clk**	Clock Source					
	clk_in	Clock Input	**mem_clk**	***exported***			
	clk_in_reset	Reset Input	**mem_reset**				
	clk	Clock Output	*Double-click to*	mem_clk			
	clk_reset	Reset Output	*Double-click to*				
	⊟ **cpu_clk**	Clock Source					
	clk_in	Clock Input	**cpu_clk**	***exported***			
	clk_in_reset	Reset Input	**cpu_reset**				
	clk	Clock Output	*Double-click to*	cpu_clk			
	clk_reset	Reset Output	*Double-click to*				
	⊟ **cpu**	Nios II Processor					
	clk	Clock Input	*Double-click to*	**cpu_clk**			
	reset_n	Reset Input	*Double-click to*	[clk]			
	data_master	Avalon Memory Mapped Master	*Double-click to*	[clk]	IRQ 0	IRQ 31	
	instruction_master	Avalon Memory Mapped Master	*Double-click to*	[clk]			
	jtag_debug_module_reset	Reset Output	**cpu_jtag_debug...**	[clk]			
	jtag_debug_module	Avalon Memory Mapped Slave	*Double-click to*	[clk]	🔒 0x0001_2000	0x0001_27ff	
	custom_instruction_master	Custom Instruction Master	*Double-click to*				
	⊟ **onchip_ram**	On-Chip Memory (RAM or ROM)					
	clk1	Clock Input	*Double-click to*	**cpu_clk**			
	s1	Avalon Memory Mapped Slave		[clk1]	🔒 0x0001_0000	0x0001_1fff	
	reset1	Reset Input	*Double-click to*	[clk1]			
	⊟ **jtag_uart**	JTAG UART					
	clk	Clock Input	*Double-click to*	**cpu_clk**			
	reset	Reset Input	*Double-click to*	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	🔒 0x0001_2800	0x0001_2807	0
	⊟ **pipeline_bridge**	Avalon-MM Pipeline Bridge					
	clk	Clock Input	*Double-click to*	**mem_clk**			
	reset	Reset Input	*Double-click to*	[clk]			
	s0	Avalon Memory Mapped Slave	*Double-click to*	[clk]	🔒 0x0000_0000	0x0000_ffff	
	m0	Avalon Memory Mapped Master	**master**	[clk]			

Streaming Interfaces

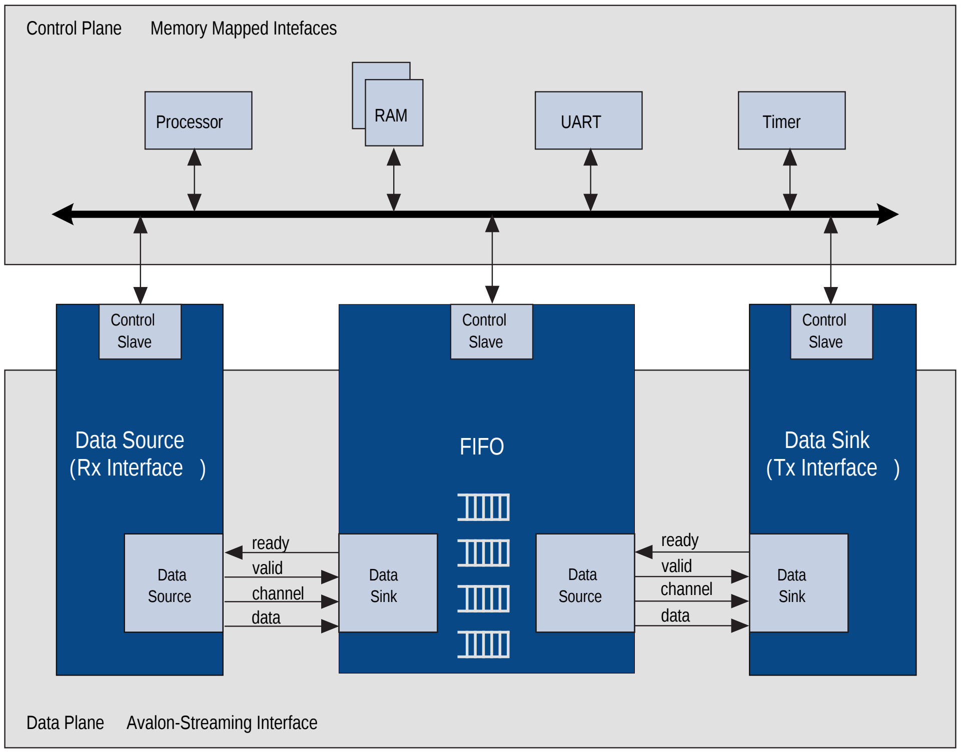
High bandwidth components with streaming data typically use Avalon-ST interfaces for the high throughput datapath. Streaming interfaces can also use memory-mapped connection interfaces to provide an access point for control. In contrast to the memory-mapped interconnect, which you can use to create a wide variety of topologies, the Avalon-ST interconnect always creates a point-to-point connection between a single data source and data sink, as the figure below illustrates.

Figure 8-20 has the following connection pairs:

- Data source in the Rx Interface transfers data to the data sink in the FIFO
- Data source in the FIFO transfers data to the `Tx Interface` data sink

In Figure 8-20, the memory-mapped interface allows a processor to access the data source, FIFO, or data sink to provide system control.

Figure 8-20: Use of the Memory-Mapped and Avalon-ST Interfaces



If your source and sink interfaces have different formats, for example, a 32-bit source and an 8-bit sink, Qsys automatically inserts the necessary adapters, which are then visible in the **System Contents** tab.

Figure 8-21 illustrates the simplest system example with an Avalon-ST connection between the source and sink. This source-sink pair includes only the `data` signal. The sink must be able to receive data as soon as the source interface comes out of reset.

Figure 8-21: Interconnect for a Simple Avalon Streaming Source-Sink Pair

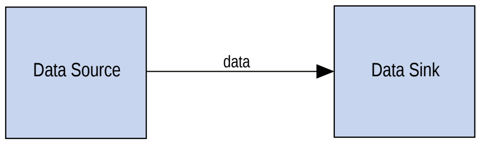
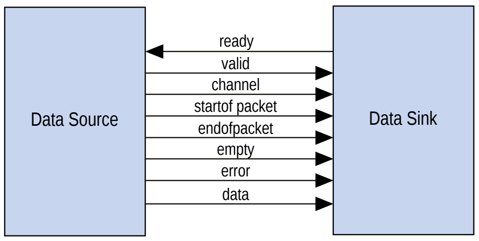


Figure 8-22 illustrates a more extensive interface that includes signals indicating the start and end of packets, channel numbers, error conditions, and back pressure.

Figure 8-22: Signals Indicating the Start and End of Packets, Channel Numbers, Error Conditions, and Backpressure



All data transfers using Avalon-ST interconnect occur synchronously to the rising edge of the associated clock interface. Throughput and frequency of a system depends on the components and how they are connected.

The Qsys Library includes a number of Avalon-ST components that you can use to create datapaths, including datapaths whose input and output streams have different properties. Generated systems that include memory-mapped master and slave components may also use these Avalon-ST components because the generation process creates an interconnect whose structure resembles a network topology, as described in *Qsys Transformations*. The following sections introduce the Avalon-ST components.

AXI streaming components are not available in Qsys, version 13.1 For details about the Avalon-ST interface protocol, refer to the *Avalon Interface Specification*.

Related Information

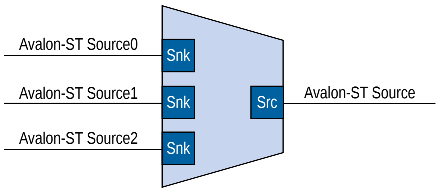
- **Avalon Interface Specification**
- **Avalon-ST Adapters** on page 8-25
- **Qsys Transformations** on page 8-8

Avalon-ST Multiplexer

The Avalon-ST Multiplexer accepts data on its Avalon-ST sink interface and multiplexes the data for transmission on its Avalon-ST source interface.

You can parameterize the multiplexer to append channel information on the source to indicate which sink is driving the source data. The multiplexer includes internal arbitration logic which selects between inputs using a round-robin arbitration algorithm. **Figure 8-23** illustrates the Avalon-ST multiplexer. Among the parameters that you can specify are the option to use packet scheduling, which guarantees that the multiplexer only changes inputs at the end of a packet.

Figure 8-23: Avalon-ST Multiplexer

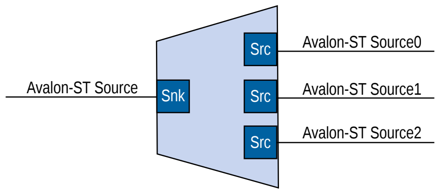


Avalon-ST Demultiplexer

The Avalon-ST Demultiplexer accepts channelized data on its sink interface, and transmits the data on one of its source interfaces.

The channel bits of the source stream indicate which port the drives the output data. Figure 8-24 illustrates the Multiplexer. Among the parameters that you can specify are the number of output ports and the width of the channel signal.

Figure 8-24: Avalon-ST Demultiplexer



Avalon-ST Adapters

Qsys automatically adds Avalon-ST adapters between two components during system generation when it detects mismatched interfaces. If you connect mismatched Avalon-ST sources and sinks, for example, a 32-bit source and an 8-bit sink, Qsys inserts the appropriate adapter type, as described below, to connect the mismatched interfaces. After generation, you can view the inserted adapters with the **Show System With Qsys Fabric Components** command on the System menu. Qsys reports the mismatched interfaces and inserted adapter with informational messages.

You can turn off the auto-inserted adapters feature by adding the `qsys_enable_avalon_streaming_transform=off` command to the **quartus.ini** file. When you turn off the auto-inserted adapters feature, if mismatched interfaces are detected during system generation, Qsys does not insert adapters and reports the mismatched interface with an error message.

Note: The auto-inserted adapters feature does not work for video IP core connections.

Data Format Adapter

The data format adapter allows you to connect interfaces that have different values for the parameters defining the data signal. One of the most common uses of this adapter is to convert data streams of different widths.

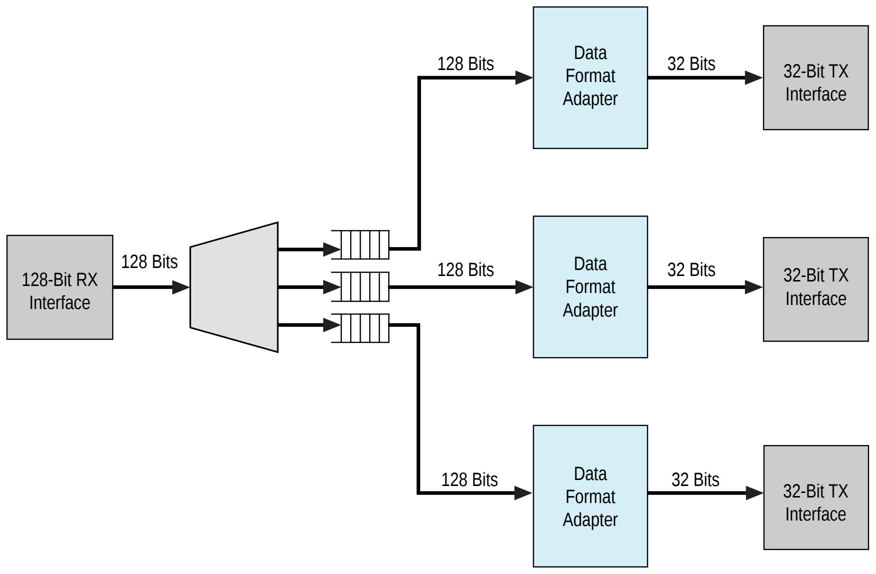
Table 8-7: Data Format Adapter Adaptations

Condition	Description of Adapter Logic
The source and sink's bits per symbol are different.	The connection cannot be made.

Condition	Description of Adapter Logic
The source and sink have a different number of symbols per beat.	The adapter converts the source's width to the sink's width. If the adaptation is from a wider to a narrower interface, a beat of data at the input corresponds to multiple beats of data at the output. If the input `error` signal is asserted for a single beat, it is asserted on output for multiple beats. If the adaptation is from a narrow to a wider interface, multiple input beats are required to fill a single output beat, and the output `error` is the logical OR of the input `error` signal.

Figure 8-25 shows a data format adapter that allows a connection between a 128-bit input data stream and three 32-bit output data streams.

Figure 8-25: Avalon Streaming Interconnect with Data Format Adapter



Timing Adapter

The timing adapter allows you to connect component interfaces that require a different number of cycles before driving or receiving data. This adapter inserts a FIFO buffer between the source and sink to buffer data or pipeline stages to delay the back pressure signals. You can also use the timing adapter to connect interfaces that support the `ready` signal, and those that do not. The timing adapter treats all signals other than the `ready` and `valid` signals as payload, and simply drives them from the source to the sink.

Table 8-8: Timing Adapter Adaptations

Condition	Adaptation
The source has `ready`, but the sink does not.	In this case, the source can respond to `backpressure`, but the sink never needs to apply it. The `ready` input to the source interface is connected directly to logical 1.
The source does not have `ready`, but the sink does.	The sink may apply `backpressure`, but the source is unable to respond to it. There is no logic that the adapter can insert that prevents data loss when the source asserts `valid` but the sink is not ready. The adapter provides simulation time error messages and an error indication if data is ever lost. The user is presented with a warning, and the connection is allowed.
The source and sink both support backpressure, but the sink's ready latency is greater than the source's.	The source responds to `ready` assertion or deassertion faster than the sink requires it. A number of pipeline stages equal to the difference in ready latency are inserted in the `ready` path from the sink back to the source, causing the source and the sink to see the same cycles as `ready` cycles.
The source and sink both support backpressure, but the sink's ready latency is less than the source's.	The source cannot respond to `ready` assertion or deassertion in time to satisfy the sink. A buffer whose depth is equal to the difference in ready latency is inserted to compensate for the source's inability to respond in time.

Channel Adapter

The channel adapter provides adaptations between interfaces that have different support for the `channel` signal, the maximum number of channels supported, or channel-related parameters.

Table 8-9: Channel Adapter Adaptations

Condition	Description of Adapter Logic
The source uses channels, but the sink does not.	You are given a warning at generation time. The adapter provides a simulation error and signals an error for data for any channel from the source other than 0.
The sink has channel, but the source does not.	You are given a warning, and the channel inputs to the sink are all tied to a logical 0.
The source and sink both support channels, and the source's maximum number of channels is less than the sink's.	The source's channel is connected to the sink's channel unchanged. If the sink's channel signal has more bits, the higher bits are tied to a logical 0.

Condition	Description of Adapter Logic
The source and sink both support channels, but the source's maximum number of channels is greater than the sink's.	The source's channel is connected to the sink's channel unchanged. If the source's channel signal has more bits, the higher bits are left unconnected. You are given a warning that channel information may be lost. An adapter provides a simulation error message and an error indication if the value of channel from the source is greater than the sink's maximum number of channels. In addition, the `valid` signal to the sink is deasserted so that the sink never sees data for channels that are out of range.

Error Adapter

The error adapter ensures that per-bit-error information provided by the source interface is correctly connected to the sink interface's input error signal. Matching error conditions processed by the source and sink are connected. If the source has an error condition that is not supported by the sink, the signal is left unconnected; the adapter provides a simulation error message and an error indication if this error is ever asserted. If the sink has an error condition that is not supported by the source, the sink's input is tied to zero.

Input Interface Parameters

Table 8-10 describes the available options for the Avalon-ST error adapter on the **Parameter Settings** page of the configuration page.

Table 8-10: Input Interface Parameters

Input Interface Parameters	
Error Signal Width (bits)	Type the width of the `error` signal. Valid values are 0–31 bits. Type `0` if the `error` signal is not used.
Error Signal Description	Type the description for each of the error bits. Separate the description fields by commas. For a connection to be made, the description of the error bits in the source and sink must match.

Output Interface Parameters

Table 8-11 describes the available options for the error adapter on the **Parameter Settings** page of the configuration page.

Table 8-11: Output Interface Parameters

Output Interface Parameters	
Error Signal Width (bits)	Type the width of the `error` signal. Valid values are 0–31 bits. Type `0` if you do not need to send error values. **Error Signal Width (bits)**

Output Interface Parameters	
Error Signal Description	Type the description for each of the error bits. Separate the description fields by commas. For a connection to be made, the description of the error bits in the source and sink must match. Refer to “C**Error Adapter” on page 9–27 for the adaptations that can be made when the bits do not match. **Error Signal Width (bits)**

Common to Input and Output Interfaces

Table 8-12 describes the available options for the Avalon-ST error adapter on the **Parameter Settings** page of the configuration wizard.

Table 8-12: Common to Input and Output Interfaces

Common to Input and Output Interfaces	
Support Backpressure with the ready signal	Turn on this option to add the backpressure functionality to the interface.
Ready Latency	When the `ready` signal is used, the value for `ready_latency` indicates the number of cycles between when the ready signal is asserted and when valid data is driven.
Channel Signal Width (bits)	Type the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is eight bits. Set to 0 if channels are not used.
Max Channel	Type the maximum number of channels that the interface supports. Valid values are 0–255.
Data Bits Per Symbol	Type the number of bits per symbol.
Data Symbols Per Beat	Type the number of symbols per active transfer.
Include Packet Support	Turn this option on if the interfaces supports a packet protocol, including the `startofpacket`, `endofpacket` and `empty` signals.
Include Empty Signal	Turn this option on if the cycle that includes the `endofpacket` signal can include empty symbols. This signal is not necessary if the number of symbols per beat is 1.

Interrupt Interfaces

Using individual requests, the interrupt logic can process up to 32 IRQ inputs connected to each interrupt receiver. With this logic, the interrupt sender connected to interrupt `receiver_0` is the highest priority with sequential receivers being successively lower priority. You can redefine the priority of interrupt senders by instantiating the IRQ mapper component. For more information refer to *IRQ Mapper*.

You can define the interrupt sender interface as asynchronous with no associated clock or reset interfaces. You can also define the interrupt receiver interface as asynchronous with no associated clock or reset interfaces. As a result, the receiver does its own synchronization internally. Qsys does not insert interrupt synchronizers for such receivers.

For clock crossing adaption on interrupts, Qsys inserts a synchronizer, which is clocked with the interrupt end point interface clock when the corresponding starting point interrupt interface has no clock or a different clock (than the end point). Qsys inserts the adapter if there is any kind of mismatch between the start and end points. Qsys does not insert the adapter if the interrupt receiver does not have an associated clock.

Related Information

IRQ Mapper on page 8-31

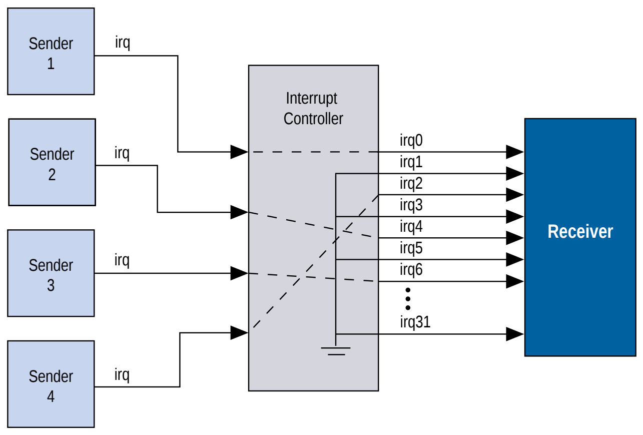
Individual Requests IRQ Scheme

In the individual requests IRQ scheme, Qsys interconnect passes IRQs directly from the sender to the receiver, without making assumptions about IRQ priority. In the event that multiple senders assert their IRQs simultaneously, the receiver logic determines which IRQ has highest priority, and then responds appropriately.

Using individual requests, the interrupt controller can process up to 32 IRQ inputs. The interrupt controller generates a 32-bit signal `irq[31:0]` to the receiver, and maps slave IRQ signals to the bits of `irq[31:0]`. Any unassigned bits of `irq[31:0]` are disabled.

Figure 8-26 shows an example of the interrupt controller mapping the IRQs on four senders to `irq[31:0]` on a receiver.

Figure 8-26: IRQ Mapping Using Software Priority



Assigning IRQs in Qsys

You assign IRQ connections on the **System Contents** tab of Qsys. After adding all components to the system, you connect interrupt senders and receivers. You can use the **IRQ** column to specify an IRQ number with respect to each receiver, or to specify a receiver's IRQ as unconnected. Qsys uses the following three components to implement interrupt handling: IRQ Bridge, IRQ Mapper, and IRQ Clock Crosser.

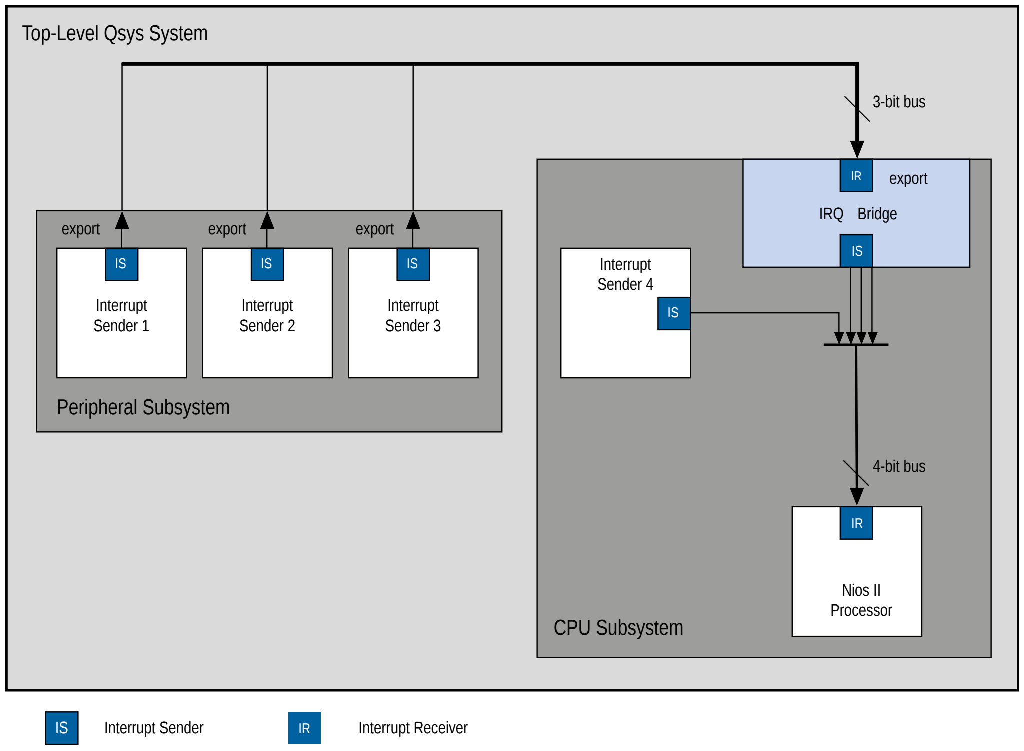
IRQ Bridge

The IRQ Bridge allows you to route interrupt wires between Qsys subsystems. These interrupts are routed to the IRQ receiver bridge in the CPU Subsystem.

Note: Nios II BSP tools do not fully support the IRQ Bridge. Interrupts connected via an IRQ Bridge will not appear in the generated **system.h** file.

In Figure 8-27, the Peripheral Subsystem has three interrupt senders that are exported to the top level of the subsystem.

Figure 8-27: Qsys IRQ Bridge Application



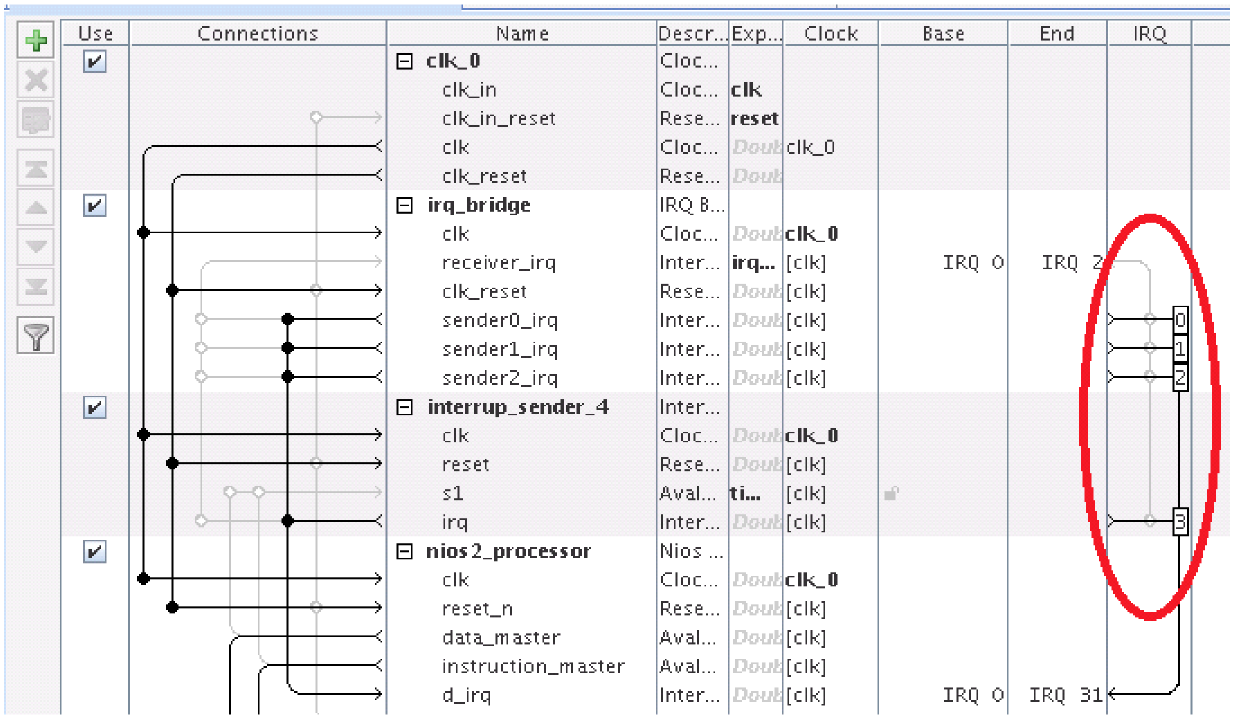
IRQ Mapper

Qsys inserts the IRQ Mapper automatically during generation. The IRQ Mapper converts individual interrupt wires to a bus, and then maps the appropriate IRQ priority number onto the bus.

By default, the interrupt sender connected to the `receiver0` interface of the IRQ mapper is the highest priority, and sequential receivers are successively lower priority. You can modify the interrupt priority of each IRQ wire by modifying the IRQ priority number in Qsys under the **IRQ** column. The modified priority is reflected in the **IRQ_MAP** parameter for the auto-inserted IRQ Mapper.

Figure 8-28 shows the **IRQ** column in Qsys and the default interrupt priority allocated for the CPU subsystem shown in *IRQ Bridge*.

Figure 8-28: IRQ Column in Qsys



Use	Connections	Name	Descr...	Exp...	Clock	Base	End	IRQ
✓		⊟ clk_0	Cloc...					
		clk_in	Cloc...	clk				
		clk_in_reset	Rese...	reset				
		clk	Cloc...	Doub	clk_0			
		clk_reset	Rese...	Doub				
✓		⊟ irq_bridge	IRQ B...					
		clk	Cloc...	Doub	clk_0			
		receiver_irq	Inter...	irq...	[clk]	IRQ 0	IRQ 2	
		clk_reset	Rese...	Doub	[clk]			
		sender0_irq	Inter...	Doub	[clk]			0
		sender1_irq	Inter...	Doub	[clk]			1
		sender2_irq	Inter...	Doub	[clk]			2
✓		⊟ interrup_sender_4	Inter...					
		clk	Cloc...	Doub	clk_0			
		reset	Rese...	Doub	[clk]			
		s1	Aval...	ti...	[clk]			
		irq	Inter...	Doub	[clk]			3
✓		⊟ nios2_processor	Nios ...					
		clk	Cloc...	Doub	clk_0			
		reset_n	Rese...	Doub	[clk]			
		data_master	Aval...	Doub	[clk]			
		instruction_master	Aval...	Doub	[clk]			
		d_irq	Inter...	Doub	[clk]	IRQ 0	IRQ 31	

IRQ Clock Crosser

The IRQ Clock Crosser synchronizes interrupt senders and receivers that are in different clock domains. To use this component, connect the clocks for both the interrupt sender and receiver, and for both the interrupt sender and receiver interfaces. Qsys automatically inserts this component when it is required.

Clock Interfaces

Clock interfaces define the clocks used by a component. Components can have clock inputs, clock outputs, or both. You can use the **Clock Settings** tab to define external clock sources, for example an oscillator on your board.

The **Clock Source** parameters allows you to set the following options:

- **Clock frequency**—The frequency of the output clock from this clock source.
- **Clock frequency is known**— When turned on, the clock frequency is known. When turned off, the frequency is set from outside the system.

  Note: If turned off, system generation may fail because the components do not receive the necessary clock information. For best results, turn this option on before system generation.

- **Reset synchronous edges**
  - **None**—The reset is asserted and deasserted asynchronously. You can use this setting if you have internal synchronization circuitry that matches the reset required for the IP in the system.
  - **Both**—The reset is asserted and deasserted synchronously.
  - **Deassert**—The reset is deasserted synchronously and asserted asynchronously.

For more information about synchronous design practices, refer to *Recommended Design Practices*

Related Information

Recommended Design Practices

(High Speed Serial Interface) HSSI Clock Interfaces

You can use HSSI Serial Clock and HSSI Bonded Clock interfaces in Qsys to enable high speed serial connectivity between clocks that are used by certain IP protocols.

HSSI Serial Clock Interface

You can connect the HSSI Serial Clock interface with only similar type of interfaces, for example, you can connect a HSSI Serial Clock Source interface to a HSSI Serial Clock Sink interface.

HSSI Serial Clock Source

The HSSI Serial Clock interface includes a source in the **Start** direction.

You can instantiate the HSSI Serial Clock Source interface in the **_hw.tcl** file as:

```
add_interface <name> hssi_serial_clock start
```

You can connect the HSSI Serial Clock Source to multiple HSSI Serial Clock Sinks because the HSSI Serial Clock Source supports multiple fan-outs. This Interface has a single **clk** port role limited to a 1 bit width, and a **clockRate** parameter, which is the frequency of the clock driven by the HSSI Serial Clock Source interface.

An unconnected and unexported HSSI Serial Source is valid and does not generate error messages.

Table 8-13: HSSI Serial Clock Source Port Roles

Name	Direction	Width	Description
clk	Output	1 bit	A single bit wide port role, which provides synchronization for internal logic.

Table 8-14: HSSI Serial Clock Source Parameters

Name	Type	Default	Derived	Description
clockRate	long	0	No	The frequency of the clock driven bythe HSSI Serial Clock Source interface.

HSSI Serial Clock Sink

The HSSI Serial Clock interface includes a sink in the **End** direction.

You can instantiate the HSSI Serial Clock Sink interface in the **_hw.tcl** file as:

```
add_interface <name> hssi_serial_clock end
```

You can connect the HSSI Serial Clock Sink interface to a single HSSI Serial Clock Source interface; you cannot connect it to multiple sources. This Interface has a single **clk** port role limited to a 1 bit width, and a **clockRate** parameter, which is the frequency of the clock driven by the HSSI Serial Clock Source interface.

An unconnected and unexported HSSI Serial Sink is invalid and generates error messages.

Table 8-15: HSSI Serial Clock Sink Port Roles

Name	Direction	Width	Description
clk	Output	1	A single bit wide port role, which provides synchronization for internal logic

Table 8-16: HSSI Serial Clock Sink Parameters

Name	Type	Default	Derived	Description
clockRate	long	0	No	The frequency of the clock driven bythe HSSI Serial Clock Source interface. When you specify a **clockRate** greater than 0, then this interface can be driven only at that rate.

HSSI Serial Clock Connection

The HSSI Serial Clock Connection defines a connection between a HSSI Serial Clock Source connection point, and a HSSI Serial Clock Sink connection point.

A valid HSSI Serial Clock Connection exists when all of the following criteria are satisfied. If the following criteria are not satisfied, Qsys generates error messages and the connection is prohibited.

- The starting connection point is an HSSI Serial Clock Source with a single port role **clk** and maximum 1 bit in width. The direction of the starting port is **Output**.
- The ending connection point is an HSSI Serial Clock Sink with a single port role **clk**, and maximum 1 bit in width. The direction of the ending port is **Input**.
- If the parameter, **clockRate** of the HSSI Serial Clock Sink is greater than 0, the connection is only valid if the **clockRate** of the HSSI Serial Clock Source is the same as the **clockRate** of the HSSI Serial Clock Sink.

HSSI Serial Clock Example

Example 8-1 shows connections that you can make to declare the HSSI Serial Clock interfaces in the **_hw.tcl**.

Example 8-1: HSSI Serial Clock Interface Example

```
package require -exact qsys 13.1

set_module_property name hssi_serial_component
set_module_property ELABORATION_CALLBACK elaborate

add_fileset QUARTUS_SYNTH QUARTUS_SYNTH generate
add_fileset SIM_VERILOG SIM_VERILOG generate
add_fileset SIM_VHDL SIM_VHDL generate

set_fileset_property QUARTUS_SYNTH TOP_LEVEL \
"hssi_serial_component"

set_fileset_property SIM_VERILOG TOP_LEVEL "hssi_serial_component"
set_fileset_property SIM_VHDL TOP_LEVEL "hssi_serial_component"

proc elaborate {} {
 # declaring HSSI Serial Clock Source
 add_interface my_clock_start hssi_serial_clock start
 set_interface_property my_clock_start  ENABLED true

 add_interface_port my_clock_start  hssi_serial_clock_port_out \

 clk Output 1

 # declaring HSSI Serial Clock Sink
 add_interface my_clock_end hssi_serial_clock end
 set_interface_property my_clock_end  ENABLED true

 add_interface_port my_clock_end  hssi_serial_clock_port_in clk
\
 Input 1
}

proc generate { output_name } {

 add_fileset_file hssi_serial_component.v VERILOG PATH \
 "hssi_serial_component.v"
}
```

If you use the components in Example 8-1 in a hierarchy, for example, instantiated in a composed component, you can declare the connections as shown in Example 8-2.

Example 8-2: HSSI Serial Clock Instantiated in a Composed Component

```
add_instance myinst1 hssi_serial_component
add_instance myinst2 hssi_serial_component
# add connection from source of myinst1 to sink of myinst2

add_connection myinst1.my_clock_start myinst2.my_clock_end \
hssi_serial_clock

# adding connection from source of myinst2 to sink of myinst1

add_connection myinst2.my_clock_start myinst2.my_clock_end \
hssi_serial_clock
```

HSSI Bonded Clock Interface

You can connect the HSSI Bonded Clock interface with only similar type of Interfaces, for example, you can connect a HSSI Bonded Clock Source interface to a HSSI Bonded Clock Sink interface.

HSSI Bonded Clock Source

The HSSI Bonded Clock interface includes a source in the **Start** direction.

You can instantiate the HSSI Bonded Clock Source interface in the **_hw.tcl** file as:

```
add_interface <name> hssi_bonded_clock start
```

You can connect the HSSI Bonded Clock Source to multiple HSSI Bonded Clock Sinks because the HSSI Serial Clock Source supports multiple fanouts. This Interface has a single **clk** port role limited to a width range of 1 to 1024 bits. The HSSI Bonded Clock Source interface has two parameters: **clockRate** and **serialzationFactor**. **clockRate** is the frequency of the clock driven by the HSSI Bonded Clock Source interface, and the **serializationFactor** is the parallel data width that operates the HSSI TX serializer. The serialization factor determines the required frequency and phases of the individual clocks within the HSSI Bonded Clock interface

An unconnected and unexported HSSI Bonded Source is valid and does not generate error messages.

Table 8-17: HSSI Bonded Clock Source Port Roles

Name	Direction	Width	Description
clk	Output	1 to 24 bits	A multiple bit wide port role which provides synchronization for internal logic.

Table 8-18: HSSI Bonded Clock Source Parameters

Name	Type	Default	Derived	Description
clockRate	long	0	No	The frequency of the clock driven bythe HSSI Serial Clock Source interface.
serializationFactor	long	0	No	The serialization factor is the parallel data width that operates the HSSI TX serializer. The serialization factor determines the necessary frequency and phases of the individual clocks within the HSSI Bonded Clock interface.

HSSI Bonded Clock Sink

The HSSI Bonded Clock interface includes a sink in the **End** direction.

You can instantiate the HSSI Bonded Clock Sink interface in the **_hw.tcl** file as:

```
add_interface <name> hssi_bonded_clock end
```

You can connect the HSSI Bonded Clock Sink interface to a single HSSI Bonded Clock Source interface; you cannot connect it to multiple sources. This Interface has a single **clk** port role limited to a width range of 1 to 1024 bits. The HSSI Bonded Clock Source interface has two parameters: **clockRate** and **serialzationFactor**. **clockRate** is the frequency of the clock driven by the HSSI Bonded Clock Source interface, and the serialization factor is the parallel data width that operates the HSSI TX serializer. The serialization factor determines the required frequency and phases of the individual clocks within the HSSI Bonded Clock interface

An unconnected and unexported HSSI Bonded Sink is invalid and generates error messages.

Table 8-19: HSSI Bonded Clock Source Port Roles

Name	Direction	Width	Description
clk	Output	1 to 24 bits	A multiple bit wide port role which provides synchroniza-tion for internal logic.

Table 8-20: HSSI Bonded Clock Source Parameters

Name	Type	Default	Derived	Description
clockRate	long	0	No	The frequency of the clock driven bythe HSSI Serial Clock Source interface.

Name	Type	Default	Derived	Description
serializationFactor	long	0	No	The serialization factor is the parallel data width that operates the HSSI TX serializer. The serialization factor determines the necessary frequency and phases of the individual clocks within the HSSI Bonded Clock interface.

HSSI Bonded Clock Connection

The HSSI Bonded Clock Connection defines a connection between a HSSI Bonded Clock Source connection point, and a HSSI Bonded Clock Sink connection point.

A valid HSSI Bonded Clock Connection exists when all of the following criteria are satisfied. If the following criteria are not satisfied, Qsys generates error messages and the connection is prohibited.

- The starting connection point is an HSSI Bonded Clock Source with a single port role **clk** with a width range of 1 to 24 bits. The direction of the starting port is **Output**.
- The ending connection point is an HSSI Bonded Clock Sink with a single port role **clk** with a width range of 1 to 24 bits. The direction of the ending port is **Input**.
- The width of the starting connection point **clk** must be the same as the width of the ending connection point.
- If the parameter, **clockRate** of the HSSI Bonded Clock Sink greater than 0, then the connection is only valid if the **clockRate** of the HSSI Bonded Clock Source is same as the **clockRate** of the HSSI Bonded Clock Sink.
- If the parameter, **serializationFactor** of the HSSI Bonded Clock Sink is greater than 0, Qsys generates a warning if the **serializationFactor** of HSSI Bonded Clock Source is not same as the **serializationFactor** of the HSSI Bonded Clock Sink.

HSSI Bonded Clock Example

Example 8-3 shows connections that you can make to declare the HSSI Bonded Clock interfaces in the **_hw.tcl** file.

Example 8-3: HSSI Bonded Clock Interface Example

```
package require -exact qsys 13.1

set_module_property name hssi_bonded_component
set_module_property ELABORATION_CALLBACK elaborate

add_fileset synthesis QUARTUS_SYNTH generate
add_fileset verilog_simulation SIM_VERILOG generate

set_fileset_property synthesis TOP_LEVEL "hssi_bonded_component"
set_fileset_property verilog_simulation TOP_LEVEL \
"hssi_bonded_component"

proc elaborate {} {
 add_interface my_clock_start hssi_bonded_clock start
 set_interface_property my_clock_start  ENABLED true

 add_interface_port my_clock_start  hssi_bonded_clock_port_out \

 clk Output 1024

 add_interface my_clock_end hssi_bonded_clock end
 set_interface_property my_clock_end  ENABLED true

 add_interface_port my_clock_end  hssi_bonded_clock_port_in \
 clk Input 1024
}

proc generate { output_name } {
 add_fileset_file hssi_bonded_component.v VERILOG PATH \
 "hssi_bonded_component.v"}
```

If you use the components in Example 8-3 in a hierarchy, for example, instantiated in a composed component, you can declare the connections as shown in Example 8-4.

Example 8-4: HSII Bonded Clock Instantiated in a Composed Component

```
add_instance myinst1 hssi_bonded_component
add_instance myinst2 hssi_bonded_component
# add connection from source of myinst1 to sink of myinst2

add_connection myinst1.my_clock_start myinst2.my_clock_end \
hssi_bonded_clock

# adding connection from source of myinst2 to sink of myinst1

add_connection myinst2.my_clock_start myinst2.my_clock_end \
hssi_bonded_clock
```

Reset Interfaces

Reset interfaces provide both soft and hard reset functionality. Soft reset logic typically re-initializes registers and memories without powering down the device. Hard reset logic initializes the device after power-on. You can define separate reset sources for each clock domain, a single reset source for all clocks, or any combination in between.

You can choose to create a single global reset domain by selecting **Create Global Reset Network** on the System menu. If your design requires more than one reset domain, you can implement your own reset logic and connectivity. The library includes a reset controller, reset sequencer, and a reset bridge to implement the reset functionality. You can also design your own reset logic.

Note: If you design your own reset circuitry you must carefully consider situations which might result in system lockup. For example, if an Avalon-MM slave is reset in the middle of a transaction, the Avalon-MM master might wait forever.

Single Global Reset Signal Implemented by Qsys

If you select **Create Global Reset Network** on the System menu, the Qsys interconnect creates a global reset bus. All of the reset requests are ORed together, synchronized to each clock domain, and fed to the reset inputs. The duration of the reset signal is at least one clock period.

The Qsys interconnect inserts the system-wide reset under the following conditions:

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

Reset Controller

Qsys automatically inserts a reset controller block if the input reset source does not have a reset request, but the connected reset sink requires a reset request.

The Reset Controller has the following parameters that you can specify to customize its behavior:

- **Number of inputs**— Indicates the number of individual reset interfaces the controller ORs to create a signal reset output.
- **Output reset synchronous edges**—Specifies the level of synchronization. You can select one the following options:
  - **None**—The reset is asserted and deasserted asynchronously. You can use this setting if you have designed internal synchronization circuitry that matches the reset style required for the IP in the system.
  - **Both**—The reset is asserted and deasserted synchronously.
  - **Deassert**—The reset is deasserted synchronously and asserted asynchronously.
- **Synchronization depth**—Specifies the number of register stages the synchronizer uses to eliminate the propagation of metastable events.
- **Reset request**—Enables reset request generation, which is an early signal that is asserted before reset assertion. The reset request is used by blocks that require protection from asynchronous inputs, for example, M20K blocks.

Qsys automatically inserts reset synchronizers under the following conditions:

- More than one reset source is connected to a reset sink
- There is a mismatch between the reset source's synchronous edges and the reset sinks' synchronous edges

Reset Bridge

The Reset Bridge allows you to use a reset signal in two or more subsystems of your Qsys system. You can connect one reset source to local components, and export one or more to other subsystems, as required.

The Reset Bridge parameters are used to describe the incoming reset and include the following options:

- **Active low reset**—When turned on, reset is asserted low.
- **Synchronous edges**—Specifies the level of synchronization and includes the following options:
  - **None**—The reset is asserted and deasserted asynchronously. Use this setting if you have internal synchronization circuitry.
  - **Both**—The reset is asserted and deasserted synchronously.
  - **Deassert**—The reset is deasserted synchronously, and asserted asynchronously.
- **Number of reset outputs**—The number of reset interfaces that are exported.

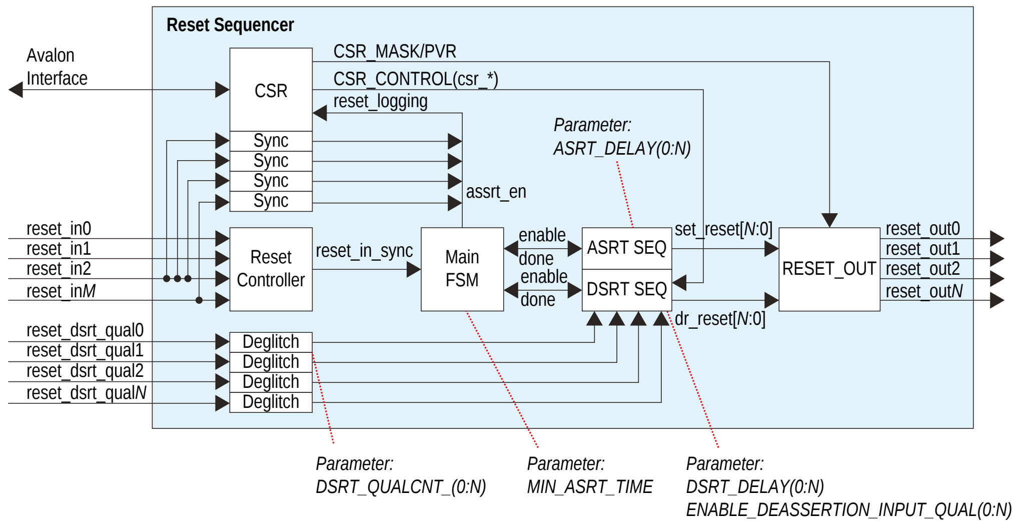
Note: Qsys supports multiple reset sink connections to a single reset source interface. However, there are situations in composed systems where an internally generated reset must be exported from the composed system in addition to being used to connect internal components. In this situation, you must declare one reset output interface as an export, and use another reset output to connect internal components.

Reset Sequencer

The reset sequencer allows you to control the assertion and deassertion sequence for Qsys system resets.

You can connect multiple reset sources to the reset sequencer, and then connect the output of the reset sequencer to components in the system. Figure 8-29 shows the elements and flow of the reset sequencer.

Figure 8-29: Reset Sequencer Top-Level Block Diagram



- *Reset Controller*—Reused reset controller block. It synchronizes the reset inputs into one and feed into the main FSM of the sequencer block.
- *Sync*—Synchronization block (double flip-flop).
- *Deglitch*—Deglitch block. This block waits for a signal to be at a level for *X* clocks before propagating the input to the output.
- *CSR*—This block contains the CSR Avalon interface and related CSR register and control block in the sequencer.
- *Main FSM* —Main sequencer. This block determines when assertion/deassertion and assertion hold timing occurs.
- *[A/D]SRT SEQ*—Generic sequencer block that sequences out assertion/deassertion of reset from 0:N. The block has multiple counters that saturate upon reaching count.
- *RESET_OUT*—Controls the end output via:
  - Set/clear from the ASRT_SEQ/DSRT_SEQ.
  - Masking/forcing from CSR controls.
  - Remap of numbering (parameterization).

Reset Sequencer Parameters

The following parameters are available for customizing the Reset Sequencer:

Parameter	Description
Number of reset outputs	Sets the number of output resets to be sequenced, which is the number of output reset signals defined in the component with a range of 2 to 10.
Number of reset inputs	Sets the number of input reset signals to be sequenced, which is the number of input reset signals defined in the component with a range of 1 to 10.
Minimum reset assertion time	Specifies the minimum assertion cycles between the assertion of the last sequenced reset, and the de-assertion of the first sequenced reset. The range is 0 to 1023.
Enable Reset Sequencer CSR	Enables CSR functionality of the Reset Sequencer through an Avalon interface.
reset_out#	Lists the reset output signals. Set the parameters in the other parameter table columns for each reset signal listed.
ASRT Seq#	Determines the order of reset assertion. Enter the values 1, 2, 3, etc. to specify the required non-overlapping assertion order. This value determines the `ASRT_REMAP` value in the component HDL.
ASRT Cycle#	Number of cycles to wait before assertion of the reset. The value set here corresponds to the `ASRT_DELAY` value in the component HDL (as indicated **Figure 8-29**). The range is 0 to1023.
DSRT Seq#	Determines the reset order of reset de-assertion. Enter the values 1, 2, 3, etc .to specify the required non-overlapping de-assertion order. This value determines the `DSRT_REMAP` value in the component HDL.
DSRT Cycle#/Deglitch#	Number of cycles to wait before de-asserting or de-glitching the reset. If the **USE_DRST_QUAL** parameter is set to 0, specifies the number of cycles to wait before de-asserting the reset. If **USE_DSRT_QUAL** is set to1, specifies the number of cycles to deglitch the input `reset_dsrt_qual` signal. This value determines either the `DSRT_DELAY`, or the `DSRT_QUALCNT` value in the component HDL (as indicated **Figure 8-29**) depending on the **USE_DSRT_QUAL** parameter setting. The range is 0 to 1023.
USE_DSRT_QUAL	If you set **USE_DSRT_QUAL** to 1 , the de-assertion sequence waits for an external input signal (shown as `reset_dsrt_qualN` in **Figure 8-29**) for sequence qualification instead of waiting for a fixed delay count. To use a fixed delay count for de-assertion, set this parameter to 0.

Note: Below the parameter settings table, the Parameter Editor displays the expected Assertion Sequence and De-assertion Sequence based on the current settings in the table.

Reset Sequencing Timing Diagrams

Figure 8-30 and Figure 8-31 are examples of generated sequenced timing diagrams.

Figure 8-30: Basic Sequencing

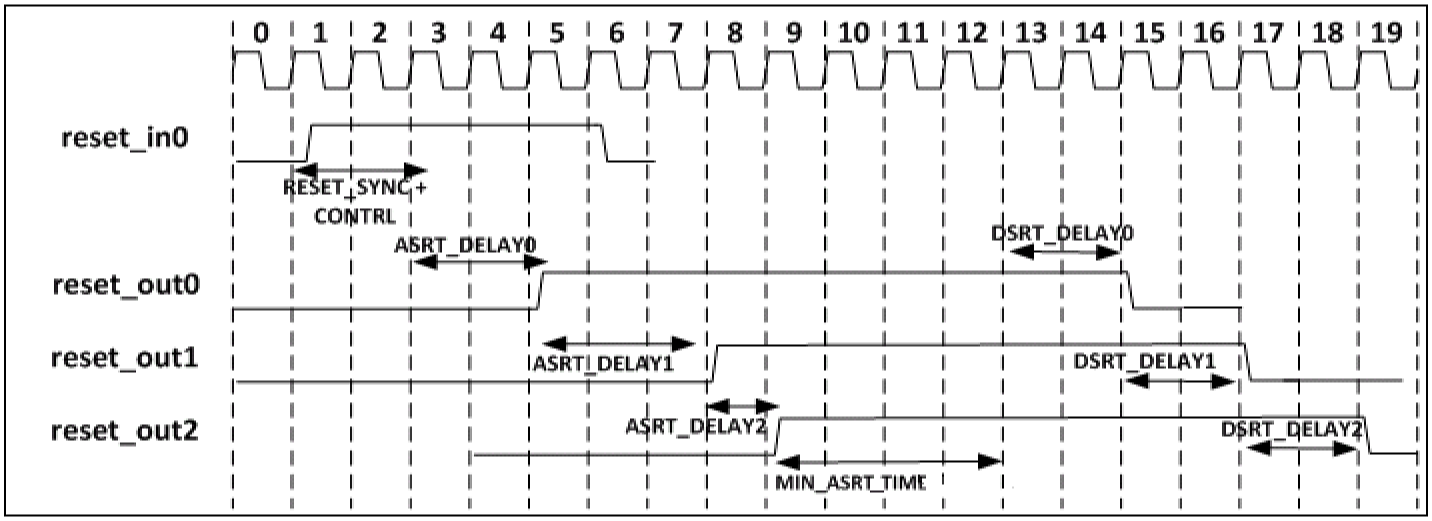
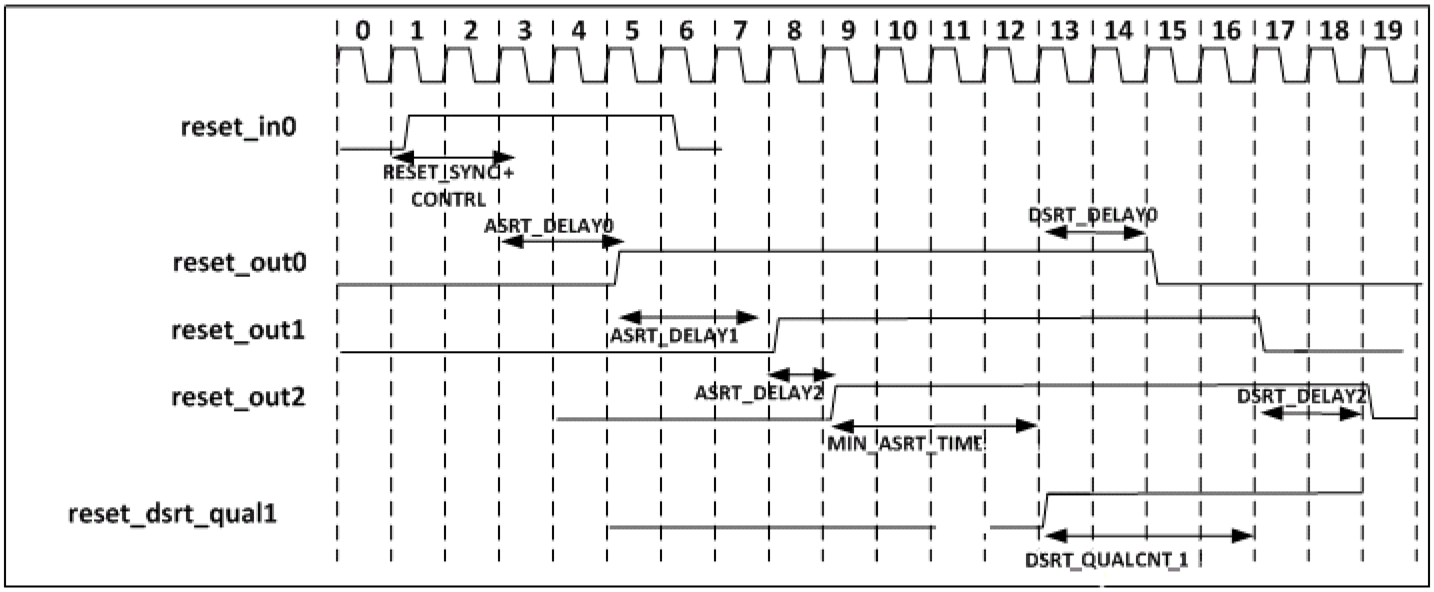


Figure 8-31: Sequencing with USE_DSRT_QUAL Set



Reset Sequencer CSR Registers

The CSR registers on the reset sequencer provide the following functionality:

- **Supports reset logging**
  - Ability to identify which reset is asserted.
  - Ability to determine whether any reset is currently active.
- **Supports software triggered resets**
  - Ability to generate reset by writing to the register.
  - Ability to disable assertion or de-assertion sequence.
- **Supports software sequenced reset**
  - Ability for the software to fully control the assertion/de-assertion sequence by writing to registers and stepping through the sequence.
- **Support reset override**
  - Ability to assert a particular component reset through software.

Reset Sequencer Status Register Offset 0x00

The **Status** register contains status bits that indicate the sources of resets that have caused a reset.

You can clear bits by writing 1 to the bit location. The Reset Sequencer ignores writes to bits with a value of 0. If the sequencer is reset (power-on-reset), all bits are cleared, except the power on reset bit.

Note: Refer to RW1C in Table 8-21.

Table 8-21: Values for the Status Register at Offset 0x00

Bit	Attribute	Default	Description
31	RO	0	**Reset Active**—Indicates that the sequencer is currently active in reset sequence (assertion or de-assertion).
30	RW1C	0	**Reset Asserted and waiting for SW to proceed:**—Set when there is an active reset assertion, and the next sequence is waiting for the software to proceed. Only valid when the **Enable SW sequenced reset entry** option is turned on.
29	RW1C	0	**Reset De-asserted and waiting for SW to proceed:**—Set when there is an active reset de-assertion, and the next sequence is waiting for the software to proceed. Only valid when the **Enable SW sequenced reset bring up** option is turned on.
28:26	RO	0	Reserved.
25:16	RW1C	0	**Reset de-assertion input qualification signal reset_dsrt_qual [9:0] status**—Indicates that the reset de-assertion's input signal qualification signal is set. This bit is set on the detection of assertion of the signal.
15:12	RO	0	Reserved.

Bit	Attribute	Default	Description
11	RW1C	0	**reset_in9 was triggered**—Indicates that `resetin9` triggered the reset. Cleared by software by writing 1 to this bit location.
10	RW1C	0	**reset_in8 was triggered**—Indicates that `reset_in8` triggered the reset. Cleared by software by writing a1 to this bit location.
9	RW1C	0	**reset_in7 was triggered**—Indicates that `reset_in7` triggered the reset. Cleared by software by writing 1 to this bit location.
8	RW1C	0	**reset_in6 was triggered**—Indicates that `reset_in6` triggered the reset. Cleared by software by writing 1 to this bit location.
7	RW1C	0	**reset_in5 was triggered**—Indicates that `reset_in5` triggered the reset. Cleared by software by writing 1 to this bit location.
6	RW1C	0	**reset_in4 was triggered**—Indicates that `reset_in4` triggered the reset. Cleared by software by writing 1 to this bit location.
5	RW1C	0	**reset_in3 was triggered**—IIndicates that `reset_in3` triggered the reset. Cleared by software by writing 1 to this bit location.
4	RW1C	0	**reset_in2 was triggered**—Indicates that `reset_in2` triggered the reset. Cleared by software by writing 1 to this bit location.
3	RW1C	0	**reset_in1 was triggered**—Indicates that `reset_in1` triggered the reset. Cleared by software by writing 1 to this bit location.
2	RW1C	0	**reset_in0 was triggered**—Indicates that reset_in0 triggered. Cleared by software by writing 1 to this bit location.
1	RW1C	0	**Software triggered reset**—Indicates that the software triggered reset is set by the software, and triggering a reset.
0	RW1C	0	**Power-On-Reset was triggered**—Asserted whenever the reset to the sequencer is triggered. This bit is NOT reset when sequencer is reset. Cleared by software by writing 1 to this bit location.

Reset Sequencer Interrupt Enable Register Offset 0x04

The Interrupt Enable register contains the interrupt enable bit that you can use to enable any event triggering the IRQ of the reset sequencer.

Table 8-22: Values for the Interrupt Enable Register at Offset 0x04

Bit	Attribute	Default	Description
31	RO	0	Reserved.
30	RW	0	**Interrupt on Reset Asserted and waiting for SW to proceed** enable. When set, the IRQ is set when the sequencer is waiting for the software to proceed in an assertion sequence.

Bit	Attribute	Default	Description
29	RW	0	**Interrupt on Reset De-asserted and waiting for SW to proceed** enable. When set, the IRQ is set when the sequencer is waiting for the software to proceed in a de-assertion sequence.
28:26	RO	0	Reserved.
25:16	RW	0	**Interrupt on Reset de-assertion input qualification signal reset_dsrt_qual_ [9:0] status**— When set, the IRQ is set when the `reset_dsrt_qual[9:0]` status bit (per bit enable) is set.
15:12	RO	0	Reserved.
11	RW	0	**Interrupt on reset_in9 Enable**—When set, the IRQ is set when the `reset_in9` trigger status bit is set.
10	RW	0	**Interrupt on reset_in8 Enable**—When set, the IRQ is set when the `reset_in8` trigger status bit is set.
9	RW	0	**Interrupt on reset_in7 Enable**—When set, the IRQ is set when the `reset_in7` trigger status bit is set.
8	RW	0	**Interrupt on reset_in6 Enable**—When set, the IRQ is set when the `reset_in6` trigger status bit is set.
7	RW	0	**Interrupt on reset_in5 Enable**—When set, the IRQ is set when the `reset_in5` trigger status bit is set.
6	RW	0	**Interrupt on reset_in4 Enable**—When set, the IRQ is set when the `reset_in4` trigger status bit is set.
5	RW	0	**Interrupt on reset_in3 Enable**—When set, the IRQ is set when the `reset_in3` trigger status bit is set.
4	RW	0	**Interrupt on reset_in2 Enable**—When set, the IRQ is set when the `reset_in2` trigger status bit is set.
3	RW	0	**Interrupt on reset_in1 Enable**—When set, the IRQ is set when the `reset_in1` trigger status bit is set.
2	RW	0	**Interrupt on reset_in0 Enable**—When set, the IRQ is set when the `reset_in0` trigger status bit is set.
1	RW	0	**Interrupt on Software triggered reset Enable**—When set, the IRQ is set when the software triggered reset status bit is set.
0	RW	0	**Interrupt on Power-On-Reset Enable**—When set, the IRQ is set when the power-on-reset status bit is set.

Reset Sequencer Control Register Offset 0x08

The Control register contains registers that you can use to control the reset sequencer.

Table 8-23: Values for the Control Register at Offset 0x08

Bit	Attribute	Default	Description
31:3	RO	0	Reserved.
2	RW	0	**Enable SW sequenced reset entry**—Enable a software sequenced reset entry sequence. Timer delays and input qualification are ignored, and only the software can sequence the entry.
1	RW	0	**Enable SW sequenced reset bring up**—Enable a software sequenced reset bring up sequence. Timer delays and input qualification are ignored, and only the software can sequence the bring up.
0	WO	0	**Initiate Reset Sequence**—Reset Sequencer writes this bit to 1 a single time in order to trigger the hardware sequenced warm reset. Reset Sequencer verifies that **Reset Active** is 0 before setting this bit, and always reads the value 0. To monitor this sequence, verify that **Reset Active** is asserted, and then subsequently de-asserted.

Reset Sequencer Software Sequenced Reset Entry Control Register Offset 0x0C

You can program the Reset Sequencer Software Sequenced Reset Entry Control register to control the reset entry sequence of the sequencer.

When the corresponding enable bit is set, the sequencer stops when the desired reset asserts, and then sets the **Reset Asserted and waiting for SW to proceed** bit. The Reset Sequencer proceeds only after the **Reset Asserted and waiting for SW to proceed** bit is cleared.

Table 8-24: Values for the Reset Sequencer Software Sequenced Reset Entry Controls Register at Offset 0x0C

Bit	Attribute	Default	Description
31:10	RO	0	Reserved.
9:0	RW	3FF	**Per-reset SW sequenced reset entry enable**—This is a per-bit enable for SW sequenced reset entry. If `bitN` of this register is set, the sequencer sets the `bit30` of the status register when a `resetN` is asserted. It then waits for the `bit30` of the status register to clear before proceeding with the sequence. By default, all bits are enabled (fully SW sequenced).

Reset Sequencer Software Sequenced Reset Bring Up Control Register Offset 0x10

You can program the Software Sequenced Reset Bring Up Control register to control the reset bring up sequence of the sequencer.

When the corresponding enable bit is set, the sequencer stops when the desired reset asserts, and then sets the **Reset De-asserted and waiting for SW to proceed** bit. The Reset Sequencer proceeds only after the **Reset De-asserted and waiting for SW to proceed** bit is cleared..

Table 8-25: Values for the Reset Sequencer Software Sequenced Bring Up Control Register at Offset 0x10

Bit	Attribute	Default	Description
31:10	RO	0	Reserved.
9:0	RW	3FF	**Per-reset SW sequenced reset entry enable**—This is a per-bit enable for SW sequenced reset bring up. If `bitN` of this register is set, the sequencer sets `bit29` of the status register when a `resetN` is asserted. It then waits for the `bit29` of the status register to clear before proceeding with the sequence. By default, all bits are enabled (fully SW sequenced).

Reset Sequencer Software Direct Controlled Resets Offset 0x14

You can write a bit to 1 to assert the `reset_outN` signal, and to 0 to de-assert the `reset_outN` signal.

Table 8-26: Values for the Software Direct Controlled Resets at Offset 0x14

Bit	Attribute	Default	Description
31:26	RO	0	Reserved.
25:16	WO	0	**Reset Overwrite Trigger Enable** —This is a per-bit control trigger bit for the overwrite value to take effect.
15:10	RO	0	Reserved.
9:0	WO	0	**reset_outN Reset Overwrite Value**—This is a per-bit control of the `reset_out` bit. The Reset Sequencer can use this to forcefully drive the reset to a specific value. A value of 1 sets the `reset_out`. A value of 0 clears the `reset_out`. A write to this register only takes effect if the corresponding trigger bit in this register is set.

Reset Sequencer Software Reset Masking Offset 0x18

You can write a bit to 1 to assert the `reset_outN` signal, and to 0 to de-assert the `reset_outN` signal.

Table 8-27: Values for the Reset Sequencer Software Reset Masking at Offset 0x18

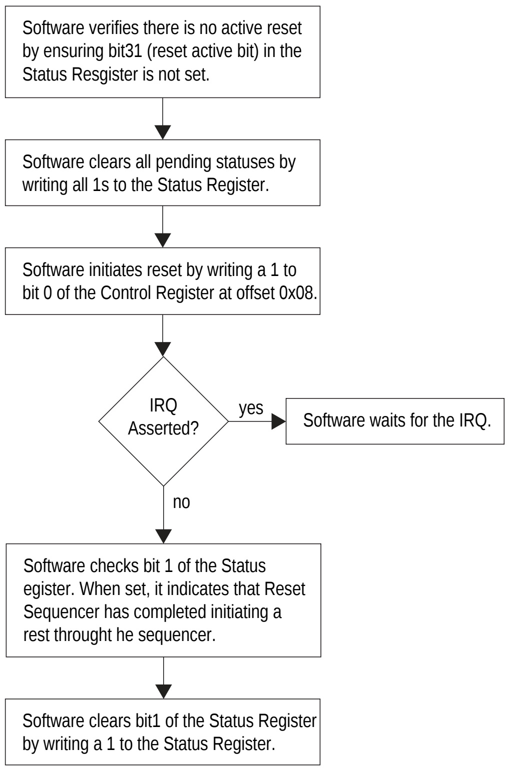
Bit	Attribute	Default	Description
31:10	RO	0	Reserved.
9:0	RW	0	**reset_outN "Reset Mask Enable"**—This is a per-bit control to mask the `reset_outN` bit. The Software Reset Masking masks the reset bit from being asserted during a reset assertion sequence. If the `reset_out` is already asserted, it does not de-assert the reset.

Reset Sequencer Software Flows

Reset Sequencer (Software-Triggered) Flow

The flow in **Figure 8-32** occurs for a Software Triggered Reset Flow:

Figure 8-32: Reset Sequencer (Software-Triggered) Flow


Software verifies there is no active reset by ensuring bit31 (reset active bit) in the Status Resgister is not set.
Software clears all pending statuses by writing all 1s to the Status Register.
Software initiates reset by writing a 1 to bit 0 of the Control Register at offset 0x08.
IRQ Asserted?
yes
Software waits for the IRQ.
no
Software checks bit 1 of the Status egister. When set, it indicates that Reset Sequencer has completed initiating a rest throught he sequencer.
Software clears bit1 of the Status Register by writing a 1 to the Status Register.

Reset Entry Flow

The following flow sequence occurs for a Reset Entry Flow:

- A reset is triggered either by the software, or when input resets to the Reset Sequencer are asserted.
- The IRQ is asserted, if the IRQ is enabled.
- Software reads the Status register to determine what reset was triggered.

Reset Bring-Up Flow

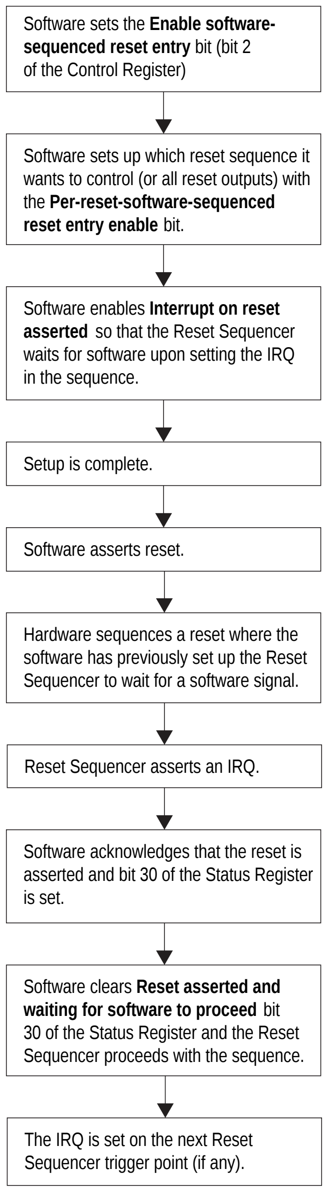
The following flow sequence occurs for a Reset Bring-Up Flow:

- When a reset source is de-asserted, or when the reset entry sequence has completed without any more pending resets asserted, the bring-up flow is initiated.
- The IRQ is asserted, if the IRQ is enabled.
- Software reads the Status register to determine what reset was triggered.

Reset Entry (Software-Sequenced) Flow

The flow in Figure 8-33 occurs for a Reset Entry (SW Sequenced) Flow:

Figure 8-33: Reset Entry (Software-Sequenced) Flow


Software sets the **Enable software-sequenced reset entry** bit (bit 2 of the Control Register)
Software sets up which reset sequence it wants to control (or all reset outputs) with the **Per-reset-software-sequenced reset entry enable** bit.
Software enables **Interrupt on reset asserted** so that the Reset Sequencer waits for software upon setting the IRQ in the sequence.
Setup is complete.
Software asserts reset.
Hardware sequences a reset where the software has previously set up the Reset Sequencer to wait for a software signal.
Reset Sequencer asserts an IRQ.
Software acknowledges that the reset is asserted and bit 30 of the Status Register is set.
Software clears **Reset asserted and waiting for software to proceed** bit 30 of the Status Register and the Reset Sequencer proceeds with the sequence.
The IRQ is set on the next Reset Sequencer trigger point (if any).

Reset Bring-Up (Software-Sequenced) Flow

The sequence and flow is similar to the **Reset Entry (SW Sequenced)** flow, though, this flow uses the **reset bring-up** registers/bits in place of the **reset entry** registers/bits.

Related Information

Reset Entry (Software-Sequenced) Flow on page 8-50

Conduits

You can use the conduit interface type for interfaces that do not fit any of the other interface types, and to group any arbitrary collection of signals. Like other interface types, you can export or connect conduit interfaces. The *PCI Express-to-Ethernet* example in *Creating a System with Qsys* is an example of using a conduit interface for export. You can declare an associated clock interface for conduit interfaces in the same way as memory-mapped interfaces with the `associatedClock`.

To connect two conduit interfaces inside Qsys, the following conditions must be met:

- The interfaces must match exactly with the same signal roles and widths.
- The interfaces must be the opposite directions.
- Clocked conduit connections must have matching `associatedClocks` on each of their endpoint interfaces.

Note: To connect a conduit output to more than one input conduit interface, you can create a custom component. The custom component could have one input that connects to two outputs, and you can use this component between other conduits that you want to connect. For information about the Avalon Conduit interface, refer to the *Avalon Interface Specifications*

Related Information

Avalon Interface Specifications

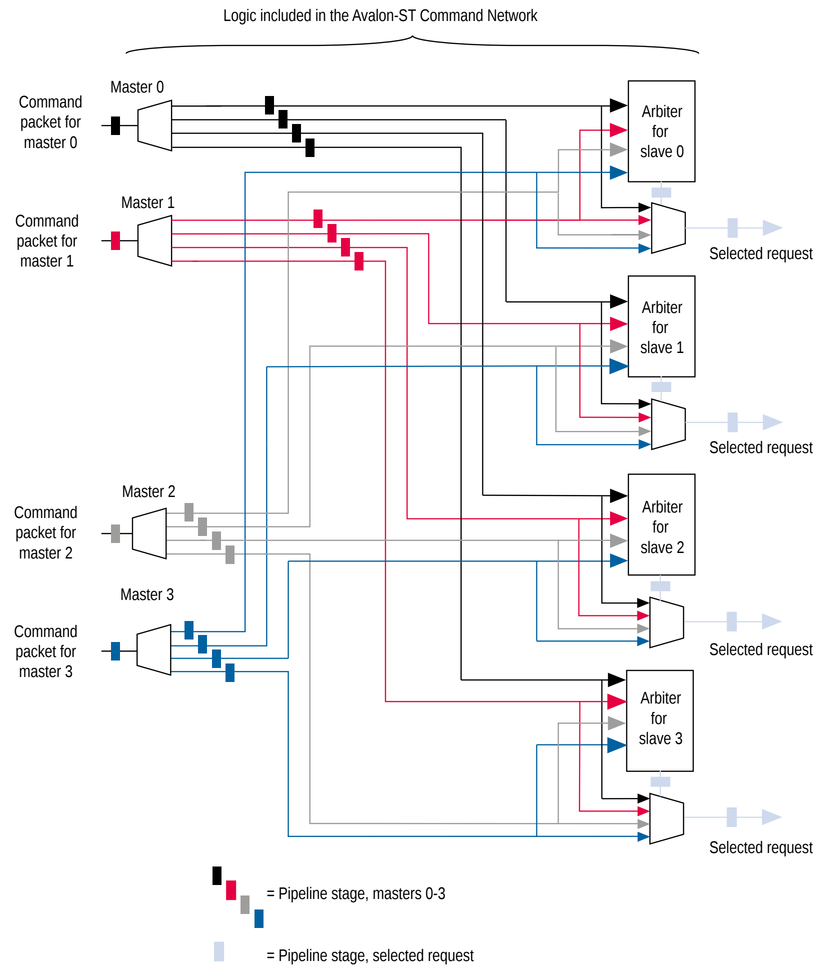
Creating a System with Qsys

Interconnect Pipelining

If you set the **Limit interconnect pipeline stages to** parameter to a value greater than 0 on the **Project Settings** tab, Qsys automatically inserts Avalon-ST pipeline stages when you generate your design. The pipeline stages increase the f_{MAX} of your design by reducing the combinational logic depth. The cost is additional latency and logic.

The insertion of pipeline stages depends upon the existence of certain interconnect components. For example, in a single-slave system, no multiplexer exists; therefore multiplexer pipelining does not occur. In an extreme case, of a single-master to single-slave system, no pipelining occurs, regardless of the value of **Limit interconnect pipeline stages to**. Figure 8-34 shows the placement of up to four potential pipeline stages inserted by Qsys before the input to the demultiplexer, at the output of the multiplexer, between the arbiter and the multiplexer, and at the outputs of the demultiplexer.

Figure 8-34: Pipeline Placement in Arbitration Logic


Logic included in the Avalon-ST Command Network
Master 0
Command packet for master 0
Master 1
Command packet for master 1
Master 2
Command packet for master 2
Master 3
Command packet for master 3
Arbiter for slave 0
Arbiter for slave 1
Arbiter for slave 2
Arbiter for slave 3
Selected request
Selected request
Selected request
Selected request
= Pipeline stage, masters 0-3
= Pipeline stage, selected request

Note: For more information about manually inserting and removing pipelines from your system, refer to *Creating a System With Qsys*. Refer to *Optimizing Qsys System Performance* for more information about pipelined Avalon-MM Interfaces.

Related Information

Creating a System With Qsys

Manually Controlling Pipelining in the Qsys Interconnect

The **Memory-Mapped Interconnect** tab allows you to manipulate pipleline connections in the Qsys interconnect. You access the **Memory-Mapped Interconnect** tab by clicking **Show System With Qsys Interconnect** command on the System menu.

Note: To increase interconnect frequency, you should first try increasing the value of the **Limit interconnect pipeline stages to** option on the **Project Settings** tab. You should only consider manually pipelining the interconnect if changes to this option do not improve frequency, and you have tried all other options to achieve timing closure, including the use of a bridge. Manually pipelining the interconnect should only be applied to complete systems.

1. In the **Project Settings** tab, first try increasing the value of the **Limit interconnect pipeline stages to** option until it no longer gives significant improvements in frequency, or until it causes unacceptable effects on other parts of the system.
2. In the Quartus II software, compile your design and run timing analysis.
3. Identify the critical path through the interconnect and determine the approximate mid-point. The following is an example of a timing report where the critical path is located in the interconnect.

```
2.800 0.000 cpu_instruction_master|out_shifter[63]|q
3.004 0.204 mm_domain_0|addr_router_001|Equal5~0|datac
3.246 0.242 mm_domain_0|addr_router_001|Equal5~0|combout
3.346 0.100 mm_domain_0|addr_router_001|Equal5~1|dataa
3.685 0.339 mm_domain_0|addr_router_001|Equal5~1|combout
4.153 0.468 mm_domain_0|addr_router_001|src_channel[5]~0|datad
4.373 0.220 mm_domain_0|addr_router_001|src_channel[5]~0|combout
```

4. **System** > **Show System With Qsys Interconnect**.
5. In the **Memory-Mapped Interconnect** tab, select the interconnect module that has the critical path. You can determine the name of the interconnect module from the hierarchical node names in the timing report.
6. Click **Show Pipelinable Locations**. Qsys display all pipelinable locations in the interconnect. You can right-click a pipelinable location to open a menu that allows you to insert or remove a pipeline stage.
7. Find the pipelinable location that is closest to the mid-point of the critical path. The names of blocks in the memory-mapped interconnect view correspond to the module instance names in the timing report.
8. Right-click the location where you want to insert a pipeline stage, and then click **Insert Pipeline**.
9. Regenerate the Qsys system, recompile the design, and then rerun timing analysis. If necessary, repeat the manual pipelining process again until timing requirements are met.

Manual pipelining has the following limitations:

- If you make changes to your original system's connectivity after manually pipelining an interconnect, your inserted pipelines may become invalid. Warning messages are displayed at generation time if invalid pipeline stages are detected. You can remove invalid pipeline stages with the **Remove Stale Pipelines** option button in the **Memory-Mapped Interconnect** tab. Altera recommends not making changes to the system's connectivity after manual pipeline insertion.
- Review manually-inserted pipelines when upgrading to newer versions of Qsys. Manually-inserted pipelines in one version of Qsys might not be valid in a future version.

Related Information

- Qsys System Design Components

AMBA AXI3 (version 1.0) Specification Support

Qsys allows connections between AXI3 components, AXI4 components, and Avalon memory-mapped interface types with some unique or exceptional support.

Refer to the *AMBA Protocol Specifications* for AXI3 on the ARM website for more information.

Related Information

AMBA Protocol Specifications

AXI3 Channels

Qsys 13.1 has the following support and restrictions for AXI3 channels.

Read and Write Address Channels

All signals are allowed with some limitations.

The following limitations are present in Qsys 13.1:

- Supports 64-bit addressing.
- ID width limited to 18-bits.
- HPS-FPGA master interface has a 12-bit ID.

Write Data, Write Response, and Read Data Channels

All signals are allowed with some limitations.

The following limitations are present in Qsys 13.1:

- Data widths limited to a maximum of 1024-bits
- Limited to a fixed byte width of 8-bits

Low Power Channel

Low power extensions are not supported in Qsys, version 13.1.

Cache Support

AWCACHE and ARCACHE are passed to an AXI slave unmodified.

Bufferable

Qsys interconnect treats AXI transactions as non-bufferable. All responses must come from the terminal slave.

When connecting to Avalon-MM slaves, since they do not have write responses, the following exceptions apply:

- For Avalon-MM slaves, the write response are generated by the slave agent once the write transaction is accepted by the slave. The following limitation exists for an Avalon bridge:
- For an Avalon bridge, the response is generated before the write reaches the endpoint; users must be aware of this limitation and avoid multiple paths past the bridge to any endpoint slave, or only perform bufferable transactions to an Avalon bridge.

Cacheable (Modifiable)

Qsys interconnect acknowledges the cacheable (modifiable) attribute of AXI transactions.

It does not change the address, burst length, or burst size of non-modifiable transactions, with the following exceptions:

- A wide transaction to a narrow slave is treated as modifiable because the size needs to be reduced.
- AXI read and write transactions might be treated as modifiable when the destination is an Avalon slave. The AXI transaction might be split into multiple Avalon transactions if the slave is unable to accept the transaction, which might occur because of burst lengths, narrow sizes, or burst types.

All other bits, for example, read allocate or write allocate, are ignored because the interconnect does not perform caching. By default, transactions issued by Avalon masters are treated as non-bufferable and non-cacheable, with the allocate bits tied low. Qsys provides compile-time options to control the cache behavior of Avalon transactions on a per-master basis.

Security Support

TrustZone refers to the security extension of the ARM architecture, which includes the concept of "secure" and "non-secure" transactions, and a protocol for processing between the designations. TrustZone security support is a part of the Qsys 13.1 interconnect.

The interconnect passes the `AWPROT` and `ARPROT` signals to the endpoint slave without modification. It does not use or modify the `PROT` bits.

Refer to *Creating a System with Qsys* for more information about secure systems and the TrustZone feature.

Related Information

Creating a System with Qsys

Atomic Accesses

Exclusive accesses are supported for AXI slaves by passing the lock, transaction ID, and response signals from master to slave, with the limitation that slaves that do not reorder responses. Avalon slaves do not support exclusive accesses, and always return `OKAY` as a response. Locked accesses are also not supported.

Response Signaling

Full response signaling is supported. Avalon slaves always return `OKAY` as a response.

Ordering Model

Qsys interconnect provides responses in the same order as the commands are issued.

To prevent reordering, for slaves that accept reordering depths greater than 0, Qsys does not transfer the transaction ID from the master, but provides a constant transaction ID of 0. For slaves that do not reorder, Qsys allows the transaction ID to be transferred to the slave. To avoid cyclic dependencies, Qsys supports a single outstanding slave scheme for both reads and writes. Changing the targeted slave before all responses have returned stalls the master, regardless of transaction ID.

AXI and Avalon Ordering

According to the *AMBA Protocol Specifications*, there is no ordering requirement between reads and writes. However, Avalon has an implicit ordering model that requires transactions from a master to the same slave to be in order. As a result, there is a potential read-after-write risk when Avalon masters transact to AXI slaves. In response to this potential risk, Avalon interfaces provide a compile-time option to enforce strict order. When turned on, the Avalon interface waits for outstanding write responses before issuing reads.

Data Buses

Narrow bus transfers are supported. AXI write strobes can have any pattern that is compatible with the address and size information. Altera recommends that transactions to Avalon slaves follow Avalon `byteenable` limitations for maximum compatibility.

Note: Byte `0` is always bits `[7:0]` in the interconnect, following AXI's and Avalon's byte (address) invariance scheme.

Unaligned Address Commands

Unaligned address commands are commands with addresses that do not conform to the data width of a slave. Since Avalon-MM slaves accept only aligned addresses, Qsys modifies unaligned commands from AXI masters to the correct data width. Qsys must preserve commands issued by AXI masters when passing the commands to AXI slaves.

Note: Unaligned transfers are aligned if downsizing occurs. For example, when downsizing to a bus width narrower than that required by the transaction size, `AWSIZE` or `ARSIZE`, the transaction must be modified.

Avalon and AXI Transactions

Qsys 13.1 supports transaction between AXI and Avalon interfaces with some limitations.

Transaction Cannot Cross 4KB Boundaries

When an Avalon master issues a transaction to an AXI slave, the transaction cannot cross 4KB boundaries. Non-bursting Avalon masters already follow this boundary restriction.

Handling Read Side Effects

Read side effects can occur when more bytes than necessary are read by the slave, and the unwanted data that are read are later inaccessible on subsequent reads. For write commands, the correct byteenable paths are asserted based on the size of the transactions. For read commands, narrow-sized bursts are broken up into multiple non-bursting commands, and each command with the correct byteenable paths asserted.

Note: Qsys always assumes that the byteenable is asserted based on the size of the command, not the address of the command. For example, for a 32-bit AXI master that issues a read command with unaligned address starting at address `0x01`, and a burstcount of 2 to a 32-bit avalon slave, are treated as having a starting address of `0x00`.

AMBA AXI4 (version 2.0) Specification Support

Qsys allows connections between AXI4 components, and AXI3 and Avalon memory-mapped interface types. The sections that follow describe unique or exceptional AXI4 support in the Qsys software.

Refer to the *AMBA Protocol Specifications* for AXI4 on the ARM website for more information.

Related Information

AMBA Protocol Specifications

Burst Support

Qsys supports `INCR` bursts up to 256 beats. Qsys converts long bursts to multiple bursts in a packet with each burst having a length less than or equal to `MAX_BURST` when going to AXI3 or Avalon slaves.

For narrow-sized transfers, bursts with Avalon slaves as destinations are shortened to multiple non-bursting transactions in order to transmit the correct address to the slaves, since Avalon slaves always perform full-sized `datawidth` transactions.

Bursts with AXI3 slaves as destinations are shortened to multiple bursts, with each burst length less than or equal to 16. Bursts with AXI4 slaves as destinations are not shortened.

QoS

Qsys routes 4-bit QoS signals (Quality of Service Signaling) on the read and write address channels directly from the master to the slave.

Transactions from AXI3 and Avalon masters have a default value of `4'b0000`, which indicates that the transactions are not part of the QoS flow. QoS values are not used for slaves that do not support QoS.

For Qsys 13.1, there are no programmable QoS registers or compile-time QoS options for a master that overrides its real or default value.

Regions

For Qsys 13.1, there is no support for the optional regions feature. AXI4 slaves with `AXREGION` signals are allowed. `AXREGION` signals are driven with the default value of `0x0`, and are limited to one entry in a master's address map.

Write Response Dependency

Write response dependency as specified in the *AMBA Protocol Specifications* for AXI4 is not supported.

Related Information

AMBA Protocol Specifications

AWCACHE and ARCACHE

For AXI4, Qsys meets the requirement for modifiable and non-modifiable transactions. The modifiable bit refers to `ARCACHE[1]`and `AWCACHE[1]`.

Width Adaptation and Data Packing in Qsys

Data packing applies only to systems where the data width of masters is less than the data width of slaves.

The following rules apply:

- Data packing is supported when masters and slaves are Avalon-MM.
- Data packing is not supported when any master or slave is an AXI3, AXI4, or APB component.

For example, for a read/write command with a 32-bit master connected to a 64-bit slave, and a transaction of 2 burstcounts, Qsys sends 2 separate read/write commands to access the 64-bit data width of the slave. Data packing is only supported if the system does not contain AXI3, AXI4, or APB masters or slaves.

Ordering Model

Out of order support is not implemented in Qsys, version 13.1. Qsys processes AXI slaves as device non-bufferable memory types.

The following describes the required behavior for the device non-bufferable memory type:

- Write response must be obtained from the final destination.
- Read data must be obtained from the final destination.
- Transaction characteristics must not be modified.
- Reads must not be pre-fetched. Writes must not be merged.
- Non-modifiable read and write transactions.

(`AWCACHE[1] = 0 or ARCACHE[1] = 0`) from the same ID to the same slave must remain ordered. The interconnect always provides responses in the same order as the commands issued. Slaves that support reordering provide a constant transaction ID to prevent reordering. AXI slaves that do not reorder are provided with transaction IDs, which allows exclusive accesses to be used for such slaves.

Read and Write Allocate

Read and write allocate does not apply to Qsys interconnect, which does not have caching features, and always receives responses from an endpoint.

Locked Transactions

Locked transactions are not supported for Qsys, version 13.1.

Memory Types

For AXI4, Qsys processes transactions as though the endpoint is a device memory type. For device memory types, using non-bufferable transactions to force previous bufferable transactions to finish is irrelevant, because Qsys interconnect always identifies transactions as being non-bufferable.

Mismatched Attributes

There are rules for how multiple masters issue cache values to a shared memory region. The interconnect meets requirements as long as cache signals are not modified.

Signals

Qsys supports up to 64-bits for the `BUSER`, `WUSER` and `RUSER` sideband signals. AXI4 allows some signals to be omitted from interfaces by aligning them with the default values as defined in the *AMBA Protocol Specifications* on the ARM® website.

Related Information

AMBA Protocol Specifications

AMBA APB (version 1.0) Specification Support

AMBA APB provides a low-cost interface that is optimized for minimal power consumption and reduced interface complexity. You can use AMBA APB to interface to peripherals which are low-bandwidth and do not require the high performance of a pipelined bus interface. Signal transitions are sampled at the rising edge of the clock to enable the integration of APB peripherals easily into any design flow.

Qsys allows connections between APB components, and Avalon, AXI3, and AXI4 Avalon memory-mapped interface types. The following sections describe unique or exceptional APB support in the Qsys software.

Refer to the *AMBA Protocol Specifications* for AXI4 on the ARM website for more information.

Related Information

AMBA Protocol Specifications

Bridges

With APB, you cannot use bridge components that use multiple `PSELx` in Qsys. As a workaround, you can group PSELx, and then send the packet to the slave directly.

Altera recommends as an alternative that you instantiate the APB bridge and all the APB slaves in Qsys. You should then connect the slave side of the bridge to any high speed interface and connect the master side of the bridge to the APB slaves. Qsys creates the interconnect on either side of the APB bridge and creates only one `PSEL` signal.

Alternatively, you can connect a bridge to the APB bus outside of Qsys. Use an Avalon/AXI bridge to export the Avalon/AXI master to the top-level, and then connect this Avalon/AXI interface to the slave side of the APB bridge. Alternatively, instantiate the APB bridge in Qsys and export APB master to the top- level, and from there connect to APB bus outside of Qsys.

Burst Adaptation

APB is a non-bursting interface. Therefore, for any AXI or Avalon master with bursting support, a burst adapter is inserted before the slave interface and the burst transaction is translated into a series of non-bursting transactions before reaching the APB slave.

Width Adaptation

Qsys allows different data width connections with APB. When connecting a wider master to a narrower APB slave, the width adapter converts the wider transactions to a narrower transaction to fit the APB slave data width. APB does not support Write Strobe. Therefore, when connecting a narrower transaction to a wider APB slave, the slave cannot determine which byte lane to write, so the data at the slave might be overwritten or corrupted.

Error Response

Error responses are returned to the master. Qsys performs error mapping if the master is an AXI3 or AXI4 master, for example, `RRESP/BRESP= SLVERR`. For the case when the slave does not use `SLVERR` signal, an `OKAY` response is sent back to master by default.

Document Revision History

Table 8-28 indicates edits made to the *Qsys Interconnect* content since its creation.

Table 8-28: Document Revision History

Date	Version	Changes
November 2013	13.1.0	• HSSI clock support • Reset Sequencer • Interconnect pipelining
May 2013	13.0.0	• Added AMBA APB support. • Added auto-inserted Avalon-ST adapters feature. • Moved Address Span Extender to the *Qsys System Design Components*.
November 2012	12.1.0	• Added AMBA AXI4 support.
June 2012	12.0.0	• Added AMBA AXI3 support. • Added Avalon-ST. • Added Address Span Extender.
November 2011	11.0.1	Template update.
May 2011	11.0.0	Removed beta status.
December 2010	10.1.0	Initial release.

Related Information

Quartus II Handbook Archive