Qsys System Design Components 11

2013.11.4

QII51025

Subscribe Send Feedback

You can use Qsys system design components and IP cores to design your Qsys systems. The Qsys interfaces define components appropriate for streaming high-speed data, reading and writing registers and memory, and controlling off-chip devices.

Related Information

Avalon Interface Specifications

AMBA Protocol Specifications

Creating a System with Qsys

Creating Qsys Components

Qsys Interconnect

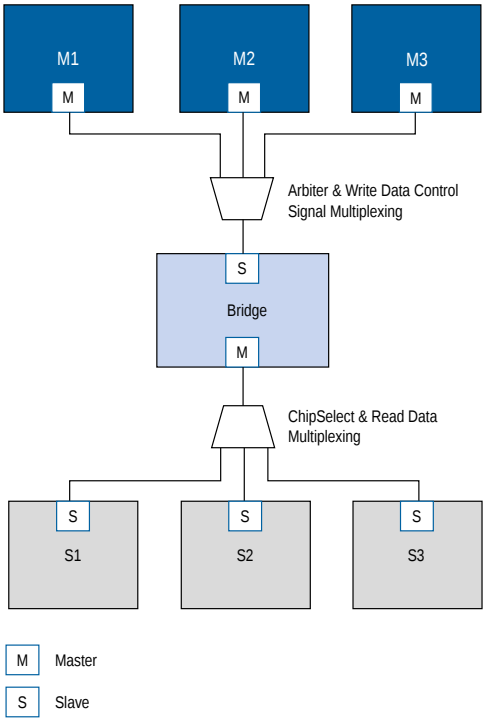
Bridges

Qsys provides bridge components to provide flexibility and control in your system implementation. Bridges are not end points for data, but rather affect the way data is transported between components. You can insert bridges between masters and slave interfaces to control the topology of a Qsys system, which affects the interconnect that Qsys generates. You can also use bridges to separate components in different clock domains and to isolate clock domain crossing logic.

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

In **Figure 1**, all three masters have logical connections to all three slaves, although physically each master connects only to the bridge.

Figure 11-1: Example of Bridge in a Qsys System



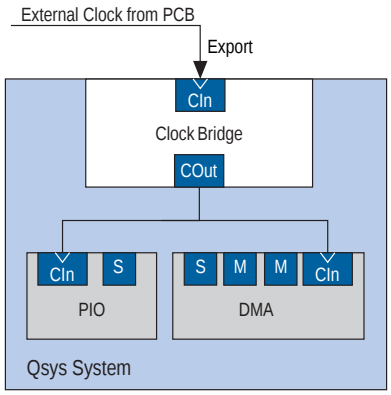
Transfers initiated to the bridge slave propagate to the bridge master in the same order in which they are initiated on the bridge slave.

Clock Bridge

The Clock Bridge allows you to connect a clock source to multiple clock input interfaces. You can use this bridge to connect a clock source that's outside the Qsys system through an exported interface to multiple clock input interfaces in the system.

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

Figure 11-2: Clock Bridge



Avalon-MM Clock Crossing Bridge

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

The Avalon-MM Clock Crossing Bridge uses asynchronous FIFOs to implement the clock crossing logic. The Clock Crossing Bridge has a number of parameters, including parameters to control the depth of the command and response FIFOs in both the master and slave clock domains. If the number of in-flight reads exceeds the depth of the response FIFO, the Clock Crossing Bridge stops sending reads. To maintain throughput for high-performance applications, increase the response FIFO depth from the default minimum depth, which is twice the maximum burst size.

Related Information

Creating a System With Qsys

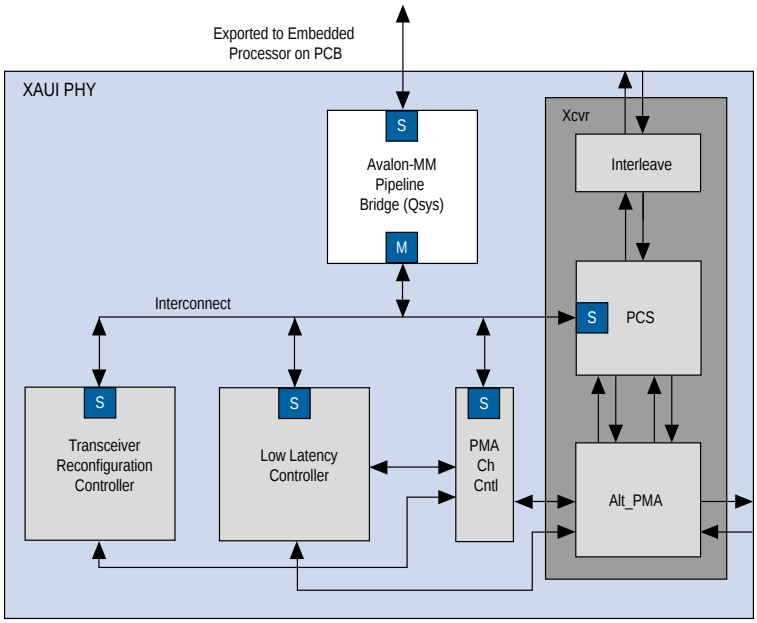
Avalon-MM Pipeline Bridge

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

You can also use the Avalon-MM bridge to export a single Avalon-MM slave interface that can be used to control multiple Avalon-MM slave devices, and you can optionally turn off the pipelining feature of this bridge. In this configuration, the bridge transfers commands received on its Avalon-MM slave interface to its Avalon-MM master port.

Figure 11-3 illustrates the use of an Avalon-MM Pipeline Bridge in a XAUI PHY transceiver IP core.

Figure 11-3: Avalon Bridge

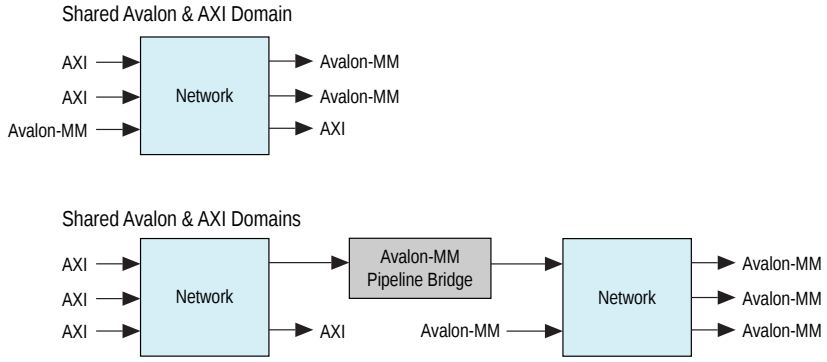


Because the Avalon-MM slave interface is exported to the pins of the device, having a single Avalon-MM slave port (rather than separate ports for each Avalon-MM slave device) reduces the pin count of the FPGA.

Bridges Between Avalon and AXI Interfaces

When designing a system, you can make connections between AXI and Avalon interfaces without the use of explicitly-instantiated bridges; the interconnect provides all necessary bridging logic. However, this does not prevent the use of explicit bridges to separate the AXI and Avalon domains. Using an explicit Avalon-MM bridge to separate the AXI and Avalon domains, as shown in **Figure 11-4**, reduces the amount of bridging logic in the interconnect, at the expense of concurrency.

Figure 11-4: Avalon-MM Pipeline Bridge Between Avalon-MM and AXI Networks



Address Span Extender

The Address Span Extender component creates a windowed bridge and allows memory-mapped master interfaces to access a larger address map than the width of their address signals allow. When connected to an address span extender, an address span restricted master can access a broader address range.

The extender splits up the larger addressable space into separate windows so that the master with a smaller address span can access the appropriate part of the memory.

For example, if the fast variant of a processor can address only 2GB of address span, and you need that processor to access a broader span, then you can use the address span extender to access the broader span by providing a window with a smaller address span. The same issue occurs with SoC devices. For example, an HPS subsystem in SoC devices can address only 1GB of address span within the FPGA. You can use the address span extender in this case, as well.

When you implement the address span extender in Qsys for a master with limited addressing space, you must first decide how large of an address space you want a particular slave to occupy in a master's address map.

This component allows you to define between 1 and 64 address windows, and accordingly, a given number of registers to hold the upper address bits for each window. In the component GUI, you must select the number of bits you want to access (**Expanded Master Byte Address Width**), the number of bits you want the master to see (**Slave Word Address Width**), and the number of sub-windows.

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

You can parametrize the address span extender with an initial fixed address value by entering an address for the **Reset Default for Master Window** option, and selecting `True` for the **Disable Slave Control Port** option, which allows the address span extender to function as a fixed, non-programmable component.

Each sub-window is equal in size and is stacked sequentially in the windowed slave interface's address space. To control the fixed address bits of a particular sub-window, you can write to the sub-window's register in the register control slave interface. Qsys structures the logic so that the Quartus II software can optimize away all unneeded bits.

The Address Span Extender component creates a windowed bridge and allows memory-mapped master interfaces to access a larger address map than the width of their address signals allow. When connected to an address span extender, an address span restricted master can access a broader address range. If **Burstcount Width** is set greater than 1, the read burst command is expressed in a single cycle, and assumes all byteenables are asserted on every cycle.

You can configure address ports within memory-mapped interfaces to be up to 64-bits wide. The address span extender enables a master to access a windowed portion of a larger memory map. The slave interface has an address port size corresponding to the address window. For example, when a component's master port is not 64-bit capable, you can use the Address Span Extender to enable it to access a specific 32-bit segment of a 64-bit address map.

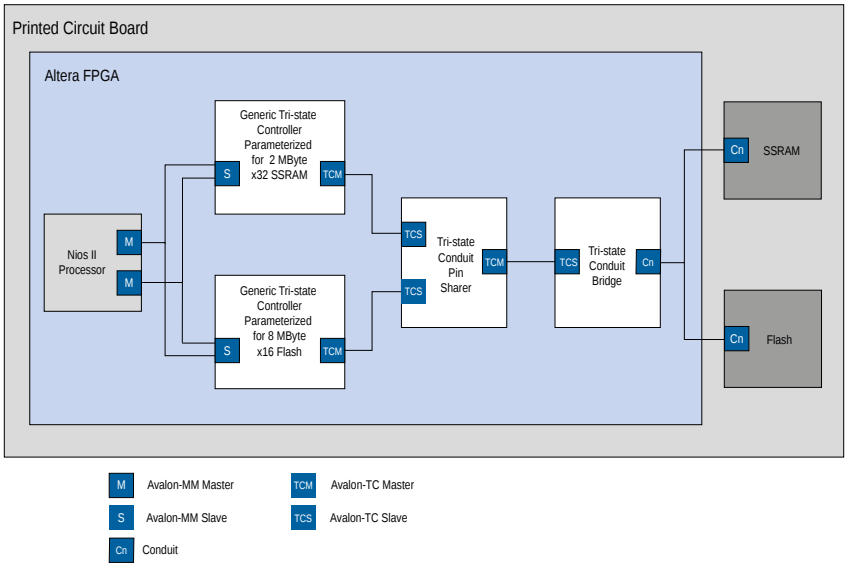
The Address Span Extender does not limit master and slave widths to a 32-bit and 64-bit configuration. You can use the Address Span Extender for other width configurations.

Tri-state Components

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

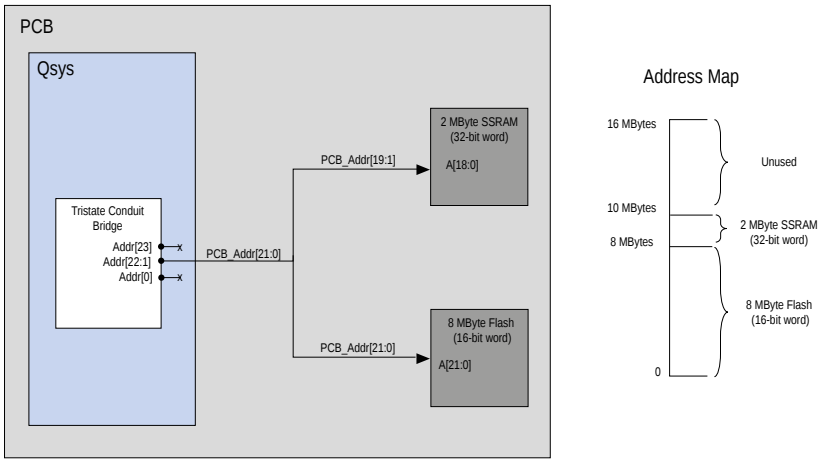
Figure 11-5 illustrates the typical use of tri-state components, and includes two Generic Tri-state Conduit Controllers. The first is customized to control a flash memory. The second is customized to control an off-chip SSRAM. The Tri-state Conduit Pin Sharer multiplexes between these two controllers, and the Tri-state Conduit Bridge converts between an on-chip encoding of tri-state signals and true bidirectional signals.

Figure 11-5: Tri-state Conduit System to Control Off-Chip SRAM and Flash Devices



By default, the Tri-state Conduit Pin Sharer and Tri-State Conduit Bridge present byte addresses. Each address location in many memory devices contains more than one byte of data. In **Figure 11-6**, the flash device operates on 16-bit words and must ignore the least-significant bit of the Avalon-MM address, and shows `addr[0]` as unconnected. The SSRAM memory operates on 32-bit words and must ignore the two, low-order memory bits. Because neither device requires a byte address, `addr[0]` is not routed on the PCB.

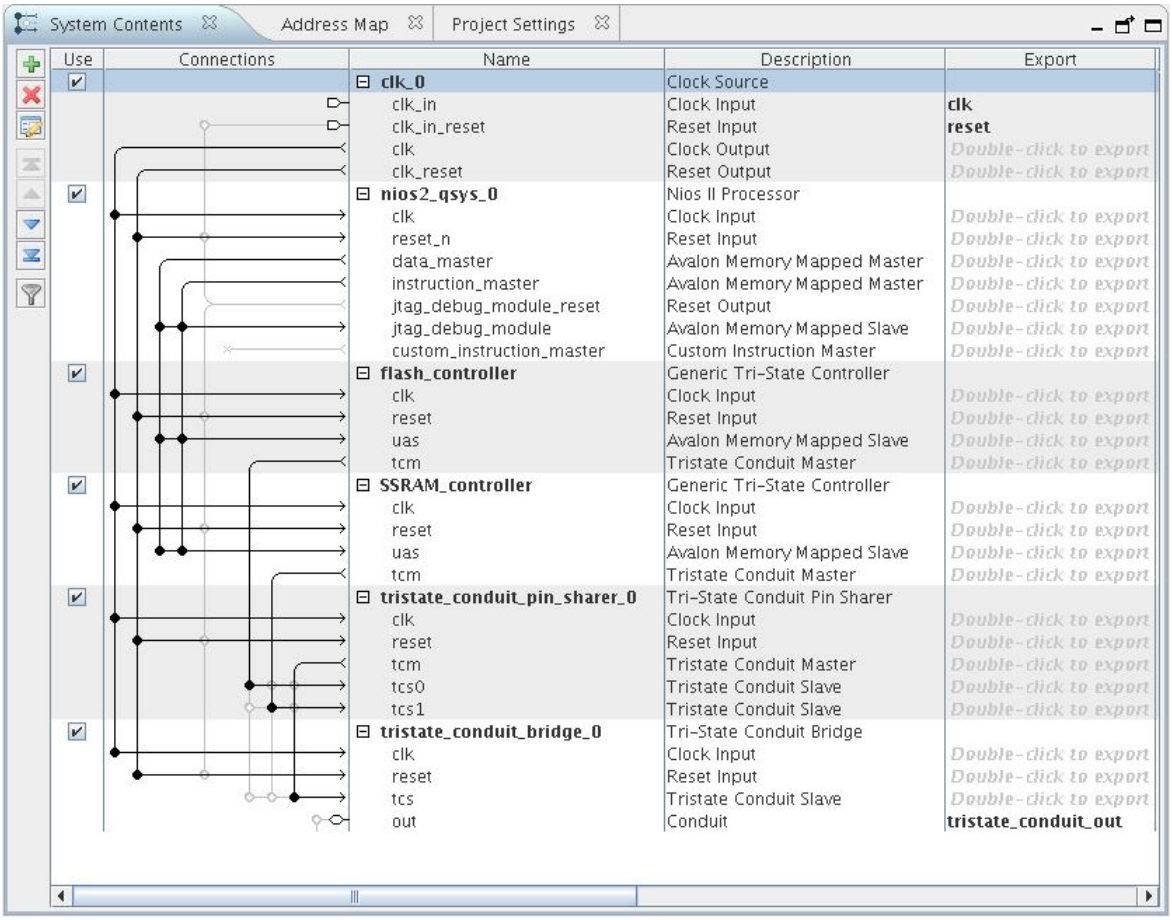
Figure 11-6: Address Connections from Qsys System to PCB



In **Figure 11-7**, the flash device responds to address range 0 MBytes to 8 MBytes-1. The SSRAM responds to address range 8 MBytes to 10 MBytes-1. The PCB schematic for the PCB connects `addr [20:2]` to `addr [18:0]` of the SSRAM device because the SSRAM responds to 32-bit word address. The 8 MByte flash device accesses 16-bit words; consequently, the schematic does not connect `addr[0]`. The `chipselect` signals select between the two devices.

Note: If you create a custom tri-state conduit master with word-aligned addresses, the Tri-state Conduit Pin Sharer does nothing to change or align the address signals. Figure 11-7 illustrates this example system in Qsys.

Figure 11-7: Tri-state Conduit System in Qsys



Related Information

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

Generic Tri-state Controller

The Generic Tri-state Controller provides a template for a controller that you can parameterize to reflect the behavior of an off-chip device.

You can use various parameters to customize the generic tri-state controller, such as the following:

- The width of the address and data signals
- The read and write wait times
- The bus-turnaround time
- The data hold time

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

The Generic Tri-state Controller includes the following interfaces:

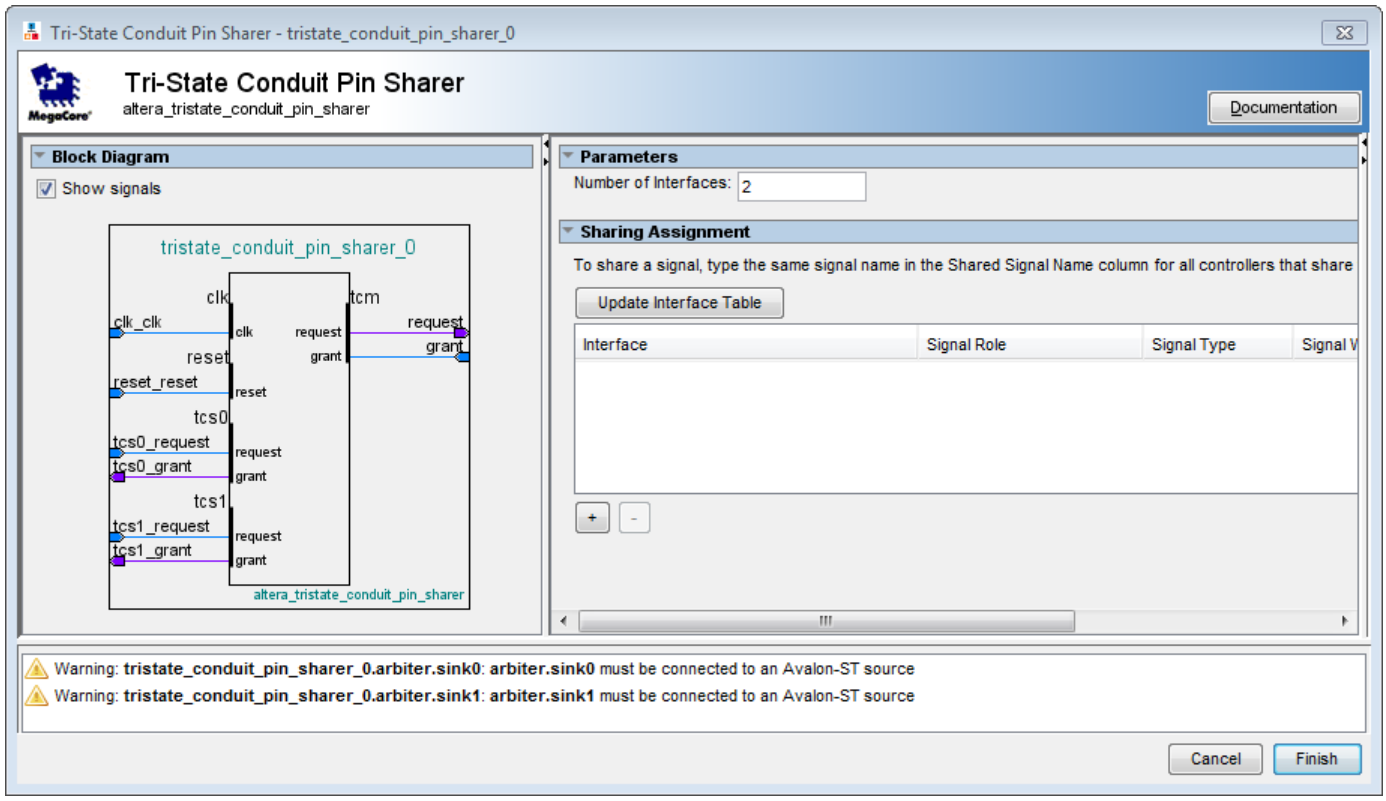
- **Memory-mapped slave interface**—This interface connects to an memory-mapped master, such as a processor.
- **Tristate Conduit Master interface**—Tri-state master interface usually connects to the tri-state conduit slave interface of the tri-state conduit pin sharer.
- **Clock sink**—The component's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

Tri-state Conduit Pin Sharer

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

The parameter editor includes a **Shared Signal Name** column, as shown in Figure 11-8.

Figure 11-8: Specifying Shared Signals Using the Tri-state Conduit Pin Sharer



If the widths of shared signals differ, the signals are aligned on their 0^{th} bit and the higher-order pins are driven to 0 whenever the smaller signal has control of the bus. Unshared signals always propagate through

the pin sharer. The tri-state conduit pin sharer uses the round-robin arbiter to select between tri-state conduit controllers.

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

Tri-state Conduit Bridge

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

Test Pattern Generator and Checker Cores

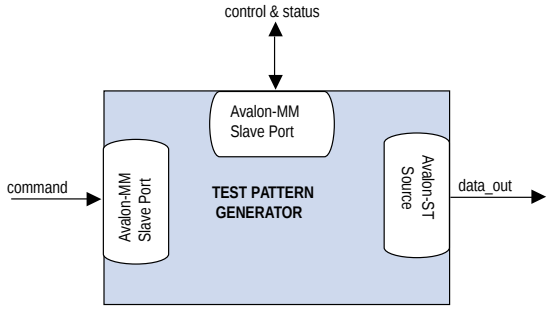
The data generation and monitoring solution for Avalon-ST consists of two components: a test pattern generator core that generates data, and sends it out on an Avalon-ST data interface, and a test pattern checker core that receives the same data and verifies it. Optionally, the data can be formatted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

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

Test Pattern Generator

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

Figure 11-9: Test Pattern Generator Core Block Diagram



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

Test Pattern Generator Command Interface

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

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

Test Pattern Generator Control and Status Interface

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

Test Pattern Generator Output Interface

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

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

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

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

Test Pattern Generator Functional Parameter

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

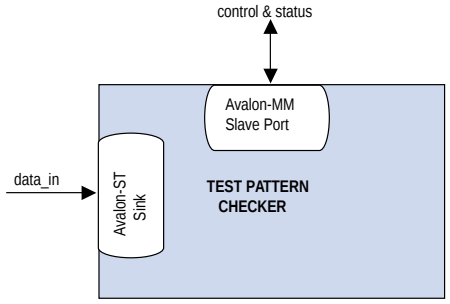
Test Pattern Checker

The test pattern checker core accepts data via an Avalon-ST interface, verifies it against the same predetermined pattern used by the test pattern generator to produce the data, and reports any exceptions to the control interface. You can parameterize most aspects of the test pattern checker's Avalon-ST interface

such as the number of error bits and the data signal width, thus allowing you to test components with different interfaces.

The test pattern checker has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register controls the rate at which data is accepted.

Figure 11-10: Test Pattern Checker



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

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

Test Pattern Checker Input Interface

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

Test Pattern Checker Control and Status Interface

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

Test Pattern Checker Functional Parameter

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

Test Pattern Checker Input Parameters

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

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

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

Software Programming Model for the Test Pattern Generator and Checker Cores

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

HAL System Library Support

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

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

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

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

Software Files Provided with the Test Pattern Generator

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

Note: Do not modify the software files.

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

Register Maps for the Test Pattern Generator and Checker Cores

Test Pattern Generator Control and Status Registers

Table 11-8 shows the offset for the test pattern generator control and status registers. Each register is 32-bits wide.

Table 11-1: Test Pattern Generator Control and Status Register Map

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

Table 11-9 describes the `status` register bits.

Table 11-2: Status Register Bits

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

Table 11-3: Control Register Bits

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

Table 11-4: Fill Register Bits

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

Test Pattern Generator Command Registers

Table 11-5 shows the offset for the command registers. Each register is 32-bits wide.

Table 11-5: Test Pattern Command Register Map

Offset	Register Name
base + 0	cmd_lo
base + 1	cmd_hi

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

Table 11-6: cmd_lo Register Bits

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

Table 11-7 describes the cmd_hi register bits.

Table 11-7: cmd_hi Register Bits

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

Test Pattern Checker Control and Status Registers

Table 11-8 shows the offset for the control and status registers. Each register is 32 bits wide.

Table 11-8: Test Pattern Checker Control and Status Register Map

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

Table 11-9: Status Register Bits

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

Table 11-10: Control Register Bits

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

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

Table 11-11: `exception_descriptor` Register Bits

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

Table 11-12: `indirect_select` Register Bits

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

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

Table 11-13: indirect_count Register Bits

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

.

Test Pattern Generator API

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

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

data_source_reset()

Information Type	Description
Prototype	`void data_source_reset(alt_u32 base);`
Thread-safe	No.

Information Type	Description
Include	**< data_source_util.h >**
Parameters	**base**—The base address of the control and status slave.
Returns	void.
Description	This function resets the test pattern generator core including all internal counters and FIFOs. The control and status registers are not reset by this function.

data_source_init()

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

data_source_get_id()

Information Type	Description
Prototype:	`int data_source_get_id(alt_u32 base);`
Thread-safe:	Yes.
Include:	< **data_source_util.h** >
Parameters:	`base`—The base address of the control and status slave.
Returns:	The test pattern generator core's identifier.
Description:	This function retrieves the test pattern generator core's identifier.

data_source_get_supports_packets()

Information Type	Description
Prototype:	`int data_source_init(alt_u32 base);`
Thread-safe:	Yes.
Include:	< **data_source_util.h** >
Parameters:	`base`—The base address of the control and status slave.
Returns:	

Information Type	Description
	1—Data packets are supported. 0—Data packets are not supported.
Description:	This function checks if the test pattern generator core supports data packets.

data_source_get_num_channels()

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

data_source_get_symbols_per_cycle()

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

data_source_set_enable()

Information Type	Description
Prototype:	`void data_source_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	< **data_source_util.h** >
Parameters:	

Information Type	Description
	`base`—The base address of the control and status slave. `value`—The ENABLE bit is set to the value of this parameter.
Returns:	`void.`
Description:	This function enables or disables the test pattern generator core. When disabled, the test pattern generator core stops data transmission but continues to accept commands and stores them in the FIFO

data_source_get_enable()

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

data_source_set_throttle()

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

data_source_get_throttle()

Information Type	Description
Prototype:	`int data_source_get_throttle(alt_u32 base);`

Information Type	Description
Thread-safe:	Yes.
Include:	< **data_source_util.h** >
Parameters:	`base`—The base address of the control and status slave.
Returns:	The throttle value.
Description:	This function retrieves the current throttle value.

data_source_is_busy()

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

data_source_fill_level()

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

data_source_send_data()

Information Type	Description
Prototype:	

Information Type	Description
	`int data_source_send_data(alt_u32 cmd_base, alt_u16 channel, alt_u16 size, alt_u32 flags, alt_u16 error, alt_u8 data_error_mask);`
Thread-safe:	No.
Include:	< **data_source_util.h** >
Parameters:	`cmd_base`—The base address of the command slave. `channel`—The channel to send the data on. `size`—The data size. `flags` —Specifies whether to send or suppress SOP and EOP signals. Valid values are `DATA_SOURCE_SEND_SOP`, `DATA_SOURCE_SEND_EOP`, `DATA_SOURCE_SEND_SUPRESS_SOP` and `DATA_SOURCE_SEND_SUPRESS_EOP`. `error`—The value asserted on the `error` signal on the output interface. `data_error_mask`—This parameter and the data are `XOR`ed together to produce erroneous data.
Returns:	Always returns 1.
Description:	This function sends a data fragment to the specified channel. If data packets are supported, user applications must ensure the following conditions are met: SOP and EOP are used consistently in each channel. Except for the last segment in a packet, the length of each segment is a multiple of the data width. If data packets are not supported, user applications must ensure the following conditions are met: No SOP and EOP indicators in the data. The length of each segment in a packet is a multiple of the data width.

Test Pattern Checker API

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

data_sink_reset() on page 11-25

data_sink_init() on page 11-25

data_sink_get_id() on page 11-25

data_sink_get_supports_packets() on page 11-26

data_sink_reset()

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

data_sink_init()

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

data_sink_get_id()

Information Type	Description
Prototype:	`int data_sink_get_id(alt_u32 base);`
Thread-safe:	Yes.
Include:	< **data_sink_util.h** >
Parameters:	`base`—The base address of the control and status slave.
Returns:	The test pattern checker core's identifier.
Description:	This function retrieves the test pattern checker core's identifier.

data_sink_get_supports_packets()

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

data_sink_get_num_channels()

Information Type	Description
Prototype:	`int data_sink_get_num_channels(alt_u32 base);`
Thread-safe:	Yes.
Include:	< **data_sink_util.h** >
Parameters:	`base`—The base address of the control and status slave.
Returns:	The number of channels supported.
Description:	This function retrieves the number of channels supported by the test pattern checker core.

data_sink_get_symbols_per_cycle()

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

data_sink_set enable()

Information Type	Description

Information Type	Description
Prototype:	`void data_sink_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	< **data_sink_util.h** >
Parameters:	`base`—The base address of the control and status slave. `value`—The ENABLE bit is set to the value of this parameter.
Returns:	`void.`
Description:	This function enables the test pattern checker core.

data_sink_get_enable()

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

data_sink_set_throttle()

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

data_sink_get_throttle()

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

data_sink_get_packet_count()

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

data_sink_get_error_count()

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

data_sink_get_symbol_count()

Information Type	Description
Prototype:	`int data_sink_get_symbol_count(alt_u32 base, alt_u32 channel);`
Thread-safe:	No.
Include:	< **data_sink_util.h** >
Parameters:	`base`—The base address of the control and status slave. `channel`—Channel number.
Returns:	The number of symbols received on the given channel.
Description:	This function retrieves the number of symbols received on a given channel.

data_sink_get_exception()

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

data_sink_exception_is_exception()

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

data_sink_exception_has_data_error()

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

data_sink_exception_has_missing_sop()

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

data_sink_exception_has_missing_eop()

Information Type	Description
Prototype:	`int data_sink_exception_has_missing_eop(int exception);`
Thread-safe:	Yes.
Include:	< **data_sink_util.h** >
Parameters:	`exception`—Exception descriptor.
Returns:	1—Missing EOP. 0—Other exception types.
Description:	

Information Type	Description
	This function checks if a given exception descriptor indicates missing EOP.

data_sink_exception_signalled_error()

Information Type	Description
Prototype:	`int data_sink_exception_signalled_error(int exception);`
Thread-safe:	Yes.
Include:	< **data_sink_util.h** >
Parameters:	`exception`—Exception descriptor.
Returns:	The signalled error value.
Description:	This function retrieves the value of the signalled error from the exception.

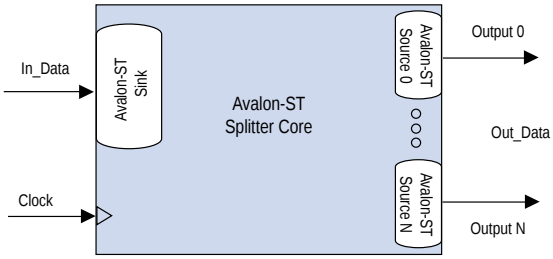
data_sink_exception_channel()

Information Type	Description
Prototype:	`int data_sink_exception_channel(int exception);`
Thread-safe:	Yes.
Include:	< **data_sink_util.h** >
Parameters:	`exception`—Exception descriptor.
Returns:	The channel number on which the given exception occurred.
Description:	This function retrieves the channel number on which a given exception occurred.

Splitter Core

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

Figure 11-11: Avalon-ST Splitter Core



The Avalon-ST Splitter core copies input signals from the input interface to the corresponding output signals of each output interface without altering the size or functionality. This includes all signals except for the `ready` signal. The core includes a clock signal to determine the Avalon-ST interface and clock domain where the core resides. Because the clock signal is unused internally, latency is not introduced when using this core.

Splitter Core Backpressure

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

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

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

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

Splitter Core Interfaces

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

Table 11-14: Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Configurable.
Channel	Supported (optional).
Error	Supported (optional).
Packet	Supported (optional).

Splitter Core Parameters

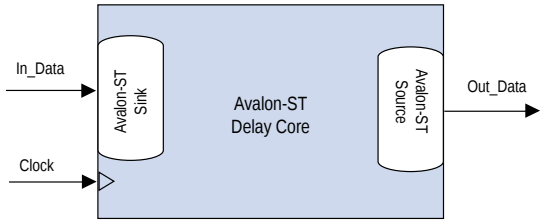
Table 11-15: Configurable Splitter Core Parameters

Parameter	Legal Values	Default Value	Description
Number Of Outputs	1 to 16	2	The number of output interfaces. The value of 1 is supported for some cases of parameterized systems in which no duplicated output is required.
Qualify Valid Out	0 or 1	1	Determines whether the `Out_Valid` signal is gated or non-gated when backpressure is applied.
Data Width	1–512	8	The width of the data on the Avalon-ST data interfaces.
Bits Per Symbol	1–512	8	The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.
Use Packets	0 or 1	0	Indicates whether or not data packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use Channel	0 or 1	0	The option to enable or disable the channel signal.
Channel Width	0-8	1	The width of the `channel` signal on the data interfaces. This parameter is disabled when **Use Channel** is set to 0.
Max Channels	0-255	1	The maximum number of channels that a data interface can support. This parameter is disabled when **Use Channel** is set to 0.
Use Error	0 or 1	0	The option to enable or disable the error signal.
Error Width	0–31	1	The width of the `error` signal on the output interfaces. A value of 0 indicates that the error signal is not used. This parameter is disabled when **Use Error** is set to 0.

Delay Core

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

Figure 11-12: Avalon-ST Delay Core



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

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

Delay Core Reset Signal

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

Delay Core Interfaces

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

Table 11-16: Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Not supported.
Data Width	Configurable.
Channel	Supported (optional).
Error	Supported (optional).
Packet	Supported (optional).

Delay Core Parameters

Table 11-17: Configurable Delay Core Parameters

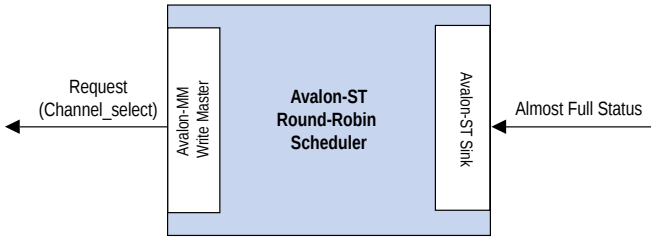
Parameter	Legal Values	Default Value	Description
Number Of Delay Clocks	0 to 16	1	Specifies the delay the core introduces, in clock cycles. The value of 0 is supported for some cases of parameterized systems in which no delay is required.
Data Width	1–512	8	The width of the data on the Avalon-ST data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

Round Robin Scheduler

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

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

Figure 11-13: Avalon-ST Round Robin Scheduler Block Diagram



Round Robin Scheduler Interfaces

The following interfaces are available in the Avalon-ST Round Robin Scheduler core:

- Almost-Full Status Interface
- Request Interface

Almost-Full Status Interface

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

Table 11-18: Avalon-ST Interface Feature Support

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

Request Interface (Round Robin Scheduler)

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

Round Robin Scheduler Operation

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

The scheduler only requests 1 beat of data from a channel at each transaction. To request 1 beat of data from channel *n*, the scheduler writes the value `1` to address (4 ×*n*). For example, if the scheduler is requesting data

from channel 3, the scheduler writes `1` to address `0xC`. At every clock cycle, the scheduler requests data from the next channel. Therefore, if the scheduler starts requesting from channel 1, at the next clock cycle, it requests from channel 2. The scheduler does not request data from a particular channel if the almost-full status for the channel is asserted. In this case, one clock cycle is used without a request transaction.

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

Table 11-19: Ports for the Avalon-ST Round Robin Scheduler

Signal	Direction	Description
Clock and Reset		
`clk`	In	Clock reference.
`reset_n`	In	Asynchronous active low reset.
Avalon-MM Request Interface		
`request_address (log2 Max_Channels-1:0)`	Out	The write address used to indicate which channel has the request.
`request_write`	Out	Write enable signal.
`request_writedata`	Out	The amount of data requested from the particular channel. This value is always fixed at 1.
`request_waitrequest`	In	Wait request signal, used to pause the scheduler when the slave cannot accept a new request.
Avalon-ST Almost-Full Status Interface		
`almost_full_valid`	In	Indicates that `almost_full_channel` and `almost_full_data` are valid.
`almost_full_channel (Channel_Width-1:0)`	In	Indicates the channel for the current status indication.
`almost_full_data (log2 Max_Channels-1:0)`	In	A 1-bit signal that is asserted high to indicate that the channel indicated by `almost_full_channel` is almost full.

Round Robin Scheduler Parameters

Table 11-20: Configurable Parameters for Avalon-ST Round Robin Scheduler

Parameters	Values	Description
Number of channels	2–32	Specifies the number of channels the Avalon-ST Round Robin Scheduler supports.

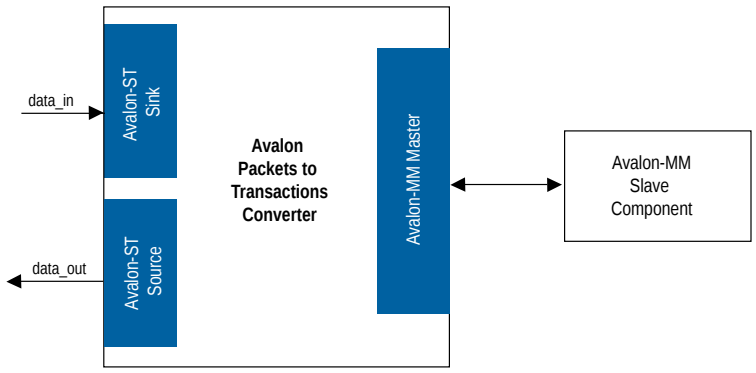
Parameters	Values	Description
Use almost-full status	0–1	Specifies whether the almost-full interface is used. If the interface is not used, the core always requests data from the next channel at the next clock cycle.

Packets to Transactions Converter

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

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

Figure 11-14: Avalon Packets to Transactions Converter Core



Related Information

Avalon Interface Specifications

Packets to Transactions Converter Interfaces

Table 11-21: Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Not supported.
Error	Not used.
Packet	Supported.

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

Packets to Transactions Converter Operation

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

Packets to Transactions Converter Data Packet Formats

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

The Packets to Transactions Converter core expects incoming data streams to be in the formats shown in Table 11-22.

Table 11-22: Data Packet Formats

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

Related Information

Packets to Transactions Converter Interfaces on page 11-38

Packets to Transactions Converter Supported Transactions

Table 11-23 lists the Avalon-MM transactions supported by the Packets to Transactions Converter core.

Table 11-23: Transaction Supported

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

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

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

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

Packets to Transactions Converter Malformed Packets

The following are examples of malformed packets:

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

Streaming Pipeline Stage

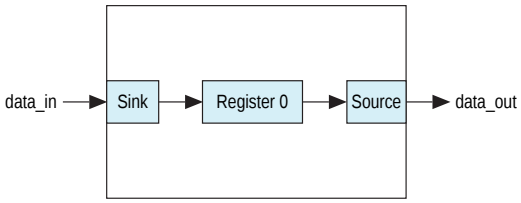
The Avalon-ST pipeline stage receives data from an Avalon-ST source interface, and outputs the data to an Avalon-ST sink interface. In the absence of back pressure, the Avalon-ST pipeline stage source interface outputs data one cycle after receiving the data on its sink interface.

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

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

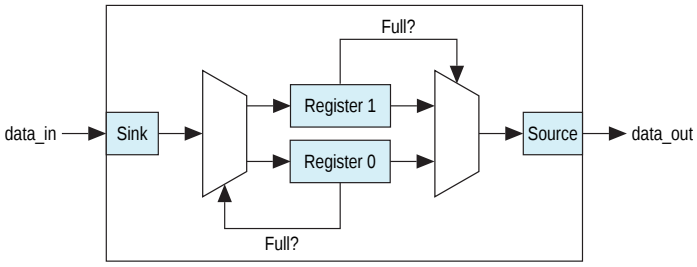
If the ready signal is not pipelined, the pipeline stage becomes a simple register, as shown in **Figure 11-15**

Figure 11-15: Pipeline Stage Simple Register



If the ready signal is pipelined, the pipeline stage must also include a second "holding" register, as shown in **Figure 11-16**.

Figure 11-16: Pipeline Stage Holding Register



Streaming Channel Multiplexer and Demultiplexer Cores

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

The multiplexer and demultiplexer cores can transfer data between interfaces on cores that support the unidirectional flow of data. The multiplexer and demultiplexer allow you to create multiplexed or demulti-

plexed datapaths without having to write custom HDL code. The multiplexer includes a Round-Robin Scheduler.

Software Programming Model For the Multiplexer and Demultiplexer Components

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

Multiplexer

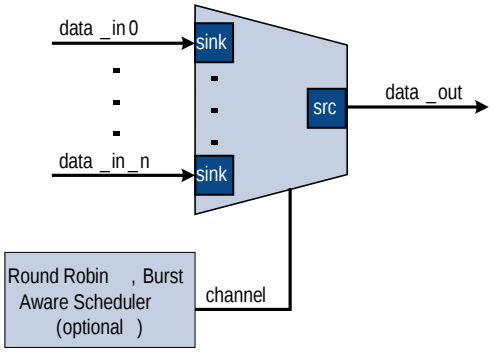
The Avalon-ST multiplexer takes data from a variety of input data interfaces, and multiplexes the data onto a single output interface. The multiplexer includes a round-robin scheduler that selects from the next input interface that has data. Each input interface has the same width as the output interface,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

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

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

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

Figure 11-17: Multiplexer



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

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

Multiplexer Input Interfaces

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

Multiplexer Output Interface

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

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

You can configure the following parameters for the output interface:

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

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

Multiplexer Parameters

You can configure the following parameters for the multiplexer:

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

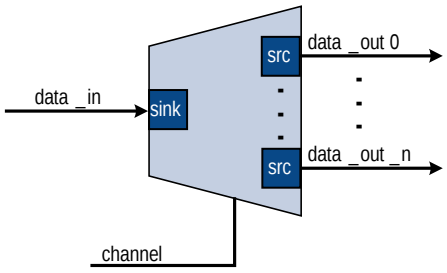
Demultiplexer

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

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

to select the output for the data; the remainder of the channel bits are forwarded to the appropriate output interface unchanged.

Figure 11-18: Demultiplexer



Demultiplexer Input Interface

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

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

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

Demultiplexer Output Interface

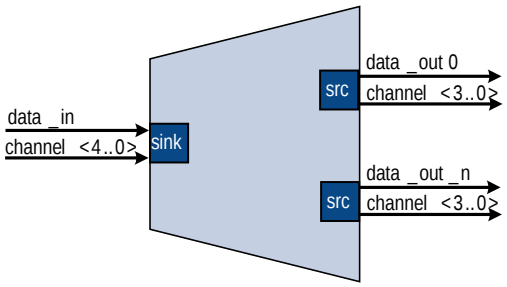
Each output interface carries data from a subset of channels from the input interface. Each output interface is identical; all have the same symbol and data widths, error widths, and channel widths. The symbol, data, and error widths are the same as the input interface. The width of the `channel` signal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

Demultiplexer Parameters

You can configure the following parameters for the demultiplexer:

- **Number of Output Ports**—The number of output interfaces that the multiplexer supports Valid values are 2 to 16.
- **High channel bits select output**—When this option is turned on, the high bits of the input channel signal are used by the demultiplexing function and the low order bits are passed to the output. When this option is turned off, the low order bits are used and the high order bits are passed through.
- Figure 11-19 illustrates the significance of the location of signals; for example, there is one input interface and two output interfaces. If the low-order bits of the channel signal select the output interfaces, the even channels goes to channel 0, and the odd channels goes to channel 1. If the high-order bits of the channel signal select the output interface, channels 0 to 7 goes to channel 0 and channels 8 to 15 goes to channel 1.

Figure 11-19: Select Bits for the Demultiplexer



Single-Clock and Dual-Clock FIFO Cores

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

Figure 11-20: Avalon-ST Single Clock FIFO Core

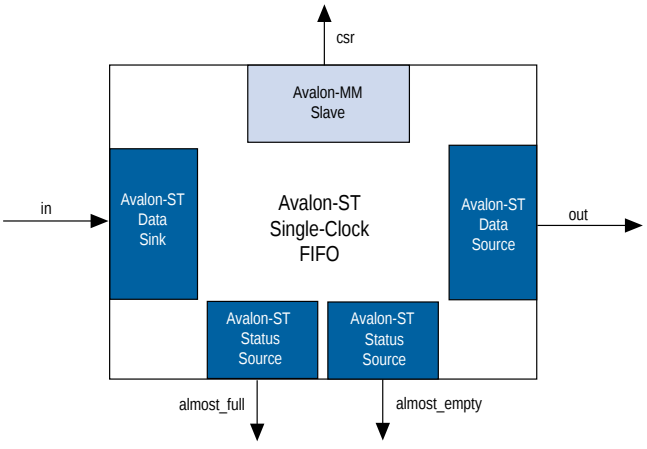
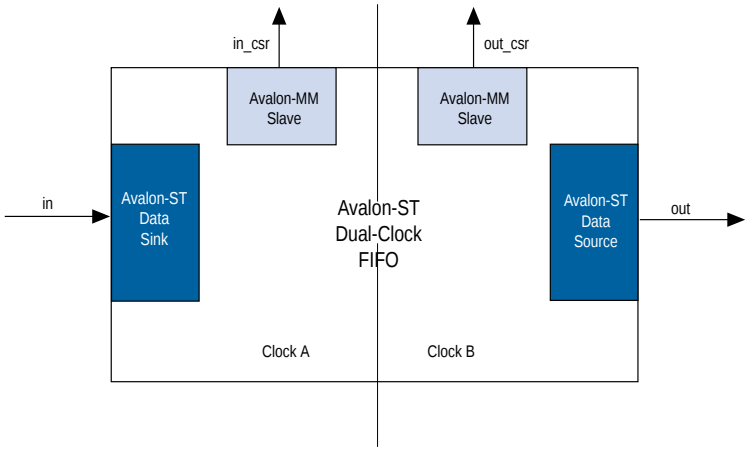


Figure 11-21: Avalon-ST Dual Clock FIFO Core



Interfaces Implemented in FIFO Cores

The following interfaces are implemented in FIFO cores:

Avalon-ST Data Interface

Each FIFO core has an Avalon-ST data sink and source interfaces. The data sink and source interfaces in the dual-clock FIFO core are driven by different clocks. **Table 11-24** shows the properties of the Avalon-ST interfaces.

Table 11-24: Properties of Avalon-ST Interfaces

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

Avalon-MM Control and Status Register Interface

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

Related Information

- **Avalon-ST Single-Clock FIFO Registers** on page 11-49

Avalon-ST Status Interface

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

FIFO Operating Modes

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

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

Fill Level of the FIFO Buffer

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

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

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

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

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

You can obtain the almost-full and almost-empty statuses from `almost_full` and `almost_empty` interfaces (Avalon-ST status source). The core asserts the `almost_full` signal when the fill level is equal

to or higher than the almost-full threshold. Likewise, the core asserts the `almost_empty` signal when the fill level is equal to or lower than the almost-empty threshold.

Related Information

- **Avalon-ST Single-Clock FIFO Registers** on page 11-49

Configurable Parameters for the Single-Clock and Dual-Clock FIFO Cores

Table 11-25 describes the parameters that you can configure for the Single-Clock and Dual-Clock FIFO cores.

Table 11-25: Configurable Parameters

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

Parameter	Legal Values	Description
Read pointer synchronizer length	2–8	The length of the read pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability.
Use Max Channel	—	Turn on this parameter to specify the maximum channel number.
Max Channel	1–255	Maximum channel number.

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

Related Information

- **Understanding Metastability in FPGAs**
- **Managing Metastability**

Avalon-ST Single-Clock FIFO Registers

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

Table 11-26: Avalon-ST Single-Clock FIFO Registers

32-Bit Word Offset	Name	Access	Reset	Description
0	`fill_level`	R	0	24-bit FIFO fill level. Bits 24 to 31 are unused.
1	Reserved	—	—	Reserved for future use.
2	`almost_full_threshold`	RW	**FIFO depth**–1	Set this register to a value that indicates the FIFO buffer is getting full.
3	`almost_empty_threshold`	RW	0	Set this register to a value that indicates the FIFO buffer is getting empty.
4	`cut_through_threshold`	RW	0	**0**—Enables store and forward mode. **Greater than 0**—Enables cut-through mode and specifies the minimum of entries in the FIFO buffer before the `valid` signal on the Avalon-ST source interface is asserted. Once the FIFO core starts sending the data to the downstream component, it continues to do so until the end of the packet. This register applies only when the **Use store and forward** parameter is turned on.

32-Bit Word Offset	Name	Access	Reset	Description
5	drop_on_error	RW	0	**0**—Disables drop-on error. **1**—Enables drop-on error. This register applies only when the **Use packet** and **Use store and forward** parameters are turned on.

The in_csr and out_csr interfaces in the Avalon-ST Dual Clock FIFO core reports the FIFO fill level. Table 11-27 describes the fill level.

Table 11-27: Register Description for Avalon-ST Dual-Clock FIFO

32-Bit Word Offset	Name	Access	Reset Value	Description
0	fill_level	R	0	24-bit FIFO fill level. Bits 24 to 31 are unused.

Refer to the *Avalon Interface Specifications* or *Avalon Memory-Mapped Design Optimizations* for more information.

Related Information

- **Avalon Interface Specifications**
- **Avalon Memory-Mapped Design Optimizations**

Document Revision History

Table 11-28 indicates edits made to the *Qsys System Design Components* content since its creation.

Table 11-28: Document Revision History

Date	Version	Changes
November 2013	13.1.0	• AXI Bridge
May 2013	13.0.0	• Added Streaming Pipeline Stage support. • Added AMBA APB support.
November 2012	12.1.0	• Moved relevant content from Embedded IP User Guide.

Related Information

Quartus II Handbook Archive