SoC Designer Plus
Version 9.0.0

AXI4 Protocol Bundle
User Guide

Non-Confidential

ARM DUI 0955C

SoC Designer
AXI4 Protocol Bundle User Guide

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Confidentiality	Change
February 2016	A	Non-Confidential	Release with 8.3
March 2016	B	Non-Confidential	Release with 8.4
November 2016	C	Non-Confidential	Release with 9.0.0

Non-Confidential Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

http://www.arm.com

Table of Contents

1 Introduction

This is the user guide for the SoC Designer AXI4 Protocol Bundle. This protocol bundle contains SoC Designer components, probes, and the transaction port interfaces for the ARM AXI4 protocol (includes support for AMBA4 AXI).

2 Requirements

The AXI4 protocol bundle requires the following:

- SoC Designer v8.4 or later
- Compilation tools as set forth in the *SoC Designer Installation Guide*.

3 Bundle Contents

This bundle contains protocol support packages for the ARM AMBA AXI4 protocol including ACE.

3.1 AXI4 Models and Examples

Generic components such as memory are included in this bundle. Also provided is example source code to help users develop custom AXI4 components.

3.2 AXI4 Probes

Probes provide visibility into transactions between two components. AXI4-specific probes are included in this protocol bundle.

3.3 AXI4 Ports

AXI4 transaction port definition header files and libraries are included in this package. These are required during runtime of any components with AXI4 ports and also when creating components with AXI4 ports.

4 AXI Adapter Compatibility

The following matrix describes the protocol support for the AXI adapters. AXI4toAXIv2, AXIv2ToAXI4, and AXI4ToAXI4 adapters are included in this protocol bundle.

Adapter Name	*Protocol Support*	*Compatibility*
AXIv2	AMBA3 AXI	Only compatible with AXIv2 ports. Connection to AXI4 ports requires an adapter component.
AXI4	AXI4 to AXI4 AXI4 to AXI-Lite+DVM AXI-Lite to AXI4	Compatible using the AXI4ToAXI4 adapter component.

Table 4-1 AXI Adapter Compatibility Matrix

5 Models

Table 5-1 lists the AXI4 components included in this bundle. These are described in more detail throughout this section.

Note: This document uses the following convention when referring to AXI4 components: AXI4*_<*ComponentName*>. The asterisk (*) represents the five variants of the AXI4 Protocol that you can specify (ACE, ACE-Lite+DVM, ACE-Lite, AXI4, or AXI4-Lite).

Component	*Description*
AXI4*_Master	This is an example AXI4 master component.
AXI4*_Slave	This is an example AXI4 slave component.
AXI4*_Mem	A generic AXI4 memory component with an AXI slave interface.
AXI4*_Stub	A scriptable AXI4 master component.
MxAXI4	A generic model of an AXI4 interconnect component.
AXI4ToAXIv2	Bridges an AXI4 master component to a legacy AXIv2 slave component.
AXIv2ToAXI4	Bridges a legacy AXIv2 master component to an AXI4 slave component.
AXI4ToAXI4	Converts AXI traffic between different AXI4 variants.

Table 5-1 AXI4 Components

The *.conf* files for AXI4 components are located under the *$MAXSIM_PROTOCOLS\AXI4\etc* directory.

5.1 AXI4*_Master

AXI4* _Master is an example AXI component using the AXI4 master port.

There is an example system which uses AXI4* _Master located in *$MAXSIM_PROTOCOLS/AXI4/examples/AXI4MasterSlave*.

This example component is provided in source code form under *$MAXSIM_PROTOCOLS/AXI4/src/AXI4_Master*.

Figure 5-1 shows the AXI4*_Master component:

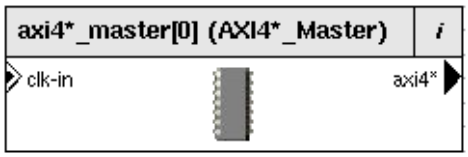


Figure 5-1 AXI4*_Master

Table 5-2 lists the component parameters.

Name	*Description*
Address Width	Width in bits of the address bus. Supported values are 32 to 63.
B ready delay	Number of delay cycles for the B channel ready response.
Data Width	Width in bits of the data bus. It must match the data bus width of the connected model. Allowed values are: 32, 64, 128, 256, 512, and 1024.
Enable	If *false*, the component will not generate any transactions.
Enable aw & w simultaneous	Used to enable the AW and W channels at the same time. Usually W follows AW, but you can change this parameter so that both channels go high at the same time.
Enable Debug Messages	When set to *true*, the model debug messages are displayed as output.
Enable Fast Debug	When set to true, the model runs a fast debug access test which writes and reads 50 MB of random binary data to memory starting from address 0x28080. This test runs each time the model is reset.
Enable multiple reads	If *true*, the component will try to issue a second read transaction even if the first read transaction has not completed.
Enable output	Enable printing of informational messages about transactions in a SoC Designer Console window.
Protocol Variant	Select from the following options: ACE, ACE-Lite+DVM, ACE-Lite, AXI4, AXI4-Lite.
R ready delay	Number of delay cycles for the R channel ready response.

Start address	The starting address of transactions. Address will be incremented by the data width for each transaction.
Start address2	Specifies the address for the second read transaction in case *Enable multiple reads* is set to *true*.
Start with writes	If *true*, the component will issue a write as the first transaction. If *false*, the component will issue a read as the first transaction.

Table 5-2 AXI4*_Master Parameters

5.2 AXI4*_Slave

AXI4* _Slave is an example AXI component using the AXI4 slave port.

There is an example system which uses AXI4* _Slave, which is located in *$MAXSIM_PROTOCOLS/AXI4/examples/AXI4MasterSlave*.

This example component is provided in source code form under *$MAXSIM_PROTOCOLS/AXI4/src/AXI4_Slave*.

Figure 5-2 shows the AXI4*_Slave component:

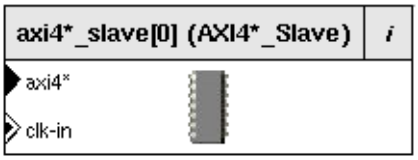


Figure 5-2 AXI4*_Slave

Table 5-3 lists the component parameters:

Name	*Description*
Address Width	Width in bits of the address bus. Supported values are 32 to 63.
AR ready delay	Number of delay cycles for the AR channel ready response.
AW ready delay	Number of delay cycles for the AW channel ready response.
B valid delay	Number of delay cycles for the B channel valid response.
Data Width	Width in bits of the data bus. It must match the data bus width of the connected model. Allowed values are 32, 64, 128, 256, 512, and 1024.
Enable Debug Messages	When set to *true*, the model debug messages are displayed as output.
Protocol Variant	Select from the following options: ACE, ACE-Lite+DVM, ACE-Lite, AXI4, AXI4-Lite.
R valid delay	Number of delay cycles for the R channel valid response.

W ready delay	Number of delay cycles for the W channel ready response.

Table 5-3 AXI4*_Slave Parameters

5.3 AXI4*_Mem

AXI4*_Mem is a generic AXI4 memory model with an AXI slave interface. Figure 5-3 shows the AXI4*_Mem component:

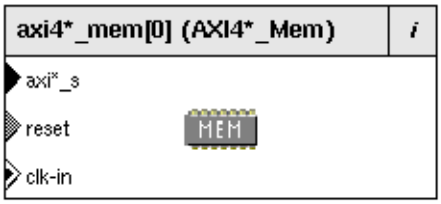


Figure 5-3 AXI4*_Mem

The AXI4*_Mem component supports both AXI4 and the ACE-Lite protocol. Use the "Protocol Variant" parameter to select between the two modes.

In ACE-Lite mode the AXI4*_Mem component accepts all ACE-Lite transactions as follows (see the ARM AMBA AXI and ACE Protocol Specification for details):

- All types of Reads and Writes are treated as ordinary Reads and Writes.
- All cache maintenance operations are responded to immediately with an OK status.
- All barriers are responded to immediately with an OK status.

Table 5-4 lists the component parameters.

Name	*Description*	*Init/Runtime*[1]
Address Width	Width in bits of the address bus. Supported range is 32 to 64. Settable at sdcanvas time only; not settable at runtime. Refer to Section 5.9 for important implementation notes on this parameter.	Init
AR-to-R delay	Amount of cycles to wait before sending the first beat of read data after an AR transfer has completed.	Runtime
axi_name[0-5] axi_size[0-5] axi_start[0-5]	These parameters are obsolete and should be left at their default values.[2]	Init

[1] Runtime means the parameter can be dynamically changed during simulation, Init means it can be changed only when building the system.

[2] ARM recommends using the Memory Map Editor (MME) in SoC Designer, which provides centralized viewing and management of the memory regions available to the components in a system. For information about migrating existing systems to use the MME, refer to the SoC Designer User Guide.

Data width	Width in bits of the data bus. It must match the data bus width of the connected model. Allowed values are 32, 64, 128, 256, 512, and 1024.	Init
Disk Backed Memory[3]	When enabled, the model uses a file on the disk to hold the simulated memory contents, to limit the RAM consumption of the simulator. The default setting is Disabled.	Init
Disk Backed Memory Size(Bytes)[3]	If 'Disk Backed Memory' is enabled, this 32-bit integer value is used as the default value of the memory being modeled. The default value is 100 GB. If 'Disk Backed Memory' is set to False, Disk Backed Memory Size (Bytes) is ignored.	Init
Disk Backed Memory-RAM Limit (MB)[3]	If 'Disk Backed Memory' is enabled, this parameter determines how much memory each instance can consume before swapping occurs. The default value is 250 MB. If 'Disk Backed Memory' is set to False, Disk Backed Memory RAM Limit (MB) is ignored.	Init
Enable Debug Messages	When set to *true*, the model debug messages are displayed as output.	Runtime
Enable Warnings	When set to *true*, this parameter enables printing of warning messages.	Init
Exclusive Monitor	When set to *true* (the default setting), the memory does additional checking for exclusive access requests and can return EXOKAY for success on RRESP or BRESP. When set to *false*, exclusive access requests always return OKAY, which is a failure code. Only the *true* value allows the memory to be fully compliant with the AMBA AXI and ACE specification.	Init
Num of Exclusive Monitors	The maximum number of exclusive monitors needed. This parameter becomes active only when the *Exclusive Monitor* parameter is set to *true*. If this value is set to 0 when *Exclusive Monitor*= true, the exclusive monitor is not turned on.	Init
Protocol Variant	AXI4 and ACE-Lite.	Init
RAM Usage Limit (MB)	If 'Disk backed memory' is set to true, this integer value will set the threshold for the memory model on the RAM consumption before starting to swap its contents to the file. If 'Disk backed memory' is set to false, it will be Ignored.	Init

[3] Refer to Section 5.10 for more information about using the Disk Backed Memory parameters.

User Width	Width of the user data.	Init
WS read	The number of wait cycles introduced for a Read access is given by this parameter.	Init
WS write	The number of wait cycles introduced for a Write access is given by this parameter.	Init
W-to-B delay	Number of cycles to wait before sending the B response after the Write data transfer has completed.	Runtime

Table 5-4 AXI4*_Mem Parameters

5.4 AXI4*_Stub

AXI4*_Stub is an AXI4 master component which can be controlled with a SoC Designer *mxscr* script. Figure 5-4 shows the AXI4*_Slave component:

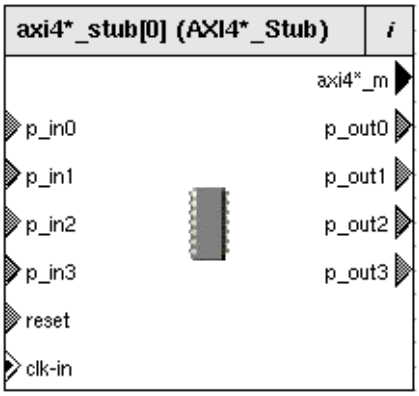


Figure 5-4 AXI4*_Stub

Table 5-5 lists the component parameters.

Name	*Description*
Address Width	Width in bits of the address bus. Supported values are 32 to 63.
axi_name[0-5] axi_size[0-5] axi_start[0-5]	These parameters are obsolete and should be left at their default values. *ARM recommends using the Memory Map Editor (MME) in SoC Designer, which provides centralized viewing and management of the memory regions available to the components in a system. For information about migrating existing systems to use the MME, refer to Chapter 9 of the SoC Designer User Guide.*
CPP include path	Additional include path for header files to be used by script preprocessor.
Data Width	Width in bits of the data bus. It must match the data bus width of the connected model. Allowed values are 32, 64, 128, 256, 512, and 1024.

Enable Debug Messages	When set to *true*, the model debug messages are displayed as output.
Memory Init Byte Value	Value to initialize the scratch memory used by the stub.
Protocol Variant	Select from the following options: ACE, ACE-Lite+DVM, ACE-Lite, AXI4, AXI4-Lite.
User Width	User signal (i.e., AWUSER) width. 1-32

Table 5-5 AXI4*_Stub Parameters

5.4.1 AXI4*_Stub Macros

Macro definitions for AXI4*_Stub are provided in the following files:

$MAXSIM_PROTOCOLS/AXI4/include/AXI4_Stub_Macros.h
$MAXSIM_ PROTOCOLS/AXI4/include/AXI4_Stub_CheckMacros.h

5.5 MxAXI4

This is a generic model of an AXI interconnect. This model supports up to 16 masters and 16 slaves. Each AXI4 slave port supports up to 4 independent memory regions which can be configured through the component parameters or via SD Memory Map Editor.

All ports have the same data width (see the Parameters table below for supported data widths). External bridges can be used to convert the data widths on different ports. Figure 5-5 shows an example system that uses a 3x2 MxAXI4.

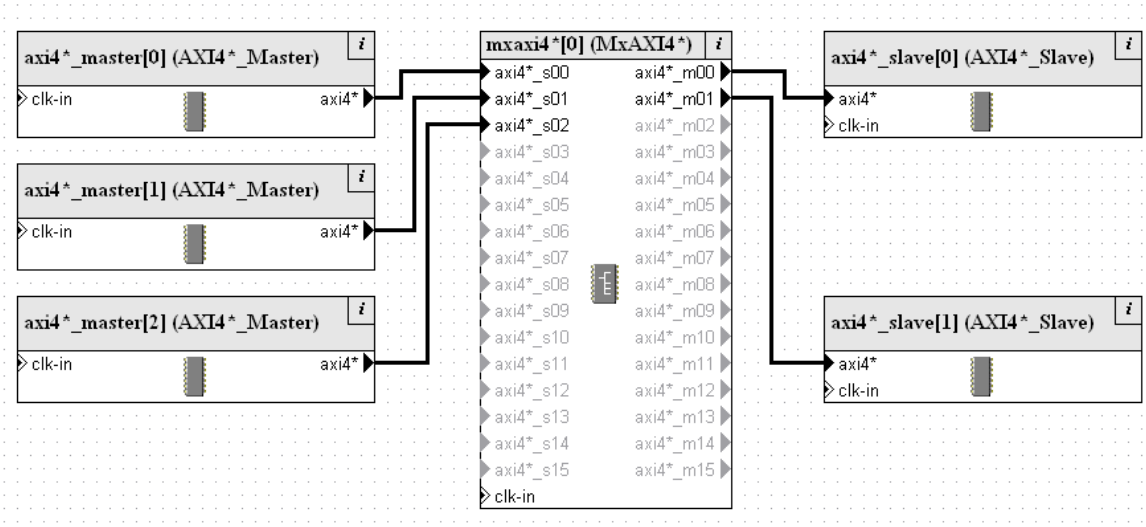


Figure 5-5 MxAXI4 with 3 masters and 2 slaves

The MxAXI4 component currently supports two protocol variants: AXI4 and ACE-Lite. You can configure this in SoC Designer Canvas by setting the Protocol Variant parameter.

The MxAXI4 passes transactions from the slave ports to the appropriate master ports based on the defined address map. It does not consider the transaction contents to determine how to accomplish this. So for example, the QoS (Quality of Service) ports are not used to prioritize the transactions. In addition, coherency information is not interpreted, but rather simply passed through the master port to be handled by the attached slave device.

5.5.1 MxAXI4 Limitations

The MxAXI4 does not include full support the AXI4 protocol. The following are some of its limitations:

1. It supports only one transaction at a time on any slave or master, whether the transaction is read or write.
2. It reports decode errors as errors, rather than returning a decode response using the R or B channels.

5.5.2 MxAXI4 Parameters

Table 5-6 below lists the MxAXI4 component parameters.

Name	*Description*
Address width	Width in bits of the address bus. Supported values are 32 to 63.
Data width	Data bus width. Applies to all ports. Supported widths are 32, 64, 128, 256, 512, and 1024.
Protocol Variant	Select between AXI4 protocol variants. Currently supports AXI4 and ACE-Lite.
s*XX*_name*Y*	The name for memory region *Y* on port *XX*. *Y*: 0 – 3. *XX*: 00 – 15.
s*XX*_size*Y*	The size for memory region *Y* on port *XX*. *Y*: 0 – 3. *XX*: 00 – 15.
s*XX*_start*Y*	The start address for memory region *Y* on port *XX*. *Y*: 0 – 3. *XX*: 00 – 15.
Use MME	Use Memory Map Editor for configuring memory regions. If set to false, component parameters are used instead.

Table 5-6 MxAXI4 Parameters

5.6 AXI4ToAXIv2

This is a generic model that converts AXI4 traffic into AXIv2 traffic. Figure 5-6 shows the component.

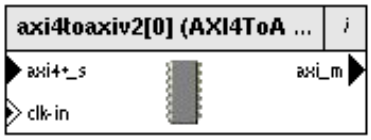


Figure 5-6 AXI4ToAXIv2 component

The AXI4ToAXIv2 currently supports only the AXI4 and ACE-Lite protocol variantes and does not cover ACE-Lite+DVM or ACE. Use this component to attach a legacy AXIv2 component to an AXI4 component.

The AXI4ToAXIv2 component handles the AXI4 to AXIv2 differences as follows:

- It splits any INCR bursts longer than 16 beats into multiple transactions.
- It generates the WID which is missing from AXI4 but is required by AXIv2.
- It handles AXIv2 early write response by delaying it until the address has been accepted.

The AXI4ToAXIv2 supports both AXI4 and the ACE-Lite protocol. Use the “Protocol Variant” parameter to select between the two modes.

In ACE-Lite mode it accepts all ACE-Lite transactions as follows (see the ARM AMBA AXI and ACE Protocol Specification for details):

- All types of reads and writes are treated as ordinary reads and writes.
- All cache maintenance operations are responded to immediately with an OK status.
- All barriers are responded to immediately with an OK status.

5.6.1 AXI4ToAXIv2 Limitations and Implementation Notes

The component currently does not support the following:

- Address widths greater than 32 bits.

  Note: For 40-bit address support, the AXI4ToAXIv2 emits a warning but allows the connection. However, the upper 8 bits must be zero; non-zero values in the upper 8 bits cause an error.
- USER signals greater than 32 bits (causes a warning).
- Non-zero values in the AxQOS or AxREGION signals (causes a warning).

5.6.2 AXI4ToAXIv2 Parameters

Table 5-7 below lists the AXI4ToAXIv2 component parameters:

Name	*Description*
Address Width	Supports 32-bits
Data Width	Width in bits of the data bus. Allowed values are 32, 64, and 128.
Protocol Variant	Currently supports AXI4 and ACE-Lite.
User Width	User signal (i.e., AWUSER) width. 1-32

Table 5-7 AXI4ToAXIv2 Component Parameters

5.7 AXIv2ToAXI4

This is a generic model that converts AXIv2 traffic into AXI4 traffic. Figure 5-7 shows the component.

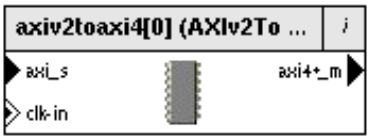


Figure 5-7 AXIv2ToAXI4 component

The AXIv2ToAXI4 can be connected to any of the AXI4 protocol variants (AXI4, ACE-Lite, ACE-Lite+DVM, and ACE). However, it generates only traffic that is consistent with the AXI3 protocol. In other words, it does not use any of the new AXI4/ACE signaling. These are treated as zero.

5.7.1 AXIv2ToAXI4 Limitations and Implementation Notes

The AXIv2ToAXI4 component currently does not support the following:

1. Address widths greater than 32-bits (causes an error).
2. USER signals greater than 32-bits (causes a warning).

5.7.2 AXIv2ToAXI4 Parameters

Table 5-8 below lists the AXIv2ToAXI4 component parameters:

Name	*Description*
Address Width	Supports 32-bits
Data Width	Width in bits of the data bus. Allowed values are 32, 64, and 128.
Protocol Variant	AXI4, ACE-Lite, ACE-Lite+DVM, and ACE
User Width	User signal (i.e., AWUSER) width. 1-32
AXI4* Domain	The component sets the AXI4* domain field in all forwarded transactions based on this value. Available settings are: • NON_SHAREABLE • INNER_SHAREABLE • OUTER_SHAREABLE • SYSTEM Refer to the ARM AMBA 4 Specification for details.

Table 5-8 AXIv2ToAXI4 Component Parameters

5.8 AXI4ToAXI4

This is a generic model that converts AXI traffic between different AXI4 variants. Use the "Protocol Variant" parameter to select the desired protocol variant separately for the master and the slave ports.

The AXI4ToAXI4 currently supports the following conversions:

Slave Side	*Master Side*
AXI4	AXI4
AXI4	ACE-Lite+DVM
ACE-Lite	AXI4

Notes:

When converting *from* AXI4, signals that are not provided by AXI4 are tied to 0 on the target side. Similarly, when converting *to* AXI4, signals that are not implemented in AXI4 are ignored.

The system in which the AXI4ToAXI4 component is used must be configured so that components connected to the AXI4ToAXI4 do *not* issue transactions that are not supported by the target protocol.

The CCI-400 can be connected to the NIC-400 via the AXI4ToAXI4. In this case, the CCI-400 should be configured for connection to AXI4 master and slaves as described in the *CCI-400 Integration Manual*.

5.8.1 AXI4ToAXI4 Parameters

Table 5-9 below lists the AXI4ToAXI4 component parameters:

Name	*Description*
Address Width	Supports 32-bits
Data Width	Width in bits of the data bus. Allowed values are 32, 64, 128, 256, 512, and 1024.
Master Protocol Variant	Currently supports AXI4 and ACE-Lite+DVM.
Slave Protocol Variant	Currently supports AXI4 and ACE-Lite.
User Width	User signal (i.e., AWUSER) width (1-32).

Table 5-9 AXI4ToAXI4 Component Parameters

5.9 Implementation Note: Address Width Parameter

The Address Width parameter allows you to create a system in which a mismatch exists between the address widths on a connection from a master to a slave device. This type of mismatch is allowed by the AXI3 and AXI4 protocols (Section A10.3.1 of the *AMBA AXI and ACE Protocol Specification* (AXI3, AXI4, and AXI4-Lite)).

If an address mismatch is detected, SoC Designer Simulator prints a warning when the system is loaded.

If the address field (AxADDR) of the slave is:

- Wider than the master - Zeros are used for the extra bits.
- Narrower than the master - The address bits beyond the width of the slave are not used by the slave.

In addition, SoC Designer Simulator watches the extra bits from the master and reports a warning if they are ever driven to non-zero values (indicating that the master is attempting to address a location that is not supported by the slave). To eliminate this warning, adjust the Address Width parameter for the master or slave so that they match.

5.10 Disk Backed Memory Functionality

The AXI4*_Mem component allows systems to model very large memories (several GBs) using a feature called *Disk Backed Memory*. This feature helps contain SoC Designer memory usage within the limits imposed by the operating systems.

Disk Backed Memory achieves this by swapping the contents of the simulated memory to the disk when the simulated memory exceeds a certain limit. The content swapped to the disk is usually the oldest content, while the most recent content remains in memory.

As general guidelines, this feature should be enabled if the memory being modeled is larger than 2GB.

***Note**: Be aware that enabling disk backed memory may affect performance. By default, the functionality is disabled.*

This feature is enabled and controlled using the following parameters:

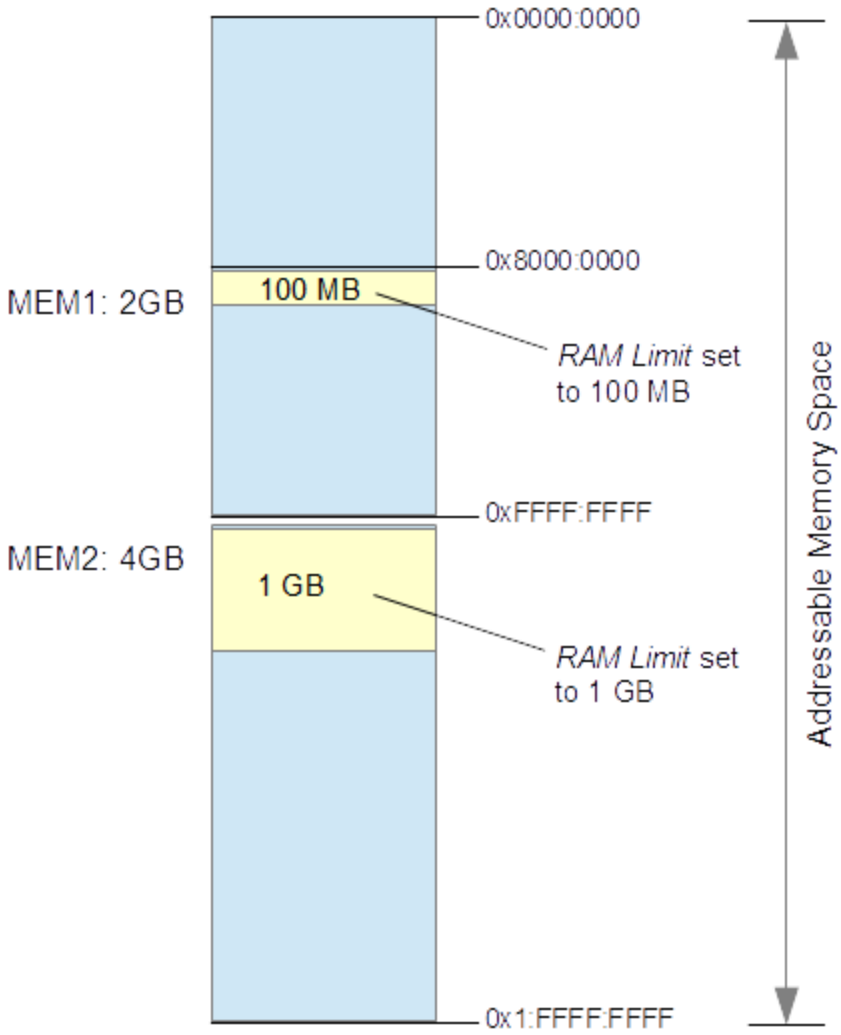
- **Disk Backed Memory** – This parameter enables/disables Disk Backed Memory functionality. It is disabled by default.
- **Disk Backed Memory Size (Bytes)** – This parameter should match the size of addressable space of the memory instance. For example, if an AXI4*_Mem instance called **Mem1** is required to address 0x8000:0000 to 0xFFFF:FFFF (2GB), then this parameter should be set to 0x80000000 (0xFFFF:FFFF minus 0x8000:0000) for **Mem1**.

  The default is 100 GB.
- **Disk Backed Memory-RAM Limit (MB)** – This parameter determines how much physical RAM each instance consumes before swapping starts. The default setting of 250 MB should work well for most users; however, if you want to fine-tune these settings, allocate more RAM to the most frequently-accessed memory instance.

  ***Note**: The sum of all RAM Limit settings should not exceed 4GB (this is the current implementation limit). This does not limit the amount of memory being modeled; rather, it limits the RAM consumption of SoC Designer.*

5.10.1 Disk Backed Memory Use Case

The following figure shows a sample use case in which Disk Backed Memory is enabled. In this use case, there are two instances of modeled memory.



The AXI4_mem component limits the RAM consumption on each instance to the value specified by the RAM Limit parameter. In this example, the total RAM used by the disk memory instances is 100 MB + 1 GB. Because MEM2 is more frequently accessed than MEM1 in this use case, the RAM Limit parameter for MEM2 has been set significantly higher (1 GB) than the RAM Limit parameter for MEM1 (100 MB). When the limit set for each instance is reached, the oldest data is swapped to disk.

6 Probes

Table 6-1 describes the simulation probes that are included in the AXI4 Protocol Bundle. These are described further in the following sections:

Name	*Description*
Tracer	Enables tracing of AXI signals on an AXI4 connection. Traced signals can be viewed in the SoC Designer simulator waveform window.
BreakPoint	Transaction breakpoint on an AXI connection.
Profiler	Profiles AXI transactions. Profiled data can be viewed in the SoC Designer simulator profiler window.
Monitor	View the activity over the connection for each cycle.

Table 6-1 AXI4 Probes

6.1 Tracer Probe

This probe allows tracing of AXI signals. Use the SoC Designer Waveform window to see the traced signals. To add a tracer probe, right-click on an AXI4 connection and select "Enable/Disable Tracing". This displays the Tracer Properties dialog shown in Figure 6-1:

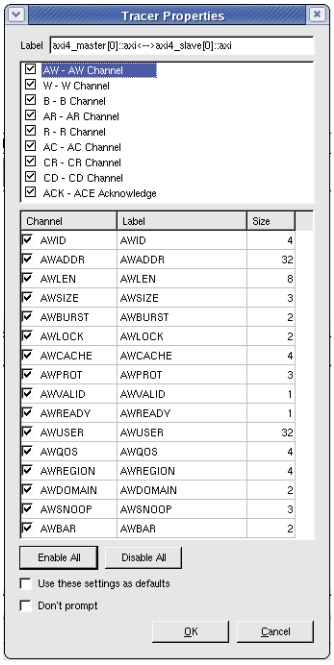


Figure 6-1 Tracer Properties

By default, all signals are traced. To disable tracing of certain signals, use the checkboxes located on the left side of the signal.

Note: AXI4 tracer properties show all AMBA4 AXI signals including the ACE channel signals regardless of whether or not the additional signals are used by the components. Disable unused signals or channels by deselecting the check-boxes next to the Channel/Signal name in the Tracer Properties dialog.

6.2 Breakpoint Probe

To insert a breakpoint probe, either:

- Double-click on the connection, or
- Right-click on the connection and select "Insert/Remove Breakpoint" from the context menu.

By default, the breakpoint is activated and breaks on any active AXI transaction across the connection.

To specify more specific breakpoint conditions:

1. Right-click on the connection.
2. Select "Edit Breakpoint Properties."

This displays the Breakpoint Condition dialog, shown in Figure 6-2:

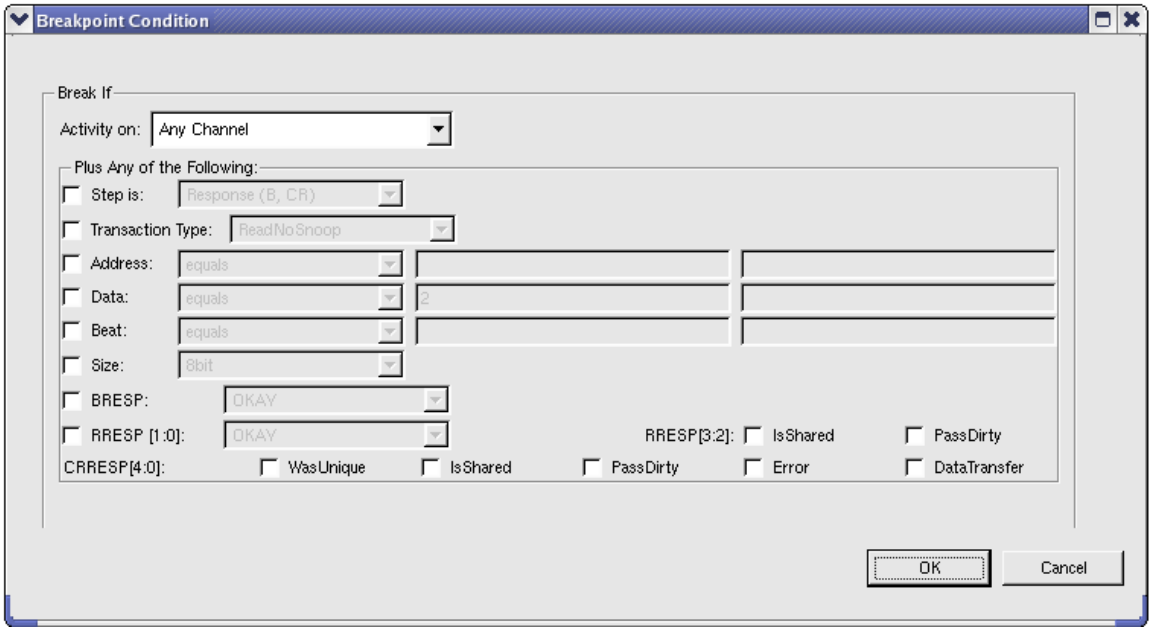


Figure 6-2 Breakpoint Condition Dialog

Channel and Signal options differ depending on the protocol variant of the connection. Options that are not available with a particular protocol variant appear grayed out. Additionally, certain options may be grayed out automatically based on your Channel and Step choices.

If you do not specify any options, SoC Designer generates a breakpoint for the VALID signals on all channels, plus RACK and WACK (if available in the protocol variant).

Table 6-2 describes the breakpoint condition options.

Option	*Description*
Activity on:	Select Read Channels, Write Channels, Coherency Channels, or Any Channel.
Step is:	Select Address, Data, Response, or ACK channels.
Transaction Type:	Choices are channel-dependent and based on the permitted address control signal combinations for read, write, and snoop transactions.**
Address: Data: Beat:	Select to specify *equals*, *greater than*, *less than*, *not equals*, *less than or equals*, or *greater than or equals* plus a value that you enter, or *within* plus two values that you enter.
Size:	Specify 8bit, 16bit, 32bit, 64bit, or 128bit.
BRESP:* RRESP[1:0]: *	Choices are the AXI-defined responses for read and write transactions: OKAY, EXOKAY, SLVERR, AND DECERR.**
RRESP[3:2]: *	Choices are the AXI-defined RRESP read response bits: PassDirty and IsShared.**
CRRESP[4:0]: *	Choices are the AXI-defined Snoop response bit allocations: WasUnique, IsShared, PassDirty, Error, and DataTransfer.**

**Refer to Section 6.2.2 for important details on response breakpoints.*
*** Refer to the ARM AMBA 4 Specification for details.*

Table 6-2 AXI4 Breakpoint Options

Note: If certain breakpoint conditions are more complex than can be specified with the Breakpoint Condition dialog, use MxScript to generate your breakpoints. Refer to the *MxScript Reference Manual* for information.

6.2.1 Multiple-channel breakpoints

When you select multiple channels in the Activity and Step menus, the SoC Designer breakpoint logic defines the broadest possible breakpoint definition. In other words, it combines all of the conditions that are applicable to the choices made in the Breakpoint Condition dialog.

Consider the example shown in Figure 6-3:

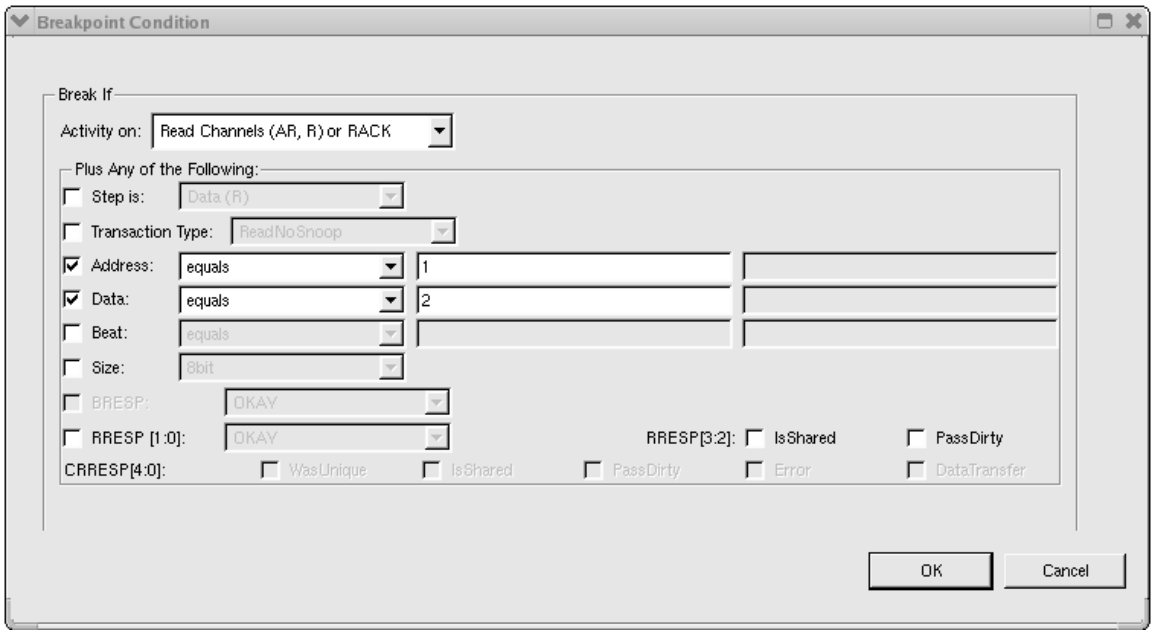


Figure 6-3 Multiple-channel example

The breakpoint generated for the configuration specified in Figure 6-3 is:

(((RVALID == 0x1) && (RDATA == 0x2)) || ((ARVALID == 0x1) && (ARADDR == 0x1)))

Because the breakpoint is restricted to channels AR and R, which cannot occur simulteneously, SoC Designer generates two breakpoints: one for RDATA and another for ARADDR.

Note: Because RACK is asserted with every transaction, SoC Designer does not include RACK as a breakpoint when additional breakpoints are specified in the "Plus Any of the Following" section of the dialog.

6.2.2 Response breakpoints for ACE and ACE-Lite+DVM

The additional RRESP bits for ACE (IsShared and PassDirty) are handled in the SoCD breakpoint logic by matching against one and a mask. Similarly, the additional CRRESP bits for ACE and ACE-Lite + DVM (WasUnique, IsShared, PassDirty, Error, and DataTransfer) are handled in the SoCD breakpoint logic by matching against one and a mask. As a result, these bits may only be compared against one (asserted) or x (don't care; not included in the mask). The option to break when these bits equal zero is not available.

The masking comparison is a breakpoint condition equal to: (SIGNAL & MASK) == MASK, where MASK is the bitwise OR (|) of the individual response bits.

For example, breaking on the CRRESP bits for IsShared and DataTransfer yields MASK = 9, because 9 is (8|1), because IsShared is CRRESP[3] and DataTransfer is CRRESP[0]. The breakpoint condition for CRRESP is (CRRESP & 0x9) == 0x9.

For RRESP, the MASK is enabled for the lower 2 bits if a comparison is made against OKAY through DECERR.

6.3 Profiling Probe

The Profiling Probe enables profiling on AXI4, ACE-Lite, ACE-Lite+DVM, and ACE connections.

To enable this probe:

1. Right-click on a connection.
2. From the context menu, select **Profiler > Enable**.
3. View the Profiler Probe dialog by right-clicking again and selecting **Profiler > Display**.

There are three separate streams available for profiling the set of AXI4 protocol variant connections:

Channels Usage: AXI events when channel is active (Valid is asserted on the channel). This captures:

- Cycle – the timestamp.
- Address – address being read or written at the cycle.
- Channel Used – one of the 5, 7, or 8 AXI4/ACE-Lite/ACE-Lite+DVM/ACE channels.

Latency: Latency for each operation type. This captures:

- Cycle – the timestamp.
- Latency – the length of time taken by an operation that finished on the cycle.
- Operation Type – the write, read, or coherency latency types: write trans, write-initial, write-burst, etc.

Address: Plots the accessed address location. This captures:

- Cycle – the timestamp.
- Address – the address being read or written at the cycle.
- Latency – the length of time taken by an operation that finished on the cycle.
- Operation Type – one of read, write, or coherency.

Browse the Profiling Manager to view the available choices for Y axis dimensions for the profiler plot.

Figure 6-4 shows a sample Profiling Probe dialog for the Channels Usage stream, with Channel Used on the Y axis:

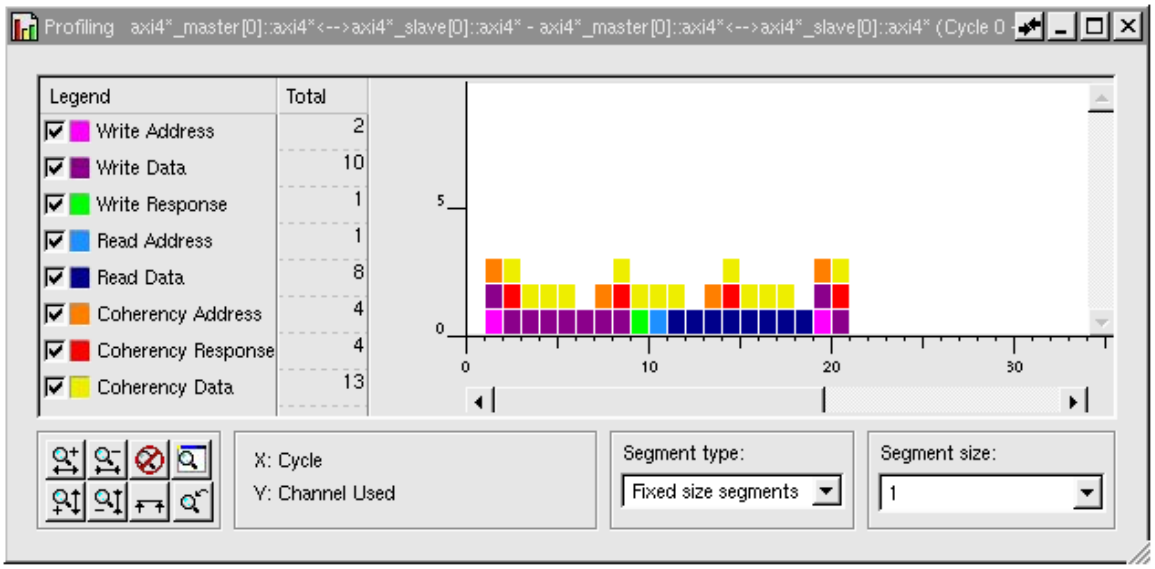


Figure 6-4 Profiling Dialog, Channels Usage

Figure 6-5 shows a sample Profiling Probe dialog for the Latency stream, with Latency used on the Y axis:

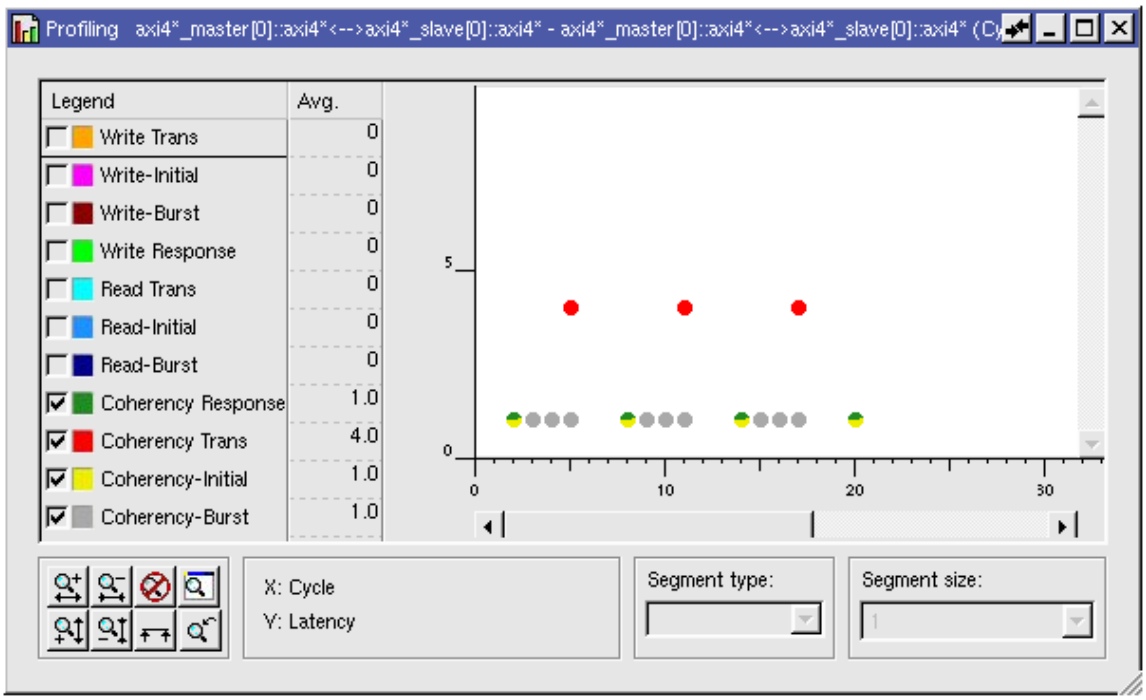


Figure 6-5 Profiling Dialog, Latency

Table 6-2 defines the channels profiled for the Channels Usage stream:

Channel Name…	*Is available on Protocol Variant…*
AR, R, AW, W, B	All AXI4/ACE variants.
AC and CR	ACE-Lite+DVM and ACE.
CE	ACE only.

Table 6-2 Channels Usage Stream Profiling Channels

Table 6-3 describes the operation types profiled on the Latency Stream. For descriptions of operation types, refer to the *SoC Designer User Guide*, section 4.9.7, Latency Profiling:

Operation Type…	*Is available on Protocol Variant…*
Write Trans, Write-Initial, Write-Burst, Write Response	All AXI4/ACE variants.
Read Trans, Read-Initial, Read-Burst	All AXI4/ACE variants.
Coherency Trans	ACE, latency of an entire transaction involving AC, CR, and CD.
Coherency Response	ACE and ACE-Lite+DVM, latency of the response involving AC and CR.
Coherency-Initial	ACE, the latency of the first coherency write involving AC and CD.
Coherency-Burst	ACE, the latency of remaining coherency writes involving AC and CD.

Table 6-3 Latency Stream Profiling Operation Types

Table 6-4 describes the operation types profiled on the Address Stream. For descriptions of operation types, refer to the *SoC Designer User Guide*, section 4.9.7, Latency Profiling:

Operation Type	*Protocol Variant and Description*
Write Trans, Write-Initial, Write-Burst, Write Response	All AXI4/ACE variants. Any operation involving the write channels AW, W, and B.
Read Trans, Read-Initial, Read-Burst	All AXI4/ACE variants. Any operation involving the read channels AR and R.
Coherency, ACE-Lite+ Trans	ACE, latency of an entire transaction involving AC, CR, and CD.

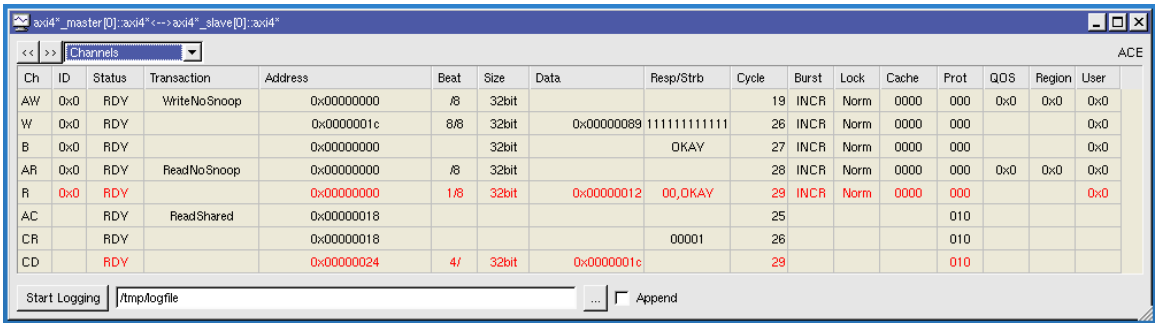
Table 6-4 Address Stream Profiling Operation Types

6.4 Monitor Probe

This probe enables monitoring of an AXI4 connection for each cycle. To enable this probe, right-click on a connection and select “Insert/Remove Monitor”.

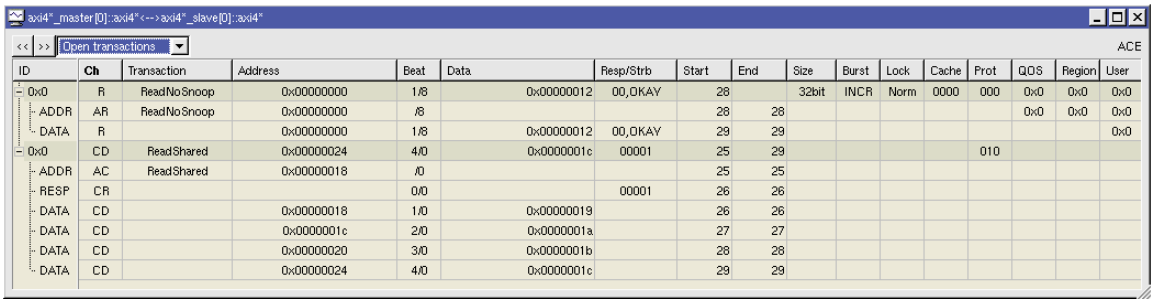
The default view shows the basic information for each of the 5 AXI channels. Use the “>>” button on the top left to expand the monitor window to show transaction details

Use the pulldown menu at the top left of the Monitor dialog to view the three available sets of data: Active Channels (Figure 6-4), Open transactions (Figure 6-5), and Closed transactions (Figure 6-6).



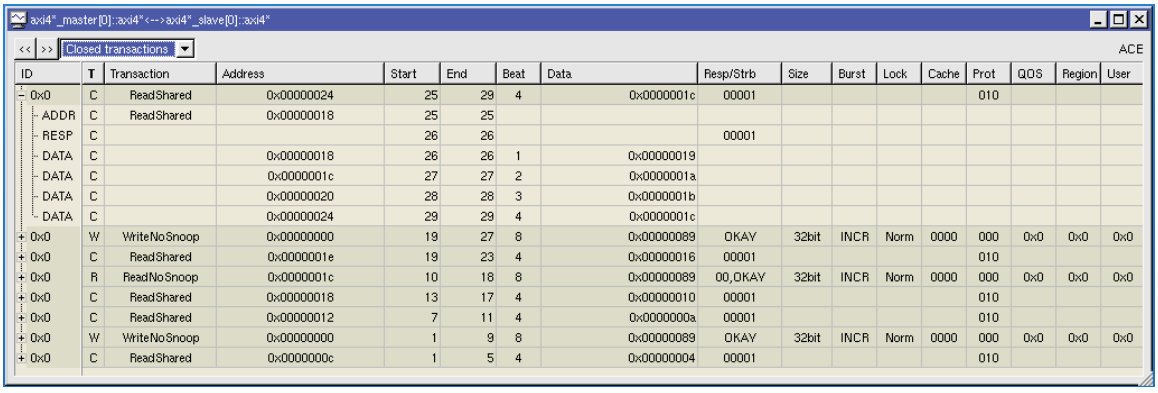
Ch	ID	Status	Transaction	Address	Beat	Size	Data	Resp/Strb	Cycle	Burst	Lock	Cache	Prot	QOS	Region	User
AW	0x0	RDY	WriteNoSnoop	0x00000000	/8	32bit			19	INCR	Norm	0000	000	0x0	0x0	0x0
W	0x0	RDY		0x0000001c	8/8	32bit	0x00000089	111111111111	26	INCR	Norm	0000	000			0x0
B	0x0	RDY		0x00000000		32bit		OKAY	27	INCR	Norm	0000	000			0x0
AR	0x0	RDY	ReadNoSnoop	0x00000000	/8	32bit			28	INCR	Norm	0000	000	0x0	0x0	0x0
R	0x0	RDY		0x00000000	1/8	32bit	0x00000012	00,OKAY	29	INCR	Norm	0000	000			0x0
AC		RDY	ReadShared	0x00000018					25				010			
CR		RDY		0x00000018				00001	26				010			
CD		RDY		0x00000024	4/	32bit	0x0000001c		29				010			

Figure 6-4 Monitor dialog, Active Channels view



ID	Ch	Transaction	Address	Beat	Data	Resp/Strb	Start	End	Size	Burst	Lock	Cache	Prot	QOS	Region	User
0x0	R	ReadNoSnoop	0x00000000	1/8	0x00000012	00,OKAY	28		32bit	INCR	Norm	0000	000	0x0	0x0	0x0
ADDR	AR	ReadNoSnoop	0x00000000	/8			28	28						0x0	0x0	0x0
DATA	R		0x00000000	1/8	0x00000012	00,OKAY	29	29								0x0
0x0	CD	ReadShared	0x00000024	4/0	0x0000001c	00001	25	29					010			
ADDR	AC	ReadShared	0x00000018	/0			25	25								
RESP	CR			0/0		00001	26	26								
DATA	CD		0x00000018	1/0	0x00000019		26	26								
DATA	CD		0x0000001c	2/0	0x0000001a		27	27								
DATA	CD		0x00000020	3/0	0x0000001b		28	28								
DATA	CD		0x00000024	4/0	0x0000001c		29	29								

Figure 6-5 Monitor dialog, Open Transactions view



ID	T	Transaction	Address	Start	End	Beat	Data	Resp/Strb	Size	Burst	Lock	Cache	Prot	QOS	Region	User
0x0	C	ReadShared	0x00000024	25	29	4	0x0000001c	00001					010			
ADDR	C	ReadShared	0x00000018	25	25											
RESP	C			26	26			00001								
DATA	C		0x00000018	26	26	1	0x00000019									
DATA	C		0x0000001c	27	27	2	0x0000001a									
DATA	C		0x00000020	28	28	3	0x0000001b									
DATA	C		0x00000024	29	29	4	0x0000001c									
0x0	W	WriteNoSnoop	0x00000000	19	27	8	0x00000089	OKAY	32bit	INCR	Norm	0000	000	0x0	0x0	0x0
0x0	C	ReadShared	0x0000001e	19	23	4	0x00000016	00001					010			
0x0	R	ReadNoSnoop	0x0000001c	10	18	8	0x00000089	00,OKAY	32bit	INCR	Norm	0000	000	0x0	0x0	0x0
0x0	C	ReadShared	0x00000018	13	17	4	0x00000010	00001					010			
0x0	C	ReadShared	0x00000012	7	11	4	0x0000000a	00001					010			
0x0	W	WriteNoSnoop	0x00000000	1	9	8	0x00000089	OKAY	32bit	INCR	Norm	0000	000	0x0	0x0	0x0
0x0	C	ReadShared	0x0000000c	1	5	4	0x00000004	00001					010			

Figure 6-6 Monitor dialog, Closed Transactions view

6.4.1 Dumping monitor contents

To dump the monitor contents to a file:

1. In the Monitor dialog, with Channels selected in the pulldown menu, click the “Start Logging” button (circled in Figure 6-7).

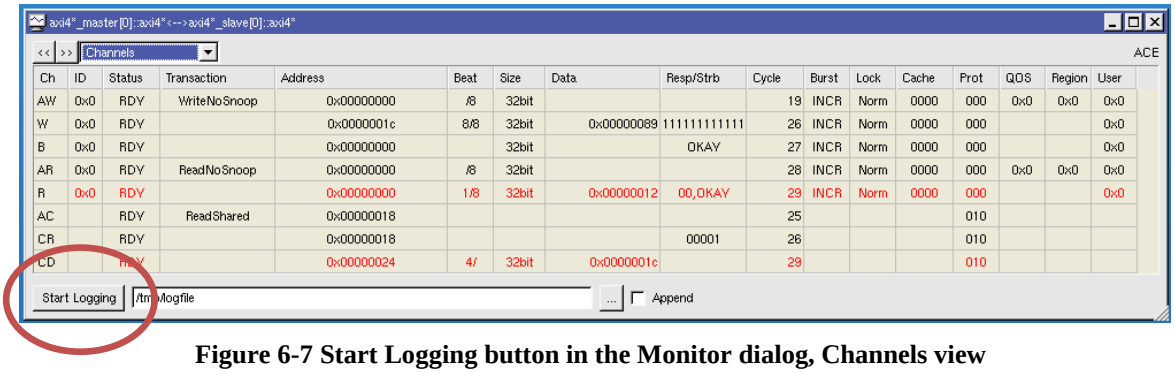


Figure 6-7 Start Logging button in the Monitor dialog, Channels view

2. Click the “**…**” button to specify the output file.

7 AXI4 Port Interfaces

AXI4 port classes are CASI implementations of the ARM AMBA4 AXI protocol, including ACE. The interfaces for AXI4 transactions are described in this chapter.

7.1 Port Classes

The port class header files are located under the *$MAXSIM_PROTOCOLS/AXI4/include* directory. These header files are needed for building SoC Designer components with AXI4 ports. All interface classes are under C++ namespace *casiaxi4*.

7.2 Channel Ports

AMBA4 AXI (ACE) uses up to eight independent channels that build up the overall transaction interface. Although the master initiates transactions, not all five channels are initiated from the master. Address (AR, AW) and write data (W) have the master-to-slave direction, whereas the read data (R) and the write response (B) are initiated from the slave. For this reason, CASI AXI4 utilizes the concept of sub-ports to represent each of the AXI channels.

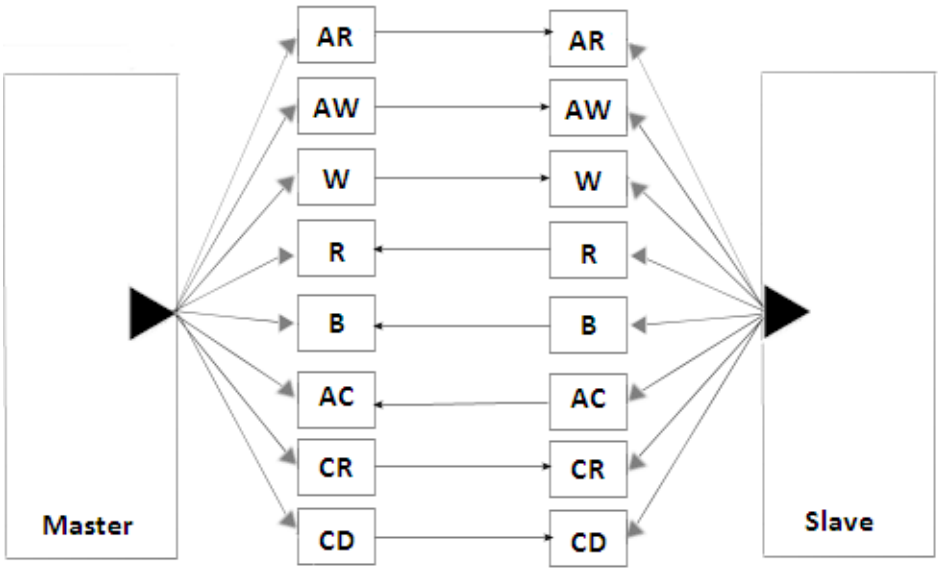


Figure 7-1 AXI Channels using Sub-ports

Using these channel ports encapsulated in the parent AXI4 master and slave ports, the model developer is able to control the communication at the AXI channel level.

Note: The channel ports are internal to the AXI4 port. They are not visible in SoC Designer Canvas or in the Simulator. The connections for the channel ports are established based on the connection of the parent AXI4 ports.

Figure 7-2 depicts where the internal ports reside for the master port (Figure 7-2, left) and slave port (Figure 7-2, right).

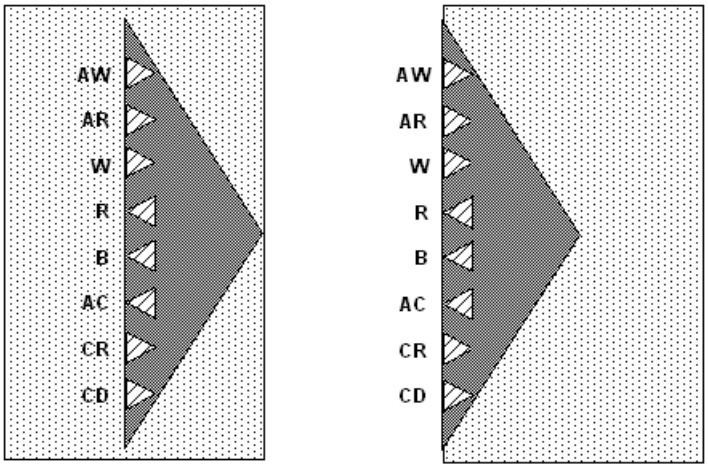


Figure 7-2 Master and Slave Ports showing Internal Channel Ports

7.3 Master Interface

7.3.1 AXI4_Master_Port

The AXI4_Master_Port class implements the parent port that encapsulates the eight AXI channel ports underneath, and provides the necessary APIs to the port owner for controlling the channel communication. AXI ports implementing the AXI master interface must derive off this class. It also defines methods used for setting AMBA4 AXI specific signals.

The location of this file is: `$MAXSIM_PROTOCOLS/AXI4/include/AXI4_Master_Port.h`.

7.3.2 AXI Sender and Receiver Ports for AXI Master Interface

`AXI4_Master_Port` instantiates the eight AXI channel ports. Channel ports are categorized as sender ports and receiver ports. For the AXI master interface, the sender ports (initiating channel) are the AR, AW, W, CR, and CD channels. The B, R, and AC channels are categorized as receiver ports.

`AXI4_Master_Port` serves as the interface for accessing each of the channels: it is not necessary or recommended for the owner to access the channels directly. `AXI4_Master_Port` provides the methods for setting and retrieving channel signal values. Signals are transferred over the channels via CASI `driveTransaction`/`notifyEvent` calls. The relationship between the AXI master port and the channel ports is depicted in the figure below. `set`*X* methods (*X* refers to the channel) are used for setting the channel signals, which occurs when `driveTransaction` is called for that channel.

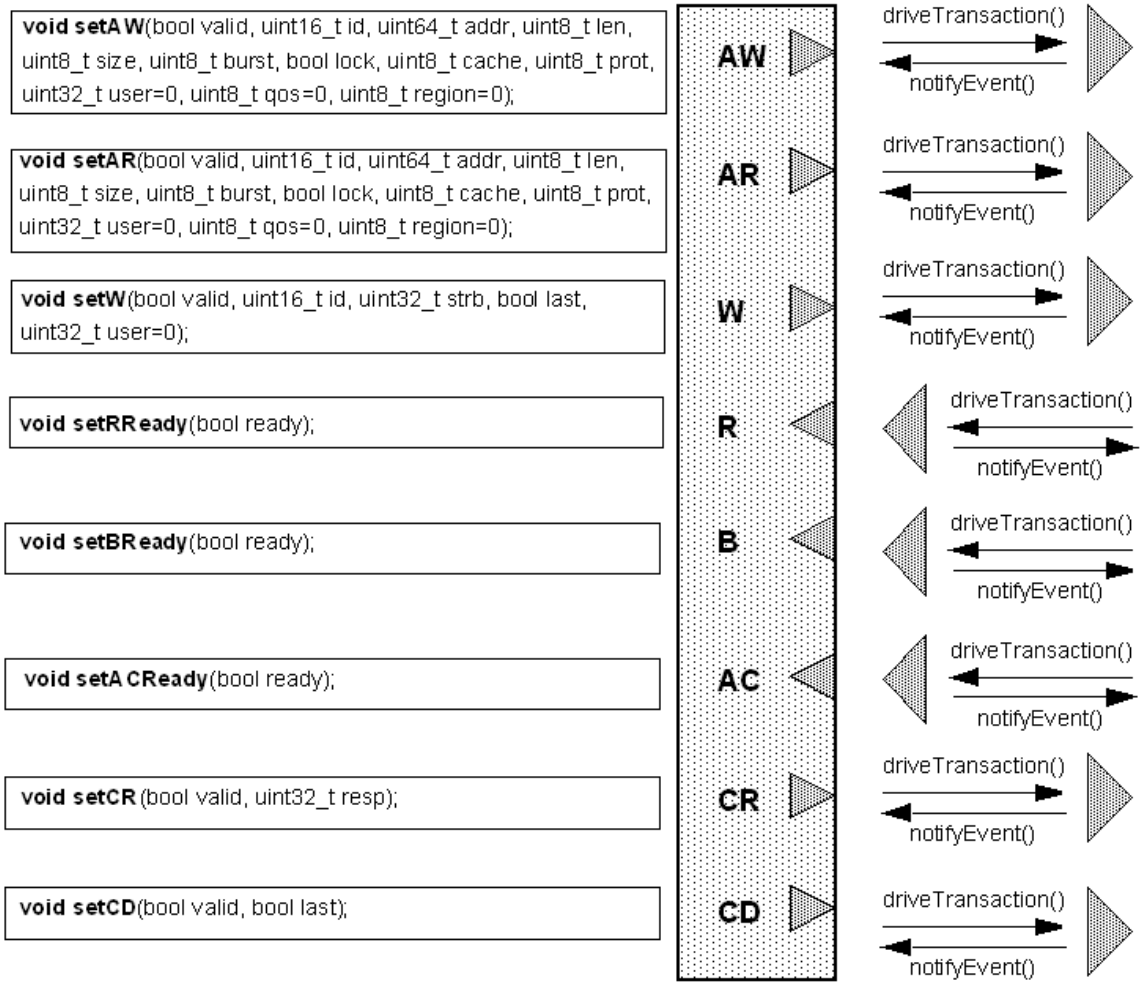


Figure 7-3 AXI Master Port and its Channel Ports

Note: `AXI4_Master_Port` implements CASI connect and disconnect methods, which handle the interconnection of all internal channel ports.

7.3.3 Methods for Setting Channel Signal Values

The following methods are provided for setting individual signals on the sender ports.

```
void setAW(bool valid, uint16_t id, uint64_t addr, uint8_t len, uint8_t size, uint8_t
burst, bool lock, uint8_t cache, uint8_t prot, uint32_t user=0, uint8_t qos=0, uint8_t
region=0);
void setAR(bool valid, uint16_t id, uint64_t addr, uint8_t len, uint8_t size, uint8_t
burst, bool lock, uint8_t cache, uint8_t prot, uint32_t user=0, uint8_t qos=0, uint8_t
region=0);
void setACReady(bool ready);
void setCR(bool valid, uint32_t resp);
void setCD(bool valid, bool last);
void setCDData(uint32_t data, uint8_t idx);
```

Call these methods only during the SoC Designer Update phase. The new values are buffered until the next Communicate phase, when they are sent out via `driveTransaction()` calls. The only time these methods can be called in the Communicate phase is when they are inside a `driveTransactionCB_X` call and signals need to be forwarded onto a different channel sender port. In this case, the channel sender port that receives the forwarded data requires that its `driveTransaction()` method also be called inside the `driveTransactionCB_X`.

For the receiver ports, the following methods are provided for responding with the AXI ready signal.

```
void setRReady(bool ready);
void setBReady(bool ready);
void setACReady(bool ready);
```

Normally, you call these methods during the Update phase. However, if you need to model combinatorial ready-on-valid behavior, call the above methods from the `driveTransactionCB_X` callback function. See Section 7.3.6 for information on handling combinatorial ready-on-valid channel handshake behavior.

The `clear()` method is useful for resetting the signal values. As with other methods for setting new values on the channel signals, this method must be called during the Update phase.

7.3.4 sendDrive

Call `sendDrive()` from the component's communicate method. This transfers the signal values across the channels via CASI `driveTransaction` method on each channel sender port.

7.3.5 Initialization and Reset

```
void init(uint32_t addrWidth, uint32_t dataWidth, uint32_t userWidth);
```

Call `init()` to initialize the port with the correct address and data bit-widths. The valid ranges are as follows:

- address bit-width – from 32 to 63
- data bit-width – between 8 and 1024 (powers of 2)
- user bit-width – 1 to 64 bits

This method should be called from the component's `init()`.

```
void reset();
```

To properly reset the channel port, you must call `reset()`. This should be called from the component's `reset()` routine.

7.3.6 Changing the protocol variant for AXI4_Master_Port

For an example of how to change the protocol variant used for the AXI4_Master_Port, refer to the example code in the `AXI_Master::init()` method defined in `$MAXSIM_HOME/Protocols/AXI4/src/AXI4_Master/AXI4_Master.cpp`. This example shows how to adjust the Protocol ID and Name based on the current value of the Protocol ID parameter.

7.3.7 Supporting Combinatorial Ready-On-Valid

`AXI4_Master_Port` provides the following interface for supporting combinatorial ready-on-valid behavior:

```
virtual void driveTransactionCB_R() {};
virtual void driveTransactionCB_B() {};
virtual void driveTransactionCB_AC() {};
```

It is not mandatory for the port owner to implement the above methods; however, they must be implemented if the master component needs to model combinatorial ready-on-valid behavior. The above methods are called during the Communicate phase when the slave's R, B and/or AC channel is initiated. Setting the AXI ready signal high in these methods means that the ready signal goes high combinatorially, based on the valid signal in the same cycle.

7.3.8 Master Port Supporting Full System Coherent Memory Views

To support full system coherent memory views, use the port class called AXI4_Master_Backtrace_Port, which drives directly from AXI4_Master_Port. This port class has the following additional function compared to its parent port:

```
eslapi::CASIStatus debugTransactionBackTrace(eslapi::CASITransactionInfo
*info);.
```

This function resides `Protocols/AXI4/ports/include/AXI4_Master_Backtrace_Port.h`.

Refer to the *SoC Designer User Guide* for more information about full system coherent memory views, and to the *MxScript Reference Manual* for details about how CADIMemWrite and CADIMemRead support full system coherent memory views.

7.4 Slave Interface

7.4.1 AXI4_Slave_Port

The `AXI4_Slave_Port` class, similar to `AXI4_Master_Port`, implements the parent transaction slave port that encapsulates the eight AXI channel ports underneath, and provides the necessary APIs to the port owner for controlling the channel communication. AXI ports implementing the AXI slave interface must derive off this class.

The location of this file is: $MAXSIM_PROTOCOLS/AXI4/include/AXI4_Slave_Port.h.

7.4.2 AXI Sender and Receiver Ports for AXI Slave Interface

`AXI4_Slave_Port` instantiates the five AXI channel ports. Channel directions are opposite to those used in `AXI4_Master_Port`: AR, AW, and W channels serve as receiver ports, and B and R channels serve as sender ports. Methods are provided for accessing the channel data.

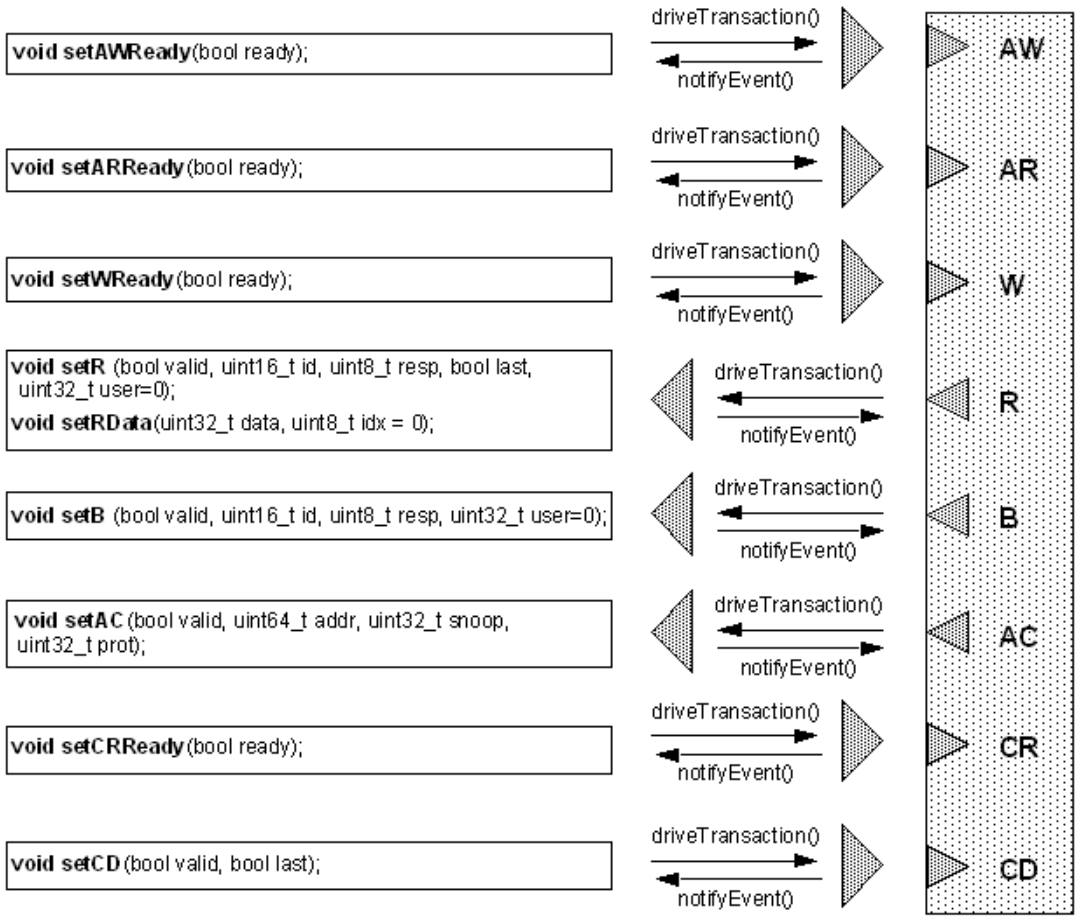


Figure 7-4 AXI Slave Port and its Channel Ports

7.4.3 Methods for Setting Channel Signal Values

The following methods are provided for setting individual signals on the sender ports.

```
void setR (bool valid, uint16_t id, uint8_t resp, bool last, uint32_t user=0);
void setRData(uint32_t data, uint8_t idx = 0);
void setB (bool valid, uint16_t id, uint8_t resp, uint32_t user=0);
void setAC(bool valid, uint64_t addr, uint32_t snoop, uint32_t prot);
```

Call these methods only during the SoC Designer update phase. The new values are buffered until the next Communicate phase, when they are sent out via `driveTransaction()` calls.

The `clear()` method is useful for resetting the signal values. As with other methods for setting new values on the channel signals, this method must be called during the Update phase.

For the receiver ports, the following methods are provided for responding with the AXI ready signal.

```
void setARReady(bool ready);
void setAWReady(bool ready);
void setWReady(bool ready);
void setCRReady(bool ready);
void setCDReady(bool ready);
```

These methods are the same as the AXI4_Master_Port's `setXReady` methods in terms of their usage.

Additional callbacks are added in AXI4_Slave_Port for ACE:

```
virtual void driveTransactionCB_CR() {};
virtual void driveTransactionCB_CD() {};
virtual void driveSignalCB_RACK() {};
virtual void driveSignalCB_WACK() {};
```

The callbacks for the CR/CD channels can be used to handle combinatorial ready-on-valid behavior (same as the AW/AR/W callbacks defined in `AXI4_Slave_Port`). For detecting changes on RACK/WACK, driveSignalCB_* callbacks are provided.

7.4.4 sendDrive

Call `sendDrive()` only from the component's `communicate` method. This transfers the signal values across the channels via CASI `driveTransaction` method for each sender channel port.

7.4.5 Initialization and Reset

```
void init(uint32_t addrWidth, uint32_t dataWidth, uint32_t userWidth);
```

Call `init()` to initialize the port with the correct address and data bit-widths. The valid ranges are as follows:

- address bit-width – from 32 to 63
- data bit-width – between 8 and 1024 (powers of 2)
- user bit-width – 1 to 64 bits

This method should be called from the component's `init()` routine.

```
void reset();
```

To properly reset the channel ports, you must call `reset()`. This should be called from the component's `reset()` routine.

7.4.6 Changing the protocol variant for AXI4_Slave_Port

For an example of how to change the protocol variant used for the AXI4_Slave_Port, refer to the example code in the `AXI_Slave::init()` method defined in `$MAXSIM_HOME/Protocols/AXI4/src/AXI4_Slave/AXI4_Slave.cpp`. This example shows how to adjust the Protocol ID and Name based on the current value of the Protocol ID parameter.

7.4.7 Slave Port Supporting Full System Coherent Memory Views

To support full system coherent memory views, use the port class AXI4_Slave_Backtrace_Port, which derives directly from AXI4_Slave_Port. This port class has the following additional function compared to its parent port:

```
eslapi::CASIStatus debugTransactionBackTrace(eslapi::CASITransactionInfo
*info);.
```

This function resides `Protocols/AXI4/ports/include/ AXI4_Slave_Backtrace_Port.h`.

Refer to the *SoC Designer User Guide* for more information about full system coherent memory views, and to the *MxScript Reference Manual* for details about how CADIMemWrite and CADIMemRead support full system coherent memory views.

7.4.8 Slave Port Supporting Fast Debug Access

Fast debug access is a form of debug access which ignores the BUS width. This is mainly used for loading larger application images into CPUs.

To support fast debug access, the AXI4 slave interface provides a fast_debug_access_if interface and introduces a new port class called AXI4_Slave_Port_fda. This derives directly from AXI4_Slave_Port.

The following class implements the fast debug access function:

```
eslapi::CASIStatus debugTransaction(eslapi::CASIDebugTransactionInfo* info);
```

7.4.9 Slave Port Supporting Full System Coherent Memory Views Plus Fast Debug Access

The AXI4 slave interface provides a port class that supports both coherent memory views (described in the previous section) and fast debug access: AXI4_Slave_Backtrace_Port_fda, which derives from AXI4_Slave_Port_fda.

7.5 Note on Using Large AXI ID Widths

AXI4 master and slave channel access methods have a limit of 16 bits for accessing the AXI ID field. If your design is using wider ID widths, you can overload the AXI4 `set` methods as shown in the following code sample:

```
class MyLargeIDWidthMasterPort : public AXI4_Master_Port {
public:
   // overloaded setAW method
   void setAW(bool valid, uint64_t id, uint64_t addr, uint8_t len, uint8_t
size, uint8_t burst, bool lock, uint8_t cache, uint8_t prot, uint32_t user=0)
   {
        AXI4_Master_Port::setAW(valid, id, addr, len, size, burst, lock, cache,
prot, user);
      setSig(AW_ID, id);
   }
};
```

In the code sample above, `AXI4_Master_Port::setAW()` is used to set the values for fields other than the ID. Then, a `setSig()` is called for setting the ID value, which can be larger than 16 bits. An example for overloading the `setAW` method is shown here; you can take a similar approach for overloading the `set` method for other channels.

7.6 Examples

Pseudo-code examples are presented in this chapter for AXI master and slave using AXI4 ports.

7.6.1 AXI Master Port

Derive the port from `AXI4_Master_Port` class, and inherit `driveTransactionCB_X` methods for supporting ready-on-valid behavior, as shown in the following example:

```
#ifndef MYAXIMASTERPORT_H
#define MYAXIMASTERPORT_H

#include "AXI4_Master_Port.h"
#include "AXI4_Receiver_Port.h"

class AXI_Master;

class MyAxiMasterPort : public casiaxi4::AXI4_Master_Port
{
public:
    MyAxiMasterPort(CASIModule* owner, std::string name);

    virtual void driveTransactionCB_R();
    virtual void driveTransactionCB_B();
    virtual void driveTransactionCB_AC();

private:
    AXI_Master* master;
};

#endif
}
```

Note: `driveTransactionCB_X` need only be defined if the master port needs to model ready-on-valid behavior.

7.6.2 AXI Master Component

The requirements for driving output and signal values are as follows:

- Drive output values on the channel ports only during the Communicate phase
- Set signal values only during the Update phase

The following example shows the sequence of events for an AXI master's `communicate` and `update` methods.

```
void AXI_Master::communicate()
{
       // send out the channel signals from previous update
       AXI_TMaster->sendDrive();
}

void AXI_Master::update()
{
       AXI_TMaster->clear();

       // handle active channel requests and responses
       if (AXI_TMaster->getSig(R_VALID) && AXI_TMaster->getSig(R_READY))
       {
              // process RDATA
              My_RDATA = AXI_TMaster->getRData(i);
              …
       }
       if (AXI_TMaster->getSig(B_VALID) && AXI_TMaster->getSig(B_READY))
       {
              // done with a write transaction
       }
       if (AXI_TMaster->getSig(AR_VALID) && AXI_TMaster->getSig(AR_READY))
       {
              …
       }
       …

       // new channel requests
       AXI_TMaster->setAR(…);
       AXI_TMaster->setAW(…);
}
```

7.6.3 AXI Master Backtrace Port

Derive the port from AXI4_Master_Backtrace_Port class, and inherit debugTransactionBackTrace methods for supporting reverse debug transaction from slave port as shown in the following example:

```
#ifndef _axi4_TM_H_
#define _axi4_TM_H_

#include "maxsimCompatibility.h"
#include "AXI4_Master_Backtrace_Port.h"
#include "xactors/arm/include/carbon_arm_adaptor.h"
typedef CarbonDebugTransactionFunction CarbonBackwardDebugTransactionFunction;
class AXI4_S2T;
class axi4_TM : public casiaxi4::AXI4_Master_Backtrace_Port
{
```

```
private:
  AXI4_S2T* owner;
  CASIModule* owner_module;
public:
  axi4_TM(CASIModule* owner, AXI4_S2T* xtor, std::string name,
CarbonBackwardDebugTransactionFunction* backDebugFunc=NULL);
  virtual ~axi4_TM() {}


  virtual eslapi::CASIStatus
debugTransactionBackTrace(eslapi::CASITransactionInfo *info);
private:
  CarbonBackwardDebugTransactionFunction* backDebugCallback;
};
#endif
```

7.6.4 AXI Slave Port

Derive the port from the `AXI4_Slave_Port` class, and inherit `driveTransactionCB_X` methods for supporting ready-on-valid behavior, as shown in the following example:

```
# ifndef MYAXISLAVEPORT_H
#define MYAXISLAVEPORT_H

#include "AXI4_Slave_Port.h"
#include "AXI4_Receiver_Port.h"

class AXI_Slave;

class MyAxiSlavePort : public casiaxi4::AXI4_Slave_Port
{
public:
    MyAxiSlavePort(CASIModule* owner, std::string name);

    virtual void driveTransactionCB_AR();
    virtual void driveTransactionCB_AW();
    virtual void driveTransactionCB_W();
    virtual void driveTransactionCB_CR();
    virtual void driveTransactionCB_CD();
    virtual void driveSignalCB_RACK();
    virtual void driveSignalCB_WACK();

private:
    AXI_Slave* slave;
};

#endif
```

The following example illustrates sample `driveTransactionCB_X` method implementations:

```
void MyAxiSlavePort::driveTransactionCB_AR()
{
  if (getSig(AR_VALID))
  {
    this->setARReady(1);
  }
}
void MyAxiSlavePort::driveTransactionCB_AW()
{
```

```
  if (getSig(AW_VALID))
  {
    this->setAWReady(1);
  }
}
void MyAxiSlavePort::driveTransactionCB_W()
{
  if (getSig(W_VALID))
  {
    this->setWReady(1);
  }
}
void MyAxiSlavePort::driveTransactionCB_CR()
{
  if (getSig(CR_VALID))
    this->setCRReady(1);
}
void MyAxiSlavePort::driveTransactionCB_CD()
{
  if (getSig(CD_VALID))
    this->setCDReady(1);
}
void MyAxiSlavePort::driveSignalCB_RACK()
{
  owner->forwardRACK(getSig(RACK));
}
void MyAxiSlavePort::driveSignalCB_WACK()
{
  owner->forwardWACK(getSig(WACK));}
```

7.6.5 AXI Slave Component

Modeling requirements for an AXI slave component are the same as those for an AXI master component: channel communication, or driving out-of-channel signal values during the Communicate phase, and sequential logic and setting of new signal values during the Update phase.

```
void AXI_Slave::communicate()
{
       AXI_TSlave->sendDrive();
}

void AXI_Slave::update()
{
       // clear signals
       AXI_TSlave->setR(false, 0, 0, 0);
       AXI_TSlave->setB(false, 0, 0);
       AXI_TSlave->setAWReady(false);
       AXI_TSlave->setARReady(false);
       AXI_TSlave->setWReady(false);

       // handle active channel requests
       if (AXI_TSlave->getSig(AR_VALID) && AXI_TSlave->getSig(AR_READY))
       {
             …
       }

       // handle new channel requests
```

```
        AXI_TSlave->setR(true, it->first, 0, isLastBeat);
}
```

7.6.6 AXI4 Slave Backtrace Port

Derive the port from the AXI4_Slave_Backtrace_Port class, and inherit the debugTransactionBackTrace methods for supporting reverse debug transaction from the slave port as shown in the following example:

```
#ifndef __AXI4_T2S_TS_H__
#define __AXI4_T2S_TS_H__
#include "AXI4_Slave_Backtrace_Port.h"
#include "AXI4_Receiver_Port.h"
#include <string>
#include <vector>
class AXI4_T2S;
class AXI4_T2S_TS : public casiaxi4::AXI4_Slave_Backtrace_Port
{
public:
  AXI4_T2S_TS(CASIModule* _owner, AXI4_T2S* xtor, std::string name);
  virtual ~AXI4_T2S_TS() {}
  AXI4_T2S* owner;
  virtual eslapi::CASIStatus debugTransaction(eslapi::CASITransactionInfo
*info);
};

#endif
```

7.6.7 AXI4 Slave Backtrace Port with Fast Debug Access

This port supports both backtrace () and fast debug access. To support reverse debug transactions, drive the port from the AXI4_Slave_Backtrace_Port_fda class and inherit debugTransactionBackTrace(). To support fast debug access, implement debugTransaction().

For example:

```
        #ifndef __AXI4_T2S_TS_H__
        #define       __AXI4_T2S_TS_H__

        #include "AXI4_Slave_Backtrace_Port.h"
        #include "AXI4_Receiver_Port.h"

        #include <string>
#include <vector>
class AXI4_T2S_TS : public casiaxi4::AXI4_Slave_Backtrace_Port_fda
{
public:
  AXI4_T2S_TS(CASIModule* _owner, AXI4_T2S* xtor, std::string name);
  virtual ~AXI4_T2S_TS() {}
  AXI4_T2S* owner;
```

```
virtual eslapi::CASIStatus
debugTransaction(eslapi::CASIDebugTransactionInfo *info);
}
       #endif
```

8 Component Wizard

The SoC Designer component wizard supports generation of AXI4 master and slave ports.

Note: Refer to the *SoC Designer User Guide* for general information regarding the Component Wizard.

8.1 Generating AXI4 Ports

To generate a model with AXI4 ports:

1. Launch the component wizard from SoC Designer Canvas and proceed to the port definition step.
2. Click **New** to create a new port, and select the desired AXI port type from the port type drop-down list, as shown in Figure 8-1:

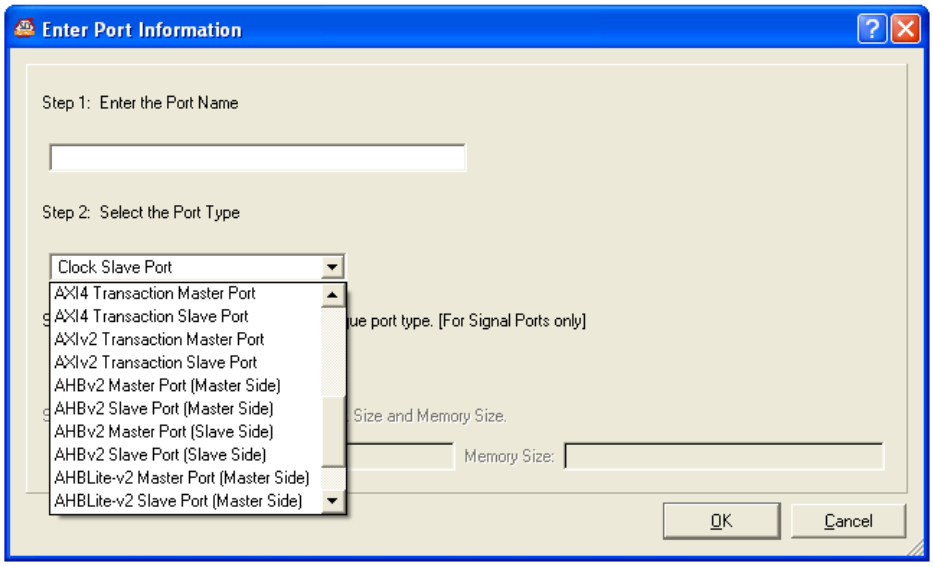


Figure 8-1 AXI Port Selection

This generates a .cpp file and a .h file for each AXI port that was selected.