OCP International Partnership

Open Core Protocol Specification

Release 3.0
OCP-IP Confidential

Open Core Protocol Specification

Document Revision 1.0

Contents

Introduction

The Open Core Protocol™ (OCP™) delivers the only non-proprietary, openly licensed, core-centric protocol that comprehensively describes the system-level integration requirements of intellectual property (IP) cores.

While other bus and component interfaces address only the data flow aspects of core communications, the OCP unifies all inter-core communications, including sideband control and test harness signals. OCP's synchronous unidirectional signaling produces simplified core implementation, integration, and timing analysis.

OCP eliminates the task of repeatedly defining, verifying, documenting, and supporting proprietary interface protocols. The OCP readily adapts to support new core capabilities while limiting test suite modifications for core upgrades.

Clearly delineated design boundaries enable cores to be designed independently of other system cores yielding definitive, reusable IP cores with reusable verification and test suites.

Any on-chip interconnect can be interfaced to the OCP rendering it appropriate for many forms of on-chip communications:

- Dedicated peer-to-peer communications, as in many pipelined signal processing applications such as MPEG2 decoding.
- Simple slave-only applications such as slow peripheral interfaces.
- High-performance, latency-sensitive, multi-threaded applications, such as multi-bank DRAM architectures.

The OCP supports very high performance data transfer models ranging from simple request-grants through pipelined and multi-threaded objects. Higher complexity SOC communication models are supported using thread identifiers to manage out-of-order completion of multiple concurrent transfer sequences.

The CoreCreator™ tool automates the tasks of building, simulating, verifying and packaging OCP-compatible cores. IP core products can be fully "componentized" by consolidating core models, timing parameters, synthesis scripts, verification suites, and test vectors in accordance with the *OCP Specification*. CoreCreator does not constrain the user to either a specific methodology or design tool.

Support

The *OCP Specification* is maintained by the Open Core Protocol International Partnership (OCP-IP™), a trade organization solely dedicated to OCP, supporting products and services. For all technical support inquiries, please contact techsupport@ocpip.org. For any other information or comments, please contact admin@ocpip.org.

Changes for Version 3.0

Changes for Version 3.0 include:

- Coherence Extensions.
- Updated semantics for the write response enable.
- Support for new sideband signals that enable the master to control the connection state of the interface based upon the input of both master and slave. The new MConnect, SConnect, SWait and ConnectCap signals implement the connection protocol and the connection parameter configures these signals.
- Advanced High-Speed Profile.

Acknowledgments

The following companies were instrumental in the development of the *Open Core Protocol Specification*, Release 3.0.

All OCP-IP Specification Working Group members, including participants from:

- MIPS Technologies Inc.
- Nokia
- Sonics Inc.
- Texas Instruments Incorporated
- Toshiba Corporation Semiconductor Company
- Cadence

1 *Overview*

The Open Core Protocol™ (OCP) defines a high-performance, bus-independent interface between IP cores that reduces design time, design risk, and manufacturing costs for SOC designs.

An IP core can be a simple peripheral core, a high-performance microprocessor, or an on-chip communication subsystem such as a wrapped on-chip bus. The Open Core Protocol:

- Achieves the goal of IP design reuse. The OCP transforms IP cores, making them independent of the architecture and design of the systems in which they are used.
- Optimizes die area by configuring into the OCP interfaces only those features needed by the communicating cores.
- Simplifies system verification and testing by providing a firm boundary around each IP core that can be observed, controlled, and validated.

The approach adopted by the Virtual Socket Interface Alliance's (VSIA) Design Working Group on On-Chip Buses (DWGOCB) is to specify a bus wrapper to provide a bus-independent Transaction Protocol-level interface to IP cores.

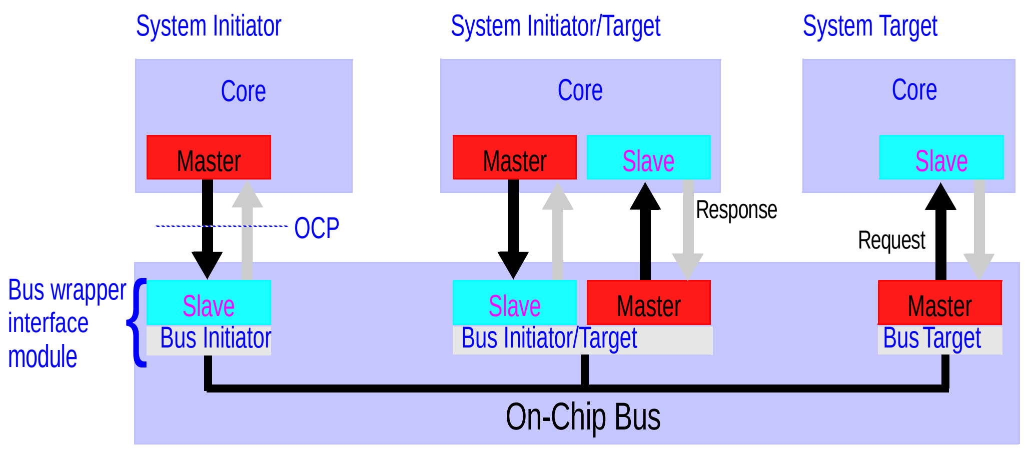
The OCP is equivalent to VSIA's Virtual Component Interface (VCI). While the VCI addresses only data flow aspects of core communications, the OCP is a superset of VCI that additionally supports configurable sideband control signaling and test harness signals. The OCP is the only standard that defines protocols to unify all of the inter-core communication.

1.1 OCP Characteristics

The OCP defines a point-to-point interface between two communicating entities, such as IP cores and bus interface modules (bus wrappers). One entity acts as the master of the OCP instance and the other as the slave. Only the master can present commands and is the controlling entity. The slave responds to commands presented to it, either by accepting data from the master, or presenting data to the master. For two entities to communicate in a peer-to-peer fashion, there need to be two instances of the OCP connecting them—one where the first entity is a master, and one where the first entity is a slave.

Figure 1 shows a simple system containing a wrapped bus and three IP core entities: one that is a system target, one that is a system initiator, and an entity that is both.

Figure 1 System Showing Wrapped Bus and OCP Instances



The characteristics of the IP core determine whether the core needs master, slave, or both sides of the OCP; the wrapper interface modules must act as the complementary side of the OCP for each connected entity. A transfer across this system occurs as follows. A system initiator (as the OCP master) presents command, control, and possibly data to its connected slave (a bus wrapper interface module). The interface module plays the request across the on-chip bus system. The OCP does not specify the embedded bus functionality. Instead, the interface designer converts the OCP request into an embedded bus transfer. The receiving bus wrapper interface module (as the OCP master) converts the embedded bus operation into a legal OCP command. The system target (OCP slave) receives the command and takes the requested action.

Each instance of the OCP is configured (by choosing signals or bit widths of a particular signal) based on the requirements of the connected entities and is independent of the others. For instance, system initiators may require more address bits in their OCP instances than do the system targets; the extra address bits might be used by the embedded bus to select which bus target is addressed by the system initiator.

The OCP is flexible. There are several useful models for how existing IP cores communicate with one another. Some employ pipelining to improve bandwidth and latency characteristics. Others use multiple-cycle access models, where signals are held static for several clock cycles to simplify timing analysis and reduce implementation area. Support for this wide range of behavior is possible through the use of synchronous handshaking signals that allow both the master and slave to control when signals are allowed to change.

1.2 Compliance

1. The core must include at least one OCP interface.
2. The core and OCP interfaces must be described using an RTL configuration file with the syntax specified in Chapter 8 on page 129.
3. Each OCP interface on the core must:
    - Comply with all aspects of the OCP interface specification
    - Have its timing described using a synthesis configuration file following the syntax specified in Chapter 9 on page 141.
4. The following practices are recommended but not required:
    a. Each non-OCP interface on the core should:
        - Be described using an interface configuration file with the syntax specified in Chapter 7 on page 123.
        - Have its timing described using a synthesis configuration file with the syntax specified in Chapter 9 on page 141.

    b. A performance report as specified in Chapter 16 on page 349 (or an equivalent report) should be included for the core.

Part I *Specification*

2 Theory of Operation

The Open Core Protocol interface addresses communications between the functional units (or IP cores) that comprise a system on a chip. The OCP provides independence from bus protocols without having to sacrifice high-performance access to on-chip interconnects. By designing to the interface boundary defined by the OCP, you can develop reusable IP cores without regard for the ultimate target system.

Given the wide range of IP core functionality, performance and interface requirements, a fixed definition interface protocol cannot address the full spectrum of requirements. The need to support verification and test requirements adds an even higher level of complexity to the interface. To address this spectrum of interface definitions, the OCP defines a highly configurable interface. The OCP's structured methodology includes all of the signals required to describe an IP cores' communications including data flow, control, and verification and test signals.

This chapter provides an overview of the concepts behind the Open Core Protocol, introduces the terminology used to describe the interface, and offers a high-level view of the protocol.

Point-to-Point Synchronous Interface

To simplify timing analysis, physical design, and general comprehension, the OCP is composed of uni-directional signals driven with respect to, and sampled by, the rising edge of the OCP clock. The OCP is fully synchronous (with the exception of reset) and contains no multi-cycle timing paths with respect to the OCP clock. All signals other than the clock signal are strictly point-to-point.

Bus Independence

A core utilizing the OCP can be interfaced to any bus. A test of any bus-independent interface is to connect a master to a slave without an intervening on-chip bus. This test not only drives the specification towards a fully symmetric interface but helps to clarify other issues. For instance, device selection techniques vary greatly among on-chip buses. Some use address decoders, while generate independent device-select signals (analogous to a board-level chip select). This complexity should be hidden from IP cores, especially since in the directly-connected case there is no decode/selection logic. OCP-compliant slaves receive device selection information integrated into the basic command field.

Arbitration schemes vary widely. Since there is virtually no arbitration in the directly-connected case, arbitration for any shared resource is the sole responsibility of the logic on the bus side of the OCP. This permits OCP-compliant masters to pass a command field across the OCP that the bus interface logic converts into an arbitration request sequence.

Commands

There are two basic commands—Read and Write—and five command extensions: WriteNonPost, Broadcast, ReadExclusive, ReadLinked, and WriteConditional. The WriteNonPost and Broadcast commands have semantics that are similar to the Write command. A WriteNonPost command explicitly instructs the slave not to post a write. For the Broadcast command, the master indicates that it is attempting to write to several or all remote target devices that are connected on the other side of the slave. As such, Broadcast is typically useful only for slaves that are in turn a master on another communication medium (such as an attached bus).

The other command extensions—ReadExclusive, ReadLinked and WriteConditional—are used for synchronization between system initiators. ReadExclusive is paired with Write or WriteNonPost, and has blocking semantics. ReadLinked, used in conjunction with WriteConditional has non-blocking (lazy) semantics. These synchronization primitives correspond to those available natively in the instruction sets of different processors.

Address/Data

Wide widths, characteristic of shared on-chip address and data buses, make tuning the OCP address and data widths essential for area-efficient implementation. Only those address bits that are significant to the IP core should cross the OCP to the slave. The OCP address space is flat and composed of 8-bit bytes (octets).

To increase transfer efficiencies, many IP cores have data field widths significantly greater than an octet. The OCP supports a configurable data width to allow multiple bytes to be transferred simultaneously. The OCP refers to the chosen data field width as the *word size* of the OCP. The term word is used in the traditional computer system context; that is, a *word* is the natural transfer unit of the block. OCP supports word sizes of power-of-two and non-power-of-two (as would be needed for a 12-bit DSP core). The OCP address is a byte address that is word aligned.

Transfers of less than a full word of data are supported by providing byte enable information that specifies which octets are to be transferred. Byte enables are linked to specific data bits (byte lanes). Byte lanes are not associated with particular byte addresses. This makes the OCP endian-neutral, able to support both big and little-endian cores.

Pipelining

The OCP allows pipelining of transfers. To support this feature, the return of read data and the provision of write data may be delayed after the presentation of the associated request.

Response

The OCP separates requests from responses. A slave can accept a command request from a master on one cycle and respond in a later cycle. The division of request from response permits pipelining. The OCP provides the option of having responses for Write commands, or completing them immediately without an explicit response.

Burst

Burst support is essential for many IP cores, to provide high transfer efficiency. The extended OCP supports annotation of transfers with burst information. Bursts can either include addressing information for each successive command (which simplifies the requirements for address sequencing/burst count processing in the slave), or include addressing information only once for the entire burst.

In-Band Information

Cores can pass core-specific information in-band in company with the other information being exchanged. In-band extensions exist for requests and responses, as well as read and write data. A typical use of in-band extensions is to pass cacheable information or data parity.

Tags

Tags are available in the OCP interface to control the ordering of responses. Without tags, a slave must return responses in the order that the requests were issued by the master. Similarly, writes must be committed in order. With the addition of tags, responses can be returned out-of-order, and write data can be committed out-of-order with respect to requests, as long as the transactions target different addresses. (Refer to Section 4.7.1 on page 57 for the case when requests from different tags of a thread target overlapping addresses.) The tag links the response back to the original request.

Tagging is useful when a master core, such as a processor, can handle out-of-order return, because it allows a slave core such as a DRAM controller to service requests in the order that is most convenient, rather than the order in which requests were sent by the master.

Out-of-order request and response delivery can also be enabled using multiple threads. The major differences between threads and tags are that threads can have independent flow control for each thread and have no ordering rules for transactions on different threads. Tags, on the other hand, exist within a single thread and are restricted to shared flow control. Tagged transactions to overlapping addresses have to be committed in order but their responses may be reordered if the transactions have different tag IDs (see Section 4.7.1 on page 57). Implementing independent flow control requires independent buffering for each thread, leading to more complex implementations. Tags enable lower overhead implementations for out-of-order return of responses at the expense of some concurrency.

Threads and Connections

To support concurrency and out-of-order processing of transfers, the extended OCP supports the notion of multiple threads. Transactions among threads have no ordering requirements, and independent flow control from one another. Transfers within a single thread must remain ordered unless tags are in use. The concepts of threads and tags are hierarchical: each thread has its own flow control, and ordering within a thread either follows the request order strictly, or is governed by tags.

While the notion of a thread is a local concept between a master and a slave communicating over an OCP, it is possible to globally pass thread information from initiator to target using connection identifiers. Connection information helps to identify the initiator and determine priorities or access permissions at the target.

Interrupts, Errors, and other Sideband Signaling

While moving data between devices is a central requirement of on-chip communication systems, other types of communications are also important. Different types of control signaling are required to coordinate data transfers (for instance, high-level flow control) or signal system events (such as interrupts). Dedicated point-to-point data communication is sometimes required. Many devices also require the ability to notify the system of errors that may be unrelated to address/data transfers.

The OCP refers to all such communication as *sideband* (or out-of-band) signaling, since it is not directly related to the protocol state machines of the dataflow portion of the OCP. The OCP provides support for such signals through sideband signaling extensions.

Errors are reported across the OCP using two mechanisms. The error response code in the response field describes errors resulting from OCP transfers that provide responses. Write-type commands without responses cannot use the in-band reporting mechanism. The second method for reporting errors across the OCP uses out-of band error fields. These signals report more generic sideband errors, including those associated with posted write commands.

Two additional groups of sideband signals—the reset signal group and the connection signal group—are used to control the state of the interface itself. The reset signals enable the master and/or slave to immediately transition the interface from normal operation into a reset state, independently from any activity on the dataflow signals. The connection signals allow the master and slave to cooperate to cleanly achieve quiescence before putting the interface into a disconnected state where none of the other in-band nor sideband signals have meaning, except for the OCP clock.

3 Signals and Encoding

OCP interface signals are grouped into dataflow, sideband, and test signals. The dataflow signals are divided into five groups: basic signals, simple extensions, burst extensions, tag extensions, and thread extensions. A small set of the signals from the basic dataflow group are required in all OCP configurations. The remaining dataflow signals are optional; optional signals can be configured as needed to support additional core communication requirements. All sideband and test signals are optional.

The OCP is a synchronous interface with a single clock signal. All OCP signals, other than the clock and reset, are driven with respect to, and sampled by, the rising edge of the OCP clock. Except for clock, OCP signals are strictly point-to-point and uni-directional. The complete set of OCP signals are shown in Figure 4 on page 36.

3.1 Dataflow Signals

The dataflow signals consist of a small set of required signals and a number of optional signals that can be configured to support additional core communication requirements. The dataflow signals are grouped into five groups: basic signals, simple extensions (options such as byte enables and in-band information), burst extensions (support for bursting), tag extensions (re-ordering support), and thread extensions (multi-threading support).

The naming conventions for dataflow signals use the prefix M for signals driven by the OCP master and S for signals driven by the OCP slave.

3.1.1 Basic Signals

Table 1 lists the basic OCP signals. Only Clk and MCmd are required. The remaining OCP signals are optional.

Table 1 Basic OCP Signals

Name	**Width**	**Driver**	**Function**
Clk	1	varies	Clock input
EnableClk	1	varies	Enable OCP clock
MAddr	configurable	master	Transfer address
MCmd	3	master	Transfer command
MData	configurable	master	Write data
MDataValid	1	master	Write data valid
MRespAccept	1	master	Master accepts response
SCmdAccept	1	slave	Slave accepts transfer
SData	configurable	slave	Read data
SDataAccept	1	slave	Slave accepts write data
SResp	2	slave	Transfer response

Clk

Input clock signal for the OCP clock. The rising edge of the OCP clock is defined as a rising edge of Clk that samples the asserted EnableClk. Falling edges of Clk and any rising edge of Clk that does not sample EnableClk asserted do not constitute rising edges of the OCP clock.

EnableClk

EnableClk indicates which rising edges of Clk are the rising edges of the OCP clock, that is. which rising edges of Clk should sample and advance interface state. Use the `enableclk` parameter to configure this signal. EnableClk is driven by a third entity and serves as an input to both the master and the slave.

When `enableclk` is set to 0 (the default), the EnableClk signal is not present and the OCP behaves as if EnableClk is constantly asserted. In that case all rising edges of Clk are rising edges of the OCP clock.

MAddr

The Transfer address, MAddr, specifies the slave-dependent address of the resource targeted by the current transfer. To configure this field into the OCP, use the `addr` parameter. To configure the width of this field, use the `addr_wdth` parameter.

MAddr is a byte address that must be aligned to the OCP word size (`data_wdth`). The parameter `data_wdth` defines a minimum `addr_wdth` value that is based on the data bus byte width, and is defined as:

$$\texttt{min\_addr\_wdth} = \max(1, \text{floor}(\log_2(\texttt{data\_wdth})) - 2)$$

If the OCP word size is larger than a single byte, the aggregate is addressed at the OCP word-aligned address and the lowest order address bits are hardwired to 0. If the OCP word size is not a power-of-two, the address is the same as it would be for an OCP interface with a word size equal to the next larger power-of-two.

MCmd

Transfer command. This signal indicates the type of OCP transfer the master is requesting. Each non-idle command is either a read or write type request, depending on the direction of data flow. Commands are encoded as follows.

Table 2 Command Encoding

MCmd[2:0]			**Command**	**Mnemonic**	**Request Type**
0	0	0	Idle	IDLE	(none)
0	0	1	Write	WR	write
0	1	0	Read	RD	read
0	1	1	ReadEx	RDEX	read
1	0	0	ReadLinked	RDL	read
1	0	1	WriteNonPost	WRNP	write
1	1	0	WriteConditional	WRC	write
1	1	1	Broadcast	BCST	write

The set of allowable commands can be limited using the `write_enable`, `read_enable`, `readex_enable`, `writenonpost_enable`, `rdlwrc_enable`, and `broadcast_enable` parameters as described in Section 4.9.1 on page 59.

MData

Write data. This field carries the write data from the master to the slave. The field is configured into the OCP using the `mdata` parameter and its width is configured using the `data_wdth` parameter. The width is not restricted to multiples of 8.

MDataValid

Write data valid. When set to 1, this bit indicates that the data on the MData field is valid. Use the `datahandshake` parameter to configure this field into the OCP.

MRespAccept

Master response accept. The master indicates that it accepts the current response from the slave with a value of 1 on the MRespAccept signal. Use the `respaccept` parameter to enable this field into the OCP.

SCmdAccept

Slave accepts transfer. A value of 1 on the SCmdAccept signal indicates that the slave accepts the master's transfer request. To configure this field into the OCP, use the `cmdaccept` parameter.

SData

This field carries the requested read data from the slave to the master. The field is configured into the OCP using the `sdata` parameter and its width is configured using the `data_wdth` parameter. The width is not restricted to multiples of eight.

SDataAccept

Slave accepts write data. The slave indicates that it accepts pipelined write data from the master with a value of 1 on SDataAccept. This signal is meaningful only when datahandshake is in use. Use the `dataaccept` parameter to configure this field into the OCP.

SResp

Response field from the slave to a transfer request from the master. The field is configured into the OCP using the `resp` parameter. Response encoding is as follows.

Table 3 Response Encoding

SResp[1:0]		**Response**	**Mnemonic**
0	0	No response	NULL
0	1	Data valid / accept	DVA
1	0	Request failed	FAIL
1	1	Response error	ERR

The use of responses is explained in Section 4.4 on page 49. FAIL is a non-error response that indicates a successful transfer and is reserved for a response to a WriteConditional command for which the write is not performed, as described in Section 4.4 on page 49.

3.1.2 Simple Extensions

Table 4 lists the simple OCP extensions. The extensions add to the OCP interface address spaces, byte enables, and additional core-specific information for each phase.

Table 4 Simple OCP Extensions

Name	**Width**	**Driver**	**Function**
MAddrSpace	configurable	master	Address space
MByteEn	configurable	master	Request phase byte enables
MDataByteEn	configurable	master	Datahandshake phase write byte enables
MDataInfo	configurable	master	Additional information transferred with the write data

Name	Width	Driver	Function
MReqInfo	configurable	master	Additional information transferred with the request
SDataInfo	configurable	slave	Additional information transferred with the read data
SRespInfo	configurable	slave	Additional information transferred with the response

MAddrSpace

This field specifies the address space and is an extension of the MAddr field that is used to indicate the address region of a transfer. Examples of address regions are the register space versus the regular memory space of a slave or the user versus supervisor space for a master.

The MAddrSpace field is configured into the OCP using the `addrspace` parameter. The width of the MAddrSpace field is configured with the `addrspace_wdth` parameter. While the encoding of the MAddrSpace field is core-specific, it is recommended that slaves use 0 to indicate the internal register space.

MByteEn

Byte enables. This field indicates which bytes within the OCP word are part of the current transfer. See Section 4.4.1 on page 50 for more detail on request and datahandshake phase byte enables and their relationship. There is one bit in MByteEn for each byte in the OCP word. Setting MByteEn[*n*] to 1 indicates that the byte associated with data wires [(8*n* + 7):8*n*] should be transferred. The MByteEn field is configured into the OCP using the `byteen` parameter and is allowed only if `data_wdth` is a multiple of 8 (that is, the data width is an integer number of bytes).

The allowable patterns on MByteEn can be limited using the `force_aligned` parameter as described on page 60.

MDataByteEn

Write byte enables. This field indicates which bytes within the OCP word are part of the current write transfer. See Section 4.4.1 on page 50 for more detail on request and datahandshake phase byte enables and their relationship. There is one bit in MDataByteEn for each byte in the OCP word. Setting MDataByteEn[*n*] to 1 indicates that the byte associated with MData wires [(8*n* + 7):8*n*] should be transferred. The MDataByteEn field is configured into the OCP using the `mdatabyteen` parameter. Setting `mdatabyteen` to 1 is only allowed if `datahandshake` is 1, and only if `data_wdth` is a multiple of 8 (that is, the data width is an integer number of bytes).

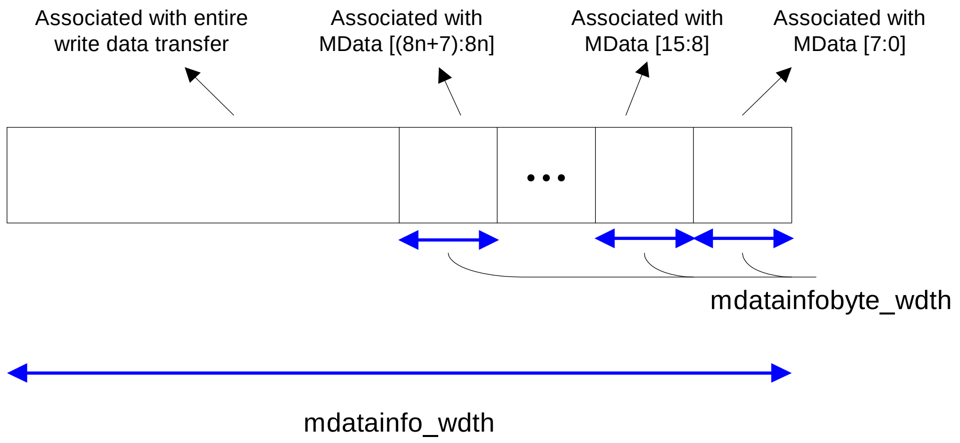
The allowable patterns on MDataByteEn can be limited using the `force_aligned` parameter as described on page 60.

MDataInfo

Extra information sent with the write data. The master uses this field to send additional information sequenced with the write data. The encoding of the information is core-specific. To be interoperable with masters that do not provide this signal, design slaves to be operable in a normal mode when the signal is tied off to its default tie-off value as specified in Table 16 on page 31. Sample uses are data byte parity or error correction code values. Use the `mdatainfo` parameter to configure this field into the OCP, and the `mdatainfo_wdth` parameter to configure its width.

This field is divided in two: the low-order bits are associated with each data byte, while the high-order bits are associated with the entire write data transfer. The number of bits to associate with each data byte is configured using the `mdatainfobyte_wdth` parameter. The low-order `mdatainfobyte_wdth` bits of MDataInfo are associated with the MData[7:0] byte, and so on.

Figure 2 MDataInfo Field


Associated with entire write data transfer
Associated with MData [(8n+7):8n]
Associated with MData [15:8]
Associated with MData [7:0]
mdatainfobyte_wdth
mdatainfo_wdth

MReqInfo

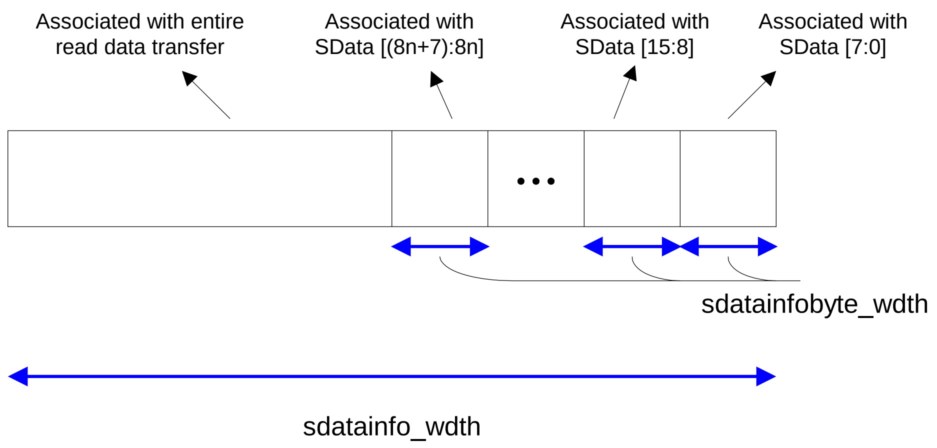
Extra information sent with the request. The master uses this field to send additional information sequenced with the request. The encoding of the information is core-specific. To be interoperable with masters that do not provide this signal, design slaves to be operable in a normal mode when the signal is tied off to its default tie-off value as specified in Table 16 on page 31. Sample uses are cacheable storage attributes or other access mode information. Use the `reqinfo` parameter to configure this field into the OCP, and the `reqinfo_wdth` parameter to configure its width.

SDataInfo

Extra information sent with the read data. The slave uses this field to send additional information sequenced with the read data. The encoding of the information is core-specific. To be interoperable with slaves that do not provide this signal, design masters to be operable in a normal mode when the signal is tied off to its default tie-off value as specified in Table 16 on page 31. Sample uses are data byte parity or error correction code values. Use the `sdatainfo` parameter to configure this field into the OCP, and the `sdatainfo_wdth` parameter to configure its width.

This field is divided into two pieces: the low-order bits are associated with each data byte, while the high-order bits are associated with the entire read data transfer. The number of bits to associate with each data byte is configured using the `sdatainfobyte_wdth` parameter. The low-order `sdatainfobyte_wdth` bits of SDataInfo are associated with the SData[7:0] byte, and so on.

Figure 3 SDataInfo Field



Associated with entire read data transfer
Associated with SData [(8n+7):8n]
Associated with SData [15:8]
Associated with SData [7:0]
• • •
sdatainfobyte_wdth
sdatainfo_wdth

SRespInfo

Extra information sent with the response. The slave uses this field to send additional information sequenced with the response. The encoding of the information is core-specific. To be interoperable with slaves that do not provide this signal, design masters to be operable in a normal mode when the signal is tied off to its default tie-off value as specified in Table 16 on page 31. Sample uses are status or error information such as FIFO full or empty indications. Use the `respinfo` parameter to configure this field into the OCP, and the `respinfo_wdth` parameter to configure its width.

3.1.3 Burst Extensions

Table 5 lists the OCP burst extensions. The burst extensions allow the grouping of multiple transfers that have a defined address relationship. The burst extensions are enabled only when MBurstLength is included in the interface, or tied off to a value other than one.

Table 5 OCP Burst Extensions

Name	**Width**	**Driver**	**Function**
MAtomicLength	configurable	master	Length of atomic burst
MBlockHeight	configurable	master	Height of 2D block burst
MBlockStride	configurable	master	Address offset between 2D block rows
MBurstLength	configurable	master	Burst length
MBurstPrecise	1	master	Given burst length is precise
MBurstSeq	3	master	Address sequence of burst

Name	Width	Driver	Function
MBurstSingleReq	1	master	Burst uses single request/ multiple data protocol
MDataLast	1	master	Last write data in burst
MDataRowLast	1	master	Last write data in row
MReqLast	1	master	Last request in burst
MReqRowLast	1	master	Last request in row
SRespLast	1	slave	Last response in burst
SRespRowLast	1	slave	Last response in row

MAtomicLength

This field indicates the minimum number of transfers within a burst that are to be kept together as an atomic unit when interleaving requests from different initiators onto a single thread at the target. To configure this field into the OCP, use the `atomiclength` parameter. To configure the width of this field, use the `atomiclength_wdth` parameter. A binary encoding of the number of transfers is used. A 0 value is not legal for MAtomicLength.

MBlockHeight

This field indicates the number of rows of data to be transferred in a two-dimensional block burst (the height of the block of data). A binary encoding of the height is used. To configure this field into the OCP, use the `blockheight` parameter. To configure the width of this field, use the `blockheight_wdth` parameter.

MBlockStride

This field indicates the address difference between the first data word in each consecutive row in a two-dimensional block burst. The stride value is a binary encoded byte address offset and must be aligned to the OCP word size (`data_wdth`). To configure this field into the OCP, use the `blockstride` parameter. To configure the width of this field, use the `blockstride_wdth` parameter.

MBurstLength

For a BLCK burst (see Table 6), this field indicates the number of transfers for a row of the burst and stays constant throughout the burst. A BLCK burst is always precise. For a precise non-BLCK burst, this field indicates the number of transfers for the entire burst and stays constant throughout the burst. For imprecise bursts, the value indicates the best guess of the number of transfers remaining (including the current request), and may change with every request. To configure this field into the OCP, use the `burstlength` parameter. To configure the width of this field, use the `burstlength_wdth` parameter. A binary encoding of the number of transfers is used. 0 is not a legal encoding for MBurstLength.

MBurstPrecise

This field indicates whether the precise length of a burst is known at the start of the burst or not. When set to 1, MBurstLength indicates the precise length of the burst during the first request of the burst. To configure this field into the OCP, use the `burstprecise` parameter. If set to 0, MBurstLength for each request is a hint of the remaining burst length.

MBurstSeq

This field indicates the sequence of addresses for requests in a burst. To configure this field into the OCP, use the `burstseq` parameter. The encodings of the MBurstSeq field are shown in Table 6. The definition of the address sequences is described in Section 4.6.1 on page 53.

Table 6 MBurstSeq Encoding

MBurstSeq[2:0]			Burst Sequence	Mnemonic
0	0	0	Incrementing	INCR
0	0	1	Custom (packed)	DFLT1
0	1	0	Wrapping	WRAP
0	1	1	Custom (not packed)	DFLT2
1	0	0	Exclusive OR	XOR
1	0	1	Streaming	STRM
1	1	0	Unknown	UNKN
1	1	1	2-dimensional Block	BLCK

MBurstSingleReq

The burst has a single request with multiple data transfers. This field indicates whether the burst has a request per data transfer, or a single request for all data transfers. To configure this field into the OCP, use the `burstsinglereq` parameter. When this field is set to 0, there is a one-to-one association of requests to data transfers; when set to 1, there is a single request for all data transfers in the burst.

MDataLast

Last write data in a burst. This field indicates whether the current write data transfer is the last in a burst. To configure this field into the OCP, use the `datalast` parameter with `datahandshake` set to 1. When this field is set to 0, more write data transfers are coming for the burst; when set to 1, the current write data transfer is the last in the burst.

MDataRowLast

Last write data in a row. This field identifies the last transfer in a row. The last data transfer in a burst is always considered the last in a row, and BLCK burst sequences also have a last in a row transfer after every MBurstLength transfers. To configure this field into the OCP, use the `datarowlast` parameter. If this field is set to 0, additional write data transfers can be expected for the current row; when set to 1, the current write data transfer is the last in the row.

MReqLast
: Last request in a burst. This field indicates whether the current request is the last in this burst. To configure this field into the OCP, use the `reqlast` parameter. When this field is set to 0, more requests are coming for this burst; when set to 1, the current request is the last in the burst.

MReqRowLast
: Last request in a row. This field identifies the last request in a row. The last request in a burst is always considered the last in a row, and BLCK burst sequences also have a last-in-a-row request after every MBurstLength requests. To configure this field into the OCP, use the `reqrowlast` parameter. When this field is set to 0, more requests can be expected for the current row; when set to 1, the current request is the last in the row.

SRespLast
: Last response in a burst. This field indicates whether the current response is the last in this burst. To configure this field into the OCP, use the `resplast` parameter. When the field is set to 0, more responses are coming for this burst; when set to 1, the current response is the last in the burst.

SRespRowLast
: Last response in a row. This field identifies the last response in a row. The last response in a burst is always considered the last in a row, and BLCK burst sequences also have a last in a row response after every MBurstLength responses. Use the `resprowlast` parameter to configure this field. When this field is set to 0, more can be expected for the current row; when set to 1, the current response is the last in the row.

3.1.4 Tag Extensions

Table 7 lists OCP tag extensions, which add support for tagging OCP transfers to enable out-of-order responses and write data commit. The binary encoded *TagID signals must each carry a value in the range 0 to (`#tags-1`) where `#tags` is the value specified by the `tags` parameter.

Table 7 OCP Tag Extensions

Name	Width	Driver	Function
MDataTagID	configurable	master	Ordering tag for write data
MTagID	configurable	master	Ordering tag for request
MTagInOrder	1	master	Do not reorder this request
STagID	configurable	slave	Ordering tag for response
STagInOrder	1	slave	This response is not reordered

MDataTagID

Write data tag. This variable-width field provides the tag associated with the current write data. The field carries the binary-encoded tag value. MDataTagID is required if `tags` is greater than 1 and the `datahandshake` parameter is 1. The field width is $\lceil \log_2(\text{tags}) \rceil$.

MTagID

Request tag. This variable-width field provides the tag associated with the current transfer request. If `tags` is greater than 1, this field is enabled. The field width is $\lceil \log_2(\text{tags}) \rceil$.

MTagInOrder

Assertion of this single-bit field indicates that the current request should not be reordered with respect to other requests that had this field asserted. This field is enabled by the `taginorder` parameter. Both MTagInOrder and STagInOrder are present in the interface, or else neither may be present.

STagID

Response tag. This variable-width field provides the tag associated with the current transfer response. This field is enabled if `tags` is greater than 1, and the `resp` parameter is set to 1. The field width is $\lceil \log_2(\text{tags}) \rceil$.

STagInOrder

Assertion of this single-bit field indicates that the current response is associated with an in-order request and was not reordered with respect to other requests that had MTagInOrder asserted. This field is enabled if both the `taginorder` and the `resp` parameters are set to 1.

3.1.5 Thread Extensions

Table 8 shows a list of OCP thread extensions that add support for multi-threading of the OCP interface. Thread numbering begins at 0 and is sequential. The binary encoded *ThreadID must carry a value less than the `threads` parameter.

Table 8 OCP Thread Extensions

Name	Width	Driver	Function
MConnID	configurable	master	Connection identifier
MDataThreadID	configurable	master	Write data thread identifier
MThreadBusy	configurable	master	Master thread busy
MThreadID	configurable	master	Request thread identifier
SDataThreadBusy	configurable	slave	Slave write data thread busy
SThreadBusy	configurable	slave	Slave request thread busy
SThreadID	configurable	slave	Response thread identifier

MConnID

Connection identifier. This variable-width field provides the binary encoded connection identifier associated with the current transfer request. To configure this field use the `connid` parameter. The field width is configured with the `connid_wdth` parameter.

MDataThreadID

Write data thread identifier. This variable-width field provides the thread identifier associated with the current write data. The field carries the binary-encoded value of the thread identifier.

MDataThreadID is required if `threads` is greater than 1 and the `datahandshake` parameter is set to 1. MDataThreadID has the same width as MThreadID and SThreadID.

MThreadBusy

Master thread busy. The master notifies the slave that it cannot accept any responses associated with certain threads. The MThreadBusy field is a vector (one bit per thread). A value of 1 on any given bit indicates that the thread associated with that bit is busy. Bit 0 corresponds to thread 0, and so on. The width of the field is set using the `threads` parameter. It is legal to enable a one-bit MThreadBusy interface for a single-threaded OCP. To configure this field, use the `mthreadbusy` parameter. See Section 4.3.2.4 on page 44 for a description of the flow control options associated with MThreadBusy.

MThreadID

Request thread identifier. This variable-width field provides the thread identifier associated with the current transfer request. If `threads` is greater than 1, this field is enabled. The field width is the next whole integer of $\lceil \log_2(\text{threads}) \rceil$.

SDataThreadBusy

Slave write data thread busy. The slave notifies the master that it cannot accept any new datahandshake phases associated with certain threads. The SDataThreadBusy field is a vector, one bit per thread. A value of 1 on any given bit indicates that the thread associated with that bit is busy. Bit 0 corresponds to thread 0, and so on.

The width of the field is set using the `threads` parameter. It is legal to enable a one-bit SDataThreadBusy interface for a single-threaded OCP. To configure this field, use the `sdatathreadbusy` parameter. See Section 4.3.2.4 on page 44 for a description of the flow control options associated with SDataThreadBusy.

SThreadID

Response thread identifier. This variable-width field provides the thread identifier associated with the current transfer response. This field is enabled if `threads` is greater than 1 and the `resp` parameter is set to 1. The field width is $\lceil \log_2(\text{threads}) \rceil$.

SThreadBusy

Slave thread busy. The slave notifies the master that it cannot accept any new requests associated with certain threads. The SThreadBusy field is a vector, one bit per thread. A value of 1 on any given bit indicates that the thread associated with that bit is busy. Bit 0 corresponds to thread 0, and so on. The width of the field is set using the `threads` parameter. It is legal to enable a one-bit SThreadBusy interface for a single-threaded OCP. To configure this field, use the `sthreadbusy` parameter. See Section 4.3.2.4 on page 44 for a description of the flow control options associated with SThreadBusy.

3.2 Sideband Signals

Sideband signals are OCP signals that are not part of the dataflow phases, and so can change asynchronously with the request/response flow but are generally synchronous to the rising edge of the OCP clock. Sideband signals convey control information such as reset, interrupt, error, and core-specific flags. They also exchange control and status information between a core and an attached system. All sideband signals are optional except for reset signals. Either the MReset_n or the SReset_n signal must be present.

Table 9 lists the OCP sideband extensions.

Table 9 Sideband OCP Signals

Name	Width	Driver	Function
MConnect	2	master	Master connection state
MError	1	master	Master Error
MFlag	configurable	master	Master flags
MReset_n	1	master	Master reset
SConnect	1	slave	Slave connection vote
SError	1	slave	Slave error
SFlag	configurable	slave	Slave flags
SInterrupt	1	slave	Slave interrupt
SReset_n	1	slave	Slave reset
SWait	1	slave	Slave delays connection change
ConnectCap	1	tie-off	Connection capability tie-off
Control	configurable	system	Core control information
ControlBusy	1	core	Hold control information
ControlWr	1	system	Control information has been written
Status	configurable	core	Core status information
StatusBusy	1	core	Status information is not consistent
StatusRd	1	system	Status information has been read

3.2.1 Connection, Reset, Interrupt, Error, and Core-Specific Flag Signals

MConnect

Master connection state. This signal indicates the current connection state of the interface. The master changes this state based upon input from the slave SConnect signal and the master's desired connection state, but state transitions must respect the slave SWait signal. Connection states are encoded as shown in Table 10.

Table 10 Connection State Encoding

MConnect[1:0]		**State**	**Mnemonic**	**Connected?**
0	0	Disconnected by master	M_OFF	No
0	1	Waiting to transition	M_WAIT	Matches prior state
1	0	Disconnected by slave	M_DISC	No
1	1	Connected	M_CON	Yes

The M_WAIT state is transient. When the master is changing the connection state between any two of the other states, it must enter M_WAIT if the slave is asserting SWait (S_WAIT). The connection status of the interface does not change while in M_WAIT. The master can only transition to a non-transient connection state once it samples SWait negated (S_OK). The MConnect signal is configured by the `connection` parameter and must maintain the value M_CON if the ConnectCap tie-off is 0. If ConnectCap is 1, the reset value of MConnect is M_OFF.

SConnect

Slave connection vote. This signal indicates the slave's willingness to have the master in the M_CON state. The slave's vote is encoded as follows.

Table 11 Slave Connection Vote Encoding

SConnect	**Connection Vote**	**Mnemonic**
0	Vote to disconnect	S_DISC
1	Vote to connect	S_CON

The SConnect signal is configured by the `connection` parameter and must maintain the value S_CON if the ConnectCap tie-off is 0. If ConnectCap is 1, the reset value of SConnect is S_DISC.

SWait

Slave delays connection change. This signal allows the slave to force the master to transition through the M_WAIT state before changing the connection state to M_OFF, M_DISC, or M_CON. This signal is encoded as follows:

Table 12 Slave Connection Change Delay Encoding

SWait	Function	Mnemonic
0	Allow connection status change	S_OK
1	Delay connection status change	S_WAIT

The SWait signal is configured by the `connection` parameter and must maintain the value S_OK if the ConnectCap tie-off is 0. If ConnectCap is 1, the reset value of SWait is S_OK.

ConnectCap
: Connection capability tie-off. This signal is tied off at component instantiation to indicate whether the interface supports the connection state machine. Tie ConnectCap to logic 0 on a master or slave if the connected slave or master, respectively, does not implement the connection protocol. In such case, the interface is always connected (i.e. it behaves as if in the M_CON state). If ConnectCap is tied to logic 1, then both master and slave must support the connection protocol. The ConnectCap tie-off signal is configured by the `connection` parameter and has no default value.

MError
: Master error. When the MError signal is set to 1, the master notifies the slave of an error condition. The MError field is configured with the `merror` parameter.

MFlag
: Master flags. This variable-width set of signals allows the master to communicate out-of-band information to the slave. Encoding is completely core-specific.

To configure this field into the OCP, use the `mflag` parameter. To configure the width of this field, use the `mflag_wdth` parameter.

MReset_n
: Master reset. The MReset_n signal is active low, as shown in Table 13. The MReset_n field is enabled by the `mreset` parameter.

Table 13 MReset Signal

MReset_n	Function
0	Reset Active
1	Reset Inactive

SError
: Slave error. With a value of 1 on the SError signal the slave indicates an error condition to the master. The SError field is configured with the `serror` parameter.

SFlag

Slave flags. This variable-width set of signals allows the slave to communicate out-of-band information to the master. Encoding is completely core-specific.

To configure this field into the OCP, use the `sflag` parameter. To configure the width of this field, use the `sflag_wdth` parameter.

SInterrupt

Slave interrupt. The slave may generate an interrupt with a value of 1 on the SInterrupt signal. The SInterrupt field is configured with the `interrupt` parameter.

SReset_n

Slave reset. The SReset_n signal is active low, as shown in Table 14. The SReset_n field is enabled by the `sreset` parameter.

Table 14 SReset Signal

SReset_n	**Function**
0	Reset Active
1	Reset Inactive

3.2.2 Control and Status Signals

The remaining sideband signals are designed for the exchange of control and status information between an IP core and the rest of the system. They make sense only in this environment, regardless of whether the IP core acts as a master or slave across the OCP interface.

Control

Core control information. This variable-width field allows the system to drive control information into the IP core. Encoding is core-specific.

Use the `control` parameter to configure this field into the OCP. Use the `control_wdth` parameter to configure the width of this field.

ControlBusy

Core control busy. When this signal is set to 1, the core tells the system to hold the control field value constant. Use the `controlbusy` parameter to configure this field into the OCP.

ControlWr

Core control event. This signal is set to 1 by the system to indicate that control information is written by the system. Use the `controlwr` parameter to configure this field into the OCP.

Status

Core status information. This variable-width field allows the IP core to report status information to the system. Encoding is core-specific.

Use the `status` parameter to configure this field into the OCP. Use the `status_wdth` parameter to configure the width of this field.

StatusBusy

Core status busy. When this signal is set to 1, the core tells the system to disregard the status field because it may be inconsistent. Use the `statusbusy` parameter to configure this field into the OCP.

StatusRd

Core status event. This signal is set to 1 by the system to indicate that status information is read by the system. To configure this field into the OCP, use the `statusrd` parameter.

3.3 Test Signals

The test signals add support for scan, clock control, and IEEE 1149.1 (JTAG). All test signals are optional.

Table 15 Test OCP Signals

Name	**Width**	**Driver**	**Function**
Scanctrl	configurable	system	Scan control signals
Scanin	configurable	system	Scan data in
Scanout	configurable	core	Scan data out
ClkByp	1	system	Enable clock bypass mode
TestClk	1	system	Test clock
TCK	1	system	Test clock
TDI	1	system	Test data in
TDO	1	core	Test data out
TMS	1	system	Test mode select
TRST_N	1	system	Test reset

3.3.1 Scan Interface

The Scanctrl, Scanin, and Scanout signals together form a scan interface into a given IP core.

Scanctrl

Scan mode control signals. Use this variable width field to control the scan mode of the core. Set `scanport` to 1 to configure this field into the OCP interface. Use the `scanctrl_wdth` parameter to configure the width of this field.

Scanin

Scan data in for a core's scan chains. Use the `scanport` parameter, to configure this field into the OCP interface and `scanport_wdth` to control its width.

Scanout
: Scan data out from the core's scan chains. Use the `scanport` parameter to configure this field into the OCP interface and `scanport_wdth` to control its width.

3.3.2 Clock Control Interface

The ClkByp and TestClk signals together form the clock control interface into a given IP core. This interface is used to control the core's clocks during scan operation.

ClkByp
: Enable clock bypass signal. When set to 1, this signal instructs the core to bypass the external clock source and use TestClk instead. Use the `clkctrl_enable` parameter to configure this signal into the OCP interface.

TestClk
: A gated test clock. This clock becomes the source clock when ClkByp is asserted during scan operations. Use the `clkctrl_enable` parameter to configure this signal into the OCP interface.

3.3.3 Debug and Test Interface

The TCK, TDI, TDO, TMS, and TRST_N signals together form an IEEE 1149 debug and test interface for the OCP.

TCK
: Test clock as defined by IEEE 1149.1. Use the `jtag_enable` parameter to add this signal to the OCP interface.

TDI
: Test data in as defined by IEEE 1149.1. Use the `jtag_enable` parameter to add this signal to the OCP interface.

TDO
: Test data out as defined by IEEE 1149.1. Use the `jtag_enable` parameter to add this signal to the OCP interface.

TMS
: Test mode select as defined by IEEE 1149.1. Use the `jtag_enable` parameter to add this signal to the OCP interface.

TRST_N
: Test logic reset signal. This is an asynchronous active low reset as defined by IEEE 1149.1. Use the `jtagtrst_enable` parameter to add this signal to the OCP interface.

3.4 Signal Configuration

The set of signals making up the OCP interface is configurable to match the characteristics of the IP core. Throughout this chapter, configuration parameters were mentioned that control the existence and width of various OCP fields. Table 16 summarizes the configuration parameters, sorted by interface signal group. For each signal, the table also specifies the default constant tie-off, which is the inferred value of the signal if it is not configured into the OCP interface and if no other constant tie-off is specified.

For the ControlBusy, EnableClk, MRespAccept, SCmdAccept, SDataAccept, MThreadBusy, SThreadBusy, SDataThreadBusy, MReset_n, SReset_n, SInterrupt, and StatusBusy signals, the default tie-off is also the only legal tie-off.

Table 16 OCP Signal Configuration Parameters

Group	Signal	Parameter to add signal to interface	Parameter to control width	Default Tie-off
Basic	Clk	Required	Fixed	n/a
	EnableClk	enableclk	Fixed	1
	MAddr	addr	addr_wdth	0
	MCmd	Required	Fixed	n/a
	MData	mdata	data_wdth	0
	MDataValid	datahandshake	Fixed	n/a
	MRespAccept[1]	respaccept	Fixed	1
	SCmdAccept	cmdaccept	Fixed	1
	SData[1]	sdata	data_wdth	0
	SDataAccept[2]	dataaccept	Fixed	1
	SResp	resp	Fixed	n/a
Simple	MAddrSpace	addrspace	addrspace_wdth	0
	MByteEn[3]	byteen	data_wdth	all 1s
	MDataByteEn[4]	mdatabyteen	data_wdth	all 1s
	MDataInfo	mdatainfo	mdatainfo_wdth[5]	0
	MReqInfo	reqinfo	reqinfo_wdth	0
	SDataInfo[1]	sdatainfo	sdatainfo_wdth[6]	0
	SRespInfo[1]	respinfo	respinfo_wdth	0

Group	Signal	Parameter to add signal to interface	Parameter to control width	Default Tie-off
Burst	MAtomicLength[7]	atomiclength	atomiclength_wdth	1
	MBlockHeight[7,8]	blockheight	blockheight_wdth[9]	1
	MBlockStride[7,8]	blockstride	blockstride_wdth	0
	MBurstLength	burstlength	burstlength_wdth[10]	1
	MBurstPrecise[7, 11]	burstprecise	Fixed	1
	MBurstSeq[7]	burstseq	Fixed	INCR
	MBurstSingleReq[7, 12]	burstsinglereq	Fixed	0
	MDataLast[7, 13]	datalast	Fixed	n/a
	MDataRowLast[7, 8, 13, 14]	datarowlast	Fixed	n/a
	MReqLast[7]	reqlast	Fixed	n/a
	MReqRowLast[7, 8, 15]	reqrowlast	Fixed	n/a
	SRespLast[1, 7]	resplast	Fixed	n/a
	SRespRowLast[1, 7, 8, 16]	resprowlast	Fixed	n/a
Tag	MDataTagID[17]	tags>1 and datahandshake	tags	0
	MTagID	tags>1	tags	0
	MTagInOrder[18]	taginorder	Fixed	0
	STagID	tags>1 and resp	tags	0
	STagInOrder[19]	taginorder and resp	Fixed	0
Thread	MConnID	connid	connid_wdth	0
	MDataThreadID	threads>1 and datahandshake	threads	0
	MThreadBusy[1, 20]	mthreadbusy	threads	0
	MThreadID	threads>1	threads	0
	SDataThreadBusy[21]	sdatathreadbusy	threads	0
	SThreadBusy[22]	sthreadbusy	threads	0
	SThreadID	threads>1 and resp	threads	0

Group	Signal	Parameter to add signal to interface	Parameter to control width	Default Tie-off
Sideband	ConnectCap	connection	Fixed	n/a
	Control	control	control_wdth	0
	ControlBusy[23]	controlbusy	Fixed	0
	ControlWr[24]	controlwr	Fixed	n/a
	MConnect[25]	connection	2	M_CON
	MError	merror	Fixed	0
	MFlag	mflag	mflag_wdth	0
	MReset_n	mreset	Fixed	1
	SConnect[25]	connection	1	S_CON
	SError	serror	Fixed	0
	SFlag	sflag	sflag_wdth	0
	SInterrupt	interrupt	Fixed	0
	SReset_n	sreset	Fixed	1
	Status	status	status_wdth	0
	StatusBusy[26]	statusbusy	Fixed	0
	StatusRd[27]	statusrd	Fixed	n/a
	SWait[25]	connection	1	S_OK

Group	Signal	Parameter to add signal to interface	Parameter to control width	Default Tie-off
Test	ClkByp	clkctrl_enable	Fixed	n/a
	Scanctrl	scanport	scanctrl_wdth	n/a
	Scanin	scanport	scanport_wdth	n/a
	Scanout	scanport	scanport_wdth	n/a
	TCK	jtag_enable	Fixed	n/a
	TDI	jtag_enable	Fixed	n/a
	TDO	jtag_enable	Fixed	n/a
	TestClk	clkctrl_enable	Fixed	n/a
	TMS	jtag_enable	Fixed	n/a
	TRST_N[28]	jtagtrst_enable	Fixed	n/a

1 MRespAccept, MThreadBusy, SData, SDataInfo, SRespInfo, SRespLast, and SRespRowLast may be included only if the `resp` parameter is set to 1.

2 SDataAccept can be included only if `datahandshake` is set to 1.

3 MByteEn has a width of `data_wdth`/8 and can only be included when either `mdata` or `sdata` is set to 1 and `data_wdth` is an integer multiple of 8.

4 MDataByteEn has a width of `data_wdth`/8 and can only be included when `mdata` is set to 1, `datahandshake` is set to 1, and `data_wdth` is an integer multiple of 8.

5 `mdatainfo_wdth` must be ≥ `mdatainfobyte_wdth * data_wdth/8` and can be used only if `data_wdth` is a multiple of 8. `mdatainfobyte_wdth` specifies the partitioning of `MDataInfo` into transfer-specific and per-byte fields.

6 `sdatainfo_wdth` must be ≥ `sdatainfobyte_wdth * data_wdth/8` and can be used only if `data_wdth` is a multiple of 8. `sdatainfobyte_wdth` specifies the partitioning of SDataInfo into transfer-specific and per-byte fields.

7 MAtomicLength, MBlockHeight, MBlockStride, MBurstPrecise, MBurstSeq, MBurstSingleReq, MDataLast, MDataRowLast, MReqLast, MReqRowLast, SRespLast, and SRespRowLast may be included in the interface or tied off to non-default values only if MBurstLength is included or tied off to a value other than 1.

8 MBlockHeight, MBlockStride, MDataRowLast, MReqRowLast, and SRespRowLast may be included or tied off to non-default values only if `burstseq_blck_enable` is set to 1 and MBurstPrecise is included or tied off to a value of 1.

9 `blockheight_wdth` can never be 1.

10 `burstlength_wdth` can never be 1.

11 If no sequences other than WRAP, XOR, or BLCK are enabled, MBurstPrecise must be tied off to 1.

12 If any write-type commands are enabled, MBurstSingleReq can only be included when `datahandshake` is set to 1. If the only enabled burst address sequence is UNKN, MBurstSingleReq cannot be included or tied off to a non-default value.

13 MDataLast and MDataRowLast can be included only if the `datahandshake` parameter is set to 1.

14 MDataRowLast can only be included if MDataLast is included.

15 MReqRowLast can only be included if MReqLast is included.

16 SRespRowLast can only be included if SRespLast is included.

17 MDataTagID is included if `tags` is greater than 1 and the `datahandshake` parameter is set to 1.

18 MTagInOrder can only be included if `tags` is greater than 1.

19 STagInOrder can only be included if `tags` is greater than 1.

20 MThreadBusy has a width equal to `threads`. It may be included for single-threaded OCP interfaces.

21 SDataThreadBusy has a width equal to `threads`. It may be included for single-threaded OCP interfaces and may only be included if `datahandshake` is 1.

22 SThreadBusy has a width equal to `threads`. It may be included for single-threaded OCP interfaces.

23 ControlBusy can only be included if both Control and ControlWr exist.

24 ControlWr can only be included if Control exists.

25 The default tie-off values for MConnect, SConnect and SWait are the only allowed tie-off values.

26 StatusBusy can only be included if Status exists.

27 StatusRd can only be included if Status exists.

28 TRST_N can only be included if `jtag_enable` is set to 1.

3.4.1 Signal Directions

Figure 4 on page 36 summarizes all OCP signals. The direction of some signals (for example, MCmd) depends on whether the module acts as a master or slave, while the direction of other signals (for example, Control) depends on whether the module acts as a system or a core. The combination of these two choices provides four possible module configurations as shown in Table 17.

Table 17 Module Configuration Based on Signal Directions

	Acts as Core	**Acts as System**
Acts as OCP Master	Master	System master
Acts as OCP Slave	Slave	System slave

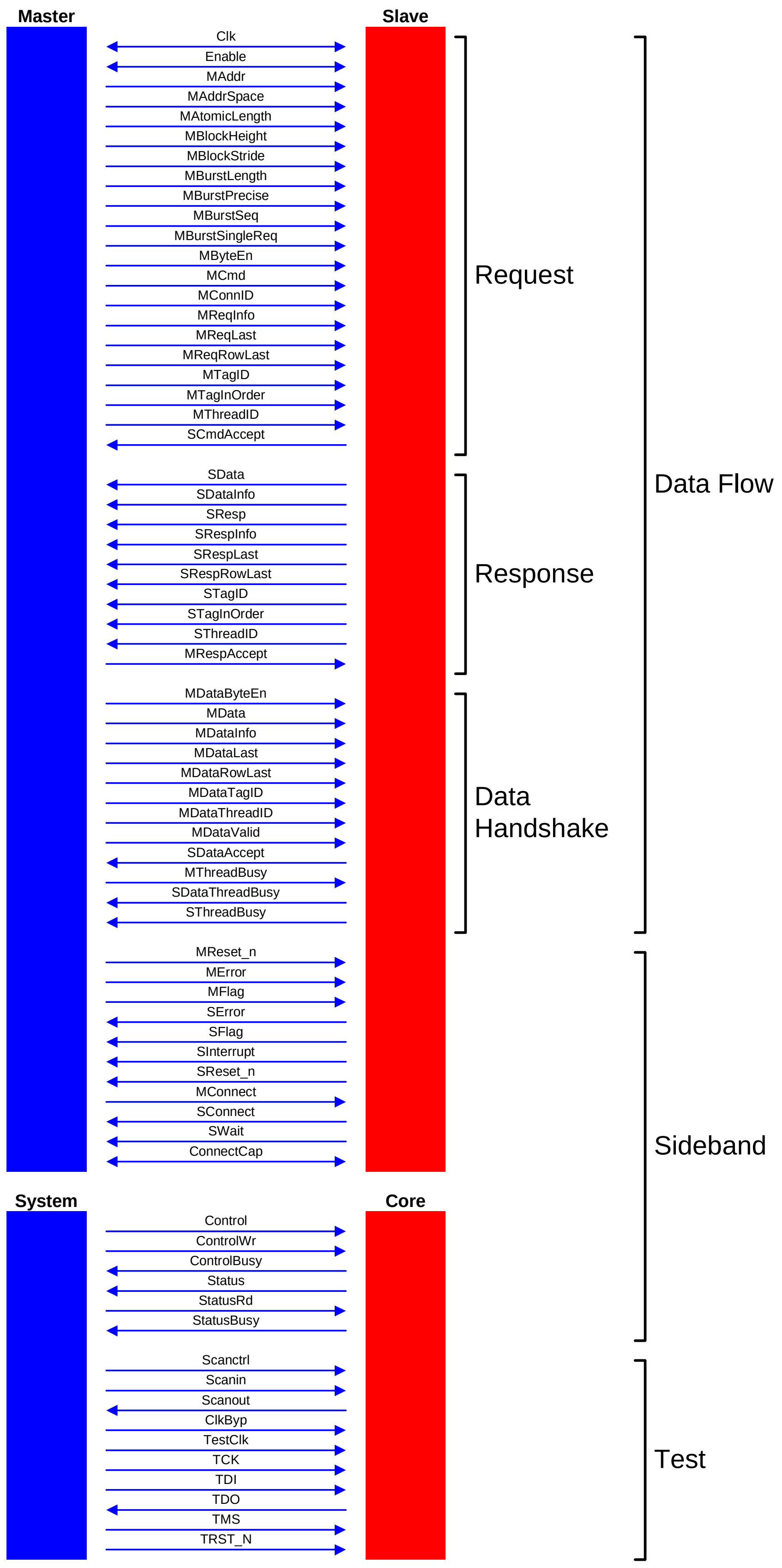
For example, if a module acts as OCP master and also as system, it is designated a system master. In addition to the notion of modules, it is useful to introduce an “interface” type. All modules have interfaces. Also, there is a “monitor” interface type which observes all OCP signals. The permitted connectivity between interface types is shown in Table 18.

Table 18 Interface Types

Type	**Connects To**	**Cannot Connect To**
Master	System slave, Slave, Monitor	Master, System master
Slave	System master, Master, Monitor	Slave, System slave
System master	Slave, Monitor	Master, System Master, System slave
System slave	Master, Monitor	Slave, System slave, System master
Monitor	Master, System master, Slave, System slave	Monitor

The Clk, EnableClk, and ConnectCap signals are special in that they are supplied by a third (external) entity that is neither of the two modules connected through the OCP interface.

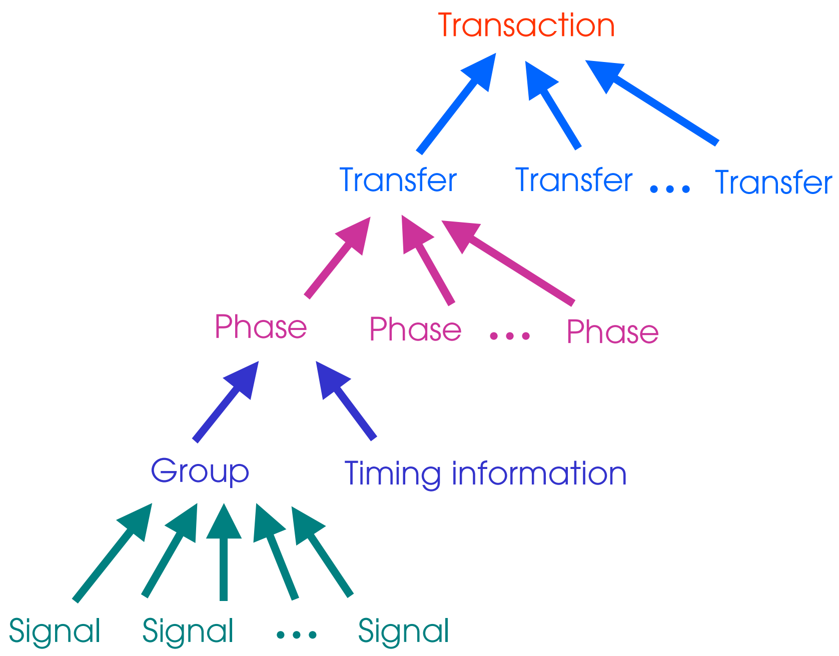
Figure 4 OCP Signal Summary



4 Protocol Semantics

This chapter defines the semantics of the OCP protocol by assigning meanings to the signal encodings described in the preceding chapter. Figure 5 provides a graphic view of the hierarchy of elements that compose the OCP.

Figure 5 Hierarchy of Elements



4.1 Signal Groups

Some OCP fields are grouped together because they must be active at the same time. The data flow signals are divided into three signal groups: request signals, response signals, and datahandshake signals. A list of the signals that belong to each group is shown in Table 19.

Table 19 OCP Signal Groups

Group	**Signal**	**Condition**
Request	MAddr	always
	MAddrSpace	always
	MAtomicLength	always
	MBlockHeight	always
	MBlockStride	always
	MBurstLength	always
	MBurstPrecise	always
	MBurstSeq	always
	MBurstSingleReq	always
	MByteEn	always
	MCmd	always
	MConnID	always
	MData*	datahandshake = 0
	MDataInfo*	datahandshake = 0
	MReqInfo	always
	MReqLast	always
	MReqRowLast	always
	MTagID	always
	MTagInOrder	always
	MThreadID	always

Group	Signal	Condition
Response	SData	always
	SDataInfo	always
	SResp	always
	SRespInfo	always
	SRespLast	always
	SRespRowLast	always
	STagID	always
	STagInOrder	always
	SThreadID	always
Datahandshake	MData*	datahandshake = 1
	MDataByteEn	always
	MDataInfo*	datahandshake = 1
	MDataLast	always
	MDataRowLast	always
	MDataTagID	always
	MDataThreadID	always
	MDataValid	always

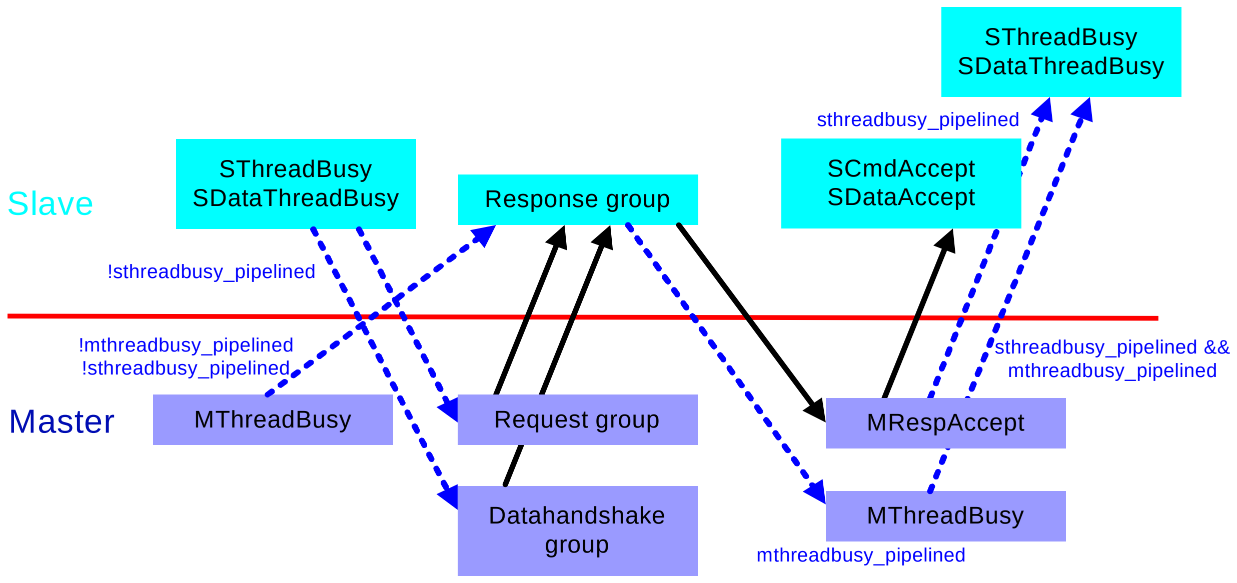
* MData and MDataInfo belong to the request group, unless the `datahandshake` configuration parameter is enabled. In that case they belong to the datahandshake group.

4.2 Combinational Dependencies

It is legal for some signal or signal group outputs to be derived from inputs without an intervening latch point, that is, combinationally. To avoid combinational loops, other outputs cannot be derived in this manner. Figure 6 describes a partial order of combinational dependency. For any arrow shown, the signal or signal group can be derived combinationally from the signal at the point of origin of the arrow or another signal earlier in the dependency chain. No other combinational dependencies are allowed.

MThreadBusy, SDataThreadBusy, and SThreadBusy each appear twice in Figure 6. The two appearances of each signal are mutually exclusive based upon the setting of the `mthreadbusy_pipelined`, `sdatathreadbusy_pipelined`, and `sthreadbusy_pipelined` parameters. Refer to Section 4.3.2.4 on page 44 for more information about these parameters.

Figure 6 Legal Combinational Dependencies Between Signals and Signal Groups



Combinational paths are not allowed within the sideband and test signals, or between those signals and the data flow signals. The only legal combinational dependencies are within the data flow signals. Data flow signals, however, may be combinationally derived from MReset_n and SReset_n.

For timing purposes, some of the allowed combinational paths are designated as preferred paths and are described in Table 65 on page 317.

4.3 Signal Timing and Protocol Phases

This section specifies when a signal can or must be valid.

4.3.1 OCP Clock

The rising edge of the OCP clock signal is used to sample other OCP signals to advance the state of the interface. When the EnableClk signal is not present, the OCP clock is simply the Clk signal. When the EnableClk signal is present (`enableclk` is 1), only rising edges of Clk that sample EnableClk asserted are considered rising edges of the OCP clock. Therefore, when EnableClk is 0, rising edges of Clk are not rising edges of the OCP clock and OCP state is not advanced.

This restriction applies to all signals in the OCP interface. In particular, the requirements for reset assertion (described on page 46) are measured in OCP clock cycles.

4.3.2 Dataflow Signals

Signals in a signal group must all be valid at the same time.

- The request group is valid whenever a command other than Idle is presented on the MCmd field.
- The response group is valid whenever a response other than Null is presented on the SResp field.
- The datahandshake group is valid whenever a 1 is presented on the MDataValid field.

The accept signal associated with a signal group is valid only when that group is valid.

- The SCmdAccept signal is valid whenever a command other than Idle is presented on the MCmd field.
- The MRespAccept signal is valid whenever a response other than Null is presented on the SResp field.
- The SDataAccept signal is valid whenever a 1 is presented on the MDataValid field.

The signal groups map on a one-to-one basis to protocol phases. All signals in the group must be held steady from the beginning of a protocol phase until the end of that phase. Outside of a protocol phase, all signals in the corresponding group (except for the signal that defines the beginning of the phase) are “don’t care.”

In addition, the MData and MDataInfo fields are a “don’t care” during read-type requests, and the SData and SDataInfo fields are a “don’t care” for responses to write-type requests. Non-enabled data bytes in MData and bits in MDataInfo as well as non-enabled bytes in SData and bits in SDataInfo are a “don’t care.” The MDataByteEn field is “don’t care” during read-type transfers. If MDataByteEn is present, the MByteEn field is “don’t care” during write-type transfers. MTagID is a “don’t care” if MTagInOrder is asserted and MDataTagID is a “don’t care” for the corresponding datahandshake phase. Similarly, STagID is a “don’t care” if STagInOrder is asserted.

- A request phase begins whenever the request group becomes active. It ends when the SCmdAccept signal is sampled by the rising edge of the OCP clock as 1 during a request phase.
- A response phase begins whenever the response group becomes active. It ends when the MRespAccept signal is sampled by the rising edge of the OCP clock as 1 during a response phase.

  If MRespAccept is not configured into the OCP interface (`respaccept` = 0) then MRespAccept is assumed to be on; that is the response phase is exactly one cycle long.
- A datahandshake phase begins whenever the datahandshake signal group becomes active. It ends when the SDataAccept signal is sampled by the rising edge of the OCP clock as 1 during a datahandshake phase.

For all phases, it is legal to assert the corresponding accept signal in the cycle that the phase begins, allowing the phase to complete in a single cycle.

4.3.2.1 Phases in a Transfer

An OCP transfer consists of several phases, as shown in Table 20. Every transfer has a request phase. Read-type requests always have a response phase. For write-type requests, the OCP can be configured with or without responses or datahandshake. Without a response, a write-type request completes upon completion of the request phase (posted write model).

Table 20 Phases in each Transfer for MBurstSingleReq set to 0

MCmd	Phases	Condition
Read, ReadEx, ReadLinked	Request, response	always
Write, Broadcast	Request	datahandshake = 0 writeresp_enable = 0
Write, Broadcast	Request, response	datahandshake = 0 writeresp_enable = 1
WriteNonPost, WriteConditional	Request, response	datahandshake = 0
Write, Broadcast	Request, datahandshake	datahandshake = 1 writeresp_enable = 0
Write, Broadcast	Request, datahandshake, response	datahandshake = 1 writeresp_enable = 1
WriteNonPost, WriteConditional	Request, datahandshake, response	datahandshake = 1

Single request, multiple data (SRMD) bursts, described in Section 4.6.5 on page 55, have a single request phase and multiple data transfer phases, as shown in Table 21.

Table 21 Phases in a Transaction for MBurstSingleReq set to 1

MCmd	Phases	Condition
Read	Request, H∗L[*] response	always
Write, Broadcast	Request, H∗L[†] datahandshake	datahandshake = 1 writeresp_enable = 0
Write, Broadcast	Request, H∗L[†] datahandshake, response	datahandshake = 1 writeresp_enable = 1
WriteNonPost	Request, H∗L[†] datahandshake, response	datahandshake = 1

* H refers to the burst height (MBlockHeight), and is 1 for all burst sequences other than BLCK.

† L refers to the burst length (MBurstLength).

4.3.2.2 Phase Ordering Within a Transfer

The OCP is causal: within each transfer a request phase must precede the associated datahandshake phase which in turn, must precede the associated response phase. The specific constraints are:

- A datahandshake phase cannot begin before the associated request phase begins, but can begin in the same OCP clock cycle.
- A datahandshake phase cannot end before the associated request phase ends, but can end in the same OCP clock cycle.
- A response phase cannot begin before the associated request phase begins, but can begin in the same OCP clock cycle.
- A response phase cannot end before the associated request phase ends, but can end in the same OCP clock cycle.
- If there is a datahandshake phase and a response phase, the response phase cannot begin before the associated datahandshake phase (or last datahandshake phase for single request, multiple data bursts) begins, but can begin in the same OCP clock cycle.
- If there is a datahandshake phase and a response phase, the response phase cannot end before the associated datahandshake phase (or last datahandshake phase for single request, multiple data bursts) ends, but can end in the same OCP clock cycle.

4.3.2.3 Phase Ordering Between Transfers

If tags are not in use, the ordering of transfers is determined by the ordering of their request phases. Also, the following rules apply.

- Since two phases of the same type belonging to different transfers both use the same signal wires, the phase of a subsequent transfer cannot begin before the phase of the previous transfer has ended. If the first phase ends in cycle *x*, the second phase can begin in cycle *x*+1.
- The ordering of datahandshake phases must follow the order set by the request phases including multiple datahandshake phases for single request, multiple data (SRMD) bursts.
- The ordering of response phases must follow the order set by the request phases including multiple response phases for SRMD bursts.
- For SRMD bursts, a request or response phase is shared between multiple transfers. Each individual transfer must obey the ordering rules described in Section 4.3.2.2, even when a phase is shared with another transfer.
- Where no phase ordering is specified, by the previous rules, the effect of two transfers that are addressing the same location (as specified by MAddr, MAddrSpace, and MByteEn [or MDataByteEn, if applicable]) must be the same as if the two transfers were executed in the same order but without any phase overlap. This ensures that read/write hazards yield predictable results.

For example, on an OCP interface with datahandshake enabled, a read following a write to the same location cannot start its response phase until the write has started its datahandshake phase, otherwise the latest write data cannot be returned for the read.

If tags are in use, the preceding rules are partially relaxed. Refer to Section 4.7.1 on page 57 for a more detailed explanation.

4.3.2.4 Ungrouped Signals

Signals not covered in the description of signal groups and phases are MThreadBusy, SDataThreadBusy, and SThreadBusy. Without further protocol restriction, the cycle timing of the transition of each bit that makes up each of these three fields is not specified relative to the other dataflow signals. This means that there is no specific time for an OCP master or slave to drive these signals, nor a specific time for the signals to have the desired flow-control effect. Without further restriction, MThreadBusy, SDataThreadBusy, and SThreadBusy can only be treated as a hint.

For stricter semantics use the protocol configuration parameters `mthreadbusy_exact`, `sdatathreadbusy_exact`, and `sthreadbusy_exact`. These parameters can be set to 1 only when the corresponding signal has been enabled.

The parameters `mthreadbusy_pipelined`, `sdatathreadbusy_pipelined`, and `sthreadbusy_pipelined` can be used to set the relative protocol timing of the MThreadBusy, SDataThreadBusy, and SThreadBusy signals with respect to their phases. The `*_pipelined` parameters[1] can only be enabled when the corresponding `*_exact` parameter is enabled.

The `*_exact` parameters define strict protocol semantics for the corresponding phase. The receiver of the phase identifies (through the corresponding ThreadBusy signals) to the sender of the phase which threads can accept an asserted phase. The sender will not assert a phase on a busy thread, and the receiver accepts any phases asserted on threads that are not busy. To avoid ambiguity, the corresponding phase Accept signal may not be present on the interface, and is considered tied off to 1. The resulting phase has exact flow control and is non-blocking. See Section 4.9.1.5 on page 61 for configuration restrictions associated with ThreadBusy-related parameters.

The `*_pipelined` parameters control the cycle relationship between the ThreadBusy signal and the corresponding phase assertion. When the `*_pipelined` parameter is disabled (the default), the ThreadBusy signal defines the flow control for the current cycle, so phase assertion depends upon that cycle's ThreadBusy values. This mode corresponds to the timing arcs in Figure 6 where the ThreadBusy generation appears at the beginning of the OCP cycle. When a `*_pipelined` parameter is enabled, the ThreadBusy signal defines the flow control for the next cycle enabling fully sequential interface behavior, where non-blocking flow control can be accomplished without combinational paths crossing the interface twice in a single cycle.

[1] The notation `*_pipelined` means the set of all parameter names ending in `_pipelined`.

Combinational paths may still be present to enable the phase receiver to consider interface activity in the current cycle before signaling the ThreadBusy signal that affects the next cycle. This corresponds to the timing arcs in Figure 6 where ThreadBusy appears at the end of the OCP cycle. When a `_pipelined` parameter is enabled, exact flow control is not possible for the first cycle after reset is de-asserted, since the associated ThreadBusy would have to be provided while reset was asserted. When `sthreadbusy_pipelined` is enabled the master may not assert the request phase in the first cycle after reset.

The effect of these parameters is as follows:

- If `mthreadbusy_exact` is enabled, `mthreadbusy_pipelined` is disabled, and the master cannot accept a response on a thread, it must set the MThreadBusy bit for that thread to 1 in that cycle. The slave must not present a response on a thread when the corresponding MThreadBusy bit is set to 1. Any response presented by the slave on a thread that is not busy is accepted by the master in that cycle.
- If `mthreadbusy_exact` and `mthreadbusy_pipelined` are enabled and the master cannot accept a response on a thread in the next cycle, it must set the MThreadBusy bit for that thread to 1 in the current cycle. If an MThreadBusy bit was set to 1 in the prior cycle, the slave cannot present a response on a thread in the current cycle. Any response presented by the slave on a thread that was not busy in the prior cycle is accepted by the master in that cycle.
- If `sdatathreadbusy_exact` is enabled, `sdatathreadbusy_piplelined` is disabled, and the slave cannot accept a datahandshake phase on a thread, the slave must set the SDataThreadBusy bit for that thread to 1 in that cycle. The master must not present a datahandshake phase on a thread when the corresponding SDataThreadBusy bit is set to 1. Any datahandshake phase presented by the master on a thread that is not busy is accepted by the slave in that cycle.
- If `sdatathreadbusy_exact` and `sdatathreadbusy_piplelined` are enabled and the slave cannot accept a datahandshake phase on a thread in the next cycle, the slave must set the SDataThreadBusy bit for that thread to 1 in the current cycle. If an SDataThreadBusy bit was set to 1 in the prior cycle, the master cannot present a datahandshake on the corresponding thread in the current cycle. Any datahandshake presented by the master on a thread that was not busy in the prior cycle is accepted by the slave in that cycle.
- If `sthreadbusy_exact` is enabled, `sthreadbusy_piplelined` is disabled, and the slave cannot accept a command on a thread, the slave must set the SThreadBusy bit for that thread to 1 in that cycle. The master must not present a request on a thread when the corresponding SThreadBusy bit is set to 1. Any request presented by the master on a thread that is not busy is accepted by the slave in that cycle.
- If `sthreadbusy_exact` and `sthreadbusy_piplelined` are enabled and the slave cannot accept a request on a thread in the next cycle, the slave must set the SThreadBusy bit for that thread to 1 in the current cycle. If an SThreadBusy bit was set to 1 in the prior cycle, the master cannot

present a request on the corresponding thread in the current cycle. Any request presented by the master on a thread that was not busy in the prior cycle is accepted by the slave in that cycle.

4.3.3 Sideband and Test Signals

4.3.3.1 Reset

The OCP interface provides an interface reset signal for each master and slave. At least one of these signals must be present. If both signals are present, the composite reset state of the interface is derived as the logical AND of the two signals (that is, the interface is in reset as long as one of the two resets is asserted).

Treat OCP reset signals as fully synchronous to the OCP clock, where the receiver samples the incoming reset using the rising edge of the clock and deassertion of the reset meets the receiver's timing requirements with respect to the clock. An exception to this rule exists when the assertion edge of an OCP reset signal is asynchronous to the OCP clock. This behavior handles the practice of forcing all reset signals to be combinationally asserted for power-on reset or other hardware reset conditions without waiting for a clock edge.

Once a reset signal is sampled asserted by the rising edge of the OCP clock, all incomplete transactions, transfers and phases are terminated and both master and slave must transition to a state where there are no pending OCP requests or responses. When a reset signal is asserted asynchronously, there may be ambiguity about transactions that completed, or were aborted due to timing differences between the arrival of the OCP reset and the OCP clock.

For systems requiring precision use synchronous reset assertion, or only apply reset asynchronously if the interface is either quiescent or hung. MReset_n and SReset_n must be asserted for at least 16 cycles of the OCP clock to ensure that the master and slave reach a consistent internal state. When one or both of the reset signals are asserted in a given cycle, all other OCP signals must be ignored in that cycle. The master and slave must each be able to reach their reset state regardless of the values presented on the OCP signals. If the master or slave require more than 16 cycles of reset assertion, the requirement must be documented in the IP core specifications.

At the clock edge that all reset signals present are sampled deasserted, all OCP interface signals must be valid. In particular, it is legal for the master to begin its first request phase in the same clock cycle that reset is deasserted.

4.3.3.2 Connection Signals

The OCP interface offers an optional connection protocol that enables the master to control the connection state of the interface based upon the input of both master and slave, which can be used to implement robust schemes for power management. The protocol makes a clear difference between an OCP disconnected state resulting solely from a slave vote (M_DISC state) versus one resulting from a master vote independently from the slave side vote

(M_OFF state). It has a single connected state (M_CON) and a transient state (M_WAIT) that allows the slave to control how quickly the master may transition from one stable state to another.

The connection protocol is implemented using fully synchronous signals sampled by the rising edge of the OCP clock and no combinational paths are allowed between the connection signals. Since any transitions between the stable connection states requires that the interface be quiescent, the interface reset is not needed explicitly by the connection protocol and connection state transitions may occur independently from the reset state of the interface. Neither data flow nor sideband communication (other than the connection signals) is allowed in a disconnected state. However, the connection signals (MConnect, SConnect and SWait) are always valid to enable proper operation of the connection protocol. Since sideband communication is only reliable in the connected state (M_CON), the 16 cycle reset assertion requirement can only be reliably met in the connected state.

MConnect[1:0] provides the OCP socket connection state and is driven by the master. The master must ensure a minimum duration of 2 cycles in a stable state (M_CON, M_OFF or M_DISC) to permit the slave to sample a new stable state and then assert SWait (to S_WAIT) to influence the next potential connection state transition. This is a side effect of the timing requirements of the connection protocol. MConnect[1:0] does not convey the master’s vote on the OCP connection state. This vote information is not explicitly visible at the interface. The four valid connection states follow.

- The M_OFF state is a stable state where the interface is disconnected due to the master’s vote, independently from any concurrent vote from the slave. It is likely required that the interface reach the M_OFF state before performing specific power reduction techniques such as powering down the master.
- The M_DISC state is a stable state where the interface is disconnected resulting solely from the slave’s vote on SConnect. Since the master is voting for connection, but prevented by the slave, the master may implement an alternate behavior for upstream traffic intended for the disconnected slave. This alternate behavior is out of the scope of the connection protocol, but may be addressed in a future extension.Transitions to M_DISC are only allowed after the master has sampled the slave’s vote to disconnect (SConnect is S_DISC).
- The M_CON state is a stable state where the interface is fully connected. It is the only state in which the master is allowed to begin any transactions, and the master may not leave M_CON unless all transactions are complete. Transitions to M_CON are only allowed after the master has sampled the slave’s vote for a connection (SConnect is S_CON). The master may not present the first transaction on the interface until the cycle after transitioning to M_CON.
- The M_WAIT state is a transient state where the master is indicating to the slave that it is in the process of changing the connection state. The master can change between stable connection states without entering M_WAIT only if the SWait signal is negated. M_WAIT is disconnected for dataflow communication but sideband communication is allowed in

M_WAIT only if the prior state was M_CON. The master and slave must cooperate to ensure that all sideband communication is complete before exiting M_WAIT for a disconnected state.

SConnect provides the slave's vote on the OCP connection state. The slave may change its vote at any time, but must be ready to support the connected state (M_CON) when driving SConnect to S_CON.

SWait allows the slave to control how the master transitions between the stable connection states. By asserting SWait (S_WAIT) in a stable state, the slave forces the master to transition through the M_WAIT state and the master may not leave M_WAIT until it has sampled SWait negated (S_OK). The slave must assert SWait in situations where the master could otherwise transition from M_CON to a disconnected state without allowing the slave to become quiescent. SWait can be tied-off to logic 0 (S_OK) in case the slave can accept immediate transitions by the master between the stable connection states.

4.3.3.3 Interrupt, Error, and Core Flags

There is no specific timing associated with SInterrupt, SError, MFlag, MError, and SFlag. The timing of these signals is core-specific.

4.3.3.4 Status and Control

The following rules assure that control and status information can be exchanged across the OCP without any combinational paths from inputs to outputs and at the pace of a slow core.

- Control must be held steady for a full cycle after the cycle in which it has transitioned, which means it cannot transition more frequently than every other cycle. If ControlBusy was sampled active at the end of the previous cycle, Control can not transition in the current cycle. In addition, Control must be held steady for the first two cycles after reset is deasserted.
- If Control transitions in a cycle, ControlWr (if present) must be driven active for that cycle. ControlWr following the rules for Control, cannot be asserted in two consecutive cycles.
- ControlBusy allows a core to force the system to hold Control steady. ControlBusy may only start to be asserted immediately after reset, or in the cycle after ControlWr is asserted, but can be left asserted for any number of cycles.
- While StatusBusy is active, Status is a "don't care". StatusBusy enables a core to prevent the system from reading the current status information. While StatusBusy is active the core may not read Status. StatusBusy can be asserted at any time and be left asserted for any number of cycles.
- StatusRd is active for a single cycle every time the status register is read by the system. If StatusRd was asserted in the previous cycle, it must not be asserted in the current cycle, so it cannot transition more frequently than every other cycle.

4.3.3.5 Test Signals

Scanin and Scanout are “don’t care” while Scanctrl is inactive (but the encoding of inactive for Scanctrl is core-specific).

TestClk is “don’t care” while `ClkByp` is 0.

The timing of TRST_N, TCK, TMS, TDI, and TDO is specified in the IEEE 1149 standard.

4.4 Transfer Effects

A successful transfer is one that completes without error. For write-type requests without responses, there is no in-band error indication. For all other requests, a non-ERR response (that is, a DVA or FAIL response) indicates a successful transfer. The FAIL response is legal only for WriteConditional commands[1]. This section defines the effect that a successful transfer has on a slave. The request acts on the addressed location, where the term address refers to the combination of MAddr, MAddrSpace, and MByteEn (or MDataByteEn, if applicable). Two addresses are said to match if they are identical in all components. Two addresses are said to conflict, if the mutual exclusion (lock or monitor) logic is built to alias the two addresses into the same mutual exclusion unit. The transfer effects of each command are:

Idle
: None.

Read
: Returns the latest value of the addressed location on the SData field.

ReadEx
: Returns the latest value of the addressed location on the SData field. Sets a lock for the initiating thread on that location. The next request on the thread that issued a ReadEx must be a Write or WriteNonPost to the matching address. Requests from other threads to a conflicting address that is locked are not committed until the lock is released. If the ReadEx request returns an ERR response, it is slave-specific whether the lock is actually set or not. Refer to Section 4.4.3 on page 51 for details.

ReadLinked
: Returns the latest value of the addressed location on the SData field. Sets a reservation in a monitor for the corresponding thread on at least that location. Requests of any type from any thread to a conflicting address that is reserved are not blocked from proceeding, but may clear the reservation.

Write/WriteNonPost
: Places the value on the MData field in the addressed location. Unlocks access to the matched address if locked by a ReadEx issued on the same initiating thread.Clears the reservations on any conflicting addresses set by other threads.

[1] For all commands except those following a posted write model, a DVA response also indicates that the transfer is committed.

WriteConditional

If a reservation is set for the matching address and for the corresponding thread, the write is performed as it would be for a Write or WriteNonPost. Simultaneously, the reservation is cleared for all threads on any conflicting address. If no reservation is set for the corresponding thread, the write is not performed, a FAIL response is returned, and no reservations are cleared.

Broadcast

Places the value on the MData field in the addressed location that may map to more than one slave in a system-dependent way. Broadcast clears the reservations on any conflicting addresses set by other threads.

If a transfer is unsuccessful, the effect of the transfer is unspecified. Higher-level protocols must determine what happened and handle any clean-up.

The synchronization commands ReadEx / Write, ReadEx / WriteNonPost, and ReadLinked / WriteConditional have special restrictions with regard to data width conversion and partial words. In systems where these commands are sent through a bridge or interconnect that performs wide-to-narrow data width conversion between two OCP interfaces, the initiator must issue only commands within the subset of partial words that can be expressed as a single word of the narrow OCP interface. For maximum portability, single-byte synchronization operations are recommended.

4.4.1 Partial Word Transfers

An OCP interface may be configured to include partial word transfers by using either the MByteEn field, or the MDataByteEn field, or both.

- If neither field is present, then only whole word transfers are possible.
- If only MByteEn is present, then the partial word is specified by this field for both read type transfers and write type transfers as part of the request phase.
- If only MDataByteEn is present, then the partial word is specified by this field for write type transfers as part of the datahandshake phase, and partial word reads are not supported.
- If both MByteEn and MDataByteEn are present, then MByteEn specifies partial words for read transfers as part of the request phase, and MDataByteEn specifies partial words for write transfers as part of the datahandshake phase.

It is legal to use a request with all byte enables deasserted. Such requests must follow all the protocol rules, except that they are treated as no-ops by the slave: the response phase signals SData and SDataInfo are “don’t care” for read-type commands, and nothing is written for write-type commands.

4.4.2 Posting Semantics

Table 22 below summarizes the posting semantics for write-type commands. WRNP and WRC are always non-posted; a DVA response indicates that the write was committed and an ERR response indicates that the write was not committed (an error occurred along the write path).

WR and BCST commands may follow a posted or non-posted model. If the OCP interface is configured to not send a completion response (`writeresp_enable` is set to 0), the write is posted upon command acceptance and is considered to be posted early. When `writeresp_enable` is set to 1, the system designer decide where along the write path the posting point is. The completion response (either DVA or ERR) is then generated from the posting point. The non-posted model has the same semantics as WRNP.

Table 22 Write Posting Semantics

Write Command	**writeresp_enable**	
	0	**1**
WR, BCST	Posted early	Posted or Non-posted
WRNP, WRC	Non-posted	Non-posted

4.4.3 Transaction Completion, Transaction Commitment

It is useful to distinguish between “commitment” of a transaction and the “completion” of a transaction. A transaction is “committed” when the transaction finishes or completes at the final target.

In cases where the completion response is sent by the slave or target after commitment, the completion response is a guarantee of transaction commitment. With a posted write model, however, the posted write completion response may be received at the master before the write commitment.

Thus, the OCP completion response implies commitment for all transactions except writes with a posted write model (e.g., WR or BCST with early posting). For posted writes, there is no relationship between commitment and completion.

4.5 Endianness

An OCP interface by itself is inherently endian-neutral. Data widths must match between master and slave, addressing is on an OCP word granularity, and byte enables are tied to byte lanes (data bits) without tying the byte lanes to specific byte addresses.

The issue of endianness arises in the context of multiple OCP interfaces, where the data widths of the initiator of a request and the final target of that request do not match. Examples are a bridge or a more general interconnect used to connect OCP-based cores.

When the OCP interfaces differ in data width, the interconnect must associate an endianness with each transfer. It does so by associating byte lanes and byte enables of the wider OCP with least-significant word address bits of the narrower OCP. Packing rules, described in Section 4.6.1.2 on page 54 must also be obeyed for bursts.

OCP interfaces can be designated as little, big, both, or neutral with respect to endianness. This is specified using the protocol parameter `endian` described in Section 4.9.1.6 on page 62. A core that is designated as both typically represents a device that can change endianness based upon either an internal configuration register or an external input. A core that is designated as neutral typically represents a device that has no inherent endianness. This indicates that either the association of an endianness is arbitrary (as with a memory, which traditionally has no inherent endianness) or that the device only works with full-word quantities (when `byteen` and `mdatabyteen` are set to 0).

When all cores have the same endianness, an interconnect should match the endianness of the attached cores. The details of any conversion between cores of different endianness is implementation-specific.

4.6 Burst Definition

A *burst* is a set of transfers that are linked together into a transaction having a defined address sequence and number of transfers. There are three general categories of bursts:

Imprecise bursts
: Request information is given for each transfer. Length information may change during the burst.

Precise bursts
: Request information is given for each transfer, but length information is constant throughout the burst.

Single request / multiple data bursts (also known as packets)
: Also a precise burst, but request information is given only once for the entire burst.

To express bursts on the OCP interface, at least the address sequence and length of the burst must be communicated, either directly using the MBurstSeq and MBurstLength signals, or indirectly through an explicit constant tie-off as described in Section 4.9.5.1 on page 66.

A single (non-burst) request on an OCP interface with burst support is encoded as a request with any legal burst address sequence and a burst length of 1.

The ReadEx, ReadLinked, and WriteConditional commands can not be used as part of a burst. The unlocking Write or WriteNonPost command associated with a ReadEx command also can not be used as part of a burst.

4.6.1 Burst Address Sequences

The relationship of the MBurstSeq encodings and corresponding address sequences are shown in Table 23. The table also indicates whether a burst sequence type is packing or not, a concept discussed on page 54.

Table 23 Burst Address Sequences

Mnemonic	Name	Address Sequence	Packing
BLCK	2D block	see below for definition	yes
DFLT1	custom (packed)	user-specified	yes
DFLT2	custom (not packed)	user-specified	no
INCR	incrementing	incremented by OCP word size each transfer*	yes
STRM	streaming	constant each transfer	no
UNKN	unknown	none specified	implementation specific
WRAP	wrapping	like INCR, except wrap at address boundary aligned with MBurstLength * OCP word size	yes
XOR	exclusive OR	see below for definition	yes

* Bursts must no wrap around the OCP address size.

The address sequence for two-dimensional block bursts is as follows. The address sequence begins at the provided address and proceeds through a set of MBlockHeight subsequences, each of which follows the normal INCR address sequence for MBurstLength transfers. The starting address for each following subsequence is the starting address of the prior subsequence plus MBlockStride.

The address sequence for exclusive OR bursts is as follows. Let BASE be the lowest byte address in the burst, which must be aligned with the total burst size. Let FIRST_OFFSET be the byte offset (from BASE) of the first transfer in the burst. Let CURRENT_COUNT be the count of the current transfer in the burst, starting at 0. Let WORD_SHIFT be the logarithm base-two of the OCP word size in bytes. Then the current address of the transfer is BASE | (FIRST_OFFSET ^ (CURRENT_COUNT << WORD_SHIFT)).

The burst address sequence UNKN is used if the address sequence is not statically known for the burst. Single request/multiple data bursts (described on page 55) with a burst address sequence of UNKN are illegal. In contrast, the DFLT1 and DFLT2 address sequences are known, but are core or system specific.

The burst address sequences BLCK, WRAP, and XOR can only be used for precise bursts. Additionally, the burst sequences WRAP and XOR can only have a power-of-two burst length and a data width that is a power-of-two number of bytes.

Not all masters and slaves need to support all burst sequences. A separate protocol parameter described in Section 4.9.1.2 on page 59 is provided for each burst sequence to indicate support for that burst sequence.

4.6.1.1 Byte Enable Restrictions

Burst address sequences STRM and DFLT2 must have at least one byte enable asserted for each transfer in the burst. Bursts with the STRM address sequence must have the same byte enable pattern for each transfer in the burst.

4.6.1.2 Packing

Packing allows the system to make use of the burst attributes to improve the overall data transfer efficiency in the face of multiple OCP interfaces of different data widths. For example, if a bridge is translating a narrow OCP to a wide OCP, it can aggregate (or pack) the incoming narrow transfers into a smaller number of outgoing wide transfers. Burst address sequences are classified as either packing or not packing.

For burst address sequences that are packing, the conversion between different OCP data widths is achieved through aggregation or splitting. Narrow OCP words are collected together to form a wide OCP word. A wide OCP word is split into several narrow OCP words. The byte-specific portion of MDataInfo and SDataInfo is aggregated or split with the data. The transfer-specific portion of MDataInfo and SDataInfo is unaffected. The packing and unpacking order depends on endianness as described on page 51.

For burst address sequences that are not packing, conversion between different OCP data widths is achieved using padding and stripping. A narrow OCP word is padded to form a wide OCP word with only the relevant byte enables turned on. A wide OCP word is stripped to form a narrow OCP word. The byte-specific portion of MDataInfo and SDataInfo is zero-padded or stripped with the data. The transfer-specific portion of MDataInfo and SDataInfo is unaffected. Width conversion can be performed reliably only if the wide OCP interface has byte enables associated with it. For wide to narrow conversion the byte enables are restricted to a subset that can be expressed within a single word of the narrow OCP interface.

Since the address sequence of DFLT1 is user-specified, the behavior of DFLT1 bursts through data width conversion is implementation-specific.

4.6.2 Burst Length, Precise and Imprecise Bursts

The MBurstLength field indicates the number of transfers in the burst.

Precise bursts (MBurstPrecise set to 1)

MBurstLength must be held constant throughout the burst, so the exact burst length can be obtained from the first transfer. A precise burst is completed by the transfer of the correct number of OCP words. Precise bursts are recommended over imprecise bursts because they allow for increased hardware optimization.

Imprecise bursts (MBurstPrecise set to 0)

MBurstLength can change throughout the burst and indicates the current best guess of the number of transfers left in the burst (including the current one). An imprecise burst is completed by an MBurstLength of 1.

4.6.3 Constant Fields in Bursts

MCmd, MAddrSpace, MConnID, MBurstPrecise, MBurstSingleReq, MBurstSeq, MAtomicLength, MBlockHeight, MBlockStride, and MReqInfo must all be held steady by the master for every transfer in a burst, regardless of whether the burst is precise or imprecise. If possible, slaves should hold SRespInfo steady for every transfer in a burst.

4.6.4 Atomicity

When interleaving requests from different initiators on the way to or at the target, the master uses MAtomicLength to indicate the number of OCP words within a burst that must be kept together as an atomic quantity. If MAtomicLength is greater than the actual length of the burst, the atomicity requirement ends with the end of the burst. Specifying atomicity requirements explicitly is especially useful when multiple OCP interfaces are involved that have different data widths.

For master cores, it is best to make the atomic size as small as required and, if possible, to keep the groups of atomic words address-aligned with the group size.

4.6.5 Single Request / Multiple Data Bursts (Packets)

MBurstSingleReq specifies whether a burst can be communicated using a single request / multiple data protocol. When MBurstSingleReq is 0, each request has a single data word associated with it. When MBurstSingleReq is 1, each request may have multiple data words associated with it, according to the values of MBurstLength and MBlockHeight. MBurstSingleReq may be set to 1 only if MBurstPrecise is set to 1. In addition, if any write-type commands are enabled, `datahandshake` must be set to 1.

When MBurstSingleReq is set to 1, write type transfers have MBurstLength * height datahandshake phases per request[1]; while read-type transfers have MBurstLength * height response phases per request as shown in Table 21 on page 42. The height is MBlockHeight for BLCK address sequences, and 1 for all others.

For write type transfers when MBurstSingleReq is set to 1 and the MDataByteEn field is present, that field in each data transfer phase specifies the partial word pattern for the phase. When MBurstSingleReq is set to 1 and the MDataByteEn field is not present, the MByteEn pattern of the request phase applies to all data transfer phases.

[1] Additionally, there is a single response phase for WRNP write type while the WR and BCST types have this phase only if `writeresp_enable` is set to 1. Note that WRC write type is not allowed in a burst.

For read type transfers when MBurstSingleReq is set to 1, the MByteEn field specifies the byte enable pattern that is applied to all data transfers in the burst.

4.6.6 MReqLast, MDataLast, SRespLast

Optional signals MReqLast, MDataLast, and SRespLast provide redundant information that indicates the last request, datahandshake, and response phase in a burst, respectively. These signals are provided as a convenience to the recipient of the signal. To avoid separate counting mechanisms to track bursts, cores that have the information available internally are encouraged to provide it at the OCP interface.

MReqLast is 0 for all request phases in a burst except the last one. MReqLast is 1 for the last request phase in a burst, for single request / multiple data bursts, and for single requests.

MDataLast is 0 for all datahandshake phases in a burst except the last one. MDataLast is 1 for the last datahandshake phase in a burst and for the only datahandshake phase of a single request.

SRespLast is 0 for all response phases in a burst except the last one. SRespLast is 1 for the last response phase in a burst, for the response to a write-type single request / multiple data burst, and for the response to a single request.

4.6.7 MReqRowLast, MDataRowLast, SRespRowLast

For the BLCK burst address sequence, the optional signals MReqRowLast, MDataRowLast, and SRespRowLast identify the last request, datahandshake, and response phase in a row. The last phase in a burst is always considered the last phase in a row, and BLCK burst sequences reach the end of a row every MBurstLength phases (at the end of each INCR sub-sequence, see page 68). To avoid separate counting mechanisms needed to track BLCK burst sequences, cores that have the end of row information available should provide it at the OCP interface.

For all request phases in a non-BLCK burst except the last one, MReqRowLast is 0. MReqRowLast is 0 for every request phase in a BLCK burst sequence that is not an integer multiple of MBurstLength. MReqRowLast is 1 for:

- The last request phase in a burst including:
  - The only request phase in a single request/multiple data burst
  - The only request phase in a single word request
- Every request phase in a BLCK burst sequence that is an integer multiple of MBurstLength

For all datahandshake phases in a non-BLCK burst except the last one, MDataRowLast is 0. MDataRowLast is 0 for every datahandshake phase in a BLCK burst sequence that is not an integer multiple of MBurstLength. MDataRowLast is 1 for:

- The last datahandshake phase in a burst including the only datahandshake phase of a single word request
- Every datahandshake phase in a BLCK burst sequence that is an integer multiple of MBurstLength

For all response phases in a non-BLCK burst except the last one, SRespRowLast is 0. SRespRowLast is 0 for every response phase in a BLCK burst sequence that is not an integer multiple of MBurstLength. SRespRowLast is 1 for:

- The last response phase in a burst including:
  - The only response phase in a write-type single request/multiple data burst
  - The only response phase in a single word request
- Every response phase in a BLCK burst sequence that is an integer multiple of MBurstLength

4.7 Tags

Tags allow out-of-order return of responses and out-of-order commit of write data.

A master drives a tag on MTagID during the request phase. The value of the tag is determined by the master and may or may not convey meaning beyond ordering to the slave. For write transactions with data handshake enabled, the master repeats the same tag on MDataTagID during the datahandshake phase. For read transactions and writes with responses the slave returns the tag of the corresponding request on STagID while supplying the response. The same tag must be used for an entire transaction.

4.7.1 Ordering Restrictions

The sequence of requests by the master determines the initial ordering of tagged transactions. For tagged write transactions with datahandshake enabled, the datahandshake phase must observe the same order as the request phase. The master cannot interleave requests or datahandshake phases from different tags belonging to the same thread within a transaction.

Tag values can be re-used for multiple outstanding transactions. Slaves are responsible for committing write data and sending responses for multiple transactions that have the same tag, in order.

Responses that are part of the same transaction must stay together, up to the `tag_interleave_size` (see Section 4.9.1.7 on page 62). Beyond the `tag_interleave_size`, responses with different tags can be interleaved. This allows for blocks of responses corresponding to `tag_interleave_size` from one burst to be interleaved with blocks of responses from other bursts.

Responses with different tags can be returned in any order for all commands that have responses. Responses with the same tag must remain in order with respect to one another. Responses to requests that are issued with MTagInOrder asserted are also never reordered with respect to one another. The value returned on STagInOrder with the slave's response must match the value provided on MTagInOrder with the master's request.

Commitment of transactions with overlapping addresses (as determined by MAddrSpace, MAddr, MByteEn [or MDataByteEn, if applicable]) on different (or the same) tags within a thread is always in order. Note, however, that the completion responses for such transactions with different tag ids may be reordered.

4.8 Threads and Connections

When using multiple threads, it is possible to support concurrent activity, and out-of-order completion of transfers. All transfers within a given thread must either remain strictly ordered or follow the tag ordering rules, but there are no ordering rules for transfers that are in different threads. Mapping of individual requests and responses to threads is handled through the MThreadID and SThreadID fields respectively. If datahandshake has been enabled when multiple threads are present, there must also be an MDataThreadID field to annotate the datahandshake phase. If datahandshake is set to 1 and the datahandshake phase has blocking flow control (as described on page 61), the order of datahandshake phases must follow the order of request phases across all threads. If the datahandshake phase has no flow control or non-blocking flow control, the request order and datahandshake order are independent across threads.

The use of thread IDs allows two entities that are communicating over an OCP interface to assign transfers to particular threads. If one of the communicating entities is itself a bridge to another OCP interface, the information about which transfers are part of which thread must be maintained by the bridge, but the actual assignment of thread IDs is done on a per-OCP-interface basis. There is no way for a slave on the far side of a bridge to extract the original thread ID unless the slave design comprehends the characteristics of the bridge.

Use connections whenever source thread information about a request must be sent end-to-end from master to slave. Any bridges in the path between the end-to-end partners preserve the connection ID, even as thread IDs are re-assigned on each OCP interface in the path. The MConnID field transfers the connection ID during the request phase. Since this establishes the mapping onto a thread ID, the other phases do not require a connection ID but are unambiguous with only a thread ID.

The SThreadBusy, SDataThreadbusy, and MThreadBusy signals are used to indicate that a particular thread is busy. The protocol parameters `sthreadbusy_exact`, `sdatathreadbusy_exact`, and `mthreadbusy_exact` can be used to force precise semantics for these signals and assure that a multi-threaded OCP interface never blocks. For more information, see Section 4.3.2.4 on page 44.

4.9 OCP Configuration

This section describes configuration options that control interface capabilities.

4.9.1 Protocol Options

4.9.1.1 Optional Commands

Not all devices support all commands. Each command in Table 24 has an enabling parameter to indicate if that command is supported.

Table 24 Command Enabling Parameters

Command	**Parameter**
Broadcast	broadcast_enable
Read	read_enable
ReadEx	readex_enable
ReadLinked and WriteConditional	rdlwrc_enable
Write	write_enable
WriteNonPost	writenonpost_enable

The following conditions apply to command support:

- A master with one of these options set to 0 must not generate the corresponding command.
- A slave with one of these options set to 0 cannot service the corresponding command.
- At least one of the command enables must be set to 1.
- If any read-type command is enabled, or if WRNP is enabled, or if `writeresp_enable` is set to 1, `resp` must be set to 1.
- If `readex_enable` is set to 1, `write_enable` or `writenonpost_enable` must be set to 1.

4.9.1.2 Optional Burst Sequences

Not all masters and slaves need to support all burst address sequences. Table 25 lists the parameter for each burst sequence. A master with the parameter set to 1 may generate the corresponding burst sequence. A slave with the parameter set to 1 can service the corresponding burst sequence. If MBurstSeq is disabled and tied off to a constant value, the corresponding burst sequence parameter must be enabled and all others disabled. If MBurstSeq is enabled at least one of the burst sequence parameters must be enabled.

Table 25 Burst Sequence Parameters

Burst Sequence	Parameter
BLCK	burstseq_blck_enable
DFLT1	burstseq_dflt1_enable
DFLT2	burstseq_dflt2_enable
INCR	burstseq_incr_enable
STRM	burstseq_strm_enable
UNKN	burstseq_unkn_enable
WRAP	burstseq_wrap_enable
XOR	burstseq_xor_enable

The BLCK burst sequence can only be enabled if both MBlockHeight and MBlockStride are included in the interface or tied off to non-default values. For additional burst information, see Section 4.6 on page 52.

4.9.1.3 Byte Enable Patterns

Not all devices support all allowable byte enable patterns. A `force_aligned` parameter limits byte enable patterns on MByteEn and MDataByteEn to be power-of-two in size and aligned to that size. The byte enable pattern of all 0s is explicitly included in the legal force aligned patterns.

- A master with this option set to 1 must not generate any byte enable patterns that are not force aligned.
- A slave with this option set to 1 cannot handle any byte enable patterns that are not force aligned.

`force_aligned` can be set to 1 only if `data_wdth` is set to a power-of-two value.

4.9.1.4 Burst Alignment

The `burst_aligned` parameter provides information about the length and alignment of INCR bursts issued by a master and can be used to optimize the system. Setting `burst_aligned` to 1 requires all INCR bursts to:

- Have an exact power-of-two number of transfers
- Have their starting address aligned with their total burst size
- Be issued as precise bursts.

The `burst_aligned` parameter does not apply to the INCR subsequences within BLCK burst sequences.

4.9.1.5 Flow Control Options

To permit the SThreadBusy and MThreadBusy signals to guarantee a non-blocking, multi-threaded OCP interface, the `sthreadbusy_exact` and `mthreadbusy_exact` parameters require strict semantics. See Section 4.3.2.4 on page 44 for a definition of these parameters. Table 26 describes the legal combinations of phase handshake signals.

Table 26 Request Phase Without Datahandshake

cmdaccept	**sthreadbusy**	**sthreadbusy_exact**	**Explanation**
0	0	0	Legal: no flow control
0	0	1	Illegal: sthreadbusy_exact must be 0 when sthreadbusy is 0
0	1	0	Illegal: no real flow control
0	1	1	Legal: non-blocking flow control
1	0	0	Legal: blocking flow control
1	0	1	Illegal: sthreadbusy_exact must be 0 when sthreadbusy is 0
1	1	0	Legal: blocking flow control with hints
1	1	1	Illegal: since SCmdAccept is present flow control cannot be exact

When datahandshake is set to 1, the preceding rules for `cmdaccept`, `sthreadbusy`, and `sthreadbusy_exact` also apply to `dataaccept`, `sdatathreadbusy`, and `sdatathreadbusy_exact`. In addition, blocking and non-blocking flow control must not be mixed for the request and datahandshake phase. A phase using no flow control can be mixed with phases using either blocking or non-blocking type flow control. The legal combinations are shown in Table 27.

Table 27 Request Phase with Datahandshake

		Datahandshake Phase Flow Control		
		None	Blocking	Non-blocking
Request Phase Flow Control	None	Legal	Legal[1]	Legal
	Blocking	Legal[2]	Legal	Illegal
	Non-blocking	Legal	Illegal	Legal[3]

[1] Only legal if reqdata_together is set to 0.

[2] Only legal if reqdata_together is set to 0. In addition the master must not assert the datahandshake phase until after the associated request phase has been accepted.

[3] Only legal if sthreadbusy_pipelined and sdatathreadbusy_pipelined are both set to the same value.

The preceding rules for the request phase using `cmdaccept`, `sthreadbusy`, and `sthreadbusy_exact` also apply to the response phase for `respaccept`, `mthreadbusy`, and `mthreadbusy_exact`.

4.9.1.6 Endianness

The `endian` parameter specifies the endianness of a core. The behavior of each endianness choice is summarized in Table 28.

Table 28 Endianness

Endianness	Description
little	core is little-endian
big	core is big-endian
both	core can be either big or little endian, depending on its static or dynamic configuration (e.g. CPUs)
neutral	core has no inherent endianness (e.g. memories, cores that deal only in OCP words)

As far as OCP is concerned, little endian means that lower addresses are associated with lower numbered data bits (byte lanes), while big endian means that higher addresses are associated with lower numbered data bits (byte lanes). This becomes significant when packing is concerned (see Section 4.6.1.2 on page 54). In addition, for non-power-of-2 data widths, tie-off padding is always added at the most significant end of the OCP word. See Section 4.5 on page 51 for additional information.

4.9.1.7 Burst Interleaving with Tags

When tags > 1, the `tag_interleave_size` parameter limits the interleaving permitted for responses with burst sequences. The parameter indicates the size of a power-of-two, aligned data block (in OCP words) within which there can be no interleaving of responses from packing bursts with different tags.

`tag_interleave_size` = 0
: No interleaving of responses between any burst sequence responses with different tags is permitted.

`tag_interleave_size` = 1
: Interleaving is permitted at OCP word granularity and is unrestricted.

`tag_interleave_size` > 1
: Interleaving of non-packing burst sequence responses is not limited by `tag_interleave_size`. Interleaving of packing burst responses is allowed whenever the next response would cross the data block boundary, regardless of whether a full data block of responses has been returned.

Restricting interleaving opportunities for packing burst responses reduces the storage required for width conversion when multiple tags are present. For slaves, enabling the parameter restricts the aligned boundary within which the slave interleaves responses with different tags. For masters, the parameter gives the minimum aligned boundary at which the master can tolerate interleaving of responses with different tags.

4.9.2 Phase Options

The `datahandshake` parameter allows write data to have a handshake interface separate from the request group.

Datahandshake

If `datahandshake` is set to 1, the MDataValid and optionally the SDataAccept signals are added to the OCP interface, a separate datahandshake phase is added, and the MData and MDataInfo fields are moved from the request group to the datahandshake group. Datahandshake can be set to 1 only if at least one write-type command is enabled.

Request and Data Together

While datahandshake is required for OCP interfaces that are capable of communicating single request / multiple data bursts, a fully separated datahandshake may be overkill for some cores. The parameter `reqdata_together` is used to specify that the request and datahandshake phases of the first transfer in a single request, multiple data (SRMD) write-type burst begin and end together.

A master with reqdata_together set to 1 must present the request and first write data word in the same cycle and can expect that the slave will accept them together. If sthreadbusy_exact and sdatathreadbusy_exact are both set to 1 and sthreadbusy_pipelined and sdatathreadbusy_pipelined are both set to 0, then a request and first write data can be presented only when both SThreadBusy and SDataThreadBusy for the corresponding thread are 0 on that cycle. If sthreadbusy_exact and sdatathreadbusy_exact are both set to 1 and sthreadbusy_pipelined and sdatathreadbusy_pipelined are both set to 1, then a request and first write data can be presented only on cycle i when both SThreadBusy and SDataThreadBusy for the corresponding thread are 0 during the prior cycle, i.e., cycle (i-1).

A slave with reqdata_together set to 1 must accept the request and first write data word in the same cycle and can expect that they will be presented together.

The parameter `reqdata_together` can only be set to 1 if `burstsinglereq` is set to 1, or `burstsinglereq` is set to 0 and MBurstSingleReq is tied off to 1.

If both `reqdata_together` and `burstsinglereq` are set to 1, the master must present the request and associated write data word together for each transfer in any multiple request / multiple data writes it issues. The slave must accept both request and write data together for all such transfers.

Write Responses

- Writes which follow a non-posted model, i.e., WRNP and WRC, always have a write response. For this case, `resp` must be set to 1.
- For writes which follow a posted model, i.e., WR and BCST: if responses are not enabled on writes (`writeresp_enable` set to 0), then they complete on command acceptance.

4.9.3 Signal Options

The configuration parameters described in Section 3.4 on page 31, not only configure the corresponding signal into the OCP interface, but also enable the function. For example, if the `burstseq` and `burstlength` parameters are enabled the MBurstSeq and MBurstLength fields are added and the interface also supports burst extensions as described in Section 4.6 on page 52.

4.9.4 Minimum Implementation

A minimal OCP implementation must support at least the basic OCP dataflow signals. OCP-interoperable masters and slaves must support the command type Idle and at least one other command type.

If the SResp field is present in the OCP interface, OCP-interoperable masters and slaves must support response types NULL and DVA. The ERR response type is optional and should only be included if the OCP-interoperable slave has the ability to report errors. All OCP masters must be able to accept the ERR response. If `rdlwrc_enable` is set to 1, the FAIL response type must be supported by OCP masters and slaves.

4.9.5 OCP Interface Interoperability

Two devices connected together each have their own OCP configuration. The two interfaces are only interoperable (allowing the two devices to be connected together and communicate using the OCP protocol semantics) if they are interoperable at the core, protocol, phase, and signal levels.

1. At the core level:
   - One interface must act as master and the other as slave.
   - If system signals are present, one interface must act as core and the other as system.
2. At the protocol level, the following conditions determine interface interoperability:
   - If the slave has `read_enable` set to 0, the master must have `read_enable` set to 0, or it must not issue Read commands.
   - If the slave has `readex_enable` set to 0, the master must have `readex_enable` set to 0, or it must not issue ReadEx commands.
   - If the slave has `rdlwrc_enable` set to 0, the master must have `rdlwrc_enable` set to 0, or it must not issue either ReadLinked or WriteConditional commands.
   - If the slave has `write_enable` set to 0, the master must have `write_enable` set to 0, or it must not issue Write commands.
   - If the slave has `writenonpost_enable` set to 0, the master must have `writenonpost_enable` set to 0, or it must not issue WriteNonPost commands.

- If the slave has `broadcast_enable` set to 0, the master must have `broadcast_enable` set to 0, or it must not issue Broadcast commands.
- If the slave has `burstseq_blck_enable` set to 0, the master must have `burstseq_blck_enable` set to 0, or it must not issue BLCK bursts.
- If the slave has `burstseq_incr_enable` set to 0, the master must have `burstseq_incr_enable` set to 0, or it must not issue INCR bursts.
- If the slave has `burstseq_strm_enable` set to 0, the master must have `burstseq_strm_enable` set to 0, or it must not issue STRM bursts.
- If the slave has `burstseq_dflt1_enable` set to 0, the master must have `burstseq_dflt1_enable` set to 0, or it must not issue DFLT1 bursts.
- If the slave has `burstseq_dflt2_enable` set to 0, the master must have `burstseq_dflt2_enable` set to 0, or it must not issue DFLT2 bursts.
- If the slave has `burstseq_wrap_enable` set to 0, the master must have `burstseq_wrap_enable` set to 0, or it must not issue WRAP bursts.
- If the slave has `burstseq_xor_enable` set to 0, the master must have `burstseq_xor_enable` set to 0, or it must not issue XOR bursts.
- If the slave has `burstseq_unkn_enable` set to 0, the master must have `burstseq_unkn_enable` set to 0, or it must not issue UNKN bursts.
- If the slave has `force_aligned`, the master has `force_aligned` or it must limit itself to aligned byte enable patterns.
- Configuration of the `mdatabyteen` parameter is identical between master and slave.
- If the slave has `burst_aligned`, the master has `burst_aligned` or it must limit itself to issue all INCR bursts using `burst_aligned` rules.
- If the interface includes SThreadBusy, the `sthreadbusy_exact` and `sthreadbusy_pipelined` parameters are identical between master and slave.
- If the interface includes MThreadBusy, the `mthreadbusy_exact` and `mthreadbusy_pipelined` parameter are identical between master and slave.
- If the interface includes SDataThreadBusy, the `sdatathreadbusy_exact` and `sdatathreadbusy_pipelined` parameters are identical between master and slave.

- All combinations of the endian parameter between master and slave are interoperable as far as the OCP interface is concerned. There may be core-specific issues if the endianness is mismatched.
- If tags > 1, the master's tag_interleave_size is smaller than or equal to the slave's tag_interleave_size.

3. At the phase level the two interfaces are interoperable if:
   - Configuration of the datahandshake parameter is identical between master and slave.
   - Configuration of the writeresp_enable parameter is identical between master and slave. Otherwise, the master only issues the write commands WriteNonPost and WriteConditional.
   - Configuration of the reqdata_together parameter is identical between master and slave.
4. At the signal level, two interfaces are interoperable if:
   - data_wdth is identical for master and slave, or if one or both data_wdth configurations are not a power-of-two, if that data_wdth rounded up to the next power-of-two is identical for master and slave.
   - The master and slave both have mreset or sreset set to 1.
   - If the master has mreset set to 1, the slave has mreset set to 1.
   - If the slave has sreset set to 1, the master has sreset set to 1.
   - The value of connection is identical for master and slave, or if ConnectCap is tied off to logic 0 on the side with connection set to 1.
   - Both master and slave have tags set to ≥1 or if only one core's tags parameter is set to 1, the other core behaves as though MTagInOrder were asserted for every request.
   - The tie-off rules, described in the next section are observed for any mismatch at the signal level for fields other than MData and SData.

4.9.5.1 Signal Mismatch Tie-off Rules

There are two types of signal mismatches: both interfaces may have configured the signal, but to different widths or only one interface may have configured the signal.

Width mismatch for all fields other than MData and SData is handled through a set of signal tie-off rules. The rules state whether a master and slave that are mismatched in a particular field width configuration are interoperable, and if so how to connect them by tying off the mismatched signals.

If there is a width mismatch between master and slave for a particular signal configuration the following rules apply:

- If there are more outputs than inputs (the driver of the field has a wider configuration than the receiver of the field) the low-order output bits are connected to the input bits, and the high-order output bits are lost. The interfaces are interoperable if the sender of the field explicitly limits itself to encodings that only make use of the bits that are within the configuration of the receiver of the field.
- If there are more inputs than outputs (the driver of field has a narrower configuration than the receiver of the field) the low-order input bits are connected to the output bits, and the high-order input bits are tied to logical 0. The interfaces are always interoperable, but only a portion of the legal encodings are used on that field.

If one of the cores has a signal configured and the other does not, the following rules apply:

- If the core that would be the driver of the field does not have the field configured, the input is tied off to the constant specified in the driving core's configuration, or if no constant tie-off is specified, to the default tie-off constant (see Table 16 on page 31). The interfaces are interoperable if the encodings supported by the receiver's configuration of the field include the tie-off constant.
- If the core that would be the receiver of the field does not have the field configured, the output is lost. The receiver of the signal must behave as though in every phase it were receiving the tie-off constant specified in its configuration, or lacking a constant tie-off, the default tie-off constant (see Table 16 on page 31). The interfaces are interoperable if the driver of the signal can limit itself to only driving the tie-off constant of the receiver.
- If only one core has the EnableClk signal configured, the interfaces are interoperable only when the EnableClk signal is asserted, matching the tie-off value of the core that has `enableclk`=0.

If neither core has a signal configured, the interfaces are interoperable if both cores have the same tie-off constant, where the tie-off constant is either explicitly specified, or if no constant tie-off is specified explicitly, is the default tie-off (see Table 16 on page 31).

While the tie-off rules allow two mismatched cores to be connected, this may not be enough to guarantee meaningful communication, especially when core-specific encodings are used for signals such as MReqInfo.

As the previous rules suggest, specifying core specific tie-off constants that are different than the default tie-offs for a signal (see Table 16 on page 31) makes it less likely that the core will be interoperable with other cores.

4.9.6 Configuration Parameter Defaults

To assure OCP interface interoperability between a master and a slave requires complete knowledge of the OCP interface configuration of both master and slave. This is achieved by a combination of (a) requiring some parameters to be explicitly specified for each core, and (b) defining defaults that are used when a parameter is not explicitly specified for a core.

Table 29 lists all configuration parameters. For parameters that do not need to be specified, a default value is listed, which is used whenever an explicit parameter value is not specified. Certain parameters are always required in certain configurations, and for these no default is specified.

Table 29 Configuration Parameter Defaults

Type	**Parameter**	**Default**
Protocol	broadcast_enable	0
	burst_aligned	0
	burstseq_blck_enable	0
	burstseq_dflt1_enable	0
	burstseq_dflt2_enable	0
	burstseq_incr_enable	1
	burstseq_strm_enable	0
	burstseq_unkn_enable	0
	burstseq_wrap_enable	0
	burstseq_xor_enable	0
	endian	little
	force_aligned	0
	mthreadbusy_exact	0
	rdlwrc_enable	0
	read_enable	1
	readex_enable	0
	sdatathreadbusy_exact	0
	sthreadbusy_exact	0
	tag_interleave_size	1
	write_enable	1
	writenonpost_enable	0
Phase	datahandshake	0
	reqdata_together	0
	writeresp_enable	0

Type	**Parameter**	**Default**
Signal (Dataflow)	addr	1
	addr_wdth	No default - must be explicitly specified if addr is set to 1
	addrspace	0
	addrspace_wdth	No default - must be explicitly specified if addrspace is set to 1
	atomiclength	0
	atomiclength_wdth	No default - must be explicitly specified if atomiclength is set to 1
	blockheight	0
	blockheight_wdth	No default - must be explicitly specified if blockheight is set to 1
	blockstride	0
	blockstride_wdth	No default - must be explicitly specified if blockstride is set to 1
	burstlength	0
	burstlength_wdth	No default - must be explicitly specified if burstlength is set to 1
	burstprecise	0
	burstseq	0
	burstsinglereq	0
	byteen	0
	cmdaccept	1
	connid	0
	connid_wdth	No default - must be explicitly specified if connid is set to 1
	dataaccept	0
	datalast	0
	datrowalast	0
	data_wdth	No default - must be explicitly specified if mdata or sdata is set to 1
	enableclk	0
	mdata	1
	mdatabyteen	0
	mdatainfo	0

Type	Parameter	Default
Signal (Dataflow)	mdatainfo_wdth	No default - must be explicitly specified if mdatainfo is set to 1
	mdatainfobyte_wdth	
	mthreadbusy	0
	mthreadbusy_pipelined	0
	reqinfo	0
	reqinfo_wdth	No default - must be explicitly specified if reqinfo is set to 1
	reqlast	0
	reqrowlast	0
	resp	1
	respaccept	0
	respinfo	0
	respinfo_wdth	No default - must be explicitly specified if respinfo is set to 1
	resplast	0
	resprowlast	0
	sdata	1
	sdatainfo	0
	sdatainfo_wdth	No default - must be explicitly specified if sdatainfo is set to 1
	sdatainfobyte_wdth	
	sdatathreadbusy	0
	sdatathreadbusy_pipelined	0
	sthreadbusy	0
	sthreadbusy_pipelined	0
	tags	1
	taginorder	0
	threads	1

Type	Parameter	Default
Signal (Sideband)	connection	0
	control	0
	controlbusy	0
	control_wdth	No default—must be explicitly specified if control is set to 1
	controlwr	0
	interrupt	0
	merror	0
	mflag	0
	mflag_wdth	No default—must be explicitly specified if mflag is set to 1
	mreset	No default—must be explicitly specified
	serror	0
	sflag	0
	sflag_wdth	No default - must be explicitly specified if sflag is set to 1
	sreset	No default - must be explicitly specified
	status	0
	statusbusy	0
	statusrd	0
	status_wdth	No default - must be explicitly specified if status is set to 1
Signal (Test)	clkctrl_enable	0
	jtag_enable	0
	jtagtrst_enable	0
	scanctrl_wdth	0
	scanport	0
	scanport_wdth	No default - must be explicitly specified if scanport is set to 1

5 OCP Coherence Extensions: Theory of Operation

There is an increasing need for SoC architectures to be built with masters which have caches. When shared memory locations are cached, there is a need for cache coherence.

The OCP Coherence Extensions are a parameterizable set of commands and signals that enable a SoC designer to build a wide variety of cache coherent architectures. The main features of the extensions are:

- OCP 3.0 with coherence extensions maintains full backward compatibility with OCP 2.2, making it possible to mix OCP 2.2 masters and slaves (that are by definition non-coherent) with coherent masters and slaves.
- Ability to build a wide range of cache-coherent architectures, from fully snoop-based to fully directory-based. Example architectures are presented in Chapter 13, beginning on page 255.
- The extensions support protocols based on MSI (and SI), MESI, and MOESI cache state combinations. Further, it is not necessary that all agents in a coherence domain enable the same set of cache states. Thus, a directory agent, for example, could be based on MSI while each of the other caching agents could be based on MSI or MESI.
- Includes support for coherence-aware masters.
- The extensions only support invalidation based protocols because of their preponderance over update based protocols. Within the gamut of invalidation based protocols, the extensions permit the use of either three-hop protocols or four-hop protocols. The Coherence Extensions are flexible, and permit protocol optimizations based on specific system requirements.

- Multiple coherence domains may coexist in a single architecture. However, only one cache line size is permitted in each coherence domain, and a coherence domain cannot share its coherence address space with any other coherence domain.

Note that an OCP coherent system permits the existence of "subsystem coherence," where a subsystem will maintain its own coherence framework and can act as a single OCP coherent agent to the system at the next hierarchical level. In fact, the subsystem coherence framework at the lower level could itself be composed of OCP agents. Hierarchical coherent subsystems are built in this manner.

5.1 Cache Coherence

A generally accepted definition of cache coherence[1], which is used in this specification, requires the following two conditions to be satisfied:

- A write must eventually be made visible to all master entities. This is accomplished in invalidate protocols by ensuring that a write is considered complete only after all the cached copies other than the one which is updated are invalidated.
- Writes to the same location must appear to be seen in the same order by all masters. Two conditions which ensure this are:
    - Writes to the same location by multiple masters are serialized, i.e., all masters see such writes in the same order. This can be accomplished by requiring that all invalidate operations for a location arise from a single point in the coherent slave and that the interconnect preserves the ordering of messages between two entities.
    - A read following a write to the same memory location is returned only after the write has completed.

5.2 Local View vs. System View

OCP 2.x is a point-to-point interface with one end being the master and the other end the slave. Thus all requests from the master agent are directed to the slave agent and all responses from the slave agent are directed to the master agent. Even when multiple agents are used in a system and a master agent needs to communicate with multiple slaves, the master agent acts as though it were communicating only with its slave (i.e., the slave agent in a single master–single slave configuration). This abstraction is made possible by a "bridge" or "interconnection" agent that acts as a slave agent for this master (and other masters). It also acts as a single master when a slave agent has to communicate with multiple masters in a system. Thus, the master and the slave agents do not need to carry explicit identifiers. Each master or slave agent maintains a "local" view even when it is part of a multi-agent system.

[1] See, for example, S. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66-76, December 1996.

Only the bridge or interconnect agent maintains a “system” view. This is a very convenient abstraction in SoC architectures that are loosely coupled with agents that are really hard or soft IPs.

With OCP 3.0 and the introduction of cache coherence, the “local” view is maintained for all master agents and all non-coherent slave agents. Only the home agent (introduced on page 79), which is a slave coherent agent, and the bridge agent need to maintain the “system view” abstraction. In this context, the “system view” refers to the explicit encoding of the master, slave, and forwarding agent identifiers (IDs) and the encoding of the address regions on an agent’s interface.

5.3 Coherent System Transactions

The notions of master, slave, and bridge entities are inherited from previous versions of the *Open Core Protocol Specification*. The master entity initiates requests and receives responses on its OCP port. The slave entity receives requests and generates responses on its OCP port. The bridge entity, if present, has one or more master and one or more slave ports.

In a coherent system, the slave may not be able to satisfy the response to a request directly since the latest copy of the requested address may reside in the coherent cache of another master and may not reside at its “home” memory. The coherence mechanism ensures that the latest copy is returned to the requester. It does this by “snooping” the set of coherent caches which has the latest data for this address, possibly updating the cache states, and finally returning the latest data to the original master. It can be inferred from this short background that a more sophisticated description of master/slave entities, ports, and address regions is needed for OCP to support cache coherence. The relevant definitions follow. (The reader is referred to standard text books and tutorials on cache coherence for a complete treatment.)

For convenience, the set of commands supported by OCP Rev. 2.2 are called **legacy commands**. The new set of commands introduced for the coherence extensions are called **coherent commands**.

5.3.1 Cache Line and Cache States

A **cache line** is the granularity of the data which participates in cache coherence. The cache line is byte addressable, has a power-of-two data size, and its address is always aligned to its line size. The current version of the OCP specification requires that all entities in the coherence domain have the same cache line size. It is expected that succeeding versions of the specification will relax this requirement. The first refinement will allow a different cache line size at each level of a hierarchical cache coherent system. Subsequent refinements will permit multiple cache line sizes in a coherence domain at the same level of the hierarchy. In such cases, the cache line sizes will be power-of-two multiples of the base size.

Note that if a master with coherent cache supports the critical word first feature, addresses of commands from the master may not be aligned to multiple of the cache line size, but the cache line boundaries should be aligned to the multiple of the cache line size by using WRAP or XOR burst address sequences.

A cache line in a master's coherent cache is always in one of several known **states**; the set of available states are summarized in Table 30. Some states are required and some are optional depending on the type of coherence protocol chosen.

Table 30 Cache Line State Definitions

Name	**Mnemonic**	**Description**	**OCP Compliance**
Invalid	I	Cache line not present in caching entity.	Required
Shared	S	Cache line is read only.	Required
Modified	M	Cache line owned exclusively by caching entity and modified by it. Memory copy is stale. All other caching entities have this line in I state.	Required[1]
Exclusive	E	Cache line is exclusively owned. Memory copy matches value. All other caching entities have this line in I state.	Optional
Owned	O	This entity has latest copy. Memory copy is stale. Other caching agents may have (latest) copy.	Optional

1. Instruction caches typically do not require this state.

5.3.2 Three Hop and Four Hop Protocols

The coherence extensions permit the implementation of both four hop and three hop protocols.

Four hop protocols are simpler to implement and are so called because the transfer of a cache line to a requester takes up to four protocol steps:

1. master's request to coherent slave;
2. slave's probe of other masters (which have coherent caches);
3. responses from masters, with one of them possibly providing the latest copy of the cache line to the slave; and,
4. the transfer of data from the slave to the requesting master.

Three hop protocols have better latency characteristics, but are more complicated to implement than four-hop protocols since they give rise to additional race conditions, deadlock, and starvation scenarios. The transfer of a modified cache line to a requester takes three protocol steps:

1. master's request to coherent slave;
2. slave's probe of other masters (which have coherent caches); and,
3. response from a master which has a cache line in the modified state directly to the requester (and a possible writeback of this data to the slave) with concurrent responses from all masters to the slave.

5.4 Address Space

The entire address space is partitioned into two non-overlapping parts: the **coherent address space** and the **non-coherent address space**. Each space is composed of regions which may be non-contiguous. The size of a region is implementation specific.

The coherent address space is kept coherent by OCP-based cache coherence protocols. Each access to this space is permitted only at cache line granularity (with optional byte enables). A read operation into this space always results in the latest completed write being read. A completed write to this space always results in this value being **visible** to all masters. Section 6.2.3.2 gives the semantics of the various types of reads and writes to this space. This space is typically accessed by coherent and coherence-aware masters (both cached and non-cached).

Coherent addresses are cacheable by coherent masters. If an address is cached, then the cache is coherent, i.e., it participates in cache coherence through the intervention port (see Section 5.5 on page 77). A coherence-aware master does not require a cache.

The non-coherent address space is not kept coherent by the OCP-based cache coherence protocol. Accesses to this space are at the OCP word granularity (with optional byte enables). Reads and writes to this space follow the semantics of legacy reads and writes.

Non-coherent addresses may be cached by a master. Such a cache does not participate in cache coherence and is not kept coherent by OCP.

5.5 Entities and Ports

A master with a coherent cache issues read and write commands that have different semantics from the read and write commands detailed in the *Open Core Protocol Specification, Release 2.2*. For example, such a master might issue a read with intent to modify the requested line (i.e., acquiring the latest copy, writing to it, and retaining it in its cache), a read only request, and a write back of a modified or dirty line when that line needs to be evicted from the cache. A master with a coherent cache is called a **coherent master** and issues requests on the **main port** of the OCP interface. The full set of main port commands and encodings are explained in Section 5.6. Legacy requests which are targeted to non-coherent address space are issued on the main port.

The coherent master also needs to satisfy requests from other coherent masters to “snoop” its cache lines and possibly respond either with the latest copy of the cache line or by giving up its ownership of the cache line. In OCP 3.0, these requests to the master and the corresponding responses are handled via the **intervention port**. The full set of intervention port commands are explained in Section 6.3.3.1 on page 115. A CPU is a typical example of an entity which would be an OCP coherent master. Section 5.9 presents an abstract model that illustrates the interaction between the main port and the intervention port, and between the coherent master and coherent slave.

A **coherence-aware master** does not have a coherent cache. For example, a DMA engine could be implemented as a coherence-aware master. A coherence-aware master has a main port but does not have an intervention port.

A coherence-aware master uses legacy commands. If the associated address is in the coherent region, then the coherent slave performs the appropriate actions depending on the request and the state of the associated cache line (as seen by the home agent), e.g., a coherent read returns the latest value written. (While processing a request from the coherence-aware master, a coherent slave may send intervention requests for the latest write to be returned, as discussed in detail in Section 13.3.4.1 on page 276.) If the associated address for a request is in the non-coherent address space, then the request has the semantics of a legacy request.

The **coherent slave** is the target of coherent request commands from all or any master in the coherence domain, depending on the type of coherence protocol used. It receives requests on its main port. Before it generates the response, it in turn sends requests on the intervention port to snoop all or a subset of the coherent caches in the coherence domain and may send a request to the memory controller. After it receives the responses to the intervention requests and/or from the memory controller, it finally sends the response to the original request on its main port. The coherent slave also ensures that writes to the same location appear to be seen in the same order by all the coherence masters. The coherent slave implementation usually takes the form of either a snoop- or directory-based scheme, as described in Section 5.11.

OCP 3.0 maintains full backward compatibility with OCP 2.2, that is, the command set for the coherence extensions is a superset of the OCP 2.2 command set. OCP 3.0 defines a new signal, MCohCmd, which, when set, indicates a coherent command. Non-coherent commands, which refer to the OCP 2.2 command set, do not have the MCohCmd bit set. In the rest of the document, the port defined by OCP 2.2 is referred to as the **legacy port**. Note that the main port defined by OCP 3.0 is capable of generating legacy transactions. Hence, a new design would not need both the legacy and main ports.

A master with a legacy port that only generates transactions to non-coherent space is called a **legacy master**. A slave with a legacy port is called a **legacy slave**.

Other terms used in this document include:

- The term **requester** is interchangeably used for a coherent master which initiates a request.
- The term **responder** is interchangeably used for a coherent master which responds to an intervention request.
- The term **owner** is interchangeably used for a coherent master which has a cache line in the M or the O state.
- The term **snoop** is interchangeably used with intervention.
- The term **home agent** is interchangeably used with coherent slave. Thus each coherent address has an associated home which is the coherent slave managing its coherency actions.

5.6 Commands

A master which can only issue legacy commands to noncoherent space is called a **legacy master**. A master which can issue only legacy commands to both coherent and noncoherent address spaces is called a **coherence aware master**. A master which can issue legacy commands to both non-coherent and coherent address spaces, and can issue coherent commands to coherent address spaces is called a **coherent master**.

It is illegal for coherent commands to be issued to non-coherent address spaces.

Legacy commands accessing the noncoherent address space are called “Legacy Commands to Noncoherent Space” (LC-NC for short). Legacy commands accessing the coherent address space are called “Legacy Commands to Coherent Space” (LC-C for short).

Table 31 summarizes the allowed combination of OCP masters and command types.

The LC-C semantics are different from LC-NC since they operate on a different address space. The LC-C and the Coherent Commands are described in Section 6.2.3.2 on page 99.

Table 31 Allowed Commands

Master Type	Legacy Commands to Noncoherent Space (LC-NC)	Legacy Commands to Coherent Space (LC-C)	Coherent Commands (CC)
Legacy	Yes	No	No
Coherence-Aware	Yes	Yes	No
Coherent	Yes	Yes	Yes

Table 32 summarizes the scope of the address space access for each command type.

Table 32 Address Space Access by Command Type

Command Type	Address Space Access Scope
LC-NC	Non-coherent address space at OCP word granularity. Bridge agent handles multiple word sizes, packing, etc.
LC-C, CC	Coherent address space at cache line granularity, aligned at cache line size boundary. Single cache line size used within entire coherence domain.

5.7 Self Intervention and Serialization

When a coherent slave receives a request *R1* from a coherent master *M1*, the slave is required to send an intervention request to *M1* in addition to any other intervention requests it needs to send as a result of request *R1*. Such a request to *M1* is called a **self intervention**. In case of conflicting requests by multiple masters to the same cache line, the self intervention request is used to establish the same order at the coherent master as the conflicting requests seen at the coherent slave. Self intervention is a key mechanism to ensure cache coherence and freedom from deadlock.

An intervention that is not a self intervention is called a **system intervention**. The term **intervention** by itself is can be used to refer to either self intervention or system intervention—with the distinction made clear by the context; self-interventions are explicitly noted.

The coherent slave or home agent plays a significant role in ensuring that conflicting write requests (i.e., write-write, read-write, or write-read access sequence to the same cache line) are serialized. The **serialization point** is the logic in the home that orders or serializes the conflicting requests. This ordering is done in an implementation-specific manner. It is necessary that the coherent masters that process these conflicting requests also see them in the same order established by the home agent. To facilitate this, OCP requires that the interconnect preserves the ordering of OCP transactions between a given pair of entities (A and B) on a per-address basis: if two transactions *T1* and *T2* with the same address are launched from A to B, the interconnect will deliver the transactions to B in the same order. These conditions, along with the self intervention mechanism, ensure that all coherent masters see writes to the same cache line in the order established at the home agent.

Thus, each coherent master has to implement logic to maintain the ordering imposed by the home. The serialization point at the home is called the *primary serialization point* and the one at each master is called the *secondary serialization point*. Unless otherwise noted, the term *serialization point* refers to the primary serialization point.

In a snoop based design, the home agent for all coherent requests is typically the interconnect agent itself, which acts as the coherent slave. Thus, snoop based designs have a single serialization point. In a directory based design, the home agents for coherent addresses may be physically centralized or may be distributed and are typically separate from the interconnect. Each physical home agent becomes the serialization point for the set of coherent addresses it controls.

5.8 Interconnect or Bridge Agent

An OCP 3.0 cache coherent system requires at least two coherent masters. It also requires at least one coherent slave. The slave in a directory based implementation needs to be able to *identify* the coherent masters to know the coherent cache state, send targeted intervention requests, etc. Hence, unlike a legacy OCP system, the slave needs to be aware of all the coherent masters in the coherence domain. Further, since there are at least three entities in the coherent system, a bridge or interconnect agent is *necessary* in OCP 3.0. The bridge has to preserve the transaction ordering property on a per-address basis as mentioned in Section 5.7.

Note that the interconnection agent is still able to preserve the abstraction that a master still communicates with a single slave and a legacy slave still communicates with a single legacy master. Thus all these entities have a “local” view as outlined in Section 5.2. The interconnect agent, in this case, acts as a proxy for these entities by assigning appropriate IDs and providing routing functionality. Thus, only coherent slaves which have directory based implementations need to have the “system view” in addition to the interconnect.

In snoopy based designs, the interconnect frequently provides additional functionality by acting as the coherent slave. This is explained in Section 5.11.2.

5.9 Port Characteristics

Table 33 captures the roles of different masters and slaves.

Table 33 Roles of Masters and Slaves

Core Type	**Legacy Port**	**Main Port**	**Inter-vention Port**	**Function**
Legacy Master	Yes	No	No	• Initiates requests to non-coherent address space only • Receives responses from non-coherent address space only
Coherence-Aware Master	No	Yes	No	• Initiates legacy and coherent requests to coherent and non-coherent spaces using legacy commands • Receives responses from coherent and non-coherent spaces
Coherent Master	No	Yes	Yes	• Initiates requests to both non-coherent and coherent address spaces on main port (including coherent commands) • Receives responses on main port • Receives intervention requests • Generates intervention responses
Legacy Slave	Yes	No	No	• Receives legacy requests • Initiates legacy responses
Coherent Slave (Home Agent)	Possible[1]	Yes	Yes	• Receives requests on main port • Generates intervention requests • Receives intervention responses • Generates legacy port requests[1] • Receives legacy port responses[1] • Generates responses on main port

1. These are requests to a non-coherent slave (typically the memory). This can be handled through a legacy OCP port or through a custom interface.

Figures 7—9 capture the port connectivities and port directions for three generic cases:

- Figure 7 for an OCP non-coherent (legacy) system
- Figure 8 for an OCP coherent system which is snoop based with the interconnect acting as the coherent slave or home.

- Figure 9 for an OCP coherent system with a centralized directory based coherent slave.

Figure 7 Block Diagram of OCP Non-Coherent System

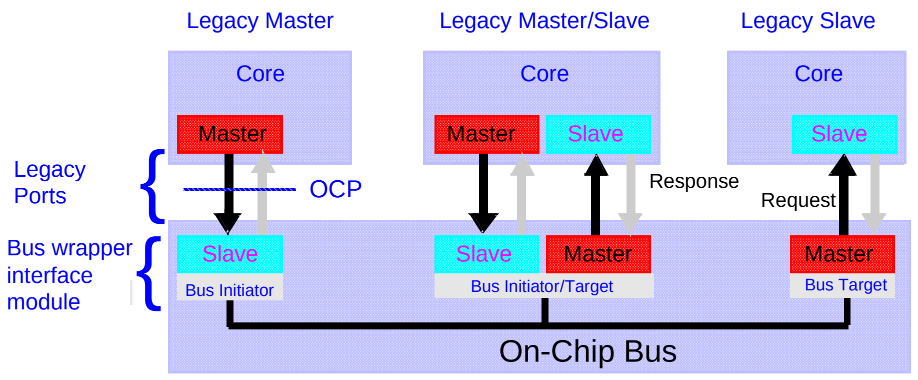


Figure 8 Block Diagram of Snoop-Based OCP Coherent System

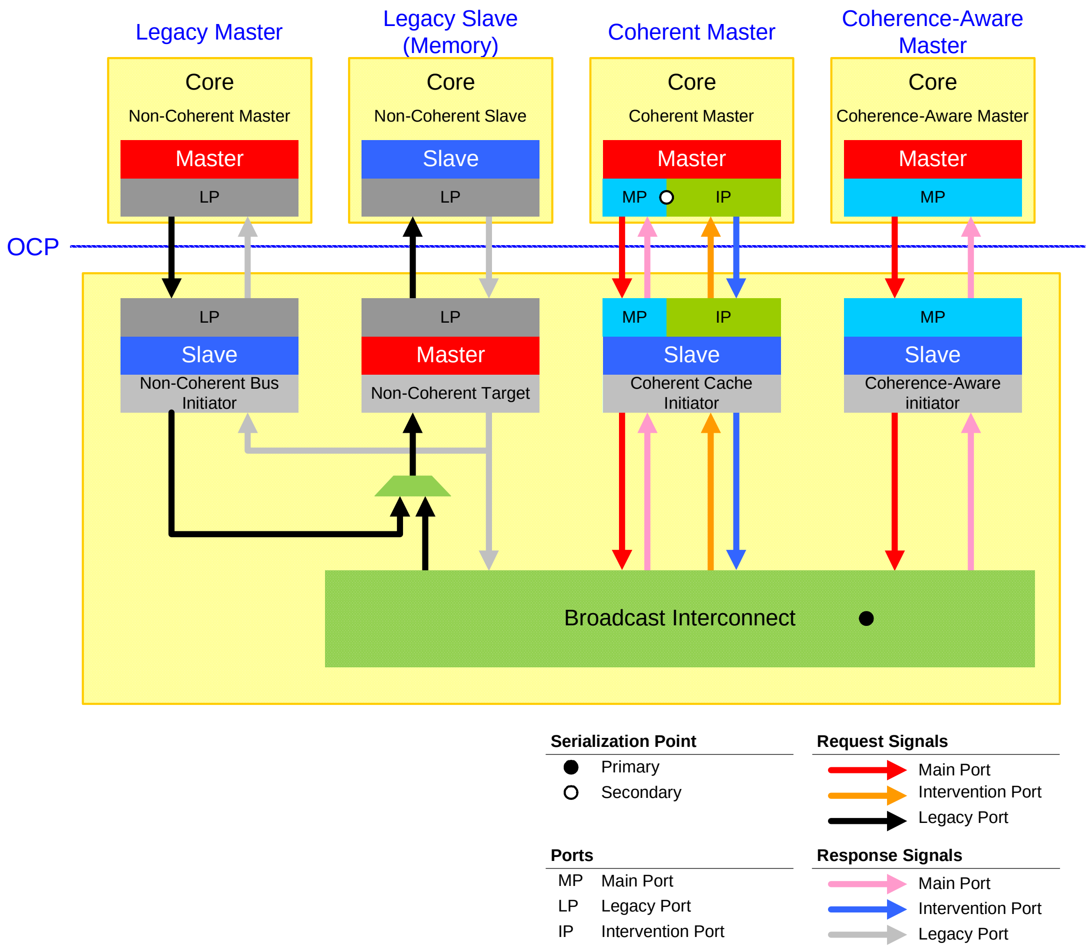
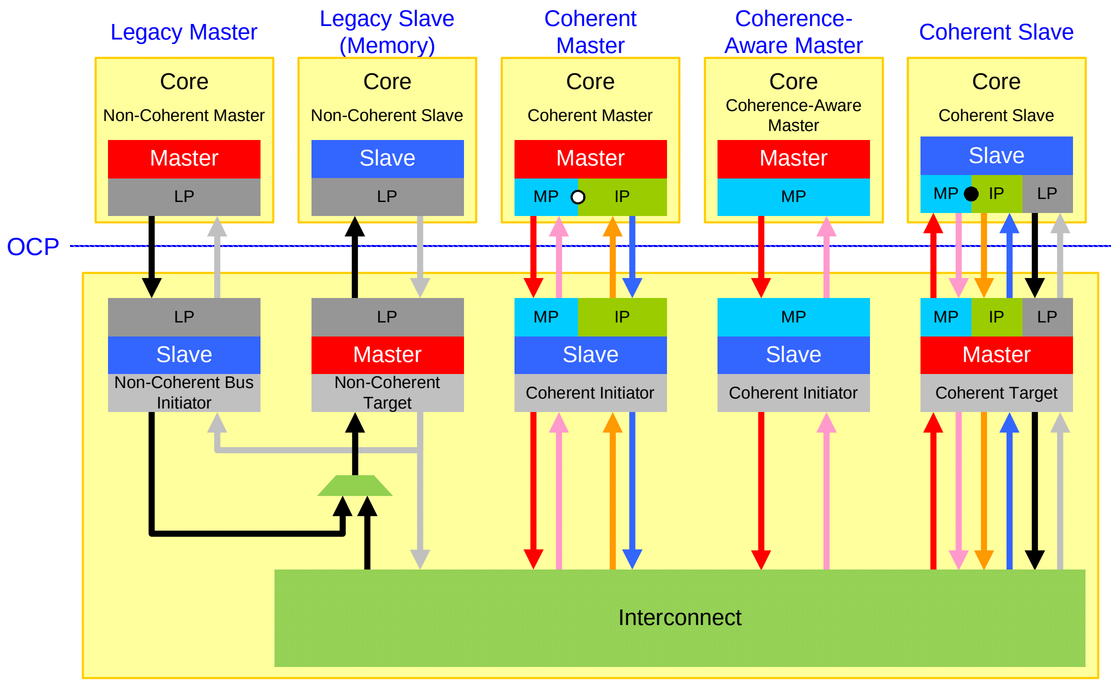
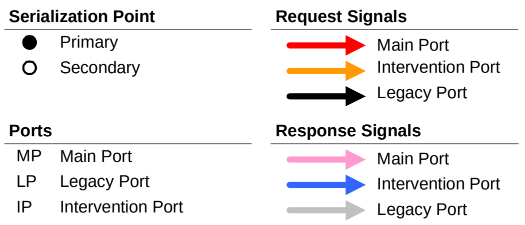


Figure 9 Block Diagram of Directory-Based OCP Coherent System





5.10 Master Models

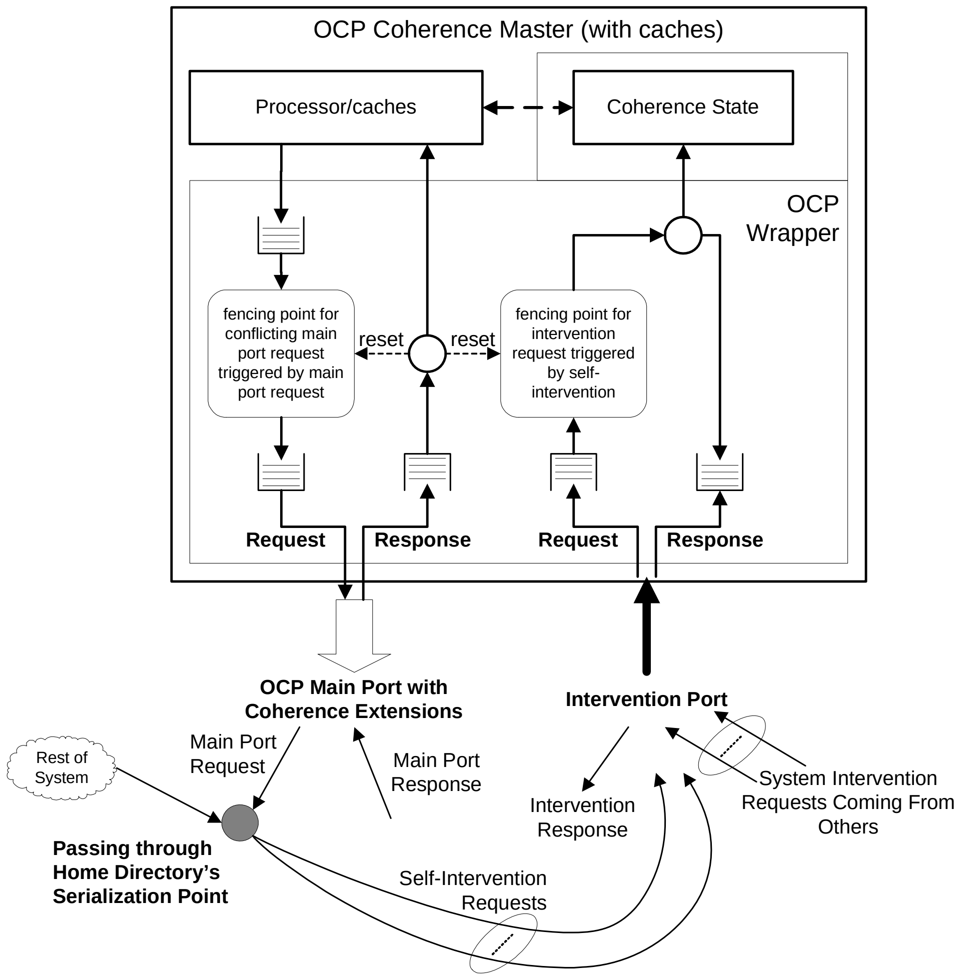
5.10.1 Coherent Master

Each request on the main port generates at most one response. This requirement makes the design of the coherent master relatively simple. The coherent slave and the interconnect have to bear additional responsibilities (as outlined in Section 5.11) to support this requirement.

Consider a coherent master sending a read request to the coherent slave and waiting for its response.

In the interim, it receives the self intervention request on its intervention port and one or more system intervention requests, one or more of which may conflict with the address of the request. Figure 10 shows the abstract model of the coherent master to deal with coherence and serialization.

Figure 10 Abstract Model of Coherent Master


OCP Coherence Master (with caches)
Processor/caches
Coherence State
OCP Wrapper
fencing point for conflicting main port request triggered by main port request
reset
reset
fencing point for intervention request triggered by self-intervention
Request
Response
Request
Response
OCP Main Port with Coherence Extensions
Intervention Port
Main Port Request
Main Port Response
Rest of System
Passing through Home Directory's Serialization Point
Intervention Response
System Intervention Requests Coming From Others
Self-Intervention Requests

An abstract implementation of the secondary serialization point is described below.

Each request is associated with two fencing points: one at the main port request path and the other at the intervention port request path. Each fencing point is associated with a fencing interval.

The main port fencing interval begins when the request enters the lower queue on the request side and lasts till all the associated response is received on the main port response queue. During this interval, the fencing logic does not accept additional conflicting requests on the main port from the master entity (the processor/cache complex).

The intervention port fencing interval begins when the self intervention enters the intervention port request side and is detected by the fencing logic. It lasts until the associated response is received on the main port response queue. During this interval, the fencing logic does not accept conflicting requests from the intervention port's request queue. Note that conflicting intervention requests arriving at the master before the self intervention request are serviced in order at the intervention port and the appropriate responses generated (i.e., cache look up, possible cache state change, generation of response).

Upon receipt of the response for the request, the fences need to be cleared. The fencing logic is implementation specific.

A non-coherent request follows the same behavior as a legacy master.

5.10.2 Coherence-Aware Master

A master sends a request on its main port. It then waits for the associated response to come back on the main port. Since the master has no coherent cache, it does not have an intervention port. Correspondingly, this simplifies the abstract model of Figure 10 (e.g., only the main port fencing logic is needed).

A non-coherent request follows the same behavior as a legacy master.

5.10.3 Legacy Master

A legacy master generates only non-coherent requests. It follows the same behavior as a traditional OCP master. The target addresses are to the non-coherent address space.

5.11 Slave Models

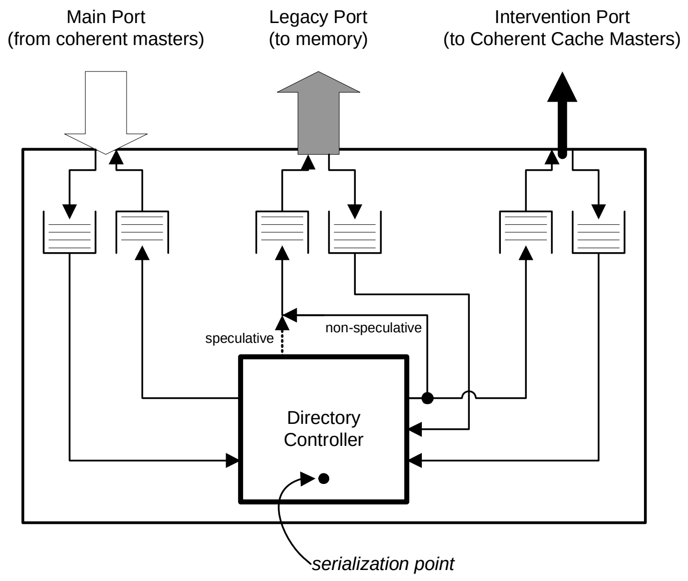
It is convenient to consider snoop-based and directory-based coherent slaves separately.

5.11.1 Coherent Slave: Directory Based

Figure 11 shows the abstract model of the directory based coherent slave to deal with coherence and serialization.

The directory is a logically centralized structure which maintains information of cache lines at each coherent master. Various directory schemes are possible depending on what information it maintains. To make this discussion concrete, it is assumed that the directory maintains the cache state for each cache line that is cached in the coherence domain. If a line is not present in any coherent master, then the most up-to-date data is present in the memory.

Figure 11 Directory-Based Coherent Slave Model



A request received on the main port is looked up in the directory controller to determine which of the coherent masters need to be sent intervention requests in addition to the self intervention request. These requests go out on the intervention port. In addition, a request to memory may also need to be sent. The figure shows a speculative path option to memory in which case the memory request is sent before the directory lookup, optimizing for latency. Alternatively, bandwidth conscious designs could do a lookup and determine if a memory request is warranted (e.g., if the line is in M state in a master, then a memory access is not necessary). The directory controller is the serialization point and determines a single order to process conflicting requests across the coherence domain.

Cache writeback requests arriving on the main port will be written to memory on the legacy port, after a directory lookup to update the state of the line.

Responses arriving at the intervention port and at the legacy port are appropriately "merged" and zero or more responses are generated depending on the nature of the request (for example, on a read request it could be the read data from memory combined with a completion response or it could be no response if a modified line was returned directly by a responding coherent master to the requesting coherent master). The main port of the requesting coherent master receives only one response for each request. With cache-to-cache transfers, the modified line that is received (DVA) also serves as a transaction completion indicator.

As already mentioned, receiving only one response makes the design of the coherent master relatively simple. It has the potential to introduce race conditions at the directory, however. It is expected that the implementation will take care to prevent such races and possible deadlocks. Some scenarios are outlined below:

In the case of a cache-to-cache transfer, the directory may choose to not generate a completion response after the transaction is complete from its perspective. If the slave does generate a response, then the interconnect agent must taken on the responsibility to drop this response.

Additionally, the race condition arising below has to be handled by the directory: Assume that a master (say, *B*) supplies a modified cache line to the original requester *A*. The response arrives at *A* and the data is consumed and the transaction is deallocated. It is possible that *A* generates another request to the same cache line even before the additional response from *B* to the directory controller has arrived at the latter. In such a case, the directory should hold off servicing the request from *A* until it has processed the response. This is typically handled by having separate request and response queues at the directory.

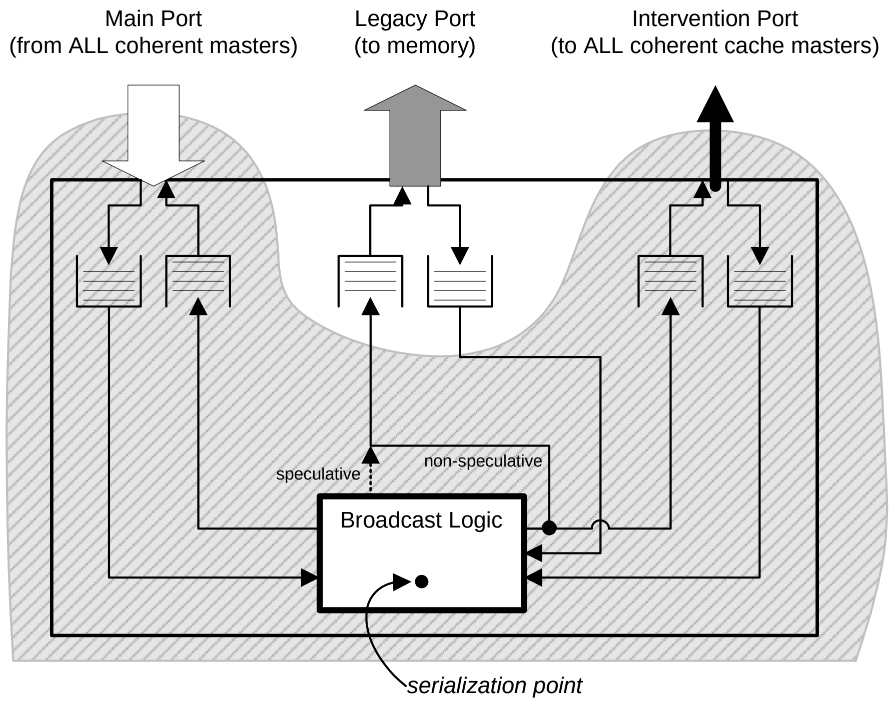
Note that the directory structure is only logically centralized. The serialization point refers to a single cache line address. Hence the directory controller may be physically distributed with each controller being "home" to a distinct set of cache line addresses. The set of addresses controlled by each controller is non-overlapping and together cover the complete coherent address space.

Note that the coherent slave is required to have a "system view"—thus, it needs to identify coherent masters as a requesters, the targets of intervention requests, and as responders who provide cache line data and cache state. This information is used to keep the directory up-to-date, to broadcast intervention requests selectively, etc. OCP provides explicit signals for this purpose on both the main port (MCohId, SCohId) and the intervention port (MCohId, SCohId, MCohFwdId, SCohFwdId).

5.11.2 Coherent Slave: Snoop Based

Figure 12 shows the abstract model of the snoop based coherent slave to deal with coherence and serialization. In a snoop based design, an intervention request is broadcast to ALL the coherent masters in its coherence domain. The directory controller is replaced by a relatively simple piece of logic which is logically and, frequently, physically a single unit which is the target of all coherent requests from all coherent masters. This piece of "bus logic" also is the coherent slave's single serialization point. Since all main port requests are targeted to a single coherent slave and the coherent slave, in turn, broadcasts to all coherent masters, it is not necessary for the coherent slave to have explicit knowledge of the master ids (the *CohId fields as in the directory based design). Apart from these changes, the functionality of the snoop based design is the same as the directory based design.

Figure 12 Home Agent Coherent Slave Model (Snoop Based)



Frequently, the coherent slave functionality (the shaded portion in Figure 12) is *provided by the interconnect agent itself*. In such a case, the main port and the intervention port of the coherent slave is not exposed as OCP ports. Only the legacy port gets exposed to the memory subsystem. In this specification, such an interconnect is called a *broadcast interconnect* or a *snoop-based interconnect*.

Just as in the case of the directory based coherent slave, the snoop based slave and the interconnect need to provide similar functionality to ensure that the coherent master's main port receives at most one response to a request.

5.11.3 Legacy Slave

A legacy slave receives only non-coherent requests and generates responses. It follows the same behavior as a traditional OCP slave. The target addresses are to the non-coherent address space.

5.12 Multi-threading and Tags

If accesses of different threads or tags are not related to cache coherency, the accesses have no ordering requirement. If accesses of different threads or tags access the same cache line (in coherent address space), the order of the accesses must be maintained properly. As described in Section 5.7, the order of all accesses to a coherent memory location is globally ordered at the serialization point.

Multithreading and tagging are supported in the OCP coherence extensions, with the above restrictions.

5.13 Burst Support

Bursts to coherent space have the following restrictions: They have to be SRMD, burst lengths have to be the cache line size. Only INCR, XOR, and WRAP burst sequences are allowed.

Bursts to non-coherent space follow legacy behavior.

5.14 Memory Consistency

The serialization mechanism, enforced through intervention in the OCP coherence extensions, imposes ordering constraints on conflicting cache line addresses which is globally seen or observed. The set of possible orderings may be further restricted by the particular memory consistency model employed by the system. The coherence extensions do not restrict the type of memory consistency used by the system.

5.15 Race Condition, Deadlock, Livelock, and Starvation

In the context of OCP cache coherence, a race occurs when two entities concurrently access the same cache line. The home agent establishes the order and this order is reflected throughout the coherence domain; the race is thus resolved.

Some possible scenarios that result from race conditions include:

1. The master's cache state may be changed before it receives its own request from its intervention request port as self-intervention.
2. If a master receives a read request to the same cache line as the master's cache write-back request, the cache write-back request may be canceled.
3. Transactions with data phase may be cancelled. The cc-WB request coming from main port may be canceled by the requester itself, due to the cache state change caused by another master's coherent request.

There are other situations that can cause races, starvation, deadlocks, and livelocks in cache coherent systems. Solutions to these problems are implementation dependent, and are outside the scope of the main specification. Some specific examples of race conditions which arise in specific implementations are discussed in the Developer's Guidelines.

5.16 Heterogeneous Coherence System

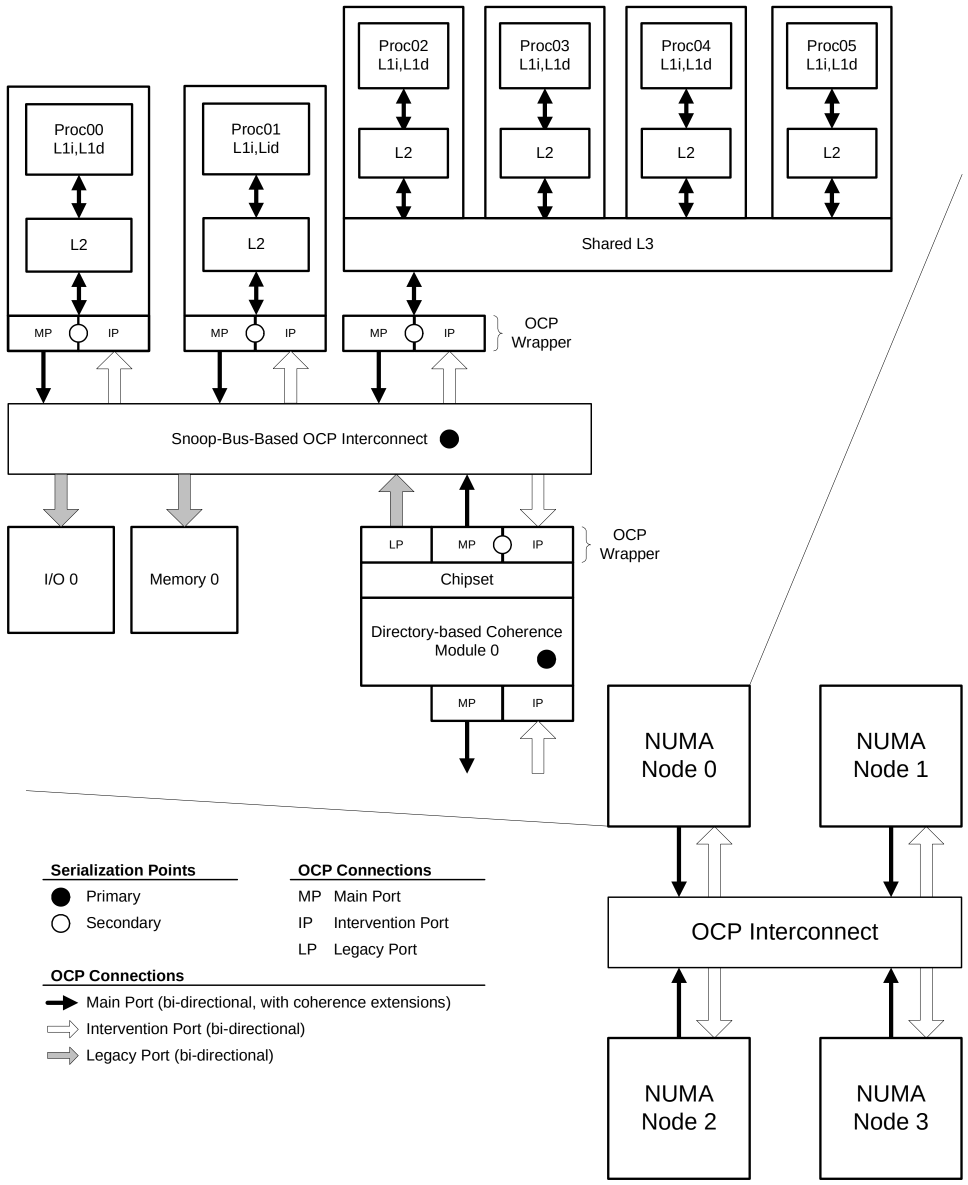
With increasing sophistication and the need for multiple functional blocks (computation intensive, graphics, video, audio, etc.), SoC architectures are being built as hierarchical subsystems. In such architectures, some subsystems could be locally cache coherent (usually referred to "subsystem coherence"). Additionally, there may be a need for cache coherence across subsystems. In fact, the subsystem cache coherence and global coherence may follow different cache coherent protocols.

The OCP coherence extensions support coherence in such sophisticated heterogeneous architectures—this is discussed through the example in Figure 13. The figure shows a hierarchical system composed of 4 "super nodes" or subsystems (NUMA nodes 0–3). Each subsystem is in turn composed of processors with differing cache hierarchies, memory, and I/O. Note that the memory is physically distributed, but is logically shared.

At the local level, a snoop bus based coherence scheme is used to keep the subsystem coherent. The directory agent at each node maintains coherence among the nodes at the global level. Thus, a read request from node 0 first snoops locally. If the read request is not satisfied, then it is routed to its "home" at the global level (say, Node 1). The directory node then handles the request. Assume that the line is modified state in Node 2 (the directory only maintains information at this granularity and not at the level of individual caching agents). This then results in snooping of Node 2 which then returns the data to the original requester, routed through the Node 0 directory controller.

Another level of heterogeneity that is permitted is in the granularity of the cache line states. For example, the directory could only implement an MSI based protocol while the snoop based protocol could be based on MESI.

Figure 13 OCP-Based Heterogeneous SoC Architecture



6 OCP Coherence Extensions: Signals and Encodings

6.1 Definitions

6.1.1 New Transaction Types

Since some of the coherent transactions do not act exactly like legacy write and read commands, new basic command types are introduced.

6.1.1.1 Message Command Type

The first new command type, called **Message**, is similar to a write command but does not do any data transfer on the port in which the command request appears.

There are two extensions of this basic type: **Posted Message** and **Non-Posted Message**.

The Posted Message command does not receive any response if port parameter `writeresp_enable` is set equal to zero.

The following MCmd commands are of type Posted Message:

- CohWriteBack when port parameter `intport_writedata=1`
- CohCopyBack when port parameter `intport_writedata=1`
- CohCopyBackInv when port parameter `intport_writedata=1`

The non-posted message command always receives a response.

The following MCmd command is of type Non-Posted Message:

- CohInvalidate

These MCmd commands are described in Section 6.2.3.2 on page 99.

6.1.1.2 Query Command Type

The second new transaction type, called **Query**, is similar to a read command and always gets a response from the slave but may not transfer any data.

The following MCmd commands are of type Query:

- CohUpgrade
- CohCompletionSync

These MCmd commands are described in Section 6.2.3.2 on page 99.

6.2 Main Port: Parameters, Signals, and Encodings

6.2.1 Introduction

The main port is an OCP port with the following extensions, new signals, and port restrictions:

- A MCohCmd signal is added to the request phase.
- The MCmd signal field is extended to allow issuing coherence commands.
- The SResp signal field is extended to support new coherence response semantics.
- A SCohState signal is added to the response phase to indicate the coherence installing state (at the response receiving side).
- SRMD bursts must be supported, i.e., the MBurstSingleReq signal must be supported or, alternatively, the signal may be tied off to a non-default tie-off value of 1. The datahandshake phase must be enabled. The `reqdata_together` parameter must be set to 1.
- If the port parameter `intport_writedata=0`, then the CohWriteback command behaves in this manner:
  1. The initial write request occurs on the Main port with the write data phase appearing on the Main port.
  2. The home agent sends a self-intervention request to the initiator on the intervention port.
  3. The initiator responds with OK to acknowledge the operation.
- If the port parameter `intport_writedata=1`, then the CohWriteback command behaves in this manner:

1. The initial write request occurs on the Main port but no write data phase appears on the Main port.
2. The home agent sends a self-intervention request to the initiator on the intervention port. No write data phase occurs with this request.
3. The initiator responds with the writeback data on the intervention port, (if the cache line hasn't been invalidated in between steps 1 and 2).

This option allows self-intervention data responses and "normal" snoop responses to use the same data paths and thus be ordered.

6.2.2 Main Port Parameters

`cohcmd_enable`

If this parameter is set, the MCohCmd signal is instantiated. The MCmd signal is extended.

`cohstate_enable`

If this parameter is set, the SCohState signal is instantiated.

`coh_enable`

If this parameter is set, the Cached Coherent commands are enabled for the port.

`cohnc_enable`

If this parameter is set, the Non-Cached Coherent command set are enable for the port. Each specific commands within the set have their own enable parameters.

`cohwrinv_enable`

If this parameter is set and both `coh_enable` and `writeresp_enable` are also set, then the CohWriteInvalidate and CohInvalidate commands are allowed on the main port.

`read_enable`, `readex_enable`, `rdlwrc_enable`, `write_enable`, `writenonpost_enable`

These parameters enable the specific commands within the Non-Cached, Coherent set.

`upg_enable`

If this parameter is set, the Coh_Upgrade command is enabled for the port.

`intport_exists`

If this parameter is set, an associated intervention port is instantiated.

`intport_writedata`

If this parameter is set and `intport_exists` is also set, then writeback data appears on the intervention port instead of the main port.

`mcohid_enable`

If this parameter is set, the MCohID signal is instantiated.

`scohid_enable`

If this parameter is set, the SCohID signal is instantiated.

`cohfwdid_enable`

If this parameter is set, the MCohFwdID and SCohFwdID signals are instantiated.

`mcohid_wdth`

Width of the MCohID signal.

`scohid_wdth`

Width of the SCohID signal.

`cohfwdid_wdth`

Width of the CohFwdID signal.

6.2.3 Signals and Encodings

Table 34 lists the OCP 2.2 signals that must be included in the main port. It also lists in bold and italic fonts the new signals and their control parameters introduced for coherent transactions. Unless specifically mentioned, the default tie-off values are the same as in the legacy specification.

Table 34 Main Port Signals

Group	**Signal**	**Enable Control Parameters**	**Width Control Parameters**	**Comments**
Basic	Clk			Required
	MAddr	addr=1	addr_wdth	Required
	MCmd ***Bus widened for coherent commands.***	To enable the coherent commands: ***cohnc_enable*** ***coh_enable*** ***cohwrinv_enable***		Required
	MData	mdata=1	data_wdth	Required for SRMD
	MDataValid	datahandshake=1		Required for SRMD
	MRespAccept	resp respaccept		Optional
	SCmdAccept	cmdaccept		Optional

	SData	resp = 1
		sdata = 1
			data_wdth	Required

	SDataAccept	datahandshake=1 dataaccept		Optional
	SResp	resp=1		Required

Group	**Signal**	**Enable Control Parameters**	**Width Control Parameters**	**Comments**
Simple	MAddrSpace	addrspace	addrspace_wdth	Optional

	MByteEn	mdata=1
		byteen=1
				Required

	MDataByteEn	mdata=1
		datahandshake=1
		mdatabyteen=1
				Required

	MDataInfo			Optional
	MReqInfo	reqinfo reqinfo_wdth		Optional
	SDataInfo	resp=1 sdatainfo sdatainfo_wdth		Optional
	SRespInfo	resp=1 respinfo respinfo_wdth		Optional
Burst	MAtomicLength	atomiclength	atomiclength_wdth	Tied off to cache line size
	MBurstLength	burstlength	burstlength_wdth	Tied off to cache line size
	MBurstPrecise	burstprecise=1		Required. Set to 1 for Coherent commands
	MBurstSeq	burstseq		Required. Only INCR, XOR, and WRAP are allowed.

	MBurstSingleReq	datahandshake=1
		burstsinglereq=1
				Required.
				Set to 1 for Coherent commands.

	MDataLast	datalast=1		Required
	MReqLast	reqlast		Optional
	SRespLast	resplast		Optional

Group	**Signal**	**Enable Control Parameters**	**Width Control Parameters**	**Comments**
Coherence	*SCohState*	*cohstate_enable*		Required Default Tie-off=0
	MCohCmd	*cohcmd_enable*		Required Default Tie-off=0
	MCohID	*mcohid_enable*	*mcohid_wdth*	Optional; required for directory based protocols and three-hop protocols
	SCohID	*scohid_enable*	*scohid_wdth*	Optional; required for directory based protocols and three-hop protocols
	MCohFwdID	*cohfwdid_enable*	*cohfwdid_wdth*	Optional; required for three-hop protocols
	SCohFwdID	*cohfwdid_enable*	*cohfwdid_wdth*	Optional; required for three-hop protocols
Thread	MConnID	connid=0		Optional
	MDataThreadID	when threads > 1 datahandshake=0	threads	Optional
	MThreadBusy	mthreadbusy threads	threads	Optional
	MThreadID	when threads > 1	threads	Optional
	SDataThreadBusy	sdatathreadbusy threads datahandshake=1	threads	Optional
	SThreadBusy	sthreadbusy threads	threads	Optional
	SThreadID	when threads > 1 resp	threads	Optional
Tags	MTagID	tags	tags	Optional
	MDataTagID	tags datahandshake	tags	Optional
	STagID	tags resp	tags	Optional
	MTagInOrder	taginorder		Optional
	STagInOrder	taginorder resp		Optional

Group	Signal	Enable Control Parameters	Width Control Parameters	Comments
Sideband	SReset_n or MReset_n	sreset=1 or mreset=1		Required
	(all others)	(all others)		Optional
Test	(all)	(all)		Optional

6.2.3.1 MCohCmd

When set to one, indicates that the command is coherent. When set to zero, the semantics of the command depend on whether the target address is in coherent address space or non-coherent address space.

6.2.3.2 MCmd

When the OCP interface supports coherency, the width of the MCmd signal is extended to five-bits to accommodate the extra coherence commands.

Commands are arranged into two groups: Non-Coherent and Coherent. Non-Coherent commands are the same set of commands as in the existing OCP 2.2 command set and are also referred to as Legacy commands. Within the Coherent set of transactions, some existing OCP 2.2 commands remain, but are re-defined as Coherence-Aware[2]. The Coherence-Aware commands are used by initiators that do not contain caches but access the coherent address space. The new coherent commands must always be issued with MCohCmd asserted. See Table 35 below for the extensions to the encoding of MCmd.

Table 35 Extended MCohCmd and MCmd Encoding

MCohCmd	MCmd	Command	Mnemonic	Data Source	Coherence State Changed?	Address Space
0	0x0	Idle	IDLE	None	No	(none)
0	0x1	Write	WR	Requester	No	Non-Coherent
0	0x2	Read	RD	Home	No	Non-Coherent
0	0x3	ReadEx	RDEX	Home	No	Non-Coherent
0	0x4	ReadLinked	RDL	Home	No	Non-Coherent
0	0x5	WriteNonPost	WRNP	Requester	No	Non-Coherent

[2] They are redefined as Non-Cached because the bulk of the use of these commands will be to satisfy Non-Cached accesses; however, they could be used by caching agents as well; examples being write-through caches and cached DMA controllers.

MCohCmd	MCmd	Command	Mnemonic	Data Source	Coherence State Changed?	Address Space
0	0x6	WriteConditional	WRC	Requester	No	Non-Coherent
0	0x7	Broadcast	BCST	Requester	No	Non-Coherent
0	0x8-0xF	(Reserved)	(Reserved)	—	—	—
0	0x1	Write	WR	Requester	Yes	Coherent
0	0x2	Read	RD	Home or Owner	Yes	Coherent
0	0x3	ReadEx	RDEX	Home or Owner	Yes	Coherent
0	0x4	ReadLinked	RDL	Home or Owner	Yes	Coherent
0	0x5	WriteNonPost	WRNP	Requestor	Yes	Coherent
0	0x6	WriteConditional	WRC	Requester	Yes	Coherent
0	0x7	Broadcast	BCST	Not Permitted		Coherent
1	0x8	CohReadOwn	CC_RDOW	Home or Owner	Yes	Coherent
1	0x9	CohReadShare	CC_RDSH	Home or Owner	Yes	Coherent
1	0xA	CohReadDiscard	CC_RDDS	Home or Owner	No	Coherent
1	0xB	CohReadShareAlways	CC_RDSA	Home or Owner	Yes	Coherent
1	0xC	CohUpgrade	CC_UPG	None or Owner	Yes	Coherent
1	0xD	CohWriteBack	CC_WB	Requester	Yes	Coherent
1	0x0E–0x0F	(Reserved)	(Reserved)	—	—	—
1	0x10	CohCopyBack	CC_CB	Requester	Yes	Coherent
1	0x11	CohCopyBackInv	CC_CBI	Requester	Yes	Coherent
1	0x12	CohInvalidate	CC_I	None	Yes	Coherent
1	0x13	CohWriteInvalidate	CC_WRI	Requester	Yes	Coherent
1	0x14	CohCompletionSync	CC_SYNC	None	No	Coherent
1	0x15–0x1F	(Reserved)	(Reserved)	—	—	—

The semantics of legacy commands targeting coherent address space are described below. Please see Section 5.6 on page 79 for a list of restrictions related to cache line granularity and Section 5.13 on page 90 for bursts.

Write (0x1, WR)

This form of coherent request is meant to transfer cache line-sized data to memory (finer granularity can be achieved through the use of byte enables). While the semantics of this command are very similar to the legacy Write (WR) command, the home invalidates cache lines for write invalidate semantics. This command is generated by non-cached or write-through stores etc. This command is enabled by the port parameter `write_enable`.

Read (0x2, RD)

Very similar to a Legacy Read command, but the system returns data from the owning agent rather than home when the former has the most recent copy. This command is generated by non-cached loads or instruction fetches; read misses for write-through memory locations, etc. This command is enabled by the port parameter `read_enable`.

ReadEx (0x3, RDEX)

Very similar to a Legacy ReadEx command, but the system returns data from the owning agent rather than home when the former has the most recent copy. This command is generated by non-cached loads. The command is enabled by the port parameter `readex_enable`.

ReadLinked (0x4, RDL)

Similar to its non-coherent counterpart (RDL), this command can be used to set a reservation at home, but in a coherent system. This command is generated by non-cached synchronizing[3] loads etc. This command is enabled by the port parameter `rdlwrc_enable`.

WriteNonPost (0x5, WRNP)

This form of coherent request is meant to transfer cache line-sized data to memory (finer granularity can be achieved through the use of byte enables). While the semantics of this command are very similar to the legacy WriteNonPost (WRNP) command, the system invalidates cache lines for write invalidate semantics. This command is generated by non-cached or write-through stores. This command is enabled by the port parameter `writenonpost_enable`.

WriteConditional (0x6, WRC)

Similar to its non-coherent counterpart (WRC), this command can be used to clear a reservation at home, but in a coherent system. This command is generated by non-cached synchronizing stores etc. This command is enabled by the port parameter `rdlwrc_enable`.

Broadcast (0x7, BCST)

This command is undefined when the target is in coherent space.

CohReadOwn (0x8, CC_RDOW)

This coherent command is used to read data from home with the intent to modify. This command is generated by processor stores that miss in the cache hierarchy. The data transfer size is a cache line.

[3] The term 'synchronizing loads' refers to conditional load instructions, which are available in the instruction sets of various architectures.

On all CPUs with coherent caches (excluding the original requester), if there is a cache line with a matching address that is in the Modified or Owned state, the implementation has the choice of:

- writing back the cache line to home, or,
- forwarding the data to the requestor directly from the cache, or,
- doing both.

These options do not affect the behavior of the intervention ports and main ports so there are no port parameters for these options.

On all CPUs with coherent caches (not including the original requester), if there is a cache line with the matching address and it is in a state other than Invalid, the cache line state transitions to Invalid.

The original requester receives the most up-to-date data.

CohReadShared (0x9, CC_RDSH)

This coherent command is used to read data from home with no intent to modify. This command is generated by processor loads that miss in the cache hierarchy. The data transfer size is a cache line.

For the MOESI protocol:

On all CPUs with coherent caches (excluding the original requestor), if there is a cache line with the matching address and it is in the Modified state, the cache line state transitions to Owned.

On all CPUs with coherent caches (excluding the original requestor), if there is a cache line with the matching address and it is in the Modified or Owned states, the data is forwarded to the requestor directly from the cache.

For the MOESI and MESI protocols:

On all CPUs with coherent caches (excluding the original requester), if there is a cache line with the matching address and it is in the Exclusive state, the cache line state transitions to Shared.

The implementation may also choose to forward the data to the requestor directly from the cache, this option is enabled by the `intport_estate_c2c` port parameter.

For the MSI and MESI protocol:

On all CPUs with coherent caches (excluding the original requestor), if there is a cache line with the matching address and it is in the Modified state, the cache line state transitions to Shared. The cache line is written back to home.

The implementation may also choose to forward the data to the requestor directly from the cache. This option does not affect the behavior of the intervention ports and main ports so there is no port parameter for this option.

If the cache line with the matching address is in the Shared state, the cache line state stays as previous[4].

For all protocols, the original requester receives the most up-to-date data.

CohReadDiscard (0xA, CC_RDDS)

This coherent command is used to read data from the processor caches and not cause any cache line state changes. It is normally generated by external agents (such as coherent DMA controllers) to read data from the processor cache hierarchy. The data transfer size is a cache line.

The cache line state is not modified. The original requester receives the data.

CohReadShareAlways (0xB, CC_RDSA)

This coherent command is used to read data from home with intent to never modify. The install state is always shared. This command is generated by processor instruction fetches for coherent instruction caches. The cache line state transitions are the same as for CohReadShared. The data transfer size is a cache line.

Coherent instruction caches are not snooped as there can never be any modified data and the install state is always shared. The original requester receives the requested data.

CohUpgrade (0xC, CC_UPG)

This coherent command is used to request ownership of a shared cache line from the system. It is usually generated for processor stores which hit cache lines with shared states. This command is of the new type “Query.” The possible responses are OK (no data) or DVA (data). The data transfer size is either zero or a cache line.

On all CPUs with coherent caches (excluding the original requester), if there is a cache line with the matching address and it is in the Modified or Owned state, the implementation has the choice of writing back the cache line to home or forwarding the data to the requestor or doing both[5]. For this case, the response is DVA. This DVA response only occurs if another agent has modified its copy of the data after receiving the CohUpgrade request (A race within the other agent between its local operations and the initiator’s command).

The more common response is OK (the original requestor has an up-to-date copy of the data), and there is no data phase.

On all CPUs with coherent caches (not including the original requester), if there is a cache line with the matching address and it is in a state other than Invalid, the cache line state transitions to Invalid.

The original requester receives the updated data.

[4] Some implementations may choose to forward the data to the requestor directly from the cache, this option requires an additional cache line state (Forwarding/Recent) that is beyond the scope of this document.

[5] These options do not affect the behavior of the intervention and main ports, so there is no port parameter for these options.

This command is enabled by the port parameter `upg_enable`. If this command is not enabled, any store which hits a shared cache line will generate a CohReadOwn command.

CohWriteBack (0xD, CC_WB)

This coherent command is used to writeback cache lines into home memory. It has posted write semantics. When `intport_writedata` is set to 0, the write data phase happens on the main port along with the request phase. The data transfer size is a cache line.

The user has the option of the write data phase to occur on the intervention port after a self-intervention (port parameter `intport_writedata=1`). For this case, this command is of the new type "Message." This option allows self-intervention data responses and "normal" snoop responses to use the same datapaths and thus be ordered.

CohCopyBack (0x10, CC_CB)

This coherent command is used to writeback cache lines into home and the cache line is not evicted from the cache hierarchy. This command is generated by processor-specific cache management instructions. It has posted write semantics. The data transfer size is either zero or a cache line.

On masters with coherent caches, if the cache line with the matching address is originally in the Modified or Owned state then the cache line will be written back to home. The cache line state transitions to Shared. When `intport_writedata` is set to 0, the write data phase occurs on the main port along with the request phase. When `intport_writedata` is set to 1, the write data phase happens on the intervention port as part of the snoop response.

On masters with coherent caches, if the cache line with the matching address is in the Shared or Exclusive state, the cache line state is unchanged and there is no data phase.

CohCopyBackInv (0x11, CC_CBI)

This coherent command is used to writeback cache lines into home and the cache line is evicted from the cache hierarchy. This command is generated by processor-specific cache management instructions. It has posted write semantics. The data transfer size is either zero or a cache line.

On masters with coherent caches, if the cache line with the matching address is originally in the Modified or Owned state then the cache line will be written back to home. The cache line state transitions to Invalid. When `intport_writedata` is set to 0, the write data phase occurs on the main port along with the request phase. When `intport_writedata` is set to 1, the write data phase happens on the intervention port as part of the snoop response.

On all CPUs with coherent caches, if the cache line with the matching address is in the Shared or Exclusive states then the cache line state transitions to Invalid. For this case, there is no data phase.

CohInvalidate (0x12, CC_I)

This coherent command is used to purge data from the cache hierarchy. This command is generated by processor-specific cache management instructions and also generated by coherent DMA controllers. This command has non-posted write semantics. The data transfer size is zero.

On masters with coherent caches, if a cache line contains the requested address, its state is set to invalid, regardless of the previous state.

There is no data phase for this command.

The port parameter `cohwrinv_enable` must be set as well for the CohWriteInvalidate command to be used on the main port.

CohWriteInvalidate (0x13, CC_WRI)

This coherent command is used to inject new data into a coherent system by simultaneously invalidating a cache line from the system and updating its value at home. The use of byte enables allows the update of partial cache lines. Typically used by coherent DMA controllers to write new values into home and remove stale copies from the cache hierarchy. This command has non-posted write semantics. The data transfer size is less than or equal to a cache line.

On all CPUs with coherent caches, if the cache line with the matching address is originally in the Modified or Owned state and the write does not modify the entire cache line, then the cache line data will be supplied so that the new write data can be merged. The cache line state transitions to Invalid. For this case, SResp is equal to DVA. The data transfer happens on the intervention port as the snoop response. For this case, the home agent is responsible to merge the newer write data with the older snoop response data before the data is written to system memory.

On all CPUs with coherent caches, if the cache line with the matching address is in the Shared or Exclusive states or if the new write modifies all bytes within the cache line, then the cache line state transitions to invalid. For this case, SResp is equal to OK and there is no data phase.

The port parameter `cohwrinv_enable` must be set as well for the CohWriteInvalidate command to be used on the main port.

CohCompletionSync (0x14, CC_SYNC)

This coherent cache command is used to maintain ordering. This command is of the new type “Query.” The slave, after receiving this command, in an implementation specific fashion, will send the response when it is satisfied that transaction ordering has been satisfied. Normally this is used to stall the initiator until all preceding requests have reached a global ordering point within the system. The slave responds with a single cycle of DVA on the SResp bus.

For this command there is no data phase.

6.2.3.3 SCohState

This signal indicates the install state and is part of the response phase and is passed back to the master with any response to a coherent request. It is also used to indicate the prior state of the cache line on interventions. For non-coherent and coherence-aware requests, this signal is a “don’t care”. SCohState is a three-bit field with encodings as shown in Table 36.

Table 36 SCohState Encoding

SCohState	**Name**	**Mnemonic**
0x0	Invalid	I
0x1	Shared	S
0x2	Modified	M
0x3	Exclusive	E
0x4–0x5	Reserved	—
0x6	Owned	O
0x7	Reserved	—

6.2.3.4 SResp

Existing responses remain as in OCP 2.2, but a new one (OK) is added to support intervention port related transactions and main port transaction (e.g., CC_UPG). The OK response indicates completion without any data transfer. If the OCP interface supports coherence extensions, SResp becomes a three-bit field with encodings as shown in Table 37, below.

Table 37 SResp Encoding

SResp Value	**Response**	**Mnemonic**
0x0	No response	NULL
0x1	Data valid / accept	DVA
0x2	Request failed	FAIL
0x3	Response error	ERR
0x4	Ack without data transfer	OK
0x5–0x7	Reserved	-

6.2.3.5 MReqInfo

MReqInfo is not explicitly defined, but mentioned to remind implementors that it is available for sending more coherency hints if desired. Some examples are Instruction or Data miss, Cache management instructions etc.

6.2.4 Transfer Phases

Table 38 Main Port transfer phases

MCmd	Phases			
	writeresp_enable=0		writeresp_enable=1	
	intport_writedata=0	intport_writedata=1	intport_writedata=0	intport_writedata=1
WR	Request (with write data)	Request (with write data)	Request (with write data); Response	Request (with write data); Response
RD	Request; Response	Request; Response	Request; Response	Request; Response
RDEX	Request; Response	Request; Response	Request; Response	Request; Response
RDL	Request; Response	Request; Response	Request; Response	Request; Response
WRNP	Request (with write data); Response	Request (with write data); Response	Request (with write data); Response	Request (with write data); Response
WRC	Request (with write data)	Request (with write data)	Request (with write data); Response	Request (with write data); Response
CC_RDOW	Request; Response	Request; Response	Request; Response	Request; Response
CC_RDSH	Request; Response	Request; Response	Request; Response	Request; Response
CC_RDDS	Request; Response	Request; Response	Request; Response	Request; Response
CC_RDSA	Request; Response	Request; Response	Request; Response	Request; Response
CC_UPG	Request; Response[1]	Request; Response[1]	Request; Response[1]	Request; Response[1]
CC_UPG	Request; Response[2]	Request; Response[2]	Request; Response[2]	Request; Response[2]
CC_WB	Request (with write data)	Request (no write data) If data is resident within local cache, the CopyBack data is supplied with intervention response on the Intervention Port.	Request (with write data); Response	Request (no write data); Response WriteBack data is supplied with self-intervention response on Intervention Port (if cache line ownership hasn't moved to another master—data race)

MCmd	**Phases**			
	writeresp_enable=0		**writeresp_enable=1**	
	intport_writedata=0	**intport_writedata=1**	**intport_writedata=0**	**intport_writedata=1**
CC_CB	Request (with write data)	Request (no write data) If data is resident within local cache, the CopyBack data is supplied with intervention response on the Intervention Port.	Request (with write data); Response	Request (no write data); Response If modified data is resident within local cache, the CopyBack Data is supplied with intervention response on the Intervention Port.
CC_CBI	Request (with write data)	Request; Response Non-Posted Write	Request (with write data); Response	Request (no write data); Response If modified data is resident within local cache, CopyBack Data is supplied with intervention response on the Intervention Port.
CC_I	Request (with write data); Response	Request; Response	Request (with write data); Response	Request; Response
CC_WRI	Request (with write data); Response	Request; Response	Request (with write data); Response If data is resident within local cache, the snoop data is supplied with the intervention response on Intervention Port.	Request (with write data); Response If modified data is resident within local cache, the snoop data is supplied with the intervention response on the Intervention Port.
CC_SYNC	Request; Response		Request; Response	Request; Response

1. Cache line ownership stays with original requesting master.
2. Data transfer only occurs if cache line ownership had moved to another master (data-race)

6.2.5 Transfer Effects

Read, CohReadOwn, CohReadShared, CohReadDiscard, CohReadSharedAlways

The master receives the requested data on SData.

ReadEx

The master receives the requested data on SData. Sets a lock on the address for the initiating thread.

ReadLinked

The master receives the requested data on SData. Sets a reservation on that address.

Write, WriteNonPost

The request phase includes the write data.

WriteConditional

If there was an existing reservation for the address by the same initiating thread, the request phase includes the write data. If the write proceeds in this manner, the reservation for the address is cleared.

CohUpgrade

If the cache line ownership is still resident within the requesting master, there is no data transfer.

If the cache line ownership had moved to another master (data race), then the master receives the requested data on SData.

CohWriteBack

If port parameter `intport_writedata`=1 there is no data transfer on the main port. The data is transferred on the intervention port.

If port parameter `intport_writedata`=0, the request phase includes the write data.

CohCopyBack, CohCopyBackInv

There is no data transfer on the main port. If the data was resident within any cache, the data is transferred on the intervention port.

CohInvalidate

The SResp value is OK and there is no data transfer phase.

CohWriteInvalidate

The write data is sent along with the Request. The SResp value is OK.

If the data was resident within any cache, the snoop data is written back on the intervention port. For this case, the home agent is responsible to merge this older snoop response data with the newer write data.

CohCompletionSync

The master receives the response from the slave that previous transactions have been made globally visible.

6.3 Intervention Port: Parameters, Signals, and Encodings

6.3.1 Introduction

The intervention port signals and encodings are similar to the main port's signals and encodings for the main port Coherent command set (CC_*). However, many of the port parameters and configurations are fixed.

- The intervention slave only sends out data, it does not receive data.
- ALL intervention port requests must have a response, e.g., the port parameter `writeresp_enable` must be set to 1.
- If port parameter `intport_writedata=0`, then the CohWriteback, CohCopyBack, and CohCopyBackInv commands behave in this manner:
    1. The initial write request occurs on the Main port with the write data phase appearing on the Main port.
    2. The home agent sends a self-intervention request to the initiator on the intervention port. No write data phase occurs with this request.
    3. The initiator responds with OK to acknowledge the operation.
- If port parameter `intport_writedata=1`, then the CohWriteback, CohCopyBack, and CohCopyBackInv commands behave in this manner:
    1. The initial write request occurs on the Main port but no write data phase appears on the Main port.
    2. The home agent coherent slave sends a self-intervention request to the initiator on the intervention port. No write data phase occurs with this request.
    3. The initiator responds with the writeback data on the intervention port, (if the cache line hasn't been invalidated in between steps 1 and 2).

  This option allows self-intervention data responses and "normal" snoop responses to use the same datapaths and thus be ordered.
- There is an option for split transactions on the Intervention Port. This option allows for responses to precede the data transfer. For this option, new data handshaking signals are added to aid in transferring data from the slave back to the master. These signals are MDataAccept and SDataValid. If threads are used with these split transactions, an additional hand-shaking signal, MDataThreadBusy, is used.

Legacy reads to coherent address space are processed as follows:

- ReadEx: The coherent slave issues I_CBI, the intervention port request to write back a possibly modified cache line to the home memory location and evict the line from the cache hierarchy of each coherent master. The memory is also read (in an implementation specific manner, either speculatively or after the response(s) to I_CBI are received). The slave then sets a lock for the initiating thread on this address at the home memory. The data is returned to the requesting master (either the contents of the modified cache line or the memory contents). It is assumed that an implementation specific mechanism ensures that this is the only ReadEx operating on this location.

- Other Read Operations: The coherent slave issues I_RDSA, the intervention port request to read a possibly modified cache line and update the home. The memory is also read (in an implementation specific manner, either speculatively or after the response(s) to I_RDSA are received). With Read Linked, the slave then sets a reservation in a monitor for the initiating thread on this address. The data is returned to the requesting master.

Legacy writes to coherent address space are processed as follows:

- Clearing Write[6]: (Note the home agent coherent slave will be able to determine if this is a clearing write). The data is written to main memory (request on legacy port of coherent slave) and the lock is cleared atomically in an implementation dependent manner.
- Write Conditional: If a reservation is set for the matching address and for the corresponding thread, the slave issues I_WRI, the request to update the value at home. If the reservation is cleared, the write is not performed, a FAIL response is returned and no reservations are cleared.
- Other Writes: Clears the reservations on any conflicting addresses set by other threads. The slave issues I_WRI, the intervention port request to update the value at home.

6.3.2 Port Parameters

`intport_writedata`

If this parameter is set, then writeback data appears on the intervention port instead of the main port.

`intport_split_tranx`

If this parameter is set, then the intervention port data phase occurs after the intervention port response phase instead of being coincident with the response phase. The signals MDataAccept and SDataValid are instantiated.

`intport_estate_c2c`

If this parameter is set, then coherent slaves supply intervention data when their matching local cache lines are in the Exclusive state.

`mcohid_enable`

If this parameter is set, the MCohID signal is instantiated.

[6] The term *clearing write* refers to the Write or WriteNonPost command to the matching address issued after a ReadEx on that thread. It is called a clearing write as it clears any reservations on the matching address set by other threads.

`scohid_enable`

If this parameter is set, the SCohID signal is instantiated.

`cohfwdid_enable`

If this parameter is set, the MCohFwdID signal is instantiated.

`mcohid_wdth`

Width of the MCohID signal.

`scohid_wdth`

Width of the SCohID signal.

`cohfwdid_wdth`

Width of the CohFwdID signal.

6.3.3 Signals and Encodings

Table 39 gives an overview of which signals can be or must be included. New signals and their control parameters introduced for the Coherent Transactions are in bold and italicized font.

Table 39 Intervention Port Signals

Group	Signal	Enable Control Parameters	Width Control Parameters	Comments
Basic	Clk			Required
	MAddr	addr=1	addr_wdth	Required
	MCmd			Required (only a subset of the coherent commands are allowed)
	MData	mdata=0		Not allowed
	MDataValid	datahandshake=0		Not allowed
	MRespAccept	respaccept		Optional
	SCmdAccept	cmdaccept		Optional
	SData	sdata=1	data_wdth	Optional
	SDataAccept	dataaccept=0		Not allowed
	SResp	resp=1		Required (only NULL, DVA, and OK responses allowed)

Group	Signal	Enable Control Parameters	Width Control Parameters	Comments
Simple	MAddrSpace	addrspace	addrspace_wdth	Optional
	MByteEn	byteen=0		Not allowed
	MDataByteEn	mdatabyteen=0		Not allowed
	MDataInfo	mdatainfo=0		Not allowed
	MReqInfo	reqinfo	reqinfo_wdth	Optional
	SDataInfo	sdatainfo	sdatainfo_wdth	Optional
	SRespInfo	respinfo	respinfo_wdth	Optional
Burst	MAtomicLength	atomiclength	atomiclength_wdth	Tied off to cache line size
	MBurstLength	burstlength	burstlength_wdth	Tied off to cache line size
	MBurstPrecise	burstprecise=1		Tied off to 1
	MBurstSeq	burstseq		Required. Only INCR, XOR, and WRAP are allowed.
	MBurstSingleReq	burstsinglereq=1		Tied off to 1
	MDataLast	datalast=0		Not allowed
	MReqLast	reqlast=0		Not allowed
	SRespLast	resplast=0		Not allowed

Group	**Signal**	**Enable Control Parameters**	**Width Control Parameters**	**Comments**
Coherence	*SCohState*			Required, used to transmit current state of the cache line
	MReqSelf			Required
	MCohID	*mcohid_enable*	*mcohid_wdth*	Optional; required for directory based protocols and three-hop protocols[1]
	SCohID	*scohid_enable*	*scohid_wdth*	Optional; required for directory based protocols and three-hop protocols[1]
	MCohFwdID	*cohfwdid_enable*	*cohfwdid_wdth*	Optional; required for three-hop protocols[1]
	SCohFwdID	*cohfwdid_enable*	*cohfwdid_wdth*	Optional; required for three-hop protocols[1]
	SDataValid	*intport_split_tranx*		Optional; needed for split transaction responses
	SDataLast			Required
	MDataAccept	*intport_split_tranx*		Optional; needed for split transaction responses

Group	Signal	Enable Control Parameters	Width Control Parameters	Comments
Thread	MConnID	connid=0		Optional
	MDataThreadID	threads datahandshake=0	threads	Not allowed
	MThreadBusy	mthreadbusy threads	threads	Optional
	MThreadID	threads	threads	Optional
	MDataThreadBusy	*mdatathreadbusy* threads	threads	Optional
	SDataThreadBusy	sdatathreadbusy=0 threads	threads	Not allowed
	SThreadBusy	sthreadbusy threads	threads	Optional
	SThreadID	threads resp	threads	Optional
	SDataThreadID	threads resp	threads	Optional
Tags	MTagID	tags	tags	Optional
	MDataTagID	tags datahandshake=0	tags	Not allowed
	STagID	tags resp	tags	Optional
	MTagInOrder	taginorder		Optional
	STagInOrder	taginorder resp		Optional
Sideband	SReset_n or MReset_n	sreset=1 or mreset=1		Required
	(all others)	(all others)		Not allowed
Test	(all)	(all)		Not allowed

1. If coherent master is responsible for providing the system view (see Section 5.2 on page 74).

6.3.3.1 MCmd

The intervention port commands are shown in the Table 40 below. The commands that are write-like (including CohWriteBack, CohCopyBack, CohCopyBackInv, CohWriteInvalidate) have no associated write data during the request phase. If the port parameter `intport_writedata=1`, the write data transfer phase occurs on the intervention port during the data response phase for the self intervention. The mnemonics for the intervention port commands are prefixed by I_ to distinguish them from the main port commands.

Table 40 Intervention Port MCohCmd and MCmd Encoding

MCmd	Command	Mnemonic
0x0	Idle	IDLE
0x1–0x7	(Reserved)	(Reserved)
0x8	IntvReadOwn	I_RDOW
0x9	IntvReadShare	I_RDSH
0xA	IntvReadDiscard	I_RDDS
0xB	IntvReadShareAlways	I_RDSA
0xC	IntvUpgrade	I_UPG
0xD	IntvWriteBack	I_WB
0xE–0xF	(Reserved)	(Reserved)
0x10	IntvCopyBack	I_CB
0x11	IntvCopyBackInv	I_CBI
0x12	IntvInvalidate	I_I
0x13	IntvWriteInvalidate	I_WRI
0x14–0x1F	(Reserved)	(Reserved)

IntvReadOwn (0x8, I_RDOW)

This coherent command is used to read data from home with the intent to modify. This command is generated by processor stores that miss in the cache hierarchy. The slave responds with either SResp=OK (no data) or DVA (data).

IntvReadShared (0x9, I_RDSH)

This coherent command is used to read data from home with no intent to modify. This command is generated by processor loads that miss in the cache hierarchy. The slave responds with either SResp=OK (no data) or DVA (data).

IntvReadDiscard (0xA, I_RDDS)

This coherent command is used to read data from the processor caches and not cause any cache line state changes. It is generated by external agents (such as coherent DMA controllers) to read data from the processor cache hierarchy. The slave responds with either SResp=OK (no data) or DVA (data).

IntvReadShareAlways (0xB, I_RDSA)

This coherent command is used to read data from home with intent to never modify. This command is generated by processor instruction fetches. The slave responds with either SResp=OK (no data) or DVA (data).

IntvUpgrade (0xC, I_UPG)

This coherent command is used to request ownership of a shared cache line from the system. It is usually generated for processor stores which hit cache lines with shared states. This is a non-posted write. The slave responds with either SResp=OK (no data) or DVA (data).

The DVA response occurs when the local CPU has modified its data after the Upgrade command was sent by the originating CPU.

IntvWriteBack (0xD, I_WB)

This coherent command is used to writeback cache lines into home. This command is generated when a cache miss causes modified cache lines to be evicted from the cache hierarchy. This is a non-posted write. The slave responds with either SResp=OK (no data) or DVA (data).

The user has the option of placing the writeback data on this port instead of the main port (Port parameter intport_writedata=1). This option allows self-intervention data responses and “normal” responses to use the same datapaths.

For the self-intervention case, it is possible for the slave to response with OK instead of DVA. This case occurs if another CPU has gained ownership of the cache line before the original writeback transaction has been processed. The cache line would have been previously been written back for this change of ownership (Race between another core requesting the line and writeback completing at the originating CPU).

IntvCopyBack (0x10, I_CB)

This coherent command is used to writeback cache lines into home and the cache line is not evicted from the cache hierarchy. This command is generated by processor-specific cache management instructions. This is a non-posted write. The slave responds with either SResp=OK (no data) or DVA (data).

The user has the option of placing the writeback data on this port instead of the main port (Port parameter intport_writedata=1). This option allows self-intervention data responses and “normal” responses to use the same datapaths.

IntvCopyBackInv (0x11, I_CBI)

This coherent command is used to writeback cache lines into home and the cache line is evicted from the cache hierarchy. Functionally, it is the same as CohWriteBack, but this command is generated by processor-specific cache management instructions. This is a non-posted write. The slave responds with either SResp=OK (no data) or DVA (data).

The user has the option of placing the writeback data on this port instead of the main port (Port parameter intport_writedata=1). This option allows self-intervention data responses and “normal” responses to use the same datapaths.

IntvInvalidate (0x12, I_I)

This coherent command is used to purge data from the cache hierarchy. If a cache line contains the requested address, its state is set to invalid, regardless of the previous state. Typically used by coherent DMA controllers to remove stale copies of data from the cache hierarchy and also by processor-specific cache management instructions. This is a non-posted write. The slave responds with SResp=OK.

IntvWriteInvalidate (0x13, I_WRI)

This coherent command is used to inject new data into a coherent system by simultaneously invalidating a cache line from the system and updating its value at home. Typically used by coherent DMA controllers to write new values into home and remove stale copies from the cache hierarchy. This is a non-posted write. The slave responds with either SResp=OK (no data) or DVA (data). The original data is merged with the new data before it is written to home.

In some systems, a third port for coherent IO traffic can be used to allow external masters (such as DMA engines) to inject these WriteInvalidate commands into the coherent memory system without requiring the CPU main ports to set `writeresp_enable=1`.

6.3.3.2 SCohState

This signal indicates the cache line state of the slave cache and is part of the intervention response phase. Its encoding is identical to the description of the signal with the same name in the main port signal descriptions (see Table 36 on page 106).

6.3.3.3 MReqSelf

MReqSelf is an output of the master and an input to the slave. It is valid when MCmd is not IDLE. It indicates to the intervention slave that this intervention request is a result of a main port request which originated from the master port of this agent (i.e., it is a self-intervention). This bit is typically asserted by the interconnect. The concept of self-intervention is critical in OCP 3.0 (along with a serialization point) to enforce global order in the coherent system.

6.3.3.4 MCohID

MCohID specifies the target of the request. It is valid when MCmd is not IDLE. For directory based coherence it is used at the intervention ports to indicate the target of the response. For an interrupt command from the main port it is used to indicate the target of the command. This is an optional signal which could be used in three hop protocols when the coherent master also provides the system view (see Section 5.2 on page 74).

6.3.3.5 SCohID

SCohID specifies the target of the response. It is valid when SResp is not NULL. For directory based coherence it is used at the intervention ports to indicate the target of the response. This is an optional signal which could be used in three hop protocols when the coherent master also provides the system view (see Section 5.2 on page 74).

6.3.3.6 McohFwdID

MCohFwdID specifies the target for a three hop transaction. It is valid when MCmd is not IDLE. Its main use is meant in directory based coherence where it is used at the intervention port to signal to the target that if a three hop transaction is required, then this is the address of the final target. This is an optional signal which could be used in three hop protocols when the coherent master also provides the system view (see Section 5.2 on page 74).

6.3.3.7 SDataValid

SDataValid is a optional signal. This signal is included if the port parameter `intport_split_tranx` is set equal to 1. It is an output from the slave and an input to the Master to denote that snoop intervention data is valid on SData.

6.3.3.8 SDataLast

SDataLast is a required signal. It is an output from the slave and an input to the Master to denote that the last data beat of the transfer is valid on SData.

6.3.3.9 MDataAccept

MDataAccept is an optional signal. This signal is included if the port parameter `intport_split_tranx` is set equal to 1. It is an output from the Master and an input to the slave to denote that the Master can accept snoop intervention data from the slave.

6.3.3.10 MDataThreadBusy

MDataThreadBusy is an optional signal used if threads have been enabled for the Intervention Port. The master notifies the slave that it cannot accept any data associated with certain threads. This field is a vector (one bit per thread). A value of 1 on any given bit indicates that the thread associated with that bit is busy. Bit 0 corresponds to thread 0, and so on. This signal is enabled by the port parameter `mdatathreadbusy`. The semantics of this signal are controlled by the port parameters `mdatathreadbusy_exact` and `mdatathreadbusy_pipelined`.

6.3.4 Signal Groups

The following table shows the Intervention Port signals placed into specific groups. All signals within one group as asserted at the same time.

Table 41 Intervention Port Signal Groups

Group	**Signal**	**Condition**
Request	MAddr	always
	MCmd	always
	MAddrSpace	always
	MReqInfo	Optional
	MAtomicLength	Optional
	MBurstLength	always
	MBurstPrecise	always
	MBurstSeq	always
	MBurstSingleReq	always
	MReqSelf	always
	MCohID	Optional
	MCohFwdID	Optional
	MTagID	Optional
	MTagInOrder	Optional
	MThreadsID	Optional
Response	SResp	always
	SRespInfo	Optional
	SCohState	always
	STagID	Optional
	STagInOrder	Optional
	SCohID	Optional
	SThreadID	Optional
RespDataHandShake	SData	Always
	SDataValid	Optional
	SDataLast	Always
	SDataInfo	Optional
	STagID	Optional
	STagInOrder	Optional
	SCohID	Optional
	SThreadID	Optional
	SDataThreadID	Optional

6.3.5 Transfer Phases

Table 42 shows the transfer phases allowed given specific values of the signal MReqSelf and the parameter `intport_writedata`.

Table 42 Intervention Port transfer phases

MCmd	**Phases**		
	MReqSelf=0	**MReqSelf=1**	
		intport_writedata=1	**intport_writedata=0**
I_RDOW	Request; Response; RespDataHandShake[1]	Request; Response;	Request; Response;
I_RDSH	Request; Response; RespDataHandShake[1]	Request; Response	Request; Response
I_RDDS	Request; Response; RespDataHandShake[1]	Request; Response	Request; Response
I_RDSA	Request; Response; RespDataHandShake[1]	Request; Response	Request; Response
I_UPG	Request; Response; RespDataHandShake[1]	Request; Response	Request; Response
I_WB	Request; Response[2]	Request; Response; RespDataHandShake[3]	Request; Response
I_CB	Request; Response[2]	Request; Response; RespDataHandShake[3]	Request; Response
I_CBI	Request; Response[2]	Request; Response; RespDataHandShake[3]	Request; Response
I_I	Request; Response	Request; Response	Request; Response
I_WRI	Request; Response; RespDataHandShake[4]	Request; Response RespDataHandShake[4]	Request; Response RespDataHandShake

1. RespDataHandShake group active if cache line was in M or O state in local cache. If port parameter `intport_estate_c2c=1`, then RespDataHandShake group also active if cache line was in E state in local cache.
2. The request and response transfers are not needed in directory based protocols since the intervention requests are only directed to the original requester. In snoop-based protocols, some implementations may choose to broadcast the intervention requests, in which case these transfers are needed.
3. RespDataHandShake phase might not occur if cache line ownership has been passed to another CPU subsequent to when the originating CC_WB command was issued. WriteBack Data is supplied with self-intervention response
4. RespDataHandShake group only active if the cache line in the local cache was in the M or O state.

6.3.6 Phase Ordering within a Transfer

The intervention port follows the legacy OCP phase ordering rules except for the following:

- If the port parameter `intport_split_tranx`=1 then it is allowed that the Response phase can begin before the associated RespDataHandShake phase.
- If the port parameter `intport_split_tranx`=1 then it is allowed that the Response phase can end before the associated RespDataHandShake phase.

These are optimizations to allow forwarding of the local cache tag lookups before the local cache data array lookup is completed.

6.3.7 Transfer Effects

All transaction requests on the Intervention Port require a response from the slave. Some of the transactions may also cause data transfer on the port.

The SCohState signal reports the cache line state prior to the intervention.

If port parameter `intport_split_tranx`=0, then the SResp signals reports whether the local slave will deliver data or not. The Response phase is coincident with the data transfer phase.

If port parameter `intport_split_tranx`=1, then the SDataValid signal reports when the local slave delivers data. The response phase is single cycle and occurs before the data transfer phase. The Response is reported on the SResp signal.

Table 43 Summary of Transfer Effects

Condition(s)	**SResp Behavior**
IntvReadOwn, **IntvReadShared**, **IntvReadDiscard**, **IntvReadSharedAlways**, **IntvUpgrade**	
MReqSelf = b0, Cache Line State = M, O	DVA (data transfer)
intport_estate_c2c=1, MReqSelf = b0, Cache Line State = E	DVA (data transfer)
All other cases	OK (no data transfer)
IntvWriteBack	
intport_writedata=1, MReqSelf = b1, Cache Line State = M, O	DVA (data transfer)
All other cases	OK (no data transfer)
IntvCopyBack, **IntvCopyBackInv**	
intport_writedata=1, Cache Line State = M, O	DVA (data transfer)
All other cases	OK (no data transfer)
IntvWriteInvalidate	
Cache Line State = M, O	DVA (data transfer)
All other cases	OK (no data transfer)
IntvInvalidate	
All cases	OK (no data transfer)

7 Interface Configuration File

The interface configuration file describes a group of signals, called a bundle. For OCP interfaces, the bundle is pre-defined, and no interface configuration file is required. If you are using an interface other than OCP in your core RTL configuration file, the interface configuration file is required.

Name the file <bundle-name>_intfc.conf where bundle-name is the name given to the bundle that is being defined in the file.

7.1 Lexical Grammar

The lexical conventions used in the interface configuration file are:

```
<name> : (<letter> | '_') (<letter> | '_' | <digit>)*
<letter> : 'a' .. 'z' | 'A' .. 'Z'
<digit>  : '0' .. '9'

<number> : <integer> | <float>

<integer> : <decimal_integer> | <hexadecimal_integer> | <octal_integer> |
            <binary_integer>
<decimal_integer> : <digit>+
<hexadecimal_integer> : '0x'<hexadecimal_digit>+
<hexadecimal_digit> : <digit> | 'a' .. 'f' | 'A' .. 'F'
<octal_integer> : '0o'<octal_digit>+
<octal_digit> : '0' .. '7'
<binary_integer> : '0b'<binary_digit>+
<binary_digit> : '0' | '1'

<float> : <mantissa> [<exponent>]
<mantissa>: (<decimal_integer> '.') |('.' <decimal_integer>) |
            (<decimal_integer> '.' <decimal_integer>)
<exponent>: ('e' | 'E') ['+' | '-'] <decimal_integer>
```

7.2 Syntax

The interface configuration file is written using standard Tcl syntax. Syntax is described using the following conventions:

Symbol	**Meaning**
[ ]	optional construct
|	or, alternate constructs
*	zero or more repetitions
+	one or more repetitions
< >	enclose names of syntactic units
( )	are used for grouping
{ }	are part of the format and are required. An open brace must always appear on the same line as the statement
\	line continuation character
#	comments

The syntax of the interface configuration file is:

```
version <version_string>
bundle <bundle_name> \
    [ revision <revision_string> ] {<bundle_stmt>+}
```

where:

```
<bundle_stmt>:
      | interface_types <interface_type-name>+
      | net <net_name> {<net_stmt>*}
      | proprietary <vendor_code> <organization_name>
          {<proprietary_statements>}

<net_stmt>:
      | direction (input|output|inout)+
      | width <number-of-bits>
      | vhdl_type <type-string>
      | type <net-type>
      | proprietary <vendor_code> <organization_name>
         {<proprietary_statements>}
```

The file must contain a single version statement followed by a single bundle statement. The bundle statement must contain exactly one `interface_types` statement, and one or more `net` statements. Each net statement must contain exactly one direction statement and may contain additional statements of other types.

`version`

The `version` statement identifies the version of the interface configuration file format. The version string consists of major and minor version numbers separated by a decimal. The current version is 4.5.

`bundle`

The `bundle` statement is required and indicates that a bundle is being defined instead of a core or a chip. Make the bundle-name the same name as the one used in the interface configuration file name.

Use a `bundle_name` of ocp for OCP 1.0 bundles, ocp2 for OCP 2.x bundles, and ocp3 for OCP 3.x bundles. The optional `revision_string` identifies a specific revision for the bundle. If not provided, the `revision_string` defaults to 0. The pre-defined ocp, ocp2, and ocp3 bundles use the default value of `revision_string` to refer to the 1.0, 2.0, and 3.0 versions of the *OCP Specification*, respectively. For ocp2 bundles, set `revision_string` to 2 to refer to the 2.2 version of the *OCP Specification*.

`interface_types`

The `interface_types` statement lists the legal values for the interface types associated with the bundle. Interface types are used by the toolset in conjunction with the direction statement to determine whether an interface uses a net as an input or output signal. This statement is required and must have at least one type defined.

Predefined interface types for OCP bundles are slave, master, system_slave, system_master, and monitor. These are explained in Table 18 on page 35.

`net`

The `net` statement defines the signals that comprise the bundle. There should be one `net` statement for each signal that is part of the bundle. A net can also represent a bus of signals. In this case the net width is specified using the `width` statement. If no width statement is provided, the net width defaults to one. A bundle is required to contain at least one net. The net-name field is the same as the one used in the net-name field of the port statements in the core RTL file described in Chapter 8.

`proprietary`

For a description, see "Proprietary Statement" on page 137.

`direction`

The `direction` statement indicates whether the net is of type input, output, or inout. This field is required and must have as many direction-values as there are interface types. The order of the values must duplicate the order of the interface types in the interface_types statement. The legal values are input, output, and inout.

`vhdl_type`

By default VHDL signals and ports are assumed to be `std_logic` and `std_logic_vector`, but if you have ports on a core that are of a different type, the `vhdl_type` command can be used on a net. This type will be used only when soccomp is run with the `design_top=vhdl` option to produce a VHDL top-level netlist.

`type`

The `type` statement specifies that a net has special handling needs for downstream tools such as synthesis and layout. Table 44 shows the allowed <net-type> options. If no <net-type> is specified, normal is assumed.

Table 44 net-type Options

<net-type>	**Description**
clock	clock net
clock_sample	clock sample net
jtag_tck	JTAG test clock
jtag_tdi	JTAG test data in
jtag_tdo	JTAG test data out
jtag_tms	JTAG test mode select
jtag_trstn	JTAG test logic reset
normal	default for nets without special handling needs
reset	reset net
scan_enable	scan enable net, serves as mode control between functional and scan data inputs
scan_in	scan input net
scan_out	scan output net
test_mode	test mode net, puts logic into a special mode for use during production testing

proprietary

For a description, see ”Proprietary Statement” on page 137.

The following example defines an SRAM interface. The bundle being defined is called sram16.

```
bundle "sram16" {

   # Two interface types are defined, one is labeled
   # "controller" and the other is labeled "memory"
   interface_types controller memory

   # A net named Address is defined to be part of this bundle.
   net "Address" {

      # The direction of the "Address" net is defined to be
      # "output" for interfaces of type "controller" and "input"
      # for interfaces of type "memory".
      direction output input

      # The width statement indicates that there are 14 bits in
      # the "Address" net.
      width 14
   }
   net "WData" {
      direction output input
      width 16
   }
   net "RData" {
      # The direction of the "RData" net is defined to be
      # "input" for bundle of type "controller" and "output" for
      # bundles of type "memory".
      direction input output
      width 16
   }
   net "We_n" {
      direction output input
   }
   net "Oe_n" {
      direction output input
   }
   net "Reset" {
      direction output input
      type reset
   }
# close the bundle
}
```

8 *Core RTL Configuration File*

The required core RTL configuration file provides a description of the core and its interfaces. The name of the file needs to be <corename>_rtl.conf, where corename is the name of the module to be used. For example, the file defining a core named uart must be called uart_rtl.conf.

For a description of the lexical grammar, see page 123.

8.1 Syntax

The core RTL configuration file is written using standard Tcl syntax. Syntax is described using the following conventions:

[ ]	optional construct
|	or, alternate constructs
*	zero or more repetitions
+	one or more repetitions
<>	enclose names of syntactic units
()	are used for grouping
{ }	are part of the format and are required. An open brace must always appear on the same line as the statement
#	comments

The syntax for the core RTL configuration file is:

```
version <version_string>
module <core_name> {<core_stmt>+}
```

`core_name` is the name of the core being described and:

```
<core_stmt>:
    | icon <file_name>
    | core_id <vendor_code> <core_code> <revision_code>
     [<description>]
    | interface <interface_name> bundle <bundle_name> [revision
     <revision_string>]
       [{<interface_body>*}]
    | addr_region <name> {<addr_region_body>*}
    | proprietary <vendor_code> <organization_name>
       {<proprietary_statements>}
```

The file must contain a single version statement followed by a single module statement. The module statement contains multiple core statements. One core_id must be included. At least one interface statement must be included. One icon statement and one or more addr_region and proprietary statements may also be included.

8.2 Components

This section describes the core RTL configuration file components.

Version Statement

The version statement identifies the version of the core RTL configuration file format. The version string consists of major and minor version numbers separated by a period. The current version of the file is 4.5.

Icon Statement

This statement specifies the icon to display on a core. The syntax is:

```
icon <file_name>
```

file_name is the name of the graphic file, without any directory names. Store the file in the design directory of the core. For example:

```
icon "myCore.ppm"
```

The supported graphic formats are GIF, PPM, and PGM. Graphics should be no larger than 80x80 pixels. Since the text used for the core is white, use a dark background for your icon, otherwise it will be difficult to read.

Core_id Statement

The core_id statement provides identifying information to the tools about the core. This information is required. Syntax of the core_id statement is:

```
core_id <vendor_code> <core_code> <revision_code> [<description>]
```

where:

vendor_code	An OCP-IP-assigned vendor code that uniquely identifies the core developer. OCP-IP maintains a registry of assigned vendor codes. The allowed range is 0x0000 - 0xFFFF. Use 0x5555 to denote an anonymous vendor. For a list of codes check www.ocpip.org.

core_code	A developer-assigned core code that (in combination with the vendor code) uniquely identifies the core. OCP-IP provides suggested values for common cores. See “Defined Core Code Values” on page 131. The allowed range is 0x000 - 0xFFF.
revision_code	A developer-assigned revision code that can provide core revision information. The allowed range is 0x0–0xF.
description	An optional Tcl string that provides a short description of the core.

Defined Core Code Values

```
0x000 - 0x7FF: Pre-defined
   0x000 - 0x0FF: Memory
   Sum values from following choices:
      ROM:
      0x0: None
      0x1: ROM/EPROM
      0x2: Flash (writable)
      0x3: Reserved
      SRAM:
      0x0: None
      0x4: Non-pipelined SRAM
      0x8: Pipelined SRAM
      0xC: Reserved
      DRAM:
      0x00: None
      0x10: DRAM (trad., page mode, EDO, etc.)
      0x20: SDRAM (all flavors)
      0x30: RDRAM (all flavors)
      0x40: Several
      0x50: Reserved
      0x60: Reserved
      0x70: Reserved
      Built-in DMA:
      0x00: No
      0x80: Yes
   Values from 0x000 - 0x0FF are defined/reserved
   Example: Memory controller supporting only SDRAM & Flash
     would have <cc> = 0x2 + 0x20 = 0x022

   0x100 - 0x1FF: General-purpose processors
   Sum values from following choices plus offset 0x100:
      Word size:
      0x0: 8-bit
      0x1: 16-bit
      0x2: 32-bit
      0x3: 64-bit
      0x4 - 0x7: Reserved
      Embedded cache:
      0x0: No cache
```

```
         0x8: Cache (Instruction, Data, combined, or both)
         Processor Type:
         0x00: CPU
         0x10: DSP
         0x20: Hybrid
         0x30: Reserved
      Only values from 0x100 - 0x13F are defined/reserved
      Example: 32-bit CPU with embedded cache
         would have <cc> = 0x100 + 0x2 + 0x8 + 0x00 = 0x10A

      0x200 - 0x2FF: Bridges
      Sum values from following choices plus offset 0x200:
         Domain:
         0x00 - 0x7F: Computing
            0x00 - 0x3F: PC's
            0x00: ISA (inc. EISA)
            0x01 - 0x0F: Reserved
            0x10: PCI (33MHz/32b)
            0x11: PCI (66MHz/32b)
            0x12: PCI (33MHz/64b)
            0x13: PCI (66MHz/64b)
            0x14 - 0x1F: AGP, etc.
            0x40 - 0x7F: Reserved
         0x80 - 0xBF: Telecom
            0xA0 - 0xAF: ATM
            0xA0: Utopia Level 1
            0xA1: Utopia Level 2
            ...
         0xC0 - 0xFF: Datacom

      0x300 - 0x3FF: Reserved

      0x400 - 0x5FF: Other processors
      (enumerate types: MPEG audio, MPEG video, 2D Graphics,
       3D Graphics, packet, cell, QAM, Vitterbi, Huffman,
       QPSK, etc.)

      0x600 - 0x7FF: I/O
      (enumerate types: Serial UART, Parallel, keyboard, mouse,
       gameport, USB, 1394, Ethernet 10/100/1000, ATM PHY,
       NTSC, audio in/out, A/D, D/A, I2C, PCI, AGP, ISA,
       etc.)

   0x800 - 0xFFF: Vendor-defined
            (explicitly left up to vendor)
```

Interface Statement

The interface statement defines and names the interfaces of a core. The interface name is required so that cores with multiple interfaces can specify to which interface a particular connection should be made. Syntax for the interface statement is:

```
interface <interface_name> bundle <bundle_name> [revision
<revision_string>]
[{<interface_body>*}]
```

Parameters lacking a default must be specified using a param statement. For a list of the required parameters, see Section 4.9.6 on page 67. All other interface body statements are optional

The <bundle_name> must be a defined bundle such as ocp or ocp2 or a bundle specified in an interface configuration file as described on page 123. The optional <revision_string> must match that of the referenced bundle. Different interfaces can refer to different revisions of the same bundle. The pre-defined ocp, ocp2, and ocp3 bundles use the default revision_string to refer to the 1.0, 2.0, or 3.0 versions of the *OCP Specification*, respectively. For ocp2 bundles, set revision_string to 2 to refer to the 2.2 version of the *OCP Specification*.

In the following example, an interface named xyz is defined as an OCP 3.0 bundle. The quotation marks around xyz are not required but help to distinguish the format.

```
interface "xyz" bundle ocp3 revision 0

<interface_body>:
        | interface_type <type_name>
        | port <port_name> net <net_name>
        | reference_port <interface_name>.<port_name> net <net_name>
        | prefix <name>
        | param <name> <value> [{(<attribute> <value>)*}]
        | subnet <net_name> <bit_range_list> <subnet_name>
        | location (n|e|w|s|) <number>
        | proprietary <vendor_code> <organization_name>
           {<proprietary_statements>}
```

Ports on a core interface may have names that are different than the nets defined in the bundle type for the interface. In this case, each port in the interface must be mapped to the net in the bundle with which it is associated. Mapping links the module port <prefix><port_name> with the bundle <net_name>.

The default rules for mapping are that the port_name is the same as the net_name and the prefix is the name of the interface. These rules can be overridden using the Port and Prefix statements.

Interface_type Statement

The interface_type statement defines characteristics of the bundle. Typically, the different types specify whether the core drives or receives a particular signal within the bundle. Syntax for the interface_type statement is:

```
interface_type <type_name>
```

The type_name must be a type defined in the bundle definition. If the bundle is OCP, the allowed types are: master, system_master, slave, system_slave, and monitor as described in Table 18 on page 35. To define a type, specify it in the interface configuration file (described on page 123).

Port Statement

Use the port statement to map a single port corresponding to a signal that is defined in the bundle. Syntax for the port statement is:

```
port <port_name> net <net_name>
```

The module port named <prefix><port name> implements the <net_name> function of the bundle. The legal net_name values are defined in the bundle definition. For OCP bundles, the net names are defined in Section 3 on page 13.

Reference_port Statement

The reference_port statement re-directs a net to another bundle. Syntax for the port statement is:

```
reference_port <interface_name>.<port_name> net <net_name>
```

The interface (in which the reference_port is declared) does not have the reference port and the bundle does not have the reference net. The reference_port statement declares that the net is internally connected to the given port of the referenced interface. For example, consider the following two interfaces:

```
interface tp bundle ocp {
   reference_port ip.Clk_i net Clk
   reference_port ip.SReset_ni net MReset_n
   reference_port ip.EnableClk_i net EnableClk
   port Control_i net Control
   port MCmd_i net MCmd
}

interface ip bundle ocp {
   port Clk_i net Clk
   port SReset_ni net SReset_n
   port EnableClk_i net EnableClk
   port Control_i net Control
   port MCmd_o net MCmd
}
```

Figure 14 Reference Port

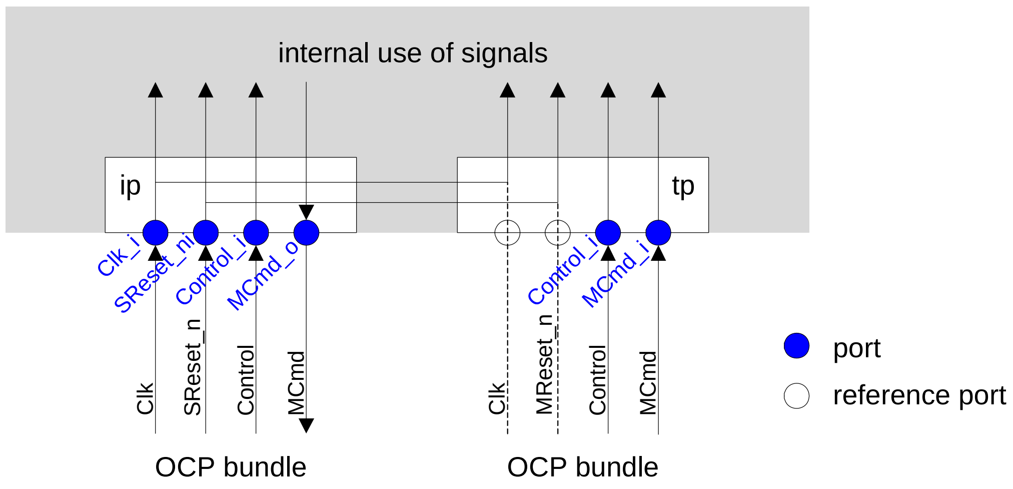


Figure 14 illustrates the operation of a reference port. In the interface tp, no ports exist for bundle signals Clk, EnableClk, and MReset_n. Neither do the bundle signals themselves exist. Instead, they reference the corresponding ports in the ip interface and nets in the bundle connected to that interface. The internal signals in the tp interface that would have been connected to the Clk, EnableClk, and MReset_n signals of the OCP bundle connected to the tp interface are instead connected to the referenced ports in the ip interface.

Prefix Command

The prefix command applies to all ports in an interface. It supplies a string that serves as the prefix for all core port names in the interface. Syntax for the prefix command is:

```
prefix <name>
```

For example, the statement prefix `external_` specifies that the names for all ports in the interface are of the form `external_*`.

If the prefix command is omitted, the interface name will be inserted as the default prefix. To omit the prefix from the port name, specify it as an empty string, that is `prefix ""`.

Configurable Interfaces Parameters

For configurable interfaces, parameters specify configurations. The specific parameters for OCP are described in Chapters 3 and 4 and summarized in Table 29 on page 68. The syntax for setting a parameter is:

```
param <name> <value> [{(<attribute> <value>)*}]
<value>: <number>|<name>
<attribute>: tie_off|width
```

If the parameter is used to configure a signal, the attribute list can be used to attach additional values to that signal. The supported attributes are the tie-off (if the signal is configured out of the interface) and the signal width (if the signal is configured into the interface). Specifying the signal width using an attribute attached to the signal parameter is equivalent to using the corresponding signal width parameter but the attribute syntax is preferred. The width of the signals MData, SData, MByteEn, and MDataByteEn are derived from the single `data_wdth` parameter, so cannot have their width specified using an attribute. For example, an OCP might be configured to include an interrupt signal as follows.

```
param interrupt 1
```

The following example shows the MBurstLength field tied off to a constant value of 4.

```
param burstlength 0 {tie_off 4}
```

The following code shows two equivalent ways of setting the address width to 16 bits though the second method is preferred.

```
param addr_wdth 16

param addr 1 {width 16}
```

Subnet Statement

The subnet statement assigns names to bits or contiguous bit-fields within a net. Syntax for the subnet statement is:

```
subnet <net_name> <bit_range_list> <subnet_name>
<bit_range_list>: <bit_range>[,<bit_range>]*
<bit_range>: <bit_number>[:<bit_number>]
```

The subnet_name is assigned to the bit_range within the given net_name. Bit_range can be either a single bit or a range. Subnet_name is a Tcl string.

For example bit 3 of the MReqInfo net may be assigned the name “cacheable” as follows:

```
subnet MReqInfo 3 cacheable
```

Location Statement

The location statement provides a way for the core to indicate where to place this interface when a schematic symbol for the core is drawn. The location is specified as a compass direction of north(n), south(s), east(e), west(w) and a number. The number indicates a percentage from the top or left edge of the block. Syntax for the location statement is:

```
location (n|e|w|s) <number>
```

To place an interface on the bottom (south-side) in the middle (50% from the left edge) of the block, for example, use this definition:

```
location s 50
```

Address Region Statement

The address region statement specifies address regions within the complete address space of a core. It allows you to give a symbolic name to a region, and to specify its base, size, and behavior.

```
addr_region <name> {<addr_region_body>*}
```

where:

```
<addr_region_body>: addr_base <integer> | addr_size <integer>
       | addr_space <integer>
       | proprietary <vendor_code> <organization_name>
       {<proprietary_statements>}
```

- The addr_base statement specifies the base address of the region being defined and is specified as an integer.
- The addr_size statement similarly specifies the size of the region.
- The addr_space statement specifies to which OCP address space the region belongs. If the addr_space statement is omitted, the region belongs to all address spaces.

Proprietary Statement

The proprietary statement enables proprietary extensions of the core RTL configuration file syntax. Standard parsers must be able to ignore the extensions, while proprietary parsers can extract additional information about the core. Syntax for the proprietary statement is:

```
proprietary <vendor_code> <organization_name>
       {<proprietary_statements>}
```

The vendor_code uniquely identifies the vendor associated with the proprietary extensions and is described in more detail on page 130.

The organization_name specifies the name of the organization that specified the extensions. Any number of proprietary statements can be included between the braces but must follow legal Tcl syntax.

The proprietary statement can be included at multiple levels of the syntax hierarchy, allowing it to use scoping to imply context. If multiple proprietary statements are included in a single scope, the parser must process these in an additive fashion.

8.3 Sample RTL Configuration File

The format for a core RTL configuration file for a core is shown in Example 1.

Example 1 Sample flashctrl_rtl.conf File

```
# define the module
version 4.5

module flashctrl {
core_id 0xBBBB 0x001 0x1 “Flash/Rom Controller”

   # Use the Vista icon
   icon “vista.ppm”

   addr_region “FLASHCTRL0” {
   addr_base 0x0
   addr_size 0x100000
   }

   # one of the interfaces is an OCP slave using the pre-defined ocp2 bundle
   # Revision is "1", indicating compliance with OCP 2.1
   interface tp bundle ocp2 revision 1 {

   # this is a slave type ocp interface
   interface_type slave

      # this OCP is a basic interface with byteen support plus a named SFlag
      # and MReset_n
      param mreset 1
      param sreset 0
      param byteen 1
      param sflag 1 {width 1}
      param addr 1 {width 32}
      param mdata 1 {width 64}
      param sdata 1 {width 64}

      prefix tp
      # since the signal names do not exactly match the signal
      # names within the bundle, they must be explicitly linked
      port Reset_ni net MReset_n
      port Clk_i net Clk
      port TMCmd_i    net MCmd
      port TMAddr_i   net MAddr
      port TMByteEn_i  net MByteEn
      port TMData_i   net MData
      port TCCmdAccept_o net SCmdAccept
      port TCResp_o   net SResp
      port TCData_o   net SData
      port TCError_o   net SFlag
      # name SFlag[0] access_error
      subnet SFlag 0 access_error

      # stick this interface in the middle of the top of the module
      location n 50

   } # close interface tp defininition

   # The other interface is to the flash device defined in an interface file
   # Define the interface for the Flash control
   interface emem bundle flash {

      # the type indicates direction and drive of the control signals
       interface_type controller

      # since this module has direction indication on some of the signals
      # ('_o','_b') and is missing assertion level indicators '_n' on
      # some of the signals, the names must again be directly linked to
      # the signal names within the bundle
       port Addr_o    net addr
       port Data_b    net data
       port OE        net oe_n
       port WE        net we_n
       port RP        net rp_n
       port WP        net wp_n

       # all of the signals on this port have the prefix 'emem_'
       prefix "emem_"

       # stick this interface in the middle of the bottom of the module
       location s 50

    } # close interface emem defininition

} # close module definition
```

The flash bundle is defined in the following interface configuration file. See Section 7 on page 123 for the syntax definition of the interface configuration file.

```
bundle flash {
   #types of flash interfaces
   #controller: flash controller; flash: flash device itself.
   interface_types controller flash
   net addr {
      #Address to the Flash device
      direction output input
      width 19
   }
```

```
      net data {
         #Read or Write Data
         direction inout inout
         width 16
      }
      net oe_n {
         # Output Enable, active low.
         direction output input
      }
      net we_n {
         # Write Enable, active low.
         direction output input
      }
      net rp_n {
         # Reset, active low.
         direction output input
      }
      net wp_n {
         # Write protect bit, Active low.
         direction output input
      }
   }
```

9 Core Timing

To connect two entities together, allowing communication over an OCP interface, the protocols, signals, and pin-level timing must be compatible. This chapter describes how to define interface timing for a core. This process can be applied to OCP and non-OCP interfaces.

Use the core synthesis configuration file to set timing constraints for ports in the core. The file consists of any of the constraint sections: port, max delay, and false path. If the core has additional non-OCP clocks, the file should contain their definitions.

When implementing IP cores in a technology independent manner it is difficult to specify only one timing number for the interface signals, since timing is dependent on technology, library and design tools. The methodology specified in this chapter allows the timing of interface signals to be specified in a technology independent way.

To make your core description technology independent use the technology variables defined in the *Core Preparation Guide*. The technology variables range from describing the default setup and clock-to-output times for a port to defining a high drive cell in the library.

9.1 Timing Parameters

There is a set of minimum timing parameters that must be specified for a core interface. Additional optional parameters supply more information to help the system designer integrate the core. Hold-time parameters allow hold time checking. Physical-design parameters provide details on the assumptions used for deriving pin-level timing.

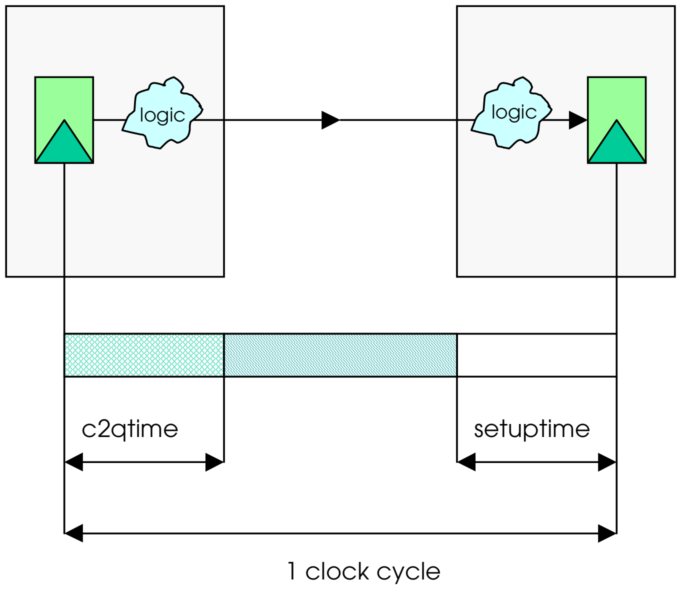
9.1.1 Minimum Parameters

At a minimum, the timing of an OCP interface is specified in terms of two parameters:

- `setuptime` is the latest time an input signal is allowed to change before the rising edge of the OCP clock.
- `c2qtime` is the latest time an output signal is guaranteed to become stable after the rising edge of the OCP clock.

Figure 15 shows the definition of `setuptime` and `c2qtime`. See Section 9.2.5.1 on page 149 for a description of these parameters.

Figure 15 OCP Timing Parameters



9.1.2 Hold-time Parameters

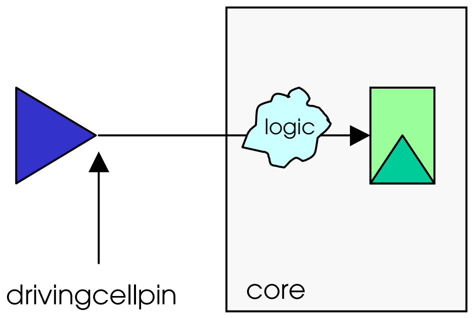
Hold-time parameters are needed to allow the system integrator to check hold time requirements. On the output side, `c2qtimemin` specifies the minimum time for a signal to propagate from a flip-flop to the given output pin. On the input side, `holdtime` specifies the minimum time for a signal to propagate from the input pin to a flip-flop.

9.1.3 Technology Variables

To give meaning to the timing values, timing requirements on input and output pins must be accompanied by information on the assumed environment for which these numbers are determined. This information also adds detail on the expected connection of the pin.

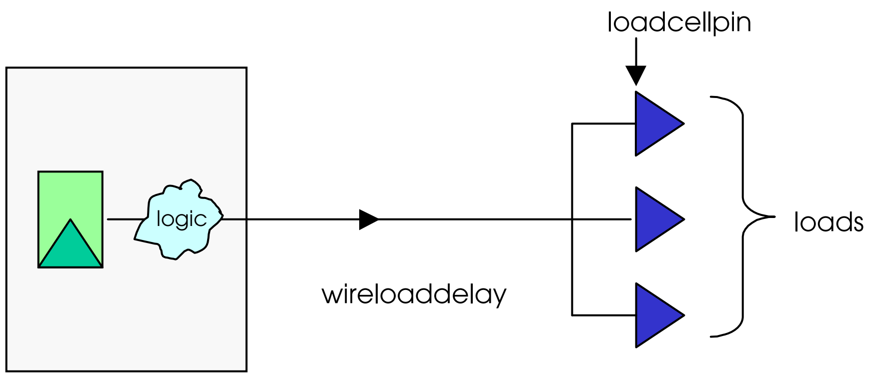
For an input signal, the parameter `drivingcellpin` indicates the cell library name for a cell representative of the strength of the driver that needs to be used to drive the signal. This is shown in Figure 16.

Figure 16 Driver Strength



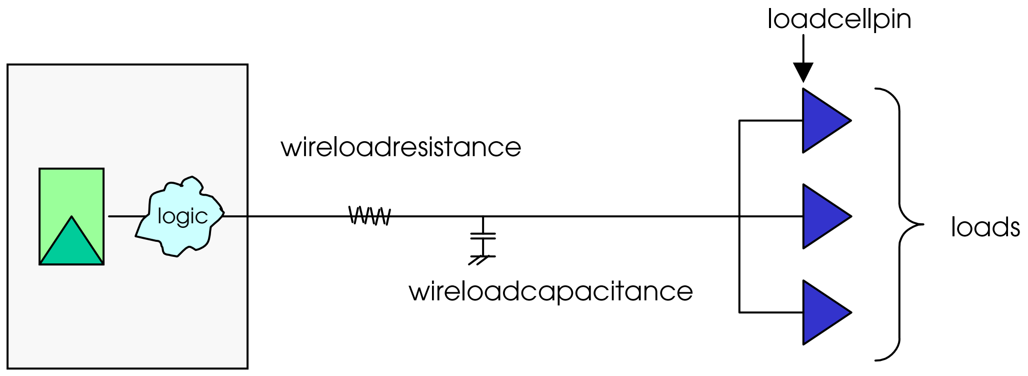
For an output signal, the variable `loadcellpin` indicates the input load of the gate that the signal is expected to drive. The variable `loads` indicates how many loadcellpins the signal is expected to drive. Additionally, information on the capacitive load of the wire must be included. There are two options. Either the variable `wireloaddelay` can be specified, as shown in Figure 17. Or, the combination `wireloadcapacitance`/`wireloadresistance` must be specified, as shown in Figure 18.

Figure 17 Variable Loads - wireloaddelay



For instructions on calculating a delay, refer to the *Synopsys Design Compiler Reference*.

Figure 18 Variable Loads - wireloadresistance/wireloadcapacitance



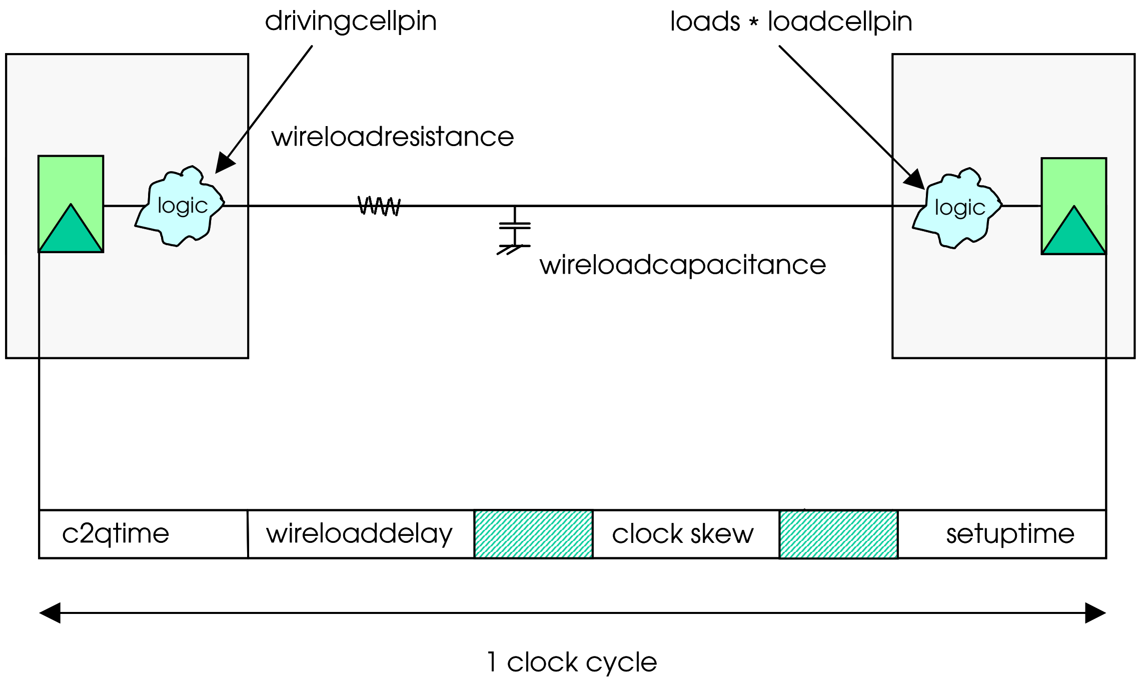
9.1.4 Connecting Two OCP Cores

Figure 19 shows the timing model for interconnecting two OCP compliant cores.

The sum of `setuptime`, `c2qtime` and wire delay must be less than the clock period or cycle time minus the clock-skew. Similarly, the minimum clock-cycle for two cores to interoperate is determined by the maximum of the sum of `c2qtime`, `setuptime`, wire delay and clock-skew over all interface signals.

The wireload delay is defined by either the variable `wireloaddelay` or the set `wireloadcapacitance/wireloadresistance`.

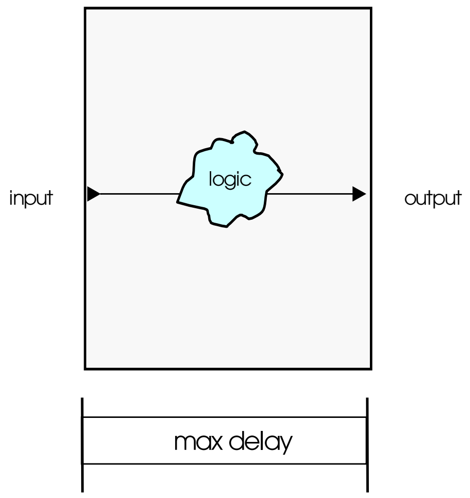
Figure 19 Connecting Two OCP Compliant Cores



9.1.4.1 Max Delay

In addition to the `setup` and `c2qtime` paths for a core, there may also be combinational paths between input and output ports. Use `maxdelay` to specify the timing for these paths.

Figure 20 Max Delay Timing



9.1.4.2 False Paths

It is possible to identify a path between two ports as being logically impossible. Such paths can be specified using the `falsepath` constraint syntax.

For instructions on specifying the core's timing parameters, see Section 9.2.7 on page 154.

9.2 Core Synthesis Configuration File

The core synthesis configuration file contains the following sections:

Version
: Specifies the current version of the synthesis configuration file format. The current version is 1.3.

Clock
: Describes clocks brought into the core.

Area
: Defines the area in gates of the core.

Port
: Defines the timing of IP block ports.

Max Delay
: Specifies the delay between two ports on a combinational path.

False Path
: Specifies that a path between input and output ports is logically impossible.

9.2.1 Syntax Conventions

Observe the following syntax conventions:

- Enclose all expr statements within braces { }, to differentiate between expressions that are to be evaluated while the file is being parsed (without braces) and those that are to be evaluated during synthesis constraint file generation (with braces).
- Although not required by Tcl, enclose strings within quotation marks "", to show that they are different than keywords.
- Specify keywords using lower case.

Parameter values are specified using Tcl syntax. Expressions can use any of the technology or environment variables, and any of the following variables:

clockperiod
: This variable should only be used in calculations of timing values for ports. When evaluating expressions that use `$clockperiod`, the program will determine which clock the port is relative to, determine its period (in nanoseconds), and apply that value to the equation. For example:

```
port "in" {
    setuptime {[expr $clockperiod * .5]}
}
```

rootclockperiod

This variable is set to the period of the main system clock, usually referred to as the root clock. It is typically used when a value needs to be a multiple of the root clock, such as for non-OCP clocks. For example:

```
clock "myClock" {
    period {[expr $rootclockperiod * 4]}
}
```

The design_syn.conf file can also use conditional settings of the parameters in the design as outlined by the following arrays. These variables are only used at the time the file is read into the tools.

param

This array is indexed by the configuration parameters that can be found on a particular instance. Only use `param` for core_syn.conf files since it is only applicable to the instance being processed. For example:

```
if { $param("dma_fd") == 1 } {
   port "T12_ipReset_no" {
    c2qtime {[expr $clockperiod * 0.7]}
   }
}
```

chipparam

This array is indexed by the configuration parameters that are defined at the chip or design level. These variables can be used in both the design_syn.conf and core_syn.conf files as they are more global in nature than those specified by `param`. For example:

```
if { $chipparam("full") == 1 } {
   instance "bigcore" {
    port "in" {
       setuptime {[expr $clockperiod * 0.7]}
    }
   }
}
```

interfaceparam

This array is indexed by the interface name and the configuration parameters that are on an interface. It should only be used for core_syn.conf files since it is only applicable to the interfaces on the instance being processed. In the following example the interface name is ip.

```
if { $interfaceparam("ip_respaccept") == 1 } {
   port "ipMRespAccept_o" {
    c2qtime {[expr $clockperiod * 21/25]}
   }
}
```

9.2.2 Version Section

The version of the core synthesis configuration file is required. Specify the version with the version command, for example: `version 1.3`

9.2.3 Clock Section

If you have non-OCP clocks for an IP block or want to specify the `worstcasedelay` of any clock (including OCP clocks) used in the core, specify the names of the clocks in the core synthesis configuration file. Use the following syntax to specify the name of the clock and its `worstcasedelay`:

```
clock <clockName> {
   worstcasedelay <delay Value>
}
```

`clockName` refers to the name of the port that brings the clock into the core for the core synthesis configuration file. For example:

```
clock "myClock"
```

worstcasedelay

The worst case delay value is the longest path through the core or instance for a particular clock. The value is used to check that the core can meet the timing requirements of the current design. To help make this value more portable, you may want to use the technology variable `gatedelay`. For example:

```
clock "myClock" {
 worstcasedelay {[10.5 * $gatedelay]}
}

clock "otherClock" {
 worstcasedelay 5
}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

9.2.4 Area Section

The area is the size in gates of the core or instance. By specifying the size in gates the area can be calculated based on the size of a typical two input nand gate in a particular synthesis library. For example:

```
area {[expr 20500 / $gatesize]}
area 5000
```

Constant values are specified in two input nand gate equivalents. For consistency, use expression that can be interpreted in gates.

9.2.5 Port Constraints Section

Use the port constraints section to specify the timing parameters. Input port information that can be specified includes the setup time, related clock (non-OCP ports), and driving cell. For output ports, the clock to output times, related clock (non-OCP ports), and the loading information must be supplied.

9.2.5.1 Port Constraint Keywords

The keywords that can be used to specify information about port constraints are:

c2qtime

The c2q (clock to q or clock to output) time is the longest path using worst case timing from a starting point in the core (register or input port) to the output port. This includes the `c2qtime` of the register. To maintain portability, most cores specify this as a percentage of the fastest clock period used while synthesizing the core. For example:

```
c2qtime {[expr $timescale * 3500]}
c2qtime {[expr $clockperiod * 0.25]}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

c2qtimemin

The c2q (clock to q or clock to output) time min is the shortest path using best case timing from a starting point in the core (register or input port) to the output port. This includes the `c2qtime` of the register. Most cores use the default from the technology section, `defaultc2qtimemin`. For example:

```
c2qtimemin {[expr $timescale * 100]}
c2qtimemin {$defaultc2qtimemin}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

clockname

This is an optional field for all OCP ports and is a string specifying the associated clock portname. For input ports, input delays use this clock as the reference clock. For output ports, output delays use this clock as the reference clock. For example:

```
clockname "myClock"
```

drivingcellpin

This variable describes which cell in the synthesis library is expected to be driving the input. To maintain portability set this variable to use one of the technology values of `high/medium/lowdrivegatepin`.

Values are a string that specifies the logical name of the synthesis library, the cell from the library, and the pin that will be driving an input for the core. The pin is optional. For example:

```
drivingcellpin {$mediumdrivegatepin}
drivingcellpin "pt25u/nand2/O"
```

holdtime

The hold time is the shortest path using best case timing from an input port to any endpoint in the core. Most cores use the default from the technology section, `defaultholdtime`. For example:

```
holdtime {[expr $timescale * 100]}
holdtime {$defaultholdtime}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

loadcellpin

The name of the load library/cell/pin that this output port is expected to drive. The value is specified to the synthesis tool as the gate to use (along with the number of loads) in its load calculations for output ports of a module. For portability use the default.

Values are a string that specifies the logical name of the synthesis library, the cell from the library, and the pin that the load calculation is derived from. The pin is optional. For example:

```
loadcellpin "pt25u/nand2/I1"
loadcellpin {$defaultloadcellpin}
```

loads

The number of loadcellpins that this output port is expected to drive. The value is communicated to the synthesis tool as the number of loads to use in load calculations for output ports of a module. The typical setting for this is the technology value of `defaultloads`. Values are an expression that evaluates to an integer. For example:

```
loads 5
loads {$defaultloads}
```

maxfanout

This keyword limits the fanout of an input port to a specified number of fanouts. To maintain portability set this variable in terms of the technology variable `defaultfanoutload`.Constant values are specified in library units. For example:

```
maxfanout {[expr $defaultfanoutload * 1]}
```

setuptime

The longest path using worst case timing from an input port to any endpoint in the core. To maintain portability, most cores specify this as a percentage of the fastest clock period used during synthesis of the core. For example:

```
setuptime {[expr $timescale * 2500]}
setuptime {[expr $clockperiod * 0.25]}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

wireloaddelay

Replaces capacitance/resistance as a way to specify expected delays caused by the interconnect. To maintain portability set this variable to use a technology value of `long/medium/shortnetdelay`.

The resulting values get added to the worst case clock-to-output times of the ports to anticipate net delays of connections to these ports. To improve the accuracy of the delay calculation cores should use the resistance and capacitance settings.

You cannot specify both `wireloaddelay` and `wireloadresistance/capacitance` for the same port. For example:

```
wireloaddelay {[expr $clockperiod * .25]}
wireloaddelay {$mediumnetdelay}
```

Constant values are specified in nanoseconds. For consistency, use expressions that can be interpreted in nanoseconds.

wireloadresistance
wireloadcapacitance

Specify expected loading and resistance caused by the interconnect. If available, specify both resistance and capacitance. To maintain portability set this variable to use one of the technology values of `long/medium/shortnetrcresistance/capacitance`.

If these constraints are specified they show up as additional loads and resistances on output ports of a module. You cannot use both wireloaddelay and `wireloadresistance/capacitance` for the same port.

Specify constant values as expressions that result in kOhms for resistance and picofarads (pf) for capacitance. For example:

```
wireloadresistance {[expr $resistancescale * .09]}
wireloadcapacitance {[expr $capacitancescale * .12]}
wireloadresistance {$mediumnetrcresistance}
wireloadcapacitance {$mediumnetrccapacitance}
```

9.2.5.2 Input Port Syntax

For input and inout ports (*inout* ports have both an input and an output definition) use the following syntax:

```
port <portName> {
   clockname <clockName>
   drivingcellpin <drivingCellName>
   setuptime <Value>
   holdtime <Value>
   maxfanout <Value>
}
```

Examples

In the following example, the clock is not specified since this is an OCP port and is known to be controlled by the OCP clock. If a clock were specified as something other than the OCP clock, an error would result.

```
port “MCmd_i” {
   drivingcellpin {$mediumdrivegatepin}
   setuptime {[expr $clockperiod * 0.2]}
}
```

In the following example, the setup time is required to be 2ns. Time constants are assumed to be in nanoseconds. Use the `timescale` variable to convert library units to nanoseconds.

```
port “MAddr_i” {
   drivingcellpin {$mediumdrivegatepin}
   setuptime 2
}
```

The following example shows how to associate a non OCP clock to a port. The example uses `maxfanout` to limit the fanout of myInPort to 1. If the logic for myInPort required it to fanout to more than one connection, the synthesis tool would add a buffer to satisfy the `maxfanout` requirement.

```
port “myInPort” {
   clockname “myClock”
   drivingcellpin {$mediumdrivegatepin}
   setuptime 2
   maxfanout {[expr $defaultfanoutload * 1]}
}
```

9.2.5.3 Output Port Syntax

For output and inout ports (inout ports have both an input and an output definition) use the following syntax:

```
port <portName> {
   clockname <clockName>
   loadcellpin <loadCellPinName>
   loads <Value>
   wireloadresistance <Value>
   wireloadcapacitance <Value>
   wireloaddelay <Value>
   c2qtime <Value>
   c2qtimemin <Value>
}
```

You cannot specify both `wireloaddelay` and `wireloadresistance/capacitance` for the same port.

Examples

In the following example, the clock is not specified since this is an OCP port and is known to be controlled by the OCP clock.

```
port "SCmdaccept_o"
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloadresistance {$mediumnetrcresistance}
   wireloadcapacitance {$mediumnetrccapacitance}
   c2qtime {[expr $clockperiod * 0.2]}
}
```

In the following example, the clock to output time is required to be 2 ns. Time constants are assumed to be in nanoseconds. Use the `timescale` variable to convert library units to nanoseconds.

```
port "SResp_o"
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloadresistance {$mediumnetrcresistance}
   wireloadcapacitance {$mediumnetrccapacitance}
   c2qtime 2
}
```

The following example shows how to associate a clock to an output port.

```
port "myOutPort"
   clockname "myClock"
   loadcellpin {$defaultloadcellpin}
   loads 10
   wireloaddelay {$longnetdelay}
   c2qtime {[expr $clockperiod * .2]}
}
```

InOut Port Example

```
port "Signal_io"
   drivingcellpin {$mediumdrivegatepin}
   setuptime {[expr $clockperiod * 0.2]}
}
port "Signal_io"
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloadresistance {$mediumnetrcresistance}
   wireloadresistance {$mediumnetrccapacitance}
   c2qtime {[expr $clockperiod * 0.2]}
}
```

9.2.6 Max Delay Constraints

Using the max delay constraints you can specify the delay between two ports on a combinational path. This is useful when synthesizing two communicating OCP interfaces. The syntax for `maxdelay` is:

```
maxdelay {
   delay <delayValue> fromport <portName> toport <portName>
   .
   .
   .
}
```

where: `<delayValue>` can be a constant or a Tcl expression.

In the following example, a `maxdelay` of 3 ns is specified for the combinational path between myInPort1 and myOutPort1. A `maxdelay` of 50% of the clockperiod is specified for the path between myInPort2 and myOutPort2. The braces around the expression delay evaluation until the expression is used by the mapping program.

```
maxdelay {
   delay 3 fromport "myInPort1" toport "myOutPort1
   delay {[expr $clockperiod *.5]} fromport "myInPort2" toport "myOutPort2"
}
```

9.2.7 False Path Constraints

Using the false path constraints you can specify that a path between certain input and output ports is logically impossible.

The syntax for `falsepath` is:

```
falsepath{
   fromport <portName> toport <portName>
.
.
.
}
```

In the following example, a `falsepath` is set up between myInPort1 and myOutPort1 as well as myInPort2 and myOutPort2. This tells the synthesis tool that the path is not logically possible and so it will not try to optimize this path to meet timing.

```
falsepath {
   fromport "myInPort1" toport "myOutPort1"
   fromport "myInPort2" toport "myOutPort2"
}
```

9.2.8 Sample Core Synthesis Configuration File

The following example shows a complete core synthesis configuration file.

```
version 1.3
port “Reset_ni” {
   drivingcellpin {$mediumgatedrivepin}
   setuptime {[expr $clockperiod * .5]}
}
port “MCmd_i” {
   drivingcellpin {$mediumgatedrivepin}
   setuptime {[expr $clockperiod * .9]}
}
port “MAddr_i” {
   drivingcellpin {$mediumgatedrivepin}
   setuptime {[expr $clockperiod * .5]}
}
port “MWidth_i” {
   drivingcellpin {$mediumgatedrivepin}
   setuptime {[expr $clockperiod * .5]}
}
port “MData_i” {
   drivingcellpin {$mediumgatedrivepin}
   setuptime {[expr $clockperiod * .5]}
}
port “SCmdAccept_o” {
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloaddelay {$mediumnetdelay}
   c2qtime {[expr $clockperiod * .9]}
}
port “SResp_o” {
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloaddelay {$mediumnetdelay}
   c2qtime {[expr $clockperiod * .8]}
}
port “SData_o” {
   loadcellpin {$defaultloadcellpin}
   loads {$defaultloads}
   wireloaddelay {$mediumnetdelay}
   c2qtime {[expr $clockperiod * .8]}
}
maxdelay {
   delay 2 fromport “MData_i” toport
   “SResp_o”
}
falsepath {
   fromport “MData_i” toport “SData_o”
}
```

Part II *Guidelines*

10 Timing Diagrams

The timing diagrams within this chapter look at signals at strategic points and are not intended to provide full explanations but rather, highlight specific areas of interest. The diagrams are provided solely as examples. For related information about phases, see Section 4.3 on page 40.

Most of the timing diagrams in this chapter are based upon simple OCP clocking, where the OCP clock is completely determined by the Clk signal. A few diagrams are repeated to show the impact of the EnableClk signal. Most fields are unspecified whenever their corresponding phase is not asserted. This is indicated by the striped pattern in the waveforms. For example, when MCmd is IDLE the request phase is not asserted, so the values of MAddr, MData, and SCmdAccept are unspecified.

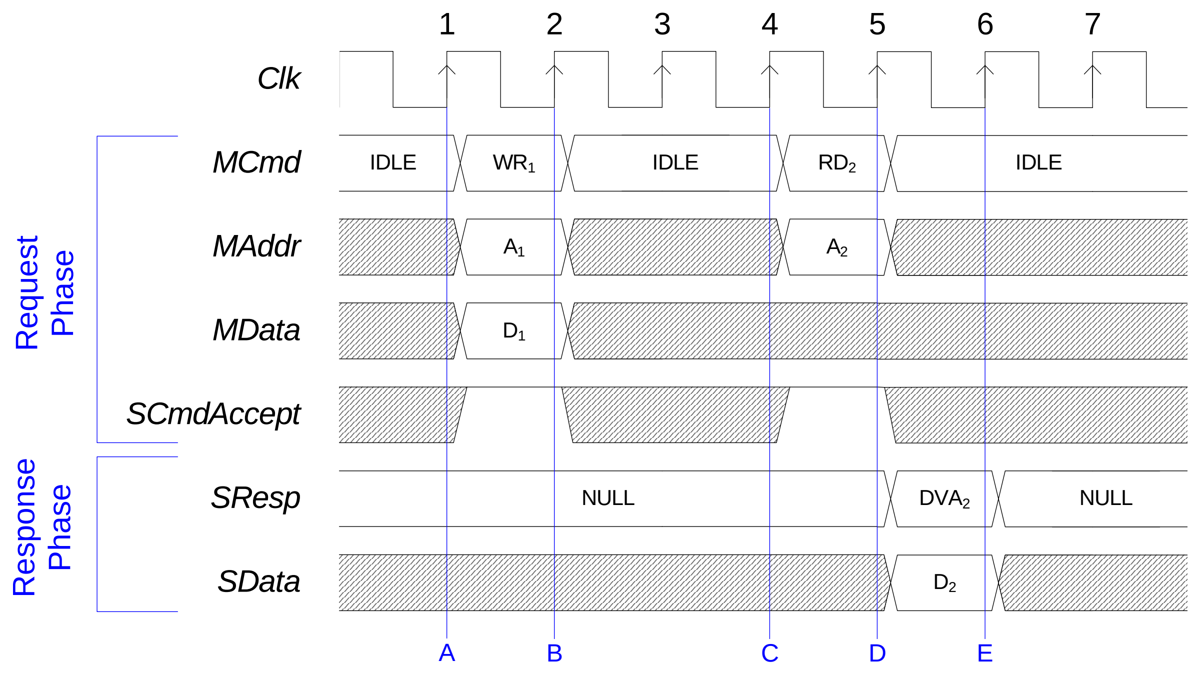
Subscripts on labels in the timing diagrams denote transfer numbers that can be helpful in tracking a transfer across protocol phases.

For a description of timing diagram mnemonics, see Tables 2 on page 15 and 3 on page 16.

10.1 Simple Write and Read Transfer

Figure 21 illustrates a simple write and a read transfer on a basic OCP interface. This diagram shows a write with no response enabled on the write, which is typical behavior for a synchronous SRAM or a register bank.

Figure 21 Simple Write and Read Transfer



Sequence

A. The master starts a request phase on clock 1 by switching the MCmd field from IDLE to WR. At the same time, it presents a valid address (A1) on MAddr and valid data (D1) on MData. The slave asserts SCmdAccept in the same cycle, making this a 0-latency transfer.

B. The slave captures the values from MAddr and MData and uses them internally to perform the write. Since SCmdAccept is asserted, the request phase ends.

C. The master starts a read request by driving RD on MCmd. At the same time, it presents a valid address on MAddr. The slave asserts SCmdAccept in the same cycle for a request accept latency of 0.

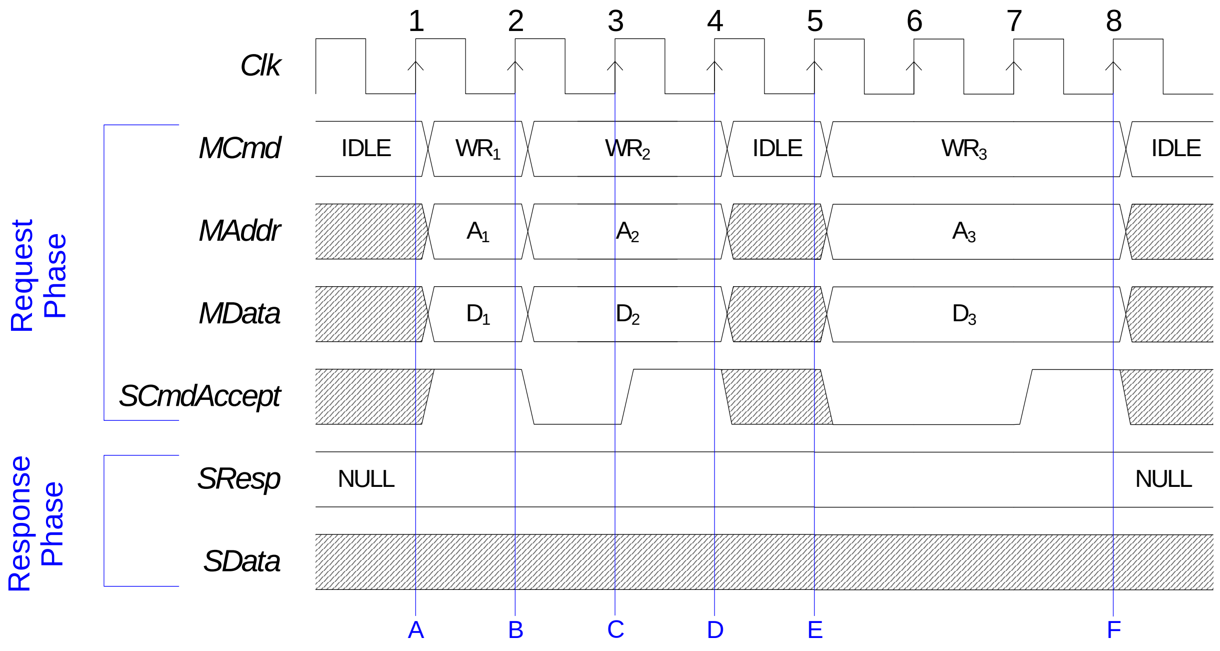
D. The slave captures the value from MAddr and uses it internally to determine what data to present. The slave starts the response phase by switching SResp from NULL to DVA. The slave also drives the selected data on SData. Since SCmdAccept is asserted, the request phase ends.

E. The master recognizes that SResp indicates data valid and captures the read data from SData, completing the response phase. This transfer has a request-to-response latency of 1.

10.2 Request Handshake

Figure 22 illustrates the basic flow-control mechanism for the request phase using SCmdAccept. There are three writes with no responses enabled, each with a different request accept latency.

Figure 22 Request Handshake



Sequence

A. The master starts a write request by driving WR on MCmd and valid address and data on MAddr and MData, respectively. The slave asserts SCmdAccept in the same cycle, for a request accept latency of 0.

B. The master starts a new transfer in the next cycle. The slave captures the write address and data. It deasserts SCmdAccept, indicating that it is not yet ready for a new request.

C. Recognizing that SCmdAccept is not asserted, the master holds all request phase signals (MCmd, MAddr, and MData). The slave asserts SCmdAccept in the next cycle, for a request accept latency of 1.

D. The slave captures the write address and data.

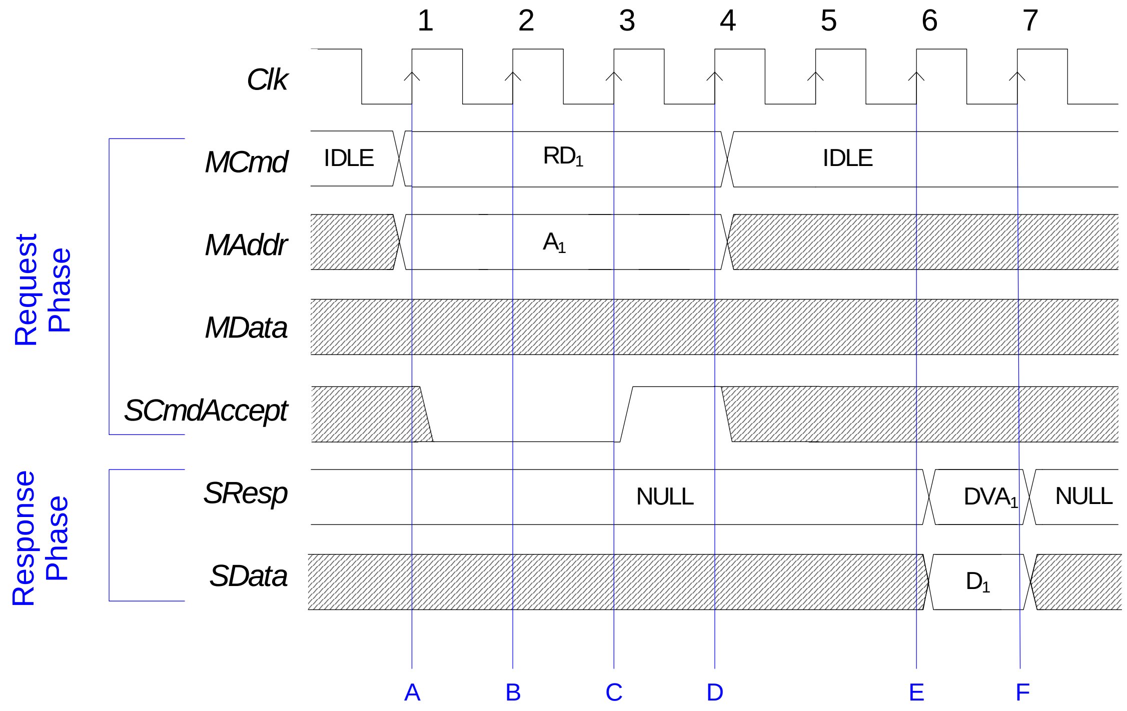
E. After 1 idle cycle, the master starts a new write request. The slave deasserts SCmdAccept.

F. Since SCmdAccept is asserted, the request phase ends. SCmdAccept was low for 2 cycles, so the request accept latency for this transfer is 2. The slave captures the write address and data.

10.3 Request Handshake and Separate Response

Figure 23 illustrates a single read transfer in which a slave introduces delays in the request and response phases. The request accept latency 2, corresponds to the number of clock cycles that SCmdAccept was deasserted. The request to response latency 3, corresponds to the number of clock cycles from the end of the request phase (D) to the end of the response phase (F).

Figure 23 Request Handshake and Separate Response



Sequence

A. The master starts a request phase by issuing the RD command on the MCmd field. At the same time, it presents a valid address on MAddr. The slave is not ready to accept the command yet, so it deasserts SCmdAccept.

B. The master sees that SCmdAccept is not asserted, so it keeps all request phase signals steady. The slave may be using this information for a long decode operation, and it expects the master to hold everything steady until it asserts SCmdAccept.

C. The slave asserts SCmdAccept. The master continues to hold the request phase signals.

D. Since SCmdAccept is asserted, the request phase ends. The slave captures the address, and although the request phase is complete, it is not ready to provide the response, so it continues to drive NULL on the SResp field. For example, the slave may be waiting for data to come back from an off-chip memory device.

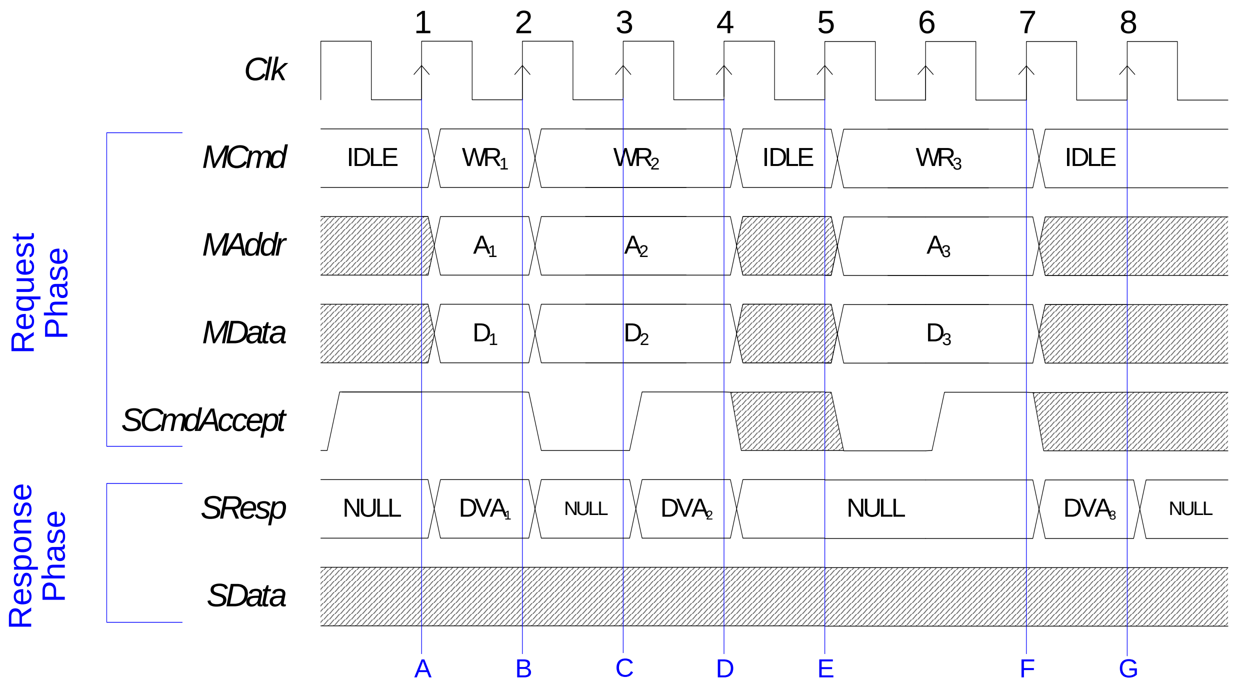
E. The slave is ready to present the response, so it issues DVA on the SResp field, and drives the read data on SData.

F. The master sees the DVA response, captures the read data, and the response phase ends.

10.4 Write with Response

Figure 24 is the same example as the waveform on page 161 but with response on write enabled. The response is typically provided to the master in the same cycle as SCmdAccept, but could be delayed (if required to perform an error check for instance). On the first write transaction, the slave uses a default accept scheme, resulting in a 0-wait state write transaction. Using fully-synchronous handshake, this condition is only possible when the slave's ability to accept a command depends solely on its internal state: any command issued by the master can be accepted. Same-cycle SCmdAccept could also be achieved using combinational logic.

Figure 24 Write with Response



Sequence

A. The master starts a write request by driving WR on MCmd and valid address and data on MAddr and MData, respectively. The slave having already asserted SCmdAccept for a request accept latency of 0, drives DVA on SResp to indicate a successful transaction.

B. The master starts a new transfer in the next cycle. The slave captures the write address and data and deasserts SCmdAccept, indicating that it is not ready for a new request.

C. With SCmdAccept not asserted, the master holds all request phase signals (MCmd, MAddr, and MData). The slave asserts SCmdAccept in the next cycle, for a request accept latency of 1 and drives DVA on SResp to indicate a successful transaction.

D. The slave captures the write address and data.

E. After 1 idle cycle, the master starts a new write request. The slave deasserts SCmdAccept.

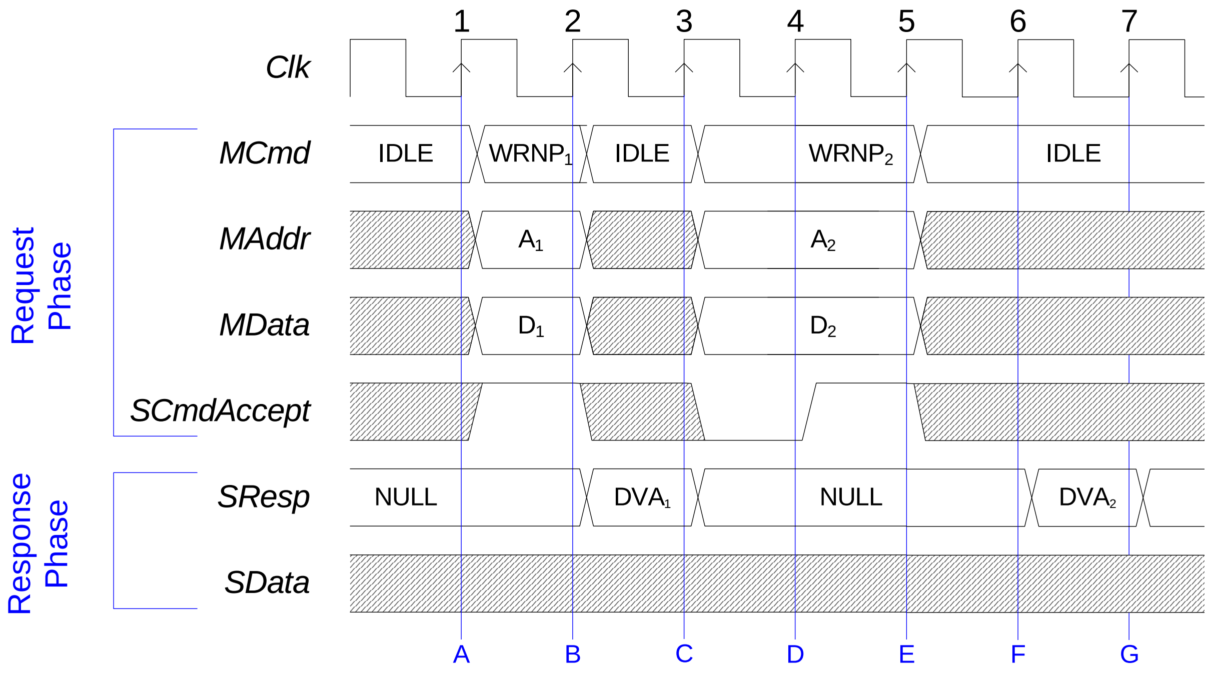
F. Since SCmdAccept is asserted, the request phase ends. SCmdAccept was low for 2 cycles, so the request accept latency for this transfer is 2. The slave captures the write address and data. The slave drives DVA on SResp to indicate a successful transaction.

G. The master samples the response.

10.5 Non-Posted Write

Figure 25 repeats the previous example for a non-posted write transaction. In this case the response must be returned to the master once the write operation commits. There is no difference in the command acceptance, but the response may be significantly delayed. If this scheme is used for all posting-sensitive transactions, the result is decreased data throughput but higher system reliability.

Figure 25 Non-posted Write



Sequence

A. The master starts a non-posted write request by driving WRNP on MCmd and valid address and data on MAddr and MData, respectively. The slave asserts SCmdAccept combinationally, for a request accept latency of 0.

B. The slave drives DVA on SResp to indicate a successful first transaction.

C. The master starts a new transfer. The slave deasserts the SCmdAccept, indicating it is not yet ready to accept a new request. The master samples DVA on SResp and the first response phase ends.

D. The slave asserts SCmdAccept for a request accept latency of 1.

E. The slave captures the write address and data.

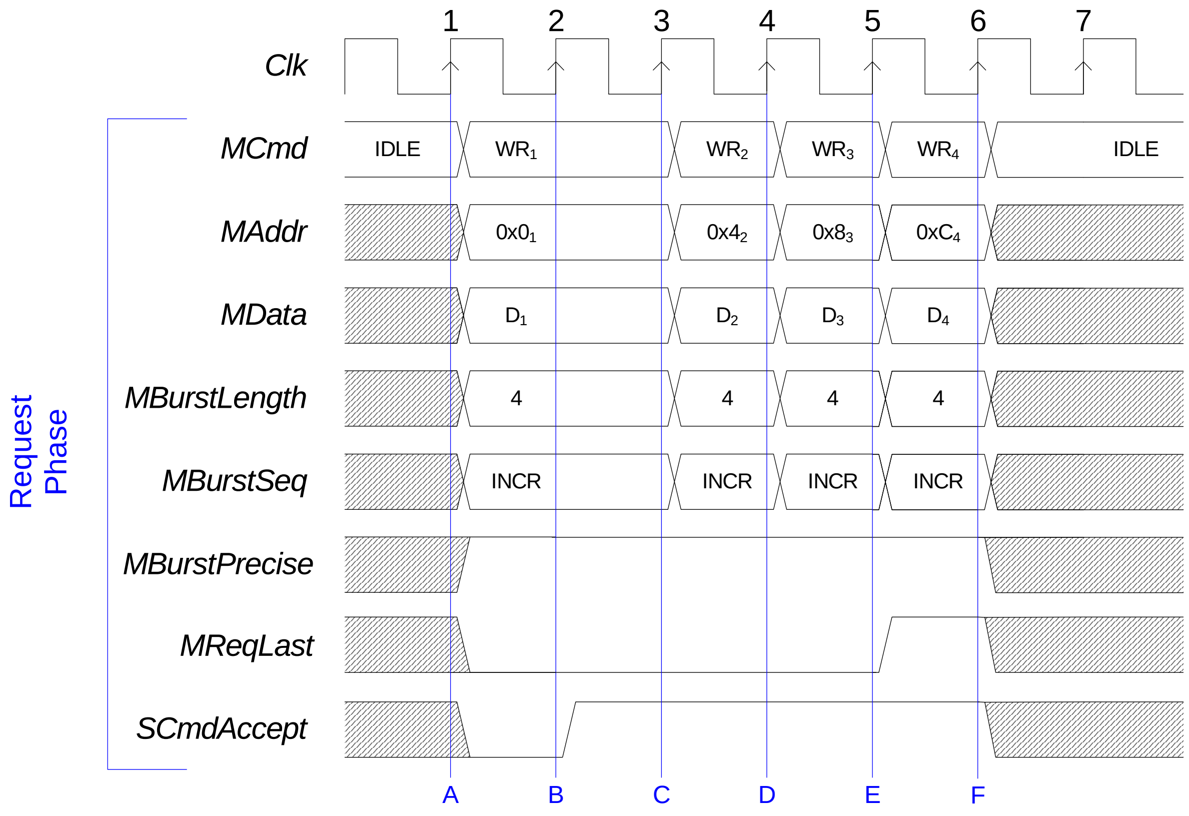
F. The slave drives DVA on SResp to indicate a successful second transaction.

G. The master samples DVA on SResp and the second response phase ends.

10.6 Burst Write

Figure 26 illustrates a burst of four 32-bit words, incrementing precise burst write, with optional burst framing information (MReqLast). As the burst is precise (with no response on write), the MBurstLength signal is constant during the whole burst. MReqLast flags the last request of the burst, and SRespLast flags the last response of the burst. The slave may either count requests or monitor MReqLast for the end of burst.

Figure 26 *Burst Write*



Sequence

A. The master starts the burst write by driving WR on MCmd, the first address of the burst on MAddr, valid data on MData, a burst length of four on MBurstLength, the burst code INCR on MBurstSeq, and asserts MBurstPrecise. MReqLast must be deasserted until the last request in the burst. The burst signals indicate that this is an incrementing burst of precisely four transfers. The slave is not ready for anything, so it deasserts SCmdAccept.

B. The slave asserts SCmdAccept for a request accept latency of 1.

C. The master issues the next write in the burst. MAddr is set to the next word-aligned address. For 32-bit words, the address is incremented by 4. The slave captures the data and address of the first request.

D. The master issues the next write in the burst, incrementing MAddr. The slave captures the data and address of the second request.

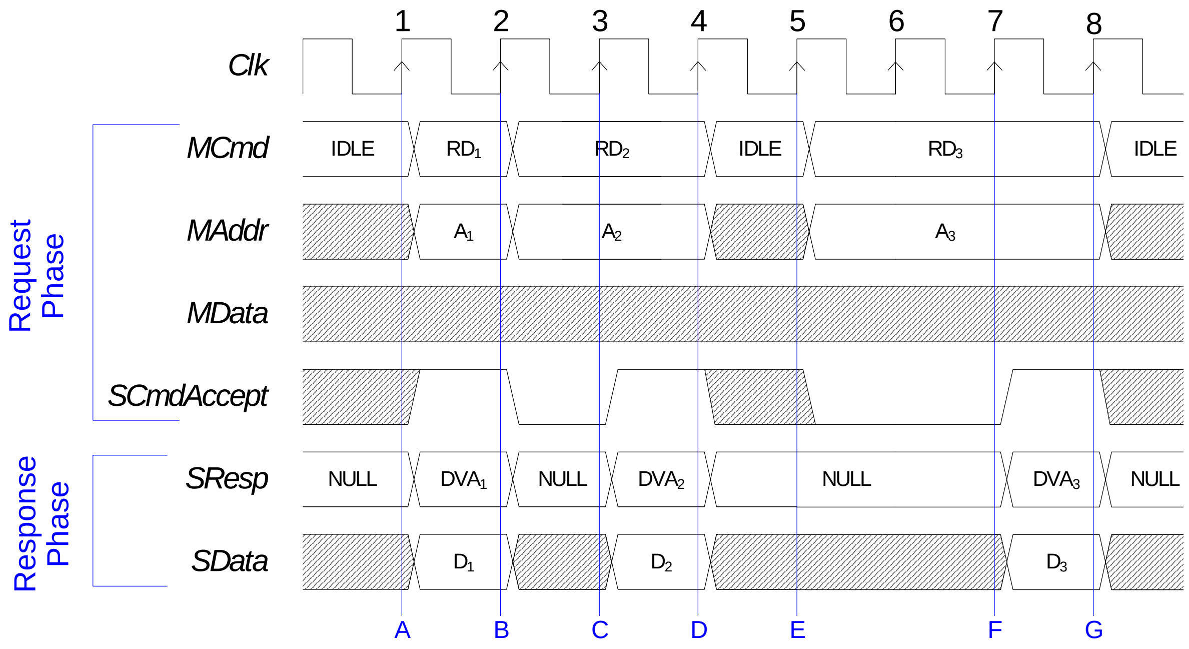
E. The master issues the final write in the burst, incrementing MAddr, and asserting MBurstLast. The slave captures the data and address of the third request.

F. The slave captures the data and address of the last request.

10.7 Non-Pipelined Read

Figure 27 shows three read transfers to a slave that cannot pipeline responses after requests. This is the typical behavior of legacy computer bus protocols with a single WAIT or ACK signal. In each transfer, SCmdAccept is asserted in the same cycle that SResp is DVA. Therefore, the request-to-response latency is always 0, but the request accept latency varies from 0 to 2.

Figure 27 Non-Pipelined Read



Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The slave asserts SCmdAccept, for a request accept latency of 0. When the slave sees the read command, it responds with DVA on SResp and valid data on SData. (This requires a combinational path in the slave from MCmd, and possibly other request phase fields, to SResp, and possibly other response phase fields.)

B. The master launches another read request. It also sees that SResp is DVA and captures the read data from SData. The slave is not ready to respond to the new request, so it deasserts SCmdAccept.

C. The master sees that SCmdAccept is low and extends the request phase. The slave is now ready to respond in the next cycle, so it simultaneously asserts SCmdAccept and drives DVA on SResp and the selected data on SData. The request accept latency is 1.

D. Since SCmdAccept is asserted, the request phase ends. The master sees that SResp is now DVA and captures the data.

E. The master launches a third read request. The slave deasserts SCmdAccept.

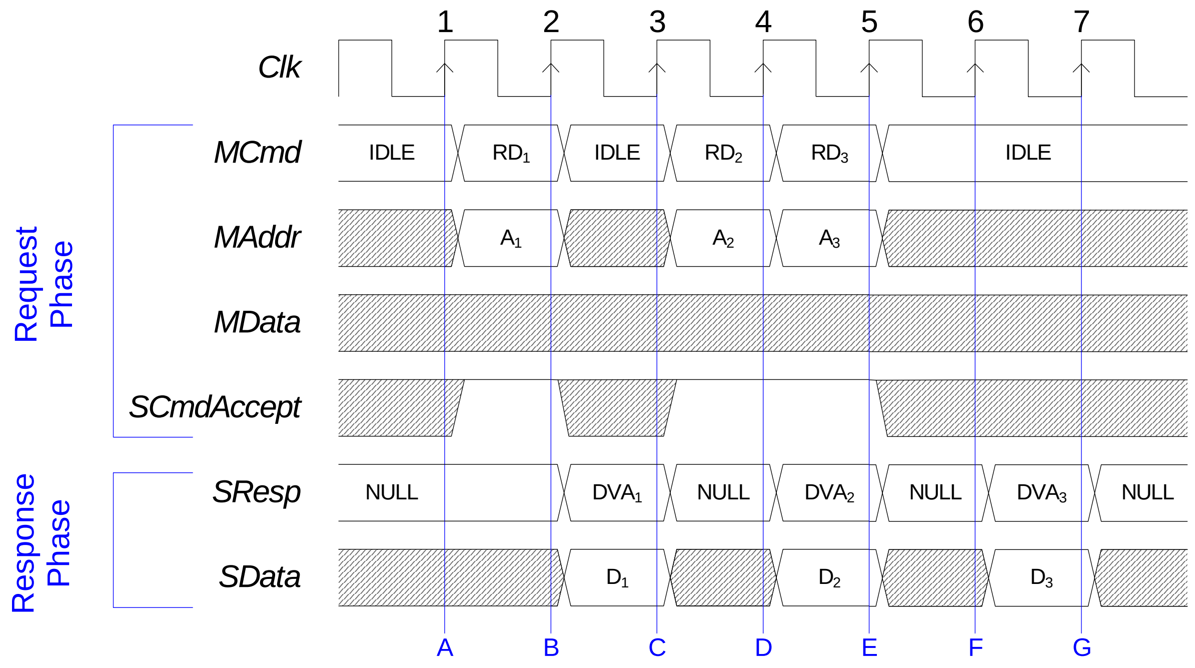
F. The slave asserts SCmdAccept after 2 cycles, so the request accept latency is 2. It also drives DVA on SResp and the read data on SData.

G. The master sees that SCmdAccept is asserted, ending the request phase. It also sees that SResp is now DVA and captures the data.

10.8 Pipelined Request and Response

Figure 28 shows three read transfers using pipelined request and response semantics. In each case, the request is accepted immediately, while the response is returned in the same or a later cycle.

Figure 28 Pipelined Request and Response



Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The slave asserts SCmdAccept, for a request accept latency of 0.

B. Since SCmdAccept is asserted, the request phase ends. The slave responds to the first request with DVA on SResp and valid data on SData.

C. The master launches a read request and the slave asserts SCmdAccept. The master sees that SResp is DVA and captures the read data from SData. The slave drives NULL on SResp, completing the first response phase.

D. The master sees that SCmdAccept is asserted, so it can launch a third read even though the response to the previous read has not been received. The slave captures the address of the second read and begins driving DVA on SResp and the read data on SData.

E. Since SCmdAccept is asserted, the third request ends. The master sees that the slave has produced a valid response to the second read and captures the data from SData. The request-to-response latency for this transfer is 1.

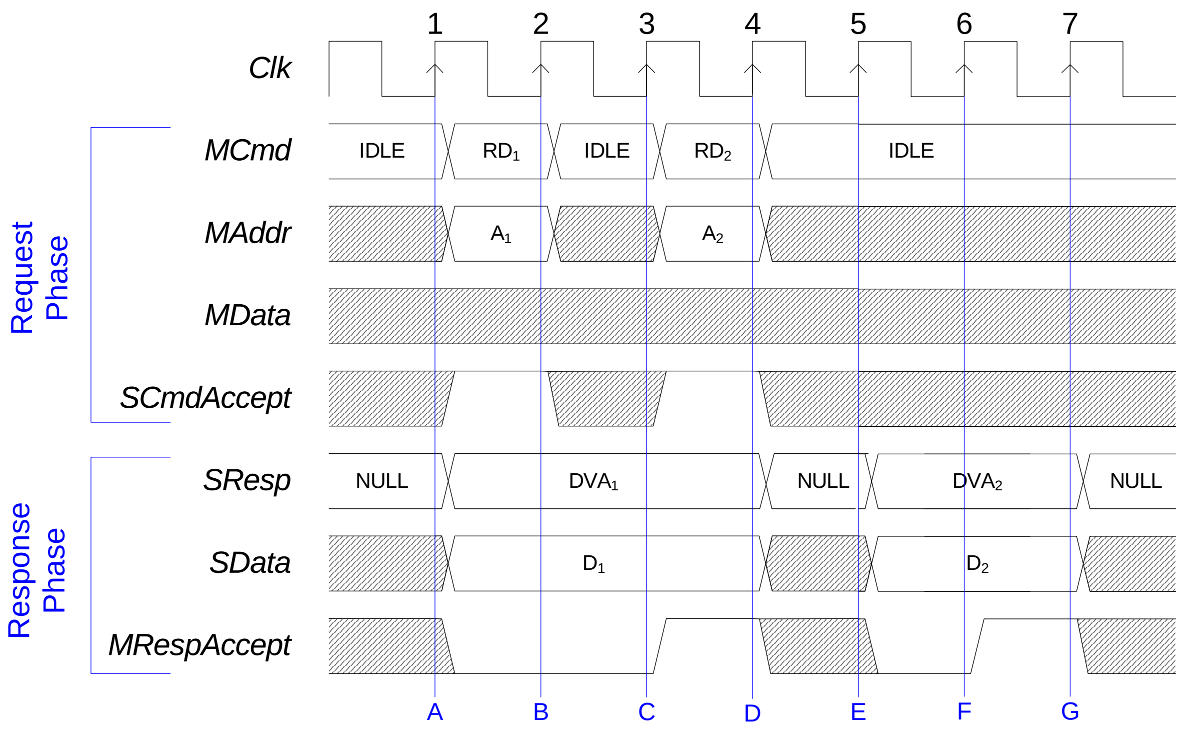
F. The slave has the data for the third read, so it drives DVA on SResp and the data on SData.

G. The master captures the data for the third read from SData. The request-to-response latency for this transfer is 2.

10.9 Response Accept

Figure 29 shows examples of the response accept extension used with two read transfers. An additional field, MRespAccept, is added to the response phase. This signal may be used by the master to flow-control the response phase.

Figure 29 Response Accept



Sequence

A. The master starts a read request by driving RD on MCmd and a valid address on MAddr. The slave asserts SCmdAccept immediately, and it drives DVA on SResp and the read data on SData as soon as it sees the read request. The master is not ready to receive the response for the request it just issued, so it deasserts MRespAccept.

B. Since SCmdAccept is asserted, the request phase ends. The master continues to deassert MRespAccept, however. The slave holds SResp and SData steady.

C. The master starts a second read request and is ready for the response from its first request, so it asserts MRespAccept. This corresponds to a response accept latency of 2.

D. Since SCmdAccept is asserted, the request phase ends. The master captures the data for the first read from the slave. Since MRespAccept is asserted, the response phase ends. The slave is not ready to respond to the second read, so it drives NULL on SResp.

E. The slave responds to the second read by driving DVA on SResp and the read data on SData. The master is not ready for the response, however, so it deasserts RespAccept.

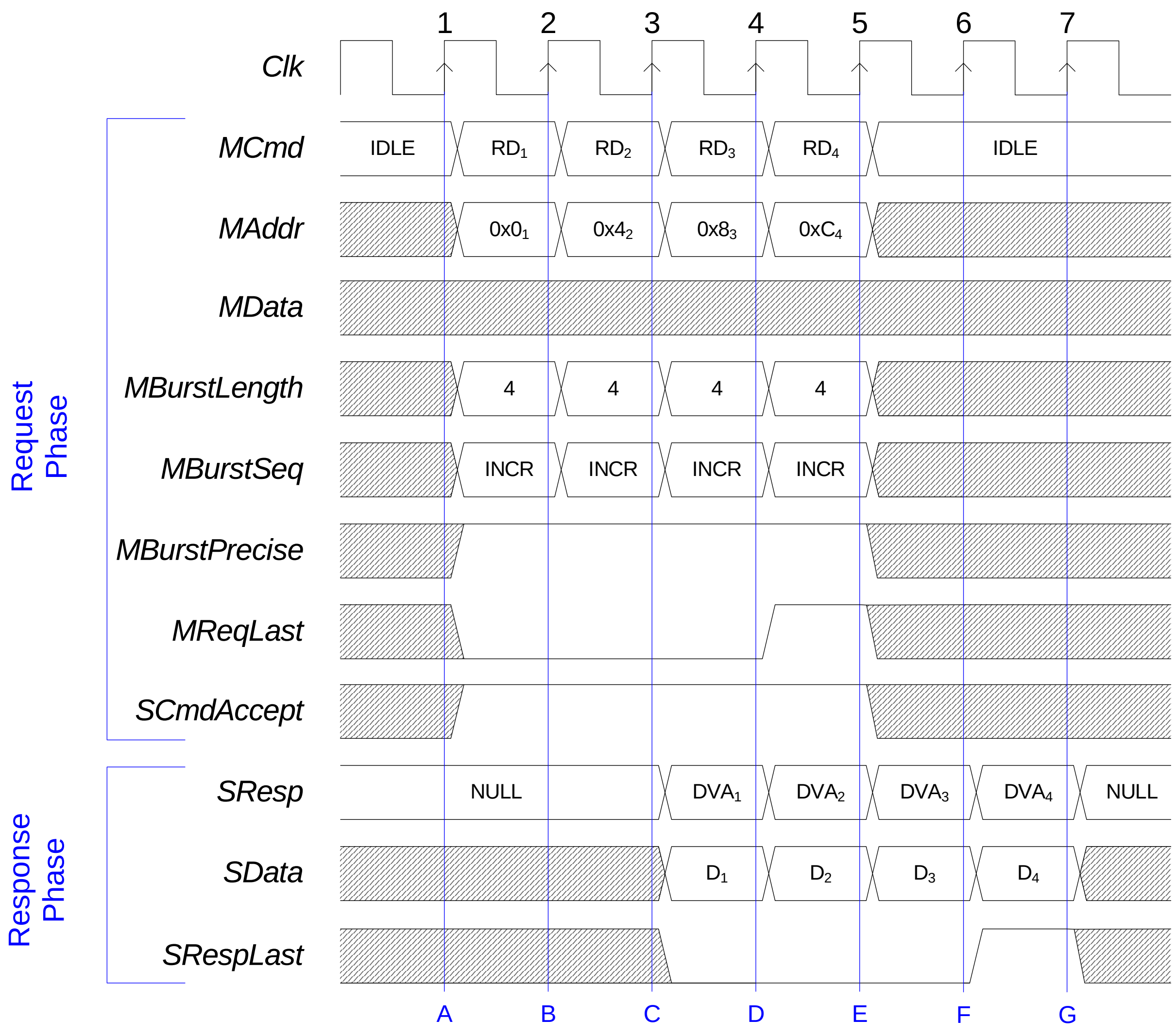
F. The master asserts MRespAccept, for a response accept latency of 1.

G. The master captures the data for the second read from the slave. Since MRespAccept is asserted, the response phase ends.

10.10 Incrementing Precise Burst Read

Figure 30 illustrates a burst of four 32-bit words, incrementing precise burst read, with burst framing information (MReqLast/SRespLast). Since the burst is precise, the MBurstLength signal is constant during the whole burst. MReqLast flags the last request of the burst, and SRespLast flags the last response of the burst.

Figure 30 Incrementing Precise Burst Read



Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, four on MBurstLength, INCR on MBurstSeq, and asserts MBurstPrecise. MBurstLength, MBurstSeq and MBurstPrecise must be kept constant during the burst. MReqLast must be deasserted until the last request in the burst. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The slave captures the address of the first request and keeps SCmdAccept asserted.

C. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The slave captures the address of the second request and keeps SCmdAccept asserted. The slave responds to the first read by driving DVA on SResp and the read data on SData.

D. The master issues the last request of the burst, incrementing MAddr and asserting MReqLast. The master also captures the data for the first read from the slave. The slave responds to the second request, and captures the address of the third request.

E. The master captures the data for the second read from the slave. The slave responds to the third request and captures the address of the fourth.

F. The master captures the data for the third read from the slave. The slave responds to the fourth request and asserts SRespLast to indicate the last response of the burst.

G. The master captures the data for the last read from the slave, ending the last response phase.

10.11 Incrementing Imprecise Burst Read

Figure 31 illustrates a burst of four 32-bit words, incrementing imprecise burst read, with burst framing information (MReqLast/SRespLast). MReqLast flags the last request of the burst and SRespLast flags the last response of the burst. The last burst request is signaled primarily by driving the value 1 on MBurstLength.

The burst length sequence (3,3,2,1) is chosen arbitrarily for illustration purposes. The protocol requires that the burst length of the last transfer of the burst be equal to one.

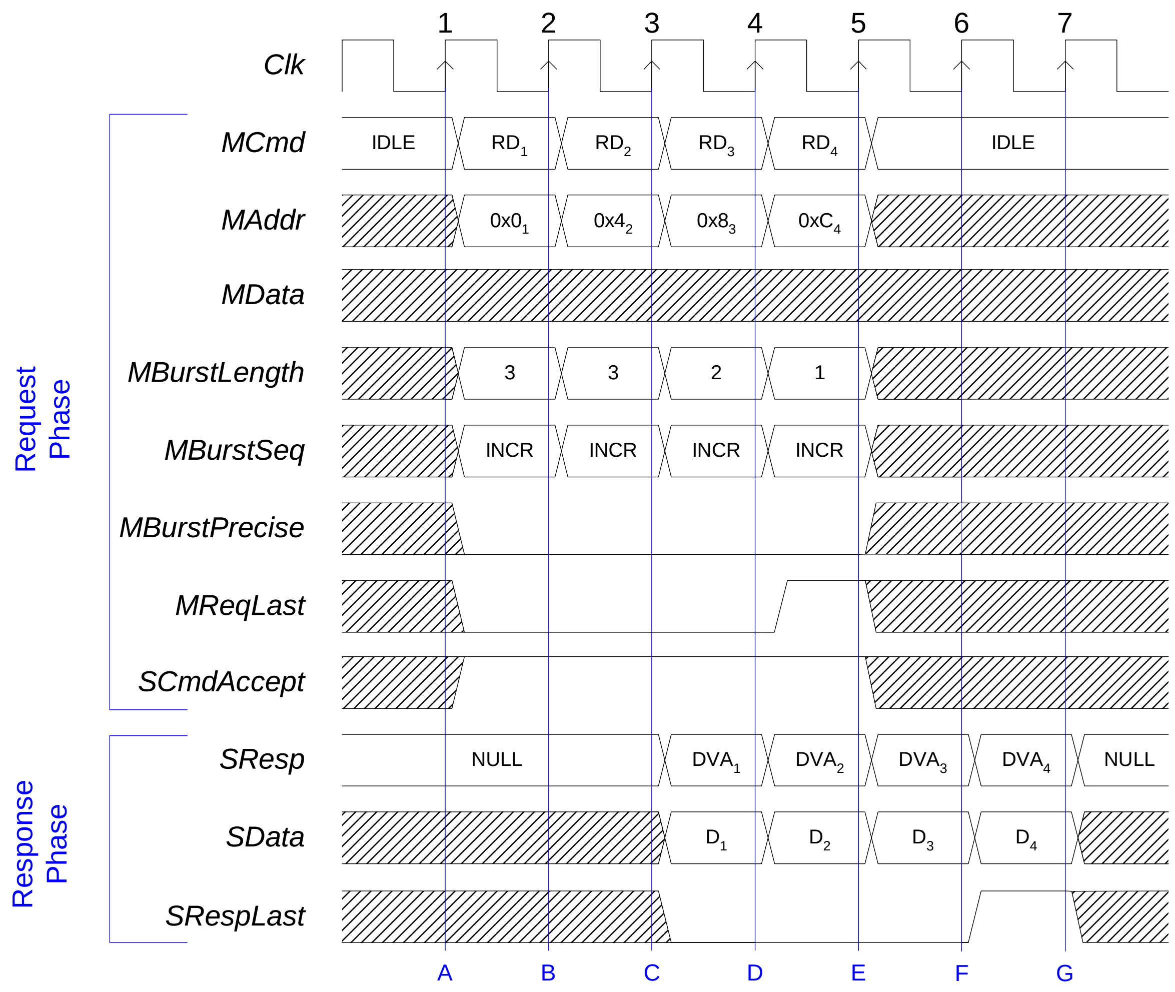
Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, three on MBurstLength, INCR on MBurstSeq, and asserts MBurstPrecise. The burst length is the best guess of the master at this point. MBurstSeq and MBurstPrecise are kept constant during the burst. MReqLast must be deasserted until the last request in the burst. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The MBurstLength is set to three, since the master knows the burst is longer than it originally thought. The slave captures the address of the first request and keeps SCmdAccept asserted.

C. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The MBurstLength is set to two. The slave captures the address of the second request, and keeps SCmdAccept asserted. The slave responds to the first read by driving DVA on SResp and the read data on SData.

Figure 31 Incrementing Imprecise Burst Read



D. The master issues the last request of the burst, incrementing MAddr, setting MBurstLength to one, and asserting MReqLast. The master also captures the data for the first read from the slave. The slave responds to the second request and captures the address of the last request.

E. The master captures the data for the second read from the slave. The slave responds to the third request.

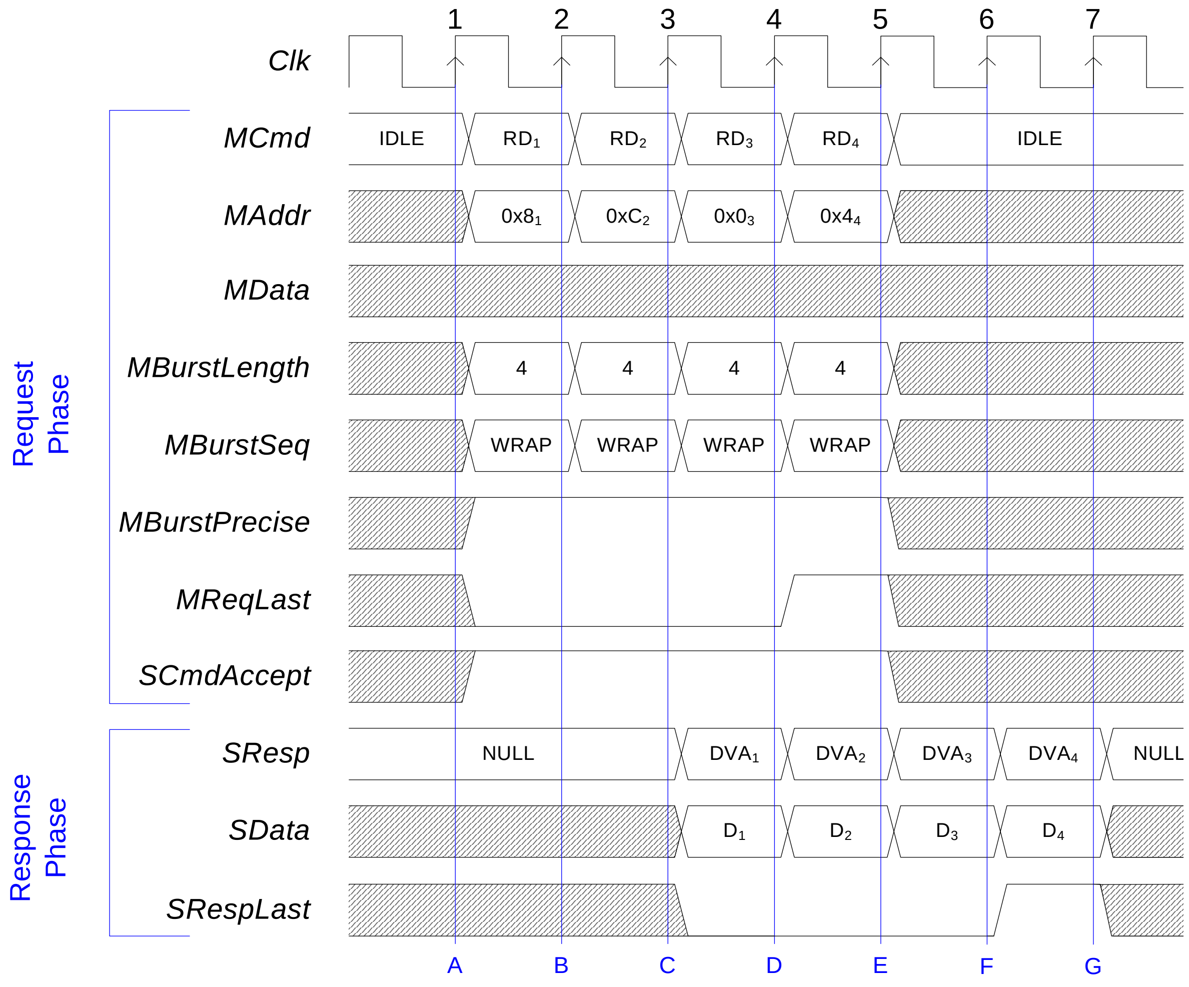
F. The master captures the data for the third read from the slave. The slave responds to the fourth request and asserts SRespLast to indicate the last response of the burst.

G. The master captures the data for the last read from the slave, ending the last response phase.

10.12 Wrapping Burst Read

Figure 32 illustrates a burst of four 32-bit words, wrapping burst read, with optional burst framing information (MReqLast/SRespLast). MReqLast flags the last request of the burst and SRespLast flags the last response of the burst. As a wrapping burst is precise, the MBurstLength signal is constant during the whole burst, and must be power of two. The wrapping burst address must be aligned to boundary MBurstLength times the OCP word size in bytes.

Figure 32 Wrapping Burst Read



Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, four on MBurstLength, WRAP on MBurstSeq, and asserts MBurstPrecise. MBurstLength, MBurstSeq, and MBurstPrecise must be kept constant during the burst. MReqLast must be deasserted until the last request in the burst. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The slave captures the address of the first request, and keeps SCmdAccept asserted.

C. If incremented, the next address would be 0x10. Since the first transfer was from address 0x8 and the burst length is 4, the addresses must be between 0 and 0xF. The master wraps the MAddr to 0, which is the closest alignment boundary. (If the first address were 0x14, the address would wrap to 0x10, rather than 0x20.) The slave captures the address of the second request, and keeps SCmdAccept asserted. The slave responds to the first read by driving DVA on SResp and valid data on SData.

D. The master issues the last request of the burst, incrementing MAddr and asserting MReqLast. The master also captures the data for the first read from the slave. The slave responds to the second request and captures the address of the third.

E. The master captures the data for the second read from the slave. The slave responds to the third request and captures the address of the fourth.

F. The master captures the data for the third read from the slave. The slave responds to the fourth request and asserts SRespLast to indicate the last response of the burst.

G. The master captures the data for the last read from the slave, ending the last response phase.

10.13 Incrementing Burst Read with IDLE Request Cycle

Figure 33 illustrates a burst of four 32-bit words, incrementing precise burst read, with an IDLE cycle inserted in the middle. The master may insert IDLE requests in any burst type.

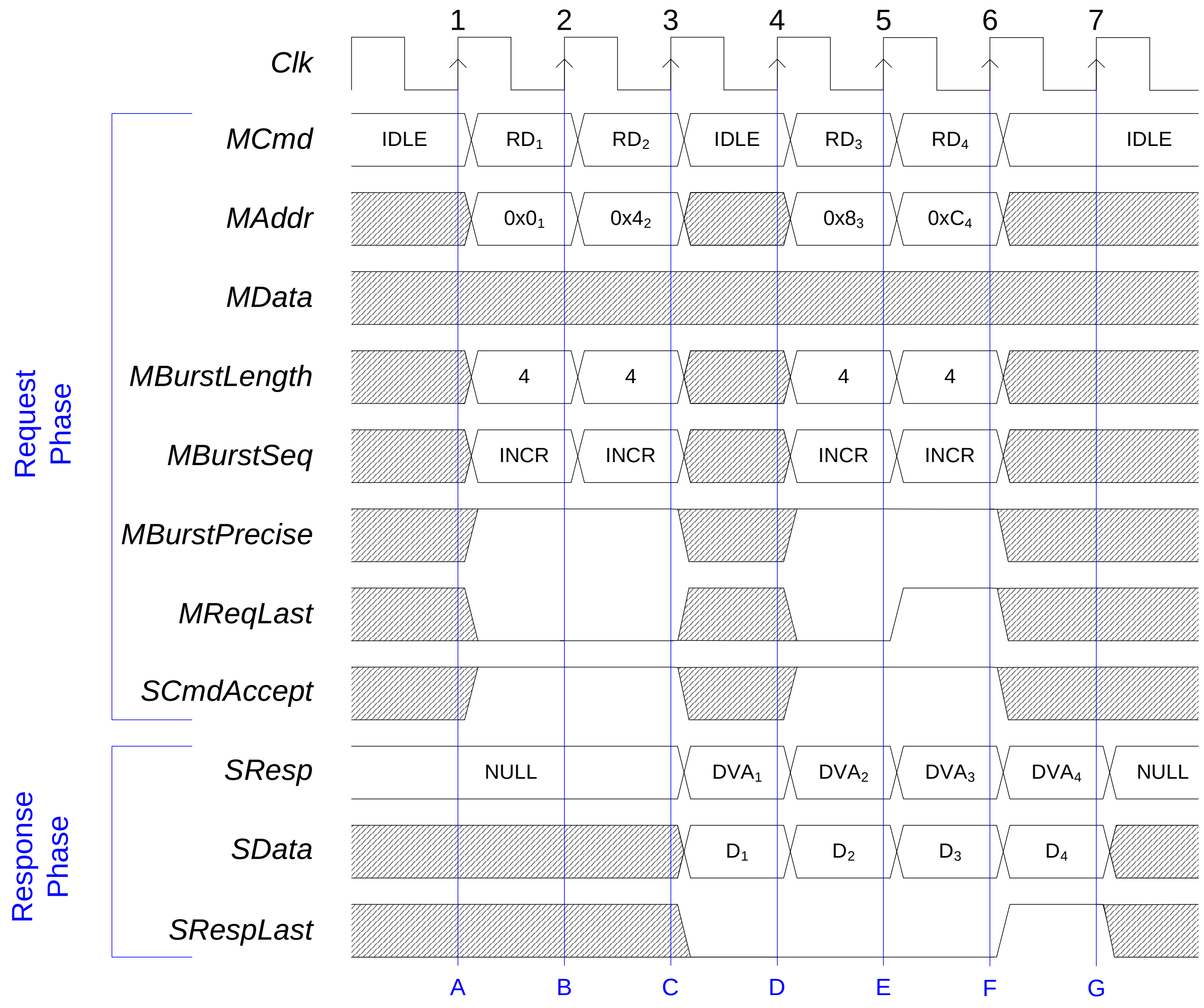
Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, four on MBurstLength, INCR on MBurstSeq, and asserts MBurstPrecise. MBurstLength, MBurstSeq, and MBurstPrecise must be kept constant during the burst. MReqLast must be deasserted until the last request in the burst. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The slave captures the address of the first request and keeps SCmdAccept asserted.

C. The master inserts an IDLE request in the middle of the burst. The slave does not have to deassert SCmdAccept, anticipating more burst requests to come. The slave captures the address of the second request. The slave responds to the first read by driving DVA on SResp and the read data on SData. The slave must keep SRespLast deasserted until the last response.

Figure 33 Incrementing Precise Burst Read with IDLE Cycle



D. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The master also captures the data for the first read from the slave. The slave responds to the second read by driving DVA on SResp and the read data on SData. If it has the data available for response, the slave does not have to insert a NULL response cycle.

E. The master issues the last request of the burst, incrementing MAddr, and asserting MReqLast. The master also captures the data for the second read from the slave. The slave captures the address of the third request and responds to the third request.

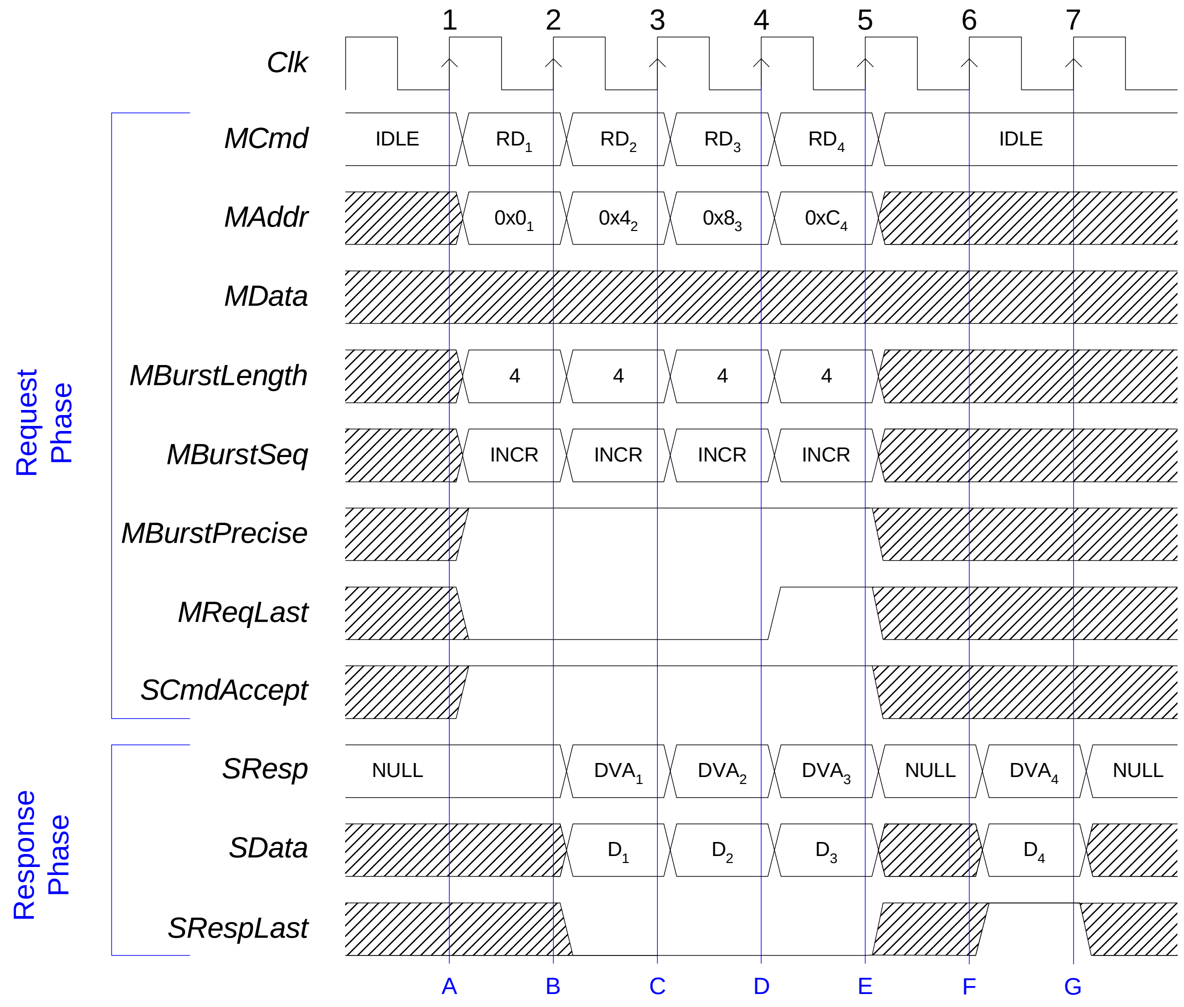
F. The master captures the data for the third read from the slave. The slave captures the address of the fourth request. The slave responds to the fourth request, and asserts SRespLast to indicate the end of the slave burst.

G. The master captures the data for the last read from the slave, ending the last response phase.

10.14 Incrementing Burst Read with NULL Response Cycle

Figure 34 illustrates a burst of four 32-bit words, incrementing precise burst read, with a NULL response cycle (wait state) inserted by the slave. Null cycles can be inserted into any burst type by the slave.

Figure 34 Incrementing Burst Read with Null Cycle



Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, four on MBurstLength, INCR on MBurstSeq, and asserts MBurstPrecise. MBurstLength, MBurstSeq and MBurstPrecise must be kept constant during the burst. MReqLast must be deasserted until the last request in the burst. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The slave captures the address of the first request and keeps SCmdAccept asserted. The slave responds to the first request by driving DVA on SResp and the read data on SData. The slave must keep SRespLast deasserted until the last response.

C. The master issues the next read in the burst. MAddr is set to the next word-aligned address (incremented by 4 in this case). The master also captures the data for the first read from the slave. The slave captures the address of the second request and keeps SCmdAccept asserted. The slave responds to the second request.

D. The master issues the last request of the burst, incrementing MAddr and asserting MReqLast. The master also captures the data for the second read from the slave. The slave captures the address of the third request and keeps SCmdAccept asserted. The slave responds to the third request.

E. The master captures the data for the third read from the slave. The slave captures the address of the fourth request and keeps SCmdAccept asserted. The slave cannot respond to the fourth request, so it inserts NULL to SResp.

F. The slave responds to the fourth request and asserts SRespLast to indicate the last response of the burst.

G. The master captures the data for the last read from the slave, ending the last response phase.

10.15 Single Request Burst Read

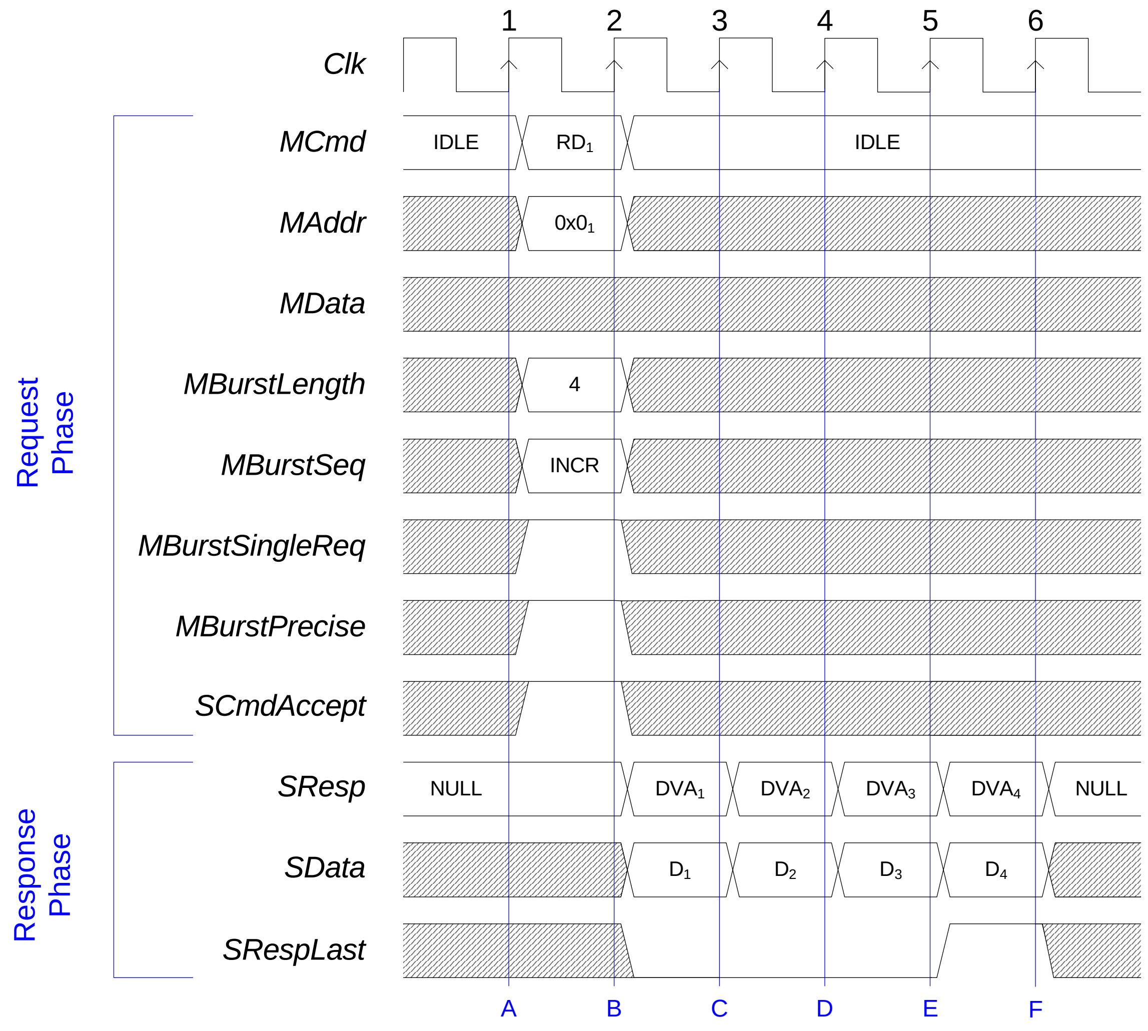
Figure 35 illustrates a single request, multiple data burst read. The master provides the burst length, start address, and burst sequence, and identifies the burst as a single request with the MBurstSingleReq signal. A single request burst is always precise.

Sequence

A. The master starts a read request by driving RD on MCmd, a valid address on MAddr, four on MBurstLength, INCR on MBurstSeq, and asserts MBurstPrecise, and MBurstSingleReq. The MBurstPrecise and MBurstSingleReq signals would normally be tied off to logic 1, which is not supplied by the master. The slave is ready to accept any request, so it asserts SCmdAccept.

B. The master completes the request cycles. The slave captures the address of the request. The slave responds to the request by driving DVA on SResp and the first response data on SData. The slave must keep SRespLast deasserted until the last response.

Figure 35 Single Request Burst Read



C. The master captures the first response data. The slave issues the second response.

D. The master captures the second response data. The slave issues the third response.

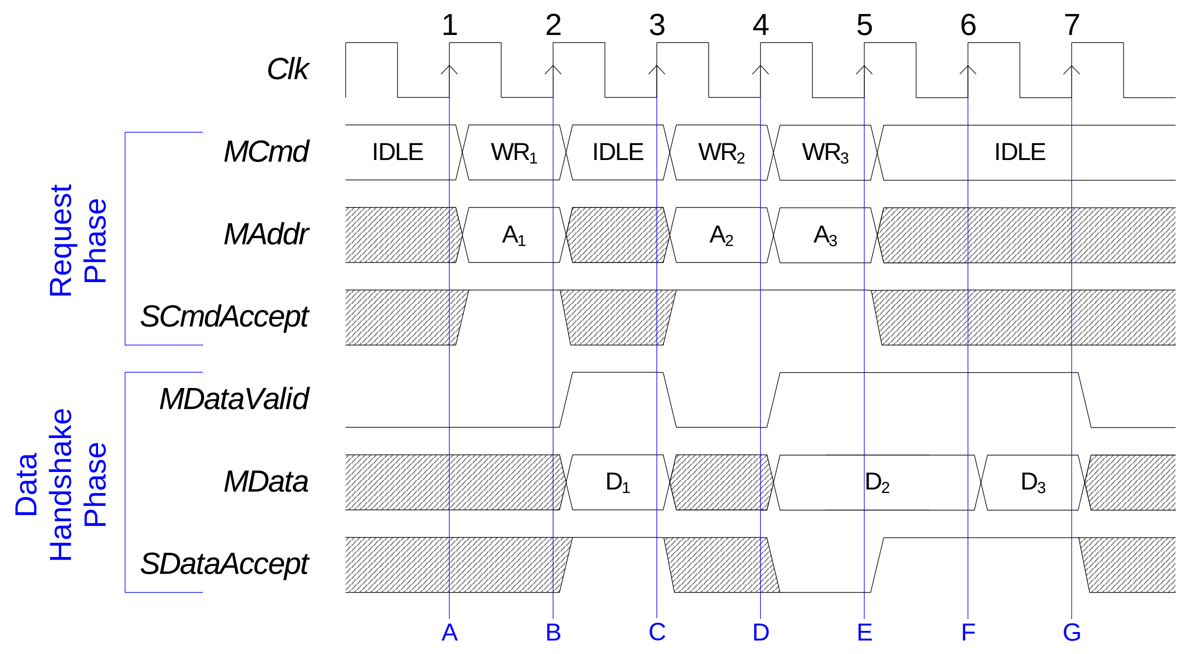
E. The master captures the third response data. The slave issues the fourth response, and asserts SRespLast to indicate the last response of the burst.

F. The master captures the last response data.

10.16 Datahandshake Extension

Figure 36 shows three writes with no responses using the datahandshake extension. This extension adds the datahandshake phase, which is completely independent of the request and response phases. Two signals, MDataValid and SDataAccept, are added, and MData is moved from the request phase to the datahandshake phase.

Figure 36 Datahandshake Extension



Sequence

A. The master starts a write request by driving WR on MCmd and a valid address on MAddr. It does not yet have the write data, however, so it deasserts MDataValid. The slave asserts SCmdAccept. It does not need to assert or deassert SDataAccept yet, because MDataValid is still deasserted.

B. The slave captures the write address from the master. The master is now ready to transfer the write data, so it asserts MDataValid and drives the data on MData, starting the datahandshake phase. The slave is ready to accept the data immediately, so it asserts SDataAccept. This corresponds to a data accept latency of 0.

C. The master deasserts MDataValid since it has no more data to transfer. (Like MCmd and SResp, MDataValid must always be in a valid, specified state.) The slave captures the write data from MData, completing the transfer. The master starts a second write request by driving WR on MCmd and a valid address on MAddr.

D. Since SCmdAccept is asserted, the master immediately starts a third write request. It also asserts MDataValid and presents the write data of the second write on MData. The slave is not ready for the data yet, so it deasserts SDataAccept.

E. The master sees that SDataAccept is deasserted, so it holds the values of MDataValid and MData. The slave asserts SDataAccept, for a data accept latency of 1.

F. Since SDataAccept is asserted, the datahandshake phase ends. The master is ready to deliver the write data for the third request, so it keeps MDataValid asserted and presents the data on MData. The slave captures the data for the second write from MData, and keeps SDataAccept asserted, for a data accept latency of 0 for the third write.

G. Since SDataAccept is asserted, the datahandshake phase ends. The slave captures the data for the third write from MData.

10.17 Burst Write with Combined Request and Data

Figure 37 illustrates a single request, multiple data burst write, with datahandshake signaling. Through the request handshake, the master provides the burst length, the start address, and burst sequence, and identifies the burst as a single request with the MBurstSingleReq signal.

The write data is transferred with a datahandshake extension (see Figure 36). The parameter `reqdata_together` forces the first data phase to start with the request, making the design of a slave state machine easier, since it only needs to track one request handshake during the burst. Without this parameter, the MDataValid signal could be asserted any time after the first request. If datahandshake is not used, a single-request write burst is not possible; instead a request is required for each burst word.

Sequence

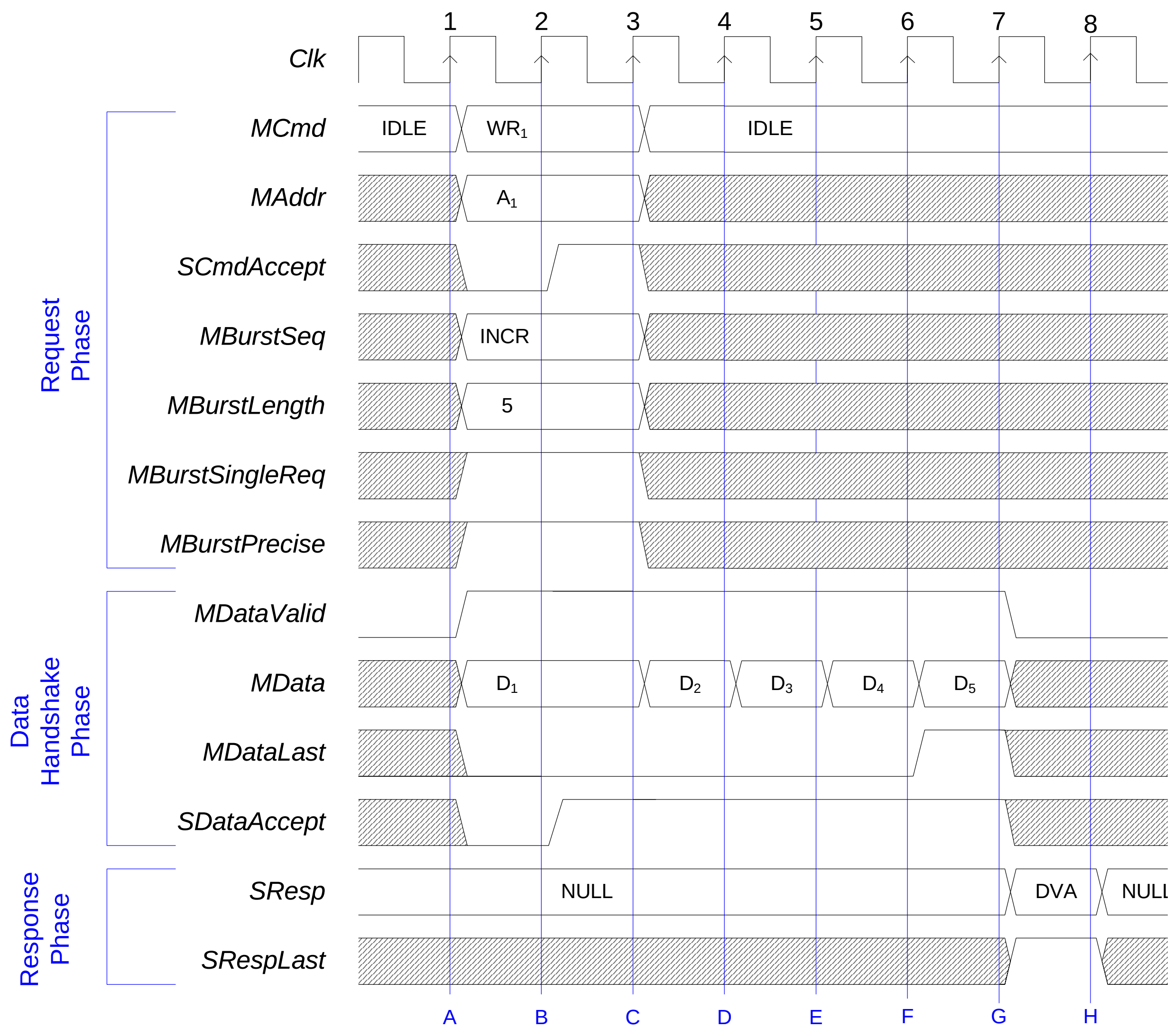
A. The master starts a write request by driving WR on MCmd, a valid address on MAddr, INCR on MBurstSeq, five on MBurstLength, and asserts the MBurstPrecise and MBurstSingleReq signals. The master also asserts the MDataValid signal, drives valid data on MData, and deasserts MDataLast. The MDataLast signal must remain deasserted until the last data cycle.

B. Since it has not received SCmdAccept or SDataAccept, the master holds the request phase signals, keeps MDataValid asserted, and MData steady. The slave asserts SCmdAccept and SDataAccept to indicate it is ready to accept the request and the first data phase.

C. The master completes the request phase, asserts MDataValid and drives new data to MData. The slave captures the initial data and keeps SDataAccept asserted to indicate it is ready to accept more data.

D. The master asserts MDataValid and drives new data to MData. The slave captures the second data phase and keeps SDataAccept asserted to indicate it is ready to accept more data.

Figure 37 Burst Write with Combined Request and Data



E. The master asserts MDataValid and drives new data to MData. The slave captures the third data phase and keeps SDataAccept asserted to indicate it is ready to accept more data.

F. The master asserts MDataValid, drives new data to MData, and asserts MDataLast to identify the last data in the burst. The slave captures the fourth data phase and keeps SDataAccept asserted to indicate it is ready to accept more data.

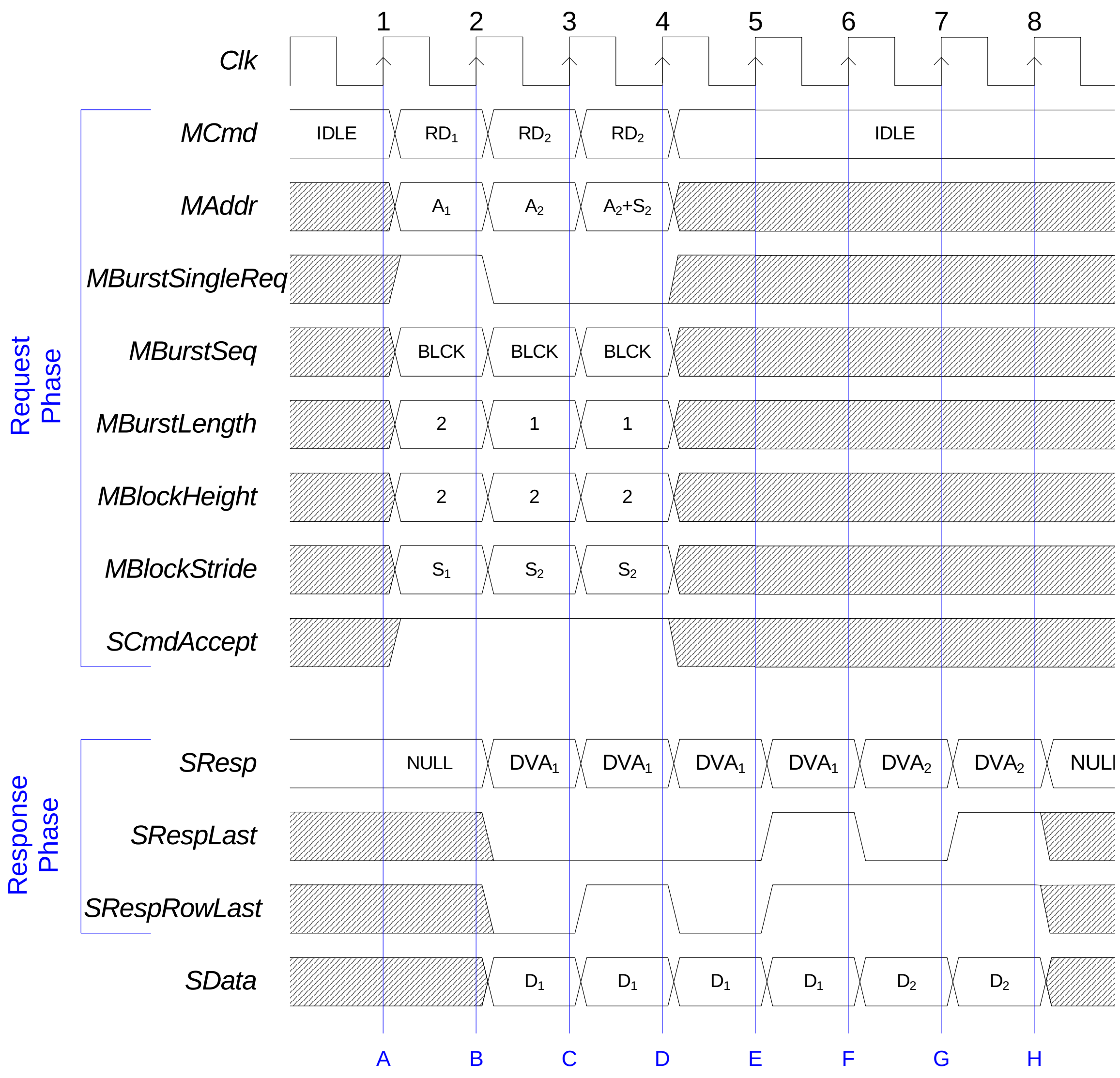
G. The slave captures the last data phase and address.

This example also shows how the slave issues SResp at the end of a burst (when the optional write response is configured in the interface). For single request / multiple data bursts there is only a single response, and it can be issued after the last data has been detected by the slave. The SResp is NULL until point G. in the diagram. The slave may use code DVA to indicate a successful burst, or ERR for an unsuccessful one.

10.18 2-Dimensional Block Read

Figure 38 illustrates two read bursts with a 2-dimensional block burst address sequence and optional response phase end-of-row (SRespRowLast) and end-of-burst (SRespLast) framing information. The first transaction is a single-request, multiple-data style block burst of two rows by two words per row, with an address stride of S_1 bytes. The second transaction is a multiple-request, multiple-data style block burst of two rows by one word per row, with an address stride of S_2 bytes. Block bursts are always precise.

Figure 38 2 Dimensional Block Read



Sequence

A. The master begins the first block read by asserting RD on MCmd, a valid address (A_1) on MAddr, BLCK on MBurstSeq, 2 words per row on MBurstLength, 2 rows on MBlockHeight, and the row-to-row spacing (S_1) on MBlockStride. The master identifies this as the only request for the read burst by asserting MBurstSingleReq. The slave asserts SCmdAccept signifying that it is ready to accept the request.

B. The rising edge of the OCP clock ends the first request phase as the slave captures the request. The master starts the second block read at address A_2, with only a single word per row, and requests 2 rows at a spacing of S_2. The master deasserts MBurstSingleReq, indicating that there will be one request phase for each data phase. The slave keeps SCmdAccept asserted. The slave also returns a response to the original block burst, including the first word of data. Since there are more data words remaining in the first row of this burst, the slave deasserts SRespRowLast and SRespLast.

C. The slave captures the first request for the second transaction, and keeps SCmdAccept asserted for the next cycle. The master presents the second (and last) request in the second block burst. The master sets MAddr to the starting address for the second row ($A_2 + S_2$). The master accepts the first response for the first burst. The slave returns the second data word for the first burst, which ends the first row, so the slave also asserts SRespRowLast.

D. The slave accepts the second request for the second transaction. The master accepts the second response for the first burst. The slave returns the third data word for the first burst, which is the first word of the second row, so the slave deasserts SRespRowLast.

E. The master accepts the third response for the first burst. The slave returns the fourth (and final) data word for the first burst, and asserts SRespLast and SRespRowLast.

F. The master accepts the last response for the first burst. The slave returns the first data word for the second burst, which ends the first row, so the slave keeps SRespRowLast asserted and deasserts SRespLast.

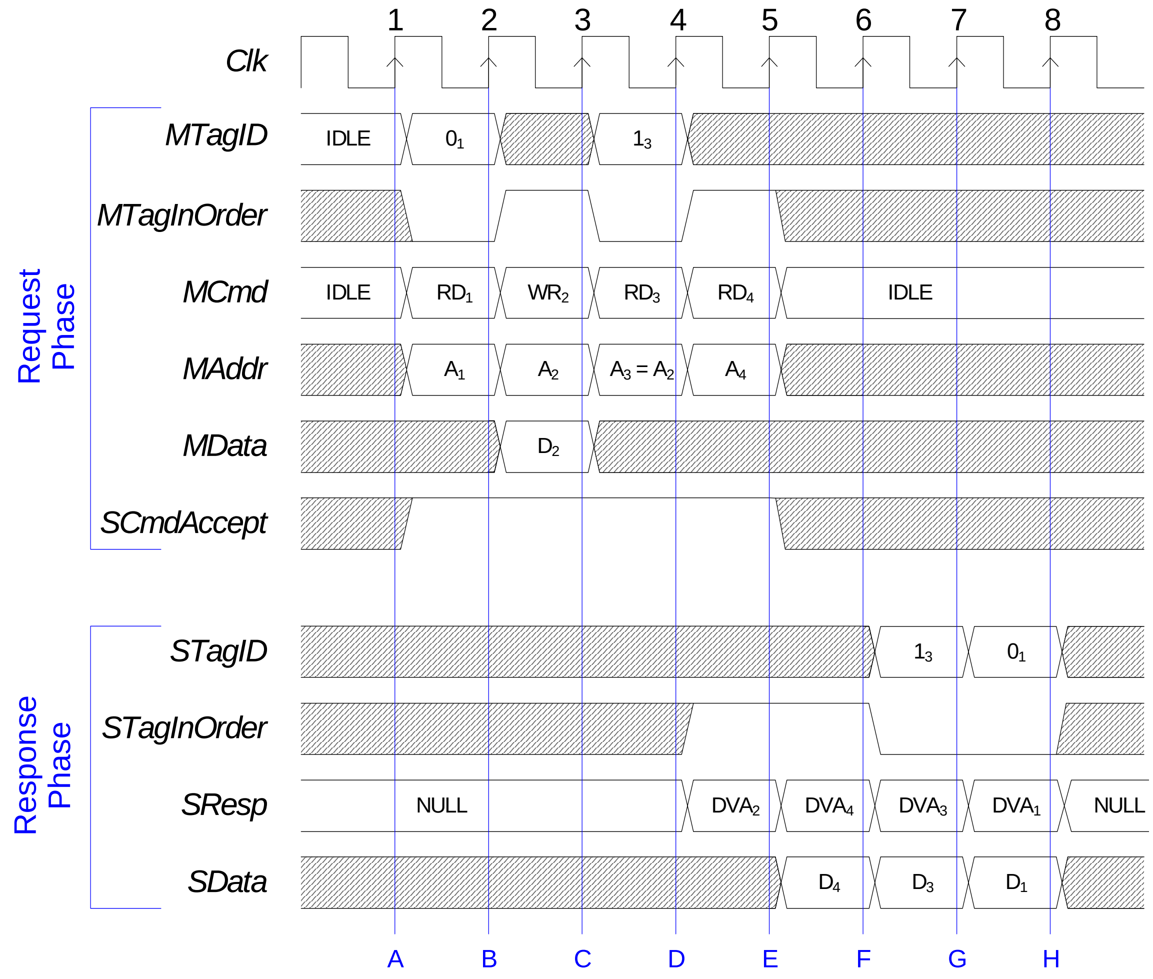
G. The master accepts the first response for the second burst. The slave returns the second (and final) data word for the second burst, and asserts SRespLast.

H. The master accepts the last response for the second burst.

10.19 Tagged Reads

Figure 39 illustrates out-of-order completion of read transfers using the OCP tag extension. The tag IDs, MTagID and STagID, have been added, along with the MTagInOrder and STagInOrder signals. Writes are configured to have responses. There is significant reordering of responses, together with in-order responses forced by both MTagInOrder and address overlap.

Figure 39 Tagged Reads



Sequence

A. The master starts the first read request, driving a RD on MCmd and a valid address on MAddr. The master drives a 0 on MTagID, indicating that this read request is for tag 0. The master also deasserts MTagInOrder, allowing the slave to reorder the responses.

B. Since SCmdAccept is sampled asserted, the request phase ends with a request accept latency of 0. The master begins a write request to a second address, providing the write data on MData. The master asserts MTagInOrder, indicating that the slave may not reorder this request with respect to other in-order transactions and that MTagID is a “don’t care.”

C. When SCmdAccept is sampled asserted, the second request phase ends. The master launches a third request, which is a read to an address that matches the previous write. MTagInOrder is deasserted, enabling reordering, and the assigned tag value is 1.

D. Since SCmdAccept is sampled asserted, the third request phase ends. The master launches a fourth request, which is a read. MTagInOrder is asserted, forcing ordering with respect to the earlier in-order write. The slave responds to the second request (the in-order write presented at B) by driving DVA on SResp. Since the transaction is in-order, STagInOrder is asserted and STagID is a “don’t care.”

E. SCmdAccept is sampled asserted so the fourth request phase ends. Since the response phase is sampled asserted, the response to the second request ends with a request-to-response latency of 2 cycles. The slave responds to the fourth request (D) by driving DVA on SResp and read data on SData. STagInOrder is asserted to match the associated request. This response was reordered with respect to the first (A) and third (C) requests, which allow reordering.

F. The response phase is sampled asserted so the response to the fourth request ends with a request-to-response latency of 1 cycle. The slave responds to the third request (C) by driving DVA on SResp, read data on SData, and a 1 on STagID. STagInOrder is deasserted, indicating that reordering is allowed. This response is reordered with respect to the first request (A), but must occur after the second request (B), which has a matching address.

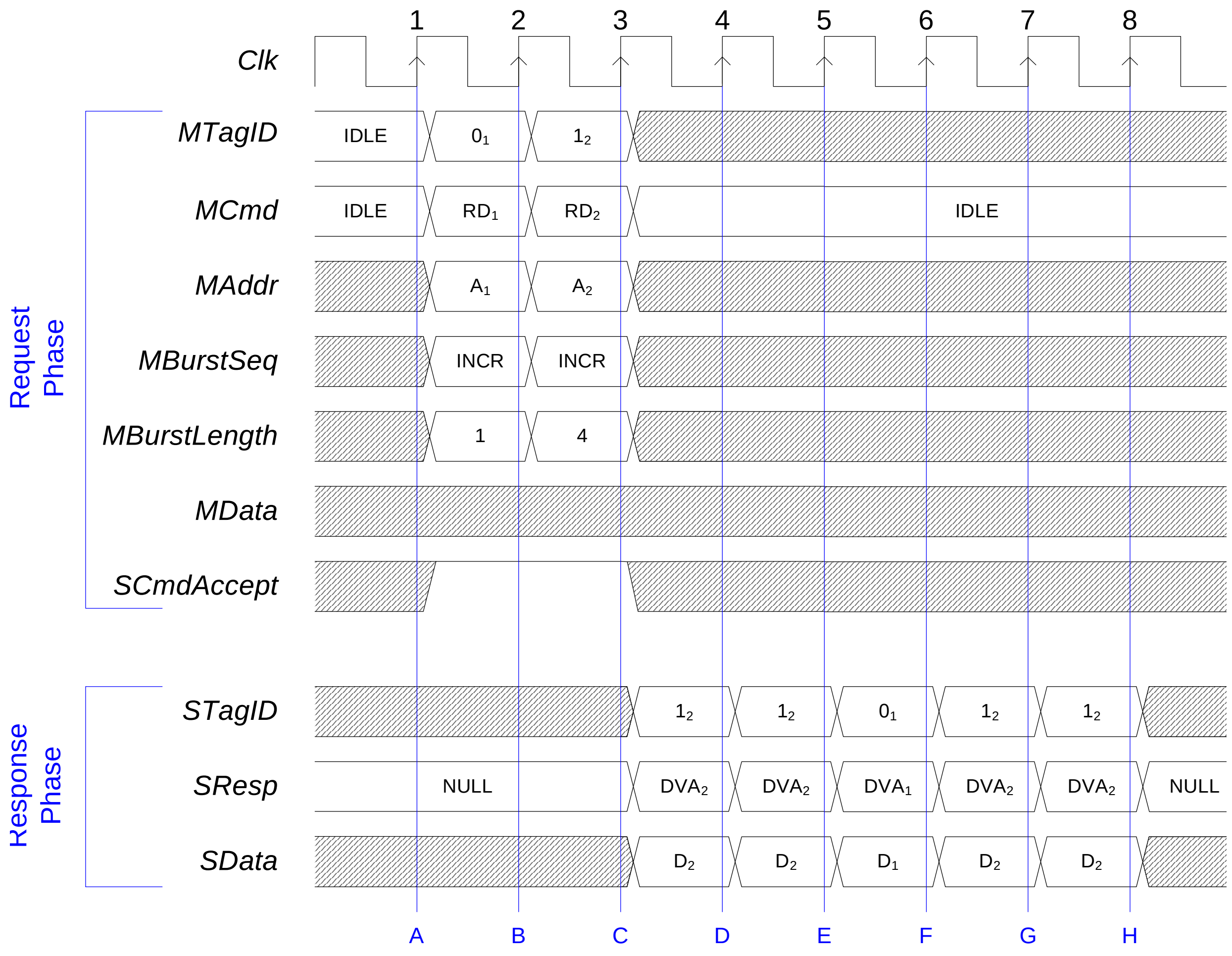
G. Since the response phase is sampled asserted, the response to the third request ends with a request-to-response latency of 3 cycles. The slave responds to the first request (A) by driving DVA on SResp, read data on SData, and a 0 on STagID. STagInOrder is deasserted.

H. When the response phase is sampled asserted, the response to the first request ends with a request-to-response latency of 6 cycles.

10.20 Tagged Bursts

Figure 40 illustrates out-of-order completion of packing single-request/ multiple data read transactions using the OCP tag extension. With the `burstprecise` parameter set to 0, and the MBurstPrecise signal tied-off to the default, all bursts are precise. The `burstsinglereq` parameter is 0, and the MBurstSingleReq signal is tied-off to 1 (not the default), so all requests have a single request phase.The `taginorder` parameter is set to 0, allowing all transactions to be reordered, subject to the tagging rules. The `tag_interleave_size` parameter is set to 2, so packing bursts must not interleave at a granularity finer than 2 words. Note that the first two words of the second read return before the only word associated with the first read.

Figure 40 Tagged Burst Transactions



Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The request is for a single-word incrementing burst, as driven on MBurstLength and MBurstSeq, respectively. The master also drives a 0 on MTagID, indicating that this read request is for tag 0.

B. Once SCmdAccept is sampled asserted, the request phase ends with a request accept latency of 0. The master begins a four-word read request to a second, non-conflicting address, on tag 1.

C. SCmdAccept is sampled asserted ending the second request phase. The slave responds to the second request by driving DVA on SResp together with 1 on STagID. The slave provides the first data word from the burst on SData.

D. When the response phase is sampled asserted, the first response to the second request ends with a request-to-response latency of 1 cycle. The slave provides the second word of read data for tag 1 on SData, together with a DVA response. Because `tag_interleave_size` is 2 and the read burst sequence is packing, the slave was forced to return the second word of tag 1's data before responding to tag 0.

E. The response phase is sampled asserted, terminating the second response to the second request with a request-to-response latency of 2 cycles. The slave responds to the first request by providing the read data for tag 0 together with a DVA response.

F. When the response phase is sampled asserted, the response to the first request ends with a request-to-response latency of 4 cycles. Since the burst length of the first request is 1, the transaction on tag 0 is complete. The slave provides the third word of read data for tag 1 on SData, together with a DVA response.

G. The response phase is sampled asserted so the third response to the second request ends with a request-to-response latency of 4 cycles. The slave provides the fourth word of read data for tag 1 on SData, together with a DVA response.

H. When the response phase is sampled asserted, the fourth and final response to the second request ends with a request-to-response latency of 5 cycles.

10.21 Threaded Read

Figure 41 illustrates out-of-order completion of read transfers using the OCP thread extension. This diagram is developed from Figure 28 on page 168. The thread IDs, MThreadID and SThreadID, have been added, and the order of two of the responses has been changed.

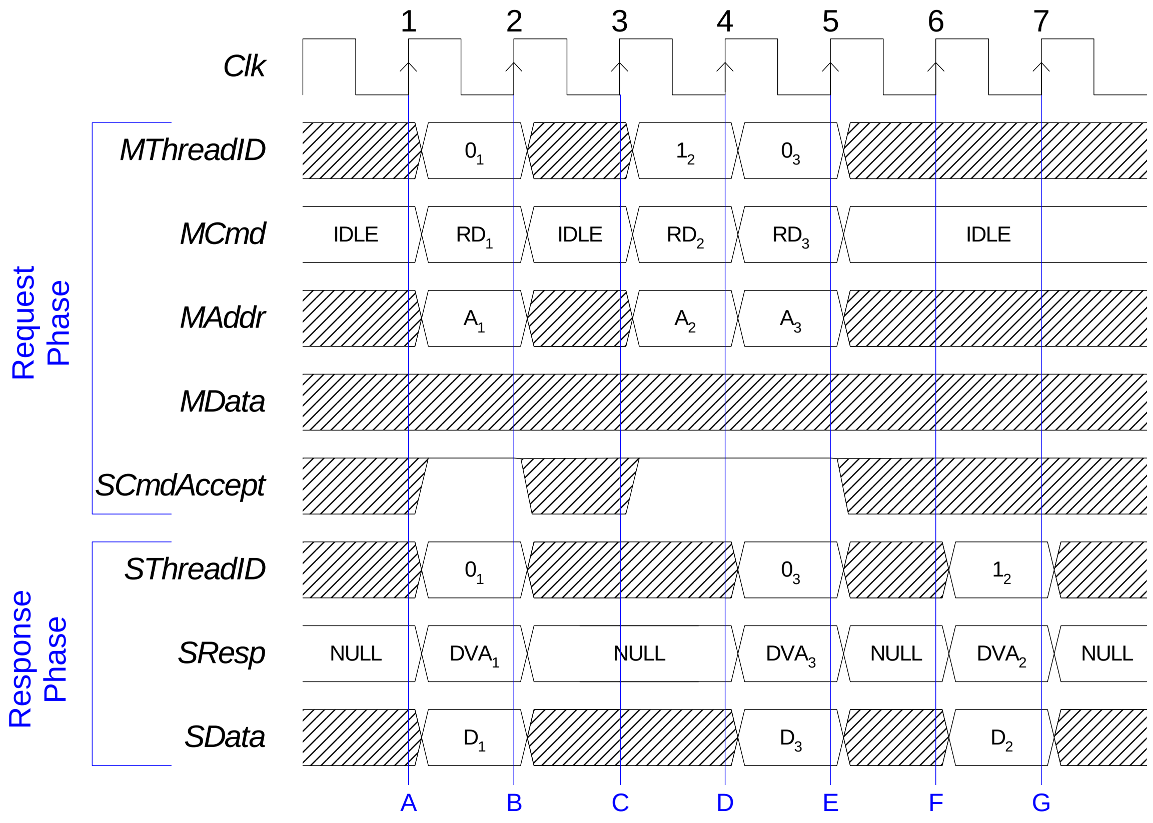
Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The master also drives a 0 on MThreadID, indicating that this read request is for thread 0. The slave asserts SCmdAccept, for a request accept latency of 0. When the slave sees the read command, it responds with DVA on SResp and valid data on SData. The slave also drives a 0 on SThreadID, indicating that this response is for thread 0.

B. Since SCmdAccept is asserted, the request phase ends. The master sees that SResp is DVA and captures the read data from SData. Because the request was accepted and the response was presented in the same cycle, the request-to-response latency is 0.

C. The master launches a new read request, but this time it is for thread 1. The slave asserts SCmdAccept, however, it is not ready to respond.

Figure 41 Threaded Read



D. Since SCmdAccept is asserted, the master can launch another read request. This request is for thread 0, so MThreadID is switched back to 0. The slave captures the address of the second read for thread 1, but it begins driving DVA on SResp, data on SData, and a 0 on SThreadID. This means that it is responding to the third read, before the second read.

E. Since SCmdAccept is asserted, the third request ends. The master sees that the slave has produced a valid response to the third read and captures the data from SData. The request-to-response latency for this transfer is 0.

F. The slave has the data for the second read, so it drives DVA on SResp, data on SData, and a 1 on SThreadID.

G. The master captures the data for the second read from SData. The request-to-response latency for this transfer is 3.

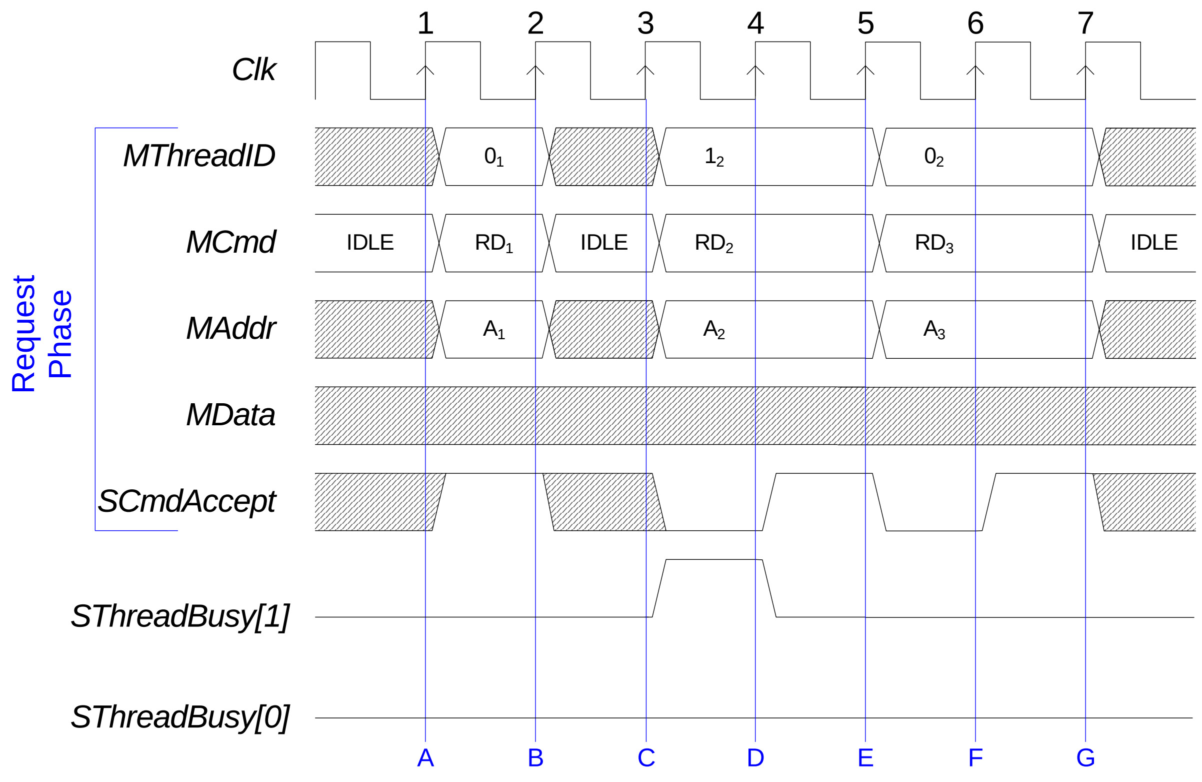
10.22 Threaded Read with Thread Busy

Figure 42 illustrates the out-of-order completion of read transfers using the OCP thread extension. The change to Figure 41 is the addition of thread busy signals. In this example, the thread busy is only a hint, since the `sthreadbusy_exact` parameter is not set. In this case the master may ignore the SThreadBusy signals, and the slave does not have to accept requests even when it is not busy.

When thread busy is treated as a hint and a request or thread is not accepted, the interface may block for all threads. Blocking of this type can be avoided by treating thread busy as an exact signal using the `sthreadbusy_exact` parameter. For an example, see Section 10.23.

This example shows only the request part of the read transfers. The response part can use a similar mechanism for thread busy.

Figure 42 Threaded Read with Thread Busy



Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The master also drives a 0 on MThreadID, associating this read request with thread 0. The slave asserts SCmdAccept for a request accept latency of 0.

B. Since SCmdAccept is asserted, the request phase ends.

C. The slave asserts SThreadBusy[1] since it is not ready to accept requests on thread 1. The master ignores the hint, and launches a new read request for thread 1. The master can issue a request even though the slave asserts SThreadbusy (see transfer 2). All threads are now blocked.

D. The slave deasserts SThreadBusy[1] and asserts SCmdAccept to complete the request for thread 1.

E. Since SCmdAccept is asserted, the second request ends. The master issues a new request to thread 0. The slave is not ready to accept the request, and indicates this condition by keeping SCmdAccept deasserted. It chooses not to assert SThreadBusy[0]. The slave does not have to assert SCmdAccept for a request, even if it did not assert SThreadbusy (see transfer 3).

F. The slave asserts the SCmdAccept to complete the request on thread 0.

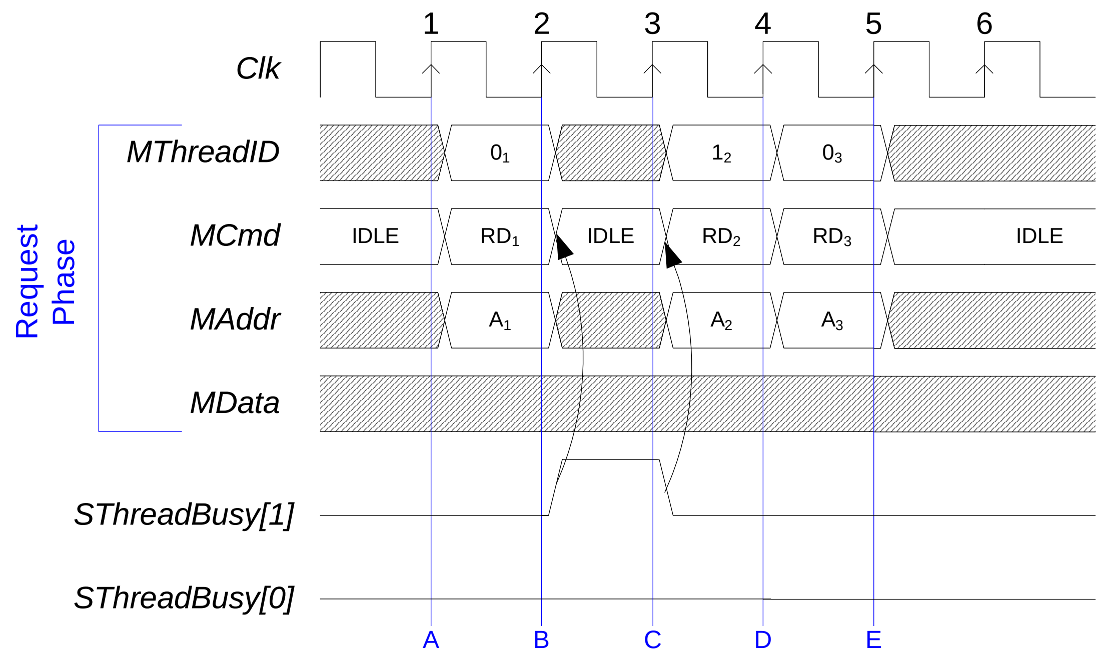
G. The master captures the SCmdAccept to complete the requests.

10.23 Threaded Read with Thread Busy Exact

Figure 43 illustrates the out-of-order completion of read transfers using the OCP thread extension. Because the `sthreadbusy_exact` parameter is set, the master may not ignore the SThreadBusy signals. The master is using SThreadBusy to control thread arbitration, so it cannot present a command on Thread 1 as the slave asserts SThreadbusy[1].

The diagram only shows the request part of the read transfers. The response part can use a similar mechanism for thread busy.

Figure 43 Threaded Read with Thread Busy Exact



Sequence

A. The master starts the first read request, driving RD on MCmd and a valid address on MAddr. The master also drives a 0 on MThreadID, indicating that this read request is for thread 0.

B. Since SThreadBusy[0] is not asserted, the request phase ends. The slave samples the data and address and asserts SThreadBusy[1] since it is unready to accept requests on thread 1. The master is prevented from sending a request on thread 1, but it can send a request on another thread.

C. The slave deasserts SThreadBusy[1] and the master can send the request on thread 1.

D. Since SThreadBusy[1] is not asserted, the request phase ends and the slave must sample the data and address. The master can send a request on thread 0 (or thread 1).

E. Since SThreadBusy[0] is not asserted, the request phase ends and the slave must sample the data and address.

10.24 Threaded Read with Pipelined Thread Busy

Figure 44 illustrates a set of threaded read requests on an interface where the `sthreadbusy_pipelined` parameter is set. Because pipelining a phase's ThreadBusy signals also forces exact flow control (`sthreadbusy_exact` must be set), the master must obey the SThreadBusy signals.

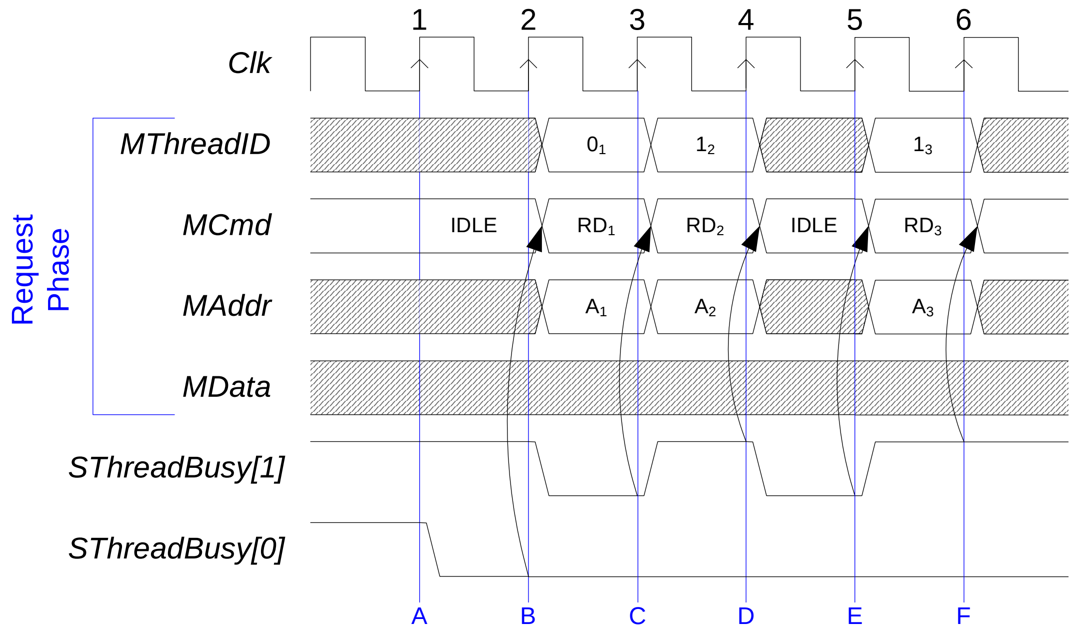
In this example, the master asserts a single read request phase on thread 0, and multiple requests on thread 1. The slave's SThreadBusy assertions control when the master may assert request phases on each thread. The diagram only shows the request part of the read transfers. The response part uses a similar mechanism for thread busy.

Sequence

A. Because both SThreadBusy signals were sampled asserted on this rising edge of the OCP clock, the master may not present requests on either thread. The slave indicates its readiness to accept a request on thread 0 in the next cycle by de-asserting SThreadBusy[0].

B. After sampling SThreadBusy[0] deasserted, the master asserts the first read request on thread 0 by driving a 0 on MThreadID, RD on MCmd and a valid address on MAddr. The slave indicates that it can accept requests on both threads in the next cycle by de-asserting SThreadBusy[1] and leaving SThreadBusy[0] deasserted.

Figure 44 Threaded Read with Pipelined Thread Busy



C. The master's first request is sampled by the slave and the request phase ends. The master samples SThreadBusy[1] deasserted and uses the information to assert a second read request, this time on thread 1. The slave asserts SThreadBusy[1] since it cannot guarantee that it can accept another request on thread 1 in the next cycle.

D. The master's second request is sampled by the slave and the request phase ends. The master samples SThreadBusy[1] asserted, and is forced to drive MCmd to IDLE, since it has no more requests for thread 0 and the slave cannot accept a request on thread 1. The slave signals that it will be ready to accept requests on both threads in the next cycle by de-asserting SThreadBusy[1] and leaving SThreadBusy[0] deasserted.

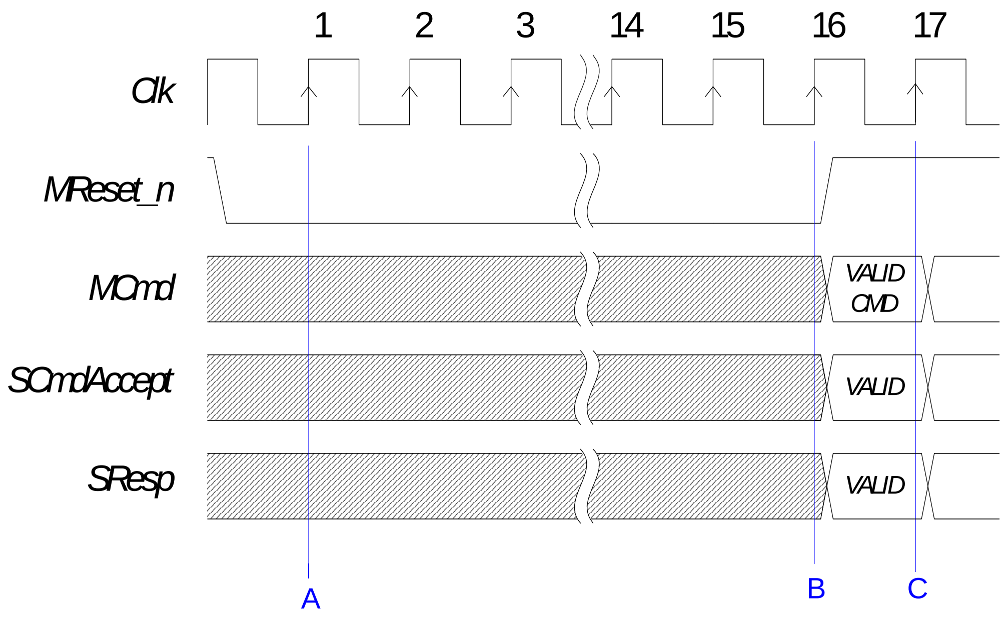
E. The master samples SThreadBusy[1] deasserted, and uses this information to assert a third read request on thread 1. The slave asserts SThreadBusy[1] since it cannot guarantee that it can accept another request on thread 1 in the next cycle.

F. The master's third request is sampled by the slave and the request phase ends. The master samples SThreadBusy[1] asserted, and is forced to drive MCmd to IDLE.

10.25 Reset

Figure 45 shows the timing of the reset sequence with MReset_n driven from the master to the slave. MReset_n must be asserted for at least 16 cycles of the OCP clock to ensure that the master and slave reach a consistent internal state. Because the interface does not include the EnableClk signal, the OCP clock is simply Clk.

Figure 45 Reset Sequence



Sequence

A. MReset_n is sampled active on this clock edge. Master and slave now ignore all other OCP signals, except for the connection signals, if present. In the first cycle a response to a previously issued request is presented by the slave and ready to be received by the master. Since the master is asserting MReset_n, the response is not received. The associated transaction is terminated by OCP reset so the response is withdrawn by the slave.

B. MReset_n is asserted for at least 16 Clk cycles.

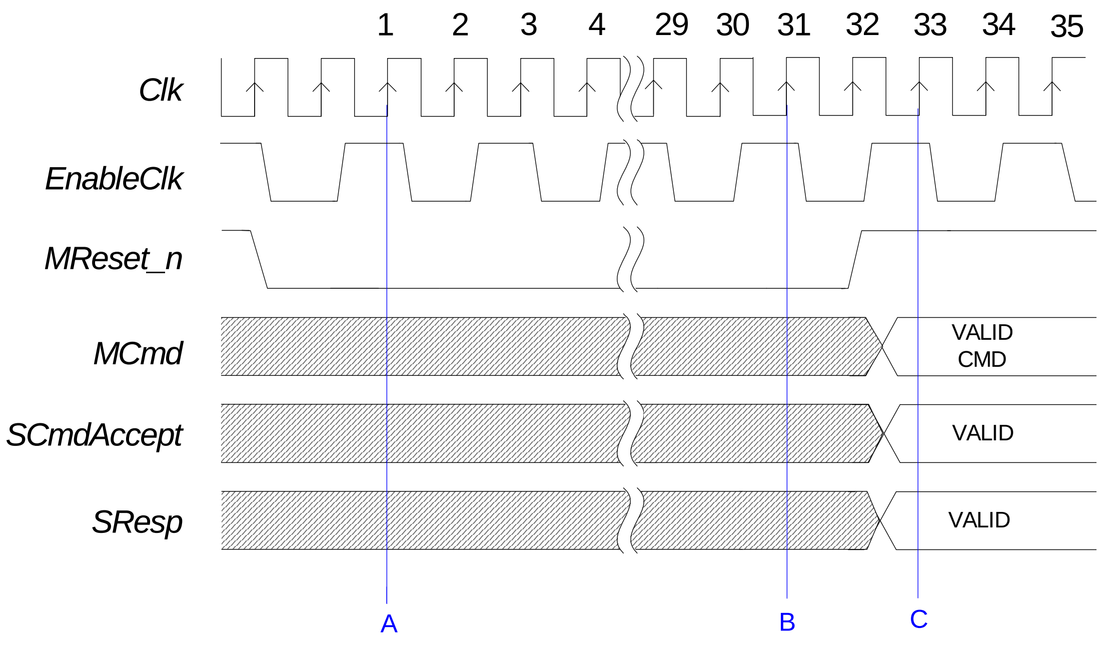
C. A new transfer may begin on the same clock edge that MReset_n is sampled deasserted.

10.26 Reset with Clock Enable

Figure 46 shows the timing of the reset signal with the EnableClk signal enabled on the interface. In this figure, the EnableClk signal is asserted on every other rising edge of Clk, delivering an OCP clock that is one-half the frequency of Clk. The MReset_n signal is driven from the master to the slave.

However, the presence of EnableClk means that MReset_n must be asserted for 16 cycles of the OCP clock (that is, when the rising edge of Clk samples EnableClk asserted), which will require 31 cycles of Clk.

Figure 46 Reset with Clock Enable



Sequence

A. MReset_n and EnableClk are sampled active on this clock edge. Master and slave now ignore all other OCP signals, except for the connection signals, if present.

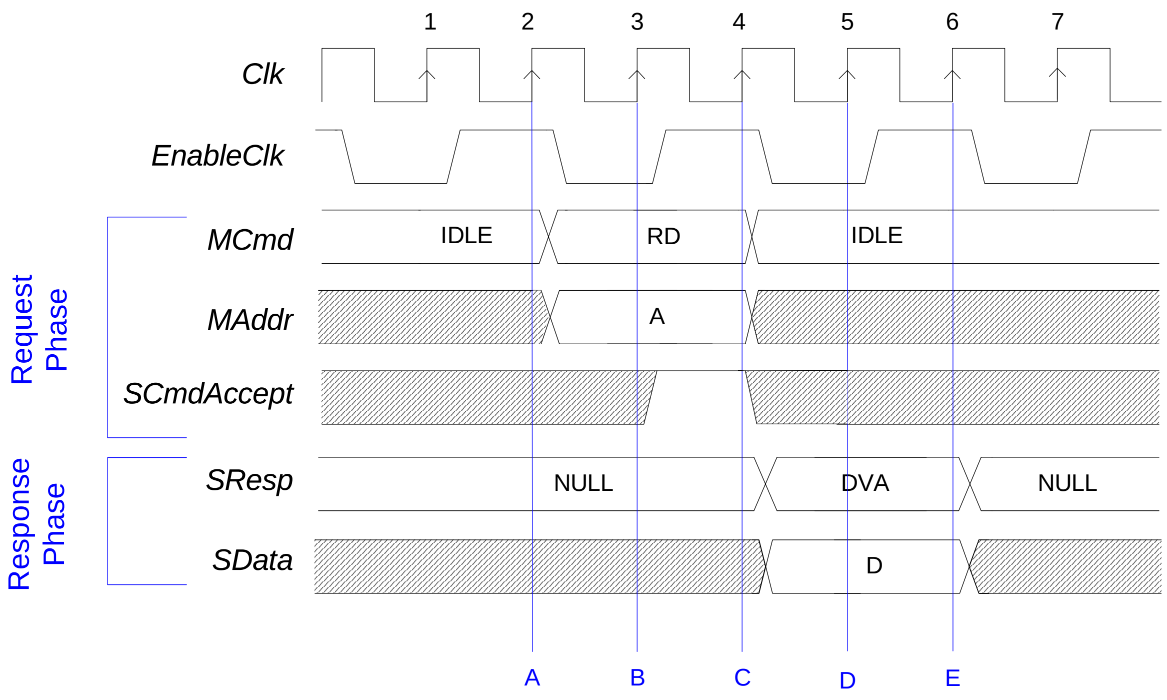
B. MReset_n is asserted for at least 16 Clk cycles with EnableClk sampled high.

C. A new transfer may begin on the same clock edge that MReset_n is sampled deasserted and EnableClk sampled high.

10.27 Basic Read with Clock Enable

Figure 47 illustrates a simple read transaction of length one with the EnableClk signal enabled on the interface. In this figure, the EnableClk signal is asserted on every other rising edge of Clk, delivering an OCP clock that is one-half the frequency of Clk. As is shown, interface state only advances on rising edges of Clk that coincide with EnableClk being asserted.

Figure 47 Basic Read with Clock Enable Signal



Sequence

A. The master starts a read request by driving RD on MCmd. At the same time, it presents a valid address on MAddr.

B. This clock edge is not valid since EnableClk is sampled low. The slave asserts SCmdAccept in this cycle for a request accept latency of 0.

C. The slave uses the value from MAddr to determine what data to return. The slave starts the response phase by switching SResp from NULL to DVA. The slave also drives the selected data on SData. Since SCmdAccept is asserted, the request phase ends.

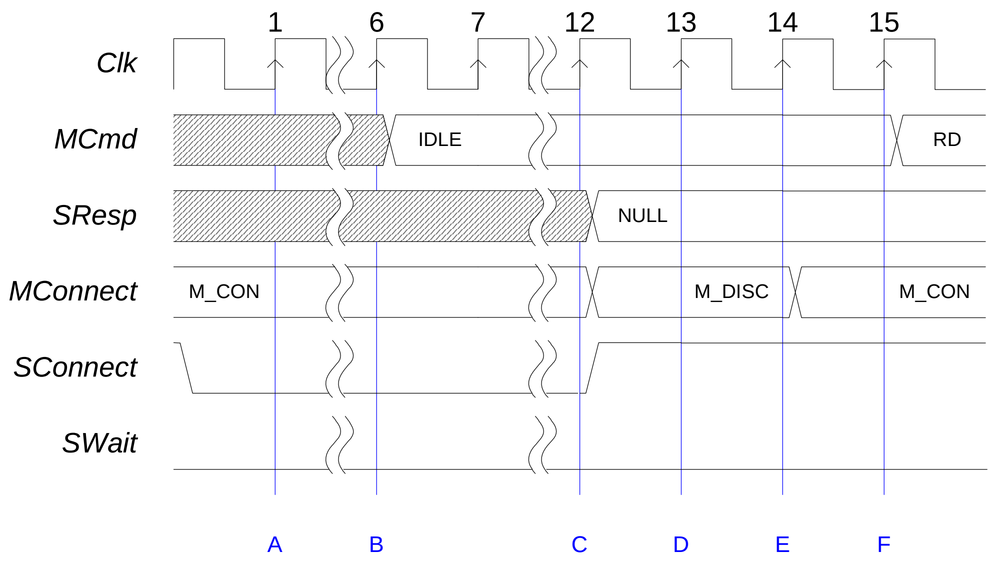
D. Invalid clock edge.

E. The master recognizes that SResp indicates data valid and captures the read data from SData, completing the response phase. This transfer has a request-to-response latency of 1.

10.28 Slave Disconnect

Figure 48 illustrates a sequence where a slave device votes for disconnecting from the master. After the master ensures that the interface is quiescent (all transactions are completed), it changes the connection state to a slave-requested disconnect. Just as the state change occurs, the slave votes to re-connect. After the minimum allowed delay of 2 cycles, the master changes the connection state back to fully connected.

Figure 48 Slave-requested Disconnect Sequence



Sequence

A. The master samples the slave’s vote to disconnect on SConnect. The master begins draining the interface by completing the request and datahandshake phases for any transactions that have already begun.

B. Having sequenced the final request phase for the last in-flight transaction, the master has drained the request phase and asserts MCmd to IDLE. The master continues draining the interface by waiting for any outstanding response phases from the slave.

C. Having sequenced the final response phase for the last in-flight transaction, the slave has drained the response phase and asserts SResp to NULL. The master samples the final response and can change the connection state directly to M_DISC without passing through M_WAIT, since SWait is negated and the interface is quiescent. Independently, the slave votes to connect by asserting SConnect.

D. The master samples the slave’s vote to connect, but cannot change the connection state until 2 cycles have passed.

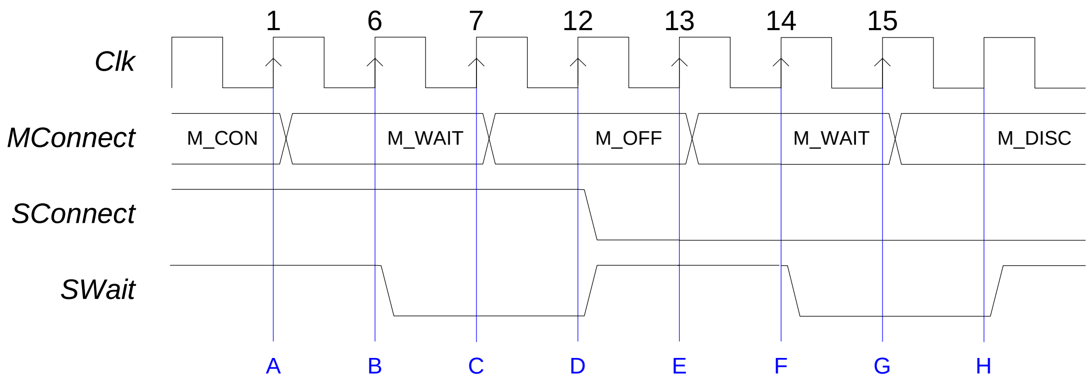
E. The master re-establishes connection by changing the connection state to M_CON. The master may not assert any new transaction until the slave samples the new connection state.

F. The slave samples MConnect and the interface is fully connected. The master asserts a read transaction on MCmd.

10.29 Connection Transitions with Slave Pacing

Figure 49 illustrates a sequence where several connection state transitions are delayed by the slave using SWait. This slave asserts SWait so it can pace (i.e., control) when the master may transition between stable connection states. In each case, the master is thus forced to transition through the M_WAIT transient state on the way to the desired stable state. The data flow and other sideband signals are not shown in the figure for clarity; their behavior is described in the sequence description, below.

Figure 49 Connection Transition Sequences with Slave Pacing



Sequence

A. The master votes to disconnect (not visible from the interface). Because all data flow transactions are complete, the interface is quiescent and the master changes the connection state by asserting MConnect to M_WAIT, since SWait is asserted.

B. The slave samples the M_WAIT state and determines that all sideband signaling is quiescent and the slave is ready to allow disconnect. The slave negates SWait to enable the master to complete the disconnection sequence.

C. The master samples SWait negated and changes the connection state to a master-requested disconnect by asserting MConnect to M_OFF.

D. The slave samples the M_OFF state and asserts SWait to pace future state changes. The slave also votes to disconnect by negating SConnect.

E. The master votes to connect (not visible from the interface) but samples SConnect negated and begins the transition to a slave-requested disconnect. Because SWait is asserted, the master first asserts MConnect to M_WAIT.

F. The slave samples the M_WAIT state and determines that it is ready to allow the connection state change, so it negates SWait.

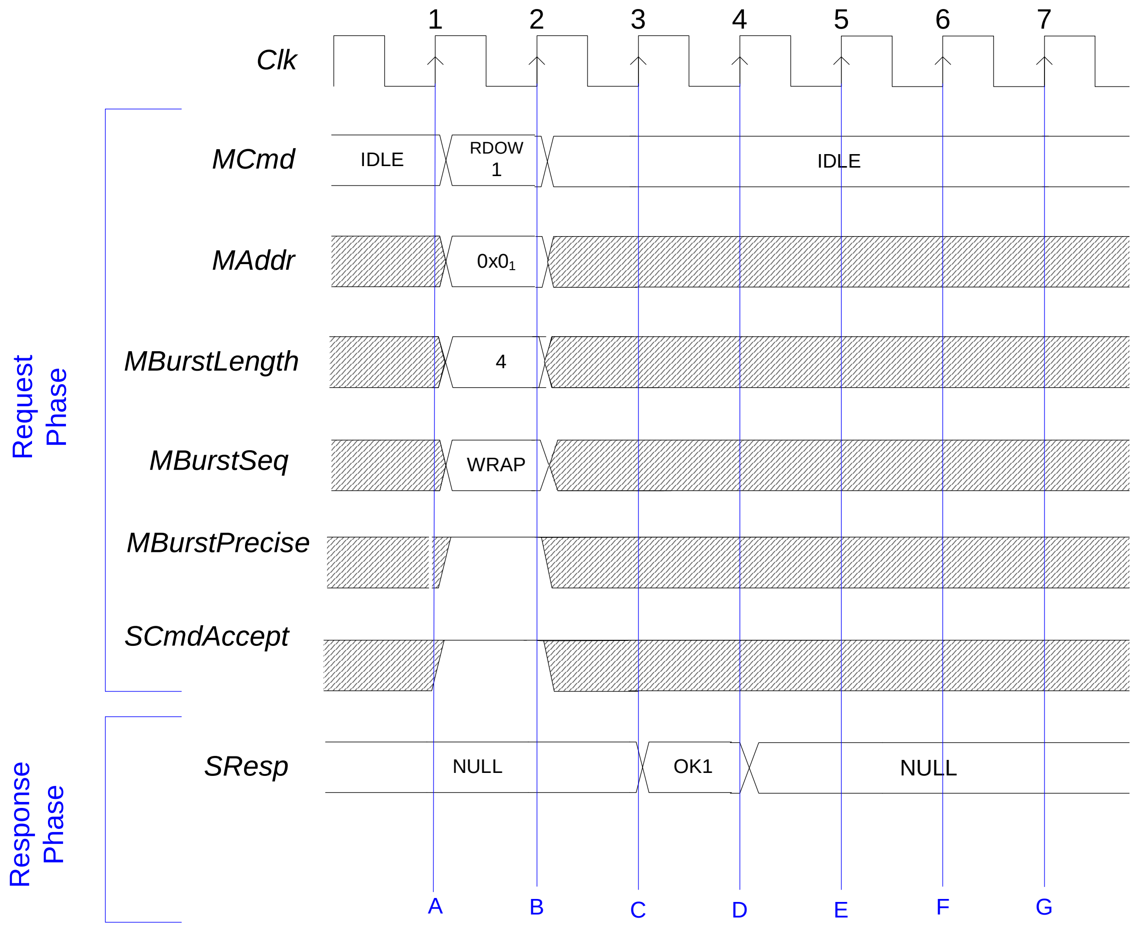
G. The master samples SWait negated and changes the connection state to a slave-requested disconnect by asserting MConnect to M_DISC.

H. The slave samples the M_DISC state and asserts SWait to pace future state changes.

11 OCP Coherence Extensions: Timing Diagrams

The following timing diagrams show the basic transfer flow on the intervention port.

Figure 50 Transfer without Data Phase, intport_split_tranx=0



This example has port parameter intport_split_tranx=0.

Sequence

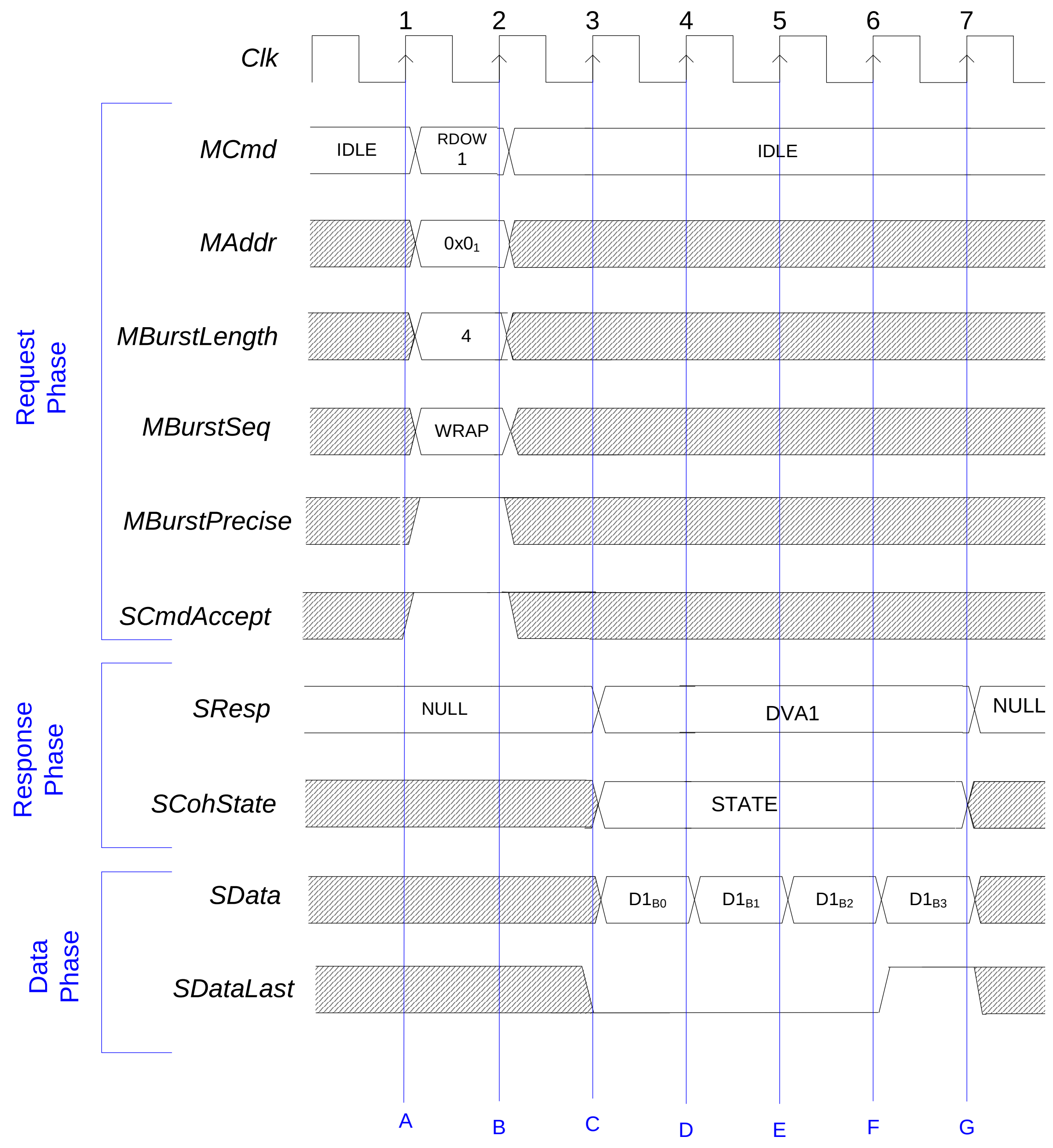
A. The master starts the request on clock 1 by driving the associated request group signals. The slave asserts SCmdAccept in the same cycle.

B. The slave captures the request signal group values and the request phase completes. The slave does the snoop intervention operation.

C. The slave reports the results of the snoop intervention operation. The slave's cache does not contain the requested address so the response of “OK” is given on the SResp signal.

D. The master recognizes the value on SResp and the transfer is completed.

Figure 51 Transfer with Data Phase, intport_split_tranx=0



This example has port parameter intport_split_tranx=0.

Sequence

A. The master starts the request on clock 1 by driving the associated request group signals. The slave asserts SCmdAccept in the same cycle.

B. The slave captures the request signal group values and the request phase completes. The slave does the snoop intervention operation.

C. The slave reports the results of the snoop intervention operation. The slave's cache contains the most up-to-date copy of the requested address so the response of “DVA” is given on the SResp signal. Simultaneously, the slave drives the first data beat onto SData. Also simultaneously, the slave drives the cacheline state onto SCohState.

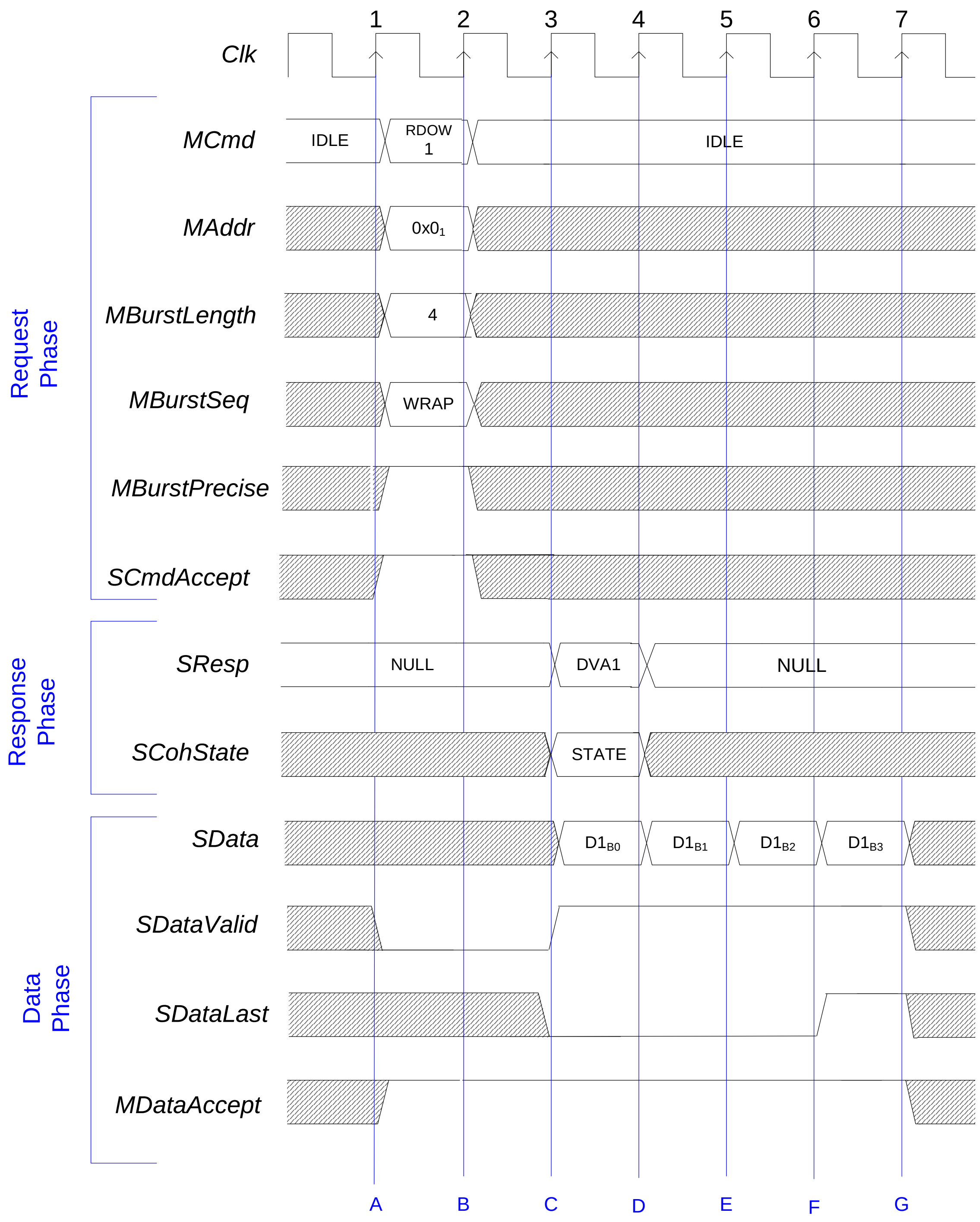
D. The Master recognizes the SResp value to denote valid data and latches the value of the first data beat on SData.

E. Similarly for the 2nd data beat

F. Similarly for the 3rd data beat. The Slave asserts SDataLast to denote it is driving the last data beat.

G. The Master latches the 4th data beat and recognizes SDataLast to complete the transfer.

Figure 52 Transfer with Data Phase, intport_split_tranx=1



When the port parameter intport_split_tranx=1, a separate handshake mechanism is used for the data phase. Two additional signals—SDataValid and MDataAccept—are used for this data handshake.

In this configuration, the signal SResp is no longer used to indicated the presence of valid data on the intervention port, instead the new signal SDataValid is used for that purpose. The signal SResp now only indicates whether the local processor contains a copy of the requested memory location or not and thus is only asserted for a single cycle per transaction.

In the current example, the data transfer is still co-incident with the response phase. In the following examples, the data transfer is delayed after the response phase using these new signals.

Sequence

A. The master starts the request on clock 1 by driving the associated request group signals. The slave asserts SCmdAccept in the same cycle.

B. The slave captures the request signal group values and the request phase completes. The slave does the snoop intervention operation.

C. The slave reports the results of the snoop intervention operation. The slave's cache contains the most up-to-date copy of the requested address so the response of “DVA” is given on the SResp signal. Simultaneously, since the MDataAccept signal is asserted, the slave drives the first data beat onto SData and asserts SDataValid. Also simultaneously, the slave drives the cacheline state onto SCohState.

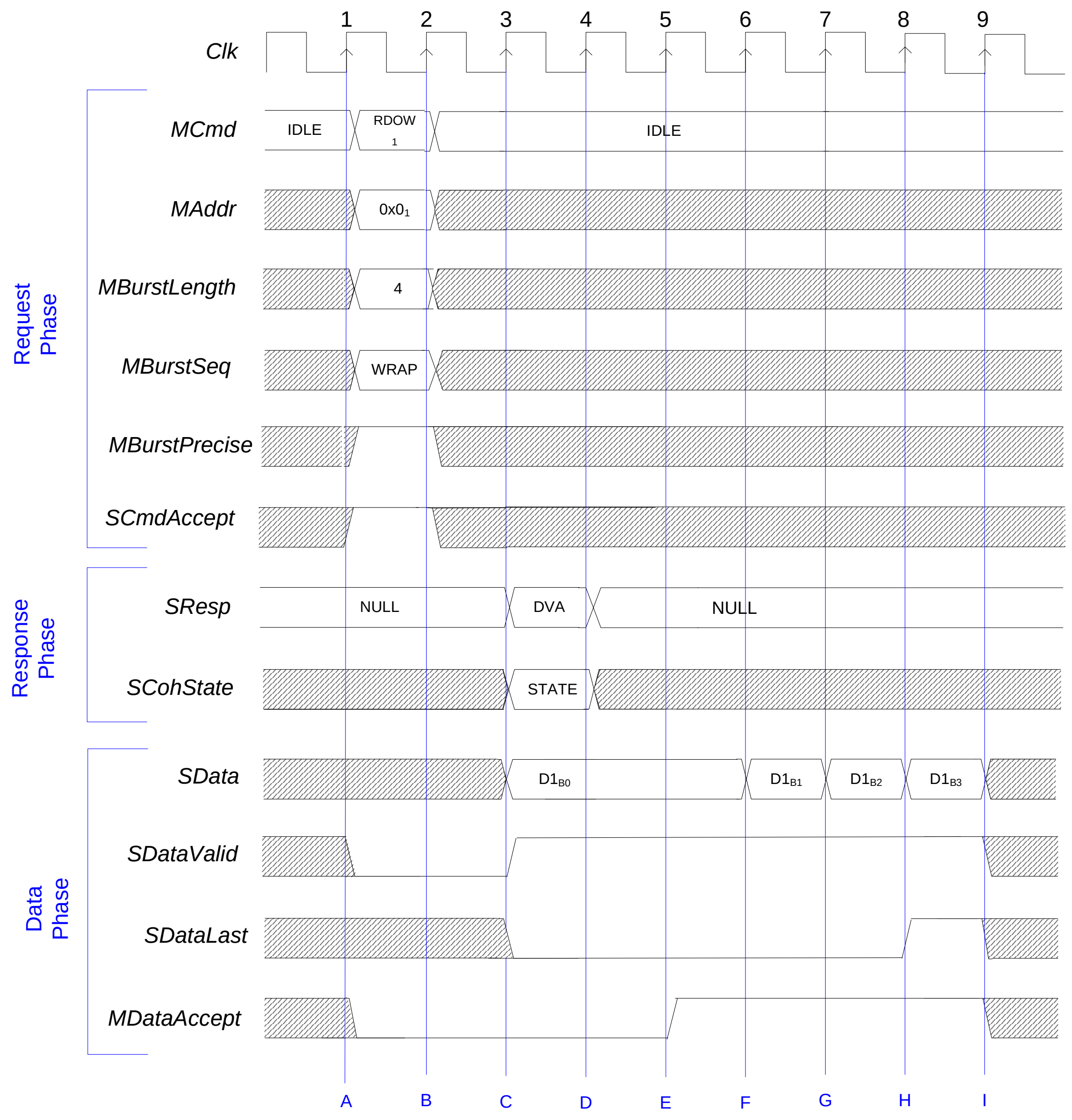
D. The Master recognizes the SResp value and the response phase completes. The Master recognizes the SDataValid signal is asserted and latches the value of the first data beat.

E. Similarly for the 2nd data beat

F. Similarly for the 3rd data beat. The Slave asserts SDataLast to denote it is driving the last data beat.

G. The Master latches the 4th data beat and recognizes SDataLast to complete the transfer.

Figure 53 Transfer with Data Phase delayed by MdataAccept, intport_split_tranx=1



The next diagram shows the use of the MDataAccept signal as a way for the Master to apply flow-control on the slave's data responses.

Sequence

A. The master starts the request on clock 1 by driving the associated request group signals. The slave asserts SCmdAccept in the same cycle.

B. The slave captures the request signal group values and the request phase completes. The slave does the snoop intervention operation.

C. The slave reports the results of the snoop intervention operation. The slave's cache contains the most up-to-date copy of the requested address so the response of “DVA” is given on the SResp signal. Simultaneously, the slave drives the first data beat onto SData and asserts SDataValid. Also simultaneously, the slave drives the cacheline state onto SCohState. However, since MDataAccept is de-asserted, the data phase signal group is held.

D. The Master recognizes the SResp value and the response phase completes. The slave continues holding the data phase signals, awaiting the assertion of MDataAccept.

E. The Master is finally ready to accept the data and asserts MDataAccept.

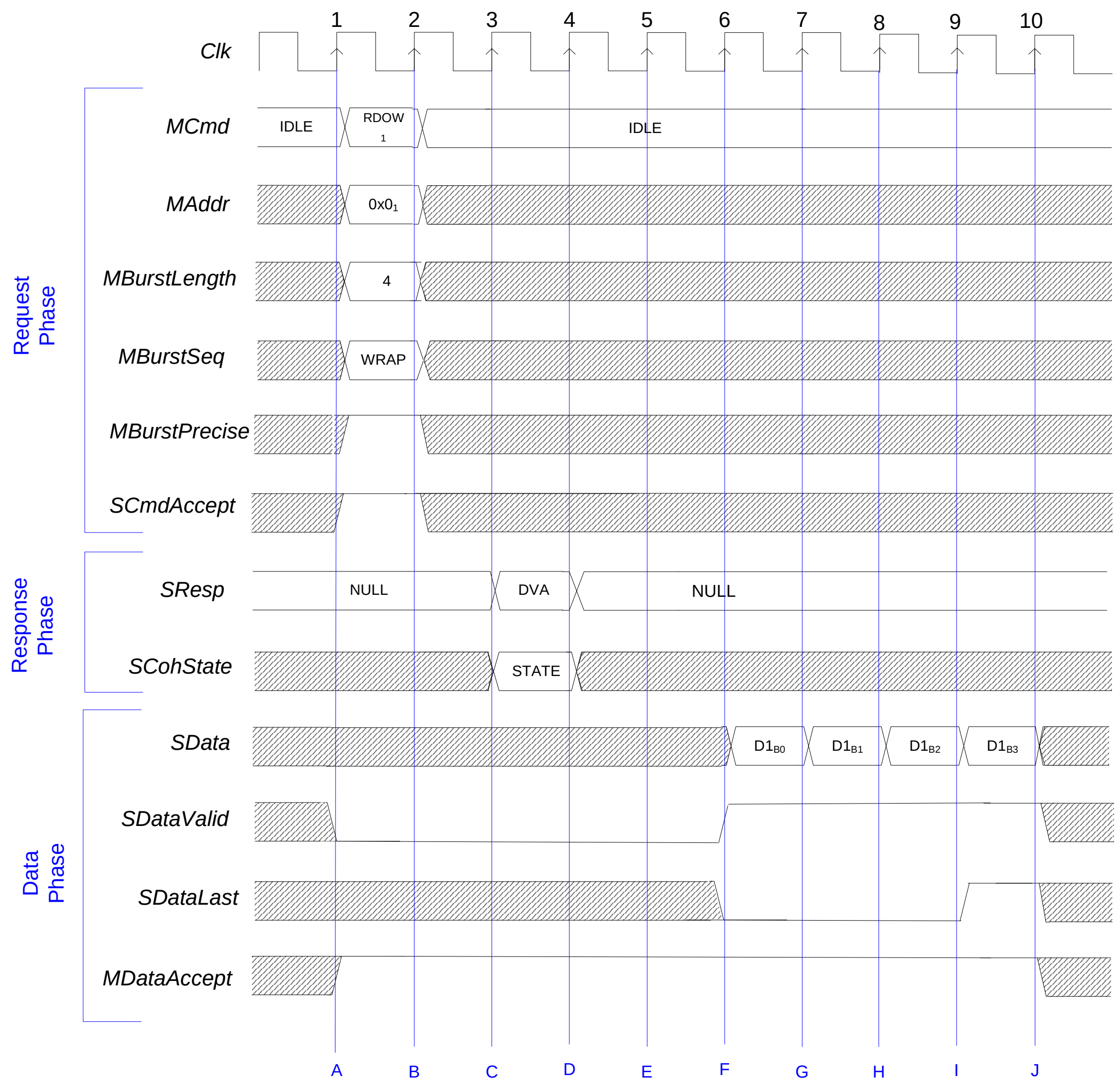
F. The Master latches the data value for the 1st data beat. The Slave recognizes MDataAccept and drives the data value for the 2nd data beat.

G. The Master latches the data value for the 2nd data beat.

H. Similarly for the 3rd data beat. The Slave asserts SDataLast to denote it is driving the last data beat.

I. The Master latches the 4th data beat and recognizes SDataLast to complete the transfer.

Figure 54 Transfer with Data Phase delayed by SDataValid, intport_split_tranx=1



The next diagram shows the use of the SDataValid signal as a way for the slave to separate the Response phase from the Data Transfer Phase. In systems with shadow copies of the cache tags, the response of the cache tags can be delivered earlier than the data.

Sequence

A. The master starts the request on clock 1 by driving the associated request group signals. The slave asserts SCmdAccept in the same cycle.

B. The slave captures the request signal group values and the request phase completes. The slave does the snoop intervention operation.

C. The slave reports the results of the snoop intervention operation. The slave's cache contains the most up-to-date copy of the requested address so the response of “DVA” is given on the SResp signal. Simultaneously, the slave drives the cacheline state onto SCohState.

D. The Master recognizes the SResp value and the response phase completes. Since SDataValid is not asserted, the Master waits for the data values.

E. The Master continues waiting for SDataValid signal to assert.

F. The Slave is finally ready to drive the data and asserts SDataValid and the first data value on SData.

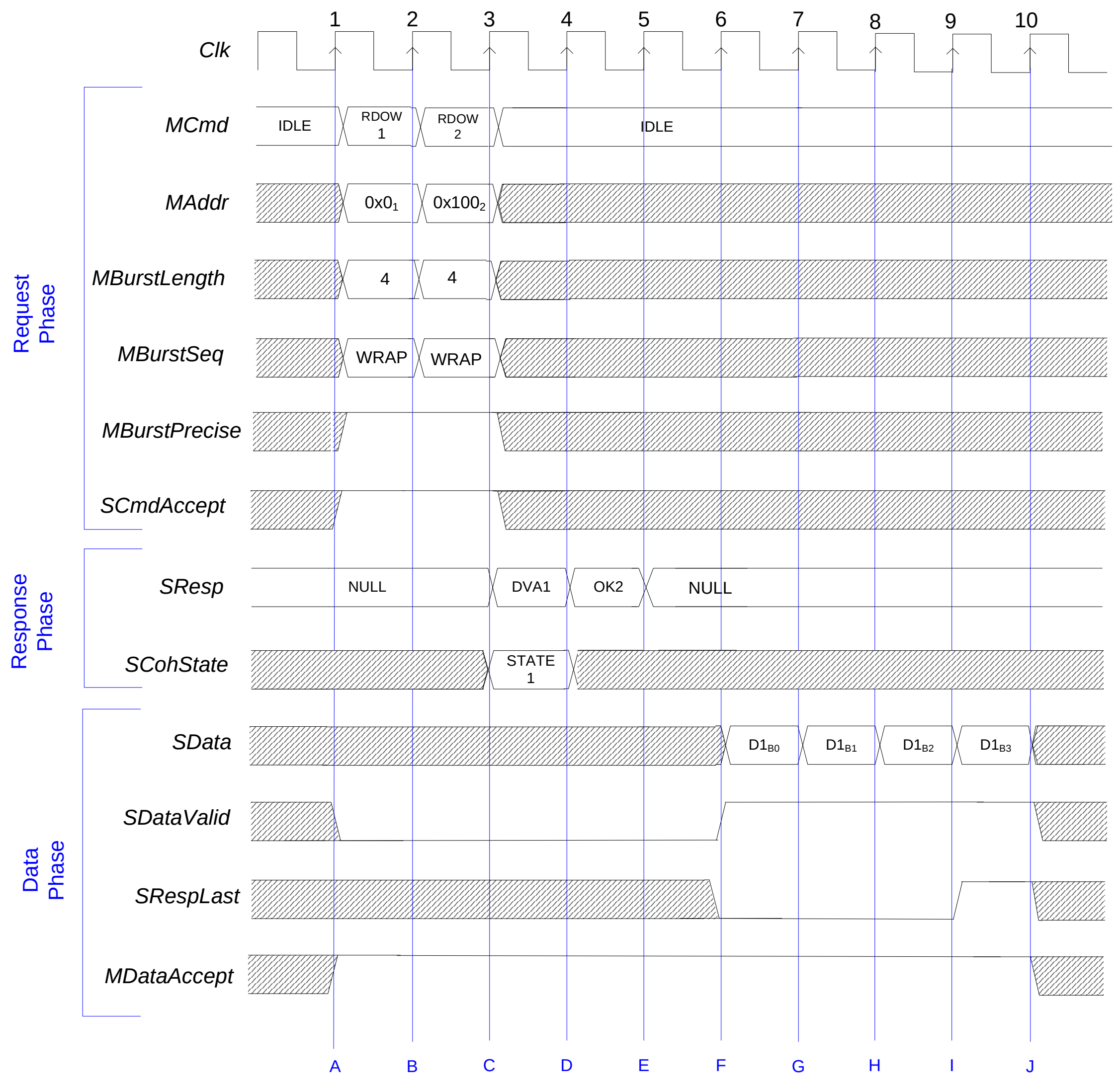
G. The Master latches the data value for the 1st data beat. The Slave drives the data value for the 2nd data beat since MDataAccept was asserted.

H. The Master latches the value for the 2nd data beat. The Slave drives the value for the 3rd data beat.

I. Similarly for the 3rd data beat. The Slave asserts SDataLast to denote it is driving the last data beat.

J. The Master latches the 4th data beat and recognizes SDataLast to complete the transfer.

Figure 55 Overlapped Transactions



The following figure shows overlapped transactions with the response phase for the second transaction happening before the data transfer of the first transaction is completed.

Sequence

A. The master starts the first request on clock 1 by driving the associated request group signals. The slave asserts SCmdAccept in the same cycle.

B. The slave captures the request signal group values and the request phase completes. The slave does the first snoop intervention operation. The master starts the second request by driving new values for the request group signals. The slave accepts the second request by asserting SCmdAccept in the same cycle.

C. The slave reports the results of the first snoop intervention operation. The slave's cache contains the most up-to-date copy of the requested address so the response of “DVA” is given on the SResp signal. Also simultaneously, the slave drives the cacheline state onto SCohState. In the same cycle, the slave does the second snoop intervention operation.

D. The Master recognizes the first SResp value and the first response phase completes. Since SDataValid is not asserted, the Master waits for the data values. The slave reports the results of the second snoop intervention operation. The slave's cache does not contain the second requested address so the response of “OK” is given on the SResp signal.

E. The Master continues waiting for SDataValid signal to assert. The Master recognizes the second SResp value and the second response phase is completed. Since the second transaction does not have a data phase, it is completed.

F. The Slave is finally ready to drive the data for the first request and asserts SDataValid and the first data value on SData.

G. The Master latches the data value for the 1st data beat. The Slave drives the data value for the 2nd data beat since MDataAccept was asserted.

H. The Master latches the value for the 2nd data beat. The Slave drives the value for the 3rd data beat.

I. Similarly for the 3rd data beat. The Slave asserts SDataLast to denote it is driving the last data beat.

J. The Master latches the 4th data beat and recognizes SDataLast to complete the transfer.

12 Developers Guidelines

This chapter collects examples and implementation tips that can help you make effective use of the Open Core Protocol and does not provide any additional specification material. This chapter groups together a variety of topics including discussions of:

1. The basic OCP with an emphasis on signal timing, state machines and OCP subsets
2. Simple extensions that cover byte enables, multiple address spaces and in-band information
3. An overview of burst capabilities
4. The concepts of threading, tagging extensions, and connections
5. OCP features addressing write semantics, synchronization issues, and endianness
6. Sideband signals with an emphasis on reset management and the connection protocol
7. A description of the debug and test interface

12.1 Basic OCP

This section considers the different OCP phases, their relationships, and identifies sensitive timing-related areas and begins with a discussion of support for variable-rate divided clocks. The section includes descriptions of OCP compliant state machines, and also discusses the OCP parameters needed to define simple OCP interfaces.

12.1.1 Divided Clocks

The EnableClk signal allows OCP to provide flexible support for multi-rate systems. When set with the `enableclk` parameter, the EnableClk signal provides a sampling signal that specifies which rising edges of the Clk signal are rising edges of the OCP clock. By driving the appropriate waveforms on EnableClk, the system can control the effective clocking rate of the interface, and frequently the attached cores, without introducing extra outputs from PLLs, or requiring delay matching across multiple clock distribution networks.

When EnableClk is on, the interfaces behave as if the EnableClk signal is not present. All rising edges of Clk are treated as rising edges of the OCP clock allowing the OCP to operate at the Clk frequency. If EnableClk is off, no rising edges of the OCP clock occur, and the OCP clock is effectively stopped.

This feature can be used to reduce dynamic power by idling the attached cores, although the Clk signal may still be active. In normal operation the system drives EnableClk with a periodic signal. For instance, asserting EnableClk for every third Clk cycle causes OCP to operate at one third of the Clk frequency. The system can modify the frequency by changing the repeating pattern on EnableClk.

OCP is fully synchronous (with the exception of reset assertion). All timing paths traversing OCP close in a single OCP clock period. If EnableClk has a maximum duty cycle less than 100%, these timing paths may be constrained as multi-cycle timing paths of the underlying clock domain.

12.1.1.1 OCP Clock Shape

The *OCP Specification* defines synchronous signals with respect to the rising edge of the OCP clock and makes no assertions about the duty cycle of the OCP clock. Since most designs use the rising-edge clocked flip-flops as the storage element in synchronous designs this is usually not an issue. The OCP Clk signal is frequently the output of a PLL or DLL, which tend to output clock signals with near 50% duty cycles.

An OCP interface with a repeating pattern on EnableClk tends to produce pulsed OCP clock waveforms. For instance, with EnableClk asserted every third Clk cycle, the rising edge of the OCP clock is coincident with the rising edge of Clk that samples EnableClk asserted. For most implementations that use this sampling function, the falling edge of the effective (internal) OCP clock is coincident with the next falling edge of Clk. The effective OCP clock is high for one-half of a Clk period every third Clk cycle, yielding an effective duty cycle of 16.7%.

12.1.1.2 Divided Clock Timing

Most design flows treat EnableClk as a standard synchronous signal that could have any value for a cycle. If EnableClk is asserted on consecutive cycles the OCP operates at the full Clk frequency, requiring internal and external timing paths to meet the maximum Clk frequency.

The internal and external timing can be relaxed by recognizing that the EnableClk signal permits a restricted duty cycle (for instance, only high for every third Clk cycle). Taking advantage of this extra timing margin requires careful control over the timing flow, which may include definition and analysis of multi-cycle paths and other challenges. The design flow must assure that the system-side logic that generates EnableClk does not violate the duty cycle assumption. Finally, the timing flow must ensure that no timing issues arise due to the low duty cycle of the effective OCP clock.

12.1.2 Signal Timing

The Open Core Protocol data transfer model allows many different types of existing legacy IP cores to be bridged to the OCP without adding expensive glue logic structures that include address or data storage. As such, it is possible to draw many state machine diagrams that are compliant with the protocol. This section describes some common state machine models that can be used with the OCP, together with guidance on the use of those models.

Dataflow signals in the OCP interface follow the general principle of two-way handshaking. A group of signals is asserted and must be held steady until the corresponding accept signal is asserted. This allows the receiver of a signal to force the sender to hold the signals steady until it has completely processed them. This principle produces implementations with fewer latches for temporary storage.

OCP principles are built around three fundamental decoupled phases: the request phase, the response phase, and the datahandshake phase.

12.1.2.1 Request Phase

Request flow control relies on standard request/accept handshaking signals: MCmd and SCmdAccept. Note that in version 2.0 of this specification, SCmdAccept becomes an optional signal, enabled by the `cmdaccept` parameter. When the signal is not physically present on the interface, it naturally defaults to 1, meaning that a request phase in that case lasts exactly one clock cycle.

The request phase begins when the master drives MCmd to a value other than `Idle`. When MCmd != Idle, MCmd is referred to as asserted. All of the other request phase outputs of the master must become valid during the same clock cycle as MCmd asserted, and be held steady until the request phase ends. The request phase ends when SCmdAccept is sampled asserted (true) by the rising edge of the OCP clock. The slave can assert SCmdAccept in the same cycle that MCmd is asserted, or stay negated for several OCP clock cycles. The latter choice allows the slave to force the master to hold its request phase outputs until the slave can accomplish its access without latching address or data signals.

The slave designer chooses the delay between MCmd asserted and SCmdAccept asserted based on the desired area, timing, and throughput characteristics of the slave.

As the request phase does not begin until MCmd is asserted, SCmdAccept is a “don’t care” while MCmd is not asserted so SCmdAccept can be asserted before MCmd. This allows some area-sensitive, low frequency slaves to tie SCmdAccept asserted, as long as they can always complete their transfer responsibilities in the same cycle that MCmd is asserted. Since an MCmd value of Idle specifies the absence of a valid command, the master can assert MCmd independently of the current setting of SCmdAccept.

The highest throughput that can be achieved with the OCP is one data transfer per OCP clock cycle. High-throughput slaves can approach this rate by providing sufficient internal resources to end most request phases in the same OCP clock cycle that they start. This implies a combinational path from the master’s MCmd output into slave logic, then back out the slaves SCmdAccept output and back into a state machine in the master. If the master has additional requests to present, it can start a new request phase on the next OCP clock cycle. Achieving such high throughput in high-frequency systems requires careful design including cycle time budgeting as described in Section 14.3 on page 316.

12.1.2.2 Response Phase

The response phase begins when the slave drives SResp to a value other than `NULL`. When SResp != NULL, SResp is referred to as asserted. All of the other response phase outputs of the slave must become valid during the same OCP clock cycle as SResp asserted, and be held steady until the response phase ends. The response phase ends when MRespAccept is sampled asserted (true) by the rising edge of the OCP clock; if MRespAccept is not configured into a particular OCP, MRespAccept is assumed to be always asserted (that is, the response phase always ends in the same cycle it begins). If present, the master can assert MRespAccept in the same cycle that MResp is asserted, or it may stay negated for several OCP clock cycles. The latter choice allows the master to force the slave to hold its response phase outputs so the master can finish the transfer without latching the data signals.

Since the response phase does not begin until SResp is asserted, MRespAccept is a “don’t care” while SResp is not asserted so MRespAccept can be asserted before SResp. Since an SResp value of NULL specifies the absence of a valid response, the slave can assert SResp independently of the current setting of MRespAccept.

In high-throughput systems, careful use of MRespAccept can result in significant area savings. To maintain high throughput, systems traditionally introduce *pipelining*, where later requests begin before earlier requests have finished. Pipelining is particularly important to optimize Read accesses to main memory.

The OCP supports pipelining with its basic request/response protocol, since a master is free to start the second request phase as soon as the first has finished (before the first response phase, in many cases). However, without MRespAccept, the master must have sufficient storage resources to receive all of the data it has requested. This is not an issue for some masters, but can be expensive when the master is part of a bridge between subsystems such as computer buses. While the original system initiator may have enough storage, the intermediate bridge may not. If the slave has storage resources (or the ability to flow control data that it is requesting), then allowing the master to de-assert MRespAccept enables the system to operate at high throughput without duplicating worst-case storage requirements across the die.

If a target core natively includes buffering resources that can be used for response flow control at little cost, using MRespAccept can reduce the response buffering requirement in a complex SOC interconnect.

Most simple or low-throughput slave IP cores need not implement MRespAccept. Misuse of MRespAccept makes the slave's job more difficult, because it adds extra conditions (and states) to the slave's logic.

12.1.2.3 Datahandshake Phase

The datahandshake extension allows the de-coupling of a write address from write data. The extension is typically only useful for master and slave devices that require the throughput advantages available through transfer pipelining (particularly memory). When the datahandshake phase is not present in a configured OCP, MData becomes a request phase signal.

The datahandshake phase begins when the master asserts MDataValid. The other datahandshake phase outputs of the master must become valid during the same OCP clock cycle while MDataValid is asserted, and be held steady until the datahandshake phase ends. The datahandshake phase ends when SDataAccept is sampled asserted (true) by the rising edge of the OCP clock. The slave can assert SDataAccept in the same cycle that MDataValid is asserted, or it can stay negated for several OCP clock cycles. The latter choice allows the slave to force the master to hold its datahandshake phase outputs so the slave can accomplish its access without latching data signals.

The datahandshake phase does not begin until MDataValid is asserted. While MDataValid is not asserted, SDataAccept is a "don't care". SDataAccept can be asserted before MDataValid. Since MDataValid not being asserted specifies the absence of valid data, the master can assert MDataValid independently of the current setting of SDataAccept.

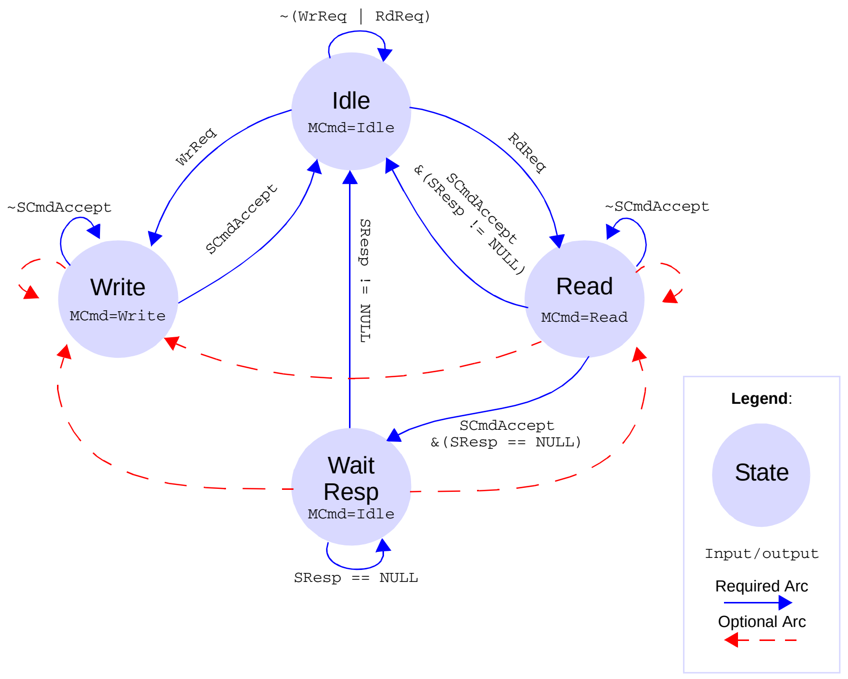
12.1.3 State Machine Examples

The sample state machine implementations in this section use only the features of the basic OCP, request and response phases (the datahandshake phase is not discussed here but can be derived). The examples highlight the flexibility of the basic OCP.

12.1.3.1 Sequential Master

The first example is a medium-throughput, high-frequency master design. To achieve high frequency, the implementation is a completely sequential (that is, Moore state machine) design. Figure 56 shows the state machine associated with the master's OCP.

Figure 56 Sequential Master



Not shown is the internal circuitry of the master. It is assumed that the master provides the state machine with two control wire inputs, WrReq and RdReq, which ask the state machine to initiate a write transfer and a read transfer, respectively. The state machine indicates back to the master the completion of a transfer as it transitions to its Idle state.

Since this is a Moore state machine, the outputs are only a function of the current state. The master cannot begin a request phase by asserting MCmd until it has entered a requesting state (either write or read), based upon the WrReq and RdReq inputs. In the requesting states, the master begins a request phase that continues until the slave asserts SCmdAccept. At this point (this example assumes write posting with no response on writes), a Write command is complete, so the master transitions back to the idle state.

In case of a Read command, the next state is dependent upon whether the slave has begun the response phase or not. Since MRespAccept is not enabled in this example, the response phase always ends in the cycle it begins, so the master may transition back to the idle state if SResp is asserted. If the response phase has not begun, then the next state is wait resp, where the master waits until the response phase begins.

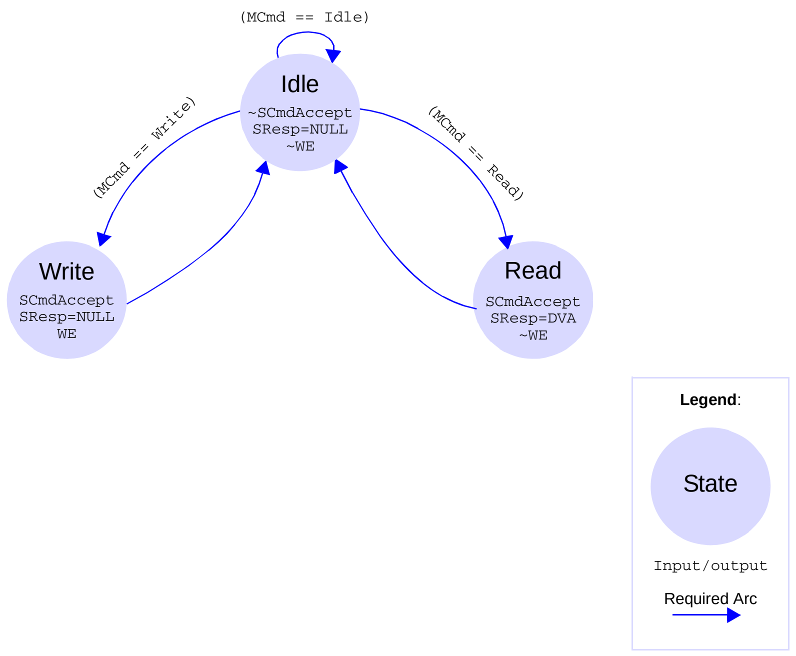
The maximum throughput of this design is one transfer every other cycle, since each transfer ends with at least one cycle of idle. The designer could improve the throughput (given a cooperative slave) by adding the state transitions marked with dashed lines. This would skip the idle state when there are more pending transfers by initiating a new request phase on the cycle after the previous request or response phase. Also, the Moore state machine adds up to a cycle of latency onto the idle to request transition, depending on the arrival time of WrReq and RdReq. This cost is addressed in Section 12.1.3.3 on page 220.

The benefits of this design style include very simple timing, since the master request phase outputs deliver a full cycle of setup time, and minimal logic depth associated with SResp.

12.1.3.2 Sequential Slave

An analogous design point on the slave side is shown in Figure 57. This slave's OCP logic is a Moore state machine. The slave is capable of servicing an OCP read with one OCP clock cycle of latency. On an OCP write, the slave needs the master to hold MData and the associated control fields steady for a complete cycle so the slave's write pulse generator will store the desired data into the desired location. The state machine reacts only to the OCP (the internal operation of the slave never prevents it from servicing a request), and the only non-OCP output of the state machine is the enable (WE) for the write pulse generator.

Figure 57 Sequential OCP Slave



The state machine begins in an idle state, where it de-asserts SCmdAccept and SResp. When it detects the start of a request phase, it transitions to either a read or a write state, based upon MCmd. Since the slave will always complete its task in one cycle, both active states end the request phase (by asserting SCmdAccept), and the read state also begins the response phase. Since MRespAccept is not enabled in this example, the response phase will end in the same cycle it begins. Writes without responses are assumed so SResp is NULL during the write state. Finally, the state machine triggers the write pulse generator in its write state, since the request phase outputs of the master will be held steady until the state machine transitions back to idle.

As is the case for the sequential master shown in Figure 56 on page 218, this state machine limits the maximum throughput of the OCP to one transfer every other cycle. There is no simple way to modify this design to achieve one transfer per cycle, since the underlying slave is only capable of one write every other cycle. With a Moore machine representation, the only way to achieve one transfer per cycle is to assert SCmdAccept unconditionally (since it cannot react to the current request phase signals until the next OCP clock cycle). Solving this performance issue requires a combinational state machine.

Since the outputs depend upon the state machine, the sequential OCP slave has attractive timing properties. It will operate at very high frequencies (providing the internal logic of the slave can run that quickly).

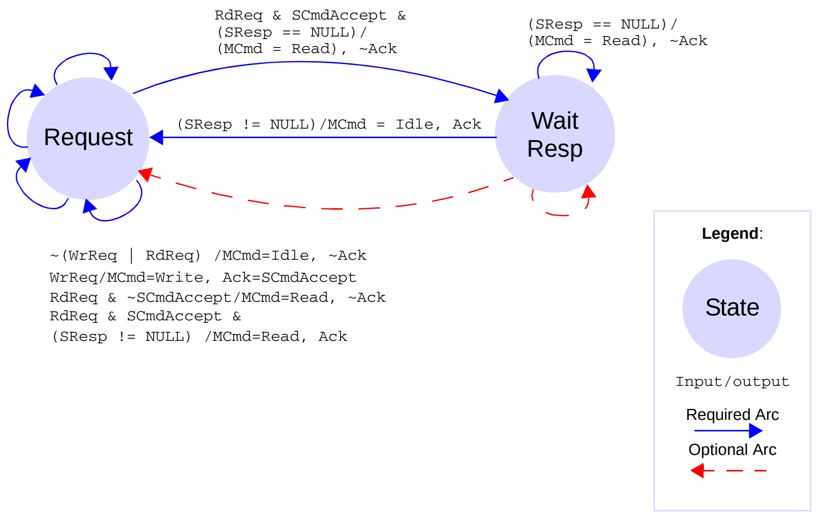
This state machine can be extended to accommodate slaves with internal latency of more than one cycle by adding a counting state between idle and one or both of the active states.

12.1.3.3 Combinational Master

Section 12.1.3.1 on page 217 describes the transfer latency penalty associated with a Moore state machine implementation of an OCP master. An attractive approach to improving overall performance while reducing circuit area is to consider a combinational Mealy state machine representation. Assuming that the internal master logic is clocked from the OCP clock, it is acceptable for the master's outputs to be dependent on both the current state, the internal RdReq and WrReq signals, and the slave's outputs, since all of these are synchronous to the OCP clock. Figure 58 shows a Mealy state machine for the OCP master. The assumptions about the internal master logic are the same as in Section 12.1.3.1 on page 217, except that there is an additional acknowledge (Ack) signal output from the state machine to the internal master logic to indicate the completion of a transfer.

This state machine asserts MCmd in the same cycle that the request arrives from the internal master logic, so transfer latency is improved. In addition, the state machine is simpler than the Moore machine, requiring only two states instead of four. The request state is responsible for beginning and waiting for the end of the request phase. The wait resp state is only used on Read commands where the slave does not assert SResp in the same cycle it asserts SCmdAccept. The arcs described by dashed lines are optional features that allow a transition directly from the end of the response phase into the beginning of the request phase, which can reduce the turn-around delay on multi-cycle Read commands.

Figure 58 Combinational OCP Master



The cost of this approach is in timing. Since the master request phase outputs become valid a combinational logic delay after RdReq and WrReq, there is less setup time available to the slave. Furthermore, if the slave is capable of asserting SCmdAccept on the first cycle of the request phase, then the total path is:

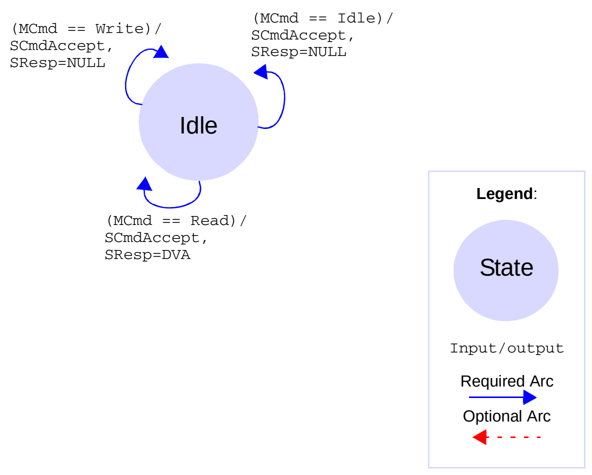
Clk -> (RdReq | WrReq) -> MCmd -> SCmdAccept -> Clk.

To successfully implement this path at high frequency requires careful analysis. The effort is appropriate for highly latency-sensitive masters such as CPU cores. At much lower frequencies, where area is often at a premium, the Mealy OCP master is attractive because it has fewer states and the timing constraints are much simpler to meet. This style of master design is appropriate for both the highest-performance and lowest-performance ends of the spectrum. A Moore state machine implementation may be more appropriate at medium performance.

12.1.3.4 Combinational Slave

Achieving peak OCP data throughput of one transfer per cycle is most commonly implemented using a combinational Mealy state machine implementation. If a slave can satisfy the request phase in the cycle it begins and deliver read data in the same cycle, the Mealy state machine representation is degenerate - there is only one state in the machine. The state machine always asserts SCmdAccept in the first request phase cycle, and asserts SResp in the same cycle for Read commands (assuming no response on writes as in the write posting model).

Figure 59 Combinational OCP Slave



The implementation shown in Figure 59, offers the ideal throughput of one transfer per cycle. This approach typically works best for low-speed I/O devices with FIFOs, medium-frequency but low-latency asynchronous SRAM controllers, and fast register files. This is because the timing path looks like:

Clk -> (master logic) -> MCmd -> (access internal slave resource) -> SResp -> Clk

This path is simplest to make when:

- OCP clock frequency is low
- Internal slave access time is small
- SResp can be determined based only on MCmd assertion (and not other request phase fields nor internal slave conditions)

To satisfy the access time and operating frequency constraints of higher-performance slaves such as main memory controllers, the OCP supports transfer pipelining. From the state machine perspective, pipelining splits the slave state machine into two loosely-coupled machines: one that accepts requests, and one that produces responses. Such machines are particularly useful with the burst extensions to the OCP.

12.1.4 OCP Subsets

It is possible to define simple interfaces - OCP subsets that are frequently required in complex SOC designs. The subsets provide simple interfaces for HW blocks, typically with one-directional, non-addressed, or odd data size capabilities. Since most of the OCP signals can be individually enabled or disabled, a variety of subsets can be defined. For the command set, any OCP command needs to be explicitly declared as supported by the core with at least one command enabled in a subset.

Some sample interfaces are listed in Table 45. For each example the non-default parameter settings are provided. The list of the corresponding OCP signals is provided for reference. Subset variants can further be derived from these examples by enabling various OCP extensions. For guidelines on suggested OCP feature combinations, see Section 15 on page 319.

Table 45 OCP Subsets

Usage	Purpose	Non-default parameters	Signals
Handshake-only OCP	Simple request/acknowledge handshake, that can be used to synchronize two processing modules. Using OCP handshake signals with well-defined timing and semantics allows routing this synchronization process through an interconnect. The OCP command WR is used for requests, other commands are disabled.	read_enable=0, addr=0, mdata=0, sdata=0, resp=0	Clk, MCmd, SCmdAccept
Write-only OCP	Interface for cores that only need to support writes.	read_enable=0, sdata=0, resp=0	Clk, MAddr, MCmd, MData, SCmdAccept
Read-only OCP	Interface for cores that only need to support reads.	write_enable=0, mdata=0	Clk, MAddr, MCmd, SCmdAccept, SData, SResp
FIFO Write-only OCP	Interface to FIFO input.	read_enable=0 addr=0, sdata=0, resp=0	Clk, MCmd, MData, SCmdAccept
FIFO Read-only OCP	Interface to FIFO output.	write_enable=0, addr=0, mdata=0	Clk, MCmd, SCmdAccept, SData, SResp
FIFO OCP	Read and write interface to FIFO.	addr=0	Clk, MCmd, MData, SCmdAccept, SData, SResp

12.2 Simple OCP Extensions

The simple extensions to the OCP signals add support for higher-performance master and slave devices. Extensions include byte enable capability, multiple address spaces, and the addition of in-band socket-specific information to any of the three OCP phases (request, response, and datahandshake).

12.2.1 Byte Enables

Byte enable signals can be driven during the request phase for read or write operations, providing byte addressing capability, or partial OCP word transfer. This capability is enabled by setting the `byteen` parameter to 1.

Even for simpler OCP cores, it is good practice to implement the byte enable extension, making byte addressing available at the chip level with no restrictions for the host processors.

When a datahandshake phase is used (typically for a single request-multiple data burst), bursts must have the same byte enable pattern on all write data words. It is often necessary to send or receive write byte enables with the write data. To provide full byte addressing capability, the MDataByteEn field can be added to the datahandshake phase. This field indicates which bytes within the OCP data write word are part of the current write transfer.

For example, on its master OCP port, a 2D-graphics accelerator can use variable byte enable patterns to achieve good transparent block transfer performance. Any pixel during the memory copy operation that matches the color key value is discarded in the write by de-asserting the corresponding byte enable in the OCP word. Another example is a DRAM controller that, when connected to a x16-DDR device, needs to use the memory data mask lines to perform byte or 16-bit writes. The data mask lines are directly derived from the byte enable pattern.

Unpacking operations inside an interconnect can generate variable byte enable patterns across a burst on the narrower OCP side, even if the pattern is constant on the wider OCP side. Such unpacking operations may also result in a byte enable pattern of all zeros. Therefore, it is mandatory that slave cores fully support 0 as a legal pattern.

An OCP interface can be configured to include partial word transfers by using either the MByteEn field, or the MDataByteEn field, or both.

- If only MByteEn is present, the partial word is specified by this field for both read and write type transfers as part of the request phase. This is the most common case.
- If only MDataByteEn is present, the partial word is specified by this field for write type transfers as part of the datahandshake phase, and partial word reads are not supported.
- If both MByteEn and MDataByteEn are present, MByteEn specifies partial words for read transfers as part of the request phase, and is don’t care for write type transfers. MDataByteEn specifies partial words for write transfers as part of the datahandshake phase, and is don’t care for read type transfers.

12.2.2 Multiple Address Spaces

Logically separate memory regions with unique properties or behavior are often scattered in the system address map. The MAddrSpace signal permits explicit selection of these separate address spaces.

Address spaces typically differentiate a memory space within a core from the configuration register space within that same core, or differentiate several cores into an OCP subsystem including multiple OCP cores that can be mapped at non-contiguous addresses, from the top level system perspective.

Another example of the usage of the addrspace extension is the case of an OCP-to-PCI bridge, since PCI natively supports address spaces for configuration registers, memory spaces and i/o spaces.

12.2.3 In-Band Information

OCP can be extended to communicate information that is not assigned semantics by the OCP protocol. This is true for out-of-band information (flag, control/status signals) and also for in-band information. The designer or the chip level architect can define in-band extensions for the OCP phases.

The fields provided for that purpose are MReqInfo for the request phase, SRespInfo for the response phase, MDataInfo for the request phase or the datahandshake phase, and SDataInfo for the response phase. The presence and width of these fields can be controlled individually.

MReqInfo

Uses for MReqInfo can include user/supervisor storage attributes, cacheable storage attributes, data versus program access, emulation versus application access or any other access-related information, such as dynamic endianness qualification or access permission information.

MReqInfo bits have no assigned meanings but have behavior restrictions. MReqInfo is part of the request phase, so when MCmd is Idle, MReqInfo is a “don’t care.” When MCmd is asserted, MReqInfo must be held steady for the entire request phase. MReqInfo must be constant across an entire transaction, so the value may not change during a burst. This facilitates simple packing and unpacking of data at mismatched master/slave data widths, eliminating the transformation of information.

SRespInfo

Uses for SRespInfo can include error or status information, such as FIFO full or empty indications, or data response endianness information.

SRespInfo bits have no assigned meaning, but have behavior restrictions. SRespInfo is part of the response phase, so when SResp is NULL, SRespInfo is a “don’t care.” When SResp is asserted, SRespInfo must be held steady for the entire response phase. Whenever possible, slaves that generate SRespInfo values should hold them constant for the duration of the transaction, and choose semantics that favor sticky status bits that stay asserted across transactions. This simplifies the design of interconnects and bridges that span OCP interfaces with different configurations. Holding SRespInfo constant improves simple packing and unpacking of data at mismatched data widths. The spanning element may need to break a single transaction into multiple smaller transactions, and so manage the combination of multiple SRespInfo values when the original transaction has fewer responses than the converted ones.

If you implement SRespInfo as specified, your implementation should work in future versions of the specification. If the current implementation does not meet your needs, please contact techsupport@ocpip.org so that the Specification Working Group can investigate how to satisfy your requirements.

MDataInfo and SDataInfo

MDataInfo and SDataInfo have slightly different semantics. While they have no OCP-defined meaning, they may have packing/unpacking implications. MDataInfo and SDataInfo are only valid when their associated phase is asserted (request or datahandshake phase for MDataInfo, response phase for SDataInfo).

Uses for the MDataInfo and SDataInfo fields might include byte data parity in the low-order bits and/or data ECC values in the high-order (non-packable) bits.

The low-order `mdatainfobyte_wdth` bits of MDataInfo are associated with MData[7:0], and so forth for each higher-numbered byte within MData, so that the low-order `mdatainfobyte_wdth`*(`data_wdth`/8) bits of MDataInfo are associated with individual data bytes. Any remaining (upper) bits of MDataInfo cannot be packed or unpacked without further specification, although such bits may be used in cases with matched data width, where no transformation is required.

The difference between MReqInfo and the upper bits of MDataInfo is that only MDataInfo is allowed to change during a transaction. Use SDataInfo for information that may change during a transaction.

A slave should be operable when all bits of MReqInfo and MDataInfo are negated; in other words, any MReqInfo or MDataInfo signals defined by an OCP slave, but not present in the master will normally be negated (driven to logic 0) in the tie off rules. A master should be operable when all bits of SRespInfo and SDataInfo are negated.

12.3 Burst Extensions

A burst is basically a set of related OCP words. Burst framing signals provide a method for linking together otherwise-independent OCP transfers. This mechanism allows various parts of a system to optimize transfer performance using such techniques as SDRAM page-mode operation, burst transfers, and pre-fetching.

Burst support is a key enabler of SOC performance. The burst extension is frequently used in conjunction with pipelined master and slave devices. For a pipelined OCP device, the request phase is *de-coupled* from the response phase - that is, the request phase may begin and end several cycles before the associated response phase begins and ends. As such, it is useful to think of separate, loosely-coupled state machines to support either the master or the slave. Decoupling for pipeline efficiency remains true even if the OCP includes a separate datahandshake phase.

12.3.1 OCP Burst Capabilities

The OCP burst model includes a variety of options permitting close matching of core design requirements.

Exact Burst Lengths and Imprecise Burst Lengths

A burst can be either precise, of known length when issued by the initiator, or imprecise, the burst length is not specified at the start of the burst.

For precise bursts, MBurstLength is driven to the same value throughout the burst, but is really meaningful only during the first request phase. Precise bursts with a length that is a power-of-two can make use of the WRAP and XOR address sequence types.

For imprecise bursts, MBurstLength can assume different values for words in the burst, reflecting a best guess of the burst length. MBurstLength is a hint. Imprecise bursts are completed by a request with an MBurstLength=1, and cannot make use of the WRAP and XOR address sequence types.

Use the precise burst model whenever possible:

- It is compatible with the single request-multiple data model that provides advantages to the SOC in terms of performance and power.
- Since it is deterministic, it simplifies burst conversion. Restricting burst lengths to power-of-two values and using aligned incrementing bursts (by employing the `burst_aligned` parameter) also reduces the interconnect complexity needed to maintain interoperability between cores.

Address Sequences

Using the MBurstSeq field, the OCP burst model supports commonly-used address sequences. Benefits include:

- A simple incrementing scheme for regular memory type accesses
- A constant addressing mode for FIFO oriented targets (typically peripherals)
- Wrapping on power-of-two boundaries
- XOR for processor cache line fill
- A block transfer scheme for 2-dimensional data stored in memory

User-defined sequences can also be defined. They must be carefully documented in the core specification, particularly the rules to be applied when packing or unpacking. The address behavior for the different sequence types is:

INCR
: Each address is incremented by the OCP word size. Used for regular memory type accesses, SDRAM, SRAM, and burst Flash.

STRM
: The address is constant during the burst. Used for streaming data to or from a target, typically a peripheral device including a FIFO interface that is mapped at a constant address within the system.

DFLT1
: User-specified address sequence. Maximum packing is required.

DFLT2
: User-specified address sequence. Packing is not allowed.

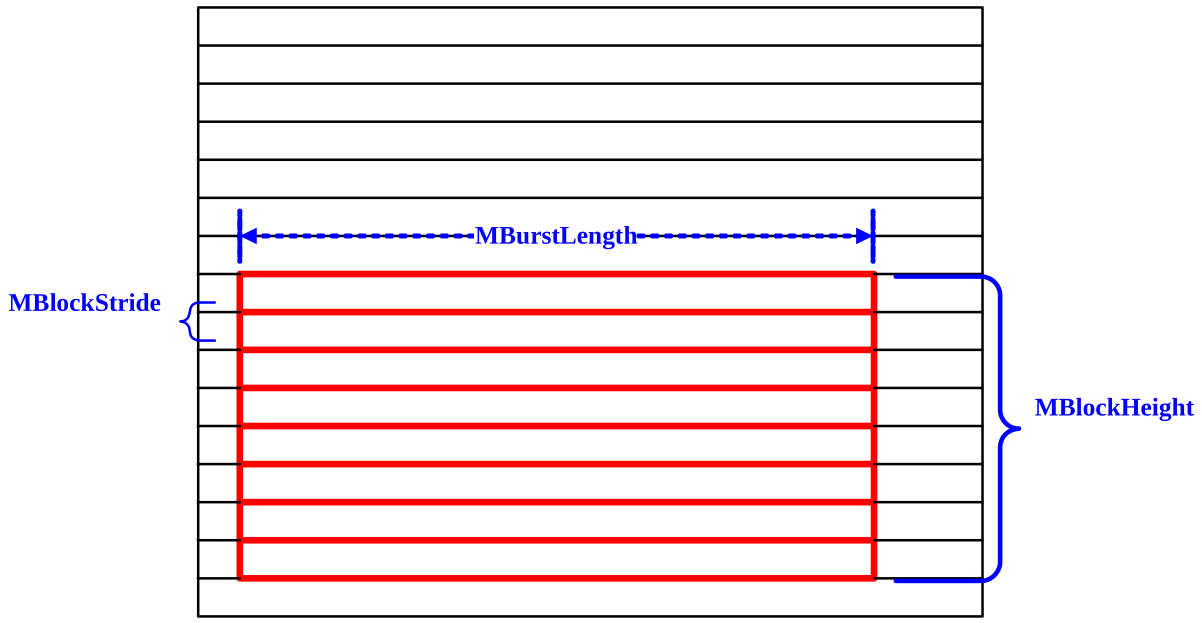
WRAP
: Similar to INCR, except that the address wraps at aligned MBurstLength * OCP word size. This address sequence is typically used for processor cache line fill. Burst length is necessarily a power-of-two, and the burst is aligned on its size.

XOR
: Addr = BurstBaseAddress + (index of first request in burst) ^ (current word number). XOR is used by some processors for critical-word first cache line fill from wide and slow memory systems.

While it does not always deliver the next sequential words as quickly as WRAP, the XOR sequence maps directly into the interleaved burst type supported by many DRAM devices. The XOR sequence is convenient when there are width differences between OCP interfaces, since the sequence is chosen to successively fill power-of-two sized and aligned words of greater width until the burst length is reached.

BLCK
: Describes a sequence of MBlockHeight row transfers, with the starting address MAddr, row-to-row offset MBlockStride (measured from the start of one row to the start of the next row), and rows that are MBurstLength words long in an incrementing row address per word. MBlockHeight and MBlockStride can be considered as don't care for burst sequences other than BLCK. Figure 60 depicts a block transaction representing a 2-dimensional region out of a larger buffer, showing the various parameters.

Figure 60 BLCK Address Sequence



The following example shows how to decompose a BLCK burst into a sequence of INCR bursts.

```
addr = MAddr;
for (row = 0; row < MBlockHeight; row++, addr += MBlockStride) {
    issue INCR using all provided fields (including
    MBurstLength), except use "addr" for MAddr
}
```

UNKN

Indicates that there is no specified relationship between the addresses of different words in the burst. Used to group requests within a burst container, when the address sequence does not match the pre-defined sequences. For example, an initiator can group requests to non-consecutive addresses on the same SDRAM page, so that the target memory bandwidth can be increased.

For targets that have support for some burst sequence, adding support for the UNKN burst sequence can improve the chances of interoperability with other cores and can ease verification since it removes all requirements from the address sequence within a burst.

For single requests, MBurstSeq is allowed to be any value that is legal for that particular OCP interface configuration.

The BLCK, INCR, DFLT1, WRAP, and XOR burst sequences are always considered packing, whereas STRM and DFLT2 sequences are non-packing. Transfers in a packing burst sequence are aggregated / split when translating between OCP interfaces of different widths while transfers in a non-packing sequence are filled / truncated.

The packing behavior of a bridge or interconnect for an UNKN burst sequence is system-dependent. A common policy is to treat an UNKN sequence as packing in a wide-to-narrow OCP request width converter, and as non-packing in a narrow-to-wide OCP request width converter.

Single Request, Multiple Data Bursts for Reads and Writes

A burst model of this type can reduce power consumption, bandwidth congestion on the request path, and buffering requirement at various locations in the system. This model is only applicable for precise bursts, and assumes that the target core can reconstruct the full address sequence using the code provided in the MBurstSeq field.

While the model assumes that the datahandshake extension is on, for those cores that cannot accept the first-data word without the corresponding request, datahandshake can increase design and verification complexity.

For such cores, use the OCP parameter `reqdata_together` to specify the fixed timing relationship between the request and datahandshake phases. When `reqdata_together` is set, each request phase for write-type bursts must be presented and accepted together with the corresponding datahandshake phase. For single request / multiple data write bursts, the request must be presented with the first write data. For multiple request /

multiple data write bursts, the datahandshake phase is locked to the request phase, so this interface configuration only makes sense when both single and multiple request bursts are mixed (that is, `burstsinglereq` is set).

Unit of Atomicity

Use this option when there is a requirement to limit burst interleaving between several threads. Specifying the atomicity allows the master to define a transfer unit that is guaranteed to be handled as an atomic group of requests within the burst, regardless of the total burst length. The master indicates the size of the atomic unit using the MAtomicLength field.

Burst Framing with all Transfer Phases

Without burst framing information, cores and interconnects incorporate counters in their control logic: To limit this extra gate count and complexity, enable end-of-burst information for each phase. Use MReqLast to specify the last request within a burst, SRespLast to specify the last response within a burst, and MDataLast to specify the last write data during the datahandshake phase.

If BLCK burst sequences are enabled, additional framing information can be provided to eliminate additional counters associated with the INCR subsequences that comprise a BLCK burst. To limit extra gate count and complexity, enable end-of-row information for each phase using:

- MReqRowLast to specify the last request in each row for multiple request/ multiple data bursts
- SRespRowLast to specify the last response in each row
- MDataRowLast to specify the last write data in each row

12.3.2 Compatibility with the OCP 1.0 Burst Model

The OCP 2.0 burst model replaces the OCP 1.0 model, providing a super set in terms of available functionality. To maintain interoperability between cores using the OCP 1.0 burst and cores using the OCP 2.0 bursts requires a thin adaptation layer. Guidelines for the wrapping logic are described in this section.

1.0 Master to 2.0 Slave

For converting an OCP 1.0 burst into an OCP 2.0 burst the suggested mapping is:

- MBurstPrecise is available only when the OCP 1.0 `burst_aligned` parameter is set. When set, all incrementing bursts once converted to OCP 2.0 stay precise. Any other OCP 1.0 burst type is mapped to an imprecise burst. When `burst_aligned` is not set, MBurstPrecise is tied off to 0, so all bursts are imprecise.
- MBurstSeq is derived from MBurst as follows:

```
MBurstSeq =    INCR for MBurst {CONT, TWO, FOUR, EIGHT}
               STRM for MBurst {STRM}
               DFLT1 for MBurst {DFLT1}
               DFLT2 for MBurst {DFLT2}
```

The logic must guarantee that MBurstSeq is constant during the whole burst and must continue driving that MBurstSeq when MBurst=LAST is detected.

- The value of MBurstLength is derived as follows:

```
MBurstLength = 8 for MBurst {EIGHT}
               4 for MBurst {FOUR}
               2 for MBurst {TWO, CONT, DFLT1, DFLT2, STRM}
               1 for MBurst {LAST}
```

For precise bursts, MBurstLength is held constant for the entire burst. For imprecise bursts, a new MBurstLength can be derived for each transfer.

- MReqLast is derived from MBurst - it is set when MBurst is LAST.
- SRespLast has no equivalent in OCP1.0, and is discarded by the wrapping logic.
- If required, MDataLast must be generated from a counter or from a queue updated during the request phase.

2.0 Master to 1.0 Slave

For converting an OCP 2.0 burst into an OCP 1.0 burst the suggested mapping is:

- MBurst is derived from MBurstPrecise, MBurstSeq, and MBurstLength, as follows:

```
MBurst =
If MBurstPrecise
if MBurstSeq {INCR}
    EIGHT if MBurstLength >= 8 at start of burst
    FOUR if MBurstLength >= 4 at start of burst
    TWO if MBurstLength >= 2 at start of burst
    load counter with MBurstLength at start of burst,
    decrement counter after every transfer
    subsequent MBurst are generated from counter logic
    LAST when counter==1
else if MBurstSeq {DFLT1, DFLT2, STRM}
    same as MBurstSeq, except when counter==1, must be LAST
else if MBurstSeq {WRAP, XOR, UNKN}
    LAST always: map to consecutive non-burst single transactions


Else if not MBurstPrecise
if MBurstSeq {INCR}
    EIGHT if MBurstLength >= 8
    FOUR if MBurstLength >= 4
    TWO if MBurstLength >= 2
    LAST if MBurstLength == 1
else if MBurstSeq {DFLT1, DFLT2, STRM}
    LAST if MBurstLength == 1
    same as MBurstSeq if MBurstLength != 1
else if MBurstSeq {WRAP, XOR, UNKN}
    LAST always (map to non-burst)
```

- MAtomicLength, MReqLast, and MDataLast have no equivalents in OCP 1.0, and are discarded by the wrapping logic.
- SRespLast must be generated from counter logic.

The logic described above is not suitable if the OCP 2.0 master generates single request / multiple data bursts. In that case, more complex conversion logic is required.

12.4 Tags

Tags are labels that associate requests with responses in order to enable out-of-order return of responses. In the face of different latencies for different requests (for instance, DRAM controllers, or multiple heterogeneous targets), allowing the out-of-order delivery of responses can enable higher performance. Responses are returned in the order they are produced rather than having to buffer and re-order them to satisfy strict response order requirements. Tagged transactions to overlapping addresses have to be committed in order but their responses may be reordered if the transactions have different tag IDs (see Section 4.7.1 on page 57). As is the case for threads, to make use of tags, the master will normally need a buffer.

The tag value generally only has meaning to the master as it is made up by the master and often corresponds to a buffer ID. The tag is carried by the slave and returned with the response. In the case of datahandshake, the master also tags the datahandshake phase with the tag of the corresponding request.

Out-of-order request and response delivery can also be enabled using multiple threads. The major differences between threads and tags are that threads can have independent flow control per thread and have no ordering rules for transfers on different threads. Tags, on the other hand, exist within a single thread so are restricted to a single flow control for all tags. Also, transfers within the same thread still have some (albeit looser) ordering rules when tags are in use. The need for independent flow control requires independent buffering per thread, leading to more complex implementations. Tags enable lower overhead implementations for out-of-order return of responses.

Tags are local to a single OCP interface. In a system with multiple cores connected to a bridge or interconnect, it is the responsibility of the interconnect to translate the tags from one interface to the other so that a target sees different tags for requests issued from different initiators. Target core implementation can facilitate the job of the bridge or interconnect by supporting a large set of tags.

The MTagInOrder and STagInOrder signals allow both tagged and non-tagged (in-order) initiators to talk to a tagged target. Requests issued with MTagInOrder asserted must be completed in-order with respect to other in-order transactions, so an in-order initiator can guarantee that its responses are returned in request order. To retain compatibility with non-tagged initiators, targets that support tags should also support MTagInOrder and STagInOrder.

When MTagInOrder is asserted any MTagID and MDataTagID values are “don’t care”. Similarly, the STagID value is “don’t care” when STagInOrder is asserted. Nonetheless, it is suggested that the slave return whatever tag value the master provided.

Multi-threaded OCP interfaces can also have tags. Each thread’s tags are independent of the other threads’ tags and apply only to the ordering of transfers within that thread. There are no ordering restrictions for transfers on different threads. The number of tags supported by all threads must be uniform, but a master need not make use of all tags on all threads.

12.5 Threads and Connections

Thread extensions add support for concurrency. Without these extensions, there is no way to apply flow control to one set of transfers while allowing another set to proceed. With threading, each transfer is associated with a thread, and independent flow control can be applied to each thread. Additionally, there are no ordering restrictions between transfers associated with different threads. Without threads, ordering is either strict or (if tags are used) somewhat looser.

12.5.1 Threads

The thread capability relies on a thread ID to identify and separate independent transfer streams (threads). The master labels each request with the thread ID that it has assigned to the thread. The thread ID is passed to the slave on MThreadID together with the request (MCmd). When the slave returns a response, it also provides the thread ID (on SThreadID) so the master knows which request is now complete.

The transfers in each thread must remain in-order with respect to each other (as in the basic OCP) or must follow the ordering rules for tagging (if tags are in use), but the order between threads can change between request and response.

The thread capability allows a slave device to optimize its operations. For instance, a DRAM controller could respond to a second read request from a higher-priority initiator before servicing a first request from a lower-priority initiator on a different thread.

As routing congestion and physical effects become increasingly difficult at the back-end stage of the ASIC process, multithreading offers a powerful method of reducing wires. Many functional connections between initiator and target pairs do not require the full bandwidth of an OCP link, so sharing the same wires between several connections, based on functional requirements and floor planning data, is an attractive mechanism to perform gate count versus performance versus wire density trade-offs.

Multi-threaded behavior is most frequently implemented using one state machine per thread. The only added complexity is the arbitration scheme between threads. This is unavoidable, since the entire purpose for building a multi-threaded OCP is to support concurrency, which directly implies contention for any shared resources.

The MDataThreadID signal simplifies the implementation of the datahandshake extension along with threading, by providing the thread ID associated with the current write data transfer. When `datahandshake` is enabled, but `sdatathreadbusy` is disabled, the ordering of the datahandshake phases must exactly match the ordering of the request phases.

The thread busy signals provide status information that allows the master to determine which threads will not accept requests. That information also allows the slave to determine which threads will not accept responses. These signals provide for cooperation between the master and the slave to ensure that requests are not presented on busy threads.

While multithreading support has a cost in terms of gate count (buffers are required on a thread-per-thread basis for maximum efficiency), the protocol can ensure that the multi-threaded interface is non-blocking.

Blocked OCP interfaces introduce a thread dependency. If thread X cannot proceed because the OCP interface is blocked by another thread, Y that is dependent on something downstream that cannot make progress until thread X makes progress, there is a classic circular wait condition that can lead to deadlock.

In the *OCP1.0 Specification*, the semantics of SThreadBusy and MThreadBusy allow these signals to be treated as hints. To guarantee that a multi-threaded interface does not block, both master and slave need to be held to tighter semantics.

OCP 2.2 allows cores to follow exact thread busy semantics. This process enables tighter protocol checking at the interface and guarantees that a multi-threaded OCP interface is non-blocking. Parameters to enable these extensions are `sthreadbusy_exact`, `sdatathreadbusy_exact`, and `mthreadbusy_exact`. There is one parameter for each of the OCP phases, request, datahandshake (assuming separate datahandshake) and response (assuming response flow control). The following conditions are true:

- On an OCP interface that satisfies `sthreadbusy_exact` semantics, the master is not allowed to issue a command to a busy thread.
- On an OCP interface that complies with `sdatathreadbusy_exact` semantics, the master is not allowed to issue write data to a busy thread.
- On an OCP interface that complies with `mthreadbusy_exact` semantics, the slave is not allowed to issue a response to a busy thread.

These rules allow the phase accept signals (SCmdAccept, SDataAccept or MRespAccept) to be tied off to 1 on multi-threaded interfaces for which the corresponding phase handshake satisfies exact thread busy semantics. By eliminating an additional combinational dependency between master and slave, an exact thread busy based handshake can be considered as a substitute for the standard request/accept protocol handshake. For more information, see Section 12.5.1.2 on page 237.

12.5.1.1 Intra-Phase Signal Relationships on a Multithreaded OCP

This section extends the timing discussion of the "Basic OCP" section, to a multithreaded interface. The ordering and timing relationships between the signals within an OCP phase are designed to be flexible. As described in Section , it is legal for SCmdAccept to be driven either combinationally, dependent upon the current cycle's MCmd or independently from MCmd, based on the characteristics of the OCP slave. Some restrictions are required to ensure that independently-created OCP masters and slaves will work together. For instance, the MCmd cannot respond to the current state of SCmdAccept; otherwise, a combinational cycle could occur.

Request Phase

If enabled, a slave's SThreadBusy request phase output should not depend upon the current state of any other OCP signal. SThreadBusy should be stable early enough in the cycle so that the master can factor the current SThreadBusy into the decision of which thread to present a request; that is, all of the master's request phase outputs may depend upon the current SThreadBusy. SThreadBusy is a hint so the master is not required to include a combinational path from SThreadBusy into MCmd, but such paths become unavoidable if the exact semantics apply (`sthreadbusy_exact` = 1). In that case the slave must guarantee that SThreadBusy becomes stable early in the OCP cycle to achieve good frequency performance. A common goal is that SThreadBusy be driven directly from a flip-flop in the slave.

A master's request phase outputs should not depend upon any current slave output other then SThreadBusy. This ensures that there is no combinational loop in the case where the slave's SCmdAccept depends upon the current MCmd.

If a slave's SCmdAccept request phase output is based upon the master's request phase outputs from the current cycle, there is a combinational path from MCmd to SCmdAccept. Otherwise, SCmdAccept may be driven directly from a flip-flop, or based upon some other OCP signals. It is legal for SCmdAccept to be derived from MRespAccept. This case arises when the slave delays SCmdAccept to force the master to hold the request fields for a multi-cycle access. Once read data is available, the slave attempts to return it by asserting SResp. If the OCP has MRespAccept enabled, the slave then must wait for MRespAccept before negating SResp, so it may need to continue to hold off SCmdAccept until it sees MRespAccept asserted.

While the phase relationships of the OCP specification do not allow the response phase to end before the request phase, it is legal for both phases to complete in the same OCP cycle.

The worst-case combinational path for the request phase could be:

```
Clk -> SThreadBusy -> MCmd -> SResp -> MRespAccept -> SCmdAccept -> Clk
```

The preceding path has too much latency at typical clock frequencies, so must be avoided. Fortunately, a multi-threaded slave (with SThreadBusy enabled) is not likely to exhibit non-pipelined read behavior, so this path is unlikely to prove useful. Slave designers need to limit the combinational paths visible at the OCP. By pipelining the read request, the previous path could be:

```
Clk -> SThreadBusy -> MCmd -> Clk
Clk -> SCmdAccept -> Clk     # Slave accepts if pipeline reg empty
Clk -> SResp -> Clk
Clk -> MRespAccept -> Clk    # Master accepts independent of SResp
```

Response Phase

If enabled, a master's MThreadBusy response phase output should not be dependent upon the current state of any other OCP signal. From the perspective of the OCP, MThreadBusy should become stable early enough in the cycle that the slave can factor the current MThreadBusy into the decision on which thread to present a response; that is, all of the slave's response phase outputs may depend upon the current MThreadBusy. If MThreadBusy is simply a hint (in other words `mthreadbusy_exact` = 0) the slave is not required to include a combinational path from MThreadBusy into SResp, but such paths become unavoidable if the exact semantics apply (`mthreadbusy_exact` = 1). In that case the master must guarantee that MThreadBusy becomes stable early in the OCP cycle to achieve good frequency performance. A common goal is that MThreadBusy be driven directly from a flip-flop in the master.

The slave's response phase outputs should not depend upon any current master output other than MThreadBusy. This ensures that there is no combinational loop in the case where the master's MRespAccept depends upon the current SResp.

The master's MRespAccept response phase output may be based upon the slave's response phase outputs from the current cycle or not. If this is true, there is a combinational path from SResp to MRespAccept. Otherwise, MRespAccept can be driven directly from a flip-flop; MRespAccept should not be dependent upon other master outputs.

Datahandshake Phase

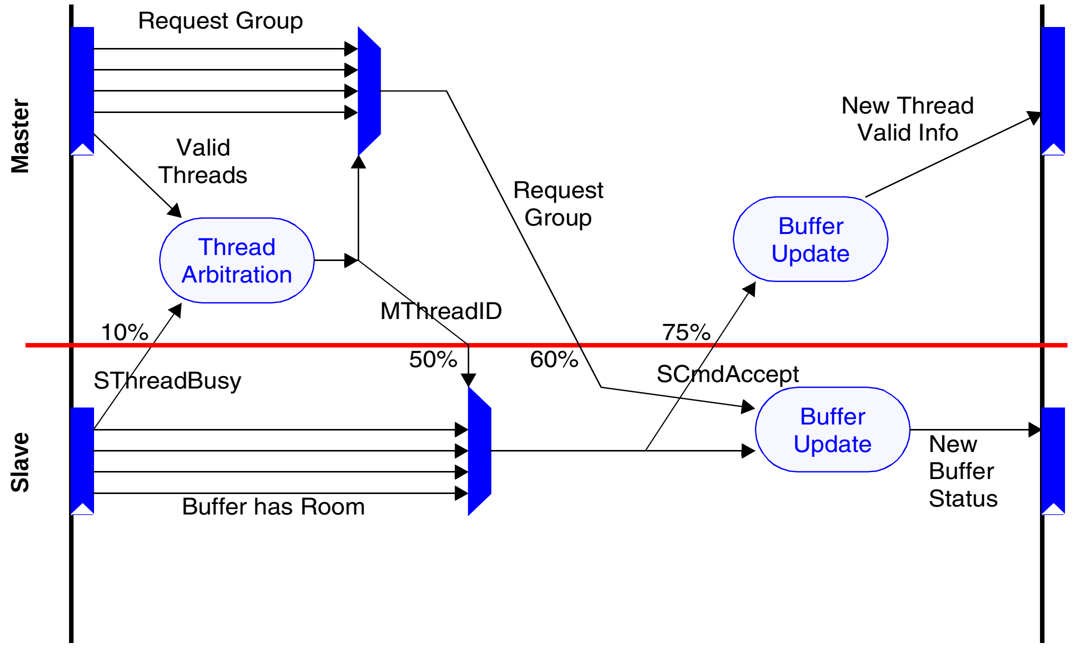
If enabled, a slave's SDataThreadBusy datahandshake phase output should not depend upon the current state of any other OCP signal. SDataThreadBusy should be stable early enough in the cycle so that the master can factor the current SDataThreadBusy into the decision of which thread to present a data; that is, all of the master's data phase outputs may depend upon the current SDataThreadBusy. If SDataThreadBusy is simply a hint (in other words `sdatathreadbusy_exact` = 0) the master is not required to include a combinational path from SDataThreadBusy into MDataValid, but such path becomes unavoidable if the exact semantics apply (sdatathreadbusy_exact = 1). In that case, the slave must guarantee that SDataThreadBusy becomes stable early in the OCP cycle to achieve good frequency performance. A common goal is that SDataThreadBusy be driven directly from a flip-flop in the slave.

The master's datahandshake phase outputs should not depend upon any current slave output other than SThreadBusy. This ensures that there is no combinational loop in the case where the slave's SDataAccept depends upon the current MDataValid. The slave's SDataAccept output may or may not be based upon the master's datahandshake phase outputs from the current cycle. In the former case, there is a combinational path from MDataValid to SDataAccept. In the latter case, SDataAccept should be driven directly from a flip-flop; SDataAccept should not be dependent upon other master outputs.

12.5.1.2 Multi-Threaded OCP Implementation

Figure 61 on page 237 shows the typical implementation of the combinational paths required to make a multi-threaded OCP work within the framework set by Level-2 timing. While the figure shows a request phase, similar logic can be used for the response and datahandshake phases. The top half of the figure shows logic in the master; the bottom half shows logic in the slave. The width of the figure represents a single OCP cycle.

Figure 61 Multithreaded OCP Interface Implementation



12.5.1.3 Slave

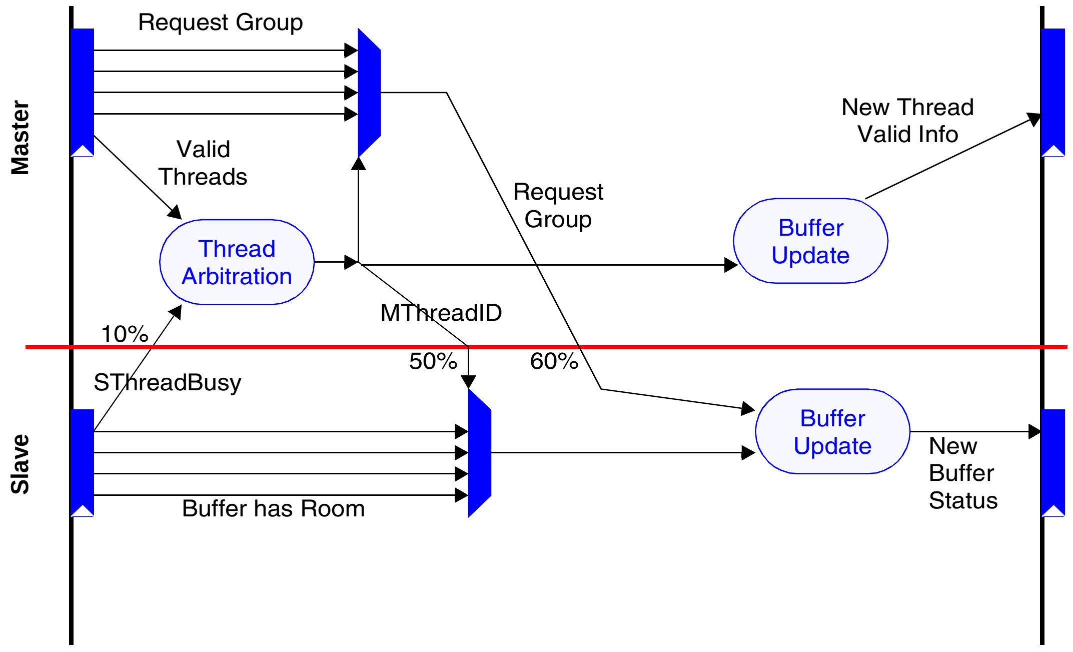
Information about space available on the per-port buffers comes out of a latch and is used to generate SThreadBusy information, which must be generated within the initial 10% of the OCP cycle (as described in Section 14.3 on page 316). These signals are also used to generate SCmdAccept: if a particular port has room, a command on the corresponding thread is accepted. The correct port information is selected through a multiplexer driven by MThreadID at 50% of the clock cycle, making it easy to produce SCmdAccept by 75% of the OCP cycle. When the request group arrives at 60% of the OCP cycle, it is used to update the buffer status, which in turn becomes the SThreadBusy information for the next cycle.

12.5.1.4 Master

The master keeps information on what threads have commands ready to be presented (thread valid bits). When SThreadBusy arrives at 10% of the OCP clock, it is used to mask off requests, that is any thread that has its SThreadBusy signal set is not allowed to participate in arbitration for the OCP. The remaining thread valid bits are fed to thread arbitration, the result is the winning thread identifier, MThreadId. This is passed to the slave at 50% of the OCP clock period. It is also used to select the winning thread's request group, which is then passed to the slave at 60% of the clock period. When the SCmdAccept signal arrives from the slave, it is used to compute the new thread valid bits for the next cycle.

The request phase in Figure 61 assumes a non-exact thread busy model. The exact model shown in Figure 62 is similar, but SCmdAccept is tied off to 1, so any request issued to a non-busy thread is accepted in the same cycle by the slave.

Figure 62 Multithreaded OCP Interface with threadbusy_exact



12.5.2 Connections

In multi-threaded, multi-initiator systems, it is frequently useful to associate a transfer request with a thread operating on a particular initiator. Initiator identification can enable a system to restrict access to shared resources, or foster an error logging mechanism to identify an initiator whose request has created an error in the system. For devices where these concerns are important, the OCP extensions support connections.

Connections are closely related to threads, but can have end-to-end meaning in the system, rather than the local meaning (that is, master to slave) of a thread.

The connection ID and thread ID seem to provide similar functionality, so it is useful to consider why the OCP needs both. A *thread ID* is an identifier of local scope that simply identifies transfers between the master and slave. In contrast, the *connection ID* is an identifier of global scope that identifies transfers between a system initiator and a system target. A thread ID must be small enough (that is, a few bits) to efficiently index tables or state machines within the master and slave. There are usually more connection IDs in the system than any one slave is prepared to simultaneously accept. Using a connection ID in place of a thread ID requires expensive matching logic in the master to associate the returned connection ID (from the slave) with specific requests or buffer entries.

Using a networking analogy, the thread ID is a level-2 (data link layer) concept, whereas the connection ID is more like a level-3 (transport/session layer) concept. Some OCP slaves only operate at level-2, so it doesn't make sense to burden them or their masters with the expense of dealing with level-3 resources. Alternatively, some slaves need the features of level-3 connections, so in this case it makes sense to pass the connection ID through to them.

A connection ID is not usually provided by an initiator core on its OCP interface but is allocated to that particular initiator in the interconnect logic of the system. The connection ID is system-specific, not core-specific; only the system integrator has the global knowledge of the number of initiators instantiated in the application, and what the requirements are in terms of differentiation.

As an exception to that rule, if the global interconnect consists of multiple hierarchical structures, a complete subsystem can be integrated (including another interconnect with multiple embedded initiators). In that case, the OCP interface between the two interconnects should implement the connid extension, so that the end-to-end meaning of that OCP field can be preserved at the system level.

For a target core, the connid extension is included when such features as access control, error logging or similar initiator-related features require initiator identification.

12.6 OCP Specific Features

12.6.1 Write Semantics

As detailed in Section 4.4.2 on page 51, OCP writes support posted and non-posted models. A non-posted write model is preferred whenever the originator of the request must be aware of the completion of its write command, i.e., command commitment. An example is clearing an interrupt in a peripheral module using a write command. In that case the processor must be sure that the interrupt line has been effectively released before it can acknowledge the interrupt service in the chip-level interrupt controller.

The concept of the posting semantics diverges from the concept of responses on writes in the following ways:

- A write with a response could have posted semantics in a system (so that a response is returned immediately) or it could have non-posted semantics (so that a response is returned only after the write is completed at the final target, i.e., the command is committed).
- A write without a response normally has posted semantics and carries forward the *OCP 1.0 Specification* for backward compatibility.
- A write without a response can be assigned non-posted semantics by not accepting the command until the write has completed, but this is not recommended since it de-pipelines the OCP interface. Since posting makes sense at a system level, adopting a delayed-SCmdAccept scheme can only be efficient locally, with no guarantee of the non-posting semantics at the system level.

The `writeresp_enable` parameter controls whether the write-type commands WR and BCST have responses. The write-type commands WRNP and WRC, which are non-posted, always have responses. Table 46 summarizes the behavior with respect to the `writeresp_enable` parameter.

Table 46 Write Command Response Behavior

writeresp_enable	
0	**1**
WR, BCST (without response) WRNP, WRC (with response)	WR, BCST, WRNP, WRC (with response)

Note that in Table 46, WR and WRNP are the general-purpose write commands; WRC is always associated with an RLC command.

Use of the Broadcast command must be limited to a specific category of designs (some interconnect designs may benefit from simultaneous update through distributed registers). It is not expected that standard cores will support the Broadcast command.

By separating whether writes have responses (`writeresp_enable`) from whether the core has control over where the responses are generated (`writenonpost_enable`), the OCP specification provides the following features:

- The simple, posted model remains intact. The simplest cores only implement WR, and need not worry about write responses.
- Cores that can generate or use write responses should enable write responses, providing support for in-band error reporting on write commands. The read and write state machines are duplicated from the standpoint of flow control, producing a simpler design. Such cores would normally only implement the WR command. In this case, the system integrator is in control of where in the write path the write response is generated, allowing a choice of the level of posting based upon performance and coherence trade-offs.
- Cores that can distinguish between performance and coherence (really only CPUs and bridges) can enable WRNP to implement dynamic choice between WR and WRNP. The additional signaling gives the system integrator the dynamic information needed to choose the posting point as the CPU requests. The only practical difference between WR and WRNP at the protocol level is the expected latency between request and response. This permits some embedded CPUs to achieve high performance—particularly as interconnects become pipelined and posting buffers are needed.

12.6.2 Lazy Synchronization

Most processors support semaphores through a read-modify-write type of instruction and swap, test-and-set, etc. Using an OCP interconnect, these instructions are mapped onto a pair of OCP commands. A RDEX command sets a lock to the memory location, followed by a WR (or WRNP) command to release the lock. The system must ensure that no other thread will be granted access to that memory location between the RDEX and the unlocking WR.

Because the Write that clears the lock must immediately follow the ReadEx (on the same thread), only a limited number of operations can be performed by a processor between RDEX and WR. Competing requests to the locked location are not committed until the lock is released. It is highly recommended that this requirement be enforced at the final target with non-blocking flow control for multithreaded applications. Otherwise, for example, if the logic is implemented on the master's side in an interconnect, part of the interconnect could be locked for the duration of the RDEX-WR or RDEX-WRNP pair. This mechanism of using a RDEX-write pair, often referred to as *locked synchronization*, is efficient for handling exclusive accesses to a shared resource, but can result in a significant performance loss when used extensively.

For these reasons, some processors use non-blocking instructions for semaphore handling, breaking the atomicity of the exclusive read/write pair. For the processor, this allows other instructions to be executed by the processor between the read and write accesses. For the system interconnect, it allows requests from other threads to be inserted between the read and write commands. Referred to as lazy synchronization, this mechanism requires read and write semantics, commonly known as LL/SC semantics, for Load-Linked and Store-Conditional.

OCP's support for lazy synchronization uses the ReadLinked, and WriteConditional commands. A single OCP parameter `rdlwrc_enable` is set to 1, to enable the commands. Because some processors might use both semantics (locked and lazy), the OCP interface supports RDEX, RDL, WR(WRNP) and WRC.

The system relies upon the existence of monitor logic, that can be located either in the system interconnect, or in the memory controller. The ReadLinked command sets a reservation in the logic, associating the accessing thread with a particular address. The WriteConditional command, being transmitted on the same thread, is locally transformed into a memory write access only if the reservation is still set when the command is received. As the tagged address is not locked, the tag can be reset by competing traffic directed to the same location from other threads between RDL and WRC.

Consequently, the WRC command expects a response from the monitor logic, reflecting whether the write operation has been performed. To answer that requirement, OCP provides the value, FAIL for the SResp field (meaning that `writeresp_enable` is on if `rdlwrc_enable` is on). WRC is the only OCP command that makes use of the FAIL code, though new commands in future revisions may. FAIL responses are frequently received in a system using lazy synchronization that operates normally. Do not confuse FAIL with SResp=ERR, which effectively signals a system interconnect error or a target error.

Both RDL and WRC commands assume a single transaction model and cannot be used in a burst.

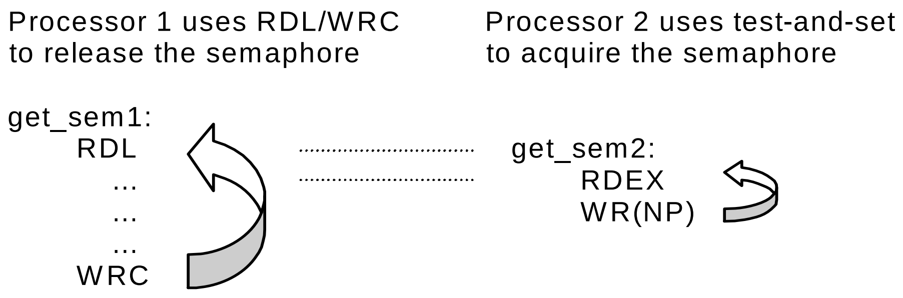
The semantics of lazy synchronization are defined on the previous page. Some specific sequences resulting from the usage of the RDL and WRC semantics are:

- A thread can issue more than one RDL before issuing a WRC, or issue more than one RDL without issuing WRC. Whether the subsequent RDL clears the reservation or sets a new one is implementation-specific, depending on the number of hardware monitors. At least one monitor per thread is required.
- If a thread issues a WR or WRNP command to an address it previously tagged with a RDL command, the write access clears all reservations from other threads for the same address (but not its own reservation).
- If a thread issues a WRC without having issued a RDL, the WRC will fail.
- If a thread issues a RDEX between the RDL and WRC, the RDEX is executed, sets the lock and waits for the corresponding write to clear the lock. RDL-WRC reservations will not be affected by the RDEX. The WR or WRNP that clears the lock, also clears any reservation set by other initiators for the same address (with the same MAddr, MByteEn and MAddrSpace if applicable).

Because competing requests of any type from other threads to a locked (by RDEX) location are blocked from proceeding until the lock is released, a RDEX command being issued between RDL and WRC commands, also blocks the WRC until the WR or WRNP command clearing the lock is issued. This favours RDEX-WR or WRNP sequence over RDL-WRC, in the sense that competing RDEX-WR or WRNP and RDL-WRC sequences will always result in having the RDEX-WR or WRNP sequence win.

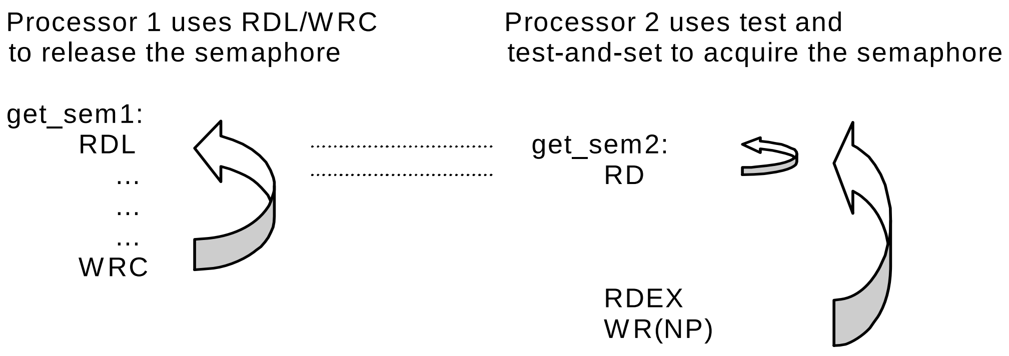
Incorrect use of the two synchronization mechanisms can result in deadlock so for example, the sequence of commands shown in Figure 63 might result in a deadlock. In this example Processor 1 tries to release the semaphore using RDL-WRC commands, Processor 2 tries to acquire the semaphore using RDEX-WR or WRNP commands. The RDEX-WR or WRNP sequence always occurs between the RDL and WRC. Because the WR or WRNP clearing the lock in Processor2 will also clear the reservation for Processor 1, the RDL-WRC sequence will never succeed. Processor 1 will never be able to release the lock or Processor2 to acquire it.

Figure 63 Synchronization Deadlock


Processor 1 uses RDL/WRC to release the semaphore
Processor 2 uses test-and-set to acquire the semaphore
get_sem1:
RDL
...
...
...
WRC
get_sem2:
RDEX
WR(NP)

The deadlock depicted in Figure 63 is a result of bad programming in Processor 2, and is very unlikely to happen in a real application environment. As shown in Figure 64, to achieve forward progress, Processor 2 should read the semaphore value and wait for the semaphore to be free before trying to retrieve it by issuing a RDEX-WR or WRNP.

Figure 64 Correct Synchronization Sequence


Processor 1 uses RDL/WRC to release the semaphore
Processor 2 uses test and test-and-set to acquire the semaphore
get_sem1:
RDL
...
...
...
WRC
get_sem2:
RD
RDEX
WR(NP)

12.6.3 OCP and Endianness

As described in Section 4.5 on page 51, OCP is nearly endian-neutral. While OCP specifies a byte address on MAddr, the address must be aligned to the data width of the interface. Sub-word quantities are specified using one bit for each enabled byte in the transfer on MByteEn or MDataByteEn.

While the bit ordering of OCP fields is consistently described in a little-endian fashion, this is conventional, where even big-endian systems tend to number their bits little-endian. Similarly, the MByteEn numbering seems to imply a little-endian byte ordering, but is simply intended to maintain consistency. For example, MByteEn[m] refers to the byte transferred on MData/SData[(8m+7):8m] (provided m < data width/8), regardless of the effective transfer endianness attributes.

If the master OCP and the slave OCP are the same data width, endianness does not matter. Addresses, data, and byte enables must remain consistent across both interfaces. (There are exceptions, since packed sub-word data objects should be swapped if the endianness does not match. OCP does not carry the required signaling to determine sub-word sizes, so full-word transfers must be assumed.)

Endianness problems arise as soon as one looks to connect a master and slave with different data widths. The narrow side has extra (non-zero) address bits, since its word-aligned addresses do not force as many bits to be zero. The wide side has extra byte lanes to carry its wider words. The association of the extra address bits (narrow side) with the extra byte lanes (wide side) specifies an endianness.

To bridge interfaces that suffer from mismatched data widths, packing and unpacking is required. Data width conversion must make some assumptions about the correspondence between the MAddr least-significant bits and the MByteEn field.

If the association maps the low-order byte lanes to lower addresses, the data width conversion is performed in a little-endian manner. If the association maps the high-order byte lanes to lower addresses, the data width conversion is performed in a big-endian manner. This operation is absolutely not an endianness conversion, but rather an endianness-aware packing or unpacking operation, so that the transaction endianness is preserved across the data width converter.

There is no attempt to perform any endian conversion in hardware. Rather, the goal is to enable interconnects that are essentially endian-neutral, but become endian-adaptive to match the endianness of the attached entities. This implies that the native endianness of an OCP core must be specified. OCP captures that property using the `endian` parameter, which can take four values:

LITTLE
: Qualifies little-endian only cores

BIG
: Qualifies big-endian only cores

BOTH

Qualifies cores that can change endianness:

- Based upon an external input such as a CPU that statically selects its endianness at boot time
- Based upon an internal configuration register such as a DMA engine, that generates OCP read and write requests in accordance with the endianness of the target, as stated by the DMA programmer
- Cores that support dynamic endianness

NEUTRAL

Qualifies cores that have no inherent endianness. Examples are simple memory devices that only work with full OCP-word quantities, or peripheral devices, the endianness of which can be controlled by the software device driver.

While not supported by the standard set of OCP features, it is possible to define a dynamic, endian-aware interconnect using in-band information. By specifying the parameters `reqinfo` (for request packing / unpacking control), `mdatainfo` (for data packing / unpacking control when datahandshake is enabled), and `respinfo` (for response packing / unpacking control), the definition of all these qualifiers becomes platform-specific.

12.6.4 Security

To protect against software and some selective hardware attacks use the OCP interface to create a secure domain across the SOC. The domain might include CPU, memory, I/O etc. that need to be secured using a collection of hardware and software features such as secured interrupts, and memory, or special instructions to access the secure mode of the processor.

The master drives the security level of the request using MReqInfo as a subnet. The master provides initiator identification using MConnID. Table 47 summarizes the relevant parameters.

Table 47 Security Parameters

Parameter	**Value**	**Notes**
reqinfo	1	MReqInfo is required
reqinfo_wdth	Varies	Minimum width is 1
connid	1	To differentiate initiators
connid_wdth	Varies	Minimum width is 1

The security request is defined as a named subnet MSecure within MReqInfo, for example:

```
subnet MReqInfo M:N MSecure, where M is >= N.
```

MSecure Bit Codes

With the exception of bit 0, bits are optional and the encoding is user-defined. Bit 0 of the MSecure field is required. The suggested encoding for the MSecure bits is:

Bit	Value 0	Value 1
0	non-secure	secure
1	user mode	privileged mode
2	data request	instruction request
3	user mode	supervisor mode
4	non-host	host
5	functional	debug

A special error response is not specified. A security error can be signaled with response code ERR.

12.7 Sideband Signals

The sideband signals provide a means of transmitting control-oriented information. Since the signals are rarely performance sensitive, drive all sideband signals stable early in the OCP clock cycle by making the sideband outputs come directly out of core flip-flops. To allow sideband inputs to arrive late in the OCP clock cycle register the inputs immediately on the receiving core.

Cores that fail to implement this conservative timing may require modification to achieve timing convergence.

12.7.1 Reset Handling

Some anomalous events can result from OCP resets. Among the situations to be aware of are the following:

Power-on reset
: At power-on or assertion of any hardware reset, an OCP reset may be asynchronously asserted. Accepting the use of asynchronous resets helps describe the interface behavior.

Asynchronous assertion of a synchronous reset
: Asynchronous assertion of resets in OCP may result in an asynchronous reset being fed to a module expecting a synchronous reset. From a design point of view this discrepancy could lead to a setup or hold violation. The required 16 clock cycles of reset guarantees enough time for recovery on the interface receiver side, allowing it to fall back to a safe functional state.

For simulation and verification, such timing violations represent a major hurdle since they may lead to inconsistent states in the master, slave and monitor interfaces. To address this problem, whenever possible, generate resets in a synchronous manner even if the protocol allows for their asynchronous assertion.

Use of OCP resets as asynchronous

OCP reset requires that a reset signal observe the setup and hold times as defined in the core's timing guidelines for at least 16 rising OCP clock edges after reset assertion, making the signal effectively synchronous except at assertion time. To satisfy this requirement do not connect an input OCP reset signal to the asynchronous clear/set pin of a D-flip-flop involved in an OCP signal logic cone. Failure to comply with this rule may violate the OCP protocol. For instance, a glitch on an OCP reset signal that would be sampled as deasserted could be interpreted as asserted, inadvertently causing the receiver to cancel pending transactions and hang the interface.

12.7.1.1 Dual Reset Signals

Many systems are fully satisfied with a single reset signal applied to both the master and the slave on an OCP interface. Either the master or the slave can drive the reset, or a third entity, such as a chip level reset manager, can provide it to both master and slave.

In some situations, it is more convenient for the master and slave to employ their own reset domains and communicate these internal resets to one another. The OCP interface is unable to communicate until both sides are out of reset since the side still in reset may be driving undetermined values (X) on their OCP outputs and cause problems for the side that is already out of reset. Examples of cases where this might arise are:

- A core with multiple OCP interfaces that are connected to different interconnects, which are each in different reset domains, plus the core has its own internal reset domain.
- Two connected interconnects that both act as initiators of transfers and each with their own reset domain.

Adding a second reset signal to the interface allows each master and slave to have both a reset output and input. The composite reset state for the OCP interface is established as the combination of the two resets, so that either side (or both) asserting reset causes the interface to be in reset. While in reset, the existing rules about the interface state and signal values apply.

Either MReset_n or SReset_n must be present on any OCP interface. Compatibility between different reset configurations of master and slave interfaces is shown in Table 48.

Table 48 Reset Configurations

Master / **Slave**	sreset=1, mreset=0	sreset=0, mreset=1	sreset=1, mreset=1
sreset=0, mreset=1	Dual resets driven by the same 3rd party	Single reset driven by master	Single reset driven by master (SReset_n input tied off to 1)
sreset=1, mreset=0	Single reset driven by slave	Incompatible	Incompatible
sreset=1, mreset=1	Single reset driven by slave (MReset_n input tied off to 1)	Incompatible	Dual resets

The rules describing this table can be stated as follows.

- Either `mreset` or `sreset` or both must be set to 1 for each core.
- The default (and only) tie-off value for MReset_n and SReset_n is 1.
- If `mreset` is set to 1 for the master and `mreset` is set to 0 for the slave, the reset configurations are incompatible.
- If `sreset` is set to 1 for the slave and `sreset` is set to 0 for the master, the reset configurations are incompatible.

Cores with a reset input are always interoperable with any other core. Add a reset output if it is needed by the core or subsystem to assure proper operation. Typically this is because both sides need to know about the reset state of the other side, or because the overall system does not function properly if the core or subsystem is in reset, while the OCP interface is not in reset.

12.7.1.2 Compatibility with OCP 1.0

OCP 1.0 cores that have a reset input or output can be converted to OCP 2 cores by renaming the Reset_n pin in the core's RTL configuration file without touching the actual HDL source of the core. The new name depends on whether the reset is an input or output and whether the core is a master or slave.

In the very unlikely situation of an OCP 1.0 core lacking a reset input or output, the conversion to OCP 2 is achieved by the addition of a dummy reset input pin that is not used inside the core.

12.7.2 Connection Protocol

The increasing importance of minimizing power and energy dissipation in integrated circuits drives designers to use a variety of power management techniques such as slowing or stopping clocks and lowering or switching off supply voltages on sections of the chip. It is therefore frequently desirable to change the power state of the OCP masters and slaves in some sections of a chip without adversely impacting the operation of the rest of the chip. The connection protocol defines a mechanism for OCP masters and slaves to indicate to each other that a power state transition is desired and then to prepare for that transition. The connection signals allow the master and slave to cooperate to cleanly achieve quiescence before putting the interface into a disconnected state where none of the other in-band nor sideband signals are active, except for the OCP clock. Once the interface has been disconnected, the system can safely transition the power state without losing any transactions or sideband events.

While the primary motivation behind the definition and inclusion of the connection protocol is to facilitate power management, there are likely other situations in which OCP masters and slaves may wish to disconnect, so the connection protocol has been defined as a general mechanism.

12.7.2.1 Goals

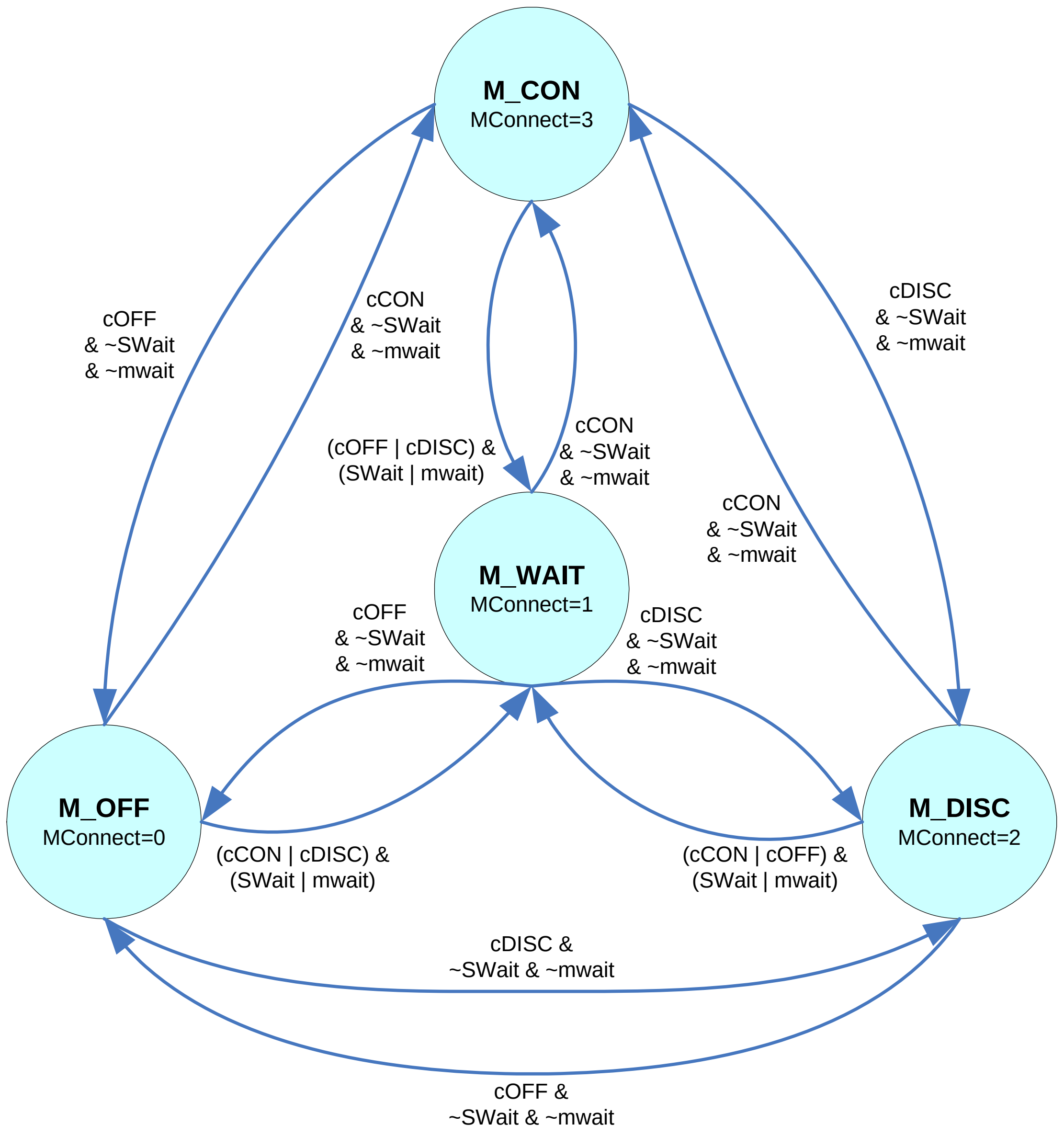
1. The OCP is connected only if both the master and the slave agree on this connected state. Either the master or the slave can independently request disconnection.
2. No restrictions on when either side can change its vote on the connection state.
3. The protocol should assure a clean disconnect. That is, no OCP transactions or sideband signal transitions can be corrupted during the disconnection process, including posted writes.
4. Connection state transitions will be performed by the master, no matter which side requested the transition.
5. The protocol should allow the side that is not requesting the connection state change to delay the state change until it is prepared to safely transition.
6. The protocol should permit the connection state to be determined from interface signals. That is, the protocol should be stateless at the interface.
7. The protocol should permit the system to distinguish between disconnected states initiated by only the slave request vs. those initiated by the master.
8. The protocol should allow an OCP-to-OCP bridge to easily manage its connection protocol responsibilities on both its upstream and downstream interfaces, without requiring that any system logic be needed to separately control the two interfaces.
9. The protocol should support the case of either side being powered down while disconnected by ensuring that a powered down side can safely ignore inputs and provide static protocol-defined default outputs, typically from isolation transceivers.
10. The protocol should ensure inter-operability with cores that do not implement the connection protocol.
11. The protocol should add a minimum of new configurability/complexity to OCP.

12. The protocol should add a minimum of new signals to the interface.

The protocol adds one new parameter to OCP, `connection`. When `connection` is one, four signals are added to the interface: MConnect, SConnect, SWait, and ConnectCap. One of these signals, ConnectCap, is intended to be tied-off at implementation time to support interoperability with cores that do not implement the connection protocol. Tie ConnectCap to logic 0 on cores that implement the protocol to force the connection state and all other signals to remain connected at all times.

The state machine in Figure 65 describes the OCP connection states and the legal transitions between the states. As described in Section 4.3.3.2, the connection state is signaled by the master on the MConnect[1:0] signal. The master and slave are free to request changes to the connection state at any time, but the master is responsible to change the state safely to ensure that no transactions or sideband events are corrupted. Safe transitions are the result of some actions solely under the control of the master, and others which involve interactions with the slave. The system conditions that cause a master or slave to request a state change, and the complete list of actions taken to ensure a safe transition are system specific and outside the scope of the protocol; Figure 65 describes the transition conditions based on both the signals defined by the protocol and internal information from the master (which controls the state machine via its MConnect output).

Figure 65 State Diagram of Connection Protocol



In addition to the connection protocol signals MConnect and SWait, Figure 65 introduces several internal master conditions, which are described in Table 49 below. Note that only one of conditions cCON, cDISC and cOFF can be simultaneously true.

Table 49 Master Condition Description for Figure 65

cCON	Master is prepared to change state to M_CON; SConnect is S_CON
cDISC	Master is prepared to change state to M_DISC; SConnect is S_DISC
cOFF	Master is prepared to change state to M_OFF; SConnect is a "don't care"
mwait	Master has chosen to transition through M_WAIT, independently from SWait

The conditions cCON, cDISC and cOFF should not be true unless the master has determined that the connection state should change to the associated stable state value and the master has completed its role in assuring a safe transition. In particular, the master must assure that all transactions that it has issued have completed and that it will not issue any new transactions before setting either cDISC or cOFF while in M_CON. Note that it is up to the master to determine which additional transactions to present to the slave before responding to SConnect of S_DISC. The master should also attempt to reach quiescence on its sideband signal outputs before leaving M_CON. Furthermore, the cCON, cDISC and cOFF conditions must also ensure that the master remains in each stable state for at least two OCP clock cycles before transitioning, as described in Section 4.3.3.2.

If the slave does not assert SWait, the master is generally free to transition directly between the stable states without entering M_WAIT. However, the master is free to enter M_WAIT independently from SWait, and this situation is modeled in the state machine using the mwait condition. Use of mwait is particularly helpful in linking the connection state machines of two OCP interfaces, such as in a bridging situation. For example, if the operation of the bridge requires the use of SWait on the upstream (bridge slave) interface, then the bridge can use mwait on the downstream interface (where it is master) so that the downstream interface can always copy the upstream interface's transitions to M_WAIT. This avoids the situation where the upstream interface has gone to M_WAIT and the downstream interface must then stay in its current stable state because the bridge cannot know what the next upstream stable state will be.

While all dataflow communication must be complete before the master may leave M_CON, the master cannot generally cause slave sideband communication to become quiescent. Slaves that have sideband outputs other than SReset_n should assert SWait to S_WAIT whenever those outputs may be active. S_WAIT forces the master to transition through M_WAIT, thereby giving the slave the opportunity to end its sideband activity before the master disconnects. Note that some sideband communication protocols may require that the master respond (via its sideband outputs) to slave sideband signals while in M_WAIT. This is why the connection protocol cannot require quiescence on master sideband outputs before leaving M_CON and why M_WAIT entered from M_CON is still considered connected. On the way back to M_CON, sideband communication is not allowed until M_CON is reached.

New transactions may not be issued in the same cycle that the interface transitions to M_CON. This is intended to allow the slave to sample the MConnect signal with the OCP clock, rather than reacting to it in the same cycle it is presented, which could adversely affect the operating frequency of the slave with little apparent benefit.

A slave initiated disconnect (MConnect of M_DISC) may frequently occur when a slave core is powered off, but the master side - which is likely a system interconnect - is still powered. In such situations, the master side may need to manage the situation where a system initiator attempts to communicate with the disconnected slave. For instance, the master could automatically respond to initiator transactions with errors when the connection state is M_DISC. Future versions of this Specification may include specific features to control the upstream behavior for situations involving disconnected slaves.

12.8 Debug and Test Interface

There are three debug and test interface extensions predefined for the OCP: scan, clock control, and IEEE 1149. The scan extension enables internal scan techniques, either in a pre-designed hard-core or end user inserted into a soft-core. Clock control extensions assist in scan testing and debug when the IP core has at least one other clock domain that is not derived from the OCP clock. The IEEE 1149 extension is for interfacing to cores that have an IEEE 1149.1 test access port built-in and accessible. This is primarily the case with cores, such as microprocessors, that were derived from standalone products.

These three extensions along with sideband signals (flags) can yield a highly debuggable and testable IP core and device.

12.8.1 Scan Control

The width and meaning of the Scanctrl field is user-defined. At a minimum this field carries a signal to specify when the device is in scan chain shifting mode. The signal can be used for the scan clock if scan-clock style flip-flops are being used. When this is a multi-bit field, another common signal to carry would be one specifying the scan mode. This signal can be used to put the IP core into any special test mode that is necessary before scanning and application of ATPG vectors can begin.

12.8.2 Clock Control

The clock control test extensions are included to ease the integration of IP cores into full or partial scan test environments and support of debug scan operations in designs that use clock sources other than the OCP clock.

When an external clock source exists (for example, non-Clk derived clock), the ClkByp signal specifies a bypass of the external clock. In that case the TestClk signal usually becomes the clock source. The TestClk toggles in the correct sequence for applying ATPG vectors, stopping the internal clocks, and doing scan dumps as required by the user.

13 Developer's Guidelines: OCP Coherent System Architecture Examples

In this chapter we show several examples of coherent systems, and discuss the implementation of system-level cache coherence using the Coherence Extensions described in Chapters 5 and 6. Please note that multiple implementation choices are permitted using the coherence extensions. For the purpose of illustration, we detail one choice and point out alternatives in a few situations.

13.1 Snoop-Based Coherent Architecture

Figure 66 shows the block diagram of a snoop-based architecture with three coherent masters: Proc0, Proc1, and Proc2. Each master has a main master port, and a intervention slave port.

Proc0 generates a read miss. This in turn generates a CC_RDSH transaction which will eventually place the cache line in Proc0's local cache in the shared (S) state. The actions resulting from this transaction are shown below:

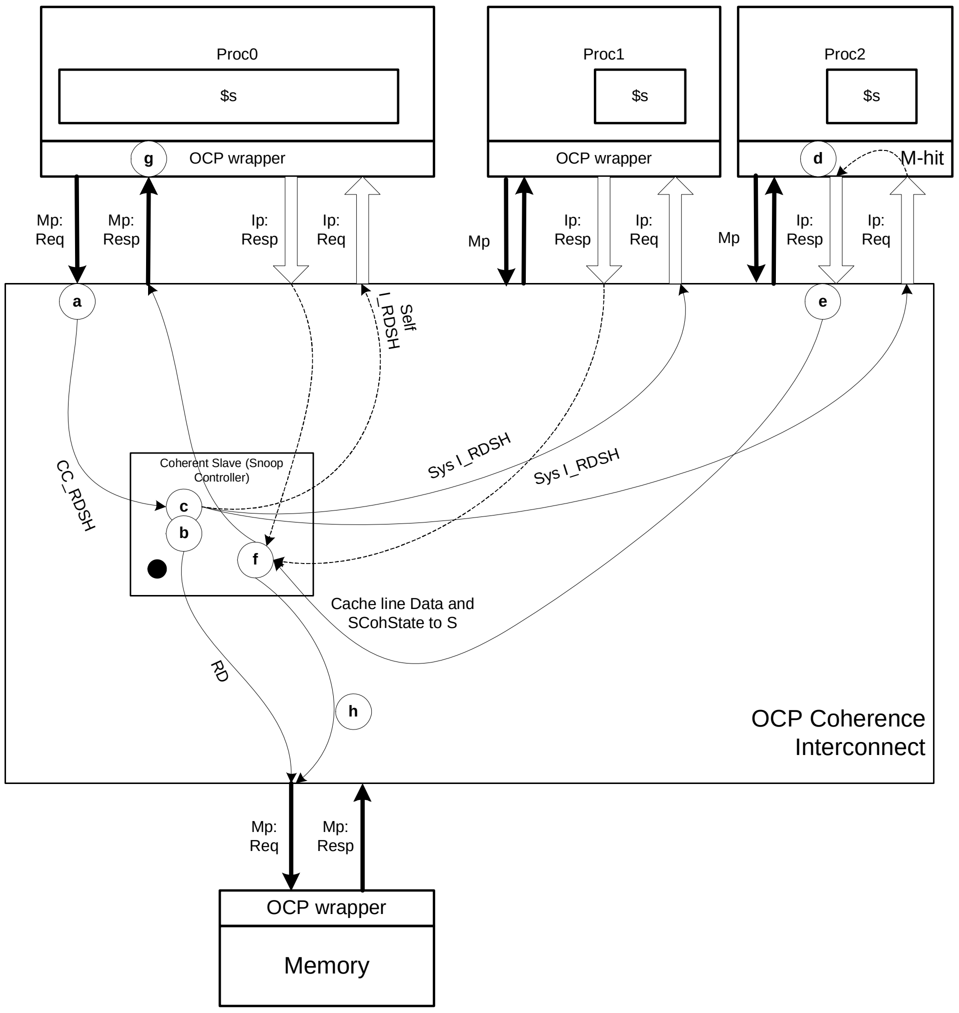
a The requesting coherent master, Proc0, initiates the CC_RDSH request transaction on the main OCP port to gain a sharing ownership on a memory address.

b This request is delivered as a read to the memory slave, Memory (the home of the read address), on its main OCP port.

c Concurrently, the coherence request is turned into a corresponding coherence intervention request (I_RDSH), which is delivered to other coherence masters on their intervention ports. Figure 66 shows three intervention requests sent to Proc0, Proc1, and Proc2, respectively. Note the self-intervention request (Self I_RDSH) sent back to Proc0.

Figure 66 Snoop-Bus-Based OCP Coherence System: Coherent Master and Slave Ports, Communication Flow of a CC_RDSH Transaction

Mp: Req	-- the request channel on the main OCP port
Mp: Resp	-- the response channel on the main OCP port
Ip: Req	-- the request channel on the intervention port
Ip: Resp	-- the response channel on the main OCP port



d Proc2, which has the cache line in M state, relinquishes its ownership on receipt of “Sys I_RDSH” request. Locally, it transitions to the S state.

e An intervention response with data is returned on master Proc2’s intervention port back to the coherent slave.

f The coherent slave sends a response to the Proc0 response main port. The coherent slave also updates the memory by sending a write transaction to it.

g The initiating master receives, on the main port, the coherence response with the latest data and can update its coherence state to S for its cached copy of data from the memory address accordingly.

h The memory is updated with the modified value and has the latest copy. This flow assumes that the memory has not returned the response to the action in step (b). It is assumed that the memory controller will squash the internal read from memory on receipt of the updated contents at this step.

Note that we have shown one implementation choice. Other implementations are permitted using the set of OCP coherence extensions.

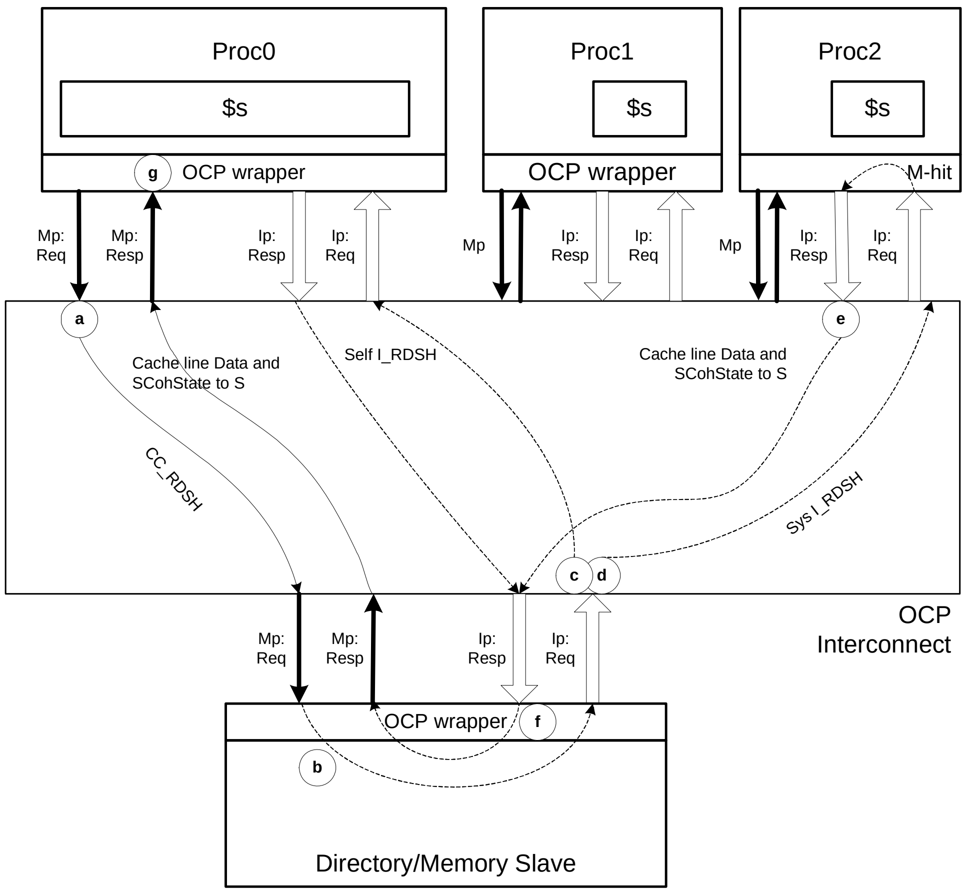
13.2 Directory-Based Coherent System

In a directory-based coherence environment, the same "read memory address and obtain a sharing ownership" transaction (CC_RDSH) can trigger a different kind of communication such as being captured in Figure 67 below:

a The initiating master, Proc0, initiates the transaction and sends a coherence CC_RDSH request, on the main OCP port, to gain a sharing ownership on a memory address. This coherence request is delivered to the Directory/Memory slave (the home of the read address) on its main OCP port.

Figure 67 Directory-Based OCP Coherence System: Communication Flow of a CC_RDSH Transaction

Mp: Req -- the request channel on the main OCP port
Mp: Resp -- the response channel on the main OCP port
Ip: Req -- the request channel on the intervention port
Ip: Resp -- the response channel on the main OCP port



b On the Directory/Memory slave, the directory-based coherence logic lookup indicates that master Proc2 has the latest dirty data.

c A self intervention request (Self I_RDSH) is returned to the initiating master, Proc0, using its intervention port.

d At the same time, a system coherence intervention request (Sys I_RDSH) is sent from the intervention port of the Directory/Memory slave to the intervention port of master Proc2 in order to retrieve the latest dirty data.

e In response master Proc2 relinquishes its exclusive ownership of the memory address, e.g., by changing its cached line's state from M (dirty) to S (shared); and returns an intervention response with data from its intervention port to the Directory/Memory slave.

f Upon receiving the latest dirty data, both the directory and the memory are updated; in addition, the Directory/Memory slave returns the response with data and with coherence state information (of S) back to the initiating master Proc0 on its main port.

g The initiating master Proc0 receives the latest data response and updates its coherence state for the memory address accordingly.

13.2.1 Legal Coherence Dependency

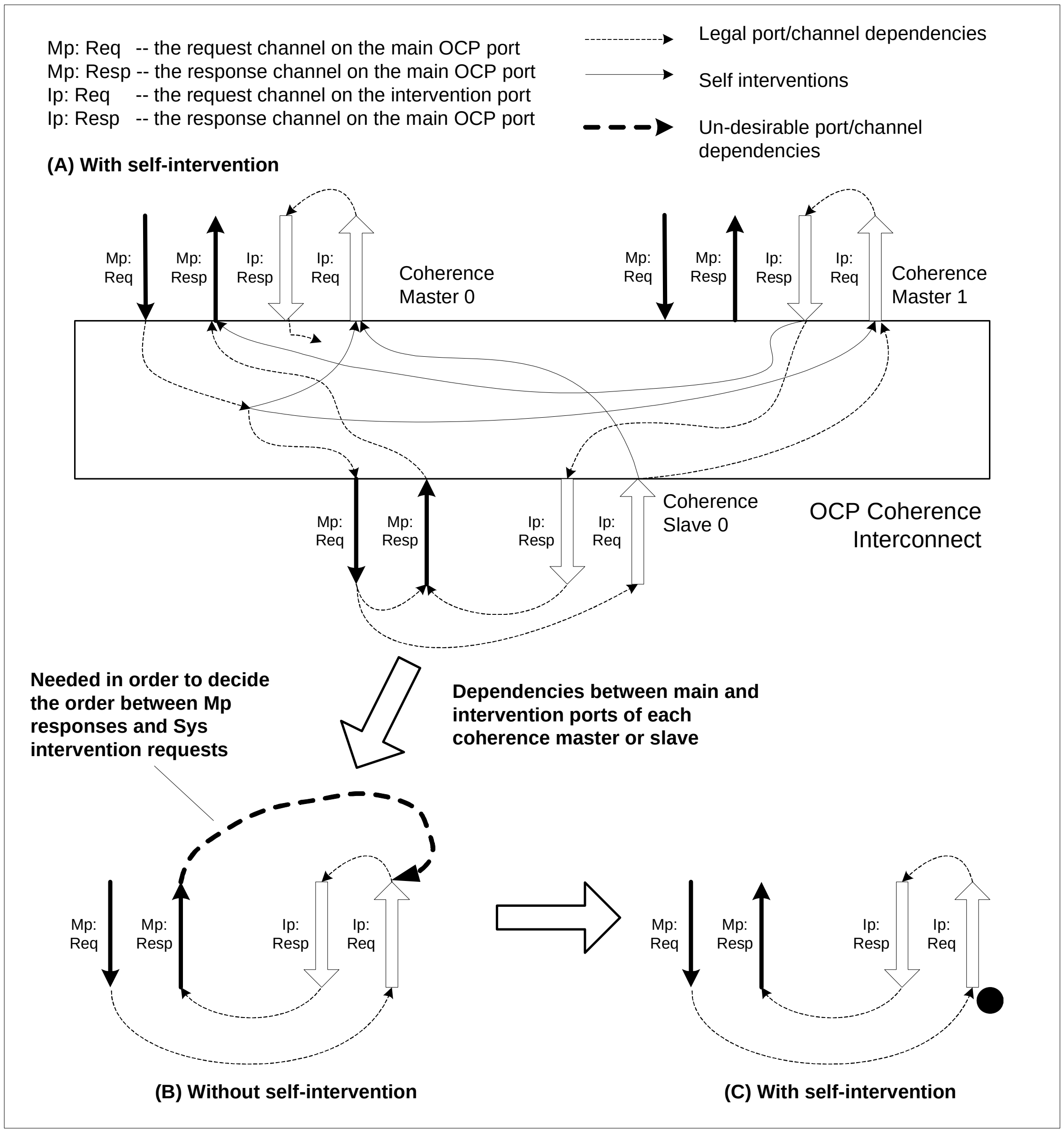
Figure 68-(A) shows legal coherence communication (shown by dependency arrows) between two coherence masters and a coherence slave (home of memory addresses), and between ports on each master or slave.

13.2.1.1 Self-Intervention

Self-intervention is important for the design of the OCP Coherence Extensions. If the self-intervention request is not sent back to the initiating master, the master must enforce the order between the initiating master's main port response to a previously sent main port request and a new conflicting intervention request. To prevent a race between the two coherence masters trying to access the same cache line, the initiating master must block the conflicting intervention request from changing the master's coherence state until after the response to its prior coherent request has been received on its main port. This additional blocking creates a new dependency as shown on top in Figure 68-(B). As shown in the diagram with the top dashed arrow going from MP:resp to Ip:request, this forms a circular dependency and violates the dependence ordering for deadlock avoidance.

Figure 68-(C) is a simplified legal dependency diagram that can be applied to the main and intervention ports of a single coherence master or slave where the self-intervention request sets the ordering of the initiator's request with respect to other coherent intervention requests at the initiating master's intervention port. With self-intervention, there is no circular dependency between ports for a coherence master or slave; therefore, self-intervention is critical to correct OCP coherence operation and to deadlock avoidance.

Figure 68 Ports/Channels Dependencies in an OCP coherence system



Mp: Req -- the request channel on the main OCP port
Mp: Resp -- the response channel on the main OCP port
Ip: Req -- the request channel on the intervention port
Ip: Resp -- the response channel on the main OCP port
Legal port/channel dependencies
Self interventions
Un-desirable port/channel dependencies
(A) With self-intervention
Mp: Req
Mp: Resp
Ip: Resp
Ip: Req
Coherence Master 0
Coherence Master 1
Coherence Slave 0
OCP Coherence Interconnect
Needed in order to decide the order between Mp responses and Sys intervention requests
Dependencies between main and intervention ports of each coherence master or slave
(B) Without self-intervention
(C) With self-intervention

13.3 OCP Coherence Models for Directory-Based Designs

Abstract examples of coherent master, slave, and interconnect are shown in the following sections and are used to express rules on how each coherent master or slave reacts to incoming requests/responses, interacts with coherence states, and produces outgoing requests/responses. Examples for directory-based scheme and snoop-bus-based scheme are demonstrated—assuming both use an invalidating-based coherence protocol. Proper serialization points in each abstract model are also identified in our examples to illustrate how memory coherency can be maintained.

13.3.1 A Directory-Based OCP Coherent System

Figure 69 shows a directory-based OCP coherent system containing two dual-CPU masters with L1/L2 caches, a with-cache coherent DMA master, a DSP master with a non-coherent data cache, a legacy DMA master, a coherent I/O slave, a Memory slave, a Directory module acting as a coherent slave and a legacy master, and a legacy non-coherent I/O slave. This example design is used through out this section to demonstrate how a directory-based coherent system can be implemented using the OCP coherence extensions.

13.3.1.1 OCP Masters

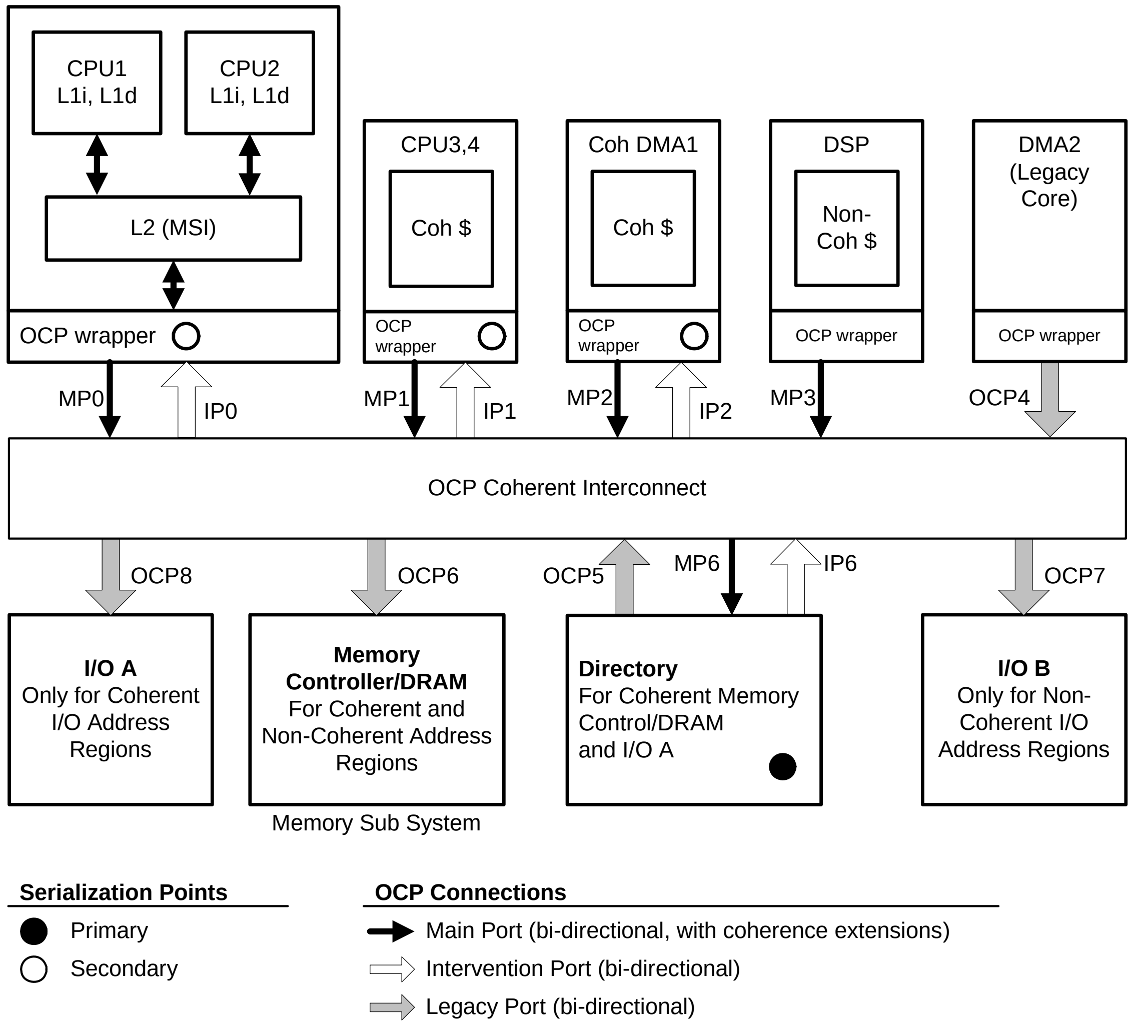
As depicted in Figure 69, CPU1, 2, 3, 4, and Coherent DMA are coherent masters with caches and each uses an outgoing coherent main port and an incoming intervention port to communicate with cores in the system and to maintain system-level (hardware) cache coherence. The DSP master has only an outgoing coherent main port, which allows it to issue both legacy and coherent OCP transactions. However, the internal, non-coherent data cache inside the DSP module needs to be updated explicitly (for instance, by using software) without relying on any hardware intervention requests. The legacy DMA master uses only a legacy OCP port to read data from and write data to the system area using only legacy OCP commands.

As mentioned, the Directory module is also an OCP master, which acts as a proxy and uses a legacy OCP port to communicate with (coherent) slaves, such as I/O "A" and Memory.

13.3.1.2 OCP Slaves

Two I/O slaves exist in the design. The legacy slave, I/O "B," has an incoming legacy OCP port. The coherent slave, I/O "A," which also has an incoming legacy OCP port but can only receive requests from the Directory module. The Directory slave uses an incoming coherent OCP main port and an outgoing intervention port to communicate with the rest of the system in order to maintain system-level (hardware) cache coherence. The Directory slave is responsible for serializing conflicting requests going to the Directory's corresponding home memory (e.g., the Memory module) and I/O module, and keeping the directory state up to date for all requests. The Directory module acts as a proxy master and reads from/writes to the Memory slave and the coherent I/O "A" through the legacy OCP port.

Figure 69 Directory-Based OCP Coherent Design Example


CPU1
L1i, L1d
CPU2
L1i, L1d
L2 (MSI)
OCP wrapper
CPU3,4
Coh $
OCP wrapper
Coh DMA1
Coh $
OCP wrapper
DSP
Non-Coh $
OCP wrapper
DMA2 (Legacy Core)
OCP wrapper
MP0
IP0
MP1
IP1
MP2
IP2
MP3
OCP4
OCP Coherent Interconnect
OCP8
OCP6
OCP5
MP6
IP6
OCP7
I/O A
Only for Coherent I/O Address Regions
Memory Controller/DRAM
For Coherent and Non-Coherent Address Regions
Memory Sub System
Directory
For Coherent Memory Control/DRAM and I/O A
I/O B
Only for Non-Coherent I/O Address Regions
Serialization Points
Primary
Secondary
OCP Connections
Main Port (bi-directional, with coherence extensions)
Intervention Port (bi-directional)
Legacy Port (bi-directional)

13.3.1.3 System Coherence

In this example design, MSI-based write-back cache coherent protocols are used among coherent masters and the Directory module. Cache-to-cache forwarding is not applied in the example. The cache line size used in the system is assumed to be the same for all hardware coherent sub modules. OCP port profiles and information regarding the system-level coherence for the example design are described in the following sub sections.

Multiple serialization points in the example design are also illustrated in Figure 69 where hardware logic is used to enforce serialization among these places.

The high-level memory consistent views applied by the example implementation include:

- Every coherence master/slave participating in the coherence scheme has the same notion of the request sequence.

- Every request reaching the serialization point will be served and completed before any other subsequent conflicting requests.
- In the Directory module, which is the home of cache lines provided by the memory module, conflicting requests are serialized (i.e., served and completed one by one) in the order they arrive to the directory module. This serialization is to prevent possible deadlock causing by two initiators concurrently accessing to the same cache line. Note that a request that hasn't reached the serialization point has no effect on the requestor's or any other coherent master's cache line state.
- Requests are also serialized in the order they arrive to the Directory module for the coherent I/O A module.

13.3.1.4 System-Level Address Map, Coherent Core Identifications, Connectivity

System Address Map

A system address map is applied by the example design where the global address space is divided into address regions and holes as displayed in Figure 70. There are 9 address regions for the design—six of them (Address Region 0, 2, 3, 4, 6, and 8) are coherent address regions and three of them (Address Region 1, 5, and 7) are non-coherent address regions, which under the assumption that only legacy commands can be issued to non-coherent address regions. Also illustrated in the address map Figure 70 is the address-region-to-slave (of proxy slave) assignments:

- Address regions 0, 2, 3, 4, 6, and 8 are coherent address regions associated with the Directory (proxy) slave with home data located at the Memory and I/O A modules.
  - Requests received by the Directory proxy and targeting at address region 2 or 3 can be re-dispatched from the Directory module to the I/O A module.
  - Requests received by the Directory proxy and targeting at address region 0, 4, 6, or 8 can be re-dispatched from the Directory module to the Memory module.
- Address regions 1 and 7 are non coherent address regions associated with the Memory module directly.
- Address region 5 is for the I/O B slave only.

Note that in the example design the OCP main port requests are routed by the interconnect module based on the OCP MAddr field and the address-region-to-slave (or proxy-slave) assignments mentioned above. In addition, care must be taken to ensure that the request re-dispatching, which also uses the MAddr field, between the Directory, I/O A, and Memory modules is handled properly without interfering with the main routing domain between masters and slaves.

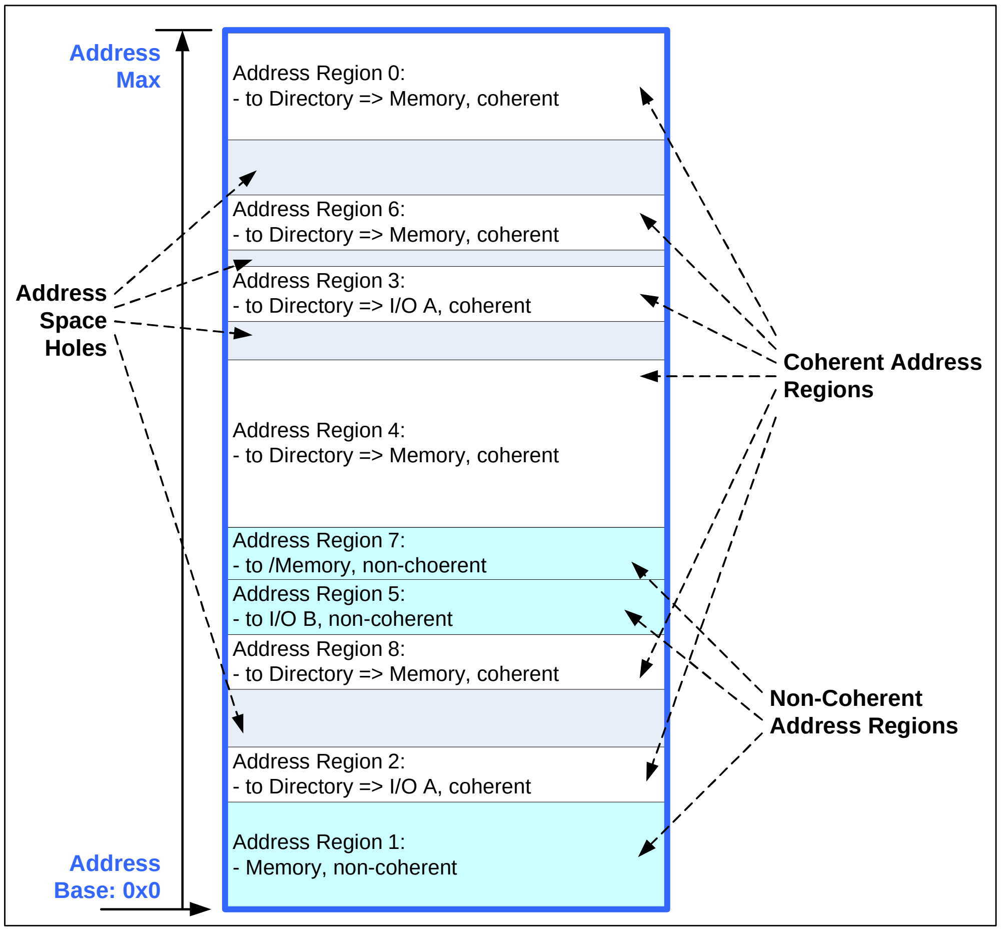
As for intervention port requests, since the MAddr field value is used to indicate which data line is of interest for an intervention transaction, the Coherent Core Identification is used to indicate the destination of each intervention request.

System Connectivity

For simplicity, in this example design, we have the following address region and command issuing limitations (note that these are not requirements imposed by the OCP coherence extensions):

- Legacy slaves can only have non-coherent address regions (such as I/O B)
- Coherent slaves can have both coherent and non-coherent address regions (such as the Memory slave), or have only coherent address regions (such as I/O A).
- OCP coherent and coherence-aware commands can only target coherent address regions.
- OCP legacy commands can be sent to only non-coherent address regions.

Figure 70 System Address Map



The following main port connectivity maps (for domains 1 and 2) between OCP masters and slaves are used for the example design in order to determine how to deliver OCP main port or legacy port requests:

Domain 1:

- CPU1/2 ⇔ Address Region 0, 1, 2, 3, 4, 5, 6, 7, and 8
  We need: a connection to the Directory module for regions 0, 2, 3, 4, 6, and 8; a connection to the Memory module directly for regions 1 and 7; and a connection to the non-coherent I/O B module for region 5.
- CPU3/4 ⇔ Address Region 0, 4, 6, and 8
  We need a connection to the Directory module.
- Coherent DMA1 ⇔ Address Region 2, 3, and 4
  We need a connection to the Directory module.
- DSP ⇔ Address Region 0, 1, 4, 6, 7, and 8
  We need a connection to the Directory module for regions 0, 4, 6, and 8, and a connection to the Memory module for regions 1 and 7.
- Legacy DMA2 ⇔ Address Region 1, 5, and 7
  We need a connection to the Memory module for regions 1 and 7, and a connection to the non-coherent I/O B for region 5.

Domain 2:

- Directory ⇔ Address Region 0, 4, 6, 8 located at the Memory module.
  We need a connection between the Directory proxy and the Memory module
- Directory ⇔ Address Region 2 and 3 located at I/O A.
  We need a connection between the Directory proxy and the I/O A module.

Based on the above main port and legacy port connectivity, plus, how caches and intervention ports are used structurally in the example design, the following intervention port connectivity needs to be established in order to deliver system-level cache interventions:

- Directory ⇔ CPU1/2, CPU3/4, and Coherent DMA1
  We need: an intervention connection between the Directory proxy and CPU1/2; an intervention connection between the Directory proxy and CPU3/4; and an intervention connection between the Directory proxy and Coherent DMA1.

Since no cache-to-cache transfers are allowed, no direct connectivity between coherent OCP masters is needed.

In the example design, the underlying OCP coherent interconnect only needs to provide connections as mentioned in this subsection; therefore, a fully connected network is not needed. Also, in contrast to a snoop-bus-based OCP coherent design and a design with cache-to-cache transfer, there is no need to:

- Copy a main port coherent request and turn it into multiple intervention port requests.
- Convert an intervention port response into a main port coherent response delivering back to the originating coherent master.

Coherent Core Identifications (CohID)

In this example design, we use the MReqInfo signal (3 bits) on the main port and on the intervention port to carry the Coherent Core Identification information for coherent requests and for intervention requests, respectively. The interconnect module utilizes these info fields to route intervention port requests and to return intervention port responses properly. The MReqSelf signal on the intervention port is used to indicate whether an intervention port request is a self-intervention request or a system-intervention one. Table 50 summarizes the coherent core identification numbers, cache protocols, cache line sizes, and OCP data word sizes used by masters and slaves capable of sending and receiving coherent requests and responses in the example.

Table 50 Coherent Core Identifications and Caching Scheme

Master Core Name	**CohID #**	**Cache**		**OCP Word Size**	**Notes**
		Scheme	**Line Size**		
Interconnect or default	0	n/a	n/a	n/a	Reserved
CPU1/2	1	Invalidate MSI	64 bytes	8 bytes	
CPU3/4	2	Invalidate MSI	64 bytes	8 bytes	
Coherent DMA1	3	Invalidate MSI	64 bytes	8 bytes	
DSP	4	Invalidate MSI	64 bytes	8 bytes	DSP has a non system coherent internal cache. And, we need a CohID for it so the coherent slave module can distinguish DSP coherent requests from others.
Directory	5	MSI	64 bytes	8 bytes	May not be needed

Return route for responses corresponding to each main port request can be determined by the interconnect module internally; therefore, no addition main port response signal is needed. Similarly, return route for responses corresponding to each intervention port request can also be determined by the interconnect module internally; thus, the coherent core identifications for the Directory slave may not be needed—this is an implementation choice.

13.3.2 Port Profiles

Profiles for all coherent main port and intervention ports are described to illustrate how the OCP coherence extensions are applied in the example design.

13.3.2.1 Coherent Master Mp0/1/2 and Ip0/1/2 Ports

As mentioned in the previous sections, for main ports Mp0 and 2, we want to be able to send coherent requests, coherence-aware requests, and legacy requests. For main ports Mp1 and Mp3, only coherent and coherence-aware requests can be issued, respectively. For intervention ports Ip0, 1, and 2, all intervention requests can be issued. Also, coherent write data words are delivered on the main port. Plus, on each main or intervention port, only one thread is configured; and, the non-blocking flow control protocol is used.

In the following examples, `writeresp_enable = 1`. Coherent write transactions such as CC_WB have their cache state finalized only on the receipt of SResp on the main port. Alternative implementations where the cache state is finalized earlier, at the time the intervention port response is generated, are also feasible.

OCP Main Port

Table 51 lists important (coherent) parameters and signals used by the main ports Mp0, 1, and 2, for the example design. Parameters not shown here are using their default settings.

Table 51 Parameters/Signals Used by Mp0, Mp1, and Mp2

Parameter	Value	Notes	Corresponding Signals
coh_enable cohcmd_enable cohnc_enable cohwrinv_enable	1 1 1 1	Coherent and legacy commands are issued on Mp0 and 2. Only coherent commands are issued on Mp1.	MCohCmd, extended MCmd
burstsinglereq datahandshake burstlength burstlength_wdth burstprecise burstseq burstseq_incr_enable burstseq_wrap_enable burstseq_xor_enable burstseq_strm_enable	1 1 1 5 1 1 1 1 1 1	Support SRMD bursts and cache line burst sequences (INCR, WRAP, XOR, and more)	MBurstSingleReq, MDataValid, MBurstLength, MBurstPrecise, MBurstSeq
byteen mdatabyteen	1 1	Allow partial cache line transfer for reads and writes	MByteEn, MDataByteEn
writeresp_enable	1	Writes have responses	
intport_writedata	0	Coherent write data words are delivered using the main port	

Parameter	**Value**	**Notes**	**Corresponding Signals**
cohstate_enable	1		SCohState
reqinfo	1	Use to carry coherent ID on bit 0, 1, and 2	MReqInfo
threads sthreadbusy sthreadbusy_exact sdatathreadbusy sdatathreadbusy_exact mthreadbusy mthreadbusy_exact	1 1 1 1 1 1 1	Use single-threaded non-blocking protocol	SThreadBusy, SDataThreadBusy, MThreadBusy

OCP Intervention Port

Table 52 lists important coherent parameters used by the intervention port 0, 1, and 2, for the example design. Parameters not shown here are using their default settings.

Table 52 Parameters Used by Ip0, Ip1, and Ip2

Parameter	**Value**	**Notes**	**Corresponding Signals**
None		Required signals	MReqSelf, MCmd, SCohState
mdata datahandshake	0 0	No write data	
reqinfo	1	Use to carry coherent ID on bit 0, 1, and 2	MReqInfo
intport_split_tranx	1	Allow response datahandshake protocol	SDataValid
burstsinglereq datahandshake burstlength burstlength_wdth burstprecise burstseq burstseq_incr_enable burstseq_wrap_enable burstseq_xor_enable	1 1 1 5 1 1 1 1 1	Support SRMD bursts and cache line burst sequences (INCR, WRAP, and XOR)	MBurstSingleReq, MDataValid, MBurstLength, MBurstPrecise, MBurstSeq

Parameter	Value	Notes	Corresponding Signals
byteen mdatabyteen	0 0	Read for ownership always return the full cache line	
writeresp_enable	1	Writes have responses	
threads sthreadbusy sthreadbusy_exact mthreadbusy mthreadbusy_exact mdatathreadbusy mdatathreadbusy_exact	1 1 1 1 1 1 1	Use single-threaded non-blocking protocol	SThreadBusy, MThreadBusy, MDataThreadBusy (intport_split_tranx == 1)

13.3.2.2 Coherence-Aware Master Mp3 Port

The Mp3 port is used to send coherence-aware requests and legacy requests. Also, coherent write data words must be delivered on the main port. Plus, only one thread is configured; and, the non-blocking flow control protocol is used. Parameters not shown here use default settings.

Table 53 Parameters/Signals Used by Mp3

Parameter	Value	Notes	Corresponding Signals
coh_enable cohcmd_enable cohnc_enable cohwrinv_enable	1 1 1 0	coherence-aware and legacy commands are issued on Mp3	MCohCmd, extended MCmd
burstsinglereq datahandshake burstlength burstlength_wdth burstprecise burstseq burstseq_incr_enable burstseq_wrap_enable burstseq_xor_enable burstseq_strm_enable	1 1 1 5 1 1 1 1 1 1	Support SRMD bursts and burst sequences INCR, WRAP, XOR, and more.	MBurstSingleReq, MDataValid, MBurstLength, MBurstPrecise, MBurstSeq
byteen mdatabyteen	1 1	Allow partial transfer for reads and writes	MByteEn, MDataByteEn
writeresp_enable	1	Writes have responses	
intport_writedata	0	Must be 0 since there is no intervention port for this master	

Parameter	Value	Notes	Corresponding Signals
cohstate_enable	0	Only issuing coherence-aware commands	NO SCohState
reqinfo	1	Use to carry coherent ID on bit 0, 1, and 2	MReqInfo
threads sthreadbusy sthreadbusy_exact sdatathreadbusy sdatathreadbusy_exact mthreadbusy mthreadbusy_exact	1 1 1 1 1 1 1	Use single-threaded non-blocking protocol	SThreadBusy, SDataThreadBusy, MThreadBusy

13.3.2.3 Legacy Master OCP4 and OCP5 Ports

A regular single-threaded, non-blocking OCP connection is configured for OCP4 and for OCP5 here (e.g., coh_enable = 0); and it allows SRMD bursts, precise bursts of INCR, STRM, and UNKN sequences, byte enable signals, and writes with responses.

13.3.2.4 Coherent Slave Mp6 and Ip6 Ports

As mentioned in the previous sub sections, for main ports Mp6, we want to be able to carry coherent requests, coherence-aware requests, and legacy requests. For intervention ports Ip6, all intervention requests can be issued. Since these ports are used by slaves, the coherent write data words must be delivered on the main port. Moreover, on each main or intervention port, only one thread is configured; and, the non-blocking flow control protocol is used.

OCP Main Port

Table 54 lists the most important (coherent) parameters and signals used by main ports 5 and 6 in the example design. Parameters not shown here are assumed to use their default settings.

Table 54 Parameters Used by Mp5 and Mp6

Parameter	Value	Notes	Corresponding Signals
coh_enable cohcmd_enable cohnc_enable cohwrinv_enable	1 1 1 1	Coherent, coherence-aware, and legacy commands can be issued on Mp6.	MCohCmd, extended MCmd
burstsinglereq datahandshake burstlength burstlength_wdth burstprecise burstseq burstseq_incr_enable burstseq_wrap_enable burstseq_xor_enable burstseq_strm_enable burstseq_unkn_enable	1 1 1 5 1 1 1 1 1 1 1	Support SRMD bursts and cache line burst sequences (INCR, WRAP, XOR, and more)	MBurstSingleReq, MDataValid, MBurstLength, MBurstPrecise, MBurstSeq.
byteen mdatabyteen	1 1	Allow partial cache line transfer for reads and writes	MByteEn, MDataByteEn
writeresp_enable	1	Writes have responses	
intport_writedata	0	Must be 0 for slaves	
cohstate_enable	1*	The master interconnect side has no coherent caches but it can deliver coh and coh-non-cached requests.	SCohState
reqinfo	1	Use to carry coherent ID on bit 0, 1, and 2	MReqInfo
threads sthreadbusy sthreadbusy_exact sdatathreadbusy sdatathreadbusy_exac t mthreadbusy mthreadbusy_exact	1 1 1 1 1 1 1	Use single-threaded non-blocking protocol	SThreadBusy, SDataThreadBusy, MThreadBusy

OCP Intervention Port Ip6

Intervention port Ip6 uses the same settings listed in Table 52.

13.3.2.5 Legacy Slave Ports: OCP6, 7, and 8

A regular single-threaded, non-blocking OCP connection is configured for OCP6, 7, and 8 here (e.g., coh_enable = 0); and it allows SRMD bursts, byte enable signals, and writes without responses. OCP6 can support precise bursts of INCR, WRAP, and XOR sequences. OCP7 and 8 can support precise bursts of INCR, STRM, and UNKN sequences.

13.3.3 Master Implementation Models

The master implementation model is functionally similar to the abstract master model discussed in Section 5.10.

13.3.3.1 Coherent Master Model

The cache controller of a caching master (e.g., an initiator with caches) is responsible for (1) determining if a memory request is coherent or not (e.g., targeting at a cacheable/coherent address space or a non-cacheable/non-coherent address space); and (2) issuing the corresponding request to the network. Table 55 below shows such a decision process and is used by coherent masters, CPU1/2, CPU3/4, and Coherent DMA1 in the example design.

Note that not all masters need to be able to issue all types of requests in the example design.

Table 55 Sending Coherent Master (with Caches) Main Port Requests

Master Core Type	**Targeting System Address Region Type**	**Intended Transaction**	**MSI Cache**		**Mp OCP Port Request**
			Status	**State**	
Coherent	Coherent Address Space	Read or LL	Hit	M	None
			Hit	S	None
			Miss	M*	CC_WB, CC_RDSH
			Miss	S†	CC_RDSH
			Miss	I	CC_RDSH
		Write or SC	Hit	M	None
			Hit	S	CC_UPG
			Miss	M*	CC_WB, CC_RDOW
			Miss	S†	CC_RDOW
			Miss	I	CC_RDOW
		Flush or Purge	Any	Any	CC_I
	Non-Coherent Address Space	Read	n/a	n/a	RD
		LL	n/a	n/a	RDL
		Write	n/a	n/a	WR or WRNP
		SC	n/a	n/a	WRC

* A cache miss is encountered and a dirty line is the victimized line. Thus, a CC_WB request is issued for the victimized line first before sending a CC request for the cache miss.

† A cache miss is encountered and a shared line is the victimized line, i.e., no writebacks are needed.

Mp Port Request Transactions

When a coherent master wants to inject a main port request, Table 55 is used to decide what main port request should be sent given the targeting address region (can be determined by the targeting cache-line address and the system address map), the intention (e.g., read or write), internal cache state of the cache line (e.g., M, S, or I), and the cache scheme used by the master. For instance, when the CPU1/2 master wants to write to a cache line, which is in the shared state, to a coherent address space, the CPU master will send a CC_UPG coherent request on the main port to be delivered to the home directory slave.

Processing Ip Port Requests and Returning Ip Port Responses

When intervention requests can be processed, the coherent master's internal cache states need to be updated accordingly; then, intervention responses need to be returned. Table 56 is used by coherent masters in the example design to decide how to update their internal caches and what intervention port responses to be returned based on: the master type, the intervention request type (e.g., self or system), the intervention command (e.g., read for ownership or read for sharing), and the current cache line state. For instance, when the CPU3/4 master receives a system I_UPG intervention request and the targeting cache line is in the S state, the CPU master will invalidate its internal copies of the cached line (i.e., change state from S to I) and return an intervention response of SResp = OK and SCohState = I on its intervention port—the corresponding row in Table 56 is shown in bold face and highlighted blue.

Please note that after receiving some self intervention requests, the cache may get into a transient state (e.g., I_to_S, SI_to_M, or MSI_to_I) as shown in Table 56. The cache line will get out of the transient state after receiving its corresponding Mp port responses. Also, the post fix letter in these transient states indicates the final cache line installing state.

Table 56 Processing Intervention Requests

Master Core Type	Ip Port Intervention Type	Received Transaction	MSI Cache (Transient) State		Generating Outgoing Ip Response and Installing Cache State (MSI)
			From	To	
Coherent	Self Intervention	I_RDOW	M	n/a	n/a
			I, S	SI_to_M	SResp = OK; SCohState = I[*]
		I_RDSH	M, S	n/a	n/a
			I	I_to_S	SResp = OK; SCohState = I[*]
		I_UPG	M	n/a	n/a
			S, I	SI_to_M	SResp = OK; SCohState = I[*]
		I_I,I_WB	M, S, I	MSI_to_I	SResp = OK; SCohState = I[*]
	System Intervention	I_RDOW	M	I	SResp = DVA, SData; SCohState = M[†]
			S, I	I	SResp = OK; SCohState = I[*]
		I_RDSH	M	S	SResp = DVA, SData; SCohState = S[‡]
			S, I	Same state	SResp = OK; SCohState = I[*]
		I_UPG	M	I	SResp = DVA, SData; SCohState = M[†]
			S, I	**I**	**SResp = OK; SCohState = I[*]**
		I_I	M, S, I	I	SResp = OK; SCohState = I[*]
		I_WR	M, S, I	Same state	SResp = OK; SCohState = I[*]

* For self intervention responses, the SCohState does not get used by the directory module; therefore, the signal value is a “don’t care”. For these system intervention responses, the SCohState is also a “don’t care”.
† The SCohState indicates the prior cache state of the targeting cache line.
‡ The SCohState indicates the installing (e.g., next) cache state of the targeting cache line.

Upon Receiving Mp Port Responses

A main port coherent transaction is completed only after the request’s main port responses are received. At that point, the master needs to decide whether to take the ownership of the cache line returned among responses or just to keep a local share copy of the line. It is also possible to do nothing. Moreover, requests blocked previously may also need to be unblocked. Table 57 lists how to decide what actions to be taken by a coherent master in the example design.

Table 57 Upon Receiving Main Port Responses

Master Core Type	Corresponding Mp Port Request Type	Received Mp Port Response	Actions (for each transient state, the final cache line state is determined by the transient state alone)
Coherent	CC_ read-type commands	SResp DVA, SData, SCohState	Send SData to Cache and Update Cache State to SCohState*. Unblock blocked fences.
	CC_ write-type commands	SResp DVA, ScohState	Update Cache State to SCohState*. Unblock blocked fences.
	CC_UPG command	SResp OK, SCohState; or SResp DVA, SCohState	Update Cache State to SCohState*. If DVA, also send SData to Cache. Unblock blocked fences.
	Other CC_ commands	SResp OK, SCohState	Update Cache State to SCohState*. Unblock blocked fences.

* The SCohState indicates the installing (e.g., next) cache state of the targeting cache line.

13.3.3.2 Coherence-Aware DSP Model

For the DSP master, in addition to legacy requests and responses (targeting non-coherent address spaces) that can be injected and received on the OCP main port, coherence-aware requests and responses (targeting coherent address spaces) are also issued and received on the OCP main port. Table 58 shows the requests generated on the main port for the intended transactions.

Table 58 Sending Coherence Aware Main Port Requests

Master Core Type	Targeting System Address Region Type	Intended Transaction	Mp OCP Port Request
Coherence Aware	Non-Coherent Region, Coherent Region	Read	RD
		LL	RDL
		Write	WR or WRNP
		SC	WRC

No hardware cache updates are considered in the example design. However, the DSP master can still have software caches where, for instance, system software is used to move a large amount of data from memory to an internal data buffer (inside the DSP) and from the data buffer to the memory before and after data processing, respectively.

When main port responses are received, the DSP master's OCP wrapper only needs to return data words for reads as indicated in Table 59.

Table 59 Upon Receiving Main Port Responses From Coherent Address Space

Master Core Type	Corresponding Mp Port Request Type	Received Mp Port Response	Actions
Coherence-aware	Legacy read-type commands	SResp DVA, SData	Return SData to DSP or CPU
	Legacy write-type commands	SResp DVA	None
	Other legacy commands	SResp OK	None

13.3.3.3 Legacy DMA Models

Only legacy requests and responses can be injected and received on the legacy OCP port. In the example design, only requests targeting at non-coherent address spaces are issued—such as shown in Table 60. When legacy responses are received, the master only needs to return data words for reads as indicated in Table 59.

Table 60 Sending Legacy Main Port Requests

Master Core Type	Targeting System Address Region Type	Intended Transaction	Mp Ocp Port Request
Legacy	Non-Coherent Region	Read	RD
		Write	WR or WRNP

13.3.4 Slave Implementation Models

In the example design, slaves receive either main port or legacy port requests from the OCP coherent interconnect and return the corresponding responses using the main port. Coherent responses are tagged with proper installing cache-line states. On the other hand, coherent slaves inject system or self intervention requests targeting at coherent masters and then wait for intervention response returned on the intervention port. The installing cache-line state tagged with a coherent response can be used, by the coherent slave, to maintain system-level coherence for the cache line targeted by the response.

13.3.4.1 Coherent Directory Model

In the example design, the coherent Directory slave keeps the outstanding cache locations (using a CohID-indexed vector) and cache states for all cache lines belonging to the Memory module and/or the coherent I/O A. Figure 71 illustrates this concept and shows only the coherent directory slave connecting not only to the interconnect on the top with Mp6/Ip6 but also a legacy port OCP5 in order to send regular OCP requests to the Memory module and the I/O A.

Checking Directory Status and Dispatching Interventions or Memory/I/O Accesses

Upon receiving a main port coherent request, the Directory slave checks its cache directory states first to figure out whether there are outstanding shared cache lines or cache owner in order to decide what actions to take. For instance, the Directory module realizes that the targeting memory data line requested for ownership by CPU1/2 is outstanding and owned by the CPU3/4 (considering dirty). A self I_RDOW intervention request and a system I_RDOW intervention request will be dispatched by the Directory slave to the CPU1/2 master and the CPU3/4 master, respectively. Table 61 is used to capture the Directory module's actions upon receiving a main port request and to decide whether home memory accesses, coherence I/O accesses, or intervention port requests (self and/or system) need to be initiated. Please note that not all commands' actions are currently listed in Table 61.

Figure 71 Directory-Based OCP Coherent Memory Slave

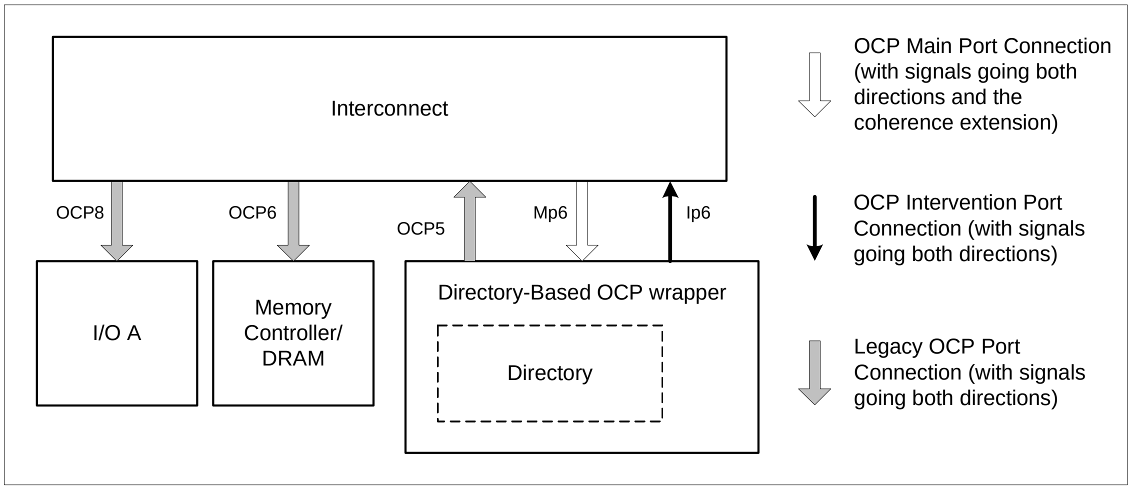

Interconnect
OCP8
OCP6
OCP5
Mp6
Ip6
I/O A
Memory Controller/ DRAM
Directory-Based OCP wrapper
Directory
OCP Main Port Connection (with signals going both directions and the coherence extension)
OCP Intervention Port Connection (with signals going both directions)
Legacy OCP Port Connection (with signals going both directions)

Table 61 Directory Module's Request Action Table

Received Mp Request	**Action**			**Comment**
	Sending Self Intervention on Ip[*]	**Sending System Intervention on Ip**[†]	**Send Legacy Requests to the Memory Module or I/O A**[†]	
CC_RDOW	ActionX(I_RDOW): • Send self I_RDOW request with proper Coh Core ID in MReqInfo	ActionY1(I_RDOW): • Send system I_RDOW with Coh Core ID in MReqInfo for each outstanding Coh Core ID • Wait for N Ip responses	If N == 0: ActionZ1: Send RD to memory or I/O	There can be no outstanding Coherent Core IDs for the targeting cache line; thus, N is 0
CC_RDSH	Take ActionX(I_RDSH)	ActionY2(I_RDSH): • Send system I_RDSH with Coh Core ID in MReqInfo for the outstanding Coh Core ID with state == M • Will wait for N responses (N can be either 1 or 0)	If N == 0: Take ActionZ1	
CC_RDDS	Take ActionX(I_RDDS)	Take ActionY2(I_RDDS)	If N == 0: Take ActionZ1	
CC_UPG	Take ActionX(I_UPG)	Take ActionY1(I_UPG)	If N == 0: Take ActionZ1	
CC_WRI + data words	Take ActionX(I_WRI)	Take ActionY1(I_WRI) + Mp data words	If N == 0: ActionZ2: • Send WR + data words to memory or I/O	May not need to wait for the previous self intervention request to be completed before launching
CC_I	Take ActionX(I_I)	Take ActionY1(I_I)	n/a	May not need to wait for the previous self intervention request to be completed before launching
CC_WB/ CC_CBI + data words	Take ActionX(I_WB/ I_CBI)	n/a	Take ActionZ2	
CC_CB + data words	Take ActinX(I_CB)	n/a	Take ActionZ2	

Received Mp Request	**Action**			**Comment**
	Sending Self Intervention on Ip[*]	**Sending System Intervention on Ip**[†]	**Send Legacy Requests to the Memory Module or I/O A**[†]	
WR + data words to coherent address space	n/a	Take ActionY1(I_RDOW)	If N == 0: Take ActionZ2	
RD to coherent address space	n/a	Take ActionY2(I_RDSH)	If N == 0: Take ActionZ1	
WR + data words to non-coherent address space	n/a	n/a	Send WR + data words to memory or I/O	
RD to non-coherent address space	n/a	n/a	Send RD to memory or I/O	

* When there are no outstanding self intervention requests.
† When the corresponding self intervention requests can be sent.

After Receiving All Intervention Responses or the Memory/I/O Responses

The Directory module updates the directory state for an outstanding coherent transaction according to responses received on the intervention port. Responses received on the intervention port include the transaction's self intervention response and one or more system intervention responses. After examining the directory state and all responses including possible home memory or I/O read responses on the legacy OCP5 port, the directory model generates and sends out main port responses (sometime with SData words) to complete the open coherent transaction. Note that other implementation choices are possible—in certain cases, the directory does not need all responses to send out the main port response.

Table 62 is used to capture the Directory module's actions upon receiving all needed information before returning main port responses.

Table 62 Directory Module's Response Action Table

Original Mp Request	**After Receiving Ip Responses or Memory/ I/O Responses: Send Mp Responses**	**Action**	
		Any Data Words Going to Memory or I/O	**Directory State Changes**
CC_RDOW	IF Ip SResp == DVA: Send Mp SResp of DVA, Ip response data words, SCohState of M* ELSE: Send Mp SResp of DVA, SData = memory or I/O data words, SCohState of M*	n/a	Reset Directory cache line state and set the originating Coh Core ID slot to M
CC_RDSH	IF Ip SResp == DVA: Send Mp SResp of DVA, Ip response data words, SCohState of S* ELSE: Send Mp SResp of DVA, SData = memory or I/O data words, SCohState of S*	Ip response SData words	IF (Ip SResp == DVA): Set Directory cache line state for the originating Coh Core ID slot to S and replace the old dirty slot's state from M to S (for the dirty Coh Core ID)
CC_RDDS	IF Ip SResp == DVA: Send Mp SResp of DVA, Ip response data words, SCohState of I* ELSE: Send Mp SResp of DVA, SData = memory or I/O data words, SCohState of I*	n/a	n/a
CC_UPG	IF Ip SResp == DVA: Send Mp SResp of OK, SCohState of M* ELSE: Send Mp SResp of OK, SCohState of M*	n/a	Reset Directory cache line state and set the originating Coh Core ID slot to M
CC_WRI + data words	Send Mp SResp of OK, SCohState I*	Mp request MData && Ip response SData words	Reset Directory cache line state
CC_I	Send Mp SResp of OK, SCohState I*	n/a	Reset Directory cache line state
CC_WB/ CC_CBI + data words	Send Mp SResp of OK, SCohState I*	IF directory slot was M: Mp request MData	IF directory slot was M: Reset Directory cache line state
CC_CB + data words	Send Mp SResp of OK, SCohState S*	n/a	IF directory slot was M: set the originating Coh Core ID slot to S

Original Mp Request	**After Receiving Ip Responses or Memory/ I/O Responses: Send Mp Responses**	**Action**	
		Any Data Words Going to Memory or I/O	**Directory State Changes**
WR + data words targeting coherent address space	Send Mp SResp of OK, SCohState I[†]	Mp request MData && Ip response SData words	Reset Directory cache line state[†]
RD targeting coherent address space	IF Ip SResp == DVA: Send Mp SResp of DVA, Ip response data words, SCohState of I[†] ELSE: Send Mp SResp of DVA, SData = memory or I/O data words, SCohState of I[†]	Ip response SData words	IF (Ip SResp == DVA): Replace the old dirty slot's state from M to S (for the dirty Coh Core ID)
WR + data words targeting non-coherent address space	Legacy SResp DVA	Mp request MData	n/a
RD targeting non-coherent address space	Legacy SResp DVA, SData = memory or I/O data words	n/a	n/a

* The SCohState indicates the installing (i.e., next) cache state of the targeting cache line.
† For self intervention responses, the SCohState does not get used by the directory module; therefore, the signal value is a "don't care." For these system intervention responses, the SCohState is also a "don't care."

13.3.4.2 Legacy I/O Model

Only legacy requests and responses can be received and returned on the legacy OCP port.

13.3.5 Directory-Based Interconnect System-Level Model

Delivery rules and capabilities employed by the directory-based interconnect include:

- Interconnect delivers all Mp requests originating by initiating masters to the Mp interface of either the Directory/Memory slave, the I/O A slave, or the I/O B slave based on the MAddr field value and the address-region-to-slave assigned described before.
- Interconnect is capable of doing reverse routing in order to return Mp responses back to their initiating masters.
- Interconnect delivers all Ip requests, based on the coherent ID # embedded in the MReqInfo signal, from coherent slaves to coherent masters.
- Interconnect is capable of doing reverse routing in order to return Ip responses from coherent masters back to coherent slaves.

- When connectivity is defined, a (virtual) data stream is maintained between a master Mp to a slave Mp, or a slave Ip to a master Ip.
- No burst interleaving when multiple initiator threads are turned into one target thread at the thread merging points in the interconnect module (we use only single-threaded, non-blocking protocol for all OCP ports in the example design).

In this example design, we do not support cache-to-cache forwarding (i.e., the 3-way communication). However, if it is supported, additional capability needs to be provided by the interconnect. For instance, a virtual data stream between a coherent master's Ip response channel and another coherent master's Mp response channel needs to be established. This also implies that each coherent transaction originated on a main port may get a Data response and a completion response where: (1) both responses may come back in the same time; (2) the Data response may come back first; or (3) the completion response may come back first inside the interconnect. Hence, special attention needs to be taken care of by the interconnect or the coherent master.

13.3.5.1 3-Way Communication Challenges

A typical optimization that a directory-based coherent system can apply is the 3-way communication (or 3-party transaction or cache-to-cache transfer). 3-way communication enables a coherence master, who is forced to write back a dirty copy due to a system intervention, to forward the intervention response and data directly to the requestor. In other words, the intervention port response is routed from the cache line owner's intervention port to the response channel of the requestor's main port directly for improving performance.

Details regarding three-way forwarding will be described in Section 13.3.7.1 on page 287.

13.3.6 Coherent and Coherent-Non-Cached Transaction Flows

A few transaction flows are listed in this sub section and the objective is to capture the relationships between masters and slaves by using transition tables defined in the previous sub sections.

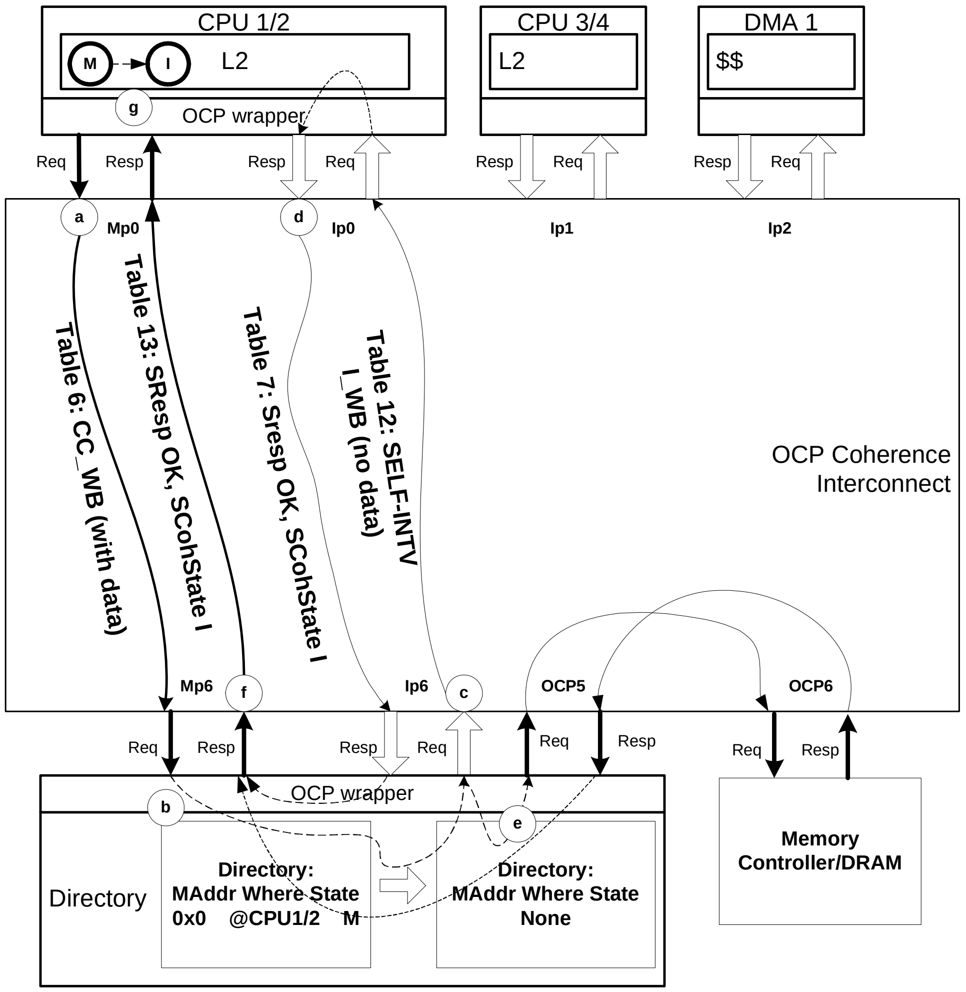
13.3.6.1 Cache Write Back Transaction Flow

Table 72 displays the high-level data flow for a coherent CC_WB transaction originating from CPU 1/2. Transition tables used at each stage is also labelled along with Mp and Ip request and response transfers.

A space-time diagram corresponding to activities shown in Figure 72 is also displayed on the next page. Note that Figure 73 is intended to be used to capture the causality among requests and responses happened for the example design on the main ports, the intervention ports, and/or the legacy ports. The absolute timing differences between messages bear no meaning here. The diagram is not intended to accurately depict the duration of any individual transaction or series of transactions. Conventions used in all space-time diagrams include:

- Time flows from top to bottom.

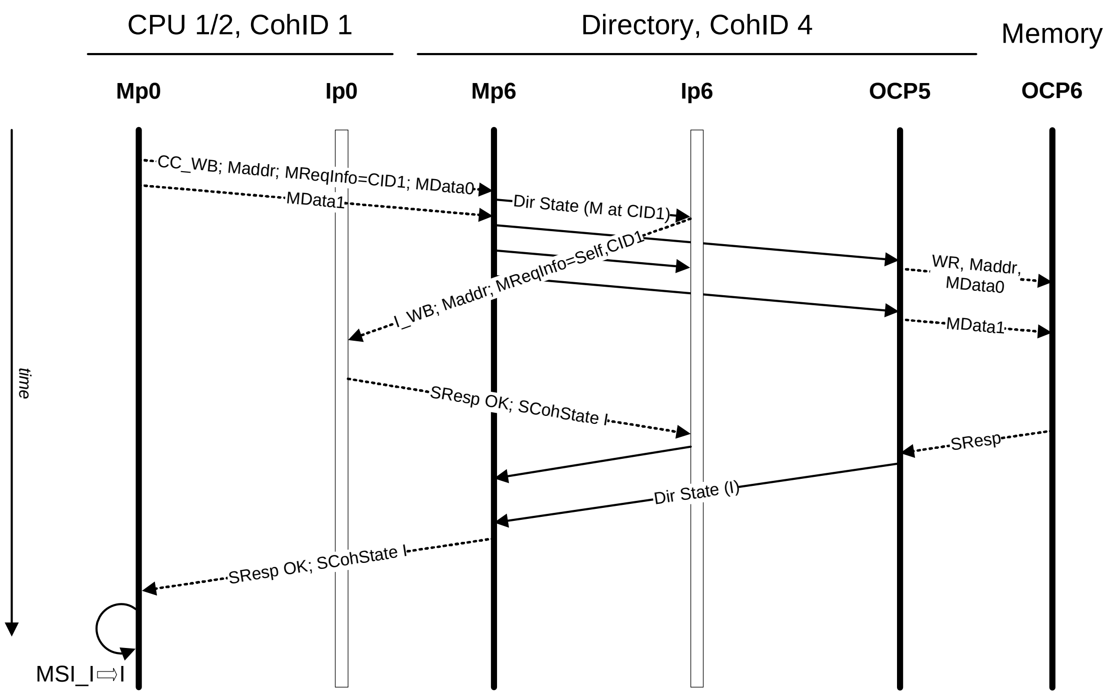
Figure 72 CC_WB Transaction High-Level Data Flow



CPU 1/2
M
I
L2
g
OCP wrapper
CPU 3/4
L2
DMA 1
$$
Req
Resp
Resp
Req
Resp
Req
Resp
Req
a
Mp0
d
Ip0
Ip1
Ip2
Table 6: CC_WB (with data)
Table 13: SResp OK, SCohState I
Table 7: Sresp OK, SCohState I
Table 12: SELF-INTV I_WB (no data)
OCP Coherence Interconnect
Mp6
f
Ip6
c
OCP5
OCP6
Req
Resp
Resp
Req
Req
Resp
Req
Resp
OCP wrapper
b
e
Directory
Directory: MAddr Where State 0x0 @CPU1/2 M
Directory: MAddr Where State None
Memory Controller/DRAM

- All ports of each OCP entity are shown.
- Transactions are represented by arrows between ports.
- Labels indicate the major action(s), including state change(s), that are performed by the transaction.

Figure 73 CC_WB Transaction Flow



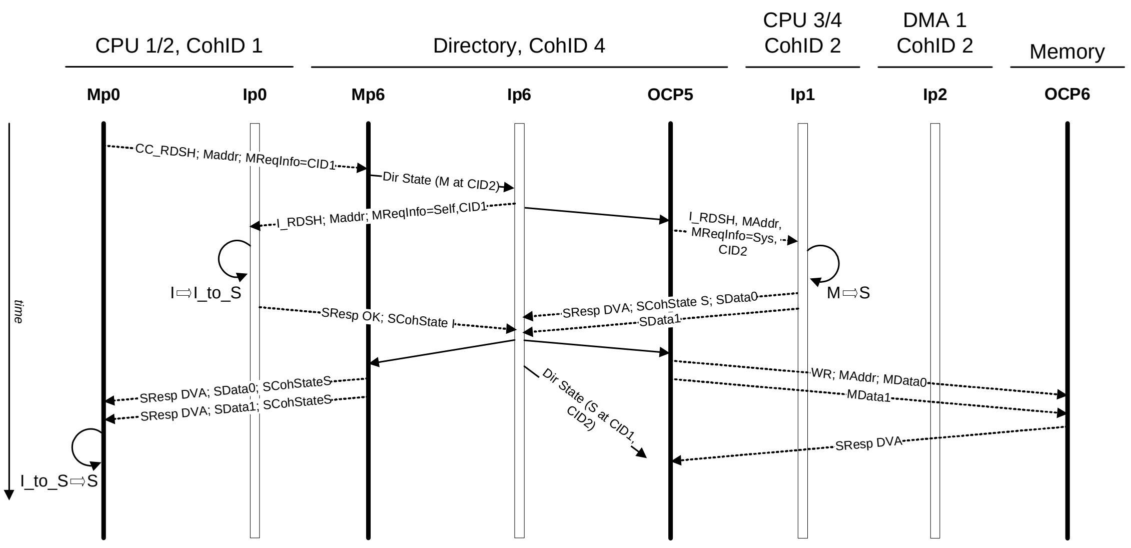
For instance, Table 73 is used to illustrate the following:

- For CPU 1/2, after it issues a CC_WB main port request and two datahandshake phases (MData0 and MData1), it will receive a self I_WB intervention request on its intervention port Ip0 and after the CPU responding with an intervention port response (SResp OK and SCohState of I), it will eventually receive a main port SResp to indicate the completion of it CC_WB transaction (and the cache line state goes to I).
- On the other hand, the Directory module not only sends a self intervention request back to CPU 1/2 (indicated by CID1), it also writes back the cache line (MData0 and MData1) to its home Memory module using its legacy OCP5 port.

13.3.6.2 Read for Share and Dirty at a Master Cache

Figure 74 displays the space-time data flow for a coherent CC_RDSH transaction originating from CPU 1/2 where the dirty data is located at the CPU 3/4 master's cache. Note that the Directory module knows that the latest dirty cache line is stored at the cache of the CPU 3/4 module (CID2). Therefore, in addition to return a self intervention request back to the originating CPU 1/2 module, it also sends a system intervention request from the Ip6 port to the CPU 3/4 module's intervention Ip1 port (the request's MReqInfo signal carries "system" and CID2). After receiving the intervention I_RDSH request, the CPU 3/4 module not only changes its cache line state from M to S but also returns the latest cache line data words (SData0 and SData1) to the directory before being copied and sent to both the CPU 1/2 module and the Memory sub system using Mp6 and Mp0 ports, and OCP5 and OCP6 ports, respectively.

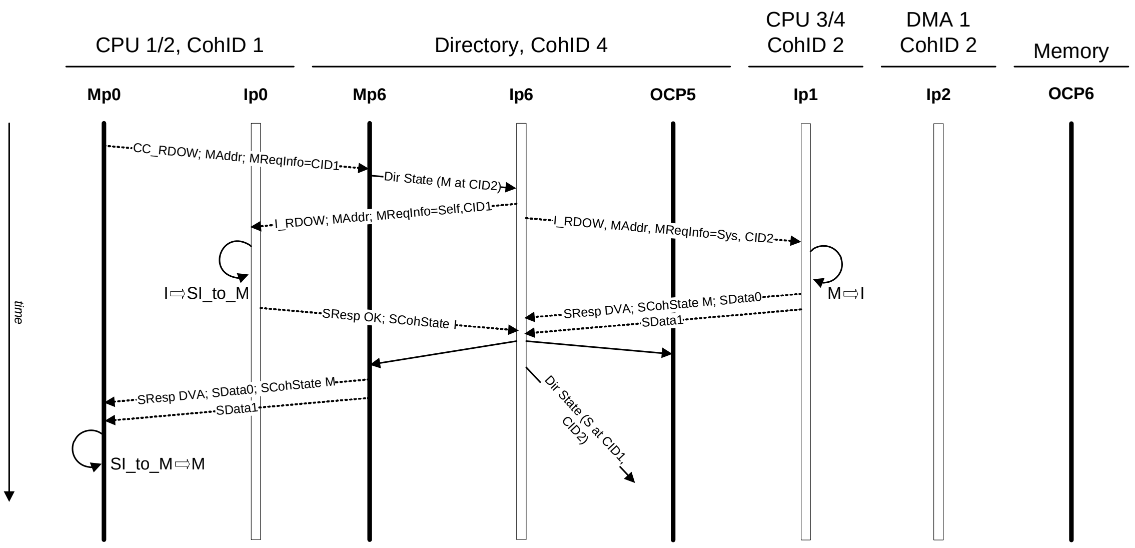
Figure 74 CC_RDSH and Dirty at Cache


CPU 1/2, CohID 1
Directory, CohID 4
CPU 3/4 CohID 2
DMA 1 CohID 2
Memory
Mp0
Ip0
Mp6
Ip6
OCP5
Ip1
Ip2
OCP6
time
CC_RDSH; Maddr; MReqInfo=CID1
Dir State (M at CID2)
I_RDSH; Maddr; MReqInfo=Self,CID1
I_RDSH, MAddr, MReqInfo=Sys, CID2
I⇨I_to_S
M⇨S
SResp OK; SCohState I
SResp DVA; SCohState S; SData0
SData1
WR; MAddr; MData0
MData1
SResp DVA; SData0; SCohStateS
SResp DVA; SData1; SCohStateS
Dir State (S at CID1, CID2)
SResp DVA
I_to_S⇨S

13.3.6.3 Read for Ownership and Dirty at a Master Cache

Figure 75 displays the space-time data flow for a coherent CC_RDOW transaction originating from CPU 1/2 where the dirty data is located at the CPU 3/4 master's cache. The communication pattern is similar to Figure 74 except that the cache installing states are different and there is no need to copy the cache line to memory. (Note: some implementations may choose to update the memory concurrently.)

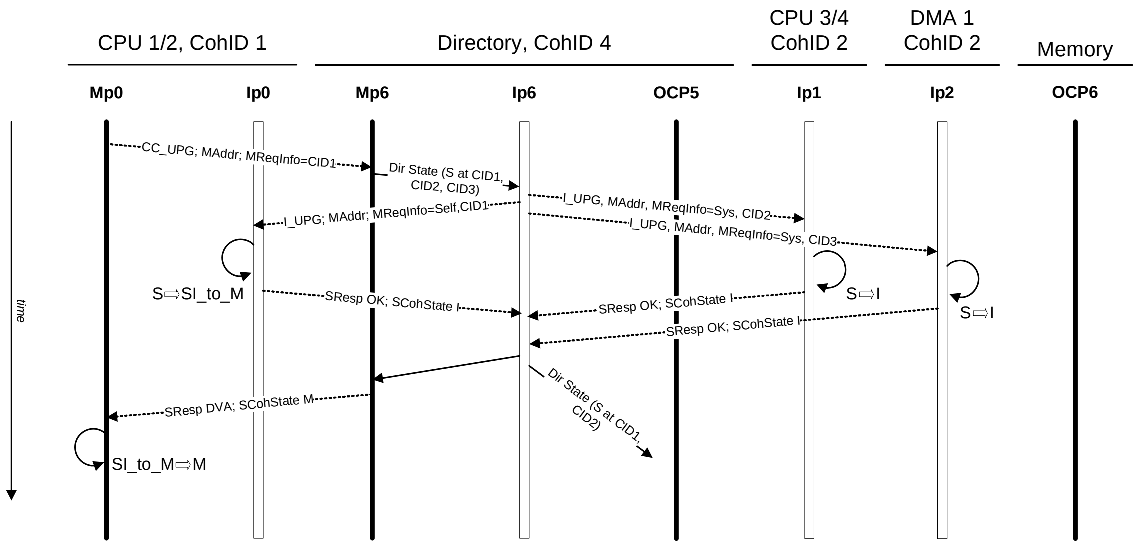
Figure 75 CC_RDOW and Dirty at Cache


CPU 1/2, CohID 1
Directory, CohID 4
CPU 3/4 CohID 2
DMA 1 CohID 2
Memory
Mp0
Ip0
Mp6
Ip6
OCP5
Ip1
Ip2
OCP6
time
CC_RDOW; MAddr; MReqInfo=CID1
Dir State (M at CID2)
I_RDOW; MAddr; MReqInfo=Self,CID1
I_RDOW, MAddr, MReqInfo=Sys, CID2
I⇨SI_to_M
M⇨I
SResp OK; SCohState I
SResp DVA; SCohState M; SData0
SData1
SResp DVA; SData0; SCohState M
SData1
Dir State (S at CID1, CID2)
SI_to_M⇨M

13.3.6.4 Cache Upgrade When the Cache Line is Shared by Multiple Masters

Figure 76 displays the space-time data flow for a coherent CC_UPG transaction originating from CPU 1/2 where the cache line is also shared at the CPU 1/2 master's cache, the CPU 3/4 master's cache, and the DMA1 master's cache. Therefore, after the Directory receives the CC_UPG request, it sends three intervention requests, a self intervention request to the CPU 1/2 master, two I_UPG system intervention requests to the CPU 3/4 master and the CPU 5/6 master each.

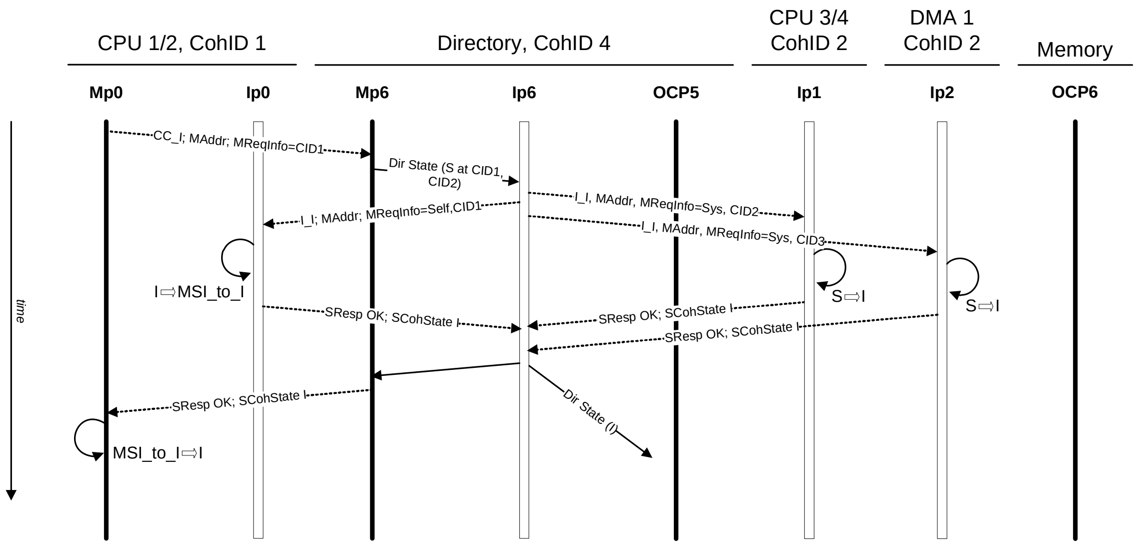
Figure 76 CC_UPG and Shared at All Caches



13.3.6.5 Cache Flush or Purge and Shared at Master Caches

Figure 77 displays the space-time data flow for a coherent CC_I transaction originating from CPU 1/2 where the cache line is shared at the CPU 3/4 and DMA1 master's cache.

Figure 77 CC_I and Shared at Others



13.3.7 Three-Way Communication

A typical optimization that a directory-based coherent system can apply is the three-way communication (or three-party transaction or cache-to-cache transfer). Three-way communication enables a coherence master, who is forced to write back a dirty copy due to a system intervention, to forward the intervention response and data directly to the requestor. In other words, the intervention port response is routed from the cache line owner's intervention port to the response channel of the requestor's main port. This transaction requires the MReqInfo signal in the intervention port request channel to indicate both "who to route to" and "to whom to forward to." In addition, there should be a SRespInfo signal on the intervention port response side to indicate "who is the recipient." The SRespInfo signal should also be included in a main port response as well, to indicate "to whom to respond," if three-way communication is used. This main port SRespInfo signal needs to be used by the delivering fabric to determine how to return the main port responses and data words.

Three-hop protocols introduce additional complexity to the design of the main response port: the design needs to accommodate the receipt of two responses for the same transaction (see below).

13.3.7.1 Cache Answers Request

Figure 78[1] shows a CC_RDSH transaction is issued from a master and gets a dirty cache-line data from another coherence master directly. In the same time, the home memory slave's directory state is updated accordingly. Steps to complete the coherence transaction are labeled alphabetically starting from a, b, c, to g, h, and i.

[1] Please note that in this figure we have intentionally made the memory module an internal module inside the Directory slave.

The corresponding space-time diagram to the above “CC_RDSH and cache answers request” transaction is displayed in Figure 79—the labels indicated on dashed communication activities match those from Figure 78.

Figure 78 CC_RDSH and Cache Answers Request High-Level Data Flow

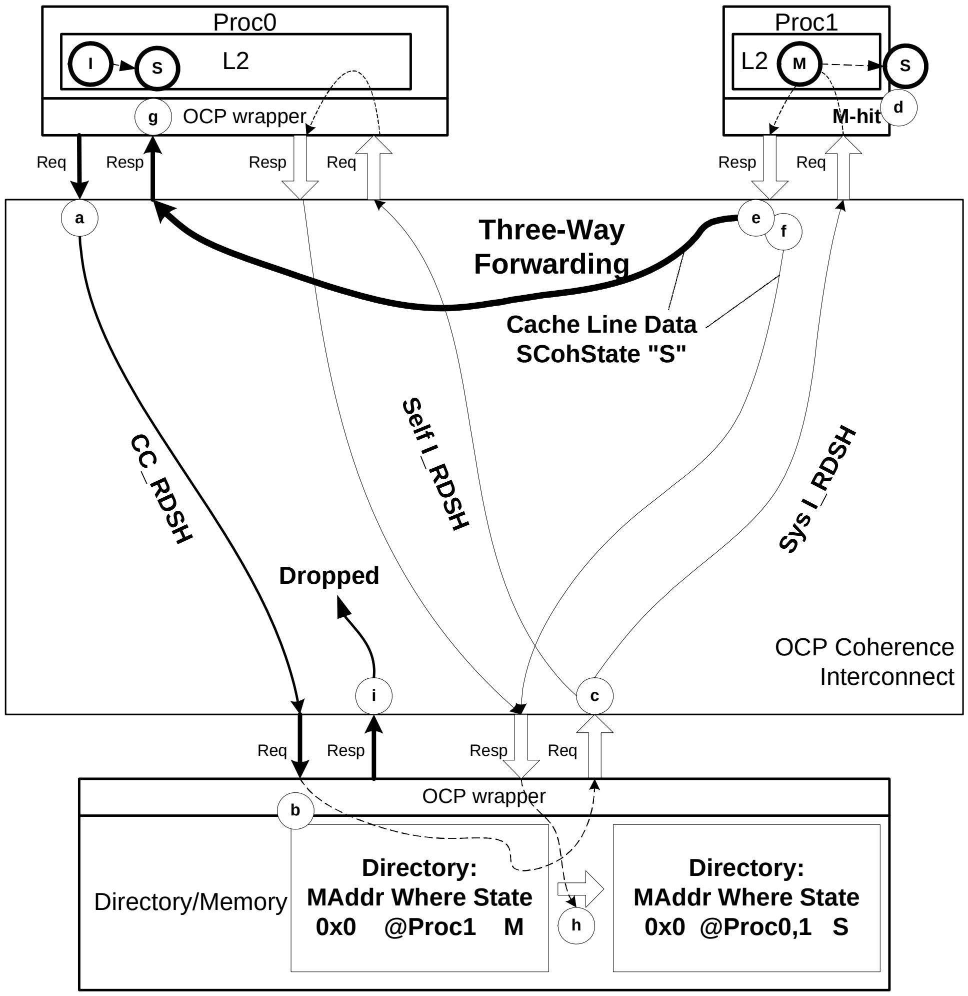
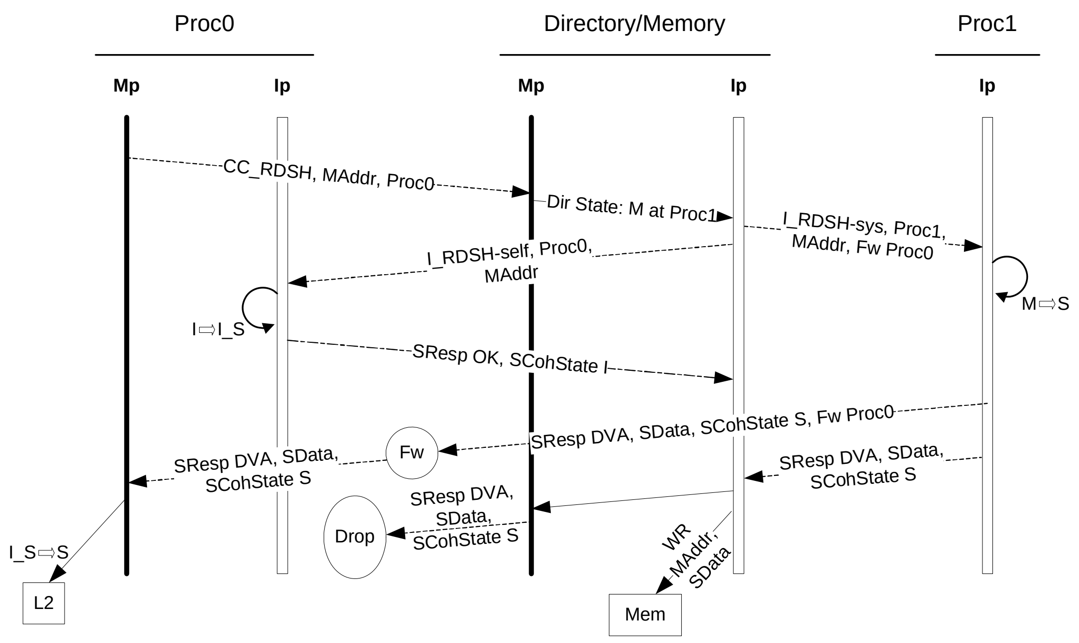


Figure 79 CC_RDSH and Cache Answers Request Space-Time Diagram


Proc0
Directory/Memory
Proc1
Mp
Ip
Mp
Ip
Ip
CC_RDSH, MAddr, Proc0
Dir State: M at Proc1
I_RDSH-sys, Proc1, MAddr, Fw Proc0
I_RDSH-self, Proc0, MAddr
M⇨S
I⇨I_S
SResp OK, SCohState I
SResp DVA, SData, SCohState S, Fw Proc0
Fw
SResp DVA, SData, SCohState S
SResp DVA, SData, SCohState S
SResp DVA, SData, SCohState S
Drop
WR MAddr, SData
I_S⇨S
L2
Mem

13.3.8 Handling Race Conditions

A few of the transitions described in Table 55 on page 272 seem unnecessary but those transactions are indeed needed for the coherence master model in order to operate correctly. They are there to handle possible race conditions between coherence masters, for instance, for a directory-based coherence system. The following scenario illustrates one possible race condition between coherence master A and B that both want to update the coherence state of cache-line address X:

- At time *t*, coherence master A issues a coherence upgrade request (CC_UPG) on the main port for cache-line address MAddr X because master A's processor finds the cache-line's state in S and wants to upgrade to M. This implies that the coherence state module inside master A must also record cache-line X in State S.
- At time *t*+1, coherence master B issues a coherence CC_RDOW request on its main port also for cache-line of address MAddr X because master B's processor finds the cache-line's state in I and wants to change it to M. This implies that the coherence state module inside master B must have no record for cache-line X.
- At time *t*+3, master B accepts a self-intervention CC_RDOW request on MAddr X and blocks its intervention port waiting for a main port response of the main port CC_RDOW request master B sent at time *t*+1.
- At time *t*+4, master A accepts an intervention port CC_RDOW request on MAddr X (coming from master B). Master A finds the coherence state of cache-line address X is in State S. Therefore, by following the master model's transition block diagram, master A (a) returns a intervention port

response of “SResp OK and SCohState I”; and (b) changes the coherence state of cache-line address X to State to I. This may ripple through caches on the processor side.

- More activities happen during time *t*+4, *t*+5, *t*+6, and *t*+7. That is, the directory returns cache line data to B and the CC_UPG coherence request coming from master A has been accepted by the directory.
- At time *t*+8, say, master A receives and accepts a self-intervention CC_UPG request on MAddr X for the CC_UPG request master A sent at time *t*. Master A first blocks its intervention port; then, checks its coherence state module and finds cache-line address X is in State I—which is different from the State when master A was issuing the CC_UPG request. However, we do have a transition from State I to M for handling the receiving of a self-intervention CC_UPG request as shown in Table 55 so master A can complete its remaining operations.

This example illustrates the race condition and explains the need for the transition row (including “Self Intervention: I_UPG” and from state I to state M).

Other transitions listed in Table 55 that are used to handle race conditions are the “S” to “SI_to_M” transition labelled with “Self Intervention: I_RDOW” and the “I” to “I” transition labelled with “Self Intervention: I_I” or “Self Intervention: I_WB”.

13.4 Implementation Models for Snoop-Bus-Based Designs

In this chapter examples of snoop-bus based OCP coherence designs are given.

13.4.1 Snoop-Bus-Based OCP Coherent Master Model

In this example, the OCP coherent master models described in the previous chapter (for directory-based designs) are reused.

13.4.2 Snoop-Bus-Based OCP Coherence Interconnect Model

The interconnect has a broadcast facility. It is convenient to think of each coherence transaction as being composed of three in-order pipeline bus stages: the Request/Write Data phase, the Snoop phase, and the Response/ Read Data phase. At most one instance of each stage can exist at each cycle; therefore, there can be at most three in-flight transactions. Stalling one stage stalls all pending transactions happening before the stalling stage. Transactions passed the stalling stage can proceed without problems.

13.4.2.1 Request/Write Data Phase

The arbitration logic in the interconnect grants at most one main port coherence request each cycle in this phase. During this phase, this coherence request is: (a) delivered to the memory slave on its main port based on the MAddr value; (b) turned into a self-intervention request (data-less) and delivered to the intervention port of the initiator; (c) turned into many intervention requests and delivered to each of the other coherence masters connected to the snoop-bus-based interconnect. When all requests are accepted by their targeting masters and slave, this phase is considered complete. Request type (MCmd), address (MAddr), and write data (MData), if any, need to be delivered in this phase.

13.4.2.2 Snoop Phase

After all intervention requests are delivered, we enter the snoop phase waiting for intervention port responses. In addition, one intervention request is expected from the slave, i.e., the "home" of the MAddr address. This intervention request is dropped by the interconnect.

After receiving all intervention responses, two possible scenarios can happen:

- If none of the intervention responses has dirty write-back data (i.e., SCohState is set to M), an aggregated intervention response is generated in this phase and delivered to the "home" slave on its intervention port. The snoop phase terminates when the aggregated intervention response is accepted.
- Otherwise, exactly one of the intervention responses must return the write-back data and the data is passed onto the next phase before the snoop phase terminates.

SResp, SCohState, and possible SData need to be delivered in this phase. Please notice that in the mean time the home slave of the MAddr address can be launching a memory read or write command

13.4.2.3 Response/Read Data Phase

If this is a response-only phase, the interconnect is waiting for a main port response coming from the home slave of the transaction's MAddr address and will relay the response to the transaction's initiating coherence master. This phase terminates when the main port response is accepted. For a response-and-read-data phase, the dirty write-back data of the previous phase is packaged into an intervention response and sent to the home slave; in addition, it is also packaged into a main port response with the dirty write-back data and sent to the transaction's initiating coherence master. The phase terminates after accepting/dropping a main port response coming back from the home slave. Note that, in parallel, a memory write-back operation should also take place at the home slave.

13.4.3 Snoop-Bus-Based OCP Coherence Slave Model

We will use the abstract model presented in Figure 12 on page 89.

13.4.4 Coherence Transactions

Figure 80 shows a CC_RDOW transaction is issued from a master and gets a dirty cache-line data from another coherence master directly. Figure 81 is the space-time diagram corresponding to the transaction shown in Figure 80. Figure 82 shows either a CC_RDOW or CC_RDSH transaction is issued and the memory slave provides the cache-line data. Figure 83 is the space-time diagram corresponding to the CC_RDSH transaction shown in Figure 82. A CC_WB transaction writes back cache-line data is illustrated in Figure 84 and the corresponding space-time diagram is shown in Figure 85.

In Figures 80, 82, and 84, the following abbreviations are used:

- Mp: Main port
- Ip: Intervention port
- Self Ip Req or Resp: Self intervention request or response
- Req: Requests
- Resp: Response

Figure 80 Snoop-Bus-Based Interconnect Example: Cache Answers Request

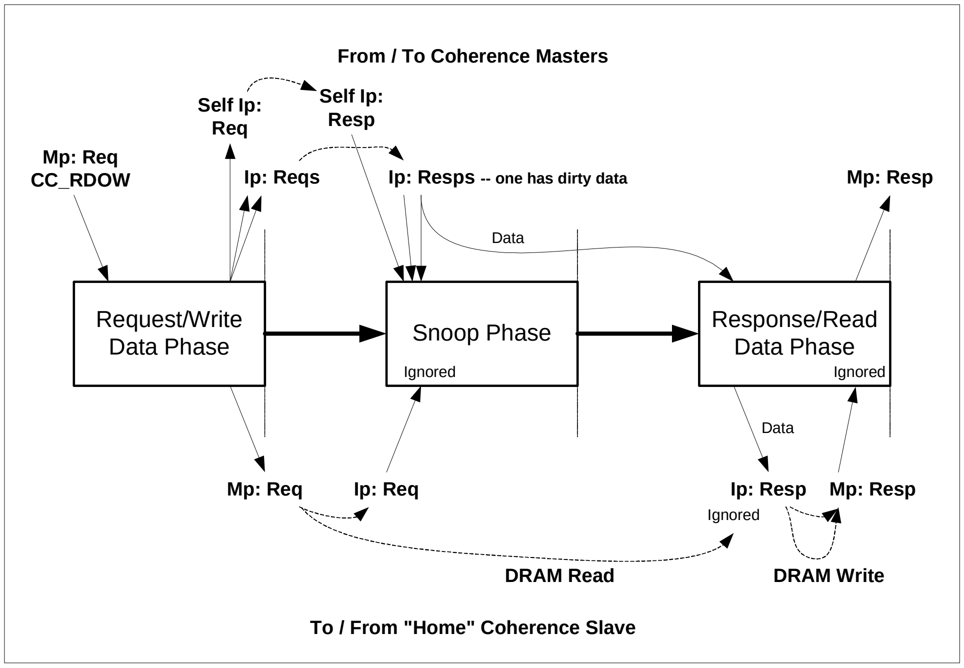


Figure 81 Space-time diagram of Figure 80

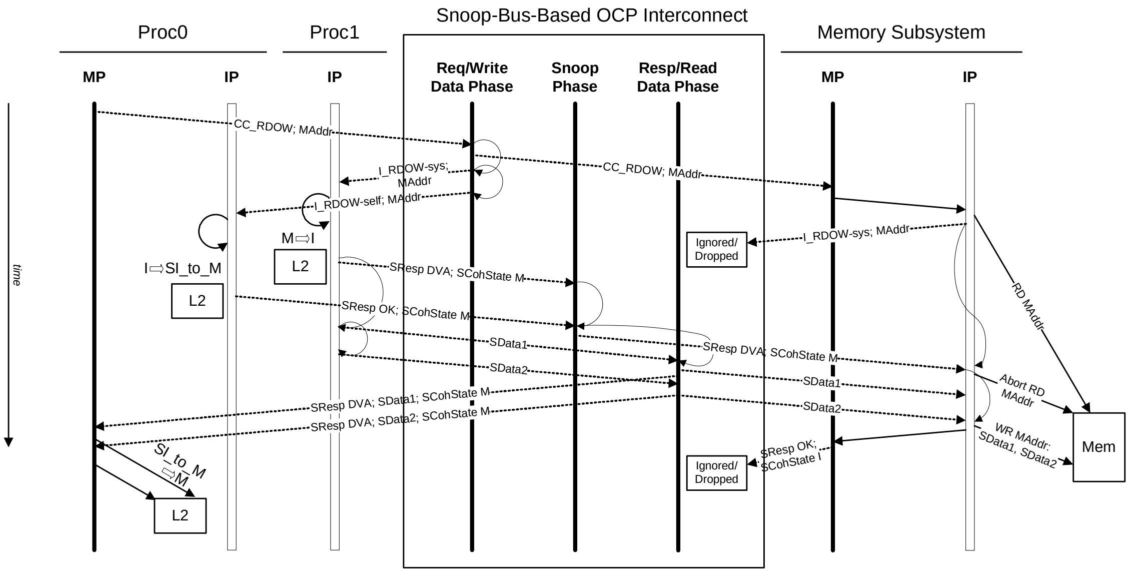


Figure 82 Snoop-Bus-Based Interconnect Example: Memory Answers Request

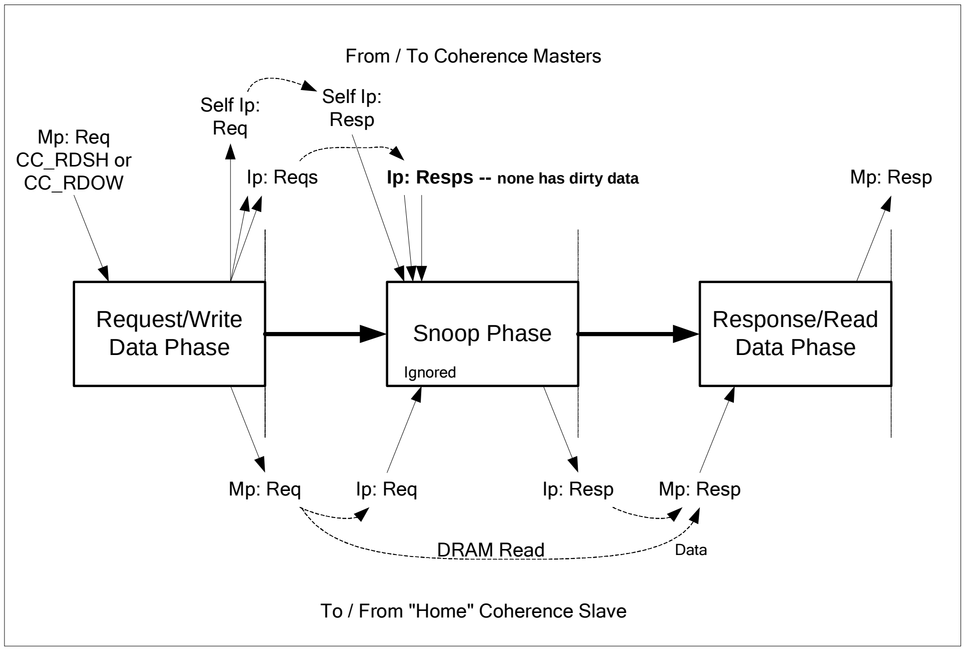


Figure 83 Space-time diagram of Figure 82's CC_RDSH Transaction

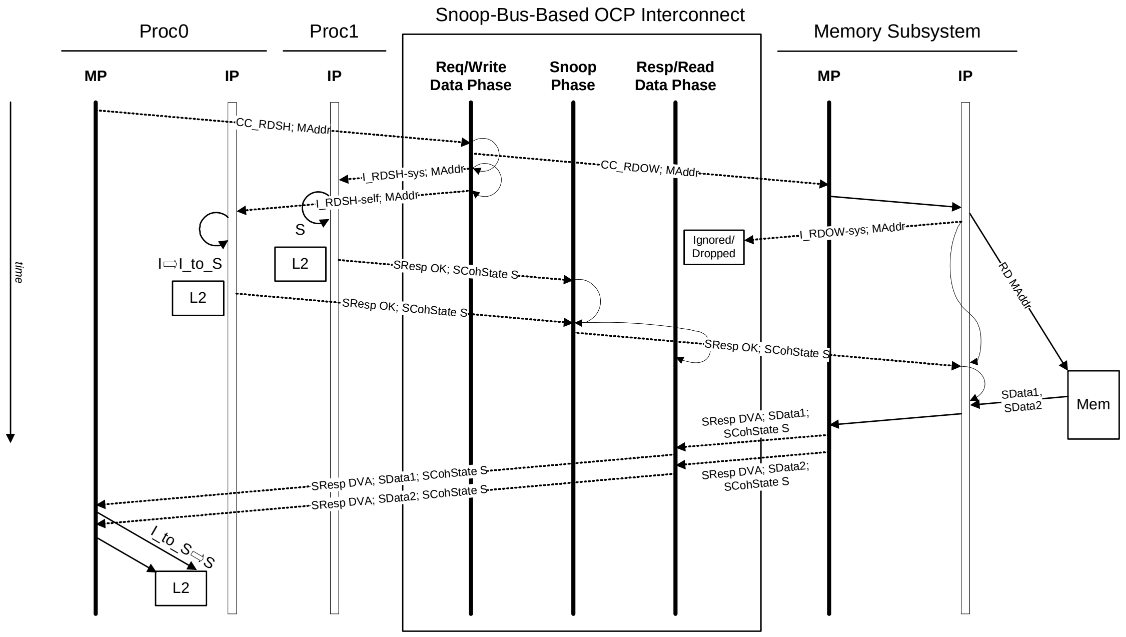


Figure 84 Snoop-Bus-Based Interconnect Example: Writeback

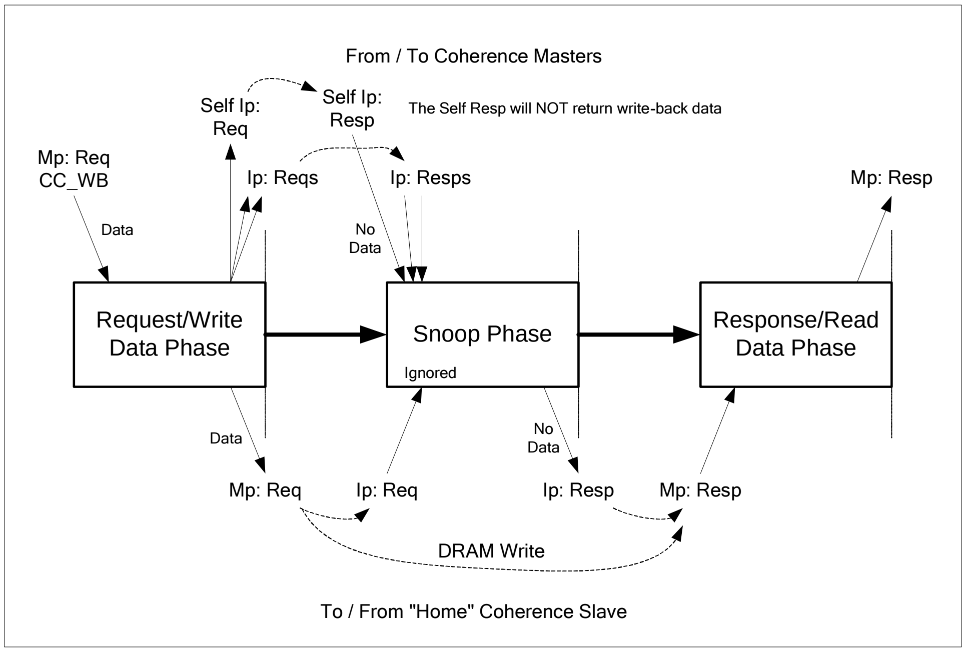
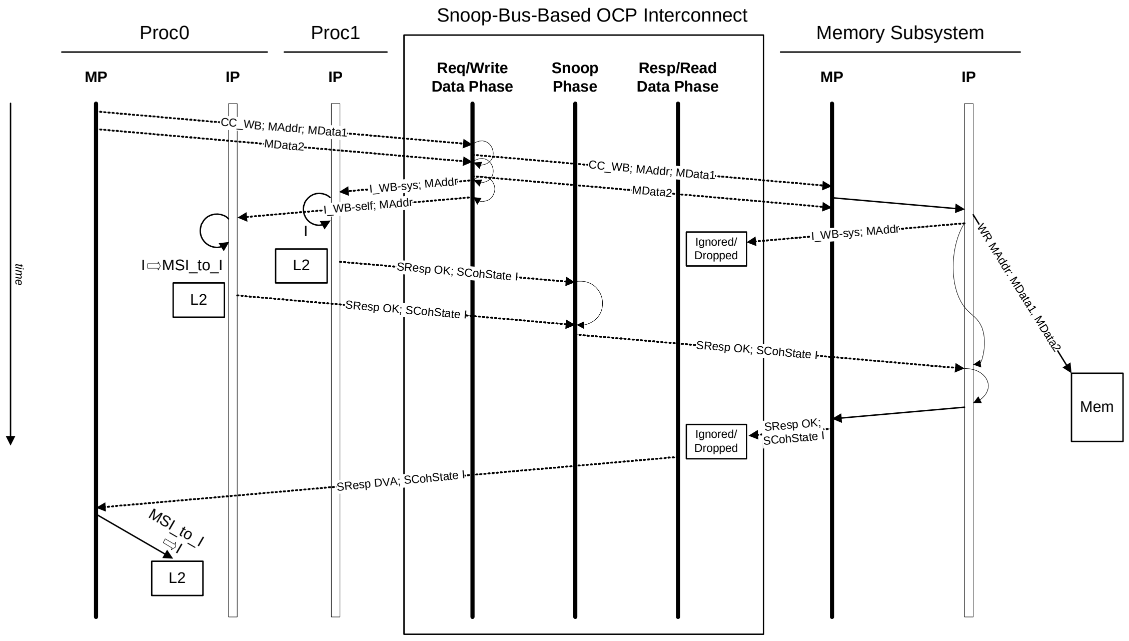


Figure 85 Space-time diagram of Figure 84



13.4.5 Snoop-Bus-Based CC_WB Race Conditions

In this subsection, several examples showing CC_WB race conditions in a snoop-bus-based system are discussed, and an implementation choice to solve each of the race conditions is also described. In the models shown in this section, the primary serialization point referred to in Section 5.11.2 and the associated serialization logic is implemented as two units: the *coherent request serialization/select logic* unit and the *SResp merge logic* unit.

13.4.5.1 Intport_writedata=0 Case (1), Proc0 Wins

The behaviors of the coherent masters (processors), interconnect, and memory subsystem are based on Tables 55–57 and Figures 80–85.

There are two choices to properly maintain cache coherency:

1. Write to coherent memory is initiated after the intervention responses are received. If this approach is chosen, the intervention response from the coherent master described in Table 55 is changed: the coherent master sends the current cache state instead of the next state for both self intervention and system intervention cases.
2. Cancel on-the-fly coherent write requests when the cache state is changed. If this approach is chosen, all write buffers in the coherent domain are required to compare the address of the coherent requests with the address of the selected command which are issued from the coherent request serialization/select logic unit.

Implementations that follow the second approach require more complex mechanisms and careful design than implementations that follow the first approach; hence, in this document, for reasons of brevity, only examples that follow the first approach are shown.

An example of the race condition of CC_WB from Proc0 and CC_RDOW from Proc1 at the same memory address are shown in Figures 86(a) and 86(b). Figure 86(a) shows the case in which Proc0 wins arbitration at the coherent request serialize/select logic unit in the OCP coherent interconnect: event (A2) appears prior to event (B2), and event (A3) appears prior to event (B3). In this case, only the order of event (A14) completes before event (B7) starts, event (A10) completes before event (B4), and no other ordering between Proc0 and Proc1 is maintained.

The behaviors related to the CC_WB request from Proc0 are shown in Figure 86(a).

(A1) Proc0 sends a Cache Coherent Write Back (CC_WB) command with data from its main request port.

(A2) The coherent request serialize/select logic unit in the interconnection network selects the CC_WB command from Proc0 and sends this command with its associated MData to the target, Memory 0.

(A3) The coherent request serialize/select logic unit sends I_WB as a self intervention request to the intervention request port of the initiator, Proc0. Since CC_WB does not require a check of the cache states of other coherent masters, the OCP coherence interconnect does not send I_WB as a system intervention request to the other coherent masters.

(A4) OCP wrapper of Memory0 receives the CC_WB command with MData from its main request port, translates the CC_WB request into an I_WB request, and sends the I_WB request to its intervention request port. (Note: Memory0 will execute the write after it receives the intervention response at (A10).)

(A5) OCP wrapper of Memory0 sends I_WB from its intervention request port. The OCP coherence interconnect ignores and drops this request.

(A6) Proc0 receives I_WB as a self intervention request from its intervention request port and checks its cache state for the requested address location. Proc0 changes the cache state from M to MS_to_I.

(A7) Proc0 sends its cache state, Modified, to its OCP wrapper.

(A8) OCP wrapper of Proc0 translates the snoop response Modified into “SResp OK, SCohState Modified,” and sends it from its intervention response port.

(A9) SResp merge logic unit in the OCP coherence interconnect receives the intervention port response (A9), generates the intervention response “SResp OK, SCohState Modified” for the coherent memory system, and sends it to the intervention port of Memory0.

(A10) OCP wrapper of Memory0 receives the intervention response “SResp OK, SCohState Modified” and executes “memory write” to Memory0.

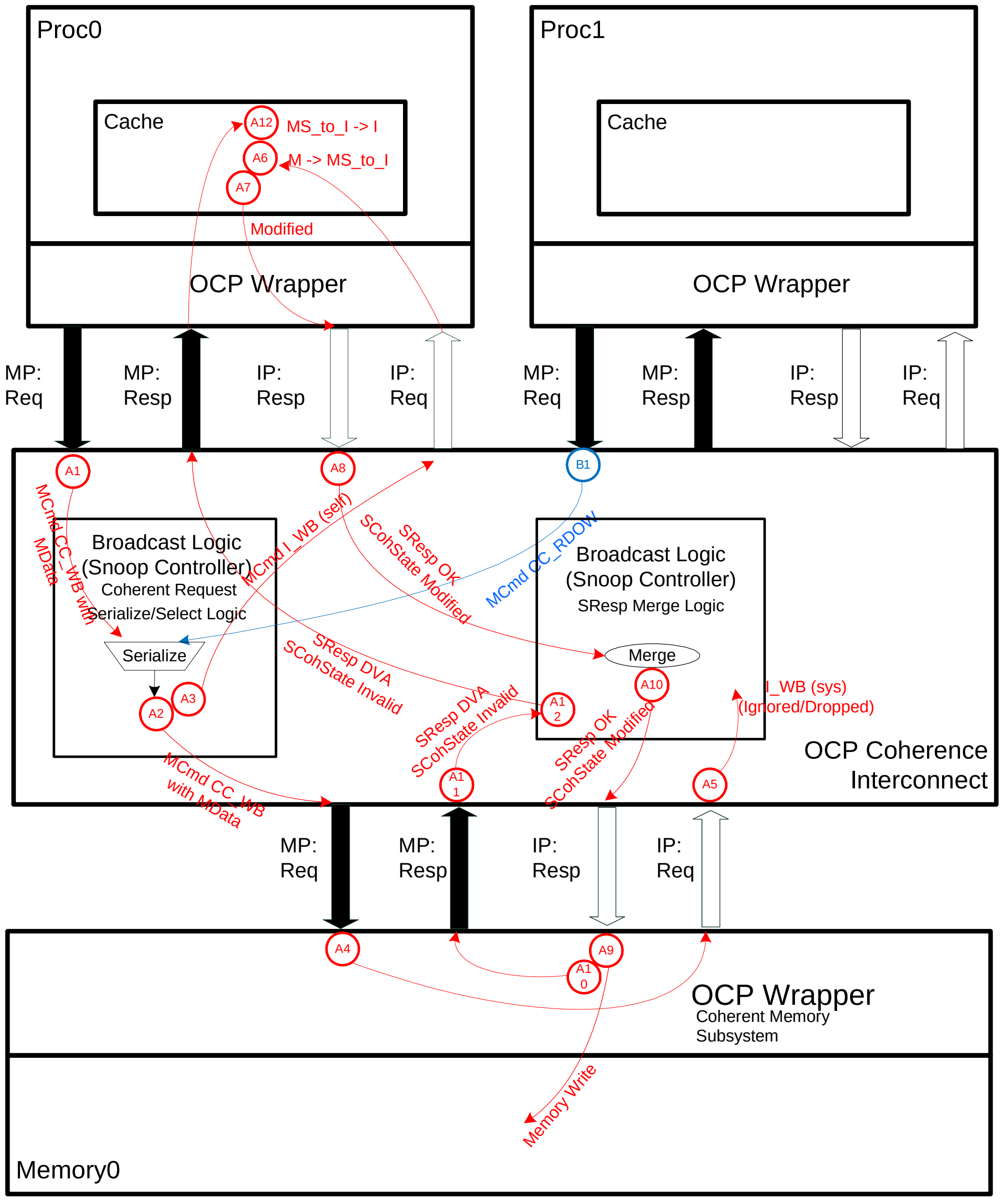
(A11) OCP wrapper of Memory0 generates the main port response "SResp DVA, SCohState Invalid," and sends it to its main response port.

(A12) OCP wrapper of Memory0 sends "SResp DVA, SCohState Invalid" from its main response port.

(A13) SResp merge logic unit in the OCP coherence interconnect receives the main port response from Memory0. It sends "SResp DVA, SCohState Invalid" as the main port response to Proc0.

(A14) Proc0 receives "SResp DVA, SCohState Invalid" as the main port response, and changes its cache state from MS_to_I to I. The CC_WB command transaction ends.

Figure 86(a) CC_WB Race Condition, Proc0



The behaviors related to the “CC_RDOW” request from Proc1 are shown in Figure 86(b).

(B1) Proc0 sends the Cache Read Own (CC_RDOW) command from its main request port.

(B2) The coherent request serialize/select logic unit in the OCP coherence interconnect network selects the CC_RDOW command from Proc1 and sends this command to the target, Memory 0. Note that at this point, the CC_WB command issued by Proc0 has already been sent to Memory0.

(B3) The coherent request serialize/select logic unit sends I_RDOW as a self intervention request to the intervention request port of the initiator, Proc1. It also sends I_RDOW as system intervention requests to all other coherent masters, including Proc0. Note that at this point, I_WB from Proc0 has already sent to Proc0.

(B4) OCP wrapper of Memory0 receives the CC_RDOW from its main request port, and sends “Memory Read” speculatively to Memory0. Memory0 executes the read.

(B5) OCP wrapper of Memory0 translates the CC_RDOW request into an I_RDOW request and sends it to its intervention request port.

(B6) OCP wrapper of Memory0 sends I_RDOW from its intervention request port. OCP coherence interconnect ignores and drops this request.

(B7) Proc0 receives I_RDOW as a system intervention request from its intervention request port and checks its cache state at the requested address location. Since the cache state is I, Proc0 does not change the cache state.

(B8) Proc0 sends the cache state “Invalid” to its OCP wrapper.

(B9) OCP wrapper of Proc0 translates the snoop response “Invalid” into “SResp OK, SCohState Invalid,” and sends it from its intervention response port.

(B10) Proc1 receives I_RDOW as a self intervention request from its intervention request port and checks its cache state at the requested address location. Proc1 changes the cache state from I to SI_to_M.

(B11) Proc1 sends the cache state “Invalid” to its OCP wrapper.

(B12) OCP wrapper of Proc1 translates the snoop response “Invalid” into “SResp OK, SCohState Invalid,” and sends it from its intervention response port.

(B13) SResp merge logic unit in the OCP coherence interconnect receives the intervention port responses (B9) and (B12), generates the intervention response “SResp OK, SCohState Invalid” for the coherent memory system, and sends it to the intervention port of Memory0.

(B14) OCP wrapper of Memory0 receives the intervention response “SResp OK, SCohState Invalid” and sends it to its main response port.

(B15) Main response port of the OCP wrapper of Memory0 waits for the read data.

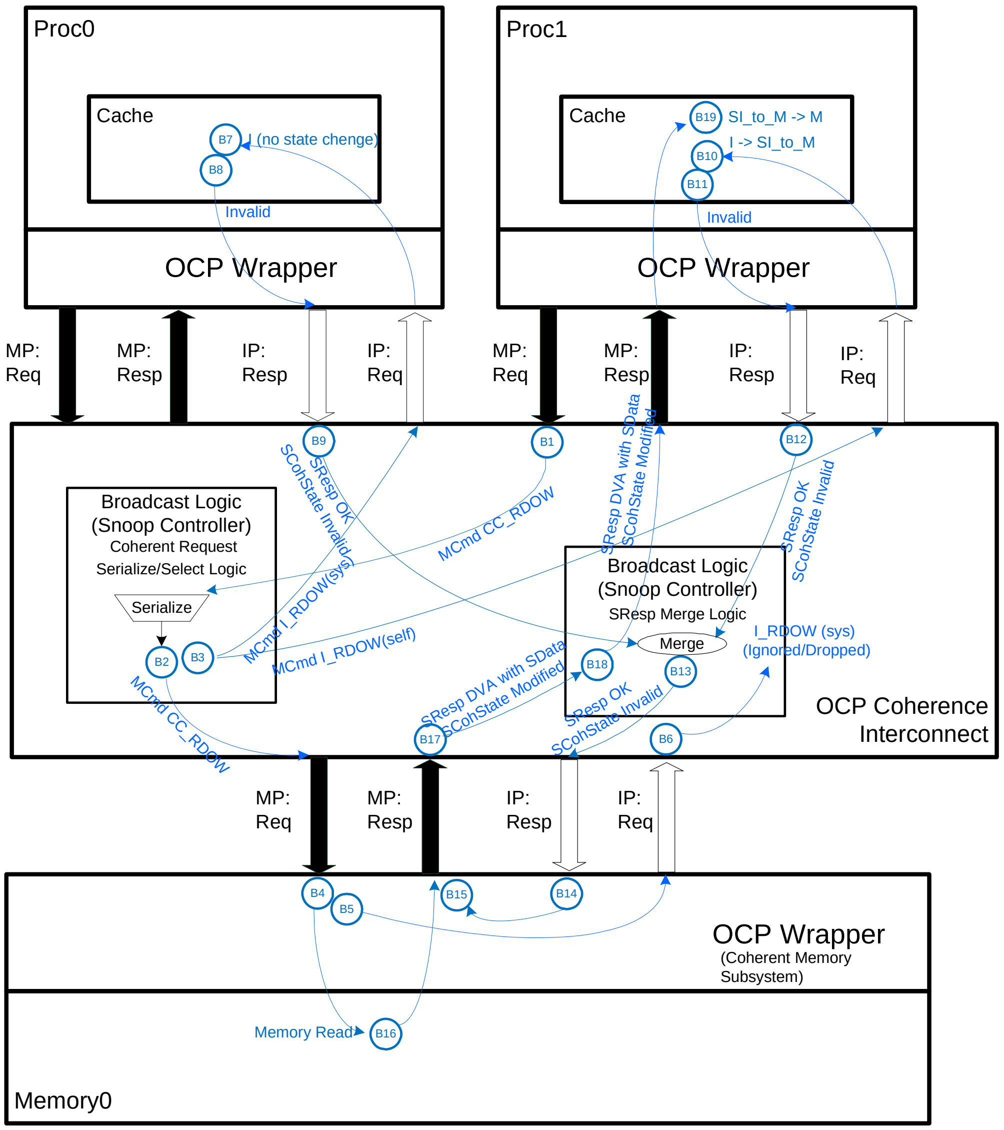
(B16) Memory0 sends the read data to its main response port.

(B17) The OCP wrapper of Memory0 sends “SResp DVA with SData, SCohState Modified” as the main port response.

(B18) SResp merge logic unit in OCP coherence interconnect receives the main port response from Memory0. It sends “SResp DVA with SData, SCohState Modified” as the main port response to the Proc1.

(B19) Proc1 receives “SResp DVA with SData, SCohState Modified” as the main port response. It updates the cache line and changes its cache state from SI_to_M to M. The CC_RDOW transaction ends.

Figure 86(b) CC_RDOW Race Condition, Proc1



13.4.5.2 Intport_writedata=0 Case (2), Proc1 Wins

When assuming `intport_writedata=0`, no response is needed and no blocking is needed.

An example of the race condition of CC_WB from Proc0 and CC_RDOW from Proc1 at the same memory address is shown in Figures 87(a) and 87(b). Figure 87(a) shows the case where Proc1 wins the arbitration at the coherent request serialize/select logic unit in the OCP coherent interconnect, event (B2) appears prior to event (A2), and event (B3) appears prior to event (A3). In this case, only the order of the event (B7) completes before the event (A6) starts, and no other ordering between Proc0 and Proc1 is maintained.

The behaviors related to the CC_WB request from Proc0 are shown in Figure 87(a).

(A1) Proc0 sends a Cache Coherent Write Back (CC_WB) command with data from its main request port.

(A2) The coherent request serialize/select logic unit in the OCP interconnect network selects the CC_WB command from Proc0 and sends this command with its MData to the target, Memory 0. Note that at this point, the CC_RDOW issued by Proc1 has already been sent to Memory0.

(A3) The coherent request serialize/select logic unit in the OCP interconnect network sends I_WB to the intervention request port of Proc0 as a self-intervention. Since I_WB is not required to be sent to other coherent masters as system intervention, the OCP interconnect network does not send I_WB as system intervention requests. Note that at this point, I_RDOW from Proc1 has already been sent to coherent masters.

(A4) OCP wrapper of the Memory0 translates the CC_WB request into an I_WB request and sends it to its intervention request port.

(A5) OCP wrapper of Memory0 sends I_WB from its intervention request port. The OCP coherence interconnect ignores and drops it.

(A6) Proc0 receives I_WB as a self intervention request from its intervention request port and checks its cache state at the requested address location. Since Proc0 already received I_RDOW as a system intervention from its intervention request port and updated its cache state, the cache state is I. Proc0 changes the cache state from I to MSI_to_I. (Note: the cache state can be I instead of MSI_to_I—this is an implementation-dependent choice.)

(A7) Proc0 sends its cache state “Invalid” to its OCP wrapper.

(A8) OCP wrapper of Proc0 translates the snoop response “Invalid” into “SResp OK, SCohState Invalid”, and sends it from its intervention response port.

(A9) SResp merge logic unit in the OCP coherence interconnect receives the intervention port response (A7), generates the intervention response “SResp OK, SCohState Invalid” for Memory0, and sends it to the intervention port of Memory0.

(A10) OCP wrapper of the Memory0 receives the response “SResp OK, SCohState Invalid” from its intervention response port, and it cancels “Memory Write” to Memory0.

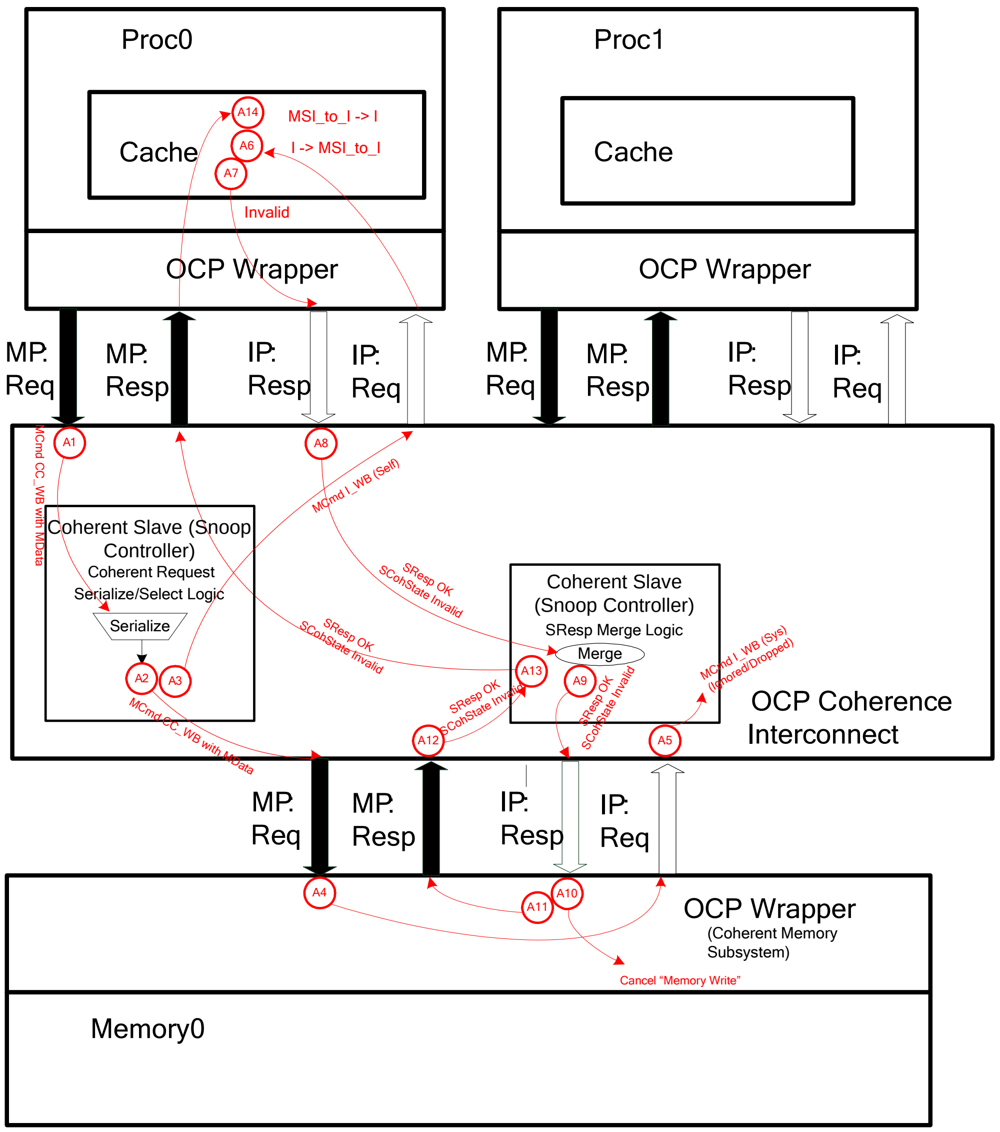
(A11) OCP wrapper of Memory0 generates the intervention response “SResp OK, SCohState Invalid” and sends it to its main response port. (Note a different implementation may generate “SResp DVA, SCohState Invalid”—the value of SResp is implementation dependent.)

(A12) OCP wrapper of Memory0 sends “SResp OK, SCohState Invalid” from its main response port. Since this is a write transaction, no data is sent from its main response port.

(A13) SResp merge logic unit in the OCP coherence interconnect receives the main port response from Memory0. It sends “SResp OK, SCohState Invalid” as the main port response to Proc0.

(A14) Proc0 receives “SResp OK, SCohState Invalid” as the main port response. Proc0 changes the cache state from MSI_to_I to I. The CC_WB transaction ends.

Figure 87(a) CC_WB Request from Proc0


Proc0
Cache
A14
MSI_to_I -> I
A6
I -> MSI_to_I
A7
Invalid
OCP Wrapper
Proc1
Cache
OCP Wrapper
MP: Req
MP: Resp
IP: Resp
IP: Req
A1
MCmd CC_WB with MData
A8
MCmd I_WB (Self)
Coherent Slave (Snoop Controller)
Coherent Request Serialize/Select Logic
Serialize
A2
A3
MCmd CC_WB with MData
SResp OK SCohState Invalid
SResp OK SCohState Invalid
Coherent Slave (Snoop Controller)
SResp Merge Logic
Merge
A13
A9
A12
SResp OK SCohState Invalid
SResp OK SCohState Invalid
MCmd I_WB (Sys) (Ignored/Dropped)
A5
OCP Coherence Interconnect
A4
A11
A10
OCP Wrapper
(Coherent Memory Subsystem)
Cancel "Memory Write"
Memory0

The behaviors related to the CC_RDOW request from Proc1 are shown in Figure 87(b).

(B1) Proc1 sends a Cache Read Own (CC_RDOW) command from its main request port.

(B2) The coherent request serialize/select logic unit in the OCP coherence interconnect network selects the CC_RDOW command from Proc1 and sends this command to the target, Memory 0.

(B3) The coherent request serialize/select logic unit sends I_RDOW as a self intervention request to the intervention request port of the initiator, Proc1. It also sends I_RDOW as system intervention requests to all other coherent masters, including Proc0.

(B4) OCP wrapper of Memory0 receives the CC_RDOW command from its main request port, and sends "Memory Read" speculatively to Memory0. Memory0 executes the read.

(B5) OCP wrapper of the Memory0 translates the CC_RDOW request into an I_RDOW request and sends it to its intervention request port.

(B6) OCP wrapper of Memory0 sends I_RDOW from its intervention request port. The OCP coherence interconnect ignores and drops it.

(B7) Proc0 receives I_RDOW as a system intervention request from its intervention request port and checks its cache state at the requested address location. Proc0 changes the cache state from M to I.

(B8) Proc0 send its cache state "Modified" to its OCP wrapper with the data of the requested cache line.

(B9) OCP wrapper of Proc0 translates the snoop response "Modified" into "SResp DVA with SData, SCohState Modified", and sends it from its intervention response port.

(B10) Proc1 receives I_RDOW as a self intervention request from its intervention request port and checks its cache state at the requested address location. Proc1 changes the cache state from I to SI_to_M.

(B11) Proc1 sends its cache state "Invalid" to its OCP wrapper.

(B12) OCP wrapper of Proc1 translates the snoop response "Invalid" into "SResp OK, SCohState Invalid", and send it from its intervention response port.

(B13) SResp merge logic unit in the OCP coherence interconnect receives the intervention port responses (B9) and (B12), generates the intervention response "SResp DVA with SData, SCohState Modified" for the coherent memory system, and sends it to the intervention port of Memory0. (Note: If the main port response of Memory0 is ignored, Memory0 does not need the data of CC_RDOW, because CC_RDOW is issued by "store miss" case and the data will be updated in the initiator.)

(B14) SResp merge logic unit in the OCP coherence interconnect generates the intervention response "SResp DVA with SData, SCohState Modified" for the initiator, Proc1, and sends it to the main port of Proc1. (Note that the OCP interconnect may skip (B14) and instead send the modified data at (B17). This is an implementation-dependent choice.)

(B15) Proc1 updates its cache and changes the cache state from SI_to_M to M.

(B16) OCP wrapper of Memory0 receives the intervention response "SResp DVA with SData, SCohState Modified" and sends it to its main response port.

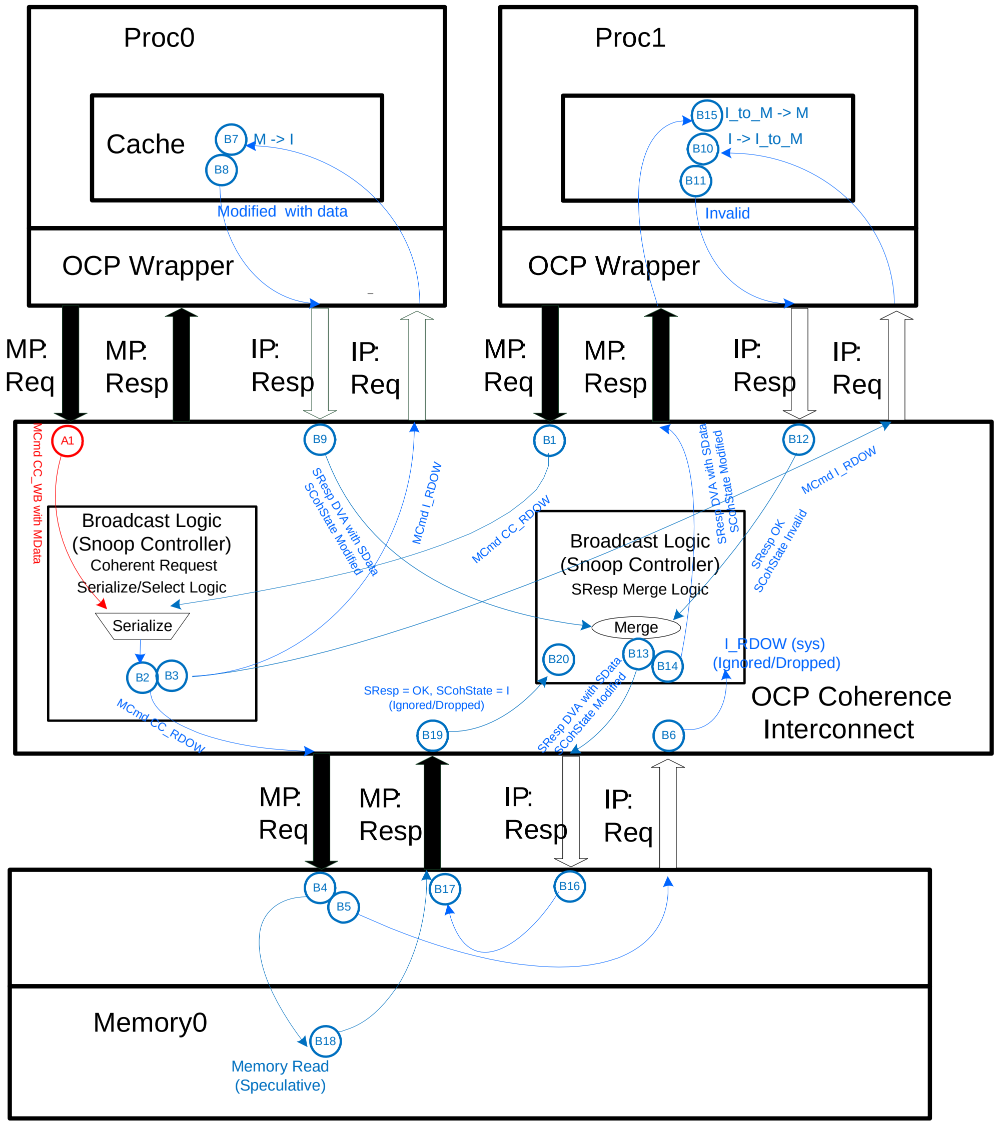
(B17) Main response port of the OCP wrapper of Memory0 waits for the read data.

(B18) Memory0 sends the read data to its main response port.

(B19) The OCP wrapper of Memory0 sends “SResp OK, SCohState Modified” as the main port response. (Note that this response is implementation-dependent: the OCP wrapper of Memory0 may also send “SRespDVA with SData, SCohState Modified” from its main response port for the OCP coherence interconnect to send to Proc1, instead of (B14).)

(B20) SResp merge logic unit in the OCP coherence interconnect receives the main port response from Memory0. It ignores/drops the response. The CC_RDOW transaction ends.

Figure 87(b) CC_RDOW Request from Proc1


Proc0
Cache
B7 M -> I
B8
Modified with data
OCP Wrapper
Proc1
B15 I_to_M -> M
B10 I -> I_to_M
B11
Invalid
OCP Wrapper
MP: Req
MP: Resp
IP: Resp
IP: Req
A1
MCmd CC_WB with MData
Broadcast Logic (Snoop Controller)
Coherent Request Serialize/Select Logic
Serialize
B2
B3
MCmd CC_RDOW
B9
SResp DVA with SData SCohState Modified
MCmd I_RDOW
B1
MCmd CC_RDOW
SResp DVA with SData SCohState Modified
B12
MCmd I_RDOW
SResp OK SCohState Invalid
Broadcast Logic (Snoop Controller)
SResp Merge Logic
Merge
B20
B13
B14
I_RDOW (sys) (Ignored/Dropped)
SResp = OK, SCohState = I (Ignored/Dropped)
B19
SResp DVA with SData SCohState Modified
B6
OCP Coherence Interconnect
B4
B5
B17
B16
Memory0
B18
Memory Read (Speculative)

13.4.5.3 Intport_writedata=1 Case (1), Proc0 Wins

There are two approaches for the implementer to handle this race condition:

1. The requestor waits for self intervention and for response. The home agent is simple: a CC_WB command has no effect on the state of the cache line until the reception of a self intervention. When this happens, the intervention port is blocked until the reception of a response which invalidates the status of the cache line.

2. When the requestor sends a CC_WB command, it commits at once (cache line state transitions to I when generating CC_WB) and does not wait for a self intervention or response.In this case, the coherence master implementation must be more complex than in the first approach.

   The coherent slave (snoop controller), which will be able to detect the race, drops the memory's response (i.e., it ignores stale data) and re-reads the data from memory (by issuing a second request) once the processor with the up-to-date copy writes back.

Since the implementation of the second approach requires complex mechanisms and careful design, this document only shows examples of the first approach.

The behaviors of the coherent masters (processors), interconnect, and memory subsystem are based on Tables 55–57 and Figures 80–85.

The behaviors related to Proc0 CC_WB are shown in Figure 88(a).

(A1) Proc0 sends a Cache Coherent Write Back (CC_WB) command. Since `intport_writedata=1`, Proc1 does not send modified data.

(A2) The coherent request serialize/select logic unit in the interconnection network selects the CC_WB command from Proc0 and sends this command to the target, Memory 0.

(A3) The coherent request serialize/select logic unit in the interconnection network sends I_WB to the intervention request port of Proc0 as a self-intervention. Since I_WB is not required to be sent to other coherent masters as a system intervention, the OCP coherence interconnect network does not send I_WB as system intervention requests.

(A4) OCP wrapper of Memory0 translates the CC_WB request into an I_WB request and sends it to its intervention request port.

(A5) OCP wrapper of Memory0 sends I_WB from its intervention request port. The OCP coherence interconnect ignores and drops it.

(A6) Proc0 receives I_WB as a self intervention request from its intervention request port and checks its cache state at the requested address location. Proc0 changes the cache state from M to MS_to_I.

(A7) Proc0 sends its cache state "Modified", and also sends the modified data of the requested cache line to its OCP wrapper.

(A8) OCP wrapper of Proc0 translates the snoop response "Modified" into "SResp DVA with SData, SCohState Invalid," and sends it from its intervention response port. (Note: this response is implementation-dependent: SCohState can also be "Modified" as shown in Figure 86(a).)

(A9) SResp merge logic unit in the OCP coherence interconnect receives the intervention port response (A8), generates the intervention response "SResp DVA with SData, SCohState Invalid" for Memory0, and sends it to the intervention port of Memory0.

(A10) OCP wrapper of the Memory0 receives the response "SResp DVA with SData, SCohState Invalid" from its intervention response port, and sends "Memory Write" to Memory0. Memory0 executes the write.

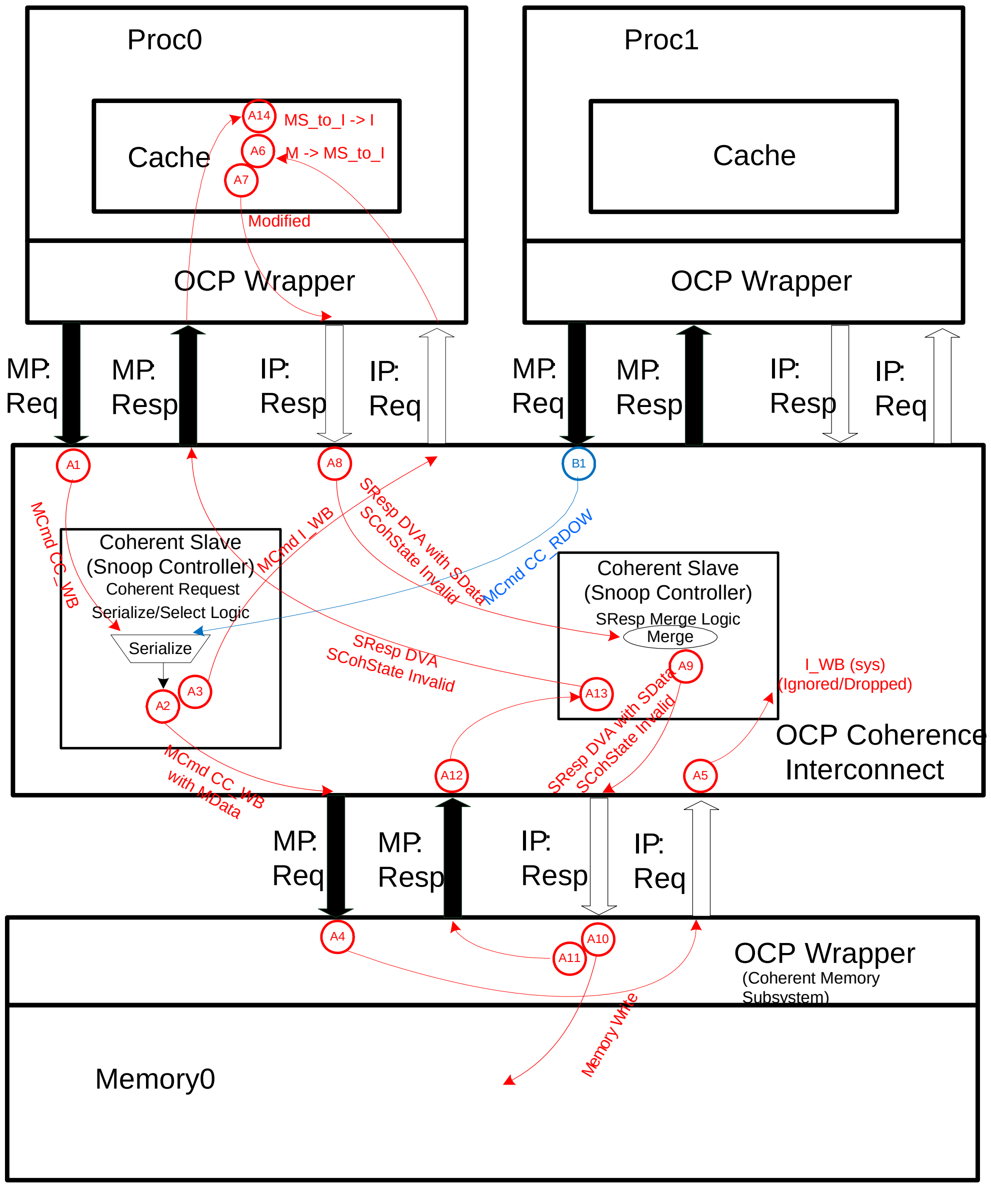
(A11) OCP wrapper of Memory0 generates the intervention response "SResp DVA, SCohState Invalid" and sends it to its main response port.

(A12) OCP wrapper of Memory0 sends "SResp DVA, SCohState Invalid" from its main response port. Since this is a write transaction, no data is sent from its main response port.

(A13) SResp merge logic unit in the OCP coherence interconnect receives the main port response from Memory0. It sends "SResp DVA, SCohState Invalid" as the main port response to Proc0.

(A14) Proc0 receives "SResp DVA, SCohState Invalid" as the main port response, and changes its cache state from MS_to_I to I. The CC_WB transaction ends.

Figure 88(a) CC_WB Request, Proc0



The behaviors related to the "CC_RDOW" request from Proc1 are shown in the Figure 88(b).

(B1) Proc1 sends a Cache Read Own (CC_RDOW) command from its main request port.

(B2) The coherent request serialize/select logic unit in the OCP coherence interconnection network selects the CC_RDOW command from Proc1 and sends this command to the target, Memory 0. Note that at this point, the CC_WB command issued by Proc0 has already been sent to Memory0.

(B3) The coherent request serialize/select logic unit sends I_RDOW as a self intervention request to the intervention request port of the initiator, Proc1. It also sends I_RDOW as system intervention requests to all other coherent masters, including Proc0. Note that at this point, I_WB from Proc0 has already been sent to Proc0.

(B4) OCP wrapper of the Memory0 receives the CC_RDOW command from its main request port, and sends “Memory Read” speculatively to Memory0. Memory0 executes the read.

(B5) OCP wrapper of the Memory0 translates the CC_RDOW request into an I_RDOW request and sends it to its intervention request port.

(B6) OCP wrapper of Memory0 sends I_RDOW from its intervention request port. The OCP coherence interconnect ignores and drops this request.

(B7) Proc0 receives I_RDOW as a system intervention request from its intervention request port and checks its cache state at the requested address location. Since the cache state is I, Proc0 does not change the cache state.

(B8) Proc0 send its cache state “Invalid” to its OCP wrapper.

(B9) OCP wrapper of Proc0 translates the snoop response “Invalid” into “SResp OK, SCohState Invalid”, and sends it from its intervention response port.

(B10) Proc1 receives “I_RDOW” as a self intervention request from its intervention request port and checks its cache state at the requested address location. Proc1 changes the cache state from I to SI_to_M.

(B11) Proc1 sends its cache state “Invalid” to its OCP wrapper.

(B12) OCP wrapper of Proc1 translates the snoop response “Invalid” into “SResp OK, SCohState Invalid”, and sends it from its intervention response port.

(B13) SResp merge logic unit in the OCP coherence interconnect receives the intervention port responses (B9) and (B12), generates the intervention response “SResp OK, SCohState Invalid” for the coherent memory system, and sends it to the intervention port of Memory0.

(B14) OCP wrapper of Memory0 receives the intervention response “SResp OK, SCohState Invalid” and sends it to its main response port.

(B15) Main response port of the OCP wrapper of Memory0 waits for the read data.

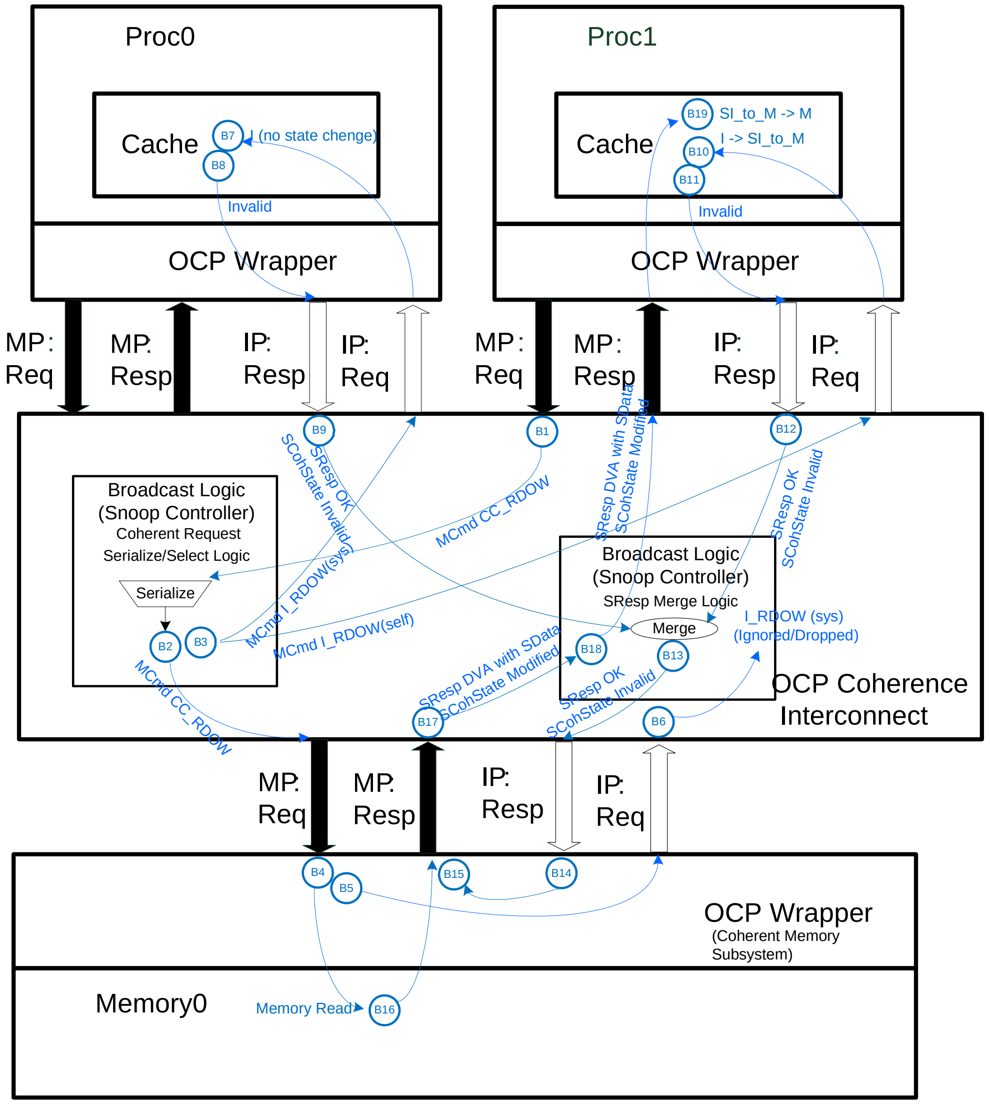
(B16) Memory0 sends the read data to its main response port.

(B17) The OCP wrapper of Memory0 sends “SResp DVA with SData, SCohState Modified” as the main port response.

(B18) SResp merge logic unit in OCP coherence interconnect receives the main port response from Memory0. It sends “SResp DVA with SData, SCohState Modified” as the main port response to Proc1.

(B19) Proc1 receives "SResp DVA with SData, SCohState Modified" as the main port response. It updates the cache line and changes its cache state from SI_to_M to M. The CC_RDOW transaction ends.

Figure 88(b) CC_RDOW Request, Proc1


Proc0
Cache
B7 I (no state chenge)
B8
Invalid
OCP Wrapper
MP: Req
MP: Resp
IP: Resp
IP: Req
Proc1
Cache
B19 SI_to_M -> M
I -> SI_to_M
B10
B11
Invalid
OCP Wrapper
B9 SResp OK SCohState Invalid
B1
MCmd CC_RDOW
SResp DVA with SData SCohState Modified
B12 SResp OK SCohState Invalid
Broadcast Logic (Snoop Controller) Coherent Request Serialize/Select Logic
Serialize
B2
B3
MCmd I_RDOW(sys)
MCmd I_RDOW(self)
MCmd CC_RDOW
Broadcast Logic (Snoop Controller) SResp Merge Logic
Merge
B18
B13
I_RDOW (sys) (Ignored/Dropped)
SResp DVA with SData SCohState Modified
B17
SResp OK SCohState Invalid
B6
OCP Coherence Interconnect
B4
B5
B15
B14
OCP Wrapper (Coherent Memory Subsystem)
Memory0
Memory Read
B16

13.4.5.4 Intport_writedata=1 Case (2), Proc1 Wins

The behaviors related to Proc0 CC_WB are shown in Figure 89(a).

(A1) Proc0 sends a Cache Coherent Write Back (CC_WB) command. Since `intport_writedata=1`, Proc1 does not send modified data.

(A2) The coherent request serialize/select logic unit in the OCP coherence interconnect network selects the CC_WB command from Proc0 and sends this command to the target, Memory 0. Note that at this point, the CC_RDOW command from Proc1 has already been sent to Memory0.

(A3) The coherent request serialize/select logic unit in the OCP coherence interconnect network sends I_WB to the intervention request port of Proc0 as a self-intervention. Since I_WB is not required to be sent to other coherent masters as a system intervention, the OCP coherence interconnect network does not send I_WB as system intervention requests. Note that at this point, I_RDOW from Proc1 has already been sent to coherent masters.

(A4) OCP wrapper of Memory0 translates the CC_WB request into an I_WB request and sends it to its intervention request port.

(A5) OCP wrapper of Memory0 sends I_WB from its intervention request port. The OCP coherence interconnect ignores and drops this request.

(A6) Proc0 receives I_WB as a self intervention request from its intervention request port and checks its cache state at the requested address location. Since Proc0 already received I_RDOW as a system intervention from its intervention request port and updated its cache state, the cache state is I. Proc0 changes the cache state from I to MSI_to_I.

(A7) Proc0 sends its cache state, “Invalid,” to its OCP wrapper.

(A8) OCP wrapper of Proc0 translates the snoop response “Invalid” into “SResp OK, SCohState Invalid”, and sends it from its intervention response port.

(A9) SResp merge logic unit in the OCP coherence interconnect receives the intervention port response (A7), generates the intervention response “SResp OK, SCohState Invalid” for Memory0, and sends it to the intervention port of Memory0.

(A10) OCP wrapper of the Memory0 receives the response “SResp OK, SCohState Invalid” from its intervention response port, and cancels “Memory Write” to Memory0.

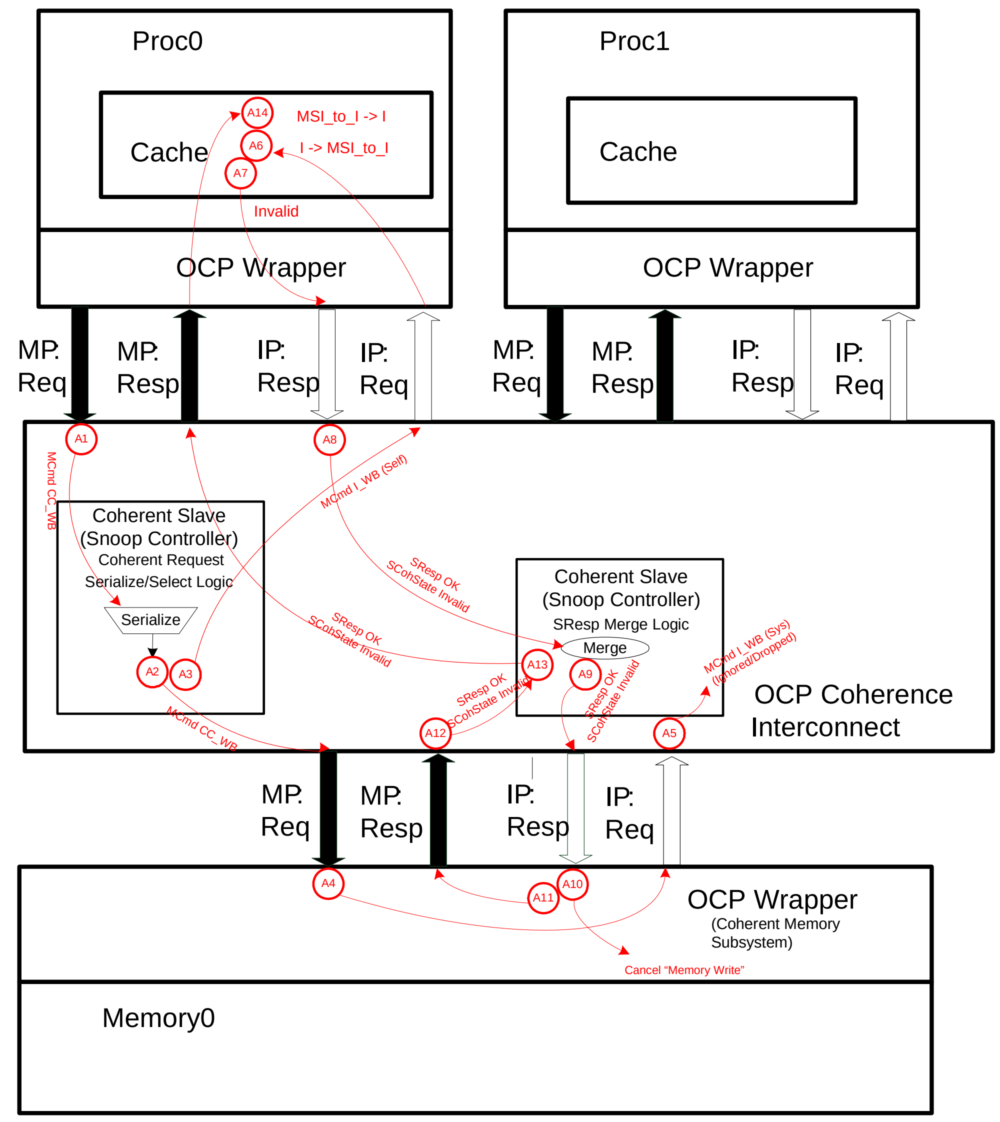
(A11) OCP wrapper of Memory0 generates the intervention response “SRespOK, SCohState Invalid” and sends it to its main response port. (Note that the value of SResp is implementation dependent: a different implementation may generate the response “SResp DVA, SCohState Invalid”.)

(A12) OCP wrapper of Memory0 sends “SResp OK, SCohState Invalid” from its main response port. Since this is a write transaction, no data is sent from its main response port.

(A13) SResp merge logic unit in OCP coherence interconnect receives the main port response from Memory0. It sends “SResp OK, SCohState Invalid” as the main port response to Proc0.

(A14) Proc0 receives “SResp OK, SCohState Invalid” as the main port response. Proc0 changes the cache state from MSI_to_I to I. The “CC_WB” transaction ends.

Figure 89(a) CC_WB Request, Proc0



The behaviors related to the CC_RDOW request from Proc1 are shown in Figure 89(b).

(B1) Proc1 sends the Cache Read Own (CC_RDOW) command from its main request port.

(B2) The coherent request serialize/select logic unit in the OCP coherence interconnect network selects the CC_RDOW command from Proc1 and sends this command to the target, Memory 0.

(B3) The coherent request serialize/select logic unit sends I_RDOW as a self intervention request to the intervention request port of the initiator, Proc1. It also sends I_RDOW as system intervention requests to all other coherent masters, including Proc0.

(B4) OCP wrapper of Memory0 receives the CC_RDOW request from its main request port, and sends “Memory Read” speculatively to Memory0. Memory0 executes the read.

(B5) OCP wrapper of the Memory0 translates the CC_RDOW request into an I_RDOW request and sends it to its intervention request port.

(B6) OCP wrapper of Memory0 sends I_RDOW from its intervention request port. The OCP coherence interconnect ignores and drops it.

(B7) Proc0 receives I_RDOW as a system intervention request from its intervention request port and checks its cache state at the requested address location. Proc0 changes the cache state from M to I.

(B8) Proc0 send its cache state “Modified” to its OCP wrapper with the data of the requested cache line.

(B9) OCP wrapper of Proc0 translates the snoop response “Modified” into “SResp DVA with SData, SCohState Modified,” and sends it from its intervention response port.

(B10) Proc1 receives I_RDOW as a self intervention request from its intervention request port and checks its cache state of the requested address location. Proc1 changes the cache state from I to SI_to_M.

(B11) Proc1 sends its cache state “Invalid” to its OCP wrapper.

(B12) OCP wrapper of Proc1 translates the snoop response “Invalid” into “SResp OK, SCohState Invalid,” and sends it from its intervention response port.

(B13) SResp merge logic unit in the OCP coherence interconnect receives the intervention port responses (B9) and (B12), generates the intervention response “SResp DVA with SData, SCohState Modified” for the coherent memory system, and sends it to the intervention port of Memory0.

(B14) SResp merge logic unit in the OCP coherence interconnect generates the intervention response “SResp DVA with SData, SCohState Modified” for the initiator, Proc1, and sends it to the main port of Proc1. (Note that the OCP coherence interconnect can skip (B14), and the modified data can be sent at (B17). This is implementation dependent.)

(B15) Proc1 updates its cache and changes the cache state from SI_to_M to M.

(B16) OCP wrapper of Memory0 receives the intervention response “SResp DVA with SData, SCohState Modified” and sends it to its main response port.

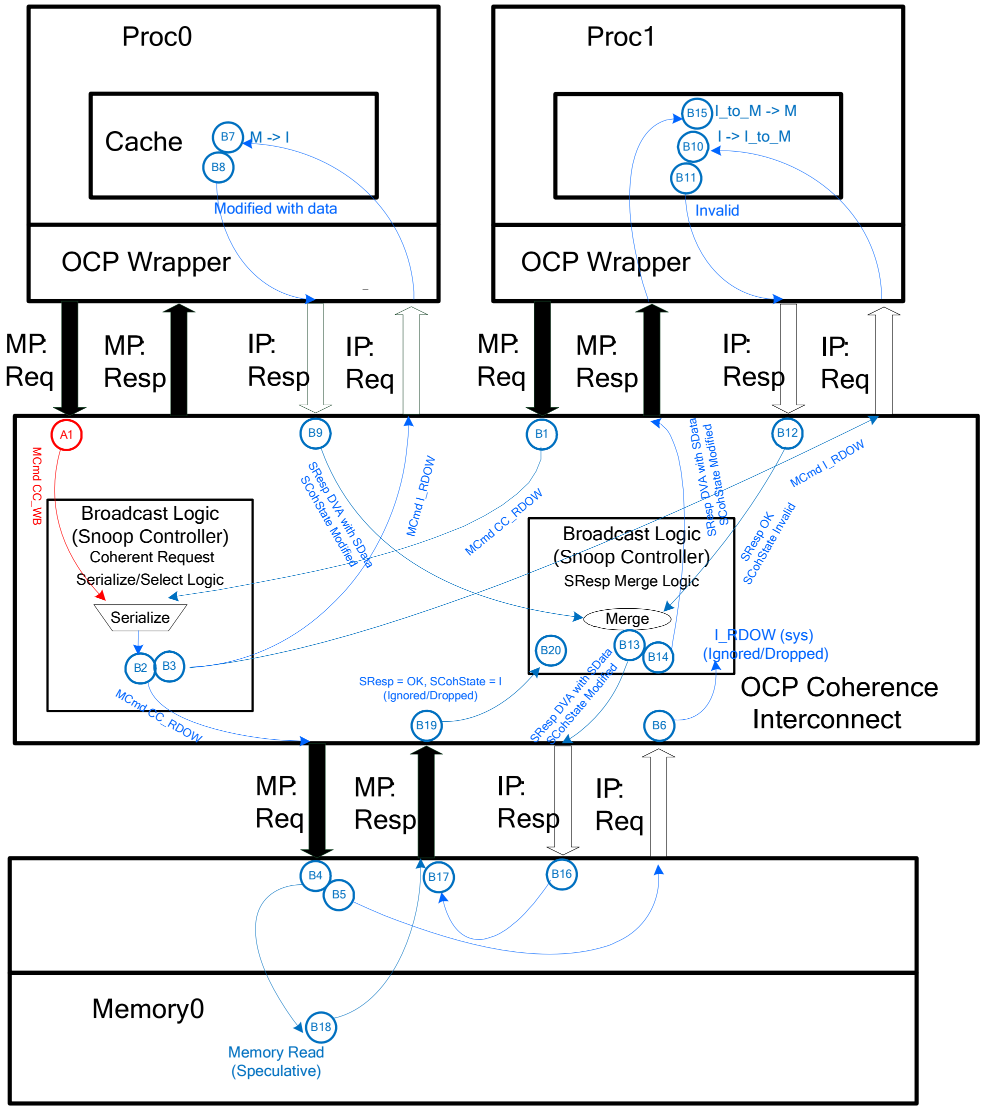
(B17) Main response port of the OCP wrapper of Memory0 waits for the read data.

(B18) Memory0 sends the read data to its main response port.

(B19) The OCP wrapper of Memory0 sends “SResp OK, SCohState Modified” as the main port response. (Note that this is implementation dependent: the OCP wrapper of Memory0 may also send “SRespDVA with SData, SCohState Modified” from its main response port and have the OCP coherence interconnect send it to Proc1 at this time, instead of at (B14).)

(B20) SResp merge logic unit in OCP coherence interconnect receives the main port response from Memory0. It ignores/drops the response. The CC_RDOW transaction ends.

Figure 89(b) CC_RDOW Request, Proc1


Proc0
Cache
B7 M -> I
B8
Modified with data
OCP Wrapper
Proc1
B15 I_to_M -> M
B10 I -> I_to_M
B11
Invalid
OCP Wrapper
MP: Req
MP: Resp
IP: Resp
IP: Req
MP: Req
MP: Resp
IP: Resp
IP: Req
A1
MCmd CC_WB
B9
SResp DVA with SData SCohState Modified
MCmd I_RDOW
B1
MCmd CC_RDOW
SResp DVA with SData SCohState Modified
B12
MCmd I_RDOW
SResp OK SCohState Invalid
Broadcast Logic (Snoop Controller)
Coherent Request Serialize/Select Logic
Serialize
B2
B3
MCmd CC_RDOW
Broadcast Logic (Snoop Controller)
SResp Merge Logic
Merge
B20
B13
B14
I_RDOW (sys) (Ignored/Dropped)
SResp = OK, SCohState = I (Ignored/Dropped)
B19
SResp DVA with SData SCohState Modified
B6
OCP Coherence Interconnect
MP: Req
MP: Resp
IP: Resp
IP: Req
B4
B5
B17
B16
Memory0
B18
Memory Read (Speculative)

14 Timing Guidelines

To provide core timing information to system designers, characterize each core into one of the following timing categories:

- Level0 identifies the core interface as having been designed without adhering to any specific timing guidelines.
- Level1 timing represents conservative interface timing.
- Level2 represents high performance interface timing.

One category is not necessarily better than another. The timing categories are an indication of the timing characteristics of the core that allow core designers to communicate at a very high level about the interface timing of the core. Table 63 represents the inter-operability of two OCP interfaces.

Table 63 Core Interface Compatibility

	Level0	**Level1**	**Level2**
Level0	X	X	X
Level1	X	V	V
Level2	X	V	V*

X	no guarantee
V	guaranteed inter-operability with possible performance loss (extra latency)
V*	high performance inter-operability but some minor changes may be required

The timing guidelines apply to dataflow and sideband signals only. There is no timing guideline for the scan and test related signals.

Timing numbers are specified as a percentage of the minimum supported clock-cycle (at maximum operating frequency). If a core is specified at 100MHz and the `c2qtime` is given as 30%, the actual `c2qtime` is 3ns.

14.1 Level0 Timing

Level0 timing indicates that the core developer has not followed any specific guideline in designing the core interface. There is no guarantee that the interface can operate with any other core interface. Inter-operability for the core will need to be determined by comparing timing specifications for two interfaces on a per-signal basis.

14.2 Level1 Timing

Level1 timing indicates that a core has been developed for minimum timing work during system integration. The core uses no more than 25% of the clock period for any of its signals, either at the input (`setuptime`) or at the output (`outputtime`). A core interface in this category must not use any of the combinational paths allowed in the OCP interface.

Since inputs and outputs each only use 25%, 50% of the cycle remains available. This means that a Level1 core can always connect to other Level1 and Level2 cores without requiring any additional modification.

14.3 Level2 Timing

Level2 timing indicates that a core interface has been developed for high performance timing. A Level2 compliant core provides or uses signals according to the timing values shown in Table 64. There are separate values for single-threaded and multi-threaded OCP interfaces. The number for each signal indicates the percentage of the minimum cycle time at which the signal is available, that is the `outputtime` at the output. `setuptime` at the input is calculated by subtracting the number given from the minimum cycle time. For example, a time of 30% indicates that the `outputtime` is 30% and the `setuptime` is 70% of the minimum clock period.

In addition to meeting the timing indicated in Table 64, a Level2 compliant core must not use any combinational paths other than the preferred paths listed in Table 65.

There is no margin between `outputtime` and `setuptime`. When using Level2 cores, extra work may be required during the physical design phase of the chip to meet timing requirements for a given technology/library.

Table 64 Level2 Signal Timing

Signal	Single-threaded Interface %	Multi-threaded Interface %	Multi-threaded Pipelined
Control, Status	25	25	25
ControlBusy, StatusBusy	10	10	10
ControlWr, StatusRd	25	25	25
Datahandshake Group (excluding MDataThreadID)	30	60	30
EnableClk	20	20	20
MDataThreadID	n/a	50	30
MRespAccept	50	75	50
MThreadBusy	10	10	50
MThreadID	n/a	50	30
Request Group (excluding MThreadID)	30	60	30
MReset_n, SReset_n	10	10	10
Response Group (excluding SThreadID)	30	60	30
SCmdAccept	50	75	50
SDataAccept	50	75	50
SDataThreadBusy	10	10	50
SError, SFlag, SInterrupt, MFlag, MError	40	40	40
SThreadBusy	10	10	50
SThreadID	n/a	50	30

Table 65 Allowed Combinational Paths for Level2 Timing (No Pipelining)

Core	From	To
Master	SThreadBusy	Request Group
	SThreadBusy SDataThreadBusy	Datahandshake Group
	Response Group	MRespAccept
Slave	MThreadBusy	Response Group
	Request Group	SCmdAccept and SDataAccept
	Datahandshake Group	SCmdAccept and SDataAccept

Table 66 Allowed Combinational Paths for Level2 Timing (Pipelined)

Core	From	To
Master	Request Group Datahandshake Group	SThreadBusy SDataThreadBusy
	Response Group	MRespAccept
Slave	Response Group	MThreadBusy
	Request Group	SCmdAccept and SDataAccept
	Datahandshake Group	SCmdAccept and SDataAccept

15 *OCP Profiles*

A strength of OCP is the ability to configure an interface to match a core's communication requirements. While the large number of configuration options makes it possible to fit OCP into many different applications, it also results in multiple implementation possibilities. This chapter provides profiles that capture the OCP features associated with standard communication functions. The pre-defined profiles help map the requirements of a new core to OCP configuration guidelines. The expected benefits include:

- Reduced risk of incompatibility when integrating OCP based cores originating from different providers
- Reduced learning curve in applying OCP for standard purposes
- Simplified circuitry needed to bridge an OCP based core to another communication interface standard
- Improved core maintenance
- Simplified creation of reusable core test benches

Profiles address only the OCP interface, with each profile consisting of OCP interface signals, specific protocol features, and application guidelines. For cores natively equipped with OCP interfaces, profiles minimize the number of interface and protocol options that need to be considered.

Two sets of OCP profiles are provided: profiles for new IP cores implementing native OCP interfaces and profiles that are targeted at designers of bridges between OCP and other bus protocols. Since the other bus protocols may have several implementation flavors that require custom OCP parameter sets, the bridging profiles are incomplete. The bridging profiles can be used with OCP serving as either a master or a slave.

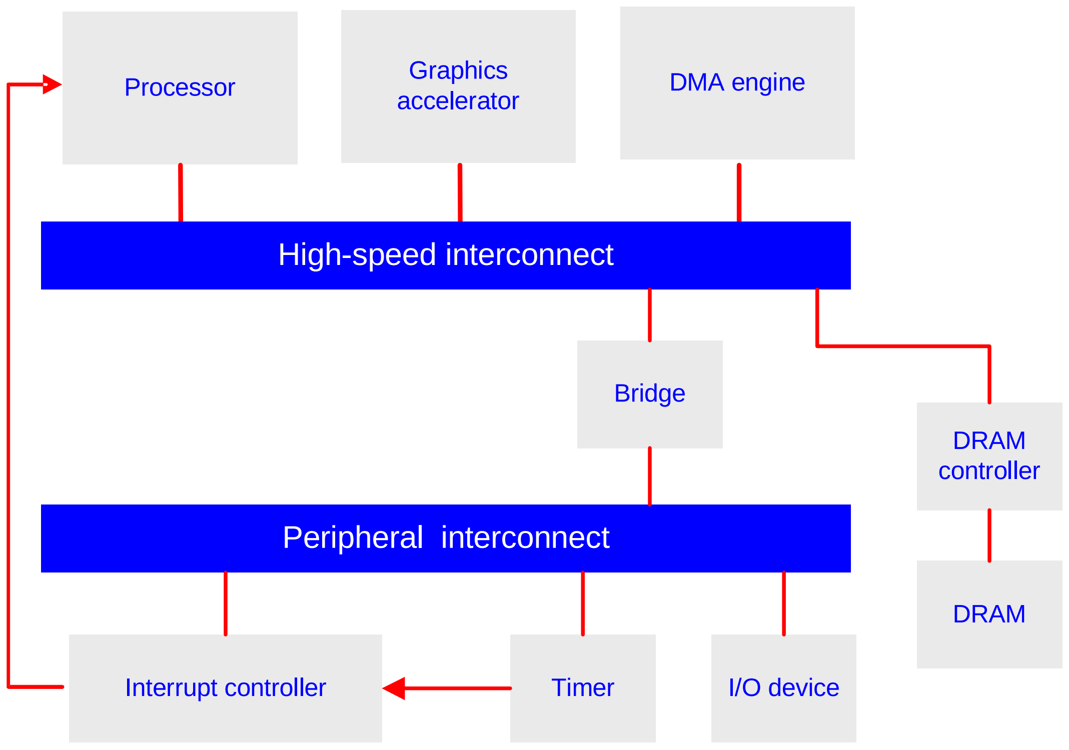
The native OCP profiles are designed for new IP cores implementing native OCP interfaces.

Consensus profiles have been jointly defined by users of OCP and are intended to supersede several of the existing profiles. The consensus profiles define a unified set of OCP interfaces for system houses, IP providers and EDA vendors.

Figure 90 depicts a system built out of diverse IP blocks. Many of the blocks can be characterized as peripherals and are often connected with a simple peripheral interconnect employing relaxed requirements for latency and throughput.

Connecting the peripheral interconnect to the high-speed interconnect through the use of a bridge component will likely increase system latency. The high-speed interconnect services the processor subsystem including processors, co-processors, DMAs, and memories. For these components high throughput is a requirement, so they frequently use more advanced communication schemes.

Figure 90 Native Profiles



15.1 Consensus Profiles

The profiles in this section define a unified set of OCP interfaces. The first is a simple slave profile that is targeted at peripheral modules and interconnects. The second and third profiles address components that require higher throughput, such as processor subsystem components.

15.1.1 Simple Slave

This profile is intended for peripheral slaves that favor simple implementation over high throughput. Interrupt controllers, timers, and I/O devices shown in Figure 90 are typical examples of this kind of slave. The signals and the parameters that control them are listed in Table 67. All signals with an *M* prefix are driven by the master while all signals with an *S* prefix are driven by the slave, with the possible exception of MReset_n and SReset_n depending on the system.

Table 67 Simple Slave Profile Signals and Parameters

Signal*	**Enabling Parameter**	**Width Parameter**	**Usage**
Clk	Required	Fixed	Clock input (page 14)
EnableClk	enableclk	Fixed	Enable interface clock input (page 14)
MReset	mreset 0/1	Fixed	Slave reset input =1 Master reset output is optional (page 27)
SReset	sreset 0/1	Fixed	Master reset input =1 Slave reset output is optional (page 28)
Request phase signals			
MAddr	addr	addr_wdth 32 max	Address of the transfer (byte address, aligned to the OCP word size)
MCmd	Required read_enable write_enable writenonpost_enable	Fixed	Command of the transfer Can be 0 for a master that issues only write operations Can be 0 for a master that only issues read operations or non-posted writes Can be 0 for a master that only issues read operations or posted writes
MData	mdata	data_wdth 16/32/64	Write data
MByteEn	byteen	data_wdth	Byte enable
SCmdAccept	cmdaccept	Fixed	Slave accepts the transfer and ends the request phase

Signal*	Enabling Parameter	Width Parameter	Usage
Response phase signals			
SData	sdata	data_wdth 16/32/64	Read data
SResp	resp	Fixed	Slave response to transfer
Additional parameters			
	endian little, big, both, neutral		All endianness options are allowed but the modules are required to state their endianness. Little endian is recommended (page 51)
	force_aligned = 1		Byte enables are power-of-two in size and aligned to that size (page 60)
	writeresp_enable = 0 or 1		Controls response to posted write (non-posted writes always provide a response).

* See Section 15.1.4 on page 332 for additional signals

Feature Set

The simple slave profile supports only single accesses, so no burst-related signals are used. The accepted commands are IDLE, RD, WR, and WRNP. Posted writes only return a response if `writeresp_enable` is enabled; non-posted writes always return a response. For non-posted writes the response must issue from the receiving slave. When responses are required for posted writes (`writeresp_enable` is enabled), any component between the master and the slave (e.g., an interconnect) can provide the response. This process represents a trade-off between the potential speed improvements of posted writes and the more reliable write completion and error tracking of the non-posted writes. The allowed responses for SResp are NULL, DVA, and ERR.

If present, the read and write data signals SData and MData must have the same width. To simplify the interface, the `force_aligned` parameter is set to 1, limiting byte enable patterns on the MByteEn signal to power-of-two in size and aligned to that size. A byte enable pattern of all 0s is legal. This means that the byte enable patterns generated by simple slave profile masters are force-aligned. In addition, slaves using this profile can assume that the incoming byte enable patterns are force-aligned. The size of the MByteEn is 2 bits when `data_wdth` is 16, 4 bits for a `data_wdth` of 32, and 8 bits for a `data_wdth` of 64. The allowed MByteEn values for a `data_wdth` of 32 are indicated by the shaded rows in Table 68.

Table 68 MByteEn Patterns for data_wdth = 32 in Simple Slave Profile

MByteEn[3]	MByteEn[2]	MByteEn1]	MByteEn[0]
0	0	0	0
0	0	0	1
0	0	1	0

MByteEn[3]	MByteEn[2]	MByteEn1]	MByteEn[0]
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Since resets are usually handled outside of OCP signaling, no module is required to have a reset output, but every module interface needs to have a reset input. In the OCP signal naming scheme, this requirement means that all masters must have SReset enabled and all slaves an MReset input. It is optional for slaves to have an SReset signal and for masters to have an MReset signal. If the cores do not drive the reset signals, they need to be driven by a system-component that generates the reset. The Clk and EnableClk signals are required inputs in both masters and slaves and they are driven by a third entity (neither the masters nor the slaves).

Slaves must be able to support the entire feature set defined in this profile. Masters do not need to be able to issue all the commands since only one WR, RD or WRNP is required. If masters only issue read commands (`write_enable` and `writenonpost_enable` parameters set to 0), they can omit the MData signal and responses to writes (`writeresp_enable` has to be 0 in the master parameter list). If a master issues only write commands, the SData signal can be omitted. Using these options does not compromise interoperability with Simple Slave Profile slaves.

Interoperability Issues

A slave that can accept commands on every cycle can permanently tie SCmdAccept high. When configured in this fashion unsupported or otherwise problematic commands are accepted, so all slaves would need to accept all commands. In case of errors, the master can be notified with the SResp signal (ERR).

15.1.2 High Speed Profile

The high-speed profile is intended for subsystem components that require high throughput, for example, processors, co-processors, DMA engines, and memory controllers. The signals for this interface are listed in Table 69. This profile adds the burst-related signals MBurstSeq, MBurstLength, MReqLast, and SRespLast to improve throughput and the MRespAccept signal to enable response-phase flow control. All signals with an M prefix are driven by the master while all signals with an S prefix are driven by the slave, with the possible exception of MReset_n and SReset_n depending on the system.

Table 69 High Speed Profile Signals and Parameters

Signal*	**Enabling Parameter**	**Width Parameter**	**Usage**
Clk	Required	Fixed	Clock input (page 14)
EnableClk	enableclk	Fixed	Enable interface clock input (page 14)
MReset	mreset 0/1	Fixed	Slave reset input =1 Master reset output is optional (page 27)
SReset	sreset 0/1	Fixed	Master reset input =1 Slave reset output is optional (page 28)
Request phase signals			
MAddr	addr	addr_wdth 64 max	Address of the transfer (byte address, aligned to the OCP word size) driven by the master (page 14)
MCmd	Required read_enable write_enable writenonpost_enable	Fixed	Command of the transfer driven by the master (page 15) Can be 0 for a master that issues only write operations Can be 0 for a master that only issues read operations or non-posted writes Can be 0 for a master that only issues read operations or posted writes
MData	mdata	data_wdth 32/64/128	Write data driven by the master (page 15)
MByteEn	byteen	data_wdth	Byte enable driven by the master (page 17)
MBurstSeq	burstseq burstseq_incr_enable burstseq_strm_enable burstseq_wrap_enable	Fixed	The address sequence of the burst, driven by the master (page 21)
MBurstLength	burstlength	burstlength_wdth 6 max	Length of the burst, driven by the master (page 20)
MReqLast	reqlast	Fixed	Last request in the burst, driven by the master (page 22)

Signal*	Enabling Parameter	Width Parameter	Usage
SCmdAccept	cmdaccept	Fixed	Slave accepts the transfer and ends the request phase, driven by the slave (page 15)
Response phase signals			
MRespAccept	respaccept	Fixed	Master accepts the transfer and ends the response phase, driven by the master
SData	sdata	data_wdth 32/64/128	Read data driven by the slave (page 16)
SResp	resp	Fixed	Slave response to transfer driven by the slave (page 16)
SRespLast	resplast	Fixed	Indicates the last response in the burst, driven by the slave (page 22)
Additional parameters			
	endian little, big, both, neutral		Modules are required to state their endianness. Little endian is recommended (page 51)
	force_aligned		Byte enables are power-of-two in size and aligned to that size (page 60)
	writeresp_enable		Posted writes expect a response. Can be 0 for a master that only issues read operations or non-posted write.

* See Section 15.1.4 on page 332 for additional signals

For this profile the `burstprecise` parameter is zero, so the MBurstPrecise signal is not used in the interface and all bursts are precise.

Feature Set

While the High-Speed Profile shares many features with the Simple Slave Profile, the force aligned requirement is eliminated since the attached modules can afford more complex interface logic. Two additional features are multiple-request multiple-data (MRMD) type bursts and the response accept signal (MRespAccept). MRespAccept grants capability to the master to block the response flow from the slave if it cannot process new responses anymore.

Bursts offer higher transfer efficiencies when compared to single transfers by introducing atomicity for multiple associated requests. This atomicity ensures that requests in the same transaction, i.e. with spatial locality, are issued back-to-back. This behavior is crucial when accessing an SDRAM memory to attain high throughputs. Since the length of the burst is transmitted in the beginning (with MBurstLength) and all bursts are precise (MBurstLength value remains constant over the whole burst), further optimizations are possible in the scheduling and arbitration processes. The address sequence of the burst is provided by MBurstSeq. Allowed sequences are:

Incrementing (INCR)
: The address is incremented with the OCP word size for each transfer

Wrapping (WRAP)
: Like incrementing burst but it does not have to start from the first address and the address wraps at the address boundary defined by MBurstLength*OCP word size

Streaming (STRM)
: The address remains constant over the whole burst

MBurstSeq and MBurstLength provide the information needed to generate and receive MRMD bursts. Additional information can help simplify interconnect and slave design. OCP offers framing signals that can be used by the master to identify the last request (MReqLast) and by the slave to identify the last response (SRespLast) in the burst. These additional framing signals are also part of this profile.

Slaves must support the entire feature set defined in this profile. Masters need not be able to issue all the commands since only one WR, RD or WRNP is required. If masters only issue read commands (`write_enable` and `writenonpost_enable` parameters set to 0), they can omit the MData signal and responses to writes (`writeresp_enable` has to be 0 in the master parameter list). If a master issues only write commands, SData can be omitted. Masters must support at least one of the burst addressing modes.

Interoperability Issues

The force aligned requirement of the simple slave profile and the MRespAccept signal and burst features of the high-speed profile present some interoperability problems between the interfaces. A bridge or some other component linking simple slave and high-speed profile interfaces needs to break transactions with misaligned byte enables (coming from the high-speed profile to the simple slave profile) into transactions with legal byte enable patterns. A similar process needs to be followed for burst accesses coming from a high-speed profile master to a simple slave profile slave. The MRMD nature of high-speed profile bursts means that the bridge can ignore the burst related signals on the request side, but needs to generate an SRespLast. In addition, if Master is High-Speed Profile, the bridge may need to limit the number of outstanding requests on the interface or to buffer Simple Slave Profile slave responses in case the master de-asserts MRespAccept. If Master is Simple Slave Profile, and the slave is High-Speed Profile, the bridge may tie MRespAccept asserted.

15.1.3 Advanced High-Speed Profile

The Advanced High-Speed profile, like the High-Speed Profile, is also aimed at processor subsystem modules that require high throughput, with added capabilities compared to the High-Speed Profile. The Advanced High-Speed Profile is targeted at systems which process large amounts of real-time, block-based data, such as high definition digital TV sets. Signals in the Advanced High-Speed Profile interface are listed in Table 70. The Advanced High-Speed Profile adds the following signals relative to the High-Speed Profile: MDataValid, MBlockHeight, MBlockStride, MBurstSingleReq, SDataAccept, MDataLast, MDataRowLast, MDataByteEn, and SRespRowLast. MReqLast is not used in the Advanced High-Speed Profile. In general, all signals with an *M* prefix are driven by the master while all signals with an *S* prefix are driven by the slave. Exceptions to this rule are made for the MReset_n and SReset_n signals, which can be driven by other entities, depending on the system.

Table 70 Advanced High-Speed Profile Signals

Name	**Width**	**Usage**
Clk	1	Clock input to masters and slaves
EnableClk	1	Enable interface clock input to masters and slaves
MReset (Slaves only)	1	Reset input to slaves
SReset (Masters only)	1	Reset input to masters
Request phase signals		
MAddr	64 (max.)	Address of the transfer (byte address, aligned to the OCP word size)
MCmd	3	Command of the transfer
MData	32, 64, 128	Write data
MDataValid	1	Write data valid
MByteEn	1	Byte enable
MDataByteEn	1	Datahandshake phase write byte enables
MBlockHeight	6 (max)	Height of 2D block burst
MBlockStride	32 (max)	Address offset between 2D block rows
MBurstSeq	3	The address sequence of the burst
MBurstLength	6 (max)	Burst length
MBurstSingleReq	1	Burst uses single request, multiple data (SRMD) protocol
MDataLast	1	Last write data in burst
MDataRowLast	1	Last write data in row
SCmdAccept	1	Slave accepts the transfer and ends the request phase
Response phase signals		
SData	32, 64, 128	Read data
SDataAccept	1	Slave accepts write data
SResp	2	Slave response to transfer
SRespLast	1	Indicates the last response in the burst
SRespRowLast	1	Last response in row
MRespAccept	1	Master accepts the transfer and ends the response phase

Feature Set

With the exception of MReqLast, which is not present in this profile, the Advanced High-Speed Profile includes all features already available in the High-Speed Profile and includes a number of additional features, discussed below.

- Two-dimensional block bursts can be used. This burst sequence allows for efficient transfers of block-based data, such as macro-blocks in a picture or a video stream. The signals used for this feature are: MBlockHeight, MBlockStride, and SRespRowLast. MBurstSeq can also use the BLCK value on top of the already allowed INCR, WRAP, and STRM encodings.
- The exclusive-OR (XOR) burst sequence is supported. This address sequence is required for optimized accesses to some high-performance memories such as XDR. There are no additional signals for this feature; only the added allowed encoding for MBurstSeq.
- Exclusive reads (MCmd set to RDEX) are supported. Exclusive reads provide strong synchronization to access shared resources. There are no additional signals for this feature; only the added allowed encoding for MCmd.
- A data handshake phase is included to optimize flow control on the write transactions. Using this third phase allows the master to decouple the generation of the command from the actual emission of data. The signals used for this feature are MDataValid, SDataAccept, MDataByteEn, and MDataLast.

Single request, multiple data (SRMD) transactions are recommended for high bandwidth data transfers. Each transaction is associated to a single command independent of the number of data phases in the burst so the use rate of the command and address lines can be reduced. It is strongly recommended for a master that supports the Advanced High-Speed Profile to only issue SRMD commands and to tie its MBurstSingleReq signal to 1. For compatibility with the High-Speed Profile, slaves that support the Advanced High-Speed Profile are required to support both SRMD and MRMD transactions. Slave devices can detect the type of transaction in progress based on the MBurstSingleReq signal. MReqLast is not present in the Advanced High-Speed Profile since the usage of SRMD transactions makes it irrelevant—MReqLast would always be asserted.

Using RDEX

RDEX is used as a synchronization primitive required for hardware support of semaphores or read/modify/write sequences. These sequences are required when a processor needs to read from a shared resource (registers or memories) and then write to the same location in an atomic manner.

Example: Semaphore is used to control a shared memory resource. When a processor needs access to this memory, it first reads from a semaphore register and writes 1 to the same location using a RDEX/WRNP transaction. If the value read was 1, the semaphore was already taken and the processor cannot access the resource. In this case the value in the register is still 1. If the value read was 0, the semaphore was clear, and the processor can now access the memory. After the RDEX/WRNP, the register value is 1 and other processors cannot access the memory. The atomicity of the transaction guarantees that no other processor may access the semaphore between the read and the write in the transaction, and that only one master can have control of the semaphore at any given time.

For transaction atomicity to be guaranteed in the system, RDEX must be supported at each arbitration point, starting from the processor and continuing down to the last arbitration point before any system target which can be used as a protected shared resource. For a regular target, e.g., a simple peripheral, RDEX support is not strictly required since the last arbitration point takes care of the atomicity requirement, and RDEX can be translated into a regular RD in this case. For some more complex targets, e.g., an interconnect or a multi-threaded target, RDEX support is mandatory because at least one arbitration point exists after this OCP interface.

To correctly support software synchronization primitives, the hardware is required to keep the RDEX/WRNP transaction atomic, i.e., ensure that no other command to the semaphore can be interleaved between the RDEX command and the corresponding write, as specified in the OCP protocol. No further action is required on the hardware side—the rest of the sharing protocol is handled through software. A master can, in theory, clear the lock set by another master, or write to a shared resource it does not have access rights to. This would be a violation of the overall system software protocol, and would result in incorrect operation, but would not be a violation of the OCP protocol.

Potential Interoperability Issues

The force_aligned requirement of the Simple Slave Profile can be worked around in the interconnect logic by splitting a non-compliant request, as discussed in the High-Speed Profile definition. This problem does not apply to transactions between devices that support the High Speed and Advanced High Speed Profiles. MRespAccept compatibility between the Simple Slave Profile and the Advanced High Speed Profile is solved the same way as for the High Speed Profile.

When connecting a master that supports the Advanced High-Speed Profile to a slave that supports either the Simple Slave Profile or the High Speed Profile, some glue logic is required to translate the transaction into multiple requests. The request phase signals are extended to issue the correct number of requests, with the exception of addresses which are computed at each phase based on the original transaction base address and on the burst sequence. When interfacing with Simple Slave Profile devices, all the generated requests are stand-alone. In the case of the High-Speed Profile, a multiple request, multiple data (MRMD) transaction is issued, including correct generation of MReqLast, unless the initial transaction is of an unsupported type, i.e., an XOR or BLCK. In the case of an XOR burst, the transaction is split into multiple single requests. For a BLCK burst, the glue logic can either issue multiple single requests or multiple INCR bursts, one for each line in the 2D block.

When connecting a Simple Slave Profile master to a slave that supports the Advanced High-Speed Profile, no glue logic is required for the request group signals. MBurstLength shall be tied to 1 and MBurstSeq to INCR on the slave interface to make sure all transactions are correctly treated as single requests.

A write response to a non-posted write burst shall be issued to the master when all responses in the transaction have been received. If at least one response in the burst had an SResp value of ERR, it is expected that an error response is forwarded to the master. For a response to a posted write command, it is acceptable to immediately send a response to the initiator after acceptance of the request or after receiving the first response from the slave and to drop the next responses, with the drawback that an ERR value for SResp in the rest of the transaction would not be detected. This is true for both Simple Slave Profile and High-Speed Profile masters connected to an Advanced High-Speed Profile slave.

When connected a High-Speed Profile master to an Advanced High-Speed Profile slave, the MBurstSingleReq slave input shall be tied to 0 so that all transactions are treated as MRMD. All High-Speed Profile burst sequences are also supported by the Advanced High-Speed Profile, so no further logic is required for the request and response phases.

The write data phase only exists within the Advanced High-Speed Profile. When translating a burst from an Advanced High-Speed Profile master, the glue logic must make sure it correctly realigns the data from the data handshake phase with the commands passed to the slave. This means that the logic must wait for the current data to be available from the master, i.e., for MDataValid to be asserted, before passing the associated command to the slave so that write data are always part of the request phase.

The RDEX transaction from Advanced High-Speed Profile to Simple Slave Profile or High-Speed Profile cannot be treated as an exclusive operation—it can only be translated into a regular RD transaction. If an Advanced High-Speed Profile master connected to a Simple Slave Profile or High-Speed Profile slave wishes to perform an exclusive access to a resource, it shall do so using other means. For instance, specific dedicated resources can be added to the system for inter-processor synchronization purposes. Another possible solution is to design the system logic, in interconnect and/or bridges, to emulate exclusive access behavior.

Parameter List

Table 71 lists the set of parameters used in the Advanced High-Speed Profile that have non-zero values.

Table 71 List of non-zero parameters in the Advanced High-Speed Profile

Parameter	**Values**	**Description**
burstseq_blck_enable	1	Block burst address sequence (BLCK) enabled
burstseq_incr_enable	1	Incrementing burst address sequence (INCR) enabled
burstseq_strm_enable	1	Streaming burst address sequence (STRM) enabled
burstseq_wrap_enable	1	Wrapping burst address sequence (WRAP) enabled
burstseq_xor_enable	1	Xor burst address sequence (XOR) enabled
endian	little, big, both, neutral	Used endianness needs to be stated
read_enable	1	Read operations (MCmd=RD) are enabled
readex_enable	1	
write_enable	1	Write operations (MCmd=WR) are enabled
writenonpost_enable	1	Non-posted write operations (MCmd=WRNP) are enabled
datahandshake	1	All writes (posted and non-posted) expect a response
addr	1	Address signal (MAddr) is enabled
addr_wdth	64 (max.)	Address signal (MAddr) maximum size is 64 bits
blockheight	1	
blockheight_wdth	6 (max)	
blockstride	1	
blockstride_wdth	32 (max)	
burstlength	1	Burst length signal (MBurstLength) is enabled
burstlength_wdth	6 (max)	Burst length signal (MBurstLength) maximum size is 6 bits
burstseq	1	Burst address sequence signal (MBurstSeq) signal enabled
burstsinglereq	1	
byteen	1	Byte enable signal (MByteEn) is enabled
cmdaccept	1	Command accept signal (SCmdAccept) is enabled
dataaccept	1	
datalast	1	
datarowlast	1	
data_wdth	32, 64, 128	Allowed sizes for data (MData/SData) are 32, 64, and 128 bits

Parameter	Values	Description
enableclk	1	Enable clock signal (EnableClk) is enabled
mdata	1	Write data signal (MData) is enabled
mdatabyteen	1	
resp	1	Response signal (SResp) is enabled
respaccept	1	Response accept signal (MRespAccept) is enabled
resplast	1	Last response indicator signal (SRespLast) enabled
resprowlast	1	
sdata	1	Read data signal (SData) is enabled
mreset	0/1	Controls MReset signal: Slaves should have a reset input (1), reset output for masters is optional (0)
sreset	1/0	Controls SReset signal: Masters should have a reset input (1), reset output for slaves is optional (0)

15.1.4 Optional Features

Some OCP features are so generic that their specific use is difficult to define and so they are listed as optional features. This means that the features are not required by the profiles but can still be used in a system dependent way.

The MAddrSpace signal separates the address regions of a module. This signal is an extension of the regular address signal and can be used in the simple slave, high-speed, and advanced high-speed profiles.

It is sometimes necessary to communicate information about requests and responses between the master and the slave, for instance about routing or security. For this kind of supplemental information OCP uses the MReqInfo signal for the request, and SRespInfo for the response. These signals can be used in all three profiles.

You can transfer information using some of the OCP sideband signals that can travel asynchronously from the request or response flow. The signals are MError to indicate a master internal error, SError to indicate a slave internal error, MFlag for additional information provided by the master, and SFlag for additional information provided by the slave. The use of these signals is optional in all of the profiles.

Tags and threads are identified using ID-signals. MTagID (request tag) and STagID (response tag) are present in the interface if the tags parameter is greater than 1. In addition, the advanced high-speed profile needs MDataTagID if tags are enabled because of datahandshaking. Similar signals exist for threads (MThreadID, SThreadID, and MDataThreadID). They are enabled if threads parameter is greater than 1 and the last one if datahandshake is used (`datahandshake = 1`).

MTagInOrder, STagInOrder, MConnID, MThreadBusy, SDataThreadBusy, and SThreadBusy are also optional signals in the high-speed and advanced high-speed profiles. For systems that use OCP threads and allow interleaving within request phases and data handshake phases, the use of threadbusy signals is recommended.

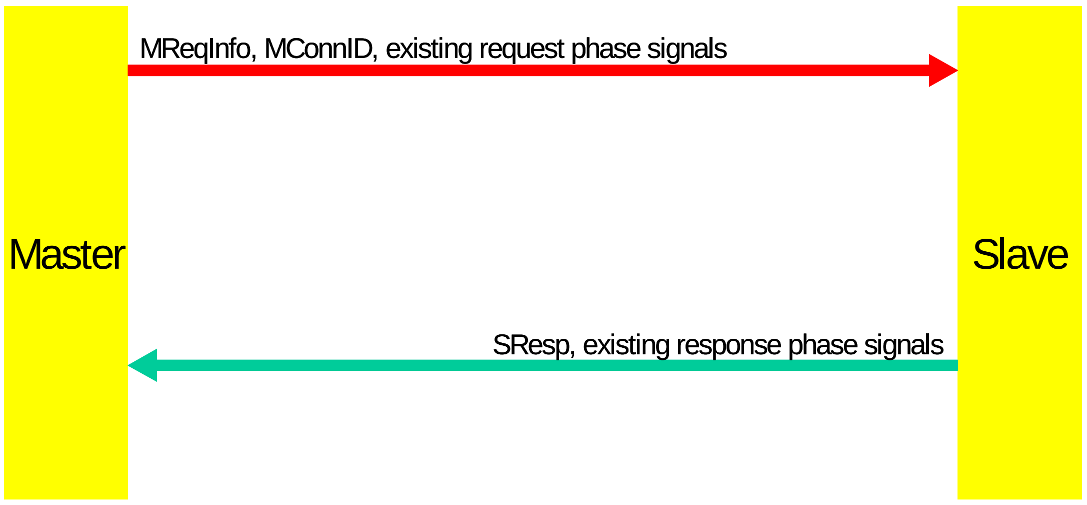
15.1.5 Security

Layered profiles extend the OCP interface as an add-on to any other profile, when additional features are required. The Security profile serves as an example of this concept.

To protect against software and some selective hardware attacks use the OCP interface to create a secure domain across the SOC. The domain might include CPU, memory, I/O etc. that need to be secured using a collection of hardware and software features such as secured interrupts, and memory, or special instructions to access the secure mode of the processor.

The master drives the security level of the request using MReqInfo as a subnet. The master provides initiator identification using MConnID.

Figure 91 Security Signal Processing


MReqInfo, MConnID, existing request phase signals
Master
Slave
SResp, existing response phase signals

Interface Configuration

Table 72 lists the OCP configuration parameters that need to be set along with the recommended values. For default values refer to Table 29, "Configuration Parameter Defaults," on page 68.

Table 72 Security Parameters

Parameter	Value	Notes
reqinfo	1	MReqInfo is required
reqinfo_wdth	Varies	Minimum width is 1
connid	1	To differentiate initiators, if required
connid_wdth	Varies	Minimum width is 1

Implementation Notes

When implementing this profile, consider the following suggestions:

- Define the security request as a named subnet MSecure within MReqInfo, for example: subnet MReqInfo M:N MSecure, where M is >= N.

  With the exception of bit 0, other bits are optional and the encoding is user-defined. Bit 0 of the MSecure field is required and must use the specified value. The suggested encoding for the MSecure bits is:

Bit	Value 0	Value 1
0	non-secure	secure
1	user mode	privileged mode
2	data request	instruction request
3	user mode	supervisor mode
4	non-host	host
5	functional	debug

- A special error response is not specified. A security error can be signaled with response code ERR.

15.1.6 Additional Profiles

In addition to the consensus profiles, a few additional profiles corresponding to typical application needs are defined. Each of the following OCP profiles defines one or more applications. The available profiles are:

- Sequential undefined length data flow
- Register access
- Block data flow (deprecated)

Each profile addresses distinct OCP interfaces, though most systems constructed using OCP will have a mix of functional elements. As different cores and subsystems have diverse communication characteristics and constraints, various profiles will prove useful at different interfaces within the system.

15.1.7 Sequential Undefined Length Data Flow Profile

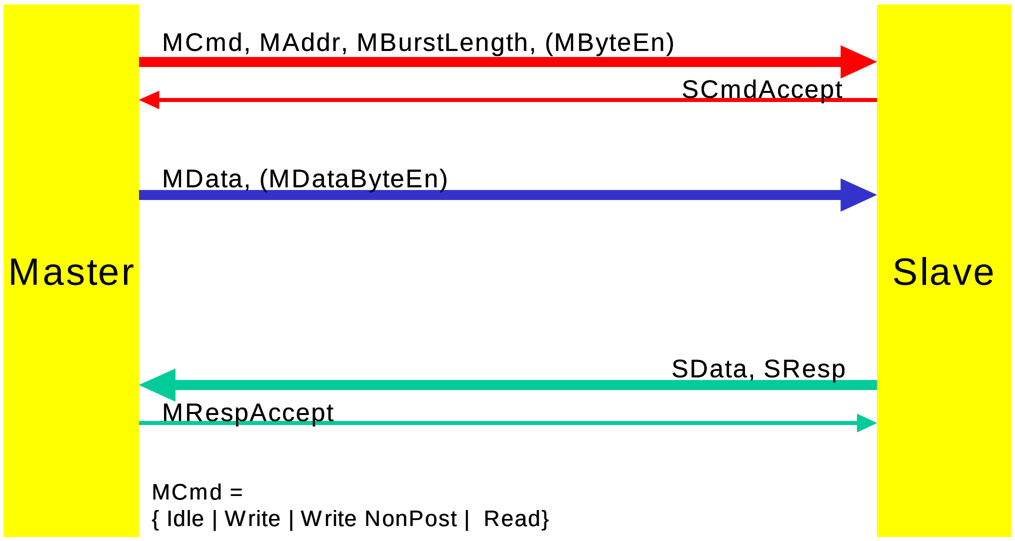
This profile is a master type (read/write or read-only, or write-only) interface for cores that communicate data streams with memory.

Core Characteristics

Cores with the following characteristics would benefit from using this profile:

- Communication of an undefined amount of data elements to consecutive addresses
- Imprecise burst model
- Aligned command/write data flows, decoupled read data flow
- 32 bit address
- Natural data width
- Optional use of byte enables
- Single thread
- Support for producer/consumer synchronization

Figure 92 Sequential Undefined Length Data Flow Signals Processing


MCmd, MAddr, MBurstLength, (MByteEn)
SCmdAccept
MData, (MDataByteEn)
Master
Slave
SData, SResp
MRespAccept
MCmd =
{ Idle | Write | Write NonPost | Read}

Interface Configuration

Table 73 lists the OCP configuration parameters that need to be set along with the recommended values. For default values refer to Table 29, “Configuration Parameter Defaults,” on page 68.

Table 73 Sequential Undefined Length Data Flow Parameter Settings

Parameter	**Value**	**Notes**
addr_width	32	
burstlength	1	
burstlength_width	2	
burstseq	1	
byteen	1 (optional)	For interfaces that are capable of partial access
data_width	core specific	Choose a data width that is natural to the operation of the core
read_enable	1 (optional)	For read capable interface only
respaccept	1	
write_enable	1 (optional)	For write capable interface only
writenonpost_enable	1	
writeresp_enable	1	

Implementation Notes

When implementing this profile, consider the following suggestions:

- The core streams data to and from a memory-based buffer at sequential addresses. When hitting the boundaries of that buffer a non-sequential address is occasionally given.

  MBurstLength equals 2 while the data stream proceeds to sequential addresses; MBurstLength equals 1 to indicate that a non-sequential address follows, or that the stream terminates.

- Start read transactions as early as possible to hide read latency behind ongoing transactions.
- To implement a producer/consumer synchronization scheme (typically the case when data written by the IP core to shared memory is read by another core in the system), the IP core should issue a synchronization request (for instance through an OCP flag interface) only after receiving a response that the last write transaction has committed. To accomplish this step, perform all write transactions up to the last one as posted writes. Make the final write transaction a non-posted write. This will lead to reception of a response once the non-posted write transaction has committed, i.e., completed at the final destination.
- Error response should lead to an interrupt.

15.1.8 Register Access Profile

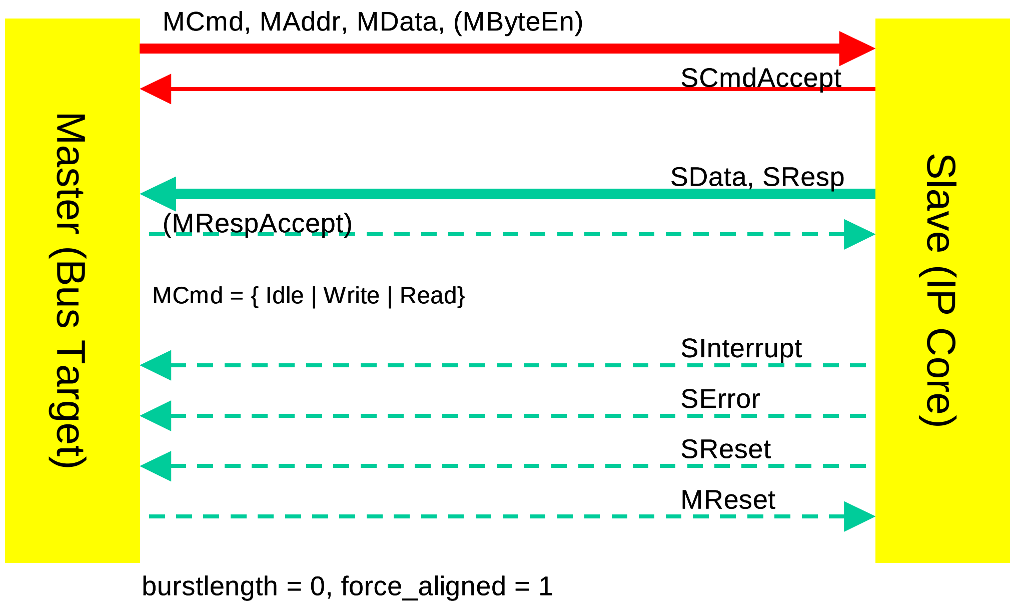
The register access profile offers a control processor the ability to program the operation of an attached core. This profile supports programmable register interfaces across a wide range of IP cores, such as simple peripherals, DMA engines, or register-controlled processing engines. The IP core would be an OCP slave on this interface, connected to a master that is a target addressed by a control processor.

Core Characteristics

Cores with the following characteristics would benefit from using this profile:

- Address mapped communication, but target decoding is handled upstream
- Natural data width (unconstrained, but 32 bits is most common)
- Natural address width (based on the number of internal registers X data width)
- No bursting
- Precise write responses indicate completion of write side-effects
- Single threaded
- Optional aligned byte enables for sub-word CPU access
- Optional response flow control
- Optional use of side-band signals for interrupt, error, and DMA ready signaling

Figure 93 Register Access Signals Processing



Interface Configuration

Table 74 lists the OCP configuration parameters that need to be set along with the recommended values. For default values refer to Table 29, “Configuration Parameter Defaults,” on page 68.

Table 74 Register Access Parameter Settings

Parameter	**Value**	**Notes**
addr	1	
addr_wdth	Varies	Num_regs * data_wdth/8
byteen	Varies	Not suggested for new designs
cmdaccept	1	
data_wdth	Varies	8, 16, 32 and 64 bits; 32 is preferred
force_aligned	1	Normally read/written by aligned CPU
interrupt	Varies	For cores with multiple interrupt lines use SFlag
mdatainfo	0	
mreset	Varies	
respaccept	1	Include when possible!
serror	Varies	If core has internally-generated errors
sreset	Varies	If core receives own (non-interface) reset
writenonpost_enable	Varies	Not usually needed
writeresp_enable	1	Precise write responses needed for posted writes.

Implementation Notes

When implementing this profile, consider the following suggestions:

- Choose a data width based on the internal register width of the core.
- Choose an address width based on the data width and number of registers.
- Design new cores so that all read/write side-effects can be managed without using byte enables.
- Design new cores so that registers are at least 32 bit aligned.
- Use force aligned byte enables for cores with side effects in multiple-use registers.
- Implement response flow control if convenient.
- Implement sideband signaling towards CPU (interrupts, sideband errors, etc.) on this interface, since it is largely controlled by the CPU.

- Select reset options based on the expected use of the core in a larger system.
- Use regular Write commands for precise completion, unless the core is capable of per-transfer posting decisions, where mixing Write and WriteNonPost helps.

15.1.8.1 Block Data Flow Profile (Deprecated)

This profile has been supplanted by the high-speed profile.

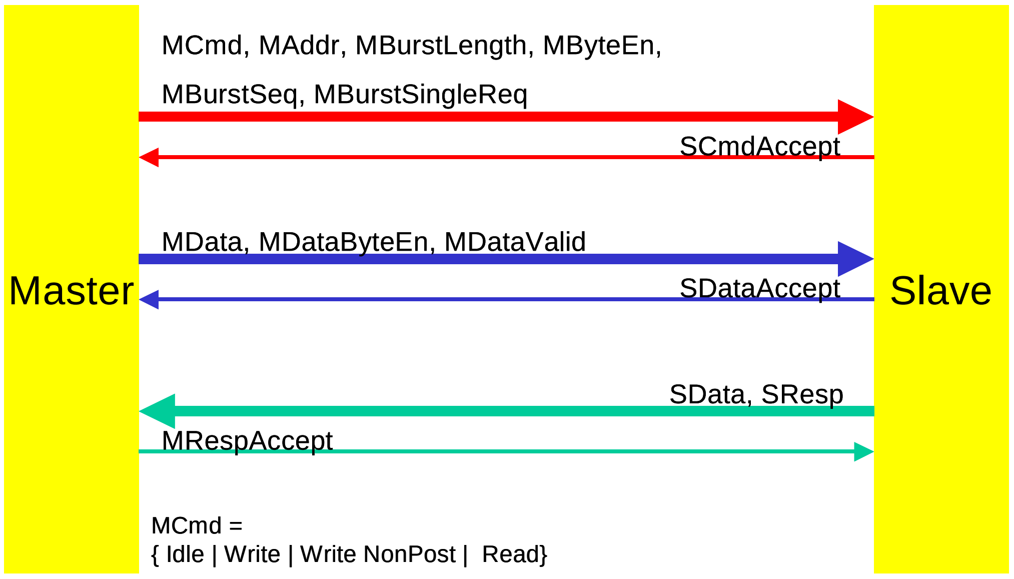
The block data flow profile is designed for master type (read/write, read-only, or write-only) interfaces of cores exchanging data blocks with memory. This profile is particularly effective for managing pipelined access of defined-length traffic (for example, MPEG macroblocks) to and from memory.

Core Characteristics

Cores with the following characteristics would benefit from using this profile:

- Block-based communication (includes a single data element block)
- A single request/multiple data burst model using incremental or a stream burst sequence
- De-coupled, pipelined command and data flows
- 32 bit address
- Natural data width and block size
- Use of byte enables
- Single threaded
- Support for producer/consumer synchronization through non-posted writes

Figure 94 Block Data Flow Signal Processing


Master
Slave
MCmd, MAddr, MBurstLength, MByteEn,
MBurstSeq, MBurstSingleReq
SCmdAccept
MData, MDataByteEn, MDataValid
SDataAccept
SData, SResp
MRespAccept
MCmd =
{ Idle | Write | Write NonPost | Read}

Interface Configuration

Table 75 lists the OCP configuration parameters that need to be set along with the recommended values. For default values refer to Table 29, “Configuration Parameter Defaults,” on page 68.

Table 75 Block Data Flow Parameter Settings

Parameter	**Value**	**Notes**
addr_width	32	
burstlength	1	
burstlength_width	Core specific	Choose a burst length that is natural to the operation of the core
burstseq	1	If the core is STRM burst capable
burstseq_strm_enable	1	If the core is STRM burst capable
burstsinglereq	1	
byteen	1	For interfaces that are capable of partial access
data_width	Core specific	Choose a data width that is natural to the operation of the core
dataaccept	1	
datahandshake	1	
mdatabyteenable	1	
read_enable	1	For read capable interface only
respaccept	1	

Parameter	Value	Notes
write_enable	1	For write capable interface only
writenonpost_enable	1	See note on synchronization below
writeresp_enable	1	Needed for posted writes

Implementation Notes

When implementing this profile, consider the following suggestions:

- Start read transactions as early as possible to minimize read latency behind ongoing transactions.
- To implement a synchronization scheme (typically the case when data written by the IP core to shared memory is read by another core in the system), the IP core should issue a synchronization request (for instance through an OCP flag interface) only after receiving a response that the last write transaction has committed. To accomplish this step, perform all write transactions up to the last one as posted writes. Make the final write transaction a non-posted write. This will lead to reception of a response once the non-posted write transaction has committed, i.e., completed at the final destination.
- Error responses should lead to an interrupt.

15.2 Bridging Profiles

The bridging profiles are designed to simplify or automate the creation of bridges to other interface protocols. The bridge can have an OCP master or slave port. There are two types:

- The simple H-bus profile is intended to provide a connection through an external bridge, for example to a CPU with an AMBA AHB protocol.
- The X-bus interfaces support cacheable and non-cacheable instruction and data traffic between a CPU and the memories and register interfaces of other targets. The X-bus profiles might be used with a CPU core that internally uses the AMBA AXI protocols, and is externally bridged to OCP.

15.2.1 Simple H-bus Profile

This profile allows you to create OCP master wrappers to native interfaces of simple CPU type initiators with multiple-request/multiple-data, read and write transactions.

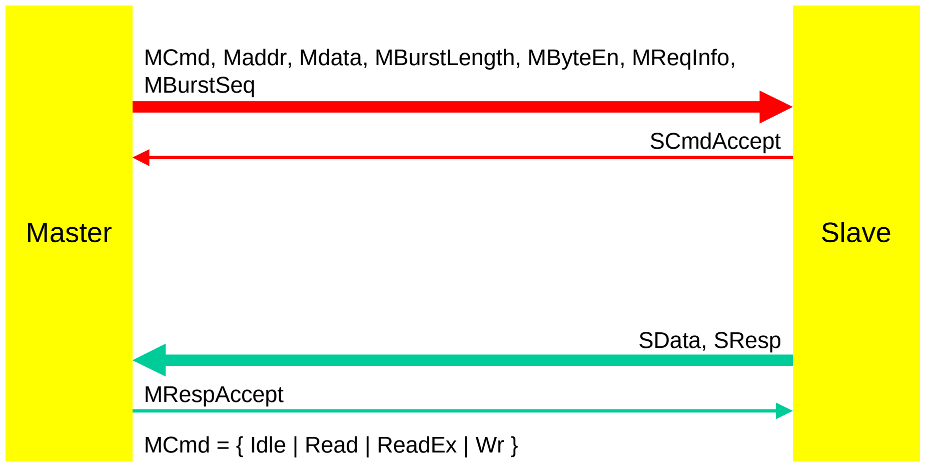
Core Characteristics

Cores with the following characteristics would benefit from using this profile:

- Address mapped communication
- Natural address width

- Byte enable
- Natural data width
- Constrained burst size
- Single thread
- Caching and similar extensions mapped to MReqInfo or corresponding OCP commands

Figure 95 Simple H-Bus Signal Processing


Master
Slave
MCmd, Maddr, Mdata, MBurstLength, MByteEn, MReqInfo, MBurstSeq
SCmdAccept
SData, SResp
MRespAccept
MCmd = { Idle | Read | ReadEx | Wr }

Interface Configuration

Table 76 lists the OCP configuration parameters that need to be set along with the recommended values. For default values refer to Table 29, “Configuration Parameter Defaults,” on page 68.

Table 76 Simple H-Bus Parameter Settings

Parameter	**Value**	**Notes**
addr_wdth	Varies	Use native address width
burstlength	1	
burstlength_wdth	5	Only short burst supported
burstprecise	0	MBurstPrecise signal is not part of the interface. All bursts are precise
burstseq	1	Subset of burst codes common with OCP and CPU
burstseq_wrap_enable	1	
byteen	1	
data_wdth	Varies	8, 16, 32 and 64 bits
force_aligned	1	
mreset	1	

Parameter	Value	Notes
readex_enable	1	For CPUs with locked access
reqinfo	1	
reqinfo_wdth	Varies	Map CPU-specific in-band info here
reqlast	1	Can be created of burst length
respaccept	1	
sreset	1	
writeresp_enable	1	

Implementation Notes

When implementing this profile, consider the following suggestions:

- If the CPU's burst parameters such as data alignment are not supported in OCP, such bursts are broken into single transactions.
- All requests have a response.
- If the CPU address and write data are pipelined, the OCP bridge will align them.
- CPU specific information is mapped to the MReqInfo field, however, since this field is often used for proprietary bits by OCP users, a bit-to-bit mapping is not provided. For this field concatenate bit fields starting from bit 0. If the in-band field contains control information that has equivalent native OCP functionality, map the information to the corresponding OCP request. For example, a bit that can be buffered can be mapped to OCP posted or non-posted write types.

15.2.2 X-Bus Packet Write Profile

This profile is designed to create OCP master wrappers to native interfaces of CPU type initiators with single-request/multiple-data, write-only transactions.

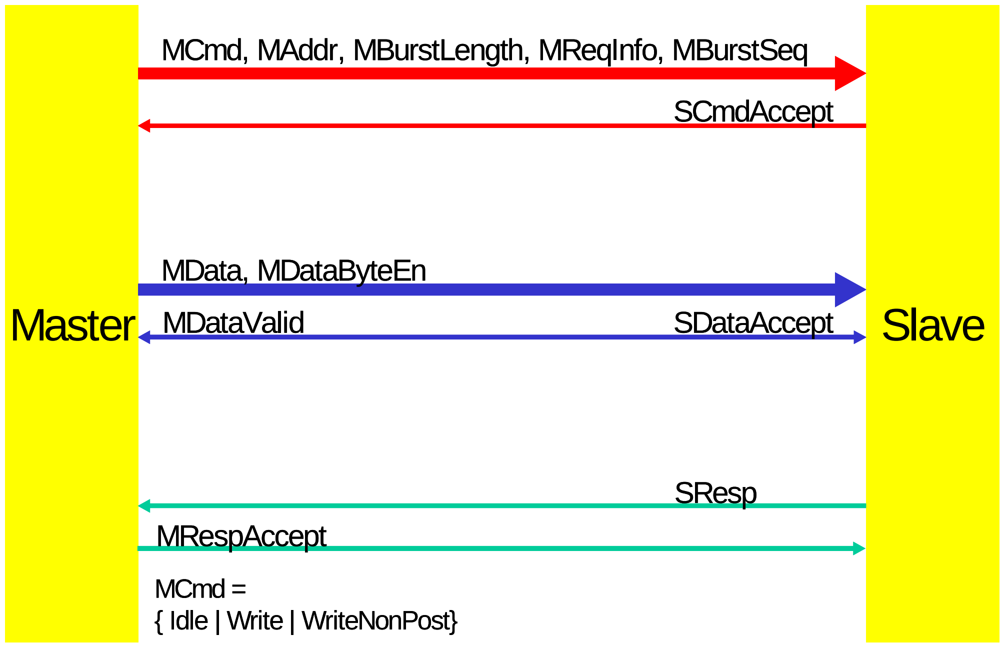
Core Characteristics

Cores with the following characteristics would benefit from using this profile:

- Packet type communication
- Natural address width
- Separate command/write data handshake
- Byte enable
- Natural data width
- Multi-thread (blocking)

- Caching and similar extensions mapped to MReqInfo

Figure 96 X-bus Packet Write Signal Processing



Interface Configuration

Table 77 lists the OCP configuration parameters that need to be set along with the recommended values. For default values refer to Table 29, “Configuration Parameter Defaults,” on page 68.

Table 77 X-bus Packet Write Parameter Settings

Parameter	**Value**	**Notes**
addr_wdth	Varies	Use native address width
burstlength	1	
burstlength_wdth	5	Only short burst support
burstprecise	0	MBurstPrecise signal is not part of the interface. All bursts are precise
burstseq	1	Subset of burst codes common with OCP and CPU
burstseq_strm_enable	1	
burstseq_wrap_enable	1	
byteen	0	Only databyteen needed for write-only
data_wdth	Varies	8, 16, 32 and 64 bits
dataaccept	1	
datahandshake	1	

Parameter	Value	Notes
datalast	1	Can be created of burst size
interrupt	0	Interrupts are not part of this interface
mdatabyteen	1	
mreset	1	
read_enable	0	Write only
reqdata_together	Varies	A simpler bridge can often be made if 1, some performance loss possible
reqinfo	1	
reqinfo_wdth	Varies	Map CPU-specific in-band info here
reqlast	1	Superfluous for single request, datalast suffices
respaccept	1	
resplast	1	
sdata	0	
sreset	1	
sthreadbusy	0	Blocking threads only
threads	Varies	Natural number of threads
writenonpost_enable	1	
writeresp_enable	1	Needed for posted writes only

Implementation Notes

When implementing this profile, consider the following suggestions:

- Data ordering among read and write port transactions is the responsibility of the CPU. The bridge must consider the read and write ports as single master with regard to exclusive access.
- Only precise bursts are supported. If CPU burst parameters do not map to OCP, break the burst into single accesses.
- CPU specific information is mapped as in the H-bus profile.

15.2.3 X-Bus Packet Read Profile

This profile helps you create OCP master wrappers for native interfaces of CPU type initiators with single-request multiple-data read-only transactions.

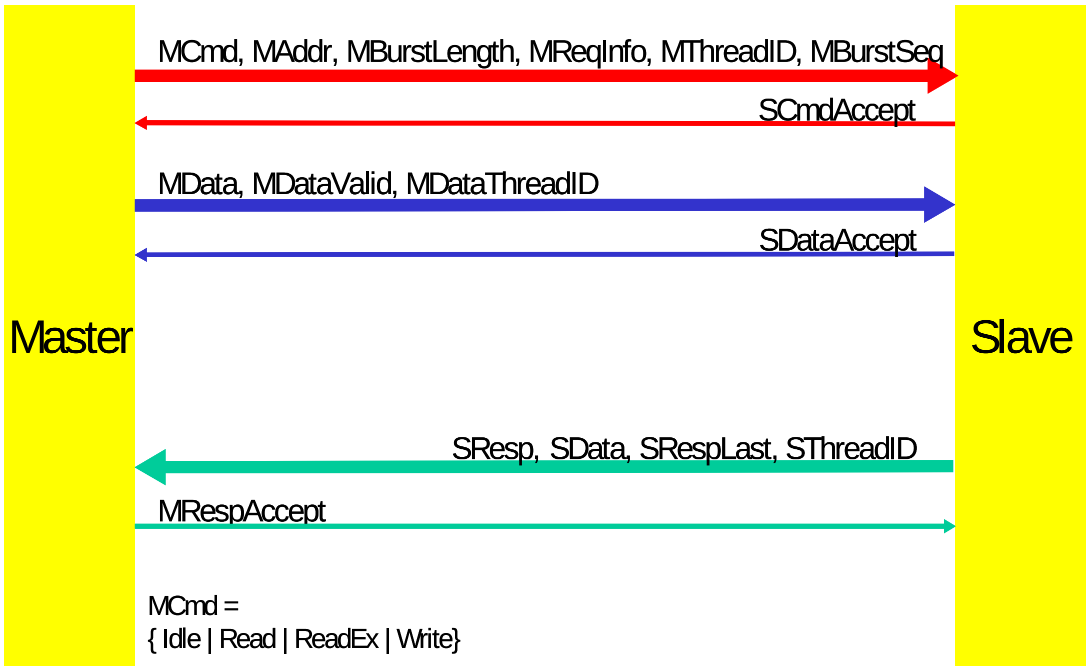
Core Characteristics

Cores with the following characteristics would benefit from using this profile:

- Packet type communication

- Natural address width
- Single-request multiple-data read
- Byte enable
- Natural data width
- Multi-thread (blocking)
- Caching and similar extensions mapped to MReqInfo
- Write support for ReadEx/Write synchronization

Figure 97 X-bus Packet Read Signal Processing


MCmd, MAddr, MBurstLength, MReqInfo, MThreadID, MBurstSeq
SCmdAccept
MData, MDataValid, MDataThreadID
SDataAccept
Master
Slave
SResp, SData, SRespLast, SThreadID
MRespAccept
MCmd =
{ Idle | Read | ReadEx | Write}

Interface Configuration

Table 78 lists the OCP configuration parameters that need to be set along with the recommended values. For default values refer to Table 29, “Configuration Parameter Defaults,” on page 68.

Table 78 X-bus Packet Read Parameter Settings

Parameter	**Value**	**Notes**
addr_wdth	Varies	Use native address width
burstlength	1	
burstlength_wdth	5	Only short burst support
burstprecise	0	MBurstPrecise signal is not part of the interface. All bursts are precise

Parameter	Value	Notes
burstseq	1	Subset of burst codes common with OCP and CPU
burstseq_strm_enable	1	
burstseq_wrap_enable	1	
burstsinglereq	1	
byteen	1	
data_wdth	Varies	8, 16, 32 and 64 bits
force_aligned	0	If CPU alignment is not supported in OCP, such bursts are broken into single transactions.
interrupt	0	Interrupts are not part of this interface
mreset	1	
readex_enable	1	
reqinfo	1	
reqinfo_wdth	Varies	Map CPU-specific in-band info here
respaccept	1	
resplast	1	
sreset	1	
sthreadbusy	0	Blocking threads only
threads	Varies	Natural number of threads
write_enable	1	To support ReadEx

Implementation Notes

When implementing this profile, consider the following suggestions:

- Data integrity among read and write port transactions is the responsibility of the CPU. The bridge must consider the read and write ports as single master with regard to exclusive access. Both transfers in the ReadEx/ Write pair should be issued on the X-bus packet read interface.
- Only precise bursts are supported. If the CPU burst parameters do not map to OCP, break the burst into single accesses.
- CPU specific information is mapped as in the H-bus profile.

16 *Core Performance*

To make it easier for the system integrator to choose cores and architect the system, an IP core provider should document a core's performance characteristics. This chapter supplies a template for a core performance report on page 354, and directions on how to fill out the template.

16.1 Report Instructions

To document the core, you will need to provide the following information:

1. Core name. Identify the core by the name you assigned.
2. Core ID. Specify the identification of the core inside the system-on-chip. The information consists of the vendor code, core code, and revision code.
3. Core is/is not process dependent. Specify whether the core is process-dependent or not. This is important for the frequency, area, and power estimates that follow.

   If multiple processes are supported, name them here and specify corresponding frequency/area/power numbers separately for each core if they are known.
4. Frequency range for this core. Specify the frequency range that the core can run at. If there are conditions attached, state them clearly.
5. Area. Specify the area that the core occupies. State how the number was derived and be precise about the units used.
6. Power estimate. Specify an estimate of the power that the core consumes. This naturally depends on many factors, including the operations being processed by the core. State all those conditions clearly, and if possible, supply a file of vectors that was used to stimulate the core when the power estimate was made.

7. Special reset requirements. If the core needs MReset_n/SReset_n asserted for more than the default (16 OCP clock cycles) list the requirement.

8. Number of interfaces.

9. Interface information. For each OCP interface that the core provides, list the name and type.

   The remaining sections focus on the characteristics and performance of these OCP interfaces.

For master OCP interfaces:

a. Issue rate (per OCP cycle for sequences of reads, writes, and interleaved reads/writes). State the maximum issue rate. Specify issue rates for sequences of reads, writes, and interleaved reads and writes.

b. Maximum number of operations outstanding (pipelining support). State the number of outstanding operations that the core can support; is there support for pipelining.

c. If the core has burst support, state how it makes use of bursts, and how the use of bursts affects the issue rates.

d. High level flow-control. If the core makes use of high-level flow control, such as full/empty bits, state what these mechanisms are and how they affect performance.

e. If multiple threads are present, explain the use of threads.

f. Connection ID support. Explain the use and meaning of connection information.

g. Use of side-band signals. For each sideband signal (such as SInterrupt, MFlag) explain the use of the signal.

h. If the OCP interface has any implementation restrictions, they need to be clearly documented.

For slave OCP interfaces:

a. Unloaded latency for each operation (in OCP cycles). Describe the unloaded latency of each type of operation.

b. Throughput of operations (per OCP cycle for sequences of reads, writes, and interleaved reads/writes). State the maximum throughput of the operations for sequences of reads, writes, and interleaved reads and writes.

c. Maximum number of operations outstanding (pipelining support). State the number of outstanding operations that the core can support, i.e. is there support for pipelining.

d. Burst support and effect on latency and throughput numbers. If the core has burst support, state how it makes use of bursts, and how the use of bursts affects the latency and throughput numbers stated above.

e. High level flow-control. If the core makes use of high-level flow control, such as full/empty bits, state what these mechanisms are and how they affect performance.

f. If multiple threads are present, explain the use of threads.

g. Connection ID support. Explain the use and meaning of connection information.

h. Use of side-band signals. For each sideband signal (such as SInterrupt, MFlag) explain the use of the signal.

i. If the OCP interface has any implementation restrictions, they need to be clearly documented.

For every non-OCP interface, you will need to provide all of the same information as for OCP interfaces wherever it is applicable.

16.2 Sample Report

1. Core name	flashctrl
2. Core identity Vendor code Core code Revision code	 0x50c5 0x002 0x1
3. Core is/is not process dependent	Not
4. Frequency range for this core	≤100Mhz with NECCBC9-VX library
5. Area	4400 gates 2input NAND equivalent gates
6. Power estimate	not available
7. Special reset requirements	
8. Number of interfaces	2
9. Interface information: Name Type	 ip slave
For master OCP interfaces:	
a. Issue rate (per OCP cycle for sequences of reads, writes, and interleaved reads/writes)	
b. Maximum number of operations outstanding (pipelining support)	
c. Effect of burst support on issue rates	
d. High level flow-control	
e. Use of threads (if any)	
f. Use of connection information	
g. Use of side-band signals	
h. Implementation restrictions	

For slave OCP interfaces:	
a. Unloaded latency for each operation (in OCP cycles)	Register read or write: 1 cycle. The flash read takes SBFL_TAA (read access time). Can be changed by writing corresponding register field of emem configuration register. The flash write operation takes about 2000 cycles since it has to go through the sequence of operations - writing command register, reading the status register twice.
i. Throughput of operations (per OCP cycle for sequences of reads, writes, and interleaved reads/ writes)	No overlap of operations therefore reciprocal of latency.
j. Maximum number of operations outstanding (pipelining support)	No pipelining support.
k. Effect of burst support on latency and throughput numbers	No burst support.
l. High level flow-control	No high-level flow-control support.
m. Use of threads (if any)	No thread support.
n. Use of connection information	No connection information support.
o. Use of side-band signals	Reset_n, Control, SError. Control is used to provide additional write protection to critical blocks of flash memory. SError is used when an illegal width of write is performed. Only 16 bit writes are allowed to flash memory.
p. Implementation restrictions	
For every non-OCP interface Provide all of the same information as for OCP interfaces wherever it is applicable.	Hitachi flash card HN29WT800 Only 1 flash ROM part is supported, therefore the CE_N is hardwired on the board. The ready signal RDY_N, is not used since not all parts support it. For the BYTE_N signal, only 16-bit word transfers are supported

16.3 Performance Report Template

Use the following template to document a core.

1. Core name
2. Core identity Vendor code Core code Revision code
3. Core is/is not process dependent
4. Frequency range for this core
5. Area
6. Power estimate
7. Special reset requirements
8. Number of interfaces
9. Interface information: Name Type
For master OCP interfaces:
a. Issue rate (per OCP cycle for sequences of reads, writes, and interleaved reads/writes)
b. Maximum number of operations outstanding (pipelining support)
c. Effect of burst support on latency and throughput numbers
d. High level flow-control
e. Use of threads (if any)
f. Use of connection information
g. Use of side-band signals
h. Implementation restrictions

For slave OCP interfaces:	
a. Unloaded latency for each operation (in OCP cycles)	
i. Throughput of operations (per OCP cycle for sequences of reads, writes, and interleaved reads/ writes)	
j. Maximum number of operations outstanding (pipelining support)	
k. Effect of burst support on latency and throughput numbers	
l. High level flow-control	
m. Use of threads (if any)	
n. Use of connection information	
o. Use of side-band signals	
p. Implementation restrictions	
For every non-OCP interface Provide all of the same information as for OCP interfaces wherever it is applicable.	

Part III *Protocol Compliance*

17 *Compliance*

This section contains the OCP compliance checks that can help you create checking solutions in the language and tool of your choice.

The guidelines listed in this section are based on the "Specification" and "Guidelines" parts of this document and allow you to verify an IP/Verification IP (VIP) for OCP compliance. In all cases, "Part I, Specification" is the definitive reference. Any references made to "Part II, Guidelines" are not definitive as Part I supersedes the guidelines.

For a core to be considered OCP compliant it must satisfy the compliance definition as described in Section 1.2 on page 3.

17.1 Configuration Compliance

17.1.1 Interface Configuration

The main challenge in developing an OCP VIP lies in accounting for the high degree of configurability of OCP. Figure 98 shows the different inputs that can affect OCP configurability. To properly define the OCP interfaces of an IP/VIP, consider the following contexts.

Open System Context

For an open system, it must be possible to setup all of the OCP interfaces with a file using the <core>_rtl.conf syntax, which is required for OCP compliance. Fixed configuration IP/VIP must be delivered with a core_rtl.conf file describing the configuration. The metadata properties for this are described in Chapter 8.

Configurable IP/VIP supporting multiple OCP configurations must support the setup of any configuration using a <core>_rtl.conf file. The mechanisms used to fix the configuration must provide a method for generating a core_rtl.conf file that represents the fixed configuration and can be used to configure the IP/VIP directly.

For IP providers <core>_rtl.conf generation likely occurs during the IP generation step. When the IP code is generated based on configuration, and other settings in the GUI, the <core>_rtl.conf file is generated along with the IP.

Closed System Context

In a closed system, the verification of an IP/VIP with one or more OCP interfaces may be driven from a <core>_rtl.conf file. A vendor is free to implement any other solution. For example, a VERA verification environment could use a VERA object to control the OCP stimuli generators instead of a <core>_rtl.conf file.

If the IP/VIP is being developed in a closed system for delivery in an open system context, then the verification must include the <core>_rtl.conf files and any applicable <core>_rtl.conf generators that are delivered with the IP/VIP.

17.1.2 Configuration Parameter Extraction

Depending on the system context, the VIP must extract the OCP configuration parameters from the <core>_rtl.conf file (open) or from any alternate solution (closed). Parameters with indeterminate values must be retrieved using the configuration parameter defaults summarized in Table 29, “Configuration Parameter Defaults,” on page 68 (Table 22 of the *OCP 2.0 Specification)*. Some parameters are required in certain configurations, and for those, no default is specified. For example: `addr_wdth` must always be specified if addr == 1.

17.2 Protocol Compliance

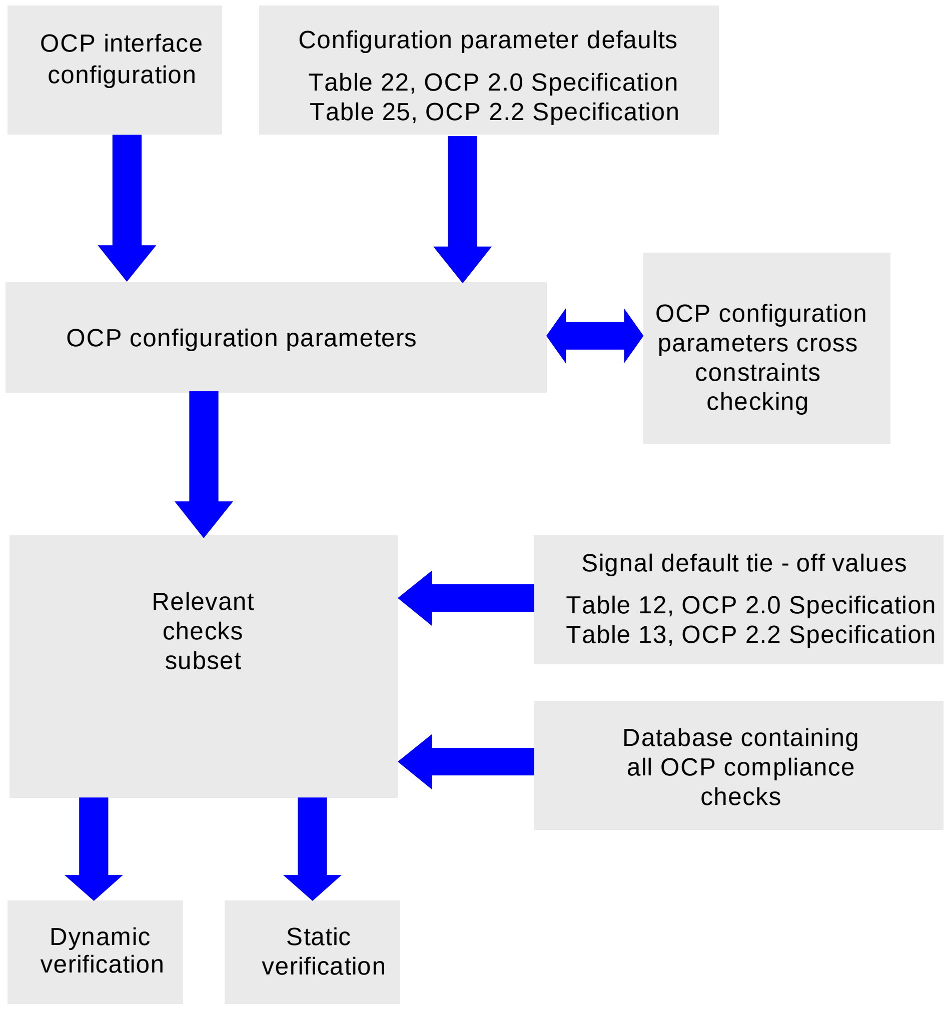
Once all the OCP configuration parameters are known, illegal OCP configurations must be flagged. Chapter 19 contains compliance checks for the configuration parameters. Chapter 4 contains most of the cross-constraints. For example: if `readex_enable` is set to 1, `write_enable` or `writenonpost_enable` must be set to 1.

17.2.1 Select the Relevant Checks

Based on the OCP configuration parameters, select a subset of the checks in the VIP OCP library. This subset is used for the actual verification. If a signal used by a check is not configured in the OCP interface and if no other tie-off value is specified, Table 16, “OCP Signal Configuration Parameters,” on page 31 (Table 12 of the *OCP 2.0 Specification*) specifies the inferred default tie-off values. For example, the MBurstPrecise default tie-off value is 1 or precise.

Check the compliance of the DUT OCP interfaces using static or dynamic verification techniques described in the next section

Figure 98 OCP Configurability


OCP interface configuration
Configuration parameter defaults
Table 22, OCP 2.0 Specification
Table 25, OCP 2.2 Specification
OCP configuration parameters
OCP configuration parameters cross constraints checking
Relevant checks subset
Signal default tie - off values
Table 12, OCP 2.0 Specification
Table 13, OCP 2.2 Specification
Database containing all OCP compliance checks
Dynamic verification
Static verification

17.3 Verification Techniques

The verification guidelines are valid for developers using static or dynamic verification methods. This section provides an overview of static and dynamic verification methods, along with guidance on how these checks can be used to support these verification efforts.

17.3.1 Dynamic Verification

Dynamic verification methodology consists of:

- Driving a set of stimuli through the OCP interface into the DUT.

- Using a protocol checker on the VIP monitor traces of OCP interface activity to make sure that the protocol is not violated.
- Assessing the quality of the stimuli using functional and code coverage.

The OCP configuration parameters determine which protocol checks must be active and how the OCP functional coverage is defined.

Stimuli

Because of the degree of difficulty of defining a golden set of stimuli for any OCP interface configuration, you will need to implement a smart and efficient set of stimuli. This may be accomplished using constraint-driven random stimuli generation. The quality of the stimuli must be assessed using both functional and code coverage as described below.

Protocol Checks

The protocol checker is a passive component that monitors a specific set of OCP configuration parameters to determine whether the OCP protocol is violated. The protocol checker can be written in a variety of languages including HDL, PSL, SVA, E, NSCa, or VERA. The protocol checker must be instantiated on each OCP interface of the DUT.

To allow this document to be easily referenced, the names of the protocol checks must match the names given to the compliance checks described in this document.

Functional Coverage

Measure the quality of the applied stimuli. The target is 100% functional coverage. Run code coverage on the RTL to determine whether there are any verification holes such as uncovered FSM states or missed branches. Based on the code coverage analysis, additional coverage metrics may be required.

The guidelines for OCP functional coverage are provided in Chapter 20. Any additional coverage metrics, based on code-coverage analysis, are design dependent and are out of the scope of this document.

17.3.2 Static Verification

The static verification approach is also referred to as formal verification and relies on the following key elements:

- OCP protocol assertions (or protocol checkers)
- OCP protocol constraints (optional)
- OCP functional coverage

This approach uses a formal tool to prove that, given the stimulus limits defined by the OCP protocol constraints, the OCP interface of the DUT never violates OCP protocol assertions. The formal proof may be exhaustive (the assertions are never violated) or bounded (up to a certain depth of the state space the assertions are never violated). Stimuli are not needed; instead the tool relies on the 'all acceptable stimuli' definitions provided by the OCP protocol constraints.

The OCP configuration parameters determine:

- Which protocol assertions must be active.
- How the functional coverage must be defined.
- Which protocol constraints must be active.

Protocol Assertions

Formal verification revolves around taking the protocol assertions and attempting to prove that they are never violated. If a violation is found, the formal tool provides a test sequence that illustrates the violation on the design. The assertions can be written in different languages such as HDL, PSL or SVA.

To allow this document to be easily referenced, the names of the assertions must match the names given to the compliance checks described in this document.

Protocol Constraints

To place bounds on the stimuli that a formal engine must consider, the design must be connected to protocol constraints or some form of generator description. Constraints can be specified using the same language as is used for protocol assertions, typically in HDL, PSL, SVA, or OVA.

Protocol constraints are not provided and must be obtained or created for use with formal tools. Constraints must be specific enough to prevent invalid test sequences that can lead to false negative test results (as indicated by protocol assertion failures) but not so specific that they prevent valid test sequences. The latter situation can lead to false positive test results implied by protocol assertion success over an incomplete set of test sequences.

Using functional coverage, false positive results can be checked, however, the false negatives cannot be checked as easily.

Functional Coverage

You must insure that all of the protocol assertions are verified to a reasonable extent, and that the protocol constraints are sound. To accomplish this, some functional coverage must be added as a function of the OCP configuration parameters. The functional coverage definition should cover:

- Which assertions warrant exhaustive proofs
- Which assertions are ok with just bounded proofs, at what depth
- Detect and correct an over-constrained environment

The guidelines for the OCP functional coverage are described in Chapter 20.

18 Protocol Compliance Checks

The compliance checks listed in this chapter are extracted from the *OCP Specification* and are intended to serve as guidelines to verify an IP for OCP compliance. In all cases "Part I, Specification" is the definitive reference.

The compliance check names have been created using the following template:

<hierarchy>_<check type>_<critical signal>_<extra details>

In which:

<hierarchy> signal, request, datahs, response, burst, transfer, rdex

<check type> valid, hold, value, exact, phase_order, lock_release, sequence, order, reorder

<critical signal> (optional) any OCP signal name that is impacted by the compliance check

<extra details> a short additional explanation

18.1 Activation Tables

Tables 79–85 list the parameters needed for each check to be initiated. The following assumptions are made with respect to these tables:

- An understanding of how a combination of these parameters can lead to an illegal configuration.
- The tables only show the minimum parameters needed for a check to be fired. Each configuration needs the parameters defined for each check plus the parameters needed to make it a legal configuration. For example, a check for INCR bursts would need some command (read_enable, write_enable, etc.) parameters defined to test the check.

Table 79 Dataflow Signal Checks

Name	Activation Parameters
1.1.1 signal_valid_<signal>_when_reset_inactive	
MCmd	-
MDataValid	datahandshake
MThreadBusy	mthreadbusy
SDataThreadBusy	sdatathreadbusy
SResp	resp
SThreadBusy	sthreadbusy
1.1.2 request_valid_<signal>	
MAddr	addr
MAddrSpace	addrspace
MAtomicLength	atomiclength
MBlockHeight	blockheight
MBlockStride	blockstride
MBurstLength	burstlength
MBurstPrecise	burstprecise
MBurstSeq	burstseq
MBurstSingleReq	burstsinglereq
MByteEn	byteen
MConnID	connid
MReqLast	reqlast
MThreadID	threads > 1
SCmdAccept	cmdaccept
1.1.3 datahs_valid_<signal>	
MDataByteEn	mdatabyteen
MDataLast	datalast
MDataThreadID	datahandshake & threads > 1
SDataAccept	dataaccept
1.1.4 response_valid_<signal>	
MRespAccept	respaccept
SRespLast	resplast
SThreadID	resp & threads > 1
1.1.5 request_valid_MTagInOrder	Taginorder
1.1.6 response_valid_STagInOrder	resp & taginorder
1.1.7 request_valid_MTagID_when_MTagInOrder_zero	tags > 1
1.1.8 datahs_valid_MDataTagID_when_MTagInOrder_zero	datahandshake & tags > 1
1.1.9 response_valid_STagID_when_STagInOrder_zero	resp & tags > 1

Table 80 Dataflow Phase Checks

Name	Activation Parameters
1.2.1 request_exact_SThreadBusy	sthreadbusy & sthreadbusy_exact & ~sthreadbusy_pipelined
1.2.2 request_pipelined_SThreadBusy	sthreadbusy & sthreadbusy_exact & sthreadbusy_pipelined
1.2.3 request_hold_<signal>	
MAddr	cmdaccept & addr
MAddrSpace	cmdaccept & addrspace
MAtomicLength	cmdaccept & atomiclength
MBlockHeight	cmdaccept & blockheight
MBlockStride	cmdaccept & blockstride
MBurstLength	cmdaccept & burstlength
MBurstPrecise	cmdaccept & burstprecise
MBurstSeq	cmdaccept & burstseq
MBurstSingleReq	cmdaccept & burstsinglereq
MByteEn	cmdaccept & byteen
MCmd	cmdaccept
MConnID	cmdaccept & connid
MData	cmdaccept & mdata & !datahandshake
MDataInfo	cmdaccept & mdatainfo & !datahandshake
MReqInfo	cmdaccept & reqinfo
MReqLast	cmdaccept & reqlast
MThreadID	cmdaccept & threads > 1
1.2.4 request_value_MCmd_<command>	
BCST	!broadcast_enable
RDL	!rdlwrc_enable
WRC	!rdlwrc_enable
RD	!read_enable
RDEX	!readex_enable
WR	!write_enable
WRNP	!writenonpost_enable
1.2.5 request_value_<signal>_word_aligned	
MAddr	addr
MBlockStride	blockstride
1.2.6 request_value_<signal>_0x0	
MAtomicLength	atomiclength
MBurstLength	burstlength
MBlockHeight	blockheight
	blockstride
1.2.7 request_value_MBurstSeq_<sequence>	
BLCK	burstlength & !burstseq_blck_enable
DLFT1	burstlength & !burstseq_dflt1_enable
DLFT2	burstlength & !burstseq_dflt2_enable
INCR	burstlength & !burstseq_incr_enable
STRM	burstlength & !burstseq_strm_enable
UNKN	burstlength & !burstseq_unkn_enable
WRAP	burstlength & !burstseq_wrap_enable
XOR	burstlength & !burstseq_xor_enable
1.2.8 request_value_MByteEn_force_aligned	byteen & force_aligned
1.2.9 request_value_MThreadID	threads > 1

Name	Activation Parameters
1.2.10 datahs_exact_SDataThreadBusy	datahandshake & sdatathreadbusy & sdatathreadbusy_exact & ~sdatathreadbusy_pipelined
1.2.11 datahs_pipelined_SDataThreadBusy	datahandshake & sdatathreadbusy & sdatathreadbusy_exact & sdatathreadbusy_pipelined
1.2.12 datahs_hold_<signal>	
MData	dataaccept & mdata & datahandshake
MDataByteEn	mdatabyteen & dataaccept & datahandshake
MDataInfo	dataaccept & mdatainfo & datahandshake
MDataThreadID	datahandshake & threads > 1 & dataaccept
MDataValid	dataaccept & datahandshake
MDataLast	datalast & dataaccept & datahandshake
1.2.13 datahs_value_MDataByteEn_force_aligned	Mdatabyteen & datahandshake & force_aligned
1.2.14 datahs_value_MDataThreadID	datahandshake & threads > 1
1.2.15 response_exact_MThreadBusy	resp & mthreadbusy & mthreadbusy_exact & ~mthreadbusy_pipelined
1.2.16 response_pipelined_MThreadBusy	resp & mthreadbusy & mthreadbusy_exact & mthreadbusy_pipelined
1.2.17 response_hold_<signal>	
SData	respaccept & sdata & resp & (read_enable | readex_enable | rdlwrc_enable)
SDataInfo	respaccept & sdatainfo
SResp	respaccept & resp
SRespInfo	respaccept & resp & respinfo
SRespLast	respaccept & resp & resplast
SThreadID	respaccept & resp & threads > 1
1.2.18 response_value_SResp_FAIL_without_WRC	resp & rdlwrc_enable
1.2.19 response_value_SThreadID	resp & threads > 1
1.2.20 request_hold_MTagInOrder	cmdaccept & taginorder & tags > 1
1.2.21 response_hold_STagInOrder	resp & respaccept & taginorder & tags > 1
1.2.22 request_hold_MTagID_when_MTagInOrder_zero	cmdaccept & tags > 1
1.2.23 datahs_hold_MDataTagID_when_MTagInOrder_zero	datahandshake & dataaccept & tags > 1
1.2.24 response_hold_STagID_when_STagInOrder_zero	resp & respaccept & tags > 1
1.2.25 request_value_MTagID_when_MTagInOrder_zero	tags > 1
1.2.26 datahs_value_MTagID_when_MTagInOrder_zero	datahandshake & tags > 1

Name	Activation Parameters
1.2.27 response_value_STagID_when_STagInOrder_zero	resp & tags > 1
1.2.28 datahs_order_MDataTagID_when_MTagInOrder_zero	burstlength & datahandshake & tags > 1
1.2.29 response_reorder_STagID_tag_interleave_size	burstsinglereq & resp & tags > 1
1.2.30 response_reorder_STagID_overlapping_addresses	resp & tags > 1 & (addr | addrspace | byteen)

Table 81 Dataflow Burst Checks

Name	Activation Parameters
1.3.1 burst_hold_MBurstLength_precise	burstlength
1.3.2 burst_hold_<signal>	
MAddrSpace	burstlength & addrspace
MAtomicLength	burstlength & atomiclength
MBurstPrecise	burstlength & burstprecise
MBurstSeq	burstseq & burstlength
MBurstSingleReq	burstlength & burstsinglereq
MCmd	burstlength
MConnID	burstlength & connid
MReqInfo	burstlength & reqinfo
SRespInfo	burstlength & respinfo
1.3.3 burst_hold_<signal>_BLCK	
MBlockHeight	burstlength & burstseq_blck_enable & burstseq & blockheight
MBlockStride	burstlength & burstseq_blck_enable & burstseq & blockstride
1.3.4 burst_hold_<signal>_STRM	
MByteEn	burstlength & burstseq_strm_enable & byteen & burstseq
MDataByteEn	burstlength & burstseq_strm_enable & datahandshake & mdatabyteen & burstseq
1.3.5 burst_phase_order_reqdata_together	reqdata_together & datahandshake
1.3.6 burst_sequence_MAddr_BLCK	burstlength & addr & burstseq_blck_enable & burstseq
1.3.7 burst_sequence_MAddr_INCR	burstlength & addr & burstseq_incr_enable & burstseq
1.3.8 burst_sequence_MAddr_STRM	burstlength & addr & burstseq_strm_enable & burstseq
1.3.9 burst_sequence_MAddr_WRAP	burstlength & burstseq_wrap_enable & addr & burstseq
1.3.10 burst_sequence_MAddr_XOR	burstlength & burstseq_xor_enable & addr & burstseq

Name			Activation Parameters
1.3.11 burst_value_<signal>_<sequence>			
	MByteEn	STRM	burstlength & burstseq_strm_enable & byteen & burstseq
	MDataByteEn	STRM	burstseq & burstseq_strm_enable & mdatabyteen & datahandshake
	MByteEn	DFLT2	burstlength & burstseq_dflt2_enable & byteen & burstseq
	MDataByteEn	DFLT2	burstseq & burstseq_dflt2_enable & mdatabyteen & datahandshake
1.3.12 burst_value_MAddr_INCR_burst_aligned			burstlength & burstseq_incr_enable & burst_aligned & burstseq
1.3.13 burst_value_MAddr_<sequence>_no_wrap			
	INCR		burstlength & burstseq_incr_enable & addr
	BLCK		burstlength & burstseq_blck_enable & addr
1.3.14 burst_value_MBurstLength_<sequence>			
	WRAP		burstlength & burstseq & burstseq_wrap_enable
	XOR		burstlength & burstseq & burstseq_xor_enable
1.3.15 burst_value_MBurstLength_INCR_burst_aligned			burstlength & burstseq_incr_enable & burst_aligned & burstseq
1.3.16 burst_value_MBurstPrecise_<sequence>			
	WRAP		burstprecise & burstseq_wrap_enable & burstlength & burstseq
	XOR		burstprecise & burstseq_xor_enable & burstlength & burstseq
	BLCK		burstprecise & burstseq_blck_enable & burstlength & burstseq
1.3.17 burst_value_MBurstPrecise_INCR_burst_aligned			burstaligned & burstprecise & burstseq_incr_enable & burstseq
1.3.18 burst_value_MBurstPrecise_SRMD			burstprecise & burstsinglereq
1.3.19 burst_value_MBurstSeq_UNKN_SRMD			burstsinglereq & burstreq & burstseq_unkn_enable
1.3.20 burst_value_MCmd_<command>			
	RDEX		burstlength & readex_enable
	RDL		burstlength & rdlwrc_enable
	WRC		burstlength & rdlwrc_enable
1.3.21 burst_value_MReqLast_MRMD			reqlast
1.3.22 burst_value_MReqLast_SRMD			Reqlast & burstsinglereq
1.3.23 burst_value_MReqRowLast_MRMD			mreqlast &mreqrowlast
1.3.24 burst_value_MReqRowLast_SRMD			mreqlast &mreqrowlast
1.3.25 burst_value_MDataLast_MRMD			datalast & mdata
1.3.26 burst_value_MDataLast_SRMD			datalast & mdata
1.3.27 burst_value_MDataRowLast_MRMD			mdatalast & mdatarowlast

Name	Activation Parameters
1.3.28 burst_value_MDataRowLast_SRMD	mdatalast & mdatarowlast
1.3.29 burst_value_SRespLast_MRMD	resplast & resp
1.3.30 burst_value_SRespLast_SRMD	resplast & resp & burstsinglereq
1.3.31 burst_value_SRespRowLast_MRMD	sresplast & sresprowlast
1.3.32 burst_value_SRespRowLast_SRMD	sresplast & sresprowlast
1.3.33 burst_hold_MTagID_when_MTagInOrder_zero	burtstlength & tags > 1
1.3.34 burst_hold_MTagInOrder	burstlength & tags > 1 & taginorder

Table 82 Dataflow Transfer Checks

Name	Activation Parameters
1.4.1 transfer_phase_order_datahs_before_request_begin	datahandshake
1.4.2 transfer_phase_order_datahs_before_request_end	datahandshake
1.4.3 transfer_phase_order_response_before_request_begin	resp
1.4.4 transfer_phase_order_response_before_request_end	resp
1.4.5 transfer_phase_order_response_before_datahs_begin	resp & datahandshake
1.4.6 transfer_phase_order_response_before_datahs_end	resp & datahandshake
1.4.7 transfer_phase_order_response_before_last_datahs_begin _SRMD_wr	resp & datahandshake & burstsinglereq
1.4.8 transfer_phase_order_response_before_last_datahs_end_ SRMD_wr	resp & datahandshake & burstsinglereq
1.4.9 transfer_phase_order_reqdata_together_MRMD	reqdata_together & burstsinglereq

Table 83 Dataflow ReadEx Checks

Name	Activation Parameters
1.5.1 rdex_hold_<signal>	
MAddr	readex_enable & addr
MAddrSpace	readex_enable & addrspace
MByteEn	readex_enable & byteen
MDataByteEn)	readex_enable & mdatabyteen & datahandshake
1.5.3 rdex_lock_release_no_burst_allowed	burstlength & readex_enable

Table 84 Sideband Checks

Name	Activation Parameters
1.6.1 signal_valid_<signal> MReset_n SReset_n	 mreset sreset
1.6.2 signal_valid_<signal>_when_reset_inactive ControlBusy ControlWr MError SError SInterrupt StatusBusy StatusRd	 controlbusy controlwr merror serror interrupt statusbusy statusrd
1.6.3 signal_hold_<signal>_16_cycles MReset_n SReset_n	 mreset sreset
1.6.4 signal_hold_Control_after_reset	control
1.6.5 signal_hold_Control_2_cycles	control
1.6.6 signal_hold_Control_ControlBusy_active	controlbusy
1.6.7 signal_hold_ControlWr_after_reset	controlwr
1.6.8 signal_value_ControlWr_Control_transitioned	control & controlwr
1.6.9 signal_value_ControlWr_ControlBusy_active	controlbusy & controlwr
1.6.10 signal_hold_ControlWr_2_cycle	controlwr
1.6.11 signal_value_ControlBusy	controlwr & controlbusy
1.6.12 signal_hold_StatusRd_2_cycles	statusrd
1.6.13 signal_value_StatusRd_StatusBusy_active	statusrd & statusbusy

Table 85 Connection Protocol Checks

Name	Activation Parameters
1.7.1 signal_valid_<signal> MConnect SConnect SWait	connection
1.7.2 signal_hold_MConnect_2_cycles	connection
1.7.3 signal_value_MCmd_MConnect_not_connected	connection
1.7.4 signal_order_MConnect_transaction	connection
1.7.5 signal_value_SWait_MConnect_stable_state	connection
1.7.6 signal_value_SConnect_MConnect_connected	connection
1.7.7 signal_value_SConnect_MConnect_connected	connection
1.7.8 signal_value_MConnect_ConnectCap	connection
1.7.9 signal_value_SConnect_ConnectCap	connection
1.7.10 signal_value_SWait_ConnectCap	connection

18.2 Compliance Checks

18.2.1 Dataflow Signals Checks

Rule 1.1.1 signal_valid_<signal>_when_reset_inactive

When reset is inactive, the following signals should never have an X or Z value on the rising edge of the OCP clock:

MCmd	MDataValid	MThreadBusy
SDataThreadBusy	SResp	SThreadBusy

Protocol hierarchy	Reset activity
Signal group	Dataflow
Critical signals	MCmd, MDataValid, MThreadBusy, SDataThreadBusy, SResp, SThreadBusy
Assertion type	X, Z
Reference	Section 4.3.3.1 on page 46

Rule 1.1.2 request_valid_<signal>

The following signals should never have an X or Z value on the rising edge of the OCP clock during a request phase:

MAddr	MAddrSpace	MAtomicLength
MBurstLength	MBurstPrecise	MBurstSeq
MBurstSingleReq	MByteEn	MConnID
MReqLast	MThreadID	SCmdAccept
MBlockHeight	MBlockStride	MReqRowLast

MBlockHeight and MBlockStride can be invalid for non-BLCK requests during the request phase.

If `datahandshake=1` and `mdatabyteen=1` then MByteEn can be invalid for write accesses during the request phase

Protocol hierarchy	Request phase
Signal group	Dataflow
Critical signals	MAddr, MAddrSpace, MAtomicLength, MBurstLength, MBurstPrecise, MBurstSeq, MBurstSingleReq, MByteEn, MConnID, MReqLast, MThreadID, SCmdAccept, MBlockHeight, MBlockStride, MReqRowLast
Assertion type	X, Z
References	Section 4.3.3.1 on page 46 Section 12.1.2.1 on page 215

Rule 1.1.3 datahs_valid_<signal>

The following signals should never have an X or Z value on the rising edge of the OCP clock during a datahandshake phase:

MDataByteEn	MDataLast	MDataRowLast
MDataThreadID	SDataAccept	

Protocol hierarchy	Datahandshake
Signal group	Dataflow
Critical signals	MDataByteEn, MDataLast, MDataRowLast, MDataThreadID, SDataAccept
Assertion type	X, Z
Reference	Section 4.3.3.1 on page 46 Section 12.1.2.3 on page 217

Rule 1.1.4 response_valid_<signal>

The following signals should never have an X or Z value on the rising edge of the OCP clock during a response phase:

MRespAccept	SRespLast	SRespRowLast
SThreadID		

Protocol hierarchy	Response
Signal group	Dataflow
Critical signals	MRespAccept, SRespLast, SRespRowLast, SThreadID
Assertion type	X, Z
Reference	Section 4.3.3.1 on page 46 Section 12.1.2.2 on page 216

Rule 1.1.5 request_valid_MTagInOrder

MTagInOrder should not be X/Z during the request phase.

Protocol hierarchy	Request
Signal group	Dataflow - tag extensions
Critical signals	MTagInOrder
Assertion type	X, Z
Reference	Section 12.4 on page 232

Rule 1.1.6 response_valid_STagInOrder

STagInOrder should not be X/Z during the response phase.

Protocol hierarchy	Response
Signal group	Dataflow - tag extensions
Critical signals	STagInOrder
Assertion type	X, Z
Reference	Section 12.4 on page 232

Rule 1.1.7 request_valid_MTagID_when_MTagInOrder_zero

If MTagInOrder is 0 during the request phase, MTagID should not be X/Z during the request phase.

Protocol hierarchy	Response
Signal group	Dataflow - tag extensions
Critical signals	MTagID, MTagInOrder
Assertion type	X, Z
Reference	Section 12.4 on page 232

Rule 1.1.8 datahs_valid_MDataTagID_when_MTagInOrder_zero

If datahandshake is active and MTagInOrder is 0 (during the request phase), MDataTagID should not be X/Z during the datahandshake phase.

Protocol hierarchy	Datahandshake
Signal group	Dataflow - tag extensions
Critical signals	MDataTagID, MTagInOrder
Assertion type	X, Z
Reference	Section 12.4 on page 232

Rule 1.1.9 response_valid_STagID_when_STagInOrder_zero

If STagInOrder is 0 during the request phase, STagID should not be X/Z during the response phase.

Protocol hierarchy	Response
Signal group	Dataflow - tag extensions
Critical signals	STagID, STagInOrder
Assertion type	X, Z
Reference	Section 12.4 on page 232

18.2.2 DataFlow Phase Checks

Rule 1.2.1 request_exact_SThreadBusy

If `sthreadbusy_exact` = 1 and `sthreadbusy_pipelined` = 0, when a given slave thread is busy, the master must stay idle on this thread.

Protocol hierarchy	Request
Signal group	Dataflow - thread extensions
Critical signals	MCmd
Assertion type	Value
References	Section 4.3.2.4 on page 44 Section 12.5.1 on page 233

Rule 1.2.2 request_pipelined_SThreadBusy

If `sthreadbusy_exact` = 1 and `sthreadbusy_pipelined` = 1, and an SThreadbusy bit was set to 1 in the prior cycle, the master cannot present a request on a thread in the current cycle.

Protocol hierarchy	Request
Signal group	Dataflow - thread extensions
Critical signals	MCmd
Assertion type	Value
Reference	Section 4.3.2.4 on page 44

Rule 1.2.3 request_hold_<signal>

Once a request phase has begun, the following signals may not change their value until the OCP slave has accepted the request.

Basic Signals
MAddr
MCmd
MData

Simple Extensions
MAddrSpace
MByteEn
MDataInfo
MReqInfo

Burst Extensions
MAtomicLength
MBurstLength
MBurstPrecise
MBurstSeq
MBurstSingleReq
MReqLast
Thread Extensions
MConnID
MThreadID
MBlockHeight
MBlockStride
MReqRowLast

The following exceptions apply:

1. If datahandshake=1 and mdatabyteen=1 then MByteEn can change for write accesses during the request phase.
2. For read requests the MData and MDataInfo fields can change during the request phase.
3. For write requests the SData and SDataInfo fields can change during the response phase.
4. Non-enabled data bytes in MData and bits in MDataInfo fields can change during the request and datahandshake phases.
5. Non-enabled data bytes in SData and bits in SDataInfo fields can change during the response phase.
6. MDataByteEn can change during read-type transfers.

7. MTagID can change if MTagInOrder is asserted, and MDataTagID can change for the corresponding datahandshake phase.

8. STagID can change if STagInOrder is asserted.

Protocol hierarchy	Request
Signal group	Dataflow
Critical signals	MAddr, MCmd, MData, MAddrSpace, MByteEn, MDataInfo, MReqInfo, MAtomicLength, MBurstLength, MBurstPrecise, MBurstSeq, MBurstSingleReq, MReqLast, MConnID, MThreadID, MBlockHeight, MBlockStride, MReqRowLast
Assertion type	Hold
References	Section 12.1.2.1 on page 215

Rule 1.2.4 request_value_MCmd_<command>

The following <Command> is illegal if the corresponding <Parameter> is set to 0.

Command	**Parameter**
BCST	broadcast_enable
RD	read_enable
RDEX	readex_enable
RDL	rdlwrc_enable
WR	write_enable
WRC	rdlwrc_enable
WRNP	writenonpost_enable

Protocol hierarchy	Request
Signal group	Dataflow - basic signals
Critical signals	MCmd
Assertion type	Value
Reference	Section 4.9.1.1 on page 59

Rule 1.2.5 request_value_MAddr_word_aligned

Signal MAddr must be OCP word aligned as follows:

if data_wdth = 16 then MAddr[0] = 0
if data_wdth = 32 then MAddr[1:0] = 0
if data_wdth = 64 then MAddr[2:0] = 0
if data_wdth = 128 then MAddr[3:0] = 0

Protocol hierarchy	Request
Signal group	Dataflow - basic signals
Critical signals	MAddr
Assertion type	Value
Reference	Section 3.1.1 on page 13

Rule 1.2.6 request_value_<signal>_0x0

During a request phase:

MAtomicLength and MBurstLength must not be zero.
If MBurstSeq != BLCK, MBlockHeight and MBlockStride values are don't care.
If MBurstSeq == BLCK, MBlockHeight must be greater than zero.
If MBurstSeq == BLCK and MBlockHeight > 1, MBlockStride must be greater than zero.

Protocol hierarchy	Request
Signal group	Dataflow - burst extensions
Critical signals	MAtomicLength, MBurstLength, MBlockHeight, MBlockStride
Assertion type	Value
References	Section 3.1.3 on page 19 Footnotes on page 34

Rule 1.2.7 request_value_MBurstSeq_<sequence>

The following <burst type> is illegal if its corresponding <parameter > is set to 0.

Burst type	**Parameter**
BLCK	burstseq_blck_enable
DLFT1	burstseq_dflt1_enable
DLFT2	burstseq_dflt2_enable
INCR	burstseq_incr_enable
STRM	burstseq_strm_enable

UNKN	burstseq_unkn_enable
WRAP	burstseq_wrap_enable
XOR	burstseq_xor_enable

Protocol hierarchy	Request
Signal group	Dataflow - burst extensions
Critical signals	MBurstSeq
Assertion type	Value
References	Section 4.9.1.2 on page 59

Rule 1.2.8 request_value_MByteEn_force_aligned

If `force_aligned`=1, the byte enable values during a request phase are restricted to the following patterns for `data_wdth` ≥ 32:

`data_wdth`=32 and MByteEn has one of the following values:

0001
0010
0100
1000
0011
1100
1111
0000

`data_wdth`=64 and MByteEn has one of the following values:

00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000
00000011
00001100
00110000
11000000
00001111
11110000
11111111
00000000

`data_wdth`=128 and MByteEn has one of the following values:

0000000000000001
0000000000000010
0000000000000100
0000000000001000

0000000000010000
0000000000100000
0000000001000000
0000000010000000
0000000100000000
0000001000000000
0000010000000000
0000100000000000
0001000000000000
0010000000000000
0100000000000000
1000000000000000
0000000000000011
0000000000001100
0000000000110000
0000000011000000
0000001100000000
0000110000000000
0011000000000000
1100000000000000
0000000000001111
0000000011110000
0000111100000000
1111000000000000
0000000011111111
1111111100000000
1111111111111111
0000000000000000

If `datahandshake`=1 and `mdatabyteen`=1 then MByteEn can change for write accesses during the request phase.

Protocol hierarchy	Request
Signal group	Dataflow - simple extensions
Critical signals	MByteEn
Assertion type	Value
References	Section 4.9.1.3 on page 60

Rule 1.2.9 request_value_MThreadID

MThreadID value is always < threads.

Protocol hierarchy	Request
Signal group	Dataflow - thread extensions
Critical signals	MThreadID
Assertion type	Value
Reference	Section 3.1.5 on page 23

Rule 1.2.10 datahs_exact_SDataThreadBusy

If `sdatathreadbusy_exact` = 1 and `sdatathreadbusy_pipelined` = 0, when a given slave data thread is busy, the master must not present a data phase on this thread.

Protocol hierarchy	Datahandshake
Signal group	Dataflow - thread extensions
Critical signals	MDataValid
Assertion type	Value
Reference	Section 4.3.2.4 on page 44

Rule 1.2.11 datahs_pipelined_SDataThreadBusy

If `sdatathreadbusy_exact` = 1 and `sdatathreadbusy_pipelined` = 1, and an SDataThreadbusy bit was set to 1 in the prior cycle, the master cannot present a datahandshake on a thread in the current cycle.

Protocol hierarchy	Datahandshake
Signal group	Dataflow - thread extensions
Critical signals	MDataValid
Assertion type	Value
Reference	Section 4.3.2.4 on page 44

Rule 1.2.12 datahs_hold_<signal>

Once a datahandshake phase has begun, the following signals may not change their value until the OCP slave has accepted the data.

Basic Signals	**Burst Extensions**
MData	MDataLast
MDataValid	MDataRowLast
Simple Extensions	**Thread Extensions**
MDataByteEn	MDataThreadID
MDataInfo	

Protocol hierarchy	Datahandshake
Signal group	Dataflow
Critical signals	MData, MDataByteEn, MDataInfo, MDataLast, MDataRowLast, MDataThreadID, MDataValid,
Assertion type	Hold
Reference	Section 12.1.2.3 on page 217

Rule 1.2.13 datahs_value_MDataByteEn_force_aligned

If `force_aligned`=1, the data byte enable values during a datahandshake phase are restricted to the following patterns for `data_wdth` ≥ 32:

`data_wdth`=32 and MDataByteEn has one of the following values:

0001
0010
0100
1000
0011
1100
1111
0000

`data_wdth`=64 and MDataByteEn has one of the following values:

00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000
00000011
00001100
00110000
11000000
00001111
11110000
11111111
00000000

`data_wdth`=128 and MDataByteEn has one of the following values:

0000000000000001
0000000000000010
0000000000000100
0000000000001000
0000000000010000
0000000000100000
0000000001000000

0000000010000000
0000000100000000
0000001000000000
0000010000000000
0000100000000000
0001000000000000
0010000000000000
0100000000000000
1000000000000000
0000000000000011
0000000000001100
0000000000110000
0000000011000000
0000001100000000
0000110000000000
0011000000000000
1100000000000000
0000000000001111
0000000011110000
0000111100000000
1111000000000000
0000000011111111
1111111100000000
1111111111111111
0000000000000000

Protocol hierarchy	Datahandshake
Signal group	Dataflow - simple extensions
Critical signals	MDataByteEn
Assertion type	Value
Reference	Section 4.9.1.3 on page 60

Rule 1.2.14 datahs_value_MDataThreadID

MDataThreadID value must be < threads.

Protocol hierarchy	Datahandshake
Signal group	Dataflow - thread extensions
Critical signals	MDataThreadID
Assertion type	Value
Reference	Section 3.1.5 on page 23

Rule 1.2.15 response_exact_MThreadBusy

If `mthreadbusy_exact` = 1 and `mthreadbusy_pipelined` = 0, when a given master thread is busy, the slave must not present a response on that thread.

Protocol hierarchy	Response
Signal group	Dataflow - thread extensions
Critical signals	SResp
Assertion type	Value
Reference	Section 4.3.2.4 on page 44

Rule 1.2.16 response_pipelined_MThreadBusy

If `sthreadbusy_exact` = 1 and `sthreadbusy_pipelined` = 1, and an MThreadbusy bit was set to 1 in the prior cycle, the slave cannot present a response on a thread in the current cycle.

Protocol hierarchy	Response
Signal group	Dataflow - thread extensions
Critical signals	SResp
Assertion type	Value
Reference	Section 4.3.2.4 on page 44

Rule 1.2.17 response_hold_<signal>

Once a response phase has begun, the following signals may not change their value until the master has accepted the response.

Basic Signals	**Burst Extensions**
SData	SRespLast
SResp	SRespRowLast
Simple Extensions	**Thread Extensions**
SDataInfo	SThreadID
SRespInfo	

Protocol hierarchy	Response
Signal group	Dataflow
Critical signals	SData, SDataInfo, SResp, SRespInfo, SRespLast, SRespRowLast, SThreadID
Assertion type	Hold
Reference	Section 12.1.2.2 on page 216

Rule 1.2.18 response_value_SResp_FAIL_without_WRC

The FAIL response can occur only on a WRC request.

Protocol hierarchy	Response
Signal group	Dataflow - basic signals
Critical signals	SResp
Assertion type	Value
References	Section 4.4 on page 49

Rule 1.2.19 response_value_SThreadID

SThreadID value must be < threads.

Protocol hierarchy	Response
Signal group	Dataflow - thread extensions
Critical signals	SThreadID
Assertion type	Value
Reference	Section 3.1.5 on page 23

Rule 1.2.20 request_hold_MTagInOrder

If taginorder = 1, the MTagInOrder signal cannot change until accepted by the OCP slave (SCmdAccept = 1).

Protocol hierarchy	Request
Signal group	Dataflow - tag extensions
Critical signals	MTagInOrder
Assertion type	Hold
Reference	Section 12.1.2.1 on page 215

Rule 1.2.21 response_hold_STagInOrder

If taginorder = 1, the STagInOrder signal cannot change until accepted by the master (MRespAccept = 1).

Protocol hierarchy	Response
Signal group	Dataflow - tag extensions
Critical signals	STagInOrder
Assertion type	Hold
Reference	Section 12.1.2.2 on page 216

Rule 1.2.22 request_hold_MTagID_when_MTagInOrder_zero

If tags > 1, the MTagID signal cannot change until accepted by the OCP slave (SCmdAccept = 1).

Protocol hierarchy	Request
Signal group	Dataflow - tag extensions
Critical signals	MTagID, MTagInOrder
Assertion type	Hold
Reference	Section 12.1.2.1 on page 215

Rule 1.2.23 datahs_hold_MDataTagID_when_MTagInOrder_zero

When tags > 1, during a datahandshake phase corresponding to a non in-order request phase (MTagInOrder = 0), the MDataTagID signal cannot change value until accepted by the OCP slave (SDataAccept = 1).

Protocol hierarchy	Datahandshake
Signal group	Dataflow - tag extensions
Critical signals	MDataTagID, MTagInOrder
Assertion type	Hold
Reference	Section 12.1.2.3 on page 217

Rule 1.2.24 response_hold_STagID_when_STagInOrder_zero

If tags > 1, the STagID signal cannot change until it is accepted by the master (MRespAccept = 1).

Protocol hierarchy	Response
Signal group	Dataflow - tag extensions
Critical signals	STagID, STagInOrder
Assertion type	Hold
Reference	Section 12.1.2.2 on page 216

Rule 1.2.25 request_value_MTagID_when_MTagInOrder_zero

The MTagID signal must always be < tags.

Protocol hierarchy	Request
Signal group	Dataflow - tag extensions
Critical signals	MTagID, MTagInOrder
Assertion type	Value
Reference	Section 3.1.4 on page 22

Rule 1.2.26 datahs_value_MTagID_when_MTagInOrder_zero

The MDataTagID signal must always be < tags.

Protocol hierarchy	Datahandshake
Signal group	Dataflow - tag extensions
Critical signals	MDataTagID, MTagInOrder
Assertion type	Value
Reference	Section 3.1.4 on page 22

Rule 1.2.27 response_value_STagID_when_STagInOrder_zero

The STagID signal must always be < tags.

Protocol hierarchy	Response
Signal group	Dataflow - tag extensions
Critical signals	STagID, STagInOrder
Assertion type	Value
Reference	Section 3.1.4 on page 22

Rule 1.2.28 datahs_order_MDataTagID_when_MTagInOrder_zero

When `datahandshake` = 1, for tagged write transactions, the datahandshake phase must observe the same order as the request phase.

Protocol hierarchy	Datahandshake
Signal group	Dataflow - tag extensions
Critical signals	MDataTagID, (MTagInOrder
Assertion type	Data_order
Reference	Section 4.7.1 on page 57

Rule 1.2.29 response_reorder_STagID_tag_interleave_size

When `tags` > 1 and `tag_interleave_size` > 0 the slave must ensure that responses associated with packing burst sequences stay together up to the `tag_interleave_size`. When `tags` > 1 and `tag_interleave_size` == 0 no interleaving of responses between any packing burst sequences with different tags is allowed.

Protocol hierarchy	Response
Signal group	Dataflow - tag extensions
Critical signals	STagID
Assertion type	Reorder
References	Section 4.7.1 on page 57 Section 4.9.1.7 on page 62

Rule 1.2.30 response_reorder_STagID_overlapping_addresses

Responses to requests with different tags on the same thread that target overlapping addresses (as determined by MAddrSpace, MAddr, and MByteEn [or MDataByteEn, if applicable]) can be re-ordered with respect to another.

Protocol hierarchy	Response
Signal group	Dataflow - tag extensions
Critical signals	STagID
Assertion type	Reorder
References	Section 4.7.1 on page 57

Rule 1.2.31 request_value_MBlockStride_word_aligned

Signal MBlockStride must be OCP word aligned as follows:

if data_wdth = 16 then MBlockStride[0] = 0
if data_wdth = 32 then MBlockStride[1:0] = 0
if data_wdth = 64 then MBlockStride[2:0] = 0
if data_wdth = 128 then MBlockStride[3:0] = 0.

Protocol hierarchy	Request
Signal group	Dataflow - burst extensions
Critical signals	MBlockStride
Assertion type	Value
References	Section 3.1.3 on page 19

18.2.3 Dataflow Burst Checks

Rule 1.3.1 burst_hold_MBurstLength_precise

For precise bursts, MBurstLength must hold its value during all request phases of the entire burst.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MBurstLength
Assertion type	Hold
References	Section 4.6.3 on page 55

Rule 1.3.2 burst_hold_<signal>

The following signals must hold the same value on all request phases of the entire burst:

MAddrSpace	MBurstSingleReq
MAtomicLength	MCmd
MBurstPrecise	MConnID
MBurstSeq	MReqInfo

The hold requirements for SRespInfo in a burst are different for the 2.0 versus 2.2 specifications.

OCP 2.0 page 44 states that:
SRespInfo must be held steady by the slave for every transfer in a burst.

OCP 2.2 page 55 states that:
If possible, slaves should hold SRespInfo steady for every transfer in a burst

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MAddrSpace, MAtomicLength, MBurstPrecise, MBurstSeq, MBurstSingleReq, MCmd, MConnID, MReqInfo, (SRespInfo)
Assertion type	Hold
Reference	Section 4.6.3 on page 55

Rule 1.3.3 burst_hold_<signal>_BLCK

<signal> must hold for BLCK bursts. Applicable to MBlockHeight and MBlockStridesignals.

Protocol hierarchy	Burst
Signal group	Dataflow - simple extensions
Critical signals	MBlockHeight, MBlockStride
Assertion type	Hold
Reference	Section 4.6.3 on page 55

Rule 1.3.4 burst_hold_<signal>_STRM

For STRM bursts, MByteEn / MDataByteEn must hold the same value on all request / datahandshake phases of the entire burst.

Protocol hierarchy	Burst
Signal group	Dataflow - simple extensions
Critical signals	MByteEn, MDataByteEn
Assertion type	Hold
References	Section 4.6.1.1 on page 54

Rule 1.3.5 burst_phase_order_reqdata_together

For single request multiple data bursts, if reqdata_together = 1, the master must present the request and first write data in the same cycle, and the slave must accept the request and the first write data in the same cycle.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MCmd, MDataValid
Assertion type	Ordering
Reference	Section 4.9.2 on page 63

Rule 1.3.6 burst_sequence_MAddr_BLCK

Within a block burst, the address begins with the provided address and proceeds through a set of MBlockHeight subsequences, each of which follows the normal INCR burst sequence for MBurstLength transfers. The starting address of each subsequence should be the starting address of the prior subsequence plus MBlockStride.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MBlockHeight, MBurstLength, MBlockStride
Assertion type	Ordering
Reference	Section 4.6.1 on page 53

Rule 1.3.7 burst_sequence_MAddr_INCR

Within an INCR burst, the address increases for each new master request by the OCP word size.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MAddr
Assertion type	Ordering
Reference	Table 23 on page 53

Rule 1.3.8 burst_sequence_MAddr_STRM

Within a STRM burst, the address remains constant on all request phases of the burst.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MAddr
Assertion type	Ordering
Reference	Table 23 on page 53

Rule 1.3.9 burst_sequence_MAddr_WRAP

Within a WRAP burst, the address increases for each new master request by the OCP word size, and wraps on the burst length x OCP word size.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MAddr
Assertion type	Ordering
Reference	Table 23 on page 53

Rule 1.3.10 burst_sequence_MAddr_XOR

Within an XOR burst, the address increases for each new OCP master request as follows:

BASE
: Is the lowest byte address in the burst, which must be aligned with the total burst size.

FIRST_OFFSET
Is the byte offset (from BASE) of the first transfer in the burst.

CURRENT_COUNT
Is the count of current transfer in the burst starting at 0.

WORD_SHIFT
Is the log2 of the OCP word size in bytes.

The current address of the transfer is BASE | (FIRST_OFFSET ^ (CURRENT_COUNT << WORD_SHIFT)).

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MAddr
Assertion type	Ordering
Reference	Section 4.6.1 on page 53

Rule 1.3.11 burst_value_<signal>_<sequence>

When mdatabyteen = 0, during STRM or DFLT2 bursts, MByteEn should never take the value 0.

When mdatabyteen = 1, during read-type STRM or DFLT2 bursts, MByteEn should never take the value 0.

When mdatabyteen = 1, during write-type STRM or DFLT2 bursts, MDataByteEn should never take value 0.

Protocol hierarchy	Burst
Signal group	Dataflow - simple extensions
Critical signals	MByteEn, MDataByteEn
Assertion type	Value
Reference	Section 4.6.1.1 on page 54

Rule 1.3.12 burst_value_MAddr_INCR_burst_aligned

When `burst_aligned`=1, the first burst request of an INCR burst must have its address aligned. The equation below indicates which MAddr bits must be 0.

Equation

MAddr [(size-1)+BL:0] = 0
Where:
size = ceil(log2(bytes(data_width)))for data_width > 1 byte
BL = log2(MBurstLength)for MBurstLength $\geq$ 1

Example

For an interface with data_width=32, size=2 and:
MBurstLength = 2:MAddr[2:0] = 0
MBurstLength = 4:MAddr[3:0] = 0

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MAddr
Assertion type	Value
References	Section 4.9.1.4 on page 60

Rule 1.3.13 burst_value_MAddr_<sequence>_no_wrap

An INCR or BLCK burst can never cross the address space boundary.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MAddr
Assertion type	Value
Reference	Section 4.6.1 on page 53

Rule 1.3.14 burst_value_MBurstLength_<sequence>

The length of a WRAP or XOR burst must be a power of two.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MBurstLength
Assertion type	Value
Reference	Section 4.6.1 on page 53

Rule 1.3.15 burst_value_MBurstLength_INCR_burst_aligned

When `burst_aligned` = 1, the length of an INCR burst must be a power of two.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MBurstLength
Assertion type	Value
References	Section 4.9.1.4 on page 60

Rule 1.3.16 burst_value_MBurstPrecise_<sequence>

BLCK, WRAP and XOR bursts can be issued only as precise bursts.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MBurstPrecise
Assertion type	Value
References	Section 4.6.1 on page 53

Rule 1.3.17 burst_value_MBurstPrecise_INCR_burst_aligned

When `burst_aligned` = 1, INCR bursts can be issued only as precise bursts.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MBurstPrecise
Assertion type	Value
Reference	Section 4.6.1 on page 53

Rule 1.3.18 burst_value_MBurstPrecise_SRMD

Single request multiple data transfers can be issued only as precise bursts.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MBurstPrecise
Assertion type	Value
Reference	Section 4.6.5 on page 55

Rule 1.3.19 burst_value_MBurstSeq_UNKN_SRMD

An unknown burst sequence (value UNKN) is illegal during a single request multiple data transfer.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MBurstSeq
Assertion type	Value
Reference	Section 4.3.2.1 on page 42

Rule 1.3.20 burst_value_MCmd_<command>

The RDEX, RDL, and WRC commands cannot be part of a burst.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MCmd
Assertion type	Value
References	Section 4.6 on page 52

Rule 1.3.21 burst_value_MReqLast_MRMD

The signal MReqLast must be 0 for all request phases of a MRMD burst, except on the last one when it must be 1. For BLCK bursts the last request phase is the last request phase of the last MBlockHeight subsequence.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MReqLast
Assertion type	Value
References	Section 4.6.6 on page 56

Rule 1.3.22 burst_value_MReqLast_SRMD

The signal MReqLast must be 1 for any single request (SRMD being active or not).

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MReqLast
Assertion type	Value
References	Section 4.6.6 on page 56

Rule 1.3.23 burst_value_MReqRowLast_MRMD

For BLCK bursts the signal MReqRowLast must be 0 for all request phases other than the last phases in each row, when it must be 1. For non-BLCK bursts the signal MReqRowLast must be 0 for all request phases of a MRMD burst, except on the last one when it must be 1. When `mreqlast` and `mreqrowlast` are both enabled, whenever MReqLast is asserted MReqRowLast must also be asserted.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MReqRowLast
Assertion type	Ordering
Reference	Section 4.6.6 on page 56

Rule 1.3.24 burst_value_MReqRowLast_SRMD

The signal MReqRowLast must be 1 for any single request (SRMD being active or not).

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MReqRowLast
Assertion type	Ordering
Reference	Section 4.6.6 on page 56

Rule 1.3.25 burst_value_MDataLast_MRMD

The MDataLast signal must be 0 for all datahandshake phases in an MRMD burst, except on the last one when it must be 1. For BLCK bursts the last datahandshake phase is the last datahandshake phase of the last MBlock-Height subsequence.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MDataLast
Assertion type	Value
References	Section 4.6.6 on page 56

Rule 1.3.26 burst_value_MDataLast_SRMD

The MDataLast signal must be 0 for all datahandshake phases of an SRMD burst, except on the last one when it must be 1. For BLCK bursts the last datahandshake phase is the last datahandshake phase of the last MBlock-Height subsequence.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MDataLast
Assertion type	Value
References	Section 4.6.6 on page 56

Rule 1.3.27 burst_value_MDataRowLast_MRMD

For BLCK bursts the signal MDataRowLast must be 0 for all datahandshake phases other than the last phases in each row, when it must be 1. For non-BLCK bursts the signal MDataRowLast must be 0 for all datahandshake phases of a MRMD burst, except on the last one when it must be 1. If `mdatalast` and `mdatarowlast` are both enabled, whenever MDataLast is asserted MDataRowLast must also be asserted.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MDataRowLast, MDataLast
Assertion type	Value
References	Section 4.6.6 on page 56

Rule 1.3.28 burst_value_MDataRowLast_SRMD

For BLCK bursts the signal MDataRowLast must be 0 for all datahandshake phases other than the last phases in each row, when it must be 1. For non-BLCK bursts the signal MDataRowLast must be 0 for all datahandshake phases of a SRMD burst, except on the last one when it must be 1. When mdatalast and mdatarowlast are both enabled, whenever MDataLast is asserted MDataRowLast must also be asserted.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MDataLast
Assertion type	Value
References	Section 4.6.6 on page 56

Rule 1.3.29 burst_value_SRespLast_MRMD

The signal SRespLast must be 0 for all response phases of an MRMD burst, except on the last one where it must be 1. For BLCK bursts the last response phase is the last response phase of the last MBlockHeight subsequence.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	SRespLast
Assertion type	Value
Reference	Section 4.6.6 on page 56

Rule 1.3.30 burst_value_SRespLast_SRMD

The signal SRespLast must be 1 for any single response (with SRMD active or not).

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	SRespLast
Assertion type	Value
Reference	Section 4.6.6 on page 56

Rule 1.3.31 burst_value_SRespRowLast_MRMD

For BLCK bursts the signal MRespRowLast must be 0 for all response phases other than the last phases in each row, when it must be 1. For non-BLCK bursts the signal MRespRowLast must be 0 for all response phases of a MRMD burst, except on the last one when it must be 1. If `sresplast` and `sresprowlast` are both enabled, whenever SRespLast is asserted SRespRowLast must also be asserted.

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MRespRowLast, SRespLast
Assertion type	Value
Reference	Section 4.6.6 on page 56

Rule 1.3.32 burst_value_SRespRowLast_SRMD

The signal MRespRowLast must be 1 for any single response (SRMD being active or not).

Protocol hierarchy	Burst
Signal group	Dataflow - burst extensions
Critical signals	MRespRowLast
Assertion type	Value
Reference	Section 4.6.6 on page 56

Rule 1.3.33 burst_hold_MTagID_when_MTagInOrder_zero

The MTagID signal must remain constant for all transfers of a burst when MTagInOrder is zero. The master cannot interleave requests (or datahandshake) phases with different tags within a transaction. This check should only focus on the request phase. The datahandshake phase is covered by phase property "datahs_order_MDataTagID_when_ MTagInOrder_zero." This last property checks that the datahandshake phase observes the same order as the request phase.

Protocol hierarchy	Burst
Signal group	Dataflow - tag extensions
Critical signals	MTagID, (MTagInOrder)
Assertion type	Hold
Reference	Section 4.7.1 on page 57

Rule 1.3.34 burst_hold_MTagInOrder

The MTagInOrder signal must remain constant for all transfers of a burst.

Protocol hierarchy	Burst
Signal group	Dataflow - tag extensions
Critical signals	MTagInOrder
Assertion type	Hold
Reference	Section 4.7.1 on page 57

18.2.4 DataFlow Transfer Checks

Rule 1.4.1 transfer_phase_order_datahs_before_request_begin

For each thread, for each transaction tag, a datahandshake phase cannot begin before the associated request phase begins, but can begin in the same clock cycle.

Protocol hierarchy	Transfer
Signal group	Dataflow - basic signals
Critical signals	MDataValid, MCmd
Assertion type	Ordering
Reference	Section 4.3.2.2 on page 43

Rule 1.4.2 transfer_phase_order_datahs_before_request_end

For each thread, for each transaction tag, a datahandshake phase cannot end before the associated request phase ends, but can end in the same clock cycle.

Protocol hierarchy	Transfer
Signal group	Dataflow - basic signals
Critical signals	MDataValid, MCmd
Assertion type	Ordering
Reference	Section 4.3.2.2 on page 43

Rule 1.4.3 transfer_phase_order_response_before_request_begin

For each thread, for each transaction tag, a response phase cannot begin before the associated request phase begins, but can begin in the same clock cycle.

Protocol hierarchy	Transfer
Signal group	Dataflow - basic signals
Critical signals	MCMd, SResp
Assertion type	Ordering
References	Section 4.3.2.2 on page 43

Rule 1.4.4 transfer_phase_order_response_before_request_end

For each thread, for each transaction tag, a response phase cannot end before the associated request phase ends, but can end in the same clock cycle.

Protocol hierarchy	Transfer
Signal group	Dataflow - basic signals
Critical signals	MCMd, SResp
Assertion type	Ordering
References	Section 4.3.2.2 on page 43

Rule 1.4.5 transfer_phase_order_response_before_datahs_begin

For each thread, for each transaction tag, when datahandshake = 1, the response phase cannot begin before the associated datahandshake begins, but can begin in the same clock cycle.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MDataValid, SResp
Assertion type	Ordering
References	Section 4.3.2.2 on page 43

Rule 1.4.6 transfer_phase_order_response_before_datahs_end

For each thread, for each transaction tag, when datahandshake = 1, the response phase cannot end before the associated datahandshake ends, but can end in the same clock cycle.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MDataValid, SResp
Assertion type	Ordering
References	Section 4.3.2.2 on page 43

Rule 1.4.7 transfer_phase_order_response_before_last_datahs_begin _SRMD_wr

For each thread, for each transaction tag, with a write-type SRMD, the response phase cannot begin before the last datahandshake phase begins, but it can begin in the same clock cycle.

Protocol hierarchy	Transfer
Signal group	Dataflow - basic signals
Critical signals	MDataValid, SResp
Assertion type	Ordering
References	Section 4.3.2.2 on page 43

Rule 1.4.8 transfer_phase_order_response_before_last_datahs_end_SRMD_wr

For each thread, for each transaction tag, with a write-type SRMD, the response phase cannot end before the last datahandshake phase ends, but it can end in the same clock cycle.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MDataValid, SResp
Assertion type	Ordering
Reference	Section 4.3.2.2 on page 43

Rule 1.4.9 transfer_phase_order_reqdata_together_MRMD

For multiple request multiple data bursts, if both reqdata_together and burstsinglereq are set to 1, the master must present the request and the associated write data in the same cycle for each transfer, and the slave must accept the request and the associated write data in the same cycle.

Protocol hierarchy	Burst
Signal group	Dataflow - basic signals
Critical signals	MCmd, MDataValid
Assertion type	Ordering
Reference	Section 4.9.2 on page 63

18.2.5 DataFlow ReadEx Checks

Rule 1.5.1 rdex_hold_<signal>

The unlocking command following a ReadEx must retain the same address and address space values

When mdatabyteen = 0, the unlocking command following a ReadEx must retain the same MByteEn value.

When mdatabyteen = 1, the unlocking command following a ReadEx must retain for MDataByteEn the value given to MByteEn during the ReadEx command. If MByteEn is absent, MDataByteEn must be all 1s.

Protocol hierarchy	ReadEx
Signal group	Dataflow - basic signals, simple extensions
Critical signals	MAddr, MAddrSpace, MByteEn, MDataByteEn
Assertion type	Hold
References	Section 4.4 on page 49 Section 4.6 on page 52

~~Rule 1.5.2 rdex_lock_release_no_WR/WRNP~~

~~If a ReadEx is issued on an address on a particular thread, no other request with the same address can be issued on any other thread until the ReadEx is unlocked.~~

~~The command following the ReadEx on the same thread must be a write command (WR or WRNP). This command unlocks the ReadEx.~~

~~**Protocol hierarchy**~~	~~ReadEx~~
~~**Signal group**~~	~~Dataflow - basic signals~~
~~**Critical signals**~~	~~MCmd~~
~~**Assertion type**~~	~~Ordering~~
~~**References**~~	~~Section 4.4 on page 49~~ ~~Section 4.6 on page 52~~

Rule 1.5.3 rdex_lock_release_no_burst_allowed

The unlocking command following a RDEX must have MBurstLength = 1.

Protocol hierarchy	ReadEx
Signal group	Dataflow - basic signals
Critical signals	MBurstLength
Assertion type	Value
Reference	Section 4.4 on page 49 Section 4.6 on page 52

18.3 Sideband Checks

Rule 1.6.1 signal_valid_<signal>

Signals MReset_n and SReset_n are never X or Z.

Protocol hierarchy	Reset activity
Signal group	Sideband - reset
Critical signals	MReset, SReset
Assertion type	X, Y
Reference	Section 4.3.3.1 on page 46 Section 4.3.3.4 on page 48

Rule 1.6.2 signal_valid_<signal>_when_reset_inactive

When reset is inactive, the following signals should never have an X or Z value on the rising edge of the OCP clock:

ControlBusy	ControlWr	MError
SError	SInterrupt	
StatusBusy	StatusRd	

Protocol hierarchy	Reset activity
Signal group	Sideband - reset
Critical signals	ControlBusy, ControlWr, MError, SError, SInterrupt, StatusBusy, StatusRd
Assertion type	X, Y
Reference	Section 4.3.3.4 on page 48

Rule 1.6.3 signal_hold_<signal>_16_cycles

If they are active, signals MReset_n and SReset_n must stay active at least 16 consecutive cycles.

Protocol hierarchy	Reset activity
Signal group	Sideband - reset
Critical signals	MReset, SReset
Assertion type	Hold
References	Section 4.3.3.1 on page 46

Rule 1.6.4 signal_hold_Control_after_reset

The Control signal must be held steady for the first two cycles after reset is de-asserted.

Protocol hierarchy	Control
Signal group	Sideband - control
Critical signals	Control
Assertion type	Hold
References	Section 4.3.3.4 on page 48

Rule 1.6.5 signal_hold_Control_2_cycles

The Control signal must be held steady for a full cycle after the cycle in which it has transitioned.

Protocol hierarchy	Control
Signal group	Sideband - control
Critical signals	Control
Assertion type	Hold
References	Section 4.3.3.4 on page 48

Rule 1.6.6 signal_hold_Control_ControlBusy_active

If the ControlBusy signal was sampled active at the end of the previous cycle, the Control signal must not transition in the current cycle.

Protocol hierarchy	Control
Signal group	Sideband - control
Critical signals	Control
Assertion type	Value
Reference	Section 4.3.3.4 on page 48

Rule 1.6.7 signal_hold_ControlWr_after_reset

The ControlWr signal must not be asserted in the cycle following a reset.

Protocol hierarchy	Control
Signal group	Sideband - control
Critical signals	ControlWr
Assertion type	Hold
Reference	Section 4.3.3.4 on page 48

Rule 1.6.8 signal_value_ControlWr_Control_transitioned

If signal Control transitions in a cycle, signal ControlWr must be driven active on that cycle.

Protocol hierarchy	Control
Signal group	Sideband - control
Critical signals	ControlWr
Assertion type	Hold
References	Section 4.3.3.4 on page 48

Rule 1.6.9 signal_value_ControlWr_ControlBusy_active

The ControlWr signal must not be asserted if ControlBusy is active.

Protocol hierarchy	Control
Signal group	Sideband - control
Critical signals	ControlWr
Assertion type	Value
Reference	Section 4.3.3.4 on page 48

Rule 1.6.10 signal_hold_ControlWr_2_cycle

The ControlWr signal must not remain asserted for two consecutive cycles.s

Protocol hierarchy	Control
Signal group	Sideband - control
Critical signals	ControlWr
Assertion type	Hold
Reference	Section 4.3.3.4 on page 48

Rule 1.6.11 signal_value_ControlBusy

The ControlBusy signal can only be asserted in a cycle after the ControlWr signal is asserted or after the reset transitions to inactive.

Protocol hierarchy	Control
Signal group	Sideband - control
Critical signals	ControlBusy
Assertion type	Value
Reference	Section 4.3.3.4 on page 48

Rule 1.6.12 signal_hold_StatusRd_2_cycles

If the StatusRd signal was asserted in the previous cycle, it must not be asserted in the current cycle.

Protocol hierarchy	Status
Signal group	Sideband - status
Critical signals	StatusRd
Assertion type	Hold
References	Section 4.3.3.4 on page 48

Rule 1.6.13 signal_value_StatusRd_StatusBusy_active

The StatusRd signal must not be asserted while StatusBusy is asserted.

Protocol hierarchy	Status
Signal group	Sideband - status
Critical signals	StatusRd, StatusBusy
Assertion type	Value
Reference	Section 4.3.3.4 on page 48

18.4 Connection Protocol Checks

Rule 1.7.1 disconnect_signal_valid_<signal>

The disconnect signals (listed below) are always valid, including during the OCP reset

MConnect	SConnect	SWait

Protocol hierarchy	Reset activity
Signal group	Sideband
Critical signals	MConnect, SConnect, SWait
Assertion type	X, Z
Reference	Section 4.3.3.2 on page 46

Rule 1.7.2 signal_hold_MConnect_2_cycles

The MConnect signal must be held steady for a full cycle after the cycle in which MConnect has transitioned to M_CON, M_DISC, or M_OFF.

Protocol hierarchy	
Signal group	Sideband
Critical signals	MConnect
Assertion type	Hold
References	Section 4.3.3.2 on page 46

Rule 1.7.2 signal_value_MCmd_MConnect_not_connected

The MCmd signal must be IDLE if MConnect is not in the M_CON state.

Protocol hierarchy	Control
Signal group	Sideband
Critical signals	MCmd, MConnect
Assertion type	Value
Reference	Section 4.3.3.2 on page 46

Rule 1.7.3 signal_order_MConnect_transaction

If signal MConnect transitions from M_CON in a cycle, there should not be any non-finshed OCP transaction at that cycle.

Protocol hierarchy	
Signal group	Sideband
Critical signals	Inband OCP signals
Assertion type	Value
References	Section 4.3.3.2 on page 46

Rule 1.7.4 signal_value_SWait_MConnect_stable_state

If signal MConnect transitions to a stable state (M_OFF, M_DISC, M_CON) in a cycle, SWait must be 0 (S_OK) at that cycle.

Protocol hierarchy	
Signal group	Sideband
Critical signals	MConnect, SWait
Assertion type	Value
References	Section 4.3.3.2 on page 46

Rule 1.7.5 signal_value_SConnect_MConnect_connected

If signal MConnect transitions to M_CON in a cycle, SConnect must be 1 (S_CON) in the previous cycle.

Protocol hierarchy	
Signal group	Sideband
Critical signals	MConnect, SConnect
Assertion type	Value
References	Section 4.3.3.2 on page 46

Rule 1.7.6 signal_value_SConnect_MConnect_disconnected

If signal MConnect transitions to M_DISC in a cycle, SConnect must be 0 (S_DISC) at that cycle.

Protocol hierarchy	
Signal group	Sideband
Critical signals	MConnect, SConnect
Assertion type	Value
References	Section 4.3.3.2 on page 46

Rule 1.7.7 signal_value_MConnect_ConnectCap

If ConnectCap is 0, Master must stay connected, so MConnect is M_CON.

Protocol hierarchy	
Signal group	Sideband
Critical signals	MConnect
Assertion type	Value
Reference	Section 3.2.1 on page 26

Rule 1.7.8 signal_value_SConnect_ConnectCap

If ConnectCap is 0, Slave must stay connected, so SConnect is S_CON.

Protocol hierarchy

Signal group	Sideband
Critical signals	SConnect
Assertion type	Value
Reference	Section 3.2.1 on page 26

Rule 1.7.9 signal_value_SWait_ConnectCap

If ConnectCap is 0, Slave must not stall disconnect interface, so SWait is S_OK.

Protocol hierarchy

Signal group	Sideband
Critical signals	SWait
Assertion type	Value
Reference	Section 3.2.1 on page 26

19 Configuration Compliance Checks

The configuration checks listed in this chapter are extracted from the OCP Specification and are intended to serve as guidelines to verify an IP for OCP compliance. In all cases "Part I, Specification" is the definitive reference.

The configuration checks listed in this chapter are based on the "Specification" and "Guidelines" parts of this document and allow you to verify an IP/VIP for OCP compliance. In all cases, "Part I, Specification" is the definitive reference. Any references made to "Part II, Guidelines" are not definitive as Part I supersedes the guidelines.

The section describes the configuration checks needed for an OCP port. The names assigned to the configuration compliance checks have been created using the following template:

<hierarchy>_cfg_<critical_param>_<relationship>_<extra_details>

In which:

<hierarchy>: request, datahandshake, response, sideband, test, master_slave
<critical_param> : any OCP parameter that is impacted by the configuration check
<relationship> : (optional) enable, depends, match
<extra details> : a short additional explanation

The majority of the configuration checks involve an enable relationship. For these enable checks 'paramA_enable_paramB' implies that paramA is somehow enabled by paramB. In these situations the individual check descriptions provide details on the enabling relationship between the parameters.

19.1 Request Group

Rule 2.1.1 request_cfg_cmd_enable

One of the command enable parameters must be enabled. The critical parameters are: `read_enable`, `readex_enable`, `write_enable`, `writenonpost_enable`, `broadcast_enable`, or `rdlwrc_enable`.

Protocol hierarchy	Request
Critical parameters	read_enable, readex_enable, write_enable, writenonpost_enable, broadcast_enable, rdlwrc_enable
Reference	Section 3.1.1 on page 13

Rule 2.1.2 request_cfg_readex_enable_write_writenonpost

`readex_enable` can only be enabled if `write_enable` or `writenonpost_enable` is enabled.

Protocol hierarchy	Request
Critical parameters	readex_enable, write_enable, writenonpost_enable
Reference	Section 4.9.1.1 on page 59

Rule 2.1.3 request_cfg_addr_wdth_depends_data_wdth

`data_wdth` defines a minimum `addr_wdth` value that is based on the data bus byte width, and is defined as:

$$\text{min\_addr\_wdth} = \max[1, \lfloor \log_2(\text{data\_wdth}) \rfloor - 2]$$

Protocol hierarchy	Request
Critical parameters	addr_wdth, data_wdth
Reference	MAddr on page 14

Rule 2.1.4 request_cfg_blockstride_wdth_depends_data_wdth

If the blockstride parameter is enabled, then data_wdth defines a minimumblockstride_wdth value:

$$\text{min\_blockstride\_wdth} = \max[1, \lfloor \log_2(\text{data\_wdth}) \rfloor - 2]$$

Protocol hierarchy	Request
Critical parameters	blockstride_wdth, data_wdth
Reference	MBlockStride on page 20

Rule 2.1.5 request_cfg_byteen_enable_mdata_sdata

`byteen` can only be enabled when either `sdata` or `mdata` is also enabled.

Protocol hierarchy	Request
Critical parameters	byteen, sdata, mdata
Reference	Section 3.1.2 on page 16

Rule 2.1.6 request_cfg_byteen_enable_data_wdth

`byteen` is only supported when `data_wdth` is a multiple of 8.

Protocol hierarchy	Request
Critical parameters	byteen, data_wdth
Reference	Section 3.1.2 on page 16

Rule 2.1.7 request_cfg_sthreadbusy_exact_enable_sthreadBusy

`sthreadbusy_exact`can only be enabled if `sthreadbusy` is enabled.

Protocol hierarchy	Request
Critical parameters	sthreadbusy, sthreadbusy_exact
Reference	Table 26 on page 61

Rule 2.1.8 request_cfg_sdata_enable_resp

`sdata` can only be enabled if `resp` is enabled.

Protocol hierarchy	Request
Critical parameters	sdata, resp
Reference	Table 26 on page 61

Rule 2.1.9 request_cfg_sthreadbusy_enable_sthreadbusy_exact_ cmdaccept

`sthreadbusy` can only be enabled if one of `sthreadbusy_exact` or `cmdaccept` is enabled.

Protocol hierarchy	Request
Critical parameters	sthreadbusy, sthreadbusy_exact, cmdaccept
Reference	Table 26 on page 61

Rule 2.1.10 request_cfg_atomiclength_enable_burstlength

`atomiclength` can only be enabled if `burstlength` is enabled.

Protocol hierarchy	Request
Critical parameters	atomiclength, burstlength
Reference	Footnotes on page 34

Rule 2.1.11 request_cfg_burstprecise_enable_burstlength

`burstprecise` can only be enabled if `burstlength` is enabled.

Protocol hierarchy	Request
Critical parameters	burstprecise, burstlength
Reference	Footnotes on page 34

Rule 2.1.12 request_cfg_burstseq_enable_burstlength

`burstseq` can only be enabled if `burstlength` is enabled.

Protocol hierarchy	Request
Critical parameters	burstseq, burstlength
Reference	Footnotes on page 34

Rule 2.1.13 request_cfg_burstsinglereq_enable_burstlength

`burstsinglereq` can only be enabled if `burstlength` is enabled.

Protocol hierarchy	Request
Critical parameters	burstsinglereq, burstlength
Reference	Footnotes on page 34

Rule 2.1.14 request_cfg_reqlast_enable_burstlength

`reqlast` can only be enabled if `burstlength` is enabled.

Protocol hierarchy	Request
Critical parameters	reqlast, burstlength
Reference	Footnotes on page 34

Rule 2.1.15 request_cfg_reqrowlast_enable_burstlength

`reqrowlast` can only be enabled if `burstlength` is also enabled.

Protocol hierarchy	Request
Critical parameters	reqrowlast, burstlength
Reference	Footnotes on page 34

Rule 2.1.16 request_cfg_reqrowlast_enable_reqlast_burstseq_blck_enable

`reqrowlast` can only be enabled if `reqlast` and `burstseq_blck_enable` are enabled.

Protocol hierarchy	Request
Critical parameters	reqlast, reqrowlast, burstseq_blck_enable
Reference	Footnotes on page 34

Rule 2.1.17 request_cfg_burstlength_enable_burstseq_enable

`burstlength` can only be enabled if at least one of the burst sequences is enabled.

Protocol hierarchy	Request
Critical parameters	burstlength
Reference	Section 3.1.3 on page 19

Rule 2.1.18 request_cfg_burstseq_<type>_enable_burstseq

`burstseq_<type>_enable` can only be enabled if `burstseq` is enabled. `burstseq` must be enabled if two or more burst sequences (specified by `burstseq_<type>_enable`) are enabled.

Protocol hierarchy	Request
Critical parameters	burstseq, burstseq_<type>_enable
Reference	Section 4.9.1.2 on page 59

Rule 2.1.19 request_cfg_reqdata_together_enable_burstsinglereq

`reqdata_together` can only be enabled if `burstsinglereq` is enabled.

Protocol hierarchy	Request
Critical parameters	reqdata_together, burstsinglereq
Reference	Section on page 63

Rule 2.1.20 request_cfg_force_aligned_enable_data_wdth

`force_aligned` can only be enabled if `data_wdth` is a power of 2

Protocol hierarchy	Request
Critical parameters	force_aligned, data_wdth
Reference	Section 4.9.1.3 on page 60

Rule 2.1.21 request_cfg_mdatainfo_enable_mdata

`mdatainfo` can only be enabled if `mdata` is enabled.

Protocol hierarchy	Request
Critical parameters	mdatainfo, mdata
Reference	MDatInfo on page 17

Rule 2.1.22 request_cfg_sdatainfo_enable_sdata

`sdatainfo` can only be enabled if `sdata` is enabled.

Protocol hierarchy	Request
Critical parameters	sdatainfo, sdata
Reference	MDatInfo on page 18

Rule 2.1.23 request_cfg_atomiclength_wdth_depends_burstlengthwdth

`atomiclength_wdth` must be less than or equal to `burstlength_wdth`.

Protocol hierarchy	Request
Critical parameters	atomiclength_wdth, burstlength_wdth
Reference	Footnotes on page 34

Rule 2.1.24 request_cfg_value_burstlength_wdth_0x1

`burstlength_wdth` must be greater than 1 if `burstlength` is enabled.

Protocol hierarchy	Request
Critical parameters	burstlength, burstlength_wdth
Reference	Footnotes on page 34

Rule 2.1.25 request_cfg_burst_aligned_enable_burstlength

`burst_aligned` can only be enabled if `burstlength` is enabled.

Protocol hierarchy	Request
Critical parameters	burst_aligned, burstlength
Reference	Section 4.9.1.4 on page 60

Rule 2.1.26 request_cfg_burstseq_enable_addr

`burstseq` can only be enabled if `addr` is enabled.

Protocol hierarchy	Request
Critical parameters	addr, burstseq
Reference	MBurstSeq on page 21

Rule 2.1.27 request_cfg_burstsinglereq_enable_burstseq_enable

`burstsinglereq` must be disabled if `burstseq_unkn_enable` is the only enabled burst sequence.

Protocol hierarchy	Request
Critical parameters	burstseq_unkn_enable, burstsinglereq
Reference	Section 4.6.1 on page 53

Rule 2.1.28 request_cfg_taginorder_enable_tags

`taginorder` is only enabled if `tags` > 1.

Protocol hierarchy	Request
Critical parameters	taginorder, tags
Reference	Footnotes on page 34

Rule 2.1.29 request_cfg_tag_interleave_size_depends_ burstlength_wdth

`tag_interleave_size` must be 1 if `burstlength_wdth` is 0 and otherwise must be 0 or a power-of-two which is less than or equal to 2**(burstlength_ wdth-1).

Protocol hierarchy	Request
Critical parameters	tag_interleave_size, burstlength_wdth
Reference	Section 4.9.1.7 on page 62

Rule 2.1.30 request_cfg_<block_signal>_enable_burstseq_blck_enable

`blockheight` and `blockstride` are only enabled when `burstseq_blck_enable` is also enabled.

Protocol hierarchy	Request
Critical parameters	blockheight, blockstride, burstseq_blck_enable
Reference	Footnotes on page 34

Rule 2.1.31 request_cfg_value_blockheight_wdth_0x1

If `blockheight` is enabled, `blockheight_wdth` must be greater than 1.

Protocol hierarchy	Request
Critical parameters	blockheight, blockheight_wdth
Reference	Footnotes on page 34

Rule 2.1.32 request_cfg_<threadbusy_pipelined_cfg>_enable_<threadbusy_exact_cfg>

The parameters `mthreadbusy_pipelined`, `sdatathreadbusy_pipelined`, and `sthreadbusy_pipelined` can be enabled to 1 only when the corresponding _exact parameter is enabled.

Protocol hierarchy	Request
Critical parameters	mthreadbusy_pipelined, sdatathreadbusy_pipelined, and sthreadbusy_pipelined
Reference	Section 4.3.2.4 on page 44

Rule 2.1.33 request_cfg_reqdata_together_enable_cmdaccept_dataaccept

`reqdata_together` can only be enabled if `cmdaccept` and `dataaccept` match.

Protocol hierarchy	Request
Critical parameters	reqdata_together, cmdaccept, dataaccept
Reference	Implicit

19.2 Datahandshake Group

Rule 2.2.1 datahandshake_cfg_datalast_enable_burstlength

`datalast` can only be enabled if `burstlength` is enabled.

Protocol hierarchy	Request
Critical parameters	datalast, burstlength
Reference	Footnotes on page 34

Rule 2.2.2 request_cfg_burstsinglereq_enable_datahandshake_cmd_enable

`burstsinglereq` can only be enabled if `datahandshake` is enabled or none of the write command types are enabled.

Protocol hierarchy	Datahandshake
Critical parameters	burstsinglereq, datahandshake, write_enable, writenonpost_enable, rdlwrc_enable
Reference	Section 4.6.5 on page 55

Rule 2.2.3 datahandshake_cfg_datahandshake_enable_mdata

`datahandshake` can only be enabled if `mdata` is also enabled.

Protocol hierarchy	Datahandshake
Critical parameters	datahandshake, mdata
Reference	Section 3.1.1 on page 13

Rule 2.2.4 datahandshake_cfg_datalast_enable_datahandshake

`datalast` can only be enabled if `datahandshake` is also enabled.

Protocol hierarchy	Datahandshake
Critical parameters	datalast, datahandshake
Reference	Section 3.1.3 on page 19

Rule 2.2.5 datahandshake_cfg_datarowlast_enable_datahandshake

`datarowlast` can only be enabled if `datahandshake` is also enabled.

Protocol hierarchy	Datahandshake
Critical parameters	datarowlast, datahandshake
Reference	Section 3.1.3 on page 19

Rule 2.2.6 datahandshake_cfg_datrowalast_enable_burstlength

`datarowlast` can only be enabled if `burstlength` is also enabled.

Protocol hierarchy	Datahandshake
Critical parameters	datarowlast, burstlength
Reference	Section 3.1.3 on page 19

Rule 2.2.7 datahandshake_cfg_datarowlast_enable_datalast_ burstseq_blck_enable

`datarowlast` can only be enabled if `datalast` and `burstseq_blck_enable` are enabled.

Protocol hierarchy	Datahandshake
Critical parameters	datarowlast, datalast, datahandshake, burstseq_blck_ enable
Reference	Section 3.1.3 on page 19

Rule 2.2.8 datahandshake_cfg_dataaccept_enable_datahandshake

`dataaccept` can only be enabled if `datahandshake` is also enabled.

Protocol hierarchy	Datahandshake
Critical parameters	dataaccept, datahandshake
Reference	Section 3.1.1 on page 13

Rule 2.2.9 datahandshake_cfg_sdatathreadbusy_enable_ datahanshake

`sdatathreadbusy` can only be enabled if `datahandshake` is also enabled.

Protocol hierarchy	Datahandshake
Critical parameters	sdatathreadbusy, datahandshake
Reference	Section 3.1.5 on page 23

Rule 2.2.10 datahandshake_cfg_mdatabyteen_enable_datahanshake

`mdatabyteen` can only be enabled if `datahandshake` is also enabled.

Protocol hierarchy	Datahandshake
Critical parameters	mdatabyteen, datahandshake
Reference	Section 3.1.2 on page 16

Rule 2.2.11 datahandshake_cfg_mdatabyteen_enable_mdata

`mdatabyteen` can only be enabled if `mdata` is also enabled.

Protocol hierarchy	Datahandshake
Critical parameters	mdatabyteen, mdata
Reference	Section 3.1.2 on page 16

Rule 2.2.12 datahandshake_cfg_mdatabyteen_depends_data_wdth

`mdatabyteen` can only be enabled if `data_wdth` is a multiple of 8.

Protocol hierarchy	Datahandshake
Critical parameters	mdatabyteen, data_wdth
Reference	Section 3.1.2 on page 16

Rule 2.2.13 datahandshake_cfg_mdatainfo_depends_data_wdth

`mdatainfo` can only be enabled if `data_wdth` is a multiple of 8.

Protocol hierarchy	Datahandshake
Critical parameters	mdatainfo, data_wdth
Reference	Section 3.1.2 on page 16

Rule 2.2.14 datahandshake_cfg_mdatainfo_wdth_depends_ mdatainfobyte_wdth

`mdatainfo_wdth` must be greater than or equal to `mdatainfobyte_wdth` * `data_wdth` / 8.

Protocol hierarchy	Datahandshake
Critical parameters	mdatainfo_wdth, mdatainfobyte_wdth, data_wdth
Reference	Section 3.1.2 on page 16

Rule 2.2.15 datahandshake_cfg_sdatainfo_depends_data_wdth

`sdatainfo` can only be enabled if `data_wdth` is a multiple of 8.

Protocol hierarchy	Datahandshake
Critical parameters	sdatainfo, data_wdth
Reference	Section 3.1.2 on page 16

Rule 2.2.16 datahandshake_cfg_sdatainfo_wdth_depends_s datainfobyte_wdth

`sdatainfo_wdth` must be greater than or equal to `sdatainfobyte_wdth` * `data_wdth` /8.

Protocol hierarchy	Datahandshake
Critical parameters	sdatainfo_wdth, sdatainfobyte_wdth, data_wdth
Reference	Section 3.1.2 on page 16

Rule 2.2.17 datahandshake_cfg_sdatathreadbusy_enable_ sdatathreadbusy_exact

`sdatathreadbusy_exact` can only be enabled if `sdatathreadbusy` is enabled.

Protocol hierarchy	Datahandshake
Critical parameters	sdatathreadbusy, sdatathreadbusy_exact
Reference	Table 26 on page 61

~~Rule 2.2.18 datahandshake_cfg_sdatathreadbusy_exact_enable_ sdatathreadbusy~~

~~sdatathreadbusy_exact can onlybe enabled if sdatathreadbusy is enabled.~~

Protocol hierarchy	Datahandshake
Critical parameters	sdatathreadbusy_exact, sdatathreadbusy
Reference	Table 26 on page 61

Rule 2.2.19 datahandshake_cfg_dataaccept_enable_ sdatathreadbusy_exact

`dataaccept` can only be enabled if `sdatathreadbusy_exact` is not enabled.

Protocol hierarchy	Datahandshake
Critical parameters	dataaccept, sdatathreadbusy_exact
Reference	Table 26 on page 61

Rule 2.2.20 datahandshake_cfg_reqdata_together_enable_ datahandshake

`reqdata_together` is only enabled if `datahandshake` is enabled.

Protocol hierarchy	Datahandshake
Critical parameters	reqdata_together, datahandshake
Reference	Table 24 on page 59

19.3 Response Group

Rule 2.3.1 response_cfg_resplast_enable_burstlength

`resplast` can only be enabled if `burstlength` is enabled.

Protocol hierarchy	Response
Critical parameters	resplast, burstlength
Reference	Footnotes on page 34

Rule 2.3.2 response_cfg_respaccept_enable_resp

`respaccept` can only be enabled if `resp` is also enabled.

Protocol hierarchy	Response
Critical parameters	respaccept, resp
References	Section 3.1.2 on page 16 Footnotes on page 34

Rule 2.3.3 response_cfg_resplast_enable_resp

`resplast` can only be enabled if `resp` is also enabled.

Protocol hierarchy	Response
Critical parameters	resplast, resp
References	Section 3.1.3 on page 19 Footnotes on page 34

Rule 2.3.4 response_cfg_resprowlast_enable_resp

`resprowlast` can only be enabled if `resp` is also enabled.

Protocol hierarchy	Response
Critical parameters	resprowlast, resp
References	Section 3.1.3 on page 19 Footnotes on page 34

Rule 2.3.5 response_cfg_resprowlast_enable_burstlength

`resprowlast` can only be enabled if `burstlength` is also enabled.

Protocol hierarchy	Response
Critical parameters	resprowlast, burstlength
References	Section 3.1.3 on page 19 Footnotes on page 34

Rule 2.3.6 response_cfg_resprowlast_enable_resplast_burstseq_blck_enable

`resprowlast` can only be enabled if `resplast` and `burstseq_blck_enable` are enabled.

Protocol hierarchy	Response
Critical parameters	resprowlast, resplast, burstseq_blck_enable
References	Section 3.1.3 on page 19 Footnotes on page 34

Rule 2.3.7 response_cfg_respinfo_enable_resp

`respinfo` can only be enabled if `resp` is also enabled.

Protocol hierarchy	Response
Critical parameters	respinfo, resp
References	Section 3.1.3 on page 19 Footnotes on page 34

Rule 2.3.8 response_cfg_mthreadbusy_enable_resp

`mthreadbusy` can only be enabled if `resp` is enabled.

Protocol hierarchy	Response
Critical parameters	mthreadbusy, resp
References	Section 3.1.3 on page 19 Footnotes on page 34

Rule 2.3.9 response_cfg_sdata_enable_resp

`sdata` can only be enabled if `resp` is also enabled.

Protocol hierarchy	Response
Critical parameters	sdata, resp
References	Section 3.1.2 on page 16 Footnotes on page 34

Rule 2.3.10 response_cfg_sdatainfo_enable_resp

`sdatainfo` can only be enabled if `resp` is also enabled.

Protocol hierarchy	Response
Critical parameters	sdatainfo, resp
References	Section 3.1.2 on page 16 Footnotes on page 34

~~Rule 2.3.11 response_cfg_<cmd_enable>_enable_writeresp_enable~~

~~writenonpost_enable and rdlwrc_enable are only enabled if writeresp_enable is enabled.~~

~~**Protocol hierarchy**~~	~~Response~~
~~**Critical parameters**~~	~~writenonpost_enable, writeresp_enable, rdlwrc_enable~~
~~**Reference**~~	~~Section 4.9.1.1 on page 59~~

Rule 2.3.12 response_cfg_<cmd_enable>_enable_resp

`read_enable` and `rdlwrc_enable` are only enabled if `resp` is enabled.

Protocol hierarchy	Response
Critical parameters	read_enable, rdlwrc_enable, resp
References	"Section 3.1.1 on page 13

Rule 2.3.13 response_cfg_mthreadbusy_exact_enable_mthreadbusy

`mthreadbusy_exact` can only be enabled if `mthreadbusy` is enabled.

Protocol hierarchy	Response
Critical parameters	mthreadbusy, mthreadbusy_exact
Reference	Section 4.9.1.5 on page 61

Rule 2.3.14 response_cfg_respaccept_enable_mthreadbusy_exact

`respaccept` can only be enabled if `mthreadbusy_exact` is not enabled.

Protocol hierarchy	Response
Critical parameters	respaccept, mthreadbusy_exact
Reference	Table 26 on page 61

Rule 2.3.15 response_cfg_mthreadbusy_enable_mthreadbusy_exact_respaccept

`mthreadbusy` can only be enabled if exactly one of `mthreadbusy_exact` and `respaccept` is enabled.

Protocol hierarchy	Response
Critical parameters	mthreadbusy, mthreadbusy_exact respaccept
Reference	Table 26 on page 61

19.4 Sideband Group

Rule 2.4.1 sideband_cfg_statusbusy_enable_status

`statusbusy` can only be enabled if `status` is enabled.

Protocol hierarchy	Sideband
Critical parameters	statusbusy, status
Reference	Footnotes on page 34

Rule 2.4.2 sideband_cfg_mreset_sreset

Either `mreset` or `sreset` must be enabled.

Protocol hierarchy	Sideband
Critical parameters	mreset, sreset
Reference	Section 3.2 on page 25

Rule 2.4.3 sideband_cfg_controlwr_enable_control

`controlwr` can only be enabled if `control` is enabled.

Protocol hierarchy	Sideband
Critical parameters	control, controlwr
Reference	Footnotes on page 34

Rule 2.4.4 sideband_cfg_controlbusy_enable_control

`controlbusy` is enabled but `control` is not enabled.

Protocol hierarchy	Sideband
Critical parameters	control, controlbusy
Reference	Footnotes on page 34

Rule 2.4.5 sideband_cfg_controlbusy_enable_controlwr

`controlbusy` is enabled but `controlwr` is not enabled.

Protocol hierarchy	Sideband
Critical parameters	controlbusy, controlwr
Reference	Footnotes on page 34

Rule 2.4.6 sideband_cfg_statusrd_enable_status

`statusrd` can only be enabled if `status` is enabled.

Protocol hierarchy	Sideband
Critical parameters	status, statusrd
Reference	Footnotes on page 34

~~Rule 2.4.7 sideband_cfg_statusbusy_enable_status~~

~~`statusbusy` can only be enabled if `status` is enabled.~~

~~**Protocol hierarchy**~~	~~Sideband~~
~~**Critical parameters**~~	~~status, statusbusy~~
~~**Reference**~~	~~Footnotes on page 34~~

19.5 Test Group

Rule 2.5.1 test_cfg_jtagreset_enable_jtag_enable

`jtagtrst_enable` can only be enabled if `jtag_enable` is also enabled.

Protocol hierarchy	Test
Critical parameters	jtagtrst_enable, jtag_enable
Reference	Footnotes on page 34

19.6 Interface Interoperability

The checks contained in this section identify configuration checks for connected devices. These checks are written under the assumption that the configurations accurately reflect the enabled protocol features of the individual devices. They do not reflect exceptions that are noted in the specification and that are acceptable when used in conjunction with tie-offs.

Rule 2.6.1 master_slave_cfg_read_enable_match

If the slave has `read_enable` set to 0, the master must have `read_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	read_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.2 master_slave_cfg_readex_enable_match

If the slave has `readex_enable` set to 0, the master must have `readex_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	readex_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.3 master_slave_cfg_rdlwrc_enable_match

If the slave has `rdlwrc_enable` set to 0, the master must have `rdlwrc_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	rdlwrc_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.4 master_slave_cfg_write_enable_match

If the slave has `write_enable` set to 0, the master must have `write_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	write_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.5 master_slave_cfg_writenonpost_enable_match

If the slave has `writenonpost_enable` set to 0, the master must have `writenonpost_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	writenonpost_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.6 master_slave_cfg_broadcast_enable_match

If the slave has `broadcast_enable` set to 0, the master must have `broadcast_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	broadcast_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.7 master_slave_cfg_burstseq_blck_enable_match

If the slave has `burstseq_blck_enable` set to 0, the master must have `burstseq_blck_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	burstseq_blck_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.8 master_slave_cfg_burstseq_incr_enable_match

If the slave has `burstseq_incr_enable` set to 0, the master must have `burstseq_incr_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	burstseq_incr_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.9 master_slave_cfg_burstseq_strm_enable_match

If the slave has `burstseq_strm_enable` set to 0, the master must have `burstseq_strm_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	burstseq_strm_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.10 master_slave_cfg_burstseq_dflt1_enable_match

If the slave has `burstseq_dflt1_enable` set to 0, the master must have `burstseq_dflt1_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	burstseq_dflt1_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.11 master_slave_cfg_burstseq_dflt2_enable_match

If the slave has `burstseq_dflt2_enable` set to 0, the master must have `burstseq_dflt2_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	burstseq_dflt2_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.12 master_slave_cfg_burstseq_wrap_enable_match

If the slave has `burstseq_wrap_enable` set to 0, the master must have `burstseq_wrap_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	burstseq_wrap_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.13 master_slave_cfg_burstseq_xor_enable_match

If the slave has `burstseq_xor_enable` set to 0, the master must have `burstseq_xor_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	burstseq_xor_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.14 master_slave_cfg_burstseq_unkn_enable_match

If the slave has `burstseq_unkn_enable` set to 0, the master must have `burstseq_unkn_enable` set to 0.

Protocol hierarchy	Request
Critical parameters	`burstseq_unkn_enable`
Reference	Section 4.9.5 on page 64

Rule 2.6.15 master_slave_cfg_force_aligned_match

If the slave has `force_aligned`, the master has `force_aligned` or it must limit itself to aligned byte enable patterns.

Protocol hierarchy	Request
Critical parameters	force_aligned
Reference	Section 4.9.5 on page 64

Rule 2.6.16 master_slave_cfg_mdatabyteen_match

Configuration of the `mdatabyteen` parameter is identical between master and slave.

Protocol hierarchy	Request
Critical parameters	mdatabyteen
Reference	Section 4.9.5 on page 64

Rule 2.6.17 master_slave_cfg_burst_aligned_match

If the slave has `burst_aligned`, the master has `burst_aligned` or it must limit itself to issue all INCR bursts using `burst_aligned` rules.

Protocol hierarchy	Request
Critical parameters	burst_aligned
Reference	Section 4.9.5 on page 64

Rule 2.6.18 master_slave_cfg_<threadbusy_param>_match

If the interface includes SThreadBusy, the `sthreadbusy_exact` and `sthreadbusy_pipelined` parameters are identical between master and slave.

Protocol hierarchy	Response
Critical parameters	sthreadbusy_exact, sthreadbusy_pipelined
Reference	Section 4.9.5 on page 64

Rule 2.6.19 master_slave_cfg_<mthreadbusy_param>_match

If the interface includes MThreadBusy, the `mthreadbusy_exact` and `mthreadbusy_pipelined` parameters are identical between master and slave.

Protocol hierarchy	Response
Critical parameters	mthreadbusy_exact, mthreadbusy_pipelined
Reference	Section 4.9.5 on page 64

Rule 2.6.20 master_slave_cfg_<sdatathreadbusy_param>_match

If the interface includes SDataThreadBusy, the `sdatathreadbusy_exact` and `sdatathreadbusy_pipelined` parameters are identical between master and slave.

Protocol hierarchy	Response
Critical parameters	sdatathreadbusy_exact, sdatathreadbusy_pipelined
Reference	Section 4.9.5 on page 64

Rule 2.6.21 master_slave_cfg_tag_interleave_size_match

If `tags` > 1, the master's `tag_interleave_size` is smaller than or equal to the slave's `tag_interleave_size`.

Protocol hierarchy	Request
Critical parameters	tag_interleave_size
Reference	Section 4.9.5 on page 64

Rule 2.6.22 master_slave_cfg_datahandshake_match

Configuration of the `datahandshake` parameter is identical between master and slave.

Protocol hierarchy	Request
Critical parameters	datahandshake
Reference	Section 4.9.5 on page 64

Rule 2.6.23 master_slave_cfg_writeresp_enable_onewaymatch

Configuration of the `writeresp_enable` parameter is identical between master and slave. If master has `writeresp_enable`=0 then slave must be configured with `writeresp_enable`=0. If master has `writeresp_enable`=1 and slave is configured with `writeresp_enable`=0 then both `write_enable`=0 and `broadcast_enable` = 0 (i.e., WR and BCST must not be enabled).

Protocol hierarchy	Request
Critical parameters	writeresp_enable, write_enable, broadcast_enable
Reference	Section 4.9.5 on page 64

Rule 2.6.24 master_slave_cfg_reqdata_together_match

Configuration of the `reqdata_together` parameter is identical between master and slave.

Protocol hierarchy	Request
Critical parameters	reqdata_together
Reference	Section 4.9.5 on page 64

Rule 2.6.25 master_slave_cfg_mreset_match

If the master has `mreset` enabled to 1, the slave has `mreset` enabled to 1.

Protocol hierarchy	Sideband
Critical parameters	mreset
Reference	Section 4.9.5 on page 64

Rule 2.6.26 master_slave_cfg_sreset_match

If the slave has `sreset` enabled to 1, the master has `sreset` enabled to 1.

Protocol hierarchy	Sideband
Critical parameters	sreset
Reference	Section 4.9.5 on page 64

Rule 2.6.27 master_slave_cfg_tags_match

The master and slave `tags` values must match.

Protocol hierarchy	Request
Critical parameters	tags
Reference	Section 4.9.5 on page 64

Rule 2.6.28 master_slave_cfg_interoperability

If either master or slave have connection parameter set to 0, then ConnectCap for master and slave that have this parameter to 1 must be tied off to 0.

Protocol hierarchy	sideband
Critical parameters	connection
Reference	Section 3.2.1 on page 26

Rule 2.6.29 master_slave_cfg_connectcap_match

Configuration of the ConnectCap signal value is identical between master and slave, if both master and slave have connection set to 1.

Protocol hierarchy	sideband
Critical parameters	connection
Reference	Section 3.2.1 on page 26

20 *Functional Coverage*

The functional coverage approach described in this chapter is bottom-up, meaning the analysis starts at the signal level and goes up to the transaction level. The transfer level has been skipped for reasons highlighted in Section 20.2 on page 447. Along this path several coverage types are used. The signal level uses toggle, state, and meta coverages, while the transaction level uses cross and meta coverages.

Toggle coverage

Toggle coverage provides baseline information that a system is connected properly, and that higher level coverage or compliance failures are not simply the result of connectivity issues. Toggle coverage answers the question: Did a bit change from a value of 0 to 1 and back from 1 to 0? This type of coverage does not indicate that every value of a multi-bit vector was seen but measures that all the individual bits of a multi-bit vector did toggle. In certain cases, not all bits can toggle. A system that only supports RD commands, (“010”) for example, will only need toggle coverage on MCmd bit1. MCmd bit0 and bit2 will always be 0. Therefore they must be filtered from the MCmd toggle coverage.

State coverage

State coverage applies to signals that are a minimum of two bits wide. In most cases, the states (also commonly referred to as coverage bins) can be easily identified as all possible combinations of the signal. For example, for the SResp signal, the states could be 00 (IDLE), 01 (DVA), 10 (FAIL) and 11 (ERR). If the state space is too large, an intelligent classification of the states must be made. In the case of the MAddr signal for example, a possible choice of the coverage bins could be one bin to cover the lower address range, one bin to cover the upper address range and one bin to cover all other intermediary addresses.

Meta coverage

Meta coverage is collecting second-order coverage data. Possible meta coverage measurements include accept backpressure delays, threadbusy backpressure delays and inter-phase delays. Meta coverage information is particularly useful to flag excessive latencies (possibly indicating dead-locks) and to evaluate the OCP backpressure mechanisms (accept / threadbusy).

Cross coverage
Cross coverage measures the activity of one or multiple categories. A category is defined at the transaction level that typically groups multiple OCP signals to form a more abstract, higher-level view of a particular aspect of the OCP protocol. The most pertinent category example is the transTypes. This category combines the MCmd, MBurstLength and MBurstSingleReq signals into a higher-level category. Cross coverage on one category, for example the transTypes category, indicates which kind of transactions were applied to the system under test (for instance, MRMD-RD-4, SINGLE-WRNP, etc.). Cross coverage on multiple categories, for example the transTypes and transResults categories, not only provides information about the transactions applied to the system, but also on their results. In essence, cross coverage measures the types of transactions passing through a system.

20.1 Signal Level

Table 86 summarizes the OCP functional coverage approach for the signal level. The table maps all OCP signals (non-sideband) into phase groups (req / datahs / resp) and provides coverage information in the two outermost right columns. Each coverage type field is colored either in green or in yellow. Green fields are mandatory for functional coverage. Yellow fields are optional for functional coverage.

Level 1—Baseline Coverage
Level 1—baseline column establishes a solid baseline for the signal level functional coverage, so it contains only mandatory coverage. The coverage type is toggle coverage. Toggle coverage provides a minimum level of confidence to the verification engineer that the device under test is alive and properly connected to the rest of the system. It proves as well that no OCP signals are stuck at 0 or 1. In some cases, filters should be applied to the toggle coverage to exclude coverage of bits that can never toggle (refer to the MCmd example on page 443).

Level 2 Coverage
The level 2 coverage type column defines additional coverage. Possible coverage types are state or meta. State coverage defines states (bins) for a multi-bit vector to provide a higher level of abstraction. Meta coverage covers accept / threadbusy backpressure delays.

Mandatory fields must be covered and are with their respective coverage type:

MAddr/MCmd/SResp	:state coverage
MAddrSpace/MByteEn/MDataByteEn	:state coverage
MAtomicLength/MBurstLength/MBurstSeq	:state coverage
MBlockHeight/MBlockStride	:state coverage

MTagID/MDataTagID/STagID	:state coverage
MThreadID/MDataThreadID/SthreadID	:state coverage

Optional column level 2 fields and their coverage types are:

MData/SData	:state coverage
MDataValid	:meta coverage
SCmdAccept/SDataAccept/MRespAccept	:meta coverage
MReqInfo/MDataInfo/SDataInfo/SRespInfo	:state coverage
MConnID	:state coverage
SThreadBusy/SDataThreadBusy/MThreadBusy	:meta coverage

Signals that are only one bit wide only have toggle coverage:

MBurstPrecise/MBurstSingleReq

MReqLast/MDataLast/SRespLast

MTagInOrder/STagInOrder

MReqRowLast/MDataRowLast/SRespRowLast

Table 86 Signal Level Functional Coverage

		Phase Groups			**Coverage Type**	
Signal Group	**Signal**	**Req**	**Datahs**	**Resp**	**Level 1 Baseline**	**Level 2**
Basic	MAddr	X			Toggle	State
	MCmd	X			Toggle	State
	MData	X			Toggle	State
	MDataValid		X		Toggle	Meta
	MRespAccept			X	Toggle	Meta
	SCmdAccept	X			Toggle	Meta
	SData			X	Toggle	State
	SDataAccept		X		Toggle	Meta
	SResp			X	Toggle	State

		Phase Groups			**Coverage Type**	
Signal Group	**Signal**	**Req**	**Datahs**	**Resp**	**Level 1 Baseline**	**Level 2**
Simple	MAddrSpace	X			Toggle	State
	MByteEn	X			Toggle	State
	MDataByteEn		X		Toggle	State
	MDataInfo		X		Toggle	State
	MReqInfo	X			Toggle	State
	SDataInfo			X	Toggle	State
	SRespInfo			X	Toggle	State
Burst	MAtomicLength	X			Toggle	State
	MBlockStride	X			Toggle	State
	MBlockHeight	X			Toggle	State
	MBurstLength	X			Toggle	State
	MBurstPrecise	X			Toggle	
	MBurstSeq	X			Toggle	State
	MBurstSingleReq	X			Toggle	
	MDataLast		X		Toggle	
	MDataRowLast		X		Toggle	
	MReqLast	X			Toggle	
	MReqRowLast	X			Toggle	
	SRespLast			X	Toggle	
	SRespRowLast			X	Toggle	
Tag	MDataTagID		X		Toggle	State
	MTagID	X			Toggle	State
	MTagInOrder	X			Toggle	
	STagID			X	Toggle	State
	STagInOrder			X	Toggle	
Thread	MConnID	X			Toggle	State
	MDataThreadID		X		Toggle	State
	MThreadBusy			X	Toggle	Meta
	MThreadID	X			Toggle	State
	SDataThreadBusy		X		Toggle	Meta
	SThreadBusy	X			Toggle	Meta
	SThreadID			X	Toggle	State

		Phase Groups			Coverage Type	
Signal Group	**Signal**	**Req**	**Datahs**	**Resp**	**Level 1 Baseline**	**Level 2**
Sideband	MConnect				Toggle	State
	SConnect				Toggle	
	SWait				Toggle	

Table 87 outlines options for signal level meta coverage. For each phase group (req/datahs/resp), two meta coverage types are identified: accept backpressure delay and threadbusy backpressure delay. Other meta coverage types could be identified.

Table 87 Signal Level Meta Coverage Examples

Phase Group	Coverage Types	Details
Request phase	Accept backpressure delay	MCmd – SCmdAccept delay
	Thread busy backpressure delay	SThreadBusy backpressure delay (per bit)
Datahs phase	Accept backpressure delay	MDataValid – SDataAccept delay
	Thread busy backpressure delay	SDataThreadBusy backpressure delay (per bit)
Response phase	Accept backpressure delay	SResp – MRespAccept delay
	Thread busy backpressure delay	MThreadBusy backpressure delay (per bit)

20.2 Transfer Level

The transfer level for functional coverage is being skipped. The underlying reasons are:

- The most obvious order for the OCP functional coverage definition is to follow the OCP hierarchy: signal, phase, transfer, transaction. However, such reasoning does not work well for SRMD bursts. SRMD bursts can be constructed as 1 req + n datahs + 1 resp. As such, the transfer concept does not apply 100% because the number of phases per transfer is not constant. Since it is desirable to have a uniform functional coverage definition, which applies to all OCP transactions (MRMD, SRMD, or SINGLE), it makes sense to skip the transfer level.
- Even if the transfer level was included, there are no valuable coverage points. The combination of phases into transfers is a pure protocol check related matter. Meta coverage to measure inter-phase delays may be useful and is discussed at the transaction level.

20.3 Transaction Level

transTypes Concept

Before discussing coverage at the transaction level, clarification is required concerning the process of getting from the signal level to the transaction level. In essence, signals are combined into phases that are then combined into transactions. The unique transaction types represented in this table are referred to as *transTypes*. Table 88 below summarizes this process and is based on the phases in a transfer described in Section 4.3.2.1 on page 42.

Table 88 transTypes

		Phases			**Enabling Condition**	
MBurstSingleReq	**MCmd**	**Req**	**Datahs**	**Resp**	**Datahandshake**	**Writeresp_enable**
0 MRMD SINGLE transfer	RD/RDEX/RDL	H*L		H*L		
	WR/BCST	H*L			0	0
	WR/BCST	H*L		H*L	0	1
	WRNP/WRC	H*L		H*L	0	don't care
	WR/BCST	H*L	H*L		1	0
	WR/BCST	H*L	H*L	H*L	1	1
	WRNP/WRC	H*L	H*L	H*L	1	don't care
1 SRMD SINGLE transfer	RD	1		H*L		
	WR/BCST	1	H*L		1	0
	WR/BCST	1	H*L	1	1	1
	WRNP	1	H*L	1	1	don't care

Notes to Table 88:

1. The table shows how phases are combined into SINGLE transfers, MRMD bursts and SRMD bursts. SINGLE transfers can be de-generated from either MRMD or SRMD bursts.
2. L stands for the MBlockLength (one in the case of a SINGLE transfer) and H stands for MBlockHeight.
3. transTypes are controlled by the signals MBurstSingleReq, MCmd and MBlockHeight(H), MBurstLength (L) and the parameters `datahandshake` and `writeresp_enable`.
4. RDEX, RDL, and WRC commands only apply to SINGLE transfers.

5. RD, RDEX, and RDL are not controlled by the `datahandshake` and `writeresp_enable` parameters.

6. WRNP and WRC are not controlled by the `writeresp_enable` parameter.

7. The possible transTypes are:

   - SINGLE transfers RD, WR, BCST, WRNP, RDEX, RDL, and WRC
   - MRMD bursts RD, WR, BCST, and WRNP
   - SRMD bursts RD, WR, BCST, snf WRNP

   The following example illustrates this.

   If MBurstSingleReq supports values 0 and 1 and
   If MCmd only supports RD,WR,WRNP,RDEX and
   If datahandshake == 1 and writeresp_enable == 1
   then the transTypes will be:
   a) SINGLE transfers, RD/WR/WRNP/RDEX
   b) MRMD bursts, RD/WR/WRNP
   c) SRMD bursts, RD/WR/WRNP

Category Concept

A category groups one or more OCP signals and serves as a building block for cross coverage. A category also represents a higher level view of the OCP protocol, allowing intelligent crosses to be made of one or more categories. Table 89 lists and describes the proposed categories:

Table 89 Categories

Name	**Description**
transTypes	Category containing the transaction types based on Table 88
transTargets	Category containing the transaction targets (MAddr / MAddrSpace)
transResults	Category containing the transaction results (SResp)
transBurstProps	Category containing the burst properties (AtomicLength / MBurstPrecise / MBurstSeq)
transByteens	Category containing the transaction byte enables (MByteEn / MDataByteEn)
flowThreads	Category containing the flows as function of the ThreadID
flowTagTypes	Category containing the flows as function of the TagInOrder / TagID

Notes to Table 89:

1. The transBurstProps category may be split into multiple categories enabling a higher granularity for cross coverage.

2. The flowTagTypes category combines the TagInOrder and TagID signals. The tag type is encoded as follows:

- If TagInOrder, then tag type == tag-in-order (1 enumerated value)
- If not, then tag type == the TagID (multiple enumerated values)

If the TagID range is too large, sub-ranges should be defined. As such, the tag type will have 1 + x enumerated values.

20.4 Mapping Signals into Categories

Table 90 shows the OCP signals (non-sideband) mapped into the categories described in the previous section. Only signals that are not optional in the level 2 column of Table 86, and are not MReqLast, MDataLast, or SRespLast, are mapped.

Table 90 Signal Mapping into Categories

Signal	**Categories**						
	transTypes	**transTargets**	**transResults**	**transBurstProps**	**transByteens**	**Flow Threads**	**flowTagTypes**
MAddr		X					
MAddrSpace		X					
MAtomicLength				X			
MBlockHeight	X						
MBurstLength	X						
MBurstPrecise				X			
MBurstSeq				X			
MBurstSingleReq	X						
MByteEn					X		
MCmd	X						
MDataByteEn					X		
MDataTagID							X
MDataThreadID						X	
MTagID							X
MTagInOrder							X
MThreadID						X	
SResp			X				

Signal	Categories						
	transTypes	transTargets	transResults	transBurstProps	transByteens	Flow Threads	flowTagTypes
STagID							X
STagInOrder							X
SThreadID						X	

20.4.1 Cross Coverage of One Category

Cross coverage can be applied to just one category. Since this kind of cross coverage only makes sense if a category contains more then one signal, the transResults and transByteens categories are excluded from this type of cross coverage.

Cross coverage of one category can be useful in measuring what kind of transTypes flowed through a design regardless of the signals contained in other categories (for example, the transResults). A more useful coverage result from applying crosses among several categories. Cross coverage of one category is considered optional while cross coverage on multiple categories is considered mandatory.

20.4.2 Cross Coverage on Multiple Categories

Cross coverage can also be applied to combinations of categories. Theoretically, many crosses are possible (128 in total), but only some will make sense for a specific OCP interface configuration and design architecture.

The crosses between the transTypes category and other categories are considered mandatory and establish a solid base for cross coverage at the transaction level. Table 91 shows some of the mandatory crosses that form a sub-set of the theoretical possibilities. It is up to the user to declare additional crosses that exclude the transTypes category, but are important for the system under test. Such crosses are considered optional.

Table 91 Mandatory Crosses of Multiple Categories (Including transType)

Categories							
transTypes	**transTargets**	**transResults**	**transBurstProps**	**transByteens**	**Flow Threads**	**flowTagTypes**	**Cross Description**
X	X						Cross all transaction types with all targets
X					X		Cross all transaction types for all threads
X		X					Cross all transaction types with all transaction results
X			X	X			Cross all transaction types for all bursts with all byte enable patterns
X		X	X				Cross all transaction types for all bursts with the transaction results

20.5 Meta Coverage

Table 92 outlines possibilities for the transaction level meta coverage. Three meta coverage types are identified: accept backpressure delays relative to the position in a transaction, threadbusy backpressure delays relative to the position in a transaction and several inter-phase delays. Other interesting meta coverage types could be identified.

Table 92 Transaction Level Meta Coverage Examples

Meta Coverage Types	**Coverage Details**
Inter-phase delays	req to req / datahs to datahs / resp to resp delays
	req to datahs / req to resp / datahs to resp delays
	First req accepted delay / last req accepted delay for MRMD bursts
Accept backpressure delays relative to the position in the transaction	Measure when accept backpressure occurs in a transaction
Threadbusy backpressure delays relative to the position in the transaction	Measure when threadbusy backpressure occurs in a transaction

20.6 Sideband Signals Coverage

Toggle coverage

Toggle coverage must be applied to each individual bit of the sideband signals to establish a solid coverage baseline.

State coverage

Sideband signals that consist of multiple bits can have state coverage similar to the dataflow signals.

Meta coverage

Meta coverage can be added for the control and status signals. Some examples of meta coverage that might be added for the control signals handshake are:

- The delay between two ControlWr signal assertions.
- The length of the ControlBusy signal assertion.
- The ControlBusy assertion relative to the previous ControlWr assertion.

Some examples of meta coverage that might be added for the status signals handshake are:

- The delay between two StatusRd signal assertions.
- The length of the StatusBusy signal assertion.

20.7 Naming Conventions

This section describes he naming conventions for functional coverage.

Signal Level (Dataflow Signals)

Naming template:

signal_<coverage type>_<signal name | meta name>_<bin>

In which:

<coverage type>:	toggle
	state
	meta
<signal name>:	OCP signal
<meta name>:	SCmdAcceptDelay
	SDataAcceptDelay
	MRespAcceptDelay
	SThreadBusyDelay
	SDataThreadBusyDelay
	MThreadBusyDelay
<bin>:	if enumerated types are defined in OCP use them
	for example: SResp in [ERR,DVA,FAIL,IDLE]
	else be free to choose a clear name

Examples:

signal_toggle_MAddr_bit0_0to1
signal_state_MByteEn_allOnes
signal_meta_SThreadBusyDelay_2

Transaction Level (Dataflow Signals)

Naming template:

trans_<coverage type>_<cross name | meta name>[_<bin>]

In which:

```
<coverage type>:cross
               meta
<cross name>:for cross coverage of 1 category:
                      transTypes
                      transTargets
                      transBurstProps
                      flowThreads
                      flowTagTypes
               for cross coverage of multiple categories:
                      trans_<list of ...>_flow_<list of ...>
                      TypesThreads
                      ResultsTagTypes
                      Targets
                      BurstProps
                      Byteens
<meta name>:ScmdAcceptDelay
               SDataAcceptDelay
               MRespAcceptDelay
               SThreadBusyDelay
               SDataThreadBusyDelay
               MThreadBusyDelay
               ReqReqPhaseDelay
               ReqRespPhaseDelay
     ...
[<bin>]:       The bin naming is optional for cross coverage.
               In most cases the bins will automatically be chosen by
               the verification tool itself. However if the cross includes
               signals which have specific OCP enumerated values defined
               (as DVA for SResp), it's advisable to use them.
```

Examples:

```
trans_cross_transTypes
trans_cross_trans_TypesResults
trans_cross_flow_ThreadsTagTypes
trans_cross_trans_TypesResults_flow_Threads
trans_meta_ReqReqPhaseDelay_4
```

Sideband Signals

Naming template:

sideband_<coverage type>_<signal name | meta name>_<bin>

In which:

<coverage type>:	toggle
	state
	meta
<signal name>:	OCP signal
<meta name>:	ControlWrControlWrDelay
	ControlBusyDuration
	ControlWrControlBusyDelay
	StatusRdStatusRdDelay
	StatusRdDuration
<bin>:	be free to choose a clear name

Examples:

```
sideband_toggle_ControlWr
sideband_state_Control_3
sideband_meta_StatusRdDuration_5
```

A OCP Trace File

The OCP trace file consists of data recorded by an SVA monitor during simulation. The name of the file generated by the OCP monitor is `<ocpName>.ocp`. Because of the variable configuration of OCP connections, SVA trace files may appear to have different formats. However, two OCP connections with identical configurations generate trace files with identical formats. The SVA trace file consists of header and trace data sections, as described below.

A.1 Header

The header section defines the parameters of the OCP connection from which the trace data originated. Example 2 shows a sample OCP trace file.

Example 2 Sample OCP Trace File

```
# ocpversion=ocp2.2-1.6
# name=ocp20_ocpmon2
# mreset=1
# addr_wdth=32
# data_wdth=64
##
   10.0 0 0 xxxxxxxx x xxxxxxxxxxxxxxxx 0 xxxxxxxxxxxxxxxx
  110.0 1 0 xxxxxxxx x xxxxxxxxxxxxxxxx 0 xxxxxxxxxxxxxxxx
  120.0
  210.0 1 1 00000000 1 0000000087654321 0 xxxxxxxxxxxxxxxx
  220.0
  230.0
  370.0 1 0 xxxxxxxx x xxxxxxxxxxxxxxxx 1 0000000087654321
  380.0 1 0 xxxxxxxx x xxxxxxxxxxxxxxxx 0 xxxxxxxxxxxxxxxx
...
```

Parameters in the header section show the parameter name and the assigned value. The `ocpversion` is derived from the install tree and provides the OCPIP revision number and release number. For example, `ocpversion=ocp2.2-1.6`.

The `name` parameter identifies the OCP monitor from which the trace data was recorded. For example, `name=ocp20_ocpmon2` indicates the OCP monitor name was `ocp20_ocpmon2`.

If a parameter is not specified in the header, the default value listed in Table 29 on page 68 is used. If signals are enabled, width parameters are required for the corresponding signals, but width parameters do not have explicit defaults. Typically, trace files generated by monitors specify all parameter values.

Since `mreset` and `sreset` do not have defaults, values must be specified for each.

The header of the trace file always ends with a double pound sign (`##`). Optional comment lines preceded by a pound sign may appear following the header and contain information about the program that generated the file.

A.2 Trace Data

Each line of trace data represents the values of the OCP signals for a cycle of data. The trace data is organized so that data remains in fixed fields with a blank space between them. The first field is the simulation time during which the sample was recorded. Some lines of data only have an entry for the simulation time. This means that the OCP signals have not changed value in that cycle. The first line of trace data must have more data than just the entry for the simulation time; that is, there is no default signal state. If the simulation time is not the only field that has an entry then all fields must have an entry. Each entry must have enough data to fill all the bits of the signal for that entry. If there is more data than there are bits, the extra most significant bits will be truncated.

Each line in the data section maps to a snapshot of the OCP connection at the rising edge of the simulation clock.

Table 93 describes each of the fields that exist in the trace data. Following the field descriptions down this table is equivalent to following trace data columns from left to right. The table indicates the required condition for the field to appear in the trace file (specifically, if the required OCP parameter condition is not met, the field will not be present in the trace data.). The table also indicates how many bits of data are required by the field and the format of the field.

For the hexadecimal format, it is possible to have value with Xs and Zs intermixed with 0s and 1s. Such a value would have brackets, {}, around 4 digits to represent a binary encoding for a byte. For example, a 12-bit binary number of 10001X011010 would be represented as 8{1X01}A. When all four bits of a byte are X, a simple X represents the entire byte. When all four bits of a byte are Z, a simple Z represents the entire byte.

Table 93 OCP Trace File, Line Field Decoding

Field	**Parameter Condition**	**Field Width in Bits**	**Format**
Simulation Time	None	(not applicable)	floating point
MReset_n	mreset parameter is 1	Always 1	hexadecimal
SReset_n	sreset parameter is	Always 1	hexadecimal
MCmd	None	Always 3	hexadecimal
MAddr	addr is 1	addr_wdth	hexadecimal
MAddrSpace	addrspace is 1	addrspace_wdth	hexadecimal
MByteEn	byteen	data_wdth / 8	hexadecimal
MConnID	connid is 1	connid_wdth	hexadecimal
MReqInfo	reqinfo is 1	reqinfo_wdth	hexadecimal
MThreadID	threads > 1	threadid_wdth[1]	hexadecimal
MTagID	tags > 0	tagid_wdth[2]	hexadecimal
MTagInOrder	taginorder is 1	Always 1	hexadecimal
MAtomicLength	atomiclength is 1	atomiclength_wdth	hexadecimal
MBurstLength	burstlength is 1	burstlength_wdth	hexadecimal
MBlockHeight[3]	blockheight is 1	blockheight_wdth	hexadecimal
MBlockStride[3]	blockstride is 1	blockstride_wdth	hexadecimal
MBurstPrecise	burstprecise is 1	Always 1	hexadecimal
MBurstSeq	burstseq is 1	Always 3	hexadecimal
MBurstSingleReq	burstsinglereq is 1	Always 1	hexadecimal
MReqLast	reqlast is 1	Always 1	hexadecimal
MReqRowLast [3]	reqrowlast is 1	Always 1	hexadecimal
SCmdAccept	cmdaccept is 1	Always 1	hexadecimal
SThreadBusy	sthreadbusy is 1	threads	hexadecimal
MData	mdata is 1	data_wdth	hexadecimal
MDataInfo	mdatainfo is 1	mdatainfo_wdth	hexadecimal
MDataValid	datahandshake is 1	Always 1	hexadecimal
MDataByteEn	mdatabyteen is 1	data_wdth / 8	hexadecimal
MDataThreadID	threads > 1 and datahandshake is 1	threadid_wdth[1]	hexadecimal
MDataTagID	tags > 1 and datahandshake is 1	tagid_wdth[2]	hexadecimal
MDataLast	datalast is 1	Always 1	hexadecimal
MDataRowLast [3]	datarowlast is 1	Always 1	hexadecimal

Field	**Parameter Condition**	**Field Width in Bits**	**Format**
SDataAccept	dataaccept is 1	Always 1	hexadecimal
SDataThreadBusy	sdatathreadbusy is 1	threads	hexadecimal
SResp	resp is 1	Always 2	hexadecimal
SRespInfo	respinfo is 1	respinfo_wdth	hexadecimal
SThreadID	threads > 1 and resp is 1	threadid_wdth [1]	hexadecimal
STagID	tags > 1 and resp is 1	tagid_wdth [2]	hexadecimal
SData	sdata is 1	data_wdth	hexadecimal
SDataInfo	sdatainfo is 1	sdatainfo_wdth	hexadecimal
SRespLast	resplast is 1	Always 1	hexadecimal
SRespRowLast [3]	resprowlast is 1	Always 1	hexadecimal
MRespAccept	respaccept is 1	Always 1	hexadecimal
MThreadBusy	mthreadbusy is 1	threads	hexadecimal
MFlag	mflag is 1	mflag_wdth	binary
MError	merror is 1	Always 1	binary
SFlag	sflag is 1	Always 1	binary
SError	serror is 1	Always 1	binary
SInterrupt	interrupt is 1	Always 1	binary
Control	control is 1	control_wdth	hexadecimal
ControlWr	controlwr is 1	Always 1	binary
ControlBusy	controlbusy is 1	Always 1	binary
Status	status is 1	status_wdth	hexadecimal
StatusRd	statusrd is 1	Always 1	binary
StatusBusy	statusbusy is 1	Always 1	binary

1. The threadid_wdth parameter is internal and calculated as follows:
   `threadid_wdth` = max(1, $\log_2$(`threads`))
2. The tagid_wdth parameter is internally derived, and is calculated as follows:
   `tagid_wdth` = max(1, $\log_2$(`tags`))
3. No signals are associated with *`threadbusy_pipelined` parameters. The existing Thread Signals are used in that case.

Index

OCP-IP Administration
3116 Page Street
Redwood City, CA 94063
Ph: +1 (512) 551.3377
Fax: +1 (650) 365.4658
admin@ocpip.org
www.ocpip.org

OCP International Partnership