OCP Protocol

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.

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.

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

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

Open Core Protocol (OCP) is a standard designed to facilitate the integration and interoperability of various components in system-on-chip (SoC) designs. It provides a framework for communication between different intellectual property (IP) cores, allowing them to work together seamlessly. Here are some key points about OCP:

1. **Purpose**: OCP aims to simplify the design process for complex systems by establishing a standardized method for communication between components, thus reducing integration time and potential compatibility issues.
2. **Protocol Layers**: The OCP consists of various layers, including the physical layer (for signal transmission) and the protocol layer (which defines the rules for data exchange). This modular approach allows for flexibility in implementation.
3. **Applications**: OCP is widely used in the semiconductor industry, particularly in the development of SoCs for applications like mobile devices, consumer electronics, and automotive systems.
4. **Community and Support**: OCP is supported by a community of developers and engineers, providing resources, documentation, and forums for collaboration and troubleshooting.

Overall, Open Core Protocol plays a crucial role in modern chip design, enabling more efficient and flexible system architectures

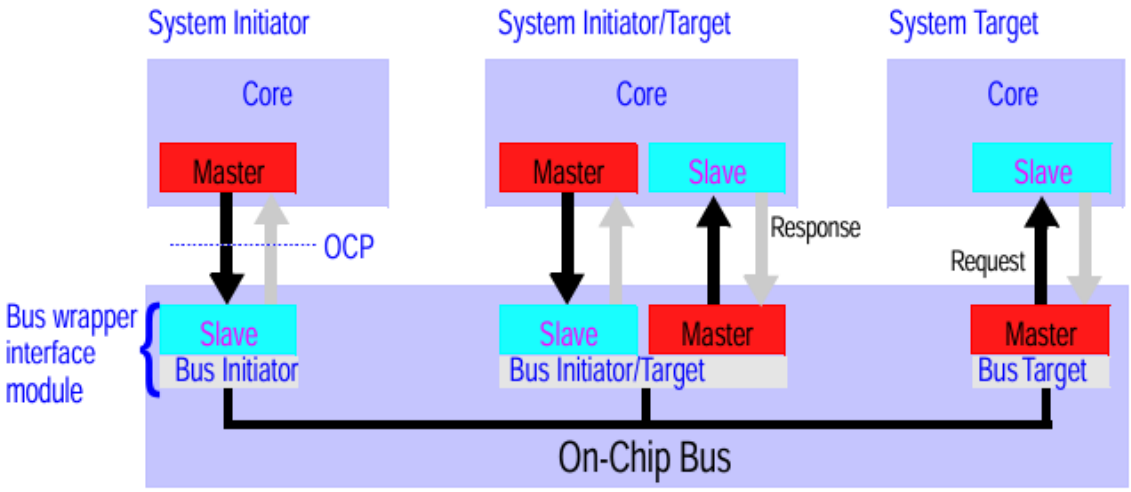
Open Core Protocol (OCP) has several key characteristics that make it an effective standard for communication between IP cores in system-on-chip (SoC) designs:

1. **Modularity**: OCP allows for modular design, enabling different components to be developed independently and integrated seamlessly. This modularity supports easier upgrades and modifications.
2. **Scalability**: The protocol is designed to accommodate a range of system sizes, from small embedded systems to large, complex SoCs, making it versatile for various applications.
3. **High Performance**: OCP supports high-speed data transfers, which is crucial for performance-sensitive applications. It can handle multiple transactions in parallel, enhancing throughput.
4. **Interoperability**: By adhering to a common standard, OCP ensures that components from different vendors can work together without compatibility issues, fostering a more open ecosystem.
5. **Flexibility**: The protocol can be adapted to different bus architectures and technologies, allowing designers to customize implementations to meet specific project requirements.
6. **Protocol Layers**: OCP includes multiple layers, such as the physical layer and the protocol layer, which define the rules for communication and data integrity. This layered approach simplifies the design process.
7. **Quality of Service (QoS)**: OCP supports features that allow for prioritization of data transactions, enabling critical tasks to receive the necessary bandwidth and reducing latency.
8. **Error Handling**: The protocol includes mechanisms for error detection and correction, which helps maintain data integrity and reliability during communication.
9. **Standardized Interfaces**: OCP defines standardized interfaces for different types of transactions (such as read, write, and response), making it easier to design and integrate new IP cores.
10. **Support for Multiple Protocols**: OCP can work alongside other protocols, enabling hybrid designs that take advantage of different communication methods.

These characteristics contribute to OCP's effectiveness in simplifying the design and integration of complex systems, promoting efficiency and reducing time-to-market for new products.

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 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.



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.

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.

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 WriteCon ditional—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 presen tation 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. 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

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. 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: min_addr_wdth = max(1, floor(log2(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.

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.

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.

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

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.

Simple Extensions:

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

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 - configurable configurable slave slave Additional information transferred with the read data Additional information transferred with the response 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 [(8n + 7):8n] 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.

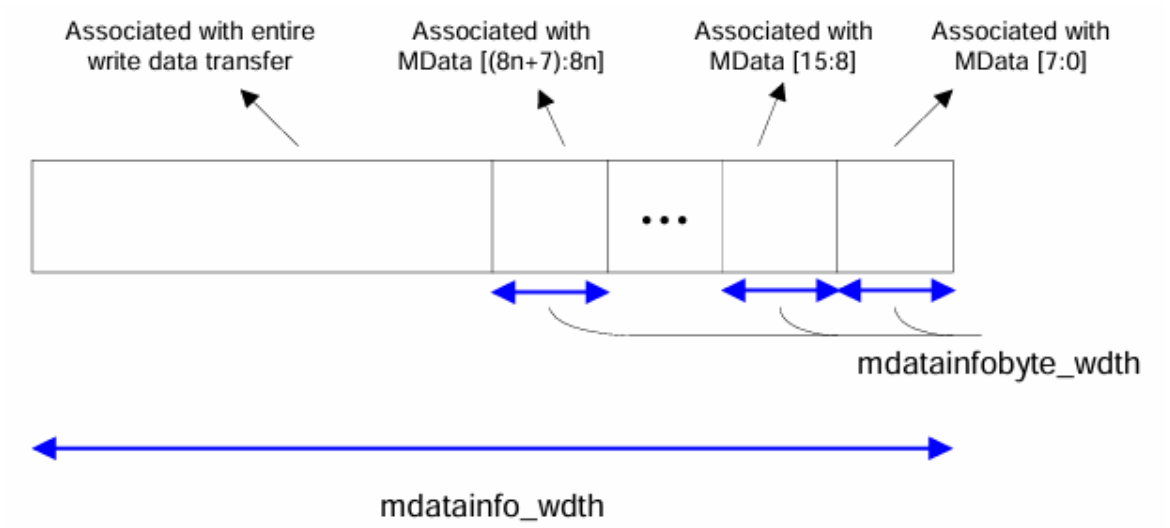
MDataByteEn - Write byte enables. This field indicates which bytes within the OCP word are part of the current write transfer. See Section 4.4.1 on page 50 for more detail on request and datahandshake phase byte enables and their relationship. There is one bit in MDataByteEn for each byte in the OCP word. Setting MDataByteEn[n] to 1 indicates that the byte associated with MData wires [(8n + 7):8n] 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.

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. 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.

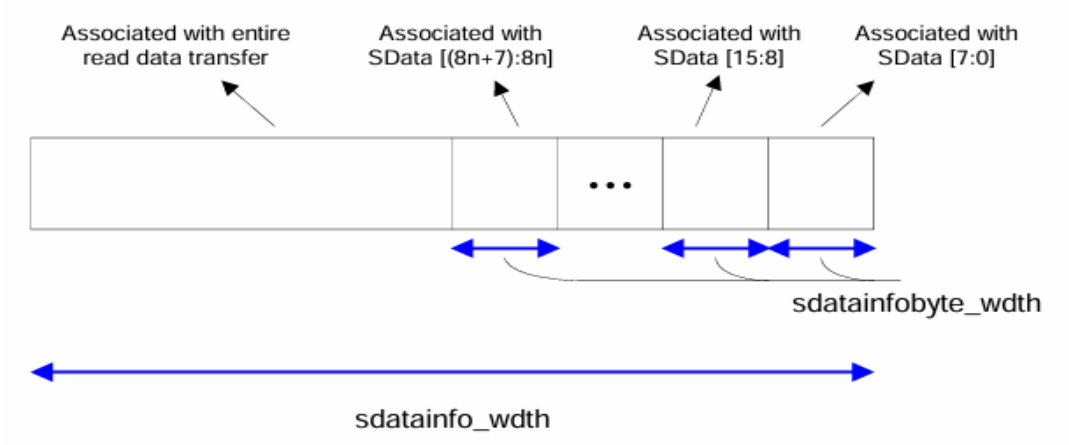
This field is divided in two: the low-order bits are associated with each data byte, while the high-order bits are associated with the entire write data transfer. The number of bits to associate with each data byte is configured using the mdatainfobyte_wdth parameter. The low-order mdatainfobyte_wdth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

MDataInfo Field:



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. 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. SDataInfo Field:



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.

Burst Extensions:

Table 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.

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, 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.

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.

Similarly we have Tag Extensions, Thread Extensions, Sideband Signals, Connection, Resetm Interrupt, Error, and Core-Specific Flag Signals, which includes: MDataTagID, MTagID, MTaginOrder, StagID, StagInOrder, MConnID, MDataThreadID, MThreadBusy, MThreadID, SDataThreadBusy, SThreadBusy, SThreadID, MConnect, MError, MFlag, MReset_n, SConnect, SError, SFlag, SInterrupt, SReset_n, SWait, ConnectCap, Control, ControlBusy, ControlWr, Status, StatusBusy, StatusRd

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

Slave connection vote encoding:

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

Slave connection change delay encoding:

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

Control and Status Signals include Control, ControlBusy, ControlWr, Status, StatusBusy, and StatusRd.

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

Scan Interface includes Scanctrl, Scanin, and Scanout.

Clock Control Interface includes ClkByp, and TestClk.

Debug and Test Interface includes TCK, TDI, TDO, TMS, and TRST_N.

OCP Signal Configuration Parameters:

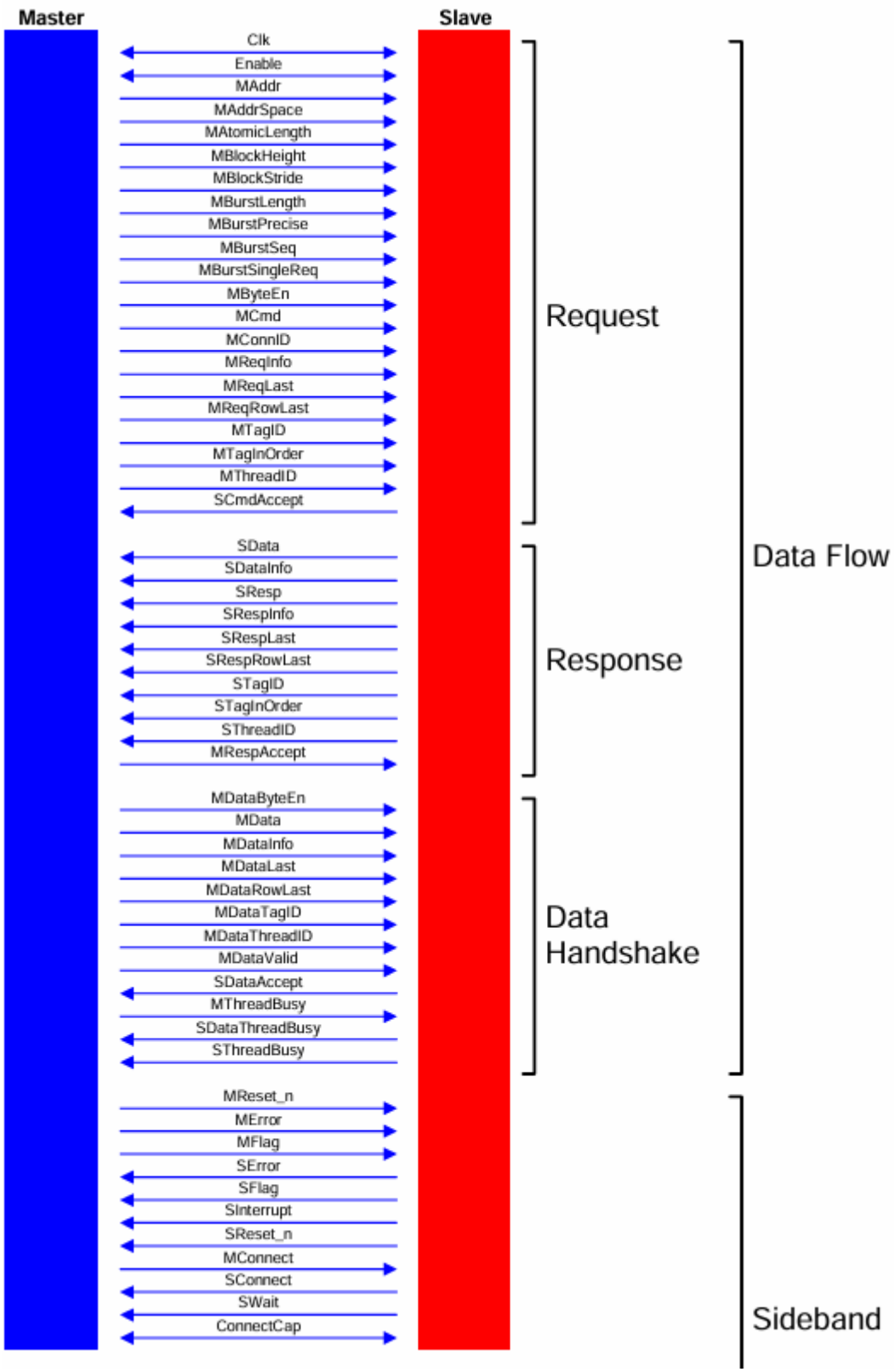
Group	Signal	Parameter to add signal to interface	Parameter to control width	Default Tie-off
Basic	Clk	Required	Fixed	n/a
	EnableClk	enableclk	Fixed	1
	MAddr	addr	addr_wdth	0
	MCmd	Required	Fixed	n/a
	MData	mdata	data_wdth	0
	MDataValid	datahandshake	Fixed	n/a
	MRespAccept[1]	respaccept	Fixed	1
	SCmdAccept	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

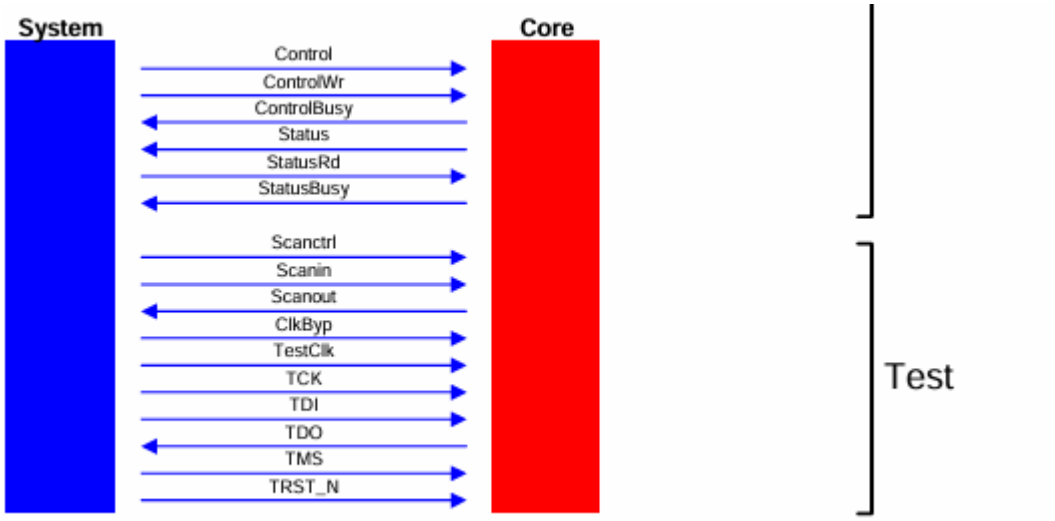
Group	Signal	Parameter to add signal to interface	Parameter to control width	Default Tie-off
Burst	MAtomicLength[7]	atomiclength	atomiclength_wdth	1
	MBlockHeight[7,8]	blockheight	blockheight_wdth[9]	1
	MBlockStride[7,8]	blockstride	blockstride_wdth	0
	MBurstLength	burstlength	burstlength_wdth[10]	1
	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

OCP Signal Summary:



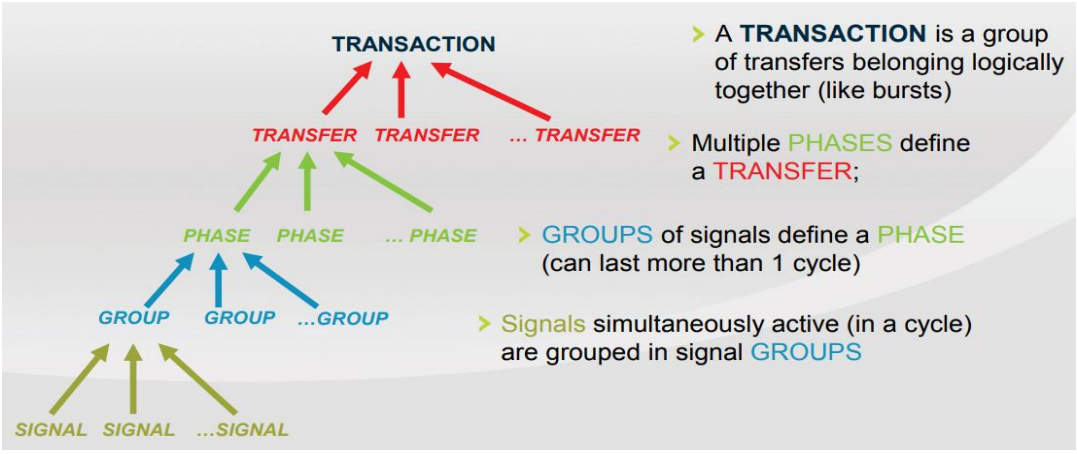


In OCP, a read/write transaction involves a master core sending a master data signal (MData) and a memory address (MAddr) to a slave core to write to the slave core's memory. The slave core then sends a signal back to the master core to acknowledge that the data has been accepted.

Here are some steps involved in a read/write transaction:

- **Master core**: Sends the MData and MAddr to the slave core.
- **Slave core**: Starts its clock, latches the MData and MAddr, and sends an SDataAccept signal back to the master core.
- **Master core**: Receives the SDataAccept signal and acknowledges that the data has been accepted.
- **Slave core**: Writes the data to the memory array and sends an SResp signal back to the master core to acknowledge the successful write transaction.
- **External reset**: Flushes out the data on the address and data buses.

Dataflow Protocol Hierarchy:



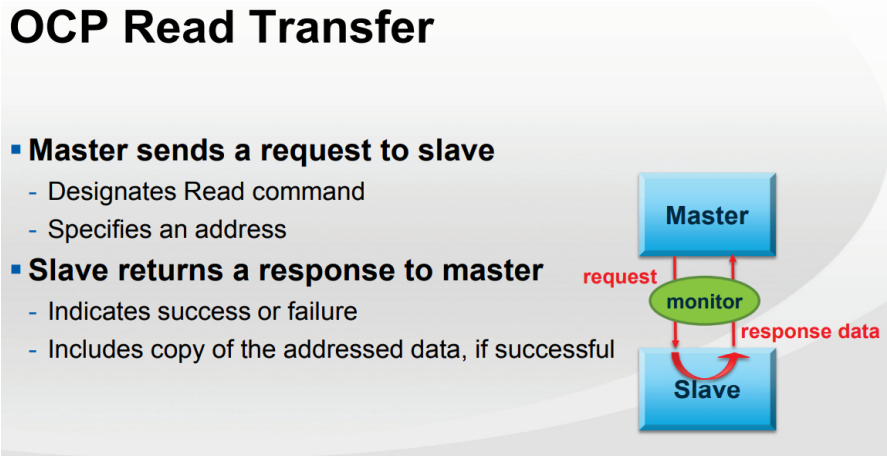
OCP Transaction Types:

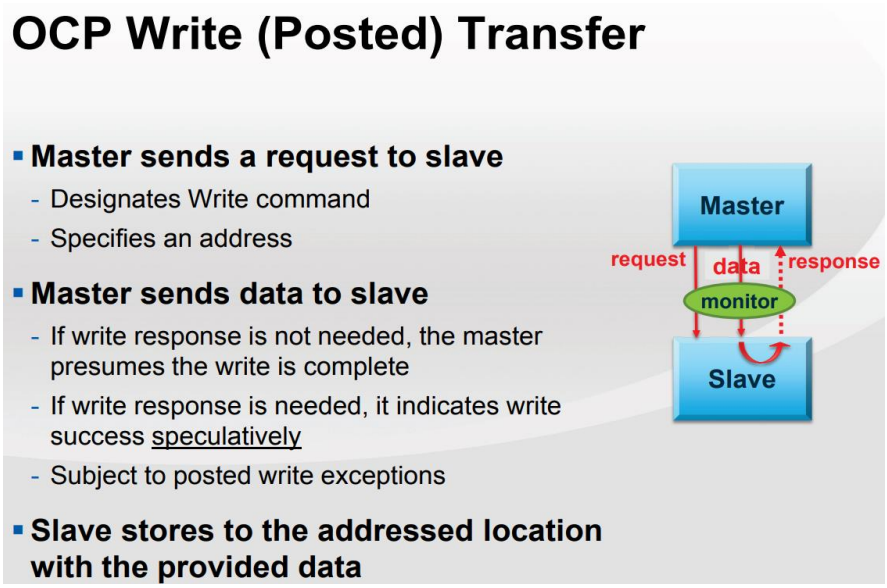
One or more transfers build a transaction

Bursts
- Reads
- Writes
  - Posted (with or without response)
  - Non-posted
- Broadcast

Semaphores
- Read-modify-write (Blocking & Non-blocking)

Configuration specifies explicitly which transactions are allowed
- Read only or Write only
- FIFO-like Write only (push) or Read only (pop) without address





OCP Non-Posted Write Transfer

- **Master sends a request to slave**
  - Designates Non Posted Write command
  - Specifies an address
- **Master sends data to slave**
- **Slave modifies the addressed location with the provided data**
- **Slave must return a response to master**
  - OCP must be configured for write responses
  - Fully synchronized write (response after write complete)

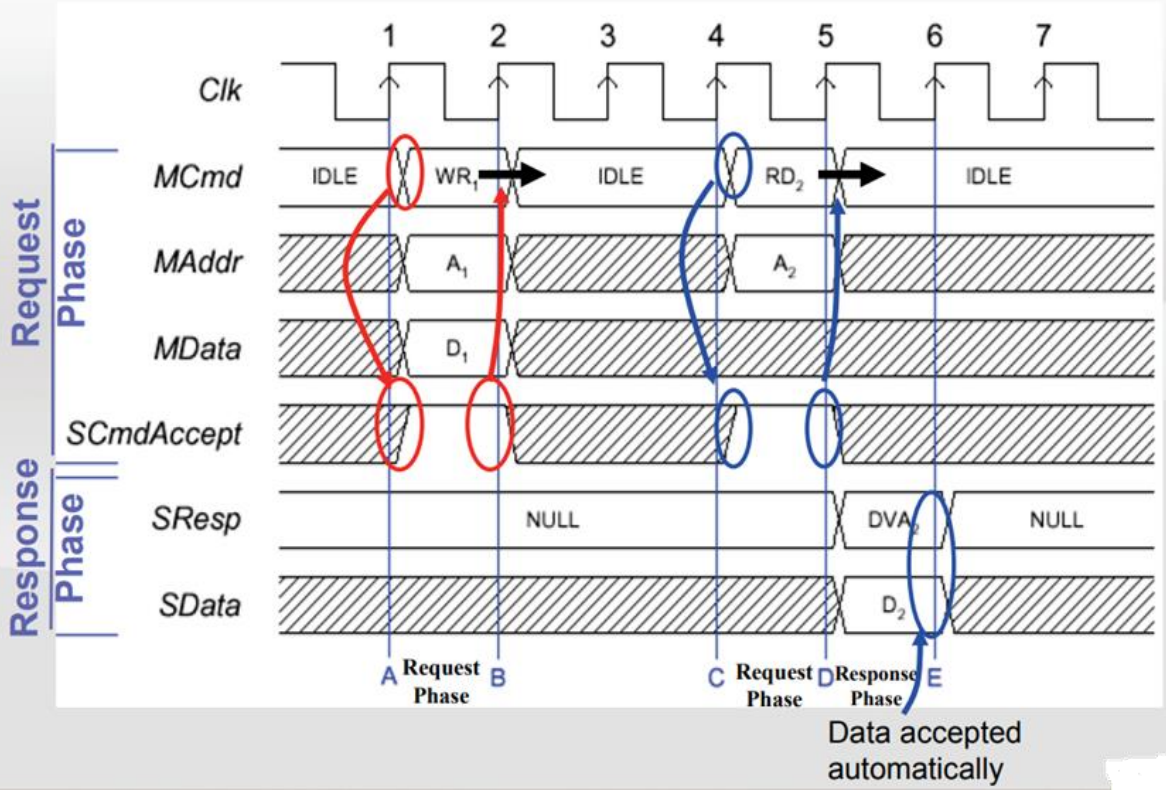
OCP Burst Transaction

- **All reads, writes and broadcasts can be grouped in bursts**
- **Bursts have a length and an address sequence**
- **Bursts of length 1 (one OCP word) are legal**
  - Partial word transfer also possible
- **3 categories of bursts are supported**
  - SRMD – Always precise
  - MRMD – Precise
  - MRMD – Imprecise
- **Burst sequence types**
  - INCR, WRAP, STRM ….

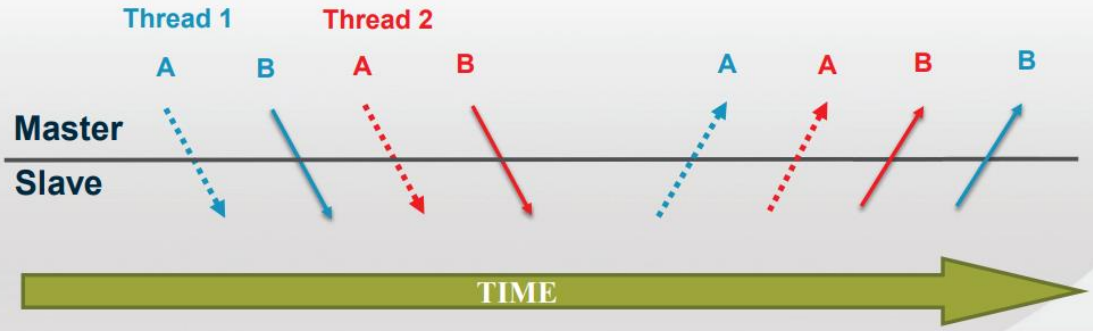
Split Transactions (i.e., Independent Request/Response)

- **OCP Transfers are split into phases**
  - Request phase
  - Data phase
  - Response phase
- **Transfer Pipelining**
  - Determined by the number of “outstanding” transaction both master and slave can support
  - Each transfer proceeds from phase to phase in a prescribed sequence
  - One transfer doesn’t need to complete all its phases before another transfer can start
  - Permits very high performance (bandwidth) socket interface regardless of system latency
  - OCP spec. doesn’t limit the max number of outstanding transactions

Simple Write & Read Transfer



Threads and Ordering



- **Strict ordering *within* a thread**
  - Thread 1: A, B →1A, 1B
  - Thread 2: A, B →2A, 2B
- **No ordering restriction between threads**
  - 1A, 1B, 2A, 2B →1A, 2A, 2B, 1B