MITRE Common Interface for Register Transfer Level Models Using Open Core Protocol Profiles

Revision 2.1

December 2009

Karl T. Wagner

MITRE
Command and Control Center (C2C)
Bedford, Massachusetts

Abstract

This document describes a set of standard interfaces for register transfer level (RTL) blocks developed at MITRE. The interfaces leverage the industry standard Open Core Protocol (OCP) to define the connections between blocks. The interfaces are only intended to describe the connections between blocks, not connections external to a device (e.g. field programmable gate array (FPGA) or application specific integrated circuit (ASIC). The initial focus of these interfaces is on blocks which generate, manipulate or route data samples through the platform. The exact definition of a block is purposely left vague since it will vary with the application. Having a common interface structure simplifies reuse and compatibility of blocks between projects.

Revision History

Revision Number	Date	Description	Sections Affected
1.0	19 Oct 2007	Initial Release	-
1.1	15 Feb 2008	Updated contractual and reference information	-
2.0	20 Feb 2009	Removed obsolete Dataflow Sink Interface	2.2, 3.2, 4, D.2
2.1	10 Dec 2009	Added Appendix E to describe Common PropertySet interface	Appendix E

Table of Contents

List of Figures

List of Tables

1 Introduction

Previous work at MITRE [1] and the industry as a whole has explored the advantages of using a structured design methodology to decompose a complex design into independent parts.

As systems become more complex and logic devices increase in size, higher levels of abstraction are required to efficiently manage logic designs. System on a Chip (SoC) and Platform FPGAs demand a more formalized design strategy. Defining a common interface is a key step toward implementing that strategy. By encapsulating algorithmic code with a common interface, system design can flow independent of algorithm implementation. Individual blocks can be designed to the interface with no regard to platform issues. Such an environment is more conducive to large development teams since each implementer does not need to understand the intricacies of the entire system. A common interface also promotes the use of common test fixtures and fosters the reuse of blocks between projects.

1.1 Purpose of Document

This document describes a set of common interfaces for register transfer level (RTL) blocks. These common interfaces are designed to work with the layered MITRE methodology for designing portable applications [2]. Connections to off-chip resources are not covered in the common interfaces since they will depend on the particular design of the platform. Access to these resources can be provided by wrapping the platform specific external connections with a gasket which translates them to the common interface. Figure 1-1 shows an example platform and some of the connections where the common interfaces are applied.

Where practical, all access to a block should be through the common interfaces; in general this excludes blocks which are closely tied to the platform. The scope of a block is left open to allow a balance between block flexibility and resource overhead. The constraints of individual projects will direct how much functionality is encompassed by each block. Designers should keep in mind the overall goals of the common interfaces when allocating blocks and choosing interfaces.

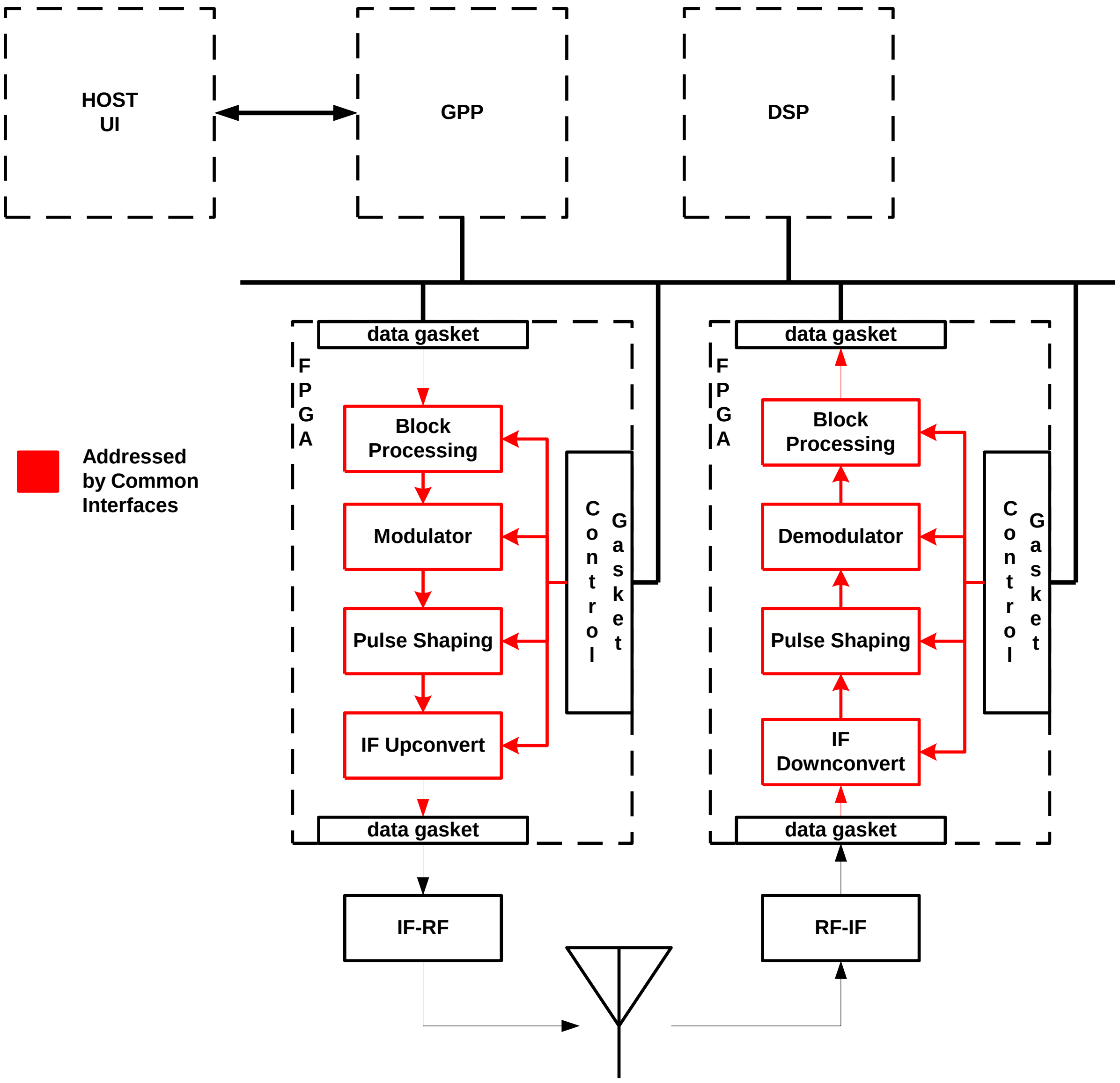

Figure 1-1: Example Connections Addressed By Common Interfaces

1.2 Leveraging Standards

Rather than arbitrarily define signals used by the interfaces, the Open Core Protocol (OCP) [3] is leveraged. OCP is an industry standard for describing the interface between blocks. It specifies a comprehensive, core-centric socket with a small set of basic operations and a large set of optional enhanced features. It also includes rules for some default mappings to seamlessly bridge between interfaces with different configurations. Each OCP interface is configured as either a master or a slave and all connections are point-to-point. However, the number of interfaces provided by a block is specific to the function of that block. Thus a routing block may have many separate master and slave interfaces while a simple translation block might have only one of each.

1.3 Profiles

To ensure interface compatibility while limiting the interface overhead, specific subsets of the OCP, referred to as a profiles, are used for the MITRE common interfaces. This document outlines the chosen profiles and describes common use cases. Specific information on the behavior of the signals can be found in the OCP specification [3]; in particular, chapters 3,4,8 and 9 contain the key information regarding signaling protocols. Since use of the OCP is limited to specific profiles, it is not necessary to understand all valid configurations of the OCP. The designer can concentrate on the specific signals outlined in the following sections.

Several distinct profiles are defined based on the expected interaction between blocks. Dataflow is a high speed streaming profile intended for high performance signal paths. It sacrifices flexibility to reduce overhead which reduces the size, weight and power (SWaP) requirements of the final platform. Data is organized as a sequential stream in a single direction. Bidirectional streaming data can be handled with the dataflow profile by using two complimentary interfaces. The more complex memory profile adds addressing and bidirectional flow. This profile covers both high speed memory accesses and low rate configuration control. While the signals and protocol are the same for both uses, the implementation of the later will typically be optimized for area while the former will be optimized for performance. A specialized system profile is also defined. It is intended to connect platform level resources to the blocks and manage their state independent of a particular data stream.

2 Profile Definitions

2.1 Common Conventions

The OCP specification names signals based on the side of the interface which drives them. All signals driven from the master side are prefixed with an “m”. All signals driven from the slave side are prefixed with an “s”.

Beyond the basic signaling of OCP, certain practices are defined to aid connectivity for both the dataflow and memory profiles. Each interface has a clock and reset signal driven from the master side. In general, these signals only control the interface itself and any buffering tied directly to the interface. However, if a block has a system profile interface, those clock and reset signals are frequently used to control the internal processing of the block. When connecting to the block, no particular relationship should be assumed between the system profile interface signals and other interfaces. Some simple blocks, such as data format converters or inline buffering, may not have a system profile interface. All internal logic of such blocks is implicitly dependent on their other interfaces.

The width of the data path signals, `MData` and `SData`, is completely dependent on the block; however, to allow a common code structure to be used, a fixed length vector is defined. Data is packed into the upper portion of this vector with any remaining bits left unconnected. The synthesis tools are expected to remove the unused bits during optimization. Care should be taken not to exceed the vector length. Assertions to check for width violations exist in the library code. Interpretation of the data signals is not part of profile definition, although an example is provided in Appendix B.

Each of the profiles defined allows flow control from both the master and slave side of the interface. This allows the dataflow rate to dynamically adjust to the requirements of the system. In a system designed to process sampled analog data, the sample rate of the Analog to Digital Converter / Digital to Analog Converter (ADC/DAC) will determine the average dataflow rate, but the flexibility provided by flow control decouples the instantaneous processing rate from the sample rate. A high precision sample clock can be used to maintain the spectral characteristics of the signal and a nominal, slightly faster, processing clock can be used on the common interfaces. Flow control also allows more flexible algorithms such as those with a variable execution speed. However, to fully leverage flow control, signals in each direction are required between master and slave blocks.

2.2 Dataflow Profile

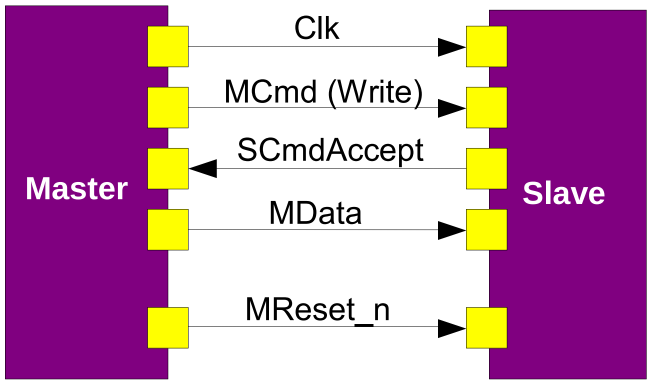

Figure 2-1: Dataflow Source Profile

The dataflow profile was chosen to provide a high performance path with minimal overhead. By reducing the number of features which must be considered, the resources required to implement an interface is kept small. It is expected that resources required for a dataflow interface will be on the same order as those required for a custom interface implementation. This is most important for platforms with critical size, weight, and power (SWaP) restrictions.

Data flows in a single direction, from the master to the slave. Thus an input to the block uses the slave interface while an output from the block uses a master interface. Table 2-1 describes the subset of OCP signals used for this profile.

Table 2-1: Dataflow Source Profile Signals

Name	**Width**	**Function**
Basic Signals		
Clk	1	Clock for interface
MCmd	3	Command – only write and idle are supported
MData	0..N	Data path
SCmdAccept	1	Slave accepts transfer
Sideband Signals		
MReset_n	1	Reset for interface

This small set of signals is adequate for streaming data with support for flow control. The master controls the flow by toggling `MCmd` between idle and write. The slave controls the flow by switching `SCmdAccept`. If a slave is always ready to accept data, it can enable single cycle transfers by holding `SCmdAccept` asserted. For example, one implementation of an infrastructure block to buffer data between consecutive processing blocks is a FIFO configured for write-through. The `SCmdAccept` signal of the infrastructure block's input port could be tied to the inverted FIFO full flag, while the `MCmd` signal of the infrastructure block's output port could be tied to the inverted FIFO empty flag. Once a master has driven `MCmd` to write, it must wait until the slave responds, holding `mData` constant while it waits. If the master determines the transaction is no longer valid, it can cancel the transaction without a slave response by asserting `MReset_n`. The disposition of any data involved in the cancelled transaction is undefined.

2.3 Memory Profile

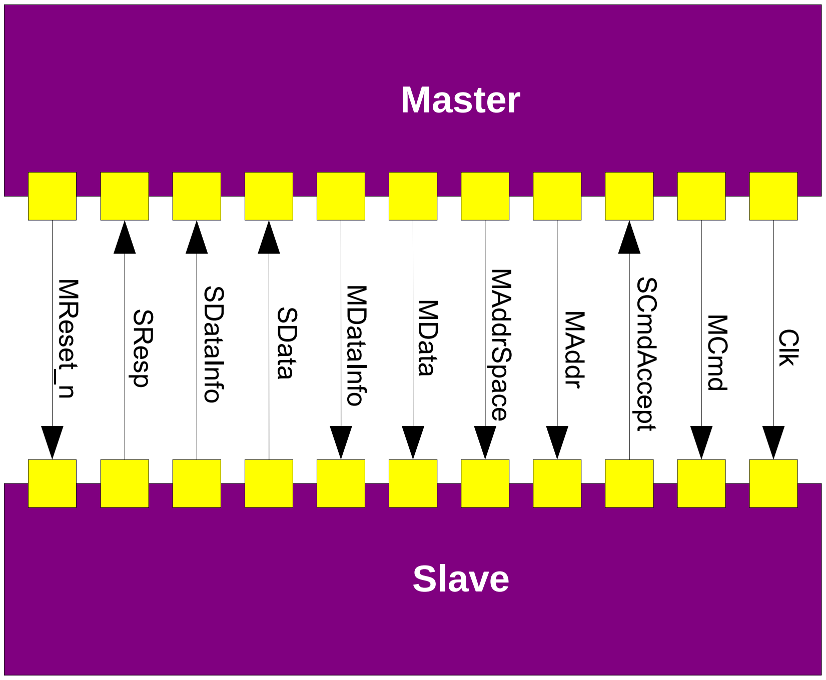

Figure 2-2: Memory Profile

The memory profile was chosen to provide a full featured interface while bounding overhead and implementation effort by excluding more complex features available in OCP. Table 2-2 describes the subset of OCP signals used by this profile.

Table 2-2: Memory Profile Signals

Name	**Width**	**Function**
Basic Signals		
Clk	1	Clock for interface
MAddr	0..N	Transfer address
MCmd	3	Command – read, write and idle are supported
MData	0..N	Write data
SCmdAccept	1	Slave accepts transfer
SData	0..N	Read data
SResp	2	Read command response
Simple Extensions		
MAddrSpace	0..N	Address region selection
MDataInfo	0..N	Control information passed with write data
SDataInfo	0..N	Control information returned with read data
Sideband Signals		
MReset_n	1	Reset for interface

As with the dataflow profile, the master can use the `MReset_n` signal to cancel a transaction without receiving a response from the slave. The reset affects only the interface, not the internal state of the block. The master may not initiate a command unless it can accept the response. The slave has different methods of adding wait states. For writes, it must assert `SCmdAccept` to complete the transaction. For reads, both `SCmdAccept` and `SResp` provide flow control. There is a subtle distinction between the effects of these two signals. The master signals (`MAddr` and `MCmd` for a read command) must be held constant until `SCmdAccept` is asserted. The read transaction completes when `SResp` is asserted. It is common to tie these signals together, however if the slave is capable of queuing the read requests, it can assert `SCmdAccept` immediately even if the data is not yet available. In this way the master can issue a continuous stream of reads despite it taking several cycles for them to complete. See Section 3 for an illustration of some of the typical scenarios. The reads will complete in the order they were issued. The `MAddrSpace` signal can be used to overlay addressable regions with distinct purposes.

2.4 System Profile

The system profile carries global control signals to the block. Unlike other profiles, the data signals are given specific meaning for use in this profile. To facilitate the platform concepts of resource allocation and power management, a set of block states are defined. Figure 2-3 shows the typical state transition flow. To limit the number of operations required make a block active, the Start and Release commands are defined to pass through the Configure state automatically.

The controller sends operations to manage the block state via OCP write commands. The block can defer an operation by holding `SCmdAccept` negated. The controller must continue to issue the request until it is accepted in accordance with the OCP signaling protocol. The system profile enables the OCP write response option to allow the slave block to accept the operation request and free the master for further accesses while providing an asynchronous notification that the operation completed. If the operation cannot be completed, the slave returns an error response.

The operation is encoded on the `MData` signal during a write command as described in Table 2-3. The encodings are defined to limit the number of bits which must be decoded by the slave. The Test state is optional and a slave block which does not implement it will return an error response to the test operation. The current state can be retrieved via a read command. Table 2-4 describes the state encodings returned in the `SData` signal. All of the state transitions are driven by operation from the system profile master.

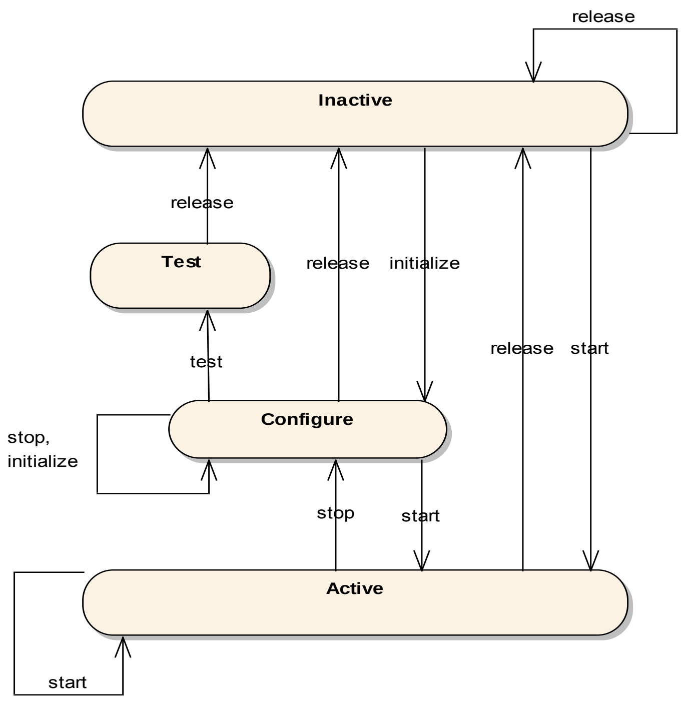

Figure 2-3: System Profile State Transition Diagram

Table 2-3: Operation Encoding

Operation	Value	Description
Initialize	0100	Advance to the Configure state. This operation is issued from the Inactive state.
Start	xx01	Advance to the Ready state. This operation may be issued from the Configure or Inactive states. If issued from the Inactive state, an Initialize operation is implied.
Stop	1100	Return to Configure state. This operation can be issued from Active state.
Release	xx1x	Return to Inactive state. This operation may be issued from any state. If issued from the Active state, a Stop operation is implied. The `MReset_n` signal executes an implied Release command.
Test	1000	Enter Test state. This operation is called from the Configure state. Support for the Test state is optional. Any block not supporting the Test state ignores this operation.

Table 2-4: State Encoding

State	Value	Condition
Inactive	000	Block has not been initialized. A block in this state should minimize its resource/power usage as much as possible. The block is only required to process the Initialize command. Other interfaces of the block must be inactive. This is the default power on state of all blocks
Configure	001	Block has been initialized and is ready to run but will not process data. It should respond to its other interfaces, but not run its internal algorithm.
Active	010	Block has been issued a start command and will process data as it becomes available
Test	1xx	Block is running a test. Operation during test mode is defined by the block. The additional bits may hold information regarding the state of the test.

Table 2-5 describes the OCP signals used by this profile. Unlike the dataflow and memory profiles, the signals on the system profile directly interact with the internal operation of the block.

Table 2-5: System Profile Signals

Name	**Width**	**Function**
Basic Signals		
Clk	1	Primary block clock
MCmd	3	Command – read, write and idle are supported
MData	0..N	Operation to perform (see Table 2-3)
SCmdAccept	1	Slave accepts transfer
SData	0..N	Current state (see Table 2-4)
SResp	2	Read command response.
Sideband Signals		
MReset_n	1	Reset block
SError	1	Block fatal error condition
SInterrupt	1	Block service request
Status	0..N	Block debug information

When `MReset_n` is sampled asserted (active low), regardless of the current state, the block returns to the inactive state and any state or configuration information of the block is cleared.

`SInterrupt` provides a mechanism for the block to signal the platform of various conditions. In most cases this prompts the platform to query the block regarding the action to be taken. The mechanism of this query is outside the scope of the common interface but could be implemented in a status register using a memory profile interface.

`SError` indicates a fatal condition in which the block may be unable to correctly respond to its other ports. It must be cleared using the `MReset_n` signal.

The Status signal is completely undefined. It is available for the block to send real-time debug information up to the platform infrastructure. The block must not expect any response to signals in the Status vector; it provides visibility into the block during debug and analysis. The Status vector is the only mechanism available to send additional information when `SError` is asserted.

3 Using Interface Profiles

Each interface has a master block that initiates transactions and a slave block that responds to them. All interfaces follow the protocol defined by the OCP. The following sections provide examples of how the protocol is used in the profiles defined by the common interfaces. Consult the OCP Specification [3] for a complete discussion of the protocol - Section 4 of the specification discusses the protocol semantics and the beginning of Section 9 provides specific implementation guidance.

3.1 Dataflow Source Profile

The dataflow source profile is used when the master produces data to send to a slave. Since only write commands are issued, the slave never returns a response to the master.

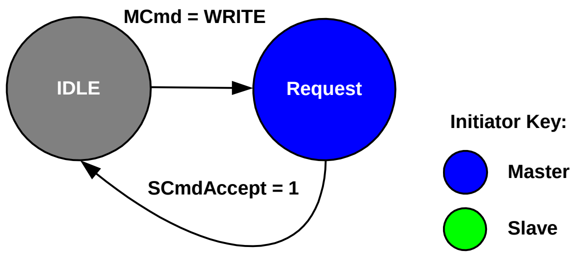

Figure 3-1: Dataflow Source Profile OCP Phase Diagram

Although the slave does not respond, it can still control the flow of data by when it accepts the command. Once the master has issued a command, it must hold the command and associated data constant until the slave accepts it. If the slave knows it will be able to accept a write, it can assert its `SCmdAccept` signal prior to the write request. This is necessary to support continuous data flow. Figure 3-2 and Figure 3-3 show two examples of a transaction sequence. In the first, the slave uses positive control, asserting `SCmdAccept` for each command. The maximum duty cycle that can be achieved is 50% since `SCmdAccept` must toggle between each transaction. In the second, the slave uses negative control, deasserting `SCmdAccept` when it cannot process data. This type of control has no limit on duty cycle beyond that imposed by the blocks.

While a block can choose how to act on its side of the interface, it must support any valid operation from the opposite side.

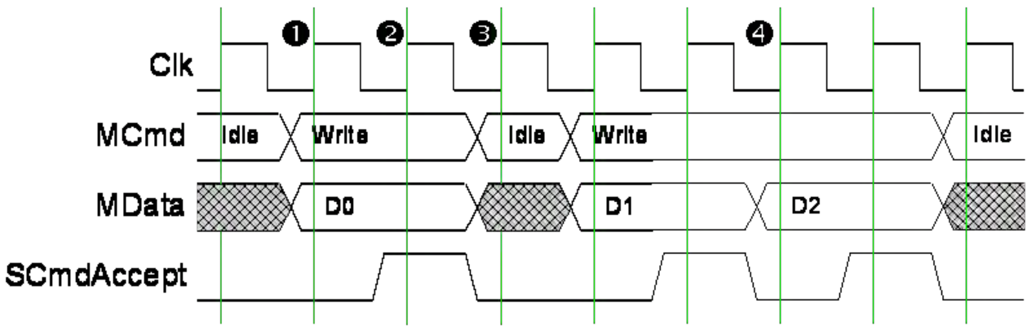

Figure 3-2: Example Timing Diagram: Responding Write

Figure 3-2 illustrates the following behavior:

1. Master issues Write and drives data on `MData`.
2. Slave accepts command by asserting `SCmdAccept`; Master is free to change.
3. Slave must deassert `SCmdAccept` unless it can immediately accept next write.
4. Example of master issuing back-to-back Write commands. If slave did not deassert `SCmdAccept` after 1 cycle, the D2 cycle would end early.

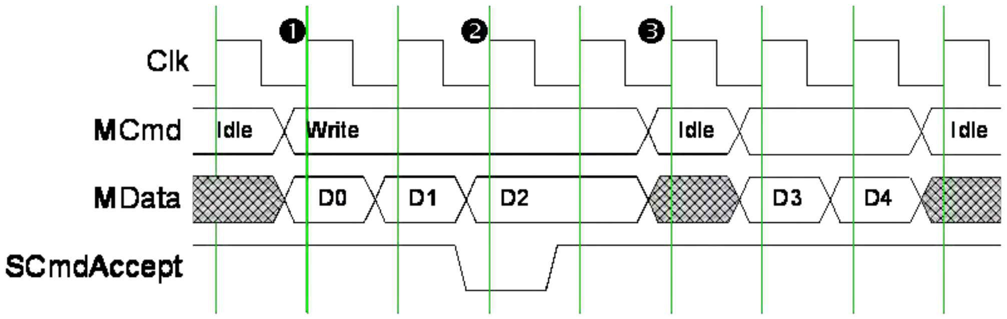

Figure 3-3: Example Timing Diagram: High Speed Write

Figure 3-3 illustrates the following behavior:

1. Master issues Write and drives data on `MData`; since this example slave can immediately accept data it holds `SCmdAccept` asserted.
2. Slave can force wait states by deasserting `SCmdAccept`; Master must hold Write request.
3. Master must deassert `MCmd` when no data is available.

3.2 Memory Profile

The memory profile adds the option for the master to read data. Both the master and the slave produce data, but the master always initiates the transactions. As with the dataflow source profile, `SCmdAccept` can be used by the slave the control the flow of data. For read accesses, the response phase can be used also be used. Whenever possible, a slave should delay the response rather than not accepting the command since this allows the master to regain control. However, every block must be able to handle any valid behavior from its counterpart.

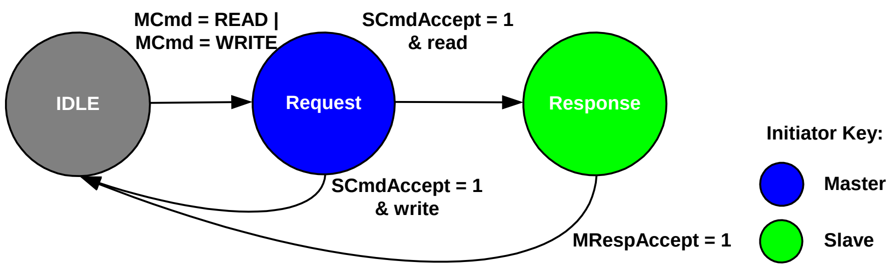

Figure 3-4: Memory Profile OCP Phase Diagram

The signal behavior during write commands is the same as described for the dataflow source profile, see Section 3.1 for examples. The address signals share the timing properties of the master request signals. For read commands, several behaviors are possible as illustrated in the following examples.

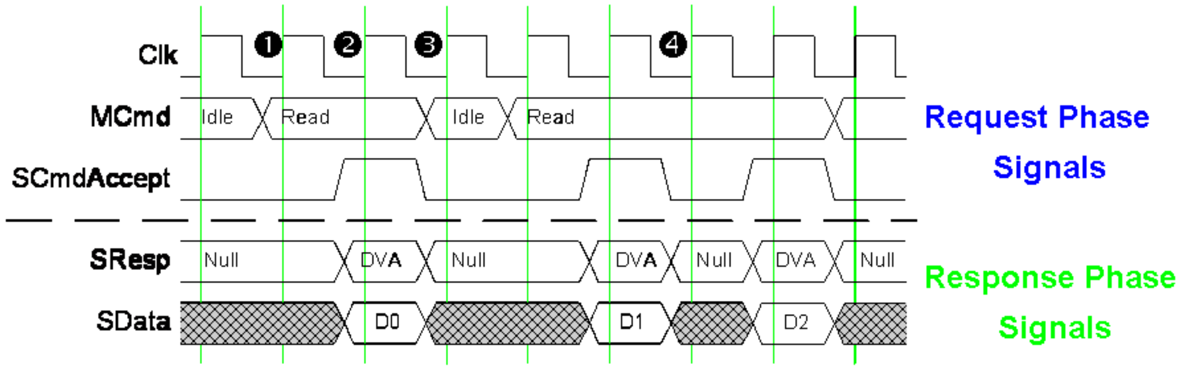

Figure 3-5: Example Timing Diagram: Responding Read

Figure 3-5 illustrates a simple slave that responds to each read as it is issued. This type of slave is appropriate for low rate control paths. The maximum duty cycle it can achieve is 50% since it must constantly toggle `SCmdAccept`. The following behavior is illustrated:

1. Master issues Read.

2. When the slave has data available, it accepts the command by asserting `SCmdAccept`, responds and drives `SData` at the same time.
3. Slave must deassert `SCmdAccept` and `SResp`; Master must latch data. Each response lasts only a single cycle (see Appendix C.2.1).
4. Example of master issuing back-to-back Read commands.

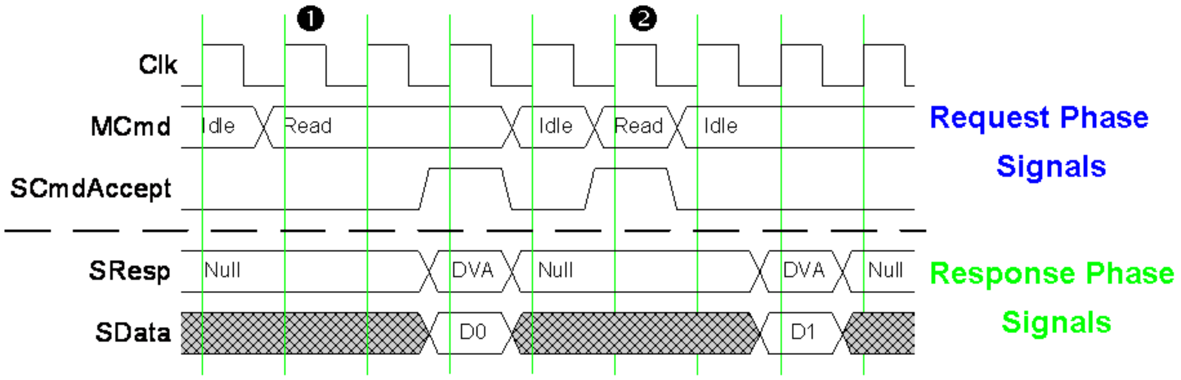

Figure 3-6: Example Timing Diagram: Delayed Read

Figure 3-6 illustrates the two mechanisms a slave can use to delay a transaction.

1. Slave does not accept the command when it is issued and the master must hold the request constant until it does.
2. Slave accepts the command immediately allowing the master to release the request and possibly make another. The slave responds later when the data is available.

A slave can also combine the two techniques, delaying the acceptance and further delaying the response. Note that if a slave uses the delayed response technique, it must consider the correct behavior if the master follows with a write request to the same address as the pending read.

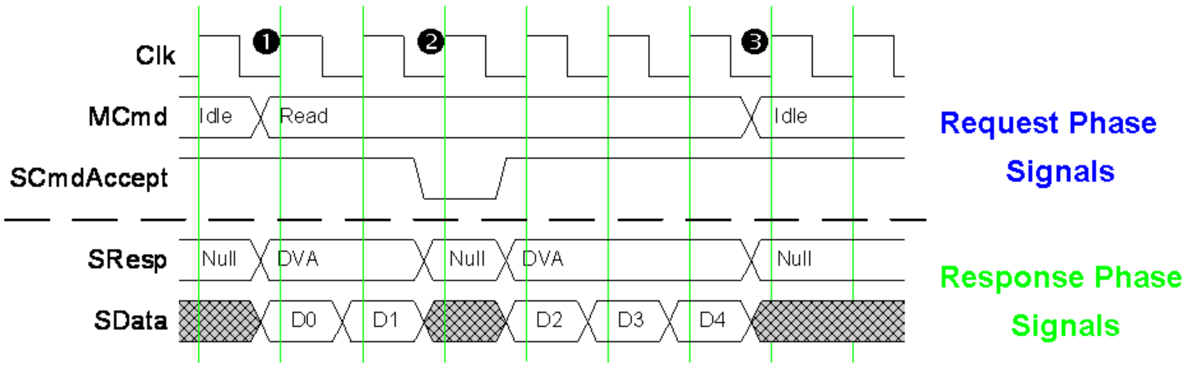

Figure 3-7: Example Timing Diagram: High Speed Read

Figure 3-7 illustrates a series of high speed read accesses. As with the high speed write accesses in the dataflow source profile, the slave can delay a transfer by deasserting `SCmdAccept`. When the master cannot accept more data, it must deassert `MCmd`.

1. Master issues Read; slave already has `SCmdAccept` asserted.
2. Slave can force wait states by deasserting `SCmdAccept`; Master must hold Read request.
3. Master must deassert `MCmd` when it cannot accept more data.

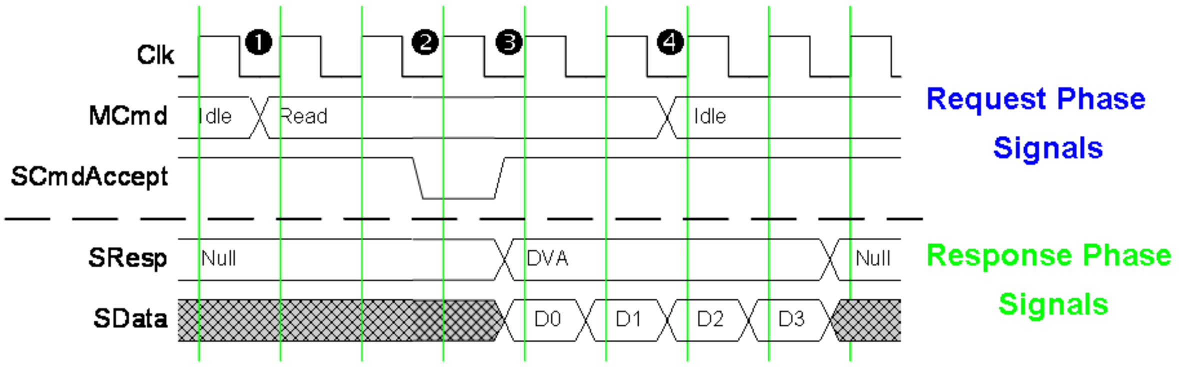

Figure 3-8: Example Timing Diagram: Queued Reads

The OCP decouples the request and response phases of the transaction allowing a master to queue multiple commands before any response is received. Queued reads have important implications for the design on masters and slaves. The memory profile allows only one cycle for each response, so the master must be able to latch the data immediately. The master must only issue the number of requests it can consecutively handle responses to. Similarly the slave must only accept the number of commands it can track. A slave may not respond to a command it has not received, so it must keep track of the number of accepted requests. In Figure 3-8, the slave queues two read requests from point 1 to point 2. If, as in this example, the slave can support a maximum of two requests, it must insert a wait state at point 2 by deasserting `SCmdAccept`. At point 3 the slave begins responding. Since it can now accept additional requests, it also reasserts `SCmdAccept`. At point 4, there are still two outstanding requests (four have been accepted; two have been responded to). If the conditions of the master block do not allow a third outstanding request, the master must immediately deassert `MCmd`.

Any queued transactions are clear if `MReset_n` is asserted.

3.3 System Profile

The system profile follows the protocol behavior outlined for the memory profile on read commands, but differs for write commands. To provide the master feedback that the write command has been processed, and not just accepted, a write command in the system profile also generates a response phase. The value of the `SData` signal during a response to a write command is ignored.

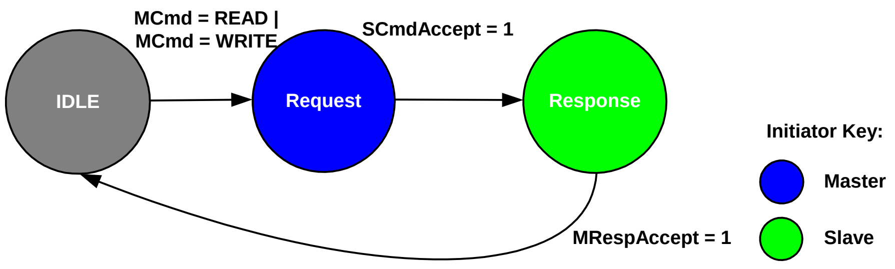

Figure 3-9: System Profile OCP Phase Diagram

The additional sideband signals of the system interface will be synchronous to the interface clock, but independent of the normal phase protocol.

4 Documenting the Interfaces

Documentation of the interfaces used on a block is critical to successful integration onto a platform. Each block must list the interfaces it provides including the profile, role, and values of all optional parameters. Additionally, performance values for the interface must be listed. The following tables provide a list of the values to specify for each interface type.

Table 4-1: Data Flow Source Interface Master Documentation

Signals	
MData	Specify the format of the data and its width. The width can also be left as a compile time parameter (e.g. in VHDL the width can be specified with a generic).
Endianness	Endianness of data if applicable
Performance	
MCmd rate	Maximum/Minimum/Typical rate of write requests as a ratio of active cycles to total cycles assuming the slave imposes no wait states. Also specify what conditions influence these values.

Table 4-2: Data Flow Source Interface Slave Documentation

Signals	
MData	Specify the format of the data and its width. The width can also be left as a compile time parameter (e.g. in VHDL the width can be specified with a generic).
Endianness	Endianness of data if applicable
Performance	
SCmdAccept rate	Maximum/Minimum/Typical number of wait states inserted per command. Also specify what conditions influence these values.

Table 4-3: Memory Interface Master Documentation

Signals	
MCmd	List supported commands (read and/or write).
MAddr, MAddrSpace	Supply the address range and how many, if any, address spaces are supported. If multiple address spaces are supported, state the difference between them. Specify the width of the busses.
MData, SData	Specify the format of the data and its width. The width can also be left as a compile time parameter (e.g. in VHDL the width can be specified with a generic). Note that the size of the read and write data must be the same.
Endianness	Endianness of data if applicable
MDataInfo, SDataInfo	If either option is enabled, specify how many bits are used and what they represent.
SResp	State the behavior if a failure or error is received.
Performance	
MCmd rate	Maximum/Minimum/Typical rate of requests as a ratio of active cycles to total cycles assuming the slave imposes no wait states. Also specify what conditions influence this value.

Table 4-4: Memory Interface Slave Documentation

Signals	
MCmd	List supported commands (read and/or write)
MAddr, MAddrSpace	Supply the address range and how many, if any, address spaces are supported. If multiple address spaces are supported, state the difference between them. Specify the width of the busses.
MData, SData	Specify the format of the data and its width. The width can also be left as a compile time parameter (e.g. in VHDL the width can be specified with a generic). Note that the padded size of the read and write data must be the same.
Endianness	Endianness of data if applicable
MDataInfo, SDataInfo	If either option is enabled, specify how many bits are used and what they represent.
SResp	State under what conditions, if any, error and/or failure is generated.
Performance	
SCmdAccept rate	Maximum/Minimum/Typical number of wait states inserted per command and what conditions influence this. Also, indicate if queued reads are supported and if so, how many.
SResp rate	Maximum/Minimum/Typical read latency. Also specify what conditions influence these values.

Table 4-5: System Interface Slave Documentation

Signals	
SError, SInterrupt	State if either of these signals is generated and what it indicates along with the desired response from the system.
MData,SData	State if the block performs advanced state control. Also describe the behavior of the Test state if implemented.
Status	State how many status bits are generated and their purpose. If they are general debug lines, it is not necessary to list the specific signals as this is likely to change during the development process.
Performance	
Response time	Indicate how quickly the block will respond to `MReset_n` and Control.

References

1. Skey, Kevin, John Bradley, Karl Wagner, October 2006, “A Reuse Approach for FPGA-Based SDR Waveforms”, *Milcom 2006*, The MITRE Corporation, Bedford, MA.
2. MITRE, February 2008, *Methodology for Design of Portable FPGA-Based Applications: Methodology Overview, Revision 1.1*, MTR070295V2, The MITRE Corporation, Bedford, MA.
3. OCP International Partnership, 2005, *Open Core Protocol Specification*, Release 2.1.

Appendix A OCP Configuration Parameters

The following tables list the OCP configuration parameters and possible values used to configure each type of profile. Parameters with a fixed value that differs from the OCP defined default value have only a single option listed in the value column. Parameters that can be selected from multiple values list the possible options with the default value first. Any open range is specified with an upper bound of N; the common use of N does not imply a relationship. Each parameter can be specified independently. Parameters which must use their OCP defined default values are not listed in the table.

Table A-1: Data Flow Source Profile OCP Configuration Parameters

Parameter	Value
Protocol	
Burstseq_incr_enable	0
Endian	Neutral
Read_enable	0
Signal	
Addr	0
Data_wdth	0..N
Resp	0
Sdata	0
Mreset	1
Sreset	0

Table A-2: Memory Profile OCP Configuration Parameters

Parameter	Value
Protocol	
Burstseq_incr_enable	0
Endian	neutral
Read_enable	1 | 0
Write_enable	1 | 0

Signal	
Addr	1| 0
Addr_wdth	0..N
Addrspace	0 | 1
Addrspace_wdth	0..N
Data_wdth	0..N
Mdata	1 | 0
Mdatainfo	0 | 1
Mdatainfo_wdth	0..N
Resp	1 | 0
Sdata	1 | 0
Sdatainfo	0 | 1
Sdatainfo_wdth	0..N
Mreset	1
Sreset	0

Table A-3: System Profile OCP Configuration Parameters

Parameter	Value
Protocol	
Burstseq_incr_enable	0
Phase	
Writeresp_enable	1
Signal	
Addr	0
Data_wdth	4
Interrupt	0 | 1
Mreset	1
Serror	0 | 1

Sreset	0
Status	0 | 1
Status_wdth	0..N
Clkctrl_enable	0 | 1
Scanctrl_wdth	0..N
Scanport	0 | 1
Scanport_wdth	0..N

Appendix B Example Data Field Mapping

The abstract interfaces used to describe the block interfaces must be encoded to pass over the OCP `MData`/`SData` signals. While outside the scope of the common profiles, this section describes the mapping used by the MITRE methodology for designing portable applications [1] as an example.

The encoding is designed to support the parallel transport of data which is common between FPGA based components and necessary to achieve arbitrarily high data rates. Parallelism is specified either implicitly where multiple operation calls are packed into a single transaction, or explicitly where the interface is designed to take a sequence type argument.

B.1 Implicitly Parallel Interfaces

An implicitly parallel interface can provide a simpler view of the block at the abstract level; see Figure B-1 for examples. The operations defined by the interface represent atomic functions. Multiple atomic function calls can be grouped to form each transaction. Any number of operations can be defined in the interface and each operation may have any number of arguments. Arguments can be doubles (encoded as fixed point values), other numeric scalars which will convert to doubles, or structures of these scalar types.

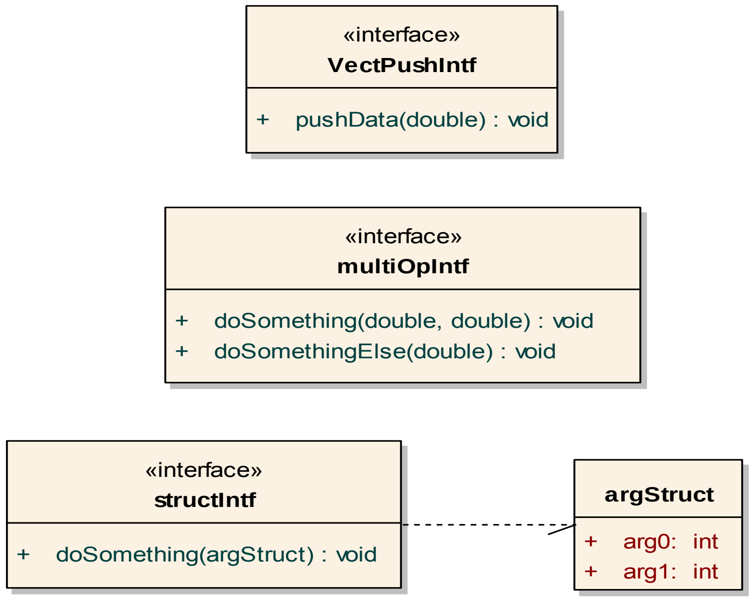
«interface»
VectPushIntf
+ pushData(double) : void
«interface»
multiOpIntf
+ doSomething(double, double) : void
+ doSomethingElse(double) : void
«interface»
structIntf
+ doSomething(argStruct) : void
argStruct
+ arg0: int
+ arg1: int

Figure B-1: Examples of Implicitly Parallel Interfaces

The OCP data word is formed by concatenating the arguments from each function call into a single string of bits. If more than one operation is defined for the interface, an ordinal value, the operation (OP) field, is added before the arguments. The ordinal value uses a binary encoding with the minimum number of bits needed to represent all operations for the interface. If only one operation is defined, the OP field is omitted. Figure B-2 shows the general structure of the encoded data word. Figure B-3 shows specific examples of how the interfaces in Figure B-1 would be encoded. The OCP data words are defined as a constant width, so pad bits are added to the least significant bits of the word. The synthesis engine is expected to remove these during optimization. For a multiple-operation interface, space is allocated in the OCP data word for the longest possible argument list. These extra bits cannot be optimized away. They will be left at their default value when encoding operations using fewer arguments.

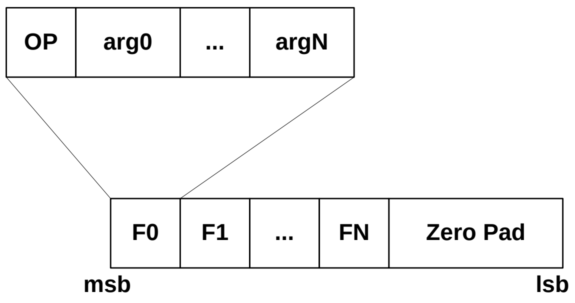

Figure B-2: OCP Data Word Encoding

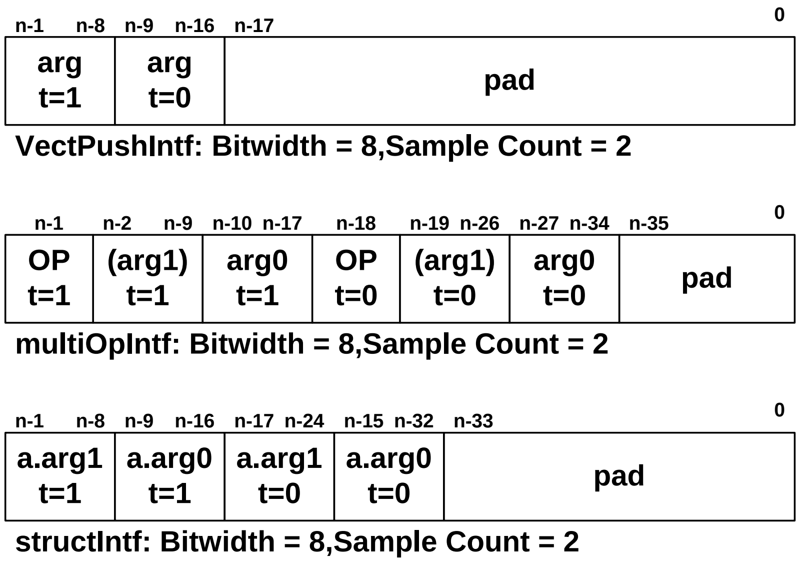

Figure B-3: Example Interface Encodings

B.2 Explicitly Parallel Interfaces

An explicitly parallel interface has one argument with a sequence type; only a single operation and single argument are allowed. The mapping of function calls to OCP transactions is slightly more complicated than the implicitly parallel case. Function calls are buffered or split into OCP transactions according to a fixed parameter. Explicitly parallel interfaces are useful for streams of arbitrarily grouped data samples. Applications requiring groups of samples to be passed from block to block are well suited for this style of interface. The base type of the sequence can be double, another numeric scalar which will convert to a double, or a structure of these scalar types.

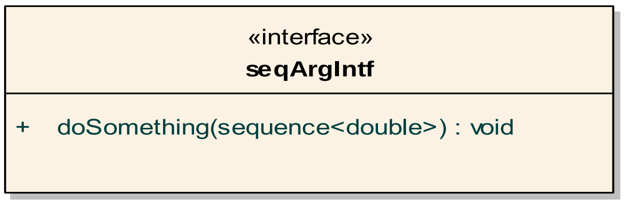

Figure B-4: Example of Explicitly Parallel Interface

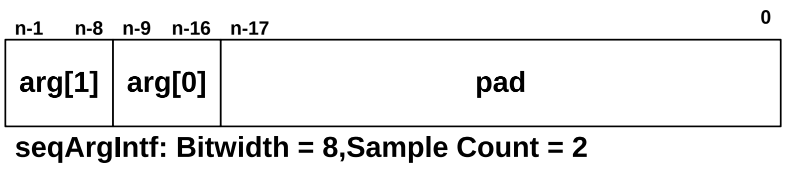

Figure B-5: Example Interface Encoding

Appendix C Profile Implementations

C.1 Interface Clocking

Each interface profile contains its own clock signal to help decouple aspects of the block. Although the layers are collapsed during synthesis, passing clock signals through multiple levels of assignment can cause errors during simulation. Chained clock assignments may also be incompatible with clock distribution methods used in very large designs. Using a system level master clock tree rather than routing the clock signal through each block is acceptable if none of the blocks modify their interface clocks.

Additionally, costly synchronization logic can be avoided if clocks are known to be the same internal to the blocks. The valid relationship between all clocks in a block is clearly documented and it is valid to restrict the relationship if it significantly simplifies the design.

C.2 Proposed Changes

While implementing the initial version of these profiles, various possible modifications to enhance their capabilities and simplify their implementations were uncovered. Although existing blocks may not support these features, they should be considered when designing new blocks to ease future upgrades.

C.2.1 MRespAccept

The Memory and System profiles currently require that a master immediately process the `SResp` signal. While the master can deassert `MCmd` to slow the rate of incoming data, when connected to a slave supporting queued reads the master may issue several reads before detecting the need to stall the flow of data. In order to conform to the one cycle response requirement, the master must be able to store any pending responses. Adding `MRespAccept` to the profiles will remove the extra buffering requirement by allowing the master to extend the response cycle.

The default value for `MRespAccept` is „1'. To be compatible with existing code, masters must be able to operate with `MRespAccept` constantly asserted. New slaves should be designed to support `MRespAccept`.

C.2.2 MDataInfo / SDataInfo

With the structured definition of the `MData`/`SData` fields as defined in Appendix B, any combination of samples can be represented removing the need for the extension signals. New components should not rely on the DataInfo signals as they will likely be removed from future versions.

Appendix D   Example Profile Use

D.1   Wrapping External Interface

To use the common interface profiles in a system, gaskets must be written to convert the external interfaces of the device to the common interface profiles. The intended use of the external connection must be identified and the appropriate common profile to connect to must be chosen. There are many standard and custom interfaces a system might use and the details of the gasket will depend on the specifics of the actual interface. The example described in this section illustrates connecting a typical processor peripheral bus to the memory profile. The processor will initiate transfers, so it implements the master side of the profile. The uPrcBusGasket entity shown in Figure D-1 provides separate read and write data paths. An appropriate structure to demultiplex the bidirectional pins is assumed to exist.

```
entity uPrcBusGasket is
generic
(
   DATA_W:   integer;
   ADDR_W:   integer
);
port
(  -- Processor Side
   clk:    in  std_logic;
   reset:  in  std_logic;
   wrData: in  std_logic_vector(DATA_W-1 downto 0);
   rdData: out std_logic_vector(DATA_W-1 downto 0);
   addr:   in  std_logic_vector(DATA_W-1 downto 0);
   wrEn:   in  std_logic;
   cs:     in  std_logic;
   ready:  out std_logic;
   -- OCP Side
   ocpClk:    out std_logic;
   ocpMaster: out MemoryMasterInterfaceType;
   ocpSlave:  in  MemorySlaveInterfaceType
);
end uPrcBusGasket;
```

Figure D-1: Example Processor Bus Gasket

When a write transaction occurs, `cs`, `wrEn`, `addr` and `wrData` are driven by the processor. For a read transaction, the processor drives `cs` and `addr`, then samples `rdData`. In both cases, the peripheral can insert wait states by delaying the assertion of `ready`. As a processor control bus, it is not necessary to support high speed data transfers. Furthermore, since each access will hold the processor bus until complete, it is not necessary to issue queued reads. The following code fragment illustrates a possible implementation of this gasket:

```
ocpClk <= clk;

accessPrc: process(clk)
begin
   if(rising_edge(clk)) then
      ocpMaster.mReset_n <= not(reset);
      if(reset = '1') then
         ready <= '0';
         ocpMaster.mCmd <= OCP_CMD_IDLE;
         fsmState <= stIdle;
      else
         ocpMaster.mData <= resizeToMSB(wrData, ocpMaster.mData'length);
         rdData <= resize(ocpSlave.sData, rdData'length);
         case fsmState is
         when stIdle =>
            if(cs = '1') then
               ocpMaster.mAddr <=resizeToLSB(addr,ocpMaster.mAddr'length);
               if(r_wn = '1') then
                  ocpMaster.mCmd <= OCP_CMD_READ;
               else
                  ocpMaster.mCmd <= OCP_CMD_WRITE;
               end if;
               fsmState <= stReq;
            end if;
         when stReq =>
            if(ocpSlave.sCmdAccept = '1') then
               ocpMaster.mCmd <= OCP_CMD_IDLE;
               if((r_wn = '1') and (ocpSlave.sResp = OCP_RSP_NULL)) then
                  fsmState <= stResp;
               else
                  ready <= '1';
                  fsmState <= stIdle;
               end if;
            end if;
         when stResp =>
            if(ocpSlave.sResp != OCP_RSP_NULL) then
               ready <= '1';
               fsmState <= stIdle;
            end if;
         when others => null;
         end case;
      end if;
   end if;
end process;
```

D.2 Wrapping a Processing Block

If a block is being designed from scratch, it can be programmed to access the common profiles natively. If code using a nonstandard interface already exists to implement the algorithmic portion of the block, it can be wrapped using the common profiles. This section describes an illustrative example of wrapping an existing block. If the MITRE methodology for designing portable applications is used, much of this wrapping can be automated through code generation scripts.

```
entity exampleBlock is
   generic
   (
      DIN_W:   integer;
      DOUT_W:  integer;
      PROP_W: integer
   );
   port
   (
      clk:    in std_logic;
      reset:  in std_logic;
      enable: in std_logic;
      din:    in std_logic_vector(DIN_W-1 downto 0);
      ready:  out std_logic;
      dout:   out std_logic_vector(DOUT_W-1 downto 0);
      prop0:  in std_logic_vector(PROP_W-1 downto 0);
      prop1:  in std_logic_vector(PROP_W-1 downto 0)
   );
end exampleBlock;
```

Figure D-2: Example Non-OCP Entity

Figure 2-1 shows an example of a typical interface to a block of data processing code. In this example, all signals are synchronous to the clock signal. If `reset` is asserted, the processing code returns to its initialization state. If `enable` is asserted, the value on `din` is loaded into the processing chain. After some pipeline latency, `ready` is asserted and the processed value is provided on `dout`. The code supports pipelined processing; therefore a new input can be loaded before the previous one has reached the output. Two run-time configurable properties, `prop0` and `prop1`, are used to control the data processing.

The first step of wrapping the block is to define the interfaces used. Control of the clock and `reset` signal is mapped to a system profile interface, the `enable` and `din` signals are mapped to a dataflow source profile slave interface, the `ready` and `dout` signals are mapped to a dataflow source profile master interface and the properties are mapped to a memory profile interface. This is the mapping used for this example, but not the only mapping possible. When mapping the interfaces, the available supporting structures and overall system configuration must be considered. Furthermore, if the interface to the block is more complex it may be necessary to provide additional control logic rather than specify a direct mapping of signals to interfaces.

```
entity exampleBlockOcp is
generic
(
   DIN_W:    integer;
   DOUT_W:   integer;
   PROP_W:  integer
);
port
(  -- System Interface
   sysClk:        in std_logic;
   sysMaster:     in SystemMasterInterfaceType;
   sysSlave:      out SystemSlaveInterfaceType;
   -- Property Interface
   ps1Clk:        in  std_logic;
   ps1Master:     in  MemoryMasterInterfaceType;
   ps1Slave:      out MemorySlaveInterfaceType;
   -- Data Input Interface
   dinClk:      out std_logic;
   dinMaster:   in DataFlowSourceMasterInterfaceType;
   dinSlave:    out DataFlowSourceSlaveInterfaceType;
   -- Data Output Interface
   doutClk:     out std_logic;
   doutMaster:  out DataFlowSourceMasterInterfaceType;
   doutSlave:   in  DataFlowSourceSlaveInterfaceType
);
end exampleBlockOcp;
```

Figure D-3: Example OCP Wrapped Entity

The system interface wrapper provides a simple state machine to implement the system profile state transition diagram in Figure 2-3. When the block is in the inactive state, the internal reset signal is asserted. Additionally, when the block in the active state, a control signal turns the dataflow interfaces on. This simple example block does not use the interrupt, error, or status features of the system interface. The following code fragment illustrates an implementation of this state machine:

```
cmdPrc: process(sysClk)
   variable op : std_logic_vector(3 downto 0);
begin
   if(rising_edge(sysClk)) then
      slv.sCmdAccept <= '0';
      slv.sResp <= OP_RSP_NULL;
      if(mst.mReset_n = '0') then
         stateFlag <= OCP_SYS_STATE_INACTIVE;
         moduleOff <= '1';
         moduleCfg <= '0';
         moduleOn  <= '0';
      else
         if(mst.mCmd = OCP_CMD_READ) then
            slv.sCmdAccept <= '1';
            slv.sResp <= OCP_RSP_ACCEPT;
            slv.sData <= resize(stateFlag, slv.sData'length)
         end if;

         if(mst.mCmd = OCP_CMD_WRITE) then
            op := resize(mst.mData,op'length);
            slv.sCmdAccept <= '1';
            slv.sResp <= OCP_RSP_ACCEPT;
            case op is
            when OCP_SYS_OP_INITIALIZE =>
               if(stateFlag = OCP_SYS_STATE_INACTIVE) then
                  stateFlag <= OCP_SYS_STATE_CONFIGURE;
                  moduleOff <= '0';
                  moduleCfg <= '1';
                elsif((stateFlag = OCP_SYS_STATE_READY) or
                      (stateFlag = OCP_SYS_STATE_ACTIVE)) then
                  slv.sResp <= OCP_RSP_ERR;
                end if;
            when OCP_SYS_OP_START =>
               stateFlag <= OCP_SYS_STATE_READY;
               moduleOff <= '0';
               moduleCfg <= '0';
               moduleOn  <= '1';
            when OCP_SYS_OP_STOP =>
               if((stateFlag = OCP_SYS_STATE_READY) or
                  (stateFlag = OCP_SYS_STATE_ACTIVE)) then
                  stateFlag <= OCP_SYS_STATE_CONFIGURE;
                  moduleCfg <= '1';
                  moduleOn  <= '0';
               elsif(stateFlag = OCP_SYS_STATE_INACTIVE) then
                  slv.sResp <= OCP_RSP_ERR;
               end if;
            when OCP_SYS_OP_RELEASE =>
               stateFlag <= OCP_SYS_STATE_INACTIVE;
               moduleOff <= '1';
               moduleCfg <= '0';
               moduleOn  <= '0';
            when others =>
               slv.sResp <= OCP_RSP_ERR;
            end case;
         end if;
```

```
      end if;
   end if;
end process;
```

The memory interface wrapper provides storage for the two properties and synchronizes their values to the system clock domain. Each property is given its own address so that it can be accessed individually. It is also helpful during system integration to allow the current value to be read back over the memory interface. The following code fragment illustrates a possible simple implementation:

```
slv.sCmdAccept <= '1' when (mst.mCmd != OCP_CMD_IDLE) else '0';
queryPrc: process(clk)
begin
   if(rising_edge(clk)) then
      slv.sResp <= OCP_RSP_NULL;
      if(mst.mCmd = OCP_CMD_READ) then
         slv.sResp <= OCP_RSP_ACCEPT;
         slv.sData <= (others => '0');
         case conv_integer(mst.mAddr) is
         when PS1_PROP0_ADDR =>
            slv.sData <= resize(prop0, slv.sData’length);
         when PS1_PROP1_ADDR =>
            slv.sData <= resize(prop1, slv.sData’length);
         when others => null;
         end case;
      end if;
   end if;
end process;

configPrc: process(clk)
begin
   if(rising_edge(clk)) then
      if(moduleOff = '1') then
         prop0 <= (others => '0');
         prop1 <= (others => '0');
      elsif(mst.mCmd = OCP_CMD_WRITE) then
         case conv_integer(mst.mAddr)is
         when PS1_PROP0_ADDR =>
            prop0 <= resize(mst.mData, prop0’length);
         when PS1_PROP0_ADDR =>
            prop0 <= resize(mst.mData, prop1’length);
         when others => null;
         end case;
      end if;
   end if;
end process;
```

The output dataflow source interface wrapper handles downstream flow control of the data. When new data is available as indicated by the worker done signal, the wrapper initiates a transaction. If the downstream connection blocks the transaction, the wrapper must insert wait states so that data is not lost. This works in conjunction with the input dataflow source interface. After the block moves to the active state, transactions are accepted from the upstream connection. If it blocks, the data processing must be stalled. The following code fragment illustrates an example implementation of the dataflow interfaces:

```
inAccept <= (roomForOutput and workerReady) when (moduleOn = '1') else
'0';
newWorkerData <= inAccept when (iMst.Cmd = OCP_CMD_WRITE) else '0';
iSlv.sCmdAccept <= inAccept;

oClk <= iClk;
oMst.mCmd <= OCP_CMD_WRITE when (doAWrite = '1') else OCP_CMD_IDLE;
oMst.mData <= dataToWrite;

roomForOutput <= '0' when (doAWrite = '1' and oSlv.sCmdAccept = '0') else
'1';
outPrc: process(iClk)
begin
    if(rising_edge(iClk)) then
      if(moduleOn = '0') then
        doAWrite <= '0';
        dataOutPending <= '0';
      else
        if(roomForOutput = '0' and workerDone = '1') then
          dataOutPending <= '1';
        end if;
        if(roomForOutput = '1' and workerDone = '1') then
          dataOutPending <= '0';
          dataToWrite <= workerDout;
          doAWrite <= '1';
        elsif(doAWrite = '1' and oSlv.sCmdAccept = '1') then
          doAWrite <= '0';
        end if;
      end if;
    end if;
end process outPrc;
```

Appendix E Common PropertySet Interface

To provide the most flexibility in interfacing to workers while still maintaining a degree of interface commonality, each component supports a set of registers broken into two groups. The first group is common to all components. It implements a simple command/flag interface in support of component generated interrupts. It also implements a worker direct access pass-through mechanism which allows workers containing their own memory style interface to connect to the standardized property set interface. The second group is custom to each component. It supports storage of properties directly in the generated code and presents their values to the workers. Table below describes the common registers implemented in all components. In the direction column, r/w describes a register that can be written to set a new value and read to find its current setting. Registers marked wtc (write-to-clear) are flags which are latched when set by the worker allowing the worker the flexibility to drive them with a single pulse. They will remain set until written with a „1'. Each bit it handled individually.

Table E-4: Common Register Interface

Address	Name	Dir	Description
0	Csr	Wtc	Unmasked command/status register
1	Istat	Wtc	Interrupt status register
2	Ien	r/w	Interrupt enable mask
3	WdIncr	r/w	Worker direct address increment value
4	WdAddr	r/w	Worker direct address register
5	WdData	r/w	Worker direct data register
6-31	Reserved		

NOTE: Although the interrupt request interface is implemented, this circuit has not been thoroughly tested and stressed over different operating condition. Most components currently do not rely on this feature to operate.

E.1 Command and Flag Interface

The Csr, Istat and Ien registers provide a number of single bit controls between the controller and component. Each register has a number of bits equal to the native bus width of the interface with the upper bit reserved as a master control for all other bits. Each csr bit is set by an active high pulse from the worker. The exact encoding of the bits is determined by the worker as specified in the worker configuration XML. Each bit is independently latched until a „1' is written to that bit by the controller. These clear transactions are also passed back to the worker as single-cycle pulses which can be used to perform worker specific side effects. The upper bit will always be read as „0'.

The ien register enables the selection of which bits drive the component interrupt. If a csr bit becomes set and the corresponding ien bit is set, the component interrupt line is asserted and will remain asserted until the mask or flag is cleared. The upper bit acts as a master enable for the component interrupt and can be used to quickly enable or disable interrupts without changing the selected mask pattern. This register can be read to determine its current setting.

The istat register can be used to determine the cause of the component interrupt. It acts much like the csr register, using the same worker flags to set the latches. However, unlike the csr register, istat register bits are only latched if the corresponding ien bit is set. The upper bit of the istat register is the logical or of the remaining bits. Clearing any istat bit will also clear the corresponding csr bit.

All latching behavior, as well as the single cycle pulse generation of the command bits is handled by the generated code. The worker is presented with an input vector and output vector each one bit less than the register interface width.

E.2 Worker Direct Memory Interface

The worker direct memory interface operates with a two access protocol. To write a value to the worker interface, the worker direct address register is written, followed by the worker direct data register. The generated wrapper will store the address. When the data register is written, the wrapper will initiate a worker transaction to write the specified data to the stored address. To maintain transaction ordering and support synchronization, the data register write transaction will not complete until the worker signals it has finished the requested write access. This handshake may or may not be propagated up to the host controller depending on the behavior of the platform specific property set controller.

To read a value from the worker interface, the worker direct address register is written after which the data register is read. Reading the data register will initiate a worker transaction to read the specified address. When the worker has the requested data available, it must acknowledge the transaction. This, in turn, causes the wrapper to complete the data register read access, returning the worker provided data.

To simplify accesses to regularly ordered addresses, the interface also contains a burst increment register. On each completed read or write of the data register, the signed value in the burst increment register is added to the stored address register. Using this feature, a new access can be executed without writing the address register. The same address can be accessed may times (e.g. to read a FIFO end point) by setting the burst address increment to 0. Note that read and write accesses use the same address register and either type of access will cause the address to increment. Both the address and increment registers can be read back to find their current values.

The length of each register is defined by the natural bus width of the property set interface. This has implications as to the size of the addressable region within the worker, although the worker is free to alias the addresses and pad the data as appropriate.

The worker is provided with a simple set of memory access signals consisting of an address and data field, a direction signal (1 = write, 0 = read), and a single cycle access strobe. The worker must drive the access acknowledge signal and the status data field.

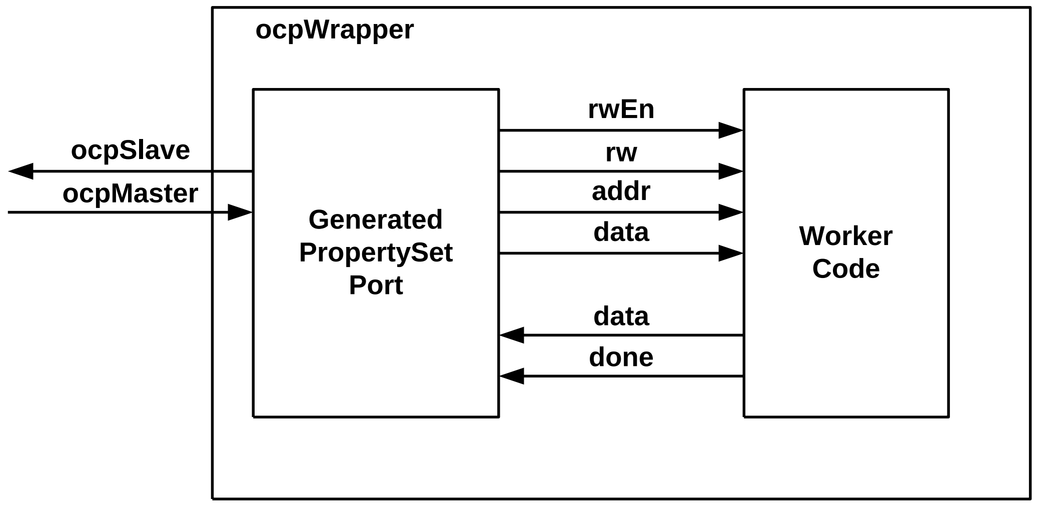

Figure E-9: Worker Direct Memory Interface

Figure E-10 and Figure E-11 show timing diagrams for the typical read and write access signals presented to the worker. The arrows marked “1” and “2” show the primary control effects of the interface. First, the rwEn field becomes asserted to mark the beginning of an access. Second, when the worker has processed the access, it asserts the done field, causing the access to end and the rwEn field to be deasserted on the next cycle. For read accesses, the worker must also drive the data field with valid data when done is asserted.

The rw, addr, and data fields of the stdParams record will be valid at least two cycles before the assertion of rwEn and will remain constant until the done field of the stStatus record is asserted. The done field is only required to pulse for one cycle, but can be held for up to three which should ease the decoding logic required in the worker. If the worker can immediately provide a response to an access, it may pre-assert the done field. In such cases, rwEn held for exactly one cycle. Data for read accesses must be supplied during that cycle. This may be reasonable for certain workers since the address will still be valid for two cycles before the pulse occurs. If the worker requires additional time to process the access, it can delay asserting done for any number of cycles. This mechanism should not be used for indefinite delays, since it is possible, depending on the system design, that all host activity will be blocked while an access is pending.

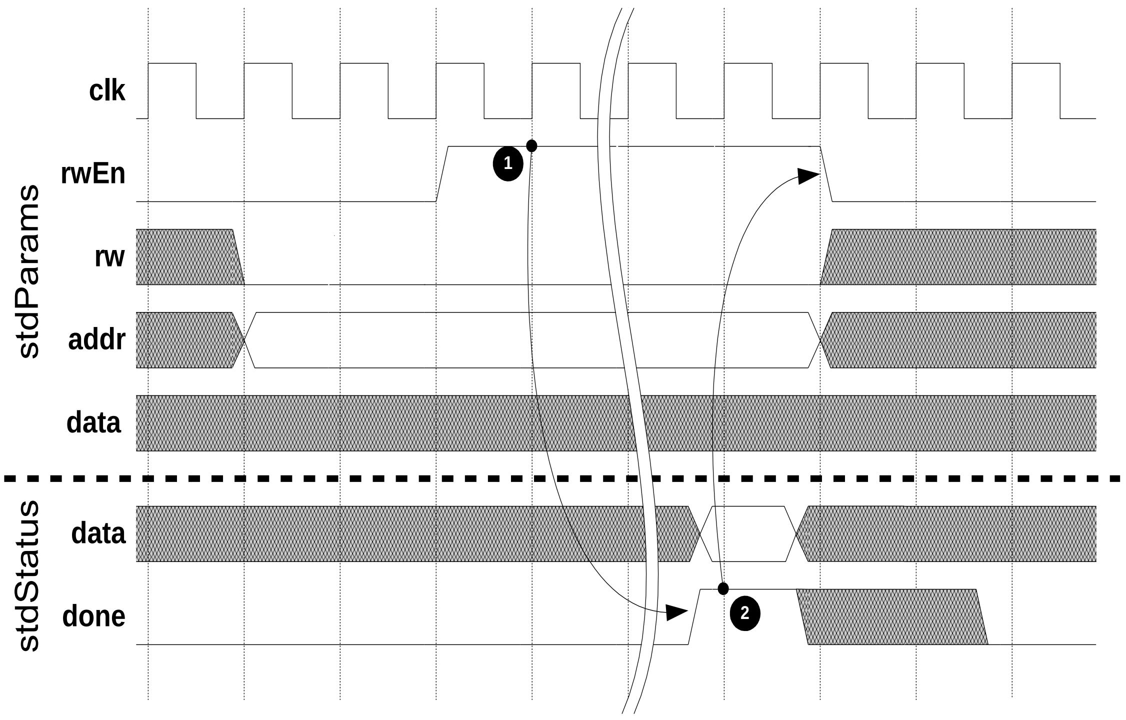

Figure E-10: Read Access Timing Diagram

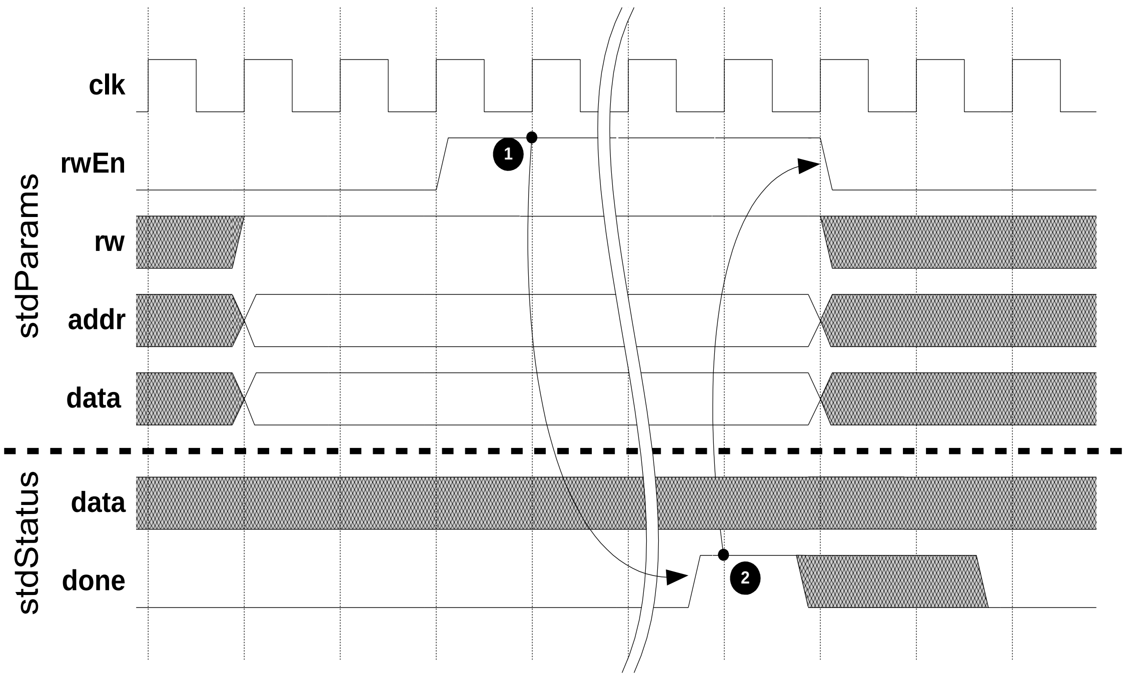

Figure E-11: Write Access Timing Diagram

E.3 Worker Custom Properties

Custom properties are parsed directly from the XML description of a component. The generated wrapper will provide storage for all writeable properties which also can be read back by the host to determine their current value. Read-only properties are provided exactly as they are fed from the worker and must be stored in the worker or its OCP wrapper if required. In general all read-only properties should be static or slowly varying since they are typically accessed from a software process. Any rapid events such as trigger pulses or pulsed error flags must be latched to ensure the host recognizes a change. Alternatively, the flag interface described in Appendix E, which was explicitly designed for this purpose, can be used.

In addition to the values of the writeable registers, the generated code provides a record with access status flags. The record contains a single bit flag for each writeable property which will pulse for one cycle each time the property is written. It also contains flags for all types of properties which pulse for one cycle when the corresponding property is read. These flags can be used by the worker to initiate internal updates or any other applicable side effect.

Acronyms

ADC	Analog to Digital Converter
ASIC	Application Specific Integrated Circuit
DAC	Digital to Analog Converter
DSP	Digital Signal Processor
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
Host UI	Host User Interface
IF-RF	Intermediate Frequency to Radio Frequency
OCP	Open Core Protocol
RF-IF	Radio Frequency to Intermediate Frequency
RTL	Register Transfer Level
SoC	System on a Chip
SWAP	Size, Weight, And Power
VHDL	Very High-level Description Language