OpenCPI Verification User Guide

Revision History

Version	Date	By	Notes
0.00	2010-11-01	Shepard Siegel	Creation
0.01	2010-11-11	Shepard Siegel	Added PMEM text
0.02	2010-11-15	Shepard Siegel	Added details and some expository
0.03	2010-11-21	Shepard Siegel	Added Protocol Monitor Composition
0.04	2010-11-30	Shepard Siegel	Switched 33b PMEM format to 32b (4B DWORD)
0.05	2010-12-06	Shepard Siegel	Process comments/nits from Jim Kulp @ Code Review
0.06	2011-01-04	Shepard Siegel	WSI Protocol Monitor format, and two-layer refactoring
0.07	2011-02-03	Shepard Siegel	Added List of test cases

1 INTRODUCTION

This document explains how industry-standard verification and test practices are used specifically in the context of OpenCPI.

1.1 PROTOCOL MONITORS

Protocol monitors may be inserted on well-known interfaces such as WIP::WCI and WIP::WSI. The resultant Protocol Monitor Event Messages (PMEMs) which are produced may be used in several ways. The protocol monitors detect and report events and errors. No additional RTL or interconnect is required in a functional simulation verification setting. In hardware verification, some means of observing the PMEMs is needed.

Beyond their fundamental use, in a conjectured embellishment, the protocol monitors may be joined together with an aggregation network. Beyond verification, this would provide utility in application debug and test as a complement to FPGA vendor-specific techniques like *ChipScope®* or *SignalTap®*.

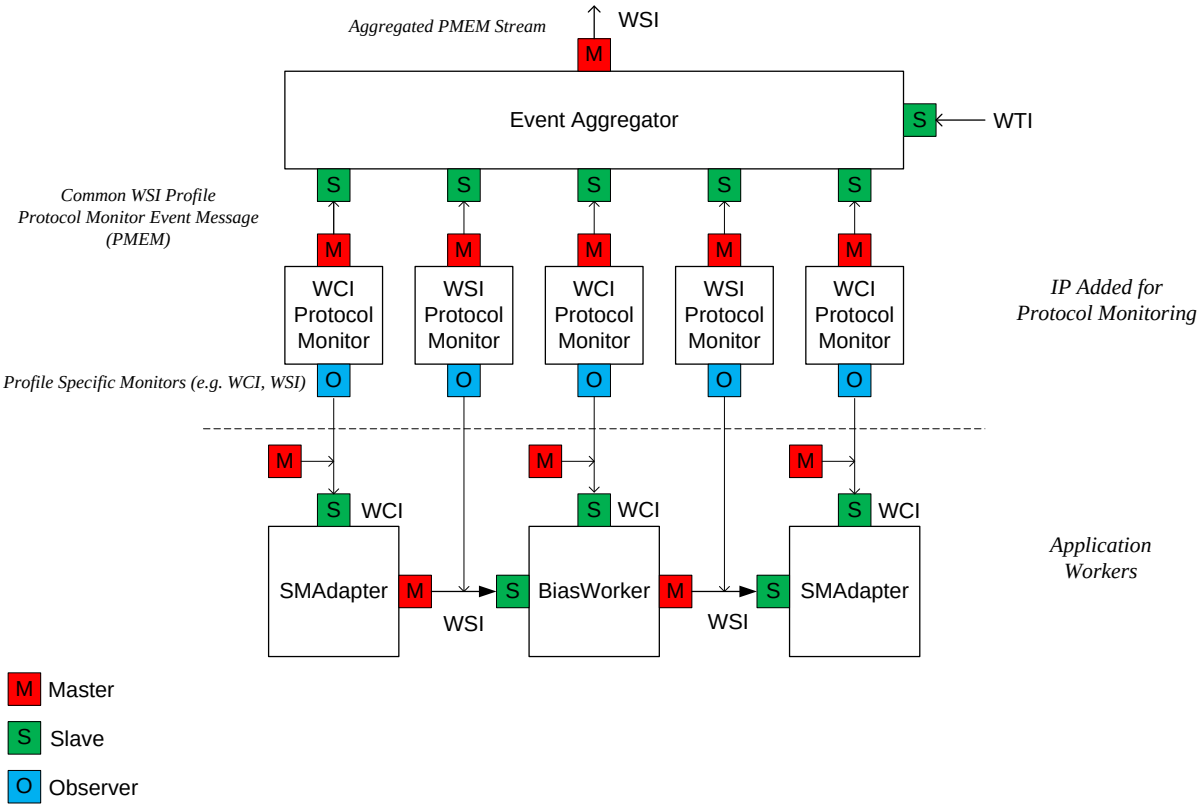


FIGURE 1- ASSEMBLY OF PROTOCOL MONITORS

2 PROTOCOL MONITORS

Protocol monitors are IP assets that may be instanced in an RTL setting and serve the purpose of passively observing a particular master-slave interface pair. By observing the interface, the protocol monitor can generate events that correspond to both normal and erroneous behavior. For example, a protocol monitor can "see" a transaction's request and response; just as it can check for and report errant behavior.

The protocol monitor does this by providing two sub-interfaces: 1) An Observer interface that passively watches the signals that are driven by both the master and the slave. 2) A message Sender interface that emits Protocol Monitor Event Messages (PMEMs). The signature of the observer interface is such that it can "listen" to the interface being monitored. The signature of the message interface is a format (described in this document) that is common across different protocol monitors. The commonality and flexibility of the PMEM format decouples a specific protocol from the apparatus used to observe events and errors. In this way, new protocol monitors can be added at a later date.

Figure 2 below shows the composition of a Protocol Monitor module and an observer and generator sub-module. While the WIP defines profile and protocol signaling; and (later in) this document the PMEM protocol; OpenCPI developers may define, design or refine Protocol Monitors implementations as they see fit. For the WCI and WSI profiles, this work has been done for them. The subsequent sections describe this protocol monitor design pattern and their specialization to specific profiles.

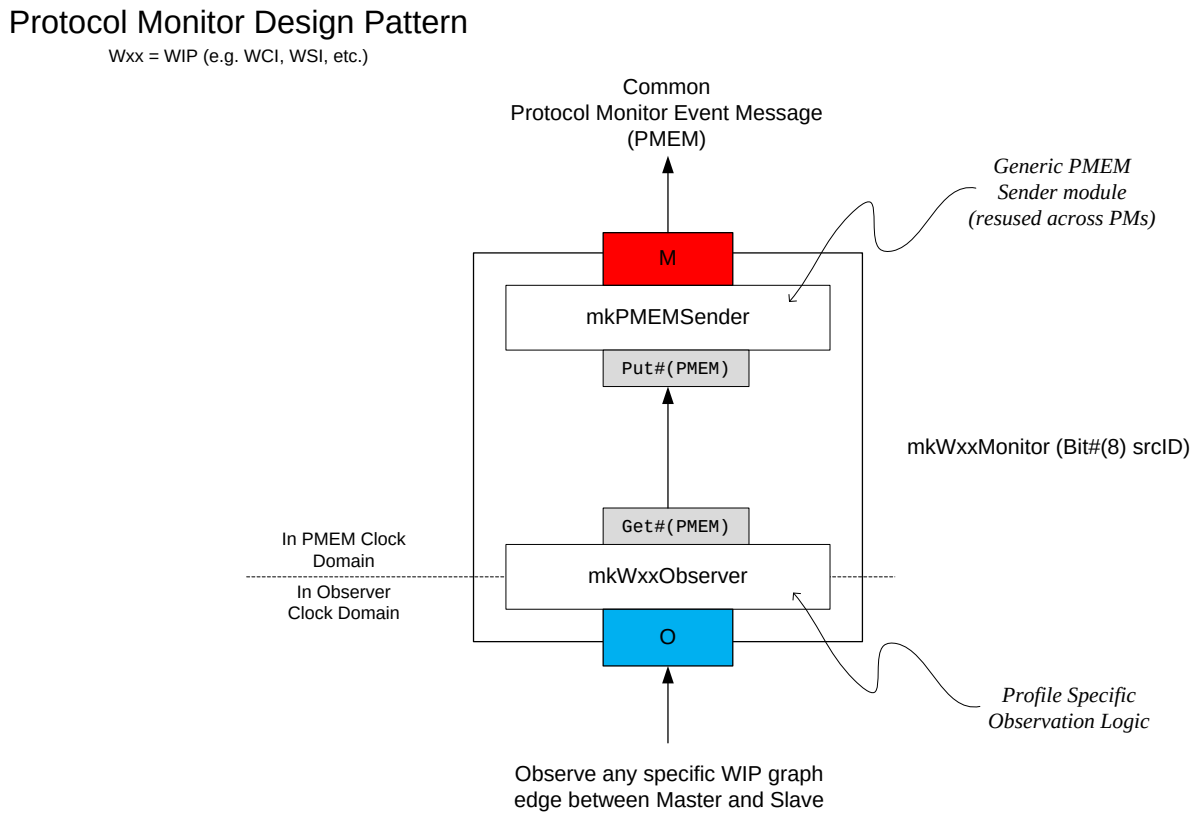


FIGURE 2 - PROTOCOL MONITOR DESIGN PATTERN

2.1 THE PROTOCOL MONITOR OBSERVER MODULE

The observer module is the component which adapts a specific WIP interface to a more-abstract sequence of observations. The WIP observer interface is fully defined by the WIP metadata. For example, an OpenCPI WIP::WCI OCP-IP interface with an instance-specific set of interface attribute metadata. This includes that interface's possibly unique clock and reset and the choices made by the attribute metadata. These are the signals that comprise the WIP observer port shown at the bottom of figure 2 prior.

Logic in the observer module watches for a set of events and errors by observing the subject interface. The output of the observer module is a sequence of observations. These "observations" are relatively low-level and often specific to the observed protocol. They are defined on a per-profile basis. The observations may be, by example: Event: Reset has been released. Event: Control request INIIALIZE. Event: Write configuration property (A) with (D). Error: Response observed without corresponding request. And so on. It is specifically *not* the intent of the observer to abstract these low-level, protocol observations into request-response-fused transactions, unless that the native nature the protocol. An observer code module (e.g. mkWxxObserver) is specialized to WIP Wxx. As such, it contains the bulk of translating observed wire-transitions to observations. In other words, wire-level to protocol-level translation.

The output of every profile-specialized observer module is a common, extendable interface Type called PMEM (note the overloaded use of the acronym; here it is a Type for an internal interface). Observer modules have event generation rules that are in the business of adapting protocol-specific *observations* made "on-the-wire" into equivalent or more-abstract *events*. Each particular implementation may decide to what degree this abstraction process occurs. It may pass or filter specific observations to the PMEM Sender. It is also allowed, but not required, that the observer module track individual low-level requests and responses, potentially abstracting multiple protocol-level events (request, response) into a single higher-level transaction-level (TL) event.

2.2 THE PROTOCOL MONITOR MONITOR MODULE

The monitor module is the component that encapsulates both the observer and sender modules. It is the component instantiated when one creates an instance of a protocol monitor. Every monitor instance is given a unique compile-time Bit#(8) srcID, and this static identifier is inserted in the event stream as events are enqueued to the sender module. The translation of observations to events is shown by the logic cloud in the center of figure 2 prior. A monitor code module (e.g. mkWxxMonitor) provides the Wxx base class that the observer inherits; but also may use this information for profile agnostic abstraction.

2.3 THE PROTOCOL MONITOR SENDER MODULE

The PMEM sender module is the component which sends Protocol Monitor Event Messages out in a ubiquitous 4 Byte WSI format. It accepts events from the monitor module, which include an instance-specific Bit#(8) eType (event type), and optionally, additional event data, such as address and data. The sender module contains the convenience, reusable code, for generating a PMEM from an event. The PMEM sender respects the PMEM message format defined in this document. The PMEM generator is shown atop figure 2 prior. A generic PMEM generator (e.g. mkPMEMSender) may be reused across protocol monitors.

3 PROTOCOL MONITOR EVENT MESSAGE (PMEM)

The PMEM is communicated with a fixed WSI profile and the message carried in a fixed format as described here.

3.1 PMEM OVERVIEW

The PMEM format facilitates the efficient communication and aggregation of event messages arising from a diverse set of event sources. Every PMEM carries a unique source ID and tag with it that originates from its source and is carried along with the message as it moves through toward its destination. An external (e.g. software) association list links a unique source ID to the type of message provided. In this way, future message formats may be invented without needing to change the PMEM infrastructure. Event messages may be as short as a single header word. The header carries a monitor-specific event-type (eType) indication that serves to abstract the event. The body contains event information; such as the address and data words for a write request or the response words for read response. In conditions of network congestion due to excessive body payload, the PMEM format facilitates the truncation of body information, while retaining the header. This capability allows PMEM devices to dynamically thin body data; as an alternative to dropping an event altogether.

3.2 PMEM WSI ATTRIBUTES

The non-default attributes are shaded in yellow.

Attribute	Type	Value	Description
DataValueWidth	unsignedInt	32	1 4B DWORD
DataValueGranularity	unsignedInt	1	1 4B DWORD per “value”
DiverseDataSizes	boolean	False	
MaxMessageValues	unsignedInt	256	256 4B DWORDs (see PMEM format)
NumberOfOpcodes	unsignedInt	1	
Producer	Boolean		Protocol Monitor out is True; Event Aggregator in is False
VariableMessageLength	Boolean	True	
ZeroMessageLength	Boolean	False	
Continuous	Boolean	True	No bubbles allowed in message source (see PMEM format)
DataWidth	unsignedInt	32	1 4B DWORD
ByteWidth	unsignedInt	32	
ImpreciseBurst	Boolean	True	(see PMEM format)
PreciseBurst	Boolean	False	
Abortable	Boolean	False	
EarlyRequest	Boolean	False	

3.3 PMEM FORMAT

Each PMEM is a comprised of exactly one PMEM header word followed by 0-255 immediately following PMEM body words.

3.3.1 PMEM HEADER

The PMEM header word is comprised of the following fields:

Name	Bits	Description
srcID	[31:24]	Source ID: 8b identifier of protocol monitor
eType	[23:16]	Event Type: 8b of srcID-specific event classification
srcTag	[15:8]	Source Tag: 8b rolling count of source event messages
length	[7:0]	Length: Length in 4B DWORDs of message, including header. 8'h00 = 256 4B DWORDs

The "srcID" field must be populated with a unique code to differentiate a particular protocol monitor instance among all instances to be aggregated.

The "eType" field provides an efficient, in-header method for any Class::Instance of protocol monitor to provide 8b of information within the header. The definition of the eType field is specific to the design of each protocol monitor. For example, the eType field may be used to discriminate between event and error; or may be used to describe what kind of payload, if any, will follow in the message body. The implementation of any Class of protocol monitor may define the meaning of eType as they see fit.

The srcTag field must be an 8b rolling count that is generated by each protocol monitor instance. The first message emitted after reset must be message 8'h00.

The length field of the header must be populated with the length of the PMEM in 4B DWORDs. A header with no payload would be 8'h01. The special case of a header and 255 4B DWORD payload would be 8'h00 (256 4B DWORDs).

3.3.2 PMEM BODY

The PMEM body may have 0-255 4B DWORDs of data. The contents of the body and their format(s) are determined by the specific protocol monitor and the eType in the header. The type of protocol monitor, the meaning of eType, and the format of the body can be looked-up from the code that assigned the srcID.

4 PROTOCOL MONITOR EVENT ENCODING

The protocol monitor observes a WIP port and generates the following PMEMs in response to the following events and errors similar to what is in the table below:

Event Name	eType	Payload Following	Event Description
NONE	8'h00	0	
UNRESET	8'h01	0	Reset is de-asserted on the observed WCI Link
RESET	8'h02	0	Reset is asserted on the observed WCI Link
UNATTENTION	8'h03	0	Slave has de-asserted Attention
ATTENTION	8'h04	0	Slave has asserted Attention
UNTERMINATE	8'h05	0	Master has de-asserted Control Op Terminate
TERMINATE	8'h06	0	Master has asserted Control Op Terminate
TIMEOUT	8'h07	0	No response after timeout interval
INITIALIAZE	8'h08	0	Initialize Control Op Observed
START	8'h09	0	Start Control Op Observed
STOP	8'h0A	0	Stop Control Op Observed
RELEASE	8'h0B	0	Release Control Op Observed
TEST	8'h0C	0	Test Control Op Observed
BEFORE_QUERY	8'h0D	0	BeforeQuery Control Op Observed
AFTER_CONFIG	8'h0E	0	AfterConfig Control Op Observed
WRITE_REQUEST	8'h1x	2 (A,WD)	Configuration Write Request (Address, Write Data) (BE in nibble x)
READ_REQUEST	8'h20	1 (A)	Configuration Read Request (Address)
WRITE_RESPONSE	8'h30	1 (D)	Configuration Write Response (Data)
READ_RESPONSE	8'h40	1 (D)	Configuration Read Response (Data)
REQUEST_ERROR	8'h8y	0	Protocol Request Error (extra info in nibble y)
RESPONSE_ERROR	8'h9z	0	Protocol Response Error (extra into in nibble z)
XACTION_ERROR	8'hAt	0	Transaction Error (extra into in nibble t)

This information is contained in the BSV source code located at `$OCPI/bsv/wip/OCPMDefs.bsv` , which serves as the specification for encoding. Protocol monitors can be implemented in any language by appropriately exporting the typedef PMEvent from this file.

5 WORKER VERIFICATION TEST CASES

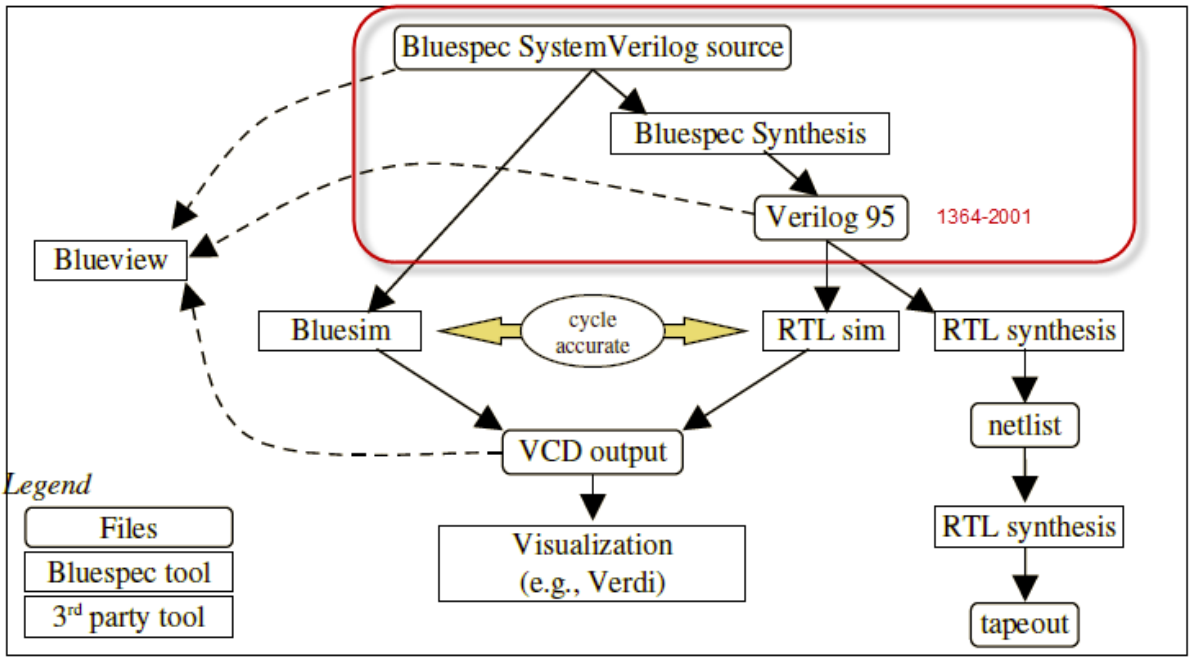
This section details the test cases provided that are used to verify a worker's function and performance. In this context, the worker is sometimes referred to as the Device-Under-Test (DUT).

A universe of methodologies exists for verification of RTL Workers. This section of the verification user doc describes one possible approach among many. It is specifically our intent not to dictate a particular methodology; but to document and demonstrate our process.

The practice described herein is based in concepts in the text "*Writing Testbenches using SystemVerilog*" by Janick Bergeron (2006, Springer). This text followed two editions of an earlier text called "*Writing Testbenches, Functional Verification of HDL Models*", by Janick Bergeron (2000, 2003, KAP). Where the first two editions provided examples with implementation languages of VHDL and Verilog; the latest edition uses SystemVerilog.

With OpenCPI, we have tried to be as agnostic to implementation language as possible; and instead focus on the interaction patterns that are agnostic to the language. One common denominator between languages we use is IEEE Verilog 1364-2001. Because of its ubiquity, we refer to this just as "RTL".

The diagram below, taken from the document $OCPI/doc/bsv-verification-guide.pdf, shows how still another implementation language, Bluespec SystemVerilog (BSV) is compiled to RTL.



Technologies such "C to Gates", "MATLAB to Gates" and "OpenCL to Gates" will mature, and almost certainly have a path for their compiler output to VHDL or Verilog RTL.

5.1 SPECIFIC TEST CASES

The table below shows at a glance the attributes of specific test cases. The source code to a particular test cases is given as TB<x> and can be found in `$OCPI/bsv/tst/TB<x>.bsv` .

ID	Worker	Notes
1	BiasWorker	Procedural source/sink in TB1; with Protocol Monitors in TB11
2	DelayWorker	Procedural source/sink in TB9
3	FFTWorker	Array source, Procedural Sink in TB10

6 REFERENCE DOCUMENTS

TABLE 1 - LIST OF REFERENCE DOCUMENTS

ID	Standard	Link
1	OCP 3.0 Specification	http://www.ocpip.org
2	OpenCPI WIP Specification	http://www.opencpi.org
3	ARM AMBA 4.0 AXI-4 Protocol Specification	
4	IEEE 1364-2001 – "Verilog 2001"	

TABLE 2 - LIST OF ABBREVIATIONS

AFRL	Air Force Research Lab
ALT	Altera
AR	Atomic Rules
AXI	Advanced Extendable Interconnect
BSV	Bluespec SystemVerilog
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GIT	A distributed version control system
HDL	Hardware Description Language (VHDL or Verilog)
IP	Intellectual Property (core, worker)
MFS	Mercury Federal Systems
OCP	Open Core Protocol
OCDP	OpenCPI Data Plane
OCPI	Open Component Portability Infrastructure
POP	Period of Performance
PCI	Peripheral Component Interconnect
PCIe	PCI Express
SOW	Statement of Work
TRL	Technology Readiness Level
WCI	Worker Control Interface
WIP	Worker Interface Profiles
WSI	Worker Streaming Interface
XIL	Xilinx