Rapid Prototyping of
Application Specific Signal Processors

RASSP

Process - Architecture - Framework

Rapid Prototyping of Application–
Specific Signal Processors (RASSP)

RASSP Model Year Architecture Specification Volume III

Standard Virtual Interface Specification

Version 1.0

September 23, 1996

ADVANCED TECHNOLOGY LABORATORIES

LOCKHEED MARTIN

Table of Contents

1. INTRODUCTION 1
1.1. Purpose 1
1.2. Scope 1
1.3. Document Outline 1

2. OVERVIEW AND CONCEPTS 2

3. SVI SPECIFICATION 5
3.1. Signal Definition 5
3.1.1. Data Interface 5
3.1.2. Interrupts 9
3.1.3. System Clock 9
3.1.4. System Reset 9
3.2. Protocol and Timing 10
3.2.1. SVI Command Definition 10
3.2.2. Command Usage and Message Composition 12
3.2.3. Data Transfer Control 12
3.2.3.1. Write Transactions 13
3.2.3.2. Read Transactions 14
3.2.3.3. Other SVI Operations 14
3.2.3.3.1. Errors 15
3.2.3.3.2. Aborts 16

4. ENCAPSULATION GUIDELINES 18
4.1. Encapsulation Structure 18
4.1.1. Internal Node Encapsulation Structure 20
4.1.2. Interconnect Fabric Encapsulation Structure 21
4.1.3. Interconnect Fabric with COTS Controller Encapsulation Structure 23
4.2. General Considerations 24
4.3. Application Notes 24
4.4. Synthesis Guidelines 25
4.4.1. Model Hierarchy 25
4.4.2. Combinational vs. Sequential Processes 26
4.4.3. Sequential Coding Specifics 26
4.4.4. Miscellaneous: Unsupported VHDL Constructs 26

List of Figures

Figure 1 MYA Functional Architecture and Reuse Elements 3
Figure 2 Standard Virtual Interface Approach 4
Figure 3 Standard Virtual Interface Signal Definition 6
Figure 4 Example of SVI Write Operation 12
Figure 5 Example of SVI Read Operation 15
Figure 6 Example of SVI Write Operation with Abort cycle. 17
Figure 7 Internal Node Encapsulation Structure 20
Figure 8 Fabric Interface Encapsulation Structure 22
Figure 9 Fabric Interface Encapsulation With COTS Interface Controller 23
Figure 10 Methods for Converting Outputs to a Single Tri–state Driver 27

List of Tables

Table 1 SVI Commands 10
Table 2 SVI Error Codes 15
Table 3 SVI Abort Codes 16

1. INTRODUCTION

1.1. Purpose

The purpose of this document is to convey a formal specification for the **Standard Virtual Interface (SVI)**, a technology–independent functional interface which facilitates hardware upgrades and technology insertion at the architectural level. This specification should be used when designing and modeling **architectural reuse library elements** to ensure interoperability among those various library elements. The use of the SVI is not applicable to board or module level COTS products. Such products are expected to be based on an industry–accepted non–proprietary open interface standard which allows upgrades and technology insertion with respect to that standard. The SVI provides an additional level of interoperability and flexibility for upgrades and technology insertion in cases where designs, while making use of COTS components at the chip level, are under control of the RASSP user community. Refer to the *RASSP Model Year Architecture Specification, Vols. I & II* for more information about Model Year Architecture concepts and applications of the SVI. The definition of the SVI presented in this specification supercedes that which was presented in the *RASSP Model Year Architecture Working Document*.

1.2. Scope

The scope of the SVI specification is defined to include the following:

– Signal / Functional Timing / Protocol Definition

– Concise definition of the signals, functional timing, and protocol of the SVI.

– Encapsulation Templates with Examples

– VHDL templates for SVI encapsulations resulting from the efforts expended on the architecture verification task. These are provided to supplement the signal, timing, and protocol definition described above for users of SVI, and are contained as SVI appendices in *MYA Specification Vol. IV.* These appendices also contain specific VHDL encapsulations done on the verification task which are based on these templates.

– Encapsulation Guidelines

– Library element design guidelines to ensure that encapsulations are interoperable and synthesizable.

1.3. Document Outline

This document is comprised of two parts; the specification proper (MYA Spec. Vol. III), and the appendices (MYA Spec. Vol. IV). Section 2. of this volume presents an SVI overview, while

Section 3. contains the actual definition of the SVI: signal definition (Section 3.1.), timing and protocol definition (Section 3.2.), and encapsulation guidelines (Section 4.). Appendix I presents the SVI types package. Appendix II presents the SVI Encapsulation Template, while Appendix III provides encapsulation examples which can be used for reference in designing new SVI encapsulations. Appendix IV contains SVI data–path width converters. The purpose of the material provided in the appendices will be explained in this Volume.

2. OVERVIEW AND CONCEPTS

The SVI enables the interoperability and upgradability of architectural level reuse library elements by defining an interface protocol and implementation approach for reuse element encapsulation. Encapsulated library elements (encapsulations) are categorized into three general types of reuse element: Internal Nodes, Fabric Interfaces, and RNI (Reconfigurable Network Interface) Bridge Elements. An internal node is essentially any architectural–level element which becomes connected (through a fabric interface) to the system interconnect fabric. Examples of internal nodes include signal processors, vector processors, or shared memory. A fabric interface is an element which translates between the native communication protocol of an internal node or RNI element, and the protocol of the interconnect fabric. The term “interconnect fabric” is used to describe any form of node–to–node communication medium. Examples of interconnect fabrics include crossbar–based point–to–point interconnect networks, rings, and multidrop buses. In Model Year Architecture nomenclature, the joining of an internal node and a fabric interface results in an Internal Module; an architecture–level element as it appears at the system level. Finally, an RNI bridge element is a specialized SVI–to–SVI bridge which sits between two fabric interface encapsulations, resulting in a fabric–to–fabric link which together is called an RNI. Figure 1 illustrates these Model Year Architecture reuse elements, and how they interface with the SVI in the MYA functional architecture.

Figure 2 illustrates the concept of the SVI. Each library element, in this case a PE and a COTS interface controller, includes an encapsulation wrapper which implements the SVI. The wrappers are described in VHDL code, and are written in Register Transfer Level (RTL) style which enables the logic described by the code to be synthesized into a targeted PLD, FPGA, or ASIC device. During the synthesis process, the wrappers from adjoining encapsulated elements (PE and fabric interface, for example) are combined to create the PE-to-controller interface device. Note that as an alternative to this example, the COTS controller could be replaced by custom fabric interface logic, also described in VHDL code. In this alternative case, the wrappers as well as the interface logic VHDL would be combined and synthesized into a single interface device, instead of the two devices (COTS and wrappers) shown in Figure 2 .

The presence of the additional hardware layer imposed by the SVI will result in some overhead in the resulting hardware implementation. As part of the hardware synthesis process however, the logic described in both encapsulations’ SVI wrappers are combined and optimized, resulting in some of the incurred overhead being reduced. This process will likely cause some of the signals defined for the SVI to become implicit in the resulting logic. What remains of the SVI will be embedded within the logic hardware device, and will not appear as explicit pins on a chip or interface connector. These characteristics give rise to the use of the term *virtual* in SVI. Electrically reconfi-

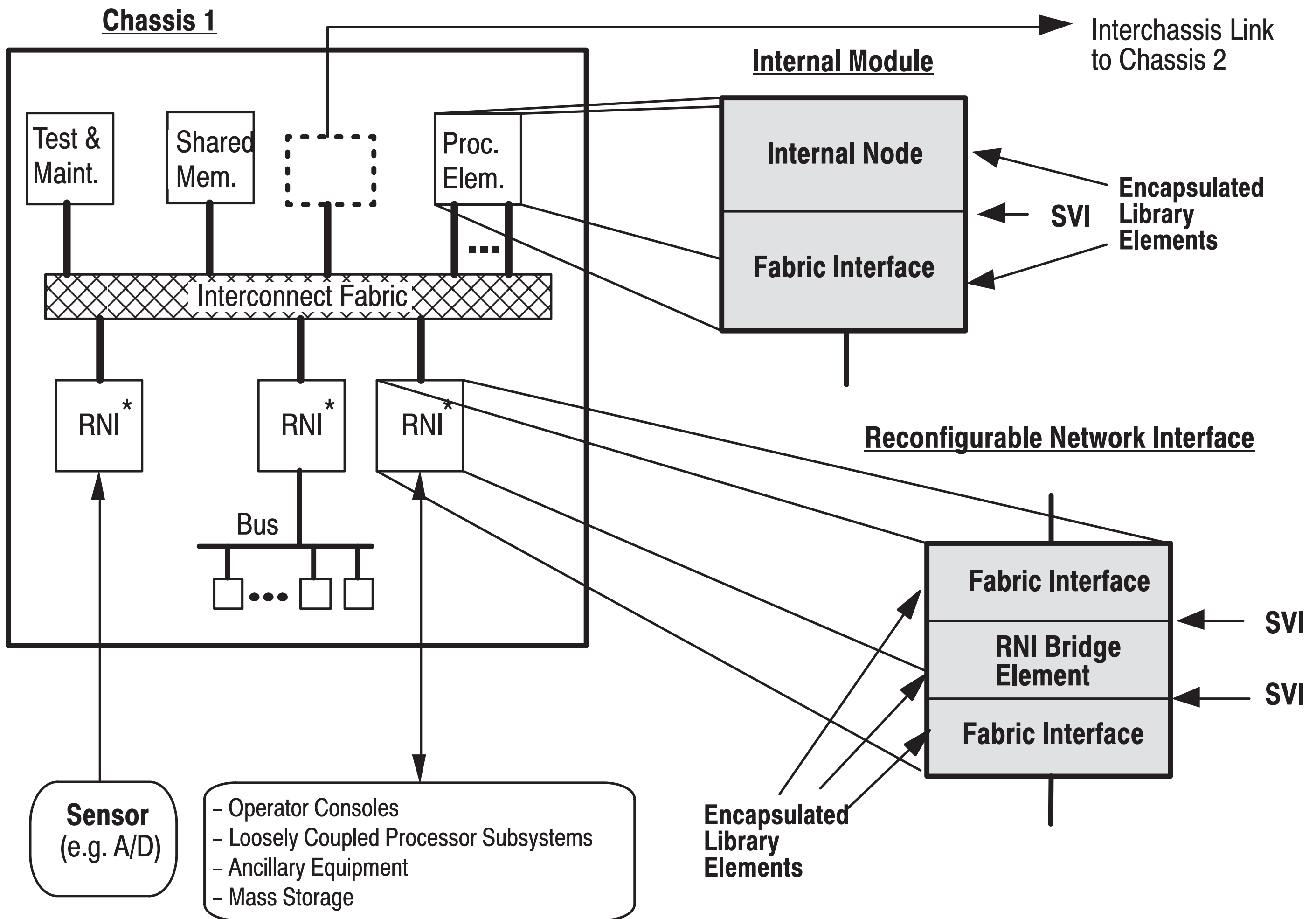

Figure 1 High Level View of MYA Functional Architecture and Reuse Elements

gurable components, such as SRAM–based FPGAs, are particularly suited to implement SVI encapsulation logic, allowing easy design upgrades.

Not every library element will support every feature of the SVI. For example, an interface that has only one physical or virtual channel into an interconnect network does not require the Channel ID (refer to section 3.1.1.), which can therefore be left unused. The SVI is defined such that any internal module may be interfaced to any fabric interface even if only a subset of the functionality is supported for the pairing of two particular elements.

The SVI definition is designed to be general enough to handle different interprocessor communication paradigms. Some interconnect networks support a message passing paradigm to interprocessor communication, while others support a global shared memory paradigm. In some cases there is synchronous operation between the internal node and the interconnect fabric, while in other cases the internal node and the interconnect fabric operate asynchronously. The SVI is by definition synchronous; that is, each word of SVI data is transferred synchronously to the SVI clock. Support

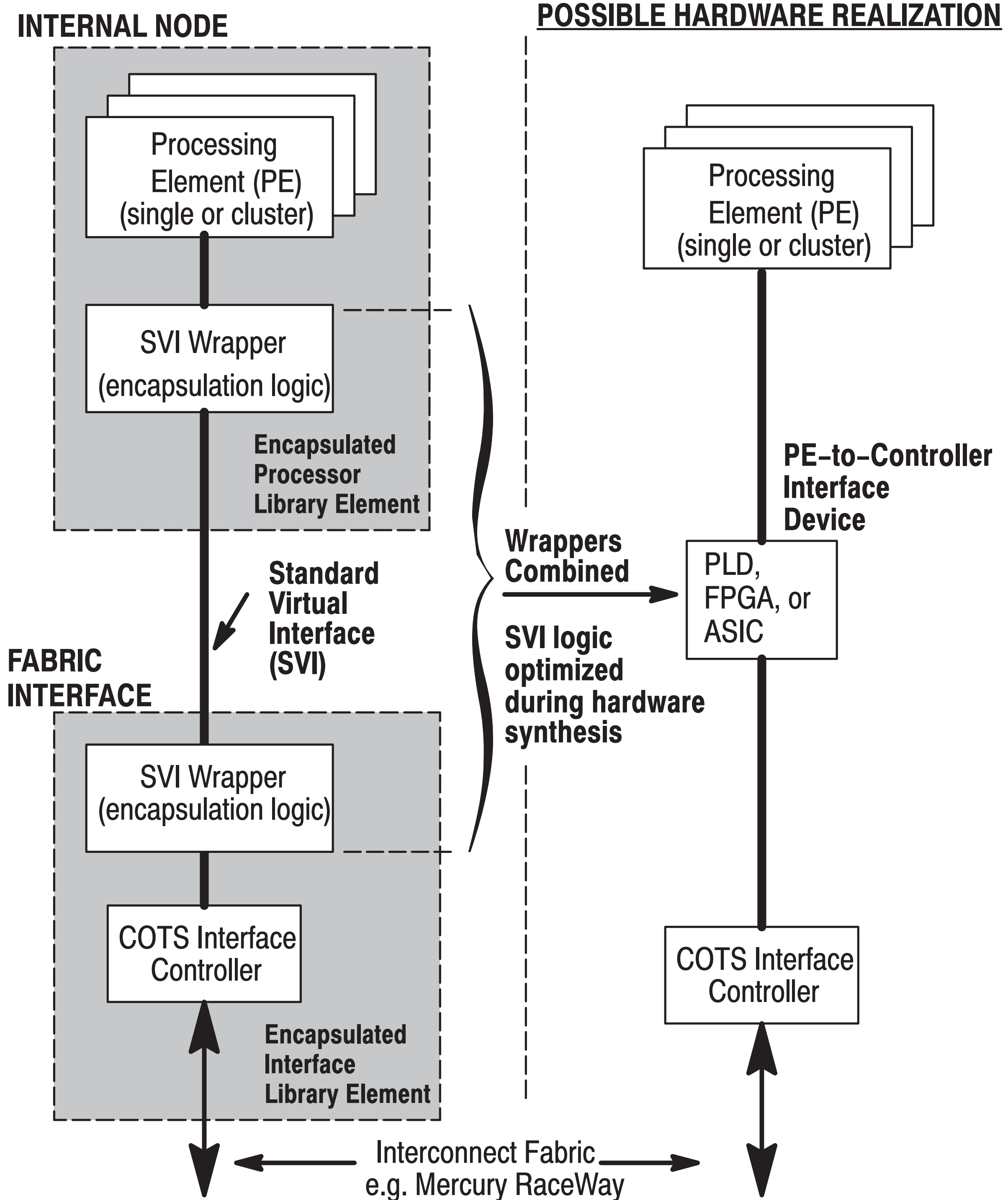

Figure 2 Standard Virtual Interface Approach using Internal Node–to–Fabric Interface example.

for asynchronous operation between an internal node and the interconnect fabric can be easily handled by the SVI encapsulation logic.

3. SVI SPECIFICATION

3.1. Signal Definition

The SVI signals are are organized into the following three interface partitions:

1) *Data Output Interface*

2) *Data Input Interface*

3) *System Feed–Thru Interface (system interrupts, clocks, and resets)*

The data path portion of the SVI is partitioned into two unidirectional data interfaces: the Data Input Interface which receives incoming SVI messages, and the Data Output Interface which transmits outgoing SVI messages. The data interfaces are implemented with a master/slave pair: the SVI master is a data source to the SVI Data Output Interface, while the SVI slave is a data sink to the Data Input Interface. The encapsulation of a typical architectural element therefore contains an SVI master/slave pair. The advantage of partitioning the data path in this way is that it supports interface architectures with independent, concurrent data passing in both directions across the SVI. A bidirectional data interface can be obtained by using both unidirectional data interfaces controlled by the SVI master and the SVI slave. The System Feed–Thru Interface provides a mechanism to feed–through essential, "real–time," system–level signals that can not be conveniently transported across the data path portion of the SVI.

The widths of all SVI signals and the values of enumerated signals, such as errors and aborts, are all defined as constants in the VHDL file "svi_types_pkg.vhd." This file is included in Appendix I of Volume IV. These constants will be referred to in the signal descriptions below. Figure 3 illustrates the configuration of a typical encapsulation, showing the unencapsulated architectural element, the SVI wrapper containing the complete SVI logic, and the SVI signals which make up the data and system feed–thru interfaces. The interconnection of two SVI encapsulations is simply a matter of connecting the mating portions of each encapsulation's SVI signals: the output interface of each encapsulation to the input interface of the opposing encapsulation, and the feed–thru inputs to their corresponding feed–thru outputs.

The following sections describe the signals defined for the SVI.

3.1.1. Data Interface

The Data Interface is the portion of the SVI which handles data transfers between two encapsulated elements and consists of both a Data Output Interface and a Data Input Interface. The Data Output Interface signals and the Data Input Interface signals are defined identically, so the following

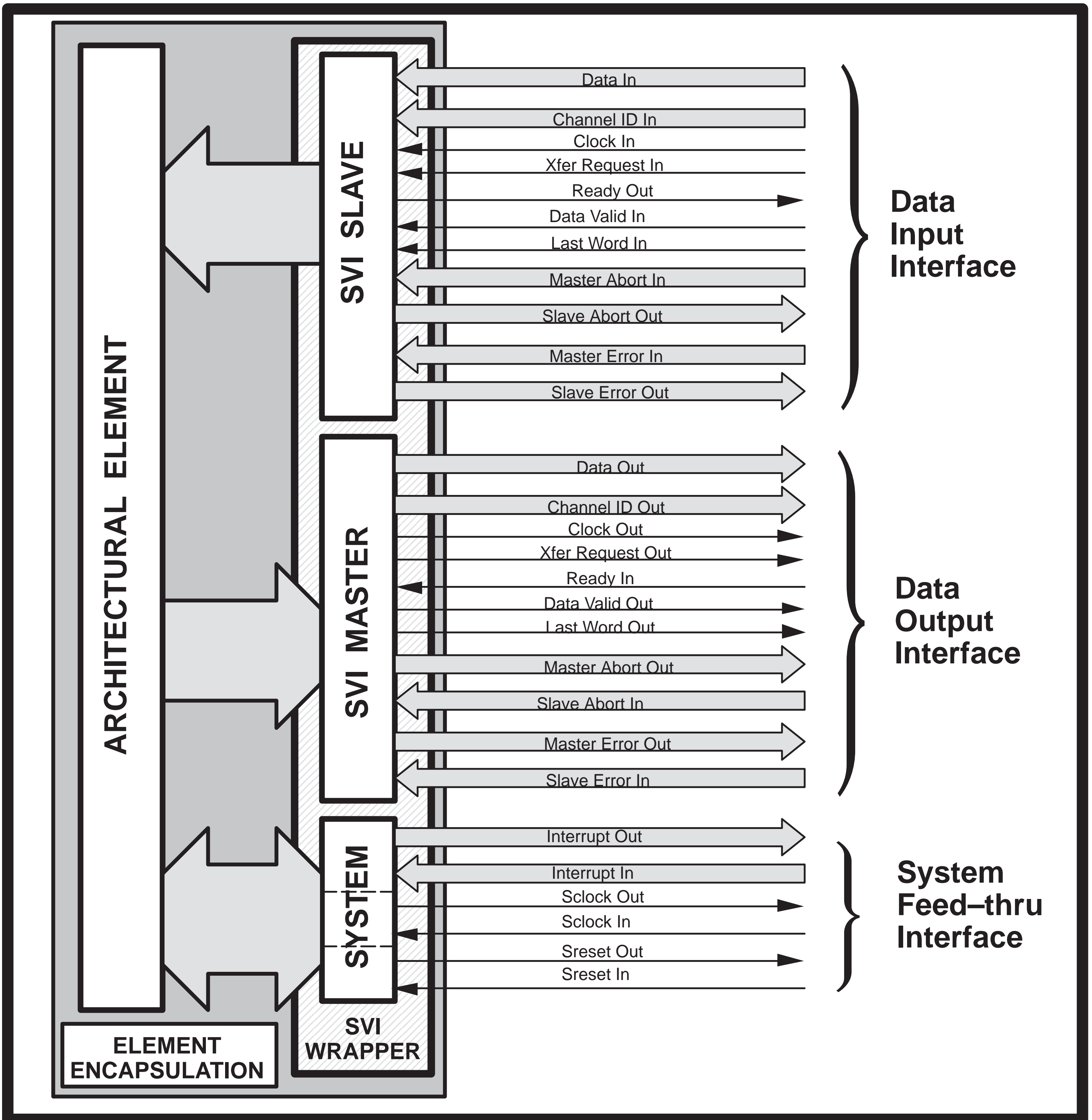

Figure 3 Standard Virtual Interface Signal Definition

signal descriptions apply to both interfaces. All of the signals described below, other than **Clock** and **Xfer Request**, are only valid when the signal **Xfer Request** is asserted. Each Data Output Interface and each Data Input Interface shall contain: **Data**, **Clock**, **Xfer Request**, **Ready**, and **Data Valid**. The remaining interface signals, **Channel ID**, **Slave Abort**, **Master Abort**, **Slave Error**, and **Master Error,** may appear in any combination as needed by the particular encapsulation.

1) **Data**

A unidirectional parallel bus used to transfer data, address, header, or other information from a master to a slave. The **Data** bus width is defined by the encapsulation designer, and is required to be in byte increments. Different **Data** bus widths will be resolved by the inclusion of width converter components which will be available from the Model Year Architecture reuse library. This will be discussed more in section 4. **Data** is valid only when **Xfer Request** and **Data Valid** are asserted and is sampled at the rising edge of **Clock**.

2) **Channel ID**

The master specifies the communication channel to be written to by the slave device via the **Channel ID**. **Channel ID** is needed when there is more than one channel (physical or virtual) defined within an interface. **Channel ID** could be used to identify a specific FIFO if there are multiple FIFOs. For example, there may be FIFO(s) dedicated to short, high–priority messages. **Channel ID** is valid only when **Xfer Request** and **Data Valid** are asserted and must maintain the same value for any given message. The width of **Channel ID** is specified by the constant channel_id_width in the VHDL file ”svi_types_pkg.vhd.”

3) **Clock**

Clock is used to synchronize the passing of data across the SVI from a master to a slave. There is no requirement on this clock’s frequency.

4) **Xfer Request**

Asserted by a master when it wishes to send a message across the SVI interface. Asserted until the last data word is transferred by the master or the message is aborted.

5) **Ready**

Asserted by a slave when it can accept data on **Data**. The slave asserts/deasserts **Ready** to modulate the rate of data transmission. The master will output data on the data interface, if available from the encapsulated element, in response to seeing **Ready** asserted on the rising edge of **Clock**. The slave must be able to accept a data word from the master on the rising clock edge immediately following the cycle in which **Ready** was asserted.

6) **Data Valid**

Asserted by a master when there is valid data on **Data**. The master asserts/deas-

serts **Data Valid** to modulate the rate of data transmission. The slave will examine **Data Valid** at the rising edge of **Clock** to determine whether or not **Data** should be sampled.

7) **Last Word**

Asserted by a master during the last data transfer cycle while **Data Valid** is asserted. The slave will examine this signal at the rising edge of **Clock** to determine if the current **Data** word is the last data word in the message. This signal does not have to be asserted during the last data transfer of an SVI message if the message is terminated due to an abort.

8) **Slave Abort**

Slave Abort is asserted by the slave when it wishes to abort an operation in progress. The value of **Slave_Abort** indicates to the opposing master the reason for the abort, which may assist the encapsulated element on the master side in determining how to respond after the abort; a null value is normally asserted. **Slave Abort** is sampled at every rising clock edge during a message transaction. The width of **Slave Abort** is specified by the constant abort_width in the VHDL file ”svi_types_pkg.vhd;” this file also contains the enumeration type values for abort signals. **Slave Abort** is left unused in those encapsulations where it is not needed.

9) **Master Abort**

Master Abort is asserted by the master when it wishes to abort an operation in progress. The value of **Master Abort** indicates to the opposing slave the reason for the abort, which may assist the encapsulated element on the slave side in determining how to respond after the abort; a null value is normally asserted. **Master Abort** is sampled at every rising clock edge during a message transaction. The width of **Master Abort** is specified by the constant abort_width in the VHDL file ”svi_types_pkg.vhd;” this file also contains the enumeration type values for abort signals. **Master Abort** is left unused in those encapsulations where it is not needed.

10) **Slave Error**

Slave Error is asserted by the slave to indicate it has detected an error (such as a parity or transmission error). The value of **Slave Error** indicates to the opposing master the type of error that was encountered, which may assist the encapsulated element on the master side in determining how to respond to the error condition; a null value is normally asserted. **Slave Error** is sampled at every rising clock edge during a message transaction. The width of **Slave Error** is specified by the constant error_width in the VHDL file ”svi_types_pkg.vhd;” this file also contains the enumeration type values for error signals. **Slave Error** is left unused in those encapsulations where it is not needed.

11) **Master Error**

Master Error is asserted by the master to indicate it has detected an error (such as a parity or transmission error). The value of **Master Error** indicates to the opposing slave the type of error that was encountered, which may assist the encapsulated element on the slave side in determining how to respond to the error condition; a null value is normally asserted. **Master Error** is sampled at every rising clock edge during a message transaction. The width of **Master Error** is specified by the constant error_width in the VHDL file ”svi_types_pkg.vhd;” this file also contains the enumeration type values for error signals. **Master Error** is left unused in those encapsulations where it is not needed.

3.1.2. Interrupts

Interrupt In/Out

The SVI allows for encapsulation elements to pass through system interrupt signals. Each encapsulation may have one input interrupt port and/or one output interrupt port. The width of **Interrupt In/Out** is specified by the constant interrupt_width in the VHDL file ”svi_types_pkg.vhd.” Interrupt acknowledge signals are to be transferred through the SVI data interfaces, using the “interrupt acknowledge” SVI command. **Interrupt In** and/or **Interrupt Out** are left unused in those encapsulations where interrupts are not needed.

3.1.3. System Clocks

Sclock In/Out

The SVI allows for encapsulation elements to pass through system clocks. Each encapsulation may have one System Clock input and/or one System Clock output. Note that **Sclock In/Out** is not necessarily the same as the clock used for data transfers. The **Clock** signal defined in Section 3.1.1. may be derived from the System Clock. For test reasons, it is preferred that all clocks be derived from **Sclock In/Out** or locked to a single source and that the provisions for control and observation of the System Clock be provided at each transition/interface point. **Sclock In** and/or **Sclock Out** are left unused in those encapsulations where **Clock** is sufficient.

3.1.4. System Reset

Sreset In/Out

The SVI allows for encapsulation elements to pass through system resets. Each encapsulation may have one System Reset input and/or one System Reset output. **Sreset In** and/or **Sreset Out** are left unused in those encapsulations where resets are not needed.

3.2. Protocol and Timing

The SVI was designed to accommodate many different processing elements, interconnects, and interface standards. In order to achieve this accommodation, the SVI definition was required to be as flexible as possible, while minimally impacting the performance and hardware characteristics of a RASSP system. Therefore, the SVI is defined as a very simple FIFO or I/O type of interface that uses simple message passing to transfer data. The protocol and functional timing will now be discussed. The SVI specification contains no absolute timing information because it is a technology independent interface. Absolute timing requirements will be established and adhered to in the implementation (synthesis) process.

3.2.1. SVI Command Definition

The first data word of every SVI message shall contain an SVI command. All current and future SVI commands shall be of a width of one byte, and shall occupy the least significant byte of the data bus, assuring that the SVI command will always be transferred in a single word transfer. Any other bits of the first data word are "don't care." The SVI command establishes the context of the data in the accompanying message, and provides information which enables the receiving SVI interface to properly interpret, and if necessary, functionally multiplex the data. The message data is contained in the data stream following the command, and may contain any combination of header(s), address, and trailer(s), as required by the encapsulation. The existing commands are shown in Table 1 .

SVI Data(7 DOWNTO 0)	Command Type
00000001	External Write Request
00000010	Internal Write Request
00000011	External Read Request
00000100	Internal Read Request
00000101	External Read Response
00000110	Internal Read Response
00000111	Read–Modify–Write Request
00001000	External Read Response – Split Read Request
00001001	Response to Split Read Request
00001010	Stop Read Request

Table 1 SVI Commands

This list contains commands needed to implement SVI encapsulations generated in the Model Year Architecture Verification Task. Additional commands may be added by the encapsulation designer

as necessary, but must be documented in the application notes for the encapsulations. The SVI commands are defined in the file ”svi_types_pkg.vhd.” This VHDL file should be used by all SVI encapsulations and is contained in Appendix I.

The *External Write Request* command is used to perform a write to some entity which is external to the encapsulation receiving this command. Only the Data Output Interface is used by the requesting (originating) encapsulation, and only the Data Input Interface is used by the destination (receiving) encapsulation.

The *Internal Write Request* command is used to perform a write to some entity which is internal to the encapsulation receiving this command. For example, this command might be used to write to a configuration register of a Commercial–Off–The–Shelf (COTS) interface controller which is internal to the interface encapsulation. Only the Data Output Interface is used by the requesting (originating) encapsulation, and only the Data Input Interface is used by the destination (receiving) encapsulation.

The *External Read Request* command is used to request a read from some entity which is external to the encapsulation receiving this command. During the read request operation, only the Data Output Interface is used by the requesting encapsulation, and only the Data Input Interface is used by the encapsulation receiving the read request. The read response operation consists of an independent SVI transaction (see *External / Internal Read Response*).

The *Internal Read Request* command is used to request a read from some entity which is internal to the encapsulation receiving this command. For example, this command might be used to read a status register of a commercial–off–the–shelf (COTS) interface controller. During the read request operation, only the Data Output Interface is used by the requesting encapsulation, and only the Data Input Interface is used by the encapsulation receiving the read request. The read response operation consists of an independent SVI transaction (see *External / Internal Read Response*).

The *External Read Response* command is used to respond to a *External Read Request*. The Data Output Interface of the encapsulation which received the *External Read Request* will send an *External Read Response* command when ready to return read data to the requestor. During the read response operation, only the Data Output Interface is used by the responding encapsulation, and only the Data Input Interface is used by the encapsulation receiving the read response.

The *Internal Read Response* command is used to respond to a *Internal Read Request*. The Data Output Interface of the encapsulation which received the *Internal Read Request* will send an *Internal Read Response* command when ready to return read data to the requestor. During the read response operation, only the Data Output Interface is used by the responding encapsulation, and only the Data Input Interface is used by the encapsulation receiving the read response.

The *Read–Modify–Write Request* command is used to initiate an atomic (locked) read–modify–write operation. The Data Output Interface is used by the initiating encapsulation to transmit the *Read–Modify–Write Request,* after which time the SVI transaction is suspended (Xfer Request remains asserted) while waiting for a *External Read Response* on the Data Input Interface. Once the read data has been received and processed by the initiating encapsulation, this encapsulation transmits the modify–write data on the Data Output Interface and terminates the transaction.

The *External Read Response – Split Read Request* command is used to respond to an *External Read Request*, when the responding encapsulation wishes to delay or "split" the read response into a later External Write operation when data is available. Only the Data Output Interface is used by the encapsulation requesting the split read (i.e. the encapsulation which received the original read request); only the Data Input Interface is used by the encapsulation receiving the split read request (i.e. the encapsulation which made the original read request).

The *Response to Split Read Request* command is used to respond to an *External Read Response – Split Read Request*. This response, from the encapsulation which initiated the original read request, acknowledges the split read request, and accompanies any additional data which may be needed by the requestor of the split read to later transact an External Write (e.g. the address of the original read requestor). Only the Data Output Interface is used by the encapsulation responding to the split read (i.e. the encapsulation which made the original read request); only the Data Input Interface is used by the encapsulation receiving the split read response (i.e. the encapsulation which received the original read request).

The *Stop Read Request* command is used by an encapsulation which has initiated a read (transmitted an *External / Internal Read Request*) to terminate the read response of the responding encapsulation. This command allows the termination of a read operation in applications where the encapsulation being read would not know otherwise when to stop returning read data. Only the Data Output Interface is used by the requesting encapsulation, and only the Data Input Interface is used by the destination encapsulation.

3.2.2. Command Usage and Message Composition

All current SVI commands are contained in the VHDL file "svi_types_pkg.vhd" included in Appendix I. It is not necessary for an encapsulation to implement all possible SVI commands. If the receiving encapsulation implements the command, the data transfer continues. If the encapsulation cannot implement the command the encapsulation must, as a minimum, flag the fact that it cannot implement the command by returning the SVI error *Unsupported SVI Command*. The encapsulation could also abort the message as described in section 3.2.3.3.2.; this decision is left to the encapsulation designer. It is also highly recommended that any attempt to exercise unsupported commands be made to result in simulation warnings through use of the VHDL ASSERT statement.

In order to achieve a successful data transfer between the encapsulated data source and encapsulated data sink, it is the SVI master's responsibility to properly encode the data in the SVI message and the SVI slave's responsibility to properly decode it. In general, this means that the SVI master of the message source and the SVI slave of the message target must have prior knowledge and agreement of each other's message data formats. The implication of this requirement is that although the SVI enables interchangeability between encapsulated elements at the data link layer, true interoperability can only be achieved when interconnected encapsulations understand each other's data stream format.

3.2.3. Data Transfer Control

Basic SVI data transfers occur across the SVI as read or write operations. These basic operations can be further modified into: a "split read" when a read target responds to a read request with

a request to return read data later with a write operation; and a read–modify–write, which allows an atomic read and write from a target encapsulation's memory. From the perspective of the SVI interface, there is no distinction between an internal or external operation; this difference simply determines the receiving encapsulation's read or write source or destination.

3.2.3.1. Write Transactions

An example of a SVI external write message is shown in Figure 4 . An internal write message would be identical with the exception of the command.

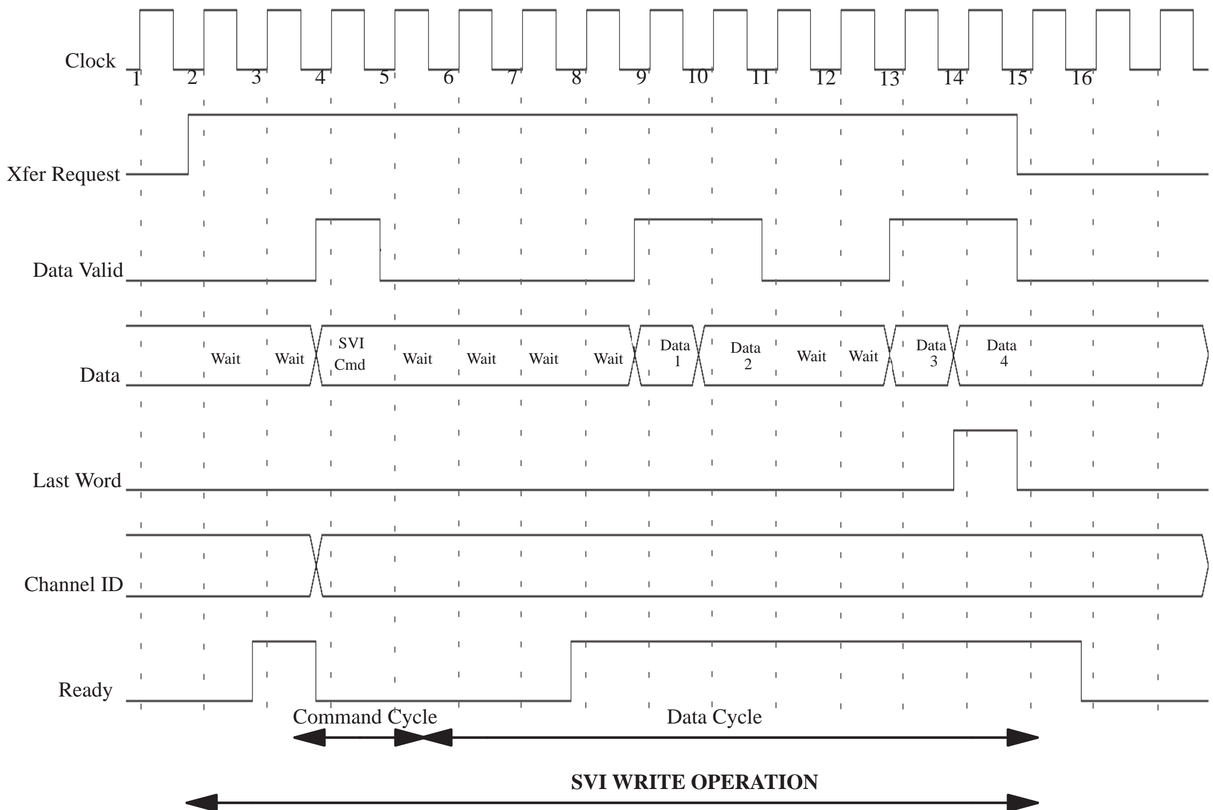

Figure 4 Example of SVI Write Operation

The message is started by the assertion of **Xfer Request** prior to clock 2. The SVI slave responds by asserting **Ready** prior to clock 3, indicating that the slave is ready for the SVI command. This is not a SVI requirement. The slave could have asserted **Ready** prior to seeing the **Xfer Request** signal asserted. This would have allowed the command cycle to occur as early as clock 2. After seeing **Ready** asserted at clock 3, the SVI master places the command on the **Data** bus for the slave by clock 4. The slave inputs the command in response to seeing **Data Valid** asserted at clock 4. This completes the command cycle. After seeing **Ready** deasserted at clock 4, the SVI master does not place any data on the **Data** bus after clock 4 until **Ready** is asserted again. Wait states occur from

clocks 5 through 8 because the SVI slave is not ready for data. At clock 8, the SVI master observes that **Ready** is asserted and places data on the **Data** bus for clock 9, indicated by the assertion of **Data Valid**. This continues until clock 10. Again wait states occur during clocks 11 and 12. However, these are due to the SVI master not supplying data. The SVI slave remains ready for data. Another data transfer occurs during clock 13, indicated by the assertion of **Data Valid**. The last data word of the message is transferred at clock 14. This is indicated by the assertion of **Data Valid** and **Last Word**. Note that **Xfer Request** remains high for this last data transfer. Thus the SVI is idle by clock 15. Notice that **Channel ID** is asserted starting with the command and ending with the last data word. The wait states at clocks 2–3, 5–8, and 11–12 are not required for an SVI write operation. They were shown to explain how data flow can be paused on the SVI. If conditions permit, data flow can be completely continuous, without any wait cycles. Also, the number of data words transferred in this message is arbitrary. There is no limit as to how many words may be transferred in a message. However, one SVI command and at least 1 data word must be transferred in any SVI message.

3.2.3.2. Read Transactions

Unlike the SVI write operation, an SVI read procedure uses both Data Interfaces of both encapsulations involved in the read and read response operations. A read is shown in Figure 5 . The Data Output and Data Input Interfaces are shown from the perspective of the encapsulation which initiates the read. The Data Output Interface of the read initiator is used to transfer a SVI message containing a SVI read request command, internal or external, along with any data necessary to perform the read by the target encapsulation, such as starting address and number of words requested. This message occurs from clock 2 until clock 6. This message must complete before a read response message can be started by the encapsulation receiving the read request. Also, there is no requirement as to how soon the encapsulation must respond with a read response after the read request message has ended. The number of data words transferred in the message is determined by the encapsulations involved in the read. There is no requirement as to how many data words are transferred, as long as one SVI command and at least one data word are transferred in each message. The read request target performs the read on the encapsulated entity, then requests a transfer on the the read initiator’s Data Input interface at clock 8. The read target then places an SVI read response, internal or external, on the SVI data bus during the command cycle. It transfers all data resulting from the read in clock cycles 8 and 9. The number of data words transferred in the message is determined by the encapsulations involved in the read. This concludes the SVI read. Note that the **Clock** on the Data Output Interface does not have to be the same frequency as the **Clock** on the Data Input Interface. Also note that behavior of both the Data Output and Data Input Interfaces each follow the protocol and timing illustrated in section 3.2.3.1.

3.2.3.3. Other SVI Operations

For both reads and writes, errors and aborts can occur during the data transfer. Errors and aborts are propagated across either the Data Input or Data Output Interface. Errors do not terminate the transfer unless an encapsulation wishes to abort the transfer based on the error. However, aborts terminate the SVI message before completion.

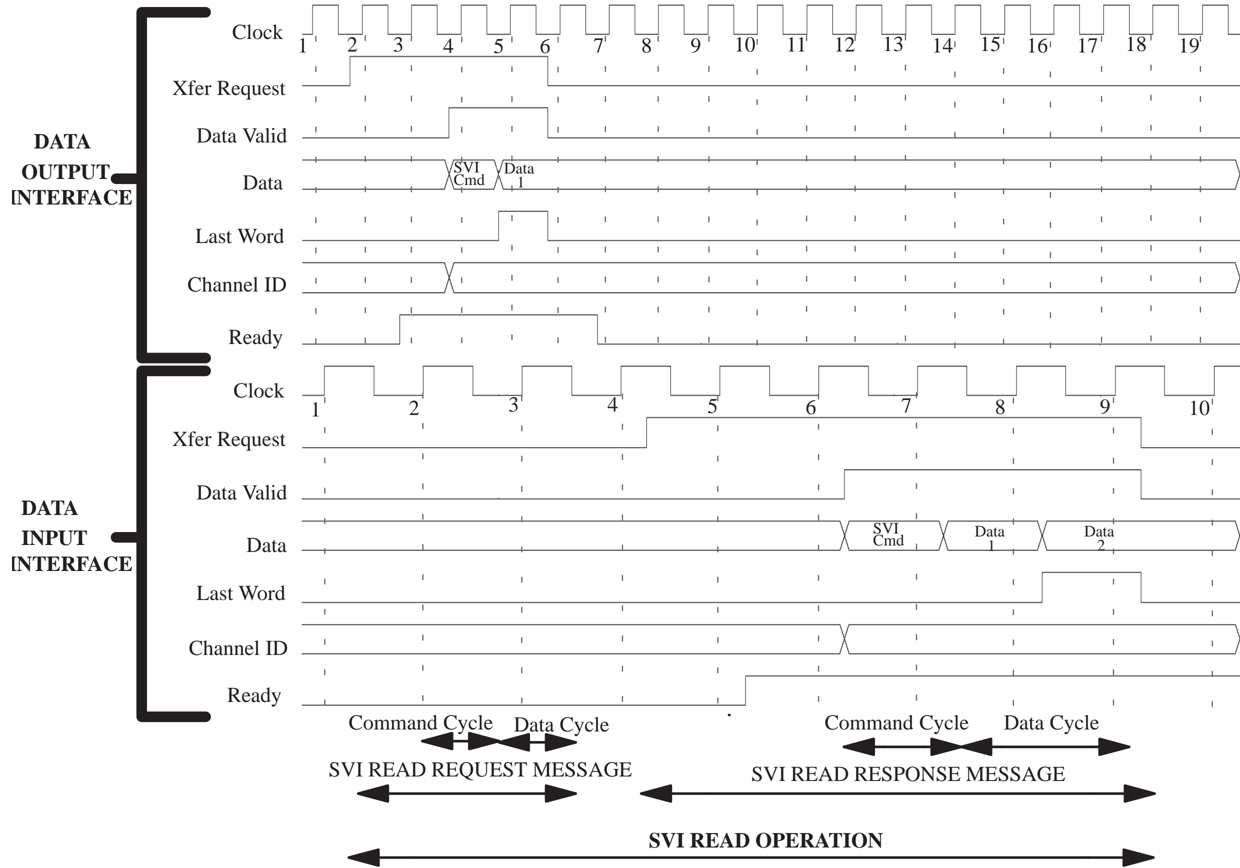

Figure 5 Example of SVI Read Operation

3.2.3.3.1. Errors

Errors can be transferred across the SVI regardless of the current state of the SVI. The **Error** signals have the following requirements. First, when asserted, the signals must contain a valid error code. Second, when sending a valid error code, the **Error** signal must remain asserted for at least one clock cycle. A list of valid error codes are shown in Table 2 .

Error (Master or Slave)	Error Type
00000000	No Error
00000001	System Error
00000010	Data Error
00000011	Synchronization Error
00000100	Unsupported SVI Command

Table 2 SVI Error Codes

These errors' meanings are left to the encapsulation designer to interpret and use. As with the SVI commands, the errors listed in Table 2 may need to be expanded; more error codes can be easily added. To insure interoperability of SVI encapsulations, these error codes are defined in the VHDL file "svi_types_pkg.vhd" contained in Appendix I. Again, all SVI encapsulations should use this file to obtain valid error codes. Even though the source of the error must use a valid error code, the encapsulation designer may decide how the encapsulation receiving the error will interpret and respond to the error. The requirements for the **Error** signal are the same for both SVI master and slave and for both Data Interfaces of an encapsulation.

3.2.3.3.2. Aborts

Aborts are used to terminate an SVI message prematurely. The **Abort** signals have similar requirements to the **Error** signals. First, when asserted, the signals must contain a valid abort code. A list of valid abort codes are shown in Table 3 .

Abort (Master or Slave)	Abort Type
00000000	No abort
00000001	Abort due to system error
00000010	Abort due to data error
00000011	Abort due to refused connection
00000100	Abort due to lost destination
00000101	Abort due to lost source
00000110	Abort due to synchronization error
00000111	Abort due to SVI error

Table 3 SVI Abort Codes

These abort codes' meanings are left to the encapsulation designer to interpret and use. As with the SVI commands, Table 3 may need to be expanded; more abort codes can be easily added. To insure interoperability of SVI encapsulations, these abort codes are defined in the VHDL file "svi_types_pkg.vhd" contained in Appendix I. Again, all SVI encapsulations should use this file to obtain valid abort codes. Even though the source of the abort must use a valid abort code, the encapsulation designer may decide how the receiving encapsulation will interpret the abort. The encapsulation designer also chooses which abort code to use when sending an abort.

In addition to using a valid abort code, the **Abort** signals are only valid during an SVI message. This starts when **Xfer Request** is asserted and ends when **Xfer Request** is deasserted. Also, the **Abort** signals must remain asserted until the SVI master's **Xfer Request** is deasserted, signaling the completed abort of the message. As with errors, aborts can occur at any time during an SVI message. The requirements for the **Abort** signals are the same for both SVI master and slave and for both Data Interfaces of an encapsulation.

An example of an SVI abort is shown in Figure 6

.

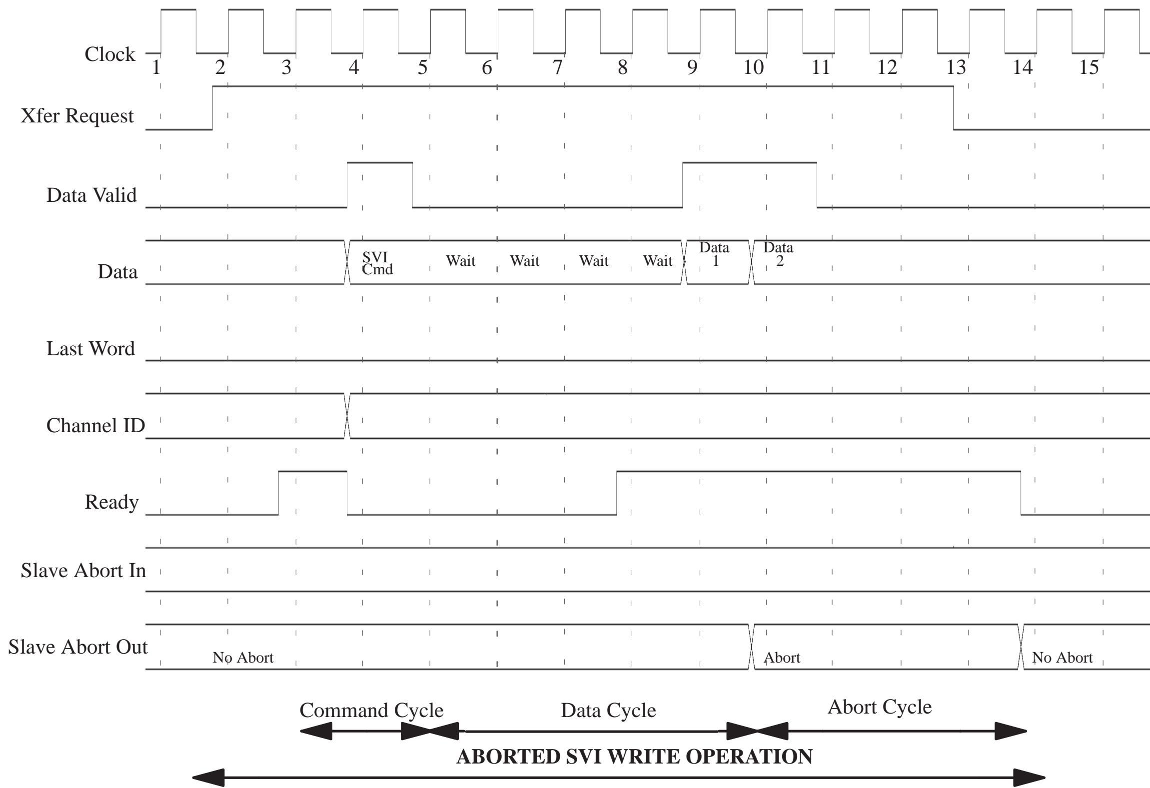

Figure 6 Example of SVI Write Operation with Abort cycle

Here, the SVI slave performs the abort of the SVI message. Detecting a condition which causes an abort, the SVI slave asserts some valid abort code at clock 10. It is not important which abort code is used for this example. The SVI slave asserts its **Abort** signals until seeing **Xfer Request** deasserted at clock 13. This signals the completed abort of the message. There is no requirement as to how long the SVI master may take to abort a message. This is left to the encapsulation designer.

The **Abort** of an SVI read operation is identical to the write with the exception that both SVI interfaces are involved. Recall from section 3.2.3.2. that the message containing the read request command must complete before a read response message can be started by the encapsulation receiving the read request. An abort then, can occur in the message requesting the read or in the message responding to the read. Regardless of which message it occurs in, the same requirements as illustrated with the abort of the SVI write operation still hold. In addition to these requirements, if the abort occurs in the message containing the read request command the encapsulation receiving this message must not send a read response message.

4. ENCAPSULATION GUIDELINES

While all SVI encapsulations must adhere to the SVI specification, it is extremely beneficial if they follow the guidelines described below. These guidelines will serve to ensure several goals. First, an SVI encapsulation should be a minimal solution in terms of hardware and performance overhead. Designers will be unwilling to incorporate the SVI into their systems unless hardware and performance overhead are small. Second, an SVI encapsulation should be robust, allowing for interoperability with other SVI encapsulations. Reuse will occur only if SVI encapsulations are robust and interchangeable. Third, an SVI encapsulation must be synthesizable. The logic necessary to implement an encapsulation should be capable of being targeted to a PLD, FPGA, or ASIC with little or no modification. Fourth, SVI encapsulations should be implemented using good coding standards. This will allow for easier understanding of an encapsulation, aiding reuse, debugging, and modification, if needed. Finally, the encapsulation should be accompanied by application notes to ensure proper use.

4.1. Encapsulation Structure

The encapsulation structure of a RASSP reuse component must provide for interoperability with other SVI encapsulations and be easily synthesized. To be interoperable with other encapsulations the encapsulation must:

- accept a clock rate into SVI slave at the Data Input Interface which is different than the clock rate for the encapsulated entity.
- accept different data–path widths, in byte increments, into SVI slave at the Data Input Interface for different applications.
- gracefully deny unimplemented SVI commands

SVI encapsulations will encompass many different interface standards and processing elements. These different entities will most likely run at different clock frequencies. Therefore, encapsulations must be able to operate with other encapsulations with different clock rates. If an encapsulation can only operate at one clock frequency, it will be less likely to be reused. However, this requirement will increase the amount of hardware needed to implement the SVI slave portion of the encapsulation. To avoid even greater hardware overhead, the SVI master should be implemented to operate at the same clock frequency as the encapsulated entity.

Different encapsulated entities will also contain varying data–path widths. By requiring that the slave of the SVI encapsulation be capable of accepting different data–path widths, two benefits are realized. First, logic to interface different data–path widths will only be required in the slave portion of the SVI encapsulation. Second, changing an encapsulation to accept different data–path widths will be made easier by substituting in different data–path width converters in the same location in the encapsulation, regardless of who designed the encapsulation. Also, this logic can be easily removed from the encapsulation if not needed for a particular application. These data–path width

converters will be available in the Model Year Architecture Reuse Library. In order to easily facilitate conversion of a slave interface from one width to another, all SVI slave elements should contain a width converter. If no width conversion is required for a given application, a one–to–one width converter should be used as a place holder.

Note that for a specific instantiation of any SVI encapsulation, the input clock rate and data–path will remain fixed; these values will not change dynamically. Encapsulations must be designed with the flexibility however, to operate properly under the operating conditions associated with one instantiation to the next.

In addition to clock rates and data–path widths, an SVI encapsulation must respond gracefully when an unimplemented SVI command is received. As a minimum, the encapsulation must return the SVI error *Unsupported SVI Command*; a more robust approach would be to also abort the transfer after reporting the error. It is also highly recommended that any attempt to exercise unsupported commands be made to result in simulation warnings through use of the VHDL ASSERT statement.

Three SVI encapsulations, illustrating good encapsulation structure, will be examined. First will be a internal node encapsulation with a Processing Element as the node, then an interconnect fabric encapsulation, and finally an interconnect fabric encapsulation incorporating a Commercial–Off–The–Shelf (COTS) interface controller. All encapsulations are shown as logical entities, or as they would appear in the Model Year Architecture Reuse Library. They are not shown as hardware entities, or as they would appear after synthesis.

4.1.1. Internal Node Encapsulation Structure

Figure 7 shows an example of an internal node encapsulation, with a processing element (PE) as the node. Note that the PE block may contain multiple processing entities, local memory, DMA controller, and whatever support logic is necessary. The shaded area shows the SVI wrapper, containing the logic necessary to implement the SVI. Explicitly shown here in the SVI Slave is a FIFO and an SVI width converter.

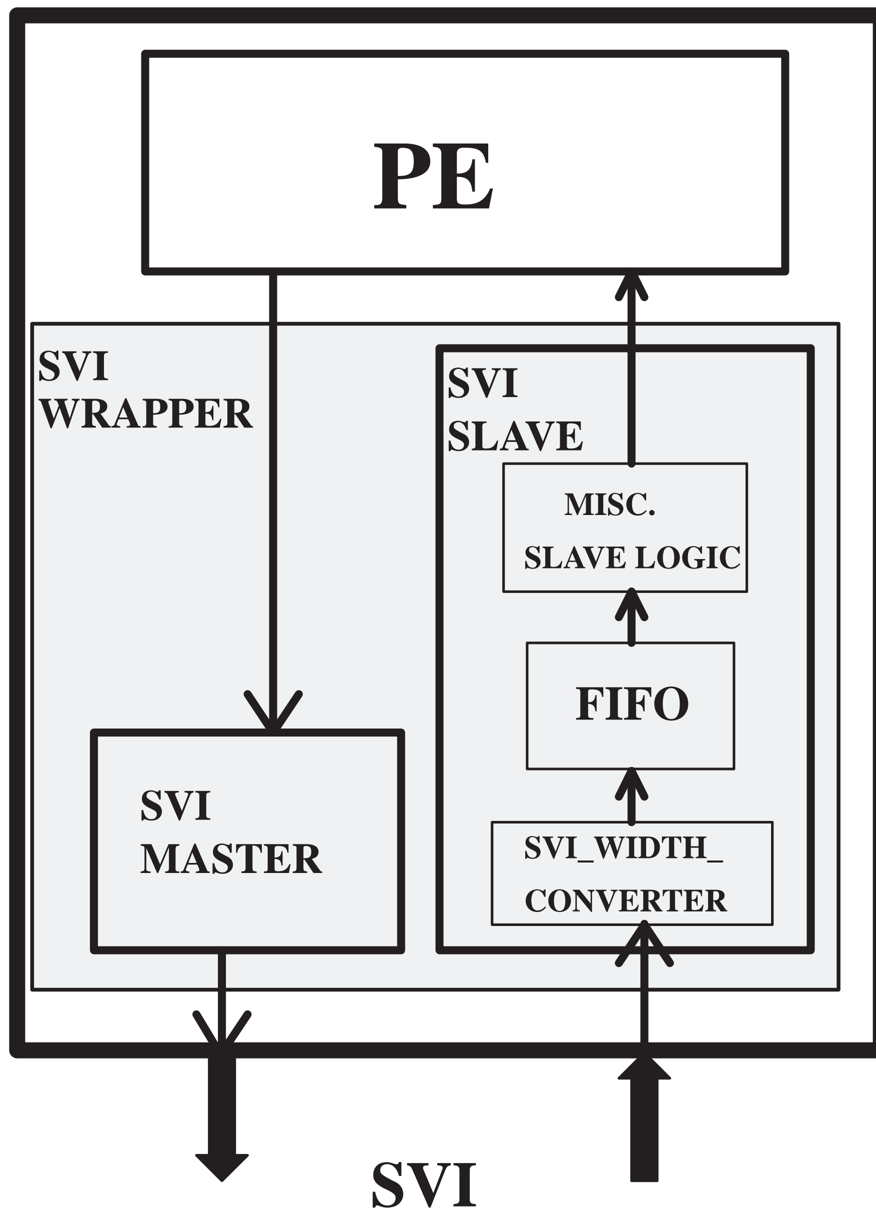

Figure 7 Internal Node Encapsulation Structure

To ensure that the encapsulation can accept different incoming data–path widths, the SVI slave includes the entity svi_width_converter. This is a library element capable of mapping data path signals from one width to another. Two converters are contained in Volume IV for reference. Even though a particular instance of an encapsulation will use a fixed data–path width, a one–to–one width converter should be used in all encapsulations to allow simple reuse of the encapsulation through the substitution of a different width converter, should a different incoming data–path width be required.

To allow for applications where the incoming SVI slave clock rate is different than the clock rate of the PE, a FIFO along with separate logic on each side of the FIFO is used. This allows the SVI slave to write to the FIFO at one data rate while the FIFO can be read the at another data rate, preferably the same as the PE. In order to reduce both the level of design effort and the quantity of hardware resources taken by the synthesized logic, a COTS FIFO may be used rather than including this data storage in the synthesizable VHDL. The pin count, PLD/FPGA/ASIC cell count, board space, and power consumption of logic device with the synthesized encapsulation should determine whether or not an external FIFO chip or internal logic ram macro–cell should be utilized. Note that whether the FIFO function is implemented internally or externally to the synthesized logic, the FIFO is still functionally considered part of the SVI wrapper. If an external FIFO is implemented, a separate "synthesis wrapper" will need to be created for synthesis, partitioning out any non–synthesis parts.

The SVI master interfaces the PE to the outgoing SVI signals. The SVI master typically drives the Data Output Interface at the same clock rate as supplied by the PE to eliminate the need for a FIFO, like the SVI slave, and to allow for a single state machine to implement the necessary logic. However, the encapsulation designer may chose to implement an encapsulation differently.

4.1.2. Interconnect Fabric Encapsulation Structure

Figure 8 shows an example of an interconnect fabric SVI encapsulation. In this example, both the SVI wrapper and the fabric interface will be implemented in the synthesized hardware, and both are described in synthesizable VHDL code. The approach of creating a custom fabric interface, as opposed to employing a COTS interface controller is recommended only when the interconnect protocol is extremely simple or no commercially available controller exists. The shaded area shows the SVI wrapper, containing the logic necessary to implement the SVI for the given interconnect fabric interface. As in the example of the PE encapsulation, in the SVI Slave is included a FIFO and and SVI width converter. The FIFO and SVI width converter ensure that a particular instance of the encapsulation will operate with any incoming data width and any incoming clock rate. Again, any particular encapsulation may have additional functional blocks in either the SVI Slave or Master, based on the functional partitioning decisions of the encapsulation designer.

The interconnect fabric interface is mapped to the outgoing SVI signals by the SVI master. The SVI master outputs data using the same clock as supplied or derived from the fabric interface to eliminate the need for a FIFO, like the SVI slave, and to allow for a single state machine to implement the necessary logic.

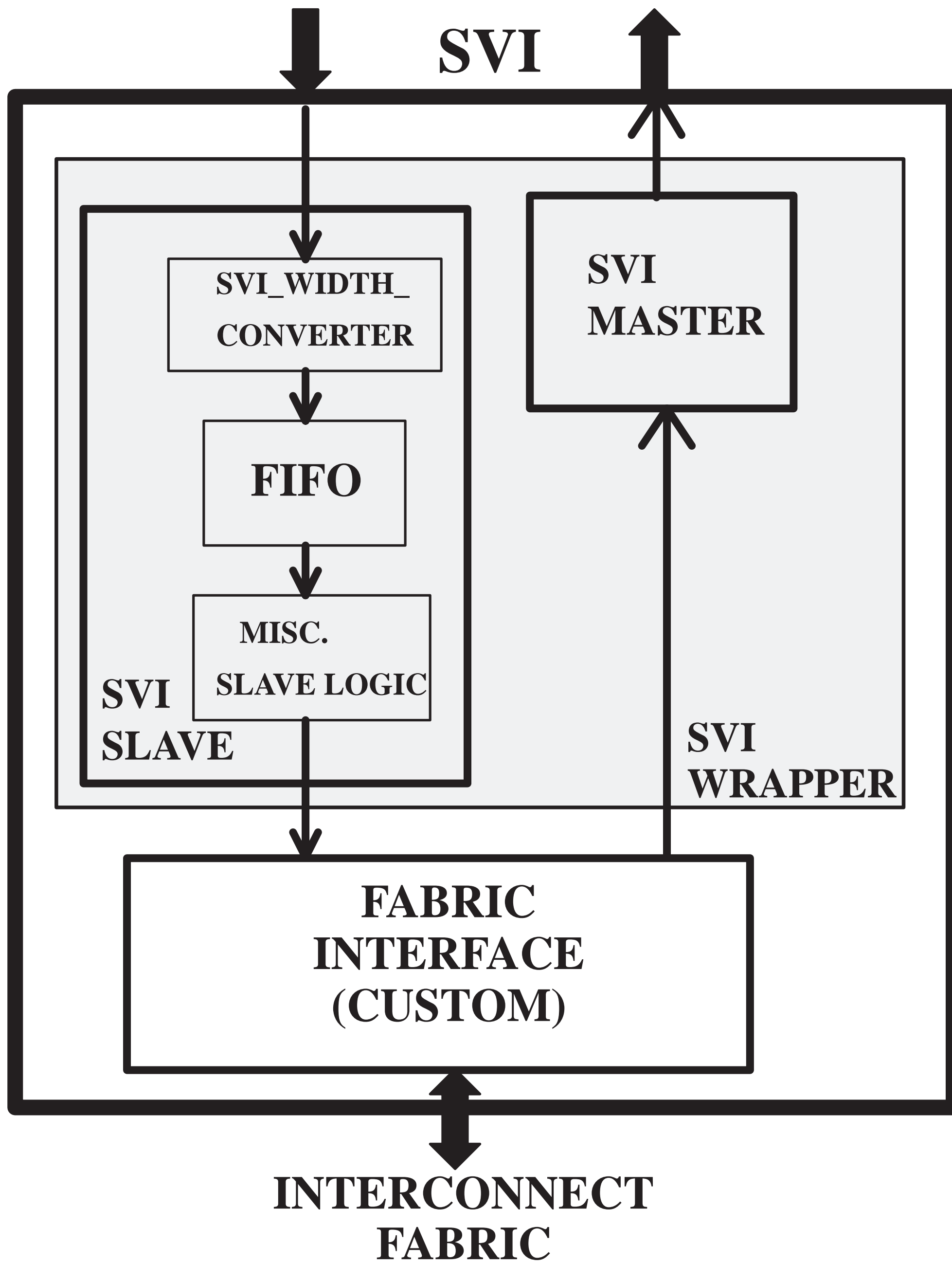

Figure 8 Fabric Interface Encapsulation Structure

4.1.3. Interconnect Fabric with COTS Controller Encapsulation Structure

Figure 9 shows an SVI fabric interface encapsulation implemented with a COTS interface controller. Unlike the fabric interconnect encapsulation shown in the previous example, this example employs a COTS interface controller to implement the the fabric interface. If such a device is available, this approach is preferred. Using a COTS interface controller minimizes the level of de-

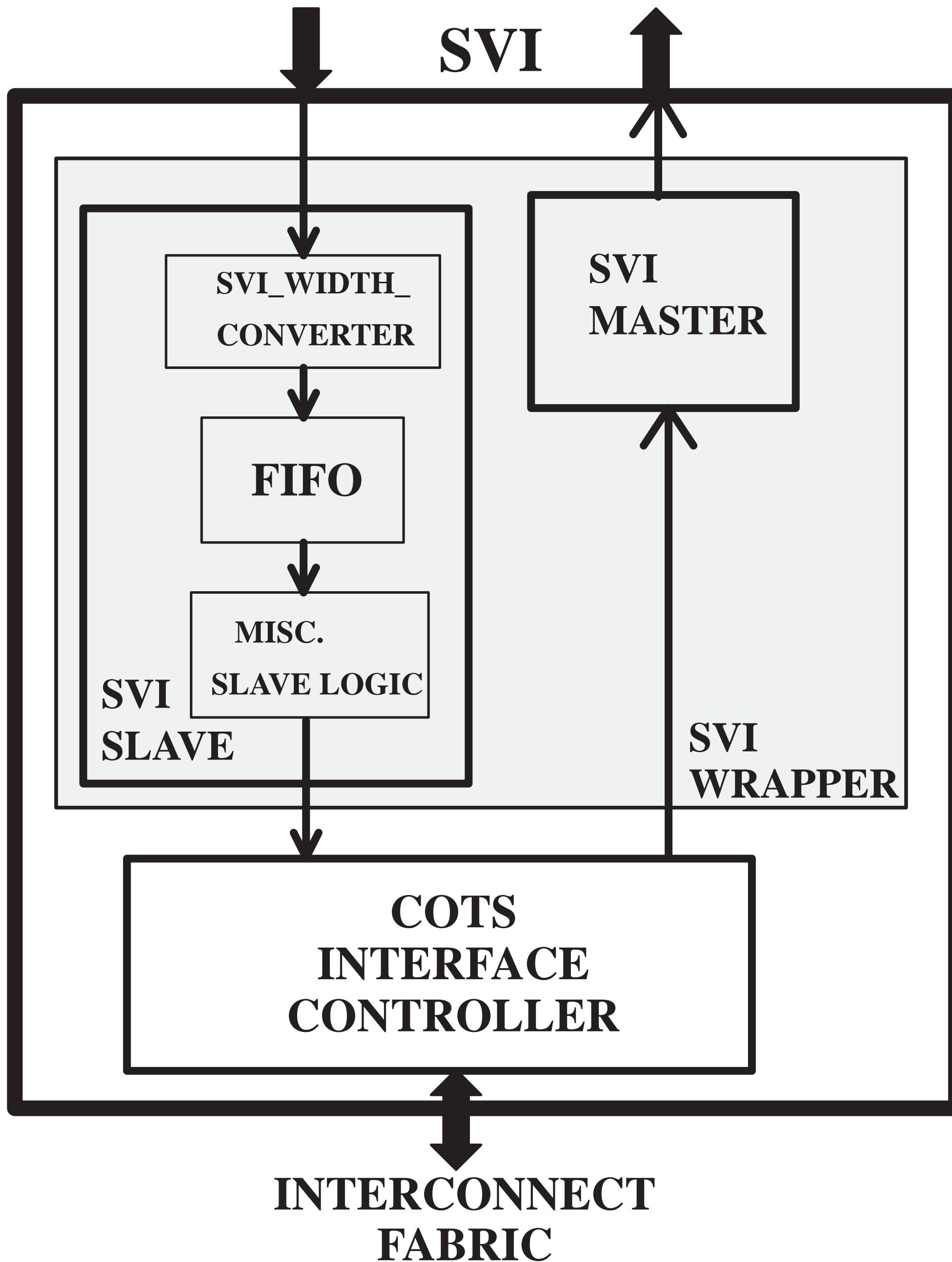

Figure 9 Fabric Interface Encapsulation With COTS Interface Controller

sign effort and the quantity of hardware resources taken by the synthesized logic. In addition, the use of a COTS interface controller also ensures correct implementation of the open interface standard being encapsulated. Note that the FIFO is optional. If the COTS interface controller includes an internal FIFO an additional FIFO would be superfluous. Also note that a separate “synthesis wrapper” will need to be created for synthesis, partitioning out the COTS controller and, if included, any external FIFO.

4.2. General Considerations

In addition to following guidelines described above, some general guidelines should be followed as well.

- Individual components of an encapsulation, such as the SVI master or slave, should be implemented as state machines. State machines are easiest to design, debug, modify, and synthesize. Random logic should be avoided where possible.
- Minimize the amount of data storage needed by controlling data flow control signals, such as the SVI signal **Ready**, asynchronously. Designing these signals as Mealy outputs helps data flow control stop data transfer more quickly, thus reducing the amount of data storage needed. Data flow information will be transferred from the SVI to the encapsulated entity and vice–versa more quickly if data flow outputs are determined by state and input values versus state alone. This will reduce the amount of FIFO cells needed to store incoming data during a pause of data flow.
- Don’t reference the falling edge of any clocks, unless required by the encapsulated entity. This can place tight timing constraints on a hardware realization of the encapsulation and also makes synthesis more difficult. More will be discussed about this in section 4.4.
- Encapsulations should have no absolute time references, such as propagation delays. These are technology dependent and are an outcome of the synthesis process.
- All hardware design guidelines, i.e. no gating of the clock etc., and software standards, i.e. extensive commenting, consistent coding style, etc., should be followed.

4.3. Application Notes

Application Notes will help a system architect determine if an encapsulation is useful for a design, and if so, how to use the encapsulation in the design. As a minimum, SVI encapsulation application notes should include:

- Block diagram of encapsulation showing all major components and their relationship to one another

- Description of each component in the block diagram
- What SVI commands are implemented by encapsulation
- What SVI commands encapsulation generates (if applicable)
- How encapsulation interprets data in SVI message for each accepted command
- How SVI encapsulation will respond to all SVI errors
- How SVI encapsulation will respond to all SVI aborts
- What SVI errors encapsulation may generate
- What SVI aborts encapsulation may generate
- How SVI encapsulation generates and responds to interrupts
- What is the native data width that the SVI slave accepts (without SVI data width converter)
- What current data width does the SVI slave accept (with SVI data width converter)
- How encapsulation responds to and generates resets
- What signals are not implemented
- Data Sheets for all COTS components, including encapsulated entity, necessary to implement encapsulation

This information should encourage reuse rather than redesign of SVI encapsulations.

4.4. Synthesis Guidelines

In order to produce an efficient and optimally synthesized realization of the SVI encapsulation, it is important that the designer understand both the supported VHDL constructs and the hardware inferencing rules of the chosen synthesis tool. Supported constructs and inferencing rules for the Synopsys Design Compiler, the synthesis tool for which early SVI synthesis trials have been conducted, are fully documented in the references listed below. Although covered in these references, the following guidelines emphasize a number of VHDL coding issues which were found to be critical in the SVI synthesis trials.

4.4.1. Model Hierarchy

It is recommended that SVI encapsulations be described in a modular, hierarchical fashion. A hierarchical design simplifies model verification, as blocks can be unit tested during VHDL simulation. Furthermore, in the synthesis phase, it is recommended that the hierarchical modules be individually synthesized prior to the synthesis of the complete design. This allows viewing of a relatively con-

tained portion of the synthesized logic schematics, in order to evaluate the design for inefficient or unintended hardware constructs. If this evaluation is attempted on a non–modular design, the schematics may prove too unwieldy to allow any meaningful analysis of the realized design.

4.4.2. Combinational vs. Sequential Processes

In general, most encapsulation blocks should be constructed with two VHDL processes: one describing the sequential (synchronous) portions of the logic (those containing storage elements) and at least one describing the combinational (asynchronous) portions of the logic. Further, as discussed in section 4.2., it is recommended that these processes follow a state machine implementation style (see examples in SVI Appendices, Vol. IV). In order to produce the most efficient hardware realization of the described logic, it is important that only those signal assignments for which registers are intended, be assigned in the sequential process. Any signals unnecessarily described in the sequential process will produce superfluous registers, which can greatly impact overall cell count.

4.4.3. Sequential Coding Specifics

The greatest challenge to writing synthesizable code is in the description of the sequential processes; both in preventing unnecessary registers and in describing the sequential process in a style acceptable to the synthesis tool. Specifically:

– One process may not contain more than one IF or WAIT UNTIL (edge expression), where (edge expression) is of the form (clk'EVENT AND clk = '1') or (NOT clk'STABLE AND clk = '1').

– An edge expression must be the only condition of an IF or ELSIF; there must be only one edge expression in an IF; and the edge conditioned IF may not have an ELSE or ELSIF clause.

– A clock signal cannot be read as data in a process.

– When assigning a high–impedance ('Z') value to a sequential signal, do so by creating a tri–state enable signal inside the sequential process, and by performing the 'Z' assignment in a combinational process or the concurrent portion of the VHDL code. If the 'Z' assignment is made in the sequential process, the synthesis tool will create a register to hold the inferred tri–state enable signal (in the case of a bussed tri–state signal, synthesis will create a unique register for the enable of *each* signal bit).

4.4.4. Miscellaneous: Unsupported VHDL Constructs

– Signal assignment delay information is unsupported (ignored) by synthesis.

– TEXTIO is unsupported by synthesis; unused references to TEXTIO must be removed.

– Synthesis does not support PHYSICAL types. Any signal assignment delays which are defined by generics of type PHYSICAL (i.e. TIME), must be either removed or replaced with explicit time delays (which will in turn be ignored by synthesis).

– Configuration statements are unsupported (ignored) by synthesis. This is only a problem when configuration statements are used to associated entities to a component which has a name different from the entity.

– Initial values are unsupported (ignored) by synthesis. If initialization is required, it must be accomplished with an explicit signal assignment, preferably during a reset cycle.

– Synthesis can not resolve the I/O pad insertion of a tri–state INOUT port driven with multiple tri–state drivers. The designer must modify the design so that each out of any INOUT port has no more than one tri–state gate connected to it. Two modification methods are illustrated in Figure 10 .

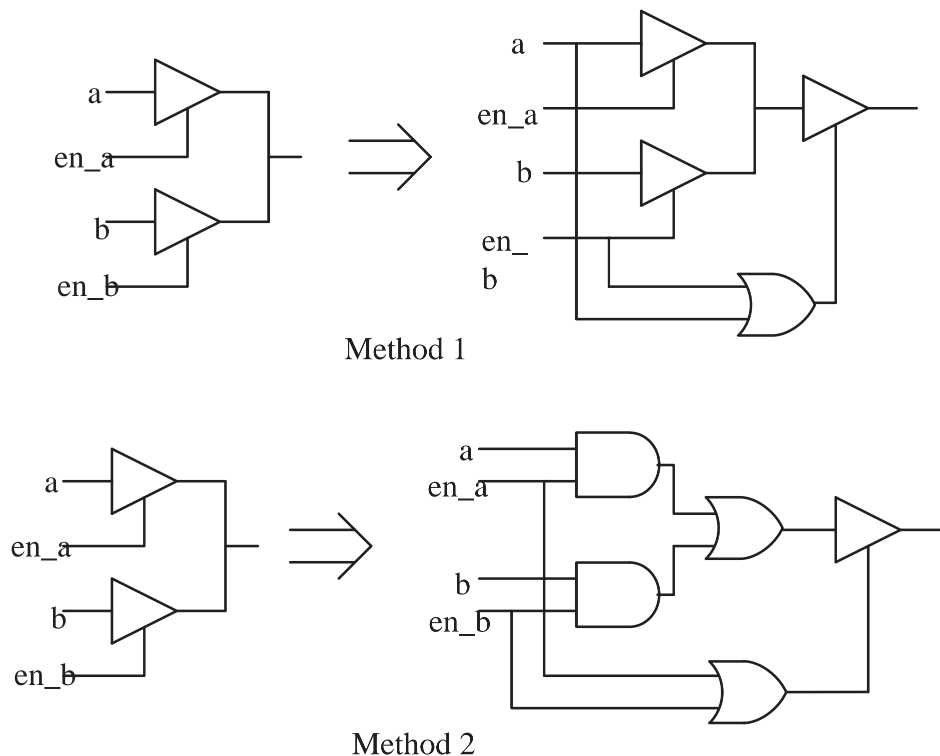

Figure 10 Methods for Converting Outputs to a Single Tri–state Driver

– Avoid unconstrained INTEGER type variables or signals. For unconstrained INTEGERS, synthesis will instantiate unnecessary hardware to support the maximum integer range (generally 32 bits).

– The use of the OTHERS construct is not supported for signal slices:
e.g. foo(6 DOWNTO 3) <= OTHERS => ’0’; is unsupported for the signal foo(7 DOWNTO 0)

– Do not make multiple assignments to the same signal within a process (this does not apply to assignments to the same signal in mutually exclusive IF or CASE statements). Instead, create a variable to temporarily receive the value of the signal, and assign the variable value to the signal before

exiting the process.

Additional References:

VHDL Compiler Reference Manual, Synopsys, Inc.

Introduction to HDL–Based Design Using VHDL, Synopsys, Inc.