LOCKHEED MARTIN
LOCKHEED MARTIN
ADVANCED TECHNOLOGIES LABORATORY

SVI Verification Study: Encapsulations of the *Benchmark II−Data I/O Board* and the *Mercury Rino−RIC Chipset*

5/17/96

This document provides a decription of the Data I/O board and Rino chipset encapsulations using the SVI methodology in accordance with the goals of the Model Year Architecture Task of the RASSP Program.

Initiated by: Amit Chhabra
AMES, Embedded Processing Lab
Lockheed Martin, Advanced Technologies Laboratory

Concurred by: Greg Buchanan
SMES, Embedded Processing Lab
Lockheed Martin Advanced Technologies Laboratory

1. Scope **2**

2. Referenced Documents **2**
2.1 Vendor Documents 2
2.2 ATL Documents 2

3. Verification Objective **2**

4. Data I/O Board — Rino IC Encapsulation Design **3**
4.1 Data I/O Board FIFOs 4
4.2 State Machine Controller Design 4
4.2.1 Transmit FIFO Control 5
4.2.2 Receive FIFO Control 6
4.3 VHDL Implementation Notes 7

5. SVI Encapsulation Performance Results **8**
5.1 Hardware Overhead 8
5.2 Latency Overhead 8
5.3 Performance Versus Message Length 9

6 Summary **9**

7 Recommendations **10**
7.1 SVI Well Suited for Data Streaming 11
7.2 Need for a Stop Read Command 11

LOCKHEED MARTIN

Advanced Technology Laboratories
1 Federal Street − A&E 2W
Camden, NJ 08102
(609) 338−4250

Date: May 17, 1996

Author: Amit Chhabra

Contact: Janet W. Wedgwood (janet.e.wedgwood@lmco.com)

Benchmark 2 Data IO and RACEway RINO Encapsulations: Final Report

1. Scope

This document serves to provide a complete description for the *SVI Verification Study* involving the *Rino IC−Data I/O Board* encapsulation.

2. Referenced Documents

2.1. Vendor Documents

Developers Guide: The RIC−RINO Chipset, Mercury Computers

RACEway Interlink − Data and Physical Layers, VITA, Feb. 1995, Rev. 1.4.

CMOS SyncFIFO (IDT722xx7B) − Specifications, Integrated Device Technology, Inc., Aug. 1993.

Programmable Skew Clock Buffer (CY7B991) − Specifications, Cypress Semiconductor Corporation, Feb. 1995.

2.2. ATL Documents

Standard Virtual Interface Specification, Version 0.5, LM − ATL, Jun. 1995.

DATA I/O Board − Benchmark II Diagrams.

3. Verification Objective

The objective of this verification study is to create and demonstrate a VHDL model of the *Standard Virtual Interface (SVI)* as specified by the *SVI Specification* and to implement this model using the existing *SVI VHDL templates*. The design and implementation of the SVI model in VHDL for particular encapsulation is to serve as a method of quantifying the merit of the SVI concept, the quality of the SVI development process, and to provide an understanding of where some of the difficulties and concerns in developing an SVI encapsulation exist. Furthermore, the VHDL model is meant to provide an entity from which the SVI overhead and performance can be measured.

For this verification study, the RASSP Benchmark II – *Data I/O Board* (as shown in Figure 1.) was used. An SVI interface was provided between the Data I/O board its network inter-

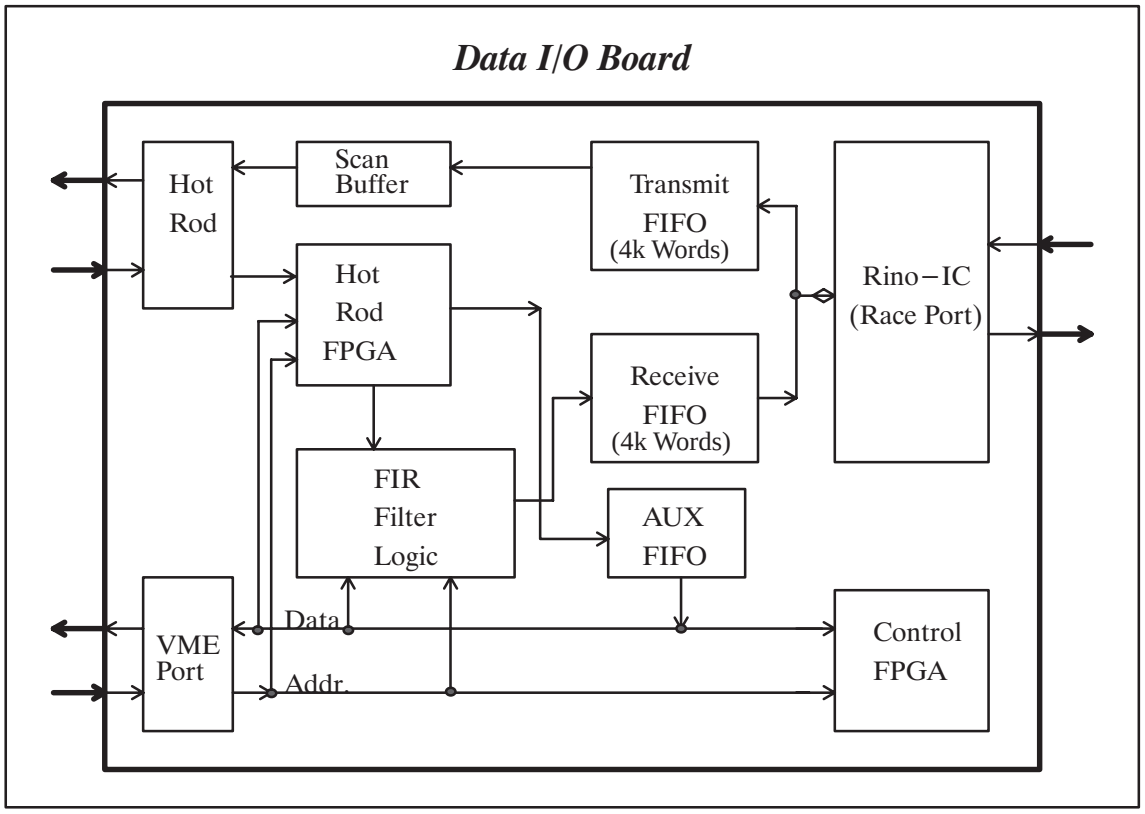

Figure 1.: Data I/O Board before the SVI Encapsulation

face, the *Mercury Rino–IC*. The SVI interface was created by effectively dividing the Data I/O board at its transmit and receive FIFOs as shown in Figure 2.. The design of the SVI

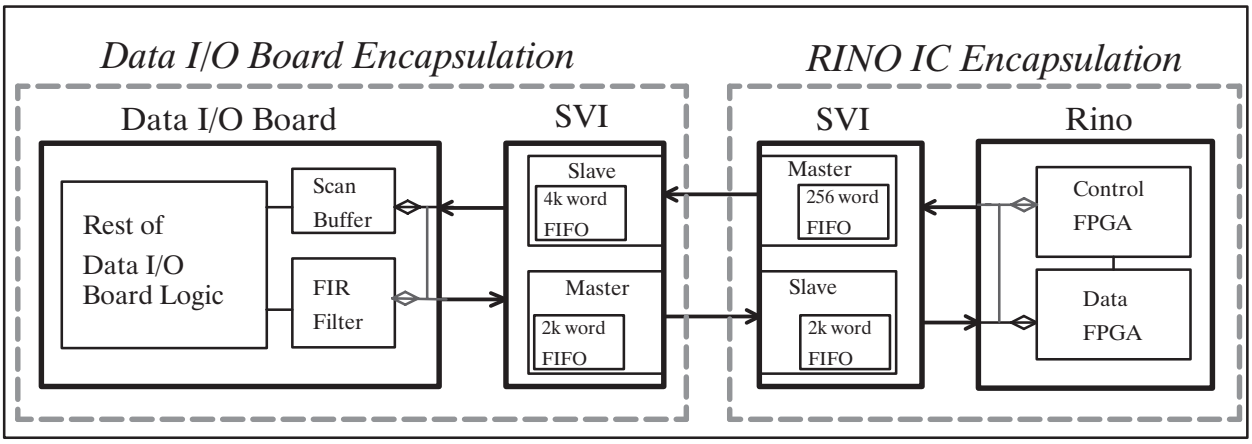

Figure 2.: Data I/O Board and Rino Chipset Encapsulated

encapsulation is discussed in the next section.

For practical purposes, the SVI encapsulations of the Data I/O Board and the Rino chipset will serve as components in the *Model Year Architecture (MYA)* reuse library. The VHDL models will be available as *plug and play* elements that can be connected with other SVI encapsulated elements. The availability of these models will provide the ability for further performance studies that will be conducted in order to fully develop the SVI concept for the MYA task as part of the RASSP Program.

4. Data I/O Board – Rino IC Encapsulation Design

Both the *master* and *slave* entities in each of the SVI wrappers shown in Figure 2. can be described in the two parts, an *SVI controller* (implemented as a state machine) and a FIFO (as shown in Figure 3.). The SVI controller and the FIFO are used to maintain a consisten-

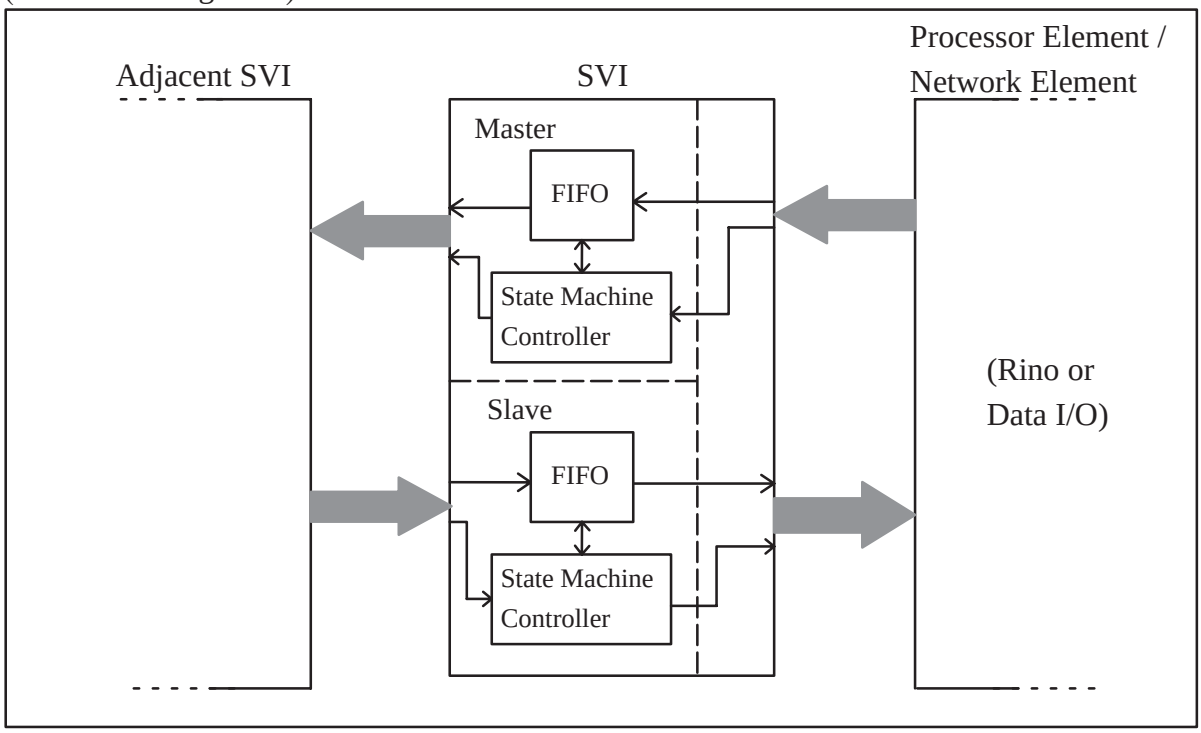

Figure 3.: SVI FIFO and State Machine Entities

cy of the data flow between the attached SVI and the encapsulated element (this is further described in the SVI Specification).

Since the implementation goal of this study was to separate and encapsulate the Data I/O board (the processing element) and the Rino chipset (the network element), it was clear that the division would be at transmit and receive FIFOs on the Data I/O board. In this way the interface from the Rino chipset to the Data I/O board would be a “straight forward” FIFO interface. Based on the notion that each SVI wrapper minimally called for a FIFO to provide a temporary storage for *slave data*, a FIFO was incorporated into each wrapper. In an effort to reduce complexity, the transmit and receive FIFOs of the Data I/O board were merged into the FIFOs in the SVI wrappers and removed from the Data I/O board (as show in Figure 2.). In order to successfully implement the merging of the FIFOs, the FIFO status flags and control signals that connected to the Data I/O board and the Rino chipset had to be maintained, and the following section further details this topic.

4.1. Data I/O Board FIFOs

Figure 4. shows the Data I/O board and Rino chipset connections to the transmit and re-

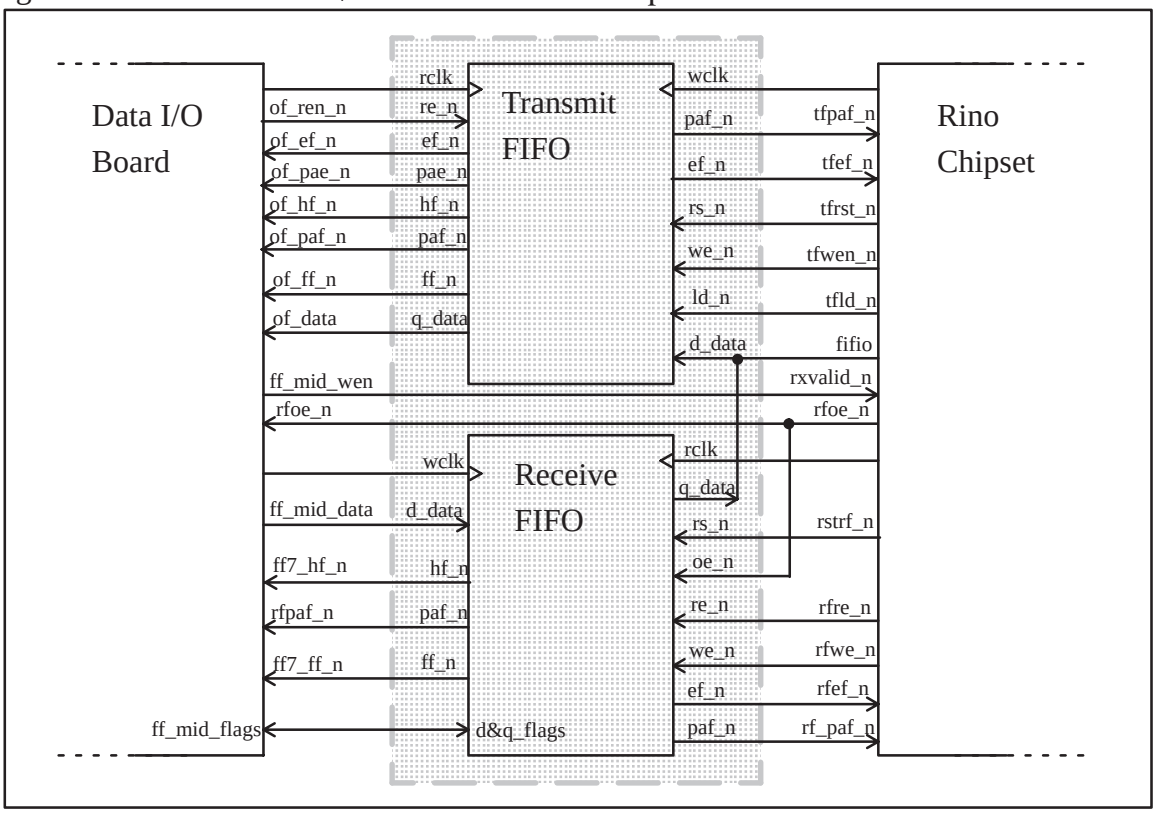

Figure 4.: Data I/O Board FIFO Configuration before SVI

ceive FIFOs before the SVI encapsulations were added to the system. These connections were maintained by implementing the SVI wrappers as shown in Figure 5. and Figure 6..

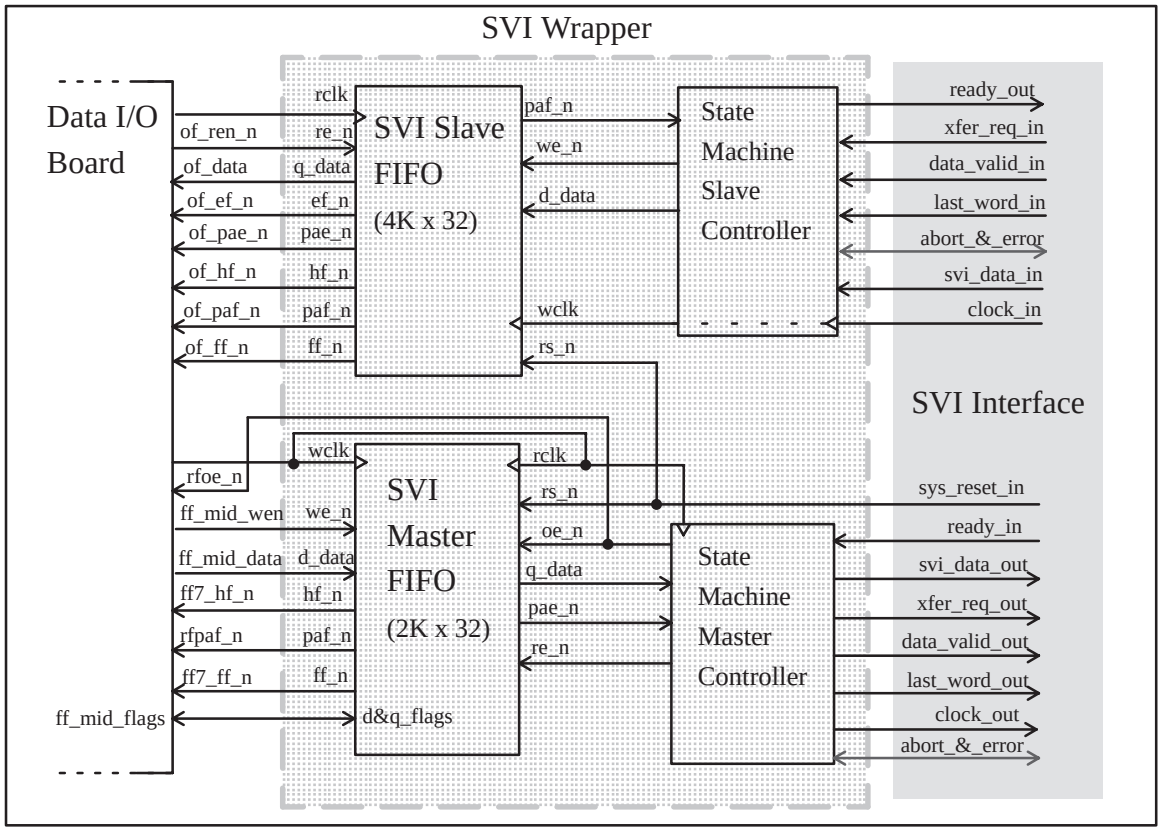

Figure 5.: Data I/O Board Wrapper

4.2. State Machine Controller Design

The *state machine controllers* implemented in each SVI entity provide control their respective FIFO in the interface. A master controller interprets signals from either the Data I/O board or the Rino chipset and translates them into FIFO flags and SVI signals. A slave controller interprets SVI signals and translates them into FIFO flags and Data I/O board or the Rino chipset signals.

4.2.1. Transmit FIFO Control

In the SVI architecture, the *transmit FIFO* is made up of a 256 word and a 4Kword FIFO. The state machine controller for the Rino's master only initiates an SVI transfer if its FIFO is half full (or its FIFOs *half full flag* becomes asserted). Once the controller initiates a transfer, it waits for the *SVI ready* to become asserted, and then it will *stream* data out of its FIFO by asserting the FIFO's *read enable* line. This will continue until the FIFO becomes empty or Data I/O board's SVI deasserts the ready signal. Essentially, the Rino's master controller uses the *SVI ready in* to control its FIFOs read enable signal. An end of transaction can only occur when the FIFO becomes empty. The *almost empty flag* from the FIFO is used to signal the controller to assert the *last word* signal.

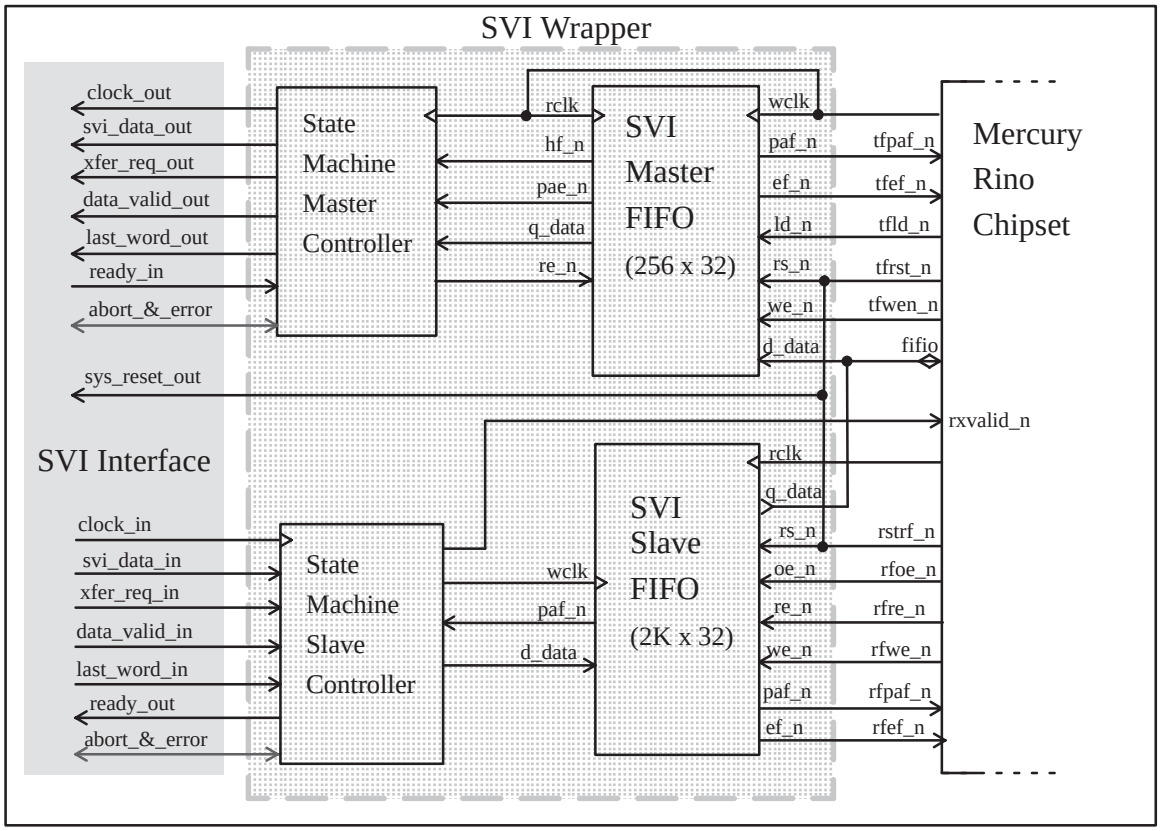

Figure 6.: Rino Chipset Wrapper

The Data I/O board's slave's controller will assert an *SVI ready* in response to a *transfer request* only if its FIFO has less then 3968 words filled (3968 words is default number of words that it takes to trigger the *programmable almost full flag* on the FIFO, see the IDT 72245b documentation referenced). Once a transfer is initiated, the slave's controller will pass the data from the SVI data lines to its FIFO until its FIFO becomes *almost full* (or its *almost full flag* is asserted) or the transaction is ended by the master. The slave controller uses the *SVI data valid* signal to control its FIFO's *write enable* signal (i.e., as long as the data is valid, the controller will keep the write enable asserted).

A more detailed description of the state machine controller for these FIFOs can be obtained by examining the VHDL code for the Data I/O board Wrapper. The code shows exactly how the above description is implemented, along with a description of the *reset* and *load* flags.

4.2.2. Receive FIFO Control

In the SVI architecture, the *receive FIFO* is made up of two 2K word FIFOs. The state machine controller for the Data I/O board's master only initiates an SVI transfer after its FIFO has filled 512 words of its space (or its FIFOs *almost empty flag* is deasserted) or the last set of *radar dwell data* (that is less than 512 words) is resident in its FIFO. Once a transfer is initiated, the master's controller will stream data from its FIFO by asserting the FIFO's *read enable* line. The controller will continue to send data out the *SVI data lines* until its FIFO becomes empty or the Rino's SVI deasserts its *SVI ready*. An end of a transaction can only occur when Data I/O board's master FIFO becomes empty. The *almost empty flag* from the FIFO is used to notify the controller to assert its *last word* signal.

The Rino's slave's controller will assert an *SVI ready* in response to a *transfer request* only if its FIFO is not *almost full*. Once a transfer is initiated, the slave's controller will pass data from the SVI data lines to its FIFO until it becomes *almost full* (or its *almost full flag* is asserted) or the transaction is ended by the master. The slave controller for the Rino will assert its FIFO's *write enable* signal as long as the *SVI data valid* signal from the Data I/O board's SVI remains asserted.

A more detailed description of the state machine controller for these FIFOs can be obtained by examining the RACE_RINO VHDL code. The code shows exactly how the above description is implemented, along with a description of the *reset* and *load* flags.

4.3. VHDL Implementation Notes

This section details the use of *VHDL Generics* in the Data I/O board and the Rino chipset encapsulations. Below, a reason for why generics are needed and how they are used in these encapsulations is provided.

The VHDL implementation of both the Data I/O board and Rino chipset encapsulations have been simulated in a testbench that forces data to flow between the encapsulations across an SVI. In this case, the operating clock frequencies are determined by the requirements of the Data I/O board and the Rino chipset. Therefore, it can be anticipated that in a configuration (or testbench) in which these encapsulations are not connected to each other (but to other SVI encapsulated components), the operating clock frequency of the SVI may change. If the operating clock frequency change, it will become important to appropriately adjust the value loaded as the *programmable almost empty* and *programmable almost full* in the FIFOs (see the IDT722x5b specifications for more details on FIFO status flags and the VHDL code to see how to implement this change). The VHDL implementations on these models facilitate the flexibility of adjusting the programmable load values for the FIFOs because the values for the almost full and empty flags have been implemented as *generics*. This allows the user to enter in a value for the particular load value during *runtime* (or in script that calls the simulation tool). If no values are provided during runtime, the VHDL models will default to the case for which the Data I/O board and the Rino chipset are connected together.

5. SVI Encapsulation Performance Results

In this verification study, it was proven that an existing radar data board could be retro−fitted with an SVI and still perform as intended. However, the tradeoff for *plug n' play* flexibility is an increase in hardware and overall latency.

5.1. Hardware Overhead

For this pair of encapsulations, the hardware is altered by replacing four 4K by 18 bit FIFOs by two 256 by 18 bit FIFOs, four 2K by 18 bit FIFOs, two 4K by 18 bit FIFOs, and a controller FPGA (that implements all four state machines). The net increase in the amount FIFO storage is the two 256 word FIFOs and an FPGA. However, because of the re−partitioning of FIFOs, the amount of board space required by these components will be about twice that of the four FIFOs in the non−encapsulated case.

5.2. Latency Overhead

The increase in latency for the *dataio_loop_tb.hdl* testbench is listed below:

Time for the first data element to make through the loop:
Original Data I/O Board: *490,600 ns*
SVI Encapsulated: *493,950 ns*
Difference: *3350 ns or* ***.68% increased latency***.

Time for one dwell of data to flow through the transmit FIFO (or SVI equivalent of)
Original Data I/O Board: *470,777 ns*
SVI Encapsulated: *472,527 ns*
Difference: *1750 ns or* ***.37% increased latency***.

Time for one dwell of data to flow through the receive FIFO (or SVI equivalent of):
Original Data I/O Board: *429,450 ns*
SVI Encapsulated: *432,488 ns*
Difference: *3038 ns or* ***.71% increased latency***.

From these encapsulations, it can be see that the increase in latency due to SVI is very insignificant compared to the time that data spends in other parts of the Data I/O board system. The location of the SVI between the Data I/O board and the Rino is not a *bottle−neck* point within the system. So, that even with the insignificant 4 cycles per SVI transaction that is added to Data I/O *pipeline*, there are other parts of the Data I/O board that slow down the data flow rate. In other words, the even though a delay is added to the system by the SVI, it is insignificant because the parts of the Data I/O board that receive data from the SVI wrapper are not ready to accept because operate at a fraction of the data rate than that of the wrapper (however, a 4 cycle latency will be added for the first word).

5.3. Performance Versus Message Length

In the above sections it has been shown how a system retro−fitted with an SVI is effected in terms of hardware and latency. This section provides an insight at how the latency overhead of can vary depending on the number words in each SVI transfer.

Figure 7. shows a graphs that describes the percentage of bandwidth used for *SVI overhead*

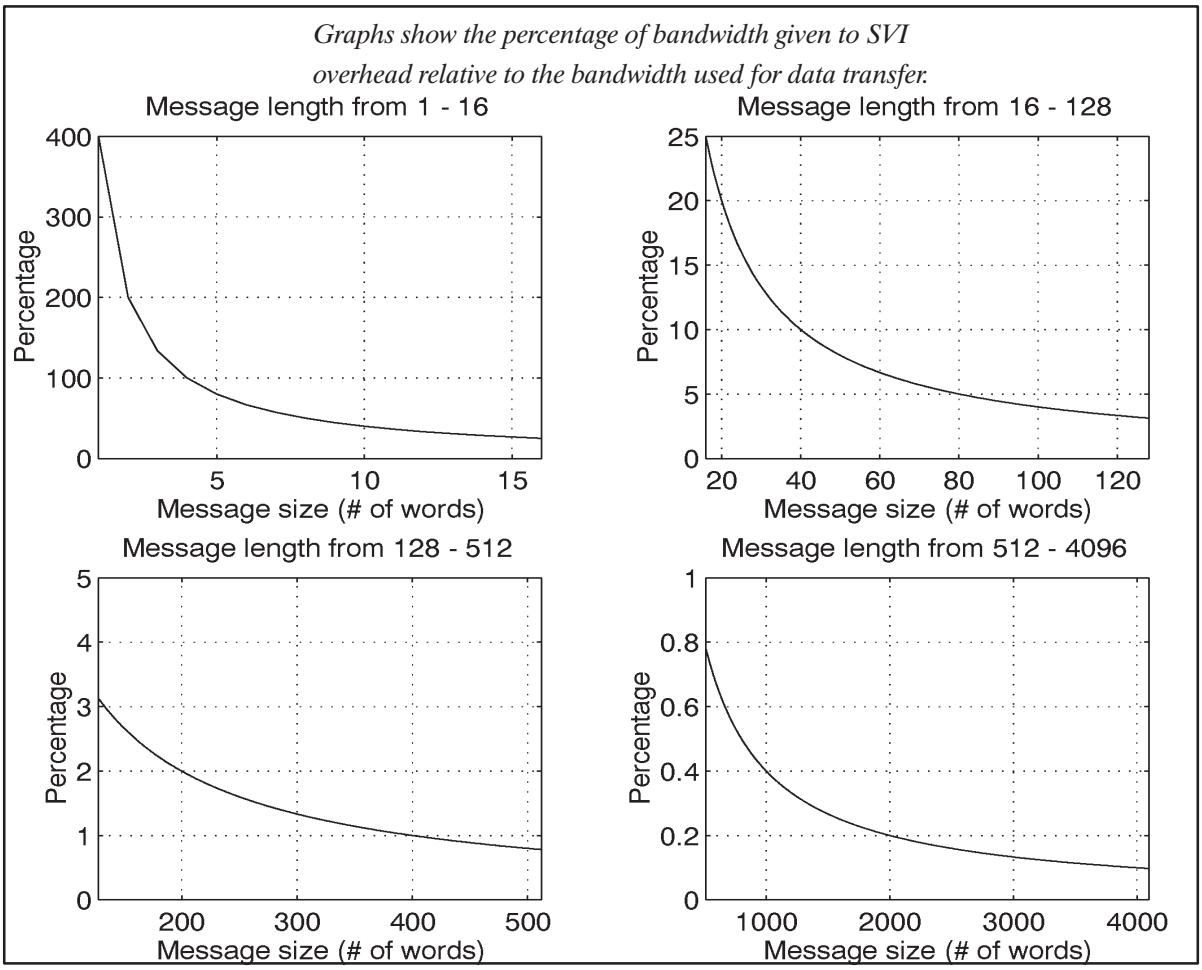

Figure 7.: Performance of SVI as Message Length Varies

relative to *data transferred*. It shows that as the number of words per message (or transaction) increases, the percentage of the bandwidth used for overhead message decreases relative to the amount of bandwidth allotted to data transfer. This is because as the number of words per message increases, the number of transactions that have to be performed for a given word size decreases, and therefore so does amount of SVI overhead because there is a fixed 4 cycles per transaction latency.

Figure 7. shows that for a case where 4096 words are to be transferred across an SVI, as the message word increases the percentage of bandwidth used for overhead decreases. This figure shows four graphs to show the relative resolution at different message sizes. It can be seen how the message length can vary the performance of the SVI because the graphs show the percentage of overhead bandwidth varying from 400% to less than .1% of the bandwidth allotted for data transfer.

6. Summary

In this SVI verification study, a Benchmark II − Data I/O board was retro−fitted with an SVI. Models for these encapsulations were implemented in VHDL and verified. The net result is that SVI can be implemented with little effect to latency in a *data sensor−type* system with little or no latency overhead and with minimal hardware overhead.

It was concluded that an SVI *wraps* well around components that provide *FIFO−like* data flow. In other words, a scenario in which streams of data are burst on to a *fabric interface* whose sole purpose is to transport data is well suited for SVI. This concept became apparent when attempting to encapsulate the Rino chipset. Even though the Rino supports a FIFO−like interface, it performs many link lists control operations that do not lend themselves to strict data flow. These other operations are *real−time* data control that make an SVI implementation cumbersome. To reduce the complexity in the Data I/O − Rino case, the transmit and receive FIFOs were partitioned and the real−time controls were handled within the respective encapsulations.

7. Recommendations

During the course of this verification study, many evaluations of SVI candidates (i.e., processor/board/product/etc.) were performed. Also, since the SVI specification is a *living document* it was under constant evaluation. The result of these evaluations led to the Data I/O board as the candidate for this verification study. As part of this study, a great deal of effort was exerted into determining how and where to split the Data I/O board for the SVI. From the above tasks of this study, two main concepts were *brought to light* and are presented in this section as recommendations for future SVI encapsulations.

7.1. SVI Well Suited for Data Streaming

From the summary in the previous section, it can be implied that the Data I/O board was not originally intended to be retro−fitted with an SVI. Many network interface signals were woven into custom FPGAs that connected to real−time status signals between processing elements and data flow elements. However after developing an implementation for the Data I/O board, a more thorough understanding of SVI was gained.

SVI is well suited for fabric interfaces and processing elements that support *streaming data*. A fabric interface that performs network *and* data control may not necessarily be a robust candidate for an SVI encapsulation. As seen in the case with the Rino chipset, the FIFOs had to be partitioned in order to maintain data synchronicity within the Data I/O board.

After having gained experience with fabric interfaces that required a FIFO port before an SVI could be implemented, it is recommended that in future encapsulations candidates that strictly support data flow be chosen. This recommendation is being made in order that the conclusion about *data streaming elements* be verified, as well to suggest a method for providing another *type* of encapsulation that would provide for an even better and more thorough understanding of SVI and its system effects. An SVI candidate as the one suggested should try to measure the effects of SVI under situations of bus contention, interrupt handling, *read and write scenarios*, etc.

7.2. Need for a *Stop Read* Command

After evaluating the SVI specification, it has become apparent that there is a need for an addition of another SVI command. This command would serve as a compliment to the SVI *External Read Request*. The proposed command will be referenced as the *External Stop Read Request*.

The external stop read request would provide a slave the ability to stop a read transfer that is in progress due to the *External Read Request* command. It can be assumed that if a read request is issued then the slave in the *issuing SVI* is controlling the interface. Without the external stop read request, the controlling slave has no method in which it can stop a read transaction that is in progress. It is true that the slave can deassert its *ready* signal, but even at that the SVI is active while no transfers are occurring. Without the external stop read request, only a master can completely end a transfer, however this will never occur in many cases because the master is not in control of the interface.

With the external stop read request, the controlling slave can notify its master that the read transaction is to be stopped. The master will send an SVI message across the SVI notifying its slave to notify the master that is sourcing data to the controlling data to assert its *last word* signal and end the transaction.

LOCKHEED MARTIN

Advanced Technology Laboratories
1 Federal Street − A&E 2W
Camden, NJ 08102
(609) 338−4250

Date: May 17, 1996

Author: Amit Chhabra

Contact: Janet W. Wedgwood (janet.e.wedgwood@lmco.com)

1. Scope

This document serves to provide a complete description for the *SVI Verification Study* involving the *Rino IC−Data I/O Board* encapsulation.

2. Referenced Documents

2.1. Vendor Documents

Developers Guide: The RIC−RINO Chipset, Mercury Computers

RACEway Interlink − Data and Physical Layers, VITA, Feb. 1995, Rev. 1.4.

CMOS SyncFIFO (IDT722xx7B) − Specifications, Integrated Device Technology, Inc., Aug. 1993.

Programmable Skew Clock Buffer (CY7B991) − Specifications, Cypress Semiconductor Corporation, Feb. 1995.

2.2. ATL Documents

Standard Virtual Interface Specification, Version 0.5, LM − ATL, Jun. 1995.

DATA I/O Board − Benchmark II Diagrams.

3. Verification Objective

The objective of this verification study is to create and demonstrate a VHDL model of the *Standard Virtual Interface (SVI)* as specified by the *SVI Specification* and to implement this model using the existing *SVI VHDL templates*. The design and implementation of the SVI model in VHDL for particular encapsulation is to serve as a method of quantifying the merit of the SVI concept, the quality of the SVI development process, and to provide an understanding of where some of the difficulties and concerns in developing an SVI encapsulation exist. Furthermore, the VHDL model is meant to provide an entity from which the SVI overhead and performance can be measured.

For this verification study, the RASSP Benchmark II – *Data I/O Board* (as shown in Figure 1.) was used. An SVI interface was provided between the Data I/O board its network inter-

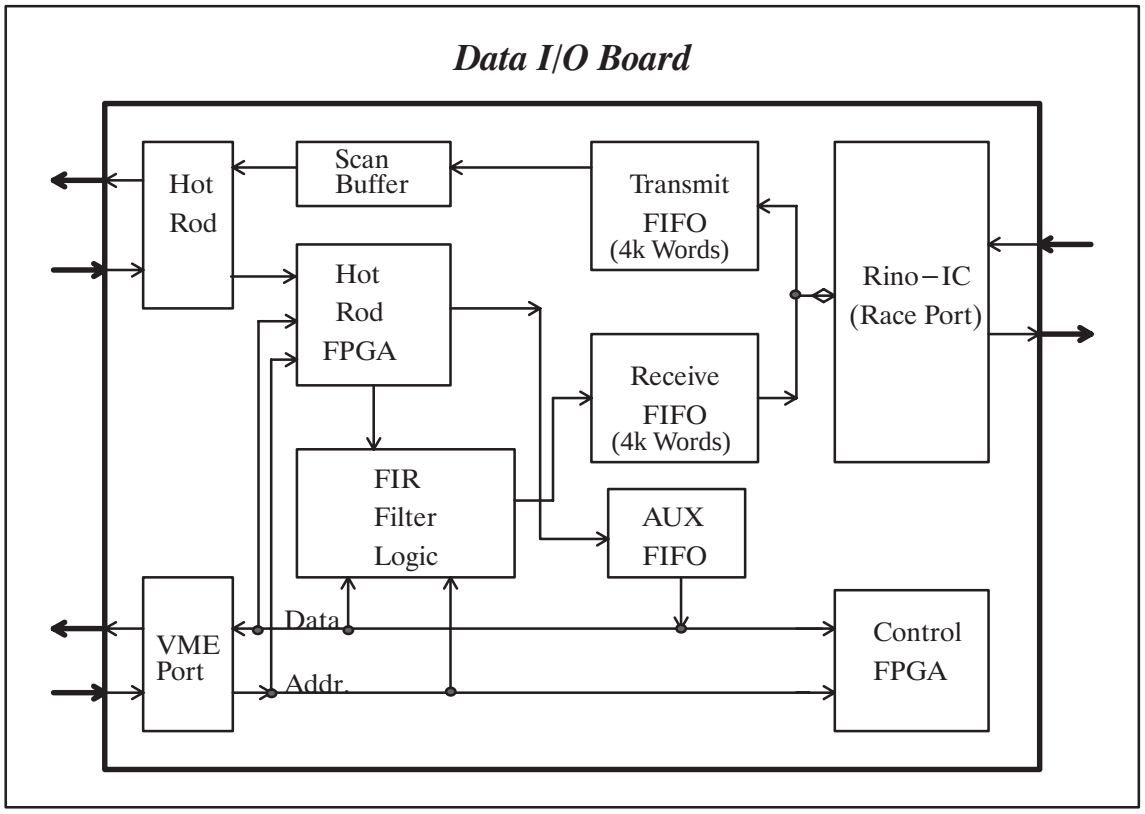

Figure 1.: Data I/O Board before the SVI Encapsulation

face, the *Mercury Rino–IC*. The SVI interface was created by effectively dividing the Data I/O board at its transmit and receive FIFOs as shown in Figure 2.. The design of the SVI

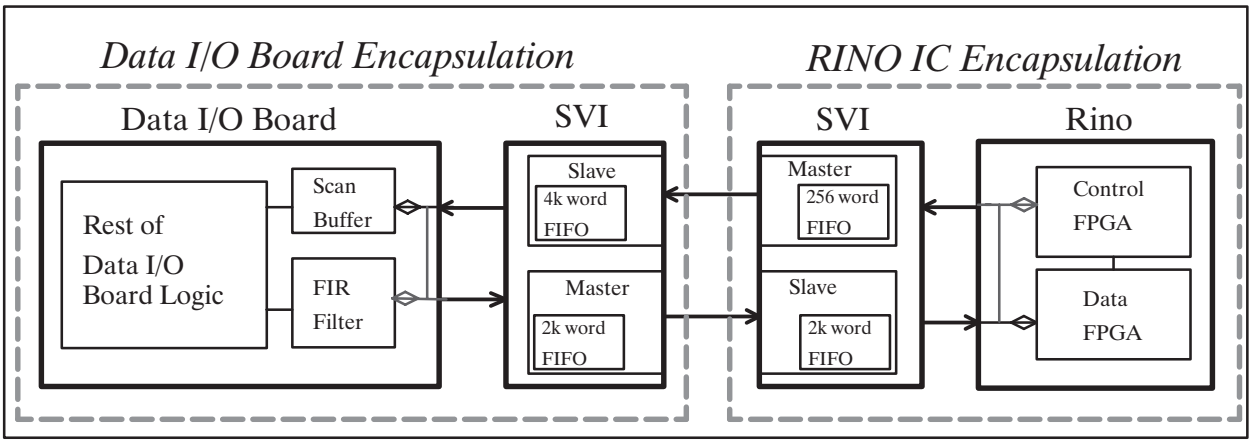

Figure 2.: Data I/O Board and Rino Chipset Encapsulated

encapsulation is discussed in the next section.

For practical purposes, the SVI encapsulations of the Data I/O Board and the Rino chipset will serve as components in the *Model Year Architecture (MYA)* reuse library. The VHDL models will be available as *plug and play* elements that can be connected with other SVI encapsulated elements. The availability of these models will provide the ability for further performance studies that will be conducted in order to fully develop the SVI concept for the MYA task as part of the RASSP Program.

4. Data I/O Board – Rino IC Encapsulation Design

Both the *master* and *slave* entities in each of the SVI wrappers shown in Figure 2. can be described in the two parts, an *SVI controller* (implemented as a state machine) and a FIFO (as shown in Figure 3.). The SVI controller and the FIFO are used to maintain a consisten-

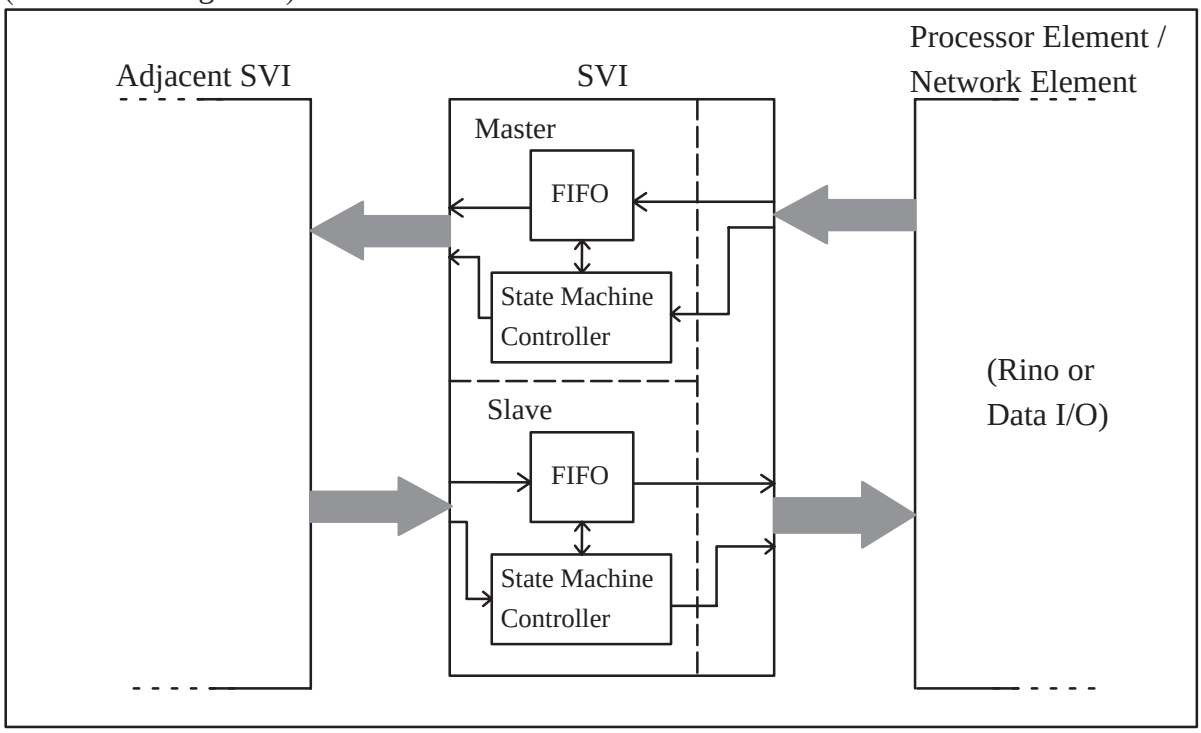

Figure 3.: SVI FIFO and State Machine Entities

cy of the data flow between the attached SVI and the encapsulated element (this is further described in the SVI Specification).

Since the implementation goal of this study was to separate and encapsulate the Data I/O board (the processing element) and the Rino chipset (the network element), it was clear that the division would be at transmit and receive FIFOs on the Data I/O board. In this way the interface from the Rino chipset to the Data I/O board would be a “straight forward” FIFO interface. Based on the notion that each SVI wrapper minimally called for a FIFO to provide a temporary storage for *slave data*, a FIFO was incorporated into each wrapper. In an effort to reduce complexity, the transmit and receive FIFOs of the Data I/O board were merged into the FIFOs in the SVI wrappers and removed from the Data I/O board (as show in Figure 2.). In order to successfully implement the merging of the FIFOs, the FIFO status flags and control signals that connected to the Data I/O board and the Rino chipset had to be maintained, and the following section further details this topic.

4.1. Data I/O Board FIFOs

Figure 4. shows the Data I/O board and Rino chipset connections to the transmit and re-

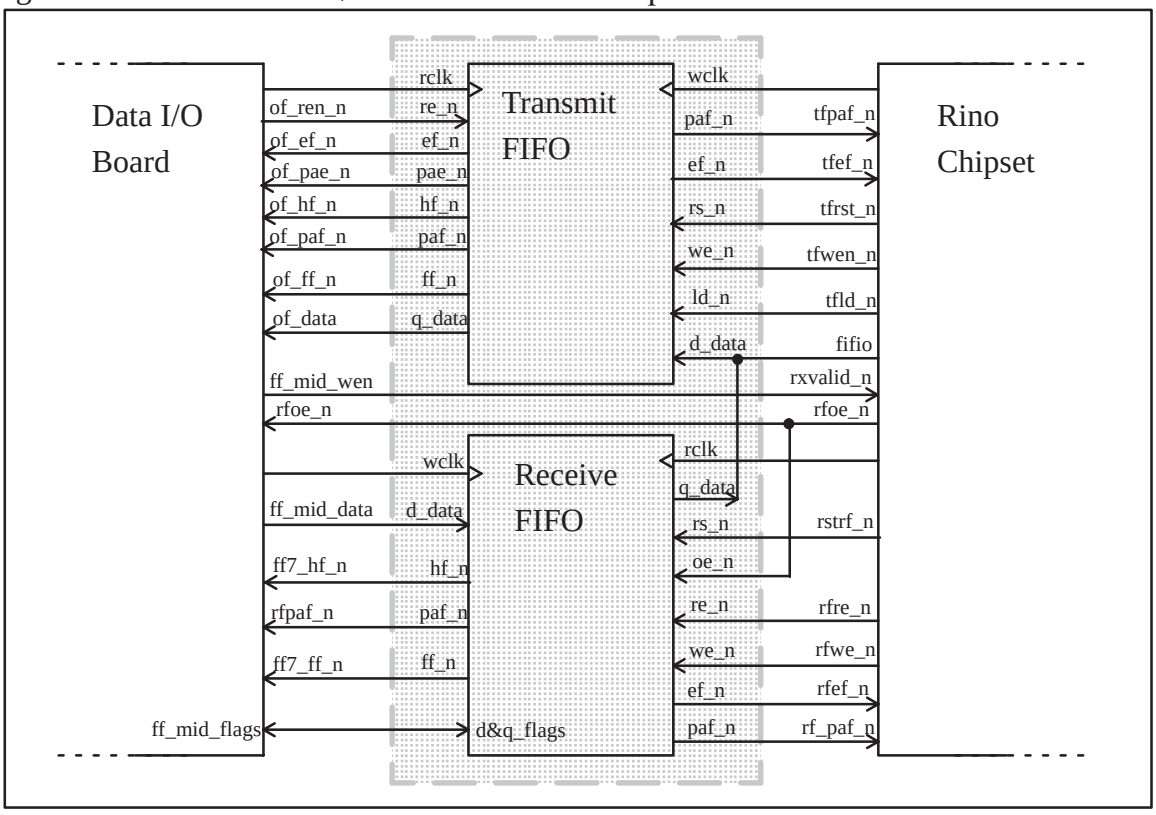

Figure 4.: Data I/O Board FIFO Configuration before SVI

ceive FIFOs before the SVI encapsulations were added to the system. These connections were maintained by implementing the SVI wrappers as shown in Figure 5. and Figure 6..

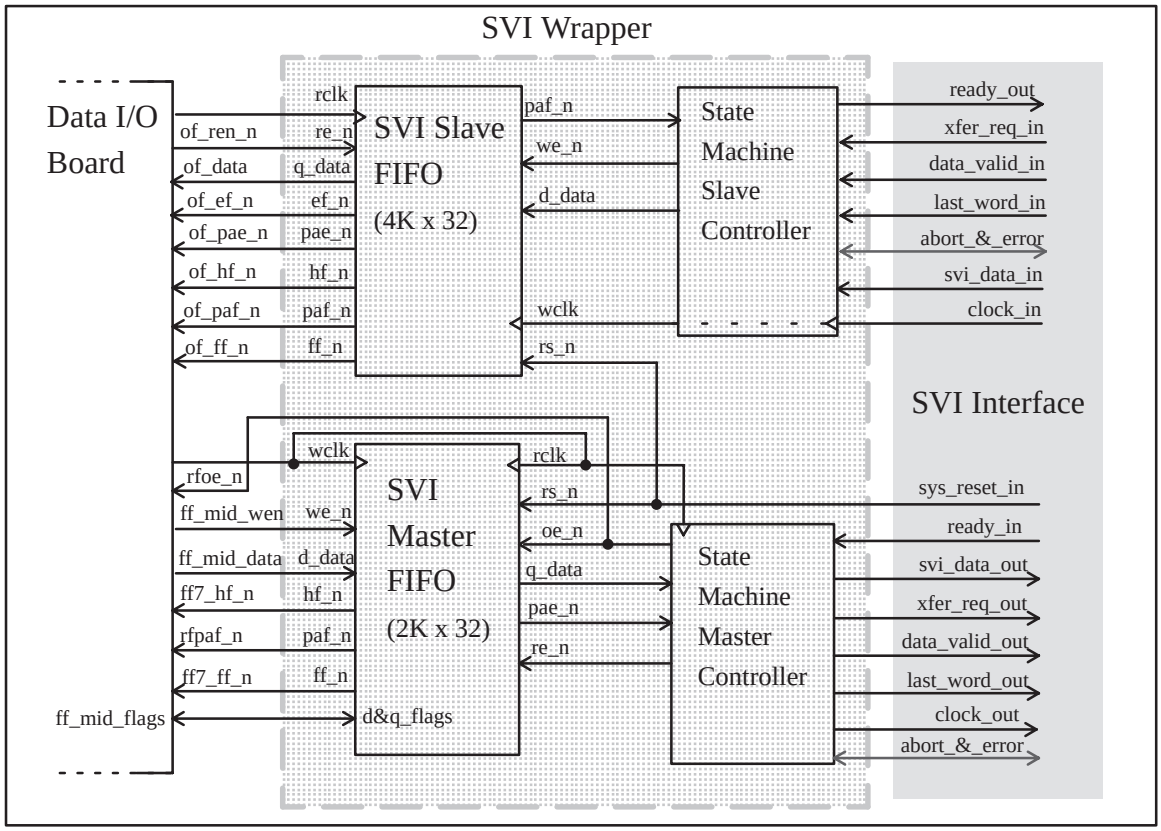

Figure 5.: Data I/O Board Wrapper

4.2. State Machine Controller Design

The *state machine controllers* implemented in each SVI entity provide control their respective FIFO in the interface. A master controller interprets signals from either the Data I/O board or the Rino chipset and translates them into FIFO flags and SVI signals. A slave controller interprets SVI signals and translates them into FIFO flags and Data I/O board or the Rino chipset signals.

4.2.1. Transmit FIFO Control

In the SVI architecture, the *transmit FIFO* is made up of a 256 word and a 4Kword FIFO. The state machine controller for the Rino's master only initiates an SVI transfer if its FIFO is half full (or its FIFOs *half full flag* becomes asserted). Once the controller initiates a transfer, it waits for the *SVI ready* to become asserted, and then it will *stream* data out of its FIFO by asserting the FIFO's *read enable* line. This will continue until the FIFO becomes empty or Data I/O board's SVI deasserts the ready signal. Essentially, the Rino's master controller uses the *SVI ready in* to control its FIFOs read enable signal. An end of transaction can only occur when the FIFO becomes empty. The *almost empty flag* from the FIFO is used to signal the controller to assert the *last word* signal.

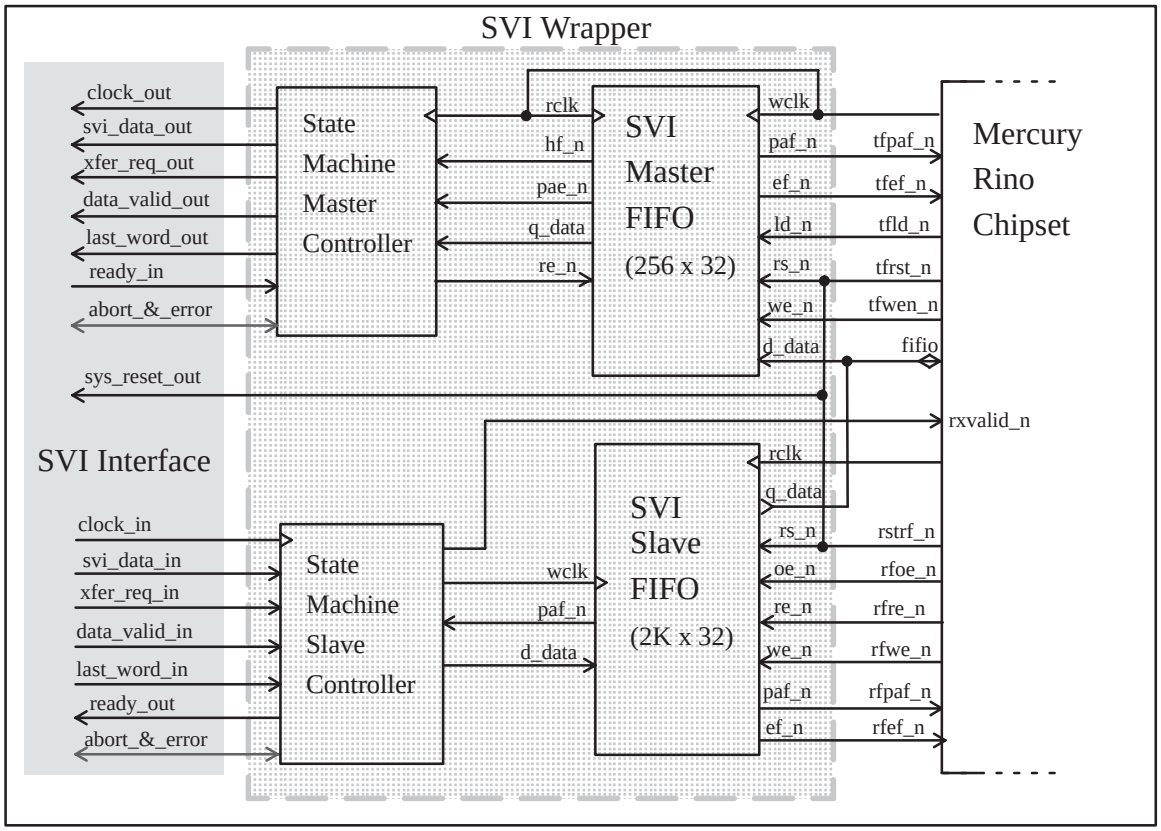

Figure 6.: Rino Chipset Wrapper

The Data I/O board's slave's controller will assert an *SVI ready* in response to a *transfer request* only if its FIFO has less then 3968 words filled (3968 words is default number of words that it takes to trigger the *programmable almost full flag* on the FIFO, see the IDT 72245b documentation referenced). Once a transfer is initiated, the slave's controller will pass the data from the SVI data lines to its FIFO until its FIFO becomes *almost full* (or its *almost full flag* is asserted) or the transaction is ended by the master. The slave controller uses the *SVI data valid* signal to control its FIFO's *write enable* signal (i.e., as long as the data is valid, the controller will keep the write enable asserted).

A more detailed description of the state machine controller for these FIFOs can be obtained by examining the VHDL code for the Data I/O board Wrapper. The code shows exactly how the above description is implemented, along with a description of the *reset* and *load* flags.

4.2.2. Receive FIFO Control

In the SVI architecture, the *receive FIFO* is made up of two 2K word FIFOs. The state machine controller for the Data I/O board's master only initiates an SVI transfer after its FIFO has filled 512 words of its space (or its FIFOs *almost empty flag* is deasserted) or the last set of *radar dwell data* (that is less than 512 words) is resident in its FIFO. Once a transfer is initiated, the master's controller will stream data from its FIFO by asserting the FIFO's *read enable* line. The controller will continue to send data out the *SVI data lines* until its FIFO becomes empty or the Rino's SVI deasserts its *SVI ready*. An end of a transaction can only occur when Data I/O board's master FIFO becomes empty. The *almost empty flag* from the FIFO is used to notify the controller to assert its *last word* signal.

The Rino's slave's controller will assert an *SVI ready* in response to a *transfer request* only if its FIFO is not *almost full*. Once a transfer is initiated, the slave's controller will pass data from the SVI data lines to its FIFO until it becomes *almost full* (or its *almost full flag* is asserted) or the transaction is ended by the master. The slave controller for the Rino will assert its FIFO's *write enable* signal as long as the *SVI data valid* signal from the Data I/O board's SVI remains asserted.

A more detailed description of the state machine controller for these FIFOs can be obtained by examining the RACE_RINO VHDL code. The code shows exactly how the above description is implemented, along with a description of the *reset* and *load* flags.

4.3. VHDL Implementation Notes

This section details the use of *VHDL Generics* in the Data I/O board and the Rino chipset encapsulations. Below, a reason for why generics are needed and how they are used in these encapsulations is provided.

The VHDL implementation of both the Data I/O board and Rino chipset encapsulations have been simulated in a testbench that forces data to flow between the encapsulations across an SVI. In this case, the operating clock frequencies are determined by the requirements of the Data I/O board and the Rino chipset. Therefore, it can be anticipated that in a configuration (or testbench) in which these encapsulations are not connected to each other (but to other SVI encapsulated components), the operating clock frequency of the SVI may change. If the operating clock frequency change, it will become important to appropriately adjust the value loaded as the *programmable almost empty* and *programmable almost full* in the FIFOs (see the IDT722x5b specifications for more details on FIFO status flags and the VHDL code to see how to implement this change). The VHDL implementations on these models facilitate the flexibility of adjusting the programmable load values for the FIFOs because the values for the almost full and empty flags have been implemented as *generics*. This allows the user to enter in a value for the particular load value during *runtime* (or in script that calls the simulation tool). If no values are provided during runtime, the VHDL models will default to the case for which the Data I/O board and the Rino chipset are connected together.

5. SVI Encapsulation Performance Results

In this verification study, it was proven that an existing radar data board could be retro−fitted with an SVI and still perform as intended. However, the tradeoff for *plug n' play* flexibility is an increase in hardware and overall latency.

5.1. Hardware Overhead

For this pair of encapsulations, the hardware is altered by replacing four 4K by 18 bit FIFOs by two 256 by 18 bit FIFOs, four 2K by 18 bit FIFOs, two 4K by 18 bit FIFOs, and a controller FPGA (that implements all four state machines). The net increase in the amount FIFO storage is the two 256 word FIFOs and an FPGA. However, because of the re−partitioning of FIFOs, the amount of board space required by these components will be about twice that of the four FIFOs in the non−encapsulated case.

5.2. Latency Overhead

The increase in latency for the *dataio_loop_tb.hdl* testbench is listed below:

Time for the first data element to make through the loop:
Original Data I/O Board: *490,600 ns*
SVI Encapsulated: *493,950 ns*
Difference: *3350 ns or* ***.68% increased latency***.

Time for one dwell of data to flow through the transmit FIFO (or SVI equivalent of)
Original Data I/O Board: *470,777 ns*
SVI Encapsulated: *472,527 ns*
Difference: *1750 ns or* ***.37% increased latency***.

Time for one dwell of data to flow through the receive FIFO (or SVI equivalent of):
Original Data I/O Board: *429,450 ns*
SVI Encapsulated: *432,488 ns*
Difference: *3038 ns or* ***.71% increased latency***.

From these encapsulations, it can be see that the increase in latency due to SVI is very insignificant compared to the time that data spends in other parts of the Data I/O board system. The location of the SVI between the Data I/O board and the Rino is not a *bottle−neck* point within the system. So, that even with the insignificant 4 cycles per SVI transaction that is added to Data I/O *pipeline*, there are other parts of the Data I/O board that slow down the data flow rate. In other words, the even though a delay is added to the system by the SVI, it is insignificant because the parts of the Data I/O board that receive data from the SVI wrapper are not ready to accept because operate at a fraction of the data rate than that of the wrapper (however, a 4 cycle latency will be added for the first word).

5.3. Performance Versus Message Length

In the above sections it has been shown how a system retro−fitted with an SVI is effected in terms of hardware and latency. This section provides an insight at how the latency overhead of can vary depending on the number words in each SVI transfer.

Figure 7. shows a graphs that describes the percentage of bandwidth used for *SVI overhead*

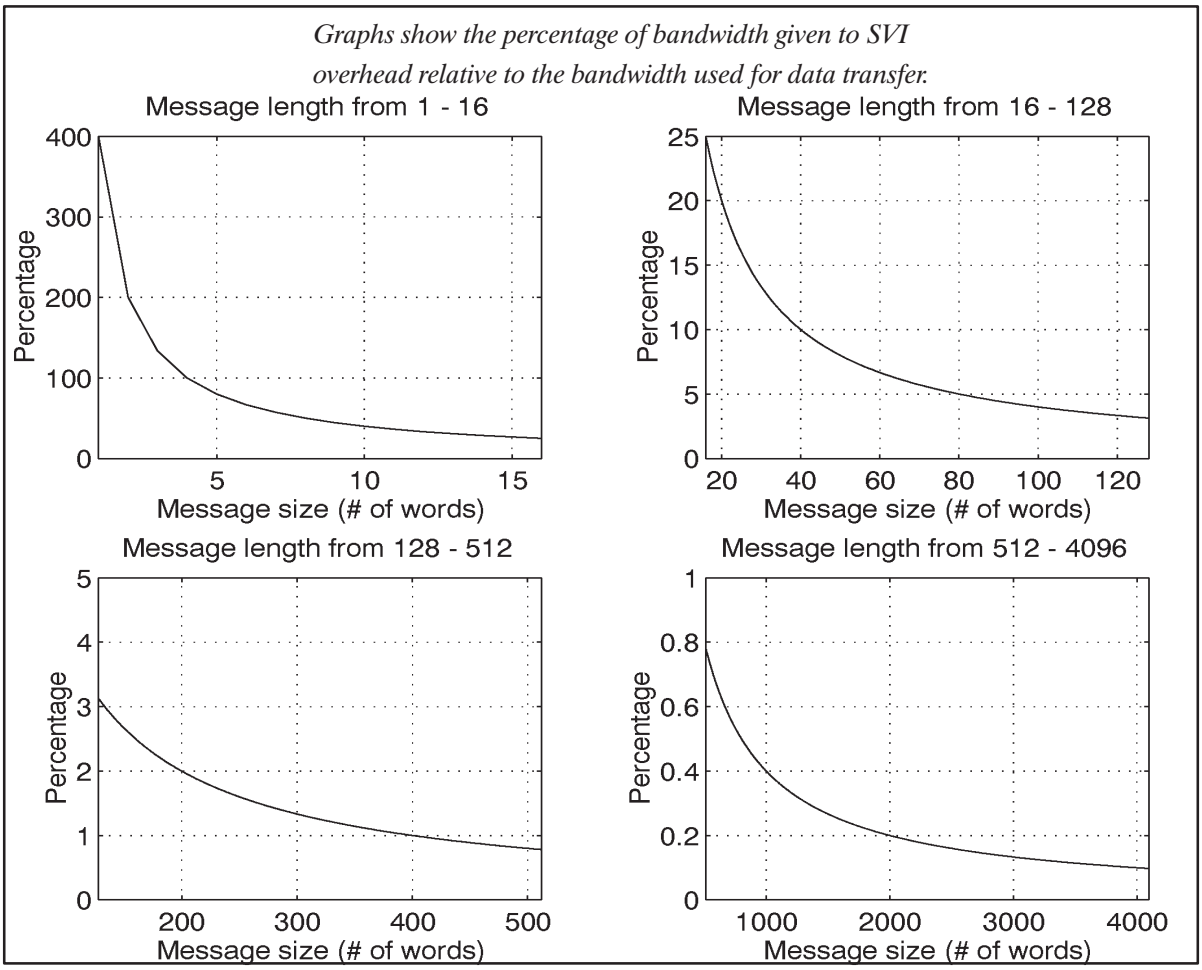

Figure 7.: Performance of SVI as Message Length Varies

relative to *data transferred*. It shows that as the number of words per message (or transaction) increases, the percentage of the bandwidth used for overhead message decreases relative to the amount of bandwidth allotted to data transfer. This is because as the number of words per message increases, the number of transactions that have to be performed for a given word size decreases, and therefore so does amount of SVI overhead because there is a fixed 4 cycles per transaction latency.

Figure 7. shows that for a case where 4096 words are to be transferred across an SVI, as the message word increases the percentage of bandwidth used for overhead decreases. This figure shows four graphs to show the relative resolution at different message sizes. It can be seen how the message length can vary the performance of the SVI because the graphs show the percentage of overhead bandwidth varying from 400% to less than .1% of the bandwidth allotted for data transfer.

6. Summary

In this SVI verification study, a Benchmark II − Data I/O board was retro−fitted with an SVI. Models for these encapsulations were implemented in VHDL and verified. The net result is that SVI can be implemented with little effect to latency in a *data sensor−type* system with little or no latency overhead and with minimal hardware overhead.

It was concluded that an SVI *wraps* well around components that provide *FIFO−like* data flow. In other words, a scenario in which streams of data are burst on to a *fabric interface* whose sole purpose is to transport data is well suited for SVI. This concept became apparent when attempting to encapsulate the Rino chipset. Even though the Rino supports a FIFO−like interface, it performs many link lists control operations that do not lend themselves to strict data flow. These other operations are *real−time* data control that make an SVI implementation cumbersome. To reduce the complexity in the Data I/O − Rino case, the transmit and receive FIFOs were partitioned and the real−time controls were handled within the respective encapsulations.

7. Recommendations

During the course of this verification study, many evaluations of SVI candidates (i.e., processor/board/product/etc.) were performed. Also, since the SVI specification is a *living document* it was under constant evaluation. The result of these evaluations led to the Data I/O board as the candidate for this verification study. As part of this study, a great deal of effort was exerted into determining how and where to split the Data I/O board for the SVI. From the above tasks of this study, two main concepts were *brought to light* and are presented in this section as recommendations for future SVI encapsulations.

7.1. SVI Well Suited for Data Streaming

From the summary in the previous section, it can be implied that the Data I/O board was not originally intended to be retro−fitted with an SVI. Many network interface signals were woven into custom FPGAs that connected to real−time status signals between processing elements and data flow elements. However after developing an implementation for the Data I/O board, a more thorough understanding of SVI was gained.

SVI is well suited for fabric interfaces and processing elements that support *streaming data*. A fabric interface that performs network *and* data control may not necessarily be a robust candidate for an SVI encapsulation. As seen in the case with the Rino chipset, the FIFOs had to be partitioned in order to maintain data synchronicity within the Data I/O board.

After having gained experience with fabric interfaces that required a FIFO port before an SVI could be implemented, it is recommended that in future encapsulations candidates that strictly support data flow be chosen. This recommendation is being made in order that the conclusion about *data streaming elements* be verified, as well to suggest a method for providing another *type* of encapsulation that would provide for an even better and more thorough understanding of SVI and its system effects. An SVI candidate as the one suggested should try to measure the effects of SVI under situations of bus contention, interrupt handling, *read and write scenarios*, etc.

7.2. Need for a *Stop Read* Command

After evaluating the SVI specification, it has become apparent that there is a need for an addition of another SVI command. This command would serve as a compliment to the SVI *External Read Request*. The proposed command will be referenced as the *External Stop Read Request*.

The external stop read request would provide a slave the ability to stop a read transfer that is in progress due to the *External Read Request* command. It can be assumed that if a read request is issued then the slave in the *issuing SVI* is controlling the interface. Without the external stop read request, the controlling slave has no method in which it can stop a read transaction that is in progress. It is true that the slave can deassert its *ready* signal, but even at that the SVI is active while no transfers are occurring. Without the external stop read request, only a master can completely end a transfer, however this will never occur in many cases because the master is not in control of the interface.

With the external stop read request, the controlling slave can notify its master that the read transaction is to be stopped. The master will send an SVI message across the SVI notifying its slave to notify the master that is sourcing data to the controlling data to assert its *last word* signal and end the transaction.