Reconfigurable Network Interface (RNI) Guideline

Myrinet to PCI RNI

8/97

1.0 Introduction

1.1 Description

This document contains a description of a particular Reconfigurable Network Interface (RNI) which was developed as part of the hardware verification study under the Model Year Architecture task of the DARPA-sponsored Rapid Prototyping of Application Specific Signal Processors (RASSP) program. The purpose of this document is to aid designers in the understanding and use of the RNI and describe the considerations which drive the design structure.

1.2 Objective

The objective of this verification task is to create and demonstrate a VHDL example of a programmable Reconfigurable Network Interface (RNI) element using the Standard Virtual Interface (SVI) technology that has been defined in RASSP Model Year Architecture (MYA). In this case, the RNI concept is demonstrated in an application which links the Myrinet network [MYRI 1994] to the PCI interconnect fabric. This guideline presents the minimal functionality necessary to implement a programmable bridge. It is not meant to constrain the user or predetermine how various functions must be implemented, but rather gives one example of an implementation. This document provides general guidelines for designers who wish to use this RNI to bridge any two networks.

2.0 Related Documents

In order to understand this guideline it is important to have a working knowledge of the RASSP MYA concepts and vocabulary. For background and introductory information, the following documents are available on the Lockheed Martin ATL external web site at http://www.atl.external.lmco.com/projects/rassp/papers/MYA/mya_papers.html:

MYA Specification:

RASSP Model Year Architecture Specification Volume I:

Introduction Version 1.0 [LMATL 1996a]

RASSP Model Year Architecture Specification Volume II:

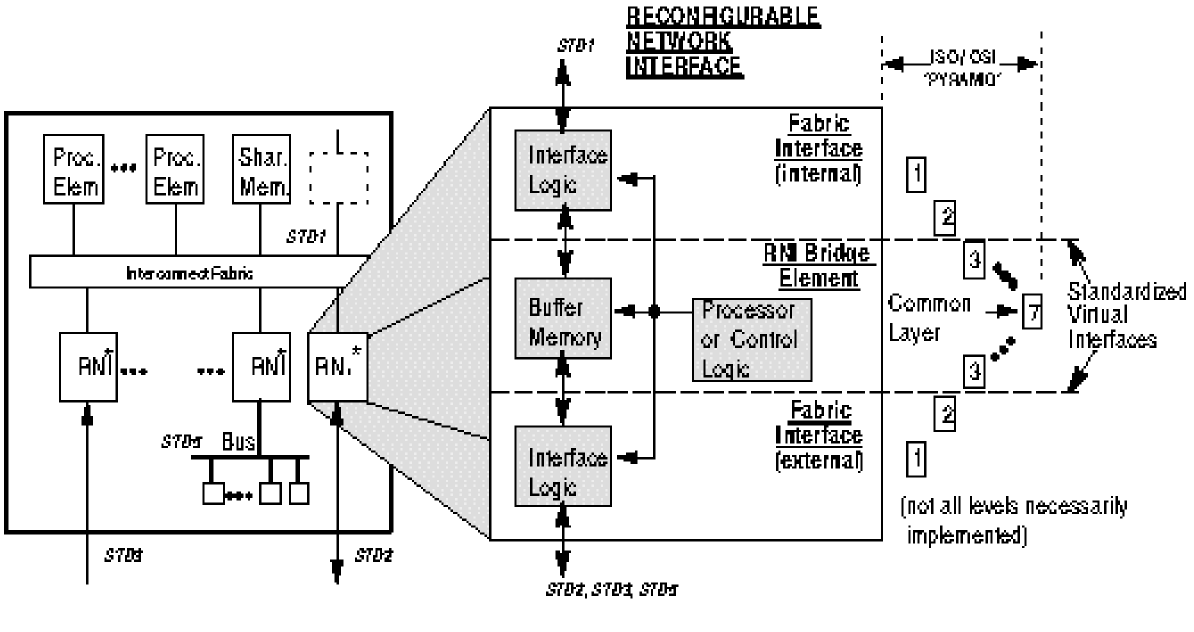
Hardware Architecture Element Specification Version 1.0 [LMATL 1996b]

In addition, there are several application notes describing other efforts of the MYA task. Copies of these documents are available on the Lockheed Martin ATL external web site at:

http://www.atl.external.lmco.com/projects/rassp/legacy/appnotes/MYA/index.html

3.0 RNI Overview

The RNI, shown in Figure 1, is divided into three logical elements: 1) fabric interface (internal), 2) RNI Bridge Element, and 3) fabric interface (external). The fabric interfaces are the specific interfaces between both the internal interconnect fabric and the Standard Virtual Interface (SVI) [LMATL 1996b], and the external interconnect fabric and the SVI. The actual bridging function is performed by the RNI bridge element, which consists of a buffer memory to facilitate asynchronous coupling between the two interfaces, and a controller which coordinates data transfers and provides flow control. The bridge element can be implemented via custom logic (e.g. FPGA, ASIC) or a programmable processor. An element whose interface has been translated to the SVI is said to be encapsulated. These encapsulations are written in synthesizable VHDL.



Fabric Interface (internal) – Implements lowest levels of Interconnect Fabric (e.g. ISO/OSI 1. Physical, 2. Data Link)
RNI Bridge Element – Handles overall control and data buffering of RNI and (optionally) higher level protocols or protocol conversion.
Fabric Interface (external) – Implements lowest levels of External Network (e.g. ISO/OSI 1. Physical, 2. Data Link)

Figure 1 Model Year Architecture Reconfigurable Network Interface

As in the case of Internal Module interfaces [LMATL 1996a], which connect processor elements to fabric interfaces, a layered communication approach is employed. Here, since the RNI provides a bridge between two interfaces, two separate layered structures exist. Shown in Figure 1, the two layered structures form a pyramid, with the lowest layers of the two interfaces implemented in the fabric interface components. The higher levels of each structure are implemented within the RNI bridge element and converge where the data interchange is stripped of all its interface specific identity by the lower layers, effectively performing a protocol conversion. The ISO/OSI model is being used as a reference and does not imply that the layering ultimately employed in defining the internal node architecture will strictly follow this model or implement all seven layers.

Two categories of the RNI can be defined based upon the bridge element implementation: the custom RNI - employing a custom hardware bridge element implementation -and the programmable RNI - employing a microprocessor - based bridge element. In general, either implementation may be chosen for a given application. The custom RNI will tend toward higher design complexity with lower latency, while the programmable RNI will tend toward lower design complexity and higher latency, but will also offer greater opportunity for reuse due to programmability. The RASSP system designer must make the necessary performance/complexity tradeoffs for each particular application.

The greater the difference in the protocols of the interconnect fabrics which are being bridged, the greater the complexity of a custom hardware bridge element design. When bridging between very different protocols, the bridging function may be more easily implemented with a programmable processor rather than with a complicated custom hardware design. However, in cases where low latency is critical, the DARPA-sponsored Virtual Microarchitecture Interface (VMI) [LMATL 1997a] or a custom hardware approach will be more attractive. The VMI provides a technology independent, functional standard interface between two networks. Its advantage over the programmable RNI is that it provides up to 100x lower latency due to its hardware intensive implementation. Its advantage over a custom hardware RNI design is that, by extending the functional interface of the SVI up to OSI layer 3, it provides a higher-level of interoperability than the RNI which is based solely on the layer 1-2 SVI.

Considering the RASSP paradigm of reuse, the programmable RNI or VMI is preferred over a custom approach. If one wanted to reuse or upgrade the internal or the external SVI encapsulated fabric interface of an RNI or VMI, one can simply remove the original SVI encapsulated interface and replace it with a new encapsulated interface. The RNI would require reprogramming of the bridge element processor and the VMI would require new state machines to perform the new protocol conversions.

The purpose of the RNI is to provide a bridge between two networks each with their own network protocol. The RNI has several tasks which it may need to perform. This includes being able to interpret the header of a message from one network, extract the addressing information, create a header for the message for the second network, and transform the address information to routing expressed in the protocol of the second network. In addition, it must recognize the end of a message so that it can properly recognize and manipulate the header of the next message. It must insert whatever pad bytes are needed to align the data and modify or calculate any tailers needed to satisfy the network protocol on the receiving side.

The RNI also has to transmit the data from one network to the other. This transmission may involve several other functions including:

- Data format and endianness translation
- Transaction type recognition and encoding
- Error detection and error code generation
- Segmentation related services
- Priority/Pre-emption
- Blocking timeouts

Although it is anticipated that some data conversion will generally be necessary, this instantiation of the RNI does not address data conversion. This will necessarily make the throughput estimates optimistic. However, the header translation function has been coded and timed using a MC68040 processor. The times are incorporated into the simulation of the bridge.

In this RNI example, there are 3 functional blocks. Each block is interconnected via an SVI interface as shown in Figure 2. Fabric A is a PCI-like interface fabric called BIN-PCI. Fabric B is the Myrinet [MYRI 1994]. The blocks are:

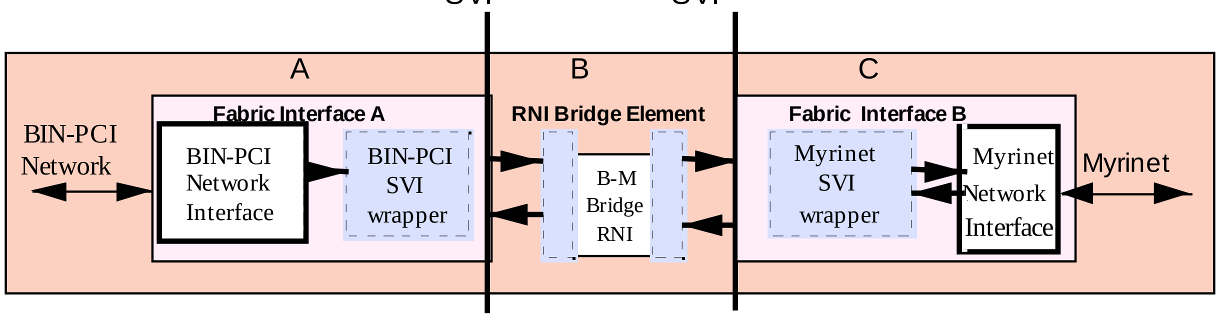


Figure 2 BIN-PCI to Myrinet RNI

A. Fabric Interface A - consisting of an SVI encapsulated BIN-PCI interface. This connection forms a half duplex network that transmits and receives 64-bit words. Since the RNI bridge element transfers 32-bit words, data flowing in the PCI to RNI direction is formatted using an SVI 2-to-1 converter (convert from 2 words wide to 1 word wide) in the SVI slave of the Bridge Element and data flowing in the RNI to PCI direction is formatted using an SVI 1-to-2 converter (convert from 1 word wide to 2 words wide)in the PCI fabric interface slave.

B. The RNI Bridge Element - consisting of 7 functional blocks as shown in Figure 3:

1) SVI wrappers, including four 32-bit wide FIFO blocks for buffering data flowing from one Fabric Interface to the other

2) Network Engine (NE) which effects the transfer of data from one side of the bridge to the other by servicing interrupts, and reading data in from the _IN FIFOs, performing header and possibly data translation, and writing data out to the _OUT FIFOs.

3) Interrupt PAL which detects all of the possible interrupts and generates the interrupt vectors

4) Address Decode PAL, which generates chip selects, and read and write enables for the FIFO, controls the direction of Data Buffer and tracks the state as the last word passes through the bridge

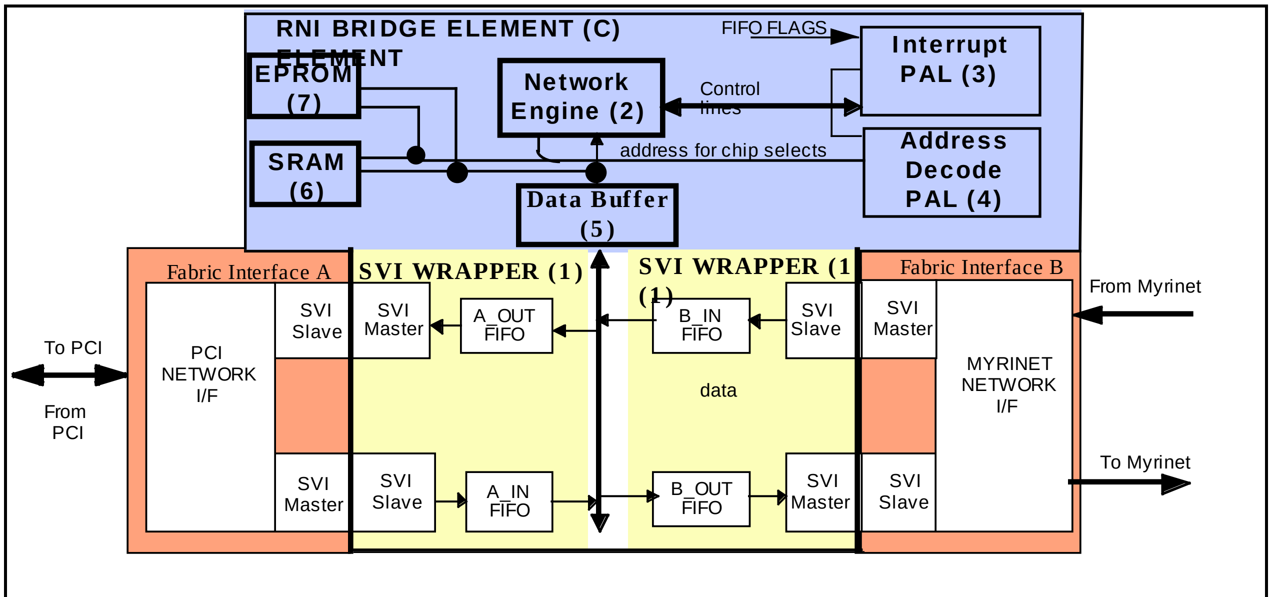


Figure 3 Implementation of BIN-PCI to Myrinet RNI Bridge Element

5) Data Buffer, which buffers the bi-directional data between the FIFOs and the NE

6) SRAM, which stores the header addresses route translations and application code

7) the EPROM, for program memory and initialization data.

C. Fabric Interface B - consisting of an SVI encapsulated Myrinet Network. The connection is a full duplex network that transmits and receives 8-bit words. Since the Myrinet Interface and the RNI bridge node are both 32 bit word interfaces, no conversion has to be made in order to transmit to and from these two interfaces.

The blocks are implemented in a virtual prototype as follows:

Block Implemented	**Code Used**
Myrinet Interface	VHDL code
PCI Interface	VHDL code
Network Engine: MC68040	Synopsys Hardware Verification SmartModel of the MC68040
Network Engine "Code"	Synopsys Processor Control Language (PCL)
All SVI Masters and Slaves	VHDL code
FIFO's	Synopsys SmartModel FIFO's
SRAM	Synopsys SmartModel SRAM
Data Buffer	Synopsys SmartModel Data Buffer's
Interrupt PAL and Address Decode PAL	VHDL Code

4.0 Concept of Operation

Starting at the PCI (A) side, we will describe the operation of the bridge element when transferring data from the A-side to the Myrinet (B) side. The transfer of data in the opposite direction proceeds very similarly. Data starting in the PCI network, travels through the PCI Network Interface which satisfies the network protocol requirements of the PCI, then through the PCI Fabric Interface SVI Master and enters the RNI A-side slave across the SVI interface. The SVI slave places the data in the A_IN FIFO. When the FIFO receives the first word of the message, an interrupt is generated to the NE. The NE determines the source of the interrupt and services it by entering a data transfer loop. The NE reads the 2 64-bit PCI header words (32-bits at a time) from the

A_IN FIFO in order to extract the addressing information. Using routing tables stored in the SRAM, it translates the addressing information in the PCI header into route and address information for the Myrinet side. The NE writes the new header into the B_OUT FIFO. At that point, the NE is free to read the next piece of data from the A_IN FIFO, perform whatever data format translations are necessary and place the data in the B_OUT FIFO. The NE will stop transferring words when one of the following two situations occur:

1) The A_IN FIFO becomes empty, meaning that the NE does not have any data to read. The A_IN FIFO becomes empty when the end of the message passes through the FIFO or when the transmission of data being sent from the A-side network is temporarily halted.

2) The B_OUT FIFO becomes full, meaning that the NE cannot write another word to the output FIFO until the Myrinet master reads a word out of it.

The RNI detects the last word of a message by monitoring the ***Last_word_passing*** signal. This signal is asserted when the SVI_slave on the input side sends the last word of a message into the input FIFO. The logic used to generate this signal depends on what mechanism is available to determine the end of a message. This can range from reading the word count from the message header to timing out waiting for more data to be placed in the input FIFO. When ***Last_word_passing*** is asserted, the A-side SVI slave will not accept any further words (a new message) until the last word of the current message has been read out of the B_OUT FIFO by the Myrinet Network Interface. Once the last word has cleared the B_OUT FIFO, the RNI will recognize the next data entering the bridge as a new message header.

5.0 Approach

There are many ways to implement this functionality. The solution chosen is dependent upon many design trade-offs including the desired throughput, latency, size and power of the resulting bridge element. The solution presented below was arrived at by the following criteria:

· it demonstrates the required functionality of the bridge element

· it uses a programmable processor to perform the network level protocol and the data format translations

no development system or hardware is required to perform design trade-offs

5.1 Building Blocks

Network Engine

The processing of the RNI bridge node is performed by the network engine (NE). The NE used was the Motorola MC68040 microprocessor. The MC68040 runs on a 33 MHz clock, has 32-bit buses for address and data and handles up to 7 levels of interrupts. This particular processor was chosen because it is fairly commonly used, it provided sufficient functionality, and there is a Synopsys SmartModel Hardware Verification simulation model available for it. This choice was not necessarily the optimal processor for this specific task. There are many tradeoffs involved in choosing an appropriate NE: speed, throughput, word size, number of interrupt levels, etc. In the simulations, the Synopsys Processor Control Language (PCL) was used to program the hardware verification version of the MC68040 with timing annotated from compiled code running on an actual MC68040.

The functionality described below is the minimum functionality required of an NE. Normally there would be data conversions and bit manipulations to be performed on the data in order to convert the data from one network representation to the other. This type of manipulation is not implemented on this bridge since it is very specific to the particular network pair being bridged. The performance of this implementation is similar to what could be achieved using a message-level standard interface such as PacketWay [Cohen, et. al. 1997] for which very little header/data manipulation is required in the bridge. There may also be a requirement to implement certain communication structures such as mailboxes and semaphores on the bridge. These requirements are described in more detail in [LMATL 1997b].

All of the data transmitted across the bridge must pass through the NE. Data is read by the NE from the input FIFO and then the data is written by the NE to the output FIFO. If both networks are simultaneously transmitting data, then the NE services the input FIFO's in a round-robin fashion. First the NE reads a word from the A_IN FIFO and then writes that word into the B_OUT FIFO. After that word is written, the NE reads the next word from the B_IN FIFO and writes that word into the A_OUT FIFO. The NE then, once again, reads the next word from the A_IN FIFO and so on. This process will continue until one side runs out of data at which time the NE will service the remaining active side, exclusively. This method allows both sides to transmit data independently of each other. The throughput of just one side transmitting is twice the throughput of both sides transmitting. This is because all of the data must go through the one path in the processor. A faster implementation would be to use the processor for header translation, but not for data transfer. Data can be transferred much more efficiently by simply piping the data from the input FIFO to the output FIFO.

If data conversions are required they can be performed more efficiently in hardware such as in an FPGA placed between the input and the output FIFOs. Control of the flow of data from the input FIFO through the data conversion hardware (if necessary) to the output FIFO can also be done in hardware. Removing the processor from the data path would significantly decrease latency through the bridge.

Header Manipulation

The program in the NE is coded to recognize the difference between header words and data words. A PCI header consists of 2 words of route and 2 words of address. When going from the PCI to the Myrinet, the NE strips off the first 4 32-bit words. The NE then reads a pre-defined Myrinet header from the SRAM and prepends it to the message.

A Myrinet header consists of one or more route bytes with the MSB = ONE for each byte. The bytes are organized into 32-bit words. The last header word contains from zero to three pad bytes to pad the last route byte(s) to 32-bits. The MSB of each pad byte is ZERO. The last word of the Myrinet header is the Packet Type word, consisting of 4 bytes, each with an MSB of ZERO. The Packet Type word is the first header word in which the MSB is ZERO. When going from the Myrinet to the PCI, the NE will discard every Myrinet header word up to and including the Packet Type word and prepend a predetermined PCI header which it reads from SRAM. The words after the Packet Type are data.

The NE Interrupt Handlers

The NE also handles the interrupts that are asserted by the Interrupt PAL. In the case of the MC68040, these are vectored interrupts. Upon recognizing an interrupt, the processor reads a specific address in the Interrupt PAL which contains the vector number of the interrupt as shown in Table 1. It then performs the appropriate interrupt service routine based on that vector. After the NE acknowledges an interrupt, the Interrupt PAL resets the interrupt indicator so it can detect the next interrupt.

Interrupt Pal

The Interrupt PAL generates the interrupts that take place as data flows through the RNI bridge node. Inputs to this PAL are the Empty Flags (EF) of the input FIFOs and the Programmable Almost Full Flags (PAF) of the Output FIFOs. Table 1 shows the source of each interrupt, the corresponding interrupt vector and a short description of the interrupt service routine.

Interrupt Vector	**Source FIFO**	**Description of Service Routine**
40	A_IN Empty	Inhibit processor read from A_IN FIFO
41	B_IN Empty	Inhibit processor read from B_IN FIFO
42	A_OUT Not Almost Full	Enable processor write to A_OUT FIFO
43	B_OUT Not Almost Full	Enable processor write to B_OUT FIFO
44	A_OUT Almost Full	Disable processor write to A_OUT FIFO
45	B_OUT Almost Full	Disable processor write to B_OUT FIFO
46	A_IN Not Empty (initial or after last word)	Enable processor read header information from A_IN FIFO
47	B_IN Not Empty (initial or after last word)	Enable processor read header information from B_IN FIFO
48	A_IN Not Empty	Enable processor read data from A_IN FIFO
49	B_IN Not Empty	Enable processor read data from B_IN FIFO

Table 1- Interrupt Vector Table

There are 5 possible interrupts corresponding to data flow in each direction. If two interrupts occur during the same cycle, the interrupt with the higher priority is processed first. If both directions experience simultaneous and equivalent interrupts, then the A-to-B direction is given priority. This is easily changed by rearranging the order in which the interrupts are processed in the interrupt code. Only one interrupt will be generated by the Interrupt PAL per clock cycle. Another interrupt from a different source will not be generated until the existing pending interrupt is acknowledged. This logic allows for them to be Daisy-chained, with the recognition of room to write into the output FIFO being the most important, and the recognition of data on the input FIFO being the least important. This is explained in more detail in Table 2.

SRAM

The header information for the Myrinet is held in the SRAM. When the processor comes to the header conversion code, it waits a length of time that can be determined by running the header conversion code on an actual MC68040. The header conversion code interprets the header of the incoming packet and replaces it with the header that would be needed in order for the message to reach its destination in the other network. In this case, the SRAM was used to store predetermined header

EF_in	6	Indicates “THE INPUT FIFO IS EMPTY AND THE WORD JUST READ IS JUNK”. This synchronous interrupt must be recognized immediately. If the EF is set by a READ command, the processor must be told immediately that the data it just read is invalid. This requirement stems from how the IDT synchronous FIFOs operate. If this interrupt were not immediately processed, and the FIFO became non-empty in the meantime (removing the interrupt), the processor would hold onto this "vapor data" thinking it were real.
NOT_PAF_out	5	Indicates “NE CAN WRITE INTO THE OUTPUT FIFO”. Throughput is improved if the NE knows when it is able to write to the output FIFO. Thus, this interrupt has higher priority than the NOT_EF_in interrupt. Software flags prevent the processor from reading a piece of data for which there is no room in the output FIFO.
PAF_out	4	This indicates "STOP WRITING TO THE OUTPUT FIFO". It has a higher priority than the "NE CAN READ FROM THE INPUT FIFO NOW" interrupt in order to prevent deadlock occuring from reading a piece of data for which there is no rooom in the output FIFO.
NOT_EF_in	3	This indicates "NE CAN READ FROM THE INPUT FIFO NOW". All other interrupts must be serviced before this one.

Table 2 Interrupt Priority Description

information that the destination network would understand. The new header information is represented very simply as a continuous list of 32-bit Myrinet header words located in the SRAM. Upon receipt of the first message's header words from the PCI, the NE discards them and reads the new header from the SRAM. The NE maintains a pointer to the next header information to be read. When the next message arrives, the NE reads the new header and discards it. Then it loads the pointer, reads out the next header, writes it to the output FIFO and updates the pointer. After the header words have been written to the output FIFO, the NE begins to read the data words in from the input FIFO and write these same data words out to the output FIFO. For a virtual prototype using this RNI model data manipulation is required, the timing for the conversion can be accounted for by causing the PCL code to wait for the amount of time determined by running the data conversion code on an actual MC68040 before writing the data to the output FIFO.The data, however, would remain unchanged. If the correct data is needed at the output, then a method of providing that data would be required.

Address Decoder Pal

The Address Decoder PAL component takes on four simultaneous, but independent tasks. It controls the direction of data flowing between the FIFOs to the NE through the data buffer based on the NE READ/WRITE signal and the desired address. It also controls the Read Enable and the Output Enable signals on the Input FIFOs and the Write Enable signals on the Output FIFOs. Both of these tasks control the flow of data between the NE and the FIFOs. This PAL also creates some of the interrupt signals that are processed further in the Interrupt PAL. Finally, the ***Last_word_passing*** state machine resides in the Address Decoder PAL.

Last_word_passing State Machine

The beginning of a message contains header information which must be treated differently from data. It is necessary to know where one message ends and the next begins. The Address Decoder PAL tracks the last word of a message through the bridge using the ***Last_word_passing*** state machine. It will not allow a new message into the bridge until the old message has been transferred to the SVI on the output side of the bridge. When an interrupt indicates that an input FIFO has gone from empty to not empty, the Address Decoder PAL determines, via the state of the ***Last_word_passing*** state machine according to Table 1, whether this interrupt is because a new packet has arrived (interrupt vector 46) or because an already existing packet has resumed transmission (interrupt vector 48). Depending on the value of the interrupt vector returned by the Interrupt PAL, the NE will interpret the data as either regular data or as header information.

Table 3 shows the states and transitions for the ***Last_word_passing*** state machine. When the SVI_slave on the input side sends its last word into the input FIFO, the slave asserts the ***Last_word_passing*** signal, indicating that the last word is in the RNI bridge element.

This assertion of the ***Last_word_passing*** signal transitions the state machine from State 1 to State 2. In this state, the input-side slave refuses data by deasserting the SVI ***Data_Ready*** signal. When the input FIFO's empty flag is asserted as a result of the NE reading the last word from the input FIFO, the state machine transitions to State 3. The transition to State 4 occurs on the next write to the output FIFO which means that the NE has written the last word to the output FIFO. Note that once the header has been stripped off, the NE reads a word from the input FIFO, puts it in the format of the receiving network and writes it to the output FIFO before it reads another word. If this sequence is changed, then the state machine would need to be changed since its operation relies on the fact that once the input FIFO goes empty after reading

State	Input to cause transition to next state	Transition to State	Action
idle	***Last_word_passing*** deasserted	1	wait for ***Last_word_passing*** to be asserted
1	***Last_word_passing*** asserted	2	sending slave withdraws ***Data Ready***, preventing next message from entering the bridge
2	EF_in asserted	3	Last word has been read by the NE
3	decode output FIFO address and write signal	4	Last word has been written to the output FIFO
4	output FIFO transitions from being NOT empty to empty	idle	Deassert ***Last_word_passing*** Next input FIFO empty to not empty transition indicates a new message

TABLE 3 *Last_word_passing* State Machine

the last word, the next write to the output FIFO will be to write the last word. After the last word has been written to the output FIFO, the state machine waits until the output FIFO goes from being NOT empty to being empty which causes it to transition to the idle state. Then ***Last_word_passing*** is deasserted, meaning the last word has passed through the bridge, and the sending side can send its next packet if it is ready to do so.

Data Buffer

The data buffer is a bi-directional buffer which directs the flow of data from the Input Fifo to the NE or from the NE to the Output FIFO. The Address Decoder PAL controls the direction of the buffers based on the NE READ/WRITE signal and the desired address.

FIFOs

The FIFOs are used to buffer and control the data as it transits the bridge element. Since the NE can only read or write one data word at a time and takes at least 6 clock cycles to perform a read and write of one data word, the FIFO is needed to temporarily store the data. In this analysis, the minimum depth needed for the maximum throughput is 2 words. At a FIFO depth of less than two words, the FIFO constantly alternates between being empty and not empty and the NE spends most of its time processing empty and not empty interrupts. The deeper the FIFOs, the less impact the bridge will have on the sending network since it will be able to dispatch large blocks of data into the FIFO and then go on with other tasks. However, the FIFOs occupy significant amounts of real estate and, depending on the speed, can dissipate a significant amount of power. The depth of the FIFO needs to be studied as one variable of the implementation trade-offs.

6.0 Special Considerations

Pad Bytes

The Myrinet protocol includes 2 tailer words at the end of the packet. The first of these words is the CRC, while the second word is the pad word which indicates how many pad bytes were in the previous word. Since these two words cannot be interpreted on the PCI interface, they must be removed from the sending packet before the packet reaches the PCI interface. In this example case, two pipeline registers were inserted in the svi_slave_ctl_32.vhd code on the Myrinet side. When a word enters this SVI slave, the word must transmit through both of these registers before entering the Input FIFO. When the last word (pad word) enters the SVI_slave, the svi_last_word_out and ***Last_word_passing*** become asserted, and no further words are put into the input FIFO. At this point, the CRC is in the pipeline register just before the FIFO and the pad word is in the register behind it. There is a Boolean flag associated with each register which prevents the two words from entering the FIFO at the beginning of the next message. Each of these Boolean flags is initialized to FALSE. When the first data word enters the first register, the flag for that register becomes TRUE. When it enters the second register, the flag for that register becomes TRUE. This flag controls the FIFO_WRITE_ENABLE signal. When it is TRUE, the data from the register gets written into the FIFO. When the svi_last_word_out is asserted and the slave goes back to idle, the register flags are deasserted, which in turn deasserts the FIFO_WRITE_ENABLE signal. The FIFO_WRITE_ENABLE signal will not become true again until two new words from the next message are in the pipeline registers and ready to be input into the FIFO [LMATL 1996b].

Similarly, when a word is sent to the Myrinet Network, the Myrinet protocol expects a CRC and a pad word to be at the end of the packet. The Myrinet SVI master must append a CRC and the pad word. How the CRC is calculated is left to the implementer. In this example, it has been pre-calculated and stored in RAM. Since the PCI sends 64 bit data words, there are no actual pad bytes that are going across the RNI bridge node. For this reason the pad word will always consist of 32 bits of all zeros.

Data Width Conversions

The RNI bridge node has a 32-bit interface, and the PCI Interface is 64-bit words. The SVI specification requires that width conversions be handled by the slave. A width converter was inserted at the PCI interface in the SVI Slave portion of the SVI Interface. When data flows from the RNI to the PCI, the data must go into a 1-to-2 converter . When the data flows from the PCI to the RNI, the data must pass through a 2-to-1 converter.

Since the Myrinet Interface and the RNI bridge node are both 32 bit word interfaces, no conversion has to be made in order to transmit to and from these two interfaces

7.0 Results

7.1 Performance Analysis- The performance results are presented as an example of the type of analysis which might be performed in order to determine the performance impact of the RNI bridge. Since the Motorola MC68040 was chosen more for its ease of programmability and the availability of a model, the actual numbers which come from the analysis are probably not as interesting as the method by which they were calculated. As seen from the graph in Figure 4, the message length has a considerable impact on the RNI throughput for small messages. The time it takes to send a packet from one network to the other network is a function of the number of words that are sent across the interface. For this RNI, the majority of the transmission time is the time it takes the MC68040 microprocessor to read the data from the input FIFO and write the data to the output FIFO. A data word becomes queued up in the Input FIFO, gets processed through the NE and enters an empty output FIFO. The output FIFO is empty since the PCI network can process a data word faster than the 68040 microprocessor can read and write a word.

Note that the peak rate of slightly more than 21 Mbytes per second corresponds very closely to the theoretical peak for a processor that can read and write one word every 6 cycles and 33 Mwords/sec (= 132 Mbytes/sec). That is, the peak rate is 132/6 = 22Mbytes/sec. So, with no attempt at optimization, we are achieving 90% of the peak bandwidth. This observation must be tempered by the fact that there is very little manipulation of the headers and no manipulation of the data occuring. From our experience with the SVI, we anticipate that even with extra processing requirements, the optimizations, especially doing any data conversions in hardware and getting the processor out of the data path, we will still be able to achieve 90% of peak throughput for moderately-sized data sets.

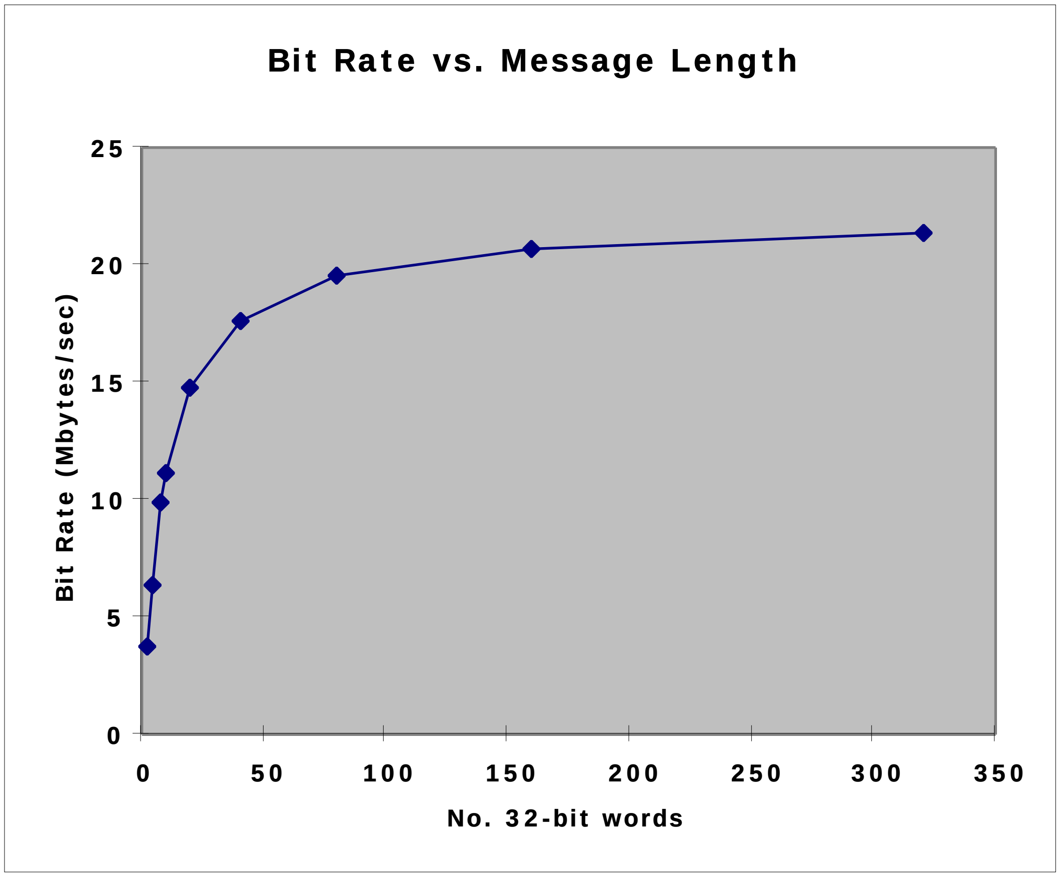


Figure 4- The BIT RATE varies with the message length, especially for small messages.

The transmission equation is shown below. The terms are defined in Table 4.

Transmission Equation

$t_{packet} = t_{M\_to\_IF} + t_{initiate\_interrupt} + t_{ack} + t_{idle\_interrupt} + n_{hm}t_{read} + n_{hp}(t_{read\_SRAM} + t_{write}) + n_d(t_{read} + t_{write} + t_{idle}) + t_{in\_OF} + t_{OF\_to\_P}$

Assumptions:

1) Data is transiting the RNI in one direction. If the RNI is processing 2 packets going in opposite directions, there will be a longer idle time between the NE's reads and writes.

2) The Myrinet Network will send data continuously, and the PCI Network will be able to keep up with the data coming across the bridge.

$t_{M\_to\_IF}$	data input to the Myrinet Network to Input FIFO write complete
$t_{initiate\_interrupt}$	first word write into Input FIFO to Input FIFO NOT Empty interrupt asserted
t_{ack}	Input FIFO NOT Empty interrupt to interrupt acknowledge complete
$t_{idle\_interrupt}$	interrupt acknowledge complete to NE read from Input FIFO.
t_{read}	NE read from FIFO
t_{read_SRAM}	NE read from SRAM
t_{write}	NE write to output FIFO
t_{idle}	idle time between read/write cycles
t_{in_OF}	time last word is in output FIFO
$t_{OF\_to\_P}$	Data read from output FIFO to data at PCI Network
n_{hm}	number of header words from Myrinet Network
n_{hp}	number of header words going to PCI Network
n_d	number of data words being sent from Myrinet Network to PCI Network

Table 4- Definition of terms

7.2 Implementation Summary

The VHDL code was broken up into 3 parts: the SVI_Myrinet, the SVI_PCI, and the Address Decoder PAL combined with the Interrupt PAL, synthesized and targeted to FPGAs in order to obtain an estimate of how large these elements will be. The results are shown in Table 5. Note that these figures are first run estimates; the

Chip Created	ORCA Device Used	IO's	PFUs
SVI_Myrinet	2C15	155 of 320	257 of 400
SVI_PCI	2C15	294 of 320	387 of 400
Address Decoder and Interrupt PALs	2C06	68 of 192	75 of 144

Table 5- Implementation Size Estimates

results have not been optimized in order to create the smallest and fastest possible chips and no timing verification was performed. Thus, the rather high utilization on the SVI_PCI FPGA could probably be lowered by optimization, thus allowing it to fit into the 400PFU ORCA device.

8.0 Enhancements

At this time, no enhancements are planned for the RNI. However, one enhancement strongly suggests itself, and that would be to let the data transit from the input FIFO to the output FIFO without intervention from the processor.

9.0 Summary

The RNI is used as a bridge between heterogeneous networks. RNS bridges can be implemented as custom hardware or with programmable processors. The custom hardware RNI tends toward lower latency while the programmable RNI provides greater opportunity for reuse. This guideline presents an example of a programmable RNI. Over 90% efficiency can be achieved using this RNI. This RNI can be reused to bridge different networks simply by reprogramming the processor. If both low-latency and high reuse are required, the VMI is an alternative.

10.0 References

Buchanan 1995a

Buchanan, G. 1995. “Hardware Synthesis Study of WSSPT SVI Interface Encapsulations,” http://www.atl.external.lmco.com/projects/rassp/legacy/appnotes/MYA/index.html, July 1995.

Buchanan 1995b

Buchanan, G., 1995. “Simple Reconfigurable Network Interface (RNI) Encapsulation Study Using the Cypress HOTLink High-Speed Serial Link and the PCI Fabric Interface to Implement a Sensor-to-PCI RNI,” http://www.atl.external.lmco.com/projects/rassp/legacy/appnotes/MYA/index.html, September 1995.

Buchanan 1996

Buchanan, G., 1996. “Myrinet to SVI External Network Interface,” http://www.atl.external.lmco.com/projects/rassp/legacy/appnotes/MYA/index.html, July, 1996.

Chhabra 1996

Chhabra, A., 1996. “SVI Verifcation Study: Encapsulations of the Benchmark II-Data I/O Board and the Mercury RINO/RIC Chipset,” http://www.atl.external.lmco.com/projects/rassp/legacy/appnotes/MYA/index.html, May, 1996.

Cohen et. al. 1997
Cohen, D., Craig Lund, Tony Skjellum, Thom McMahon, Robert George, 1997. “Proposed Specification for the PacketWay Protocol,” ftp://ftp.ietf.org/internet-drafts-ietf-pktway-protocol-spec-03.txt, February 1997.

LMATL 1995
Lockheed Martin Advanced Technology Laboratories, 1995. “RASSP Methodology Version 2.0 Volume I,” http://www.atl.external.lmco.com/projects/rassp/legacy/appnotes/MYA/index.html, October 1995.

LMATL 1996a
Lockheed Martin Advanced Technology Laboratories, 1996. “RASSP Model Year Architecture Specification Volume I: Introduction Version 1.0,” http://www.atl.external.lmco.com/projects/rassp/legacy/appnotes/MYA/index.html, September 1996.

LMATL 1996b
Lockheed Martin Advanced Technology Laboratories, 1996. “RASSP Model Year Architecture Specification Volume II: Hardware Architecture Element Specification Version 1.0,” http://www.atl.external.lmco.com/projects/rassp/legacy/appnotes/MYA/index.html, September 1996.

LMATL 1997a
Lockheed Martin Advanced Technology Laboratories, 1997. “Virtual Microarchitecture Interface,” http://www.atl.external.lmco.com/projects/VMI/VMI.html, August 1997.

LMATL 1997b
Lockheed Martin Advanced Technology Laboratories, 1997. “Virtual Microarchitecture Interface (VMI) Specification (Draft) Volume II: Communication Services- Pre Release Version,” July, 1997.

MYRI 1994
Myrinet Links and Routing Specification, Myricom, Inc., Arcadia, CA, May, 1994.

Wedgwood 1997
Wedgwood, J., 1997. “Reconfigurable Network Interface (RNI) Study: Myrinet to PCI,” http://www.atl.external.lmco.com/projects/rassp/legacy/appnotes/MYA/index.html, August 1997. (This document)