Host Interfacing: A Coprocessor Approach

by

Nikhil Chinmaya Gautam

B.S., Electrical Engineering
Columbia University, 1990

Submitted to the Department of Electrical Engineering and
Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1993

© Massachusetts Institute of Technology 1993. All rights reserved.

Author ...........................................................

Department of Electrical Engineering and Computer Science
29 January, 1993

Certified by ...........................................................

David L. Tennenhouse
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ......................................................

Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

MAR 24 1993
ARCHIVES
LIBRARIES
Host Interfacing: A Coprocessor Approach

by

Nikhil Chinmaya Gautam

Submitted to the Department of Electrical Engineering and Computer Science on 29 January, 1993, in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science

Abstract

The interface between networks and workstations is increasingly becoming a bottleneck, with the increase in the bandwidths of networks and the speeds of workstations. As a result, the design of host interfaces has become a critical area in the research community. In this thesis, we identify a relatively neglected issue—that of the point of attachment of the network on a workstation—and present a model framework within which we can study this issue.

As a specific instance, we define a coprocessor style of host interface, in which the network is made to appear to reside in the registers of the coprocessor. We describe a VLSI chip implementation of a coprocessor host interface for the MIPS R3000 architecture which we designed and fabricated. We ran a series of experiments in which we compared the potential performance of the coprocessor approach with that of I/O bus-based (programmed I/O as well as DMA) approaches. In an environment in which the host processor needs to touch the network data, the coprocessor approach gives a factor of 1.3 to 6.6 higher throughput (depending on the machine used, and the direction of transfer), compared to a programmed I/O approach. In comparison to a DMA approach, the coprocessor approach is also faster, by around 8 to 25%.

Thesis Supervisor: David L. Tennenhouse
Title: Assistant Professor of Electrical Engineering and Computer Science
Acknowledgments

*I am a part of all that I have met.*

Alfred Lord Tennyson.

They say no man's an island, entirely on its own.
There seems no greater truth, at the time I'm writing this,
For many the friends and guides, to me who have shown,
The paths to take, and those not to, in writing my thesis.

I take this opportunity, at the very outset,
To thank Professor Tennenhouse, whose invaluable guidance,
Encouragement and support, I shall not likely forget.
This work would not be possible, without his assistance.

To the members of TNS, I would like to show,
My gratitude for their help, be it large or be it small:
Joel Adam, David Carver, Mike Ciholas, Henry Houh,
Chris Lindblad, David Martin and David Wetherall.

Of our various group secretaries, I could ask for nothing more:
K.J., Newton, and Tanie, were as patient as can be.
I also give my deep thanks, to friends I met at Beilcore,
Tony Bogovic, Jason Hickey, whose wisdom guided me.

For fun times at MIT, special thanks to all my friends,
Especially Bharat, Hitesh, Milan, whose friendship I'll cherish,
With whom many hours were spent, talking of means and ends.
Thanks to Satbir and Vishnu, bearing with me at the finish.

For the moral support to which, I could turn at any time,
I finally thank my parents, dear Mummy and Bapu.
Their encouragement meant a lot, so at the end of this rhyme,
I say with all my heart, "this one's for you!"

The work in this thesis was funded by the Defense Advanced Research Projects Agency,
and the National Science Foundation, through the auspices of the Corporation for National
Research Initiatives.
In loving memory of Darshi.
### Contents

1 Introduction .......................... 10
   1.1 Trends in Host Interfacing .......... 11
   1.2 Networked Workstations .......... 12
   1.3 Outline of Thesis .................. 12
   1.4 Organization of Thesis .......... 13

2 Background .......................... 14
   2.1 Framework: Processor-Network Interaction .......... 15
   2.2 Previous Work .................. 18
   2.3 Summary: Issues of Host Interfacing .......... 25

3 The Connection Issue ............... 30
   3.1 Model Connection Architecture .......... 31
   3.2 Methodology: Applying the Model .......... 34
   3.3 Points of Attachment ............... 37
   3.4 Case Studies .................. 48
   3.5 Conclusion .................. 52

4 The Coprocessor Approach .......... 54
   4.1 Introduction .................. 54
   4.2 Background .................. 55
   4.3 Architectural Overview .......... 58

5 The Coprocessor Host Interface Chip .......... 68
List of Figures

2-1 ATM Cell Structure ............................................. 15
2-2 Turbochannel DMA Host Interface .......................... 19
2-3 Micro Channel Host Interface ................................. 21
2-4 The Alewife Machine ........................................... 23
2-5 Design Space of Host Interfacing ............................ 26

3-1 Model Connection Architecture ............................... 32
3-2 Data Flows: Network in I/O Device ......................... 38
3-3 Data Flows: Network in Memory ............................. 42
3-4 Data Flows: Network in Cache ............................... 44
3-5 Data Flows: Network in Registers ......................... 45
3-6 Data Flows: Data Not Processed ............................ 46

4-1 ViewStation Architecture ..................................... 55
4-2 Coprocessor Interface Architecture: A User's View .... 59
4-3 Partial Register Map of R3000 with Coprocessor Interface .... 61

5-1 Functional Block Diagram ..................................... 70
5-2 Pipeline Stages .................................................. 72
5-3 External Interface Signals ................................... 73
5-4 R3000 CPU System Bus ....................................... 76
5-5 Network Receive Timing ...................................... 77
5-6 Pipeline Decoder ............................................... 78
5-7 Cell Buffers ..................................................... 79
List of Tables

3.1 Normalized Throughput Index: Network in I/O Device .......... 39
3.2 Normalized Throughput Index: Network in Memory ............ 42
3.3 Normalized Throughput Index: Network in Cache ............. 43
3.4 Normalized Throughput Index: Network in Registers .......... 44
3.5 RS/6000 Throughputs ........................................... 49
3.6 Normalized Throughput Index: RS/6000 ......................... 50
3.7 DECstation 5000/240 Throughputs .............................. 52
3.8 Normalized Throughput Index: DECstation 5000/240 .......... 52

5.1 Coprocessor Interface Instructions .............................. 71
5.2 Coprocessor Interface Signals .................................. 74
5.3 Network Interface Signals ...................................... 76

6.1 Interface models .................................................. 95
6.2 Sample time measurements, normalized per cell ............... 96
6.3 Peak Throughput (with header byte swap) ...................... 96
6.4 Peak Throughput (without header byte swap) .................. 96
6.5 Predicted vs. Observed Throughputs (in Mbps) ................. 97

B.1 Times for Entire Cell Burst: DECstation 5000/240 ............ 126
B.2 Times for Entire Cell Burst: DECstation 5000/200 .......... 127
Chapter 1

Introduction

Once begun,
A task is easy; half the work is done.

Horace.

As the bandwidths of networks and the speeds of workstations increase, the interface between the two increasingly becomes a bottleneck. Technology has pushed networks into the Gigabits per second range [9] and a new generation of workstations is running at speeds up to 200 MHz [18]. However, the true impact of these improvements will be felt only if the mechanisms for moving data between a workstation and a network are of high-performance.

In this thesis, we use the term “host interface” to denote the hardware components and the software routines that implement these mechanisms for connecting a workstation to a network. An important issue in host interface design is the point of attachment of a network to a workstation. We have presented a model which helps us study this question. We have also developed a specific instance of an interface. The host interface that we designed appears to the workstation as a closely-coupled coprocessor, and is a high-performance design that eases this bottleneck.
1.1 Trends in Host Interfacing

Traditionally, a standard I/O channel that is offered on today's workstations has been perceived as the only path to transfer data in and out of a workstation. Indeed, a survey of the host interfaces that have been designed until now shows this channel to be the chosen point of attachment [11, 14, 21, 42]. However, since these I/O channels are standards, they have not evolved as much as the rest of the workstation. Further, these channels are far removed from the internal central processing unit (CPU) of the workstation, and thus the path for data to move between the network and the CPU is relatively slower than if the network could somehow be placed very near the CPU. No such trend has been observed in the host interfaces of today's networked workstations.

However, we have seen this trend towards network proximity to the CPU in a different area: the world of multiprocessors. The MOSAIC [39] distributed-memory multicomputer was one of the earlier projects in which the network interface was brought on-chip. Other designs have also brought the interconnection communication mechanism close to the processor [12, 27]. In these projects, besides the network interface, the processor unit is also custom-designed, and hence the systems are tuned for fast communications.

An even more recent trend has been towards designing such closely-coupled network interfaces, but with a standard RISC processor (slightly modified) as the core processing engine of a node in a multiprocessor system. The cache controller of the Alewife machine [2, 32] is such an example, which works in conjunction with a modified SPARC chip. The on-chip coprocessor interface of the *T project connects a multiprocessor network to a Motorola 88110 processor.

We felt it would be useful to consider the design of a closely-coupled interface in the more traditional workstation environment. The coprocessor approach seemed interesting, as it had not been tried before in such an environment, and was only beginning to be researched in the multiprocessor world (with the *T coprocessor). We describe our design in more detail later on.
1.2 Networked Workstations

The idea of connecting workstations to networks is an old one, and the earliest Local Area Networks (LANs) started appearing in the work environment in the mid-seventies [15]. The proliferation of Ethernet, token ring and token bus LANs throughout the eighties reflected the popularity of the concept of having convenient access to other computers, printers and plotters in other offices from one's own desk. The 10 Mbps data rates of these networks were sufficient for such low bandwidth communications. As the number of workstations needing to be interconnected grew, and services like voice and video were added, newer networks were developed, such as FDDI (Fiber Distributed Data Interface) and DQDB (Dual Queue Dual Bus), which have data rates of around 100 Mbps. (A description of these various networks can be found in several sources, including [6, 15].)

The newest high-speed networks can support rates exceeding 1000 Mbps, and are expected to carry a wide range of services, from voice to data to high quality video. Multimedia applications seem to be the driving force behind such networks. For example, at the heart of the ViewStation project (a distributed video system being developed in our research group [41], described later in Section 4.2.1) lies a high-speed network that uses Asynchronous Transfer Mode (ATM) as a communications mode. ATM [15] (see Chapter 2) has been developed as a standard for the Broadband Integrated Services Digital Network (B-ISDN) by the CCITT, but other technologies, like plaNET [20] have also been developed.

1.3 Outline of Thesis

This thesis presents a general framework for host interface design, and also details a specific example within this framework:

A study of current host interface designs is first presented, and the issues which have been focussed upon by the research community are summarized. The importance of an additional issue relatively neglected so far—that of the point of attachment of
a network in a workstation—is discussed next. This leads to the development of a model connection architecture, which helps us analyze different points of attachment of a network on a workstation.

As a specific example of how the point of connection on a workstation can affect throughput, we define, in this thesis, the architecture of a coprocessor-style of host interface. The main implementation thrust of this thesis is on the design of a VLSI Coprocessor Host Interface Chip. A series of performance experiments were conducted that measured and compared the throughput of different interface strategies, including the coprocessor approach. These experiments are described, and the results are also analyzed in this thesis.

### 1.4 Organization of Thesis

In Chapter 2 we present a study of previous work done in host interface design. Chapter 3 then develops a model architecture of a hierarchical memory-I/O system, with which we can analyze different points of connection of a host interface.

In Chapters 4 through 6, details of the coprocessor approach are presented, and experiments that were made to compare the performance of this approach with other approaches are also described.

Chapter 7 summarizes the contributions of this thesis. Appendices are provided at the end to give details of the instruction set, chip layout, and code segments of the experiment routines.
Chapter 2

Background

Today is yesterday’s pupil.

Thomas Fuller.

Several host interfaces of various flavors have been designed, and some studies have been done on the performance benefits of particular designs. This chapter attempts to describe the design work and research that has been done in the field of host interfacing.

We start this chapter by outlining in Section 2.1 the basic operating framework in which a host interface is expected to perform. With this operating environment as a backdrop, we then proceed to describe some host interfaces which have been designed by other research groups (Section 2.2). In summarizing this current work done by the research community, we then identify, in Section 2.3 the issues of host interfacing which have so far been focussed upon.

Since ATM is mentioned at times in the following discussion, we first describe this technology briefly. More details can be found in several sources, including [7, 15].

Asynchronous Transfer Mode

The Broadband Integrated Services Digital Network (B-ISDN), developed by the CCITT as a standard for high-speed communications, uses Asynchronous Transfer Mode (ATM), which is a fast packet switching protocol, but that operates in
a connection-oriented mode. Thus, a virtual connection must be set up between two points, after which data can flow in packets on that connection.

The packet is of fixed size—53 bytes—and is called a cell. Fast switching is achieved by having fixed size cells, and also by reducing the functionality of the header [15]. The main function of the header is to provide the virtual connection number, also known as the Virtual Channel Identifier (VCI), which represents routing information. The format of the B-ISDN cell is shown in Figure 2-1 [7].

![ATM Cell Structure](image)

Figure 2-1: ATM Cell Structure

### 2.1 Framework: Processor-Network Interaction

Before delving into the details of the design of host interfaces we should pause to consider why the processor needs to interact with the network in the first place. In other words, what is the driving application program that needs to talk with the network?

There are myriad examples of programs that require a processor system to interact with network data. In a distributed memory multiprocessor system, a particular
processing node might access remote data through the inter-processor network [2, 12, 38]. In a distributed video system, a host workstation might be used for manipulating video data in some manner, like compression or filtering, or for simply outputting that data onto a video display unit. In any distributed system, a workstation might serve as a network file server.

While there seems to be a great variety of such application programs, we can actually classify them into two categories according to the degree of interaction that the processor needs to have with the network data. Put another way, the basic question here is how much “contact” is the host processor going to have with the network data. We discuss this issue next.

2.1.1 Does the processor touch the data?

This discussion is particularly relevant when we consider the receive direction, i.e. the data flows from the network into the workstation. Once the network data is at the host workstation interface, there are essentially two things that can happen: the processor can touch the data, or the processor can forward the network data unchanged, to some other location in main memory, to a disk file, or to some other processor on the network.

When we say that the processor does not touch the data, we mean that the data coming in from the network simply needs to go directly into a designated area in the main memory of the workstation. To be sure, the host (or host interface—depending on the design) might have to look at the header of an arriving network packet, (perhaps of the first of a burst of packets). However, the application is such that the major portion of the packet (i.e. the payload) goes to memory unchanged by the processor.

One example of such an application is the file server. When the processor is implementing a file server, it does not need to observe or change the actual data in any way. All it needs to do is make sure that the data reaches the correct place in memory, from which it can be copied to disk, without ever passing through the cache and/or the CPU registers. Another example is one in which the processor is simply transferring the video data from the network onto a screen.
The other environment is one in which the processor needs to modify the data coming in from the network, before finally placing it in its appropriate memory location.

One example of such an application is filtering in a distributed video system. In this case, the processor might want to operate upon all the bits of the incoming data payload, before placing that data into an appropriate location in memory.

2.1.2 Effect on Interface Evaluation

In evaluating host interfaces, we feel it is critical for us to keep in mind what sort of processor-network interaction we expect the system under study to operate under. Some host interfaces are better suited for an environment in which the processor needs to change the network data, and some are inherently ideal for environments in which the processor does not touch the network data. In the next section we shall study some previous host interfaces that have been designed, and we shall see how this interaction model will be helpful in our evaluations of each design.

For a brief example that illustrates the importance of this issue in evaluating as well as designing host interfaces, let us consider the system being developed in our group. In a distributed video system such as ours, video equipment such as cameras and potentially displays is connected directly to the network. Video data, thus, can move between the various components without needing to go via any workstation. If data does arrive at a workstation, then, it is most likely because some application-specific processing (such as filtering) needs to be done on that data. Thus, most of the applications that drive the host interface are those in which the processor will touch the data.

Given that we are operating in this regime, the path from the network to the processor's registers becomes a critical one, because it is on the registers that a processor (especially a RISC processor) operates [24]. Thus, in such an environment, we try to design a host interface that focusses on getting the data into the CPU registers or even to cache, rather than memory, quickly. We shall see that the coprocessor approach fits very well with this goal of feeding the registers of the CPU directly.
2.2 Previous Work

We now describe some network interfaces that have recently been designed, either to connect host workstations to networks, or in multiprocessor environments. As we analyze each design, we shall keep in mind the processor network interaction discussed above, and also try and identify some of the design decisions made for each host interface.

2.2.1 Traditional Workstation Interfaces

We see that the host interfaces of traditional networked workstations typically sit on the standard I/O channel of the workstation. For example, the 'Yes V3' [21] and the Bellcore host interface [14] connect the network to the DEC Turbochannel. Fore Systems is also developing ATM host interfaces for the DEC Turbochannel, as well as the Sun SBUS [11]. The UPenn host interface sits on the Microchannel of an IBM RS/6000 [42]. The Medusa interface [5] is slightly different, in that the adaptor sits on the graphics I/O bus, which is highly optimized for block transfers. We now describe some of these interfaces in more detail.

Turbochannel DMA Interface

In the host interface described in [14], the ATM network interface board is a DMA controller on the Turbochannel (the standard I/O bus of a DECstation). Thus the interface adaptor board is capable of gaining control of the Turbochannel, and transferring data on it directly to areas of the main memory of the workstation.

The adaptor board for this host interface comprises an embedded controller for each direction (transmit and receive), and some hardware which is dedicated to moving data between the host and the network. Figure 2-2\(^1\) shows the main blocks of this host interface.

A typical illustration of the use of this interface follows. In the transmit direction, once the host decides to send data into the network, it communicates with the Trans-

\(^1\)Diagram taken from [14].
Figure 2-2: Turbochannel DMA Host Interface

The Transmit Microprocessor Subsystem then decides the segmentation\(^2\) that needs to be done, and presents instructions to a FIFO which is found on the Transmitter Hardware. The latter subsystem is then responsible for actually moving data from the workstation memory, and for transmitting that data to the SONET link\(^3\).

Similarly in the receive direction, the host must specify the areas in memory where the payload of the cells is to be placed. This information is sent to the Receive Microprocessor Subsystem, which then instructs the Receiver Hardware to perform the necessary assembly of the payload in the cells.

\(^2\)The breaking up of higher layer data units into ATM cells is called segmentation, and the building up of higher layer data units from ATM cells is called reassembly. The network protocol layer that is responsible for these operations is usually referred to as the Segmentation and Reassembly (SAR) layer.

\(^3\)SONET—Synchronous Optical Network—is an optical transmission system. A SONET STS-3c link is defined to have a data rate of 155 Mbps, and an STS-12 has a rate of 622 Mbps.
The main component of the Microprocessor Subsystem in each direction is an embedded controller. In the current implementation, an Intel 80960CA Risc processor is used.

The Receiver/Transmitter data-moving hardware is responsible for performing the instructions given to it via a FIFO by the Microprocessor Subsystem. The data-moving hardware has an interface with the host bus, which is implemented in the form of a dual-ported memory, and has an FSM associated with it. There is also the interface with the transmission link, which consists of four 8-bit FIFOs, a SONET framer and associated control circuitry.

This interface is an attractive solution because of the high degree of flexibility that it provides. There is a powerful microprocessor dedicated for each direction, and hence a great deal of experimentation is possible in the algorithms for segmentation and reassembly. For example, when transmitting data, the host might multiplex cells destined for different locations, or might give priority to smaller messages. These different strategies can be tried and studied with ease because of the presence of the microprocessors.

This flexibility, however, comes at the cost of considerable complexity in the design. For example, the presence of microprocessors on the interface board prompts the need for some additional communication mechanisms (namely the FIFO's) between the actual data-moving hardware and the microprocessors, to account for the difference in speed of the two. While this is a benefit in the sense of temporally decoupling the data-moving of the FSMs from the decision-making of the microprocessors, it does introduce an additional layer of communication, and hence more complexity.

Further, this design is ideally suited to an environment in which the processor does not need to touch the data at all. The slowness of the Turbochannel is overcome in this design by using the DMA capability of the Turbochannel, which does provide a high peak data transfer rate. However, this leaves the host with the only choice of putting the network data straight into main memory. If the data does need to be touched by the host processor, it will have to first be placed in main memory, and then be accessed from there. Thus the bottleneck may then be the path from main
memory to the CPU registers. We discuss the performance of this interface in more detail in Chapter 6.

Microchannel Interface

This host interface [42] connects to the workstation via the Micro Channel of an IBM RS/6000 workstation. The Micro Channel is the standard I/O bus supported on an RS/6000. In a manner similar to the Turbochannel-based interface described earlier, the ATM network can be read and written via memory-mapped addresses.

The design of the interface is divided into two parts: the segmenter for transmitting, and the reassembler for the receiving. Each part consists of controllers, and buffers, and other circuitry. Figure 2-3\(^4\) shows the various blocks in the design of this host interface.

The segmenter consists of a controller, a SONET framer, and some memory in the form of FIFOs and registers. This section is responsible for calculating the header checksum, segmenting data into cells and transmitting the cells into the network, via the SONET framer.

\(^4\)Diagram taken from [42].
The transmitter consists of a Linked List Manager, a SONET Interface and a Reassembly Buffer. The SONET Interface is responsible for splitting the header from the cell body, and validating the checksum. Another function for this section is to figure out the linked list reference location. The Linked List Manager section maintains the data structures that point to locations in memory for particular VCIs. And finally the Reassembly Buffer actually stores the cell bodies into the host memory.

The functionality of this design is similar to Turbochannel interface described above—both perform segmentation and reassembly on board. The custom hardware approach employed in this design results in a reasonably low cost implementation, but the loss in programmibility decreases the flexibility of this design.

Further, as in the Turbochannel design of [14], the interface board sits on the I/O bus (the Microchannel, in this case) of the workstation. This bus is slow, and is the bottleneck of this design. In the streaming mode used in this design, the fastest data transfer rate of the bus is one 32-bit word every 100ns\footnote{Faster clock rates and wider word transfers will increase this rate in newer versions of the Microchannel.}.

### 2.2.2 Multiprocessor Interfaces

The networks in multiprocessor systems are internal and homogeneous, as opposed to the external networks connecting workstations, in that they connect together the processors of a single multiprocessing machine. The networks in these machines are more closely-coupled to the processor of a node. In the design of the MOSAIC node [35, 39], the packet interface to the network is on-chip and lies on the internal memory bus. The communication controller in the MDP [12] is also on-chip. In the Transputer [27], too, the on-chip links connect the network to the internal data bus. These systems all use custom CPU designs. We now describe two network interfaces which connect a multiprocessor network to a more standard RISC architecture.
Alewife cache controller Interface

In the Alewife distributed-memory multiprocessor [2], the interface of each processing node with the network is done via the cache controller [32]. The cache controller is compatible with a modified SPARC chip called SPARCLE.

We can see in Figure 2-4 how the network is attached to the cache controller. In the Alewife machine, if the processor tries to access data which is not in the cache, and which is in some remote node, a cache miss occurs, and the controller, recognizing the fact that the address is not local, sends out a transaction to the network requesting that data. The processor, meanwhile, switches to another process. When the data arrive, the controller places it in a small buffer, called the transaction store [33]. When the processor finally switches back to the original process and tries to access the same data, it is missed in the cache, but this time it is found in the transaction store, and read in to cache and registers. The transaction store, thus, appears as a second-level cache.

Thus, we see that in this approach, the network appears to reside in the cache. An interesting point to note here is that the network is really transparent to the user. Messages are sent to the network as a side-effect of accessing certain addresses.

---

6Diagram adapted from [2].
**Message Coprocessor Interface**

The Message Coprocessor is part of the *T project of the Computation Structures Group, at the Laboratory for Computer Science at MIT [38]. *T is the name of a node architecture which employs features from traditional von Neumann processors as well as dataflow processors, and which will be used in a multiprocessor environment. In terms of its message functionality, the *T coprocessor (being implemented as an on-chip unit in the 88110 by Motorola [22]) is very similar to the coprocessor described in this thesis. There are two other layers of functionality in the *T coprocessor (the scheduling layer, and the statistics gathering layer) which shall not be touched upon, since they are not directly relevant to the discussion.

The *T Coprocessor has Receive and Transmit buffers, each arranged as two queues, (which can be forwarded separately) one for user messages, and one for kernel messages. The Receiver logic is responsible for examining the incoming data, and determining which buffer to put the packet in.

The data is read out of the buffers by message handlers in the main processor. The instructions are sent out by the main processor as if to a coprocessor. This is also how the coprocessor described in this thesis behaves.

While the similarity in functionality is significant, there are some differences in this design compared to the coprocessor of this thesis. First of all there are no instructions that transfer words directly between storage on the *T coprocessor and main memory. Thus, to perform such a transfer, the user would have to use the general registers of the 88110 processor. With the R3000 coprocessor, however, it is possible to effect such a direct transfer.

Further, the two coprocessors are coupled to different RISC processors—the Motorola 88110, and the MIPS R3000. Thus alternate RISC architectures are under study.
2.3 Summary: Issues of Host Interfacing

The brief survey of current work on host interfacing leads us to make some observations. First of all, we see that the host interfaces for *workstations* connecting to networks all seem to attach to the standard I/O channel of the workstation. For example, the first two interfaces described in Section 2.2 attach to the Turbochannel and the Microchannel, respectively.

With the point of attachment of a network on a workstation assumed to be fixed at the standard I/O channel, host interface designers have focussed on certain issues, which they have considered critical in affecting the performance. We can identify two such issues that have been studied. One relates to the *division of protocol processing* between the interface board and the host processor. The other issue concerns the *method of data transfer* between the processor and the interface board.

On the other hand, we also notice that the network interfaces of the *multiprocessor* systems tend to connect the processor to the network at *different* places in the processor system. The Alewife cache controller network interface and the *T coprocessor* interface described above are two different examples.

The idea of this thesis, is to consider using such *alternative points of attachment*, but between a *workstation* and a network. We consider this issue to be the third constituent in the design space of host interfacing. In figure 2-5, we see how the three issues of host interfacing can be considered the axes of a design space. The designer can select the required combination that best suits his or her needs.

In the *transfer method* axis, programmed I/O and DMA are shown as two extremes. However, there do exist transfer mechanisms that could be considered somewhere in between these extremes. The programmed I/O streaming mode in the Microchannel of an RS/6000 [37], for example, allows a burst of data to be transferred, as opposed to just a single word at a time.

We shall next describe these issues in a little more detail. We note that while the first two issues have been identified and studied before [11, 10, 30, 19, 21], the last one has been left relatively neglected, especially in the context of *networked workstations*.
2.3.1 How much processing?

One of the questions that has been asked by designers of host interfaces is how much of the network protocol processing needs to be done by the interface board, and how much by the host processor.

There exists a wide range of host interface adaptors with varying degrees of intelligence onboard. Some adaptors perform only very few functions onboard [14, 42], leaving the rest of the protocol processing to be done by the host workstation. Other designs put most of the protocol processing into specialized hardware on the host interface [42, 29]. Still other network adaptors can be programmed to implement a specific protocol [3]. In addition to this varied range of host interfaces, there are a few studies that have been made that actually contrast some of these approaches. Some of the questions that arise in studying this issue are mentioned below.

Instruction Cycle Count. In [11], the authors describe their work in comparing an interface in which ATM protocol processing is done on the host with an interface in
which that processing is done on the interface board. The paper presents analysis done keeping in mind a generic RISC workstation architecture, in which the host interface sits on the same bus as main memory. The authors derive equations that they use to predict the behavior of each kind of host interface. These equations are based on a study of the different kinds of instructions that are required for segmentation and reassembly processing, and the number of cycles that each instruction takes.

In another study [10], the authors looked at the processing overhead in a popular transport protocol (TCP). They also used instruction count as the method for determining this overhead, but also measured this directly, with a logic analyzer. The authors concluded that protocol processing costs are in fact small compared to the costs of the operating system, as well as the costs of per-byte operations (like checksum and various memory copies). The authors of the study suggest that the protocol processing can be performed by the host processor. The host interface should be designed such that per-byte operations like copying and checksum can be performed on the outboard adaptor, or such that the CPU performs such operations but with a tight and efficient interaction with the interface adaptor.

Host Bus Traffic. Another comparison of “intelligent” interfaces (meaning those that perform some of the protocol functions) and “dumb” interfaces is presented in [30]. In this analysis, the author looks at intelligent interface adaptors not only as a way of reducing the cycle count for the host processor, but also as a way of minimizing the data movement on the host bus, an angle slightly different from that in [11], in which the instruction cycle count is the only motivating factor. The suggestion made is to have the host interface adaptor perform certain aspects of the transport-level protocol such as transport-level checksumming, encryption and decryption, and packetizing. According to the author, giving the host interface such functionality would minimize the flow of data on the host bus thereby easing the bottleneck between the host and the network.
2.3.2 Programmed I/O or DMA?

Another question faced by designers of host-workstation interfaces is whether the interface should follow a programmed I/O model or a DMA model. This issue is orthogonal to the issue of the level of intelligence onboard an interface adaptor, and deals solely with the mode of data transfer between the interface board and workstation memory.

In programmed I/O, the transfer of each word (or short sequence of words) occurs as a result of an instruction issued by the processor. In DMA, the processor issues a few instructions at the start of the transfer, setting various control registers on the adaptor, and it is the adaptor which transfers the words to or from the memory system.

Both kinds of host interfaces have been designed. The network interface described in [14] uses DMA to transfer ATM cells between the network and the host workstation memory. We shall later describe a Turbochannel interface designed in our group (the Vupboard), which uses programmed I/O as the model for data transfer. In [30], the host interface uses a combination of the DMA and programmed I/O models for transfer, depending on the size of the payload to be transferred. Here are some of the questions involved in this issue of data transfer.

Interference with Processor. One fundamental issue relates to how much each mode of data transfer affects the performance of the processor. In the Programmed I/O model, the data travels via the processor's registers. This not only uses up processor cycles, but also uses up register space. In the DMA case, the data travels directly between the network and memory. This asynchronous mechanism keeps the processor relatively free to perform other operations. However, the fact that the DMA controller is also a memory bus master is cause for some degradation in the performance of the processor, because memory cycles are now being used up by the DMA controller, thereby causing the processor to stall more in its memory accesses.

Overhead and Burst Size. One question that arises in the comparison of programmed I/O versus DMA is the tradeoff between overhead and burst size. The DMA overhead consists of the setup time for the software to go to a device routine,
as well as the actual hardware costs in accessing the memory bus. DMA is a more advantageous data transfer mechanism when large bursts are used (the Turbochannel in a DECstation 5000/200, for example, can support burst sizes of up to 512 bytes), as the overhead incurred in setting up for the DMA transaction is then amortized over a large chunk of data. However, in some environments, DMA needs to be done for small bursts (in some ATM host interface designs [21], the DMA needs to be done on a per-cell basis, to check the header of each cell.) The resulting overhead tends to considerably diminish the advantage to be gained from using DMA.

DMA and Block Boundaries. An interesting though relatively minor problem that arises in DMA is when a cell in memory straddles a block boundary. For example in the Turbochannel, if the packet segment (say 44 bytes) crosses a 2Kbyte boundary, the DMA has to be broken up into two transfers, because transfers across such boundaries are not allowed [14]. This adds some complexity to the host interface, as it has to figure out if such a condition exists, and then break up the transfers into two.

2.3.3 Where to connect?

This issue has been left relatively neglected until now. In the multiprocessor area, there has been some study of different points of connection of network interfaces [25], but we have not found any study that identifies different attachment points in the network-workstation domain.

We can identify several “plug-points” in a workstation architecture that could be potentially used to connect to networks. We have seen in Figure 2-5 some such points on the Location axis. We shall further develop this issue in the next chapter.
Chapter 3

The Connection Issue

*The difficulty in life is the choice.*

George Moore.

While setting out to design an interface, the designer should first consider the problem of figuring out where to fit the network interface on a workstation. We intuit that the other two issues, namely the level of protocol processing on-board, and the mode of data transfer, do indeed affect performance, but that the point of attachment might have an even greater impact.

As we saw in Section 2.3.1, studies that have compared interfaces with different levels of processing on-board the interface adaptor (with the remaining processing being performed on the host) have concluded that the majority of the work in a host interface relates to data-moving instructions, and hence the level of processing on the host interface board is not that significant a factor [10] in affecting performance. Calculations made in [11] find this difference to be at most 60%.

The mode of transfer might be a larger factor in affecting throughput. In the Turbochannel, the DMA-type transfer rates are higher (about 5-10 times), depending on the workstation [16]. In the Microchannel, this difference is also around 6-9 times [37].

The point of attachment has the potential of making as large, if not a larger difference in the performance. Consider, for example, that the Microchannel in an
RS/6000 can sustain a maximum throughput of 320 Mbps in streaming mode, and the cache-memory bandwidth is 3200 Mbps [23, 37]. This is a factor of about 10.

The intent of the above discussion is merely to illustrate that the potential performance impact of the attachment issue could be larger than the first two, and hence merits some study. In Chapter 6, we present the results of some experiments which compare the throughput performance of two host interfaces that attach to different points on a workstation.

We first present an outline of our model of a memory hierarchy system in Section 3.1. In Section 3.2, we show how this model can be used to find the best point of connection. Then, in Section 3.3, we traverse this model, and study the different attachment points for a network. Finally, in Section 3.4 we apply our connection model to some commercial workstations, and see what attachment point might be suited for that particular workstation.

3.1 Model Connection Architecture

It is unfortunate that the network finds no place of its own in today’s workstations! Until that happens, the host interface designer must live with the systems that exist, and try and make the network emulate some other existing module of the workstation.

We now model the hierarchical memory-I/O system of a conventional workstation. With this model, we can try to identify the different attachment points for a network. We will also attempt to find what bottlenecks might exist with certain attachment points. Thus, the aim of this model is not to identify the highest throughput that might be achieved, rather, it is to find where to connect the network to the workstation in order to achieve that highest throughput.

A study of most of today’s RISC architectures reveals a remarkable similarity in the organization of the various modules of the memory and I/O system. The architectures we looked at were the MIPS R3000 [28], the Motorola 88100 [36], the IBM RS/6000 [4], and the SPARC [34]. We found that we could outline a generic memory-I/O architecture, which could describe any of these commercial RISC engines. This
"Model Connection Architecture" is shown in Figure 3-1. The figure reflects the functional relationship of the various modules of a memory hierarchy system, and does not necessarily describe the physical placement of the various modules.

Figure 3-1: Model Connection Architecture

In Figure 3-1, we see the different modules that a network adaptor can emulate, to make the network appear to reside at the different levels of this hierarchical architecture. Thus, the network can be made to appear to reside in an I/O device of the workstation, in main memory, in cache\(^1\), or in registers. We shall see later some issues that arise when the network adaptor emulates each such level.

The arrows in our connection model reflect the different flows of data possible in a typical workstation. Each arrow is labeled with what we call the normalized

\(^1\)Some systems might have several levels of cache hierarchy. In our model, however, we shall deal with a single-level cache. It might be an interesting study to see how this model might be extended to account for a multi-level cache. Also, the cache that we refer to in this chapter is the data cache. We assume that there exists another cache for instructions, but that is not shown in the diagram.
throughput (we shall explain this term shortly), and the subscripts in the labels provide an easy indication of which modules that particular arrow interconnects. The first letter of the subscript denotes the source of the data, and the second letter denotes the destination. Thus, $T_{CM}$ denotes the normalized throughput of data transfer from Cache to Memory.

It is important to note that our model distinguishes between the two directions of flow that exist between two modules. This is necessary because this difference can be quite significant in some cases. For example, in a DECstation 5000/200, the data transfer from the registers to the I/O bus is greater than the data transfer from the I/O bus to the registers (i.e. $T_{RI}$ is greater than $T_{IR}$) by a factor of more than 2 [16]. This distinction between the directions of flows of transfer also relates to the interesting research question of Bidirectional versus Unidirectional network data flow. Typically in a host interface, data will flow in one direction, at any given time. Network switches, on the other hand, can transfer in both directions at the same time. The determination of whether the switch or the interface is faster, then, depends on whether one decides to consider the fact that the network switch is capable of bidirectional transfer.

In order to provide an easy comparison between different workstations, we shall deal with normalized throughputs, rather than the raw throughput numbers. To obtain these ratioed throughputs, the raw throughputs for a particular workstation are normalized to the throughput between the processor and the registers for that architecture. We shall define $R_{PR}$ as the throughput from the processor to the fixed point (general purpose) registers for a particular workstation. We shall assume the same throughput in both directions. Thus,

$$T_{PR} = T_{RP} = 1$$

(3.1)

In determining the raw throughputs for a particular workstation, we should also realize that many of the data transfers can occur in "burst mode". Bursts of data are usually transferred at a higher rate than single word transfers, because the costs of
setting up a transaction can get amortized over the entire burst. For example, in the Microchannel, the initial set up time for a transfer is 100 ns, and the actual transfer of one word also takes 100 ns. Thus, a basic transfer takes 200 ns. However, if burst mode is used, the initial set up time of 100 ns occurs just at the beginning, and a word is transferred every 100 ns thereafter. If the burst size is large enough, the set up time becomes negligible compared to the total time to transfer the burst, and now the transfer rate is almost twice that of the basic transfer [37].

For the purposes of this discussion, we shall assume that the network data we deal with is in large enough bursts, so that when we consider a particular workstation, we shall deal with "burst mode" figures, if any. Thus, in our connection model, we are concerned with the maximum possible throughput between any two modules.

### 3.2 Methodology: Applying the Model

We now have a connection model of a workstation which we can apply to a specific architecture to try and identify which point of attachment might be the most appropriate for a network adaptor. The first step is to determine all the various raw throughputs in the workstation, and from that, the normalized throughputs $T$.

The next step is to take each point of connection in turn, and trace out the flow of data between that point and the final destination of the data. We discussed before (Section 2.1) that the level of interaction between the processor and the network data must be considered when evaluating and designing host interfaces. Thus, in tracing the path of data through our connection model, we must have a particular mode of processor-network interaction in mind—whether the processor needs to touch the network data, or not.

For each path that we trace out to and from each point of connection, we then find the path normalized throughput, $P$. In the simplest case, we could define $P$ for a particular path as the minimum $T$ encountered between any two adjacent modules of that path. Thus, if $[1, 2, \ldots, N]$ are the various modules along a path, from module 1 to module $N$, and $T_{12}, T_{23}, \ldots, T_{(N-1)N}$ are the corresponding normalized throughputs
between the modules, then a simple definition of the path normalized throughput for that path is,

\[ P_{1N} = \min(T_{12}, T_{23}, \ldots, T_{(N-1)N}) \]  

(3.2)

This relation, however, holds true only in cases in which the packet of network data is moved only once before reaching its destination. We need to modify this relation when the network data is moved twice, for example in cases when the network data has to be first touched by the processor, and then put in some buffer in main memory.

Let us consider, for example, the receive direction of the latter case. Typically, the innermost data-moving loop will consist of a number of instructions bringing a packet, say a cell, of the network into the processor's registers. This will also contain some instructions that perform whatever protocol or application-specific operations need to be performed on the body of the cell, such as the checksum. Then, the loop will contain instructions to put the cell into an appropriate location in memory. No matter where we connect the network, the protocol and application-specific operations on the cell will take the same number of clock cycles, so we can ignore these instructions, since the aim of this model is to study in what ways the throughput will differ with different points of attachment.

Thus, we are left with instructions that move the entire cell once into the registers (from wherever the network is attached), and an equal number of instructions (assuming the header is also written back) that move the entire cell into its location in memory. To put this in general terms, we can say that the path of the data now consists of two segments, say from module 1 to module \( M \) and from module \( M \) to module \( N \). We can find the path normalized throughputs \( P_{1M} \) and \( P_{MN} \) by using the relation 3.2 above. Then, we define the total normalized throughput index to be,

\[ \frac{1}{P_{1N}} = \frac{1}{P_{1M}} + \frac{1}{P_{MN}} \]  

(3.3)

This relation can be generalized even further, in case more than 2 copies of the data need to be performed, but we shall not consider those cases here. The reciprocal

35
relationship comes from the fact that the overall throughput is expected to decrease as the number of copies of the data increases. A quick example to get a feel for this relation follows.

Let us assume that our network resides in some uncached memory locations, and reads and writes to that region occur at the same rate, $T$. Our receive routine reads in the data from the network, then writes it to an appropriate location in memory, which also happens to be an uncached area. Now, using 3.2 and 3.3, we get a total normalized throughput index, $P$, of $T/2$. This is what we would expect, since we have moved each cell twice to put it in its final destination, and thus, in this case, have taken twice the amount of time to move each cell.

It should be stressed again, that this total normalized throughput index, $P$, should be viewed as an index of the maximum throughput that can be achieved with a particular point of connection. The actual observed throughput might be lower. In the above example, there could be some other device unrelated to the network interface, such as a DMA controller, that might be using up some memory cycles, and thus we might observe a throughput that is lower than $T/2$. However, this index provides us with a way of comparing different points of connection.

For example, in the above environment, we might now want to consider a network interface such that the network resides in the registers of a coprocessor, say. Let us say that the throughput between the coprocessor registers and general purpose registers is $2T$, and our receive routine brings in the cell to the general purpose registers, and then puts the cell in its uncached location in memory. Using 3.2 and 3.3, we get a total normalized throughput, $P$, of $2T/3$. Thus we would conclude that fitting the network as a register would give us a higher maximum throughput than fitting it as memory.

So far, in the examples we have considered, the data has flowed between two modules only, on each segment along its path. We might be tempted to consider the relation 3.2 to be redundant, since these examples have had only one $T$ to consider along each segment. However, there can be cases in which data can flow through another intermediate module on a segment of its path. For example, if data is read
into the registers from a cached location in memory, then we have to consider both $T_{MC}$ and $T_{CR}$, if we need to find the maximum throughput between (cached) memory and registers.

In summary, the methodology that we use to apply our connection model is as follows:

1. Parametrize the throughputs for a given workstation architecture.

2. Identify the points of attachment.

3. Evaluate the normalized throughput indices for the different approaches.

In Section 3.4, we illustrate this methodology by analyzing some workstation architectures as case studies.

### 3.3 Points of Attachment

We now analyze the different points of network attachment on our connection architecture, and apply our model on them. We shall consider the four cases mentioned before, in which the network data appears to reside in an I/O device, in main memory, in cache or in registers. For each point of attachment, we shall consider the scenario in which the processor needs to touch the data. We shall then briefly consider some cases of interest in which the data does not need to be touched by the processor. Further, for each path of data, the performance might also depend on the direction of flow. Thus, we have several different scenarios to consider.

We have so far defined our model in terms of throughput only. This will provide us with only a very rough estimate of what to expect, in terms of interface performance. However, as we go through the following cases, we shall discuss some other issues, such as latency, cache line size, cache-write policy, DMA interference which can affect performance, and which we should also keep in mind.

It should be noted that while we examine the different points at which a network may reside, we are not providing implementations of how the network interface should
be designed to achieve this effect. The actual implementations depend on the designer, who may want to consider the feasibility, ease and cost of a particular approach, before deciding on the implementation.

3.3.1 Network Emulates I/O Device

First, let us consider the case in which the network interface adaptor emulates an I/O device sitting on the I/O bus of a workstation. The flows of data are shown in Fig 3-2\(^2\). This is the model of host interfacing that is the most prevalent in the world of networked workstations [14, 42, 11].

![Diagram of data flows in an I/O device]

**Figure 3-2: Data Flows: Network in I/O Device**

There are basically three routes to consider. One path includes a direct segment

---

\(^2\)Note that in this and all subsequent figures of Section 3.3, the arrows indicate the receive direction of flow. The transmit direction is traced by reversing the arrow.
<table>
<thead>
<tr>
<th>Path</th>
<th>Throughput Index  ( P )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Receive</td>
</tr>
<tr>
<td>IO1</td>
<td>( \frac{1}{P} = \frac{1}{T_{IR}} + \min(T_{RC}, T_{CM}) )</td>
</tr>
<tr>
<td>IO2</td>
<td>( \frac{1}{P} = \frac{1}{\min(T_{IM}, T_{MR})} + \min(T_{RC}, T_{CM}) )</td>
</tr>
<tr>
<td>IO3</td>
<td>( \frac{1}{P} = \frac{1}{\min(T_{IM}, T_{MC}, T_{CR})} + \frac{1}{\min(T_{RC}, T_{CM})} )</td>
</tr>
</tbody>
</table>

Table 3.1: Normalized Throughput Index: Network in I/O Device

between the register and the I/O network device. The other paths involve a DMA between main memory and the I/O device, then transferring the data from memory, the difference being that in one case the DMA takes place on an uncached area of memory, and in the other, on a cached area of memory. These three paths are labeled IO1, IO2 and IO3 respectively in Figure 3-2. We have assumed that incoming data needs to ultimately reach some buffer in cacheable memory, and outgoing data comes from some buffer in cacheable memory\(^3\). This assumption is made for all the cases we study in the entire Section 3.3. We can now use our model to arrive at a *total normalized throughput index* for each of these paths (see Table 3.1). This table presents a simple approach, in which only the maximum \( T's \) have been considered. For a more detailed analysis, several other factors should also be taken into account, which we explain below.

1. One issue that should be considered in the receive direction (for any of the paths IO1, IO2 and IO3) is whether the cache is write-through or write-back. If we assume a write-through cache, then the effective throughput of the register-to-memory segment can be predicted by finding the minimum of \( T_{RC} \) and \( T_{CM} \). This is because data is written both to cache and memory, and the entire transfer can take place only as fast as the slower transfer. If we assume a write-back cache, we only need to consider \( T_{RC} \), since some other system module, like a cache controller, will be responsible for transferring this data into memory at

---

\(^3\)We could very well have assumed that these buffers exist in uncached memory, or even on an I/O device, as in the case of a display frame buffer. The discussion would be valid in all cases, with slight modifications to our throughput predictions.
some later time when those locations in cache need to be flushed. In all the examples that follow, we shall assume a write-through cache, but also keep in mind how to change the calculation accordingly for a write-back cache.

2. There is another issue relating to caches. Let us assume that the cache line size in bits is \( c \). When reading from the cache, we observe that a cache miss will occur after every \( c \) bits of loads, because when a missed word is fetched from main memory, an entire cache line is loaded from memory into cache at the same time. The expected throughput in Table 3.1 is based on the assumption that \( c \) is very large, and so a cache miss will occur infrequently enough to be ignored. However, a more accurate analysis should account for the cache miss. Let us assume that a cache miss results in a memory access latency of \( l_{MEM} \) seconds, and \( R_{PR} \) is the raw maximum throughput rate between the processor and the registers in bits/second. Then the raw maximum throughput (as opposed to the normalized throughput) from the memory to cache is \( R_{MC} = T_{MC} R_{PR} \). We can then arrive at the new raw throughput from memory to cache by dividing the number of bits transferred between every cache miss by the time taken between every cache miss.

\[
R_{MC_{corr}} = \frac{c}{(\frac{c}{R_{MC}} + l_{MEM})} \tag{3.4}
\]

From this, we obtain the corrected normalized throughput from the memory to cache, which can then be substituted for the original value used in Table 3.1 for path 3.

\[
T_{MC_{corr}} = \frac{R_{MC_{corr}}}{R_{PR}} = \frac{c}{R_{PR}(\frac{c}{R_{MC}} + l_{MEM})} \tag{3.5}
\]

So, if we want a more detailed analysis, we should substitute \( \min(T_{MC_{corr}}, T_{CR}) \) for \( \min(T_{MC}, T_{CR}) \) whenever it appears in Tables 3.1 through 3.4.

3. A third issue (in paths IO2 and IO3) is the effect of DMA on the memory
bandwidth. Since both the processor and the I/O device would be writing to the memory, the throughput numbers associated with memory transfers can be expected to be less than the maximum numbers we had assumed for that particular workstation ($T_{CM}$, $T_{MC}$, $T_{MI}$, $T_{IM}$, $T_{RM}$, and $T_{MR}$). One way of dealing with this problem is to apply a correction factor (as we did for the cache) that takes into account the percentage of memory bandwidth utilization for each device on the memory bus. However, we do observe in some systems, that the memory system bandwidth is greater than (or nearly equal to) the sum of the maximum throughputs between the processor and memory, and between an I/O device and memory. This implies that the memory system can, in the long run, accommodate transfers with the processor as well as the I/O device, without a significant loss in performance, and hence we can ignore the interference. This turns out to be a fair assumption, in the case of a DECstation 5000/240. We observe that the average throughput between memory and registers is about 420 Mbps (taking the average of the receive and transmit directions, in $Cop6$ in Table 6.4). The corresponding figure for a DMA device is about 430 Mbps (from figures taken from [13]). This gives a combined throughput of about 850 Mbps, which is very close to the maximum memory bandwidth of 800 Mbps [17]. This is the simple assumption made in Table 3.1.

There appears to be no obvious bottleneck that would drive us towards using any one of these paths. Thus, the best strategy would be to actually calculate these indices, and then determine the best option.

### 3.3.2 Network Emulates Memory

The next point of connection on a workstation can be such that the network appears to reside in memory. We have not seen any such network interface for workstations. (The Medusa interface [5] attaches to the graphics bus of an HP Apollo Series 700 workstation. This represents a move towards attachment to the memory system, since the graphics bus is highly optimized for block transfers, and stores to this bus have
only a 1 cycle latency.) However, the multiprocessor world provides examples of such interfaces. For example, in the Message Driven Processor [12], the incoming messages are accessed as data in main memory. We now study some of the paths that data might trace if a network were connected to the workstation such that the network adaptor emulated a memory module of the workstation (Figure 3-3).

![Diagram](image)

Figure 3-3: Data Flows: Network in Memory

<table>
<thead>
<tr>
<th>Path</th>
<th>Throughput Index $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Receive</td>
</tr>
<tr>
<td>MEM1</td>
<td>$\frac{1}{P} = \min(T_{MC}, T_{CR}) + \frac{1}{\min(T_{RC}, T_{CM})}$</td>
</tr>
<tr>
<td>MEM2</td>
<td>$\frac{1}{P} = \frac{1}{T_{MR}} + \frac{1}{\min(T_{RC}, T_{CM})}$</td>
</tr>
</tbody>
</table>

Table 3.2: Normalized Throughput Index: Network in Memory

Again, the case in which the processor needs to touch the data implies that the network data has to go via the registers. The different paths that exist arise from
the fact that the network adaptor might lie in cached or uncached area of memory (paths MEM1 and MEM2, respectively). The normalized throughput index for each path is given in Table 3.2.

We can see that these paths are very similar to the paths 2 and 3 of the I/O device case, except that the segment between memory and I/O are not involved. Thus, the same discussion regarding the cache write strategy, and the cache line size applies here too, which we shall not repeat.

Again, the best way to predict which of the two paths is better is to determine the throughput index, according to the formulae of Table 3.2. However, for an intuitive speculation, one could observe that accesses to cacheable memory are much faster than accesses to non-cacheable memory. Thus, in the receive direction at least, path MEM1 would be the faster way to go.

### 3.3.3 Network Emulates Cache

The network interface could also be designed so that the network data appears to reside in cache. Thus, the interface could be designed by modifying the cache controller of the workstation (an approach taken in the Alewife project [32]). We see in Figure 3-4 the path that can be taken by the data. Table 3.3 shows the formula for finding the throughput index for this path.

<table>
<thead>
<tr>
<th>Path</th>
<th>Throughput Index P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Receive</td>
</tr>
<tr>
<td>CC1</td>
<td>$\frac{1}{P} = \frac{1}{T_{CR}} + \frac{1}{\min(T_{RC},T_{CM})}$</td>
</tr>
</tbody>
</table>

Table 3.3: Normalized Throughput Index: Network in Cache

If data has to reach the general purpose registers, there is one path that can be followed, shown as CC1 in the above figure. Thus, the idea is that the data resides at a particular block of addresses in cache, and a cache hit will always occur. Thus, the path between the network and the registers depends on $T_{RC,CR}$. Since this value is usually very high (approaching 1) in most workstations, we can expect that the registers will be fed with network data at a high rate.
3.3.4 Network Emulates Registers

This is yet another location in the workstation in which the network data can be made to appear to reside. This idea is fairly new, and is exemplified by the coprocessor design presented as part of this thesis. The coprocessor approach of the *T project, in the multiprocessor area, was started at around the same time as our design.

<table>
<thead>
<tr>
<th>Path</th>
<th>Throughput Index P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Receive</td>
</tr>
<tr>
<td>REG1</td>
<td>$\frac{1}{P} = \frac{1}{T_{RR}} + \frac{1}{\min(T_{RC}, T_{CM})}$</td>
</tr>
</tbody>
</table>

Table 3.4: Normalized Throughput Index: Network in Registers

Clearly, this approach puts the network the closest to the processor. The path between the network and the general purpose registers can now be covered simply
by the use of register-to-register move instructions, which usually occur at the high throughput of $T_{RR} = 1$. This path REG1 is shown in Figure 3-5, and the throughput index is shown in Table 3.4.

In all the "Data Processed" examples, we have seen many different paths. However, for the sake of comparison, we have kept the final destination/source in the workstation to always be some area in cached memory. In other words, the path from the registers to some receive buffer in cached memory (for the receive direction) and from some transmit buffer in cached memory to registers (for the transmit direction) has been the same in all cases. The difference in these paths, then, arises from the different routes that the data has to travel between the network and the general purpose registers. Thus, one quick way of estimating the highest throughput in all the examples is to determine the fastest way for data to travel simply between the network and the registers, without considering where the final (receive) destination or
initial (transmit) source of the data lies in the workstation. With this quick estimate, we can see that, if the fastest rate at which data can be transferred in and out of a register is from and to another register, this approach will yield the best performance of any of the "Data Processed" examples.

3.3.5 Data Not Processed

We now consider some cases in an environment in which the processor does not have to touch the data. Figure 3-6 illustrates some of the more interesting paths along which data might flow. We can make several comments regarding these cases.

![Diagram of data flows](image)

Figure 3-6: Data Flows: Data Not Processed

1. If the processor does not need to touch the data, then we can identify two basic paths from the I/O device, to main memory. One is a direct path between the I/O device and memory, and the other path involves the registers (labeled paths
IO4 and IO5, in Figure 3-6). We do observe, however, that the DMA mechanism of most I/O buses provides a faster transfer rate than the programmed I/O mechanism\(^4\) (i.e., typically, \(T_{IM} > T_{IR}\) and \(T_{MI} > T_{RI}\)). Thus, path IO4 clearly seems to be a better choice than path IO5 (note that IO5 is the same as IO1 of Figure 3-2).

2. Involving the registers as in IO5 might be an even worse choice than the throughput index shows. Besides the fact that the throughput between registers and I/O is generally lower than that between memory and I/O, the consumption of processor instruction cycles further degrades the performance of the system.

3. The fact that the cache is also involved in path IO5 is also a further reason for loss of performance. This is because cache misses will occur in both directions (for every cache line), since network data is new and will never be in the cache from before. Thus, the cache will be filled up with network data, and this will result in a loss of locality for the processor as a whole.

4. The approach in path IO4 (DMA) does have the problem regarding the interference of the DMA controller with the processor for access to memory, which we should keep in mind for particular workstation architectures. Despite this point, we feel that the other factors are much stronger in affecting performance. It seems clear that in this case, path IO4 is the better choice for the highest performance.

5. If the network were to appear to reside in memory, a possible path for data to flow is shown as MEM3. The network adaptor would then emulate a memory module. Any other device needing the network data would then DMA the data directly out of the space occupied by the "network-memory module". This scheme would work if the data present in this network module were completely ready (for example, the packets of data were arranged in the order specified by the headers) for consumption by the workstation device needing the network

\(^4\)This is mainly due to the possibility of burst transfer in DMA.
data. Thus, some mechanism would have to be provided to keep the network data in a state ready for consumption (the solution might lie in hardware, or in giving the processor programmed I/O access to the header bytes). This discussion focussed on the receive direction, but similar arguments can be made in the transmit direction.

6. If the network were to reside in cache, the data would have to actually be pushed down the hierarchy into memory. This matches the complexity of a write-back cache controller. Further, the network software would have to have control of such a device. It appears that the network is almost too close to the processor. Thus, it seems that the memory level is the natural boundary below (and at) which the network attachment makes sense, if the processor does not need to touch the network data. (The words “down” and “below” are mentioned with Figure 3-6 in mind.)

7. If the network were to reside in registers, and if allowed by a particular workstation architecture, data can move directly between memory and the “network registers” (which, in the case of the host interface designed as part of this thesis, happen to be the registers of a coprocessor). This is path REG2, shown in Figure 3-6. If this path is not allowed in a particular workstation, then the registers would have to be involved (as shown in REG1, Figure 3-5). Path REG2 is clearly the better option in this case. First, the data is moved only once. Further, the general purpose registers do not get filled up with network data. This might result in faster programs, as fewer general purpose registers would need to be saved in a context stack upon entering the data-moving inner loop routines.

3.4 Case Studies

We have gone through the different points of connection, and identified the various paths that can be taken from each such point. We have indicated the normalized
throughput index for each path. We should be careful not to use these throughput numbers too literally, as there are factors which should be taken into account that might affect the throughput index. The cache-write policy, the using up of memory cycles by DMA, the cache line size, memory access latency, the saving of general purpose registers, and the loss of cache locality are some of the issues which we examined.

Having identified such points, we would like to compare the different approaches, and try and estimate which would be the best one. We now consider some actual workstation architectures, and apply our throughput index equations to determine which point of connection might be the most suitable for that workstation.

### 3.4.1 IBM RS/6000

The various throughputs and normalized throughputs for an RS/6000 running at a clock rate of 25 MHz are shown in Table 3.5. Some numbers were found explicitly in several technical articles [4, 23, 37], and others were arrived by studying the architecture (with information such as the width of buses and clock speeds coming from these articles). This machine has a write-back cache, with a cache line size of 128 bytes. We see that this machine has very high throughputs between the various modules, especially the cache-to-memory bandwidth, which is 4.

<table>
<thead>
<tr>
<th>Path</th>
<th>Maximum Throughput</th>
<th>Path</th>
<th>Maximum Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Raw (Mbps)</td>
<td>Normalized</td>
<td></td>
</tr>
<tr>
<td>PR</td>
<td>800</td>
<td>1</td>
<td>RP</td>
</tr>
<tr>
<td>RC</td>
<td>800</td>
<td>1</td>
<td>CR</td>
</tr>
<tr>
<td>CM</td>
<td>3200</td>
<td>4</td>
<td>MC</td>
</tr>
<tr>
<td>MI</td>
<td>320</td>
<td>0.4</td>
<td>IM</td>
</tr>
<tr>
<td>RI</td>
<td>320</td>
<td>0.4</td>
<td>IR</td>
</tr>
<tr>
<td>RM</td>
<td>N.A</td>
<td>-</td>
<td>MR</td>
</tr>
<tr>
<td>RR</td>
<td>800</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.5: RS/6000 Throughputs**

---

Some lower-cost models have a cache line size of 64 bytes. Also, the machine for which we could obtain this information has an older version of the Microchannel, hence the low throughputs.
In order to find the best point of attachment, we should first describe the intended environment for this host interface. Let us assume that the network data needs to be touched by the processor. Further, all areas of memory are cached, in the RS/6000\(^6\). The normalized throughput index has been calculated for a few cases, with the network being attached at different locations (see Table 3.6). The "path" column refers to the labeled flows of data in the appropriate diagrams (Figures 3-2 to 3-5).

<table>
<thead>
<tr>
<th>Attachment</th>
<th>Path</th>
<th>Throughput Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Receive</td>
</tr>
<tr>
<td>I/O Device</td>
<td>IO1</td>
<td>0.28</td>
</tr>
<tr>
<td>I/O Device</td>
<td>IO3</td>
<td>0.28</td>
</tr>
<tr>
<td>Memory</td>
<td>MEM1</td>
<td>0.38–0.49</td>
</tr>
<tr>
<td>Cache</td>
<td>CC1</td>
<td>0.5</td>
</tr>
<tr>
<td>Registers</td>
<td>REG1</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 3.6: Normalized Throughput Index: RS/6000

The ranges of values are a result of the range of memory read latencies. We have taken a range from one to twenty 25 MHz clock cycles. The throughput from the cache to the registers has been corrected to take into account this latency (thus, instead of \(T_{MC} = 1\), we are using \(T_{MC_{corr}} = [0.97 – 0.62]\)).

The rough estimates provided by the throughput index suggests that the worst results would be obtained by the I/O device approach. The path that we have shown for this approach, is the "programmed I/O" path (labeled IO1 and IO3, in Figure 3-2), clearly indicating that the I/O bus is the bottleneck. The other three approaches shown, all predict about the same maximum throughput. The cache and registers approaches are problematic in that they are not practical to design. The RS/6000 does not support registers other than those already defined (for example, no external custom coprocessors are supported). And the cache controller is also a proprietary unit, making it difficult for others to emulate it.

---

\(^6\)It is possible to write directly to memory, bypassing the cache unit, but that mechanism is for diagnostic purposes only.
Thus, the memory approach seems the best approach for the RS/6000. The throughput index predicts high performance, and the design of a network interface that emulates a memory card is possible. The predicted throughput is around 300 Mbps to 400 Mbps, with data traveling via the processor's registers. The best performance possible in the Microchannel interface described in [42] is 320 Mlps\textsuperscript{7}, but since this figure is for data not being touched, it is difficult to make a comparison with our estimates. Clearly, a lower throughput will be expected in the latter case, if the data has to be touched by the processor.

3.4.2 DECstation 5000/240

The throughput numbers for the DECstation were found in [17, 16]. Some values were calculated based on information given, such as the cache write policy, and memory latency. The cache is write-through, and the cache line is 32 bytes. The memory read latency is taken to be 8 cycles.

Again, we assumed an environment in which the processor will need to touch the data, and we calculated the normalized throughput index for some interesting paths (Table 3.8), with the network residing at different levels in the hierarchical connection model that we described earlier. We should consider the latency of memory accesses. Assuming read latency is 8 cycles, the cache line size is 32 bytes. Equation 3.5, we get $T_{MC_{corr}} = 0.38$, which we use in our calculations to obtain the index results below.

The values for RI and IR have been taken from specifications for the Turbochannel [16], which state that the minimum cycles for writes are 3, and reads are 8, for a DECstation 5000/200. Since we did not find any published numbers for the DECstation/240, we are assuming the same values\textsuperscript{8}.

These numbers predict that putting the network at the cache or the registers would yield the best performance, in terms of throughput. The nearer the network

\textsuperscript{7}A scaling study performed by the UPenn group predicts a throughput of 600-1000 Mbps with some chip changes and a new Microchannel version.

\textsuperscript{8}We experienced in our performance experiments, that the Turbochannel performance is actually much worse in a "240" compared to the "200", which we believe to be due to the mismatch between the Turbochannel clock and the processor clock. This point is mentioned later, in Section 6.3.4. Thus, the normalized throughput values for RI and IR are optimistic.
<table>
<thead>
<tr>
<th>Path</th>
<th>Maximum Throughput</th>
<th>Path</th>
<th>Maximum Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Raw (Mbps)</td>
<td>Normalized</td>
<td>Raw (Mbps)</td>
</tr>
<tr>
<td>PR</td>
<td>1280</td>
<td>1</td>
<td>RP</td>
</tr>
<tr>
<td>RC</td>
<td>1280</td>
<td>1</td>
<td>CR</td>
</tr>
<tr>
<td>CM</td>
<td>800</td>
<td>0.62</td>
<td>MC</td>
</tr>
<tr>
<td>MI</td>
<td>640</td>
<td>0.5</td>
<td>IM</td>
</tr>
<tr>
<td>RI</td>
<td>266</td>
<td>0.20</td>
<td>IR</td>
</tr>
<tr>
<td>RM</td>
<td>800</td>
<td>0.62</td>
<td>MR</td>
</tr>
<tr>
<td>RR</td>
<td>1280</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.7: DECstation 5000/240 Throughputs

<table>
<thead>
<tr>
<th>Attachment</th>
<th>Path</th>
<th>Throughput Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Receive</td>
</tr>
<tr>
<td>I/O Device</td>
<td>IO1</td>
<td>0.07</td>
</tr>
<tr>
<td>I/O Device</td>
<td>IO3</td>
<td>0.23</td>
</tr>
<tr>
<td>Memory</td>
<td>MEM1</td>
<td>0.23</td>
</tr>
<tr>
<td>Cache</td>
<td>CC1</td>
<td>0.38</td>
</tr>
<tr>
<td>Registers</td>
<td>REG1</td>
<td>0.38</td>
</tr>
</tbody>
</table>

Table 3.8: Normalized Throughput Index: DECstation 5000/240

resides in the memory hierarchy, for this architecture, the faster the rate at which the processor's general purpose registers can be fed with network data. We would, thus, be driven towards a cache or register approach. In Chapter 6, we compare these predictions which measured throughputs of a Turbochannel Programmed I/O board (the Vupboard, described later), the Turbochannel DMA board of Bellcore [14], and also the coprocessor approach which puts the network in the registers.

### 3.5 Conclusion

One of the conclusions we can derive from this study is that different architectures drive us towards different approaches. For example, in the DECstation 5000/240, we note that if the network appeared to reside in memory, the performance would not be expected to be as high as if it were attached at the register/cache level. However in the RS/6000 case, the cache-memory bandwidth is very high, and the memory
solution appears to be about as good as the register/cache approach, and hence is a viable option.

We have seen in the case studies, that the programmed I/O approach, with the host interface attaching to the standard I/O bus tends to yield low throughput, and the main bottleneck in this path is the slow accesses between this bus and the registers of the workstation.

In the DMA approach (I/O Device, path 3), the bottleneck occurs because the memory system has to be traversed twice before data reaches its final destination (assuming that the data has to be touched by the processor). However, workstation architects have been placing increasing emphasis on memory system design [8]. The RS/6000 is a good example of a very high-bandwidth memory system. In the Alpha-based[18] workstations, too, the memory system connects to the cache with a wide datapath. This trend suggests that these systems could make DMA a viable option even in cases in which the processor needed to touch the data.
Chapter 4

The Coprocessor Approach

_We often get in quicker by the back door than by the front._

Napoleon I.

4.1 Introduction

We have seen how the focus of research on host interfaces has tended to neglect the issue of finding the best point of attachment in a workstation. We discussed how this overlooked issue might be of importance in the quest for faster host interfaces.

In order to study this issue, we chose the DECstation 5000 as an experimental platform, since we already had an interface board for this workstation against which we could compare our results. This interface—the Vupboard—is described later in Chapter 6. Also, a DMA host interface board had been designed at Bellcore [14]. Further, the coprocessor interface for an R3000 is well documented. Thus, the coprocessor approach seemed to be an interesting choice.

In Section 4.3, we explain at an architectural level what exactly is meant by a “coprocessor-style” host interface. But before describing this approach, we first discuss (in Section 4.2) certain aspects of the working environment of this project, and how they are relevant to the approach taken in this thesis.
4.2 Background

The Viewstation Project is first described. The host interface of this thesis, while applicable to other environments, was initially designed for the Viewstation Project. We then discuss the compatibility of RISC workstations and ATM networks. Next we summarize some choices we made that define the design space of our host interface.

4.2.1 The ViewStation Project

The ViewStation Project of the Telemedia, Networks and Systems group in the Laboratory for Computer Science involves the design and construction of a distributed video system [41]. Figure 4-1 shows the hardware architecture of the ViewStation Project. The ViewStation essentially consists of switches, links and clients.

![ViewStation Architecture Diagram]

Figure 4-1: ViewStation Architecture

The basic communications infrastructure in the ViewStation Project is the VuNet,
a high-speed ATM network, which uses cells as the communications standard\(^1\). The building blocks of VuNet are links and switches.

In the current implementation, a VuNet switch consists of 4 ports, each of which has a receive and transmit section. The ports are interconnected by a crossbar matrix. Each transmit and receive section is buffered—the buffers can hold up to 256 cells in the receive direction, and 64 cells in the transmit direction. The ports of the switch can be attached to links, host workstation interfaces, or other clients.

The links make up the communications channel between two switches. There currently exist coaxial and fiber optic links in the project. These perform the necessary parallel to serial conversion. The links also perform the remapping of VCI's to determine the path of the cells.

Clients are another important block of the ViewStation project. One key client is the host workstation. We have DEC and SUN workstations which can function as hosts, and other workstations are being considered too. In order for these workstations to be attached to the VuNet, a host interface needs to be designed, and that is the topic of this thesis.

Another important client is the Vidboard, a peripheral device on the VuNet that can capture, process and transmit full-motion color video. It can generate a wide variety of video streams [1].

There are certain characteristics of the operating environment of this project that make the solution presented here particularly appealing. We shall see below why this is so. But it is also important to note that the characteristics in question are not limited to the Viewstation environment, and it is quite reasonable to expect to find these characteristics in other systems also.

\(^1\)In comparison to B-ISDN, VuNet cells are 56 bytes, rather than 53 bytes. The three extra bytes are added to the structure of a B-ISDN cell, to achieve convenient byte-alignment. The bytes are inserted after the HEC field in the header. The interface boards to B-ISDN WANs and LANs will simply remove these three bytes before transmission.
4.2.2 Our Host Interface Model

In Section 2.3, we described what we considered to be axes that define the design space of a host interface. For the coprocessor host interface that is designed in this thesis, we have already explained our choice in the point of attachment. We now talk about the other two issues—the level of on-board processing, and the mode of data transfer—as they relate to our environment.

No Processing on Adaptor

Our focus in host interfacing is to design interfaces which are faster, rather than smarter. The underlying belief in this focus is that the host processor is generally the most powerful engine for performing protocol processing on a workstation, and thus it would be advantageous to try and tap this powerful resource, rather than trying to design custom hardware to do the protocol processing.

Another motivating force in our approach of keeping the on-board adaptor intelligence at a minimum is the realization that workstation processing power is increasing every year, and the technology will soon reach a point at which the host will be able to handle the protocol processing quite easily. This fits in with our research group's ideology that the problems of distributed video systems can be solved essentially in software.

Programmed I/O Data Transfer

The host interface in this design performs data transfers between the network and the host in a programmed I/O style. One reason for this choice of data transfer is that the environment of this project is one in which the processor touches the data. Thus the data from the network needs to go to the host's general purpose registers, rather than main memory directly. This clearly decreases the value of DMA as it deals with main memory only.

With the host processor issuing instructions for data transfer, the programmer has more control over where the network data can go. Thus, the programmer can
bring all the network data into the host registers, or the programmer can put all the network data directly into main memory. The programmer can also bring parts of the network data (like the header) into the host registers, and put the rest into main memory. Thus, the programmed I/O model also provides for a more flexible research environment.

Another reason for avoiding DMA is the increased complexity associated with that model. Thus, simplicity of design was another factor influencing the choice of programmed I/O for data transfer.

4.2.3 Compatibility of RISC and ATM

It turns out to be a matter of great convenience that the cell length of an ATM cell (53 bytes) is such that the cell can fit comfortably into the register space of most of today's RISC workstations [31, 34, 36]. We can thus deal with entire ATM cells at a time, without bothering about breaking up a cell further.

This characteristic is particularly noteworthy, since we are dealing with an environment in which the network data has to reach the processor registers. If the processor did not have to touch the data, then this characteristic would not be significant, as the data would simply have to move directly between the network and main memory.

The host interface design of this thesis is intended for the MIPS R3000 architecture which has 32 general-purpose registers, which are 32 bits wide. Also, as explained earlier, the cell in VuNet is 56 bytes long. Thus, we see that an entire cell can fit in 14 of these general registers, with many registers to spare.

4.3 Architectural Overview

A high-level architectural description of the coprocessor host interface is now presented. We have seen above that our operating environment is one in which the host processor needs to operate upon the network data, and hence the architecture presented below attempts to bring the network data close to the processor.
4.3.1 Programmer’s View

The architecture of the coprocessor interface will be described from the point of view of the user. Specific implementation details of the coprocessor host interface chip appear in Chap 5.

We see in Figure 4-2 the user’s view of the coprocessor interface\(^2\).

![Diagram of Programmer's View](image)

**Figure 4-2: Coprocessor Interface Architecture: A User’s View**

As far as the programmer is concerned, the network data is mapped onto the register file. Thus, data coming in from the network, or on its way out, resides in the coprocessor registers. In order to accomodate the different rates of generation and consumption of network data, network interfaces usually have input and output buffers associated with them. In this architecture too, we define an input and an output buffer. The input buffer would be connected to a transmit port of a network switch, and the output buffer, to a receive port. Each buffer in this architecture is a

---

\(^2\)In Figure 4-2, we can see the parts of the coprocessor interface that are visible to the programmer. A value is “visible” to the programmer if it can be read and also perhaps changed. The shaded parts in the diagram are those that are not directly accessible by the programmer.
First-In First-Out buffer (FIFO). It can contain network data in units of packets (or cells). The first cell to enter a FIFO buffer (whether input or output) will be the first to leave it. The depth of each FIFO buffer is implementation-dependent.

For added flexibility, we can also have more than two FIFO buffers, (or network ports). In the implementation described in Chapter 5, we have designed a total of four ports, each of which can be configured to be input or output in direction.

There are typically two mechanisms through which the programmer can find out if any network data has arrived from the network, for consumption by the host: polling and interrupts. This architecture will support both mechanisms, and this aspect is dealt with in more detail in Section 4.3.1.

The next few issues deal with more characteristics of this architecture, and explain the rest of the blocks which are shown in Figure 4-2.

Relation to Floating Point Registers

The network data appears to reside in the coprocessor registers much in the same way that floating point information resides in the registers of the floating point coprocessor. The floating point coprocessor (R3010) for the MIPS R3000 architecture consists of 32 registers. These registers can be accessed like general purpose registers in the main processor, i.e. they can be read or modified using move or load/store instructions.

In the same manner, the coprocessor registers which contain the network data can be accessed by using move instructions (which transfer data between the coprocessor registers and general purpose registers), or load/store instructions (which transfer data between the coprocessor registers and main memory). Figure 4-3 shows how the network is register-mapped within the framework of the R3000 architecture.

The purpose of Figure 4-3 is to show that the registers of the CPU, floating point unit, and the interface coprocessor are logically equidistant from the memory system. The details of the number of registers shown for the coprocessor are not important from an architectural standpoint, and will be discussed later with the rest of the hardware aspects. For the sake of completeness, it should be noted that Figure 4-3 shows only the “general purpose” registers, and each unit has “control” and “status”
Figure 4-3: Partial Register Map of R3000 with Coprocessor Interface

registers also, which are not shown.

Since the host processor interacts with the registers of the host interface coprocessor and the registers of the floating point coprocessor in almost the same manner, it was possible for us to study the benefits of using a coprocessor interface by reading and writing the floating point registers before the coprocessor host interface chip was designed. It should be mentioned that, in reality, there is a way in which the interface coprocessor and the floating point coprocessor differ in their interaction with the host processor. If it is not ready with a result of an operation, the floating point coprocessor has the ability to stall the host processor. The architecture of the interface coprocessor, on the other hand, is such that the coprocessor cannot stall the main processor. However, in the case of a network interface program, the main instructions issued by the host processor to the interface coprocessor are typically register-to-register moves, or loads-stores, and these are instructions on which the floating point
coprocessor will not stall. Thus, it is reasonable to conclude that the floating point coprocessor provides a fairly accurate reflection of the performance of the interface coprocessor. Details of this performance study are presented in Chapter 6.

Polling and Interrupts

We mentioned earlier that typically host interfaces have certain input and output buffers associated with them. In the case of the receive direction, the host processor needs to know when data has arrived from the network, in other words, when an input buffer is non-empty. There are typically two mechanisms by which this information can be conveyed to the host processor: polling and interrupts, and both these mechanisms can be used with the coprocessor interface architecture defined here.

For the use of a polling mechanism, a Status Register is defined which contains the necessary information of the state of the input (and output) buffers. Basically, a bit in this register is reserved for each input (and output) buffer which indicates whether there is data present in that buffer. Thus, the architecture of the coprocessor interface allows for the use of polling as a detection mechanism for incoming network data.

In the architecture of the coprocessor interface, we also define an interrupt signal, which signals the arrival of data from the network into an input buffer. If there is more than one input buffer, then it is possible to mask the buffers, so that one can choose which buffer(s) can cause the interrupt to happen. In order to specify this masking, a Control Register is defined, which can be written into by the host processor.

To better understand the coprocessor host interface architecture, we should look at what happens to the host processor and the interface coprocessor in both the receive and transmit directions.

4.3.2 Receive Direction

When data arrives from the network (in the form of cells, in the case of the VuNet), the first event that needs to occur is for the host processor to be made aware that data has arrived. We have seen that this can take place by either polling or interrupts.
Once the host processor knows that there is data, it needs to be able to see that data, and determine what to do with it.

It should be noted that when we say that "data has arrived" from the network, we mean that a cell has been placed in an input buffer of the host interface. We shall ignore, for now, the exact details of how a cell is placed there.

To touch the data, the host processor issues an instruction that transfers one cell into the coprocessor register file. This transfer is done in one cycle, i.e. all the bits of the cell are transferred from the input buffer to the coprocessor register file in a single clock cycle. After this operation, the network data is register-mapped, as far as the programmer is concerned.

The host processor can now decide to bring the data into the general-purpose registers of the workstation, in order to operate upon that data. The programmer does this by issuing move instructions that transfer the data from the coprocessor registers to the host registers in word chunks (32 bits). Alternatively, the host can decide to place that network data into an area in main memory. This can be done by issuing store instructions that transfer data from the coprocessor registers directly to main memory in word chunks.

Even if the data needs to go directly into main memory, the host may need to first look at the header, in order to determine where in memory to place the data. The header information can be looked at by transferring the appropriate words from the coprocessor registers to the general purpose registers, and then performing the usual compare operations.

At this point, the host processor has dealt with one cell of incoming network data. However, there is a high probability that there are more cells waiting to be consumed by the host. A lot of cycles would be used up if the host were now to go back to polling or waiting for an interrupt to detect whether there is, indeed, more data waiting in an input buffer. Instead, a conditional signal is defined in the coprocessor interface architecture, which indicates the state of the buffer. This signal is a mirror of the interrupt signal, but can be directly used in conditional branch instructions.

Thus, the host can use such a conditional branch instruction and learn whether
there is more data or not. If there is, the host can repeat the entire process described above. Otherwise, the host can exit this loop and wait for the next burst of network data.

We can see below a sample piece of code that would constitute the inner loop of a receive routine. Since the specific implementation details will be explained a little later, this code is written in simple pseudo-assembly language, and consists of instructions which are common to most of today's RISC processors. The code describes the innermost loop, i.e. the host is already aware (through polling or interrupts) of the existence of incoming data. In the sample code, the transfers the cell data directly into main memory.

# Sample pseudo-assembly code to receive cells.
# All data is written consecutively after base
# address stored in register $1.

set_to_receive  # Some implementation-dependent
                # initialization instruction(s).

start:

cop_receive_cell 0  # Transfer cell from input buffer
                   # to coprocessor register file.

store:

store_cop $c1, 0($1)  # Store data from coprocessor registers
store_cop $c2, 4($1)  # to main memory. The $c prefix refers
store_cop $c3, 8($1)  # to coprocessor registers, and the $1
store_cop $c4, 12($1) # identifies a base address of the
store_cop $c5, 16($1) # main memory. In this case, there are
store_cop $c6, 20($1) # 56 bytes or 14 words in a cell.
store_cop $c7, 24($1) #
store_cop $c8, 28($1) #
store_cop $c9, 32($1) #
store_cop $c10, 36($1) #
store_cop $c11, 40($1) #
store_cop $c12, 44($1) #
store_cop $c13, 48($1) #
store_cop $c14, 52($1) #

add $1, $1, 56       # Update base address.
branch_cop start    # Use coprocessor conditional signal to
                   # go back to start if more cells.
end:

The code above does not show the saving of registers in a stack that must be done upon entering a subroutine. The code can, of course, be made much more complex. The program might need to have some kind of byte-swapping or error-checking. The host might also need to examine the header of each cell and place its payload in an appropriate location. All these functions are not shown above for the sake of clarity.

4.3.3 Transmit Direction

In the transmit direction, it is the host itself which decides to send data out into the network. For now we shall say that the data is in the network when it has been placed into an output buffer of the interface.

The first thing that the host processor needs to do is determine if there is any space available in the desired output buffer. This can be done by reading the same Status Register that was mentioned in Section 4.3.1. Another way is for the host processor to use the same condition signal that is used in the receive direction. This condition signal can be used for both directions. One can control which particular buffer the condition signal is meant to represent by writing to the Control Register. This condition signal is then used with the conditional branch instructions.

If there is room in the output buffer, the host processor then transfers a cell's worth

---

3Another way is for the processor to test this after moving data to the coprocessor registers.
of data into the coprocessor register file. This is done by using load instructions which transfer data from main memory to the coprocessor registers.

To release the data into the network, the host then places that cell into an output buffer. The architecture of this host interface defines a single instruction which transfers all the bits of the cell from the coprocessor registers to the output buffer in one cycle. The cell is now ready to be injected into the network. It is the responsibility of some circuitry on the network side to now transport this cell to its destination. The details of that process are being glossed over, for this discussion.

Typically, there will be more than one cell that needs to be sent out. Since the host is doing the sending, it will be aware of how many cells it wants to send. However, it will still need to check the condition signal before sending each cell, to make sure that there is room in the output buffer. If there is no room, the host can wait until there is space. This process will repeat till the host has sent out all the desired cells.

The sample piece of code below suggests a possible sequence of instructions that are used with the coprocessor host interface architecture described above. Again, simple pseudo-assembly code is used to avoid cluttering the discussion with implementation details.

*************************************************************
# Sample pseudo-assembly code to transmit cells.
# Starting memory address of data to send is in $1.
# Number of cells to send (non-zero) is stored in $2.
*************************************************************

    set_to_transmit        # Some implementation-dependent
                          # initialization instruction(s).
      start:
          branch_cop start     # Use coprocessor conditional signal to
                              # wait till room in output buffer.
          load_cop $c1, 0($1)  # Load data from memory to coprocessor
          load_cop $c2, 4($1)  # registers. The $c prefix refers
          load_cop $c3, 8($1)  # to coprocessor registers, and the $1
load_cop $c4, 12($1)  # identifies a base address of the
load_cop $c5, 16($1)  # main memory. In this case, there are
load_cop $c6, 20($1)  # 56 bytes or 14 words in a cell.
load_cop $c7, 24($1)  #
load_cop $c8, 28($1)  #
load_cop $c9, 32($1)  #
load_cop $c10, 36($1) #
load_cop $c11, 40($1) #
load_cop $c12, 44($1) #
load_cop $c13, 48($1) #
load_cop $c14, 52($1) #

cop_transmit_cell 0   # Transfer cell from coprocessor register
                      # file to output buffer 0.
add $1, $1, 56        # Update base address.
sub $2, $2, 1         # Decrement cell counter.
br_not_zero $2, start  # Continue if there are more cells to
                      # transmit.

end:

Again, the code above does not show the saving of registers necessary upon en-
tering the subroutine. There might also be other strategies employed when a host
finds that there is no room in an output buffer—the host could abort, or try another
output buffer—but only the simplest one, i.e. waiting, is shown.
Chapter 5

The Coprocessor Host Interface Chip

*If we have our own “why” of life,*
*We shall get along with almost any “how”.*

Nietzsche.

We described in Section 4.3 our architecture of the coprocessor style of host interfacing. We now present an implementation of that architecture. The Coprocessor Host Interface Chip provides an interfacing mechanism between the MIPS R3000 processor and the VuNet. The chip thus presents a route for network cells between the network and the workstation register/memory system. Parts of the design of the chip came from Bellcore, which also arranged for the fabrication of this chip.

This chapter describes the coprocessor chip at a hardware level. Section 5.1 describes the functionality of the chip. Section 5.2 details the external interfaces of the chip, and describes the various signals that need to be provided to the chip. A description of the various functional locks in the hardware design is presented in Section 5.3. Some issues that arise when programming the R3000 with this coprocessor are presented in Section 5.4.
5.1 Design Overview

In this section, we describe the design of the Coprocessor Host Interface Chip. We first give a functional overview of the design, which highlights the various modules in the design, and explains their use. We then briefly describe the pipelining which is a characteristic of the implementation of this chip.

5.1.1 Functionality

In basic terms, the Coprocessor Host Interface chip acts upon four types of coprocessor instructions that it receives from the MIPS R3000 processor system. These instructions can tell the chip to transfer words between the coprocessor's internal registers and either the general registers or the main memory of the workstation. These and other instructions are listed in Table 5.1.

The simplified block diagram (Figure 5-1) illustrates the various modules in the coprocessor. The chip consists of cell buffers, a register file, an instruction decoder, control registers, memory drivers and timing logic. The cell buffers and register file were designed for another project: the ATM Cell Processor [26]. The same designs were appropriate for this project and were reused here.

In this chip, a cell buffer is a FIFO of length 4 cells. Thus, a cell buffer can queue upto 4 cells, and the first cell to enter the buffer will be the first to exit it. These buffers can be set to be input or output in direction. From the network side, a cell can exit or enter in bit-serial or byte-parallel fashion. The details of this mechanism are explained in Section 5.2.2.

The technical details of the various other blocks are laid out in Section 5.3. However, to get a feeling for the functionality of some of these blocks, we can take a brief look at the sequence of actions that must occur during the reception or transmission of data. These procedures mirror the architectural descriptions given earlier.

To send a cell into the network, the CPU needs to fill up the coprocessor's register file. It does this by issuing instructions that transfer data to the coprocessor registers from the R3000's general registers (moves), or from main memory (loads), depending
upon where in the host the network data resides at that time. These transfers take place in 32-bit word chunks, and the data flows from the CPU System Bus, through the Data Bus Drivers and into the Register File. The host CPU needs to determine whether there is room in an output buffer. This it does by reading the Status Register, or by looking at the conditional signal, CpCond (see Table 5.2). Next, the CPU issues an instruction, transmit, that transfers the entire cell from the coprocessor’s register file to an output cell buffer on the coprocessor. This 512-bit transfer takes place in one cycle. Then, it is up to the network to clock the data out from the cell buffer.

In the receiving direction, a cell is first clocked into an input cell buffer by the network. The CPU can learn of the arrival of the network data either by polling the Status Register, or by enabling the Interrupt line (see Table 5.2). The host processor then reacts by issuing a single instruction receive that transfers the entire cell from the input cell buffer to the coprocessor register file. As in the transmit direction, this 512-bit transfer takes place in one cycle. Finally, the CPU transfers that cell data from the coprocessor register file to the CPU general purpose registers (with move
instructions) or to an area in main memory (with store instructions), depending on where the host wants the data to be placed. These transfers take place in 32-bit chunks, and the data flows from the coprocessor registers via the Data Bus Drivers and onto the CPU System Buses.

The coprocessor host interface chip responds to the coprocessor instructions that are defined in the MIPS R3000 instruction set. A list of instructions supported by the coprocessor chip is given in Table 5.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Mnemonic</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>MTC</td>
<td>Move Word from CPU Reg to Cop Reg</td>
</tr>
<tr>
<td></td>
<td>MFC</td>
<td>Move Word from Cop Reg to CPU Reg</td>
</tr>
<tr>
<td></td>
<td>CTC</td>
<td>Move Word from CPU Reg to Cop Control Reg</td>
</tr>
<tr>
<td></td>
<td>CFC</td>
<td>Move Word from Cop Control Reg to CPU Reg</td>
</tr>
<tr>
<td>Load/Store</td>
<td>LWC</td>
<td>Load Word from Memory to Cop Reg</td>
</tr>
<tr>
<td></td>
<td>SWC</td>
<td>Store Word from Cop Reg to Memory</td>
</tr>
<tr>
<td>Coprocessor</td>
<td>TRA</td>
<td>Transmit Cell from Cop Reg File to Cell Buffer</td>
</tr>
<tr>
<td></td>
<td>REC</td>
<td>Receive Cell from Cell Buffer to Cop Reg File</td>
</tr>
<tr>
<td></td>
<td>ENA</td>
<td>Enable Interrupts in Coprocessor</td>
</tr>
<tr>
<td></td>
<td>DIS</td>
<td>Disable Interrupts in Coprocessor</td>
</tr>
<tr>
<td>Branch</td>
<td>BCT</td>
<td>Branch on Coprocessor True</td>
</tr>
<tr>
<td></td>
<td>BCF</td>
<td>Branch on Coprocessor False</td>
</tr>
</tbody>
</table>

Table 5.1: Coprocessor Interface Instructions

A word referred to in the table is 32 bits. The coprocessor is passive during the Branch instructions, in that there is no change of state in any of its components. The CPU simply looks at the CpCond line which is set by the coprocessor as appropriate and indicates whether the cell buffer FIFOs are full or empty.

Another component of the design which we can see in Fig 5-1 is the control registers. There are two such control registers in this design. One is called the Cell Buffer Control Register, and by writing to it, the host can control various characteristics of the cell buffers, like their direction (input or output). The other is the Coprocessor Control Register which controls some other characteristics of the coprocessor chip, like the interrupts-enabling. Details of these registers are given in Section 5.3.
5.1.2 Implementation: Pipelining

The coprocessor chip is tightly coupled with the MIPS R3000 CPU and hence follows the same pipeline stages as the R3000. Basically, the R3000 pipeline stages are: Instruction Fetch, Read, ALU, Mem, and Write Back.

As we can see in Figure 5-2, the instruction decoder unit of the coprocessor forms a five-stage pipeline, and parallels the pipelining of the MIPS R3000. During the first stage, the instruction is read into the coprocessor. In the next stage, the instruction is decoded and the appropriate control signals are generated. In the third stage, the bus drivers in the coprocessor are set up for outgoing (from coprocessor to CPU) data transfers. In the fourth stage, the pads are driven, either by the coprocessor chip, or by the CPU System Bus. In the fifth stage, incoming data transfers and internal data transfers (between cell buffer FIFOs and coprocessor register file) take place.

Thus, the coprocessor communicates with the host CPU in exactly the same manner as a floating point coprocessor would, driving data or accepting data from the CPU System Bus at the appropriate stage in the pipeline. And, as mentioned in Section 4.3, the coprocessor does not stall the host under any circumstance (unlike the floating point unit, which has been given that capability within the R3000 architecture). The reasons for excluding stalling capabilities were pragmatic: there are no "long" operations that the coprocessor has to deal with (all can occur within a clock cycle), and the design became simpler when this functionality was dropped.
5.1.3 Design History

The initial design work for the decoder module was done by Dave Martin, then a graduate student in our group, who wrote some functional descriptions for this module, and started the initial logic-level schematics.

The register file, and the cell buffers were borrowed from the ATM cell processor, designed by Jason Hickey, of Bellcore. This was an interesting experiment, in which silicon from another design was lifted and adapted to fit our design.

5.2 External Interfaces

This section describes the external interfaces through which the Coprocessor Host Interface Chip communicates. There are two such interfaces, one with the MIPS R3000, and one with the network. The chip interacts with the R3000 processor according to the standard coprocessor specification described in [31]. On the network side, the interface is simple, and governed by external clocks (Figure 5-3).

![Diagram of External Interface Signals]

Figure 5-3: External Interface Signals

Figure 5-3 shows that the width of the data bus on the CPU side of the chip is 32 bits. On the network side, the coprocessor can be connected to up to four ports of one or more network switches, as there are four cell buffers on the chip. Since the direction of the cell buffers is programmable, there can be any combination of receive
and transmit ports attached to the host via this coprocessor.

In the following descriptions, the directions “input” and “output” are relative to the coprocessor. Also the “receive” and “transmit” directions are taken relative to the host, i.e. receive means that data flows from the network to the host.

5.2.1 Coprocessor-R3000 Interface

Table 5.2 describes the various signals that pass between the coprocessor and the R3000 system. The CPU System Bus of the R3000 connects to the Data Bus Driver unit of the chip, and the various clock signals coming from the R3000 go to the Timing Logic unit of the chip.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CpSync</td>
<td>I</td>
<td>Timing signal sent out by the R3000 processor to synchronize data transfers between the processor and the coprocessors.</td>
</tr>
<tr>
<td>Run</td>
<td>I</td>
<td>Indicates whether R3000 is in a run or stall cycle. The coprocessor can advance its pipeline as long as Run* is asserted.</td>
</tr>
<tr>
<td>Exception</td>
<td>I</td>
<td>Indicates additional information about an exception. It can be asserted during either phase 1 or 2 in run or stall cycles.</td>
</tr>
<tr>
<td>CpBusy</td>
<td>O</td>
<td>Indicates start/endpoint of a coprocessor busy stall. This signal will not be used for this coprocessor, since the chip will be synchronous with the R3000, and no extra cycles will be needed to transfer data in and out of the chip.</td>
</tr>
<tr>
<td>CpCond</td>
<td>O</td>
<td>This is the condition line to the R3000. It will be asserted to indicate the condition of the cell buffers on the chip.</td>
</tr>
<tr>
<td>Interrupt</td>
<td>O</td>
<td>This is the interrupt line to the R3000, and will be asserted for the same conditions as the CpCond line if the interrupts are enabled on the chip.</td>
</tr>
<tr>
<td>Data_Bus[31:0]</td>
<td>I/O</td>
<td>Bus for the transfer of Instruction and Data to or from the R3000 registers, the memory system and the coprocessors.</td>
</tr>
<tr>
<td>Parity[3:0]</td>
<td>O</td>
<td>These four bits generate the parity of the four bytes of the data bus. These bits are generated by the coprocessor during store instructions.</td>
</tr>
</tbody>
</table>

Table 5.2: Coprocessor Interface Signals

The R3000 distinguishes two types of cycles: run cycles and stall cycles. While
the coprocessor does not stall the processor, it must be able to react if another unit causes a stall. During run cycles, the coprocessor simply needs to advance its own pipeline appropriately. When the R3000 deasserts \( \overline{\text{Run}} \), it indicates a processor stall and the coprocessor will disregard the last instruction/data pair presented to it. The coprocessor remains passive as long as the R3000 is in stall cycles, (i.e. the data in the pipeline registers is not advanced to the next pipeline stage). A stall can be initiated by other units in the host workstation, like the floating point unit, or the system coprocessor.

**CPU System Bus**

The *CPU System Bus* to which the coprocessor connects is a 32-bit bidirectional bus, in which instructions and data are time-multiplexed. Thus, on each clock cycle, an Instruction-Data pair (ID pair) is put on the bus. The instruction appears in the first half of the clock cycle, and the data, in the second half. When there is neither an instruction nor data on the bus, the bus goes into the high-impedance state (tristate). The ID pairs follow the pipeline stages mentioned earlier in Section 5.1.2. On a DECstation 5000/240, this bus runs at 40Mhz, which puts the maximum possible throughput on this bus at 1.28Gbps. We can see how this bus is multiplexed in Figure 5-4. More detailed diagrams can be found in [28].

### 5.2.2 Coprocessor-Network Interface

The coprocessor interfaces to the network via its cell buffer FIFOs, which are described in more detail in Section 5.3.2. Depending on how we configure the cell buffers, data flows between them and the network either bit-serially or in byte-parallel fashion. The signals involved in this data transfer between the coprocessor cell buffers and the network are shown in Table 5.3.

The operation of this interface is as follows. If a cell buffer is an input buffer, then the data, data clocks (Clkp, Clkm) and cell boundary clock (Stin) will be provided by the network. If a cell buffer is an output buffer, then the data clocks and the cell boundary clock will be provided by the network, and the data will be provided by the
<table>
<thead>
<tr>
<th>Cycle</th>
<th>Run</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr</td>
<td>CpStore</td>
<td>CpLoad</td>
</tr>
<tr>
<td>Phase</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

CpsSync

Data

Figure 5-4: R3000 CPU System Bus

<table>
<thead>
<tr>
<th>Signal</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLKP, CLKM</td>
<td>I</td>
<td>Complementary data clock. Data is latched on the rising edge of CLKP.</td>
</tr>
<tr>
<td>STIN</td>
<td>I</td>
<td>Strobe input. Falls on first bit/byte of cell to signal beginning of new cell.</td>
</tr>
<tr>
<td>STOUT</td>
<td>O</td>
<td>Strobe output. Delayed version of STIN.</td>
</tr>
<tr>
<td>DATA[8]</td>
<td>I/O</td>
<td>Input/output data from network to coprocessor.</td>
</tr>
<tr>
<td>IOFLAG</td>
<td>O</td>
<td>Flag indicates status of cell buffer. The flag is raised if an input cell buffer is full, or an output cell buffer is empty.</td>
</tr>
</tbody>
</table>

Table 5.3: Network Interface Signals

coprocessor cell buffer. Figure 5-5 illustrates the timing of a receive operation (from the network to the host). More details on the cell buffers can be found in [26].

The external flags (IOFLAG) shown in Table 5.3 indicate to the network the status of each cell buffer. If the IOFLAG of a cell buffer is asserted, it means that that particular cell buffer is empty, if it is an output buffer, or full, if it is an input buffer.

If the cell buffers are clocked serially, the minimum clock cycle allowed is 5ns, and if the mode is byte-parallel, the minimum clock cycle is 13ns. This gives a maximum possible throughput of 200Mbps for the serial mode, and 615Mbps for the
parallel mode. Note that these numbers just provide an upper limit on the possible throughput.

The list of signals in Table 5.2 is given for one cell buffer only. However, since the coprocessor chip has four such cell buffers, it consequently has four such ports, and hence four sets of the signals defined above.

5.3 Hardware Details

We saw in Figure 5-1 that the coprocessor consists of the following modules: a decoder unit, 4 cell buffers, a register file, 2 control registers, a status register, a timing logic, and data bus drivers. A more detailed description of each of these modules follows.

5.3.1 Decoder Unit

The Decoder Unit is responsible for decoding the instruction coming from the CPU and generating the appropriate signals for the other parts of the chip. The necessary pipelining in the coprocessor is implemented in the Decoder Unit, by way of registers, as seen in Figure 5-6.

In the first pipeline stage, the instruction word from the R3000 data bus flows
through the data bus drivers and is latched onto the first pipeline register within the decoder unit. In the next stage, the instruction word reaches the control logic within the decoder unit, and is decoded. Various control signals are generated. In the third stage, data is read from the register file and is latched in the output registers of the data bus driver unit (if so required by the instruction). In the fourth stage, the memory drivers output the data onto the bus. In the fifth stage, data can flow between the register file and the cell buffers, or from the memory drivers to the register file, depending on the instruction.

Since these pipeline stages are tightly coupled with the MIPS R3000 stages it is important to synchronize the coprocessor with the CPU. The CPU sends out a dummy clock signal, \( CpSync \), which is used by the Timing Logic unit of the coprocessor for synchronization (see Section 5.2.2 for details of the R3000-coprocessor interface).

### 5.3.2 Cell Buffers

As shown in Figure 5-7 there are four cell buffers in the coprocessor chip. Each cell buffer is FIFO which can contain four 64-byte cells. These cell buffers act as the external ports to the network for the coprocessor. As we have seen, they can
be clocked by the network at a rate convenient to the network. On the internal side, the cell buffers have a 512 bit interface to the coprocessor register file, and this interface is clocked at the CPU System Bus rate (40MHz, for a DECstation 5000/240). Thus these cell buffers provide the temporal decoupling that is necessary between the workstation and the network.

![Cell Buffers Diagram](image)

**Figure 5-7: Cell Buffers**

The cell buffers can be programmed to be input or output buffers, or to have a bit-serial or byte-wide interface. Further, each cell buffer can be programmed with a cell offset, which determines where within the 64 byte space a cell will be placed (this is explained in more detail in Section 5.3.4). This feature is useful in that the chip does not assume a particular cell length, hence allowing the coprocessor to be used in different networks which use fixed-size cells with lengths ranging from 1 to 64 bytes. These features can be changed by writing the desired value in the Cell Buffer Control Register.

We have already seen (in Section 5.2.2) how the IOFLAGs indicate the state of the cell buffers to the network. The cell buffers also indicate their state to the other parts of the coprocessor itself, and from thence to the host processor, via internal FLAGs.

There are four FLAGs in the chip, one for each buffer. If a cell buffer is an input buffer, then FLAG=HIGH means that the buffer is empty. If a cell buffer is an output buffer, then FLAG=LOW means that the buffer is full. These FLAGs are fed to the
Status Register, from there they can be read. They also appear as test pins on the chip, and are used for debugging purposes. The FLAG signals cause the Interrupt and CcCond lines to the R3000 to be asserted, provided the interrupts and conditionals are properly enabled. Details of this enabling scheme are outlined in Section 5.3.4.

In the implementation described above, the input and output buffer sizes are 4 cells long. However, within the ViewStation system each such cell buffer is directly connected to a switch on the network, and the receive and transmit buffers of a switch are much larger (256 and 64 cells, respectively, for a VuNet switch). So the effective buffer sizes of the host interface include the buffering provided in a port of the switch.

5.3.3 Register File

The register file consists of 64 32-bit registers. It has one 32-bit read port, one 32-bit write port and one 512-bit bidirectional port. The word-wide ports are used for transferring data to (and from) the R3000 data bus, via the data bus driver unit of the coprocessor. The cell-wide port (512 bits, or 64 bytes) is used for data transfer between the register file and a cell buffer (see Figure 5-8).

![Figure 5-8: Register File](image)

The register file can be thought of as consisting of four 64-byte (cell-sized) registers. Conceptually, the register file is then divided into two halves, one for receiving cells from the network, and one for sending cells to the network. Thus two cell-sized registers are available for each direction. Since the instruction set only supports a
5-bit register address, and this register file needs to be addressed with 6 bits, it was necessary to add a bit, called \textit{CBAADDR.S}, which resides in the Coprocessor Control Register, and represents the high bit in the register address. The \textit{CBAADDR.S} bit determines which “half” of the register file the user intends to use for the SEND direction, and which for the RECEIVE direction.

If \textit{CBAADDR.S}=HIGH, it means that the lower half of the register file (words 0:31) will be used for the send direction, and the upper half (words 32:63) for the receive direction. So now, if the user wants to read, say, word 34 from the register file, he/she need only specify the lower bits of the address, (i.e. 00010 = 2). The high bit will be determined by the chip as follows: since the user has issued a read instruction (move/store from coprocessor register), it indicates that the direction of transfer must be RECEIVE (since normally a user reads data received from the network, and writes data going to the network), and thus the required bit is HIGH, as determined by \textit{CBAADDR.S}.

The addressing scheme works similarly in the interface of the register file to the cell buffers. In this interface, one cell gets transferred between the cell buffer and one of the four cell-sized registers that constitute the register file. Since there are four such conceptual registers, we would need 2 bits to specify the address, however, it is the \textit{CBAADDR.S} bit which represents the high bit in the address.

Note that it is conceivable that a user might want to read a word from a location in the register file that is intended for the SEND direction (to perform diagnostics, for example). In that case the user would have to set the \textit{CBAADDR.S} bit appropriately and access the required word.

\textbf{5.3.4 Control Registers}

There are two read/write control registers and one read only status register which are described in this section.
Cell Buffer Control Register

This register is readable and writeable. It can be programmed to control the configuration of the cell buffers. The characteristics which are programmable are: the direction of a cell buffer (whether input or output), the external width of a cell buffer on the network side (whether bit-serial or byte-wide) and the offset. Figure 5-9 shows the placement of the relevant bits in this register.

![Cell Buffer Control Register Diagram]

- Ser[3:0]: Bits 3:0 set port configuration of each of the cell buffers[3:0]. If bit=LOW, that buffer is bit-serial, else byte-wide.
- Dir[3:0]: Bits 7:4 set direction of each cell buffer. If bit is LOW, input buffer, else output buffer.
- Offs[3:0][5:0]: Bits 31:8 determine an offset of a cell within the 64-byte cell-space of a buffer.

Figure 5-9: Cell Buffer Control Register

The byte-offset mechanism is depicted in Figure 5-10. We have seen how a cell buffer is a FIFO of cells, and it has also been mentioned that the width of a cell is programmable. This width is specified by writing an appropriate offset value for a particular cell buffer into the Cell Buffer Control Register. To deal with 56-byte cells, we would write an offset value of 8, for 53-byte cells, the offset would be 11, and so on. The design allows for different size cells in different cell buffers.

Coprocessor Control Register

This control register is also readable and writeable. It contains the number of the coprocessor chip. The MIPS supports up to 4 coprocessors, but COP0 is the system control coprocessor, and COP1 is usually the Floating Point coprocessor. This chip resets to being COP3, but this value can be changed by writing the appropriate bits in
the register. It also contains the \texttt{CBADDR.S} bit, which is the high bit in the register address. See Figure 5-11 for the location of the relevant bits in the Coprocessor Control Register.

\textbf{Interrupt and Conditional Enabling:} There is one bit for each cell buffer that determines whether that cell buffer can affect the \texttt{Interrupt} and also the \texttt{CpCond} line. The \texttt{CpCond} line will always be set according to the masking programmed into this register. The \texttt{Interrupt} line, on the other hand, will be set according to the masking specified only if the overall interrupt capability of the chip has been enabled. This overall interrupt enabling is controlled by writing the \texttt{ENINT} bit appropriately. If this bit is set, it means that the interrupts are enabled. The \texttt{CpCond} line is used as a condition by the Coprocessor Branch instructions, which have been explained in previous sections.

\textbf{Status Register}

The \textit{Status Register} is a read-only register, and contains flag bits which show which cell buffer is full or empty. If a cell buffer is an input buffer, and the flag is set, it indicates that the buffer is empty. If the cell buffer is an output buffer, and the flag is set, it indicates that the buffer is full. The register also contains bits which indicate whether the interrupt line (\texttt{Interrupt}) or the condition line (\texttt{CpCond}) to the R3000 is set or not. The bits which are used are shown in Figure 5-12.
• CPMASK[3:0]: Bits 3:0 determine which cell buffer can set the CpCond line to the R3000. If bit=HIGH, that buffer can set line.
• INTMASK[3:0]: Bits 7:4 determine which cell buffer can set the Int line to the R3000. If bit=HIGH, that buffer can set the line.
• CBADDR_S: Bit 28 determines the high bit in the register address, as well as the cell buffer address.
• WHOAMI: Bits 30:29 identify the number of the coprocessor. This bit can be written, but upon reset the value is set to #3.
• ENINT: Bit 31 determines whether the coprocessor can signal an interrupt to the R3000.

Figure 5-11: Coprocessor Control Register

5.3.5 Bus Drivers

This module serves as the interface between the coprocessor and the system bus of the MIPS R3000 processor system. It reads instructions and data from the bus, and outputs data onto the bus, at the appropriate time. The timing signals used come from the Timing Generator unit.

This unit is designed to latch onto the instruction and data at different phases of a clock cycle. Thus there are two sets of registers for this purpose, which are fed by different clocks.

5.3.6 Timing Unit

This module is responsible for buffering the CPSYNC clock signal supplied by the CPU. This clock is specially supplied for the purpose of coprocessor synchronization. The Timing unit circuitry also handles the logic necessary to detect a stall cycle in the system bus, and to generate the appropriate signal to the rest of the coprocessor
- FLAG[3:0]: Bits 3:0 determine the state of fullness of the corresponding cell buffer. If an input cell buffer is empty, or an output cell buffer is full, then the flag will be set=HIGH.
- CPCOND: Bit 30 indicates whether the CpCond line to the R3000 is set or not.
- INT: Bit 31 indicates whether an interrupt situation exists that would set the Int line to the R3000. Note that the Int line would only be set if interrupts are enabled.

Figure 5-12: Status Register

5.4 Programming Issues

There are a few issues that need to be kept in mind while programming the R3000 with the coprocessor chip.

Delayed Instruction Slot

The load, jump and branch instructions of the MIPS R3000 have a delay of one clock cycle, so any assembly programming must take this into account.

Enabling Coprocessor Usability on MIPS

The Status Register of the MIPS system control coprocessor (CP0) has certain bits (CuX) that indicate whether a particular coprocessor is usable or not. In order to be able to operate this coprocessor, the Cu3 bit (bit 31) should be set (since this is coprocessor #3). If it is not set and an attempt is made to access coprocessor #3, then a Coprocessor Unusable Exception (CpU) results. When a CpU Exception is
taken, the Coprocessor Error (CE) bits of the CP0 *Cause Register* (bits 29:28) will indicate the number of the coprocessor referenced, and also the ExcCode bits of this register (5:2) will have the value "11", indicating that the exception was a CpU.

**Interrupting the MIPS**

The coprocessor can interrupt the MIPS R3000 by means of an External Interrupt. There are 5 external interrupts allowed on the MIPS, but the coprocessor will just use one of them\(^1\). The interrupts can be individually enabled by setting the IntMask bits (15:10) on the CP0 *Status Register*. When the coprocessor performs an External Interrupt, the ExcCode bits (5:2) on the CP0 *Cause Register* will have the value "0", which signifies that the exception was caused by an External Interrupt. The IP bits (15:10) of the CP0 *Cause Register* indicate the external interrupts that are pending.

### 5.5 Status

The chip was manufactured in a 1.2 micron process by VTI. The die size is 7.6 mm x 5.2 mm, and is packaged in a 223-lead high-performance ceramic 18x18 pin grid array. There are about 118 K gates on this chip.

Through testing of the chip, we have been able to read and write the control registers and the register file, thereby verifying the functionality of the decoder unit and the data bus drivers as well. We have also been able to receive data into the cell buffers, transfer it to the register file, and then read it out of the register file, from the processor side. So far, the processor bus has been emulated with a logic analyzer.

So far, there appear to be some bit errors in the transmit side of the cell buffers, as certain random bits get reset. Further, the stall mechanism seems to cause the chip to not respond to later instructions. These problems are currently being further analyzed.

---

\(^1\)Alternatively, in the case of a DECstation, the coprocessor could be made to interrupt the processor by using the interrupt signal of the coprocessor as an interrupt on the Turbochannel.
Chapter 6

Performance

*Nothing is good or bad but by comparison.*

Thomas Fuller.

We have presented an architecture for a fast host interface, and this chapter now discusses the performance analysis that was done to determine the speedup that this design offers. We compared this interface with a programmed I/O Turbochannel-based host interface board, the Vupboard, which was also designed in our group. The coprocessor measurements are based on the floating point coprocessor, since the coprocessor is not fully tested and integrated. We also compare these measurements against a DMA style of host interface.

In the first two sections of this chapter, we shall describe the hardware and software setup of the measuring environment. In Section 6.3, we shall present results from our performance experiments. An analysis of these results is then given in Section 6.5.

6.1 Hardware Environment

In order to better understand the performance measurements, we should first look at some of the aspects of the setup that was used. The hardware platform on which the measurements were done is first discussed. A brief look is provided at the Vupboard, the host interface with which this design is compared.
6.1.1 Measurement Platform

To measure the throughput attainable by the coprocessor host interface, we performed timing experiments on a floating point coprocessor. We have seen earlier (Section 4.3.1) that the user’s view of the coprocessor interface essentially consists of coprocessor general purpose registers and coprocessor control registers. These are available on a floating point coprocessor as well, and since the timing of all coprocessors (whether floating point or network interface) with respect to the main processor in the R3000 architecture is the same, the results we obtained can be expected with the network interface cell coprocessor also.

The workstations used were a DECstation 5000/200 and a DECstation 5000/240. For the alternative host interface, the programmed I/O Vupboard was used, and is described briefly next.

6.1.2 The Vupboard

VuNet is the network around which the Viewstation Project is built, and this was the platform used for the experiments. One main focus of the analysis was to compare a host interface in which the network resided close to the host processor, in a coprocessor’s registers, to one in which the network resided further away, on the I/O bus of the workstation. As an example of the latter design, the Vupboard (VuNet Programmed I/O board) was used.

The Vupboard is a host interface board which sits on the Turbochannel. The model for data transfer is programmed I/O, and protocol processing is the responsibility of the host. (The Vupboard was intended as a simple host interface that could be quickly designed to enable our research group to start testing the different hardware and software components of the Viewstation project.) Except for the fact that the coprocessor approach brings the network much closer to the host CPU, the programming environment for the Vupboard and the coprocessor is almost the same. Thus, a comparison of these two host interfaces provides a good idea of how the location of a host interface on a workstation affects the throughput, since the other variables
in the two interfaces (software overheads, mode of data transfer, level of on-board processing) are the same.

6.2 Software Environment

We saw in Chapter 4 a simple inner loop of a receive and transmit routine. The code was simplified to illustrate how the coprocessor interface might be used. We shall now look at the routines of our diagnostic software which were actually used to perform the measurements.

The diagnostic software (Vuptest) used in our experiments, was developed in our group by Chris Lindblad and David Wetherall. This was written essentially to test the Vupboard host interface. It allows us to issue various commands to the Vupboard, like read and write a burst of cells, read status flags, flush and fill input and output buffers. The routines used here are the Receive Burst and the Transmit Burst. As their names suggest, these routines of Vuptest receive and send bursts of cells. The size of the bursts, and the number of bursts are set by the user.

These routines reflect the common case in the Viewstation environment, in which the VCI's of successive cells are the same. In order to test the other models of interfacing, like the coprocessor and the DMA model, we needed to modify the inner data-moving loops in these routines (either in the c-code, or at the assembly level).

We start by explaining the receive direction. The flowchart in Figure 6-1 shows the steps involved in receiving a burst of cells from the Vupboard. For a listing of the code, as well as code for the other models of host interfacing, see Appendix C.

The code is entered with a given number of cells and cycles that it has to receive. It is also given a pointer in main memory where the burst of cells is finally destined for. Our primary concern is with the inner data-moving loop of the receiving process, and hence the mechanisms used for obtaining the number of cells, cycles and the pointer in memory are ignored in this discussion.

The next step, as seen in Figure 6-1 is to perform some checks to ensure the validity of the data. Among other things, the code makes sure that the correct number of
arguments have been provided.

The routine now enters the data-moving loops. The outer loop counts the number of cycles that have to be done (i.e. the number of bursts to perform), while the inner loop counts the number of cells that have to be received (i.e. the size of each burst of cells). This inner loop contains the instructions to move the data from the network into the workstation. In the Vupboard case, this consists of 14 instructions to load the data from the Turbochannel to the general purpose registers, followed by 14 instructions to store this data from the registers into the given area in main memory. These 28 instructions will take more than 28 clock cycles, because of the latencies involved in accessing the Turbochannel and main memory. This inner loop may also contain any cell operations that may need to be performed, like the header byte swap operation.
The transmit routine mirrors the receive routine, except for the innermost loop, in which data is moved from the workstation out into the network. One difference is that the transmit routine has to indicate which port of the switch the data is supposed to be sent out of.

6.3 Throughput Experiments

We now present some results of the performance experiments that we ran, in the environment described above. We shall compare the throughput obtained in the Turbochannel Vupboard interface with the expected throughput of the coprocessor interface. We also briefly look at an alternative: a DMA interface, and contrast the performance with the coprocessor interface.

The throughput experiments were aimed at comparing the bandwidths achievable by different interfaces. Several transmission and reception routines were timed, as mentioned above, and throughputs were determined from these times. Our experimental space has the following four parameters which are under our control.

- **Interface Model.** By changing the instructions that the receive and transmit routines issue, we could study various models with which the processor can talk to the network. Table 6.1 lists the different models we considered. We discuss this experiment variable below.

- **Transfer Direction.** We could observe the differences between the receive and transmit directions of network transfer.

- **Cell Operations.** Two sets of experiments were conducted, one in which the processor performed certain byte operations per cell, and one in which it did not. The operation performed was the byte-swapping of the two header words, and took 18 instructions in the inner loop.

- **Processor Speed.** A DECstation 5000/200 (25MHz) and a DECstation/240
(40MHz) were used for the experiments. We could study the effect of faster processors on network interfacing.

We ran a series of experiments in which each of these variables was changed independently of the other. The results are presented later, in Tables 6.2–6.4. Most of the above parameters are fairly self-explanatory, but the interface models need more explanation.

### 6.3.1 Interface Paradigms

The interface models that we studied are summarized in Table 6.1. The approaches are different in that they make the network appear to reside at different points in the memory hierarchy: in an I/O device, and in registers. In all the cases, the receive data is finally destined for cacheable areas in memory, and similarly transmit data is sent from cacheable memory locations. Accessing uncached memory is possible in an R3000-based machine, like the DECstation, however, this can be done only in kernel-mode. Since application programs are more likely to be in user-mode, we only study the cacheable memory scenarios.

**Network in I/O Device: Programmed I/O.** This approach is labeled Vup in Table 6.1. Data flows between the network and a cacheable area in the host memory via the general purpose registers. To measure the performance of this model, we used a Vupboard plugged into the Turbochannel of the two different DECstations. The code we used is given in Appendix C, and follows the same flow as described earlier with Figure 6-1.

We also considered what the performance might be if this approach were to be used in an environment in which the data did not need to be touched by the processor (VupI). Since the mode of data transfer is programmed I/O, however, the registers still have to be involved. The only difference between Vup and VupI, then, is that in the latter case, the code excludes the header byte swap operations, so that the only

---

1We note here that the “240” machine actually resulted in a worse Turbochannel performance, than the “200”, and we attribute this to the fact that the clock rate of the “200” matches with that of the Turbochannel (25 MHz), thereby giving better results.

92
function of the inner loop was to move data between the network and the memory.

Network in Registers: Coprocessor. There are several approaches we considered, in which the network appeared to reside in the registers of a coprocessor. These are summarized in Table 6.1, so we shall only describe the more interesting models here.

The Cop1 model is one in which data needs to be touched by the processor, and hence the data is brought into the general purpose registers. The operation performed, again, is the header byte swap. As explained before, we emulated our coprocessor architecture by performing our experiments on a floating point coprocessor of the DECstation. We used essentially the same routines as for the Vup, except that the inner loop was hacked at the assembly language level, so that the data transfer operations occurred between memory and coprocessor registers, rather than memory and the I/O bus. Thus, for example, loads from the I/O bus were replaced by register moves from the coprocessor to the CPU.

The Cop6 model reflects an environment in which data does not need to be touched by the processor. In the architecture of the R3000, which is at the core of the DECstation, a direct transfer is possible between the coprocessor register file and memory. Thus, we modified the code from Vup so receive data was stored directly from the coprocessor registers to memory, and transmit data was loaded directly from memory.

The other models are variations of these two extremes. For example, Cop2 reflects an environment in which the processor needs to touch only the header, and perform a header byte swap. The body of the cell is transferred directly between memory and coprocessor registers.

Network in I/O Device: DMA. This approach again places the network in an I/O device. However, this time, the network data can travel directly between memory and the I/O channel. The first case (shown in Table 6.1 as DMA) is one in which data needs to travel to the general purpose registers, to be operated upon by the processor. The following scenario would occur in the receive direction. While the processor is busy with its other functions, the I/O network device would DMA
the data into an area in memory (again, we consider the *cacheable* area in memory). Once the processor learns that the DMA transfer is complete, it starts reading in the data from the locations in memory into which that data had been placed.

We simulated the most optimistic results possible with this approach, thereby providing an upper boundary to the performance possible. Our experiments measure the data transfer rate between memory and the registers. Thus, we are assuming that the DMA transfer has occurred at little interference to the processor, and the rate of this DMA transfer is much higher than the rate at which data can be accessed by the processor. For the Turbochannel, the maximum rate at which data can be transferred is 800 Mbps. However, in Section 6.5 we compare this number with the DMA board designed at Bellcore [14]. The code we used was the same as the Vup case, except that a simulation mode designed into the Vup test environment was used, so that the code accessed areas in memory, rather than the I/O bus. For the sake of completeness, we have also performed the same measurements, but without the header byte swap (*DMA1*).

We have seen, in Section 6.2, how the receive and transmit routines typically consist of some decision-making sections, followed by the actual data-moving loop. In order to identify the peak throughput achieved in these tight loops, the following methodology was used. First the observed times were recorded for the different receive and transmit routines. Then timings for the same routines were measured again, but with the data-moving instructions commented out (these are referred to as "bare" routines in Table 6.2). From these numbers, the time spent in the tight loop was determined, and hence the peak bandwidth achieved was calculated.

A sample listing of the times that were measured is given in Table 6.2. Since the granularity of the workstation’s measuring clock is fairly coarse (100ms), large bursts were used. The times are normalized per cell, so that the figures we see represent the time taken to receive or transmit one cell.

One important point of note here is the size of the cell burst used. We wanted to avoid the scenario in which successive accesses to memory would find the data in cache. In a real environment, a large burst of data would consist of data that is "new"
<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vup</td>
<td>Programmed I/O Vupboard interface. For receive routines, this consists of loads from Turbochannel to processor registers, followed by stores into memory. For transmit, it consists of loads from memory to the processor registers, followed by stores to the Turbochannel.</td>
</tr>
<tr>
<td>Vup1</td>
<td>Same as Vup, except with no header byte swap.</td>
</tr>
<tr>
<td>Cop1</td>
<td>Similar to VUP, using the coprocessor approach.</td>
</tr>
<tr>
<td>Cop2</td>
<td>For receive, move header from coprocessor to processor registers. Then store body directly from coprocessor register file to memory. For transmit, load header into CPU registers, and load body directly from memory to coprocessor registers.</td>
</tr>
<tr>
<td>Cop3</td>
<td>In the receive direction, store entire cell directly into memory. For transmit, load entire cell directly from main memory to coprocessor registers.</td>
</tr>
<tr>
<td>Cop4</td>
<td>Same as Cop1, except with no header byte swap.</td>
</tr>
<tr>
<td>Cop5</td>
<td>Same as Cop2, except with no header byte swap.</td>
</tr>
<tr>
<td>Cop6</td>
<td>Same as Cop3, except with no header byte swap.</td>
</tr>
<tr>
<td>DMA</td>
<td>In the receive mode, load data that has already been stored into main memory by DMA. In the transmit mode, store data into main memory to be read by the network through DMA.</td>
</tr>
<tr>
<td>DMA1</td>
<td>Same as DMA, except with no header byte swap.</td>
</tr>
</tbody>
</table>

Table 6.1: Interface models

to the processor, and hence would not be found in the cache (except for the fact that reading one word from memory would fill up a cache line's worth of network data in the cache). To recreate such an environment, we used a burst size of more than twice the size of the cache (which is a direct-mapped 16K word cache, in the DECstation [17]). This ensures that subsequent accesses to a particular address will always result in a cache miss.

We can see from the measurements given in Table 6.2, that more than 90% of the time is spent in the execution of the tight data-moving loop. The peak throughput of this loop is calculated using the total number of bytes moved (cell sizes are 56 bytes), divided by the time spent in this loop (for an expanded version of Table 6.2, listing all the times measured, see Appendix B). These throughput figures are presented in Tables 6.3 and 6.4.
<table>
<thead>
<tr>
<th>Burst Size</th>
<th>Receive on 5000/240 (time in $\mu$s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cells</td>
<td>Cycles</td>
</tr>
<tr>
<td>4000</td>
<td>1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Burst Size</th>
<th>Transmit on 5000/240 (time in $\mu$s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cells</td>
<td>Cycles</td>
</tr>
<tr>
<td>4000</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 6.2: Sample time measurements, normalized per cell

<table>
<thead>
<tr>
<th>Direction</th>
<th>Vup</th>
<th>Cop1</th>
<th>Cop2</th>
<th>Cop3</th>
<th>DMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECstation 5000/200 (in Mbps)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>57.2</td>
<td>210.3</td>
<td>292.4</td>
<td>304.4</td>
<td>176.2</td>
</tr>
<tr>
<td>Transmit</td>
<td>79.7</td>
<td>104.6</td>
<td>120.7</td>
<td>124.9</td>
<td>96.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DECstation 5000/240 (in Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive</td>
</tr>
<tr>
<td>Transmit</td>
</tr>
</tbody>
</table>

Table 6.3: Peak Throughput (with header byte swap)

## 5.4 Validation

Before analyzing these results, we first compare them with our expected throughput index numbers, which we calculated in Section 3.4.2. These indices were determined with no byte operations on the cell. As explained before, the Vup1, Cop4, Cop5, Cop6, and DMA1 approaches are modeled by routines that do not have the 18 header byte swap routines, i.e. no byte operations on the cell. Thus, in validating our results, we need to look at Table 6.4. Our observations can be summarized in Table 6.5, and

<table>
<thead>
<tr>
<th>Direction</th>
<th>Vup1</th>
<th>Cop4</th>
<th>Cop5</th>
<th>Cop6</th>
<th>DMA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECstation 5000/200 (in Mbps)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>63.1</td>
<td>337.1</td>
<td>512.6</td>
<td>535.8</td>
<td>239.3</td>
</tr>
<tr>
<td>Transmit</td>
<td>91.4</td>
<td>127.6</td>
<td>150.3</td>
<td>152.3</td>
<td>116.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DECstation 5000/240 (in Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive</td>
</tr>
<tr>
<td>Transmit</td>
</tr>
</tbody>
</table>

Table 6.4: Peak Throughput (without header byte swap)
<table>
<thead>
<tr>
<th>Direction</th>
<th>Predicted</th>
<th>Observed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Path Rate</td>
<td>Model Rate</td>
</tr>
<tr>
<td>Receive</td>
<td>IO1 89</td>
<td>Vup1 40.9</td>
</tr>
<tr>
<td></td>
<td>IO3 294</td>
<td>DMA1 290.7</td>
</tr>
<tr>
<td></td>
<td>REG1 486</td>
<td>Cop4 379.7</td>
</tr>
<tr>
<td>Transmit</td>
<td>IO1 166</td>
<td>Vup1 111.9</td>
</tr>
<tr>
<td></td>
<td>IO3 768</td>
<td>DMA1 196.0</td>
</tr>
<tr>
<td></td>
<td>REG1 358</td>
<td>Cop4 259.7</td>
</tr>
</tbody>
</table>

Table 6.5: Predicted vs. Observed Throughputs (in Mbps)

we discuss these observations below.

- The predictions made by the indices with regard to the comparative performance of the various approaches seem fairly accurate. Those numbers suggested that the I/O device (path IO1) approach would yield the lowest throughputs. The I/O device (path IO3) would be better, but the best results would be obtained by the Registers (path REG1) approach. These three paths are represented by the models \textit{Vup1}, \textit{DMA1}, and \textit{Cop4}, respectively. And we observe that the throughputs increase as expected.

- The index numbers also reflect the difference in throughput between the receive and the transmit directions for each approach. These are also fairly indicative of the observed results. For example, in the \textit{Vup1} case, we observed the transmit direction is faster than the receive direction, however in the \textit{Cop4}, the reverse was true. This is indeed what the index numbers had predicted.

- As expected, the index numbers predicted a throughput higher than observed. In the programmed I/O path, the receive and transmit throughputs suggested by the index numbers are 89 Mbps and 166 Mbps. This can largely be accounted for by the fact that we assumed that Turbochannel writes and reads took 3 and 8 cycles respectively. Some logic analyzer experiments conducted by Chris Lindblad and Dave Tennenhouse indicate that the read performance on the Turbochannel is 14 Turbo-cycles. This changes our predicted value in the receive
direction to 48 Mbps, which is much closer to the observed value. Unfortunately, no experimental values are available for Turbochannel write direction.

- In the Cop case, the numbers predicted were 486 Mbps and 358 Mbps respectively. These numbers are higher than the observed ones because of the latencies involved in the memory system. For example, in the receive case, we did not take into account the write latency, which would result in a loss of performance. There are other factors which degrade the performance in the transmit direction. The read latency might be more than the 8 cycles we have used in our calculations. Or, a cache entry might be invalidated by some write to a location, causing the network data which had been brought in as part of a line to be flushed out. These are some of the arguments which can also explain the lower DMA throughput.

- The observed throughputs in the DMA case are 290 Mbps and 196 Mbps, in the receive and transmit directions, respectively. These values take into account the entire path of data, all the way from the network DMA device, to the correct location in memory. The Bellcore board [13] runs at 480 Mbps and 384 Mbps. However, these figures are for the direct path between the DMA device and memory, and do not take into account the copying of this data into some final location in memory. Our predicted throughput figures (294 Mbps and 268 Mbps), are limited by the memory sub-system, rather than the throughput between the memory and DMA device ($T_{MI}, T_{IM}$). Thus these provide an upper bound on the overall performance achievable, if a DMA board, such as the Bellcore board, were used. Even if the DMA board could transfer at higher rates into memory, the total throughput would be bound by the performance of the memory system.

We see that the connection model outlined in Chapter 3 succeeds in providing us with an idea of which attachment point might be suited for a DECstation 5000/240. The index numbers that we determine from this connection model also correctly predict which direction of transfer would be of higher throughput. Finally, we see
that, as explained before, these numbers provide us with an estimate of the upper bound of the throughput.

6.5 Analysis

In this section, we analyze the results of Tables 6.3 and 6.4. We saw in Section 6.3 how four variables (the interface model, the transfer direction, the per-cell operations and the processor speed) were changed, and the corresponding measurements taken. We shall now discuss the effects we observed due to each of these variables.

6.5.1 Interface Model

We can make several observations with regard to the effect of the host interface model on the throughput. The first two points are summarized in Figure 6-2.

![Figure 6-2: Throughput Comparison of Different Interface Models](image)

- A tightly coupled host interface clearly increases the throughput of data between the host and the network. This observation stands out when we compare the throughput numbers of the Vup and the Cop1 models (or the Vup1 and the Cop1 models). In these two pieces of code, the number of instructions is the same. Yet, the coprocessor model results in a throughput increase of 1.3 to
about 9.4 times that of the Vupboard (I/O bus) model (depending on which machine was used, and which direction). Figure 6-2 shows the comparative performance of the Vup and the Cop1 models (based on the data in Table 6.3). This performance increase results from the fact that in the coprocessor interface model, the network resides in registers and is not memory-mapped. The path between the network and the general purpose registers is made much faster, because data can traverse this segment by one-cycle moves, whereas from the I/O bus, these accesses are very slow.

- The coprocessor model is faster than the DMA model also (see Figure 6-2, assuming we are dealing with an environment in which the processor's registers are the critical path (i.e. the processor needs to touch the data). Comparing the numbers for the Cop1 and the DMA models, we can see that the coprocessor interface is about 10-40% faster than the DMA interface. The reason for this increase in performance is again the avoidance of the memory-hierarchy in the segment of the path directly between the network and the general purpose registers.

- If we considered using the coprocessor host interface in an environment wherein the network data only needed to get transferred between the network and main memory, without needing to be touched by the processor, (as modelled by Cop3) then the performance gain compared to a programmed I/O interface is even higher. By comparing the figures of the Vup and the Cop3 models (or the Vup1 and the Cop6 models), we see that the Cop3 model is now between 1.6 and 14 times faster than the Vup model (again, depending on the machine used, and the direction of data). The reason for the added performance increase in this case is the fact that the Cop3 model can be coded with 14 fewer instructions in the tight data-moving loop (in which most of the time is spent by the receive and transmit routines). This is because it is possible to load and store data directly between main memory and the coprocessor registers, thus the cell, which is 14 words long, need only be moved once. In the Vupboard, however, data is moved
twice because it *has* to move via the processor registers.

- In such an environment, with the data needing to move directly between the memory and network, it might be more relevant to consider the DMA approach. The Turbochannel DMA board of Bellcore runs at 384 Mbps and 480 Mbps in the transmit and receive direction, respectively [13]. The maximum DMA rate achievable in the Turbochannel for a DECstation 5000/200 is 800 Mbps [14], but the observed rates are lower. One reason for lower observed rates is the overhead incurred in setting up a DMA transaction. The corresponding coprocessor model Cop6 in a "240" is comparable in its performance: the receive direction is faster by about 15%, whereas the transmit direction is slower by about 20%.

- Thus, while the coprocessor model's *raison d'être* was to provide a faster route between the network and the processor's registers, this model turns out to be a fairly good one—near the performance of a DMA model—even in environments that do not require the CPU to process the network data.

### 6.5.2 Transfer Direction

The transfer direction is another variable in our experiments which was under our control. The effect on throughput of the receive and transmit directions is summarized below. Figure 6-3 compares the directions of the Vup, the Cop1 and the DMA models.

- In the Vupboard models, the transmit direction was 1.5 to 2.8 times faster than the receive direction. We have already mentioned earlier that a write to the Turbochannel is known to require at least 3 cycles, while a read takes at least 11 cycles. This is what accounts for a major portion of the observed differences in throughput.

- The Cop6 model basically provides us with an indication of the relative performance of reads versus writes in a DECstation, because the inner loop consists mainly of 14 loads or 14 stores. Comparing the receive and transmit directions
leads us to learn that stores to memory are faster than loads from memory. In
the DECstation 5000/200, this difference is about 3.5 times, and in the DEC-
station 5000/240, it is about 1.8 times. The stores are faster probably because
of the write buffer that exists. In a write-through cache like the DECstations,
each write to cache results in a write to memory. However, the presence of a
write buffer improves the performance, as the writes get stored into this buffer,
and thus the memory latency does not have to be incurred by the processor.

6.5.3 Cell Operations

Often, the processor might need to perform certain transport level functions on the
cell, like swapping the bytes in a header for compatibility with the endian-ness of
a workstation, or some sort of checksumming. To determine how much such an
operation affects the throughput, we used the header byte swap as an example. This
operation takes 18 instructions, comparable, perhaps, to a checksum operation that
adds all the 14 words of a cell. The following observations can be made from the
results of Tables 6.3 and 6.4.

- In the coprocessor model, the throughput performance went up a fairly large
  amount (by about 22-75%) when the header byte swap operation was removed.
  This is because of the obvious speed to be gained from removing 18 instructions
from the data-moving loop. In fact, if we compare \textit{Cop1} to \textit{Cop3} (or \textit{Cop4} to \textit{Cop6}), the effect of removing the swap instructions increases. This is because the total number of instructions in the tight loop decreases and thus, the 18 swap instructions form a greater portion of the total.

- In the \textit{Vup} model, however, we see only very small increases in throughput—the most we see is 15%. This indicates that the removal of 18 instructions of a swap operation in the tight loop is not a major factor in this I/O channel interface. The delays incurred in accessing the I/O channel considerably diminish the effect of the swap operation, and the swap instructions seem to overlap the I/O instructions to some extent.

\subsection*{6.5.4 Processor Speed}

The two machines used for our measurements allowed us to study the effect of different processor speeds on throughput performance. Both DECstations had a Turbochannel running at 25 Mhz. The 5000/200 processor ("200") ran at 25 Mhz, and the 5000/240 processor ("240") ran at 40 Mhz. Some interesting points were observed with these comparisons.

- The receive throughput performance in the \textit{Vup} case is actually worse in a "240" machine than a "200", even though the processor is faster. This is explained by the fact that the Turbochannel and the processor are mismatched in speed in a "240" (the Turbochannel runs at 25 Mhz, which is mismatched with the 40 Mhz processor clock—unlike in a "200" in which case, the processor clock is also 25 MHz) Hence it takes longer to synchronize data transfers from the channel to the internal system bus.

- The throughput increases, as expected, if we look at the coprocessor approach, when we go from a "200" to a "240". Clearly, one factor in this increase is the faster clock rate. However, if this was the only factor, we would observe equal increases in both directions. However, we notice that the increases in the
transmit direction are greater. This leads us to conclude that the loads in a "240" are faster than the loads in a "200".

6.5.5 Summary

The results we have produced have enabled us to study two major areas. One is the comparison of different approaches to host interfacing. We observe that the coprocessor (registers) model yields the best performance if the processor needs to touch the data. Even if the network does not have to be touched by the processor, the coprocessor approach equals the performance of a DMA interface. This high performance in two different working environments makes the coprocessor approach a preferred design.

The analysis of these results has also enabled us to learn about some characteristics of a particular workstation (the DECstation, in this case). In our discussions, we have gained knowledge about aspects like the difference in the speeds of loads and stores, and the access rates to memory versus I/O.
Chapter 7

Conclusion

The woods are lovely, dark and deep,
But I have promises to keep,
And miles to go before I sleep,
Miles to go before I sleep...

Robert Frost.

In this thesis, we first described the work that was being done in the area of host interfacing. A study of this work led us to identify the key issues that had been the focus of attention by the designers of host interfaces. We identified one other issue—the location of the connection—which had not been given enough attention in the context of networked workstations, although this issue has been focussed upon in the context of multiprocessor systems. We then suggested the architecture of a coprocessor-style of host interfacing, and also described the work done in the design of a VLSI implementation of this architecture. Next, we performed some measurements that helped us analyze the performance improvements that can be achieved by this architecture.

In concluding this thesis, we first give suggestions for future directions which should be taken in the field of host interfacing. We then highlight some of the main contributions of the work presented herein.
7.1 Future Work

The work described in this thesis suggests further directions which can be taken in the design of host interfaces. These suggestions are presented in this section.

7.1.1 Tightly-Coupled Interfaces

We have seen that a tightly-coupled coprocessor network interface for a host DECstation yields a high throughput, and provides the processor with a flexible mode of interfacing to the network. This suggests to us that similar tightly-coupled interfaces should be designed for other processor architectures too.

We have seen designs of closely-coupled network interfaces in the context of multiprocessor systems. The coprocessor interface for the MC88110 being designed for the *T project [22], and the SPARC cache controller of the Alewife project [32] are two such examples. One possibility, then, is to adapt these interfaces for the context of networked workstations. For example, a host SPARCstation can be modified to contain the special SPARC processor designed in the Alewife group, connected to their cache controller network interface. This interface could then be compared to one which attaches the network to the Standard I/O Bus (the SBus, in a SPARCstation [40]), such as the SBus interface designed in our group, or the one described in [11]. In the memory-based messaging system of [8], the network is also made to appear to reside in cache, but messages can be sent to specific addresses explicitly. This approach could also be extended to the area of networked workstations.

7.1.2 Memory Bus Interface

In Section 3.4.1, we saw a high throughput might be achieved between the processor and the network if the interface board substituted for a memory bank, and the workstation architecture had a wide cache line. The IBM RS/6000 was deemed to be the perfect candidate for this model of host interfacing. The design of such an interface would provide an interesting research project, since it would explore a connection to the Memory Bus, which we have not fully considered in this thesis.

106
Figure 7-1: Memory-Bus Interface for RS/6000

We see in Figure 7-1 a possible design of such an interface. In this design, the instructions to the interface board are memory-mapped addresses. For this purpose, the Address Decoder Logic detects the addresses on the memory bus that cause the board to transfer data, or read/write control and status information. The buffers help to implement the pipelining that would be required to adhere to the protocol of the memory bus. The Finite State Machine consists of routines that generate signals to receive and transmit data to a network (VuNet, in our case) switch.

7.2 Summary

The contributions of this thesis are both general and specific. In general terms, this thesis has attempted to present a framework within which we can design and understand host interfaces. In specific terms, we have designed a particular host interface—the coprocessor model—within this framework, and have studied its per-
formance benefits. We now summarize these two-fold contributions.

7.2.1 Framework for Host Interface Design

In this thesis, we have identified the key issues which we believe define the design space of host interfaces. A thorough approach to host interface design can be taken by answering these three questions—the division of protocol processing between the host and the interface board, the method of data transfer, and the point of attachment in the memory hierarchy of a workstation. These issues are not new and, in fact, were identified after a study of current host interface designs. However most of the previous work on host interfacing, including comparative studies of different approaches, has focussed on one of these issues at a time. Since these issues are fairly orthogonal (see Figure 2-5), they can be answered independent of each other (usually), and hence we feel that all these issues need to be considered.

We have also suggested that these questions should be answered keeping in mind the operating environment of the particular system under study. Specifically, we believe that it is important to consider how much interaction the processor of the host is going to have with the network data. The answers to the design questions can be significantly different, depending on whether the processor is expected to touch all the words of the network data or whether the processor is going to ignore the contents of the network data.

In order to concentrate on the relatively under-emphasized question of the choice of the point of attachment, we have also presented a model of the hierarchical memory system of a workstation which allows us to identify such potential "plug-points" for a network interface. In discussing these points of connection, we have suggested some flavors of the considerations that need to be given, like the bandwidth of the various buses involved, or the organization of the cache. We have also provided a simple, but fairly accurate indicator (the normalized throughput index) of the relative performance of these different attachment points.
7.2.2 Coprocessor Interface Performance

Within this general framework of host interfacing, we designed a coprocessor-style host interface, which is tightly coupled to the host processor, sitting on the System Bus of the workstation (a DECstation 5000).

One of the specific research thrusts of this project was to study the performance difference of host interfaces which attach to the workstation at different locations. With the results of Chapter 6, we have shown that the location of attachment of a network to a workstation is an important issue that should be considered while designing host interfaces. The difference in throughput between two interfaces which differ only in their location of attachment has been shown to be as large as a factor of 9. We thus think that future designers of host interfaces should consider which point of connection in the memory hierarchy of a workstation would provide the best performance. The other issues involved in the design of host interfaces, mentioned earlier, have been the focus of attention in the past, and should continue to be considered important also.

As a particular example, our performance analysis shows that, in an environment in which the processor needs to touch the network data, a closely-coupled coprocessor interface yields a higher throughput between the workstation and the network, compared to an interface which is attached to the Standard I/O Bus. We saw that the throughput can be 1.5 to about 9 times higher, depending on the workstation we use, and the direction of data transfer.

The coprocessor interface also is shown to have a high throughput in an environment in which the network data does not have to reach the processor for consumption, but can reach the main memory directly. Thus, a tightly-coupled programmed I/O interface is seen to have throughput rates comparable to a loosely-coupled DMA interface (by "loosely-coupled", we mean an interface connected to a bus far away from the processor, namely the Standard I/O Bus, in this case). This result further emphasizes the importance of the point of connection in the design process of host interfaces.

109
Appendix A

Instruction Set

In this appendix, we present details of the instruction set supported by the Co-processor Host Interface Chip. These instructions follow the coprocessor interface instructions defined for the MIPS R3000 processor. Note that all the instructions are described with the assumption that the coprocessor number is 3. The coprocessor number can be changed, as explained in Chapter 5. For the full instruction set of the MIPS R3000, the reader is referred to [31], from which these descriptions are derived. The instructions described are the ones listed in Table 5.1.
BC3F: Branch on Coprocessor 3 False

Format:

```
  COP3  Branch  False  offset
  010011  01000  00000  offset
```

Syntax: BC3F offset

Description: A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended to 32-bits. If the contents of coprocessor 3’s condition line (CpCond) is false, then the program branches to the target address with a delay of one instruction.
BC3T: Branch on Coprocessor 3 True

Format:

<table>
<thead>
<tr>
<th></th>
<th>COP3</th>
<th>Branch</th>
<th>True</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td>01000</td>
<td>00001</td>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>

Syntax: BC3T offset

Description: A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended to 32-bits. If the contents of coprocessor 3’s condition line (CpCond) is true, then the program branches to the target address with a delay of one instruction.
CFC3: Move Control from Coprocessor 3

Format:

<table>
<thead>
<tr>
<th>COP3</th>
<th>CF</th>
<th>rt</th>
<th>rd</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td>00010</td>
<td>rt</td>
<td>rd</td>
<td>0000000000</td>
</tr>
</tbody>
</table>

Syntax: CFC3 rt, rd

Description: The contents of coprocessor control register rd of coprocessor 3 are loaded into general register rt.
**DIS:** Disable Interrupts of Coprocessor 3

**Format:**

![Table]

**Syntax:** COP3 c0

**Description:** The interrupts on the coprocessor chip are disabled by resetting the interrupt enable (bit 31) of the coprocessor control register.
ENA: Enable Interrupts of Coprocessor 3

Format:

Syntax: COP3 c1

Description: The interrupts on the coprocessor chip are enabled by setting the interrupt enable (bit 31) of the coprocessor control register.
TRA: Transmit Cell from Register File of Coprocessor 3

Format:

```
  31  26  25  24  23  22  21  20  19  18  17  16  15  14  0

COP3  c3  0  ct  cd  unused
```

```
010011  11000  0  ct  cd  0000000000000000
```

Syntax: COP3 c3, ct, cd

Description: A 64-byte cell is transferred from coprocessor register file cell location cd to cell buffer ct of coprocessor 3.
REC: Receive Cell to Register File of Coprocessor 3

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP3</td>
<td>c2</td>
<td>0</td>
<td>ct</td>
<td>cd</td>
<td>unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010011</td>
<td>10010</td>
<td>0</td>
<td>ct</td>
<td>cd</td>
<td>0000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax: COP3 c2, ct, cd

Description: A 64-byte cell is transferred from cell buffer ct to coprocessor register file cell location cd of coprocessor 3.
CTC3: Move Control to Coprocessor 3

Format:

Syntax: CTC3 rt, rd

Description: The contents of general register rt are loaded into coprocessor control register rd of coprocessor 3.
LWC3: Load Word to Coprocessor 3:

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWC3</td>
<td>base</td>
<td>rt</td>
<td></td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110011</td>
<td>base</td>
<td>rt</td>
<td></td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax: LWC3 rt, offset(base)

Description: The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address. The contents of the word at the memory location specified by the effective address are loaded into coprocessor register rt of coprocessor 3.
MFC3: Move from Coprocessor 3

Format:

```
+---+---+---+---+---+---+---+---+---+
| 31| 26| 25| 24| 23| 22| 16| 15| 11| 10|  0 |
+---+---+---+---+---+---+---+---+---+---+---+
| C | P3| MF| rt| rd|   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+
| 01| 00| 11| 00| 00| rt| rd|   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+
```

Syntax: MFC3 rt, rd

Description: The contents of coprocessor register rd of coprocessor 3 are loaded into general register rt.
MTC3: Move to Coprocessor 3

Format:

```
  31  26 25  21 20  16 15 11 10  0

  COP3   MT  rt  rd  unused
  010011 00100 rt  rd  0000000000
```

Syntax: MTC3 rt, rd

Description: The contents of general register rt are loaded into coprocessor register rd of coprocessor 3.
SWC3: Store Word from Coprocessor 3:

Format:

Syntax: SWC3 rt, offset(base)

Description: The 16-bit offset is sign-extended and added to the contents of general register base to form a 32-bit unsigned effective address. The contents of coprocessor register rt of coprocessor 3 are loaded into the word at memory location specified by the effective address.
Appendix B

Diagrams and Tables

This appendix gives the floorplan of the design. A layout of the chip within the PGA package is shown, showing the density of the design. Detailed timing measurements are then presented in Tables B.1 and B.2.
Figure B-1: Chip Floorplan
Figure B-2: Chip Layout
Timing Measurements

These tables give the time it takes to transfer a burst of cells in the receive and transmit directions. The burst size used was 4000 cells, and the loop was run 1000 times. The models shown are explained in Table 6.1.

<table>
<thead>
<tr>
<th>Model</th>
<th>Burst: (4000 Cells) x1000 (Time in s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Receive</td>
</tr>
<tr>
<td>Bare</td>
<td>0.35</td>
</tr>
<tr>
<td>Vup</td>
<td>46.19</td>
</tr>
<tr>
<td>Vup1</td>
<td>44.65</td>
</tr>
<tr>
<td>Cop1</td>
<td>7.51</td>
</tr>
<tr>
<td>Cop2</td>
<td>6.05</td>
</tr>
<tr>
<td>Cop3</td>
<td>5.22</td>
</tr>
<tr>
<td>Cop4</td>
<td>5.15</td>
</tr>
<tr>
<td>Cop5</td>
<td>3.74</td>
</tr>
<tr>
<td>Cop6</td>
<td>3.56</td>
</tr>
<tr>
<td>DMA1</td>
<td>6.59</td>
</tr>
</tbody>
</table>

Table B.1: Times for Entire Cell Burst: DECstation 5000/240
<table>
<thead>
<tr>
<th>Model</th>
<th>Burst: (4000 Cells) x1000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Time in s)</td>
</tr>
<tr>
<td></td>
<td>Receive</td>
</tr>
<tr>
<td>Bare</td>
<td>0.49</td>
</tr>
<tr>
<td>Vup</td>
<td>31.76</td>
</tr>
<tr>
<td>Vup1</td>
<td>28.87</td>
</tr>
<tr>
<td>Cop1</td>
<td>8.99</td>
</tr>
<tr>
<td>Cop2</td>
<td>6.61</td>
</tr>
<tr>
<td>Cop3</td>
<td>6.37</td>
</tr>
<tr>
<td>Cop4</td>
<td>5.80</td>
</tr>
<tr>
<td>Cop5</td>
<td>3.87</td>
</tr>
<tr>
<td>Cop6</td>
<td>3.82</td>
</tr>
<tr>
<td>DMA</td>
<td>10.62</td>
</tr>
<tr>
<td>DMA1</td>
<td>7.96</td>
</tr>
</tbody>
</table>

Table B.2: Times for Entire Cell Burst: DECstation 5000/200
Appendix C

Diagnostic Code

This appendix contains segments of diagnostic code. The main c-code was written by Chris Lindblad. This code was used for testing the Vup case (see Table: 6.1 for more information). This code was then disassembled, and the MIPS R3000 assembly language segments were changed to simulate the other environments discussed in Section 6.3. Note that for the sake of conciseness, we have only included relevant segments of the code (the inner-most loop).

- C-code: Vup model.

- Assembly code:
  - Vup model.
  - Vup1 model.
  - Cop1 model.
  - Cop2 model.
  - Cop3 model.
  - Cop4 model.
  - Cop5 model.
  - Cop6 model.
C-Code: Vup Model

/**
 * This code segment uses programmed I/O to receive a given burst size of cells from the Turbochannel Vupboard for a given number of cycles. The number of cells and cycles are input by the user.
 */

static int
RxBurstCmd(clientData, interp, argc, argv)
    ClientData clientData;
    Tcl_Interp *interp;
    int argc;
    char **argv;
{
    /**** Initialization ************/

    int i, j, cells;
    int cycles = 1;
    int nowait = 0;
    int status, header_0, header_1;
    int payload_0, payload_1, payload_2, payload_3;
    int payload_4, payload_5, payload_6, payload_7;
    int payload_8, payload_9, payload_10, payload_11;
    int *rbufp;
    volatile int *vup_cell = &vup_reg->vr_cell[0];

    /**** Check validity of inputs *******/

    if (argc < 2 || argc > 4) return
        TclXtErrArgCnt(interp, argv[0], "cells ?cycles? ?nowait?");
    if (TclXtGetInt(interp, argv[1], &cells) != TCL_OK)
        return TCL_ERROR;
    if (argc > 2) {
        if (TclXtGetInt(interp, argv[2], &cycles) != TCL_OK)
            return TCL_ERROR;
        if (argc > 3) {
            if (TclXtGetInt(interp, argv[2], &nowait) != TCL_OK)
                return TCL_ERROR;
        }
    }
    if (cells > TXRX_BUFSIZE/14)
        return TclXtErrBadVal(interp, "few enough cells to fit in the buffer", argv[1]);
for (j=0; j<cycles; j++) {
    rbufp = rdbuf;
    for (i=0; i<cells; i++) {
        do {
            status = vup_reg->vr_csr;
        } while (!status & VUP_REFMASK) && !nowait);
        /*
        * vup -> registers
        */
        (void) vup_cell[2]; /* Cause an RCLK with no HDR */
        header_0 = vup_cell[0];
        header_1 = vup_cell[1];
        payload_0 = vup_cell[2];
        payload_1 = vup_cell[3];
        payload_2 = vup_cell[4];
        payload_3 = vup_cell[5];
        payload_4 = vup_cell[6];
        payload_5 = vup_cell[7];
        payload_6 = vup_cell[8];
        payload_7 = vup_cell[9];
        payload_8 = vup_cell[10];
        payload_9 = vup_cell[11];
        payload_10 = vup_cell[12];
        payload_11 = vup_cell[13];
        /*
        * Byte swap header
        */
        header_0 = VUP_MTOH(header_0);
        header_1 = VUP_MTOH(header_1);
        /*
        * registers -> memory
        */
        rbufp[0] = header_0;
        rbufp[1] = header_1;
        rbufp[2] = payload_0;
        rbufp[3] = payload_1;
        rbufp[4] = payload_2;
        rbufp[5] = payload_3;
        rbufp[6] = payload_4;
        rbufp[7] = payload_5;
        rbufp[8] = payload_6;
        rbufp[9] = payload_7;
        rbufp[10] = payload_8;
rbufp[12] = payload_10;
rbufp[13] = payload_11;
rbufp += 14;
}
}
return TclXtReturnNull(interp);
}

一口气用程序I/O传输给定大小的burst cell数组到Turbochannel Vupboard并给定周期数。用户输入细胞数和周期数。

```
static int
TxBurstCmd(clientData, interp, argc, argv)
    ClientData clientData;
    Tcl_Interp *interp;
    int argc;
    char **argv;
{

    /**** Initialization ***********/

    int i, j, cells;
    int cycles = 1;
    int tport = 0;
    int status;
    int header_0, header_1;
    int payload_0, payload_1, payload_2, payload_3;
    int payload_4, payload_5, payload_6, payload_7;
    int payload_8, payload_9, payload_10, payload_11;
    int *tbufp;
    volatile int *vup_cell = &vup_reg->vr_cell[0];

    /**** Check validity of inputs *******/

    status = vup_reg->vr_csr;
tport = VUP_GETIPT(status);
if (argc < 2 || argc > 4) return
    TclXtErrArgCnt(interp, argv[0], "cells ?cycles? ?tport?");
if (TclXtGetInt(interp, argv[1], &cells) != TCL_OK)
    return TCL_ERROR;
if (argc > 2) {
    if (TclXtGetInt(interp, argv[2], &cycles) != TCL_OK)
        return TCL_ERROR;
    if (argc > 3) {
        if (GetPort(interp, argv[3], &tport) != TCL_OK)
```
return TCL_ERROR;
}
}

if (cells > TXRX_BUFSIZE/14)
    return TclXtFrrBadVal(interp, "few enough cells to
                fit in the buffer", argv[1]);

/**** Start inner loop *************/

for (j=0; j<cycles; j++) {
    tbufp = txbuf;
    for (i=0; i<cells; i++) {
        /*
         * memory -> registers
         */
        header_0 = tbufp[0];
        header_1 = tbufp[1];
        payload_0 = tbufp[2];
        payload_1 = tbufp[3];
        payload_2 = tbufp[4];
        payload_3 = tbufp[5];
        payload_4 = tbufp[6];
        payload_5 = tbufp[7];
        payload_6 = tbufp[8];
        payload_7 = tbufp[9];
        payload_8 = tbufp[10];
        payload_9 = tbufp[11];
        payload_10 = tbufp[12];
        payload_11 = tbufp[13];
        tbufp += 14;
        /*
         * Byte swap header
         */
        header_0 = VUP_HTON(header_0);
        header_1 = VUP_HTON(header_1);
        /*
         * registers -> vup
         */
        vup_reg->vr_csr = VUP_SETTPT(tport);
        vup_cell[0] = header_0;
        vup_cell[1] = header_1;
        vup_cell[2] = payload_0;
        vup_cell[3] = payload_1;
        vup_cell[4] = payload_2;
        vup_cell[5] = payload_3;
        vup_cell[6] = payload_4;
vup_cell[7] = payload_5;
vup_cell[8] = payload_6;
vup_cell[9] = payload_7;
vup_cell[10] = payload_8;
vup_cell[12] = payload_10;
vup_cell[13] = payload_11;
} }

return TclXtReturnNull(interp);
}
Assembly Code: Vup Model

/**************************************************************************
This is the inner loop of routine 'RxBurstCmd' which receives cells from the Turbochannel Vupboard.
**************************************************************************/

RxBurstCmd:

/***** Vup --> Registers ****************************/

[vupTests.c: 609] 0x401704: 8e390000 lw t9,0(s1)
[vupTests.c: 610] 0x401708: 8e300004 lw s0,4(s1)
[vupTests.c: 611] 0x40170c: 8e2d0008 lw t5,8(s1)
[vupTests.c: 612] 0x401710: 8e2e000c lw t6,12(s1)
[vupTests.c: 613] 0x401714: 8e2f0010 lw t7,16(s1)
[vupTests.c: 614] 0x401718: 8e2c0014 lw t4,20(s1)
[vupTests.c: 615] 0x40171c: 8e2b0018 lw t3,24(s1)
[vupTests.c: 616] 0x401720: 8e2a001c lw t2,28(s1)
[vupTests.c: 617] 0x401724: 8e290020 lw t1,32(s1)
[vupTests.c: 618] 0x401728: 8e280024 lw t0,36(s1)
[vupTests.c: 619] 0x40172c: 8e270028 lw a3,40(s1)
[vupTests.c: 620] 0x401730: 8e26002c lw a2,44(s1)
[vupTests.c: 621] 0x401734: 8e250030 lw a1,48(s1)
[vupTests.c: 622] 0x401738: 8e240034 lw a0,52(s1)

/***** Byte Swap Header ****************************/

[vupTests.c: 626] 0x401744: 3322ff00 andi v0,t9,0xff00
[vupTests.c: 626] 0x401748: 00021200 sll v0,v0,8
[vupTests.c: 626] 0x40174c: 00191e00 sll v1,t9,24
[vupTests.c: 626] 0x401750: 00621821 addu v1,v1,v0
[vupTests.c: 626] 0x401754: 03361024 and v0,t9,s6
[vupTests.c: 626] 0x401758: 00021202 srl v0,v0,8
[vupTests.c: 626] 0x40175c: 00621821 addu v1,v1,v0
[vupTests.c: 626] 0x401760: 00191602 srl v0,t9,24
[vupTests.c: 626] 0x401764: 0062c821 addu t9,v1,v0
[vupTests.c: 627] 0x401768: 3202ff00 andi v0,s0,0xff00
[vupTests.c: 627] 0x40176c: 00021200 sll v0,v0,8
[vupTests.c: 627] 0x401770: 00101e00 sll v1,s0,24
[vupTests.c: 627] 0x401774: 00621821 addu v1,v1,v0
[vupTests.c: 627] 0x401778: 02161024 and v0,s0,s6
[vupTests.c: 627] 0x40177c: 00021202 srl v0,v0,8
[vupTests.c: 627] 0x401780: 00621821 addu v1,v1,v0

134
[vupTests.c: 627] 0x401784: 00101602 srl v0,s0,24
[vupTests.c: 627] 0x401788: 00628021 addu s0,v1,v0

/****** Registers --> Memory *************/

[vupTests.c: 631] 0x40178c: ae790000 sw t9,0(s3)
[vupTests.c: 632] 0x401790: af10ffd0 sw s0,-48(t8)
[vupTests.c: 633] 0x401794: af04ffdc sw t5,-44(t8)
[vupTests.c: 634] 0x401798: af0eef8 sw t6,-40(t8)
[vupTests.c: 635] 0x40179c: af0fffc sw t7,-36(t8)
[vupTests.c: 636] 0x4017a0: af0cffe0 sw t4,-32(t8)
[vupTests.c: 637] 0x4017a4: af0bbe4 sw t3,-28(t8)
[vupTests.c: 638] 0x4017a8: af0affe8 sw t2,-24(t8)
[vupTests.c: 639] 0x4017ac: af09ffec sw t1,-20(t8)
[vupTests.c: 640] 0x4017b0: af08fff0 sw t0,-16(t8)
[vupTests.c: 641] 0x4017b4: af07ffff sw a3,-12(t8)
[vupTests.c: 642] 0x4017b8: af06ffff sw a2,-8(t8)
[vupTests.c: 643] 0x4017bc: af05ffff sw a1,-4(t8)
[vupTests.c: 644] 0x4017c0: af040000 sw a0,0(t8)

/* This is the inner loop of routine 'TxBurstCmd' which transmits
   cells to the Turbochannel Vupbard. */

TxBurstCmd:

/****** Memory --> Registers **************/

[vupTests.c: 420] 0x40119c: 8f19fffd0 lw t9,-48(t8)
[vupTests.c: 421] 0x4011a0: 8f05ffdc lw a1,-44(t8)
[vupTests.c: 422] 0x4011a4: 8f06fffd lw a2,-40(t8)
[vupTests.c: 423] 0x4011a8: 8f07fffd lw a3,-36(t8)
[vupTests.c: 424] 0x4011ac: 8f08ffee lw t0,-32(t8)
[vupTests.c: 425] 0x4011b0: 8f09ffee lw t1,-28(t8)
[vupTests.c: 426] 0x4011b4: 8f0afffe lw t2,-24(t8)
[vupTests.c: 427] 0x4011b8: 8f0bffee lw t3,-20(t8)
[vupTests.c: 428] 0x4011bc: 8f0cffff lw t4,-16(t8)
[vupTests.c: 429] 0x4011c0: 8f0dfffe lw t5,-12(t8)
[vupTests.c: 430] 0x4011c4: 8f0effff lw t6,-8(t8)
[vupTests.c: 431] 0x4011c8: 8f0ffffe lw t7,-4(t8)
[vupTests.c: 432] 0x4011cc: 8f100000 lw a0,0(t8)
[vupTests.c: 419] 0x4011d0: 8e640000 lw a0,0(s3)

/****** Byte Swap Header **********************/

135
[vupTests.c: 437] 0x4011e0: 3082ff00 andi v0,a0,0xff00
[vupTests.c: 437] 0x4011e4: 00021200 sll v0,v0,8
[vupTests.c: 437] 0x4011e8: 00041e00 sll v1,a0,24
[vupTests.c: 437] 0x4011ec: 00621821 addu v1,v1,v0
[vupTests.c: 437] 0x4011f0: 00951024 and v0,a0,s5
[vupTests.c: 437] 0x4011f4: 00021202 srl v0,v0,8
[vupTests.c: 437] 0x4011f8: 00621821 addu v1,v1,v0
[vupTests.c: 437] 0x4011fc: 00042602 srl a0,a0,24
[vupTests.c: 438] 0x401220: 3322ff00 andi v0,t9,0xff00
[vupTests.c: 438] 0x401224: 00021200 sll v0,v0,8
[vupTests.c: 438] 0x401228: 00191e00 sll v1,t9,24
[vupTests.c: 438] 0x40122c: 00621821 addu v1,v1,v0
[vupTests.c: 438] 0x401230: 03351024 and v0,t9,s5
[vupTests.c: 438] 0x401234: 00021202 srl v0,v0,8
[vupTests.c: 438] 0x401238: 00621821 addu v1,v1,v0
[vupTests.c: 438] 0x40123c: 8fa40030 lw a0,48(sp)
[vupTests.c: 438] 0x401244: 00191602 srl v0,t9,24
[vupTests.c: 438] 0x401248: 0062c821 addu t9,v1,v0

/*** Registers --> Vup **********************************************/
[vupTests.c: 443] 0x40121c: ae240000 sw a0,0(s1)
[vupTests.c: 444] 0x40124c: ae390004 sw t9,4(s1)
[vupTests.c: 445] 0x401250: ae250008 sw a1,8(s1)
[vupTests.c: 446] 0x401254: ae26000c sw a2,12(s1)
[vupTests.c: 447] 0x401258: ae270010 sw a3,16(s1)
[vupTests.c: 448] 0x40125c: ae280014 sw t0,20(s1)
[vupTests.c: 449] 0x401260: ae290018 sw t1,24(s1)
[vupTests.c: 450] 0x401264: ae2a001c sw t2,28(s1)
[vupTests.c: 451] 0x401268: ae2b0020 sw t3,32(s1)
[vupTests.c: 452] 0x40126c: ae2c0024 sw t4,36(s1)
[vupTests.c: 453] 0x401270: ae2d0028 sw t5,40(s1)
[vupTests.c: 454] 0x401274: ae2e002c sw t6,44(s1)
[vupTests.c: 455] 0x401278: ae2f0030 sw t7,48(s1)
[vupTests.c: 456] 0x40127c: ae300034 sw t8,52(s1)
Assembly Code: Vup1 Model

 Madagascar

/*****************************/
This is the inner loop of routine 'RxBurst1Cmd' which receives
cells from the Turbochannel Vupboard.
/****************************/ 

RxBurst1Cmd:

/***** Vup --> Registers ***********/

[vupTests.c: 703] 0x401978: 8e020000 lw v0,0(s0)
[vupTests.c: 704] 0x40197c: 8e030004 lw v1,4(s0)
[vupTests.c: 705] 0x401980: 8e040008 lw a0,8(s0)
[vupTests.c: 706] 0x401984: 8e05000c lw a1,12(s0)
[vupTests.c: 707] 0x401988: 8e060010 lw a2,16(s0)
[vupTests.c: 708] 0x40198c: 8e070014 lw a3,20(s0)
[vupTests.c: 709] 0x401990: 8e080018 lw t0,24(s0)
[vupTests.c: 710] 0x401994: 8e09001c lw t1,28(s0)
[vupTests.c: 711] 0x401998: 8e0a0020 lw t2,32(s0)
[vupTests.c: 712] 0x40199c: 8e0b0024 lw t3,36(s0)
[vupTests.c: 713] 0x4019a0: 8e0c0028 lw t4,40(s0)
[vupTests.c: 714] 0x4019a4: 8e0d002c lw t5,44(s0)
[vupTests.c: 715] 0x4019a8: 8e0e0030 lw t6,48(s0)
[vupTests.c: 716] 0x4019ac: 8e0f0034 lw t7,52(s0)

/***** Registers --> Memory *************/

[vupTests.c: 725] 0x4019b4: ae220000 sw v0,0(s1)
[vupTests.c: 726] 0x4019b8: af03ff00 sw v1,-48(t8)
[vupTests.c: 727] 0x4019bc: af04ff04 sw a0,-44(t8)
[vupTests.c: 728] 0x4019c0: af05ff08 sw a1,-40(t8)
[vupTests.c: 729] 0x4019c4: af06ff0c sw a2,-36(t8)
[vupTests.c: 730] 0x4019c8: af07ffe0 sw a3,-32(t8)
[vupTests.c: 731] 0x4019cc: af08fffc sw t0,-28(t8)
[vupTests.c: 732] 0x4019d0: af09ffe8 sw t1,-24(t8)
[vupTests.c: 733] 0x4019d4: af0affec sw t2,-20(t8)
[vupTests.c: 734] 0x4019d8: af0bff00 sw t3,-16(t8)
[vupTests.c: 735] 0x4019dc: af0cf004 sw t4,-12(t8)
[vupTests.c: 736] 0x4019e0: af0dfffc sw t5,-8(t8)
[vupTests.c: 737] 0x4019e4: af0efffc sw t6,-4(t8)
[vupTests.c: 738] 0x4019e8: af0f0000 sw t7,0(t8)
/********************************************************************
This is the inner loop of routine 'TxBurst1Cmd' which transmits
cells to the Turbochannel Vupboard.
*********************************************************************/

TxBurst1Cmd:

/***** Memory --> Registers ***************/

[vupTests.c: 514] 0x401450: 8e640000 lw a0,0(s3)
[vupTests.c: 515] 0x401414: 8f05ff0d lw a1,-48(t8)
[vupTests.c: 516] 0x401418: 8f06ff0d lw a2,-44(t8)
[vupTests.c: 517] 0x40141c: 8f07ff0d lw a3,-40(t8)
[vupTests.c: 518] 0x401420: 8f08ff0d lw t0,-36(t8)
[vupTests.c: 519] 0x401424: 8f09ff0d lw t1,-32(t8)
[vupTests.c: 520] 0x401428: 3f0aff0d lw t2,-28(t8)
[vupTests.c: 521] 0x40142c: 8f0bff0d lw t3,-24(t8)
[vupTests.c: 522] 0x401430: 8f0cff0d lw t4,-20(t8)
[vupTests.c: 523] 0x401434: 8f0dff0d lw t5,-16(t8)
[vupTests.c: 524] 0x401438: 8f0eff0d lw t6,-12(t8)
[vupTests.c: 525] 0x40143c: 8f0fff0d lw t7,-8(t8)
[vupTests.c: 526] 0x401440: 8f10ff0d lw s0,-4(t8)
[vupTests.c: 527] 0x401444: 8f110000 lw s1,0(t8)

/***** Registers --> Vup ***************/

[vupTests.c: 538] 0x401464: ae400000 sw a0,0(s2)
[vupTests.c: 539] 0x401478: ae450004 sw a1,4(s2)
[vupTests.c: 540] 0x40147c: ae460008 sw a2,8(s2)
[vupTests.c: 541] 0x401480: ae47000c sw a3,12(s2)
[vupTests.c: 542] 0x401484: ae480010 sw t0,16(s2)
[vupTests.c: 543] 0x401488: ae490014 sw t1,20(s2)
[vupTests.c: 544] 0x40148c: ae4a0018 sw t2,24(s2)
[vupTests.c: 545] 0x401490: ae4b001c sw t3,28(s2)
[vupTests.c: 546] 0x401494: ae4c0020 sw t4,32(s2)
[vupTests.c: 547] 0x401498: ae4d0024 sw t5,36(s2)
[vupTests.c: 548] 0x40149c: ae4e0028 sw t6,40(s2)
[vupTests.c: 549] 0x4014a0: ae4f002c sw t7,44(s2)
[vupTests.c: 550] 0x4014a4: ae500030 sw s0,48(s2)
[vupTests.c: 551] 0x4014a8: ae510034 sw s1,52(s2)

138
Assembly Code: Cop1 Model

/********************
This is the inner loop of routine 'RxBurstCop1Cmd' which receives
cells from the Coprocessor.
**************************

RxBurstCop1Cmd:

/***** Cop Regs --> Registers **************

[vupTests.c: 997] 0x401e08: 44192000 mfc1 t9,$f4
[vupTests.c: 998] 0x401e0c: 44102800 mfc1 s0,$f5
[vupTests.c: 999] 0x401e10: 440d3000 mfc1 t5,$f6
[vupTests.c: 1000] 0x401e14: 440e3800 mfc1 t6,$f7
[vupTests.c: 1001] 0x401e18: 440f4000 mfc1 t7,$f8
[vupTests.c: 1002] 0x401e1c: 440c4800 mfc1 t4,$f9
[vupTests.c: 1003] 0x401e20: 440b5000 mfc1 t3,$f10
[vupTests.c: 1004] 0x401e24: 440a6000 mfc1 t2,$f12
[vupTests.c: 1005] 0x401e28: 44096800 mfc1 t1,$f13
[vupTests.c: 1006] 0x401e2c: 44087000 mfc1 t0,$f14
[vupTests.c: 1007] 0x401e30: 44078000 mfc1 a3,$f16
[vupTests.c: 1008] 0x401e34: 44068800 mfc1 a2,$f17
[vupTests.c: 1009] 0x401e38: 44059000 mfc1 a1,$f18
[vupTests.c: 1010] 0x401e40: 44042000 mfc1 a0,$f4

/***** Header Byte Swap *******************

[vupTests.c: 1014] 0x401e48: 3322ff00 andi v0,t9,0xff00
[vupTests.c: 1014] 0x401e4c: 00021200 sll v0,v0,8
[vupTests.c: 1014] 0x401e50: 00191e00 sll v1,t9,24
[vupTests.c: 1014] 0x401e54: 00621821 addu v1,v1,v0
[vupTests.c: 1014] 0x401e58: 03361024 and v0,t9,s6
[vupTests.c: 1014] 0x401e5c: 00021202 srl v0,v0,8
[vupTests.c: 1014] 0x401e60: 00621821 addu v1,v1,v0
[vupTests.c: 1014] 0x401e64: 00191e00 srl v0,t9,24
[vupTests.c: 1014] 0x401e68: 0052c821 addu t9,v1,v0
[vupTests.c: 1015] 0x401e6c: 3202ff00 andi v0,s0,0xff00
[vupTests.c: 1015] 0x401e70: 00021200 sll v0,v0,8
[vupTests.c: 1015] 0x401e74: 00101e00 sll v1,s0,24
[vupTests.c: 1015] 0x401e78: 00621821 addu v1,v1,v0
[vupTests.c: 1015] 0x401e7c: 02161024 and v0,s0,s6
[vupTests.c: 1015] 0x401e80: 00021202 srl v0,v0,8
[vupTests.c: 1015] 0x401e84: 00621821 addu v1,v1,v0
[vupTests.c: 1015] 0x401e88: 00101602 srl v0,s0,24
[vupTests.c: 1015] 0x401e8c: 00628021 addu s0,v1,v0

/TR****** Registers --> Memory ***********************/

[vupTests.c: 1019] 0x401e90: ae790000 sw t9,0(s3)
[vupTests.c: 1020] 0x401e94: af10ff00 sw s0,-48(t8)
[vupTests.c: 1021] 0x401e98: af0dfffd sw t5,-44(t8)
[vupTests.c: 1022] 0x401e9c: af0effd8 sw t6,-40(t8)
[vupTests.c: 1023] 0x401ea0: af0ffffc sw t7,-36(t8)
[vupTests.c: 1024] 0x401ea4: af0cffe0 sw t4,-32(t8)
[vupTests.c: 1025] 0x401ea8: af0bffe4 sw t3,-28(t8)
[vupTests.c: 1026] 0x401eac: af0affe8 sw t2,-24(t8)
[vupTests.c: 1027] 0x401eb0: af09ffec sw t1,-20(t8)
[vupTests.c: 1028] 0x401eb4: af08ff00 sw t0,-16(t8)
[vupTests.c: 1029] 0x401eb8: af07fff4 sw a3,-12(t8)
[vupTests.c: 1030] 0x401ebc: af06fff8 sw a2,-8(t8)
[vupTests.c: 1031] 0x401ec0: af05fff0 sw a1,-4(t8)
[vupTests.c: 1032] 0x401ec4: af040000 sw a0,0(t8)

/******************************************************************************
This is the inner loop of routine 'TxBurstCop1Cmd' which transmits
cells to the Coprocessor.
*******************************************************************************/

TxBurstCop1Cmd:

/***** Memory --> Registers ***************/

[vupTests.c: 800] 0x401910: 8f13ffdf 1w t9,-48(t8)
[vupTests.c: 801] 0x401914: 8f05fffd 1w a1,-44(t8)
[vupTests.c: 802] 0x401918: 8f06fffd 1w a2,-40(t8)
[vupTests.c: 803] 0x40191c: 8f07ffdc 1w a3,-36(t8)
[vupTests.c: 804] 0x401920: 8f08ffee 1w t0,-32(t8)
[vupTests.c: 805] 0x401924: 8f09ffee 1w t1,-28(t8)
[vupTests.c: 806] 0x401928: 8f0affe8 1w t2,-24(t8)
[vupTests.c: 807] 0x40192c: 8f0bffe0 1w t3,-20(t8)
[vupTests.c: 808] 0x401930: 8f0cfff0 1w t4,-16(t8)
[vupTests.c: 809] 0x401934: 8f0dfffd 1w t5,-12(t8)
[vupTests.c: 810] 0x401938: 8f0effe0 1w t6,-8(t8)
[vupTests.c: 811] 0x40193c: 8f0ffff0 1w t7,-4(t8)
[vupTests.c: 812] 0x401940: 8f100000 1w s0,0(t8)
[vupTests.c: 799] 0x401944: 8e640000 1w a0,0(s3)
/* Header Byte Swap */

[vupTests.c: 817] 0x401954: 3082ff00 andi v0,a0,0xff00
[vupTests.c: 817] 0x401958: 00021200 sll v0,v0,8
[vupTests.c: 817] 0x40195c: 00041e00 sll v1,a0,24
[vupTests.c: 817] 0x401960: 00621821 addu v1,v1,v0
[vupTests.c: 817] 0x401964: 00961024 and v0,a0,s6
[vupTests.c: 817] 0x401968: 00021202 srl v0,v0,8
[vupTests.c: 817] 0x40196c: 00621821 addu v1,v1,v0
[vupTests.c: 817] 0x401970: 00041602 srl v0,a0,24
[vupTests.c: 817] 0x401974: 00622021 addu a0,v1,v0
[vupTests.c: 818] 0x401980: 3322ff00 andi v0,t9,0xff00
[vupTests.c: 818] 0x401984: 00021200 sll v0,v0,8
[vupTests.c: 818] 0x401988: 00191e00 sll v1,t9,24
[vupTests.c: 818] 0x40198c: 00621821 addu v1,v1,v0
[vupTests.c: 818] 0x401990: 03361024 and v0,t9,s6
[vupTests.c: 818] 0x401994: 00021202 srl v0,v0,8
[vupTests.c: 818] 0x401998: 00621821 addu v1,v1,v0
[vupTests.c: 818] 0x40199c: 00191602 srl a0,t9,24
[vupTests.c: 818] 0x401a8: 0062c821 addu t9,v1,v0

/***** Registers --> Cop Regs ***********/

[vupTests.c: 823] 0x401978: 44842000 mtc1 a0,$f4
[vupTests.c: 824] 0x4019ac: 44992800 mtc1 t9,$f5
[vupTests.c: 825] 0x401b0: 44853000 mtc1 a1,$f6
[vupTests.c: 826] 0x401b4: 44863800 mtc1 a2,$f7
[vupTests.c: 827] 0x401b8: 44874000 mtc1 a3,$f8
[vupTests.c: 828] 0x401bc: 44884800 mtc1 t0,$f9
[vupTests.c: 829] 0x401c0: 44895000 mtc1 t1,$f10
[vupTests.c: 830] 0x401c4: 448a6000 mtc1 t2,$f12
[vupTests.c: 831] 0x401c8: 448b6800 mtc1 t3,$f13
[vupTests.c: 832] 0x401cc: 448c7000 mtc1 t4,$f14
[vupTests.c: 833] 0x401d0: 448d8000 mtc1 t5,$f16
[vupTests.c: 834] 0x401d4: 448e9000 mtc1 t6,$f17
[vupTests.c: 835] 0x401d8: 448f0000 mtc1 t7,$f18
[vupTests.c: 836] 0x401dc: 44902000 mtc1 s0,$f4

141
Assembly Code: Cop2 Model

This is the inner loop of routine 'RxBurstCop2Cmd' which receives cells from the Coprocessor.

RxBurstCop2Cmd:

******/ Header: Cop Regs --> Registers ************/

[vupTests.c: 1092] 0x402044: 44042000 mfc1 a0,$f4
[vupTests.c: 1093] 0x402048: 44052800 mfc1 a1,$f5

******/ Header Byte Swap **********************

[vupTests.c: 1097] 0x40204c: 3082ff00 andi v0,a0,0xff00
[vupTests.c: 1097] 0x402050: 00021200 sll v0,v0,8
[vupTests.c: 1097] 0x402054: 00041e00 sll v1,a0,24
[vupTests.c: 1097] 0x402058: 00621821 addu v1,v1,v0
[vupTests.c: 1097] 0x40205c: 00691024 and v0,a0,t1
[vupTests.c: 1097] 0x402060: 00021202 srl v0,v0,8
[vupTests.c: 1097] 0x402064: 00621821 addu v1,v1,v0
[vupTests.c: 1097] 0x402068: 00041602 srl v0,a0,24
[vupTests.c: 1097] 0x40206c: 00622021 addu a0,v1,v0
[vupTests.c: 1098] 0x402070: 30a2ff00 andi v0,a1,0xff00
[vupTests.c: 1098] 0x402074: 00021200 sll v0,v0,8
[vupTests.c: 1098] 0x402078: 00051e00 sll v1,a1,24
[vupTests.c: 1098] 0x40207c: 00621821 addu v1,v1,v0
[vupTests.c: 1098] 0x402080: 00a91024 and v0,a1,t1
[vupTests.c: 1098] 0x402084: 00021202 srl v0,v0,8
[vupTests.c: 1098] 0x402088: 00621821 addu v1,v1,v0
[vupTests.c: 1098] 0x40208c: 00051602 srl v0,a1,24
[vupTests.c: 1098] 0x402090: 00622821 addu a1,v1,v0

******/ Registers/Cop Regs --> Memory ************/

[vupTests.c: 1102] 0x402094: acc40000 sw a0,0(a2)
[vupTests.c: 1103] 0x402098: acc50004 sw a1,4(a2)
[vupTests.c: 1104] 0x40209c: e4c60008 swc1 $f6,8(a2)
[vupTests.c: 1105] 0x4020a0: e4c7000c swc1 $f7,12(a2)
[vupTests.c: 1106] 0x4020a4: e4c80010 swc1 $f8,16(a2)
[vupTests.c: 1107] 0x4020a8: e4c90014 swc1 $f9,20(a2)

142
This is the inner loop of routine 'TxBurstCop2Cmd' which transmits cells to the Coprocessor.

TxBurstCop2Cmd:

/***** Header: Memory --> Registers ***********/

[vupTests.c: 894] 0x401b68: 8d650000 lw a1,0(t3)
[vupTests.c: 895] 0x401b70: 8d24ff00 lw a0,-48(t1)

/***** Header Byte Swap **********************/

[vupTests.c: 899] 0x401b74: 30a2ff00 andi v0,a1,0xff00
[vupTests.c: 899] 0x401b78: 00021200 sll v0,v0,8
[vupTests.c: 899] 0x401b7c: 00051e00 sll v1,a1,24
[vupTests.c: 899] 0x401b80: 00621821 addu v1,v1,v0
[vupTests.c: 899] 0x401b84: 00ad1024 and v0,a1,t5
[vupTests.c: 899] 0x401b88: 00021202 srl v0,v0,8
[vupTests.c: 899] 0x401b8c: 00621821 addu v1,v1,v0
[vupTests.c: 899] 0x401b90: 00051602 srl v0,a1,24
[vupTests.c: 899] 0x401b94: 00622821 addu a1,v1,v0
[vupTests.c: 900] 0x401b98: 3082ff00 andi v0,a0,0xff00
[vupTests.c: 900] 0x401b9c: 00021200 sll v0,v0,8
[vupTests.c: 900] 0x401ba0: 00041e00 sll v1,a0,24
[vupTests.c: 900] 0x401ba4: 00621821 addu v1,v1,v0
[vupTests.c: 900] 0x401ba8: 008d1024 and v0,a0,t5
[vupTests.c: 900] 0x401bac: 00021202 srl v0,v0,8
[vupTests.c: 900] 0x401bb0: 00621821 addu v1,v1,v0
[vupTests.c: 900] 0x401bb4: 00041602 srl v0,a0,24
[vupTests.c: 900] 0x401bb8: 00622021 addu a0,v1,v0

/***** Memory --> Cop Regs *************/

[vupTests.c: 901] 0x401bbc: c524ffcc lwci $f4,-52(t1)
[vupTests.c: 902] 0x401bc0: c525ffd0 lwc1 $f5,-48(t1)
[vupTests.c: 903] 0x401bc4: c526ffd4 lwc1 $f6,-44(t1)
[vupTests.c: 904] 0x401bc8: c527fffd lwc1 $f7,-40(t1)
[vupTests.c: 905] 0x401bcc: c528ffdc lwc1 $f8,-36(t1)
[vupTests.c: 906] 0x401bd0: c529ffe0 lwc1 $f9,-32(t1)
[vupTests.c: 907] 0x401bd4: c52affe4 lwc1 $f10,-28(t1)
[vupTests.c: 908] 0x401bd8: c52cffe8 lwc1 $f12,-24(t1)
[vupTests.c: 909] 0x401bdc: c52dffec lwc1 $f13,-20(t1)
[vupTests.c: 921] 0x401be0: c52efff0 lwc1 $f14,-16(t1)
[vupTests.c: 922] 0x401be4: c530fff4 lwc1 $f16,-12(t1)
[vupTests.c: 923] 0x401be8: c531fffe lwc1 $f17,-8(t1)
[vupTests.c: 924] 0x401bec: c532ffff lwc1 $f18,-4(t1)
[vupTests.c: 925] 0x401bf0: c5240000 lwc1 $f4,0(t1)
Assembly Code: Cop3 Model

/* ******************************************************/
This is the inner loop of routine 'RxBurstCop3Cmd' which receives
cells from the Coprocessor.
/* ******************************************************/

RxBurstCop3Cmd:

/*******/ 'Header' Byte Swap ***********************

[vupTests.c: 1193] 0x402244: 3222ff00 andi v0,s1,0xff00
[vupTests.c: 1193] 0x402248: 00021200 sl1 v0,v0,8
[vupTests.c: 1193] 0x40224c: 00111e00 sl1 v1,s1,24
[vupTests.c: 1193] 0x402250: 00621821 addu v1,v1,v0
[vupTests.c: 1193] 0x402254: 02271024 and v0,s1,a3
[vupTests.c: 1193] 0x402258: 00021202 srl v0,v0,8
[vupTests.c: 1193] 0x40225c: 00621821 addu v1,v1,v0
[vupTests.c: 1193] 0x402260: 00111602 srl v0,s1,24
[vupTests.c: 1193] 0x402264: 00628821 addu s1,v1,v0
[vupTests.c: 1194] 0x40226c: 3242ff00 andi v0,s2,0xff00
[vupTests.c: 1194] 0x402270: 00021200 sl1 v0,v0,8
[vupTests.c: 1194] 0x402274: 00121e00 sl1 v1,s2,24
[vupTests.c: 1194] 0x402278: 00621821 addu v1,v1,v0
[vupTests.c: 1194] 0x40227c: 02471024 and v0,s2,a3
[vupTests.c: 1194] 0x402280: 00021202 srl v0,v0,8
[vupTests.c: 1194] 0x402284: 00621821 addu v1,v1,v0
[vupTests.c: 1194] 0x402288: 00121602 srl v0,s2,24
[vupTests.c: 1194] 0x40228c: 00629021 addu s2,v1,v0

/******* Cop Regs --> Memory ***********************

[vupTests.c: 1198] 0x402268: e4840000 swc1 $f4,0(a0)
[vupTests.c: 1199] 0x402290: e4850004 swc1 $f5,4(a0)
[vupTests.c: 1200] 0x402294: e4860008 swc1 $f6,8(a0)
[vupTests.c: 1201] 0x402298: e487000c swc1 $f7,12(a0)
[vupTests.c: 1202] 0x40229c: e4880010 swc1 $f8,16(a0)
[vupTests.c: 1203] 0x4022a0: e4890014 swc1 $f9,20(a0)
[vupTests.c: 1204] 0x4022a4: e48a0018 swc1 $fa,24(a0)
[vupTests.c: 1205] 0x4022a8: e48c001c swc1 $fc,28(a0)
[vupTests.c: 1206] 0x4022ac: e48d0020 swc1 $fd,32(a0)
[vupTests.c: 1207] 0x4022b0: e48e0024 swc1 $fe,36(a0)
[vupTests.c: 1208] 0x4022b4: e4900028 swc1 $f16,4^+0)
This is the inner loop of routine 'TxBurstCop3Cmd' which transmits cells to the Coprocessor.

****** 'Header' Byte Swap **************

****** Memory --> Cop Regs ***************
[vupTests.c: 1021] 0x401df8: c531fff8 lwc1 $f17,-8(t1)
[vupTests.c: 1022] 0x401dfc: c532ffff lwc1 $f18,−4(t1)
[vupTests.c: 1023] 0x401e00: c5240000 lwc1 $f4,0(t1)
Assembly Code: Cop4 Model

/*****************************************************************************************
This is the inner loop of routine 'RxBurstCop4Cmd' which receives
cells from the Coprocessor.
*******************************************************************************************/

RxBurstCop4Cmd:

/*** Cop Regs --> Registers ***************/

[vupTests.c: 1286] 0x40243c: 44022000 mfc1 v0,$f4
[vupTests.c: 1287] 0x402440: 44032800 mfc1 v1,$f5
[vupTests.c: 1288] 0x402444: 44043000 mfc1 a0,$f6
[vupTests.c: 1289] 0x402448: 44053800 mrc1 a1,$f7
[vupTests.c: 1290] 0x40244c: 44064000 mfc1 a2,$f8
[vupTests.c: 1291] 0x402450: 44074800 mfc1 a3,$f9
[vupTests.c: 1292] 0x402454: 44085000 mfc1 t0,$f10
[vupTests.c: 1293] 0x402458: 44096000 mfc1 t1,$f12
[vupTests.c: 1294] 0x40245c: 440a6800 mfc1 t2,$f13
[vupTests.c: 1295] 0x402460: 440b7000 mfc1 t3,$f14
[vupTests.c: 1296] 0x402464: 440c8000 mfc1 t4,$f16
[vupTests.c: 1297] 0x402468: 440d8800 mfc1 t5,$f17
[vupTests.c: 1298] 0x40246c: 440e9000 mfc1 t6,$f18
[vupTests.c: 1299] 0x402470: 440f2000 mfc1 t7,$f4

/*** Registers --> Memory ***************/

[vupTests.c: 1308] 0x402478: ae220000 sw v0,0(s1)
[vupTests.c: 1309] 0x40247c: af03ff0 sw v1,-48(t8)
[vupTests.c: 1310] 0x402480: af04ff4 sw a0,-44(t8)
[vupTests.c: 1311] 0x402484: af05ff8 sw a1,-40(t8)
[vupTests.c: 1312] 0x402488: af06ffdc sw a2,-36(t8)
[vupTests.c: 1313] 0x40248c: af07ffe0 sw a3,-32(t8)
[vupTests.c: 1314] 0x402490: af08ffe4 sw t0,-28(t8)
[vupTests.c: 1315] 0x402494: af09ffe8 sw t1,-24(t8)
[vupTests.c: 1316] 0x402498: af0affec sw t2,-20(t8)
[vupTests.c: 1317] 0x40249c: af0bfff0 sw t3,-16(t8)
[vupTests.c: 1318] 0x4024a0: af0cffff sw t4,-12(t8)
[vupTests.c: 1319] 0x4024a4: af0dffff sw t5,-8(t8)
[vupTests.c: 1320] 0x4024a8: af0effec sw t6,-4(t8)
[vupTests.c: 1321] 0x4024ac: af0f0000 sw t7,0(t8)
This is the inner loop of routine 'TxBurstCop4Cmd' which transmits cells to the Coprocessor.

TxBurstCop4Cmd:

/***** Memory --> Registers ***********/

[vupTests.c: 1086] 0x401f94: 8f03ff0d lw v1,-48(t8)
[vupTests.c: 1089] 0x401f98: 8f04ff0d lw a0,-44(t8)
[vupTests.c: 1090] 0x401f9c: 8f05ff0d lw a1,-40(t8)
[vupTests.c: 1091] 0x401fa0: 8f06ff0c lw a2,-36(t8)
[vupTests.c: 1092] 0x401fa4: 8f07ffe0 lw a3,-32(t8)
[vupTests.c: 1093] 0x401fa8: 8f08ffe4 lw t0,-28(t8)
[vupTests.c: 1094] 0x401fac: 8f09ffe8 lw t1,-24(t8)
[vupTests.c: 1095] 0x401fb0: 8f0affec lw t2,-20(t8)
[vupTests.c: 1096] 0x401fb4: 8f0bfffc lw t3,-16(t8)
[vupTests.c: 1097] 0x401fb8: 8f0cff4 lw t4,-12(t8)
[vupTests.c: 1098] 0x401fbc: 8f0dfff8 lw t5,-8(t8)
[vupTests.c: 1099] 0x401fc0: 8f0efffc lw t6,-4(t8)
[vupTests.c: 1100] 0x401fc4: 8f0f0000 lw t7,0(t8)
[vupTests.c: 1087] 0x401fc8: 8e220000 lw v0,0(s1)

/***** Registers --> Cop Regs ***********/

[vupTests.c: 1111] 0x401fd0: 44822000 mtc1 v0,$f4
[vupTests.c: 1112] 0x401fe4: 44832800 mtc1 v1,$f5
[vupTests.c: 1113] 0x401fe8: 44843000 mtc1 a0,$f6
[vupTests.c: 1114] 0x401fec: 44853800 mtc1 a1,$f7
[vupTests.c: 1115] 0x401ff0: 44864000 mtc1 a2,$f8
[vupTests.c: 1116] 0x401ff4: 44874800 mtc1 a3,$f9
[vupTests.c: 1117] 0x401ff8: 44885000 mtc1 t0,$f10
[vupTests.c: 1118] 0x401ffc: 44896000 mtc1 t1,$f12
[vupTests.c: 1119] 0x402000: 448a6800 mtc1 t2,$f13
[vupTests.c: 1120] 0x402004: 448b7000 mtc1 t3,$f14
[vupTests.c: 1121] 0x402008: 448c8000 mtc1 t4,$f16
[vupTests.c: 1122] 0x40200c: 448d8800 mtc1 t5,$f17
[vupTests.c: 1123] 0x402010: 448e9000 mtc1 t6,$f18
[vupTests.c: 1124] 0x402014: 448f2000 mtc1 t7,$f4
Assembly Code: Cop5 Model

/************************************************************
This is the inner loop of routine 'RxBurstCop5Cmd' which receives
cells from the Coprocessor.
************************************************************/

RxBurstCop5Cmd:

/********************************** Header: Cop Regs --> Registers ************/

[vupTests.c: 1381] 0x40261c: 44022000 mfc1 v0,$f4
[vupTests.c: 1382] 0x402620: 44032800 mfc1 v1,$f5

/********************************** Registers/Cop Regs --> Memory ************/

[vupTests.c: 1383] 0x402624: ac820000 sw v0,0(a0)
[vupTests.c: 1384] 0x402628: ac830004 sw v1,4(a0)
[vupTests.c: 1385] 0x40262c: e4860008 swc1 $f6,8(a0)
[vupTests.c: 1386] 0x402630: e487000c swc1 $f7,12(a0)
[vupTests.c: 1387] 0x402634: e4880010 swc1 $f8,16(a0)
[vupTests.c: 1388] 0x402638: e4890014 swc1 $f9,20(a0)
[vupTests.c: 1389] 0x40263c: e48a0018 swc1 $f10,24(a0)
[vupTests.c: 1390] 0x402640: e48c001c swc1 $f12,28(a0)
[vupTests.c: 1391] 0x402644: e48d0020 swc1 $f13,32(a0)
[vupTests.c: 1392] 0x402648: e48e0024 swc1 $f14,36(a0)
[vupTests.c: 1393] 0x40264c: e4900028 swc1 $f16,40(a0)
[vupTests.c: 1394] 0x402650: e491002c swc1 $f17,44(a0)
[vupTests.c: 1395] 0x402654: e4920030 swc1 $f18,48(a0)
[vupTests.c: 1396] 0x402660: e4840034 swc1 $f4,52(a0)

/************************************************************
This is the inner loop of routine 'TxBurstCop5Cmd' which transmits
cells to the Coprocessor.
************************************************************/

TxBurstCop5Cmd:

/********************************** Header: Memory --> Registers ************/

[vupTests.c: 1182] 0x402190: 8d22ffcc lw v0,-52(t1)
[vupTests.c: 1183] 0x402194: 8d23ffd0 lw v1,-48(t1)
Assembly Code: Cop6 Model

/*************************************************************************
This is the inner loop of routine 'RxBurstCop6Cmd' which receives
   cells from the Coprocessor.
***************************************************************************/

RxBurstCop6Cmd:

    /***********************************************************************/
    /* Cop Regs --> Memory *******************************************************************/

[vupTests.c: 1477] 0x4027cc: e5640000 swc1 $f4,0(t3)
[vupTests.c: 1478] 0x4027d0: e5650004 swc1 $f5,4(t3)
[vupTests.c: 1479] 0x4027d4: e5660008 swc1 $f6,8(t3)
[vupTests.c: 1480] 0x4027d8: e567000c swc1 $f7,12(t3)
[vupTests.c: 1481] 0x4027dc: e5680010 swc1 $f8,16(t3)
[vupTests.c: 1482] 0x4027e0: e5690014 swc1 $f9,20(t3)
[vupTests.c: 1483] 0x4027e4: e56a0018 swc1 $f10,24(t3)
[vupTests.c: 1499] 0x4027ec: e56c001c swc1 $f12,28(t3)
[vupTests.c: 1500] 0x4027f0: e56d0020 swc1 $f13,32(t3)
[vupTests.c: 1501] 0x4027f4: e56e0024 swc1 $f14,36(t3)
[vupTests.c: 1502] 0x4027f8: e5700028 swc1 $f16,40(t3)
[vupTests.c: 1503] 0x4027fc: e571002c swc1 $f17,44(t3)
[vupTests.c: 1504] 0x402800: e5720030 swc1 $f18,48(t3)
[vupTests.c: 1505] 0x402804: e5640034 swc1 $f4,52(t3)

/*************************************************************************
This is the inner loop of routine 'TxBurstCop6Cmd' which transmits
cells to the Coprocessor.
***************************************************************************/

TxBurstCop6Cmd:

    /***********************************************************************/
    /* Memory --> Cop Regs *******************************************************************/

[vupTests.c: 1278] 0x402358: c524ffcc lwc1 $f4,-52(t1)
[vupTests.c: 1279] 0x40235c: c525ffd0 lwc1 $f5,-48(t1)
[vupTests.c: 1280] 0x402360: c526ffd4 lwc1 $f6,-44(t1)
[vupTests.c: 1281] 0x402364: c527ffd8 lwc1 $f7,-40(t1)
[vupTests.c: 1282] 0x402368: c528ffdc lwc1 $f8,-36(t1)
[vupTests.c: 1283] 0x40236c: c529ffe0 lwc1 $f9,-32(t1)
[vupTests.c: 1277] 0x402370: c52affe4 lwc1 $f10,-28(t1)
[vupTests.c: 1301] 0x402374: c52cffe8 lwc1 $f12,-24(t1)
[vupTests.c: 1302] 0x402378: c52dffec lwc1 $f13,-20(t1)
[vupTests.c: 1303] 0x40237c: c52eff0 lwc1 $f14,-16(t1)
[vupTests.c: 1304] 0x402380: c530fff4 lwc1 $f16,-12(t1)
[vupTests.c: 1305] 0x402384: c531ffff lwc1 $f17,-8(t1)
[vupTests.c: 1306] 0x402388: c532fffc lwc1 $f18,-4(t1)
[vupTests.c: 1307] 0x40238c: c5240000 lwc1 $f4,0(t1)
Bibliography


[34] LSI Logic Corporation, Milpitas, CA. *SPARC Architecture Manual (Version 7)*.


157