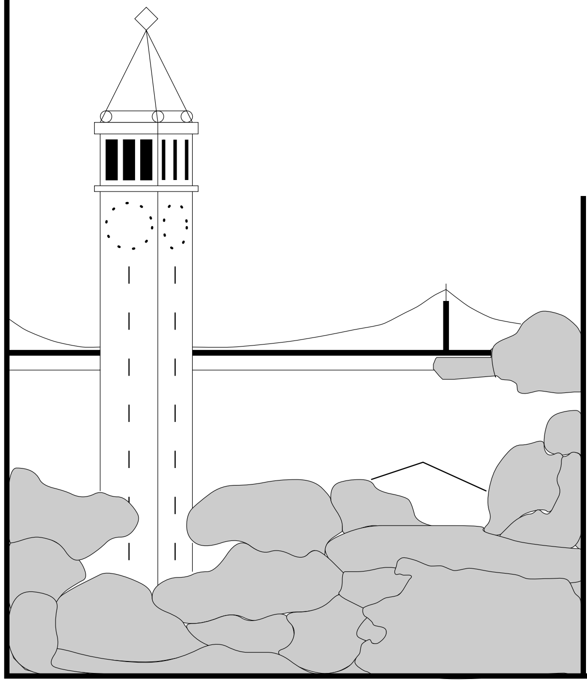
Network Interface Specification for the T1 Microprocessor

Timothy J. Callahan

timothyc@CS.Berkeley.EDU



Report No. UCB//CSD-94-823

May 1994

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Network Interface Specification for the T1 Microprocessor [*][†]

Timothy J. Callahan
timothyc@CS.Berkeley.EDU

May 1994

Abstract

The overall performance of a multicomputer depends heavily on the interface between the software and the communication hardware. As pointed out in von Eicken's thesis, this *communication architecture* should be *versatile* in that it be able to support a variety of different communication models, including shared memory, dataflow, and send&receive; it should support an *efficient* implementation of each model; and it should be *incremental* in that it shouldn't interfere with the computation performance of the processor.

Active Messages communication architectures have been shown to satisfy these criteria. Software implementations of Active Messages have reduced communication overhead by over an order of magnitude to near the minimum possible given existing hardware. This project takes the next step and defines a hardware implementation of an Active Message communication architecture, resulting in another order of magnitude reduction in communication overhead.

The Active Message communication architecture defined in this report is an extension to the MIPS-II instruction set architecture. The resulting architecture features data transfer directly to/from processor registers, hardware dispatch directly to Active Message handlers (along with limited context preservation), automatic atomicity of handlers, cheap synchronization operations, and hardware support for multicast.

[*]This report was submitted in partial satisfaction of the requirements for the author's M.S. degree.

[†]This research is supported in part by a National Science Foundation Graduate Fellowship, Grant URI-N00014-92-J-1672 from the Office of Naval Research, and National Science Foundation Infrastructure Grant number CDA-8722788. Additional support comes from the International Computer Science Institute.

Contents

1 Introduction

Continuing improvements in VLSI technology, apparent in the outstanding performance/price ratios for microprocessors, have spurred the development of massively parallel processors, which are constructed by taking dozens to thousands of microprocessors and connecting them with an interprocessor communication network. While the network represents additional complexity and cost, solutions have been engineered that result in high-performance machines that for many applications provide performance/price ratios superior to those of traditional supercomputers.

The easiest way to build a massively parallel system is to physically distribute the memory among the processors. Essentially the nodes are similar to a modern workstations, with a dedicated, high-bandwidth, low-latency network connecting them. Such distributed memory systems grow to large size more gracefully than systems where all the memory is located in one central location.

The software running on each node must have some way of accessing the communication network. This interface, as seen by the compiler or assembly language programmer, is called the *communication architecture*[1] (see Figure 1). The functions defined by the communication architecture are implemented by some combination of hardware, kernel code, run-time systems, and library routines.

The communication architecture is distinguished from the *communication micro-architecture*, which is composed of the implementation-specific details of the data transfer functional units, interconnect structure, and network operation.

Application Layer
Communication Model
Communication Architecture
Communication Micro-architecture
Communication Hardware

Figure 1: Communication Layers

The functionality provided by the communication architecture is typically used by a parallel language (or extensions to a sequential language) to provide a *communication model* to the application programmer. An example of a communication model is shared memory, in which the application programmer sees a global address space. In this model communication is performed by one processor writing to a memory location and another processor reading from that location.

[1]The definition of this term, as well as most of the ideas contained in this section, are taken from Thorsten von Eicken's Ph.D. thesis, [vE93].

Traditional communication architectures have often been overambitious and have tried to do too much; they try to provide functionality that belongs more appropriately in the communication model layer. By fixing a specific model in the communication architecture, a *semantic clash* is risked. A semantic clash occurs when an interface provides a certain high-level function, but the client needs slightly different functionality and therefore cannot use the provided function. If the interface succeeds in providing truly general high-level functions, usually there is a performance penalty for clients that don't need all the functionality.

For example, consider a communication architecture that provides only a shared-memory abstraction to higher levels, but is built on top of a packet-based interconnection network. The [KJA+93] study found that many applications require communication operations that are less efficiently implemented with shared-memory than with message-passing. For these operations, the shared-memory layer is a hindrance; the messaging efficiency of the underlying hardware has been hidden. A semantic clash has occurred: the abstraction provided does not match exactly what the application needs, and moreover prevents the efficient construction of the abstraction the application does need.

Another example is the general send&receive mechanism, which typically requires the use of buffers in the kernel address space. The kernel buffers are required because a send may be performed before the user on the receiving node has allocated its own buffer for the data. However, applications that have pre-allocated the buffers still pay the cost of copying data between buffers in kernel and user space. Here again, the general mechanism gets in the way of efficient use of the resources.

These examples give insight into which design criteria should be used for judging communication architectures. A well-designed communication architecture should be *versatile* enough so that it can support many different communication models; it should be *efficient* in that the operations of each communication model should map efficiently to the primitives provided; and it should be *incremental* in that it does not disrupt the versatility and efficiency of the sequential architecture.

Active Messages communication architectures [vE93], described in the next section, have been show to satisfy these criteria. Active Messages exposes the efficiency of the underlying communication micro-architecture as much as possible, and doesn't make the mistake of trying to implement functions that really belong in a communication model.

Active Message communication architectures implemented in software on top of existing communication micro-architectures have been successful in reducing the software overhead of communication by more than an order of magnitude. It seems unlikely that any further improvement can be made as long as the network interface is implemented as external circuitry complementing a standard microprocessor. In fact, if the current trend in microprocessor design continues, first- and second-level caches will increasingly be integrated on-chip; an external network interface will be pushed logically farther from the CPU core, and the cost of communication will increase. The most direct way to counteract this trend is to integrate Active Messages with the processor instruction set architecture.

In this project, an Active Messages communication architecture implementation is defined competely in the actual instruction set of a microprocessor, *i.e.* zero or minimal kernel, run-time, or library code is required to complete the Active Message layer. Perhaps this could also be interpreted as the design of a communication micro-architecture such that an Active Message communication architecture maps to it in a direct, almost trivial, way. The resulting network interface exhibits very low overhead for all communication operations, including sending messages, receiving messages and dispatching to the appropriate handler, and disabling message reception to form critical sections.

The rest of this report is organized as follows. Section 2 describes why Active Messages makes a good communication architecture. Section 3 describes and justifies the design decisions made in this implementation of Active Messages. In Section 4 this implementation is compared to previous research involving the integration of a network interface into a CPU. Section 5 describes ongoing research related to the implementation of the architecture described in this report. A brief conclusion is found in Section 6, which is followed by the bibliography, and finally the network interface specification in detail is found in Appendix A.

2 Active Messages

2.1 The Main Idea

"Active Messages" is a philosophy, or class of communication architectures, analogous to the term "RISC" for a class of computational architectures. To quote from [vECGS92] (see also Figure 2),

> Active Messages is an asynchronous communication mechanism intended to expose the full hardware flexibility and performance of modern interconnection networks. The underlying idea is simple: each message contains at its head the address of a user-level handler which is executed on message arrival with the message body as argument. The role of the handler is to get the message out of the network and into the computation ongoing on the processing node. The handler must execute quickly and to completion.

For some parallel operations, enough may be known about the communication patterns that increased performance can result from breaking some of the Active Message conventions — for example, by performing some real computation in an Active Message handler. While such use of the architecture described in this report is possible, the design has been optimized for conventional Active Messages.

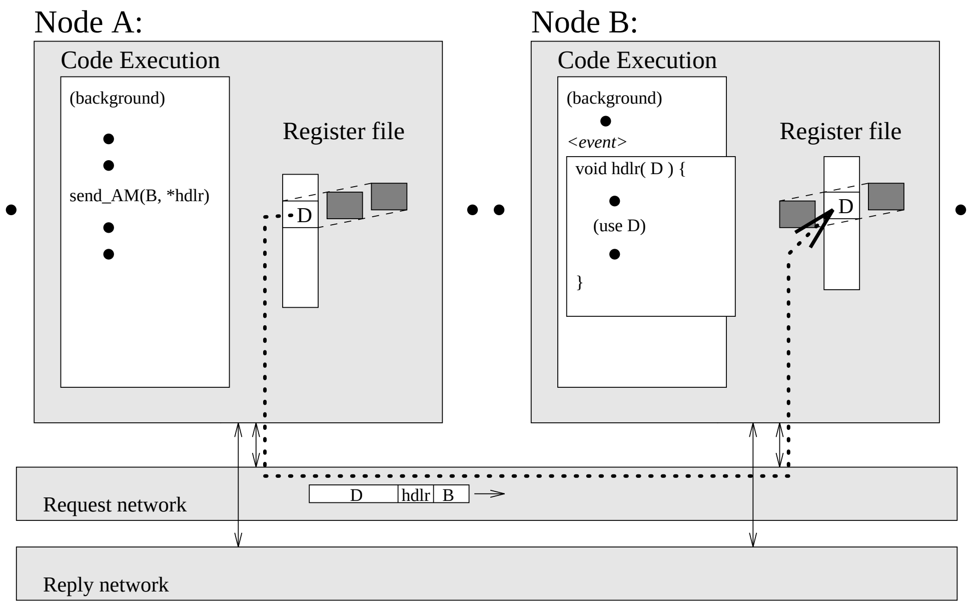


Figure 2: Active Message in the CNS-1. The primary computation on Node B is asynchronously interrupted, and control is transferred to the address contained in the first word of the message (*hdlr*). The code at *hdlr* takes care of integrating the message data (*D*) with the primary computation.

2.2 Support of Communication Models

An Active Messages communication architecture, just like a RISC instruction set, is not meant for direct use by the application programmer. Rather, Active Messages is meant to be used by a parallel language or communication library to provide a communication model to the application programmer. Active Messages has been shown to efficiently support most common communication models. This section contains examples of how Active Messages can support both shared memory and dataflow models, and point out where hardware support for Active Messages is especially useful. The goal of this subsection is to justify hardware support for Active Messages, based upon two points:

1. that performance will benefit significantly from hardware support of Active Messages, and
2. that the Active Message communication architecture can support most communication models efficiently, and thus there is no loss of flexibility by implementing it in hardware.

Earlier examples demonstrated that trying to support any more functionality than that provided by Active Messages in hardware is often a mistake. The conclusion to be drawn is that Active Messages is an optimal communication interface to fix between hardware and software.

2.2.1 Shared Memory and Split-C

Split-C [CDG+93], a parallel extension to C developed at UC Berkeley, is a good example of the use of Active Messages to support a shared-memory programming model. The description here is based on the implementation of Split-C for Thinking Machines' CM-5, which is similar to the version envisioned for the CNS-1. However, Split-C has been ported to widely varying architectures with accordingly varying implementations, as described in [Lun94].

In the Split-C model, there is one persistent thread per processor. Split-C uses a global, two-dimensional address space; each global address is a <*processor, offset*> pair. Global pointers can be dereferenced just as normal pointers in C can, so that `*gp = lv` copies the contents of local variable `lv` to the location specified by the global pointer `gp`. The reverse operation is performed by `lv = *gp`. Normal (local) pointers are still available and should be used for work local to a processor, since accesses through global pointers will incur an extra cost to check whether the location is local or remote, even if the location is local.

Split-C also allows for split-phase assignments as shown in later examples, allowing the overlap of global operations with local computation. Simple counters are used to deduce when all operations have completed for synchronization purposes.

Split-C is implemented as a modification of the GCC compiler along with a runtime system. The runtime system for Split-C consists of a collection of library routines and Active Message handlers, contained in Libsplit-C.

For example, consider Processor A performing a *put* (an asynchronous, acknowledged write) to a location contained in Processor B's memory. During parsing the Split-C compiler replaces the *put* syntax, `*gv := lv`, with an Active Message send. Thus Processor A actually sends a *put* Active Message containing the value and address to Processor B. Processor A returns to computation immediately. The *put* handler on Processor B writes the value contained in the message to the specified location, and then sends a *put reply* message back to Processor A. On Processor A the *put reply* handler for this message increments a counter to acknowledge the completion of the *put*. Some time later Processor A may execute a *sync* call, which busy-waits on the counter until all *put*s have been acknowledged (non-blocking checks of the counter are also possible).

Support for more complex shared memory operations is also straightforward using Active Messages. A fetch-and-add operation, for example, is implemented using a *fetch-and-add* handler on the node containing the counter (see Figures 3, 4). Note that since Active Message handlers execute to completion, no special effort is required to ensure

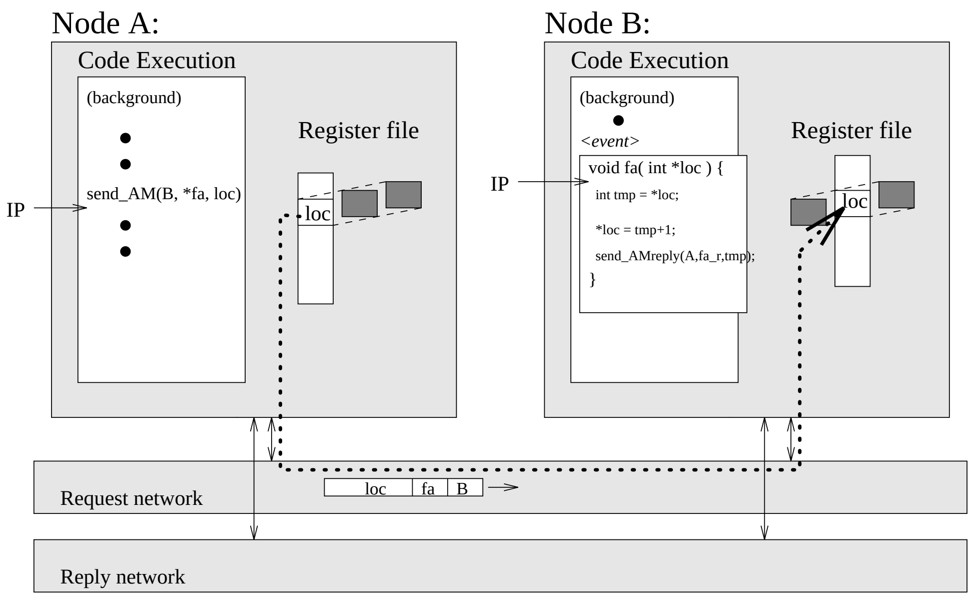


Figure 3: Fetch-and-Add, Part One. Node A has sent a request message containing both a pointer to the fetch-and-add request handler (*fa*) and a pointer to the desired counter on Node B (*loc*). In the picture, Node B is in the process of handling the request but has not replied yet.

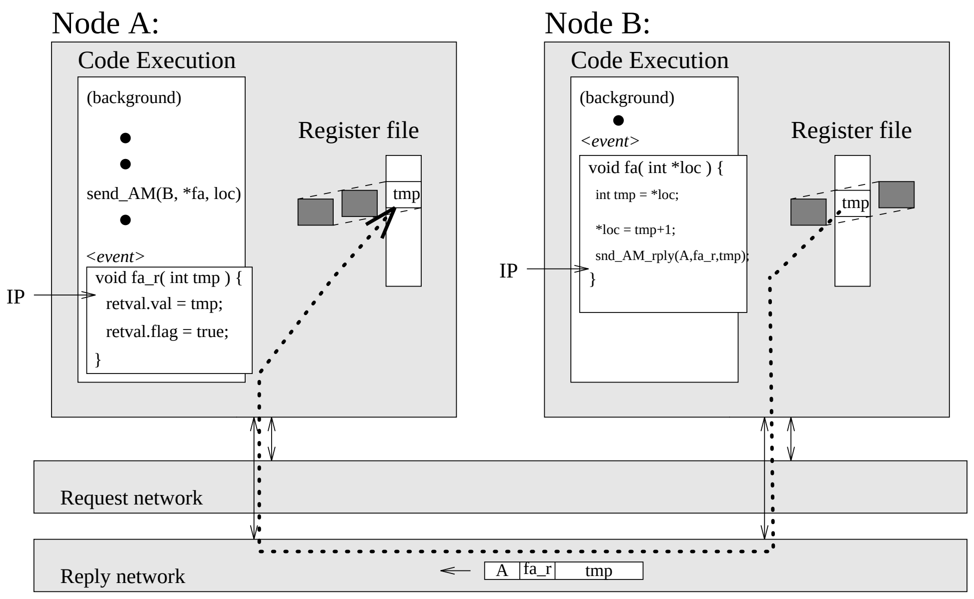


Figure 4: Fetch-and-Add, Part Two. The reply message containing the value from Node B (*tmp*) has been received at Node A, where a reply handler has been invoked. The reply handler stores the value and sets a flag. After the reply handler has completed, the primary computation on Node A notices that the flag has been set, and then can use the returned value.

the atomicity of the fetch-and-add operation.

In addition to these integer or floating point variable remote memory operations, Split-C also provides bulk transfer primitives, which are optimized for medium to large data transfers.

Benefit of Hardware Support

In comparison to software implementations, the low overhead of hardware dispatch to the Active Message handler greatly reduces the number of cycles consumed for handling messages, leaving more cycles for the primary computation. (For comparison, the total overhead using CMAM on the CM-5 for sending, receiving, and dispatching to handler for a 5-word message is approximately 100 cycles, while with the T1 the total overhead is expected to be less than 10 cycles for any message size up to the maximum of 37 words.) A related benefit is that round-trip latencies are greatly reduced, meaning that it will be easier for the Split-C programmer to find computation to overlap with remote memory accesses. Even when applications need fine-grained data sharing, hardware support of Active Messages combined with Split-C's exposure of local vs. global references will allow performance to approach that possible with a hardware-implemented shared-memory model[2], while allowing much greater flexibility. Coarse-grain computations also benefit from hardware support; the bulk transfer primitives will be able to use vector Active Messages (described in the next section), which can transfer up to 32 words of data with the same small overhead.

Another advantage is that a Split-C compiler knowledgeable of the hardware support for Active Messages will be able to make optimizations that cannot typically be made if the communication operations are encapsulated in library routines.

2.2.2 Message-Driven Models

Active Messages running on conventional microprocessors has also been shown to support message-driven computation models at least as efficiently as processors designed specifically to support those models. This is made possible by the use of the Threaded Abstract Machine (TAM) model as a compilation target for dataflow-style languages [CGSvE93]. TAM exposes the scheduling of threads to the compiler. Related threads in the same activation frame share the same processor state. By scheduling related threads consecutively, the overhead of thread switching is greatly reduced, while the benefits of a large register file and RISC instruction set are available to the threads. Such intelligent scheduling would likely not be possible if the general thread scheduler were built into the hardware.

[2]This will not be true in cases where coherent caches are very beneficial, such as when data is written once and then read several times by another random processor. Software caching can be useful in some cases, but will not give the same level of performance as would be achieved with hardware cache coherence.

In the TAM implementation, Active Messages are used for data movement and the associated synchronization typical of dataflow. The tailored Active Message handler stores the data at a specified offset in a specified frame, records this action in a counter, and possibly performs a scheduling operations if a thread becomes enabled. The Active Message handler itself does not perform the computation, and thus does not require much context preserving overhead. In TAM, events affecting scheduling occur often, but do not cost much. Scheduling actions (changing threads) occur less often but still do not cost much, and finally frame switches occur relatively rarely.

Benefit of Hardware Support

The paper [SGS$^+$93] evaluated support mechanisms for TAM and came to the conclusions that hardware support can significantly improve performance as long as the interaction of all components is considered. Specifically, integrating the network interface with the processor register file and supporting fast dispatch to user-level handlers can both significantly reduce communication overhead (a first-order effect for fine-grained computation), as long as the operations for synchronization and atomicity are similarly efficient. This lends support to the claim that thoughtful hardware support for Active Messages will lead to significant performance benefits for fine-grained computation.

2.2.3 Conclusion

In this subsection a brief description has been given of how Active Messages can efficiently support two of the common communication models. By providing a clean, simple mechanism rather than attempting a complete solution, Active Messages allows each programming model to use the native communication resources in the way best suited for it. This combination of efficiency and flexibility make Active Messages an optimal platform to fix in hardware. Putting any more functionality in hardware would reduce flexibility and risk a semantic clash, without much performance benefit. Putting less functionality in hardware would reduce performance without giving a significant gain in flexibility.

2.3 Four Key Issues

As mentioned above, Active Messages describes a *class* of communication architectures. In fully describing a specific communication architecture implementation based on Active Messages, the designer must address four key issues concerning the relationship to the underlying communication micro-architecture:

- **Data Transfer** into and out of the network
- **Synchronization** between message arrival and computation

- how to deal with **Send Failure** due to network congestion, without risking deadlock
- **Network Virtualization** – how to share the network between multiple user processes

If the Active Message communication architecture is implemented as a software layer on top of existing hardware, the resolution of these issues is heavily influenced by the communication micro-architecture of the machine. In the case here, there exists the luxury that design of the communication micro-architecture can be guided by the desired Active Message communication architecture that it will support. This gives the freedom to make choices regarding the four main issues based on overall performance, rather than having to contort the Active Messages communication architecture to fit existing hardware. In the next section, the choices made will be examined in detail.

3 Implementation Choices

In designing a communication architecture based on Active Messages, there are a number of key issues to be addressed. Von Eicken lists them as data transfer, synchronization, send failure, and network virtualization. The choices made in the handling of these issues, along with additional implementation choices regarding the dispatch mechanism and multicast support, are described and justified below. First, however, a brief summary of how the communication architecture fits into the MIPS-II instruction set architecture will be given, followed by a short description of the context of this project.

3.1 Overview

This projects defines a communication architecture to extend the MIPS-II instruction set architecture [Kan88]. The network interface is logically part of a coprocessor. In the specification in the appendix, coprocessor 2 (COP2) is used; this is an arbitrary choice and may be changed. Other specific details such as instruction encoding and network interface control register numbering may also change.

New instructions for sending messages and returning from handlers have been added in the coprocessor opcode space. Existing MIPS instructions are used for transfers to and from network interface control registers, and for branching on the coprocessor condition flag. The MIPS-II ISA specification states that coprocessor operations should not affect CPU state, and that data transfer between the CPU and the coprocessor should occur only through MFCz/MTCz instructions. These specifications are not strictly followed; violations include:

- CPU control flow is asynchronously preempted due to the arrivals of messages and other *events*.
- Arriving data is accessible in CPU registers, and outgoing data is taken directly from CPU registers.
- Some of the CPU registers have been triplicated; which copy is visible to the processor depends on the context: primary computation, request handler, or reply handler.

To give a very brief summary of how inter-processor communication works, a processor's SEND instruction constructs and sends a packet containing (i) a header indicating the destination node, (ii) a pointer to the desired handler for the message on the destination processor, and (iii) data from processor registers. At the destination node, the arrival of the packet causes an inlet event, which (all in hardware) saves a portion of the current state of the processor, places the data from the packet into the appropriate processor registers, and starts execution of the handler specified in the packet. The handler is ended by an HRET instruction, which returns the processor to its state before preemption. Message reception can be disabled by clearing an event enable bit in a coprocessor control register, allowing cheap construction of critical sections. While message reception is disabled, messages will back up in the network interface and eventually back up in the network.

3.2 Context

This project is one small component of the construction of the CNS-1 Connectionist Network Supercomputer, a large-scale multicomputer being constructed with the target application of neural network algorithms.

The building block of the CNS-1 is the T1 microprocessor, an implementation of the Torrent architecture. This architecture integrates a MIPS scalar processor core along with a fixed-point vector unit, the network interface described in this report, and a mesh router[3] The T1 processor has 128b-wide path to the local memory on the CNS-1 node. The memory will be high-performance DRAM, such as Rambus or Synchronous-DRAM, providing a large memory bandwidth.

The topology of the CNS-1 data network is a two-dimensional mesh that wraps around in one of the dimensions. This topology maps directly to its physical packaging, a cylindrical

[3]Although the T1 has integrated the router to achieve better density on the CNS-1 node, a alternative implementation would have a single network link leaving the processor, which would connect to a separate routing chip, as with the *T project [PBGB93]. This would allow the same processor to be used with a variety of different network topologies and technologies, and also reduce the main chip's pin requirements for the network, with only a small latency penalty.

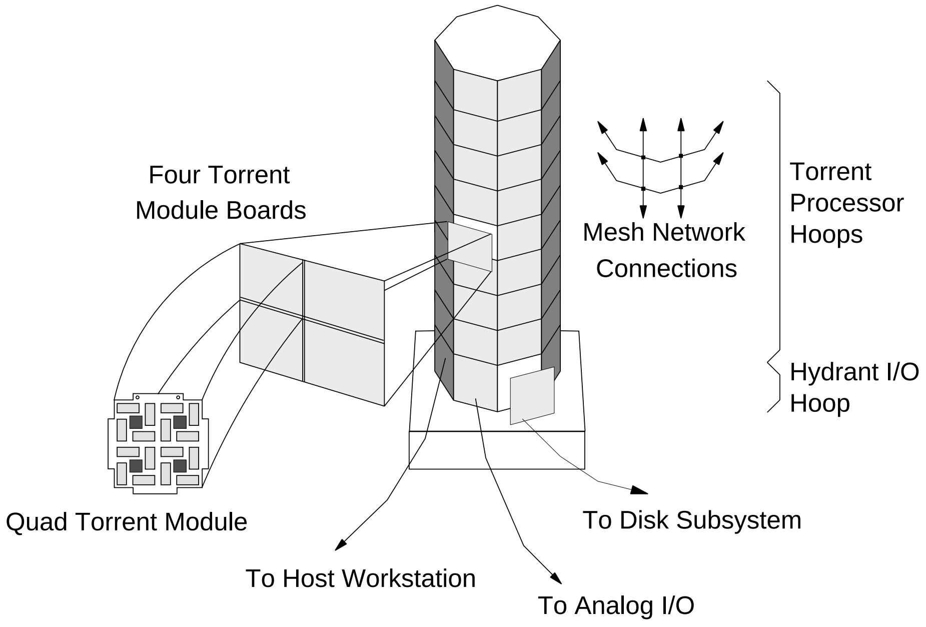


Figure 5: CNS-1

tower (see Figure 5). I/O to the computing nodes is performed through special interface nodes along the bottom of the cylinder. In addition to the data network, there is a diagnostic network which is accessible to the kernel only.

Although the CNS-1 is being built with the main goal of performing neural network algorithms at supercomputer speeds, it should find a broad range of applications. The T1 processor contains a standard MIPS processor core (although some software emulation is required for IEEE floating point), and the network interface specified in this report is general purpose. Since this design will be implemented as a real machine, the advantages and disadvantages of this communication architecture can be quantitatively compared against others in the context of real applications and operating systems, providing new insight for parallel systems design.

3.3 Data Transfer

One of the major points of an Active Message implementation is the data placement – where it is sent from and where it arrives, as seen by the user. There are many possibilities regarding which part of the storage hierarchy data is transferred from/to. From closest to farthest from the CPU, end points of data transfer can be:

- processor registers
- coprocessor registers
- data cache
- memory-mapped I/O — the network interface is mapped into the memory space of the processor, and is accessed using load/store instructions (bypassing the cache).
- memory buffers (i.e. DMA) — the processor gives the network interface the address of a buffer in memory; then the network interface autonomously accesses the buffer over the memory bus while the processor continues its computation (arbitration between the processor and network interface for the memory bus is required).

In [HJ92] the performance differences between these various options were quantitatively studied. They found that sending and receiving messages from processor registers resulted in the best performance. Moving the data back and forth to coprocessor registers would likely require an additional cycle per word at each end of the transfer. Cache and memory-mapped end-points of data transfer incur additional costs for accessing busses. If the data's ultimate source and destination are CPU registers, then a DMA data transfer will add even more latency for accessing memory.

Not surprisingly, having to go through the kernel to access the network interface incurred an additional substantial penalty.

The study also noted that random-access mechanisms were superior to FIFO mechanisms, especially when replying to messages or forwarding them. This is because with thoughtful design of message formats, much of the data can be reused without being moved. Also, an immediate send retry is trivial since the message data is still there; just execute the SEND instruction again. With a FIFO interface, the entire message may have to be pushed out to the network interface again.

Transferring data to/from processor registers, while providing the best performance, is the most intrusive to the microprocessor architecture, requiring a new chip design; the other schemes can be implemented by adding external circuitry to a normal commodity microprocessor. A related drawback of the intrusion is its impact upon the use of existing software tools. For example, is might be that a handful of registers would have to be reserved for incoming data, and thus be off-limits for computation – not only would computational performance suffer from not having as many registers available, but compilers would have to be altered.

Despite these potential difficulties, it was decided to transfer data directly to/from processor registers for the sake of performance. The demands of supporting both fine- and coarse-grained parallelism, along with the increasing computational performance of microprocessors, necessitate an on-chip network interface. In particular, the T1 microprocessor with its vector unit requires a similarly high-performance network interface to remain balanced.

The registers used for data transfer are the four argument-passing registers and the frame pointer register, by MIPS convention. A SEND instruction uses the contents of (the currently visible copies of) these registers to construct the data portion of the outgoing packet. Upon packet arrival, the scalar data of the packet is placed directly into those registers. The use of the argument-passing registers highlights the similarity between Active Messages and RPC calls (the key differences are that Active Message handlers neither perform serious computation nor automatically return a result). The addition of the frame pointer extends this to a remote closure invocation for use in object-based programming — the object, the method, and the arguments are specified in the message.

In order that an arriving message does not overwrite data being used by the primary computation, there are additional copies of the registers used for data transfer (see Figures 6 and 7, and also Figures 3 and 4). Which set of these registers are visible, *i.e.* mapped into the register file, depends on the context of the CPU: primary computation, request handler, or reply handler (request vs. reply is discussed in Subsection 3.5). At the point when control is dispatched to a message handler, the message data is transferred into the the register set of the new context. Restrictions on handler nesting prevent a message from overwriting a previous message before it has been used; this is further discussed in Subsection 3.5. When the handler ends and control is returned to the previous computation, the registers of the previous context once again become visible. A request handler can compose a new reply message (possibly reusing data from the request message it is handling) without having to preserve the registers it uses. In addition to the registers used for data transfer, two temporary registers are also replicated for use by handlers, and their contents are preserved between handler calls.

This scheme has two distinct performance benefits. Handlers will not have to perform callee-saving of register contents to memory as long as they can operate within the replicated register subset (this should be true, since Active Message handlers are not supposed to perform any substantial computation). Also, handlers do not have to load incoming message data from coprocessor registers, memory, or memory-mapped I/O — the data is immediately usable in computation. Yet primary computation sees no difference in its register file — sequential efficiency will not suffer, and existing compilers can be used. The resulting communication architecture is both efficient and incremental: low overhead communication has been achieved without disturbing the sequential architecture.

Note that the majority of the CPU registers are still shared between the main code and the handler code; this allows an intimate coupling between the primary computation and the handlers, which was found to be important in the [SGS+93] study.

One interpretation of this design is that arriving message data is placed in coprocessor registers, and that these registers are mapped into the CPU register address space for

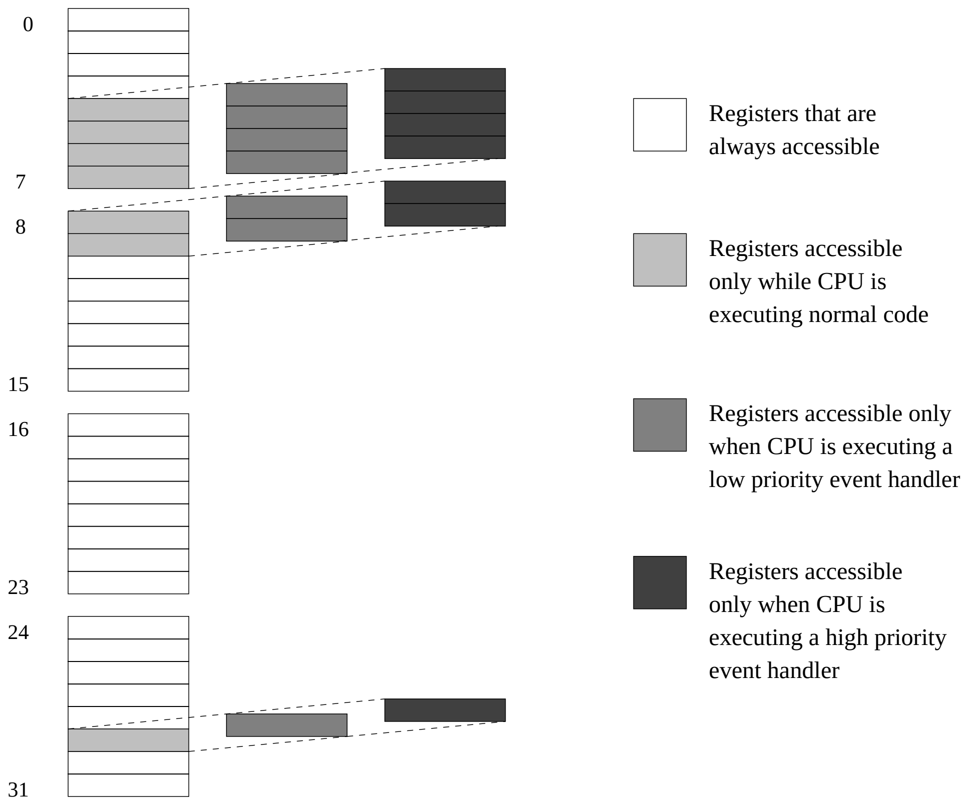

0
7
8
15
16
23
24
31
Registers that are always accessible
Registers accessible only while CPU is executing normal code
Registers accessible only when CPU is executing a low priority event handler
Registers accessible only when CPU is executing a high priority event handler

Figure 6: CPU register file, extended with multiple copies of some registers.

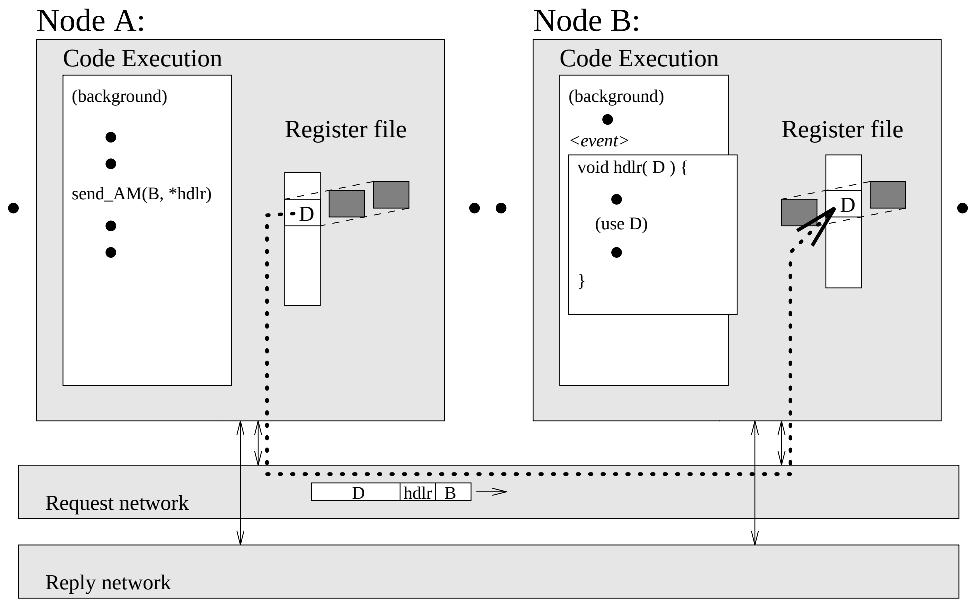


Figure 7: Data transfer. The dark-shaded copies of registers are not accessible in the states shown (register $29 not shown to simplify picture).

the duration of the handler. If this mapping were not implemented, then the design would degenerate to the less intrusive but less efficient design using coprocessor registers for data transfer endpoints.

In the T1 microprocessor, the vector unit provides an extension to this mechanism for large data transfers. In addition to the 5 words from the CPU registers, a send can optionally include the contents of a vector register (up to 32 words) in the message being sent. At the receiving node, the data appears in vector register `$vr1`. Again, each event context (primary computation, request handler, and reply handler) has its own copy of this register to prevent live data from being overwritten. The high-bandwidth path between the vector unit and the memory system makes vector load-send-receive-store a very efficient way of performing the transfer of large data blocks between the memories on different nodes.

Also, large multi-message data transfers are simplified by the fact that the architecture guarantees *in-order* delivery of messages between any pair of nodes. The implementation of the CNS-1 network, which uses dimension-order routing on a two-dimensional mesh, guarantees this behavior. Even so, it was not clear that it should be part of the specification, because a different implementation may wish to use adaptive routing or multiple virtual channels on the mesh, or perhaps even change the topology to a fat-tree network, none of which can guarantee in-order delivery. However, in a project performed jointly between the graduate computer architecture and VLSI design classes at UC Berkeley it was shown that packet reordering can be performed efficiently in hardware in the network interface [GWC93]. Thus even if a new network transport mechanism delivers packets out-of-order, in-order delivery as seen by the software can be guaranteed with the addition of some circuitry.

By transferring data directly to/from processor registers, problems with cache incoherency are avoided. The main processor and the coprocessors all access memory through the same cache, so that they have a consistent view of memory. Since any multiple copies of data on different nodes are created explicitly in software, the burden of consistency management in such cases is also in software. A somewhat related problem, however, may occur if the compiler for the primary computation does not take the actions of handlers into account. For example, consider a situation where the primary computation is busy-waiting on a flag that will be set by a handler:

```
$L1: lw  $6, flag($sp)
     nop
     beq $6, $0, $L1
     nop
```

An optimizing compiler, seeing that location *flag* is not modified within the loop, might move the load outside of the loop:

```
      lw  $6, flag($sp)
      nop
$L1:  beq $6, $0, $L1
      nop
```

Now even when the handler writes a non-zero value to location *flag* in memory, the primary computation will not see it. The most straightforward way of avoiding this situation in C is to declare the variable *flag* to be volatile. This tells the compiler that the value of the variable may change in ways not known to the compiler, and will prevent the compiler from making optimizations leading to incorrect behavior. Better yet, the compiler should use one of the registers shared between the handlers and primary computation for the variable *flag*.

3.4 Synchronization

There are two basic ways an Active Message implementation can deal with synchronizing arriving Active Messages with the primary computation executing on the node. Under an *interrupt-driven* model, incoming messages can interrupt the processor asynchronously as soon as they arrive. Under a *polling* model, the processor periodically checks the network to see if a message is waiting, and if so extracts and handles it.

With the interrupt-driven model, messages are usually extracted immediately from the network, resulting in both reduced network congestion and also lower average round-trip latency for request-reply exchanges. Another advantage is that there is no need for the user or compiler to explicitly add polls to the code doing the primary computation. However, when an operation must be performed atomically with respect to message arrival, such as the modification of a global data structure, some technique must be used to temporarily disable message arrivals. The mechanism for forming these critical sections should be inexpensive in terms of cycles consumed.

With a polling model, the formation of such critical sections is trivial; simply don't poll. The burden of adding polls can be reduced if the network is always polled during send attempts; then polls only have to be explicitly added to computation-only phases of program execution. Polling may have the benefit that many waiting messages can be handled by one poll, reducing the number of context switches which may be costly in some implementations. Also, it may be easier for a programmer to manage the complexity of interacting pieces of code (primary computation and multiple handlers) if the possible asynchronicity is restricted.

There was no need to choose between the two models. Interrupt-driven message reception is the more general mechanism, and can emulate a polling mechanism. This is done by simply having message reception disabled by a default. A poll is then emulated by the sequence "enable message reception, disable message reception". It is important that these enable/disable operations are inexpensive. This is true in this design; message reception is disabled or enabled simply by moving a zero or one, respectively, to the event enable register. Thus the cost of forming a critical section in an interrupt-driven model is identical to the cost of performing a network poll in a polling model – 3 instructions. Hazards involving transfers to and from network interface control registers are not known at this time; there may be some delay slots in the code fragments that would have to be filled with unrelated instructions or with `nop` instructions. See Figure 8.

```
#define ENTER_CRIT                   \
            CTC2  $event_enable, $0  \! clear event enable bit
            ---delay slot ?---        ! wait for it to take effect

#define EXIT_CRIT                    \
            ORI   $t1, $0, 0x0001    \
            CTC2  $event_enable, $t1  ! set event enable bit

#define NI_POLL                      \
            ORI   $t1, $0, 0x0001    \
            CTC2  $event_enable, $t1 \! set event enable bit
            ---delay slot ?---       \! wait for it to take effect
            CTC2  $event_enable, $0  \! clear event enable bit
```

Figure 8: Synchronization Code Fragments

3.5 Send Failure

Any network can get congested to the point that it cannot accept any more outgoing messages from a node. What is done by the node in this situation requires careful design. The result otherwise could be either deadlock or unbounded buffering requirements.

The solution adopted here is almost identical to that used by CMAM, the Active Messages implementation on the CM-5. There are two logically disjoint networks, a request (low-priority) network and a reply (high-priority) network. A handler for a message from the request network can be preempted only by a message from the reply network. A handler for a message from the reply network cannot be preempted at all. This allows at most the nesting of two handlers; this small fixed amount of nesting allows for the implementation of state preservation in hardware.

A request handler is allowed to loop trying to send reply messages[4] until it succeeds; a reply handler is not allowed to loop indefinitely for any reason, but must execute quickly to completion.

[4] *Reply* message simply means a message sent on the reply (high-priority) network. A reply message can be sent to any node, not necessarily the node that sent the request message being handled.

The restricted nesting of handlers is enforced in the CM-5 implementation by simply not polling the request network while executing a request handler, and not polling at all during a reply handler. The hardware solution here is similar, but adapted to the interrupt-driven synchronization. At the beginning of a request handler (*i.e.* at the inlet event of a request message), the low-priority event enable bit is automatically cleared, disabling the reception of any other request messages but allowing the reception of reply messages[5]. At the beginning of a reply handler (i.e. at the inlet event for an incoming reply message), the event enable bit is cleared, disabling the reception of *any* other message. At the end of the handler, the HRET (return from handler) instruction automatically restores these enable bits to their prior values.

If the network cannot accept another message when a SEND instruction is executed, the coprocessor flag is cleared to indicate a failure, and execution continues. This is different than the CM-5 software implementation of Active Messages, in which a send will loop indefinitely until it is successful pushing the message out to the network. A "failable" send has the drawback that requires additional instructions to test for its success. In this architecture, however, this test is typically only one instruction, a BC2F (branch on coprocessor flag false) instruction.

Having a failable sends can be useful in some situations. With retry-until-successful sends as in CM-5 Active Messages, handlers of request messages can only send reply messages, and handlers of reply messages cannot send any message. With a failable send software can attempt any priority send from any priority handler, as long as it is prepared to give up after a fixed time and take alternate actions.

Another possible way of dealing with send failures that was briefly considered is to invoke a special fault handler at that time. This approach is attractive because it eliminates the need to check for success after each send. It is especially attractive when send failures are rare. This approach is not taken here for a number of reasons. The first is that the cost for the check of send success is exceptionally cheap in the T1 — one instruction. Also, it is expected that the action to be taken on a send failure may be different in each case; having one global fault handler would be awkward in this situation.

Also, due to the efficiency of the network interface, a rapid sequence of sends may exceed the network bandwidth. Thus transient send failures may be common in situations where a node is doing nothing other than sending out large quantities of data. Trapping to a fault handler in such situations would not be appropriate.

Some applications may only need one priority because of characteristics of the communication pattern or because high-level flow control is used to bound the amount of message buffering required at each node. With such applications it would be desirable to get

[5]This differs slightly from the CM-5 implementation, in which request handlers are atomic except when they attempt to send. In this implementation, a request handler that must execute atomically with respect to reply handlers must explicitly form a critical section. This is not a large penalty due to the low cost of forming critical sections.

the full physical bandwidth of the network while only using one priority. If the two priorities are implemented with physically disjoint networks, the applications may have to artificially split their communication between the two priorities in order to make full use of the communication resources. With the CNS-1 , however, the same physical links are demand-multiplexed between the two priorities; if only one priority is being used, it gets the full physical bandwidth. This allows applications to use the two priorities in whichever way is natural while making optimal use of the communication resources.

3.6 Network Virtualization

The CNS-1 is designed for single-user, batch-style processing; no timesharing or space partitioning among multiple users will be done. Thus network virtualization is not an issue; however, this subsection examines what changes would have to be made in order to support network virtualization.

In a general communication architecture, it would be desirable to perform some type of timesharing. The most straightforward way of doing this would be to adopt the approach taken with the CM-5: gang-schedule all of the nodes synchronously [L$^+$93]. The network must be drained of messages when a process is preempted. During the draining phase, each message in the network simply goes to the nearest node, whether or not that is its destination; this limits the maximum amount of buffering done at any one node to V/N rather than V, where V is the total network volume and N is the number of nodes.

To put hooks in for this type of operation, there would have to be a way for the kernel to set a flag in the network interface that causes all user packets to drain to the nearest node, where the kernel could store them in memory. These messages would then be resent when that process resumes control. Some special precautions may be necessary due to the possibility of send failures at this point (the network may fill up before the kernel can put all the buffered packets back on the network; the user program must be run for a short while to consume some of the messages before the kernel can send the remainder of the buffered packets). Also, some mechanism must ensure that even after preemption and subsequent rescheduling, a process' messages still satisfy the in-order delivery specified in the architecture.

Of course, there would have to be some way for the kernel on each node to know that it is time for a context switch. The simplest way is for each kernel to set its own timer, and when the quantum is over, drain the network of user packets so that the kernel can communicate with the other kernels or the host. If preemption is triggered from an external source, then there must be some reliable way to communicate with each kernel. It the CNS-1 this is not possible using the data network, since a user program could deadlock both network priorities. The CNS-1 would be able to perform such signaling to the kernels using the TSIP network, a separate diagnostic network similar to JTAG. Another option would to be to add a new priority on the data network reserved exclusively for kernel use.

3.7 Protection

Even though the nodes in the CNS-1 support only a single user (no timesharing), a protection mechanism is still necessary. While there is no need to make sure that a user does not corrupt another user's code or data, the single user must still be prevented from accessing the resident kernel's code and data. Why do we need a protected kernel at all, if the nodes are being used by a single user? The main reason is that nodes will directly access I/O devices, and such accesses must be regulated by the operating system; hence the need for a protected kernel. In addition, a debugger must be protected from errant writes of the program being debugged.

The memory protection mechanism for the planned implementation of the T1 architecture is straightforward: a write barrier is specified in a coprocessor 0 (COP0) register, and any attempt to write past this barrier while in user mode results in an exception.

The extension of this protection across the network is similarly straightforward. When a message is sent, a bit in the packet indicates whether the sending processor was operating in user or kernel mode. This mode is the mode the receiving node runs at while handling the message. If a packet sent at user mode contains a handler IP that points to code in kernel space, an address error exception will occur when the fetch of the first handler instruction is attempted on the receiving node. This prevents user code from invoking kernel code (or modifying kernel data) on another (or the same) node by way of the Active Message mechanism.

In general, message reception should be disabled while the processor is running in kernel mode. This is because even though a user-level packet arriving at a node will cause the processor to switch to user mode, preventing invalid memory accesses, the handler may still corrupt any values the kernel has stored in processor registers.

The kernel/user bit is used only when the message has arrived at the destination; kernel packets get no special priority on the network. If the application program has deadlocked both priority networks, not even kernel packets can get through. For debugging in such situations, the kernel can be accessed via the TSIP network.

3.8 Events vs. Exceptions

Some may question the wisdom of adding a completely new event mechanism to the MIPS instruction set architecture. Why can't a message arrival simply cause an interrupt as is already defined?

The main reason is performance; using the existing interrupt mechanism would require first going through the generic kernel exception handler, which would then have to dispatch to the Active Message handler. A well-tuned kernel dispatch routine could possibly execute in 15-20 cycles; a more realistic estimate would be at least 100 cycles. The newly defined event mechanism normally operates completely at user level and dispatches directly to the required handler, with an estimated overhead of 3-5 cycles.

At a more basic level, events and exceptions serve different purposes. Events are intended to be an integral part of computation, occurring frequently and at user level, while exceptions signal a situation where the kernel must get involved.

3.9 Extensions

The implementation described up to this point is a complete Active Messages communication architecture. However, because of known characteristics of the intended applications for the machine being built, some extensions to accelerate certain operations or to provide more functionality have been added in a way consistent with Active Messages.

Other Events

In addition to inlet events caused by arriving messages, there are two additional events that are anticipated to help facilitate efficient use of the communication network.

The first is an *outlet* event. This occurs when both the network is ready to accept another additional outgoing message and the outlet enable bit is set. This is analogous to an interrupt-driven message reception at the destination (having the sending node periodically attempt to send would be analogous to a polling receive at the destination). This will be useful in the case of large memory-to-memory data transfers occuring in the background.

The second additional event is the *timelet* event. This occurs when both the timelet enable bit is set and also the free-running timer matches the count register (both the timer and count register are in the network interface and are user-accessible). Although this function is not strictly communication-related, it is anticipated that it may serve a function similar to the outlet event.

Hardware Implementation of Handlers

The inclusion of hardware-interpreted Active Message handlers does not conflict with the Active Message philosophy, as long as it does not interfere with the efficiency and versatility of the basic communication architecture. No hard-wired handlers are defined in this specification; however, the implementor is free to define some. To allow for this, handler addresses in the range 0xFFFF0000 - 0xFFFFFFFC (i.e. small negative integers) are reserved for implementation-specific hardware-interpreted handlers.

When considering the addition of a hard-wired handler, a RISC approach should be taken: unless it can be shown to result in at least a 1% overall improvement of the computer's performance, it is not worth the trouble. The efficient hardware dispatch to user-level code means that a short software handler may require a total (dispatch, execution, and return) of only about 10 cycles, so it is unlikely that handler invocation will occur frequently enough that saving a few cycles per invocation will result in a total performance improvement of more than 1%.

One possible advantage of using hardware-interpreted handlers is the elimination of competition between handler code and primary computation code for space in the I-cache. However, such competition may be eliminated by other means in the implementation, such as having a separate lockable I-cache for handler code.

Possible Hard-wired Handlers

One possibility for hard-wired handlers would be to handle remote memory accesses such as fetches. A fetch request packet would contain a memory address to fetch, the address of the requesting node, and the pointer to the handler on the requesting node that will handle the reply. The network interface at the node holding the memory location(s) would automatically access the cache or memory to retrieve the value(s) and send it back in a reply active message. Because much flexibility is lost in dealing with synchronization, the hard-wired handlers would probably only be used in bulk transfers. A sequence of messages using a hard-wired handler would likely be followed by a message using a software handler, which would increment the correct counter or return an acknowledgement message to indicate the completion of the transfer.

If messages were limited to a payload of the contents of five scalar registers, such hard-wired handlers could result in much more efficient memory-to-memory transfers of large blocks of data. However, due to the efficiency of the existing vector message transfers, a large amount of data can be transferred while stealing only a few cycles from the CPU; thus there is little incentive for adding hard-wired remote memory access handlers.

Another possible function that could be handled in hardware is forwarding, or indirection. The packet would contain the address and handler of its ultimate destination, but would be sent to a different node (say Processor X). Processor X would read in the packet, see that its handler IP indicates that it should be forwarded to a different node, reformat the packet with its new network address and handler, and put it back on the network. This function would be useful in some randomized algorithms, or whenever the user wants more control over the path taken by packets.

Multicast

Many applications require the dissemination of data from one node to many other nodes. Such multicasts can be handled simply by having the source send a series of identical messages, one to each receiving node. If this simple approach is found to be consuming too much bandwidth and/or time (to send the repeated messages), a tree distribution scheme can be set up in software: the source sends the message to two other nodes; each of these nodes resends the messages to two other nodes, and so on.

With hardware support for multicasts, latency and bandwidth can be reduced below that possible with either of the software methods described above. However, the hardware support for multicast should not have a negative impact on normal unicast traffic, or else overall performance may actually decrease. The multicast support must be useful yet also simple and easy to implement.

The CNS-1 multicast support mechanism possesses these characteristics. The mechanism is simple: drop of a copy of the packet at every node that is passed on the journey from the source to the destination, not including the source itself. The topology of the CNS-1 is basically a two-dimensional mesh. The path between two nodes in the same column or same row is guaranteed to be the shortest-path straight line, but if the source and destination are in both different rows and different columns, the route is not specified. Thus the behavior of multicasts is defined only when the source and destination are in the same row or the same column. This leaves the implementation some leeway in the exact routing protocol used, and in fact makes no restrictions at all on unicast routing.

With this mechanism, the distribution of data to an entire row or column of nodes takes only as long as it would take to send a unicast message to the far node, and consumes exactly as much network bandwidth. To cover a two-dimensional patch of processors, a two-step distribution would be used (see Figure 9).

Actually the CNS-1 mesh wraps around in one dimension (see Figure 5). This complicates sending a multicast to an entire ring of processors; the "shortest-path straight line" between any two nodes can cover at most half of the nodes on a ring, because the message will go the shortest way around the cylinder. A node wanting to multicast to every node on a ring will have to send two multicasts, one each direction around the cylinder. Another possibility is to modify the routers so that when a node sends a multicast to itself, it will actually travel all the way around a ring.

Using this mechanism forces the software to be aware of the network topology, the relative positions of the nodes in the network, and part of the routing algorithm. These are parts of the communication micro-architecture and should not really be visible in the communication architecture; however, most of this information is "performance visible" and would have to be known in order to optimize for locality even for normal unicast communications.

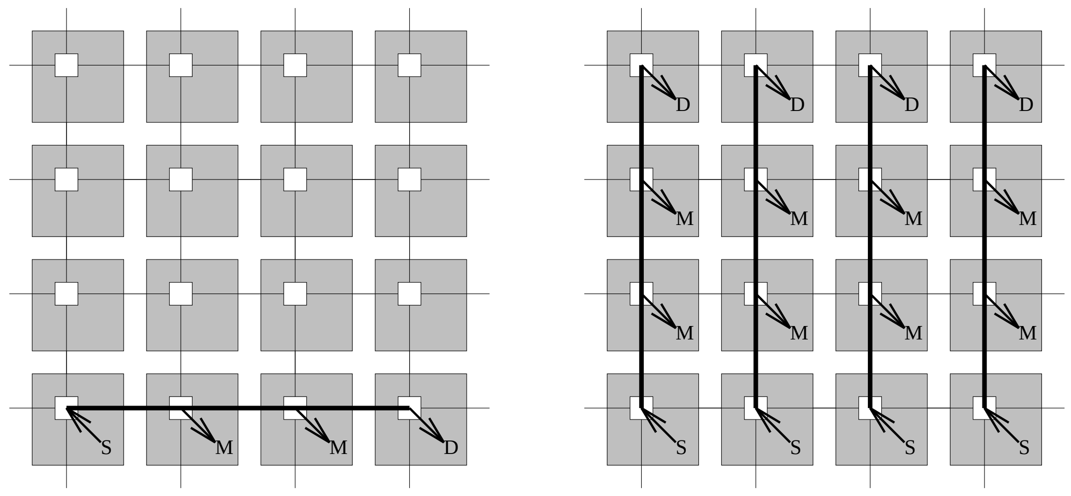


Figure 9: Two-phase multicast to two-dimensional region. S signifies source of message; M signifies reception of multicast copy; D signifies actual destination of message. In actuality, receiving nodes can not tell whether or not they are the final destination, but they can tell whether the message received is part of a multicast or not. The target region for the multicast does not have to be rectangular.

For each basic send instruction, a multicast version has been added to the instruction set. Each multicast instruction behaves identically to its non-multicast version except that the message is received at each node along the path that is traveled.

The only change in the network interface registers is that a new bit field has been added to the status register; this bit indicates whether or not the packet that is currently being handled was part of a multicast.

3.10 Conclusion

In this section many of the design details of the T1's Active Message communication architecture have been described along with some of the reasoning behind each decision. The resulting architecture is straightforward to use and lends itself to an efficient VLSI implementation. Although the exact numbers will not be known until the design of the T1 is complete, it is estimated that a message send (normal or multicast) of up to 37 data words will cost approximately 5 cycles; message reception and the dispatch to the appropriate handler will also have an overhead of about 5 cycles (neglecting I-cache misses). Forming a critical section or performing a network poll would cost less than 5 cycles.

For more details of the interface between software and the network interface, refer to the specification itself, included as Appendix A.

4 Previous Research

This project, the network interface for the T1 processor, is not the first attempt at defining a coherent communication architecture. Many earlier academic projects have added support for multicomputing to a microprocessor.

The MDP processor in the J-Machine [DCC$^+$87] was designed to provide hardware support for message-driven programming languages. However, it was not able to implement the full message-driven architecture because the resulting complexity would have been overwhelming. It was restricted to hardware support of one thread at each of three priorities; a context switch between threads of the same priority involved moving register contents back and forth to memory. In reality, the MDP would support Active Messages very well if it weren't for the fact that the register sets for the different threads are disjoint, preventing a close coupling between the primary computation and Active Message handlers. Another disadvantage of the disjoint register sets is that the number of registers available to each thread is greatly reduced. This prevents the high computational efficiency witnessed in conventional RISC processors resulting from having a large amount of state close to the functional units (see [SGS$^+$93] for a detailed comparison between the J-Machine and the CM-5).

A more recent project at MIT is the Alewife multiprocessor and its Sparcle chip [AKK$^+$93]. The approach taken in the Sparcle chip is similar to the work described in this report, in that incremental changes are made to a mainstream microprocessor, Sparc in the case of Sparcle. The current implementation of Sparcle is not as ambitious as the T1 as far as providing an intimate coupling between computation and messaging (this was not a primary objective, as the Alewife's main communication mechanism is cache-coherent shared memory). Message send and receive is handled by the communications and memory management unit (CMMU) located on a separate chip. The CMMU signals the reception of a message via a dedicated trap line into the Sparcle chip, causing control to be vectored to a trap handler which then can load and interpret the message. The Sparcle chip supports a limited number of threads by segmenting the Sparc register file into four non-overlapping windows. While this may superficially appear similar to the multiple physical copies of some of the registers in the T1 architecture, the difference is that Sparcle supports three general computational threads plus one context for traps and handlers, while T1 supports only one computational thread along with two priorities of handlers; these handlers, while tightly coupled to the primary computation, do not perform any computation themselves.

The *T project at MIT has incorporated a network interface with another commercial processor, the Motorola M88110, to make the M88110MP [PBGB93]. The resulting communication architecture is very similar to that presented here. There is a minor difference in data placement; the M88110MP has added separate send registers and receive registers, which are accessed through new instructions (but cannot be used directly in normal computation). Also, although it is straightforward to implement a polling

Active Message layer on top of the existing hardware, the current implementation has no support for hardware dispatch to the Active Message handler. To its credit, the *T has attacked the UNIX protection issues that were not addressed in this project. Also, the *T added a network interface to a *superscalar* processor, which posed a somewhat greater challenge.

A slightly different architecture for multicomputers results when there is a separate communication processor, as in the Meiko CS-2 [BCM94]. This type of architecture can be thought of in the Active Message framework as having a processor dedicated for Active Message handlers (although the CS-2 does not currently support user Active Message handlers on its communication processor). The advantage of having a separate communication processor is that when the handlers are independent of the computation on the main processor, as in the case of a remote fetch, no cycles are stolen from the primary computation. However, when the message does interact with the primary computation, there is typically extra latency and overhead for data transfer and synchronization. When the CNS-1 is completed, quantitative comparison studies should provide useful data about these tradeoffs.

5 Concurrent Work

There has been a substantial amount of work done towards the implementation of the communication micro-architecture and communication hardware to go with the communication architecture presented here. This work falls under two main categories: the creation and use of a simulator running on a Thinking Machines' CM-5 to compare different design options for the communication micro-architecture (e.g. buffering and flow-control strategies), and the subsequent creation of an RTL model of the communication hardware.

6 Conclusion

This report contains a full specification of a communication architecture to be incorporated into the MIPS instruction set architecture. The communication architecture is based on the Active Messages mechanism, which has been shown to possess the desirable attributes of versatility, efficiency, and incrementality. The resulting architecture will be able to provide high performance under any of a variety of programming models: send & receive, shared memory, or dataflow.

Because the communication architecture is based on Active Messages, the CNS-1 will be able to benefit from much of the work that has gone on here at Berkeley in the area of parallel language development. Split-C will provide a global address space with the familiar C interface. Dataflow-style languages such as Id90 will be supported efficiently by the TAM model. These languages have already been implemented on top of other similar Active Message implementations, so that porting them to the CNS-1 should be straightforward. However, in both compilers there will be many new opportunities for optimization for the T1 architecture, e.g. allocating registers to pass data or synchronization information between handlers and the primary computation.

The fruits of this project will be used in a chip that will be fabricated and used in a real multiprocessor computer. I hope that the success of this chip will demonstrate the benefits of adding a tightly-coupled network interface to a RISC microprocessor. Hopefully MIPS and other major chip makers will follow Motorola's lead and devote some real estate to a similar network interface to create a multiprocessing version of their microprocessor. Perhaps eventually an integrated network interface will be a standard part of every RISC microprocessor.

6.1 Acknowledgements

Most if not all of the ideas concerning communication architectures and Active Messages presented here were developed by Thorsten von Eicken. Even many of the implementation details are similar to those in CMAM, the CM-5 Active Message implementation by von Eicken. The rough sketch of the communication architecture desired for T1 was done by Krste Asanovic, the head architect of T1, and John Wawrzynek, the project leader. I have regularly consulted Krste Asanovic, John Wawrzynek, Brian Kingsbury, Bertrand Irissou, Sven Meier, and Stelios Perissakis about the VLSI implications of my choices. The development of the multicast extension described here was spurred by discussions with Jerry Feldman. Useful feedback has been gathered from many of the people working on the CNS-1 project, including David Johnson, Silvia Meuller, James Beck, Phil Kohn, and others at ICSI, including Stephan Murer and Steve Omonohundro. Thanks go to Katherine Yelick, John Wawrzynek, and Andrea Dusseau for reading and commenting on earlier drafts of this report; their feedback resulted in many significant improvements.

This research is supported in part by a National Science Foundation Graduate Fellowship, Grant URI-N00014-92-J-1672 from the Office of Naval Research, and National Science Foundation Infrastructure Grant number CDA-8722788. Additional support comes from the International Computer Science Institute, whose funds are provided by the ministries of research of Germany, Italy, and Switzerland, and cooperating companies.

References

[AKK+93] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[BCM94] Eric Barton, James Cownie, and Moray McLaren. Message passing on the Meiko CS-2. *Parallel Computing*, (20):497–507, 1994.

[CDG$^+$93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proc. Supercomputing '93*, Portland, Oregon, November 1993.

[CGSvE93] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. In *Journal of Parallel and Distributed Computing, Special Issue on Dataflow*, June 1993.

[DCC$^+$87] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a Message-Driven Processor. In *Proc. of the 14th Int'l Symposium on Computer Architecture*, June 1987.

[GWC93] Seth Copen Goldstein, Su-Lin Wu, and Timothy John Callahan. Hardware Support for Packet Reordering and Flow Control in Multicomputer Networks. CS252 and CS250 class projects, University of California at Berkeley, November 1993.

[HJ92] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proc. of 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, October 1992.

[Kan88] Gerry Kane. *MIPS RISC Architecture*. Prentice-Hall, 1988.

[KJA$^+$93] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Practice and Principles of Parallel Programming (PPoPP) 1993*, pages 54–63, San Diego, CA, May 1993. ACM. Also as MIT/LCS TM-478, January 1993.

[L$^+$93] Charles E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the 5th Annual Symp. on Parallel Algorithms and Architectures*, 1993.

[Lun94] Stephan S. Luna. Implementing an Efficient Portable Global Memory Layer on Distributed Memory Multiprocessors. Technical Report UCB//CSD-94-810, University of California at Berkeley, May 1994.

[PBGB93] G.M. Papadopoulos, G.A. Boughton, R. Greiner, and M.J. Beckerle. *T: Integrated Building Blocks for Parallel Computing. In *Proc. Supercomputing '93*, Portland, Oregon, November 1993.

[SGS$^+$93] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proc. of the 20th Int'l Symp. on Computer Architecture*, San Diego, CA, May 1993.

[vE93] Thorsten von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. PhD thesis, University of California at Berkeley, December 1993.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. (Also available as Technical Report UCB//CSD-92-675, University of California at Berkeley).

Appendix A
Network Interface Specification

A.1 Overview

The Torrent network interface is implemented as part of coprocessor two (COP2), which is shared with the Torrent vector unit.

The network interface has been designed to support the *Active Message* communication mechanism (Thorsten von Eicken et.al.). This mechanism minimizes communication overhead, allows communication to overlap computation, and coordinates the two without sacrificing processor performance.

Network Interface Registers

There are a number of network interface registers located in COP2 register space; they are accessed via the `ctc2` and `cfc2` instructions. These registers are used to configure the network interface, report interface status, set the address of certain event handler routines, and report auxiliary information about incoming messages.

Message Data Transfer

Message transmission and reception are modeled as a remote procedure invocation that causes an asynchronous subroutine call. Scalar headers are built up in the CPU registers, using registers `$4-$7` and `$29`. These correspond to the argument and frame pointer registers in the MIPS procedure call convention. A message send instruction (implemented as a COP2 instruction) specifies the destination processor in a register, the scalar message length in words, and a handler address. The contents of the scalar registers at the point of the send instruction are used for the data component of the message.

On message arrival, an event causes CPU control to be transferred to the handler address specified in the message. As control is transferred to the handler, part of the CPU register set is swapped to an event set that includes the newly arrived message data in the same registers used for sends. Specifically, the argument passing registers (`$4-$7`) and the stack pointer register (`$29`) are swapped to an event set containing the newly arrived data, and two of the temporary registers (`$8` and `$9`) are swapped to an event set which can be used as scratch space or to maintain state between handler invocations. See Figure 1.

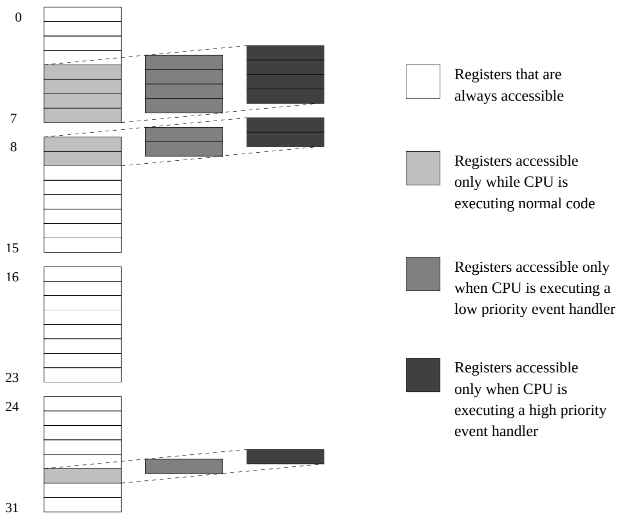

0
7
8
15
16
23
24
31
Registers that are always accessible
Registers accessible only while CPU is executing normal code
Registers accessible only when CPU is executing a low priority event handler
Registers accessible only when CPU is executing a high priority event handler

Figure 1: CPU register file

Optionally, a data vector is included in the message from a vector register in the vector unit. The data vector is always received into an event copy of `$vr1` which is swapped in during the message handler (analogous to the CPU register swapping). This allows large data blocks to be sent and received with low CPU overhead.

Network Interface Instructions

The instruction set has been extended in the COP2 opcode space to deal with the network interface. The standard `ctc2/cfc2` instructions are used accessing network interface control registers.

The standard MIPS branch-on-coprocessor instructions are used to branch conditionally upon the condition flag in the network interface. This flag is set or reset by the send instruction depending upon whether it was successful.

There exist four different versions of the send instruction. A message can be sent at either high priority or at low priority, and a message may or may not include the contents of a vector register in addition to its scalar data.

The HRET instruction is used to return the CPU to its previous processing state and resume program execution at the point it was preempted.

Message Delivery

While the exact performance of the network will depend upon the network implementation and network load, some general characteristics of the network can be described.

The network will not "drop" any messages – if the network accepts a message, it is guaranteed to eventually get through to its destination. If messages are sent to a processor faster than it processes them, the messages will back up through the network, and eventually the sending processor(s) will be inhibited from sending any more messages.

There are two priority levels for messages – *low* and *high*. These can be thought of as two logically separate networks. Even if low priority messages are backed up so that a processor cannot send any more, it can still send high priority messages that can get to the destination regardless of the low priority traffic. High priority messages can only be blocked by other high priority messages.

It is guaranteed that the delivery of messages is *in-order* between any pair of processors. That is, the messages sent from processor A directly to processor B at the same priority level are guaranteed to arrive at processor B in exactly the same order as they were sent from processor A.

For further discussion of properties of the network and its usage, see Subsection A.10.

Network Interface Events

There are three different *events* associated with the network interface which can interrupt the normal operation of the CPU. The first type of event is the arrival of a message. In this case CPU control is transferred to the handler specified in the message. The second type of event is an *outlet*, which can cause CPU control to be transferred to an outlet handler (specified in a coprocessor register) whenever the network interface can accept another outgoing message from the CPU. The third type of event is a *timelet*, which causes CPU control to be transferred to a timelet handler (also specified in a coprocessor register) after a certain amount of time has elapsed.

Events are divided into two priority levels – high and low. The arrival of a high priority message is the only high priority event. Outlets, timelets, and arrivals of low priority messages are all low priority events.

Low priority events can preempt normal computation but not other event handlers. High priority events can preempt normal computation or low priority event handlers but not high priority event handlers. This allows at most two levels of event handling. See Subsections A.9 (Events) and A.7 (The HRET Instruction).

Events are enabled or disabled depending on the setting of two bits in the network interface status register: `EE` (event enable) and `LPEE` (low priority event enable). If `EE` is clear, then *all* events are disabled. If `LPEE` is clear, then low priority events are disabled. Stated another way, high priority events are enabled only if `EE` is set; low priority events are enabled only if both `EE` and `LPEE` are set. In addition, there are event mask bits which can be used to individually disable the different types of events.

A.2 Network Interface Registers

The network interface registers are listed in Table 1.

#	Register	Description
6	`timeh`	Timelet handler.
7	`timec`	Timelet compare register.
9	`Count`	Count register.
18	`enable`	Network interface enable register.
19	`nistat`	Network interface status register.
20	`LPEPC`	Low priority event PC.
21	`HPEPC`	High priority event PC.
22	`niid`	Network interface ID.
23	`dmh`	Default message handler.
24	`outh`	Outlet handler.
25	`inh`	Incoming handler.

Table 1: Network interface registers in COP2.

A.2.1 Timelet Handler (NIR6)

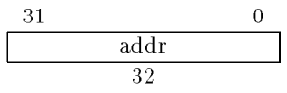


Figure 2: Timelet Handler Register Format

The timelet handler (`timeh`) register is a 32b read/write register that specifies the handler to be invoked at the occurrence of a timelet event. A timelet event occurs when the both the `tp` (timelet pending) and `tm` (timelet mask) bits are set, and low priority events are enabled. The `tp` bit gets set when the value in the `Count` register is equal to the value in the timelet compare (`timec`) register.

Since instructions must be word aligned, this register must contain an address whose two low order bits are zero. Otherwise, an address exception will occur when the CPU attempts to invoke the timelet handler.

A.2.2 Timelet Compare Register (NIR7)

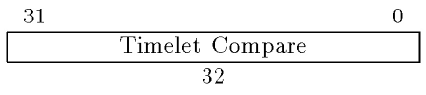


Figure 3: Timelet Compare Register Format

The timelet compare (`timec`) register is a 32b read/write register. When the value of the `Count` register equals the value of the timelet compare register, the `tp` (timelet pending) bit in the EP field of the network interface status register gets set. This causes a timelet event to occur on the next cycle in which timelets are enabled. Timelets are enabled if `tm`, `LPEE`, and `EE` are all set.

The `timec` register can be read or written at any time using the `ctc2/cfc2` instructions. Writing to the `timec` register has the side effect of clearing the `tp` bit.

A.2.3 Count Register (NIR9)

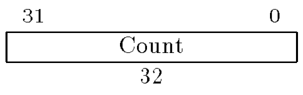


Figure 4: `Count` Register Format

The `Count` register acts as a timer, incrementing at a constant rate whether or not an instruction is executed, retired, or any forward progress is made. The rate at which the `Count` register is incremented is dependent upon its implementation. The `Count` registers of all nodes are reset and incremented simultaneously, providing a universal time reference.

A.2.4 Network Interface Enable Register (NIR18)

31 … 1	0
0	EE
31	1

Figure 5: `enable` Register Format

The `enable` register contains only one active bit: `EE` (event enable).

The `EE` (event enable) bit is read/write; it disables *all* types of events when it is clear. This bit is automatically cleared when an high priority event occurs to prevent further events from occurring, and it is automatically set by the HRET instruction of the high priority event handler. *Events should not be enabled within an event handler by manually setting the* `EE` *bit.*

Writing a 0 to this register atomically disables all types of events. This is useful for critical sections.

A.2.5 Network Interface Status Register (NIR19) – High 16 Bits

31	30	29	28
pend	nicond	onicond	0
1	1	1	1

27 24	23 20	19	18	17	16
EP	EM	LPEE	hpa	lpa	ame
4	4	1	1	1	1

Figure 6: Network Interface Status Register Format – High 16 Bits

The network interface status register, `nistat`, is formatted as shown in Figures 6, 7, and 8.

Accesses to `nistate` are fully interlocked; reads from it will reflect the results of all previous instructions, and writes to it will not be affected by any previous instructions.

The `pend` (event pending) bit is read-only. It indicates whether one of the events that is individually enabled by the Event Mask field is pending. When the `EE` bit is clear (i.e. events are disabled), this bit can be checked for a quick poll.

The `nicond` (network interface condition flag) bit is read/write. It is set or cleared by SEND instructions to indicate whether the attempted send was successful or not. This bit can be tested using the BC2F/BC2T instructions.

The `onicond` (old network interface condition flag) bit is read/write. It is used to save the contents of `nicond` when control is transferred to an event handler.

The EP (Event Pending) field contains four read-only bits indicating which of the four events are pending. This is described in detail later.

The EM (Event Mask) field contains four read/write bits which control the enabling of the four events. This is described in detail later.

The `LPEE` (low priority event enable) bit is read/write; it disables low priority events when it is clear. It is automatically cleared when a low priority event occurs, and it is automatically set by the HRET instruction of the low priority event handler. *Events should not be enabled within a handler by manually setting the* `LPEE` *bit.*

The `lpa` (low priority network available) bit is read-only; when set, it indicates that the network interface is currently willing to accept a low priority message from the CPU.

The `hpa` (high priority network available) bit is read-only; when set, it indicates that the network interface is currently willing to accept a high priority message from the CPU.

The `ame` bit is read/write; it specifies whether Active Messages are enabled or not. With Active Messages enabled (`ame` set), messages are handled by the handler specified in the message. When `ame` is clear, all messages are handled by a default message handler, which is specified by the `dmh` register.

A.2.6 Network Interface Status Register (NIR19) – Event Pending and Event Mask Fields

31 28	27	26	25	24
	hpmp	lpmp	op	tp
4	1	1	1	1

23	22	21	20	19 0
hpmm	lpmm	om	tm	
1	1	1	1	20

Figure 7: Network Interface Status Register Format – EP and EM Fields

The Event Pending field is read-only and indicates which of the four events are pending. The Event Mask field is read/write and is used to individually enable/disable each event.

The `hpmp` (high priority message pending) bit, when set, indicates that a high priority message has arrived. When it is set, a high priority message arrival event will occur on the next cycle in which both the `hpmm` and `EE` bits are set. This bit will automatically clear when there are no more incoming high priority messages waiting to be handled.

The `lpmp` (low priority message pending) bit, when set, indicates that a low priority message has arrived. When it is set, a low priority message arrival event will occur on the next cycle in which all of the `lpmm`, `LPEE`, and `EE` bits are set. This bit will automatically clear when there are no more incoming low priority messages waiting to be handled.

The `op` (outlet pending) bit, when set, indicates that the network interface can accept an outgoing message (this occurs when both priority networks are available, i.e. both `lpa` and `hpa` are set). When the `op` bit is set and also all of the `om`, `LPEE`, and `EE` bits are set, an outlet event will occur. This bit will automatically clear when the low priority network can no longer accept an outgoing message.

The `tp` (timelet pending) bit is set when the

`Count` register contains the same value as the timec register. When this bit is set, a timelet event will occur on the next cycle in which all of the `tm`, `LPEE`, and `EE` bits are set. This bit is cleared either by the occurrence of a timelet event or by a write to the timec register.

The `hpmm` (high priority message mask), `lpmm` (low priority message mask), `om` (outlet mask), and `tm` (timelet mask) bits are used to individually enable/disable each type of event (0 $\leftrightarrow$ disable, 1 $\leftrightarrow$ enable).

A.2.7 Network Interface Status Register (NIR19) – Low 16 Bits

15	14 13	12 10	9 0
`pri`	0	`slen`	`vlen`
1	2	3	10

Figure 8: Network Interface Status Register Format – Low 16 Bits

The network interface status register, `nistat`, is formatted as shown in Figures 6, 7, and 8. Bits 15-0 are only valid within a message handler; they contain auxiliary information about the incoming message.

Accesses to `nistate` are fully interlocked; reads from it will reflect the results of all previous instructions, and writes to it will not be affected by any previous instructions.

The `pri` bit indicates the priority of the incoming message (0 $\leftrightarrow$ low priority, 1 $\leftrightarrow$ high priority).

The `slen` field indicates how many scalar words are in the message, while the `vlen` field indicates how long the vector component of the message is.

A.2.8 Low Priority Event Program Counter (NIR20)

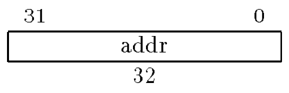


Figure 9: `LPEPC` Register Format

The `LPEPC` is a 32-bit, read/write register used to store the address where processing resumes after the completion of a low priority event handler.

A.2.9 High Priority Event Program Counter (NIR21)

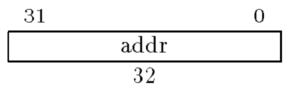


Figure 10: `HPEPC` Register Format

The `HPEPC` is a 32-bit, read/write register used to store the address where processing resumes after the completion of a high priority event handler.

A.2.10 Network Interface ID (NIR22)

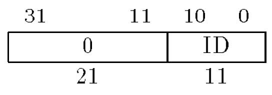


Figure 11: Network Interface ID Register Format

The network interface ID (`niid`) register is a 32b read only register that contains a value giving the unique identity (i.e. network address) of the processing node.

The `niid` register is formatted as shown in Figure 11. Bits 10-0 provide a unique identifier for the processor; its interpretation is implementation dependent.

A.2.11 Default Message Handler Register (NIR23)

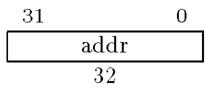


Figure 12: Default Message Handler Register Format

The default message handler (`dmh`) register is a 32b read/write register that specifies the address of the handler which is to be invoked when any message arrives while Active Messages are disabled (i.e. while `ame` is clear).

Since instructions must be word aligned, this register must contain an address whose two low order bits are zero. Otherwise, an address exception will occur when the CPU attempts to invoke the default message handler.

A.2.12 Outlet Handler Register (NIR24)

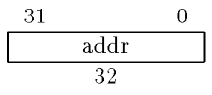


Figure 13: Outlet Handler Register Format

The outlet handler (`outh`) register is a 32b read/write register that specifies the address of the handler which is to be invoked when an outlet event occurs. An outlet event occurs when both the `op` (outlet pending) and `om` (outlet mask) bits are set, and low priority events are enabled. The `op` bit is set if and only if the network interface has room to accept another low priority message. Typically an outlet handler will prepare and send a message.

Since instructions must be word aligned, this register must contain an address whose two low order bits are zero. Otherwise, an address exception will occur when the CPU attempts to invoke the outlet handler.

A.2.13 Incoming Handler Register (NIR25)

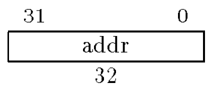


Figure 14: Incoming Handler Register Format

The incoming handler (`inh`) register is a 32b read-only register that holds the address of the handler specified in the incoming message. The contents of this register are only valid inside of a message handler. While this is redundant information when Active Messages are enabled, it is useful when Active Messages have been disabled (possibly for debugging purposes). In this case, a default message handler is always invoked when a message arrives; it can then use `inh` to deduce which type of message it actually was that arrived. (Also, it is possible to implement a more primitive type of message passing mechanism by disabling Active Messages and using the handler address field as a message type tag, which can be read from `inh`.)

A.3 Instruction Overview

Network Instruction Classes

There are a number of additional instructions for dealing with the network interface, divided into four classes:

- **Network Interface Register** instructions that read and write network interface registers.
- **Network Interface Branch** instructions which conditionally cause a CPU control transfer based on the condition flag in the network interface.
- **Send** instructions which cause the construction and transmission of an interprocessor message.
- **Handler Return** instruction, HRET, which causes the resumption of normal processing after an event handler has completed its work.

Network Instruction Formats

Instructions for reading or writing to the network interface registers use the standard MIPS `ctc2/cfc2` encodings.

Similarly, network interface branch instructions use the standard MIPS `bc2t/bc2f` encodings.

Since HRET has no modifiers or arguments, its encoding is straightforward.

The SEND instructions extend the MIPS ISA using COP2 opcode space. The general format of SEND instructions is shown in Figure 15.

31 26	25 24	23 21	20 16	15 11	10 6	5 0
COP2 010010	10	*n*	rt	rd	vd	SENDxx 1000xx
6	2	3	5	5	5	6

Figure 15: SEND instruction general format.

The *n* field specifies how many scalar words to include in the message.

The *rt* field specifies a register which contains the address of the remote handler for this message. The *rd* field specifies register which contains an implementation-dependent identifier of the destination node. The *vd* field specifies the vector register to be optionally included in the message. The length of the vector is obtained from the `vlr` (vector length register) in the vector unit.

The low two bits (shown as xx) determine the priority of the message and whether a vector register is included. Each different combination of these two bits and the *n* field leads to a different version of the SEND instruction. These are listed in Subsection A.6.

A.4 Network Interface Register Instructions

The network interface register instructions move values between the CPU registers and the network interface registers. These operations use the standard MIPS coprocessor register operations.

The results of these operations are unpredictable if the coprocessor register field is neither one of the valid network interface register numbers as listed in Table 1 nor one of the other registers defined in COP2 for exception handling or memory management.

CFC2 Move Control From COP2

31 26	25 21	20 16	15 11	10 0
COP2	CF	rt	nirs	
010010	00010			00000000000
6	5	5	5	11

Format:

CFC2 rt, nirs

Description:

The contents of the network interface's control register *nirs* are loaded into CPU register *rt*.

This operation is only defined when *nirs* is a valid coprocessor register.

Operation:

```
rt := nirs;
```

Exceptions:

Coprocessor unusable exception.

CTC2 Move Control To COP2

31 26	25 21	20 16	15 11	10 0
COP2	CT	rt	nirs	
010010	00110			00000000000
6	5	5	5	11

Format:

CTC2 rt, nirs

Description:

The contents of CPU register *rt* are loaded into the network interface's register *nirs*.

This operation is only defined when *nirs* is a valid coprocessor register.

Operation:

```
nirs := rt;
```

Exceptions:

Coprocessor unusable exception.

A.5 Network Interface Branch Instructions

The network interface status register contains a single condition flag, `nicond`, which is set and reset by the SEND instruction depending upon its success. A conditional branch may be performed on the contents of this flag using the standard MIPS branch-on-coprocessor instructions.

There must at least one instruction between the SEND instruction and a following network interface branch instruction which uses the `nicond` bit.

BC2T Branch on Network Interface True

31 26	25 21	20 16	15 0
COP2	BC	BCT	offset
010010	01000	00001	
6	5	5	16

Format:

BC2T offset

Description:

A branch address is computed from the sum of the address of the instruction in the delay slot and the 16b *offset*, shifted left two bits and sign-extended to 32 bits.

If the network interface condition flag is set, the program branches to the target address, with a delay of one instruction.

Operation:

```
// do next instruction;
if (nicond) goto label;
```

Exceptions:

Coprocessor unusable exception.

BC2F Branch on Network Interface False

31 26	25 21	20 16	15 0
COP2	BC	BCF	offset
010010	01000	00000	
6	5	5	16

Format:

BC2F offset

Description:

A branch address is computed from the sum of the address of the instruction in the delay slot and the 16b *offset*, shifted left two bits and sign-extended to 32 bits.

If the network interface condition flag is clear, the program branches to the target address, with a delay of one instruction.

Operation:

```
// do next instruction;
if (!nicond) goto label;
```

Exceptions:

Coprocessor unusable exception.

A.6 Send Instructions

SEND instructions specify the destination processor in a register, the message length in words (as part of the opcode), and a handler address in a register. If the low two bits of the handler address are not both zero, then the coprocessor exception "Handler Address Error" is raised. The data for the message is gathered from CPU scalar registers and/or a vector unit register.

The message which is formed includes a copy of the KU_c (current kernel/user) bit; this is used so that the handler on the receiving node executes in the same mode, kernel or user, in which the sending node was operating when the message was sent.

SENDn Send Message

31 26	25 24	23 21	20 16	15 11	10 6	5 0
COP2		n	rt	rd	vd	SENDn
010010	10	xxx				100000
6	2	3	5	5	5	6

Format:

SENDn vd, rd, rt

Description:

If the network cannot accept the outgoing message, `nicond` (the network interface condition flag) is cleared and the instruction is aborted.

If the network can accept the outgoing message, `nicond` is set, and the message is constructed and sent. The contents of *rt* are used as the address of the handler for the message. The contents of *rd* specify the destination processor for the message in an implementation-dependent way. The contents of the first n of `$4, $5, $6, $7, $29` are used as the data component of the message. Encodings with n greater than 5 are not valid instructions.

The message will have low priority.

Although this instruction may take multiple cycles, it is atomic in that interrupts or events cannot occur in the middle of its execution.

Exceptions:

Coprocessor unusable exception.

Coprocessor Exceptions:

Handler address exception.

SENDn.V Send Message w/ Vector

31 26	25 24	23 21	20 16	15 11	10 6	5 0
COP2 010010	10	n xxx	rt	rd	vd	SENDn.V 100010
6	2	3	5	5	5	6

Format:

SENDn.V vd, rd, rt

Description:

If the network cannot accept the outgoing message, `nicond` (the network interface condition flag) is cleared and the instruction is aborted.

If the network can accept the outgoing message, `nicond` is set, and the message is constructed and sent. The contents of *rt* are used as the address of the handler for the message. The contents of *rd* specify the destination processor for the message in an implementation-dependent way. The contents of the first n of `$4, $5, $6, $7, $29` are used as the scalar component of the message, appended by the vector data contained in *vd* (vector control register `vlr` is read to get the vector length). Encodings with n greater than 5 are not valid instructions.

The message will have low priority.

Although this instruction may take multiple cycles, it is atomic in that interrupts or events cannot occur in the middle of its execution.

Exceptions:

Coprocessor unusable exception.

Coprocessor Exceptions:

Handler address exception.

SENDnH Send High Priority Message

31 26	25 24	23 21	20 16	15 11	10 6	5 0
COP2 010010	10	n xxx	rt	rd	vd	SENDnH 100001
6	2	3	5	5	5	6

Format:

SENDnH vd, rd, rt

Description:

If the network cannot accept the outgoing message, `nicond` (the network interface condition flag) is cleared and the instruction is aborted.

If the network can accept the outgoing message, `nicond` is set, and the message is constructed and sent. The contents of *rt* are used as the address of the handler for the message. The contents of *rd* specify the destination processor for the message in an implementation-dependent way. The contents of the first n of `$4, $5, $6, $7, $29` are used as the data component of the message. Encodings with n greater than 5 are not valid instructions.

The message will have high priority.

Although this instruction may take multiple cycles, it is atomic in that interrupts or events cannot occur in the middle of its execution.

Exceptions:

Coprocessor unusable exception.

Coprocessor Exceptions:

Handler address exception.

SEND*n*H.V Send High Priority Msg. w/ Vector

31 26	25 24	23 21	20 16	15 11	10 6	5 0
COP2		*n*	rt	rd	vd	SEND*n*H.V
010010	10	xxx				100011
6	2	3	5	5	5	6

Format:

SEND*n*H.V vd, rd, rt

Description:

If the network cannot accept the outgoing message, `nicond` (the network interface condition flag) is cleared and the instruction is aborted.

If the network can accept the outgoing message, `nicond` is set, and the message is constructed and sent. The contents of *rt* are used as the address of the handler for the message. The contents of *rd* specify the destination processor for the message in an implementation-dependent way. The contents of the first *n* of `$4, $5, $6, $7, $29` are used as the scalar component of the message, appended by the vector data contained in *vd* (vector control register `vlr` is read to get the vector length). Encodings with *n* greater than 5 are not valid instructions.

The message will have high priority.

Although this instruction may take multiple cycles, it is atomic in that interrupts or events cannot occur in the middle of its execution.

Exceptions:

Coprocessor unusable exception.

Coprocessor Exceptions:

Handler address exception.

A.7 HRET Instruction

The HRET instruction is used to return from an event handler and resume processing where it was interrupted.

Assuming the `LPEE` and `EE` bits are correctly set (that is, they contain the same values as they did when the event handler began execution), the HRET instruction will correctly return to either normal code or to a low priority handler, whichever was interrupted by the event which was just handled.

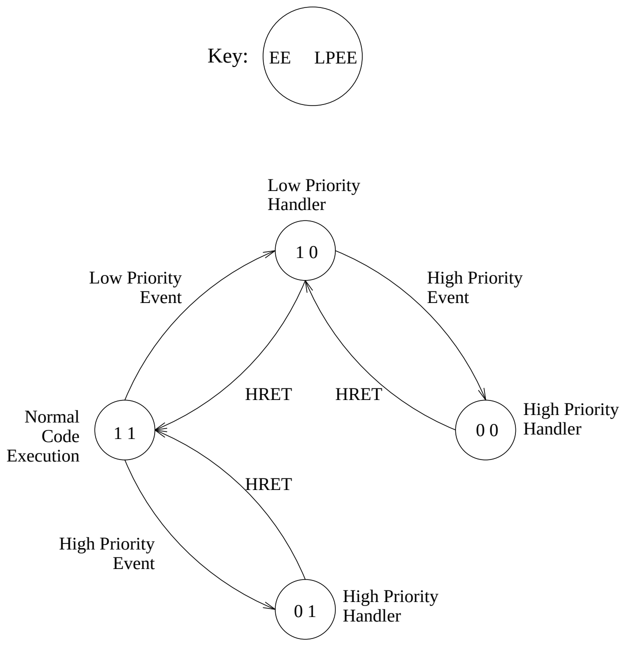


Figure 16: Event State Transition Diagram

HRET Return from Event Handler

31 26	25	24 6	5 0
COP2 010010	CO 1	0 000 0000 0000 0000 0000	HRET 011100
6	1	19	6

Format:

HRET

Description:

An HRET instruction is executed at the end of an event handler to restore the state of the CPU and network interface and resume processing where it was interrupted by the event.

The following actions are performed:

- The `EE` and `LPEE` bits are updated to reflect that one level of event handling is being exited; see Figure 16.
- If a low priority handler is being exited, the `onicond` (old network interface condition) bit is copied back into the `nicond` (network interface condition) bit.
- The previous registers (both the scalar set and `$vr1`) are swapped back into the CPU register address space.
- The contents of the LPEPC (low priority event program counter) or the HPEPC (high priority event program counter), depending on the priority of the event which has just been handled, are copied back into PC.
- If a message arrival handler is being exited, the KU_c (current kernel/user) bit is restored to the value it had before the handler was entered.

A.8 Extensions for Multicast

A.8.1 Overview

In some applications the operation of distributing identical data from one node to many other nodes is common. In these cases some hardware support for multicasting not only allows simpler software, but also allows for more efficient utilization of the network resources, leading to better performance.

In this subsection a simple extension to the network interface specification is described; it adds a mechanism for efficient multicasting. This mechanism does *not* provide fully general multicast capability; rather, it exposes enough of the underlying hardware's flexibility and performance so that software libraries may use it in whichever way is best suited for different communication patterns.

The multicast mechanism simply adds the option for a send to not only deliver the message to the destination, but to also deliver a copy of the message to every node along the path from the source to the destination. Thus no more network resources are consumed than for the simple one-to-one send.

A.8.2 Additional State for Multicast

Only one addition is made to the network interface registers. This is the addition of a bit in the network interface status register indicating whether or not an incoming message originated from a multicast send or from a normal (one-to-one) send. See Figure 17.

15	14	13	12 10	9 0
pri	mcst	0	slen	vlen
1	1	1	3	10

Figure 17: Network Interface Status Register Format (with Multicast Extension) – Low 16 Bits

All fields in the low 16 bits of the network status register (`nistat`) are read only. They are only valid within a message handler; they contain auxiliary information about the incoming message.

The `mcst` bit indicates whether the incoming message was multicast ($0 \leftrightarrow$ unicast, $1 \leftrightarrow$ multicast).

A.8.3 Additional Instructions for Multicast

The network interface register access instructions described earlier are sufficient for dealing with the multicast extensions. Likewise, no additional network interface branch instructions are needed. The only area where the instruction set has been extended is for the SEND instructions.

For each of the SEND instructions listed in Subsection A.6, a MCST version has been added which is identical except that it forks a copy of the message to each intermediate node on the path from the sender to the receiver. This path should be well defined for each implementation.

On the following pages, the MCST versions of the SEND instructions are described in detail.

MCST*n* **Send Multicast**

31 26	25 24	23 21	20 16	15 11	10 6	5 0
COP2 010010	10	*n* xxx	rt	rd	vd	MCST*n* 100100
6	2	3	5	5	5	6

Format:

MCST*n* vd, rd, rt

Description:

If the network cannot accept the outgoing message, the `nicond` (network interface condition) bit is cleared and the instruction is aborted.

If the network can accept the outgoing message, the `nicond` bit is set, and the message is constructed and sent. The contents of *rt* are used as the address of the handler for the message. The contents of *rd* specify the destination processor for the message in an implementation-dependent way. The contents of the first *n* of `$4, $5, $6, $7, $29` are used as the data component of the message. Encodings with *n* greater than 5 are not valid instructions.

The message will be received at the destination node and also by every intermediate node on the path from the source to the destination. The path is determined by the implementation.

The message will have low priority.

Although this instruction may take multiple cycles, it is atomic in that interrupts or events cannot occur in the middle of its execution.

Exceptions:

Coprocessor unusable exception.

Coprocessor Exceptions:

Handler address exception.

MCSTn.V Send Multicast w/ Vector

31 26	25 24	23 21	20 16	15 11	10 6	5 0
COP2		n	rt	rd	vd	MCSTn.V
010010	10	xxx				100110
6	2	3	5	5	5	6

Format:

MCSTn.V vd, rd, rt

Description:

If the network cannot accept the outgoing message, the `nicond` (network interface condition) bit is cleared and the instruction is aborted.

If the network can accept the outgoing message, the `nicond` bit is set, and the message is constructed and sent. The contents of *rt* are used as the address of the handler for the message. The contents of *rd* specify the destination processor for the message in an implementation-dependent way. The contents of the first n of `$4, $5, $6, $7, $29` are used as the data component of the message. Encodings with n greater than 5 are not valid instructions.

The message will be received at the destination node and also by every intermediate node on the path from the source to the destination. The path is determined by the implementation.

The message will have low priority.

Although this instruction may take multiple cycles, it is atomic in that interrupts or events cannot occur in the middle of its execution.

Exceptions:

Coprocessor unusable exception.

Coprocessor Exceptions:

Handler address exception.

MCSTnH Send High Priority Multicast

31 26	25 24	23 21	20 16	15 11	10 6	5 0
COP2		n	rt	rd	vd	MCSTnH
010010	10	xxx				100101
6	2	3	5	5	5	6

Format:

MCSTnH vd, rd, rt

Description:

If the network cannot accept the outgoing message, the `nicond` (network interface condition) bit is cleared and the instruction is aborted.

If the network can accept the outgoing message, the `nicond` bit is set, and the message is constructed and sent. The contents of *rt* are used as the address of the handler for the message. The contents of *rd* specify the destination processor for the message in an implementation-dependent way. The contents of the first n of `$4, $5, $6, $7, $29` are used as the data component of the message. Encodings with n greater than 5 are not valid instructions.

The message will be received at the destination node and also by every intermediate node on the path from the source to the destination. The path is determined by the implementation.

The message will have high priority.

Although this instruction may take multiple cycles, it is atomic in that interrupts or events cannot occur in the middle of its execution.

Exceptions:

Coprocessor unusable exception.

Coprocessor Exceptions:

Handler address exception.

MCST*n*H.V Send High Priority Multicast w/ Vector

31 26	25 24	23 21	20 16	15 11	10 6	5 0
COP2		*n*	rt	rd	vd	MCST*n*H.V
010010	10	xxx				100111
6	2	3	5	5	5	6

Format:

MCST*n*H.V vd, rd, rt

Description:

If the network cannot accept the outgoing message, the `nicond` (network interface condition) bit is cleared and the instruction is aborted.

If the network can accept the outgoing message, the `nicond` bit is set, and the message is constructed and sent. The contents of *rt* are used as the address of the handler for the message. The contents of *rd* specify the destination processor for the message in an implementation-dependent way. The contents of the first *n* of `$4, $5, $6, $7, $29` are used as the data component of the message. Encodings with *n* greater than 5 are not valid instructions.

The message will be received at the destination node and also by every intermediate node on the path from the source to the destination. The path is determined by the implementation.

The message will have high priority.

Although this instruction may take multiple cycles, it is atomic in that interrupts or events cannot occur in the middle of its execution.

Exceptions:

Coprocessor unusable exception.

Coprocessor Exceptions:

Handler address exception.

A.9 Events

Events asynchronously interrupt the ongoing computation and cause the invocation of a handler. In this way they are similar to standard interrupts, but the event mechanism is different from the interrupt mechanism in many ways.

There are currently three types of events defined: message arrival events, outlet events, and timelet events. Events are also classified as either high priority or low priority. The arrival of a high priority message is the only high priority event. Outlets, timelets, and arrivals of low priority messages are all low priority events.

In general, only one level of event handling is allowed. If while one event is being serviced another event occurs, the latter event will wait until the earlier event has completed its handling. This enforced using the `LPEE` (low priority event enable) bit in the network interface status word.

The exception to this is that a high priority event will preempt the handling of a low priority event. In this case two levels of event handling are allowed. To support these two levels of events, there are two different register event sets in addition to the normal register set (see Figure 1). A high priority event cannot preempt the handling of another high priority event. This is enforced using the `EE` (event enable) bit in the network interface status word.

In the case where two low priority events occur simultaneously, or when multiple low priority events are waiting at the completion of the handling of another event, a fixed ordering decides which event gets handled. Message arrival events get handled first, then timelet events, and finally outlet events get handled only when no other unmasked events are pending.

For all events, after handling has been completed, the previously ongoing execution is resumed by the execution of the HRET instruction (see Subsection A.7).

A.9.1 Message Arrival Events

A message arrival event is caused when a message has been completely received into the network interface. At this point several things occur as control is transferred to the handler:

- The address of the next instruction to be executed when normal processing resumes after the event has been serviced is stored in the LPEPC register if it is a low priority message or is stored in the HPEPC register if it is a high priority message. In the case that this instruction would be a branch delay slot, the address of the previous branch instruction is instead stored.
- If the message has low priority, the `LPEE` bit in the network interface status word is cleared, preventing further low priority events from occurring.
- If the message has high priority, the `EE` bit in the network interface status word is cleared, preventing *any* further events from occurring.
- The vector data component of the incoming message, if present, is transferred into the appropriate copy of vector unit register `$vr1`. This register is then swapped into the vector unit address space so that it is accessible (in agreement with the new setting of `EE` and `LPEE`).
- The scalar data component of the incoming message is transferred into the appropriate event set registers (either low priority or high priority) in the CPU, and these event set registers are swapped into the CPU register address space, specifically at addresses `$4-$7` and `$29` (see Figure 1). The normal copy of these registers cannot be accessed within the handler.
- Two event set temporary registers are also swapped into the CPU register space at ad-

dresses `$8` and `$9`. The contents of the normal registers `$8` and `$9` cannot be accessed. See Figure 1. The contents of the event set temporary registers are preserved across handler invocations.

- If the message has low priority, the `nicond` (network interface condition) bit in the network interface status word is saved into the `onicond` (old network interface condition) bit. (It is not necessary to preserve the `nicond` bit for a high priority message since a high priority event handler should not attempt to send any messages; if it does, it is responsible for saving and restoring `nicond`.)
- The KU_c (current kernel/user) bit is saved, then set or cleared according to whether the sending node was in kernel or user mode when the message was composed.
- If Active Messages are enabled, the address of the handler specified in the message is loaded into the PC so that control is transferred to the handler. If Active Messages are disabled, the contents of the `dmh` (default message handler) register are loaded into the PC.

A.9.2 Outlet Events

An outlet event is triggered when the `op` (outlet pending) and the `om` (outlet mask) bits in the network interface status word are both set, *and* low priority events are enabled. The `op` bit is set if and only if the network interface can accept a low priority message from the CPU.

The outlet mechanism facilitates background data transfers and eliminates the need to poll the `op` bit in the network interface status word. An outlet handler must send a message or disable outlet events; otherwise outlet events will occur endlessly.

When an outlet event occurs, several things happen:

- The address of the next instruction to be executed when processing resumes after the outlet event has been serviced is stored in the LPEPC register. In the case that this instruction would be a branch delay slot, the address of the previous branch instruction is instead stored.
- The `LPEE` (low priority event enable) bit in the network interface status word is cleared, preventing further low priority events from occurring.
- The low priority event set registers are swapped into CPU register address space at `$4-$9, $29` (the normal contents of these registers cannot be accessed within the handler). The contents of `$4-$7` and `$29` are unpredictable, while the contents of `$8` and `$9` are preserved across handler invocations. None of these registers have to be preserved by the handler, speeding the process of message composition. See Figure 1.
- The low priority event copy of `$vr1` is swapped into the vector unit address space, with unpredictable initial contents. Its contents do not have to be preserved, making it useful for composing vector messages.

- The address of the outlet handler, specified in the `outh` network interface register, is jammed into the PC so that control is transferred to the handler.
- The `nicond` (network interface condition) bit in the network interface status word is saved into the `onicond` (old network interface condition) bit.

A.9.3 Timelet Events

A timelet event occurs when both the `tp` (timelet pending) and the `tm` (timelet mask) bits are set, *and* low priority events are enabled. The `tp` bit gets set when the value in the `Count` register is the same as the value in the Timelet Compare (`timec`) register. The `tp` bit gets cleared either when a timelet event occurs or when the `timec` register is written.

When a timelet event occurs, the following things happen as control is transferred to the handler:

- The address of the next instruction to be executed when processing resumes after the outlet event has been serviced is stored in the LPEPC register. In the case that this instruction would be a branch delay slot, the address of the previous branch instruction is instead stored.
- The `LPEE` (low priority event enable) bit in the network interface status word is cleared, preventing further low priority events from occurring.
- The `tp` (timelet pending) bit is cleared.
- The low priority event set registers are swapped into CPU register address space at `$4-$9, $29` (the normal contents of these registers cannot be accessed within the handler). The contents of `$4-$7` and `$29` are unpredictable, while the contents of `$8` and `$9` are preserved across handler invocations. None of these registers have to be preserved by the handler, speeding the process of message composition. See Figure 1.
- The low priority event copy of `$vr1` is swapped into the vector unit address space, with unpredictable initial contents. Its contents do not have to be preserved, making it useful for composing vector messages.
- The address of the outlet handler, specified in the `timeh` network interface register, is

jammed into the PC so that control is transferred to the handler.

- The `nicond` (network interface condition) bit in the network interface status word is saved into the `onicond` (old network interface condition) bit.

A.10 Usage Notes

Network Behavior

Messages in the network are guaranteed to be delivered in order. If processor A sends a number of messages to processor B, it is not possible that processor B may receive those messages in a different order than they were sent from processor A.

The lowest level of flow control between two processors operates as follows: if messages arrive at a node faster than the node processes them, the messages will start to back up in various buffers along the paths) between the sending node(s) and the receiving node. Once the messages back up all the way to the network interface of a sending node, the sending processor will be prevented from sending any more messages: the `lpa/hpa` bit will be clear, and any SEND attempt will result in the `nicond` (network interface condition) bit being clear, indicating that the attempt failed.

This describes what occurs considering messages of only one priority level. Even if the low priority network is backed up with messages, high priority messages can still get through (unless the high priority network is also backed up).

Hazards

Because there might not be any bypassing between network interface control registers and the CPU registers, there is an exposed hazard involving data transfers between them. The following is a list of hazards which are due to transfer delays.

- For a CFC2 instruction, the data being transferred to the CPU register is not available until the second instruction after the CFC2 instruction.
- For a CTC2, any changes in the network interface state will not take effect until the third instruction after the CTC2 instruction.
- There must be one instruction between a SEND instruction and a BC2F/BC2T instruction which tests its success.
- There must be two instructions between a CTC2 instruction and a network interface branch instruction which tests a bit affected by the CTC2.

An implementation may implement bypassing between network interface control registers and the CPU registers, in which case the above hazards to not exist.

Typical Send Sequence

The availability of the network may be polled by looking at the appropriate bit in the network interface status register. Once it is determined that the network is available, a SEND can be attempted. In general, the success of the SEND must still be checked using the BC2T/BC2F instructions. This is because an asynchronous event or interrupt may occur between the poll and the SEND, and the invoked handler may execute a SEND of its own, causing the network to become unavailable.

If it is known that the SEND will likely succeed, the SEND can be tried directly without polling the network first. However, a failed SEND / check for success sequence will likely consume more cycles than the four required for a poll.

Between a SEND and the check of success, an event or interrupt may occur. If this event or interrupt performs a SEND, the `nicond` bit will be overwritten. In the case of a low priority event, the `nicond` bit is automatically saved to `onicond` at the beginning of the event handler and restored at the end of the handler, so the information is not lost. However, an interrupt or high priority event handler which performs a SEND will have to preserve the nicond bit by reading and saving the network interface status word before the SEND and then restoring it after the SEND.

Figure 18 shows a simple sequence showing a poll / send / check for success sequence.

```
         ...
         CFC2     $1, $nistat    % get nistat
         NOP
         %---shift lpa bit to sign bit---
         SLL      $1, $1, 14
         %----branch if sign bit is set----
         BLTZ     $1, $sendit
         %----otherwise continue----
         ...

$sendit: <prepare data in $4-$7, $29>
         <move address of handler into $8>
         <move identifier of destination into $9>
         <load vector data into $vr4>
         SEND5.V  $8, $9, $vr4
         NOP
         BC2F     $send_fail
         ...
```

Figure 18: Example showing some network interface instructions

Initialization of Event Set Registers

In general, the event set registers are only accessible from within event handlers. However, it is often useful to initialize event set registers $8 and $9 from within the main code segment prior to the invocation of any handlers. To facilitate this, we have made the visibility of the registers controlled by the setting of the LPEE/EE bits. These bits are normally set and reset as side effects of entering and exiting event handlers, but they can also be changed manually (by writing to the network interface status register) from normal code.

LPEE and EE control the visibility of the event set registers as follows:

EE	LPEE	Register Set Visible
1	1	Normal set
1	0	Low priority event set
0	1	High priority event set
0	0	High priority event set

Note that when the low priority event set is visible from normal code, low priority events are disabled. When the high priority event set is visible from normal code, all events are disabled.

Active Messages

For the background, motivation, and advantages of Active Messages, refer to

> Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser. "Active Messages: a Mechanism for Integrated Communication and Computation", in *Proceedings of the* 19^{th} *International Symposium on Computer Architecture*, May 1992.

To quote from this paper,

> Active Messages is an asynchronous communication mechanism intended to expose the full hardware flexibility and performance of modern interconnection networks. The underlying idea is simple: each message contains at its head the address of a user-level handler which is executed on message arrival with the message body as argument. The role of the handler is to get the message out of the network and into the computation ongoing on the processing node. The handler must execute quickly and to completion.

Some restrictions are placed on what handlers are allowed to do, for performance and deadlock avoidance reasons. Handlers should not perform serious computation; they should perform operations on the order of placing data into buffers, queuing up work for the primary computation, or replying to a remote memory fetch.

For request-reply exchanges such as "get" memory operations, the request must be a low priority message and the reply must be a high priority message in order to avoid deadlock. In general, a low priority event handler is not allowed to loop waiting to send a low priority message (although it is allowed to loop waiting to send a high priority message). A handler for a high priority message is not allowed to wait to send any kind of message.

As an example, consider the typical usage of outlet events: background data transfer. It is desirable to use low priority transfer for these large data transfers, so that they do not interfere with time-critical communications such as barrier synchronizations. In fact, outlets are designed for sending low priority messages, since they are triggered by the low priority network being willing to accept an outgoing message. But since an outlet handler is a low priority handler, isn't it a bad idea to send a low priority message from within it? No, not as long as the processor doesn't busy wait while trying to send the message. The typical outlet handler should look like this (notice that there is no looping):

```
attempt low priority send of data block
was it successful?
        yes: update pointer or queue
        no: don't update state
return from handler
```

Usually the send will succeed, since what set the outlet pending bit in the first place was that the network interface had room to accept another message.