VLSI Design of a Network Interface Processor

Jeffrey Rothman

Department of Electrical Engineering and Computer Science
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

Abstract

An important piece of parallel computer systems is an interface between the processing node and the communications network. In many systems, this interface is largely handled by software. This document describes a general purpose interface processor, which performs requests from the network. This component can also be used with the Fluent multiprocessor system. This design was motivated by the need of a fast interface for Fluent, but can be used in many network applications.

Acknowledgements

I would like to thank my advisor Abhiram Ranade for his advice and direction in the research and his comments for this report. I would also like to thank David Cross, Jim Hunt, and other members of the Fluent group for their help and comments. Thanks also to Andrea Casotto for his efforts in helping me with the Octtools.

VLSI Design of a Network Interface Processor

Jeffrey Rothman

Department of Electrical Engineering and Computer Science
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

1. Introduction

In the past, improvements in material processing would lead to increase performance of computer processing power. Now, the rate of increase of processing power for sequential processors has been slowing down due to limitations of physics. One solution to this dilemma is using many processors working on the same problem in parallel. Along with this solution comes the problem of communications between these processors.

Communications between processors can be broken into two basic kinds of operations: data transfer, and synchronization. In some systems, the processors operate in lock-step on the same instruction stream, each processor working on its own data. This model is referred to as single-instruction, multiple-data (SIMD). Since all processors work on the same instruction at the same time, synchronization is implicit. In other systems, each processor has its own instruction stream as well its own data. This is referred to as multiple-instruction, multiple-data (MIMD). The processors in the MIMD model must explicitly communicate to be able to synchronize.

Data transfers between processors can be accomplished in many ways. Processors can use shared or distributed memory. The processors can communicate by sending messages to other processing nodes, or by reading and writing to specific memory locations. Communications can take place through a network, which can have a variety of forms.

The network consists of processing node with some kind of set pattern of wiring between them. For the Fluent Multi-processor System, which this report partially describes, the network is a radix-2 butterfly network. At either end of the network are processing nodes, which consist of a Central Processing Unit (CPU), a memory, a memory cache for the CPU, and a communications processor which interfaces with the network (Figure 1).

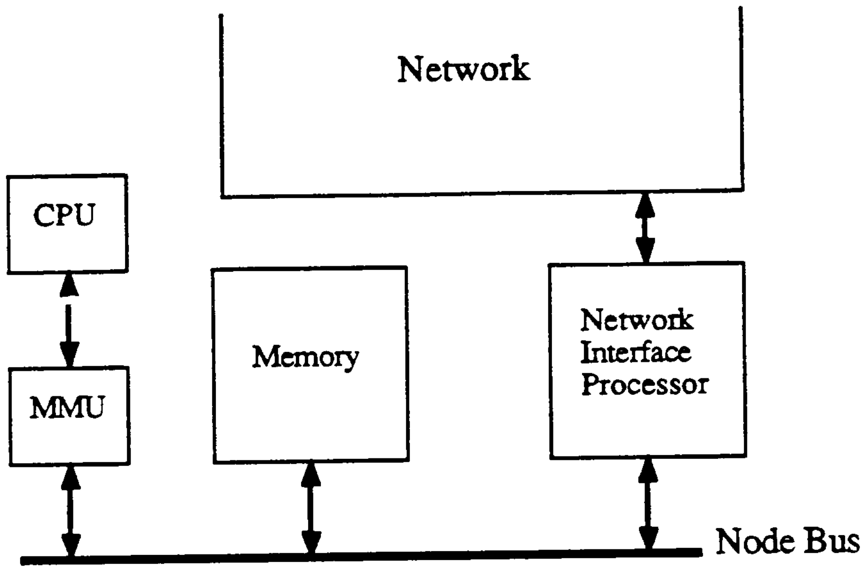

Figure 1. A Fluent Processing Node

We designed a fast network interface processor (NIP) to aid in communications between processors over a network (Figure 2). It can support shared memory and message passing models of computation, and does so with little overhead to the CPU and the node bus. It can also be used with the Fluent multi-processor system (Appendix A), and handles a few extra primitives necessary to perform that task. This chip has the ability to load node memory at boot time, eliminating the need for read only memory (ROM) on each node. It can also detect errors in messages and tries to minimize error impacts on the whole system.

The NIP consists of two major components: the front-end and the back-end (Figure 2). The front-end of the NIP monitors the node bus for commands to perform in the network, generates the appropriate message, and inserts it into the network. It also receives replies from the network and installs them in local memory. The back-end of the NIP receives requests from the network, performs the operations required, and sends back a response, if appropriate. This report describes the VLSI implementation of the back-end of the NIP chip. The implementation of the front-end of the NIP is described in more detail in a companion report [HUNT].

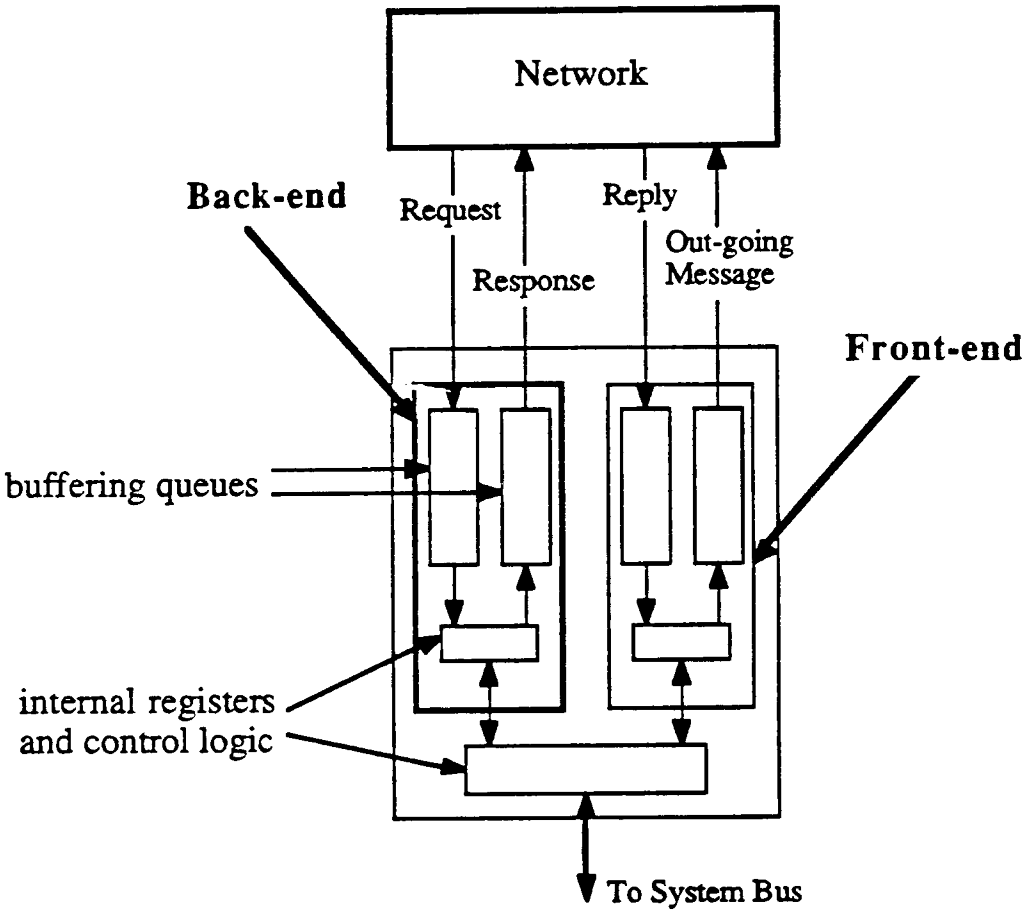

Figure 2. The Network Interface Processor

In section 2, the operations that the NIP supports are described. Section 3 contains an overview of the back-end of NIP. Sections 4 provides the details of the implementation of the back-end of the NIP. Section 5 describe the special message buffers that the NIP manages. Section 6 covers the error handling capabilities of the NIP and discusses loading node memories at boot time. Section 7 discusses how the whole back-end was tested, and section 8 provides the layout size and performance information. Section 9 compares the NIP with existing parallel computers. Section 10 is a conclusion of the design report, and a summary of the results. Appendix A is an explanation of the Fluent multi-processor system, for which the NIP was designed. Appendix B describes the different programming models the Fluent system is capable of emulating.

2. Detailed Overview of NIP Operation

There are six basic classes of operations that a processor can execute for which the NIP performs network and shared memory operations. Each processing node has the entire shared memory mapped into its local memory. A processor can manipulate shared memory by performing a memory write into a local memory location corresponding to the shared memory region. The front-end of the NIP monitors the node bus, and upon recognizing shared memory addresses on the bus, takes actions to send an appropriate message through the network. The front-end encodes a message with a destination, an opcode, and data. The message traverses the network, and is received and processed by the back-end of the NIP at the destination node. For some instructions, the processing is finished. For a read operation, data will be put into a reverse network by the back-end, traverse the network, and be written to memory by the front-end.

2.1 Processor-NIP Interface

These are the six kinds of operations the processor can request of the NIP:

(1) Read *remote address, local address, number*: The READ operation specifies a remote address to read from in shared memory, and the number of words to read. The result of the read is sent back across the network and written to the local address. The current implementation of the NIP can read one, two, or four words (32-bits per word) from shared memory. The word in local memory eight bytes after the local address is used to signal the processor that the data has returned from remote memory.

(2) Write *remote address, local address, number*: The WRITE operation reads one, two, or four words from local memory, and sends them across the network to a remote location in shared memory.

(3) Multi-prefix *remote address, local address*: There are two types of multi-prefix operations: ADD and MAX. In both types of operations, a word is read from the local address, and sent across the network The data from the remote location is read and sent back to the network. For the ADD operation, the data from the local address is added to the data from remote memory, and stored back at the local address. For the MAX operation, the data from the local address and the data from shared memory are compared, and the maximum of the two are written back to the local address.

(4) Broadcast *remote offset, local address, number*: The BROADCAST is either a Write or Notify operation to all the nodes in the network. Associated with the BROADCAST are one, two, or four words of data. The data is read from the local address. For a Write operation, the data is written to an offset within the shared memory located on each node, including the originating node. For the Notify, the data is put into a special buffer reserved for message passing.

(5) Notify *remote node, local address, number*: The NOTIFY operation is how messages are passed between processing nodes The *number* words of data associated with the operation, are read from local memory and set to a remote node. The data is put into a special buffer on the remote node, which is jointly managed by the remote node's NIP and processor. These messages are a way of specifying operations that are too complex for the NIP to handle, and the message gets passed to the processor, which will handle the operation. Some of the possible uses for the Notify message are communications between processes, thread (light weight process) creation, or a remote procedure call. This operation is how message passing operations get performed in the system when an alternative to shared memory operations is required.

(6) Synchronization Operations: Several kinds of synchronization operations are supported. The back-end recognizes these operations, but passes them on to the

front-end to be acted upon. These operations are discussed in more detail in the appendicies.

2.2 Message Formats

The front-end converts the requests from the processor into appropriate message packets, which are sent across the network. Messages that the back-end of the NIP can receive from the network consist of two address packets (except for synchronizations), and up to eight data packets. The operations requested of the NIP are encoded in one or two 17-bit address packets (Figure 3). The upper two bits of the network packet encode the type of operation, and the 15 lower bits are used for address information. For all other operations, except synchronization, another packet is necessary to completely specify which

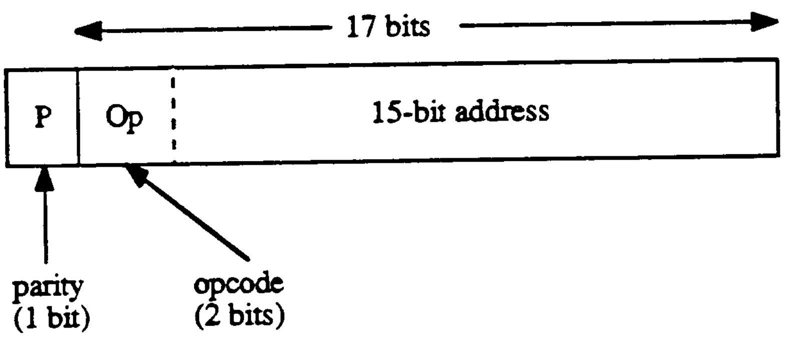

Figure 3: Components of a Network Opcode

operation is to be performed (Table 1). The complex message format was used to minimize the amount of traffic through the network, and to keep the pin count as low as possible on the chips we needed to design [Cross]. This message format allows:

(1) Efficient pipelining of message through the network.

(2) The synchronization message can be sent with only one packet.

The highest two of the four bits arrives with the first address packet. To try and minimize the number of packets in the network, it was decided that an EOS should only take one address packet to specify, since there will be many EOS messages in the network. Since a synchronization does not need a destination, the address bits are used to specify the kind of synchronization, which is described in Appendix A.

For the READ and WRITE operations, the high two bits specify the operation, and the low order bits specify the size of the operation.

The miscellaneous operations are specified by 10 as the high order two bits. Two of the combinations are used for Multi-prefix operations, and the last combination is used to specify BROADCAST.

Three of the 13 kinds of messages are GHOST messages. These are used in the network to aid in sorting, but are ignored by the back-end. These messages all have 11 as the low order two bits.

Opcode	Operation
00 00	Read 1 Word
00 01	Read 2 Words
00 10	Read 4 Words
01 00	Write 1 Word
01 01	Write 2 Words
01 10	Write 4 Words
10 00	Multi-prefix Add
10 01	Multi-prefix Max
10 10	Broadcast
XX 11	Ghost
11	End of Stream

Table 1: Network Operation Codes

Note that in table 1 there are no opcodes to represent the Notify operation. This is due to the small number of opcodes available to be used for encoding operations. The Notify is actually a Write (or Broadcast) to the remote address that corresponds to the base of the Notify buffer. This is discussed in more detail in section 5.

3.0 Overview of the Back-end

The back-end consists of four distinct kinds of state machine functional units which cooperate in accepting and performing commands from the network (Figure 4). The four basic units are input finite state machine (infsm), the back-end controller (befsm), the output finite state machine (outfsm), and a circular queue. The missions of each of the pieces of the back-end will be described in the order that information is processed through the network.

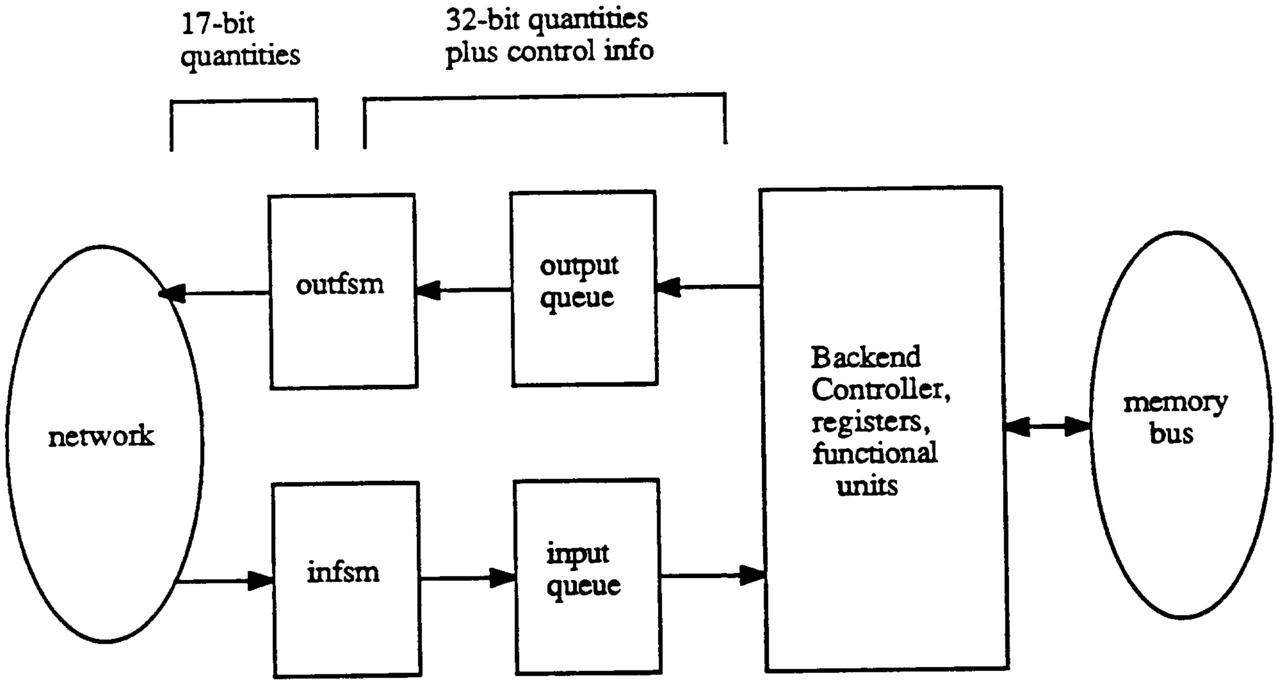
17-bit quantities
32-bit quantities plus control info
network
outfsm
output queue
infsm
input queue
Backend Controller, registers, functional units
memory bus

Figure 4: Structure of the Back-end

3.1 Infsm

The input FSM accepts 17-bit data packets (plus parity) from the network, and pairs the packets together to form 32-bit data packets with associated 5 bits of opcode. It also does a partial interpretation of the incoming messages to weed out the simpler types, such as ghost messages and synchronization messages. The synchronization message is the only type of message that comes in a single 17 bit quantity. The infsm informs the front-end of the arrival of a synchronization message, and resets to accept another packet.

Ghost messages can also arrive at the network input to the back-end. Ghosts are used to keep the network "fluent," but are not to be processed. The infsm recognizes ghosts, and kills them by accepting them and throwing them away.

The infsm refuses to accept packets under two conditions:

(1) Parity error: If the incoming parity bit does not match the parity bit the infsm generates, then it refuses to accept a message. However, the address packets must be accepted in pairs, to prevent deadlock (unless the message happens to be an EOS) [Cross].

(2) Full input queue: Incoming packets may be ignored if the queue is full. From the network side, it doesn't know what is keeping a packet from being accepted, so it just keeps re-transmitting until the infsm accepts it.

Once data is accepted by the infsm, it is passed into an input queue, from which the back-end controller eventually reads it.

3.2 The Back-end Controller

The back-end controller is the heart of the back-end. It interprets messages from the input queue and performs the required actions. The controller consists of two parts: the back-end finite state machine (befsm) and the datapath. The befsm interprets the four bit operation code in incoming messages, and performs combinations of memory and functional unit operations in the datapath. The controller consumed the most design time of any of the different parts of the NIP.

Data flows from the input queue onto a bus, from which it is routed by the befsm to different internal registers, depending on the operation to be performed. All operations require a memory access. If the operation is a read or multi-prefix operation, then data

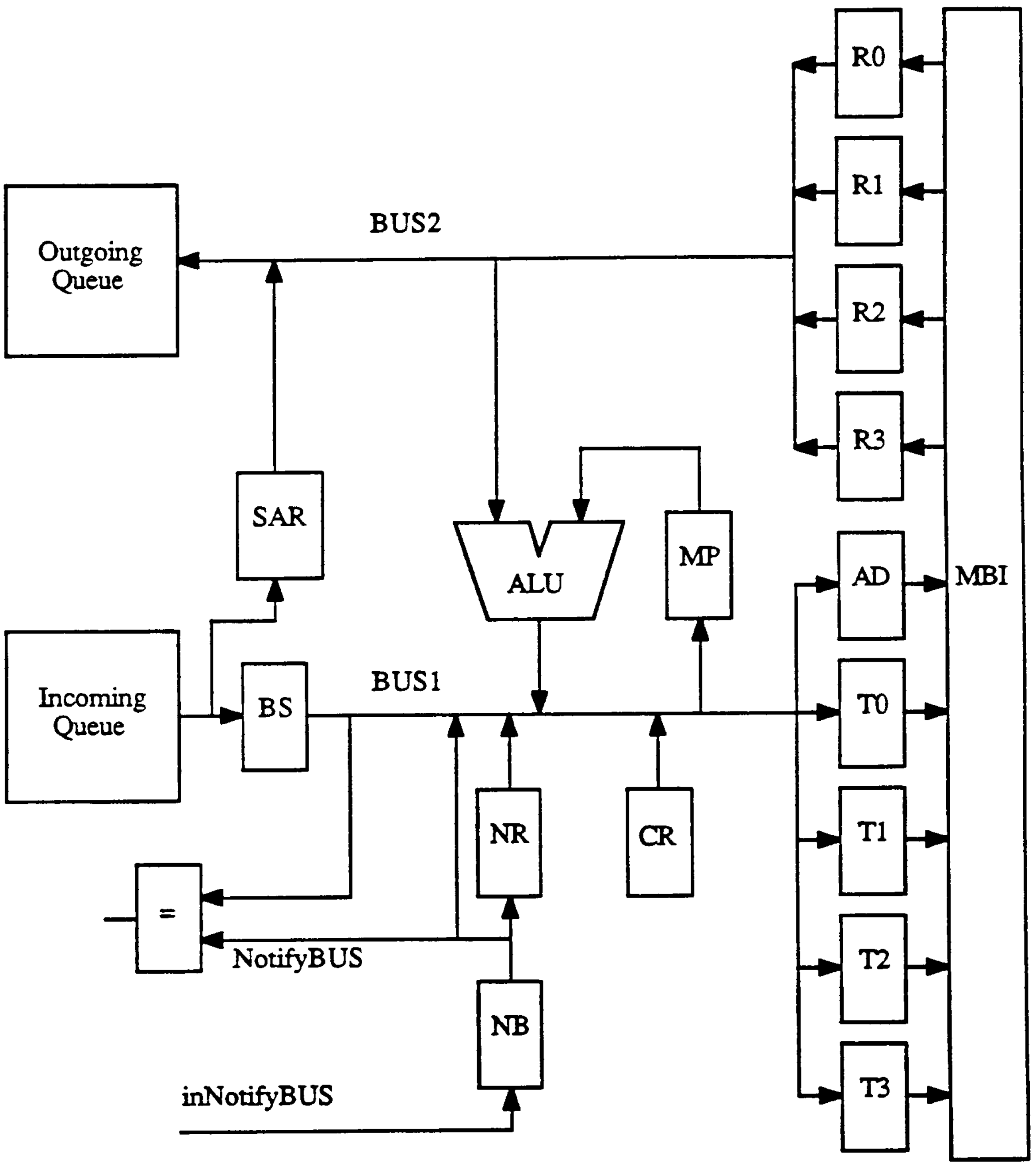

Figure 5: Back-end Control Data Path
See Table 2 for the figure legend

Abbreviation	Long Form
MBI	Memory Bus Interface
ALU	Arithmetic Logic Unit
SAR	Saved Address Register
CR	Condition Register
MP	Multi-prefix Register
NR	Notify Register
NB	Notify Base Register
BS	Barrel Shifter
T0 - T3	Transmission Registers
R0 - R3	Receiving Registers
AD	Address Register
=	Equivalence Unit

Table 2: Legend for Figure 5

from memory is routed to the output queue, and from there throught the output state machine to the network. The operation of this unit is described in more detail in section 4.

3.3 Outfsm

The outfsm is the simplest unit in the back-end. Its job is to take a 33-bit word (32 data bits plus a chaining bit) from the output queue and break it into 16-bit packets for transmission over the network. It also generates a parity bit and a chaining bit, and deals with re-transmissions.

4.0 The Back-end Controller

The finite state machine of the back-end controller (befsm) has to perform four basic types of operations, with variations on each kind (Figure 5 and Table 2). To do so, it has to manipulate a collection of registers and functional units. The four types of operation are read, write, multi-prefix, and notify. The opcodes used to internally encode these operations are in the appendix. The logic for the befsm was entirely written using bdsyn.

4.1 Read

The read operation has three variations: one word, two words, and four words. The address is loaded over BUS1 into the AD (Address) register through the barrel shifter (BS), the unshifted version of the address is put into SAR, and the memory bus interface (MBI) unit is started up. When the read has finished, the results are put into the outbound queue, followed by the unshifted address packet. The befsm just has to make sure the outbound queue has room for each word as they are removed from the receiver registers (R0 - R3).

4.2 Write

The write operation writes one, two, or four words to memory. The befsm loads the data words as they arrive in the transmitter registers (T0 - T3). When the proper number of words have been written into the registers, the MBI is informed of the operation, and the befsm waits until the MBI is done before proceeding to the next operation in the queue. The write operation does not require any information to be returned to the network.

4.3 Broadcast

A broadcast is treated by the back-end in a manner similar to be a write. There were not enough bits in our four-bit message opcode to encode all the broadcast lengths, so the befsm depends on the chaining bits to determine the number of words to be written. If the broadcast is a write to the base of the Notify buffer, then the Notify operation is performed.

4.4 Multi-prefix

This type of operation requires several accesses to memory in a read-modify-write (RMW) cycle. The two variations are ADD and MAX. The address packet is followed by one data packet. A RMW is started using the address packet, and the memory bus is held until the appropriate operation can be performed by the ALU and written to the T0 register, from which the value is written into memory and the RMW cycle completes. The value that was read from memory is written into the outbound queue over BUS2, followed by the unshifted address packet whenever the queue has room.

4.5 Notify

The Notify operation is a fairly complex operation, and takes up most of the code in the befsm specification. The Notify message is a message from another processor that writes data into a buffer rather than a specific memory location. We view it as a fixed length remote procedure call, but the hardware puts no constraint on how the messages are treated; it just delivers them to the buffer. A Notify message is four words long. One possible message interpretation consists of a header and three parameters (Figure 6).

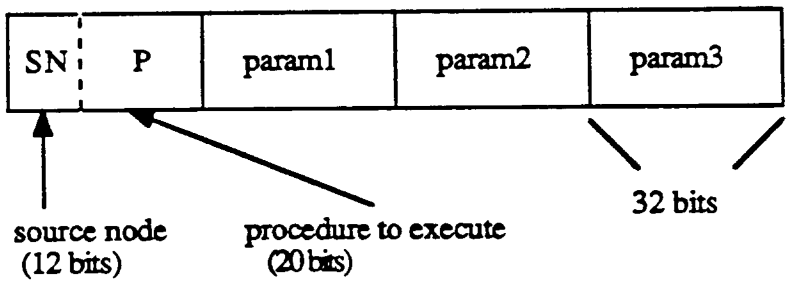

Figure 6: Possible Interpretation of a Notify Message

The header consists of a source node, which is the ID of the node that sent the notify message. The procedure field specifies an index into a table of procedures to execute. The three parameters are parameters for the procedure to use. If there are up to three arguments needed for a procedure, then these parameters are the actual data for the invocation. If more than three arguments are needed, then *param1* is a pointer to an argument area on another node, and *param2* is a counter of the number of arguments that need to be read. The procedure would then have to initiate the appropriate number of READ messages to get the data back for its execution. We expect the procedure to know whether it takes three or less parameters, or needs to make accesses to remote memory. If there are less than three arguments, then the unused parameter fields still exist and will be garbage, so the procedure must ignore them.

A Notify message is recognized as a WRITE or BROADCAST to a special memory location. The special memory location is whatever the base of the Notify buffer happens to be. When a message comes in, it is compared to the base of the Notify buffer, and is determined to be a Notify message if it matches. In that case, it is put into a slot in the buffer, and not sent to the Notify address. There is also a special Notify, called an Interrupting Notify, which causes the CPU to be interrupted and informed immediately of its arrival. The back-end writes a status word to memory, as well as the address of the Notify, and it may be processed at the CPU's convenience. The Interrupting Notify is detected as a write to the memory location one byte above the base address of the Notify buffer. This would cause a bus error if attempted, but the back-end intercepts the address of the message, and handles the storage to memory correctly.

4.5.1 NIP Notify Processing Specifics

As the address packet is written into the AD register, it is compared to the NB register. If the values are equal, the AD register is reloaded with the value in the Notify Register (NR) on the next cycle. Since the Notify messages are expected to be in the minority of messages received, it is more efficient to load the address into the AD register before determining if a Notify operation is in the works. Once the AD register is set up, the data associated with the Notify is written into the transmission registers.

At this point, the befsm must determine what kind of Notify is to be executed. If the address in the address packet is one higher than the NB value, then the Notify is the interrupting type. The normal case, which is a non-interrupting Notify, will be discussed first.

4.5.2 Non-interrupting Notify

If this is the type of Notify, then the MBI can be informed that it has a write to perform. The next step is to increment the NR by four, using a specially designed increment-by-four register. The NR is then compared to the NM (Notify Middle) if Notify queue 0 is being used, otherwise it is compared to NE (Notify End) for Notify queue 1. If the comparison shows equivalence, the end of the Notify queue has been reached. A status word is written into the Condition Register (CR). The AD register is loaded with the NE value. T0 is loaded with the CR value, and T1 is loaded with the NR value, so the CPU will know which queue to read. The MBI performs a write of the information, and the CPU is interrupted, so that it looks at the NE location for the reason it was interrupted. A bit is set in the NIP that reminds it that it can not perform another Notify operation until the CPU turns the other queue over to the NIP for writing. The NIP goes on to the next operation.

4.5.3 Interrupting Notify

An interrupting Notify is written to the location the NR points to in memory, and the CPU is immediately interrupted. The AD register has been load with the NR value before the befsm discovers what kind of Notify is to take place, but that does not cause a problem. CR is loaded with an condition vector indicating that an interrupting Notify is taking place. The MBI is activated, and when finished, a few registers are reloaded. The AD register is loaded with NE. The CR register is transferred to T0, and the NR value is transferred to T1, so the CPU knows where to look for the Notify. The MBI is activated again, and then the CPU is interrupted. No further Notify operations can take place until the CPU has acknowledged reading the interrupting Notify. Since the Notify has been read from the slot the NR points to, that space is re-used to write the next Notify message.

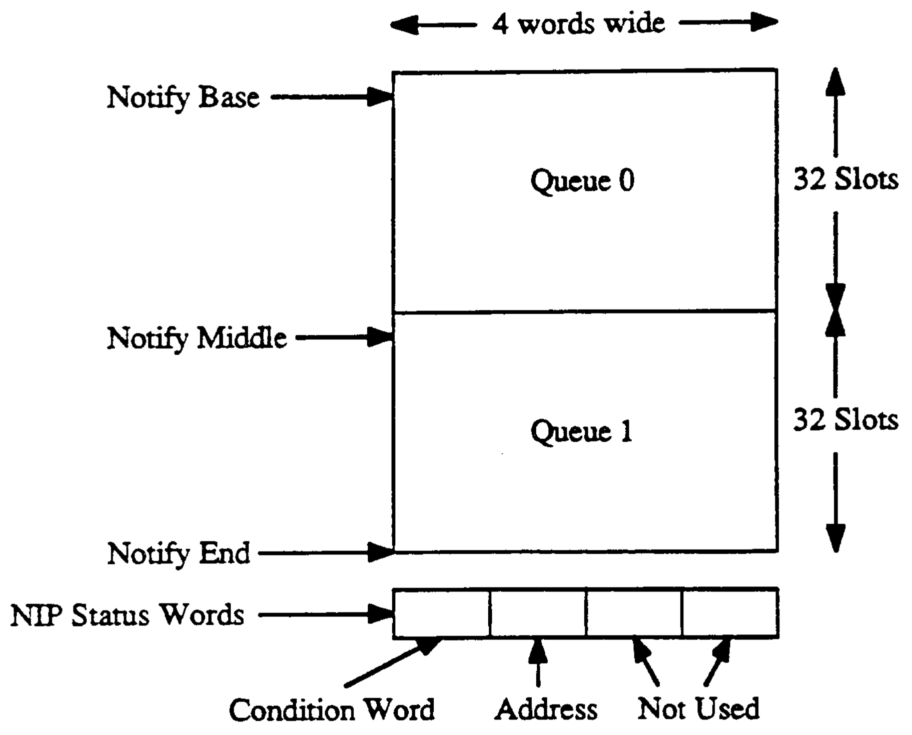

Figure 7: Structure of the Notify Buffer

5.0 Notify Buffer Management

The Notify buffer is really a pair of queues which are jointly managed by the NIP and the CPU (Figure 7). Each queue holds 32 slots for Notify messages, for a total buffer size of 1 kilobyte. At any given time, one queue receives messages from the network (through the NIP) and the other queue can be read by the CPU. When the back-end determines that its queue is full, it interrupts the CPU and informs it that the buffer is full. After the CPU tells the back-end to switch queues, the functions of the queues are swapped (i.e. the receiving queue is available to be read by the CPU), and the back-end uses the other queue as the message buffer. The CPU can also request that the queues be swapped if it needs more tasks to work on. The back-end can write status information to memory and interrupt the processor to acknowledge the switch. Part of the status will be the address of the next slot in the queue that was to be written with the next Notify message. The processor can figure out how many messages are available to be read based on that information.

When either the NIP or the CPU request a switch, a condition word is written to the word located right past the end of the second queue. The next word in memory contains the address of the word after the last valid message in the queue. The status word is used to inform the CPU of certain conditions within the NIP. Along with the address word written into memory, there is enough information for the NIP to communicate the status of the Notify buffers. The conditions that the back-end tells the CPU about are NIP errors, and Notify buffer status. The different conditions are:

(1) Data Error. The NIP back-end controller sees data packets when it doesn't expect to see data.

(2) Write Error. The number of words specific by the opcode are inconsistent with the number of words received by the NIP.

(3) Queue Full. The NIP tells the CPU which queue is full, and awaits release of the other queue to write into.

(4) Interrupt Notify. The address specified in the address area of the status words points to a high priority message in the queue. After the CPU acknowledges, this location will be overwritten.

(5) Queue Available. After the CPU requests the NIP to switch queues, the NIP informs the CPU that it has switched queues, and tells the CPU from which queue it had switched (redundant information to help detect errors).

Upon detecting an error condition, the CPU executes routines to rectify the situation, or to signal to the user that some kind of error has occurred. The other information in the status word is about the Notify queues.

The NIP maintains three pointers to indicate the position of the queues in memory, as well as a pointer to the next slot to be filled in memory. The base pointer can be set to point to 1 K boundaries, which determines where the Notify buffer is located in node memory. This location can be set by the CPU. The other two pointers are offset from the base pointer; 512 bytes for the middle pointer, and 1024 bytes for the end pointer. After each Notify message, the NIP increments the slot pointer, and tests to see if the end of a queue has been reached by comparing it with the middle or end pointer. If the end of a queue has been reached, the NIP fills the status words with a condition vector and interrupts the processor. The next incoming Notify message will be stalled until the CPU signals that the other queue is ready to be written to.

When an Interrupting Notify is received, it is put into the slot the next Notify would be put in, but the address of that location is written to the Address area, and the status word indicates that there is a special Notify. The CPU is interrupted, and any incoming Notifies have to wait until the CPU acknowledges. The next Notify message will then overwrite the location of the interrupting Notify. The interrupting Notify has supposedly been executed or removed before the CPU acknowledges, so it should be removed/overwritten from the queue.

The two queue method of buffering Notify messages was chosen to minimize the amount of accesses to the system bus. We considered several different schemes, but the two queue method seemed the best. One method would be to have one buffer, and the NIP could set a bit in some memory location to indicate that the word had a message in it. The CPU could clear a bit when the corresponding message had been processed. The problem with this method is that each write to a buffer takes three memory operations: a read to find a location to write the message, a write of the message, and a write to set a bit. Each successful read by the CPU would take two cycles, one to see if a message were available, and another read to get the message. For each successful read operation, there could be many failed reads, each of which would take a memory bus cycle.

A slightly better method would be to use information in the Notify message slot to tell if the slot were full. If an all zero message were disallowed, then the NIP could read the words in the message to see if the slot were empty, and the CPU could clear the Notify slot once it had read the message. However, this method has the disadvantage of requiring an extra access on each read and each write. The strategy used in the Fluent system minimizes the overhead number of accesses to the memory for each message by establishing a handshaking protocol between the CPU and the NIP which is not that much more sophisticated than the alternative methods, but with many fewer accesses to the memory.

5.1 CPU Requested Queue Switch

Under some circumstances, the CPU may request that the NIP switch queues, so that the CPU can process the messages. At the end of each operation, the befsm checks if a bit in the NIP has been set to indicate that a switch should take place. If so, the befsm immediately jumps into the middle of the code for a Notify operation that switches queues. The CR register is loaded with a vector indicating which queue it had been using (which the CPU should know anyway). The AD register is loaded with the NE register value. T0 is loaded with CR, and T1 is loaded with the value in NR. The MBI performs a write, and the NR is loaded with NB or NM, depending on which queue it is switching to.

6.0 Error Handling and Booting in the NIP Back-end

This section describes special functions of the NIP.

6.1 Error Handling

To help in detecting errors, the back-end has a significant amount of redundancy built in. When data is put into the input queue, two bits of the associated opcode are used to direct the data to the appropriate transmission register. The controller also keeps a counter of the number of data words associated with the instruction being processed. If there is a mismatch between the counter and the opcode, the NIP enters an error mode. During the error mode, the rest of the instruction is discarded, and the NIP informs the CPU of the error by writing the error vector to the status words, and interrupting the CPU. The CPU should invoke the operating system to deal with the failure.

The highest order bit of the five bit opcode in the queue is the complement of the chaining bit used in the network to pass data. This provides a third way of making sure that the opcode in the queue and the counter in the befsm are correct. In the case of a broadcast, the highest bit provides the only way that the befsm can know how much data to expect. The broadcast instruction can be one, two, or four words, but there is no way of encoding the information in the opcode, so only the chaining bit in the data packets can be used to know the amount of data. All the befsm knows is that that there can not be more than four words of data. If there is no data, or too much data, the befsm enters data mode and clears the data from the queue until it finds a valid instruction to perform.

Parity is used for the transmissions between the network and the back-end, as well as throughout the network. If the parity calculated by the data receiver does not match the transmitted parity, then the receiver does not accept the data. The transmitter will keep sending the data until accepted by the receiver. The receiver can also refuse data if the input queue is full. The transmitter does not know the reason the data was refused, so it can just assume some kind of error occurred, and keep transmitting.

6.2 Booting the System

Because the NIP uses DMA accesses to local node memory, it is possible to load the node memory with the CPU in the reset mode. When the reset signal is turned off, the CPU will start executing instructions at some predefined location. By clever use of the NIP, the boot code for the CPU can be loaded over the network, and it will not be necessary for a ROM (read only memory) to be part of a processing node. By using the BROADCAST instructions in the network, four words of code can be loaded into all the processing nodes simultaneously. There is a slight overhead (one word of address and opcode for every four words of instructions), but many thousands of processors could be loaded at the same time. Once enough BROADCASTS have been placed into node memory, the reset signal could be released, and the CPU can start executing the boot code.

7.0 VLSI Implementation of the Back-end

The most of the work on this project was done using standard cells from the MSU library for layout, and U.C. Berkeley Oct tools. Oct is a database system which is used at Berkeley for CAD layout and synthesis tools. Standard cells were used extensively, to increase the speed of production by allowing rapid creation of logic units, and allowing quick debugging of circuits. Standard cells in the library were sufficient for most of the design. The tools **Bdsyn, MisII, Bdnet, and Wolfe** were used almost exclusively in this chip. Bdsyn was used to take behavioral descriptions and turn them into logic, which misII attempts to minimize and generate standard cells. Bdnet was used to connect blocks of circuits together. Wolfe performed the standard cell placement and routing. The Oct tools automated synthesis tools were very useful in creating the design.

Because there is no standard cell to perform the functions necessary for the queue array in the design, a queue cell was laid out by hand using **Vem**, which is a visual editor for VLSI design. The only part laid out by hand was the queue cell, which is used extensively in the NIP. The queue cells were tiled together by **Boss**, which is a C++ interface to Oct.

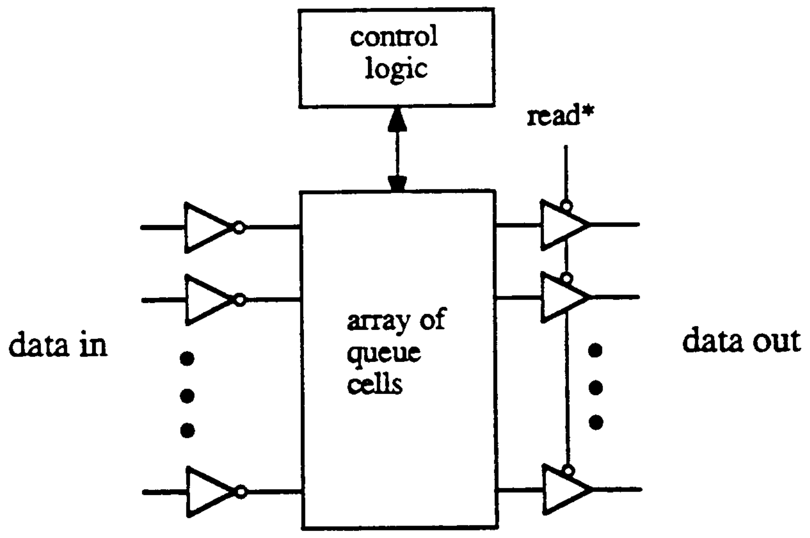

Figure 8: Schematic of a Queue Array with Buffers

7.1 Queue Array

The array of queue cells (figure 8) was laid out using **Boss**, which is a C++ object oriented interface to the Oct database. Since the queue cells were designed to overlap, Boss allowed very close control over the degree of overlap. The Oct tools can be used to automatically route blocks of logic together. However, since the array of cells is a very regular structure, it was better to specify the exact layout. Using Boss saved chip space by allowing routing by cell abutment, as well as saving CPU time wasted routing a structure which was very regular.

7.1.1 Control Logic for the Queue

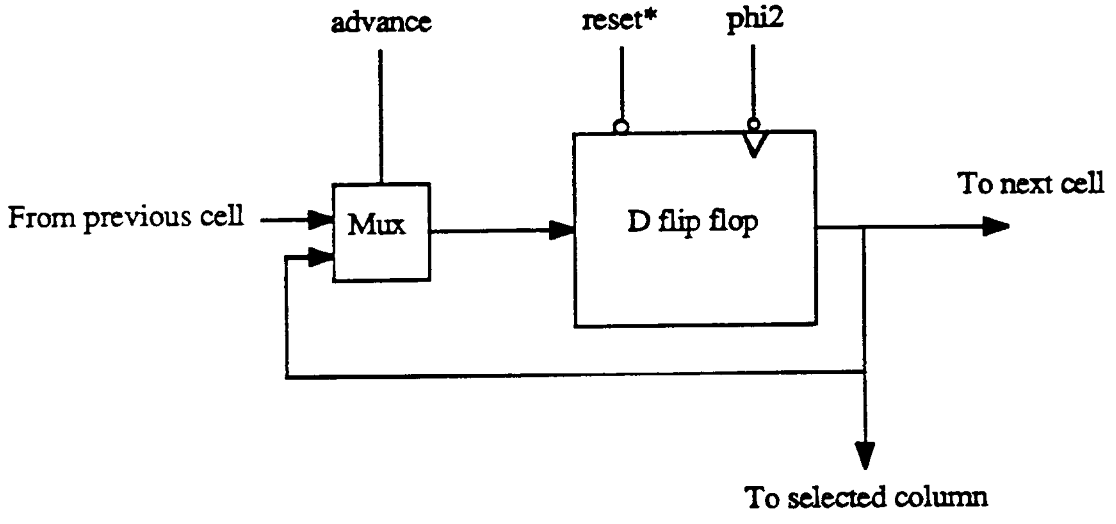

Figure 9: Circular Queue Column Selector Cell

The packet buffers in the back-end are maintained as circular queues. The goal of the control logic is to be able to keep all the queue columns available for use. To implement the circular queue, there are two pointers to each column (read and write) (Figure 9) and a full bit for the whole queue. If the read and write pointers are pointing to the same queue column, the full bit is used to determine whether the queue is empty or full, so all the elements can be used.

The pointers for read and write are implemented as a D flip flop with an input multiplexor. On reset, all the flip flops are reset to zero, except for the first column, which is set to one. If the *advance* line is low, then the pointer does not move. If the advance line is high, then the pointer moves to the next column.

There are a few differences about how the pointers are used. Since write is a destructive operation, the pointer can only be used when a write is actually requested. It would be bad for a column to be overwritten, so the full bit is gated with the write signal to prevent that from happening. After a write occurs (during phi1), the advance signal will be high, and the pointer will advance to the next column on phi2.

The read pointers work in a similar way. After a read occurs, the read pointer is advanced by the same mechanism as above. When the end of the queue is reached, the pointer propagates back around to the beginning of the queue automatically. However, since read is a non-destructive operation, a read can be performed every phi1. If the data consumer wants the data, it will be available, but it does no harm if the data is available every cycle without being used.

To detect if the queue pointers are pointing to the same column, all the corresponding read and write pointers are ANDed together, and these results are ORed together to form an *equal* bit. If the equal bit is one, and the last operation was a read, then the queue must be empty. If the last operation on the queue was a write, then the queue must be full, since the write pointer has advanced to be equal to the read pointer. So the data producers and data consumers can detect if the queue can be written to or read from by examining the full and equal bits. The producers and consumers respect the full and equal bits of the queue, and make sure not to over-write data in the queue, or read non-existent data from the queue..

7.2 The Queue Cell

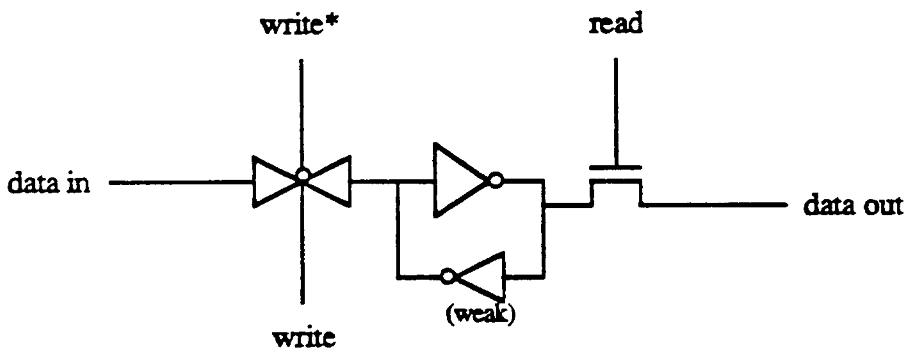

Figure 10: Schematic of a Queue Cell (7 transistors)

Because there is no memory cell in the MSU library that performs the exact functions that the queue cell requires, the queue cell was laid out by hand (Figure 10). The actual design is a static dual-ported cell that requires only seven transistors, and fits into a fairly small area (61 x 66 lambdas). It has access to two buses that can be used simultaneously, but a particular cell will not be read and written at the same time. The input to the cell is a transmission gate that is opened only during a write. The input signal must force the cross coupled inverters to have the correct value by the end of the write cycle. Once the inverters get the correct value, it will remain set with that value until it is changed again, i.e. it requires no refreshing.

During phi2, the output bus is pre-charged in anticipation of the cell being read. If the value of the cell to be read is zero, then if it is read, it will discharge the bus to zero volts during phi1. If the value of the cell is Vdd (logic one), then the bus will maintain Vdd, even though the NMOS pass gate is a poor conductor of Vdd. The inverter that has to force the cell value onto the bus is fairly large, so that it can overpower the value on the bus without resulting in the cell value being erroneously changed.

Since the output of the queue cell is the inverse of what the value written into the cell, either the input to the cell or the output from the cell needs to be inverted. Since very powerful inverter standard cells are available, we chose the input to be inverted (Figure 7). It is important that the input signal be exceptionally strong, because this is the one part of the system where it is necessary for a write to take place on phi1 of the clock. It is safe to drive the inputs to the queue all the time, since the data will be ignored by the cells unless a write is taking place.

The queue cell was tested using SPICE, and was verified to work correctly.

7.3 Testing

The entire back-end of the Fluent chip was verified to be correct by simulation with Musa. Test vectors were created by hand to test the simpler parts of the system, and the whole system was tested by vectors created by a computer program.

7.3.1 The Controller

As the controller for the back-end was the hardest to implement, so it was the hardest to test. Testing pointed out errors in the bdsyn code that could not be perceived run simply analyzing the code. Some of the errors in the code were due to infamiliarity with certain latches in the MSU standard cell collection, which created race conditions in the state machine when the latches were used improperly. Once the simple errors were eliminated, the process of handling complex back-end operations had to be corrected. The read, write, and multi-prefix operations were easily verified. The Notify operations, which take up most of the controller code, were the most difficult to debug. Most of the code was modified to make sure that any sequence of CPU and NIP operations on the Notify queue would be correct, no matter how byzantine the interleaving of interactions.

7.3.2 The Queue

The queue cell for buffering data to and from the network is made of an array of queue cells. The cells were verified using SPICE 3. The difficult part about testing the cells in Musa was the need to attach special properties to the transistors in the cell. Since Musa does not understand transmission gates, pass gates, or cross-coupled inverters, special direction properties and strength attributes had to be attached to some of the transistors. Once all the properties were attached correctly, the queue cell was simulated by Musa in the manner expected.

7.3.3 Infsm and Outfsm

Both of these state machines are simple, and it only took a few days to create them and verify the correctness. During the simulation of Infsm, it was noticed that our communication protocol was incorrect. We had decided that both address information packets had to be accepted or rejected together. However, Infsm would not know the difference between an EOS or the first address packet if a parity error occurred, since it would not be correct to make inferences about garbled data. The new protocol still requires address packets to be taken in pairs, but a rejection signal may be sent after the first packet. If the first packet is an address, then the rejection is ignored, and the second packet follows. An EOS can be rejected by the Infsm, and the network will keep re-transmitting the EOS until accepted. In the previous protocol, the EOS would be automatically accepted whether correct or not, and the communications would become confused.

8.0 Layout Information

The use of the automatic layout generation tools had a major impact in the size of the back-end of the NIP. The parts of the system laid out by hand are the parts that would be expected to be a large part of the space, but turn out to be a minor amount of area. The queue arrays, which can be considered to be a register file, are fairly small. Out of a total layout area of approximately 50 million square lambdas, the queue arrays each take up about 2 million square lambdas each, or less than a tenth of the area.

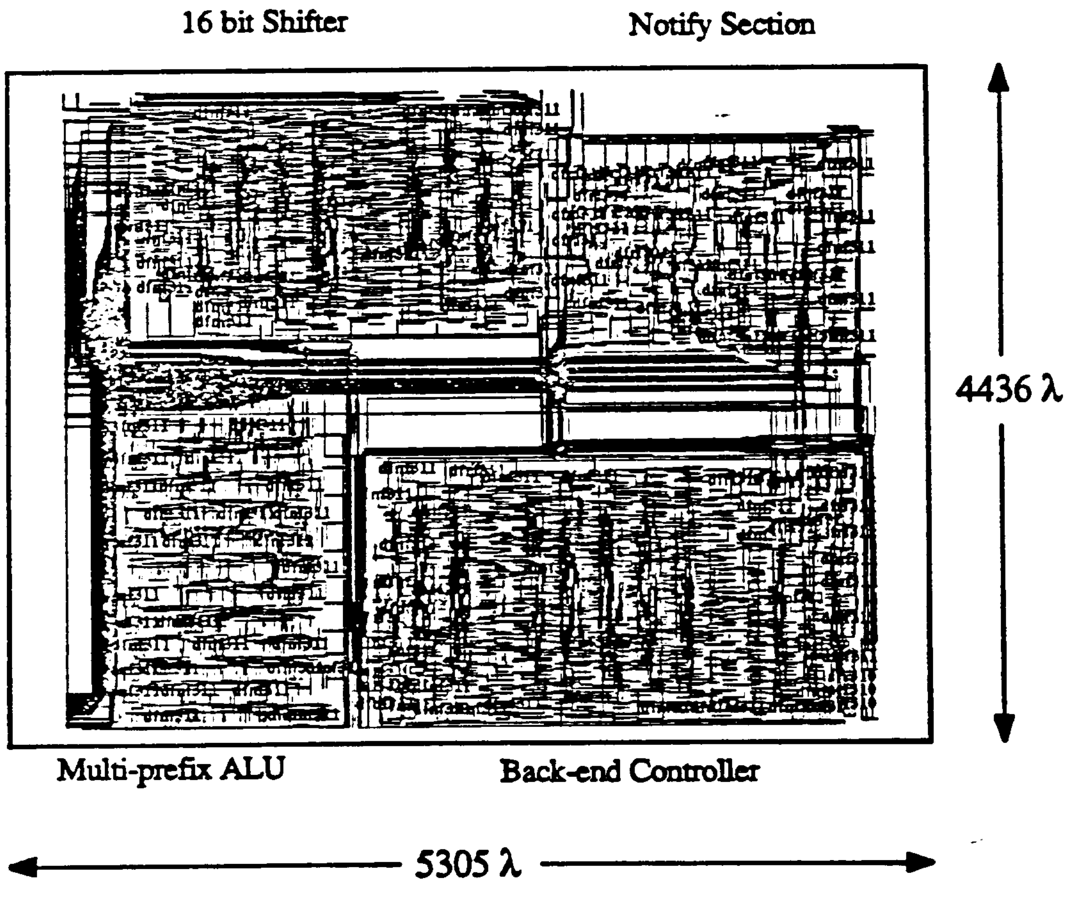

Figure 11: Layout of the Controller Section

By using hand layout, it would have been possible to reduce major areas on the chip. Standard cells were used to implement logical, arithmetic, and shifting units, which take up huge areas. Standard cell registers and signal buffers also took up a good amount of space (Figure 11).

The whole chip together has no set structure to it. Because standard cells were used, a large amount of control was lost over the placement of the small pieces. Wolfe, the Oct interface to the standard cell placer, has very little concept of how buses run through the macro cells, so buses are longer than they should have to be. There is no way to intelligently pass a bus though a macro cell that is not needed by the macro cell, so buses must run around the perimeter of the cells. This results in an increase in size, and loss of good cell placement. Using Puppy to place the macro cells resulted in a different placement depending on how the lower levels of the chip hierarchy changed. But the layout stabilized to the final layout as the sub-pieces of the chip stopped having major modifications made (Figure 12).

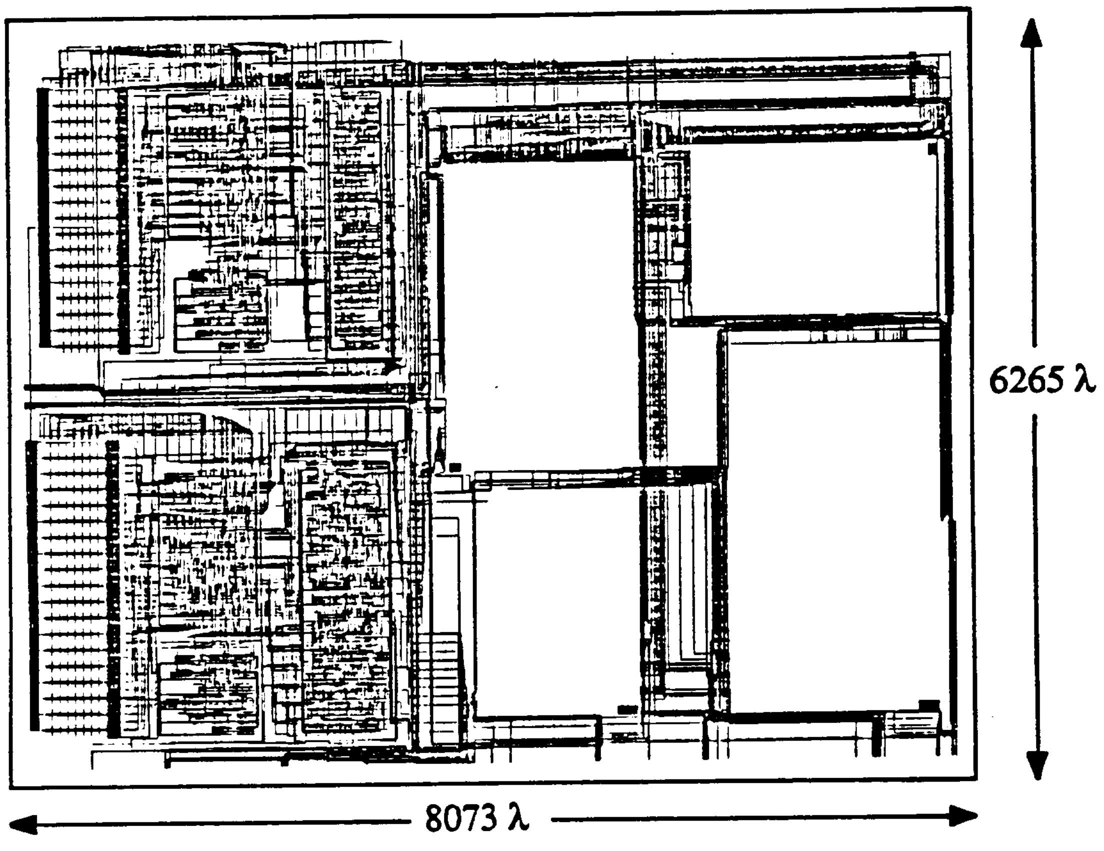

Figure 12: Final Layout of the Back-end

9.0 Performance and Comparison to Related Work

The parallel system we plan to construct is called *Fluent*. To give an estimate of performance, we present a comparison of Fluent with two parallel systems: the J-Machine, and the Transputer. To compare the three systems, we analyze the cost in terms of the number of instructions and system overhead of sending network transactions for three operations: read, write, and remote procedure call. In all cases, we assume that the message to be sent has already been constructed in local memory (or registers) and the necessary pointers to the message reside in the register bank.

Before comparing the systems, we provide an overview of the J-Machine and the Transputer.

9.1 The Jellybean Machine

The J-Machine is a three dimensional mesh of processing nodes **[Dally]**. Each processing node has a 36 bit processor, 4 kilowords (36 bit) of memory, and a router. Messages can be sent from any processor in the system to any other processor through a deterministic path. Messages are built up, and then sent using the *send* primitive. On arrival, the message gets buffer space allocated for it immediately in memory. The processor reads in each message in order, and performs the required operations that the messages request. When there are no messages to process, the node works on background jobs.

The J-Machine was designed to handle very fine grain parallelism, with jobs on the order of 20 instructions, called a task (which is a *code object*). Messages in the network are communications between objects, sent using a SEND primitive. When a message arrives, it is automatically buffered, and eventually a task is created to evaluate it. Messages are not explicitly received, they are put into a message queue and evaluated in turn.

9.2 The INMOS Transputer

The Transputer is a 32-bit microprocessor that has four built-in bi-directional communications links **[Mitchell]**. The processor was built to support the occam model of concurrency. Every statement is a process, which may require input or output. While a process is waiting for a data transfer, a context switch may take place, and the process state placed on an internal stack. Data is evaluated on a register stack. The register stack is small to allow fast context switches (under a microsecond to switch contexts).

The instruction length of the Transputer is one byte, which consists of a four bit opcode with a four bit operand. A special register is used with special instructions to build up longer eight bit opcodes, and which can be executed using an OPR (operate) instruction. All of the eight bit instructions operate on the register stack, and do not have operands directly specified within them.

9.3 Comparison of Write Operation

To perform a write operation in the Fluent system requires only one write transaction (store) on the node bus. The NIP examines the address and data bus, and uses the information to possibly perform other reads to memory. At the same time, the processor can be working on an independent task, and let dedicated hardware pack the messages to be sent into the network. On the receiving end, the node CPU needs to take no action for the write. The NIP at the destination takes care of all the required memory transations.

On the Transputer, a write to a neighbor's memory would be cheap in terms of instructions for the sending processor, but quite costly for the receiving processor. To transmit, a single *out* instruction is executed. This instruction takes two cycles for each word transmitted, plus 19 cycles for context switching overhead. The receiving processor needs to have a server process waiting. The server would execute an *in* instruction, and context switch while it waited for a message. The number of cycles to receive the message would two cycles for each word of the message, and 19 cycles for the context switch. Once the message was received, the server would have to execute several instructions to interpret the message and determine where to place the data in memory.

The J-Machine needs to send a message to perform any kind of operation. To send a message, a SEND2 instruction would be executed for each pair of words to be sent across the network, and a SENDE instruction terminates the message, and releases it to the network. So the instructional overhead would be roughly one instruction for each pair of words to be sent. On the receiving end, there are no explicit instruction to receive a message. Special hardware queues the message, and eventually code to perform the write operation would put the data into memory.

9.4 Comparison of Read Operations

The read operation for each of the systems takes about roughly twice the number of cycles of processor overhead, except for the Fluent system. On Fluent, once the store instruction is executed, that is the end of processor involvement. From that point, the local NIP and the foreign NIP perform all the network and memory operations. So the overhead for a read is the same as the overhead for a write operation

On the Transputer, the overhead would be close to double. The local node first sends a message (using *out*), which requires two cycles for each word of the message plus 19 for context switching. The remote node would execute an *in*, and have the same overhead as the sending node to receive the message. The remote node server process would execute some instructions to read memory, and would have to construct a message buffer and start sending the data back. The local node would have to have a process waiting for the data, and it would have the same overhead as the receiver process on the remote node.

The J-Machine would also require the remote node to receive the requested data in a message, and the data from the message would have to be extracted and written into local memory. The remote node overhead would not increase much; it would just have to execute SEND2 instructions for each pair of words to send back across the network, and a SENDE instruction to end the message.

9.5 Comparison of Remote Procedure Calls

One feature the three systems share is the ability to pass messages between nodes, with the message delivered to a message buffer in the node rather than a specific memory address.

On the Transputer, sending a RPC would take the same overhead as a write operation. The sender would have to execute an *in* instruction, and the message would be passed across to the remote processor. On the receiving end, the processor would execute an *out*, and there will be overhead to interpret the message.

The J-Machine would also have the same overhead for sending an RPC as a write operation. Several SEND2 and a SENDE instruction would be executed to send the message on its way, and the sending processor is finished. On the receiving end, the message would automatically be put into a buffer, which the processor would interpret when it needed a task to perform.

9.5.1 Fluent

The Notify message will fulfill the role of an RPC in our system. The overhead for a Notify is exactly the same as a Write, with just one store instruction required by the processor. Once the Notify message traverses the network, it is put into a Notify slot in the Notify queue.

On the receiving end, the processor must have a server which checks for messages in the Notify queue, and executes the routine indicated by the message.

Operation	Where	Fluent	J-Machine	Transputer
Read	Local	1 Cycle	4 + *	38 + *
Read	Remote	0 Cycles	*	38 + *
Write	Local	1 Cycle	4 Cycles	19 Cycles
Write	Remote	0 Cycles	*	19 + *
RPC	Local	1 Cycle	4 Cycles	19 Cycles
RPC	Remote	*	*	19 + *

* = cycles to execute server process

Table 3: Processor Overhead for 4 Word Network Operations

9.6 Summary of Comparison

The Fluent system compares well with the Transputer and the J-Machine (table 3). With a modest amount of effort by the CPU, the Fluent system can quickly perform the same types of message transfers as the J-Machine and the Transputer, but it is much more flexible. Fluent can send a message (RPC) as easily as the J-Machine (and with fewer cycles than the Transputer), but it is not limited to strict message sending. It can also perform shared memory operations (read and write), which require one cycle of CPU usage, significantly less than either the Transputer and the J-Machine.

10.0 Conclusion

The Fluent multi-processor system is a powerful system for computation. A key part of the system is a communications co-processor, the NIP, which handles many of the transactions needed by the network. I discussed the protocols necessary for communications required between the back-end and the CPU, as well as between the back-end and the network, and believe that the back-end of the NIP embodies the implementation of the best of the protocols.

The major cost of the NIP is the necessity of having another chip for each node in the system. However, this is will actually contain some circuits, such as an MBUS arbiter, which would have required an extra chip anyhow. The layout area required for the back-end of the NIP is 50 million square lambda. When the whole chip is finished, it is estimated to require around 100 million square lambda, or to be 1 centimeter on a side using a 2 micron process.

The NIP will run at 20 megahertz internally, with a section of the MBI that runs at 40 MHz to interface with the MBUS. To interface with the snooping caches used on the MBUS, the NIP will support cache line operations. By supporting the cache line operations, the processor will be able to look in its cache for results of a network operation, instead of having to look into non-cacheable node memory, which would slow down system performance.

The Fluent system has much lower overhead to perform different kinds of communications between processors. The dedicated hardware of the NIP makes this possible, and it takes a significant load of work from the processor. As well as supporting shared memory and message passing operations, the NIP also assists in system synchronization operations, and performs multi-prefix operations.

Bibliography

[Boothe] Robert Boothe. *Tolerating Memory Latency through Multithreading.* PhD. thesis, University of California, Berkeley. In preparation.

[Cross] Davis Cross. *VLSI implementation of the Fluent routing chip.* Master's thesis, University of California, Berkeley. In preparation.

[Dally] William J. Dally. *The J-Machine: System Support for Actors.*

[Gottlieb] A. Gottlieb et al. *The NYU Ultracomputer - designing a MIMD shared memory parallel computer.* IEEE Transactions on Computers, C-32:175-189, February 1983.

[Hunt] James Hunt. *VLSI implementation of the Fluent Front-end.* Master's thesis, University of California, Berkeley. In preparation.

[Mitchell] D. Mitchell et all. *Inside the Transputer.* Blackwell Scientific Publications, London, England, 1990.

[Ranade] Abhiram G. Ranade. *Fluent Parallel Computation.* PhD thesis, Yale University, 1988. Department of Computer Science TR-663.

[Sun] *Mbus specification.* Sun internal document.

Appendix A: The Fluent System

The Fluent Multiprocessor System (FMS) is an extensible network of processing nodes. It is a scalable system, that can scale to 1000 processors or more without having to modify hardware. To implement the system, we expect to use commercially available processors, along with custom integrated circuits to handle communications.

The nodes are connected by a butterfly network that performs combining and multi-prefix operations [**Ranade**] (Figure 13). The network has separate forward and reverse paths, which are both butterfly networks. A message between a pair of processing nodes (PNs) traces a unique path, and if the message requires a response, the message will follow the path backwards through the reverse network. Special status bits are kept in the network nodes (NNs) to help undo the multi-prefix and combining operations when the replies return [**Cross**].

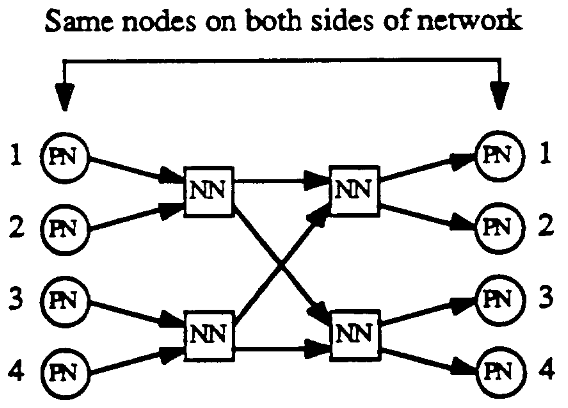

Figure 13: The Fluent System Butterfly Network

A.1 Fluent Processing Nodes

A processing node consists of a Network Interface Processor (NIP), memory, and a CPU connected to the system bus through a memory management unit (MMU) (Figure 1). The CPU is a SPARC chip, which was picked because it has register windows that can be used to store different process contexts (threads). The processor can issue requests to the network by accessing high addresses, which are mapped to special locations that the front-end interprets [HUNT]. These messages get queued in the front-end, and sent into the network. Part of the data sent during the access to the NIP includes information about where to put the result or the location of more data to send in the network, depending on the message size and type. The processor also reads tasks from a special queue in system memory called the Notify queue. The CPU runs several tasks (threads) concurrently, and switches threads upon needing a response that has not been received back from the network. When a thread finishes executing, the CPU can get another task from the Notify queue.

The back-end of the NIP has the ability to perform direct memory access (DMA), and most network operations can be performed independently from CPU execution. The only operations that get passed on to the CPU are the Notify operations, so the CPU can work on its tasks with less interrupts then would be possible otherwise. The back-end can perform a small number of operations, like accessing memory, and performing additions and comparisons. All the operations that the back-end must respond to can be decomposed into those kind of operations.

A.2 Messages in the network

Messages propagate through the forward network in wavefronts, which serve to keep messages in the network synchronized in terms of a distributed global clock. This allows for a close simulation of a CRCW (concurrent-read, concurrent-write) PRAM (parallel random-access machine) model. The ideal PRAM model specifies that all memory operations, whether to local or global memory, will complete in one cycle. To closely simulate the PRAM model, memory accesses to shared memory are constrained within a *wave*.

All the messages within a wave are considered to have been issued at the same time, even if in reality they were put into the network at different times. Through sorting of tags associated with each message (actually the address bits that have been deterministically scrambled), the messages within one wave can be combined if appropriate. Messages leave the front-end of the NIP in sorted order, and through merge sorts, messages going to the same address on the same node can be detected and combined. In a previous system that provided combining, NYU Ultracomputer [Gottlieb], the messages arriving at a switching node had to be compared with all the other messages currently stored in the node. By maintaining sorted messages in each wave, only the messages at the head of internal switching node queues have to be examined to determine if they can be combined.

A.3 End of Stream

To mark the boundaries between waves, a special marker called an end-of-stream (EOS) is sent. EOS is what the synchronization operations referred to earlier in this document are called. The only requirement about the messages the front-end injects during a particular waves is that the messages be in sorted order, but the number of messages injected during a particular wave is not restricted. In some cases, if the front-end has no messages, an EOS must be sent with no preceding messages to keep the network flowing (or fluent).

A.4 Network Nodes

Each NN performs sorting of the messages coming in within a particular wave, and we are guaranteed that messages bound for the same location will be detected. Messages going into an NN enter a queue. Each NN has two inputs and two outputs in the forward direction, and two inputs and two outputs in the reverse direction. When a message reaches the head of the input queue, it is allowed to progress if it is judged to be "less" than the message at the head of the other queue, as judged by the NN comparator. If the messages have the same value and are the type that can be combined (read and multi-prefix), then some status about the messages is kept in the NN, and a combined message is passed on to the next stage. A "large value" message will remain at the front of the queue until a larger value message comes along, or an EOS comes to the front of the other queue. An EOS will block a queue until a matching EOS is seen, and then the EOS's progress through both outputs. Thus, an EOS has the function of forcing the wave in front of it through the network, and makes it necessary for every processor to generate EOS's at about the same rate for the network to function correctly. An unmatched EOS could backup the entire network, but under normal operation, this will not be allowed to happen.

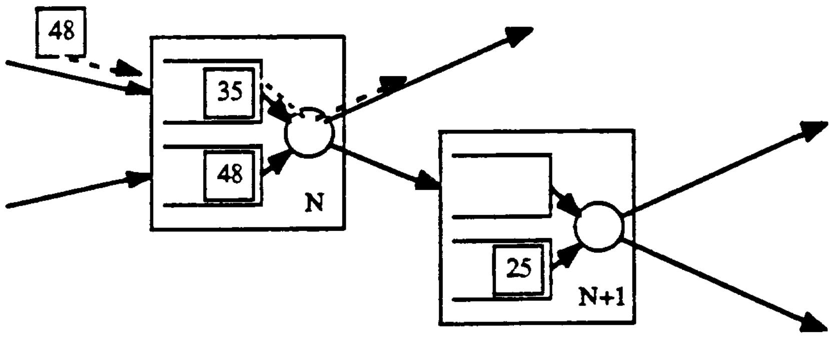

Figure 14: Message sorting in the Fluent network

In figure 14, the message with value *35* will be allowed to proceed to stage *N+1*, because it has a smaller value than the message with value *48*. When the new message with value *48* arrives, then the messages can combine, and progress to stage *N+1*. Eventually all messages get to their intended destination. By the nature of the network, one whole wave must be delivered before the next wave can have its messages forwarded, because any processing node that fails to inject an EOS into the network (a stingy node) will cause the network to stop delivery. Any EOS at the front of a queue will cause the messages behind it in the queue to be blocked until the EOS gets matched and continues to the next stage. All links in the network would be blocked except for a set of links that form a tree leading to the stingy node. The stingy node will get quick delivery of all its messages, and only after it sends an EOS into the network will the blocked messages of the next wave get delivered, thus guaranteeing that the wave *N* messages get delivered to a particular node before wave *N+1* messages.

The reply messages in the reverse direction will trickle back without being constrained to waves, but will be returned in the order the waves were sent in. A processing node will return responses to messages in the order that the messages were received, and status bits remembered from the forward trip of a message will take care of splitting combined operations and performing the correct forwarding for the multi-prefix operations. The return traffic will tend to be fewer messages than went in the forward direction, because some operations do not require a response.

Appendix B: Variable Programming Model

By using different values of EOS's, the Fluent System can support synchronous and asynchronous program models. A synchronous model would require that the processor specifically order an EOS to be sent. The order would be in the form of a write to a special register within the front-end of the NIP, and an EOS would be queued to be sent out to the network. In asynchronous mode, the front-end and the back-end of the NIP cooperate to keep track of the number of EOS's circulating in the network, and to keep the system flowing.

The different values of EOS provide the ability to switch back and forth quickly between synchronous and asynchronous style of programming. The different values provide different priorities for the EOS. In theory it is possible to generate 15 bit values, using the 15 bit field in the EOS opcode, but only four values of EOS are going to be used. The lowest level EOS is level 0, which is called EOS0. The other values are EOS1, EOS2, and EOS3. The higher value EOS's push the lower value EOS's ahead of them when they get matched up at the head of a queue (Figure 15). This feature is used to change between modes quickly, and to clear the network when a program stops executing.

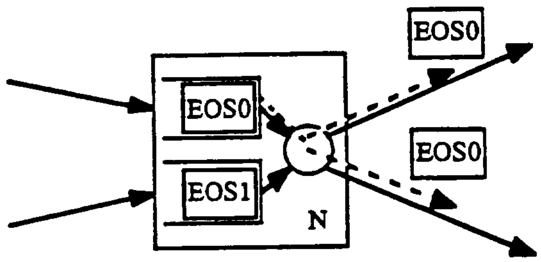

Figure 15. Using an EOS1 to push an EOS0 wave along

In the asynchronous model, the front-end is "given" a certain number of EOS0 tokens as currency, and a target number of tokens to keep in circulation. Every time an EOS0 comes in the back-end, the front-end is informed, and the EOS0 can be recirculated. The target amount is a way of keeping a certain number of EOS0's in the network, so that the wave can propagate even if a particular processor is not sending messages. The front-end also puts an EOS0 between messages if they are not issued in sorted order. The EOS circulator in the front-end can go below the target if a bunch of messages need to be sent in non-sorted order, but has to wait if it runs out of tokens. In this mode, the NIP is totally responsible for keeping the network flowing, and network flow management is invisible to the processor. The processor is not really concerned about delivery order in this mode; it is a very MIMD-like model. The processors operate independently, from each other and have no sense of time in the global sense, since it is not explicitly issuing the EOS's.

In the synchronous model, the front-end is explicitly told each time an EOS is to be issued, and the NIP sends an EOS1 along. The back-end keeps track of how many EOS1's come back, but for reasons that will be discussed later. The processor must do all the work of keeping track of the number of waves that have been sent out, so it "knows" what the global time is for the messages being sent out. We consider this model to be a SIMD-like model because the processors have an awareness of what waves the messages go out in, and rely on the network for messages to be delivered in the correct order. The processors may be executing different instruction streams, but in terms of knowledge of message delivery, it is more SIMD-like.

B.2 Rapidly switching models

The Fluent system can switch back and forth quickly between synchronous and asynchronous modes without having to wait for the network to clear (Figure 16). The circulation system keeps track of how many waves are in the network, and will make sure the system keeps flowing. We view the asynchronous mode as being little, possibly incomplete, wavelets between the waves of synchronous EOS1's.

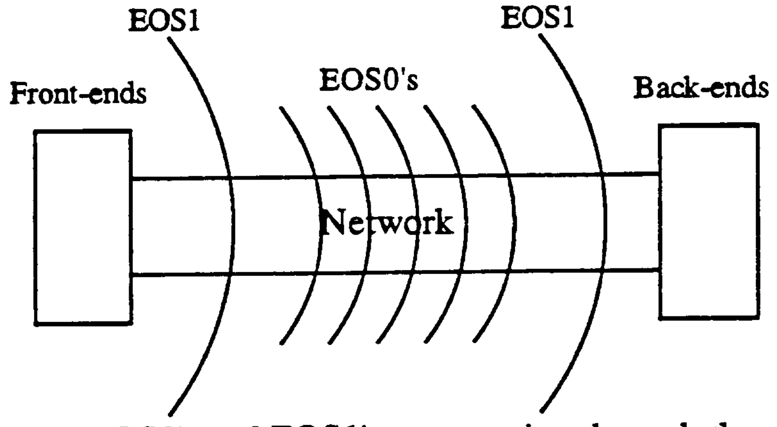

Figure 16: EOS0's and EOS1's propagating through the network

It is possible to keep count of EOS1's, but since an EOS1 matching up with an EOS0 will create two EOS0's and leave the EOS1 in place, keeping absolute count of the EOS0's is very difficult. However, it is reasonable to keep track of the number of EOS0's issued since the last EOS1 was seen.

In asynchronous mode, each EOS0 sent out the front-end will make its way to the back-end as the wave washes across all the back-ends. So a count is made of outstanding EOS0's, so the front-end can keep putting them out, and the back-end can inform the front-end of the number that it gets back. Upon issuing an EOS1, the EOS0 counter goes to zero, and doesn't decrement the EOS0 outstanding counter until the EOS1 comes to the back-end. At this time, the system knows that all the EOS0's have been pushed out of the system, so any further EOS0's must come after the EOS1 wave. So after an EOS1 is sent, the EOS0 counter is reset to zero, and is only incremented until the EOS1 comes to the back-end, and the EOS0 counter can be incremented and decremented after that. In effect, an EOS1 wipes the slate clean, and the circulation counters start over again. In synchronous mode, it is not necessary for the NIP to keep track of the waves, because it knows that any EOS0 waves will get pushed through the network without any problem.

In synchronous mode, all the PNs know the global time, and will agree to stop synchronizing at the same time, so no outstanding EOS1's will be in the network when it is time to switch to asynchronous mode. The network will be clear for EOS0's to propagate, so the EOS0 counter will be correct until the next EOS1 comes through. The end result of the counting strategy is that the circulation counters will be correct when the NIP is responsible for wave propagation, and will be cleared correctly upon the CPU taking responsibility for waves. The counters will remain correct upon the resumption of NIP responsibility for wave motion in the network.

Since it is possible that the system might switch rapidly between the synchronous and asynchronous modes, so there may be several EOS1's in the network simultaneously. The count of EOS0's may not be correct during this time, since any nodes could get away without sending an EOS0 before the time to send an EOS1 comes. So if any EOS0's are injected between quickly succeeding waves of EOS1's, the EOS0 counter may be incorrect, since the EOS1's force propagation of the EOS0 waves. The solution is not to decrement the EOS0 counter until the EOS1 outstanding counter has reached zero. Each time an EOS1 is sent out, the EOS0 counter is reset, and is only modified downward after all outstanding EOS1's have been received.

B.3 Higher valued EOS's

The Fluent system also allows higher level EOS's for clearing the network between programs and for error detection. EOS2 is designated as a special EOS for cleaning the network between programs. The EOS2 will push EOS0's and EOS1's ahead of it, so it is an operating system operation that can be invoked between distinct program, just to make sure the network has been reset to the correct state of having no outstanding EOS's in the network. The EOS0 and EOS1 counters will be reset, and will not decrement until the EOS2 comes to the back-end. Only one EOS2 can be in the network at a time, but EOS0's and EOS1's can be sent out immediately after an EOS2 has been sent. The reason for using the EOS2 is in case a processor has crashed and messed up the global clock.

The EOS3 is an emergency EOS that clears the network of all lower value EOS's. When an EOS3 is generated, no other EOS's are allowed in the network until the EOS3 propagates to the back-end. This can be helpful for diagnostics, since it can help determine which processor crashed in the system. If an EOS3 doesn't get propagated, then a tree of blocked EOS3's will form, and the particular bad processor can be found by seeing which nodes report that the EOS3 has not be returned. This type of EOS will be used only in the situation of a serious system failure, but it may turn out to have uses in legitimate system operation as well.

B.4 Synchronization

In many multi-processor systems, synchronization consists of processors polling a shared memory location until some kind of state change occurs, and the processors continue on from there. This kind of synchronization barrier is rather inefficient, since the processors have to busy-wait and waste cycles that could be put to productive use. However, busy-waiting is the only way to guarantee that parts of the program execution are mutually exclusive from other parts. But, in the Fluent system, the waves in the network guarantee that each set of messages can't be mixed together, and are thus mutually exclusive. Instead of blocking at a barrier, the PN could just send an EOS1, and continue on with its work. The next wave of messages can't be delivered until the previous wave has been taken care of, and the barrier becomes nothing more that a synchronization point.

Appendix C: The Front-end of the NIP

The front-end of the NIP is responsible for monitoring CPU commands to send a message, and processing the request. The message is formed, and sent over the network. When the result of the network comes back (if any), it is responsible for placed the result into a memory location specified by the CPU.

The front-end watches the bus transactions on the MBUS, and detects attempts of the CPU to write to high node memory. If the upper two bits of an address are anything other than 00, then the address and data on the MBUS lines are really a message to the front-end to send out a message (Figure 17).

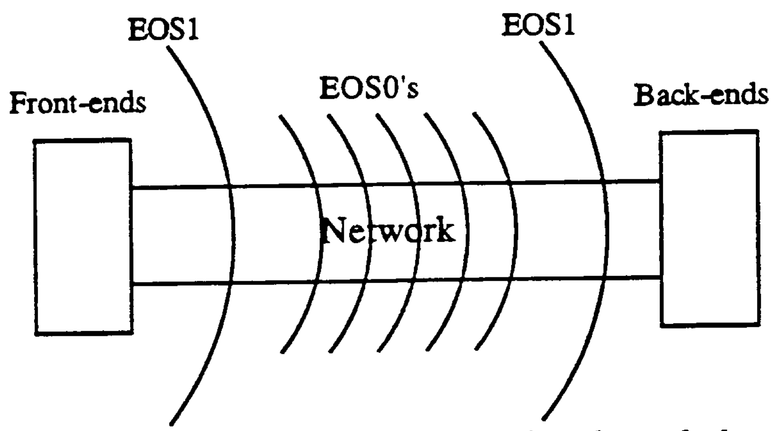

Figure 16: EOS0's and EOS1's propagating through the network

There are three possibilities for the upper two bits:

(01) Multi-word Operation. The Op field contains the operation that the front-end of the NIP should execute. The data word associated with the address on the MBUS contains further information about which node to write to, and a local address to read more data from for the network operation.

(10) Single word read. The Op field contains the node number of the foreign node to access, and the Address field points to the address on that node to read. The data word contains the local address where the result should go.

(11) Single word write. The Op field contains the node number of the foreign node to access, and the Address field points to the address to which the write will take place. The data word is the value to be written across the network.

Our motivation for this format was a desire to minimize the number of memory accesses the CPU must make to send a message across the network. For a single word read or write, only one MBUS transaction is required to provide all the information required by the NIP. We wanted to make the single word accesses fast, because we believe that these operations will be among the more common network instructions issued.

The Multi-word operations require more information from the CPU than can be packed into 64 bits, so the NIP uses the data word associated with the MBUS operation as a local address to read in the extra information. Some of the operations that require this format are multiple word writes, multiple word reads, broadcast, notify, and multi-prefix operations.

The values of the Op field are transformed by the front-end to a new format that the back-end of the NIP recognizes, which were specified in a previous section. The front-end also remembers where to write returning values by putting the addresses specified by the CPU onto an internal queue. Since the messages return from the network in the order they were sent, it is not difficult to match the returning result with the location it is to be written to.

Associated with data returning from the network is a *probe* location. This location is eight bytes after the local address specified by the operation issued by the processor. Since all return locations must be at the beginning of a cache line, and the largest read is eight bytes, the probe location will be in the same cache line as the returned data. The processor checks for the probe location to change value, so it knows when the data has returned from the network. The cache line corresponding to the local destination of the data will be cached by the CPU cache, and it picks up the data when the NIP writes it to local memory. Eventually, the CPU will find the probe location modified, and act on that information. By putting the probe location on the same cache line as the data, and supporting MBUS cache operations, the system will save memory access by finding the commonly checked probe locations in the cache.

For further information on the front-end of the NIP, please refer to the companion report [HUNT].