LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

# The Network Interface Chip

Computation Structures Group Memo 331
June 11, 1991

**Dana S. Henry**
**Christopher F. Joerg**

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

# Contents

# Chapter 1

# Introduction

The Network Interface Chip, NIC, provides network access for the 88100 processor. NIC provides the processor with the capability of sending messages of fixed format into the network, and of reading the fields of an arrived message.

The interface between NIC and the processor is memory mapped. NIC emulates one of (up to four) 88200 cache chips which constitute the processor's first level data cache.

The 88100 communicates with NIC using the 88100's load and store instructions. By mapping NIC into a sufficiently large address space, the processor can trasfer in one clock cycle one 32 bit value along the data bus and several commands along the address bus.

Specifically, NIC occupies a region of the processor's virtual memory determined by the 16 high bits of an address. The processor sends commands to NIC via the 16 low bits of the address bus and the r/w line. It sends data to NIC via the data bus.

At the network end, NIC interfaces to a PaRC chip, a fast network switch [1].

The message format consists of five 32-bit words plus a 5-bit field typically used to indicate the type of message. The upper 8 bits of the first message word must contain the processor ID of the destination processor. The five words of a message can be loaded from 88100's general registers. The 5-bit field is immediate; it is set at compile time.

# Chapter 2

# Programmer's View

In this chapter, we describe the programmer's model of the Network Interface Chip. Using a small set of commands, an 88100 programmer manipulates NIC's visible state in order to

- read an incoming message,
- compose an outgoing message,
- send an outgoing message, or
- read or update NIC's parameters.

In the following Sections, we describe NIC's user visible state, list available commands to NIC, and show how they accomplish the set goals. We conclude with a description of less general, more optimized version of message/overflow handling.

## 2.1 Message Format

NIC supports a single, static length message format. Each message consists of five 32-bit words plus a 5-bit field typically used to indicate the type of message. The upper 8 bits of the first message word must contain the processor ID of the destination processor. The five words of a message can be loaded from 88100's general registers. The 5-bit field is immediate; it is set at compile time.

## 2.2 State

The NI state visible to the user consists of 14 registers, one input queue and one output queue. Figure 2.1 illustrates.

### 2.2.1 Queues

As messages arrive from the network, they are buffered in a fifo input queue inside NIC. The programmer cannot directly access messages inside the input queue. It can only access the message at the head of the queue by popping it into designated registers.

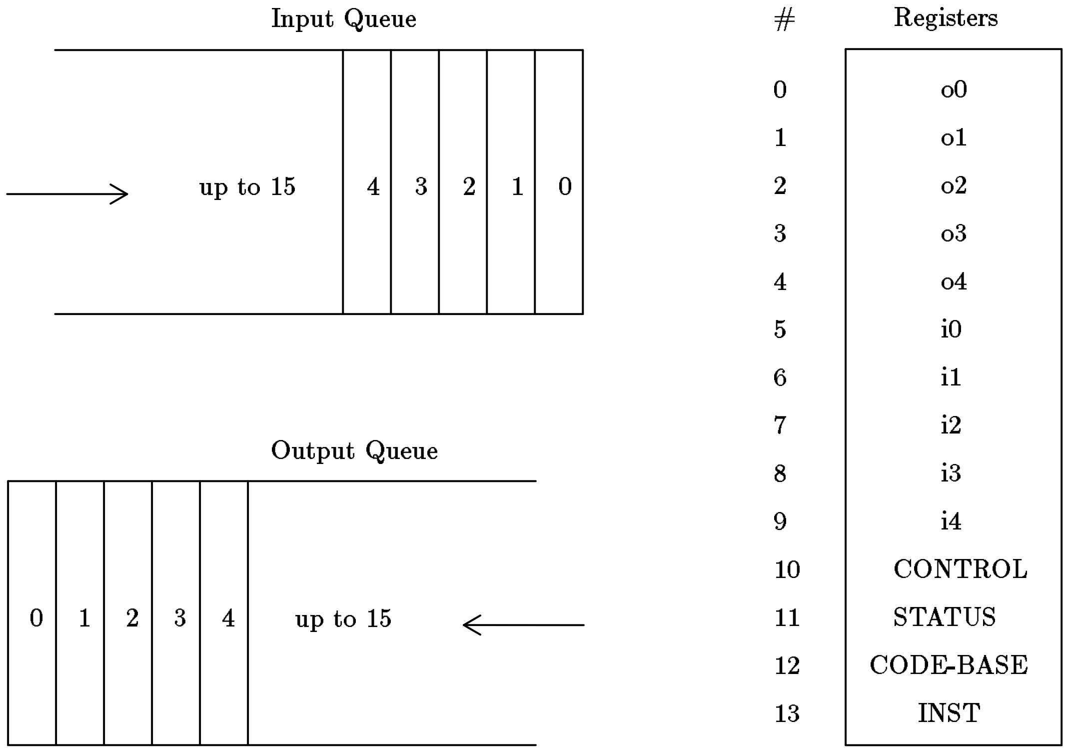


Figure 2.1: The user visible state of NIC.

Posted messages await injection into the network inside a fifo output queue. The programmer composes a new message by writing the message's fields into designated NIC registers. It posts the message by appending a copy of the registers' contents to the output queue. Once posted, messages cannot be accessed inside the output queue.

Each queue can hold up to 16 messages. Section 2.6 describes how overflow is handled for each queue.

### 2.2.2 Registers

The first five registers of NIC contain the five words of an outgoing message that is being composed. Once the 88100 processor has loaded all fields of an outgoing message, it posts the message via the NSEND command. The 5-bit type field is explicitly included with the command (see Section 2.3). Once a message has been posted, the outgoing registers can be reused to compose the next message.

**o0** The first word of the outgoing message that is being composed.

**o1** The second word of the outgoing message that is being composed.

**o2** The third word of the outgoing message that is being composed.

**o3** The fourth word of the outgoing message that is being composed.

**o4** The fifth word of the outgoing message that is being composed.

There are analogous incoming registers. These registers contain the fields of the message that is currently being processed. Once the the 88100 processor has processed the message, it reloads the incoming registers with the next incoming message via the NEXT command.

**i0** The first word of the current incoming message.

**i1** The second word of the current incoming message.

**i2** The third word of the current incoming message.

**i3** The fourth word of the current incoming message.

**i4** The fifth word of the current incoming message.

The remaining four registers provide status and control information, and optimize dispatch to the appropriate message handler for the incoming message:

**CONTROL** The control register has the following fields:

| Encoding of the CONTROL register | | | | | |
|---|---|---|---|---|---|
| 11 | 10 | 9 | 8:5 | 4:1 | 0 |
| IP_ON | OP_ON | ROUTE | oTHRESH | iTHRESH | F/W |

| Encoding of the CONTROL register | | | | |
|---|---|---|---|---|
| 31:25 | 24 | 23:20 | 19:18 | 17:12 |
| reserved | RESET_NIC | OUT_EXCS | RESET_NET_EXCS | NOTE_EXCS |

**F/W** The F/W (Fault/Wait) flag determines the response to a send request when the output queue is full. If the F/W flag is set, the NI replies with memory FAULT on the next cycle. If the F/W flag is not set, the NI keeps replying with memory WAIT. Section 2.6 describes the overflow handling mechanism in full.

**iTHRESH[3:0]** The CONTROL[iTHRESH] field controls the threshold at which the STATUS[iaFULL] is high. The STATUS[iaFULL] flag is high whenever the length of the output queue exceeds the threshold set by CONTROL[iTHRESH]. The flag is low otherwise.

**oTHRESH[3:0]** The CONTROL[oTHRESH] field controls the threshold at which the STATUS[oaFULL] is high. The STATUS[oaFULL] flag is high whenever the length of the output queue exceeds the threshold set by CONTROL[oTHRESH]. The flag is low otherwise.

**ROUTE** The ROUTE flag determines which mapping NIC uses to generate a routing address from the destination processor's id number (the high 8 bits of o0.) If the ROUTE flag is set, NIC generates routing addresses for a 4-processor ring. If the ROUTE flag is not set, the NI chip generates routing addresses for a switching network. Section 2.9 describes both topologies and their corresponding mappings of processor id numbers to routing addresses.

**OP_ON** The OP_ON flag determines if NIC's Output Port is turned on. If high the port is on and can send out messages. If low the port will not start to send any more messages. (If a message is being sent when the port is turned off, the port will transmit the rest of the message normally.)

**IP_ON** The IP_ON flag determines if NIC's Input Port is turned on. If high the port is on and can receive messages. If low the port will ignore its input. (If a message is being received when the port is turned off, the port will receive the rest of the message normally.)

**NOTE_EXCS[5:0]** Each bit in the NOTE_EXCS field determines wheather the INST register ought to point to an EXCEPTION handler when the corresponding exception is detected via the STATUS register. This is described in 2.7

**RESET_NET_EXCS[1:0]** The RESET_NET_EXCS[1:0] bits are 'command bits' used to reset STATUS[EXCS][1:0] bits, the network exception bits. Whenever a 1 is 'written' into one of these bits, the corresponding STATUS[EXCS] bit is reset to zero. The actual value of the RESET_NET_EXCS field remains zero.

**OUT_EXCS[3:0]** The OUT_EXCS field determines the values of NIC's four outgoing exceptions. See Section 2.7.

**RESET_NIC** The RESET_NIC bit is a 'command bit'. Whenever a 1 is written into this bit the reset_nic command is performed. The actual value of the RESET_NIC bit remains zero. This command is used to reset NIC to a known state. NIC is initialized and all queues are emptied. All bits in the STATUS register are reset to 0. Once this command is given the reset begins and may take up to 6 (??) cycles to complete. No other access should be made to NIC during this time.

**STATUS** The status register, read-only, has the following fields:

| Encoding of the STATUS register | | | | | |
|---|---|---|---|---|---|
| 31:18 | 24:20 | 19:15 | 14 | 13 | 12:8 |
| reserved | oLENGTH | iLENGTH | oaFULL | iaFULL | iTYPE |

| Encoding of the STATUS register | | |
|---|---|---|
| 7 | 6:1 | 0 |
| reserved | EXCS | VALID |

**VALID** This flag specifies whether the incoming registers contain a valid message.

**EXCS[5:0]** The EXCS field reflects the state of NIC's exceptions. EXCS[1:0] bits, the network exception bits, get set whenever an exception is raised and remain set until explicitly reset by CONTROL[RESET_NET_EXCS]. EXCS[5:2] bits mirror the values along NIC's four incoming exception pins. See Section 2.7 for a discussion of exceptions.

**iTYPE[4:0]** The 5-bit type field of the current incoming message.

**iaFULL** The iaFULL ("input queue almost full") flag is 1 whenever the length of the input queue exceeds CONTROL[iTHRESH] and 0 at all other times.

**oaFULL** The oaFULL ("output queue almost full") flag is 1 whenever the length of the output queue exceeds CONTROL[oTHRESH] and 0 at all other times.

**iLENGTH[4:0]** iLENGTH specifies the number of messages in the input queue.

**oLENGTH[4:0]** oLENGTH specifies the number of messages in the output queue.

**CODE-BASE[31:0]** This read-only register can be loaded with the base address of processor's message handler codes. It is used by the INST register to compute the exact message handler to jump to. The low 15 bits of this value are hardwired to 0.

**INST[31:0]** The INST register is a special read only location. It is not a register in hardware terms since its value is computed every cycle. This value can be used to specify a handler which can process the current message.

The exact way in which the content of the INST register is computed is defined in Section 2.8.

## 2.3 Interface Commands

An 88100 programmer can send up to three simultaneous commands to NIC along the address bus:

1. It can request to either load or store one NIC register via the NLOAD/NSTORE command.
2. It can advance to the next incoming message via the NNEXT command.
3. It can send an outgoing message via the NSEND command.

In the next few paragraphs we describe in detail the three command types and give an example of their use.

**NOTE:** Any combination of these commands fits into the low 16 bits of an address. As a result, all combinations simply translate into a single 88100 load(.d) or store(.d) instruction with a different immediate offset. The actual translation of commands into address bits can be found in Chapter 3.

**(NLOAD(.d)** gr nr)

The NLOAD command loads the content of a network register, nr, into one of the processor's general registers, gr. For instance, the command

NLOAD i3 r10

loads the content of the NI read-only register i3 into the processor's general register r10.

A special form of the NLOAD command, NLOAD.d, loads from two successive processor registers, gr and gr+1, into two successive network registers, nr and nr + 1.

**WARNING:** NLOAD.d executes NLOAD two times. Any other command issued together with NLOAD.d will **also** be executed two times.

**(NSTORE(.d)** nr gr)

Similarly, the NSTORE command stores the content of one the processor's general registers, gr, into a network register, nr.

A special form of the NSTORE command, NSTORE.d, stores the content of two successive processor registers, gr and gr+1, into two successive network registers, nr and nr + 1.

**WARNING:** NSTORE.d executes NSTORE two times. Any other command issued together with NSTORE.d will **also** be executed two times.

**(NNEXT )**

The NNEXT command is used to inform NIC that the message in the incoming registers is no longer needed. This will allow NIC to load the next message into these registers.

When the NNEXT command is received NIC immediately resets the STATUS[VALID] bit and attempts to reload the incoming registers with the head message of the incoming message queue. If the incoming message queue has a message when NNEXT occurs, that message will be loaded into the incoming registers, and STATUS[VALID] set high, in time to be read by the next NIC transaction. If the incoming message queue is empty when NNEXT occurs, STAUS[VALID] will stay low. Whenever STATUS[VALID], is low, NIC waits until a new message arrives and then automatically loads that message into the incoming registers. Once the incoming registers have been loaded with the contents of a new message, NIC sets the STATUS[VALID] bit to 1.

Note that upon reset, the STATUS[VALID] bit is reset to 0, so as soon as a message arrives it will be loaded into the incoming registers

**(NSEND** message-type bypass-option cs)

The NSEND command is used to place an outgoing message into the outgoing message queue. Messages in this queue are sequentially transmitted into the network via the PaRC protocol.

The NSEND command informs NIC that the outgoing registers contain a valid message to be sent. NIC appends the outgoing message to the outgoing message queue. It uses the message-type field of the NSEND command for the type bits of the message.

If a write of one of the outgoing registers is being done at the same time a send command is given, the updated value of that location will be used in the message. A send command leaves the values of the outgoing registers unchanged.

The NSEND command supplies two additional arguments, bypass and cs. The bypass argument is optional. When present, it tells NIC to bypass the contents of certain outgoing registers with those of certain incoming registers. This option is useful for replying to an incoming message or for forwarding parts of an incoming message.

The possible values of the bypass argument are:

**Reply** Fields o0 and o1 of the outgoing message are replaced with fields i1 and i2 of the incoming message.

**Forward** Fields o3 and o4 of the outgoing message are replaced with fields i3 and i4 of the incoming message.

If a NNEXT is done during the same transaction as a NSEND with bypass option, the message is composed using the values of the old incoming message, not the newly loaded incoming message.

By default, outgoing messages are not circuit switched[1]. The optional argument, cs-flag, can cause a message to be circuit switched. The optional cs-flag argument has only one possible value:

**cs** The message is sent circuit switched.

[1]For a description of PaRC's circuit switched mode, see [1].

**Command Interactions:** Concurrent NLOAD/NSTORE, NNEXT, and NSEND commands can interact with each other. For example, if an NSTORE of an outgoing register is done at the same time as an NSEND, is the value sent in the message the old or the new value of the outgoing register. NIC will perform these commands such that they appear to have occured sequentially in the following order: NLOAD/NSTORE, NSEND, NNEXT.

## 2.4 Receiving Incoming Message

To receive incoming messages, the 88100 processor must periodically poll NIC for a new message. If a new message is waiting, the processor must find the type of the message and transfer control to the appropriate message handler. Finally, once the processor is done reading the fields of the current message it must signal NIC via the NNEXT command. A sample code for receiving messages follows.

The processor first checks the STATUS[VALID] bit to see if a new message is waiting in the input registers:

```
LOAD STATUS r10            ;r10 ←NI status register
bb0 VALID r10 no-message   ;check if VALID bit is set
```

If a new message is waiting, the processor must find the type of the message in order to transfer control to the appropriate message handler. In general, the type of the incoming message can be specified by any and all fields or subfields of the message. To illustrate, let us adopt the convention by which the fourth word of the incoming message, the i3 register, defines the type of the message. Specifically, we assume that the i3 register contains the message handler offset from some base stored in register r2:

```
NLOAD i3 r14   ;r14 ←i3
add r11 r2 r14 ;r11 ←message handler address
jsr r11        ;jump to message handler
```

While processing a message, the processor can read the fields of the received message by reading the appropriate registers (i0, i1, i2, i3). When done reading these registers, the processor tells NIC to move the next message in the input queue into the interface registers. For instance:

```
NLOAD i0 r11         ;r11 ←i0
NLOAD i1 r12         ;r12 ←i1
NLOAD i2 r13         ;r13 ←i2
NLOAD i4 r15, NEXT   ;r15 ←i4, next
⋮
process message
```

Using NLOAD.d, we can compress the above sequence to:

```
NLOAD.d i0 r11       ;r11, r12 ←i0, i1
NLOAD i2 r13         ;r13 ←i2
NLOAD i3 r14, NEXT   ;r14 ←i3, next
⋮
process message
```

## 2.5 Sending Outgoing Message

To send an outgoing message, the 88100 processor must write the corresponding fields of the message into the output registers and initiate a send via the SEND command.

A sample code for sending a message follows. Here we assume that the five words of the outgoing message can be found in processor registers r10 through r14 and that the type field of the outgoing message is 4:

```
NSTORE o0 r10              ;o0 ←r10
NSTORE o1 r11              ;o1 ←r11
NSTORE o2 r12              ;o2 ←r12
NSTORE o3 r13              ;o3 ←r13
NSTORE o4 r14; SEND 4      ;o4 ←r14, send
```

Using NSTORE.d, we can compress the above sequence to:

```
NSTORE.d o0 r10            ;o0, o1 ←r10, r11
NSTORE.d o2 r12            ;o2, o3 ←r12, r13
NSTORE o4 r14; SEND 4      ;o4 ←r14, send
```

## 2.6 Handling Overflow

Two overflow cases must be addressed in NIC: the input queue overflow and the output queue overflow. The first case arises when a new message arrives from the network while the input queue is full. The second arises when a send command cannot be honored because the output queue is full.

### 2.6.1 Input Queue Overflow

The sender NI, together with PaRC, guarantees not to overflow the input queue of the destination NI. In case of a full input queue, both chips guarantee to stop sending a message to the destination NI chip before its queue overflows[2]. However, this blocks a network output. If it stays blocked for long, it could prevent inputs from sending any more messages. For this reason, it is desirable to avoid filling the input queue by periodically checking the STATUS[iaFULL] flag.

### 2.6.2 Output Queue Overflow

In contrast, the output queue may overflow. The output queue overflows when NI chip receives a NSEND command while the output queue is full. NIC replies to the processor with a memory WAIT state or a FAULT state. The choice of response is determined by the CONTROL[F/W] flag.

[2] See [1] for further details.

#### Fault Response

If the F/W flag is set, NIC ignores the send request. If the NI chip receives any other commands at the same time as the NSEND command, it ignores them.

On the following clock cycle, NIC asserts the memory FAULT state. The FAULT state causes the processor to transfer control to a trap handler. This trap handler can then process incoming messages until the network unclogs and a new send request succeeds.

#### Wait Response

If the F/W flag is not set, NIC keeps asserting the memory WAIT state until there is space available in the output queue at which time the stalled NSEND and any simultaneous commands are executed.

It is important to keep in mind that, while the WAIT state is asserted, the processor is blocked and cannot process incoming messages. This mode is useful if individual NI chips are being used only to send messages, and not to receive messages. In such a case, asserting WAIT instead of FAULT avoids a needless trap.

#### Avoiding Overflow

We can avoid overflow of the output queue alltogether by enforcing the following two software conventions:

1. We limit the number of messages that can be sent by any one message handler to less than (16 - CONTROL[oTHRESH]).
2. At the same time, we promise to check the STATUS[oaFULL] flag before dispatching a new incoming message.

## 2.7 Exceptions

NIC can detect up to four external events from four input pins and signal four events to four output pins. These external events, together with two types of locally detected network errors, comprise all of NIC's exceptions.

A protocol is defined for detecting and asserting exceptions via the STATUS, INST, and CONTROL registers.

### 2.7.1 External Exceptions

The contents of the incoming exceptions pins are directly mapped into the EXCS[5:2] field of the STATUS register. Correspondingly, the contents of the OUT_EXCS[3:0] field of the CONTROL register are directly mapped to the outgoing exceptions pins. If any one of the incoming exceptions is asserted and the corresponding CONTROL[NOTE_EXCS] bit is high, the INST register will point to the exceptions handler.

It is up to the system designer to decide how to make the best use of the four available input and output exception pins. Possible usage includes detecting PaRC's error signal(s) and global synchronization.

### 2.7.2 Network Errors

One special type of exceptions are network errors. It is possible that an error may be detected on NIC's connection to the network. There are two types of errors that could occur: room_error and idle_error. These two conditions map into the two lowest bits of STATUS[EXCS]:

| Encoding of STATUS[EXCS] | | |
|---|---|---|
| 5:2 | 1 | 0 |
| external exceptions | IDLE_ERROR | ROOM_ERROR |

The network protocol should prevent packets from arriving when there is no room for them. If a packet does arrive when there is not room for it a room error is said to have occured. This is a serious condition because it indicates that a message was probably lost. When a room error occurs the STATUS[EXCS][0] bit is set in the status register. This bit stays set until it is explicitly reset by storing 1 in the RESET_NET_EXCS[0] bit of the CONTROL register. Until then, if CONTROL[NOTE_EXCS][0] is high, the INST register points to the exceptions handler (Section 2.8).

A second type of error is an idle error. This condition also indicates that data may have been lost. (See Section 3.4 for more details on this error condition.) When an idle error occurs the STATUS[EXCS][1] bit is set in the status register. This bit stays set until it is explicitly reset by storing 1 in the RESET_NET_EXCS[1] bit of the CONTROL register. Until then, if CONTROL[NOTE_EXCEPTIONS][1] is high, the INST register points to the exceptions handler (Section 2.8).

## 2.8 Optimized Message Handling Using INST

The INST register optimizes message handling for the special case where:

- The type of the message is determined by the 5-bit type field.
- Message types 30 and 31 and reserved for the no-message handler and for the exception handler respectively.
- Messages of type 0 supply the location of their message handler in the second word of the message.
- Messages of type 1 through 29 are handled by a handler specific to that message type.
- The input queue exceeding threshold set by CONTROL[iaFULL] and/or the output queue exceeding threshold set by CONTROL[oaFULL], divert attention to different handlers. These handlers are specific to each message type, allowing each message type the option to ignore these conditions.

- Exception conditions (see Section 2.7) may divert attention to one of the two exception handlers.

The INST register reduces the cost of dispatching to the appropriate message handler by precomputing the message handler's address in NIC.

The INST register is normally loaded with the message handler corresponding to the incoming message type. If there are no errors, neither queue is "almost" full, and there is a valid message of type 0, the i1 field of the message is the value of INST. Otherwise the INST value is built up from pieces of the STATUS and CODE-BASE registers. This "constructed" INST is formed by concatenating the bits of CODE-BASE, CONTROL[iafull], CONTROL[oafull], and CONTROL[type]. There are, however, two reserved types: 30 and 31. If there is no valid message and, no notable exceptions, then 30 will be returned in the type field. If there is a notable exception, 31 will be returned in the type field.

Here is the exact algorithm for computing the value of INST:

```
if    (STATUS[VALID] && STATUS[iTYPE] == 0 &&
      !STATUS[iaFULL] && !STATUS[oaFULL] &&
      ((STATUS[EXCEPTIONS] && CONTROL[NOTE_EXCEPTIONS]) == 0))
      /* message type = 0 and no unusual conditions reported */
      INST[31:0]= i1;                    /* return IP of code to be run */
else {
      /* Build the INST value from other registers */
      INST[31:15] = CODE-BASE[31:15];
      INST[14] = STATUS[oaFULL]
      INST[13] = STATUS[iaFULL]
      INST[12:8] = STATUS[iTYPE];
      if    (!STATUS[VALID])
            INST[12:8] = 30;
      for (i = 0; i =< 5; i++)
      if    (STATUS[EXCEPTIONS][i] && CONTROL[NOTE_EXCEPTIONS][i])
            INST[12:8] = 31;
      INST[7:0] = 0};
```

Using the INST register, we can check whether a new message has arrived, check whether either queue is almost full, and dispatch to the appropriate handler in two instructions:

```
NLOAD INST r14    ;r14 ←INST
jsr r14           ;jump to message handler
```

## 2.9 Message Routing

NIC supports two kinds of network topologies: A ring of up to four nodes where each node has two NIC chips and Omega (butterfly) type networks of up to 256 NIC chips. Bits in the header of each message control how that message is routed. NIC assumes that the upper 8 bits of the o0 field of the outgoing message is the node number that the message should go to. NIC uses this node number to generate the appropriate header routing information. If CONTROL[ROUTE] is 1 the routing information is set to provide ring routing, otherwise to provide routing in an Omega network.

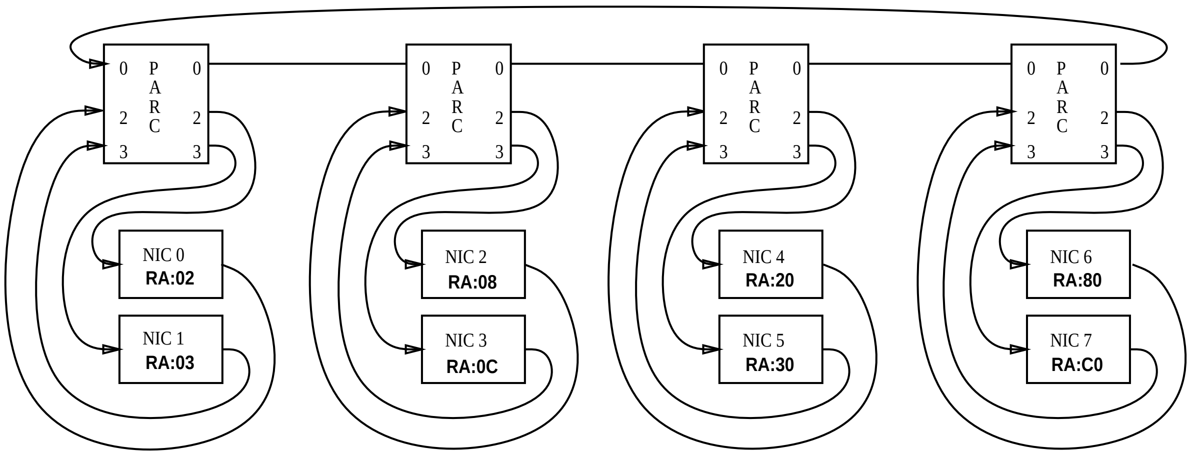


Figure 2.2: A Ring Network

The ring topology is shown in Figure 2.2. This topology has four nodes and each node has two NIC chips. Bits 2 and 1 of the node number are used to specify which node a message should go to. Bits 0 specifies one of the two NIC chips on that node. From this information NIC conputes the routing address of that NIC chip, and places that address in the header. The figure shows the node number used to specify each NIC chip, and what routing address is generated for that NIC.

For butterfly routing, the job is easier. NIC simply outputs the node number as the routing information. The network can then route the message to the appropriate node.

# Chapter 3

# MicroArchitecture

## 3.1 Overview

The Network Interface Chip (NIC) is made up of 5 sections as shown in Figure 3.1. The P bus Interface section is used to connect the chip to a processor. Through this interface the processor is able to send messages to and receive messages from the network. The outgoing message queue stores messages that are waiting to be sent, and the output port takes each message out of this queue and sends it into the network. Similarly the PaRC input port receives messages from the network and places them into the incoming message queue. From here they can be given to the procesor over the Pbus Interface.

## 3.2 P Bus Interface

The NI chip provides a P bus compatible interface which allows a processor to interact with NIC. Since not all the flexibility of other P bus slaves (such as the 88200) is needed, a simplified version of the P bus protocol is implemented. This interface uses the following signals:

| **Pbus Interface** | | |
|---|---|---|
| Signal | Dir. | Description |
| da[13:0] | I | bits 15:2 of the address bus |
| d[31:0] | I/0 | bidirectional DATA bus |
| r_w | I | Read/nWrite line |
| ncs | I | active low Chip Select |
| dr[1:0] | I/O | bidirectional data reply signals |
| dbe | I | is high if one of dbe3-dbe0 is high (ie. indicates a Pbus transaction is occurring) |
| clk | | Pbus clock |

The signal names are chosen to match the corresponding Pbus signals. A series of transactions will occur on the Pbus, only some of them are transactions for the NI chip. During each transaction

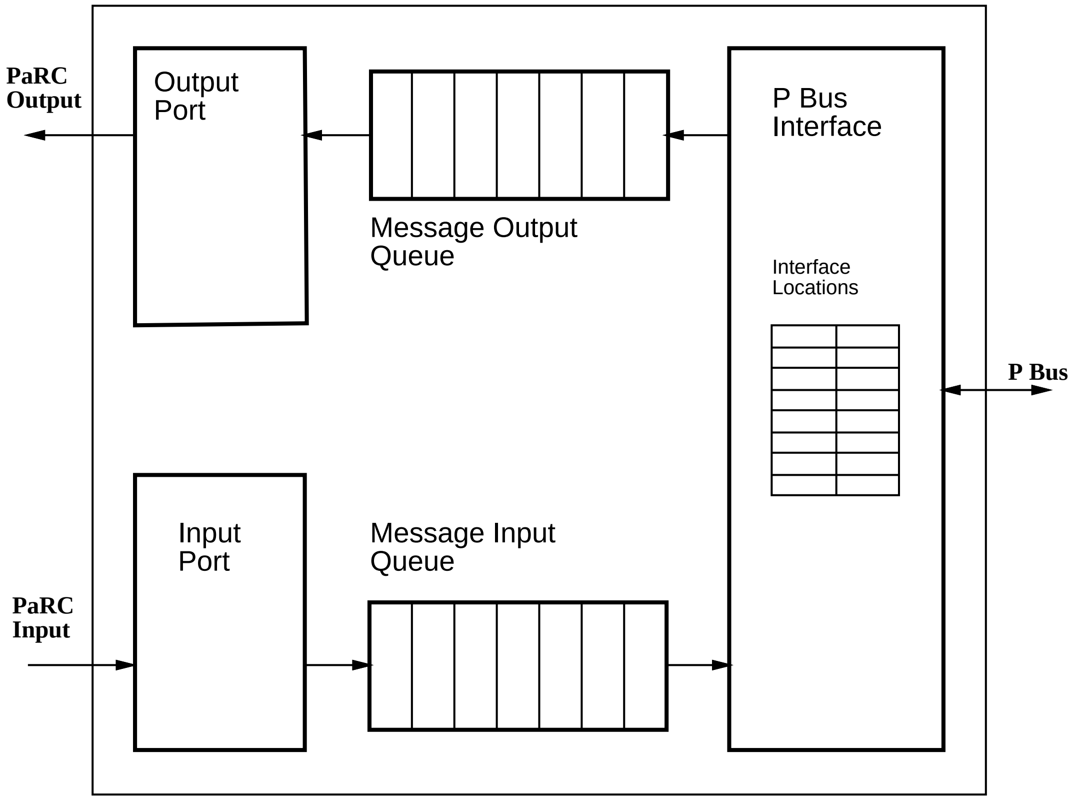


Figure 3.1: Overview of the NI chip

external logic looks at the upper bits of the address and activates either the NI chip (by asserting ncs) or one of the other P bus slaves (such as an 88200 cache chip.) Since external logic makes this decision, these bits of the address are not needed by the NI chip itself. Since no xmem operations will be performed by this chip, the Pbus dlock signal is also not needed. The $S/\overline{U}$ is also not needed since the chip does not differentiate between the supervisor and normal users. Also, Since no byte operations are done, dbe3-dbe0 are not needed. However, we still need to be able to detect NULL transactions. (A NULL transaction is a Pbus cycle in which no operation is performed.) For this reason dbe (= dbe3 + dbe2 + dbe1+ dbe0) is provided. This signal will be low only during NULL transactions. (We need to identify NULL transactions because if a P bus slave is selected on the cycle following a non-NULL transaction, it must check the reply lines (dr) to see if the previous transaction has completed successfully. If not then the current transaction should be ignored.)

The NI chip monitors each Pbus transaction to determine if it is a transaction for the NI chip. If so, the NI responds according to the rules of the Pbus protocol. When responding to a transaction, the NI chip treats da[13:0] as specifying a command to perform. Each transaction is either a read transaction or a write transaction. When responding to a read transactions the NI chip will place data onto the data bus. When responding to a write transaction the NI chip may read data from the data bus. Also, when responding to a transaction the NI chip must drive the reply lines (dr[1:0]). This reply will indicate whether or not the NI chip was able to successfully complete the transaction.

### 3.2.1 P Bus Commands

The Pbus is used to give commands to the NI chip, and to give and receive data from the NI chip. When a transaction occurs during which the NI chip is activated, the da[13:0] bus is used to tell the chip what actions to perform. The da bus is broken into a series of fields.

| **Command Fields in da** | | | | |
|---|---|---|---|---|
| 12 | 11:10 | 9 | 8:4 | 3:0 |
| CSP | SEND | NNEXT | OTYPE | LOC |

where,

LOC Specifies which register location is being addressed.

OTYPE Specifies the type of the message to be constructed.

NNEXT Specifies if the chip should advance to the next incoming message.

SEND Specifies if a message should be sent and how to construct it.

CSP Specifies if the message should be sent as a circuit switched packet.

NIC contains 16 interface locations (2 of which are unused) Each Pbus transaction is either a read or a write to one of these locations. The d[31:0] bus is used for sending or receiving the data. The LOC field specifies which NI register is being read or written.

| Encoding of LOC | | | | | |
|---|---|---|---|---|---|
| LOC | register | R/W | LOC | register | R/W |
| 0 | o0 | r/w | 8 | i3 | r/w |
| 1 | o1 | r/w | 9 | i4 | r/w |
| 2 | o2 | r/w | A | CONTROL | r/w |
| 3 | o3 | r/w | B | STATUS | r/w |
| 4 | o4 | r/w | C | CODE-BASE | r/w |
| 5 | i0 | r/w | D | INST | r |
| 6 | i1 | r/w | E | reserved | r |
| 7 | i2 | r/w | F | reserved | |

A write done to a read only location will be ignored.

#### NI locations

The use of each location ios explained in the preceding chapter.

### NI Commands

The SEND field of a command indicates if a message should be sent and, if so, how the message should be constructed. The typical way to send a message is to use all of the values stored in the outgoing registers. Variations of this allow two of the incoming registers to be used in place of two outgoing registers. If a send occurs as a value is being written into an outgoing register, the new value of the outgoing register will be used in constructing the message. If a send occurs as a value is being written into an incoming register, that register should not be used in constructing the message.

Each message sent must have a type field. The value of the OTYPE field of the command is directly used as the type of the message constructed.

| Encoding of SEND | |
|---|---|
| send | action |
| 00 | NO-MSG |
| 01 | SEND |
| 10 | REPLY |
| 11 | FORWARD |

where,

NO-MSG Do not perform a send.

SEND Send a message (use outgoing values o0 through o4).

REPLY a message using i1, i2 in place of o0,o1.

FORWARD Send a message using i3,i4 in place of o3,o4.

If the CSP command bit is set then the circuit switch bit will be set in the message's header.

The NEXT field is used to instruct the NI chip to load the next message into the incoming message registers.

When the NEXT field is asserted, the incoming message valid bit (VALID) is set to 0. Whenever this bit is 0 the NI chip will remove the head message from the incoming message queue and place it into the interface registers. It then also sets the valid bit back to 1. If there is a waiting message when NEXT is asserted, then this transfer will occur soon enough so that values from the new message can be accessed on the transaction immediately following the one which asserted NEXT. If the valid bit is low and the incoming queue is empty then there is no new message to be loaded, so the VALID bit will be left low.

If an access of an incoming message register is attempted concuurently with a NEXT command, the current values of those registers will be returned. This is true both for read operations and for sends that use values from the incoming message registers. Any access after that will return a value from the new message. If there is no new message (ie. VALID is 0) then reads will return an indeterminate value. For testing purposes the incoming message registers are writable from the Pbus interface. However these registers should not be written when the incoming message registers are being loaded from the queue. If this occurs the value is indeterminate.

## 3.3 Network Output Port

The Network Output Port takes the head message off of the outgoing message queue and sends it into the network following the PaRC protocol. The message fields will be placed into the packet in the following way:

| **Message Format** | | |
|---|---|---|
| word | Upper Byte | Lower Byte |
| 0 | header[15:8] | header[7:0] |
| 1 | o0[15:8] | o0[7:0] |
| 2 | o0[31:24] | o0[23:16] |
| 3 | o1[15:8] | o1[7:0] |
| 4 | o1[31:24] | o1[23:16] |
| 5 | o3[7:0] | o2[7:0] |
| 6 | o3[22:16] | o3[15:8] |
| 7 | o4[7:0] | o3[31:24] |
| 8 | o4[22:16] | o4[15:8] |
| 9 | o2[15:8] | o4[31:24] |
| 10 | o2[22:16] | o2[31:23] |
| 11 | 0x55(ignored) | 0x55(ignored) |

where word 0 is more precisely defined as:

| **Header** | | | | |
|---|---|---|---|---|
| 15 | 14 | 13 | 12:8 | 7:0 |
| 1 | CSP | 1 | OTYPE | o0[31:24] |

This somewhat bizzare arrangement was chosen since it would allow the existing I-structure boards to be used. This creates a correspondence in the message between the following message fields and Monsoon token fields:

| **Correspondance between messages and Tokens** | | |
|---|---|---|
| Message field | use in RSTART | Token field |
| o4,o3 | Value | Value field of Value-part |
| o0 | FP | PE:FP field of Tag-part (bits 31:0) |
| o1 | IP[23:0] | Port:IP field of Tag-part (bits 55:32) |
| o1 | IP[31:24] | Rest of Tag-part (bits 63:56) |
| o2 | Unused | Type Info and Unused |

I-structure boards could be used in the following way: When sending to an I-structure board the o0 field would be used to indicate the location being accessed; the o4-o3 fields would be used to indicate the value being stored, or the IP-FP that the result should be sent to; and the o1 field would be used to indicate what type of operation was being done. The I-structure board would return (rstart) messages in a similar way, giving values for the o0(FP), o1(IP), o3-o4(VAL) fields. The IS board's ability to modify the last 3 bits of the IP would be used to allow defered lists to be implemented.

This message is sent into the Network according to the PaRC protocol. The Output port interface consists of the following signals.

| Network Output Port signals | | |
|---|---|---|
| Signal | Dir. | Description |
| nodata[15:0] | O | Output Data bus |
| noclk | 0 | Output Clock |
| nowait | I | Incoming WAIT signal |

Data is sent out on the nodata bus, the value on this bus is expected to be stable during each rising edge of the noclk signal. When no message is being transmitted, the value on nodata will always be the idle pattern 0x5555. When ready to send a message, the output port must first look at nowait, the incoming wait signal. If nowait is low the output port can begin to send a message, otherwise it must wait until nowait becomes low. Once a message can be sent, each of the 12 words of the message is sent out on consecutive cycles. Once a message is completed, a new message can be sent out immediately following it. Of course, this can be done only if nowait is low and there is a waiting message, otherwise the idle pattern must be sent out until a new message does begin. The PaRC protocol uses the uppermost bit of the data bus to determine when a packet begins. This bit should be 0 when no packet is being sent, and must be 1 in the first word of the packet. During the other words of a packet it can take on any value. The above description guarantees that the NI chip conforms to this specification as this bit is 0 during idle words (0x5555) and is always 1 in the header.

The outgoing clock, noclk can run at a faster speed than the main clock. To allow us to do this a separate clock, netclk, will be input to the chip and will be used to create noclk. It has not yet been determined if this clock will be synchronized to the main pbus clock.

## 3.4 Network Input Port

The Network Input Port receives messages from the network and places them onto the tail of the incoming message queue. The message fields are extracted from the packet according to the chart given in the section on the network output port. The signals making up this interface are:

| Network Input Port | | |
|---|---|---|
| Signal | Dir. | Description |
| nidata[15:0] | O | Incoming Data bus |
| niclk | 0 | Input Clock |
| niwait | I | Outgoing WAIT signal |

Packets are received on the NIDATA bus; this bus is assumed to have valid data on each rising edge of niclk. niclk may be completely asynchronous from the main pbus clock.

In order to determine where packets begin and end, the uppermost bit of the input data is used as a start-of-packet bit. This bit is always set to 1 in the header of a packet, but is used as a normal data bit during the rest of the packet. When no packet is being sent this bit will always be a 0. When not receiving a packet, the input port watches this bit to determine if a packet is starting. Once it goes high the receiver know it has the header of the packet. Since packets can not be interrupted, the next 11 words must be the rest of the packet. After those words the input port begins watching the start-of-packet bit again. When not receiving a packet the input port expects the data lines to contain one of two specified idle words (0x5555 or 0x2AAA). If no packet is being received and the data lines do not contain one of these patterns, an idle_error occurs.

Once a packet is received it is placed onto the input message queue. When this queue is close to being full, the input port must assert the niwait signal. This signal will tell the chip sending to this port not to begin to send any more packets. The input port asserts this signal soon enough so that it will not receive a packet that it does not have room for. If a packet arrives for which there is no room a room_error occurs.

# Bibliography


[1] Christopher F. Joerg. "Design and Implementation of a Packet Switched Routing Chip." *MIT/LCS/TR-482*, December 1990.

[2] Motorola Inc. "MC88100 RISC Microprocessor User's Manual" 1990.