JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 8, 169–179 (1990)

NAP (No ALU Processor): The Great Communicator

BRADLEY C. KUSZMAUL*

Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts 02139

AND

JEFF FRIED†

Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts 02139; and GTE Laboratories, Waltham, Massachusetts 02254

Message routing networks are acknowledged to be one of the most critical portions of massively parallel computers. This paper presents a processor chip for use in a massively parallel computer. The programmable approach used in this processor provides enough flexibility to make it a "universal" part for building a wide variety of interconnection networks and routing algorithms. A SIMD control scheme is used to make programming and synchronizing large numbers of processors simple. In the course of designing this processor, we were faced with the decision of which logic operations to implement in an Arithmetic-Logic Unit (ALU); informal design studies showed that it was best to provide none. The processor performs all computations by a sophisticated table lookup mechanism, and has no ALU; it is thus called the No ALU Processor (NAP). Using tables rather than an ALU provides a very flexible instruction set, and in real programs often allows more than one "operation" to be done in one cycle. Benchmarks written for the NAP show that indirect addressing mechanisms can speed many common operations by a factor of about log N on an N-processor machine. We have therefore provided hardware to support indirect addressing, or a Multiple Address Multiple Data operation. In addition, the NAP contains local storage used for flexible instruction decoding: the same instruction can result in different operations on different chips. These two mechanisms allow programmers to write programs for NAP machines easily using SIMD style, and also provide the power of different computations happening simultaneously in different parts of the machine. It is possible to build and efficiently simulate, using NAP chips, a wide variety of communications networks, including hypercubes, butterflies, fat-trees, and networks for computing parallel prefix operations. By this informal measure, the NAP architecture is a universal part for building interconnection networks and running network algorithms. © 1990 Academic Press, Inc.

* Partially supported by ONR, Reference Number N000 14-83-K-0 125, and by ONR, Reference Number N000 14-84-K-0099.

† Supported by GTE Laboratories, DARPA Contract N00014-87-K-0825, and ONR Contract N00014-86-K-0593.

I. INTRODUCTION

Message routing networks for parallel supercomputers occupy a unique place in the spectrum from specialized to general-purpose machines. Although these routing networks can be used to build general-purpose parallel computers (as well as specialized computers), they themselves are usually built out of very specialized hardware. This paper presents a single processor design which is useful for building a variety of different networks; in this sense it is a general-purpose element within the specialty of interconnection networks. This processor is an experimental design incorporating several novel architectural features which make it simple to program, general purpose, and efficient. Specifically, no Arithmetic-Logic Unit (ALU) is provided in the processor. The arithmetic functions normally performed by an ALU are instead performed by table lookups into memory. In addition, a very flexible programming model is provided, which supports indirect addressing and multiple concurrent instructions while operating in a Single-Instruction Multiple-Data (SIMD) mode or Multiple SIMD (MSIMD) mode.

The No ALU Processor (NAP) chip described in this paper is the result of a design experiment which explores architectures for communication network support. The experiment has three main design goals:

- Act as a "universal" element for routing networks. By universal we mean both general purpose and efficient. The performance of the NAP when used as a node within a network should be as close as possible to the performance of a special-purpose chip designed especially for that network.
- Provide communications control which is as flexible as possible.
- Keep the processor's input-output channels (which connect to other NAP chips) and memory as busy as possible performing useful work.

In the course of designing the NAP, we were faced with the decision of which logic operations to implement in an ALU; informal design studies showed that it was best to provide none. Using tables rather than an ALU provides a very flexible instruction set, and in real programs often allows more than one "operation" to be done in one cycle. One of the most interesting lessons from the design of the NAP was that table lookup is a very powerful mechanism.

It is interesting to note that there are two other examples of ALU-less processors, both from the early 1960s. One of these, called the CADET, was built and marketed by IBM [1]. The other was proposed by Ferroxcube Corp. [9], a manufacturer of core memory. Both of these machines were explored because the cost and performance of memory had been greatly improved with the invention of core memory. As the density of memory compared to logic continues to increase, a somewhat similar situation exists today.

A collection of NAP chips can be wired together and can be programmed to simulate many things. We have programmed our simulators to perform several important parallel algorithms, including reduction and parallel prefix in a tree network [2], connection-machine style routing on a cube-connected cycle [5], and cellular automata programs (such as Conway's game of Life) [11]. We are able to support any network with a large number of nodes (up to about 2^{32} nodes) of constant degree, including fat-trees [6, 4], butterfly networks [12, 8, 10], cube-connected cycles, trees, and meshes.

Section II of this paper describes the instruction set architecture of the NAP. Section III gives three examples of the programming and operation of the NAP. Section IV discusses the implementation of the NAP chip. Finally, Section V evaluates the NAP in the light of our design goals, and summarizes the lessons learned from this project.

II. INSTRUCTION SET ARCHITECTURE

We adopt the (M)SIMD model, where one or more microcontrollers broadcast microinstructions to sets of processors; each set of processors is controlled by one microcontroller. The microcontroller handles all instruction sequencing (e.g., loops and branches) and, as such, must be as powerful as a conventional computer. In this SIMD model all processors are globally synchronized at the instruction level.

A top-level view of a NAP computer system is shown in Fig. 1. Global microcontrollers broadcast instructions (a single microcontroller is shown at the left) to collections of NAP chips (shown at the right). Each NAP is connected to its neighbors through up to eight bidirectional data channels. Each bidirectional data channel is capable of sending 1 bit and receiving 1 bit per clock cycle. The NAP processors also have an off-chip static random access memory (SRAM).

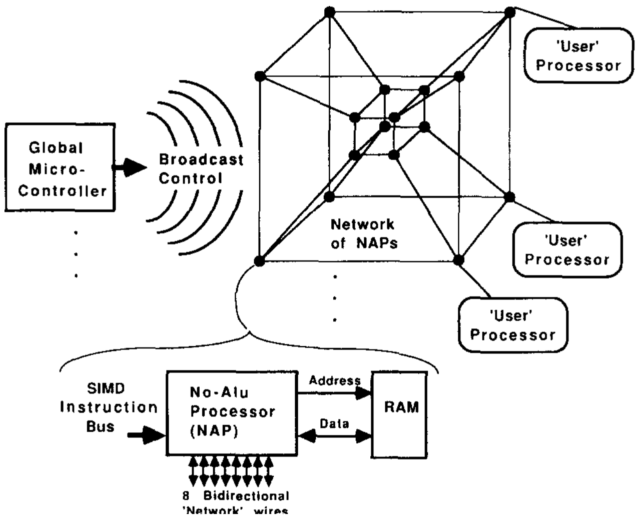

FIG. 1. System-level view of a NAP-based computer. Global microcontrollers broadcast instructions to sets of NAP chips. Each NAP chip is connected to an off-chip SRAM, a SIMD instruction broadcast bus, and eight bidirectional network lines.

The bidirectional channels may be connected in any fashion to form an interconnection network; the NAP chips form the nodes of that network, and may do computations in parallel to perform routing, do actual computing for the system, or both. Examples of networks which can be built using NAPs are butterfly or fat-tree networks, banyan or flip-type networks, hypercubes (more than 2^8 processors require multiple NAPs per node), cube-connected cycles, shuffle-exchange networks; torus and mesh networks, restructurable networks, and trees. One restriction is that the networks are regular enough to have fewer than 16 distinct types of nodes; most practical networks have 1 or 2.

A. *Indirect Addressing and MIMD*

One very important mechanism provided by the NAP which is not found in conventional SIMD computers is indirect addressing. We support indirect addressing because of the wave nature of the computations performed by many routing networks. Consider, for example, parallel prefix [2], which is a class of parallel algorithms which use a tree interconnection structure between processors to perform many operations (such as addition) in log N time on N processors. At any stage of a parallel prefix computation, different levels of the tree may be accessing bits at different addresses. Conventionally, this would be handled by enabling or disabling the processors at different levels of the tree, and running the computation on different levels at different times, thus slowing down the overall computation to time $\log^2 N$. Indirect addressing provides a mechanism for different processors to access different memory addresses at the same time under SIMD control. The result is that parallel programming can be done more flexibly and more efficiently (by a factor of log N).

In addition to indirect addressing, there are three means of differentiating processors within the SIMD control structure and hence making programming more flexible and efficient:

1. Conditional execution: the instructions broadcast on the SIMD bus can conditionally load a local instruction store called the nanostore, conditionally load the memory, conditionally load configuration bits within the NAP (called Input-or-State-Select or ISS bits), and conditionally execute sequences of instructions. An instruction may be conditionally executed depending on a combination of state bits and data from the input channel.
2. The instructions stored within the nanostore of each NAP may be different, so that different processors may perform totally different operations in response to the same broadcast instruction.
3. Processors can have different tables at the same address in local memory, and thus perform different functions even while they are accessing the same address.

These three mechanisms, which are explained in more detail below, provide a large degree of flexibility to NAP programmers.

B. *Instruction Philosophy*

We assume that off-chip wire delays are large compared to on-chip cycle times and local memory access time, since we are implementing systems with long wires (we use a capacitive model for the wires). Therefore, we chose an approach where there is a slow clock for operations between processors (the microcycle) and a fast clock for operations within a processor (the nanocycle). During one microcycle, the microcontroller broadcasts a microinstruction, which is executed by the processors. Each processor can read or write from each of its eight single-bit input-output channels. During one microcycle, there are four minor cycles called nanocycles. During one nanocycle, a nanoinstruction is executed. A nanoinstruction may access the external memory with a read-modify-write (which gives the processor the old value in memory and allows the processor to modify 1 bit of memory or to write a complete different value back to memory). The nanoinstruction also updates some internal processor state. The memory address may be changed once per nanocycle.

The NAP depends heavily on memory. As we have seen, a nanocycle may involve a complex memory access, so that the performance of the NAP is driven by memory performance. Most programs written for the NAP are also very memory-oriented. Operations are performed using tables in memory under the control of broadcast microinstructions. Typically, these table-based operations take as operands an arbitrary combination of state and input channel values, an integer, or an address. Each table (called a function table) requires 256 words (8 bits each). An example function table for adding two 2-bit numbers is shown in Fig. 2. Note that multiple functions can be combined in one table; the table shown in Fig. 2 has a number of unused bits (denoted by "X") that could be used for another function. Our prototype supports up to 2K words of external SRAM, so that up to eight different tables can be stored in memory at once; additional tables are downloaded as needed. Tables may be accessed using either direct or indirect addressing.

We adopted the principle that the implementation should allow as much flexibility as possible, unless that flexibility would be expensive. For example, we tried to avoid making any of the state bits special, but we needed 2 state bits to control the conditionalization. We decided that adding a multiplexor or increasing the width of the CC field to allow full generality would be too expensive. The conditionalization of processors is very general, since it is done by table lookup in the 4-bit CC field.

C. *The NAP Microword*

Table I shows the format of the NAP microword. This word is the instruction broadcast from a microcontroller to a number of NAP chips in (M)SIMD fashion each microcycle. The 39 bits of the microword are common to all the NAPs in a set. Each microword contains distinct operation codes for every nanocycle, as well as condition codes, a direct memory address, and two table offsets used for indirect addressing or table-based logical operations. The mi-

Word Address	output
xxxx 00 00	xxxxx 000
xxxx 00 01	xxxxx 001
xxxx 00 10	xxxxx 010
xxxx 00 11	xxxxx 011
xxxx 01 00	xxxxx 001
xxxx 01 01	xxxxx 010
xxxx 01 10	xxxxx 011
xxxx 01 11	xxxxx 100
xxxx 10 00	xxxxx 010
xxxx 10 01	xxxxx 011
xxxx 10 10	xxxxx 100
xxxx 10 11	xxxxx 101
xxxx 11 00	xxxxx 011
xxxx 11 01	xxxxx 100
xxxx 11 10	xxxxx 101
xxxx 11 11	xxxxx 110

FIG. 2. Example function table for two-input 2-bit addition. The word address is composed of some unused high-order bits (which could be used for another function), and two binary 2-bit inputs. The output consists of 5 unused high-order bits, and a 3-bit result. Unused bits are denoted by "X". Note that each line of this description specifies 16 words of the function table.

TABLE I

The Microinstruction Word Format Shows the Mnemonics, Functions, and Width of Each Instruction Field

Mnemonic	Function	Width (bits)
INIT	Initialization and download control	2
OP0	Four-bit indexes into the nanostore which	4
OP1	specify which nanoinstruction to perform	4
OP2	in each nanocycle; OPs share one address	4
OP3	and condition code	4
C C	Condition code; this decodes to 16 conditions	4
MIP	Memory address (for direct addressing)	II
FA	Function table offsets (for direct addressing)	3
FB	normally contain the start address of a table	3
Total number of microword bits		39

croword is also very memory-oriented; 17 of its 39 bits are used for memory addressing.

The microword does not contain the actual nanoinstructions executed each nanocycle by the NAPs. Rather, it contains four 4-bit OP codes which specify an address in an on-chip memory called the nanostore. The nanostore contains the nanoinstructions in the form of a bit for every control line needed by the NAP hardware. The nanoinstructions are loaded under program control and may differ for different processors. The OP fields give the "address" of the nanoinstruction within the nanostore. This approach reduces the number of bits broadcast to the processors and thus economizes on chip pins. In addition, it provides a mechanism for different processors to perform different work under the control of the same microinstruction, since different processors may have different nanoinstructions loaded into the same address in the nanostore.

D. Processor Organization

A block diagram of the NAP processor is shown in Fig. 3. Table II shows the mnemonics used in the datapaths of the NAP processor, and Table III shows the fields of the nanoinstruction. An instruction latch (shown on the left in Fig. 3) is used to hold each microinstruction from the SIMD instruction stream. It operates once per microcycle. There are multiple NAP processors per chip; our implementation has four processors per chip. Only one instruction latch is needed per chip, although in our implementation we provided one latch per processor for expediency. The nanocontrol (shown at upper left) sequences through the four OPs, one per nanocycle, translating each OP into control signals. This translation must be provided on an individual processor basis, since different processors should be able to execute different nanoinstructions while executing the same OP. The nanocontrol also reduces the number of bits in the microinstruction (and hence reduces the num-

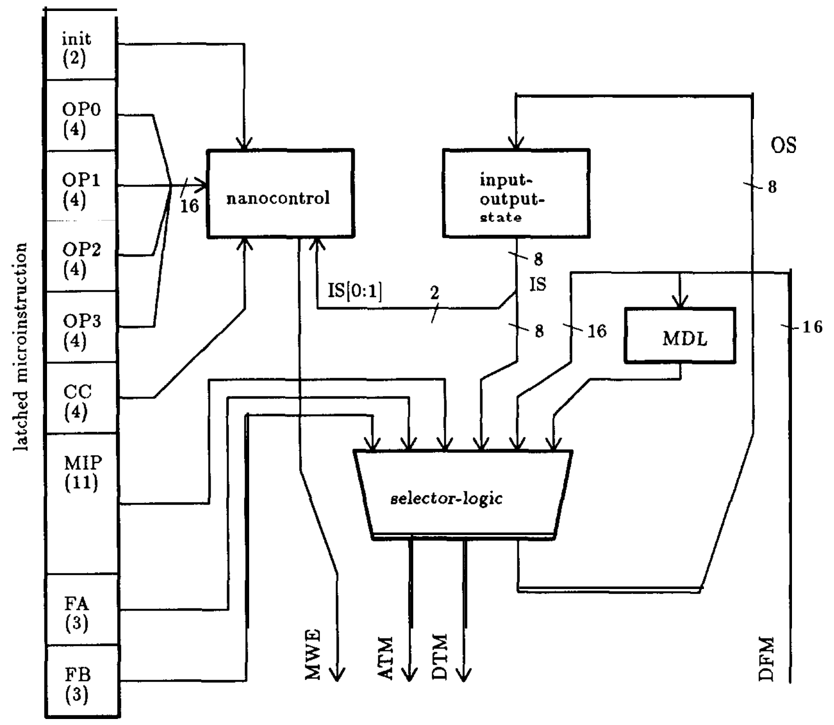

FIG. 3. The block diagram of the NAP chip shows (in clockwise order) the SIMD instruction latches (left), the nanocontrol, the input-output-state, the MDL, and the selector-logic. The RAM interface is shown at the bottom.

TABLE II

Mnemonics Used in the Datapaths of the NAP Processor

Mnemonic	Description	Width (bits)
ATM	Address To Memory	II
DFM	Data From Memory	8
DTM	Data To Memory	8
BITADDR	BIT ADDRess	3
BITVAL	BIT VALue	1
I	data from Input channels	8
0	data to Output channels	8
S	State data	8
IS	combined Input-State data	8
OS	combined Output-State data	8
M M	Mix-and-Match value	8
ISS	Input-State-Select (register)	8
MDL	Memory-Data-Latch (register)	8

ber of pins per chip) since each of the four nanoinstructions is encoded into 4 rather than 28 bits.

The input-output-state (shown at upper right) controls all the input-output data channels (i.e., wires) for the processor. Each input-output-state unit can be individually programmed to act as a bidirectional input-output channel or as a single bit of internal state.

The memory-data-latch (MDL, shown at middle right in Fig. 3) holds a word from memory. The MDL is simply a latch with write-enable. The nanoinstruction determines whether the MDL retains its old value or latches the current value being read from memory.

The selector-logic provides address calculations (for indirect addressing) and data selection. These calculations are performed by a combinational circuit that computes the function described in Subsection E, below. The selector-logic produces the address to memory (ATM), data to memory (DTM), and data to the input-output-state (OS).

Within each nanocycle the NAP processor behaves as follows (refer to Fig. 3) . The data flow through the NAP circuitry as follows. Each nanocycle, the selector-logic uses the MIP, FA, and FB fields of the microinstruction, the current value of the MDL, and the current IS value (derived from the S-bit state register and the data input pads) to compute a memory address (ATM, shown leaving downward from the selector logic). The memory responds with data from memory (DFM, shown arriving at the lower right). The value of DFM is supplied to the MDL and the selector-logic. The selector-logic then uses the previously available data plus the value of DFM to compute new DTM. The value of DTM may be written to the same memory location indexed by the memory address generated earlier in the nanocycle. Whether DTM is actually written to memory is controlled by the memory-write-enable (MWE, which is a 1 -bit nanoinstruction field). The DTM and MWE signals are shown leaving at bottom center.

The nanocontrol (shown in Fig. 4) generates all of the control signals for the other parts of the NAP processor. Generally, the control signals are not shown leaving the nanocontrol in the figures. The nanocontrol consists of a nanostore (a 16 by 28 SRAM, shown center right) and a

TABLE III

Fields of the Nanoinstruction

Mnemonic	Description	Width (bits)
MMM	Mix-and-Match M UX	8
BDAS	Bit-Direct-Address-Select	1
FAS	Function-Address-Select	2
MAS	Middle-Address-Select	1
LAS	Index-Address-Select	1
MWBS	Memory-Write-Bit-Select	3
MWS	Memory-Write-Select	1
RMWBIT	Read-Modify-Write BIT	1
MWE	Memory-Write-Enable	1
OSWE	Output-State-Write-Enable	8
LMDL	Latch MDL	1
Total number of nanostore bits		28

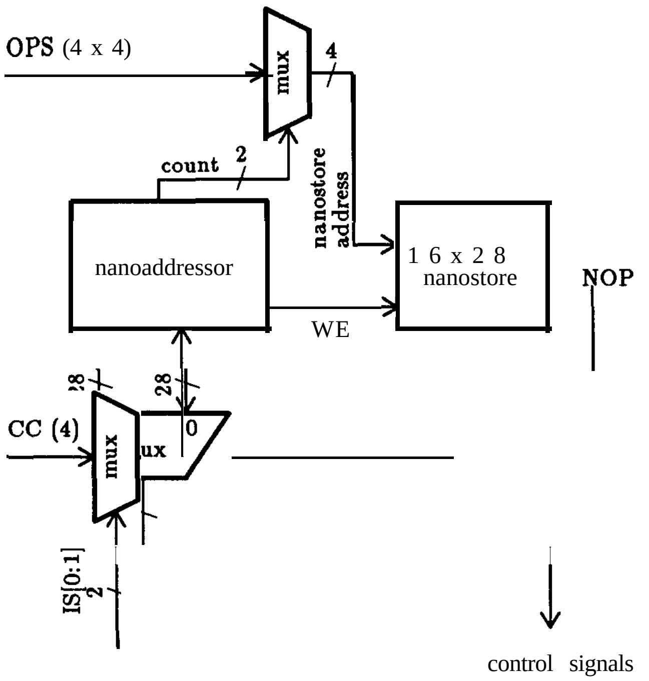

FIG. 4. The nanocontrol consists of a nanoaddressor (left center), a mux to select the appropriate OP for each nanocycle (top), the nanostore (a RAM, at right center), a mux to select the condition code (bottom left), and a mux (bottom right) to conditionally turn off the processor by selecting either the control signals from the nanostore or the constant encoding for a no-operation.

nanoaddressor (shown center left) , which is essentially a 2-bit counter with some extra control logic. The value of the 2-bit counter is used to select which of the four OPs to use. The selected OP (a 4-bit value) is used to address 1 of the 16 words of the nanostore (shown center right). One of the four CC bits from the microinstruction is selected by the low-order 2 bits of IS to conditionalize the processor. If the processor is disabled (the CC bit is zero), then the control signals are set to values which result in no state change, and if the processor is active (the CC bit is one), the outputs of the nanostore are the control bits used directly by the logic in the processor.

The input-output-state (shown in Fig. 5) consists of input pads, output pads, some latches, and some selectors. The OS signal comes from the selector-logic and is conditionally latched into the S bits (shown as the top row of boxes). The write-enables for the S bits, named OSWE[0] through OSWE [7], are part of the nanoinstruction. The ISS bits (part of the nanoinstruction) select between the I bits (from the input pads) and the S bits to generate the IS bits. On nanocycle 3, the output-latch latches the S bits and sends the value to the output pads for transmission during the next microcycle.

There are provisions in the NAP chip to conditionally initialize the ISS, the nanostore, and other internal state. The INIT field of the microinstruction is used to distinguish between a normal operation and an initialization operation. It is important that such initialization be through the SIMD instruction stream (rather than, e.g., through a diagnostic scan path) so that the initialization can be done in parallel. The details of the initialization circuitry are not described in this paper.

E. *The Function Computed by the Selector-Logic*

A number of memory addressing modes and data selection operations are supported by the NAP. Bit-read, bit-write, word-read, and word-write modes are supported (with 8-bit words). To achieve the effect of these various addressing modes, the programmer must carefully write code to generate the address, modify the bits, and write values back when appropriate. The hardware supports these operations so that they can be done quickly, but the programmer is still aware of the hardware.

Every nanoinstruction, a new memory address is constructed. This address is used in a read-modify-write operation. To read the memory without changing it one inhibits the write-enable to the memory. To modify 1 bit of memory, one reads the memory, computes a new byte with 1 bit modified, and writes the modified byte back to memory. To perform a write, one ignores the data being read.

Input, Output, and State. An 8-bit state register, called S, and an 8-bit input signal, called I, are available on every NAP processor. Those 16 bits are combined into an 8-bit value, called IS. Another 8-bit register, called ISS, controls which 8 of the input-output and state bits are available to

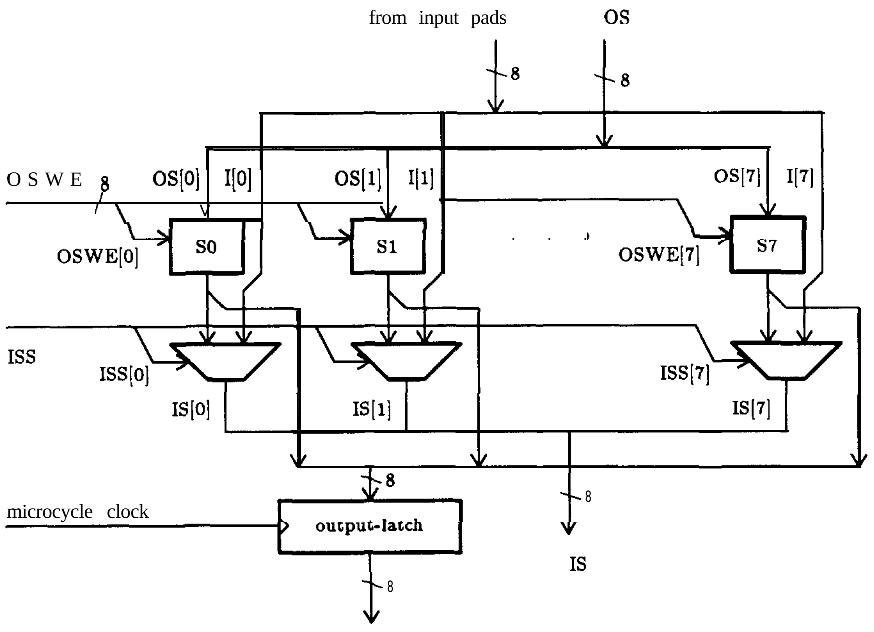

FIG. 5. The input-output-state consists of eight l-bit latches, with individual write-enables (controlled by OSWE shown coming from the left), eight muxes (below the boxes) individually controlled by ISS (shown coming from the left), the eight input pads (the data from the input pads is shown at top left), and the eight output pads (not shown).

the user as the IS value, according to the following function: For each $i \in \{0, \ldots, 7\}$,

$$\mathrm{IS}[i] := \begin{cases} \mathrm{I}[i] & \text{if ISS}[i] = 0, \\ \mathrm{S}[i] & \text{otherwise.} \end{cases}$$

The ISS is set under program control.

The MDL and the Mix-and-Match Value. An 8-bit register called the MDL can be loaded with the DFM on every nanocycle. Whether the MDL latches a new value or keeps its old value is determined by LMDL, a l-bit field in the nanoinstruction.

An 8-bit intermediate value, called MM (for mix-and-match), is a combination of MDL and IS. A programmer can use MM to combine some bits from memory, some bits from input data channels, and some bits from internal state, in order to compute the next state through a table lookup. The value of MM is determined by the following equation: For each $i \in \{0, \ldots, 7\}$,

$$\mathrm{MM}[i] = \begin{cases} \mathrm{IS}[i] & \text{if MMM}[i] = 0, \\ \mathrm{MDL}[i] & \text{otherwise,} \end{cases}$$

where MMM is an 8-bit field in the nanoinstruction.

Generating the Memory Address. The memory address is used to provide computation via a function table, or to load and store data from memory. It is an 11 -bit value, generated as follows:

$$\mathrm{ATM}[0:4] := \begin{cases} \mathrm{MM}[0:4] & \text{if IAS} = 1, \\ \mathrm{MIP}[0:4] & \text{otherwise.} \end{cases}$$

$$\mathrm{ATM}[5:7] := \begin{cases} \mathrm{MM}[5:7] & \text{if MAS} = 1, \\ \mathrm{MIP}[5:7] & \text{otherwise.} \end{cases}$$

$$\mathrm{ATM}[8:10] := \begin{cases} \mathrm{MIP}[8:10] & \text{if FAS} = 0, \\ \mathrm{FA} & \text{if FAS} = 1, \\ \mathrm{FB} & \text{if FAS} = 2. \end{cases}$$

The definition of MM is given above. The microinstruction provides MIP, FA, and FB. The nanoinstruction provides IAS, MAS, and FAS. A new memory address is generated every nanocycle, although the values of MIP, FA, and FB are held constant for the whole microcycle.

An additional 3 -bit intermediate value called the BITADDR is computed according to the function

$$\mathrm{BITADDR} = \begin{cases} \mathrm{MDL}[0:2] & \text{if BDAS} = 1, \\ \mathrm{FA} & \text{otherwise,} \end{cases}$$

where BDAS is a l-bit nanoinstruction field. The value of BITADDR is used to index a single bit inside a byte to implement bit-read and bit-write operations.

Generating the Data to Memory (DTM). Every nanocycle, a byte of data is written back to memory. That byte is computed by the function

$$\mathrm{DTM} = \begin{cases} \mathrm{DFM}[\mathrm{BITADDR} \rightarrow \mathrm{BITVAL}] & \text{if RMWBIT} = 1, \\ \mathrm{MM} & \text{if RMWBIT} = 0 \text{ and MWS} = 0, \\ \mathrm{OP2} \parallel \mathrm{OP3} & \text{otherwise.} \end{cases}$$

In words, if RMWBIT = 1, we modify the BITADDRth bit of DFM to be BITVAL and store the modified value back to memory. Otherwise we may store the mix-and-match value to memory, or we may store an immediate value (e.g., for a function table initialization).

A l-bit intermediate value, BITVAL, is computed by

$$\mathrm{BITVAL} = \mathrm{IS}[\mathrm{MWBS}].$$

The nanoinstruction provides RMWBIT (1 bit), MWS (1 bit), and MWBS (3 bits). Except during initialization, MWS is always zero. In one of the initialization modes (indicated by the INIT microinstruction field), only OPO is allowed to execute. In that case, OP2 and OP3 can be used as immediate data to memory.

Generating OS. The value of OS, which is supplied to the input-output-state, is computed as

$$\mathrm{OS} = \begin{cases} \mathrm{fanout}(\mathrm{DFM}[\mathrm{BITADDR}]) & \text{if MWE} = 1 \text{ and RMWBIT} = 1, \\ \mathrm{DFM} & \text{otherwise,} \end{cases}$$

where the fanout function produces an 8-bit value from a 1 -bit value according to

$$\mathrm{fanout}(0) = 0$$
$$\mathrm{fanout}(1) = 255$$

and where MWE and RMWBIT are both 1 -bit nanoinstruction fields.

F. *Providing a Global "OR" Tree*

A global "or" line to the microcontroller (the computer which broadcasts the SIMD instruction stream) can be derived from any of the output channel bits by ORing one data output wire from each chip together to compute a global "or". This capability is extremely useful for checking a global condition (e.g., whether any processor contains zero, or whether any processor's memory contains a pattern which matches the broadcast pattern). One such global "or" can be returned to the global microcontroller every microcycle, although the time between when the microcontroller broadcasts the microinstruction to when the corresponding global "or" arrives back at the microcontroller may be several microcycles due to pipelining. The distance from the microcontroller to the NAP chips through the SIMD bus and back through the global "or" tree might be farther than can be achieved during one microcycle (200 ns), so that programmers using the global "or" mechanism must take account of the pipelining effect. Any data output pin on the NAP may be used to construct a global "or" tree. In fact, it is possible to have several global "or" trees, by devoting several data output pins on each NAP to the global "or" tree.

III. PROGRAMMING EXAMPLES

One of the best ways to appreciate the programming mechanisms provided by the NAP is to study some example programs. Three examples are presented in this section. For each example, some tables have been preloaded into the external SRAM and some nanoinstructions have been preloaded into the nanostore. Each program is one microcycle in length, and the four OPs are bracketed by (OP). The notation M[a‖b‖ c] in the following programs indicates that a memory reference is being made using an address built by concatenating bits a, b, and c, where c is the low-order bit.

In our notation, "Mwrite" is the symbol for a memory write cycle, and "RMW" indicates a read-modify-write cycle (to modify a bit). The latter consists of a read from a given address in the first phase of a nanocycle, followed by a write of different bits to the same memory address in the second phase of the same nanocycle. Finally, the values of the condition codes, the MIP field, and the FA and FB pointers are specified.

A. *Example 1: Queuing Data in Memory*

Data queuing is a good example of a need for indirect addressing; for example, each processor may wish to receive or send a message at a locally specified address. This program takes one data input and buffers it in a queue in memory. The input is the bit at IO, and the queue's tail pointer is at location Q.

```
OPS=(M[MIP],   LatchMDL),
    ⟨M[MIP[5:10] ‖ MDL[3:7]],
     RMW(IO, MDL[0:2])⟩,
    ⟨M[FA ‖ MDL], LatchMDL),
    (MwriteMDLintoM[MIP])
CC=true,MIP=Q,FA=(theINCtable),
FB=don't-care
```

The way this program works is that word Q contains an 8-bit "bit pointer" to a bit on the same 256-bit page as Q. During OPO, the MDL is loaded with that bit pointer. During OP1, the NAP uses the high-order bits of MIP and the high-order bits of that bit pointer (in MDL) to address the word containing the bit which needs to be written. It then does a read-modify-write on that word, modifying the bit specified by the low-order 3 bits of MDL. The new bit is IO. The modified word is written back to memory. During OP2, the program uses FA and MDL to look up 1 + MDL (i.e., the increment of MDL) and latches that value into MDL. This is the new queue pointer. Finally, the NAP stores the new queue pointer back to M [MIP] during OP4.

This program requires four words of the nanostore to have been preloaded, and it requires one function table (the increment table). It could be used as an instruction within another program, or could be generalized to other applications. For example, to move N different input bits to different buffers in memory requires N more preloaded nanoinstructions and would take N more microcycles to execute. (Note the nanoinstructions addressed by OPO, OP2, and OP3 could be reused in the other microcycles; only the nanoinstruction corresponding to OP1 would have to be changed.)

B. *Example 2: Circuit Switching Using Permutations*

We want to take four inputs, from I, and put them to the four outputs, 0, by some permutation which is specified in memory. When this is done repetitively, it provides four circuit-switched channels through the NAP, which operate at 1 -bit per microcycle. This permutation could be a different one for each microcycle, so that the circuit-switched connections set up could be as short as 1 bit time.

We will use the low-order bits of word 0 to specify the routing for 04 and 05, and we will use the low-order bits of word #x 100 to specify the routing for 06 and 07. (Note that #x 100 is hexadecimal notation, so that #x 100 = 256.)

```
OPS=(M[MIP], Latch(S0, S1, S2, S3)⟩,
    (M[FA ‖ Is], latch04, O5⟩,
    ⟨M[FB ‖ MIP[0:7]],
     Latch(S0, S1, S2, S3)⟩,
    ⟨M[FA ‖IS], latch06, O7⟩
CC=true,MIP=0, FA=(permutationtable),
FB=1
```

In this program, 2 bits of the permutation are computed every two nanoinstructions. During OPO, the NAP uses MIP = 0 to load word 0 and latch the low-order 4 bits into

SO, S 1, S2, S3. During OP 1, FA is used to name a function table, and IS is the index into the table. The table gives us two useful outputs (from bits 4 and 5) which are latched into 04 and 05. During OP2, the NAP uses FB = 1 and MIP = 0 to load word **#x** 100 and latch the low-order 4 bits into SO, S 1, S2, **S3**. Finally, during OP3, FA is used to name the same function table as in OP 1, which provides another two useful outputs (on bits 6 and 7) which are latched into 06 and 07.

This example uses four nanostore locations and one function table (which could even be stored on page 0 or 1 since we only use the high-order bits of any word for the table, and the permutations only use the low-order bits of the table). Since the permutations could also depend on the bits, this example also generalizes to some cases of sorting or packet switching.

C. *Example 3: Parallel Prefix Addition*

This program does the inner loop of a global reduction using addition for a tree machine (i.e., to sum up the values stored in all the processors to get a global total). This is one example of the class of parallel prefix programs (for a more detailed treatment of parallel prefix, see Borodin and Hopcroft [2]). The program uses one nanocycle per microcycle, and calculates 1 bit of the sum per microcycle. The performance of this algorithm is to sum bits at 5 megabits per second (with our 200-ns microcycle clock period). One improvement on this program would be to speed things up by using four different input-output channels to effectively ship 1 bit of data up the tree every nanocycle instead of every microcycle so that the data rate for this program could be 20 megabits per second.

```
OPS=⟨M[FA ∥ IS], latch 0, S0),
     ⟨nop⟩, ⟨nop⟩, ⟨nop⟩
CC=true, MIP=⟨don't-care⟩, FA=⟨full-adder table⟩, FB=⟨don't-care⟩
```

The single nanoinstruction takes two l-bit inputs, has 1 bit of state (the carry bit), and writes 1 bit of output. Only one table is needed (a full adder table, similar to the one shown in Fig. 2).

IV. IMPLEMENTATION

The NAP is designed using a fully static CMOS circuit methodology in MOSIS scalable CMOS design rules. A two-phase nonoverlapping clocking approach is used; approximately half of the circuitry on the chip (and exactly half of the control lines) is “active” on phase 1, while the other half is active on phase 2. The MAGIC layout system was used for the layout of the chip. Each chip contains four NAPs, although only one of these processors is fully connected to the pins of the chip. The other three processors are accessible through scan path circuitry. The overall circuit is 7900 by 9200 μm in a 3-μm CMOS process. The complete simulation and layout are complete, but the chip has not yet been fabricated.

The nanostore is an on-chip 16 by 28 SRAM with decoders, write amps, and sense amps. This SRAM has an access time goal of 25 ns, and is 1974 by 1620 μm in area.

The speed of the NAP processor is primarily determined by the speed of the off-chip SRAM (which must do a read-modify-write operation every nanocycle) and by the speed of the nanostore (which must do a read every nanocycle). It would otherwise be possible to speed up the rest of the NAP chip by at least a factor of two. In a future implementation, it would make sense to move this memory on-chip. This would increase the speed of the processor, reduce the number of pins per chip, and reduce the system cost by eliminating the relatively expensive off-chip high-speed SRAM. In our implementation, we chose to use external SRAM in order to reduce the complexity of the design. We chose 2K by 8 high-speed SRAM with 35 ns access time because they were the fastest affordable memory components available at the time of the original design (spring of 1987). This provides eight full function tables. We have found that eight tables is enough to avoid reloading function tables during the computation in most applications.

All latches within the design have been built with scan paths incorporated within them. Generally, neighboring latches will scan on alternate phases. The external clock signals are thus converted on-chip to two nonoverlapping active-high clock signals if the scan pin is low, or to scan clocks if the scan pin is high.

Our layout approach for NAP is straightforward: we run first-level metal horizontally to distribute power, ground, and clocks, and second-level metal vertically for data signals. Data signals are routed horizontally on either first-level metal or polysilicon. The Magic [7] router is used to route power and ground connections, connections between subcircuits, and connections between the NAPs and the padframe.

The NAP was simulated at the functional and the circuit level using simulators written in LISP. A high-level simulator, which includes some programming tools for defining function tables and nanocode, allowed programs to be written as the design of the NAP was progressing, which stimulated a number of design changes.

An RTL-like language was also constructed on top of LISP; this language allows us to compose parts to make larger parts, and to generate test vectors for ESIM, which we can use to drive the circuit extracted from the layout. A register transfer-level simulation was written in this language, which allowed the verification of individual cells and the composition of cells, all the way up to a full chip-level simulation. The results of this simulation were compared with the results of ESIM simulations to verify the layout. The RTL simulator generates test vectors to drive ESIM and it generates a file which contains the outputs that ESIM should generate if the layout is correct.

Although we have not implemented a controller, we believe that the microcycle-nanocycle approach would simplify its design. This is because the microcontroller is required to produce microinstructions at only 5 MHz (rather than at the 20-MHz nanocycle rate). The latency from the microcontroller to the NAP chip is not critical.

V. EVALUATION AND CONCLUSIONS

We have shown that it is feasible to design a processor chip which supports a variety of bit-serial routing networks efficiently. This type of chip is a step toward understanding how to build and operate interconnection networks for massively parallel computers. The NAP chip we have designed provides very flexible addressing mechanisms, and allows indirect addressing so that MAMD operation is possible. This chip also supports three distinct means of multithread operation, so that different processors operating off the same instruction stream can do different things. Finally, this processor chip has no ALU; table lookup is used for all operations. We have found all of these mechanisms useful in writing example programs, and believe that the NAP approach can teach designers about how to provide addressing and processor selection mechanisms in SIMD processors, and about the issues involved in providing flexible and high-performance interconnection networks.

How well has the NAP design stood up to its original design goals? Let us examine those goals one by one:

- *Provide communications control which is as flexible as possible.* The operation of the processor is completely programmable at both the microinstruction and the nanoinstruction level. Processors have considerable flexibility in addressing modes, and indirect addressing at both the bit and the word level is well supported. In addition, there are three distinct means for processors operating from the same instruction stream to do different things: in addition to the standard conditional execution (which is made very general in the NAP), they can have different nanoinstructions in their nanostore, or use different operation tables in their memory. In practice, this allows programmers to write programs with the simplicity implicit in SIMD control and synchronization, yet keep processors efficiently utilized doing different things at the same time. Essentially, one can program a machine built of NAPs as sets of processors, even if those processors share the same microcontroller.
- *Keep the input-output channels and memory as busy as possible performing useful work.* Each microinstruction may make up to four memory references, each of a read-modify-write nature. Every microinstruction executed by the processor can be able to read from and write to up to eight input-output channels on the processor. All of the programs written on NAP to date have been able to keep the input-output channels active at least 1 bit per microcycle, which corresponds to our assumptions about wire latency. Similarly, most of these programs use most of the nanocycles in a microcycle to perform useful work, so that memory is well utilized. The cycle time of the NAP is also in good agreement with the speed available from state-of-the-art commercial SRAMs or on-chip dRAM.
- *The NAP should serve as a universal element for routing networks.* To date, we have written NAP programs for message routing using algorithms designed for butterfly networks [10] using the same number of cycles as a node designed specifically for that purpose. We have also written NAP programs for parallel prefix [2] which execute in one microcycle per bit. Although these examples are not sufficient evidence to prove that NAP is in fact a universal communication element, they do indicate that NAPs would be useful in a number of different networks.
- *Experiment with an ALU-less processor.* Our experience in writing NAP programs using tables for operations is that "compressed tables," which do more than one thing in one operation, are immediately of use. For example, one portion of the table might be used to increment a pointer while another part might perform a Boolean operation on a few bits from input channels. We had hoped that experimenting with table-based operations might lead us to a choice of which operations to put into an ALU; instead, we discovered that the generality offered by these tables was just the right thing for programming.

The assumptions that were used in designing the NAP system should be carefully examined. In particular, our microcycle-nanocycle approach is implicitly based on the assumption that long-distance communication (e.g., between NAP chips) is much slower than short-distance communication (e.g., between the NAP processor and the function table memory). This assumption may be incorrect. It is true that the latency for long-distance communication is higher than for short-distance communication, but research done since the implementation of the NAP processor indicates that the clock rate for long wires might be substantially the same as the clock rate for short wires [3]. It may be difficult to use a NAP-like chip to effectively emulate special-purpose chips that treat their long-distance wires as transmission lines.

We hope that the NAP chip will eventually serve as a test-bed for experimentation with new interconnection networks and parallel algorithms. We plan to test the NAP design using a variety of "benchmark" programs and networks to test its utility as a general-purpose network element. Measurement of effect of indirect addressing and our processor differentiation mechanisms on processor utilization will tell us something about the efficiency of our approach. Finally, these mechanisms may lead to insights about how to write effective parallel programs for communication networks.

ACKNOWLEDGMENTS

The authors thank Tom Cormen and Elliot Kolodner for their hard work on the design, layout, and programming of the NAP; Bill Dally for his direction and feedback during the course of the NAP project; and Charles Leiserson for his insightful comments, direction, and consultation in the early stages of the NAP design.

REFERENCES

1. Bashe, C. J., Johnson, L. R., Palmer, J. H., and Pugh, E. W. *IBM's Early Computers.* MIT Press Series in the History of Computing, Cambridge, MA, 1986.
2. Borodin, A., and Hopcroft, J. E. Routing, merging, and sorting on parallel models of computation. In *Proc. 14th Annual ACM Symposium on the Theory of Computing, 1982,* pp. 338-344.
3. Knight, T., et al. Self terminating low voltage swing CMOS output driver. In *Proc. Custom Integrated Circuits Conference, 1987.*
4. Greenberg, R. I., and Leiserson, C. E. Randomized routing on fat-trees. In *Proc. 26th Annual IEEE Symposium on the Foundations of Computer Science, Nov. 1985.*
5. Hillis, W. D. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.
6. Leiserson, C. E. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* **C-34, 10** (Oct. 1985).
7. Osterhout, J. K. *Magic User's Manual.* U.C. Berkeley, Berkeley, CA, 1986.
8. Pippenger, N. Parallel communication with limited buffers. In *Proc. 25th Annual IEEE Symposium on the Foundations of Computer Science,* Oct. 1984.
9. Private correspondence. Ferroxcube/Amperex Electronic Corp.
10. Ranade, A. G. How to emulate shared memory. In *Proc. 28th Annual IEEE Symposium on the Foundations of Computer Science,* Oct. 1987, pp. 185-194.
11. Toffoli, T., and Margolus, N. *Cellular Autonoma Machines.* MIT Press. Cambridge, MA, 1987.
12. Valiant, L. G., and Brebner, G. J. Universal schemes for parallel communication. In *Proc. 13th Annual ACM Symposium on the Theory of Computing,* May 198 I.

Received February 1, 1989; revised July 1, 1989

BRADLEY C. KUSZMAUL received two S.B. degrees (in mathematics and in computer science and engineering) from MIT in 1984. He received a S.M. degree in computer science from MIT in 1986, and is now a Ph.D. candidate in the MIT Department of Electrical Engineering and Computer Science. His research interests include all aspects of parallel computing, including processor architecture, interconnection networks, algorithms, and programming methodologies. He worked on interconnection networks and parallel processing at Thinking Machines, in Cambridge, Massachusetts, during the summers of 1984 through 1987, and worked full-time at Thinking Machines during the 1987-1988 academic year. He is a member of ACM and IEEE.

JEFF FRIED is a founder and director of hardware development for Connect Telemanagement Systems Corp. From 1983 to 1989 he worked at GTE Laboratories, where his research on advanced switching architectures included building prototypes of broadband switches and developing system-level modeling and analysis tools. He received the B.S., M.S., and E.E. degrees from MIT, where he is a Ph.D. candidate in the Department of Electrical Engineering and Computer Science. His research interests include computer and telecommunication system architecture, VLSI and WSI design, optical interconnects, and parallel and distributed algorithms. He is a member of the ACM, IEEE, SPIE, and SIAM, has published over 25 technical papers, and holds three patents.