NAP (NO ALU PROCESSOR) THE GREAT COMMUNICATOR

§†Jeff Fried and §‡Bradley C. Kuszmaul

§MIT Laboratory for Computer Science
Cambridge, MA 02139

†GTE Laboratories
Waltham, MA 02254

‡Thinking Machines Corporation
Cambridge, MA 02139

ABSTRACT

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 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 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 $\log N$. We have therefore provided hardware to support indirect addressing, or Multiple Address Multiple Data (MAMD) 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.

Keywords: Universal, Table lookup, ALU, Parallel, Processor, Network, VLSI

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 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 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 SIMD or Multiple SIMD (MSIMD) mode.

The 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 I/O pins (which connect to other NAP chips) and memory as busy as possible performing useful work.

CH2649-2/89/0000/0383$01.00 © 1988 IEEE

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.

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 (Ref. 1), connection-machine style routing on a cube connected cycle (Ref. 3), cellular automata programs (such as Conway's game of Life) (Ref. 7). 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 (Refs. 4, 2), butterfly networks (Refs. 8, 5, 6), cube connected cycles, trees, and meshes.

Section of this paper describes the instruction set architecture of the NAP. Section discusses the processor design and the implementation of the NAP chip. Finally, Section evaluates the NAP in the light of our design goals, and summarizes the lessons learned from this project.

INSTRUCTION SET ARCHITECTURE

We adopt the (M)SIMD model of one or more controllers broadcasting microinstructions to sets of processors; each set of processors is controlled by one controller. The controller handles all instruction sequencing, like loops or branches. In this SIMD model all processors are globally synchronized at the instruction level. Each processor communicates with other processors through eight bidirectional wires. The bidirectional wires 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. A system-level view of the NAP is shown in Figure 1. 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. An important restriction is that the networks are regular enough to have fewer than 16 distinct types of nodes; most practical networks have one or two.

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 (Ref. 1), which is a class of parallel algorithms which use a tree interconnection structure between processors to perform many operations (such as addition) in logN time. At any stage of a parallel prefix computation, each level of the tree may be accessing a bit at a different address than other levels of the tree are accessing. 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. 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.

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 I/O-or-State-Select or ISS bits), and conditionally execute sequences of instructions. An instruction may be conditioned on any of the 16 bits of state within the NAP.

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.

Instruction Philosophy

We assume that off-chip wire delays are slow compared to on-chip cycle times and local memory access time, since we are implementing systems with long wires.

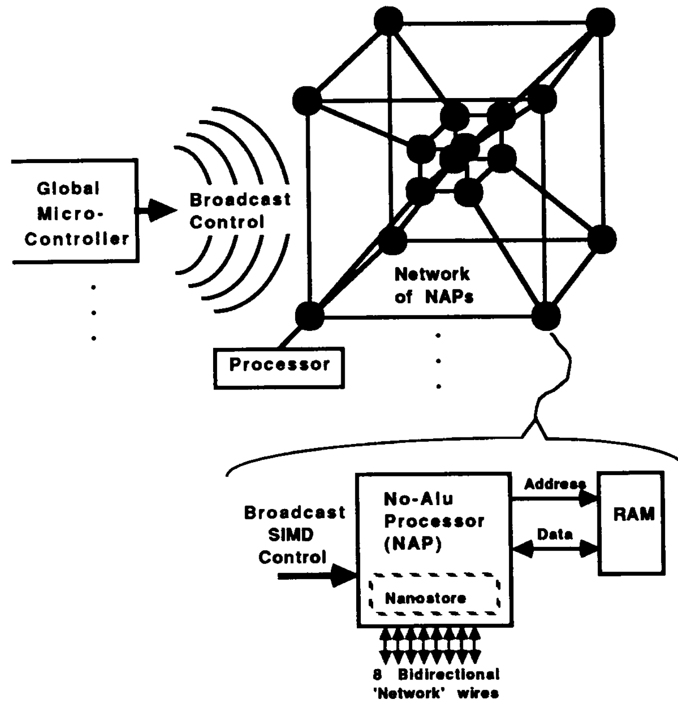

Figure 1. System-level view of a NAP-based computer. Global microcontrollers broadcast instructions to sets of NAP chips. Each NAP chip is connected to an off-chip RAM, a SIMD instruction broadcast bus, and 8 bidirectional network lines.

Therefore, we chose a microcycle/nanocycle timing approach. At each microcycle, the controller broadcasts a global microinstruction, and each processor can read or write from each of its eight pins. Within each microcycle, there are four minor cycles called nanocycles. During each nanocycle, a nanoinstruction is executed which nearly always references the external memory twice (one read and one write or write-back). Thus, the NAP uses a two-phase timing methodology internally, and the memory may be accessed during each phase. Two phases make a nanocycle, and four nanocycles make a microcycle. The memory address may be changed once a nanocycle.

The NAP is heavily memory based. As we have seen, each phase of a nanocycle may involve a 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 wire values, an integer, or an address. Each table (called a function table) requires 256 words (8 bits each). Our prototype supports up to 2K words of external RAM, 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.

The NAP microword

Figure 2 shows the format of the NAP microword. This word is the instruction broadcast from a controller 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 microword is also very memory-oriented; 17 of its 39 bits are used for memory addressing.

The microword does not contain the actual nanoin-

Name	Function	Width
INIT	initialization and download control	1 bit
OP0	four-bit indexes into the nanostore	4 bits
OP1	which specify which nanoinstruction	4 bits
OP2	to perform in each nanocycle	4 bits
OP3	OPs share one address and condition code	4 bits
CC	Condition code; this decodes to 16 conditions	4 bits
MIP	Memory address (for direct addressing)	11 bits
F0	Function table offsets (for indirect addressing)	3 bits
F1	normally contains the start address of a table	3 bits
	total number of microword bits	39

Figure 2. The microinstruction word format shows the mnemonics, functions, and width of each instruction field.

structions executed each nanocycle by the NAPs. Rather, it contains four four-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 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.

Memory Addressing Modes

A number of memory addressing modes are supported by the NAP. Bit-read, bit-write, word-read, and word-write modes are supported, and each of these may be addressed using any combination of bits available to the address multiplexors. A memory address is built as shown in Figure 3. Bits are multiplexed onto the SRAM address pins from the microinstruction (the MIP, F0, and F1 fields), or from internal state bits. There are sixteen bits of state in the NAP: eight bits from the external SRAM held in a Memory Data Latch (MDL), and eight bits which can be configured as any arbitrary combination of I/O bits or additional State bits (called IS bits). All of the these state bits may control the memory address.

A memory address specifies an eight-bit word. Within that word, the low order three bits of the MDL specify a bit in that word. A memory address is 11 bits (providing 8K bits of address space) in the NAP chip. Each memory address is used for one nanocycle only, although the memory addressing fields are held constant for a whole microcycle.

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 I/O/State bits by ORing the external wires together. This capability is extremely useful. For example, when checking for a condition (e.g. does any processor contain zero, or does any processor's memory contain a pattern which matches the broadcast pattern), the result can be returned to the microcontroller within a microcycle. The distance from the microcontroller to the NAP chips through the SIMD bus and back through the global-or tree might be more than 200 ns, so that programmers using the global-or mechanism might have to take account of the pipelining effect. Any bidirectional communications pin on the NAP may be used to construct a wired-OR tree.

PROCESSOR DESIGN

A block diagram of the NAP processor is shown in Figure 4. The major subcircuits are a set of datapath circuitry, a Nanocode store consisting of 16 by 28 bits of static RAM, Nanosequencer logic to control the execution of instructions, and a set of Instruction pipeline registers. The NAP uses a three-phase (1.5 nanocycle) pipeline internally: operation lookup, nanocode access, and datapath operations happen sequentially in every nanoinstruction.

The NAP is designed to work with 35-ns external Static RAMs. These are expensive. It would make sense to move this memory on-chip.

Sixteen words by 28 bits of nanocode store are provided which are addressed in sequence by each of four opcode fields in the microinstruction. These nanoinstructions are downloadable and may differ for different processors. The outputs of the nanostore are the con-

Address bits	bit source 0	bit source 1	bit source 2
Bits 0:4	MIP[0:4]	MDL[3:7]	IS[0:4]
Bits 5:7	MIP[5:7]	IS[5:7]	
Bits 8:10	MIP[8:10]	F0[0:2]	F1[0:2]

Figure 3. The memory address is constructed from combinations of the Memory Immediate Pointer (MIP), the Memory Data Latch (MDL), the I/O-State bits (IS), and the Function table pointers (F0 and F1).

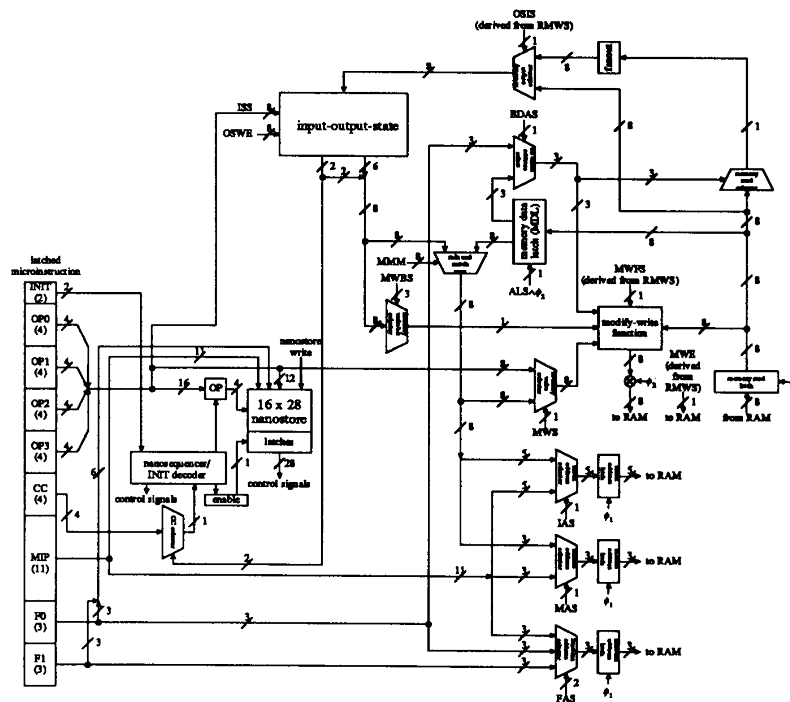

Figure 4. The block diagram of the NAP chip shows the SIMD instruction latches (left), the nanosequencer (lower left), the state and I/O circuitry (top center), the data-path (right), and the RAM interface (lower right).

trol bits used directly by the logic in the processor; the nanostore itself is a static RAM with decoders, write amps, and sense amps. This RAM has a access time goal of 25 ns, and is 1974 by 1620 microns in area using a 3 micron CMOS technology.

Conditional instruction execution is provided in the nanosequencer via an enable control which may disable all outputs of the nanostore. This disabling happens if the bit in the microinstruction condition code field selected by the state bits is high. This mechanism allows up to 16 different classes of processors.

The NAP is designed using a fully static CMOS circuit methodology in MOSIS scalable CMOS design rules. A two phase non-overlapping clocking approach is used; Approximately half of the circuitry on the chip (and exactly half of the control lines) are '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 microns in a 3 micron CMOS process.

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 towards 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 nanoinstruction levels. Processors have considerable flexibility in addressing modes, and indirect addressing at both the bit and 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 controller.

- **Keep the I/O pins 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 I/O pins on the processor. All of the programs written on NAP to date have been able to keep the I/O pins active at at least one 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 (Ref. 6) using the same number of cycles as a node designed specifically for that purpose. We have also written NAP programs for parallel prefix (Ref. 1) 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 I/O bits. 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.

We hope that the NAP chip will eventually serve as a testbed 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, using these mechanism to write programs may lead to future insights about what programming constructs are useful for writing effective parallel programs for communication networks.

ACKNOWLEDGEMENTS

The authors would like to 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. A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. In *Proceedings of the 14th Annual ACM Symposium on the Theory of Computing*, pages 338–344, 1982.

2. Ronald I. Greenberg and Charles E. Leiserson. Randomized routing on fat-trees. In *Proceedings of the 26th Annual IEEE Symposium on the Foundations of Computer Science*, November 1985.

3. W. Daniel. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

4. Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10), October 1985.

5. Nicholas Pippenger. Parallel communication with limited buffers. In *Proceedings of the 25th Annual IEEE Symposium on the Foundations of Computer Science*, October 1984.

6. Abhiram G. Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual IEEE Symposium on the Foundations of Computer Science*, pages 185–194, October 1987.

7. Tommaso Toffoli and Norman Margolus. *Cellular Autonoma Machines.* MIT Press, Cambridge, MA, 1987.

8. L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on the Theory of Computing*, May 1981.