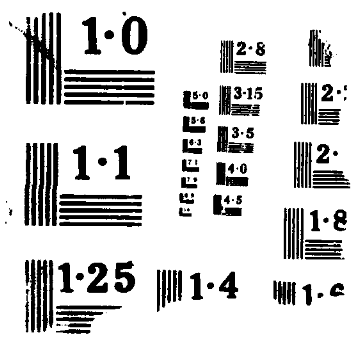
AD-A195 477 ARCHITECTURE FOR OPTICAL DIGITAL COMPUTERS(U) RUTGERS - THE STATE UNIV NEW BRUNSWICK NJ LAB FOR COMPUTER SCIENCE RESEARCH S Y LEVY ET AL 31 AUG 87 1/1

UNCLASSIFIED AFOSR-TR-88-0531 AFOSR-86-0294 F/G 20/6 NL


1·0
2·8
3·15
5·0
5·6
6·3
3·5
4·0
4·5
1·1
2·
1·8
1·25
1·4

DTIC FILE COPY

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE — Form Approved OMB No. 0704-0188

AD-A195 477

(1)

DTIC SELECTE MAY 2 0 1988 S D

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 88-0531
6a. NAME OF PERFORMING ORGANIZATION Rutgers State Univ / 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION AFOSR/NE
6c. ADDRESS (City, State, and ZIP Code) Lab for Computer Sci Rsch New Brunswick, NJ 08903	7b. ADDRESS (City, State, and ZIP Code) Bldg 410 Bolling AFB, DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR / 8b. OFFICE SYMBOL (If applicable) NE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AFOSR-86-0294
8c. ADDRESS (City, State, and ZIP Code) Bldg 410 Bolling AFB, DC 203332-6448	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2305; TASK NO. B1; WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
ARCHITECTURES FOR OPTICAL COMPUTING

12. PERSONAL AUTHOR(S)
Professor Levy

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Annual	FROM 01Sep86 TO 31Aug87		

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We are investigating a Content Addressable Memory architecture for a digital optical computer. This investigation is being funded by AFOSR, and is being carried out in conjunction with work on digital optical computing devices at Bell Laboratories in Holmdel, NJ. Because of the massive parallelism inhernet in the CAM organization it is ideal candidate to constructed of the types of optical components being investaged at the Labs.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL DR GILES	22b. TELEPHONE (Include Area Code) 202-767-4933 / 22c. OFFICE SYMBOL NE

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

Architecture for Optical Digital Computers

AFOSR-TR- 88-0531

Annual ~~Research Progress and Forecast~~ Report

Laboratory for Computer Science Research
Rutgers University
Saul Y. Levy
J. Storrs Hall
Miles J. Murdocca

Abstract

We are investigating a Content Addressable Memory architecture for a digital optical computer. This investigation is being funded by AFOSR, and is being carried out in conjunction with work on digital optical computing devices at Bell Laboratories in Holmdel, NJ. Because of the massive parallelism inherent in the CAM organization it is an ideal candidate to be constructed of the types of optical components being investigated at the Labs.

Overview

Optical computing devices, while holding great promise for major advances in computing power and speed, are currently still laboratory prototypes at best. The Labs are investigating at least three qualitatively different kinds of devices whose properties have considerably varying implications at the low levels of architecture.

Devices and Logic

Huang's group at the Labs is investigating SEEDs, etalons, and QWEST-effect devices. Of these, the SEEDs are being built into small circuits but run at electronic speeds; the etalons run fast but singly, as cascadability is still around the corner; and QWEST-effect devices are as yet unbuilt.

We have been focusing most of our efforts on architectures based on the etalons. However, even the etalons have varying implications for the architecture. For example, the currently projected cascadability technique for etalons is a double frequency scheme where complementary pairs of etalons are used. Under this scheme each closed circuit in the logic diagram must consist of an even number of gates.

Under changing constraints such as these, our strategy has been to adopt a top-down and generalized design process. We have designed a virtual machine (see Virtual Machine Model attachment) as a starting point. We do architecture/circuit designs to the dictates of this model, and independently do software and algorithms research based on the model. The former is to prove the model is possible and feasible; the latter is to prove it useful and generally applicable.

CAM Architecture

The CAM model is similar to a conventional machine. In its essence it is simply a processor and a memory. However, where in a conventional machine the processor specifies loading and storing in memory by a single address at a time, the CAM processor can specify conditions, ranges, and collective functions.

The CAM memory consists of a set of words which are addressed either conventionally or associatively. The conventional addressing is extended to allow specifying ranges. Where in a conventional machine one can specify, "***word 4256***", in the CAM we can specify "***words 4201 through 5557 inclusive***", or, addressing the words by their contents: "***all words containing a number greater than 17***".

If we can specify a selected subset of the words in memory at once, we can do something to them at once. For example, we can set them all to a new value (it must be the same value), or add something to each one (it must be the same something). We can also discover the address (instantly) of the first word which we have specified in some way, ie, "***give the address of the first word whose contents are greater than 17 and less than 39.***"

Furthermore we can divide the memory into sections and use a different value in each section: "Add 20 to all words in section 1, 52 to all words in section 2, etc." The sizes of the sections are restricted; they must correspond with the tree structure (see below). Furthermore, the value to be used with each section must come from a word in that section.

Optical Implementation

We have been doing simulations at the bit level of various concepts of optical machine organization. These are not part of an overall design process but merely to test particular ideas. They are not realizeable with current devices, since the power dissipation problem prevents us using a whole plane of the devices simultaneously.

The Connection Machine attachment gives a sample of the lower-level design strategies we have been pursuing. We have done several such designs, though this is by far the most ambitious. This design is actually the CAM design. The paper is aimed at the Connection Machine architecture in order to display the generality of the design techniques; however, the processing elements and memory are the major parts of the CAM and the "as yet unimplemented" routers are not part of the CAM.

CAM Software and Simulation

CAML is a lower-level programming language whose primitives match those of the CAM virtual machine (see the CAML attachment). CAML is our major vehicle for proving the usefulness of our architecture. actually a subset of this design. Again, we must stress that these designs are subject to constant change. We are developing a CAML compiler/CAM simulator for testing and demonstrating algorithms on the architecture. This is a high-level simulator in that it models the abstract operations of the virtual machine. We have written the preliminary version of the compiler front end and are currently engaged on the simulator.

The CAML compiler is a non-trivial exercise in state-of-the-art compiler design. The processor allocation and similar tasks it performs, form a substantial factor in the ease of parallel programming CAML provides. The compiler is constructed in Common Lisp.

Algorithms and Applications

To prove the general usefulness of the CAM, we are investigating the set of algorithms that underlie most common computing tasks. Our results to date are very encouraging: the CAM shows applicability to a wide range of fields.

In this area we have been doing higher level algorithm design and some CAML coding. Early this summer the simulator will be in a state where we can test CAML codings. After this we will begin to put these algorithms into code.

- Although the CAM is not intended as a numeric engine, it is necessary that its numeric performance be good if it is to be used as a general-purpose machine. Thus we have looked at several typical numeric tasks and find promising results in areas like graphics, where operations like multiplying many small vectors by the same small matrix, occur frequently.

- The search and inferencing operations of AI/expert systems are well matched to the strengths of the CAM. Indeed, most operations on a knowlege base can be facilitated considerably, because we can use simple associative data structures which support fast accesses and operations of all kinds, rather than having to use complex and restrictive indexing schemes to speed one particular operation.
  - The case- or example-based expert systems (Hillis & Steele) or statistics-based ones (Wexelblat) can give useful results where a body of knowlege is available but no structured theory of the problem is known. They require enormous computational power, but are highly parallelizeable. This makes them better suited to the CAM than a conventional architecture.
  - The distributed hash encoding methods found in Knuth Vol 3 Sec 6.5, based on single-bit manipulations, are particularly suited for CAM implementation. (These encodings are the mathematical link between conventional "frame-based" AI and some of the neural net work such as that of Touretzky.)
- We have produced good results in non-numeric programming fields such as graph theory, network optimization and computational geometry. The set operations basic to these algorithms are near-primitives on the CAM. We will produce algorithms for:
  - Min, max, member, insert, delete, intersection, union, find.
  - Spanning trees, shortest path, connectivity, biconnectivity, strong connectivity, planarity, partitioning, depth and breadth-first search, and transitive closure.
  - Convex hull, 3-D convex hull, Delaunay triangulation, line and curve fitting.
  - Sorting and searching (although sorting is usually rendered unnecessary and searching is usually a simple primitive operation).
  - Means, medians, modes, histogramming, generation of permutations, binomial coefficients, random numbers, and so forth.

References

Hillis, D. and G. L. Steele, *Data Parallel Algorithms*, Communications of the ACM, December 1986

Wexelblat, R. (private communications) Wexelblat implemented a proprietary (ITT-ATC) object recognition system which had teachability and partial pattern recognition capability similar to a neural network model but was based on the collection and recombination of well-defined statistical parameters of the domain.

Touretzky, D. and G Hinton, *Symbols Among the Neurons*, International Joint Conference on Artificial Intelligence 1985

The Virtual Content Addressable Machine Model

J. Storrs Hall
Laboratory for Computer Science Research
Rutgers University

Introduction

The CAM model is an architecture intended to bridge the gap between optical and conventional electronic computers.

It derives in large part from the work of Caxton Foster.

It is a SIMD architecture with a very regular interconnection scheme.

Overall Structure

The CAM consists of three main parts: cpu, ram, and cam. The cpu and ram form a conventional Von Neumann computer. In operation as a CAM, however, the ram is largely used to hold scalar data and programs.

The cam (content addressable memory) consists of a large number (in prototypes, thousands; eventually, millions) of small processing elements. Each element is called a *cam cell.*

Each cam cell consists of some bits and the mechanism to perform simple operations on them (a one-bit ALU). The number of bits per cell depends on the technology and on the number of cells it is feasible to implement. If there are relatively few cells (as in the prototype) we try to provide as many bits as possible, since the cam cells may have to be time shared to represent a larger virtual cam. In this case the data for all the virtual cells implemented on a single physical cell, would be stored on that cell.

The cells operate in strict lockstep under broadcast instructions from the CPU. Thus they are capable as acting as an array processor or doing associative matching. There is the provision for doing communication between the processors; it is simple and regular due to several constraints, and amounts to doing shifts up and down the cells arranged in a single one-dimensional vector.

There is provision for preforming collective functions on the cam data, ie, counting or summing items from all the cam cells. This is implemented as a tree in most designs.

Functionality of the Cam

Single Cell Operations

Each cell can perform a variety of bit operations between bits in its memory. These are sufficient to allow bit-serial arithmetic (even floating-point) between numbers. This arithmetic requires time proportional to the number of bits involved, but since usually numbers are short (32 bits or less) and of fixed width, arithmetic is considered a constant time operation.

Broadcast

The CPU can broadcast values which each cell may store or use as an operand.

Selection

As a result of comparisons, bit manipulations, or any other operation giving rise to a boolean value, each cell may ***respond***, setting a flag bit.

Check / Count

The CPU may discover if, in a given operation, any cell has responded and if so how many did so.

Activity

There is a flag bit in each cell which can be used to "turn on or off" the cell causing it to obey or not to obey the broadcast instructions.

The active bit may or may not be physically the same as the response bit. given the ability to move values between the bit(s) and the cell's memory, the functionalities are equivalent. There may be no special bit at all, but a dynamic choice of active or response use from among a cells bits.

Address Ranges

As part of any instruction, the CPU may specify a range of addresses (upper and lower bound) in which the operation is to take place. Only active cells inside the range take part.

Address / Max / Min / Sum

The CPU may obtain in a single step the address of the first responder, the maximum or minimum value of all active cells, or the sum of all active cells. The partial sum, min, or max (ie that of all cells with a lower address) may be formed in all active cells in one step.

The address of each cell may be formed therein for all cells in one step.

In the electronic cam there is a treeshaped connection between the cells which forms them into a systolic array for performing these functions. In the optical architecture the tree connections are present as a subset of the standard connectivity for the machine.

Shifting

In one step, data from all cells may be shifted in parallel some distance up or down the array. In the optical architecture this is a simple consequence of the connectivity; in the electronic version it is more problematical and is implemented as a series of shift registers, kludges, and barrel shifters. The time to shift is independent of the distance moved and the number of cells participating, but depends on the amount of data moved from each cell. Again, since this is small and fixed, it is usually considered constant.

Local Broadcast

Local broadcasting means dividing the cam into a set of contiguous substrings of cells, and then taking a value from the first cell of each substring and sending it to the other cells in the substring, all at the same time. Local broadcasting can be done on the cam if the length of each substring is the same, and is a power of two, and all the substrings are aligned to addresses that are multiples of the length.

Formation of running sums, maxes, and mins can also be done in substrings of the same constraints. For broadcasting, the value broadcast does not have to be from the first cell, but can be from any cell, say the ith cell, where i is the same for each substring; or it can be from any active cell, assuming each substring has only one active cell.

DESIGN TECHNIQUES FOR AN OPTICAL CONNECTION MACHINE

J. S. Hall
S. Levy
M. J. Murdocca

Rutgers University, New Brunswick, New Jersey

ABSTRACT

The Connection Machine (CM) is an advanced parallel processing computer designed and built by Thinking Machines Corporation. The architecture is noted for high connectivity between a large number of small processors. Free-space optics can provide a large number of regular connections such as perfect shuffles and banyans, which we explore in an optical design of the CM. Recent work on optical connections provides design techniques for using regular free-space interconnects for small logic units and for random access memory. We show designs for the memories, arithmetic logic units, flag registers, routers, and hypercube using free-space interconnects. The final design consists of optically nonlinear arrays of logic devices interconnected in free space with simple passive components such as beam-splitters, lenses, and mirrors. The design presented in this paper demonstrates that regular free space interconnects are suitable for use in the design of an optical computer composed of densely connected simple circuits, without incurring significant costs in circuit depth or component count.

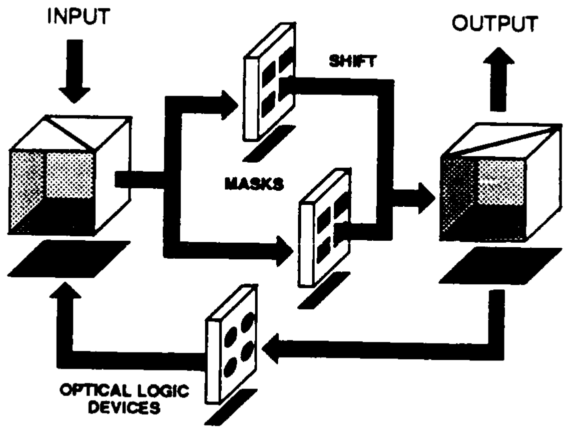


Fig. 1. *Alpha* module. A two-dimensional input image is split into two identical images that are each masked, combined, and regenerated. The array of logic devices performs a simple Boolean function such as logical NOR, and regenerates the signals.

1. BACKGROUND

A number of optical logic devices have been proposed in recent years for applications in optical computing [1-4]. From the view of a computer designer, there is a fundamental difference between these types of devices and conventional electronic devices. That difference is that the logic gates are oriented normal to the surface of the substrate, with no interaction between logic gates on the chip. Components are interconnected in free space, using optical means such as mirrors, lenses, gratings, beam-splitters, and masks to achieve a desired interconnection topology. Fig. 1 shows how an array of optical logic devices can be used in an optical computing system.

A two-dimensional image is passed through a beam-splitter where it is split into two identical images. Each image is passed through a two-dimensional mask, where selected bits are forced off. The images are shifted with respect to each other and are combined onto an array of optical logic devices. An output can be produced or the system can be wrapped back onto itself. The array of optical logic devices regenerates the signals so that infinite cascadability can be achieved. The goal of this setup is to connect the output of every logic gate with the ouput of each gate's neighbor a fixed distance away, except for connections that are masked out. The logic operation performed by the array of optical logic devices is assumed to be the same over the entire array, such as 2-input 2-output NOR gates in order to place the fewest requirements on the logic devices, which are currently the most critical components in the realization of an optical computer.

There is a 3-level hierarchy of optical setups that we consider here. There are *alpha* modules as shown in Fig. 1, *beta* modules, and *gamma* modules. The *alpha* module implements a simple split-shift-combine operation among the elements of an array, producing interconection patterns as shown in Fig. 2, where two *alpha* modules are arranged in tandem.

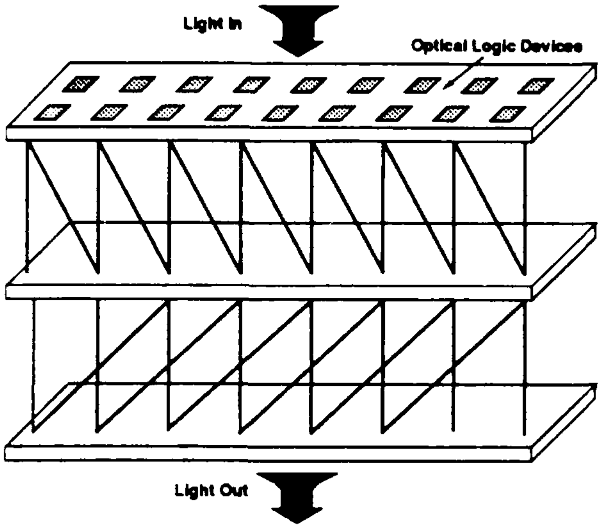


Fig. 2. Interconnection patterns produced by *alpha* modules arranged in tandem.

Alpha modules can be arranged in tandem with a different shift applied in each stage as shown in Fig. 2, so that greater flexibility in interconnection topology is achieved as circuits grow deeper.

It is reasonable to expect the size of optical logic arrays to reach 256 x 256 = 65536 elements, but not much larger due to a limitation on the amount of information that can be passed through a practical lens, commonly referred to as ***space-bandwidth product***. The Connection Machine [5] is a computer that is made up of over a billion switching elements, so we can expect that a single 65536 element array in an *alpha* module will not be sufficient to implement such a computer effectively. For this reason we propose a second level of hierarchy containing ***beta*** modules, which are composed of a number of ***alpha*** modules arranged in tandem as shown in Fig. 3.

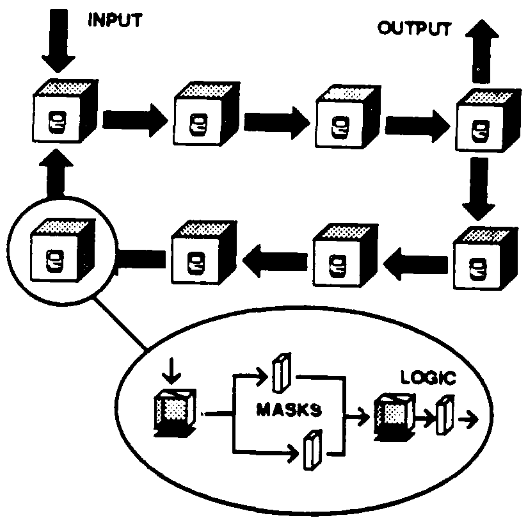


Fig. 3. *Beta* module. A two-dimensional input image is passed through a number of ***alpha*** modules before an output is produced. In this example, the output is looped back to the input. The amount of shift produced in each *alpha* stage can be different, which allows for greater flexibility in the interconnection topology.

The use of ***beta*** modules improves the amount of information that can be processed in a system by providing alternative interconnection schemes, but a large computer cannot be made into a single narrow pipeline without incurring an intolerable latency. For this reason, we propose a higher level in the hierarchy, a *gamma* module which is composed of a number of *beta* modules in parallel. A *gamma* module is shown in Fig. 4.

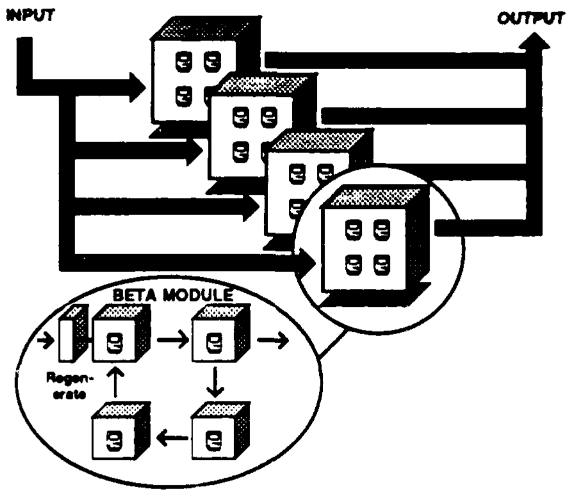


Fig. 4. *Gamma* module. A two-dimensional input image is split into four identical copies that are regenerated and passed to *beta* modules where they are processed. Outputs are produced in parallel and are collected.

The input is split and regenerated as appropriate before being passed to *beta* modules, where the data is processed and output is produced. The outputs are collected and regenerated as needed. We use these three levels of hierarchy in designing the architecture discussed in the remainder of this paper.

The next section describes the Connection Machine (CM) and identifies the significant parts that must be redesigned for free-space interconnects. Section 3 details the designs for each of these components. Section 4 describes the design of the hypercube interconnect. Section 5 discusses pipelining at the gate level. Finally, we conclude with the remark that the techniques used here in the design of an optical CM provide sufficient connection power and flexibility despite the strict regularity of the interconnects.

2. THE CONNECTION MACHINE

The Connection Machine [5] is a massively parallel computer architecture consisting of a large number of 1-bit processors arranged at the vertices of an n-space hypercube. Each processor communicates with other processors via *routers* that send and receive messages along each dimension of the hypercube. A block diagram of the CM as described in [5] is shown in Fig. 5.

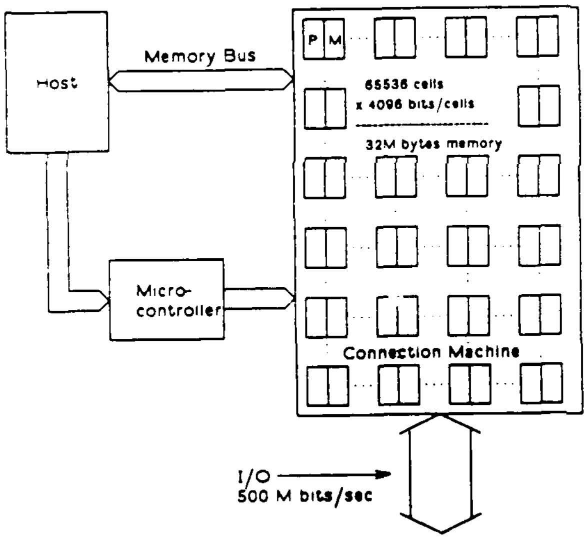


Fig. 5. Block diagram of the CM-1 prototype Connection Machine.

The host computer is a conventional von Neumann machine such as a VAX or a SUN computer that runs a program written in a high level language such as LISP or C. Parallelizeable parts of a high level program are farmed out to 2^n processors (2^{16} processors is the size of the CM-1) via a memory bus (for data) and a microcontroller (for instructions) and the results are collected up via the memory bus. A separate high bandwidth datapath is provided for input and output (I/O) directly to and from the hypercube.

The CM makes use of a regular interconnect (an n-space hypercube) between the processing elements (PE's). A 4-space hypercube is shown in Fig. 6.

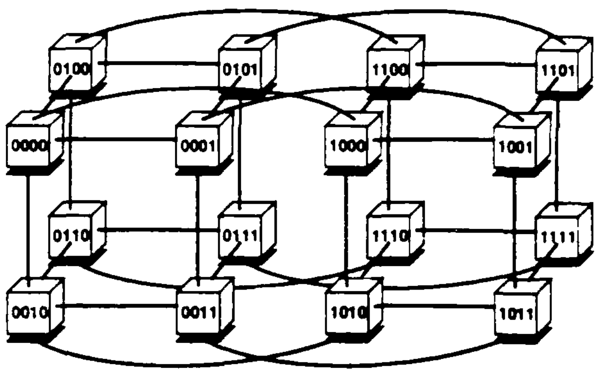


Fig. 6. 4-space hypercube. Each vertex of the hypercube is a PE that has a unique binary address. PE's that are directly connected to other PE's can be found by inverting any one of the 4 bits in the address.

Free-space optics is good at providing regular connections via perfect shuffles [6] and banyans [7] which can both be used to implement the hypercube (see Section 4), so that we may consider optical means for implementing the interprocessor interconnect. Within each PE regular interconnects can be used as well, using techniques described in [7-9] which are repeated here for reference. The use of regular interconnects for the PE's is particularly interesting because it allows pipelining at the gate level, which increases throughput and decreases the size of the machine by sharing logic and memory among PE's. On the down side, gate-level pipelining increases latency and introduces the problem of synchronizing PE's that are in different phases of operation, and we address these issues in Section 5. Gate-level pipelining is made possible by pulsed optical power which embeds a clock into the power supply, and by reducing signal skew with optical connections that are all of virtually equal length [10].

As described in the previous section, an *alpha* module is the most primitive optical setup we consider. When the shifted arrays of an *alpha* module are superimposed, a large number of connections are made, with one connection made per element per array. The use of a magnification step allows the superimposed images to form a perfect shuffle pattern as described in [6], or a simple banyan network can be implemented by varying the amount of shifts in each *alpha* stage. A ***beta*** module making use of banyan connected logic gates is shown in Fig. 7.

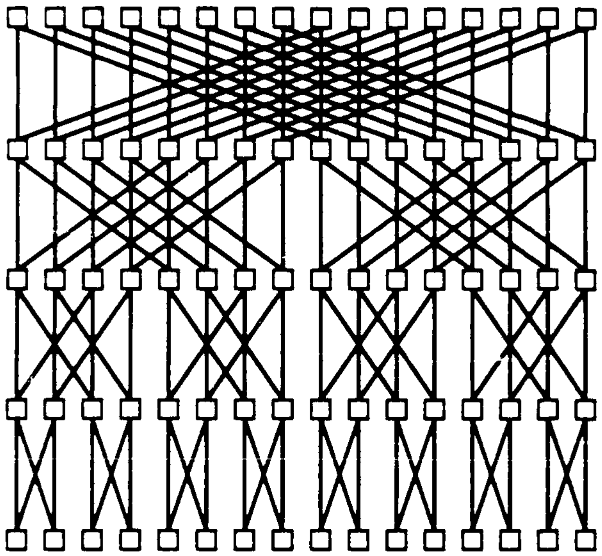


Fig. 7. Banyan connected optical logic gates.

The key to effectively using this regular interconnect to implement arbitrary functions lies in masking out unwanted connections and positioning inputs and outputs appropriately. Algorithms for doing this are presented in [7-9], and are repeated here in small detail to show how the design techniques extend to the the various parts of the CM architecture. We make no attempt to design the host computer or the microcontroller since they contain a fair number of embedded state machines, which the design techniques used here are not easily equipped to handle.

Significant components in the CM that must be redesigned for free-space optical computing include the hypercube, the PE's, and the routers. The PE is further broken down into one 16-bit flag register, a 3-input 2-output ALU, and a 4096 bit random access memory, as shown in Fig. 8.

This model differs from the original design of a CM PE by dedicating one router to one PE, as opposed to one router serving 16 PE's as in the original design. This tradeoff was made in order to simplify the design of the router so that the techniques we use here apply directly. More sophisticated design techniques that handle embedded state machines are needed in order to implement the original design of the routers effectively.

Designs of these components using free-space design techniques are described in the following sections.

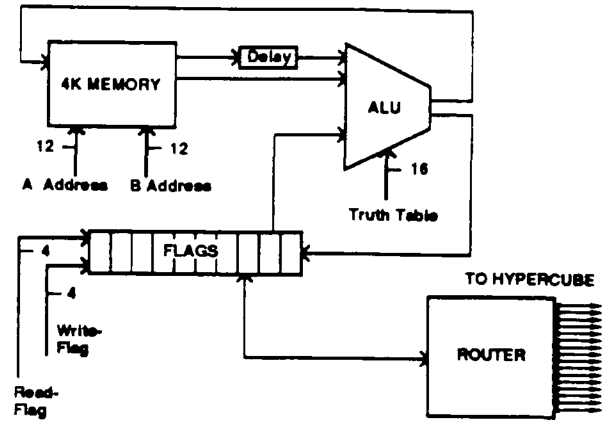


Fig. 8. Block diagram of a single Connection Machine processing element, as described in [5].

3. THE PROCESSING ELEMENTS AND ROUTERS

In the model of a PE shown in Fig. 8, an external controller (the microcontroller shown in Fig. 5) selects 2 bits of memory via the A address and B address lines. Only one value can be read from memory at a time, so the A value is buffered (delayed, here) while the B value is fetched. The controller selects a flag to read, and feeds the flag and the A and B values into an algorithmic logic unit (ALU) whose function it also selects. The result of the computation produces a new value for the A addressed location and one of the flags. The ALU can be implemented with a simple combinational logic unit as described in the next section. Memory is an interesting case, and is described in detail in the section that follows. The Flag register is a special case of memory since it needs address decoding as well, so it is described as a small example of a random access memory. The router is discussed in Section 3.3.

3.1. The ALU

The ALU takes three 1-bit data inputs, two from the memory and one from the flag register, and 16 control inputs from the microcontroller and produces two 1-bit data outputs for the memory and flag registers. The ALU generates all $2^3 = 8$ combinations (minterms) of the input variables for each of the 2 outputs. 8 of the 16 control lines turn off the minterms that are not needed in the sum-of-products form of each output. The technique described in [7] generates a near-optimal depth circuit for a programmable logic array (PLA) that implements the ALU, as shown in Fig. 9.

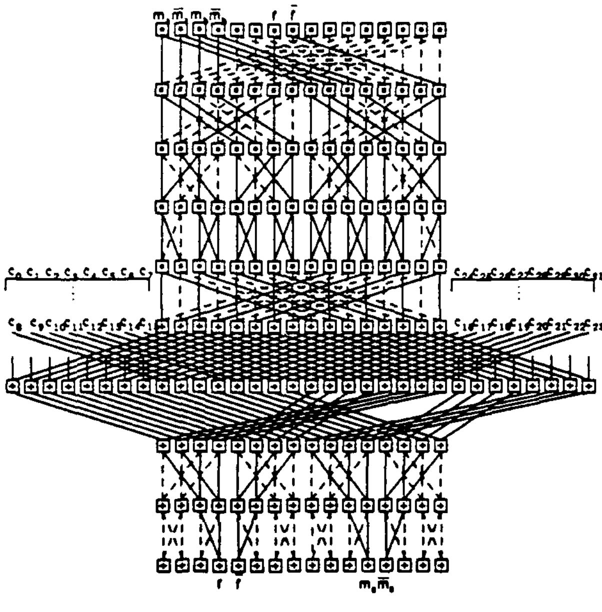


Fig. 9. Externally controlled ALU using the PLA design technique. Implementation is shown for dual-rail logic using AND gates (boxes with inscribed squares) and OR gates (boxes with inscribed crosses). Dashed lines indicate masked connections.

The PLA design technique works by generating all 2^n minterms of n variables in $n+1$ levels of shuffle connected (or banyan connected) AND logic gates, followed by generating functions in $n+1$ levels of shuffle connected (or banyan connected) OR logic gates. In Fig. 9, all combinations of the inputs m_a, m_b, and f are formed after the fourth level of connections (on the fifth row of logic gates from the top). In order to generate a function, the next step is to logically OR the minterms into functions via another $n+1$ level deep banyan structure. Here, we violate that method by adding a fifth level of twice the width which contains a non-standard interconnect. We do this in order to share the minterm generation logic over both output functions f and m_a.

The AND and OR gates can be replaced with all NOR gates or all NAND gates as the implementation may require as long as the connections are changed to correspond to the new logic. Only 3 angles of connections are needed per level including the control level where the external control lines enable and disable selected combinations. The regularity of the interconnect reduces signal skew, and the pulsed optical power supply used for logic gates such as SEED's [1] and OLEs [2] guarantee that any signal skews will not accumulate for more than one level, so that each level of logic can work on a different problem without interference from the other levels. Thus, even though the ALU shown here is made up of 176 two-input, two-output AND and OR gates in 10 levels, it is responsible for implementing 10 ALU's each in a different phase of operation. Synchronization among phases is readily handled and is discussed in Section 5. The actual cost per ALU then is $176/10 = 17.6$ logic gates. This number decreases as fanin and fanout become greater than 2. It is reasonable to average the hardware over the total number of operations being performed, at least for the CM, because the CM is made up of a large number of identical elements that would have to be realized in one form or another regardless of pipelining. In a $64K$ node design of a CM, only $64K/10$ nodes need actually be constructed.

3.2. The Memory and Flag Register

A computer memory is called ***random access*** if any word of the memory can be accessed in an equal amount of time, independent of the position of the word in the memory. Usually the time is logarithmic in the size of the memory. That is, if a random access memory (RAM) contains N words, then any element of the memory can be accessed in $C\lceil log_f N\rceil$ time, where f is the fanout (here, we assume a fanout of 2) and C is some constant. For a RAM of size N, $M = \lceil log_2 N\rceil$ address bits are needed to uniquely identify each word. The address bits are fed to the address decoder of the RAM which selects a word for reading or writing via an M level deep decoder tree. Read and Write control lines determine whether the operation is to read or write at the addressed location, and data lines provide a means for transferring a word to and from the memory.

Optical implementations of the decoding function and the storage function must both be implemented in order to realize an all optical RAM. The decoding function of a RAM can be satisfied with the top half of a PLA structure [7] as shown in Fig. 10. The bottom of the decoder tree is 32 bits wide in this example and there are 16 unique addresses, so 16 two-bit words of memory can be accessed in five levels of logic. If we allow both sides of the decoder tree to be traversed simultaneously, by forcing an address line to be active as well as its complement, then the

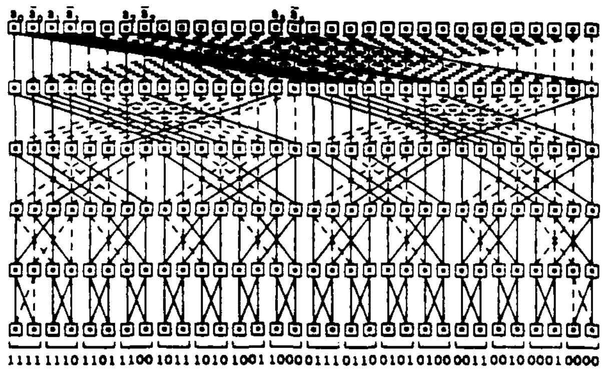


Fig. 10. Decoder tree for 16 word, 2-bit random access memory using the AND matrix of a PLA structure. All boxes are 2-input / 2-output AND gates. Address inputs a_i are at the top and addressed words are at the bottom.

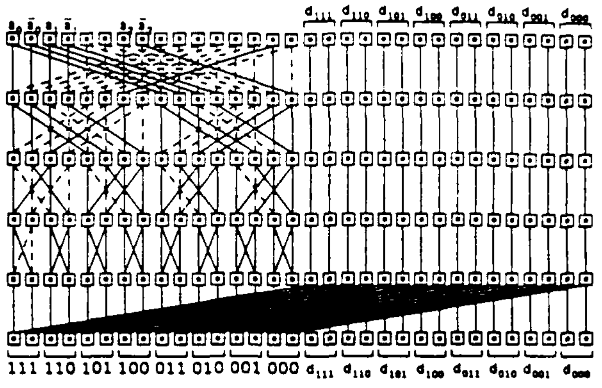


Fig. 11. 8-word decoder (left half) and 8-word memory (right half). All boxes are 2-input / 2-output AND gates. a_i are address bits and d_j are data bits.

size of the accessed word doubles and the number of stored words halves. This can be continued until the word size equals the size of the memory and the entire memory is read out in parallel. We can make use of this property here to share one memory structure among a number of PE's, but a better approach is to make use of gate-level pipelining as described in Section 3.1. A large parallel readout is advantageous for applications where all PE's access the same locations in the same sequence. An 8-word, 2-bit memory will be used for the remainder of this section for space considerations, but extensions to larger memories should be obvious.

The stored words of the memory travel alongside the decoder tree as shown in Fig. 11.

A level has been added to the bottom of the memory so that the uniquely defined address pair from the decoder tree is ANDed with the stored words of the memory, resulting in the selection of a single word. We are allowed to deviate from the traditional $log_2 N$ interconnects such as the perfect shuffle and banyan and use a specialized regular interconnect in the last stage of Fig. 11, as long as the interconnect can be implemented with simple split and shift operations, which is the case here. The result of the AND operation is funneled through a $log_2 N$ fanin structure to a fixed output location as shown in Fig. 12.

Note again that the stored words of the memory travel alongside the memory collection tree, because every signal moves on every time step, and we must account for the state of the memory at every stage. The memory collection tree is necessary only if we are selecting

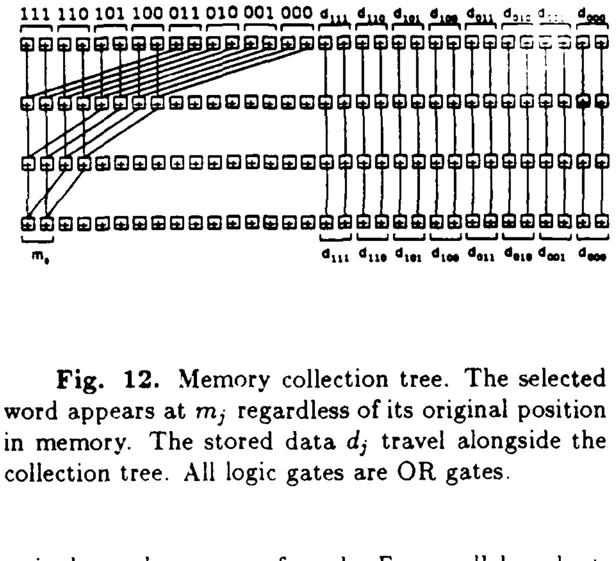


Fig. 12. Memory collection tree. The selected word appears at m_j regardless of its original position in memory. The stored data d_j travel alongside the collection tree. All logic gates are OR gates.

a single word or group of words. For parallel readout, there is no need for the collection tree.

Writing into memory can be accomplished in a similar fashion, by using the decoder tree to enable the position to be written and then writing the element at that location. Since the location to be written is not known *a priori*, a $log_2 N$ expansion tree can be used to fan the word to all memory locations before the location to be written is enabled.

The cost of implementing the memory in terms of gate count is slightly more than one switching component per stored bit of information, and the latency through the memory is $2log_2 N + 2$ gate delays for an N bit memory. This cost is arrived at by using gate-level pipelining and free-space optical delays for storage. For example, consider one row of logic gates in Fig. 11. For an N bit memory, the width of the address decoding portion is N gates and the width of the stored words of the memory is N gates. If the memory is pipelined at the gate level, then each stage of logic services a different memory and address request, so that the cost of implementing a single memory is $2N$ gates for the Read logic. When the stored words of the memory move from one stage to the next, no logic function or regeneration is needed except at the bottom of the address decoder where the selected word is extracted. Thus, a simple optical delay will keep the stored words of the memory intact while the address is being decoded, giving a final cost for the memory of between one and two switching components per stored bit of information. The number varies depending on how large the memory is. The cost of the stage where the selected word is extracted becomes smaller for large memories.

A similar analysis can be used for the Write logic, which would bring the total cost of the memory to less than 2 switching components per stored bit of information, and no more than $2N$ gate delays for an N bit memory.

The Flag register is a small example of a RAM. Address decoding is needed in the CM flag register to isolate a single bit, and the method of address decoding just described will suffice for this purpose. Reading and writing the register is the same as for the RAM. The major difference between the Flag register and the RAM is size: the Flag register will be only 64 gates wide (for 32 dual-rail logic bits) and 10 levels deep.

3.3. The Router

PE's communicate with other PE's through routers. Each router services communication between a PE and the network (for simplicity, we assume there is one router for one PE, unlike the CM-1 design where one router services 16 PE's) by receiving packets from the network intended for the attached PE, injecting packets into the network, buffering when necessary, and forwarding messages that use the router as an intermediary to get to their destinations.

When a packet comes into the router from either the network or from the PE, an available buffer is identified and is used to store the packet as shown in Fig. 13.

Data to be written is fanned out from the Serial data in line to all 4 buffers, and data that is to be read is fanned in from all 4 buffers to the Serial data out line. A separate control box for each buffer

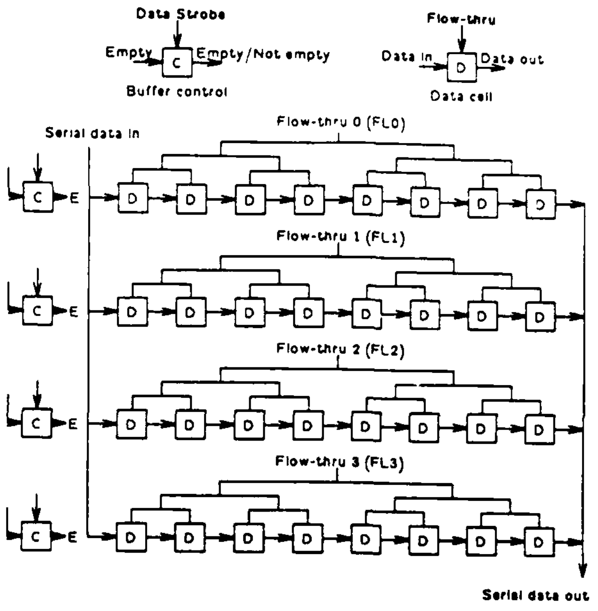


Fig. 13. Serial writing into and reading out of 4 buffers. Control boxes (marked with C's) keep track of whether a buffer (marked with D's) is empty or full. Data enters and leaves the buffer bank in serial fashion.

records the current state of the buffer (EMPTY / NOT EMPTY) and is used to generate the Flow-thru control signals which cause serial data to be written into an empty buffer or read out of a full buffer. The diagram shown in Fig. 14 shows how the appropriate Flow-thru signals for writing into the buffers can be generated with a PLA.

The mechanism for generating the control lines can be implemented with a 5-input 4-output PLA using regular interconnects between stages of identical logic gates as described in [7]. The actual design of the PLA is not detailed here because any of a number of priority schemes can be used, simply by changing the mask patterns of the PLA. The Read control circuitry is similar to the Write circuitry just described except with a different PLA and with a line to the PE indicating there is a packet waiting to be delivered.

4. THE HYPERCUBE

The 16-space hypercube provides 16 connections, one for each dimension, for each PE. In any technology fanin and fanout are limited to a number typically less than 16, so that 16-dimensional connectivity requires more than just one level of logic or regeneration. Here, we assume fanin and fanout are both limited to two

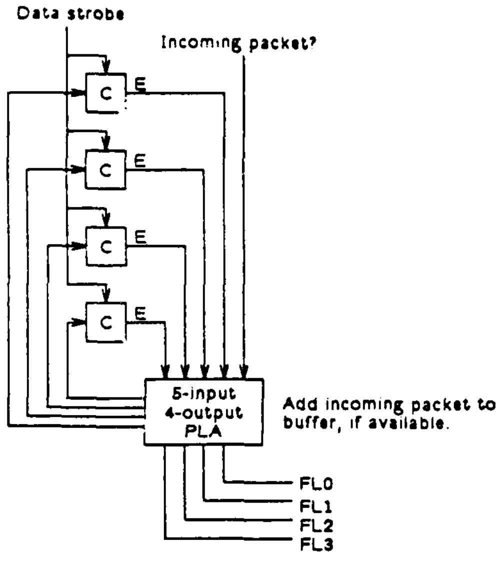


Fig. 14. Control circuitry for finding and enabling an available buffer.

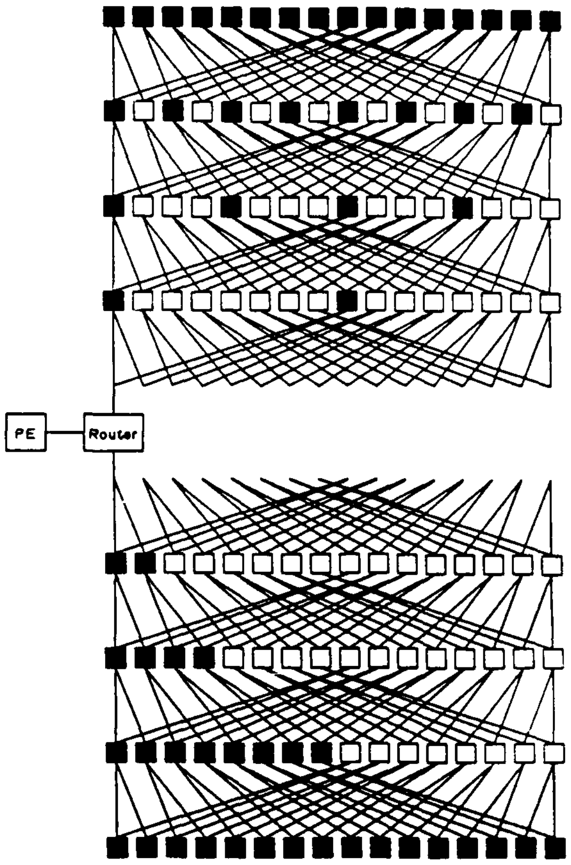


Fig. 15. Perfect shuffle connection pattern at a vertex of a 16-space hypercube. Upper half is the incoming network, bottom half is the outgoing network. Shaded boxes indicate nodes that make up the shuffle equivalent hypercube. There is one input/output box for each dimension of the hypercube.

so that 16-space connectivity is achieved in no less than $\lceil log_2 16 \rceil$ levels regardless of the means used for interconnection. A banyan or perfect shuffle can be used as the $\lceil log_2 16 \rceil$ interconnection pattern for the input and output trees of a 16-space hypercube vertex as shown in Fig. 15, where one leaf node of each tree connects the PE to one of the 16 dimensions of the hypercube.

The upper half of the diagram shows the incoming $\lceil log_2 16 \rceil = 4$ level perfect shuffle equivalent of the 16-space hypercube vertex while the bottom half shows the outgoing shuffle equivalent hypercube vertex. A hypercube interconnect can be realized with perfect shuffles in a comparable component count as would be needed for a direct implementation given the same fanin and fanout constraints, for machines with small dimensionality (≈ 16). We cannot implement the entire CM hypercube with one array, however, so we are forced to break up the hypercube over a number of arrays. In order to break up the hypercube without adding significant cost to the design, we can use the property that a large perfect shuffle (or nearly any other $log_2 N$ interconnect) can be realized with a perfect shuffle of smaller shuffles. Fig. 16 shows a 16 wide shuffle implemented with four 4-wide shuffles that are shuffled together.

The full connectivity of the original shuffle is still realized, but the cost of an additional stage of logic has been added. In general, this cost adds up to one stage of logic for each decomposition of the shuffle. There are on the order of 2^{30} switching components in the CM-1 and about 2^{16} switching components can be expected in an optical array, which means that approximately $2^{30}/2^{16} = 2^{14}$ arrays will be needed. In the worst case, every array will need to be connected to every other, so that $log_2(2^{14}) = 14$ additional levels will be needed across the entire hypercube to compensate for the fact that 2^{30} components cannot be placed on a single array. This cost of 14 gate delays is quite small when compared with the size of the entire machine and when consideration is made that this cost may be the most significant packaging constraint, unlike other technologies where capacitance and inductance from densely packed conductors limit the speed through a computer.

5. DISCUSSION

All PE's must be equally accessible from any other

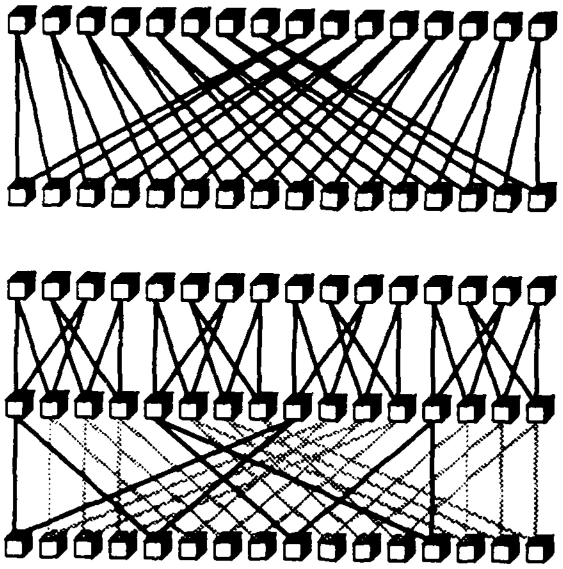


Fig. 16. A 16-wide perfect shuffle decomposes into a shuffle of four 4-wide perfect shuffles.

PE, so that the pipelined PE's that are in different phases of operation must be synchronized. Fig. 17 shows how this can be done using a fanout tree with optical delays.

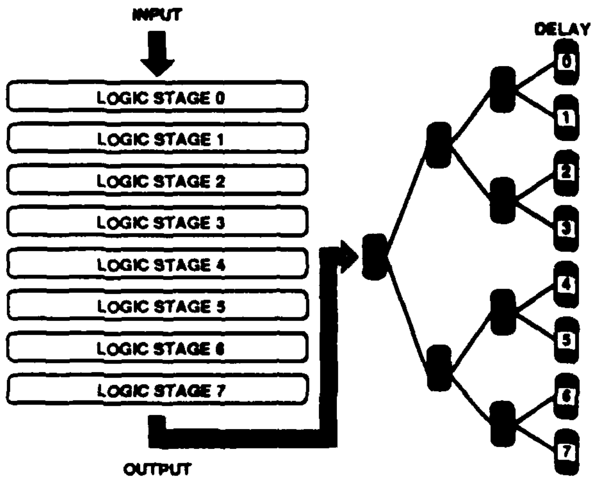


Fig. 17. An 8 level pipelined logic unit.

The output of the 8-level logic unit is fed to a delay tree, where each output is marked in the diagram by the amount of delay along that path. The cost of the fanout tree is $8 - 1 = 7$ fanout nodes, and in general the cost is $n - 1$ additional nodes for an n level deep decoder tree. The increased latency is n levels, so that the overall cost of pipelining at the gate level is a factor of two in gate count and and a factor of 2 in delay, while the saving in system gate count is $n - 2$ logic units. This tradeoff is made possible here because of the large number of identical logic units in the CM. In this way, the additional gate count caused by the strict use of regular interconnects at the gate level is offset by the savings gained in making use of gate-level pipelining.

6. CONCLUSION

An all-optical design was presented for the Connection Machine, using regular free-space interconnects between arrays of optical logic devices. The design demonstrates that existing digital optical design techniques based on free-space optics are capable of implementing efficient parallel optical architectures. Pipelining at the gate level is made possible by regular interconnection patterns. Component count is increased due to the forced regularity of the interconnects, which is offset by increased utilization from gate-level pipelining. In general, regular interconnects such as perfect shuffles, banyans, and crossovers are easier to implement optically using mirrors, lenses, and gratings than more complex interconnects that may require holograms or waveguides. We conclude that the strict use of regular free-space interconnects in the design of an optical CM incurs little or no additional cost when compared with a design using arbitrary interconnects.

7. REFERENCES

[1] D. A. B. Miller, D. S. Chemla, T. C. Damen, T. H. Wood, C. A. Burrus, A. C. Gossard, W. Wiegmann, "The Quantum Well Self-Electrooptic Effect Device: Optoelectronic Bistability and Oscillation and Self-Linearized Modulation", *IEEE J. Quant. Electron.*,QE-21, 1462, (1985).

[2] J. L. Jewell, A. Scherer, S. L. McCall, A. C. Gossard, and J. H. English, "GaAs-AlAs monolithic microresonator arrays," *Appl. Phys. Lett.*, **51**, 2, pp. 94-96, July 13, 1987.

[3] H. M. Gibbs, *Optical Bistability: Controlling Light with Light*, Academic Press Inc., New York, 1985.

[4] S. D. Smith, A. C. Walker, F. A. P. Tooley, and

B. S. Wherret, "The demonstration of restoring optical logic," *Nature*, **325**,pp. 27-31, Jan. 1, 1987.

[5] W. D. Hillis, *The Connection Machine*, The MIT Press, 1985.

[6] A. W. Lohmann, "Optical Perfect Shuffle," *Appl. Opt.*, **25**, No. 10, 1530, (May 15 1986).

[7] M. J. Murdocca, A. Huang, J. Jahns, N. Streibl, "Optical Design of Programmable Logic Arrays," to appear in *Appl. Opt.*

[8] M. J. Murdocca and B. Sugla, Design for an Optical Random Access Memory, submitted to *Appl. Opt.*

[9] M. J. Murdocca and N. Streibl, "A Digital Design Technique for Optical Computing," Topical Meeting on Optical Computing, Technical Digest Series 1987, 11, (Optical Society of America, Washington, D. C., 1987), pp. 9-11.

[10] Prise, M. E., Streibl, N., and Downs, M. M., "Computational Properties of Nonlinear Optical Devices," Topical Meeting on Optical Computing, Technical Digest Series 1987, 11, (Optical Society of America, Washington, D.C., 1987), pp. 110-112.

This project was funded in part by Air Force Office of Scientific Research grant AFOSR-86-0294.