# A Logic-in-Memory Architecture for Large-Scale-Integration Technologies

A.M. Peskin, Brookhaven National Laboratory

A computing machine is described which is structured around a distributed logic storage device called the Processing Memory. This machine, the Brookhaven Logic-In-Memory Processor (BLIMP), is meant only as a vehicle for simulating and evaluating its concepts, rather than for eventual fabrication. In particular, it is shown that the architecture used is very well suited to large-scale-integration (LSI) implementation technologies. It was first necessary to redefine the various goals of logic design optimization in the context of LSI implementation. Then an elemental building block of the Processing Memory is described as having evolved from associative memory circuits. It is shown that a computer such as BLIMP which utilizes the Processing Memory concept can meet the goals of design optimization for LSI. Design techniques for this project were developed as they were required. Of particular importance is a simulation system called MODEL, which documents the structure and analyzes the behavior of the proposed system.

KEY WORDS AND PHRASES: Associative Memory, Computer Architecture, Digital Simulation, Logic-in-Memory.
CR CATEGORIES: 6.20, 6.34, 8.1.

*Work performed under the auspices of the Atomic Energy Commission.

## MOTIVATION

In addition to providing greater speed and compactness in digital systems, large-scale-integration integrated circuit technology has redirected many of the goals of the digital system designer. In an era where hundreds of logic gates can be placed on a single semiconductor chip, design optimization must necessarily take on new meanings. It is no longer cogent to perform component counts alone to assess cost, for example, because external signal conductors are likely to be a much more precious commodity than semiconductor junctions or capacitance. Similarly, a design which can be broken up into a number of repeated modular circuits of few different types is optimized in a much more relevant fashion than a design which is optimized in the more classic sense of Boolean minimization.[1]

In the context of LSI, the impact on digital processor design has been to impose new goals and eliminate some of the old ones. Modular partitioning of logic into identical cells of as few types as possible becomes more important than the total number of cells. Partitioning, in turn, must minimize the number of binary arguments externally passed from one module to another. In addition, in the faster logic families, care must be taken to ensure that interconnection length is minimized, thus restricting intermodule communication to "near neighbor" cells only.

Under these constraints, design complexity increases rapidly but in a predictable way. The basic building blocks or cells remain relatively simple but the manner in which they are assigned or adapted to problems requires the designer to build an intricate set of cross-referenced lists of information. In its simplest case, this information is merely a wire list but in more intricate circuits, the lists imply all the levels of nested modularity, and all the functional dependencies in the system. The problem of storing, processing, analyzing and modifying this information is best suited to computer-aided logic design techniques. Thus, for large systems

optimized for LSI, computer-aided-design becomes not just desirable, but mandatory.

A general purpose computer, called the Brookhaven Logic-In-Memory Processor (BLIMP) is described. This machine is imaginary in the respect that it will never be built or even fully designed, but is real in the sense that it serves as a vehicle for the analysis of several fully developed concepts. The architecture is designed to attempt to optimize implementation with high speed LSI. The requisite computer-aided desigh (CAD) techniques for this project were developed in the form of a language called MODEL. And methods were developed for performing computerized dynamic simulation of the design as a procedure for verifying the underlying concepts of the system as well as the details of the implemented design.

## THE ARRAY

The most unique feature of the Brookhaven Logic-In-Memory Processor is that most of the data processing, including many arithmetic operations, is performed on the data words while they are resident in a scratchpad storage buffer. Each scratchpad storage location is augmented with sufficient circuitry to perform the logical combinations or modifications without having to fetch the word into a separate mainframe processing element and then having to restore it. Hence, the computer is termed a logic-in-memory processor whose central arithmetic and logical unit is the scratchpad or processing memory (PM).

The idea of including in each PM data word the capability for arithmetic and accumulation is, by the standards of conventional computers, quite extravagant. But again, this is extravagance as measured by an archaic yardstick, namely component count. In terms of the goals of LSI optimization, a single augmented memory element may be propagated into an i by j array of identical elements to form a two-dimensional matrix. The PM then consists of "j" words, each of "i" bits in length. Most of the element interconnections are made with "nearest neighbors", and the ratio of external connections to internal nets is small. The large amount of duplication of function in each word of the PM can be used to good advantage to provide increased parallelism of operation for processing vector instructions. In terms of the new criteria, then, the PM can be thought of as a highly optimized design.

The development of the circuitry of each PM element can be described in terms of the evolution of a memory element from one having only conventionally addressed read and write capability, to an element augmented to exhibit associative or content addressable reference capability, and onward to its final form as an LIM array element. This process is exemplified in Figure 1.

Figure 1a depicts the logical equivalent of a conventional memory element, Mij, consisting of a flip-flop and associated address and timed gating circuits. The address line is unidirectional and the data line is bidirectional.

In Figure 1b, the associative property has been added to Mij by attaching an "exclusive or" function of Mij and the Data line i and their complements, under control of the ASSOC line. The Address line is now bidirectional. If the ASSOC line is active and a match occurs between the value of the Data i and Mij, Address j will be activated at the memory element as the partial result for that element. The Address line can be forced active by raising the Mask i line which has the effect of removing Mij from contention in the associative process, that is, rendering bit i as a "don't care".(2)

In Figure 1c, two new outputs, Si and Ci have been generated and these are the elemental sum and carry of Data i and Mij under the control of line ADD. The Mask i line now assumes the added role of carry propagation. Thus with a proportionally small increase in the number of logic gates, each word of an associative memory can be fashioned into a conventional binary adder.

Figure 1d shows the elemental circuitry in its complete form. The addition of a feedback gate on the Si line creates one bit of an Accumulator register under the control of the ACCUM line. A five input and-or circuit allows the adder to be used for numerous other functions under program control. One's complement, left and right shifts and push-up or push-down stacking turn the array into a general purpose processor. Two new busses are brought into Mij under control of selective gating. One of these, Si, is the same net as output Si. Tk is called the transpose bus and is used for vector operations. The merged data line may be temporarily stored under control of the LATCH line, and it will be shown that this feature decreases the total number of PM interconnections by a significant factor.

The Mij element design implies the creation of five types of information busses in addition to the control inputs and logic outputs. The Data bus is used for the conventional fetching or storing of information as well as associative reference input words. The Sum bus contains arithmetic results. The Mask/Carry bus is used to control the masking of operands during an associate and, during arithmetic operations, propagates carry or borrow information. The Address bus, of course, contains word selection information.

To provide true parallelism of operation, a separate set of these busses

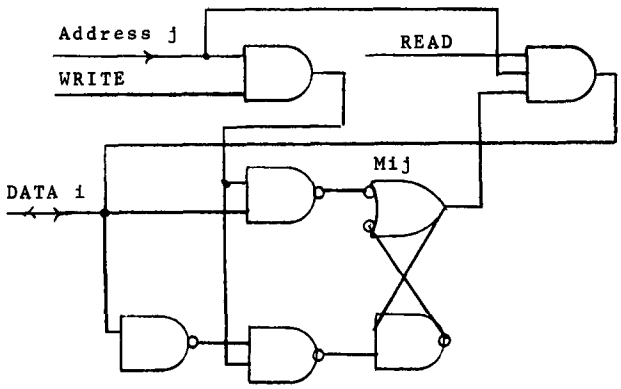



FIGURE 1a

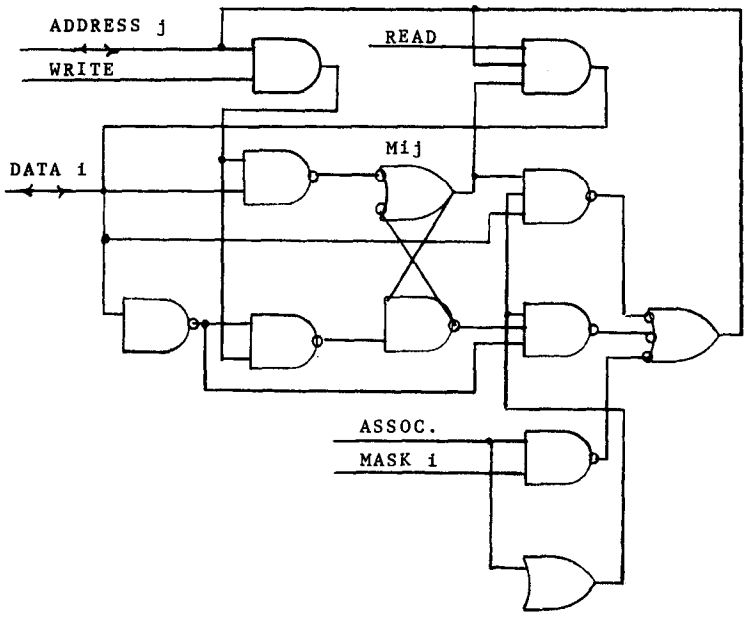



FIGURE 1b

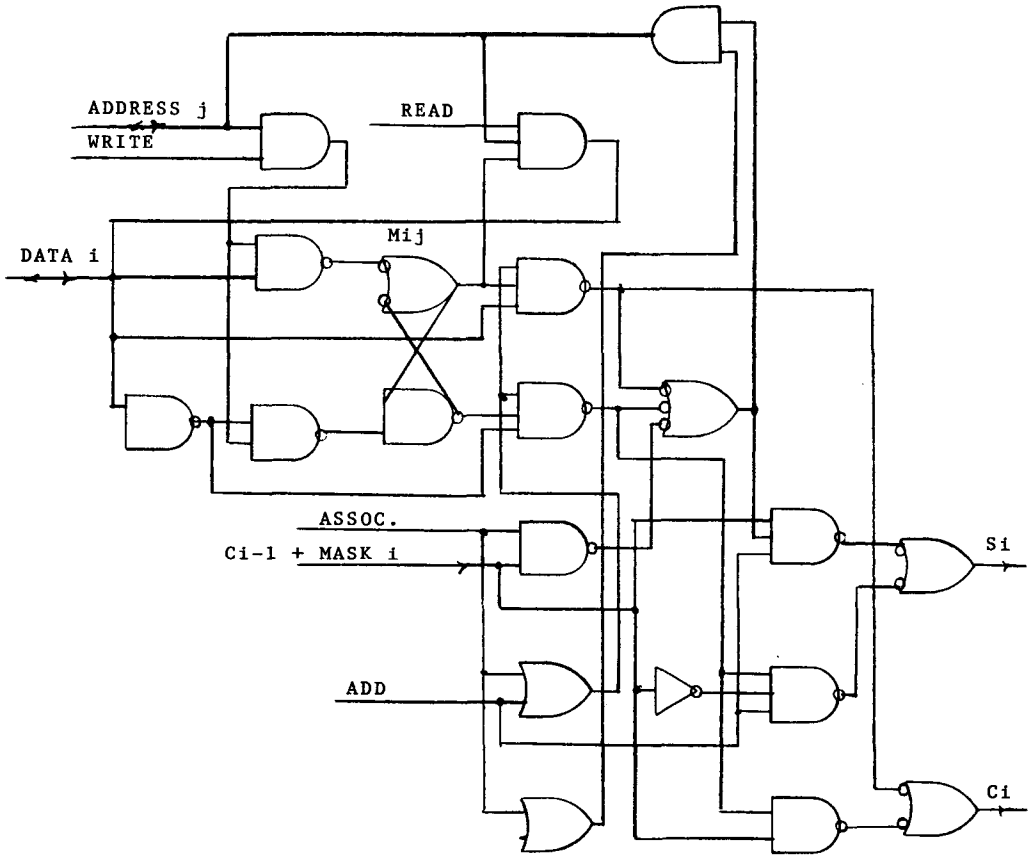



FIGURE 1c

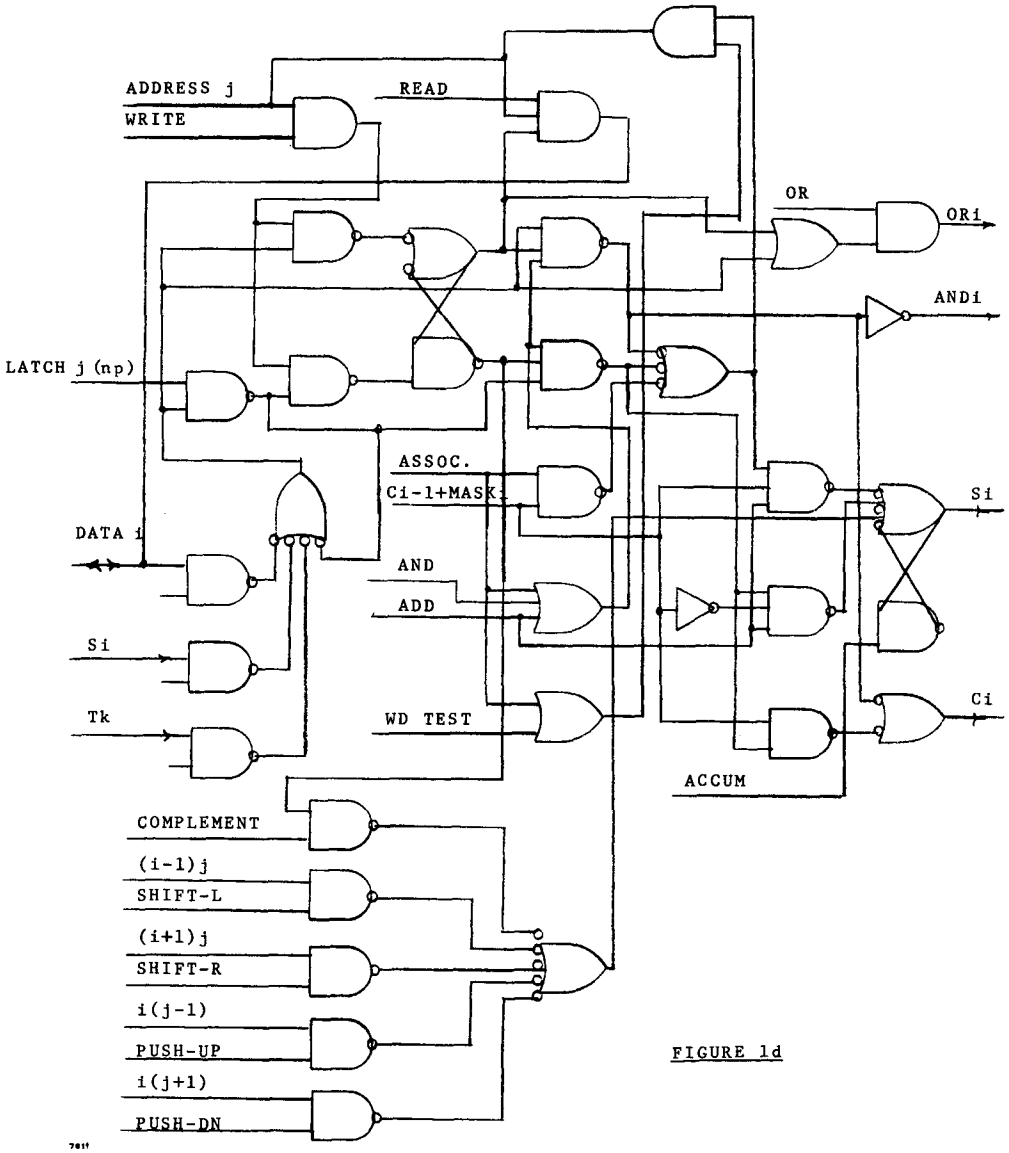

ADDRESS j
WRITE
READ
OR
ORi
ANDi
LATCH j (np)
ASSOC.
Ci-1+MASK
Si
DATA i
AND
ADD
Si
Tk
WD TEST
Ci
ACCUM
COMPLEMENT
(i-1)j
SHIFT-L
(i+1)j
SHIFT-R
i(j-1)
PUSH-UP
i(j+1)
PUSH-DN

FIGURE 1d

would be required for each set of operands. The number of busses for the entire processing memory then, would reach the hundreds, and the packaging of such a system would become quite wasteful if not impossible. To alleviate this situation and yet leave the basic concept intact, the PM is divided up into a number of sectors, with "p" words per sector. The value of p will be shown to be the central parameter of this architecture. There is a separate Data, Sum, and Carry/Mask bus for each sector of the PM, and the busses of each sector are independent of those of the other sectors. Intersector information exchange will take place on the Transpose bus.

Sharing of busses among the words of a single sector is accomplished through time division multiplexing of the busses with time slices for a given word assigned in accordance with the value of the lower order bits of the PM word address. It is most judicious, therefore, that p be a binary power. It is the time sharing of the busses that creates the need for the LATCH line of Figure 1d. Data for each word location can be sampled in a phased manner and held in the input flip-flop, creating the effect of the data being present on the bus for as long as it is required.

The central parameter p can be chosen by analyzing the relationships of certain other properties of the PM. We can define the following:

1) t; the propagation time from the output of one word to the input of another over a bus. This number reflects primarily the setting speed of the latching flip-flops.
2) s; The scalar processing cycle time of the PM. This is essentially the add time.
3) q; The projected ratio of vector instructions per instruction, based upon some knowledge of the job and instruction mix.

A certain processing balance may be said to be achieved if the time of scalar instruction processing is rendered equal to the time required for complete processing of vector instructions, as this situation will maximize the utilization ratio of much of the control circuitry. Then the central parameter, p, can be computed from this relationship.

$$p = \frac{s}{tq}$$

This computation is subject to the admonition that for design convenience, p be a binary power. The value of p is used several places in the design process. It represents the number of words per sector and, as such, it also implies the smallest usable vector size, and the number of time slices needed for each bus. In terms of loading and unloading the PM from a larger, slower, main storage, the value of p represents the minimum degree of interleaving necessary to match main storage to the PM speed. Furthermore, if we define n as the number of sectors needed, then (p)(n)=m, the number of words in the processing memory. Experience garnered by others in systems where contiguous fields of operands are prefetched indicates that n need be at least 4 but more probably 8 in most multiprogrammed environments.(3)

Representative numerical values can be chosen for other parameters as well. Assuming an implementation in high speed TTL or medium speed ECL, s can be assumed to be 50 nanoseconds, and t can be set at 12 nanoseconds. The value of q may not exceed .5 for even the most highly parallel problems. The value of the central parameter, then, would be close to eight. This in turn indicates that a main storage cycle of 400 nanoseconds is sufficient to keep the PM active. Maximum effective instruction processing rates would approach 20 million scalar instructions per second, or 80 million vector elements per second in these technologies.

Of course, the above analysis, to be exhaustive, would have to take many more factors into account. Such considerations as instruction power, operating system philosophy, and job stream control should be considered, and these factors make the system design job much more complex. But the above analysis provides a method of determining hardware balance within a given technology for this particular architecture.

## LOADING THE PROCESSING MEMORY

It is obvious that the ultimate improvement in performance derived from the PM concept is highly dependent upon the probability of finding the needed data already in the PM. If the data can be found resident in the PM before processing a large percentage of the time, then the overhead incurred in unloading and reloading the memory is minimized. For this reason, a study was made of scratchpad loading strategies and their effect on scratchpad storage latency as it affects the performance of BLIMP.

Two strategies were investigated. These appear to be the general case of almost all reasonable particular strategies currently used or proposed. The performance of each was simulated using numbers representing actual reference address sequences obtained from a variety of computers' object codes. The strategies were analyzed comparatively for several different buffer sizes. The number of hits (that is, the number of times the reference address was found to be in the scratchpad) and the reference time, including overhead, were recorded and comparatively analyzed.

Strategy I is the most commonly used

scratchpad loading strategy. It is the strategy employed by the IBM 360/85 and 195 cache storage.[3] It requires that if a referenced address is not contained in the cache, the buffer locations currently residing are restored to those addresses in main storage and a new set of contiguous data words, beginning with the location currently of interest, is loaded from main memory into the cache. Strategy II, which at first glance may appear less appealing, calls for replacement of only that word which is of current interest. The word is placed in an address which is the modulo address of its main storage location, modulus the buffer size. For example, if a word at main storage location $257_{10}$ were called for, it would be stored in location 1 of a 256 word buffer. As with Strategy I, the data which is replaced in the scratchpad is returned to main storage.

In the simulation, buffer sizes of 64, 256, 1024 and 4096 words were used for each strategy. In calculating the time overhead, a main storage interleaving factor of 4 was assumed. Most programs simulated required 10,000 to 20,000 memory references from approximately 8,000 main storage locations. A figure of merit was calculated as follows:

$$x = e^{5.0(\frac{h}{r}) - 1}$$

Here h is the number of hits and r is the number of simulated time units incurred for storage reference. x=1 indicates a break-even point; that the scratchpad technique neither enhanced nor detracted from the performance. Some results are plotted in Figure 2.

The results of this sketchy simulation experiment are by no means conclusive, but certain interesting points can be made. The relative performance of the two strategies was found to depend not only on the size relationship of the buffer and program but also to some extent on the correlation coefficient of the reference address sequence, with more highly correlated sequences tending to favor Strategy I. The effect of having multiple buffers in either strategy increases the performance of that strategy, and, as one might expect, the greater the number of buffers, the more similar the performance characteristics of the two schemes. In general, it appears that smaller machines would benefit most from a scratchpad adhering to Strategy II, while larger ones would perform better with Strategy I.

In terms of BLIMP, where the architectural behavior may be varied through a microprogram, each strategy would have its place. Strategy I would be employed for highly correlated or vector operations, and Strategy II would be used for a list, logical and other less correlated operations. All this is of course dependent upon the ability to have a priori knowledge (perhaps from compile time analysis) of the execution time behavior of the data.

## THE BLIMP PROCESSOR

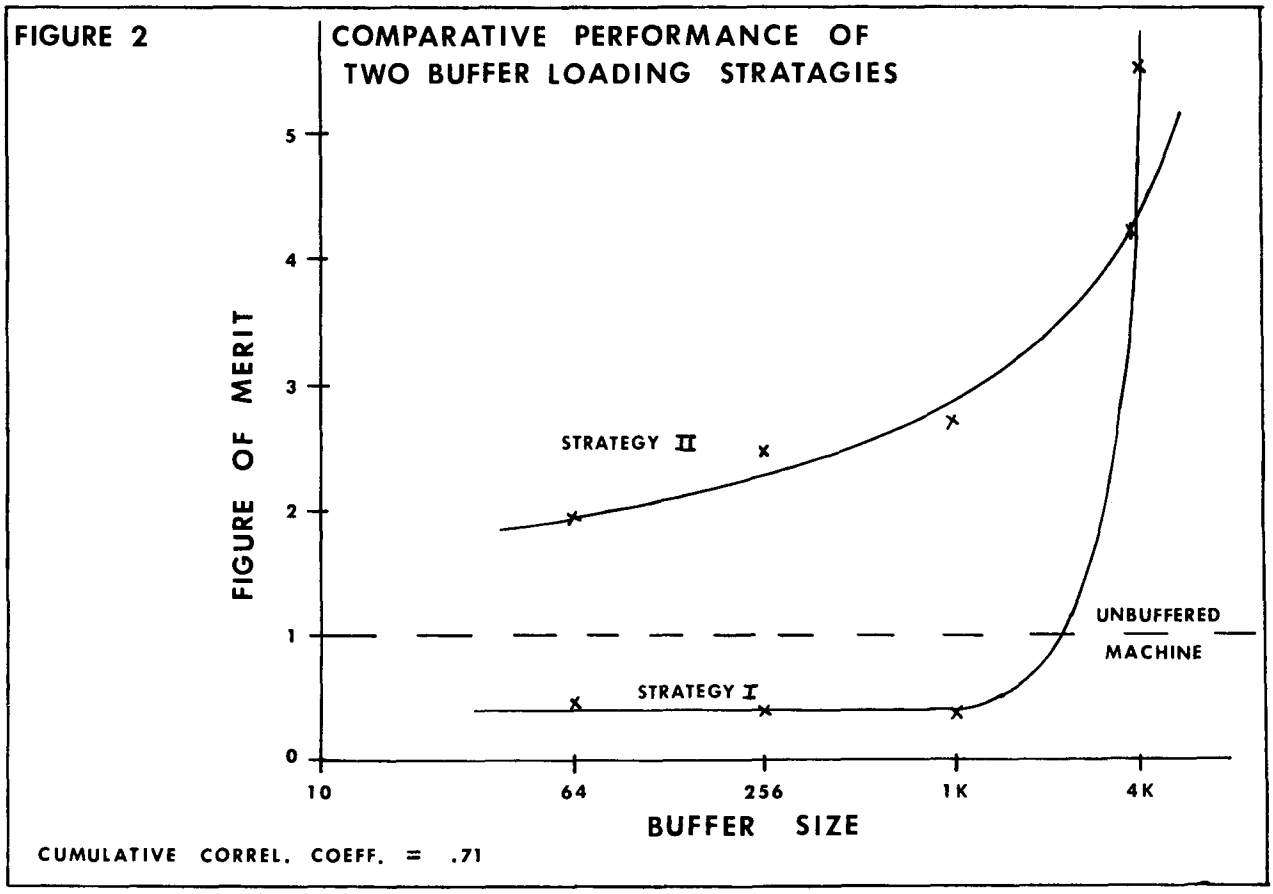
It is difficult to analyze the operation or assess the value of the Processing Memory described, except in conjunction with the other components of a computer system. For such a purpose, a fictionalized system is proposed which contains all the elements necessary to implement a general purpose computer that utilizes the logic-in-memory approach. The other units, however, need only be specified and designed to the level of detail necessary to analyze the PM and its architecture and simulate the salient interactions. Hence, it is not the purpose of the BLIMP to become an actual computer, which has freed investigators from the tedious tasks of designing the more commonplace modules completely. Rather, the BLIMP outlines an apprach to the utilization of such storages as the central processing element of a high performance LSI machine.

The block diagram of the system is shown in Figure 3. In this machine, the PM size is 64×64 bits with a central parameter of 8. The memory is imbued with transfer and processing rates discussed previously; the add speed and block transfer bandwidth is 50 nanoseconds, and thus the "p" element vector processing time is 100 nanoseconds.

There are several floating point processes which do not seem amenable to the logic-in-memory approach. For this reason a scalar Floating Point Unit has been attached to the PM, to perform normalization and the more complex operations such as division and root calculations. The speed of the binary multiply may also be increased by the use of dedicated shift circuitry external to the PM, but this was not considered essential to the BLIMP design.

Also connected to the PM Data bus is the Instruction Decode Unit. It is the purpose of this unit to fetch instruction words out of the PM and initiate the appropriate action to execute the instruction. The Instruction Decode Unit is connected to the I/O Processor to initiate I/O transfers and pass arguments necessary for I/O processing. It is also connected to the Multiloader, a device which initiates eight word transfers between the PM and Main Storage. In the event that an instruction cannot be executed entirely from information or operands currently resident in the PM, a transfer will take place to bring in the necessary blocks and return the least recently used block back to Main Storage. Similarly, if the IDU determines that the next instruction to be executed is not in the PM, a transfer will be initiated to bring in the new instruction stream. In this respect, the PM load algorithms are

FIGURE 2


COMPARATIVE PERFORMANCE OF
TWO BUFFER LOADING STRATAGIES
FIGURE OF MERIT
5
4
3
2
1
0
STRATEGY II
UNBUFFERED
MACHINE
STRATEGY I
10
64
256
1K
4K
BUFFER SIZE
CUMULATIVE CORREL. COEFF. = .71

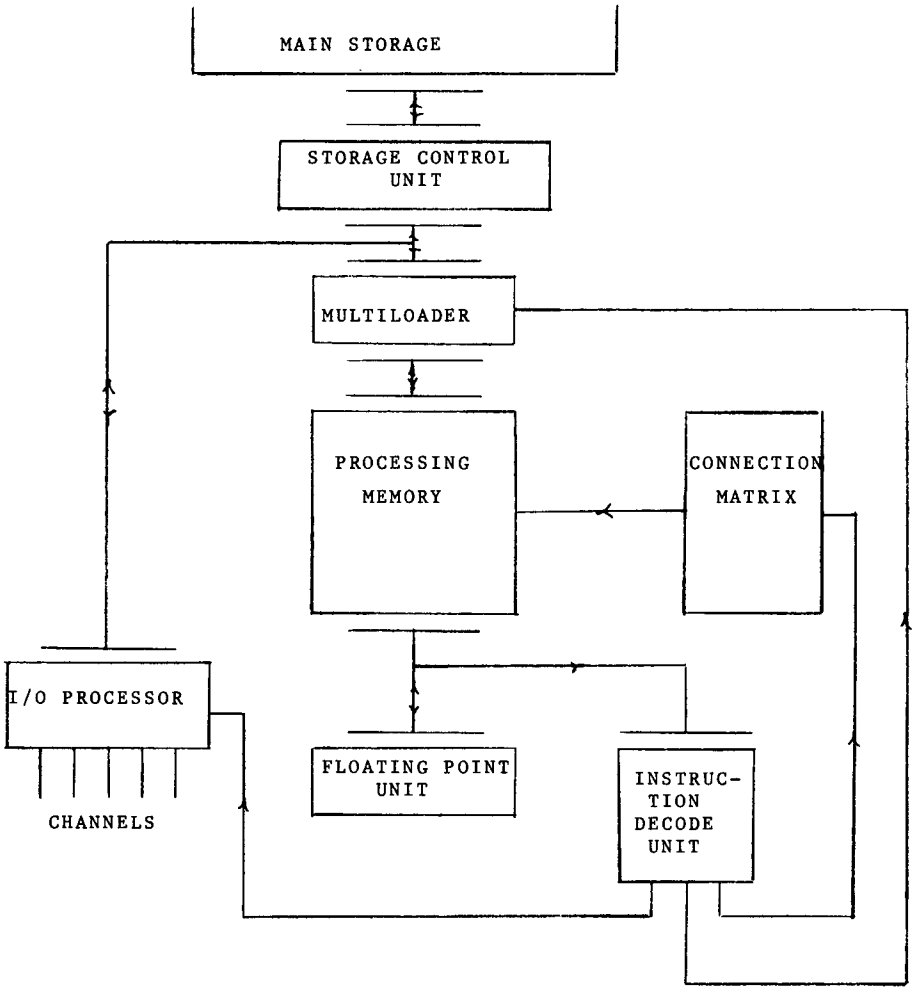



FIGURE 3

BLIMP CONFIGURATION

very similar to the Cache concept used by IBM, and to the load Strategy I described in the section, Loading the Processing Memory. The IDU, then, must itself have a small associative storage to determine if a given Main Storage address to be referenced has an image residing in the PM.[4,5]

When the IDU determines that all data processing is ready to take place inside the PM, it passes the instruction information to the Connection Matrix whose purpose is to manipulate the control lines for the various PM words including the time-division-multiplexed timing and gating pulses. The smallest allowable distinct operand is sixteen bits long, so only four independent sets of control lines per word are required. The maximum number of lines from the Connection Matrix to a 64×64 PM of the design of Figure 1d is over 2,000, but since not all operations are subject to byte manipulation requirements, and certain combinations are mutually exclusive, this number can be reduced somewhat. The Connection Matrix, under control of the Instruction Decode Unit, provides all the gate and inhibit controls necessary to forge the "plasma" of interconnects surrounding each PM word into the appropriate specialized functional unit to execute the instruction.

The Storage Control Unit is a high bandwidth data exchange device which handles storage requests from either the Multiloader or the I/O Processor to Main Storage, which is specified as a 400 nanosecond, eight-way interleaved conventional array. Each eight-word storage cycle is honored through a d-c interlocked, ready-response hardware protocol to allow for the inclusion of slower buffered storages or facilities in the configuration. None of the unit specifications mentioned go beyond the current state of the art and are in fact, exceeded in some commercially available computers.[6]

The design outlined above implies the nature of this machine's instruction set. The complete list of instructions appears in Table 1. Most of the operations are self-explanatory. Since the BLIMP is designed for simulation and analysis only, a simplifying extravagance has been used in the instruction format. The instruction word size is given as 64 bits long and all operands are directly addressable. All instructions have two 24-bit address operands plus an "op code" portion. In any future implementation, several different modes of addressing must be provided, thus considerably enriching the instruction repertory.

The instruction set of Table 1 is certainly quite minimal, but several of the less conventional entries require some explanation. Foremost among these are the Vector DEFINE instructions. Each word in both main storage and the PM has two tag bits prefixed to them which are interpreted by the internal logic as "continue" indicators for element strings. Thus a vector (or an I/O buffer) is defined by specifying its upper and lower address limits and one or the other of the tag bits in each word after the first are set. Similarly, a stack may be defined; in this case, both tag bits are set for each continuation word. The PUSH-UP and PUSH-DN transfer functions are only defined for element fields which are defined as stacks. A vector, buffer or stack may be specified by any of the addresses included in its range.

Table 1. BLIMP Instructions

| Transfer Instructions | Operands | Comments |
|---|---|---|
| POP | A,B | Word from vector B stored into location A |
| PUSH | A,B | Word from location A stored into vector B |
| MOVE | A,B | |
| **Binary Arithmetic Instructions** | | |
| ADD | (A)+(B)=(A) | |
| SUBTRACT | (A)-(B)=(A) | |
| SHIFT L | A,B | Shift word A into number of places specified by count in B |
| SHIFT R | A,B | |
| COMPLEMENT | A | One's complement |
| TWOS COMP. | A | Two's complement |
| MULTIPLY | $(A)\times(B)=A$ | |
| DIVIDE | $(A)\div(B)=A$ | |
| INCREMENT | A | |
| DECREMENT | A | |
| **Floating Point Instructions** | | |
| ADD F | (A)+(B)=(A) | |
| SUBTRACT F | (A)-(B)=(A) | |
| MULTIPLY F | $(A)\times(B)=(A)$ | |
| DIVIDE F | $(A)\div(B)=(A)$ | |
| **Logical Instructions** | | |
| AND | (A)-(B)=(A) | |
| OR | (A)+(B)=(A) | |
| EXCLOR | $((A)+(B))\overline{((A)\cdot(B))}=(A)$ | |
| ASSOC. & COUNT | A,B | Address by content of A, searching through vector B, put match count in A |
| ASSOC. & STACK | A,B | Address by content of A, searching through vector B, stack addresses in vector starting at address A. |
| SET MASK | A | Use contents of A. |
| **Vector Instructions** | | |
| DEFINE BUFFER | A,B | A is lower address limit |
| DEFINE VECTOR | A,B | B is upper address limit |
| DEFINE STACK | A,B | |
| TRANSPOSE | $(A)^{T}=(B)$ | |
| COFACTOR | $(A)^{C}+(B)$ | |
| MULTIPLY V | $(A)\times(B)=(A)$ | |
| ADD V | (A)+(B)=(A) | |
| SUBTRACT V | (A)-(B)=(A) | |
| DETERMINANT | $\lvert A\rvert=(B)$ | |
| DIVIDE V | $(A)\div(B)=(A)$ | $(A_i)\div(B_i)=(A_i)$ for all i |

| Control Instructions | Operands | Comments |
|---|---|---|
| HALT | | |
| NOOP | | |
| INT. ENTABLE | A | A Contains interrupt mask |
| INT. CLEAR | A | |
| VECTOR STATUS | A,B | {Checks vector (B) for element overruns, arithmetic |
| ERROR STATUS | | {errors, etc., puts status words in A. |
| SKIP EQ | A,B | Skip if (A)=(B) |
| SKIP Z | A | Skip if (A)=0 |
| SKIP L | A,B | Skip if (A)≥(B) |
| SKIP G | A,B | Skip if (A)<(B) |
| GO TO | | |

## THE MODEL LANGUAGE

The intent of the MODEL language is to provide a simulation system which allows one to analyze the static structure and dynamic behavior of a digital system. This is the simulator used to study the LIM array in the BLIMP architecture. There are several unusual characteristics of this simulation system that uniquely qualify it for the job.

Two such features are its macro and functional unit specifications. As previously stated, it was not considered necessary to design all BLIMP subsystems in identical levels of detail; in many cases, a functional unit is described only in terms of its external characteristics. Yet, because these units may interact in critical paths of the BLIMP, it is essential that the impact of these units is felt in the simulation. The FUNC statement in the MODEL language makes this possible.

Even for those subsystems whose internal design is specified in detail, the FUNC statement is useful if that subsystem is to be utilized in more than one place in the total system. The statement allows the designer to assign a designation to the unit, and that designation may be a dimensioned name. This is a particularly convenient option because one of the primary goals of LSI implementation is the partitioning of systems into repeated arrays of modules.

It is also convenient to be able to refer to commonly used sequences of dynamic operations (called Action statements) by a single name or macro instruction. The statement MACRO enables a user to define such sequences. The individual Action statements then represent activities at the internal clock level, and MACROs may be assigned to defined machine instructions as well as sequences of instructions.

The input and output formats in the simulation system are those with which a logic designer would be familiar. A complete list of statements appears in Table 2. Outputs take the form of multiple oscilloscope traces for added familiarity. To increase the versatility of MODEL and allow it to run on most large-scale scientific computers, the base language is Fortran.

Table 2. MODEL Statements

Device Definition Statements:

NAME(I,J)/TYPE/INPUTS/OUTPUTS/DELAY
(Types are: JK,RS,AND,OR,INV,NAND,NOR, and TIE)
NAME(I,J)/FUNCTION/INPUTS/OUTPUTS

Connection Statements:

FROMTO/Name,Output No./Name, Input No.
BUS/Name(i,J), Output K/Name (M,N),
Output L/I=A,B/J=C,D/M=E,F/N=G,H

Monitoring Statements:

SCOPE/Output list
SNAP/output list

Control Statements:

END FUNCTION
END MACRO

Action Statements:

START
STOP
MACRO
IF
GOTO
GENSIG/VAR/Input/Duration
WAIT
Q=
T=
QUOTE

Referring to Table 2, it is seen that there are sixteen statements currently defined for MODEL which are divided into six major categories. Device Definition statements allow the user to specify any of the several common logic building blocks currently available such as JK-type flip-flops or nand gates. One must also

specify the number and types of inputs and outputs available or utilized, and the nominal circuit delay. New or more complicated building blocks are specified by the FUNCTION statement. Device interconnections are stated explicitly for ease of wire sorting operations. They are effected by either the FROMTO or BUS statements, the latter being used for convenience when array interconnects lend themselves to automatic handling.

There is actually only one "event" in the simulation and that is initiated by the GENSIG Action statement. The other Action statements provide only for program control. Execution of GENSIG modifies a logic level which generally starts a chain reaction of sequential instabilities, hopefully a predictable one, which leads to the desired result.

Program output is generated in response to one or more of the Monitoring statements. SNAP elicits a binary snapshot of the desired outputs at a specified point in time. SCOPE gives a time-history of those outputs.

## FUTURE INVESTIGATIONS

It is already clear that logic-in-memory processors are particularly well suited to LSI, but various improvements in both design and technique appear worthy of investigation. For instance, in such a highly parallel system as BLIMP, it is anticipated that a single instruction stream is not sufficient to keep the various partitions of the system at a high utilization level. An Instruction Decode Unit is needed which is capable of sustaining two or more instruction streams, each eminating from an independent task. Plots of unit utilization versus the number of streams for different central parameter values are expected to yield quite definitive information about the optimization of design.

The method of encoding the binary arithmetic function in the LIM was chosen for simplicity and the fact that it illustrates the evolution of the LIM from an associative memory. There are several other methods of performing binary arithmetic which may, in the final analysis, be more suited to a logic-in-memory machine. Both "bit-serial, element-parallel" methods and highly parallel recoded methods will be investigated.[7,8]

With the simulation techniques afforded by such systems as MODEL, it seems no longer necessary to build machines in order to test out machine design concepts. But because a physical realization resulting in a usable prototype has not been accomplished, there is no sure way of verifying these concepts in an operating environment. However, MODEL can be used to extract information about instruction execution speeds and inter-instruction parallelism and interference. There appears to be no reason why a compiler cannot be written for a proposed processor which results in machine level instructions which in turn can be translated to MACRO statements for MODEL processing. In this way, "benchmark timing" and perhaps even some elementary processed results can be obtained, thus verifying the machine design and measuring its performance.

## REFERENCES


1. Stone, Harold S., "A Logic in Memory Computer", IEEE Transactions on Computers, January 1970.

2. Peskin, Arnold M., "Associative Capabilities for Mass Storage Through Array Organization", Proceedings of the 1970 FJCC.

3. "IBM System/360 Model 195 Functional Characteristics", IBM Systems Reference Library, A22-6943-0, 1969.

4. Ibid.

5. Aspinall, Kinnitment, Edwards, "Associative Memories in Large Computer Systems", Proceedings of the 1968 IFIPS Congress.

6. "Control Data 7600 Computer System Reference Manual", Control Data Corporation, 1970.

7. Robertson, James S., "A Deterministic Procedure for the Design of Carry-Save Adders and Borrow-Save Subtracters", University of Illinois Report, 1967.

8. Gilmore, Paul A., "Matrix Computation on an Associative Processor", Goodyear Aerospace Corporation Report, 1971.