Illiac IV

R.N.F.

Chapter 27

The ILLIAC IV computer[1]

George H. Barnes / Richard M. Brown / Maso Kato
David J. Kuck / Daniel L. Slotnick / Richard A. Stokes

Summary The structure of ILLIAC IV, a parallel-array computer containing 256 processing elements, is described. Special features include multiarray processing, multiprecision arithmetic, and fast data-routing interconnections. Individual processing elements execute 4×10^6 instructions per second to yield an effective rate of 10^9 operations per second.

Index terms Array, computer structure, look-ahead, machine language, parallel processing, speed, thin-film memory.

Introduction

The study of a number of well-formulated but computationally massive problems is limited by the computing power of currently available or proposed computers. Some involve manipulations of very large matrices (e.g., linear programming); others, the solution of sets of partial differential equations over sizable grids (e.g., weather models); and others require extremely fast data correlation techniques (phased array signal processing). Substantive progress in these areas requires computing speeds several orders of magnitude greater than conventional computers.

At the same time, signal propagation speeds represent a serious barrier to increasing the speed of strictly sequential computers. Thus, in recent years a variety of techniques have been introduced to overlap the functions required in sequential processing, e.g., multiphased memories, program look-ahead, and pipeline arithmetic units. Incremental speed gains have been achieved but at considerable cost in hardware and complexity with accompanying problems in machine checkout and reliability.

The use of explicit parallelism of operation rather than overlapping of subfunctions offers the possibility of speeds which increase linearly with the number of gates, and consequently has been explored in several designs [Slotnick et al., 1962; Unger, 1958; Holland, 1959; Murtha, 1966]. The SOLOMON computer [Slotnick et al., 1962], which introduced a large degree of overt parallelism into its structure, had four principal features.

1 A large array of arithmetic units was controlled by a single control unit so that a single instruction stream sequenced the processing of many data streams.

2 Memory addresses and data common to all of the data processing were broadcast from the central control.

3 Some amount of local control at the individual processing element level was obtained by permitting each element to enable or disable the execution of the common instructions according to local tests.

4 Processing elements in the array had nearest-neighbor connections to provide moderate coupling for data exchange.

Studies with the original SOLOMON computer indicated that such a parallel approach was both feasible and applicable to a variety of important computational areas. The advent of LSI circuitry, or at least medium-scale versions, with gate times of the order of 2 to 5 ns, suggested that a SOLOMON-type array of potentially 10^9 word operations per second could be realized. In addition, memory technology had advanced sufficiently to indicate that 10^6 words of memory with 200 to 500-ns cycle times could be produced at acceptable cost. The ILLIAC IV Phase I design study during the latter part of 1966 resulted in the design discussed in this paper. The machine, to be fabricated by the Defense Space and Special Systems Division of Burroughs Corporation, Paoli, Pa., is scheduled for installation in early 1970.

Summary of the ILLIAC IV

The ILLIAC IV main structure consists of 256 processing elements arranged in four reconfigurable SOLOMON-type arrays of 64 processors each. The individual processors have a 240-ns ADD time and a 400-ns MULTIPLY time for 64-bit operands. Each processor requires approximately 10^4 ECL gates and is provided with 2048 words of 240-ns cycle time thin-film memory.

Instruction and addressing control

The ILLIAC IV array possesses a common control unit which decodes the instructions and generates control signals for all

[1] *IEEE Trans., C-17,* vol. 8, pp. 746–757, August, 1968.

processing elements in the array. This eliminates the cost and complexity for decoding and timing circuits in each element.

In addition, an index register and address adder are provided with each processing element, so that the final operand address a_i for element i is determined as follows:

$$a_i = a + (b) + (c_i)$$

where a is the base address specified in the instruction, (b) is the contents of a central index register in the control unit, and (c_i) is the contents of the local index register of the processing element i. This independence in operand addressing is very effective for handling rows and columns of matrices and other multidimensional data structures [Kuck, 1968].

Mode control and data conditional operations

Although the goal of the ILLIAC IV structure is to be able to control the processing of a number of data streams with a single instruction stream, it is sometimes necessary to exclude some data streams or to process them differently. This is accomplished by providing each processor with an ENABLE flip-flop whose value controls the instruction execution at the processor level.

The ENABLE bit is part of a test result register in each processor which holds the results of tests conditional on local data. Thus in ILLIAC IV the data conditional jumps of conventional computers are accomplished by processor tests which enable or disable local execution of subsequent commands in the instruction stream.

Routing

Each processing element i in the ILLIAC IV has data routing connections to 4 of its neighbors, processors $i + 1$, $i - 1$, $i + 8$, and $i - 8$. End connection is end around so that, for a single array, processor 63 connects to processors 0, 62, 7, and 55.

Interprocessor data transmissions of arbitrary distance are accomplished by a sequence of routings within a single instruction. For a 64-processor array the maximum number of routing steps required is 7; the average overall possible distances is 4. In actual programs, routing by distance 1 is most common and distances greater than 2 are rare.

Common operand broadcasting

Constants or other operands used in common by all the processors are fetched and stored locally by the central control and broadcast to the processors in conjunction with the instruction using them. This has several advantages: (1) it reduces the memory used for storage of program constants, and (2) it permits overlap of common operand fetches with other operations.

Processor partitioning

Many computations do not require the full 64-bit precision of the processors. To make more efficient use of the hardware and speed up computations, each processor may be partitioned into either two 32-bit or eight 8-bit subprocessors, to yield 512 32-bit or 2048 8-bit subprocessors for the entire ILLIAC IV set.

The subprocessors are not completely independent in that they share a common index register and the 64-bit data routing paths. The 32-bit subprocessors have separate enabled/disabled modes for indexing and data routing; the 8-bit subprocessors do not.

Array partitioning

The 256 elements of ILLIAC IV are grouped into four separate subarrays of 64 processors, each subarray having its own control unit and capable of independent processing. The subarrays may be dynamically united to form two arrays of 128 processors or one array of 256 processors. The following advantages are obtained.

1 Programs with moderately dimensioned vector or matrix variables can be more efficiently matched to the array size.

2 Failure of any subarray does not preclude continued processing by the others.

This paper summarizes the structure of the entire ILLIAC IV system. Programming techniques and data structures for ILLIAC IV are covered in a paper by Kuck [1968].

ILLIAC IV structure

The organization of the ILLIAC IV system is indicated in Fig. 1. The individual processing elements (PEs) are grouped in four arrays, each containing 64 elements and a control unit (CU). The four arrays may be connected together under program control to permit multiprocessing or single-processing operation. The system program resides in a general-purpose computer, a Burroughs B 6500, which supervises program loading, array configuration changes, and I/O operations internal to the ILLIAC IV system and to the external world. To provide backup memory for the ILLIAC IV arrays, a large parallel-access disk system (10 bits, 10^9 bit per second access rate, 40-ms maximum latency) is directly coupled to the arrays. There is also provision for real-time data connections directly to the ILLIAC IV arrays.

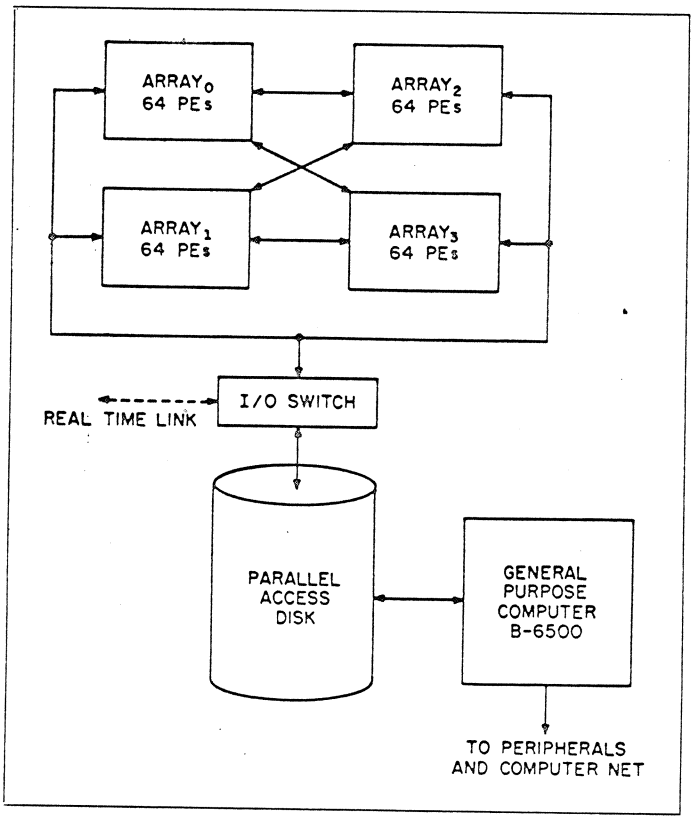


Fig. 1. ILLIAC IV system organization.

Array organization

The internal structure of an array is indicated in Fig. 2. The 64 processing elements in each array are arranged in a string and are controlled by the control unit (CU) which receives the instruction string, generates the appropriate control signals and address parameters of the instructions, and transmits them to the individual processing elements for execution. In addition, each CU can broadcast via the common data bus operands for common use (e.g., constant).

Full word length (64 bits) communication exists between the processing elements for exchange of information by organized routing of words along the string array. Direct routing connections exist for nearest neighbors and also for processing elements 8 units away. Routing for intermediate distances are generated via sequences of routes of $+1$, -1, $+8$, or -8. The end connections of the string are circular, but can be broken and connected to the ends of other arrays when the system is organized in one of the multiarray configurations.

All processing elements of an array execute, of course, the sam instruction in unison under the control of the CU; local contr is provided by the mode bit in each processing element whic enables or disables the execution of the current instruction. Th control unit is able to sense the mode bits of all processing ele ments under its control and thereby monitor the state of operation

Multiarray configurations

To permit more optimal matching of array size to problem struc ture, the four arrays may be united in three different configura tions, as shown in Fig. 3. To enlarge the arrays, the end connection of the PE strings are decoupled and attached to the ends of the other arrays to form strings of 128 or 256 processors. For multiarray configurations all CUs receive the same instruction string and any data centrally accessed. The control units execute the instructions independently, however, with inter-CU synchronization occurring only on those instructions in which data or control information must cross array boundaries. This simplifies and speeds up the instruction execution in multiarray configurations. The multiplicity of array configurations introduces complexities in memory addressing which will be discussed in a later section.

Control unit

The array control unit (CU) has the following five functions.

1 To control and decode the instruction streams
2 To generate the control pulses transmitted to the processing elements for instruction execution
3 To generate and broadcast those components of memory addresses which are common to all processors
4 To manipulate and broadcast data words common to the calculations of all the processors

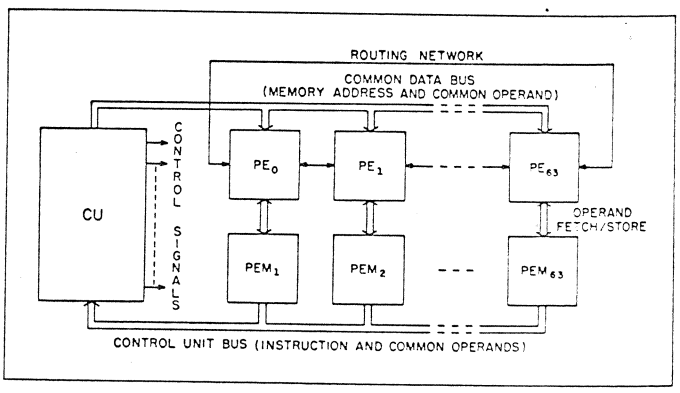


Fig. 2. Array structure.

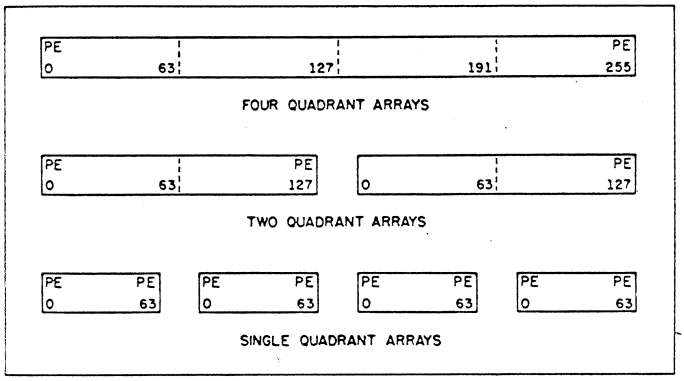


Fig. 3. Multiarray configurations.

5 To receive and process trap signals arising from arithmetic faults in the processors, from internal I/O operations, and from the B 6500.

The structure of the control unit is shown in Fig. 4. Principal components of the CU are two fast-access buffers of 64 words each, one associatively addressed, which holds current and pending instructions (PLA), and the other a local data buffer (LDB). The four 64-bit accumulator registers (CAR) are central to communication within the CU and hold address indexing information and active data for logical manipulation or broadcasting. The CU arithmetic unit (CULOG) performs addition, subtraction, and Boolean operations; more complex data manipulations are relegated to the PE's. To specify and control array configurations, there are three 4-bit configuration control registers whose use will be described in another section.

Instruction processing

All instructions are 32 bits in length and belong to one of two classes: CU instructions, which generate operations local to the CU (e.g., indexing, jumps, etc.), and PE instructions, which are decoded in the CU and then transmitted via control pulses to all the processing elements. Instructions flow from the array memory upon demand in blocks of 8 words (16 instructions) into the instruction buffer. As the control advances, individual instructions are extracted from the instruction buffer and sent to the advanced instruction station (ADVAST) which decodes them and executes those instructions local to the CU. In the case of PE instructions, ADVAST constructs the necessary address or data operands and stacks the result in a queue (FINQ) to await transmission to the PEs. PE instructions are taken from the bottom of the stack to the final instruction station (FINST) which controls the broadcast of address or data and holds the PE instruction during the execution period.

The use of the PE instruction queue permits overlap between the CU and PE instruction executions; the amount of overlap depends, of course, on the distribution of CU and PE instructions. As in all overlap strategies, careful attention to the instruction sequence by the programmer or compiler can result in considerable speedup of program execution.

The instruction buffer holds a maximum of 128 instructions, sufficient to hold the inner loop of many programs. For such loops, after initial loading, instructions are fetched from the buffer with minimal delay.

A variety of strategies for instruction buffer loading were examined, and the following straightforward approach was taken. When the instruction counter is halfway through a block of 8

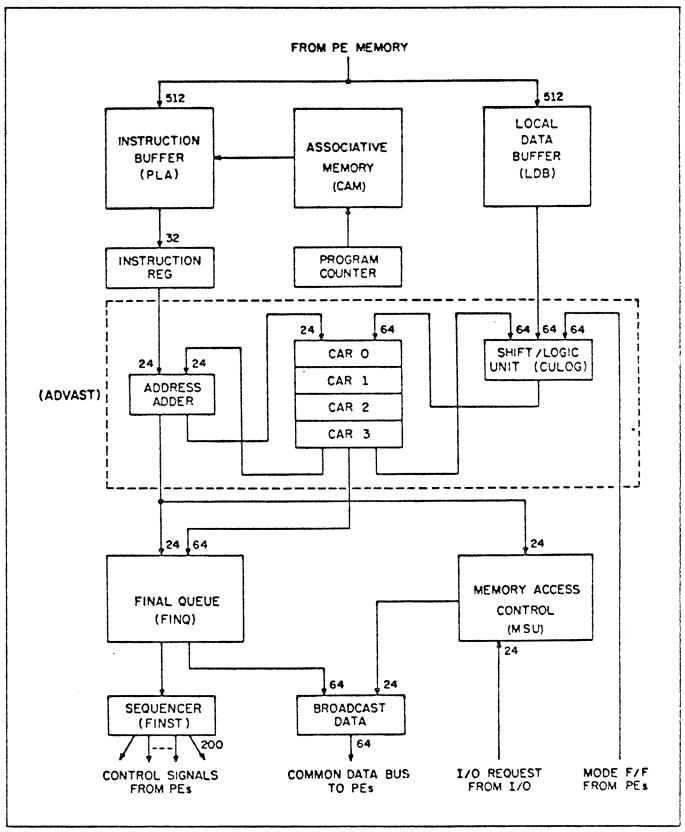


Fig. 4. Control-unit block diagram.

words (16 instructions), fetch of the next block is initiated; the possibility of pending jumps to different blocks is ignored. If the next block is found to be already resident in the buffer, no further action is taken; else fetch of the next block from the array memory is initiated. On arrival of the requested block, the instruction buffer is cyclically filled; the oldest block is assumed to be the least required block in the buffer and is overwritten. Jump instructions initiate the same procedures.

Fetch of a new instruction block from memory requires a delay of approximately three memory cycles to cover the signal transmission times between the array memory and the control unit. On execution of a straight line program, this delay is overlapped with the execution of the 8 instructions remaining in the current block.

In a multiple-array configuration, instructions are fetched from the array memory specified by the program counter, and broadcast simultaneously to all the participating control units. Instruction processing thereafter is identical to that for single-array operation, except that synchronization of the control units is necessary whenever information, in the form of either data or control signals, must cross array boundaries. CU synchronization must be forced at all fetches of new instruction blocks, upon all data routing operations, all conditional program transfers, and all configuration-changing instructions. With these exceptions, the CUs of the several arrays run independently of one another. This simplifies the control in the multiple-array operation; furthermore, it permits I/O transactions with the separate array memories without stealing memory cycles from the nonparticipating memories.

Memory addressing

Both data and instructions are stored in the combined memories of the array. However, the CU has access to the entire memory, while each PE can only directly reference its own 2,048-word PEM. The memory appears as a two-dimensional array with CU access sequential along rows and with PE access down its own column. In multiarray configurations the width of the rows is increased by multiples of 64.

The resulting variable-structure addressing problem is solved by generating a fixed-form 20-bit address in the CU as shown in Fig. 5. The lower 6 bits identify the PE column within a given array. The next 2 bits indicate the array number, and the remaining higher-order bits give the row value. The row address bits actually transmitted to the PE memories are configuration-dependent and are gated out as shown.

Addresses used by the PE's for local operands contain three components: a fixed address contained in the instruction, a CU index value added from one of the CU accumulators, and a local PE index value added at the PE prior to transmission to its own memory.

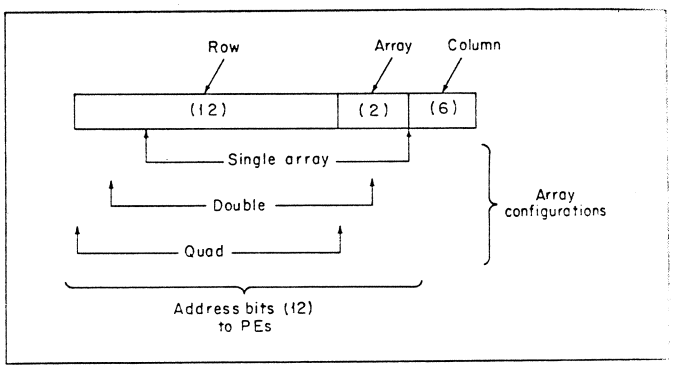


Fig. 5. Memory address structure.

CU data operations

The control unit can fetch either individual words or blocks of 8 words from the array memory to the local data buffer. In addition, it can fetch 1 bit selected from the 8-bit mode register of each processing element to form a 64-bit word read into the CU accumulator. The CU program counter (PCR) and the configuration registers are also directly addressable by the CU. Data manipulations (+, −, Boolean) are performed on a selected CAR and the result returned to the CAR. Data to be broadcast to the processing elements is inserted into the FINQ along with the accompanying instruction and transmitted to the PEs at the appropriate time.

Configuration control

With the variety of array configurations for ILLIAC IV, it is necessary to specify and control the subarrays which are conjoined and to designate the instruction and data addressing. For this purpose each CU has three configuration control registers (CFC), each of 4-bit length, where each bit corresponds to one of the four subarrays. The CFC registers may be set by the B 6500 or a CU instruction.

CFC0 of each CU specifies the array configuration in which it is participating by means of a 1 in the appropriate bits of CFC0. CFC1 specifies the instruction addressing to be used within the array. In a united configuration it is thus possible for the instruction stream to be derived from any subset of the united arrays. CFC2 specifies the CU data addressing form in a manner similar to the CFC1 control of instruction addressing.

The addressing indicated by both CFC1 and CFC2 must be consistent with the actual configuration designated by CFC0, else a configuration interrupt is triggered.

Trap processing

Because external demands on the arrays will be preprocessed through the B 6500 system computer, the interrupt system for the control units is relatively straightforward. Interrupts are provided to handle B 6500 control signals and a variety of CU or array faults (undefined instructions, instruction parity error, improper configuration control instruction, etc.). Arithmetic overflow and underflow in any of the processing elements is detected and produces a trap.

The strategy of response to an interrupt is an effective FORK to a single-array configuration. Each CU saves its own status word automatically and independently of other CU's with which it may previously have been configured.

Hardware implementation consists of a base interrupt address register (BIAR) which is dedicated as a pointer to array storage into which status information will be transferred. Upon receipt of an interrupt, the contents of the program counter and other status information and the contents of CAR 0 are stored in the block pointed to by the BIAR. In addition, CAR 0 is set to contain the block address used by BIAR so that subsequent register saving may be programmed. Interrupt returns are accomplished through a special instruction which reloads the previous status word and CAR 0 and clears the interrupt.

Interrupts are enabled through a mask word in a special register. The interrupt state is general and not unique to a specific trigger or trap. During the interrupt processing, no subsequent interrupts are responded to, although their presence is flagged in the interrupt state word.

The high degree of overlap in the control unit precludes an immediate response to an interrupt during the instruction which generates an arithmetic fault in some processing element. To alleviate this it is possible under program control to force nonoverlapped instruction execution permitting access to definite fault information.

Processing element (PE)

The processing element, shown in Fig. 6, executes the data computations and local indexing for operand fetches. It contains the following elements.

1 Four 64-bit registers (A, B, R, S) to hold operands and results. A serves as the accumulator, B as the operand register, R as the multiplicand and data routing register, and S as a general storage register.
2 An adder/multiplier (MSG, PAT, CPA), a logic unit (LOG), and a barrel switch (BSW) for arithmetic, Boolean, and shifting functions, respectively.
3 A 16-bit index register (RGX) and adder (ADA) for memory address modification and control.
4 An 8-bit mode register (RGM) to hold the results of tests and the PE ENABLE/DISABLE state information.

As described earlier, the PEs may be partitioned into subprocessors of word lengths of 64, 2×32, or 8×8 bits. Figure 7 shows the data representations available. Exponents are biased and relative to base 2. Table 1 indicates the arithmetic and logical operations available for the three operand precisions.

PE mode control

Two bits of the mode register (RGM) control the enabling or disabling of all instructions; one of these is active only in the 32-bit precision mode and controls instruction execution on the second operand. Two other bits of RGM are set whenever an arithmetic fault (overflow, underflow) occurs in the PE. The fault bits of all PEs are continuously monitored by the CU to detect a fault condition and initiate a CU trap.

Data paths

Each PE has a 64-bit wide routing path to 4 of its neighbors (± 1, ± 8). To minimize the physical distances involved in such routing, the PEs are grouped 8 to a cabinet (PUC) in the pattern shown in Fig. 8. Routing by distance ± 8 occurs interior to a PUC; routing by distance ± 1 requires no more than 2 intercabinet distances.

CU data and instruction fetches require blocks of 8 words, which are accessed in parallel, 1 word per PUC, into a CU buffer (CUB) 512-bit wide, distributed among the PUCs, 1 word per

Table 1 PE data operations

		Operation time per element	
Operation	*64 bit*	*2 × 32 bit*	*8 × 8 bit*
+, −	200 ns	240 ns	80 ns
×	400 ns	400 ns	
÷	2200 ns	3040 ns	
Boolean	80 ns		
Shift	80/240 ns†	160 ns	

† (Single length)/(double length)

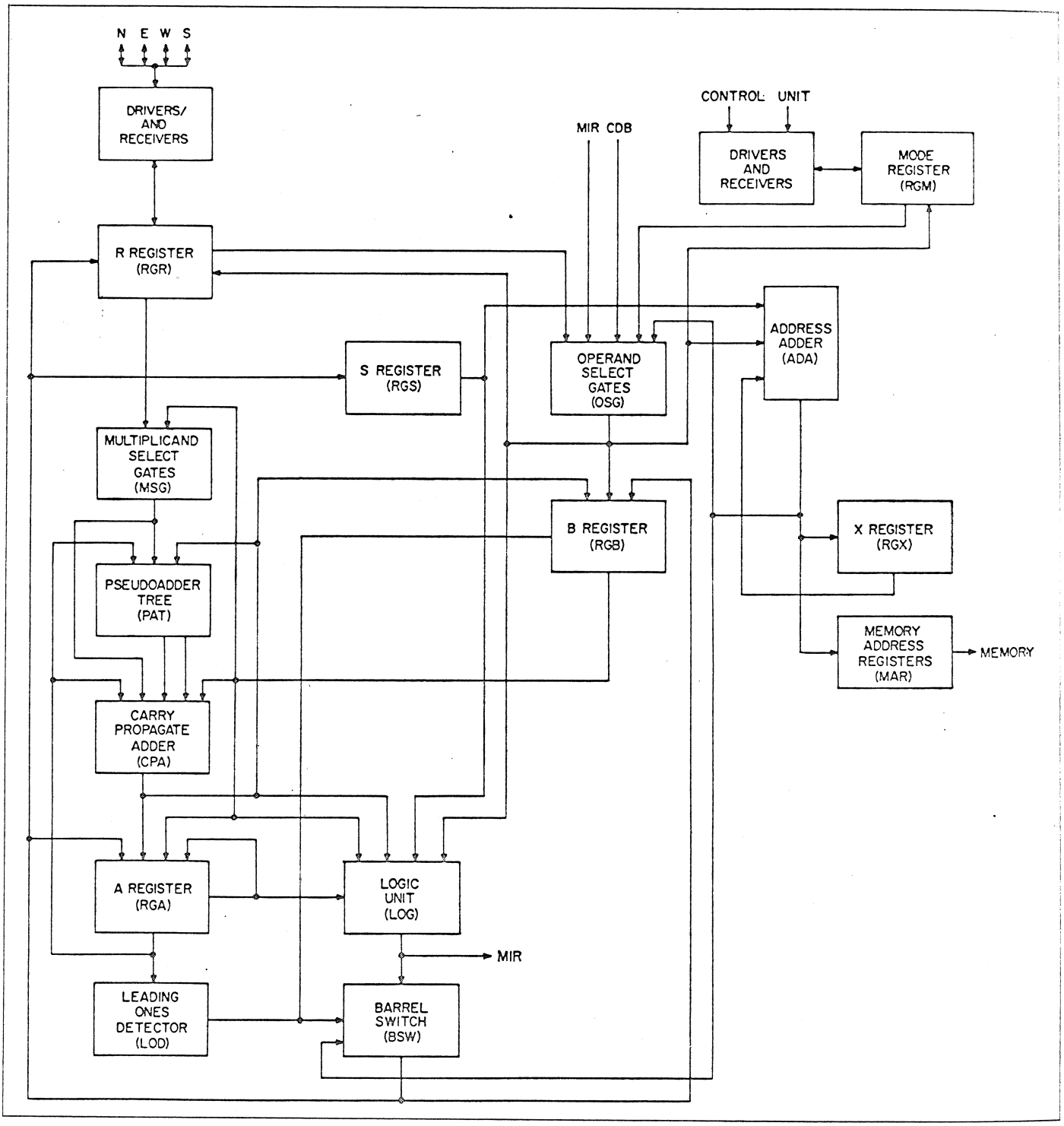


Fig. 6. Processing-element block diagram.

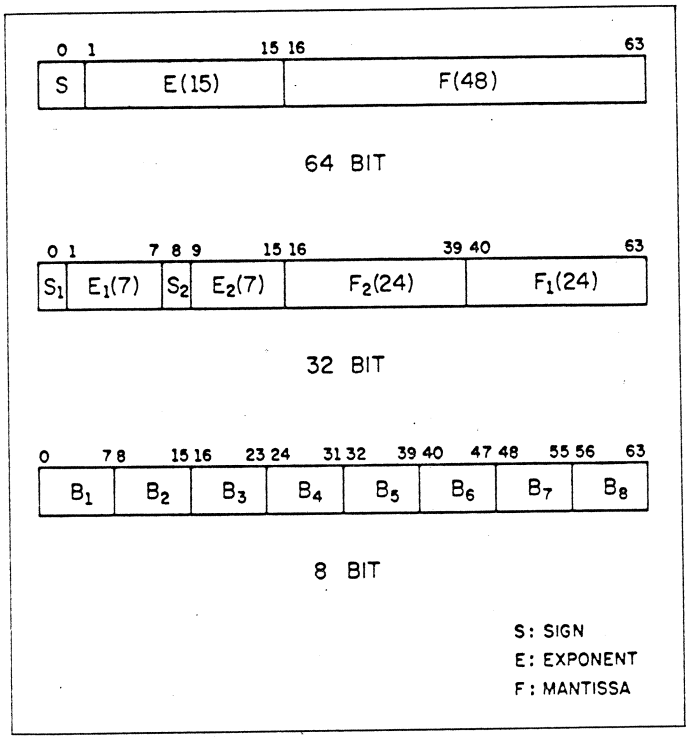


Fig. 7. ILLIAC IV data representation.

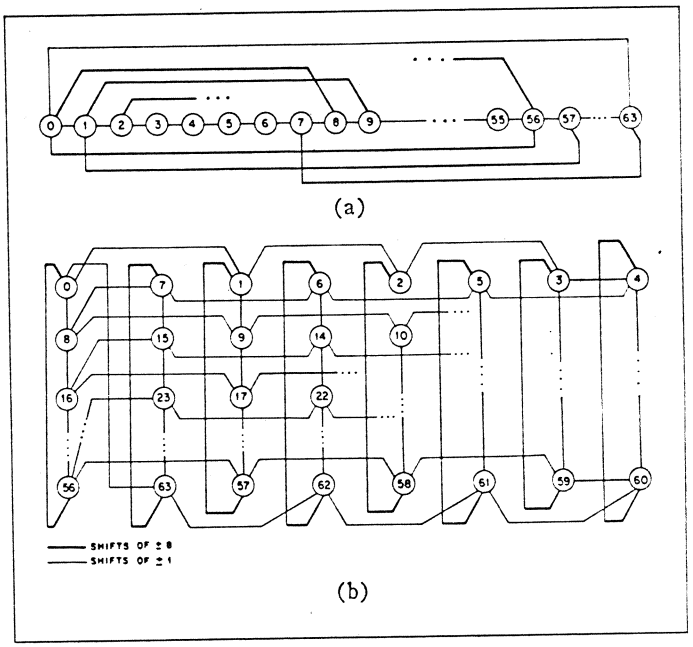


Fig. 8. (*a*) Electrical connectivity for routing. (*b*) Physical layout.

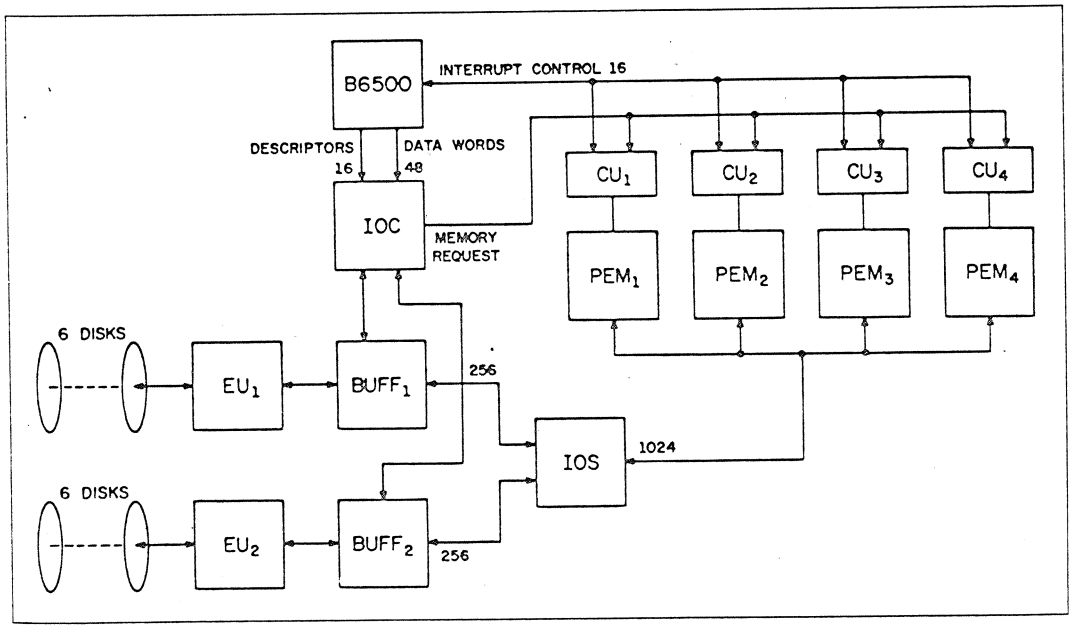


Fig. 9. I/O data path.

cabinet. Data is transmitted to the CU from the CUB on a 512-line bus.

Disk and on-line I/O data are transmitted on a 1024-line bus which can be switched among the arrays. Within each array, parallel connection is made to a selected 16 of 64 PEs, 2 per PUC. Maximum data rate is one I/O transaction per microsecond or 10^9 bits per second. The I/O path of 1024 lines is expandable to 4096 lines if required.

Processing element memory (PEM)

The individual memory attached to each processing element is a thin-film DRO linear select memory with a cycle time of 240 ns and access time of 120 ns. Each has a capacity of 2048 64-bit words. The memory is independently accessible by its attached PE, the CU, or I/O connections.

Disk-file subsystem

The computing speed and memory of the ILLIAC IV arrays require a substantial secondary storage for program and data files as well as backup memory for programs whose data sets exceed fast memory capacity. The disk-file subsystem consists of six Burroughs model IIA storage units, each with a capacity of 1.61×10^8 bits and a maximum latency of 40 ms. The system is dual; each half has a capacity of 5×10^8 bits and independent electronics capable of supporting a transfer rate of 500 megabits per second. The data path from each of the disk subsystems becomes 1024 bits wide at its interface with the array. Figure 9 shows the organization of the disk-file system.

B 6500 control computer

The B 6500 computer is assigned the following functions.

1. Executive control of the execution of array programs
2. Control of the multiple-array configuration operations
3. Supervision of the internal I/O processes (disk to arrays, etc.)
4. External I/O processing and supervision
5. Processing and supervision of the files on the disk file subsystem
6. Independent data processing, including compilation of ILLIAC IV programs

To control the array operations, there is a single interrupt line and a 16-bit data path both ways between the B 6500 and each of the control units. In addition, the B 6500 has a control and data path to the I/O controller (IOC) which supervises the disk, and also direct connections to the array memories.

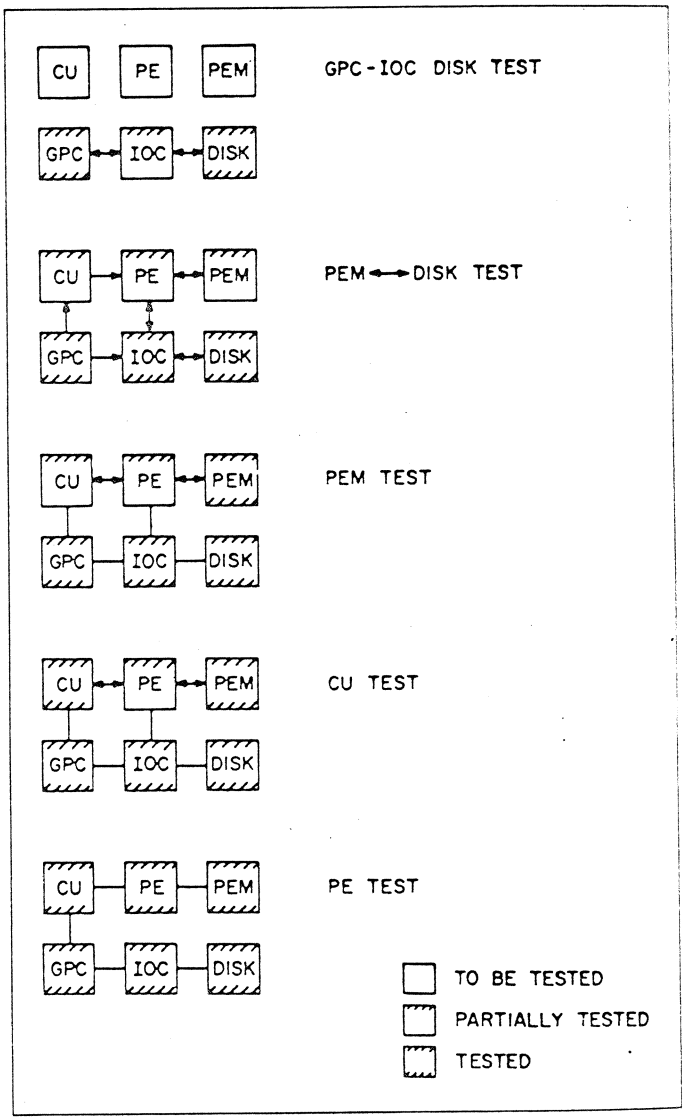


Fig. 10. System diagnostic sequence.

Reliability and maintenance of the ILLIAC IV

The progress in computer components from vacuum tubes to semiconductors over several generations has improved the mean-time-between-failures for computers from tens of hours to several thousand hours. By using larger scale integration, a tenfold increase

in number of gates per system should be possible with comparable reliability.

It is only by virtue of high-density integration (50- to 100-gate package) that the design of a three-million-gate system can be contemplated. Reliability of the major part of the system, 256 processing elements and 256 memory units, is expected to be in the range of 10^5 hours per element and 2×10^3 hours per memory unit.

The organization of the ILLIAC IV as a collection of identical units simplifies its maintenance problems. The processing elements, the memories, and some part of power supplies are designed to be pluggable and replaceable to reduce system down time and improve system availability.

The remaining problems are (1) location of the faulty subsystem, and (2) location of the faulty package in the subsystem.

Location of the faulty subsystem assumes the B 6500 to be fault-free, since this can be determined by using the standard B 6500 maintenance routines. The steps to follow are shown in Fig. 10.

The B 6500 tests the control units (CU) which in turn test all PEs. PEMs are tested through the disk channel. This capability for functional partitioning of the subsystems simplifies the diagnostic procedure considerably.

References

HollJ59; KuckD68; MurtJ66; SlotD62; UngeS58

electronic hardware. But if we restrict ourselves, for instance, to space curves defined by the intersection between a second-degree surface and a plane, the situation is quite different. The case where both f and g are planes has been illustrated above and is seen to be extremely simple.

IV. Conclusions

The problem of incremental stepping, also known as interpolation, is rarely treated in the literature. We have in this paper found it suitable to make a clear distinction between parametric and nonparametric curve representation. The parametric representation leads to the well known DDA technique. The nonparametric representation may at first glance seem to be less straightforward. However, as has been shown in Section III, the method of using the signs of $f(x, y)$, $\partial f/\partial x$, and $\partial f/\partial y$ for prediction of the steps makes the nonparametric representation fully competitive. It is not probable that one method will outdo the other one in all applications. Nevertheless, it is natural that we now sum up advantages and disadvantages for both cases.

Parametric curves can be handled with ordinary DDA technique and are easily extended to three or more coordinates. Keeping constant speed demands special arrangements. Curve degradation is a latent threat which may be hard to predict in some applications.

Nonparametric curves require a modified DDA technique which seems comparable in cost to the parametric case provided the curves have the same complexity. Speed control is no problem since it is inherent in the construction. Curve degradation can easily be exterminated for second-degree curves. Extension to three coordinates is possible but becomes very expensive for higher degree curves.

References

[1] T. R. H. Sizer, Ed., *The Digital Differential Analyser*. London: Chapman and Hall, 1968.
[2] P. E. Danielsson, "Converting a curve to right-angled increments," *Nord. Tidsk. Informationsbehandling* (*BIT*), vol. 3, pp. 213–221, 1963.

A Preprocessing High-Speed Memory System

DAVID J. KUCK, MEMBER, IEEE

Abstract—The fastest parallel and pipeline computers presently being designed use interleaved memory systems with more than 16 individual memory units. A common difficulty in these machines is the alignment of data before it is arithmetically processed. Usually the arithmetic unit is used to preprocess the data. This may increase the computation time by a factor of two or more. This paper proposes a programmed memory preprocessing system which aligns the data before it is passed to the arithmetic processor.

Three problem areas are presented and an example of each of these is solved using the system. Sparse matrix multiplication and a table lookup problem are discussed in detail to explain the operation of the system.

Index Terms—Array processing, data prefetching, interleaved memories, parallel processing, pipeline processing, sparse matrix operations, table lookup.

I. Introduction

THIS paper discusses a memory system organization for use with high-speed arithmetic processors. About ten years ago, computer control units which were highly pipelined (several instructions in process at once) began to appear (e.g., STRETCH, ILLIAC II). More recently, arithmetic units have become pipelined (e.g., CDC 7600, IBM 360-91). The memory system presented here can be regarded as an operand fetch pipeline operation in series with the arithmetic processing unit. The notions of prefetching data (e.g., IBM 360-85 [8]) and indirect addressing are also old ideas. But in the context of highly parallel interleaved memory systems (say, 16 or more individual random access units), these ideas need some expansion. Highly parallel random access memory systems will evidently become more common as processor speeds increase. They are now designed into the two fastest computers under construction: ILLIAC IV, a parallel array processor, and CDC STAR, a serial (or pipeline) array processor.

In these machines quite simple connections are provided between the memory and processor. ILLIAC IV [2] depends entirely on the processing element routing logic to realign data from its stored locations. Pipeline systems typically rely on a prepass through the data, although no descriptions have yet appeared in the literature. Various machines have been described in the literature which have a complex connection between memory and processor systems. Such papers as [6] and [7] mention the interconnection require-

Manuscript received July 10, 1969; revised March 5, 1970.
The author is with the Department of Computer Science, University of Illinois, Urbana, Ill. 61801.

ments but only describe the details superficially. Our system is relatively simple and depends in part on a systematic way of storing the elements of sparse arrays in the memory. Much less logic is required than in systems such as [3] which do more processing in the memory.

The main objective of this memory system is to provide operands to the particular memory unit from which they are to be processed (and presumably, into which the result is to be stored). By requesting the data somewhat before it is to be processed, the relatively small amount of logic provided is able to move data to appropriate places before it is needed. This relieves the computation unit of such overhead processing. The system uses various kinds of identifiers for the words which are being routed. We refer to these identifiers as "tags" and we shall discuss tag registers and tag logic. It is important to realize that various kinds of tags will be explained and used in the three problems which we shall discuss. In the first problem, the tags are simply addresses of data items and are used to indirectly address the data items in a particular memory unit. In the third problem the tags are also addresses of data items, but these data items may be stored in a memory unit some distance from where the tags are originated. Hence, we must provide a scheme for routing tags to the appropriate memory units and returning the appropriate data. The second problem also requires the routing of data from one memory unit to another. In this case, however, the tags are not explicit memory addresses. Rather they are single bits which mark the zero and nonzero elements in an array. Logic is provided which operates on these tag bits to set up the correct routing of data.

Now we present three problems that are rather messy to solve using a high-speed machine with a highly parallel memory. Following these, the memory system is presented. Subsequently, we show how the system can be used in the solution of these problems.

Problem 1: Indirect Addressing—In the solution of a number of problems, it becomes necessary to partition a dense array of data into several sparse arrays and apply a separate algorithm to each one. The data portion of Fig. 1 (c.f., discussion of Section III) shows an array with three types of data. In current machines such computations are carried out by storing some kind of tag bits to identify the data types, and applying the algorithm corresponding to the tag bits for each datum. For example, tables containing only the addresses of data of one type may be used, in which case the tags are addresses of the data. It is important to understand an additional constraint in either a high-speed pipeline or parallel machine with an interleaved memory. To maintain high processing speed, it is necessary to arrange matters so that vectors of data of the same type are fetched from each memory unit because either machine is slowed down unless each instruction is applied to many data elements. By being careful where the data is stored and keeping an array of tags or by making a prepass through the data, this problem could be solved with standard memories. However, this requires some arithmetic processor time. We shall offer a solution which indirectly addresses one data

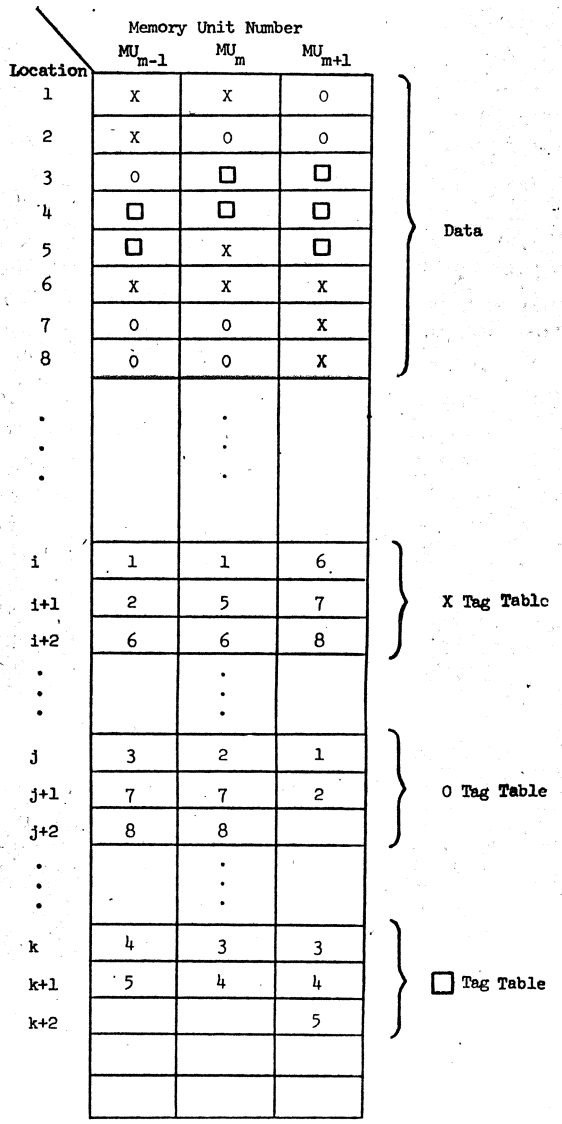


Fig. 1. Indirect addressing example.

element of each type through the tag tables as shown in Fig. 1. This allows sequences of data elements of a single type to be prefetched for processing by a serial or parallel array processor.

Problem 2: Sparse Matrix Multiplication—Many real world matrices are sparse, either randomly or as an array of dense submatrices. The processing of random ones is a particularly messy problem, even on present machines. The arrays must be compressed by rows or columns and identification tags must be matched before any matrix processing can be done. For example, one tag bit may be stored for each element in the array: 0 if the element is zero, and 1 if the element is nonzero. On a parallel or pipeline machine these difficulties are compounded. As an example, a good deal of high-speed computation time is spent on the solution of linear programming problems which are formulated using rather sparse matrices. In the solution of such problems, operations similar to matrix multiplication are commonly used. In Section IV we shall show how stored tags can be preprocessed in the memory system to provide the processor with just nonzero pairs of operands for processing.

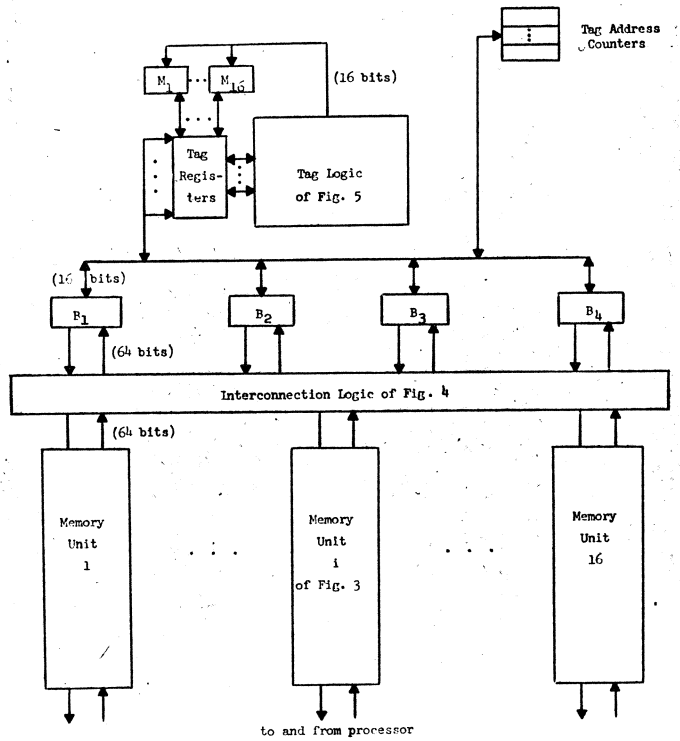


Fig. 2. Memory system.

Problem 3: Table Lookup—We define a table lookup problem to be one in which during the course of a computation a number is generated which serves as an address in a table stored in memory. The computation may require a store or a fetch at this location. As one example, many of the highest speed computers are used to solve physics problems whose equations of state solutions require table lookups. These tables can contain many thousands of entries. As another example, almost all computers are required to do compilation. This process always generates a number of tables, each of which may contain hundreds of entries. It is necessary that each identifier have a unique entry in a table. In a parallel array computer, this can be a major difficulty since each processor must access the entire memory. In Section V we show how global addresses can be routed to the correct memory unit and the fetched contents can be returned to the processor which generated the request.

II. System Organization

In this section we shall give an overview of the hardware organization proposed to solve the problems mentioned in Section I. Some points about the software and use of the system will be covered briefly. The three subsequent sections are intended to give a detailed explanation of the system's operation and effectiveness in solving these problems.

Fig. 2 shows the overall memory system and Figs. 3–5 show some of its details. A 16-memory unit system with 64-bit words is described, but these are arbitrary numbers. Fig. 2 shows the 16 memory units (details in Fig. 3), the interconnection logic (details in Fig. 4), four broadcast registers (B_i), and some tag logic and registers. The tag registers, tag logic, and tag address counters of Figs. 2 and 5 may be regarded as the control logic of the processes performed in the memory system. The circled elements in the figures are gates, and the rectangular elements are registers. The sequencing of this control logic, as well as the logic within the memory units shown in Fig. 3, is assumed to be done from the same control unit which sequences the arithmetic processor or processors associated with the memory system. This is not necessary and it would be in the spirit of this paper to have the memory system sequencer in the control part of Fig. 2. We choose to leave it in the standard control unit in order to avoid discussing its details and its interface with the processor's sequencer. Several details have been left unspecified, partly because they depend on more detailed hardware design, and partly because they should be influenced by the applications for which a particular machine will be used.

If there are n memory units (MU's), then we require $\sqrt{n}$ B_i registers. These are used to broadcast a word to each MU in a row as illustrated in Fig. 4. There, the S_i gates select an operand from some MU in each row. It is transmitted to the B_i broadcast register and from there the M_i gates determine which MU's in the row receive the operand. It is also possible to transmit from the B_i to the tag registers. We shall discuss these in terms of Fig. 4 later. Briefly, they are used to hold operands for the tag logic which determines how words are transmitted. Thus the T_A and T_B of Fig. 5 set the M_i gates of Figs. 3–5. We also assume that these results can be held in 16 M_i registers of one bit each as shown in Fig. 2. Within each memory unit (Fig. 3) there should be at least two index registers (XR). There should be i_A and i_B registers (64 bits) and most conveniently two other 64-bit memory information registers (MIR). More MIR's would be a luxury, and fewer would be tolerable since the memory itself could be used as backup (slowing the system down). These registers

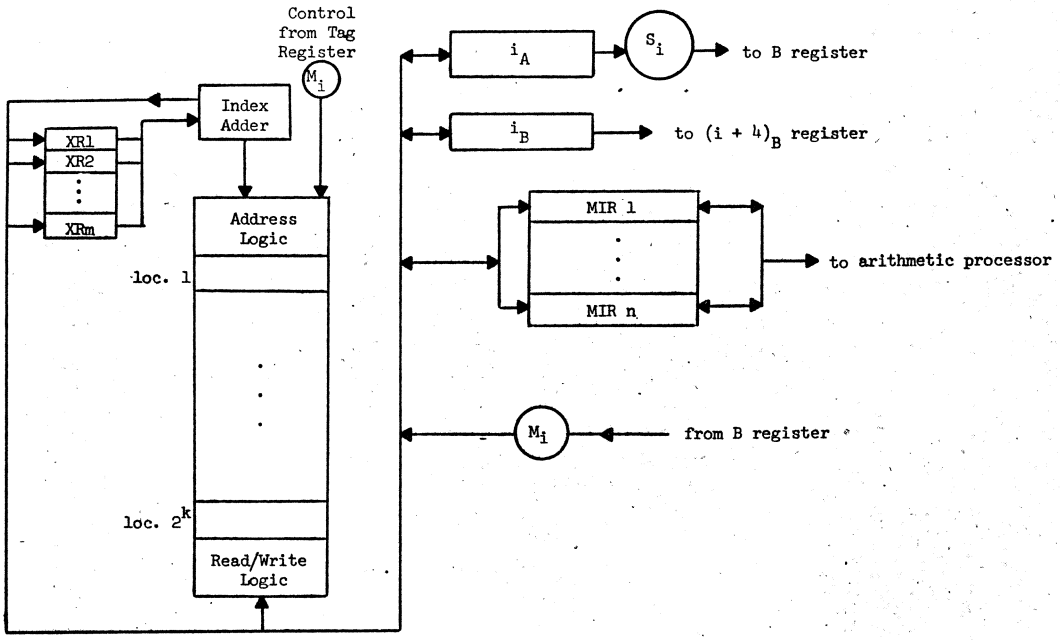


Fig. 3. *i*th memory unit.

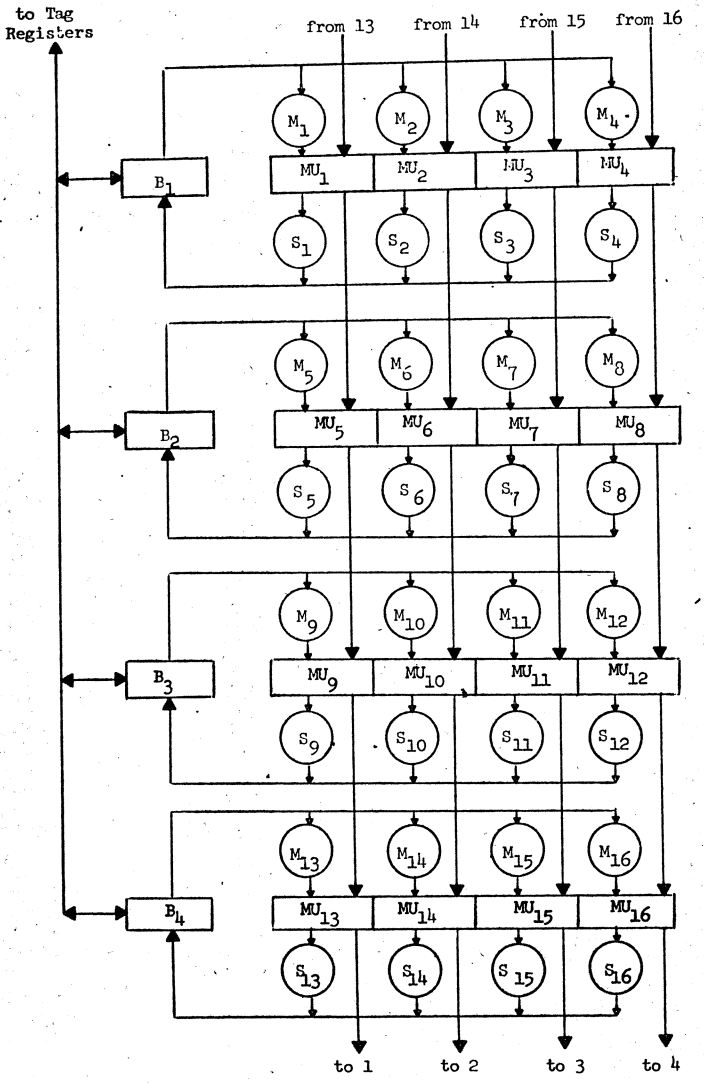


Fig. 4. Memory interconnection scheme.

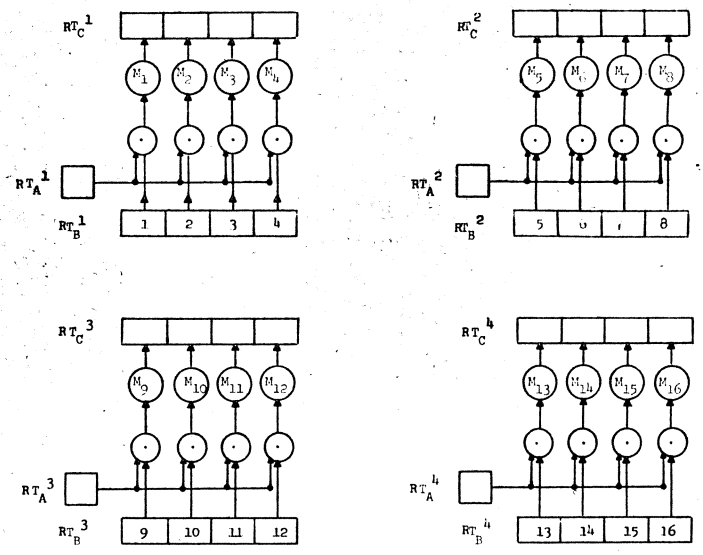


Fig. 5. Tag logic detail.

are shown connected by a bus, but direct gating of i_A, i_B, and MIR to and from memory, i_B to an MIR, and M_i to an MIR are all that would be necessary. Note that for an n memory unit system, while the number of parts in a crossbar switch grows as n^2, the number in this system only grows as n. In order to explain the functioning of the memory, we shall now sketch some uses of this memory for solving the problems of Section I.

As we shall see in Section III, Problem 1 of Section I may be solved by using the index registers to point to tables of indirect addresses. Each of these tables contains only addresses of all data elements of one type. We refer to these addresses as tags for the data elements. The memory information registers may be used to hold prefetched, indirectly addressed data which is of one type and which may be pro-

cessed by the same algorithm. In sparse matrix multiplication, Problem 2, and table lookup, Problem 3, the B registers, the tag registers, and logic, together with the M and S gates are used to route each word to the memory unit from which it will be processed. In the case of a parallel array machine, this also may be the memory unit into which the result of processing will be stored. In sparse matrix multiplication, the tag bits mark the nonzero data elements, and we route the matrix elements appropriately. In table lookup, the tags are used to refer explicitly to the destination memory unit for local addresses generated elsewhere in the array.

The memory system should operate under stored program control. As we shall see later, various sequences of operations are desirable, depending on the kind of problem being solved. Since there is relatively little logic in the memory system, a standard set of instructions could be defined. On the other hand, since such a memory system might find a large number of specialized uses, this would seem to be an ideal place to use a microprogrammed memory controller. The instructions which control the memory system are to be executed simultaneously with the processor instructions —much like data channel commands. As we shall see, the memory system instructions would be compiled in the normal instruction stream and would be dispatched to the memory control unit sequencer by a control unit lookahead station. This is commonly done in machines with a pipeline control unit. If the memory system were designed to be completely autonomous, perhaps with its own clock, we could imagine instructions being fetched from memory and executed entirely within the memory system. This system could be described as a multiprocessor system consisting of an arithmetic processor and a memory processor, and would require some kind of intercommunication between the two processors from time to time. The arithmetic processor would, of course, fetch from the memory in the standard way.

The programming languages currently being implemented for high-speed computers are tending to include more statements which explicitly generate and operate on index sets. For example, in APL [4] and TRANQUIL [5] a vector of integers which controls the indexing through an array may be generated at compile time or execution time. The compilation of instructions which implement these index set statements in an arithmetic processor is discussed in [1]. Whether an index set is generated by the compiler or during execution by the processor, it could be passed to the memory system to control its sequence of operations. It is neither the purpose of this paper to discuss the details of programming the proposed memory system nor a compiler for it; however, we shall outline some software aspects of the three problems here and give more details in later sections.

In the indirect addressing problem, the generation of initial indirect address tags may be done by the compiler, but generally they may be changed at execution time. The compiler must provide for the manipulation of pointers to the tag tables. In the sparse matrix multiplication example, the tags denote the zero and nonzero elements of the matrices. If the matrices are known before execution of the program, then the tags may be prepared ahead of time. If not, the tags may be generated during execution by the processor associated with this memory system. It will be shown in Section IV that if proper tags are provided for operand matrices, then the hardware provided will generate tags for the result matrix.

In the case of the table lookup problem, the proposed memory system really removes some programming problems rather than introducing any. The system hardware simply allows the processor associated with one memory unit to generate addresses for any other memory unit. Each address is transmitted to the appropriate memory unit and the accessed data is returned to the unit which requested it.

III. Indirect Addressing

Problem 1 may be easily solved using this memory system. Consider the example of Fig. 1 which shows three memory units each containing three types of data elements denoted by X, 0, and $\square$. Each type of element requires processing by a different algorithm, but the processing also requires the values of adjacent elements in the array, independently of their type. Thus, the data are stored in a way which represents their physical configuration. This is a common requirement in the solution of partial differential equations.

Assume that by indirectly addressing the data through one tag table, a data element of one type is fetched from each memory unit, and that after processing, some data elements may have changed their types. The new data elements are stored in the same memory locations from which their predecessors were fetched. However, the tag tables are modified in each memory to hold a pointer from the appropriate tag table to the newly stored data element. Thus, when a subsequent processing step occurs, a row of elements of the same type can again be accessed by indirectly addressing the array via the tag tables. The convenience of having several index registers to point to several tag tables and the data array should be clear. Depending on the machine's word length, the size of the data array, and the size of memory, several tags may be packed in each memory location to decrease the burden of the extra memory locations required for the tag tables.

In a number of applications, each data element is in fact a vector of several memory words. In a partial differential equation problem, each mesh point may have up to ten variables associated with it. Let us say there are k related elements stored for each point in physical space. Thus, if we are using n memory locations per memory unit for data, there are just n/k independent sets of data (vectors) which need to be referenced through the tag tables. If the memory has a word length of p bits, then the number of words required to store tags is just $n/pk \log_2 n/k$, assuming they may be packed across word boundaries. If $n=2^{10}$, $p=2^6$, and $k=2^2$, then the ratio of the number of words required for

tags to the number of words required for data is $1/pk \log_2 n/k = \frac{1}{32}$, which is an overhead of about 3 percent.

The compiler must provide for pointer manipulation in this problem. Since we generally cannot accurately predict how many data words of each type there will be at any step of the calculation, we might assume some number n_i for data of type i such that $\sum n_i = n$, the total amount of data. Assume we reserve $\alpha n_i/kp \log_2 n_i/k$, $\alpha \geq 1$, words of memory for tags of type i data. This wastes some space (e.g., 3α percent for the example above), but gains us two things. First, it allows for errors in the estimates. Also, it allows a relatively simple solution to the pointer manipulation problem mentioned above.

At some step of the calculation, assume there are $t_i = n_i/kp \log_2 n_i/k$ tag words for data of type i and assume that we have αt_i locations set aside for type i tags. Finally, assume we have three pointers: pointer one to the first location in the t_i tag table, say β; pointer two to the last location occupied, $\beta + t_i - 1$; and pointer three to the last location reserved for t_i tags, $\beta + \alpha t_i - 1$. When used, these pointers are in index registers, so we could handily use three index registers here. Otherwise, we could store pointers in memory until needed.

Now let us consider the processing of the type i data. We access the first element through pointer two. The first data element may be a vector as discussed earlier, but all components are accessed relative to the first element in the vector, so just one pointer is required. Also recall that several index values may be packed into a word so the first pointer two value may be used for several computational steps. When the first value of pointer two is exhausted, we decrement pointer two and test the new value against pointer one. This provides a limit for terminating the processing of type i data. When we are ready to store newly computed type i data, a tag is loaded into the memory word indicated by pointer three. When the first pointer three word is packed full of tags, we decrement pointer three.

Recall that as a result of processing type i data, data of some other type may be generated. Thus, the computed data must be tested against whatever criteria are used to determine its type. This testing would most likely be done in the processor but if the criteria were simple enough, memory unit logic could be used. As data of other types is processed, more type i data may be generated, necessitating further decrementing of pointer three. These pointer three values must be tested against the original top of the type i tag table, i.e., the values of pointers one and two after one step for type i data processing was finished. On the next processing step for type i data, an analogous sequence of events will occur, with the pointers being incremented instead of decremented. On the step following this, the process described above will be repeated. In case some tag region overflows, another part of memory will be required and we may sequence through it just as in the above discussion.

IV. Sparse Matrix Multiplication

Consider the matrix multiplication problem $C = A * B$, where A, B, C are 8×8 matrices sequenced according to Fig. 6 and stored as shown in Fig. 7(a). The circled elements

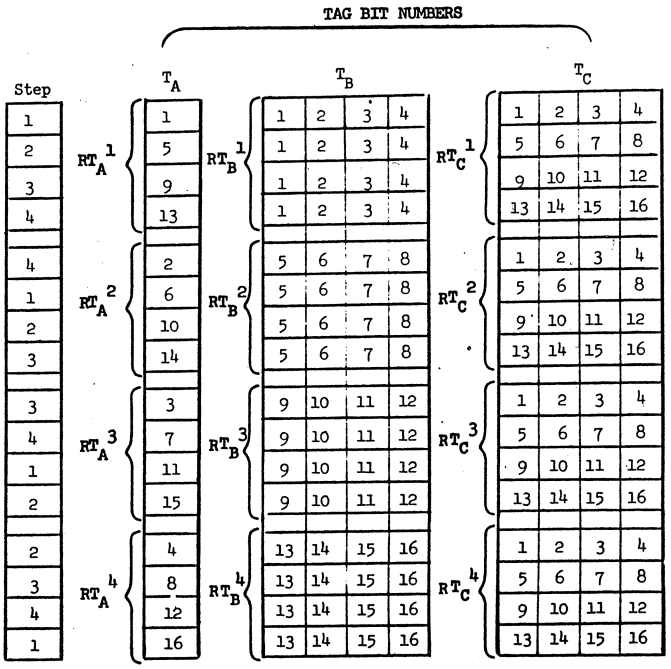


Fig. 6. Matrix multiply memory control sequence.

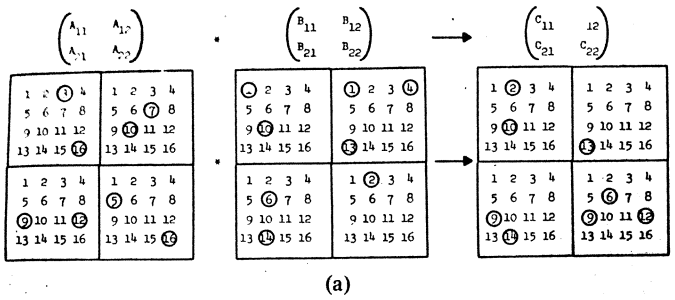
are assumed to be nonzero valued. We will show how the product C is formed using Figs. 2 through 7. In order to solve this problem, the definition

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

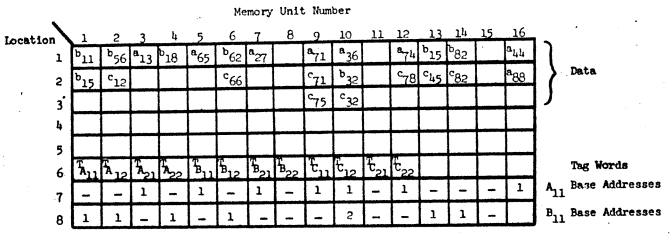
requires for each element of C, the corresponding row elements of A and column elements of B. If A and B are sparse, we need only consider nonzero (a_{ik}, b_{kj}) pairs. On each step, when a nonzero pair occurs, the scheme to be presented broadcasts an element from each row and column of A into appropriate elements of B. The sequences of operations described below may be regarded as standard memory system instructions executed for this particular problem.

We begin by discussing the multiplication of the partitions A_{11} and B_{11} as shown in Fig. 7(a). In Fig. 7(a) each of the three matrices is represented by four partitions. The square arrays of integers $1, \cdots, 16$ represent the numbers of the memory units in which the corresponding matrix elements are stored. The circled elements are nonzero valued. Thus, the circled 3 in partition A_{11} corresponds to a_{13} in memory location 1 of memory unit 3 in Fig. 7(b), and the circled 1 in partition B_{12} of Fig. 7(a) corresponds to b_{15} in location 2 of memory unit 1 of Fig. 7(b). Using a 16-bit tag word for each partition, Fig. 7(b) shows four tag words per matrix in location 6 (several of these could, of course, be packed into one memory location).

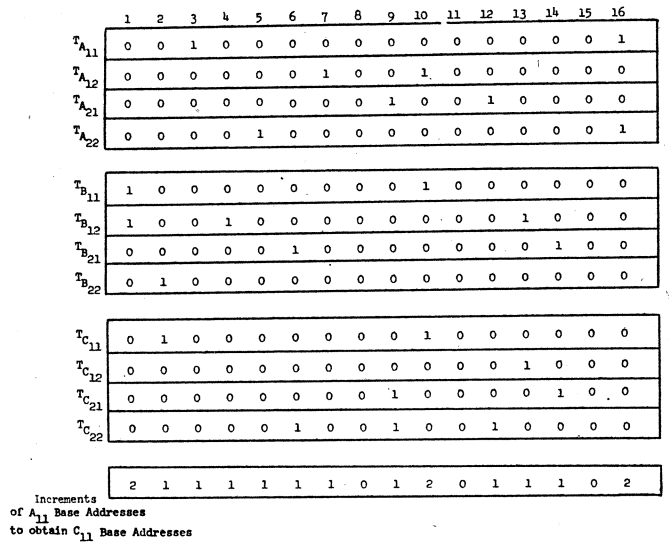
Fig. 7(c) shows the bit patterns stored in each of the tag words. Thus, $T_{A_{11}}$ has ones in bits 3 and 16 corresponding to the circled elements of A_{11} in Fig. 7(a). To begin the multiplication of A_{11} and B_{11}, first $T_{A_{11}}$ and $T_{B_{11}}$ are fetched from memory and by the appropriate S_i, gated through a B register to the tag registers shown in Fig. 2. These two tag words indicate which elements of their respective partitions are zero or nonzero. Next we must prepare to access the data of the A and B matrices by loading an index register in



(a)



(b)



(c)

Fig. 7. Matrix multiplication example. (a) Sparse matrices. (b) Memory layout. (c) Tag values.

each memory unit with the base addresses of A and B, respectively. These base addresses are shown stored in locations 7 and 8 of Fig. 7(b). Next the i_A registers are loaded with all the nonzero elements of the A matrix partition under consideration and the i_B registers are similarly loaded with nonzero B matrix elements. If some partition has an all zero tag word, it is skipped.

Now the appropriate data movement to align nonzero pairs (a_{ik}, b_{kj}) is carried out according to the four steps shown at the left of Fig. 6. The numbers in the T_A, T_B, and T_C columns of Fig. 6 identify the particular tag bits corresponding to elements of A, B, and C as shown in Fig. 7(c). The logic is shown in Fig. 5. Thus, as indicated by Fig. 6, on Step 1, RT_A^1 of Fig. 5 will be loaded with bit 1 of an A matrix tag, e.g., bit 1 of $T_{A_{11}}$ of Fig. 7(c); similarly, RT_A^2, RT_A^3, and RT_A^4 will be loaded with bits 6, 11, and 16, respectively. Also, as indicated by Fig. 6, on Step 1, RT_B^1 will be loaded with bits (1, 2, 3, 4) of a B matrix tag, e.g., $T_{B_{11}}$ of Fig. 7(c); similarly, bits (5, 6, 7, 8) go to RT_B^2, bits (9, 10, 11, 12) go to RT_B^3, and bits (13, 14, 15, 16) go to RT_B^4.

Step 1 (left column, Fig. 6) is executed by activating S_1, S_6, S_{11}, and S_{16}, thereby loading B_1, B_2, B_3, and B_4 with A matrix elements from memories 1, 6, 11, and 16, respectively. At the same time, the RT_A registers of Fig. 5 are loaded with the A matrix tag bits from memories 1, 6, 11, and 16. Thus, $RT_A^1=0$, $RT_A^2=0$, $RT_A^3=0$, and $RT_A^4=1$; while $RT_B^1=(1000)$, $RT_B^2=(0000)$, $RT_B^3=(0100)$, and $RT_B^4=(0000)$. The AND of the A_{11} and B_{11} matrix tag bits as shown in Fig. 5 activates the appropriate mode gating $M_1, \cdots, M_{16}$ and the B register values are broadcast to the activated memories and stored in an MIR. In Step 1 the results are all zero, so nothing happens.

Now Step 2 of Fig. 6 is performed by setting $RT_A^1=0$, $RT_A^2=0$, $RT_A^3=0$, and $RT_A^4=0$, and shifting the B matrix rows down one row using the alternate paths of Fig. 4. Thus, data from memory units 1, 2, 3, and 4 go to 5, 6, 7, and 8 and so on, with 13, 14, 15, and 16 going to 1, 2, 3, and 4. At this step S_5, S_{10}, S_{15}, and S_4 allow memory units 4, 5, 10, and 15 to load B_1, B_2, B_3, and B_4, respectively. The process continues as above and is followed by Steps 3 and 4. In terms of the example of Fig. 7, nothing happens until Step 3. When $RT_A^1=0$, $RT_A^2=0$, $RT_A^3=1$, and $RT_A^4=0$, while $RT_B^1=(1000)$, $RT_B^2=(0000)$, $RT_B^3=(0100)$, and $RT_B^4=(0000)$. At this time a_{13} is sent to an MIR of memory 2. At this point, b_{32} has been shifted to i_B of memory 2 and is now gated to another MIR of memory 2. The pair can now be multiplied and stored in memory 2. This is controlled by the T_C tag bits which are generated by the logic shown in Fig. 5; in this case $RT_C^3=(0100)$. On Step 4 the AND logic of Fig. 5 generates all zeros, so nothing happens. After these first four steps, we have formed the product $A_{11}*B_{11}$ which in this example resulted in just one product, $a_{13}*b_{32}$.

In Step 3 we generated our only nonzero result. From Fig. 6 we see that on Step 3 this results in a 1 in position 2 of the T_C tag word. The T_C results of each step are ORed together with the previously generated T_C word. Thus, when we finish the product $A_{11}*B_{11}$, the $T_{C_{11}}$ tag word has only a single 1 in position 2. The 1 in position 10 shown in Fig. 7(c) will be generated by the $A_{12}*B_{21}$ operation.

Notice that the above process has taken a time equal to two memory cycles plus some logic roughly equal to another memory cycle. In these three memory cycles, we have moved all corresponding elements of two partitions to the correct position for their storage after processing. In this time estimate we have ignored the tag fetching and index register loading because it is expected that the time for these memory cycles would be distributed over operations on a number of partitions. We shall see below how subsequent index values may be obtained from previous ones. Also, depending on the number of memory units and the word length, several tags may be packed into one memory word.

Before describing operations on subsequent partitions, it is important to notice that a standard sequencing through the partitions must be chosen for storing them. We have chosen to store them by rows from left to right. Thus, given the base addresses for any matrix, we can compute the base

addresses for any subsequently stored partition by simply summing up the nonzero tag bits in each memory for all intervening partitions. For example, in Fig. 7, we show how to generate values by which the A_{11} base addresses must be incremented to obtain base addresses for C_{11}. These are obtained by summing the tags in each memory across all A and B matrix partitions; the index adders or the arithmetic processor could do this. Note that we show only the A_{11} and B_{11} base addresses stored in memory in Fig. 7. This economizes storage locations and is sufficient, given the above simple procedure for obtaining increments for any subsequent partitions.

Using the above storage format, we must generate the C matrix partitions in the order C_{11}, C_{12}, C_{21}, C_{22}. So we next process A_{12} and B_{21} and add the result to the previously generated $A_{11}B_{11}$ product to complete C_{11}. We OR any new $T_{C_{11}}$ elements generated with those generated on the first step. The tag address counters of Fig. 2 are used to index through the tags in the memory. Note that the base addresses for A_{12} may be obtained by simply incrementing the index register values used for A_{11}, and B_{21} may be obtained using the process described above. The processing continues to generate the next three partitions.

Assuming a p bit machine and an $n \times n$ matrix of density d, this scheme requires dn^2p bits of storage for data, plus n^2 bits for tags. This is a total of $n^2(dp+1)$ and if d is quite small, say $1/p$, then the space required for storing tag bits becomes relatively high. Also, for matrices of very low density, most partitions will contain all zeros, and it would be desirable to skip all zero pairs in a quick way. One way to treat both of these difficulties for low density matrices is to store another level of tags (level 2 tags) for entire partitions and only store the tags discussed above (level 1 tags) for partitions with at least one nonzero element.

In terms of Fig. 7, we could add level 2 tag words, such that if a partition were all zeros, its level 2 tag word position would contain a zero. If it had at least one nonzero element, its level 2 tag position would contain a one. Thus, in Fig. 7 we could interpret $T_{A_{11}}$ and $T_{B_{11}}$ as level 2 tag words for 16 partitions. Thus $T_{C_{11}}$ would indicate just the partitions which were candidates for a nonzero product. Continuing the analogy, a_{13} and b_{32} would correspond to the level 1 tags for the partitions which must be multiplied together. Notice that we could select and route these pairs of level 1 tag words together just as we selected and routed data words in our above discussion. Now the pairs of level 1 tag words could be handled as they were in the above discussion. With this scheme, we avoid storing and testing level 1 tag words for all partitions which contain only zero elements. Only a zero in a level 2 tag word is used. The amount of storage required depends on the distribution of nonzero elements. For an m memory unit machine, if k nonzero elements were in each nonzero partition, then the bit requirements for storage would be dn^2p for data, plus dn^2m/k for level 1 tags, plus n^2/m for level 2 tags since we need one bit per partition and m is the partition size.

Thus, the break even point in terms of memory space required for this scheme versus the earlier one is when

$$n^2(dp + 1) = n^2\left(dp + \frac{dm}{k} + \frac{1}{m}\right)$$

or

$$1 = \frac{dm}{k} + \frac{1}{m}.$$

In terms of k we have

$$k = \frac{dm}{1 - (1/m)} \approx dm$$

for a highly phased memory. If k is larger than dm, then the second scheme is better. For example, if $n=10^4$, $d=10^{-4}$, $p=10^2$, $m=10^2$, $k=10$ (density in nonzero blocks averages 0.1), then the former scheme requires about 10^8 bits, while the latter requires about 2×10^6 bits.

V. Table Lookup

This example illustrates how data can be transmitted from any memory unit to any other memory unit using this system. (This is slower than with a crossbar switch, but the system contains many fewer gates.) The example will be discussed in terms of an array of 16 parallel processors, but the method could be used to align operands from several memory banks for a single pipeline processor.

Assume that each of 16 processors needs to access a table which is so large that it must be spread out across the entire memory system. On the basis of computed data, each processor derives an address. Each address may be regarded as four destination tag bits which indicate one of the 16 memory units, concatenated with a local address within that memory unit (say, 16 bits). The problem that must be solved is to transmit the local address to the memory unit indicated by the four destination tag bits, to access that memory unit using the local address, and to return the resulting word (or sequence of words) to the processor (memory unit) in which the address originated. Further, we wish to solve this problem for 16 addresses at once.

Consider the example shown in Fig. 8. The "implicit source" column is a binary numbering of the memory units from 0 to 15. The four destination tag bits of the address generated in each of these units is given in the corresponding "explicit destination" column. In this example, the first processor (implicit source label 0000) generated an address for the sixth memory unit (explicit destination bits 0101). The four-bit source and destination numbers can be regarded as row and column indicators in a 4×4 array as shown in Fig. 8 column headings. These may be interpreted as the rows and columns of memory units as indicated in Fig. 4. Thus, memory units 5, 6, 7, and 8 form row 01 and memory units 2, 6, 10, and 14 form column 01. The decoding of the destination tag bits will be performed in the memory system tag logic of Fig. 2. They may be transmitted there via the B registers by ORing together in the B register a four-bit tag from each element of a row of memory units. Then each B register is transmitted to a different tag register as shown in the row groupings of Fig. 8. The following is a description of the sequence of events which occurs on each of

Implicit Source row		Implicit Source column		Explicit Destination row		Explicit Destination column		
0	0	0	0	0	1	0	1	to Tag Register 00
0	0	0	1	0	1	1	1	
0	0	1	0	0	0	0	0	
0	0	1	1	1	1	0	0	
0	1	0	0	1	0	0	0	to Tag Register 01
0	1	0	1	1	1	1	1	
0	1	1	0	1	1	0	1	
0	1	1	1	0	0	1	0	
1	0	0	0	0	0	0	0	to Tag Register 10
1	0	0	1	1	0	0	1	
1	0	1	0	1	0	1	1	
1	0	1	1	1	0	0	0	
1	1	0	0	0	1	1	0	to Tag Register 11
1	1	0	1	0	0	1	1	
1	1	1	0	0	0	0	1	
1	1	1	1	1	1	1	0	

Fig. 8. Table lookup example.

four steps to distribute the local addresses to the memory units indicated by the destination bits of the respective local addresses.

On the first step, the following functions are performed by the tag logic shown in Fig. 2 on the tag registers which contain the destination tags of Fig. 8. First, the destination tags whose row bits match their tag register number have their column bits recorded. These two-bit binary column numbers are mapped into four-bit destination masks which control the M gates as follows:

$$00 \rightarrow 1000$$
$$01 \rightarrow 0100$$
$$10 \rightarrow 0010$$
$$11 \rightarrow 0001.$$

Similarly, source column masks are generated in the tag logic unit and these control S gates. These correspond to the implicit source column numbers of Fig. 8. For example, in Fig. 8 tag register 00 has its 0000 destination element mapped into a destination mask 1000 and a source mask 0010 is generated from the corresponding 10 implicit source column bits. At this time, one local address per row may be transmitted through the B register to its correct memory unit. In our example, this process must be repeated twice for tag register 10 (with tag registers 00, 01, and 11 idle the second time) because it has two identical destination row addresses to transmit. The third identical one (1001) matches the source address and can be ignored. In the notation of Fig. 4, after this first step we have transmitted addresses from memory unit 3 (source 0010 of Fig. 8) to memory unit 1 (destination 0000 of Fig. 8), from memory unit 11 (source 1010) to memory unit 12 (destination 1011), from memory unit 12 (source 1011) to memory unit 9 (destination 1000), and from memory unit 16 (source 1111) to memory unit 15 (destination 1110).

To prepare for the second step, the tag logic increments the implicit source row binary tags of Fig. 8 which it generated on step one; the local addresses are transmitted from i_B to $(i+4)_B$ (mod 16) in all memory units, and the above process is repeated. In other words, we shift the rows down (end around) and transmit all of the local index values which are to be received by this row. This process is carried out four times (i.e., two more times) and the address transmission is complete. When the local addresses have arrived at their destinations, we load an index register and cycle all the memories at once. The data is gated to the i_A register for transmission back to the source of its address. This transmission is carried out just as the local addresses were distributed, except that the source and destination masks' roles are now reversed. Source masks now control the M gates and destination masks control the S gates.

We shall sketch several variations in detail which could be designed into this process. Since more than one local address may be received by a particular memory unit, some provision must be made either to interrupt the process or to switch the destination gating to 16 new bits after each transmission to a memory unit. This is one point at which microprogrammed control would be very useful because a great variety of table lookup problems exist. We may even wish to transmit several sets of addresses in the above manner before accessing the data. Thus, if we transmitted addresses until each memory unit had at least one address, we could attempt to achieve a balance of work across all memory units.

The time required for one cycle of a table lookup can vary depending on whether we do one step at a time, as in the above example, or comingle several, as mentioned in the previous paragraph. Assume one step is done at a time and there are at most two fetches from the same memory. The overall number of steps to get the local addresses to their destinations is from 4 to 16. Assume an average of 12 such steps at one clock time each, followed by two memory cycles of six clock times each, followed by 12 steps to route the accessed data back to where it belongs. This is a total of 42 clock times or seven memory cycle time units. Thus, for rather pessimistic average conditions, we could have one table lookup cycle performed during the execution of seven rather fast (memory cycle length) instructions. In most calculations, it would not be difficult to mask this amount of time.

VI. Conclusion

We have described the design and operation of a memory system for use with high-speed processors. Since such processors depend on many memory units to achieve a high data rate, the ordering and matching of data can be rather consuming of arithmetic processor time. By building a relatively simple system which is pipelined in series with arithmetic processing, most of this arithmetic processor time can be saved. The use of the memory system requires some

planning of how arrays are mapped into the memory (as in our matrix multiplication example). Some comments about compilation are included.

Acknowledgment

The final form of this paper is due in a large part to the careful reading and comments given by the referees. Thanks are also due to P. Budnik, D. Lawrie, and Y. Muraoka.

References

[1] N. E. Abel *et al.*, "TRANQUIL: a language for an array processing computer," *1969 Spring Joint Computer Conf., AFIPS Proc.*, vol. 34. Montvale, N. J.: AFIPS Press, 1969, pp. 57–73.
[2] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV computer," *IEEE Trans. Computers*, vol. C-17, pp. 746–757, August 1968.
[3] B. A. Crane and J. A. Githens, "Bulk processing in distributed logic memory," *IEEE Trans. Electronic Computers*, vol. EC-14, pp. 186–196, April 1965.
[4] A. D. Falkoff and K. E. Iverson, "The APL/360 terminal system," IBM Watson Research Center, Yorktown Heights, N. Y., Res. Rept. RC-1922, March 1968.
[5] D. J. Kuck, "ILLIAC IV software and application programming," *IEEE Trans. Computers*, vol. C-17, pp. 758–770, August 1968.
[6] J. Schwartz, "Large parallel computers," *J. ACM*, vol. 13, pp. 25–32, January 1966.
[7] D. N. Senzig, "Computer organization for array processing," *1965 Fall Joint Computer Conf., AFIPS Proc.*, vol. 27. Washington, D. C.: Spartan, 1965, pp. 117–128.
[8] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Trans. Electronic Computers* (Short Notes), vol. EC-14, pp. 270–271, April 1965.

A Novel Rotate and Shift Circuit Using Bidirectional Gates

WALTER R. NORDQUIST AND WING N. TOY, MEMBER, IEEE

Abstract—A bidirectional gate is described which would allow information contained in one register to be transferred to another register or vice versa. The rotate/shift functions may be implemented by gating the data between two flip-flop registers through the shifted or skewed gates connecting the two. The unique property of bidirectional transfer of data signals between two registers permits a simplified and interesting design of a rotate and shift circuit. The implementation of this rotate/shift circuit into a data processing configuration is also illustrated.

Index Terms—Bidirectional gating, data bus, data transfer circuits, logic circuits, rotation, shift function, switching circuits.

Introduction

IN MANY digital computers, certain arithmetic operations and data manipulations require the basic operation of rotation and shifting of binary words. Rotation involves moving the data bits within a given flip-flop register to the right or left by a predetermined number of bit positions. The end bits are interconnected to allow the information to rotate in a circular arrangement. The shift operations differ from rotation only in that the end bits are not interconnected. When the bits move past the end of the register, they are lost and the opposite end is filled in with either ones or zeros. Some arithmetic operations demand that vacated bits become ones. This paper is concerned with the logical function whereby the opposite end of the register is filled in with zeros for both right and left shift.

Manuscript received October 22, 1968.
The authors are with Bell Telephone Laboratories, Inc., Naperville, Ill.

The rotate and shift functions can be implemented in many ways. The choice depends considerably upon system requirements. Several commonly used schemes will be discussed first. Then, after describing two bidirectional gating devices and illustrating their operation, a detailed presentation will be given of a novel rotate/shift circuit employing the bidirectional transfer properties of these gates. Finally the implementation of this novel circuit into a data processing arrangement will be illustrated.

Rotate and Shift Function with a Shift Register

One method of implementation is the use of binary cells [1] as shown in Fig. 1. The binary cells are made up of two parts: the flip-flop (FF) and the steering network (SN). This is a discrete component design where capacitors are relatively cheap and are used as storage elements or memory for transfer of data signals. In an integrated circuit design, diodes and transistors are preferred components over capacitors. Even though diffused junction-type capacitors can be fabricated on the same substrate as transistors or diodes, their maximum capacitance per unit area is relatively low. They have inherently large parasitics and are subject to voltage modulation. The best approach in an integrated circuit design is to avoid the use of capacitors. Consequently, the JK flip-flop [2] has been designed primarily for an integrated circuit using only transistors, diodes, and resistors. The logical properties are similar. For our discussion, however, we will not concern ourselves

parallelism

AN INTRODUCTION TO THE ILLIAC IV COMPUTER

by David E. McIntyre

This introduction to ILLIAC IV is written for a computer user who has only an acquaintance with the hardware involved in a conventional digital computer. For a more complete description consult references 1, 2 and 3.

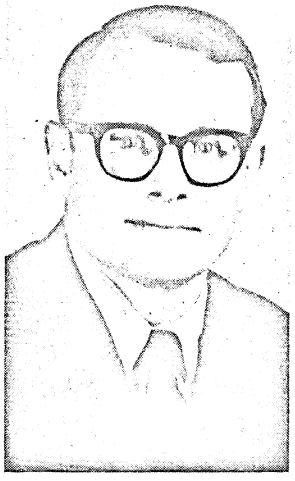
A stereotype computer can be functionally characterized as shown in Fig. 1a. It is composed of: (1) a memory that holds operands and instructions; (2) a control unit that fetches instructions from the memory, decodes them, and issues control signals (microsequence pulses) that operate, or drive; and (3) an arithmetic unit that performs the computation operations (addition, logical operations and multiplication) on operands that have been supplied from memory, and returns the result to the memory. In effect, the control unit monitors and controls the flow of information between the memory and the arithmetic unit and operates the arithmetic unit.

A typical sequence of events that takes place during operation is:

1. An instruction is fetched from memory to the control unit.
2. When it arrives in the control unit, it is decoded.
3. If the instruction involves operands in memory, the operands are fetched to the arithmetic unit.
4. When the operands arrive in the arithmetic unit, the computation (for instance, subtraction) is initiated and monitored by the control unit until complete.
5. After completion, the result is stored in memory.

There are several ways one can modify this stereotype design to achieve an increase in computing speed. One way would be to add additional control units and arithmetic units. This is the multiprocessor configuration. Another way (Fig. 1b) would be to divide the arithmetic unit into a group of functionally independent subunits, each of which could be operated independently by the control unit. As the control unit decodes instructions, it will determine when two or more consecutive instructions use separate functional subunits and are independent of each other. If this is the case, the separate instructions are allowed to proceed concurrently in the separate functional units rather than sequentially, as was the case with the stereotype machine. This is the approach employed in the CDC 6600, IBM 360/9X series and other current generation computers. If the number of

1. Slotnick, D. L., et al. "The ILLIAC IV Computer," *IEEE Transactions on Computers*, V. C-17, No. 8, August 1968, pp. 746-757.
2. Kuck, D. J., "ILLIAC IV Software and Application Programming," *IEEE Transactions on Computers*, V. C-17, No. 8, August 1968, pp. 758-770.
3. Davis, R. L., "The ILLIAC IV Processing Element," *IEEE Transactions on Computers*, V. C-18, No. 9, September 1969, pp. 800-816.



Mr. McIntyre is a senior engineer, ILLIAC IV project, University of Illinois. Previously he was a captain, USAF, and chief of the computer group at the Air Force Weapons Lab, Kirtland AFB, Albuquerque, N.M. He has a BA in mathematics from Southern Illinois University and an MS in engineering physics from the Air Force Institute of Technology.

The innovations of new large-scale computers often show what design characteristics will become standard in medium-sized machines for the commercial market a few years later. Thus, to give you a look at new ideas that may help shape the future, we have gathered together in this issue some examples of very large systems. There are descriptions of the IBM 360/195 and CDC 7600, with emphasis on the latter's software, plus an introduction to one of the supercomputers—the ILLIAC IV. In addition, Dr. Graham of RAND discusses the advantages and drawbacks of parallel versus pipeline design, using ILLIAC IV and CDC's forthcoming STAR as examples of these contrasting approaches.

operands that can be processed by the arithmetic unit is increased, the speed with which the arithmetic unit can obtain and store operands from the memory must also be increased to avoid a bottleneck. This can be achieved by partitioning the memory into several sets of memory banks. A memory operation can then take place simultaneously in separate memory banks.

Fig. 2 (p. 62) shows how the stereotype design has been modified in the design of ILLIAC IV. This figure describes one quadrant, or one-fourth of the ILLIAC IV array. The control unit operates in very much the same manner as the control unit in the stereotype computer. Instructions are fetched from the memories to the control unit where they are decoded and microsequence signals are produced. The microsequence signals are duplicated 64 times, and each set of microsequence signals is passed to a separate arithmetic unit. The same set of signals operates 64 different arithmetic units and increases the number of arithmetic operations that can be performed by a factor of 64. An arithmetic unit is referred to as a "processing element" (PE). Each of the 64 PE's has one memory bank and can fetch or store operands only to or from its own unique memory bank. The control unit, however, can fetch instructions from any of the 64 memory banks. The restriction that each arithmetic unit perform memory operations only with its unique memory solves some problems and poses some others.

If the memory banks of ILLIAC IV were arranged with a large crossbar switch so that any PE could access data from any memory bank, there would be delays imposed because of the distance that the signals would have to travel. Furthermore, if two different PE's required a datum that was stored in a third PE's memory bank, one of the two PE's would be forced to wait a complete memory cycle before it could receive its operand because only a single memory operation can take place using the same memory bank at one time. These delays are referred to as bank conflicts and are encountered even in concurrent computers. Therefore, by assigning each processing element its unique memory near the PE, signal line delays can be minimized and the possibility of bank conflict will be eliminated because only one PE will be making demands on a given memory bank.

Programs run most efficiently if all the operands used by PEi can be stored in memory i. This is a restrictive condition

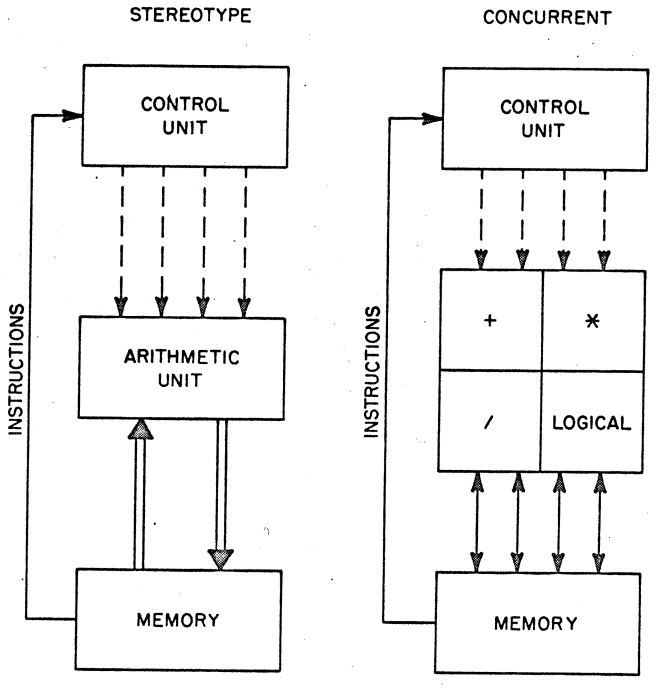


Fig. 1a (left). Functional block diagram for a stereotype computer.

Fig. 1b (right). Functional block diagram for a concurrent computer.

and is not always possible. Occasionally PEi needs to use an operand that is stored in PEj's memory. When this is the case, the operand in PEj's memory can be transmitted to a register in PEj and then again transmitted to the corresponding register in PEi. PEi can then store the operand in its memory. This process is called routing and will be explained in more detail in the programming section of this article.

The processing element (Fig. 3) in the ILLIAC IV is basically a four-register arithmetic unit. There is an A register and a B register used to hold the operands for arithmetic and logical operations. Operands for arithmetic operations are placed one in the A register and one in the B register when the operation is performed and the result is left in the A register. The S register is provided as temporary storage to avoid making repeated accesses to memory to fetch or store intermediate results. The R register is used to transfer information among the PE's in the routing operation. Each of these registers is 64 bits long.

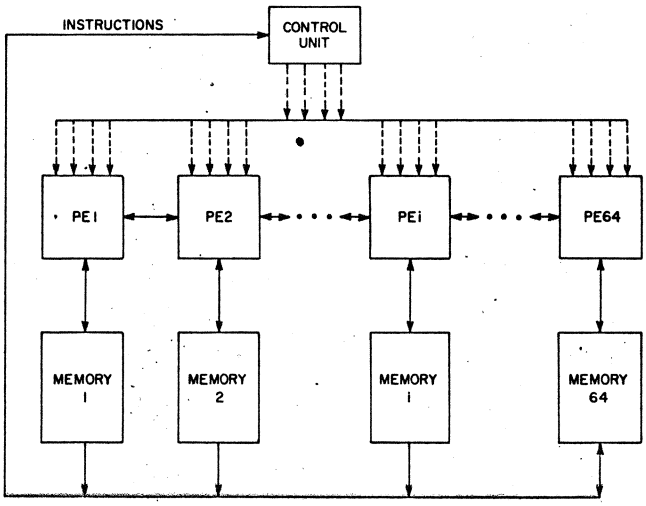


Fig. 2. ILLIAC IV quadrant.

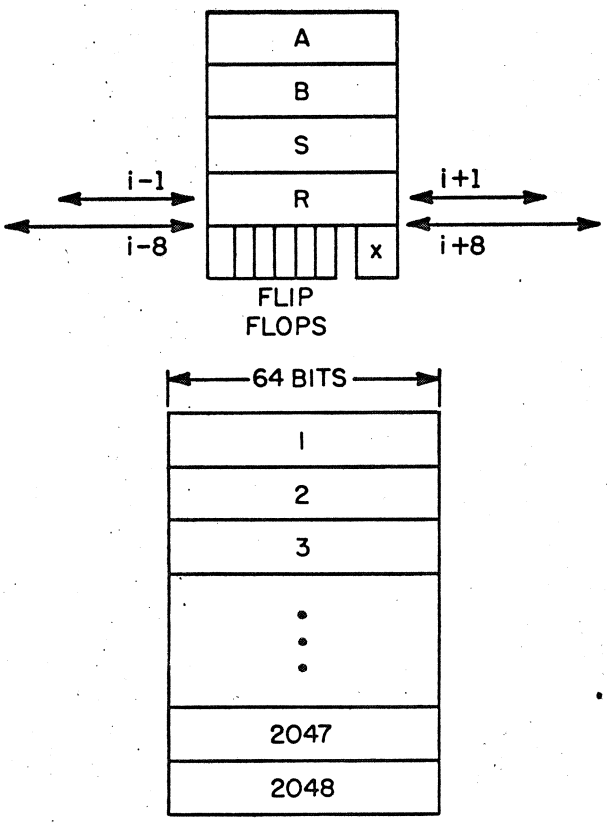


Fig. 3. ILLIAC IV processing element and associated processing element memory.

The R register of every PE is wired directly to the R registers of four other PE's. PEi is connected to PEi+1, PEi−1, PEi+8, and PEi−8 via the R register. The routing operation uses the R registers and can be visualized by considering the 64 R registers as a large 4096-bit register. Upon executing the command "route 1 to the right," this long register is shifted end around 64 bits to the right. (See Fig. 4.) Routes can be performed toward the right (in the direction of increasing PE number) or left. A distance 8

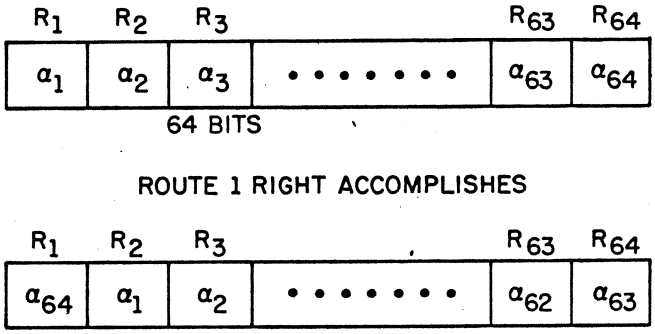


Fig. 4. Route instruction.

route (shift of 512 bits) is provided so that information can be rapidly sent between PE's with greatly different numbers. Displacements of ±1 and ±8 require about 100 nsec. Arbitrary distance routes are decomposed by the hardware into several consecutive routes of distances 8 or 1.

There is also an 18-bit index register (x register), which is used mainly to increment a basic memory address. Finally, there are eight 1-bit flip-flops that can be used to store the results (true or false) of tests, logical operations, etc. Each PE memory is composed of 2048 64-bit words. It is a semiconductor memory with a cycle time of roughly 200 nsec.

It should be stressed that the microsequence stream controlling PEi is exactly the same stream that controls PEj, and the PE's are constrained to execute exactly the same instructions at exactly the same time. When PE1 is performing an addition, PE5 can not perform a multiplication. There are two degrees of local autonomy provided for a PE. The first degree of autonomy involves "turning off" or disabling a PE. A disabled PE can perform no operations. A PE can be disabled either on command from the control unit or as a result of some conditional test. For instance, at the end of an arithmetic operation, the control unit can issue a command that is interpreted as "any PE that has computed a negative result, turn yourself off." Once a PE is turned off, it can no longer turn itself back on and must be enabled on command from the control unit. The other degree of independence available is that each PE may use a different memory location for a memory operation. This can be done by incrementing a base address by the contents of the index register in each PE. Suppose PE17 is to store the contents of its A register in memory location 35, while PE18 is to store the contents of its A register in memory location 45. The index register in PE17 would be set to zero, the index register in PE18 would be set to 10, and the control unit would issue the instruction "store to location 35 incremented by the index

register." In PE17 the memory address would be incremented by zero, and the store would be to location 35. In PE18, the address 35 would be incremented by 10, and the store would be performed into location 45. These two degrees of freedom associated with each PE actually provide a great deal of flexibility in programming ILLIAC IV.

PE's can be operated either in 64-bit mode or in 32-bit mode. In the 32-bit mode, each 64-bit word is considered as two 32-bit words and two 32-bit floating point operations can be performed in roughly the time required to perform one 64-bit operation. In 64-bit mode, floating point numbers have 48-bit mantissas, leaving 16 bits for exponent and sign. In 32-bit mode, mantissas are only 24 bits long.

Table 1 compares the execution times for common operations between a single processing element and a CDC 6600. In all fairness, it should be pointed out that there is often a great deal of concurrency obtained using the CDC 6600. That is, several separate floating point operations can be going on at one time. There is also a limited amount of concurrency in ILLIAC IV. If two consecutive instructions to be sent to the PE's are independent and do not require the same components in the PE, they may be executed concurrently. Acknowledging that this is rather a rough comparison, it is fairly reasonable to equate in floating point computing power a single PE and one or two CDC 6600's. If all 64 PE's are enabled and doing useful work, they can produce floating point operations at a rate comparable to between 64 and 128 CDC 6600's.

	PE nsec	6600 nsec
Memory to Operating Register (fetch)	350	800
Floating Add	350	600
Floating Multiply	450	1000
Floating Divide	2750	3000
Register-Register Transfer	50	300
Operating Register to Memory (store)	300	1000

Table 1. Comparison between PE and CDC 6600 of execution times for some common operations

control unit

Fig. 5 gives a functional representation of the major components in the control unit (CU). There is a local data memory composed of 64 words. This local memory can be filled from any location in any PE memory and also stored to any location in any PE's memory. There is a block of 64 words called the Program Look Ahead (PLA). This block of words provides an instruction queue and its operation will be explained in a later paragraph. The arithmetic unit in the control unit is a very simple unit and is restricted to performing logical operations and fixed-point addition and subtraction, obtaining operands and storing results only within the local data memory.

The instruction decoding logic decodes instructions provided from the program look ahead (PLA). If the instruction is of the type to be executed by the array of processing elements, the decoded instruction is fed into the microsequence generator where the microsequence pulses are generated and sent down control lines to drive the processing elements. If the instruction is one to be executed in the control unit, the decoded instruction is issued to the simple arithmetic unit. Most of the instructions executed by the control unit involve housekeeping operations associated with loops or indices. These housekeeping instructions can be executed concurrently with arithmetic instructions fed to the PE's. (This concurrency is not related to the concurrency in PE instructions, which was mentioned previously.)

An ILLIAC IV machine language instruction is composed of 32 bits. The 64 words (each word is 64 bits) in the PLA provide a queue of 128 instructions. Loops containing up to 128 instructions can be executed without any reference to PE memory. The 64 words are divided into 8 sections of 8 words each. When the control unit is executing the instruction contained in the fifth word of the 8-word section of

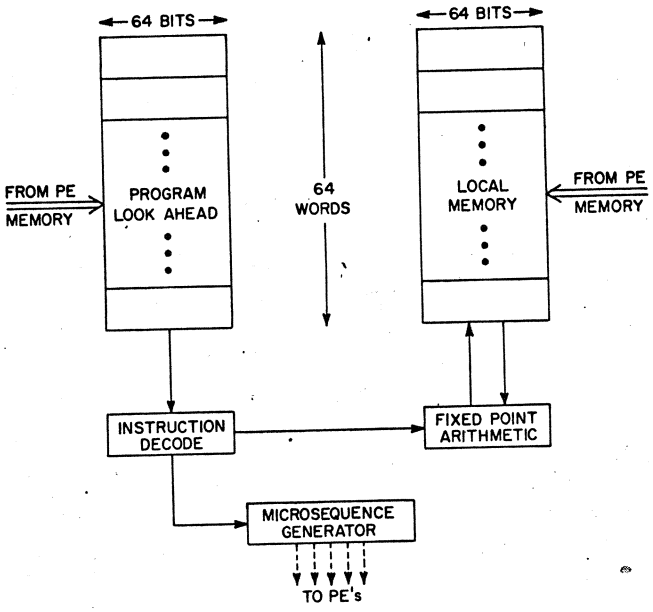


Fig. 5. Control unit.

instructions, it checks to see if the next 8 words of instructions are already contained in the PLA. If the next 8 words are not contained in the PLA, it will issue commands to bring them to the PLA and destroy the oldest subsection of 8 words. This effectively eliminates many of the delays imposed by instruction fetching, except for the case in which a jump is made to a section of the program that is not contained in the PLA. For a large range of programs that have been simulated, it has been found that the control unit is delayed waiting for instructions to be fetched from memory much less than 1% of the time.

cu-pe communication

Operands and control information can be transferred between the control unit and the PE's in several possible ways:

1. The control unit can broadcast a 64-bit word to all PE's simultaneously. The word originates in the CU local memory, or the arithmetic unit in the CU, and the destination can be any of the 64-bit operating registers in the PE.

2. The control unit can broadcast a 64-bit word with one bit going to each PE. That is, bit one would go to PE1, bit two to PE2, . . . , bit 64 to PE64. The destination of the bit going into a PE can be any of eight 1-bit registers in each PE. This is a method by which PE's are enabled and disabled. For instance, if it is desired to enable all even-numbered PE's and to disable all odd-numbered PE's, the control unit (using its arithmetic unit and logical operations) constructs a 64-bit word which has alternating ones and zeros. This word is then passed to the microsequence generator where one bit is sent to each PE, disabling all PE's

that receive a zero, and enabling all PE's that receive a one. The 1-bit register that specifies whether a PE is enabled or disabled is called the mode register.

3. The control unit receives information from the processing element in the reverse of the method previously described. That is, a bit is sampled from a 1-bit register in each processing element and assembled into a 64-bit word in the control unit. The control unit can use this facility to determine which PE's are enabled by assembling a 64-bit word from the single bit mode registers in 64 different PE's.

4. The control unit can fetch words from the memory of any PE into the local data memory or into the PLA. The fetch can consist of a transfer of one 64-bit word or a transfer of 8 contiguous 64-bit words. The fetch of 8 contiguous words requires only slightly longer than the fetch of one word, and thus is a high-speed method of getting large amounts of data into the control unit from the PE memory. All of the fetching to the PLA is automatic, as was previously pointed out.

the illiac iv system

Fig. 6 shows the ILLIAC IV system organization. It is composed of four identical control units, each control unit driving 64 PE's with 64 PE memories. The CU's are connected by lines that allow all to execute exactly the same instruction stream. In this "united" mode of operation, routing is provided across quadrants and end around from PE256 to PE1.

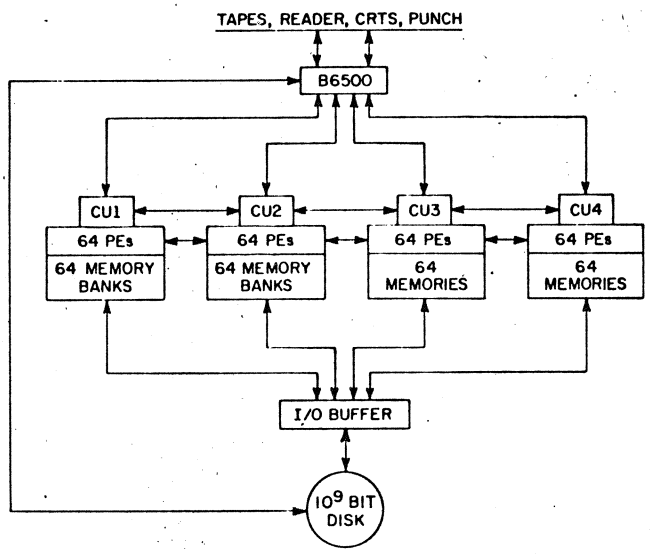


Fig. 6. ILLIAC IV system.

ILLIAC IV can be operated in several configurations. For example, all control units can be executing the same instruction stream, or each could be executing a different instruction stream; also, two control units could be executing one instruction stream and two executing another. It is possible to change the configuration during the execution of a program, but it is felt that this is not an extremely practical facility and does require certain careful programming considerations.

ILLIAC IV performs its input/output (I/O) through the Burroughs B6500 computer. The B6500 is very similar to the B5500, but is about five times faster. The data base for programs that are not core contained resides on a 10^9 bit head-per-track disc. This disc has two controllers, and each controller is capable of transferring into or out of the ILLIAC IV memory at the rate of 500×10^6 bits per second. If input and output were being carried on simultaneously by using both controllers, the effective transfer rate can be 10^9 bits per second. This disc has a revolution time of 40 msec, giving an average access time of 20 msec.

The average effective access time can be decreased considerably below 20 msec when several I/O requests can be accumulated in the I/O controller. There is a mechanism in the I/O controller that compares the beginning disc address of all I/O operations in a queue of requests with the address of the section of the disc that is passing under the read-write heads. As soon as a match is found, an I/O operation is initiated. For example, suppose two I/O descriptors reside in the descriptor queue with the lowest (oldest) descriptor, descriptor a, referencing a disc address that is located 270 degrees away from the disc address that is passing under the read-write heads, and the second I/O descriptor, descriptor b, referencing a disc address that is located only 90 degrees away from the disc address under the read-write head. (See Fig. 7.) The logic in the I/O controller would initiate the I/O operation b first, requiring only a disc rotation of 90 degrees, or a lateney of only 10 msec, instead

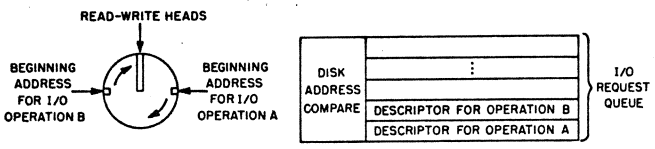


Fig. 7. I/O descriptor queue.

of initiating the I/O operation a, which would require a disc rotation of 270 degrees, or 30 msec latency time. The queue in the I/O controller can contain 24 I/O descriptors.

The B6500 exercises control over the CU's and all of the interactions between the disc and the computing array. The control units request I/O of the B6500, and it coordinates all I/O requests and initiates all I/O transfers between the disc and the array. At the end of an I/O transfer, it signals the control unit that the transfer is complete. In addition to performing in this supervisory capacity it also does all of the compiling for programs to be executed on ILLIAC IV. All external data used by the ILLIAC IV array goes first to the ILLIAC IV disc. For instance, if an executing ILLIAC IV program needs to read a tape unit, it makes the request of the B6500. The B6500 then reads one of its tape units, writes the result onto the ILLIAC IV disc and initiates the operation to bring the record from the ILLIAC IV disc into the ILLIAC IV memory.

programming illiac iv

It was mentioned previously that the separate memories assigned to each processing element do provide operands at a very high rate to each processing element, but they also cause a complication when programming ILLIAC IV. The programmer must arrange a storage allocation scheme so that when PEi needs a datum of information, it is "easily" accessible to PEi. This does not necessarily mean that the information must be stored in PEi's memory, since the route instruction makes it possible to obtain operands from memories other than that associated with PEi. However, it does mean that the transfer of operands from one memory to another by using the routing instruction must be done in some regular fashion.

Suppose PE7 needs a datum of information that is stored in PE2's memory at the same time that PE12 requires a datum of information that is stored in PE7's memory. This

requires a parallel operation because both PE2 and PE7 can fetch operands from their memory and simultaneously route a distance 5 to the right (by performing a route 8 to the right and 3 consecutive routes of 1 to the left), making operands available to PE7 and PE12. However, if PE7 needed the information from PE2, and PE12 needed the information from PE19, this transfer of information would be impossible to implement in parallel because one involves a route to the right while the other involves a route to the left.

Programming ILLIAC IV involves a rather alien situation for the programmer who is accustomed to the conventional machine. He must not only think of some way to implement his mathematical algorithm, but he must also think of a memory allocation scheme for storing data which allows the algorithm to be implemented in a parallel fashion. However, memory allocation problems are not totally new to the FORTRAN programmer. An experienced FORTRAN programmer can use the DIMENSION, COMMON and EQUIVALENCE statements to force the FORTRAN compiler to allocate data storage as the programmer chooses. It is often a common programming trick to reference a doubly dimensioned variable as a singly dimensioned variable. For instance, if A were dimensioned 10 × 10, a programmer will sometimes reference A(27) knowing that he is actually referring to A(7,3). Occasionally it is expedient to write a subprogram with arguments that are singly dimensioned arrays but to call, or enter, that subprogram using a doubly dimensioned argument.

In order to effectively use the EQUIVALENCE and COMMON statements and employ some simple programming practices, the programmer must know how arrays are stored or distributed in memory. When using ILLIAC IV, the programmer must consider the memory allocation while he is composing the program rather than simply regarding it as a casual "after-the-fact" problem.

	PE1	PE2	PE3
loc 1			
	o	o	o
	o	o	o
	o	o	o
loc 10	X_{11}	X_{12}	X_{13}
loc 11	X_{21}	X_{22}	X_{23}
loc 12	X_{31}	X_{32}	X_{33}
	o	o	o
	o	o	o
	o	o	o
loc 25	Y_{11}	Y_{12}	Y_{13}
loc 26	Y_{21}	Y_{22}	Y_{23}
loc 27	Y_{31}	Y_{32}	Y_{33}
	o	o	o
	o	o	o
	o	o	o
loc 102	Z_{11}	Z_{12}	Z_{13}
loc 103	Z_{21}	Z_{22}	Z_{23}
loc 104	Z_{31}	Z_{32}	Z_{33}
	o	o	o
	o	o	o
	o	o	o
loc 2048			

Fig. 8. Straight storage.

Perhaps it is worthwhile to illustrate the problem of memory allocation and also to demonstrate how parallelism can be achieved by specifying the steps that are programmed to perform a matrix multiply. For this example, ILLIAC IV will be constrained to use 3 PE's. The first step is to arrange to store the elements of the 3 × 3 matrices, X, Y, and the product matrix Z (we will compute X ° Y = Z) in the memories as is shown in Fig. 8. This form of storage is commonly used for doubly dimensioned variables and is referred to as "straight storage." X_{12} is stored in location 10 in PE2's memory, Y_{33} is stored in location 27 in PE3's memory, etc.

Step 1: Copy the first row of X to the CU local data memory (the contents of memory location 10 in the first three PE's).

Step 2: Simultaneously fetch the first row of Y to the B registers. Each PE fetches from location 25.

Step 3: Broadcast X_{11} from the CU to the A registers of all PE's. The contents of the registers are shown in Fig. 9a.

Step 4: Multiply (the contents of A times the contents of B replace what was in A), and store contents of A in the S register.

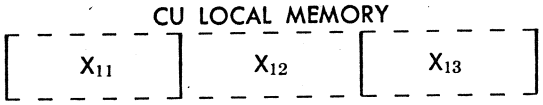
Step 5: Fetch the second row of Y to the B registers. Each PE fetches from location 26.

Step 6: Broadcast X_{12} to all A registers. Contents of A, B, and S registers are shown in Fig. 9b.

Step 7: Multiply (X_{12} ° Y_{21} is formed in PE1's A register, while X_{12} ° Y_{23} is formed in PE3's A register).

Step 8: Add contents of S to contents of A and store results in S.

Step 9: Fetch third row of Y to B registers. Each PE fetches from location 27.



	PE1	PE2	PE3
A reg	X_{11}	X_{11}	X_{11}
B reg	Y_{11}	Y_{12}	Y_{13}

Fig. 9a. Contents of A and B registers after Step 3 of matrix multiply.

	PE1	PE2	PE3
A reg	X_{12}	X_{12}	X_{12}
B reg	Y_{21}	Y_{22}	Y_{23}
S reg	X_{11} *Y_{11}	X_{11} *Y_{12}	X_{11} *Y_{13}

Fig. 9b. Contents of A, B, and S registers after Step 6 of matrix multiply.

Step 10: Broadcast X_{13} to all A registers. (See Fig. 9c, p. 66.)

Step 11: Multiply

Step 12: Add contents of A to contents of S. Fig. 10 (p. 66) shows contents of A registers after addition. Note that the first row of the product matrix has been formed, Z_{11} in PE1, Z_{13} in PE3.

Step 13: Store contents of A registers to first row of Z. All PE's store to location 102.

Now, in order to compute the second row of the product

matrix, the second row of X is fetched to the local data memory in the CU, and the process is repeated.

In fact, what we have demonstrated through the description of a matrix multiply is that ILLIAC IV can do elementary row operations "in parallel." However, there will be many applications in which it is desirable to perform column operations in parallel, as well as row operations. In particular, matrix inversion and numerical solution of partial differential equations require this facility. An alternate method of memory allocation, called "skewed storage," permits row and column operations to be performed in parallel.

	PE1	PE2	PE3
A reg	X_{13}	X_{13}	X_{13}
B reg	Y_{31}	Y_{32}	Y_{33}
S reg	$X_{11} * Y_{11}$ + $X_{12} * Y_{21}$	$X_{11} * Y_{12}$ + $X_{12} * Y_{22}$	$X_{11} * Y_{13}$ + $X_{12} * Y_{23}$

Fig. 9c. Contents of A, B, and S registers after Step 10 of matrix multiply.

	PE1	PE2	PE3
A reg	$X_{11} * Y_{11}$ + $X_{12} * Y_{21}$ + $X_{13} * Y_{31}$	$X_{11} * Y_{12}$ + $X_{12} * Y_{22}$ + $X_{12} * Y_{32}$	$X_{11} * Y_{13}$ + $X_{12} * Y_{23}$ + $X_{13} * Y_{33}$

Fig. 10. Contents of A register after Step 12 of matrix multiply.

Fig. 11b shows the skewed storage technique for a 4×4 matrix A. As in straight storage, the first row is stored across the PE's at some location ζ in the PE memory. The second is then "skewed" or rotated once to the right, so that a_{21} is stored in PE2 (instead of PE1 as would have been the case with straight storage). The third and fourth rows are skewed two and three PE's to the right, respectively.

Now in order to perform row operations involving the third row, the contents of memory location $\zeta + 2$ is simultaneously copied to the PE operating registers. To perform a column operation involving the first column, the index registers are loaded as shown in Fig. 11a. The index register in PE1 is loaded with 0; the index register in PE3 is loaded with 2. Then, on the command "fetch from location ζ incremented by the index register," the first column (circled elements in Fig. 11b) is simultaneously copied to the PE operating registers. In order to fetch the second column, the contents of the index registers are simply rotated one PE to the right (dotted portion of Fig. 11a), "fetch from location ζ incremented by index register" is executed, and the second column (triangles) is obtained.

the higher level language

Because the architecture of ILLIAC IV is so different from that of a conventional machine, and because there is so much potential computing power available with ILLIAC IV, it is quite a challenge to construct a higher level language that will allow the user to easily and effectively use the machine. Such a language, called TRANQUIL, is being developed at the University of Illinois. It is very similar to ALGOL and is, in fact, a superset containing ALGOL. It differs from ALGOL in several areas, principally in the use of index "sets" for loop control and by permitting two types of loops: a sequential loop and a loop that is executed simultaneously (in separate PE's).

Instead of attempting to detail the language and compiler, some illustrations of TRANQUIL source code will be given and the method of implementation by the compiler will be explained.

```
FOR(I,J) SEQ([1, 2, . . ., 10], [5, 10, . . ., 50]) DO
        A[I] ← B[I+1] + C[J]
```

The FOR statement is the loop statement like DO in FORTRAN. This is interpreted as: for I sequentially taking on

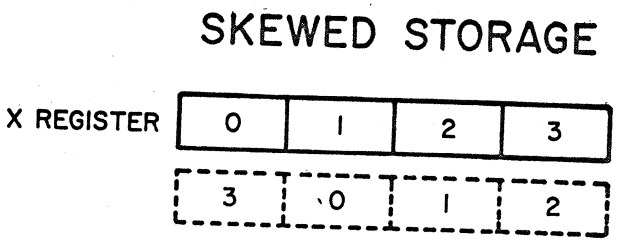


Fig. 11a. X register rotation scheme for skewed storage.

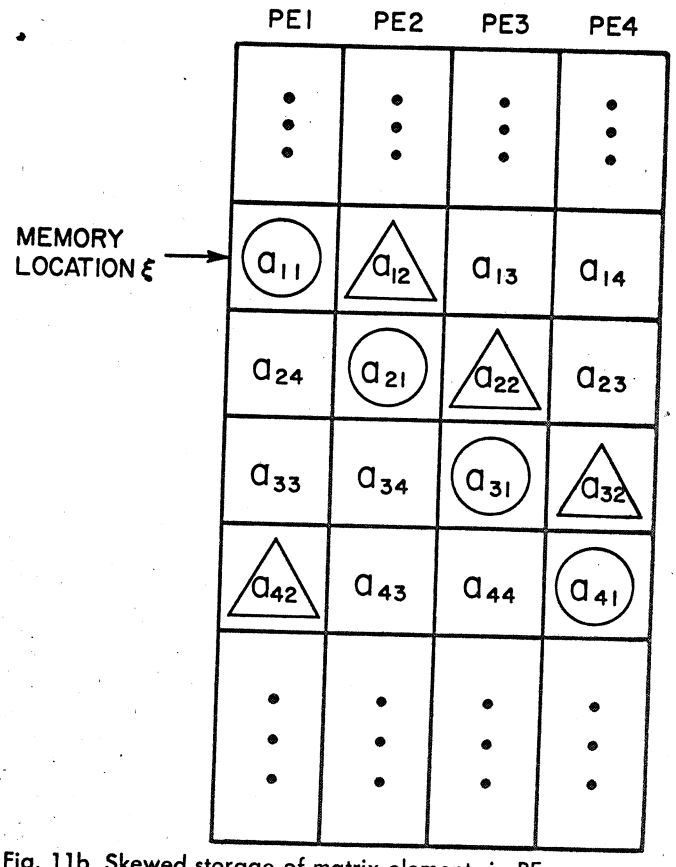


Fig. 11b. Skewed storage of matrix elements in PE memory.

the values 1, 2, etc. up to 10; and J sequentially taking on the values 5, 10, etc. up to 50; replace A(I) with the sum of B(I+1) and C(J). The object code would first locate B(2) and C(5) in the PE memories (according to some memory allocation scheme that had been previously specified to the compiler by the programmer), copy them to some PE, perform the sum and store the result in the location assigned to A(1). This process would then be repeated with B(3), C(10) and A(2). After a total of 10 repetitions,

A(10) would contain the sum of B(11) and C(50). Total operations involved in this loop, neglecting set-up time, is 20 locate-fetches, 10 adds and 10 stores.

FOR (I,J) SIM ([1, 2, . . ., 10], [5, 10, . . ., 50] DO
A[I] ← B[I+1] + C[J]

This construction will implement the loop simultaneously. B(2), B(3), . . . , B(11) will be located and fetched to PE operating registers simultaneously, as will C(5), C(10) . . . , C(50). The sums will be computed in separate PE's at the same time, and the results stored to A. Total operations involved for this loop are 2 locate-fetches, one add and one store. If some naive programmer had assigned the arrays B, C, and A all to PE5, this simultaneous implementation would not be possible, and the statement would necessarily be executed sequentially. The compiler can only implement parallelism that has been enhanced by the programmer (through judicious choice of memory allocation schemes and index sets); it cannot impose parallel or simultaneous operations in all circumstances. Some future version of TRANQUIL may be able to distinguish the implicit parallelism in an instruction string, but certainly for the first few months the user must be explicit in his definition of what portions of the program can be implemented in parallel. A sample TRANQUIL program is shown in Fig. 12.

Line Number	SAMPLE TRANQUIL PROGRAM
1	REAL SKEWED ARRAY A, B[0:100, 0-100];
2	INCSET JJ
3	MONOSET II(1) [27:6], KK(1) [100:60];
4	INTEGER I, J, K, L;
5	II ← [2, 10, 13, 21, 24];
6	JJ ← [2, 4, . . ., 98];
7	FOR (I) SEQ (II) DO
8	BEGIN FOR (J) SIM (JJ) DO
9	KK ← SET (J:A [I, J] < B [I, J]);
10	FOR (K) SIM KK) DO
11	A [I, K] ← A[I+1, K] + B[I, K+1];
12	END

Fig. 12. Sample TRANQUIL program.

Some of the features of this TRANQUIL program will be described. Lines 1 through 4 are declaration statements. Lines 5 and 6 define sets. Lines 7 through 11 are the "executable" portions.

Line 1: Informs the compiler that arrays A and B are type real and should be stored in skewed storage fashion. The programmer can then do row or column operations with equal facility (although line 11 involves only a row operation). Limits the range of the two subscripts to between 0 and 100.

Line 2: Specifies that JJ is to be considered an "increment" set, a set whose succeeding elements differ by a constant increment.

Line 3: Specifies that II and KK are sets of monotone "one-tuples". II has no more than six elements (max element no more than 27) and KK has no more than 60 elements (max element no more than 100).

Line 4: Types I, J, K, L as integer.

Line 5: Defines the set II to be the set (2, 10, 13, 21, 24).

Line 6: Defines the set JJ to be the set (2, 4, 6, 8, 10, . . . , 96, 98).

Line 7: The start of a loop which extends through line 11 (the commands contained between the BEGIN of line 8 and the END of line 12). This loop will be performed first with I = 2, then with I = 10, then I = 13, I = 21, and finally I = 24.

Line 8: The start of a simultaneous loop with J taking on values 2, 4, 6, . . . , 98.

Line 9: Defines a new set KK. KK is the set of even J, such that A[I, J] is less than B[I, J]. Observe that KK will, in general, be a different set each time I, the index on the outer loop, changes.

Line 10: The start of a simultaneous loop in which the index K takes on the values of the new set KK that has just been generated.

Line 11: This arithmetic and replacement is done simultaneously in several PEs (provided KK has more than one element.

Line 12: Ends the inner loop.

TRANQUIL probably will not be used as the primary language to compose ILLIAC IV programs. Since descriptions of the language are available in the open literature (references 2, 4, 5) it was referred to here to illustrate the type of language extensions that are necessary to reflect the features of ILLIAC IV.

illiac iv operation

Because of economic considerations, only one quadrant of the four-quadrant system is currently under construction. This quadrant will be delivered to the University of Illinois, where it will be available for continued language and software development, and for use by the supercomputer users in the weather community, Department of Defense and AEC. At the university a 10^{12} bit mass storage device will be integrated with the system.

summary

ILLIAC IV, in order to achieve a cost-effective solution to the speed problem, has decentralized the central processing unit into four control units (each capable of executing independent instruction streams) each of which drive 64 processing elements—a total of 256 processing elements. Each processing element acts as an arithmetic and logic unit and has its own 2048-word (64 bits per word) memory capable of communicating with all of the other processing elements.

Distributing the arithmetic and logic functions over 256 processing elements allows the ILLIAC IV to perform operations simultaneously on many types of data structures. In addition to this parallelism, the processing elements are extremely fast computers in their own right: memory cycle time is less than 300 nsec; a 64-bit floating point add takes 250 nsec; a floating point multiply of two 64-bit numbers takes 450 nsec.

At the present time, however, ILLIAC IV is not "transparent" to the programmer; that is, he can not merely write his program in some high level language and remain unaware of the "insides" of the machine. He must have some knowledge of the computer at a machine language level in order to fully exploit the parallelism of ILLIAC IV. Of course, work continues in the development of a high-level language for ILLIAC IV so that it will be easier for the novice programmer to use.

4. Northcote, Robert S., "Software Development for the Array Computer ILLIAC IV," University of Illinois at Urbana-Champaign, Department of Computer Science, Urbana, Illinois, Document No. 214, March, 1969.
5. Abel, N. E., et al. "TRANQUIL: A Language for an Array Processing Computer," Proceedings AFIPS Spring Joint Computer Conference, 1969, pp. 57-75.

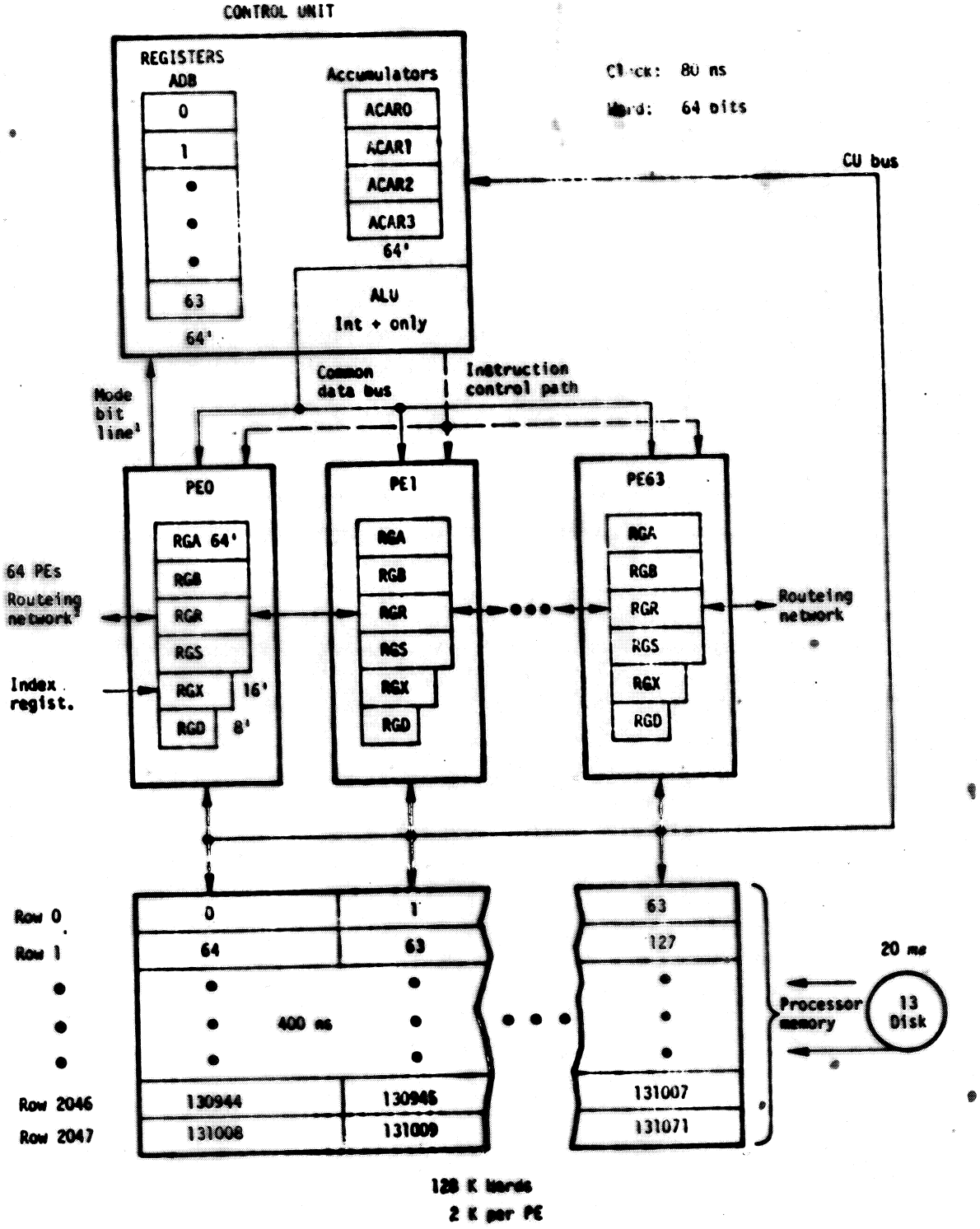


Figure 15: Logical layout of the ILLIAC IV