49, p. 439, 1966.
[158] W. H. F. Talbot, "Photogenic Drawing," **The Athenaeum**, No. 589, p. 114, 1839.
[159] K. Tamaribuchi and M. L. Smith, "Charge Determining Species in Non-Aqueous Solvents," **J. Colloid Interface Sci.**, Vol. 22, p. 404, 1966.
[160] T. Tani and S. Kikuchi, "Spectral Sensitization in Photography and Electrophotography," Report Inst. Industrial Sci., Univ. Tokyo, Vol. 18, p. 51, 1968.
[161] A. Terenin and I. Akimov, "Some Experiments on the Photosensitization Mechanism of Semiconductors by Dyes," **J. Phys. Chem.**, Vol. 69, p. 730, 1965.
[162] V. D. Tughan and R. C. Pink, "Solutions of Metal Soaps in Organic Solvents Part II," **J. Chem. Soc.**, p. 1804, 1951.
[163] V. Tulagin, "Imaging Method Based on Photoelectrophoresis," **J. Opt. Soc. Amer.**, Vol. 59, p. 328, 1969.
[164] E. J. Verwey and J. Th. G. Overbeek, **Theory of the Stability of Lyophobic Colloids**, Amsterdam: Elsevier, 1948.
[165] J. Viscakas, et al, "Recombination in Selenium Electrophotographic Layers," **Appl. Optics**: Suppl. =3 on Electrophotography, p. 27, 1969.
[166] J. Viscakas and V. Gaidelis, "The Role of Intercrystalline Barriers in ZnO Electrophotographic Layers," Reprographie II, IInd Int. Conf. Cologne, Helwich Darmstadt, 1969.
[167] O. Von Bronk, British Patent No. 188,030, 1922.
[168] E. W. Wagner, British Patent No. 1,065,796, 1967.
[169] L. E. Walkup, U.S. Patent No. 2,777,957, 1957.
[170] L. E. Walkup, U.S. Patent No. 2,825,814, 1954.
[171] P. J. Warter, Jr., "Factors Determining Xerographic Photoreceptor Performance," **Appl. Optics**: Suppl. =3 on Electrophotography, p. 65, 1969.
[172] H. Watanabe, et al, "The Activation Energy for Oxygen Desorption from Zinc Oxide Surfaces," **Jap. J. Appl. Phys.**, Vol. 4, p. 945, 1965.
[173] J. W. Weigl, **Photographic Science**, Ed. by W. F. Berg. New York: Focal Press, 1963.
[174] R. D. Weiss, "Electrolytic Photography," **Phot. Sci. and Eng.**, Vol. 11, p. 287, 1967.
[175] P. H. Wersema, et al, "Calculation of the Electrophoretic Mobility of a Spherical Colloid Particle," **J. Colloid Interface Sci.**, Vol. 22, p. 78, 1966.
[176] H. Wielicki, private communication.
[177] N. E. Wolff, "A Photoconductive Thermoplastic Recording System," **RCA Review**, Vol. 25, p. 200, 1964.
[178] W. Yellin, et al, British Patent No. 1,016,072, 1966.
[179] W. C. York, U.S. Patent No. 3,135,695, 1964.
[180] C. J. Young and H. G. Grieg, "Electrofax: Direct Electrophotographic Printing on Paper," **RCA Review**, Vol. 15, p. 469, 1954.
[181] Anon., **Electronic News**, p. 68, February 16, 1970.

The Illiac IV System

W. J. BOUKNIGHT, STEWART A. DENENBERG, DAVID E. McINTYRE, J. M. RANDALL, AMED H. SAMEH, AND DANIEL L. SLOTNICK, SENIOR MEMBER, IEEE

Invited Paper

Abstract—The reasons for the creation of Illiac IV are described and the history of the Illiac IV project is recounted. The architecture or hardware structure of the Illiac IV is discussed—the Illiac IV array is an array processor with a specialized control unit (CU) that can be viewed as a small stand-alone computer. The Illiac IV software strategy is described in terms of current user habits and needs. Brief descriptions are given of the systems software itself, its history, and the major lessons learned during its development. Some ideas for future development are suggested. Applications of Illiac IV are discussed in terms of evaluating the function $f(x)$ simultaneously on up to 64 distinct argument sets x_i. Many of the time-consuming problems in scientific computation involve repeated evaluation of the same function on different argument sets. The argument sets which compose the problem data base must be structured in such a fashion that they can be distributed among 64 separate memories. Two matrix applications: Jacobi's algorithm for finding the eigenvalues and eigenvectors of real symmetric matrices, and reducing a real nonsymmetric matrix to the upper-Hessenberg form using Householder's transformations are discussed in detail. The ARPA network, a highly sophisticated and wide ranging experiment in the remote access and sharing of computer resources, is briefly described and its current status discussed. Many researchers located about the country who will use Illiac IV in solving problems will do so via the network. The various systems, hardware, and procedures they will use is discussed.

I. Introduction

IT ALL BEGAN in the early 1950's shortly after EDVAC [1] became operational. Hundreds, then thousands of computers were manufactured, and they were generally organized on Von Neumann's concepts, as shown and described in Fig. 1. In the decade between 1950 and 1960, memories became cheaper and faster, and the concept of archival storage was evolved; control-and-arithmetic and logic units became more sophisticated:

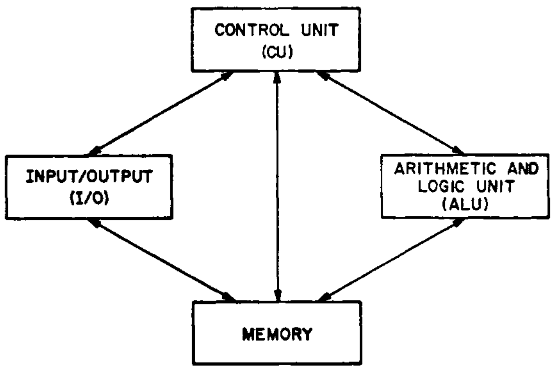


Fig. 1. Functional relations within a conventional computer. The CU has the function of fetching instructions which are stored in memory, decoding or interpreting these instructions, and finally generating the microsequences of electronic pulses which cause the instruction to be performed. The performance of the instruction may entail the use or "driving" of one of the three other components. The CU may also contain a small amount of memory called registers that can be accessed faster than the main memory. The ALU contains the electronic circuitry necessary to perform arithmetic and logical operations. The ALU may also contain register storage. Memory is the medium by which information (instructions or data) is stored. The I/O accepts information which is input to or output from Memory. The I/O hardware may also take care of converting the information from one coding scheme to another. The CU and ALU taken together are sometimes called a CPU.

I/O devices expanded from typewriter to magnetic tape units, disks, drums, and remote terminals. But the four basic components of a conventional computer (control unit (CU), arithmetic-and-logic unit (ALU), memory, and I/O) were all present in one form or another.

The turning away from the conventional organization came in the middle 1960's, when the law of diminishing returns began to

Manuscript received December 10, 1971; revised January 17, 1972. *This invited paper is one of a series planned on topics of general interest—The Editor.*

The authors are with the Center for Advanced Computation and the Illiac IV Project, University of Illinois, Urbana, Ill. 61801.

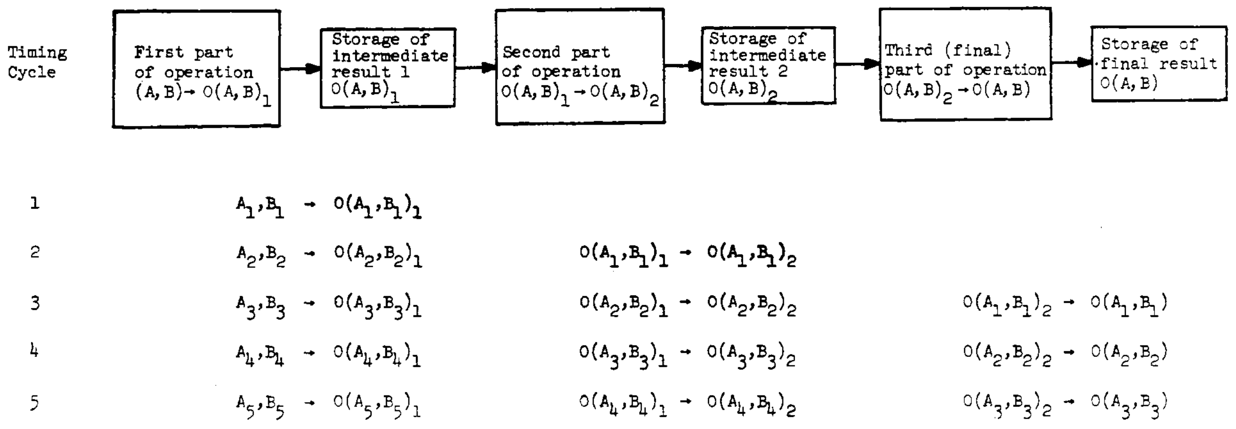

Timing Cycle
First part of operation $(A,B) \rightarrow O(A,B)_1$
Storage of intermediate result 1 $O(A,B)_1$
Second part of operation $O(A,B)_1 \rightarrow O(A,B)_2$
Storage of intermediate result 2 $O(A,B)_2$
Third (final) part of operation $O(A,B)_2 \rightarrow O(A,B)$
Storage of final result $O(A,B)$
1 $A_1,B_1 \rightarrow O(A_1,B_1)_1$
2 $A_2,B_2 \rightarrow O(A_2,B_2)_1$ $O(A_1,B_1)_1 \rightarrow O(A_1,B_1)_2$
3 $A_3,B_3 \rightarrow O(A_3,B_3)_1$ $O(A_2,B_2)_1 \rightarrow O(A_2,B_2)_2$ $O(A_1,B_1)_2 \rightarrow O(A_1,B_1)$
4 $A_4,B_4 \rightarrow O(A_4,B_4)_1$ $O(A_3,B_3)_1 \rightarrow O(A_3,B_3)_2$ $O(A_2,B_2)_2 \rightarrow O(A_2,B_2)$
5 $A_5,B_5 \rightarrow O(A_5,B_5)_1$ $O(A_4,B_4)_1 \rightarrow O(A_4,B_4)_2$ $O(A_3,B_3)_2 \rightarrow O(A_3,B_3)$

Fig. 2. Pipelined operation. The large boxes represent the circuits required to transform the operands A and B into the quantity $O(A, B)$ (some function of A and B, say the sum of A and B). The smaller boxes represent storage stages for the intermediate results $O(A, B)_1$ and $O(A, B)_2$, and the desired result $O(A, B)$. The operation O has been broken down into three stages, each of which accepts as input the output of the previous stage, and all of which perform a stage of the operation at the same time. At each step of the timing cycle, the pipeline accepts a new pair of operands (A, B) and the previous pair moves to the next stage. This mode of operation causes results (the sum in this example) to appear at the end of the pipeline at time intervals equal to the time of operation of the slowest stage of the pipeline.

take effect in the effort to increase the operational speed of a computer. Up until this point the approach was simply to speed up the operation of the electronic circuitry which comprised the four major functional components. (See Fig. 1.)

Electronic circuits are ultimately limited in their speed of operation by the speed of light (light travels about one foot in a nanosecond) and many of the circuits were already operating in the nanosecond time range. So, although faster circuits could be made, the amount of money necessary to produce these faster circuits was not justifiable in terms of the small percentage increase of speed.

At this stage of the problem two new approaches evolved.

1) *Overlap:* The hardware structure of the conventional organization was modified so that two or more of the major functional components (or subcomponents within a major component) could overlap their operations. Overlap means that more than one operation is occurring during the same time interval, and thus total operation time is decreased.

Before operations could be overlapped, control sequences between the components had to be decoupled. Certainly the CU could at least be fetching the next instruction while the ALU was executing the present one.

2) *Replication:* One of the four major components (or subcomponents within a major component) could be duplicated many times. (Ten black boxes can produce the result of one black box in one-tenth of the time if the conditions are right.) The replication of I/O devices, for example, was a step taken very early in the evolution of digital computers—large installations had more than one tape drive, more than one card reader, more than one printer.

Since the above two philosophies do not mutually exclude each other, a third approach exists which consists of both of them in a continuously variable range of proportions.

The overlapping philosophy was implemented largely through the buffer and pipeline mechanisms. The pipeline mechanism (see Fig. 2) breaks down an operation into suboperations, or stages, and decouples these stages from each other. After the stages are decoupled they can be performed simultaneously or, equivalently, in parallel. The buffer mechanism allows an operation to be decoupled into parallel operation by providing a place to store information.

The replication philosophy is exemplified by the general multiprocessor which replicates three of the four major components (all but the I/O) many times. The cost of a general multiprocessor is, however, very high and further design options were considered which would decrease the cost without seriously degrading the power or efficiency of the system. The options consist merely of recentralizing one of the three major components which had been previously replicated in the general multiprocessor—the memory, the ALU, or the CU. Centralizing the CU gives rise to the basic organization of a vector or array processor such as Illiac IV. This particular option was chosen for two main reasons.

1) *Cost:* A very high percentage of the cost within a digital computer is associated with CU circuitry. Replication of this component is particularly expensive, and therefore centralizing the CU saves more money than can be saved by centralizing either of the other two components.

2) *Structure:* There is a large class of both scientific and business problems that can be solved by a computer with one CU (one instruction stream) and many ALUs. The same algorithm is performed repetitively on many sets of different data: the data are structured as a vector, and the vector processor of Illiac IV operates on the vector data. All of the components of data structured as a vector are processed simultaneously or in parallel.

The Illiac IV project was started in the Computer Science Department at the University of Illinois with the objective of developing a digital system employing the principle of parallel operation to achieve a computational rate of 10^9 instructions/s. In order to achieve this rate, the system was to employ 256 processors operating simultaneously under a central control divided into four subassembly quadrants of 64 processors each. Due primarily to subcontractor problems several basic technological changes were necessitated during the course of the program, principally, reduction in individual logic-circuit complexity and memory technology. These resulted in cost escalation and schedule delays, ultimately limiting the system to one quadrant with an overall speed of approximately 200 million instructions/s. It is this one-quadrant system that will be discussed for the remainder of this paper.

The approach taken in Illiac IV surmounts fundamental limitations in ultimate computer speed by allowing—at least in principle—an unlimited number of computational events to take place simultaneously. The logical design of Illiac IV is patterned after that of the Solomon [2], [3] computers, prototypes of which were built by the Westinghouse Electric Corporation in the early 1960's. In this design a single master CU sends instructions to a sizable number of independent processing elements (PEs) and transmits addresses to individual memory units associated with these PEs ("PE memories," PEMs). Thus, while a single sequence of instructions (the program) still does the controlling, it controls a number of PEs that execute the same instruction simultaneously on data that can be, and usually are, different in the memory of each PE.

Each of the 64 PEs of Illiac IV is a powerful computing unit in its own right. It can perform a wide range of arithmetical operations on numbers that are 64 binary digits long. These numbers can be in any of the six possible formats: the number can be processed as a single number 64 bits long in either a fixed or a "floating" point representation, or the 64 bits can be broken up into smaller numbers of equal length. Each of the memory units has a capacity of 2048 64-bit numbers. The time required to extract a number from memory (the access time) is 188 ns, but because additional logic circuitry is needed to resolve conflicts when two or more sections of Illiac IV call on the memory simultaneously, the minimum time between successive operations of memory is increased to 350 ns.

Each PE has more than 100 000 distinct electronic components assembled into some 12 000 switching circuits. A PE together with its memory unit and associated logic is called a processing unit (PU). In a system containing more than six million components one can expect a component or a connection to fail once every few hours. For this reason much attention has been devoted to testing and diagnostic procedures. Each of the 64 processing units will be subjected regularly to an extensive library of automatic tests. If a unit should fail one of these tests, it can be quickly unplugged and replaced by a spare, with only a brief loss of operating time. When the defective unit has been taken out of service, the precise cause of the failure will be determined by a separate diagnostic computer. Once the fault has been found and repaired, the unit will be returned to the inventory of spares.

Illiac IV could not have been designed at all without much help from other computers. Two medium-sized Burroughs 5500 computers worked almost full time for two years preparing the artwork for the system's printed circuit boards and developing diagnostic and testing programs for the system's logic and hardware. These formidable design, programming, and operating efforts were under the direction of Arthur B. Carroll, who, during this period, was the project's deputy principal investigator.

The Illiac IV system is scheduled for completion by the end of this calendar year; the fabrication phase is essentially complete with some final assembly and considerable debugging yet to be completed.[1]

II. Hardware Structure

A. Illiac IV in Brief

As stated in the Introduction, the original design of Illiac IV contained four CUs, each of which controlled a 64-ALU array processor. The version being built by the Burroughs Corporation will have only one CU which drives 64 ALUs as shown in Fig. 3. It is for this reason that Illiac IV is sometimes referred to as a

[1] All of this work was sponsored under a Grant (Contract USAF 30(602)4144) from the Advanced Research Projects Agency.

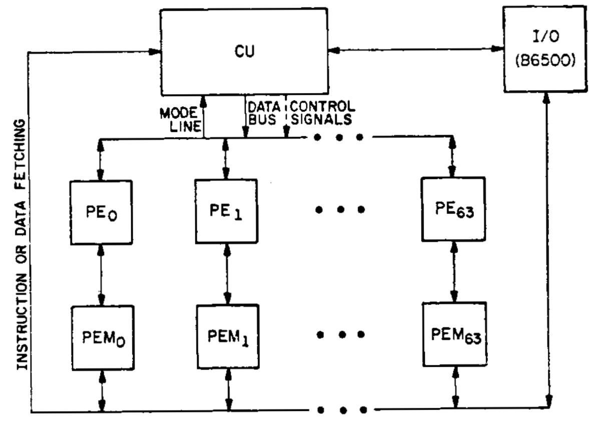


Fig. 3. Functional block diagram of Illiac IV.

quadrant (one-fourth of the original machine) and it is this abbreviated version of Illiac IV that will be discussed for the remainder of this paper. For a more complete description of the Illiac IV architecture see [4]–[6].

One difference between Illiac IV and a general array processor is that the CU has been decoupled from the rest of the array processor so that certain instructions can be executed completely within the resources of the CU at the same time that the ALU is performing its vector operations. In this way another degree of parallelism is exploited in addition to the inherent parallelism of 64 ALUs being driven simultaneously. What we have is 2 computers inside Illiac IV: one that operates on scalars, and one that operates on vectors. All of the instructions, however, emanate from the computer that operates on scalars—the CU.

Each element of the ALU array is not called by its generic name (ALU) but is called a PE. There are 64 PEs, and they are numbered from 0 to 63. Each PE responds to appropriate instructions if the PE is in an active *mode*. (There exist instructions in the repertoire which can activate or deactivate a PE.) Each PE performs the same operation under command from the CU in the lock-stepped manner of an array processor. That is, since there is only one CU, there is only one instruction stream and all of the ALUs respond together or are lock-stepped to the current instruction. If the current instruction is ADD for example, then all the ALUs will add—there can be no instruction which will cause some of the ALUs to be adding while others are multiplying. Every ALU in the array performs the instruction operation in this lock-stepped fashion, but the operands are vectors whose components can be, and usually are, different.

Each PE has a full complement of arithmetic and logical circuitry, and under command from the CU will perform an instruction "at-a-crack" as an array processor. Each PE has its own 2048 word 64-bit memory called a PE memory (PEM) which can be accessed in no longer than 350 ns. Special routing instructions can be used to move data from PEM to PEM. Additionally, operands can be sent to the PEs from the CU via a full-word (64-bit) one-way communication line and the CU has eight-word one-way communication with the PEM array (for instruction and data fetching).

An Illiac IV word is 64 bits, and data numbers can be represented in either 64-bit floating point, 64-bit logical, 48-bit fixed point, 32-bit floating point, 24-bit fixed point, or 8-bit fixed point (character) mode. By utilizing the 64-bit, 32-bit, and 8-bit data formats, the 64 PEs can hold a vector of operands with either 64,

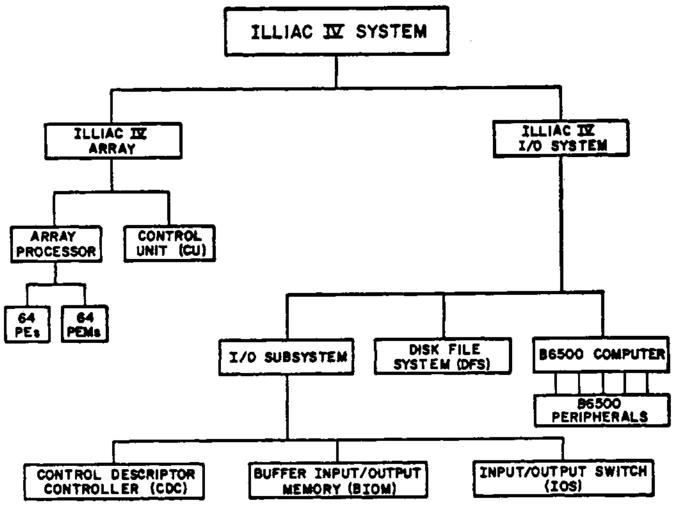


Fig. 4. Illiac IV system organization.

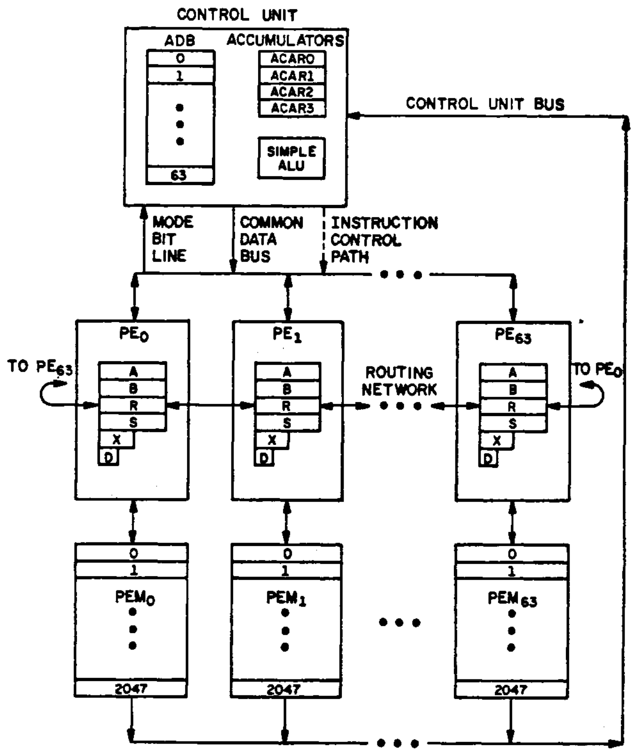


Fig. 5. Illiac IV array.

128, or 512 components. Since Illiac IV can add 512 operands in the 8-bit integer mode in about 66 ns, it is capable of performing almost 10^{10} of these "short" additions/s. Illiac IV can perform approximately 150 million 64-bit rounded normalized floating-point additions/s.

The I/O is handled by a B6500 computer system. The operating system, including the assemblers and compilers, also resides in the B6500.

B. The Illiac IV System

The Illiac IV system can be organized as in Fig. 4. The Illiac IV system consists of the Illiac IV array plus the Illiac IV I/O system. The Illiac IV array consists of the array processor and the CU. In turn, the array processor is made up of 64 PEs and their 64 associated memories—PEMs. The Illiac IV I/O system comprises the I/O subsystem, the disk file system (DFS), and the B6500 control computer. The I/O subsystem is broken down further to the CDC, BIOM, and IOS. The B6500 is actually a medium-scale computer system by itself.

The Illiac IV array will be discussed first, in a general manner, followed by two illustrative problems which indicate some of the similarities and differences in approach to problem solving using sequential and parallel computers. The problems also serve to illustrate how the hardware components are tied together. Finally, the Illiac IV I/O system is discussed briefly.

1) The Illiac IV Array: Fig. 5 represents the Illiac IV array—the CU plus the array processor.

a) CU: The CU is not just the CU that we are used to thinking of on a conventional computer, but can be viewed as a small unsophisticated computer in its own right. Not only does it cause the 64 PEs to respond to instructions, but there is a repertoire of instructions that can be completely executed within the resources of the CU, and the execution of these instructions is overlapped with the execution of the instructions which drive the PE array. Again, it is worthwhile to view Illiac IV as being two computers, one which operates on scalars and one which operates on vectors.

The CU contains 64 integrated-circuit registers called the ADVAST data buffer (ADB), which can be used as a high-speed scratch-pad memory. ADVAST is an acronym for advanced station and is one of the five functional sections of the CU. Each register of the ADB (D0 through D63) is 64 bits long. The CU also has 4 accumulator registers called ACAR0, ACAR1, ACAR2, and ACAR3, each of which is also 64 bits long. The ACARs can be used as accumulators for integer addition, shifting, Boolean operations, and holding loop-control information—such as the lower limit, increment, and upper limit. In addition the ACARs can be used as index registers to modify storage references within the memory section (PEM).

b) PE: Each PE is a sophisticated ALU capable of a wide range of arithmetic and logical operations. There are 64 PEs numbered 0 through 63. Each PE in the array has 6 programmable registers: the A register (RGA) or accumulator, the B register (RGB), which holds the second operand in a binary operation (such as ADD, SUBTRACT, MULTIPLY, or DIVIDE), the R or routing register (RGR), which transmits information from one PE to another, the S register (RGS) which can be used as temporary storage by the programmer, the X register (RGX) or index register to modify the address field of an instruction, and the D or mode register (RGD), which controls the active or nonactive status of each PE independently. The RGD determines whether a PE will be active or passive during instruction execution. Since this register is under the programmer's control, individual PEs within the array of 64 PEs may be set to enabled (active) or disabled (passive) status based on the contents of one of the other PE registers. For example, there are instructions which disable all PEs whose RGR contents are greater than their RGA contents. Only those PEs in an enabled state are able to execute the current instruction. All registers are 64 bits except RGX which is 16 bits, and RGD which is 8 bits.

c) PEM: Each PE has its own 2048-word 64-bits per word random-access memory. Each memory is called a PEM, and they are numbered 0 through 63 also. PE and PEM taken together are called a *processing unit* or PU. PE_i may only access PEM_i so that one PU cannot modify the memory of another PU. Information

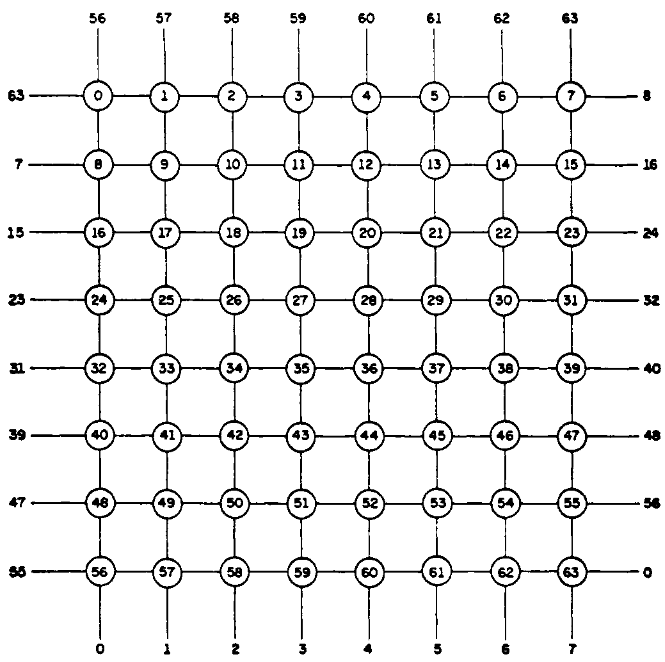


Fig. 6. PE routing connections.

can, however, be passed from one PU to another via the routing network, which is one of the 4 paths by which data flow through the Illiac IV array.

d) Data paths: There are four paths by which data flow through the Illiac IV array. These paths are called the CU bus, the common-data bus (CDB), the routing network, and the mode-bit line.

1) *CU bus:* Instructions or data from the PEMs in blocks of eight words can be sent to the CU via the CU bus. The instructions to be executed are distributed throughout the PEMs and are fetched in blocks of eight words to the CU via the CU bus as necessary. Although the operating system takes care of fetching and executing instructions, data can also be fetched in blocks of eight words under program control using the CU bus.

2) *CDB:* Information stored in the CU can be "broadcast" to the entire 64 PE array simultaneously via the CDB. A value such as a constant to be used as a multiplier need not be stored 64 times in each PEM; instead this value can be stored within a CU register and then broadcast to each enabled PE in the array. In addition the operand or address portion of an instruction is sent to the PE array via the CDB.

3) *Routing network:* Information in one PE register can be sent to another PE register by special routing instructions. (Information can be transferred from PE register to PEM by standard LOAD or STORE instructions.) High-speed routing lines run between every RGR of every PE and its nearest left and right neighbor (distances of -1 and $+1$, respectively) and its neighbor 8 positions to the left and 8 positions to the right (-8 and $+8$, respectively). Other routing distances are effected by combinations of routing -1, $+1$, -8, or $+8$ PEMs; that is, if a route of 5 to the right is desired, the software will figure out that the fastest way to do this is by a right route of 8 followed by three left routes of 1. Fig. 6 shows one way to view the connectivity which exists between PEs. As can be seen from the figure, PE_0 is connected to PE_{56}, PE_1, PE_8, and PE_{63}.

4) *Mode-bit line:* The mode-bit line consists of one line coming from the RGD of each PE in the array. The mode-bit line can transmit one of the eight mode bits of each RGD in the array up to an ACAR in the CU. If this bit is the bit which indicates whether or not a PE is on or off, we can transmit a "mode pattern" to an ACAR. This mode pattern reflects the status or on-offness of each PE in the array; then there are instructions which are executed completely within the CU that can test this mode pattern and branch on a zero or nonzero condition. In this way branching in the instruction stream can occur based on the mode pattern of the entire 64-PE array.

2) *Some Illustrative Problems*

a) Adding two aligned arrays: Let us first consider the problem of adding two arrays of numbers together. The Fortran statements for a conventional computer might look like:

$$\text{DO } 10\ I = 1,\quad N$$
$$10\ A(I) = B(I) + C(I).$$

The two Fortran instructions are compiled to a set of machine-language instructions which include initialization of the loop, looping instructions, and the addition of each element of the B array to the proper element in the C array, and storage to the A array. Except for the initialization instructions, the set of machine-language instructions is executed N times. Therefore, if it takes M μs to pass once through the loop, it will take about N times M μs to perform the above Fortran code.

Now suppose the same operations are to be performed on Illiac IV. Arrangement of the data in memory becomes a primary consideration—the data must be arranged to exploit the parallelism of operation of the PEs as effectively as possible. The worst way to use the PEs would be to allocate storage for the A, B, and C arrays in just one PEM. Then instructions would have to be written just as they were in a conventional machine to loop through an instruction set N times.

Let us consider the problem as consisting of three cases: $N=64$, $N<64$, and $N>64$, and then see what each case entails in terms of programming for Illiac IV.

1) *$N=64$:* To reflect the case where $N=64$, we have arranged the data as shown in Fig. 7. In order to execute the two lines of Fortran code, only the three basic Illiac IV machine-language instructions are necessary: 1) LOAD all PE Accumulators (RGA) from location $\alpha+2$ in all PEMs. 2) ADD to the PE Accumulators (RGA) the contents of location $\alpha+1$ in all PEMs. 3) STORE result of all PE Accumulators to location α in all PEMs.

Since every PE will execute each instruction *at the same time* or in parallel, accessing its own PEM when necessary, the 64 loads, additions, and stores will be performed while just three instructions are executed. This is a speedup of 64 times for this case, in execution time.

The three instructions to perform the 64 additions in Illiac IV assembly language (Ask) would actually look like:

```
LDA    ALPHA + 2;
ADRN   ALPHA + 1;
STA    ALPHA;
```

(note that since each instruction operates on a vector, a memory location can be considered a *row* of words rather than a single word).

2) *$N<64$:* Since there are exactly 64 PEs to perform calculations, a proper question is: what happens if the upper limit of the

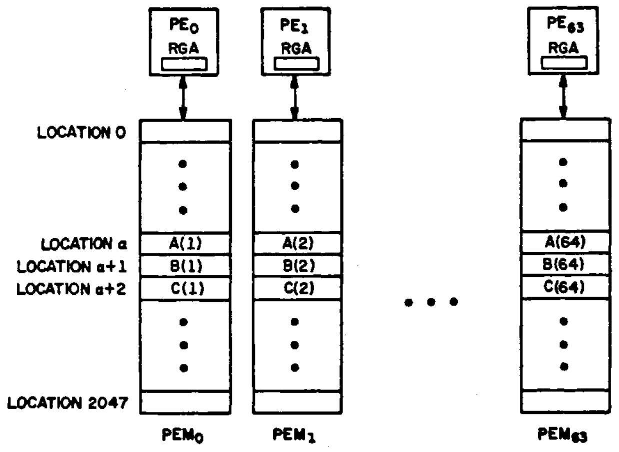


Fig. 7. Arrangement of data in PEM to accomplish
DO 10 $I = 1, 64$
10 $A(I) = B(I) + C(I)$.

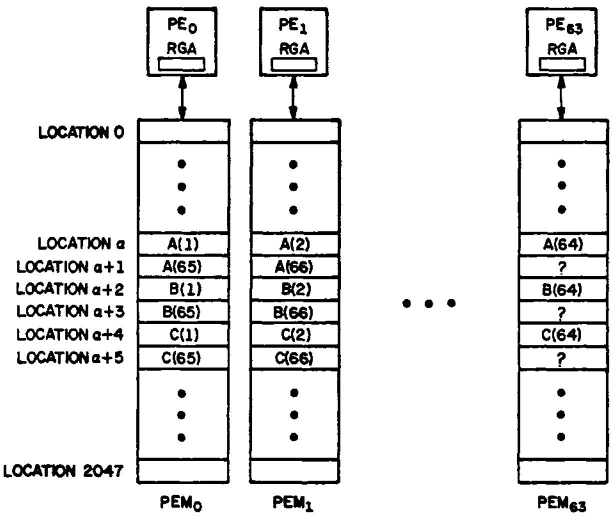


Fig. 8. Arrangement of data in PEM to accomplish
DO 10 $I = 1, 66$
10 $A(I) = B(I) + C(I)$.

loop is not exactly equal to 64? If the upper limit is less than 64, there is no problem other than that the total PE array will not be utilized.

The tradeoff the potential user of Illiac IV must consider here is how much (or how often) is Illiac IV underutilized? If the underutilization is "too much" then the problem should be considered for running on a conventional computer. However, the user should keep in mind that he usually does not feel too guilty if he underutilizes the resources of a conventional system—he does not use every tape drive, every bit of available core, every printer, and every byte of disk space for most of his conventional programs.

3) *$N>64$:* When the upper limit of the loop is greater than 64, the programmer is faced with a storage allocation problem. That is, he has various options for storing the A, B, and C arrays and the program he writes to perform the 2 Fortran statements will vary considerably with the storage allocation scheme chosen. To illustrate this let us consider the special case where $N=66$ with the A, B, and C arrays stored as shown in Fig. 8.

To perform the 66 additions on the data stored as shown in Fig. 8, *six* Illiac IV machine-language instructions are now necessary:

LOAD RGA from location $\alpha+4$.
ADD to RGA contents of location $\alpha+2$.
STORE result to location α.
LOAD RGA from location $\alpha+5$.
ADD to RGA contents of location $\alpha+3$.
STORE result to location $\alpha+1$.

The addition of two more data items to the A, B, and C arrays not only necessitates extra Illiac IV instructions but complicates the data storage scheme. In this instance, the programmer might as well DIMENSION the A, B, and C arrays to 128 as 66. Note that the particular storage scheme shown in Fig. 8 wastes almost 3 rows of storage (186 words). The storage could have been packed much closer so that $B(1)$ followed $A(66)$ in PE_2 of row $\alpha+1$, *but* the program to add the arrays together would have to do much more shuffling to properly align the arrays before adding. An Illiac IV program is highly dependent on the storage scheme chosen.

b) Uncoupling sequential code: Finally let us consider the Fortran code:

$$\text{DO } 10\ I = 2,\ 64$$
$$10\ A(I) = B(I) + A(I-1).$$

How would we do the above instructions on a parallel computer such as Illiac IV? At first, it appears we cannot perform the above algorithm on Illiac IV because it is inherently sequential. If we recognize that the 2 Fortran statements above are only a shorthand for 63 Fortran statements:

$$\begin{aligned} A(2) &= B(2) + A(1) \\ A(3) &= B(3) + A(2) \\ &\vdots \\ A(63) &= B(63) + A(62) \\ A(64) &= B(64) + A(63) \end{aligned}$$

and that each of the 63 statements is executed sequentially, we see that each statement in the sequence relies on the result computed from the previous statement. That is, $A(3)$ cannot be computed until the statement above it has computed $A(2)$. Therefore, the 63 additions cannot be done in parallel if we literally try to apply the 2 Fortran statements as they stand. However, using mathematical subscript notation:

$$\begin{aligned} A_2 &= B_2 + A_1 \\ A_3 &= B_3 + A_2 = B_3 + B_2 + A_1 \\ A_4 &= B_4 + A_3 = B_4 + B_3 + B_2 + A_1 \\ &\vdots \\ A_N &= B_N + B_{N-1} \cdots B_2 + A_1. \end{aligned}$$

We see that the elements of the A array can be computed independently using the formula

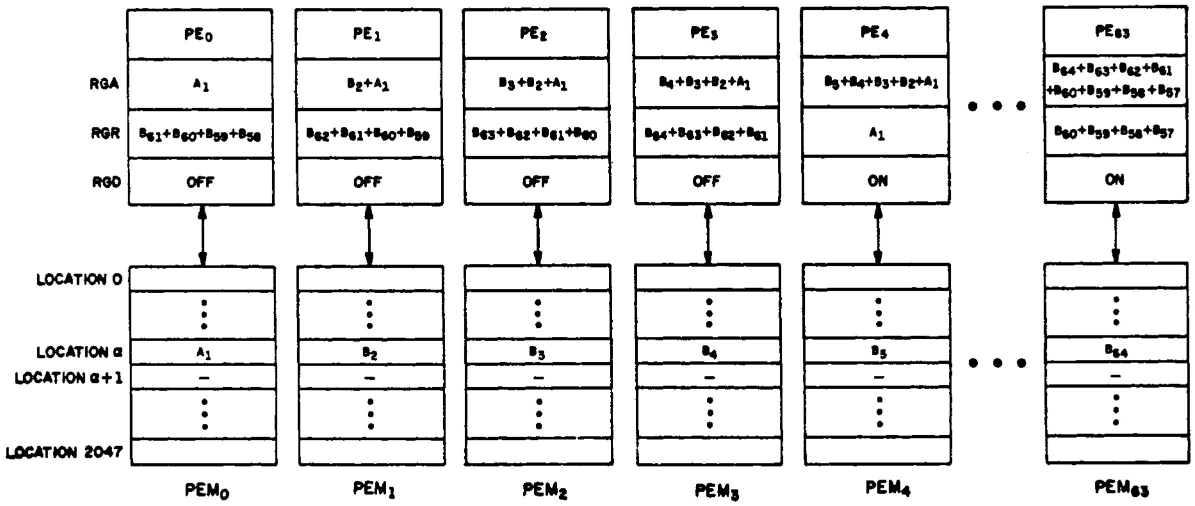


Fig. 9. Status of data in PEM, RGA, RGR, and mode status (RGD) while executing

DO 10 $I = 2, 64$

10 $A(I) = B(I) + A(I - 1)$.

The mode status (RGD) and the contents of PEM, RGA, and RGR are shown after step 8) ($i=2$) of the program.

$$A_N = A_1 + \sum_{i=2}^{N} B_i, \qquad \text{for } 2 \le N \le 64.$$

The Fortran code to perform the above formula would be:

$S = A(1)$

DO 10 N = 2,64

$S = S + B(N)$

10 $A(N) = S$.

The above Fortran code is equivalent to the original code (its end results are the same) but now the computation of the A array has been *decoupled* so that each value of A in the array can be computed *independently*.

An arrangement of data to effect this program is shown in Fig. 9 and the program might be as follows.

1) Enable all PEs. (Turn ON all PEs.)

2) All PEs LOAD RGA from location α.

3) $i \leftarrow 0$.

4) All PEs LOAD RGR from their RGA. [This instruction is performed by *all* PEs, whether they are ON (enabled) or OFF (disabled).]

5) All PEs ROUTE their RGR contents a distance of 2^i to the right. (This instruction is also performed by all PEs, regardless of whether they are ON or OFF.)

6) $j \leftarrow 2^i - 1$.

7) Disable PEs number 0 through j. (Turn them OFF.)

8) All enabled PEs ADD to RGA the contents of RGR. (Fig. 9 shows the state of RGR, RGA, and RGD (the mode status)—which PEs are ON and which are OFF—after this step has been executed when $i=2$.)

9) $i \leftarrow i+1$.

10) If $i<6$ go back to step 4), otherwise to the step 11).

11) Enable all PEs.

12) All PEs STORE the contents of RGA to location $\alpha+1$.

Note that this same algorithm can be applied to the solution of problems where the recurrence is of the form: $F_i = C_i * F_{i-1}$ which decouples to $F_N = (\prod_{i=2}^{N} C_i) F_1$. All that need be done is that step 8) be changed to MULTIPLY rather than ADD. Note also that if $C_i = i\ (i=1, 2, \cdots, 64)$ and $F_1 = 1$ we have an algorithm for computing $N!$ on Illiac IV; that is, when the algorithm is complete PE$_N$ will contain $(N+1)!$

This example tries to illustrate that it is not always immediately clear if an algorithm can be decoupled so that it can operate in parallel, or is so dependent on what happened before that it can only be executed sequentially. In this example, it appears that the algorithm is sequential, but upon closer inspection, the parallelism appears. Potential Illiac IV users will probably need much practice in analyzing problems using a parallel viewpoint, especially if they have already been conditioned to viewing their problems only in terms of solving them on a sequential conventional computer. The tool, for better or for worse, shapes the uses it is put to.

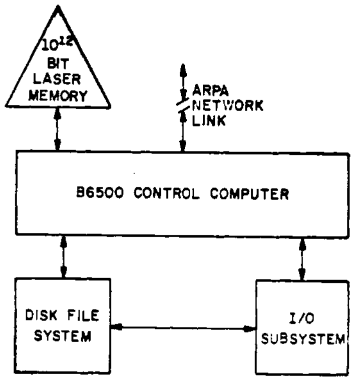


Fig. 10. Illiac IV I/O system.

3) Illiac IV I/O System: The Illiac IV array is an extremely powerful information processor, but it has of itself no I/O capability. The I/O capability, along with the supervisory system (including compilers and utilities), resides within the Illiac IV I/O System. The Illiac IV I/O system (see Fig. 10) consists of the I/O subsystem, a DFS, and a B6500 control computer (which in turn

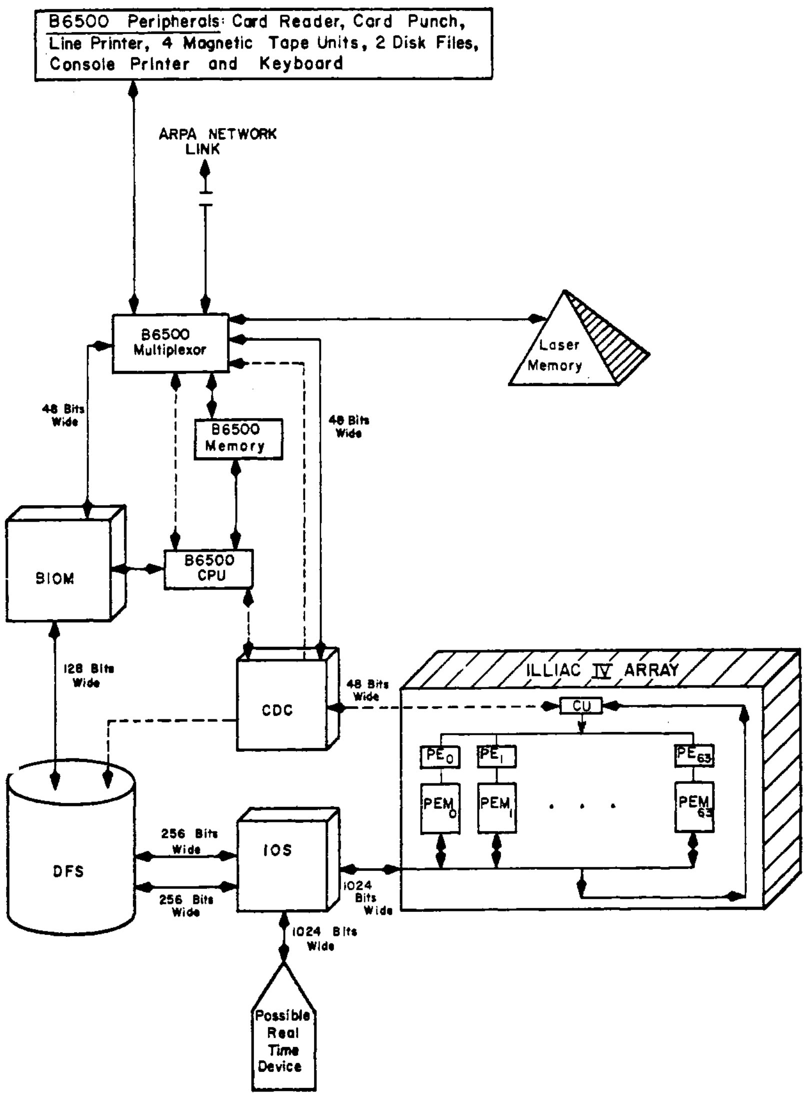


Fig. 11. Illiac IV system.

supervises a large laser memory and the ARPA network link). The total Illiac IV system consisting of the Illiac IV I/O system and the Illiac IV array is shown in Fig. 11. All system configurations shown are transitory, and more than likely will have changed several times in the next year or so.

a) I/O subsystem: The I/O subsystem consists of the control descriptor controller (CDC), the buffer I/O memory (BIOM), and the I/O switch (IOS).

1) *CDC:* The CDC monitors a section of the CU waiting for an I/O request to appear. The CDC can then interrupt the B6500 control computer which can, in turn, try to honor the request and place a response code back in that section of the CU via the CDC. This response code indicates the status of the I/O request to the program in the Illiac IV array.

The CDC causes the B6500 to initiate the loading of the PEM array with programs and data from the Illiac IV disk (also called the DFS). After PEM has been loaded, the CDC can then pass control to the CU to begin execution of the Illiac IV program.

2) *BIOM:* The B6500 control computer can transfer information from its memory through its CPU at the rate of 80×10^6 bits/s. The Illiac IV DFS accepts information at the rate of 500×10^6 bits/s. This factor of over six in information transfer rates between the two systems necessitates the placing of a rate-smoothing buffer between them. The BIOM is that buffer. A buffer is also necessary for the conversion of 48-bit B6500 words to 64-bit Illiac IV words which can come out of the BIOM two at a time via the 128-bit wide path to the DFS. The BIOM is actually four PE memories providing 8192 words of 64-bit storage.

3) *IOS:* The IOS performs two functions. As its name implies, it is a switch and is responsible for switching information from either the DFS or from a port which can accept input from a real-time device. All bulk data transfers to and from the PEM array are via IOS. As a switch it must ensure that only one input is sending to the array at a given time. In addition, the IOS acts as a buffer between the DFS and the array, since each channel from the Illiac IV disk to the IOS is 256 bits wide and the bus from the IOS to the PEM array is 1024 bits wide.

b) DFS: The DFS consists of two storage units, two electronics units and two disk file controllers. The DFS is also called the Illiac IV disk or simply, the Disk. The Disk is of 10^9-bit capacity, having 128 heads, with one head per track. The DFS has two channels, each of which can transmit or receive data at a rate of

0.5×10^9 bits/s over a path 256 bits wide; however, if both channels are sending or receiving simultaneously the transfer rate is 10^9 bits/s.

c) B6500 control computer: The B6500 control computer consists of a central processing unit (CPU), a memory, a multiplexor, and a set of peripheral devices (card reader, card punch, line printer, 4 magnetic tape units, 2 disk files and a console printer, and a keyboard). It is the function of the B6500 to manage all programmers' requests for system resources. This means that the operating system will reside on the B6500. All compiling and assembling of programs is also performed on the B6500. Utilities, such as card-to-disk, card-to-tape, etc., are also executed on the B6500. From a total system standpoint, the Illiac IV array can be considered as a special-purpose peripheral device of the B6500 capable of solving certain classes of problems with extremely high speed.

1) *Laser memory:* The B6500 supervises a 10^{12}-bit write-once read-only laser memory developed by the Precision Instrument Company. The beam from an argon laser records binary data by burning microscopic holes in a thin film of metal coated on a strip of polyester sheet, which is carried by a rotating drum. Each data strip can store some 2.9 billion bits. A "strip file" provides storage for 400 data strips containing more than a trillion bits. The time to locate data stored on any one of the 400 strips is 5 s. Within the same strip data can be located in 200 ms. The read and record rate is four million bits per second on each of two channels. A projected use of this memory will allow the user to "dump" large quantities of programs and data into this storage medium for leisurely review at a later time; hard copy output can optionally be made from files within the laser memory.

2) *ARPA network link:* The ARPA network is a group of computer installations separated geographically but connected by high-speed (50 000 bits/s) data communication lines. On these lines, the members of the "net" can transmit information—usually in the form of programs, data, or messages. The link performs an information switching function and is handled by an interface message processor (IMP) and a network control program stored within each member installation's "host" computer. Each IMP operates in a "store and forward mode," that is, information in one IMP is not lost until the receiving IMP has signalled complete reception and retention of the message. The IMP interfaces with each member's computer system and converts information into standard format for transmission to the rest of the net. Conversely, the IMP accepts information in a standard format and converts it to the particular data format of the member installation. In this way, the ARPA network is a form of a computer utility with each contributing member offering its unique resources to all of the other members. The Illiac IV system then is an ARPA network resource that will be shared by the members of the ARPA network; even the host site of the Illiac IV, Ames Research Center at Moffett Field, Calif., will be constrained to access Illiac IV via the ARPA network.

III. Software

A. Introduction

It should be remembered that the Illiac IV project was initially directed toward experimenting with the feasibility of building a massive hardware configuration and most of the software described here (defined as operating system, compilers, debugging aids, and necessary library functions) could have been developed with suitably sophisticated simulators, without any reference to a "real" Illiac IV; indeed, most of the truly innovative software envisioned has yet to be built. This Section, then, is devoted primarily to a discussion of a sound software strategy rather than to a minutely detailed description of the initial software [6]–[8].

B. Software Strategy

The main reason for building Illiac IV was to provide a facility of massive computing power especially suited to the solution of partial differential equations and matrix manipulation. A considerable amount of this work was already being done on less powerful serial machines by users who demanded execution speed at almost any price and who used their machines largely in batch-processing mode (as opposed to time sharing), relying very much on punch-card input devices and magnetic tape and line printer for intermediate storage and for printing final results. It was estimated that users would utilize a substantial proportion of Illiac IV time during its first year of operation in debugging and refining their programs or "code," thus justifying the creation of the machine. Nearly all these users desired that languages and operating systems provide machine efficiency rather than "ease of use" or "programmability." Many of the codes for existing Illiac IV applications have evolved over the last ten years and have displayed a remarkable architectural similarity. They comprise the following three parts:

1) A "preprocessor" section, wherein the problem is initialized and decimal-to-binary conversion (reading data) is performed. This is usually a serially oriented section.

2) A "kernel" where the main problem is addressed, usually inherently parallel, and therefore considered a job for Illiac IV. The kernel occupies between 5 percent and 10 percent of the source and object code, and on serial machines uses 80 to 95 percent of the time.

3) A "postprocessor" section, wherein results are stored on archive files, necessary binary-to-decimal (writing data) conversions are made, graphs are plotted, and line printer output formatting is set. This is usually a serially oriented section. From here control may loop back to the kernel in order to complete a further set of iterations on the data.

Generally speaking, Illiac IV jobs will be presented to the B6500 as card decks, tape files, or as files received over the ARPA network [9]. B6500 disk files which have originated from one of these sources, but have been edited through local or remote on-line consoles, may also be presented as Illiac IV jobs.

Results produced by B6500 or Illiac IV programs may be printed in the conventional manner locally, displayed on local or remote on-line consoles, or transmitted over the ARPA network to output devices local to the user. Later, it is expected that provision for microfilm graphics and selective viewing and editing of results will be made available locally.

However, because of the high disk latency (40 ms) compared with processor speed, it should be remembered that the Illiac IV hardware, as it stands, is not particularly amenable to a "time-sharing" operation if "time-sharing" implies "time-slicing." All the usual interactive and debugging facilities will be provided on the B6500.

From the intended user's point of view, then, it seemed adequate initially to provide simple batch-processing software that would enable jobs to be run efficiently, even though only one job kernel would be active at a time. This attitude was reenforced by questions of reliability. The initial mean time between failures for the Illiac IV array was envisioned to be 2–4 h. Debugging "state-of-the-art" software on untried "state-of-the-art" hardware demanded either more bravery or more foolhardiness than most ordinary mortals were prepared to volunteer. On the other hand, the B6500 promised to be much more reliable and offered Burroughs Algol as an implementation language.

The wisest course was clearly to aim for simple batch-processing systems software executed as much as possible on the B5500. As experience leads to greater familiarity and confidence with the array hardware and initial software performance, those modules whose operation is inherently parallel could be moved onto Illiac IV if the B6500 became overloaded, or if it seemed otherwise advisable to do so.

One of the first software tasks was to write an Illiac IV simulator that would handle all operations in the CU and PEs and all Illiac IV machine instructions. Although this simulator ran approximately a million times more slowly than Illiac IV on a B6500, and later, about 200 000 times more slowly on a B6500, it became the basis for all language and algorithm development for Illiac IV. Recently, the simulator was extended to simulate Illiac IV I/O subsystem logic and its interaction with the Illiac operating system. At the time of writing, potential users may "run Illiac IV jobs" on this complete simulator as if they were using the complete Illiac IV installation, although the length of the Illiac IV program must, of course, be of limited execution time. This simulated operating system is built so that it may be incrementally transferred onto the real Illiac IV hardware. The present software will be operational very shortly after the hardware is built and working.

It is also worth noting that all Illiac IV languages except the Assembler, but including the Illiac control language (the operating-system control-card language), have been implemented using a compiler–compiler system called TWST [10]. This has allowed a certain amount of language experimentation, and has resulted in the early availability of usable languages despite the fact that the team providing the basic Illiac IV software included never more than fourteen professionals. This relatively small team has provided a coherent set of software, although the relatively mechanized approach has caused some degradation in compile-time speed for instance. Consequently, there is a considerable amount of refinement to be done. However, this will not affect the users, whose programs will not have to be changed while the overall compiler efficiency increases.

C. Operating System

The Illiac IV operating system [11] operates in a "diagnostic" or "normal" mode. The main task of the diagnostic mode is the testing and diagnosis of possible faults in the Illiac IV I/O subsystem and the Illiac IV array itself. These diagnostic programs are designed to identify faults and to automatically identify the pluggable unit in which they occur. The unit is then replaced and the operating system automatically reruns the test program that identified the fault to ensure that it has in fact been remedied. While in this mode, an interactive routine is available to the engineers to enable them to either call specific diagnostic programs or generate new ones. This system may also be used to interrogate Illiac IV registers and to change their values. While in the diagnostic mode, the B6500 is available to carry on its usual work of preparing jobs to be run on Illiac IV and to process the output from those which have already run. However, it cannot use any major unit that is being diagnosed at the time by the diagnostic routines. In the "normal" mode, the operating system administers the running on Illiac IV and the use of the I/O subsystem.

The Illiac IV operating system consists of a set of asynchronous processes which run under the control of the B6500 master control program (MCP). When a user submits an Illiac IV job to the B6500, it usually consists of the following parts:

1) B6500 programs usually written in B6500 Algol or Fortran, which transform and prepare binary input files (input under format control, i.e., character-to-binary conversion) to be used by the Illiac IV program called "Preprocessor."

2) Illiac IV programs usually written in Ask, Glypnir, or Illiac IV Fortran, which use Illiac IV to operate on the files prepared by the B6500 programs and to prepare binary output files ("Kernel").

3) B6500 programs usually written in B6500 Algol or Fortran which transform binary files from Illiac IV to the required external form for use or storage ("Postprocessor").

4) An Illiac control language (ICL) program which defines the job. The ICL controls the operating system for the job which it defines, and thus may be seen as "driving" the operating system for that particular job, although the operating system may be operating on several ICL programs concurrently.

As each job enters the system through the B6500, it is assigned a priority (if one has not been specified by the user). Higher priority jobs, as well as being processed more favorably by the B6500, have the privilege of having their Illiac IV program parts preempting running Illiac IV programs of lower priority. The preempting program will run to completion unless it is itself preempted by a program of higher priority. The number of stacked-up preempted jobs may be arbitrarily set when the system is initialized. However, the main function of the preemption scheme is to allow debugging runs and short production runs to move along the job stream quite rapidly, while longer production runs and jobs of lower priority ensure that the Illiac IV itself is rarely idle for any length of time. If the work load permits, non-Illiac IV jobs may be run on the B6500 while it is not compiling, preprocessing, postprocessing, and administering Illiac IV jobs.

The B6500 programs and Illiac IV programs communicate via Illiac IV disk files (for data) and the 48-bit path through the TMU of the CU (interrupt signals). The protocol for these signals over the 48-bit path is administered by two modules. The first is a small executive program residing in Illiac IV itself (called OS4) which processes all interrupts for the array, handles all communications between the user program and the rest of the operating system, and provides a few standard functions for use in the array. OS4 communicates with a module (known as the "job partner") in the B6500, which acts as a clearing house for all communication between OS4 and thus the user program running on Illiac IV. The job partner thus initiates all data transfers between the B6500 and Illiac IV, B6500 and Illiac IV disk, and between Illiac IV and Illiac IV disk. This arrangement emphasizes the rate of the B6500 as an I/O computer for the Illiac IV or, conversely, the Illiac IV as a peripheral for the B6500.

ICL is used to coordinate the execution and communication of the set of B6500 and Illiac IV programs that constitute an Illiac IV job. ICL is an Algol-like block-structured language in which FILES, PROGRAMS, and INTEGERS may be declared.

FILES may be declared in formats and number representations appropriate to either machine. Additionally, Illiac IV files may be laid out on the Illiac IV disk in a very flexible manner to allow the user to maximize their availability as they spin on the disk. When files are moved from one machine to another, i.e., when Illiac IV files are equated to B6500 files, they are automatically transformed to the appropriate format and binary representation.

PROGRAMS may be declared as Illiac IV programs, B6500 compilers, job partners, ICL programs, or other B6500 programs. A program may be viewed as acting much like as INTEGER PROCEDURE in Algol with monstrous side effects. The files which the program uses for data and results and the file on which the program is to be found are also included in the declaration.

INTEGERS are used mainly for manipulating a single-integer result always delivered by a program when it is called by the ICL program. This integer may be set by the user, and normally indicates the degree of success of the program execution.

The availability of IF and CASE statements allows considerable flexibility in job control, and facilities are also provided to allow simultaneous execution of interlocking B6500 and Illiac IV programs within the same job.

The ICL allows the user to construct a simple operating system appropriate to each of his jobs. The Illiac IV operating system then allows each of these "job-operating systems" to overlap each other in their utilization of time and resources. It also encourages the division of the job into manageable modules, thus making it more resilient to hardware and software malfunction.

The present operating system can undergo considerable development with the present hardware configuration. Apart from simply enhancing the existing code and making it part of the Burroughs MCP, many of the file-conversion modules and the disk allocator, whose operations are inherently parallel, may be moved to the Illiac IV array. Preparation for this work is well underway. Major changes in operating philosophy, however, are best left until hardware performance has been evaluated in the real world.

D. Languages

1) Background: The strategy adopted in language development for Illiac IV was more liberal than that adopted for the operating system. Apart from the assembler, three "higher level languages" were attempted. Two survived to be usable. The problem of defining requirements for the languages was more difficult than that of the operating system, where user habits and needs were fairly well understood.

The first major difficulty is that of formal representation. Fortran, Algol, and other numerically oriented higher level languages are based primarily upon traditional mathematical formalism. However, there is no comparably suitable formalism that can be easily used as a basis for describing the kind of operations that Illiac IV does easily and, at the same time, can be extended to a truly nontrivial set of array or matrix operations. For while vector and matrix operations such as addition and multiplications by a scalar quantity are trivial to implement, matrix multiplication and inversion require reasonably sized subroutines. In addition, in the field of partial differential equations the large number of treatments of boundary conditions, discontinuities, and other special cases have presented a variety of approaches that has only recently been attacked from the higher level language point of view [12].

The second problem is that Illiac IV can be fiendishly difficult to program properly if one does not banish nearly all serial machine preconceptions and habits. Realizing this, the language designer is faced with choosing between completely disguising the architecture of Illiac IV, thus helping the programmer by doing quite a lot of dirty housekeeping for him, or fearlessly exposing the architecture of Illiac IV, thus in all probability, forcing the programmer to rethink his algorithm in Illiac IV terms. The first approach was applied in the design of the Tranquil programming language, the second in Glypnir. The nature and success of each project will be discussed later.

The third problem was the unexpected amount of code that had to be produced for any given problem, and the difficulty of optimizing this code automatically. A study of Illiac IV assembler codes indicates that for every arithmetical operation (including fetch and store), at least two others were required to direct and define the scope of the arithmetic instructions. About half of these extra instructions were CU instructions setting up loops, constants, administering ADB storage, and storing stacks of mode patterns. The other half were concerned with mode-pattern calculation. Thus it was difficult to see what efficient code generation meant for Illiac IV, and to make allowance for it, especially in view of the fact that Tranquil and Glypnir were being implemented before the Illiac IV instruction code had been finalized.

In this latter respect, it may be interesting to note that all compilers were designed to translate source code into Illiac IV assembler code. This will make the compilers themselves more resilient in the face of machine modification, and also, since neither of the higher level languages do any automatic optimization, give the enthusiastic user a chance to optimize his own code should he wish to do so.

The Illiac IV assembler [13] is fairly conventional as assemblers go nowadays. It has a very sophisticated macro-definition facility which may be used to include standard I/O facilities and other communications with OS4. It also has "pseudo operations" to help with storage layout and PE address allocation, of which there are four types: syllable (half-word) addresses, word addresses, row addresses, and I/O word addresses. All of these may be used by some instructions.

Before embarking upon a more detailed discussion of the Illiac IV higher level languages, it is worth pointing out that their differences with existing languages center around the following subjects.

1) Storage allocations in a two-dimensional store. The "natural" method of addressing PEM is by row (or 64 words). Single words of PEM may be addressed individually. However, a "column," that is, a group of words in the memory of a single PE, may not be addressed as a group.

2) The expression of parallelism and mode control. To a certain extent, a vector may be the natural expression of parallelism, and Illiac IV languages should in some sense be able to allow operations on vectors or rows of matrices. The length of the vector and the elements within it to be operated upon are defined by the mode pattern. Illiac IV languages should allow efficient and comprehensive calculation and manipulation of mode patterns.

3) The expression of routing and indexing. The language should allow reasonable expression of routing, and of indexing by a different amount in each PE.

2) Tranquil: The first Illiac IV language attempted was "Tranquil" [14], an Algol-like language entirely independent of machine organization. It was designed to allow programmers to operate on arrays of data in a simultaneous way. An algorithm may be viewed as being applied to any set of elements of an array at the same logical time, thus often allowing a programmer to avoid thinking about such details as how to index through an array and which elements to save in temporary arrays.

The Tranquil compiler was brought to a fairly advanced state of completion when experience with the object code indicated that the overhead involved in completely masking the machine's architecture was too high in relation to the users' demand for fast execution of their programs. This was not just a matter of the payoff between the cost of programming time and execution time, but of making existing programs, whose execution times for proposed problems were unfeasibly long on existing equipment, run in a realistic time on the Illiac IV. Users were thus prepared to do a certain amount of reprogramming in, say, Fortran and to restructure their algorithms to suit Illiac IV architecture, but they were unwilling to reprogram the whole problem in Illiac IV assembler.

As it seemed unlikely that Tranquil could be made sufficiently efficient before the machine was then projected to be delivered, work on Tranquil was halted, and the job of providing an extended Fortran was begun. However, Tranquil should not be shrugged off. It represents the level of language that implementors would like to provide. Continued experience with extended Fortran has shown, moreover, that the inefficiencies of Tranquil could have been remedied with time, and that with slight modification to the language, Tranquil could have become a viable and extremely interesting and useful Illiac IV language.

3) Glypnir: Glypnir [15], [16] is also an Algol-like block structured language and in many ways an Illiac IV equivalent of Burroughs B6500 Algol in the sense that it was written to allow the knowledgeable user to exploit Illiac IV architecture to considerable advantage.

Program data types or procedures may be declared as being either CU or PE variables. In the first case, they refer to one 64-bit value corresponding to an Illiac IV word, and in the second case, they refer to a row of Illiac IV words and thus may have up to 64 different values simultaneously. Boolean values, however, are neither CU or PE variables but refer to 64 true–false values. Thus a Boolean variable is said to be TRUE when all its constituent bits are TRUE and FALSE when all its constituents bits are FALSE. Two extra operators, SOME and EVERY, are used to provide a bridge between word and bit level logic.

All arithmetic operations are carried out under the control of a MODE pattern. The MODE allows the 64 true–false values of a Boolean to be associated with each of the PEs. When a bit of the MODE pattern is TRUE, the corresponding PE is enabled and may thus deliver the results of an operation.

The Glynpir expression $A := B * C$;, where A, B, and C are PE variables, means that each value of A is multiplied by C and delivered to A for each enabled PE. If C were a CU variable, it would be repeated 64 times in an invisible PE variable before the statement was evaluated.

Special facilities exist to allow the rotation and shifting of rows to the right and left, in a way similar to the more familiar operations conventionally carried out on words, thus allowing the "route" instruction to be utilized.

FOR and IF statements are also provided, but often give unconventional results. For instance, given PE variables A, B, and C, the statement

$$\text{IF } A > B \text{ THEN } C := A \text{ ELSE } C := B$$

will deliver the maximum elements of A and B to C, and may result in both the THEN and ELSE statements being executed.

Glypnir will alow the explicit inclusion of blocks of assembler language for the optimization of any section of code, and also has facilities within the language to refer explicitly to selected hardware registers for those who wish to optimize without going into assembler language.

The language has now been in use with the Illiac IV simulator for over a year and a considerable amount of useful experience has been gained from it. Projected future developments include the addition of a 32-bit mode and the facility to allow binding to separately compiled assembler, Glypnir, or Illiac IV Fortran subroutines.

4) Illiac IV Fortran: Illiac IV Fortran [17] was provided more for the general Illiac IV user than for the specialist. Glypnir demands that the programmer undertake the detailed supervision of storage allocation and be constrained to think in terms of Illiac IV rows or vectors of Illiac IV rows. Illiac IV Fortran allows the user to think in terms of rows of any length in "straight" and "skewed" storage (see Section IV-B). Skewed storage allows rows and columns to be accessed with equal facility.

In order to provide mode control, the data type BINARY has been added to the language which allows arbitrary arrays of bits (or true–false values) to be declared in a way similar to arrays of LOGICAL values.

The DO statement has been extended to allow parallel execution of arithmetic expressions, and extra constructs have been added to the language to allow the shifting and rotation of vectors and array rows.

The only significant change in definition applies to EQUIVALENCE and COMMON statements, where the two-dimensional store of Illiac IV imposes restrictions on the usual serial definition. However, as these restrictions are those usually forced upon the programmer by "real" problems, they may often be a help rather than a hindrance.

One very important option allowed in Illiac IV Fortran is that of taking an existing serial Fortran code with suitably adjusted I/O statements and running it on Illiac IV serially (in one PE) to test its validity before incrementally parallelizing it.

An Illiac IV Fortran-to-Glypnir translator has now been in use for several months and is considered to be very successful. Illiac IV Fortran exhibits many of the operational features of Tranquil, and has benefitted from both the Tranquil and the Glypnir experiences.

5) Some Other Considerations: Because of the large disk latency compared with processing speed, the I/O software for Illiac IV languages tends to be elaborate in order to enable the programmer to synchronize his I/O functions with his computing. A considerable amount of disk latency may be removed by using the elaborate disk-layout facilities provided by the operating system. This latency may be reduced further by the I/O intrinsics in both Glypnir and Illiac IV Fortran.

File declarations are modeled on those of the Burroughs B6500, with the option of implicit buffering to a level given by the user. The user is also given the option of driving his program by I/O interrupt. At a lower level, standard assembler macros provide a similar facility for the assembler programmer.

In terms of what could be done, and, by analogy, has been done in producing software for some serial machines, Illiac IV software development is still very much in its infancy. While the present software is more than adequate to get a considerable amount of useful work out of Illiac IV for a few years after its completion there are several areas where new work would produce valuable payoffs.

Illiac IV offers hardware facilities for a "32-bit mode," that is, the ability to store and operate on two 32-bit numbers of differing value stored in one Illiac IV word. None of the present compilers offer a 32-bit mode as a complete data type that can be used in harmony with the 64-bit mode. Work in this area is continuing at the University of Illinois, and consists not only of inserting the necessary language constructions into the compilers and developing the necessary standard algorithms, but also bringing 32-bit user experience to some sort of parity with 64-bit user experience. The effective use of a 32-bit mode may essentially double the power of Illiac IV, but much remains to be done before patterns of effective use are established.

The automatic overlaying of array and program segments is also particularly important in view of the relatively small core-storage capacity of the Illiac IV. This, of course, impinges upon the operating system as well as the compilers, and although automatic overlay has always been desirable, its detailed investigation has always, quite rightly, been postponed until sufficient experience with existing hardware sheds more light on its feasibility. At present, the same effect may be gained by coupling program modules together with a suitable Illiac control program, or by explicitly overlaying file segments into array space.

Lastly, neither of the compilers optimize code in the sense that they scramble instructions to make the best use of CU–PE overlap. Considerable savings are envisioned with this kind of optimization but, once again, this is difficult to do with a simulator which takes no account of this overlay.

All the software described in this paper, and much more that is not, has been tried by users through the agency of simulation. The major task of transferring this software to the real machine still lies ahead, but all of the software, especially the languages, has gone through several stages of appraisal and revision in the light of informed user comment. It is hoped that this relatively conservative software policy will provide a usable Illiac IV very soon after the hardware is completed, and a firm base for a more adventurous second generation of software.

IV. Applications

A. General Considerations

Most of the numerical algorithms available on conventional machines cannot be readily modified for efficient parallel computations. Two difficulties face the users of parallel machines. First, algorithms must be devised that are suited to the array nature of their arithmetic units. Second, data may have to be stored in highly interleaved memories so that proper combinations of elements are available simultaneously at various steps of the algorithm.

Fig. 5 illustrates that Illiac IV is designed to execute the same instruction streams simultaneously on 64 data sets that are stored in separate memories. Thus Illiac IV is well suited to evaluate the same function f on 64 sets of arguments, provided that the data base from which the argument sets are drawn can be structured in a certain fashion. The data base must be distributed among the PEs so that the argument sets required in the ith evaluation of the function f can be stored in the ith memory or in a memory that is "close" to the ith memory.

A primary requirement for many of the time-consuming problems in scientific computation is repeated evaluation of the same functional form on different argument sets. Consider the following examples:

1) Matrix-Multiply: The elements z_i of a column of the matrix $Z=AX$ are produced by forming

$$z_i = \sum_{j=1}^{n} a_{ij}x_j$$

where x_j are the elements of the corresponding column in X. This requires evaluation of the function

$$f(\boldsymbol{y}, \boldsymbol{x}) = \sum_{j=1}^{n} y_jx_j$$

where the argument $(y_1, y_2, \cdots, y_n)$ is repeatedly replaced by $(a_{i1}, a_{i2}, \cdots, a_{in})$, for $i=1, 2, \cdots n$.

2) Solution of Simultaneous Linear Equations: Using the Gauss–Jordan method, the dominant computation is

$$a_{ij}' = a_{ij} - c_ja_{kj}, \qquad j = k, k+1, \cdots, n$$
$$i = k, k+1, \cdots, n$$

or repeated evaluation of $f(x, y, c)=x-cy$, where the three-tuple argument (x, y, c) is repeatedly replaced by (a_{ij}, a_{kj}, c_j).

3) Two-Dimensional Finite Difference Schemes: An explicit scheme for solving the "heat equation" $\partial u/\partial u=\Delta^2 u$ is

$$u_{ij}^{n+1} = u_{ij}^n + C(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n)$$

where n refers to the nth time step.

This requires that

$$f(x, y, z, u, v, c) = x + c(y + z + u + v - 4x)$$

be evaluated with suitable argument sets in which x takes on values of u_{ij}, and y, z, u, and v take on corresponding values of the "neighbors" of u_{ij}.

Note that if one sampled the instruction stream on a conventional machine during execution of these algorithms, the computation of f would dominate the calculation. For these illustrations, the data base can be stored among the distinct PEMs so that there is no difficulty in accessing the appropriate arguments and performing up to 64 simultaneous evaluations of f.

The two central considerations in programming Illiac IV are 1) the exploitation of the simultaneous arithmetic capability, and 2) the distribution of operands in the memories so that the required argument sets can be accessed without time-consuming rearrangement of storage. Exploiting simultaneous arithmetic is generally straightforward. Often cursory inspection of the loops in a conventional program will reveal methods of utilizing the simultaneous arithmetic feature. There are cases where it is not straightforward, i.e., cases where, because of data dependent conditions or other considerations, it may be desirable to evaluate several different functional forms $f_1, f_2, \cdots, f_k$ in different PEs at the same time. This can be accomplished either by turning off groups of PEs and sequentially evaluating $f_1, f_2, \cdots, f_k$ or by imbedding one functional form f_i in the instruction stream of another functional form f_j [18]. On a digital computer all functions, regardless of how dissimilar their forms are, are expressed as sequences of fetches, stores, and arithmetic instructions. It is trivial to imbed the evaluation of $f=xy$ in the instruction stream calculating $g=a+bc$ and it is possible to imbed the evaluation of $f+g$ in the computation $h=\sin(x)$.

Memory allocation is somewhat less straightforward. Generally if the data base for the problem lends itself to representation either in vector form or in matrix form there are schemes for storing the vectors or matrices among the 64 distinct memories which preserve the topology or connectedness of the elements in the data base [19]–[22]. The function evaluation involved in example 3 requires that the PEM which contains the element u_{ij} also have easy access to $u_{i,j+1}$, $u_{i,j-1}$, $u_{i+1,j}$, $u_{i-1,j}$. There are storage schemes which ensure that access to the "neighbors" of an element are always convenient. Furthermore there are schemes which allow simultaneous access of both rows and columns of a matrix.

The applicability of Illiac IV to large problems in scientific computation has been studied for a wide variety of problems [23]. As one might expect, problems have been found which are ideally suited for Illiac IV or which can be made ideally suited with very little modification to the algorithms. Problems have also been

TABLE I
Illiac Applications and Efficiency

Applicable with High PE Utilization
Finite-difference calculations in one, two, and three dimensions.
Matrix arithmetic.
Quadrature (including fast Fourier transform).
Signal processing.
Linear programming (this application could drop to low PE utilization when I/O is taken into consideration).
Applicable with Moderate PE Utilization
Particle-moving problems (nonlinear Monte Carlo transport, etc.).
Nonsymmetric eigenvalue calculation.
Solution of linear equations and matrix inversion (if iterative methods are used this application could move up to high utilization).
Solution of nonlinear equations.
Polynomial root finding (in some instances, this application moves up to high PE utilization).
Applicable with Low PE Utilization
Inversion of tridiagonal matrices.
Table lookup in large unstructured tables.

found which can be adapted for Illiac IV at some sacrifice in utilization of the PEs, and some problems have been found which defy modification to exploit the machine architecture. Table I gives some examples of types of calculations which fall in each of these categories.

It is not uncommon to find that numerical techniques which are generally regarded as superior to others for conventional machines might not be superior for minimizing execution time on Illiac IV. For example, in the solution of linear equations on a conventional computer, Gaussian elimination and back substitution is generally regarded as superior to the Gauss–Jordan algorithm because there are fewer arithmetic operations involved. However, on Illiac IV, the execution time required to solve a set of linear equations employing the Gauss–Jordan algorithm is shorter than that required for Gaussian elimination and back substitution because, among other things, the back substitution portion of the algorithm is essentially a sequential process.

B. Two Example Applications

We shall close the application section of the paper by considering two sample problems in the area of matrix computation: Jacobi's algorithm for finding the eigenvalues and eigenvectors of real symmetric matrices, and reducing a real nonsymmetric matrix to the upper-Hessenberg form using Householder's transformations.

1) Jacobi's Algorithm for Finding the Eigenvalues and Eigenvectors of a Real Symmetric Matrix: In the classical method of Jacobi [24], a real symmetric matrix can be reduced to a diagonal matrix by a sequence of plane rotations,

$$A_{k+1} = R_k A_k R_k^t, \qquad k = 1, 2, \cdots \tag{1}$$

where $A_1 = A$ is the $n \times n$ original matrix, and each rotation R_k differs from the identity matrix only in the elements,

$$\begin{aligned} R_{pp}^{(k)} &= R_{qq}^{(k)} = \cos\theta_{pq}^{(k)} \\ R_{pq}^{(k)} &= -R_{qp}^{(k)} = \sin\theta_{pq}^{(k)}, \qquad p < q. \end{aligned} \tag{2}$$

By properly choosing the angle $\theta_{pq}^{(k)}$, the off-diagonal elements

$$a_{pq}^{(k+1)} = a_{qp}^{(k+1)}$$

of A_{k+1} in (1) are eliminated. It can be easily shown [25] that $\theta_{pq}^{(k)}$ is given by,

$$\begin{aligned} \cos^2\theta_{pq}^{(k)} &= \frac{1}{2}\left(1 + \frac{x_k}{y_k}\right) \\ \sin^2\theta_{pq}^{(k)} &= \frac{1}{2}\left(1 - \frac{x_k}{y_k}\right) \end{aligned} \tag{3}$$

where

$$\begin{aligned} x_k &= \left| a_{pp}^{(k)} - a_{qq}^{(k)} \right| \\ y_k &= (4a_{pq}^2 + x_k^2)^{1/2}. \end{aligned}$$

Restricting $\theta_{pq}^{(k)}$ by $|\theta_{pq}^{(k)}| \le \pi/4$, then $\cos\theta_{pq}^{(k)}$ will always be taken positive and $\sin\theta_{pq}^{(k)}$ will be of the same sign as $[2a_{pq}^{(k)}/(a_{pp}^{(k)} - a_{qq}^{(k)})]$. From the definition (2) of the rotations R_k we can see that only the rows and columns p and q of A_k are affected by the transformation (1). Clearly the implementation of such an algorithm on a parallel machine would be highly inefficient. For, even if updating of the $2(n-1)$ elements (A_{k+1} is symmetric, and $a_{pq}^{(k+1)} \equiv 0$ hence need not be computed) is performed with all the PEs working; it requires only one or two PEs at the most to compute $R_{pp}^{(k)}$ and $R_{pq}^{(k)}$.

It is possible, however, to modify Jacobi's algorithm so as to eliminate more than one off-diagonal element (above the main diagonal) per transformation, and hence make the method more suited for parallel computations (2). For example, for a matrix A of an even order $n=4$, if the orthogonal transformation R_1 is chosen as

$$\begin{bmatrix} R_{11}^{(1)} & R_{12}^{(1)} & 0 & 0 \\ -R_{12}^{(1)} & R_{22}^{(1)} & 0 & 0 \\ 0 & 0 & R_{33}^{(1)} & R_{34}^{(1)} \\ 0 & 0 & -R_{34}^{(1)} & R_{44}^{(1)} \end{bmatrix}$$

where the $R_{pq}^{(1)}$ for all the pairs (p, q) are given by (2), then $\theta_{12}^{(1)}$ and $\theta_{34}^{(1)}$ can be independently chosen such that the elements $a_{12}^{(2)}$ and $a_{34}^{(2)}$ of $A_2 = R_1 A_1 R_1^t$ are annihilated. Similarly, by appropriately choosing the elements of

$$R_2 = \begin{bmatrix} R_{11}^{(2)} & 0 & R_{13}^{(2)} & 0 \\ 0 & R_{22}^{(2)} & 0 & R_{24}^{(2)} \\ -R_{13}^{(2)} & 0 & R_{33}^{(2)} & 0 \\ 0 & -R_{24}^{(2)} & 0 & R_{44}^{(2)} \end{bmatrix}$$

$$R_3 = \begin{bmatrix} R_{11}^{(3)} & 0 & 0 & R_{14}^{(3)} \\ 0 & R_{22}^{(3)} & R_{23}^{(3)} & 0 \\ 0 & -R_{23}^{(3)} & R_{33}^{(3)} & 0 \\ -R_{14}^{(3)} & 0 & 0 & R_{44}^{(3)} \end{bmatrix}$$

the elements $a_{13}^{(3)}$ and $a_{24}^{(3)}$ of A_3, and $a_{14}^{(4)}$ and $a_{23}^{(4)}$ of A_4 are eliminated, respectively. Consequently, after $(n-1)$ transformations each of the $\frac{1}{2}n(n-1)$ off-diagonal elements (above the main diagonal) has been eliminated once, each transformation eliminating $n/2$ such elements. Let each $(n-1)$ orthogonal transformation be denoted by a sweep; then for our specific example $(n=4)$ the second sweep will consist of the orthogonal matrices R_4, R_5, and R_6 which have the same construction as R_1, R_2, and R_3, respectively, and so on. Thus for matrices of even order (see [25] for the general case), the elements of each of the $(n-1)$ orthogonal matrices R_k within each sweep are given by

$$R_{pp}{}^{(k)} = R_{qq}{}^{(k)} = \cos\theta_{pq}{}^{(k)}$$
$$R_{pq}{}^{(k)} = -R_{qp}{}^{(k)} = \begin{cases} \sin\theta_{pq}{}^{(k)}, & p < q \\ -\sin\theta_{pq}{}^{(k)}, & p > q \end{cases} \tag{4}$$

where p and q are sequences defined as follows.

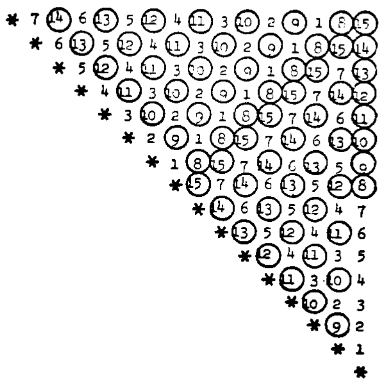
a) For
$$k = 1, 2, 3, \cdots, n/2 - 1$$
$$q = n/2 - k + 1, n/2 - k + 2, \cdots, n - k$$
$$p = \begin{cases} (n-2k+1)-q, & \frac{n}{2}-k+1 \le q \le n-2k \\ 2(n-k)-q, & n-2k < q \le n-k-1 \\ n, & n-k-1 < q. \end{cases} \tag{5}$$

b) For
$$k = n/2, n/2+1, \cdots, n-1$$
$$q = n-k, n-k+1, \cdots, (3/2)n - k - 1$$
$$p = \begin{cases} n, & q < n-k+1 \\ 2(n-k)-q, & n-k+1 \le q \le 2(n-k)-1 \\ (3n-2k-1)-q, & 2(n-k)-1 < q. \end{cases} \tag{6}$$

The remaining elements of R_k are, of course, zero. For each k the angles $\theta_{pq}{}^{(k)}$ are determined such that the elements $a_{pq}{}^{(k)}$ are annihilated for all the pairs (p, q). For example, for $n=8$ and $k=4$ the pairs (p, q) are determined by (6): $\{(8, 4); (3, 5); (2, 6); (1, 7)\}$, and R_4 is of the form

$$\begin{bmatrix} R_{11}{}^{(4)} & & & & & & R_{17}{}^{(4)} & \\ & R_{22}{}^{(4)} & & & & R_{26}{}^{(4)} & & \\ & & R_{33}{}^{(4)} & & R_{35}{}^{(4)} & & & \\ & & & R_{44}{}^{(4)} & & & & R_{48}{}^{(4)} \\ & & -R_{35}{}^{(4)} & & R_{55}{}^{(4)} & & & \\ & -R_{26}{}^{(4)} & & & & R_{66}{}^{(4)} & & \\ -R_{17}{}^{(4)} & & & & & & R_{77}{}^{(4)} & \\ & & & -R_{48}{}^{(4)} & & & & R_{88}{}^{(4)} \end{bmatrix}. \tag{7}$$

Therefore, denoting each element eliminated in a given sweep in the kth transformation by the integer k, the pattern of the annihilated elements for a matrix of order 16 is



Let us consider now the implementation of this modified scheme on a parallel machine. For the sake of simplicity of the illustrations we will assume that the size of the matrix is equal to the number of PEs, and furthermore we will ignore the fact that the matrix is symmetric and hence we need only operate on those elements on and above the main diagonal. Because each row of the matrix $B=R_kA_k$ is obtained by summing multiples of two rows of A_k, and each column of $A_{k+1}=BR_k{}^t$ is obtained by summing multiples of two columns of B, in order to minimize the number of idle PEs in the matrix multiplication $R_kA_kR_k{}^t$, it is essential to store the elements of A_k such that each row or column can be accessed with one memory fetch. This is achieved by storing A_k in the "skewed" form. In such a storage scheme we map the array elements into PEM as follows:

$$a_{ij} \rightarrow \text{row } i, \text{ of PE } (i+j-1) \pmod N \tag{8}$$

where N is the number of PEs being used. For example, for $n=8$ the elements of an 8×8 matrix are stored as follows:

LOC.	PE1	PE2	PE3	PE4	PE5	PE6	PE7	PE8
1	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8
2	2,8	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3	3,7	3,8	3,1	3,2	3,3	3,4	3,5	3,6
4	4,6	4,7	4,8	4,1	4,2	4,3	4,4	4,5
5	5,5	5,6	5,7	5,8	5,1	5,2	5,3	5,4
6	6,4	6,5	6,6	6,7	6,8	6,1	6,2	6,3
7	7,3	7,4	7,5	7,6	7,7	7,8	7,1	7,2
8	8,2	8,3	8,4	8,5	8,6	8,7	8,8	8,1

Let us consider the transformation $A_5=R_4A_4R_4{}^t$ where R_4 is given by (7). By routing, the following configuration can be obtained.

PE1	PE2	PE3	PE4	PE5	PE6	PE7	PE8
$a_{11}^{(4)}$	$a_{17}^{(4)}$	$a_{22}^{(4)}$	$a_{26}^{(4)}$	$a_{33}^{(4)}$	$a_{35}^{(4)}$	$a_{44}^{(4)}$	$a_{48}^{(4)}$
$a_{77}^{(4)}$		$a_{66}^{(4)}$		$a_{55}^{(4)}$		$a_{88}^{(4)}$	

However, some of the PEs have to remain idle since the routing distances for all of the elements are not the same. Now, from (3) $\sin\theta_{pq}{}^{(5)}$ and $\cos\theta_{pq}{}^{(5)}$ for the pairs (p, q): $\{(1, 7); (2, 6); (3, 5); (4, 8)\}$ are computed simultaneously with roughly half the PEs being idle. Once all the elements of R_4 are computed, the matrix multiplication $R_4A_4R_4{}^t$ is performed with all the PEs working.

LOC. \ PE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1, 1	1, 2	1, 3	1, 4	2, 1	2, 2	2, 3	2, 4	3, 1	3, 2	3, 3	3, 4	4, 1	4, 2	4, 3	4, 4
2	4, 5	4, 6	4, 7	4, 8	1, 5	1, 6	1, 7	1, 8	2, 5	2, 6	2, 7	2, 8	3, 5	3, 6	3, 7	3, 8
3	3, 9	3,10	3,11	3,12	4, 9	4,10	4,11	4,12	1, 9	1,10	1,11	1,12	2, 9	2,10	2,11	2,12
4	2,13	2,14	2,15	2,16	3,13	3,14	3,15	3,16	4,13	4,14	4,15	4,16	1,13	1,14	1,15	1,16
5	8,14	5, 1	5, 2	5, 3	5, 4	6, 1	6, 2	6, 3	6, 4	7, 1	7, 2	7, 3	7, 4	8, 1	8, 2	8, 3
6	7, 8	8, 5	8, 6	8, 7	8, 8	5, 5	5, 6	5, 7	5, 8	6, 5	6, 6	6, 7	6, 8	7, 5	7, 6	7, 7
7	6,12	7, 9	7,10	7,11	7,12	8, 9	8,10	8,11	8,12	5, 9	5,10	5,11	5,12	6, 9	6,10	6,11
8	5,16	6,13	6,14	6,15	6,16	7,13	7,14	7,15	7,16	8,13	8,14	8,15	8,16	5,13	5,14	5,15
9	12, 3	12, 4	9, 1	9, 2	9, 3	9, 4	10, 1	10, 2	10, 3	10, 4	11, 1	11, 2	11, 3	11, 4	12, 1	12, 2
10	11, 7	11, 8	12, 5	12, 6	12, 7	12, 8	9, 5	9, 6	9, 7	9, 8	10, 5	10, 6	10, 7	10, 8	11, 5	11, 6
11	10,11	10,12	11, 9	11,10	11,11	11,12	12, 9	12,10	12,11	12,12	9, 9	9,10	9,11	9,12	10, 9	10,10
12	9,15	9,16	10,13	10,14	10,15	10,16	11,13	11,14	11,15	11,16	12,13	12,14	12,15	12,16	9,13	9,14
13	16, 2	16, 3	16, 4	13, 1	13, 2	13, 3	13, 4	14, 1	14, 2	14, 3	14, 4	15, 1	15, 2	15, 3	15, 4	16, 1
14	15, 6	15, 7	15, 8	16, 5	16, 6	16, 7	16, 8	13, 5	13, 6	13, 7	13, 8	14, 5	14, 6	14, 7	14 8	15, 5
15	14,10	14,11	14,12	15, 9	15,10	15,11	15,12	16, 9	16,10	16,11	16,12	13, 9	13,10	13,11	13,12	14, 9
16	13,14	13,15	13,16	14,13	14,14	14,15	14,16	15,13	15,14	15,15	15,16	16,13	16,14	16,15	16,16	13,13

Fig. 12. Storage scheme.

For example, to compute the first row of $B=R_4A_4$: 1) broadcast $R_{11}^{(4)}$ and $R_{17}^{(4)}$ to all PEs, 2) fetch the first row of A_4 and multiply all its elements by $R_{11}^{(4)}$; 3) fetch the seventh row of A_4, multiply all its elements by $R_{17}^{(4)}$, route all the elements of the resultant vector a distance of 2 to the right, and add to 2).

This process is repeated until all the rows are computed. Storing the matrix B in the skewed form, each column of A_5 is obtained by a similar process; for example the eighth column of A_5 is obtained as follows: 1) broadcast $-R_{48}^{(4)}$ and $R_{88}^{(4)}$ to all PEs; 2) fetch the fourth column of B, multiply all its elements by $-R_{48}^{(4)}$, and then route them a distance of 4 to the right; 3) fetch the eighth column of B, multiply all its elements by $R_{88}^{(4)}$, add to 2), and store in the same locations of the eighth column of A_4.

Assuming that the matrix A has converged (using some criterion [24]), to the diagonal form after u sweeps or after r orthogonal transformations, where $r=u(n-1)$, then the diagonal elements of $A_{r+1}=WAW^t$ are taken to be the eigenvalues. The columns of $W^t=V_1{}^tV_2{}^t\cdots V_u{}^t$ are the corresponding eigenvectors, where for the jth sweep

$$V_j{}^t=(R_1{}^t)_j,(R_2{}^t)_j\cdots(R_{n-1}{}^t)_j.$$

2) *Reducing a Real Nonsymmetric Matrix to the Upper-Hessenberg Form:* The second example is that of reducing a real nonsymmetric matrix to the upper-Hessenberg form using Householder's transformations [24]. Such a reduction always precedes the application of Francis' QR transformations [26], [27] for finding the eigenvalues of a nonsymmetric matrix. For an $n\times n$ matrix the reduction is completed in $(n-2)$ steps, the kth of which is given by

$$A_k=A_{k-1}-v_kp_k{}^t-(q_k-\alpha_ku_k)v_k{}^t \tag{9}$$

where $k=1,2,\cdots,n-2$ and $A_0=A$ is the original matrix,

$$u_k{}^t=(0,\cdots,0,a_{k+1,k}\pm S_k,a_{k+2,k},\cdots,a_{n,k})$$

$$S_k=\left[\sum_{i=k+1}^{n}a_{i,k}{}^2\right]^{1/2}$$

$$p_k{}^t=u_k{}^tA_{k-1}$$

$$q_k=A_{k-1}u_k$$

$$v_k=u_k/S_k(S_k\pm a_{k+1,k}) \tag{10}$$

and

$$\alpha_k=p_k{}^tv_k.$$

This reduction requires approximately (5/3) n^3 operations. From (9) and (10) it is clear that the major computations involved in each step are those of forming the vectors p_k and q_k. For computing p_k, we would like to store the matrix A_{k-1} such that we can access any row by one memory fetch, and similarly for the columns when computing q_k. One solution would be to store the matrix A in the skewed form described in the previous example. However, since the order of the remaining matrix under operation is reduced by one after each transformation, then it is highly desirable to store A in such a form that we can have simultaneous access to each of the N elements of any $\sqrt{N}\times\sqrt{N}$ submatrix of A. Here, N is the number of PEs of the parallel machine. Thus during each step we can operate on any $\sqrt{N}\times\sqrt{N}$ submatrix that does not contain eliminated elements, with none of the PEs idle. For $n=N=16$, the storage scheme [28] is shown in Fig. 12. This storage scheme maps an element a_{ij} into the memory as follows:

$$a_{ij}\rightarrow\text{row }[\sqrt{N}\lfloor(i-1)/\sqrt{N}\rfloor+\lfloor(j-1)/\sqrt{N}\rfloor+1]$$

$$\text{PE}[\{((i-1)\bmod\sqrt{N})\sqrt{N}+\lfloor(i-1)/\sqrt{N}\rfloor+j-1\}\,(\bmod\sqrt{N})+1] \tag{11}$$

where $\lfloor x\rfloor$ is the greatest integer less than or equal to x.

Let us demonstrate the computations involved in the first transformation ($k=1$). The elements $a_{i,1}{}^2$ ($i=2, 3, \cdots, 16$) are obtained simultaneously by one multiplication, while $S_1{}^2$ in (10) is obtained in $\log_2 16=4$ additions. Evaluating the square root S_1 is, of course, an inefficient operation since only few PEs can be used. Obtaining the vectors u_1 and v_1 is rather trivial. Now, if we observe that the elements of each of the rows {16, 12, 8, 4; 15, 11, 7, 3; 14, 10, 6, 2; 13, 9, 5, 1} are separated by a routing distance of 1 from those of the following row, then $p_1{}^t=u_1{}^tA_0$ is obtained as follows:

1) broadcast the element $u_{1,16}=a_{1,16}$ to all PEs, fetch the 16th row of A_0, multiply all its elements by $a_{1,16}$, and route the elements of the resulting vector a distance of 1 to the left;

2) broadcast $u_{1,12}=a_{1,12}$ to all PEs, fetch the 12th row, multiply all its elements by $a_{1,12}$, add to 1), and route all the elements of the resulting vector a distance of 1 to the left, and so forth for the rest of the rows 8, 4, 15, 11, 7, $\cdots$, 1 in order until we obtain the row vector $p_1{}^t$. Similarly, we can see that the column vector $q_1=A_0\ u_1$ may be obtained as follows: a) broadcast $u_{1,1}$ to all PEs, fetch the first column of A_0, multiply all its elements by $u_{1,1}$, and route all the elements of the resulting vector a distance 1 to the right; b) broadcast $u_{1,2}$ to all PEs, fetch the second column of A_0, and multiply all its elements by $u_{1,2}$. Add the elements of the resultant vector to those of a), and route them a distance of 1 to the right.

Repeating the process for the rest of the columns 3, 4, $\cdots$, 16, we obtain the vector q_1. Thus each element of p_1 and q_1 has been obtained by 16 multiplications and 16 additions. The scalar α_1 is computed by one multiply and $\log_2 16=4$ adds, and the vector $w_1=q_1-\alpha_1u_1$ is computed by one multiply and one add. Partitioning the matrix A_0 into 4×4 submatrices $[A_{RC}]_0$, ($R, C=1, 2, 3, 4$), and similarly partitioning the vectors v_1, p_1, and w_1 into subvectors each of $\sqrt{N}=4$ components, then (9) can be written as follows:

$$[A_{RC}]_k = [A_{RC}]_{k-1} - v_k{}^{(R)}[p_k{}^{(c)}]^t - w_k{}^{(R)}[v_k{}^{(c)}]^t \quad (12)$$

for $k=1, 2, \cdots, n-2$. Thus by properly storing the vectors v_k, p_k, and w_k, all the elements of each submatrix $[A_{RC}]_k$ can be computed simultaneously.

The same storage scheme can be used for the reduction of a nonsymmetric matrix to an upper triangular one using Householder's transformations. Such a reduction is used in solving systems of linear equations.

V. The ARPA Network

A. Introduction

One of the unusual things about Illiac IV as a new computer architecture is the fact that there already exists a large body of problems to be solved for which Illiac IV is particularly well suited in terms of size, speed, and sophistication. Additionally, there is a growing community of scientists and researchers located all about the country who are eagerly awaiting the ability to use Illiac IV in solving these problems.

It became apparent in the mid 1960's with the explosive growth and variety of computer systems and services being developed in the country that some means were going to have to be found in order to couple computer systems and computer users together so that unique features at different sites could be utilized by people other than at that site, and in this way sharing of mutually beneficial resources could be undertaken to provide more economical and convenient ability to solve problems [29]. A prodigious experiment to this end was initiated by the Advanced Research Projects Agency of the Department of Defense and is called the ARPAnet. The ARPAnet network proposes to connect together, via high-speed data transmission lines, a number of ARPA research centers and projects located all about the United States [9], [30], [31].

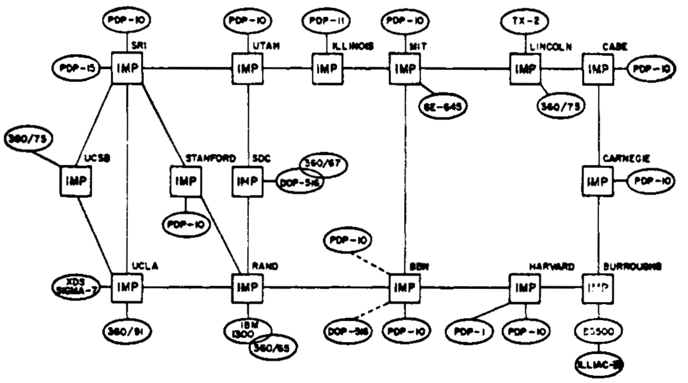


Fig. 13. ARPAnet. April 1971.

B. The Design

Fig. 13 shows the current status of the ARPAnet as of April 1971. Indicated are the universities and research centers who are members of the network as well as various computer systems that exist at each of those centers. Illiac IV is one of those computer systems and will be located at NASA's Ames Research Center, Moffett Field, Calif.

The network itself is a full duplex high-speed (50 000 bits/s) data-transmission network developed by Bolt Beranek and Newman, at Cambridge, Mass.[2] The network itself is a store-and-forward message-transmission network with the nodes of the network occupied by a sophisticated terminal called the IMP [32]. The heart of the IMP is a Honeywell DDP-516 computer which takes care of such tasks as error control, message routing, network "tuning" (adjusting its operation to maximize its efficiency), and statistics gathering. Much care has been devoted in the design and implementation of the network to insure an ultra-high level of reliability (currently no more than one single bit error per year should go undetected). By the store-and-forward mechanism, error control is implemented by retransmission of a message or section of message on which an error was detected. Error correcting codes are appended to each message to allow the detection of a wide class of errors and contribute to the reliability of transmission. All of the IMPs are passive devices in that they perform no other service besides message routing and control.

At any given node in the network, one or more HOST computers may be attached providing a service center or research project with access to the network.[3] The connection between a HOST and the IMP is made over a high-speed interface at 100 000 bits/s, full duplex. Typical HOST computers in use around the network are the DEC PDP-10, IBM 360/75, IBM 360/91, IBM 360/67, and Burroughs B6500. While most of the HOST computers are associated with specific projects sponsored by ARPA, several locations are designated as service HOST sites. For instance, the 360/91 at the Campus Computing Network at the University of California at Los Angeles (UCLA) is available on the network most of the time for use as a general computing ser-

[2] Full duplex means that the user has the ability to send and receive simultaneously—as opposed to half duplex, where the user can only be sending or receiving at any instant in time.

[3] HOST refers to a computer system capable of supplying a full range of computer services to users on the network or at that site.

vice. Several other places, the University of California at Santa Barbara (UCSB), Bolt Beranek and Newman in Cambridge, and Massachusetts Institute of Technology's Multics system also serve as HOST sites providing capabilities and services praticular to their installations and computer systems.

C. The Tower of Babel

The current conceptionalization of the ARPAnet presents, what some people call, "the Tower of Babel" problem in computer resource sharing. One can easily see that there are a wide variety of computer systems in geographically distant locations involved in many dissimilar projects, all speaking a variety of computer languages. While it was easy, relatively speaking, to bring up the hardware portion of the network, i.e., design and build IMPs, establish phone line connections for the 50 000-bits/s transmission lines, etc., much effort has been expended in trying to solve the "software" problem of the network; in other words, how does a human user talk to a remote HOST computer system, and also how does each HOST system talk to the other remote HOST systems?

For several years now, a collection of representatives from each site has been periodically meeting and corresponding with each other to design and implement a series of protocols (procedures) which lay down the ground rules of access and conversation between people and computers on the network. The most basic of these protocols is the IMP to HOST protocol. This protocol contains the procedures by which each host talks to its IMP and data and routing information are exchanged.

The next level of protocol is that of HOST-to-HOST communication [33]. This protocol contains the procedures and ground rules for information transfer between HOSTs and incorporates such features as connection procedures, space allocation, flow control, error control, etc. The HOST-to-HOST protocol was, in fact, the most difficult protocol to develop since it deals most closely with the specifics and idiosyncracies of the various computer hardware configurations and their respective operating systems.

Once the HOST-to-HOST protocol was formulated, a number of applications protocols have followed. These protocols in general allow the proper use of the HOST-to-HOST protocol by systems and applications programs desiring network access. The most important of these is the initial connection protocol which allows the establishment of a full-duplex path of transmission between two systems or programs on the network. This protocol in general is masked from the user and is handled automatically by the network control programs in the HOST sites.

For transmission of data on the network, there is the data-transfer protocol which is concerned with the transmission of large blocks of data from point to point. Since most computer systems treat these large blocks of data as files and have various and elaborate file naming and access mechanisms implemented for the handling of these files, there is also a file-transfer protocol which facilitates the handling of files between arbitrarily different systems on the network and allowing the mapping of their naming and access conventions. In addition to the variances in naming and access, there is the data formatting problem which exists in data transfers. The approach taken to solve this problem was to develop a data reconfiguration protocol and supporting systems whereby the user may specify a transformation function on his data string. He passes it through a reconfiguration service moving it from the sending site to the receiving site satisfying the format constraints at both sites. An example of this might be the conversion of a 36-bit floating-point format for the PDP-10 into a 48-bit floating-point format for the Burroughs B6500.

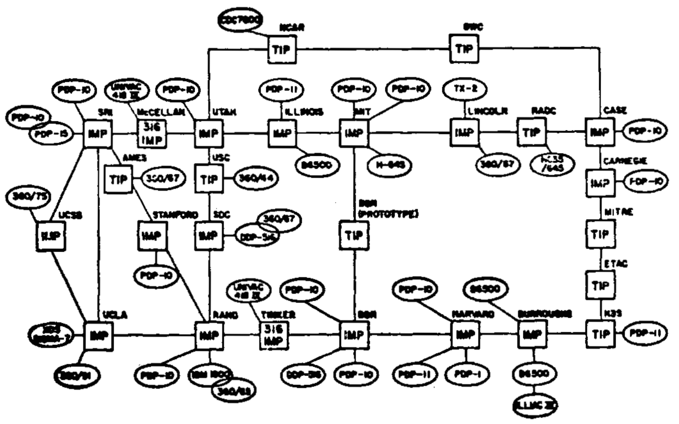


Fig. 14. Projected ARPAnet, April 1972.

Most activities on the network for the individual user will be accomplished in a remote-access manner to the various time-sharing systems on the network. Since there is a wide variance in the types of terminals that exist on the network, a terminal network protocol (TELNET) was developed which attempts to mask all terminals into a network virtual terminal, pass the streams of data and control between that terminal and remote sites in both directions, then unmask the network virtual terminal at the receiving site into the form best desired by that site. For example, both Teletype terminals (33, 35, 37) and IBM 2741 terminals may have access to programs which expect one or the other of those terminals, both with equal facility.

The TELNET protocol handles the usual alphanumeric and hard-copy terminals but does not handle the graphics terminals such as storage scopes or refresh graphics display terminals. To that end, a graphics network protocol is under development at this time. Results so far show, however, that a unified approach to the entire problem of graphics display and interaction is probably not feasible. Indeed, there may eventually be several protocols at various levels dealing with various classes of graphics devices and various modes of graphics display.

D. Expansion Plans

With the initial success of the protocol efforts, the second phase of network development has been entered. During this phase, additional nodes will be added to the network. These nodes, however, differ widely from the IMP of the initial configuration. These new nodes provide for connection of terminal hardware directly to the network and, therefore, are called terminal IMPs or TIPs. A TIP is a parasite node and provides no service capability to the network on its own. Users attached to the network via a TIP must derive all of their computational power and service from remote HOSTs on the network. Thus the second phase of network development sees the introduction of user oriented groups to complement the present research and service organizations. Fig. 14 depicts a projected arrangement of the network and its members as of April 1972.

In addition to the TIP, an effort has been underway for some time at the University of Illinois to develop a "mini HOST" computer system based on the configuration of a small minicomputer (DEC PDP-11) acting as a full cpacity HOST (from the protocol standpoint) and attached to a standard IMP or a TIP. The PDP-11 based system (called the ARPA network terminal system or

ANTS) serves essentially the same functions as a TIP. However, with the splitting of the network node and terminal functions into two separate computers, and providing additional capability in the PDP-11 in the form of a disk, a wider variety of peripherals, more memory, etc., a limited amount of on-site processing power is provided for the user to do housekeeping functions such as network accounting, on-site card deck-to-printer listings, data storage on magnetic tape and disk, etc., and to provide him with a higher level interface into the various protocol streams which will be directed to and from his site.

E. Promising Results

A prominent example of network usage during its earliest implementation phase, when a subnet comprised of University of Utah, Stanford, UCLA, and UCSB existed for a short time, occurred during the development of Stanford Research Institute's SRI's Intellect Augmentation research project. SRI had been using an XDS 940 computer to service their documentation aids and support system. During 1969–1970, they were in the process of selecting and moving to a new central processing system to support that effort, a Digital Equipment Corporation PDP-10.

Prior to receiving their system, SRI had good contact with the University of Utah's graphics research project who already had a PDP-10. After the initial subnet implementation was completed, the SRI group had remote access to Utah's PDP-10 on which they began the conversion of their XDS 940 systems programs. The use of this subnet to connect their terminals and hard-copy devices (printers, etc.) to the Utah PDP-10 system, as if they were at the Utah site, enabled the systems programmers at SRI to complete their initial conversion project in less than half a year. Activities during this time included remote text editing, compilations of systems programs, debug executions, and subsequent shipment of printer listings, debug results, etc., back to SRI for listing on their printer.

As an example of the more current type of operation on the network, the UCLA IBM 360/91 computer system at the Campus Computing Network division has been selected for inclusion in the network as a major service site. Software development on the 360/91 has been directed mainly at providing capability for any remote user to access UCLA and obtain the standard services provided by the OS 360 system. The most prominent feature of this implementation is the remote job entry (RJE) capability where a given site can supply an input device, a listing output device and a punch output device plus a full-duplex operator's console, and bind these all together via the network to the 360/91, allowing submission and control of batch-type jobs for the OS 360 system. A number of people at Rand Corporation have been operating from Rand to UCLA in this manner for some months now and their success is highly promising.

Since any one of the TIPs or the "mini-host" PDP-11 terminal systems can provide RJE capability, the group of users who need and can gain access to large computer facilities such as the model 91 is greatly expanded. At costs in line with many RJE terminal systems on the market today, these users gain high-speed access and wide ranging services.

Finally, an example of the use of the network in the future when a large number of TIPs and PDP-11 "mini-host" systems will be attached is a research effort to be undertaken at the National Bureau of Standards. Using the ANTS "mini-host" system, researchers at the National Bureau of Standards site intend to do performance-measuring research utilizing the network as a test bed. Such areas as the performance of remote access to and from various sites on the network, performance measurement of activity between sites on the network both at the user level and at the system level, and simulation of activity sequences to obtain normalized measurements of the response and processing activities of the various sites will be investigated.

F. ILLIAC IV Approaches

In 1972, Illiac IV will join the ARPAnet to provide it with one of its most powerful service sites. Users of Illiac IV will also heavily access the mass data storage system (laser-memory data computer) to aid them in preparing and processing the gigantic amounts of data and enormous problems which they will attempt to solve utilizing Illiac IV. Conversely, the future development of Illiac IV appears to benefit from the network in two ways. First, there are quite a number of people located all over the country who need direct access to Illiac IV for their particular applications and the network provides a very economical framework in which to dispense this service. Secondly, network users represent a large body of technical knowledge and experience in the craft of computer systems building, and the development of Illiac IV should be very much enriched by being placed within this flow of knowledge throughout the ARPAnet community.

G. The Future

Up to now, concern has been with the network itself, its concept, and physical manifestation. Now that the first edition is out, efforts will be directed at developing applications utilizing the diverse capabilities of the member HOST systems and the intellectual community surrounding them. The keyword of the future will be sharing.

Hardware, software, and expertise will be the first sharable quantities. Initially their forms will be traditional, the large 360/91 at UCLA as a service facility, for instance. Specialization will then set in yielding network usage of particular specialized functions at several sites in the solution of individual problems. Systems will subdivide to subsystems and programs, evolving to a veritable smorgasbord of hardware and software problem-solving tools.

Data sharing will become prominent. Specialized data bases will appear as a function of the growing needs of users of the network and their freedom of access to remote network resources. Library and documentation production retrieval systems will evolve to enhance and extend the intellectual capabilities of network researchers.

In the end, the network as a concept may foster a new intellectual revolution as it multiplies the number of minds and tools which can be applied to specific research projects. As one of the first, the ARPAnet should occupy a classical position in the evolution of computer systems and applications technology.

References

[1] "Electronic computers: A historical survey," *Computing Surveys*, vol. 1, no. 1, pp. 7–36, Mar. 1969.

[2] D. L. Slotnick, W. C. Borck, and R. C. McReynolds, "The SOLOMON computer," in *1962 Fall Joint Computer Conf., AFIPS Conf. Proc.* Washington, D. C.: Spartan, 1962, pp. 97–107.

[3] D. L. Slotnick, "Unconventional systems," in *1967 Spring Joint Computer Conf., AFIPS Conf. Proc.* Washington, D. C.: Spartan, 1967, pp. 477–481.

[4] ——, "The fastest computer," *Sci. Amer.*, vol. 224, no. 2, pp. 76–87, Feb. 1971.

[5] S. A. Denenberg, "An introductory description of the ILLIAC IV system," Center for Advanced Computation, University of Illinois

at Urbana-Champaign, Urbana, Ill., ILLIAC IV Doc. 225, Dep. Computer Sci. File 850, July 15, 1971.

[6] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746–757, Aug. 1968.

[7] D. E. McIntyre, "An introduction to the ILLIAC IV computer," *Datamation*, Apr. 1970.

[8] D. J. Kuck, "ILLIAC IV software and applications programming," *IEEE Trans. Comput.*, vol. C-17, no. 8, pp. 758–770, Aug. 1968.

[9] L. G. Roberts and B. D. Wessler, "Computer network development to achieve resource sharing," in *1970 Spring Joint Computer Conf., AFIPS Conf. Proc.* Washington, D. C.: Spartan, 1970, pp. 543–549.

[10] H. R. G. Trout, "A BNF-like language for the description of syntax directed compilers," M.S. thesis, Dep. Computer Sci. Also ILLIAC IV Project, University of Illinois at Urbana-Champaign, ILLIAC IV Doc. 161, Jan. 13, 1969.

[11] *ILLIAC IV Software Reference Manual*, ILLIAC IV Project, University of Illinois at Urbana-Champaign, Urbana, Ill., 1971, vol. 1, ch. 2.

[12] J. Gary, "PDELAN—A programming language for the solution of partial diffential equations," ILLIAC IV Project, University of Illinois at Urbana-Champaign, Urbana, Ill., ILLIAC IV Doc. 229, July 16, 1970.

[13] *ILLIAC IV Software Reference Manual*, ILLIAC IV Project, University of Illinois at Urbana-Champaign, Urbana, Ill., vol. 2, ch. 1.

[14] N. E. Abel *et al.*, "TRANQUIL, a language for an array processing computer," in *1969 Spring Joint Computer Conf., AFIPS Conf. Proc.* Washington, D. C.: Spartan, 1969.

[15] *ILLIAC IV Software Reference Manual*, ILLIAC IV Project, University of Illinois at Urbana-Champaign, Urbana, Ill., 1971, vol. 2, ch. 2.

[16] Duncan H. Lawrie, *GLYPNIR Programming Manual*, ILLIAC IV Project, University of Illinois at Urbana–Champaign, Urbana, Ill., ILLIAC IV Doc. 232, Aug. 27, 1970.

[17] *ILLIAC IV Software Reference Manual*, ILLIAC IV Project, University of Illinois at Urbana-Champaign, Urbana, Ill., 1971, vol. 2, ch. 3.

[18] Toshio Yasui, "Programming the Tillotson equation of state for ILLIAC IV Project," ILLIAC IV Project, University of Illinois at Urbana-Champaign, Urbana, Ill., ILLIAC IV Doc. 189, Aug. 20, 1969.

[19] M. Knowles, B. Okawa, and Y. Muraoka, "Matrix operations on ILLIAC IV." ILLIAC IV Project, University of Illinois at Urbana–Champaign, Urbana, Ill., ILLIAC IV Doc. 52, Mar. 1, 1967.

[20] James E. Stevens, Jr., "Matrix multiplication algorithms for ILLIAC IV," Center for Advanced Computation, University of Illinois at Urbana-Champaign, Urbana, Ill., ILLIAC IV Doc. 231, Aug. 26, 1970.

[21] V. Benokraitis, "Alternate storage methods for two-dimensional hydrodynamics calculations," ILLIAC IV Project, University of Illinois at Urbana-Champaign, Urbana, Ill., ILLIAC IV Doc. 136, May 27, 1968.

[22] Paul W. Kraska, "Array storage allocation," ILLIAC IV Project, University of Illinois at Urbana–Champaign, Urbana, Ill., ILLIAC Doc. 186, Aug. 1, 1969.

[23] *ILLIAC IV Research Documents Abstracts*, ILLIAC IV Project, University of Illinois at Urbana-Champaign, Urbana, Ill., July 1, 1971.

[24] J. H. Wilkinson, *The Algebraic Eigenvalue Problem.* Oxford: Clarendon, 1965.

[25] A. H. Sameh, "On Jacobi and Jacobi-like algorithms for a parallel computer," in *Mathematics of Computation*, vol. 25, pp. 579–590, 1971.

[26] J. G. F. Francis, "The QR-transformations," pts. I and II, *Comput. J.*, vol. 4, pp. 265–271 and 332–345, 1961–1962.

[27] R. S. Martin *et al.*, "The QR-algorithm for real Hessenberg matrices," *Numer. Math.*, vol. 14, pp. 219–231, 1970.

[28] D. Kuck and A. Sameh, "Parallel computation of eigenvalues of real matrices," in *Proc. IFIP Cong.*, Yugoslavia, 1971.

[29] T. Marill and L. G. Roberts, "Toward a cooperative network of time-shared computers," in *1966 Fall Joint Computer Conf., AFIPS Conf. Proc.* Washington, D. C.: Spartan, 1966.

[30] L. G. Roberts, "The ARPA network," in *Invitational Workshop on Networks of Computer Proc.*, Nat. Security Admin., 1968.

[31] ——, "Resource sharing computer networks," presented at IEEE Int. Conf., Mar. 1969.

[32] F. E. Heart, R. E. Kahn *et al.*, "The interface message processor for the ARPA network," in *1970 Spring Joint Computer Conf., AFIPS Conf. Proc.* Washington, D. C.: Spartan, 1970.

[33] S. Carr, S. Crocker, and V. Cerf, "HOST–HOST communication protocol in the ARPA network," in *1970 Spring Joint Computer Conf., AFIPS Conf. Proc.* Washington, D. C.: Spartan, 1970.