Chapter 27

The ILLIAC IV computer[1]

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

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

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

Introduction

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

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

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

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

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

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

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

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

Summary of the ILLIAC IV

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

Instruction and addressing control

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

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

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

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

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

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

Mode control and data conditional operations

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

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

Routing

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

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

Common operand broadcasting

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

Processor partitioning

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

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

Array partitioning

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

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

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

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

ILLIAC IV structure

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

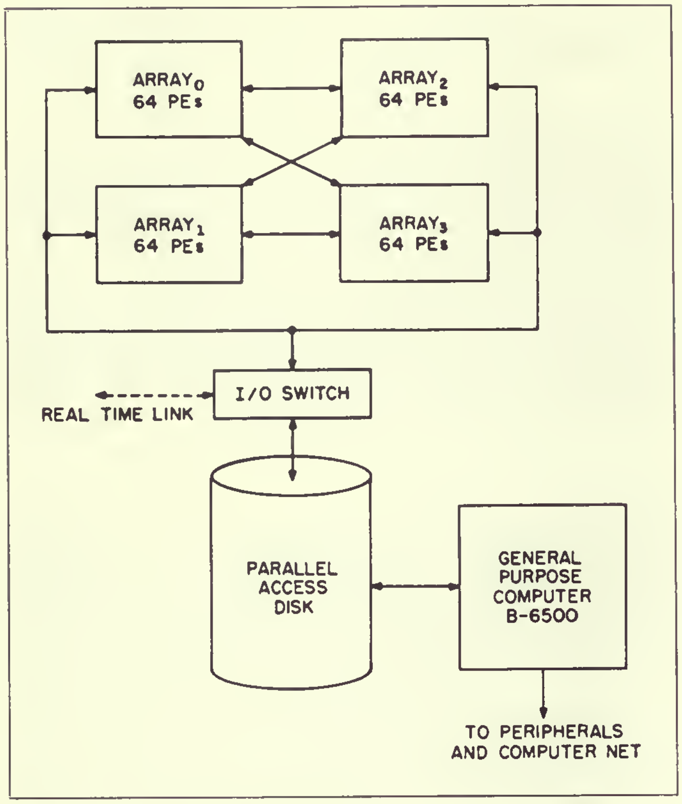

Fig. 1. ILLIAC IV system organization.

Array organization

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

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

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

Multiarray configurations

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

Control unit

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

1 To control and decode the instruction streams

2 To generate the control pulses transmitted to the processing elements for instruction execution

3 To generate and broadcast those components of memory addresses which are common to all processors

4 To manipulate and broadcast data words common to the calculations of all the processors

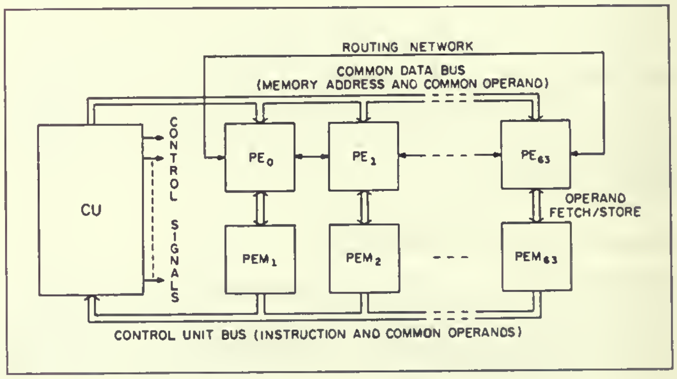
ROUTING NETWORK
COMMON DATA BUS (MEMORY ADDRESS AND COMMON OPERAND)
CU
CONTROL SIGNALS
PE$_0$
PE$_1$
PE$_{63}$
PEM$_1$
PEM$_2$
PEM$_{63}$
OPERAND FETCH/STORE
CONTROL UNIT BUS (INSTRUCTION AND COMMON OPERANDS)

Fig. 2. Array structure.

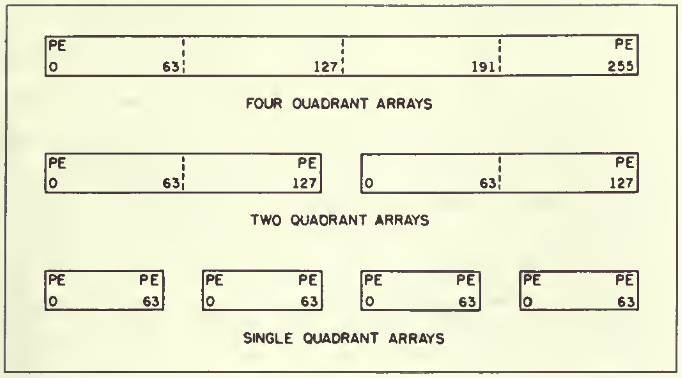

Fig. 3. Multiarray configurations.

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

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

Instruction processing

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

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

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

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

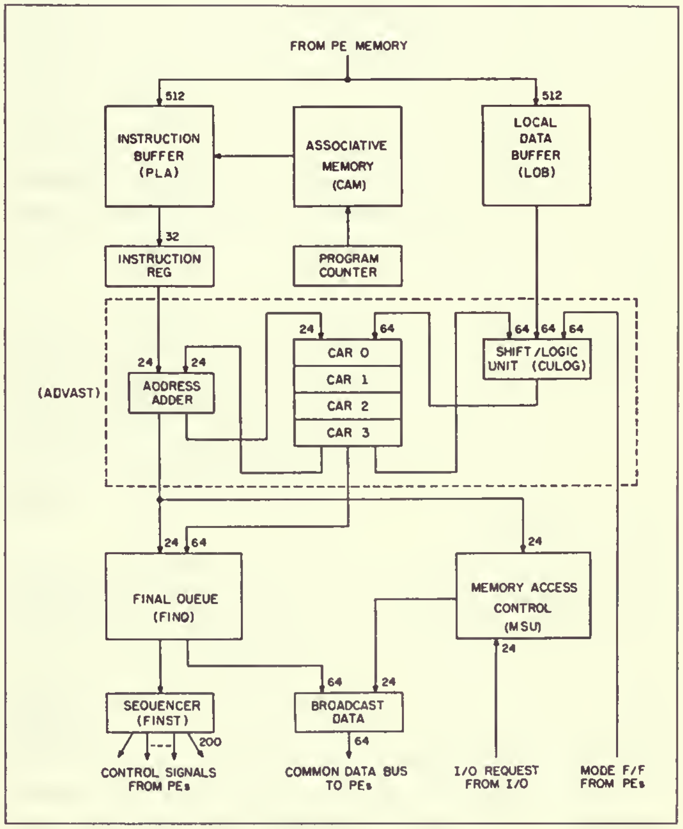

Fig. 4. Control-unit block diagram.

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

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

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

Memory addressing

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

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

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

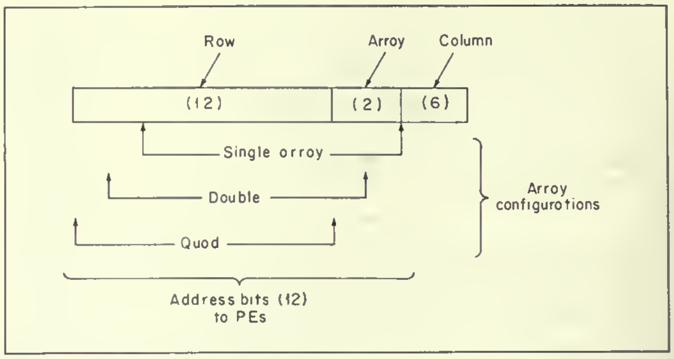

Fig. 5. Memory address structure.

CU data operations

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

Configuration control

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

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

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

Trap processing

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

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

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

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

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

Processing element (PE)

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

1 Four 64-bit registers (A, B, R, S) to hold operands and results. A serves as the accumulator, B as the operand register, R as the multiplicand and data routing register, and S as a general storage register.

2 An adder/multiplier (MSG, PAT, CPA), a logic unit (LOG), and a barrel switch (BSW) for arithmetic, Boolean, and shifting functions, respectively.

3 A 16-bit index register (RGX) and adder (ADA) for memory address modification and control.

4 An 8-bit mode register (RGM) to hold the results of tests and the PE ENABLE/DISABLE state information.

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

PE mode control

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

Data paths

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

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

Table 1 PE data operations

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

† (Single length)/(double length)

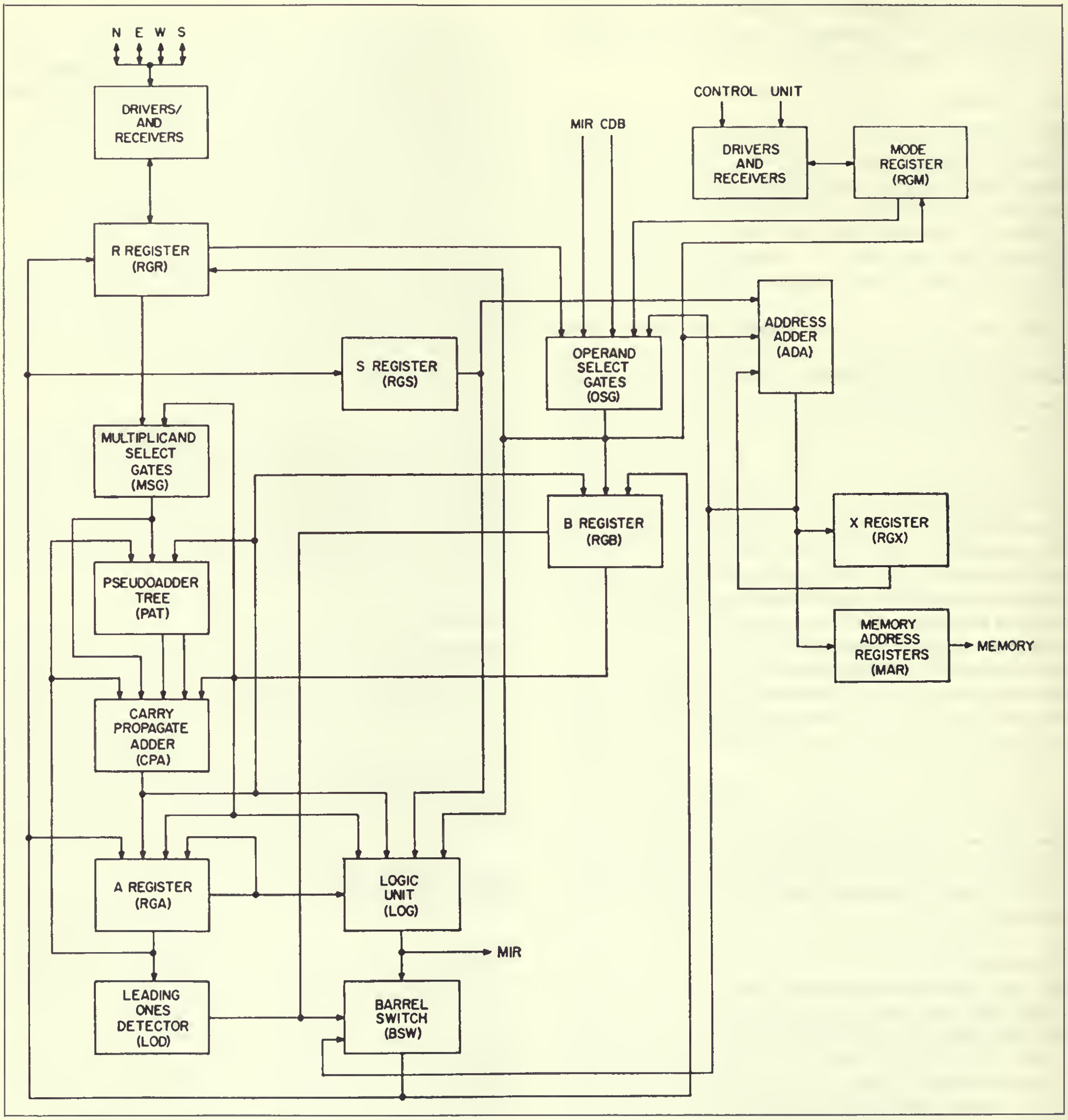

Fig. 6. Processing-element block diagram.

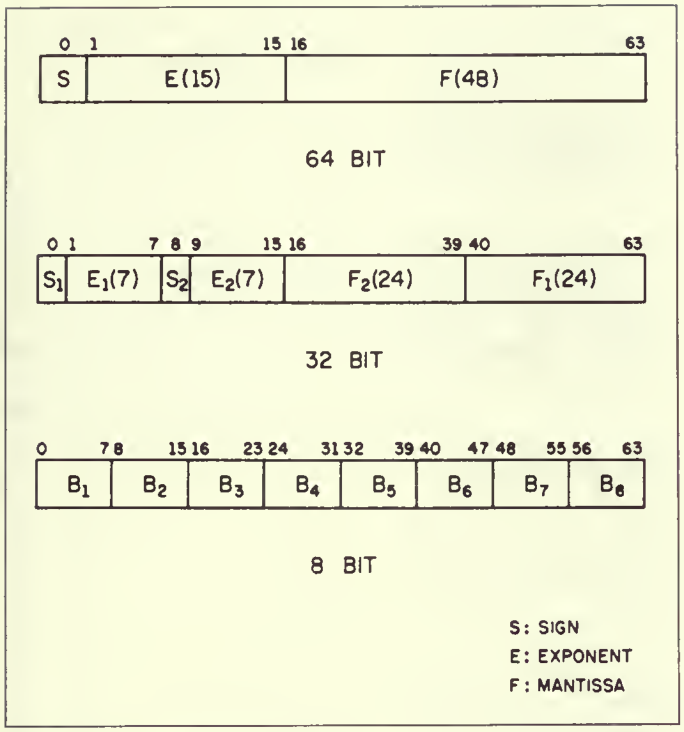

Fig. 7. ILLIAC IV data representation.

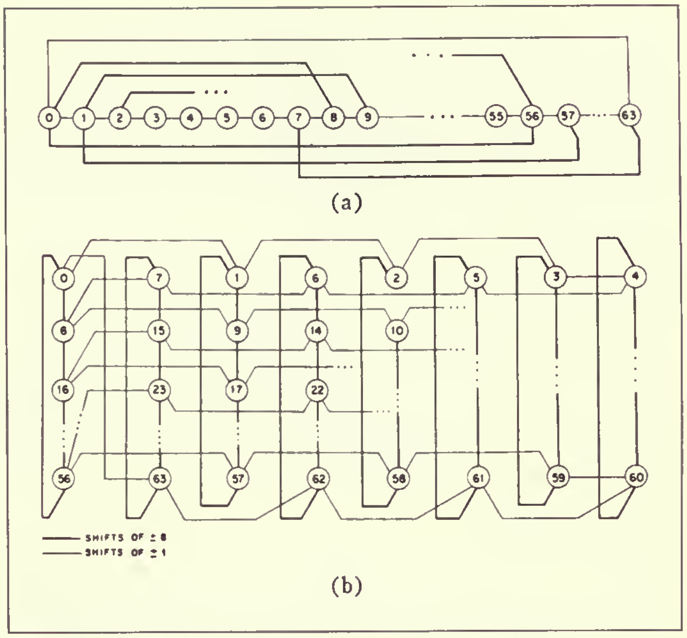

Fig. 8. (a) Electrical connectivity for routing. (b) Physical layout.

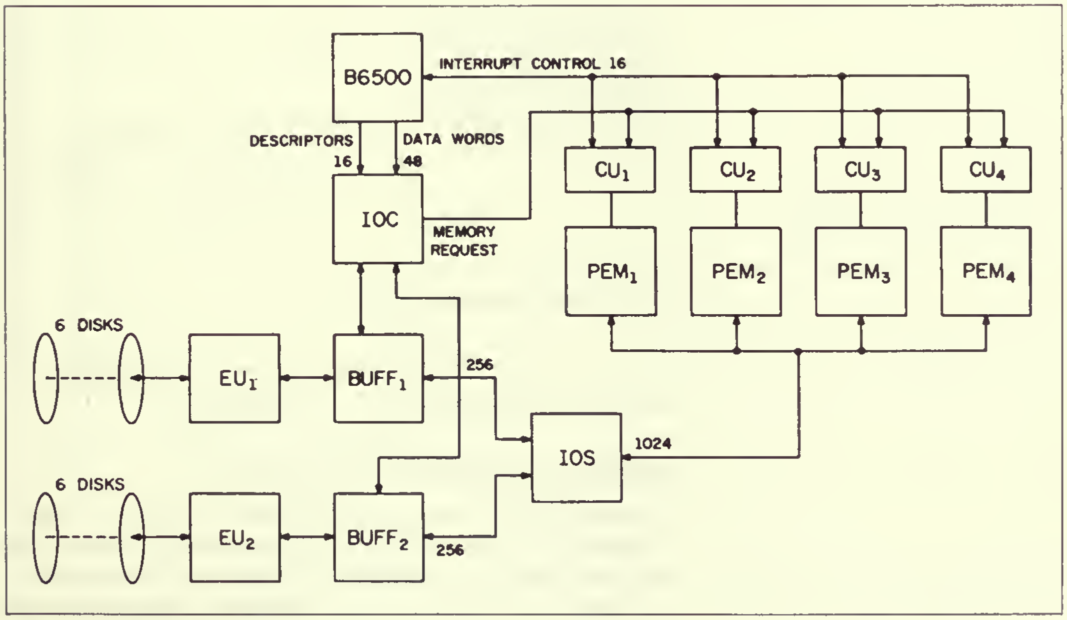

Fig. 9. I/O data path.

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

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

Processing element memory (PEM)

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

Disk-file subsystem

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

B 6500 control computer

The B 6500 computer is assigned the following functions.

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

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

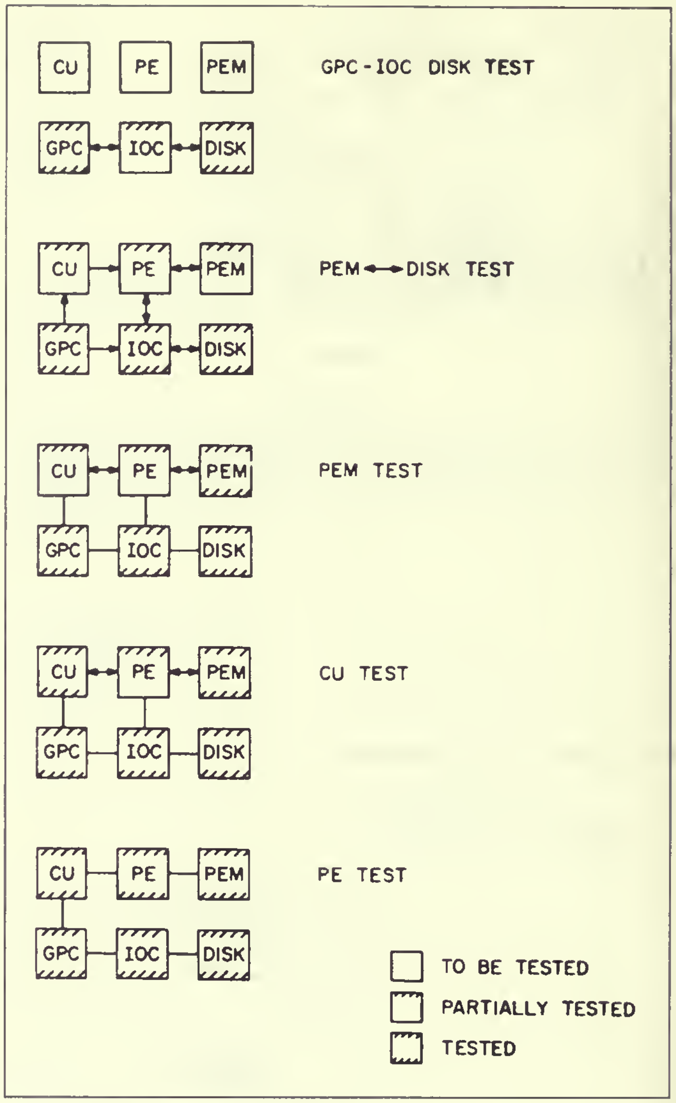

Fig. 10. System diagnostic sequence.

Reliability and maintenance of the ILLIAC IV

The progress in computer components from vacuum tubes to semiconductors over several generations has improved the mean-time-between-failures for computers from tens of hours to several thousand hours. By using larger scale integration, a tenfold increase in number of gates per system should be possible with comparable reliability.

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

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

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

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

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

References

HollJ59; KuckD68; MurtJ66; SlotD62; UngeS58

APPENDIX 1
A1. CLASSIFIED LIST OF CU INSTRUCTIONS

A1.1 Data transmission

ALIT	Add literal (24 bit) to CAR.
BIN	Block fetch to CU memory.
BINX	Indexed (by PE index) block fetch.
BOUT	Block store from CU memory.
BOUTX	Indexed block store.
CLC	Clear CAR.
COPY	Copy CAR into CAR of other quadrant.
DUPI	Duplicate inner half of CU memory address contents into both halves of CAR.
DUPO	Duplicate outer half of CU memory address contents into both halves of CAR.
EXCHL	Exchange contents of CAR with CU memory address contents.
LDL	Load CAR from CU memory address contents.
LIT	Load CAR with 64-bit literal following the instruction.
LOAD	Load CU memory from contents of PE memory address found in CAR.
LOADX	Load CU memory from contents of PE memory address found in CAR, indexed by PE index.
ORAC	OR all CARS in array and place in CAR.
SLIT	Load CAR with 24-bit literal.
STL	Store CAR into CU memory.
STORE	Store CAR into PE memory.
STOREX	Store CAR into PE memory, indexed by PE index.
TCCW	Transmit CAR counterclockwise between CUs in array.
TCW	Transmit CAR clockwise between CUs in array.

A1.2 Skip and test

$\mathrm{CTSB}\left\{\begin{matrix}\mathrm{T, A}\\ \mathrm{F}\end{matrix}\right\}$	Skip on *n*th bit of CAR. If *T* is present, skip if 1; if *F* is present, skip if 0. If *A* is present, AND together bits from all CUs in array before testing; if absent, OR together bits from all CUs in array before testing.
4 Instructions:	CTSBT, CTSBTA, CTSBF, CTSBFA.
$\mathrm{EQL}\left\{\begin{matrix}\mathrm{T, A}\\ \mathrm{F}\end{matrix}\right\}$	Skip on CAR equal to CU memory address contents. The letters *T*, *F*, and *A* have the same meaning as in CTSB above.
4 Instructions:	EQLT, EQLTA, EQLF, EQLFA.
$\mathrm{EQLX}\left\{\begin{matrix}\mathrm{T, A}\\ \mathrm{F}\end{matrix}\right\}$	Skip on index portion of CAR (bits 40 through 63) equal to bits 40 through 63 of CU memory address contents. The letters *T*, *F*, and *A* have the same meaning as in CTSB above.
4 Instructions:	EQLXT, EQLXTA, EQLXF, EQLXFA.
$\mathrm{GRTR}\left\{\begin{matrix}\mathrm{T, A}\\ \mathrm{F}\end{matrix}\right\}$	Skip on index part of CAR (bits 40 through 63) greater than bits 40 through 63 of CU memory address contents. The letters *T*, *F*, and *A* have the same meaning as in CTSB above.
4 Instructions:	GRTRT, GRTRTA, GRTRF, GRTRFA.
$\mathrm{LESS}\left\{\begin{matrix}\mathrm{T, A}\\ \mathrm{F}\end{matrix}\right\}$	Skip on index part of CAR (bits 40 through 63) less than bits 40 through 63 of CU memory address contents. The letters *T*, *F*, and *A* have the same meaning as in CTSB above.
4 Instructions:	LESST, LESSTA, LESSF, LESSFA.
$\mathrm{ONES}\left\{\begin{matrix}\mathrm{T, A}\\ \mathrm{F}\end{matrix}\right\}$	Skip on CAR equal to all 1's. The letters *T*, *F*, and *A* have the same meaning as in CTSB above.
4 Instructions:	ONEST, ONESTA, ONESF, ONESFA.
$\mathrm{ONEX}\left\{\begin{matrix}\mathrm{T, A}\\ \mathrm{F}\end{matrix}\right\}$	Skip on bits 40 through 63 of CAR equal to all 1's. The letters *T*, *F*, and *A* have the same meaning as in CTSB above.
4 Instructions:	ONEXT, ONEXTA, ONEXF, ONEXFA.
$\mathrm{SKIP}\left\{\begin{matrix}\mathrm{T, A}\\ \mathrm{F}\end{matrix}\right\}$	Skip on *T–F* flip-flop previously set. The letters *T*, *F*, and *A* have the same meaning as in CTSB above.
4 Instructions:	SKIPT, SKIPTA, SKIPF, SKIPFA.
SKIP	Skip unconditionally.
$\mathrm{TXL}\left\{\begin{matrix}\mathrm{T, A, I}\\ \mathrm{F}\end{matrix}\right\}$	Skip on index portion of CAR (bits 40 through 63) less than limit portion (bits 1 through 15). The letters *T*, *F*, and *A* have the same meaning as in CTSB above. If *I* is present, the index portion of CAR is incremented by the increment portion of CAR (bits 16 through 39) while the test is in progress; if *I* is not present, no incrementing takes place.
8 Instructions:	TXLT, TXLTI, TXLTA, TXLTAI, TXLF, TXLFI, TXLFA, TXLFAI.
$\mathrm{TXE}\left\{\begin{matrix}\mathrm{T, A, I}\\ \mathrm{F}\end{matrix}\right\}$	Skip on index portion of CAR (bits 40 through 63) equal to limit portion of CAR (bits 1 through 15). See CTSB for the meaning of *T*, *F*, and *A*; see TXL above for the meaning of *I*.

8 Instructions:	TXET, TXETI, TXETA, TXETIA, TXEF, TXEFI, TXEFA, TXEFIA.
$\text{TXG}\begin{Bmatrix}\text{T, A, I}\\ \text{F}\end{Bmatrix}$	Skip on index portion of CAR (bits 40 through 63) greater than limit portion of CAR (bits 1 through 15). See CTSB for the meaning of *T*, *F*, and *A*; see TXL above for the meaning of *I*.
8 Instructions:	TXGT, TXGTI, TXGTA, TXGTAI, TXGF, TXGFI, TXGFA, TXGFAI.
$\text{ZER}\begin{Bmatrix}\text{T, A}\\ \text{F}\end{Bmatrix}$	Skip on CAR all 0's. See CTSB for the meaning of *T*, *F*, and *A*.
4 Instructions:	ZERT, ZERTA, ZERF, ZERFA.
$\text{ZERX}\begin{Bmatrix}\text{T, A}\\ \text{F}\end{Bmatrix}$	Skip on index portion of CAR (bits 40 through 63) all 0's. See CTSB for the meaning of *T*, *F*, and *A*.
4 Instructions:	ZERXT, ZERXTA, ZERXF, ZERXFA.

A1.3 Transfer of control

EXEC	Execute instruction found in bits 32 through 63 of CAR.
EXCHL	Exchange contents of CAR with contents of CU memory address.
HALT	Halt ILLIAC IV.
JUMP	Jump to address found in instruction.
LOAD	Load CU memory address contents from contents of PE memory address found in CAR.
LOADX	Load CU memory address contents from contents of PE memory address found in CAR, indexed by PE index.
STL	Store CAR into CU memory.

A1.4 Route

RTE	Route. Routing distance is found in address field (CAR indexable), and register connectivity is found in the skip field.

A1.5 Arithmetic

ALIT	Add 24-bit literal to CAR.
CADD	Add contents of CU memory address to CAR.
CSUB	Subtract contents of CU memory address from CAR.
INCRXC	Increment index word in CAR.

A1.6 Logical

CAND	AND CU memory to CAR.
CCB	Complement bit of CAR.
CLC	Clear CAR.
COR	OR CU memory to CAR.
CRB	Reset bit of CAR.
CROTL	Rotate CAR left.
CROTR	Rotate CAR right.
CSB	Set bit of CAR.
CSHL	Shift CAR left.
CSHR	Shift CAR right.
LEADO	Detect leading ONE in CAR of all quadrants in array.
LEADZ	Detect leading ZERO in CAR of all quadrants in array.
ORAC	OR all CARS in array and place in CAR.

A2. CLASSIFIED LIST OF PE INSTRUCTIONS

A2.1 Data transmission

LDA	Load *A* register.
LDB	Load *B* register
LDR	Load *R* register.
LDS	Load *S* register.
LDX	Load *X* register.
LDC0	Load CAR 0 from PE register.
LDC1	Load CAR 1 from PE register.
LDC2	Load CAR 2 from PE register.
LDC3	Load CAR 3 from PE register.
LEX	Load exponent of *A* register.
ONES	Load all ONES into *A* register.
STA	Store *A* register.
STB	Store *B* register.
STC	Store *C* register.
STR	Store *R* register.
STS	Store *S* register.
STX	Store *X* register.
SWAPA	Interchange inner and outer contents of *A* register.
SWAP	Interchange the contents of *A* register and *B* register.
SWAPX	Interchange outer operand of *A* register and inner operand of *B*.

A2.2 Index operations

$\text{IX}\begin{Bmatrix}\text{L}\\ \text{E, I}\\ \text{G}\end{Bmatrix}$	Set *I* on comparison of *X* register and operand. The presence of *L* means set *I* if *X* is less than operand; the presence of *E* means set *I* if *X* is equal to operand; the presence of *G* means set *I* if *X* is greater than operand. If *I* is present, increment *X* while performing test; if *I* is absent, do not

6 Instructions:	IXL, IXLI, IXE, IXEI, IXG, IXGI.
$JX\left\{\begin{matrix}L\\E, I\\G\end{matrix}\right\}$	Set *J* on comparison of *X* register and operand. See above for meaning of *L*, *E*, *G*, and *I*.
6 Instructions:	JXL, JXLI, JXE, JXEI, JXG, JXGI.
XI	Increment PE index (*X* register) by bits 48 through 63 of operand.
XIO	Increment PE index of bits 48 through 63 of operand plus one.

A2.3 Mode setting/comparisons

EQB	Test *A* and *B* for equality bytewise.
GRB	Test *B* register greater than *A* register bytewise.
LSB	Test *B* register less than *A* register bytewise.
CHWS	Change word size.
$I\left\{\begin{matrix}L\\A\\M\end{matrix}\right\}L$	Set *I* if *A* register is less than operand. *L* means test logical; *A* means test arithmetic; *M* means test mantissa.
3 Instructions:	ILL, IAL, IML.
$I\left\{\begin{matrix}L\\A\\M\end{matrix}\right\}E$	Set *I* if *A* register is equal to operand. See above for meaning of *L*, *A*, and *M*.
3 Instructions:	ILE, IAE, IME.
$I\left\{\begin{matrix}L\\A\\M\end{matrix}\right\}G$	Set *I* if *A* register is greater than operand. See above for meaning of *L*, *A*, and *M*.
3 Instructions:	ILG, IAG, IMG.
$I\left\{\begin{matrix}L\\A\\M\end{matrix}\right\}Z$	Set *I* if *A* register is equal to all zeros.
3 Instructions:	ILZ, IAZ, IMZ.
$I\left\{\begin{matrix}L\\A\\M\end{matrix}\right\}O$	Set *I* if *A* register is equal to all ONES.
3 Instructions:	ILO, IAO, IMO.
$J\left\{\begin{matrix}L\\A,\\M\end{matrix}\right.\left.\begin{matrix}L\\E\\G\end{matrix}\right\}\begin{matrix}Z\\O\end{matrix}$	Set *J* under conditions specified in set of instructions immediately above.
15 Instructions:	JLL, JAL, JML, JLE, JAE, JME, JLG, JAG, JMG, JLZ, JAZ, JMZ, JLO, JAO, JMO.
$IX\left\{\begin{matrix}L\\E, I\\G\end{matrix}\right\}$	Set *I* on comparison of X register and operand. See Section A2.2 for meaning of *L*, *E*, *G*, and *I*.
6 Instructions:	IXL, IXLI, IXE, IXEI, IXG, IXGI.
$JX\left\{\begin{matrix}L\\E, I\\G\end{matrix}\right\}$	Set *J* on comparison of *X* register and operand. See Section A2.2 for meaning of *L*, *E*, *G*, and *I*.
6 Instructions:	JXL, JXLI, JXE, JXEI, JXG, JXGI.
$IS\left\{\begin{matrix}L\\E\\G\end{matrix}\right\}$	Set *I* on comparison of *S* register and operand. See Section A2.2 for meaning of *L*, *E*, and *G*.
3 Instructions:	ISL, ISE, ISG.
$JS\left\{\begin{matrix}L\\E\\G\end{matrix}\right\}$	Set *J* on comparison of *S* register and operand. See Section A2.2 for meaning of *L*, *E*, and *G*.
3 Instructions:	JSL, JSE, JSG.
ISN	Set *I* from the sign bit of *A* register.
JSN	Set *J* from the sign bit of *A* register.
SETE	Set *E* bit as a logical function of other bits.
SETEO	Set *E*1 bit similarly.
SETF	Set *F* bit similarly.
SETFO	Set *F*1 bit similarly.
SETG	Set *G* bit similarly.
SETH	Set *H* bit similarly.
SETI	Set *I* bit similarly.
SETJ	Set *J* bit similarly.
SETC0	Set *P*th bit of CAR 0 similarly.
SETC1	Set *P*th bit of CAR 1 similarly.
SETC2	Set *P*th bit of CAR 2 similarly.
SETC3	Set *P*th bit of CAR 3 similarly.
IBA	Set *I* from *N*th bit of *A* register; bit number is found in address field.
JBA	Set *J* from *N*th bit of *A* register; bit number is found in address field.

A2.4 Arithmetic

ADB	Add bytewise.
SBB	Subtract operand from *A* register bytewise.
ADD	Add *A* register and operand as 64-bit operands.
SUB	Subtract operand from *A* register as 64-bit quantities.
AD{R, N, M, S}	Add operand to *A* register. The *R*, *N*, *M*, *S* specify all possible variants of the arithmetic instruction. The meaning of each letter, if present in the mnemonic, is

R	round result
N	normalize result
M	mantissa only
S	special treatment of signs.

16 Instructions:	ADM, ADMS, ADNM, ADNMS, ADN, ADNS, ADRM, ADRMS, ADRM, ADRNMS, ADRN, ADRNS, ADR, ADRS, AD, ADS.
ADEX	Add to exponent.
DV{R, N, M, S}	Divide by operand. See AD instruction for meaning of *R*, *N*, *M*, and S.
16 Instructions:	DVM, DVMS, DVNM, DVNMS, DVN, DVNS, DVRM, DVRMS, DVRNM, DVRNS, DVRN, DVRNS, DVR, DVRS, DV, DVS.
EAD	Extend precision after floating point ADD.
ESB	Extend precision after floating point SUBTRACT.
LEX	Load exponent of *A* register.
ML{R, N, M, S}	Multiply by operand. See AD instruction for meaning of *R*, *N*, *M*, and *S*.
16 Instructions:	MLM, MLMS, MLNM, MLNMS, MLN, MLNS, MLRM, MLRMS, MLRNM, MLRNMS, MLRN, MLRNS, MLR, MLRS, ML, MLS.
SAN	Set *A* register negative.
SAP	Set *A* register positive.
SBEX	Subtract exponent of operand from exponent of *A* register.
SB{R, N, M, S}	Subtract operand from *A* register. See AD instruction for meaning of *R*, *N*, *M*, and S.
16 Instructions:	SBM, SBMS, SBNM, SBNMS, SBN, SBNS, SBRM, SBRMS, SBRNM, SBRNMS, SBRN, SBRNS, SBR, SB, SBS.
NORM	Normalize *A* register.
MULT	In 32-bit mode, perform MULTIPLY and leave outer result in *A* register and inner result in *B* register, with both results extended to 64-bit format.

A2.5 *Logical*

$\begin{Bmatrix}N\\Z\\O\end{Bmatrix}$ AND $\begin{Bmatrix}N\\Z\\O\end{Bmatrix}$	AND *A* register with operand. The left-hand set of letters specifies a variant on the *A* register, the right-hand set, on the operand. The meaning of these variants is

not present	use true
N	use complement
Z	use all ZEROS
O	use all ONES.

16 Instructions:	AND, ANDN, ANDZ, ANDO, NAND, NANDN, NANDZ, NANDO, ZAND, ZANDN, ZANDZ, ZANDO, OAND, OANDN, OANDZ, OANDO.
CBA	Complement bit of *A* register.
CHSA	Change sign of *A* register.
$\begin{Bmatrix}N\\Z\\O\end{Bmatrix}$ EOR $\begin{Bmatrix}N\\Z\\O\end{Bmatrix}$	Exclusive OR *A* register with operand.
16 Instructions:	EOR, EORN, EORZ, EORO, NEOR, NEORN, NEORZ, NEORO, ZEOR, ZEORN, ZEORZ, ZEORO, OEOR, OEORN, OEORZ, OEORO.
LEX	Load exponent of *A* register.
$\begin{Bmatrix}N\\Z\\O\end{Bmatrix}$ OR $\begin{Bmatrix}N\\Z\\O\end{Bmatrix}$	OR *A* register with operand.
16 Instructions:	OR, ORN, ORZ, ORO, NOR, NORN, NORZ, NORO, ZOR, ZORN, ZORZ, ZORO, OOR, OORN, OORZ, OORO.
RBA	Reset bit *A* register to ZERO.
RTAL	Rotate *A* register left.
RTAML	Rotate mantissa of *A* register left.
RTAMR	Rotate mantissa of *A* register right.
RTAR	Rotate *A* register right.
SAN	Set *A* register negative.
SAP	Set *A* register positive.
SBA	Set bit of *A* register to ONE.
SHABL	Shift *A* and *B* registers double-length left.
SHABR	Shift *A* and *B* registers double-length right.
SHAL	Shift *A* register left.
SHAML	Shift *A* register mantissa left.
SHAR	Shift *A* register right.
SHAMR	Shift *A* register mantissa right.

Chapter 20

The Illiac IV System[1]

W. J. Bouknight / Stewart A. Denenberg
David E. McIntyre / J. M. Randall
Amed H. Sameh / Daniel L. Slotnick

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.

Introduction

It all began in the early 1950's shortly after EDVAC ["Electronic Computers," 1969] 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: 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

[1]*Subsetted from Proc. IEEE*, April 1972, pp. 369–388.

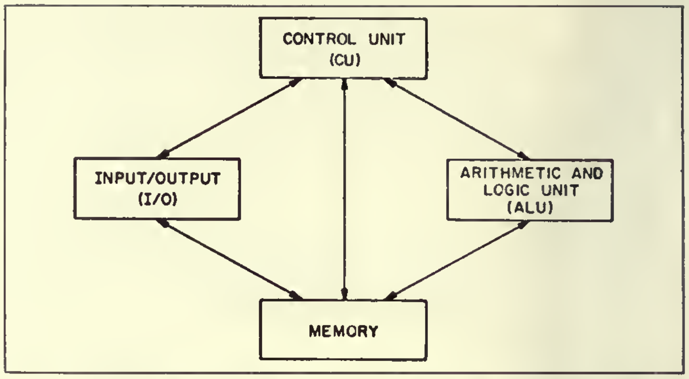

Fig. 1. Functional relations within a conventional computer. The CU has the function of fetching instructions which are stored in memory, decoding or interpreting these instructions, and finally generating the microsequences of electronic pulses which cause the instruction to be performed. The performance 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.

the middle 1960's, when the law of diminishing returns began to 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

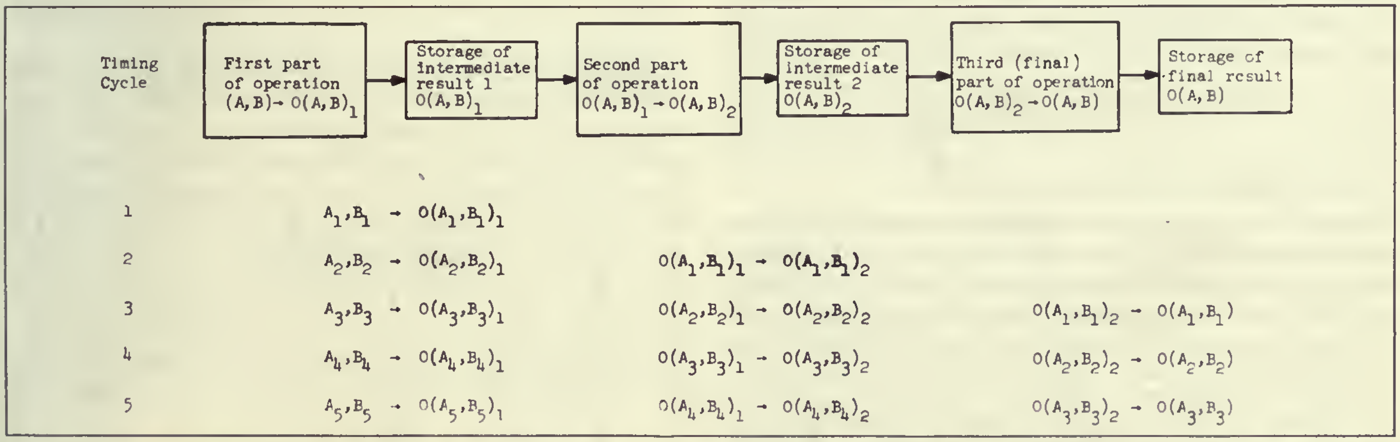
Timing Cycle
First part of operation (A,B)→ O(A,B)₁
Storage of intermediate result 1 O(A,B)₁
Second part of operation O(A,B)₁ → O(A,B)₂
Storage of intermediate result 2 O(A,B)₂
Third (final) part of operation O(A,B)₂ → O(A,B)
Storage of final result O(A,B)
1 A₁,B₁ → O(A₁,B₁)₁
2 A₂,B₂ → O(A₂,B₂)₁ O(A₁,B₁)₁ → O(A₁,B₁)₂
3 A₃,B₃ → O(A₃,B₃)₁ O(A₂,B₂)₁ → O(A₂,B₂)₂ O(A₁,B₁)₂ → O(A₁,B₁)
4 A₄,B₄ → O(A₄,B₄)₁ O(A₃,B₃)₁ → O(A₃,B₃)₂ O(A₂,B₂)₂ → O(A₂,B₂)
5 A₅,B₅ → O(A₅,B₅)₁ O(A₄,B₄)₁ → O(A₄,B₄)₂ O(A₃,B₃)₂ → O(A₃,B₃)

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.

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 [Slotnick, Borck, and McReynolds, 1962; Slotnick, 1967] 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]

Hardware Structure

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 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 Slotnick [1971]; Denenberg [1971]; and Barnes et al. [1968].

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

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

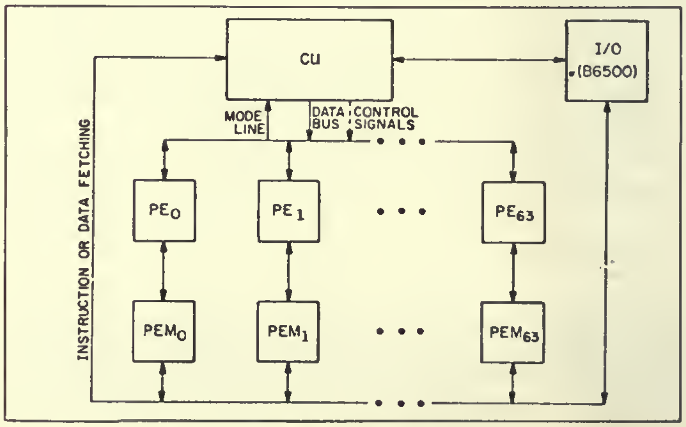

Fig. 3. Functional block diagram of Illiac IV.

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, 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.

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.

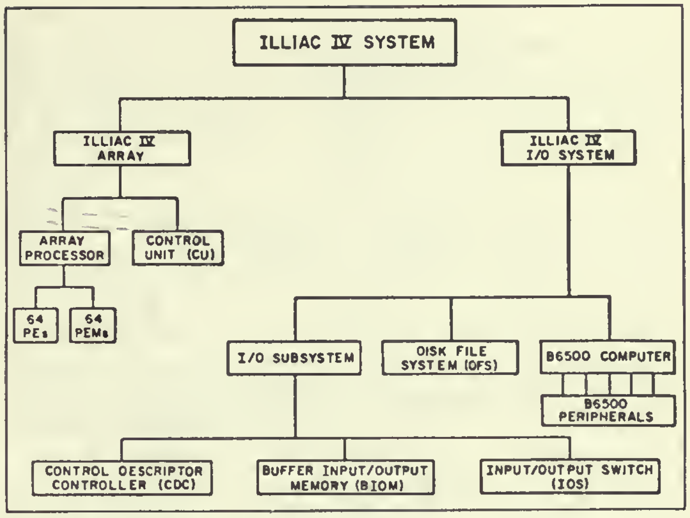

Fig. 4. Illiac IV system organization.

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.

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

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

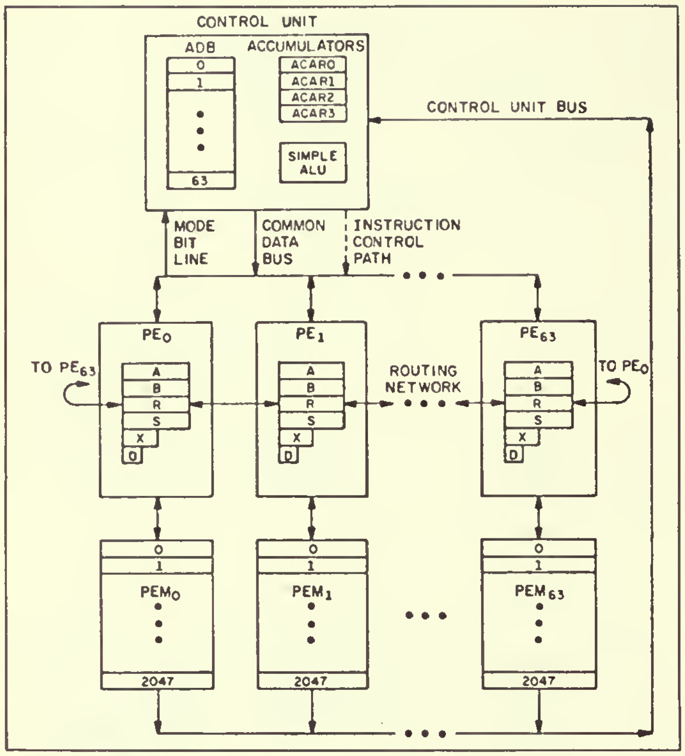

Fig. 5. Illiac IV array.

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).

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 PE's 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.

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 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.

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}.

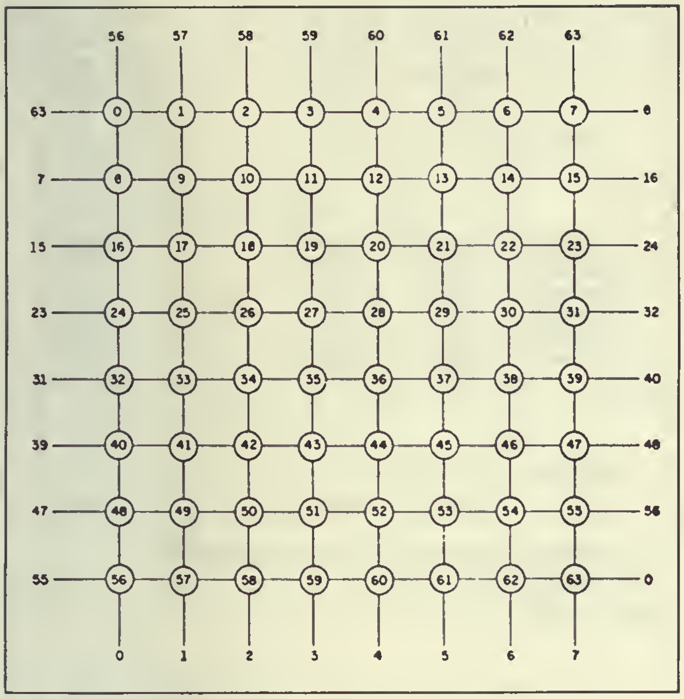

Fig. 6. PE routing connections.

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.

Some Illustrative Problems

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:

DO 10 I = 1, 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.

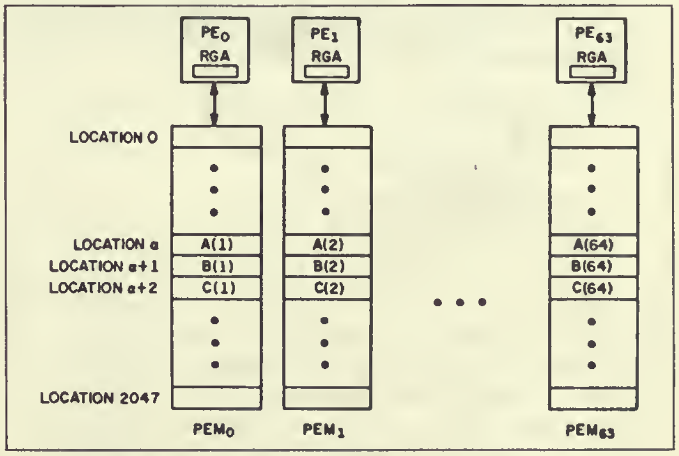

Fig. 7. Arrangement of data in PEM to accomplish
DO 10 I = 1, 64
10 $A(I) = B(I) + C(I)$.

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 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 under-utilization 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 α + 4.
ADD to RGA contents of location α + 2.
STORE result to location α.
LOAD RGA from location α + 5.
ADD to RGA contents of location α + 3.
STORE result to location α + 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 α + 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.

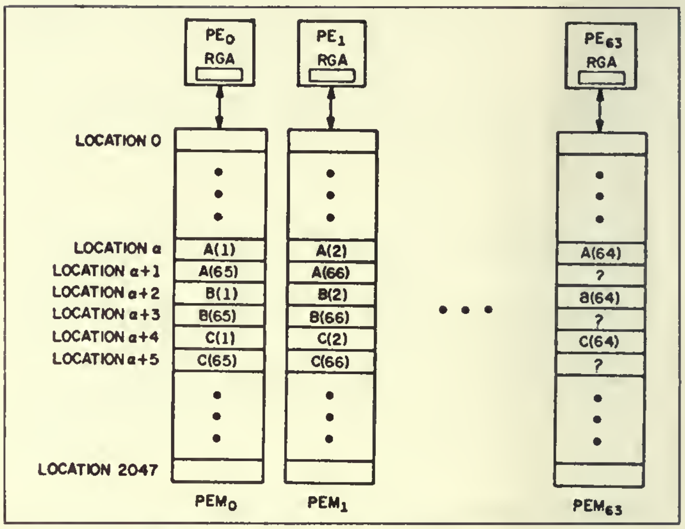

Fig. 8. Arrangement of data in PEM to accomplish DO 10 *I* = 1, 66 10 *A*(*I*) = *B*(*I*) + *C*(*I*).

Uncoupling sequential code. Finally let us consider the Fortran code:

```
   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:

$A(2) = B(2) + A(1)$
$A(3) = B(3) + A(2)$
.
.
.
$A(63) = B(63) + A(62)$
$A(64) = B(64) + A(63)$

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:

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

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

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

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 step 11.

11 Enable all PEs.

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

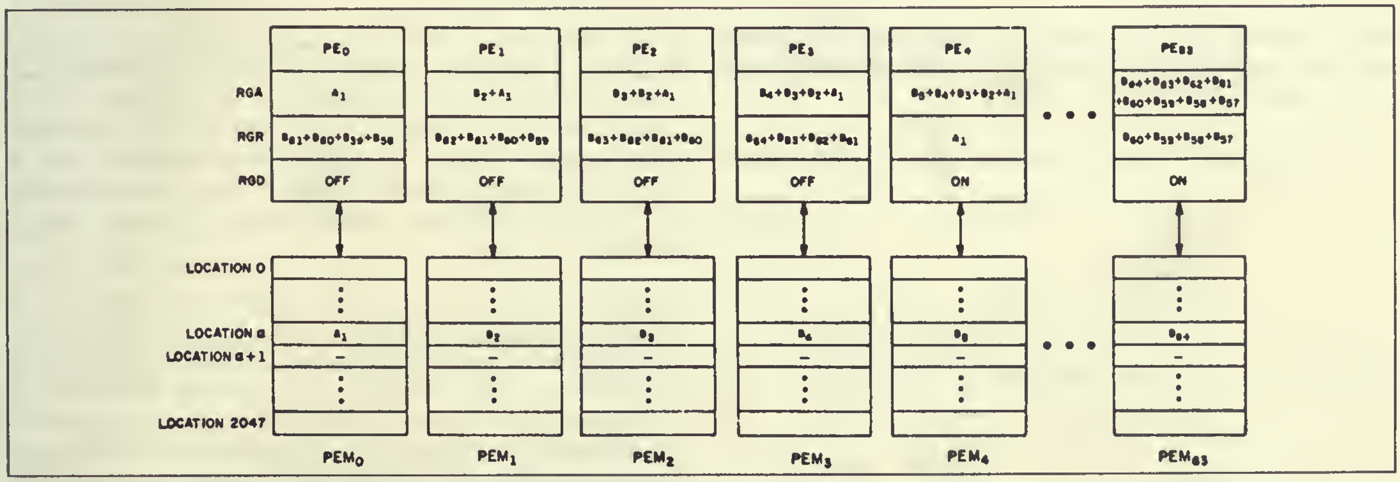

**Fig. 9. Status of data in PEM, RGA, RGR, and mode status (RGD) while executing DO 10 I = 2, 64
10 $A(I) = B(I) + A(I-1)$.
The mode status (RGD) and the contents of PEM, RGA, and RGR are shown after step 8 (I = 2) of the program.**

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 $C_i = i$ ($i = 1,2,\ldots,64$) 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.

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 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.

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.

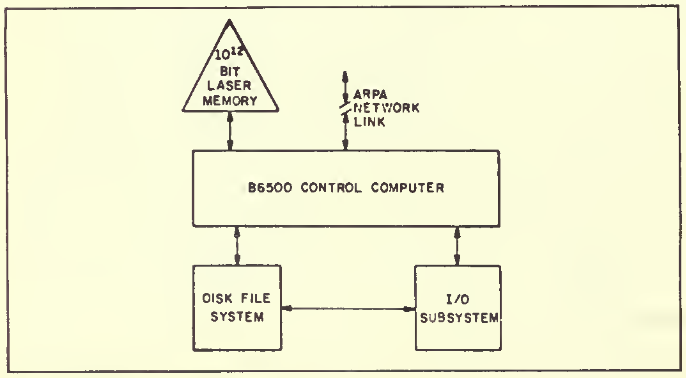

Fig. 10. Illiac IV I/O system.

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.

B6500 control computer. The 6500 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

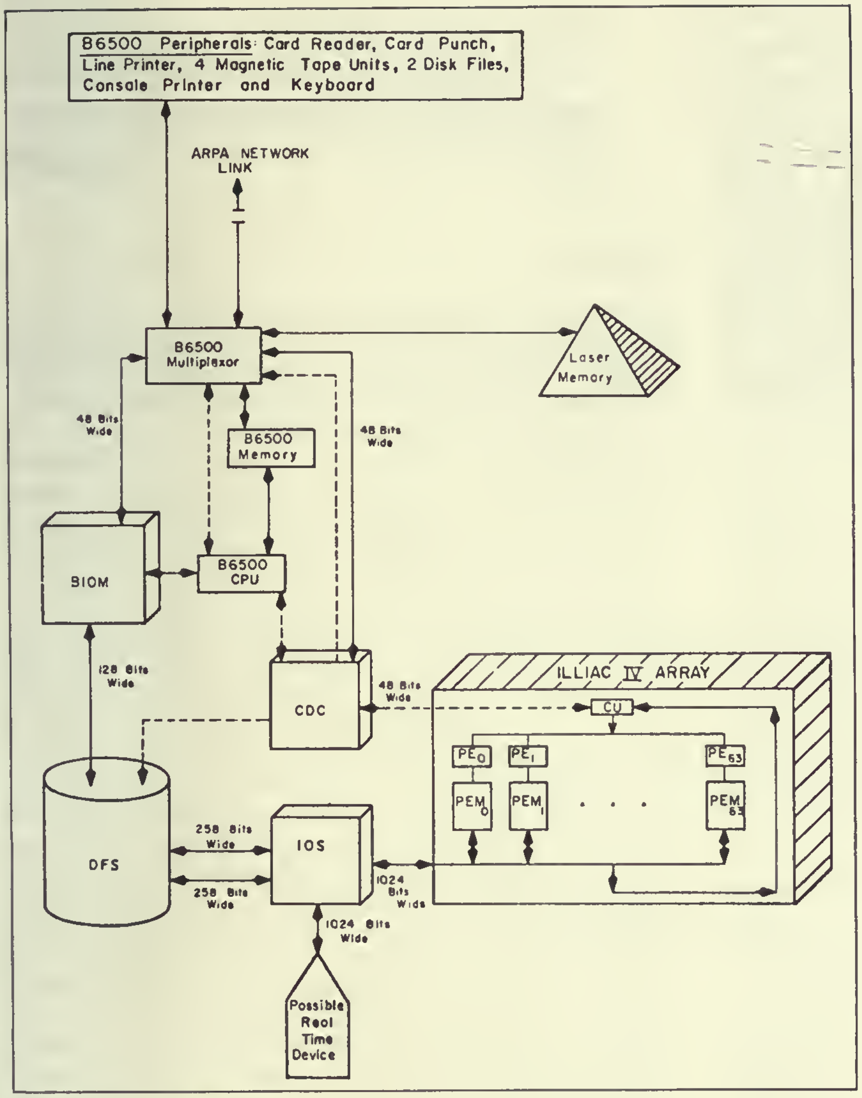

Fig. 11. Illiac IV system.

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.

References

Bouknight, Denenberg, McIntyre, Randall, Sameh, and Slotnick [1972]; Barnes, Brown, Kato, Kuck, Slotnick, and Stokes [1968]; Denenberg [1971]; "Electronic Computers" [1969]; Slotnick [1967]; Slotnick [1971]; Slotnick, Borck, and McReynolds [1962].