(1)

AD-A954 882

DTIC FILE COPY

This document has been approved for public release and sale; its distribution is unlimited.

DTIC
AUG 8 1985
E

85-8-01-158

John A. Perry
ARPA/IPT
1400 Wilson Blvd.
Arlington, Va. 22209

ILLIAC IV Document No. 225

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

AN INTRODUCTORY DESCRIPTION OF THE ILLIAC IV SYSTEM

Volume I

by

Stewart A. Denenberg

Department of Computer Science File No. 850
July 15, 1971

This work was supported by the Advanced Research Projects Agency, as administered by the Rome Air Development Center, under Contract No. USAF 30(602)-4144.

To Claire

Read This First

This book was written for an applications programmer who would like a tutorial description of the ILLIAC IV System before attempting to read the reference manual. As a tutorial, the level of detail presented in this book is fairly general; particular information can be found in the Burroughs Reference Manual "ILLIAC IV Systems Characteristics and Programming Manual."

In order to use this book most effectively, the Chapters should be read in order. The reader who wants a very quick look at the capabilities of ILLIAC IV may skim just the summaries of parts A, B and C of Chapter I and begin reading on page I-55. He may then read pages II-1 through II-20, skipping the detailed description of the ILLIAC IV Array (pages II-21 to II-41). Pages II-41 through II-73 are optional; the reader should at least look at them and decide for himself. As much of Chapter III as possible should be read--the instruction repertoire, more than anything else, defines the capabilities of a computer. A valid answer to the question "What is ILLIAC IV?" would be to hand the questioner a description of each instruction in the repertoire.

For a more complete understanding, however, the reader should come back and read the sections he skipped on the first pass. It is the nature of ILLIAC IV, to a degree much greater than the conventional computers, that its hardware structure is bound up very closely with its

capabilities. It is therefore necessary that the reader spend the time necessary to understand the architecture of ILLIAC IV.

The Table of Contents in the front of the book is in an abbreviated format while each chapter will be preceded by a finer Table of Contents. A Hardware Glossary which is essentially a glossary for Chapter II is at the end of the book.

Chapter I presents the background concepts necessary for an understanding of ILLIAC IV. A short section is devoted to the historical development of digital computers and their evolution is described in terms of the problems that had to be solved. After conventional computer organizations are described, unconventional ones are presented as design options to speed up the operation of a computer. Two design philosophies, overlap and replication, represent two major methods used to increase the computer's operational speed. Overlap is effected by the buffer and pipeline mechanism and replication is embodied in the general multiprocessor. ILLIAC IV is shown to be a variant of a general multiprocessor using buffering and a modified pipeline mechanism in the instruction execution section.

Chapter II describes the architecture or the hardware structure of ILLIAC IV. The ILLIAC IV Array is discussed in broad terms followed by some illustrative problems which point out some of the similarities and differences between problem-solving on sequential and parallel machines. The problems also serve to illustrate how the hardware components are tied

together. Following is a more detailed description of the ILLIAC IV Array, then another illustrative problem (Laplace's equation describing steady-state temperature distribution in two-dimensions) followed by some data allocation considerations; the ILLIAC IV I/O System is discussed briefly, and some conclusions and opinions end the chapter.

Chapter III presents the Assembly Language ASK in a functional and pragmatic way: a problem is described and then only those ASK instructions necessary for the solution are described. In this way the five problems introduce forty ASK instructions and the flavor of the assembly language which, from a programmer's standpoint, is an indication of the capabilities of ILLIAC IV itself. The five problems are: Summing an array of numbers, Finding the maximum value in an array of numbers, Matrix multiplication, Matrix transpose, and Laplace's equation described in Chapter II.

This book will be issued in three volumes. The first three Chapters represent Volume 1, Chapters IV through VII will comprise Volume 2, and Chapters VIII through XI will be Volume 3. Volumes 2 and 3 will be supplied as soon as they are available.

Abstract

Written specifically for an applications programmer, the book presents a tutorial description of the ILLIAC IV System. Volume 1 contains three chapters -- Background, Hardware Structure, and The Assembly Language-- ASK, as well as a Hardware Glossary. Many illustrative problems are used to educate the beginner in the use of the ILLIAC IV System.

TABLE OF CONTENTS

Volume 1

Volume 2

Volume 3

Foreword

This book is based upon the many reports and documents generated at the University of Illinois and the Burroughs Corporation during the design and development of the ILLIAC IV computer. In addition, much of the content of the book was influenced by the material offered in the graduate level computer science course, CS 491, "Architecture, Applications, and Languages for a Parallel Computer" as well as the many one-day, two-day and one-week seminars on ILLIAC IV. I learned a great deal from my "students".

I would like particularly to thank Professor Daniel Slotnick and my friend Mr. George Westlund who provided the overall guidance for this book and whose idea it was to create it in the first place. Much specific help was given me by Walt Heimerdinger in the area of hardware structure, and Jim Stevens and John McMillan in the area of ASK. Mike Sher and Cal Corbin helped proofread and make final suggestions before this book when to press. I am also very grateful to Joyce Fasnacht who cheerfully typed and retyped the many versions of the text with incredible accuracy, and who drafted the original versions of all of the figures from my pencil scratchings.

Any errors you may find are not only my responsibility but become yours also. If you inform me of them I will correct them in the next edition.

Stewart A. Denenberg
Urbana, Illinois
1971

CHAPTER I -- BACKGROUND

TABLE OF CONTENTS

LIST OF FIGURES

TABLES

CHAPTER I

BACKGROUND

A. Summary

Chapter I traces out some of the background concepts necessary for an understanding of ILLIAC IV. A short section is devoted to the historical development of digital computers, indicating how computer systems evolved to the Von Neumann state of organization. Also discussed is the tendency computers have had in creating problems themselves. The first computers were designed to solve specific applications problems such as computing a table of values for a certain mathematical function or solving a differential equation which described the ballistic path of an artillery shell. As computers became more useful, they started to contribute problems of their own to be solved such as the need for easier-to-use programming languages. The most pressing of these problems was the need for faster and faster operating speeds. If the computer could be made to process information at a faster rate, and costs could be held constant, then the per-unit-time cost of processing information would be effectively lowered. The remaining sections of the Chapter describe how Von Neumann organization may be modified to increase operational speed. Two design philosophies to achieve increased speed are discussed: 1) Overlapping the operation of two or more of the functional components of a conventional computer and 2) Replication of one or more of the functional components many times. Since these philosophies are not mutually exclusive, a third option exists whereby both 1) and 2) are effected.

Overlap can be achieved by utilizing the Buffer and Pipeline mechanisms; however, the Pipeline is limited to the number of stages into which an operation can be decomposed, and ultimately by the speed of light. The replication philosophy is typified by the general Multiprocessor, but the cost is extremely high. Various re-designs of the Multiprocessor are explored in order to reduce its high cost: Re-centralizing Memory, the Arithmetic and Logic Unit, or the Control Unit. ILLIAC IV is represented as a Multiprocessor with the Control Unit re-centralized. This particular option was chosen for two main reasons: 1) much of the cost of a digital computer is tied up in the Control Unit and 2) there are large classes of problems that can be solved by a single instruction stream which operates on data that can be structured as a vector. ILLIAC IV also utilizes the Buffer and modified Pipeline mechanisms to overlap the operation of its instruction execution unit.

B. A Review of Digital Computing Machines

1. Summary

Perhaps the first computer was a coin. If a computer is a tool used by man to solve a problem, then a coin fits the description. A coin was (and still is) used as a tool to help men make decisions. It is a true binary decision maker: a flip of a coin and a decision is automatically made: heads, one course of action is taken--tails, another. Whether the first computer was a coin, an abacus, Pascal's Calculator or Jacquard's Loom is not argued here; instead the starting point is arbitrarily chosen

with Babbage's machines. The Automatic Sequence Controlled Calculator (Mark I), ENIAC, EDSAC, the University of Manchester Computers, and EDVAC are used to represent the major chain of machines which evolved to the Von Neumann organization.

2. Babbage's Difference Engine and Analytical Engine

a. The Difference Engine

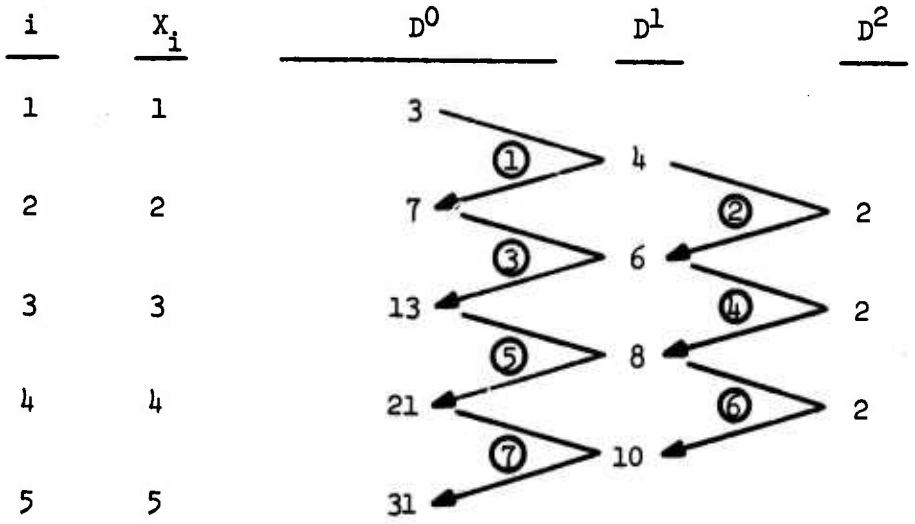
In 1812, when Charles Babbage was an undergraduate at Trinity College, Cambridge, mathematical tables of functions were generated by hand. The production of a table of values for just one mathematical function was a tedious and cumbersome job. A group of over 100 people, called "computers," were trained to follow a finite difference algorithm to compute values of the function over a specific range and for specific interval widths within the specified range.

Let us consider how the function $f(X) = X^2 + X + 1$ would be calculated over the range $1 \leq X \leq 5$ and for an interval width of 1. (See Table I-1.)

It was known at the time that the nth differences of an n-degree polynomial are constant. By convention, the zero-differences (D^0) are the values of the function. D_i^0 is the value of the function at X_i: $f(X_i) \equiv D_i^0$. For the simple example used here, when $i = 3$, $X_i = 3$, and $D_3^0 = 13$. The first differences (D^1) are found by subtracting previous values of D^0 from succeeding values of D^0:

Table I-1. Difference Method for Evaluating Polynomial Function $X^2 + X + 1$

$f(X) = X^2 + X + 1$



Step No.	Contents of D^0	D^1	D^2
0 (Initial Value)	3	4	2
1	7	4	2
2	7	6	2
3	13	6	2
4	13	8	2
5	21	8	2
6	21	10	2
7	31	10	2

$$D_i^1 = f(X_{i+1}) - f(X_i) = D_{i+1}^0 - D_i^0$$

The second differences are the differences of the first differences and are calculated the same way:

$$D_i^2 = D_{i+1}^1 - D_i^1$$

The top part of Table I-1 shows D^0, D^1 and D^2 for the X values. Note that the second differences are constant (the value 2). Also shown is how we can work backwards if we are given D_1^0, D_1^1, and D_1^2 by summing instead of subtracting:

$$D_{i+1}^0 = D_i^0 + D_i^1$$

and

$$D_{i+1}^1 = D_i^1 + D_i^2$$

Therefore, if we have 3 registers to store the values of D_0, D_1 and D_2 as we sequentially apply the above two equations we can generate D_i^0 for as long as we wish to compute. All we need are the 3 initial values

$$D_1^0 = 3, \quad D_1^1 = 4 \text{ and } D_1^2 = 2$$

In Table I-1, each step is numbered and the direction of addition indicated by an arrow.

The lower part of Table I-1 displays the contents of the three registers, D^0, D^1 and D^2 after each step circled in the top part of the table.

It was Babbage's contention that not only could a mechanical machine be built to perform the finite difference algorithm, it would be faster and much more accurate. He had even designed his Difference Engine to print the results directly from the wheels which displayed the numbers, thus eliminating the possibility of a human transcription error.

Babbage fabricated a small Difference Engine which could tabulate a second degree polynomial (or any other function whose second differences were constant) to 8 decimal digits of accuracy. In 1823 he was given a grant by the British government to build a machine that could generate tables for a function whose seventh differences were constant to an accuracy of 20 decimal digits. His ambitious project was never completed. Work stopped in 1833 when Babbage ran into financial difficulties with his engineer who resigned from the project taking with him all of the specially constructed tools for the building of the Engine (under English law at the time, the engineer had the right to do so).

Babbage was probably the first computer designer to run into financial difficulties because the state-of-the-art of technology lagged too far behind the state-of-the-art of conception. His ideas were sound, but his funds were hopelessly inadequate to create the technology which in turn would be used to create his computer.

The Difference Engine was more than just an automatic calculator capable of addition, subtraction and multiplication--it could also perform a procedure or a program. There was only <u>one</u> program that it could perform, however, and that was the finite difference algorithm. From the

point of view of modern computing, the Difference Engine was a single-purpose computer with no program software; the program was intrinsically part of the machine, imbedded into the configuration of the gears and shafts.

b. The Analytical Engine

The Difference Engine had failed, but Babbage had even greater plans for a new machine, the Analytical Engine. Either he did not realize that his machines could not be built by the existing technology or he was optimistic enough to believe he could supply the ideas for both.

Babbage designed the Analytical Engine to be able to perform more than only one algorithm; so that the program as well as the data could be supplied to the machine as an input, and the machine would process the data according to the instructions of the program.

In order to create a machine of this far-reaching capability, Babbage foresaw the four main functional sections of the modern-day computer:

- Control Unit
- Memory Unit
- Arithmetic and Logic Unit
- Input/Output Unit.

The Control Unit was to act on the same principle as the Jacquard Loom Controller: a sequence of plaques with holes punched in them drawn

over a drum by chains. Where Jacquard's Loom used a particular combination of holes in a plaque to specify a weaving operation, Babbage's Difference Engine used each plaque to store an instruction which specified an arithmetic operation. The plaques were drawn over a drum one at a time and the pattern of the holes was sensed mechanically. Each plaque instructed the Engine to perform one well-defined operation and a set of plaques, therefore, constituted a program. Groups of sets of plaques represented a Program Library.

Not only did Babbage design a machine that would execute a program of instructions, he also included the Test-and-Branch type of instruction which is at the very heart of using a program to solve a problem. In his plan, the Analytical Engine had the ability to roll the chain of instruction plaques forward or backward depending on whether the contents of a specified register turned negative during execution of the program. Rolling the chain in either direction is equivalent to a "jump" in the opposite direction in the program.

The Test-and-Branch instruction provides the programmer with an "alternate route" capability while his program is executing. Different sections of the program may be entered and executed based on the values of numbers that were computed in previous sections of the program. An additional benefit of a Test-and-Branch capability is that it affords the programmer a shorthand by which he can specify a large number of program operations with a small number of instructions. By decrementing or incrementing a register until it reaches a specified value, a section of

the program can be executed repeatedly. Without the Test-and-Branch type of capability, a programmer would have to specify every operation with at least one instruction.

The Memory or "store" as Babbage referred to it, consisted of wheels. The position of a wheel denoted the value it was storing. Numbers were transmitted to and from the "store" by means of racks. The racks were cut to engage the gears of a wheel so that the position of one wheel could be transmitted to another. The racks, of course, could be connected to rods, shafts, or other racks to further transmit the motion. (See Figure I-1.) Since each wheel would store 1 decimal digit and Babbage proposed

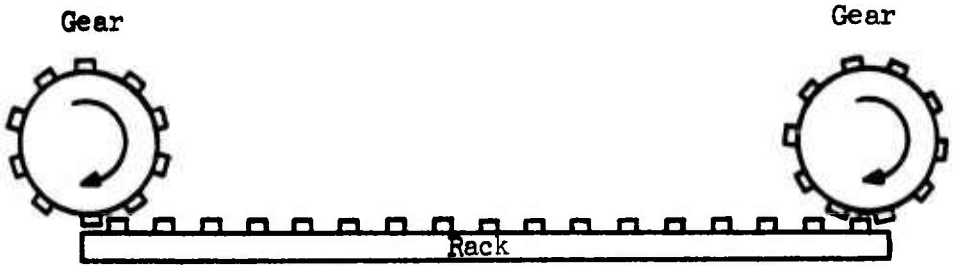


Figure I-1. Transmission of Data in Babbage's Machine

that the "store" have a capacity of 100 numbers of 50 decimal accuracy, this meant the Engine would have 50,000 wheels. Since the instructions were not stored in the memory but were punched into the plaques and thus would not be modified during program execution, Babbage's Analytical Engine was not a stored-program computer.

The Arithmetic Unit was called the "mill". Babbage went to great pains to optimize the design of the mill, particularly the problem of carrying when the sum of two numbers is greater than nine and a digit must be carried over to the next significant position. With customary fastidiousness and foresight, Babbage represented the algebraic sign of a number as a separate wheel which would not be connected to the other wheels during carries.

The Input/Output was to be effected by punch cards much like the punch cards or plaques that supplied instructions to the Engine. Some of the input was to be done manually--the initial settings of the wheels of the "store" were to be done by hand. Babbage also considered the possibility of printing output directly from the wheels of the "store" as he had with the Difference Engine. By embossing the digits on each wheel, they could be inked at the end of a calculation and the results transferred directly to paper. This not only made the results neat and legible but completely bypassed the possibility of a human transcription error.

Babbage estimated the following operation times:

Addition/Subtraction	1 second
Multiplication (50 decimals by 50 decimals)	60 seconds
Division (100 decimals by 50 decimals)	60 seconds

The description for the Analytical Engine prompted some scientifically-inclined people of the day to try their hand at programming. L. F. Menabrea, a General in the Italian Army, was at the Military Academy

in Turin when he heard Babbage speak on his Analytical Engine to the Italian mathematicians. Menabrea demonstrated how one would solve two simultaneous equations in two unknowns with Babbage's Analytical Engine.

Lady Lovelace, Lord and Lady Byron's daughter, devised many programs; among them, one to calculate Bernoulli numbers from a recurrence formula. In order to calculate the Bernoulli number B_n, n + 1 operations must be performed. Lady Lovelace described how she could store the quantity "n" in a register and decrease it by 1 each time an operation within the cycle was performed; when the number finally turned negative, the cycle had been repeated n + 1 times and control could be passed to the next part of the program. She had invented the concept of a loop.

Although Babbage did not build his Analytical Engine, he left the detailed drawings and notebocks which are currently in the Science Museum at South Kensington, England. He defined most of the concepts used in a modern computer, including the most important one which Jacquard had sensed before him: it was possible to build a machine that would automatically simulate a process if the process could be described in terms of a sequence of well-defined operations.

3. Automatic Sequence Controlled Calculator (Mark I)

Babbage's work was soon forgotten, because his Analytical Engine was never completed. In 1937, Professor Howard Aiken designed and developed an automatic calculator based on components currently available in IBM

punched card equipment. In cooperation with IBM, Aiken built and presented the calculator to Harvard University in 1944. Harvard named the calculator "Mark I".

The Control Unit of Mark I was primarily a paper tape reader. Each instruction was punched into a paper tape that was 24 holes wide and fed past a set of 24 rods that made an electrical contact if a hole existed. The first version of Mark I had no Test-and-Branch capability; the best it could do was compare two numbers in different registers and if one was greater, the machine would stop. We might say the machine had a Test-and-Stop instruction.

Mark I was later modified to include a conditional type of instruction. The conditional instruction caused control to be switched from the currently executing paper tape to any of three alternative tapes if the contents of a specified register were zero. Once control had passed to a specified alternative tape, the program was executed from instructions punched on that tape until either the program ended or control was passed back to the original or yet another tape. If control was passed back to the original tape, it would start executing where it left off by virtue of the fact that its physical position in the tape reader had not changed. Endless tapes were used for looping. This method of passing control to a new tape was faster than the method of rolling a set of cards backward or forward as Babbage had proposed, but Babbage's technique is still conceptually closer to the kind of program control that is used today. Neither Mark I nor Babbage's Analytical Engine were stored program computers.

The Storage section of Mark I consisted of wheels as did the Babbage Machine. There were 72 Accumulator Registers each capable of holding a 23-digit computed value, plus 60 sets of switches for holding constants. The switches were set manually and were not under program control.

As with Babbage's Engine, numbers were transmitted to and from Storage by rotating shafts connected to the wheel storage.

Input-Output consisted of a typewriter as well as punched-cards.

The operation speeds of Mark I were:

Addition/Subtraction	.3 seconds
Multiplication (23 digits by 23 digits)	6 seconds
Division	11.4 seconds

There was also built-in hardware which computed:

Sin (X) in 60 seconds

10^X in 61.2 seconds

and Log_{10} X in 68.4 seconds

all to 23 decimals of accuracy.

Mark I contained more than 760,000 parts and the sound of its thousands of electromechanical relays in operation has been likened to a roomful of ladies knitting.

4. Electronic Numerical Integrator and Calculator (ENIAC)

In 1946, the first electronic computer was built by J. Presper Eckert and John W. Mauchly at the University of Pennsylvania. ENIAC was built for the U. S. Army to calculate ballistic tables by integrating an ordinary differential equation. Another type of problem, the interaction of shock waves in a fluid, prompted John Von Neumann to modify some of the logical design of ENIAC.

The Memory Storage section consisted of tubes--triodes and pentodes. The flip-flops were triodes and along with the pentodes (that were used as "AND" circuits and "OR" circuits), there were over 18,000 vacuum tubes and about 1,500 relays in a 20 feet by 40 feet box for the entire machine. In addition there were about 6,000 switches for storing constants that could not be changed by the program.

The Control section consisted of a 100 kc/sec oscillator which produced pulses 2 μsec wide. As the clock generated pulses, the program was executed through the many wires that connected one part of the machine with the others. The programmer did the actual wiring through plugs, sockets, and switches; the various components of the machine were "stimulated" or not depending on whether a wire carrying a pulse reached that component. For example, if an accumulator received a program pulse it would be stimulated to add. Since both instructions and data were represented as trains of electronic pulses, a conditional operation on the sign of a number could easily be programmed by running the wire that carried the sign bit of that number to an accumulator. If a negative sign

is represented by the presence of a pulse, then the accumulator would be "stimulated" if the number was negative; if the number was positive, no pulse would appear and the accumulator would not be "stimulated" and hence not enter into the program. Thus ENIAC had its program "wired" into its hardware. ENIAC also had external switches which caused certain operations to be performed more than once, giving the programmer a looping capability (there were extra switches so that a programmer could loop within loops).

ENIAC had an advantage over Mark I in terms of speed; once initial program wiring had been done, instructions could be executed at electronic speed rather than at the speed of a paper tape reader. Changing programs, however, meant a massive rewiring job. Many hundreds of wires had to be re-plugged in order to instruct the machine to perform a different algorithm. At the time it was recognized that switch settings and plugged-wire connections could also be coded in the same way that numbers were coded. If a large capacity storage device were to become available, then the program as well as the data could be stored in the machine. Although ENIAC had only 20 storage locations, one must remember that ENIAC was a special purpose machine built to solve a specific problem--to compute values for ballistic tables, and it performed this function very well.

Each of the 20 storage locations was also an accumulator which could add, subtract, store or fetch independently and simultaneously so that its effective calculating time was very creditable:

Addition/Subtraction	200 μs
Multiplication (10 decimals by 10 decimals)	2.8 ms
Division	6 ms

The Input was 80 column IBM cards and the output was either cards or lights on a display panel.

Although ENIAC actually had its program stored inside of it in the form of wire connections, it was not a stored program machine. The definitive characteristic of a stored program computer is not the fact that a program is stored internally in the computer as opposed to outside on paper tape, for instance. A stored program computer has the ability to modify its instructions as well as its data while it is executing the instructions since both the instructions and the data are "inside" the machine using the same storage medium. Looping and indexing can be done by modifying the address field of an instruction while the program is executing. Instructions can modify, destroy or create other instructions as the program runs. (The stored program concept was responsible for the term "word" coming into use to describe what existed at a location in the memory store. In order to avoid specifying whether the content of a given storage location was to be regarded as a number or an instruction, it became convenient to refer to it as a word of storage.)

5. Electronic Delay Storage Automatic Calculator (EDSAC)

EDSAC was the first operational stored program, electronic computer. EDSAC ran its first program at the University of Manchester in May

of 1949. The EDVAC, discussed in the next section, was the first stored program, electronic computer to be designed. (Design started on the EDVAC in 1945, while design started on EDSAC at the end of 1946.)

Both EDVAC and EDSAC are considered to be IAS computers since their development was guided by the reports generated at the Institute for Advanced Study (IAS) at Princeton, New Jersey by John Von Neumann and his colleagues in 1945. IAS eventually put forth their own computer in 1952 and the ILLIAC I (University of Illinois), Johniac (RAND Corporation), MANIAC (Los Alamos) and WEIZIAC (Weizman Institute of Israel) soon followed and were patterned after the IAS machine. They all had addition times of about 60 μs and multiplication times on the order of 700 μsec.

The storage device which permitted both data and instructions to be stored together in EDSAC was a mercury delay line or ultrasonic store. (See Figure I-2.)

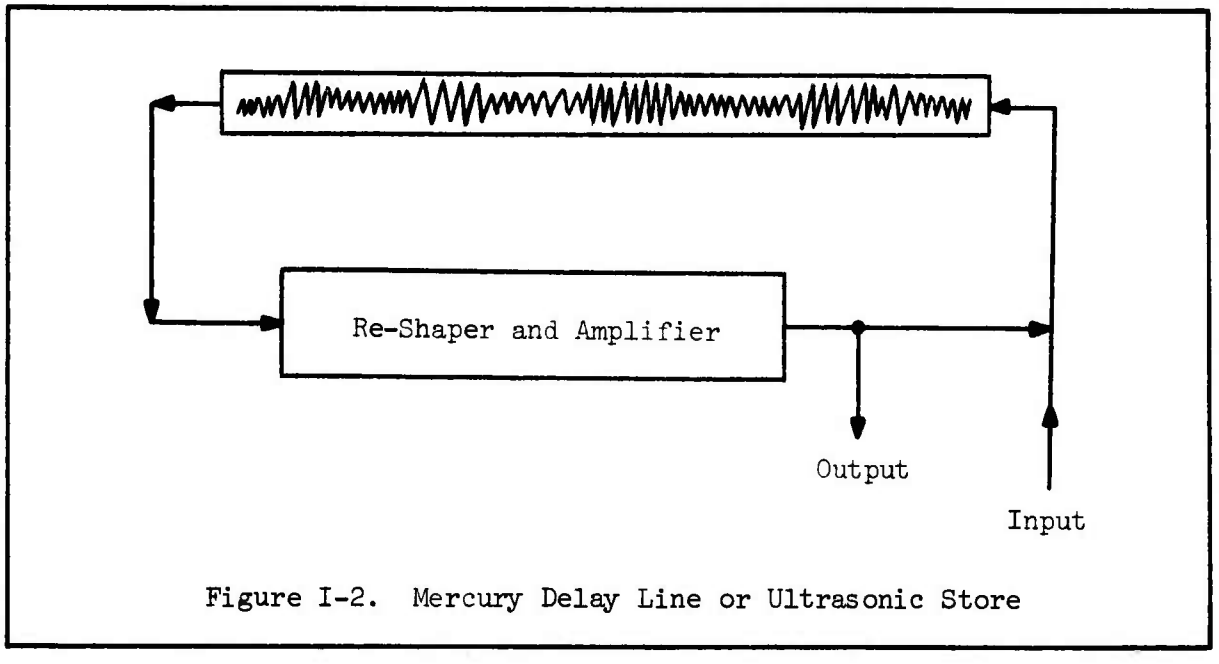


Figure I-2. Mercury Delay Line or Ultrasonic Store

A mercury delay line is a tube filled with mercury. A wire coming into the tube carries a train of electronic pulses which are transformed into mechanical vibrations by means of a piezo-electric crystal. The vibrations are transmitted through the column of mercury to another crystal at the other end of the tube which converts the mechanical vibrations into electronic signals. These signals are a bit distorted at this point, so they pass through an electronic network which reshapes and amplifies the pulses before sending them back through the tube again.

The length of the tube and the velocity of a disturbance in mercury define the memory cycle time. The number of bits that can be stored depends on the pulse rate of the clock. A major disadvantage of ultrasonic storage is the long access time. The time required for an accumulator to access a bit in storage varies from near zero, to the time it takes a bit to travel the length of the tube. The access time is on the average, one half the time it takes for a bit pulse to travel from one end of the tube to the other.

Another problem one encounters using the ultrasonic store is the interleaving of instructions and data in the pulse train so that the arithmetic and logic unit is waiting for data a minimum amount of time. (For example, it would not be wise to have an instruction that loaded the accumulator with a number that was stored <u>ahead</u> of the instruction; the accumulator would have to wait a whole memory cycle to get hold of that number.) The practice of laying out the instructions and data in the ultrasonic store in an optimal manner was called optimum programming.

Although vacuum tube flip-flops would have provided a faster-access storage medium, they were not yet economical. EDSAC had 30 mercury delay lines, each of which could hold thirty-two 17-bit numbers. There were also short mercury "tanks" that held just one number and were used as registers. The access time in these registers containing only one number circulating through a tank was shorter than the access time to a number circulating in main memory. For the main memory, the circulation time or memory cycle time was 1.1 ms. The other operation times were:

Addition/Subtraction	1.5 ms
Multiplication	4 ms

Division was a subroutine which had a variable operation time.

EDSAC had a single-address instruction format which necessitated the placing of an accumulator in the arithmetic and logic unit to accumulate the results of the one-address operations. EDSAC had two types of Test-and-Branch instructions; one which branched on the contents of a storage location being less than zero and the other which branched on the contents being greater than or equal to zero. It was admitted at the time that even though <u>two</u> tests were redundant, the extra one was included for programming convenience. It must have been around this point in time that the programming profession began.

Input and output were combined on a teleprinter unit which could both type and punch five-position paper tape. Input data could be punched onto paper tape which in turn was fed into EDSAC and output could be displayed via the typewriter part of the teleprinter.

6. University of Manchester Computers

EDSAC was merely the name given to the world's first operational computer developed at the University of Manchester. As time passed, EDSAC evolved into a computer system with refinements that expanded the state-of-the-art of computing.

The Williams Tube memory was developed at Manchester in order to increase the speed of memory access. Basically the tube was just a cathode ray tube (CRT) that could store an electrostatic pattern of bits on the face of the tube. Moreover, the bits could be fetched or changed by directing the cathode ray to the appropriate place on the tube. The tubes at Manchester held 1024 spots and could therefore represent 1024 bits of information; the access time was on the order of microseconds.

One of the uses of the Williams tube was what we now call indexing. A Williams tube, called the B-tube (presumably because the letters A and C were already used) was used to represent two registers. When the programmer wrote an instruction, he also referenced the contents of either one of these two registers. The contents of the specified register was added to the address field of the instruction. In practice, the contents of one of the registers was always zero so that when the programmer did not wish to modify his address field, he could reference the register containing the zero value. At the time, some people felt the B registers were of little scientific value and that they were included merely for programmer convenience. It seems the hardware design philosophy was

beginning to change--a problem that now deserved consideration was programming ease. Computers were still being built to solve specific problems, but they were starting to create problems of their own to be solved. The problem set had started to divide into "applications" problems and "systems" problems.

The Manchester computers added a 128 word drum--each word was 40 bits. The drum was slower than tube memory but it was cheaper in terms of cost per bit stored. Where the access time to tube memory was on the order of microseconds, access to the drum was measured in milliseconds. Therefore, the programmers at the University of Manchester were among the first to contend with the problems of memory hierarchy and cost-effectiveness in computer operations: if you have a larger, cheaper, and slower memory and a smaller, more expensive, and faster memory, both of which can be accessed by the arithmetic unit, you must consider the problem of making the most effective use of the total computer. If your criterion for effective use is to minimize the idle time of the arithmetic unit then you must keep it supplied with data as fast as you can. One method of achieving this is to feed the small, fast storage from the large, slow one, transferring data in large blocks. The arithmetic unit then fetches from the faster storage. Results from the arithmetic unit are stored to the faster memory, if possible, and eventually can be sent to the large, slow memory.

7. Electronic Discrete Variable Automatic Calculator (EDVAC)

EDVAC was the first stored program computer to be designed. In 1945 a report, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" by A. W. Burks, H. H. Goldstine, and J. Von Neumann, was prepared under contract to the ENIAC project. This report described the concept of the stored program computer, and made the recommendation that instructions and data be coded using a binary representation.

The report pointed out that although the ENIAC appeared to be a decimal machine, the decimal capability was built up from binary components grouped to respond as decimal components. It was recommended that numbers and instructions be represented inside the machine in terms of the existing binary components and that conversion to a decimal representation be performed in the Input/Output phase by means of a program. In other words, it was proposed to use software rather than hardware to take care of converting from the binary to decimal system and back. The report was distributed at a summer meeting at the University of Pennsylvania in 1946 and was a strong influence on the design of all future computers, in particular EDSAC and EDVAC.

As its primary storage, EDVAC used ultrasonic delay tanks similar to the mercury delay lines used by EDSAC. A tank was 58 cm long and it took 384 μs for a disturbance to travel that length, thus the memory cycle time was 384 μs. The clock rate was 1 Megacycle so that the tank could hold 384 pulses or bits of information. Each number was 44 bits long

followed by 4 "blank" pulses so that a tank stored 8 numbers. The total EDVAC memory was 128 tanks and could store 1024 numbers.

A wire recorder acted as a secondary store with a capacity of 20,000 numbers. As with EDSAC, a memory hierarchy existed with a smaller, faster tank memory to be traded off against a larger, slower wire memory. Numbers were transferred from the wire memory to tank memory in blocks of 50 to 100 so as not to slow the arithmetic and logic unit.

EDVAC used a four-address instruction format. The address field of an instruction, instead of denoting a single address, denoted four locations: the first two locations signified the addresses of the two operands to be used in a binary operation (a binary operation is an operation such as addition, subtraction that involves the use of <u>two</u> operands), the third address indicated where the result was to be stored and the fourth address pointed to the location where the next instruction to be executed was stored. The fourth address has proved to be superfluous if the computer has a test and branch capability and otherwise executes its instructions in sequence. (Assuming that the instructions are stored in a memory where the time to fetch an instruction is not dependent on where in the memory it is stored--this type of memory is sometimes called "random-access".) EDVAC pointed the way to a three-address scheme whereby the instructions were executed in sequence and the three addresses were used in the same way as the first three addresses described above.

A three-address scheme can be very powerful if the programs involve many three-step operations such as A = B + C. However, the trend

was to grow away from a three-address scheme which was more useful in scientific problems than commercial ones (as well as being more costly than a one-address scheme) and eventually settled into the familiar one-address scheme we have on most current generation machines.

Here is another example of the applications problems creating systems problems concerning the shaping of the design of the machine. It would not be useful to design a two-address machine if there were no problems that could be solved with that kind of instruction format. The repertoire of instructions has also evolved under the demands of the problems to be solved. Character handling instructions would not have been implemented so soon and so fully if all problems had been scientific.

The average operation times for EDVAC were:

Addition/Subtraction	864 μs
Multiplication	2.9 ms
Division	2.9 ms

EDVAC appears to have the unhappy distinction of being the first computer to experience large time delays in fabrication even though the proposed design was well within the technical resources available at that time. EDVAC design was started in 1945, but was not considered to be a working machine until 1952. M. V. Wilkes attributes the problems to the much faster clock rate used in EDVAC which necessitated higher quality circuitry that could handle pulses of shorter duration without degradation.

It appears that there is a principle of natural selection that applies to the evolution of computers. Computers are designed to respond to the needs of the environment. If the environment changes too rapidly, some classes of computers may be subject to the fate of the dinosaur. More important, the environment is not a closed system outside of the computer; the computer, as it responds to its environment becomes a part of the environment, and creates new problems to be solved. Machines are then created to solve these problems. We create tools to solve problems that our tools have created.

C. Unconventional Digital Computer Organizations

1. Summary

After EDVAC, in the early 1950's, the deluge began. Hundreds, then thousands of computers were manufactured; still, they were generally organized on Von Neumann's concepts. The conventional or Von Neumann organization is shown and described in Figure I-3. 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 typewriters to magnetic tape units, disks, drum and remote terminals. But the four basic components of a conventional computer (Control Unit, Arithmetic and Logic Unit, 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 take effect in

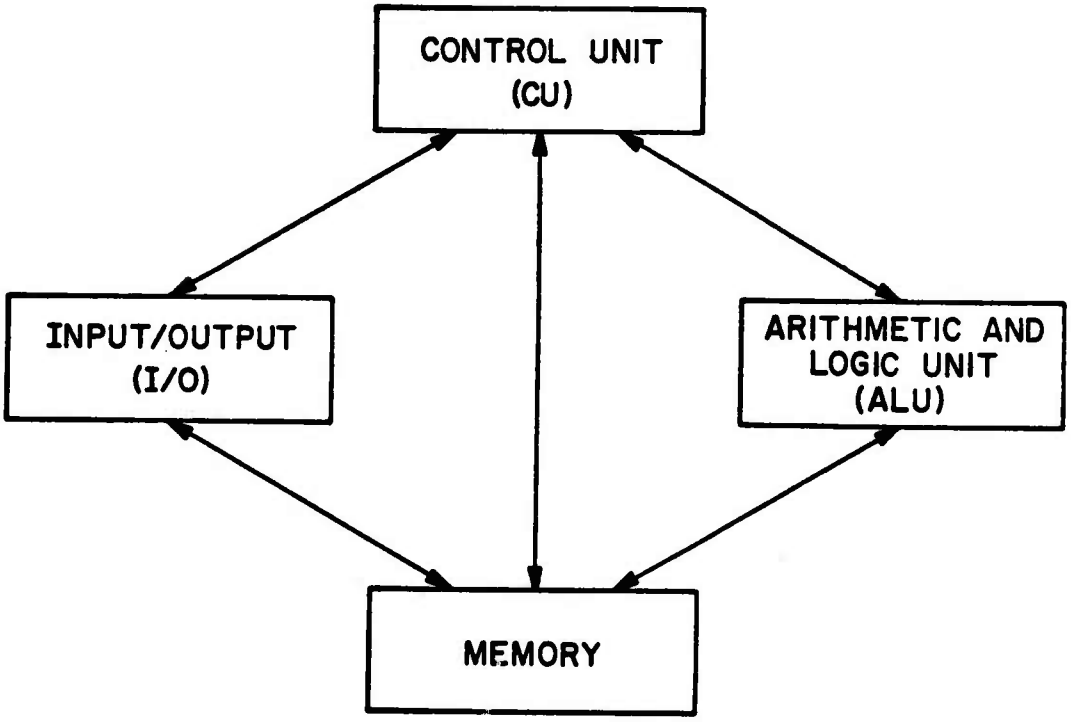


Figure I-3. Functional Relations within a Conventional Computer

The Control Unit (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 or Central Processing Unit.

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 figure I-3.)

Electronic circuits appear to be 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 an increase in speed 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 de-coupled. Certainly the Control Unit could at least be fetching the next instruction while the Arithmetic and Logic Unit was carrying out 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 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 re-centralizing one of the three major

components which had been previously replicated in the general Multiprocessor--the Memory, the Arithmetic and Logic Unit or the Control Unit. Centralizing the Control Unit 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 Control Unit circuitry. Replication of this component is particularly expensive and therefore centralizing the Control Unit saves more money than can be saved by centralizing either of the other two components.

2) There is a large class of both scientific and business problems that can be solved by a computer with one Control Unit (one instruction stream) and many Arithmetic and Logic Units. The same algorithm is performed repetitively on many sets of different data; the data is 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.

ILLIAC IV also utilizes the Buffer and Pipeline mechanism to overlap the execution of instructions. This allows a further increase in operational speed as both the replication and overlap design philosophies are applied simultaneously.

2. Overlap Mechanisms

a. Buffer

A buffer is a mechanism which allows a process to be broken down into subprocesses so that the execution of the subprocesses can be overlapped.

Let us use an analogy to demonstrate what a buffer is and why we would like to use one:

Suppose you are mowing your front lawn and you have a bag attached to your mower to collect the grass clippings. Each time this bag fills up, you must stop the mower, detach the bag, and walk around to the back of your house where the trash barrels are. You must then empty the bag of accumulated clippings into the trash barrel, walk back to your mower, attach the bag, and continue mowing.

After some time you come to the realization that you are spending a lot of your time detaching the bag, walking to the trash barrels, emptying the bag, walking back and re-attaching the bag. You remember that you also own a large wheelbarrow that could hold many bag-loads of grass clippings. You now recognize the option of placing the wheelbarrow on the front lawn, and when the grass bag becomes full, you could walk over to the nearby wheelbarrow and empty the bag into the wheelbarrow. When the wheelbarrow became full, then you would have to push it to the trash barrels behind the house, empty the wheelbarrow, and push the wheelbarrow back to the front lawn.

Very naturally the question arises: How many bag-loads must the wheelbarrow be able to hold to justify its use? Fortunately, this problem is very easily solved. Let us look at the times associated with each method.

Method 1: No wheelbarrow used

T_1 = Time to detach bag from mower

T_2 = Time to walk from mower to trash barrel

T_3 = Time to empty bag into trash barrel

$T_4 = T_2$ = Time to walk back from trash barrel to mower

$T_5 = T_1$ = Time to attach bag to mower

Method 2: Wheelbarrow is used as a Buffer

$T_6 = T_1$ = Time to detach bag from mower

T_7 = Time to walk from mower to wheelbarrow

$T_8 = T_3$ = Time to empty bag into wheelbarrow

$T_9 = T_7$ = Time to walk from wheelbarrow to mower

$T_{10} = T_1$ = Time to attach bag to mower

T_{11} = Time to push wheelbarrow to trash barrel

T_{12} = Time to empty wheelbarrow into trash barrel

$T_{13} = T_{11}$ = Time to push wheelbarrow from trash barrel to front lawn

(Even though the wheelbarrow or bag is lighter on the walk back from the trash barrel, we are assuming it will take the same time as the walk to the trash barrel since grass clippings are very, very light. We also equate the time to attach a grass catcher bag and the time to detach it--

based on actual experience.) Finally, in order to relate all the times (T_1 through T_{13}) to each other we assume that the wheelbarrow holds N bag-loads. Therefore, repeating Method 1 N times is equivalent (in terms of area of lawn mowed) to performing Method 2 once.

The question then becomes: When is

Total time for Method 1 $\geq$ Total time for Method 2?

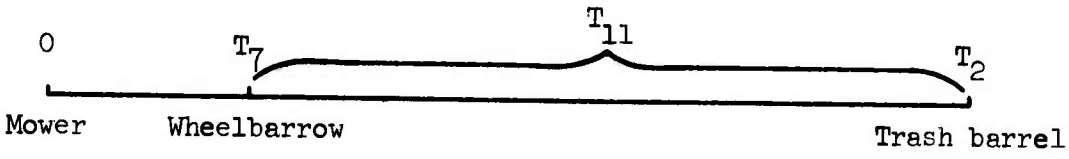
or for what value of N is

$$N(T_1 + T_2 + T_3 + T_2 + T_1) \geq N(T_1 + T_7 + T_3 + T_7 + T_1) + T_{11} + T_{12} + T_{11}$$

which reduces to

$$N \geq \frac{2T_{11} + T_{12}}{2(T_2 - T_7)}$$

We can see from the diagram below that $T_2 > T_7$ and assuming $T_{11} = T_2 - T_7$,



we therefore arrive at

$$N \geq \frac{2T_{11} + T_{12}}{2T_{11}}$$

so that in order for Method 2 to be feasible, the wheelbarrow must hold N bag-loads where

$$N \geq 1 + \frac{T_{12}}{2T_{11}}$$

We now see that the size of our buffer wheelbarrow depends only on T_{11} and T_{12} or viewed somewhat differently, that the larger N is (the bigger the wheelbarrow we have) the less we have to worry about the effect of T_{11} and T_{12}.

If we now enlist another person to help us by emptying the wheelbarrow when it gets full and bringing it back in time to receive the next bag-load, this will reduce the total time of Method 2 by making $T_{11} = T_{12} = 0$ since these subprocesses are being performed <u>simultaneously</u> with the other subprocess times. Now the question of what size N justifies Method 2 over Method 1 becomes: For what N is:

$$N(T_1 + T_2 + T_3 + T_2 + T_1) \geq N(T_1 + T_7 + T_3 + T_2 + T_1)$$

Using the same reasoning as before we see that

$$2N(T_2 - T_7) \geq 0$$

This relation holds true for all N since $T_2 > T_7$. Therefore, this scheme of having a helper who runs the wheelbarrow is a better way to mow a lawn than by yourself. One may have guessed that fact intuitively; however, it is not always clear how a process can be broken down into autonomously performed subprocesses as it is with this particular analogy.

This analogy, although simple-minded, does illustrate what a buffer is and how it works: If a process consists of a series of subprocesses and this process takes "too long" from beginning to end, we can speed up the process time by dividing the total process into at least two subprocesses each of which control themselves autonomously. Between the two subprocesses we place a buffer so that the output from subprocess 1 goes into the buffer and the input to subprocess 2 comes from the buffer. Since the two subprocesses operate autonomously they speed up total process time by overlapping (in time) their performance. The buffer acts as a decoupler of control between subprocess 1 and subprocess 2 and a place to save things which must be passed between the subprocesses.

It may usually turn out in practice that one process occurs at one rate of speed while another occurs at a greatly different rate of speed. In this case, the processes already existed as separate and distinct, and the placing of a buffer between them is necessary only to insure that the high speed process is not held up by the low speed one. The placing of a buffer between the processes again decreases the total process time by overlapping operations. (See Figure I-4.)

Suppose, for example that subprocesses P_1, P_2 and P_3 occur at a very fast rate and that P_4 and P_5 occur slowly. A buffer could be placed between them as shown in the lower part of Figure I-4 and the $P_1P_2P_3$ process would not be held up waiting on P_4 and P_5.

Buffers may have another effect on autonomous processes. They not only speed up the rate at which information flows through the

two-process system, they may smooth out the rate of information flow. Without the buffer, one process must wait on another and the outputs of the first process appear and then must wait a variable time until accepted by the second process. This results in a "jerky" flow of information through the system. The buffer acts to accept outputs from the first process as soon as they are generated and will save these outputs until the second process is ready to accept them.

Summing up: a buffer decouples control between a previously sequential set of processes, transforming them into at least two parallel or simultaneous processes; and provides a place to store information which must be passed between the processes.

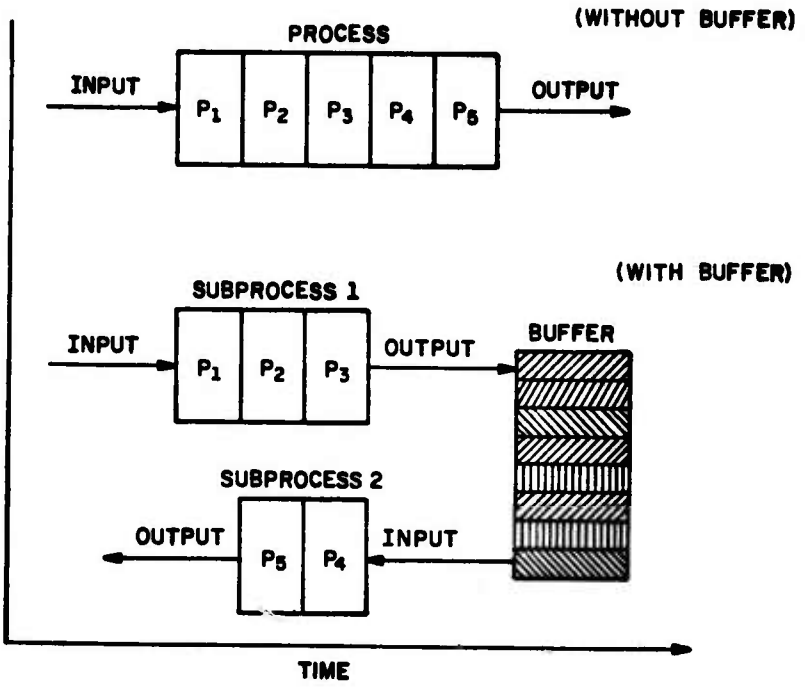


Figure I-4. Process Execution with and without Buffer

b. Pipeline

i. Summary

A sequential process can be viewed as a black box that accepts inputs and produces outputs with the added stipulation that the black box cannot accept a new input until the output has been generated for the previous input. In other words, the black box is tied up all of the time in processing just one input.

As an example let us consider a black box (an Adder) that adds two numbers together. Say there are two inputs (the numbers to be added) and one output (the result). If it takes M seconds for the Adder to perform the operation it will take N * M seconds to add N pairs of numbers. However, if the Adder would accept additional operands to be added while the ones ahead were still in the box then the total time to add N pairs of numbers would certainly decrease. We can do this if the add operation can be broken down into independent stages; as soon as an operand passed through the first stage, the next pair of operands could be accepted by the Adder. This method of dividing the adder into stages and letting the stages run independently is called a "pipeline." The total time to process N operands is speeded up because, once all of the stages in the pipeline are full, results appear out of the end of the pipeline in time increments equal to the processing time of the slowest stage.

ii. Background

The pipeline mechanism can be applied to a process that is able to be broken down into two or more stages that can operate independently; the only dependence between stages is that the output of a previous stage becomes the input to a succeeding stage. For example, suppose we have a process that upon closer inspection can be viewed as being made up of three subprocesses. If each subprocess time is P_1, P_2 and P_3 then it takes $P = P_1 + P_2 + P_3$ units of time to transform an input to an output of the process, and consequently if we have N inputs to process then it will take $N(P_1 + P_2 + P_3)$ units of time to complete the job. Figure I-5 shows how two inputs I_1 and I_2 proceed through our example three-stage sequential process. The outputs O_1 and O_2 are both ready after 2P units of time.

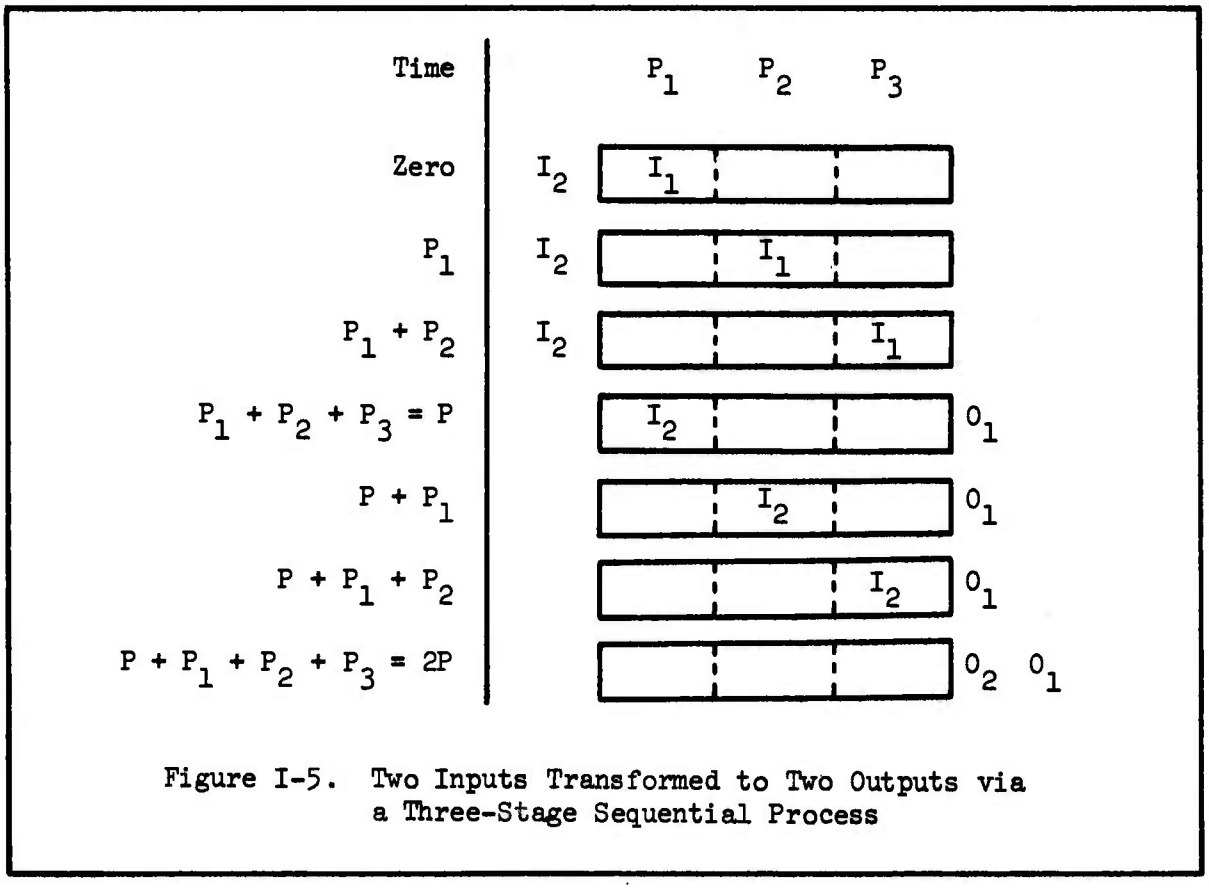


Figure I-5. Two Inputs Transformed to Two Outputs via a Three-Stage Sequential Process

Now let us apply the pipeline mechanism to our example. First we decouple each subprocess by placing a one item "holding buffer" after each subprocess; when a subprocess or stage has completed its job, it places its output into its holding buffer. When all stages are finished they simultaneously pass their outputs to the input part of the next stage. Although this slows the operation of the pipeline down to the rate of the slowest stage, inputs do not have to wait outside the process until the previous input is completely finished--inputs can enter into the process as soon as the first stage has passed its results to the second stage. Since the stages have been decoupled, they can be processing different items or operands simultaneously. Each item moves through the stages of the pipeline or pipe in a semi-finished state of completion (not holding up a following operand) until it appears at the end of the pipe completely processed. See Figure I-6 which shows how two inputs produce two outputs in a three-stage pipeline.

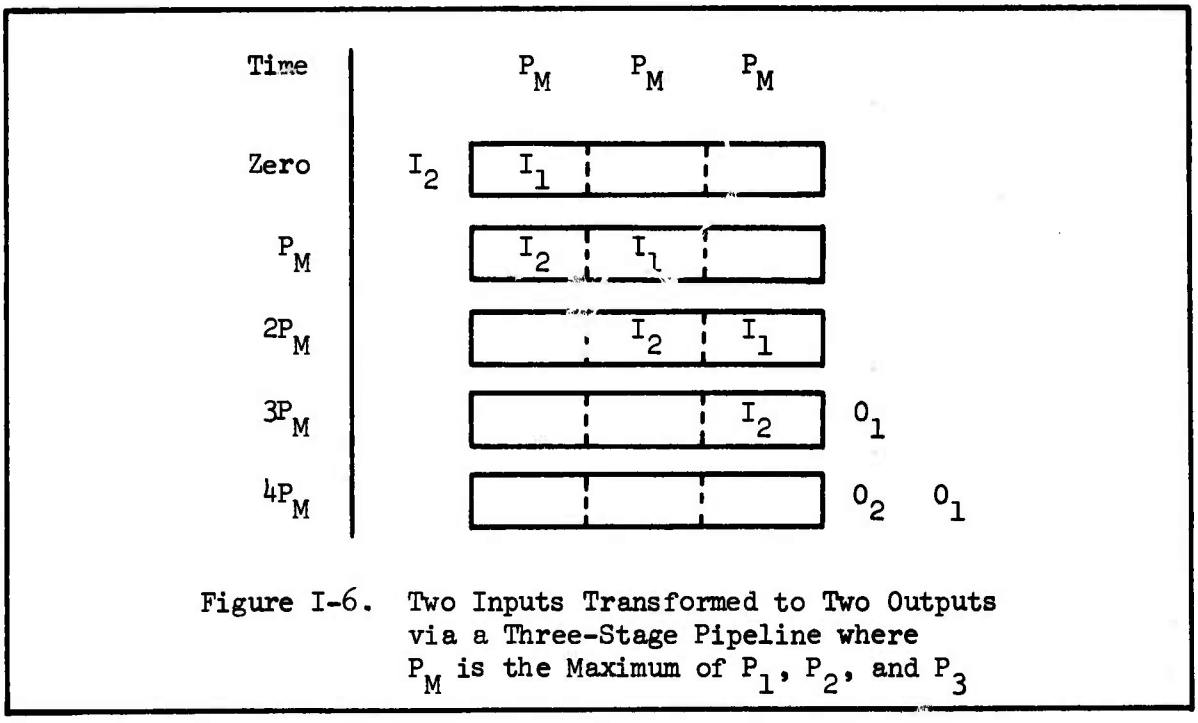


Figure I-6. Two Inputs Transformed to Two Outputs via a Three-Stage Pipeline where P_M is the Maximum of P_1, P_2, and P_3

It takes $P_M + P_M + P_M$ units of time to make the initial filling of the pipeline--after that a finished item appears at the end of the pipe every P_M units of time where P_M is the maximum of P_1, P_2 and P_3. Thus the time to process N items via the pipeline mechanism for our 3 stage example is $3P_M + (N-1)\ P_M$ units of time.

We can now ask the question (as we did with the Buffer): for what value of N is the sequential process time greater than the pipelined process time, i.e., for what value of N is:

$$N(P_1 + P_2 + P_3) > 3P_M + (N-1)\ P_M$$

or $$N(P_1 + P_2 + P_3) > (N+2)\ P_M$$

Let us say that $P_M = P_2$, then

$$N(P_1 + P_3) + NP_M > NP_M + 2P_M$$

or $$N > \frac{2P_M}{P_1+P_3} = \frac{2P_2}{P_1+P_3}$$

That is, for the pipelined process to be faster than the sequential one, N must be such that the relation $N > 2P_2/P_1+P_3$ is true. For the example we are considering in Figure I-6, N = 2 so that

$$2 > \frac{2P_2}{P_1+P_3} \text{ or } \frac{P_2}{P_1+P_3} < 1$$

must hold for the pipeline to be quicker;

thus if $P_1 = 3$, $P_2 = 4$ and $P_3 = 2$ so that

$$\frac{P_2}{P_1+P_3} < 1$$

then for this choice of subprocess times the pipeline would be faster.

The larger N is, the greater the chance that the pipeline is faster than the sequential process for a given set of subprocess times. Let us devise a more general formula for S stages and N items to be pipeline processed:

The sequential process time to process N items through S stages is

$$N \sum_{i=1}^{S} P_i$$

The pipelined time is

$$SP_M + (N-1)\, P_M$$

So we ask: for what N does the following relation hold true?

$$N \sum_{i=1}^{S} P_i > SP_M + (N-1)\, P_M$$

or $$N \sum_{i=1}^{S} P_i > NP_M + (S-1)\, P_M$$

Now, say $P_M = P_j$ where $1 \leq j \leq S$, then

$$N \sum_{\substack{i=1 \\ i \neq j}}^{S} P_i + NP_M > NP_M + (S-1)\, P_j$$

or

$$N \sum_{\substack{i=1 \\ i \neq j}}^{S} P_i > (S-1)\, P_j$$

or (1)

$$N > \frac{(S-1)\, P_j}{\sum_{\substack{i=1 \\ i \neq j}}^{S} P_i}$$

Since the right hand side of the relation (1) is always greater than 1, we can say that for the pipelined process to be faster than the sequential one, the number of items, N, must be larger than one--again, we might have guessed this intuitively.

Additionally, the gain of the pipeline approach over the sequential one is a function of the number of stages, S, and the distribution of the subprocess times, P_i. Let us consider two possible distributions for P_i: The best case (the one in which the pipeline outperforms the sequential method by the highest time ratio) is when all of the subprocess times are equal:

$$P_i = K \qquad i = 1, 2, \ldots, S \qquad \text{then } P_M = K$$

and the ratio of sequential time to pipelined time becomes:

$$R = \frac{N \sum_{i=1}^{S} P_i}{SP_M + (N-1)\,P_M} = \frac{NSK}{SK + (N-1)\,K} = \frac{NS}{S + N - 1}$$

and $\lim_{N\to\infty} R = S$; so we see for this case the pipeline can be up to S times as fast as the sequential process (where S is the number of stages in the pipeline) if we can keep the pipeline full all of the time ($N \to \infty$). The relationship (1) on page I-41 for this case becomes:

$$N > \frac{(S-1)\,K}{(S-1)\,K} = 1 \qquad \text{or} \quad N > 1$$

which means all we need is for N to be greater than one (two items) for the pipeline to be more effective; and the larger N is the better the pipeline looks.

Now let us compare the pipelined and sequential times when the P_i have a linear distribution, say

$$P_i = i \qquad i = 1, 2, \ldots, S \qquad \text{then } P_M = S$$

and the ratio of sequential time to pipelined time becomes:

$$R = \frac{N \sum_{i=1}^{S} P_i}{SP_M + (N-1)\,P_M} = \frac{N(S/2)(S+1)}{S^2 + (N-1)\,S} = \frac{N(S+1)}{2(S+N-1)}$$

and $\lim_{N\to\infty} R = (S+1)/2$. Comparing this ratio, R, to the constant distribution ($P_i = K$) where the ratio is S:

When is $S > \frac{S + 1}{2}$?

The answer is whenever $S > 1$ or whenever there is more than one stage. So we see that the pipeline outperforms a sequential process by a factor of S when the subprocess times are all the same, and by a factor of (S+1)/2 (not quite as good) when the subprocess times are linearly distributed. Both of these factors are based on the assumption that the pipeline is kept full all of the time.

The relationship (1) on page I-41 for this second case becomes

$$N > \frac{(S-1)\ S}{((S-1)/2)\ S} = 2 \qquad \text{or } N > 2$$

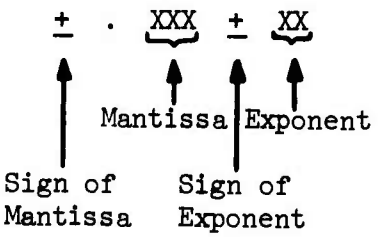
which means all we need is for N to be greater than two (three items) for the pipeline to be more effective than the sequential process. Note that this is a more stringent requirement than for the constant distribution ($P_i = K$) case described first.

There is, however, a finite limit on the number of autonomous subprocesses a process can be broken down into, so that efficiency does reach a maximum value. It should also be clear that for the pipeline mechanism to function at its best efficiency, it should be kept full as much as possible. If the pipe ever drains (runs out of items to be processed) the initial filling up time is very costly since each stage operates at the rate of the slowest stage in the pipe.

iii. A Pipeline Adder

We shall now apply the pipeline mechanism to the adder section of the Arithmetic and Logic Unit of a computer. To illustrate the time speed-up, let us assume that we must add seven pairs of floating point numbers with rounding and normalization. First, let us briefly review the process by which two floating point numbers are added:

Using a decimal notation we represent a number in the floating point format as follows:



We have allowed 3 significant digits in the Mantissa and two for the Exponent. Thus -.123 + 01 is the same as $-.123 \times 10^{1}$ in scientific notation or -1.23. Also +.014 - 02 is the same as .00014. We say that a number in floating point format is normalized when the Mantissa is greater than or equal to .1 but less than 1.

$$.1 \leq \text{Mantissa} < 1.$$

Thus +.014 - 02 is not normalized but

+.140 - 03 is normalized.

In order to add two numbers in floating point format, we must first equalize their exponents so that their mantissas can be added.

However, when equalizing the exponents we always take the number with the smaller exponent and "promote" the smaller exponent up to the larger one and adjust the mantissa of this number by right shifting by the difference of the exponents. We could not perform a left shift or a significant digit or a normalized number would appear to the left of the decimal point.

After addition of the mantissas, we normalize the result if necessary and finally we round the result so that it can be expressed within 3 significant digits.

In order to perform the four operations

1. Adjust Exponents
2. Add Mantissas
3. Normalize (if necessary)
4. Round

we must have an accumulator in our adder that can hold more information than the format we have specified for our floating point numbers. For our 3 significant digit case let us use an accumulator capable of holding numbers of the form

$$\pm\ X\ .\ XXXXXX\ \pm\ XX$$

This accumulator has an extra position to the left of the decimal point to temporarily store a digit which might overflow as a result of an addition, and it has 6 significant digits to insure accuracy when rounding takes place.

Consider two examples of what steps can occur using actual numbers as operands. We assume that all operands enter into the floating point addition process in normalized form and that the number with the smaller exponent has been placed in the extra length accumulator.

Example 1 Add 123 to 45.6; that is, perform the operation:

123. + 45.6

In normalized form the numbers are:

(+.123 + 03) + (+.456 + 02)

1. Adjust Exponents: .123 + 03

   +0.045600 + 03 ← This number is in the accumulator

2. Add Mantissas: +0.168600 + 03 ← Result in accumulator also

3. No Normalization Necessary

4. Round Result: +.169 + 03 ← Number is now back in 3 significant digit form.

Example 2 Add 9.99 to .0147; that is, perform the operation:

9.99 + .0147

In normalized form the numbers are:

(+.999 + 01) + (+.147 - 01)

1. Adjust Exponents: +.999 + 01

   +0.001470 + 01

2. Add Mantissas: +1.000470 + 01

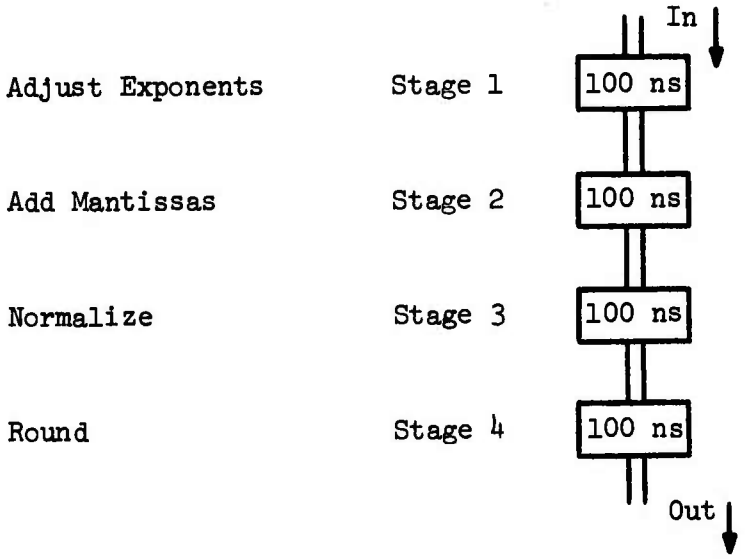
3. Normalize: +0.100047 + 02

4. Round: +.100 + 02

Now suppose that the stages of the four step addition process took 70, 100, 60 and 50 nanoseconds (ns) respectively, then the total time to add our seven pairs of numbers in a sequential manner would be

$$7 \times (70 + 100 + 60 + 50) = 1960 \text{ ns}$$

Since each stage can perform its operation autonomously on different operand-pairs, let us "pipeline" the four-stage addition process. Since the slowest stage operates at 100 ns we have



At the end of 400 ns the first result appears at the end of the pipe; after that results come out every 100 ns. (Note that in the sequential process the first result appears after only 280 ns but they continue to be created at that rate.) See Figure I-7 for a snapshot of the pipelined adder every 100 ns. We see from the figure that the total time to add 7 numbers using our four-stage pipeline adder takes only 1000 ns as compared with the 1960 ns sequential addition.

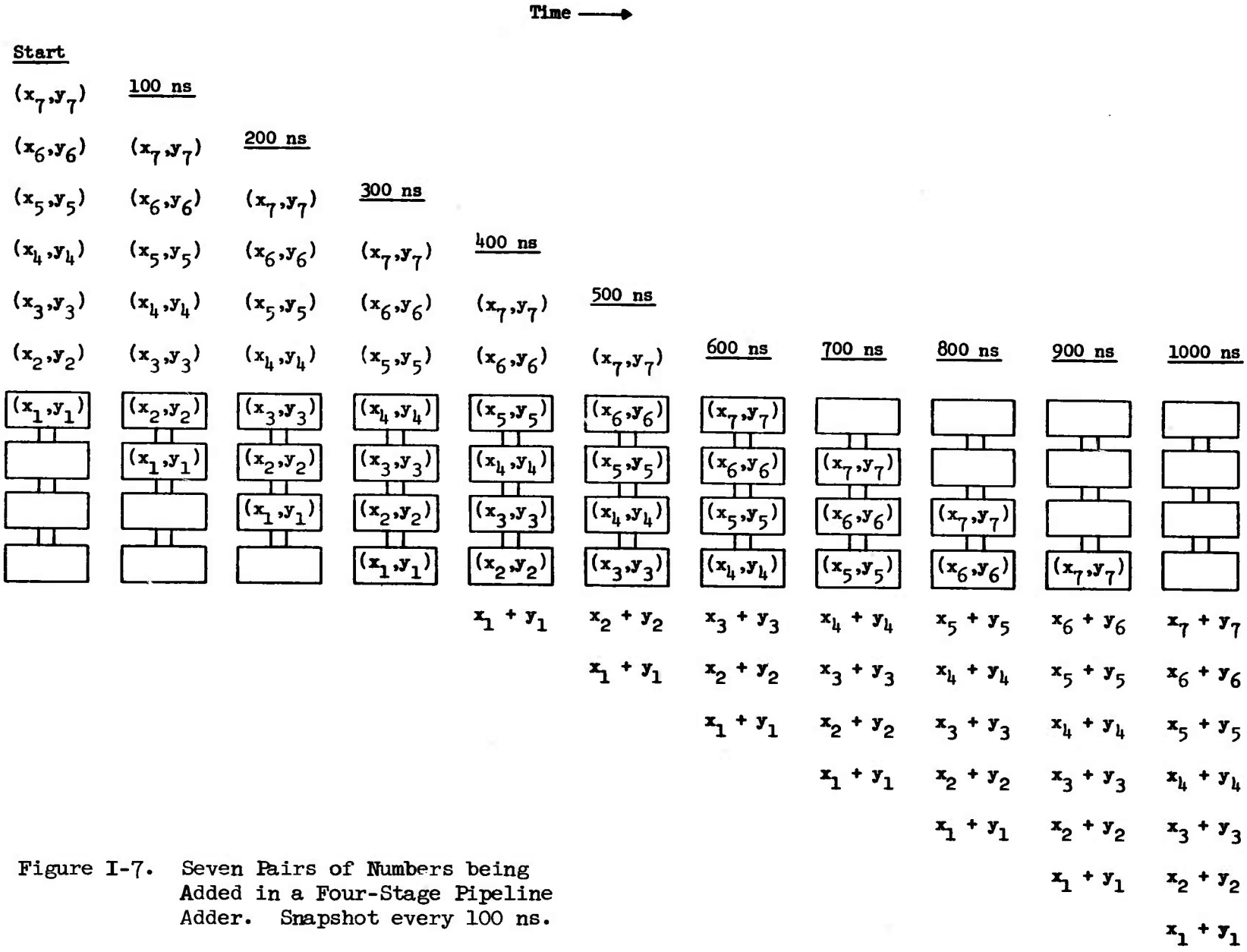


Figure I-7. Seven Pairs of Numbers being Added in a Four-Stage Pipeline Adder. Snapshot every 100 ns.

It is, of course, of primary importance to keep the pipe filled with operands. If the pipe is used sporadically instead of continuously, its purpose is defeated. Other problems may arise if more than one pipeline unit exists--for example, an Arithmetic and Logic Unit may have a Pipeline Adder and a Pipeline Multiplier. If one pipeline is performing an operation that needs the result of another operation which is in another pipe, then efficiency drops while one unit waits on another.

iv. A Pipeline Instruction Execution Unit

Arithmetic and Logic Unit components are not the only sections of a computer that can be pipelined to increase execution speed. It is also possible (although a trickier proposition) to pipeline the instruction execution section of the Control Unit. (This approach was taken by the IBM STRETCH computer.)

The process of interpreting and executing an instruction can be decoupled into several autonomous stages and therefore instructions can be executed through a pipeline--each instruction in the pipe being in a semi-finished state of execution. The tricky part of this proposition comes in when one instruction in the pipe needs the results from the total execution of another instruction in the pipe. At this stage, the pipelining process must stop and all instructions ahead of the instruction must be processed through the pipe so that the instruction which needed the complete results can be given them. Another problem would be a Test-and-Branch instruction proceeding through the pipe. From where do you fetch the instructions following the Test-and-Branch? Also, instructions

which modify fields of other instructions (such as the address field) could not both be allowed in the pipe at the same time. When certain possibilities such as the ones described above do occur, the pipe must be allowed to drain or is "flushed out" and the benefits of the pipeline are temporarily wasted.

In the pipelined Instruction Look Ahead Unit of STRETCH, instructions were fetched while their predecessors were being executed and operands were made ready, if necessary. Each instruction was in a stage of partial completion in the pipeline. The problem of how to handle a Test-and-Branch or Conditional Branch instruction was solved very straightforwardly: the assumption was made that the test would always fail so that succeeding instructions were fetched from the location contiguous to the branch instruction. About half the time this guess would be right. Once programmers were aware of this type of bias they could then write their programs to take advantage of it.

Summing up: A Pipeline is a mechanism by which a previously sequential process is broken down into stages, each of which can operate independently of the other. When the slowest stage is finished, the output from Stage i is passed on to become the input to Stage i + 1 for all stages simultaneously. Once the pipe is full, output appears at the end of the pipe at a time increment equal to the operation time of the slowest stage.

3. Replication--The Multiprocessor

The general Multiprocessor is the embodiment of the replication design philosophy; three of the four components of a conventional computer are replicated many (N) times resulting in a system that can be up to N times as powerful. See Figure I-8. We can think of a general multiprocessor as N conventional computers in one system, all sharing the I/O resources of that system. There may be some information flow from Control Unit to Control Unit but the main idea is that each Control Unit can independently and simultaneously execute the program in its memory. Since the multiprocessor can be executing N distinct streams of instructions simultaneously, it can, under optimum conditions, effect a time speedup by a factor of N.

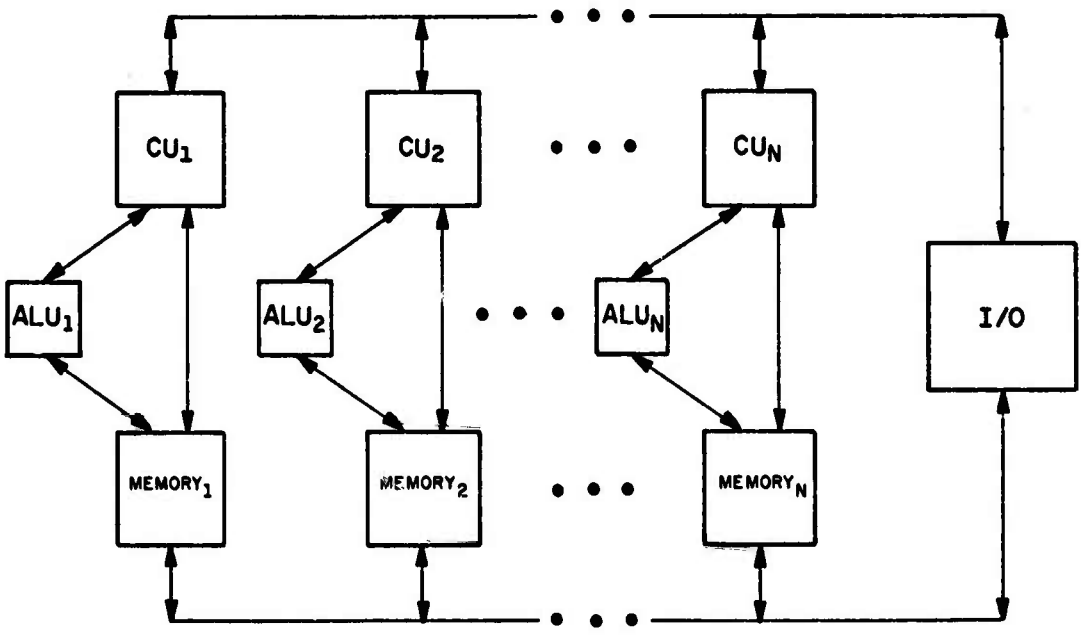


Figure I-8. Functional Relations within a General Multiprocessor

It is, of course, very expensive to build a true multiprocessor as outlined by Figure I-8. There are N times as many Control Units, N times as many ALUs and N times as many Memories as there are in a conventional computer. In order to keep the cost at a minimum, the following question is asked: Which of the functional components: Memory, ALU, or Control Unit, could be centralized with little or no loss to the power of the multiprocessing system?

a. Centralize Memory

Memory could be lumped into one large memory of N times as many words instead of N separate memories but little savings in cost would result--you essentially still have to pay for the same number of bits of storage. (See Figure I-9.) The most severe problem that comes from

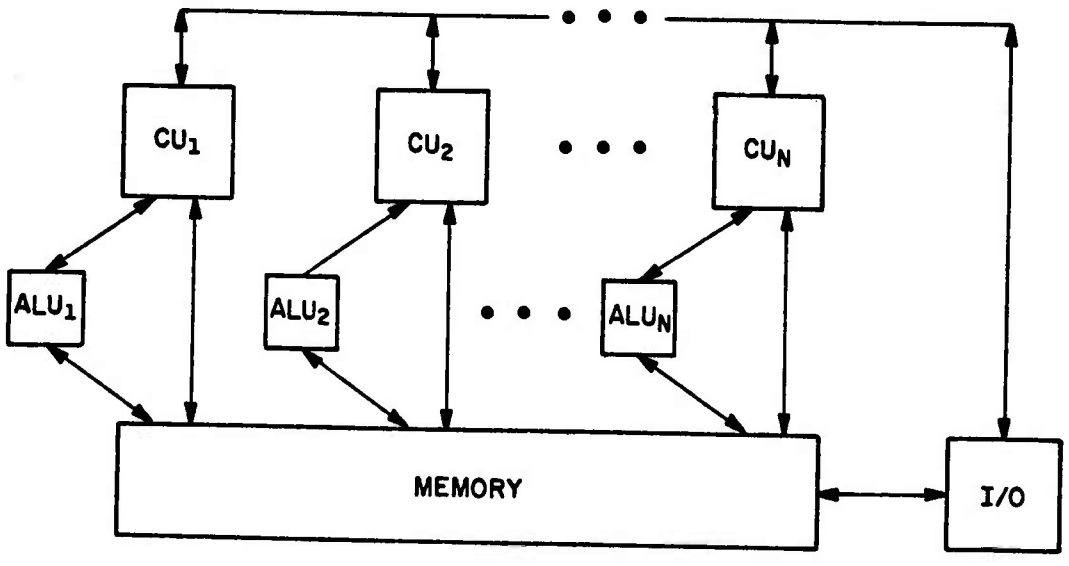


Figure I-9. Multiprocessor with Common (Lumped) Memory

sharing a common memory is the potential bottleneck that exists when more than one Control Unit wants to store/fetch data or instructions at the same memory location. Additional hardware lines (at additional cost) can be added to take care of the fetching problem but devising the software to decide which unit will store first is relatively complicated. Keeping one Control Unit out of the program instruction area reserved for another Control Unit requires that appropriate software or hardware be produced to maintain program integrity. Even if the manufacturer provides the hardware and software, it costs money and that cost is usually passed on to the customer. Sharing a common memory might end up costing more money than distributing the memories among the CUs and ALUs.

b. Centralize the Arithmetic and Logic Unit (ALU)

Another approach is to centralize the ALU into an extremely fast, high-quality ALU that could service all N Control Units. This design is called the Intrinsic Multiprocessor (see Figure I-10).

The ALU section of the Intrinsic Multiprocessor is comprised of many specialized and powerful processing units--some of which may be replicated (such as the Adder). (These units could be pipelined for a further increase in speed.)

The Control Units (CUs) can each be executing independent streams of instructions. When an instruction needs to use one of the processing units in the ALU, a request is placed in the Selector. If the desired unit is free, the operation requested is performed. If all of the units which

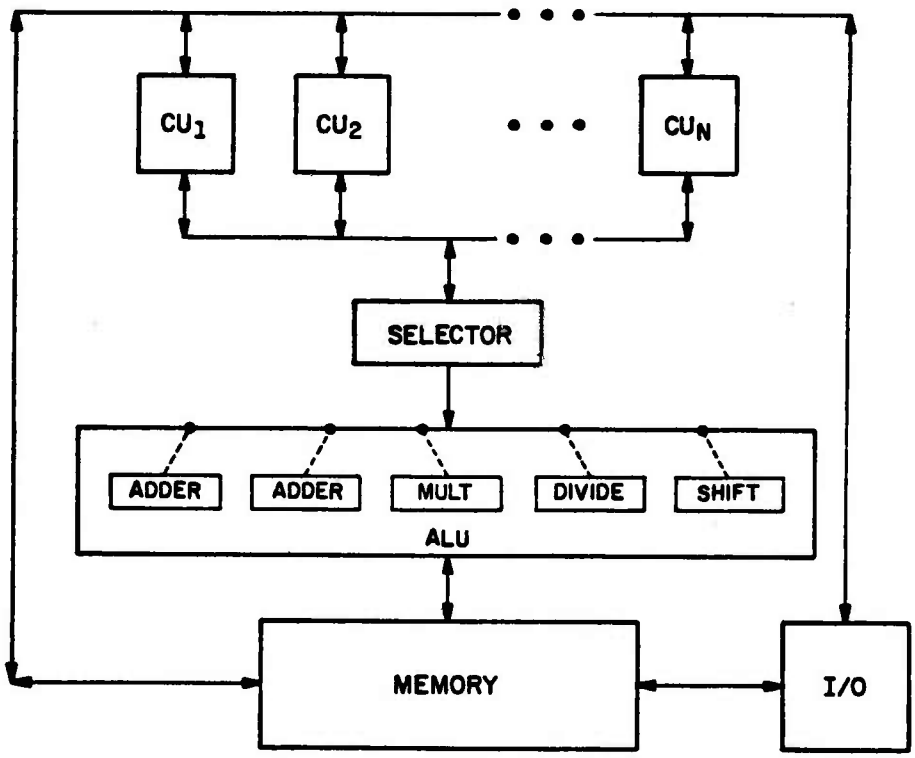


Figure I-10. Functional Block Diagram of Intrinsic Multiprocessor

could have performed the operation are currently busy, the request is placed in a queue to be serviced when the requested unit(s) become free. If the latter is the case, the CU which requested a unit that was busy will be temporarily halted in its execution of instructions by the Selector.

Note that memory is also centralized in this version of the Intrinsic Multiprocessor so that a common memory is shared by all CUs for instruction and data storage. Since memory is centralized it is necessary that this type of multiprocessor have a specialized instruction

repertoire. Results could not be stored temporarily in a register within the ALU, since it might hold up the results requested by another CU's operation. A solution would be to design the instructions to be of the three-address form: the two operands and the location in memory indicating where to store the result.

The design is effective if all of the specialized units in the ALU can be kept busy for a high percentage of the time. This means that the instructions coming to the ALU from the CUs be mixed in roughly the same proportion as the processing units present in the ALU. For further efficiency, the same instruction type should not appear at the same time in all CUs. If all conditions are right, a speedup is gained since the processing function (in the ALU) has been decoupled from the control function and both of these operations can proceed simultaneously--the ALU is not waiting on the CU to fetch and decode instructions. Rather than have many ALUs not being 100% utilized (as is usually the case in the general multiprocessor) the one Super-ALU of the Intrinsic Multiprocessor shares its resources among the many CUs.

c. Centralize the Control Unit (CU)

When the Control Unit is centralized (the design option taken by ILLIAC IV) the array of ALUs is called an Array or Vector Processor. "Array" is perhaps not the best choice of words because it can bring to mind a two-dimensional picture. In all further discussions it is very important for the reader to understand that the term "Array" refers to a

one-dimensional array--a row, a column, or still better, a vector. Now, what does it mean when we say a computer has an Array Processor or a Vector Processor? Before we answer this question, let us recall some history:

In the early days of computing (late forties and early fifties) data was processed by the CPU in a serial mode. Pulses representing the bits which in turn represented numbers went into the CPU "one-at-a-time" and were processed (added, subtracted, etc.) sequentially. The process could not be completed until after the last pulse had entered the CPU.

In order to speed up the operation of the CPU, its design was changed to accept data in a parallel mode or "all-at-once." Thus, if a word was N bits long, the parallel CPU could operate N times as fast as a serial CPU. See Figure I-11.

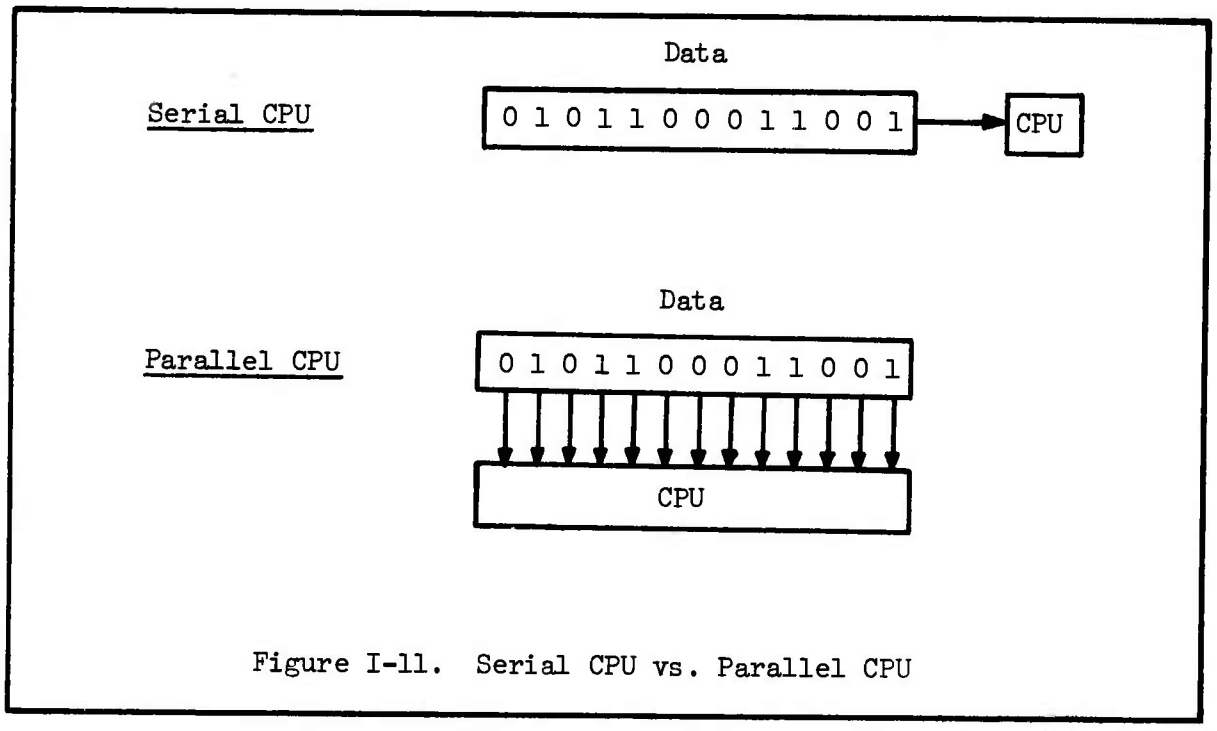


Figure I-11. Serial CPU vs. Parallel CPU

If we now extend this concept from dealing with the N bits in a word all-at-one-time to dealing with N words in a vector all-at-one-time we have the gist of a vector processor. Although a conventional computer can operate on the many bits within a word in parallel, the contents of the word is just one single number, or a scalar. If we devise a computer with an Arithmetic and Logic Unit that can deal with N words simultaneously, then we can view each word as a component of a vector and say that the machine has a Vector or Array Processor. (See Figure I-12.) Each ALU within the ALU array deals with one component of the vector.

Since an Array Processor performs its operations (+, -, x, ÷, AND, NOT, OR, etc.) on operands that are vectors, not scalar numbers, when you execute the instruction "Add A to B and store the result in C" what you

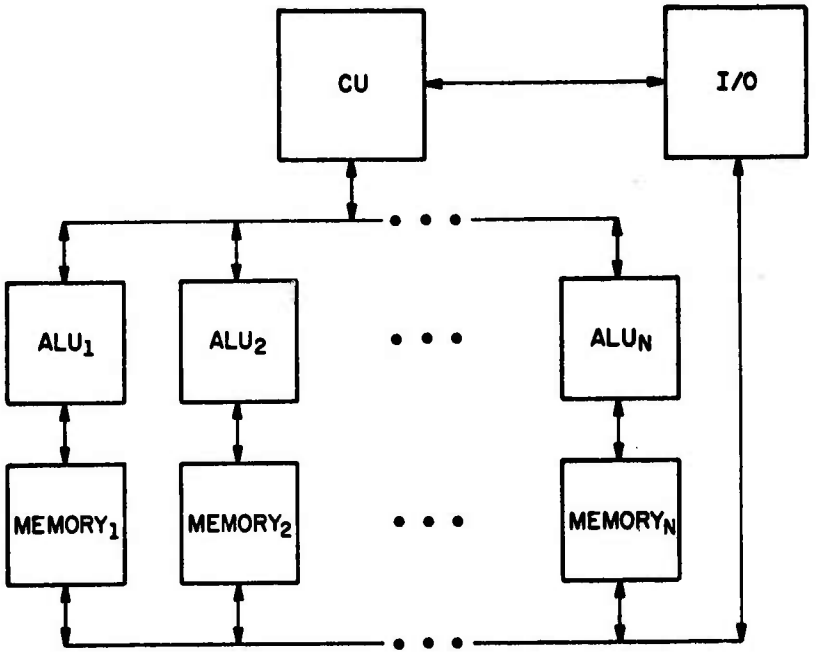


Figure I-12. A Vector or Array Processor

are actually doing is adding the vector A to the vector B and storing the result to a vector named C:

$$A = (a_1, a_2, \ldots, a_n)$$

$$B = (b_1, b_2, \ldots, b_n)$$

$$C = (a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n)$$

Since there is only one Control Unit, the ALU Array can only respond in a "lock-stepped" mode to each instruction. For example, if the instruction is ADD, then all N of the ALUs perform the ADD operation; there is no instruction which can cause some ALUs to add while others are multiplying. Every ALU of the Array performs the operation in this lock-stepped fashion, but the operands are vectors whose components can be and usually are different.

There is a nice distinction that can be drawn at this point between the operation of a Pipeline Processor and that of an Array Processor:

In a Pipeline, each stage performs a different operation simultaneously. In an Array Processor, each ALU performs the same operation simultaneously.

4. ILLIAC IV

ILLIAC IV is a direct descendant of the SOLOMON Computer which was designed by D. L. Slotnick and built by the Westinghouse Corporation in prototype. Before we take our first look at ILLIAC IV, let us briefly examine the SOLOMON Computer.

SOLOMON has 1024 Processing Elements (PE) each element having 4096 bits of storage. However, since all operations performed within a PE are serial-by-bit (and not parallel as in current conventional machines) the speedup factor is not 1024 but 1024/N where N is the number of bits in a word. The serial-by-bit operation of the PEs decreases the speed of the machine but it also lowers its cost and makes possible variable word lengths.

Each PE has its own memory but can be instructed to reference the memory of its four closest neighbors. What constitutes a neighbor is shown by Figure I-13. If the PEs are viewed as a 32 x 32 array, each PE (except the border ones) has a closest North, East, South and West

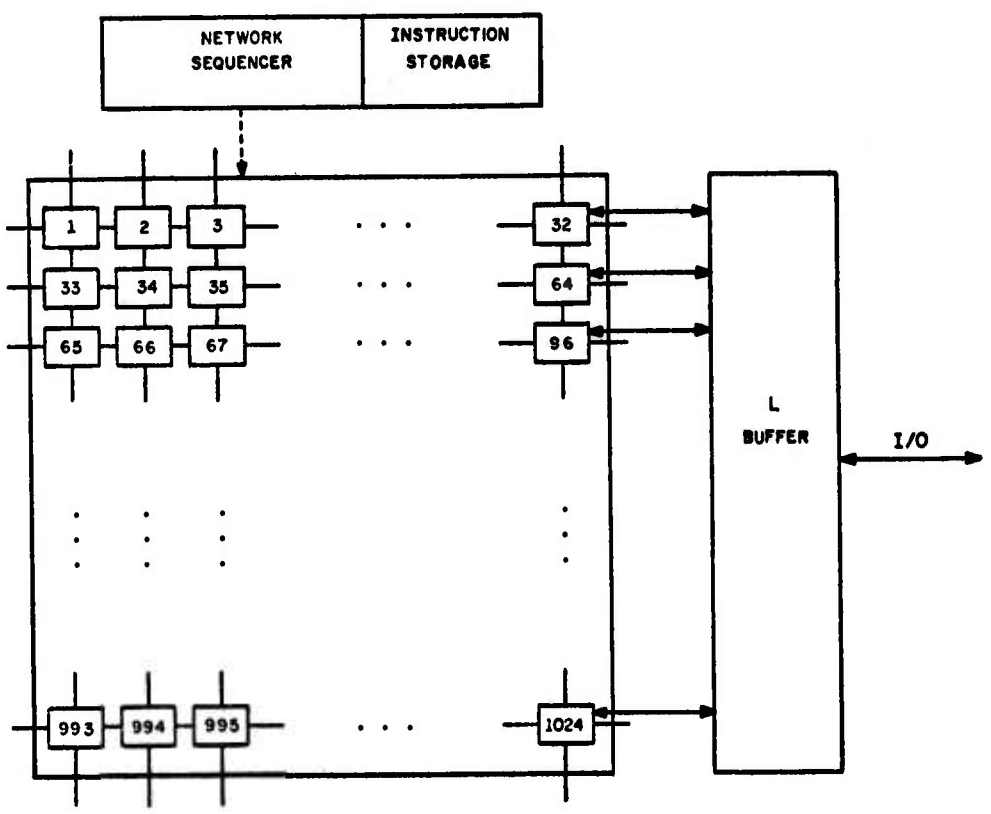


Figure I-13. Functional Block Diagram of SOLOMON

neighbor. Hardware connections between these PEs allow for the transfer or routing of information from PE to PE.

Also shown in Figure I-13 is the Network Sequencer or Control Unit which interprets the instructions stored from a special Instruction Storage Memory. (Data is stored within an individual PE memory and so program and data are not stored together on SOLOMON.)

The border PEs (Numbers 1, 2, 3, ..., 32; 33 and 64; 65 and 96; ..., 993, 994, 995, ..., 1024) all have at least one free connection that can, under program control be linked to other border PEs. This allows the programmer to configure the PE memory routing connections to suit the problem.

The Input/Output is handled by the L-Buffer which has direct connections only to the rightmost column of PEs (Numbers 32, 64, 96, ..., 1024). Once data has been loaded into these PEs via the L-Buffer, it can be further distributed via the "4 nearest neighbor" connections which exists within the array.

Each PE also contains a programmable mode register which determines whether or not that PE will or will not respond to an instruction generated by the Network Sequencer.

A later version of SOLOMON, SOLOMON II, upgraded each PE to parallel bit operation and added an index register so that each PE can access different locations within its memory as all PEs perform the same operation simultaneously.

The original design of ILLIAC IV contained four Control Units: each of which controlled a 64 ALU Array Processor. The version being built by the Burroughs Corporation will have only one Control Unit which drives 64 ALUs as shown in Figure I-14. 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 book.

The Control Unit (CU) has been decoupled from 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

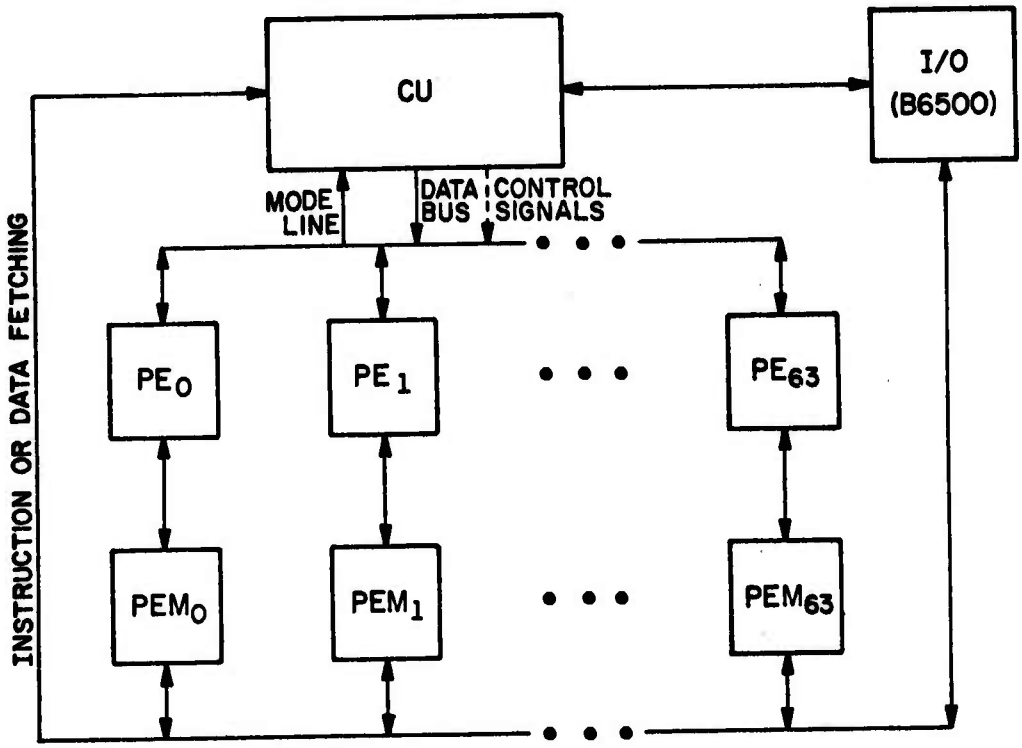


Figure I-14. Functional Block Diagram of ILLIAC IV

operations. In this way another degree of parallelism is exploited in addition to the inherent parallelism of 64 ALUs being driven simultaneously.

Each ALU responds to appropriate instructions if the ALU is in an active mode. (There exist instructions in the repertoire which can activate or de-activate an ALU.) Each ALU performs the same operation under command from the CU in the lock-stepped manner of an Array Processor. Each element of the ALU Array is not called by its generic name (ALU) but is called a Processing Element or PE.

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 Processing Element Memory (PEM) which can be accessed in about 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 first, fourth and sixth data formats listed above 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 nanoseconds, it is capable of performing almost 10^{10} of these "short"

additions per second. ILLIAC IV can perform approximately 150 million 64-bit, rounded, normalized floating-point additions per second.

The I/O is handled by a B6500 Computer System. The Operating System, including the assemblers and compilers, also reside in the B6500.

The specific option of centralizing the CU of the general multiprocessor as a basis for ILLIAC IV was chosen for two main reasons:

1) Cost. A very high percentage of the cost within a digital computer is tied up in the Control Unit circuitry. Replication of this component becomes very expensive and therefore choosing the option of centralizing the Control Unit can save more money than centralizing either Memory or the ALU.

2) There exist large classes of problems where the data to be manipulated can be expressed as vectors and not scalars. These problems range from scientific ones dealing with matrices and the solution of ordinary and partial differential equations to business problems as practical as payroll. In a business problem such as payroll, the same algorithm (payroll deduction) is applied to different data (each individual in the company has a different base pay and has chosen different deduction options). One Control Unit can apply the same algorithm repetitively to the different data (each data point can be thought of as a component of a vector--each component is operated on by a different PE). ILLIAC IV was designed especially to solve large problems wherein the same algorithm is performed repetitively on data that can be structured as components within a vector.

CHAPTER I -- REFERENCES

1. Graduate Catalog for the Department of Computer Science, University of Illinois.

2. R. K. Richards, Electronic Digital Systems. New York: John Wiley and Sons, Inc., 1966.

3. R. V. Wilkes, Automatic Digital Computers. New York: John Wiley and Sons, Inc., 1956.

4. Saul Rosen, "Electronic Computers: A Historical Survey," Computing Surveys, March 1969, Vol. 1, No. 1, pp. 7-36.

5. Lee A. Hollaar, "A History of Computer Organizations," ILLIAC IV Document No. 212, Department of Computer Science File No. 819, Urbana, Ill.: University of Illinois at Urbana-Champaign, May 8, 1970.

6. D. L. Slotnick, W. C. Borck, and R. C. McReynolds, "The SOLOMON Computer," Proceedings AFIPS 1962 Fall Joint Computer Conference, Vol. 22, pp. 97-107 (Spartan Books, Washington, D. C.).

7. D. L. Slotnick, "Unconventional Systems," Proceedings AFIPS 1967 Spring Joint Computer Conference, Vol. 30, pp. 477-481 (Spartan Books, Washington, D. C.)

CHAPTER II -- HARDWARE STRUCTURE

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

CHAPTER II

HARDWARE STRUCTURE

A. Summary

The ILLIAC IV System can be organized as in Figure II-1. 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 Control Unit. In turn, the Array Processor is made up of 64 Processing

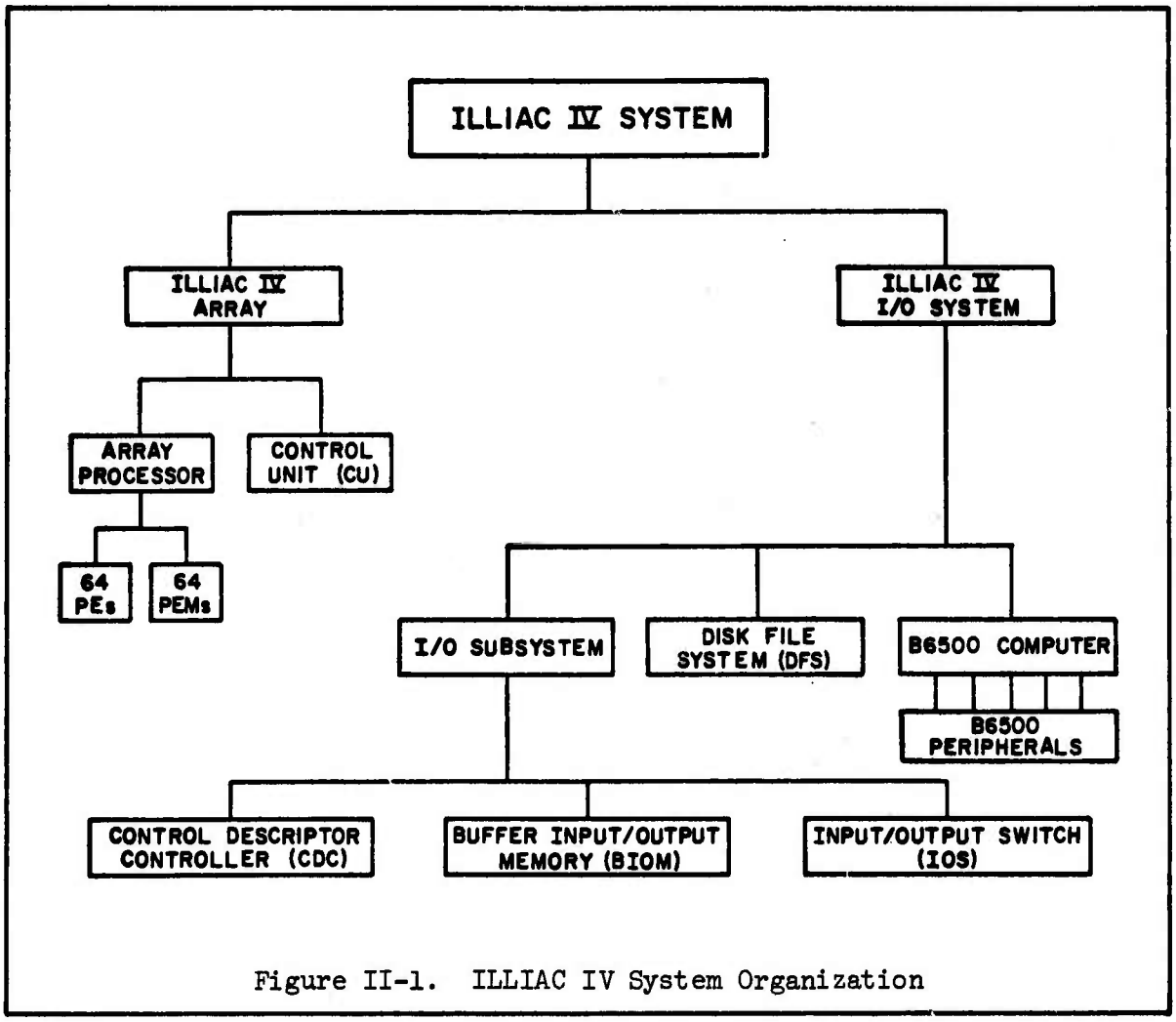


Figure II-1. ILLIAC IV System Organization

Elements (PEs) and their 64 associated memories--Processing Element Memories (PEMs). The ILLIAC IV I/O System comprises the I/O Subsystem, the Disk File System and the B6500 computer. The I/O Subsystem is broken down further to the CDC, BIOM and IOS. The B6500 is actually a large-scale computer system by itself.

The ILLIAC IV Array will be discussed first, in a general manner, followed by some 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. Following is a more detailed description of the ILLIAC IV Array, then another illustrative problem, this time a more realistic one--solution of the temperature distribution on a two-dimensional slab; some data allocation considerations are then discussed. The ILLIAC IV I/O System is discussed briefly, and some conclusions and opinions end the chapter.

B. ILLIAC IV Array--General Description

Figure II-2 represents the ILLIAC IV Array--the Control Unit plus the Array Processor.

1. Control Unit (CU)

The Control Unit can be viewed as a small unsophisticated computer in its own right. Not only does it cause the 64 Processing Elements to respond to instructions, there is a repertoire of instructions

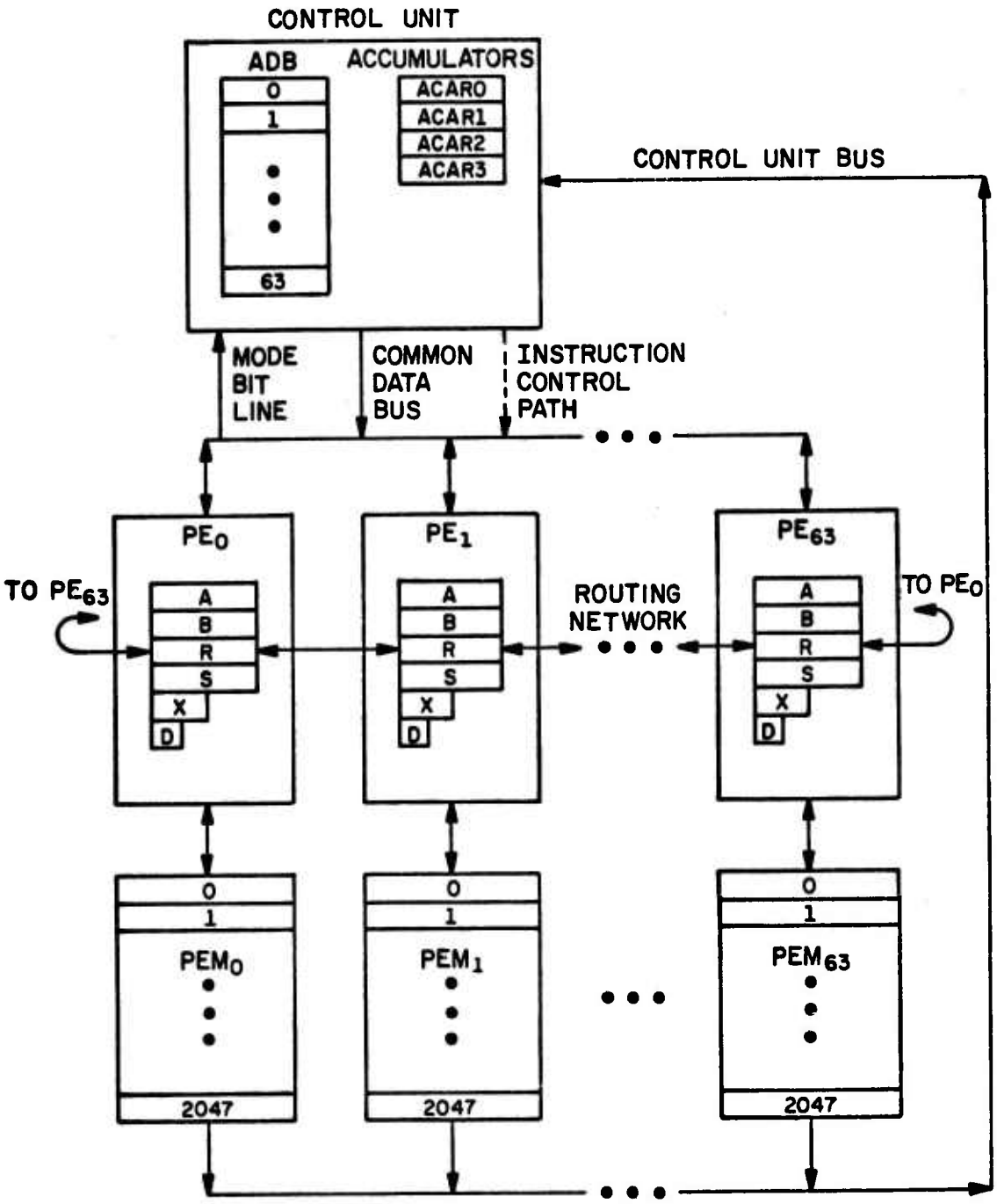


Figure II-2. ILLIAC IV Array

that can be completely executed within the resources of the Control Unit, and the execution of these instructions is overlapped with the execution of the instructions which drive the Processing Element Array.

The Control Unit contains 64 integrated circuit registers called the ADVAST Data Buffer (ADB) which can be used as a high speed scratch-pad memory. ADVAST is one of the five functional components of the CU and will be described in greater detail in section D 3 a. 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).

2. Processing Element (PE)

Each Processing Element (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 mode register 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.

3. Processing Element Memory (PEM)

Each PE has its own 2048 word, 64-bits per word, random access memory. Each memory is called a Processing Element Memory or PEM and they are numbered 0 through 63 also. A PE and PEM taken together is 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 described next in section B 4 c.

4. Data Paths

Besides the Instruction Control Path which drives the 64 PEs during the execution of an instruction there are four paths by which data flows through the ILLIAC IV Array. These paths are called the Control Unit Bus (CU Bus), the Common Data Bus (CDB), the Routing Network, and the Mode Bit Line.

a. Control Unit Bus (CU Bus)

Operands or data from the PEMs in blocks of eight words can be sent to the CU via the Control Unit Bus (CU Bus). The instructions to be executed are distributed throughout the PEMs and are also fetched in blocks of eight words to the CU via the CU Bus as necessary. Some of the instructions are completely executed within the CU; these are called ADVAST instructions. Most of the instructions, however, cause the 64 PEs to perform an operation simultaneously or in parallel; these are called FINST/PE instructions and are made ready for execution by the PE Array in a section of the Control Unit called FINST. The operation of ADVAST and FINST will be more fully described in section D of this chapter.

b. Common Data Bus (CDB)

Information stored in the Control Unit can be "broadcast" to the entire 64 PE Array simultaneously via the Common Data Bus (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.

c. 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. See Figure II-3 for a picture of the connectivity which exists between PEs. As can be seen from Figure II-3, PE_9 is connected by routing lines to PE_1, PE_{10}, PE_{17}, and PE_8. PE_0 is connected to PE_{56}, PE_1, PE_8, and PE_{63}.

d. 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 Control Unit. Since each bit of an ACAR holds one bit of each RGD for the PE array, special Control Unit instructions can test and branch on the "mode pattern" in the ACAR.

In a very gross fashion then, this is the ILLIAC IV Array. In order to illustrate how all of this hardware is tied together, we shall next look at some simple problems which utilize the Array Processor of ILLIAC IV.

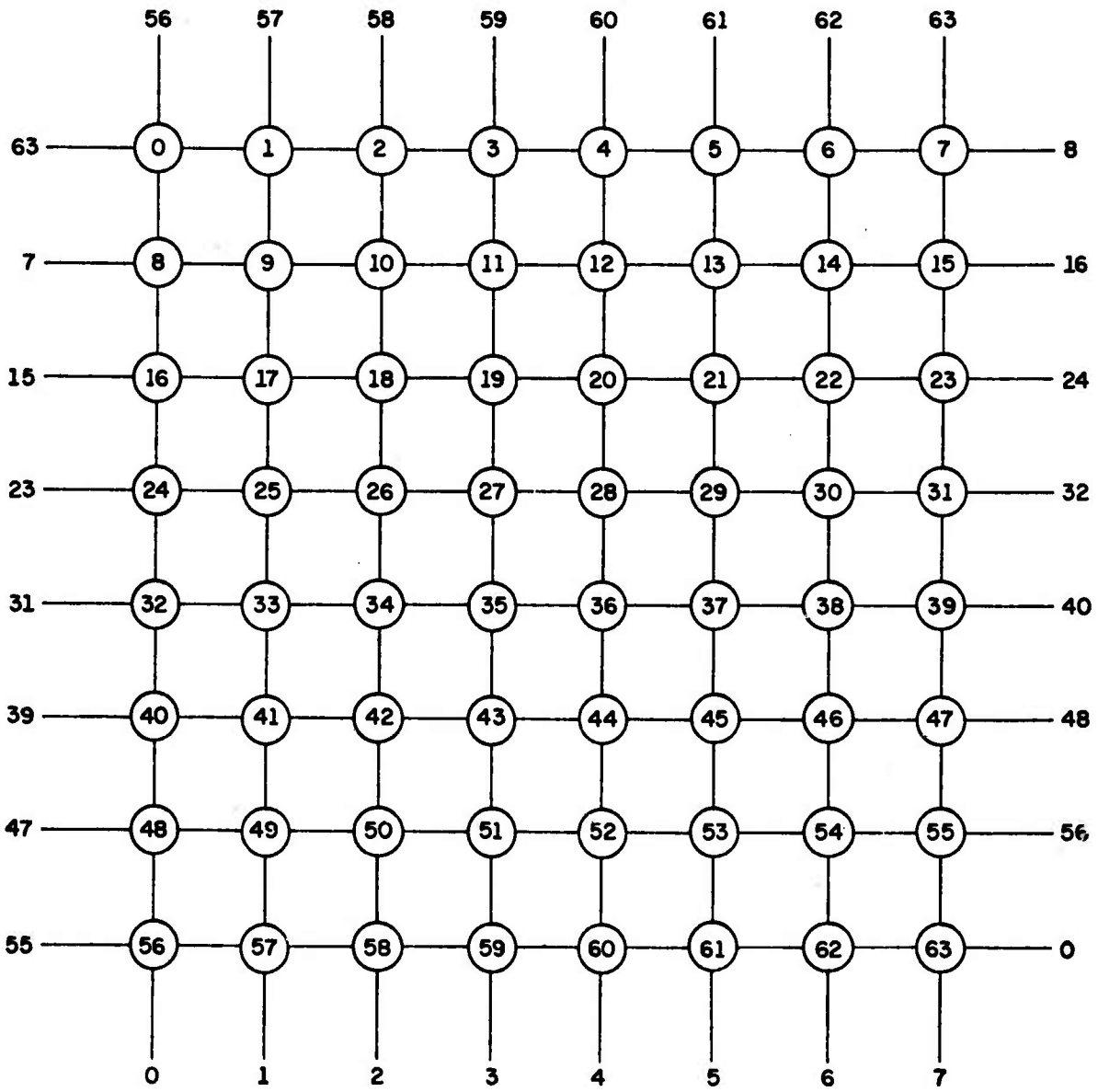


Figure II-3. PE Routing Connections

C. Some Illustrative Problems

1. 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 are executed N times. Therefore, if it takes M microseconds to pass once through the loop, it will take about N times M microseconds 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 PE Memory. 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.

a. $\underline{N = 64}$

To reflect the case where $N = 64$, we have arranged the data as shown in Figure II-4. 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.

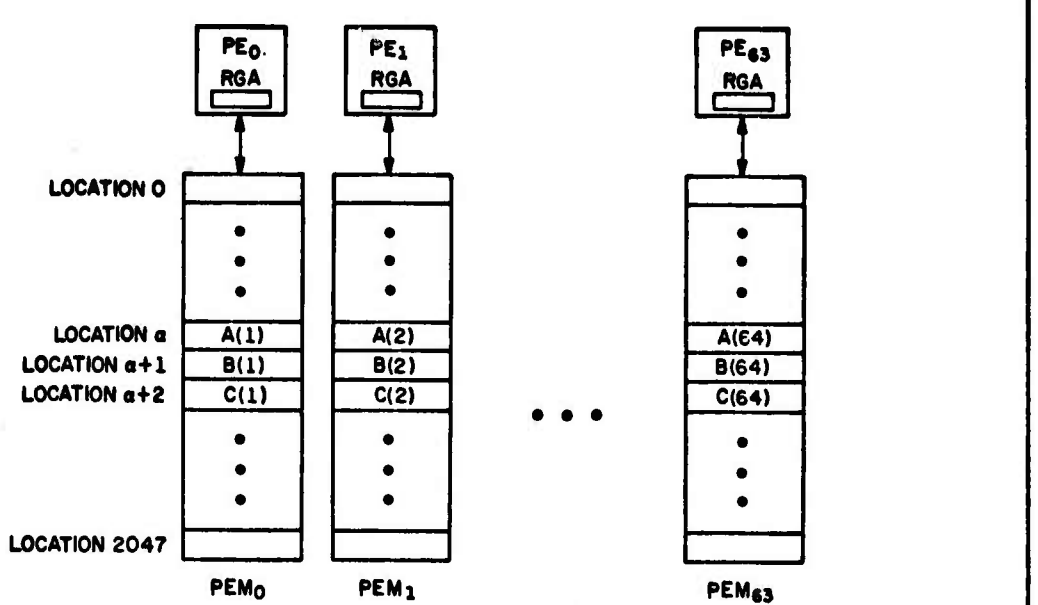


Figure II-4. Arrangement of Data in PEM to Accomplish

```
DO  10  I = 1, 64
10  A(I) = B(I) + C(I)
```

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, 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.)

b. $N < 64$

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 the total PE array will not be utilized.

The trade-off the potential user of ILLIAC IV must consider here is how much (or how often) is ILLIAC IV under-utilized? 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 doesn't feel too guilty if he under-utilizes the resources of

a conventional system--he doesn't use every tape drive, every bit of available core, every printer and every byte of disk space for most of his conventional programs.

c. 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 Figure II-5.

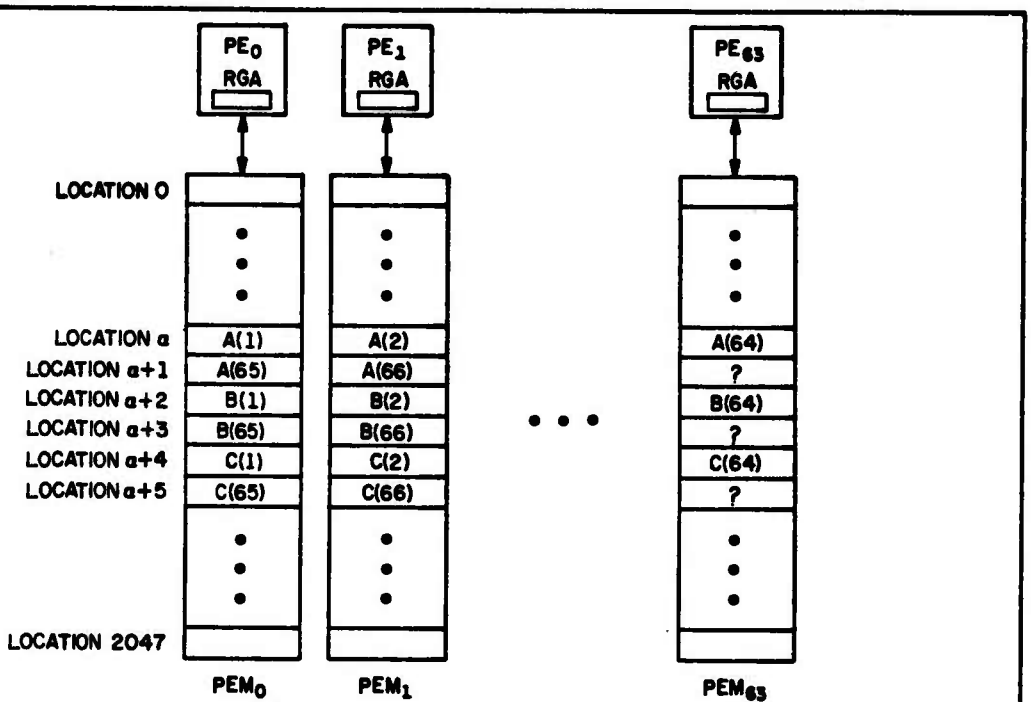


Figure II-5. Arrangement of Data in PEM to Accomplish

```
DO  10  I = 1, 66
10  A(I) = B(I) + C(I)
```

To perform the 66 additions on the data stored as shown in Figure II-5, 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 Figure II-5 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.

2. Adding Two Unaligned Arrays

Now let us consider how we would perform the following FORTRAN statements using ILLIAC IV:

```
      DO  10  I = 2, 64
10    A(I) = B(I) + C(I-1)
```

This program could be effected in either of two ways: One way would be to store the C array "skewed" or offset one element to the right at compilation time; the other way is to store the C array normally and perform the skewing at execution time.

a. Store the C Array Skewed

When we choose this method to effect the FORTRAN program, we store the data as shown in Figure II-6. Before executing the program, the user may wish to disable PE_0. By storing the data skewed to begin with, we

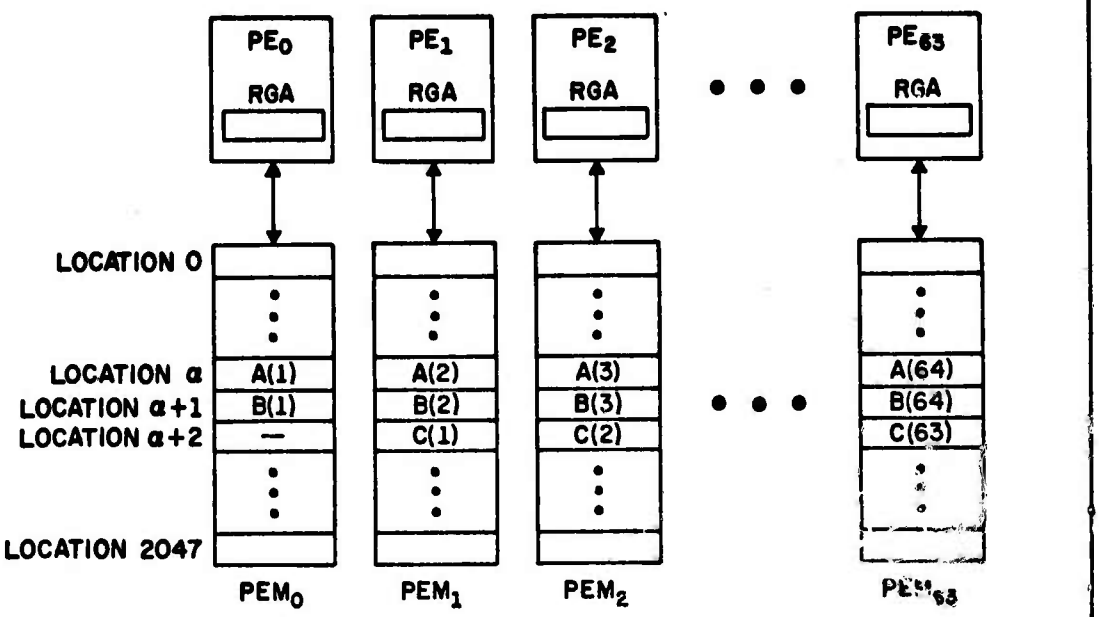


Figure II-6. Arrangement of Data in PEM to Accomplish

```
DO  10  I = 2, 64
10  A(I) = B(I) + C(I-1)
```

accomplish our goal at compilation time and the execution time instructions would be of the form

DISABLE PE_0 (optional)

LOAD RGA from Location $\alpha + 2$;

ADD to RGA contents of Location $\alpha + 1$;

STORE result to Location α;

b. Skewing at Execution Time

The second way to effect this program is to store the data "straight", i.e., exactly as shown in Figure II-4 but to have the ILLIAC IV program skew the data using the ROUTE instruction; then the addition is performed as above. The ILLIAC IV commands would be of the form:

1) All PEs LOAD RGR from Location $\alpha + 2$.
2) All RGRs ROUTE contents one PE to the right (the route is end-around so that RGR of PE_{63} goes to RGR of PE_0).
3) All PEs LOAD RGA from RGR.

   Note: After this third instruction is executed the data is stored as shown in Figure II-7.
4) All PEs ADD to RGA the contents of Location $\alpha + 1$.
5) All PEs STORE the result in RGA to Location α.

Note that after the execution of the above five instructions, A(1) will contain B(1) + C(64) if PE_0 was not disabled.

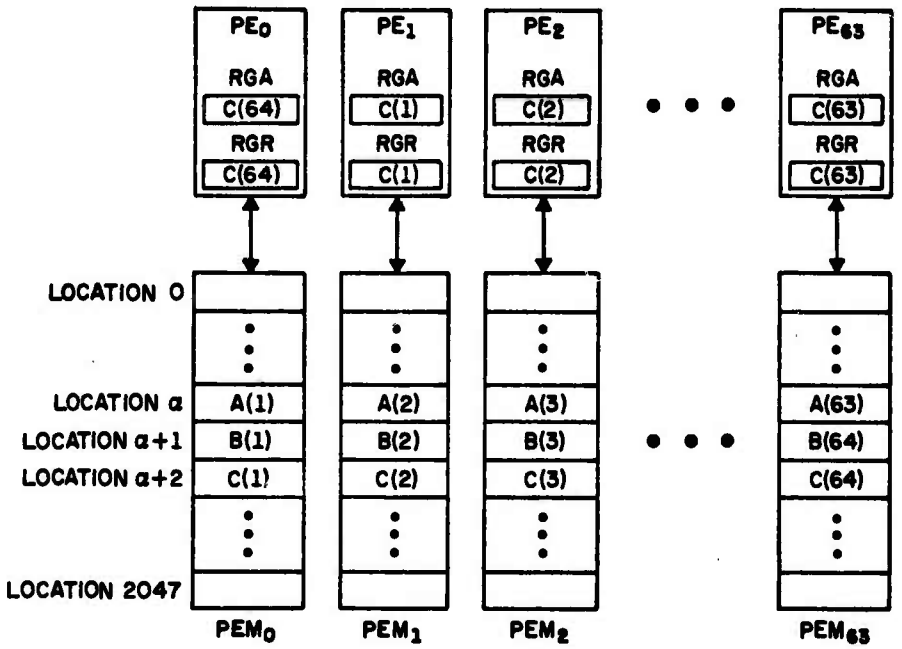


Figure II-7. Status of Data in PEM, RGA and RGR while Executing

```
DO  10  I = 2, 64
10  A(I) = B(I) + C(I-1)
```

The contents of PEM, RGA and RGR are shown after Step 3 of the program on page II-15.

3. 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 two FORTRAN statements as they stand. However, using mathematical subscript notation:

$$A_2 = B_2 + A_1$$

$$A_3 = B_3 + A_2 = B_3 + B_2 + A_1$$

$$A_4 = B_4 + A_3 = B_4 + B_3 + B_2 + A_1$$

$$\vdots$$

$$A_N = B_N + B_{N-1} + \ldots B_2 + A_1$$

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 Figure II-8 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 numbered 0 through j. (Turn them OFF.)
8) All enabled PEs ADD to RGA the contents of RGR. (Figure II-8 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 go to STEP 11.
11) Enable all PEs.
12) All PEs STORE the contents of RGA to Location $\alpha + 1$.

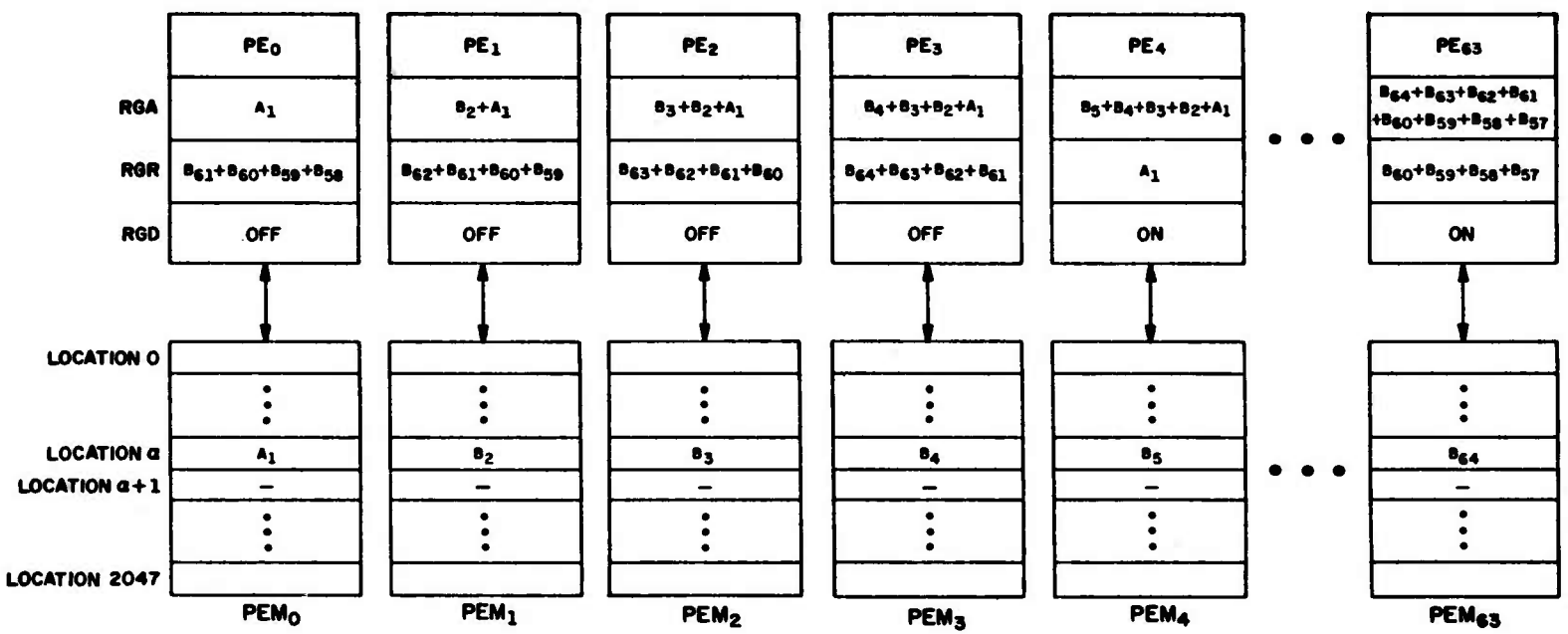


Figure II-8. Status of Data in PEM, RGA, RGR, and Mode Status (RGD) while Executing

```
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 on page II-18.

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.

D. ILLIAC IV Array--A More Refined Description

Section B presented a general description of the functional components of the ILLIAC IV Array. This section will expand on that description.

1. Processing Element (PE)

The Processing Element (PE) is shown in Figure II-9. For the sake of clarity, all of the interconnections between the six registers have not been shown in Figure II-9.

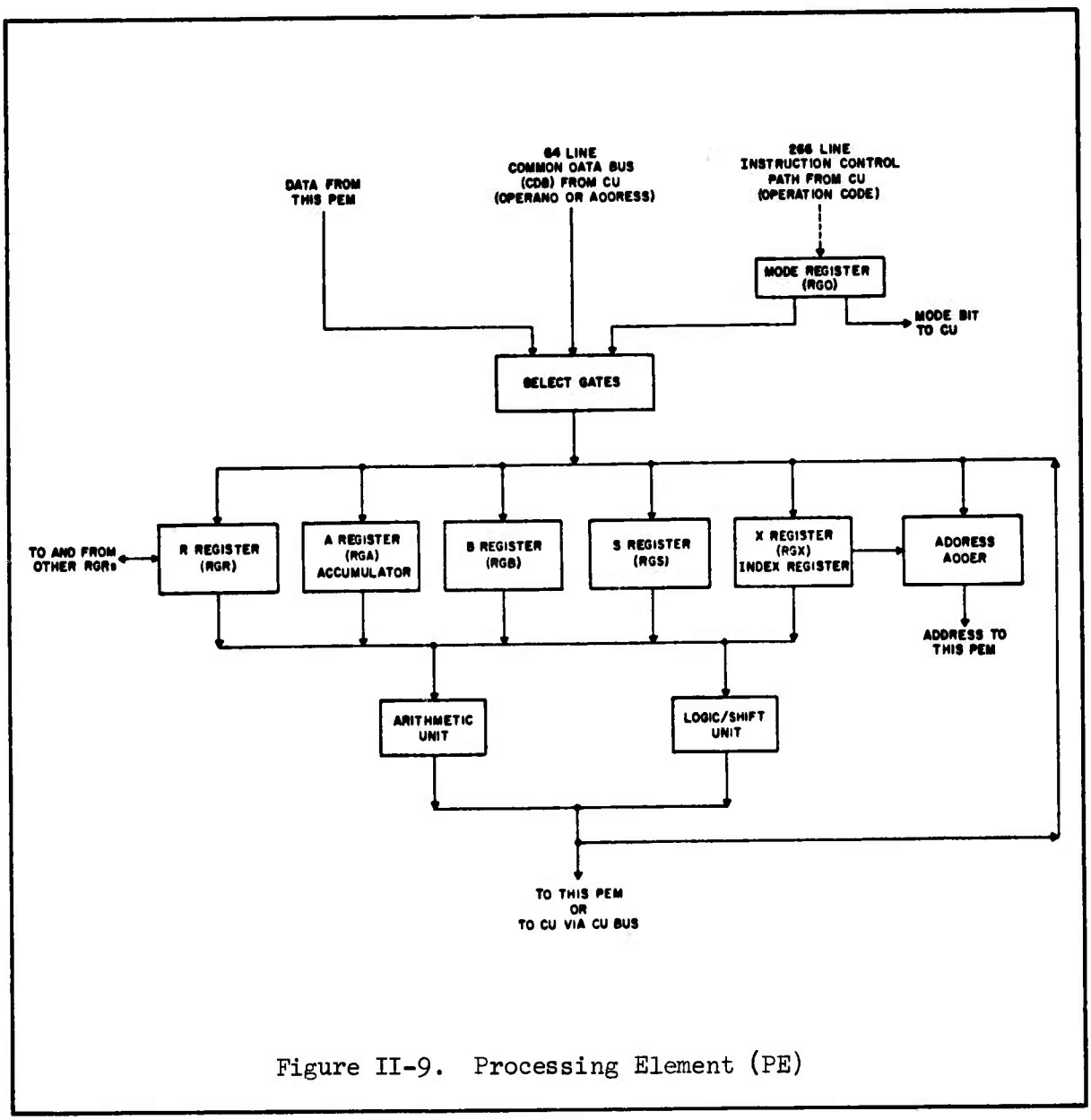


Figure II-9. Processing Element (PE)

Note that information enters a PE either from its own PEM, from another PE (via RGR) or from the Control Unit (CU). The CU sends the electrical pulses down the Instruction Control Path which are the microsequences that define the operation code of the instruction to be executed by the PE Array. The Instruction Control Path consists of 266 control lines which drive the 64 PEs simultaneously; i.e., all PEs execute the same instruction in parallel. The microsequences pass through the Mode Register (RGD) before going on to the Select Gates of the PE. The Mode Register is an eight-bit register which contains (in addition to other information) the status of the PE. In section B 2 it was pointed out that if the PE is in the enabled status or mode, then the instruction is completely executed (the proper gates will be selected); if the PE is in a disabled mode, then it will not respond to the instruction. As a general description, this is true but it presents an incomplete picture of the operation of the mode register. Following is a more complete description:

a. Mode Register (RGD)

The mode register (RGD) has eight bits called the E, E1, F, F1, G, H, I and J bits. The E and E1 bits are used to reflect the status of a PE.

If both E and E1 are zero then writes (storing of information) to RGS, RGA, RGX and PE Memory are prohibited or locked out; writes to RGR, RGB, and RGD are allowed--they are not locked out. When both E and

El are zero we refer to the PE as being in the disabled state even though RGR, RGB and RGD can be changed by an instruction that references these registers; however, any part of an instruction which seeks to modify RGS, RGA, RGX or a PE Memory location will not be performed. Reads of all registers and PE Memory are not locked out when a PE is disabled.

If E and El are one then we refer to the PE as being in the enabled state and all instructions are completely executed--no PE registers or part of PE Memory is locked out.

In brief then, when a PE is disabled, its RGS, RGA, RGX and PE Memory are protected--when a PE is enabled, its RGS, RGA, RGX and PE Memory are unprotected.

Let us now take a closer look at what it means when a PE is disabled: When a PE is disabled (E = El = 0) RGR, RGB and RGD are unprotected and one of two things may occur:

1) An instruction which directly modifies only RGR, RGB, or RGD will be completely executed. For example, the following types of instructions will be executed when a PE is disabled:

LOAD RGR from RGA

ROUTE RGR N PEs to the Right (or Left)

LOAD RGB from RGA

LOAD RGD from RGB (the eight high order bits of RGB go into RGD)

SET one of the eight bits of RGD.

2) An instruction which <u>indirectly</u> modifies RGR, RGB or RGD will be partially executed. (An indirect instruction is taken to mean one that is intended to change the contents of RGS, RGA, RGX or PE Memory, but in doing so must use and change the contents of RGR or RGB.) For example, if an ADD instruction is sent to a disabled PE, the PE will actually perform all of the microsequences necessary for addition, changing the value of RGB, but RGA will not be changed--the answer will not appear in the accumulator. Since the second operand of a binary operation is fetched to Register B, RGB gets modified (indirectly) during an ADD operation in a disabled PE.

For example, the following types of instructions cause indirect modification of either RGB or RGR:

ADD to RGA the contents of PEM location X (RGB is modified)

MULTIPLY the contents of RGA by the contents of PEM location X (RGB and RGR are modified)

DIVIDE the contents of RGA by the contents of PEM location X (RGR is modified)

However, none of the above instructions modify RGS, RGA, RGX or PE Memory, since we are considering the case when a PE is disabled (E = E1 = 0).

There are no ILLIAC IV instructions which modify RGD indirectly, so the programmer does not have to worry about inadvertently changing the mode pattern of the PE Array (the mode pattern is just the 64 states of the E and E1 bits in the PE Array). The programmer must, however, have the

capability to modify the mode of a disabled PE, else after he turned it off he could never turn it back on. Since RGD can be modified directly when a PE is disabled, the programmer is afforded this capability by various instructions in the ILLIAC IV repertoire.

The general rule which always holds true is: When a PE is disabled (E = E1 = 0) RGS, RGA, RGX and PE Memory are protected (writes are locked out). When a PE is enabled (E = E1 = 1) RGS, RGA, RGX and PE Memory are not protected (writes are not locked out).

If the programmer remembers this rule he can understand better the operation of each instruction in the repertoire. Another way to say the rule is: "Not all parts of a PE are disabled in a disabled PE; RGR, RGB and RGD can still respond to an instruction in a disabled PE. The PE is disabled, not dead when its E and E1 bits of RGD are zero".

Still this is not yet the complete story for up to now we have only been considering instructions which process operands in the 64-bit mode. (The word "mode" here has nothing to do with the mode register RGD--it is used only to be consistent with other literature; "code" or "format" in place of the word "mode" would be a better choice.) Actually the E1 bit protects the inner part of a word (bits 8-39) in RGS, RGA, RGX or PE Memory and the E bit protects the outer part of a word (bits 0-7 and 40-63) in RGS, RGA, RGX or PE Memory. The convention described above still holds: If E or E1 is <u>zero</u> the appropriate bits within RGS, RGA, RGX or PE Memory are protected; if E or E1 is <u>one</u> the appropriate bits are unprotected.

In the 64-bit mode where all 64 bits are necessary to represent one number, E and E1 work together to protect the word in which the number is stored. Since it makes no sense to protect the inner part of 64 bit floating point numbers and not protect the outer part, we always have E equal to E1 when executing instructions in the 64-bit mode. However, there are instructions which assume that their operands are in the 32-bit mode in which case we have two numbers per ILLIAC IV word. In this case the E and E1 operate independently and can be of opposite values. This type of operation and the operation of the E and E1 bits with the fault bits, F and F1, is described more fully on pages 4-14 through 4-16 of Reference 1.

b. The Rest of the PE

The Common Data Bus (CDB) carries the address portion of the instruction to be executed and is 64 bits wide (consists of 64 lines). The signals on these 64 lines go to every PE in the 64 PE array. As is the case with a conventional computer, the operand may be an address, a count, or a number.

Depending on the type of instruction, either the Arithmetic Unit or the Logic/Shift Unit is actuated and the result is sent to PEM, to the appropriate PE register, or to the CU via the CU Bus.

RGA is the Accumulator and acts like an accumulator on a conventional machine. RGB can be used to hold the second operand in a binary operation or act as an extension to RGA for double length operands. RGS is

a temporary storage register and may be used as the programmer sees fit. Since RGR, RGB and RGD can be modified in disabled PEs, RGS is a good, safe place for the programmer to store intermediate results. RGR is called the Routing Register and can be viewed as a port to transfer data to and from other PEs. Every PE has four bi-directional lines from its RGR to the RGR of the PEs a distance of +1, -1, +8 and -8 away. RGX is an index register and is used to modify the address portion of an instruction in the same manner as on a conventional computer. All registers are 64 bits long except for RGX which is 16 bits long and RGD which is 8 bits long.

The Mode Bit Line consists of a unidirectional one-bit line running out from the RGD of each PE to the register storage section of the CU. Using this path, the programmer can load an ACAR register with a pattern of 64 bits, each one coming from the same mode register bit from each of the 64 PEs. Conversely, the contents of an ACAR can be used to set a specified bit within the mode register of each PE in the array: bit 0 of the ACAR is transmitted to the specified bit of RGD of PE_0 ... bit 63 of the ACAR is transmitted to the specified bit of RGD of PE_{63}. A special version of this instruction exists whereby bit i of the ACAR is transmitted to <u>both</u> the E and El bits of RGD of PE_i so that the entire array can take on a specified mode pattern in just one instruction. The transmission of a mode pattern stored in an ACAR down to the PE array does not take place over the one-bit mode line (which is unidirectional <u>from</u> the PE <u>to</u> the CU); this transmission comes via the CDB.

2. Processing Unit (PU)

Figure II-10 depicts a Processing Unit (PU). A Processing Unit (PU) consists of three components: 1) a PE, 2) a Memory Logic Unit (MLU) and 3) a PE Memory (PEM). The PE has already been described.

a. Processing Element Memory (PEM)

The 2048 word PEM has an effective 350 nanosecond (ns) access time. This 350 ns effective access time is comprised of a 250 ns Read or Write Cycle Time and a 100 ns delay due to the additional logical checking

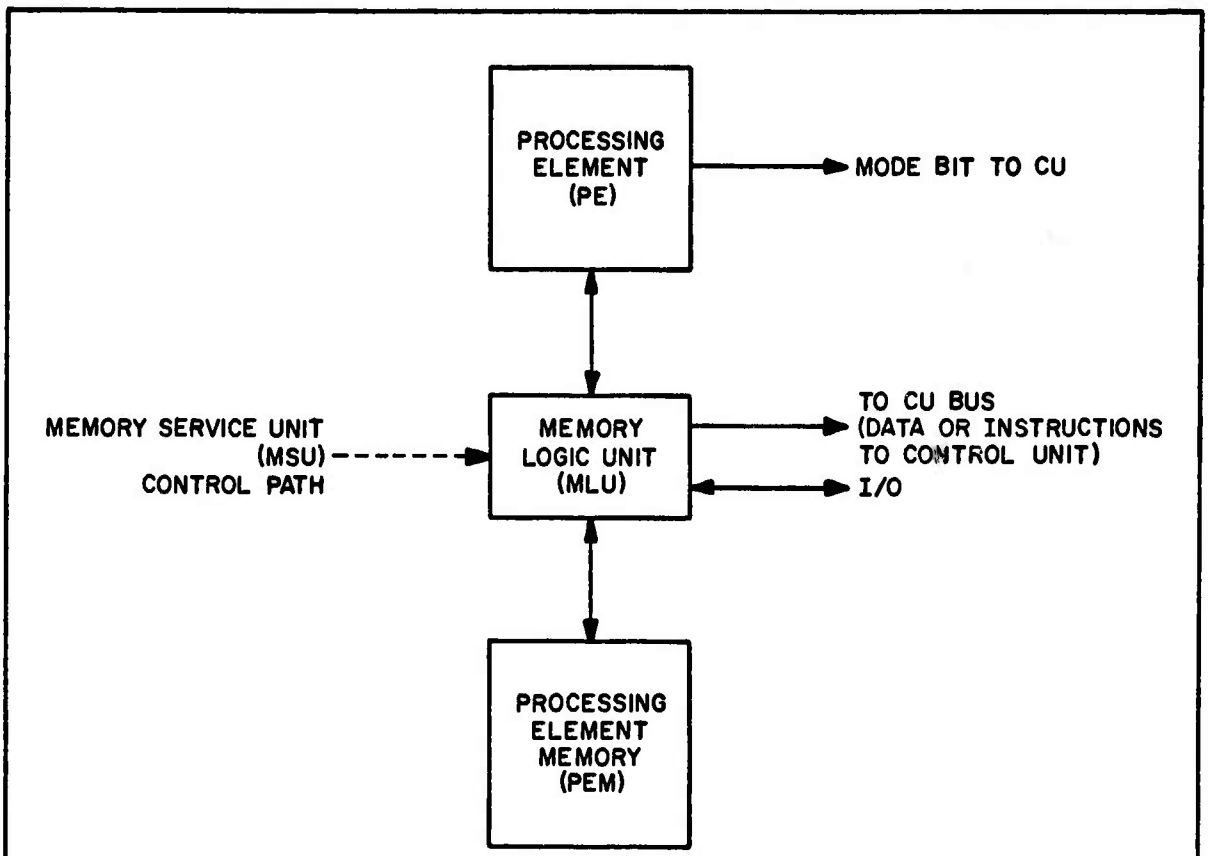


Figure II-10. Processing Unit (PU)

circuitry of the MLU. The 250 ns Read or Write Cycle time consists of 188 ns data access time and up to 62 ns to complete the cycle. READ and WRITE work in the following manner:

READ: Data can be accessed (sent on its way to the PE or elsewhere) in 188 ns but it takes 62 more ns for the memory to complete the cycle, during which time, memory is locked out or is not interrogatible.

WRITE: The data word is written into memory in 188 ns and control can return to processing. However, memory cycle is not over for another 62 ns, so memory cannot be interrogated for 250 ns as above.

In general, this means that if the next instruction after a memory reference does not also reference memory, it can be performed 188 ns later; however if that instruction does reference memory, it will be performed up to 350 ns later.

b. Memory Logic Unit (MLU)

The MLU acts as a "switch" in that it resolves conflicts involving simultaneous accesses to the PEM. The MLU of each PE in the Array receives signals from the Memory Service Unit (MSU) in the Control Unit which allows one of the three possible users of PEM to gain access to the PEM. The three users are:

1) CU Bus to fetch instructions to the Instruction Look-Ahead Section (ILA) of the CU, or to fetch data to the register storage of the CU.

2) PE itself (Loading a PE register from PE Memory).

3) Input/Output devices (I/O).

Note that Figure II-10 has an arrow coming out of the MLU with the caption "To CU Bus". This is meant to imply that this line is not the CU Bus itself but is just a 64 bit line to the CU Bus. The CU Bus carries eight 64-bit words at a time from a PU to register storage in the CU. The CU Bus fetches words (in blocks of 8) from PE Memory, through the MLU, up to 8 specified locations in the ADB of the CU (there does exist another instruction whereby only one word is fetched and can therefore be stored in a CU register other than the ADB) when a certain ILLIAC IV instruction called BIN is executed. The CU Bus is also used by the Operating System to fetch instructions (which are also stored in PE Memory) up to ILA, the instruction execution section of the CU. The eight words (data or instructions) that are transmitted via the CU Bus are in contiguous PUs and always start at a PU number that is an exact multiple of eight.

Since there are 64 PUs and only 8 of them can use the CU Bus at one time, there is a switch which selects which group of 8 PUs will be connected to the CU Bus. There are eight PU Cabinets (PUCs) in the ILLIAC IV Array, each of which holds 8 PUs as shown in Figure II-11. The figure shows how groups of 8 words in contiguous PUs but at the same row memory location are connected to the CU Bus.

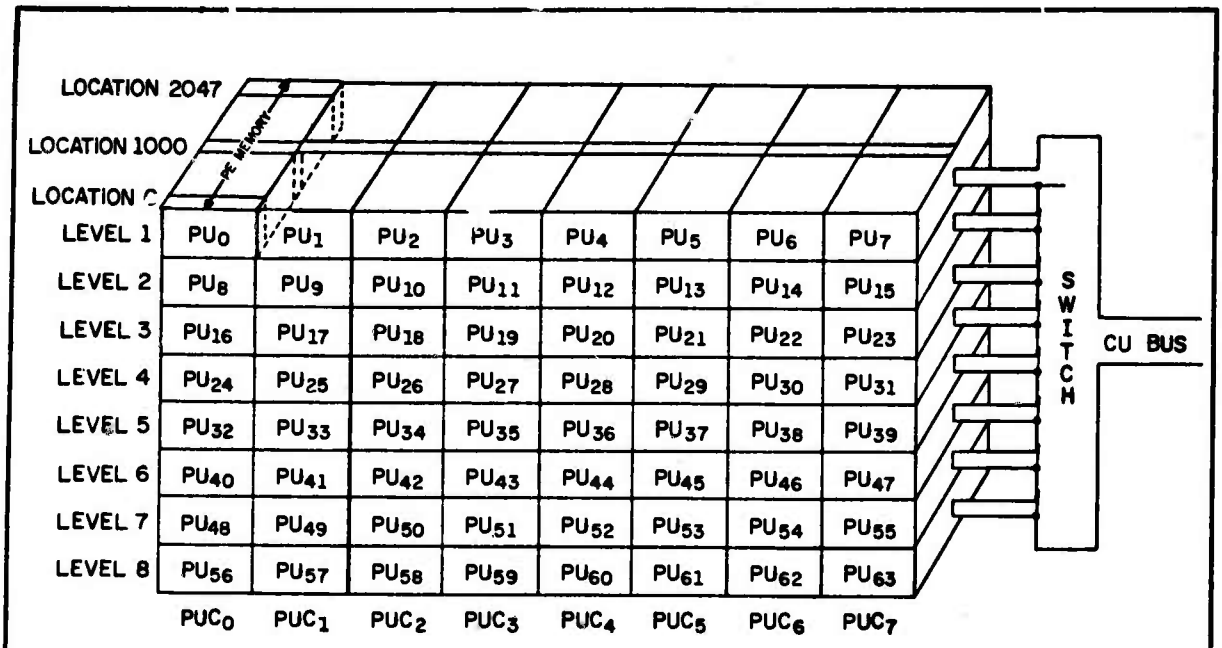


Figure II-11. PU Cabinets (PUCs) and CU Bus

Eight PUCs contain the 64 PUs in the ILLIAC IV Array. PE Memory is displayed in the depth dimension of each PU. The switch which is between the CU Bus and the PUCs selects (under program control) one of the eight levels (each level cuts across the 8 PUCs) and the same location of PE Memory for each of the 8 PUs within that level is sent to the CU Bus. The CU Bus then transmits these 8 words to the CU. The figure is drawn to depict location 1000 of level 1 (PUs numbered 0, 1, ..., 7) to be sent to the CU Bus.

3. Control Unit (CU)

The Control Unit may be viewed as consisting of five functional sections: ADVAST, FINST, MSU, TMU, and ILA.

a. ADVAST

ADVAST, the Advanced Station of the CU, is shown in Figure II-12. Its main area of responsibility is to execute instructions that do not reference information in the PUs as well as to pre-process the instructions

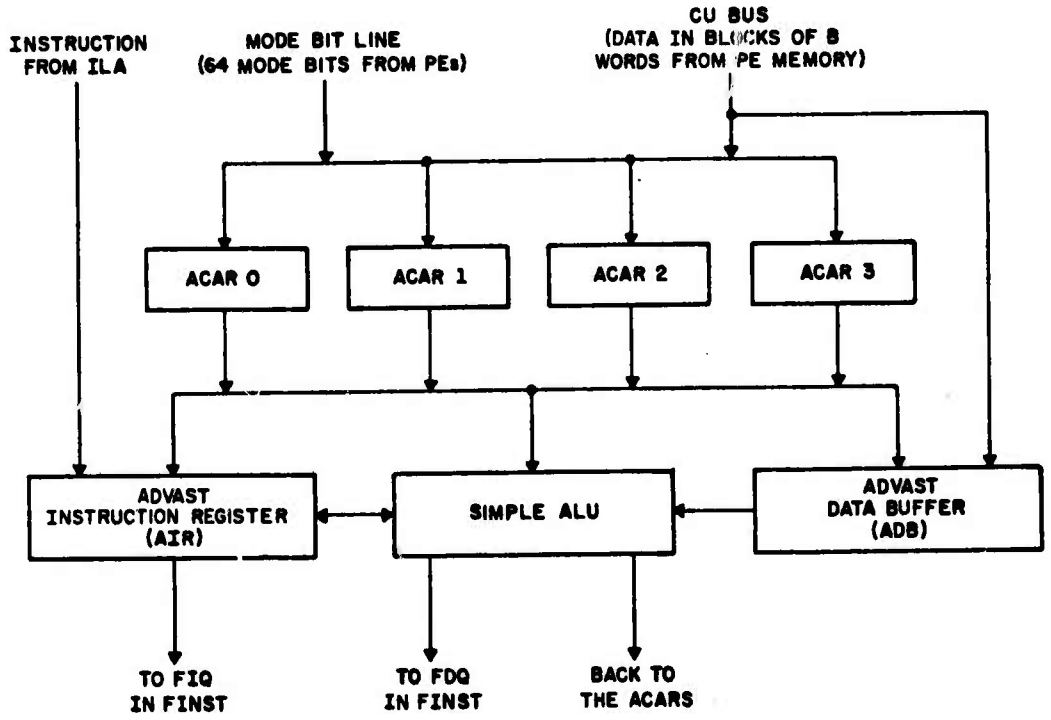


Figure II-12. Advanced Station (ADVAST) Section of the Control Unit

that do drive the PE Array. The instructions that drive the PEs are sent on to FINST which sends out the microsequences to the PE Array.

The ADVAST Section of the Control Unit may be viewed as a small computer by itself. It has four 64-bit accumulator registers--ACAR0, ACAR1, ACAR2, and ACAR3 and a 64-word, 60 nanosecond integrated circuit memory called the ADVAST data buffer (ADB); each word in ADB is also

64 bits long.* In addition it contains a simple arithmetic and logic unit (ALU) capable of instruction indexing, 24-bit integer addition and 64-bit logical operation on data from CU registers. ADVAST also has its own instruction repertoire (CU instructions) and is capable of executing them while the 64 PEs are simultaneously executing their own instructions.

As previously mentioned in section D 1 b, a special instruction executed by ADVAST can load up the bits of any ACAR to match a mode bit of RGD for each PE in the array at any time during program execution. Another instruction allows the programmer to set a mode bit in each RGD of the total PE array to match the contents of any specified ACAR. These two instructions allow the programmer to set up patterns to control which PEs will be enabled and which will be disabled during program execution. These instructions will be covered in Chapter III.

As Figure II-12 indicates, there are three sources of input to ADVAST: the instruction to be executed, the Mode Bit Line, and the CU Bus. The CU Bus brings in data in blocks of eight words from PE Memory. Since the Control Unit has access to PE Memory via the CU Bus, reference is sometimes made to "CU Memory"--this should not be confused with CU register storage (four ACARs, 64 ADB locations, thirteen other registers). CU Memory is actually part of PE Memory which is accessed by two ADVAST instructions which move the contents of PE Memory to the ADB or one of the four ACARs (or several of the other thirteen registers) via the CU Bus.

---

*There are thirteen other CU registers that can be accessed by the programmer; these are more fully described in Reference 1.

Instructions are also fetched via the CU Bus but they go to ILA, the instruction look-ahead section of the CU, not ADVAST.

The Mode Bit Line (one line from each RGD of each PE in the array) comes into ADVAST where a mode pattern can be stored in one of the four ACARs.

Instructions are sent from ILA to the ADVAST instruction register (AIR) where the instruction is interpreted. If it is an instruction that can be executed entirely by the CU, then it is executed; if it is not, it is sent on to FINST for execution.

Another possible input source to ADVAST is output from the ALU which can return as input to any of the four ACARs.

b. FINST

FINST is the Final Station of the Control Unit and receives only those instructions that require PE action. See Figure II-13.

Since ADVAST controls the instruction stream, all instructions pass through it for decoding first. If the operation involves only ADVAST hardware, then ADVAST completely executes the instruction so that the instruction never reaches FINST. If the instruction is a PE instruction, ADVAST decodes it, provides any indexing operations necessary at the Control Unit level (the address portion of an instruction can be indexed by the contents of one of the ACARs in the CU), and passes the recoded

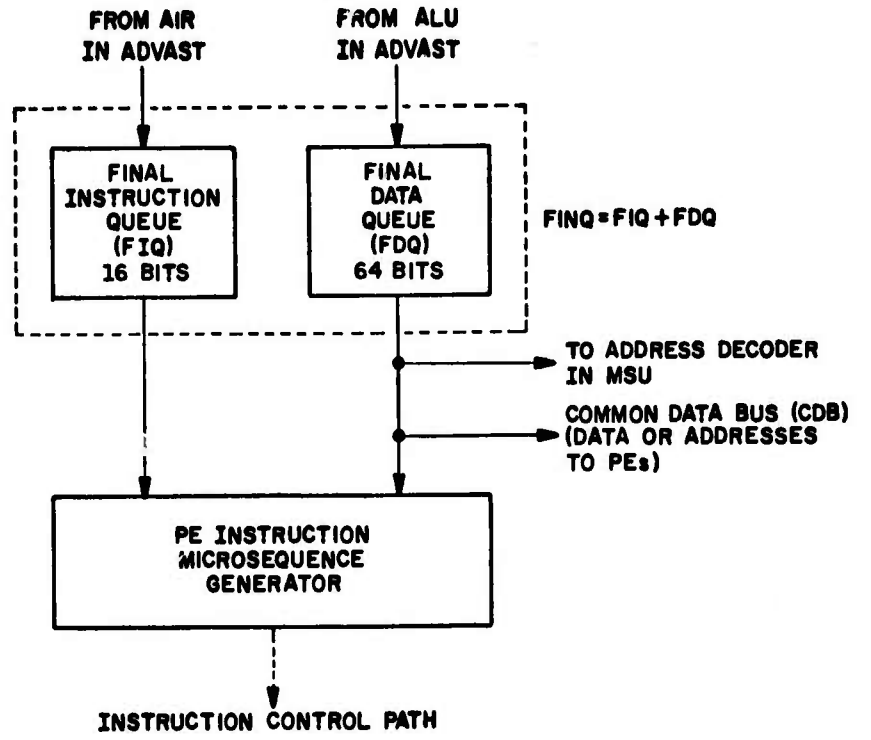


Figure II-13. Final Station (FINST) Section of the Control Unit

instruction on to FINST. Thus, some instructions may be entirely processed by ADVAST while others may pause in ADVAST only long enough for decoding before being sent to FINST for execution. To avoid situations where either ADVAST or FINST is idle waiting for the other section, the instructions are passed from ADVAST to FINST through an eight word first-in, first-out Final Queue named FINQ.* FINQ, in turn, consists of two parts. The operation code part of the instruction resides in the Final Instruction Queue (FIQ) and the address or operand is in the Final Data Queue (FDQ). FIQ consists of eight 16-bit words and FDQ of eight 64-bit words. FINQ allows FINST and ADVAST to be executing instructions concurrently.

*See Figure II-14. FINQ is a buffer as described in Chapter I which allows two processes to proceed autonomously and which effects a speedup by overlapping time.

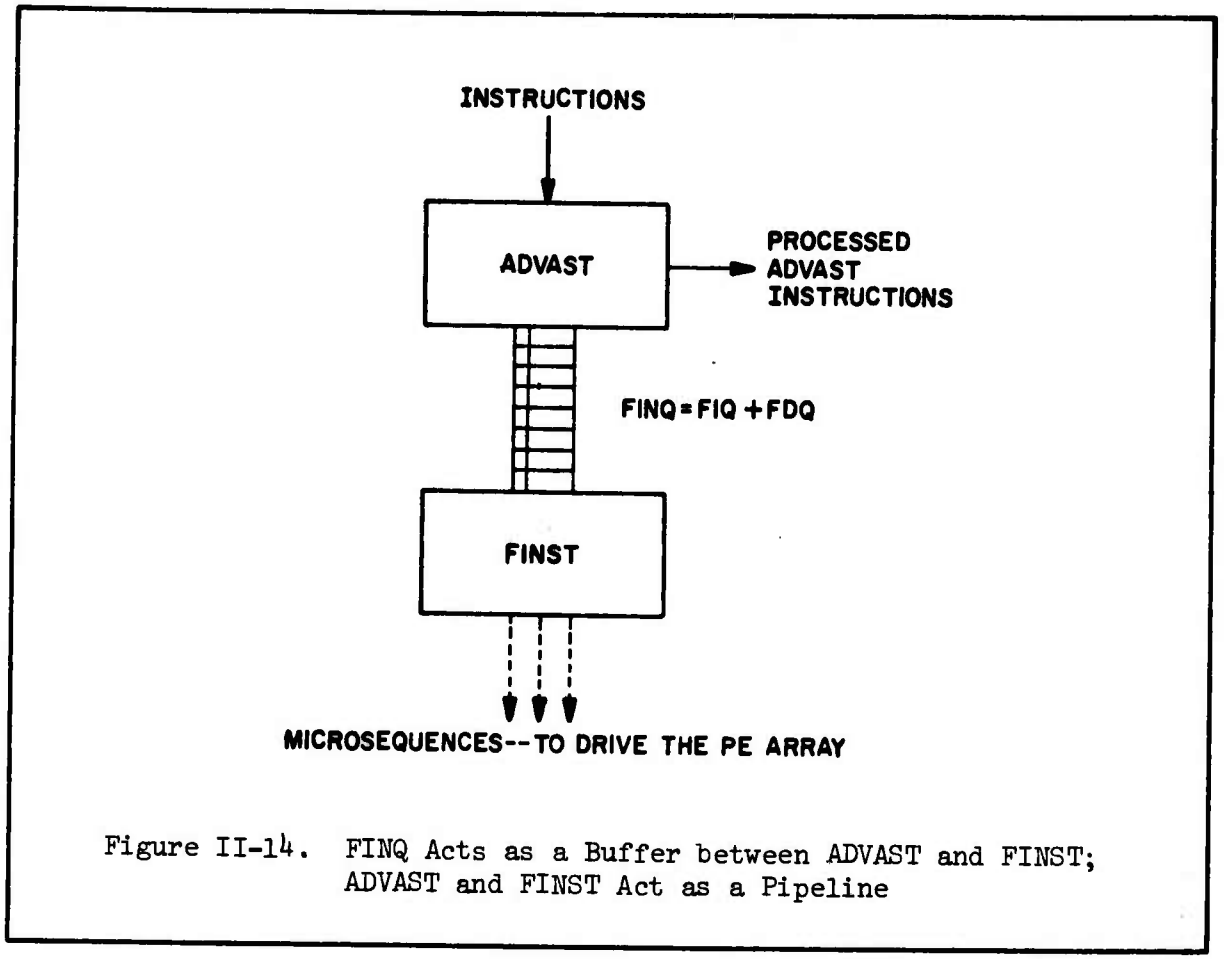


Figure II-14. FINQ Acts as a Buffer between ADVAST and FINST; ADVAST and FINST Act as a Pipeline

ADVAST and FINST, whose operation is decoupled by the holding buffer FINQ, also acts as a modified two-stage pipeline. Inputs (instructions) come into the first stage (ADVAST) and are partially processed, then they are passed on to the second stage, FINST, via FINQ if the instruction was a FINST/PE type instruction. If the instruction was an ADVAST instruction, it is completely processed in ADVAST and exits out the "side" of the pipe never making it to the second stage.

From Figure II-13, we see that two taps come off the FDQ. One of these is the CDB, already discussed in section D 1 b. The address of an

operand also goes to the MSU which controls the 64 MLUs of the 64 PUs. (The instruction to be executed may be such that an operand in FDQ wants to be stored in a location in one specific PE Memory, in which case it is the MSU's job to signal all the MLUs but the correct one to lock out writes to PE Memory.)

Occasionally a situation will arise that will stall the overlap between ADVAST and FINST: Suppose an ADVAST instruction wants to read a value from (or write a value into) an ACAR, but ahead of that ADVAST instruction, waiting in FINQ, is an instruction that FINST will cause to write a value from PE Memory (or a PE Register) into just that ACAR. Certainly the ADVAST instruction should not be executed until the instruction ahead of it in FINQ has had a chance to be fully executed. In this case, the operation of ADVAST is automatically halted (ADVAST is stalled) until FINQ drains and FINST causes the value to be written into the ACAR. After FINST has executed the last instruction in the FINQ, the operation of ADVAST continues and the ADVAST instruction which waited to read a value from (or write a value into) the ACAR is executed in its proper order. There are only four instructions which will cause ADVAST to stall.

In all other cases, FINQ makes possible an execution overlap of ADVAST and FINST/PE instructions. This overlap capability provided by FINQ makes program timing estimation difficult, since the total execution time is rarely the sum of ADVAST and FINST/PE time, although it cannot exceed this sum.

From the above discussion, it appears as if PE instructions are not really executed in the PE but in the FINST section of the Control Unit. This is partially true: The microsequences which will actually drive the PEs are set up in FINST--however, the actual "happening" of the instruction takes place within the PEs as the microsequences pulse through the appropriate PE gates. Together, ADVAST and FINST act very similarly to a pipelined instruction execution unit as described in Chapter I. ADVAST and FINST can be viewed as a two-stage pipeline operating on different instructions simultaneously. However, ADVAST instructions never reach the second stage (FINST).

At this point, the reader should refer back to Figure II-9 and note that two of the outputs from FINST (Instruction Control Path and Common Data Bus) are two of the possible inputs to a PE.

c. MSU

The Memory Service Unit (MSU) acts as an arbitrator in the various requests for access to the PEMs. PE Memory can be accessed by FINST in the execution of a PE instruction, by the ILA section of the CU which fetches the program instructions from memory (via the CU Bus), by an ADVAST instruction which fetches data from PEM (via the CU Bus), or by the I/O System. (These are the same "users" mentioned under the description of the MLU, in the PU description of section D 2 b.) The MSU in a four-quadrant ILLIAC IV consists of an address decoder and three registers which reference the other quadrants. Since we are only

considering a one-quadrant ILLIAC IV, we may view the MSU as a memory access arbitrator with only one address decoder.

The MSU controls the 64 MLUs of the PU array, locking out or allowing access to individual PEMs according to the instruction being executed (see Figure II-10).

d. TMU

The Test and Maintenance Unit (TMU) is the section of the CU that communicates with the operator's maintenance panel and display and with the I/O System. The TMU Section has two registers TRO and TRI. If Input from the I/O System is to be performed, a TMU instruction can place a request for such an action in the TRI register; if Output, then the request is placed in TRO. A hardware component of the I/O subsystem is constantly monitoring the TRO and TRI registers, waiting for an I/O request to appear. When this occurs, the I/O subsystem is interrupted, the I/O request is honored, and a response code may be placed back in TRO or TRI. The program executing on the ILLIAC IV Array can then test TRO or TRI and take an appropriate action.

e. ILA

The Instruction Look-Ahead (ILA) section is responsible for maintaining a steady flow of instructions to the ADVAST Instruction Register, AIR, in ADVAST. (See Figure II-12.) To accomplish this, ILA is arranged as shown in Figure II-15.

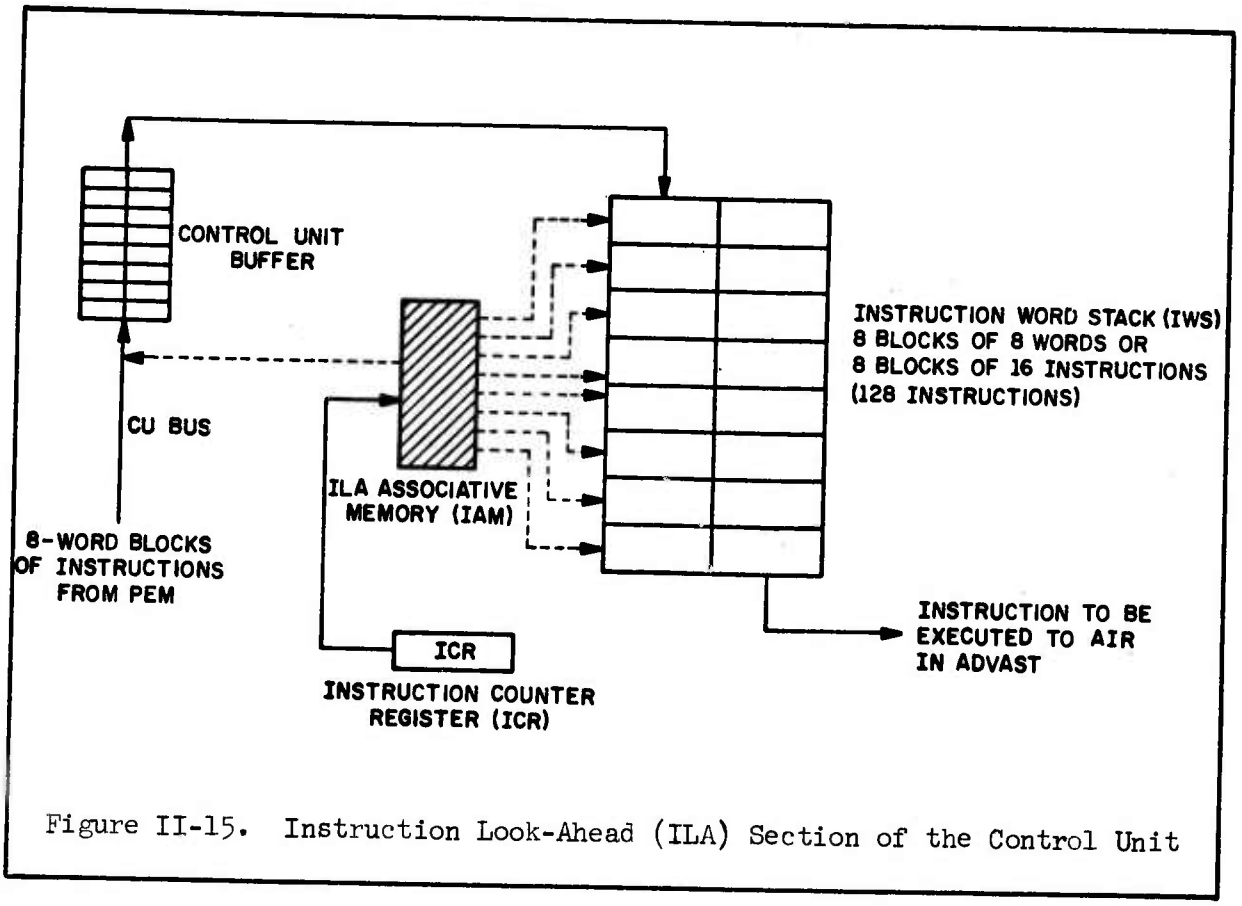


Figure II-15. Instruction Look-Ahead (ILA) Section of the Control Unit

Instructions which are stored in PE Memory are fetched to the Control Unit Buffer in blocks of 8 words via the CU Bus; since each instruction is 32 bits and there are 64 bits in a word, there are two instructions per word. These 8 word (16 instructions) blocks are relayed to the Instruction Word Stack (IWS) until it is full. The IWS holds 8 blocks of 8 words or 128 instructions.

After an instruction is sent to AIR from IWS, the contents of ICR, the Instruction Counter Register, are replaced by the proper amount. (If the previous instruction was not a branch instruction, then the contents of ICR are increased by one.) ICR then contains the location of the next instruction to be executed by AIR in ADVAST. ICR sends the

location of the next executable instruction to the ILA Associative Memory (IAM) which continuously monitors the contents of IWS. IAM is a hardware table-look-up device or "scoreboard" that can sense the locations of the instructions stored in IWS. If IAM senses that the instruction pointed to by the contents of ICR is in the IWS, then that instruction is sent on to AIR for decoding and interpretation. If the next instruction is not in IWS, then the Control Unit Buffer fetches the block of 8 words (16 instructions) from that part of PE Memory that contains the next instruction to be executed. (If the programmer can keep his program loops to within 128 machine language instructions, he can execute his program at the most efficient rate.) The Control Unit Buffer then places its block of 16 instructions over that block that has resided in IWS the longest time.

In all cases, whenever the eighth instruction in a block of 16 instructions within IWS has been executed, IAM will check IWS to see if the next block of 16 instructions is in IWS--if it is, then operation continues normally; if it is not, then the Control Unit Buffer fetches that block of 16 instructions and writes it over the block of IWS that is the oldest in time.

Both the Control Unit Buffer and IWS are buffers that smooth and speed up the instruction execution rate.

E. Another Illustrative Problem

Since ILLIAC IV is an array or vector processor, it is clear that problems involving matrix computations are ready-made for solution.

There is, however, another very large class of problems whose calculation can be performed in an "all-at-once" fashion and that is the area of Ordinary and Partial Differential Equations.

As another example of how the functional parts of ILLIAC IV can be used to solve problems, let us work through a solution of Laplace's equation describing temperature distribution on a slab. The reader who does not have a background in mathematics should not shy away from this example since the method for solution relies completely on the common sense notion that the value of any temperature on the slab tends to become the average of the surrounding temperatures.

Laplace's equation

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0$$

describes the temperature U as a function of the position (x and y) on a two-dimensional slab. That is, if we take a two-dimensional slab of material and keep the edges at certain temperatures (see Figure II-16) then, after a sufficiently long time the interior of the slab will reach a specific temperature distribution. This distribution is called the steady-state temperature distribution. The reason we talk about a temperature <u>distribution</u> is that the temperature U at any position within the interior of the slab is not constant but is a function of where it is within the slab. The temperatures on the edge of the slab are called boundary conditions and do remain constant. If we impose an x,y

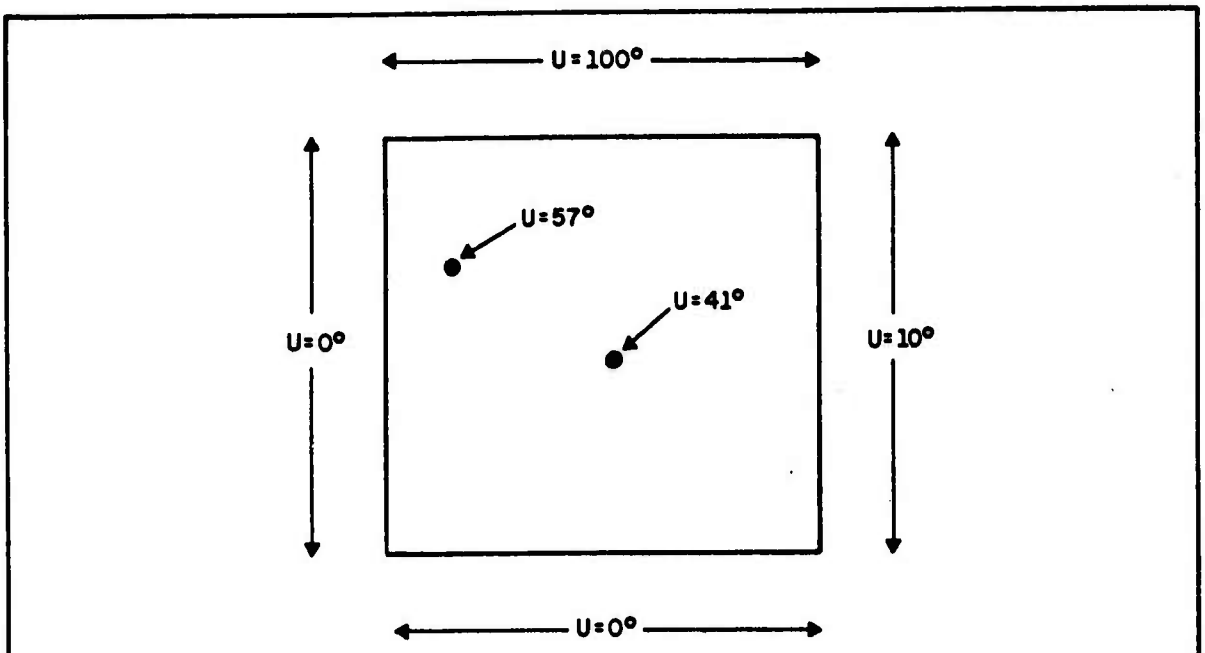


Figure II-16. Steady-State Temperature Distribution on a Slab

coordinate system over the slab we can say that the temperature at any point is a function of x and y or U = U(x,y). See Figure II-17 which assumes the slab is a units by b units. Thus every point (x,y) within the slab has associated with it a temperature U(x,y).

When we make this problem ready for solution on a digital computer we can no longer represent the temperature U as a function of the continuous variables x and y. We must discretize or digitize the problem so that instead of obtaining solutions over a continuous range for x and y, namely:

$$0 \leq x \leq a, \qquad 0 \leq y \leq b$$

we obtain solutions only on a finite set of points. See Figure II-18 where the variables x and y have been digitized every h units--we say h is

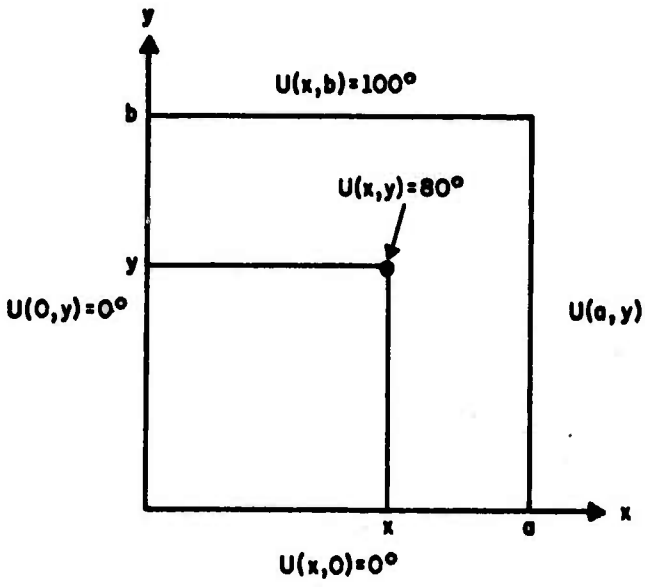


Figure II-17. Temperature on a Slab a Units by b Units is a Function of x and y

the mesh size. In Figure II-18, for simplicity, we let b = a and digitize the slab into a set of 64 x,y values or mesh points.

The method of solution for the problem may now be stated very simply: The temperature at any interior mesh point (this excludes the 28 points along the edges which must remain at constant temperatures) is the average of the temperatures of the four closest mesh points. See Figure II-19 for a blow-up picture of this property.

Thus in order to obtain a solution we apply the equation

$$(1) \qquad U(x,y) = \frac{U(x,y+h) + U(x+h,y) + U(x,y-h) + U(x-h,y)}{4}$$

to all interior points on our digitized slab until equation (1) is true. This method is called relaxation.

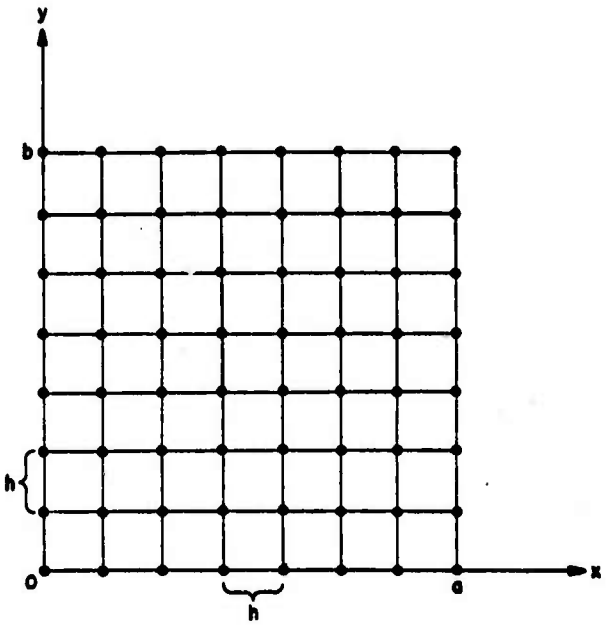


Figure II-18. Digitized Slab: 64 x,y values with a mesh size of h in the x and y directions

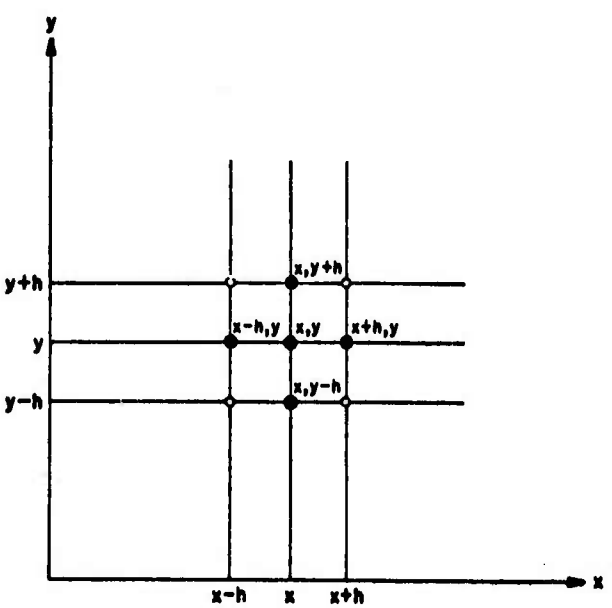


Figure II-19. Graphical Description of Solution: Temperature at any point is average of temperatures at four closest mesh points:

$$U(x,y) = \frac{U(x,y+h) + U(x+h,y) + U(x,y-h) + U(x-h,y)}{4}$$

The manner in which relaxation is usually applied on a sequential or conventional computer is to start at the top left of the digitized slab and apply equation (1) at each interior point proceeding from left to right along each "row" of points and proceeding downward row by row. Since the boundary points do not enter into the calculation, equation (1) is applied 36 times--once at each of the interior points. For the sample case of 64 points, 36 applications of equation (1) is one relaxation of the relaxation method. As enough relaxations are performed on the set of 64 mesh points, equation (1) will tend to become true (the equation will be exact within a specified error tolerance) for all of the 36 interior points. When this stage in the calculation has been reached, the steady state solution has been achieved.

There is one more change of notation that is usually applied to the problem before it is actually run on a digital computer. Since x and y have been discretized they can be viewed as indices within a two-dimensional U array. That is, x and y are merely positional indicators that can be replaced by the more familiar i and j notation of FORTRAN arrays.

Therefore if we replace x by i and y by j and further let i increase downward we can represent the mesh points as in Figure II-20.

If we use the i,j notation then equation (1) becomes

$$U_{i,j} = \frac{U_{i-1,j} + U_{i,j+1} + U_{i+1,j} + U_{i,j-1}}{4} \tag{2}$$

We apply this equation for $2 \leq i \leq 7$, $2 \leq j \leq 7$.

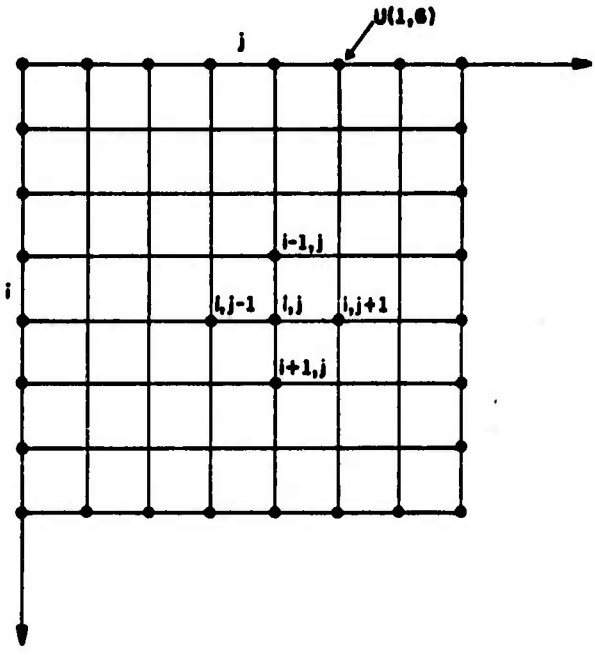


Figure II-20. Temperature as a Function of i and j

We are now ready to solve a real problem. We shall use integer values for the boundary values chosen so that the exact solution for the interior values will also be integers. These values are shown in Figure II-21. Note that the values of the boundary temperatures are constant (0°) along the bottom and right edges, but vary with position along the top and left edges. The values of the temperatures at the interior points are to be solved using equation (2)--they are initially set to 0° before the calculation begins. We will solve for the temperature distribution given the initial conditions as shown in Figure II-21 in two ways: first the sequential solution as described above will be obtained, then a method of parallel solution will be described and executed. The exact solution to the problem is shown in Figure II-22.

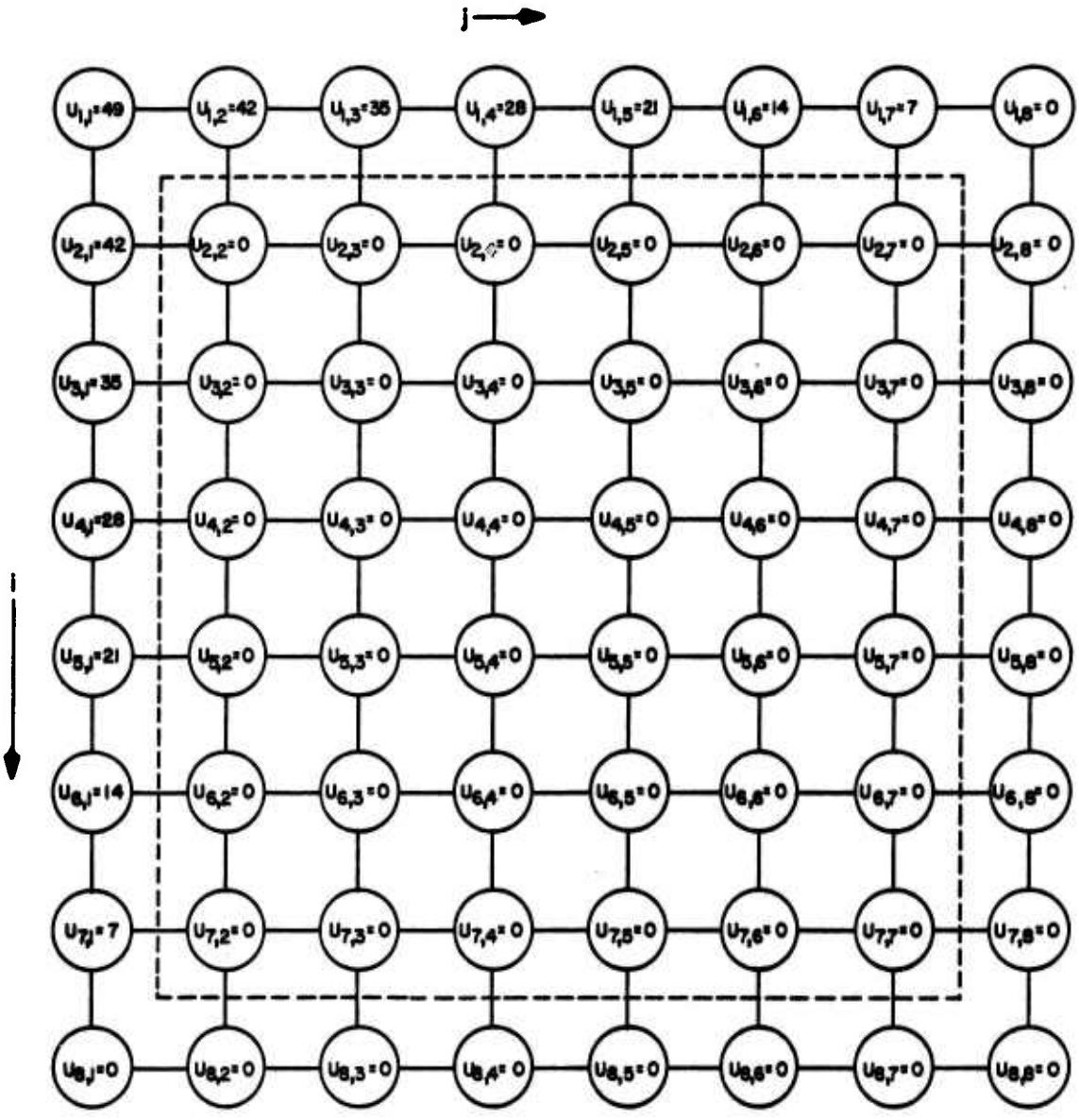


Figure II-21. Specific Interior and Boundary Conditions for Sample Problem

The boundary or edge temperatures vary with position along the left and top edges and are constant along the lower and right edges. The boundary temperatures do not change during the calculation. The interior temperatures are enclosed by dotted lines and they are initially set to zero before the calculation begins.

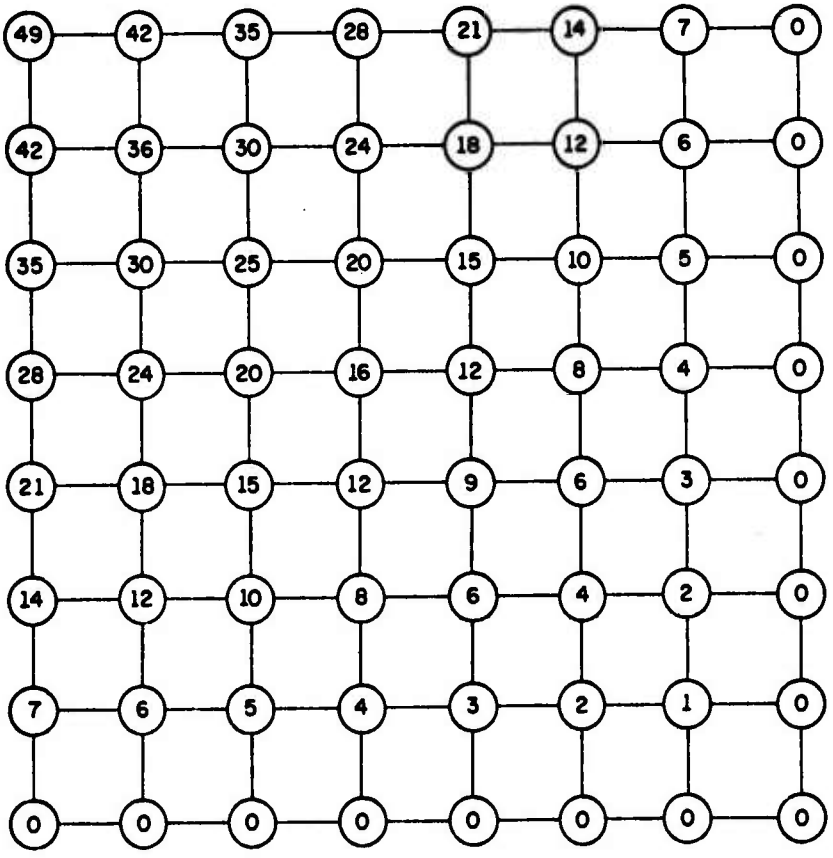


Figure II-22. Exact Solution for the Interior and Boundary Conditions given in Figure II-21

1. A Sequential Solution to the Problem

Step 1: Start at the top left interior point (i = 2, j = 2) of row 2 and calculate its new value using equation (2):

$$U_{2,2} = \frac{U_{1,2} + U_{2,3} + U_{3,2} + U_{2,1}}{4}$$

or using the numbers given in Figure II-21:

$$U_{2,2} = \frac{42 + 0 + 0 + 42}{4} = 21$$

Step 2: Moving to the right along the same row calculate the new value of $U_{2,3}$ using the new value of $U_{2,2}$ calculated in the previous step:

$$U_{2,3} = \frac{U_{1,3} + U_{2,4} + U_{3,3} + U_{2,2}}{4} = \frac{35 + 0 + 0 + 21}{4} = 14$$

(Note $U_{2,2}$ = 21 not 0 since it was previously computed as such in Step 1.)

Steps 3-6: Continuing to move to the right along row 2 we calculate the new values of $U_{2,4}$, $U_{2,5}$, $U_{2,6}$, and $U_{2,7}$ using previously computed values:

$$U_{2,4} = \frac{U_{1,4} + U_{2,5} + U_{3,4} + U_{2,3}}{4} = \frac{28 + 0 + 0 + 14}{4} = 10.5$$

$$U_{2,5} = \frac{U_{1,5} + U_{2,6} + U_{3,5} + U_{2,4}}{4} = \frac{21 + 0 + 0 + 10.5}{4} = 7.9$$

$$U_{2,6} = \frac{U_{1,6} + U_{2,7} + U_{3,6} + U_{2,5}}{4} = \frac{14 + 0 + 0 + 7.9}{4} = 5.5$$

$$U_{2,7} = \frac{U_{1,7} + U_{2,8} + U_{3,7} + U_{2,6}}{4} = \frac{7 + 0 + 0 + 5.5}{4} = 3.1$$

Steps 1 through 6 are now repeated for rows 3, 4, 5, 6 and 7. After new values have been computed for every interior point on all rows, we have finished one relaxation of the relaxation method. The values of the temperatures converge to the exact solution as shown in Figure II-22 as more and more relaxations are performed.

If we denote the value of $U_{i,j}$ at the nth relaxation as $U^n_{i,j}$ then we say a solution has been reached and we can stop the relaxation process when

$$|U^{n+1}_{i,j} - U^n_{i,j}| \leq \epsilon \qquad \text{for } 2 \leq i \leq 7, \quad 2 \leq j \leq 7 \tag{3}$$

where ϵ is our tolerance or desired degree of accuracy. Therefore, in our computer program which performs the sequential relaxation described above we save the old and new values of the U array, compare them, and if every interior temperature in the array satisfies equation (3), we end the computation. Figure II-23 shows a FORTRAN program that performs the

```
      :
C     READ IN BOUNDARY AND INITIAL VALUES FOR TEMPERATURES (U)
C     READ IN NUMBER OF ROWS (NROWS), NUMBER OF COLUMNS (NCOLS)
C     AND EPSILON CONVERGENCE VALUE (EPS)
1     M = NROWS - 1
2     N = NCOLS - 1
3     IFLAG = 0
4     DO  9  I = 2, M
5     DO  9  J = 2, N
6     TEMP = (U(I,J+1) + U(I-1,J) + U(I,J-1) + U(I+1,J)) * 0.25
7     IF (ABS(TEMP - U(I,J)) . LE . EPS) GO TO 9
8     IFLAG = 1
9     U(I,J) = TEMP
10    IF(IFLAG . EQ . 1) GO TO 3
11    END
```

Figure II-23. A FORTRAN Program for a Sequential Solution to the Sample Problem

relaxation algorithm described above and Figure II-24 shows the values of the U array after one, ten and fifty relaxations; the exact solution is also shown in Figure II-22.

Let us briefly consider the FORTRAN program as shown in Figure II-23:

The three COMMENT statements at the beginning of the program indicate that the initial values of the temperature U (see Figure II-21),

One Relaxation

49	42	35	28	21	14	7	0
42	21.00	14.00	10.50	7.88	5.47	3.12	0
35	14.00	7.00	4.38	3.06	2.13	1.31	0
28	10.50	4.38	2.19	1.31	0.86	0.54	0
21	7.88	3.06	1.31	0.66	0.38	0.23	0
14	5.47	2.13	0.86	0.38	0.19	0.11	0
7	3.12	1.31	0.54	0.23	0.11	0.05	0
0	0	0	0	0	0	0	0

Ten Relaxations

49	42	35	28	21	14	7	0
42	35.37	29.01	22.90	17.03	11.31	5.66	0
35	29.01	23.41	18.24	13.44	8.88	4.44	0
28	22.90	18.24	14.05	10.26	6.75	3.38	0
21	17.03	13.44	10.26	7.44	4.88	2.44	0
14	11.31	8.88	6.75	4.88	3.19	1.60	0
7	5.66	4.44	3.38	2.44	1.60	0.80	0
0	0	0	0	0	0	0	0

Fifty Relaxations

49	42	35	28	21	14	7	0
42	36.00	30.00	24.00	18.00	12.00	6.00	0
35	30.00	25.00	20.00	15.00	10.00	5.00	0
28	24.00	20.00	16.00	12.00	8.00	4.00	0
21	18.00	15.00	12.00	9.00	6.00	3.00	0
14	12.00	10.00	8.00	6.00	4.00	2.00	0
7	6.00	5.00	4.00	3.00	2.00	1.00	0
0	0	0	0	0	0	0	0

Figure II-24. Values of the Temperature after One, Ten, and Fifty Relaxations using Sequential Method

the number of rows, NROWS (for our case NROWS = 8), the number of columns, NCOLS (for our case NCOLS = 8), and the epsilon convergence value, EPS have all been read in through the appropriate input statements.

Statements 1 and 2 compute values for M and N to be one less than the number of rows and columns respectively, and the calculation starts at 2 not 1 since the edge values will not change throughout the computation.

At statement 3 a flag, IFLAG, is set to zero. IFLAG will act as a signal to the program indicating whether convergence has been reached after each relaxation (each relaxation consists of 36 applications of equation (2)): If IFLAG is still zero after a relaxation then all of the U values are within epsilon of their previous value; if IFLAG has been set to one, then at least one U value was not within the convergence criterion and another relaxation must be made.

Statements 4 and 5 initialize the DO LOOP counters I and J that step us through the rows and columns starting at the top left and proceeding to bottom right.

Statement 6 is equation (2).

Statement 7 is equation (3). If the statement is true (TEMP is within epsilon (EPS) of the last value of U), IFLAG is not changed and control jumps to Statement 9 where U assumes its new value of TEMP. If the statement is false (TEMP is <u>not</u> within epsilon of the last value of U)

IFLAG is set to 1 and control falls to Statement 9, where U assumes its new value of TEMP.

Statement 8, if control reaches it, sets the value of IFLAG equal to 1.

Statement 9 replaces the old value of U with TEMP--the new value of U.

Statement 10 tests IFLAG. If it is true (IFLAG = 1) then at least one value of U has not yet reached convergence and control is passed to Statement 3 where IFLAG is re-initialized back to zero. If it is false (IFLAG ≠ 1), then IFLAG must be equal to zero and there exists no U that was not within epsilon of convergence and therefore convergence has been attained. Control then drops to Statement 11.

Statement 11 is reached only when convergence has been reached and the program ends.

This program is still very primitive; it makes no allowance for the possibility that an overly stringent choice for EPS might result in an infinite amount of looping between Statements 10 and 3, but it illustrates a sequential solution to our sample problem.

2. A Parallel Solution to the Problem

Let us next consider how this same problem could be solved in parallel on ILLIAC IV:

If each value for U were placed in a separate Processing Element Memory or register, then the calculation of equations (2) and (3) could proceed in parallel for all 36 inner values in the U array. A program could be written to compute new values for $U_{i,j}$ $2 \leq i \leq 7$, $2 \leq j \leq 7$, not from top left to bottom right but <u>all at once</u>. As we did with the sequential solution let us write down the steps for a parallel solution:

<u>Step 1</u>: Assume the initial conditions are as shown in Figure II-21.

<u>Step 2</u>: Disable all edge or border PEs. (These PEs contain the boundary values for U and must not change during the calculation.)

<u>Step 3</u>: Simultaneously calculate:

$$(2) \quad U_{i,j} = \frac{U_{i-1,j} + U_{i,j+1} + U_{i+1,j} + U_{i,j-1}}{4}$$

for $2 < i \leq 7$, $2 < j \leq 7$.

Rather than write out the values for all 36 interior points, let us just look at the interior points of the second row ($i = 2$, $j = 2, 3, 4, 5, 6, 7$) after equation (2) is applied simultaneously to all 36 interior points:

$$U_{2,2} = \frac{U_{1,2} + U_{2,3} + U_{3,2} + U_{2,1}}{4} = \frac{42 + 0 + 0 + 42}{4} = 21$$

$$U_{2,3} = \frac{U_{1,3} + U_{2,4} + U_{3,3} + U_{2,2}}{4} = \frac{35 + 0 + 0 + 0}{4} = 8.8$$

$$U_{2,4} = \frac{U_{1,4} + U_{2,5} + U_{3,4} + U_{2,3}}{4} = \frac{28 + 0 + 0 + 0}{4} = 7$$

$$U_{2,5} = \frac{U_{1,5} + U_{2,6} + U_{3,5} + U_{2,4}}{4} = \frac{21 + 0 + 0 + 0}{4} = 5.3$$

$$U_{2,6} = \frac{U_{1,6} + U_{2,7} + U_{3,6} + U_{2,5}}{4} = \frac{14 + 0 + 0 + 0}{4} = 3.5$$

$$U_{2,7} = \frac{U_{1,7} + U_{2,8} + U_{3,7} + U_{2,6}}{4} = \frac{7 + 0 + 0 + 0}{4} = 1.8$$

Note that the value for $U_{2,2} = 21$ was not used in calculating $U_{2,3}$ because $U_{2,2}$ and $U_{2,3}$ were calculated <u>at the same time</u> and a new value for $U_{2,2}$ is not ready until all of the 36 values for U have been calculated.

<u>Step 4</u>: Repeat Step 3 until convergence is satisfied.

Figure II-25 shows values of the temperature U after one, ten, and fifty relaxations using this parallel method of solution.

Not only are the two algorithms different, but the <u>way</u> the temperatures converge is also different (as can be seen by comparison of Figures II-24 and II-25), although the end result approaches the same steady-state temperature distribution. When we use the sequential method of sweeping from left to right along rows and proceeding from the top to bottom row, the temperatures at the bottom right converge faster to the exact solution than those at the top left. This type of convergence

occurs because in sweeping from top left to bottom right we always use more of the data we just computed as we reach the end of the sweep, i.e., the bottom right. The computations at the bottom right contain more new information since they are computed at the end of the sequence of calculations.

When we use the parallel algorithm of computing a set of new values at one crack, the values closest to the boundary (the edge values)

One Relaxation

49	42	35	28	21	14	7	0
42	21.	8.75	7.00	5.25	3.50	1.75	0
35	8.75	0	0	0	0	0	0
28	7.00	0	0	0	0	0	0
21	5.25	0	0	0	0	0	0
14	3.50	0	0	0	0	0	0
7	1.75	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Ten Relaxations

49	42	35	28	21	14	7	0
42	34.3	27.1	20.5	14.8	9.6	4.7	0
36	27.1	19.9	14.1	9.5	5.9	2.8	0
28	20.5	14.1	9.1	5.6	3.2	1.5	0
21	14.8	9.5	5.6	3.1	1.6	0.7	0
14	9.6	5.9	3.2	1.6	0.7	0.3	0
7	4.7	2.8	1.5	0.7	0.3	0.1	0
0	0	0	0	0	0	0	0

Fifty Relaxations

49	42	35	28	21	14	7	0
42	35.98	29.96	23.96	17.96	11.96	5.98	0
35	29.96	24.94	19.92	14.92	9.94	4.96	0
28	23.96	19.92	15.90	11.90	7.92	3.96	0
21	17.96	14.92	11.90	8.90	5.92	2.96	0
14	11.96	9.94	7.92	5.92	3.94	1.96	0
7	5.98	4.96	3.96	2.96	1.96	0.98	0
0	0	0	0	0	0	0	0

Figure II-25. Values of the Temperature after One, Ten and Fifty Relaxations using Parallel Method

converge faster than the values in the center of the mesh. This type of convergence occurs because the outer values are closest to the boundary values and have more new data to use sooner than the inner values. Since a relaxation consists of 36 computations done at once, the inner values do not get to use previously computed values until several relaxations have been performed. After each relaxation more inner values have more new data to use to compute their next value.

If we liken the convergence process to freezing, we can say that the sequential algorithm begins freezing at the bottom right and proceeds to the top left; the parallel algorithm begins freezing around the edges and proceeds towards the center.

The savings in time of the parallel method over the sequential one is dependent upon the number of relaxations necessary to produce convergence. If the same number of relaxations to convergence are necessary for both the sequential and parallel algorithms and each processes P interior values, then the parallel process is faster by a factor of P. However, since the parallel algorithm uses less new information for each relaxation, it may take more parallel relaxations (which consist of one application of equation (2)) to produce the same degree of accuracy as a sequential relaxation (which consists of 36 applications of equation (2)). That is, if a solution can be reached in 10 sequential relaxations, it could take more than 10 parallel relaxations to reach a solution of the same accuracy. On ILLIAC IV, though, the parallel relaxation is 36 times as fast as a sequential one and this speedup far

outweighs the few extra relaxations necessary for equal accuracy in the solution (for our particular sample problem).

F. Some Data Allocation Considerations

If we divide our slab into an 8 x 8 array of mesh points to solve Laplace's equation governing heat distribution on a slab, the data allocation scheme to be used on ILLIAC IV is straightforward--one value of U can be assigned to each PEM (see upper portion of Figure II-26).

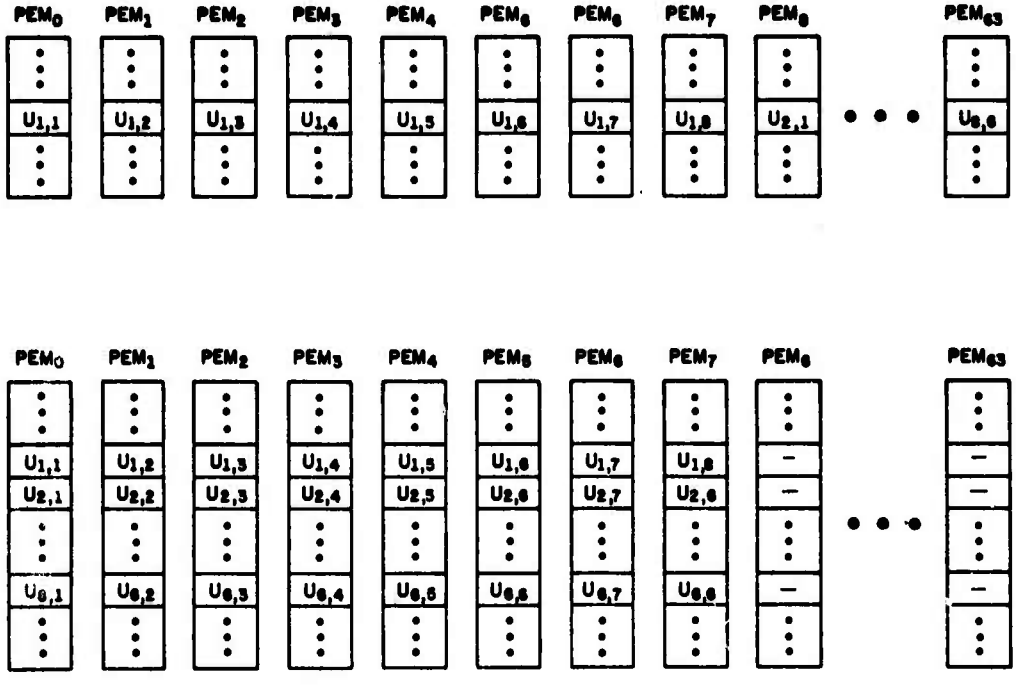


Figure II-26. Comparison of Storage: Upper portion shows 8 x 8 U array stored one value per PEM, Lower portion shows 8 x 8 U array stored 8 values per PEM

Another possible data allocation scheme is shown in the lower portion of Figure II-26. This scheme allocates 8 values of U per PEM. Although this lower scheme is not as efficient in terms of execution time necessary to solve the problem, it is very similar to the way a 64 x 64 U array would be stored. The program for the two data allocation schemes shown in Figure II-26 will be developed in Chapter III and it will be seen that these programs are substantially different--indicating that the form of an ILLIAC IV program is highly dependent on the data allocation scheme chosen.

Let us now consider what to do if we wish to impose a finer grid over the slab. There is certainly no value in using a 50 x 50 grid if we can use a 50 x 64 just as cheaply. That is, since ILLIAC IV has exactly 64 PEs, this physical fact might as well be capitalized upon when choosing a mesh size. If the mesh size is arbitrary within defined limits, the user should choose the closest multiple of 64--it costs no more and may actually speed up the calculation, since less code will be generated.

Suppose we have divided our slab into a 64 x 64 set of mesh points. We could store these 4096 values of $U_{i,j}$ as shown in Figure II-27. Each PEM holds 64 values of U. For this case, one parallel calculation of equation (2) could process only one <u>row</u> at a time. Since the border values do not enter into the calculation, 62 parallel calculations of equation (2) would have to be performed to effect one relaxation of the entire U array. This is a speedup of 62 over the 3844 calculations necessary for one sequential relaxation of a 64 x 64 U array.

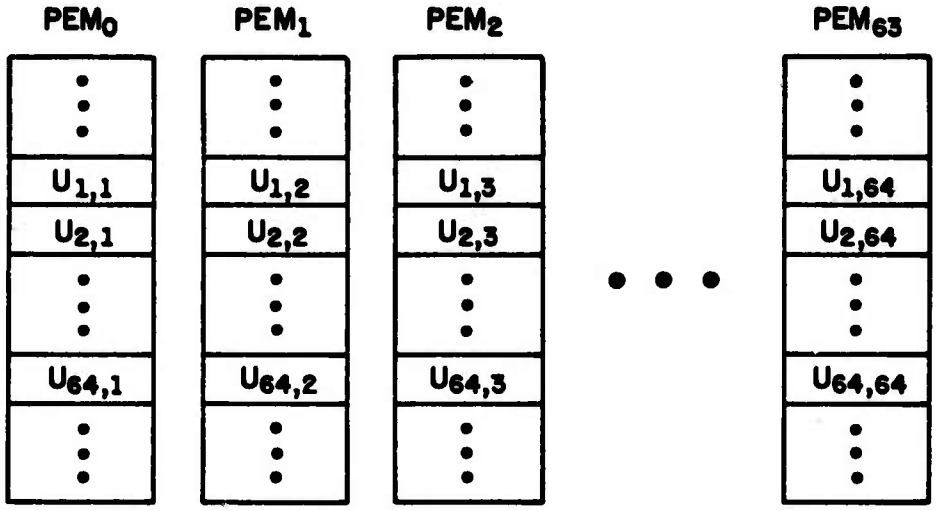


Figure II-27. Storage Allocation of U Array for 64 x 64 Set of Mesh Points

It is very important for the reader to understand that the ILLIAC IV program to solve a problem is very much dependent on the data allocation scheme chosen by the programmer. This is true also, but to a lesser extent, for a conventional sequential computer.

G. ILLIAC IV Input/Output (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) reside within the ILLIAC IV I/O System. The ILLIAC IV I/O System consists of the I/O Subsystem, a Disk File System (DFS) and a B6500

Control Computer (which in turn supervises a large Laser Memory, a Data Communications Processor, and the ARPA Network Link). See Figure II-28. The total ILLIAC IV System consisting of the ILLIAC IV I/O System and the ILLIAC IV Array is shown in Figure II-29. The reader is warned that all system configurations shown are transitory, and more than likely will have changed several times before this book is published.

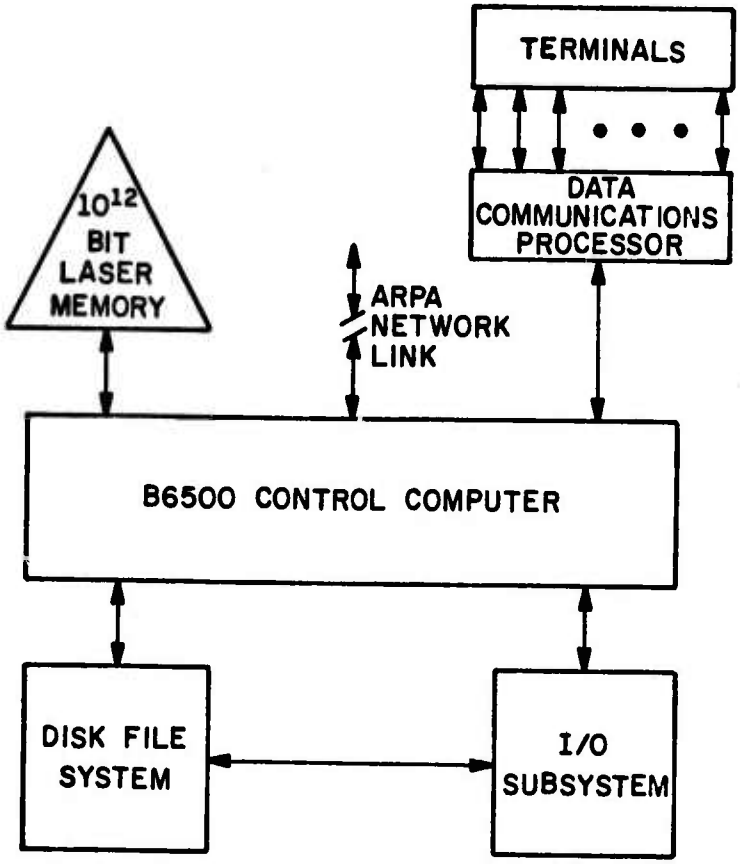


Figure II-28. ILLIAC IV I/O System

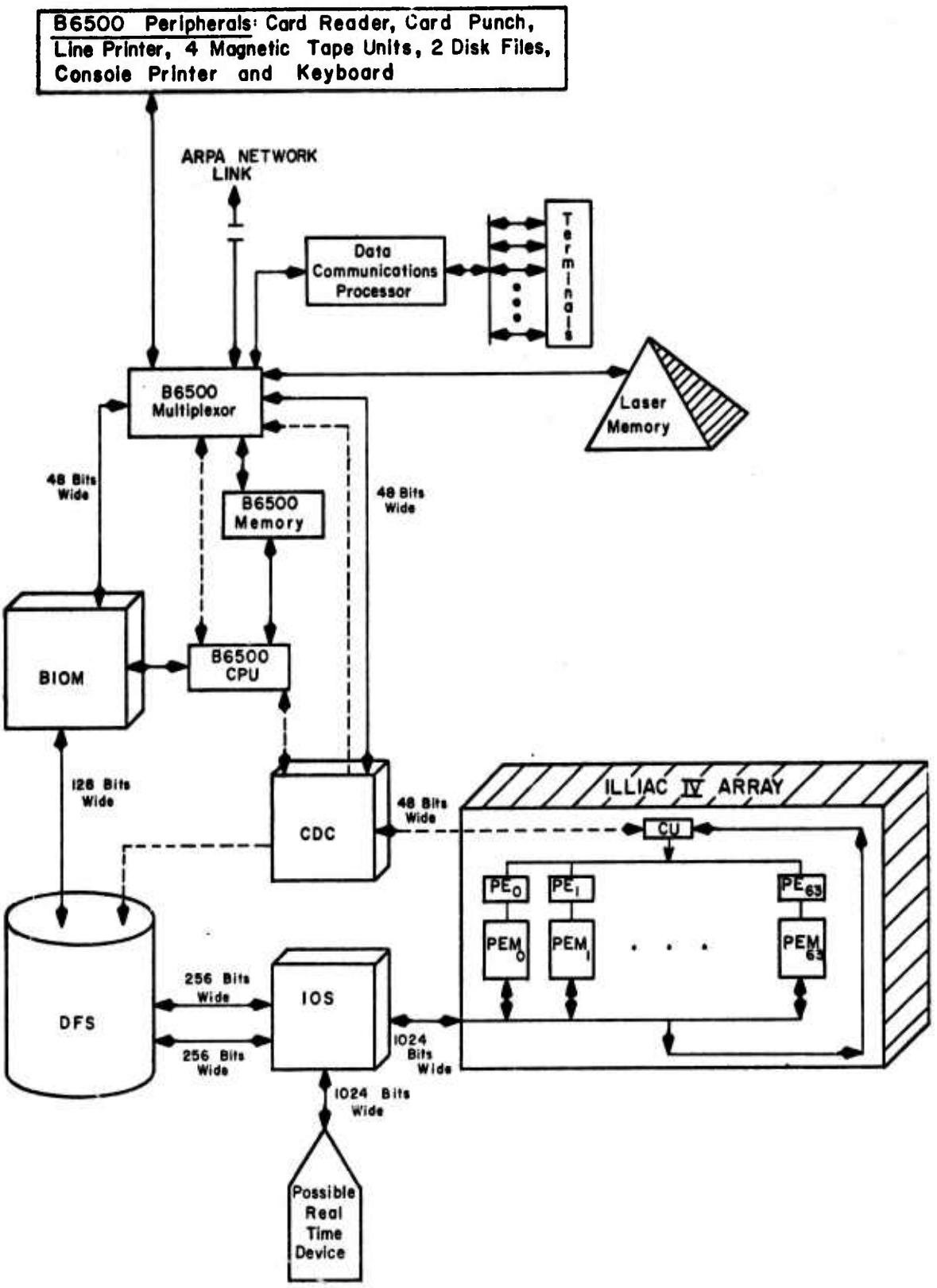


Figure II-29. ILLIAC IV System

1. I/O Subsystem

The I/O Subsystem consists of the Control Descriptor Controller (CDC), the Buffer Input/Output Memory (BIOM) and the Input/Output Switch (IOS).

a. Control Descriptor Controller (CDC)

The CDC is that component of the I/O Subsystem (mentioned in section D 3 d) which monitors the TMU 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 the TMU 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 PE Memory Array with programs and data from the ILLIAC IV Disk (also called the Disk File System or DFS). After PE Memory has been loaded, the CDC can then pass control to the CU to begin execution of the ILLIAC IV Program.

b. Buffer Input/Output Memory (BIOM)

The B6500 Control Computer can transfer information from its memory through its CPU at the rate of 80×10^6 bits/second. The ILLIAC IV Disk (DFS) accepts information at the rate of 500×10^6 bits/second. 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 Disk File System. See Figure II-29. The BIOM is actually four PE memories providing 8192 words of 64-bit storage.

In addition to the data link to the B6500 CPU, the BIOM is also connected to the B6500 Multiplexor which, in turn, is linked to the B6500 Peripheral set. A typical path for a user's program and data might be: Magnetic Tape through the B6500 Multiplexor to BIOM to ILLIAC IV Disk to IOS to the PE Memory Array.

c. Input/Output Switch (IOS)

The IOS performs two functions. As its name implies, it is a switch and is responsible for switching information from either the Disk File System or from a port which can accept input from a real time device. All bulk data transfers to and from the PE Memory Array are via IOS. As a switch it must insure that only one input is sending to the Array at a given time. In addition, the IOS acts as a buffer between the Disk File System 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 PE Memory Array is 1024 bits wide.

2. Disk File System (DFS)

The Disk File System (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 $.5 \times 10^9$ bits/second over a path 256 bits wide; however, if both channels are sending or receiving simultaneously the transfer rate is 10^9 bits/second.

The Disk revolves once every 40 milliseconds and thus has an average access time of 20 milliseconds. Processing of I/O requests to the Disk is enhanced by the operation of the Disk Queuer hardware. The Disk Queuer can store up to 24 I/O requests in a hardware table. This table is constantly monitored as the disk spins under its heads. If a block of I/O comes under a head that is referenced in the Queuer table--regardless of its position in the table--then that block is transferred as per the request.* As an example see Figure II-30.

The DFS has data paths to the Array via the IOS and with the B6500 via the BIOM; there is also a control path from the CDC to the DFS which is used in the last stages of initiating Disk to Array transfers of programs or data.

---

*Data transfer to and from the Disk can also be effected in the conventional first-come, first-serve manner when so specified by the programmer.

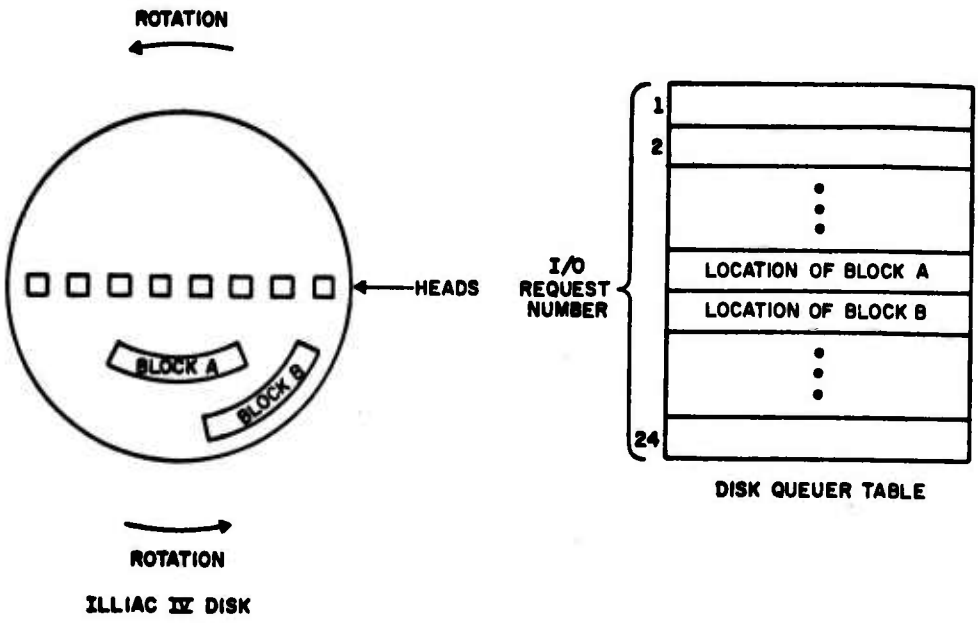


Figure II-30. Example of Disk Queuer Function

Even though the I/O request for Block A was entered into the Table before the I/O request for Block B, the request using Block B will be performed first in the above situation since it will pass under the Disk heads sooner than Block A.

3. B6500 Control Computer

The B6500 Control Computer consists of a Central Processing Unit (CPU), Memory, a Multiplexor and a set of Peripheral Devices (Card Reader, Card Punch, Line Printer, 4 Magnetic Tape Units, 2 Disk Files and Console Printer and 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. Managing requests includes scheduling and eventually instituting the process which utilizes the resource.

All compiling and assembling of programs is 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.

a. B6500 Central Processing Unit (CPU)

The B6500 CPU provides the Control and the Arithmetic and Logical processing capability to the B6500 Control Computer. The B6500 CPU operates at 5 megacycles.

b. B6500 Memory

The B6500 memory contains 65,536 48-bit words and has a memory cycle time of 1.2 μsecs. The B6500 Memory can be considered tertiary memory (over 3 million bits) in the total system, with the ILLIAC IV Disk being secondary memory (one billion bits) and the PEMs of the ILLIAC IV Array being primary memory (over 8 million bits).

c. B6500 Multiplexor

The B6500 Multiplexor is the heart of the B6500 I/O System as can be seen by the number of lines coming into and going out of it in Figure II-29. It is linked to the BIOM, the B6500 CPU, B6500 Memory,

the CDC, and the B6500 Peripheral Set as well as the ARPA Network Link, the Data Communications Processor and the Laser Memory. The Multiplexor may be viewed as a switching network or a small I/O control computer which operates asynchronously with the B6500 CPU.

d. B6500 Peripherals

The B6500 Control Computer has a standard set of input and output peripheral devices:

1 Card Reader (800 cards per minute)

1 Card Punch (300 cards per minute)

1 Line Printer (132 print positions; 1100 lines per minute)

4 Magnetic Tape Units (9 channel; 1600 bits per inch; 45 inches per second)

2 Disk Files (5 million 48-bit words, 20 milliseconds average access time per disk file)

1 Console Printer and Keyboard.

e. Data Communications Processor

The Data Communications Processor will supervise a set of remote terminals. The terminals are devices such as Teletypes or CRT displays that allow the user access to the ILLIAC IV System. Users will be able to enter their jobs into the system in either a batch or interactive mode via the terminals. The terminals can also be used to monitor jobs while in execution and to scan the ILLIAC IV Disk or Laser Memory which will contain

the output from a job. If the user decides he needs a hard copy of his output, he can then signal the system from his terminal to activate a printer.

f. Laser Memory

The B6500 supervises a 10^{12}-bit 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 five seconds. Within the same strip data can be located in 200 milliseconds. The read and record rate is four million bits a 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. The laser memory can be considered fourth-order memory in the ILLIAC IV System (one trillion bits).

g. ARPA Network Link

The ARPA Network is a group of computer installations separated geographically but connected by high speed (50,000 bits/second) 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 IMP (Interface Message Processor) 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. See Reference 2 for a complete description of the ARPA Network.

H. Conclusions and Opinions

It is useful to view the ILLIAC IV System as a set of resources; each member of the set having special capabilities. If the programmer can define his problem in terms of the unique capabilities of this set of resources he has effected a computer solution to his problem.

The set of resources afforded by the ILLIAC IV System is:

- A B6500 Computer System which also supervises
  - A Laser Memory
  - The ARPA Network
  - A Terminal System
- A very fast Disk Storage System
- An extremely fast Array Processor.

If the problem to be solved on the ILLIAC IV System involves vector manipulation or systems of differential equations to be approximated by finite difference schemes, then the Array Processor resource can be utilized. Many applications in Numerical Weather Prediction, Linear Programming, Hydrodynamics, Signal Processing and the response of coupled mechanical and electrical systems are in this category.

Information Storage and Retrieval processes can be performed using the B6500, the ILLIAC IV Disk and the huge Laser Memory. Large data bases can be accumulated on the large slow Laser Memory and can be sent via the ILLIAC Disk to either the B6500 or the ILLIAC IV Array for processing.

Computer Aided Instruction (CAI) is another area of application that can be exploited using the large Laser Store coupled to the terminals. A remote file editing capability including interactive compiling facilities and job monitoring during execution is afforded via the terminals. Programmer convenience is further enhanced by possible debugging systems using the terminals. Also, the B6500 can provide a full range of compilers, assemblers and utilities for programming support. At present, no one is sure how to use ILLIAC IV to assemble or compile programs.

As mentioned previously, the ARPA network represents a potential broad range of resources within itself.

A satisfying Artificial Intelligence model has not yet been produced by any available computer resource. Hopefully the 10^{12} bit

Laser Memory will encourage a grander, more comprehensive approach to intelligence models.

Perhaps it is too glib to say that the ILLIAC IV System is a set of resources; if the user is resourceful, he can use them. Unfortunately the statement is partly true--there are no recipes yet for solving the large, the important and interesting problems. There is no computer science out on the edge of research. The answer to the question "How do you use ILLIAC IV?" cannot be answered by a statement, but with another question:

You know the resources, the tools you have to work with, the next time a problem passes close by, ask yourself "Can this problem be described in terms of the resources which are now available to me?"

CHAPTER II -- REFERENCES

1. ILLIAC IV Systems Characteristics and Programming Manual, Paoli, Pa.: Burroughs Corporation (IL4-PM1 Revised), (30 June 1970).

2. Lawrence G. Roberts and Barry D. Wessler, "Computer Network Development to Achieve Resource Sharing," 1970 SJCC, AFIPS Proceedings, Vol. 36, pp. 543-549.

References 3 and 4 are excellent additional material to augment the reader's understanding of ILLIAC IV. He is warned, however, that many aspects of the ILLIAC IV Computer have changed since these two papers were written.

3. G. H. Barnes, et al., "The ILLIAC IV Computer," IEEE Transactions on Computers, August 1968, Vol. C-17, pp. 746-757.

4. D. J. Kuck, "ILLIAC IV Software and Application Programming," IEEE Transactions on Computers, August 1968, Vol. C-17, pp. 758-770.

CHAPTER III -- THE ASSEMBLY LANGUAGE--ASK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER III

THE ASSEMBLY LANGUAGE -- ASK

A. Summary

The ILLIAC IV Assembler is called ASK. It is a two-pass assembler that accepts a program written in the ASK language and converts it to an ILLIAC IV binary object code. Although there are almost 300 instructions in the ILLIAC IV repertoire only those few ASK instructions required to write simple programs will be described here.

The approach taken in this chapter will be to state a problem, then to learn only those instructions necessary for the solution of that problem. The first problem is that of summing an array of numbers. After the instructions for the solution of the first problem are learned only a few more are needed to solve the second problem: finding the largest value in an array of numbers. Problem three describes a parallel algorithm for matrix multiplication that differs from the standard sequential algorithm. The ASK instructions to implement this parallel algorithm are then developed. The fourth problem, transposing a matrix, is used to develop the concept of skewed storage which is one solution to the problem of accessing the columns of a matrix as efficiently as the rows. The fifth and final problem develops the ASK instructions necessary to solve Laplace's equation which models the steady-state temperature distribution on a slab as discussed in Chapter II. Two cases are considered illustrating two possible data allocations; the first case allocates the

64 temperatures so that one temperature is stored in one PEM, the second allocates eight temperatures per PEM and is representative of all allocations requiring a finer mesh spacing up to and including a 64 x 64 mesh. After the five problems have been programmed, the programmer has learned 40 ASK instructions and some useful programming techniques.

B. Review

Let us review the registers that are programmable:

In the CU, we have four 64-bit registers named ACAR0, ACAR1, ACAR2, and ACAR3. There is also a 64-word scratch pad memory called the ADB. Each word is 64 bits long.

In each PE, we have six programmable registers:

RGA, the A register and Accumulator, is 64 bits
RGB, the B register is 64 bits
RGR, the R register or routing register is 64 bits
RGS, the S register or temporary storage register is 64 bits
RGX, the X register or index register is 16 bits
RGD, the mode register is 8 bits.

There are two basic types of instructions, FINST/PE and ADVAST. ADVAST instructions are executed in the ADVAST section of the Control Unit (CU) and are of the type that can be fulfilled within the resources of the CU. FINST/PE instructions may be partially processed within ADVAST but

to be fully executed they are sent on to FINST which sends out the micro-sequences necessary to drive the PE array. Since there exists other literature which refers to FINST/PE instructions as PE instructions and to ADVAST instructions as CU instructions, that naming will be used alternatively in this chapter.

C. Notation

ASK instructions will be described using the following format:

Label: Opcode Operand;

Metalinguistic symbols or symbols which stand for other symbols are written in script, i.e. *Label*. A value that the variable symbol *Label* may be is LOC or LOOP. Symbols which stand only for themselves (reserved symbols) are written in upper case type and are underlined, for example:

STA *Operand;*

(STA is a possible value for *Opcode*.) Reserved symbols will usually be Operation Codes or Opcodes and must be written exactly as they appear.

Symbols which are partly variable and partly reserved will have the reserved part written in upper case type and will be underlined, and the variable part will be denoted by a lower case Greek letter, for example,

LDα *Operand;*

where α can be A, B, S, X, R or D. This means that LDA, LDB, LDS, LDX, LDR, or LDD are all possible variations for the LDα operation code.

The *Label* field is optional and represents a symbolic location or address within the program; *Label* must be followed by a colon (:).

The *Opcode* is the operation code portion of the instruction, i.e., ADD is an *Opcode*. A blank must follow the *Opcode*.

The *Operand* is the address or operand portion of the instruction and may be an address, a count, or data. If it is an address, it denotes the location in memory where data resides.

A semicolon (;) must follow the *Operand* and indicates the end of an ASK instruction.

D. Operands

With each instruction type, PE and CU, there is associated a permissible set of operands.

1. PE Operands

A PE Operand may be a PE Memory Address, PE Register, Literal, ACAR, Routing Operand, or Mode Setting Operand:

a. PE Memory Address

A PE Memory Address (PEM Address) refers to the contents of a Row of PE Memory locations. The name of a PEM Address must be symbolic

and is created by the programmer. Symbolic names consist of alphameric characters (letters and digits); they can be up to 63 characters in length but the first character must be a letter.

Examples of PEM addresses:

X

LOCATION

L123

A PEM Address will alternatively be called a "Row", "Row location", "PE Memory location", "PEM location", or "PEM Row location" in the following text. Note also that PEM is short for PE Memory.

b. PE Register

A PE Register has the following format:

$ *Register Name*

where *Register Name* can be A, B, R, S, or X and

$A stands for RGA

$B stands for RGB

$R stands for RGR

$S stands for RGS

$X stands for RGX

(The total mode register, RGD, can never be a PE operand, except in one special case which will not be covered here.)

c. Literal

A Literal is usually a value that stands for itself, not a location where the value can be found. It is of the form:

= *Literal*

where *Literal* (for the purposes of this brief explanation) is a number, i.e.,

= 12:8 is a Literal of value twelve in the base 8 number system (and is equal to ten in base 10).

= 10 is an integer ten in the base ten number system.

PE Literals are constrained to be representable within 16 bits so that floating point numbers (which require 32 or 64 bits) are not allowed as PE Literals. Integers which can be represented within 16 bits are allowed however:

= 0

= 15

= 65535

are all valid integers which can be used as PE Literals.

d. ACAR

Only one of the four ACARs in the CU can be used as a PE Operand; they are referenced by the names $C0, $C1, $C2, and $C3. (The

contents of the specified ACAR is transmitted or "broadcast" to all PEs via the Common Data Bus.)

e. Routing Operand

A Routing Operand is a highly specialized type of Operand that is only used with one PE instruction (the Route instruction, RTL). This Operand will be discussed when the Route instruction is discussed.

f. Mode Setting Operand

The Mode Setting Operand is another specialized Operand which is used with a certain class of instructions which set bits within the mode register (RGD). This operand will be discussed when an instruction of this class is encountered.

g. Indexing

PE Operands can be indexed in two ways: either by RGX or RGS within a PE, or by one of the four ACARs in the CU.

Usually the PE Operand to be indexed is a PEM Address or location. An indexed location is in one of two forms:

** Location*

or

 # Location

where the asterisk (*) means the *Location* is indexed by the contents of RGX and the sharp (#) means that the *Location* is indexed by the contents of RGS. The contents of the specified index register is added to the value of *Location* and that sum is used as the effective PEM Address or location.

LDA *A;

would load the RGA of every PE with the contents of location (A + contents of RGX). RGX may contain a different value in every PE, in which case the RGA of every PE in the array will be loaded from a different location in PEM. This situation is shown in Figure III-2 on page III-17 and will be discussed later.

The contents of one of the four ACARs in the CU can also be used to index a location or PEM Address. The ACAR which contains the indexing value is specified in parentheses after the location, i.e.

LDA A(1);

would load the RGA of every PE with the contents of location (A + contents of ACAR1). Since the contents of ACAR1 is a scalar (and not a vector or row) quantity, the RGA of each PE will be loaded from the same location--A + contents of ACAR1. This is not necessarily the case when indexing is done using RGX or RGS within each PE.

2. CU Operands

A CU Operand may be a CU Memory Address, CU Register, Literal, or Skip Operand.

a. CU Memory Address

A CU Memory Address refers to the contents of a single location in PE Memory (whereas a PE Memory Address refers to the contents of a row of locations in PE Memory). CU Memory is not to be confused with CU Register Storage (the 64 ADB locations, the 4 ACARs and thirteen other CU registers). CU Memory lies within PE Memory. There are CU instructions which reference a single word stored in CU Memory which means that the single word resides in a specific Row and a specific PEM within that row. As we shall see, it takes two coordinates to specify a CU Memory Address: the Row number and the PEM number.

b. CU Register

The available CU registers are the four ACARs ($C0, $C1, $C2, or $C3) or any of the 64 locations within the ADB ($D0 through $D63). Additionally, there are thirteen other CU registers that can be used as a CU Operand, but they will not be covered here.

c. Literal

A CU Literal Operand is usually in one of two forms. One format is the same as the PE Operand, i.e.,

= *Literal*

where *Literal* can be a number or an address. When a CU *Literal* is an address, it refers to a single location in PE Memory.

For example =X refers to the word in PEM_0 of Row X in PEM; while =X+1 would refer to the word in PEM_1 of Row X in PEM. More will be said about this when the Matrix Multiplication problem is discussed.

The other format is used for loop control and looks like:

Increment, Limit, Starting Value

where *Starting Value* is the Initial value of the loop counter

Limit is the upper limit of the loop counter

and *Increment* is the increment for the loop.

Unlike PE instructions, there are CU instructions which can create literals. These instructions and the loop control literal will be discussed later.

d. Skip Operand

A Skip Operand is a special CU operand used to transfer control to another location with the program. It is usually of the form:

, *Location*

where *Location* is the location in the program that will be skipped to based on the results of a test defined by the *Opcode*. There is a strong constraint on the value of Location; it is limited to be within ±127 instructions from the transfer instruction being executed. There is another instruction which can jump to any location in the program that

has as its Operand just the location and is not preceded by a comma. This instruction, called JUMP will not be discussed since it is not necessary for the solution of any of the problems.

E. CU (ADVAST) Instructions

If an accumulator is needed for the execution of a CU instruction one must be specified since there are four accumulators to choose from. Accordingly, the *Opcode* of many CU instructions is followed by a number in parentheses that specifies which ACAR is to be the accumulator for that instruction. The format is:

Opcode (ACAR Number) Operand;

An example might be

LDL(3) $D14;

which means: Load $C3 (ACAR3) from $D14 (Location 14 of the ADB).

F. PE (FINST/PE) Instructions

PE Instructions may be partially processed by the Control Unit but they are not completely executed until the PE array performs the operation. It is very important to remember that one PE instruction does not cause only one action to occur as is the case with CU instructions, it causes 64 actions to occur.

LDA X;

causes the RGA of 64 PEs to be loaded from location X of each PEM or, equivalently, from Row X of PEM.

G. Convention

The following convention will be observed in the prose description of the operation of the instruction types: The words "enabled" and "disabled" will not be mentioned. For example, if an instruction is described as loading the RGA of all PEs, this description is meant to imply all enabled PEs and certainly not the disabled ones. However if an instruction is described as loading the RGB of all PEs, then this is meant to imply all PEs, disabled or enabled. All of the rules which describe the operation of enabled and disabled PEs previously described in Chapter II are in effect.

H. Warning

The description of PE and CU Operands and Instructions will be necessarily incomplete to avoid getting bogged down in details. Also discussion of a third class of instructions (TMU instructions) which are used primarily by the Systems Programmer will be skipped entirely. The intent is to present the minimum amount of detail which will allow the programmer to use a small instruction repertoire to solve the sample problems. This method allows the user to gain a "feel" for how the language works without having to learn all of the intricacies of the language.

I. Sample Problems

1. Summing an Array of Numbers

The first problem a programmer usually solves is to write the algorithm that sums an array of numbers. The problem is stated as follows:

Given an array of numbers $X_1, X_2, X_3, \ldots X_N$, find

$$\sum_{i=1}^{N} X_i$$

and store the result in S.

If we were to solve this problem on a conventional machine, we might use a language like FORTRAN:

```
      S = 0.
      DO  10  I = 1, N
10    S = S + X(I)
```

Now let us consider the instructions in the ILLIAC IV repertoire that we will use to solve this problem.

The first instruction we need is a "load" instruction; one that loads the contents of a PE Register from a PE Memory location or from another PE Register. It is of the form

LDα *Operand;*

where α = A, B, S, X, R or D, and specifies RGA, RGB, RGS, RGX, RGR, or RGD respectively.

Operand is usually a PE Memory Address or location, a PE register, or an ACAR.

LDα is a FINST/PE or a PE instruction.

Examples:

```
LDA   X;
```

means all PEs load their RGA from the contents of PEM location X, or Row X.

```
LDB   $R;
```

means all PEs load their RGB from their RGR.

```
LDA   X(1);
```

means: 1) The contents of ACAR1 ($C1) are added to Row location X. Call this value Y.

2) RGA is loaded from the contents of Row Y.

Once again it is stressed that PEM location X is at the same place in every PEM and can therefore be viewed as a vector or "Row" of PEM locations (see Figure III-1). This concept is, of course, at the very heart of the ILLIAC IV, a memory or register access does not access only one operand, it accesses a vector-full of operands.

Associated with the "load" instruction is the "store" instruction. Its format is similar to the "load" but its operation is just the

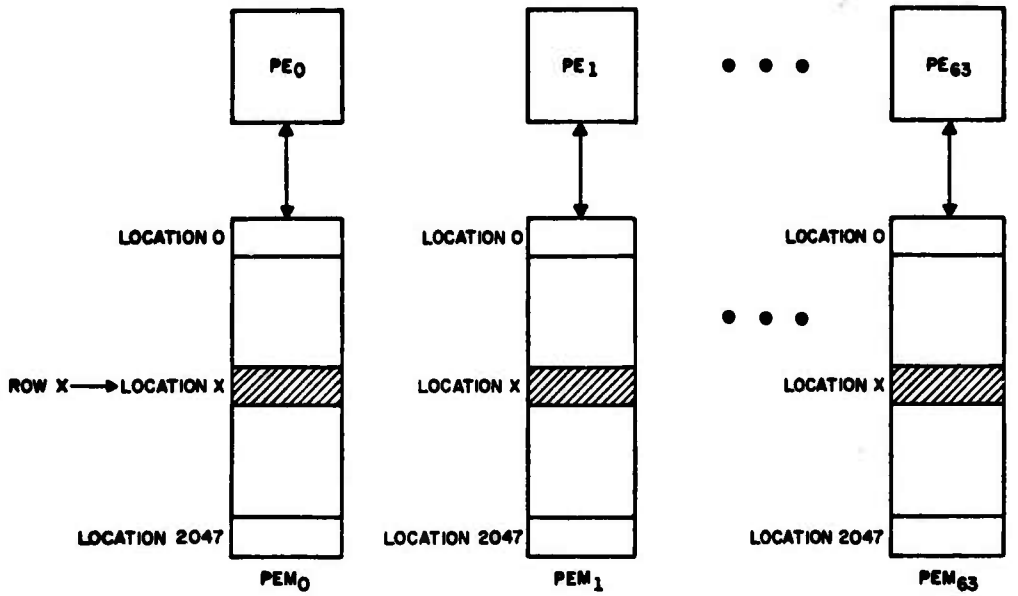


Figure III-1. Symbolic Location X is a "Row" of Processing Element Memory (PEM)

reverse: the contents of a PE register are stored to PE Memory. PE stores are always to PE memory, and never to another PE register; register to register transmission is effected by the load instruction. The store instruction is of the form:

STα *Operand;*

where α = A, B, S, X, or R and specifies RGA, RGB, RGS, RGX or RGR respectively.

Operand is always a PE Memory location.

Like LDα, STα is a PE instruction.

Examples:

STA X;

means all PEs store the contents of their RGA in their PEM Row location X.

STS *A;

means: 1) The contents of RGX of each PE is added to PEM location A; call this new location Y (Y may have a different value in each PE).

2) The contents of RGS of each PE is stored in PE Memory location Y of each PEM.

See Figure III-2 for a picture of how Y can vary within PEM if RGX holds a different value for each PE. Row A has a variable offset specified by the contents of RGX which results in a different location being referenced in each PEM.

Another instruction we need to solve our first problem is one that will add:

ADRN *Operand;*

where *Operand* is usually a PEM Address, a PE Register or an ACAR which contains the value of operand to be added to the accumulator RGA. The result appears in RGA.

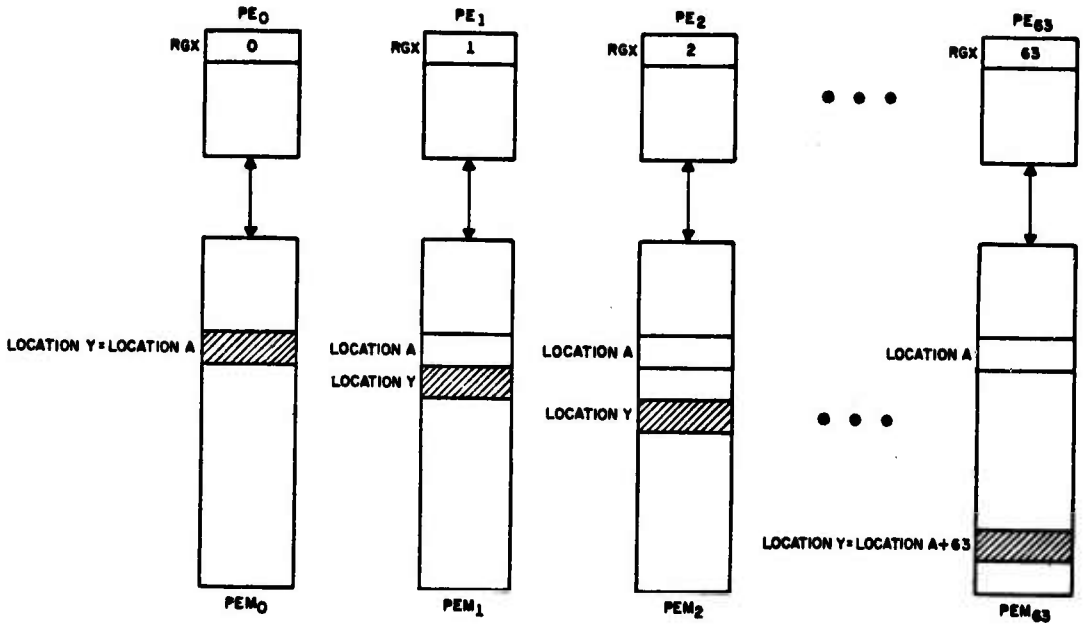


Figure III-2. Indexing by RGX can cause Different Locations within PEM to be Referenced

RGX in PE_0 is 0

RGX in PE_1 is 1

RGX in PE_2 is 2

⋮

RGX in PE_i is i

⋮

RGX in PE_{63} is 63.

STS *A causes the contents of RGS to be stored in location Y.

ADRN is one variation of the PE add instruction; it adds two 64 bit floating point numbers using rounding (R) and normalization (N).

Examples:

```
ADRN   X;
```

means: to the contents of the RGA of every PE add the contents of PEM Row location X and place the result back in RGA.

```
ADRN   X(0);
```

means: 1) Add the contents of ACAR0 ($C0) to PEM location X; call this location Y.

2) The contents of location Y are added to RGA in every PE simultaneously and the result is placed back in the RGA of each PE.

In order to use ILLIAC IV effectively we must use the "ROUTE" instruction in programming a solution to our problem. (The total array of numbers to be summed could be stored entirely within the PEM of one PE, but the computational power of the rest of the PEs in the array would be totally wasted. The scheme we shall use to sum the numbers will attempt to use as many PEs as possible.)

The Route instruction is used to send data from some PE register (RGA, RGB, RGX, or RGS) to the routing register, RGR, and from there route that data a specified distance to another PE's routing register. One form of the Route instruction is:

RTL *Source Register, Routing Distance;*

where *Source Register* can be $A, $B, $X, $R or $S. If *Source Register* and the following comma are not present, the source register is assumed to be $R.

Routing Distance is a number indicating how many PEs to the left or right the data should be routed.

A positive number denotes a route to the right.

A negative number denotes a route to the left.

The "L" in RTL stands for "Local" and not for "Left".

RTL is, of course, a FINST/PE Instruction.

Example:

```
RTL   $A, -3;
```

would cause the following to happen:

1) For all PEs, the contents of RGA are placed in RGR.
2) The contents of RGR of each PE is routed 3 PEs to the left. The results always end up in the R register, RGR; the contents of RGA are unchanged. The Route is always end-around so that, in this case, the contents of RGR of PE_0 would end up in the RGR of PE_{61}.

Since a Route Instruction only changes RGR, it is always executed by the entire PE array, regardless of whether or not a PE is enabled or disabled.

The second operand, the *Routing Distance* can be indexed by an ACAR. If this is the case, the contents of the specified ACAR is added to

the *Routing Distance* and the route is then performed. The general form is:

RTL *Source Register, Routing Distance (ACAR Number);*

where *ACAR Number* is 0, 1, 2 or 3 specifying $C0, $C1, $C2 or $C3.

Examples:

```
RTL   $S, 12(1);
```

would place the contents of RGS in RGR then route RGR a distance of (12 + contents of $C1) to the right.

```
RTL   $S, 0(0);
```

To the *Routing Distance*, in this case zero, is added the contents of $C0. This distance is then used to route the contents of RGR after it has been loaded from RGS. If an ACAR is used to index the *Routing Distance* it is extremely important (for reasons too complicated to describe here) that the number in the ACAR be positive.

We may now program a solution to our problem. For ease of illustration let us assume that we have an eight PE (rather than a 64 PE) machine and that N = 8 (we have 8 numbers to sum) and that they are given to us stored across one Row of PEM at location X. Also, since PE numbering begins at zero let us label our array X_0, X_1, X_2, X_3, X_4, X_5, X_6 and X_7.

The ASK program to perform the sum might look like

```
LDA     X;
RTL     $A, -1;
ADRN    $R;
RTL     $A, -2;
ADRN    $R;
RTL     $A, -4;
ADRN    $R;
STA     S;
```

Table III-1 shows the first seven steps of the above assembly language program. The contents of RGA and RGR of each PE are also shown after the execution of each step. After Step 7 has been executed

$$\sum_{i=0}^{7} X_i$$

is in the RGA of each PE of our 8 PE array. The last instruction, STA S; stores this result to location S as the problem requires.

It should be clear now that we could sum 64 numbers on our 64 PE ILLIAC IV using the following instructions:

Table III-1. Assembly Language Steps to Sum Eight Numbers Using Eight PEs

```
Step 1.  LDA   X;
Step 2.  RTL   $A, -1;
Step 3.  ADRN  $R;
Step 4.  RTL   $A, -2;
Step 5.  ADRN  $R;
Step 6.  RTL   $A, -4;
Step 7.  ADRN  $R;
```

Step No.	Contents of	PE_0	PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	PE_7
1.	RGA	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
	RGR	-	-	-	-	-	-	-	-
2.	RGA	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
	RGR	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_0
3.	RGA	X_0+X_1	X_1+X_2	X_2+X_3	X_3+X_4	X_4+X_5	X_5+X_6	X_6+X_7	X_7+X_0
	RGR	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_0
4.	RGA	X_0+X_1	X_1+X_2	X_2+X_3	X_3+X_4	X_4+X_5	X_5+X_6	X_6+X_7	X_7+X_8
	RGR	X_2+X_3	X_3+X_4	X_4+X_5	X_5+X_6	X_6+X_7	X_7+X_0	X_0+X_1	X_1+X_2
5.	RGA	$X_0+X_1+X_2+X_3$	$X_1+X_2+X_3+X_4$	$X_2+X_3+X_4+X_5$	$X_3+X_4+X_5+X_6$	$X_4+X_5+X_6+X_7$	$X_5+X_6+X_7+X_0$	$X_6+X_7+X_0+X_1$	$X_7+X_0+X_1+X_2$
	RGR	X_2+X_3	X_3+X_4	X_4+X_5	X_5+X_6	X_6+X_7	X_7+X_0	X_0+X_1	X_1+X_2
6.	RGA	$X_0+X_1+X_2+X_3$	$X_1+X_2+X_3+X_4$	$X_2+X_3+X_4+X_5$	$X_3+X_4+X_5+X_6$	$X_4+X_5+X_6+X_7$	$X_5+X_6+X_7+X_0$	$X_6+X_7+X_0+X_1$	$X_7+X_0+X_1+X_2$
	RGR	$X_4+X_5+X_6+X_7$	$X_5+X_6+X_7+X_0$	$X_6+X_7+X_0+X_1$	$X_7+X_0+X_1+X_2$	$X_0+X_1+X_2+X_3$	$X_1+X_2+X_3+X_4$	$X_2+X_3+X_4+X_5$	$X_3+X_4+X_5+X_6$
7.	PGA	$\sum_{i=0}^{7} X_i$	$\sum_{i=0}^{7} X_i$	$\sum_{i=0}^{7} X_i$	$\sum_{i=0}^{7} X_i$	$\sum_{i=0}^{7} X_i$	$\sum_{i=0}^{7} X_i$	$\sum_{i=0}^{7} X_i$	$\sum_{i=0}^{7} X_i$
	RGR	$X_4+X_5+X_6+X_7$	$X_5+X_6+X_7+X_0$	$X_6+X_7+X_0+X_1$	$X_7+X_0+X_1+X_2$	$X_0+X_1+X_2+X_3$	$X_1+X_2+X_3+X_4$	$X_2+X_3+X_4+X_5$	$X_3+X_4+X_5+X_6$

```
LDA     X;
RTL     $A, -1;
ADRN    $R;
RTL     $A, -2;
ADRN    $R;
RTL     $A, -4;
ADRN    $R;
RTL     $A, -8;
ADRN    $R;
RTL     $A, -16;
ADRN    $R;
RTL     $A, -32;
ADRN    $R;
STA     S;
```

The general rule for routing using this kind of algorithm to sum numbers is:

To sum N numbers where $N = 2^I$ and I is a positive integer, we perform I routes using a routing distance starting at 2^0 and ending at 2^{I-1}.

This method of summing numbers is sometimes called a "Logsum" since the routing distance increases as a power of 2. The reader should also check for himself at this point that the Logsum algorithm will work when the *Routing Distance* is positive and all routes take place to the right. The algorithm is independent of the direction of the routes; the result will be the same since routing is end-around in both cases.

Unfortunately the solution to the problem presented above is not very elegant, nor is it concise. It could be made more concise and elegant by using looping instructions.

Of the fourteen instructions needed to sum 64 numbers six pairs of them are of the form:

```
RTL    $A, D;
ADRN   $R;
```

If we could set up instructions around the above instruction pair which would cause the pair to be executed 6 times and would cause the value of *D* to take on the values 1, 2, 4, 8, 16 and 32 consecutively, then the solution would not only be better--it would be a representative method for summing arrays.

There are many looping instructions in the ILLIAC IV repertoire; one of them is TXLTM. Before we discuss the operation of TXLTM we must first describe the LIT instruction which will place the starting value, increment, and upper limit of our loop into a specified ACAR. (This ACAR is then referenced by the TXLTM instruction.) For looping purposes, the LIT instruction loads up a specified ACAR with an increment, a limit, and a starting value. It is of the form:

```
LIT (ACAR Number)      Increment, Limit, Starting Value;
```

where *ACAR Number* is either a 0, 1, 2, or 3 specifying $C0, $C1, $C2 or $C3.

Increment, *Limit* and *Starting Value* have already been briefly discussed and they work in the following way

if *Increment* = 2

Limit = 11

Starting Value = 1

then the sequence 1, 3, 5, 7, 9, 11 can be generated while a loop is being executed 6 times when TXLTM is used. (The TXLTM will "bump up" the value of *Starting Value* by the value of *Increment* each time the loop is traversed.)

Examples:

```
LIT(3)    1, 3, 0;
```

will cause an increment of 1, a starting value of 0 and an upper limit of 3 to be placed in $C3 to be used as loop control variables.

LIT can also be used in a less sophisticated manner to load an ACAR with just one number:

```
LIT(0)    =4;
```

will place the integer 4 (right-adjusted and zero-filled) into $C0. LIT is a CU or ADVAST instruction which is executed completely in ADVAST.

The TXLTM instruction is of the form:

```
TXLTM (ACAR Number)      ,Location;
```

where *ACAR Number* is either a 0, 1, 2, or 3 specifying $C0, $C1, $C2 or $C3.

Location is the symbolic location within the program (somewhere in the program is a *Label* that is the same symbol as *Location*) to which a jump is made if the *Starting Value* is less than the *Limit* in the ACAR specified by *ACAR Number*. If the *Starting Value* is not less than the *Limit*, the next instruction is executed. In either case, the *Starting Value* is increased by *Increment* and placed back in the *Starting Value*.

TXLTM is also an ADVAST instruction and the operand ,*Location* is a Skip Operand.

Warning: Location cannot refer to any location within the program--only to a position within 127 instructions of the TXLTM instruction.

Examples:

```
        TXLTM(1)    ,ALPHA;
```

means "transfer to location ALPHA if the *Starting Value* in ACAR1 is less than the *Limit* in ACAR1. If not, execute the next instruction. In either case, replace the *Starting Value* in ACAR1 by the value: *Starting Value + Increment*."

Let us consider how we would set up a summation loop using the LIT and TXLTM instructions:

```
        LIT(0)      1, 3, 0;
        LIT(1)      =0;
        LDA         $C1;
LOOP:   ADRN        $R;
        TXLTM(0)    ,LOOP;
```

The first instruction sets up an *Increment* of 1, a *Limit* of 3 and a *Starting Value* of 0 in $C0. The second instruction places a zero in ACAR1 which is then loaded into RGA in the third instruction. The fourth instruction at location LOOP adds the contents of RGR to the accumulator (RGA). The last instruction checks to see if the *Starting Value* (0) is less than the *Limit* (3) and then increments the Starting Value by *Increment* (1). The condition will be true the first three times sending control back to LOOP.* After $R is added for the fourth time, the *Starting Value* will be 3, and will not be less than the *Limit*, and so control will drop to the sixth instruction, whatever that may be. The loop is executed four times and a value of 4 times the contents of RGR will be in RGA at the completion of the loop.

The LIT and TXL/TM instructions can provide the loop control for our summation problem but we will need one more instruction to double the routing distance each time through the loop. (We want $\mathcal{D}$ in the instruction pair

```
RTL     $A, D;
ADRN    $R;
```

to start at the value 1 and double each time through the loop.) Since $\mathcal{D}$, the routing distance, can be a constant indexed by an ACAR (remember the example: RTL $S,12(1)) we can place the value 1 into some ACAR, say $C0,

*Since ILLIAC IV has a single instruction stream and a multiple data stream, it is convenient to think of locations holding data as Rows, but locations in the instruction stream are considered as scalars--just like they are on a conventional computer.

using a LIT(0) =1; instruction and double it by shifting the contents of $C0 left one bit each time we pass through the loop. All we need is a shift instruction:

CSHα (ACAR Number) Operand;

where α = L or R denoting a left (L) or right (R) shift

ACAR Number is either 0, 1, 2, or 3 and specifies which ACAR is to be shifted.

Operand is a number or count which specifies how many bits the specified ACAR is to be shifted. This shift is end-off. *Operand* can be ACAR indexed. CSHα is an ADVAST instruction.

Example:

```
CSHL(0)    4;
```

will shift the contents of $C0 four bits to the left end off.

We now have enough instructions to set up a loop for our Logsum program:

```
        LDA       X;
        LIT(0)    =1;
        LIT(1)    1, 6, 1;
LOOP:   RTL       $A, 0(0);
        ADRN      $R;
        CSHL(0)   1;
        TXLTM(1)  ,LOOP;
        STA       S;
```

This represents one possible solution to the problem; there are many others which use different ASK instructions. As an example of a slightly different solution, let us learn one more type of looping instruction.

LESST (ACAR Number) CU Register, Location;

will cause a jump to *Location* if the contents of the ACAR specified by ACAR *Number* (0, 1, 2 or 3) are less than the contents of the Control Unit Register specified by *CU Register* (*CU Register* can be $C0, $C1, $C2, $C3 or $D0 through $D63). LESST is an ADVAST instruction.

Warning: Location cannot refer to any location within the program--only the position within 127 instructions of LESST.

Example:

```
LESST(0)    $C1, LOOP;
```

means that a jump to location LOOP will be made if the contents of $C0 are less than the contents of $C1. If this is not the case, the next instruction will be executed.

Now we may rewrite our Logsum program as follows:

```
        LDA       X;
        LIT(0)    =1;
        LIT(1)    =33;
LOOP:   RTL       $A, 0(0);
        ADRN      $R;
        CSHL(0)   1;
        LESST(0)  $C1, LOOP;
        STA       S;
```

Note that ACARO is being used in dual role: it contains the *Routing Distance* variable and also helps control the loop.

2. Finding the Maximum Value in an Array of Numbers

The problem is to find the largest value in a given array of numbers X_1, X_2 ... X_{64} and place that value in RGA of every PE.

Before attempting to program the solution in ASK, let us look at the top of Table III-2 which lists the steps for solution for 8 specific values assuming an 8 PE array. The lower part of Table III-2 displays the contents of RGR, RGA, and bits E and E1 of RGD after each step. For this example we have used the following specific values for the array of 8 numbers:

$$X_0 = 2, \quad X_1 = 0, \quad X_2 = 5, \quad X_3 = 7,$$

$$X_4 = 1, \quad X_5 = 3, \quad X_6 = 4, \quad X_7 = 6$$

Table III-2. Solution to Problem #2--Finding the Maximum Value in an Array of Eight Numbers

Step 1. The 8 values are in RGA of each PE; $PE_i \leftarrow X_i$, i = 0, 1, ... 7. Enable all PEs. N ← 0.

Step 2. Route RGA 2^N to the right (end around).

Step 3. Compare RGR to RGA; If RGA ≥ RGR, disable PE (Set E = E1 = 0).
If RGA < RGR, leave PE enabled.

Step 4. For all enabled PEs: RGA ← RGR.

Step 5. Enable all PEs (Set E = E1 = 1); then N ← N + 1.

Step 6. If N < 3, Go back to Step 2.
If N ≥ 3, STOP, the largest value is in RGA of every PE.

Step No. and Value of N	Contents of	PE_0	PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	PE_7
1. N = 0	RGA	2	0	5	7	1	3	4	6
	RGR	-	-	-	-	-	-	-	-
	E, E1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
2. N = 0	RGA	2	0	5	7	1	3	4	6
	RGR	6	2	0	5	7	1	3	4
	E, E1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
3. N = 0	RGA	2	0	5	7	1	3	4	6
	RGR	6	2	0	5	7	1	3	4
	E, E1	1,1	1,1	0,0	0,0	1,1	0,0	0,0	0,0
4. N = 0	RGA	6	2	5	7	7	3	4	6
	RGR	6	2	0	5	7	1	3	4
	E, E1	1,1	1,1	0,0	0,0	1,1	0,0	0,0	0,0
5 and 6 N = 1	RGA	6	2	5	7	7	3	4	6
	RGR	6	2	0	5	7	1	3	4
	E, E1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
2. N = 1	RGA	6	2	5	7	7	3	4	6
	RGR	4	6	6	2	5	7	7	3
	E, E1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
3. N = 1	RGA	6	2	5	7	7	3	4	6
	RGR	4	6	6	2	5	7	7	3
	E, E1	0,0	1,1	1,1	0,0	0,0	1,1	1,1	0,0
4. N = 1	RGA	6	6	6	7	7	7	7	6
	RGR	4	6	6	2	5	7	7	3
	E, E1	0,0	1,1	1,1	0,0	0,0	1,1	1,1	0,0
5 and 6 N = 2	RGA	6	6	6	7	7	7	7	6
	RGR	4	6	6	2	5	7	7	3
	E, E1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
2. N = 2	RGA	6	6	6	7	7	7	7	6
	RGR	7	7	7	6	6	6	6	7
	E, E1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
3. N = 2	RGA	6	6	6	7	7	7	7	6
	RGR	7	7	7	6	6	6	6	7
	E, E1	1,1	1,1	1,1	0,0	0,0	0,0	0,0	1,1
4. N = 2	RGA	7	7	7	7	7	7	7	7
	RGR	7	7	7	6	6	6	6	7
	E, E1	1,1	1,1	1,1	0,0	0,0	0,0	0,0	1,1
5 and 6 N = 3	RGA	7	7	7	7	7	7	7	7
	RGR	7	7	7	6	6	6	6	7
	E, E1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1

When we are operating on 64-bit floating point operands, both the E and E1 bits must be on or set (equal to 1) for the PE to be enabled. (In the 32 bit mode the E bit enables one 32-bit floating point word and the E1 bit enables the other.)

Note that at Step 4 and N = 0, when the contents of RGR are placed into RGA of enabled PEs, two values of the largest number (7) appear in RGA of the PEs; at Step 4 and N = 1, four values of the largest number appear, and at Step 4 and N = 2 all eight PEs contain the largest value. Since we use routing distances which are powers of 2, this solution to the problem is sometimes called "Logmax."

In order to code this problem in ASK we will need two more instructions: SET, to set the E and E1 bits and thus enable a PE, and IAL which will set the I bit of RGD based on the results of an arithmetic comparison of the contents of two registers. Unfortunately the E and E1 bits cannot be set directly on the result of an arithmetic comparison; however, another bit of RGD (called the I bit) can be, and E and E1 can be set equal to the I bit. Thus, the E and E1 bits can finally be set based on an arithmetic comparison of the contents of two registers.

The SET instruction can set any bit (E, E1, F, F1, G, H, I or J) of RGD, the mode register. It is of the form:

SETα *Mode Bit . Logic . E Bit;*

where α can be E, E1, F, F1, G, H, I or J.

Mode Bit can be E, E1, F, F1, G, H, I or J also.

Logic is a logical or Boolean operator and can either be OR or AND.

E Bit can only be E or E1.

The AND and OR operations operate on Boolean variables that can only take the values 0 or 1. The tables below define the AND and OR operations:

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

For example: 0 AND 1 = 0

0 OR 1 = 1

The construction: *Mode Bit* . *Logic* . *E Bit* is a Mode Setting Operand as described in section D 1 f.

Either the *Mode Bit* or the *E Bit* can be preceded by a minus (-) sign denoting the logical "NOT" or complement function, i.e., if E is zero then -E is one; if E is one then -E is zero.

The bit of RGD specified by α is set to 1 if the result of *Mode Bit* . *Logic* . *E Bit* equals 1; the bit specified by α is set to zero if the result is zero.

SET is a FINST/PE instruction.

Examples:

SETE I.OR.E;

Suppose I was 1 and

E was 0

before the execution of the above instruction. After the above instruction is executed

E will be equal to 1 because

I.OR.E results in 1.OR.0 = 1

The instruction then says to "Set the E bit equal to 1".

SETE1 I.OR.E;

would result in E1 being set to 1.

If the programmer wished to enable (set equal to 1) the E and E1 bits for all PEs so that he could do 64-bit floating point arithmetic he could do so with the following pair of instructions:

SETE E.OR.-E

SETE1 E.OR.-E

The first instruction insures that the E bit will be set to 1 since the logical expression

E.OR.-E

has the value 1 regardless of whether E is 0 or 1:

if E = 0 E.OR.-E becomes 0.OR.1 = 1

if E = 1 E.OR.-E becomes 1.OR.0 = 1

The same reasoning applies to the setting of the E1 bit in the second instruction.

Next we need an instruction that can set the I bit of RGD based on the results of an arithmetic comparison:

```
IAL   Operand;
```

where *Operand* can be a PE Register, a PE Memory Address, or an ACAR.

When *Operand* specifies a PE Memory Address the IAL instruction arithmetically compares the contents of RGA to the contents of the PE Memory Address and sets the I bit to one if the contents of RGA are less; or sets the I bit to zero if the contents of RGA are not less.

IAL is a FINST/PE instruction.

Examples:

```
IAL   LOC;
```

if contents of RGA < contents of LOC, then set I to 1

if contents of RGA $\geq$ contents of LOC, then set I to 0

Although the comparison is done in every PE simultaneously, the results can vary between different PEs; that is, the I bit in each PE is one or zero depending on the arithmetic comparison after IAL is executed.

There is another instruction:

```
IAG    Operand;
```

which works just like IAL but it uses a "greater than" test rather than a less than test, i.e., the I bit is set to one if the contents of RGA are greater than the contents specified by *Operand*.

Since the IAL instruction sets the I bit and it is the E and E1 bits which enable or disable a PE we must devise a method to set the E and E1 bits based on the value of the I bit. Assume the E bit has been set to one, what instruction will cause the E bit to take on the value of the I bit?

```
SETE    I.AND.E;
```

will set the E bit to the value of the I bit assuming the E bit has been previously set to one.

Consider the following instructions:

```
1    SETE     E.OR.-E;
2    SETE1    E.OR.-E;
3    IAG      $R;
4    SETE     I.AND.E;
5    SETE1    I.AND.E;
6    LDA      $R;
```

Instructions 1 and 2 enable all of the PEs, setting their E and E1 bits to 1. Instruction 3 sets the I bit equal to 1 for every PE whose RGA contents is greater than its RGR contents. If RGA $\leq$ RGR then the I bit is set to 0. Instructions 4 and 5 disable all PEs whose I bit is zero, thus if

RGA > RGR	PE remains enabled
RGA $\leq$ RGR	PE is disabled

Instruction 6 is executed only by those PEs which are in the enabled state. Thus

if RGR < RGA	RGA is loaded from RGR
if RGR $\geq$ RGA	RGA remains unchanged

The above set of six instructions will compare the contents of RGA to the contents of RGR for every PE in the array and the lesser of the two will appear in RGA.

Note that instruction 5 uses the value of E in its Mode Setting Operand that was calculated in instruction 4, so that instruction 2 is really not necessary.

We may now write the ASK code necessary to select the largest number from an array of 64 values. Assume the 64 values are stored at Row X and the largest value is to appear in RGA of every PE:

```
 1            SETE        E.OR.-E;
 2            SETE1       E.OR.-E;
 3            LDA         X;
 4            LIT(0)      =1;
 5            LIT(1)      1, 6, 1;
 6    LOOP:   RTL         $A, 0(0);
 7            IAL         $R;
 8            SETE        I.AND.E;
 9            SETE1       I.AND.E;
10            LDA         $R;
11            SETE        E.OR.-E;
12            SETE1       E.OR.-E;
13            CSHL(0)     1;
14            TXLTM(1)    ,LOOP;
```

Instructions 1 and 2 enable the entire PE array.

Instruction 3 brings the array of numbers from PE Memory Address X to the RGA of each PE.

Instruction 4 initializes $C0 (which will be used to control the routing distance) to 1.

Instruction 5 sets up loop control in $C1.

Instruction 6 performs the route.

Instructions 7, 8, 9, and 10 are the heart of the program: Instruction 7 sets the I bit to one of all PEs whose RGA < RGR. Instructions 8 and 9 disable all PEs whose RGA $\geq$ RGR and leave enabled those that have RGA < RGR. Instruction 10 loads the RGA of every enabled PE from RGR. Therefore, only those PEs whose RGA < RGR will have their RGA loaded from RGR and RGA will contain the larger of RGA and RGR or Max (RGA, RGR). After this process is repeated six times through the loop the largest value in the array will appear in the RGA of every PE.

Instructions 11 and 12 re-enable the entire PE array previous to the next pass through the loop.

Instruction 13 doubles the routing distance.

Instruction 14 transfers control back to Instruction 6 if the loop is not finished.

It is very important to understand the operation of Instructions 7, 8, 9 and 10:

```
 7    IAL     $R;
 8    SETE    I.AND.E;
 9    SETE1   I.AND.E;
10    LDA     $R;
```

They perform the following logic:

If the contents of RGA < contents of RGR, set I to 1.

then I.AND.E has the value 1 also

thus E and E1 are set to 1 and the PE is enabled.

If the contents of RGA $\geq$ contents of RGR, set I to 0.

then I.AND.E has the value 0 also

thus E and E1 are set to 0 and the PE is disabled.

Instruction 10 is only executed by enabled PEs and this means the larger of RGR and RGA ends up in RGA--which is just what we want.

3. Matrix Multiplication

Given a 4 x 4 Matrix X = $\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix}$

and a 4 x 4 Matrix Y = $\begin{bmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \\ y_{41} & y_{42} & y_{43} & y_{44} \end{bmatrix}$

we can compute the matrix Z = X · Y using the definition:

$$z_{ij} = \sum_{k=1}^{4} x_{ik}\, y_{kj} \qquad \text{for } i = 1, 2, 3, 4 \quad j = 1, 2, 3, 4$$

The FORTRAN Program to compute the product of X and Y based on the above definition might look like

```
      DO  20  I = 1, 4
      DO  20  J = 1, 4
      SUM = 0.
      DO  10  K = 1, 4
   10 SUM = SUM + X(I,K) * Y(K,J)
   20 Z(I,J) = SUM
```

The method using the definition of matrix multiplication is not the best algorithm for ILLIAC IV, however. Since ILLIAC IV can handle a whole row or vector of values simultaneously, a matrix multiplication program should take advantage of that fact to be efficient.

Therefore, consider the following algorithm for matrix multiplication:

1. Take x_{11} and multiply it times the 1st row of Y
2. Take x_{12} and multiply it times the 2nd row of Y
3. Take x_{13} and multiply it times the 3rd row of Y
4. Take x_{14} and multiply it times the 4th row of Y

Each of the above 4 operations is multiplying a scalar value (x) times a vector (Y) resulting in a vector or row quantity. If we sum the above rows we have the 1st row of Z:

$$
\begin{array}{lcl}
x_{11} * (\text{1st Row of Y}) & = & (x_{11}y_{11} \quad x_{11}y_{12} \quad x_{11}y_{13} \quad x_{11}y_{14}) \\
+ & & + \\
x_{12} * (\text{2nd Row of Y}) & = & (x_{12}y_{21} \quad x_{12}y_{22} \quad x_{12}y_{23} \quad x_{12}y_{24}) \\
+ & & + \\
x_{13} * (\text{3rd Row of Y}) & = & (x_{13}y_{31} \quad x_{13}y_{32} \quad x_{13}y_{33} \quad x_{13}y_{34}) \\
+ & & + \\
x_{14} * (\text{4th Row of Y}) & = & (x_{14}y_{41} \quad x_{14}y_{42} \quad x_{14}y_{43} \quad x_{14}y_{44}) \\
\hline
(\text{1st Row of Z}) & = & \sum_{k=1}^{4} x_{1k}y_{k1} \quad \sum_{k=1}^{4} x_{1k}y_{k2} \quad \sum_{k=1}^{4} x_{1k}y_{k3} \quad \sum_{k=1}^{4} x_{1k}y_{k4}
\end{array}
$$

To get the 2nd row of Z we take the second row of X and multiply each element by the 1st, 2nd, 3rd and 4th rows of Y and sum the rows again. Similarly for the third and fourth rows of Z. The algorithm might appear as:

1. $i \leftarrow 1$ (set i equal to 1)
2. Take x_{i1} and multiply it times the 1st row of Y
3. Take x_{i2} and multiply it times the 2nd row of Y
4. Take x_{i3} and multiply it times the 3rd row of Y
5. Take x_{i4} and multiply it times the 4th row of Y
6. Take sum of the above 4 rows and store in i^{th} Row of Z
7. $i \leftarrow i + 1$ (Bump up i)
8. If $i > 4$ STOP, otherwise go back to Step 2.

The flow chart for performing Z = X · Y

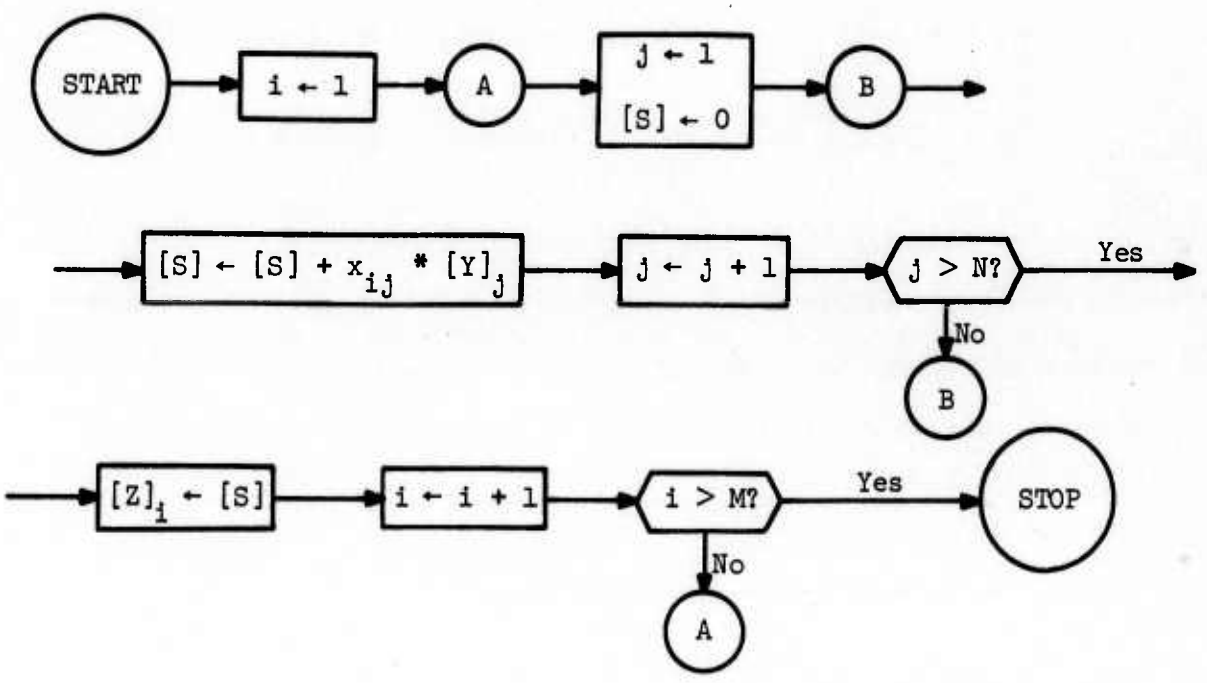
where X is M rows by N columns

Y is N rows by K columns

and Z is M rows by K columns is:

Notation: $[Y]_j$ = jth row of Matrix Y

[S] = row or vector of values



In order to write the ASK code for the matrix multiplication problem we must learn a few more instructions.

SLIT (ACAR Number) =Literal;

SLIT is called a "short literal" instruction and it works just like the literal instruction LIT except that the value of =Literal is placed in the

low order 24 bits of the ACAR specified by *ACAR Number*. The other 40 bits of the specified ACAR are unchanged--as opposed to the LIT instruction which operated on the entire 64 bits of the specified ACAR. SLIT is an ADVAST instruction.

Example:

```
SLIT(0)    =15;
```

will place the binary integer 15 in the low-order 24 bits of $C0.

```
SLIT(1)    =X;
```

will place the location of the variable X into the low-order 24 bits of $C1. In the above example (X) is a symbolic CU Memory Address and refers to one word of storage in PE Memory, not to a whole row. This will be discussed in further detail shortly.

```
ALIT (ACAR Number)      =Literal;
```

The ALIT instruction adds the value of *=Literal* to the low-order 24 bits of the ACAR specified by *ACAR Number*. The other 40 bits of the specified ACAR are unchanged. ALIT is an ADVAST instruction.

Example:

```
ALIT(3)    =1;
```

will increase the low-order 24 bits of $C3 by 1.

It can now be pointed out that when a LIT instruction of the form:

```
LIT(3)    1, 6, 0;
```

is used the *Starting Value* of 0 is placed in the low-order 24 bits of $C3. Therefore if the programmer wishes to modify the *Starting Value*, he can do so with an ALIT or a SLIT instruction. ACAR indexing is also done with only the *Starting Value* field of the ACAR so that offsets from a particular Row can increase (or decrease) as a loop is traversed. More will be said about this point later.

We will, of course, also need a multiply instruction:

```
MLRN      Operand;
```

where *Operand* can be a PE register, a PE Memory Address or an ACAR.

The value specified by *Operand* is multiplied by the contents of RGA and the result of the multiplication appears back in RGA. (Lower significant bits also appear in RGB but we will not use this information.) Rounding and normalization will occur. MLRN is a FINST/PE instruction.

Examples:

```
LDA     X;
MLRN    Y;
```

causes the product XY to appear in the RGA of every PE.

```
LIT(2)    =2.0;
MLRN      $C2;
```

causes the contents of every RGA in the PE Array to be doubled. The constant 2.0, stored in $C2 is sometimes called a "broadcast" Operand since it is a scalar value that is transmitted or broadcast to all PEs for multiplication via the Common Data Bus.

The method we shall use to perform the matrix multiplication algorithm will transmit the appropriate X values to an ACAR in the Control Unit, which in turn will be "broadcast" to the PE array as part of the multiply instruction (MLRN). It is easy enough to get a scalar constant into a specified ACAR using a LIT instruction as above, but how can we "get" the scalar elements within the X matrix? We need an instruction that will LOAD a specified CU register from a single location of a single PE memory. We do this with the LOAD instruction:

LOAD (ACAR *Number*) *CU Register;*

where ACAR *Number* can be 0, 1, 2 or 3 and specifies the ACAR ($C0, $C1, $C2 or $C3) which contains the CU Memory Address whose contents is to be loaded into *CU Register*. *CU Register* can be $C0, $C1, $C2, $C3 or $D0 through $D63, and can be ACAR indexed.

Example:

```
SLIT(2)    =X;
LOAD(2)    $C3;
```

The SLIT instruction will load $C2 with the CU Memory Address of X. The LOAD instruction will then load the contents of CU Memory Address X into $C3.

As was pointed out in section D 2 a, a CU Memory Address references a single word in PE Memory (and not a row of words as is the case with a PE Memory address). This single word referenced by a CU Memory Address resides in a specified row and in a specified PE within that row and so it requires two coordinates to specify a CU Memory Address. The scheme for presenting the two coordinates is as follows:

A CU Memory Address consists of two parts, a PE row address followed by an offset which indicates how many PEs to the right or left is the single word referenced. For example, if X + 5 is a CU Memory Address then the specified row is PE Memory Row X, the plus sign indicates the direction of offset is to the right and precedes the offset distance 5 so that the word referenced by CU Memory Address X + 5 is the word in PEM_5 of Row X in PE Memory. The general form is:

Row Address ± Offset

If the sign is positive the offset is to the right, if negative the offset is to the left.

If X - 1 is a CU Memory Address then the offset is 1 to the left so that the word referenced by CU Memory Address X - 1 is in PEM_{63} of the Row preceding X or Row X - 1 (see Figure III-3).

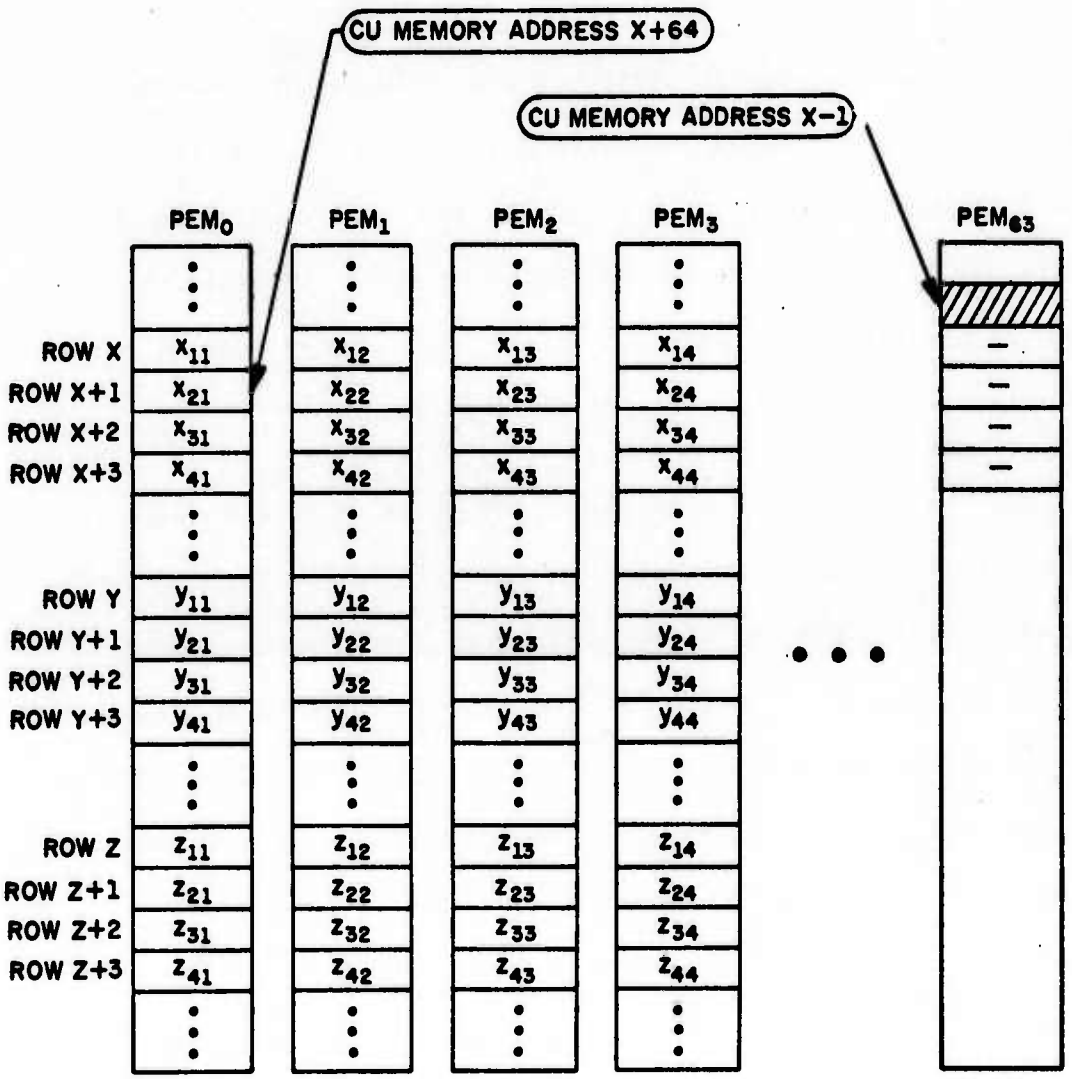


Figure III-3. Memory Storage for Matrix Multiply Problem

If X is a CU Memory Address then the offset is presumed to be zero and the word referenced by CU Memory Address X is in PEM_0 of Row X.

If X + 64 is a CU Memory Address then the offset is 64 to the right so that the word referenced by CU Memory Address X + 64 is in PEM_0 of the Row following X or Row X + 1 (see Figure III-3).

There is no problem in ascertaining whether an address is a PE or CU Memory Address since a CU Memory Address can only appear as an Operand for the CU (or ADVAST) instructions LIT, SLIT or ALIT.

Summarizing: Operands which are part of CU instructions are called CU Operands, and if this Operand is a Memory Address then the Operand is called a CU Memory Address and it refers to a single word (and not a row) in PE Memory. Operands which are part of PE instructions are called PE Operands and if this Operand is an address then the Operand is called a PE Memory Address or a Row and it refers to a Row of PE Memory.

Let us now see how CU Memory Addressing works within the context of our Matrix Multiplication Problem:

Suppose the X, Y, and Z Matrices are stored as shown in Figure III-3. The following pair of instructions:

```
SLIT(2)   =X+1;
LOAD(2)   $C3;
```

would cause x_{12} which is stored in PEM_1 to be loaded into $C3. Note that LOAD is a CU instruction which loads one word from PE Memory (called CU Memory) into $C3.

Now consider how the following sequence of instructions operate:

```
1          SLIT(2)     =X-1;
2          LIT(1)      1, 3, 0;
3   LOOP:  ALIT(2)     =1;
4          LOAD(2)     $C3;
5          TXLTM(1)    ,LOOP;
```

Instruction 1 places the location of CU Memory Address X - 1 into the low order 24 bits of $C2. The location referred to is CU Memory Address X - 1 which is indicated by the shaded portion of PEM_{63} of Row X - 1 in Figure III-3. Note that CU Memory Address X - 1 is one word back from CU Memory Address X. Since SLIT is a CU instruction, CU Memory Address X - 1 is not back one whole row but only one word.

Instruction 2 sets up the loop control using $C1.

Instruction 3 increases the low order 24 bits of $C2 by 1. In other words, the location now referenced by $C2 is CU Memory Address X which contains x_{11}.

Instruction 4 will load $C3 from CU Memory Address X, i.e., x_{11} will be transmitted to $C3.

Instruction 5 will loop us back up to LOOP until we have executed our loop 4 times.

The second time through the LOOP we start at instruction 3 where $C2 is increased again by 1. The location now referenced by $C2 is CU Memory Address X + 1 which contains x_{12}.

Instruction 4 will now load $C3 with x_{12}.

The third time through the loop instruction 4 will load $C3 with x_{13}. The fourth time through the loop instruction 4 will load $C3 with x_{14}.

It will be in this way we shall transmit the appropriate value of x_{ij} to ACAR3 to be "broadcast" as a multiplier.

Let us again review the convention of referencing one value that is stored in PE Memory by a CU Memory address:

When a PE instruction references memory location X + 1 it is referring to a whole Row of values (in our example the <u>second</u> row of the X matrix as shown in Figure III-3). Since the PEs are an array this type of referencing is possible. However, when a CU instruction references a location in PE memory it can only refer to the contents of one location (a scalar value) since it has no capability to store a row or vector. Therefore when X + 1 appears as a CU Memory Address it must be interpreted differently: the first location of Row X is found in PE Memory. X + 1 then refers to the next word in row X which is in PEM_1 and contains x_{12}. Similarly X + 3 as a CU Memory Address refers to x_{14}. Finally, and most important:

X + 64 as a CU Memory Address refers to the first location in PE row X + 1 which contains x_{21}. (See Figure III-3.)

For our problem since we are multiplying a 4 x 4 matrix by a 4 x 4 matrix we shall have to remember to skip 60 locations to bring us to the beginning of the next row of X. We can do this with an

```
ALIT(2)    =60;
```

instruction whenever we are ready to reference the next X Row of PE Memory.

Before we write the Matrix Multiply program, there is one more fine point that we must cover. Consider the following ASK instructions:

```
LIT(0)     2,5,1;
LDA        Y(0);
```

The first instruction sets up a *Starting Value* of 1, a *Limit* of 5 and an an *Increment* of 2 in $C0. The second instruction loads the RGA of all PEs in the array from PE Memory Address Y indexed by the contents of $C0. But $C0 contains three values--which one is used to index location Y? The answer is that only the low-order 16 bits of an ACAR are used to index a PE Memory Address. Since the *Starting Value* resides in the low-order 24 bits of $C0, it is the *Starting Value* that is used to index Y and therefore the

```
LDA    Y(0);
```

instruction will cause Row Y + 1 to be loaded into RGA of all PEs in the array, since $C0 contains a *Starting Value* of 1.

The following ASK instructions:

```
        LIT(1)      1,3,0;
LOOP:   LDA         Y(1);
        TXLTM(1)    ,LOOP;
```

would cause RGA of each PE in the array to be loaded consecutively with ROW Y, ROW Y + 1, ROW Y + 2 and ROW Y + 3 as the loop is traversed.

We are now ready to write the Matrix Multiply program. We assume the X and Y matrices are given and the storage of their elements is as shown in Figure III-3.

```
                1          LIT(0)      1, 3, 0;
                2          SLIT(2)     =X-1;
                3   LOOP2: LDS         =0;
                4          LIT(1)      1, 3, 0;
OUTER LOOP      5   LOOP1: ALIT(2)     =1;
  INNER LOOP    6          LOAD(2)     $C3;
                7          LDA         Y(1);
                8          MLRN        $C3;
                9          ADRN        $S;
               10          LDS         $A;
               11          TXLTM(1)    ,LOOP1;
               12          STS         Z(0);
               13          ALIT(2)     60;
               14          TXLTM(0)    ,LOOP2;
```

Note that the outer loop (LOOP2) is controlled by $C0 and that RGS is used to accumulate the sums. $C0 also indexes Z at instruction 12 where the contents of RGS are stored in ROW Z, ROW Z + 1, ROW Z + 2 and ROW Z + 3 as the *Starting Value* in $C0 takes on the values 0, 1, 2 and 3 as LOOP2 is traversed.

$C1 is used in much the same way. It controls the inner loop (LOOP1) and indexes Y at instruction 7. As is the case with Z, using $C1 to index Y makes reference to succeeding rows of Y, i.e., the first time through the loop instruction 7 loads RGA of every PE from Row 1 of the Y matrix, the second time through from Row 2, etc.

Table III-3 presents the contents of pertinent registers as the ASK code executes through LOOP1 and LOOP2.

The method used here to multiply two 4 x 4 matrices is, of course, only a choice from many possible algorithms. For larger matrices there are more efficient methods which utilize the BIN instruction:

```
BIN (ACAR Number)      ADB Location;
```

The BIN works just like the LOAD except that it moves eight values from PE Memory instead of one. *ACAR Number* specifies the ACAR (0, 1, 2, or 3) which contains the CU Memory Address of the first of the eight values to be moved. The eight values must be stored contiguously. The *ADB Location* denotes the starting location in the ADB where the eight values are to be stored. The *ADB Location* can also be ACAR indexed. BIN is an ADVAST instruction.

Table III-3. Step-by-Step Display of ASK Instructions and Pertinent Registers for Matrix Multiply Problem

ASK Instruction and Number	Contents of	for PE_0	PE_1	PE_2	PE_3	$C3	$C2	$C1	$C0
1 LIT(0) 1,3,0;	RGA	-	-	-	-	-	-	-	1,3,0
	RGS	-	-	-	-				
2 SLIT(2) =X-1;	RGA	-	-	-	-	-	X-1	-	1,3,0
	RGS	-	-	-	-				
3 LOOP2: LDS =0;	RGA	-	-	-	-	-	X-1	-	1,3,0
	RGS	0	0	0	0				
4 LIT(1) 1,3,0;	RGA	-	-	-	-	-	X-1	1,3,0	1,3,0
	RGS	0	0	0	0				
5 LOOP1: ALIT(2) =1;	RGA	-	-	-	-	-	X	1,3,0	1,3,0
	RGS	0	0	0	0				
6 LOAD(2) $C3;	RGA	-	-	-	-	x_{11}	X	1,3,0	1,3,0
	RGS	0	0	0	0				
7 LDA Y(1);	RGA	y_{11}	y_{12}	y_{13}	y_{14}	x_{11}	X	1,3,0	1,3,0
	RGS	0	0	0	0				
8 MLRN $C3;	RGA	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$	x_{11}	X	1,3,0	1,3,0
	RGS	0	0	0	0				
9 ADRN $S;	RGA	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$	x_{11}	X	1,3,0	1,3,0
	RGS	0	0	0	0				
10 LDS $A;	RGA	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$	x_{11}	X	1,3,0	1,3,0
	RGS	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$				

Table III-3 (continued)

ASK Instruction and Number	Contents of	for PE_0	PE_1	PE_2	PE_3	$C3	$C2	$C1	$C0
11 TXLTM(1) ,LOOP1;	RGA	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$	x_{11}	X	1,3,1	1,3,0
	RGS	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$				
5 LOOP1: ALIT(2) =1;	RGA	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$	x_{11}	X+1	1,3,1	1,3,0
	RGS	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$				
6 LOAD(2) $C3;	RGA	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$	x_{12}	X+1	1,3,1	1,3,0
	RGS	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$				
7 LDA Y(1);	RGA	y_{21}	y_{22}	y_{23}	y_{24}	x_{12}	X+1	1,3,1	1,3,0
	RGS	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$				
8 MLRN $C3;	RGA	$x_{12}y_{21}$	$x_{12}y_{22}$	$x_{12}y_{23}$	$x_{12}y_{24}$	x_{12}	X+1	1,3,1	1,3,0
	RGS	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$				
9 ADRN $8;	RGA	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$	x_{12}	X+1	1,3,1	1,3,0
	RGS	$x_{11}y_{11}$	$x_{11}y_{12}$	$x_{11}y_{13}$	$x_{11}y_{14}$				
10 LDS $A;	RGA	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$	x_{12}	X+1	1,3,1	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$				
11 TXLTM(1) ,LOOP1;	RGA	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$	x_{12}	X+1	1,3,2	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$				
5 LOOP1: ALIT(2) =1;	RGA	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$	x_{12}	X+2	1,3,2	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$				
6 LOAD(2) $C3;	RGA	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$	x_{13}	X+2	1,3,2	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$				

Table III-3 (continued)

ASK Instruction and Number	Contents of	for PE_0	for PE_1	for PE_2	for PE_3	$C3	$C2	$C1	$C0
7 LDA Y(1);	RGA	y_{31}	y_{32}	y_{33}	y_{34}	x_{13}	X+2	1,3,2	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$				
8 MLRN $C3;	RGA	$x_{13}y_{31}$	$x_{13}y_{32}$	$x_{13}y_{33}$	$x_{13}y_{34}$	x_{13}	X+2	1,3,2	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$				
9 ADRN $S;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$	x_{13}	X+2	1,3,2	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}$	$x_{11}y_{12}+x_{12}y_{22}$	$x_{11}y_{13}+x_{12}y_{23}$	$x_{11}y_{14}+x_{12}y_{24}$				
10 LDS $A;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$	x_{13}	X+2	1,3,2	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$				
11 TXLTM(1) ,LOOP1;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$	x_{13}	X+2	1,3,3	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$				
5 LOOP1: ALIT(2) =1;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$	x_{13}	X+3	1,3,3	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$				
6 LOAD(2) $C3;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$	x_{14}	X+3	1,3,3	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$				
7 LDA Y(1);	RGA	y_{41}	y_{42}	y_{43}	y_{44}	x_{14}	X+3	1,3,3	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$				
8 MLRN $C3;	RGA	$x_{14}y_{41}$	$x_{14}y_{42}$	$x_{14}y_{43}$	$x_{14}y_{44}$	x_{14}	X+3	1,3,3	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$				
9 ADRN $S;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$	x_{14}	X+3	1,3,3	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}$				

Table III-3 (continued)

ASK Instruction and Number	Contents of	for PE_0	for PE_1	for PE_2	for PE_3	$C3	$C2	$C1	$C0
10 LDS $A;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$	x_{14}	X+3	1,3,3	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$				
11 TXLTM(1) ,LOOP1;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$	x_{14}	X+3	1,3,4	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$				
12 STS Z(0);	All Registers remain the same and the contents of RGS are stored away to the 1st Row of Z.								
13 ALIT(2) =60;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$	x_{14}	X+63	1,3,4	1,3,0
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$				
14 TXLTM(0) ,LOOP2;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$	x_{14}	X+63	1,3,4	1,3,1
	RGS	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$				
3 LDS =0;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$	x_{14}	X+63	1,3,4	1,3,1
	RGS	0	0	0	0				
4 LIT(1) 1,3,0;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$	x_{14}	X+63	1,3,0	1,3,1
	RGS	0	0	0	0				
5 LOOP1: ALIT(2) =1;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$	x_{14}	X+64	1,3,0	1,3,1
	RGS	0	0	0	0				
6 LOAD(2) $C3;	RGA	$x_{11}y_{11}+x_{12}y_{21}+x_{13}y_{31}+x_{14}y_{41}$	$x_{11}y_{12}+x_{12}y_{22}+x_{13}y_{32}+x_{14}y_{42}$	$x_{11}y_{13}+x_{12}y_{23}+x_{13}y_{33}+x_{14}y_{43}$	$x_{11}y_{14}+x_{12}y_{24}+x_{13}y_{34}+x_{14}y_{44}$	x_{21}	X+64	1,3,0	1,3,1
	RGS	0	0	0	0				
7 LDA Y(1);	RGA	y_{11}	y_{12}	y_{13}	y_{14}	x_{21}	X+64	1,3,0	1,3,1
	RGS	0	0	0	0				

Table III-3 (continued)

ASK Instruction and Number	Contents of	for PE_0	PE_1	PE_2	PE_3	$C3	$C2	$C1	$C0
8 MLRN $C3;	RGA	$x_{21}y_{11}$	$x_{21}y_{12}$	$x_{21}y_{13}$	$x_{21}y_{14}$	x_{21}	X+64	1,3,0	1,3,1
	RGS	0	0	0	0				
9 ADRN $S;	RGA	$x_{21}y_{11}$	$x_{21}y_{12}$	$x_{21}y_{13}$	$x_{21}y_{14}$	x_{21}	X+64	1,3,0	1,3,1
	RGS	0	0	0	0				
10 LDS $A;	RGA	$x_{21}y_{11}$	$x_{21}y_{12}$	$x_{21}y_{13}$	$x_{21}y_{14}$	x_{21}	X+64	1,3,0	1,3,1
	RGS	$x_{21}y_{11}$	$x_{21}y_{12}$	$x_{21}y_{13}$	$x_{21}y_{14}$				
11 TXLTM(1) ,LOOP1;	RGA	$x_{21}y_{11}$	$x_{21}y_{12}$	$x_{21}y_{13}$	$x_{21}y_{14}$	x_{21}	X+64	1,3,1	1,3,1
	RGS	$x_{21}y_{11}$	$x_{21}y_{12}$	$x_{21}y_{13}$	$x_{21}y_{14}$				

Instructions 5 through 11 are executed until the 2nd Row of Z is formed.

Instructions 12, 13 and 14 store the sum in the 2nd row of Z and transfer control back to LOOP2 where the sum is reset to zero and the inner loop is also reset.

Instructions 5 through 11 are executed until the 3rd row of Z is formed.

Instructions 12, 13 and 14 store the sum in the 3rd row of Z and transfer control back to LOOP2 where the sum is reset to zero and the inner loop is also reset.

Instructions 5 through 11 are executed until the 4th Row of Z is formed.

Instructions 12, 13 and 14 store the sum in the 4th row of Z and the program ends.

Examples:

```
SLIT(2)    =X;
BIN(2)     $D8;
```

would cause the first eight values stored starting at CU Memory Address X (the values in PEM_0, PEM_1 ... PEM_7 of row X) to be placed in $D8, $D9 ... $D15 respectively.

```
LIT(1)     =8;
SLIT(2)    =X;
BIN(2)     $D0(1);
```

would cause the first eight values starting at CU Memory Address X to be placed in $D8, $D9 ... $D15.

The BIN instruction allows us to send eight values from PE Memory to a section of the ADB. With reference to our Matrix Multiplication problem, we would next need an instruction that could transfer the values from the ADB into an ACAR so that they could be broadcast as multipliers. The ADB locations $D0 through $D63 cannot be used as PE operands (see section D 1 d), only one of the four ACARs is permissible. LDL could be used in this case to transfer information between CU Registers:

LDL (*ACAR Number*) *CU Register*;

LDL loads the specified *ACAR Number* (0, 1, 2 or 3) from the specified *CU Register* ($C0, $C1, $C2, $C3 or $D0 through $D63). The *CU Register* can be ACAR indexed. LDL is an ADVAST instruction.

Example:

```
LDL(3)   $D8;
```

would load $C3 from $D8. The LDL instruction would have to be executed 8 times (within a loop) to transmit the contents of $D8 through $D15 (which we previously transmitted from PE Memory via the BIN instruction).

The instruction pair

```
LIT(1)   =2;
LDL(3)   $D0(1);
```

would load $C3 from $D2, since $D0 is indexed by $C1.

If we were multiplying two 64 x 64 matrices the combination of BIN and LDL in a loop would be more efficient than LOAD which transmits just one value at a time.

4. Matrix Transpose

We obtain the transpose of a matrix by switching the rows for the columns and the columns for the rows, that is,

if A is N x M matrix comprised of elements a_{ij}

and B is M x N matrix comprised of elements b_{ij}

then B is the transpose of A, or $B = A^T$ if and only if

$$b_{ij} = a_{ji} \text{ for } \quad 1 \leq i \leq M$$
$$1 \leq j \leq N$$

A FORTRAN SUBROUTINE to transpose A and store the result in B might look like:

```
      SUBROUTINE TRANS(A, B, M, N)
      DIMENSION A(N,M), B(M,N)
      DO  10  I = 1,M
      DO  10  J = 1,N
10    B(I,J) = A(J,I)
      RETURN
      END
```

Note that the FORTRAN program moved only one element of A at a time to the appropriate position within the B matrix. Using ASK we will be able to move a whole row at a time.

There is a problem in data storage we must consider before we even attempt to write down the algorithm to transpose a matrix. Within the transpose program we will be accessing the rows of matrix A and storing them to the appropriate column of B. Now, if we store a matrix in PE Memory as we did in the Matrix Multiply problem then rows can be accessed very easily and efficiently with one ASK instruction. For example, since the rows of the matrix are stored across Rows of PE Memory, the instruction LDA =X+1 would access the second row of matrix X (see Figure III-3). Getting hold of columns in a simultaneous manner is more difficult when we use this "straight" (no change to the topology of the matrix--rows are stored as rows and each element is in its proper location) storage scheme as shown in Figure III-3. Since column j of the matrix is completely

contained within PEM_{j-1} there is no single ASK instruction which can access a column simultaneously.

Since we will be wanting to access (read or write) columns as well as rows with equal facility for our Matrix Transpose problem, we must first develop a storage scheme which will allow us to do this. What we want is a storage allocation such that each element of a column is also in a different PEM. One such allocation is called "skewed storage" and is shown for a 4 x 4 Matrix in Figure III-4. For simplicity we assume a 4 PE ILLIAC until we actually code the problem for a 64 x 64 matrix.

Note that the skewed storage scheme as shown in Figure III-4 accomplishes our goal: each element of each row is in a different PEM and each element of each column is in a different PEM. Now let us look at how this type of storage can be used to access columns as efficiently as rows:

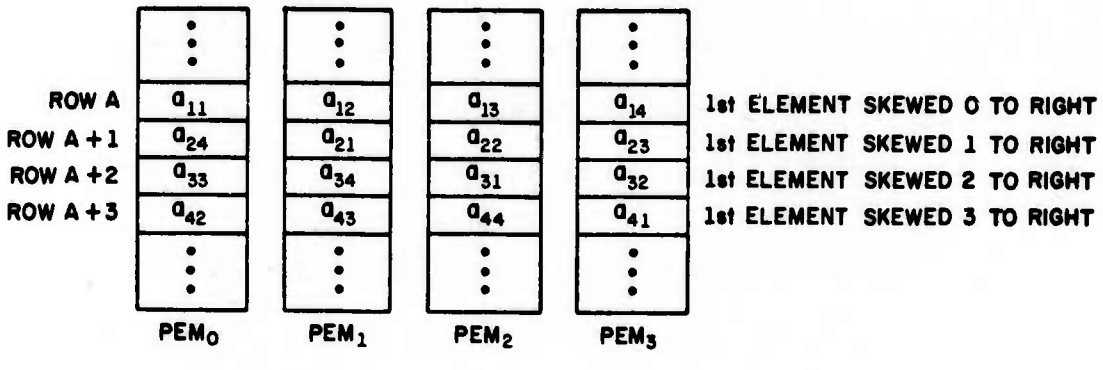


Figure III-4. Example of Skewed Storage. The 4 x 4 Matrix A is stored skewed in a 4 PE ILLIAC.

Suppose RGX has been loaded with the values 0, 1, 2 and 3 as shown in Figure III-5(a). If our 4 x 4 matrix A is stored skewed, what will appear in RGA if we execute the ASK instruction?

LDA *A;

Since the asterisk (*) denotes indexing by RGX, we can view the value in RGX as an offset to Row A so that RGA will appear as shown in Figure III-5(a) after LDA *A; has been executed. A closer look at RGA shows that it now contains the 1st column of the matrix A.

Now suppose that we rotate the values in RGX one place to the right so that they appear as

3, 0, 1, 2 as shown in Figure III-5(b)

Now if we perform the same instruction

LDA *A;

the offsets are different so that the 2nd column of Matrix A will appear in RGA after the instruction has been executed. However, the 2nd column is not in the right order; it must be rotated one place to the left before storing it as a row in our transpose problem.

Figure III-5(c) shows how the 3rd column is accessed after we rotate RGX one more place to the right so that it contains the pattern 2, 3, 0, 1. However the 3rd column must be rotated two places to the left to get it back in the proper order.

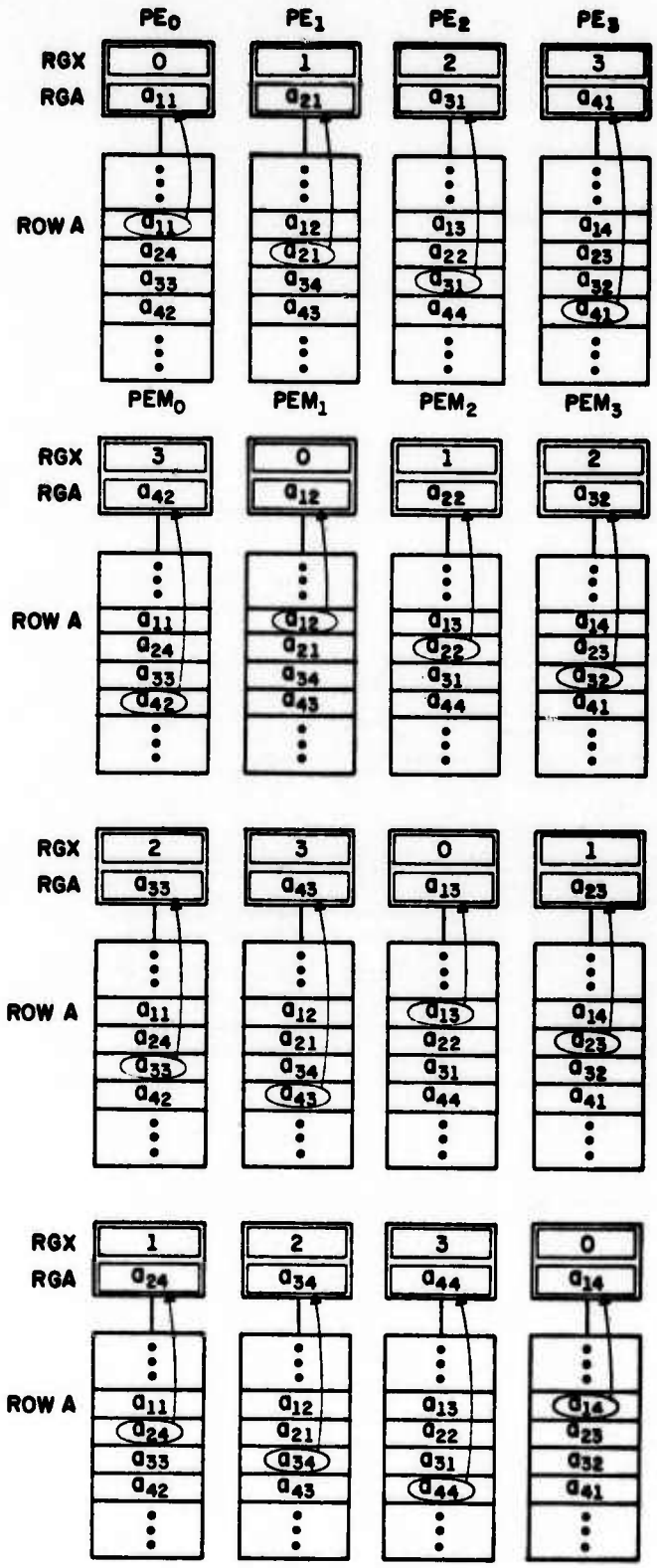


Figure III-5(a)

RGA contains Column 1 of Matrix A (which has been stored skewed) after the instruction LDA *A; has been executed. Circled elements are accessed by the instruction.

Figure III-5(b)

RGA contains Column 2 of Matrix A after LDA *A; has been executed. RGA must be rotated one place to the left if the column is to be used in a transpose.

Figure III-5(c)

RGA contains Column 3 of Matrix A after LDA *A; has been executed. RGA must be rotated two places to the left if the column is to be used in a transpose.

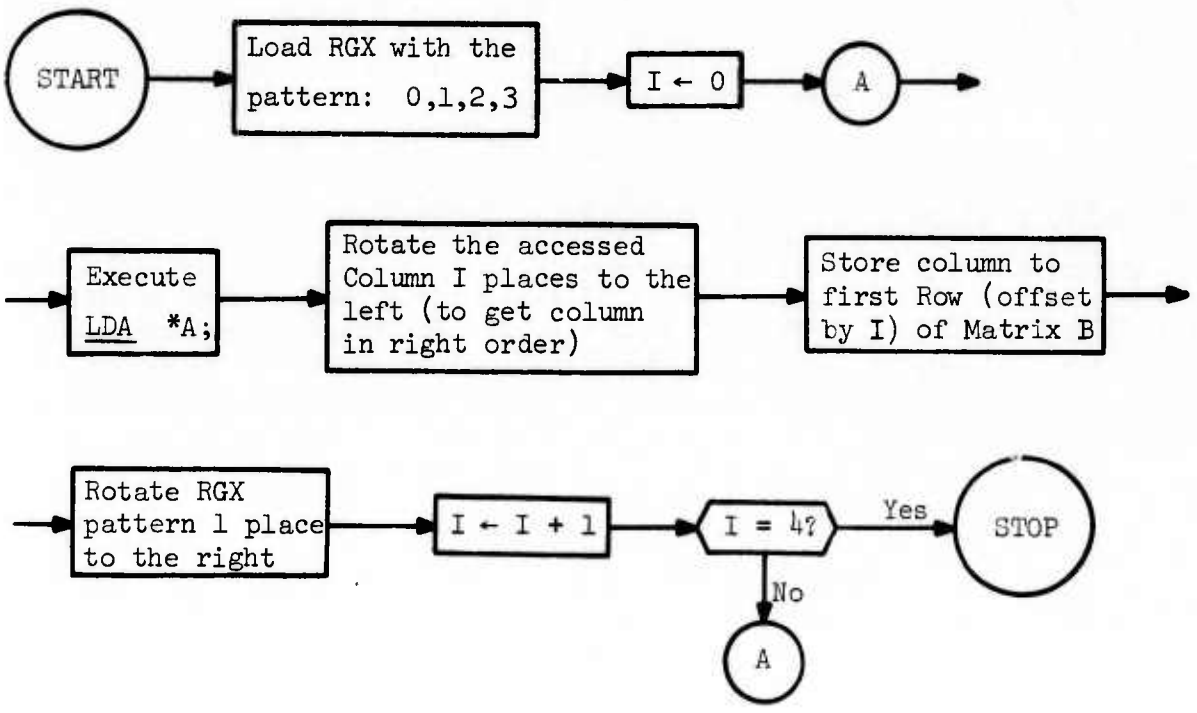
Figure III-5(d)

RGA contains Column 4 of Matrix A after LDA *A; has been executed. RGA must be rotated three places to the left if the column is to be used in a transpose.

Figure III-5. Skewed Storage is Used to Simultaneously Access Columns as well as Rows

Finally the 4th column is accessed as shown in Figure III-5(d) when the pattern in RGX is rotated once more to the right (so that it contains 1, 2, 3, 0) and the LDA *A; is executed. As might be expected, the 4th column must be rotated three places to the left to restore the proper order.

Now that we have a method of storing our matrix so that both columns and rows can be accessed simultaneously let us consider the flow-chart necessary to transpose our 4 x 4 matrix A, storing the transpose in B:



Let us now write the ASK code to transpose the 64 x 64 matrix A and store the result in the matrix B. The flow chart indicates that there are five basic instructions we want to loop through:

```
1.  LDA    *A;
2.  RTL    $A,-D;
3.  STR    B+D;
4.  RTL    $X,1;
5.  LDX    $R;
```

where we wish D to take on the values 0, 1, 2 ... 63 as we traverse the loop. Assuming that we have already loaded RGX with the proper pattern, the first instruction accesses a column of Matrix A. The second instruction routes the column back to the proper order. The third instruction stores the column to the appropriate Row of B. Instructions 4 and 5 rotate RGX one place to the right by Routing, then reload RGX with the routed pattern. (Remember the result of a ROUTE appears in RGR only.)

The following ASK instruction will transpose Matrix A and store the result in Matrix B assuming A has been stored skewed as shown in Figure III-6.

```
1          LDX        XROW;
2          LIT(0)     1,63,0;
3          LIT(1)     =64;
4  BEGIN:  LDA        *A;
5          RTL        $A,0(1);
6          STR        B(0);
7          ALIT(1)    =77777777:8;
8          RTL        $X,1;
9          LDX        $R;
10         TXLTM(0)   ,BEGIN;
```

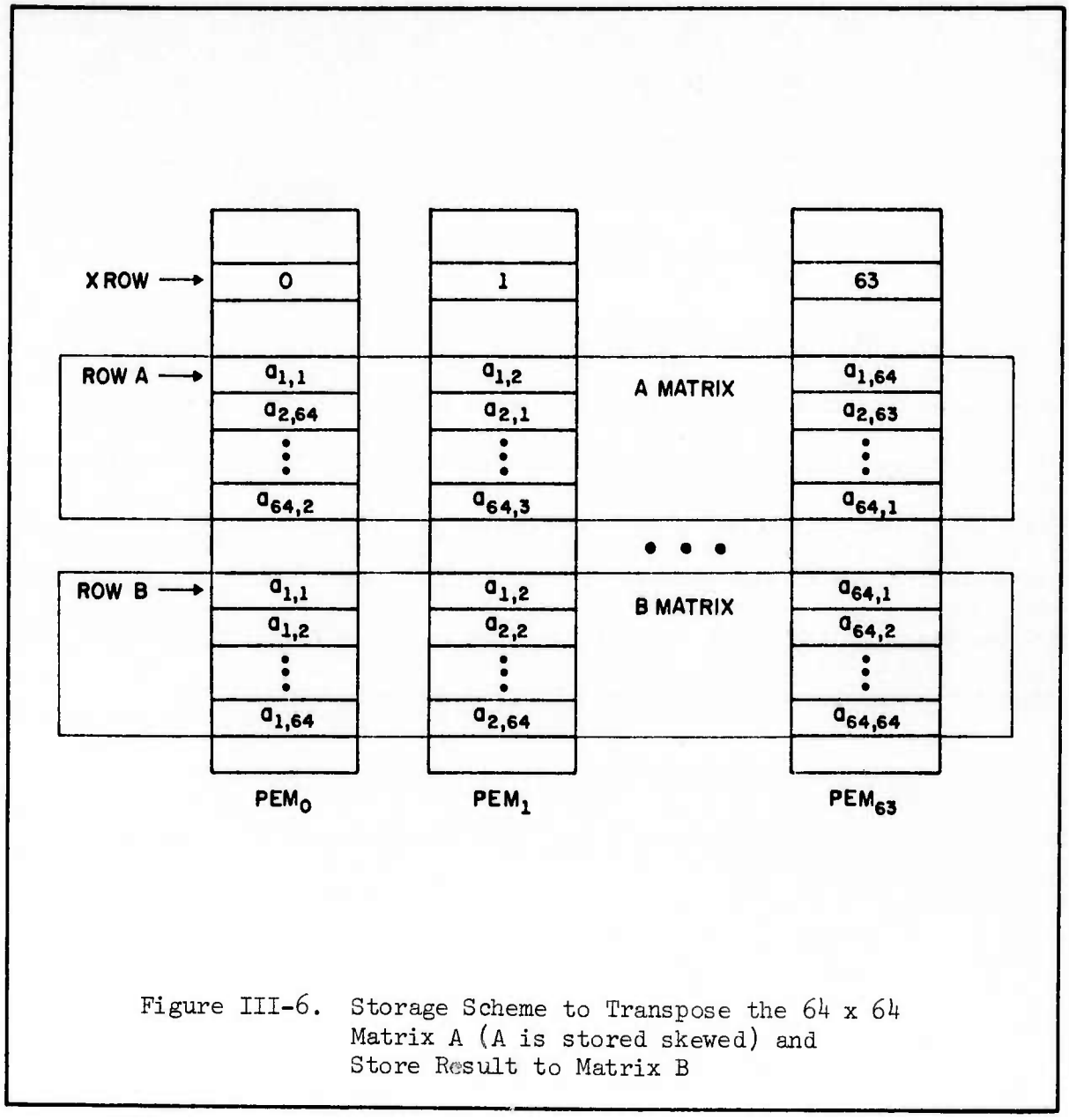


Figure III-6. Storage Scheme to Transpose the 64 x 64 Matrix A (A is stored skewed) and Store Result to Matrix B

Instruction 1 loads up RGX with the initial pattern of values (0, 1, 2 ... 63) to be used to offset the reference to Row A at instruction 4. Instruction 2 sets up the loop control variables in $C0 so that the 64 columns of A are stored to the 64 rows of B. Instruction 3 initializes the Routing Distance to be used in Instruction 5. Since we

wish to perform left routes and we should never do this by using negative numbers in an ACAR to be used as an index (see warning on page III-20) we use the fact that Routes are end-around so that a left Route of I can be accomplished by a right Route of 64 - I. Specifically:

A Left Route of 0 is equivalent to a Right Route of 64

A Left Route of 1 is equivalent to a Right Route of 63

⋮

A Left Route of 63 is equivalent to a Right Route of 1

Since we wish to start with a left Route of 0 (the first column is already in proper order) and then increase the Routing Distance by one each time through the loop, we start with a Right Route of 64 and decrease that value by one (at Instruction 7) each time we traverse the loop.

Instruction 4 accesses a column of Matrix A.

Instruction 5 puts the column in proper order.

Instruction 6 stores the column to the appropriate Row of B.

Instruction 7 decreases the Routing Distance stored in $C1 by one. (An ALIT(1) =-1 would not work since the -1 would be generated in the sign-magnitude representation. What we need for our problem is a minus one in two's complement notation so that as it is added to the low order 24 bits of $C1, the integer 64 will become 63 then 62, etc. as the loop is traversed.)

Instruction 8 rotates RGX one place to the right.

Instruction 9 places that new pattern back in RGX.

Instruction 10 loops the program back to Instruction 4 where the next column is accessed.

After control has passed through Instruction 10, the 64 x 64 Matrix A has its transpose stored in the Matrix B.

Now that we know how to use skewed storage to access columns as well as rows of a matrix, a new matrix multiplication algorithm could be designed: If one of the two matrices to be multiplied were stored skewed, then a column of that matrix could be accessed and multiplied by a row of the other matrix. The result would be in RGA of the PE array and if these values were summed (using a Logsum algorithm) then one element in the product matrix has been formed. This approach to matrix multiplication is not as efficient as the one we developed in section I 3; however, the reader should attempt by himself to write the ASK instructions that will perform this algorithm.

5. Temperature Distribution on a Slab

In Chapter II we discussed a method of solution to the boundary value problem of temperature distribution on a slab (see Section E of Chapter II). On page II-46 the basic equation for the relaxation method is given:

$$U_{i,j} = \frac{U_{i-1,j} + U_{i,j+1} + U_{i+1,j} + U_{i,j-1}}{4}$$

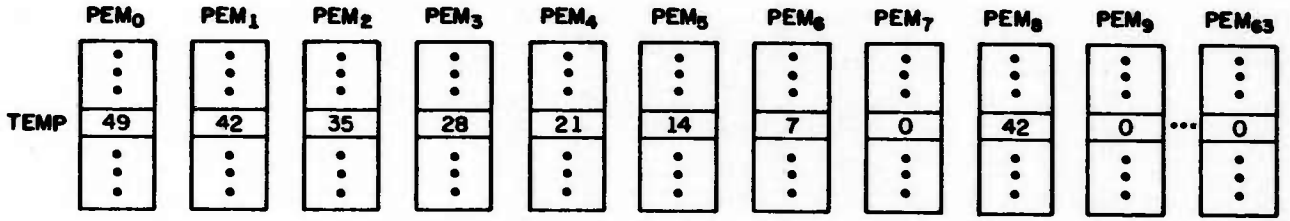
which tells us that the temperature at any point should be equal to the average of the temperatures at the 4 closest neighbors.

We shall develop the ASK code necessary to provide a solution for two cases: The first case will relax an 8 x 8 array of mesh points with initial conditions as shown in Figure II-21 on page II-48, using only one Row of PE Memory. For the second case, we will solve the same 8 x 8 array with the same initial conditions, but we shall use 8 Rows of PE Memory. (See Figure III-7.) Figure III-8 shows the exact solution for either case. Case 1 represents the most efficient solution to the problem since it utilizes all 64 PEs but Case 2 is presented to show how the data allocation influences the program necessary to process the data; i.e., the program that processes the data as allocated in Case 1 cannot be used to process the data as allocated in Case 2. Case 2 is also representative of the type of solution necessary for a problem with more than 64 mesh points.

a. Case 1. One Temperature per PEM

Before presenting the ASK program, we shall have to learn a few more ASK instructions. We will need a method of disabling the "border" PEs during the calculation since they represent the boundary values for the temperatures and must remain at a constant value throughout the

Case 1: One Temperature per PEM



Case 2: Eight Temperatures per PEM

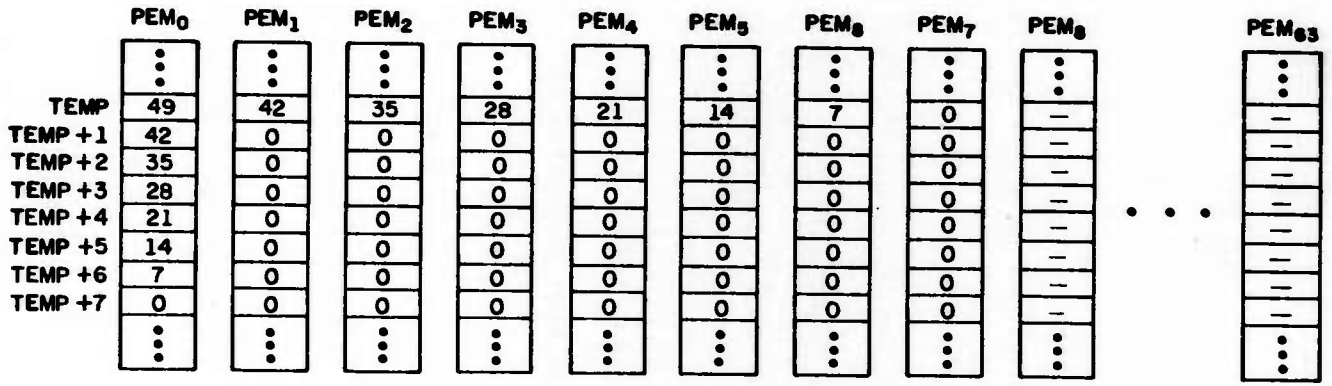


Figure III-7. Initial Conditions and Storage Allocation for Two Cases of Temperature Distribution on a Slab

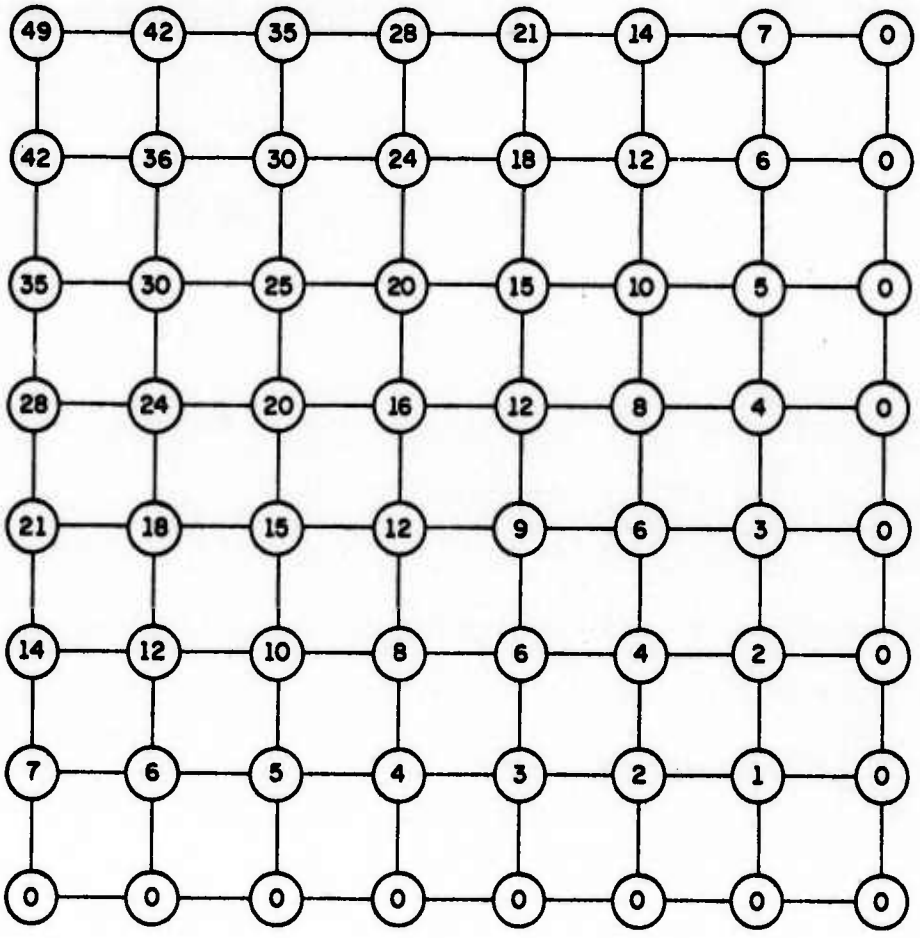


Figure III-8. Exact Solution for both Case 1 and Case 2

relaxation process. For Case 1 we are allocating the two-dimensional array of temperatures across one PE Row, thus the following PE numbers must be disabled:

0,	1,	2,	3,	4,	5,	6,	7
8,							15
16,							23
24,							31
32,							39
40,							47
48,							55
56,	57,	58,	59,	60,	61,	62,	63

The LDEE1 instruction will load the E and E1 bits with the bit pattern which was previously stored in a specified ACAR by a LIT instruction:

```
LDEE1        ACAR Number;
```

where *ACAR Number* can be $C0, $C1, $C2, or $C3 and specifies which ACAR contains the bit pattern that is used to set the E and E1 bits of RGD for every PE in the array.

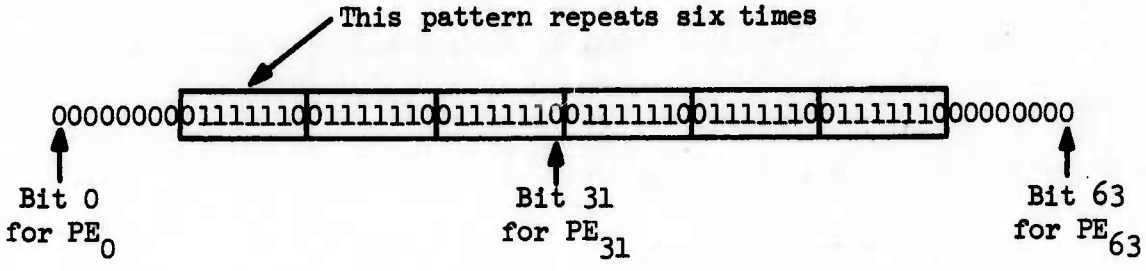
An ACAR contains 64 bits and there are 64 PEs in the array. If bit i of the specified ACAR is one then the E and E1 bits of RGD of PE_i are set to one, enabling that PE. If bit i is zero then the E and E1 bits of PE_i are set to zero, disabling that PE. LDEE1 is a FINST/PE instruction

and uses the Common Data Bus to transmit information from the specified ACAR to each RGA of the array as described in Chapter II section B 4 b.

Example:

```
LIT(0)    =007E7E7E7E7E7E00:16;
LDEE1     $C0;
```

The LIT instruction will load the hexadecimal (base 16) constant shown above into $C0. That hexadecimal constant can be written in binary and it becomes clear that the pattern of ones and zeros is the one we need to disable the border PEs for Case 1:



then execution of

```
LDEE1     $C0;
```

uses the above bit pattern within $C0 to set the E and E1 bits.

If the programmer wished to enable all PEs he could use

```
LIT(0)    =17777777777777777777:8;
LDEE1     $C0;
```

or

```
SETE      E.OR.-E;
SETE1     E.OR.-E;
```

We shall also want to apply the epsilon convergence criterion as described on page II-51.

$$|U^{n+1}_{i,j} - U^{n}_{i,j}| \leq \epsilon \qquad \text{for } 2 \leq i \leq 7, \quad 2 \leq j \leq 7$$

This means that we will stop the iteration when succeeding values of calculated temperatures differ by only a prescribed ϵ. This condition must be true for all temperature values simultaneously. We choose ϵ to be 1 degree as we did in Chapter II. Also, to insure that we finish the program within a finite time, we shall include a loop control that will end the calculation after 50 iterations regardless of whether the epsilon convergence criterion is met. In order to apply the epsilon convergence criterion we learn the following instructions:

SBRN *Operand;*

SBRN will subtract from the contents of RGA the value specified by *Operand* and place the result, rounded and normalized back in RGA. *Operand* is usually a Literal, a PE register, an ACAR, or a PE Memory Address. SBRN is a FINST/PE instruction.

Example:

```
SBRN    $S;
```

will subtract the contents of RGS from RGA and place the result back in RGA.

In order to take the absolute value of a quantity we have the simple instruction:

```
SAP    ;
```

which sets the sign bit of RGA to positive (+).

Example:

```
SAP    ;
```

If RGA contained a negative number before execution of this instruction, it will be made positive. If RGA contained a positive number, it will be unchanged.

Finally we shall require an instruction that can simultaneously sense the contents of a specified bit of RGD for all PEs and branch if they are all zero. We will need then an instruction of this type to implement the ϵ convergence criterion. This instruction is ZERT. First, however we must take all 64 of the specified bits and place them in an ACAR before ZERT can test them; we do this with the SETC instruction:

```
SETC (ACAR Number)      Mode Bit;
```

where *ACAR Number* can be 0, 1, 2 or 3 denoting $C0, $C1, $C2 or $C3 and specifies which ACAR will be set with the values of *Mode Bit* for all PEs in the array.

Mode Bit specifies one of the eight mode bits of RGD and can be E, E1, F, F1, G, H, I or J.

SETC works somewhat like LDEE1 in reverse: If the *Mode Bit* of PE_i is one, then bit i of the specified ACAR is set to one; if the *Mode Bit* of PE_i is zero, then bit i of the specified ACAR is set to zero. SETC is an ADVAST instruction and uses the Mode Bit Line discussed in Chapter II section B 4 d.

Example:

```
SETC(3)    I;
```

will set the 64 bits of $C3 to the corresponding 64 values of the I bit in RGD of all 64 PEs in the array.

ZERT can now test the contents of the ACAR set by SETC and branch if all bits are zero:

```
ZERT (ACAR Number)     ,Location;
```

will cause a jump to *Location* if every bit of the ACAR specified by *ACAR Number* is zero. *Location* must be ±127 of the ZERT instruction. ZERT is an ADVAST instruction.

Example:

```
1    IAG       $C0;
2    SETC(3)   I;
3    ZERT(3)   ,OUT;
```

The first instruction, IAG, will set the I bit to one for every PE whose RGA is greater than the contents of $C0 and set the I bit to zero otherwise. Instruction 2 will then transmit the I bits of the PE array to $C3. Instruction 3 will jump to location OUT if all of the I bits were zero, and will execute the next instruction otherwise. Thus, if RGA $\leq$ $C0 for every PE in the array, Instruction 3 will jump to OUT.

The ZERT instruction sometimes necessitates the writing of a HALT instruction if the jump is to be made to the last statement of the program. HALT is of the form:

```
HALT    ;
```

and stops the program from executing. HALT is an ADVAST instruction.

Example:

```
        ZERT(3)     ,OUT;
        TXLTM(1)    ,LOOP;
OUT:    HALT;
```

If the contents of $C3 are all zero then a jump is made to location OUT where the program ends; otherwise a transfer is made to LOOP based on the contents of $C1.

We shall introduce one more notation before writing the ASK code for Case 1: The percentage sign (%) in an ASK statement signifies that a comment to the reader is about to follow. The ASK assembler will not interpret any character on a card following a % sign. We assume the initial value of the temperatures are stored in location TEMP as shown in the upper portion of Figure III-7.

```
        SETE     E.OR.-E;
        SETE1    E.OR.-E;     % Enables all PEs
        LIT(0)   =007E7E7E7E7E7E00:16;
        LDS      TEMP;        % Do this before disabling
        LIT(1)   1,50,1;      % Set up max of 50 relaxations
        LDA      =0;          % Make sure that PEs to be disabled have
                                 a number less than EPS in their RGA
                                 so ZERT will work right later
        LDEE1    $C0;         % Disable border PEs
LOOP:   RTL      $S,1;        % Get value from left neighbor
        LDA      $R;          % Place in RGA
        RTL      $S,-1;       % Get value from right neighbor
        ADRN     $R;          % Add to RGA
        RTL      $S,8;        % Get value from top neighbor
        ADRN     $R;          % Add to RGA
```

```
        RTL     $S,-8;      % Get value from bottom neighbor
        ADRN    $R;         % Add to RGA
        LIT(3)  =0.25;      % Place constant of 1/4 in $C3
        MLRN    $C3;        % Divide by 4; relaxation done.
                              (New value is in RGA)
        LDR     $A;         % Save new value in RGR
        SBRN    $S;         % Subtract old value in RGS from
                              new value in RGA
        SAP     ;           % Take absolute value of New - Old
        LIT(0)  =1.0;       % Set up EPS value of one degree
        IAG     $C0;        % Set I to one if ABS (New - Old) is
                              greater than one degree
        SETC(3) I;          % Transmit I bit pattern to $C3
        ZERT(3) ,OUT;       % Jump out if no I bits are one
        STR     TEMP;       % Otherwise store new value back
                              in TEMP
        LDS     $R;         % Also place new value in RGS
        TXLTM(1) ,LOOP;     % and jump back to LOOP for start of
                              next relaxation.
OUT:    HALT;
```

Although the comments attempt to explain the operation of our program there are a few points that should be discussed:

The reader may have noticed that we multiplied by 0.25 rather than dividing by 4. This is a good programming trick to remember since ILLIAC IV can multiply two 64 bit floating point values about 6 times

as fast as it can divide them. Another apparent inefficiency may have caught the reader's attention in this same area of the program:

Why is the constant 0.25 being created in this particular part of the program--it is in the middle of a loop of calculations; shouldn't the

```
LIT(3)    =0.25;
```

be performed outside this loop so that it is done once and not up to 50 times?

Although this is a valid argument for a conventional computer it is not for ILLIAC IV, because CU instructions in ADVAST can be executed concurrently with PE instructions and since LIT is a CU instruction imbedded in a loop of PE instructions it is actually more efficient to leave it where it is because it requires literally no time for execution within a PE instruction loop (its execution is completely overlapped with PE instructions); it would require a small amount of time to execute if it were at the beginning of the program in a place where there was no opportunity for overlap of instruction execution.

Another programming hint to remember is the following:

Occasionally a programmer will write a set of instructions that modify fields of other instructions (usually the address field). That type of code must be used with extreme care on ILLIAC IV because of the 128 word Instruction Word Stack (IWS) which acts as a buffer to store

impending instructions to be executed. The instruction to be modified will have its image in memory modified but will not be affected itself by the instruction if it is already in the IWS. The next time through that set of instructions, however, it will be modified causing possible strange behavior of the program. The programmer can not modify an instruction within 128 instructions of the modifier instruction and expect it to work when control reaches the modified instruction the first time.

Two final comments on the program:

The LDA =0; (the sixth instruction) is necessary since the IAG $C0; instruction later acts on all of the I bits in the array, since RGD is not protected. We can insure that the I bits in the disabled PEs (the ones which contain the edge temperatures) get set to zero by placing any value less than 1.0 (such as 0) in their RGA before the calculation begins.

It was not necessary to store the newly computed values back in TEMP (using the STR TEMP; instruction) until OUT is reached, however, that choice was made so that TEMP could be displayed as the calculation progressed. (See Case 1 of Table III-4.)

b. Case 2. Eight Temperatures per PEM

For this approach to the problem, the data is allocated starting at Row TEMP as shown in the lower portion of Figure III-7. The main differences in this approach are:

1) The PE disabling pattern must be changed to reflect the different storage allocation of the temperature values.

2) The left and right neighboring values can be received using the ROUTE instruction but the above and below values must be handled differently since they will be in the same PE Memory. If we are referring to Row TEMP then TEMP - 1 and TEMP + 1 as PE Memory Addresses will reference the above and below values respectively.

3) Since we have allocated the data as 8 PE rows we shall have to perform 6 iterations (the top and bottom rows contain boundary temperatures and do not change) to complete one relaxation instead of just 1 iteration for 1 relaxation as we did with Case 1. This means that we shall need 2 loops; one to step us down from row 2 to row 7 (TEMP + 1 to TEMP + 6) for a given relaxation, and one to step us to the next relaxation.

4) We shall leave out the epsilon convergence test in the interest of simplicity (it can be done in exactly the same way as we did it for Case 1).

An explanation of the finer points will follow the ASK code for Case 2:

```
SETE        E.OR.-E;
SETE1       E.OR.-E;
LIT(0)      =7E00000000000000:16;  % Set up enable bit pattern
```

```
        LIT(2)    1,50,1;      % Set up max of 50 relaxations
LAAP:   LIT(1)    1,6,1;       % Set up iteration counter I to step
                                   from row I = 2 (TEMP + 1) to
                                   row I = 7 (TEMP + 6)
        LDEE1     $C0;         % Disable border PEs and rest of PEs
                                   not in calculation
LOOP:   LDR       TEMP(1);     % Load RGR from Row I of TEMP
        RTL       $R,1;        % Get value from left neighbor
        LDA       $R;          % Place in RGA
        RTL       $R,-2;       % Get value from right neighbor
        ADRN      $R;          % Add to RGA
        ADRN      TEMP-1(1);   % Add in value of top neighbor
                                   (Row I - 1)
        ADRN      TEMP+1(1);   % Add in value of bottom neighbor
                                   (Row I + 1)
        LIT(3)    =0.25;       % Place constant of 1/4 in $C3
        MLRN      $C3;         % Divide by 4.  Iteration on Row I
                                   complete
        STA       TEMP(1);     % Store Row values in RGA back to
                                   Row I of TEMP
        TXLTM(1)  ,LOOP;       % Go back and pick up next row
        TXLTM(2)  ,LAAP;       % One relaxation (all rows done) is
                                   complete.  Perform next
                                   relaxation.
```

Comments:

1) The 3rd instruction

```
LIT(0)    =7E00000000000000:16;
```

sets the bit pattern which will be used to disable PE_0, enable PEs 1 through 6 and disable PE_7. The rest of the PEs do not enter into the calculation but are disabled in case of an arithmetic fault (underflow or overflow) occurring in a PE that is not used in the calculation but might contain strange data from a previous user.

2) The 7th instruction

```
LOOP:  LDR    TEMP(1);
```

will load the Row specified by $C1 into the RGR of all the PEs (regardless of whether or not they are enabled at this point in the execution). The starting value field of $C1 acts as an index register to step down through the rows of TEMP--the first time through the LOOP, $C1 has a value of 1 and Row TEMP + 1 (the second row) is referenced by this instruction.

Since we use RGR as a temporary storage register and as the Routing register, it is necessary to route values a little differently than we did in Case 1. The first route of one to the right allows each PE to receive a value from its left neighbor; but when a PE needs the value from its right neighbor it has already been shifted one to the right by the first ROUTE so that a left shift of two is necessary. This type of routing could also have been used to implement the algorithm for Case 1.

3) Note that PE_0 and PE_7 are disabled using the LDEE1 $C0; instruction but the 1st row and 8th row are effectively "disabled" (they

do not change during the calculation) by stepping the row counter from Row 2 to Row 7 instead of from Row 1 to Row 8.

4) The 12th and 13th instructions

```
ADRN    TEMP-1(1);
ADRN    TEMP+1(1);
```

also use $C1 to pick the appropriate Row. Note that the Operands are PE Memory Addresses and specify a Row. The ACAR indexing appears after the Operand.

Table III-4 shows intermediate values of the temperatures at one, ten and fifty relaxations; the exact solution is shown in Figure III-8 and at the bottom of Table III-4. As expected Case 2 shows a closer convergence to the exact solution for the same number of relaxations since it iterates one Row at a time and each new Row gets the benefit of values computed in the Row above it, while Case 1 relaxes the whole array in one iteration. Even closer would be the sequential solution for a conventional computer as shown in Chapter II. However, Case 1 performs 6 times fewer calculations and is therefore about 6 times as fast as Case 2 and is about 36 times as fast a sequential method.

Table III-4. Comparison of Results for Case 1 and Case 2 for Temperature Distribution on a Slab

Case 1

One Relaxation

49	42	35	28	21	14	7	0
42	21.	8.75	7.00	5.25	3.50	1.75	0
35	8.75	0.	0.	0.	0.	0.	0
28	7.00	0.	0.	0.	0.	0.	0
21	5.25	0.	0.	0.	0.	0.	0
14	3.50	0.	0.	0.	0.	0.	0
7	1.75	0.	0.	0.	0.	0.	0
0	0	0	0	0	0	0	0

Ten Relaxations

49	42	35	28	21	14	7	0
42	34.3	27.1	20.5	14.8	9.6	4.7	0
35	27.1	19.9	14.1	9.5	5.9	2.8	0
28	20.5	14.1	9.1	5.6	3.2	1.5	0
21	14.8	9.5	5.6	3.1	1.6	0.7	0
14	9.6	5.9	3.2	1.6	0.7	0.3	0
7	4.7	2.8	1.5	0.7	0.3	0.1	0
0	0	0	0	0	0	0	0

Fifty Relaxations

49	42	35	28	21	14	7	0
42	35.98	29.96	23.96	17.96	11.96	5.98	0
35	29.96	24.94	19.92	14.92	9.94	4.96	0
28	23.96	19.92	15.90	11.90	7.92	3.96	0
21	17.96	14.92	11.90	8.90	5.92	2.96	0
14	11.96	9.94	7.92	5.92	3.94	1.96	0
7	5.98	4.96	3.96	2.96	1.96	0.98	0
0	0	0	0	0	0	0	0

Case 2

49	42	35	28	21	14	7	0
42	21.	8.75	7.	5.25	3.5	1.75	0
35	14.	2.19	1.75	1.31	0.88	0.44	0
28	10.5	0.55	0.44	0.33	0.22	0.11	0
21	7.88	0.14	0.11	0.08	0.05	0.03	0
14	5.47	0.03	0.03	0.02	0.01	0.01	0
7	3.12	0.01	0.01	0.01	0.00	0.00	0
0	0	0	0	0	0	0	0

49	42	35	28	21	14	7	0
42	34.8	27.9	21.5	15.6	10.2	5.0	0
35	28.1	21.6	16.0	11.1	7.0	3.4	0
28	21.9	16.3	11.5	7.7	4.7	2.2	0
21	16.0	11.7	8.0	5.1	3.0	1.4	0
14	10.6	7.6	5.1	3.2	1.8	0.8	0
7	5.3	3.8	2.5	1.6	0.9	0.4	0
0	0	0	0	0	0	0	0

49	42	35	28	21	14	7	0
42	36.00	29.99	23.99	17.99	11.99	6.00	0
35	29.99	24.99	19.99	14.99	9.99	4.99	0
28	23.99	19.99	15.98	11.98	7.99	3.99	0
21	17.99	14.99	11.98	8.98	5.99	2.99	0
14	11.99	9.99	7.99	5.99	3.99	1.99	0
7	6.00	5.00	3.99	2.99	2.00	1.00	0
0	0	0	0	0	0	0	0

Exact Solution

49	42	35	28	21	14	7	0
42	36	30	24	18	12	6	0
35	30	25	20	15	10	5	0
28	24	20	16	12	8	4	0
21	18	15	12	9	6	3	0
14	12	10	8	6	4	2	0
7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

J. Conclusion

The instructions presented in this chapter

LDA	ADRN	SETG	LDL
LDB	RTL	SETH	LDEE1
LDR	LIT	SETI	SBRN
LDS	TXLTM	SETJ	SAP
LDX	CSHL	IAL	SETC
LDD	CSHR	IAG	ZERT
STA	LESST	SLIT	HALT
STB	SETE	ALIT	
STR	SETE1	MLRN	
STS	SETF	LOAD	
STX	SETF1	BIN	

are not the comprehensive set of ILLIAC IV instructions; neither are they completely described in many cases. They were presented only as the problems demanded them and not in any functional order. I hope, however, that the reader has acquired some of the flavor of ILLIAC IV assembly language and is now confident enough to try to master the complete language.

CHAPTER III -- REFERENCES

1. David M. Grothe, "A Macro-Assembler for ILLIAC IV," ILLIAC IV Document No. 200, Department of Computer Science Report No. 364. ILLIAC IV Project, University of Illinois at Urbana-Champaign, (December 1, 1969).

2. Reference Manual for ILLIAC IV Assembler (ASK). Paoli, Pennsylvania: Burroughs Corporation (IL4-PM2), March 17, 1969.

HARDWARE GLOSSARY

ACAR -- See Accumulator Register

Accumulator Register (ACAR) -- There are 4 ACARs in the ADVAST section of the Control Unit. Each is 64 bits long. They are called ACAR0, ACAR1, ACAR2, and ACAR3, and each acts like an accumulator on a conventional computer.

ADB -- See ADVAST Data Buffer

Advanced Station (ADVAST) -- ADVAST is one of the five sections of the Control Unit and processes all instructions as an ILLIAC IV program executes. If the instruction can be executed completely within the resources of the Control Unit it never leaves ADVAST; if however, the instruction involves driving the 64-PE Array it passes on to FINST. ADVAST consists of 4 ACARs, the ADB, a simple ALU and the ADVAST Instruction Register as well as thirteen other registers not dealt with in this book.

ADVAST -- See Advanced Station

ADVAST Data Buffer (ADB) -- The ADB consists of 64 registers within the ADVAST Section of the Control Unit. Each word in the ADB is 64 bits long and access time is about 60 ns. Each word in the ADB is labelled: D0, D1 ... D63.

ADVAST Instruction Register (AIR) -- AIR is the 32-bit instruction execution register of the ADVAST Section of the Control Unit. AIR causes instructions which can be completely executed within the resources of the Control Unit (ADVAST instructions) to happen. If the instruction involves driving the 64 PE Array, it is sent on to FINST.

AIR -- See ADVAST Instruction Register

ALU -- See Arithmetic and Logic Unit

Arithmetic and Logic Unit (ALU) -- That set of circuitry within an electronic computer that performs arithmetic (+, -, ÷, x) and logical (AND, NOT, OR) operations.

ARPA Network -- The ARPA Network is a group of computer installations located throughout the country but connected via high-speed (50,000 bits/sec) telephone lines. Member installations will share hardware and software resources of other members on the "Net".

Array Processor -- The Array Processor comprises 64 PEs and 64 PEMs. See Figure II-1.

BIOM -- See Buffer Input/Output Memory

Buffer Input/Output Memory (BIOM) -- The BIOM is a rate smoothing buffer placed between the B6500 Computer and the Disk File System. It consists of four PE Memories and provides 8192 words of 64-bit Storage. See Figure II-1.

B6500 -- See B6500 Computer

B6500 Computer (B6500) -- The B6500 is the control computer for the ILLIAC IV System. It holds part of the Operating System as well as the utilities, compilers, and assemblers for ILLIAC IV. See Figure II-1.

CDB -- See Common Data Bus

CDC -- See Control Descriptor Controller

Common Data Bus (CDB) -- The CDB is one of the four paths by which data flows through the ILLIAC IV array. It is one word (64 bits) wide and runs in one direction from the Control Unit to the 64 PEs. It may be used to "broadcast" operands to the 64 PEs.

Control Descriptor Controller (CDC) -- The CDC is part of the I/O Subsystem and controls the transmission of data and programs between the Disk File System and the ILLIAC IV Array. See Figure II-1.

Control Unit (CU) -- That part of the ILLIAC IV Array responsible primarily for driving the 64 PEs in their instruction execution but may be viewed as a small unsophisticated computer in its own right capable of executing ADVAST instructions. The CU consists of five functional sections: ADVAST, FINST, MSU, TMU and ILA.

Control Unit Buffer -- The Control Unit Buffer is part of the ILA section of the Control Unit. It is an 8 word (64 bits per word) buffer which feeds the Instruction Word Stack (IWS).

Control Unit Bus (CU Bus) -- The CU Bus is one of the 4 paths by which data flows through the ILLIAC IV Array. It is 8 words (512 bits) wide and runs in one direction from the 64 PEMs to the CU. The CU Bus can fetch instructions (not under programmer control) or data (under programmer control).

CU -- See Control Unit

CU Bus -- See Control Unit Bus

Data Communications Processor -- The Data Communications Processor supervises a set of remote terminals and is supervised by the B6500 Computer. The remote terminal capability will allow users to run ILLIAC IV programs remotely.

DFS -- See Disk File System

Disk -- See Disk File System

Disk File System (DFS) -- The DFS, as part of the ILLIAC IV I/O System, has the main responsibility in transmitting and receiving data and programs to and from the ILLIAC IV Array. Its capacity is 10^9 bits and its effective transmission rate is 10^9 bits/sec over 2 channels. See Figure II-1.

FDQ -- See Final Data Queue

Final Data Queue (FDQ) -- FDQ is part of the Final Queue (FINQ) of the Final Station (FINST) of the Control Unit. It is 64 bits long

and holds the address or operand part of instructions which drive the 64 PEs.

Final Instruction Queue (FIQ) -- FIQ is part of the Final Queue (FINQ) of the Final Station (FINST) of the Control Unit. It is 16 bits long and holds the operation code of instructions which drive the 64 PEs.

Final Queue (FINQ) -- FINQ is an 8 word, 80 bits per word, buffer in the FINST section of the Control Unit. FINQ consists of the Final Instruction Queue (FIQ) which is 16 bits long and the Final Data Queue (FDQ) which is 64 bits long. It stores instructions on a first-in, first-out basis which are to drive the 64 PE Array.

Final Station (FINST) -- FINST is one of the five sections of the Control Unit. If an instruction involves the driving of the 64 PE array, FINST generates the microsequences necessary for the instruction to happen. FINST consists of the Final Instruction Queue (FIQ) and the Final Data Queue (FDQ) collectively called the Final Queue (FINQ) and a PE Instruction Microsequence Generator.

FINQ -- See Final Queue

FINST -- See Final Station

FIQ -- See Final Instruction Queue

IAM -- See ILA Associative Memory

ICR -- See Instruction Counter Register

ILA -- See Instruction Look Ahead

ILA Associative Memory (IAM) -- The IAM is a hard-wired device within the ILA section of the Control Unit which senses if the next instruction to be executed (pointed at by ICR) resides in the Instruction Word Stack (IWS).

ILLIAC IV Array -- The ILLIAC IV Array comprises the Array Processor and the Control Unit. See Figure II-1.

ILLIAC IV Disk -- See Disk File System

ILLIAC IV I/O System -- The ILLIAC IV I/O System comprises the I/O Subsystem, the Disk File System (DFS) and the B6500 Computer. See Figure II-1.

ILLIAC IV System -- The ILLIAC IV System comprises the ILLIAC IV Array and the ILLIAC IV I/O System. See Figure II-1.

Input/Output Switch (IOS) -- The IOS is a switch which insures that only one device (the DFS or the possible Real Time Device) is transmitting to or from the ILLIAC IV Array. It is also a buffer between the DFS and the ILLIAC IV Array. See Figure II-1.

Instruction Control Path -- The 266 line Instruction Control Path comes from the FINST Section of the Control Unit and drives the 64 PEs in the execution of their instructions.

Instruction Counter Register (ICR) -- ICR is a 25 bit register in the ILA section of the Control Unit which holds the address of the next instruction to be executed.

Instruction Look-Ahead (ILA) -- The ILA is one of the five sections of the Control Unit. It is responsible for maintaining a steady flow of instructions to the ADVAST Instruction Register (AIR) in ADVAST.

Instruction Word Stack (IWS) -- The IWS is a buffer which is fed by the Control Unit Buffer in the ILA Section of the Control Unit. The IWS holds 128 ILLIAC IV instructions.

IOS -- See Input/Output Switch

I/O Subsystem -- The I/O Subsystem comprises the Control Descriptor Controller (CDC), the Buffer Input/Output Memory (BIOM), and the Input/Output Switch (IOS). See Figure II-1.

IWS -- See Instruction Word Stack. (IWS is also called "ILA Instruction Word Storage".)

Laser Memory -- Laser Memory is supervised by the B6500 Computer and can be considered as fourth-order storage in the ILLIAC IV System. It holds 10^{12} bits and access time ranges from 200 ms to five seconds. Transmission rate is 8×10^{6} bits/second over two channels.

Memory Logic Unit (MLU) -- Each PE Memory has an MLU that resolves conflicting accesses to that memory. There are 64 MLUs and they are driven from the MSU Section of the Control Unit.

Memory Service Unit (MSU) -- The MSU is one of the five sections of the Control Unit. The MSU resolves PE Memory access conflicts and sends appropriate signals to the 64 MLUs.

MLU -- See Memory Logic Unit

Mode Bit Line -- The Mode Bit Line is one of the four paths by which data flows through the ILLIAC IV Array. It is one bit wide and runs in one direction from the RGD of each PE to the ACARs in the Control Unit.

MSU -- See Memory Service Unit

PE -- See Processing Element

PE Instruction Microsequence Generator -- That part of FINST responsible for generating the microsequences for instructions which drive the 64 PE array.

PEM -- See Processing Element Memory

PE Memory -- See Processing Element Memory

Processing Element (PE) -- There are 64 PEs in the Array Processor of the ILLIAC IV Array. Each PE is a sophisticated Arithmetic and Logic Unit capable of performing a wide range of arithmetic and logical operations. A PE has six programmable registers but is devoid of control logic (except for certain data-dependent conditions) being driven by the Control Unit.

Processing Element Memory (PEM) -- There are 64 PEMs in the Array Processor of the ILLIAC IV Array. Each PEM consists of 2048 words at 64 bits per word. Average access time is approximately 350 ns.

Processing Unit (PU) -- A Processing Unit consists of a Processing Element (PE) plus a Memory Logic Unit (MLU) plus a Processing Element Memory (PEM) i.e. PU ≡ PE + MLU + PEM. There are 64 PUs in the Array Processor of the ILLIAC IV Array.

Processing Unit Cabinet (PUC) -- Each PUC holds 8 PUs. They are called PUC_0, PUC_1 ... PUC_7.

PU -- See Processing Unit

PUC -- See Processing Unit Cabinet

RGA -- RGA is the Accumulator Register of a PE and acts like an accumulator on a conventional computer. RGA is 64 bits.

RGB -- RGB is the B register of a PE and can be used for temporary storage, however it usually holds the second operand in a binary operation so it is not a safe place to store data. RGB is 64 bits.

RGD -- RGD is the D register or Mode Register of a PE and reflects the active or non-active status of the PE in one or two of its 8 bits. The bits are called E, El, F, Fl, G, H, I and J. Certain Mode Bits can be set based on arithmetic comparisons. Other bits can reflect fault and overflow conditions.

RGR -- RGR is the R Register or Routing Register of a PE and can be used for temporary storage; however RGR is also a port to exchange information between PEs, so it is not a safe place to store information. The RGR of PE_i is connected by routing lines directly to the RGR of PE_{i-1}, PE_{i+1}, PE_{i+8}, and PE_{i-8}. RGR is 64 bits.

RGS -- RGS is the S Register of a PE and its intended use is for temporary storage. RGS is 64 bits.

RGX -- RGX is the X Register or Index Register of a PE and operates like an index register on a conventional machine, modifying the address field of an instruction. RGX is 16 bits.

Routing Network -- The Routing Network is one of the four paths by which data flows through the ILLIAC IV Array, and consists of the 64 RGRs. Each RGR is connected to the RGR immediately to the left and right as well as to the RGR eight to the right and eight to the left. The connection is end-around so that the RGR of PE_0 is connected to the RGR of PE_{63} and vice-versa.

Test and Maintenance Unit (TMU) -- The TMU is one of the five sections of the Control Unit. It is connected to the operation maintenance panel and via the CDC of the I/O Subsystem can cause communication to occur between the B6500 and the ILLIAC IV Array.

TMU -- See Test and Maintenance Unit

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

3. REPORT TITLE

AN INTRODUCTORY DESCRIPTION OF THE ILLIAC IV SYSTEM

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Research Report

5. AUTHOR(S) (First name, middle initial, last name)

Stewart A. Denenberg

6. REPORT DATE

July 15, 1971

7a. TOTAL NO. OF PAGES

255

7b. NO. OF REFS

9

8a. CONTRACT OR GRANT NO.

USAF 30(602)-4144

b. PROJECT NO.

ARPA Order No. 788

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

ILLIAC IV Document No. 225

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

Dept. of Computer Science File No. 850

10. DISTRIBUTION STATEMENT

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION IS UNLIMITED (A)

Copies of this report may be obtained from the address in (1) above.

11. SUPPLEMENTARY NOTES

None

12. SPONSORING MILITARY ACTIVITY

Rome Air Development Center
Griffiss Air Force Base
Rome, New York 13440

13. ABSTRACT

Written specifically for an applications programmer, the book presents a tutorial description of the ILLIAC IV System. Volume 1 contains three chapters -- Background, Hardware Structure, and The Assembly Language--ASK, as well as a Hardware Glossary. Many illustrative problems are used to educate the beginner in the use of the ILLIAC IV System.

DD FORM 1 NOV 68 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Reference and Learning Manuals						
Texts; Handbooks						
ASK (Assemblers)						
Storage Allocation						
Computer Systems (General)						