Multi Associative Processor Systems Architecture*

by

Roger D. Arnold
University of Colorado
Boulder, Colorado

Report #CU-CS-051-74 August 1974

* This work was supported by NSF Grants #GJ-660 and GJ-42251.

I. Introduction

With operation speeds for sequential operations in high speed digital computers approaching the limits imposed by signal propogation times, it has been apparent for some time that radical improvements in computer systems power will require architectures that make extensive use of parallelism. In fact architectures for modern high speed systems already do exploit parallelism to a fair extent. In general, however, such parallelism applies only at a micro level, in areas such as instruction look-ahead and overlapping computations, to speed the execution of what remains a strictly sequential program.

If one is interested in achieving extensive high level parallelism, it is necessary to look to architectures that differ substantially from the traditional. This paper will develop and examine one such architecture.

There are many problems whose most "natural" solution algorithms involve extensive parallel computation [2, 5, 7]. In practice, algorithms for these problems have been formulated as sequential routines only to fit common architectures. The distinguishing feature of such problems is that they require some procedure to be executed a number of time, using different data for each execution. The procedure may be as simple as comparing the contents of a memory location to a fixed reference value during a memory search, or as complex as evaluating the consequences of choosing a particular action from among a set of alternatives in an artificial intelligence program. Frequently it will involve numerical operations on elements of arrays. In any case the nature of this type of problem is such that all executions could be carried out in parallel, provided only that a suitable architecture were available.

Although the concept is not especially new, it is only recently that operational systems oriented toward this type of problem have been built in even moderate numbers. Examples of such systems are the well-known ILLIAC IV computer and its predecessor SOLOMAN, the Goodyear Company's STARAN computer, and Bell Laboratory's PEPE system for radar signal processing [1, 5, 8, 9]. These systems differ radically from one another in the details of their design and operation, but share the central characteristic that a large number of processing elements (PEs) execute instructions concurrently under the direction of a single control unit (CU). This makes for execution which is potentially very efficient, as well as fast, since individual PEs need not duplicate the functions of instruction fetching and decoding. This advantage may be offset by poor utilization of individual PEs in actual applications, but in those cases where high utilization can be maintained, it is difficult to conceive of any conventional architecture which could match these systems in overall economy and speed of operations.

Processors of the above type are variously referred to as associative processors, array processors, or associative array processors. "Associative processor" refers most naturally to systems such as the STARAN computer, in which the primary orientation of the system is toward associative accessing and updating of data bases. "Array processor" is the more apt description for the ILLIAC IV, which is oriented toward vector arithmetic operations on data arrays. However there is not always a clear distinction between one type of system and the other -- or none at least that has been widely formalized. We will usually use the abbreviation AP to mean "associative

processor", but there will be cases in which it might as easily stand for "array processor". The ambiguity is not serious, since what is meant in any case is a system in which there is concurrent execution by multiple processing elements of a single instruction stream.

The type of system with which this paper is concerned is an extension of the concept of an AP with a single control unit to one having multiple CUs, each of which may be simultaneously active over some subset of elements from among a large pool of shared PEs. We refer to these as multi-associative processor, or MAP systems. A MAP system may be viewed as a multi-processing system in which each processor is an AP.

There are a number of points motivating an investigation of MAP systems. Many of them are analogous to arguments that might be applied to any multi-processing system, including most obviously increased computing power. There are also, in many cases, both conceptual and practical advantages in being able to organize jobs into systems of independent but co-operating processes. This is perhaps even more true when the processes are associative than when they are ordinary sequential processes. When a sequential process is interrupted, the processor merely switches the focus of its activity. The only cost associated with the idle period of the interrupted process is due to the memory it holds (assuming it remains in memory). An associative process, on the other hand, holds more than just memory. It also holds a number of processing elements, whose idleness during interruption may constitute loss of a significant fraction of total system computing power. They may, of course, be reassigned during the interruption, but only at the cost of what may be substantial overhead.

And reassignment is pointless if the new process already has its own PEs. With multiple control units available to handle parallel processes, individual processes will be more consistently active, and PEs will have a higher net utilization.

The above refers to utilization of allocated PEs -- those owned by a process and potentially active as long as the process itself is active. The existence of multiple control units also allows higher utilization of PEs in terms of the fraction of PEs allocated. AP jobs usually have "natural" solutions requiring specific numbers of PEs. In air traffic control, for instance, the number of PEs required would correspond to the number of objects on track. When the number of PEs required for the natural solution exceeds the number actually available, alternate solutions are still possible, but always at a cost in program clarity and efficiency. With multiple CUs, the number of PEs in the system may be large enough to handle all but the most demanding problems according to their "natural" solutions. At the same time, the operating system can take advantage of a job mix and multi-processing to maintain good overall utilization while running smaller problems. These advantages are, of course, typical of the types of advantages motivating any shared resource system.

The purpose of this paper is to develop a tentative architecture for a multi-associative processor system, identifying as much as possible the constraints that apply and the tradeoffs that might be made. The architecture will serve as the basis for the development of an interpretive MAP simulator, and as a takeoff point for further studies undertaken in this department (Computer Science) at the University of Colorado.

II. General Design Considerations

Figure 1 shows a block diagram for a general MAP system. Each control unit potentially corresponds to one active process. With m control units, there may be up to m independent processes concurrently running at any one time. For our studies, we have assumed $m \leq 8$.

Instructions for the active processes reside in a common central memory. Use of a common central memory facilitates inter-process communication as well as resulting in more efficient memory utilization. Each control unit contains one or more high speed buffers for instructions and data to be broadcast to its PEs, and the central memory uses interleaving, extended words, or a combination of the two in order to support the high bandwidths required by eight concurrently active CUs and I/O processor (cf. ref. [4]). Instructions go out from the CUs along instruction busses to the PEs, which then, if activated, execute the instructions using either local operands previously stored in their individual PE memories (PEMs), or global operands broadcast along with the instructions over the common PE input data bus.

There are several features of the system illustrated in figure 1 worth noting:

-- There are no direct data paths between individual PEs. All data communication is performed via common busses running to all PEs.

-- There are no physical characteristics of individual PEs which distinguish one from another, or restrict their ability to be interchanged.

-- External I/O is centralized via the I/O subsystem processor. All I/O is buffered through central memory, and individual PEs have no direct I/O channels.

-- Instruction busses are narrow (6 bits) and separate for each control unit. A PE "listens" only to the bus for the control unit to which it is assigned.

-- Data busses are wide (32 bits) and shared among the various CUs and their assigned PEs on a cycle by cycle basis. PE registers and operations are word oriented, rather than bit serial, and PE memory size is appreciable (1024 16-bit words).

The PE memory size of 1024 words corresponds to the address space of the 1024-bit IC chip now widely manufactured for mini-computer memories. While the selection of this precise value is somewhat arbitrary, it falls within a general range whose choice is not arbitrary. Nor are most of the other features described above arbitrary. They tend to follow from various considerations arising from the initial decision to design an associative processor with multiple control units. These considerations are discussed below.

A principle advantage of a multi control unit system, as discussed in Section I, is that PEs become a shared system resource. Dynamic allocation of PEs, however, requires that individual PEs be indistinguishable to the CUs using them, except, of course, in terms of their data contents. This is also very desirable in terms of system reliability, but it immediately rules out the use of "hard" parallel PE to PE data communication paths of the type employed in ILLIAC IV to achieve extremely high inter-PE

data communication bandwidths. The latter communication scheme requires a correspondence between the physical location of the PE within the array of PEs and its logical function within a program.

Dynamic allocation of PEs also tends to rule out the extremely simple, bit serial PEs of the type used in the STARAN computer. There is a certain amount of overhead involved in providing for the switching of PEs from one control unit to another, and individual PEs must be sufficiently powerful to justify this overhead. Moreover, a major part of the justification for bit serial PEs is lost in a MAP system. In a uni-control unit AP, each PE may have its own external I/O channel, and a bit-serial PE is well matched to the bit-serial I/O from one track of a head-per-track parallel I/O device. In a MAP system, this type of parallel I/O becomes unattractive. Because of the uncertainty as to which actual PEs will be available for a particular execution of a job, parallel I/O direct to the PEMs would require a very large switching matrix, allowing any channel to connect to whatever PE happened to be available. This problem does not arise in an ordinary AP, since the entire pool of PEs is available for each job. It is therefore feasible for individual PEs to be dedicated to individual channels.

The lack of direct parallel I/O to the PEs in a MAP system is not necessarily the handicap it might at first appear. Comparable I/O bandwidths for loading the unloading PE memories can be achieved through the use of a single high speed, word parallel bus operating between a central memory buffer and an interleaved set of PEs. Assuming an 80 ns cycle time for the bus and a width of 32 bits, the bandwidth is 4×10^8 bps, or

the equivalent of 4000 one-bit channels operating at the 10^5bps typical for direct channels to mass storage devices. Of course with the CM to PE bus it is necessary to get the data into central memory before it can be transmitted to the PEs. Imposition of the CM as a buffer for PE I/O, however, has the advantage of isolating the I/O subsystem from the AP parts of the system, and allowing it to be designed independently, according to the requirements of any particular installation, without affecting the design of the MAP system itself. PE I/O is discussed in more detail in Section VI. In that section are discussed also portions of the "auxiliary control systems" of figure 1 which relate to control of the data buss. Other portions relating to selection and counting of PEs are discussed in section V.

Of the features initially cited above, we have not yet discussed the instruction bus and the motivation for the design choosen. This, however, is a major topic and will be taken up in detail in the next section. Remaining sections will then consider selective activation of PEs, CU architecture and the co-ordination and control of concurrent processes, and management of communication busses and other shared facilities.

III. Processing Elements and Instruction Broadcast Lines

As mentioned earlier, one of the attractions of associative processors is that individual processing elements need not duplicate the functions of instruction fetching and decoding. In a conventional AP with a single control unit, the CU may broadcast instructions to the PEs as timed sequences of gating signals, resulting in the simplest possible PE design. In a MAP system this is not feasible. Control lines of this type cannot readily be shared among multiple CUs, and the large number of lines involved makes switching between independent sets of lines for each CU infeasible. However it is possible to broadcast instructions as encoded micro signals over a much smaller set of lines. In this case the switching overhead needed to allow dynamic allocation of PEs is not excessive. These signals do require decoding, but it is a much simpler decoding than that needed for higher level machine instructions. Additional PE costs due to the necessity of decoding are therefore held to a minimum. This approach does have the potential disadvantage of sequentializing some operations that might otherwise be carried out in parallel, since only one micro instruction may be broadcast at a time. However, with appropriate CU design and a careful choice of micro instructions, this effect can be masked, or at least kept to a minimal level. (See section V, on optimizing the stream of PE micro instructions.)

Given the above system for instruction broadcasting, the method of handling operands is straightforward. They are broadcast over the time-shared CU to PE data buss as ordinary data. At the same time, signals to

gate the operands into appropriate PE registers are broadcast over the instruction bus. These signals are elements of the PEs regular micro instruction repertoire. Before broadcasting an operand to its PEs, however, a CU must first reserve a cycle on the data bus by signaling to a central scheduler. Scheduler operation is discussed in section VI. The cycle time and width of the data bus must be such that it can deliver operands at a rate sufficient to satisfy all processes whose control units are concurrently active. The data bus is therefore a very critical subunit of a MAP system, and special techniques may be needed to increase its bandwidth and, coincidentally, to increase system reliability by providing a capability for "graceful degradation" in the case of partial failures in the bus. These techniques are also discussed in section VI; they consist, briefly, of splitting the bus into a number of shorter, semi-independent busses, with each segment having cross-bar connections to every CU.

Figure 2 is a block diagram of a processing element. It consists of three main subunits -- the associative unit, or AU; the arithmetic and logical unit, or ALU; and the PE memory, or PEM. The associative unit is responsible for determining which instructions from the broadcast instruction stream apply to that PE, and is discussed in section IV. The ALU performs all operations which test or manipulate data. The registers and switches shown in figure 2 act as interfaces between the ALU and the AU, PEM, INPUT bus, and OUTPUT bus. (The INPUT and OUTPUT busses of figure 2 together comprise what is labeled simply as the data bus in figure 1. The two halves of the data bus operate independently.

Each is 32 bits wide, and the INPUT bus may itself be split into two independent halfword busses for delivery of short integer or address operands.) The ALU is shown in somewhat greater detail in figure 3.

The architecture for the ALU is somewhat arbitrary, but is assumed to be comparable to that of a small, modern minicomputer without the circuitry for instruction fetching and decoding. We have assumed a PEM word size of 16 bits, partly for efficiency in storing integer and address operands, and in part simply to be compatible with commercial minicomputer memories. Registers are 32 bits, and may require two memory cycles to load. Arithmetic operands are either 16-bit integers or 32-bit floating point numbers, with a 24-bit fraction, 7-bit hexadecimal exponent, and high order sign bit. One's complement notation is assumed. Extended precision is provided through the accumulator extension register, labeled ACX in figure 3. Autonomous circuitry to perform floating point multiplication, division, normalization, and truncation is assumed. All two-operand arithmetic and logical operations (+, *, /, $\vee$, $\wedge$, and $\oplus$) apply between the contents of the accumulator and one of the three registers IDR (Input Data Register), RMR (Read Memory Register), or WMR (Write Memory Register), according to which of the three is currently selected as the second operand register. The use of separate registers for reading from memory, writing to memory, and receiving data from the INPUT bus is not strictly necessary, but it gives better utilization of ALU circuitry by allowing operations to overlap that would otherwise have to be done in strict sequence. For instance, the RMR may hold an operand which is currently being used in an arithmetic

operation, while the results of a previous operation, transferred to WMR, are being stored. The use of separate registers also provides an intermediate storage capability that helps to reduce the number of fetches and stores to local memory.

Table 1 is a list of the PE instructions associated with the ALU, which comprise the greater part of the PE's instruction repertoire. Most of them are reasonably self-explanatory. Most of them are either simple gating signals or signals to set or reset various flip flops controlling gates. In any case their execution takes only one cycle. The instructions labeled "extended operations" are a little different, in that they initiate action by the autonomous floating point circuitry. After initiation, the operation continues on its own until completion. Meanwhile the instruction bus is free to transmit other instructions that do not interfere with the extended operation. Note that addition is not listed as an extended operation. The ALU adder is permanently active between the AC and the selected operand register. The output of the adder can be gated to the AC at any time in one cycle, provided that the adder has had time to stabilize since the last change in either of its inputs. It is up to the CU to know when this has occurred, based on the instructions it has previously issued, and not to issue the gating signal too soon. Similar considerations apply for the logical sum, product, and difference of the AC and the selected operand register. The arithmetic difference must be obtained through a complement and add.

In computing the arithmetic sum of the AC and the selected operand register, the ALU circuitry does not consider the 7-bit exponent. The operation is essentially a 25-bit integer add (counting the sign bit), and

whatever exponent was previously present in the AC remains unaltered. If the numbers to be added are floating point numbers, then instruction 29 of table 1 must be issued to align the exponents of the two registers prior to the add. The adder does detect integer overflow, and sets an internal indicator that overflow has occurred. If a normalization instruction detects an integer overflow indicator, it automatically shifts the fraction right by four bits, decrements the exponent by 1, sets bit 20 to a "1" if the number is positive or a "0" otherwise, and sets bits 21-23 to the same value as the sign bit. If no indicator is set, normalization proceeds conventionally.

It may be noticed that individual PEs do not have index registers, per se. However the ability to load the MAR directly from the RMR (instruction 6) makes for efficient indirect addressing, giving comparable capabilities in those instances where individual PEs require distinct index values. For accessing arrays in the more common case where all PEs address the same relative word at any given time, a central CU index register eliminates the need for index registers in individual PEs.

The remainder of the PE's instruction set will be taken up in the next section following a general discussion of the problem of selective activation of PEs. (The instructions of Table 1 relating to the ICTL and OCTL registers are an exception. These registers and instructions are intended to facilitate I/O operations and are discussed in section VI.)

IV. Instruction Associativity

A commonly held opinion with regard to array processors is that they are of quite limited utility as general purpose machines, coming into their own only for a narrow class of special problems. Those would-be problems in which large masses of data are to be manipulated according to fixed patterns, with little or no conditional testing required. Classic examples of such problems would be solving systems of partial differential equations.

There is at least some justification to this belief. Every conditional test which must be applied splits the set of PEs into two distinct groups -- those for which the test holds true, and those for which it does not. A CU can broadcast only one stream of instructions at a time, and if those are not the instructions appropriate to the results of a test in a given PE, that PE must simply wait until the CU gets around to broadcasting the instructions appropriate to its particular set of conditions. If conditional tests are deeply nested, it is easy for the set of PEs to become fractionated to the point that the CU is serving an average of only one or two PEs at a time. One approach to this problem is to limit the applications of the AP to the straight-forward "number crunching" problems that avoid such highly data dependent processing. This is the approach taken with the ILLIAC IV. As D. J. Kuck notes [6], the range of problems which satisfy or can readily be cast into this format through programming techniques is considerably larger than it might at first appear. The approach is therefore not unusually

restrictive, and in any case there are more than enough problems of this type to justify -- at least in principle -- such a machine. The STARAN computer, on the other hand, represents an approach in which the reduction of the set of active PEs to a single PE through conditional tests is not only acceptable, but may be a principle objective of the program. Here the PEs are an integral part of the memory structure, and cheap enough that idleness is not a major concern. They serve primarily to implement associative memory and do parallel I/O, working collectively as an auxiliary unit to a mainframe sequential processor.

In terms of PE activity, as in other matters already discussed, the APs of a MAP system are intermediate between the extremes represented by the ILLIAC IV and the STARAN. Their PEs are simpler and more numerous than those of the ILLIAC, so that there is less need to select and tailor applications so as to keep them all busy. At the same time they are not so simple and numerous that they can be regarded simply as mechanisms for implementing associative memory. They are significant processors in their own right, and any devices which will help to achieve efficient utilization are likely to be justified. The role of multiple control units in achieving high PE utilization was discussed in the introduction. Of equal, though perhaps less obvious significance is the mechanism controlling selective activation of PEs according to the results of conditional tests. If the range of useful AP applications is to be significant in a MAP system, a more sophisticated selection mechanism is needed than those available in either the STARAN or the ILLIAC IV.

PE selection in the STARAN and the ILLIAC computers both involve a one-bit activity switch which can be set by the results of local data tests. Through the mechanism of this switch, PEs can be deactivated when the CU is broadcasting instructions not appropriate to their local conditions. This method is in principle adequate for any algorithm, and for the simplest it is straightforward and efficient. However it can become unwieldy for algorithms involving even moderately complex conditional execution structures, and may tend to obscure the natural structure of the algorithm with operations necessary to insure that the proper processors and no others are activated.

As an illustration of this problem, consider the flowchart of figure 4. All PEs are to perform the initial processing, P1, following which an accumulator register is tested to determine activation for the next portion of processing. Processors with accumulators greater than zero are activated for execution of processing step 2, while all others are deactivated. Following P2, a test is again made on the accumulator among active PEs, and those with a non-zero accumulator are activated for processing step 3. Now following P3, it is necessary to reactivate for step P4 those processors deactivated following the second test because of zero accumulators. At the same time, the set of processors deactivated after the first test because of zero accumulators must not be reactivated. There is, however, no immediate way to distinguish the two sets, and the algorithm fails.

Naturally, there are ways around the above problem. For instance, it would be possible to follow a policy of activating after a test the complement of the set ultimately desired, and storing in that set some type of identifier. The activation would then be reversed, and the next processing step would commence on its proper set of processors. For reactivation of old sets, then, all processors would be reactivated for a test of their set identifiers, and all of those not in the desired set would again be deactivated. This method is inelegant at best, and wastes a lot of time re-establishing conditions that are already "known" (in the sense that they have been previously established by the system and that no intervening steps could have altered them). This is mainly a consequence of having only one bit of information concerning processor status -- the activity switch -- available at an immediate hardware level. The problem is not actually unique to APs. It has an analog in ordinary sequential processors, where instructions, if encountered, must be executed. They cannot be conditionally skipped based on status information held in the processor except through explicit testing or retesting of flags and status data prior to their execution. In programs where efficiency is important and memory is not critical, programmers may take advantage of the so-called "space/time tradeoff" to eliminate such redundant tests by -- in effect -- storing status information in the instruction counter. Each conditional test directs the processor to a distinct section of code, and status information is preserved to the extent that a given section of code can only be reached as a result of a

specific sequence of test conditions. To store 4 bits of status information requires 16 distinct sections of code. Individual sections may be highly similar, differing only in one or two instructions. The differences, however, are critical.

In an AP, this type of space/time tradeoff does not work. For any conditional test, there are likely to be some PEs representing either result, so that all, or nearly all sections of code must eventually be executed. This actually results in an inverse to the usual space/time tradeoff. The more compact the code, the faster the program executes. (Or stated otherwise, the more compact the code, the higher the PE utilization due to code sharing.) This places a burden on the architecture, however, to provide an efficient means for storing status information in individual PEs. The mechanism must allow selective exeuction of instructions while eliminating redundant tests.

Such a mechanism is possible, and is illustrated in figure 5 for our hypothetical MAP system. Status information is stored in an 8-bit select register in each PE. Each instruction has associated with it two 8-bit tag fields called a key and a mask, which define the set of PEs to which that instruction applies, based on the contents of the select registers. For an instruction to apply to a given PE under normal selection procedures, the contents of the select register must match the key bitwise in all positions designated by "0" bits in the mask. There is also complement selection, in which mode the set of PEs to which an instruction applies is the complement of the set to which it would apply under normal selection.

As an example of how PE selection operates, suppose that bits 0, 1, and 2 of the select register (numbering from low order to high order, or right to left) record the boolean values of conditions designated as A, B, and C respectively. An instruction applying to PEs for which A and B are true but C is false can be designated by a key of "03_{16}" and a mask of "$F8_{16}$" under normal selection. An instruction applying to all PEs for which A was true or B was false -- with C irrelevant -- could be designated by a key of "01_{16}" and a mask of "FC_{16}" under complement selection. (The latter makes use of the logical equivalence between $A \vee \overline{B}$ and $\overline{(\overline{A} \wedge B)}$.) Translating boolean expressions for the set of PEs to which an instruction applies into appropriate keys and masks would normally be a compiler function and not a problem for the programmer. However the selection power of such "associative instructions" eliminates the need for much duplicated code and results in extremely compact programs that can execute efficiently on an AP regardless of the conditional dependency of individual instructions.

Although keys and masks apply conceptually to every instruction, it is necessary for the CU to broadcast them only when they change -- and perhaps not even then if the new key and mask have been recently used. The associative unit of the PE, shown in figure 6, contains two registers, K1 and K2, for storing the two most recently used combinations of key and mask. A third register, K0, is a virtual register corresponding to a permanently stored key/mast configuration of all zeros, defining the universal set of all PEs assigned to a given CU. If the new key and mask

are present in one of the registers, it requires only a single micro instruction to indicate which of the registers should currently control selection. These are instructions 55, 56, and 57 of table 2, and their operation is illustrated by the signal lines labelled with these numbers in figure 6. Likewise it takes only a single micro instruction to switch between normal and complement selection modes (instructions 53 and 54, also labelled as signal lines in figure 6). These features save considerable time in the relatively common case in which instructions apply in rapid succession between the universal set of PEs and sets designated by two recurring combinations of key and mask. In the case where the new key and mask are different from any stored in the K registers, instructions 49-52 allow new combinations to be loaded from either half of the INPUT bus.

The selection mechanism described above is of course useless without some means of storing conditional results into the select register in the first place. One of the two main mechanisms for doing this is shown in figure 7. There are seven switches, labelled SW1 thru SW7 in figure 7, six of which are also shown in figure 2 as interfaces between the associative unit and the ALU. (SW7 is a virtual switch corresponding to a defined true, or a permanently set "1".) These switches carry information on various conditions within the ALU. For instance, SW1 is set whenever the contents of the AC register are less than those of the selected operand register. SW2 is set when the two contents are equal, while SW3 being set indicates that the contents of the AC are greater. SW4 is a copy of the sign bit of the accumulator, and so is

set whenever the contents of the AC are negative. (This includes the case of a negative zero, arising due to the use of ones complement notation.) The meanings of SW5 and SW6 depend on the particular instruction last executed. Following a multiplication or divide, for example, SW5 is set to indicate floating point overflow, while SW6 is set to indicate underflow. Other uses will be discussed later.

The contents of the select register are altered by writing the value of a specified switch or its complement into all positions of the select register designated by "1" bits in an 8-bit write mask. This is instruction 42 of table 2. The write mask for this instruction is taken from the low order byte of the IDR during execution, while the switch designator comes from the nextmost low order byte. Bits 1-7 of this byte (bits 9-15 of the IDR) correspond to SW1 - SW7, respectively, while bit 0, if set, indicates that the complement of the indicated switch (or switches) is to be used. If more than one switch is indicated by having its corresponding bit set, the value written is the logical sum of all indicated switches, or their complements if appropriate. As an example, the select register may be cleared by issuing instruction 42 after loading the IDR with the 16-bit designator-mask combination of "$81FF_{16}$". The "81_{16}" corresponds to an 8-bit switch designator of "10000001", indicating that the complement of SW7, a defined "0", is the value to be written. The "FF_{16}" defines an 8-bit write mask of "11111111", indicating that the value is to be written into all eight positions of the select register. The same operation with a write mask

of "$0F_{16}$" would clear the four low order bits of the select register, while leaving the upper four unchanged.

With instructions 43-46 of Table 2, it is also possible to alter the contents of the select register based on its own prior contents. This corresponds to defining new boolean variables as logical functions of others previously defined. In this type of instruction, the low order byte of the IDR serves as a write mask just as above; the second byte, however, specifies not a switch or set of switches, but rather a set of bits from the select register itself which are to participate in the computation of the function. The function may be the logical sum or product of the indicated bits, or the complements of the same. Finally, instructions 47 and 48 of Table 2 allow the entire contents of the select register to be loaded from or stored to the accumulator register of the ALU. This allows processes to be interrupted when necessary and later restored.

The final group of instructions of Table 2 require some comment. It is frequently necessary to find which PE among a set has the highest or lowest value for some particular variable. Instructions 61 - 63 provide the basis for a fast means of doing so. The procedure involves a comparison in each PE between the value in its accumulator register and a value which is the logical sum of all the values participating in the comparison*. If the logical sum contains a "1" in a position preceding

---

*The logical sum is obtained by writing the accumulator values for all participating PEs to the OUTPUT bus simultaneously. The CU then rebroadcasts this value over the INPUT bus to all PEs participating in the comparison.

any in which a PE's own accumulator contains a "1", then some other accumulator must have contained a larger value. (It is necessary to bias the sign bit prior to the compare operation so that positive numbers will compare as larger than negative numbers.) One such operation does not in general find the element with the maximum value, but it may eliminate a number of contenders. Each PE contains a special associative compare cursor, which is initially set to the leftmost bit of the accumulator. During the compare, the cursor is advanced to the right until a position is found where the logical sum of all values contains "1", but the local accumulator value contains a "0". If the scan encounters such a "1/0" condition before the first "1/1" condition is found, then SW5 is cleared, indicating that the PE is no longer a candidate for having the maximum value. After the scan, each PE broadcasts a word over the OUTPUT bus consisting of a single "1" bit in the stopping location of its cursor. The logical sum of these words is returned over the INPUT bus to the IDR in each PE still active. The PE then checks for "1" bits to the right of its own cursor location. If any are found, then it knows that some other PE contains a value larger than its own, and SW5 is cleared. PEs with SW5 cleared then drop out, and the cycle is repeated for the remaining PEs. The process continues until the cursor for at least one PE has advanced all the way to the right, which the CU detects by finding a "1" in bit location 0 on the OUTPUT bus when the PEs broadcast their cursor locations. At the completion of the cycle in which that condition occurs,

any PEs with SW5 still set share the same maximum value. If it is necessary to further reduce this set to a single element, the PEs are instructed to signal to the count and selection unit, which then selects one of them and sets SW6 in that element, simultaneously resetting SW6 in all other elements which signaled to it.

The above procedure generally requires only a few cycles to find the element with the largest value in its accumulator. Since the cycles themselves are micro-level cycles, the whole procedure can be implemented as a single machine instruction, as viewed from the control unit instruction repertoire. A similar instruction to find the minimum value would operate by finding the maximum value of the complements of the values in the PE accumulators.

V. Control Units and Inter-Process Communication

Since they drive the remainder of the system, the control units are in many respects the most important elements of a MAP system. Yet from an architectural design viewpoint, they are perhaps the least critical elements as well. Given any reasonable amount of CU design effort, it is unlikely that CUs will constitute limiting factors in system performance. Those would almost certainly be found, rather, in instruction and data bus cycle times, and in PE operation and memory access times. Furthermore CUs are so far outnumbered by PEs that they may be considerably more powerful and complex than individual PEs without greatly impacting overall system cost; hence the options available for CU design are wider and less critical than they are for other parts of the system. Accordingly we will not consider CU design in any great detail, but will discuss it in general terms and will specify only a "first pass" design for the purposes of our simulation studies.

The essential functions of the CU are: 1) to interpret the program and to generate and broadcast the appropriate stream of instructions and operands to the PEs; 2) to perform certain conditional tests which apply directly at the control unit level - as, for instance, to test the number of PEs currently active, or to perform loop counts; 3) to co-ordinate its operations with those of other processes and to transmit and process messages; and 4) to perform various housekeeping chores connected with its other functions.

Of the above functions, the most critical is generation of the stream of micro-instructions and data to be broadcast to the PEs. Since operands are broadcast along a time-shared bus, it is desirable that the CU be able to broadcast operands, along with the instructions for the PEs to accept such operands, in a quasi-parallel manner with other PE instructions. If properly done, this allows the CU to take advantage of available slots on the data bus as they cccur, without forcing frequent short waits in the flow of instructions to the PEs. An example of such quasi-parallelism is illustrated in figure 8 for a hypothetical three-address instruction, "ADD A, B, C", where A, B, and C are assumed to be base addresses in PE memory for full word, integer operands.

Figure 8 illustrates a number of important features of both the control unit operation and the PE instruction set. The instructions whose boxes are marked with an "s" in the upper left-hand corner are those which require utilization of the data bus, and hence must be scheduled through the central controller for data bus accesses. The operation of this controller is discussed in the next section on data communication and I/O. Figure 8 is actually a precedence graph for the execution of individual micro-instructions. The instruction appearing in the left-hand column at time slot 4, for instance, may be executed any time after the instruction in the central column at slot 3, and before the one at slot 11. If the control unit can gain access to the data bus at any one of slots 4, 5, 8, or 9, the execution of this instruction does not impede the overall speed of execution for the micro-routine,

since it occurs during a forced idle period in the main line of execution. (Reads and writes to PE memory are assumed to require a period of time equal to three instruction bus cycles.) If none of these slots is available, the CU takes the earliest alternate slot that is, pre-empting any instruction that might otherwise occupy that slot, and shifting subsequent time slots accordingly. Notice that this instruction ($IDR_{0-15} \leftarrow B$) may correspond to either of instructions 1 or 2 in table 2, according to which half of the INPUT bus is used to broadcast the value of the address B. When the control unit requests a slot on a halfword bus, the central controller tells it which bus it is getting, and this in turn determines which of the two instructions the CU actually broadcasts.

This example also illustrates the nature of the PE adder circuitry discussed in section III. Note that after the RMR has received the contents of memory representing the B operand, no particular instruction is necessary to initiate the add. Rather a two cycle waiting period follows to allow the sum in the permanently active adder to stabilize. When this has occurred, the sum is simply gated into the accumulator in a single cycle. Of course this immediately begins to change the contents of the adder, but with master/slave logic on the AC register, no race condition can occur. In the meantime, the cycles during which the adder was stabilizing have been available for other instructions, such as that shown in the left-hand column at slot 19.

A further type of optimization which might be performed by the CU involves carry-over of information from one micro-routine to the next.

For instance if the instruction "ADD A, B, C" were followed by "ADD C, D, E", it would be highly desirable if the CU were able to recognize that the contents of address C were already contained in the AC following the first instruction, so that operations intended to fetch this operand could be bypassed in the second instruction. This type of optimization requires a highly sophisticated CU design, but could be attractive given the number of PEs potentially driven by a single CU. Fortunately one of the distinguishing features of AP programs, as a number of workers in the area have noted [5, 6, 8], is that they tend to be "straight line" with much less conditional branching than is found in ordinary programs. This makes look-ahead both easier and considerably more profitable than it would otherwise be, and allows the control unit to be built as a fairly long pipeline. It takes in machine encoded instructions at one end, interprets them with the aid of possibly several concurrent micro-programs, and spews out a stream of optimized PE instructions and operands at the other. An individual machine instruction may spend considerable time in the pipeline while it is interpreted, global operands are fetched, appropriate slots on the data bus are arranged, and optimization of PE instructions is performed. As long as a high throughput rate can be maintained, actual dwell time in the pipe is unimportant.

In order to perform loop counts and to access array data, the CU must have at least one testable index register. For the sake of simplicity, we have assumed one and only one such register. The first section of Table 3 lists the instructions associated with the maintainance of this register (load from memory, store to memory, initialize to a specified

value, increment by a specified value, and add or subtract the contents of a specified memory location). With these instructions, along with the conditional branch instructions in the second part of Table 3, a single register is sufficient for any program needs. Additional registers would eliminate the large number of loads and stores necessary when only one register is available, but we have not felt that this was sufficiently relevant to the AP aspects of the system to cause us to revise our initial design.

The second part of Table 3 lists CU branch instructions. Two of them are straightforward conditional branches based on tests of the index register. A third is a special branch used for subroutine calls, which loads the current value of the instruction counter into the index register just before making the transfer. Return is effected by using the unconditional jump to address zero, with indexing specified. This results in a jump to the address contained in the index register, which as a result of the earlier subroutine branch is simply the appropriate return address.

The three remaining conditional branches require some comment. They are conditional branches based on counts of the number of active PEs for a given CU. Their meanings are obvious, but their implementations involve significant architectural features of the MAP system. The CU initiates the operation by broadcasting instruction 58 of Table 2 to its PEs, causing those active to signal to the count and selection unit. The count and selection unit is a shared facility, so the CU must reserve its use for one cycle. It must also indicate to the unit

which operation it is that it wants it to perform. The count and selection unit has subsystems to perform a high speed analog count, returning answers of zero, one, or more than one; to perform a digital count, returning an exact answer to the originating CU; or to return a signal setting SW6 in one of the signaling PEs, while returning a signal to reset SW6 in all others, as discussed in the last part of section IV. It is the first of these subsystems that controls conditional branching for the last three instructions in the second part of Table 3.

The third group of instructions in Table 3 are those which relate to inter-process communication, and co-ordination of parallel tasks within the MAP environment. The system considered here is designed to implement the protection and communication strategy of G. J. Nutt and C. A. Ellis of the University of Colorado Department of Computer Science [3]. The central feature of this strategy is the use of an 8-bit ID register associated with each CU. (The number of bits corresponds to the number of CUs envisioned for the system.) The ID registers of the set of CUs are used to establish what are potentially very general communication hierarchies among the CUs, controlling the extent and nature of communications possible between individual members of the hierarchy.

Let: $S_i \in 2^{\{0, 1, 2, 3, 4, 5, 6, 7\}}$

be the set of ID register bits which are set for CU_i. If $S_j \subseteq S_i$, then CU_i can communicate with CU_j in a "privileged" manner. Specifically, it

may pre-empt the operation of CU_j, broadcasting instructions to the PEs allocated to CU_j using CU_j itself as a passive relay for its own instructions. In this mode, it may also transfer PEs allocated to CU_j to itself and vice versa. This is, in fact, the only way in which a PE may be switched from one CU to another. CU_i also has the ability during pre-emption to reset ID_j to any new $S_j \varepsilon 2^{S_i}$, or to reset the instruction counter of CU_j so that it resumes execution at a new location when it is released from pre-emption.

In the alternate case where $S_j \not\subset S_i$ but $S_j \cap S_i \neq \{\varepsilon\}$, then CU_i may communicate with CU_j, but only through the much weaker medium of a message buffer and a "message pending" flag which serves as an interrupt to the signaled CU if it is enabled to accept such message interrupts. Each CU has individual control over whether or not it will accept such interrupts and how it will process them if it does. Finally, if $S_j \cap S_i = \{\varepsilon\}$, the two control units cannot communicate with each other at all.

The precise details of message processing and inter-CU communication in the non-preemptive mode will not be developed here, since the subject depends heavily on details of the operating system design and is not germane to the AP aspects of the architecture which are of primary concern. In general, however, such communication requires that the two units involved share some mutually accessible central memory and some common usage conventions. It is also useful, though perhaps not absolutely necessary, if there is special hardware to pass at least one initial argument to the interrupted routine so that the burden of mutual

cognizance is not so severe. Consistent with the strategy of Nutt and Ellis referenced earlier, mutually accessible central memory is provided by allocating memory in blocks, with each block having associated with it an 8-bit "OWNER" register and a 4-bit "MODE" register. The OWNER register controls which CUs may access the block, and the MODE register controls how it may be accessed.

VI. Data Communication and I/O

As mentioned in section II, all I/O to and from PEs in a MAP system is buffered through central memory, with high speed, word parallel busses transferring data from central memory to individual PEs, and vice versa. The data bus is capable of operating on a much faster cycle than an individual PE memory, so that in order to drive the bus at capacity, data from a large set of PEs must be interleaved. The interleaving is accomplished through the mechanism of the ICTL and OCTL registers of figure 3, which stand for "input control" and "output control".

The ICTL and OCTL registers work in a similar manner. A register is initially loaded with a value representing a delay time. For each cycle to which the CU controlling the operation has access to the data bus, the register is decremented by one. When the register reaches zero, a gating signal is generated which, in the case of the ICTL register, causes a word to be gated from the INPUT bus to the IDR. For the OCTL register, the signal causes the contents of the ODR to be gated to the OUTPUT bus. Generally the initial contents of the ICTL or OCTL register will be different for each individual PE. The result is that each PE reads or writes one word from a stream of words on the data bus. It is, however, entirely possible for two or more PEs to have the same starting value. If a set of PEs share a common ICTL value, for instance, then each element of that set will accept the same word from the data stream on the INPUT bus.

Data streams need not always originate from or terminate to central memory. The CU may cause the contents of the OUTPUT bus to be fed back

to the INPUT bus, so that its PEs may act simultaneously as source and destination of the data stream. This allows exchange of data among PEs in arbitrary patterns, according to the initial values loaded into the ICTL and OCTL registers.

As an example, figures 9A and 9B illustrate how data would be exchanged among PEs in a 5 x 7 array when every PE requires input from each of its four nearest neighbors (as, for instance, in solving partial differential equations in two dimensions). The operation consists of either four or five data exchange operations in which designated subsets of the PEs act as broadcastors, creating the data stream, and others act as data receivers. Multiple exchange operations are necessary, since we are assuming that an individual PE can receive only one word of data in each operation. If the set of broadcasters and receivers are to be non-intersecting -- i.e., a particular PE may operate as a broadcastor or receiver, but not both on the same operation -- then five exchanges are required. The patterns of broadcastors and receivers would then be as shown in figure 9A. If PEs may act as both broadcasters and receivers within the same operation, then the complete exchange can be accomplished in four operations, using the patterns shown in figure 9B. The difference is not especially significant, since in either case the number of data bus cycles required is the same -- one cycle for each PE in the array. The second method saves a little setup overhead in having to initialize ICTL and OCTL registers one less time.

The communication bandwidth for data exchange in the above type of operation is less than it would be in a machine such as ILLIAC IV with hard parallel PE to PE data communication channels, but can be quite respectable all the same. In the above example, each word of data broadcast goes to four PEs simultaneously (ignoring boundary effects where a PE has fewer than four neighbors), so the communication bandwidth is four times the bandwidth of the data bus itself. If the data bus operates on an 80ns cycle as suggested earlier, this amounts to a bandwidth for data communication in the above example of 1.6×10^9bps. With higher multiplication factors arising when each PE broadcasts to a large subset of other PEs, the figure would be still higher; however the real advantage of the scheme is that it is completely general. It does not depend on hard paths to specific PEs, and extends readily to three dimensional arrays or irregular sets.

The above discussion ignores attenuation of the data communication bandwidth for a given CU due to competition for the data bus from other CUs. Although data streaming operations represent the only instances in which individual control units can utilize more than a fraction of the data bus bandwidth, they may be sufficiently common as to justify measures for reducing competition. This can be done by sectioning the data bus into a number of semi-independent "sectors". Each "sector" then has cross-bar connections to every CU, as illustrated in figure 10 for the INPUT bus. If two CUs have no common sector in which each owns active PEs, then they may have simultaneous access to the busses in their respective sectors. If there are enough sectors that each sector contains

PEs from only one CU, then each CU has full access to a bus on every cycle. In general, though, this cannot be counted on. One of the required capabilities in a MAP system is the ability of a CU to transfer a single PE or an arbitrary set of PEs to another CU. Except in the extreme case where each sector serves a single PE, this leads to the possibility of one sector serving PEs belonging to two or more different CUs. When this happens, concurrent access by those CUs is excluded. Sectoring therefore does not eliminate the need for bus scheduling, but significantly reduces the number of CUs which mutually contend for cycles. It also has implications for the design of MAP operating systems, since the operating system should allocate PEs in such a way as to minimize contention. This means trying to keep the number of CUs owning PEs in any one sector down to a minimum, and where sharing is necessary, trying to fulfill additional PE requirements from sectors where the PEs are owned only by the requesting CU or CUs with which the requesting CU already shares sectors.

As to the optimal size and number of sectors, this is a principle parameter for system tuning. The more sectors there are, the less bus conflict there will be under normal conditions. But each sector increases the cost of the system somewhat, due to the wide crossbar connections and to the enlargement of the conflict detection and resolution circuitry. Furthermore, the benefit of additional sectors falls off fairly rapidly beyond a certain point, as control units come to have essentially full access to the bus. At that point, conflicts which still exist are likely to be between control units that exchange processing elements as part of

an integrated multi-processing scheme. In the latter case it is almost inevitable that the two units will at times own active PEs in the same sector, and conflicts arising in this manner are relatively insensitive to the number of sectors. However it should be noted that a fairly high level of conflict can be tolerated before significant degredation shows up. For instance, if a CU requires access to a data bus one cycle out of five, and the level of conflict is such that it must wait an average of one cycle for every cycle it uses the bus, its performance is degraded by at most ten percent, and possibly much less if dynamic optimization of micro instructions, as discussed in section V, is employed.

References

1. Barnes, G. H., Brown, R. M., Kato, M., Kuck, D. J., Slotnick, D. L., and Stokes, R. A., "The ILLIAC IV Computer", IEEE Transactions on Computers, Vol. C-17, No. 8 (Aug. 1968), pp. 746-757.

2. Ellis, C. A., "Parallel Compiling Techniques", Proceedings ACM 1971 National Conference.

3. Ellis, C. A. and Nutt, G. J., "Protection in a Multi-Associative Processor System", unpublished.

4. Fuller, S. H. and Chen, R. C., "The I/O Port Architecture for Computer Modules", Carnegie-Mellon University Dept. of Computer Science.

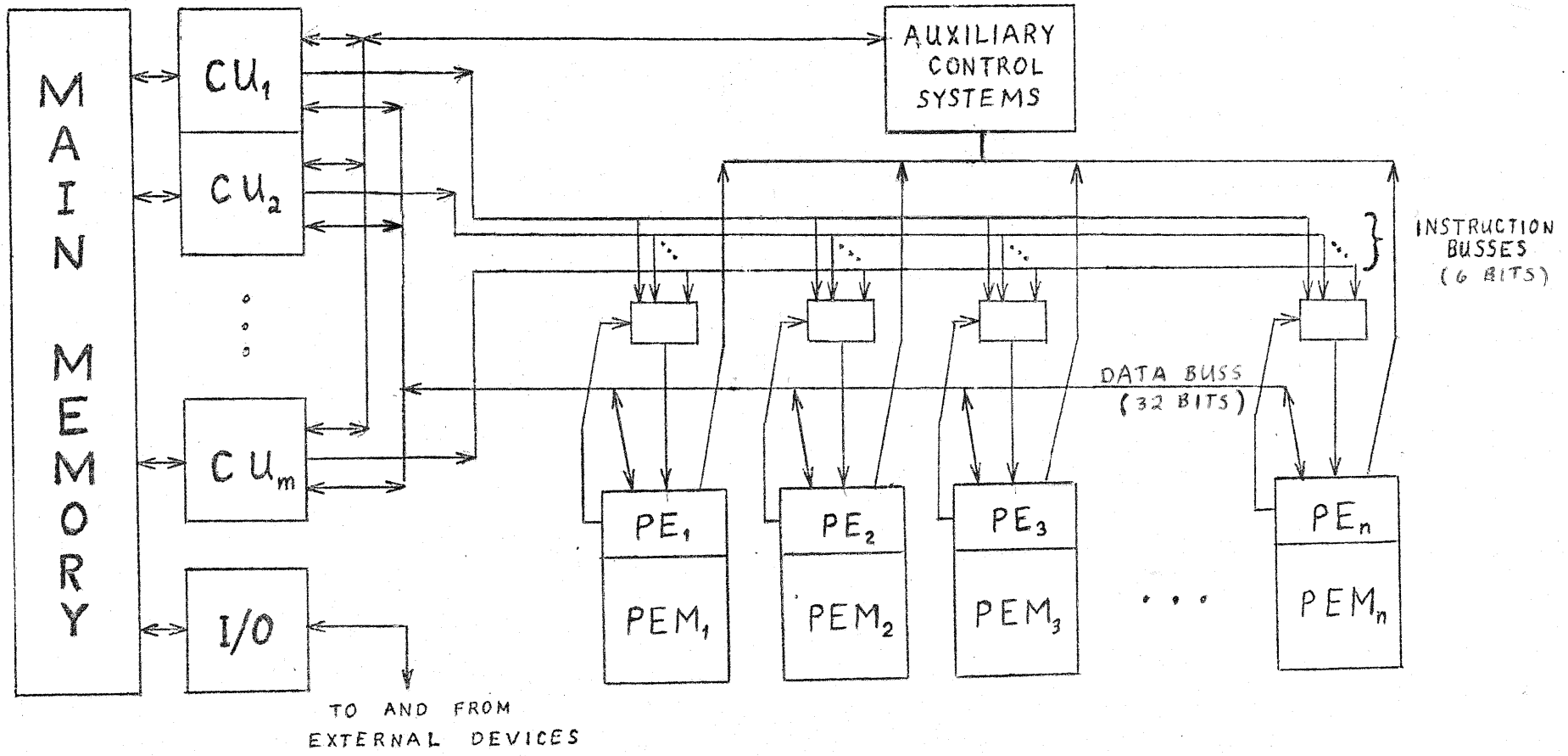
5. Githens, J. A., "A Fully Parallel Computer for Radar Data Processing", NAECON '70 Record, (1970), pp. 290-297.

6. Kuck, D. J., "Illiac IV Software and Application Programming", IEEE Transactions on Computers, Vol. C-17, No. 8, (Aug., 1968), pp. 758+.

7. Miranker, W. L., "A Survey of Parallelism in Numerical Analysis", SIAM Review, Vol. 13, No. 4, (Oct. 1971), pp. 524-547.

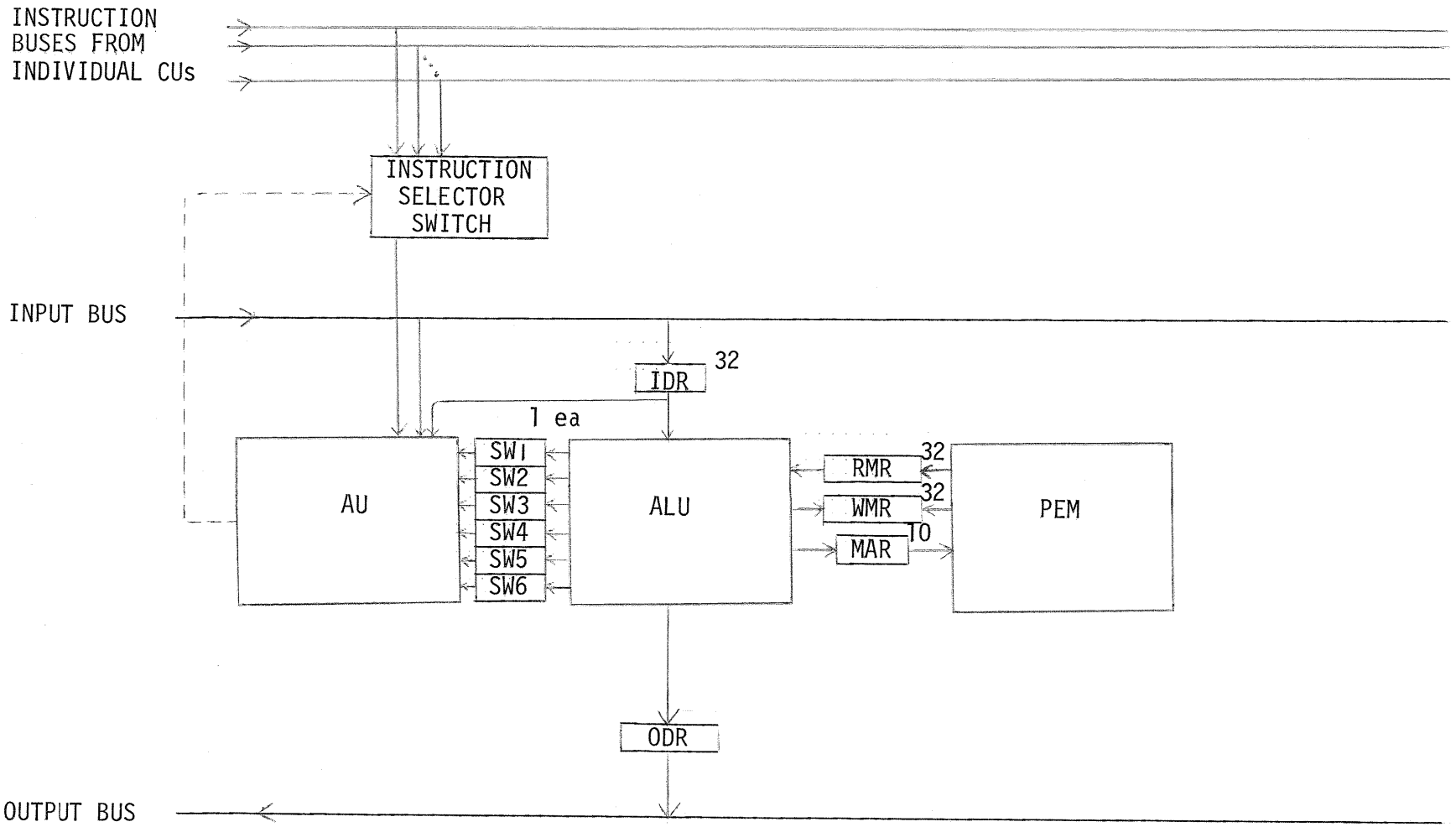
8. Rudolf, J. A., "A Production Implementation of an Associative Array Processor - STARAN", Proceedings of the FJCC, Vol. 41, Pt. I, (1972), pp. 229-241.

9. Slotnick, D. L., Borek, W. C., and McReynolds, R. C., "The SOLOMON Computer", Proceedings of the FJCC, Vol. 22 (1972), pp. 97-107.



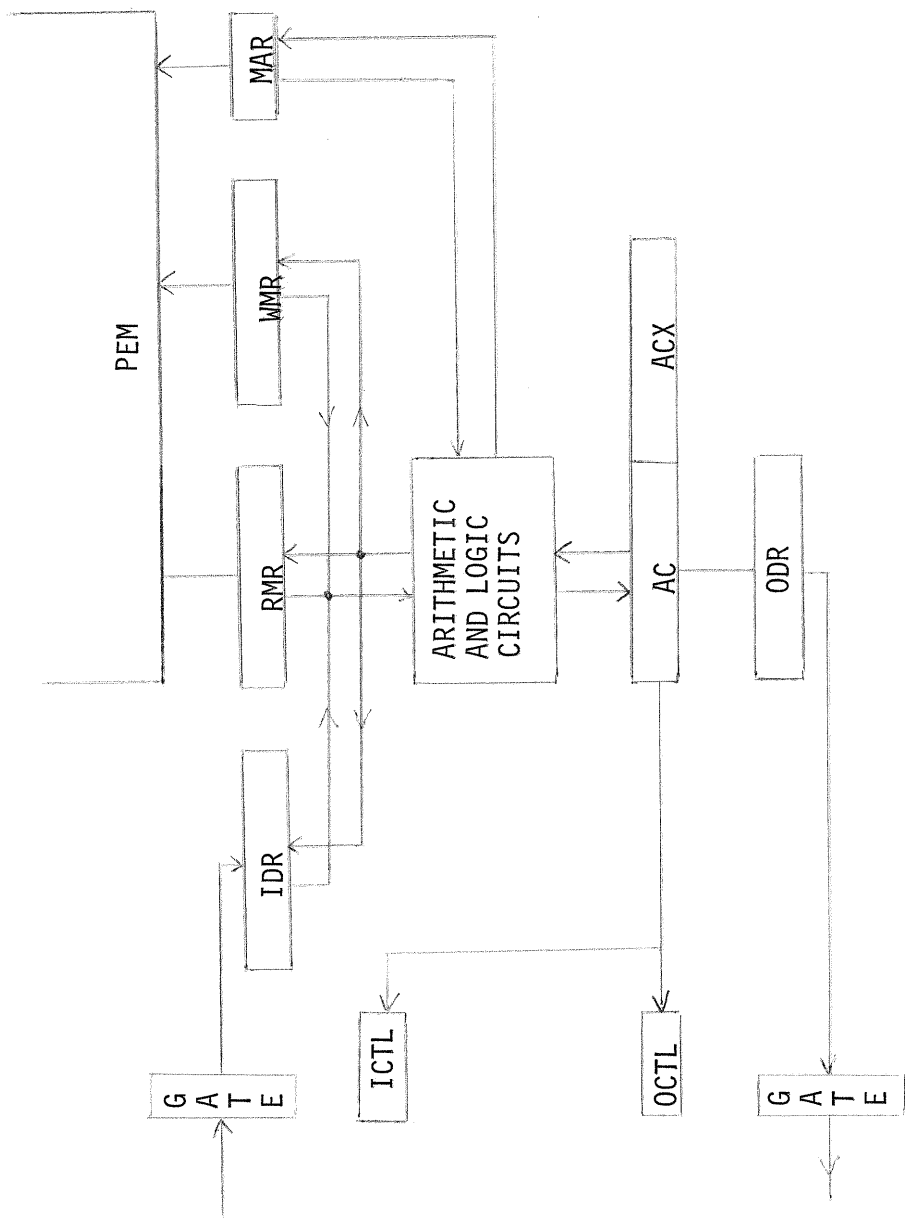
MULTI ASSOCIATIVE PROCESSOR SYSTEM

FIGURE 1



PROCESSING ELEMENT

FIGURE 2



PE ARITHMETIC AND LOGIC UNIT

FIGURE 3

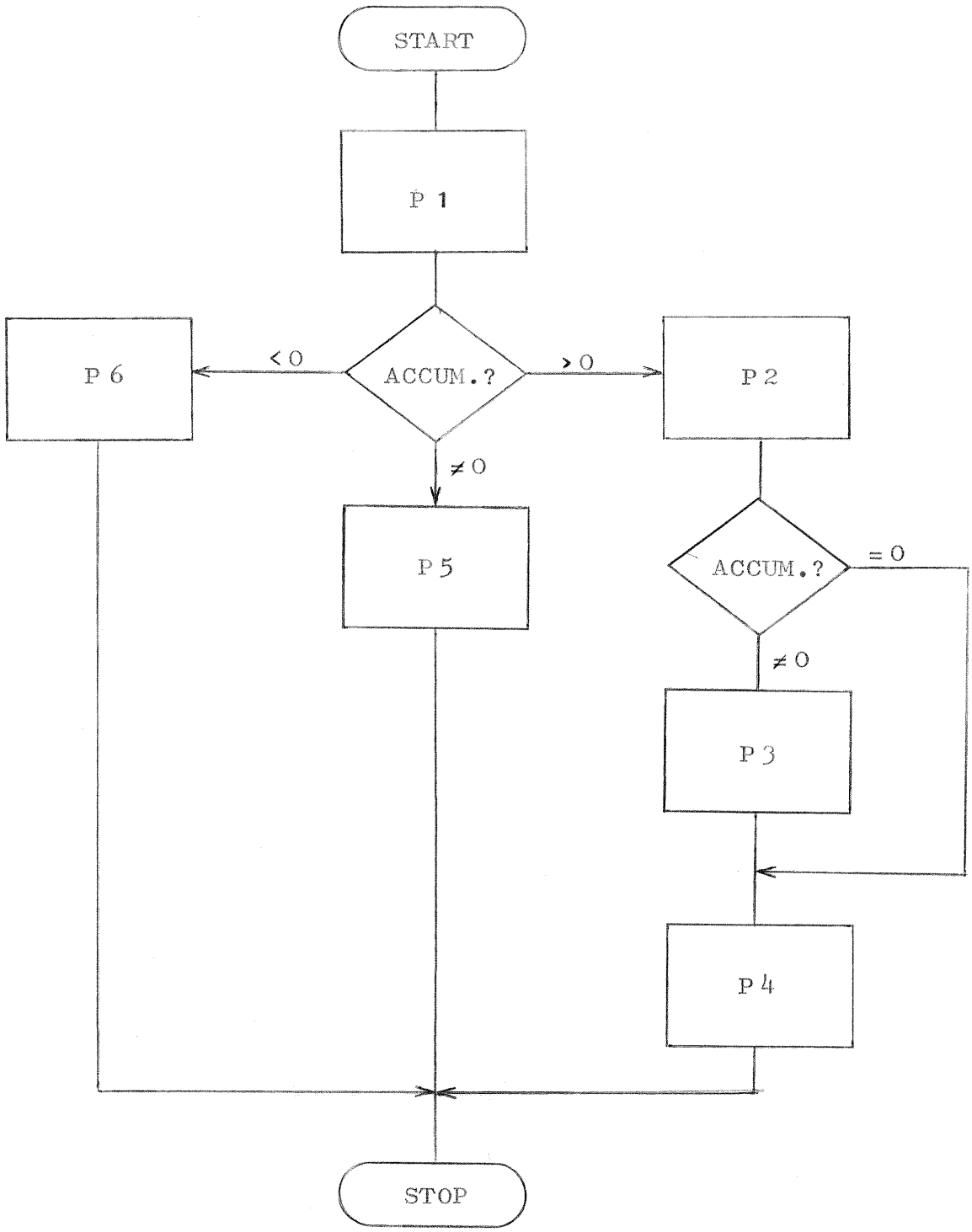
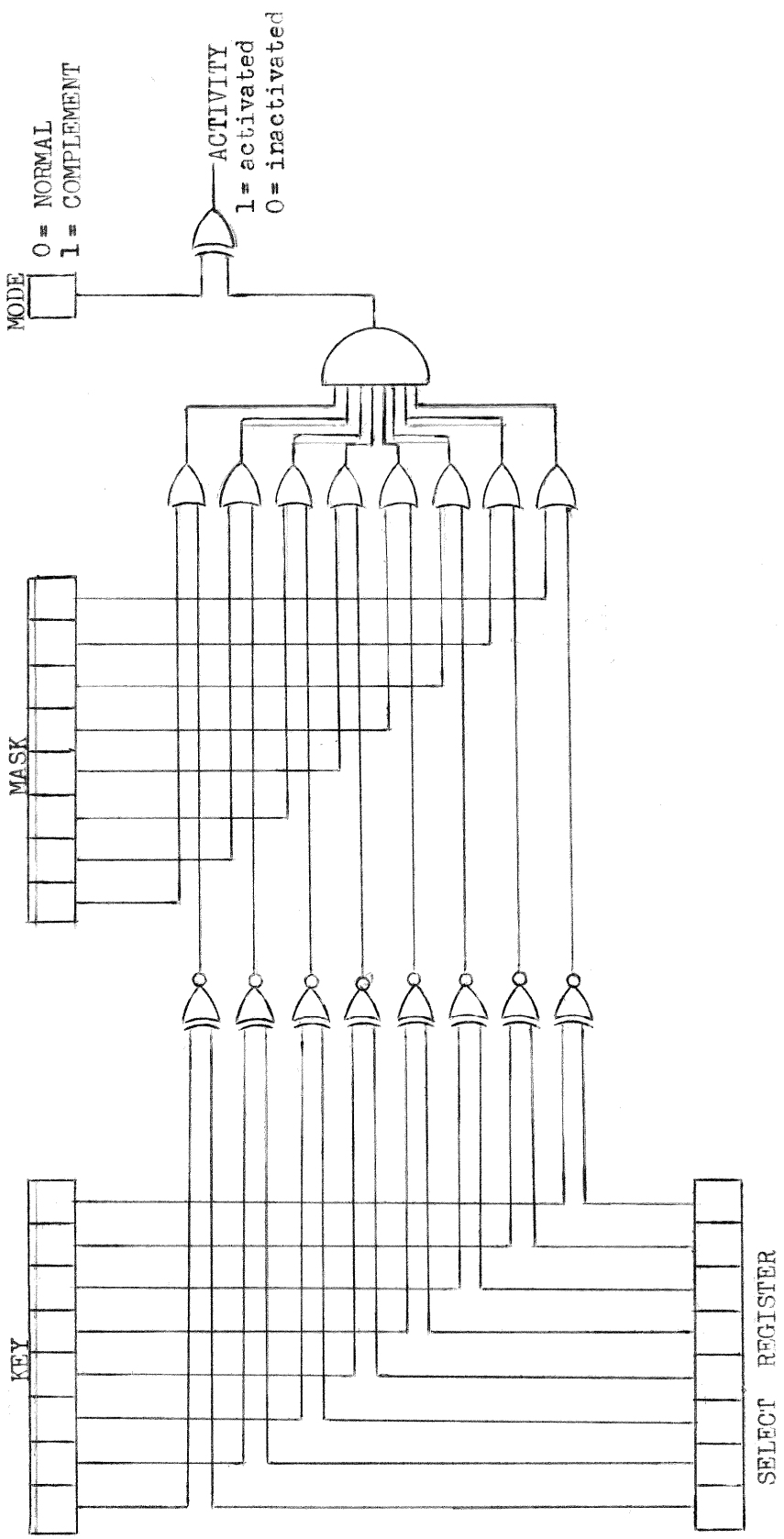
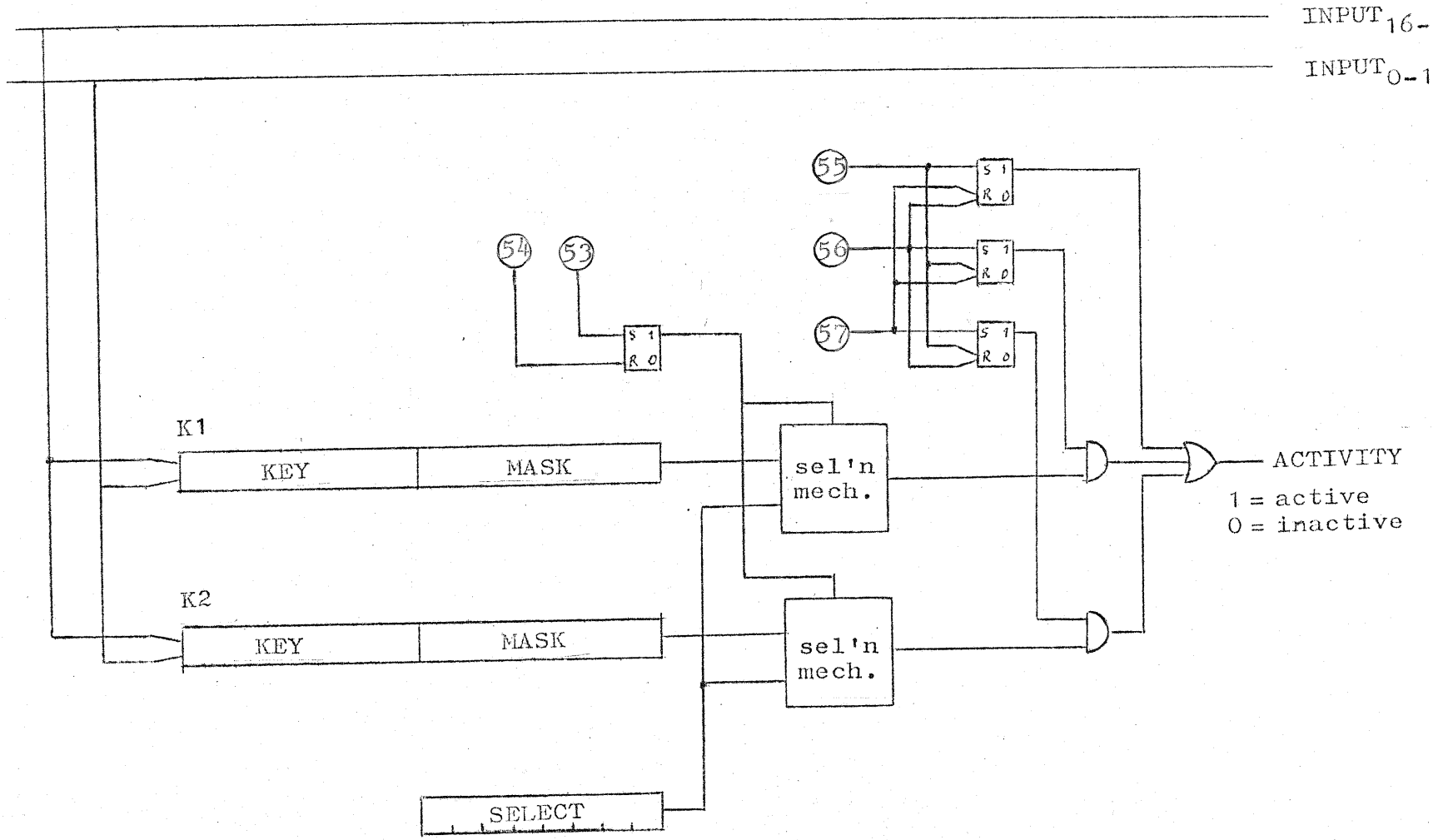


Figure 4



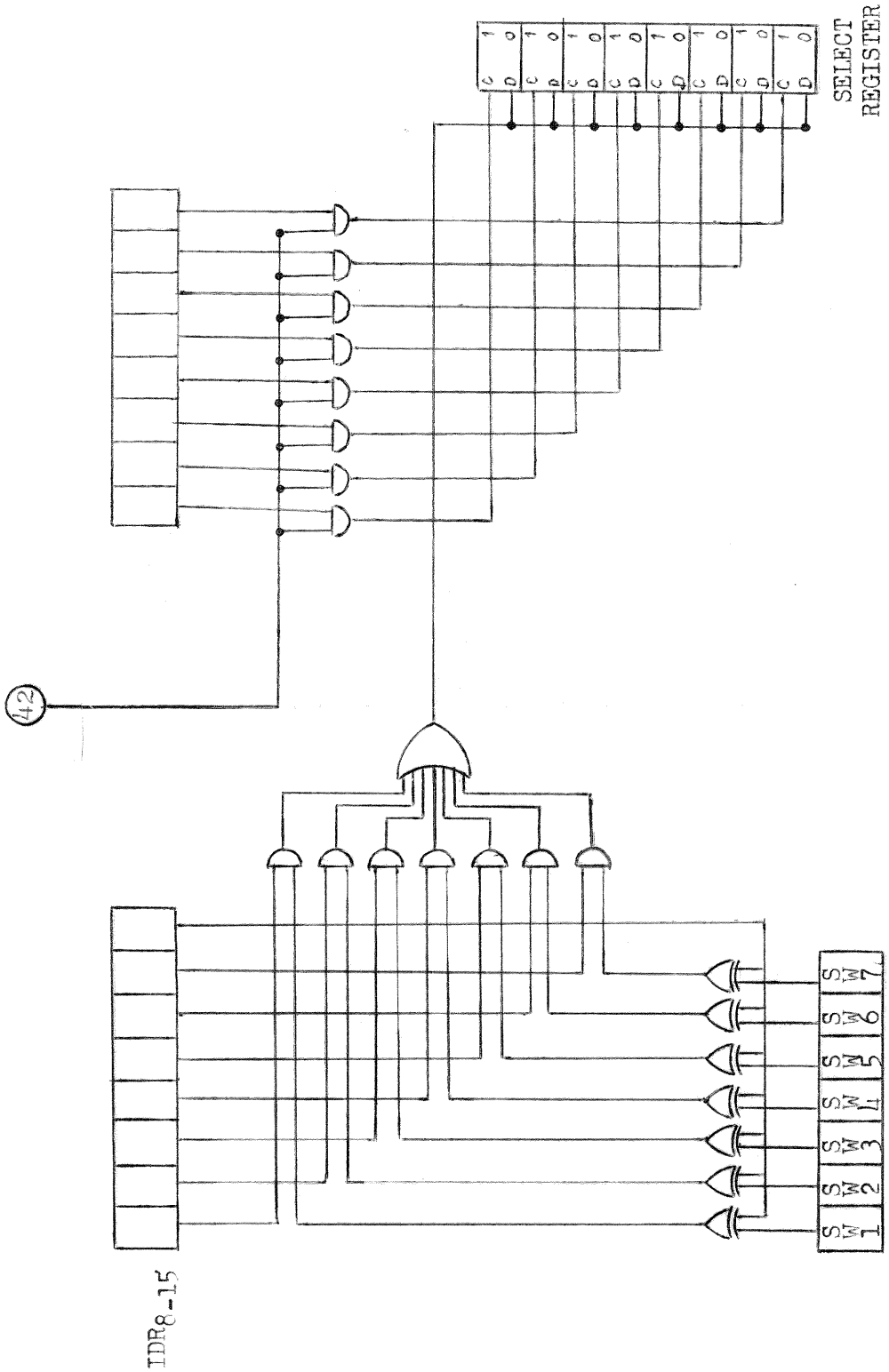
PE SELECTION MECHANISM

Figure 5



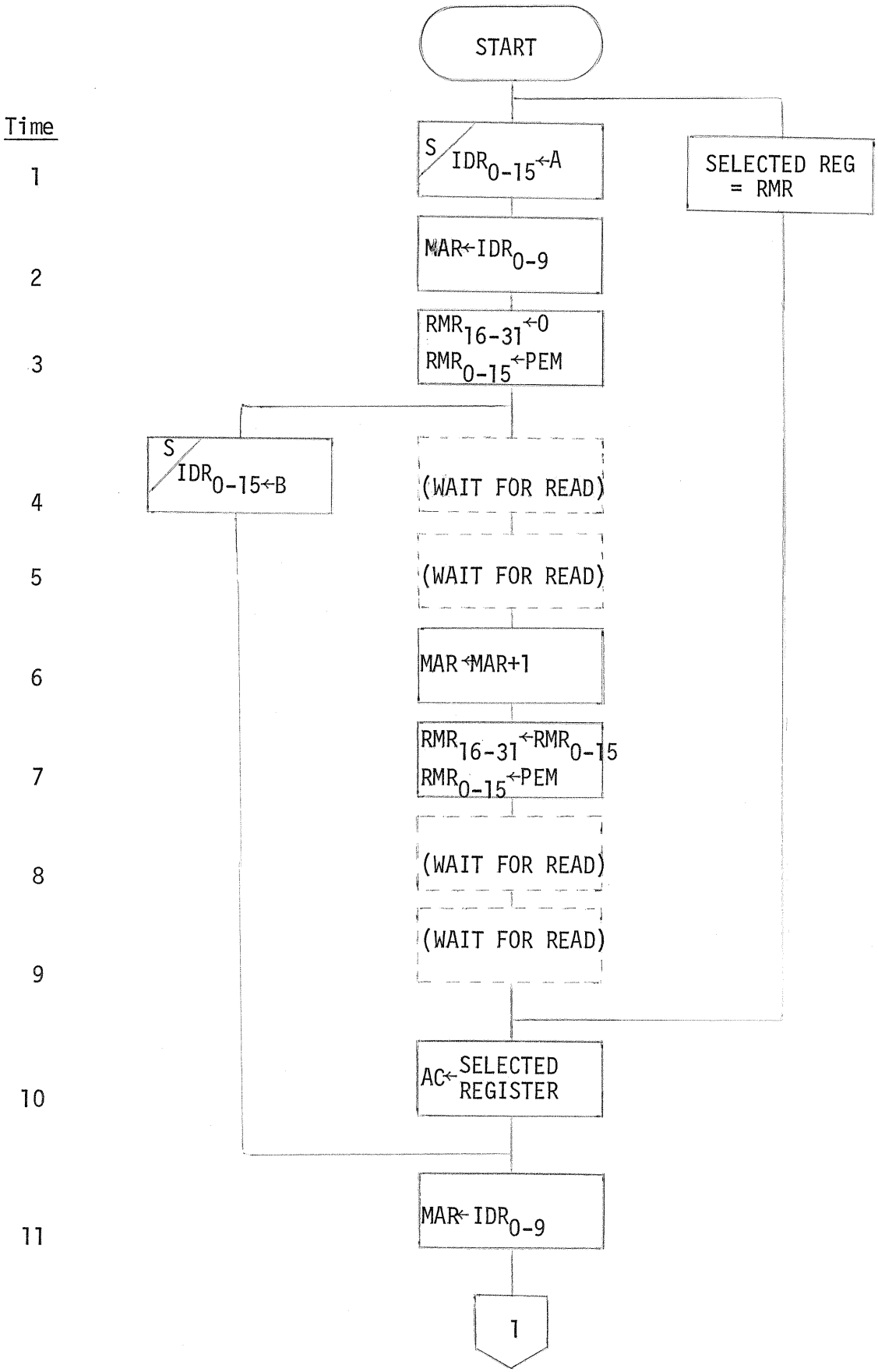
ASSOCIATIVE UNIT

Figure 6



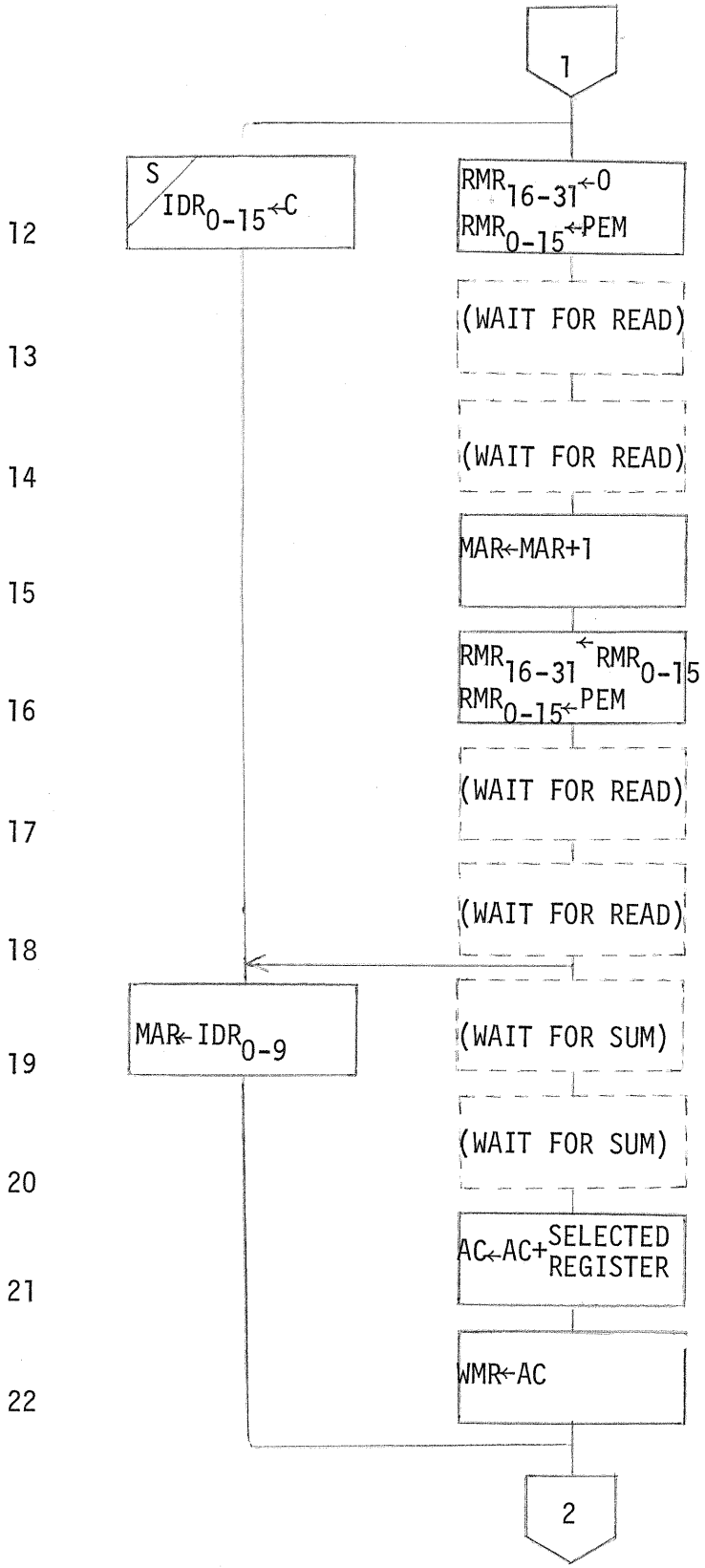
SWITCH RECORDING MECHANISM

Figure 7



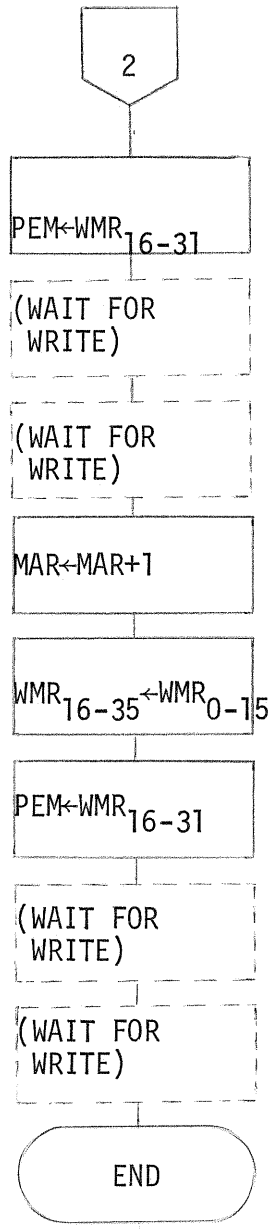
MICRO INSTRUCTIONS FOR "ADD A,B,C"

FIGURE 8



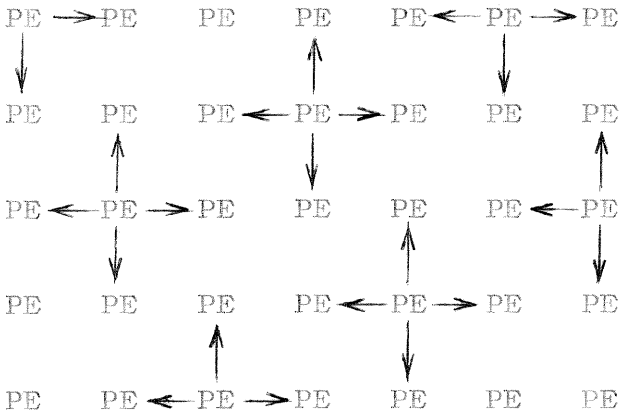
MICRO INSTRUCTIONS FOR "ADD A,B,C"

FIGURE 8 (cont.)

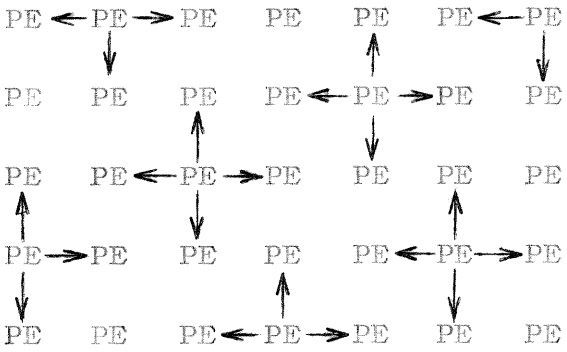


MICRO INSTRUCTIONS FOR "ADD A,B,C"

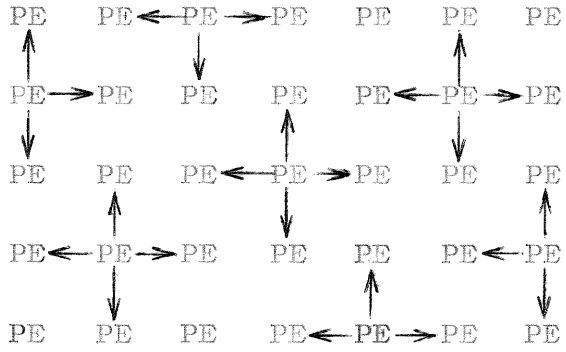
FIGURE 8 (cont.)



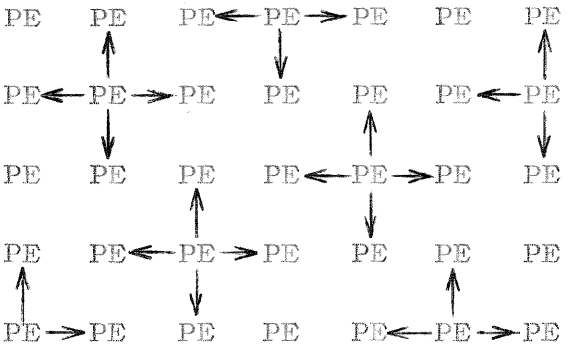
1st Exchange



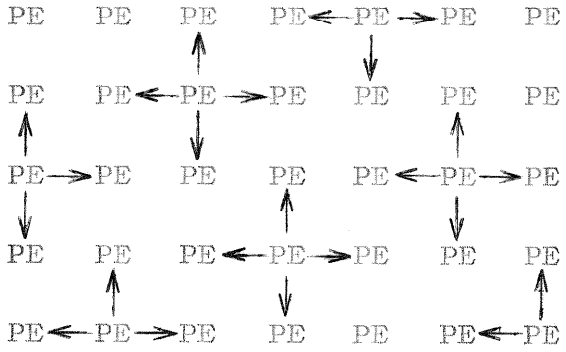
2nd Exchange



3rd Exchange



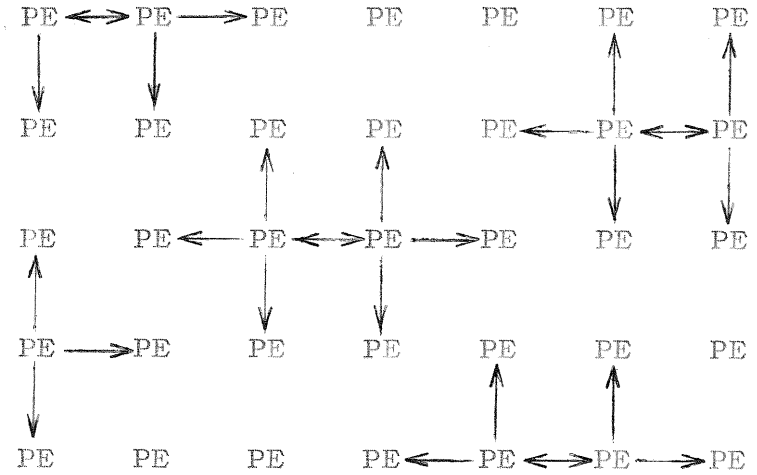
4th Exchange



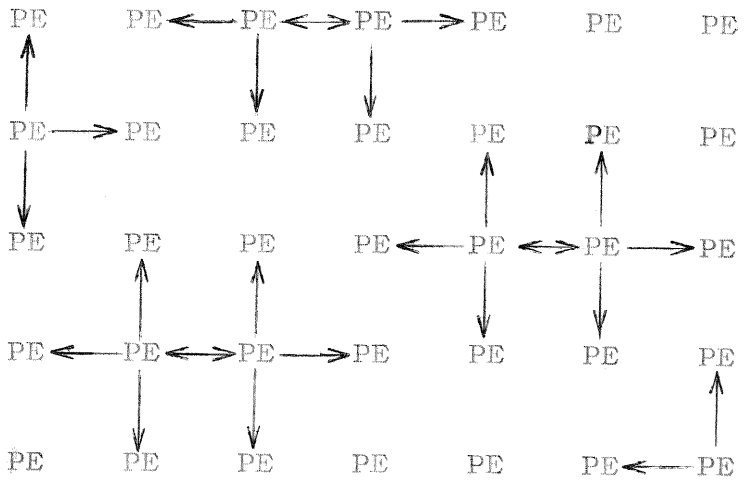
5th Exchange

FIVE CYCLE ARRAY DATA EXCHANGE

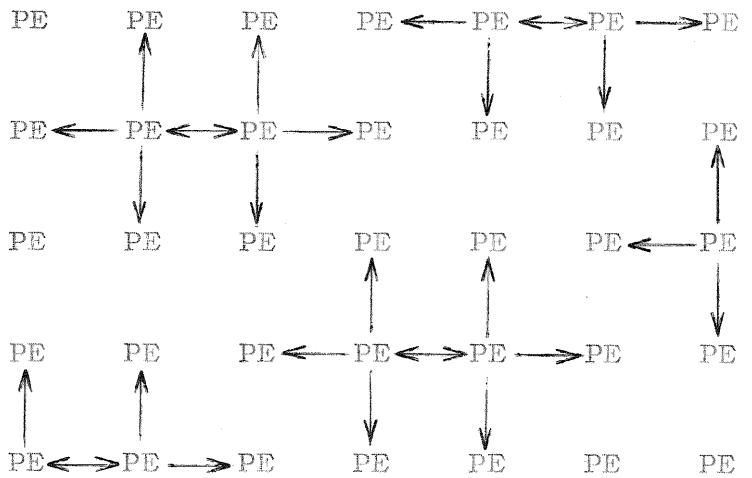
Figure 9A



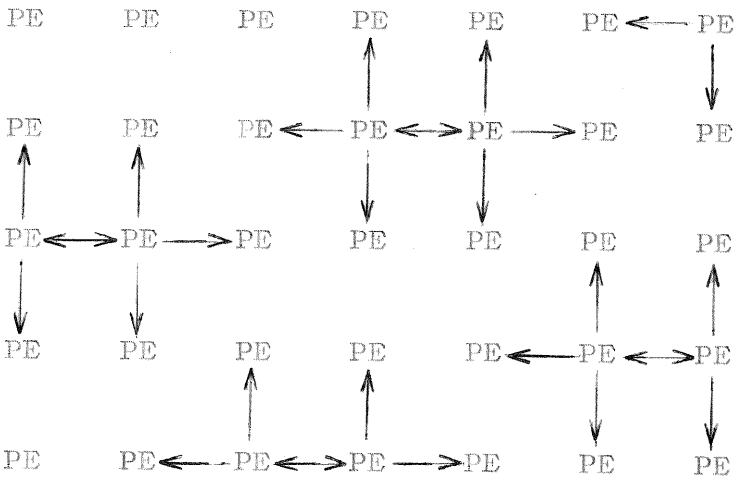
1st Exchange



2nd Exchange



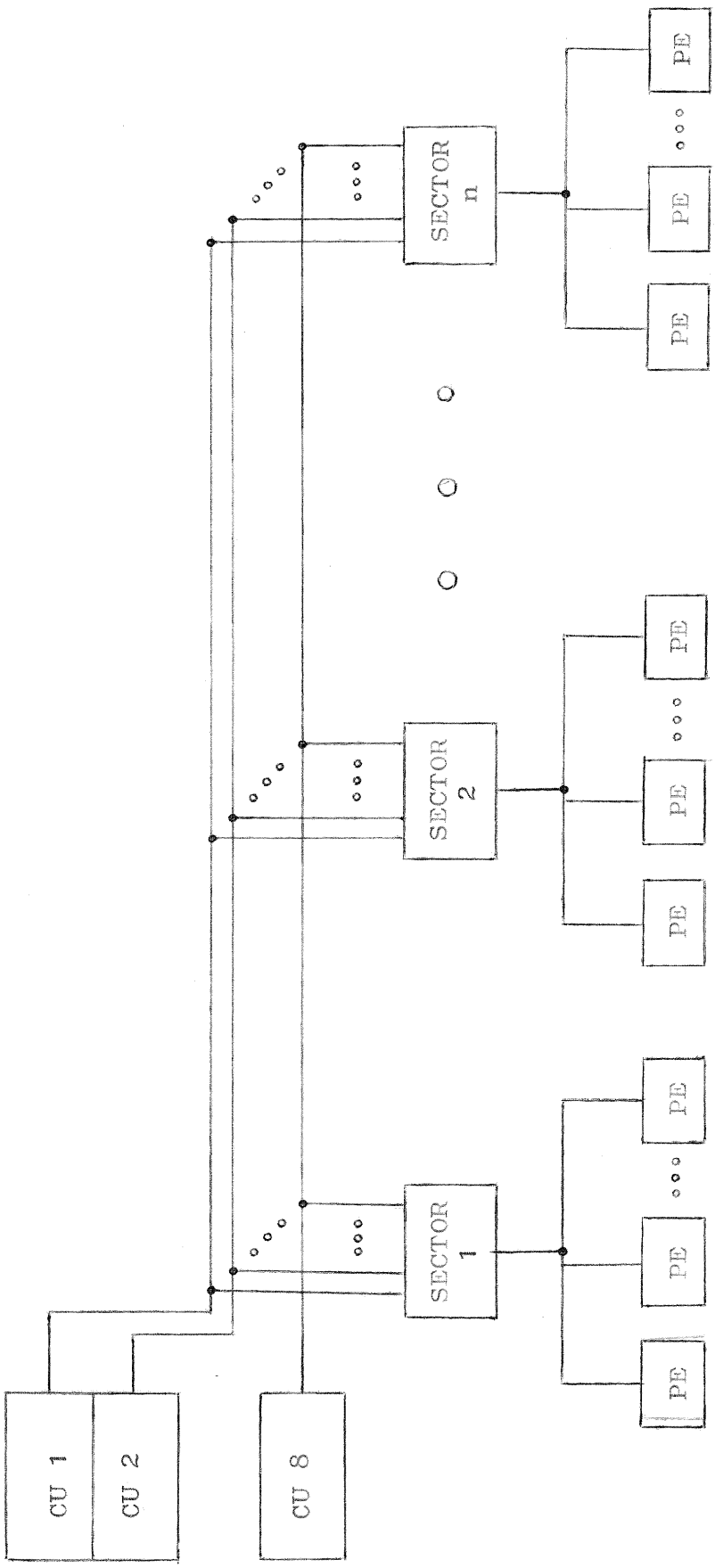
3rd Exchange



4th Exchange

FOUR CYCLE ARRAY DATA EXCHANGE

Figure 9B



DATA BUSS SECTOR ORGANIZATION

Figure 10

TABLE 1

ALU Gating Signals

1. $IDR_{0-15} \leftarrow INPUT_{0-15}$
2. $IDR_{0-15} \leftarrow INPUT_{16-31}$
3. $IDR_{0-31} \leftarrow INPUT_{0-31}$
4. MAR ← MAR + 1
5. $MAR \leftarrow IDR_{0-9}$
6. $MAR \leftarrow RMR_{0-9}$
7. $MAR \leftarrow AC_{0-9}$
8. AC ← selected register
9. AC ← .NOT. AC
10. AC ← AC + selected register
11. AC ← AC .OR. selected register
12. AC ← AC .AND. selected register
13. AC ← AC .EOR. selected register
14. $AC_{0-31} \leftarrow AC_{31} \cdot AC_{0-30}$ (left circular shift 1 bit)
15. $AC_{0-31} \leftarrow AC_{28-31} \cdot AC_{0-27}$ (left circular shift 4 bits)
16. $AC_{0-31} \leftarrow AC_{1-31} \cdot AC_{31}$ (arithmetic right shift 1 bit)
17. $AC_{0-31} \leftarrow AC_{4-31} \cdot 4(AC_{31})$ (a.r.s. 4 bits)
18. AC ← ACX
19. AC ⇋ selected register (simultaneous exchange)
20. WMR ← AC
21. $WMR_{16-31} \leftarrow WMR_{0-15}$
22. ODR ← AC
23. ICTL ← AC
24. OCTL ← AC
25. ACX ← AC

Read and Write

26. $RMR_{16-31} \leftarrow 0$; $RMR_{0-15} \leftarrow PEM(MAR)$
27. $RMR_{16-31} \leftarrow RMR_{0-15}$; $RMR_{0-15} \leftarrow PEM(MAR)$
28. $PEM(MAR) \leftarrow WMR_{16-31}$

TABLE 1 (Cont.)

ALU Extended Operations

29. align exponents of AC and selected register
30. normalize AC
31. convert AC to integer
32. AC.ACX ← product of AC and selected register
33. AC ← AC•ACX / selected register; ACX ← remainder

ALU Control Signals

34. designate IDR as selected register
35. designate RMR as selected register
36. designate WMR as selected register
37. designate zero as selected register
38. activate ICTL for countdown
39. activate OCTL for countdown
40. clock for countdown of ICTL and/or OCTL
41. write ODR to OUTPUT buss

TABLE 2

Associative Unit Instructions

42. Write value of designated switch into SELECT
43. Write OR of designated SELECT bits into SELECT
44. Write NOR of designated SELECT bits into SELECT
45. Write AND of designated SELECT bits into SELECT
46. Write NAND of designated SELECT bits into SELECT
47. SELECT ← AC
48. AC ← SELECT
49. K1 ← $INPUT_{0-15}$
50. K1 ← $INPUT_{16-31}$
51. K2 ← $INPUT_{0-15}$
52. K2 ← $INPUT_{16-31}$
53. set MODE switch for NORMAL
54. set MODE switch for COMPLEMENT
55. designate K0 as control register
56. designate K1 as control register
57. designate K2 as control register
58. signal to count and selection unit
59. set OWNER switch to designated CU

Associative Compare Instructions

60. initialize for associative compare:
    - transfer contents of AC to ODR, inverting sign bit
    - reset associative compare cursor
    - set SW5
61. compare AC to IDR using associative compare cursor
    - resets SW5 if comparison shows AC not maximum
62. broadcast local cursor location over OUTPUT buss
63. compare local cursor position to bits in IDR
    - resets SW5 if there are any cursors farther right.

TABLE 3

Control Unit Instructions

Mneumonic	Operand(s)	Indexing?	Description
Index Register Instructions			
LDX	CM address	yes	Load index register from CM
STX	CM address	yes	Store contents of IR to CM
SETX	immediate	N/A	Load IR with specified value
INCR	immediate	N/A	Increment IR by specified value
ADDX	CM address	yes	Add contents of CM address to IR
SUBX	CM address	yes	Subtract contents of CM address from index register.
CU Branch Instructions			
BXP	CM address	no	Branch to specified address if contents of IR > 0
BXNP	CM address	no	Branch to specified address if contents of IR ≤ 0
SUBR	CM address	no	Load IR with current IC value + 1 and go to specified address
JUMP	CM address	yes	Go to specified address
BCT0	CM address	no	Branch if count of active PEs = 0
BCT1	CM address	no	Branch if count of active PEs = 1
BCTG1	CM address	no	Branch if count of active PEs > 1
Process Communication and Control Instructions			
PREMPT	8-bit ID	N/A	Pre-empt operation of CU with specified ID, if allowed
DROP	CM address	no	Drop any pre-emption currently in effect, setting IC of pre-empted CU to specified address. (if addr = 0, IC unchanged)
TAKE	none	N/A	Transfer active PEs of pre-empted CU to pre-empting CU.
GIVE	none	N/A	Transfer active PEs of pre-empting CU to pre-empted CU

TABLE 3 (Cont.)

Mneumonic	Operand(s)	Indexing?	Description
SIGNAL	8-bit ID and CM address	no	Generate message interrupt to CU having specified ID and pass specified address as argument
DISABLE	none	N/A	Disable message interrupts
ENABLE	CM address	no	Enable message interrupts, setting specified address as address of interrupt processing routine.
STOP	none	N/A	Halts all activity within a CU until CU is pre-empted.
Miscellaneous CU Instructions			
COUNT	CM address	no	Count active PEs and store count at specified address.