NOTES ON
A MAP MICROPROCESSOR IMPLEMENTATION*

by

Gary J. Nutt
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

#CU-CS-102-77

January, 1977

* The work described here was funded by the National Science Foundation under grant number MCS74-08328 A01.

TABLE OF CONTENTS

I. Overview of the Architecture

This is a working paper intended to describe certain facets of implementing a parallel processor as a collection of conventional bit slice microprocessor components. The ideas expressed in the paper are subject to revision as the implementation is refined. Comments and criticisms on the work are invited by the author.

The Multi Associative Processor (MAP) computer system is an array processor capable of simultaneously executing up to eight programs, where each program has a single-instruction-stream multiple-data-stream organization. The architecture includes eight control units (CUs) to execute the instruction streams and 1024 processing elements, (PEs) that may be dynamically allocated to the CUs. Each PE has a private PE memory (PEM) to contain a data stream. The machine makes no provision for the inclusion of a host processor to handle compilation and operating system tasks; instead, the language processors, utilities, operating system programs, etc., are expected to execute on one or more of the identical CUs in conjunction with one or more PEs, as required by the given program. The PEs are identical and are allocated to the CUs from a common pool, the number allocated depending on the requirements of the process currently executing on the CU.

MAP has been called an associative processor (AP) because of the nature of the mechanism used by a single CU in activating and deactivating the subset of PEs currently allocated to it. This content addressability feature and PE organization differ substantially from other associative processors such as the Goodyear STARAN computer.

In the remainder of this introductory section, the basic components of MAP will be introduced with more detailed discussions following in later sections. To provide a global view of the machine's operation, the remainder of this section includes a trace of a typical instruction execution.

The discussion of the MAP architecture given in this report differs from that given in Technical Report CU-CS-070-75 in that the design now takes bit slice microprocessor technology as a possible implementation mechanism.

A. MAP Components

Figure 1 shows a conceptual block diagram of the MAP computer system. The I/O subsystem consists of all mass storage, peripheral equipment and data channels to establish a data path between the main memory of the machine and the external environment. Although no detailed design of this subsystem currently exists, it is assumed that this component is similar to a conventional I/O system on a conventional architecture.

The Main Memory system, (MM), is used for three purposes:

- An I/O buffering mechanism,
- Instruction storage for all processes that execute on the eight CUs,
- Storage for global data used by a single CU and multiple PEs.

Details of the main memory organization are provided in section VI.

The eight control units each fetch and decode a single instruction stream on multiple PEs. The conceptual view of a CU relies on the unit to be able to manipulate index registers, form main memory addresses using indirection and pre-indexing, maintain a program counter, communicate with (processes on) other CUs, and broadcast data and instructions to the set of PEs currently allocated to the given CU. The Control Unit Processor Interface, (CUPI) is a special unit to execute instructions that apply to more than one control unit, e.g. control unit synchronization operations. CUPI operation is discussed at more length in section V.

The distribution switch is used to handle instruction and data routing from CUs to PEs, to route data from PEs to CUs, and to route data from PE to PE. The distribution switch employs bus sharing in a crossbar switch, allowing any CU-PE combination for routing. This switch is discussed in Section IV.

The processing elements perform arithmetic and logical operations on data stored in their own PEM or on a datum that has been broadcast from the owning CU. Some features worth noting include:

- There are no direct data paths between individual PEs. All data communication is performed via common buses running to all PEs.
- There are no physical characteristics of individual PEs which distinguish one from another.

- 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.

B. A Typical Instruction Execution Trace

For the following discussion, assume that a given CU has previously been allocated a subset of n_i PEs, and that all of the PEs are active, i.e., any command broadcast by the control unit, CU_i, is to be executed by all of the allocated PEs. The following hypothetical instructions are to be executed:

LØAD	3,*20,2	Load each PE's register 3 from the effective PEM address computed using single level indirect addressing (*), pre-indexed by the contents of each PE's register 2 with a base address of 20. Note that the index register offset may vary from PE to PE.
GLOBAL LOAD	3,*20,2	Load each PE's register 3 from the effective main memory address using single level indirect addressing, pre-indexed using the CU's register 2, with a base address of 20. Each active PE receives the same main memory location content.
SELECT POSITIVE	3	Deactivate all PEs whose register 3 contain a negative value.
SIGNAL		Pass a signal from the executing CU to a CU pointed to by a register in the executing CU.

Each instruction is first fetched from the main memory location specified by the program counter of the control unit. The CU inspects an 8-bit operation code and chooses an execution unit based on the 3 most significant bits of the operation code. An instruction may be executed by the control unit processor interface (e.g., the SIGNAL instruction); by the control unit itself (e.g. a branch instruction); a

set of processing elements (e.g. the LOAD instruction); or a combination of the control unit and the processing element subset, (e.g. the GLOBAL LOAD instruction).

The decode unit recognizes the LOAD instruction as being executable by the PEs; therefore, it broadcasts the instruction word to the set of active PEs (over the distribution switch). The CU idles for the amount of time required for the instruction execution, while the PEs compute their respective effective addresses and load their respective registers. The CU is then ready to fetch the next instruction word.

The GLOBAL LOAD instruction causes the CU to decode the instruction, to recognize that it is partially executed by the CU and partially by the PE subset, and to broadcast the instruction to the PE subset. The PE decodes the instruction and then waits for data to be broadcast by the CU. Meanwhile, the CU computes the effective address, fetches an operand from main memory, and broadcasts the operand to the PE subset. The PEs receive the data and load their respective registers.

SELECT POSITIVE is executed as an associative instruction within each PE. All active PEs test their respective registers for a positive result; if the result is negative, the PE deactivates itself. (Another selection instruction can be used to activate previously inactive PEs). SELECT-type instructions allow the programmer to process data streams on the basis of previous computations within that data stream.

SIGNAL is executed by the control unit processor interface, CUPI. The CUPI determines if the CU attempting to execute the SIGNAL is entitled to do so; if privilege is present the CUPI determines the receiving CU and "attaches" the signal to it, (more details of this operation appear in Section V).

C. Observations on the Overall Architecture

The MAP architecture provides a mechanism for exploiting parallelism at two different levels. The first level is the SIMD parallelism, allowing multiple data streams to be simultaneously handled by one instruction stream. The second level of parallelism is between control units, i.e. parallel instruction streams can exist in the system. The advantages of each level of parallelism have been argued at length by SIMD proponents and by MIMD proponents and are not repeated here. However, MAP provides the generality of both methods. Combining the two methods

allows an operating system to make efficient use of the PE resource, as well as Main Memory. Whenever a process executing on a particular CU is interrupted, the entire PE subset is not dormant; instead, only those PEs currently allocated to the (process on the) CU are dormant.* The remaining PEs can be used by other CUs. Thus multiple control units tend to increase overall PE utilization.

Multiple control units also allow higher utilization of PEs in terms of the number of allocated PEs. SIMD tasks frequently have "natural" solutions requiring specific numbers of PEs; e.g. in air traffic control, the number of PEs required corresponds 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 at a cost in program clarity and efficiency. With multiple CUs, the number of PEs in the system may be large enough to handle more demanding problems according to their "natural" solutions. At the same time, the operating system can take advantage of a batch job mix and multiprocessing to maintain overall utilization while running smaller problems. These advantages are typical of advantages motivating any shared resource system.

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

---

* Multiprogrammed CUs may not leave PEs idle.

parallel I/O becomes unattractive. Because of the uncertainty of which 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 any PE. This problem does not arise in a single CU parallel processor, since the entire pool of PEs is available for each job.

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 and 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 10^5 bps (typical for direct channels to mass storage devices). However, MAP requires that the data be loaded into central memory before it can be transmitted to the PEs.

II. Control Unit Organization

The purpose of a control unit is to provide a mechanism to implement the software notion of a process, where the process itself is composed of parallel tasks, i.e., each control unit must manage control and data flow for a particular SIMD program. To implement this function, each CU must:

- Fetch instructions stored in MM,
- Form MM global operand addresses,
- Determine the flow of control of the program,
- At least partially decode instructions to be executed by CUPI and the PE subset,
- Broadcast instructions, addresses and/or operands to PEs,
- Coordinate its operation with other control units via CUPI.

Figure 2 is a conceptual diagram of a control unit as it might be implemented with bit slice microprocessor components. The interface to CUPI and MM consists of the two registers CUMAR and CUMDR. CUMAR is a 22 bit MM address register, and CUMDR is a 32 bit data/instruction register. CUPI is treated as a memory location in MM space by the con-

trol unit, i.e. the CU communicates with CUPI by storing information into a MM address corresponding to CUPI. CUMDR could be replaced by a more elaborate buffer to reduce memory conflicts among the CUs, and to logically match the speeds of MM access and the CU cycle time. For the current version of MAP, buffering is not used since memory technology provides memories that have lower access time than current bit slice microprocessor cycle times.

The microprogrammed decode unit is implemented as a bit slice microprocessor control unit such as the Intel 3001 or a set of AMD 2909 chips. The unit also includes a control store of appropriate width and a pipeline register to allow the decode unit to overlap control store fetch and microinstruction execution. The 3 most significant bits of the MAP op code (stored in CUMDR during the instruction fetch phase) determine the instruction type. Type 0 instructions are those that affect the operation of another CU, and are passed, via CUMDR, to CUPI. Type 1-3 instructions are to be executed (primarily) by the CU itself, and are gated into the decode unit for processing. Type 4-7 instructions are executed by the PE subset. One feature of the microprocessor implementation is that the MAP instruction set is implemented in microprograms, and is relatively easy to modify and expand; a sample instruction set is included in the Appendix, and further description of the architecture will use that set as an example to further explain system operation.

After the decode unit recognizes the instruction type, it will route the instruction as appropriate. The decode unit must be responsible for the timing of all instructions, and either of two methods can be used to accomplish synchronization among the CU, CUPI, and PE subset. Type 0 instructions are written to a predetermined MM location, which will invoke CUPI to execute the instruction. The decode unit then enters a microprogram loop to await an interrupt from the CUPI to indicate that it has finished processing the instruction. Since the PE subset allocated to a CU is variable with respect to physical PE identities, the interrupt method for indicating instruction completion is untenable. Instead, the decode unit employs a synchronous mechanism where the (maximum) PE instruction execution time is determined by the decode unit when it allocates work to the PE subset. This requires that indirect addressing within PEs be

limited to a single level. Global operations, such as the Global Load discussed earlier, tend to complicate this method but do not obviate it; the decode unit recognizes a global instruction and immediately broadcasts it to the allocated PE subset. The decode unit then performs the CU portion of the instruction, synchronizes with the PE subset by a bus signal, and performs the information exchange.

The Control Unit Instruction processor is implemented as a 32 bit microprocessor chip set such as the Intel 3002 or the AMD 2901 chips. Each of these ALU sets incorporate internal registers, and hardware to perform elementary arithmetic and logical operations. The internal registers are used to implement MAP programmable registers required to perform loop counts, to index MM, and to allow operand testing that influences the flow of program control. These registers are designated as CAC[0]-CAC[3] in Figure 2. Additionally, the program counter, PC, is implemented as an internal microprocessor register. A typical set of instructions to be executed, at least in part, by the CU Instruction Processor is shown as Type 1-3 instructions in the Appendix. They include CAC load and store operations, CAC addition and subtraction, and a variety of branch instructions based either on CAC contents or other conditions that might exist in the machine.

The remaining three registers shown in Figure 2 are external to the microprocessor ALU chips. The broadcast register, DBR, and the ID register are the interface between the CU and the Distribution Switch. DBR is a 32 bit register used to broadcast instruction words and operands to allocated PEs, and to receive operands from the PEs. ID is an 8-bit register containing an identifier unique to (the process executing on) the CU; it is used by the Distribution Switch to route DBR contents to/from PEs allocated to the CU. SIG is an 8-bit register used to point to another (process on another) CU during inter CU communication. Although ID and SIG are conceptually a part of the CU, both are implemented in CUPI.

The ALU microprocessor chips that implement the CU instruction processor require 5 internal registers; the remaining registers, (6 in the case of Intel 3001, 11 in the case of AMD 2901), are used by the microprograms.

The entire operation of the CU depends heavily on the Distribution Switch to rapidly route data and instructions to the PE subset. Therefore, the bus system used to route information is a multiplexed crossbar switch, although the conflict resolution hardware at the cross connections is eliminated by establishing the information flow path at PE allocation time, i.e., once a PE is allocated to a CU, it cannot be shared with another CU. These conventions make the crossbar switch for instruction broadcasting plausible.

III. Processing Element Organization

The purpose of each processing element is to carry out single data stream operations as they are required by the control unit. The operation of a PE is complicated by the need to selectively activate and deactivate the unit for certain instruction sequences as specified by the program, i.e. the program may wish to deactivate a PE based on conditions that exist locally in the PE. The list of functions to be implemented in each PE include:

- Receipt of the PE instruction,
- Execution of arithmetic and logical instructions,
- PE memory address formation and access,
- Receiving and sending data from/to a CU,
- Determining the activity state of the PE.

Figure 3 is a conceptual diagram of a processing element as it might be implemented using bit slice microprocessors. The overall operation of a PE is simpler than that of a CU, since the PE need not include logic to handle instruction fetching. On the other hand, the PE must provide an interface to the Distribution Switch as indicated in the upper portion of Figure 3. The 8-bit OWNER register is set when the PE is allocated to a CU so that its content matches the content of the ID register of the corresponding control unit. Whenever information is placed on the data bus of the Distribution Switch and the ID bus matches the OWNER content, the data is gated onto the PE data bus. Data can also be gated back onto the Distribution Switch if the ID and OWNER tags match.

If the PE is waiting for an instruction from the CU, the PE data bus content will be routed to the microprogrammed decode unit. This unit is similar to the CU decode unit discussed earlier, and is implemented as a bit slice microprocessor control unit. The decode unit need only handle 7 bit op codes, since only Type 4-7 instructions will be received by the PE. Most of the action to be taken by the decode unit are straight forward, with the exception of the way PE activity is handled. If a PE is currently inactive, the microprogram enters a loop that decodes the instruction but does not execute it (unless the instruction causes the PE to change its activity). Thus, the decode unit in a PE is always active whether or not computations are to be carried out in the PE.

The PE Instruction Processor is similar in construction to the CU Instruction Processor, i.e. it is a 32 bit microprocessor chip set incorporating an ALU and a set of 32 bit registers. The internal registers are used to provide a set of eight general purpose MAP registers, denoted AC[0]-AC[7], i.e., these registers are used by the MAP assembly language programmer to perform arithmetic/logical operations, and as index registers. The remaining registers are used by the microprograms and implement the SELECT register.

The SELECT register shown in Figure 2 is the mechanism used to save the results of up to eight conditions in the PE. Any MAP instruction set should include a set of associative instructions similar to the type 4 instructions (codes 4 01 through 4 12) shown in the Appendix. Each associative instruction includes an 8 bit Key field and an 8 bit Mask field. The "SELECT" instruction (code 4 00) activates all PEs such that

$$(C(SELECT)=Key) \wedge Mask$$

results in all eight bits being set. Thus, the SELECT instruction computes the activity of all PEs as a function of the contents of the SELECT register, Key field, and Mask field. In the Appendix, the instruction "SELECT,R" is used to restrict the domain of selection to the currently active PE subset. The "COMSEL" instruction performs the same function as SELECT, except that it complements the activity state of the PEs after performing the normal SELECT. The remaining Type 4 associative instructions are used to set/reset bits in the SELECT regis-

ter, depending on conditions that exist in the PE at the time of their execution. For example, "SETPL" is used with a Key and a Mask field to manipulate the contents of the SELECT register as follows: If a designated AC is greater than or equal to zero, then the SELECT register is changed by

$$\text{SELECT} \leftarrow \text{SELECT} \wedge \neg \text{MASK}$$
$$\text{SELECT} \leftarrow \text{SELECT} \vee \text{KEY} .$$

Thus, the Mask field specifies one or more bits that should be set or reset if the AC is nonnegative, and the Key field determines their settings.

The associative instructions are implemented in firmware, with the SELECT register being implemented as one of the ALU registers. The decode unit is responsible for both testing and setting the activity flag for the PE. Since associative instructions are implemented almost entirely by microprograms, the associative instruction repertoire is as flexible and general as experience dictates that it should be; the hardware organization does not fix the selection strategy. The SELECT register can be loaded and stored just as any accumulator. This feature allows an almost unlimited number of selection conditions to be saved with the PE.

The associative instructions with codes 4 0D through 4 10 imply some form of comparison and selection across PEs. This can be done in one of two ways: Incorporate additional hardware as suggested in Technical Report No. CU-CS-070-76, (pages 16-18); or to implement the test within the CU such that each active PE's designated register is sequentially tested for maximum (or minimum) by the CU. The first method requires extensive hardware, and the second method is inefficient at program execution time; for simplicity, the latter method is assumed.

IV. Distribution Switch Organization

The Distribution Switch must allow any control unit to pass/receive information to/from any subset of processing elements in the system. This requires a switching mechanism logically equivalent to

an 8 × 1024 crossbar switch in one extreme; or a single, common multiplexed bus in the opposite extreme. The full crossbar organization minimizes bus conflicts at the expense of complex hardware, while the shared bus minimizes hardware complexity at the expense of expected bus transfer time. Each machine instruction (PE or CU) requires multiple microcycles to carry out execution; for bit slice microprocessors, each microcycle is in the range of 100-200 ns and preliminary estimates indicate that the average microprogram length exceeds 10 microinstructions. The bus can be constructed to transfer information in 80-100 ns; hence, it is possible to share data paths between the CUs and the PEs. The mechanism used in MAP, shown in Figure 4, is composed of an 8 × 16 crossbar switch where each CU has a dedicated crossbar, and sets of 64 PEs share an orthogonal crossbar. Each crossbar shared by PEs is referred to as a bus sector.

A control unit broadcasts information to its PEs by causing crosspoint connections at its crossbar and all sector crossbars where the sector contains PEs currently allocated to the CU. If two CUs have no PEs allocated within a common sector, then they can simultaneously transmit data to their PEs; otherwise a transmission conflict arises, and hardware is provided to arbitrate the conflict. The Bus Sector Allocation unit is the conflict arbiter, and is discussed later in this section. Since PEs share crossbars, any operating system implemented on MAP must pay particular attention to the physical location of PEs allocated to a given control unit. Although PEs are logically identical to user programs executing on CUs, they must be treated as individuals by the operating system.

Since the bus is shared, data to be broadcast are placed on the bus only when the bus is allocated and the PEs have been set up to receive data using the input control register (ICTL) and the output control register (OCTL), shown in Figure 3.

The ICTL and OCTL registers work in a similar manner. A register is intially loaded with a value representing a delay time. For each cycle to which the CU controlling the operation is not blocked from the data bus, the register is decremented by one. When the register reaches zero, a signal is generated which, in the case of the ICTL register, causes a word to be gated from the PE data bus to the PE Instruction Processor. For the OCTL register, the signal causes the data

to be gated to the PE data bus from the PE processor. 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 bus.

Data streams need not always originate from or terminate to central memory. The CU may cause the contents of the bus to be fed back to the bus, so that 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. One PE may pass data to any other PE in a direct manner, but all PEs cannot simultaneously shift data to "adjacent" PEs.

The Bus Sector Allocation unit is a critical unit of the Distribution Switch which provides the following functions:

- Establish data routes through the switch,
- Resolve conflicting transmission requests,
- Enable the ICTL/OCTL countdown clocks whenever an owning CU is not blocked by transmission conflict.

The inputs to this unit are the ID register contents, Bus request signals, and a Sector Mask for each CU. The Sector Mask is a 16 bit register, maintained by CUPI, which contains a 1 in each bit position (corresponding to a sector) in which the CU is allocated PEs. An outline of the Bus Sector Allocator is shown in Figure 5. The Sector Conflict Test subunit determines if any two CUs share a sector; the Transmission Conflict unit determines if two CUs are simultaneously attempting to use the Distribution Switch. If there is no sector conflict and no transmission conflict, then the ICTL/OCTL clocks for all PEs are enabled. In the event of transmission conflict, the Transmission Conflict Unit arbitrates to select one CU for transmission; the Route Unit then uses the appropriate Sector Mask to broadcast the ID and the CU's DBR to the appropriate sectors.

V. Process Intercommunication

The process intercommunication scheme and supporting hardware discussed in this section is rather specific in nature, and it presumes that a significant portion of the mechanism is fixed by firmware in the CUPI. The approach illustrates an integration of parts of the operating system into the firmware and hardware of a machine. More general intercommunication mechanisms can be supported by the hardware by altering the microprograms. It is assumed that the operating system is distributed over all CUs, and that each CU supports 4 levels of multiprogramming. The scheme is open to revision and refinement as experience with the system dictates.

A. Intercommunication Instructions

Each process supported by MAP is identified by its ID register content. Let $ID_k[i,j]$ denote the k^{th} bit of the ID register for process j on control unit i, $(0 \le i \le 7, 0 \le j \le 3)$. If

$$ID_k[i,j] \wedge ID_k[i',j'] = 1 \quad \text{for some } k \ (0 \le k \le 7)$$

then process j on CU_i can <u>cooperatively communicate</u> with process j' on $CU_{i'}$, and vice versa; otherwise, no communication is possible between the two processes. If, in addition

$$ID_k[i',j'] = 1 \Rightarrow ID_k[i,j] = 1 \text{ for all } k \ (0 \le k \le 7)$$

then process j has <u>privilege</u> with respect to process j', (no two processes may have identical ID contents).

Op codes 0 00 through 0 16 implement interprocess communication. Each process state is partially described by two single bit registers to aid in communication: ARM[i,j] is set (true) if a privileged interrupt can be accepted by process j; ABLE[i,j] is set (true) if a cooperative interrupt can be accepted by process j. Thus ARM[i,j] and ABLE[i,j] are binary semaphores to protect critical sections within process j. The 22 bit INT[i,j] register is used for cooperative interrupts by forcing the interrupted process to branch to the location contained in INT whenever a cooperative interrupt is received by process j. The 8 bit SIG[i,j] register identifies a receiver process for an interprocess communication.

The mechanism described in the preceding two paragraphs allows privileged and cooperative interrupts to take place (if certain condi-

tions are true). Privileged interrupts correspond to the case of a higher priority process preempting a lower priority process, where priority (privilege) is determined by the settings of the relevant two ID registers. Cooperative interrupts, i.e. message passing, are allowed between two processes whose ID registers satisfy the cooperative communication predicate. The notion of privilege is also used to control the alteration of critical registers of one process by another. Since interprocess communication may require that ID, ARM, ABLE, and certain other registers be tested before communication can take place, and because these registers exist (logically) in distinct CUs, the CUPI is used to implement process intercommunication. Before discussing CUPI organization, the details of privileged and cooperative communication will be given.

Op codes 0 00 through 0 0D are privileged instructions that can be executed by process j with process j' receiving the effect of the instruction only if process j has privilege with respect to process j'. Each instruction is executed as a CUPI microprogram, and the descriptions shown in the Appendix include the two microprogram subroutines ENQUEUE and DEQUEUE used to handle interrupt queuing, (explained later in this section).

The PREEMPT instruction (op code 0 01) can take place if the preempted process is not in a critical section (i.e., ARM[i,j'] is true). If ARM[i',j'] is false, then the interrupt is queued and the preempting process is released; otherwise, ARM[i',j'] is reset, the instruction counter for process j' is saved, a parameter is passed via CAC[0] in each process, and the preempted process is started at an effective address specified by the preempting process. Setting and resetting the ARM flag is a privileged instruction, but processes have privilege with respect to themselves. The CLEAR instruction (op code 0 04) restores the preempted process. The remaining privileged instructions allow loading and storing of the ID register, INT register, SEC (sector mask) register, and process state registers (as specified in the CUPI description). A further restriction is placed on the ID register loading as follows: A new ID content is restricted such that

$$ID_k[i',j'] \leftarrow ID_k[i,j]-y$$

where

$$0 \leq y \leq ID_k[i,j]$$

i.e. process j cannot define a new ID content where more bits are set in the new ID register than are set in ID[i,j]. This prevents a CU from increasing its communicative power either directly or indirectly.

LDSIG and STSIG are unprivileged instructions to allow a process to manipulate its SIG register in setting up for a communication.

Cooperative communication is accomplished using the instructions with op codes 0 10 through 0 13. When SIGNAL is executed by process j, a check is made to see that communication is allowable between processes j and j' (the ID of j' is again loaded into SIG[i,j]). In the case that process j' is in a critical section and cannot receive a message, the cooperative message is queued and the sending process is released. If the message can be received by process j', then the instruction counter is saved, a parameter is passed via the CAC[0] registers, and process j' resumes execution at the address specified by INT[i',j']. Process j' will normally have loaded INT[i',j'] with the address of a message receive procedure, but if process j privilege over j', then it has the option of forcing process j' to other code by altering its INT immediately before signalling. Race conditions on the setting of INT may arise.

B. CUPI Organization

The Control Unit Processor Interface is a single bit slice microprocessor used to implement the intercommunication instructions and to control the Bus Sector Allocation unit. CUPI is invoked by CU_i when CU_i stores a type 0 instruction into MM location 3FFFF8+i. If more than one CU calls CUPI at one time, CUPI will service the requests on a round robin basis. The calling CU is blocked in a busy wait until CUPI has finished processing the instruction. CUPI also uses other high address MM locations to save the state of all 32 processes and to implement the interrupt queue. Table 1 shows the organization of high address MM, and in particular, locations 3FFF80-3FFFAF are used to maintain the interrupt queue. All queue operations are implemented in CUPI microcode. MM locations 3FFF90-3FFFAF contain the ID register contents in the 8 most significant bits, and a pointer to a queue

header in the least significant bits for each process respectively. The queue header is a double word containing the given process's ARM, ABLE, SECTOR, SIG, and INT register contents and a queue length and link. Each entry in the queue contains another link and the calling process's CAC[0] (argument pointer). The CUPI microprogram subroutine, ENQUEUE, enters messages into the queue, and DEQUEUE removes messages from the queue. The queue routines are used as mentioned above and in the Appendix. Locations 3FFFB0-3FFFB7 are used to save the previous instruction counter content during a SIGNAL operation, and 3FFFB8-3FFFBF serve the same purpose for PREEMPT instructions.

CUPI is organized around a bus similar to that of a PE, (see Figure 6). The bus is used to maintain the current ID, SECTOR, and INT registers for the 8 processes that are active on control units. As a process is loaded onto a CU, the registers are loaded within CUPI. The ID and SECTOR registers interface with the distribution switch via the Bus Sector Allocator as indicated in Figure 5. The INT registers are gated back to appropriate CUs during SIGNAL instruction execution. Since CUPI must access MM, a 32 bit data register and a 22 bit address register are included as shown in the figure. The SIG registers are implemented within the CUPI instruction processor (ALU). The operation of CUPI is described by the op codes in the Appendix. Notice that the strategy for interprocess communication remains much more flexible than the particular scheme given here, since it is almost totally implemented in the CUPI firmware.

VI. Main Memory and the I/O System

A block diagram of the Main Memory and I/O subsystem appears in Figure 7. Main Memory is divided into 16 physically distinct 256K word modules each with three ports. Each module may, in turn be subdivided to incorporate interleaving if there is a need. MM modules 0 through 7 are configured such that CU_i has a preferred path to module i, and CUPI has a preferred path to module 15. Any of the nine units has access to any of the 16 memory modules via a shared memory bus. Likewise, the I/O subsystem uses a distinct bus to access any of the modules. No priority among the three ports has been set at this time, although the most likely

scheme would give the I/O subsystem the highest memory access priority, followed by the preferred CU, followed by the shared memory bus. This arrangement was chosen to reduce memory conflicts among CUs as much as possible, where a CU will normally have its instruction stream loaded in its corresponding MM module. In the event that an instruction stream (program) and associated global data do not fit within the preferred module, the memory can be allocated by the operating system so that most of the code fits within the preferred module, but may overlap into neighboring modules. If all CUs require more than 256K words of MM, then memory bus conflicts will occur. The code for the distributed operating system will be loaded into modules 8 to 14.

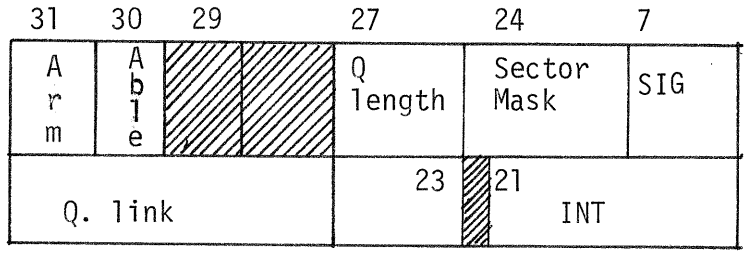
The CUMAR for each CU can hold a 22 bit address, hence any CU can generate any MM address. However, memory loading is greatly simplified if the programs for a given CU are assembled relative to location 0. This requires that some form of base register be incorporated into each CU so that addresses that are apparently absolute within the CU are relative to the first allocated location within MM. This problem is solved by constructing each CU's microprograms such that effective addresses are relocated by an internal base register (set on process allocation) before being loaded into CUMAR. Firmware address relocation is used, rather than hardware or software modification of the address. This scheme implies that a process uses a contiguous set of memory locations within MM. CUPI calls will bypass the relocation mechanism.

The I/O subsystem must have access to each memory module in order to load programs and to accomplish normal I/O. An analysis of the I/O subsystem may possibly require that multiple shared buses are required to perform I/O, depending on the bandwidth of the I/O subsystem. The initial design of the entire system discourages high operating rates for the I/O system, since each I/O implies more use of the data bus system interconnecting CUs and PEs. The overall design philosophy of MAP dictates that high I/O programs will not perform well on the machine.

MM ADDRESS RESERVATION

3FFFFF:	CUPI call from CU_7	
⋮	⋮	
3FFFF8:	CUPI call from CU_0	
3FFFF7:	110 Device codes	58 device codes
⋮		
3FFFC0:	" " "	
3FFFBF:	IC_7 save space for preempt	
3FFFBE:	IC_6 " " " "	
3FFFBD:	IC_5 " " " "	
3FFFBC:	IC_4 " " " "	
3FFFBB:	IC_3 " " " "	
3FFFBA:	IC_2 " " " "	
3FFFB9:	IC_1 " " " "	
3FFFB8:	IC_0 " " " "	
3FFFB7:	IC_7 " " " signal	
⋮		
3FFFB0:	IC_0 " " " "	
3FFFAF:	ID_7 for process 3	32 ID registers. (8 bit ID and address for message queue)
3FFFAE:	ID_7 " " 2	
3FFFAD:	ID_7 " " 1	
3FFFAC:	IC_7 " " 0	
3FFFAB:	ID_6 " " 3	
⋮		
3FFF90:	ID_0 " " 0	
⋮		
3FFF8F:	Que header for CU_7	
3FFF8E:	" " " "	
⋮		
3FFF81:	Que header for CU_0	
3FFF80:	" " " "	
3FFF7F:	1024_{10} word queue space	
⋮		
3FFB80:		

Que Header (typical)



Que entry (typical)

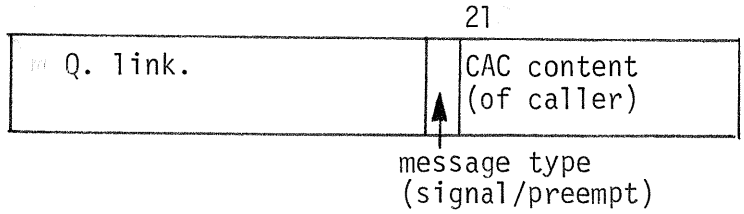
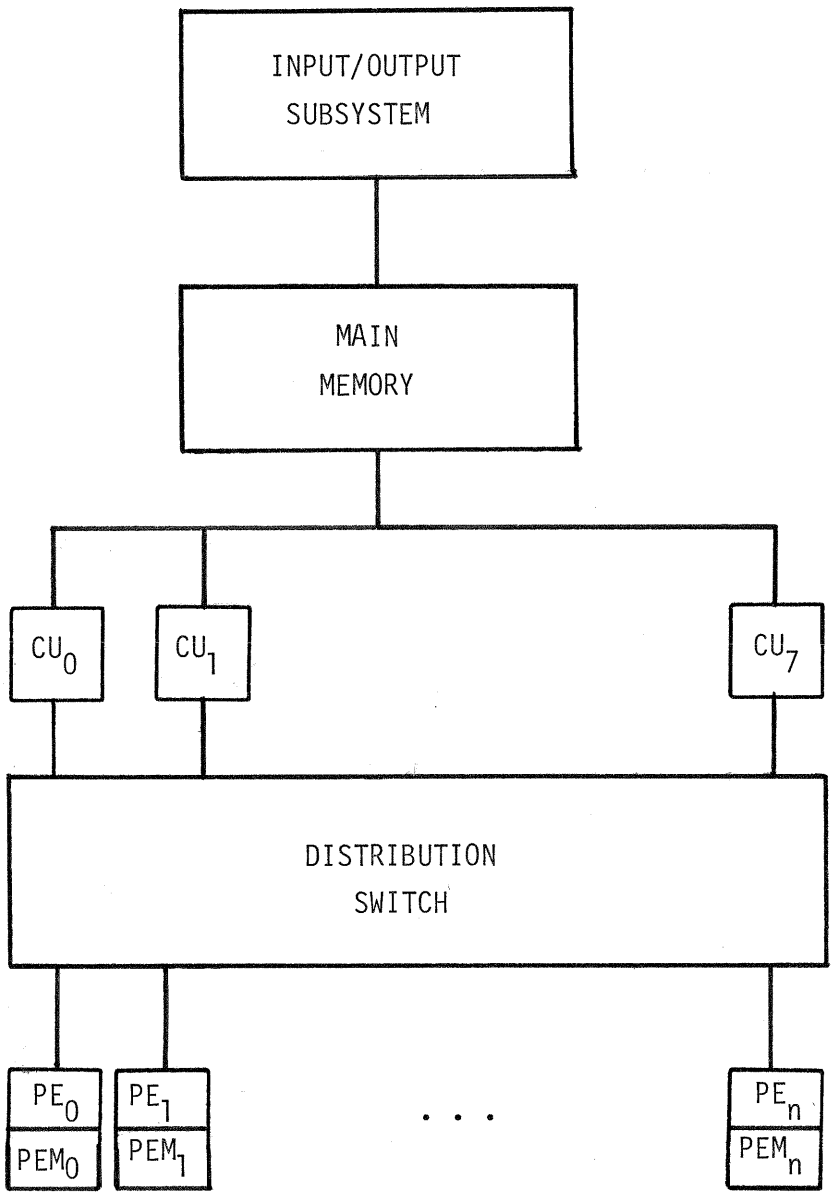
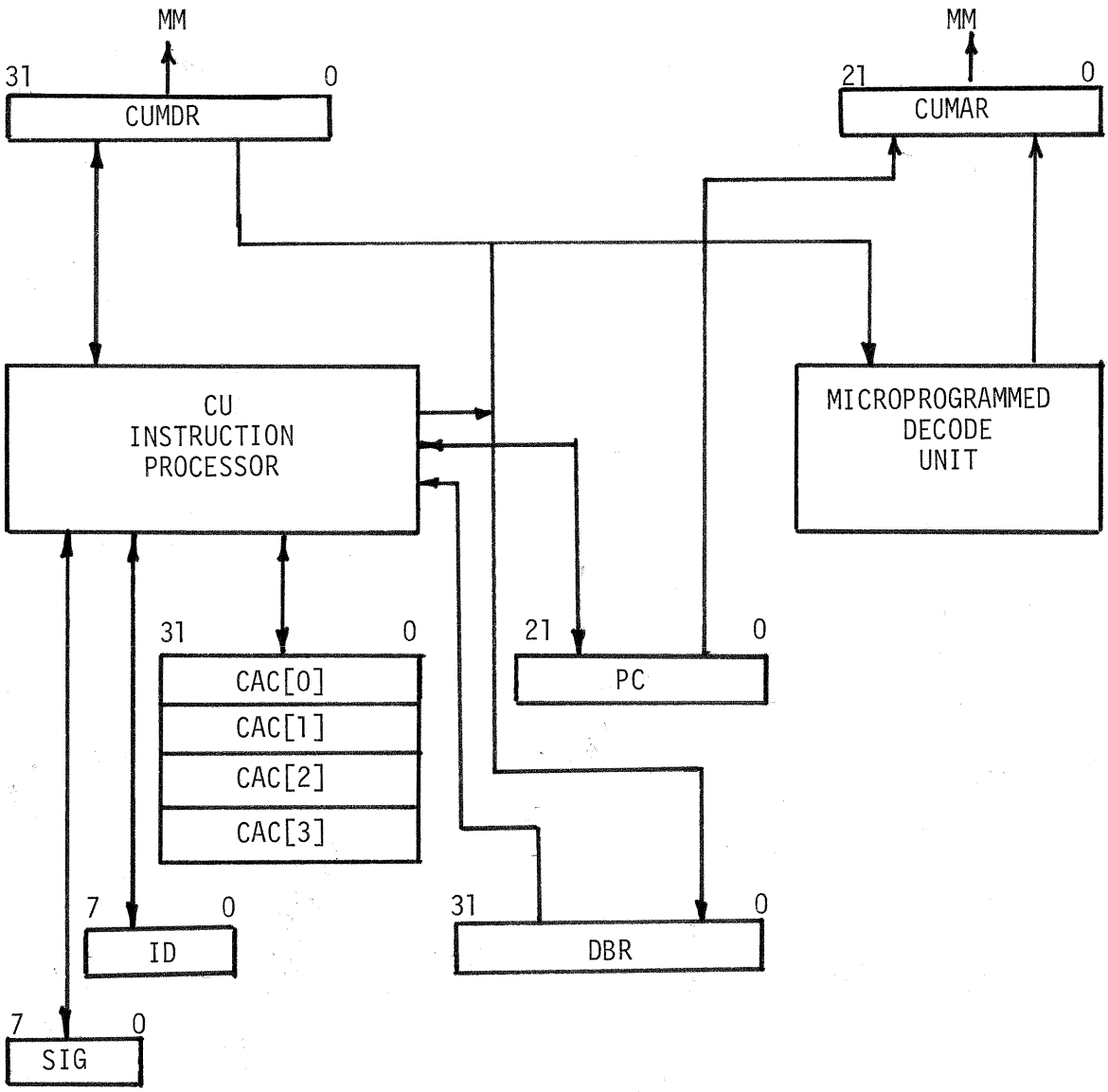


Table 1



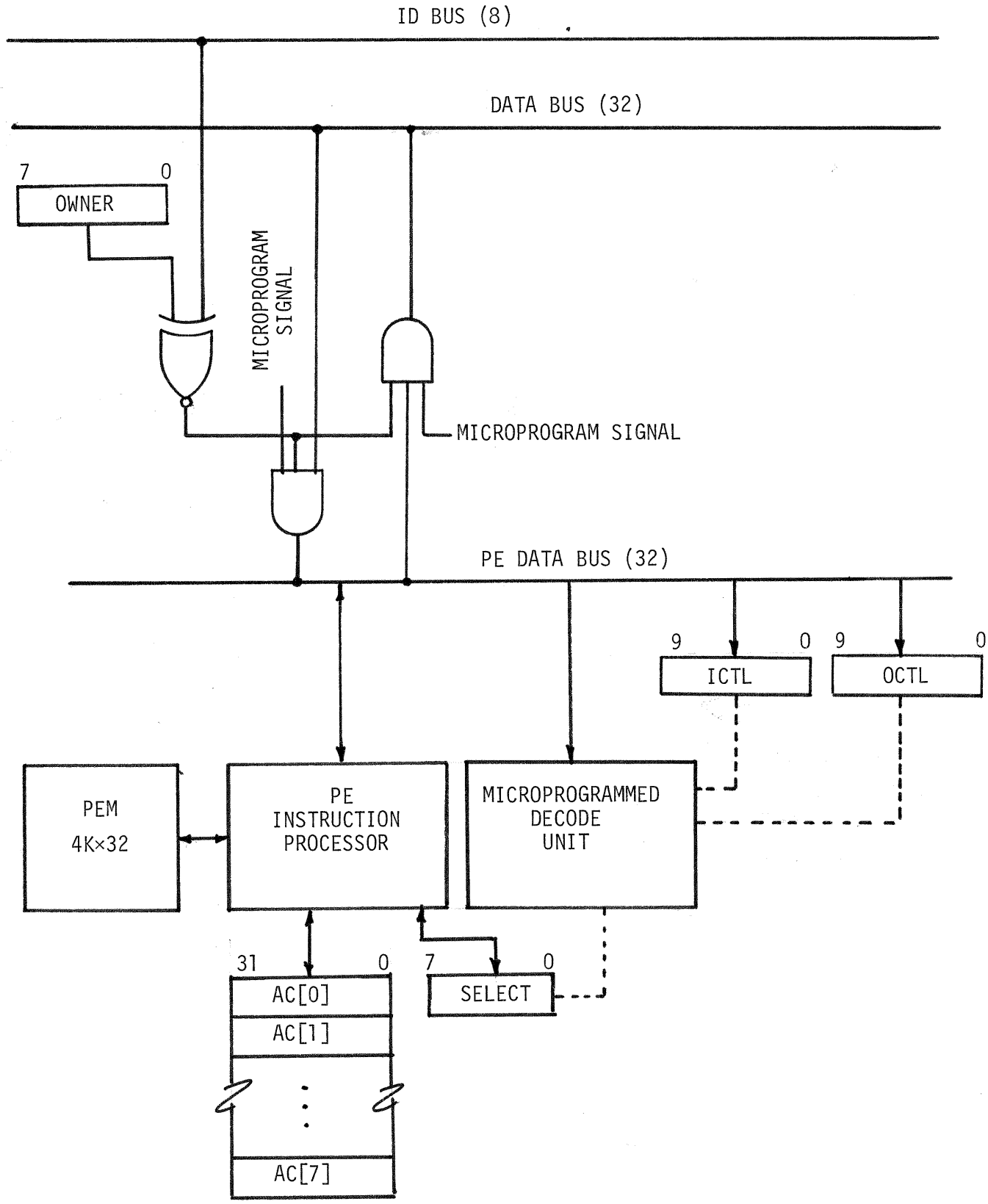
MAP ORGANIZATION

FIGURE 1



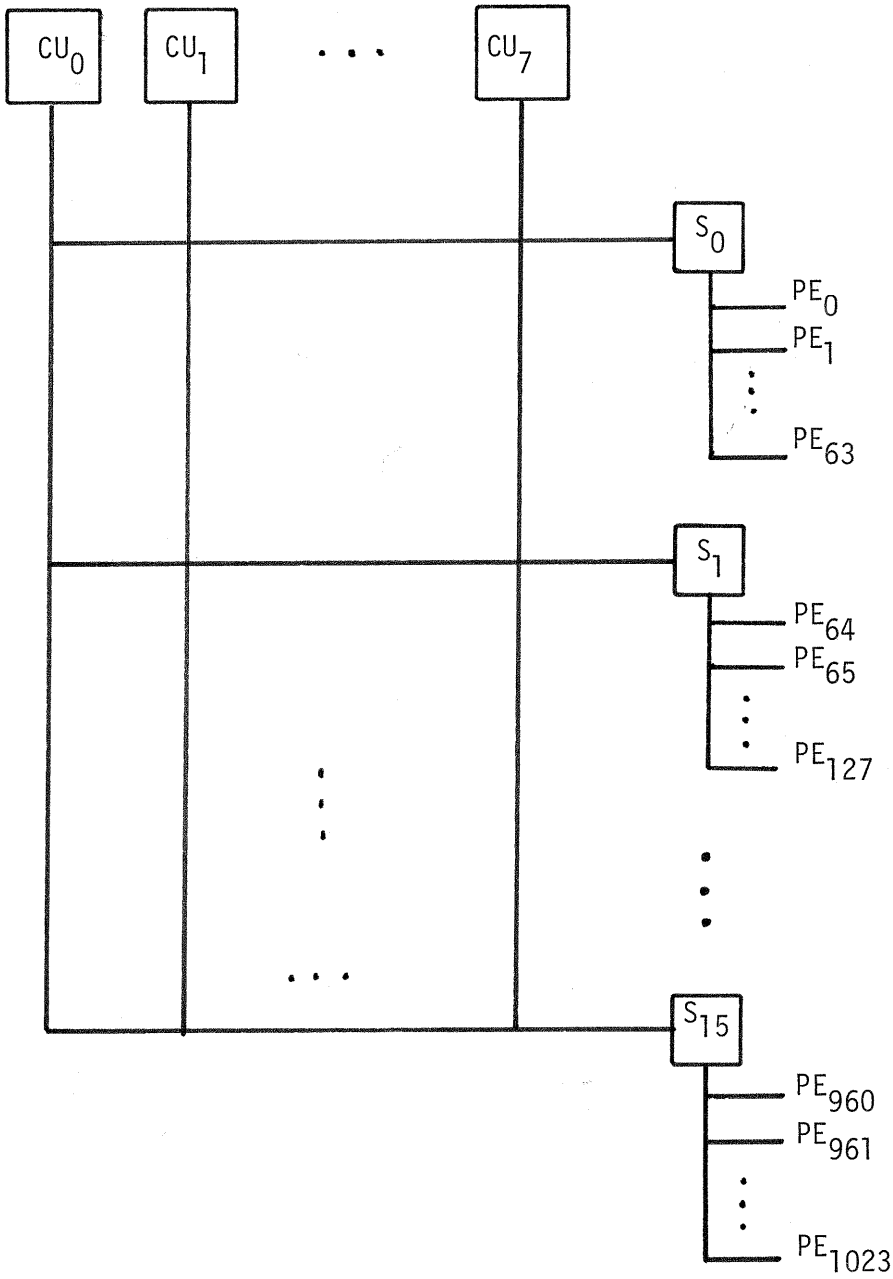
CONTROL UNIT ORGANIZATION

FIGURE 2



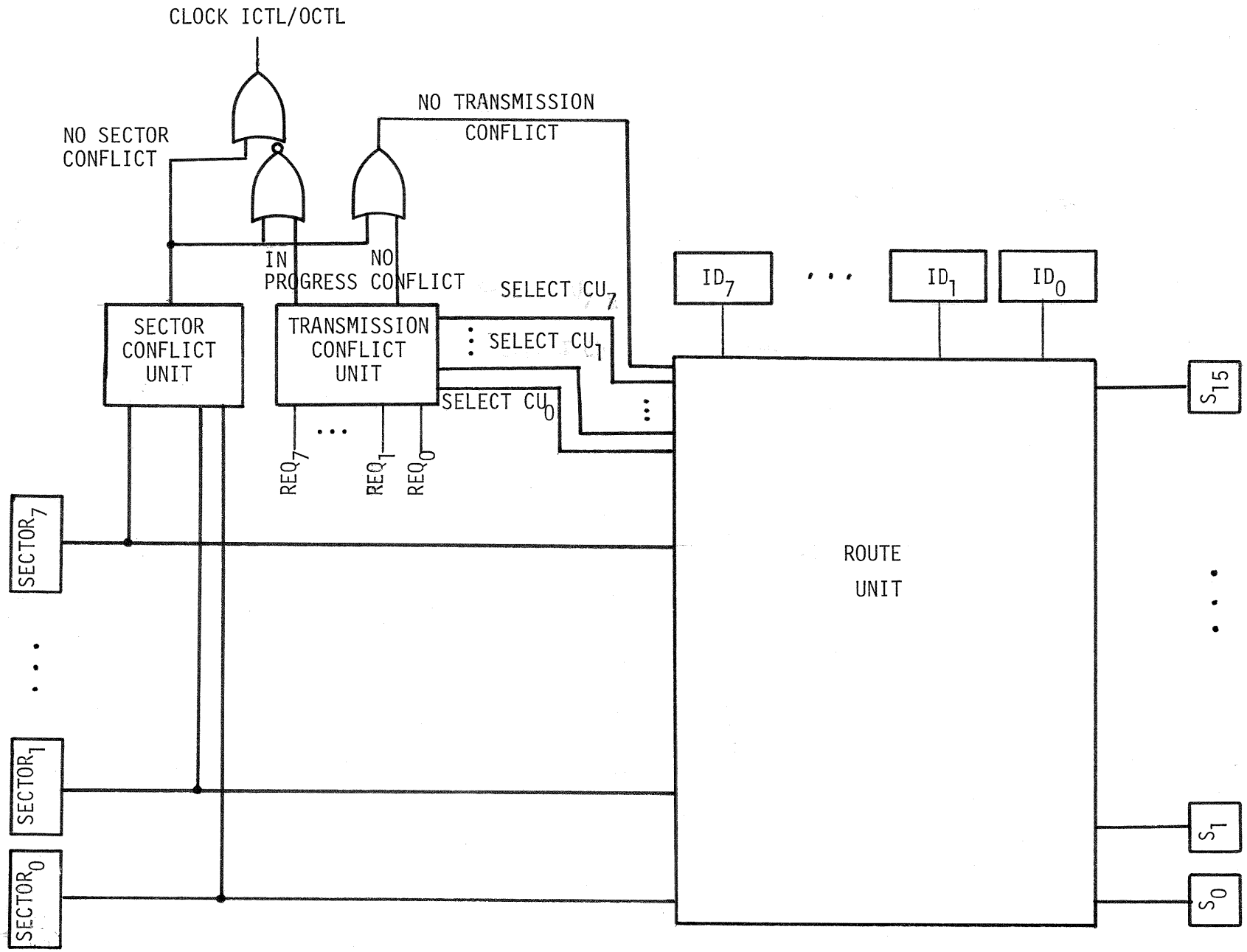
PROCESSING ELEMENT ORGANIZATION

FIGURE 3



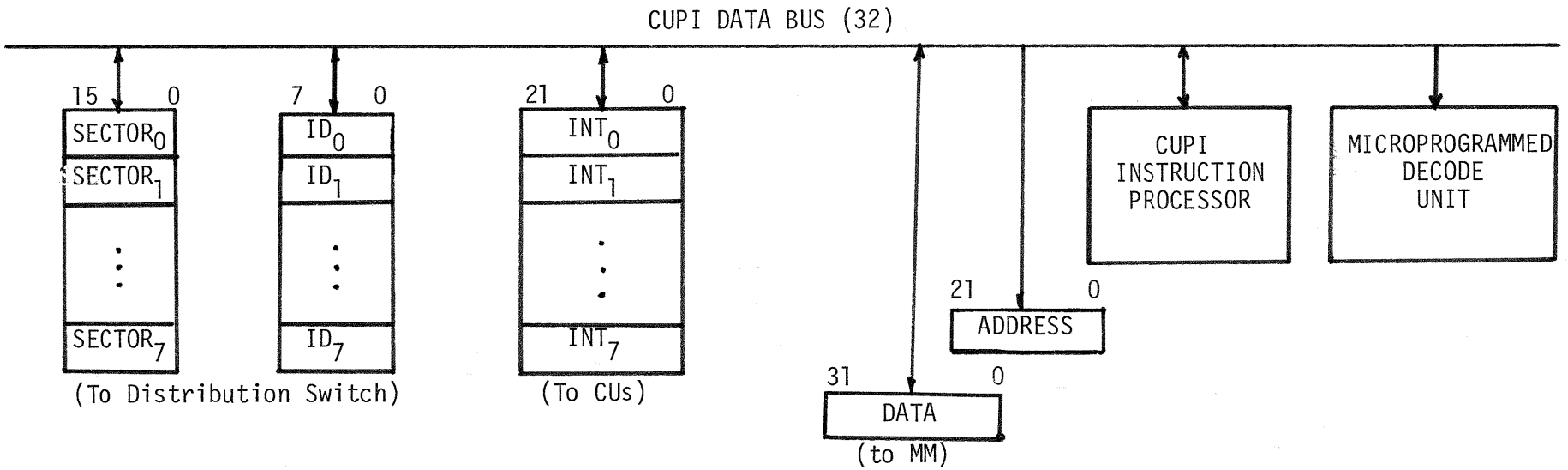
DISTRIBUTION SWITCH

FIGURE 4



BUS SECTOR ALLOCATOR

FIGURE 5



FLIPFLOPS

ARM

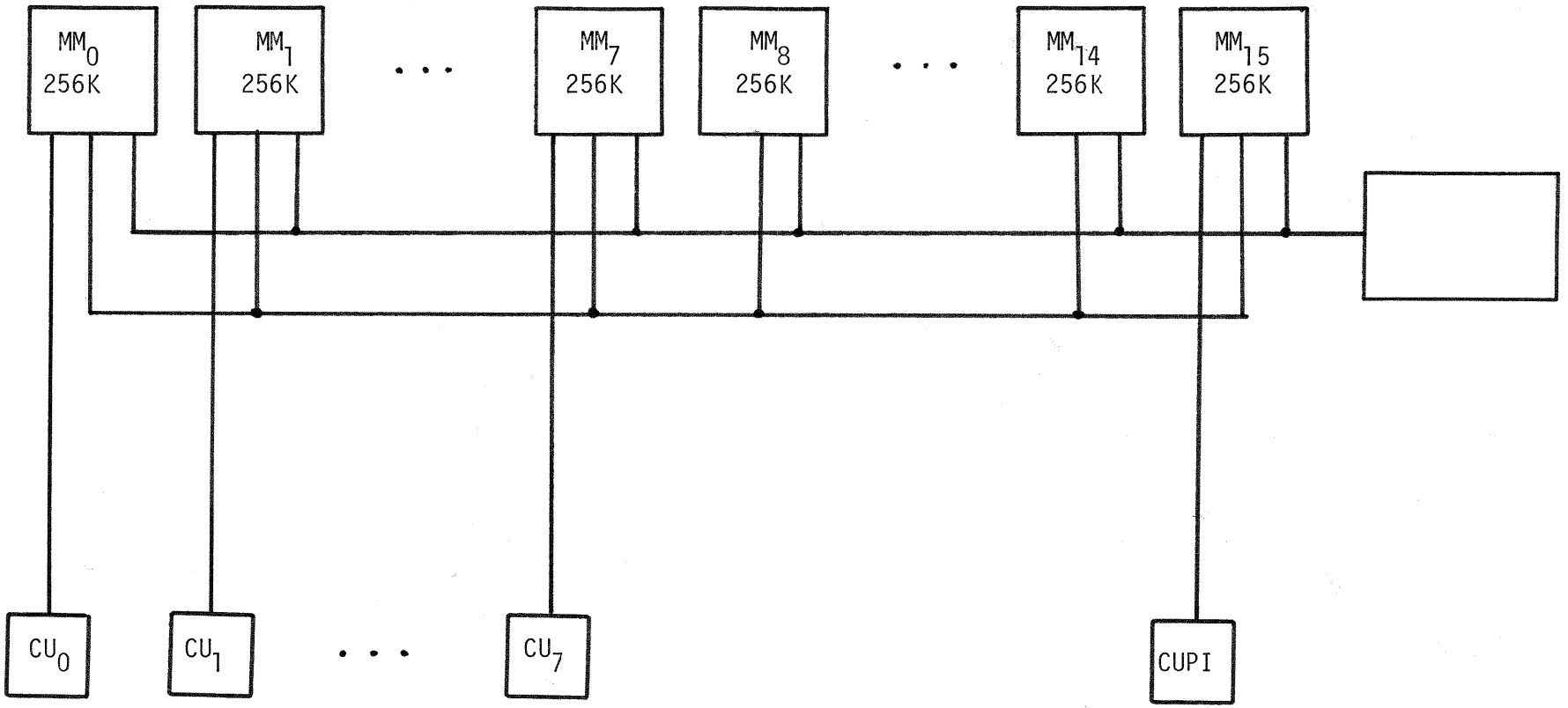
ABLE

OTHER REGISTERS IN INSTRUCTION PROCESSOR

SIG_0-SIG_7 (8-bit)

CU PROCESSOR INTERFACE

FIGURE 6



MAIN MEMORY ORGANIZATION

FIGURE 7

A P P E N D I X
A SAMPLE INSTRUCTION SET

CU REGISTERS
PC - PROGRAM COUNTER.
ID - 8 BIT IDENTIFICATION REGISTER
SIG - 8 BIT REGISTER USED FOR SIGNAL/PREEMPT INSTRUCTIONS.
INT - 22 BIT REGISTER CONTAINING THE ADDRESS OF NEXT INSTRUCTION TO BE EXECUTED AFTER A SIGNAL INSTRUCTION (CODE 0,21) IS ACCEPTED BY THE CU.
SECTOR - 16 BIT REGISTER CONTAINING A SECTOR REQUEST MASK FOR THIS CU. BIT I IS SET IF THE CU HAS PE-S ALLOCATED IN SECTOR I.
CAC[0] - 32 BIT REGISTER FOR GENERAL PURPOSE USE.
CAC[1..3] - 32 BIT REGISTERS FOR GENERAL PURPOSE USE, INCLUDING MAIN MEMORY INDEXING.

PE REGISTERS
ICTL - 10 BIT REGISTER TO CONTROL PE INPUT OPERATIONS,(ASSEMBLES AS AC[8]).
OCTL - 10 BIT REGISTER TO CONTROL PE OUTPUT OPERATIONS,ASSEMBLES AS AC[9]).
SELECT - 8 BIT REGISTER TO DETERMINE THE STAT OF THE COMPUTATION IN THE GIVEN PE, (ASSEMBLES AS AC[10]).
OWNER - 8 BIT REGISTER THAT MATCHES THE I REGISTER OF THE CU TO WHICH THE PE IS ALLOCATED.
AC[0] - 32 BIT REGISTER FOR GENERAL PURPOSE USE, EXCEPT INDEXING.
AC[1..7] - 32 BIT REGISTERS FOR GENERAL PURPOSE USE, INCLUDING INDEXING PE MEMORY.

```
TYPE OP MNEMONIC DESCRIPTION
                (***THE FOLLOWING PREDICATE IS USED IN A NUMBER OF OP CODE
                    DESCRIPTIONS FOR TYPE 0  INSTRUCTIONS
                     P(I,J,I',J') = IF ID[I',J']=SIG[I,J] AND
                                       ID[I',J'] .CONTAINED-IN. ID[I,J]
                                        THEN TRUE ELSE FALSE ***)
                (*** ALL INSTRUCTIONS ARE EXECUTED BY PROCESS J ON CU I ***)
                (*** EACH TYPE 0 DESCRIPTION SHOULD BE FOLLOWED BY
                     AN INTERRUPT TO CU[I] ***)
  0 00 HALT     HALT CONTROL UNIT I'.
  0 01 PREEMPT  IF P(I,J,I',J') THEN
                 BEGIN
                  ENQUEUE;
                  IF ARM[I',J'] THEN
                   BEGIN
                    ARM[I',J'] <- FALSE;
                    CU[I'] FINISHES CURRENT INSTRUCTION CYCLE;
                    MM[F(I',J')] <- IC[I',J'];
                    IC[I',J'] <- EA;
                    CAC[0][I',J'] <- CAC[0][I,J];
                   END
                 END
  0 02 ARM        IF P(I,J,I',J') THEN ARM[I',J'] <- TRUE
  0 03 DISARM   IF P(I,J,I',J') THEN
                 BEGIN
                  ARM[I',J'] <- FALSE;
                  ABLE[I',J'] <- FALSE
                 END
  0 04 CLEAR    IF P(I,J,I',J') THEN
                 BEGIN
                  IC[I',J'] <- MM[F(I',J')];
                  ARM[I',J'] <- TRUE
                 END
  0 05 LDID     IF P(I,J,I',J') THEN ID[I',J'] <- MM[EA]
  0 06 STID     IF P(I,J,I',J') THEN MM[EA] <- ID[I',J']
  0 07 LDINT    IF P(I,J,I',J') THEN INT[I',J'] <- MM[EA]
  0 08 STINT    IF P(I,J,I',J') THEN MM[EA] <- INT[I',J']
  0 09 LDSEC    IF P(I,J,I',J') THEN SECTOR[I',J'] <- MM[EA]
  0 0A STSEC     IF P(I,J,I',J') THEN MM[EA] <- SECTOR[I',J']
  0 0B LDST     IF P(I,J,I',J') THEN
                 BEGIN
                  LOAD ARM, ABLE, SECTOR, SIG, AND INT FROM MM[F(I',J')];
                   IF QUEUE LENGTH > 0 THEN
                    BEGIN
                     DEQUEUE;
                     IF SIG-TYPE THEN SIGNAL ELSE PREEMPT
                    END
                 END
  0 0C STST     IF P(I,J,I',J') THEN
                    SAVE ARM,ABLE,SECTOR,SIG, AND INT AT MM[F(I',J')]
  0 0D MTRCTL   IF P(I,J,I',J') THEN
                 CASE R-FIELD OF
                  0:DISCONTINUE MONITORING;
                  1:START MONITORING J;
                  2:START MONITORING J';
                  3:UNUSED
                 END
```

```
TYPE OP MNEMONIC DESCRIPTION
  0 0E LDSIG     SIG[I,J] <- MM[EA]
  0 0F STSIG     MM[EA] <- SIG[I,J]
  0 10 SIGNAL    IF ID[I',J'] = SIG[I,J] AND
                    ID[I',J'] .INTERSECT. ID[I,J] > 0 THEN
                     BEGIN
                      ENQUEUE;
                      IF ARM[I',J'] THEN
                      BEGIN
                        IF ABLE[I',J'] THEN
                         BEGIN
                          ABLE[I',J'] <- ARM[I',J'] <- FALSE;
                          CU[I'] FINISHES CURRENT INSTRUCTION;
                          MM[F(I',J')] <- INT[I',J'];
                           IC[I',J'] <- INT[I',J'];
                          CAC[0][I',J'] <- CAC[0][I,J];
                          ARM[I',J'] <- TRUE
                         END
                       END
                      END
  0 11 ENABLE    ABLE[I,J] <- TRUE
  0 12 DISABLE   ABLE[I,J] <- TRUE
  0 13 BMQE      IF MESSAGE QUEUE FOR PROCESS J IS EMPTY, THEN GOTO EA
  0 14 BINA      IF NO ID EXISTS SUCH THAT
                    ID[I',J'] = SIG[I,J] THEN GOTO EA
  0 15 BDARM     IF .NOT.ARM[I'J'] THEN GOTO EA
  0 16 BDABL     IF .NOT.ABLE[I',J'] THEN GOTO EA
  0 17 -
  0 18 (RESERVED FOR I/O INSTRUCTIONS)
  0 19 (DITTO)
  0 1A (DITTO)
  0 1B (DITTO)
  0 1C (DITTO)
  0 1D (DITTO)
  0 1E (DITTO)
  0 1F (DITTO)

  1 00 -
  1 01 GMCP      AC[R3] <- CAC[R1]
       GMPC      CAC[R3] <- OR(AC[R1])
       GMICR     CAC[R3] <- ICTL
       GMRIC     AC[8] <- CAC[R1R]
       GMOCR     CAC[R3] <- OCTL
       GMROC     AC[9] <- CAC[R1R]
  1 02 CM        CAC[R3] <- CAC[R1]
  1 03 CAR       CAC[R3] <- CAC[R1] + CAC[R2]
  1 04 -
  1 05 CSR       CAC[R3] <- CAC[R1] - CAC[R2]
  1 06 -
  .
  .
  .
  1 1F -
```

```
TYPE OP MNEMONIC DESCRIPTION
  2 00 CLI       CAC[R] <- OPD
  2 01 -
  2 02 -
  2 03 CAI       CAC[R] <- CAC[R] + OPD
  2 04 -
  2 05 CSI       CAC[R] <- CAC[R] - OPD
  2 06 -
  2 07 -
  2 08 GL        AC[R] <- MM[EA]
       GLIC      ICTL <- MM[EA]
       GLOC      OCTL <- MM[EA]
       GLOW      OWNER <- MM[EA]
  2 09 GS        MM[EA] <- OR(AC[R])
       GSIC      MM[EA] <- OR(ICTL)
       GSOC      MM[EA] <- OR(OCTL)
       GSOW      MM[EA] <- OWNER
  2 0A -
  .
  .
  .
  2 1F -


  3 00 CL        CAC[R] <- MM[EA]
  3 01 CS        MM[EA] <- CAC[R]
  3 02 XCHNG     EXCHANGE CONTENTS OF CAC[R] AND MM[EA].
  3 03 CAM       CAC[R] <- CAC[R] + MM[EA]
  3 04 -
  3 05 CSM       CAC[R] <- CAC[R] - MM[EA]
  3 06 BR        GOTO MM[EA]
  3 07 SUBR      CAC[R] <- PC; PC <- EA
  3 08 LSTR      AC[R] <- STREAM- MM[EA]
  3 09 SSTR      MM[EA] <-STREAM- AC[R]
  3 0A XSTR      AC[R(EVEN)] <-STREAM-> A[R(EVEN)+1]
  3 0B -
  3 0C -
  3 0D BCT0      IF NO ACTIVE PE-S, BRANCH TO EA
  3 0E BCT1      IF ONE ACTIVE PE, BRANCH TO EA
  3 0F BCTG1     IF MORE THAN ONE ACTIVE PE, BRANCH TO EA
  3 10 BXEQ      IF CAC[R1]=CAC[R2], BRANCH TO EA
  3 11 BXNE      IF CAC[R1]<>CAC[R2], (DITTO)
  3 12 BXGT      IF CAC[R1]>CAC[R2], (DITTO)
  3 13 BXLE      IF CAC[R1]<=CAC[R2], (DITTO)
  3 14 BXLT      IF CAC[R1]<CAC[R2],  (DITTO)
  3 15 BXGE      IF CAC[R1]>=CAC[R2], (DITTO)
  3 16 BXZE      IF 0=CAC[R2],BRANCH TO EA
  3 17 BXNZ      IF 0<>CAC[R2], (DITTO)
  3 18 BXPL      IF 0<CAC[R2],  (DITTO)
  3 19 BXNP      IF 0>=CAC[R2], (DITTO)
  3 1A BXNG      IF 0>CAC[R2],  (DITTO)
  3 1B BXNN      IF 0<=CAC[R2], (DITTO)
  3 1C BXCHNG    EXCHANGE THE CONTENTS OF CAC[R2] AND MM[EA];
                 IF CAC[R2] = 0, BRANCH TO ADDRESS SPECIFIED BY CAC[R1].
  3 1D PUSH      MM[MM[EA]] <- CAC[R2]; MM[EA] <- MM[EA] + 1.
  3 1E PULL      MM[EA] <- MM[EA] - 1; CAC[RR] <- MM[MM[EA]].
  3 1F -
```

```
TYPE OP MNEMONIC DESCRIPTION
                 (***FOR THE SELECT AND COMSEL INSTRUCTIONS, KEY AND MASK
                     FIELDS CAN BE COMBINED IN THE ASSEMBLY LANGUAGE
                     AS FOLLOWS
                     OPERAND FIELD = 010XX1X0
                     ASSEMBLES WITH MASK = 11100101, KEY = 01000100 ***)
  4 00 SELECT    (N=0) ACTIVATE ALL PE-S.
                 DEACTIVATE PE-S SUCH THAT THE FOLLOWING IS FALSE
                   (SELECT .EQUIV. KEY) .OR. (.NOT.MASK)
  4 00 SELECT,R  (N=1) SELECT ON CURRENTLY ACTIVE PE-S ONLY.
  4 01 COMSEL    (N=0) PERFORM SELECT INSTRUCTION AND COMPLEMENT
                 ACTIVITY OF ALL PE-S.
       COMSEL,R  (N=1) PERFORM SELECT,R INSTRUCTION AND COMPLEMENT
                 ACTIVITY OF PREVIOUSLY ACTIVE SUBSET ONLY.
                 (***ASSEMBLER CAN BE WRITTEN TO USE ORSEL IN PLACE OF COMSEL,
                     AS INCORPORATED IN THE VERSION 2 INSTRUCTION SET***)

                 (***FOR SETREL OPERATIONS IN THE ASSEMBLY LANGUAGE, A SET
                     MACHINE INSTRUCTION (CODE 4,02) MUST PRECEED THE
                     CORRESPONDING SETREL MACHINE INSTRUCTION,I.E.,
                       SETREL XCXXC101
                     SHOULD ASSEMBLE WITH
                       SET KEY = 00000101, MASK = 00000111
                       SETREL KEY = 01001000, MASK = 01001000 ***)
                 (***SETREL ASSEMBLES WITH N=0 AND CAUSES A NEW ACTIVITY
                       STATE TO BE COMPUTED.
                     SETREL,NC ASSEMBLES WITH N=1 AND SUPPRESSES
                       ACTIVITY CHANGES. ***)
  4 02 SET       SET THE SELECT REGISTER BY KEY AND MASK .
  4 03 SETPL     SET BIT IF AC[R] >= 0.
  4 04 SETNG     SET BIT IF AC[R] < 0.
  4 05 SETZR     SET BIT IF AC[R] = 0.
  4 06 -
  4 07 SETEQ     SET BIT IF AC[R] = AC[R2].
  4 08 SETNE     SET BIT IF AC[R] <> AC[R2].
  4 09 SETLT     SET BIT IF AC[R] < AC[R2].
  4 0A SETGT     SET BIT IF AC[R] > AC[R2].
  4 0B SETLE     SET BIT IF AC[R] <= AC[R2].
  4 0C SETGE     SET BIT IF AC[R] >= AC[R2].
  4 0D SETMAX    SET BIT IF AC[R] IS MAXIMUM VALUE.
  4 0E SETMX1    SET BIT OF ONE PE IF AC[R] IS MAXIMUM VALUE.
  4 0F SETMIN    SET BIT OF AC[R] IS MINIMUM VALUE.
  4 10 SETMN1    SET BIT OF ONE PE IF AC[R] IS MINIMUM VALUE.
  4 11 SETFST    SET BIT OF ONE PE.
  4 12 CLRFST    RESET BIT OF ONE PE.
  4 13 -
  .
  .
  .
  4 17 -
  4 18 SCI        AC[R] <- SHIFT CIRCULAR(AC[R],OPD)
  4 19 SLI        AC[R] <- SHIFT LOGICAL(AC[R],OPD)
  4 1A SAI        AC[R] <- SHIFT ARITHMETIC RIGHT(AC[R],ABS(OPD))
  4 1B -
  4 1C -
  4 1D -
  4 1E -
  4 1F -
```

```
TYPE OP MNEMONIC DESCRIPTION
  5 00 -
  5 01 -
  5 02 M          AC[R3] <- AC[R1]
       MICR       AC[R3] <- ICTL
       MRIC       ICTL <- AC[R1]
       MOCR       AC[R3] <- OCTL
       MROC       OCTL <- AC[R1]
       MSLR       AC[R3] <- SELECT
       MRSL       SELECT <- AC[R1]
  5 03 AR         AC[R3] <- AC[R1] + AC[R2] INTEGER
  5 04 FAR        AC[R3] <- AC[R1] + AC[R2] FLTG PT
  5 05 SR        AC[R3] <- AC[R1] - AC[R2]
  5 06 FSR        AC[R3] <- AC[R1] - AC[R2] FLTG PT
  5 07 -
  5 08 MR        AC[R3] <- AC[R1] * AC[R2] INTEGER
  5 09 FMR        AC[R3] <- AC[R1] * AC[R2] FLTG PT
  5 0A DR         AC[R3(EVEN)] <- AC[R1] DIV AC[R2]
                  AC[R3(EVEN)+1] <- AC[R1] MOD AC[R2]
  5 0B FDR        AC[R3] <- AC[R1] / AC[R2] FLOATING POINT
  5 0C ORR        AC[R3] <- AC[R1] OR AC[R2]
  5 0D ANDR       AC[R3] <- AC[R1] AND AC[R2]
  5 0E EORR       AC[R3] <- AC[R1] EOR AC[2]
  5 0F NOT        AC[R3] <- NOT AC[R1]
  5 10 NORM       AC[R3] <- NORMALIZATION(AC[R1]), AC[R2] <- SHIFT COUNT
  5 11 FIX        AC[R3] <- ENTIER(AC[R1])
  .
  .
  .
  5 18 SC         AC[R3] <- SHIFT CIRCULAR(AC[R1],AR[R2])
  5 19 SL         AC[R3] <- SHIFT LOGICAL (AC[R1],AC[R2])
  5 1A SA         AC[R3] <- SHIFT ARITHMETIC RIGHT(AC[R1],ABS(AC[R2])
  5 1B -
  5 1C -
  5 1D -
  5 1E -
  5 1F -

  6 00 LI         AC[R] <- OPD
       LICI       ICTL <- OPD
       LOCI       OCTL <- OPD
  6 01 -
  6 02 -
  6 03 AI         AC[R] <- AC[R] + OPD
       AICI       ICTL <- ICTL + OPD
       AOCI       OCTL <- OCTL + OPD
  6 04 -
  6 05 SI         AC[R] <- AC[R] - OPD
       SICI       ICTL <- ICTL - OPD
       SOCI       OCTL <- OCTL - OPD
  6 06 -
  .
  .
  .
  6 1F -
```

```
TYPE OP MNEMONIC DESCRIPTION
  7 00 L          AC[R] <- PEM[EA]
       LIC        ICTL <- PEM[EA]
       LOC        OCTL <- PEM[EA]
       LSL        SELECT <- PEM[EA]
  7 01 S          PEM[EA] <- AC[R]
       SIC        PEM[EA] <- ICTL
       SOC        PEM[EA] <- OCTL
       SSL        PEM[EA] <- SELECT
  7 02 -
  7 03 AM         AC[R] <- AC[R] + PEM[EA]
       AMIC       ICTL <- ICTL + PEM[EA]
       AMOC       OCTL <- OCTL + PEM[EA]
  7 04 FAM        AC[R] <- AC[R] * PEM[A]
  7 05 SM         AC[R] <- AC[R] - PEM[EA]
       SMIC       ICTL <- ICTL - PEM[A]
       SMOC       OCTL <- OCTL - PEM[EA]
  7 06 FSM        AC[R] <- AC[R] - PEM[EA] FLOATING POINT
  7 07 -
  7 08 MM         AC[R] <- AC[R] * PEM[EA]
  7 09 FMM        AC[R] <- AC[R] * PEM[EA] FLOATING POINT
  7 0A DM         AC[R] <- AC[R] DIV PEM[EA],
                  ACR(EVEN)+1] <- AC[R(EVEN)] MOD PEM[EA]
  7 0B FDM        AC[R] <- AC[R] / PEM[EA] FLOATING POINT
  7 0C ORM        AC[R] <- AC[R] OR PEM[EA]
  7 0D ANDM       AC[R] <- AC[R] AND PEM[EA]
  7 0E EORM       AC[R] <- AC[R] EOR PEM[EA]
  7 0F -
```