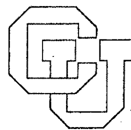
The Architecture of a Multi Associative Processor*

Gary J. Nutt

CU-CS-070-75 June 1975



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

*This report supercedes report number CU-CS-051-74, CU-CS-10277, and earlier versions of CU-CS-070-75 (coauthored by Roger D. Arnold).
Foundation under grant number MCS74-08328 A01.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

TABLE OF CONTENTS

I. Overview of the Architecture
  A. MAP Components
  B. A Typical Instruction Execution Trace
  C. Observations on the Overall Architecture

II. Control Unit Organization

III. Processing Element Organization

IV. Distribution Switch Organization

V. Control Unit Processor Interface

VI. Main Memory Subsystem

Appendix A: Related Papers and Reports

Appendix B: A Sample Instruction Set

I. Overview of the Architecture

This techical report is intended to serve as the most accurate formal documentation of the Multi Associative Processor as of the last revision date. It is a working reference manual, and as such, its contents change from time to time.

The Multi Associative Processor (MAP) computer system is a hypothetical parallel processor ensemble 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 up 1024* processing elements (PEs), that may be dynamically allocated to the CUs. Each PE has a local 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. Specific facilities for implementing a suitable operating system directly on MAP are provided in the machine architecture. The PEs are identical and are allocated from a common pool to the CUs, the number being allocated depending on the requirements of the process currently executing on the CU. MAP is a special purpose machine in that it is not intended to support I/O-bound programs. The architecture would be more effective on programs that have a relatively large amount of processing with respect to I/O. It is assumed that an LSI technology, such as that employed in bit slice microprocessor components, is used to implement the architecture.

MAP has been called an associative processor 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 some 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

*These numbers are arbitrary; no studies have been performed to support the given values.

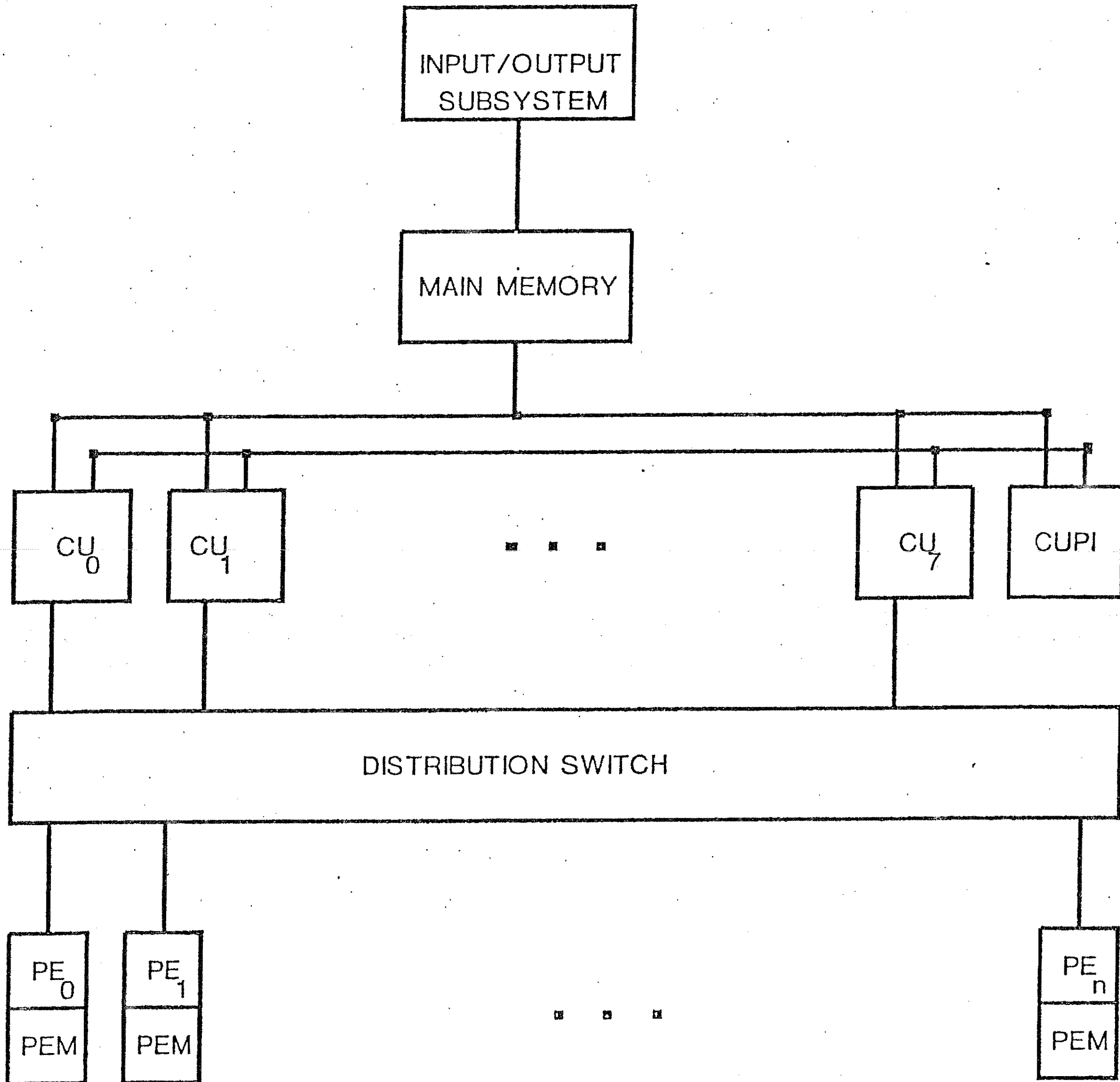
later sections. To provide a global view of the machine's operation, the remainder of this section includes a trace of a typical instruction execution and then a brief discussion of the assets and liabilities of the architecture.

Technical Report No. CU-CS-102-77 (January, 1977) represented a departure from several ideas discussed in earlier versions of this report. Those ideas have now been incorporated into this report, and the reader may ignore CU-CS-102-77.

A. MAP Components

A conceptual block diagram of the MAP computer system is shown in Figure 1. This report discusses the design of each component shown in the figure with the exception of the Input/Output subsystem. The I/O subsystem is assumed to be traditional in its design, and thus similar to one found in any large scale conventional computer system. The subsystem is made up of peripheral devices, controllers, and channels to support all input/output operations. It is assumed that the I/O subsystem can be directed to read/write information to/from the Main Memory by a single command which specifies a device number, a Main Memory buffer address, and a buffer length. The I/O subsystem performs the block transfer directly to the Main Memory buffer and then returns an interrupt indicating the I/O completion status.

The next component shown in Figure 1 is the Main Memory (MM). The function of the Main Memory differs only slightly from the corresponding component in a conventional machine. The MM serves as a storage for each program that is being executed on a Control Unit. In a SIMD architecture, data are ordinarily stored in the local Processing Element Memories; for the MAP system, those data are first read into MM and then loaded into the Processing Element Memories; i.e., MM is used as an I/O buffer for data. In some SIMD computations, it is convenient to use a single datum as an operand for all data streams, e.g., multiplying a vector by a scalar. The MAP architecture allows such a datum to be stored in the MM (as opposed to storing copies of the same datum in each Processing Element Memory). These uses of MM create a situation in which multiple processors (i.e., Control Units and data channels) may simultaneously attempt to access MM.



MAP ORGANIZATION

FIGURE 1

The MM design takes these conflicts into account in its design, discussed in Section VI.

A control unit (CU), is used to control the instruction stream execution for a given program, i.e., a CU implements a process (in terms of flow of control). In order to manipulate the instruction stream, the CU is capable of fetching instructions from MM, decoding those instructions, and causing them to be executed. A certain amount of computational facility is incorporated in the CU so that it can manipulate index registers, form main memory addresses using indirection and preindexing, maintain a program counter, communicate with (processes on) other CUs, and broadcast data and instructions to a set of processing elements currently allocated to the given CU. In terms of control flow of a MAP process, a CU corresponds to a processor in other computer architectures. The differences between MAP processes and processes on conventional (SISD or MIMD) machines lie in the handling of data computations. Each MAP process performs multiple, concurrent data computations. Control unit organization is discussed in the next section.

The control unit processor interface (CUPI), is a unit to execute certain instructions in the MAP repertoire which apply to more than one CU. The best example of such an instruction is a process synchronization instruction. All instructions executed by CUPI are concerned with the flow of control in a process, thus the design of CUPI is independent of the SIMD nature of MAP; the design is equally applicable to other multiple processor architectures. Section V discusses CUPI organization and operations.

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 PEs 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, PEs, perform arithmetic and logical operations on each data stream, stored in the respective PE memories. Each PE, then, roughly corresponds to an arithmetic/logical unit within a conventional processor. One level of concurrent operation is handled by applying the single instruction stream to the multiple processing elements. (The other level of concurrency is among the set of CUs). Processing

elements are described in Section III.

The idea of concurrency at the data computation level and at the process control level was proposed in the ILLIAC IV design, although it was not actually implemented. Some features that distinguish MAP from many other parallel processors can be mentioned:

- There are no direct data paths between individual PEs. All data communication is performed via common buses running to all PEs.
- There are no logical characteristics of individual PEs which distinguish one from another.
- The CUPI implements data abstraction as it applies to process state description and manipulation.
- Control units and processing elements are microprogrammed (bit slice microprocessors).
- 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.

Each of these points will be elaborated in the respective sections of the report.

B. A Typical Instruction Execution Trace

The state of a processing element can be described as deallocated, allocated and inactive, or allocated and active. A deallocated PE is not assigned to an active process on a CU, and hence it does not perform a (useful) computation. Inactive PEs are allocated, but are temporarily suppressed from executing instructions broadcasted from the owning CU. Active PEs execute instructions broadcasted by the owning CU.

For the following discussion, assume that a given control unit, CU, has been allocated a subset of n_i PEs, and that all of the PEs are active. The following list of hypothetical instructions are to be executed one after the other.

LOAD	3,*20,2	Load register 3 of each PE from the effective PEM address computed using single level indirect addressing (*), preindexed 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 register 3 of each PE from the effective main memory address using single level indirect addressing, preindexed 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 contains 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 CU recognizes the LOAD instruction as being executable by the PEs; therefore, it broadcasts the instruction to the set of active PEs (over the distribution switch). The CU may idle for the amount of time required for the instruction execution or continue concurrently with PE operation while the PEs compute their respective effective addresses and load their respective registers. The CU is then ready to process the next instruction.

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. Each PE tests its respective register 3 for a positive result; if the result is nonpositive, 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 (interrupted 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,

*Multiprogrammed CUs may or may not leave PEs idle.

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. In order to take advantage of the physical connections, the problem topology must fit the physical topology of the machine with fixed interconnection.

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 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 switching matrix (analogous to the existing distribution switch) 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 to each job.

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 one or more SIMD programs. 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.

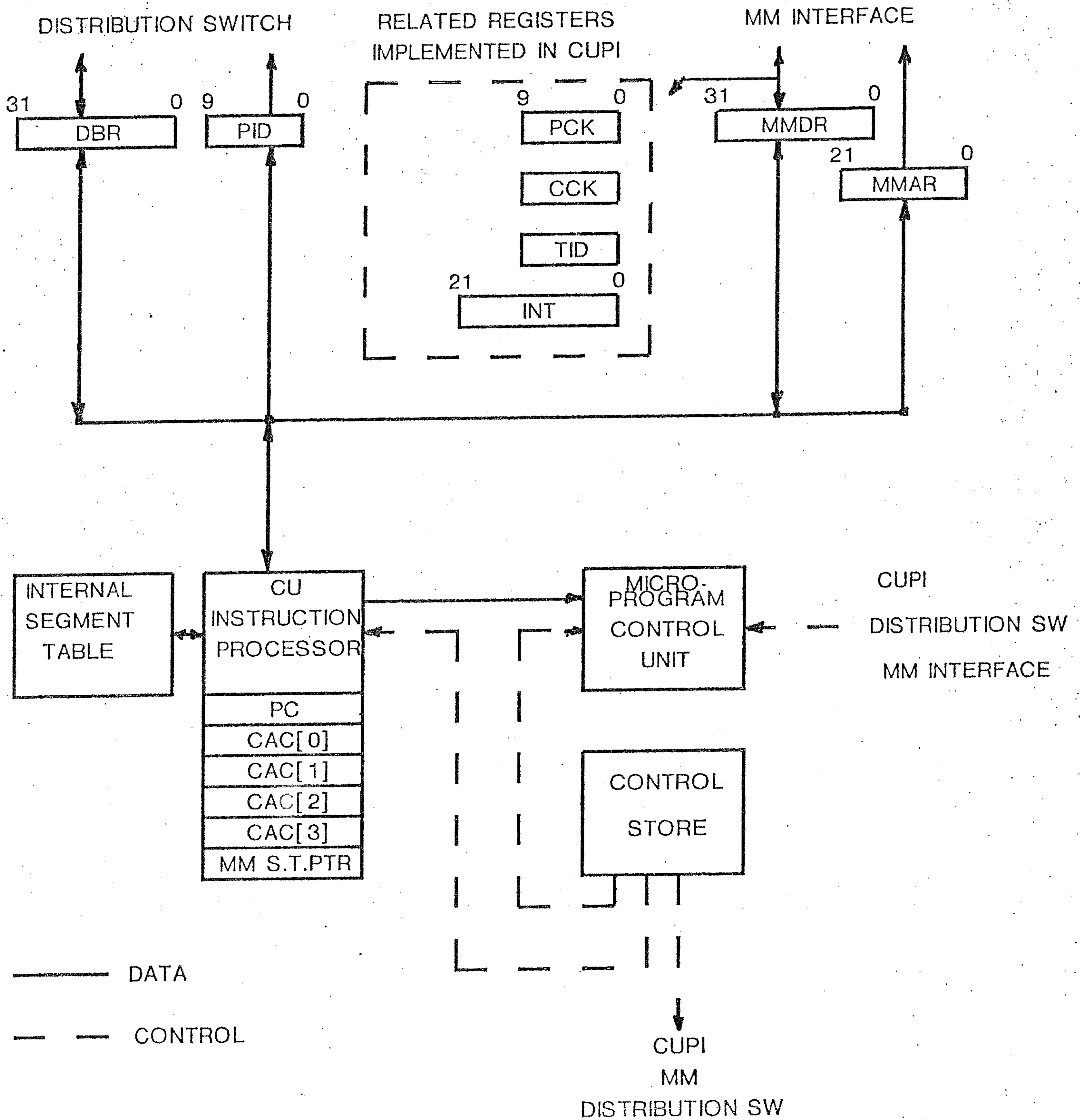
Figure 2 is a diagram of a control unit as it might be implemented with bit slice microprocessor components. In this report the high level design of the bit slice CU (and PE) are discussed. The detailed designs, including microprograms, are fully described in the M.S. thesis by Bruce Sanders (Department of Computer Science, University of Colorado, 1978). The interface to CUPI and MM consists of the two registers MMAR, MMDR, and control lines. MMAR is a 22 bit MM address register, and MMDR is a 32 bit data/ instruction register.* CUPI is referenced as a set of memory locations in the MM space by the control unit; i.e., a CU exchanges information with CUPI via a conceptual mailbox located at a predefined MM address.

The MMDR is a single register in each CU, although it might be implemented as a more elaborate lookahead buffer.** The MM design (see Section VI) and existing technologies employed in memory chips and bipolar microprocessors tend to match the speeds of a CU and memory; thus the MMDR can be simplified to a single register while still maintaining a high memory bandwidth.

---

*An arbitrary word size of 32 bits has been chosen for the architecture although no studies have been done to support this choice.

**In his M.S. thesis Heinrich Siegmann has considered the design and applicability of a lookahead mechanism for the MAP system (Department of Computer Science, University of Colorado, 1976). He argues that the nature of main memory references suits the lookahead technique in a SIMD machine.



CONTROL UNIT ORGANIZATION

FIGURE 2

Main memory addresses are absolute addresses although each process executing on a CU operates in a multiple segment virtual memory environment. The rationale for segmentation is based not so much on traditional reasons suchas the expansion of the process name space, but rather on the ability to provide precise memory protection and a flexible relocation mechanism. The protection is desirable for the obvious reasons of security and correctness, while the need for relocation is related to the goal of reducing memory conflicts through MM design and MM allocation. The tentative design for the segmentation mechanism uses data structures addressed by the MM Segment Table Pointer register, and the internal segment table. The segmentation algorithm is implemented in microcode rather than in hardware. The name space of a process is allowed to include as many distinct segments as any operating system for MAP is willing to support. Descriptors for these segments are maintained in the MM in a table addressed by the MM Segment Table Pointer. However, a fixed number of descriptors may be "paged" into the Internal Segment Table whenever they are being referenced by the address formation firmware (cf. the GE645 address formation hardware). Notice that the Internal Segment Table paging algorithm is also determined by microcode, and may be redefined for each MAP configuration. In general, the entire address formation algorithm is implemented in microcode and can be easily altered. For example, a microprogram to form addresses (given the segment number, offset within the segment, and type of memory access) might be specified as:

```
procedure FormAddress (Segment Number, Offset, Access Type);
begin
    if Segment Number not in Internal Segment Table
                then Load (Segment Number, MM.S.T.PTR);
    if 0 < Offset ≤ Limit [Segment Number]
                then Bound Error;
    if - Access Check (Access Type, Capability [Segment Number]
                then Protection Violation;
    MMAR: = Base [Segment Number] + Offset.
end;
```

The implementation of the Internal Segment Table is vague at this

point in the development. Two approaches to be considered employ either associative memory or random access memory for the Internal Segment Table. In the CAM case, the internal memory is small, with the bulk of the segment table residing in MM. In the RAM case, the internal memory may contain the entire segment table for a process, or a subset. If a subset is loaded, then a linear search of the internal memory is required to retrieve the segment descriptor word. If the entire table is loaded into an internal RAM, then the RAM must be reloaded when a process is multiplexed onto the CU. However, the RAM implementation is much less expensive than a CAM implementation. Notice that the address formation algorithm is microprogrammed, thus it is flexible but potentially slow. One aspect of the flexibility is that the mapping mechanism can be bypassed as a function of dynamic and/or static conditions that exist in the CU. For example, "privileged instructions" (such as CUPI-executable instructions) may reference absolute MM addresses.

A control unit may communicate with CUPI via MMDR, MMAR, and control lines. In order for a (process on a) CU to cause an instruction execution on CUPI, the CU microprogram performs the following tasks:

- Determine that the instruction is executed by CUPI.
- Compute the effective address.
- If the address is to a scalar operand, then fetch the operand and prepare to pass it to CUPI, else prepare to pass the operand address to CUPI.
- Format the instruction and operand (address).
- Store the command in a predetermined MM address.
- Wait for an acknowledge from CUPI. (The acknowledge either indicates that the instruction was completed, or that it was accepted and will be completed later.)

Inasmuch as the CUPI implements interprocess communication and virtual processor (control unit) implementation, certain registers associated with a CU process are maintained exclusively by CUPI. PCK, CCK, TID, and INT are such registers; their use will be explained when CUPI is discussed (see Section V).

The distribution switch interface is composed of two registers and some control lines. The data broadcast register, DBR, is used to pass information to/from the distribution switch, while the PID register identifies the process for which the information interchange is taking place. It is necessary to put the PID content into the distribution switch in order to determine the subset of active PEs allocated to the process being executed on the given CU. The control lines are used to request the use of the distribution switch and to acknowledge such a request.

The three remaining components of Figure 2 are the instruction processor, a microprogram control unit, and a control store. It is this portion of the CU that is implemented as a bipolar bit slice microprocessor chip set such as the AMD 2900 series or the Intel 3000 series. These components appear to be ideal as building blocks for the MAP CU since they are expandable to arbitrary word size, the cycle time is relatively fast, the components are modular, and the resulting processor is microprogrammable. (The low cost of the components has also allowed the MAP design to be changed significantly from early versions, particularly in the area of the distribution switch.) The overall operation of the 3 components is well documented in manufacturers' pamphlets, and in the open literature; thus the discussion here will be directed at the usage only in the MAP context.

All microprogram execution takes place on the Instruction Processor, (a bit slice ALU array). This unit executes the address formation algorithm discussed earlier, manages the program counter, fetches and decodes instruction words, executes branching instructions, sets up CUPI requests, etc. The microprogram control unit sequences the microprocess through the appropriate microprogram in control store. After the instruction processor fetches an instruction word into its internal registers, it can proceed with decoding. The 3 most significant bits of the MAP op code determine the instruction type. Type 0 instructions are those that affect the operation of another CU, and are passed to CUPI as described above. Type 1-3 instructions are to be executed (primarily) by the CU itself, and are processed by the Instruction Processor itself. Type 4-7 instructions are executed by the PE subset. A sample instruction set is included in Appendix B, and further description of the architecture

will use that set as an example to explain system operation.

After the Instruction Processor recognizes the instruction type, it will route the instruction to the appropriate execution unit. The microprogram must be cognizant of the timing of all instructions; either of two methods can be used to accomplish synchronization among the CU, CUPI, and PE subset during instruction execution. 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 Instruction Processor then performs the CU portion of the instruction, synchronizes with the PE subset by a bus signal, and performs the information exchange. It would be possible to design an acknowledge line to indicate the completion of PE instruction execution, but due to its complexity the synchronous approach is used in the current design. Bit slice microprocessor 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, and the MM Segment Table Pointer are implemented as internal microprocessor registers. 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 Appendix B. 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 ALU microprocessor chips that implement the CU instruction processor require 6 internal registers; the remaining registers (5 in the case of Intel 3001, 10 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.

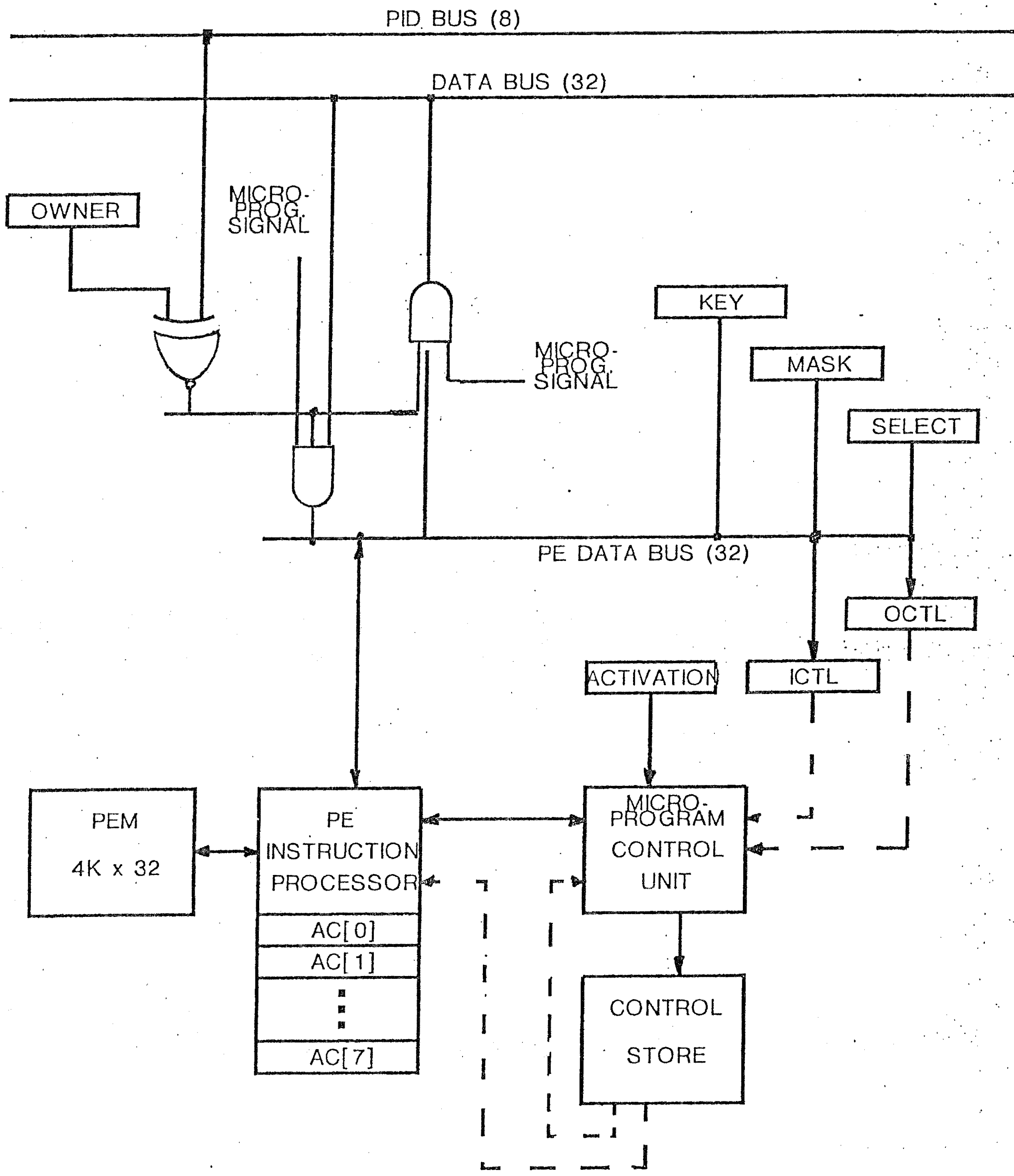
III. Processing Element Organization

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 itself 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 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 nor does it communicate with CUPI. 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 PID 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 PID and OWNER tags match. In order for a data transfer in either direction, a signal from the microprogram control store must be available; thus, a PE accepts data from the Distribution Switch only when the microprogram is ready to synchronize.

Although the PE does not fetch instructions from a memory, it still has a fetch-execute cycle. Whenever the PE is in a fetch state, it waits for data (i.e. an instruction word) from the Distribution Switch. The Distribution Switch interface gates data onto the internal PE data bus where it is available to the PE Instruction Processor. Preliminary decoding in the CU is used to determine the execution unit for a given instruction. Final decoding and subsequent execution all take place in the PE Instruction Processor. This approach is counter to the original



PROCESSING ELEMENT ORGANIZATIONff

FIGURE 3

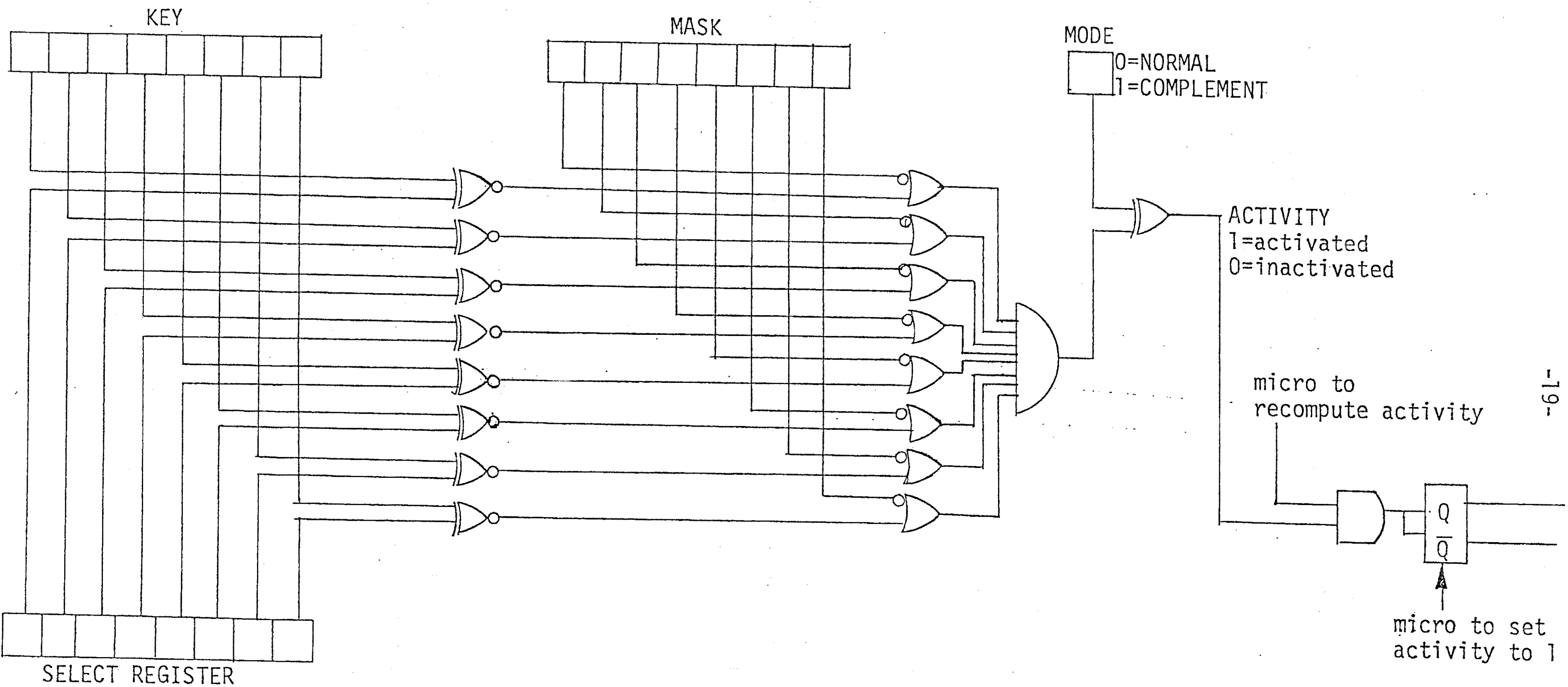
philosophy of SIMD machines, viz. that hardware costs could be minimized by centralizing all traditional control unit functions. However, hardware costs are no longer as important, and so the other important property of SIMD machines remains a justification for MAP (i.e. MM conflicts are reduced over multiple processor designs).

The Instruction Processor 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 Instruction Processor are straightforward, 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 PE is always physically active whether or not computations are to be logically carried out in the PE. The Activity register (flag) is an external register that is set and reset by the microprogram control unit to accomplish this task. 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 internal registers are used by the microprograms.

The SELECT register shown in Figure 3 is the mechanism used to save the results of up to eight conditions in the PE. Any MAP instruction set must include a set of <u>associative instructions</u> similar to the type 4 instructions shown in Appendix B. Each associative instruction includes an 8 bit Key field and an 8 bit Mask field, see Figure 4. The "SELECT" instruction 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 Appendix B 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 register, depending on conditions that exist in the PE at the time of their execution. For



CONCEPTUAL PE SELECTION MECHANISM

Figure 4

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.

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 "003_8" and a mask of "007_8" 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_8" and a mask of "003_8" under complement selection. Although keys and masks apply conceptually to every instruction, it is necessary for the CU to broadcast them only when they change.

The associative instructions are implemented in firmware. The Instruction Processor 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.

Two major difficulties with implementing processing elements have to do with synchronization between the CU and its PEs, and with synchronization among the set of PEs allocated to a CU. Synchronization between the CU and its PEs was briefly described in section II. The primary difficulty in implementation arises with the stream instructions (LSTR, SSTR, and XSTR in Appendix B). The problem to be overcome is that the CU must synchronize with the distribution switch, the

main memory, and all PEs in order to cause an information transfer to take place. The ICTL and OCTL registers of Figure 3 are used to implement this form of synchronization; an explanation of the solution is presented in Section IV. As mentioned in Section II, the CU must also synchronize with its PEs at the beginning of each fetch-execute cycle. (The approach taken here was to have the CU microprogram time the execution of the PE microprograms.)

PE-PE synchronization is required for inter-PE comparison and selection instructions, e.g., SETMAX sets a bit in the SELECT register of a PE which contains the maximum value in a corresponding register over all active PEs. Each PE transmits the appropriate register content to the CU simultaneously, resulting in a logical OR of all such registers on the data bus in the distribution switch. The CU retrieves the logical sum and recognizes that a PE with maximum value must have had a bit set in the position of the most significant bit of the sum. All PEs that do not satisfy the above condition are temporarily deactivated and the process is repeated until a set of PEs with the maximum value is determined. One PE is chosen from the set as the PE with maximum value. Such an approach relies on multiple distribution switch transactions with corresponding synchronizations. However, it precludes the requirement of a separate switching network especially designed for multiple PE comparisons.

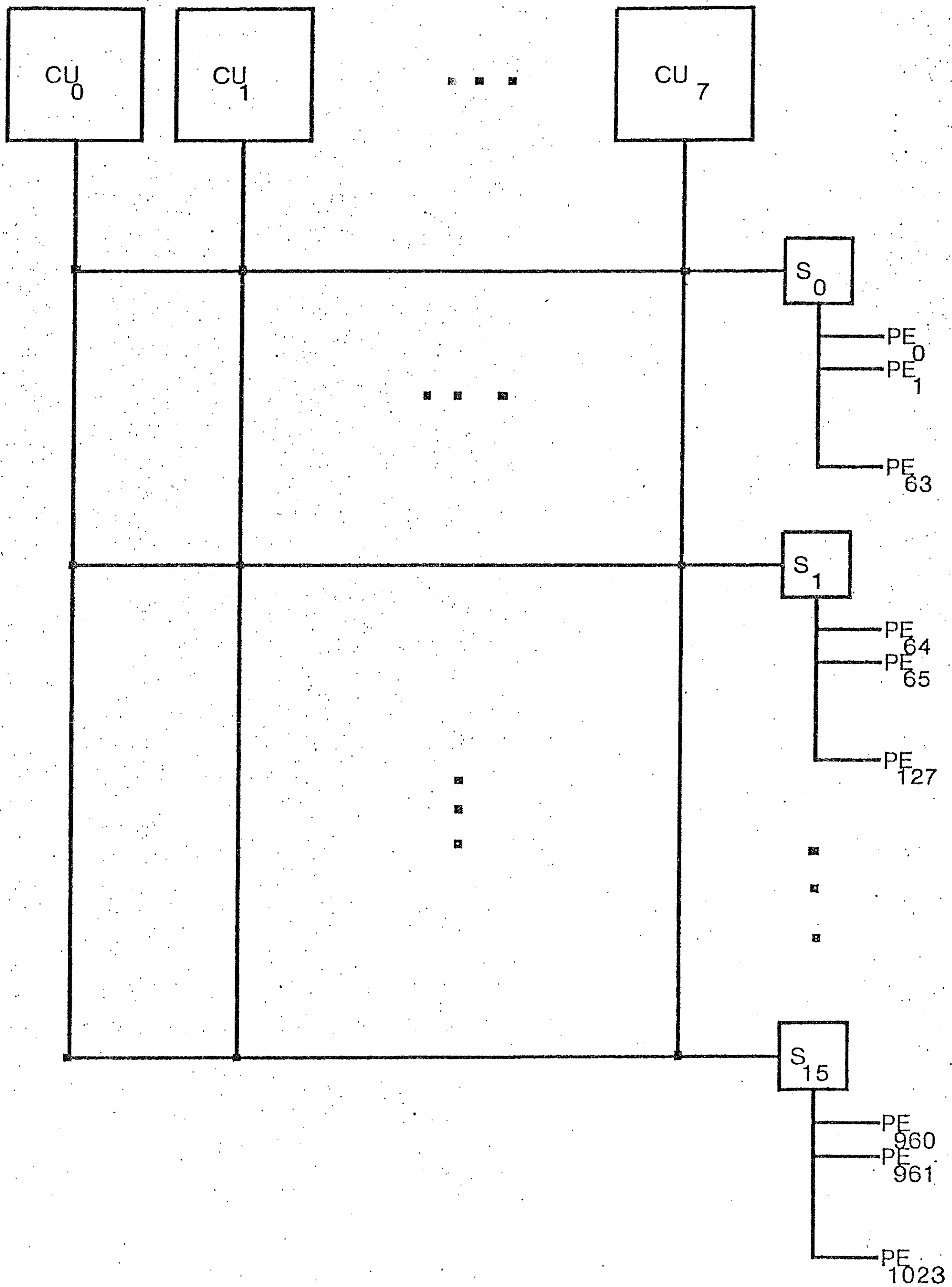
IV. Distribution Switch Organization

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 5, 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 <u>bus sector</u>.

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 initially loaded with a value representing a delay time. For each



DISTRIBUTION SWITCH

FIGURE 5

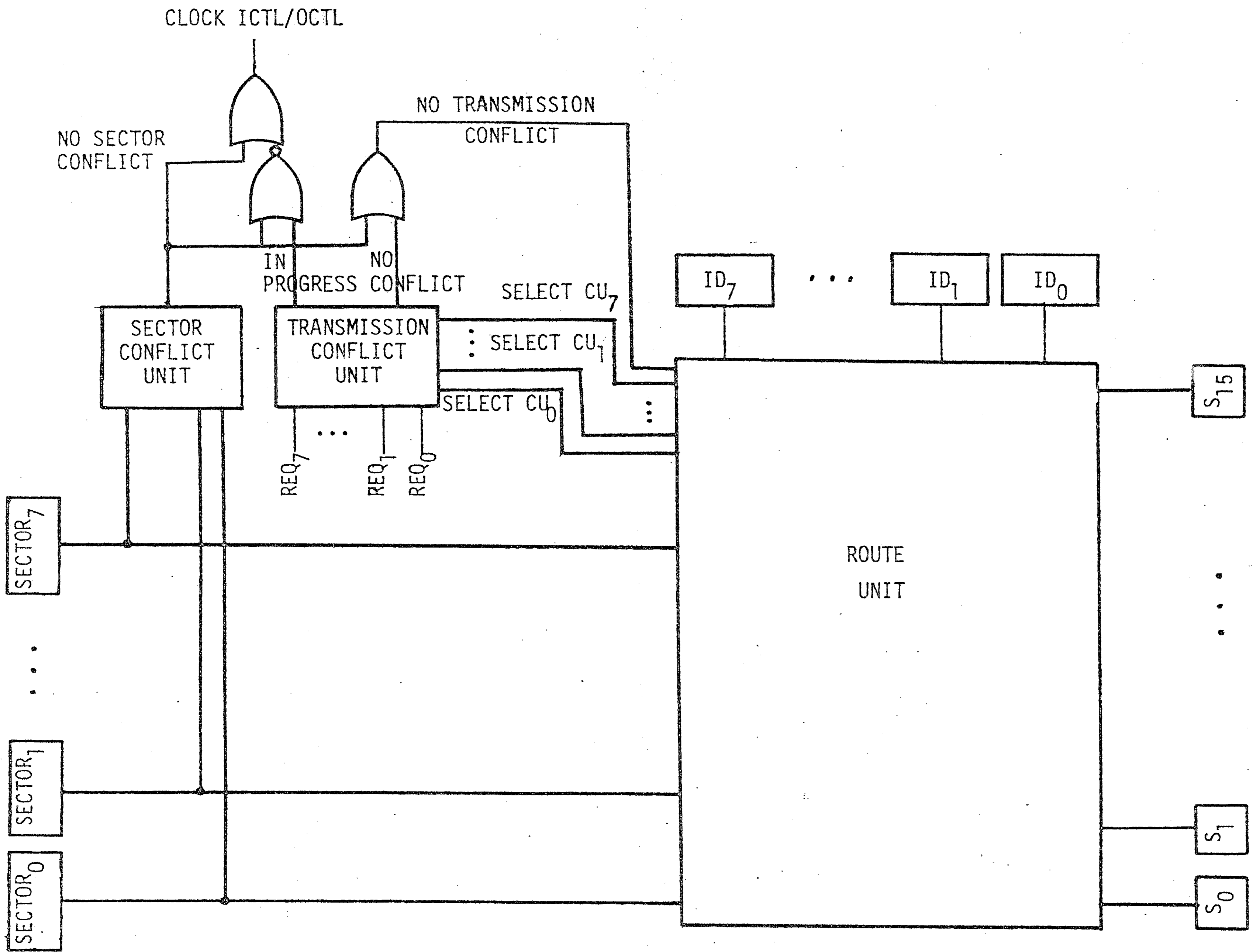
cycle to which the CU controlling the operation is not blocked from the data bus, the register is decremented by one. When the register reaches zero, a signal is generated which, in the case of the ICTL register, causes a word to be gated from the PE data bus to the PE Instruction Processor. For the 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 different 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 <u>Sector Mask</u> 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 6. 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



BUS SECTOR ALLOCATOR
FIGURE 6

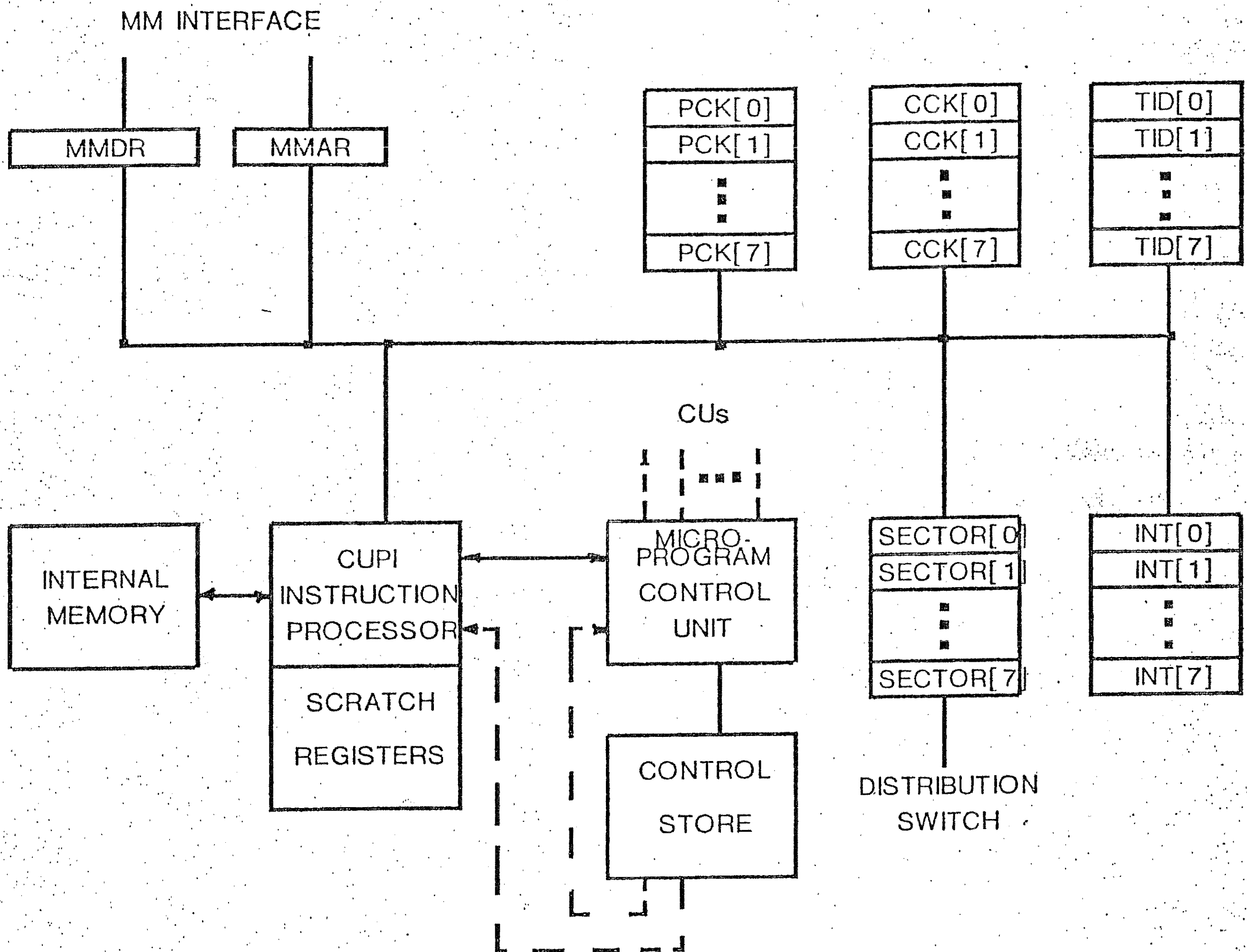
PEs are enabled. In the event of transmission conflict, the Transmission Conflict Unit arbitrates to select one CU for transmission; the Route Unit then uses the appropriate Sector Mask to broadcast the ID and the CU's DBR to the appropriate sectors.

V. Control Unit Processor Interface

The function of the CUPI is to execute instructions which effect more than one process. These instructions are primarily intended to provide a mechanism that is useful in managing processes. In particular, the CUPI-executed instructions establish means for process communication, process creation, process destruction, and processor (i.e.,CU) multiplexing. CUPI is a mechanism which is controlled by software executing on CUs (and microprograms executing within the CUPI itself) which is used to set policy. In the following discussion of CUPI, certain policy decisions are reflected by the choice of an instruction repertoire and internal CUPI data structures; note that these aspects of CUPI are strictly a function of the microprograms which control the CUPI hardware. Thus, a goal of the design is to isolate all policy decisions relating to the CUPI process management in CU software and CUPI microprograms. A more complete discussion of the CUPI design is found in the M.S. thesis of Kimbal Smith (Department of Computer Science, University of Colorado, 1978).

The physical organization of the CUPI is similar to that for a PE or a CU (see Figure 7). The architecture is that of a common bus, microprogrammed minicomputer system. The CUPI behaves in a manner analogous to a PE with respect to the instruction fetch-execute cycle, i.e., during the fetch phase of execution, the CUPI receives a single instruction from a CU without sequencing through a program in a memory module. Thus, CUPI is really a special purpose execution unit shared by the 8 control units.

The CUPI external interface to control units is through the MMDR and the control lines. Information is passed from a CU to CUPI by storing a formatted message in MM and then passing an interrupt to the CUPI microprogram control unit. This unit receives interrupts by having the fetch-execute cycle poll an interrupt request line. When an interrupt is pending, the microprogram services the interrupt by retrieving the formatted instruction from MM, executing the instruction, and achnowledging the execution by sending a similar interrupt to the microprogram control unit of the requesting CU. (Note that this sequence of activities causes the requesting CU



CU PROCESSOR INTERFACE

FIGURE 7

to treat type 0 instructions as indivisible machine instructions similar to type 1-7 instructions.) This CU-CUPI interface protocol does create a situation in which type 0 instructions are executed in an indeterminate amount of time. If the CUPI is saturated with CU requests, then the response to those requests will be influenced by the order in which the CUPI microprogram chooses to service simultaneous requests.

Figure 7 also shows an internal memory for the CUPI instruction processor as well as five banks of 8 registers each. Inasmuch as CUPI is the only processor capable of directly reading or writing those registers, it is entirely reasonable to implement them as memory locations in the internal memory. The purposes of the register banks with the exception of the SECTOR registers, is explained below. (The SECTOR registers are described in the previous section, and also appear in Figure 6.)

Process Intercommunication

PCK, CCK, TID, and INT registers are all used for process intercommunication. Recall that each process is also assigned a unique process identifier, which is stored in the PID register of the CU executing the process. (Thus one can refer to process number i as that process which has the integer i loaded into its PID.) If process i wishes to communicate with process j, then process i first identifies the target (receiver) process by requesting CUPI to load the integer j into TID_i. Process i may they communicate with j, via CUPI, provided that it has been allocated the right to participate in such a communication. PCK and CCK are used to specify those communication rights. Let $PCK_i[k]$ denote bit number k in the PCK register for process i; similarly define $CCK_i[k]$.

If

$$CCK_i[k] \wedge CCK_j[k] = 1 \qquad \text{for some } k \ (0 \leq k \leq q)$$

then processes i and j can <u>cooperatively communicate</u> with one another.

If

$$PCK_j[k] = 1 \Rightarrow PCK_i[k] = 1 \qquad \text{for all } k \ (0 \leq k \leq q)$$

then process i has <u>privilege</u> with respect to process j.

The basic idea for Type 0 instructions is that each instruction applies to more than one process; process i requests that CUPI execute the instruction with respect to the target process j (identified by TID_i). Almost all Type 0 instructions require that process i have appropriate right for that instruction, viz. cooperative or privileged rights. Thus, when the CU which is executing process i requests a Type 0 instruction execution with $c(TID_i) = j$ then CUPI first checks TID_i and then either PCK_i and PCK_j or CCK_i and CCK_j before actually executing the instruction.

The two most important Type 0 instructions for process intercommunication are SIGNAL and PREEMPT (see Appendix B). SIGNAL is used to cooperatively pass a message from process i to process j, and PREEMPT is used to pass a message in a privileged manner. Each instruction acts essentially like an interrupt to the receiving process, thus semaphore-like mechanisms can be used by a process to suppress incoming message processing. If ARM_j is cleared, then process j will temporarily ignore PREEMPT messages directed at it. If ARM_j is cleared or $ABLE_j$ is cleared (i.e., false), then process j will temporarily ignore incoming SIGNALs.

The 22 bit INT_j register is used during the SIGNAL instruction execution when the signal is directed at process j. If process i can cooperatively communicate with process j, ARM_j is set, and $ABLE_j$ is set, then when process i signals process j, process j interrupts normal processing and resumes execution at the address specified by INT_j.

The CUPI mechanism to implement SIGNAL and PREEMPT also handles other instructions to load, store, and test the relevant registers (see Appendix B). In most cases the PCK and CCK register contents are used to determine the right to execute these "privileged" instructions. In the case of register load instructions for PCK and CCK an additional restriction must be imposed to preserve the integrity of those registers. A process is not allowed to set bits in the target process' PCK or CCK if the corresponding bit is not set in the executing processes respective register.

Process Creation, Destruction, and Multiplexing

In the MAP context, a process is a logically executing program assigned to a process descriptor. A process is created when a descriptor is allocated, and the process is destroyed when the descriptor is deallocated. Within the CUPI design processes are created at systems generation time and destroyed when the machine is powered down. Processes known to CUPI are neither created nor destroyed -- they only change states.

When a system is generated, space is allocated to the CUPI (possibly in its own internal memory) to be used as internal process descripors. The descriptor specifies a PID content, PC, general registers, segment table pointer, SECTOR, PCK, CCK, TID, and INT for a process (as well as other information internal to CUPI such as the process state). Thus the internal process description corresponds to the machine state for a process. An internal process may be dead, static, or dynamic. If the status is dead, then no external process descriptor exists for the process, i.e., there is no corresponding external process. If the operating system wishes to create a process in its domain, then it requests process creation to CUPI while setting up its own external process descriptor to reflect resource allocation, accounting, etc. (but not machine state). When CUPI executes this creation command, it sets the PID of an internal process descriptor and changes the status to static. (Note that the newly created process is not presumed to be logically running; it is neither physically running nor dead). When the process is destroyed (halted) then the PID is cleared and the status is returned to dead. Thus creation and destruction within CUPI corresponds only to state transformations, while the corresponding functions in an operating system that utilizes CUPI cause processes to appear and disappear.

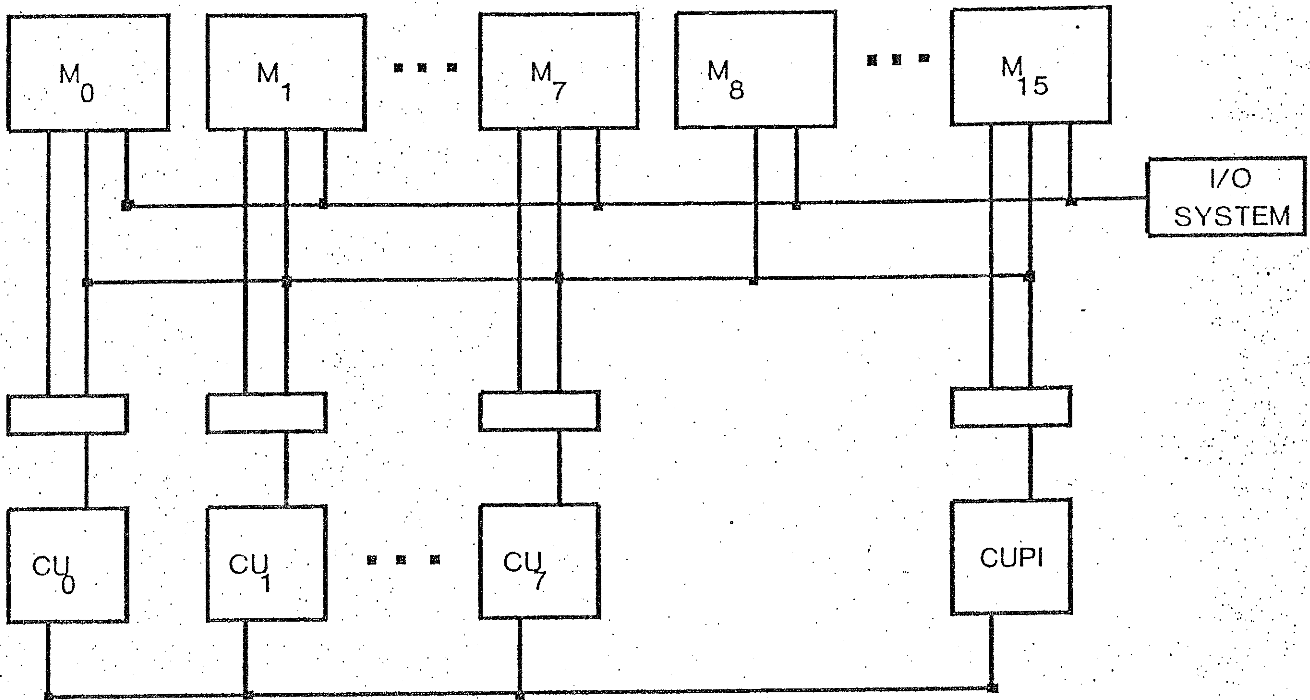
CU multiplexing is also handled by the CUPI (under the policy set by CU software). A process must be in the static state in order to be allocated a physical CU. Upon CU allocation, CUPI loads the physical registers from the internal process descriptor and sets the internal process state to dynamic. When the process is multiplexed off of the CU, its registers are saved in the internal process descriptor and the state is transformed to static. The mechanism used to introduce a scheduling policy relies on the schedule being implemented as a process that executes on a

CU. That process maintains a ready list of logically executing processes. Whenever an external process is logically running, then its internal counterpart is in the ready list. The scheduling process then selects processes from this ready list for physical execution(transformation from internal static to internal dynamic status). Process suspension and activation are handled by signaling the scheduling process. In order for this scheme to work, CUPI must be aware of the one special scheduling process at system creation time.

VI. Main Memory and the I/O System

A block diagram of the Main Memory and I/O subsystem appears in Figure 8. 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 sectors containing 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 sectors fit within the preferred module, but others may be loaded 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 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.



MAIN MEMORY SUBSYSTEM

FIGURE 8

Appendix A

List of Publications Related to MAP

1. Arnold, R. D., "Multi Associative Processor Systems Architecture", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-051-74, August, 1974.

2. Nutt, G. J., "Sample Programs for a Hypothetical Computer", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-058-74, October, 1974.

3. Nutt, G. J., "An Overview of a Multi Associative Processor System", Proceedings of the National ACM Conference, pp 101-104, November, 1974.

4. Nutt, G. J., "The Architecture of a Multi Associative Processor", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-070-75, June, 1975, (revised July, 1978).

5. Nutt, G. J., "Some Uses of Simulation in System Design", Proceedings of the Third Symposium on the Simulation of Computer Systems, pp 221-228, August, 1975.

6. Nutt, G. J., "Computer System Monitors", IEEE Computer, Vol. 8, No. 11, pp 51-61, November, 1975.

7. Nutt, G. J., "Measuring User Programs For a SIMD Processor", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-089-76, April, 1976.

8. Nutt, G. J., "A Parallel Processor For Evaluation Studies", AFIPS Proceedings of the NCC, Vol. 45, pp 769-775, 1976.

9. Nutt, G. J., "Some Resource Allocation Policies in a Multi Associative Processor", Vol. 6, pp 211-225, 1976.

10. Nutt, G. J., "Notes on a MAP Microprocessor Implementation", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-102-77, March, 1977.

11. Nutt, G. J., "Microprocessor Implementation of a Parallel Processor", Proceedings of the Fourth Annual Computer Architecture Symposium, pp 147-152, March, 1977.

12. Nutt, G. J., "Integrating Architecture and Operating Systems", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-104-77, March, 1977.

13. Nutt, G. J., "Memory and Bus Analysis of an Array Processor", IEEE Transactions on Computers, Vol. C-26, No. 6, pp 514-521, June, 1977.

14. Nutt, G. J., "A Parallel Processor Operating System Comparison", IEEE Transactions on Software Engineering, Vol. SE-3, No. 6, pp 467-475, November, 1977.

15. Nutt, G. J., "An Example of Simulation as a Computer System Design Tool", to appear in IEEE Computer.

16. Nutt, G. J., "Some Measure of Performance in SIMD Machines" submitted for publication.

17. Nutt, G. J., K. S. Smith, and B. W. Sanders, "Evaluation During Machine Design: A Case Study", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-125-78, October, 1977.

18. Nutt, G. J., W. A. Schulz, and K. H. Williamson, "Generating Code for a Hypothetical Computer Using a Production Assembler", Software -- Practice and Experience, Short Communication, Vol. 7, No. 1, pp 147-148, January-February, 1977.

19. Osterweil, J. P. and G. J. Nutt, "Modeling Process-Resource Activity", to appear in International Journal of Computer Mathematics.

20. Sanders, B. W., "The Design and Simulation of a Bit Slice Implementation for MAP", M. S. thesis, University of Colorado, Department of Computer Science, 1978.

21. Siegmann, H., "Design and Simulation of a Main Memory - Control Units Interface for the Multi Associative Processor System", M. S. thesis, University of Colorado, Department of Computer Science, 1976.

22. Smith, K. S., "An Interface for Interprocess Communication and Control within the MAP Architecture", M. S. thesis, University of Colorado, Department of Computer Science, 1978.

Appendix B
A Sample Instruction Set

An assembler for all instructions other than the type 0 instructions has been written, and the resulting machine code can be interpreted on a Nova or a CDC 6400. The same instructions have also been implemented in Intel 3000 microprograms.

TYPE 0 INSTRUCTIONS
This entire set of instructions is executed on the CUPI. The list of instructions is incomplete since the full list includes several instructions that are not explained in this report. The full Type 0 set can be found in the M.S. thesis by Kimbal Smith, along with a complete discussion of the CUPI mechanism.
In order to abbreviate the instruction descriptions, a shorthand notation will be used, where the following "procedure" is used in the semantic descriptions.
It is assumed that the process whose c(PID)=i is executing the given instruction, and that c(TID)=j.

i.privilege.j means that $PCK_j[k]=1 \Rightarrow PCK_i[k]=1$
for all k $(0 <= k <= 9)$

HALT
if i.privilege.j *then*
begin
halt the process with c(PID)=j;
change the CUPI state of j to dead
end

PREEMPT <k>
if i.privilege.j *then*
if arm[j] *then*
begin
arm[j] := *false*;
MM[f(j)] := PC[j];
PC[j] := g(k);
CAC[0,j] := CAC[0,i]
end

SIGNAL
if $\exists k(CCK_i[k]=1$ & $CCK_j[k]=1)$ *then*
if arm[j] & able[j] *then*
begin
able[j] := *false*;
arm[j] := *false*;
MM[f(j)] := PC[j];
PC[j] := INT[j];
CAC[o,j] := CAC[0,i];
arm[j] := *true*
end

The remaining Type 0 instructions allow process i to manipulate the various registers of process j, to test values, and to manipulate values maintained by the CUPI.

TYPE 1 INSTRUCTION
These instructions are used to perform operations on the CU general purpose registers. Their general form is

<op-code> <destination>, <left-opd>, <right-opd>

The semantic descriptions shall refer to the destination as D and the left and right operands as R1 and R2, respectively, e.g.,

OP D,R1,R2

CLR - CU Load Register from Register
D := c(R1)

CRR - CU Right Rotate Register
D := right-rotate c(R1) by c(R2)

CAR - CU Integer Add Registers

D := c(R1) + c(R2)

CNR - CU logical AND Registers
D := c(r1) & c(R2)

CSR - CU Integer Subtract Registers
D := c(R1) - c(R2)

TYPE 2 INSTRUCTIONS

These instructions apply primarily to CU operations, but in some cases PE operations are also involved in the instruction execution. The set includes immediate operand instructions of the form

<op-code> <destination>, <immediate opd>

where the immediate operand is designated as I in the following. Global instructions apply to the CU and its PEs; these op codes are generally of the form

G<op-code> <effective address>, <effective address>

where each effective address may be a main memory address, M; a CU destination register, D; a PE destination register, d; a PE operand register, r; or a PE register such as ICTL, OCTL or OWNER.

CLI - CU Load Register with Immediate Operand
D := I

CRI - CU Register Right Circular Shift by Immediate
D := right-rotate c(D) by I

CAI - CU Add Immediate Operand to Register
D := c(D) + I

CNI - CU Register Logical AND with Immediate Operand
D := c(D) & I

CSI - CU Integer Subtract Immediate Operand from Register
D := c(D) - I

GL - Global Load PE registers
d := c(M)

GLIC - Global Load ICTL Registers
ICTL := c(M)

GLOC - Global Load OCTL Registers
OCTL := c(M)

GLOW - Global Load OWNER Registers
OWNER := c(M)

GS - Global Store PE Registers
M := c(r) ;union over all active PEs

GSIC - Global Store ICTL Registers
M := c(ICTL) ;union over all active PEs

GSOC - Global Store OCTL Registers
M := c(OCTL) ;union over all active PEs

GSOW - Global Store OWNER Registers
M := c(OWNER) ;union over all active PEs

LSTR - Load Stream from Memory
d{of active PEs} := c(MM[M+f(ICTL)])

SSTR - Store Stream into Memory
MM[M+f(OCTL)] := d{of active PEs}

GMPC - Global Move from a PE register to a CU Register
D := c(d) ;union over all active PEs

GMICR - Global Move from a PE ICTL to a CU Register
D := c(ICTL) ;union over all active PEs

GMOCR - Global Move from a PE OCTL to a CU Register
D := c(OCTL) ;union over all active PEs

GMCP - Global Move from a CU Register to a PE Register
d := c(D)

GMRIC - Global Move from a CU Register to a PE ICTL

ICTL := c(D)

GMROC - Global Move from a CU Register to a PE OCTL
OCTL := c(D)

TYPE 3 INSTRUCTIONS

This class of instructions is intended to be used for manipulating the CU general purpose registers. Full arithmetic capability has not been included, but it is easy to see that such capability could be included in the existing hardware design of the CU by utilizing appropiate microprograms. The general format for these instructions is

<op-code> <register>, <register> <memory address>

where the registers will be designated as D, R1 and R2, and the memory address will be referred to as M.

CL - CU Register Load
D := c(M)

CS - CU Register Store
M := c(D)

CRM - CU Register Right Rotate by Memory Content
D := right-rotate of c(R1) by c(M)

CAM - CU Register Added to Memory, Result in a Register
D := c(R1) + c(M)

CNM - CU Register Logically ANDed with Memory
D := c(R1) & c(M)

CSM - CU Register Subtract Memory
D := c(R1) - c(M)

XCHNG - Exchange Register and Memory Contents
exchange c(R1) and c(M)

B - Unconditional Branch
goto M

BOVF - Branch on Overflow Condition
if overflow flag is set *then goto* M

BCTZ - Branch if the Count of Active PEs is Zero
if no active PEs *then goto* M

BCTO - Branch if the Count of Active PEs is One
if exactly one active PE *then goto* M

BCTGO - Branch if the Count of Active PEs is Greater than One
if more than one active PE *then goto* M

BXEQ - Branch on Registers Equal
if c(R1) = c(R2) *then goto* M

BXGT - Branch if One Register Greater Than the Other
if c(R1) > c(R2) *then goto* M

BXGE - Branch if One Register Greater Than or Equal the Other
if c(R1) >= c(R2) *then goto* M

BXLT - Branch if One Register Less Than the Other
if c(R1) < c(R2) *then goto* M

BXLE - Branch if One Register Less Than or Equal the Other
if c(R1) <= c(R2) *then goto* M

BXCHNG - Branch on Zero after an Exchange
XCHNG;
if c(R1) = 0 *then goto* M

BXZR - Branch if Register is Zero
if c(R1) = 0 *then goto* M

BXNZ - Branch if Register is Nonzero
if c(R1) < > 0 *then goto* M

BXPL - Branch if Register is Plus
if c(R1) >= 0 *then goto* M

BXNG - Branch if Register is Negative
if c(R1) < 0 *then goto* M

TYPE 4 INSTRUCTIONS
These instructions are primarily associative instructions, executed by the PEs. The variable field of each mnemonic instruction can contain a flag to specify whether or not the instruction applies to all PEs allocated to the CU, or just to those currently allocated and active. Each instruction specifies a key (K) and a mask (MSK) to manipulate bits in the SELECT register -- see text.

SELECT - Select Activity based on the current K and MSK
  if K .equivalent. (MSK & SELECT) *then* activate

COMSEL - Complement Select
  SELECT;
  deactivate all active PEs and activate all inactive PEs

SET - Set the SELECT register
  SELECT := (c(SELECT) & ~MSK) | K

SETPL - Set Active if the Register is Plus
  if c(d) >= 0 *then* SET

SETNG - Set Active if the Register is Negative
  if c(d) < 0 *then* SET

SETZR - Set Active if the Register is Zero
  if c(d) = 0 *then* SET

SETMAX - Set active if this PE's Register is Maximum over all PEs
  if c(d[*]) is maximum *then* SET

SETMIN - Set active if this PE's Register is Minimum over all PEs
  if c(d[*]) is minimum *then* SET

SETEQ - Set active if Registers are Equal
  if c(r1) = c(r2) *then* SET

SETEQ - Set active if Registers are Not Equal
  if c(r1) <> c(r2) *then* SET

SETLT - Set active if one Register is Less Than the Other
  if c(r1) < c(r2) *then* SET

SETLE - Set active if one Register is Less Than or Equal the Other
  if c(r1) <= c(r2) *then* SET

SETGT - Set active if one Register is Greater Than the Other
  if c(r1) > c(r2) *then* SET

SETGE - Set active if one Register is Greater Than or Equal the Other
  if c(r1) >= c(r2) *then* SET

SETFST - Set the First PE Active
  Set the active PE with the lowest hardware address;
  then deactivate all others

CLRFST - Clear the First PE Active
  Clear bits in SELECT as specified by K and MSK

TYPE 5 INSTRUCTIONS
This class is used for general arithmetic and logical manipulations of data within the PE. In this section, d refers to a general PE register, m refers to an address in the PE memory, and r1 and r2 refer to operands in PE registers.

LR - Load PE Register
  d := c(m)

LICR - Load PE ICTL Register
  ICTL := c(m)

LOCR - Load PE OCTL Register
  OCTL := c(m)

LSLR - Load PE SELECT Register
  SELECT := c(m)

LRIC - Load Register from ICTL
  d := c(ICTL)

LROC - Load Register from OCTL
  d := c(OCTL)

LRSL - Load Register from SELECT
  d := c(SELECT)

LICOC - Load ICTL from OCTL
ICTL := c(OCTL)
LOCIC - Load OCTL from ICTL
OCTL := c(ICTL)
XSTR - Exchange Data Streams Among PEs
see text {multiple data exchange across PEs}
FIX - Floating Point to Integer Conversion
d := *entier*(c(r1))
NORM - Normalize Register
d := *normalize*(c(r1))
COMP - Two-s Complement of a Register
d := 2's complement of c(r1)
NOT - logical Complement
d := ~c(r1)
AR - Integer Add Registers
d := c(r1) + c(r2)
FAR - Floating Point Add Registers
d := c(r1) + c(r2)
SR - Integer Subtract Registers
d := c(r1) - c(r2)
FSR - Floating Point Subtract Registers
d := c(r1) - c(r2)
MIR - Multiply Integer Registers
d := c(r1) * c(r2)
FMR - Floating Point Multiply Registers
d := c(r1) * c(r2)
MODR - Integer Modulo Operation on Registers
d := c(r1) *mod* c(r2)
FDR - Floating Point Divide Registers
d := c(r1) / c(r2)
ORR - Logical OR Registers
d := c(r1) | c(r2)
ANDR - Logical AND Registers
d := c(r1) & c(r2)
EORR - Logical EOR Registers
d := c(r1) .eor. c(r2)
SC - Left Circular Shift on Registers
d := left-rotate c(r1) by c(r2)
SL - Signed Logical Shift on Registers
d := left-logical c(r1) by c(r2)
{left on positive c(r2), right otherwise}
SA - Arithmetic Shift on Registers
d := arithmetic-right c(r1) by c(r2)

TYPE 6 INSTRUCTIONS
Immediate operand instructions for use within the PEs. The immediate operand is referred to as i, and the other operand as d.
LI - Load with Immediate Operand
d := i
LICI - Load ICTL with Immediate Operand
ICTL := i
LOCI - Load OCTL with Immediate Operand
OCTL := i
AI - Add Immediate Operand to Register
d := c(d) + i
AICI - Add ICTL to Immediate Operand
ICTL := c(ICTL) + i
AOCI - Add OCTL to Immediate Operand

OCTL := c(OCTL) + i

SI - Subtract Immediate Operand from Register
d := c(d) - i

SICI - Subtract Immediate Operand from ICTL
ICTL := c(ICTL) - i

SOCI - Subtract Immediate Operand from OCTL
OCTL := c(OCTL) - i

SCI - Shift Circular Immediate
d := left-rotate c(d) by i

SLI - Shift Logical Immediate
d := left-logical c(d) by i

SAI - Shift Arithmetic Immediate
d := arithmetic-right c(d) by i

TYPE 7 INSTRUCTIONS

General PE register/memory manipulation instructions. PE registers are referred to as d and r; PE memory addresses as m.

L - Load Register
d := c(m)

LIC - Load ICTL Register
ICTL := c(m)

LOC - Load OCTL Register
OCTL := c(m)

S - Store Register
m := c(r)

SIC - Store ICTL Register
m := c(ICTL)

SOC - Store OCTL Register
m := c(OCTL)

AM - Add Memory to Register
d := c(d) + c(m)

AMIC - Add Memory to ICTL Register
ICTL := c(ICTL) + c(m)

AMOC - Add Memory to OCTL Register
OCTL := c(OCTL) + c(m)

SM - Subtract Memory from Register
d := c(d) - c(m)

SMIC - Subtract Memory from ICTL Register
ICTL := c(ICTL) - c(m)

SMOC - Subtract Memory from OCTL Register
OCTL := c(OCTL) - c(m)

FSM - Floating Point Subtract Memory from Register
d := c(d) - c(m)

MIM - Multiply Integer Memory by Register
d := c(d) * c(m)

FMM - Floating Point Multiply Memory by Register
d := c(d) * c(m)

MODM - Modulo Opeation by Memory
d := c(d) *mod* c(m)

FDM - Floating Point Divide by Memory
d := c(d) / c(m)

ORM - Logical OR by Memory
d := c(d) | c(m)

ANDM - Logical AND by Memory
d := c(d) & c(m)

EORM - Logical EOR by Memory
d := c(d) .eor. c(m)