A Collection of Papers on Cm*:

A Multi-Microprocessor Computer System

S.H. Fuller, A.K. Jones, D.P. Siewiorek, R.S. Swan,
A. Bechtolsheim, R.J. Chansler, I. Durham, P. Feiler,
K.W. Lai, J. Ousterhout and K. Schwans

February, 1977

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

This work was supported in part by the Advanced Research Projects Agency under contract number F44620-73-C-0074, which is monitored by the Air Force Office of Scientific Research, and in part by the National Science Foundation Grant GJ 32758X. The LSI-11's and related equipment were supplied by Digital Equipment Corporation.

Preface

This report is a collection of three papers that describe the design and implementation of the multi-microprocessor computer system (Cm*) currently under developement at Carnegie Mellon University. The three papers are:

Cm*: A Modular Multiprocessor

The architecture of a new multiprocessor that supports a large number of processors (on the order of 100) is described in this paper. The system enables very close cooperation between the large numbers of inexpensive processors. All processors share access to a single virtual memory address space. There are no arbitrary limits on the number of processors, amount of memory or communication bandwidth in the system. Considerable support is provided for low level operating system primitives and inter-process communication.

The Implementation of the Cm* Multi-Microprocessor

The implementation of the Cm* multiprocessor multiprocessor is presented. The lowest level of the structure, a Computer Module, is a processor-memory pair. Computer Modules are grouped to form a cluster; communication within the cluster is via a parallel bus controlled by a centralized address mapping processor. Clusters communicate via intercluster busses. A memory reference by a program may be routed, transparently, to any memory in the system. This paper discusses the hardware used to implement the communication mechanism. The use of special diagnostic hardware and performance models is also discussed.

Software Management of Cm*, a Distributed Multiprocessor

This paper describes the software system being developed for Cm*, a distributed multi-microprocessor. This software provides for flexible, yet controlled, sharing of code and data via a capability addressed virtual memory, creation and management of groups of processes known as task forces, and efficient interprocess communication.

Cm*: a Modular, Multi-Microprocessor

R. J. Swan
S. H. Fuller
D. P. Siewiorek

Carnegie-Mellon University
Pittsburgh, PA 15213

November 30, 1976

Abstract

This paper describes the architecture of a new large multiprocessor computer system being built at Carnegie-Mellon University. The system allows close cooperation between large numbers of inexpensive processors. All processors share access to a single virtual memory address space. There are no arbitrary limits on the number of processors, amount of memory or communication bandwidth in the system. Considerable support is provided for low level operating system primitives and inter-process communication.

CR Categories: 6.20, 4.30, 4.32

Keywords: Multiprocessor, microprocessor, computer architecture, virtual memory

This work was supported in part by the Advanced Research Projects Agency under contract F44620-73-C-0074, which is monitored by the Air Force Office of Scientific Research, and in part by the National Science Foundation Grant GJ 32758X. The LSI-11's and related equipment were supplied by Digital Equipment Corporation.

1. Introduction

Cm* is an experimental computer system designed to investigate the problems and potentials of modular, multi-microprocessors. The initial impetus for the Cm* project was provided by the continuing advances in semiconductor technology as exemplified by processors-on-a-chip and large memory arrays. In the near future processors of moderate capability, such as a PDP-11, and several thousand words of memory will be placed on a single integrated circuit chip. If large computer systems are to be built from such chips, what should be the structure of such a 'computer module'?

Initial versions of the Cm* architecture [Fuller, et al. 73] grew in part as an extension to the modular design of systems from register transfer modules, or RTMs [Bell et al, 72]. In addition there was substantial interest in the development of large multiprocessor systems such as Pluribus [Heart, et al. 73] and C.mmp[Wulf and Bell, 72]. Cm* is intended to be a testbed for exploring a number of research questions concerning multiprocessor systems, for example potential for deadlocks, structure of inter-processor control mechanisms, modularity, reliability, and techniques for decomposing algorithms into parallel cooperating processes.

The structure of Cm* is very briefly described in Section 2. Section 3 is a description of the address structure and discusses the support given for the operating system. The use of the addressing structure for inter-process communication and control operations is discussed in Section 4. A companion paper [Swan, et al. 77] discusses the various mechanisms used to implement the complex address mapping and routing structure of Cm*. Some results from the performance modelling of Cm* are also presented. A second companion paper [Jones, et al. 77] describes the structure of the basic operating system and support software.

2. The Structure of Cm*

There is a surprising diversity of ways to approach the interconnection of processors into a computing system [Anderson and Jensen, 76]. The processors could be interconnected with several serial I/O links to form a computer network; they could be interconnected in a tight synchronous fashion to build an array processor; or the processors could be organized to share primary memory. This last approach, a multiprocessor organization, was chosen for Cm* because it offers a closer degree of coupling, or communication, between the processors than would a multicomputer or network configuration. Multiprocessors also have a more general range of applicability than other multiple processor systems.

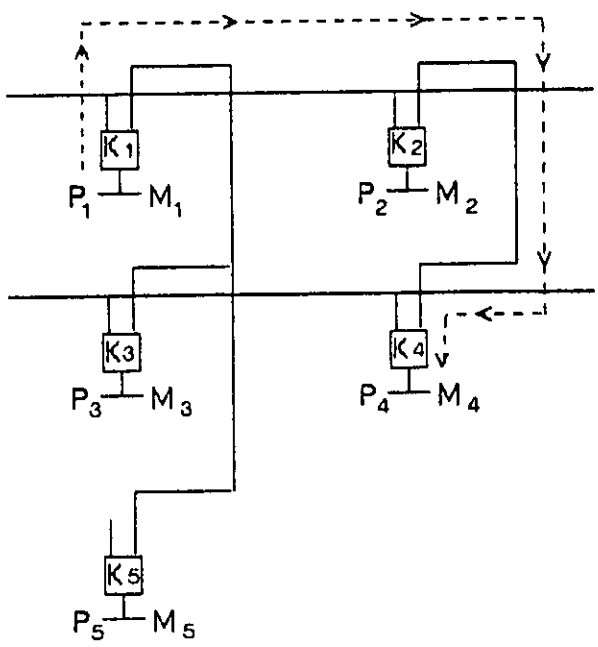


Figure 2.1 Canonical Computer Module Structure

During the development of the Cm* structure a wide variety of multiprocessor switch structures were considered [Swan, et al. 76B]. The basic structure selected is represented in Figure 2.1. The essential feature which distinguishes it from other multiprocessor structures is that shared memory is not separated from the processing elements, but rather a unit of memory and a processor are closely coupled in each module and a network of buses gives a processor access to nonlocal memory. This structure allows modular expansion of the number of processors and memory modules without a rapid increase in the interconnection costs. Memory can be shared even though there is no direct physical connection between the requesting processor and the required memory. For example, consider a request by processor, P1, to the memory, M4, in Figure 2.1. The address mapping element, K1, directs the reference from P1 onto the intermodule bus. The address is recognized by K2, which directs it onto a second inter-module bus. The reference is finally accepted by K4, which accesses the request memory location and passes back an acknowledgement or data to the requesting processor. The need for high inter-module communication rates will be minimized if a large fraction of each processor's references to primary memory 'hit' the section of memory local to the processor. (Preliminary experiments in the Fall of 1976 indicate that hit ratios of better than 90% can be expected provided that the code executed is normally held local to the processor.)

2.1 Deadlock with References to Nonlocal Memory

Almost all computer systems implement accesses from processor to primary memory with *Circuit Switching*, that is, a complete path is established from a processor to the memory being referenced. Circuit switching is not feasible for a structure like Cm* where local memory is also accessible as shared memory. Figure 2.1 shows the path used for P1 to access M4 via K2. Consider a concurrent attempt by P4 to access M1 via K2. With a circuit switch implementation, a situation could arise where P1 held its local memory bus and the bus connecting K2, while P4 also holds its own memory bus plus the bus connecting K4 to K2. Neither memory reference could complete without one processor first releasing the buses it holds. There are numerous situations where deadlock over bus allocation can occur. Resolving this deadlock requires, at the very least, a timeout and retry mechanism.

The alternative to circuit switching is *Packet Switching*. In a packet switched implementation, the address from the processor is latched at each level in the bus structure. Buses are not allocated for the full duration of a memory reference, but just for the time taken to pass a 'packet', containing an address and/or data, from one node on the bus to another. Therefore packet switching allows significantly better bus utilization and significantly reduced bus contention in Cm*-like structures. The use of packet switching eliminates the possibility of deadlock over bus allocation but introduces the possibility of deadlock over buffer allocation. [Fuller, et al. 73; Swan, et al. 76A] Buffers, or intermediate registers, are resources which can be provided very cheaply, relative to providing additional inter-Cm buses, with present technology.

2.2 The Actual Structure of Cm*

Design studies indicated that very little performance loss would result from combining several individual Computer Modules into a cluster and providing a shared address mapping and routing processor, *Kmap*, which allowed communication with other clusters. Because the cost of the Kmap is distributed across many processors it can be endowed with considerable flexibility and power at relatively little incremental cost. Because of its commanding position in the cluster, the Kmap can ensure mutual exclusion on access to shared data structures with very little overhead.

The full structure of Cm* is shown in Figure 2.2. Individual Computer Modules, or Cms, consist of a DEC LSI-11 processor, an Slocal and standard LSI-11 bus memory and devices. The processor is program compatible with PDP-11s; thus a large body of software is immediately available. The prime function of the Slocal, or local switch, is to direct references from the processor selectively either

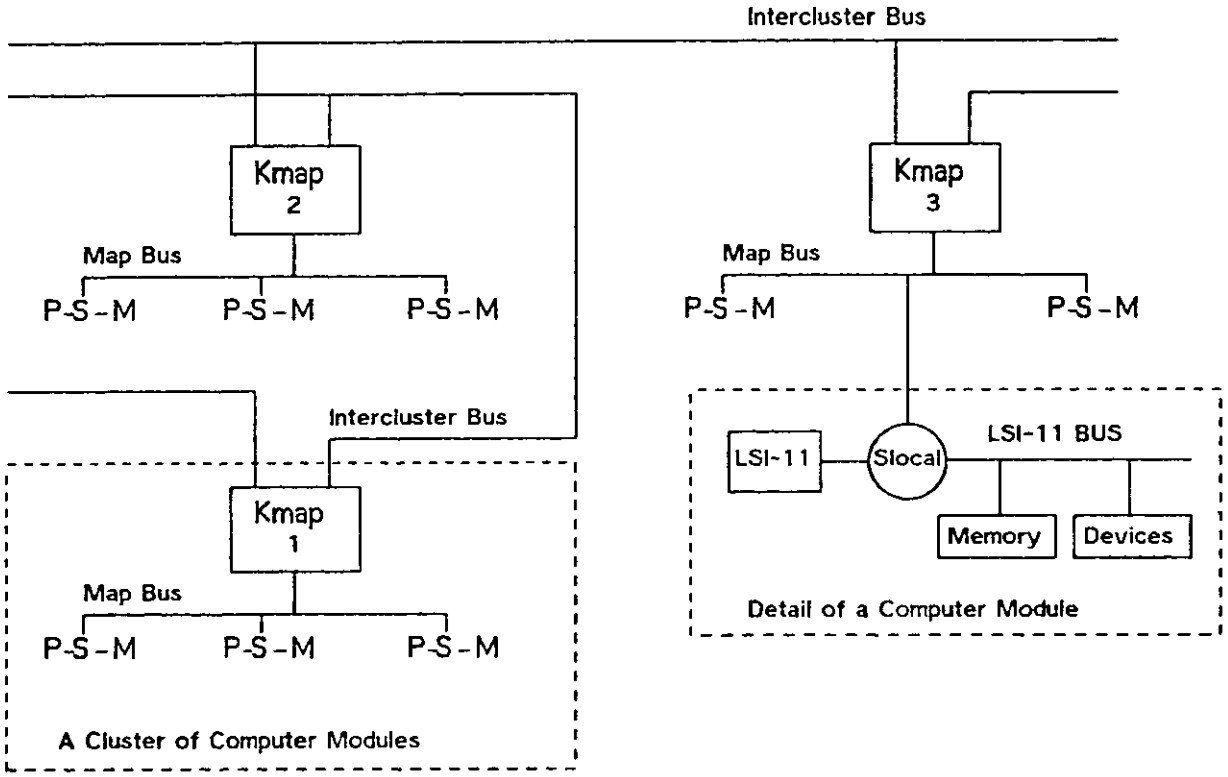


Figure 2.2 A Simple 3 Cluster Cm* System

to local memory or to the Map Bus, and to accept references from the Map Bus to the local memory.

Up to 14 Computer Modules and one Kmap form a cluster. The Kmap, or mapping processor, consists of three major components. The Kbus arbitrates and controls the Map bus. The Pmap is a horizontally microcoded 150 ns cycle time processor. The basic configuration has 1 K x 80 bits of writable control store and 5K x 16 bits of bipolar RAM for holding mapping tables etc. The third level of the Cm* structure is provided by the Intercluster buses which allow communication between clusters. The Linc provides the interface to two intercluster buses.

There are no arbitrary limits to the size of a Cm* system. Memories, processors and Kmaps can be incrementally added to suit needs. Any processor can access any memory location in the system. The routing of a processor's reference to a target memory is transparent to the program, thus the system can be reconfigured dynamically in response to hardware failures.

3. Architecture of the Address Translation Mechanisms

Many of the more conventional aspects of the architecture of the Cm* system are consequences of using LSI-11's for the central processing elements. The organization and encoding of the instructions, interrupt and trap sequencing, and the 64K byte processor address space of a Cm* system are all a result of the PDP-11 architecture as implemented on the LSI-11. By selection of the LSI-11, however, we do not want to imply that the PDP-11 architecture is ideally suited to multiprocessor systems. The ideal solution would have been for us to have designed our own processors. However, practical considerations of time, money, and existing support software lead us in early 1975 to recognize that by chosing the LSI-11 we could concentrate on those aspects of the Cm* architecture unique to multiprocessor systems. This section, and the following section on control structures, will discuss the Cm* architecture as we extended it beyond the standard PDP-11 architecture.

The addressing structure is one of the the most important aspects of any computer architecture, it is even

more significant when cooperation between multiple processors is to be achieved by sharing an address space. Denning [1970] lists four objectives for a memory mapping scheme:

(a) Program modularity: the ability to independently change and recompile program modules.

(b) Variable size data structures.

(c) Protection

(d) Data and program sharing: allowing independent programs to access the same physical memory addresses with different program names.

For Cm*, where we are using processors with only a 64K byte address space, we must add the following requirement:

(e) Expansion of a processor's address space.

Cm* has a 2^{28} byte segmented virtual address space. Segments are of variable size up to a maximum of 4K bytes. There is a capability-based protection scheme enforced by the Kmap. The addressing structure provides considerable support for operating system primitives such as context switching and interprocess message transmission.

3.1 The Path from Processor to Memory

The Slocal (see Figures 2.2 and 3.1) provides the first level of memory mapping. A reference to local memory is simply relocated, on 4K byte page boundaries, by the relocation table in the Slocal. As discussed above, it is assumed that most memory references will be made by processors to their local memory. Relocation of local memory references can be implemented with no performance overhead because the synchronous processor has sufficiently wide timing margins at the points where address relocation is performed. For segments which are not in a processor's local memory the relocation table has a status bit which causes the address to be latched, the processor forced off the LSI-11 bus, and a Service Request to be signalled to the Kmap. All transactions on the Map bus are controlled by the Map bus controller, or Kbus, which is a component of the Kmap. The address generated by the processor is transferred via the Map bus to the Pmap, the microprogrammed processor within the Kmap. If the reference is for memory within the cluster then the Pmap generates a physical address and sends it to the appropriate Slocal. If it is a write operation, data is passed directly from the source Slocal to the destination Slocal; the data does not have to be routed through the Kmap. The selected destination Slocal performs the requested memory reference and the processor in the destination Computer

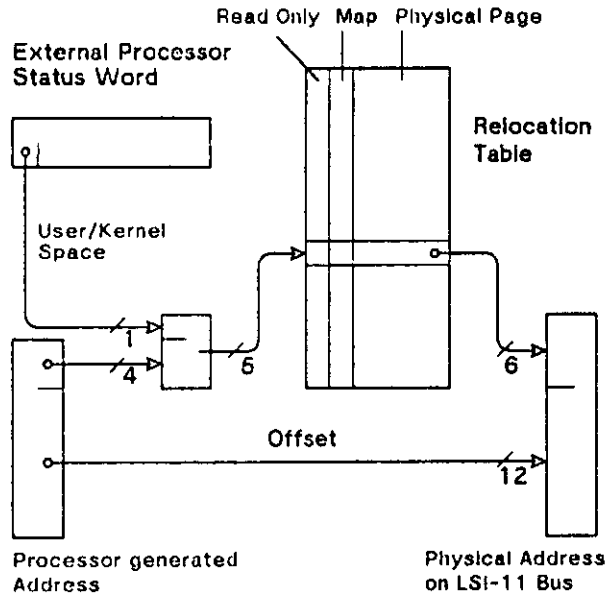


Figure 3.1
Addressing Mechanism for Local Memory References

Module is not involved. When the reference is complete the Kbus transfers the data read from the destination Slocal directly back to the requesting processor via the Map bus and its Slocal.

If the processor references a segment in another cluster then the Pmap will transmit a request to the desired cluster via the Linc and the Intercluster buses. (See Fig. 2.2.) If the destination cluster is not directly connected to the source cluster, that is, if it does not share a common Intercluster bus, then the message will be automatically routed via intermediate clusters. When the message reaches the destination cluster, the memory reference is performed similar to a request from a processor within the cluster. An acknowledgement, or Return, message (containing data in the case of a read) is always sent back to the source cluster and subsequently to the requesting processor.

3.2 The Addressing Environment of a Process

The virtual address space of Cm* is subdivided into up to 2^{16} *Segments*. Each segment is defined by a *Segment Descriptor*. The standard type of segment is similar to segments in other computer systems; it is simply a vector of memory locations. The segment descriptor specifies the physical base address of the segment and the length of the segment. Segments are variable in size from 2 bytes to 4 K bytes. However, other segment types may be more

than simple linear vectors of memory; references to segments may invoke special operations. Segments may have the properties of stacks, queues or other data structures. Some segments may not have any memory associated with them, and a reference to the segment would invoke a control operation. For each segment type, up to eight distinct operations can be defined. For normal segments the operations are Read and Write. Conceptually, segments are never addressed directly; they are always referenced indirectly via a *Capability*. A capability is a two-word item containing the name of a segment and a *Rights* field. Each bit in the rights field indicates whether the corresponding operation is permitted on the segment.

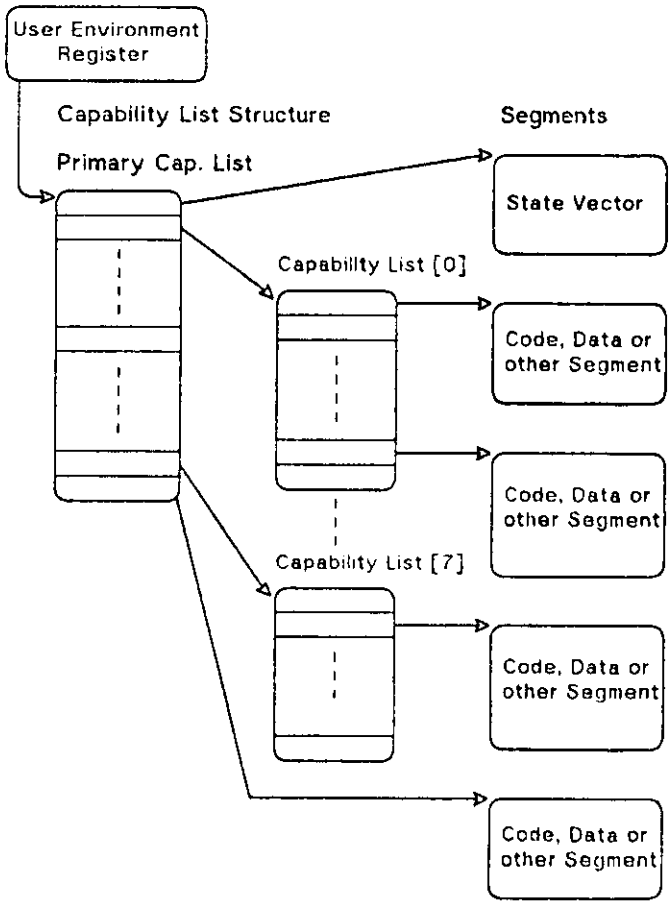


Figure 3.2 The Environment of a User Software Module

To provide efficient support for context swapping, message-sending etc., it is necessary for the Kmap microcode to understand some of the structure of an executable software module (variously called a process, activity, address space etc.). Each executable software module is represented by an *Environment*, Figure 3.2. An environment is a three-level structure composed of segments. The first level in the structure is a *Primary Capability List*, CL[0]. The first entry in CL[0] is a Capability for a *State Vector*, which holds the process state while it is not executing on a processor. Entries CL[0](1) to CL[0](7) in the Primary Capability list may contain Capabilities for *Secondary Capability Lists* referred to as CL[1] through CL[7] respectively. The remaining entries in the Primary Capability List and all the entries in the Secondary Capability Lists contain Capabilities for segments which can be made directly addressable by the process when it executes. These may be code, data or any other type of segment. The provision of up to eight Capability Lists facilitates the sharing of segments and sets of segments by cooperating processes. A software module can only access those segments for which it has capabilities and perform only those operations permitted by the capabilities.

3.3 Virtual Address Generation

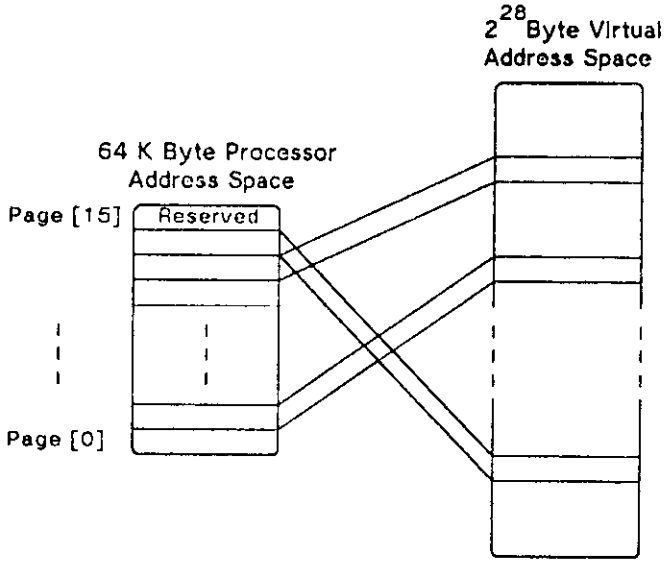


Figure 3.3
Windows from the Processor's Immediate Address Space to the Virtual Address Space

The processors in Cm*, LSI-11s, can directly generate only a 16 bit address. This 64 K byte address space is divided into 16 pages of 4 K bytes each. Each page provides a window into the system wide 2^{28} byte virtual address space, (see Figure 3.3) and can be independently bound to a different segment in the virtual address space. The top page in the processor's address space, page 15, is reserved for direct program interaction with the Kmap. This mechanism is analogous to the I/O page convention in standard PDP-11s. In page 15 there are 15 pseudo

registers, called *Window Registers*. These define the binding between page frames in the processor's immediate address space and segments in the virtual address space. This binding is done indirectly via capabilities. Each window register holds an index for a capability in the currently executing software module's capability list structure. A Capability List index consists of a three bit field to select one of the up to eight Capability Lists, plus an offset within the C-List.

To overlay the processor's address space, ie. to change the mapping from page frame to segment in the virtual address space, a program simply writes a new capability index into the appropriate window register. This overlay operation is completely protected; the program can only reference segments for which it has a Capability. The act of writing the Capability index into the window register activates the Kmap. The Kmap retrieves the selected Capability from main memory and places it in its "Capability cache". The Kmap adjusts its internal tables so that subsequent references to the page frame will map to the segment specified by the Capability. If the segment is local to the processor then the Kmap may also change the relocation register in the Slocal so that references to the segment can be performed at full speed without the intervention of the Kmap. The Slocal, for cost and performance reasons, does not have the hardware necessary for bounds checking on variable sized segments. Thus only fixed size 4 K byte segments can be accessed without Kmap assistance.

The Cm* mechanism for address space overlaying should be contrasted with mechanisms in other computer systems. When executing a large program on a processor with a small immediate address space, the time taken to overlay the address space can have a crucial effect on performance. Measurements made of the execution of the operating system HYDRA [Wulf et al, 76] on the C.mmp multiprocessor showed that relocation registers were being changed approximately every 12 instructions. (This does not, however, imply that user programs perform overlay operations this frequently.) Within the operating system this overlay operation is a single PDP-11 MOVE instruction because no protection is involved. However for user programs running under HYDRA, an overlay operation requires invocation of the operating system with several hundred instructions of software overhead. Subsequent optimization, and partial microcoding, have greatly reduced this overhead.

Figure 3.4 shows the conceptual translation from a 16 bit processor-generated address to a virtual address. The four high order address bits from the processor select one of 15 Window registers. The Window register holds an index for a Capability in the executing software modules Capability List structure. The 16 bit segment name from the selected Capability is concatenated with the 12 low order bits from the processor to form a 28 bit virtual address.

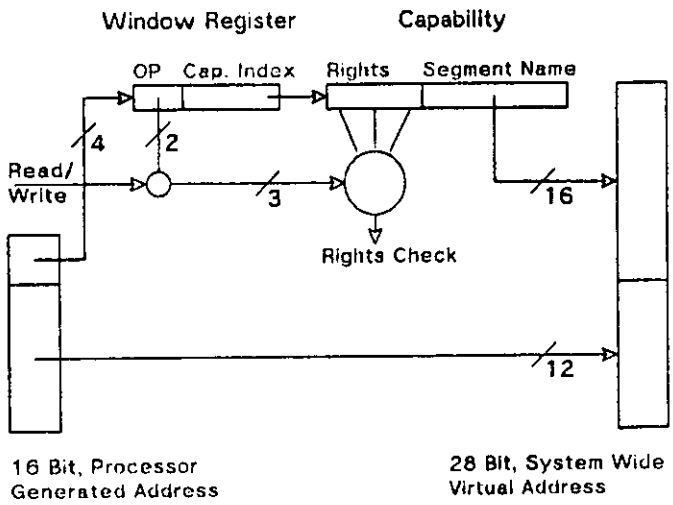


Figure 3.4
Conceptual Virtual Address Generation and Rights Checking

Figure 3.4 also shows the read/write indicator from the processor being concatenated with two bits in the address expansion registers to form a three bit opcode. The corresponding bit in the Capability rights field is selected and tested. If the operation is not permitted then an error trap is forced.

3.4 Virtual to Physical Address Mapping

The mapping from virtual to physical address depends on the location of the segment in the network and, of course, on the type of the segment. We begin with the case of a simple read/write segment residing within the same cluster as the processor referencing the segment. This mapping is shown in Figure 3.5. The segment name is used to access the corresponding segment descriptor. The descriptor provides a limit value which is checked against the 12 bit offset in the virtual address. If the reference is out of the bounds of the segment then an error trap occurs. The offset is added to the physical base address from the descriptor. The resulting 18 bit value is a physical address within the 256 K byte address space of the computer module also specified in the descriptor.

If the virtual address references a segment outside the source cluster then the segment name is used to access an *Indirect Descriptor Reference* rather than the descriptor itself. The indirect reference simply indicates in which cluster the segment resides. The Kmap then passes the virtual address to that cluster via the inter-cluster buses. An alternative approach would be to have duplicate copies of the segment descriptors in every cluster. Thus the virtual-to-physical mapping could be done at the source

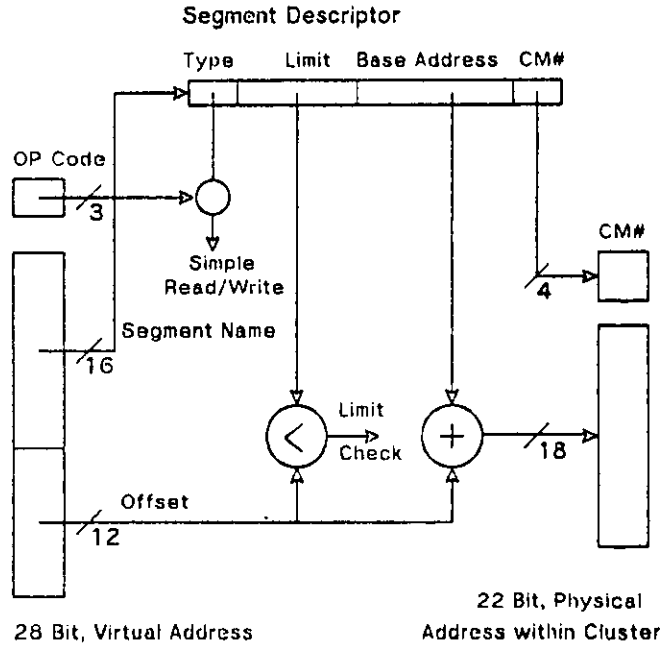


Figure 3.5 Virtual to Physical Address Mapping for a Variable Sized Segment

cluster, with possibly some savings in overhead. However, any attempt to change the virtual-to-physical binding of a segment (e.g. moving it to a different memory module or onto backing store) would require an effectively simultaneous change to all copies of the segment descriptor. In a large network this operation would be slow and cumbersome, if not impossible. A further advantage to ensuring that only a single descriptor exists for each segment is that a *Lock Bit* can be provided in the descriptor. The lock bit can be used to ensure mutual exclusion for special segment operations.

3.5 The Kernel Address Space

Each processor can execute in either of two address spaces. One is the *User Address Space* which was described above. The second is the *Kernel Address Space*, which is similar to a user address space with the addition of some mechanisms reserved for the operating system. The currently executing address space is selected by a bit in the Processor Status Word of the LSI-11. A *Kernel Environment* is similar to a User Environment; however segments at the third level of the Capability List structure (Figure 3.2) can be User Primary Capability Lists. That is, a Kernel Capability list structure can have user environments as substructures.

There are several additional pseudo registers provided in page 15 of the kernel address space. One of these, the *User Environment* register, holds an index for a Capability in the kernel environment which points to a user environment. This register specifies the current user environment for this processor. If the kernel writes a new index into the register the addressing state of the old user process is saved by the Kmap in the state vector part of the old user environment. The addressing state of the new user is then loaded from the specified new user environment. The addressing state is the value of the window and other system registers in page 15 of the executing program. Ideally, this operation, which performs a context swap by saving one addressing state and loading another, would also save the internal processor registers. Unfortunately there is no way for the Kmap to access the internal registers of an LSI-11. Thus internal registers must be saved and restored under program control.

4. Control Operations

The philosophy in Cm* is to implement all special control operations, such as interprocessor interrupts, by references to the physical address space. This not only avoids a proliferation of special control signals, but also allows the power of the system's address mapping and protection mechanisms to be applied to control operations.

The Slocal provides a three priority level interrupt scheme. An interrupt is invoked by writing into the appropriate physical address on the LSI-11 bus of the target processor. Thus an interrupt can be requested by a process anywhere in the network, provided the process has a Capability for a segment which maps to the correct physical address. Another example is the abort operation. If the appropriate bit is written, a NXM (Non Existent Memory) trap by the local processor is forced. This mechanism will be used when an error occurs during a remote reference by the processor.

The following examples show how references to special typed segments, or special operations on standard segments, are used to invoke microcoded operations in the Kmap.

4.1 Primitive Lock Operations

For processors in the PDP-11 family, most write operations are part of a read-modify-write sequence. In standard PDP-11s (including LSI-11's) this sequence is implemented as an indivisible, single bus operation. This improves performance by reducing bus overhead and allowing optimization of references to memory with destructive read operations (e.g. core and dynamic MOS memory). In C.mmp the indivisibility of these operations is

maintained through the switch to shared memory. This allows the implementation of Locks and Semaphores because a memory location can be both tested and set without fear of an intervening access by some other processor. Indivisible read-modify-write operations to nonlocal memory will not be implemented in Cm* because of increased bus and memory contention and hardware complexity. We will provide an equivalent function by making use of the Kmap's ability to lock a segment descriptor while it makes a series of references to the segment. To implement a basic lock mechanism two special segment operations are defined:

> Inspect the word addressed. If greater than zero, then decrement. Return the original value.
>
> Increment the word addressed. Return the original value.

4.2 An Inter-Process Message System

Message systems can provide particularly clean mechanisms for communication between processes [Brinch-Hansen, 73. Jefferson , 77]. In the past, a drawback to message systems has been the substantial operating system overhead in transferring a message from one process to another in a fully protected way. The architecture of Cm* provides an opportunity to build a fully protected message system which can be used with very low overhead.

A message port, or mail box, will be a special segment type. Messages will either be entire segments, passed by transferring capabilities, or will be single data words encoded as data capabilities. Two representative operations on Mailbox segments are:

Send (Message, ReplyMailBox, MailBox)

> This transfers capabilities for a message and a reply mail box from the caller's Capability List to the Mail box. If the mailbox is full then the caller is suspended.

Receive (MailBox)

> If the mailbox contains a message then a Capability for the message and a Reply Mailbox will be transferred into the caller's Capability List. Otherwise the caller is suspended.

Provided that the above operations are successful, they are performed completely in Kmap microcode, and messages may be passed with probably less than 100 microseconds delay. If the operation cannot be completed because the Mailbox is full or empty, then the operating system is invoked to suspend the requesting process. The Kmap can also request the operating system to wake up a suspended process when the operation is complete.

5. Development Aids

The development of hardware and software for a new computer system is a major undertaking. We have attempted to ease this burden by using a variety of aids. All the major hardware components were drafted using an interactive drawing package (a version of the Stanford Drawing Package). To facilitate the development of software, prior to the availability of hardware, a functional simulation of Cm* was programmed, which executes on C.mmp. Development of the Kmap hardware and microcode has been greatly benefited by the use of the "hooks" mechanism in the Kmap. This connection to the Kmap allows a program executing on an LSI-11 almost complete access to the internal state of the Kmap.

In order to expedite hardware debugging and software development, a host program development system was constructed. The host is connected to each Cm in the system by a Serial Line Unit (SLU) to allow down line memory loading and dumping from the associated Cm. In addition, the SLU makes console control functions for each LSI-11 available to the host computer [van Zoren, 75]. The Host in turn is connected to a PDP-10 timesharing system.

6. Concluding Remarks and Project Status

Cm* is projected to be constructed in three stages. The first stage is a ten-processor, three Kmap system. The subsequent stages will include 30-processors and later 100-processors. Detailed hardware design began in late July, 1975. As of late summer, 1976, a three-processor, one-Kmap system was operational. It is expected that the first stage Cm* configuration will be operational in the second quarter of 1977. The initial operating system is described in [Jones, et al. 77] and is being developed both on the Cm* simulator which runs on C.mmp and on the real hardware with the support of the Host Development system.

The essential features of the Cm* architecture have been presented. Both the coupling of a processor directly with each unit of shared memory and the three level bus structure which makes all memory accessible by every processor are primary features of the Cm* structure. Much of the sophistication in the architecture is associated with the address translation mechanisms. A description has been given of how the small processor address space of the PDP-11 is mapped into the larger global virtual address space of the Cm* system and how the global virtual address space is mapped onto the distributed physical address space of the Cm* system.

A number of important aspects of the Cm* project are outside the scope of this paper and interested readers are referred to other papers for a more complete discussion [Jones, et al. 77, Swan, et al. 76A, 76B, 77, Ingle and Siewiorek, 76A, Ingle and Siewiorek, 76B, Siewiorek, et al. 76], Reliability and performance models have been developed concurrently with the hardware design of the system and have been used to guide several important decisions concerning the structure of the Cm* implementation.

Acknowledgements

During the years of its initial development, many individuals have contributed to this project. Gordon Bell, Bob Chen, Doug Clark and Don Thomas contributed ideas to earlier versions of this architecture. Anita Jones and Victor Lessor have contributed to the present architecture. Miles Barel, Paulo Corrulupi, Levy Raskin and Paul Rubinfeld have all contributed to bringing the hardware to an early fruition. Kwok-Woon Lai and John Ousterhout are largely responsible for the successful development of the Kmap. Andy Bechtolsheim designed the Linc. Lloyd Dickman, Rich Olsen, Steve Teicher and Mike Titelbaum at Digital Equipment Corporation have provided information, ideas, and support critical to the success of the project.

References

[Anderson and Jensen, 76] Anderson, G. A. and E. D. Jensen., "Computer Interconnection Structures: Taxonomy, Characteristics and Examples", *Computing Surveys 7*, 4, December 1975, 197-213.

[Bell et al. 1972] Bell, C. G., J. L. Eggert, J. Grason, and P. Williams, "The Description and the Use of Register Transfer Modules (RTMs)," *IEEE Transactions on Computers*, Vol. C-21, No. 5, May 1972, 495-500.

[Bell et al. 1973] Bell, C. G., R. C. Chen, S. H. Fuller, J. Grason, S. Rege, and D. P. Siewiorek, "The Architecture and Applications of Computer Modules: A Set of Components for Digital Design," *IEEE Computer Society International Conference*, CompCon 73, March 1973, 177-180.

[Bell and Newell 1971] Bell, C. G. and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, New York, 1971.

[Brinch-Hansen 1973] Brinch-Hansen, Per, *Operating System Principles*, Chapter 8, "A Case Study: RC-4000," Prentice Hall, 1973.

[Denning 1970] Denning, P. J., "Virtual Memory," *Computing Surveys*, Vol. 2, No. 3, September 1970, 153-190.

[Fuller et al. 1973] Fuller, S. H., D. P. Siewiorek, and R. J. Swan, "Computer Modules: An Architecture for Large Digital Modules," *Proceedings of the First Annual Symposium on Computer Architecture*, University of Florida, Gainesville. Also in ACM SIGARCH, *Computer Architecture News*, Vol. 2, No. 4, December 1973, 231-236.

[Heart et al. 1973] Heart, F. E., S. M. Ornstein, W. R. Crowther, and W. B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," *IFIPS Conference Proceedings*, Vol. 42, NCC 1973, 529-537.

[Ingle and Siewiorek, 1976] Ingle, Ashok and D. P. Siewiorek, "Reliability Modeling of Multiprocessor Structures," *Proceedings IEEE CompCon '76*, September 1976.

[Ingle and Siewiorek, 1976B] Ingle, Ashok and D. P. Siewiorek, "Reliability Models for Multiprocessor Systems with and without Periodic Maintenance", Computer Science Technical Report, Carnegie-Mellon University, September 1976.

[Jefferson, 1977] Jefferson, David, "The Hydra Message System," to be published.

[Jones, et al. 77] Jones, A. K., R. J. Chansler, I. Durham, P. Feiler and K. Schwans. "Software Management of Cm*, a Distributed Multiprocessor, Submitted to the 1977 National Computer Conference.

[Siewiorek, et al. 76] Siewiorek, D. P., W. C. Brantley Jr., and G. W. Lieve, "Modeling Multiprocessor Implementations of Passive Sonar Signal Processing", Final Report, Carnegie-Mellon University. Pittsburgh, Pa. 15213, October 1976.

[Swan et al. 1976A] Swan, R. J., L. Raskin, and A. Bechtolsheim, "Deadlock Issues in the Design of the Linc," Internal Memo, March 1976.

[Swan et al., 1976B] Swan, R. J., S. H. Fuller and D. P. Siewiorek, "The Structure and Architecture of Cm*: A Modular, Multi-Microprocessor". *Computer Science Research Review 1975-76*, Carnegie-Mellon University, Department of Computer Science, Pittsburgh, Pa., December 1976, pp 25-47.

[Swan, et al. 1977] Swan, R. J., A. Bechtolsheim, K. Lai and J. Ousterhout. "The Implementation of the Cm* Multi-Microprocessor", submited to the 1977 National Computer Conference.

[Van Zoren, 75] Van Zoren, H. "Cm* Host User's Manual", Department of Computer Science, Carnegie-Mellon University, December 1975.

[Wulf and Bell 1972] Wulf, W. A. and C. G. Bell, "C.mmp - A Multi-Mini-Processor," *AFIPS Conference Proceedings*, Vol. 41, part II, FJCC 1972, 765-777.

[Wulf et al. 1974] Wulf, W., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications of the ACM*, Vol. 17, No. 6, June 1974, 337-345.

The Implementation of the Cm* Multi-Microprocessor

Richard J. Swan
Andy Bechtolsheim
Kwok-Woon Lai
John K. Ousterhout

Carnegie-Mellon University
Pittsburgh, PA 15213

December, 1976

Abstract

The implementation of a hierarchical, packet switched multiprocessor is presented. The lowest level of the structure, a Computer Module, is a processor-memory pair. Computer Modules are grouped to form a cluster; communication within the cluster is via a parallel bus controlled by a centralized address mapping processor. Clusters communicate via intercluster busses. A memory reference by a program may be routed, transparently, to any memory in the system. This paper discusses the hardware used to implement the communication mechanism. The use of special diagnostic hardware and performance models is also discussed.

Computing Reviews Category: 6.20

Keywords and Phrases: multiprocessor, microprocessor, packet switching, virtual memory

This work was supported in part by the Advanced Research Projects Agency under contract number F44620-73-C-0074, which is monitored by the Air Force Office of Scientific Research, and in part by the National Science Foundation Grant GJ 32758X. The LSI-11's and related equipment were supplied by Digital Equipment Corporation.

1 Introduction

The companion paper, [Swan et al. 1977], has introduced Cm* as a large, extensible multiprocessor architecture. It has an unusually powerful and complex addressing structure which allows close, protected cooperation between large numbers of inexpensive processors. This paper describes the combination of hardware and firmware which implements the address space sharing and interprocessor communication mechanisms.

Cm* is a multiprocessor system as we define it (rather than a network of independent computers) because the processors share a common address space. All processors have immediate access to all memory. The structure of Cm* is shown in Figure 1.1. The primary unit is the *Computer Module* or Cm. This consists of a processor, memory and peripherals interfaced to a local memory bus and a "local switch". The local switch, or *Slocal*[1], interconnects the processor, its local memory bus and the *Map Bus*. The Map Bus provides communication between up to fourteen Computer Modules within a *cluster*, and is centrally controlled by the *Kmap*, a high performance microprogrammed processor. Each Kmap interfaces to two *Intercluster busses*, by means of which it communicates with the other clusters in the system.

There is a system-wide 28 bit virtual address space. This address space is divided into *segments* with a maximum size of 4096 bytes. Programs refer to segments indirectly via *Capabilities*, which are two-word items containing the global name of a segment and specifying access rights to the segment. The processors have a 16 bit address space which is divided into 16 pages. A mechanism is provided which allows a program to associate any Capability it posesses (and hence any segment to which it is allowed access) with any page in its immediate address space. A full description of the address mapping scheme is given in [Swan et al. 1977].

To demonstrate the viability of a structure it is necessary to build a pilot system with currently available components. To be a successful demonstration, the pilot system has to be a useful, economical computing resource in its own right. Therefore, in the Cm* network described here, many design tradeoffs were made on the basis of current technology and the resources available. The highly experimental nature of the project encouraged an emphasis

[1]The names used for hardware components of Cm* are derived from PMS notation [Bell and Newell, 71]. The leading, capitalized letter indicates the primary function of the unit, eg. Computer, Processor, Kontroller, Link, Switch. The subsequent letters, optionally separated with a period, give some attribute of the unit. For example, Slocal is a local switch. Pmap is a mapping processor. The name Cm* derives from (Computer.modular)* where * is the Kleene star.

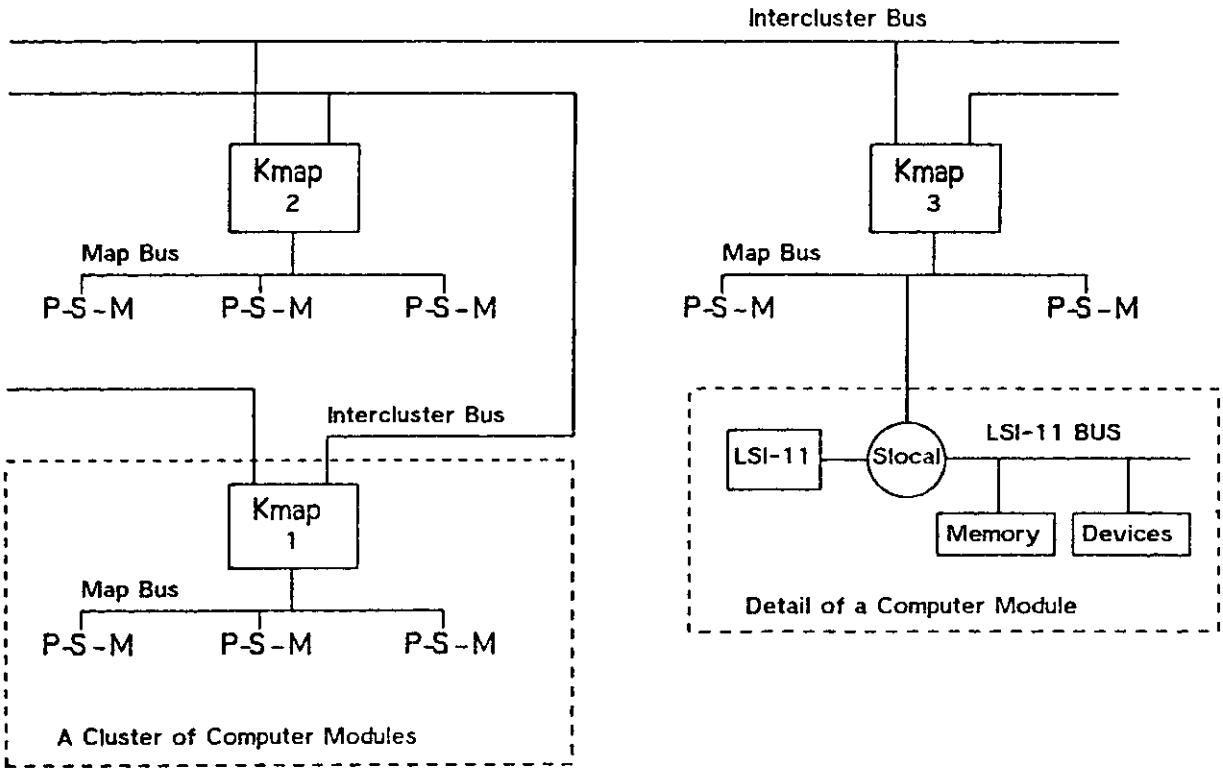


Figure 1.1 A Simple 3 Cluster Cm* System

on generality and ease of debugging in the hardware components, rather than just minimization of costs. There are many aspects of the detailed design which would have to be re-evaluated if the structure were to be implemented in a different technology or built as a commercial product. In particular the distribution of functions between the processors and the Kmap would be carefully reconsidered. The modular nature of Cm* makes it particularly suitable for implementation in LSI.

Section 2 illustrates the mechanism for memory references. The various hardware components of Cm* are described in the following six sections. Section 3 describes the processor-memory pairs and their interface to the Map Bus. In Section 4 opportunities for parallelism in the address mapping mechanism are considered. Three autonomous functional units of the Kmap are presented in Sections 5, 6, and 7. Section 8 describes the support given to hardware diagnosis and microcode development in the Kmap. For an effective implementation it was necessary to find a reasonable performance balance between system components. Some of the performance modelling which guided our judgement is presented in Section 9.

2 The Mechanism for Local and Nonlocal References

Addresses generated by processors in a Cm* system may refer to memory anywhere within the system. Mapping of an address and routing to the appropriate memory are performed in a way that is totally transparent to the processor generating the address. If an address is to refer to the memory local to that processor, the memory reference is performed in a completely standard way except that the Slocal relocates the high-order four bits of the address. See Figure 2.1.

When the page being referenced is not local (i.e. the "Map" bit for the referenced page is set in the Slocal) a *service request* is made to the Kmap by the Slocal. Upon receiving the service request the Kmap executes a Map Bus cycle to read in the processor-generated address from the Slocal, as well as the number of the Cm making the request, and two status bits indicating which address space was executing on the processor and whether the reference was a read or a write (see Figure 2.2). If the segment being referenced is local to the cluster, the Kmap will use information cached in its high-speed buffers to bypass most

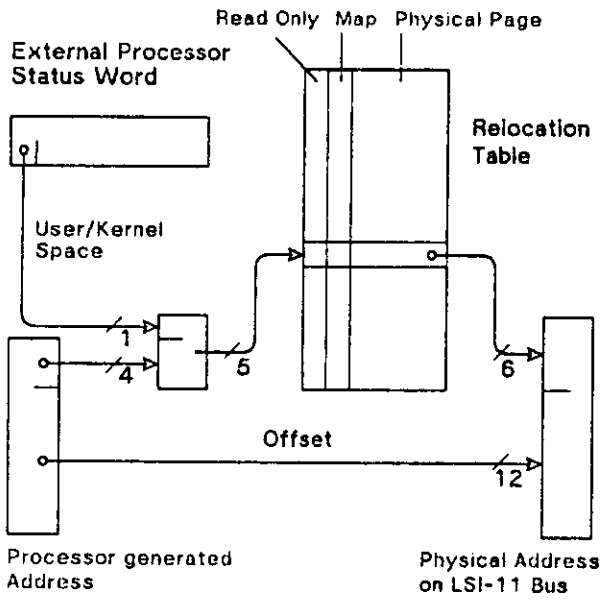


Figure 2.1
Addressing Mechanism for Local Memory References

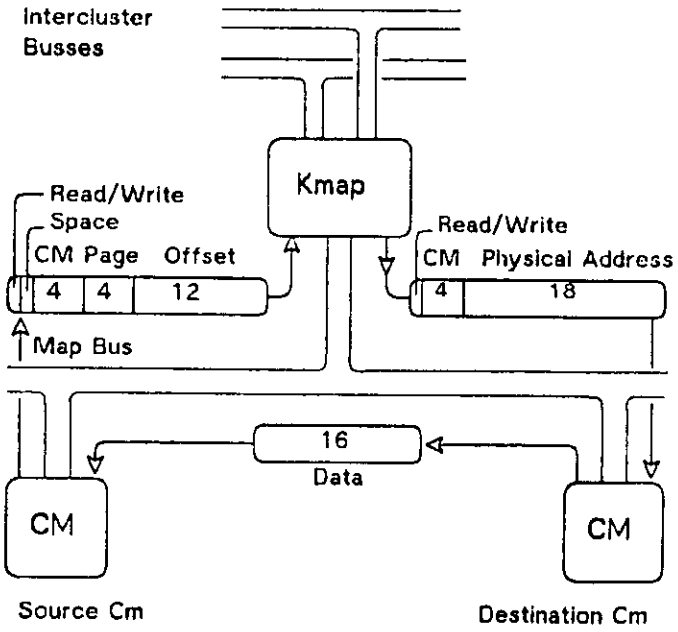


Figure 2.2 The Mechanism for Cluster-local References

of the processor-to-virtual-to-physical address mapping. Thus it can quickly translate from the page number referenced by the processor to a physical address consisting of the number of the Cm containing the physical location and an eighteen-bit local address. A second Map Bus transaction is executed to pass this address, and a bit indicating whether a read or a write is to be performed, to the destination Slocal. If the operation is a write, the data may be passed directly from the Cm making the reference to the Cm containing the word to be written. The destination Slocal performs the read or write via a Direct Memory Access. When this is completed it issues a *return request* to the Kmap to acknowledge completion. A third Map Bus cycle is performed to transfer the data back to the processor that made the reference (in the case of a read) and to acknowledge completion of the reference so that the requesting processor may resume activity.

A second alternative when the Kmap receives an address to map is that the physical location being referenced is not local to the cluster. In this case the information cached in the Kmap for the page being referenced will not indicate a physical location directly; instead it will give a sixteen-bit segment name, the number of the cluster containing the physical memory allocated to the segment, and two bits used to extend the read/write bit to a three-bit *op code*. This information is combined with the twelve low-order bits of the original processor address to form the full virtual address of the object being referenced. See Figure 2.3. The virtual address, along with the processor data (if a write is being performed) is sent via an Intercluster Bus to the Kmap of the cluster containing the segment (if there is no Intercluster Bus directly connecting the two Kmaps the message will be steered from Kmap to Kmap until it reaches the destination cluster). The destination Kmap will then map the virtual address to a physical one within its cluster. Map Bus transactions will be executed to pass the physical address (and data if needed) to an Slocal which in turn performs the operation and returns acknowledgement (and, perhaps, data) back to the destination Kmap. A return message is used to pass back acknowledgement and data to the Kmap of the originating cluster. Finally, this Kmap will relay the data and acknowledgement back to the initiating Cm to complete the reference.

Several points are worth noting with respect to the above schemes. Except at the local memory bus level, where conventional circuit switching is used, all communication is performed by *packet switching*. That is, busses are allocated only for the period required to transfer data. The data is latched at each interface, rather than establishing a continuous circuit from the source to the destination. This approach gives greater bus utilization and avoids deadlock over bus allocation. All transactions are completely interlocked with positive acknowledgement being required to signal completion of an operation (it is possible to allow a processor executing a nonlocal write to proceed as soon as the data for the write has been received by the Kmap or destination Slocal, without waiting for completion of the operation; however in this case the Kmap will expect to receive acknowledgement in place of the processor so that appropriate actions may be taken if none is received).

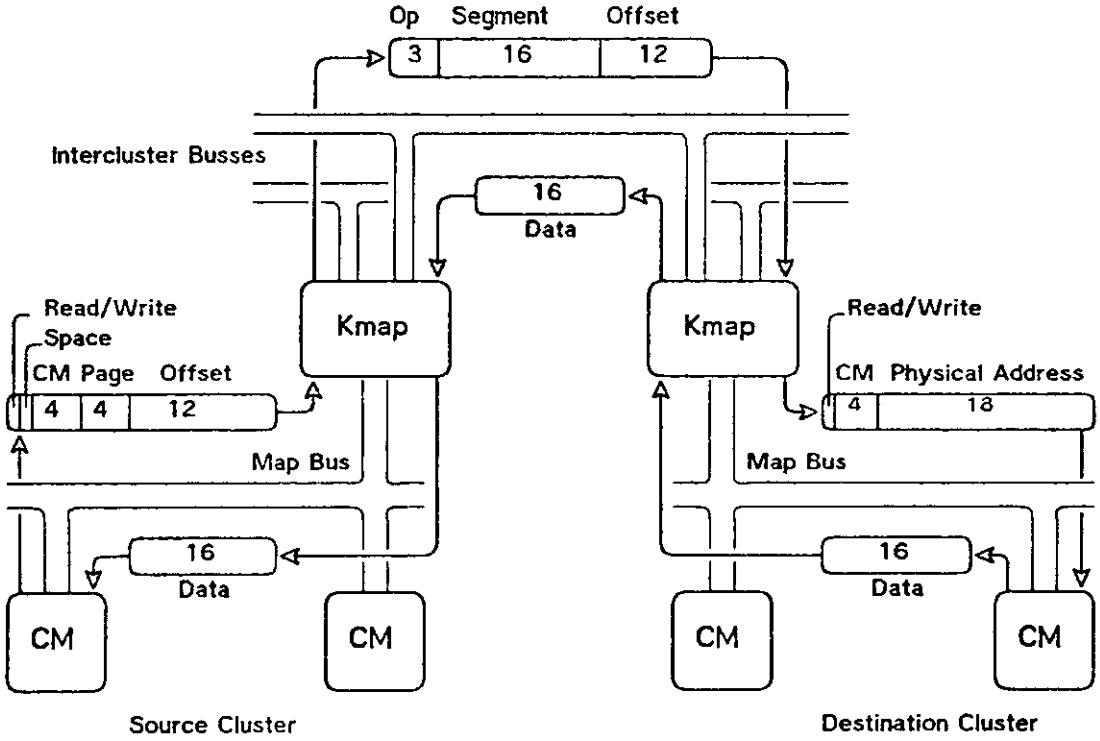


Figure 2.3 The Mechanism for Intercluster References

The complete processor-to-virtual-to-physical address mapping is performed only in the case of intercluster references. As the locality of a reference increases the amount of this mapping that may be bypassed (and hence the speed of the reference) increases, with local caches of certain mapping information used to effect the bypass. An important characteristic of the addressing structure is that there is exactly one Kmap that may perform the virtual-to-physical mapping for a given segment. The requirement that all references to a segment occur with the cognizance of a single Kmap greatly simplifies the moving of segments and the implementation of operations requiring mutual exclusion.

3 The Computer Module

The first level of the Cm* network hierarchy is the *Computer Module*, or Cm. The Cm's provide both the memory and processing power for the multiprocessor system.

The decision to use a standard, commercially available processor (the DEC LSI-11) has had a considerable impact on the design. Use of a standard instruction set has made a large pool of software and software development aids directly available. The not inconsiderable effort to design and implement a new processor has been avoided.

At the software level, the prime disadvantage of the LSI-11 instruction set is that only 16 bit addresses can be directly manipulated. The companion architecture paper discusses in detail the mechanism used to expand a processor's address space from 16 bits to 28 bits.

3.1 The Components of a Computer Module

A Computer Module, Figure 3.1, can act as a stand alone computer system. The standard commercially available components include the DEC LSI-11 processor and dynamic MOS memory. Any LSI-11 peripheral may be used on the bus, including serial and parallel interfaces, floppy and fixed head disks, etc. The standard 16 bit memory has been extended with byte parity. Memory refresh is normally performed by microcode in the LSI-11; however, the fact that a processor may be suspended indefinitely while awaiting the completion of a complex external reference has made it necessary to augment each Cm with a special bus device to perform refresh.

The most important component which has been added to each Cm is the Slocal. This provides the interface between the processor, the Map Bus and the LSI-11 Bus. The prime function of the Slocal is to selectively pass references from

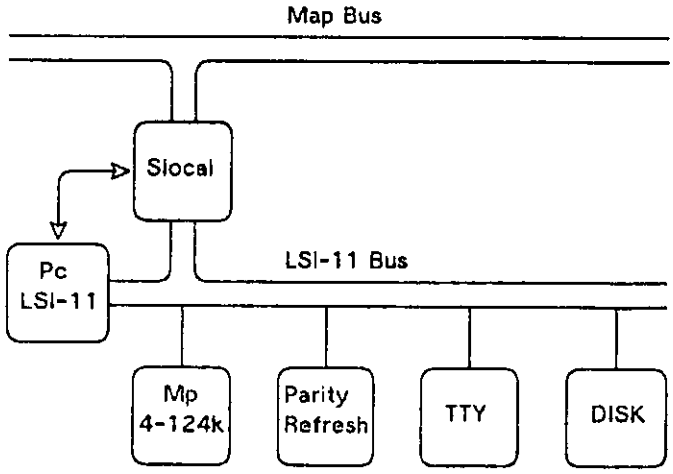


Figure 3.1 Details of a Computer Module

the processor to either the LSI-11 Bus or the Map Bus and to accept references from the Map Bus to the LSI-11 Bus. The Slocal also provides simple address relocation for references made by its processor to local memory. Figure 2.1 shows how this relocation is performed; the "Map Bit" in the local relocation table is set for pages which are not in the local memory of the processor.

In addition to the Local Relocation Table the Slocal provides a number of other control registers. All these registers are addressable as memory locations on the LSI-11 bus; however only the Kmap and highly privileged system code will have direct access to them. One of the key registers is the *eXternal Processor Status Word* (XPSW<15:8>). The LSI-11 implements only the low order byte of the standard PDP-11 *Processor Status Word* (PSW<7:0>). Logic in the Slocal (with assistance of standard signals from the LSI-11) allows the XPSW to be saved and restored during interrupt, trap and other operations in unison with the internal PSW. The XPSW allows selective enabling of various Slocal functions and controls a simple three level interrupt scheme. On power-up the XPSW is cleared, which disables all special operations by the Slocal including the relocation of local memory references. In this mode the processor acts as a bare, unmodified LSI-11. The Local Relocation Table can be initialized either by console operations, execution of local bootstrap code or remotely by any processor in the network. After initialization, enabling Reloc Mode (XPSW<11>) will allow local relocation and give access to the rest of the network.

Incorrect use of PDP-11 instructions such as HALT, RESET, Move-To-Processor-Status-word, Return from Interrupt, etc. can cause loss of a processor, garbling of an I/O operation or enable circumvention of the system's protection scheme. The Privileged Instruction Mode bit (XPSW<13>) enables logic in the Slocal which detects the fetching of any "dangerous" instruction. An immediate error trap is forced if an unprivileged program attempts to execute a privileged instruction.

Several registers in the Slocal are concerned with providing diagnosis and recovery information after a software or hardware error is detected. Almost all errors are reported to the processor by forcing a NXM (Non eXistent Memory) trap. This includes errors detected by the Kmap during remote references. The Kmap signals the error by writing to the "Force NXM" bit in an addressable register in the Slocal. The Local Error Register indicates the nature of the error and whether the erroneous reference was mapped. The "Last Fetch Address" register is updated to hold the address of the first word of an instruction every time the LSI-11 fetches a new instruction. If an error is detected, this register is frozen until the Local Error Register is explicitly cleared. Also frozen in the Local Error Register is a count of the number of memory references performed in the execution of the instruction. In conjunction, these two registers provide sufficient information to restore the state of the LSI-11 for retry of the instruction during which the error was detected.

The Slocal also provides two interrupt request registers. Interrupt enable bits in the external processor status word allow masking of the interrupt requests. Provided reference is permitted by the memory protection scheme, any processor in the network can interrupt any other processor simply by writing to the correct address.

3.2 Data Paths for Nonlocal References

An idealized form of the basic data paths and latches within a Cm* cluster is shown in Figure 3.2. Depending on the address generated, a reference from the processor is passed either to the local memory bus or to the Map Bus. A local memory reference is performed in a conventional way. For a nonlocal reference, the address (and possibly data) is latched and a service request is issued to the Kmap. The broken line in Figure 3.2 shows the path of a read to the memory of another Cm in the cluster. The address from the source processor is read by the Kmap which translates it into a physical address within the memory of a Computer Module. This physical address is placed onto the Map Bus by the Kmap and latched at the target Cm. A conventional Direct Memory Access (DMA) cycle is performed by the destination Slocal, the data read is latched and the Kmap is again requested, this time with a return request. To complete the operation, the Kmap responds by transferring the data over the Map Bus from the target Cm to the requesting Cm (this simply requires the latch at the target Cm to be enabled onto the Map Bus and the latch at the

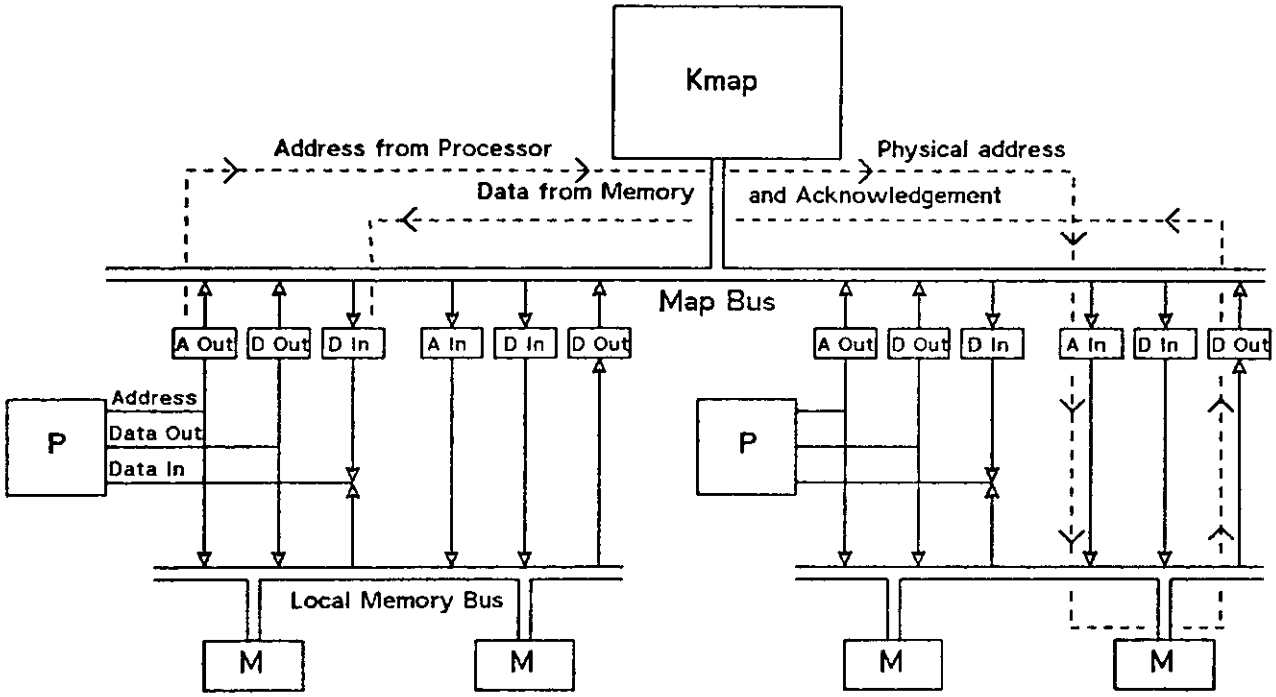


Figure 3.2 An Idealized and Simplified Representation of the Data Paths in a Cluster

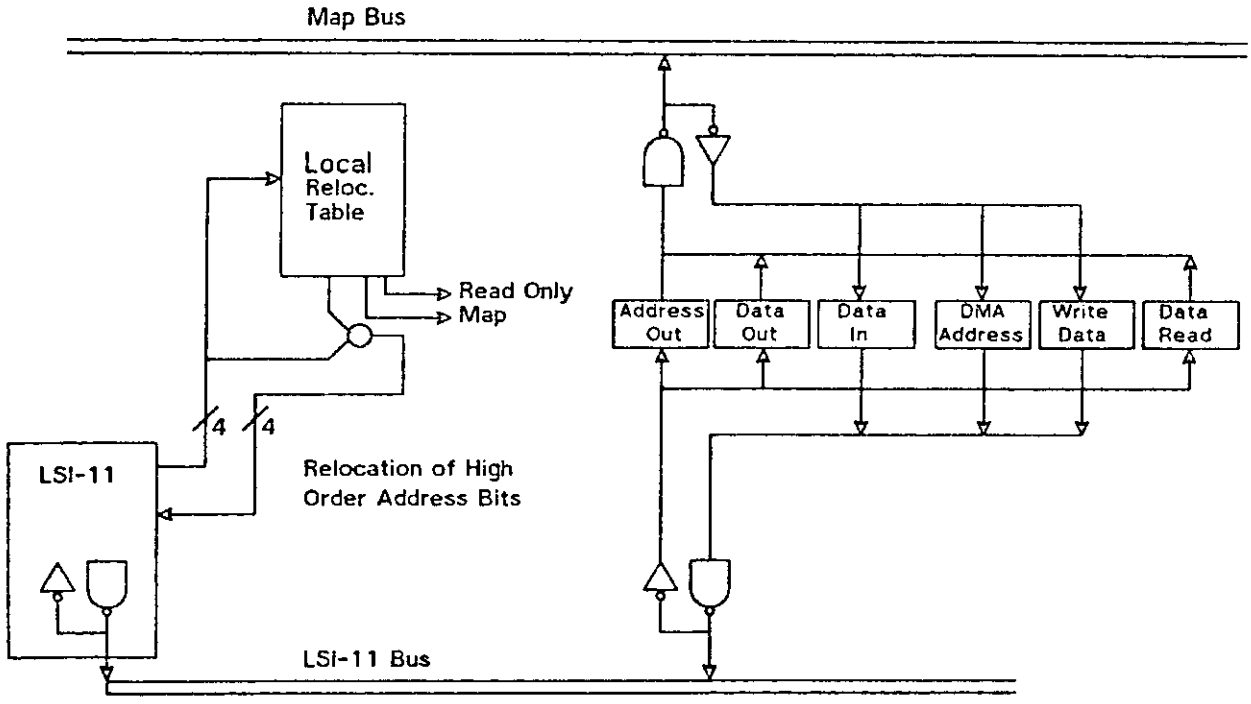


Figure 3.3 Simplified LSI-11 - Slocal Data Paths

requesting Cm to be strobed). At this point the source processor, which was suspended, is given the data as if a normal memory reference had been performed.

This simplified description of a Computer Module has been presented to emphasize the simplicity of the basic mechanisms required for an intra-cluster reference in Cm*. In the actual implementation using the LSI-11 processor the data paths are rather different than the idealized structure shown in Figure 3.2. The differences are due primarily to the need to minimize the changes to the LSI-11. Although still simplified, Figure 3.3 is a more accurate representation of the data paths and latches used to interface the LSI-11 and the LSI-11 bus to the Map Bus.

The processor board is modified so that the Local Relocation Table in the Slocal can be inserted in the data path of the four high order address bits. The timing margins in the processor's address path are wide enough to allow insertion of this delay without loss of performance. The LSI-11 Bus is the only data path from the processor for both local and non local references. If the processor were permitted to hold the LSI-11 bus while waiting for completion of a nonlocal reference then references from other processors in the network to memory on the LSI-11 bus would be blocked. This could very easily lead to deadlock situations. To give greater concurrency and to eliminate the deadlock potential, the Slocal is able (using simple microcoded state sequence logic) to force the processor off the LSI-11 bus while it is waiting for completion of nonlocal references. While the processor is forced off the local bus the Slocal takes over DMA bus arbitration for the suspended processor.

4 Concurrency within the Mapping Mechanism

Early in the design of Cm* the speeds of the various components in the system began to appear as follows: the time for a "typical" Map Bus transaction was about 0.5 microseconds; the time required in the computational unit of the Kmap for an address mapping was 1-2 microseconds; the time to transfer a message on an Intercluster Bus was 2-4 microseconds; and the time for an Slocal to execute a read or write requested by the Kmap was 3-4 microseconds. In referring to the mechanisms for nonlocal mappings it can be seen that no single component is responsible for a very large fraction of the time required for a nonlocal reference. Thus if each cluster had a mapping concurrency of one (only one nonlocal reference could be processed at a time per cluster) both the utilization of the mapping components and the throughput of the mechanism would be low (the effect of concurrency on system performance is discussed quantitatively in Section 9). In addition the possibility of deadlock in intercluster references is introduced.

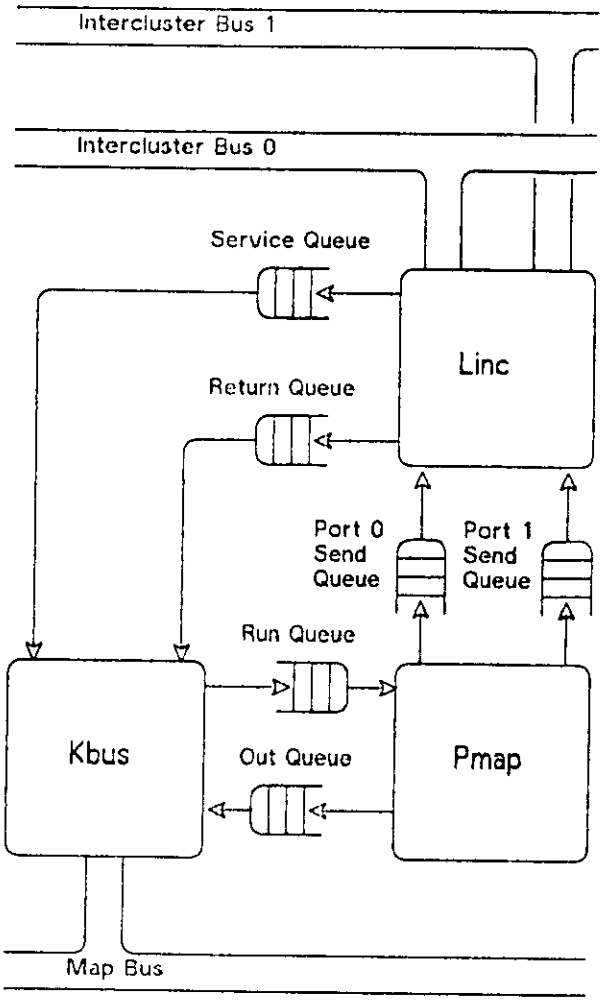


Figure 4.1 The Components of the Kmap

The solution adopted for Cm* was to separate the four functions whose timings are given above and to allow a concurrency of eight in the mapping mechanism of each cluster. The packet-switched nature of Cm* yields cleanly to this approach, and requires only that queues be implemented to store messages at the interfaces between the components. Figure 4.1 depicts this structure, in which the Kmap has been logically sub-divided into three separate units: the *Kbus*, which is master of the Map Bus and controls all transactions on it; the *Pmap*, or mapping processor, which does all the address translation and maintains the cache used to speed up mapping; and the *Linc*, or intercluster link, which presides over the transmission of messages between clusters.

One other notion must be introduced before proceeding to a detailed discussion of the components of the Kmap, namely that of a *context*. Operations requiring mutual

exclusion (for example, changing the virtual-to-physical mapping of the system) will be implemented in Cm* as memory references to "special" segments which will then cause the Kmap to perform the desired operations in a protected way. In general these operations will require several references by the Kmap to main memory. If the Pmap is to be used for other mappings while these main-memory references are being made by the Kbus and Slocals, there must be some means of saving and restoring its state so that processing can be resumed when the memory reference has been completed. The solution adopted is to provide registers in the Kmap to save and restore state for up to eight overlapping operations. A mapped operation in some stage of processing by the Kmap is referred to as a *context*. Each context has allocated to its exclusive use eight general-purpose registers and four subroutine linkage registers (one of which is used to save the microprogram address while awaiting the completion of Map Bus transactions).

The Kbus maintains the status of the eight Pmap contexts and allocates them to new service requests. The context number and other status are then placed in the *Run Queue* to signal the Pmap that the context is runnable. The mapping processor activates the context by removing its number from the Run Queue and starting execution of microcode at an address determined by the status bits. When the new context is activated the processor address is mapped, and a request for a main-memory reference is placed in the *Out Queue* (during this time the Kbus has been free to read in service requests or perform functions requested by the Pmap). A *context swap* is executed in the Pmap to deactivate the current context pending the completion of the memory reference and to activate the next one in the Run Queue. The Kbus transfers address and data to the destination Slocal, then processes other requests while the memory reference is being performed. When the memory reference is completed the Kbus either reads the acknowledgement and/or data back into the Kmap and places the context back in the Run Queue for reactivation, or it sends the acknowledgement back to the processor that originally made the service request (thereby completing the mapping operation) and marks the associated context as "free" for reallocation to a new service request. The fact that a context remains allocated to each nonlocal reference until that reference is completed (regardless of whether or not more Pmap processing is expected to be needed) means that if an error is detected the context can be reactivated and will have enough state information to handle the error in an intelligent fashion.

Communication between the Linc and Pmap is similar to that between the Kbus and Pmap; the Pmap queues a request for an intercluster message to be sent (separate queues are provided for each Intercluster Bus) and suspends the requesting context. When a return message is received for the context the Linc causes the Kbus to reactivate the context in the Run Queue. When an incoming intercluster message is received by one of the Linc's Intercluster Bus Ports, it is queued and a request is issued to the Kbus to allocate a free context to the request and activate it in the Run Queue.

5 The Kbus and the Map Bus

Because of the great variety of tasks it must perform and the necessity that it be able to respond to errors in an intelligent way, the Kbus was designed as a microprogrammed processor controlled by 256 40-bit words of read only memory. It has a microcycle time of 100 nanoseconds which is synchronized with the 150 nanosecond clock of the Pmap and Linc at 50 nanosecond intervals. Figure 5.1 shows the major elements of the bus controller.

The Map Bus contains 38 signals, of which 20 are bidirectional lines used to transmit addresses and data between the Slocals and Kbus of the cluster. The Kbus is master of all transactions on the bus; as such it specifies a source and destination for each cycle as well as status bits indicating the use of the data (address, data, etc.). The bus is synchronous, with the Kbus generating all of the strobes used to transmit data. Each Slocal is provided with private service and return request lines to the Kbus. The arbiter section of the Kbus scans these in a pseudo round robin priority scheme.

The Kbus maintains the queues and registers used for communication with the Pmap. The Run Queue contains eight eight-bit slots (and thus is guaranteed never to overflow), each containing a three-bit context name and five additional bits of activation status. The Out Queue contains four 39-bit entries. The Pmap loads this queue to request Kbus operations and must check its state before loading to insure that it never overflows. Each Out Queue slot contains an op code used to select one of thirty-two Kbus operations, and additional address, data, and context information relevant to the operation. Two registers are loaded by the Kbus on behalf of each Pmap context. They are readable only by the Pmap and writable only by the Kbus. The *Bus Data Register* contains the last data word read in from the Map Bus for the context and the *Bus Condition Register* gives control and status information for the transaction.

The Kbus is responsible for the allocation and deallocation of contexts, and maintains the status of each context for this purpose. It also keeps two additional bits of status for each context which are used to insure that, when a context suspends itself to await the execution of a main-memory reference or the sending of an intercluster message, an acknowledgement of the completion of the operation is received within a reasonable time (two milliseconds). If a suspended context times out it is forcibly reactivated with status bits indicating the error.

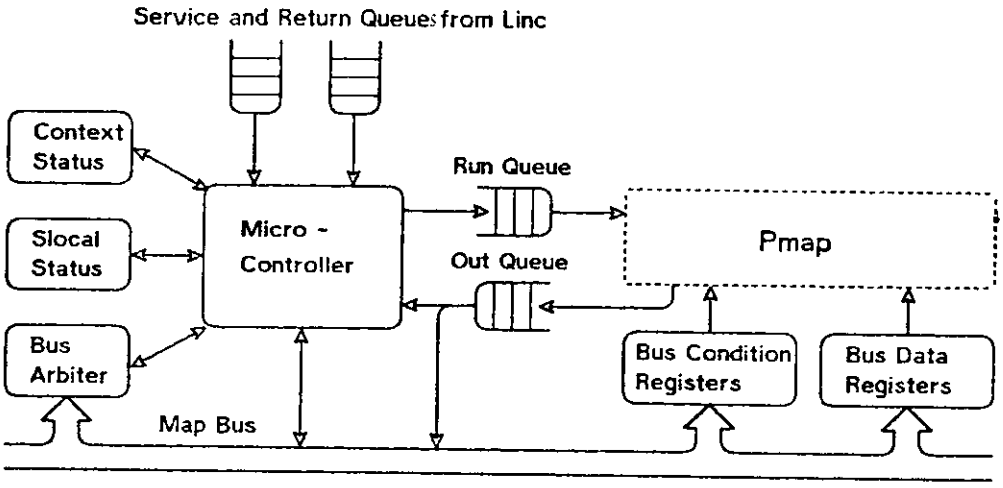


Figure 5.1 The Components of the Kbus

The Kbus also maintains nine bits of status for each Slocal in the cluster indicating whether the Slocal is busy with a Kmap-requested memory reference and, if so, what to do with the information returned at the end of the transaction. This status is set whenever a local memory reference is initiated and is used to insure that two contexts do not simultaneously try to request a memory access through the same Slocal.

6 The Pmap, the Address Mapping Processor

The mapping processor of the Kmap, or Pmap, is a sixteen-bit horizontally microprogrammed processor. It occupies a central position within the Kmap, coordinating the activities of the other components. It is pipelined and has a cycle time of 150 nanoseconds. Microinstructions are 80 bits wide; a 1K*80 bipolar RAM is used as a writable microstore. The Pmap also uses a high-speed 5K*16 RAM to store the active Capabilities and segment descriptors. In addition to performing the basic address translation for the nonlocal references of a cluster, the Pmap must support certain operating system primitives, statistics gathering, and other experimental functions without excessive performance degradation.

6.1 Data Paths

A register transfer level diagram of the Pmap is given in Figure 6.1. The main data paths consist of three internal high speed tri-state busses. Two of these, the *A* and *B busses*, take data from various sources and feed them to the inputs of the Arithmetic Logic Unit. The third bus, the *F Bus*, takes the ALU output and distributes it to various parts of the Kmap. The Kbus and Linc are also connected to these busses. Pipeline latches are used to overlap fetch of operands with current data operations.

The *Shift and Mask Unit* provides the ability to perform field-extraction on one of the ALU operands. This capability is important since the Pmap frequently deals with packed information in segment descriptors, intercluster messages, etc. The input to the Shift and Mask Unit is rotated by an arbitrary amount and then masked by one of 32 16-bit standard masks stored in a PROM.

For efficient address mapping, it is crucial that the Kmap have fast access to the information it needs to perform the virtual-to-physical address translation. This information consists largely of the active Capabilities and segment descriptors, of which up to 448 may exist in the cluster at a time (sixteen in each of two address spaces for each of fourteen processors). Although content addressable memory was not used because of the large capacity needed, the careful positioning of tables within the data memory, combined with a hash-coded list structure used for storing descriptors, has produced a cache-like structure.

The data memory, or Mdata, is divided into 1024 (expandable to 4096) *records*, each record containing five 16-bit words. The record organization was chosen because the segment descriptors, with cacheing information, fit comfortably within this 80-bit space. Each word has associated with it two parity bits, one for each byte. The memory is word addressable, with the record address coming from the *Data Address Register* (DADR) and three-bit

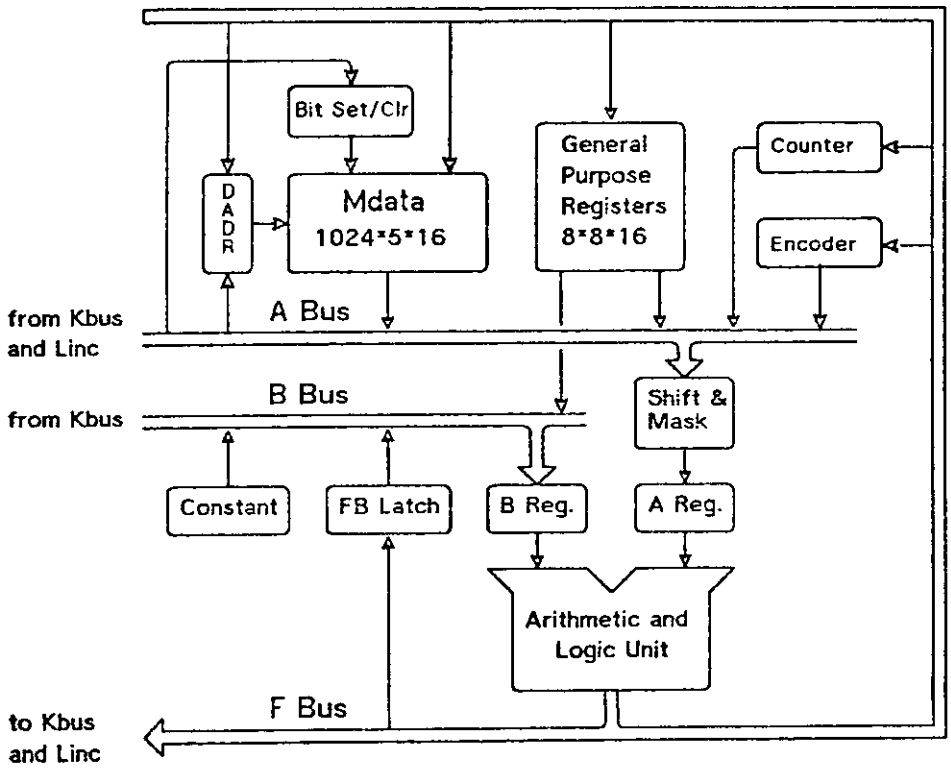


Figure 6.1 Data Paths in the Pmap

word indices from fields in the current microinstruction. Thus once the record address of a descriptor or capability has been computed, the individual subwords may be accessed without expending further cycles to generate data memory addresses.

Data to be written in the Mdata may be taken either from the A Bus or F Bus. Because it is frequently necessary to set and clear status bits in segment descriptors (for example the "dirty" and "use" bits used for demand paging, and the lock bit used for mutual exclusion) bit set and clear logic is provided for data input from the A Bus. It provides for the setting or clearing of either or both of the two high-order bits of the input word. To further increase parallelism, it is possible to simultaneously read and write different words of the same record. It is therefore possible, say, to set the "use bit" in one word of a segment descriptor and at the same time extract the segment limit from another word of the same descriptor.

6.2 Microprogram Sequencing Logic

One characteristic of the Cm* address mapping algorithms is the large number of conditions to be tested. The service of a typical request will require testing of request status, operation type, and segment type and checking of the following conditions: protection violation, descriptor locked, segment localizable etc. To perform address mapping within a reasonable number of cycles requires the Pmap to have a flexible multi-way branch capability.

A block diagram of the microprogram sequencing logic is given in Figure 6.2. A *Base Address* is selected from either the *Next Address* field in the current microinstruction or the output of the *Subroutine Linkage Registers*. Two bits in the microinstruction select the mode of branching (two-way, four-way, sixteen-way) and two three-bit fields control six 8-to-1 condition code multiplexers. Multi-way branching was implemented in the conventional way by OR'ing the selected condition codes with the Base Address. The address thus generated is stored in *MADR*, the *Microprogram Address Register*, to fetch the next microinstruction. There is a conditional override mechanism that can prohibit a potential 16-way branch. When the override condition is true, a branch is taken to a seventeenth location regardless of the value of the 16-way branch condition code.

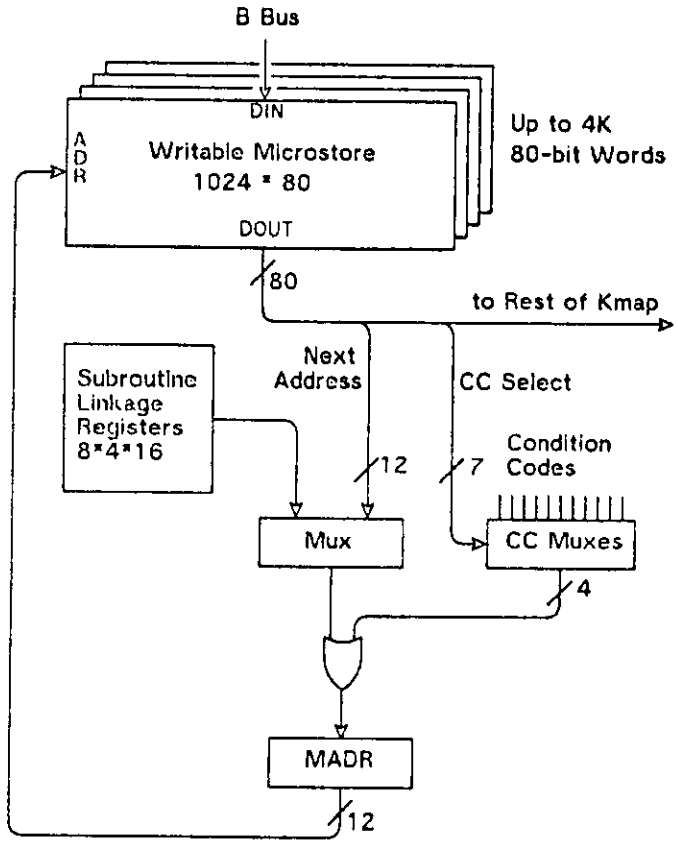


Figure 6.2 Microinstruction Address Generation Logic

6.3 Context Considerations

There are a total of 64 general purpose and 32 subroutine linkage registers, allowing each context exclusive use of eight general purpose registers and four subroutine linkage registers. The *Current Context Number*, stored in the *Context Register*, selects the current register bank. Normally this register is loaded from the Run Queue when a context swap is executed. For diagnostic purposes the Pmap may directly load the Context Register, hence if required a microprogram may access the registers of any context. Each context may nest subroutine calls up to four levels deep. By convention, the zeroth linkage register is also used to store the reactivation address of a suspended context. The status bits in the Run Queue indicate whether a context is to be activated at its reactivation address (to continue an ongoing operation) or to be explicitly started at one of the first sixteen locations in the microstore (to begin a new operation, or handle certain error conditions).

7 The Linc and Intercluster Bus Structure

The Linc provides intercluster communication by connecting the Pmap to two *Intercluster busses*. Communication is in the form of short messages passed between Kmaps. Messages are stored in a *Message RAM* which is shared between the Pmap and the two Intercluster Bus *Ports*. Pointers to messages pass through an automatic system of queues. Messages are usually sent directly from source to destination cluster, but they can also be forwarded by intermediate clusters (thus allowing arbitrary network topologies to be constructed). Message routing is controlled by Pmap microcode. The goal in the Linc design was to provide fast, deadlock-free intercluster communication with a minimum of Pmap overhead.

7.1 Intercluster Bus Protocol

The intercluster busses contain 26 lines: 16 data, 2 parity, and 8 control. They operate in an asynchronous, interlocked fashion at a transfer rate of 450 nanoseconds per word. Mastership is passed cyclicly between requesting ports, effectively implementing a round robin priority scheme. The current bus master arbitrates future mastership in parallel with its current data transfers.

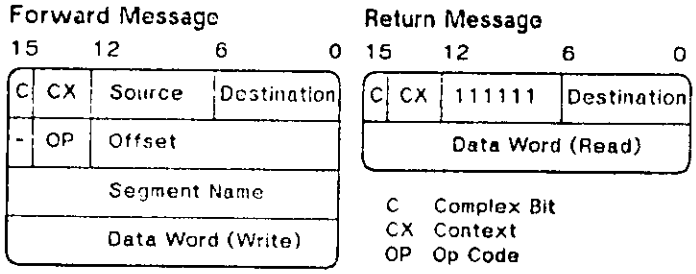


Figure 7.1 Standard Message Formats

Intercluster messages consist of one to eight 16 bit words. The most common formats are shown in Figure 7.1. The header word contains a six bit identifier for source and destination cluster, the source context number and the complex bit. A return message has a unique source field of all ones. The source context number is sent with the message to allow a direct reactivation of the suspended source context. The complex bit provides an escape mechanism to other message formats, eg for error messages or block transfers.

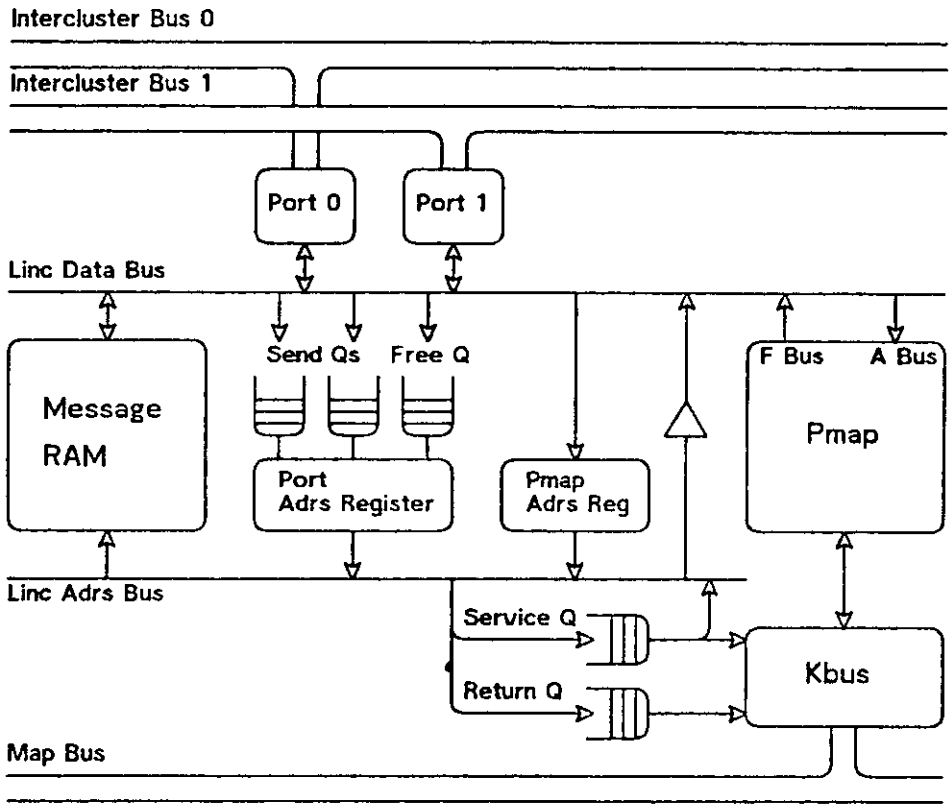


Figure 7.2 Components of the Linc

7.2 Components of the Linc (Figure 7.2)

Buffer space for messages is provided in the central 1K*18 Message RAM, divided into 128 buffers of eight words each. This is sufficient to avoid any possibility of deadlock over buffer allocation except in very large systems [Swan et al. 1976b]. The Pmap has priority for access to the Message RAM, although it is also directly accessible by the Ports. Several contexts may use the Linc in an overlapped fashion without interference since each context has private facilities for addressing message buffers. A context has two ways to address message buffers. It may use its context number to access a *reserved buffer* which is used for the creation of forward messages and to receive return messages. There is also a *Pmap Address Register* for each context to deal with incoming forward messages. Words within a buffer are selected by a Pmap microcode field. Each Port section has an address register and a word count register for accessing the Message RAM.

Five queues are maintained by the Linc. Two *Send Queues*, one for each Port, are used by the Pmap to request transmission of messages. To request that a message be sent on an Intercluster Bus, the Pmap places the address of the message buffer in the appropriate Send Queue. The *Free Queue* keeps the addresses of all the message buffers not currently in use. The *Service Queue* is used by the Linc to notify the Kbus and Linc of the addresses of incoming forward messages, and the *Return Queue* to request that the Kbus reactivate contexts when replies to their forward messages are received. All of the queues are implemented as partitions of a single 1K*11 bipolar RAM.

The Linc uses the same 150 nanosecond clock as the Pmap. For diagnostic purposes the Pmap has access to almost all of the internal state of the Linc and may execute all the internal microcycles executable by the Ports.

7.3 An Intercluster Message Transaction

A complete message transfer is shown in Figure 7.3. The Pmap at the source cluster creates the forward message in a reserved context buffer. Then its pointer is put into the appropriate Send Queue. The Linc pops the pointer off the Send Queue into the *Port Address Register*, acquires mastership of the corresponding bus and transfers

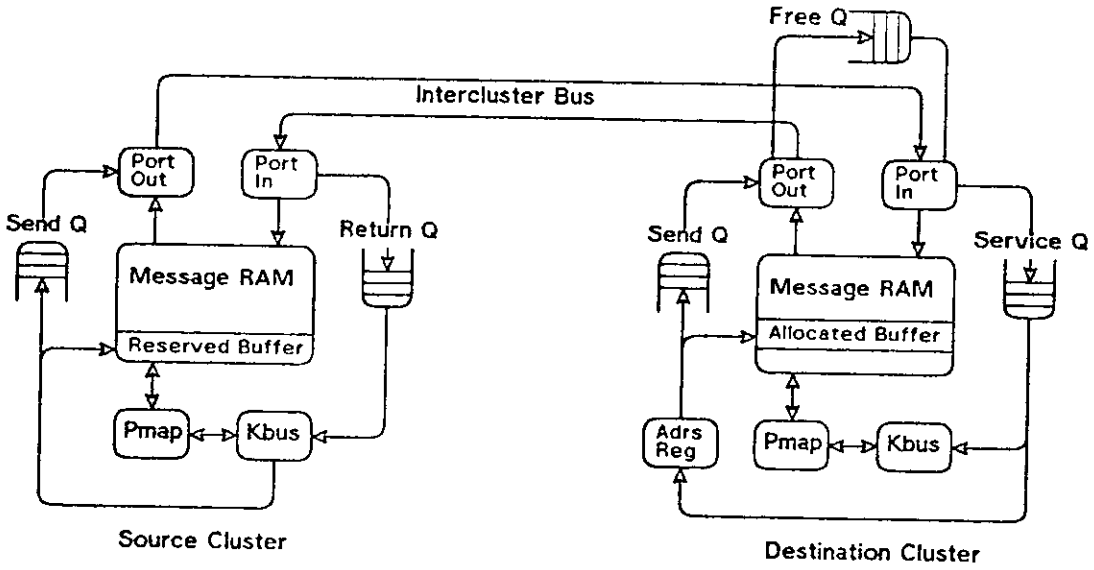


Figure 7.3 An Intercluster Message Transaction

the message, one word at a time, from its Message RAM onto the Intercluster Bus and into the Message RAM of the destination Linc.

At the destination side the receiving Port has already obtained a buffer from the Free Queue. If the message is received completely without error, then its pointer is placed into the Service Queue (if not, the message is ignored; a timeout will occur at the source). The Service Queue requests the Kbus to allocate a free Pmap context to service the message. It includes status bits to start up specific microcode. The context will transfer the pointer from the Service Queue into the Pmap Address Register and process the message, making appropriate main-memory references. It then creates a return message in the same buffer, setting the source field to ones to indicate this. On a Read, the data word will be appended. The buffer pointer of the completed return message is queued again in the Send Queue. When the message has been sent, the pointer is released into the Free Queue. At the original source the return message is placed in the reserved buffer for the requesting context. Its context number plus status is passed to the Return Queue and the context is reactivated to send data or an acknowledgement back to the requesting processor.

8 Development and Diagnostic Aids

A common strategy used to aid in hardware and/or microcode development is to construct a software simulator for the hardware. This allows initial debugging to be performed before the actual hardware is available and can provide a more comfortable environment in which to work. However, simulators are expensive both in terms of development effort and computer time; furthermore they cannot give an exact reflection of the hardware. Thus this approach leaves the final bugs to be found using the real hardware, and is of no aid in diagnosing component failures (rather than design errors). The alternative approach adopted for Cm* was to incorporate special hardware, called *Hooks*, directly into the Kmap for use in hardware and microcode development. The interfacing of the Hooks to a standard LSI-11 allows extensive software support for hardware development and diagnostics while at the same time providing a convenient environment for the debugging of microcode on the real hardware.

The Hooks give to an LSI-11, referred to as the *Hooks Processor*, the ability to intimately examine and change the internal state of the Kmap. They provide the capability for the Hooks Processor to load microcode into the writable control store of the Pmap, read the values on the A and B busses of the Pmap, and to independently start, stop, and single-cycle the Pmap-Linc and Kbus clocks. An interrupt is generated for the Hooks Processor whenever the Pmap clock stops (either due to a microprogram-invoked halt or a memory parity error on the control or data stores). Furthermore, several of the internal registers of the Pmap have "twin registers" associated with them which may only be loaded by the Hooks Processor. These alternate registers may be enabled via the Hooks to override microprogram-controlled values. The presence of the Hooks added approximately ten percent to the cost of the Pmap while enormously reducing system development time.

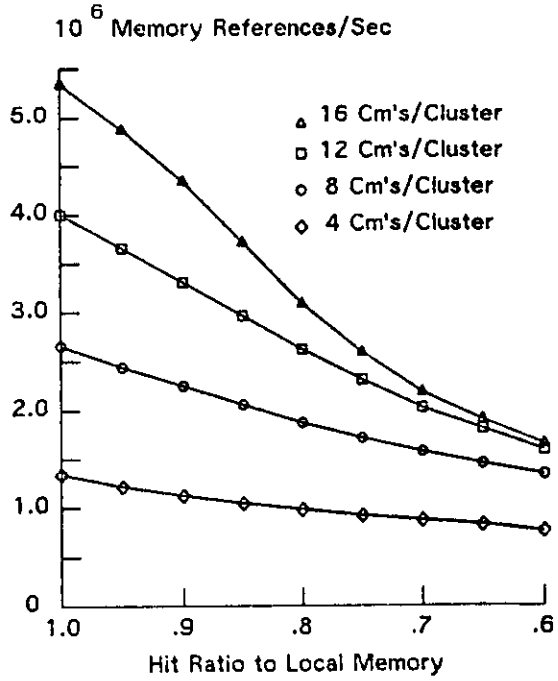
9 Performance: Measurements and Predictions

Before discussing the models used to estimate the performance of a Cm* cluster, several simple measurements (made on a cluster containing two processors) will be presented. The average time between memory references (including both code and data) made by a single LSI-11 executing entirely out of local memory varies between 2.5 and 4.0 microseconds, depending on the mix of instructions being executed. For a "typical" code sequence, based on measurements of compiled BLISS-11 programs, the inter-reference time was 3.0 microseconds. Measurements made on the same "typical" code sequence, except with all references mapped via the Kmap to the other processor in the cluster, yielded an average time between references of 7.7 microseconds. With the latter measurement there was no contention for use of the Map Bus, Kmap, or destination Slocal. Although no actual measurements were available at the time of this writing, it is expected that the time for intercluster references will be between 15 and 20 microseconds.

A simple queueing model was developed to estimate the performance of a cluster [Swan et al., 1976a]. The model assumed an exponential distribution of nonlocal requests, exponential service time in the Pmap, and exponential distribution of the total non-Pmap overhead incurred during a nonlocal reference. It is assumed that the Pmap is the primary cause of contention hence the waiting time for other facilities is ignored. Figure 9.1 plots the results of this analysis. The relative rate of memory referencing in a cluster is plotted as a function of the number of active processors and their *hit ratio* to local memory.

Because of the inability of the queueing analysis to model contention for all cluster facilities it was feared that the results would prove to be an optimistic estimate of cluster performance. Therefore a series of simulations was performed in order to model more closely the true operation of a cluster [Brown 1976]. The simulation and queueing results were in close agreement and so the simulation study will not be discussed further.

Figure 9.1 indicates that system performance is extremely dependent on the local hit ratio. It has been hypothesized that the local hit ratio would lie in the range between 85% and 95%, in which case the effect of the nonlocal references would be "reasonably" small. Unfortunately, this implies that code must be entirely local to the processor executing it. Two memory-intensive programs, a quicksort and a memory diagnostic, have been run on the initial Cm* system (one cluster, two modules). Measurements of the performance degradation when code and local variables are kept local but the area being sorted or diagnosed is moved to the other processor in the cluster indicate that local hit ratios of 90% or higher are being obtained in both cases. Expensive operating system functions such as block transfers are expected to lower



Local Reference Time = 3.0 uSec
External Service Time = 1.5 uSec
plus 6.5 uSec Constant Overhead

Figure 9.1 Absolute Cluster Performance

this figure, but it is also expected that most user programs will make less intensive use of shared databases than the above examples.

The queueing model was used to predict the degradation of cluster performance if either the Pmap were made slower (and thus cheaper) or if the concurrency of the mapping mechanism were eliminated. The results for a cluster containing twelve processors are shown in Figure 9.2. A slower Pmap was modelled by increasing its service time from 1.5 to 3.0 microseconds. The last model represents a cluster implementation where each external reference is carried to completion before servicing subsequent requests. This would be the situation if only one Pmap Context were provided, i.e. eliminating the concurrency between the Map Bus and the Pmap. Both the slow and non-concurrent clusters show enormous performance losses, especially at the low end of the 85% to 95% hit ratio range. The inability of slower or non-concurrent Kmaps to support large numbers of modules implies a need for more Kmaps per Cm* system. It also suggests that more intercluster communication will be required since each module will have fewer immediate neighbors.

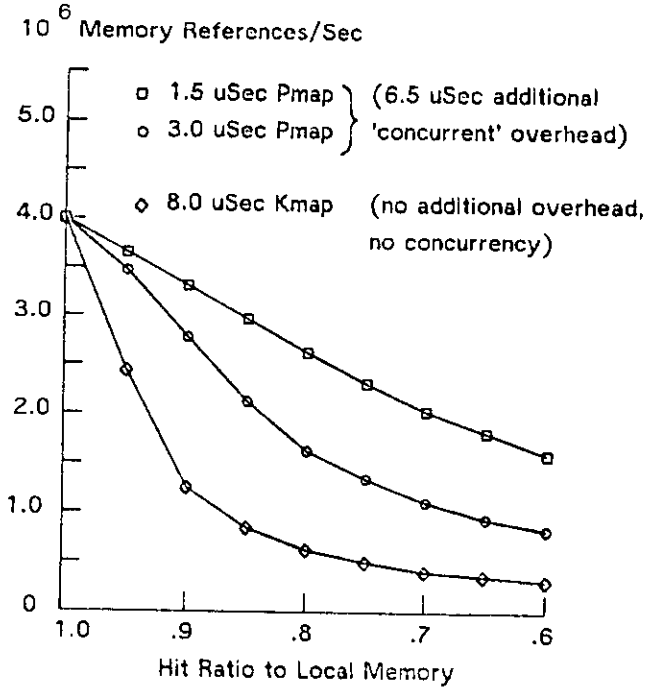


Figure 9.2 Cluster Performance with Slower Pmap or without Concurrency between Pmap and Map Bus

10 Conclusion

Detailed hardware design of Cm* begain in late July, 1975. The initial goal of a 10 processor, three cluster system is expected to be realized in the first quarter of 1977. Considering the Kmap alone, the time from the beginning of design to a working prototype (excluding the Linc) was less than nine months. It is felt that this relatively short development time is due to extensive use of automated design aids, microprogramming at almost every level and the inclusion of additional hardware to aid in debugging. The Hooks facility in the Kmap has been particularly successful. However it will not be possible to declare the overall system a success until it is regularly and reliably supporting a community of satisfied users.

References

[Bell and Newell 1971] Bell, C. G. and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, New York, 1971.

[Brown 1976] Brown, K. Q., "Simulation of a Cm* Cluster", Internal Memo, Computer Science Dept., Carnegie-Mellon University, May 1976.

[Swan et al. 1976a] Swan, R. J., S. H. Fuller, and D. P. Siewiorek, "The Structure and Architecture of Cm*: A Modular, Multi-Microprocessor", *The Computer Science Department Research Review 1975-1976*, Carnegie-Mellon University, December 1976.

[Swan et al. 1976b] Swan, R. J., L. Raskin and A. Bechtolsheim, "Deadlock Issues in the Design of the Linc", Internal Memo, Computer Science Dept., Carnegie-Mellon University, March 1976.

[Swan et al. 1977] Swan, R. J., S. H. Fuller, and D. P. Siewiorek, "Cm*: a Modular, Multi-Microprocessor", submitted to the 1977 National Computer Conference.

Software Management of Cm*, a Distributed Multiprocessor

Anita K. Jones
Robert J. Chansler, Jr.
Ivor Durham
Peter Feiler
Karsten Schwans

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

December, 1976

Abstract

This paper describes the software system being developed for Cm*, a distributed multi-microprocessor. This software provides for flexible, yet controlled, sharing of code and data via a capability addressed virtual memory, creation and management of groups of processes known as task forces, and efficient interprocess communication. Both the software and hardware are currently under construction at Carnegie-Mellon University.

CR Categories: 4.32, 4.35, 6.29

Keywords: modular decomposition, capability addressing, computer modules, virtual memory, multiprocessor scheduling, task force

This work was supported by the Defense Advance Research Projects Agency under contract F44620-73-C-0074 which is monitored by the Air Force Office of Scientific Research.

Introduction

Semiconductor technology advances are leading toward the inexpensive production of computer modules (i.e. a processor plus memory of a moderate size) on a single chip. Multiple computer modules interconnected to form a multiprocessor or a network offer a large number of processing cycles far more inexpensively than an equally fast uniprocessor. Yet, such a computer module system is useful only if a suitable fraction of the processing cycles can actually be used for applications.

The software designed to manage a computer module system can contribute substantially to making the system a cost effective environment in which to program applications. This paper discusses the software designed to manage a computer module system called Cm* which is currently under construction at Carnegie-Mellon University. We pay particular attention to the philosophy of software construction that influenced many of the design decisions.

For the purposes of this paper we will only review some attributes of the architecture that are salient to the design of operating system software. Companion papers [Swan et al., 77a Swan et al. 77b] describe and discuss the Cm* architecture in detail.

Cm* is a multiprocessor composed of computer modules, each consisting of a DEC LSI-11, a standard LSI-11 bus, memory and devices. We describe Cm* as a multiprocessor because the system's primary memory forms a single virtual address space; any processor can directly access memory anywhere in the system. To implement such a virtual memory, we introduced into each computer module a local switch, the Slocal[1] which routes locally generated references selectively to local memory or to the Map Bus (when the reference is to memory in another computer module). The Slocal likewise accepts references from distant sources to its local memory.

Connected to a single Map Bus may be up to fourteen computer modules that share a single address mapping and routing processor, called the Kmap. The computer modules, Kmap, and Map Bus together comprise a *cluster*. A Cm* configuration can be grown to arbitrary size by interconnecting clusters via Inter-cluster Busses (see Figure 1). (A cluster need not have a direct bus connection to every other cluster in a configuration.) Collectively, the Kmaps mediate each non-local reference made by a computer module, thus sustaining the appearance of a single virtual address space.

Because processors are numerous, applications of any size will tend not to be designed in the form of a single program executed by a sequential process. Instead we

[1] In several cases names of Cm* components are derived from the PMS notation described in [Bell and Newell 71].

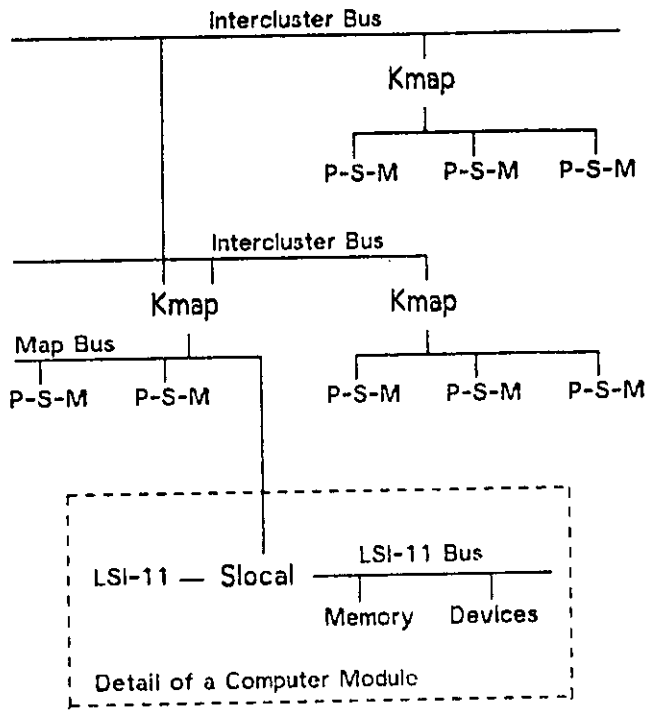


Figure 1. A Simple 3 Cluster Cm* System

expect users to create *task forces*, i.e. groups of processes cooperating to achieve a goal. Because the number of processes in a task force may vary with the available resources and task parameters, and because processes tend to be small (due to the relatively slow processors or limitations on the amount of local memory), a user will often be unconcerned with individual processes, communicating only with the task force itself.

The Cm* architecture offers to a user the option of employing tightly or loosely coupled processes. Loosely coupled processes communicate rarely, usually in conventional ways via a message transmission mechanism. Tightly coupled processes communicate often, sometimes using the efficient unconstrained paths provided by shared memory. Cm* permits both types of communication since it provides a message transmission facility as well as direct addressing of shared memory. Effectively, a user is free to view Cm* as either a multiprocessor or a computer network.

Software Design Methodology

Cm* is a vehicle for experimentation, particularly in the area of parallel decomposition of algorithms and their efficient implementation on a computer module processing resource. We expect it to be rare that an experimenter (which we will refer to as a user hereafter) is confident that all his code is debugged, since he will routinely alter parameters and even the code for his task forces in substantial ways. We also expect users to incrementally construct experiments. In addition we expect users to reconfigure modules (of software) combining them to form a new experiment.

Such a view of the user has led us to believe that it is as important for the kernel (or lowest level) software to support the user's software construction activities as it is to provide the primitive runtime facilities required for multiple users to share the computer resources in a disciplined cooperative fashion. Consequently, the software design reflects this concern. We view users as constructing their experiments by incrementally building *modules*[2]. Each module implements some abstraction useful to other modules that will come to depend upon it. A module then is a 'unit of abstraction'. It is implemented as

--code and data private to the module,
--a set of externally known functions that can be invoked by other modules making use of the abstractions, and
--a set of references to externally defined modules defining functions used in implementing the abstraction.

The kernel software supports the notion of a module by providing user facilities to create modules and to invoke functions of a module in a protected way. An invoked function is executed in an environment that gives it access to code and data that are part of the module, together with any actual parameters specified by the invoker. Thus the software enforces the boundaries of a module by providing a well defined transition between execution in one module and execution in another. Hopefully this will help contain the influence of errors and expedite debugging.

This notion of module is based on earlier work. In particular it is built on the ideas of modular decomposition discussed in [Parnas 73] and abstract data types [Liskov 74] as used in language design.

Module boundaries are used for protection purposes at runtime. Each function is executed with access only to those objects which it requires. In designing the kernel software, we have found that some of its modules implement rather complex abstractions. Yet not all uses of a module require the entire abstraction; some uses rely only on part of the abstraction while others rely on a simplified abstraction. For design purposes a module may be

[2]This paper always uses the words "computer module" to refer to the hardware structure, and will in the sequel use the (commonly accepted) single word "module" to refer to a programming abstraction. Context should also serve to eliminate any ambiguity.

partitioned into a strictly ordered set of *levels* as described in [Habermann et al. 76]. The purpose of dividing a module's design into levels is to permit either incremental introduction of the differrent parts of one abstraction or increasingly more complex (and powerful) versions of the entire abstraction. The introduction of complexity is postponed until it is truly required. Multiple levels of one module share data structures and even code.

The first level within a multi-level module may define only a subset of the functions of the complete abstraction, but that subset of functions is a useful self-contained, but limited version of the abstraction. Subsequent levels are introduced into the hierarchy as needed. Additional levels of a module may introduce entirely new data structures or extend existing ones. No protection boundaries exist between levels so that higher level code may manipulate data structures introduced in lower levels. Consequently, though module boundaries are translated into runtime protection boundaries, the boundaries between 'levels of design' are not detectable in the runtime implementation structures. We will illustrate this difference between modules and levels later when we discuss the Cm* message transmission module.

Levels within a module are strictly ordered. We can define a level A to be 'higher' than level B in another module in case A invokes a function defined in B. The set of all levels (of all modules) is partially ordered by dependency. In the design of operating system software there is not necessarily a cleanly identifiable division of a hierarchy of levels into supervisory and user software. The operating system facilities required by one user differ from those required by another, particularly in an experimental setting. The partially ordered system structure is in a form such that it is readily possible to replace 'upper' portions of the dependency hierarchy since level boundries are clear and the dependency relations between levels are known.

Cm* Software System Design

Before describing the kernel software design, we will define two notions that play an important part in that design: objects and capability addressing of objects. The basic unit which can be named, shared and individually protected, and for which memory is allocated for representation purposes is the *object*. Each object has a unique name and a definitive description used by the software system. Every object has a type that determines the structure of its representation and the operations or accesses which can be performed on it. Current design specifies three types of objects: *data segments*, which are linear arrays of words that may be read and written; *capability lists*, which are structures containing capabilities (to be discussed below); and *mailboxes*, which are structures containing messages.

Objects are named (addressed) using *capabilities* [Dennis and Van Horn 66, Lampson 69]. A capability may only be created and manipulated in controlled ways (by kernel provided capability functions). Since users cannot create or forge capabilities, possession of a capability is evidence that the user can reference the object whose unique name appears within the capability. A capability not only identifies a unique object, it records a set of *rights* indicating which of the defined operations (accesses) are permitted to be performed on the object. Controlled use of objects is enforced because an object can be accessed only if a program presents a capability naming that object which contains a right for the desired access. Since possession of a capability endows the possessor with the ability to perform accesses, capabilities also record those rights which a possessor may exercise with respect to the capabilities themselves. (For example, copying a particular capability may not be permitted.)

Based on the above discussion, we next describe the Cm* kernel software. The purpose of the initial levels of software is to provide facilities required for shared usage of resources in an 'enforcably cooperative' way. In addition we wish to assist users in programming and executing their experiments by providing convenient structures and functions for creating and executing modules. The operating system software itself is composed of a partially ordered set of levels. In several instances two modules are divided into a pair of levels. For convenient reference levels are labeled with a tag in the format 'module-level'. Modules are given alphabetic names; levels are numbered in increasing order as they appear in the system construction hierarchy. The kernel levels to be discussed in this paper are:

CAP-1: Capability referencing Performs mapping from a capability via a segment descriptor to physical representation of segment (including access control checking)

CAP-2: Capability addressing and memory allocation Defines an object address space and interpretation of an address; performs memory allocation ensuring that the segments used to represent objects are pairwise exclusive

ME-1: Environments and Modules Implements the creation and deletion of modules and execution environments

MSG-1: Conditional message transmission Defines the structures message and mailbox; permits sending and receiving of messages when process suspension is not required

DSP: Dispatching Defines hardware implemented data structures used to 'load' an environment onto the processor and commence execution

MPX: Multiplexing Selects the next environment to execute on a processor

ME-2: Environment relations Records the ancestry by which environments are related; provides for nested and parallel execution of environments

MSG-2: Unconditional message transmission Provides for sending, receiving, and replying to messages even if environments involved are forced to wait for an indeterminate time to complete message transmission

TI: Trap and Interrupt handling Provides routing of control when either interrupts or traps occur

A diagram indicating the dependency relations among these levels appears as Figure 2. An arrow from level A to level B indicates that a funtion in level B is invoked in level A. In addition, it is possible that level A invokes functions in any of the levels 'below' B in the dependency graph.

Capability Addressing

Module CAP provides capability addressing. Level CAP-1, which is implemented in Kmap microcode, interprets capability references to objects, i.e. it maps a capability to the physical representation of the object named by the capability. Because the state of an object may change and its physical representation may move, the system maintains a single definitive description of each object called a *descriptor* or *segment descriptor*. It records the type of the object, the physical description of its representation (including cluster, module, starting address, and size), state information (e.g. whether the representation is in core, dirty, or locked for Kmap usage), and the (reference) count of the number of outstanding capabilities for the object.

Every existing object has a unique name--the memory address of its descriptor. To perform a mapping from a capability to a object, the identity of the object's descriptor is determined from the capability. It, in turn, is referenced to determine the physical representation of the object. A capability reference fails if the right required to perform the operation desired by the addressing environment originating the reference is not in the capability.

Level CAP-2 extends level CAP-1 to provide for the generation of capability references (we refer to this as capability addressing), and for capability manipulation. Capabilities used for addressing purposes are stored in capability array objects called *capability lists*. Given a capability list CL and an index X, one can determine the X-th capability in capability list CL. This may be a capability for an object of arbitrary type, including a capability list object. By repeated application of capability indexing,

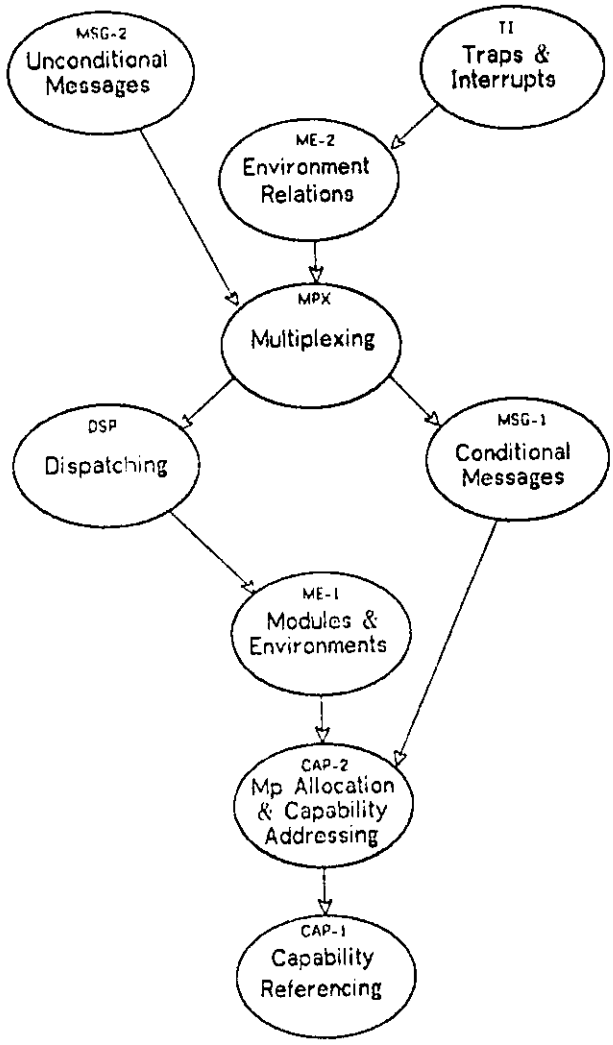


Figure 2. Levels and Modules of Cm* Software

objects to any depth can be addressed. Because capability list indexing is performed in microcode as well as in software, the architecture restricts indexing to depth 2 in any single operation. This means that in a single addressing operation the path to a target object may 'indirect through' at most two capability lists before arriving at the (third) target object. Whenever a processor is executing (i.e. generating capability addresses) one capability list is distinguished as the *primary capability list*. The first index of a capability address is an offset into this primary capability list.

CAP-2 also defines (microcoded) functions for creating, copying, moving, and deleting capabilities as well as for manipulating the rights encoded within a capability.

A Cm* processor (an LSI-11) has a word size of only 16 bits. To permit 16 bit addresses to be mapped to the arbitrarily sized Cm* memory, the notion of a *window* was introduced. It consists of 15 *window registers*, each of which can be thought of as holding a capability. (Actually, in the current design, each window register holds an index to a capability which can be indexed via the current primary capability list.) CAP-2 provides two (microcode implemented) functions *Segload* and *Unload* to associate and de-associate, a window register and a capability. To read or write a data segment, a capability for the segment must be *Segloaded* into a window register.

A 16 bit machine address is interpreted to select a window register (and thus a capability) and possibly to specify an offset into a segment of memory. For enhanced performance of capability referencing, the descriptors for the objects named in the capabilities associated with the window registers are cached in the Kmap. This mechanism provides virtual addressing and allows for conventional relocation of physical memory. It is sufficiently general to support the definition of Kmap microcoded operations on capability lists and mailboxes.

The last facility introduced in CAP-2 is that of memory allocation. Physical memory is allocated to hold segments so that no two segments overlap.

Modules and Environments

Level ME-1 provides for the creation and deletion of modules (as discussed earlier) and for executing invoked functions. A module is implemented by a *module capability list* containing

--capabilities for the code and data segments required to perform the functions defined in this module,
--a data segment containing a vector of *function descriptors* which specify the code to be executed when a particular function is invoked (e.g. the index into the module capability list for the segment containing code for this function), the number of parameters expected and the size of stack required to perform the function,
--a list of other 'known' modules containing functions that can be invoked by this module.

ME-1 also defines an *environment*, the structure created as a result of a function invocation. An environment is defined by several objects; one is the primary capability list which is private to a function invocation and acts as the root capability list for all addressing of objects during execution of the function.

The primary capability list contains capabilities for

--the execution stack (private to the environment)
--the module capability list which defines the module containing the invoked function,
--a *state vector* (private to the environment) which contains the processor and addressing state when the environment is not executing on a processor. (The state vector includes processor registers, processor status word, scheduling data, trap and error masks for communicating with the Kmap, and indices of the capabilities *Segloaded* into the window registers during the environments execution.)
--parameter objects specified by the invoker.

The module capability list contains capabilities for those objects shared by all who invoke a function in the module. The primary capability list contains capabilities which are local to a particular invocation of a function.

Level ME-1 provides functions for the creation, initialization and deletion of modules and environments. These, in turn, are used by level ME-2 in providing functions relating the execution of different environments. Functions *Call* and *Return* allow nested execution, i.e. the *Calling* environment is suspended for the duration of the execution of the newly created (*Called*) environment which terminates when the *Called* environment *Returns*. The function *Fork* permits an environment to request that a function be invoked to execute in parallel with its invoker until the function *Join* is performed.

ME-2 initializes a newly created environment to record priority information for scheduling purposes and to record the existence of a newly created environment in the *lineage* (family tree) of its creator. It is this lineage which is used by still higher levels to keep track of a task force, the set of environments which are cooperating to achieve some goal.

Message Transmission

The members of a task force need to be able to synchronize their actions and to communicate with one another. To this end module MSG defines an abstraction of a *mailbox* which can contain *messages*. A mailbox is capable of containing some fixed finite number of messages maintained in FIFO order. To permit users to communicate arbitrary objects to one another, rather than data only, messages are pairs of capabilities. (To transmit 16 bits of information, a user can create a *data capability* to contain this user specified information.)

Levels MSG-1 and MSG-2 differ in that MSG-1 provides only the functions *CondSend* and *CondReceive* to transmit messages when these functions can be completed without suspension of the invoker. *CondSend* succeeds in depositing a message into a mailbox only if the mailbox has room for it. *CondReceive* is a function which returns the oldest message in case the mailbox is not empty. Hence *CondReceive* can be used for polling. A received message is

placed in the receiving environment's *message-pouch*, a designated pair of positions in the environment's primary capability list. *CondSend* and *CondReceive* will return an error code if the mailbox overflows (is full) or underflows (is empty), respectively.

The second level, MSG-2, extends the set of message transmission functions to provide a synchronization as well as a communication mechanism. MSG-2 is relies on the hierarchy above the MPX level where the notion of blocked environments was introduced. MSG-2 provides the unconditional message functions: *Send*, *Receive*, and *Reply*. *Send* performs the same tasks as *CondSend*; except when the target mailbox is full, *Send* will cause the sending environment to be blocked awaiting an opportunity to deliver its message. Likewise, the *Receive* function causes the environment attempting to *Receive* a message from an empty mailbox to become blocked. *Send*ing a message to an empty mailbox on which an environment is waiting will cause that environment to *Receive* the message and become unblocked. Similarly, if *Receive* causes a full mailbox to no longer be full, it will awaken the oldest environment awaiting to deposit a message.

MSG-2 also defines a *Reply* function for mailboxes. This function differs from *Send* in that after executing the *Reply* function on a mailbox as permitted by a capability for that mailbox, the right to *Reply* to that mailbox is removed from the capability.

The two levels of the message transmission module provide an excellent example of a decomposition of a single module. MSG-1 defines both message and mailbox data structures, but provides functions which are of limited applicability; in some situations the functions fail returning an error code. Conditional functions are used to transmit messages in a well-defined fashion, but do not perform synchronization.

MSG-2 extends the definition of the mailbox data structure so that waiting environments can be recorded when necessary. It also provides new functions extending the usefulness of mailboxes, but not 'covering up' or subsuming the conditional functions which are useful when polling is desired. The multiplexing module relies on the conditional message functions of MSG-1 and implements blocking and unblocking on which the second level of MSG depends.

Dispatching and Multiplexing

Dispatching (DSP) and Multiplexing (MPX) are both levels and entire modules. DSP defines the hardware implemented state vector and its associated *Envload* function which loads an environment onto a computer module and begins execution. *Envload* is implemented in a combination of Kmap microcode and software. Software portions of *Envload* locate the process register values and the processor status word values in the state vector and load them into the physical processor registers. The software then stores the index of its capability for the environment in a special location which alerts the Kmap that an *Envload* is in progress. The Kmap portion of this function loads appropriate values found in the state vector into the window registers and various Slocal registers.

Functions in DSP are used exclusively by the multiplexing module (MPX) which is responsible for selecting the next environment to be *Envload*ed. Module MPX defines a set of *Runqueues*, each of which is a mailbox. If an environment is eligible for execution, i.e. it is not blocked nor already executing on some processor, then there is a message containing a capability for it in one of the runqueues.

Associated with each processor is an ordered list of at least some of the runqueues. The ordering selects the priority with which that processor services the mailboxes. The same Runqueue may appear in various positions in the ordered list of runqueues of different processors. The *Multiplex* function, invoked by the superior levels ME-2 and TI, cycles down the list of runqueues (private to the processor executing *Multiplex*) performing *CondReceives* on the runqueues. If the *CondReceive* is successful, then the result is a capability for the next environment to be *Envload*ed on the executing processor.

Trap and Interrupt Handling

Software traps and interrupts signal exceptional conditions caused by program action and external asynchronous events, respectively. With only a few exceptions (e.g. responding to a clock interrupt or to a high speed device interrupt), hardware traps and interrupts are translated into software traps and interrupts, so that modules can indicate what action is to be taken when they occur.

Defining a new trap (interrupt) means defining a new *trap (interrupt) vector entry* indicating what funtion in what module is to be invoked if the trap (interrupt) occurs. When a trap occurs, it was caused by the executing environment, so a *Call* is perfomed to suspend the current environment and cause the function named in the appropriate trap vector entry to be executed.

Interrupts are asynchronous and are not necessarily related to the current processor execution. TI offers two options. As a result of an interrupt a *Fork* can be performed to the function named in the associated interrupt vector. This will cause the interrupt to be serviced in parallel with execution of other environments. Alternatively, an interrupt vector or trap vector entry may direct that as a result of an interrupt, status information be sent as a message to a specified mailbox. Presumably some environment capable of handling the interrupt will *Receive* or *CondReceive* to get the

message. Interrupts would then be processed sequentially by order of occurence.

Two observaions are appropriate here. On is that using the trap and interrupt mechanism, any level above TI can define vector entries so that code from higher levels can respond to exceptional conditions encountered when code from lower levels is executing. This effects 'outward calls' so that lower levels can rely on higher levels when exceptional conditions arise. The second observation is that the trap and interrupt module is quite small, relying heavily on ME for *Fork* and *Call*, and on MSG for mailboxes.

The Kernel System

The Cm* architecture provides alternative ways to implement functions. A function may be implemented in Kmap microcode, or it may be implemented in software to be executed by one or more of the computer modules. A computer module may execute a function in either of two address spaces (user or kernel space). The decision where to place a particular function of a particular level of

a particular module is determined by considerations such as maximizing performance, providing for proper synchronization, and ease of implementation, as well as maintaining protection boundaries between module. Because of this independence between the design and the physical realization, alternative implementations of a function are possible. This facility is expected to be valuable in a system designed for experimental use because it allows for function substitution and redesign.

The kernel software system described here is implemented in two parts: Kmap microcode and a set of programs which run in the kernel space of the computer module processors. It is intended that in the initial system all of the capability functions and message functions will be performed by Kmap microcode. The remaining functions will be implemented in software to be executed from the kernel space of the computer modules.

The kernel and user spaces have symmetric data structures because both are executing environments. Both the user and the kernel system have a primary capability list which acts as a 'root' for capability addressing purposes. Both primary capability lists include a capability for a state vector and for a module capability list. It is the primary capability list and the state vector of the kerenel space that maintain information particular to a processor. Shared data and code in the kernel are referenced via capabilities in the kernel's module capability list.

Status of Software Development

As of December 1976, the microcode available provided only for simple relocation of physical addresses with no capability referencing. Development of microcode to support capability operations and the message facility will follow shortly.

Kernel space programs have been coded in BLISS-11 [Wulf et al. 71], a system implementation language. This set of programs is being tested using a simulator for the Cm* machine [Chansler 76] which executes on C.mmp, another multiprocessor system developed at Carnegie-Mellon University [Wulf et al. 74]. The simulator models multiple computer modules as multiple processes, and is able to run at about half the speed of a Cm* processor by exploiting the writable control store features of the C.mmp multiprocessor. Since the kernel code is successfully executing on the simulator, it is expected that the software kernel will be available for use shortly after the completion of the Kmap microcoding.

Future Software Development

The kernel system modules as described constitute a very primitive system. A number of additional software levels and new modules are in various stages of design. It is expected that most of the levels in these modules will be implemented as programs in the user space. Modules under development include:

Secondary Store Management--Current design proposes adding some disk memory local to some clusters, with large file storage accessible via a high speed link to either the C.mmp or the DEC KL-10.

Linkediting--The creation and management of modules as Cm* modules will be performed by a linkeditor intended to simplify the construction and management of function tables, segments of code, and invocation sequences.

Command Interpreter--This module will provide on-line, interactive access to the Cm* machine. This will allow a programmer to dynamically manage a task force. Currently interactive terminal communication is provided by a PDP-11 connected to each computer module by a serial line unit [Van Zoeren 76].

ALGOL 68 Runtime System--The first programming system to be available on the Cm* machine is expected to be ALGOL 68. (Until such a system is available, code will be cross-compiled on another machine). This version of ALGOL 68 will be designed to exploit the multiprocessing facilities of the Cm* machine.

Resource Policy Modules--A task force requires many runtime decisions concerning scheduling and resource allocation. It is the task of a policy module to provide for

these decisions based up on the dynamic state of the task force and the Cm* machine as a whole.

Acknowledgements

The Cm* software design was strongly influenced by the ideas which emerged from the family of operating system project (reported in [Habermann et al. 76]) and the virtual memory project which predated the family of operating system effort (reported in [Parnas et al. 73]).

In addition we would like to thank three individuals, Richard Swan, Victor Lesser, and Lee Cooprider for their comments and suggestions.

Summary

This paper represents a status report on the design of the firmware and software for management of a distributed multiprocessor called Cm* and the software construction philosophy which influenced its design. We have described the lowest levels of the kernel; some of the microcode and all of the software implementing what we have described now exists.

Besides continuing with the design and implementation of further levels of software, we intend to experiment with the placement and execution of kernel code within different Cm* configurations. Parameters of these experiments will include varying the physical location of the kernel code, the number of copies of that code as well as which computer modules can execute different portions of the code.

For example, one experiment is to limit the number of processors that can execute ME-2 code to (say) two processors in a cluster. If user programs executing on processors other than the designated two request ME-2 functions, their requests will be recorded so that the designated two processors can process these requests at some later time. The motivation for such an arrangement is that a processor is much more efficient if it executes code from its local memory.

In addition to such operating system experiments, we plan a number of experiments employing Cm* in the solution of different types of applications problems.

References

[Bell and Newell 71] Bell, C. Gordon and Allen Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[Chansler 76] Chansler, R. J., "Cm* Simulator Users' Manual", Department of Computer Science, Carnegie-Mellon University, 1976.

[Dennis and Van Horn 68] Dennis, J. B. and E. C. Van Horn, "Programming Semantics for Multi-programmed Computations", *Communications of the ACM 11*,3 (March 1968).

[Habermann et al. 76] Habermann, A. N., Lawrence Flon and Lee Cooprider, "Modularization and Hierarchy in a Family of Operating Systems", *Communications of the ACM 19*,4 (April 1976).

[Lampson 69] Lampson, B. W., "Dynamic Protection Structures", *Proc. AFIPS 1969 FJCC 35*, AFIPS Press, Montvale, N. J., (1969).

[Liskov 74] Liskov, B. and Steven Zilles, "Programming with Abstract Data types", *SIGPLAN Notices 9*,4 (April 1974).

[Parnas 72] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM 15*,12 (Dec. 1972).

[Parnas et al. 73] Parnas, D. L., and W. R. Price, "The Design of the Virtual Memory Aspects of a Virtual Machine", Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, 1973.

[Swan et al. 77a] Swan, R. J., S. H. Fuller and D. P. Siewiorek, "Cm*: a Modular, Multi-Microprocessor", submitted to the 1977 National Computer Conference.

[Swan et al. 77b] Swan, R. J., A. Bechtolsheim, K. Lai and J. Ousterhout, "The Implementation of the Cm* Multi-Microprocessor", submitted to the 1977 National Computer Conference.

[Van Zoeren 75] Van Zoeren, H. "Cm* Host User's Manual", Department of Computer Science, Carnegie-Mellon University, December 1975.

[Wulf et al. 71] Wulf, W., et al., "Bliss: A Language for Systems Programming", *Communications of the ACM 14*, 12 (Dec. 1971).

[Wulf et al. 74] Wulf, W., et al., "HYDRA: the Kernel of a Multiprocessor Operating System", *Communications of the ACM 17*, 6 (June 1974).