A SHARED MEMORY MULTI-MICROPROCESSOR SYSTEM WITH HARDWARE SUPPORTED MESSAGE PASSING MECHANISMS

By
LAM Chin Hung

A THESIS

Submitted to
The Chinese University of Hong Kong
in partial fulfillment of the requirements
for the degree of

MASTER OF PHILOSOPHY

Department of Computer Science

May 1990

ABSTRACT

Although message-passing is an elegant communication paradigm highly recognized in areas such as object-oriented systems, parallel and distributed processing systems, it is unfortunately not as efficient as the shared-memory paradigm when implemented on bus-based multiprocessors. Since building distributed global-memory systems around a common bus is a very simple, flexible and economical way to taste the advantages of multiprocessing, a design that combines the strong aspects of the two paradigms will help popularize parallel processing.

We have taken a hardware approach to alleviate the problem. On top of a shared-memory architecture, message-passing is supported by hardware. A dedicated processor called the **Message-Passing Coordinator (MPC)**, which is a value-added switch box, manages the message traffics in the system. While point-to-point messages are handled in the form of DMA transfers, broadcasted messages should be implemented in a way that can utilize the intrinsic characteristics of a shared-bus. Thus, a 1-to-N DMA mechanism is introduced which is a very effective way of handling broadcast messages on similar architectures. The workstation introduced above is called **SM3**. It is suitable for concurrent program development, distributed problem solving, and other computation bounded jobs.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude and appreciation to my supervisor Mr. K. H. Lee, who gave me invaluable advice and showed the greatest patience. The gentle assistance from the staff of the Microprocessor Laboratory are indispensable.

I also wish to thank Motorola Semi-conductor (HK) Ltd. for providing major chips and helpful consultation, which makes this project possible.

The author is grateful to the Sir Edward Youde Memorial Fund Committee for their appreciation and financial assistance.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 Gaining performance with multiprocessing

Data processing power is in great demand in this information age. For instance, the enormous amount of data sent back to the earth from space crafts has to be stored on tapes because the computation power currently available cannot process them immediately. It will take years to analyze the stored data. For the example given in [Ware72], the intensive computation required to process images from the spacecraft Mariner VI and VII was shown to be too demanding for the computers available then. Unfortunately, the problem is getting worse.

Since any computer system consists of hardware and software, the performance problem can be tackled in two different ways. By **software method**, we mean to develop a better algorithm and to improve the coding manifestation of a good algorithm. By **hardware method**, we refer to the improvement in the component technologies or the parallel architecture where more operation units are incorporated. Figure 1.1 summarizes this picture.

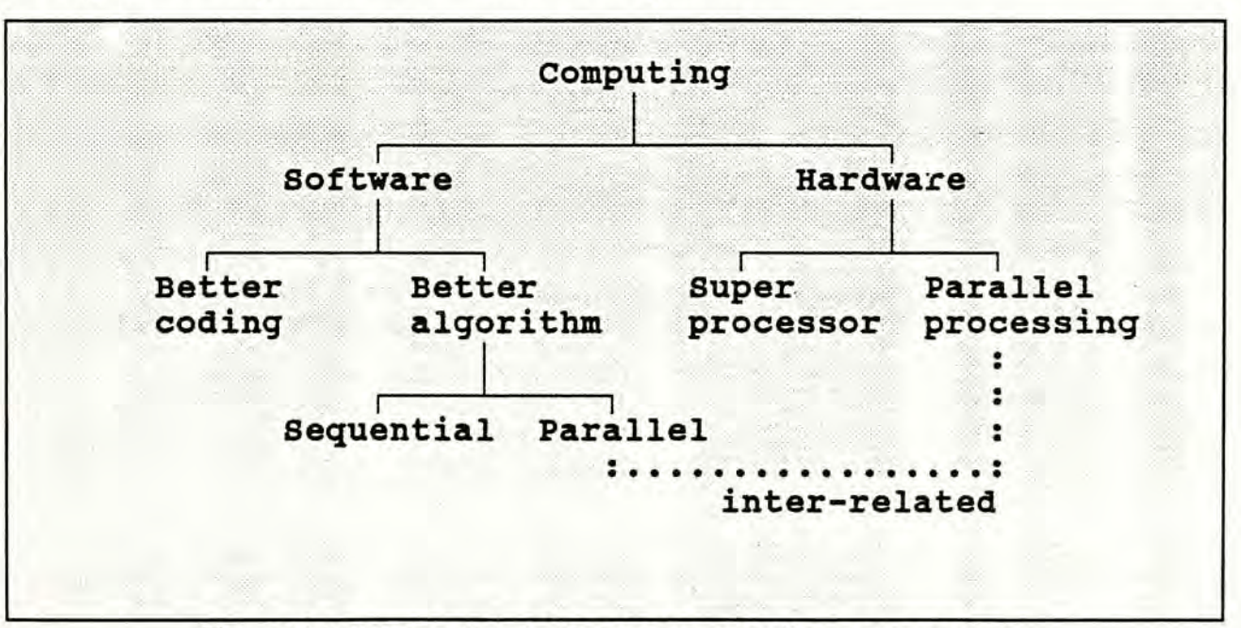


Figure 1.1 Approaches to improve computing performance.

1.1.1 Software approach

To speed up a computation task, one can devise a faster algorithm. But unlike the micro-electronics technology, the discovery of an algorithm with smaller time complexity for a problem is somewhat unpredictable. For certain problems, we do not even know whether we have already found the quickest algorithm or not. Moreover, the behavior of algorithms is usually data dependent. As a result, software efforts may be unrewarding during some occasions.

Coding skill is another important factor. Experienced programmers can devise clever tricks and short cuts to gain speed. Unfortunately, the clarity of the program is completely upset for incremental speed up. The state-of-the-art software engineering principles stress elegance, reliability, and portability because of the constantly rising software complexity and labor cost. The declination of hardware cost has diminished the effect of small reductions in computation time and code size. Thus, it is very difficult to multiply performance of a conventional uni-processor computer by solely refining the software.

Parallel algorithms have revived the study of fast software. However, a truly concurrent architecture is mandatory in order to benefit from them (figure 1.1). Since this thesis will not focus on software aspects, we shall postpone our discussion on software here and come back to it when we discuss the performance and applications of the proposed architecture.

1.1.2 hardware approach

Of course, another way to speed up computations is to work on the hardware. Two independent approaches are available. The first one is to build **super-processors**. Such machines usually carry some of the following attributes:

- very fast logic circuits (high clock rate)

- wide data path with high band width
- highly optimized, overlapped internal operations
- good vector processing power
- highly compact package

Although more delightful attributes can be appended to the above list as more and more surprisingly powerful processors appear, we are not so optimistic about the future of this direction of endeavors as the physical limitations are within sight. Physicists pointed out that there is an ultimate limit to the speed at which any component can operate. For example, it is impossible to eliminate the delay due to the time required to charge the intrinsic capacitance of a transistor using the finite current available. Even new technologies, such as Gallium Arsenide (GaAs), have their limitations [Heard84]. Actually, no data processing system can process information faster than C^2/h (ie. $2x10^{47}$ bits per second-gram), where h is Planck's constant [Ware72]. Obviously, the speed of light is a very tight limit.

Apart form the processing device, the memory also suffers from severe limitations. The following example is given by [Landa61]. To store binary information, we need a device that has two potential wells (stable states) separated by a barrier. Energy must be inserted in order to change the state of the device, and be removed when the new state is reached. [Marko65] shows that the uncertainty principle imposes a time limit of about 10^{-15}s to inject energy into or remove energy from any information storing device. Moreover, energy is inevitably dissipated as heat, but dense packaging is inconsistent with heat dissipation.

The current technology can produce VLSI chips that have certain parameters within an order of magnitude away from the physical limit. For instance, the internal power density of a p-n junction is within a factor of ten from the maximum cooling rate at room temperature [Ware72].

Over 99 percent of computer arithmetics are implemented in the form of binary transistor logic. Contemporary machines can perform additions in 60-80%, multiplications in 25%-30%, of the Winograd time which is also a theoretical lower bound [Winog65, Winog67]. This limit gives the minimum time required to perform an operation on two numbers. It varies with the number length, radix, fan-in and delay of logic elements. The only way to break this limit is to precalculate all results and use the table look up method.

It is evident that to get more computing power out of a single processing device will become quite difficult as we are approaching the physical limits. To cope with the rapid growth of data volume, we have to exploit other ways. **Parallel processing** is the other hardware approach we are going to discuss in detail.

1.2 Parallel processing

A natural way to overcome the speed limitation of any physical devices is to arrange a large number of them to work concurrently. This idea is not restricted to the central processor. Parallelism can be applied at **program-level, unit-level** (multiple memory modules, peripheral devices), and even **instruction-level** (multiple ALUs and other functional units). Although virtually there is no limit to the degree of parallelism, but [EliMo83] shows that the throughput of a parallel computer system will actually saturate at some point due to the contention problem and coordination overhead. Figure 1.2 depicts this.

The seriousness of the saturation effect depends strongly on the particular application. A number of physically implemented systems, although limited to a small set of applications, can relieve this problem so well that actually they do not degrade the performance significantly. An example system BBN Butterfly consisting of 256 processors is shown to give close to linear speedup in [RetTh86].

According to the degree and nature of parallelism, computers can be classified

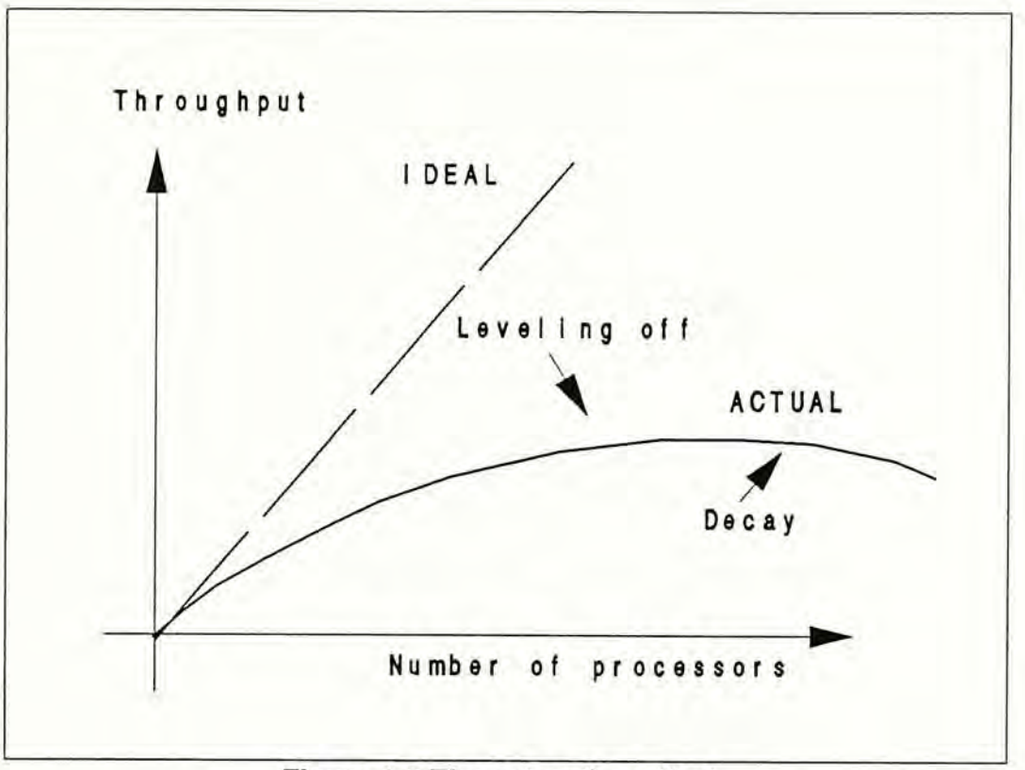


Figure 1.2 The saturation effect.

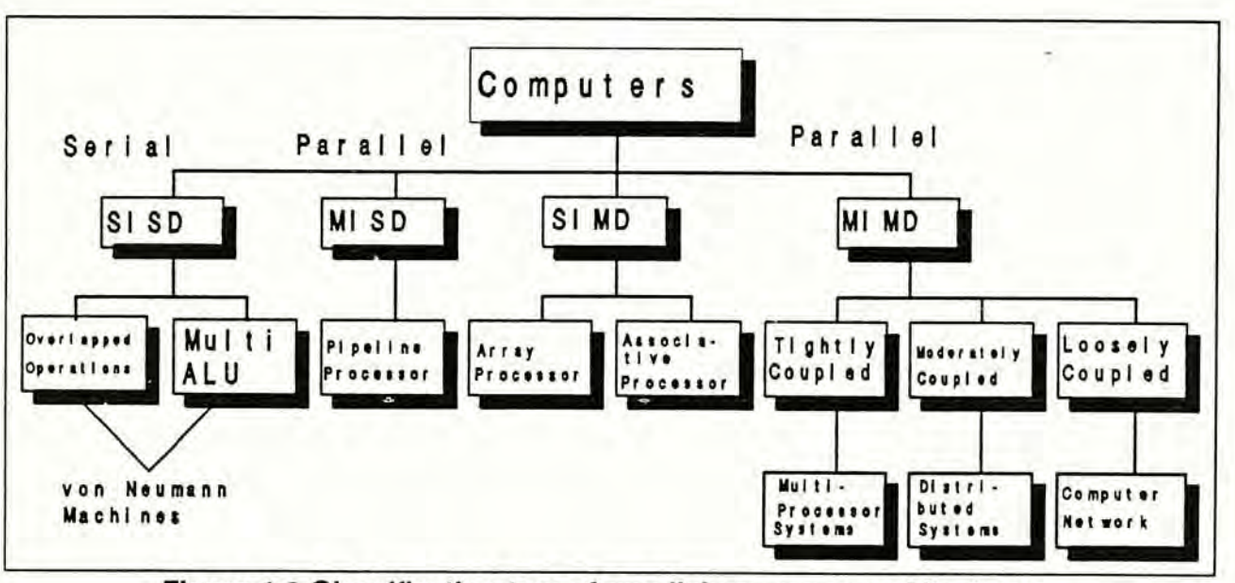


Figure 1.3 Classification tree of parallel processor architectures.

hierarchically as shown in figure 1.3 [FatKr83]. Here 'S' stands for Single, 'M' stands for multiple, 'I' stands for instruction stream, 'D' stands for data stream.

Though **SISD**s are marked serial, they still exploit different levels of parallelism. All contemporary computers, for instance VAX-11 and IBM 3090 series, have their peripheral devices arranged to operate concurrently with the CPU. Even in microprocessors such as the MC68030, instruction fetching, decoding, and execution are pipelined and performed in parallel. However, SISD computers

still conforms to von Neumann's model.

MISD computers are usually designed for a special application. For instance, in some encryption systems, a series of processing elements form a processor pipeline and they work on the same data stream consecutively.

Most **SIMD** computers are number crunchers. They are specialized in matrix or vector manipulations. Array processor is the most common form of SIMD computers. Due to their high cost and limited applications, they are not so popular. Illiac-IV is a well known example [GeRiM68].

MIMD computers can be general or special purpose. The degree of coupling between processors is application dependent and it determines how the processors should be linked. Communication capability is expressed in terms of transmission band width and latency of inter-processor links [Cleme88c]. Based on the degree of coupling, three major classes are identified:

a. In **Loosely-coupled system (LCS)**, processors communicate with each other by passing messages via physical channels provided by a local or wide area network. Most LCSs only have simple bit-serial links because of cost considerations. Inter-processor traffic must be kept light. Well known LCSs include the Cm* [JonSc79] and the Syte workstation [BruMi84], which consists of several processor modules based on NS16032s.

b. Distributed system may be classified as **moderately-coupled system (MCS)** but it must be stressed that there is no clear cut difference between this class and LCSs. It is claimed that since the communication between processors is more frequent, so each processing element is equipped with separated application and communication processors [FatKr83]. The author agrees that multi-microcomputer systems should be put under this category. In [Russo77], readers can find an example system made up of COSMAC microprocessors.

c. **Tightly-coupled systems (TCS)** have a common clock for all the processors. It is also refereed as **multiprocessor systems**. Due to the extensive interaction between processors, more complex hardware is needed. The Balance multiprocessor system [ShPaG88], which may have up to 30 processors, is a typical example. LCSs or MCSs can be constructed by linking TCSs together, where each multiprocessor is virtually a uni-processor. In other words, TCSs may serve as the building block of large scale LCSs or MCSs. Examples will be presented in chapter 2. The proposed workstation described here belongs to this class so we shall restrict our discussion to TCSs.

1.3 Gaining performance with multiprocessing

A true multiprocessor system incorporates two or more processors in the same housing, and the physical distance between the processors is important [Cleme88a]. The processors operate cooperatively on a logically coherent task. They communicate intimately with each other and share common resources. Since a given task can be carried out by several low-cost processors concurrently, relatively little additional cost is needed to increase the power of a multiprocessor. This favors incremental growth of the system. Other known advantages are the provision of graceful degradation and fault tolerance.

1.3.1 Multiprocessor configurations

Multiprocessor systems can be further subdivided into two classes of configuration known as **processor-to-memory** and **processing-element-to-processing-element (PE-PE)**. Figure 1.4 illustrates their differences [Cleme88a].

a. Processor-to-memory:

As shown figure 1.4a, an interconnection network bridges M processors and N memory modules. Processors communicate by sharing memory modules. Omega, Butterfly, and cross-bar are typical interconnection networks. An introduction to interconnection networks can be found in [Feng81]. For performance impro-

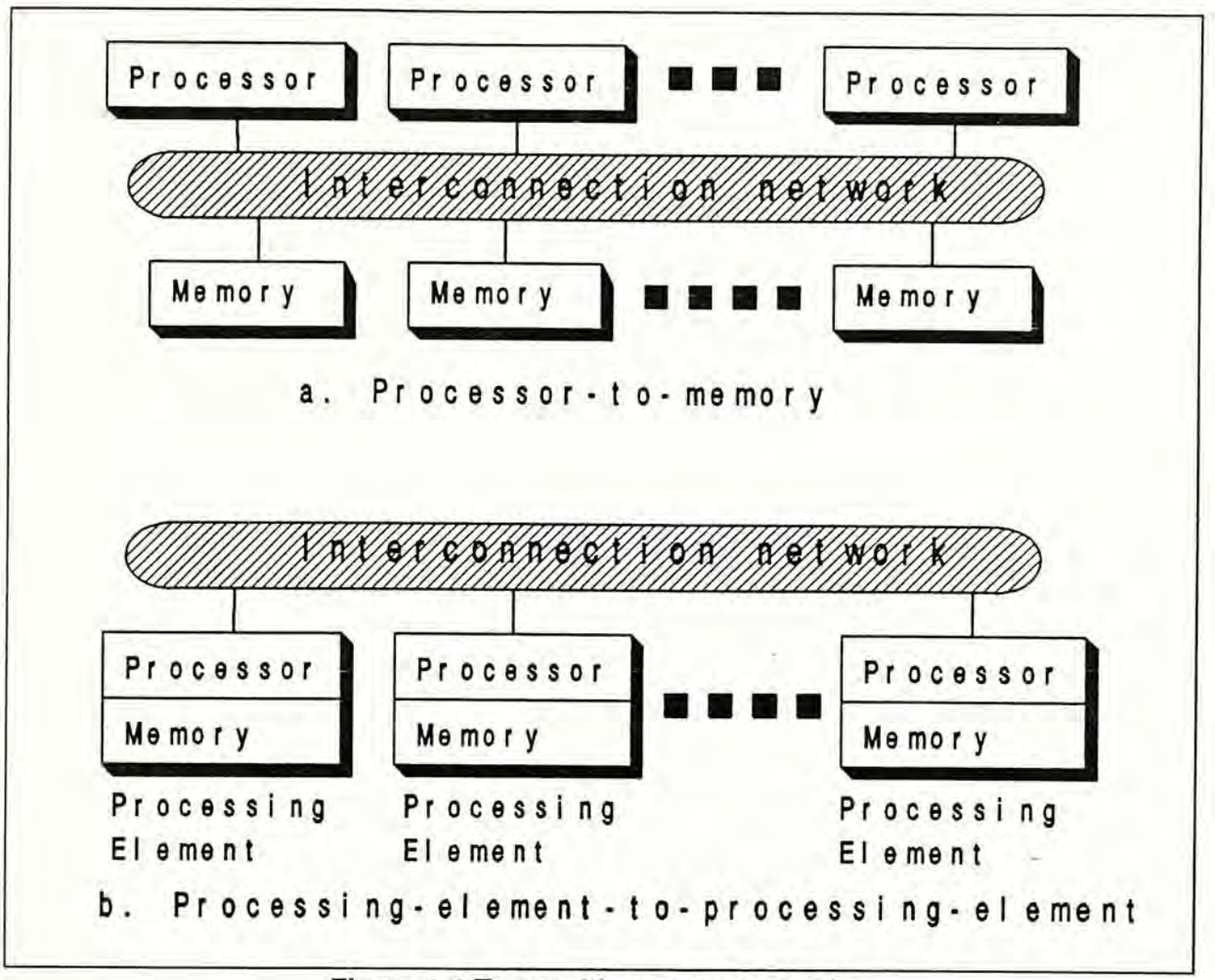


Figure 1.4 Two multiprocessor models.

vement, processors may have cache memory. The success of this configuration gears to the band width of the interconnection network and/or cache performance. The simplest form of the interconnection network is a **time-shared bus**, where requests from two or more processors are time-division multiplexed onto a single bus. Buses may be synchronous or asynchronous.

Asynchronous bus may have centralized bus control/arbitration, where one processor is the master, or distributed bus control, which requires extra hardware and software to resolve the contention towards a consensus. Bus users can request for the mastership of the bus at any time and the right of use is granted to it if the bus is free and there is no other competitors. Asynchronous bus is suitable for systems with a very irregular bus request pattern.

On a **synchronous bus** system, every processor can take control of the bus whenever it requires, like the asynchronous one, independent of the other pro-

cessors. A system clock on the bus synchronizes all the bus operations. To improve availability, the bus may be time shared evenly. A common way to provide each processor a time slot for this purpose is to shift the phase between the clocks of the processors. However, band width is wasted when free time slots are not fully utilized. More examples and information can be found in [Labib88].

b. PE-to-PE:

Every PE consists of a processor and a private memory module as figure 1.4b shows. Each PE is nearly a complete computer except they don't have local peripheral devices (that distinguishes them from multi-computer systems). An interconnection network provides the PEs with a communication media. Information is exchanged in the form of messages. RIMMS described in [LeDaR84] is a typical example.

In most cases the PEs are not allowed to access directly the memory of other PEs, such a configuration is somewhat similar to the MCS described above. However, we have chosen an alternative that is also taken by a number of designers. The local memory of each PE is also accessible from other PEs. In other words, the memory modules form a **distributed global memory**. Actually this is generally not realizable for MCSs and LCSs, so it is an identification character of TCSs. Figure 1.5 summarizes the variations in the TCS.

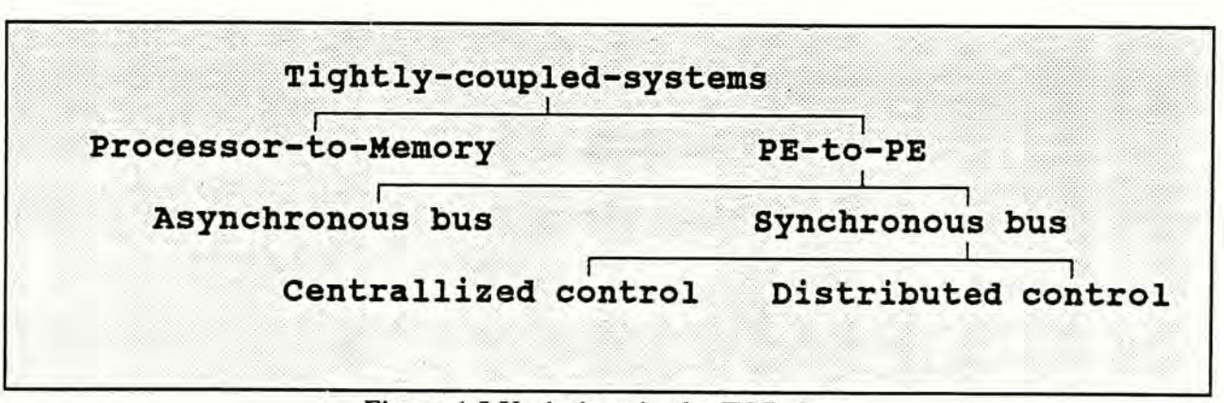


Figure 1.5 Variations in the TCS class.

A combination of the two models is possible. [FiJoS83] describes a system with both private (means 'local' and 'unshared') and shared memory. Although the system looks complicated, the high flexibility it offers is surely an advantage. There are still many other techniques for inter-processor communication. On a system mentioned in [Cleme88c] processors are coupled by multi-port memory. An even more interesting system shown in [HeMaN88] uses video-RAM to link up processors. In these systems the latency is rather small due to the ease of control and arbitration. But painfully, extensibility and compatibility with standard devices are traded.

1.3.2 Multiprocessor design issues

After we had summarized general parallel processing styles, let us focus on multiprocessor systems. The importance of this class calls for more attention. This is one of the most common trends of parallel processing because it may be the building block of larger systems. They are suitable for general purpose as well as special purpose computing.

Several common design issues of multiprocessors (TCSs) are presented below. Most of them may be applied to MCSs and LCSs too. Readers may refer to [Cleme88a] for more details.

a. Metastability

A general purpose machine with high flexibility requires a memory such that every word is accessible to every processor at any time, with access time comparable to that of an unshared memory. Usually an arbiter is responsible for the scheduling of the competing memory accesses. Communication between the processors via the shared memory must be reliable. Typically, a hardware semaphore is used.

However, there is a finite possibility that a processor may request access at the exact moment that the arbiter is making a decision. The state of the system is

undefined in this cases. This phenomenon is known as the **metastability** problem, which is identified as a soft failure [Cleme88c]. The machine will not be brought down if it is robust enough.

b. Distribution of tasks

The distribution of tasks in a multiprocessor system is strongly determined by the nature of the particular application. An effective multiprocessor must be able to allocate resources to contending processors without seriously degrading the overall performance of the system. This is the duty of and also the challenge to the multiprocessor operating system designer.

c. Interconnection topology of the processors

The success of a multiprocessor system is closely geared to the effectiveness of the interconnection topology. Designer must consider the cost, band width, and reconfigurability of the network. Usually tradeoffs have to be made under the constraints imposed by an application.

d. Management of the memory resources

Design decisions on the control/arbitration logic, security measures, mode of memory access and cache organization are critical issues. Things will be much more complicated when virtual memory and multi-programming are supported.

e. Avoidance of deadlock

Deadlock may occur at different levels: from high level inter-processor communication to physical signal protocols used for system synchronization and control. A priority system is probably the simplest, and also the preferred solution in many systems.

f. Control of input/output devices

Subject to cost considerations, it is usually not necessary and also not feasible to allocate peripheral devices to every processor node. The question is who will control which device, and whether the operation of the devices will interfere

with the normal work flow of the system. For instance, a device that is directly connected to the shared bus may degrade the performance of the multiprocessor when the application program is IO-bounded. Sharing, allocation and security protection of devices are difficult and also crucial problems.

g. Choice of operating system

Master-slave type operating system is straight forward, simple and efficient if the vulnerability of the controller is not a problem. On the contrary, the distributed control type is much more complicated but robust.

1.3.3 Using microprocessors

An old problem with multiprocessor systems is the cost. Connecting a number of powerful processors to work in parallel is quite expensive if not prohibitive. The solution to this problem is related to the fact that the performance of a processor is not directly proportional to its cost [EliMo83], as figure 1.6 shows.

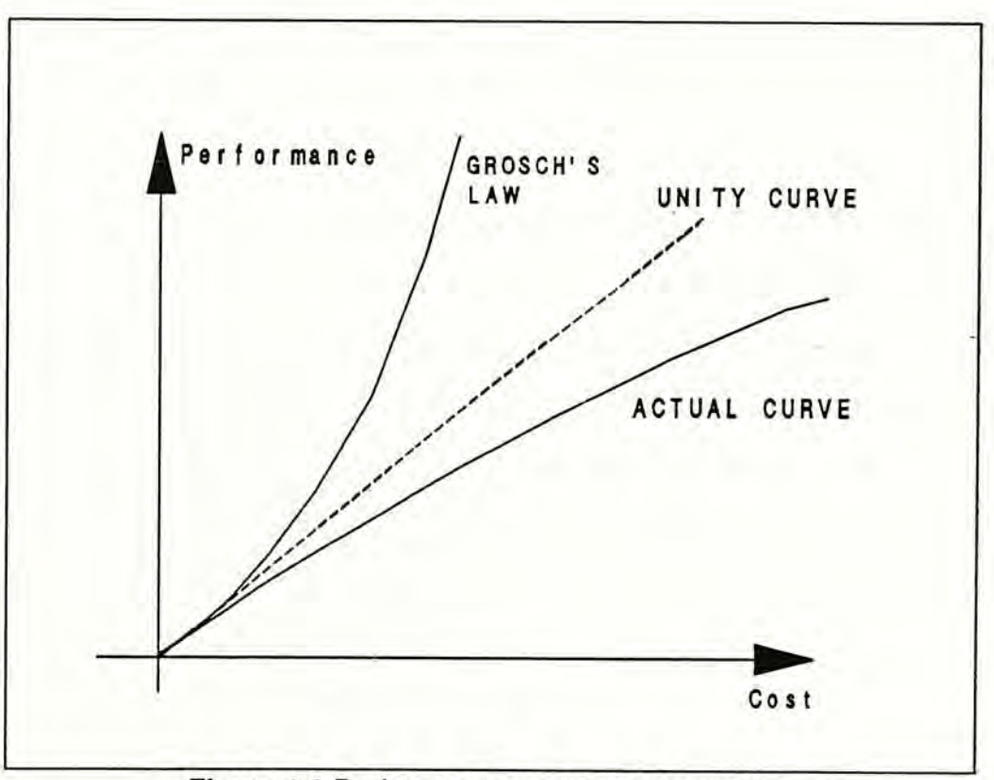


Figure 1.6 Performance versus system cost.

Grosch's Law [BauSe75] suggests that processor performance is proportional to the square of its cost. But this is no longer true now.

Nowadays, a costly processor does not outperform a cheap processor by the ratio of their cost, as the lowest curve in the figure indicates. That means the designer should use a large number of low-cost processors to obtain more processing power instead of using a small number of powerful and expensive ones. Due to the rapid development of VLSI technology, microprocessors are now so cheap that large scale multiprocessors are realizable. Some mass-produced standard microprocessors such as MC680x0, M88000, 80386, and NS32532 even have architectural support for multiprocessing [Tabak90]. They are the most basic building block of many existing multi-microprocessor systems.

1.3.4 Bus based systems

Recall that we have discussed the use of a shared bus as the simplest way of interconnecting PEs. The argument was made on the ground of cost and simplicity. This will be elaborated in the following paragraphs.

Undoubtedly, a shared bus is the cheapest way of interconnecting processors. The hardware requirement is minimal because there is nearly no active logic. For instance, multi-stage networks and cross-bar need switch boxes plus many wires. On the other hand, no special software, such as routing algorithm, is required for bus system. An important advantage is the ease of scaling up and down the system by varying the number of processors. Very few other network topologies provide this flexibility. Bus systems can simulate virtually all other topologies without much difficulties but the reverse is not true. It is an ideal choice for prototyping.

A well known disadvantage of a shared bus is the limitation of the band width. Communication intensive computations may face a bottle neck at the shared bus. The degree of parallelism can be seriously degraded. Another difficulty with

bus systems is that heterogeneous systems are hard to build. Interfacing various types of processor nodes and devices to a single bus by force may degrade the performance of the system significantly.

It is reasonable to assume processor nodes do not communicate with each other extensively, otherwise the problem should not be solved in a distributed way. A SIMD type computer may be employed instead. Apart from cost and simplicity considerations, fast prototyping is also a primary concern for many designers.

Even if the design do not have a heterogeneous architecture in mind, this expansion is easy at a higher level, eg. at the MCSs or LCSs level as we have mentioned in section 1.2. So the use of a bus architecture does not close the expansion path. Basically, the tightly-coupled nature of multiprocessor systems do not encourage heterogeneity.

1.4 Shared memory and message passing

In multiprocessor systems, user processes are spread over several processor nodes. Usually, they are working cooperatively to achieve a common goal. Information exchange is inevitable. The two most common paradigms for inter-process communication are called **message passing** and **memory-sharing**. Both paradigms are well applicable to uni-processor and multiprocessor systems.

1.4.1 Shared memory

The principle of this paradigm is simple. Two or more processes have access to a shared area, which may be as large as the whole memory space or as small as a single word, where communication data can be stored. This idea is roughly represented in figure 1.7a.

Depending on the particular application, some processes may have READ/WRITE access right to the area while others have READ access right

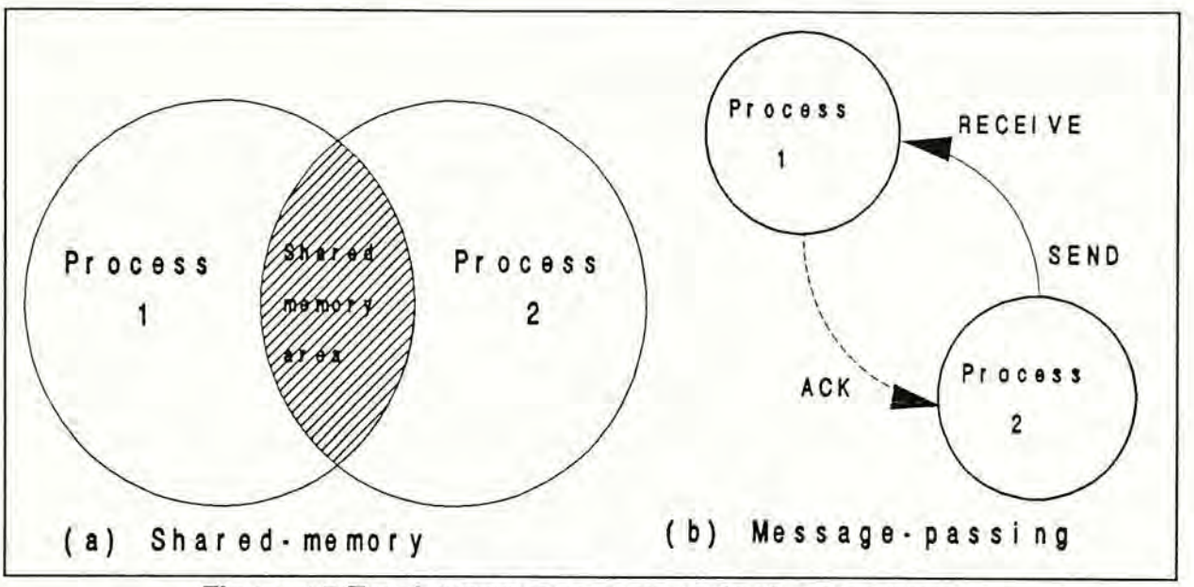


Figure 1.7 Two inter-process communication paradigms.

only. All the processes that manipulate or read from this shared memory area must observe a set of predefined rules about the data structure, how to flag the availability of valid data, locking of resources, and so on. The programmer must pay attention to all this matters. The effect caused by a faulty program may be catastrophic so the programmer's duty is quite heavy. If security must be enforced, special hardware and/or software mechanisms must be incorporated.

Note that so far we are talking about the communication between software processes. Ultimately, having a shared virtual memory space is enough for this level of communication, no matter it is a uni- or multi-processor system. Programmers are not required to know how such an environment is provided. The mailbox concept in multi-programming uni-processor systems is an example. In [Russo77], similar mailbox concept is applied to a shared memory multiprocessor system for inter-processor communication.

1.4.2 Message passing

The basic idea of message passing is no more complex than memory-sharing. Processes do not need to share an addressing space. They interact with each other by exchanging messages in the way figure 1.7b summaries. Such messages bear sender and receiver qualifiers. Normally, system calls are available for the

processes to send, receive, reject, and acknowledge messages. The sophistication of message systems can range from fixed length un-typed messages with single priority, to variable length messages from a hierarchical type structure with priority levels.

Message communication can be classified as **synchronous** and **asynchronous**. The former class implies that the message sender will be blocked (halted) until the receiver has acknowledged the message or the time out period expires. The sending of such messages also serves as a synchronization point of the sending process. Occasionally, the arrival of a message means much more than the data it carries. As a matter of fact this is a way of implementing synchronization primitives. Obviously, for this class of message systems the reservation of buffer area for a single message is enough. The communication logic is fairly simple too. However, parallelism is traded because the sender may be blocked frequently when it is coupled with a slow receiver, and vice versa.

Asynchronous Sending means the sender does not wait for the acknowledgement from the receiver before it goes to the next program statement. The data field of the message contains all the information to be conveyed. A message queue must be maintained by the system in order to keep track of the sequence of the messages, and to decouple a fast sender and a slow receiver. However, the time and memory space overhead of manipulating this queue must be considered.

Although this paradigm looks more sophisticated, the programmer's duty is even lighter than using shared memory since the burden of supporting message passing has shifted to the operating system. The programmer does not need to know how the messages are handled. In secure systems, user processes cannot access the memory spaces of other processes since message passing is the only way of communication. Hence, data encapsulation is easily enforced.

1.4.3 Comparisons of the two paradigms

This section tries to contrast the two paradigms. It is clear that none of them wins in all aspects. Table 1.1 lists the main points in our following discussion.

	Shared-memory	**Message-passing**
Speed	Higher	Lower
Hardware requirement	Memory controller	Serial links
Software requirement	None	Communication kernel
Programmer's model	Very Primitive	Bases on processes
Data encapsulation	None	Yes, ease to enforce
Level of abstraction	Lower	Higher
Portability	not so good	Good
Security enforcement	Difficult	Easy
Application examples	Real-time systems	Object-oriented systems

Table 1.1 Summary of the two paradigms.

Due to the fact of memory-sharing, the communication speed in such systems can be as fast as normal memory accesses in both multi- and uni-processor systems. On the contrary, message passing systems are usually slower owing to the extra message queue manipulation overhead. In uni-processor system, this is merely the effect of extra software housekeeping work. But in multiprocessor system, bit-serial links are predominantly used (for cost cutting) so off-board communication is an order of magnitude slower than intra-board access. The delay due to extra communication hardware logic and message buffering are also significant. Very few message systems can afford full inter-connection so delay caused by intermediate hops is great. Moreover, message passing is implemented at the subroutine level and it requires cooperation at the receiving end.

A point worthy of mentioning is that the memory access time for shared memory systems is very uniform. In contrast, the long delay of messages forces the programmer (or operating system) to adjust his task assignment strategy on message passing system. In a word, memory-sharing normally implies better speed.

However, we shall see later that this gap can be bridged.

A good memory controller is all it needs for a shared memory system. The arbitration and control logic inevitably takes time. It contributes to the saturation effect mentioned in section 1.2. Fortunately the price is inexpensive. Message passing systems requires physical links, associated control logic and software.

Concerning software requirements, shared memory systems require virtually no special provision as oppose to the need for a **communication kernel** in message passing systems. For message passing, the kernel is a set of communication primitives. Some useful primitives can be found in [Ng86]. According to [Gentl81], the semantics of primitives must be easy to understand, efficient to implement, encourage the execution of processes in parallel, and not error prone. The programmer interfaces with this kernel which is usually a part of the operating system. An example of such a kernel is introduced in section 2.5.

The programmer's model for memory-sharing is quite primitive. Most critical issues, such as security and consistency, must be addressed by the programmer. On the other hand, the idea of message passing is more elegant. The freedom of the programmer is limited but his duty is also lightened. In shared memory system, rarely any special software support is available so the program must handle all the details. Evidently, life is easier for a "message passing system" programmer.

Although the portability of concurrent programs has never been satisfactory, message passing as a vehicle for expressing interaction has a leading edge. Due to the versatility of this model, the only variable in the system is the ratio of local memory access time and non-local message delay. This parameter governs the process scheduling policy. Other issues, such as interconnection topology and connectivity, are handled by the system software. For example, if a process graph is to be mapped onto a grid array of transputers, a layer in the operating system

has to hide the fact that each transputer has only four bi-directional channels for communication. This situation is typical because cost and fan out limitations restrict direct linkages to neighboring nodes only.

On the contrary, programs for shared memory systems are hard to port due to their strong dependency on the particular hardware environment (address of the shared area, access protocol, etc.). In summary, we can say that the message passing paradigm has a higher level of abstraction.

An interesting point is that after eliminating all the hardware parameters, memory-sharing is so simple that it has become a good computation model for parallel algorithm study. This model is know as PRAM [Akl89]. A possible explanation is that porting programs for a conventional serial computer onto a shared memory machine is easier than restructuring them in the message passing paradigm.

For message passing, if the memory spaces of any two processes are strictly separated and isolated, then data encapsulation can be easily enforced. Security measures can be imposed because the communication kernel is a part of the operating system. Relatively, shared memory systems are more difficult to monitor and control.

Both paradigms are extensively used as idealized computation models. Message passing is quite suitable for object-oriented systems due to its data encapsulation and abstraction property. General issues and examples of object-oriented architectures can be found in [SiMiM86, WiLoE87, TasPl89]. A whole class of concurrent programming languages called **Communicating Sequential Processes (CSP)** [ShMiS78] employs this paradigm. Occam [Inmos83] is one example. In [AthSi88] a number of message passing systems are discussed. On the other side, existing examples of memory sharing include the Balance system [ThGiF88] and Encore system [Tabak90].

Along with the improvement of performance, it is now possible to build object-oriented systems, which employ message passing, for real-time applications too. The project pdvPOOL described in [TasPl89] is one of such new attempts.

1.5 Summary and comment

The above discussion focuses on qualitative aspects of the two paradigms. For computer architects, actual system performance is usually the primary concern. Both modelling and analytical studies have attracted wide attention. Since message passing systems vary significantly in complexity, and their performance depends strongly on the particular implementation, theoretical studies received less notice. In [Sangu86], the performance of a message-based multiprocessor is analyzed. It was shown that super-linear speed-up is possible for computation bounded workloads running in a mulit-programming environment. The treatment can be generalized to other message passing systems.

Theoretical studies of the shared memory architecture can be found in [Nader88a, Nader88b, BodLi89, Zhang88]. Naderi modelled shared memory multiprocessor systems with Markovian chains and queuing network. The resulting expressions are later generalized for systems that have multiple shared memory modules. Zhang discussed the effects that influence the performance of bus-based multiprocessors while Bodnar and Li analyzed the performance of such systems with a probabilistic, hierarchical model.

In this chapter, we figured out the background from which our proposed multi-microprocessor workstation emerges. Other approaches of gaining computation power are briefed. The rationale supporting the choice of a shared-bus architecture was discussed. The seemingly contradicting communication paradigms, message passing and memory-sharing, are introduced. Actually, these two paradigms are not exclusive, and they can even cooperate smoothly as we shall see later. Before the proposed machine is presented, we shall look at several typical machines in chapter 2.

CHAPTER 2
AN OVERVIEW OF COMMON APPROACHES

In this chapter, we shall examine others' research efforts related to this project. Due to the explosive increase of multiprocessor systems, this overview is by no means exhaustive. Emphasis will be placed on aspects that resemble or contrast with our approach.

2.1 SUPRENUM

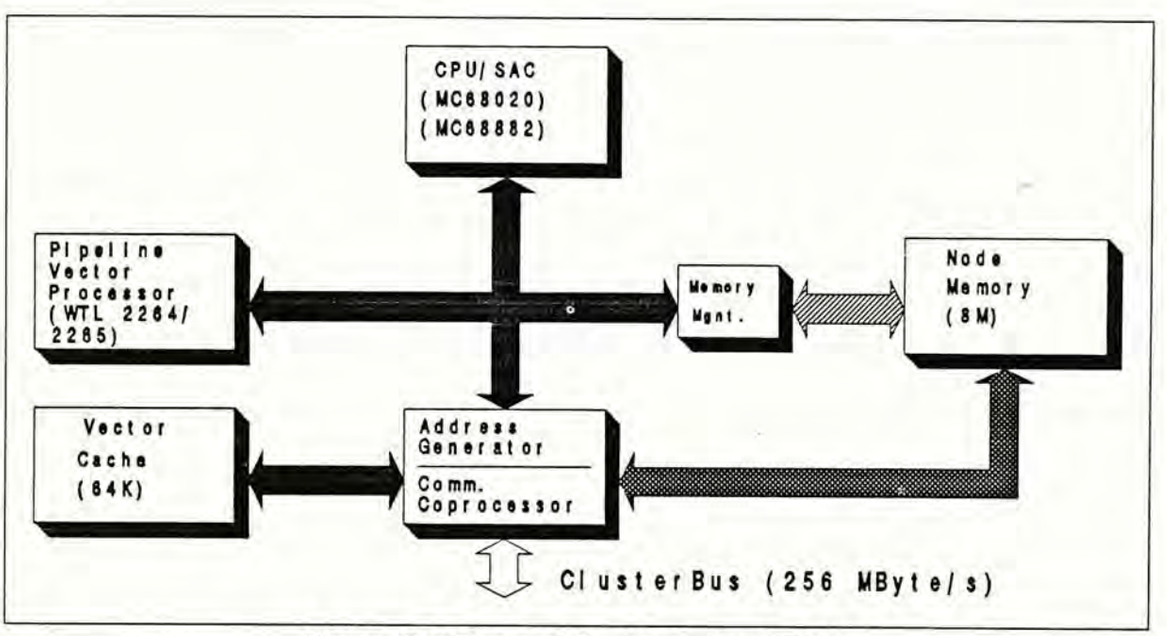


Figure 2.1 The architecture of a GPPN.

SUPRENUM is the German supercomputer project aiming at the development and construction of a distributed-memory multiprocessor system. The structure of a basic node, as the designer called **GPPN** (General Purpose Processor Node), is shown in figure 2.1. Maximally, 16 GPPNs connected to a shared bus form a cluster like the one in figure 2.2. At a higher level, 16 such clusters form a 4 by 4 grid as shown in figure 2.3. MC68020 is the heart of a GPPN. The communication processor is a dedicated to sending and receiving messages. The 256 nodes are connected via a 2-level interconnection network of buses. Despite of the fixed topology, the logical structure of the software processes is

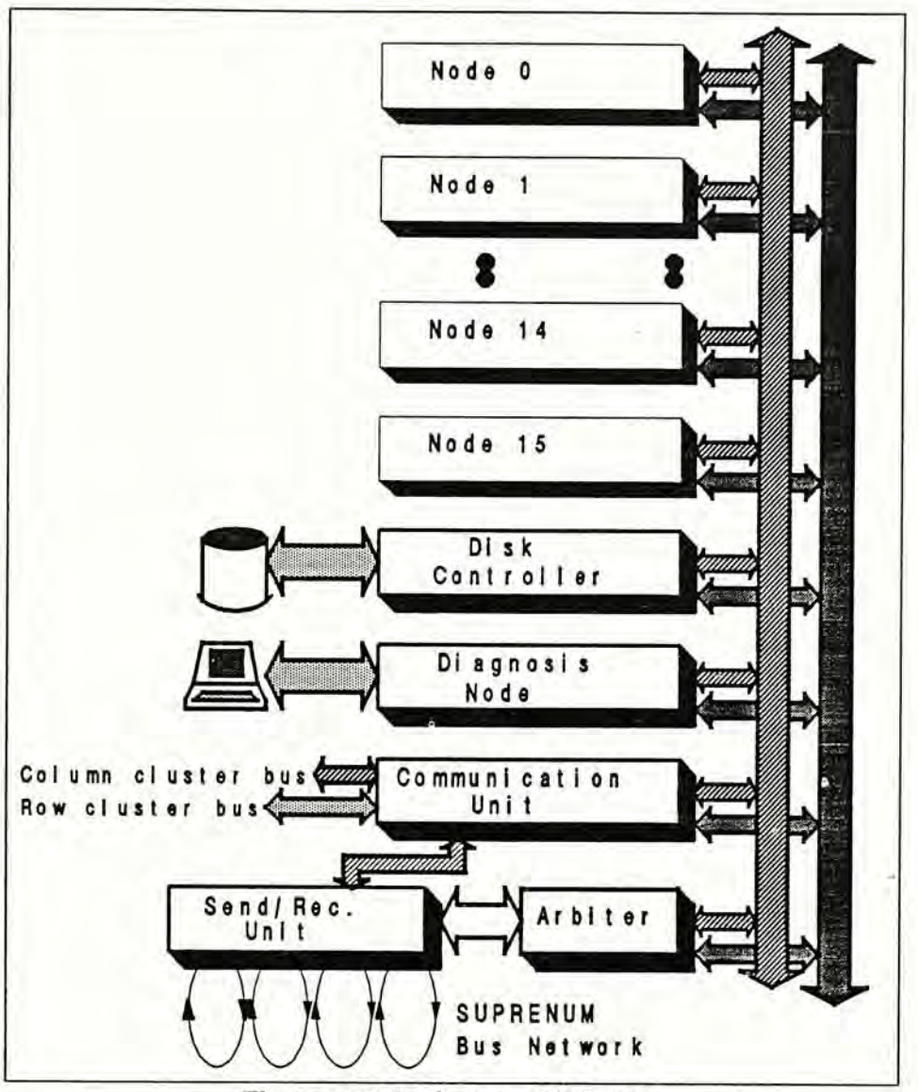


Figure 2.2 A cluster of GPPN's.

dynamically reconfigurable so the system can be configured to fit different computation structures. The mapping library provides optimal processes-to-processors mapping strategies for some standard process structures, and uses heuristic for unfamiliar irregular process structures.

This multiprocessor aims at numerical applications such as the simulation of fluid dynamics systems. It exploits coarse gain parallelism. The message-based operating system PEACE employs distributed control and provides load balancing. Unix V is chosen as the front end for interfacing with the user. The programming model bases on the process concept. For more details on the machine and its operating system, reader can consult [SchSo90, Giloi87].

2.2 MEMSY

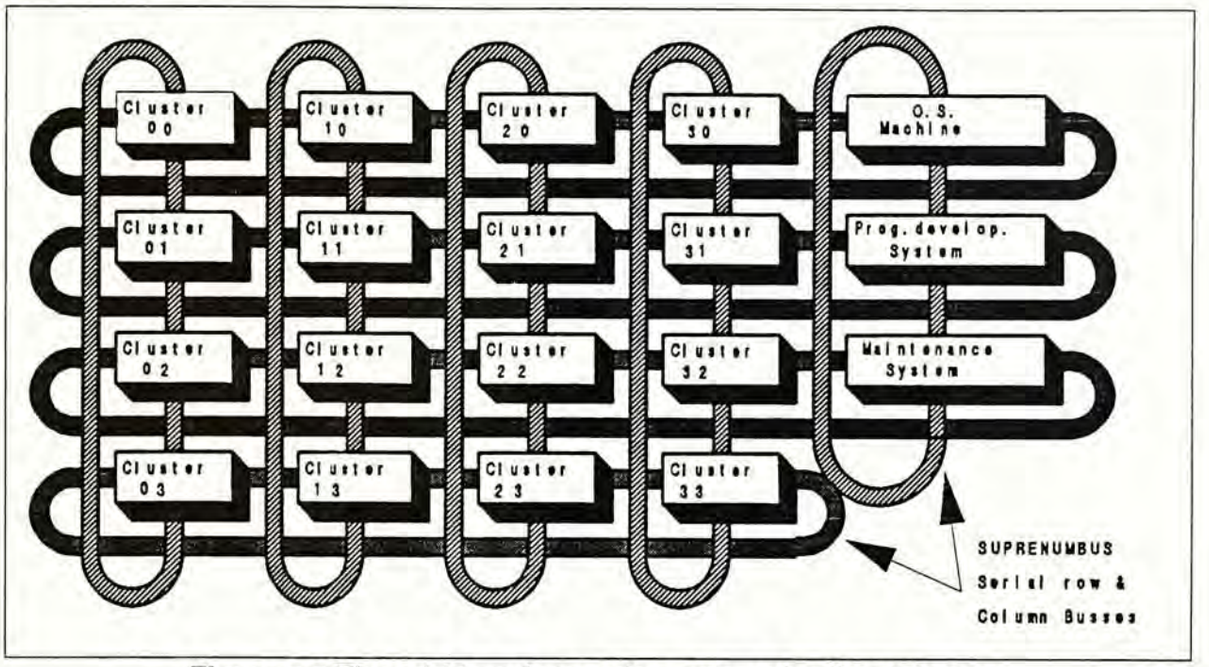


Figure 2.3 The grid topology using row and column buses.

MEMSY (Modular Expandable Multiprocessor System) [FrHeH90] is a very new MIMD multiprocessor with Distributed Shared Memory (DSM, similar to SM3's) organization. Since MEMSY aims at numerical simulation problems, it is designed to deal with the locality character of the physical problem. Data can be exchanged rapidly between two adjacent nodes.

Figure 2.4 depicts the structure of a processor-memory module (PMM). P is a commercially available microprocessor which performs inter-node- and I/O-communication. SP is a special processor, say i860 or 88100, dedicated to perform user's task. SP communicates with P via a dual-port memory SM. Standard and special software is stored in the local memory LM. The connection to the global bus will only be used at level B and C (figure 2.5).

PMMs are arranged in a 3-level pryramidal hierarchy as shown in figure 2.5. Level A (256, 1024,... nodes), which consists of worker-PMMs, computes the user problem while level B (64 or more nodes) accommodates the OS functions and performs I/O. Level C is a supervising PMM which connects the system to a host machine. At each level, the processors form a toroidally closed NN-system (Nearest Neighbor). Coupling between neighboring PMMs is realized by a multi-port memory so data exchange between adjacent PMMs is particularly fast. The

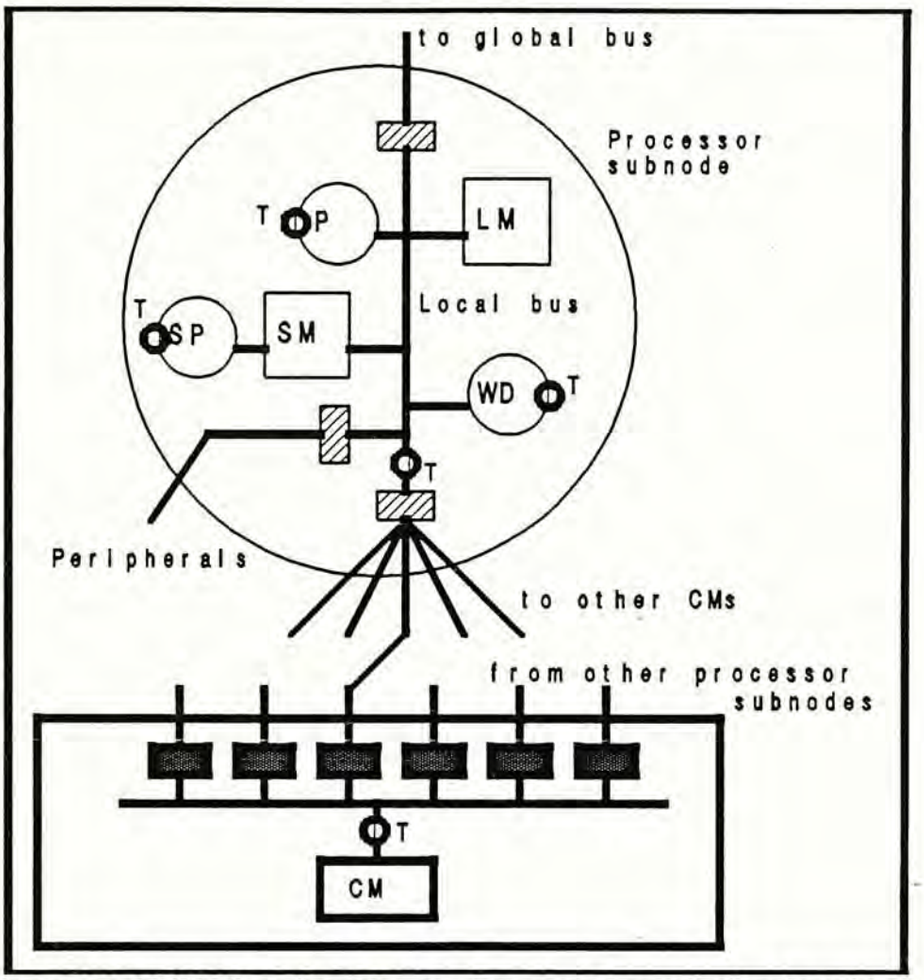


Figure 2.4 MEMSY node structure. T:test and monitoring interfaces.

top processor is connected to the B level PMMs by a common bus with broadcasting feature. An array of 32x32 worker-PMMs could achieve 20 GFLOPS peak. Watch dog processor WD is for fault diagnosis.

Three major features of this system are:

1. **Scalable** - a family concept allows for composing from small to large system. The interconnection network is regular and easily expandable.
2. **Distributed OS** - each node has its local OS kernel. The OS supports an object-oriented programming environment.
3. **Observability** - a hybrid monitor ZM4 helps to gain insight into parallelized execution of large jobs.

In summary, MEMSY is an example of hierarchical structure systems. It combines the advantages of using a shared bus and multi-port memory. Note that I/O operations are distributed to level B nodes, which is a common feature

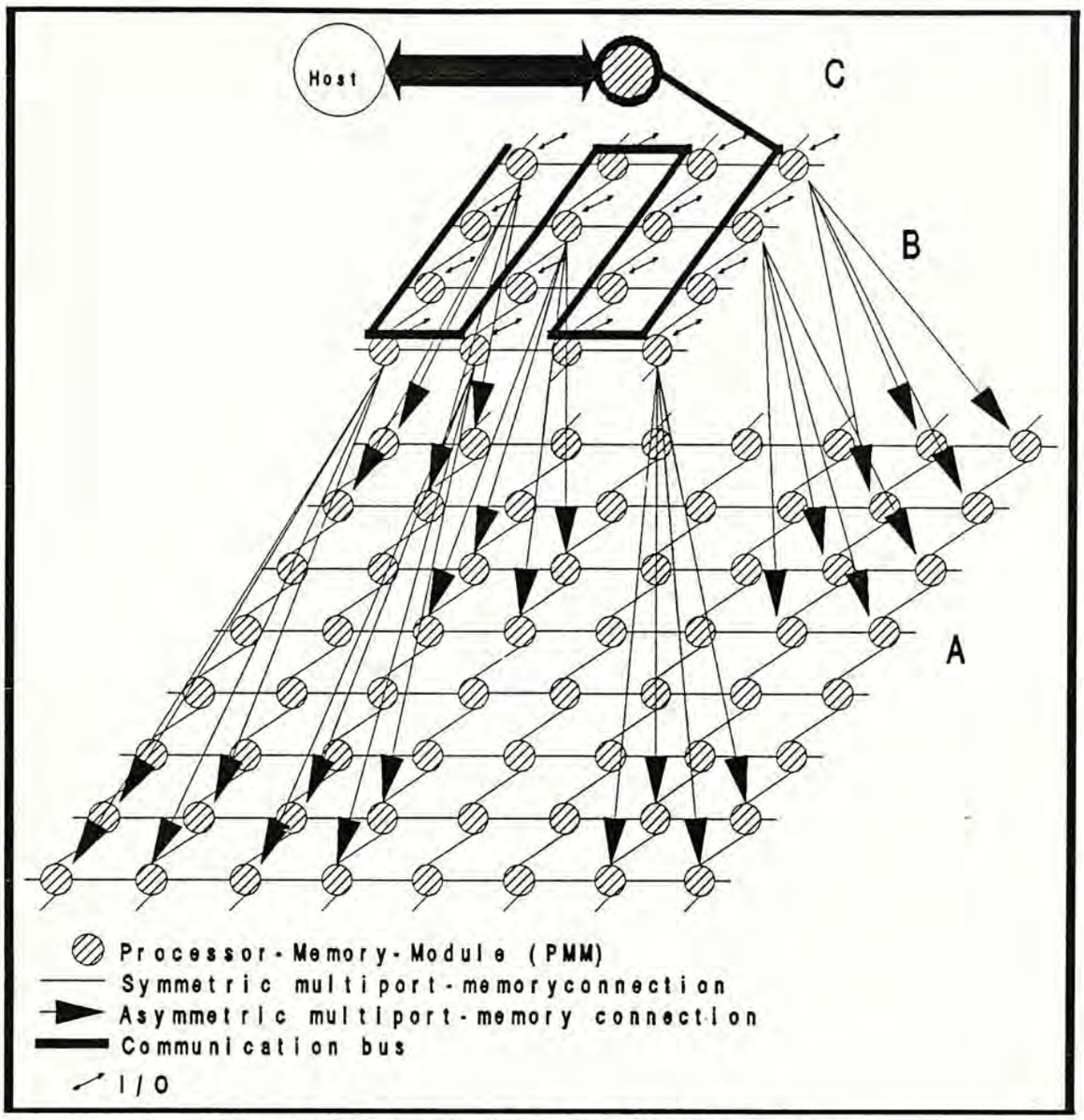


Figure 2.5 Example of a 3-level MEMSY structure.

for multi-computer systems but not for multiprocessor systems.

2.3 ELXSI

ELXSI System 6400 is a commercial product that features up to 12 CPUs. As shown in figure 2.6, all components are connected to a shared bus. ELXSI uses proprietary 64-bit CPUs. Depending on the processing requirement, 3 types of CPU can be chosen. Model 6410, 6420, and 6460 have the same architecture but increasing performance. Different models can coexist in a system.

The CPUs support the IEEE floating point standard. Each CPU has 16 sets of

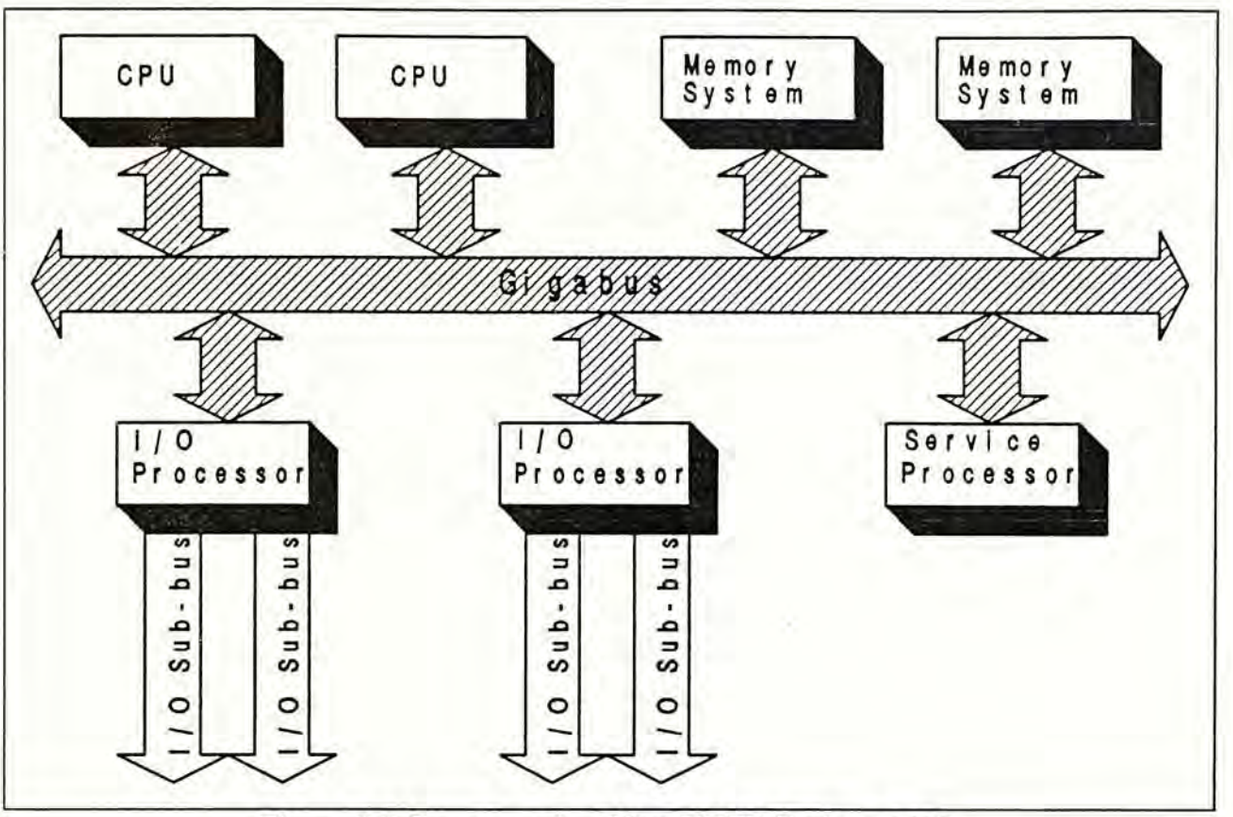


Figure 2.6 Components of the ELXSI System 6400.

16x64-bit general purpose registers and 16 sets of process context registers. Architectural support of the OS is evident.

Gigabus is a very fast (25ns cycle time), 110-bit proprietary bus. Virtual address of 6400 is 4 GBytes, with 2 GBytes per program space. ELXSI offers a virtual machine interface called the System Foundation. 25 system processes form an OS environment. Unix is also supported as the front end. Inter-process and device controller communication are accomplished via message passing.

In summary, this is a simple architecture with high performance bus and CPUs. But the use of proprietary devices raises the problem of incompatibility. Interested readers may see [Tabak90, Olson85, Sangu86] for more details.

2.4 Sequent

This well known bus based commercial multiprocessor system divides into the Balance (B) and the Symmetry (S) series. They are substantially different at the assembly language level but very similar at the high-level language level. A

general structure of the Sequent system is shown in figure 2.7.

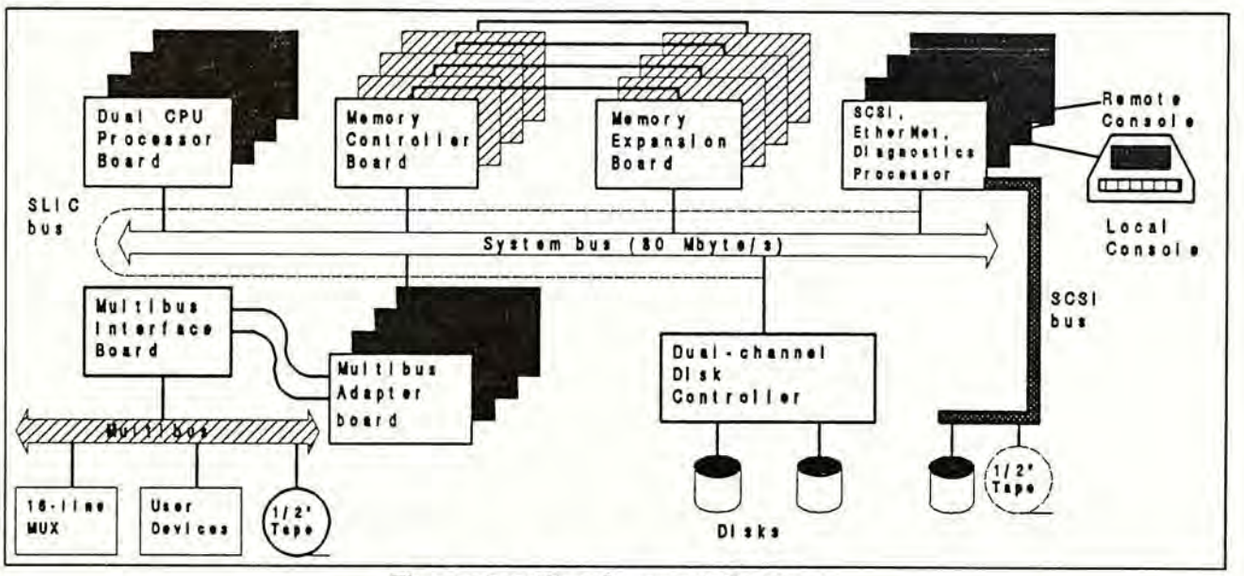


Figure 2.7 The Sequent System.

The B-series uses the NS32032 family, with the associated NS32081 floating point processor, and the NS32082 memory management unit. The S-series uses the i80386 and i80387 correspondingly. Up to 30 of these dual-processor boards can be connected in a system. Optionally, a floating point accelerator can be added for each S-series CPU.

The shared bus is a 80 Mbytes/s Multibus. Every device connected to the bus has a System Link and Interrupt Controller (SLIC) proprietary IC chip. It manages the control of multiple processors. All SLICs are connected by a bit-serial SLIC bus. It uses a high-speed, synchronous protocol independent of the system bus. SLICs communicate by exchanging command and response packets that are 17 bytes long.

The DYNIX operating system for the machine is UNIX compatible. It virtually supports any number of CPUs. System configuration is defined during start-up time. Ada is the chosen programming model and language.

As a whole, this is a very flexible and powerful system. The design decisions are reasonable and typical. Compatibility is improved by the incorporation of the SCSI bus. Information about this system can be found in [Tabak90, ThGiF88].

Although this system does not bear any surprising attribute, it is a commercial success and was used as the hardware background for the Yackos project described in section 2.5.

2.5 YACKOS

Although it is not a hardware construction project, this effort is closely related to our project. The support of message passing on top of a shared memory architecture is the project's main theme. Detials are given in [FinHe88].

YACKOS (Yet Another Communication-Kernel Operating System), like other communication-kernels, provides three major functions: process support, memory-management support, and inter-process communication. It aims at providing very high band width communication with very low latency. The message passing facilities have been implemented above DYNIX on the Sequent machine, which is described in section 2.4.

The reason for building Yackos message passing on top of a shared memory architecture is to allow the user the convenience of message passing with nearly the performance of shared memory. Similar attempts have been made by many researchers. [RetTh86] says that shared memory can support message passing for easier program decomposition.

Yackos reduces context switches by letting its processes communicate with the kernel by writing to and reading from a shared data area, the Interface Area. Processes may change part of their interface area by writing directly to it; some other parts may be changed only by calling interface functions that are linked with the process. The process and kernel together maintain busy and free pools for both incoming and outgoing messages. The headers for these pools are stored in the interface area. Processes allocate buffers by calling the function InitPool. Each process has its own set of buffers.

Large message buffers are 1024 bytes while small ones are 32 bytes. Message headers are stored directly in the buffers. Messages are stored in the order of reception on the busy input queue. A process calls GetOutput and PutOutput to get a free buffer then enqueue the message on the busy output queue. Corresponding GetInput and PutInput are called to receive messages. Figure 2.8 summaries the idea. Destinations are given as process identifiers.

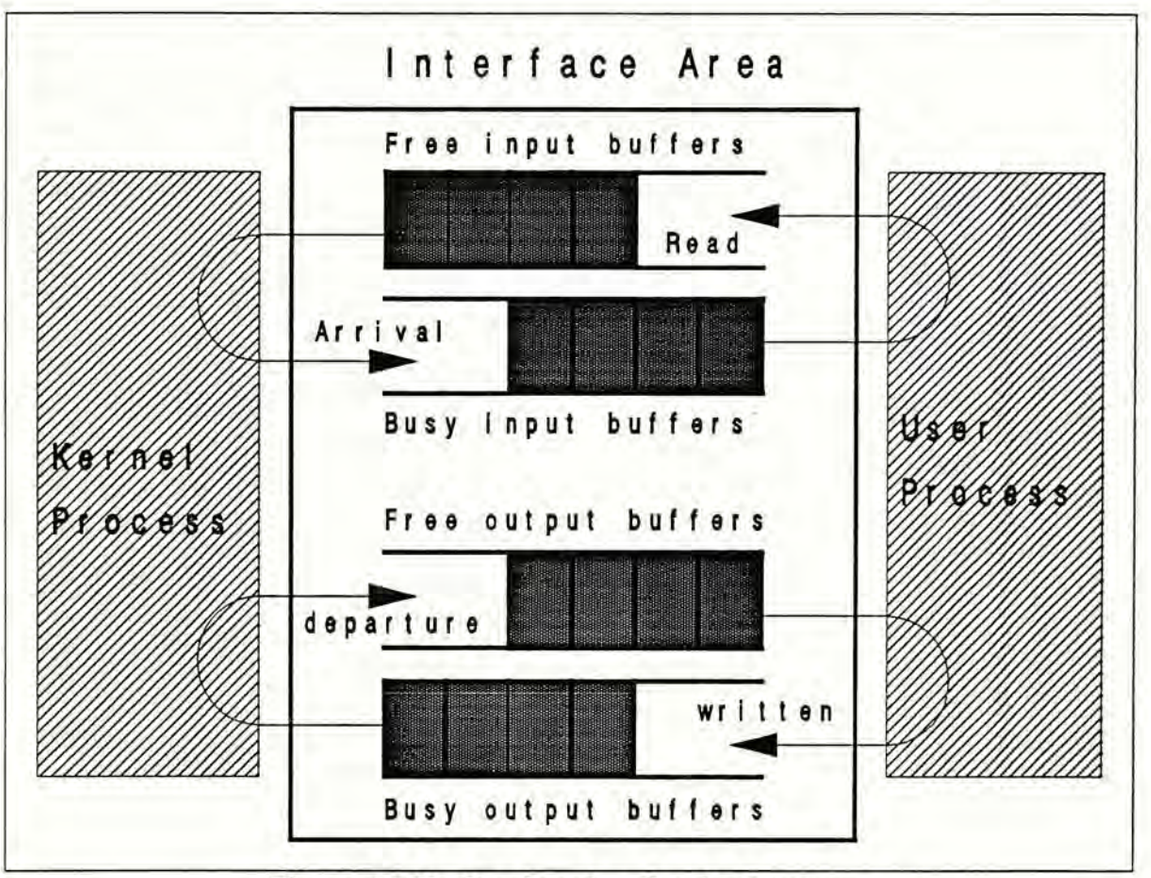


Figure 2.8 Yackos input and output queues.

Messages are sent either best-effort (try to complete a transmission but still allow it to fail) or reliably (every transmission must succeed). For this purpose, the sender's interface area variable InputStrategy together with the flag in the message header determines the kernel's actions when the receiver's input queue is full. The message is either discarded without notice, or returned to the sender, or kept in the sender's output queue.

On a native uni-processor implementation, calling an interface function should not switch context to the kernel. Only calls to the service routine Block (until buffer available) and NoOperation cause context switching. This design allows

a process to manipulate many messages before switching to the kernel for message passing operations.

DYNIX allows different processes to map the same physical space into their virtual memory so independent heavy-weight processes may share memory. This is the key feature to combine shared memory and message passing with Yackos.

The current implementation of Yackos uses DYNIX for process support and memory management. In place of a true native kernel, Yackos introduced a DYNIX process known as the **message passer** (MP). Processes that want to use Yackos for inter-process communication call an interface initialization function that maps the common data region into their virtual spaces and assigns them a Process IDentifier (PID). Given a PID, the MP can quickly find the interface area for a sender or receiver.

After initialization, processes may look up PIDs of other processes. This feature helps to establish process connections. Buffer queues are arrays of addresses of message buffers and are organized as circular queues. Locks are not needed because each queue has only one producer and one consumer.

Large message are not copied. The **address** of the message is placed in the busy output queue of the receiver, a free input buffer of the receiving process is removed and placed on the sender's free output queue (processes may share message space). Since small messages are only 32 bytes, they are copied.

The MP continually looks for messages that need to be sent. To speed up the search for new messages, they employed a small circular **hint queue**. Processes write their PIDs into this queue as a side-effect of calling PutOutput to send a message. Access to this queue is not locked so entries may overwrite each other and stale entries may remain. Therefore, all hints must be checked against the appropriate outgoing message queue. The MP only cycles through outgoing message queues when the hint queue is empty. The hint queue is inspected

again after each outgoing message queue. Current size of the hint queue is 5. The MP runs simultaneously with client processes so they are not blocked when they send or receive messages (ie. the sender and receiver are decoupled).

With the trickily designed message passer of Yackos, the frequency of context switching can be cut down. The key point is that now processes can send more than one message before reaching a context switch, provided that it is common to send messages in an intermittent manner. Such a design does not benefit interleaved conversations. However, measurements show that Yackos is a fast method which supports message passing on top of a shared memory architecture. Typical speed-up factor is 2.

In short, Yackos provides a faster way of message communication. Speed up comes from the saving of context switches and redundant memory copies for large messages. However, it is evident that context switching cannot be totally avoided if the message passer is executed by the same processor that is running the sender or receiver process.

2.6 Summary

Undoubtedly, shared memory multiprocessor systems built around a shared bus is a very cheap and direct way of getting cost effective computing power. That is the reason why they are commercially viable and competitive. However, it is not a good practice to let the programmer (user processes) handle all inter-process communication details due to security and effectiveness considerations. Thus, some multiprocessor operating systems allow message passing even on shared memory architectures.

Probably, the most straight forward approach is to provide system calls for sending and receiving messages. However, this is very ineffective because the service routines will be very bulky. Many other functions must be embedded into the service routines such as security checks, buffer allocation, synchronization

control, queue management, and so on. A better solution is necessary, otherwise the advantages of supporting message passing will be negated.

Yackos exemplifies a typical solution. A system process known as the Message Passer (MP) is introduced. It is executed simultaneously with user processes. Processes send and receive messages by invoking primitives (they are relatively short) which write down the requests on the shared memory area. The message passer performs all the housekeeping and checking procedures. Security is improved because the user processes theoretically need not manipulate the shared area directly.

Although the Yackos solution sounds good, the message delay is significant when the idea is implemented. It is evident that every message transfer involves at lease two context switches (to and from the message passer). The cost is so high that the user cannot benefit from the shared memory architecture. Even if the message passer is kept permanently resident in the main memory (not swapped out to the system disk), the two context switches still takes a long time for many microprocessors. So Yackos is forced to allow the primitives to allocate message buffers such that more than one messages can be sent before a context switch. Whereas the improvement is impressive, the problem is not yet eliminated. Actually, the assumption that messages appear in an intermittent manner may not hold for some applications.

Up to this moment, we have figured out the status of supporting message passing on shared memory architectures. The next chapter will describe our solution to this problem and show how the message passing paradigm can be implemented effectively on a shared memory system.

CHAPTER 3
THE MPC APPROACH

3.1 A shared memory multiprocessor architecture

In order to explain the concept of Message Passing Coordinator, which involves both hardware and software, we have to give a brief description of our proposed machine architecture first. Details will be given later in this chapter.

After the discussions in chapters 1 and 2, it is now clear that a bus-based shared memory multi-microprocessor system is a good choice for getting more processing power. Advantages can be gained in many aspects: low cost, simplicity, fast implementation, expansibility, and so on.

To reduce the use of the shared bus, each processor in the multiprocessor system should have local memory. A centralized, shared memory module is not necessary if the local memory of a processor can be accessed by other processors. So we get a distributed shared memory multiprocessor architecture. Based on this architecture, we will start our discussion on message passing as a way of inter-process communication.

3.2 Message passer for inter-process communication

From the discussion in chapter 2, we learnt that the Message Passer (**MP**) approach is an elegant way to support message passing on top of a shared memory architecture. The MP approach can be dated back to the work published by University of Waterloo in 1981, or earlier. However, their motivation was somewhat different. Let us have a quick look at it.

In [Gentl81], the Administrator concept was introduced for message passing between concurrent processes. The administrator is a software process for

managing worker processes (similar to today's servers) by handling the messages for them. Client processes and worker processes are decoupled. Other features such as message format checking and deadlock detection are built into the administrator. We must stress that this is a pure software project. Actually they were trying to support message passing with the help of the operating system. The underlying machine may be any serial computer that supports multi-programming.

Unfortunately, the MP is not efficient when really implemented. To explain our solution, we have to look at message passing using the MP approach.

3.2.1 A review of the message passer approach

Figure 3.1 briefly illustrates the message passer approach. As a system process, the MP is executed concurrently with user processes. When process A wants to send a message to process B, a system call (also called primitive) is invoked which writes the request to the mailbox in the shared memory area. The MP, as a middle-man, services the request and then marks the arrival of a new message for process B. Once B wants to receive a message, it invokes a system call and gets the message from the mailbox.

With the MP as a system process, security checks can be enforced. Functionally, the MP acts as a routing center. Point-to-point and broadcast messages can be handled efficiently. A hierarchical, typed message system is easy to support. The service routines in the communication kernel will be short and simple. They may be either run-time library routines or embedded into a high level programming language in the form of system macros.

Collectively, the basic duties of the massage passer include:

- manage the shared memory area
- enforce protection scheme

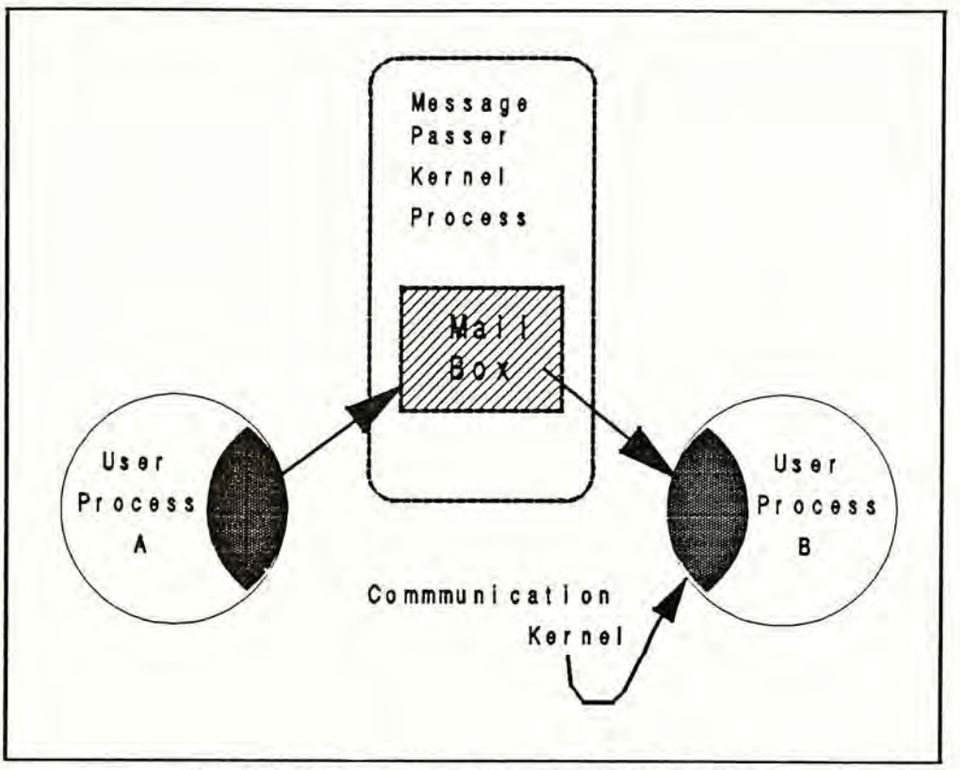


Figure 3.1 The role of the message passer.

- maintain message queues

The operation of the MP is transparent to the programmer. The MP buffers the outgoing and incoming messages for the user processes so that the sender and receiver(s) are decoupled. This is the basic requirement for asynchronous communication. Thus, the message queues should be maintained by the MP.

3.2.2 Pit-falls of the message passer approach

As aforementioned, the inefficiency of the MP is basically due to the cost of two context switches for every message. Although this can be cut down using the Yackos approach (refer to section 2.5), the remedy does not universally apply to all applications. Actually, messages generation pattern may vary dynamically for an application. Moreover, a multiprocessor operating system with special support for mapping virtual memory to physical shared memory is required.

One fatal problem concerns load balancing and the degree of parallelism. Recall that the MP may share a physical processor with other user processes, when user processes on other processor nodes request the service of the MP, context switch

to the MP is necessary. Evidently, the processor node where the MP resides carries a workload that varies according to the communication requirement of the processes on other processor nodes.

In order to deliver reasonable throughput, it is natural to raise the priority of the MP so that it can provide timely services. As a result, other user processes that share the same processor with the MP may suffer from series neglect. The overall effect of the problems mentioned in this paragraph makes the system behave in a complicated way. Consequencely, load balancing becomes quite difficult and the system performance is hard to predict and analyze. Finally, the degree of parallelism is lowered and uncontrollable.

To illustrate the above discussion, let us consider the situation shown in figure 3.2. When the message traffic in the system is light, more processes can be assigned to processor B that is running the MP so the given situation is possible. Some time later, most of other processes have finished their current jobs and try to exchange messages at roughly the same time. Processor B will then suddenly become overloaded and consequently most other processes have to wait, either for the MP or their conversation partner in processor B. The situation is even worse when some of the processes want to broadcast messages.

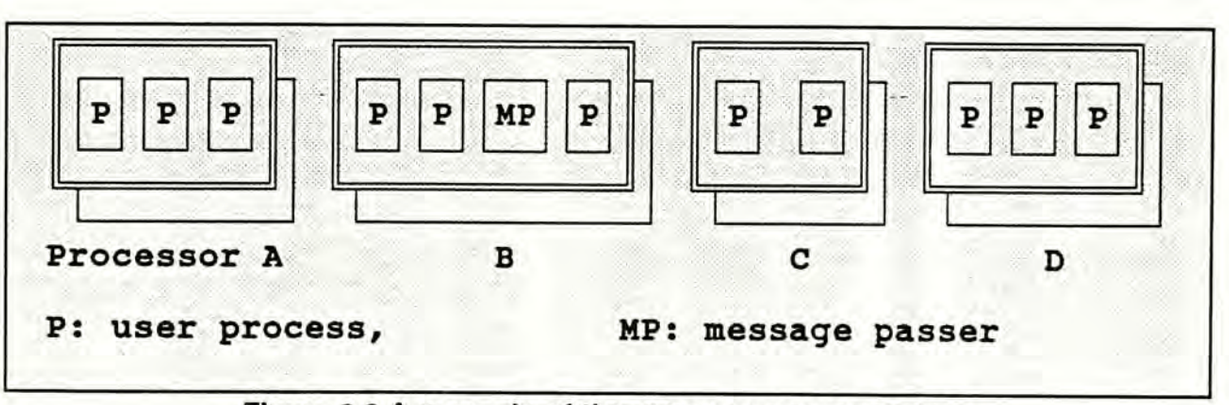


Figure 3.2 A scenario of the message passer system.

This simple example has demonstrated vividly that the simple message passer approach may cause series performance problems. Although Yackos [FinHe88] attempted an ingenious solution (section 2.5), the problem is still there because the applicability of the solution is very limited. We need a better solution.

3.3 The role of the MPC

3.3.1 The quest for the MPC

Recall that in section 3.2.2, we found that the work load the MP brings to a processor is difficult to estimate, so it is difficult to allocate this process to a physical processor. During busy time, other user processes being executed must yield the right of execution to the MP so the equilibrium is destroyed. Even if the system supports process migration, it is too late to move the MP to a less busy processor. Moving the user processes around is also very costly.

This line of through leads us to allocate a dedicated processor to execute the MP. An obvious motivation is that we must treat the MP differently because its behavior does not resemble other user processes. The first impact of taking this move is that a shared memory multiprocessor will no longer be homogeneous. To distinguish the processor that executes the MP, it is given the name **Message Passing Coordinator/Controller (MPC)**, which is inherited from its message passer antecedent. We define it as a couple of a software process and a physical processor. In our later discussions, when we talk about the MPC the context will determine whether the software process or the processor is referred.

For a message system using a MPC, the new picture is analogous to the **PABX** (Private Automatic Branch eXchange) [March77, ScoWa84] telephone system, where an "intelligent" switch box manages the message traffic. Such a switch box is usually computer-based. Figure 3.3 contrasts the two concepts. It is interesting to locate their similarities.

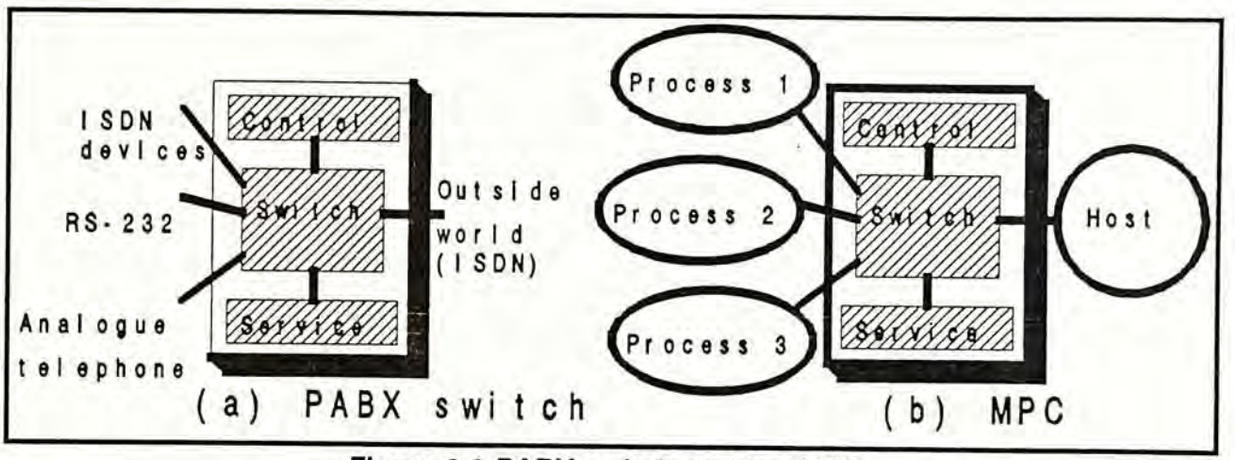


Figure 3.3 PABX switch and the MPC.

The PABX switch box connects the local site to the outside world. The MPC also connects the pool of processors (or processes) to the host computer which interacts with the external world. The users of a PABX switch are physical devices while the users of the MPC are processors (or processes).

The Switch part of a PABX switch box supports any connection combination (cross bar). The Control part is a computer that exercises the connection policy. The Service part provides dial tone, busy tone and other signals. Analogously, the Switch part of the MPC is the mailbox area connecting the processes. The Control part applies the predefined connection policy with the help of message queues. The Service part of the MPC provides ready signals, automatic acknowledgement and other services. As a whole, the correspondence is obvious.

3.3.2 Duties of the MPC

The basic responsibility of the MPC remains intact after evolving from the MP: it acts as a message control center of the programming environment. The difference is that the message passer had changed from a floating software process into a static, hardware and software combination. To look into the details, we have to consider software and hardware aspects separately.

3.3.2.1 Software aspects

Apart from message routing, the MPC must be responsible for other

housekeeping duties related to message traffic control. In some sense, it is a value-add MP. We shall start the discussion with its basic duties first:

a. Message routing and multi-casting

Each message (more precisely the message header) bears the sender and receiver PIDs. The MPC enqueues new messages on the input message queue of the receiving process, or a set of message queues when it is a broadcast message. The number of successful receptions is recorded. In order to achieve this purpose, processes that want to enjoy the service of the MPC have to register first. The MPC allocates mailboxes and creates message queues for freshly joined processes.

b. Message buffer allocation

We have stressed that it is very desirable to decouple the sending and receiving processes. To achieve this goal, the MPC must be able to buffer messages so that the sending process can resume its work once the message is handed to the MPC. Maintaining a free buffer pool is also a primary task of the MPC.

c. Message queue manipulation

In order to maintain the arrival sequence of the messages, there must be at least one input message queue for each participating process. The number of queues for each process depends on the complexity of the message system. For instance, more queues may be necessary if priority is enforced. The MPC is solely responsible for the manipulation of these message queues.

Now, let us turn to the optional duties of the MPC:

The MPC can offer optional functionalities if the application requires them and the work load of the MPC allows. Obviously, if the message traffic in a system is heavy, the MPC may not have sufficient capacity to perform extra works efficiently. Otherwise, it will become a bottle neck again.

a. A message filter

With the MPC as a middle-man, a process can specify the message type(s) that it interests in receiving, and the set of valid users. The MPC can automatically filter undesirable messages without disturbing the receiving process.

b. Deadlock detection/prevention

In message passing system, deadlocks can arise from the invocation of blocking Send and Receive primitives. With a centralized message controller, the problem of distributed deadlock detection may be avoided. Since the status of a process can be stored as a side effect when it invokes the communication primitives, the MPC can detect the existence of deadlock condition. Deadlocks can also be prevented by failing requests that lead to a deadlock situation. The detailed detection and prevention scheme is out of the scope of this thesis. If constructing a global process graph for deadlock detection is too expensive, the MPC may use a time stamp to check on unattended message and handle them accordingly.

c. Security checking

The message type, length, priority and destination PID can be checked by the MPC. Faulty messages can be found and discarded before they are actually processed by the MPC or the receiving process. Receiving processes can assume that arriving messages are alright. Security is improved and the disturbance due to defective messages can be minimized.

d. Acknowledgement generation

In many cases, the sending process may like to receive an acknowledgement from the receiver before it resumes its work. The MPC can generate a reply signal instead of requesting the receiver to create another message. Once the MPC finds that a receiving process has picked up the message, an acknowledgement signal is delivered to the sending process. This method is more economical and faster.

e. Performance analysis

The programmer can monitor the message traffic using the MPC. The MPC is easy to collect information about message frequency, generation pattern, and length spectrum. Information such as average queue length, average wait time and other statistical data are useful for parallel algorithm design. For a pure shared memory system, it is relatively difficult and inefficient to take such data.

f. Load balancing

Although this functionality seems ambitious, it may be useful and feasible for some applications. Imagine that several processes in the system are different instances of a class of server. If these processes call a primitive that performs blocking Receive, the MPC can easily identify which process is idle and consequently passes the next request, which is in the form of a message, to an idle server. Even if all the servers are busy, it is sometimes possible to estimate their workload in the near future from their input queue length. Then the next request can be passed to the process with the lightest future workload. Since the workload of a process does not necessary reflect the workload of the physical processor that the process resides, this feature is more suitable for systems that every processor executes only one process.

3.3.2.2 Hardware aspects

Basic duty

The primary duty of the MPC, form a hardware point of view, is to manage the shared memory and execute the MP program. Although the MPC can manage the shared area as good as MP, the MPC approach is more secure.

Strictly speaking, each processor needs to have access to no other shared memory locations but its mailbox. Only the MPC must access all the shared memory areas that are mailboxes. Owing to the hardware nature of the MPC, it is easy to design a address decoder that prevents a processor from accessing (or writing to) other processors' mailbox areas. Hence, the security problem of the shared memory architecture can be solved for the case in which each pro-

cessor executes a single process. Even if more than one process are allocated to one physical processor, the damage that a process can cause is restricted to the processes residing in the same physical processor.

If the private memory of every processor is hardware-protected from external access, some mechanism must be devised to load the program into this area. The simplest solution is the use of Read Only Memory (ROM). Another method is to install a switch to enable or disable the protection.

Optional duties

Since the MPC has a hardware portion, it is possible to support message passing by additional hardware. For instance:

In the MP approach, a process that wants to know whether a new message has arrived must invoke a primitive and request for the attention of the MP. The reply is very often "no message has arrived." Such polling operations will probably involve context switches as well as the use of the shared bus, given that the user process and the MPC are probably not on the same processor node.

A hardware alternative is to add a signal, such as MESSAGE AVAILABLE, for each processor. When the MPC has handled a new message for a processor (the MPC knows the process-processor mapping), the signal for that receiving processor is activated. A single instruction is enough for doing this. The status of this signal can be read from a status register on every processor node. In this way, the number of requests to the MPC and shared bus usages can be reduced.

This technique can be applied to the detection of buffer full, acknowledgement, and other similar situations. With considerable simplicity in software design, extra hardware cost is justified. The simple MP approach can also apply this technique but the hardware cost will be nearly doubled. Since the MPC is not adhered to a fixed processor, every processor must be capable of reading from and writing to the signal line. Only the MPC needs to drive the signal.

3.4 Advantages and disadvantages

At this juncture, the characteristics of the MP and MPC approach should be quite clear. Let us contrast the pros and cons of the MPC approach in the following sections. If not specified, the MP approach is used as a reference.

3.4.1 Advantages

a. Context switching is completely eliminated

Since the MPC is a dedicated processor which runs a MP-like process exclusively. No context switching is necessary for user processes and the MPC.

b. Hardware support is favored

The MP approach discourages hardware support for message passing because the MP is a floating process. As the example in section 3.3.2.2 shows, it is easy to add hardware for performance improvement in the MPC approach. Of course, the system designer must make cost trade-offs according to the application.

c. Improved security

In section 3.3.2.2 we discussed the possibility of preventing a processor from corrupting other processor's memory area with the MPC approach. Message type, priority, and length checking are introduced in section 3.3.2.1. With the MP approach, these security measures are difficult to enforce since a bulky MP will eat up memory spaces for user processes and the MP lacks hardware support.

d. Easy to monitor the system

We have mentioned that the MPC is a good place to monitor the message traffic in the system. Statistical data can be easily collected. If cost justifies, special hardware can be installed for diagnostic and monitoring purpose.

e. Easier to predict system performance

Since the power of the MPC depends only on its hardware configuration, the

service standard of the MPC can be accurately controlled. Concurrency is easier to control. On the contrary, we have shown in section 3.2.2 that the performance of the MP approach depends strongly on the task assignment and communication property of the processes.

f. Flexible MPC power

If the whole multiprocessor is dedicated to a single application, it is possible to estimate the communication load at the beginning of the hardware design stage. If communication is a bottleneck, a processor that is more powerful than other processors can be chosen for the MPC. Otherwise, a less powerful processor may be sufficient. Although object code compatibility should be maintained through out the system, alternatives are normally still available because many microprocessors appear in families. Other parameters of the MPC, such as private memory size, clock rate, and bus access priority can be adjusted too.

g. Primitives are kept simple

One of the original motivation of using the MP approach is to simplify the work of the communicating processes. This will shorten the time required to execute the SendMessage primitive and will also reduce the size of the primitives. Hence, the processes in the system will have more memory space for the application program because they do not have to carry the bulky, redundant codes in their object programs. On the contrary, the Yackos approach violates this principle because their primitives are responsible for queue and buffer manipulation, and hence much more bulky.

3.4.2 Disadvantages

a. Vulnerability of the MPC

All along our discussion, we have implicitly inherited from the MP approach that there is only one MPC in a multiprocessor system. But since the MPC is a dedicated processor, it may be difficult to replace if it crashes. There are several solutions for this problem:

1. **Have one more standby MPC** - it is a natural but costly choice. Actually, the existence of such a non-productive processor in a bus system with limited fan-out usually implies that the place for a productive processor is wasted. However, this approach exhibits the highest fault tolerance. The crash of the MPC does not need a system cold start to recover.
2. **Equip all processor with the capabilities of the MPC** - if special hardware is designed for the MPC to enhance performance, then implement all or part of this extra hardware on other processors. In case of MPC crash, select one of the processors to take up the job of the MPC. The extra hardware may not be as efficient as the full MPC but degraded performance may be acceptable.
3. **Software emulation** - a standby process that can emulate the function of the MPC is activated as soon as the MPC becomes non-operational. Remember we can resort to the MP approach in the worst case.

b. The multiprocessor is no longer homogeneous

Since the MPC approach encourages tailor-made MPC hardware, the system is no longer homogeneous. Usually, performance will not be degraded. But now we cannot use a single regular building blocks to construct the system. Apart from the lost of simplicity, the only implication is the vulnerability of the MPC.

c. Additional hardware and software necessary

This is an inevitable cost the designer has to pay. Fortunately, the amount of extra hardware and special software needed are subjected to the trade-offs between cost and performance.

3.4.3 Other discussions

Although the MPC approach is an improved solution for supporting message passing on top of a shared memory architecture, we must add that not all the problems are solved. A problem inherited from the MP approach is that the introduction of an agent between the message sender and receiver will more or

less lead in inefficiency, although the MPC approach has improved on this.

Since every message transferred requires the service of the MPC, and the MPC itself must service the incoming requests in a round-robin manner, the MPC cannot deliver all the processing power to a sending process even other processes are idle. The author had thought of using an interrupt driven MPC but it also becomes very inefficient when many processes raise requests in a short time.

However, we can borrow the philosophy of the Yackos approach to gain some improvement: a process can register more than one messages before the MPC services this process; and a hint queue can be used to speed up the polling job of the MPC. But evidently, the primitives will become much more complicated and this somewhat violates our goal. Design trade-offs have to be made here.

3.5 Summary

In this chapter, I revealed the serious problems with the MP approach. Accordingly, the MPC was introduced, which in a sense can be viewed as a value-added MP running on a dedicated processor. This approach favors hardware support for message passing. The extra functionalities that could be put into the MPC was discussed. Finally, we looked at the advantages and disadvantages of the MPC approach. All in all, the MPC is a more effective method to support message passing on top of a shared memory architecture.

CHAPTER 4
THE DESIGN OF SM3

4.1 Introduction to SM3

After we had decided to build a multiple processor workstation in order to get good processing power, we had also decide to use off-the-shelf microprocessors as basic building blocks. Their excellent availability, low cost, and high performance/cost ratio are strong reasons.

To construct powerful systems, the coarse gain MIMD class shown in figure 1.3 is a good choice for the basic model. The next major problem is how to interconnect these microprocessors. We found that a tightly-coupled system is favorable because loosely- or moderately-coupled systems can be built using TCSs as building blocks. Moreover, a TCS requires only one operating system. The amount of resources, such as wires and peripheral devices, is also minimal.

When choosing among interconnection topologies, the shared-bus architecture emerged immediately when cost, simplicity, expansibility and ease of prototyping were considered. Additionally, it can simulate virtually all other topologies.

Concerning the processor-memory interconnection style, the PE-to-PE type is adopted, with the local memory modules being globally accessible. In other words, this is a distributed shared memory system. Figure 4.1 illustrates such an architecture. It is a good choice because the great flexibility. The local memory for each processor node is for normal data processing while off-board memory access via the shared bus is strictly reserved for inter-processor communication. A processor should not place its data in the other processors' local memories.

At last the bus characteristics must be determined. We found that a synchronous bus with centralized control is easy to work on and interface with. The hardware

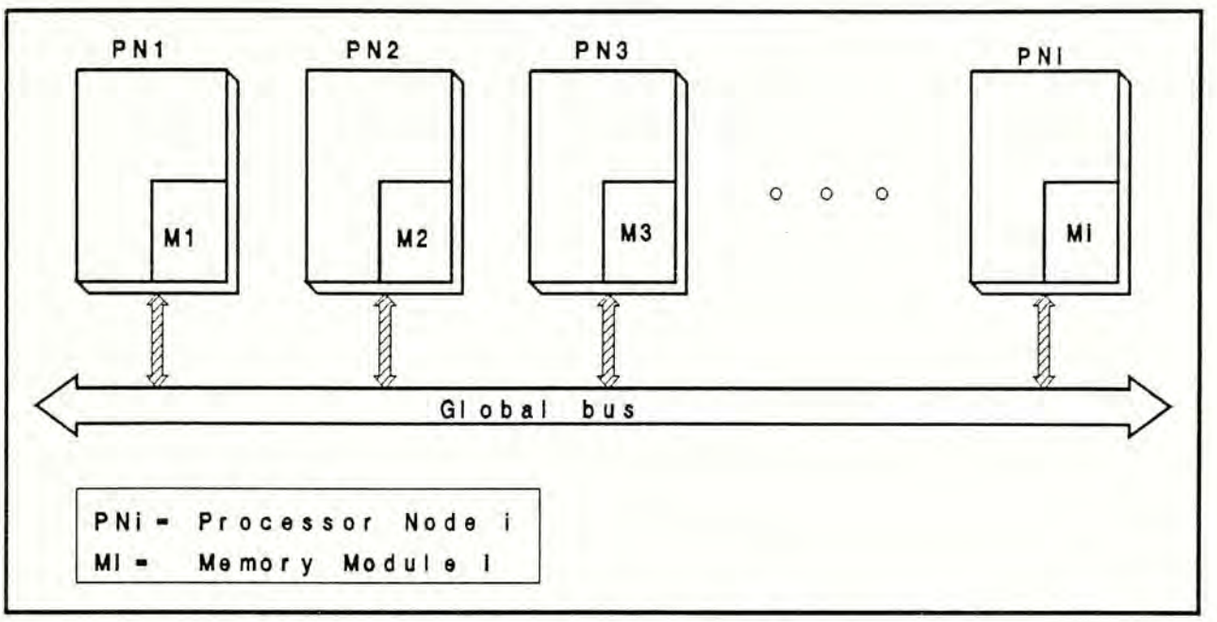


Figure 4.1 A distributed global memory multiprocessor system.

required is simple and not costly. Hence, we can indicate the position of our multiprocessor workstation in figure 4.2 by a box.

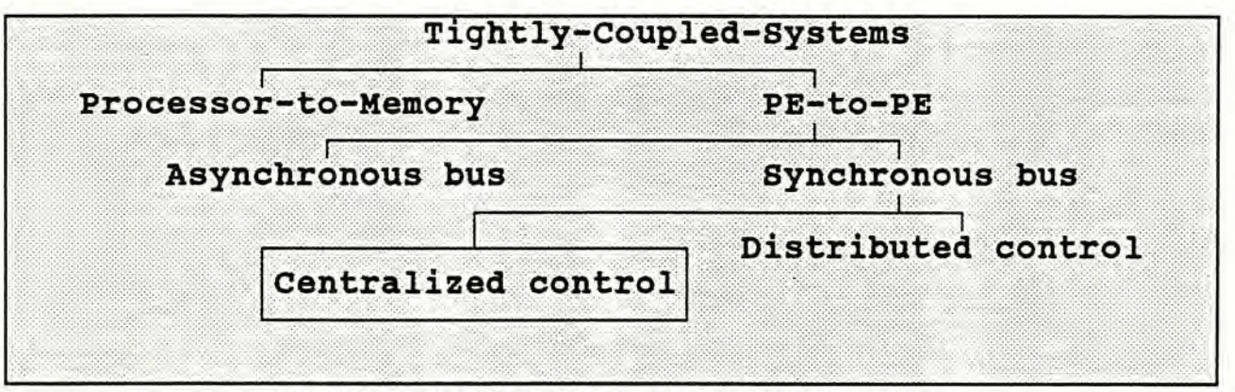


Figure 4.2 Position of the proposed architecture.

Since the workstation is a Shared Memory Multi-Microprocessor system, we called it **SM3** for short. In addition to normal processors, a host computer interfaces the pool of processors to the outside world via the front end operating system Unix. Let's look at the software and hardware aspects of SM3 now.

4.2 Software aspects

Although SM3 is a typical shared memory multiprocessor, we decided to support message passing on top of this architecture for easier programming. The details have been given in section 1.4. In chapter 3, we have introduced the MPC

approach as a better way to support message passing in contrast with the MP approach. Message passing with the help of the MPC provides a good environment for programming and inter-process communication on SM3.

4.2.1 Programming model

In chapter 3, we have introduced the basic and optional duties of the MPC. The programming model of SM3 is described in this section according to the basic duties of the MPC. For the currently constructed SM3, not all the optional duties of the MPC are included. To simplify our discussion, the part of the programming model concerning the optional duties will not be described in detail.

4.2.1.1 Logical entities

For the sake of simplicity, we **assume that each processor executes one software process only.** That is, the process-processor mapping is one-to-one. Note that the MPC approach does not impose this restriction.

The two basic classes of entities are user processes and the MPC. Processes communicate by passing massages. All messages are processed by the MPC. To maintain the order of and to buffer messages, an Input Message Queue (**IMQ**) is maintained for each process. This queue is created when the process registers to the MPC. Figure 4.3 is a simplified programming model of the MPC.

4.2.1.2 Communication procedure

After a process has registered in the MPC, it may converse with other participating processes by sending messages via the MPC. Every interaction involves the three phases described below. However, the procedure may differ slightly for various types of messages. We shall talk about the standard procedure first.

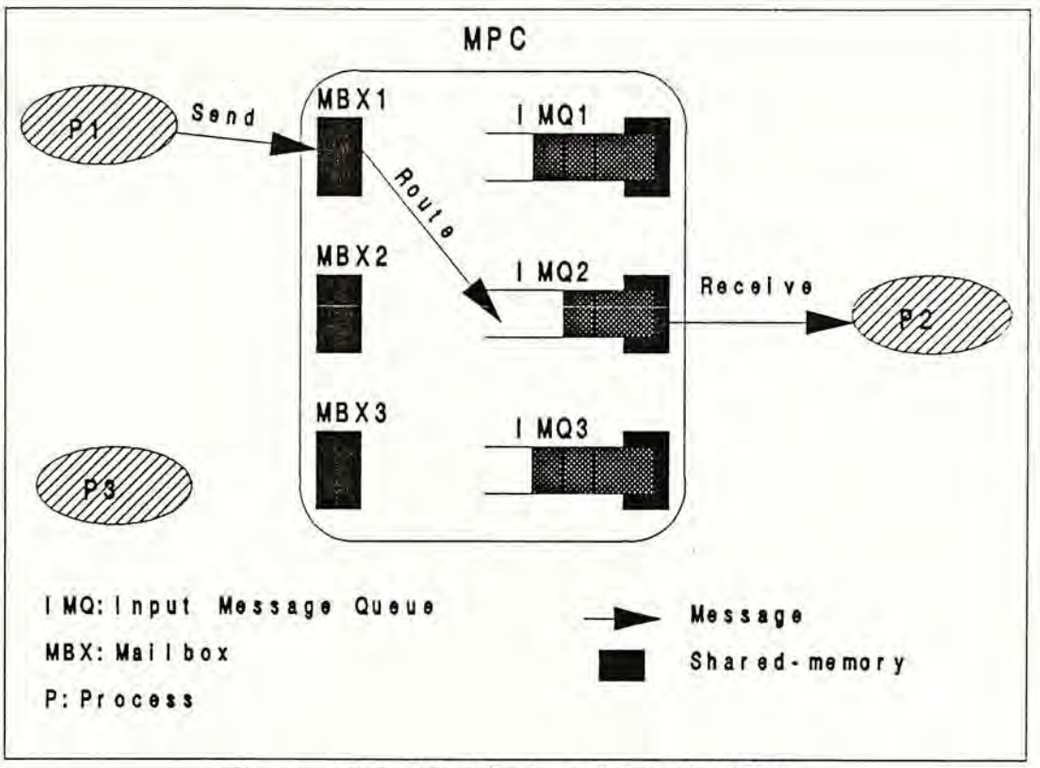


Figure 4.3 A simplified model of the MPC.

a. **Send message**

A communication primitive SendMessage is invoked whenever a process wants to send a message. Related information is passed as input arguments as the primitive is called. The caller must supply receiver PID, message content, type, priority, blocking option, and broadcast option. The primitive then writes the message header to the mailbox of the sending process.

b. **Route message**

The MPC cycles through the MBXs (MailBoX) endlessly. Once it finds a new message, it checks the validity of the message first. Then the message is enqueued onto the IMQ of the prospective receiving process. If it is a broadcast message, the header will be replicated and appended to all potential receivers' IMQs. The MBX of the sending process will be freed again after the message has been successfully processed. In case of buffer full, the message will remain in the MBX until free buffer is available. Although this phase is called Route Message, nearly all optional duties of the SM3 are performed in this phase. For instance, type checking and filtering before the routing is done.

c. **Receive message**

When a process wants to pick up a message from the MPC, it will invoke a primitive ReceiveMessage. The primitive will read in the first available message from the IMQ of this process. After a message is read, a flag is set by the primitive to tell the MPC to get the next message. Now let us turn to the acknowledgement procedure.

If the sending process waits for an acknowledgement before resuming its work, it is called a **Blocking** Send. Similarly, a receiving process waits until a new message arrives if it is performing a blocking Receive. In the first case, the acknowledgement is generated by the MPC. The detailed difference between blocking and non-blocking Send are:

Non-blocking - after the SendMessage primitive has successfully written the request to the mailbox, it returns straightaway. For a non-blocking Receive, the primitive also returns as soon as it has found that there is no available message. This result is then reported to the receiving process.

Blocking - after the SendMessage primitive has written the request to the mailbox, it then waits for the acknowledgement. When the MPC finds that the receiving process has picked up the message, it sets a status bit in the shared memory to inform the sender of the acknowledgement. To avoid busy waiting and shared-bus contention, the technique described in section 3.3.2.2 is applied.

During the course of execution, some processes may want to transfer an urgent message. The following example will explain this requirement:

Suppose SM3 is working on a matrix chain product problem. Each processor is responsible for calculating the partial product of two or more matrices. Computation job assignment is conveyed by a message. At an instance, a process may have several pending messages in its IMQ. If one of the processes found that the result of its partial product is zero, then the final answer is also zero and other pending computations can be canceled. This condition may be indicated by a

message with higher priority in order to overtake other normal messages.

In SM3, 3 priority levels are supported. In the order of decreasing priority, they are **Express messages, Normal messages,** and **Broadcast messages**. Express messages may be point-to-point or broadcast messages that are urgent. Normal messages means ordinary point-to-point messages. Since the receiver is less specific, broadcast messages yield priority to normal messages.

In order to support these 3 levels of message priority, a separated IMQ is created for each priority level. The names of the IMQ for Express, Normal, and Broadcast messages are called EIMQ, NIMQ, and BIMQ respectively. Figure 4.4 shows a view of the IMQs.

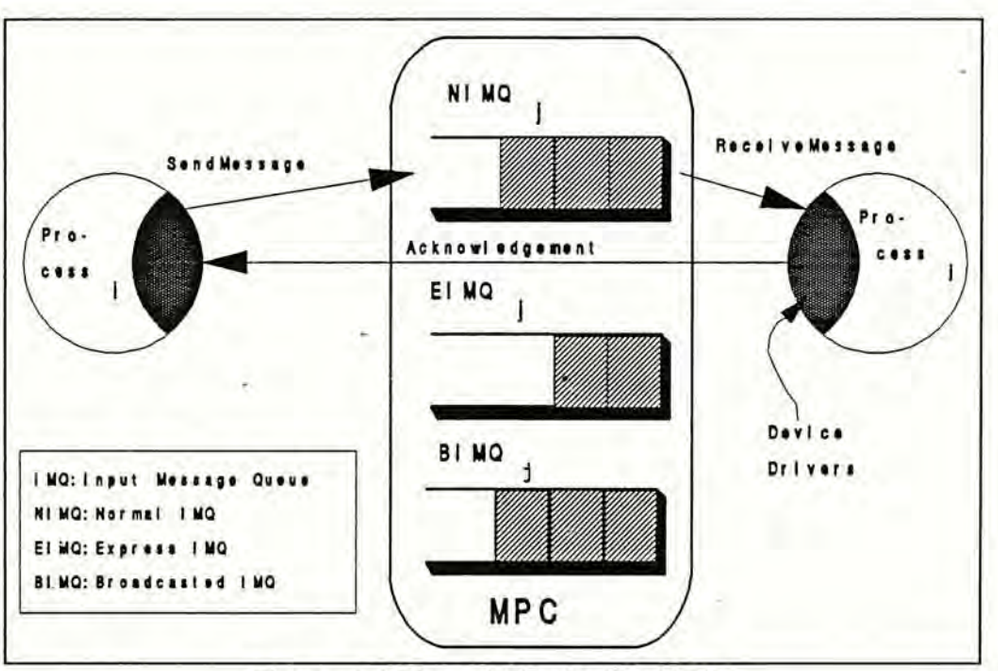


Figure 4.4 The 3 IMQs in the MPC.

The SendMessage primitive writes the message frame to the mailbox as usual. The MPC will then despatch the incoming message to the appropriate IMQ of the receiving process. When the receiver calls ReceiveMessage to receive any new message, the IMQs are examined according to the priority order. Hence, higher priority messages will reach the receiver earlier. However, the receiving process can also specify that only messages from a specific queue is desired.

4.2.2 Message structure

A build-in hierarchically typed message structure is highly desirable for any message-based systems. Different messages types can be handled in different ways by the system in order to improve efficiency. Programmers can save their development effort because they don't need to deal with the operational details. The programmer is only required to pass suitable arguments to the communication primitives SendMessage. On top of a basic build-in message structure, the programmer can also add their user defined message types, such as finely classified message types.

In SM3 a hierarchical message structure shown in figure 4.5 is supported with the help of the MPC. Message type is determined by 3 basic properties: priority, blocking option, and broadcast option. We shall elaborate the tree structure according to these properties.

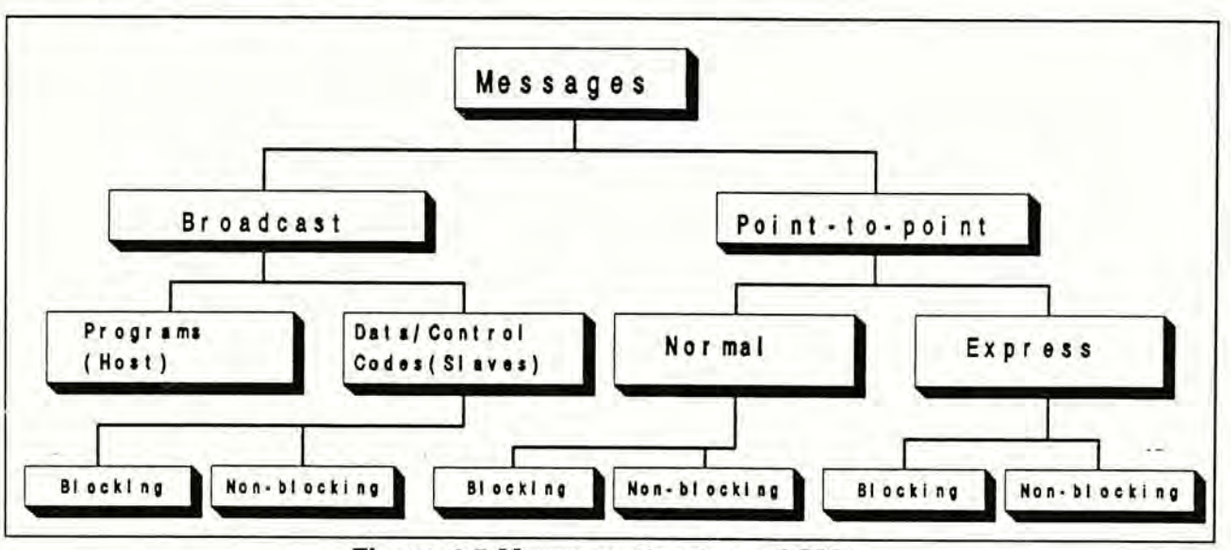


Figure 4.5 Message structure of SM3.

4.2.2.1 Broadcast versus point-to-point messages

Point-to-point is the most basic form of communication. A sending process identifies the receiving process uniquely by specifying the PID when the message is sent. All the MPC has to do is a check of the existence of the target process and status of the associated IMQ.

In any system with a number of cooperating processes working on the same problem, broadcasting is a very common form of communication. Although broadcasting can be done by the sender using a sequence of point-to-point messages, it is bothersome and inefficient. So, the burden of replicating messages for broadcasting is taken up by the MPC.

Furthermore, it is easy to support selective broadcast (multi-casting). When a sending process calls the SendMessage primitive, a list (in the form of a bit-mask) is supplied in place of the input argument receiver PID. The MPC is also responsible for keeping track of successful receptions and reporting them to the sending process for the sake of error handling.

4.2.2.2 Message priority

We discussed the reason for having more than one priority level in section 4.2.1.2. The number of levels is kept small because it is evident that a separate message queue for each priority level is very expensive. So in the message structure hierarchy, we have 3 levels. Normal and Express messages were explained in enough detail in section 4.2.1.2. Broadcast messages require further elaboration. In the message structure, broadcast messages are divided into **Programs** and **Data/Control Codes** for slaves.

The existence of program type is because during system start-up, the host machine must load programs into the local memory of each processor. In many cases, the programs running at all processors are the same, only data values are different. Thus the host can broadcast the program like a message to all processors using this message type. This type is special because only the host machine may initialize such a request. Moreover, the MPC must **directly write the message content**, ie. the program, into the local memory of a processor instead of just leaving it in the mailbox. The program load origin and entry point must be generated by the MPC or supplied by the host pocesses. For this type of messages, the blocking and non-blocking option is inapplicable.

Another broadcast type messages are initiated by slave processor. The message contents may be purely passive data or control commands. Normally no executable code will be send so starting address is not required.

4.2.2.3 Blocking versus non-blocking

When the blocking option is on, a sending process must wait for the acknowledge from the MPC before it returns from the communication primitive and continue its work. For a receiving process, the primitive does not return until a new message is available. Naturally, this message type can be used for inter-process synchronization as well as data passing. However, it should only be used if necessary because the processing time will be wasted. Degree of parallelism is also reduced.

If the non-blocking option is chosen, the SendMessage primitive returns immediately, without waiting for the acknowledgement, to the next statement in the user process. At the receiver's side, the ReceiveMessage returns immediately no matter a new message is available or not. Of course, this result will be reported to the process. Using this option, the sending and receiving processes are effectively decoupled. A fast sender will not be dragged down by a slow receiver, except when the IMQ of the receiver is full. This option is good for passing data or control commands because of the loose coupling.

Sometimes the receiving process has to supply more information with an acknowledgement. The blocking-Send with automatic generation of acknowledgement by the MPC is inadequate. A **higher level acknowledgement** is necessary. Figure 4.6 presents a simple solution. It is interesting to note that both the sender and receiver use blocking and non-blocking options. In particular, Sends are non-blocking while Receives are blocking. The following paragraph explains the solution.

After the sender has used a non-blocking Send to deliver the message to the

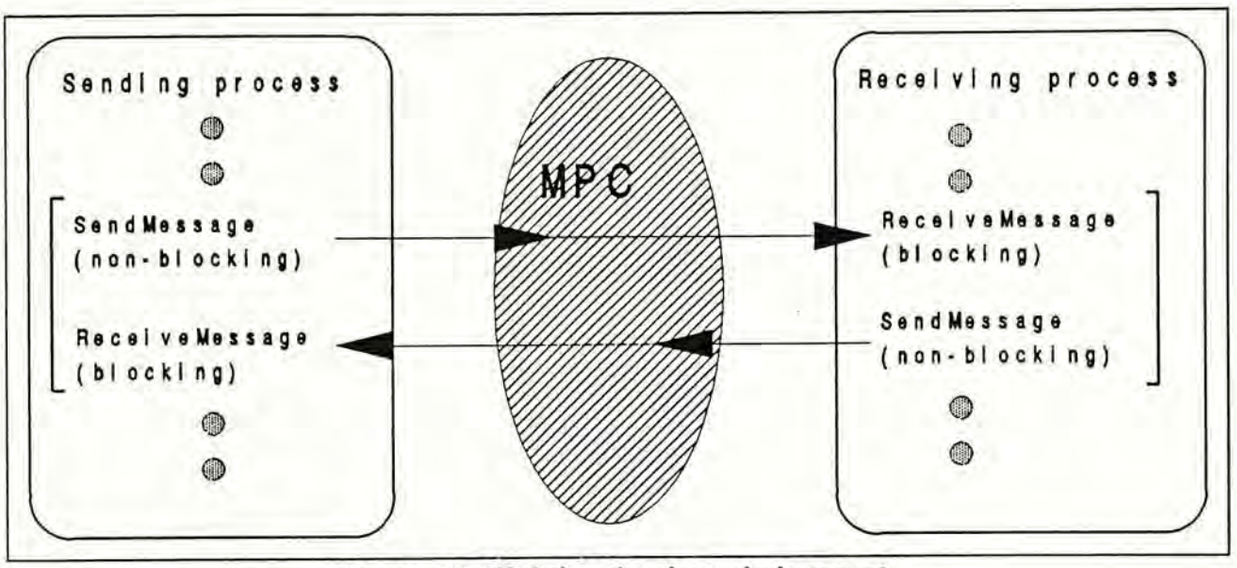


Figure 4.6 High level acknowledgement.

receiver, it initiates a blocking Receive to collect the expected acknowledgement. On the other side, the receiver uses a blocking Receive to get the incoming message. Once the message is captured, a non-blocking Send then transfers the acknowledgement message carrying the reply information to the sender. After this, the receiver can continue its work immediately. Upon receiving the acknowledgement, the sender can also return to its work.

The two statement at each side can be grouped together to form a higher level blocking Send and blocking Receive, respectively. Such kind of communication allows the receiver to return a data message as the acknowledgement. Except the above interesting example, the same option, either blocking or non-blocking, are used consistently at both sides.

4.3 Hardware aspects

4.3.1 Overall architecture

SM3 is a shared memory multi-microprocessor system. An asynchronous bus with a centralized controller is chosen as the backbone. A number of **Processor Nodes (PNs)** are attached to the shared bus, and so is the MPC. We have a **host machine** also attach to bus for handling peripheral devices. A portrait of this configuration is shown in figure 4.7.

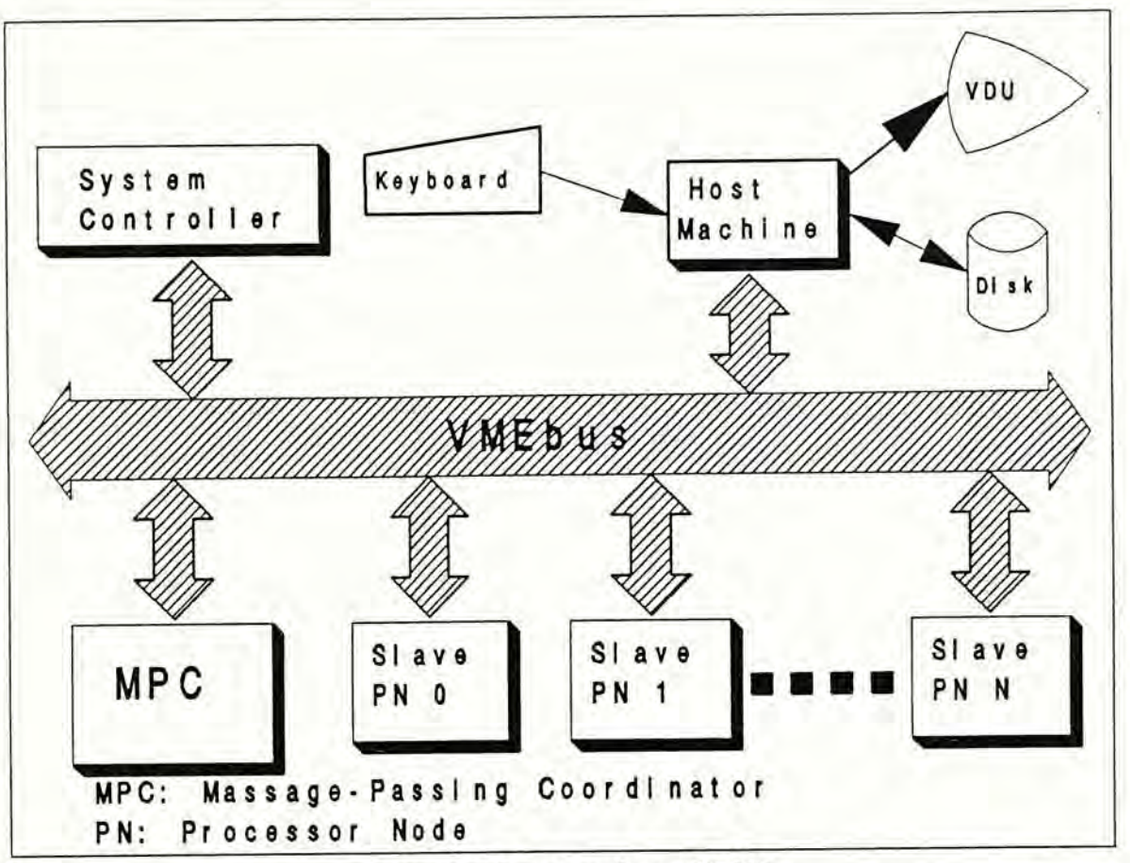


Figure 4.7 An overview of SM3.

The **system controller** in the figure is responsible for bus arbitration and control. It may be integrated into the host machine or exist as a separate circuit board. For further expansion, an interface module can be attached to the bus in order to form a loosely- or moderately-coupled system at a higher level. A clustered approach can be found in figure 7.1. Although not shown in figure 4.7, additional pure memory modules can be added to the bus as a sort of shared resources.

4.3.2 The host machine

As shown in figure 4.7, the host machine is responsible for handling peripheral devices. Actually it is a complete microcomputer system with a uni-processor operating system. The host machine needs not base on the same microprocessor on the PNs. But for simpler operation, the host machine should be compatible with the PNs at object code level. The internal structure of the host machine dependents on the implementation choice. However, it will be particularly good if its organization is similar to that of PNs'. The duties of the host machine is detailed below. Some of them are basic duties while some are optional.

a. Peripheral control

All peripheral devices are connected to the host machine. A standard configuration includes: video display unit, keyboard, fixed and removable disk drivers, and printer. In order to control the devices, the host machine must be equipped with a good operating system. Besides, a good I/O co-processor is highly desirable for better external parallelism.

b. User interface

The user issues commands to the host machine to start the application program, monitor the system, collect results, and terminate the program. The host machine connects the pool of slave processors to the outside world - the user. For easier operation, a user friendly operating system should be chosen.

c. Program development

Very likely, program development will be conducted on the host machine. So text editor, compiler/assembler, linker, and loader must be present. A debugger is highly desirable too. Note that the programming language used need not have build-in concurrent constructs. At this first version of SM3, distribution of processes is not done by the operating system since the host machine only has a uni-processor operating system. Task assignment is the programmer's duty so the underlying language can be a conventional one.

d. System monitoring, control and diagnosis

Since the host machine controls the operation of the whole system, it is natural that the user also monitors the operation of the system here. Actually, the host machine is like the operator's console of a large computer system. During development stage, the PNs and the MPC are tested by the host machine individually and then incorporated into the system. Then, integration test is conducted. All diagnostic work will become easier with the help of the host machine, especially due to the memory-sharing nature of SM3. The host machine must also be responsible for overall system control, such as cold reset, warm reset, shut down, and slave processor halting.

e. System initialization

Before the execution of the application program, the host machine must distribute the object program modules to the slave PNs. The initialization of the MPC is also its duty. After the program modules have been loaded, the host signals the PNs to start their jobs. Note that all the PNs must be halted before the programs are loaded, and after the application programs have terminated. Otherwise, free-running of the PNs will cause unpredictable effects.

f. Execution of the root process

When the system initialization is completed and the PNs are started, the host machine must issue data packets and/or control commands to the PNs. Hence, the execution of the root process is also a primary duty of the host machine. When the results become available, the host machine collects them from the PNs possibly in the form of messages.

4.3.3 Slave processor nodes

Figure 4.8 is a simplified functional architecture of a PN. A detailed diagram for the current implementation of SM3 will be presented in chapter 5. Again, a PN employs the bus structure, which this is the universal choice for single board computers due to efficiency and cost considerations.

The Bus Interface links the local bus to the shared bus. If the local bus is not completely compatible with the external shared bus, conversion logic is necessary. The Shared memory is a part of the system address space so we placed an index i there. The actual address format will be given in chapter 5. This local memory can be accessed from the shared global bus via the Bus Interface. Of course, the Microprocessor is the heart of a PN. It may access other off-board memory as well as the local memory.

The Communication Logic part connects the PN to the MPC. Hand shaking protocols are implemented here for supporting message passing. Some of the signals

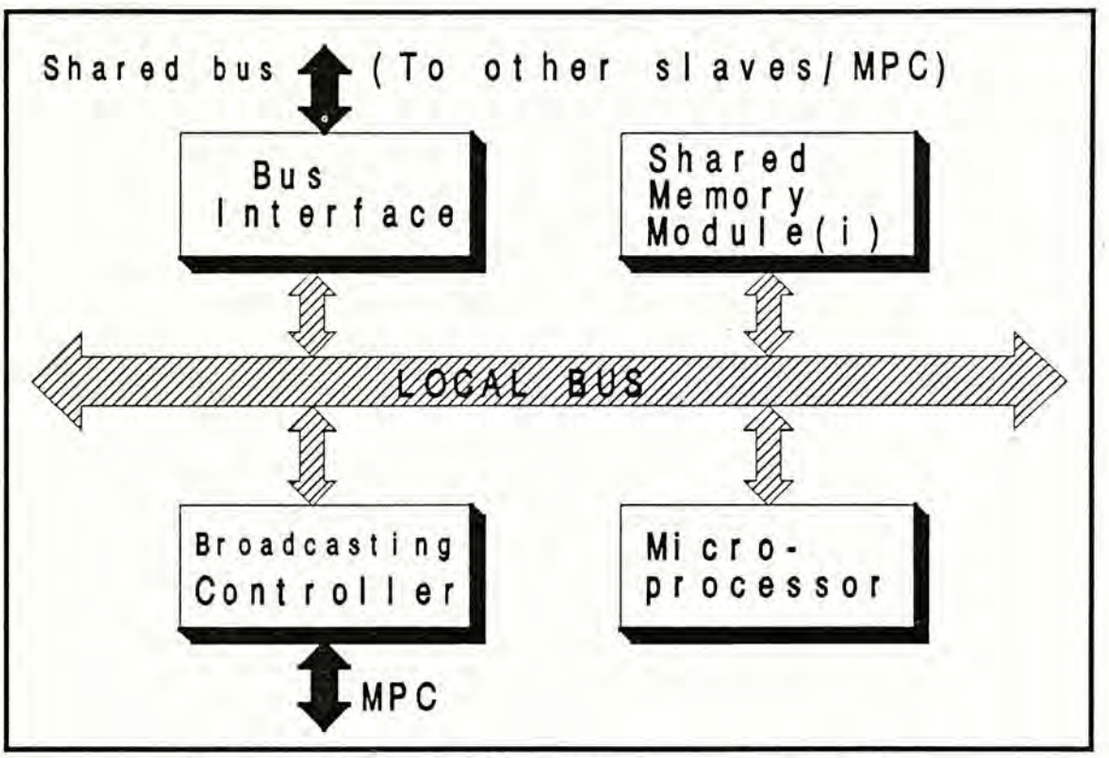


Figure 4.8 General architecture of a processor node.

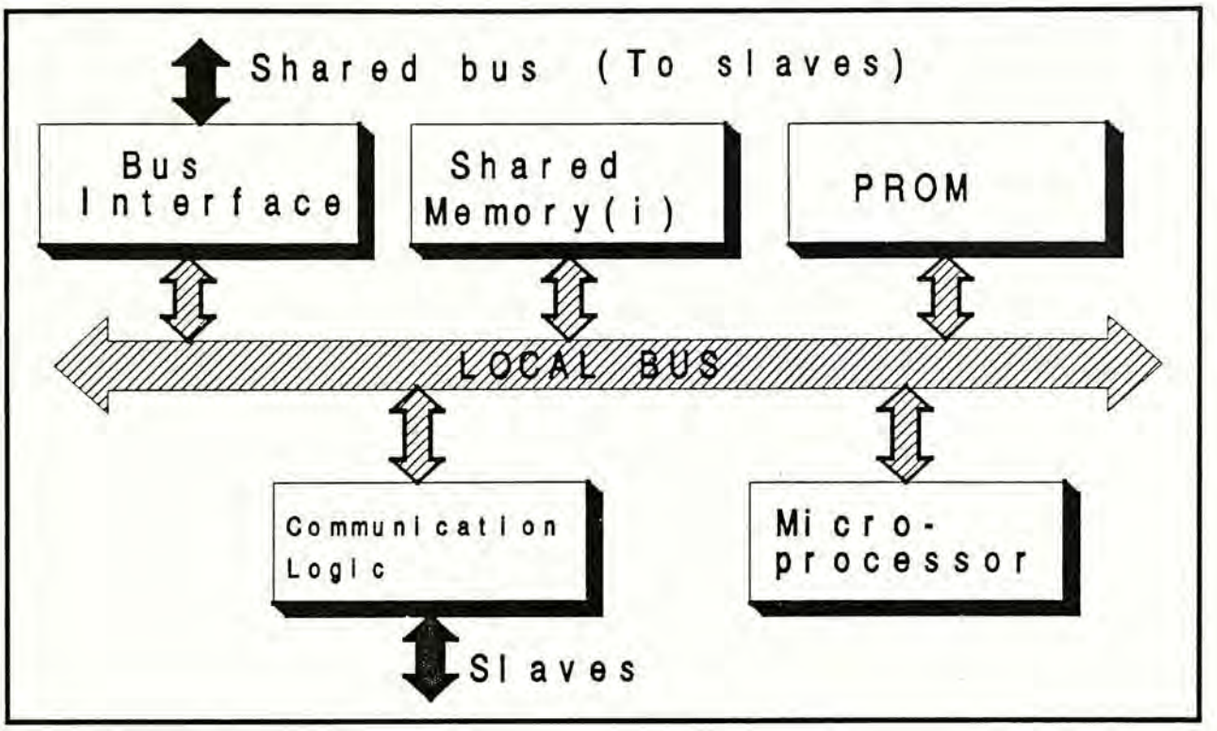
between a PN and the MPC are optional. They are added for reducing global bus access and MPC interrogation as described in section 3.3.2.2.

Other devices, such as DMA controllers and mathematical co-processors, can be added to the PN local bus for better performance.

4.3.4 The MPC

The MPC structurally resembles a PN. Figure 4.9 is a simplified diagram of the MPC. The Microprocessor should be object-code compatible with that on PNs. The Bus Interface and Shared memory parts are virtually identical to their counterparts on PNs, although the memory size need not be the same. If the MPC software is bug free, it can be burnt into a PROMM (Programmable Read Only Memory) chip in order to save the initialization time and to ensure reliability. But it should be noted that PROM are usually slower than RAM (Random Access Memory). In our prototype design, PROM is not necessary.

The Communication Logic links up the MPC and PNs. The ultimate goal is to provide an alternative channel for handshaking signals that speeds up message



passing. Actually, it may be called a **communication sub-bus**. We must stress that the communication logic is not mandatory because handshaking can be done with software. However, the advantage of minimizing of global bus access usually forces the designer to add this logic.

4.4 Communication protocols

The communication protocols discussed in this section are not for high level message exchange between processes. Instead, they are designed for supporting efficient message transfers at the processor level. According to the nature of different message types, specific mechanisms are designed to handle them in the most suitable way. Table 4.1 summarizes the mechanisms used. We shall elaborate this table in the remainder of this section.

4.4.1 Short and long messages

From the basic principle of the MP and MPC approach, we found that messages are stored in message queues before they reach their target processes. Obviously, it is not feasible to put long messages into the queues because of the

	Long messages	Short messages
Point-to-point	Normal DMA	Shared-memory
Broadcast	1-to-N DMA	Shared-memory

Table 4.1 Mechanisms for message-passing.

memory size limitation. The time wasted in copying long messages may also be very significant. But the most critical point is the hold up of the shared bus, which is a potential bottle neck.

For the above reasons, the SendMessage primitive only puts **the pointer to the message body** into the message header, which is actually enqueued onto the IMQs, if it finds the message is a long one. The receiving process is responsible for reading the message body from the local memory of the sending process, using the pointer given in the message header it has received. Note that this mechanism is feasible only on distributed shared memory systems such as SM3.

On the contrary, using this indirection method for short messages will be quite inefficient because two memory accesses are required. Thus, the SendMessage primitive will embed the message body into the message header if it finds that the message is short enough. Upon arrival, the receiving process just extract the message body from the message header. This mechanism does not require all the memory be globally accessible. Only the mailboxes must be shared.

4.4.2 Point-to-point messages

Recall that a point-to-point message has exactly one sending and one receiving processes defined uniquely. Short point-to-point messages are efficiently handled by the simple shared memory access mechanism described in last section.

Long messages require a lot of consecutive off-board memory accesses initiated by the receiving process, in order to bring the message body from the sending processor to the local memory of the receiving processor. Such a job fits a

DMAC (Direct Memory Access Controller) perfectly. Figure 4.10 illustrates the whole picture.

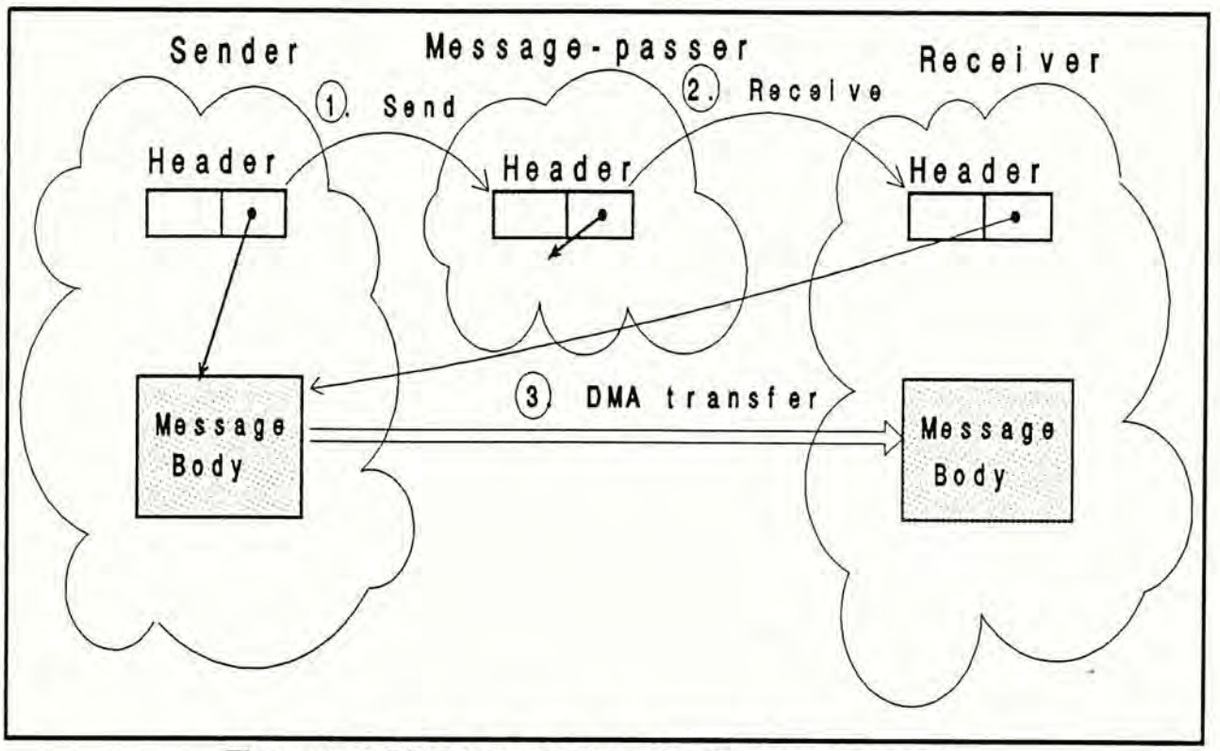


Figure 4.10 DMA transfer for point-to-point messages.

After the receiving process has read in the header, it will initialize the DMAC on that PN based on the information provided in the header. The header supplies the pointer to the starting address and the message length. The argument passed to ReceiveMessage supplies the buffer address. The DMAC can use these 3 parameters to perform a memory-to-memory transfer. Let us look at the advantages of such an indirection method:

a. The bus is not used extensively until the receiver really wants the data. For early termination, bus cycles are not wasted.

b. The receive buffer organization is not complicated. On each PN, one buffer queue for each possible receiver is enough. Note that these buffer queues are inevitable if asynchronous communication is supported.

c. When a sender suddenly issues many long messages, the MPC will not suffer from buffer shortage and subsequently cause performance degradation. It is the sender who suffers. This is more reasonable.

d. The system bus holdup due to long messages is relieved because each

message go through the bus just once. It can be generalized for broadcast messages. We will see that the savings is great.

e. Sender and receiver are decoupled effectively.

On the contrary, there are some draw backs:

a. The receiving PNs have to read the sender's memory, this inevitably affects the operation of the sender.

b. If many receivers want to read the long messages sent by a PN simultaneously, contention arises. For the case that the message body is embedded into the header, there is no contention because the transmissions are serialized by the sender or the MPC. Moreover, the average wait time of receivers is roughly half of the total time for all messages. We assumed that messages are fixed in length. On the contrary, the average wait time is nearly the whole transmission period for all messages when using the indirection method - if each receiver accesses the sender's memory in an interleaved way due to contention and random bus arbitration.

c. When the receiver needs data immediately, delay due to the DMA is undesirable, although the DMA is fast. But actually, any transfer of data to the receiver's memory affects the receiver so it is not an extra cost.

One question with this DMA method is how to find suitable program code for the CPU to execute while the DMAC is in operation. Although the CPU and DMAC can operate in parallel with the DMAC programmed in the cycle stealing mode, sharing of the local bus between the two devices limited the concurrency. Moreover, if the message body is not yet available, user code may be unable to proceed and thus it becomes pointless to use the cycle stealing mode. Hence, the DMAC is programmed in burst mode.

4.4.3 1-to-N DMA for broadcast messages

Although broadcast messages can be treated in the same way as a series of point-to-point messages, we found that it is very inefficient even with the DMA method described in section 4.4.2. The message has to go through the shared bus (N + 1) times if the message is bufferred in the MPC, and N times if the DMA method is used, where N is the number of receivers. Such long hold-ups of the shared bus will degrade the system performance. Obviously, this is an inherent property of the MP and MPC approach.

Fortunately, the author found that the problem can be solved satisfactory on SM3-like architectures. The necessary condition is the presence of the MPC, a shared bus, dual-mode DMACs, and some clever hardware logic. We shall explain the details below.

4.4.3.1 Introducing 1-to-N DMA

Although the bus topology favors broadcasting, it is not directly available for use at the high level programming language level. Moreover, there is a fundamental difference between the kind of broadcasting that a "bare bus" (with no additional software or hardware) supports and the kind of message broadcasting me are looking for. A "bare bus" only provides "blind" broadcast. That is, all PNs can listen to the broadcast and they determine whether to get the data or not. But we should allow the sending process to choose potential receivers too. Other PNs not chosen should not listen to the message. Another difficulty is that the PNs are running asynchronously so that a chosen PN may not be prepared to receive a broadcast because it is lagging behind the sending process. These problems made the "bare bus" unusable for broadcasting.

The basic idea of 1-to-N DMA is quite simple. It can be viewed as a conventional DMA that has multiple destinations. When the sending process broadcasts a message, the selected PNs are coerced to perform the DMA operation in parallel.

Before we look at the 1-to-N DMA mechanism, the two modes of a typical DMAC should be understood first. Some modern DMACs have more than one channels, Each channel of the DMAC can be independently programmed into **explicit mode** or **implicit mode**. The implicit mode, also called **memory-device mode**, means that the "peripheral device" is already available on the bus and no addressing is needed. Thus, the DMAC only issues one address for the memory. Conversely, the explicit mode, also called **memory-memory mode**, means the "peripheral device" must be addressed explicitly, so the DMAC has to issue 2 addresses in 2 bus cycles. Figure 4.11 shows the two modes of a typical DMAC.

4.4.3.2 1-to-N DMA operation

Every PN has a buffer area for communication. Without loss of generality, message length is limited to L bytes (a block) and the buffer area can hold N messages. The buffer area is treated as a **FIFO (First In First Out) circular queue.**

When a message is to be broadcasted, the sender first mails the header to the MPC as before. Then with the assistance from the MPC, a 1-to-N DMA is carried out and the message is transferred to all potential receiving PNs. The message is stored into the circular buffer of each receiver (recall that for a long point-to-point message, a DMA brings the message to a target location explicitly specified by the receiving process).

When a receiving PN really wants to accept a message, it can read the message header from the MPC, which will inform it that the message body is already in its message buffer. In other words, messages can be received in advance. Figure 4.12 highlights the roles of the 1-to-N DMA and the MPC. This figure is similar to figure 4.10 so explanation is not necessary. Note the event sequence is different this time.

The 1-to-N DMA operation via the global bus is feasible only if all the receivers

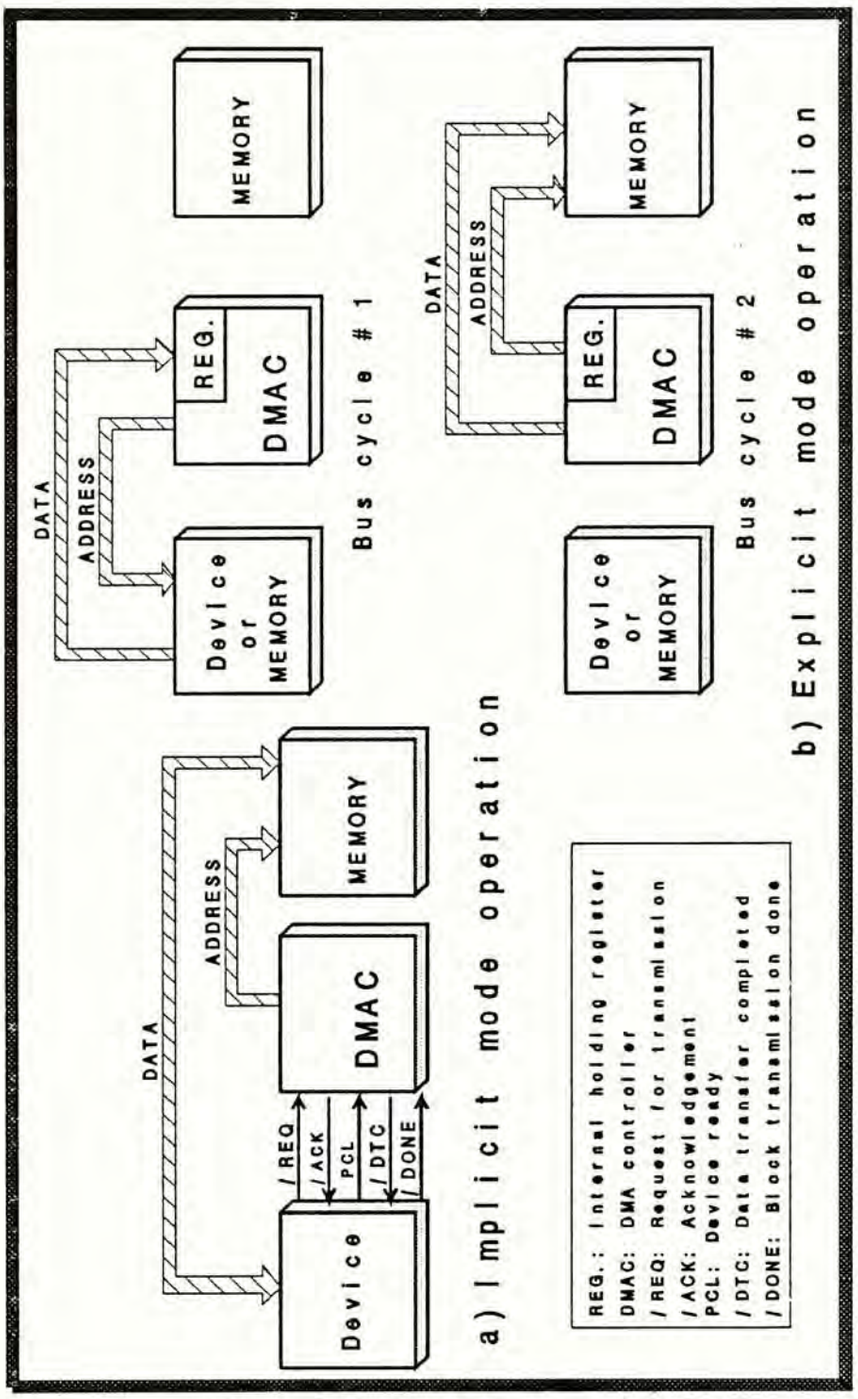


Figure 4.11 The two modes of a DMAC.

can pick up the data driven onto the bus by the sender in an orderly manner. The key is that **every party in the deal has the illusion that it is communicating with a single partner**. From the sender's point of view, the message is transferred to a **virtual device** connected to the global bus. From the receiver's point of view, a virtual device on the global bus supplies data to it. With little modification to the conventional hardware configuration for DMA, a mechanism is designed to overlap these two scenes and to ensure that every critical event is synchronized.

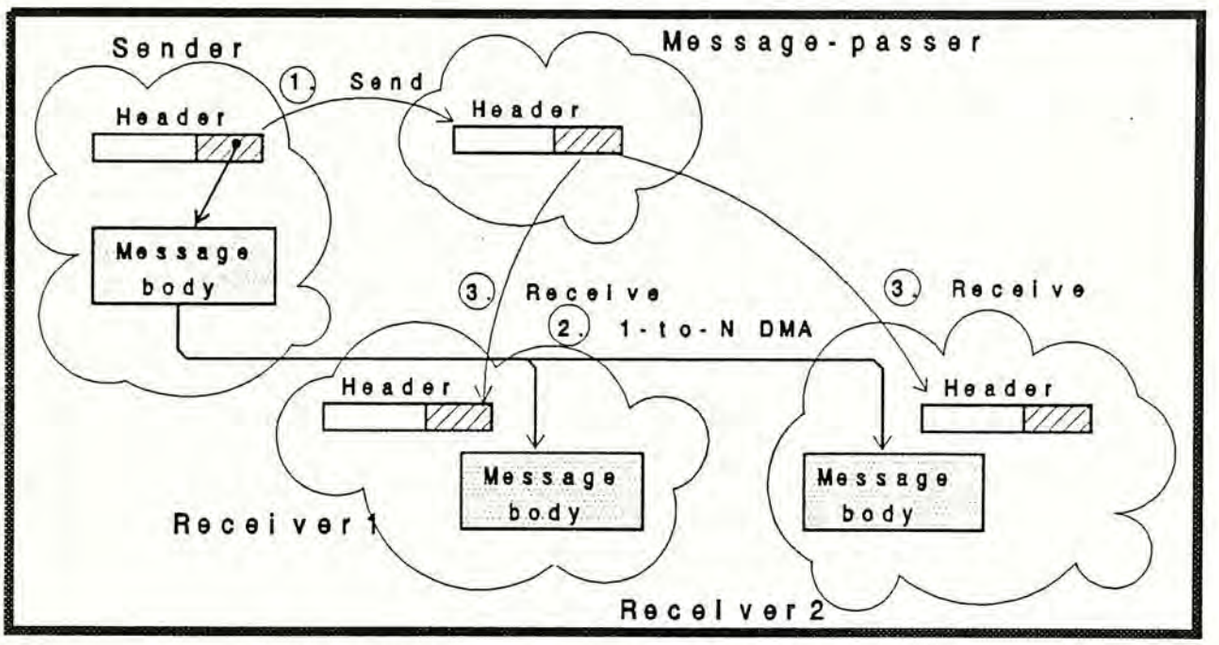


Figure 4.12 Roles of the 1-to-N DMA and the MPC.

In principle, the DMAC of the sender simply puts a data word onto the shared bus while all receivers grab the data as if it is performing a simple 1-to-1 DMA. So a 1-to-N DMA is the parallel collective result of (N + 1) normal DMAs. After a word has been transferred via the global bus, the buffer addresses for the sender's and the receiver's memories have to be incremented by the DMACs automatically. In this case, the DMAC is operating in the device-memory mode. Since most common DMACs have two or more channels, it is convenient to program one channel into the memory-memory mode and one channel into the device-memory mode.

By adapting common DMACs, only a little control logic has to be added on each PN. This control logic coordinates the operation of the DMAC and the memory, and synchronize the sender and the receivers. Its ultimate goal is to create the illusion of a virtual device with which the sender and the receivers communicate. Figure 4.13 is a scenario of the 1-to-N DMA process.

One necessary condition for performing the 1-to-N DMA is:

> ***All the potential receivers must be halted and forced to receive the broadcast message before they can resume their works.***

Since all parties are running asynchronously, a receiver chosen by the sender

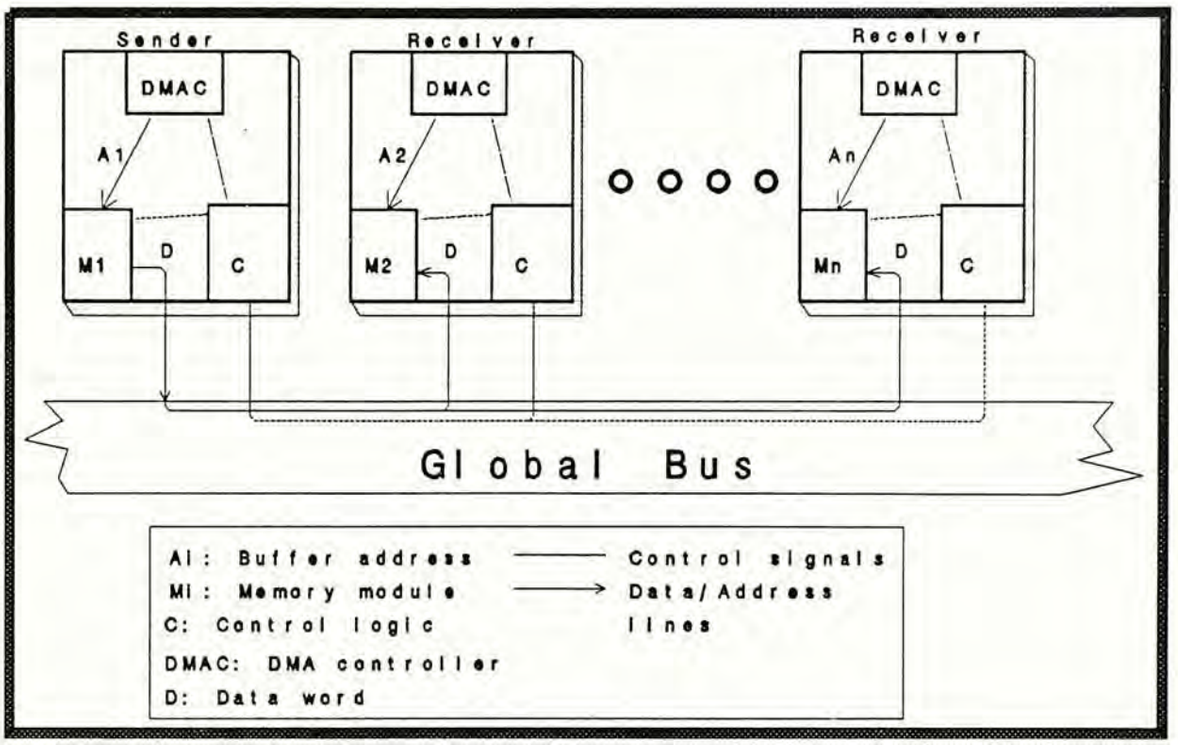


Figure 4.13 A scenario of the 1-to-N DMA process.

may not be prepared for the message. As only one transmission across the shared bus is allowed, all parties must participate in the 1-to-N DMA simultaneously, no matter a receiver is ready or not.

In order to ensure the broadcast can be done without intervention of global bus requests due to the other PNs, cycle stealing mode DMA is NOT used (similar to the case of point-to-point messages). In case of receive buffer full or other problems, the PIDs of the questionable receivers are recorded and reported to the sender later.

4.4.3.3 Merits and demerits of 1-to-N DMA

Table 4.2 contrasts the merits and demerits of the 1-to-N DMA mechanism. The speed up of this mechanism depends partly on the number of receivers in a broadcast, and partly on the message length. For a typical distributed global memory system, about 10-16 PNs can be attached to the same bus depending ont the granularity of the problem [Paker83]. So a ten-fold speed up is achievable.

Advantages	Disadvantages
Greatly speed up the broadcast.	Additional communication logic and synchronization signals necessary.
The cost is low since conventional DMACs, which are used for point-to-point messages too, can carry out a large part of the operations.	Additional receive buffers on each PN required.
Completely transparent to the application program.	Hardware becomes more complicated.

Table 4.2 Merits and demerits of the 1-to-N DMA mechanism.

Although some PNs may refuse the broadcast message, it does not lead to a waste of time because the message goes through the global bus just once, independent of the number of receivers. The PN that ignores the message may be slightly delayed due to the data transfer, but in general this rarely happens. We implicitly assumed it is very unlikely that no one wants to receive the message. Only in this case the 1-to-N DMA will be a waste of time. For the receivers, it is a matter of bringing forward the required DMA operation.

Comparing with conventional broadcasting systems such as Ethernet [MetRo76], our mechanism is much faster. The speed of such conventional systems is tightly bounded by the bandwidth of the serial links. In contrast, most wide buses (32-bits, say) can attain much higher transfer rate. The broadcasting power of our prototype system is essentially the same as that of a star topology. In general, the star topology is more efficient for broadcasting compared with other topologies such as the token-wing [IEEE83].

Another advantage of the 1-to-N DMA mechanism over its conventional counterparts is the absence of message arrival interrupt for a non-receiver (who is not a potential receiver recommended by the sender). This simplifies the control logic and keeps the disturbance of a broadcast to the lowest level. Unauthorized receivers cannot listen to the broadcast so security is enforced.

4.5 Summary

As it is always difficult to separate the software and hardware aspects of a computer system, I prefer to present an outline of our workstation architecture before we start the discussion on the software programming model of SM3. Then we explained the hardware design and the associated rationales. It is followed by the presentation of the communication protocols of SM3. In particular, a novel 1-to-N DMA mechanism suitable for SM3-like architectures was introduced. All along our discussion in this chapter, I have tried my best to present only the implementation independent aspects of the software and hardware features of SM3. The details of a particular implementation of SM3, and more specifically the MPC, will be presented in chapter 5.

CHAPTER 5
IMPLEMENTATION ISSUES OF SM3

5.1 The shared bus - VMEbus

5.1.1 Why VMEbus

A number of criterions have to be considered when choosing a shared bus for a multiprocessor system. Some typical issues are:

- ease of interfacing
- components support
- compatibility
- speed and mode of operation (eg. synchronous or asynchronous)
- cost and availability
- multiprocessor support
- expansibility

In the current prototype SM3, we selected the **VMEbus** as the backbone of the workstation. The VMEbus specification was firstly developed by Motorola, Mostek, and Signetics/Philips. This VMEbus project is described in [Fisch84]. It is an asynchronous bus having 3 available configurations: 8-, 16-, and 32-bits. The highest bandwidth is 24 Mbytes/sec. It has a master/slave asynchronous non-multiplexed data transfer structure so interfacing is easy and control is simple. Since its introduction in 1981, the bus have been accepted by more than 100 manufacturers worldwide so compatibility and components support are excellent. It has become an IEEE (P1014) and IEC standard.

The signals of the VMEbus are simple and well defined. The VMEbus is suitable for building multiprocessor systems. In fact, the **VMSbus** which supports serial communication is defined as a sub-bus of the VMEbus for multiprocessor

systems. For an introduction to the VMEbus and VMSbus please see [Motor84].

If the VMEbus was not available, another good choice for building bus-based multiprocessor systems is the MultiBus. It is a widely recognized bus, designed to suit for multiprocessor systems. While the MC680X0 family uses VMEbus, the i80X86 family employs MultiBus.

5.1.2 Customizing the VMEbus

In the VMEbus, there are 64 user-defined pins so it is easy to customize the bus for special functionalities. It is a common practice to customize industrial standard buses for special applications. Development time and cost can be cut down while compatibility is partially preserved. An example in [Bybee89] shows how the VMEbus can be adapted for a bus-based multiprocessor graphics system using these pins. To support message-passing by hardware and the MPC, SM3 also uses the 64-user defined pins so we shall have a closer look at it.

Physically, the VMEbus consists of two 96-pin connectors like the one shown in figure 5.1. They are called P1 and P2 respectively. All the pins of P1 are defined, while row a and c of P2 are not defined. Since there are 32 pins in a row, totally there are 64 undefined pins. SM3 uses them for conveying handshaking signals.

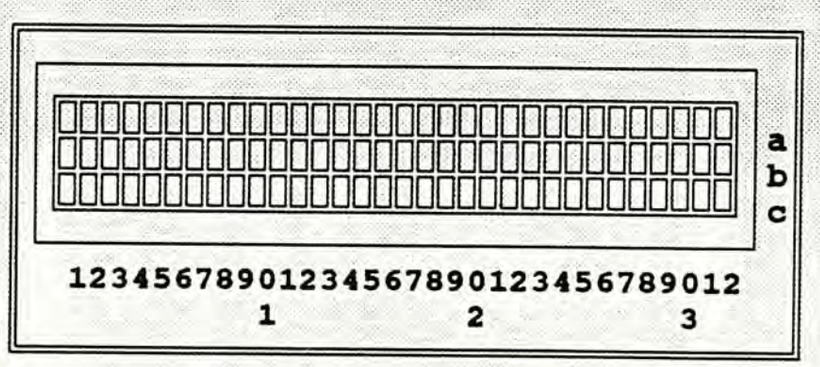


Figure 5.1 A 3-row, 96-pin VMEbus connector.

5.2 The host machine

Obviously, the choice of a suitable host machine for SM3 is closely connected to the choice of the system bus. For convenience, the host machine must be plug-compatible with the VMEbus. The choice of the microprocessor on the host machine partially determines the microprocessor type for the PNs because they must belong to the same family for object code compatibleness. Other important issues include: availability of a mature operating system, peripheral devices support, high-level language development environment, and diagnostic tools.

There are many VME-modules that are well designed Single Board Computers (**SBC**). A wide spectrum of peripheral devices is available for these VME-based computers. We selected a relatively new product from Motorola called MVME147SA-1 [Motor89]. A short profile of the technical aspects of this SBC is shown in figure 5.2. The presence of the SCSI implies that a great variety of peripheral devices, such as floppy disk drivers, fixed disk drivers, and tape drivers are immediately accessible. The CPU from the MC68000 family offers us many choices for the processors on the PNs and the MPC to satisfies our requirement for easy object code compatibility. A key feature is the VMEbus controller capability. Slaves can be plugged onto the bus directly without extra interface adapter.

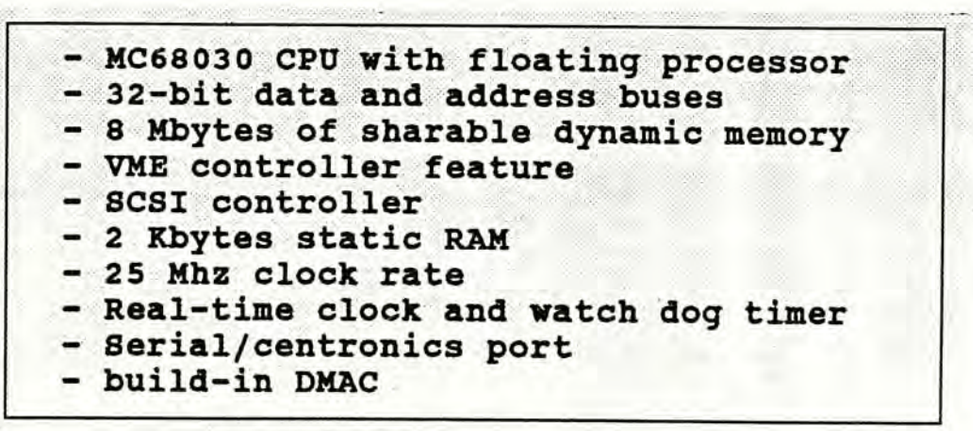


Figure 5.2 A short profile of MVME147SA-1.

As important as the high performance of this host machine, a well designed debugging package called 147BUG [Motor88b] is available in the non-volatile memory. This is a good tool for system diagnosis.

The current setting of the host machine also includes a 80 Mbytes fixed disk and a dump terminal. Figure 5.3 shows current SM3 configuration.

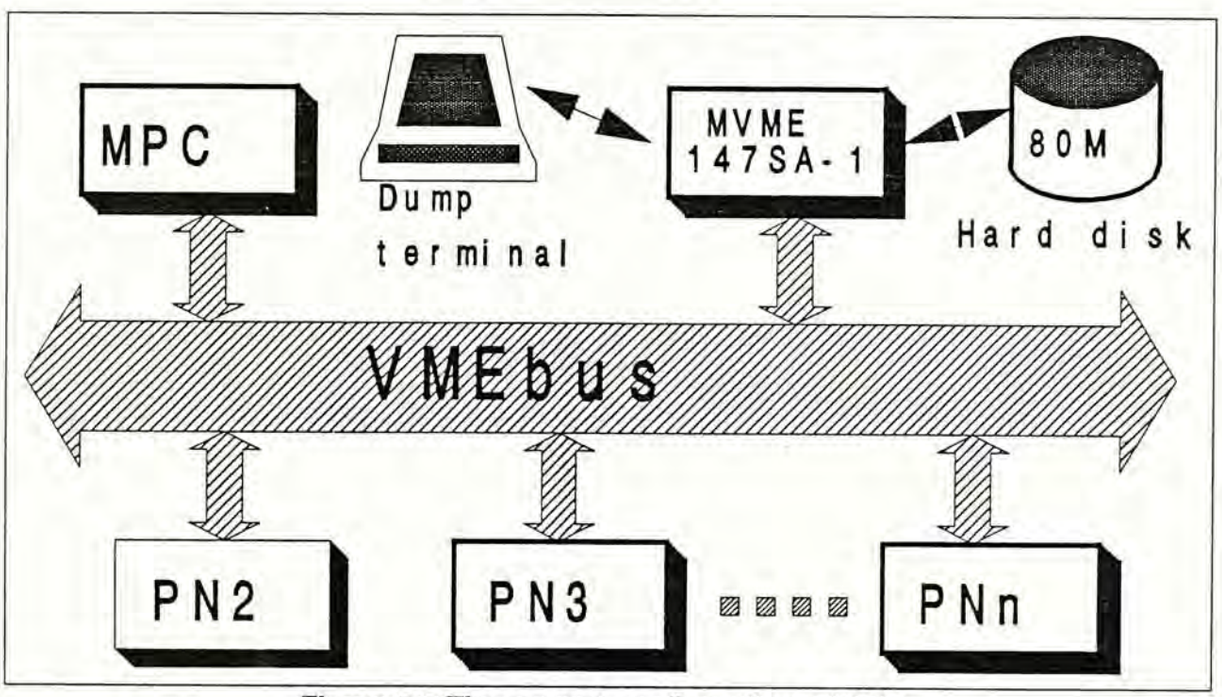


Figure 5.3 The current configuration of SM3.

5.3 Slave processor nodes

While the host machine is a commercial product, the PNs are specifically designed for SM3. Slave PNs in SM3 are basically SBCs that are attached to the VMEbus. There is no peripheral devices allocated for the slave PNs because they are designed to be pure computation machines. We shall examine each part of a PN in detail. A simplified view of a PN is shown in figure 5.4. It can be compared with the general structure of a PN shown in figure 4.8.

5.3.1 Overview of a PN

Physically, a PN module is a double-height Euro-card (VMEbus standard) pro-

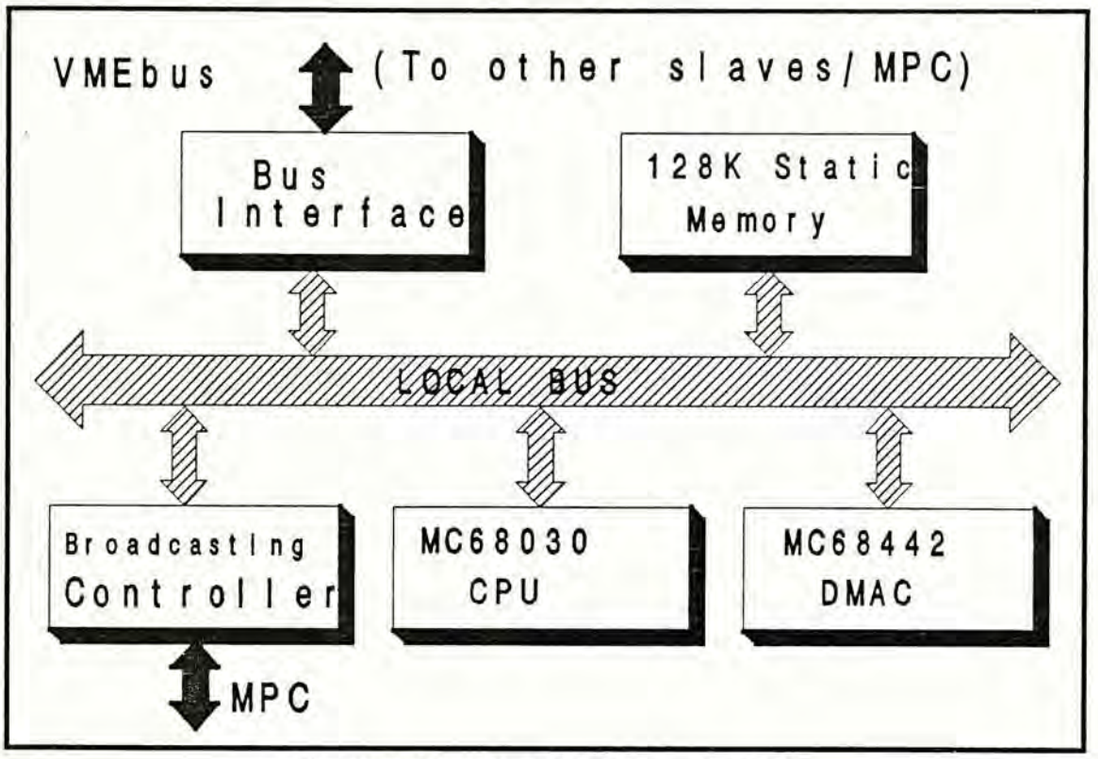


Figure 5.4 A simplified view of a PN.

cessor board with an attached extension board (Appendix C). The major components on the board is shown in figure 5.5. The major components will be described in the following sub-sections while minor components are mentioned below. Currently, the chip count of a PN is roughly 70. The Board Diagram and Schematic Circuit Diagrams can be found in Appendix A and B respectively.

Memory

There are 128 Kbytes of Static Random Access Memory (SRAM), expansible to 1 Mbytes. The use of this kind of memory is for easier circuit design and faster response time. The basic PN memory consists of four 32K x 8-bit MCM60256P10 SRAM chips. It supports byte-, word-, and longword- accesses. The address map is shown in Appendix E.

Local System Clock

A local oscillation circuity delivers a 30 MHz 50% duty cycle clock signals as the time base for every PN. This signal is bufferred and divided. The Programmable Array Logic (PAL) chips where the finite-state machines reside are driven by the 30 MHz clock and the CPU is running on the 15 MHz clock. This signal is further divided to 7.5 MHz for the DMAC, which uses a lower clock rate then

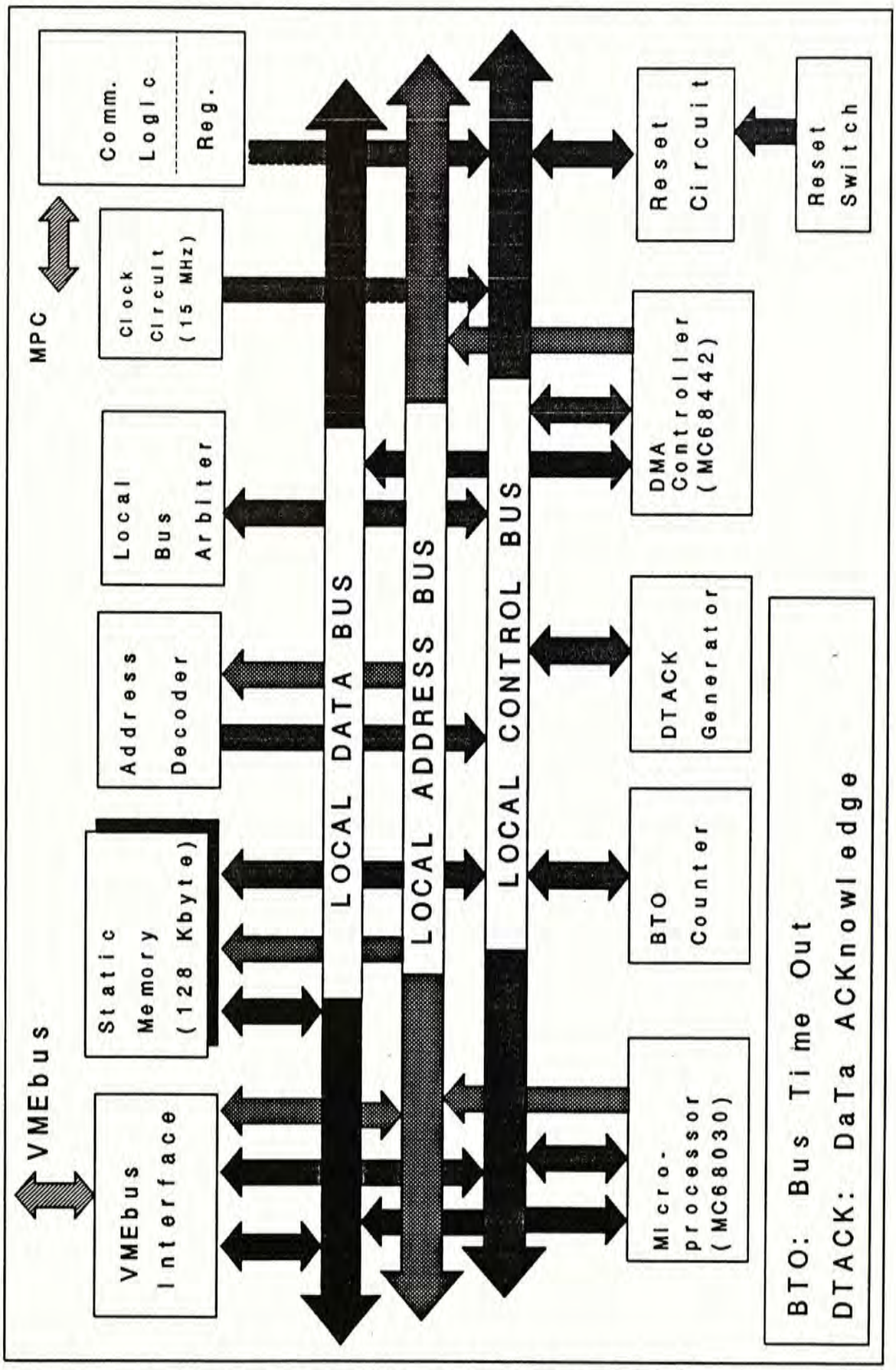


Figure 5.5 An overview of the PN architecture.

the CPU. Figure 5.6 summarizes the clock system.

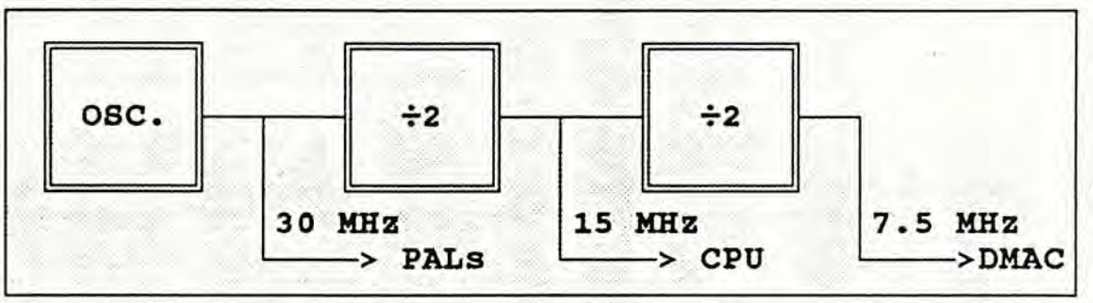


Figure 5.6 An overview of the clock generator.

Reset

The reset circuit and switch provides local reset to make diagnosis and control easier. The local reset circuit is responsible for system power up reset and initialization.

Bus time out (BTO) counter

Since bus accesses are asynchronous, a separate watch-dog timer is required to count the time-out period. When the time-out period of a bus access expires, no matter that is a local or an off-board one, a Bus Error (/BERR) is generated to signal this event to the initiator. (From now on, active-low signals will be prefixed by '/' or suffixed by '*'.)

DTACK generation

In the MC68000 family, bus accesses must be acknowledged due to the asynchronous design. A DaTa ACKnowledgemet (/DTACK) signal serves this purpose. It should be activated when the addressed device, most likely the memory, has finished (or should have finished) the operation. It is generated by a timing circuit similar to the BTO counter just described.

Address decoder

The decoder must look at both the incoming VMEbus address and the local address bus in order to generate Board Select (/BSEL), Remote Access, and other local enable (/LOCAL) signals. The VMEbus address of a PN can be set

by a DIP switch. The address spaces, which is specific to the case of using VME147SA-1, to be decoded is shown in Appendix D and E. The address format for 1-to-N DMA is show in table 5.1. For normal memory access, the last 3 fields are merged to give a 20-bit field (1 Mbytes).

Physical PID (4-bit Dip switch)	Buffer offset (6 bits)	Tail pointer (4 bits)	Byte offset (10 bits)

Table 5.1 Global address format for the PNs of SM3.

Bus arbiter

Around the local bus, there are three active requesters. The CPU and the DMAC are local requesters while the request from the VMEbus can be view as a remote one. Since the local bus is an extension of the CPU bus, the CPU is the primary bus arbiter while the external bus arbiter handles the requests from the DMAC and the VMEbus. Figure 5.7 shows the two-level arbitration procedure. Currently, VMEbus access is given higher priority but this can be easily reversed if necessary.

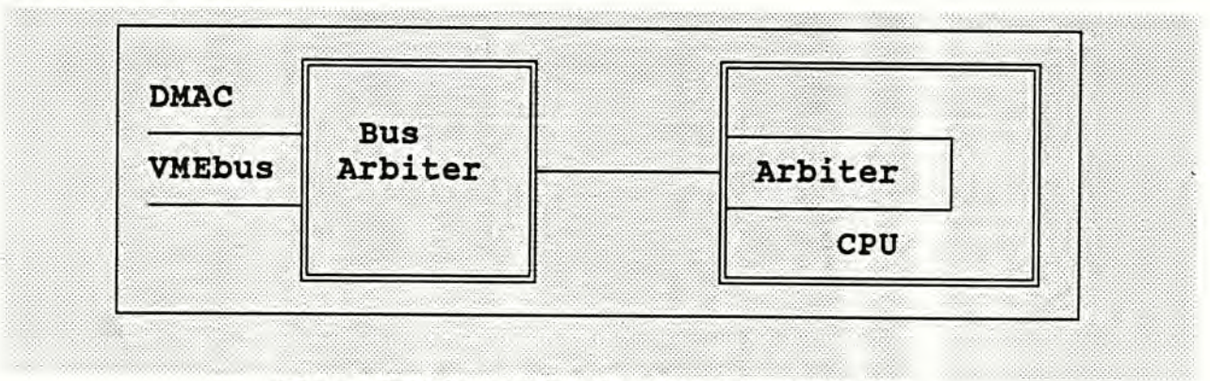


Figure 5.7 The two-level arbitration procedure.

5.3.2 The MC68030 microprocessor

Currently the MC68030 is adopted as the CPU. The MC68000 family microprocessors are elegantly designed and have good performance. New members, such as MC68020 and MC68030, have special support for multiprocessing [Beims84]. A bus arbitration scheme is designed to allow several M68000 bus masters to share the same bus (in our case the DMAC and the CPU share the local bus),

and 3 instructions that use interlocked bus cycles for inter-processor communication in a loosely-coupled system is available. Moreover, there are 5 instructions that utilize the M68000 Family Coprocessor Interface to communicate with tightly coupled coprocessors. However, SM3 does not benefit from these features directly.

Another good reason for using the MC68000 family is that we can choose microprocessors from a wide spectrum of processors. Code compatibility is virtually maintained from MC68000 to MC68040. This matches our goal described in chapter 4.

Due to their popularity, the MC68020 and MC68030 are well documented. Informative books such as [Harma89, JaBaP88] are extremely helpful to the design work. The manufacturer also publishes supplementary information, for example, "The M68000 family reference" [Motor88a].

A great number of projects employed processors from the M68000 family so we can gain valuable experience by studying these projects. The SUPRENUM project described in chapter 2 is a good example. As briefed in [Pount88], a VMEbus-based system with 12 MC68030 boards having 4 Mbytes each is constructed for parallel operating system research. A multi-microprocessor system using M68000s for image processing and pattern recognition, which can be configured into SIMD or MIMD mode is introduced in [KuSiP82]. More examples can be found in [AthSi88].

Although the local bus is an extension of the CPU bus, buffers are required to isolate the CPU and DMAC when memory accesses from the VMEbus are serviced.

5.3.3 The DMAC M68442

Once the CPU is selected, there are not many choices for a DMAC (Direct

Memory Access Controller). For cost and availability reasons, we adopted MC68442. This is a Dual-channel DMAC (as the manufacturer called DDMA). Detailed information can be found in [Motor88a]. As we described in section 4.3.3, one channel can be programmed into the implicit mode for 1-to-N DMA while the other channel operates in the explicit mode for normal DMA.

We must reiterate that the DMAC is optional if efficiency is unimportant. In SM3, memory-to-memory DMA that handles long, point-to-point messages can be replaced by simple shared-memory access without suffering from great performance drop. The effect of losing the 1-to-N DMA capability depends on the proportion of broadcast messages.

Although the chosen DMAC is also in the same family as MC68030, some interface logic is necessary when it is used with MC68030 because this chip was designed for older members in the M68000 family. Obviously, the CPU and DMAC are operating in an exclusive manner so they can shared the same row of address gates.

5.3.4 Registers

There are 3 registers for each PN. They can be viewed as a part of the communication logic. Each register is 8-bit wide currently and can be expanded to 32 bits. The addresses of these registers are shown in Appendix E while the detailed layout is shown in Appendix F. All registers can be reached from a remote processor like a normal shared-memory location.

PN Status Register (PNSR)

A process running on a PN can monitor several hardware signals by accessing this read-only register. Signals such as /BFn (Buffer Full for the n-th PN) and /VGRANT (a local signal VMEbus GRANT) are localized PN signals. They are included in the PNSR for diagnostic purpose. Signal /MPCRDY (MPC ReaDY) can help to reduce shared bus and MPC usage by the method explained in

section 3.3.2.2. Hand-shaking signals for carrying out the 1-to-N DMA efficiently are shown in table 5.2.

Signal	Meaning
GPCL	Global signal PCL for the DMAC
/BROADCAST	BROADCASTing a long message
/SYNCDMA	SYNChronization signal for 1-to-N DMA
/BCST	BroadCast STart

Table 5.2 Hand-shaking signals in the PNSR.

PN Control Register (PNCR)

This write-only register allows the software to set values for physical signals. /BCEND (BroadCast ENDed) is a system-wide signal while /VMESEL2 (VME SELect, for keeping the VMEbus mastership) is a local signal. More signals can be added for diagnostic and monitoring purpose.

Buffer Pointer Register (BPR)

BPR is a read-write register which is separated into two 4-bit nibbles. They are the head and tail pointers to the 16-block circular message queue (will be introduced later in this chapter) for 1-to-N DMA. Buffer full condition is generated by comparing the two pointers using hardware logic.

5.3.5 Shared-bus interface

Apart from gating between the VMEbus and the local bus, this bus interface must be also responsible for VMEbus hand-shaking signals. For instance, the VMEbus request acknowledgement originated from the bus controller propagates in a daisy chain. The bus interface must participate in this operation. The VME signals that must be asserted by a PN when it becomes a VME-master or -slave are also generated by the state machines in this interface.

Besides, the bus interface is responsible for bus conversion work. The local bus

is not completely compatible with the VMEbus because it is designed for older members in the M68000 family. Data conversion is necessary in order to change the signals of the local bus, which is an extension of the CPU (MC68030 is relatively new member in the family) bus, into VME signals.

5.3.6 Communication logic

Actually this part is active only when the PN is performing 1-to-N DMA for broadcasting messages. It cooperates with the hardware logic of the MPC via the communication sub-bus to create the illusion of a memory-device DMA. The majority of this logic is implemented in PAL. There are roughly two parts, one for broadcasting and one for receiving. The functionality of this logic is best introduced in section 5.5 where the implementation of the communication protocols are presented.

5.4 The MPC

Although the MPC is more sophisticated than a PN concerning the functional complexity, the hardware architecture of the MPC is even simpler then a PN. Since we have made a trade-off between speed and flexibility, many functions that can be implement by custom hardware logic are reserved for the software. It is reasonable for such a prototype system. Figure 5.8 depicts the MPC architecture using functional blocks. This figure should be compared with the PN architecture shown in figure 5.4.

5.4.1 Overview of the MPC

The MPC module is also a double-height Euro-card circuit board with an extension board. Its physical size and floor plan can be found in Appendix C. Circuit diagram of the MPC can be found in Appendix B. Currently, the chip count of the MPC is roughly 60. The major functional blocks of the MPC architecture is shown in figure 5.9. This figure should be compared with figure

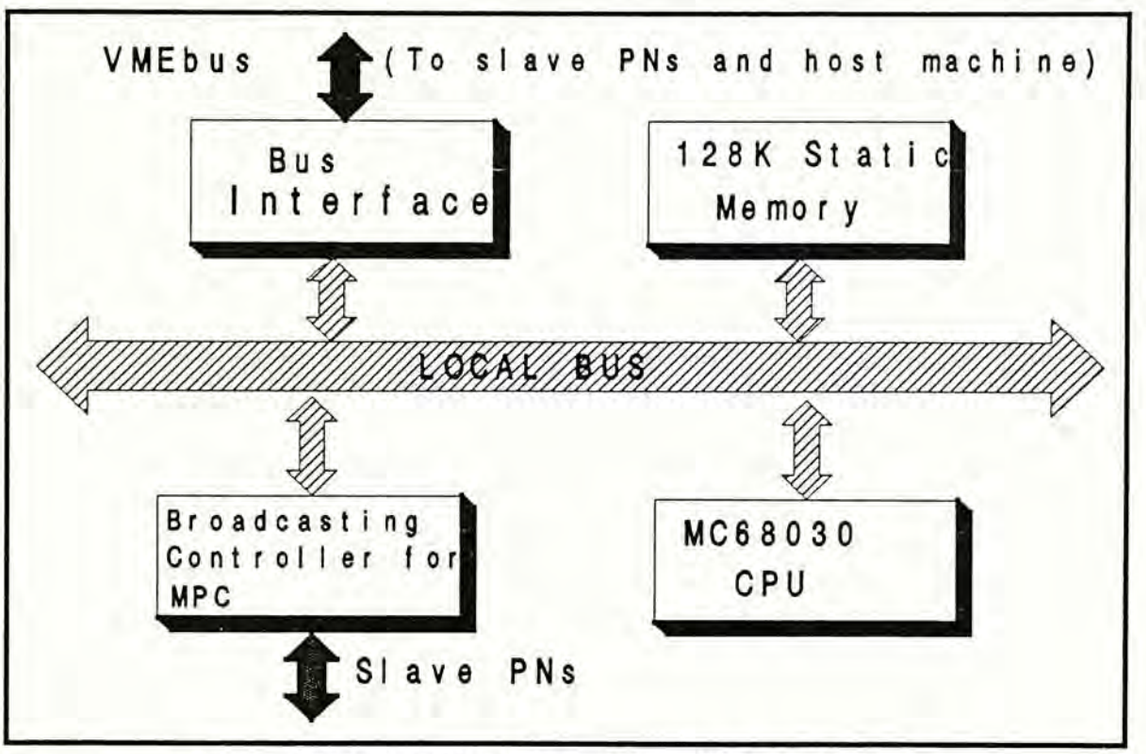


Figure 5.8 A simplified view of the MPC architecture.

5.5 which is for a PN.

Note that a DMA controller is not necessary for the MPC because it will not initiate message transfers. The VMEbus Interface, Static Memory, Microprocessor, BTO Counter, DTACK Generator, and Reset Circuit are identical to and that of a PN. For uniformity and further expansion, the Local Bus Arbiter for a PN is used but with the input request line from the DMAC disabled. The Address Decoder is modified because the MPC has a slightly different register file definition. This will be discussed in the next sub-section. The address areas to be decoded can be found in Appendix D and E.

5.4.2 Registers

There are 4 special purpose registers on the MPC. They are allocated to nearly the same area as their counterparts on slave PNs. Their layout is shown in Appendix F. These registers are closely related to the broadcasting logic. They can be reached from the VMEbus like shared-memory.

MPC Status Register (MPCSR)

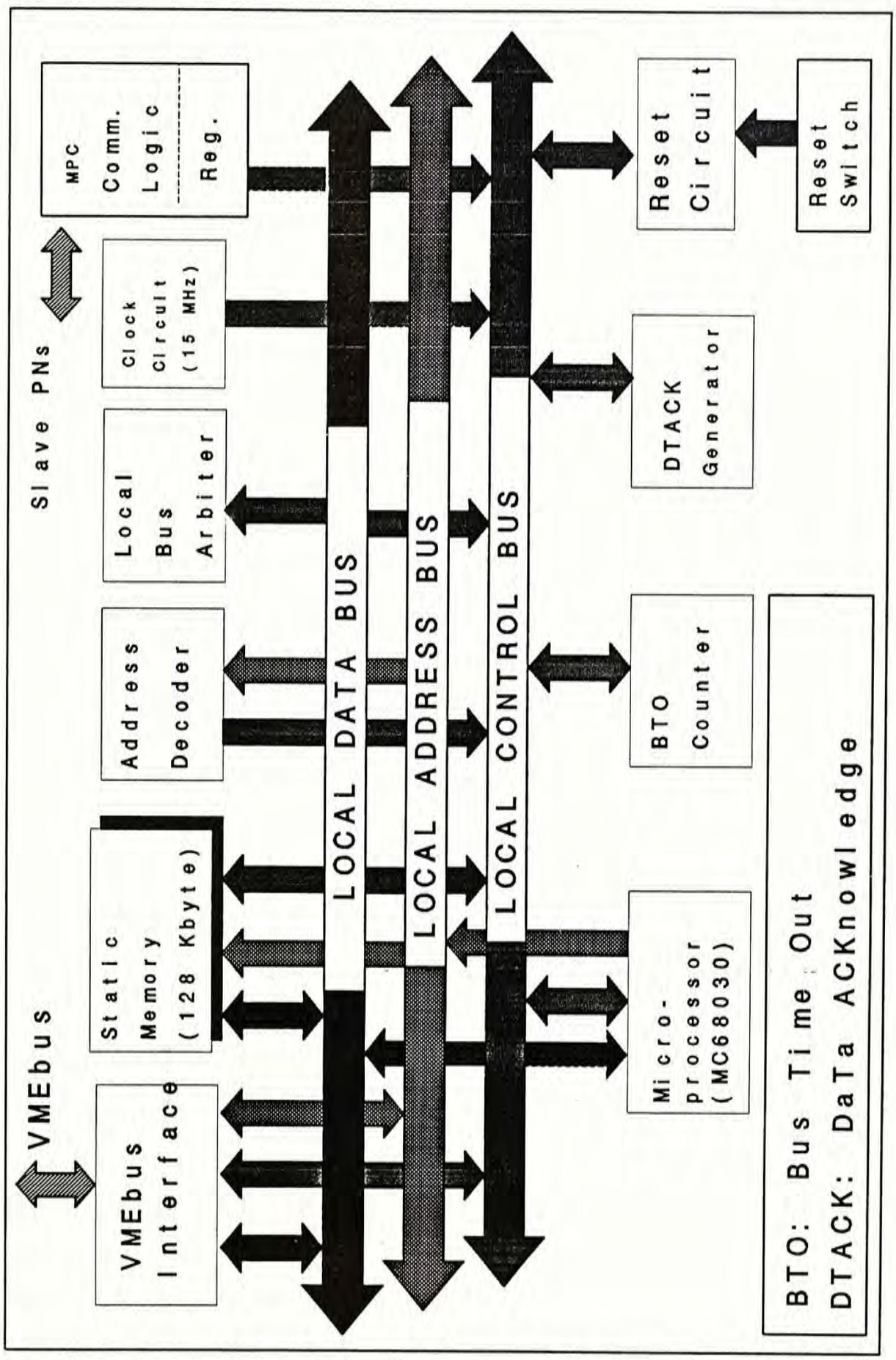


Figure 5.9 The architecture of the MPC.

This read-only register corresponds to the PNSR of a PN. /BCEND (BroadCast ENDed) is asserted by the PNs to indicate the completion of a broadcast. GPCL, /BROADCAST, SYNCDMA, /BCST, and /VGRANT are the same signals found in the PNSR of a PN.

MPC Control Register (MPCCR)

Similar to the PNCR, control signals are injected into the communication sub-bus through this write-only register by the MPC. /BROADCAST (1-to-N DMA for BROADCASTing in progress) and /MPCRDY (signal a blocked process that the MPC is ReaDY for competition) are for hand-shaking and efficiency enhancing respectively.

MPC Buffer Full Register (MPCBFR)

This read-only register reflects the 1-to-N DMA circular buffer queue status of all the PNs. The status is represented by the signals /BFn (Buffer Full of PN number n). Each bit is for one PN so there are at most 8 PNs.

Halt Register (HALTR)

Each bit of this write-only register is connected to the HALT pin circuity of a PN. The control process, either the MPC or the root process, may use this register to temporary stop any processor selectively. Up to 8 PNs, including the MPC can be handled. The current prototype system does not provide special protection for writing to this and other control registers. However, protection from unauthorized writes by user processes is desirable for a later version.

5.4.3 Communication logic

We will learn from this section that the MPC needs very little control logic for supporting 1-to-N DMA. Although the control logic is easy to implement in hardware, we still deliberately keep it in the software for flexibility and easy modification in this prototype system, although full speed cannot be achieved. By software control, we mean program-controlled logic using the control

registers. Thus, the communication logic of the MPC is only a register file plus some interface circuities.

5.5 Protocol implementation

The implementation strategy of communication protocols discussed in section 4.4 depends strongly on the underlying architecture. Before the protocols are described in detail, a simplified model of SM3 is included in figure 5.10 which also summarizes the design philosophy. Note the difference the PNs, the MPC and the host machine.

The mechanisms that handle different sort of messages are shown in table 4.1. Evidently short messages are easier to handle because they do not occupy much buffer space and do not cause series system bus hold up. Only long messages need special treatment. The following discussion will emphasis on long messages.

5.5.1 Point-to-point messages

Recall that whenever a process wants to send or receive a message, the SendMessage or the ReceiveMessage primitive is invoked. The sequence of events had been described in chapter 4. It is suitable to summarize the function of the two primitives at this moment. Table 5.3 outlines the operations of the two primitives.

Communication primitives are written in the form of device drivers for the communication channel. Apart from SendMessage and ReceiveMessage, other desirable primitives may be included. For instance, IMQ status enquiry and flushing.

When ReceiveMessage finds that an incoming message is a long point-to-point one, the DMAC channel working in explicit mode is initialized with the source address, destination address, and message length. The DMAC will interrupt the CPU upon completion. Then the application program can read the message

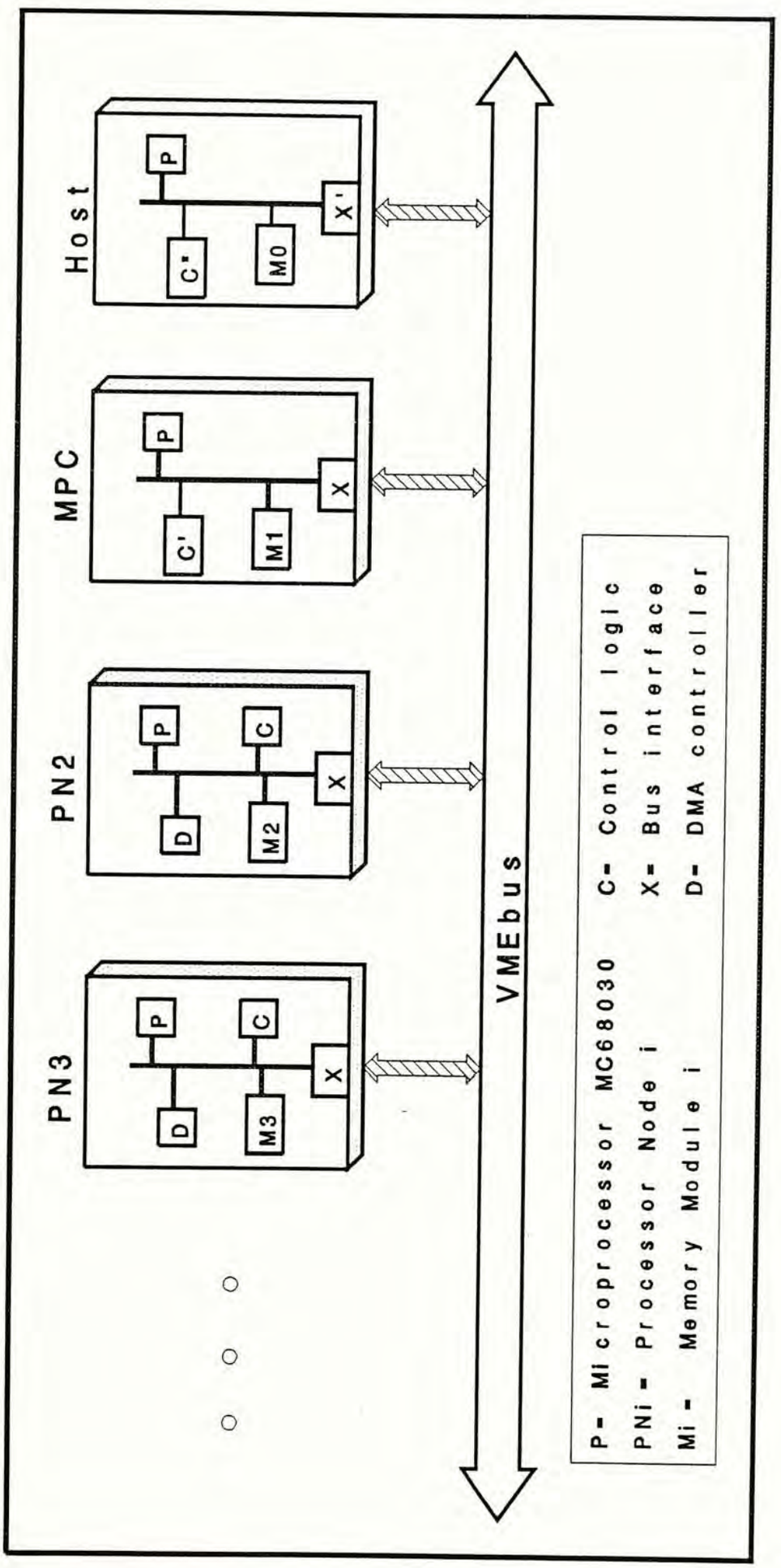


Figure 5.10 Simplified view of the SM3 prototype system.

	Sender	Receiver
1.	Invoke the SendMessage primitive.	Call the ReceiveMessage primitive.
2.	If an acknowledgement is not required, the primitive returns.	If a new message is available, read the message header and the primitive returns.
3.	Wait for the acknowledgement from the MPC.	If this is a non-blocking receive, primitive returns.
4.	Primitive returns.	Wait until a new message arrives.
5.		Read the message header and the primitive returns.

Table 5.3 Summary of the primitives SendMessage and ReceiveMessage.

from the local memory of the PN. The whole protocol and its direct implementation are simple comparing with that of broadcast messages.

When this message system is actually implemented for an application, a hidden problem may show up. When the PNs in the system have performed a certain amount of computation, they may ask for more data to process or exchange results at roughly the same time due to the even distribution of tasks. Exactly one of the PNs will win and the communication requests will henceforth be serialized. The PNs will be running at shifted computation phrases. If communication cost is low compared with computing delay, there will be no serious contention.

If the first PN comes round for more message before other slaves have finished their tasks, that implies the task distribution strategy is not suitable. Appendix I discusses an experimental way to judge whether the computation overhead is too heavy for a particular task distribution plan.

5.5.2 Broadcast messages

Recall that long broadcast messages are handled by 1-to-N DMA. The protocol

for this kind of message transfers must be designed with great care because performance will suffer if there are many redundant operations as in the conventional approach. Architectural support and special software are both critical for the success of this protocol. Let us start with the special hardware required.

5.5.2.1 Circular buffer queue

In order to support 1-to-N DMA, every PN has to buffer the broadcast message itself. Hence, there is a 16-block circular queue on every PN for buffering messages as shown in figure 5.11. Two 4-bit pointers in a hardware registers point to the head and tail of the FIFO queue respectively. These two pointers are NEQ-ed (Not EQual) to give the signal /BFn (Buffer Full for the n-th processor). This signal tells the MPC that a slave processor is ready for accepting a broadcast message. The block size is arbitrary chosen to be 1 Kbytes, which is also the choice of some researchers [FinHe88]. The buffer full and empty conditions shown in the figure are designed to favor the detection of buffer full. Note that only 15 out of 16 buffer blocks are usable with this convention.

When a 1-to-N DMA transfer is initiated, the message will be place at the empty slot indicated by the tail pointer. After the transfer, the communication kernel must update the tail pointer. We will see that this is done in an Interrupt Service Routine (**ISR**). Later, when the user process calls the ReceiveMessage primitive, the head pointer is read and the oldest message is captured. The head pointer is advanced subsequently. A subtle point that I must clarify is that there is no lost update although the ReceiveMessage primitive and the ISR may access the register in an inter-locked way. It is simply because they are updating separate parts of the register (as just described) without corrupting the other part (use a bit-mask to select the active region for updating).

5.5.2.2 Participating entities

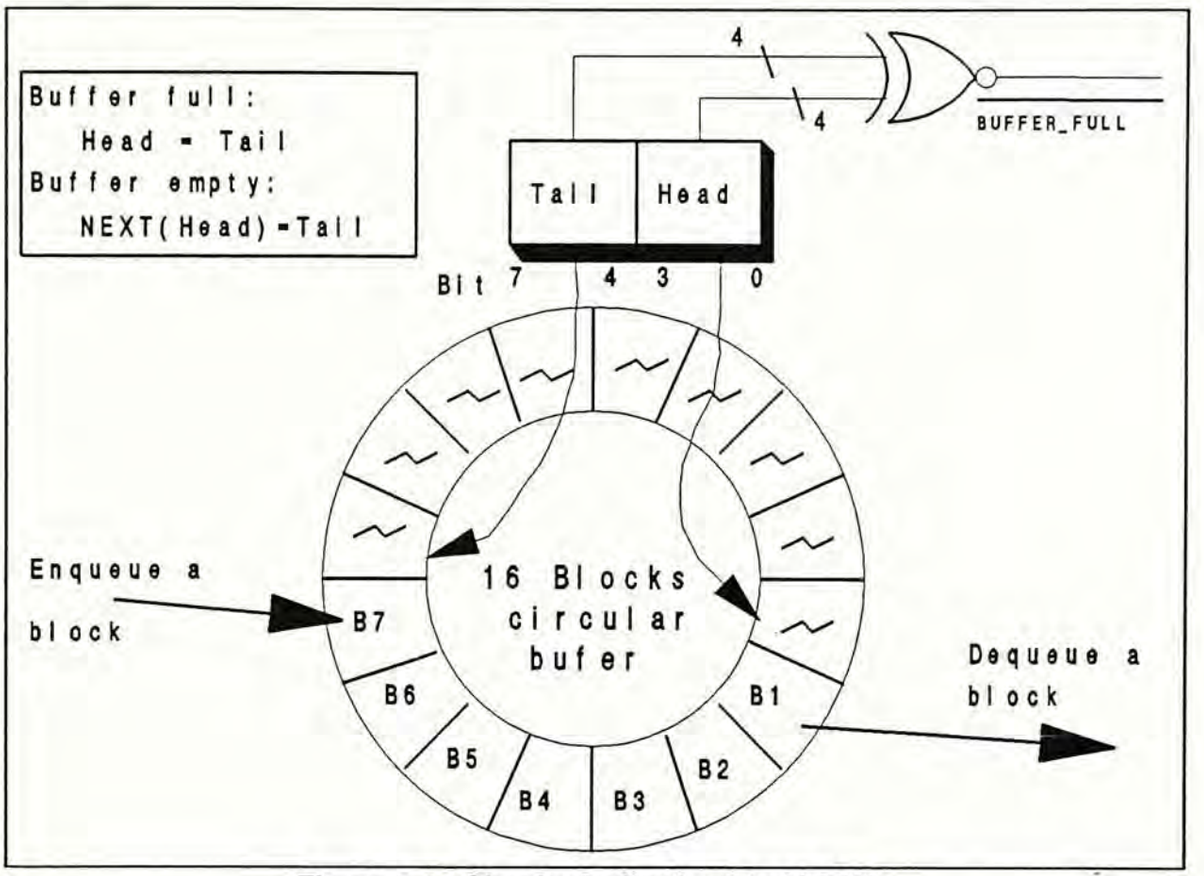


Figure 5.11 Circular buffer for 1-to-N DMA.

To delineate the scope of our future discussion, a view of the hardware entities and signals participating in the 1-to-N DMA is presented in figure 5.12. This figure can be viewed as a static summary of the protocol. For simplicity, the MPC and other possible receivers have been omitted. Interestingly, the MPC takes on a minor role in this protocol and it is not involved in the actual data transfer stage. The DMACs are active components so they are distinguished by circles.

5.5.2.3 Protocol details

The SendMessage primitive does not return to the calling process even if the non-blocking option was active once the broadcast message is longer than a predefined limit. It must cooperate with the MPC and other PNs to finish the transfer.

The protocol can be divided into 3 phrases as shown in figure 5.13. Bold broken lines separate different nodes attached to the VME bus. Normal broken lines

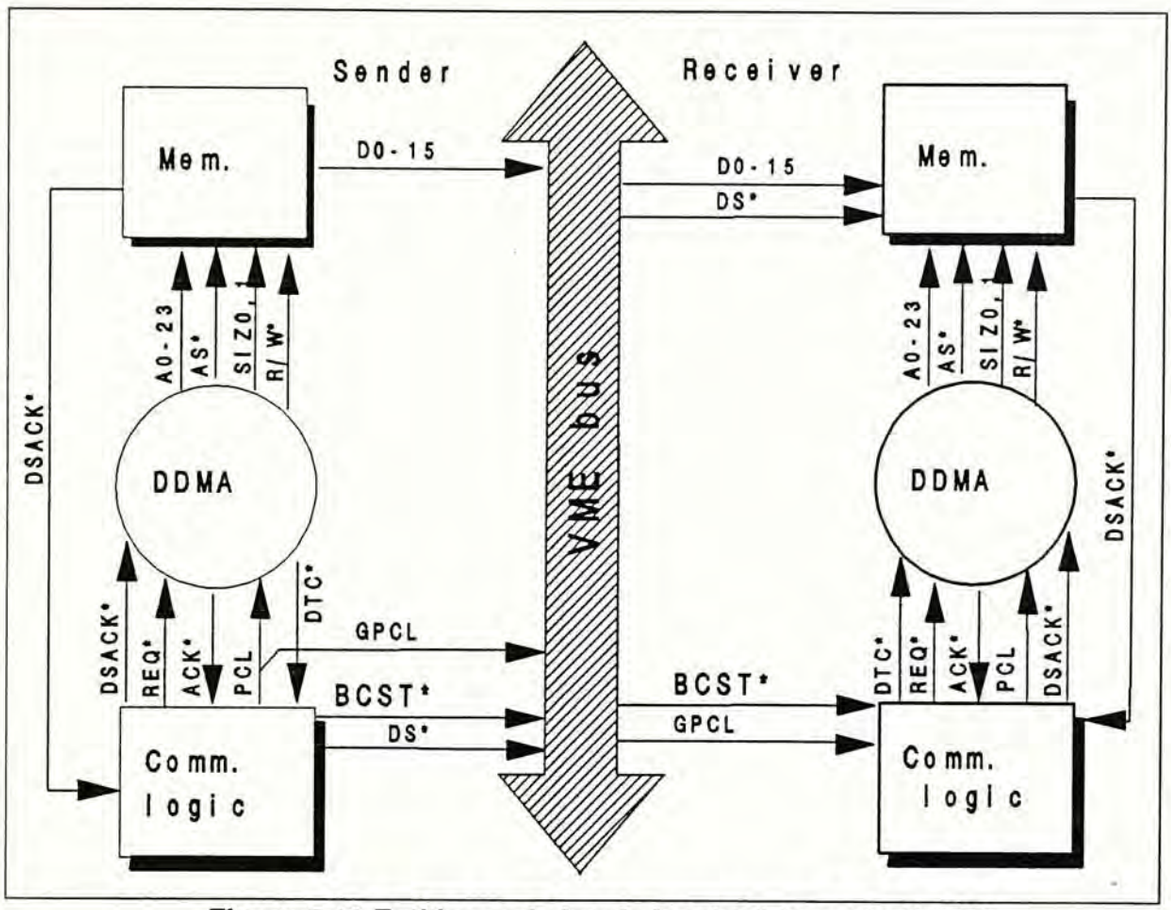


Figure 5.12 Entities and signals involved in a 1-to-N DMA.

differentiate discrete components on a node. Arrows manifest the precedence of events. The following three sub-sections are detailed descriptions of figure 5.13. The control flow between the phases is shown in figure 5.14.

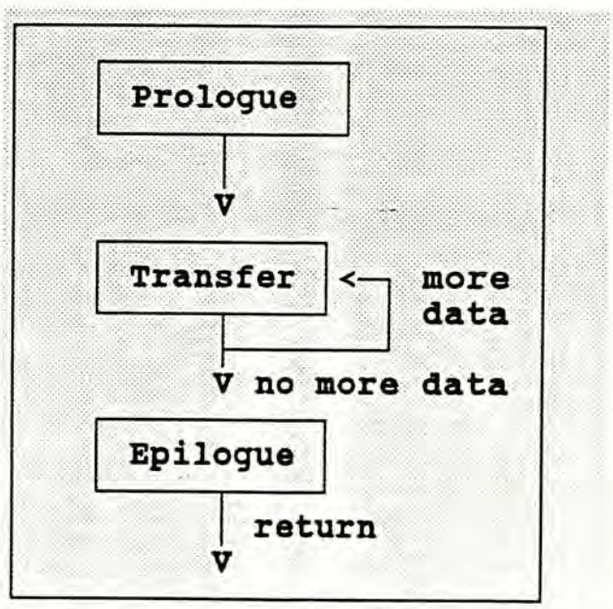


Figure 5.14 1-to-N protocol control flow.

Prologue phase

At the beginning, the sender notifies the MPC of a broadcast request. Then the sending PN, under the control of the SendMessage primitive, programs the implicit mode channel of the DMAC into the sender mode (Initially, this channel is set to receiver mode). For the sender, the transfer is from the memory to a 'device'. After gaining the mastership of the VME bus by activating the /VMESEL2

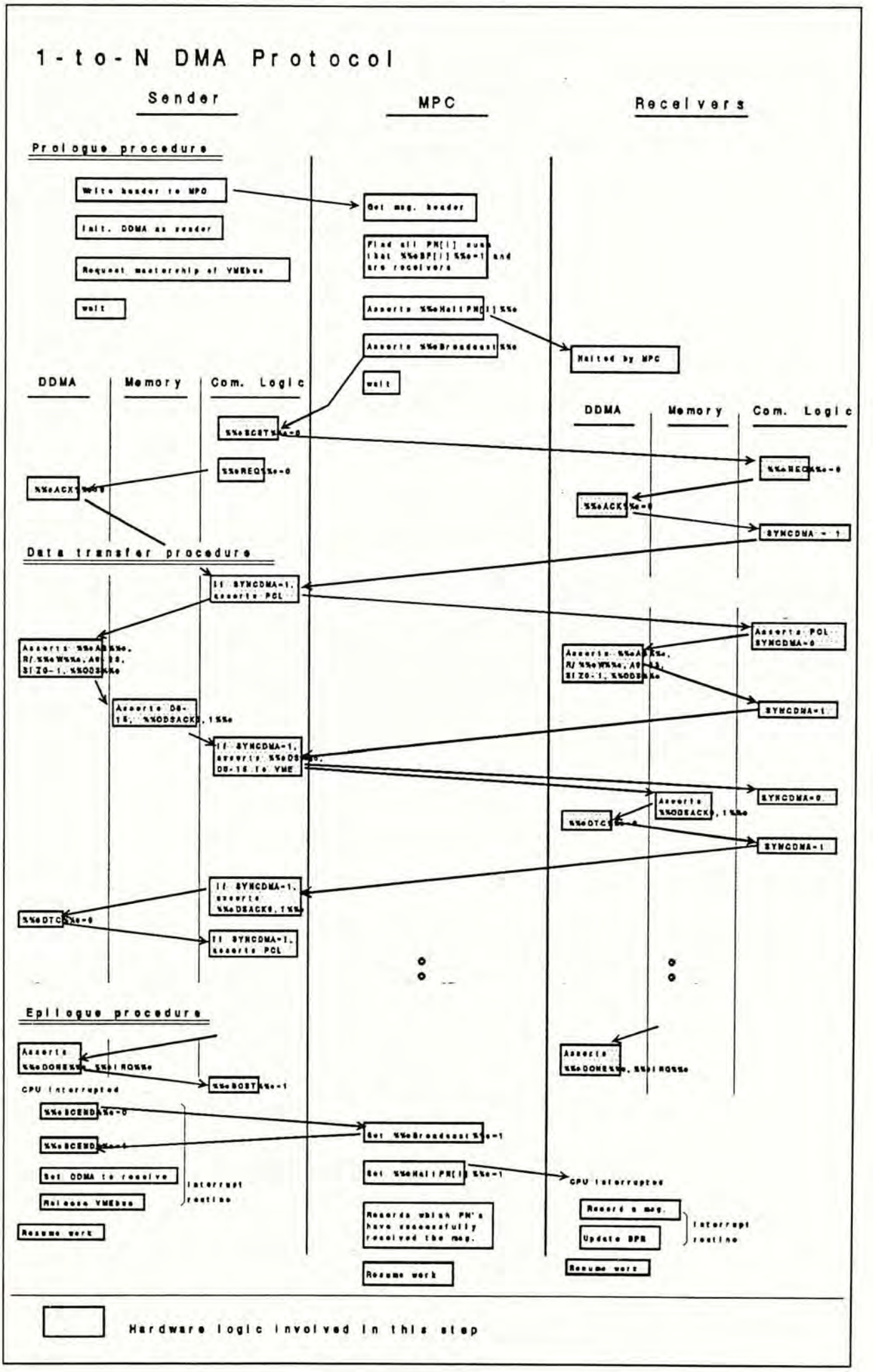


Figure 5.13 Detailed 1-to-N DMA protocol example.

signal in the PNCR, it waits for the signal /BROADCAST from the MPC. The role of the MPC in this protocol is like a traffic-light.

Upon acceptance of the broadcast request, all receivers except those being masked (sender's choice) and/or lacking buffer are halted by the MPC. Figure 5.15 shows the detail. Then, the MPC asserts /BROADCAST to instruct the sender to proceed.

After the sender has received /BROADCAST, it asserts /BCST to indicate that it has finished the preparation work and the broadcasting can commence. The sender and all receivers subsequently request the service of their DMACs immediately. The assertion of SYNCDMA by all receivers signifies the completion of this stage and tells the sender to go ahead.

```
        /HALTPNn = /MASKn and /BFn,

ie.  processor n is halted if /MASKn is 1
     and BFn is 1.
```

Figure 5.15 Halting of active receivers in 1-to-N DMA.

SYNCDMA is a tri-state signal. It is normally pulled HIGH at idle state. At any stage of the protocol, a PN sets it to LOW to indicate the PN is NOT ready, and to HIGH to show readiness. Only when all the receivers are ready will the sender find SYNCDMA to be HIGH. This signal is important for synchronization.

Transfer phase

Upon completion of the prologue phase, the sender and all receivers are ready to transfer data. The transfer phase is run repeatedly until all the data words have been transferred. The assertion of PCL/GPCL (refer to figure 4.11 and 5.13) by the sender means the 'device' is ready for the DMAC.

During a transfer cycle each DMAC involved executes either a read (sender) or a write (receivers) cycle. Signals like /AS, A0-23 are asserted accordingly. Due

to the design of the MC680X0 family of processors, the memory must return the acknowledgement signals /DSACK0 and /DSACK1, which flag the success of the memory access and confirms the word length. As soon as the sender's memory can drive a word out and all the receivers have started their write cycles (indicated by the SYNCDMA line), the sender strobes the data word out to the VMEbus using /DS. This will complete the write cycles of the receivers.

After the sender's DMAC has accepted the acknowledgement from its local memory after the read cycle, it asserts /DTC to mark the end of one transfer. The sender informs its DMAC the completion of a word transfer only after all the receivers have caught the data successfully (also flagged by the SYNCDMA line). After the DMAC of the sender has confirmed the success of the transfer by asserting /DTC, the sender's communication logic will assert GPCL again to trigger another data transfer cycle.

The whole phase described is repeated until the counters of all the DMACs, both on the sending and the receiving nodes, have reached the target. Since the message length is fixed, all counters will reach zero at the same time and all nodes are expected to stop simultaneously.

Epilogue phase

The DMACs issue /DONE after the transmission of the last word in a block. This causes the sender's communication logic to release /BCST. Although all the DMACs will interrupt their microprocessors upon completion of the transfer, only the interrupt at the sender node will be serviced because all the receivers are still halted by the MPC.

ISR of the sender informs the MPC the completion of the broadcast by issuing /BCEND. The MPC then deactivates /BROADCAST and frees all the receivers by releasing /HALTPN for each receiver. The sender's ISR then sets its DMAC back to the receiver mode and relinquishes the VMEbus. In the mean time, the MPC records the list of successful receivers in a bit-map for future use (retry).

The receivers' ISRs are now executed because their processors have been released by the MPC. The arrival of a message is marked. The tail pointer in BPR (Buffer Pointer Register) is updated to prepare for the next message.

Up to this point, the message body is available at the buffer of every potential receiver. When a receiver wants to get a message, the header will be read from the MPC. Then the receiver will recognize that the message is already residing in its buffer and no remote access is necessary. The ReceiveMessage primitive picks up from the front of the circular buffer and updates the head pointer in the BPR accordingly.

Hardware logic

To realize this protocol, the communication logic on each PN works according to the finite state machine depicted in figures 5.15 and 5.16. These figures will not be explained in detail because they follow naturally from the protocol shown in figure 5.13.

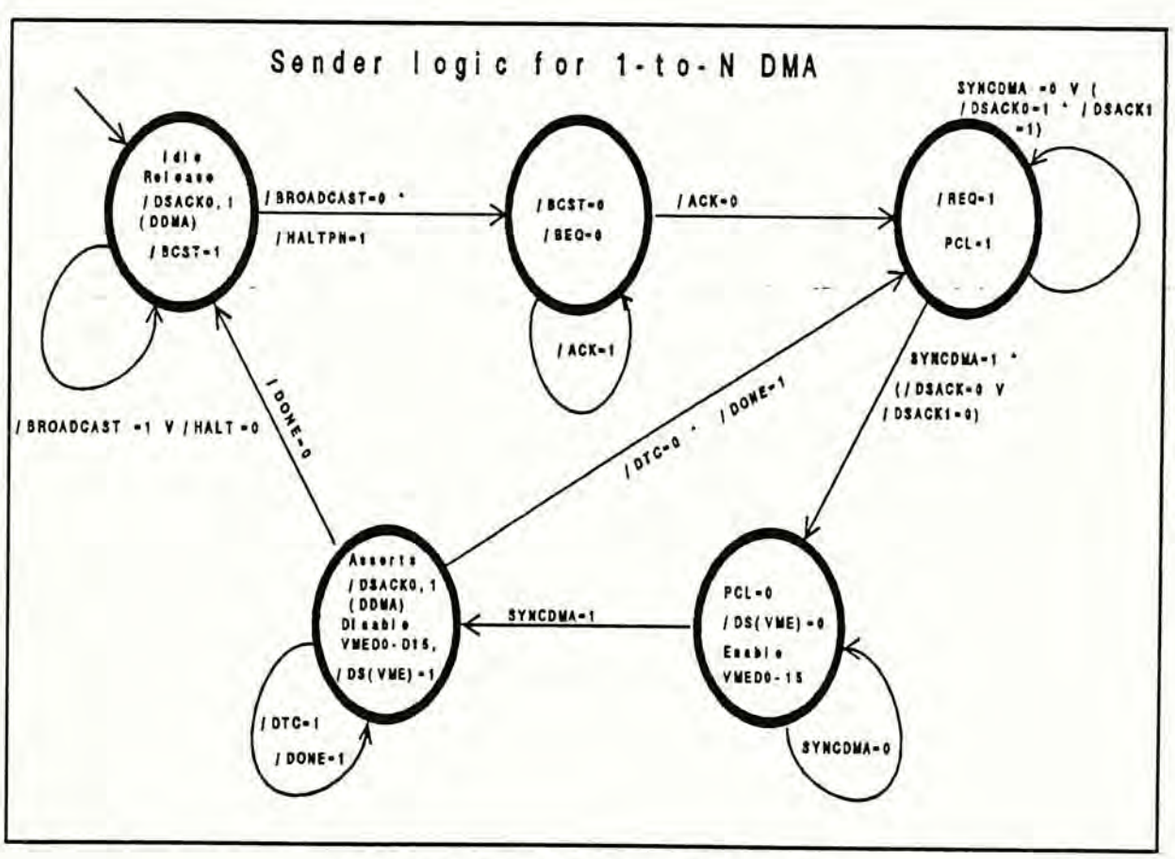


Figure 5.15 State diagram for sender.

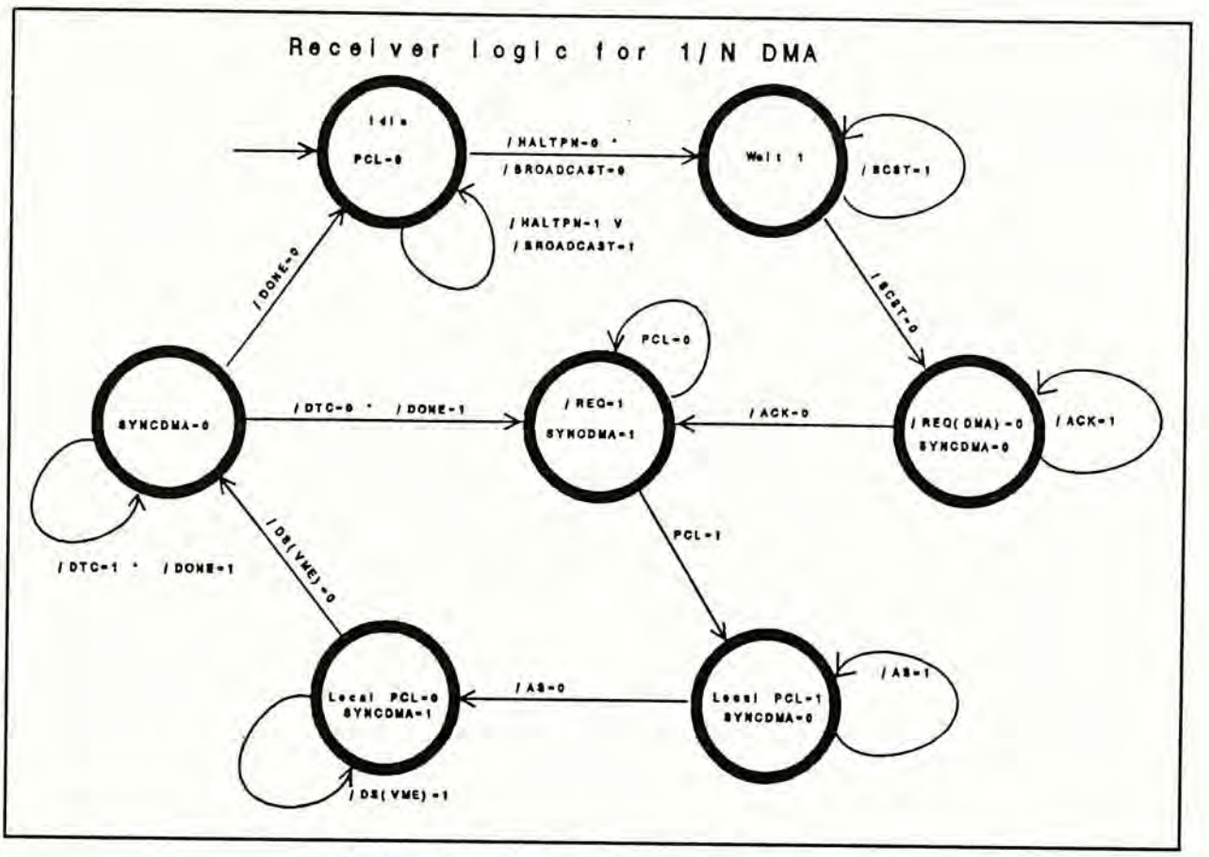


Figure 5.16 State diagram for receiver.

The logic of a PN use /HALTPN to determine its identity as a sender or receiver, given that the sender is not halted by the MPC. Signals added to the VMEbus for the purpose of hand-shaking include: /BROADCAST, /BCST, SYNCDMA, GPCL, and /BCEND. Other supplementary signals includes /MPCRDY, /BFn and /HALTPNn. All these signals are transmitted via the spare pins of the VME bus. The pin assignments are given in Appendix H.

In our prototype machine five global signals (/BROADCAST, /BCST, ...) are introduced for hand-shaking. This number varies with different hardware implementations (choice of bus and processor). The number of supplementary signals like /HALTPNn and /BFn depends on the number of slave PN's.

5.6 System start-up procedure

The start-up procedure can be divided into 2 phases. Firstly, the PNs are power up and a reset sequence is executed. Secondly, the slave PNs cannot cooperate with each other before they are properly initialized by the root process on the

host machine.

5.6.1 Power up reset of PNs

As soon as the MC68030 on a PN is powered up, it goes through a reset sequence. The program counter and stack pointer are read from the first two long words (address 00000000 - 00000007) of the memory. In order to prevent a PN from accessing these locations via the VMEbus, the first 64K bytes (00000000 - 0000FFFF) in the address space of a PN are mapped onto the locations starting at 00m00000, where m is (PID + 8). The PID can be set by DIP-switch setting. This location is just the starting point of a PN's local memory. Hence, remote access for reset vectors is eliminated. Once powered up, the PNs are all halted until they are explicitly released by the root process.

5.6.2 Initialization of the processor pool

Before a PN is released for free running, a program is load into its local memory. The reset vector, including the program counter and stack pointer, are initialized by the host machine (or alternately the MPC). Then, /HALTPN is deactivated so the microprocessor of a PN can fetch its reset vector correctly.

Before the real application is executed, an optional diagnostic program can be run. For example, the host may load a very short program (eg. add two numbers to give an answer byte) to a PN, then starts the program to see if the PN is in normal condition. Failed processors are not used for future processing.

After all the initialization, the PNs are now ready to execute the application program. The first task is to identify themselves to the MPC by calling a registration primitive. A message will be sent to the MPC (or host machine) for error detection and availability test.

5.7 Summary

In this chapter, the implementation details of SM3 is discussed. The hardware and software aspects are treated separately. The communication protocol is the most substantial significant design work so a lot of pages are devoted to this part. The 1-to-N DMA mechanism is the focus. Afterwards, the start up procedure of SM3 is briefly mentioned.

Note that the current implementation of the 1-to-N DMA bases on an asynchronous bus (VMEbus). It can be anticipated that less hand-shaking signals (5 in our case) are needed to be introduced if a synchronous system bus (and synchronous CPU bus) is used. In the mean time, the protocol shown in figure 5.13 will be simpler because some synchronization points can be removed.

CHAPTER 6
APPLICATION EXAMPLES

6.1 Introduction

An efficient message passing environment is useful for concurrent program development, parallel execution of logic programs and parallel algorithm design. SM3 can be adapted to work as the message driven OR-parallel machine described in [DelRe89], or to execute the algorithms for solving equation systems presented in [YanWe89].

To deal with distributed problems, the broadcasting feature in SM3 is suitable for informing the slave PNs of the problem configuration to be exploited. The host can use a normal message to instruct a slave PN which alternative it should exploit. Express messages can be used for conveying urgent information.

Three simple applications are given here to show how SM3 can be adapted to a particular problem.

6.2 Matrix Multiplication

Matrix multiplication is one of the problem classes that can benefit from the use of parallel computers. Suppose there are 16 square matrices and 4 PNs (including the host) are available. Figure 6.1 is a feasible plan to find the chain product. The steps are:

1. Host computer initiates the system, down-loads application programs to the slave processor nodes.
2. Each slave processor accepts 2 matrices from the host. These 8 matrices are multiplied by the slaves and 4 partial products are passed back to the host.

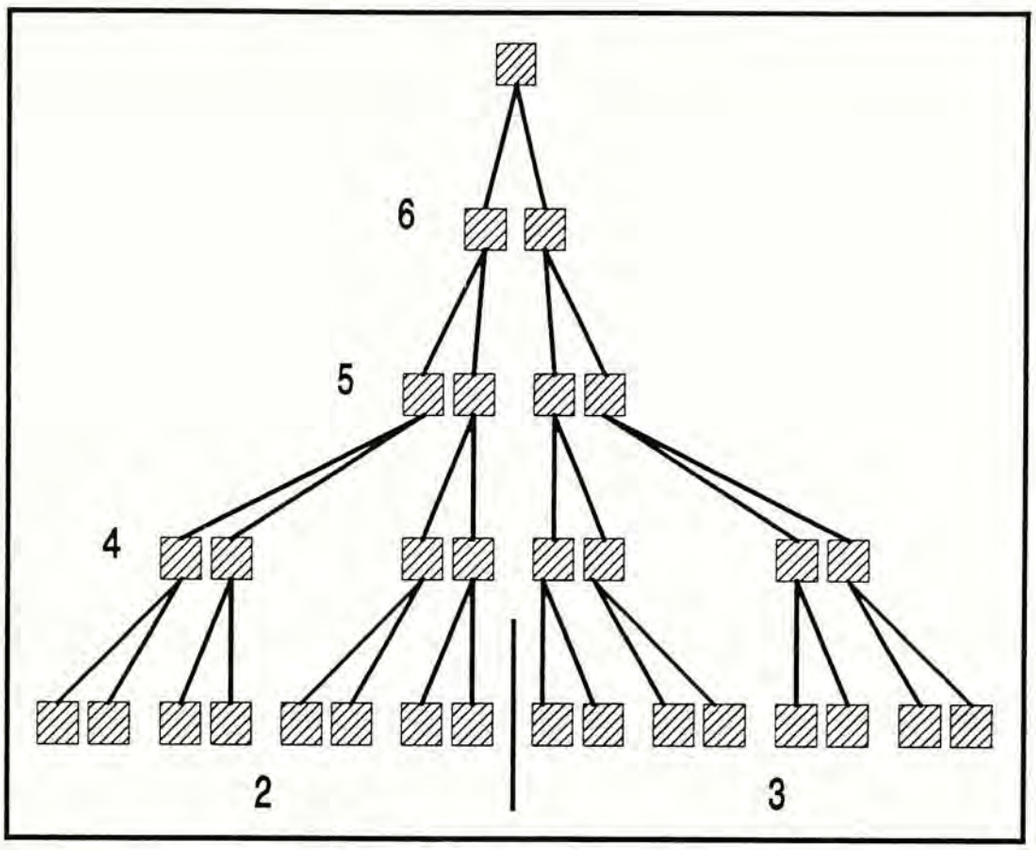


Figure 6.1. Multiplication plan of square matrices.*

3. Repeat step 2 with the remaining 8 matrices.
4. The host distributes the 8 partial products to the 4 slave processors for multiplication.
5. The 4 partial products are collected by the host and redistributed to 2 slaves for multiplication.
6. The host collects the 2 partial products and performs the final multiplication.

The matrices mentioned above are transported by normal messages. If any of the processor detects a zero matrices, the slave PN can tell the host this fact by sending him an express message. The host may shut down the system and return the answer zero matrix immediately.

From figure 6.1, we found that there are 15 multiplications. With a parallel computer like SM3 with 3 slave PNs and the host machine (degree of parallelism is 4) to do data processing, the result can be obtained in roughly 5 multiplication time units plus the communication overhead. Maximum practical speed

up is 15/5 = 3, which is smaller than the ideal factor 4.

6.3 Parallel QuickSort

Another example is a multiprocessor implementation of the well known QuickSort algorithm. Parallel QuickSort is a straight forward enhancement of the classical sequential QuickSort [Quinn87]. Although this is not an efficient parallel sorting algorithm, it qualifies as a clear demonstration example due to its simplicity and popularity. The basic idea is illustrated below:

Every slave PN executes an instance of the parallel algorithm. The elements to be sorted are stored in an array in the global memory (probably at the host machine). A stack, maintained by the host machine stores the indices of sub-arrays that are still unsorted. The picture is shown in figure 6.2.

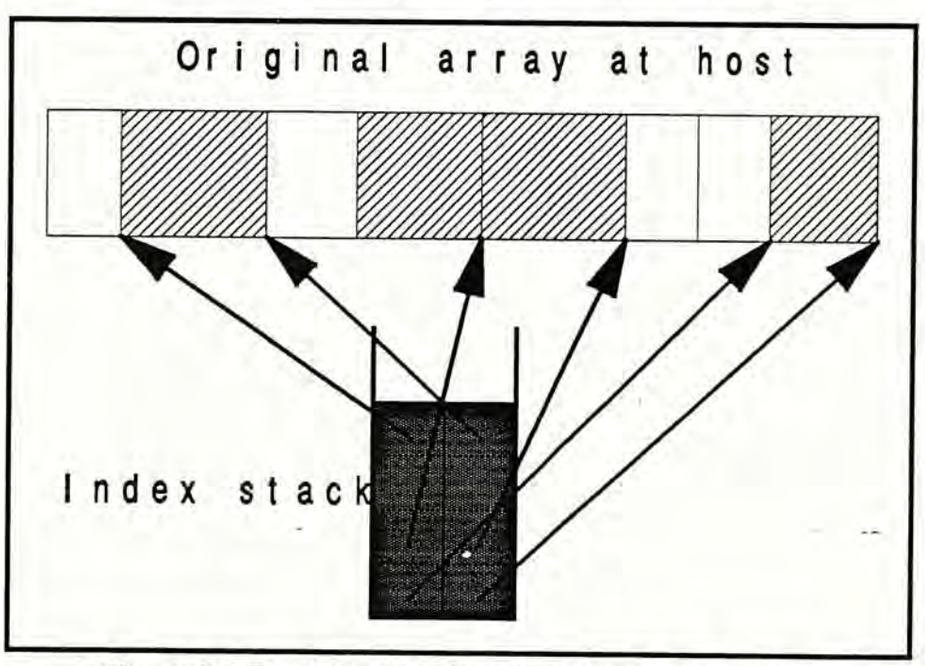


Figure 6.2 The index stack and unsorted sub-arrays.

When a PN is free, it attempts to pop the indices for an unsorted sub-array off the global stack by sending a request message to the host machine. If it is successful, a sub-array is delivered to the PN in the form of a message. The PN then partitions the sub-array, based on a supposed median element, into two smaller arrays, containing elements less than or greater than the supposed

median value. After the partitioning step, which is identical to the partitioning step performed by the serial QuickSort algorithm, the process pushes the indices of one of the sub-arrays onto the global stack of unsorted sub-arrays and repeats the partitioning process on the other sub-array.

In order to improve the performance of this algorithm, sub-arrays smaller than a certain number of elements may be sorted using other algorithms such as merge sort. Gehringer [GeJoS82] discusses how to reduce the stack access frequency, this helps to minimize the communication overhead of the algorithm.

6.4 Pipeline Problems

Some scientific calculations involve the transformation procedure. A series of matrices is to be multiplied to a set of vectors or matrices. Each transformation matrix performs an operation on the incoming data. This type of computation is very common in graphics systems.

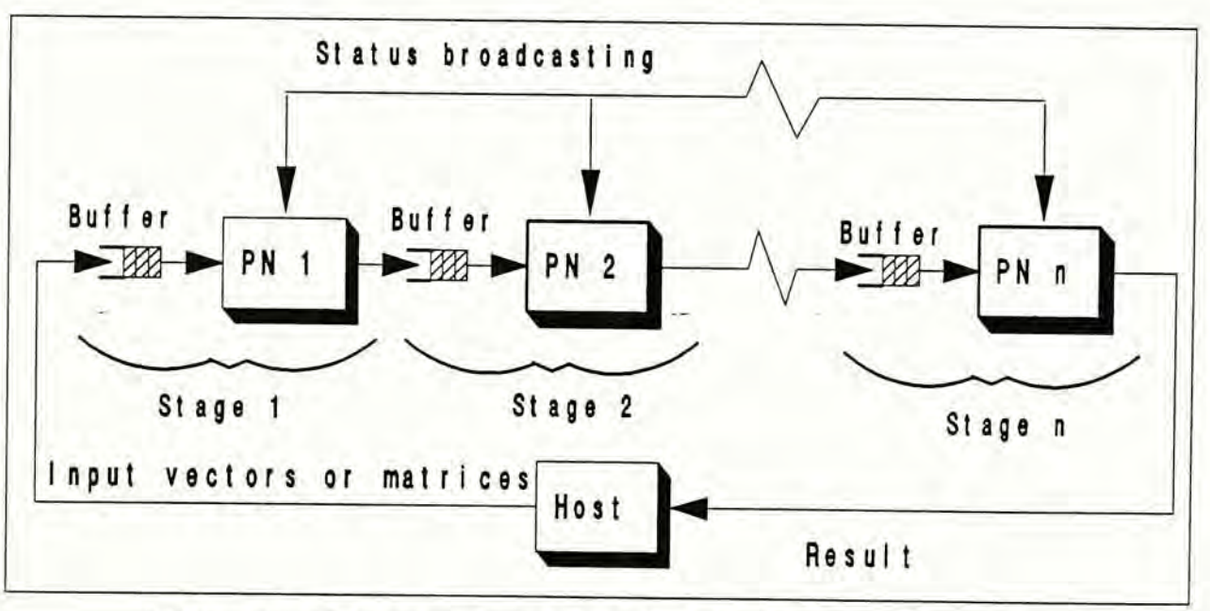


Figure 6.3 SM3 as a processor pipeline for data transformation.

The PNs in SM3 may be configured into a functional pipeline for this application, as shown in figure 6.3. The Buffer queues in the figure are the IMQ on the MPC. Each PN performs a particular operation on the incoming data and passes the result to the next PN logically following this PN. The PNs execute their programs asynchronously so the input and output requests of the PNs generally arrive at different instances. Since the pipeline is asynchronous, bus contention should not be too serious if communication overhead is not excessive. Since the PNs are coupled by non-blocking message-passing , the pipeline is very flexible and efficient (physical pipelines are synchronous). Status information are conveyed by broadcast messages while the vehicle for control commands may be express messages.

CHAPTER 7
UNSOLVED PROBLEMS AND FUTURE DEVELOPMENT

7.1 Current Status

Due to the extensive hardware construction exercise in the project, time has been tight. Since wire-wrapping was chosen as the implementation method, the amount of time needed was probably greater than using the printed-circuit board method. Subject to time and cost considerations, 3 slave PN and one MPC modules would be constructed.

The circuit design and floor plan design stages have been completed. Since the MPC is simpler than a PN, we tried to construct a PN first. Another reason is that the workstation can be operated under the direct control of the host without the MPC.

One complete PN has been constructed but not yet fully debugged. That includes all the PAL design work and the wire-warp exercise. The PN has been attached to the VMEbus with the host machine. Local memory of the PN could be accessed (read and write) from the host machine correctly. That implies the following parts are operational: Clock system, VMEbus Interface as a slave, Address decoder, /DTACK generation, /BERR generation, Memory System, Local Bus Controller and Arbiter, Reset circuity, and Register File. Furthermore, an off-line test showed that the Communication Logic on PAL chips were functionally correct.

The remaining 2 PNs and 1 MPC modules have been equipped with the following parts because they were verified to be correct: Clock system, Local Bus Gate, VMEbus Gate, Watch-dog Timer, and Memory System. Moreover, the memory of on all modules had been confirmed to be correctly accessible (read and write). As the sockets for all chips had been fixed onto the circuits boards,

it would be easy to duplicate more parts onto these incomplete modules once the parts were fully debugged.

Although the debugging phase has not yet been completed, some possible immediate improvements can be identified. The future development direction can also be figured out from the current trend, state-of-the-art and experience.

7.2 Possible immediate enhancements

Since the current version of SM3 is a prototype system, many parameters are arbitrary and very conservative. Let us examine the major components one by one and see what improvements are possible.

More peripherals should be added to the host machine whenever possible. A backup tape and a printer are in great demand for the current configuration. A floppy disk driver will make data exchange easier. All these equipment are already available and can be installed once the workstation becomes operational.

7.2.1 Enhancement to the PNs

Microprocessor MC68030

Although at the time this thesis was written MC68040 had been announced, it was not available in large quantities and at a low cost. Before it becomes a popular CPU, documentation and support chips are difficult to access. Moreover, should the MC68040 becomes available, because the interface requirement of MC68040 is different from its predecessor, the PNs must be modified. A simpler upgrade is to replace the current 16 MHz MC68030s with 25 MHz ones. However, faster SRAMs and clock rate are required.

DMAC MC68442

The current DMAC is a 10 MHz one, a faster one is highly desirable. Moreover, a 32-bit one may replace the operating 16-bit one. Since the current DMAC has

a slightly different bus definition comparing with that of the CPU (eg. some data and address lines are multiplexed), signal conversion is needed. This inefficiency should be eliminated as soon as possible.

Static memory

Expanding the memory size to nearly 1 Mbytes per PN is straight forward. For cost consideration, dynamic RAM will be more desirable for such a size. However, a memory size larger than 1 Mbytes will call for a redesign of the decoder logic. The use of dynamic RAM will complicate the 1-to-N DMA logic due to refresh requirement. If refreshing on all PNs are synchronous (currently the clocks are independent so modification is necessary) then the work will be easier.

Virtual memory

This is a natural enhancement since the memory size of even 1 Mbytes for each PN may be inadequate for some applications. However, I/O requirement will be intensive so it must be carefully designed. The load balancing and task distribution strategy must be reviewed.

7.2.2 Enhancement of the MPC

The architecture of the MPC resembles that of the PNs so similar enhancements can be applied. The shift of the software communication protocol logic to hardware is a possible move. Finally, a fast PROM may house the MPC software so more space is available for buffer queues. Diagnostic and performance monitoring hardware are essential for the production model.

From the software point of view, the MPC can support many functions in addition to its basic duties. Message filtering, security check, and load balancing are good features to incorporate.

7.2.3 Communication kernel enhancement

More communication primitives are required in order to provide a convenient and powerful environment. The following primitives may be added:

a. primitives for high level acknowledgement - the idea is shown in section 4.2.2.3. It is desirable to provide a construct for this purpose.
b. dynamic IMQs manipulation - in order to cope with the change of communication demand during the course of execution, a process may like to vary its maximum IMQ size dynamically. Moreover, it is desirable to allow a process to delete or create one or more of its 3 IMQs.

7.3 Limitation of a shared bus

The well known problem of a shared bus is contention. Depending on the application and the particular bus characteristics, the maximum number of nodes that can be attached to a shared bus varies from 4 to around 16 (eg. SUPRENUM system). The direct expansion by adding extra PNs onto the shared bus is not practical. A different way of expansion is necessary.

Research works shown that a **ring bus** structure provides quite good cost/performance ratio [Halst87]. For SM3 we can employ about 6-8 PNs in a cluster with one MPC. Clusters are then inter-connected with a ring bus as shown in figure 7.1.

In [RetTh86] several ways to relieve contention problems on shared-memory multiprocessor systems are introduced. Processor-memory interconnect contention, especially for the case of common bus, was studied and a number of interconnection networks are suggested. Besides, contention for a path through the interconnect, for a memory module, and for memory locations are also discussed. We can adapt some suggested solutions to SM3 if necessary.

7.4 Number crunching capability

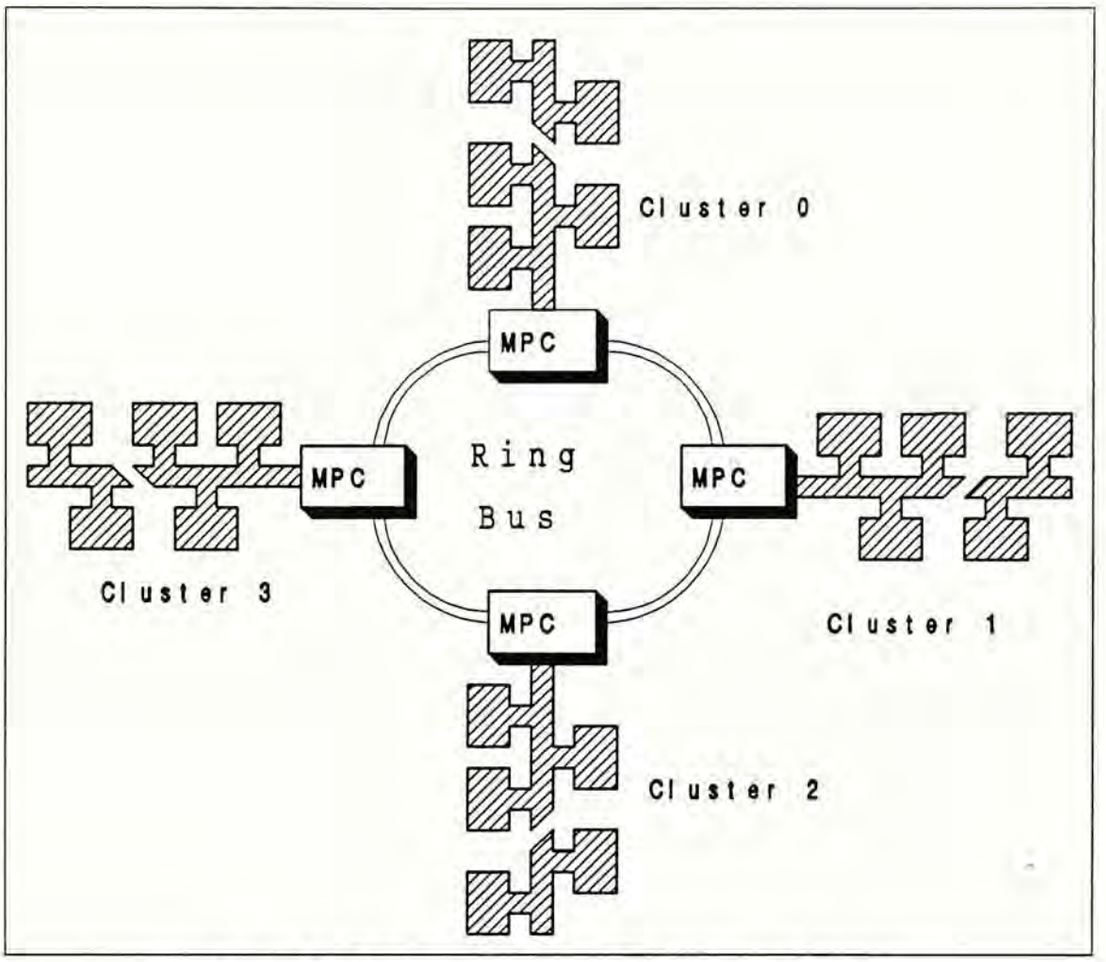


Figure 7.1 A clustered approach to expand SM3.

If SM3 is adapted to a numeric computation problem, the floating-point/ multiplication power of the MC68030 will be insufficient. A floating point coprocessor MC68881 (or the enhanced model M68882) [Motor88a] should be added. Since this chip can be directly interfaced to the MC68030 bus, the amendment to the PN circuit will be minor. It is not included in our current prototype due to cost problem. The address decoder and local bus arbiter must be modified to incorporate this new component.

If the computation workload consists of many vectors operations, a vector coprocessor is also desirable. In the SUPRENUM project, each processor node has a vector coprocessor. Although this enhancement is rather expensive, we should avoid adding the vector coprocessor to only some of the PNs since the uniformity of the system will be scarified.

7.5 Parallel programming environment

Providing a user-friendly parallel programming environment is a burning issues for all parallel computers. SM3 is not an exception. In general, there are 2 directions for us to choose:

7.5.1 Conform to serial language

It is also the current choice of SM3. The advantage of this approach is that it does not required a multiprocessor operating system. The programmer uses conventional sequential programming language, such as Fortran and C, to write programs. When a sequential language with no extension is used, as for the case of SM3, the parallelization of application programs is done manually. As SM3 explores coarse gain parallelism at the procedure level, the job of the programmer is not so tedious. But it is somewhat inconvenient and error prone. Let us see an example. In the SUPRENUM project mentioned in chapter 2, nearly 1/3 of the effort was on software issues. While one major task is to parallelize conventional numeric algorithms by hand.

Life of the programmer will be much easier if parallelization can be automated. That will be the parallelizing compiler's duty to exploit hidden parallelism in a programs written in a sequential high level programming language. However, procedure level parallelism is more difficult to detect and control comparing with fine gain (instruction level) parallelism because the behavior of a procedure (the scheduling unit) is extremely difficult, if not impossible, to predict while the behavior of an instruction is well known.

For the case of SM3, a parallelizing compiler can be constructed in an over-simplified way. Using the parallel quicksort described in chapter 6 as an example, we can detect the major recursion (or iteration) in the serial version of quicksort automatically. If this part is written in the form of a function or procedure, the compiler can generate the root process code, initialization code, task distribution code, and message communication code in a mechanical way. Of course, the effectiveness a task assignment plan strongly depends on the data

dependency, and hence the communication pattern, between interacting processes. Since this problem is still an active research area, an immediate remedy is to allow the programmer to assist in the task distribution work interactively.

7.5.2 Moving to parallel programming languages

Using a parallel language can greatly increase the reliability of an application program and significantly simplifies the work of the programmer. Parallel languages can range from very primitive and propriety ones, such as PL/M from Intel, to quite abstract and popular ones, such as CSP (Communicating Sequential Process) languages. These parallel languages requires a **parallelizing compiler** and an execution environment.

Now the duties of the compiler is to handle those parallel language constructs. For scientific and prototyping purpose. A multiprocessor operating system (OS) is not mandatory because the user may directly control the system. On the other hand, a standard multiprocessor OS must be available if we supports general purpose computing. Compatibility with existing commonly used uni-processor OSs is a critical issue for the popularity of multiprocessor systems. Since the advance of software technology is lagging behind that of hardware technology, this point is even more important then the hardware enhancements described in earlier sections in this chapter.

Moreover, we can release the requirement that each PN in SM3 executes a single process. Task distribution and load balancing will be automated. Then, the system will be more user friendly, and will be able to serve a wider range of users.

We shall discuss this point in greater details. The evolution path from a uni-processor OS to a multiprocessor OS is traced.

7.5.2.1 Uni-processor Unix

Unix is widely claimed as the most standardized, most portable, and full-function operating system. Although Unix started out as a "departmental-level" OS, it has expanded both upwards and downwards. In fact, Unix is the first full-function OS that runs on many types of computers belonging to a wide range of manufactures [Jeffr84]. It has become an industrial standard for small workstation-class micro-computers intended for multi-tasking or multi-user applications. The vast amount of software available such as word-processors, type-setters and engineering packages adds extra value to this small, elegant operating system.

Unix consists of a kernel of about 15,000 lines of source codes, and 300,000-odd lines of utilities programs mostly written in C [Jeffr84]. The kernel is the heart of an operating system, as shown in figure 7.2 [JaAnV86]. It includes a number of mechanisms from which a set of OS primitives and policies can be flexibly, efficiently, and reliably constructed.

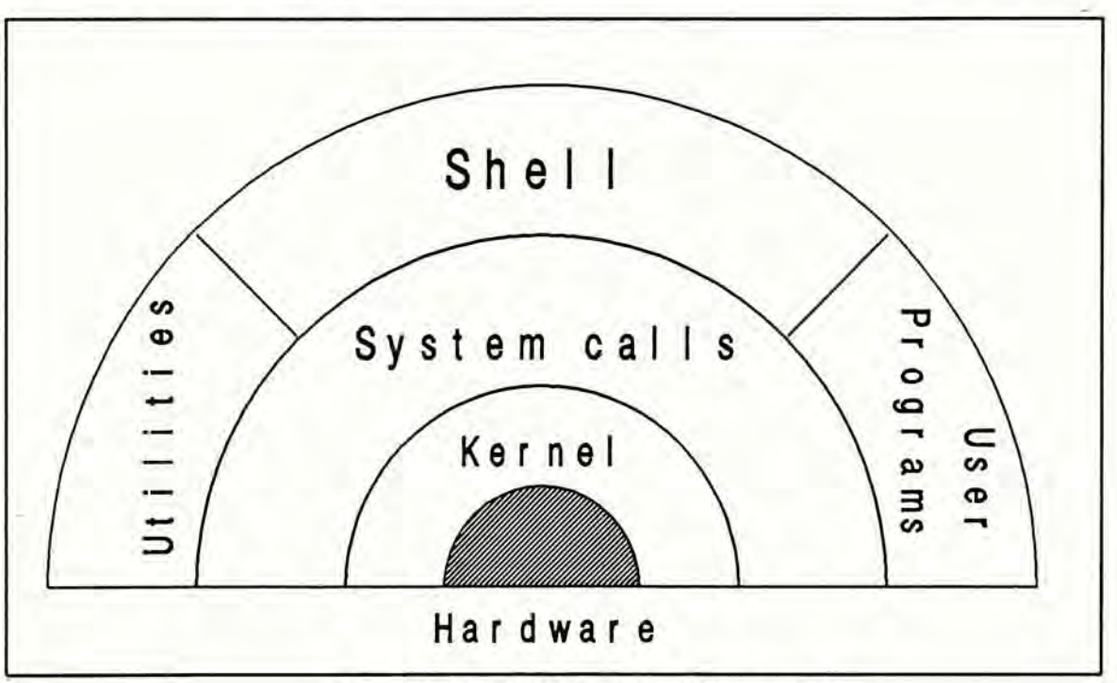


Figure 7.2 The Unix operating system.

7.5.2.2 Porting Unix

Porting Unix to a new machine is not an easy job despite Unix is the most portable OS. Figure 8.3 summarizes the major issues. About 20-30% of the kernel has to be changed which includes the device drivers, the memory

management unit (MMU), the process management unit, and the resources allocation codes. The utilities have to be examined for accidentally introduced machine-dependent codes. Compatibility between different releases of Unix must also be catered for [Jeffr84].

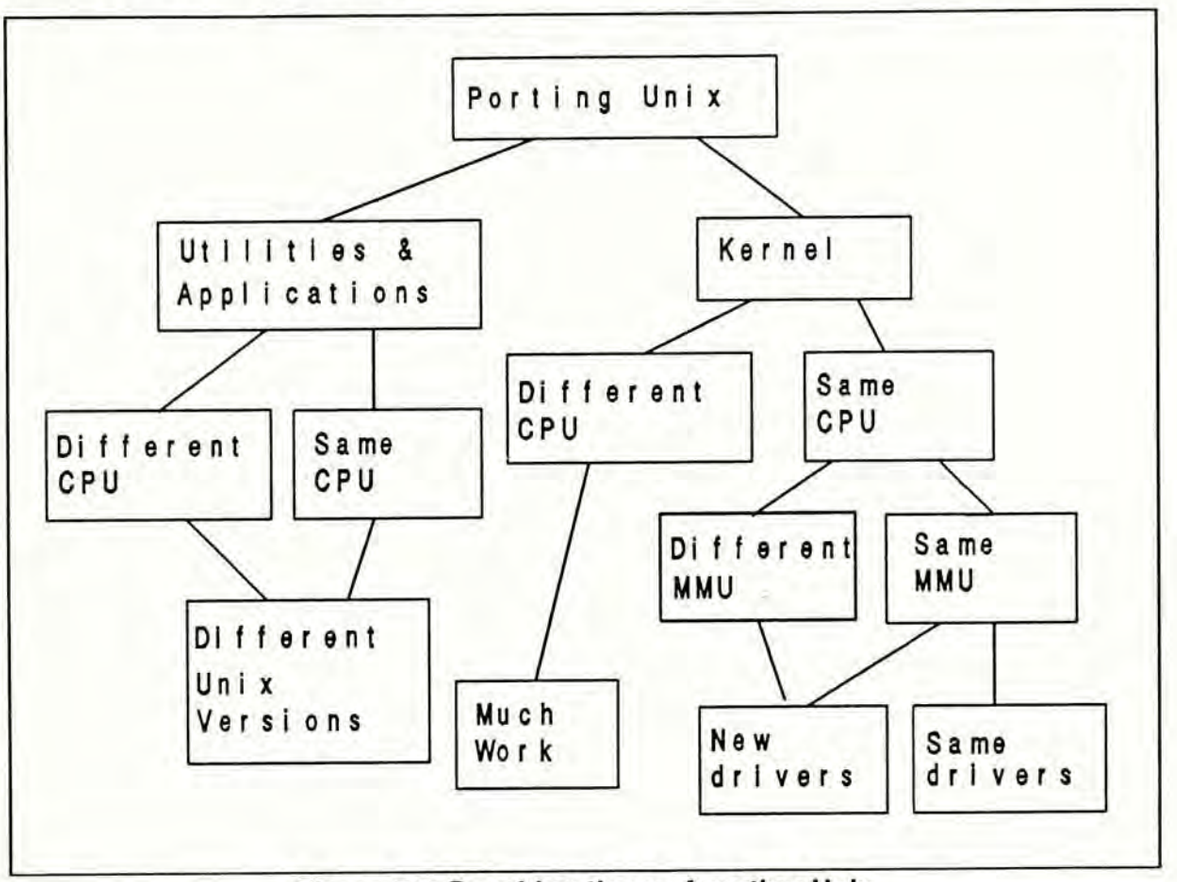


Figure 7.3 Considerations of porting Unix

Although Unix looks good for general purpose and engineering computing, it is unable to cope with the need for multiprocessing as the workload grows. Originally, Unix was a uni-processor OS.

7.5.2.3 Multiprocessor Unix

To implement a multiprocessor OS, there are two approaches [JaAnV86]:

a. Design a new operating system in a new or existing language. The special features supported by the system can be exploited. However, portability and design cost are fatal problems.

b. Adapt an existing operating system for the new hardware. The system can

be brought up quickly at a reasonable cost.

The latter approach is usually chosen under cost and compatibility considerations. The kernel of Unix, which is not requested like a system call, performs scheduling and it makes Unix a multi-tasking system. In [JaAnV86], a systematic approach to modify Unix is presented. We are going to discuss this briefly.

The protection mechanisms for critical sections in the Unix kernel must be modified when Unix is adapted for a multiprocessor environment. We have to detect and protect the critical sections of the Unix kernel. It is necessary to scan the complete Unix kernel source codes and examine every line to see whether or not it belongs to a critical section. But for reasons of optimization, Unix does not always follow its algorithm exactly. Experience showed that it is easy to detect the beginning of a critical section but very difficult to detect precisely where the critical section ends. Other problems are:

a. Unix kernel is not highly structured.
b. Owning to optimization, similar codes are by no means amenable to standard structures.
c. Because critical sections can be nested, deadlock can arise.
d. Multiple paths frequently offer a way to remain at a certain place in the kernel code.

There are three possible solutions to those problems :

a. Really go through the complete kernel. Examine every line and its environment and add locks if necessary. A completed project shows that there are many difficulties. For details please see [BacBu84].
b. Allowing only one processor at a time to execute kernel code. Parallel processing is applicable only to user processes. The Munix project [AnnJa85, MeyHa75] follows this approach. But statistics shows that the CPU of a single processor system uses half of its time to execute kernel

processes so concurrency cannot be fully exploited [JaAnV86].

c. Rewrite Unix. It presents an opportunity to improve modularity, understandability, and reliability at the same time. But a considerable amount of effort is required. For example, the internal structure of Tunis, which is a Unix-compatible system, is completely different from that of Unix. It was written in a concurrent language named Concurrent Euclid [EwHoF86, Holt82].

7.5.3 Object-oriented approach

Since message-passing is the only way of communication in a object-oriented system, SM3 can be tailored to support an object-oriented software environment. The programming model will be even simpler and more elegant than the current model. We shall briefly discuss this approach.

At the very beginning, programmers wrote straight-line codes to command the computer to carry out a series of operations that mimics operations in the real world. Then procedures were widely used for task partitioning and program structuring. But actually, the world is more process-shaped rather than procedure-shaped. The majority of available software doesn't reflect that reality. The process is a better modularization vehicle than the procedure because we can scatter processes over a collection of computers [Pete88]. This favors parallel processing. Process-oriented parallel languages such as Occam [Inmos83] are becoming more and more popular. Interacting processes need special coordination mechanisms to make sure that they are operating correctly [ShMiS78].

Based on the process-oriented approach mentioned, a higher-level method to conceptualize the world is introduced. In the **object-oriented approach**, a system is decomposed into objects. The decomposition may yield coinciding notions for both information hiding and protection, and concurrent execution. Objects in the system may communicate by exchanging messages. The internal details of an object is hidden from the outside world. Small-talk, and POOL-T [Ameri86] are

typical object-oriented languages.

To facilitate efficient execution of programs written in object-oriented languages, architectural support is desirable [WiLoE87]. Object-oriented architectures have the following characteristics [Rober81]:

a. Specification of data in programs is kept separated from how the data are referenced by program instructions or represented in memory.
b. Hardware or software controls access to different types of data with passwords (descriptors). Programmers can use modular programming and structured design more efficiently.
c. Hardware algorithms to check whether each type of object is associated with operators that make sense for it.

The object-oriented style is very promising among parallel systems. This natural method for structuring and partitioning, combined with a message-passing mechanism for communication and synchronization, greatly relieve these aspects of the programming task [WiLoE87]. Released from these responsibilities, programmers can concentrate on the problems at hand. Since SM3 supports message passing in an efficient way, it is very desirable to adopt an object-oriented environment on SM3.

7.6 Summary

We have enumerated some possible enhancements for short-term and long-term development of SM3. Both hardware and software aspects are covered. Although performance improvement is desirable, providing a good programming environment for the user should be given first attention. Without a standardized, easy to use multiprocessor OS, SM3-like systems are accessible for expert users only. Unix is a suitable choice for the application domain that SM3 aims to serve.

To provide a better programming environment, SM3 can be further developed

into a object-oriented architecture with ease because of the message-based background. A library of generic object manipulation functions, such as creation, destruction, and migration, must be developed. Data encapsulation and object distribution must be supported by the operating system. Thus, an object-oriented environment has to be integrated into the operating system. This is really a large piece of challenging work.

CHAPTER 8
CONCLUSION

8.1 Thesis summary

At the beginning of this thesis, we started our discussion by justifying the quest for parallel processing. A quick tour of the current status of parallel processing was presented, using Micheal Flynn's classification scheme [FatKr83] as a road map. Then we converged to multiprocessor systems, which is recognized as the basic building block for larger systems. We appreciated the cost effectiveness and simplicity of shared-memory bus architectures.

To solidify our discussion, several real machines are briefly introduced. That include the SUPRENUM, MEMSY, ELXSI, and SEQUENT. All of them are, more or less, bus-based shared-memory systems. Some of them employ the cluster approach for higher level expansion. Yackos is a software project aiming at providing faster message transfer on top of a shared-memory architecture. We noticed that the software MP approach was unsatisfactory.

To eliminate the context switching problem of the MP approach and enforce performance stability, a dedicated processor is reserved for the MP process. We called this software and hardware combination the Message-Passing Coordinator, or MPC for short. The basic idea is analogous to a PABX telephone switch box. We are pleased to find that the MPC can deliver much better functionalities than the MP. Expectedly, some drawbacks were found but all of them are minor problems.

After the MPC had been introduced, we presented the design of the multiprocessor workstation SM3. From a hardware point of view, SM3 is a simple distributed shared-memory bus-based multi-microprocessor system. The MPC and the host machine are the traffic and control centers of SM3. In software

aspects, SM3 offers a message-passing environment for inter-process communication. With the MPC as an agent, message traffic is regulated while delay is small. A hierarchically-typed message structure for SM3 was discussed. Most importantly, message communication protocols were tailor-made for different message types for better performance and ease of use.

We explored the intrinsic broadcasting nature of the shared bus, a 1-to-N DMA concept was devised for handling broadcast message in an efficient way. This is a by-product of employing the MPC approach. Actually, it is a generalized DMA transfer that can be implemented with many conventional DMA devices. The savings will be significant when there are many receivers. A good property of the 1-to-N DMA mechanism is that it allows the sender to select potential receivers.

After the general issues of building SM3 had been presented, we looked into the implementation details. Design decisions concerning both software and hardware were explained. We particularly included the start-up procedure of SM3 for a better understanding of the system operations.

Application examples served to illustrate how SM3 can be adapted to solve some common problems using a message-passing environment, with the hierarchical message system and the help of priority levels. In these examples, it is evident that automatic exploration of coarse gain parallelism is quite difficult. That is one of the unsolved problems. The availability of a standard multiprocessor OS in the near future is another question. Enhancement issues of the software and hardware of SM3 was discussed too. Future developments may be based on the improvements that are achievable in a short period.

In conclusion, we have designed a low cost, easy to program workstation that applies the idea of multiprocessing. The shared-memory offers a lot of advantages for communication. Such a system is suitable for prototyping, industrial engineering calculations, parallel program development, object-oriented system support and parallel execution of logic programs.

8.2 Author's comment

Undoubtedly, designing and implementing a multiprocessor system is laborious but extremely rewarding. Design trade-offs can only be made on the ground of personal experience, which is exactly what the author initially lacked but now had gained. As parallel computing is the current trend, the invaluable experience of designing and building a parallel computer outweighs the inevitable frustrations it brings.

Let us return to SM3 and the MPC approach. Although parallel computers have not walk out of the ivory tower completely, the practical experience that brought to us during the design and implementation of SM3 will certainly be helpful to the literacy of parallel computers. Definitely, the author enjoys the achievements of the very pragmatic design of SM3, the advancement of the MPC approach, and the delighting discovery of the 1-to-N DMA mechanism. The flexibility and expansibility of the SM3 workstation offer a very promising development opportunity, in particular for the design of concurrent programs for distributed problem solving, and for the provision of an architecture suitable for supporting an object-oriented environment.

The author would like to point out that the movement from the MP approach to the MPC approach exhibits the favor of **software oriented architecture.** Traditional computer architects satisfy on delivering machines that can execute any program, which may be any mix of machine instructions, correctly and smoothly. But the current trend is to put the programming model into consideration early in the architectural design phase. In these system, acceptance by the programmer is much better, and very likely the performance is also improved due to the architectural support of a predefined, extensively used computation model. In this sense, the MPC approach is certainly an advance. (fixing a process to a processor seems a stepping-back action because the process cannot migrate and share the processor with others. From the operating system point of view, we have retreated from multiprogramming to conventional single-

programming mode. This is a backwards move.)

SM3-like multiprocessor workstations will not win the appreciation of the general user if good parallelizing compilers and a standard multiprocessor OS are not available. Moreover, a friendly and elegant programming environment is indispensable. Although some initial works had been done, such as the MUPPET programming environment for message-based multiprocessors described in [MuLiS86], no widely accepted system has yet emerged. The author believes that object-oriented systems will gradually win the competition.

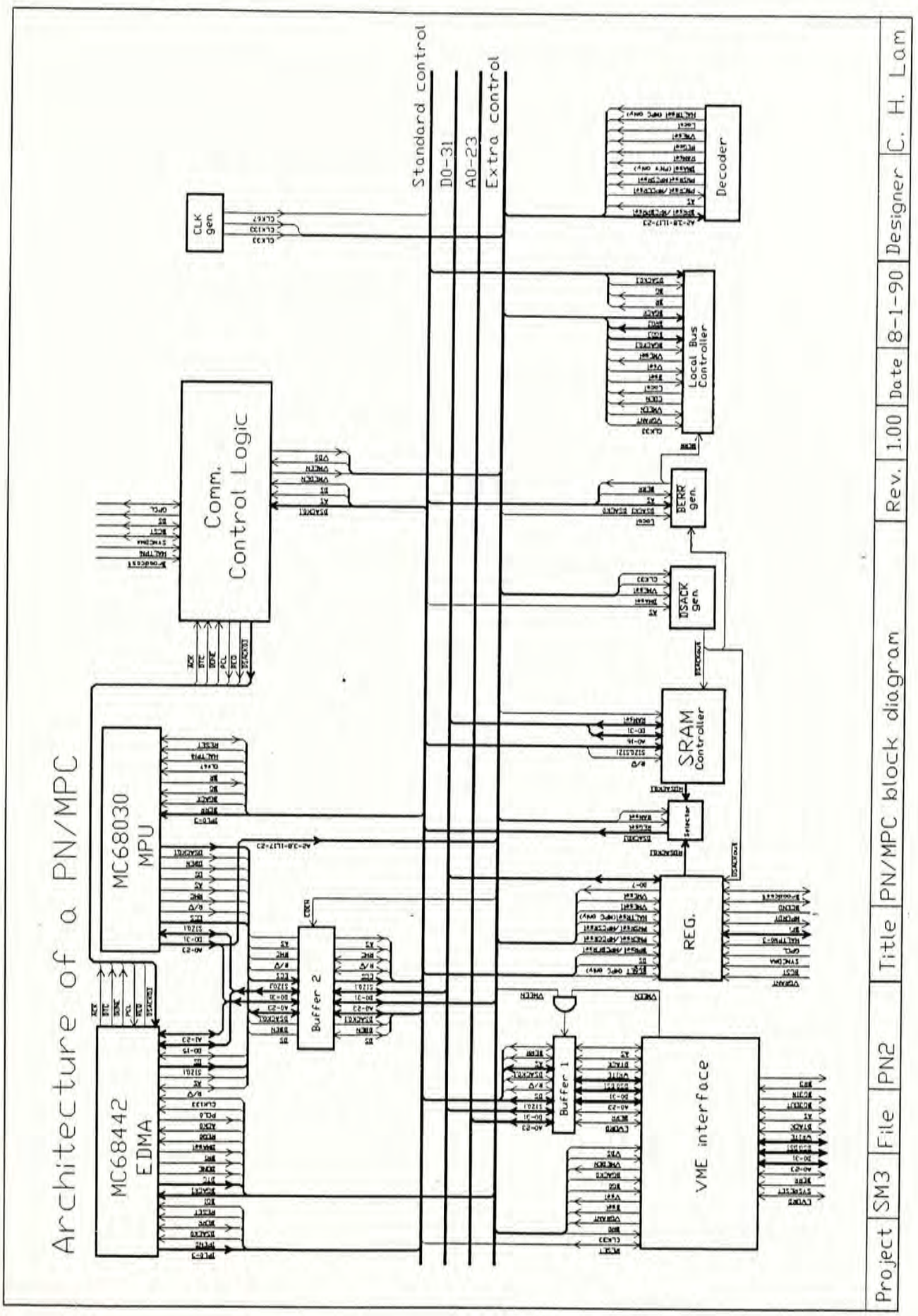
8.3 Looking into the future

Recall that SM3 explores coarse gain parallelism. It is foreseeable that future multiprocessors will explore all levels of parallelism, with a very high degree of automation. Although we emphasis software oriented architecture, it does not mean that we should build hardware in a fixed software-dependent way. On the contrary, performance fluctuation due to data variation will eventually pull out a lot of reconfigurable designs. As the size of computer systems grows monotonically, fault-tolerance will continue to be a desirable feature of multiprocessor systems that lies anywhere in the multiprocessing spectrum.

In the short term, software advances is much more desirable because there are already many parallel computers that cannot be easily programmed. Actually, SM3 is one of the vast amount of projects that attempts to surmount that barrier. In the long term, the number of processors in a system and their performance will be climbing upwards in different rates, giving more variations of machines, and also more programmability problems for our energetic fellow researchers.

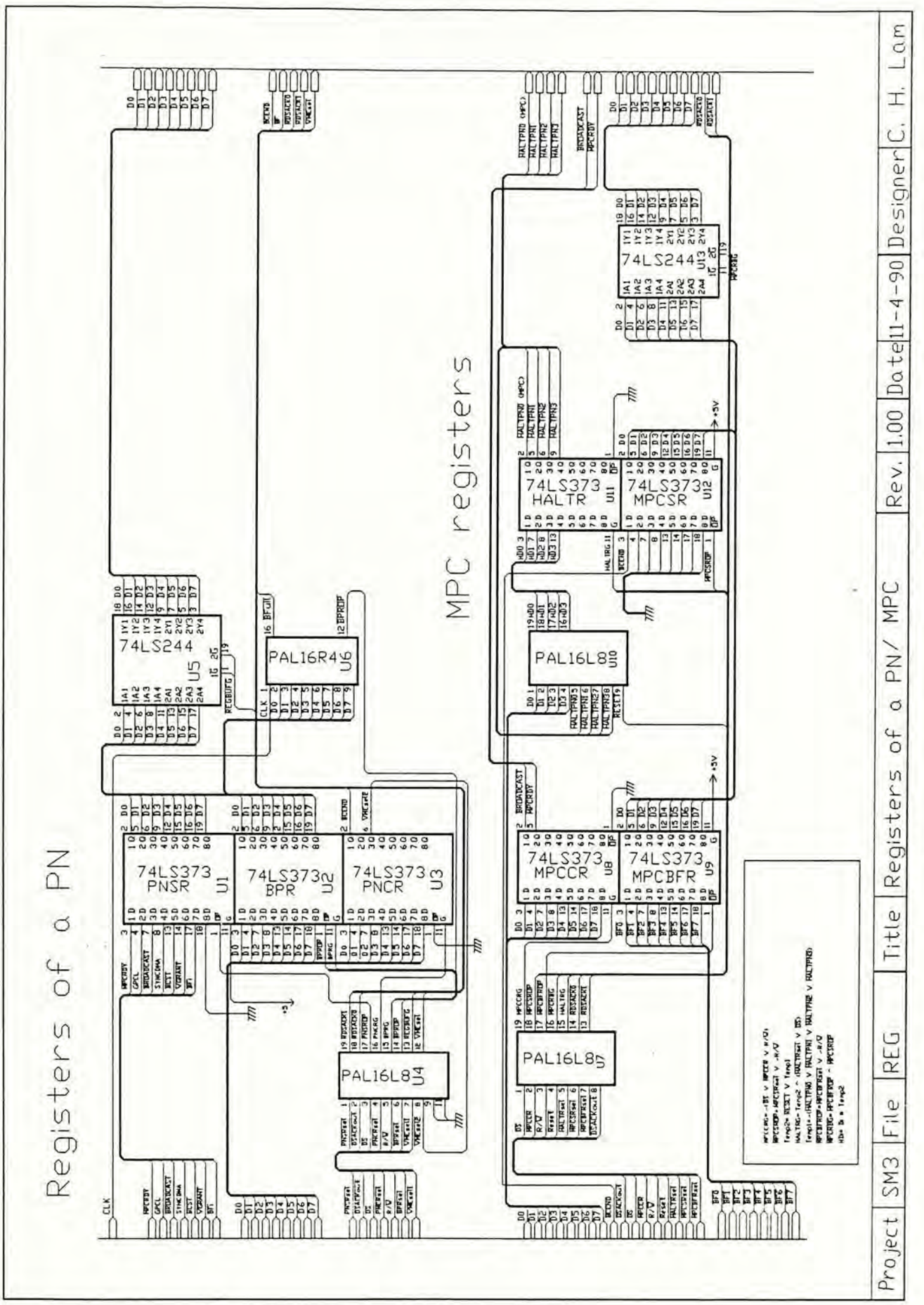
APPENDIX A
BLOCK DIAGRAM

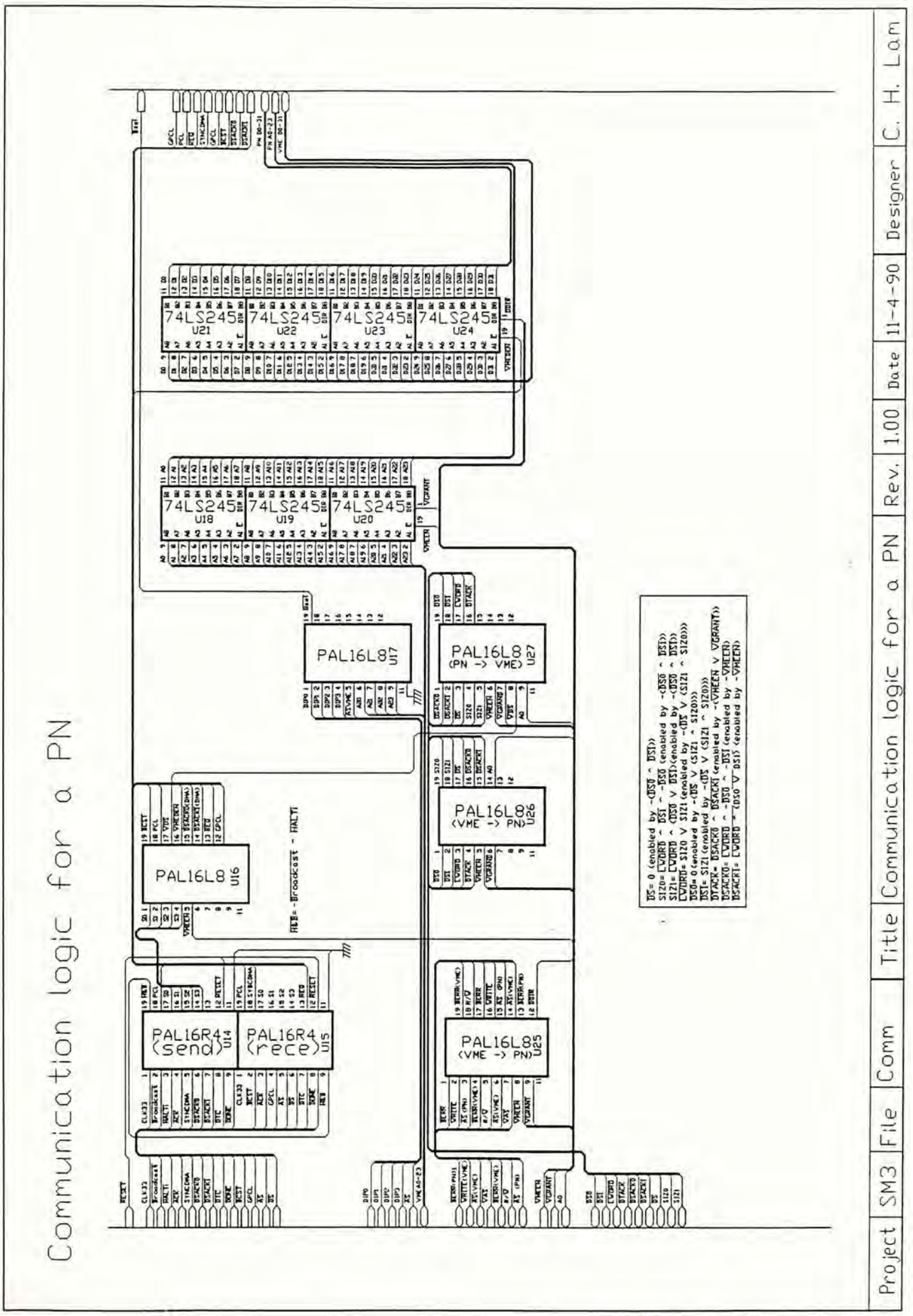
The block diagram of a PN/MPC is shown on the next page. A MPC does not have the DMAC that a PN have.

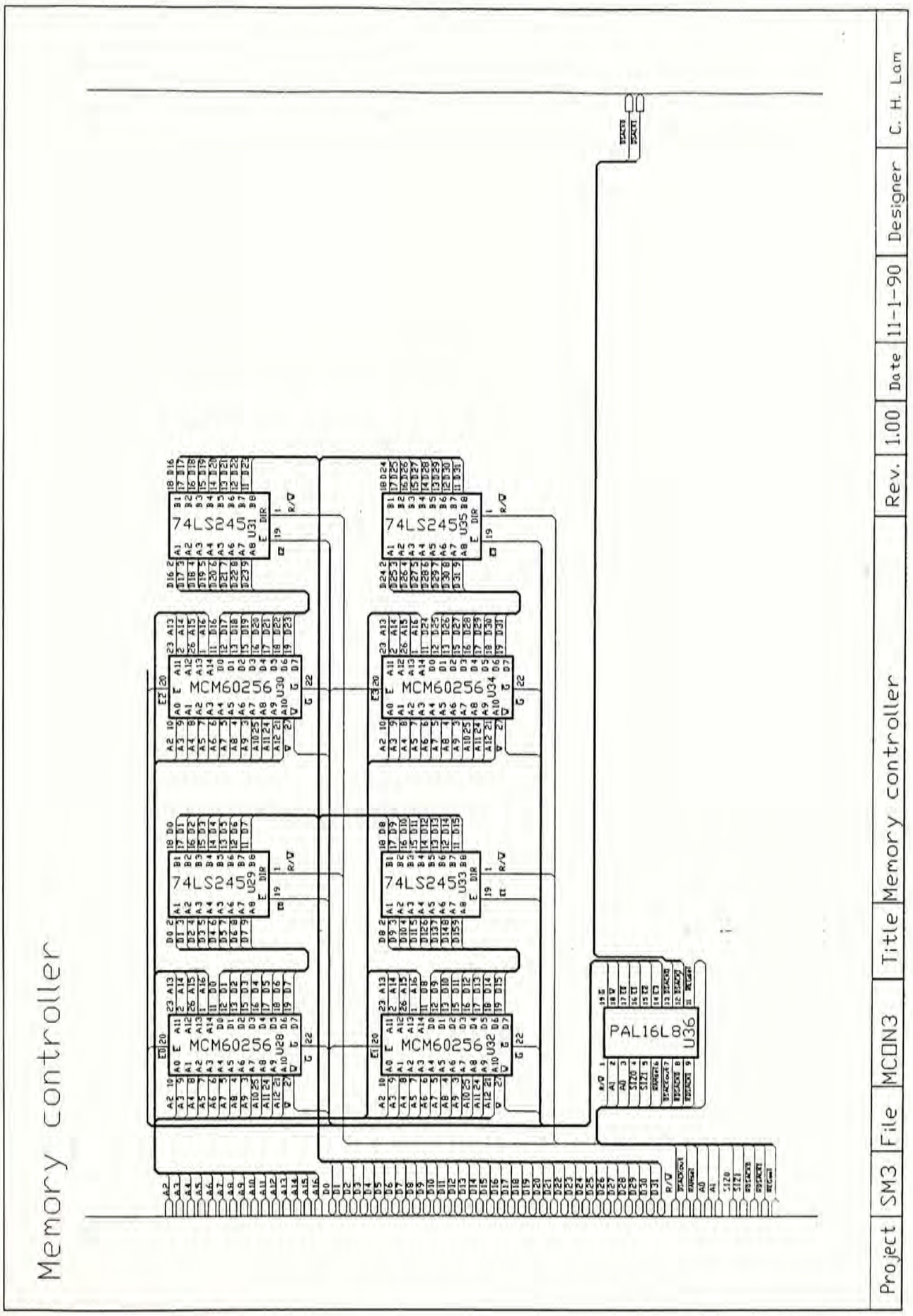

Architecture of a PN/MPC
Standard control
D0-31
A0-23
Extra control
MC68030 MPU
MC68442 EDMA
Comm. Control Logic
CLK gen.
Decoder
Local Bus Controller
BERR gen.
DSACK gen.
SRAM Controller
REG.
Buffer 1
Buffer 2
VME interface
Project SM3 File PN2 Title PN/MPC block diagram Rev. 1.00 Date 8-1-90 Designer C. H. Lam

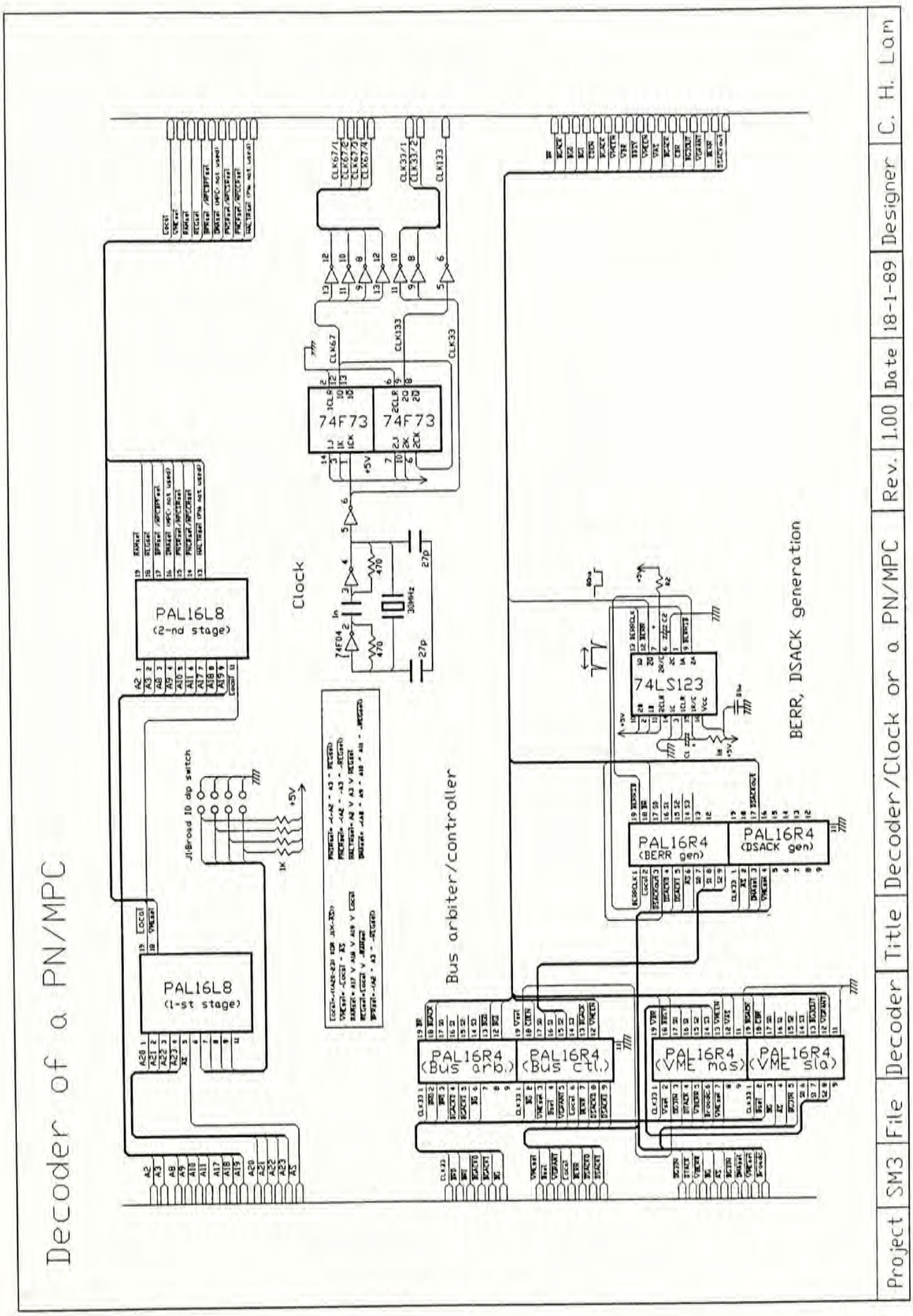
APPENDIX B
CIRCUIT DIAGRAMS

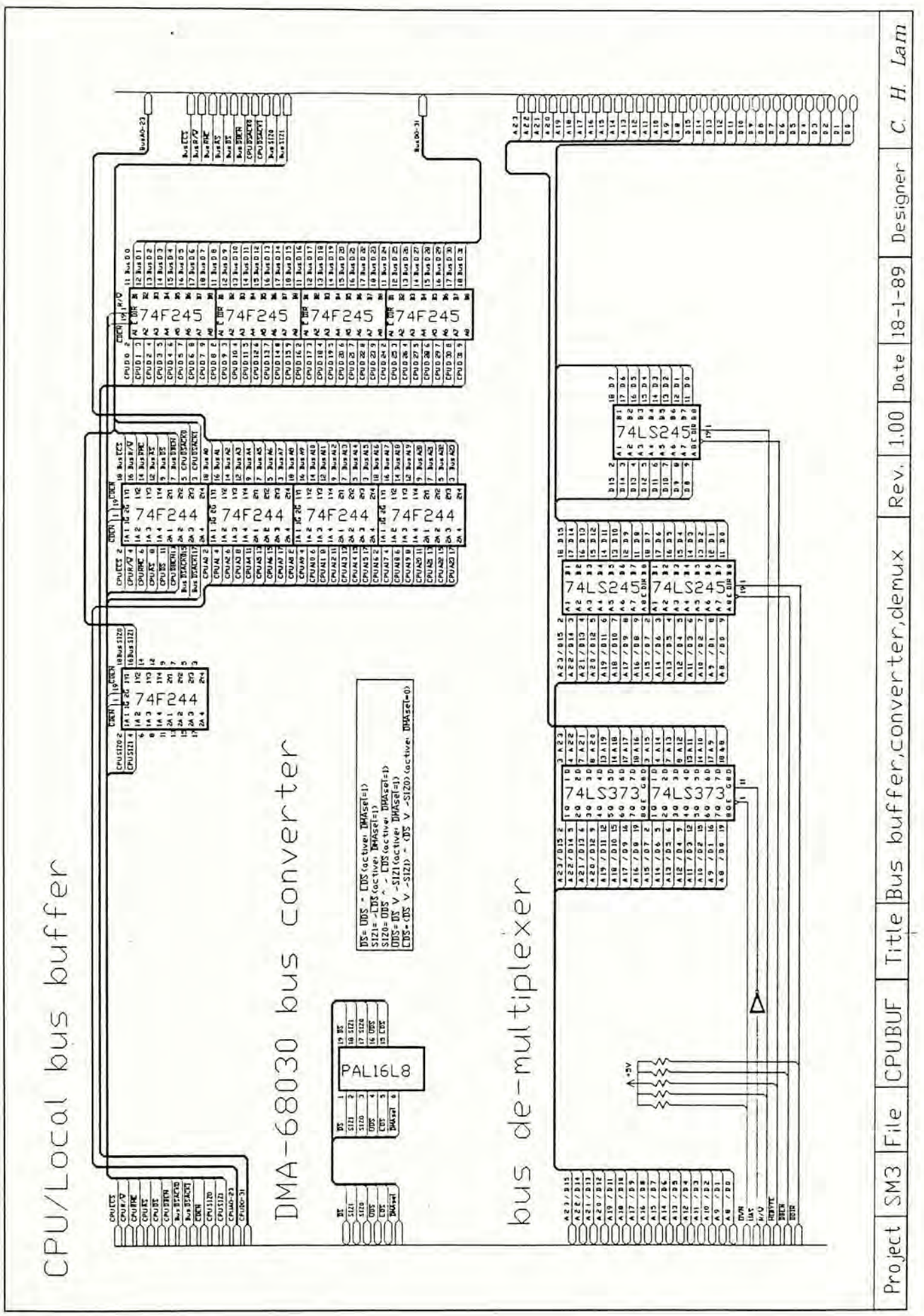
The circuit diagrams of SM3, including the PNs and the MPC, is printed on the following 6 pages. Parts for the MPC specifically are highlighted.

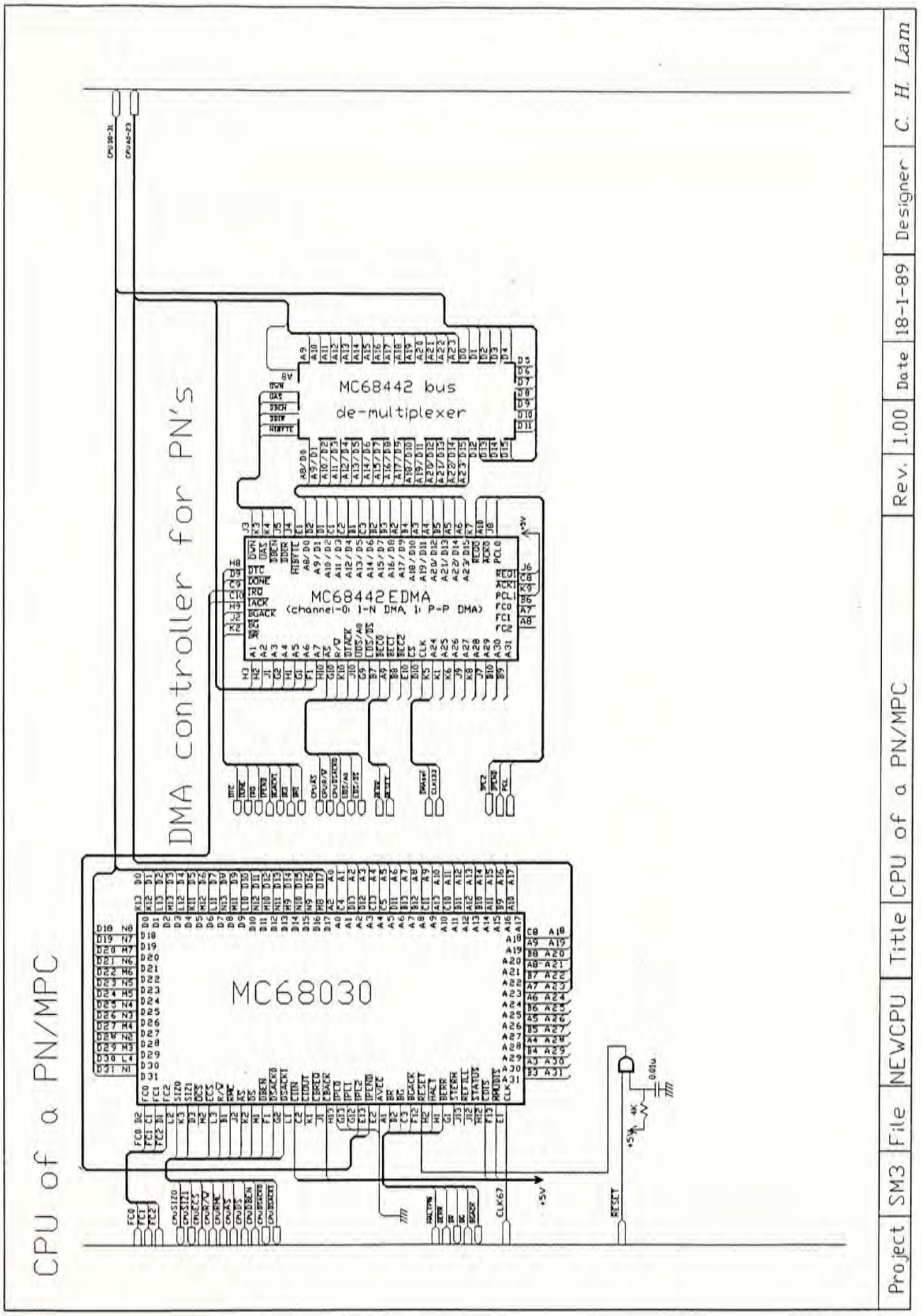

Registers of a PN
MPC registers
74LS373 PNSR U1
74LS373 BPR U2
74LS373 PNCR U3
74LS373 MPCCR U8
74LS373 MPCBFR U9
74LS373 HALTR U11
74LS373 MPCSR U12
74LS244 U5
74LS244 U13
PAL16R4 U6
PAL16L8 U4
PAL16L8 U7
PAL16L8 U10
Project SM3 File REG Title Registers of a PN/ MPC Rev. 1.00 Date 11-4-90 Designer C. H. Lam


Communication logic for a PN
74LS245 U21
74LS245 U22
74LS245 U23
74LS245 U24
74LS245 U18
74LS245 U19
74LS245 U20
PAL16L8 U17
PAL16L8 (PN -> VME) U27
PAL16L8 U16
PAL16L8 (VME -> PN) U26
PAL16R4 (send) U14
PAL16R4 (rece) U15
PAL16L8 (VME -> PN) U25
RESET = Broadcast + HALT
DS= 0 (enabled by -(DS0 ^ DS1))
SIZ0= LVDWRD ^ DS1 (enabled by -(DS0 ^ DS1))
SIZ1= LVDWRD ^ (DS0 V DS1)(enabled by -(DS0 ^ DS1))
LVDWRD= SIZ0 V SIZ1 (enabled by -(DS V (SIZ1 ^ SIZ0)))
DS0= 0 (enabled by -(DS V (SIZ1 ^ SIZ0)))
DS1= SIZ1 (enabled by -(DS V (SIZ1 ^ SIZ0)))
DTACK= DSACK0 ^ DSACK1 (enabled by -(VMEEN V VGRANT))
DSACK0= LVDWRD ^ -DS1 (enabled by -VMEEN)
DSACK1= LVDWRD ^ (DS0 V DS1) (enabled by -VMEEN)
Project SM3 File Comm Title Communication logic for a PN Rev. 1.00 Date 11-4-90 Designer C. H. Lam


Memory controller
74LS245
MCM60256
PAL16L8
Project
SM3
File
MCON3
Title
Memory controller
Rev.
1.00
Date
11-1-90
Designer
C. H. Lam

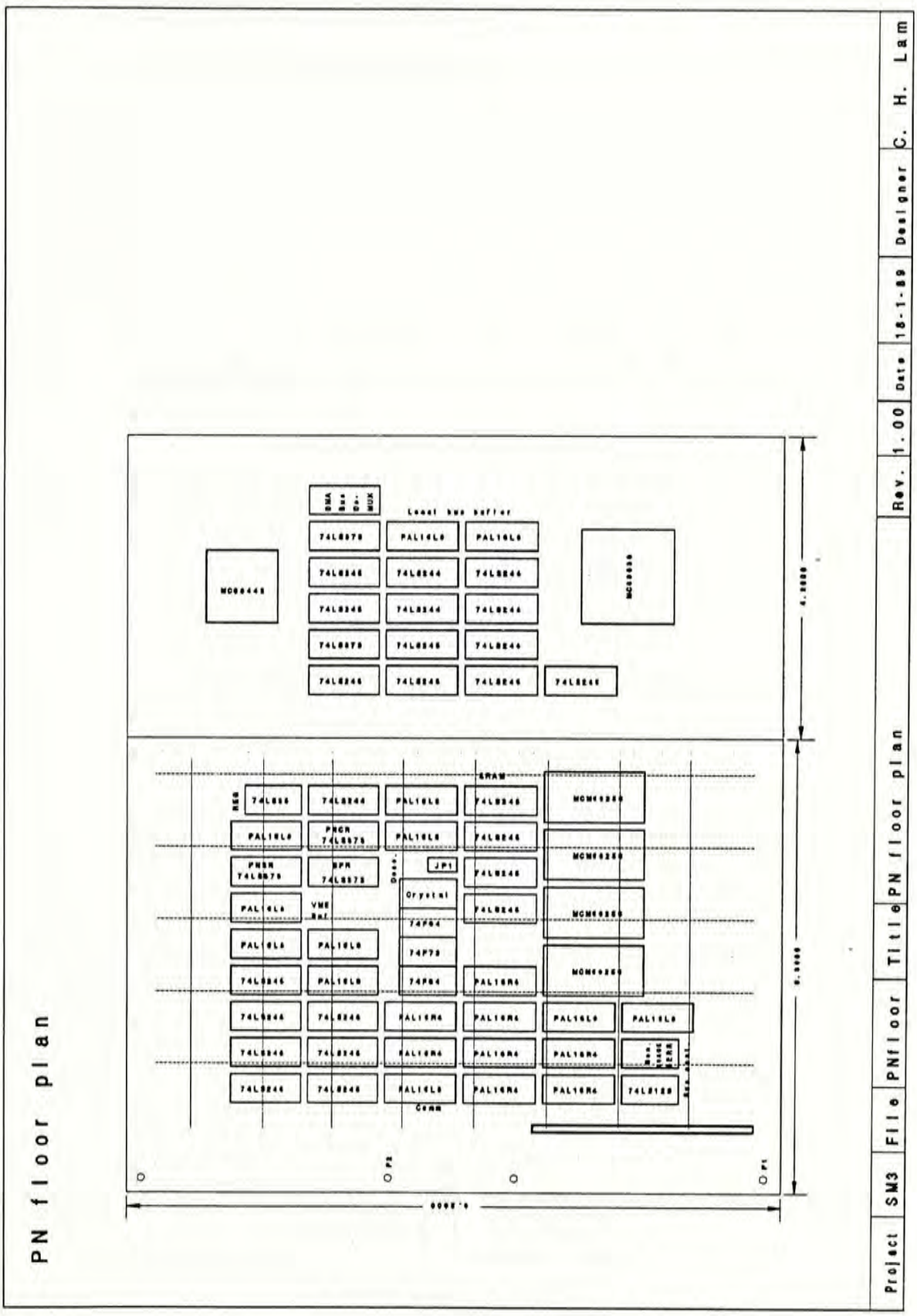

Decoder of a PN/MPC
PAL16L8
(2-nd stage)
PAL16L8
(1-st stage)
Clock
74F73
74F73
74LS123
Bus arbiter/controller
PAL16R4
(Bus arb.)
PAL16R4
(Bus ctl.)
PAL16R4
(VME mas)
PAL16R4
(VME sla)
PAL16R4
(BERR gen)
PAL16R4
(DSACK gen)
BERR, DSACK generation
J1Broad 10 dip switch
Project
SM3
File
Decoder
Title
Decoder/Clock or a PN/MPC
Rev.
1.00
Date
18-1-89
Designer
C. H. Lam

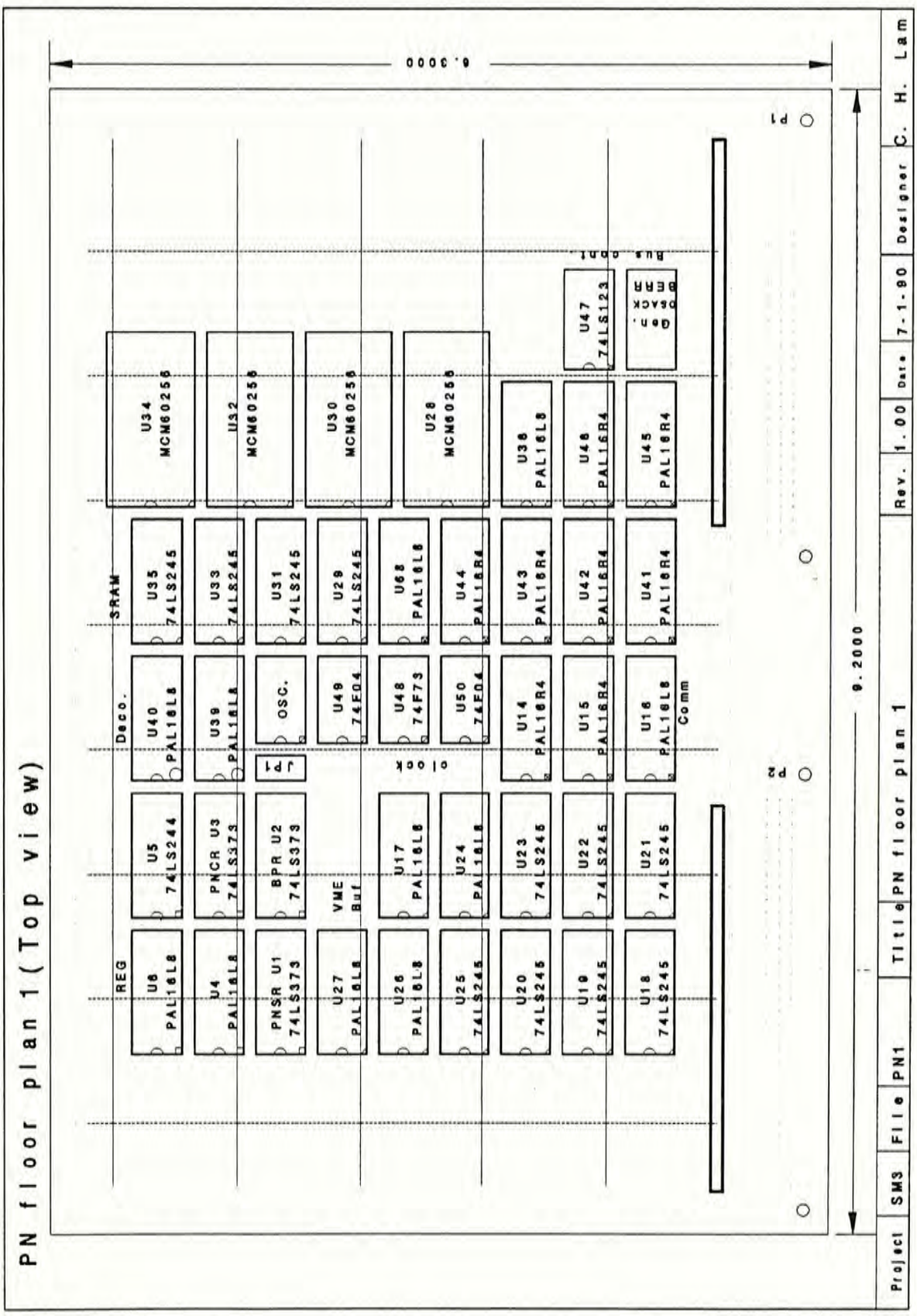

CPU/Local bus buffer
DMA-68030 bus converter
bus de-multiplexer
74F245
74F244
74LS245
74LS373
PAL16L8
DS= UDS * LDS (active: DMASel=1)
SIZ1= -LDS (active: DMASel=1)
SIZ0= UDS ^ - LDS (active: DMASel=1)
UDS= DS V -SIZ1 (active: DMASel=1)
LDS= (DS V -SIZ1) * (DS V -SIZ0) (active: DMASel=0)
+5V
Project SM3 File CPUBUF Title Bus buffer,converter,demux Rev. 1.00 Date 18-1-89 Designer C. H. Lam


CPU of a PN/MPC
DMA controller for PN's
MC68442 bus
de-multiplexer
MC68442EDMA
(channel-0: I-N DMA, 1: P-P DMA)
MC68030
Project
SM3
File
NEWCPU
Title
CPU of a PN/MPC
Rev.
1.00
Date
18-1-89
Designer
C. H. Lam

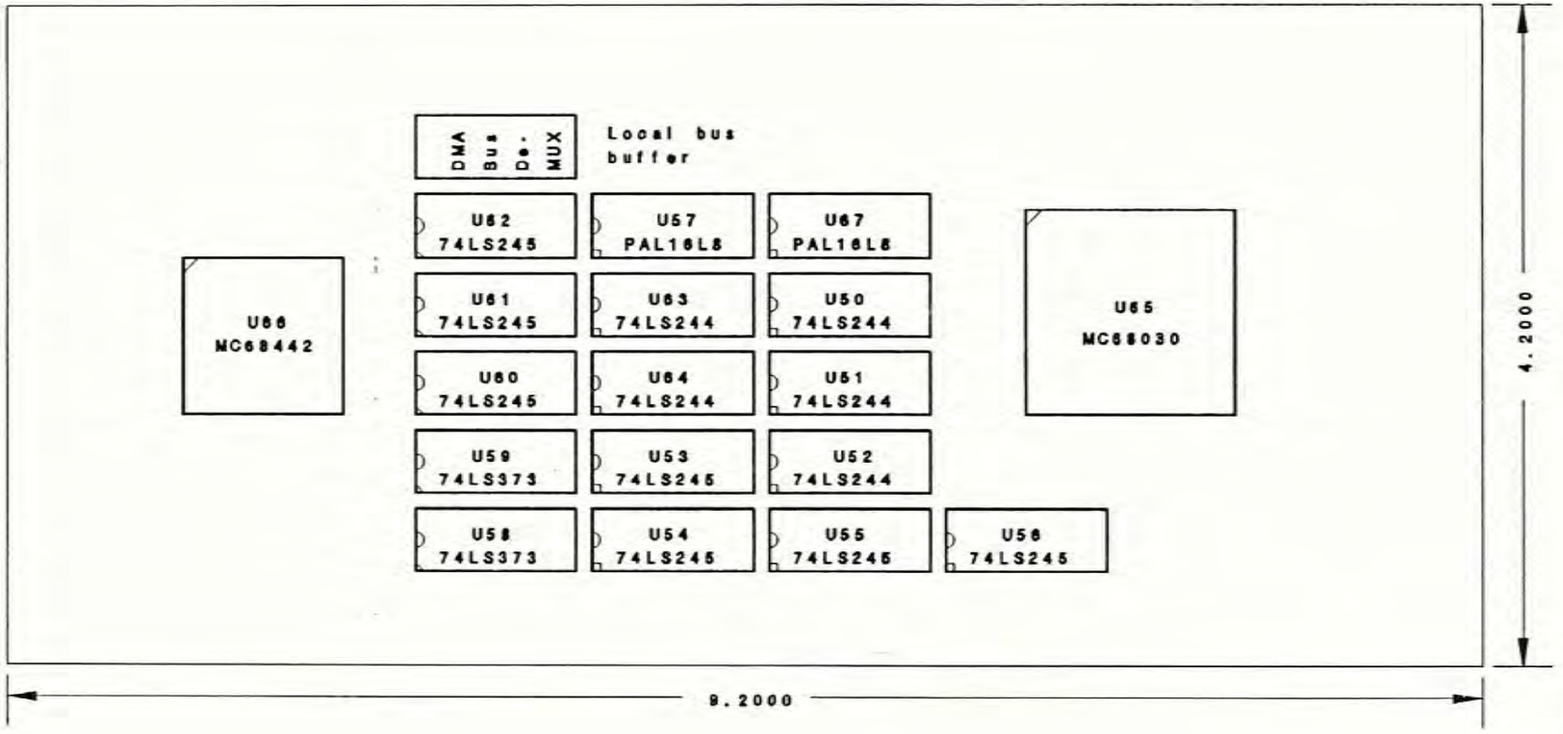
APPENDIX C
PCB LAYOUT

The PCB layout diagrams of the PNs and the MPC are printed on the following pages.


PN floor plan
Project
SM3
File
PNfloor
Title
PN floor plan
Rev.
1.00
Date
18-1-89
Designer
C. H. Lam
Local bus buffer
SRAM
JP1
Oryvtal
VME Rel
Comm

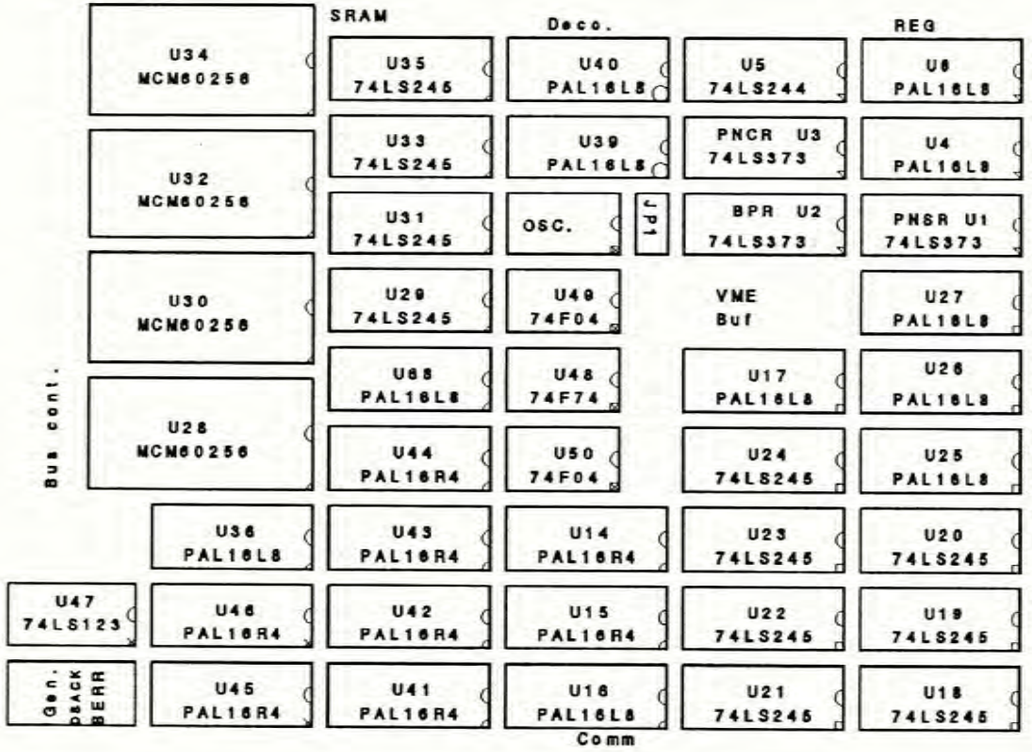

PN floor plan 1 (Top view)
6.3000
9.2000
REG
Deco.
SRAM
VME Buf
Comm
U6 PAL16L8
U4 PAL16L8
PNSR U1 74LS373
U27 PAL16L8
U26 PAL16L8
U25 74LS245
U20 74LS245
U19 74LS245
U18 74LS245
U5 74LS244
PNCR U3 74LS373
BPR U2 74LS373
U17 PAL16L8
U24 PAL16L8
U23 74LS245
U22 74LS245
U21 74LS245
U40 PAL16L8
U39 PAL16L8
OSC.
U49 74E04
U48 74F73
U50 74E04
U14 PAL16R4
U15 PAL16R4
U16 PAL16L8
U35 74LS245
U33 74LS245
U31 74LS245
U29 74LS245
U68 PAL16L8
U44 PAL16R4
U43 PAL16R4
U42 PAL16R4
U41 PAL16R4
U34 MCM60256
U32 MCM60256
U30 MCM60256
U28 MCM60256
U36 PAL16L8
U46 PAL16R4
U45 PAL16R4
U47 74LS123
Project SM3 File PN1 Title PN floor plan 1 Rev. 1.00 Date 7-1-90 Designer C. H. Lam

PN floor plan 2 (Top view)



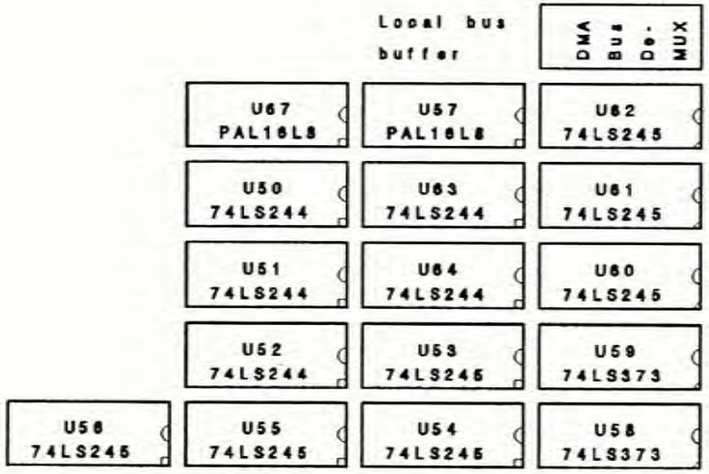
Project	SM3	File	PN2	Title	PN floor plan 2	Rev.	1.00	Date	7-1-90	Designer	C. H. Lam

PN floor plan 1 (Bottom view)



Project	SMS	File	BACK1	Title	PN floor plan 1 (Bottom)	Rev.	1.00	Date	7-1-90	Designer	C. H. Lam

PN floor plan 2 (Bottom view)



Project	SMS	File	back2	Title	PN floor plan 2 (Bottom)	Rev.	1.00	Date	7-1-90	Designer	C. H. Lam

APPENDIX D
VMEBUS ADDRESS MAP

Given that 24 out of the 32 VME bus address lines are used, the active addressing space is 16 MBytes at most. We represent the VME address by 8 hexadecimal digits. The address map is shown in figure D.1.

VME addr.	Space allocation
00000000 007FFFFF	Host machine
00800000 008FFFFF	MPC
00900000 009FFFFF	PN2
00A00000 00AFFFFF	PN3
00B00000 : 00FFFFFF	PN4 : PN8
01F00000 : FFFFFFFF	Reserved

Figure D.1 VME address map

The map implies that each processor node may equip up to 1 Mbytes private memory. The maximum number of add-on boards is 8 regardless of whether it is a slave PN or a MPC, except the host machine.

APPENDIX E
PROCESSOR NODE ADDRESS MAP

Local addr.	PN definition	MPC definition
xx000000 xx00FFFF	Private RAM	Private RAM
xx01FFFF xx7FFFFF	VME address	VME address
xxn00000 xxn1FFFF	Private RAM	Private RAM
xxn20000 xxnFFEEF	RAM expansion	RAM expansion
xxnFFEF0 xxnFFEF3	reserved	HALTR
xxnFFEF4 xxnFFEF7	PNCR	MPCCR
xxnFFEF8 xxnFFEFB	PNSR	MPCSR
xxnFFEFC xxnFFEFF	BPR	MPCBFR
xxnFFF00 xxnFFFFF	DDMA registers	reserved

Figure E.1 Local address map of a PN/MPC.

The local address map for a PN/MPC is presented in figure E.1. Notice that the register assignment for this two kind of boards are slightly different. According to the address map shown in appendix D, 'n' is 8 for MPC, 9 for PN2 and so forth. Another irregularity is that the first 64 Kbytes starting at 00000000 are mapped to the PRIVATE memory of a PN/MPC instead of the host. It is due to the reset vector requirement described in chapter 5.

APPENDIX F
REGISTER LAYOUT

F.1 Registers on a PN

The Buffer Pointer Register (BPR) of a PN is simply an index to the message

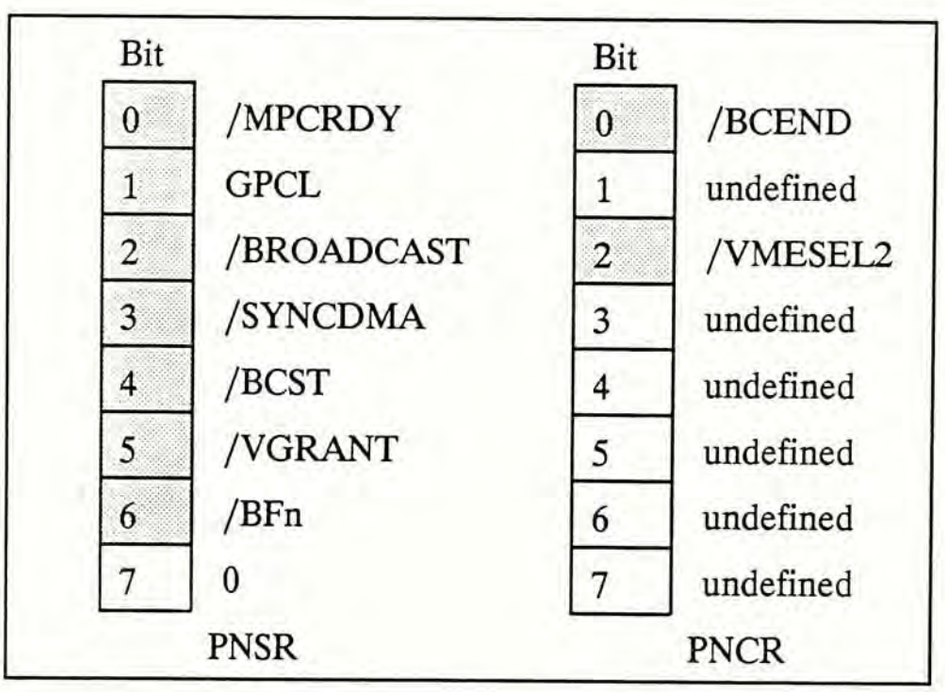


Figure F.2 Definition of PNCR and PNSR

queue. The picture is shown in figure F.1. The definition of the Status Register and Control Register is shown in figure F.2.

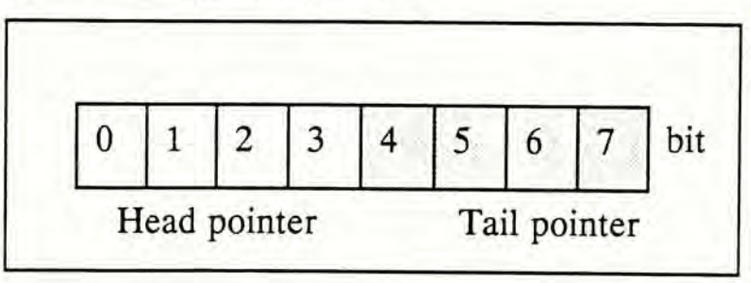


Figure F.2 Definition of the BPR.

F.2 Registers on the MPC

The definition of the Status Register and Control Register is shown in figure F.3. The definition of HALTR can be found in figure F.4.

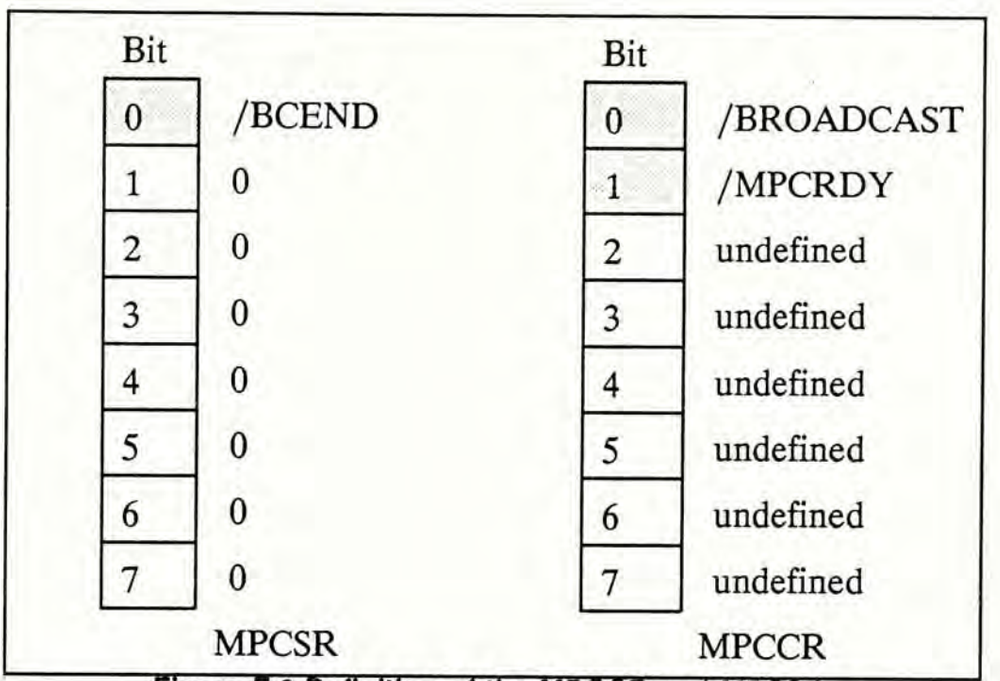


Figure F.2 Definition of the MPCCR and MPCSR.

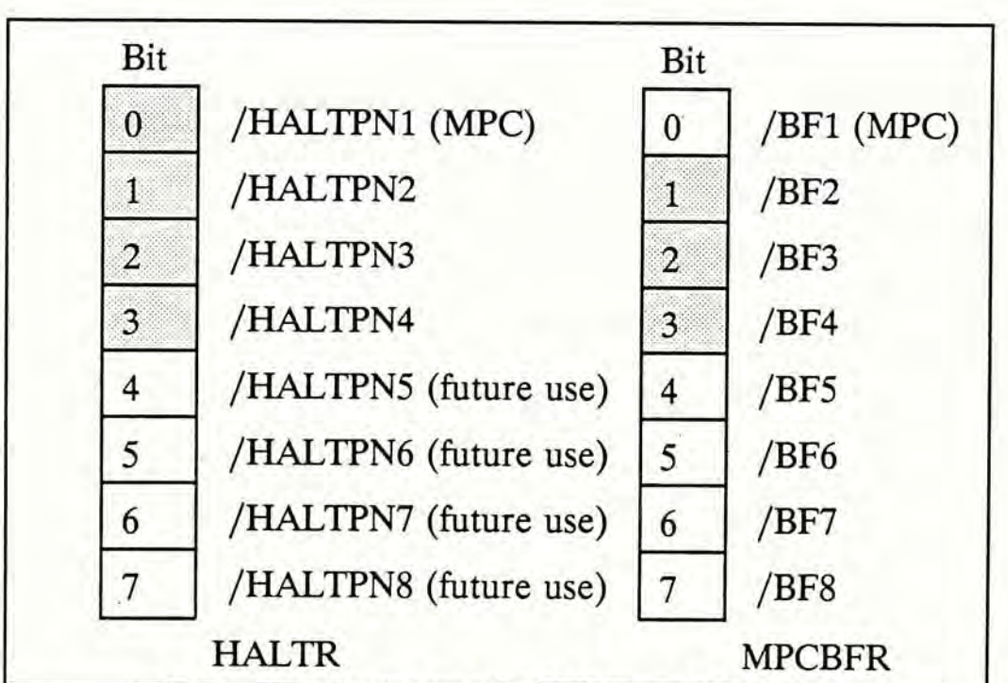


Figure F.3 Definition of the HALTR and MPCBFR.

APPENDIX G

PAL DESIGN

```
;*************************************************************
;*        P A L s - D O C U M E N T A T I O N   F I L E      *
;*************************************************************
PAL16L8 ; Register signals decoder (U4)
/PNSRS /DSKO /DS /PNCRS RW /BPRS /VMES1 /VMES2 /EN1 GND
/EN2 /VMESEL /REGBUFG /BPROP BPRG PNCRG /PNSROP /RDSK1 /RDSK0 VCC
/PNSROP = /PNSRS*RW*/EN2
/PNCRG = DS + PNCRS + EN2
/BPRG = BPRS + RW + EN2
/BPROP = RW*/BPRS*/EN2
/REGBUFG = /PNSRS*RW*/EN2 + RW*/BPRS*/EN2
/RDSK0 = EN2*/EN2  ; 0
/RDSK1 = EN2 + /EN2 ; 1
/VMESEL = /VMES1*/EN1 + /VMES2*/EN1 ; ** not a 3-state signal ! **

;-------------------------------------------------------------
PAL16R4 ; Buffer full generation (U6)
CLK A0 A1   A2 A3 B0     B1 B2   B3   GND
/EN /BPROP /FB0 NC NC /BFULL NC /FB1 /FB2 VCC
/FB0 = A0*B0 + /A0*/B0
/FB1 = A1*B1 + /A1*/B1
/FB2 = A2*B2 + /A2*/B2
/BFULL := /BPROP*/FB0*/FB1*/FB2*A3*B3 + /BPROP*/FB0*/FB1*/FB2*/A3*/B3

;-------------------------------------------------------------
PAL16L8; MPC registers decoder (U7)
/DS /MPCCR RW /RESET  /HALTRS /MPCSRS /MPCBFRS  /DSACKO  /EN1  GND
/EN2 /RDSACK1 /RDSACK0 NC HALTRG /MPCRBG /MPCBFROP /MPCSROP MPCCRG VCC
/RDSACK1      = /EN + EN ; 0
 RDSACK1.TRST = /DSACKO
/RDSACK0      = /EN*EN   ; 1
 RDSACK0.TRST = /DSACKO
/HALTRG = DS*RESET + HALTRS*RESET
/MPCBFROP = /MPCBFRS*RW
/MPCSROP = /MPCSRS*RW
/MPCRBG = /MPCBFRS*RW + /MPCSRS*RW
/MPCCRG = MPCCR + DS + RW
```

```
;-------------------------------------------------------------
PAL16L8 ; MPC HALT-REG DECODER (U10)
D0  D1 D2 D3 /HALTPN0 /HALTPN1 /HALTPN2 /HALTPN3 /RESET GND
/EN NC NC NC NC       HD3      HD2      HD1      HD0    VCC
/HD0       = /D0 + /RESET
 HD0.TRST = /EN
/HD1       = /D1 + /RESET
 HD1.TRST = /EN
/HD2       = /D2 + /RESET
 HD2.TRST = /EN
/HD3       = /D3 + /RESET
 HD3.TRST = /EN

;-------------------------------------------------------------
PAL16R4 ; PN Sender logic (U14)
CLK /BROADCAST /HALTI /ACK SYNCDMA /DSACK0 /DSACK1 /DTC /DONE GND
/EN /RESET     NC     /S3  /S2     /S1     /S0     PCL  /HB   VCC
/S0 := RESET*S2*S1*S0*HALTI*/BROADCAST   + RESET*S2*S1*/S0
     + RESET*S2*/S1*/S0*/SYNCDMA    + RESET*S2*/S1*/S0*DSACK0*DSACK1
     + RESET*/S2*/S1*S0                  + RESET*S2*/S1*S0*/DTC*DONE
     + RESET*/S2*/S1*/S0*/SYNCDMA
/S1 := RESET*S2*S1*/S0*/ACK*SYNCDMA      + RESET*S2*/S1*/S0
     + RESET*/S2*/S1*S0 + RESET*/S2*/S1*/S0 + RESET*S2*/S1*S0*DONE
/S2 := RESET*S2*/S1*/S0*SYNCDMA*/DSACK0 + RESET*/S2*/S1*S0*SYNCDMA
     + RESET*S2*/S1*/S0*SYNCDMA*/DSACK1 + RESET*/S2*/S1*S0*/SYNCDMA
     + RESET*/S2*/S1*/S0*/SYNCDMA
/HB = /HALTI*/BROADCAST
 PCL.TRST = S2*/S1*/S0
/PCL       = /HALTI*HALTI          ; GPCL=1      3-STATE

;-------------------------------------------------------------
PAL16R4 ; PN Sender logic (U14)
CLK /BROADCAST /HALTI  /TEM SYNCDMA /DSACK0 /DSACK1 /DTC /DONE GND
/EN /ACKIN     /ACKOUT /S3  /S2     /S1     /S0     PCL  /HB   VCC
/S0 := S2*S1*S0*HALTI*/BROADCAST   + S2*S1*/S0
     + S2*/S1*/S0*/SYNCDMA         + S2*/S1*/S0*DSACK0*DSACK1
     + /S2*/S1*S0                  + S2*/S1*S0*/DTC*DONE
     + /S2*/S1*/S0*/SYNCDMA
/S1 := S2*S1*/S0*/TEM + S2*/S1*/S0 + /S2*/S1*S0 + S2*/S1*S0*DONE
     + /S2*/S1*/S0
/S2 := S2*/S1*/S0*SYNCDMA*/DSACK0  + S2*/S1*/S0*SYNCDMA*/DSACK1
     + /S2*/S1*S0*/SYNCDMA         + /S2*/S1*S0*SYNCDMA
     + /S2*/S1*/S0*/SYNCDMA
/HB = /HALTI*/BROADCAST
/PCL = /HALTI*HALTI          ; GPCL=1      3-STATE
 PCL.TRST = S2*/S1*/S0
/ACKOUT = /ACKIN*SYNCDMA  ; External feedback,
```

```
;-----------------------------------------------------------------
PAL16R4 ; PN Receiver logic (U15)
CLK33 /BCST  /ACK GPCL /AS /DSVME /DTC /DONE   /HB GND
/EN   /RESET /REQ /TM  /S2 /S1    /S0  SYNCDMA PCL VCC
/S0 := RESET*S2*S1*S0*/HB + RESET*S2*S1*/S0 + RESET*S2*/S1*/S0*ACK
     + RESET*/S2*/S1*S0*/AS + RESET*/S2*S1*/S0*DSVME
/S1 := RESET*S2*S1*/S0*/BCST + RESET*S2*/S1*/S0 + RESET*S2*/S1*S0
     + RESET*/S2*/S1*S0*AS + RESET*/S2*S1*S0*/DTC*DONE
/S2 := RESET*S2*/S1*S0*GPCL + RESET*/S2*/S1*S0 + RESET*/S2*S1*/S0
     + RESET*/S2*S1*S0*DTC*DONE
/TM := RESET*S2*S1*S0*/HB + RESET*S2*S1*/S0 + RESET*S2*/S1*/S0*ACK
     + RESET*S2*/S1*S0*GPCL + RESET*/S2*/S1*S0*AS
     + RESET*/S2*S1*/S0*/DSVME + RESET*/S2*S1*S0*DTC*DONE
/REQ      = /BCST + BCST      ; /REQ = 0 if S2*/S1*/S0 otherwise X
 REQ.TRST = S2*/S1*/S0
/SYNCDMA      = /BCST + BCST  ; 0:not ready, 1:pulled up, X:OK
 SYNCDMA.TRST = /TM
/PCL      = /BCST*BCST        ; /PCL = 1 if /S2*/S1*S0 otherwise X
 PCL.TRST = /S2*/S1*S0

;-----------------------------------------------------------------
PAL16R4 ; PN Receiver logic (U15)          [[[ OLD VERSION ]]]
CLK33 /BCST /ACK GPCL /AS /DSVME /DTC /DONE   /HB GND
/EN   TEMP  /REQ /S3  /S2 /S1    /S0  SYNCDMA PCL VCC
/S0 := S2*S1*S0*/HB + S2*S1*/S0 + S2*/S1*/S0*ACK + /S2*/S1*S0*/AS
     + /S2*S1*/S0*DSVME
/S1 := S2*S1*/S0*/BCST + S2*/S1*/S0 + S2*/S1*S0 + /S2*/S1*S0*AS
     + /S2*S1*S0*/DTC*DONE
/S2 := S2*/S1*S0*GPCL + /S2*/S1*S0 + /S2*S1*/S0 + /S2*S1*S0*DTC*DONE
/REQ      = /BCST + BCST      ; /REQ = 0 if S2*/S1*/S0 otherwise X
 REQ.TRST = S2*/S1*/S0
/TEMP = S2*/S1*/S0 + /S2*S0 + S2*S1*/S0
/SYNCDMA      = /BCST + BCST  ; 0:not ready, 1:pulled up, X:OK
 SYNCDMA.TRST = /TEMP
/PCL      = /BCST*BCST        ; /PCL = 1 if /S2*/S1*S0 otherwise X
 PCL.TRST = /S2*/S1*S0
```

```
;-----------------------------------------------------------------
PAL16L8 ; PN Sender FSM decoder (U16)
S0 S1    S2    S3   /VMEEN  NC       NC    NC     NC     GND
NC GPCL /REQ /DSACK0 /DSACK1 /VMEDEN /VDS /TEMP /BCST VCC
/TEMP = /S2*/S1*/S0 + /S2*/S1*S0 + S2*/S1*/S0 + S2*/S1*S0 + S2*S1*/S0
/BCST       = /VMEEN + VMEEN      ; 0
 BCST.TRST = /TEMP
/GPCL       = /VMEEN*VMEEN        ; 1
 GPCL.TRST = S2*/S1*/S0
/REQ       = /VMEEN + VMEEN       ; 0
 REQ.TRST = S2*S1*/S0
/DSACK0       = VMEEN*/VMEEN      ; 1
 DSACK0.TRST = S2*/S1*S0
/DSACK1       = VMEEN + /VMEEN    ; 0
 DSACK1.TRST = S2*/S1*S0
/VDS = /BCST*/S2*/S1 + /S2*/S1*S0   ; ==> /S2*/S1
/VMEDEN = /VMEEN + /BCST*/S2*/S1 + /S2*/S1*S0

;-----------------------------------------------------------------
PAL16L8 ; VME address decoder (U17)
DIP0 DIP1 DIP2 DIP3 /VAS A20  A21 A22 A23   GND
/EN1 NC   B1   B2   B3   B4   NC  NC  /BSEL VCC
/B1 = A20*/DIP0 + /A20*DIP0          ; internal feedback
/B2 = A21*/DIP1 + /A21*DIP1          ; internal feedback
/B3 = A22*/DIP2 + /A22*DIP2          ; internal feedback
/B4 = A23*/DIP3 + /A23*DIP3          ; internal feedback
/BSEL = B1*B2*B3*B4*/VAS*/EN1

;-----------------------------------------------------------------
PAL16L8 ; VME to PN decoder 1 (U25)
/BERRI /WRITEI /ASI /BERRVMEI RWI  /ASVMEI /VAS   /VMEEN /VGRANT   GND
NC     NC      DDIR /ASVMEO   /ASO /WRITEO /BERRO RWO    /BERRVMEO VCC
/BERRVMEO       = /BERRI
 BERRVMEO.TRST = /VMEEN*VGRANT       ; as a slave, output BERR
/RWO       = /WRITEI
 RWO.TRST = /VMEEN*VGRANT
/BERRO       = /BERRVMEI
 BERRO.TRST = /VMEEN*/VGRANT         ; as a master
/WRITEO      = /RWI
 WRITEO.TRST = /VMEEN*/VGRANT
/ASO       = /ASVMEI
 ASO.TRST = /VMEEN*VGRANT
/ASVMEO       = /ASI*/VAS           ; /VAS
 ASVMEO.TRST = /VMEEN*/VGRANT
/DDIR = VGRANT*WRITEI + /VGRANT*/RWI
```

```
;-------------------------------------------------------------
PAL16L8 ; VME to PN decoder 1 (U25)
/BERRI /WRITEI /ASI /BERRVMEI RWI  /ASVMEI /VAS   /VMEEN /VGRANT   GND
NC     NC      NC   /ASVMEO   /ASO /WRITEO /BERRO RWO    /BERRVMEO VCC
/BERRVMEO       = /BERRI
 BERRVMEO.TRST = /VMEEN*VGRANT       ; as a slave, output BERR
/RWO       = /WRITEI
 RWO.TRST = /VMEEN*VGRANT
/BERRO       = /BERRVMEI
 BERRO.TRST = /VMEEN*/VGRANT         ; as a master
/WRITEO       = /RWI
 WRITEO.TRST = /VMEEN*/VGRANT
/ASO       = /ASVMEI
 ASO.TRST = /VMEEN*VGRANT
/ASVMEO       = /ASI*/VAS
 ASVMEO.TRST = /VMEEN*/VGRANT

;-------------------------------------------------------------
; DS1 | 0 0 0 0 1 1 1 1
; DS0 | 0 0 1 1 0 0 1 1
; LWD | 0 1 0 1 0 1 0 1
; ----+----------------
; SI1 | 0 0 1 1 1 1 1 1
; SI0 | 0 1 0 0 0 0 1 1
; A0  | 0 0 0 0 1 1 1 1
; DK1 | 0 0 x 1 x 1 1 1
; DK0 | 0 1 0 0 0 0 1 1
PAL16L8 ; VME to PN decoder 2 (U26)
/DS0 /DS1      /LWORD /DTACK /VMEEN  /VGRANT NC  NC   NC   GND
NC   /NEWSIZ1 NEWA0  A0      /DSACK1 /DSACK0 /DS SIZ1 SIZ0 VCC
/SIZ0       = /DS0*/LWORD + /DS1*DS0  + DS1*/DS0
 SIZ0.TRST = /VMEEN*VGRANT                          ; as a slave
/SIZ1       = /DS0*/LWORD + /DS1*/DS0 + /DS1*LWORD
 SIZ1.TRST = /VMEEN*VGRANT
/DS       = /DS0 + /DS1
 DS.TRST = /VMEEN*VGRANT
/DSACK0      = /DTACK*/DS0*/LWORD + /DTACK*/DS1*DS0 + /DTACK*DS1*/DS0
 DSACK0.TRST = /VMEEN*/VGRANT
/DSACK1            =  /DTACK*/DS0*/LWORD  +  /DTACK*/DS1*/DS0  +
/DTACK*/DS1*LWORD
 DSACK1.TRST = /VMEEN*/VGRANT
/A0       = /DS0                    ; void, should be deleted !!!!
 A0.TRST = /VMEEN*VGRANT
/NEWA0       = /DS1
 NEWA0.TRST = /VMEEN*VGRANT
/NEWSIZ1       = /DS1*/DS0
 NEWSIZ1.TRST = /VMEEN*VGRANT

;-------------------------------------------------------------
;                    S1 | 0 0 0 0 1 1 1 1
;                    S0 | 0 0 1 1 0 0 1 1
;                    A0 | 0 1 0 1 0 1 0 1
;                    ---+----------------
;                    D1 | 0 0 0 0 1 0 1 1
;                    D0 | 0 0 0 0 0 1 1 1
```

```
PAL16L8 ; PN to VME decoder (U27)
/DSACK0 /DSACK1 /DS SIZ0 SIZ1 /VMEEN /VGRANT /VDS A0   GND
NC      NC      NC  NC   NC   /DTACK /LWORD  /DS1 /DS0 VCC
/LWORD       = /SIZ0*/SIZ1
 LWORD.TRST = /DS*/VMEEN*/VGRANT
/DTACK       = /DSACK0 + /DSACK1
 DTACK.TRST = /VMEEN*/VGRANT
/DS1         = /SIZ1 + /SIZ0*A0 + /VDS
 DS1.TRST = /DS*/VMEEN*/VGRANT
/DS0         = /SIZ1 + /SIZ0*/A0 + /VDS
 DS0.TRST = /DS*/VMEEN*/VGRANT

;---------------------------------------------------------------
PAL16L8 ; SRAM controller 1 (U36)
RW    A1   A0   SIZ0 SIZ1 /RAMS /DSKOUT /RDSK0 /RDSK1 GND
/REGS DSK1 DSK0 /E3  /E2  /E1   /E0     /W     /G     VCC
/G = RW*/RAMS
/W = /RW*/RAMS
/E0 = /A1*/A0*/RAMS
/E1 = /A1*A0*/RAMS + /A1*SIZ0*/RAMS + /A1*/SIZ1*/RAMS
/E2 = A1*/A0*/RAMS + /A1*/A0*/SIZ0*/SIZ1*/RAMS
/E3 = A1*A0*/RAMS + A1*SIZ0*/RAMS + /A0*/SIZ1*/SIZ0*/RAMS
/DSK0        = /RDSK0*/REGS + /RAMS*SIZ0
 DSK0.TRST = /DSKOUT
/DSK1        = /RDSK1*/REGS + /RAMS*SIZ1
 DSK1.TRST = /DSKOUT

;---------------------------------------------------------------
PAL16L8 ; Local address decoder (U39)
A20  A21      A22 A23 /AS DIP0 DIP1     DIP2    DIP3   GND
/EN1 /VMESEL1 L1  L2  L3  L4   /VMESEL2 /VMESEL /LOCAL VCC
/L1 = A20*/DIP0 + /A20*DIP0          ; internal feedback
/L2 = A21*/DIP1 + /A21*DIP1          ; internal feedback
/L3 = A22*/DIP2 + /A22*DIP2          ; internal feedback
/L4 = A23*/DIP3 + /A23*DIP3          ; internal feedback
/LOCAL = L1*L2*L3*L4*/AS*/EN1 + /A20*/A21*/A22*/A23*/AS*/EN1
/VMESEL = /AS*/L1*/EN1 + /AS*/L2*/EN1 + /AS*/L3*/EN1 + /AS*L4*/EN1
/VMESEL1 = /AS*/L1*/EN1 + /AS*/L2*/EN1 + /AS*/L3*/EN1 + /AS*/L4*/EN1
/VMESEL2 = /AS*/L1*LOCAL*/EN1 + /AS*/L2*LOCAL*/EN1
         + /AS*/L3*LOCAL*/EN1 + /AS*/L4*LOCAL*/EN1
;---------------------------------------------------------------
PAL16L8 ; local address decoder 2 (U40)
A2     A3 A8      A9     A10    A11   A17   A18   A19   GND
/LOCAL NC /HALTRS /PNCRS /PNSRS /DMAS /BPRS /REGS /RAMS VCC
/HALTRS = /A2*/A3*A11*A10*A9*/A8*A17*A18*A19*/LOCAL
/PNCRS  = A2*/A3*A11*A10*A9*/A8*A17*A18*A19*/LOCAL
/PNSRS  = /A2*A3*A11*A10*A9*/A8*A17*A18*A19*/LOCAL
/DMAS   = A8*A9*A10*A11*A17*A18*A19*/LOCAL
/BPRS   = A2*A3*A11*A10*A9*/A8*A17*A18*A19*/LOCAL
/REGS   = A17*A18*A19*/LOCAL
/RAMS   = /A17*/LOCAL + /A18*/LOCAL + /A19*/LOCAL
```

```
;---------------------------------------------------------------
PAL16R4 ; Local bus arbiter (U41)
CK  /BR0 /BR1 /BGACK0 /BGACK1 /BG /RESET NC     NC   GND
/EN /BG1 /BG0 NC      /S2     /S1 /S0    /BGACK /BR  VCC
/S0 := RESET*S2*S1*S0*/BR0 + RESET*S2*S1*/S0 + RESET*/S2*S1*/S0*BGACK0
/S1    :=    RESET*S2*S1*S0*BR0*/BR1    +    RESET*S2*/S1*S0    +
RESET*/S2*/S1*S0*BGACK1
/S2    :=    RESET*S2*S1*/S0*/BG    +    RESET*/S2*S1*/S0    +
RESET*/S2*S1*S0*/BGACK0
     +    RESET*/S2*S1*S0*/BGACK1    +    RESET*S2*/S1*S0*/BG    +
RESET*/S2*/S1*S0
/BR = S2*S1*/S0     + S2*/S1*S0
/BGACK = /S2*S1*S0
/BG0 = /S2*S1*/S0
/BG1 = /S2*/S1*S0

;---------------------------------------------------------------
PAL16R4 ; Local bus controller (U42)
CLK33 /BG    /VMESEL /BSEL /VGRANT /LOCAL /BERR /DSACK0 /DSACK1 GND
/EN   /VMEEN /RESET  /CDEN S2      S1     S0    /BGACK  /VSEL   VCC
/S0 := RESET*S2*S1*S0*/VMESEL*BSEL + RESET*S2*S1*/S0
     + RESET*S2*/S1*/S0*/VGRANT + RESET*/S2*S1*/S0 + RESET*/S2*/S1*/S0
     + RESET*/S2*/S1*/S0*/BSEL
/S1 := RESET*S2*S1*S0*/BSEL + RESET*S2*/S1*S0
     + RESET*/S2*/S1*S0*BERR*DSACK0*DSACK1 + RESET*/S2*S1*/S0*/BG
     + RESET*/S2*/S1*/S0*BERR*DSACK0*DSACK1
     + RESET*S2*S1*/S0*/VGRANT + RESET*S2*/S1*/S0*/VGRANT
     + RESET*/S2*/S1*/BSEL
/S2 := RESET*S2*/S1*S0*/BG + RESET*S2*S1*S0*/LOCAL*BSEL
     + RESET*S2*S1*/S0*/BSEL + RESET*/S2*/S1*/S0*BERR*DSACK0*DSACK1
     + RESET*/S2*/S1*S0*BERR*DSACK0*DSACK1
     + RESET*/S2*S1*S0*BERR*DSACK0*DSACK1 + RESET*/S2*S1*/S0
     + RESET*/S2*/S1*/BSEL
/CDEN := RESET*S2*S1*S0*/LOCAL*BSEL +
         RESET*/S2*S1*S0*BERR*DSACK0*DSACK1
       + RESET*S2*S1*/S0*/VGRANT + RESET*S2*/S1*/S0*VGRANT
/VMEEN      = /BG + BG   ; /VMEEN=0 if /S2*/S1, otherwise X
 VMEEN.TRST = S2*/S1     ; !!! wrong !!! should be /S2*/S1
/BGACK      = /BG + BG   ; /BGACK=0 if /S2*/S1, otherwise X
/VSEL      = /BG + BG    ; 0
 VSEL.TRST = S2*/S0
```

```
;-------------------------------------------------------------
PAL16R4 ; VME master logic (U43)
CLK33 /VSEL /BG3IN /DTACK /VBERR /BROADCAST /BCST /OS0  /RESET GND
/EN   /VAS  /VMEEN /U43TEMP     S2     S1   S0    /BBSY /VBR VCC
/S0 := RESET*S2*/S1*S0 + RESET*S2*/S1*/S0 + RESET*S2*S1*/S0
     + RESET*/S2*S1*/S0*BROADCAST + RESET*/S2*/S1*/S0*DTACK*VBERR
/S1 := RESET*S2*S1*S0*/VSEL + RESET*S2*/S1*S0 + RESET*S2*/S1*/S0*BG3IN
     + RESET*/S2*S1*/S0*BROADCAST + RESET*/S2*/S1*/S0
     + RESET*/S2*S1*S0*BCST
/S2 := RESET*S2*S1*/S0*BG3IN + RESET*/S2*S1*/S0 + RESET*/S2*S1*S0
     + RESET*/S2*/S1*/S0
/U43TEMP := S2*/S1*/S0*/BG3IN       + /S2*S1*/S0      + S2*S1*/S0
          + /S2*/S1*/S0*DTACK*VBERR + /S2*S1*S0*/BCST
/VAS = RESET*/S2*/S1*/S0
/VMEEN      = /VSEL + VSEL
 VMEEN.TRST = /OS0
/BBSY       = /VSEL + VSEL
 BBSY.TRST = /U43TEMP
/VBR        = /VSEL + VSEL
 VBR.TRST = /VSEL*S2

;-------------------------------------------------------------
PAL16R4 ; VME slave logic (U44)
CLK33 /BSEL   /BG    /AS     /BG3IN IS0 IS1 IS2  /U43TEMP GND
/EN   /VGRANT /RESET /BG3OUT /OS0   S1  S0  /CBR /BGACK   VCC
/S0 := RESET*S1*S0*/BSEL + RESET*S1*/S0 + RESET*/S1*/S0*/AS
/S1 := RESET*S1*/S0*/BG + RESET*/S1*/S0*/AS + RESET*S1*S0*/BG3IN
     + RESET*/S1*S0*/BG3IN
/OS0 := RESET*IS2*IS1*/IS0*BG3IN
/BG3OUT := RESET*S1*S0*/BG3IN + RESET*/S1*S0      ; 1-st board diff.
/CBR       = /BG + BG                             ; 0
 CBR.TRST = S1*/S0
/BGACK      = /BG + BG                            ; 0
 BGACK.TRST = /S1*/S0
/VGRANT      = /BG + BG                          ; 0
 VGRANT.TRST = /U43TEMP
```

```
;-------------------------------------------------------------
PAL16R4 ; BERR generation (U45)
CLK33 Q  /VGRANT /DSACK0 /DSACK1 /AS IS0 IS1  IS2   GND
/EN   /A /CLR    /S3     /S2     /S1 /S0  /BR1 /BERR VCC
/S0 := S3*S2*/S1*S0*/AS*DSACK0*DSACK1*VGRANT      ; 1101*/C1
     + S3*S2*/S0*/AS*DSACK0*DSACK1*VGRANT         ; 11?0*/C1
     + S3*/S2*S1*/AS*DSACK0*DSACK1*VGRANT         ; 101?*/C1
     + /S3*/S2*/S1*/AS*DSACK0*DSACK1*VGRANT       ; 000?*/C1
     + /S3*S2*S1*S0                               ; 0111
/S1 := S3*S2*S1*S0 + /S3*S2*S1*/S0                ; 1111 + 0110
     + S3*S2*S1*/S0*/AS*DSACK0*DSACK1*VGRANT      ; 1110*/C1
     + /S1*/S0*/AS*DSACK0*DSACK1*VGRANT           ; ??00*/C1
     + S3*/S2*/S1*S0*Q*/AS*DSACK0*DSACK1*VGRANT   ; 1001*C3
     + /S3*/S2*S1*S0*/Q*/AS*DSACK0*DSACK1*VGRANT  ; 0011*C2
     + /S3*/S2*/S1*S0*/AS*DSACK0*DSACK1*VGRANT    ; 0001*/C1
     + /S3*S2*/S1*S0*Q*/AS*DSACK0*DSACK1*VGRANT   ; 0101*C3
/S2 := S3*S2*/S1*/S0*/Q*/AS*DSACK0*DSACK1*VGRANT  ; 1100*C2
     + S3*/S2*/S1*/AS*DSACK0*DSACK1*VGRANT        ; 100?*/C1
     + /S2*S1*/AS*DSACK0*DSACK1*VGRANT            ; ?01?*/C1
     + /S3*/S2*/S1*S0*/AS*DSACK0*DSACK1*VGRANT    ; 0001*/C1
     + /S3*/S2*/S1*/S0*Q*/AS*DSACK0*DSACK1*VGRANT ; 0000*C3
/S3 := S3*/S2*S1*/S0*/Q*/AS*DSACK0*DSACK1*VGRANT  ; 1010*C2
     + /S3*/S2*/AS*DSACK0*DSACK1*VGRANT           ; 00??*/C1
     + /S3*S2*/S1*/AS*DSACK0*DSACK1*VGRANT        ; 010?*/C1
     + /S3*S2*S1*S0                               ; 0111
/BR1 = IS2*/IS1*IS0 + /IS2*IS1*/IS0
/BERR       = Q + /Q
 BERR.TRST = /S3*S2*S1
/A = S3*S2*S1*/S0 + S3*/S2*/S1*/S0 + S3*/S2*S1*S0 ; 1110 + 1000 + 1011
   + /S3*/S2*S1*/S0 + /S3*/S2*/S1*S0              ; 0010 + 0001 + 0100
   + /S3*S2*/S1*/S0
/CLR = S3*S2*S1*S0

;-------------------------------------------------------------
PAL16R4 ; DSACK generation + SRAM (U46)
CK  /AS  /DMASEL /VMESEL IS1 IS2 /BSEL     NC       DEL    GND
/EN /ASO /DSKO   NC      S1  S0  /DSACKOUT /BGACK0 /VMEEN VCC
/DSACKOUT := DSACKOUT*/AS*DMASEL*VMESEL + /DSACKOUT*/AS     ; obsolete
/BGACK0       = /AS + AS
 BGACK0.TRST = /IS2*/IS1
/VMEEN       = /AS + AS
 VMEEN.TRST = /IS2*/IS1
/ASO = /BSEL*/VMEEN*DSACKOUT + /DSACKOUT
     + /DSACKOUT*/VMEEN
/DSKO       = S1*/S0
 DSKO.TRST = DMASEL*VMESEL
/S0 := /S1*S0*DEL + /S1*/S0 + S1*/S0*/AS*/VMEEN + /S1*S0*/DEL
     + S1*S0*/AS*/VMEEN*/DEL
/S1 := S1*S0*/AS*/VMEEN*DEL + /S1*S0*DEL

;-------------------------------------------------------------
```

```
PAL16L8 ; CPU bus converter (U57)
/DSIN SIZ1IN SIZ0IN  /UDSIN  /LDSIN  /DMASEL /OWN    /CPUAS CPURW GND
NC    NC     /BUFDIR /LDSOUT /UDSOUT SIZ0OUT SIZ1OUT /DSOUT /BUSAS VCC
/DSOUT      = /UDSIN + /LDSIN
 DSOUT.TRST = /OWN
/SIZ1OUT     = LDSIN
 SIZ1OUT.TRST = /OWN
/SIZ0OUT     = /UDSIN + LDSIN
 SIZ0OUT.TRST = /OWN
/UDSOUT     = /DSIN*SIZ1IN
 UDSOUT.TRST = /DMASEL
/LDSOUT     = /DSIN*SIZ1IN + /DSIN*SIZ0IN
 LDSOUT.TRST = /DMASEL
/BUSAS      = /DMASEL + DMASEL
 BUSAS.TRST = /CPUAS
/BUFDIR = CPURW
;---------------------------------------------------------------
PAL16L8 ; Miscellaneous functions (U67)
/RES /INHIBIT RW    /HALT  /BERR NC NC NC    NC     GND
NC   /BEC1    /BEC0 /RETRY NC    NC NC NOTRW /RESET VCC
/RESET = /RES + /INHIBIT
/NOTRW = RW
/BEC1 = /RETRY + /BERR + /RES + /INHIBIT
/BEC0 = /RETRY + /HALT + /RES + /INHIBIT
/RETRY = /BERR*/HALT + /BERR*/RETRY + /HALT*/RETRY

;---------------------------------------------------------------
PAL16L8 ; VME signals demux/mux (U68)
/HALT0 /HALT1 /HALT2 /HALT3 /HALT4 DIP0 DIP1 DIP2 DIP3 GND
/BF    NC     NC     /HALTI /BF0   /BF1 /BF2 /BF3 /BF4 VCC
/HALTI = /HALT0*/DIP0*/DIP1*/DIP2*/DIP3
       + /HALT1*DIP0*/DIP1*/DIP2*/DIP3 + /HALT2*/DIP0*DIP1*/DIP2*/DIP3
       + /HALT3*DIP0*DIP1*/DIP2*/DIP3 + /HALT4*/DIP0*/DIP1*DIP2*/DIP3
/BF0 = /BF
 BF0.TRST = /DIP0*/DIP1*/DIP2*/DIP3
/BF1 = /BF
 BF1.TRST = DIP0*/DIP1*/DIP2*/DIP3
/BF2 = /BF
 BF2.TRST = /DIP0*DIP1*/DIP2*/DIP3
/BF3 = /BF
 BF3.TRST = DIP0*DIP1*/DIP2*/DIP3
/BF4 = /BF
 BF4.TRST = /DIP0*/DIP1*DIP2*/DIP3

;---------------------------------------------------------------
PAL16L8 ; VME buffer control (U69)
/VMEDEN SIZ0 SIZ1 A0   A1  NC  NC   NC  NC  GND
NC      NC   NC   NC   /E5 /E4 /E23 /E1 /E0 VCC
/E0 = A0*/A1*/SIZ0*/VMEDEN + /A0*/A1*/SIZ1*/VMEDEN
/E1 = /A0*/A1*/SIZ0*/VMEDEN + /A0*/A1*/SIZ1*/VMEDEN
/E23 = /SIZ0*/SIZ1*/A0*/A1*/VMEDEN
/E4 = A0*A1*/SIZ0*/VMEDEN + /A0*A1*/SIZ1*/VMEDEN
/E5 = /A0*A1*/SIZ0*/VMEDEN + /A0*A1*/SIZ1*/VMEDEN
```

APPENDIX H
COMMUNICATION SUB-BUS

H.1 Signal definition

Signal	Meaning
/BROADCAST	BROADCASTing using 1-to-N DMA in progress.
SYNCDMA	SYNChronization signal for 1-to-N DMA.
GPCL	Global PCL (means ready) derived from the DMAC.
/MPCRDY	MPC is ReaDY for servicing new requests.
/BCST	BroadCasting using 1-to-N DMA STarted.
/BCEND	BroadCasting using 1-to-N DMA ENDed.
/HALTPNn	HALTs the PN with number n.

H.2 Pin assignment

Signal	**P2 pin**	**Signal**	**P2 pin**
/BROADCAST	C1	/HALTPN3	C12
SYNCDMA	C2	/HALTPN4	C13
GPCL	C3	/BF1	C17
/MPCRDY	C4	/BF2	C18
/BCST	C5	/BF3	C19
/BCEND	C6	/BF4	C20
/HALTPN0	C9		
/HALTPN1	C10		
/HALTPN2	C11		

APPENDIX I
FEASIBILITY OF TASK DISTRIBUTION PLAN

A major principle of task distribution on a multiprocessor system is to make sure that the communication overhead is small relative to the computation workload. We must pin-point the cases in which the computation should be done locally instead of distributing it because of the excessive communication overhead.

Assume the PNs are prioritized from the view point of the bus arbiter. Let PN(1) has the highest priority while PN(n) has the lowest. This configuration is easily achievable because the VMEbus supports fixed priority interrupt and a daisy-chained acknowledgement propagation mechanism is available.

If the task is evenly distributed to the PNs at the beginning, it is obvious that they will finish the first stage of their jobs almost simultaneously. Then the PNs will compete for the use of the global bus for getting messages. PN(1) will win and the bus requests will be serialized. However, if the computation workload is not large compared with the communication overhead, PN(1) will need more information before all the bus requests have been serviced. Since PN(1) has the highest priority, he will get the bus, obtain more information and then start the third stage of the computation. At this moment, PN(n) just has not started his second stage of computation. Eventually, PN(1) will finish its job far ahead of PN(n).

Now, the symptom of excessive communication overhead can be identified: low priority PNs need substantially longer time to finish their jobs than the high priority PNs. We can check this condition easily: Each PNs should send a message to the host machine when it has completed its share of the task. Hence, the start and finish time of the PNs are available. We can check the above condition to estimate the feasibility of the task distribution plan.

APPENDIX J
COMMUNICATION PRIMITIVES

Four proposed communication primitives are discussed below. They are implemented in the form of device drivers in the communication kernel.

SendMessage(	**SenderPID: integer,**	**{ input }**
	ReceiverPID: integer,	**{ input }**
	MessagePtr: pointer,	**{ input }**
	MessageLength: integer,	**{ input }**
	MessageType: integer,	**{ input }**
	BlockingOption: integer)	**{ input }**

This primitive sends out a message when it is called. SenderPID and ReceiverPID are the logical PID number of the sending and receiving PN respectively. If ReceiverPID is BROADCASTING, which is a pre-defined constant equal to zero, then the message will be broadcasted. Otherwise, it is a point-to-point message. MessagePtr and MessageLength describes the location and length of the message body. MessageType can be EXPRESS or NORMAL, both are pre-defined constants. BlockingOption can be BLOCKING or NON_BLOCKING.

ReceiveMessage(	**SenderPID: integer,**	**{ input/output }**
	MessagePtr: pointer,	**{ output }**
	MessageLength: integer,	**{ output }**
	MessageType: integer,	**{ input/output }**
	BlockingOption: integer)	**{ input }**

This primitive tries to accept a message when it is called. SenderPID may be ANY or a valid logical PID. The caller declares from whom the next message is accepted. When the call returns, the logical PID of the actual sender is available if the ANY option is given when the primitive is called. MessagePtr and MessageLength describes the message body received. Note that MessagePtr points to a local memory location. The MPC initiates a DMA to transfer a message to the receiving PN before the ReceiveMessage primitive completes.

If the BlockingOption is NON_BLOCKING, the returned MessageLength is 0 and MessagePtr is NULL if there is no message available.

```
GetStatus(  MyPID: integer,          { input }
            NIMQLength: integer,     { output }
            EIMQLength: integer,     { output }
            BIMQLength: integer)     { output }
```

GetStatus returns the status of the message queues for a PN. MyPID is the logical PID of the caller. The queue lengths of the three message queues are returned.

```
FlushQueue( MyPID: integer,          { input }
            FlushNIMQ: integer,      { input }
            FlushEIMQ: integer,      { input }
            FlushBIMQ: integer)      { input }
```

A PN may clear/initialize its message queues by calling this primitive. If the NIMQ is to be flushed, then the user should set the FlushNIMQ argument to ON, otherwise, it is set to OFF. Other queues are treated similarly.

APPENDIX K
PHOTOGRAPHS OF SM3

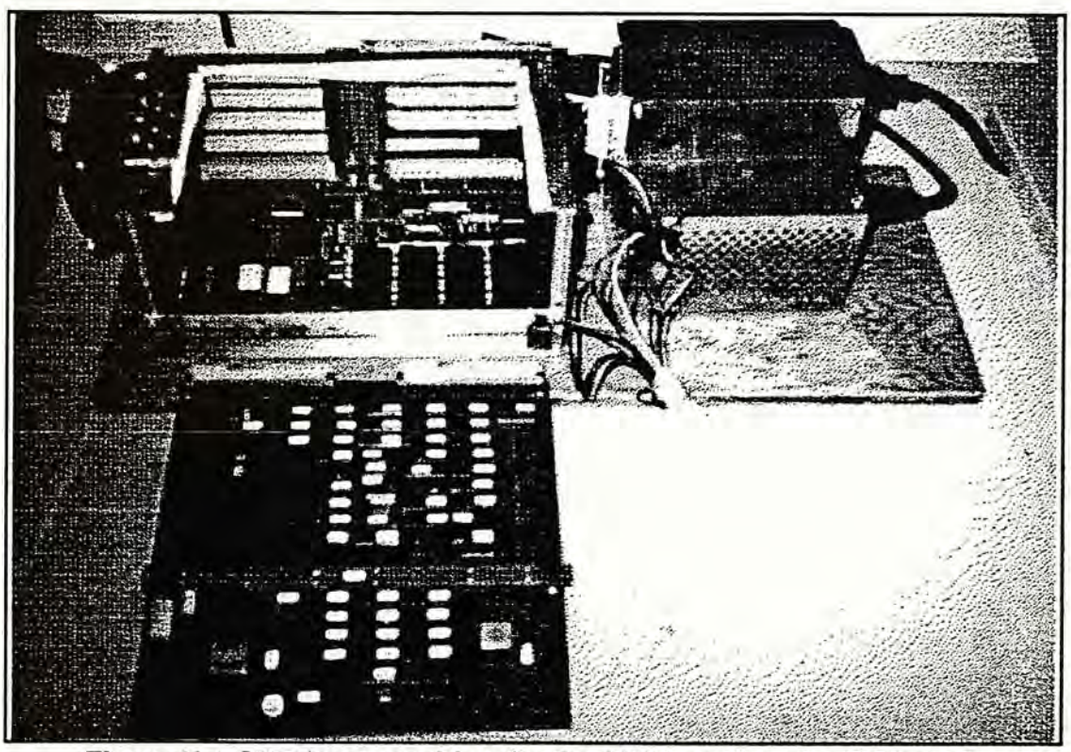


Figure K.1 SM3 host machine (in the VME-rack) and a PN board.

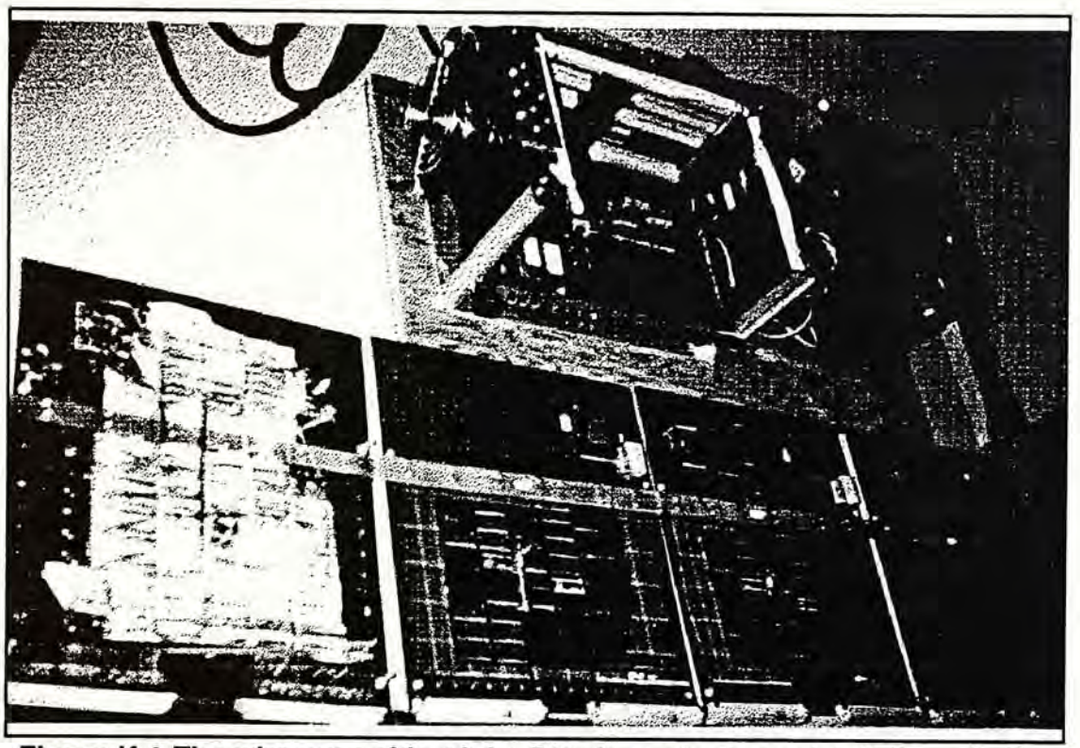


Figure K.2 The wire-wrap side of the PN shown in figure J.1 (leftmost), the MPC (middle-left), and two PNs (rightmost).

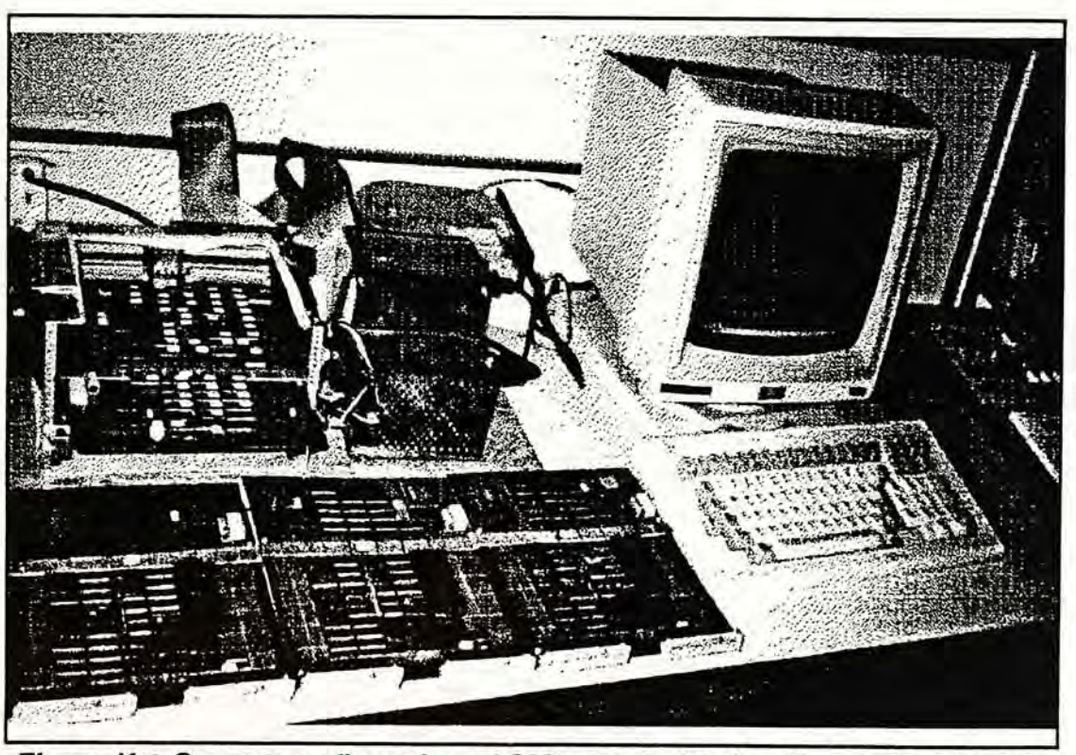


Figure K.3 Current configuration of SM3: hard disk (back), terminal (right), and the VME rack (left).

APPENDIX L
PROTOCOL STATE DIAGRAMS

L.1 Predefined partial state diagrams

There are three predefined partial state diagrams shown in figure L.1. The upper-left one shows how a sender writes the message header to the mailbox at the MPC. Before the actual writing, the sender must check that the mailbox is free, ie. the previous one has been processed by the MPC. A simple status bit in the mailbox serves this purpose. Whenever the MPC has processed a message, the signal /MPCRDY is asserted to awake all waiting processes at all processors. They should accordingly check the status bit of their mailbox. Now senders only need to poll the local status register (PNSR) once they find that the MPC has not processed their previous messages. This simple technique can prevent processes from using the system bus extensively for busy-waiting.

The upper right one in figure L.1 illustrates a similar situation when a receiver wants to read from its mailbox. Note that only when the non-blocking option was used for the previous Receive (Send) would the Receiver (Sender) possibly find the mailbox was not ready.

The lower one in the figure depicts how a the sender polls the local status register (PNSR) in order to capture the acknowledgement from the MPC. Since there is only one /MPCRDY signal but there may be many processes waiting, the sender must check the status bit again and find out whether that assertion of the /MPCRDY really benefits it. This argument also applies to the previous two cases.

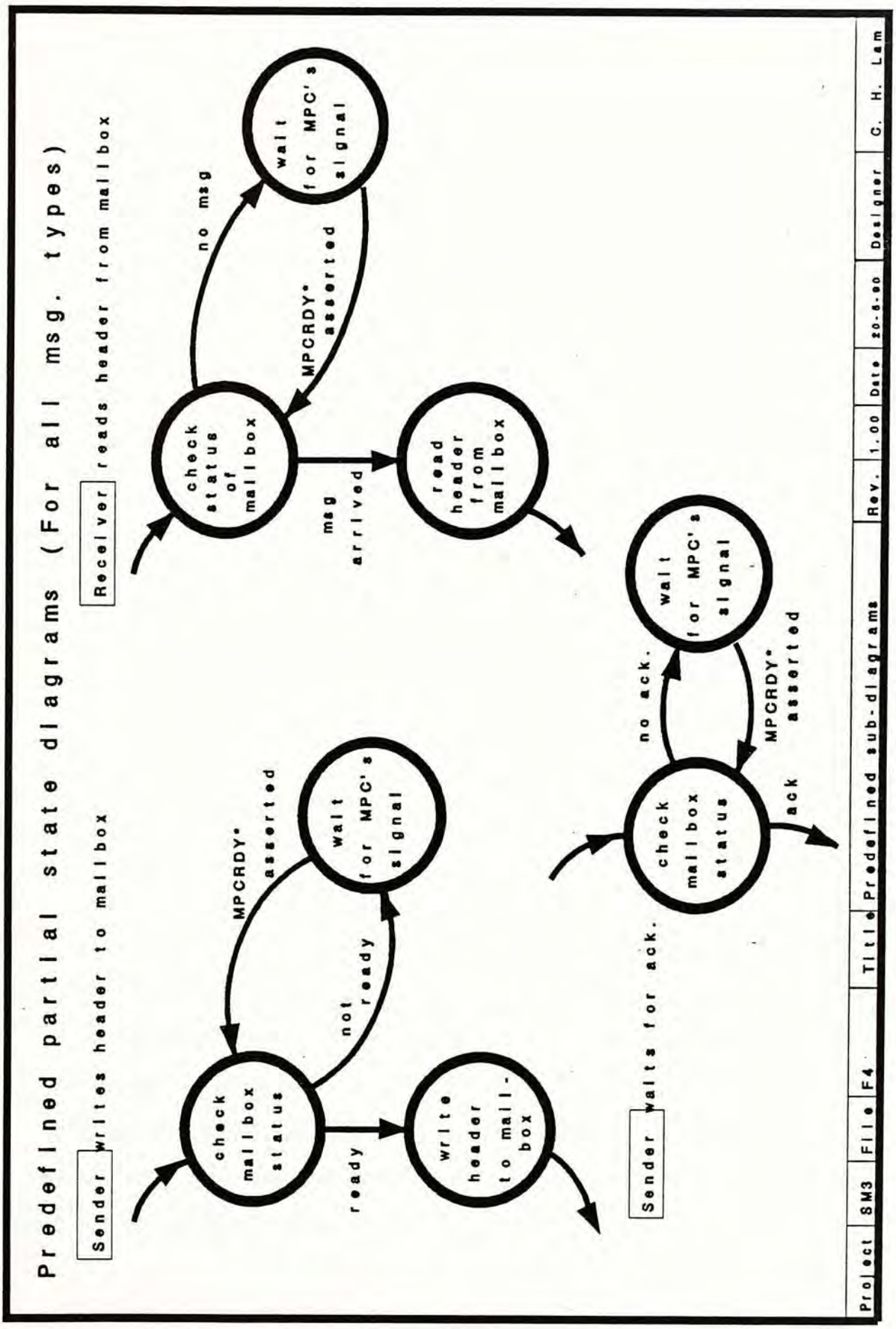


Figure L.1 Predefined partial state diagrams.

L.2 Point-to-point messages

Figure L.2 includes the state diagrams for the three parties involved in a point-to-point communication. The sending PN is initially running the user process. The SendMessage primitive writes the header to the mailbox when a message is to be sent. The user process is resumed if it is a non-blocking Send. Otherwise, the sender must capture the acknowledgement from the MPC.

The MPC is normally at the busy-waiting state. After routing a newly arrived non-blocking message to a suitable IMQ, the job of the MPC is over. For a blocking message, the MPC must acknowledge the sender once the receiver has accepted the header (not including the body for a long message). Of course, the MPC has to service other requests while it is waiting for the response from the receiver.

The receiver distinguishes between long and short messages (cf. blocking and non-blocking messages for a sender). For a long message, a DMA brings the message body from the sender.

L.3 Broadcast messages

State diagram for the sender of a broadcast message is shown in figure L.3. The sender (and also the MPC) must distinguish between long and short, blocking and non-blocking messages. For short messages, the sender behaves exactly like the case for point-to-point messages. For a long message, the sender uses 1-to-N DMA to broadcast the message via the VMEbus. The assertion of /BCST and /BCEND marks the beginning and ending of a transfer. After the 1-to-N DMA, the SendMessage primitive determines whether it should return to the user process or wait for an acknowledgement.

The procedure of the MPC is identical to the case of point-to-point messages if the broadcast message is short. Otherwise, the MPC halts checks the availability

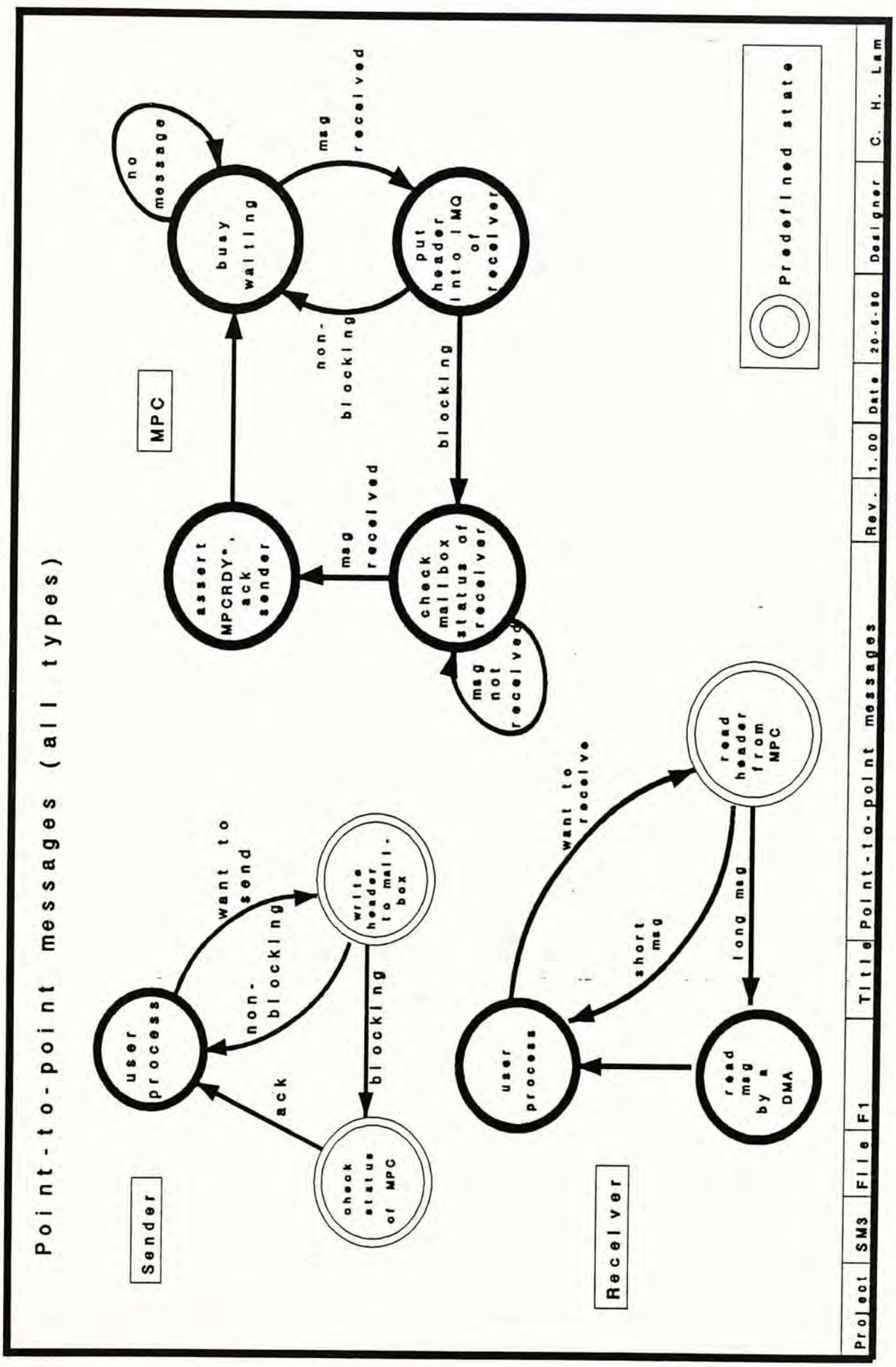


Figure L.2 State diagram for point-to-point messages.

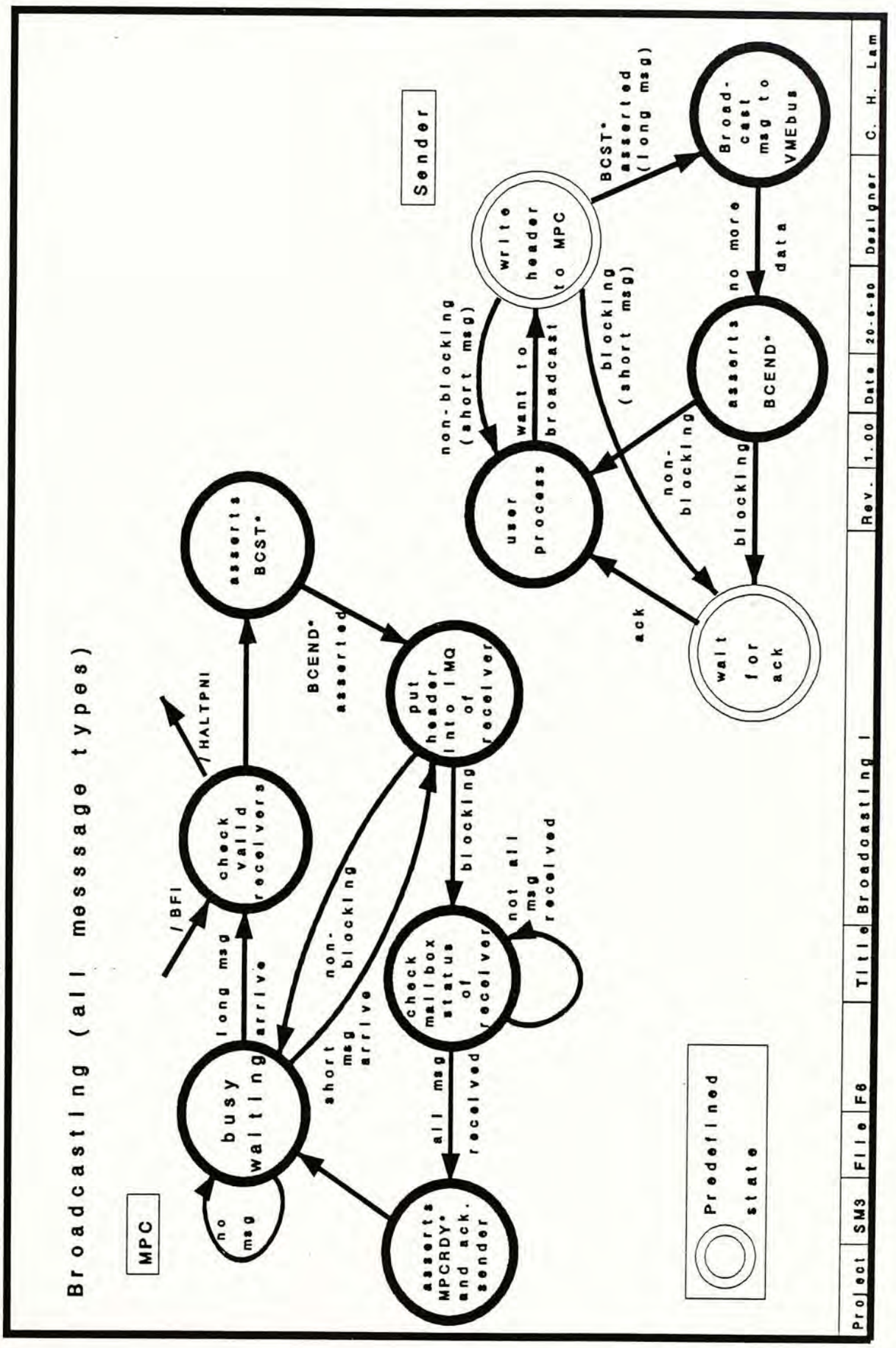


Figure L.3 State diagrams (sender and MPC) for broadcasting.

of receive buffers and halts potential receiving PNs. Then it initiates the 1-to-N DMA by asserting /BCST. It knows the transfer is over when /BCEND is asserted by the sender. After that, the MPC can treat the message like a short one.

The state diagram for the receiver can be found in figure L.4. The left side one highlights that for a long message, the receiver must read the message body form the circular buffer and update the pointer accordingly. Note that if the receiver is running ahead of the sender, then it will be blocked (for a blocking Receive) at the state Read-Header-From-Buffer. The right side one shows how the 1-to-N DMA affects the receiver. The details can be found in the core of this thesis so it is not explained here.

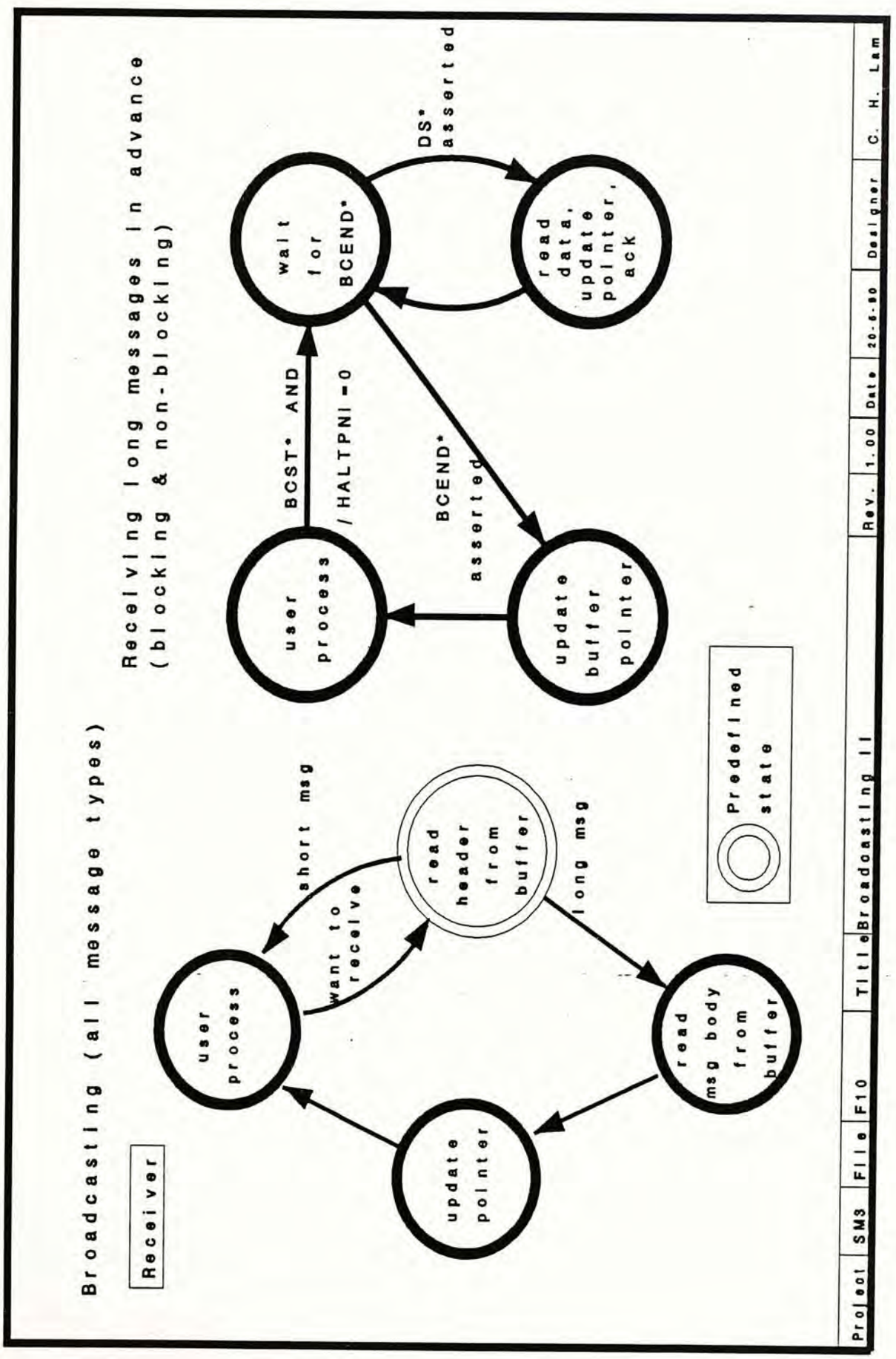


Figure L.4 Receiver state diagram for broadcasting.

APPENDIX M
BOOT-UP PROCEDURE OF SM3

Figure M.1 illustrates the proposed boot-up procedure for the current implementation of SM3. When the system is powered up, the host machine executes its original boot-up procedure independently. The hardware logic on the PNs and the MPC will hold the /RESET line for at least 512 cycles as specified in the data sheet. Before this period expires, hardware logic on the MPC initiates the HALTR (refer to APPENDIX F) to assert the /HALTPNn line. All PNs and the MPC are halted in this way so they cannot proceed even after the reset period has expired.

Once the host machine is up, it initializes the reset and interrupt vector tables of the PNs and the MPC. Programs are then load onto all processor boards. At last, the host can clear the HALTR directly and release the PNs and the MPC. Now, they can start their cold reset routines, which are the user programs. Termination of execution can be achieved by writing to the HALTR directly.

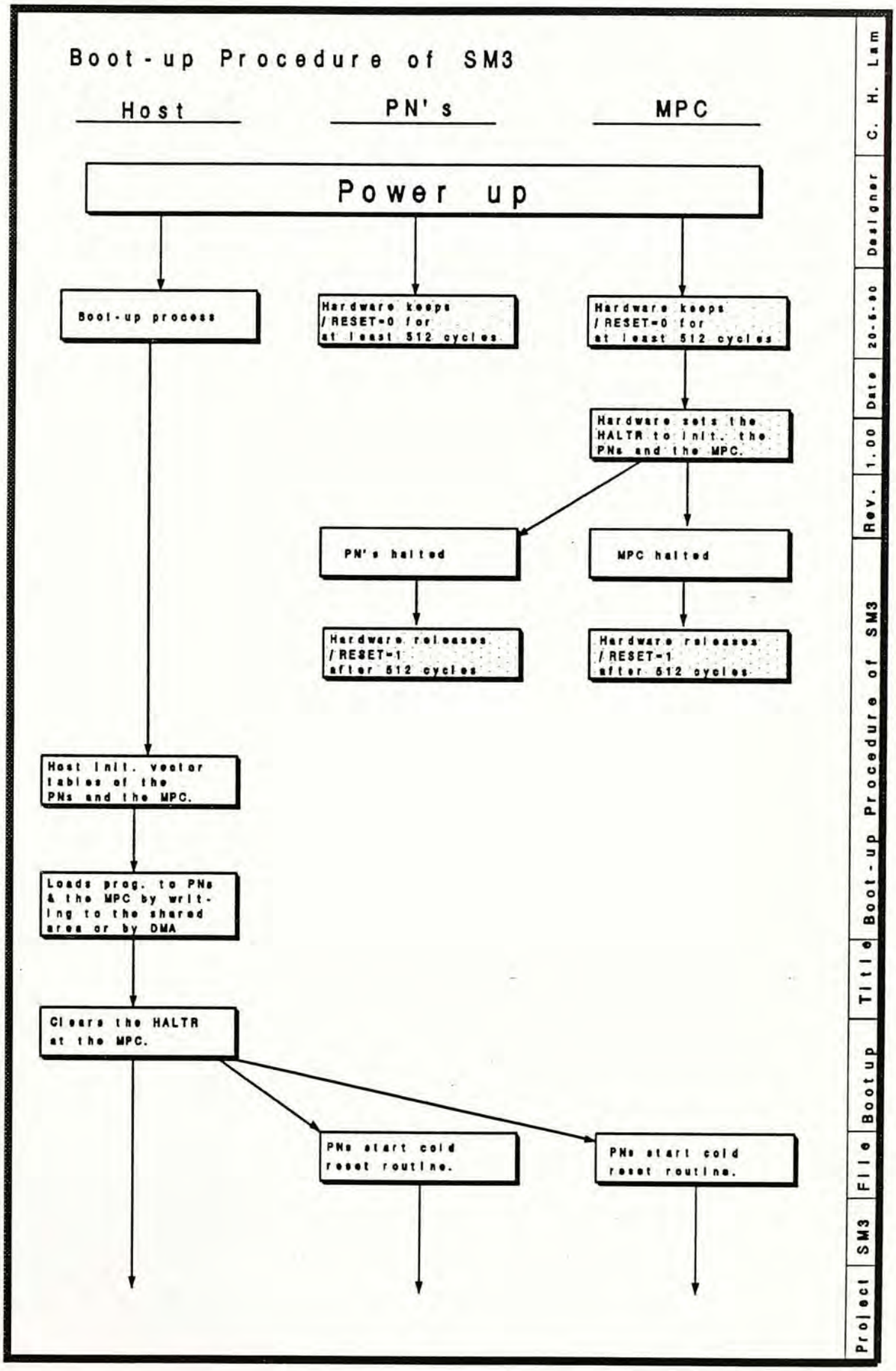


Figure M.1 Boot-up procedure of SM3.

PUBLICATIONS

1. *"Message-passing Controller for a Shared-memory Multi-processor," Computer Architecture News, Vol.17, No.6, Dec. 1989, pp.142-149.*

2. *"The Design of a Hardware Supported Message-based Multi-microprocessor Workstation," Proceedings of the 13-rd Australian Computer Science Conference, 1990.*

REFERENCES

[Akl89] Selim G. Akl, "*The Design and analysis of parallel algorithms*," Prentice-Hall international, Inc., 1989.

[Cleme88a] Alan Clements, "*Multiprocessor systems (1 of 4)*," Electronics and Wireless World, June 1988, VOl.94, no. 1628, pp.534-536.

[Cleme88b] Alan Clements, "*Multiprocessor systems (2 of 4)*," Electronics and Wireless World, July 1988, Vol.94, no. 1629, pp.703-706.

[Cleme88c] Alan Clements, "*Multiprocessor systems (3 of 4)*," Electronics and Wireless World, Sept. 1988, Vol.94, no. 1631, pp.875-881.

[Allis88] Andrew Allison, "*Where there's RISC, there's opportunity*," Mini-micro systems, January 1988, pp.49-62.

[Ander75] L. H. Anderson, "*The Microcomputer as Distributed Intelligence*," Proceedings of the International Symposium on Circuits and Systems, Boston, Mass., April 1975, pp.337-340.

[AnnJa85] Annot, J. K., and Janssens, M. D., "*Multiprocessor Unix*," Master's thesis, Department of Electrical Engineering, Delft University of Technology, The Netherlands, 1985.

[AthSi88] William C. Athas and Charles L. Seitz, "*Multicomputers: Message-passing Concurrent Computers*," IEEE Computer, August 1988, pp.9-24.

[BacBu84] Bach, M. J. and Buroff, S. J., "*Multiprocessor Unix operating systems*," Bell Systems Technology Journal 63, 8 (Oct. 1984), pp.1733-1749.

[Baude77] Gerard M. Baudet, "*Iterative Methods for Asynchronous Multiprocessors*," High Speed Computer Architecture and Algorithm Organization, 1977, pp.309-310.

[BauSe75] A. Baum and D. Senzig, "*Hardware Considerations in a Microcomputer Multiprocessing System*," Digest of Papers Compcon Spring 75, San Francisco, Calif., Feb. 1975, pp.27-30.

[Beims84] Bob Beims, "*Multiprocessing capabilities of the MC68020 32-bit Microprocessor*," User document AR220, reprinted from WESCON, 1984, pp.1-16.

[BodLi89] B. L. Bodnar and A. C. Liu, "*Modelling and Performance Analysis of Single-bus Tightly-coupled Multiprocessors*," IEEE Transactions on Computers, vol. 38, no. 3, March 1989, pp.464-467.

[BraGr89] J. H. Brand and L. de Graaf, "*Message protocols unloads VMEbus processors*," EDN magazine, November 9, 1989, pp.255-258.

[BruMi84] Bruce Hamilton and Mike Fischer, "*A high performance workstation using a closely coupled architecture*," Digest of papers Compcon 84, spring 1984, pp.207-209.

[BuCoD89] F. J. Burkowski, G. V. Cormack and G. D. P. Dueck, "*Architectural Support for Synchronous Task Communication*," Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems 1989, ACM, pp.40-53.

[Bybee89] Robert Bybee, "*Customized Buses Build on Industry Standards to Enhance Performance*," Computer Design, February 1, 1989, pp.109-112.

[CheKa88] D. J. Chen and K. M. Kavi, "*A Qualitative Assessment of Object-oriented Architectures: SWARD, INTEL 432 and IBM S/38*," Proceedings of International Computer Symposium 1988, December 15-17, pp.175-181.

[Cheva73] R. J. Chevance, "*A COBOL Machine*," Proceedings of the ACM SIGPLAN-SIGMicro Interface Meeting, N.Y. : ACM, 1973, pp.139-144.

[DanPe85] Daniel D. Gajski and Jih-Kwon Peir, "*Essential issues in multiprocessor systems*," IEEE computer, June 1985, pp.9-27.

[DelRe89] Sergio A. Delgado-Rannauro and T. J. Reynolds, "*A Message Driven OR-parallel Machine*," Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems 1989, ACM, pp.217-226.

[EwHoF86] P. A. Ewens, R. C. Holt, M. J. Funkenhauser, and D. R. Blythe, "*The Tunis report : design of a Unix-compatible operating system*," technical report CSRI-176, January 1986, Computer systems research institute, University of Toronto.

[FatKr83] Eli T. Fathi and Moshe Krieger, "*Multiple microprocessor systems: what, why, and when*," IEEE computer, March 1983, pp.23-32.

[Feng81] Tse-yun Feng, "*A survey of interconnection networks*," IEEE tutorial on supercomputers: design and application, 1981. pp.109-124.

[FiJoS83] M. Firlmeier, G. Joubert, and U. Schendel, "*A new type of parallel computer using microprocessors*," Parallel computing 1983, North-Holland, pp.527-532.

[FinHe88] Raphael Finkel and Debra Hensgen, "*YACKOS on a Shared-memory Multiprocessor*," Computer Architecture News, September 1988, pp.31-36.

[Fisch84] Wayne Fischer, "*The VMEbus Project*," Compcon, spring 1984, pp.376-378.

[FrHeH89] G. Fritsch, W. Henning, H. Hessenauer, R. Klar, C. U. Linster, C. W. Oehlrich, P. Schlenk and J. Volkert, "*Distributed Shared Memory Multiprocessor Architecture MEMSY for High Performance Parallel Computation*," Computer architecture news, Dec. 1989, pp.22-35.

[GajPe85] Daniel D. Gajski, Jih-Kwon Peir, "*Essential Issues in Multiprocessor Systems*," IEEE Computer, June 1985, pp.115-133.

[GeJoS82] Gehringer, E. F., Jones, A. K., and Segal, Z. Z., "*The Cm* testbed*," Computer, October 1982, pp.40-53.

[Gentl81] W. Morven Gentleman, "*Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept*," Software - Practice and Experience, Vol.11, 1981, pp.435-466.

[GeRiM68] George H. Barnes, Richard M. Brown, Maso Kalo, D. Kuck, D. Slotnick, and R. Stokes, "*The Illiac IV computer*," IEEE Transaction on Computers, C-17, vol.8, pp.746-757, Aug. 1968.

[Giloi87] Wolfgang K. Giloi, "*SUPRENUM - A Trendsetter in Modern Supercomputer Development*," Proc. of the 2nd Symposium on Vector and Parallel Processors for Scientific Computation, Sept. 21-23, 1987, Rome, pp.29-45.

[Halst87] Robert H. Halstead, Jr., "*Overview of Concert Multilisp: a Multiprocessor Symbolic Computing System*," Computer Architecture News, 1987, vol. 15, no. 1, pp.5-11.

[Harma89] Thomas L. Harman, "*The Motorola MC68020 and MC68030 Microprocessors: Assembly Language, Interfacing, and Design*," Prentice-Hall International Editions, 1989.

[Heard84] Heard, G. H., "*GaGs - A material that promises high speed for supercomputers*," SUPERNET - Technology Development of California (TDC), Santa Clara, California, 1984.

[HeMaN88] R. D. Hersch, F. Maddaleno, C. Nicks et al, "*The video-RAM multiprocessor architecture*," Microprocessing and microprogramming 24(1988), pp.503-510.

[Holt82] R. C. Holt, "*An overview of Tunis : a Unix look-alike written in Concurrent Euclid*," technical report CSRG-140, April 1982, Computer systems research institute, University of Toronto.

[HwaFa85] Kai Hwang and Faye A. Briggs, "*Computer architecture and parallel processing*," McGraw-Hill Inc., 1985.

[IEEE83] IEEE, "*Local Area Network - Token Ring Access Method*," IEEE 802.5, 1983.

[Inmos83] INMOS, "*OCCAM programming manual," INMOS limited, 1983.*

[JaAnV86] M. D. Janssens, J. K. Annot, and A. J. Van De Goor, "*Adapting UNIX for a multiprocessor environment*," Communications of the ACM, September 1986, Vol.29, no.9, pp.895-901.

[JaBaP88] P. Jaulent, L. Baticle and P. Pillot, "*68020, 68030 Microprocessors and their Coprocessors*," Macmillan Education press, 1988.

[Jeffr84] Jeffrey Schriebman, "*Unix portability*," Digest of papers Compcon 84, spring 1984, pp.499-501.

[JonSc79] Jones, A. K., and Gehringer, E. F. (Editor), "*Cm* Multiprocessor Project: A Research Review*," Tech. Rept. CMU-CS-80-131, Carnegie-Mellon Univ., July 1980.

[Kirrm89] Hubert Kirrmann, "*Multiprocessors and Supercomputer research in Europe*," IEEE micro, February 1989, pp.7-8.

[KuSiP82] James T. Kuehn, Howard J. Siegel, and Peter D. H., "*Design and Simulation of an MC68000-based Multi-microprocessor System*," Proceedings of IEEE International Conference on Parallel Processing 1982, pp.353-362.

[Labib88] G. A. M. Labib, "*Multiprocessor systems*," Electronics and Wireless World, January 1988, Vol.94, no.1623, pp.43-44.

[Landa61] Landauer, R., "*Irreversibility and heat generation in the computing process*," IBM Journal of Research and Development, Vol.5,

pp.183-191, July 1961.

[Lee77] Ruby Bei-Loh Lee, "*Performance Bounds in Parallel Processor Organizations*," High Speed Computer Architecture and Algorithm Organization, 1977, pp.453-455.

[March77] P. Marcham, "*Data Transmission via PABXs*," The NCC publications, 1977.

[Marko65] Marko, H., "*Physikalische und biologische Grenzen der Informationsuebermittlung*," Kybernetik, Vol.2, pp.274-284, Oct. 1965.

[MetRo76] Metcalf, R, M., Roggs, D. R., "*Ethernet: Distributed Packet Switching for Local Computer Networks*," Commun. ACM, 1976, pp.394-404.

[MeyHa75] de Brito Meyer, W. and Hawley, J. A., III., "*Munix, a multiprocessor version of Unix*," Master's thesis, Naval Postgraduate School, Monterey, Calif., 1975.

[Milut85] V. M. Milutinovic, Editor, "*Advanced Microprocessors and High-Level Language Computer Architecture*," Washington, D.C., IEEE press Tutorial, 1985.

[MuLiS86] H. Muehlenbein, F. Limburger, S. Streitz, and S. Warhaut, "*MUPPET - A Programming Environment for Message-based Multiprocessors*," Proceedings of the ACM/IEEE joint conference, 1986, pp.336-343.

[Motor84] Motorola Inc., "*VMEbus Specification Summary*," 16/32-bit Microcomputer System Components, Motorola Inc., 1984.

[Motor88a] Motorola Inc., "*MC68000 Family Reference*," 1988.

[Motor88b] Motorola Inc., "*MEME147BUG: 147Bug Debugging Package User's Manual*," User document MVME147BUG/D1, 1988.

[Motor89] Motorola Inc., "*MVME147S: MPU VMEmodule User's Manual*," User document MVME147S/D1, 1989.

[Nader88a] M. Naderi, "*Modelling and Performance Evaluation of Multi-processors organization with shared memories*," Computer Architecture News, September 1988, pp.51-74.

[Nader88b] M. Nader, "*Modelling and Performance Evaluation of Multi-processor Organization with Multi-Memory Units*," Computer

Architecture News, December 1988, pp.35-51.

[Ng86] K. W. Ng, "*Message-passing Primitives for Multi-microprocessor Systems*," Microprocessors and Microsystems, vol. 10, no. 3, April 1986, pp.156-160.

[Olson85] Robert Olson, "*Parallel Processing in a Message-Based Operating System*," IEEE software, July 1985, pp.39-49.

[Paker83] Y. Paker, "*Multi-microprocessor Systems*," Academic Press, 1983.

[PapPi88] M. P. Papazoglou and P. E. Pintelas, "*A versatile kernel proposal for a multi-microprocessor system environment*," Microprocessing and microprogramming, Vol.22, No.1, January 1988, pp.11-20.

[Pete88] Pete Wilson, "*The CPU wars*," Byte, May 1988, pp.213-234.

[Pount88] Dick Pountain, "*Equus: A Parallel Operating System*," Byte, September 1989, pp.3-8.

[Quinn87] Michael J. Quinn, "*Designing Efficient Algorithms for Parallel Computers*," McGraw-Hill, 1987, Chapter 4.

[RetTh86] Randall Rettberg and Robert Thomas, "*Contention is No Obstacle to Shared-Memory Multiprocessing*," Communications of the ACM, Vol.29, No.12, Dec. 1986, pp.1202-1212.

[Rober81] Robert Bernhard, "*More hardware means less software*," IEEE spectrum, December 1981, pp.30-37.

[RudPe87] Rudy Lauwereins and J.A. Peperstraete, "*An integrated software-hardware multiprocessor project*," Proceeding of the 1987 international conference on parallel processing, pp.618-620.

[Russo77] Paul M. Russo, "*Interprocessor Communication for Multi-Micro-computer Systems*," IEEE Computer, April 1977, pp.67-76.

[Sangu86] John Sanguinetti, "*Performance of a Message-Based Multiprocessor*," IEEE computer, Sept. 1986, pp.47-55.

[SchSo90] Bernd Schwister and Karl Solchenbach, "*SUPRENUM - A European Made Supercomputer*," Future Generation Computer Systems 5 (1989/1990) pp.381-385.

[ScoWa84] P. R. D. Scott, J. B. Waites et al, "*Introducing Computerized Telephone Switch Boards (PABXs)*," The NCC Publications, 1984.

[ShMiS78] Shan S. Kuo, Michael H. Linck and S. Saadat, "*A guide to communicating sequential processes*," Technical monograph PRG-14, August 1978, Oxford University computing laboratory, Programming research group.

[SiMiM86] A. Silbey, V. Milutinovic and V. Mendoza-Grado, "*A Survey of Advanced Microprocessors and High-Level Language Computer Architecture*," in "Tutorials on Advanced Microprocessors and High-Level Language Computer Architectures," IEEE press, 1986, pp.118-141.

[Tabak90] Daniel Tabak, "*Multiprocessors*," Prentice-Hall series in computer engineering, Prentice-Hall International, Inc., 1990.

[TasPl89] L. Tassakos and K. W. Plessmann, "*pdvPOOL: A Real-Time Object-Oriented Multiprocessor Systems*," Microprocessing and Microprogramming 25 (1989), pp.221-228.

[ThGiF88] Shreekant Thakkar, Paul Gifford, and Gary Fielland, "*The Balance Multiprocessor System*," IEEE micro, Feb. 1988, pp.57-69.

[Tom88] Tom Williams, "*Software machine model blazes trail for parallel processing*", Computer design, October 1, 1988, pp.20-26.

[VaMaB88] F. A. Vaughan, C. D. Marlin and C. J. Barter, "*A Distributed Operating System Kernel for a Closely-Coupled Multiprocessor*," The Australian Computer Journal, Vol.20, No.2, July 1988.

[Ware72] W. H. Ware, "*The ultimate computer*," IEEE spectrum, March 1972, pp.84-91.

[Weidn86] P. Weidner, "*MIMD Algorithms and Their Implementation*," Proceedings of Workshop of Parallel Processing: Logic, Organization, and Technology, July 1986, pp.75-86.

[WiLoE87] Wim J. H. J Bronnenberg, Loek Nijman, Eddy A. M. Odijk, and Rob A. H. van Twist, "*DOOM : a decentralized object-oriented machine*," IEEE micro, October 1987, pp.52-69.

[Winog65] Winograd S., "*On the time required to perform addition*," J. Association of Computing Machineries, Vol.12, pp.277-285, April 1965.

[Winog67] Winograd S., "*On the time required to perform multiplication*," Journal of Association of Computing Machineries, Vol.14, pp.793-802, Oct. 1967.

[YanWe89] Zheng Yanheng and Zhong Wen, "*High Performance Algorithms for Solving Equation Systems on Loosely Coupled Multiple Microcomputer Systems*," Proceedings of International Symposium on Computer Architecture and Digital Signal Processing 1989, pp.316-319.

[ZehUn89] E. Zehendner and Th. Unqerer, "*A Simulation Method for Parallel Computer Architectures*," Microprocessing and Microprogramming 25 (1989), pp.209-212.

[Zhang88] Xiaodong Zhang, "*Experiments and Analysis of the Various Effects on Shared-memory Multiprocessor Performance*," internal report, Computer Science Department, University of Colorado at Boulder, Colorado 80309, USA, 1988.