NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

The Homogeneous Multiprocessor: Memory Modules and the Interbus Switch Controller

Terry L. Segal

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering at
Concordia University
Montréal, Québec, Canada

February 1989

ISBN 0-315-53037-5

ABSTRACT

The Homogeneous Multiprocessor:
Memory Modules and the Interbus Switch Controller

Terry L. Segal

The Homogeneous Multiprocessor is a tightly-coupled, MIMD architecture consisting of a number of identical processing elements, and two types of interconnection pathways: (1) a distributively controlled network of switches ("extended buses") which permits each processor to access the memory of its two immediate neighbours, and (2) a fast local area network (H-Network) which permits each processor to communicate with all the other processors in a point-to-point or broadcast mode.

The design and implementation of two memory systems is given, one being a dynamic design incorporating error detection and correction to boost the reliability of the system and the other a static design using the newest 32 Kilobyte static RAMs. These two memory modules are prototypes that will be used in the Homogeneous Multiprocessor. Depending on cost, efficiency, speed of operation and area required on the circuit boards, one may be a better solution than the other. The Interbus Switch Controller has been designed and tested using standard *off the shelf* integrated circuits. The Interbus Switch Controller is a network of switches that are controlled in a distributive fashion. The Interbus Switch Controller is capable of creating the extended bus to allow a processor to access the memory modules of its two immediate neighbours.

ACKNOWLEDGEMENTS

I would like to thank my parents Carole and Martin and my fiancee, Kelly for inspiring and supporting me during the endless hours of work.

I wish to express my sincerest gratitude to my thesis supervisor, Dr. N. J. Dimopoulos, for his caring guidance throughout my undergraduate and graduate studies. He was always available to direct and to advise whenever the occasion arose.

I would like to thank all the people who have been associated with this project, especially Kin Li, M. Robillard and G. Gosselin for their assistance in making the hardware operative.

Table of Contents

List of Figures

List of Tables

List of Symbols

FE	Front End	17
HS_i	Network station i	17
$IBS_{i,j}$	Interbus Switch Controller located between processor i and processor j	17
$IBSW_{i,j}$	Interbus Switch located between processor i and processor j	17
$LC_{i,j}$	Logical Closure for Interbus Switch located between processor i and processor j	25
$\overline{LDS}$	Lower Data Strobe	33
LE IN	Latch Enable Input	75
LE OUT	Latch Enable Output	76
m	# parity bits	66
M_i	Memory Module i	17
MMU	Memory Management Unit	5
MS	Mass Storage	17
$\overline{MULT\ ERROR}$	Multiple bit error	70
n	# message bits	66
N_i	Message bit i	67
$\overline{OE}$	Output Enable	41
$\overline{OE}_i$	Output Enable for Processor i	51
$\overline{OE}_{A,C,D}$	Output Enable for Address, Control and Data lines	43
$\overline{OE}_{A,C,D}IBS_{i,j}$	Output Enable for Address, Control and Data lines on Interbus Switch between processor i and processor j	25
$\overline{OE}_{A,C,D}PS_i$	Output Enable for Address, Control and Data lines on Processor Switch i	25
P_i	Processor i	17
$\overline{PAS}$	Physical Address Strobe	38
$\overline{PAS}_i$	Physical Address Strobe originating from Processor i	25
$\overline{PBGACK}$	Physical Bus Grant Acknowledge	38

T	Buffer Direction control	41
$T_{A,C}$	Directional Control for Address and Control lines	41
T_D	Directional Control for Data Lines	43
$T_{A,C}IBS_{I,J}$	Directional Control for Address and Control lines on Interbus Switch between processor i and processor j	25
$T_DIBS_{I,J}$	Directional Control for Data lines on Interbus Switch between processor i and processor j	25
$T_{A,C}PS_I$	Directional Control for Address and Control lines on Processor Switch i	25
T_DPS_I	Directional Control for Data lines on Processor Switch i	25
TE	Terminal	17
$\overline{UDS}$	Upper Data Strobe	33
$\overline{WE}$	Write Enable	74
$\overline{Y}_I$	Decoder Output i	93

List of Appendices

1. Introduction

In the past decade, the development of microprocessor technology and electronic circuits along with the desire to have larger and faster computers have prompted an increase in the research activities in the area of computer architecture. Computer designers have proposed numerous distributed and multiprocessing architectures. These systems have used mostly *off the shelf* commercially available components and more recently, implementing specific architectures or specialized components, Very Large Scale Integration (VLSI).

The definition of a multiprocessor is a computer that contains two or more processor units working on a shared memory under integrated control. However, the processor units can have their own small local memories. There must exist an integrated operating system which controls all of the hardware and software of the system. There are many ways of organizing the interconnection network required by the multiprocessors such as: a timed-shared bus; a crossbar switch; an n-dimensional cube; nearest neighbour matrices; and a cluster bus to name a few.[12]

As defined in the past, a multiprocessing system consists of two or more processors each of which can operate independently but at the same time can exchange information via an interconnection network and/or shared memory. In the Homogeneous Multiprocessor Proper an interconnection network known as the InterProcessor Bus allows processors to access the memories of their two immediate neighbours.

Some of the more popular multiprocessing systems that have been proposed and built that contain varying degrees of coupling and homogeneity are the Butterfly [2], the Cm* [19], the C.mmp [21], the Cosmic Cube [18], the NYU Ultracomputer [7], etc.

1.1 Classification of Architectures

There are four basic classifications for a computer. The four categories determined by the multiplicity of instruction and data streams. These classifications were introduced by Michael J. Flynn.

SISD : Single Instruction stream - Single Data stream

SIMD : Single Instruction stream - Multiple Data stream

MISD : Multiple Instruction stream - Single Data stream

MIMD : Multiple Instruction stream - Multiple Data stream

Most commercially available computers available today are SISD, they contain one processor, control unit, and memory.

Array processors are SIMD. They contain multiple processing elements and one control unit. Each of the processing elements receives the same instructions to execute on different data.

MISD computers contain n processing units. Each processing unit receives different instructions to execute on the same data. The output of processor i is used as the input to processor j.

An MIMD computer consists of n processors that interact on an input derived from the same data space shared by all processors. The computer is said to be *tightly-coupled* if the degree of interaction among the processors is high.[10] In an MIMD system, care must be taken to avoid unecessary overhead due to interference caused by accessing shared resources such as main memory and synchronization.[6]

1.2 Applications for Parallel Processors

Just about any application can be accomplished on a parallel computer. The justification for using one is a significant decrease in the time required to complete the job. A most obvious example is weather forcasting. There is no point forcasting the weather using a computer if the time required for the computations is greater than the forcast time less the current time, i.e. if it is 9:00 AM, the weather being forcast is for 10:00 AM and the computer requires two hours to do the forcast. Reduction in computation time is a major justification, but it is not the only one. In some instances, the sheer complexity of the problem may be a factor.

There are various categories where parallel computers can play a major role.

Weather forcasting: Solving the general circulation model equations.

Oceanography and Astrophysics: Ocean climate, fishery management, ocean resources, and tides.

Artificial Intelligence and Automation: Image processing, pattern recognition, computer vision, speech understanding, machine interface, CAD/CAM/CAI/OA, intelligent robotics, expert systems, and knowledge engineering.

Finite element analysis: Design of dams, bridges, ships, jets, buildings, and space vehicles. I.E. large systems of algebraic an partial differential equations.

Computational dynamics: Aircraft and spacecraft lift, turbulence studies.

A host of other applications are also applicable: remote sensing, energy resources exploration, seismic exploration, reservoir modeling, plasma fusion power, nuclear reactor safety, computer assisted tomography, genetic engineering, weapon research and defense.[10]

1.3 Parallel Processor Structures

Parallel computers are divided into three configurations:

Pipeline computers

Array processors

Multiprocessor systems.

There are four major steps involved in normal processing within a Pipelined processor. They are:

Instruction Fetch (IF): From main memory

Instruction Decoding (ID): Identify the operation

Operand Fetch (OF): If required for execution

Execution (EX): Execute the operation

In a non-pipelined processor, once an instruction fetch is complete, the next one cannot proceed until the current instruction has run to completion, i.e. instruction decode, operand fetch, and execution. In a pipelined processor, at any given time, there can be as many as four (or as many stages as there are in the *pipe*) instructions *running* at different stages of completion. The rate at which instructions are *clocked* through the pipe is equal to the slowest stage in the pipe. In theory, a pipelined processors with n stages can be at most n times faster than a non-pipelined processor.

Array processors are synchronous parallel computers comprising of processing elements (PE). A processing element contains an ALU with registers and local memory. The processing elements are connected by a data-routing network. The processing elements are synchronized to perform the same function at the same time, and of course the data-routing mechanism must be established among them. There is also a control unit which fetches and decodes instructions from local and control memory and executes the scalar and control-type instructions. The data routing is accomplished by the control unit. Vector instructions are received by the processing elements from the control unit

for distributed execution for use with different operands fetched directly from the PE's local memory

A multiprocessor contains two or more processors that have comparable capabilities. Memory modules, I/O channels, and peripheral devices are shared between all of the processors. Each processor also has its own local memory and peripheral devices. A multiprocessor is controlled by a single integrated operating system that provides interaction between processors and their programs at various levels. The processors, memories, and peripheral devices are organized by the interconnection structure. Some of these interconnection structures are a time-shared common bus, a crossbar switch network, and multiport memories.[10]

1.4 Existing Multiprocessors

Currently, there are many multiprocessors that have been designed and built. Some of them are research projects while others are commercially available. In the following sections, descriptions of some of the multiprocessors that have been conceived through University research are given. These multiprocessors are: The Butterfly; the Cm*; the C.mmp; the NYU Ultracomputer; and the Homogeneous Multiprocessor.

1.4.1 The Butterfly

The Butterfly Multiprocessor consists of nodes each of which contains a processor and local memory and is connected to a high performance switch. Each node comprises of a Motorola MC68000 with up to 4 Megabytes of local memory, an Advanced Micro Devices AMD2901 processor which implements the Processor Node Controller (PNC) function, and a Memory Management Unit (MMU). Shown in figure 1.1 is the block diagram of the node.[3]

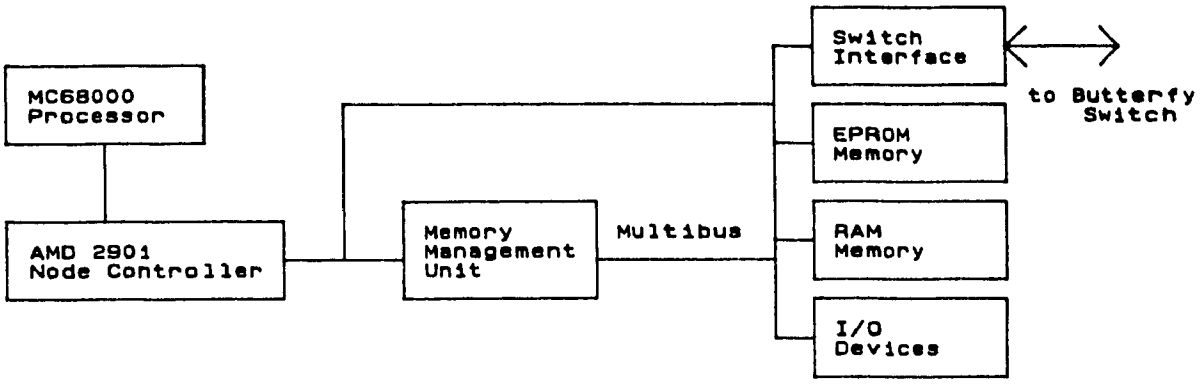


Figure 1.1: The Block Diagram of the Node.[3]

The Butterfly can contain up to 256 nodes. Through a virtual memory interface access to the Butterfly switch is accomplished. This is done by the PNC, it interprets the virtual address to determine if the request is local or on a remote node. If the request is local, the virtual address is passed to the Memory Management Unit for translation into a physical memory location. If the request is destined for a remote node, the PNC sends a message to through the Butterfly switch to the remote node, where the reference is satisfied and if necessary, a reply is returned. The message contains a physical memory address and a node address for the remote node.

The backbone of the processor is the Butterfly Switch. Shown in figure 1.2 is the building block of the Butterfly Switch, a 4 x 4 switch.

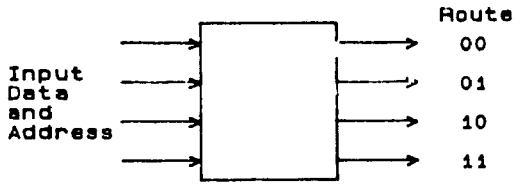


Figure 1.2: A 4 x 4 Butterfly Switch.[3]

When accessing the Butterfly Switch, a remote address and data arrive on an input path, and depending on the address, exit the appropriate path. Refer to figure 1.3, for a 16-node Butterfly processor.

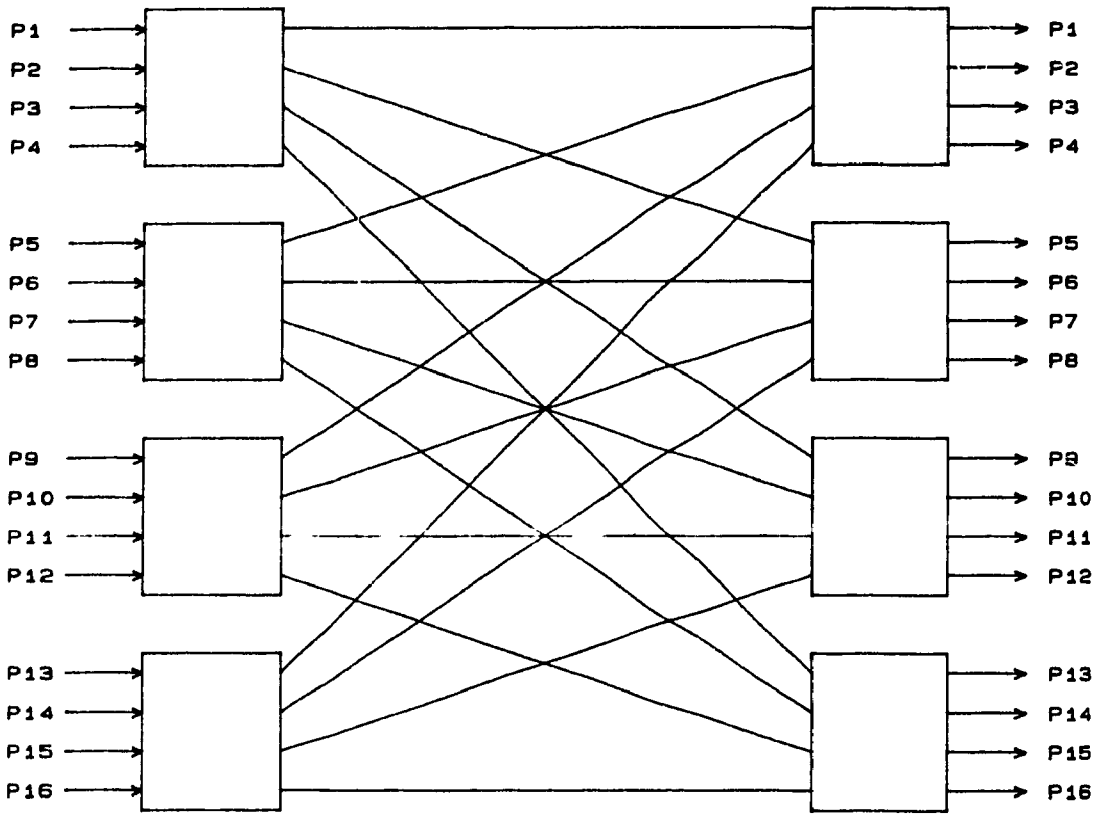


Figure 1.3: A 16-Node Butterfly Processor.[3]

1.4.2 The Cm*

The Cm* is an MIMD distributed multiprocessor. It was developed at Carnegie Mellon University in the mid 1970's. All of the processors in the Cm* share a single virtual address space, i.e. each processor has associated with it, a portion of the total memory. Processors are grouped into clusters and processors in different clusters communicate via packet-switches. The hierarchy of the Cm* consists of three levels: modules (processors); clusters (groups of modules); and system.

A Module comprises of a processor, a local memory, and peripheral devices. The contents of the Module communicate with each other via the Modules' processor bus. The processor within the Modules is a DEC LSI-11. The LSI-11 is a 16-bit computer with an address range of 64 Kilobytes. Shown in figure 1.4 is the block diagram of a Module in the Cm*. Each of the modules is connected to the other Modules in the Cluster via the local switch.[3]

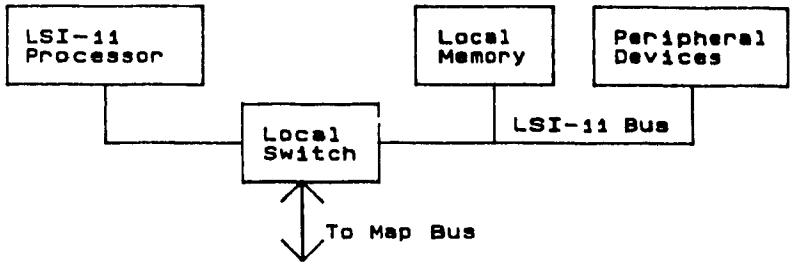


Figure 1.4: The Block Diagram of a Module in the Cm*.[3]

A Cluster is a group of 14 Modules or Cms. The Clusters are connected via the Intercluster buses. Referring to figure 1.5, a cluster of computer modules.

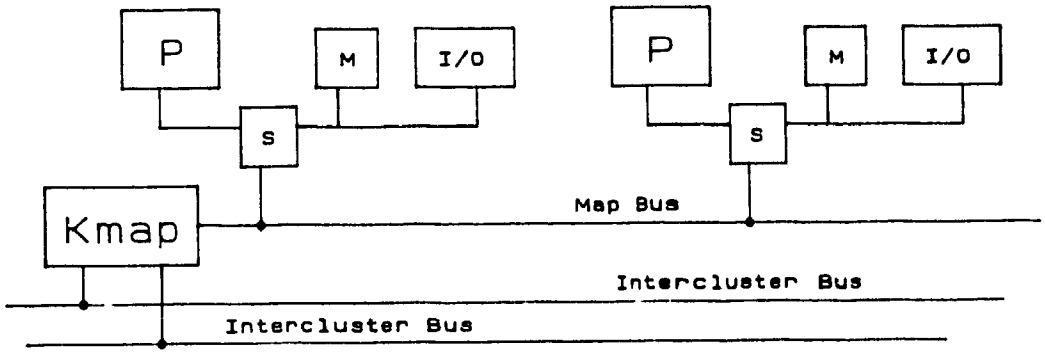


Figure 1.5: A Cluster of Computer Modules.[10]

The Local Switch in the Module allows the processor to communicate with its memory and peripheral devices. The Local Switch maps local addresses by using an internal relocation table that segments local memory on 4 Kilobyte boundaries. If a reference on non-local, it is determined by the Local Switch and it is routed automatically onto the Map bus. The Kmap unit arbitrates access to the Map bus. The Pmap unit, located within the Kmap unit is a microprogrammed processor that determines if a reference is local or to the cluster.[3] Finally, in figure 1.6, a network of clusters.

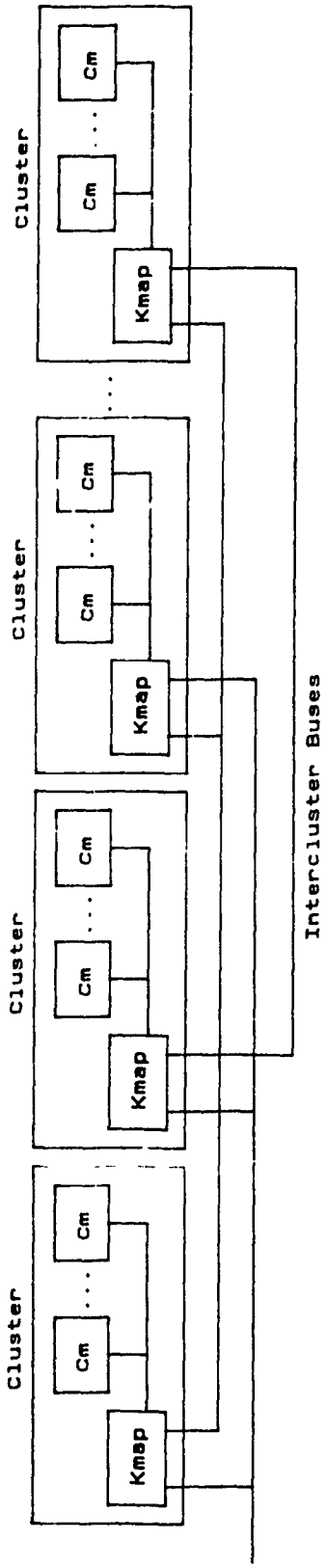


Figure 1.6: A network of clusters.[10]

1.4.3 The C.mmp

The C.mmp was designed and built at Carnegie-Mellon University in the early 1970's. It was built from slightly modified Digital Equipment Corp. PDP-11/40E computers. The modification was to remove some of the instruction set.[10] Shown in figure 1.7 is the basic architecture.

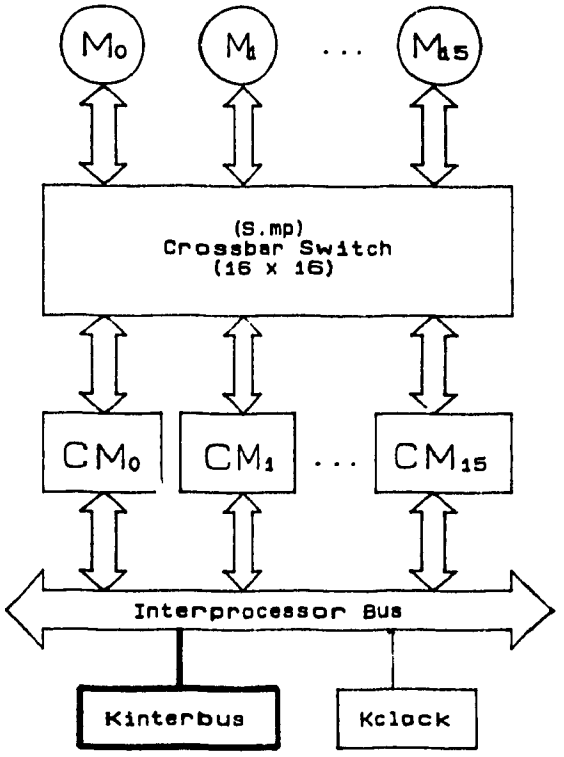


Figure 1.7: The architecture of the C.mmp.[10]

The C.mmp comprises of 16 computer modules connected to 16 shared memory modules via a 16 x 16 crossbar switch. Refer to figure 1.8 for the structure of a processor

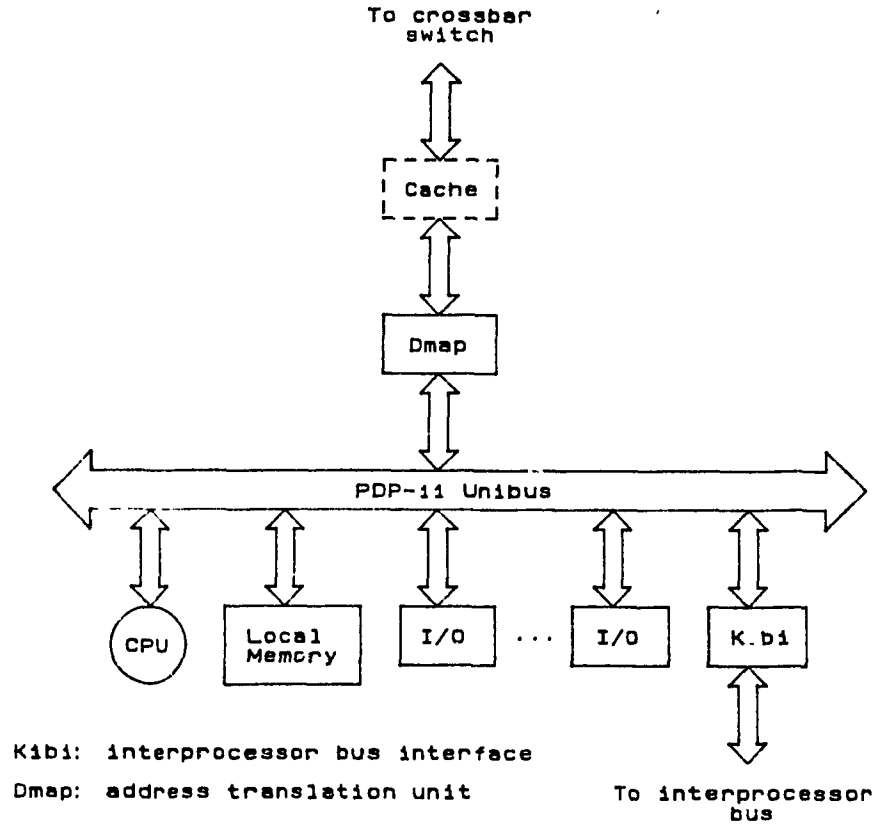


Figure 1.8: The structure of a processor in the C.mmp. [10]

Each processor in the C.mmp contains the following: 8 Kilobytes of local memory for operating system functions, four 40 Megabyte disk controllers and three 30 Megabyte disk controllers. The operating system that the C.mmp uses is called the Hydra.

1.4.4 The Cosmic Cube

The Cosmic Cube, designed and built at Caltech consists of 64 small computers that are connected by a network of point-to-point communication channels arranged as a six-dimensional hypercube. The hardware used in each of the nodes consists of an Intel 8086 processor along with an Intel 8087 floating point coprocessor. The memory consists of 128 Kilobytes of dynamic RAM with with parity check.[18]

An n dimensional hypercube contains 2^n processing elements, each of which is connected to n of its neighbours. In the case of the Cosmic Cube, n=6.[3] Therefore, there are 2^6 processing elements or 64, each connected to six of its neighbours as shown in figure 1.9.

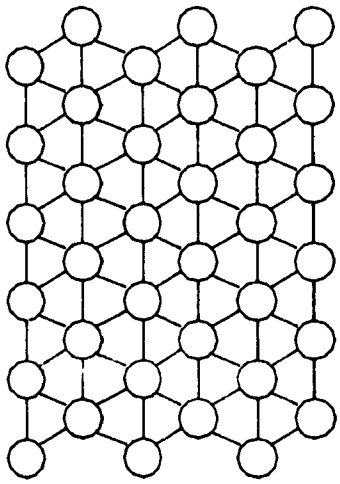


Figure 1.9: The Cosmic Cube Interconnection Network.

1.4.5 The NYU Ultracomputer

The NYU Ultracomputer is a project at New York University which proposes the design of a shared-memory MIMD parallel machine composed of thousands of autonomous processing elements. An Omega network is used for enhanced message switching between the processors. It is proposed that each of the processing elements be *custom made* and will be attached to the network via a processor network interface. Each of the memory modules will be attached to the network via a memory network interface. The *fetch-and-add* primitive is used to achieve interprocessor synchronization.

Shown in figure 1.10, the block diagram of the NYU Ultracomputer. Each of the processing elements (PE) is connected to the network using a processor network interface (PNI). Each of the memory modules (MM) is connected using a memory network interface (MNI).[7]

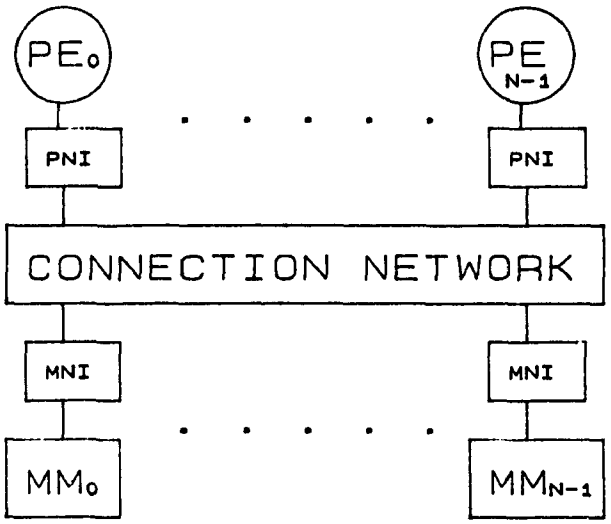


Figure 1.10: The Block Diagram of the NYU Ultracomputer.[7]

The communication network incorporated in the NYU Ultracomputer is of the Omega type. An Omega network is similar in design and operation to a Butterfly network. An address is placed on the bus and it is then decoded to route the data through the appropriate *switches* to connect to the destination memory module. Refer to figure

1.11 for the block diagram of an Omega network.

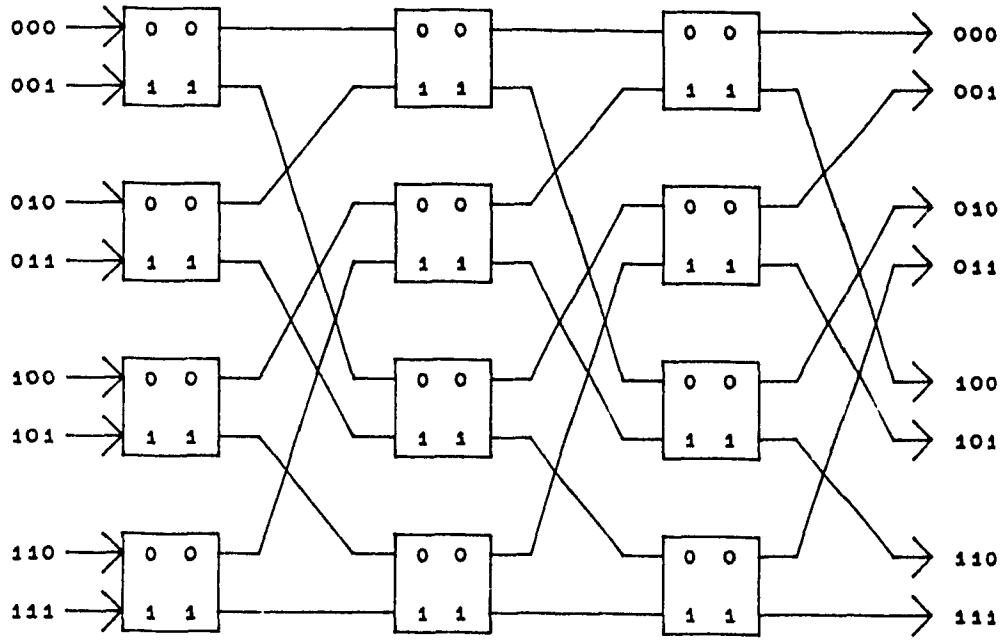


Figure 1.11: The Block Diagram of an Omega Network for N = 8.[7]

1.5 The Homogeneous Multiprocessor Proper

In the design of multiprocessors, the availability of information pathways between processors is a major architectural concern. Many of the established MIMD multiprocessors are either very expensive or slow due to their designs which have opted for a complete graph network which incorporates either crossbar switches or microprogrammed controllers.

Problems can be formulated in a way such that each of the computational processes would require information from only its nearest neighbour process to obtain the final result. By reducing the scope of the interprocessor communication, and at the same time making these communication pathways very fast, these problems would benefit greatly. This has been the realization of the homogeneous multiprocessor.[4]

The Homogeneous Multiprocessor Proper is a microprocessor based, tightly coupled, multiple instruction, multiple data (MIMD) machine that incorporates shared memory and networking. Some of the applications that are intended for the Homogeneous

multiple instruction, multiple data (MIMD) machine that incorporates shared memory and networking. Some of the applications that are intended for the Homogeneous Multiprocessor are: distributed simulation; relaxation processing; image processing; and speech processing.

As shown in figure 1.12, a block diagram of a node in the Homogeneous Multiprocessor. Each node consists of an 8 MHz Motorola MC68000 processor, a MC68451 Memory Management Unit (MMU), 16 Kilobytes of EPROM, 32 Kilobytes of static RAM, and a main memory. The main memories that have been designed and built are a 1 Megabyte dynamic memory with error detection and correction, and a 0.5 Megabyte static memory.

The memory management unit performs virtual to physical address mapping by information stored in the 32 MMU descriptors. If the processor attempts to access a protected or out-of-range segment the MMU will generate an interrupt. The MC68000 has two modes of operation both of which can be used, a supervisor state or a user state. A low level protection scheme can be implemented by using both the MMU protection mechanism and the two modes of operation of the processor.

Currently, a Motorola Tutor monitor [14], is placed in the 16 Kilobytes of EPROM. This allows for testing and debugging the hardware and software.

The Homogeneous Multiprocessor has a tightly coupled architecture. The block diagram of the Homogeneous Multiprocessor is displayed in figure 1.13. The standard system comprises k ($k \geq 3$) processing elements, k memory modules, k-1 interbus switches that control the Interprocessor Bus, and the H-Network which is a fast local area network used for point to point communication between nonadjacent processor and for broadcast mode communication.

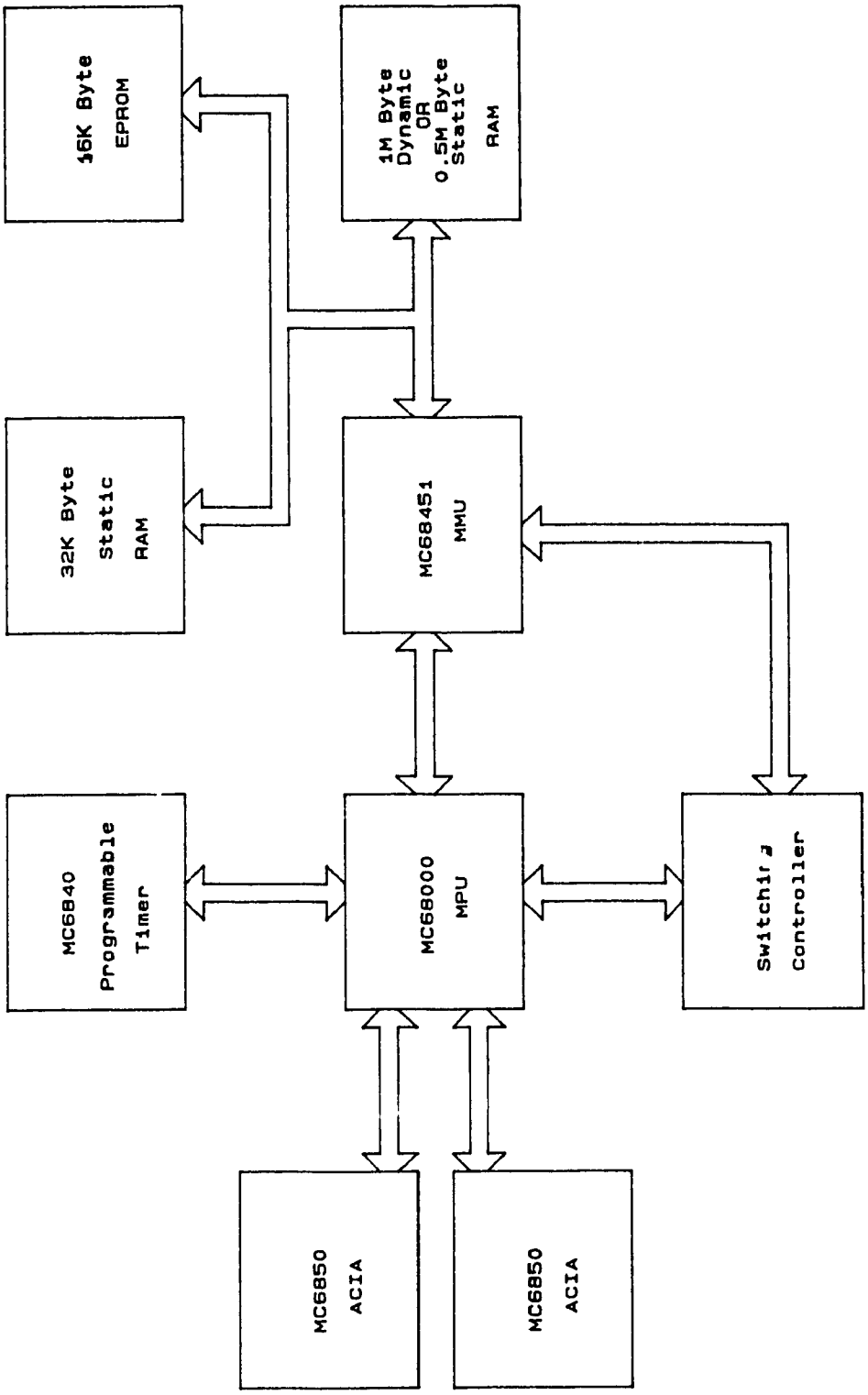


Figure 1.12: A Block Diagram of a Node in the Homogeneous Multiprocessor.

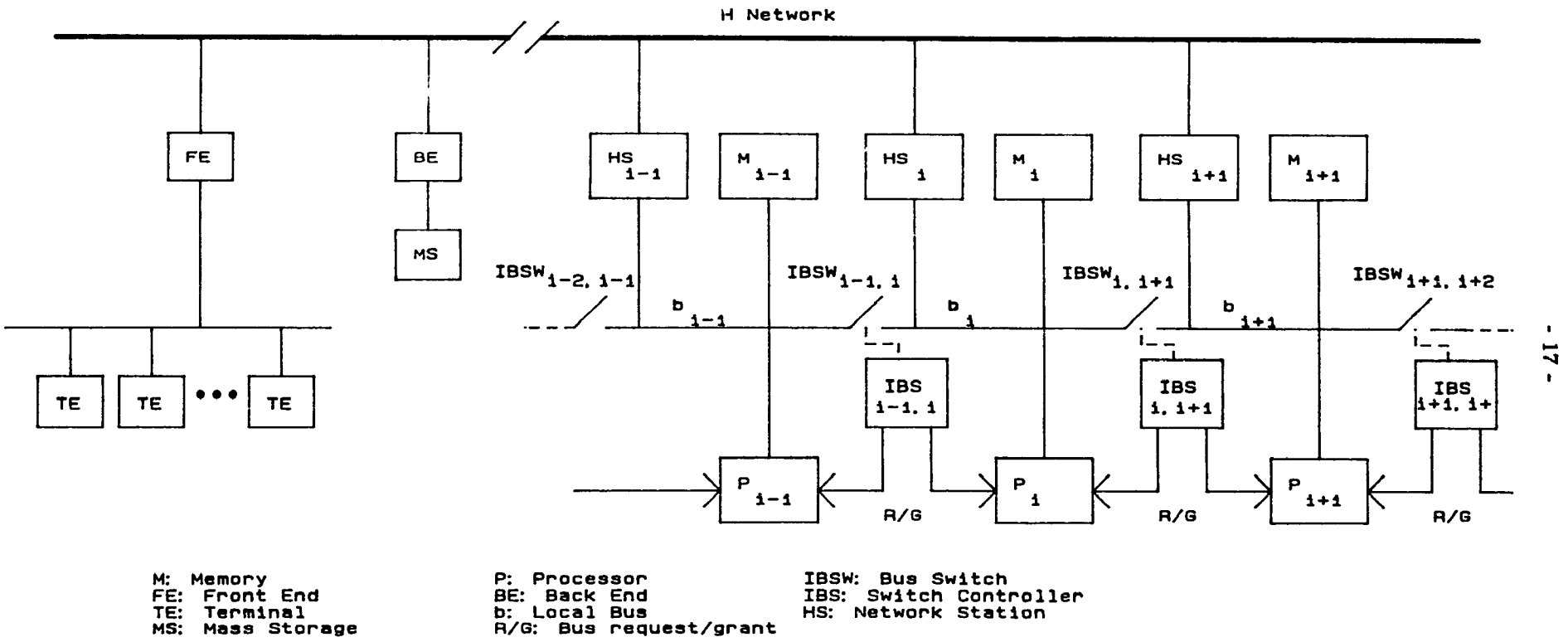


Figure 1.13: The Homogeneous Multiprocessor Proper. [5]

Each of the Processor Elements consists of the following: a processor P_i; a memory module M_i; a local bus to facilitate communication between the processor and memory M_i. The Interbus Switch is physically connected to the local buses of two adjacent Processing Elements. The Processing Elements communicate over the local area network via their network stations HS_i which are also connected to the local buses. The network interface, the H-Station is currently in the design stages. The submodule that is responsible for network acquisition within the H-Station has been designed using CMOS-VLSI technology has been sent for fabrication.

As stated previously, the Interbus Switches are used for nearest neighbour communication. When a processor accesses a neighbour's memory, an Extended bus is created. The switch physically extends a requesting processor's bus to include the memory module of the neighbour. At the same time it excludes the neighbouring processor from accessing its memory module.

An Extended Bus is defined as the dynamic fusion of two neighbouring local buses effected through the closing of the intervening switch after a request from either or both processors adjacent to the Interbus Switch. Once this Extended Bus is created, it will exist for one memory access cycle, which is also the duration of the request. The creation of the extended bus comprises of two parts commonly known as Phase I and Phase II. Phase I, which is processor independent, ensures that deadlock scenarios will never exist and the operation of the switches are mutually exclusive. During Phase II, which is processor dependent, the switch physically closes. Once the first phase has completed with success, the second phase begins.

After the cycle, the Extended Bus deteriorates to its original separate local buses. Once this happens, the Interbus Switch is ready to comply with any pending or next request for an Extended Bus. More details on the operation of the Interbus Switch can be found in chapter 2.

The second method of communication is the H-Network. The H-Network is a high speed (approximately 7 Megabytes/second) Local Area Network similar in structure to the Ethernet. It differs in that it contains distinct pathways for data transmission, network acquisition, and collision detection. The signal propagation delay is extremely short because the H-Network has been designed with a span of the order of 10 meters. This, coupled with the parallelism of the distinct pathways has the effect of increasing the performance of the network.

The Homogeneous Multiprocessor is built on Versabus boards and utilizes the Versabus card cage. The Versabus *system* is used because at the time of adoption (circa 1982), it was the only 32 bit bus supported by Motorola. A note of interest is that the VME bus was adopted by Motorola in subsequent years. The Versabus has been modified for our purposes. The main connector has been left unmodified. This provides the processor memory interface, and it contains all the bus control signals coming out of the MC68000. The alternate or second connector, is used for access to neighbouring processors (using the Interbus Switch). The original purpose of the alternate connector of the bus was to extend it to 32 bit, as well as to provide I/O lines.

Architectural Innovation is present in the design of the Homogeneous Multiprocessor in that it is capable of behaving as a distributed system coupled through a fast local area network and as a tightly coupled pipeline of processors. Applications such as distributed file systems, distributed databases, distributed simulation, etc, exist in an ideal environment provided by the H-Network. Finally, algorithms such as the ones found in vision and digital signal processing, and relaxation processing can be implemented efficiently due to the Extended Bus mechanism.

At the time of this writing there are three Processing Elements that are fully operational (processor and memory subsystems). One of the Processing Elements contains 1 Megabyte of Dynamic RAM while the other two contain 0.5 Megabyte of Static RAM. In the memory design incorporating static memory it is possible to expand this to one

Megabyte. Included in the dynamic design, there is also error detection and correction which facilitates the most reliable and cost effective method for reducing corruption of data when using dynamic memories. The static memory design does not require error detection and correction due to the nature of the memory chips. As far as expandability is concerned, the only limiting factor is space and the addressing capabilities of the MC68000.

1.5.1 Performance Analysis of the Homogeneous Multiprocessor

The structure of the Homogeneous Multiprocessor has been simulated to obtain a performance evaluation. The simulator that was written is based on an 8 MHz MC68000 that implements the second phase of the interbus switch controller which is processor dependent. The first phase which is processor independent uses Algorithm 1.2 (described in chapter 2).

The behaviour of the Homogeneous Multiprocessor was obtained for two scenarios. The first scenario simulates the performance of the system based on the demand to the Interbus switching network and the second scenario simulates a distributed algorithm. The distributed algorithm determines the autocorrelation functions of a given signal.

The first simulation determined that the idle time of a processor never exceeded 30% of the total processing time. The idle time of a processor is attributed to waiting while a neighbouring processor accesses its memory module. Even in the most demanding cases, the idle time remained within acceptable bounds.

The results of the second simulation were as follows. The simulation comprised of 19 processors processing 1024 samples. This obtained a speed up factor of 11.85 which is approximately 62% of the theoretical value.[5]

1.5.2 Software Design for the Homogeneous Multiprocessor

To provide users with a programming environment as well as controllable access to the hardware, an important feature to any computing system is the operating system. While constructing and testing the hardware, the operating system was being designed. Using the first prototype board, the lower layers of a preliminary single node operating system was tested and debugged. It should be noted that some hardware defects were uncovered through the operating system design.

One of the major concerns to the designers of the operating system for the Homogeneous Multiprocessor was to give to the users the *raw* architectural features in a usable fashion. The main objective in the design of the operating system was to enable the addition or subtraction of processing elements without having to do major redesigning in either the hardware or the software. The operating system is designed in a modular fashion thus enabling additional routines or software packages to be added to the system with grace and ease.

The operating system is based on a nucleus structure - the HM-Nucleus. It provides the basic mechanisms for utilising the the bare hardware. This enables resource management and application software to be conveniently built on top of it. Primitives for interprocess communication, capability checking, memory management, process management, an I/O handling are provided. Each processor has a copy of the nucleus residing in it. Therefore there is no globally shared memory in the Homogeneous Multiprocessor. When the HM-Nucleus is complete, an objective is to have the operating system utilities residing only in some nodes and applications will be assigned to them on demand. There will be no swapping of memory, an application will remain resident until execution is complete.[11]

1.8 Objectives of This Research

Summarizing, the objectives of the research presented in this thesis are the following. In chapter 2, the design and implementation of the Interbus Switch, as per the specifications, are presented. In appendix A and appendix B, the designs of the dynamic and static memory are presented, respectively. In appendix C, the numerical analysis and schematics for the interbus switch are provided. And finally in appendix D, the schematics for the Homogeneous Multiprocessor are presented.

In this thesis, the entire process of design and implementation for the memory systems is presented. For the Interbus Switch, the design, prototyping in discrete components and testing are presented, however, final implementation by using high density technology (eg. PLA / gate array) is yet to be accomplished.

2. The Interbus Switch

The interbus switch is used to connect a processor's local bus to the bus of one of its neighbours. The primary objective of this scheme, is to enable a processor to access memory belonging to a neighbour, and thus establish interprocessor communication. This process of communication is accomplished by an *extended bus.* (An extended bus is formed by physically connecting the local buses of two neighbouring processors. This is accomplished by opening/closing appropriate intervening switches). In doing this, the processor has potential access to three memory modules. It can access its own memory (normally), the memory of its neighbour on the left, or its neighbour on the right (through the creation of an extended bus).

In order for the processor to access a memory module, it presents an address on the Interbus Switch Controller. The two most significant bits of the address are used to determine which memory module the processor wants to access. Shown in table 2.1 is the memory map that is used to determine which module the processor is requesting. From this point onward, it is up to the interbus switches to give access to the memory module in question.

Table 2.1: The Interbus Memory Map.

A_{23}	A_{22}	Mapping
0	0	I/O
0	1	Memory on the Left
1	0	Memory on the Right
1	1	Own Memory

Once the interbus switch intercepts the address, it will determine where the processor wants to access and then, depending on the surrounding environment, eventually allow access to the memory module in question. The processes that occur in the interbus switch are divided. The first process that occurs is identified as Phase I, followed by Phase II. Once these processes are completed, the memory access is granted.

Phase I determines whether it is safe to close the switch it controls. The determina-

tion of safe closure is based on the state of the two neighboring switches. If either one or both are physically closed, or is about to close, the present switch must stay open so as not to cause collisions on the *extended bus.*

Phase I, more commonly known as Logical Closure of the switch is used to determine the state(s) of the neighbouring switches. The state of the neighbouring switches is very important. If a neighbouring switch is Physically or Logically Closed, then Phase I must interpret this information and take an appropriate action in order not to cause collisions on the *extendea bus.* A collision could cause disastrous effects on the Homogeneous Multiprocessor.

Collisions can occur if for example, two processors are connected to the same memory module. If this were to happen, both processors would attempt to access the memory module, and their address lines would short with each other. Hence, a collision. Another example of a collision is if a processor is connected to two memory modules, and it attempts a read access. The data lines carrying the information from the memory modules would be shorted and a collision would occur. These types of occurrences must be prevented at all costs to ensure the proper operation of the software and hardware.

Phase II, also known as Physical Closure of the switch consists of two parts. The first part is a bus arbiter that adheres to the operation of the 68000 CPU. It uses the method of arbitration required by the 68000 CPU for a bus request that is identical in every respect to one that a coprocessor would adhere to. Once arbitration is completed (if necessary), various combinational circuits will Physically Open and/or Close the bus switches. Depicted in figure 2.1 is the block diagram of the Interbus Switch Controller ($IBS_{I,J}$).

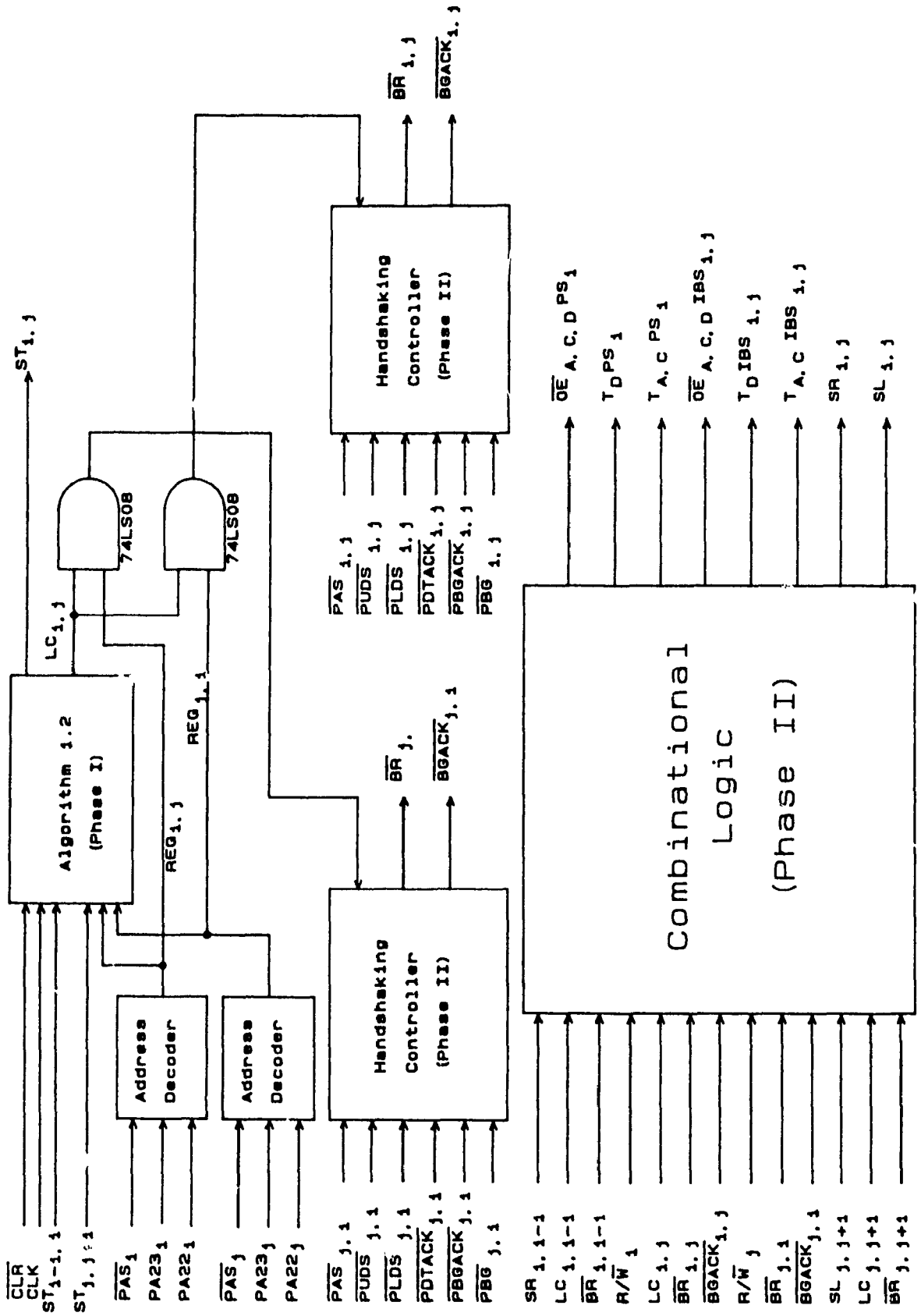


Figure 2.1: The Block Diagram of the Interbus Switch Controller.

2.1 Phase I

As stated in the previous section, Phase I is used to determine the Logical Closure of a switch. The Logical Closure of the switch is based on an algorithm known as Algorithm 1.2.[5] Algorithm 1.2 describes a procedure required for the safe (i.e. no two adjacent switches will close at the same time) and live (i.e. a switch requested to close will eventually do so).

The switches exist in one of three states. They are Open, Grey, and Closed. Their definitions are as follows:

OPEN: This state signifies that no request exist or if a request exists it will not be honoured immediately, because a neighbouring switch is currently servicing a request.

GREY: This state signifies that a request is acknowledged and that service (i.e. switch closure) will be granted in the immediate future.

CLOSED: This state signifies that it is safe for a switch to close. The actual closure of the switch will take place during Phase II which commences immediately after a switch enters the Closed state.

For a switch $IBS_{i,j}$

If no other request exists, it becomes Open;

Otherwise, if a request exists then:

If Open, it becomes Grey provided that the switch to its left, $IBS_{i-1,i}$, is Open.

Otherwise, it remains Open.

If Grey, it becomes Closed provided that the switch to its right, $IBS_{j,j+1}$, is open.

Otherwise, it remains Grey.

If Closed, it remains Closed.

The leftmost switch $IBS_{0,1}$ and rightmost switch $IBS_{n,n+1}$ are always Open.

When a processor requests access to its neighbours' memory, the request remains asserted during the request period which terminates with an acknowledgement (from the

requested Memory Module to the requesting Processor). This request is intercepted by the appropriate Interbus Switch Controller and is forwarded to the requested Memory Module after the switch closes, then the Memory Module acknowledges the transfer of data to the requesting Processor, and the cycle terminates. Algorithm 1.2 guarantees the safe (i.e. no two adjacent switches will close at the same time) and live (i.e. a switch requested to close will eventually do so) operation of the network of Interbus Switches.

In order for a switch to actually close, the Processor of the requested Memory Module must take itself off the Local Bus, in order for the memory access to proceed without interference. We have tried to make the Interbus Switch Controller compatible with the Motorola MC68000 CPU operating at a frequency of 8 MHz. Algorithm 1.2 requires three clock cycles to reach a stable state reflected by the number of transitions (Open-Grey-Closed). It is therefore expected that the state of the Phase I will stabilize within one CPU clock cycle (i.e. Phase I operating at least at 24 MHz).

2.1.1 Design and Implementation of Phase I

Shown in figure 2.2 below is the state diagram for Algorithm 1.2. R is the existence of a request which is determined by the logical OR of a request from the processor on the left, denoted as R_I, and a request from the processor on the right, denoted as R_J. $ST_{I-1,I}$ is the status of the Interbus Switch on the left and $ST_{J,J+1}$ is the status of the Interbus Switch on the right. The status indicators $ST_{I,J}$ denote whether the switch is OPEN ($ST_{I,J}$) or non-OPEN ($\overline{ST_{I,J}}$).

The hardware was designed and implemented to execute Algorithm 1.2. The design was carried out extensively to determine the minimum amount of integrated circuits required. The design was carried out using various state assignments and then determining the circuits using both J-K flip flops and D-type flip flops. Only the most efficient design procedure will be derived in the following text. The hardware consists of 2 J-K flip flops arranged as a three state machine. The fourth possible state that is not used is setup such that if it is ever entered, then the circuit will be forced into the initial state.

Shown in table 2.2 below is the state assignment for the machine which is derived from the state diagram.

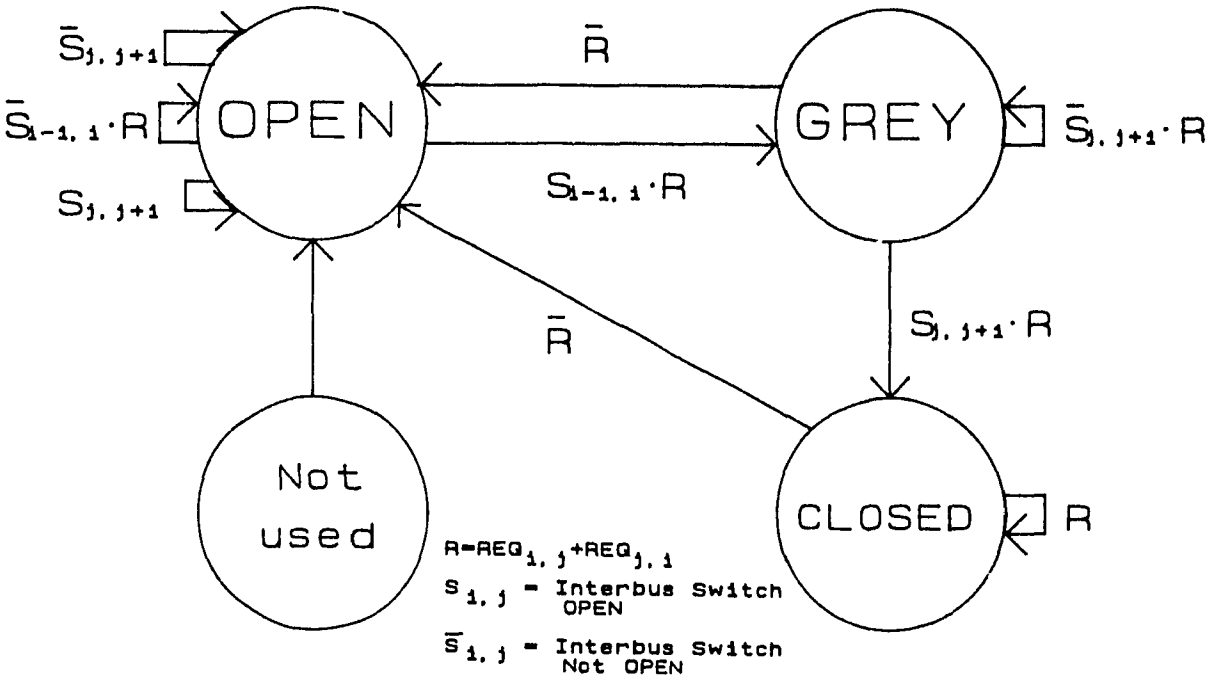


Figure 2.2: State Diagram For Phase I.

Table 2.2: State Assignment For Phase I.

State	Q_1 Q_0
Open	0 0
Grey	0 1
not used	1 0
Closed	1 1

Upon power up or reset, the machine enters the OPEN state. If there is no request, the switch remains in the OPEN state. If the switch on the left, $IBS_{i-1,i}$ is non-OPEN and a request exists, then the machine also remains in the OPEN state. Once the switch on the left, $IBS_{i-1,i}$ opens, the machine changes to the GREY state. Once in the GREY state, if the request ceases to exist, then the machine returns to the OPEN state. Otherwise, if the request continues to exist and the switch on the right, $IBS_{j,j+1}$, is non-OPEN, then the machine remains in the GREY state. If the switch on the right, $IBS_{j,j+1}$ is open and

the request still continues to exist, then the machine changes to the CLOSED state. Once in the CLOSED state, as long as the request exists, the machine will remain in the CLOSED state. When the request ceases to exist, then the machine returns to the initial OPEN state. The existence of a request is only dependent on the logical OR of the request line from the adjacent processing elements. As far as Phase I is concerned, it does not matter which processor exerts the request.

In the design of Phase I, only three of the four states are required. The operation of the sequential machine is quite simple. At first, the flip-flops are cleared, denoting the OPEN state. The first operation that will occur is changing Q_0 from a logical 0 to a 1. This is equivalent to changing from the OPEN state to the GREY state. To allow the machine to enter into the GREY state, it must verify that the neighbouring switch on the left, $IBS_{I-1,I}$ is open. The following equations, 2.1 and 2.2 denoted the equations for J_0 and K_0, respectively.

$$J_0 = \overline{Q_1} \cdot R \cdot ST_{I-1,I} \tag{2.1}$$

$$K_0 = \overline{R} \tag{2.2}$$

The first equation (2.1), J_0 is used to change the sequential machine from the OPEN state to the GREY state. This is achieved by checking for the existence of the request R, the switch on the left, $IBS_{I-1,I}$, is OPEN, and that the machine is in the OPEN state ($Q_1 = 0$). If at any time, the request (R) ceases to exist, K_0, equation 2.2, will return the flip flop to the initial state ($Q_0 = 0$). The next two equations, 2.3 and 2.4 represent J_1, and K_1, respectively.

$$J_1 = Q_0 \cdot R \cdot ST_{J,J+1} \tag{2.3}$$

$$K_1 = \overline{Q_0 \cdot R} \tag{2.4}$$

The second flip-flop only comes into action once the first flip-flop is set, (GREY state). This is accomplished by J_1 checking the existence of the request (R), the switch on the right ($IBS_{J,J+1}$) is OPEN, and that the sequential machine is in the GREY state ($Q_0 = 1$). If all of the above conditions are true, then the flip flop changes state putting

the sequential machine into the CLOSED state. Once in the CLOSED state, the machine will remain there until the request ceases to exist, as shown by equation 2.4. The other term in equation 2.4 (Q_0) is used to reset the machine if it enters the unused state ($Q_1Q_0 = 10$). The derivation of these equations can be found in Appendix C.

The schematic shown below in figure 2.3 is the hardware implementation of Algorithm 1.2. It is composed of five standard 74LSXX series chips. The inputs to the circuit are the address strobe and two most significant address bits of the processors on the left and right. Also, the status of the Interbus switchs on the left and right. There are also a clock and a clear. The clock is active all the time while the clear is only activated during power up or reset. Based upon the data sheets supplied by TI, the circuit should operate at least at 24 MHz so that the algorithm can run from Open to Closed in one CPU clock cycle (8 MHz) given that the neighbouring Interbus Switches are open. If one or both of the Interbus switches are not Open, then there will be a time delay before the Algorithm runs to completion. This is due to the waiting period until the neighbouring switch that is not Open returns to the Open state. At that point, the Algorithm will continue to run until it reaches the Closed state. Upon "power up" of the Multiprocessor, it is possible for Phase I to exist in the state 10 (not used), however, the reset signal that is supplied by the system at power up time, will force the switch to the open state. The Logical Closure signal is fully decoded and the unused state will not affect the operation. If by chance the sequential machine enters the unused state, the circuit is designed to force the machine into the reset state which will open the Interbus Switch.

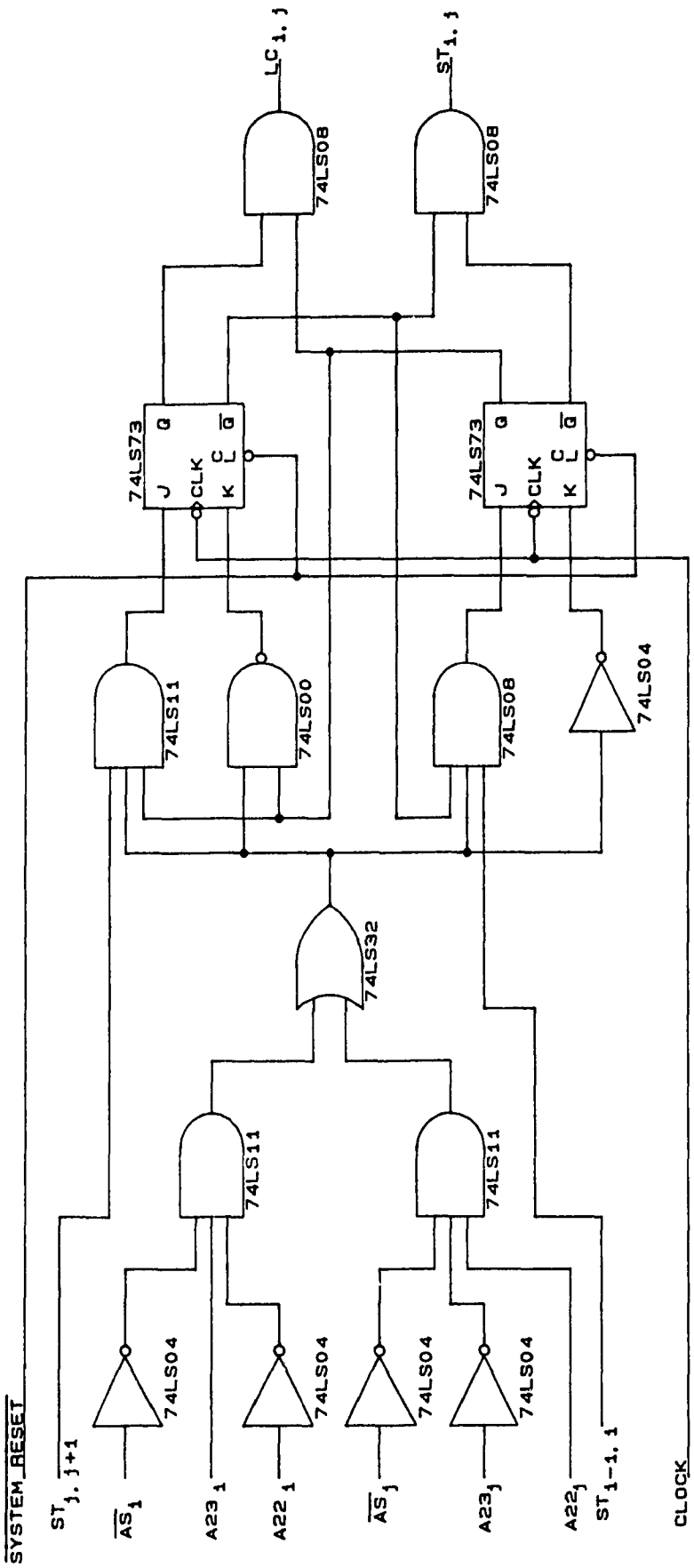


Figure 2.3: Hardware Implementation For Phase I.

In the design of the hardware for Phase I, the following inputs were required: clear the flip-flops (denoted as $\overline{CLR}$); a 24 MHz clock (denoted as CLK); the status of the neighbouring switches (denoted as $ST_{i-1,i}$ and $ST_{j,j+1}$); and the request inputs (denoted as R_i and R_j). There are two outputs from Phase I, they are: Logical Closure (denoted as LC) which will be used as an input to Phase II; and the Status (denoted as ST) which is the status bit used for the neighbouring Switch Controllers.

The $\overline{CLR}$ input is used to reset any flip-flops in Phase I during a power up or reset of the system. The CLK input is used to clock data into the J-K flip-flops that are used in Phase I. Requests are generated for an Interbus Switch to physically close via the R_i and R_j inputs. These inputs are obtained by using gates to check the upper two address bits and the address strobe (A_{23}, A_{22}, and $\overline{AS}$) of a processor. If the appropriate address and strobe are produced then the existence of a request is concrete. The status bits are used by Phase I to determine if a neighbouring switch is closed or about to close. This is important because two neighbouring switches should never be closed at the same time. These inputs are obtained from the neighbouring Phase I circuits. They are the Status outputs (ST).

The Logical Closure (LC) is used to inform Phase II that it may start its own procedure which is described below. The status output is used as an input to the neighbouring Phase Is. Each Phase I has two status bits, one from the neighbour on the left, and the other from the neighbour on the right. Therefore the neighbouring Phase Is will use the ST output to determine the status of this switch. As long as this switch is not Gray or Closed, the ST signal will be active.

2.2 Phase II

Once Phase I has run to completion (i.e. in the Closed state), Phase II begins. During Phase II, the Physical Closure of the switch is accomplished. Phase II consists of two parts, one of them is a handshaking controller which conforms to the protocol of Mastering the Bus of a MC68000 Processor (i.e. equivalent to that of a DMA for example), and

the second is an arbiter. The purpose of the arbiter is to resolve potential deadlocks in the system.

In the simplest case, the requesting processor in Phase II will output a Bus Request ($\overline{BR}$) to a neighbouring processor for its memory. It will then wait until it receives a Bus Grant ($\overline{BG}$). Finally the requesting processor will issue its own Bus Grant Acknowledge ($\overline{BGACK}$) at the end of the previous cycle (i.e. when all $\overline{AS}$, $\overline{DTACK}$, $\overline{UDS}$ and $\overline{LDS}$ at the bus of the requested processor are negated) for the duration of the memory access. Once the memory access is completed, the $\overline{BGACK}$ is removed and normal processing continues. Refer to figure 2.4 for the proper timing for Mastering the Bus. However, as stated previously, this is not always the case. The deadlock situations (discussed below) occasionally interfere with this protocol. Therefore some extra hardware is required to rectify this problem.

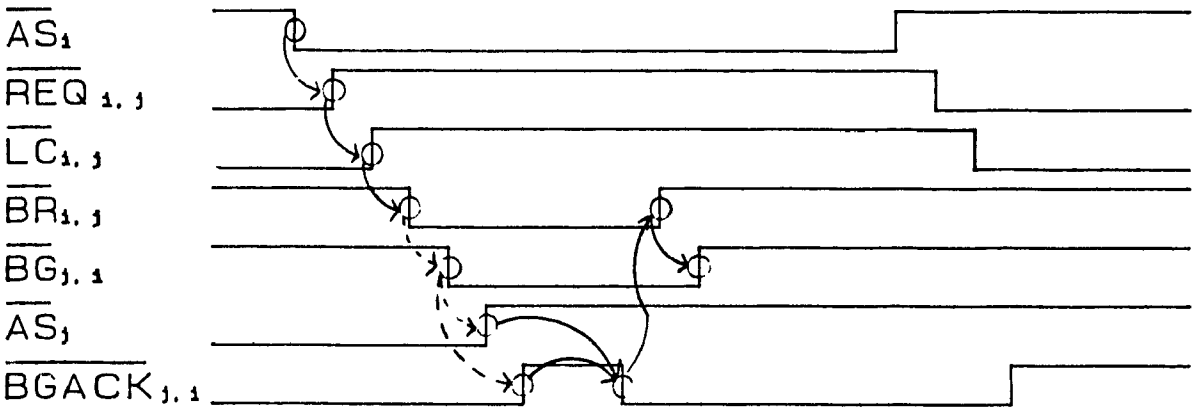


Figure 2.4: Timing Diagram For Mastering the Bus.

As previously stated, Phase II must first check for potential deadlocks in the system before it allows the arbiter to start (i.e. activate the $\overline{BR}$). There are three cases of potential deadlock. Each one of them will be discussed in the following sections. A deadlock can occur when two adjacent processors request the same switch to close. Upon the completion of Phase I, the processors are waiting for a Bus Grant ($\overline{BG}$). However, if the processor that is supposed to assert this Bus Grant is itself waiting for a Bus Grant

then, both processors will wait indefinitely and hence never complete the pending Extended Memory access cycles. Phase II has been designed to recognize these potential deadlock cases and resolve them by giving priority to one of the requesting processors.

In the first case of potential deadlock, two adjacent processors request the Interbus Switch located in between them to close as shown in figure 2.5. According to the asynchronous bus protocol of the processors, each processor will wait for the Data Transfer Acknowledge ($\overline{\text{DTACK}}$) from the Memory Module they have requested, respectively. The $\overline{\text{DTACK}}$ will never arrive due to the deadlock and hence the memory access cycle can never run to completion. Ideally, Phase II would request the bus from one of the two processors and wait for the Bus Grant ($\overline{\text{BG}}$) which would be asserted from the requested processor at the end of its current bus access cycle. Since both processors are in the middle of a bus access cycle waiting for the Interbus Switch to close, they will not assert the Bus Grant until they finish their bus access cycle and at the same time they cannot complete the access cycle because the Interbus Switch is waiting for their Bus Grant in order to close the Interbus Switch to allow them to complete their access cycles.

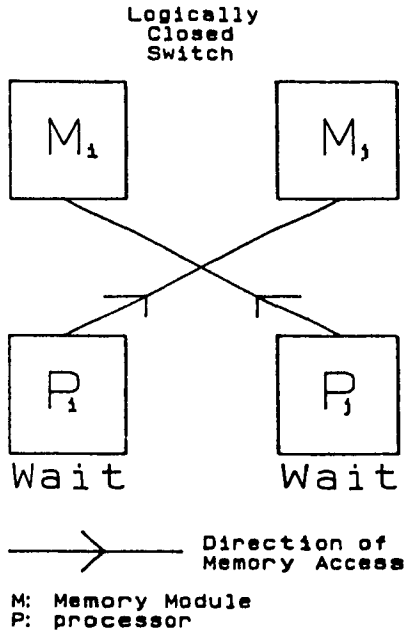


Figure 2.5: Case I of Potential Deadlock.

Phase II will resolve this case of potential deadlock by first granting access to the processor located on the right side of the Interbus Switch. This is accomplished by forcing the processor located on the left of the Interbus Switch off the bus. Then the processor on the right will complete its Extended Memory access by receiving the $\overline{\text{DTACK}}$. Finally the processor on the left, which has been waiting for this to occur will be granted the bus allowing it to complete its Extended Memory access cycle.

The second case of potential deadlock is depicted in figure 2.6. In this scenario two adjacent processors, P_i and P_j, request their neighbours memory modules on the left, M_{i-1} and M_i, respectively. Both of the processors initiate a bus access cycle and issue Bus Requests to Interbus Switches, $IBS_{i-1,i}$ and $IBS_{i,j}$, respectively. Now, if processor P_j requests switch $IBS_{i,j}$ to close at the same time or earlier than processor P_i requests switch $IBS_{i-1,i}$ to close, then according to Algorithm 1.2, switch $IBS_{i,j}$ will eventually Logically Close while switch $IBS_{i-1,i}$ will remain either Logically Open or Logically Gray.

At this point, processor P_j is in the middle of Phase II, having asserted its Bus Request and awaiting a Bus Grant. Processor P_i has asserted the appropriate address on the bus and is in the middle of Phase I, but not yet Logically Closed. Now, even though processor P_i has not yet reached Phase II, it still cannot assert a Bus Grant to processor P_j because it is waiting to complete its memory cycle in order to do so. It is waiting for Interbus Switch $IBS_{i,j}$ to become Logically Open in order to finish its cycle. Interbus Switch $IBS_{i,j}$ cannot become Logically Open until processor P_j completes its own Extended Memory access cycle, which at the moment is waiting for processor P_i to assert the Bus Grant. Hence, the deadlock situation.

The potential deadlock situation is corrected in Phase II by granting mastership of the bus to the processor on the right, P_j. During this time processor P_i waits for processor P_j to complete its Extended Memory access cycle. Once completed, processor P_i will eventually become Logically Closed and then assert the Bus Request for the bus and wait for the Bus Grant to complete its own Extended Memory access cycle.

Finally, the third case of potential deadlock. As shown in figure 2.7, it is very similar to the previous situation. In this case, processors P_i and P_j request the memory modules of the neighbours on their right, M_j and M_{j+1}, respectively.

In this situation, processor P_i has completed Phase I and now has the Logical Closure required to start Phase II. At the same time, processor P_j has requested its neighbouring memory by outputting the appropriate address but it is in the Logical Open state due to Algorithm 1.2 (i.e. the neighbour on the left is Logically Closed, therefore wait). Now, processor P_i is waiting for the Bus Grant from processor P_j, but it will never appear because processor P_j is itself in the middle of a memory access cycle which can only be completed once it becomes Logically Closed and runs through Phase II and eventually receives its $\overline{\text{DTACK}}$.

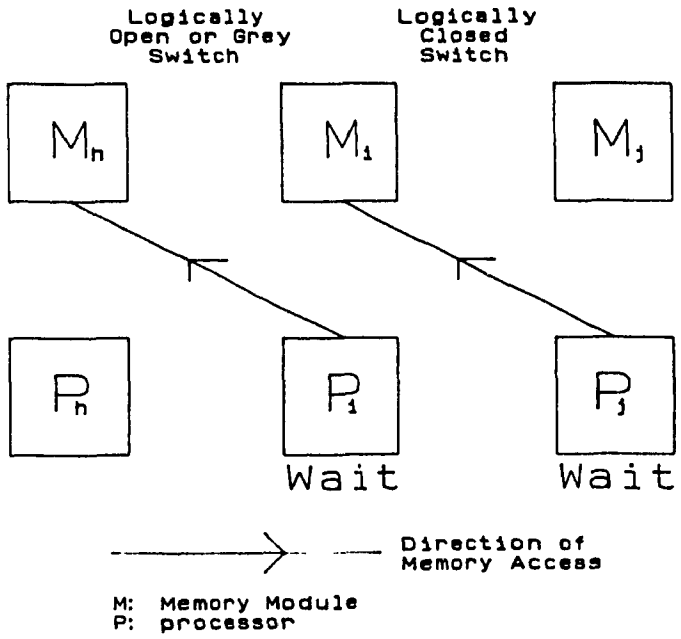


Figure 2.6: Case II of Potential Deadlock.

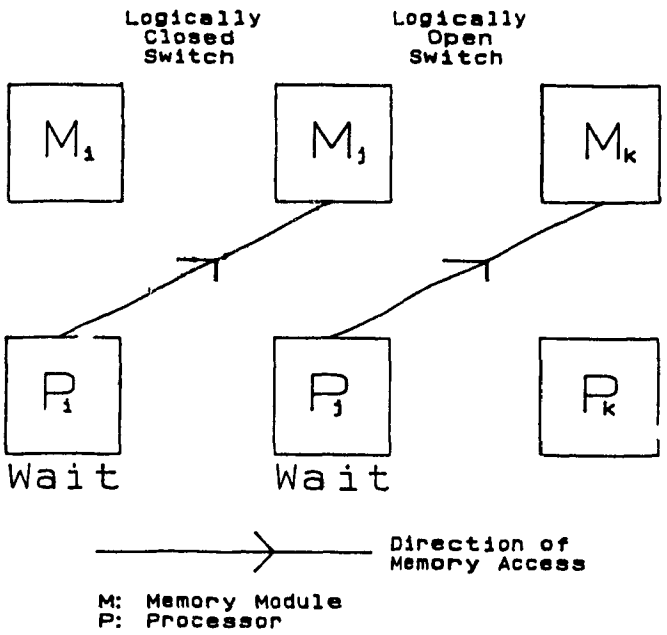


Figure 2.7: Case III of Potential Deadlock.

Phase II will resolve this case of potential deadlock by forcing processor P_j off the bus. Then processor P_i completes its Extended Memory access cycle and finally processor P_j receives the bus to complete its own Extended Memory access cycle.

2.2.1 Design and Implementation

In this section we present the design of Phase II. Phase II, consists of two parts, the first part is a simple three state sequential machine that is used for mastering the bus (i.e. for a processor to obtain the neighbours bus). The second part of Phase II is the switch controlling circuitry which consists of two parts. The switch controller must operate the switches during normal access which consists of regular processor - memory interactions and processor - neighbouring memory interactions. The second part must also resolve the three cases of potential deadlock so that processing can continue.

In order for a processor to obtain the neighbours bus, the handshaking described in the previous section must be implemented. The processor that makes the request puts the appropriate address on the bus. This then generates the request R. Eventually, the Logical Closure appears from Phase I. These two signals logically ANDed create an

enable line for the handshaking signal. This enable is required because there will be two of these circuits in each of the Interbus Switch Controllers. One of them is for the processor on its left to obtain the bus of the processor on its right, and the other for the processor on its right to obtain the bus of the processor on its left. Since the Logical Closure does not in any way describe which processor wants to access the bus of the other, two separate enables are required. Figure 2.8 depicts the state diagram for the handshaking required during Phase II.

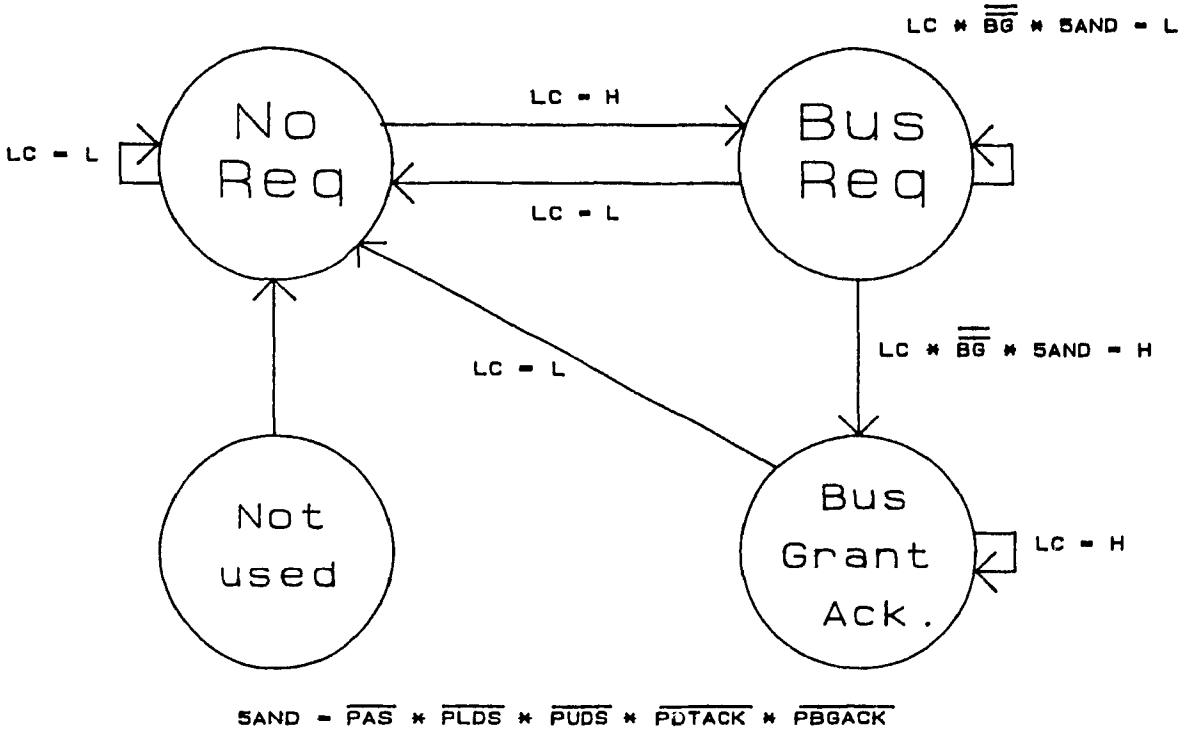


Figure 2.8: State Diagram for Handshaking During Phase II.

As stated earlier, the handshaking can only begin once both the Logical Closure and the Request are asserted. The first state, A, represents the sequential machine when neither the $\overline{BR}$ nor the $\overline{BGACK}$ are asserted. Once the Logical Closure and Request signals are active, the machine enters state B. During state B, the $\overline{BR}$ is asserted and the machine waits for the requested processor to assert its $\overline{BG}$ and at the same time wait for the current cycle of the requested processor to reach completion by negating its $\overline{BGACK}$, $\overline{AS}$, $\overline{LDS}$, $\overline{UDS}$, and $\overline{DTACK}$. Once this occurs, the sequential machine enters state C.

While in state C, the $\overline{BGACK}$ gets asserted and the requested processor will inactivate its $\overline{BG}$. During the time that the $\overline{BGACK}$ is active, the requesting processor has control of the requested processors bus (unless there is a potential deadlock). When the requesting processor finishes its cycle, it removes the address and address strobe from the bus, thereby deactivating the request R, causing the handshaking controller to return to the initial state. Shown in table 2.3 below is the state assignment for the revised state diagram.

Table 2.3: State Assignment for the Handshaking in Phase II.

State	Q_1	Q_0
NO REQ	0	0
BR	0	1
BGACK	1	1
N.U.	1	0

The Following are the equations that describe the handshaking controller.

$$J_0 = \overline{Q_1} \cdot LC \tag{2.5}$$

$$K_0 = \overline{LC} \tag{2.6}$$

$$J_1 = Q_0 \cdot LC \cdot \overline{BG} \cdot \overline{PAS} \cdot \overline{PDTACK} \cdot \overline{PUDS} \cdot \overline{PLDS} \cdot \overline{PBGACK} \tag{2.7}$$

$$K_1 = \overline{Q_0} + \overline{LC} \tag{2.8}$$

$$\overline{BR} = \overline{Q_1 \cdot Q_0} \tag{2.9}$$

$$\overline{BGACK} = \overline{Q_1 \cdot Q_0} \tag{2.10}$$

Shown in figure 2.9, the hardware implementation for the handshaking during Phase II.

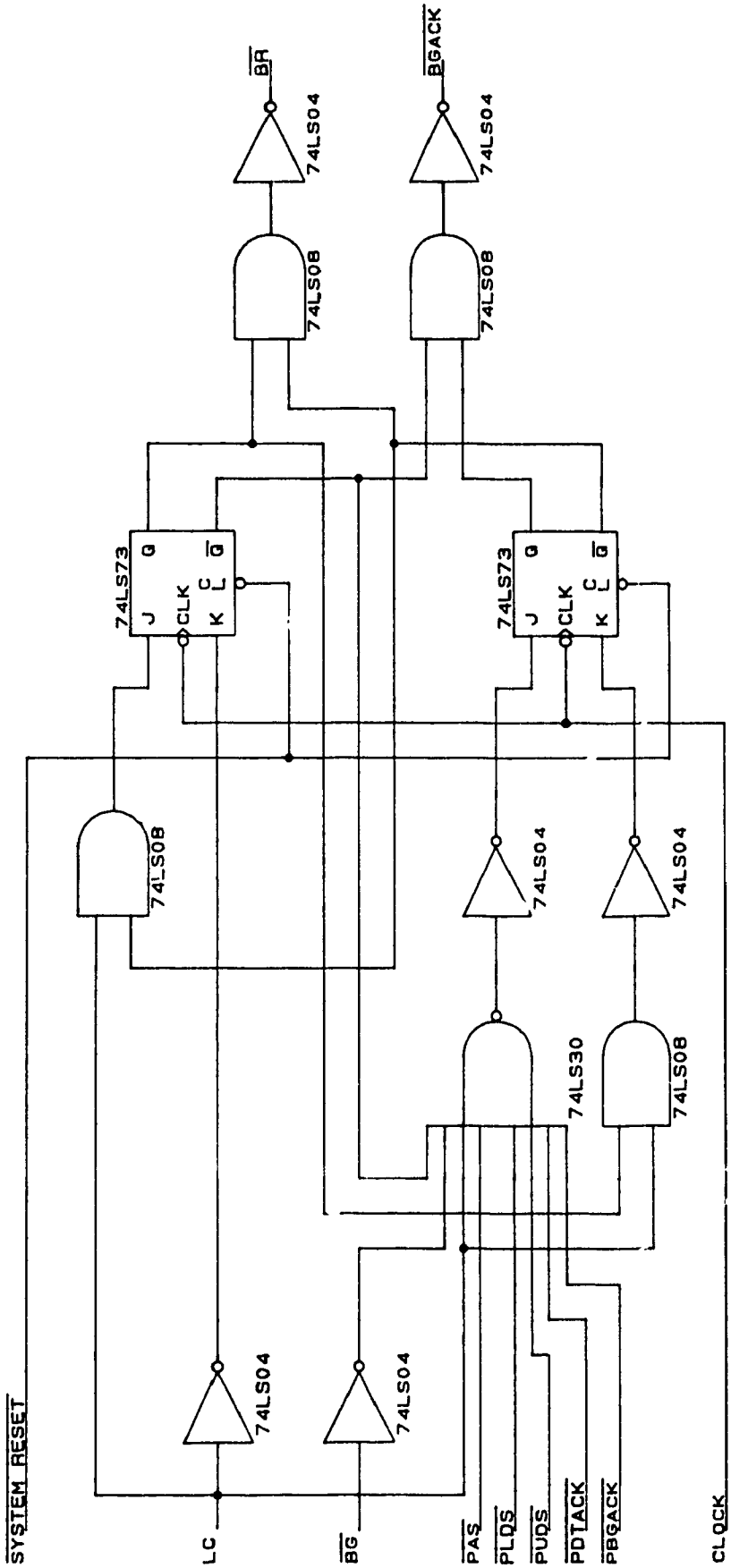


Figure 2.9: Hardware Implementation for Handshaking During Phase II.

The switches that are used in the Interbus Switch and Processor Switch are manufactured by Intel. They are the 8286 Bidirectional Bus Buffers. The buffers contain an output enable ($\overline{OE}$), a direction control (T), and 8 sets of bidirectional buffers. The Processor Switch is used to tri-state the Interbus at the point of the processor, so that when a neighbour obtains the bus, there are no collisions. The 8286 buffers have designated inputs/outputs on each side, they are labeled A and B. Shown in table 2.4 is the truth table for the 8286.

Table 2.4: Truth Table for the 8286 Buffer.

$\overline{OE}$	T	INPUT	OUTPUT
1	X	Tri-state	Tri-state
0	0	B	A
0	1	A	B

2.2.1.1 Normal Access - No Potential Deadlock

Shown in figure 2.10 is a detailed block diagram of the Homogeneous Multiprocessor to aid the reader with the following discussion.

As part of the Interbus Switch, there must be buffers on the local bus of a Processing Element so that the Processor can eliminate itself from the bus in the case of potential deadlock. The 8286 Bidirectional Bus Buffers are connected to the processor on the *B side* while the Memory Module and Interbus are connected to the *A side*. Given this notation we can design the controller taking into consideration the direction of information depending on whether it is a read or write cycle. Note that when the buffer is tri-stated, direction (T) is of no consequence. The buffers are divided into two groups, one used for address and control lines and the other for the data lines. The address and control lines need never change direction depending on the type of transaction, they only have to be tri-stated while the neighbour uses the memory. From this we can deduce that the direction control for address and control lines (T_{AC}) can be pulled high (+5 volts), and signals coming out of the processor will be connected to the *A side* while signals coming into the processor will be connected to the *B side*. The control line for the

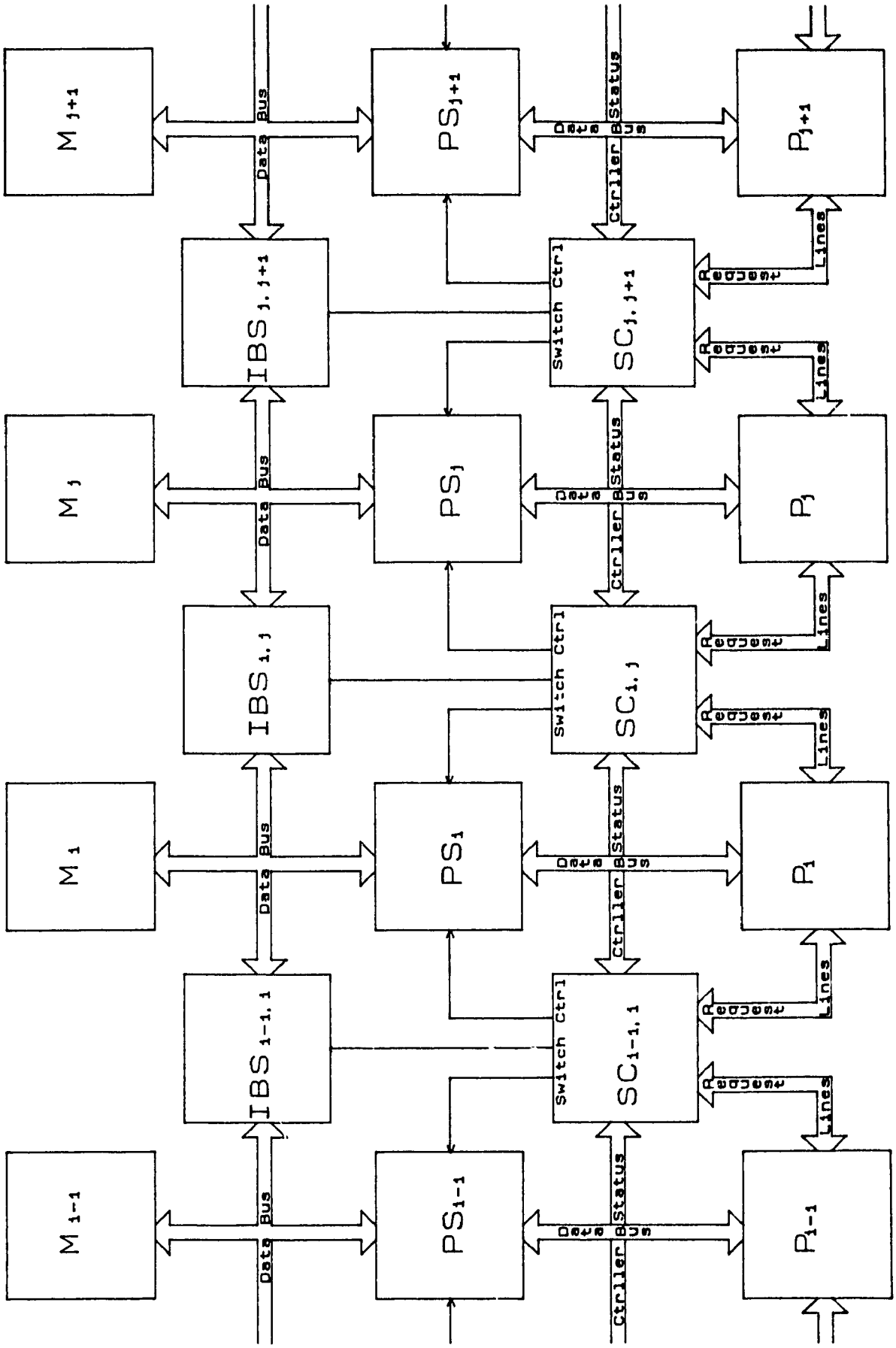


Figure 2.10: A Detailed Block Diagram of the Homogeneous Multiprocessor.

data bus (T_D), has to be able to change directions depending on whether it is a read cycle, ($A \rightarrow B$), or a write cycle ($B \rightarrow A$). This can be easily implemented by connecting T_D to the inverted $R/\overline{W}$ of the processor. Finally, both the address and control buffers and the data buffers will all be enabled or disabled together. One enable signal can be used to control all of the buffers ($\overline{OE}_{A,C,D}$). If the output enable is high, then the buffers are tri-stated and the neighbour can obtain the bus, if it is low, then the processor can access its own memory module. If a neighbouring processor is using the memory then the $\overline{BGACK}$ is low. By inverting the $\overline{BGACK}$, we obtain the control signal for $\overline{OE}$. Shown below are the equations.

$$T_{A,C}PS_i = 1 \tag{2.11}$$

$$T_DPS_i = \overline{R/W} \tag{2.12}$$

$$\overline{OE}_{A,C,D}PS_i = \overline{\overline{BGACK}} \tag{2.13}$$

For the Interbus Switch, for two adjacent processors, the 8286 buffers are configured with the A bus connected to the processor on the right, P_j, and the B bus connected to the processor on the left, P_i. Now, when the processor on the left requests the memory module on the right, the Bus Grant Acknowledge signal that enters the processor on the right is denoted as $\overline{BGACK}_{i,j}$ In the reverse situation when the processor on the right requests the memory on its left, the $\overline{BGACK}_{j,i}$ signal enters the processor on the left. Based on these two signals, we can determine which direction the buffers should *point*. Referring to table 2.5 below, we can determine when the switch should be closed.

Table 2.5: Truth Table for Closing the Interbus Switch ($\overline{OE}_{A,C,D}$).

$\overline{BGACK}_{i,j}$	$\overline{BGACK}_{j,i}$	Interbus Switch
0	0	X (Impossible)
0	1	0 (Closed)
1	0	0 (Closed)
1	1	1 (Open)

If we set the impossible state (0,0) to output 1 (Open) which will be safe, then it is obvious that an XOR gate can be used. The next signal that is developed is the direction for the address and control ($T_{A,C}$). Again it is possible to derive this signal from the two

Bus Grant Acknowledge signals. The truth table for the operation of $T_{A,C}$ is shown in table 2.6.

Table 2.6: Truth Table for the Directional Signal $T_{A,C}$.

$\overline{BGACK_{I,J}}$	$\overline{BGACK_{J,I}}$	Direction
0	0	X (Impossible)
0	1	0 (I → J)
1	0	1 (J → I)
1	1	X (Switch Open)

By setting the impossible state (0,0) to output 0, and the Switch Open state (1,1) to output 1, again it is obvious that the direction control for address and control is simply $\overline{BGACK_{I,J}}$. Finally, the last signal to derive for normal memory requests without deadlocks is the direction control for the data buffers in the Interbus Switch. In this derivation, the $R/\overline{W}$ of the processors is required to determine the direction. If the processor on the left of the Interbus Switch requests a read cycle with the memory module of the processor on the right, then the data in the buffers will flow from bus A to bus B, or T_D=1. For a similar write cycle the data must flow from bus B to bus A, or T_D=0. This is equivalent to the $R/\overline{W}$ signal generated by the processor on the left. In the reverse situation when the processor on the right requests a write cycle with the memory module of the processor on the left, the data will flow from the A bus to the B bus (T_D=0) and for a write cycle, T_D=1. This is equivalent to the inverted $R/\overline{W}$ signal obtained from the processor on the right. Shown in table 2.7, the truth table for the direction control of the buffers in the Interbus Switch.

Table 2.7: Truth Table for T_D in the Interbus Switch.

$\overline{BGACK_{I,J}}$	$\overline{BGACK_{J,I}}$	Direction
0	0	X (Impossible)
0	1	$R/\overline{W}_I$
1	0	$\overline{R/\overline{W}_J}$
1	1	X (Switch Open)

Now, all of the equations for the Interbus Switch are given below.

$$T_{A,C}IBS_{I,J} = \overline{BGACK_{I,J}} \tag{2.14}$$

$$T_D IBS_{I,J} = \overline{BGACK}_{J,I} \cdot R/\overline{W}_I + \overline{BGACK}_{I,J} \cdot \overline{R/W}_J \quad (2.15)$$

$$\overline{OE}_{A,C,D} IBS_{I,J} = \overline{BGACK}_{I,J} \cdot \overline{BGACK}_{J,I} \quad (2.16)$$

2.2.1.2 Potential Deadlock

In this section, the control signals for the Processor Switch and the Interbus Switch are redesigned for use only in the three cases of potential deadlock. This design will be accomplished so that there are three sub-outputs one for each case of potential deadlock that will be used to control the switches. It should be noted that in these designs we cannot use the $\overline{BG}$ or $\overline{BGACK}$ signals, because during potential deadlock, these signals are always inactive (high). We are therefore limited to using the Logical Closure signal LC, and the $\overline{BR}$ signals.

In the first case of Potential Deadlock, we find two neighbouring processors attempting to access each others memory modules. We see that the processor on the left P_I requesting the memory module on the right M_J asserts its $Request_{I,J}$. Note that this request consists of placing a specific address on the address bus. The processor on the right P_J, asserts its $Request_{J,I}$. Eventually, the Logical Closure appears for the switch in between these two adjacent processors. In this case, we want to force the Processor Switch on the left (PS_I) processor to open, and the Processor Switch on the right processor (PS_J) and the Interbus Switch ($IBS_{I,J}$) to close. Shown in table 2.8, the truth table for determining the first case of potential deadlock.

Table 2.8: Case I of Potential Deadlock

$Request_{I,J}$	$Request_{J,I}$	LC	Case
0	0	0	No request, No LC
0	0	1	Impossible
0	1	0	J→I, No LC
0	1	1	J→I with LC
1	0	0	I→J, No LC
1	0	1	I→J with LC
1	1	0	I→J, J→I, No LC
1	1	1	Case I, I→J, J→I with LC

From the table above, we see the potential deadlock at the last entry. Both proces-

sors have asserted their Requests and the switch is logically closed. This leads to the following equation for Case I of Potential Deadlock.

$$I = Request_{I,J} \cdot Request_{J,I} \cdot LC \qquad (2.17)$$

In the next case of potential deadlock, we find two adjacent processors attempting access the memory modules of their neighbours on the left. Of the two processors asserting the requests, we find the processor on the right requesting the memory module of the processor on the left, and the processor on the left requesting the memory module of the processor on its left. We will label the request of the processor on the right as $Request_{J,I}$ and the request of the processor on the left as $Request_{I,I-1}$. In the end, Logical Closure will appear on the switch between processors P_J and P_I, while Logical Closure will not appear on the switch between processors P_I and P_{I-1}. The Interbus Switch located between processors P_I and P_J ($IBS_{I,J}$) will be forced to close granting priority to the processor on the right, P_J. This can be clearly shown by referring to table 2.9.

Table 2.9: Case II of Potential Deadlock

$Request_{I,I-1}$	$Request_{J,I}$	$LC_{I-1,I}$	$LC_{I,J}$	Case
0	0	0	0	No Request
0	0	0	1	I→J, No Significance
0	0	1	0	I-1→I, No Significance
0	0	1	1	Impossible
0	1	0	0	J→I wait for LC
0	1	0	1	J→I
0	1	1	0	I-1→I, No Significance
0	1	1	1	Impossible
1	0	0	0	I→I-1 wait for LC
1	0	0	1	Impossible
1	0	1	0	I→I-1
1	0	1	1	Impossible
1	1	0	0	I→I-1, J→I, wait for LC
1	1	0	1	Case II, I→I-1 No LC, J→I with LC
1	1	1	0	I→I-1
1	1	1	1	Impossible

Referring to entry number fourteen in the above table, we find the second case of potential deadlock. The deadlock can be summarized by the following equation.

$$\Pi = \text{Request}_{i,i-1} \cdot \text{Request}_{j,i} \cdot \overline{LC}_{i-1,i} \cdot LC_{ij} \tag{2.18}$$

In the third and final case of potential deadlock, we encounter the following. A processor P_i requests the memory module of the neighbour on its right M_j. It eventually obtains the Logical Closure between the processors P_i and P_j. But before the Bus Request $\overline{BR}_{i,j}$ becomes active, the processor on the right P_j performs a request the neighbour on its right, M_{j+1}. The processor on the left, P_i never receives the $\overline{BG}$ from the processor P_j (because it is in the middle of a cycle) who will never obtain the Logical Closure because the switch on its left is already Logically Closed. The situation is resolved by forcing the processor P_j *off the bus* and granting priority to P_i. Refer to table 2.10 for clarification.

Table 2.10: Case III of Potential Deadlock

$\text{Request}_{i,j}$	$\text{Request}_{j,j+1}$	$LC_{i,j}$	$LC_{j,j+1}$	Case
0	0	0	0	No Request
0	0	0	1	J+1→J, No Significance
0	0	1	0	J→I, No Significance
0	0	1	1	Impossible
0	1	0	0	J→J+1 wait for LC
0	1	0	1	J→J+1
0	1	1	0	Impossible
0	1	1	1	Impossible
1	0	0	0	I→J wait for LC
1	0	0	1	J+1→J, No Significance
1	0	1	0	I→J
1	0	1	1	Impossible
1	1	0	0	I→J, J→J+1 wait for LC
1	1	0	1	J→J+1
1	1	1	0	Case III, I→J with LC, J→J+1 no LC
1	1	1	1	Impossible

Referring to the fifteenth entry in the table above, we find the third case of Potential Deadlock. The following equation describes the scenario.

$$III = Request_{I,J} \cdot Request_{J,J+1} \cdot LC_{I,J} \cdot \overline{LC}_{J,J+1} \tag{2.19}$$

Now there exist three equations to describe the three cases of Potential Deadlock. These three equations are then used to determine which switches are to be opened or closed, and the direction of flow of information in these switches. By logically ORing the three equations we obtain a signal that will tell us if there is a case of Potential Deadlock. If the signal is high, then a case exists.

$$PD = I + II + III \tag{2.20}$$

There are regularities in the three cases of Potential Deadlock. In case I and case II, the processor on the right, P_J, is given priority for a request. In the third case, it is the processor on the left, P_I, that is given priority. Shown in table 2.11 below, the state of the switches is given depending on the case of Potential Deadlock.

Table 2.11: State of the Switches During Potential Deadlock

Case	PS_I	$IBS_{I,J}$	PS_J
I	Open	Closed	Closed
II	Open	Closed	Closed
III	Closed	Closed	Open

From the three equations that describe the three cases of Potential Deadlock, we need to determine appropriate control signals for the three switches affected by the Interbus Switch Controller. Referring to equation 2.20, a truth table can be set up to help decode the information to control the switches as shown in table 2.12. Keep in mind that a logical 1 for the cases (I, II, III) reflects the existence of the potential deadlock and a logical 0 for a switch implies the switch is closed.

For the Interbus Switch $IBS_{I,J}$, we must also determine the directional control for the address, control, and data lines. By examining table 2.13 we can find the solution. Please keep in mind that a logical 0 for the directional control T refers to information flowing from the processor on the left to the processor on the right.

Table 2.12: Switch Control Information

Inputs			Outputs			
I	II	III	PS_I	$IBS_{I,J}$	PS_J	Comment
0	0	0	0	1	0	Normal, No Request
			0	0	1	I→J, No Deadlock
			1	0	0	J→I, No Deadlock
0	0	1	0	0	1	Case III
0	1	0	1	0	0	Case II
0	1	1	X	X	X	Impossible
1	0	0	1	0	0	Case I
1	0	1	X	X	X	Impossible
1	1	0	X	X	X	Impossible
1	1	1	X	X	X	Impossible

From the table above, the following equations can be clearly obtained.

$$\overline{OE}_{A,C,D}PS_I = III \tag{2.21}$$

$$\overline{OE}_{A,C,D}IBS_{I,J} = I \cdot II \cdot III \tag{2.22}$$

$$\overline{OE}_{A,C,D}PS_J = I \cdot II \tag{2.23}$$

Table 2.13: Directional Control for the Interbus Switch

Inputs				Outputs		
PS_I	$R/\overline{W}_I$	PS_J	$R/\overline{W}_J$	$T_{A,C}$	T_D	Comment
0	0	0	0	X	X	Impossible
0	0	0	1	X	X	Impossible
0	0	1	0	0	0	I→J, Write
0	0	1	1	0	0	I→J, Write
0	1	0	0	X	X	Impossible
0	1	0	1	X	X	Impossible
0	1	1	0	0	1	I→J, Read
0	1	1	1	0	1	I→J, Read
1	0	0	0	0	1	J→I, Write
1	0	0	1	1	1	J→I, Read
1	0	1	0	X	X	No request
1	0	1	1	X	X	No request
1	1	0	0	1	1	J→I, Write
1	1	0	1	1	0	J→I, Read
1	1	1	0	X	X	No request
1	1	1	1	X	X	No Request

By taking into consideration the *don't care* terms, we obtain the following equations for the Interbus Switch.

$$T_{A,C} = PS_i \tag{2.24}$$

$$T_D = \overline{PS_i} \cdot R/\overline{W_i} + \overline{PS_j} \cdot \overline{R/\overline{W_j}} \tag{2.25}$$

$$\overline{OE}_{A,C,D} = I + II + III \tag{2.26}$$

It is imperative that one keeps in mind that a Processor Switch that is referred to as PS_i for one Interbus Switch Controller, is referred to as PS_j by the Interbus Switch Controller on the immediate left. The same holds true for a Processor Switch that is referred to as PS_j. The Interbus Switch Controller on the immediate right will refer to this Processor Switch as PS_i. This must be taken into consideration during the design of the hardware required to control the Processor Switches.

The main signal that states whether or not there exists a case of Potential Deadlock is used to control the state of the Interbus Switch, $IBS_{i,j}$. If this signal is high, there is no Potential Deadlock, and the Interbus Switch is open. If the signal is low, then Potential Deadlock exists, and the switch is closed. By logically ANDing terms I and II, a control for the state of the Processor Switch on the right, PS_j, is obtained. The last term, III, is used to control the Processor Switch on the left, PS_i.

The neighbouring Interbus Switch Controller on the left, will also want to control the Processor Switch on its right, denoted as PS_j, which is denoted as PS_i by the Interbus Switch Controller on the right. More specifically, on either side of a processor and its Processor Switch, there is an Interbus Switch Controller. These two adjacent Interbus Switch Controllers must *work* together to control the Processor Switch that is between them. The Interbus Switch Controller on the right of the Processor Switch will refer to the Processor Switch as PS_i while the Interbus Switch Controller on the left of the Processor Switch will refer to the Processor Switch as PS_j.

The solution to this problem is very simple, the Interbus Switch Controller on the left ($IBS_{i-1,i}$) of the processor in question (P_i) will output a signal $PS_iIBS_{i-1,i}$, denoted as the Processor Switch control line for the processor on the right generated by the Interbus Switch Controller located between processors P_{i-1} and P_i. In other words, the processor

on the right denoted as P_i will be partially controlled by the Interbus Switch Controller on its immediate left denoted as $IBS_{i-1,i}$ which generated the control signal $PS_iIBS_{i-1,i}$. The Interbus Switch Controller on the immediate right, $IBS_{i,j}$ will output a control signal to the processor on its left, denoted as $PS_iIBS_{i,j}$ (but the same one as described above). These two control signals will simply be logically ANDed to produce the Processor Switch control line.

$$\overline{OE}_i = PS_iIBS_{i-1,i} \cdot PS_iIBS_{i,j} \tag{2.27}$$

The Processor Switch address, control and data lines direction control signal will be those described in the previous section. There need not be any control of direction of this information because these signals are derived directly from the processor irrespective of actual requests or pending requests. Finally, these are the equations that describe the operation of the switches during a Potential Deadlock situation.

For the Processor Switch we have obtained the following:

$$T_{A,C}PS_i = 1 \tag{2.28}$$

$$T_DPS_i = \overline{R/\overline{W}} \tag{2.29}$$

In reference to the Interbus Switch controller on the right of this switch.

$$\overline{OE}_{A,C,D}PS_i = PS_iIBS_{i-1,i} \cdot PS_iIBS_{i,j} \tag{2.30}$$

For the Interbus Switch:

$$T_{A,C}IBS_{i,j} = PS_i \tag{2.31}$$

$$T_DIBS_{i,j} = \overline{PS}_i \cdot R/\overline{W}_i + \overline{PS}_j \cdot \overline{R/\overline{W}}_j \tag{2.32}$$

$$\overline{OE}_{A,C,D}IBS_{i,j} = I + II + III \tag{2.33}$$

2.2.1.3 Amalgamation - Normal Access Including Potential Deadlock

In this final section of chapter 2, we will amalgamate the equations obtained in the previous two sections. The switches must operate properly during all types of neighbouring accesses. We will start off by completing the Processor Switch. In the previous section we have determined that the directional control lines for the Processor Switch are the following.

$$T_{A,C}PS_i = 1 \tag{2.34}$$

$$T_DPS_i = \overline{R/W} \tag{2.35}$$

The last signal for the Processor Switch is the $\overline{OE}_{A,C,D}$. This control signal will be a combination of the two individual signals derived in the two previous sections. One of them handles normal requests while the other handles Potential Deadlock scenarios. Shown in table 2.14 below, the truth table for the Processor Switch.

Table 2.14: Truth Table for the Processor Switch

Inputs		Outputs
Potential Deadlock	Normal	$\overline{OE}_{A,C,D}$
$PS_iIBS_{i-1,i} \cdot PS_iIBS_{i,j}$		
0	0	X (Impossible)
0	1	0 (Potential Deadlock)
1	0	0 (Normal Access)
1	1	1 (No Request)

It is obvious from the table above that the two signals can be logically ANDed to produce the final output.

$$\overline{OE}_{A,C,D}PS_i = PS_iIBS_{i-1,i} \cdot PS_iIBS_{i,j} \cdot \overline{BGACK} \tag{2.36}$$

The next three equations will describe the control for the Interbus Switch. Using equations 2.14 and 2.24 we can produce the following table 2.15 to form $T_{A,C}$.

Table 2.15: Truth Table for Address and Control Signals of the Interbus Switch

Inputs		Outputs
Potential Deadlock	Normal	$T_{A,C}$
PS_i	$\overline{BGACK}_{i,j}$	
0	0	X (Impossible)
0	1	0 (Potential Deadlock)
1	0	0 (Normal I→J)
1	1	1 (No Request or Normal J→I)

This is simply the logical AND of the two signals. Equation 2.36 will control the direction of address and control signals through the Interbus Switch.

$$T_{A,C}IBS_{i,j} = PS_i \cdot \overline{BGACK}_{i,j} \tag{2.37}$$

The next signal to be produced is T_D, the directional control signal for the data bus. Table 2.16 is produced from equations 2.15 and 2.25.

Table 2.16: Truth Table for Data Bus Direction Through the Interbus Switch

Inputs				Outputs
PS_I	$\overline{BGACK_{I,J}}$	PS_J	$\overline{BGACK_{J,I}}$	T_D
0	0	0	0	X (Impossible)
0	0	0	1	X (Impossible)
0	0	1	0	X (Impossible)
0	0	1	1	X (Impossible)
0	1	0	0	X (Impossible)
0	1	0	1	X (Impossible)
0	1	1	0	X (Impossible)
0	1	1	1	$R/\overline{W}_I$ (Case III)
1	0	0	0	X (Impossible)
1	0	0	1	X (Impossible)
1	0	1	0	X (Impossible)
1	0	1	1	$R/\overline{W}_I$ (I→J)
1	1	0	0	X (Impossible)
1	1	0	1	$\overline{R/\overline{W}_J}$ (Case I or II)
1	1	1	0	$\overline{R/\overline{W}_J}$ (J→I)
1	1	1	1	X (No request)

From the above table, we can deduce the following formula for the directional control for the data bus in the Interbus Switch.

$$T_D IBS_{I,J} = \overline{(\overline{BGACK_{I,J}} \cdot PS_I)} \cdot R/\overline{W}_I + \overline{(\overline{BGACK_{J,I}} \cdot PS_J)} \cdot \overline{R/\overline{W}_J} \qquad (2.38)$$

The final control signal to be determined for the Interbus Switch is the $\overline{OE}_{A,C,D}IBS_{I,J}$. By combining equations 2.16 and 2.22 we can produce a control signal for the state of Interbus Switch that will handle both normal requests and potential deadlock situations. Referring to table 2.17 this can be done.

Table 2.17: Truth Table for the State of the Interbus Switch

Inputs		Outputs
Potential Deadlock	Normal	$\overline{OE}_{A,C,D}IBS_{I,J}$
$\overline{OE}_{A,C,D}IBS_{I,J}$	$\overline{OE}_{A,C,D}$	
0	0	X (Impossible)
0	1	0 (Potential Deadlock)
1	0	0 (Normal Request)
1	1	1 (No Request)

In the above table, by setting the *don't care* state to 0, we can clearly see that the two control lines should be logically ANDed to produce the final control signal for the state of the Interbus Switch.

$$\overline{OE}_{A,C,D}IBS_{I,J} = \overline{BGACK}_{I,J} \cdot \overline{BGACK}_{J,I} \cdot (I + II + III) \tag{2.39}$$

Finally, the hardware implementation for Phase II, including Phase I, the handshaking controllers, and the combinational logic required to control the 8286 buffers that are used as switches is shown in figure 2.11.

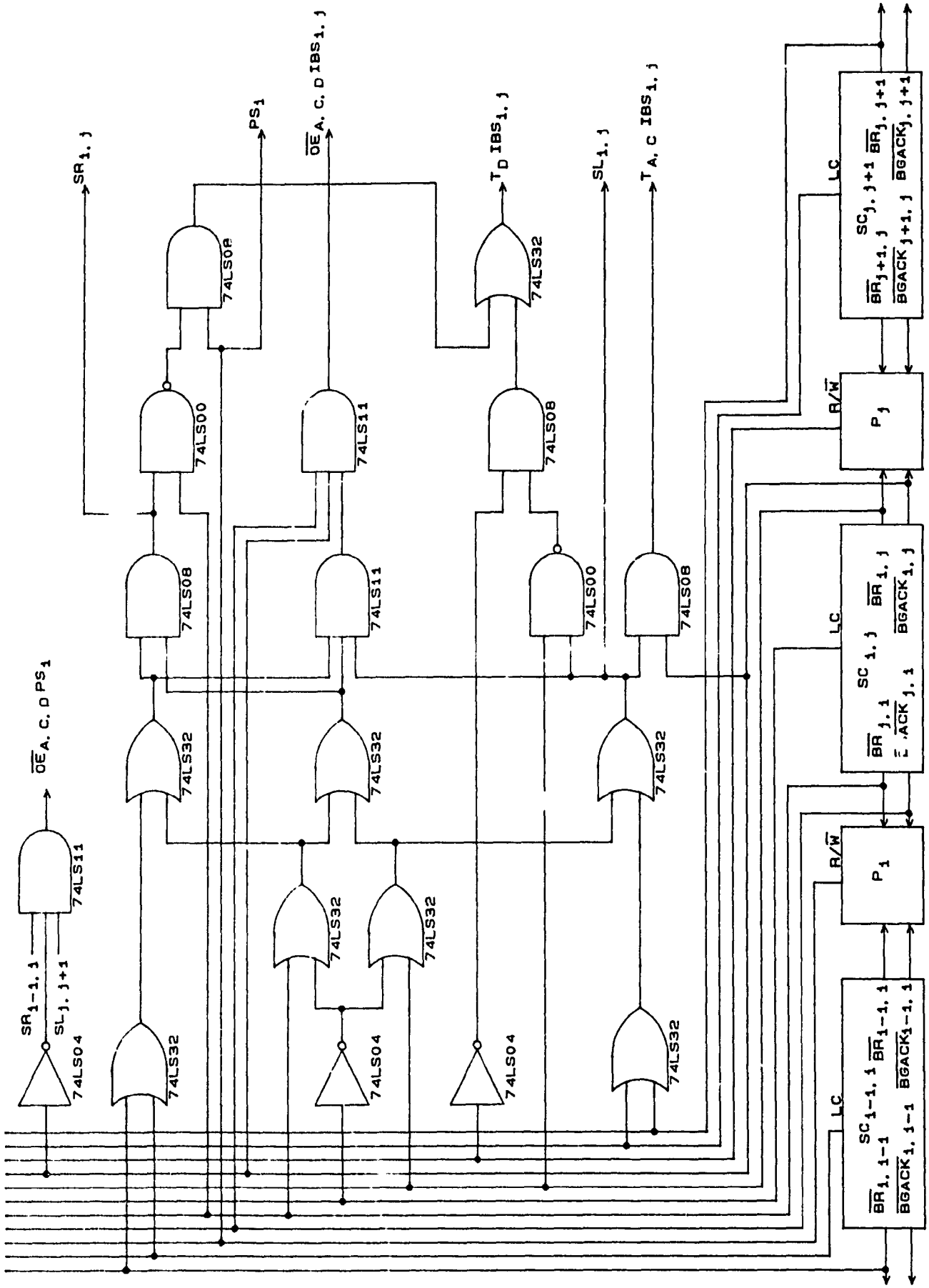


Figure 2.11: Hardware Implementation of Phase II.

2.3 Experimental Results

The Interbus Switch Controller has not been built on the Versabus board as of yet. However, some testing has been done on the subsystem. Phase I and Phase II have been tested on a bread board and they operate according to specification. Neither of the circuits have been tested for maximum speed of operation. Based upon the information obtained from the TTL Data Book, we estimate that Phase I can operate at the desired 24MHz to produce a Logical Closure in one CPU cycle (running at 8MHz). For Phase II, it should operate at no less than 8MHz, so that it can change state as fast as the CPU will.

3. Conclusion and Future Work

In this thesis, we have presented three different subsystem designs for the Homogeneous Multiprocessor Proper. The first being a 1 Megabyte dynamic RAM memory system, the second, a 1 Megabyte static RAM memory system, and finally, a design for the Interbus Switch Controller which facilitates communication between adjacent Processing Elements.

The dynamic memory system was the first of the memory modules that have been designed. It contains an Error Detection and Correction Unit that is used to boost the reliability of the overall system. The bulk of the integrated circuits used in the design part of a *chip set* manufactured by Advanced Micro Devices. This allows straight forward implementation of the system except for a few 74LSXX series chips. Dynamic memory was implemented as opposed to static due to the high price of static memory circuits. The average memory access cycle time is 700 nS.

The second memory module that was designed used static memory integrated circuits. Error detection and correction is not required in a static memory design because the integrated circuits are not susceptible to alpha particles. At the time of this design, the price of the static memory chips was considerably reduced since the dynamic memory design. The cycle time for memory access is considerably less than the dynamic design due to the lack of an error detection and correction unit. The average memory access cycle time is 300 nS.

Finally, the last design was the Interbus Switch Controller. This design is fairly complex due to large number of control signals required by neighbouring processors. Currently, the design is implemented with standard 74LSXX series integrated circuits. This facilitates a slower design than can be achieved through other methods. In addition, there are several areas in the designs that can be carried out in the future:

I. Implementation in VLSI

Various parts of the Homogeneous Multiprocessor Proper can be implemented in VLSI. Much of the decoding circuitry in the processor can be fabricated in silicon. The Interbus Switch Controller and its switches can also be manufactured. Finally the decoding circuitry in the static memory design can be implemented in VLSI. This would considerably reduce the area required by the various systems incorporated in the Homogeneous Multiprocessor thereby reducing the cost of implementation, debugging time, and layout. This would allow more circuitry per board thereby reducing the number of boards required. Currently, we have estimated that two boards will be required for each Processing Element. It is not unforseeable to reduce this number to one by using VLSI.

II. Upgrading with Technology

As time passes, the cost of new products such as larger dynamic memory integrated circuits will decrease, such as the 1 Megabit memory. Once a controller becomes available, the dynamic memory system can be upgraded using less chips to obtain a similar design. The same applies for the static memory design. Currently, there are larger static memories that are contained in a Single In-line Package (SIP) that are 256 Kilobytes. These memories contain 8 surface mount 32 Kilobyte memories and some additional hardware. Unfortunately they are extremely expensive. Once they become reasonable priced, it is feasible to have a 1 Megabyte static memory system that only requires 20 to 30 square centimeters of area. This would greatly reduce the current requirements of a much larger area.

III. Printed Circuit Boards

Currently, the Processing Elements and associated hardware are implemented on wire wrap boards. These boards have various restrictions, the two most prominent one being speed of operation of the circuits and the other is the inability to put certain types of chip packages on the boards. One example in the Motorola MC68020 which comes in

a square leadless package. Other circuits which cannot be used are surface mount. If printed circuit boards are implemented, then there will be no restrictions as to package types.

References

1 Advanced Micro Devices, *Bipolar Microprocessor Logic and Interface Data Book*, pp. 4.1 - 4.120, 1985.

2 Crowther, W., et. al., The Butterfly Parallel Processor, *Computer Architecture Technical Committee Newsletter*, Sept./Dec. 1985.

3 Desrochers, G.R., *Principles of Parallel and Multiprocessing* pp. 266 - 269, 276 - 280, 294 - 295, Intertext Publications, Inc., McGraw-Hill Book Co., 1987.

4 Dimopoulos, N.J., On the Structure of the Homogenous Multiprocessor, *IEEE Transactions on Computers*, Vol. C-34, No. 2, pp. 141 - 150, February 1985.

5 Dimopoulos, N.J. and K.F. Li, The Performance Analysis of the Homogenous Multiprocessor Proper., *Canadian Journal of Electrical Engineering*, pp. 3 - 10, January 1987.

6 Evans, D.J. (Editor), *Parallel Processing Systems An Advanced Course*, p. 89, Cambridge University Press, 1982.

7 Gottlieb, A., et. al., The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer., *IEEE Transactions on Computers*, Vol. C-32, No. 2, pp 175 - 189, February 1983.

8 Hamming, R.W. *Coding and Information Theory*, pp. 39 - 42, Prentice-Hall, 1980.

9 Hitachi, *The Memory Data Book*, 1988.

10 Hwang, K. and F.A. Briggs, *Computer Architecture and Parallel Processing* pp. 20 - 27, 40 - 49, 463, 645 - 659, McGraw Hill Inc. 1984.

11 Li, K.F., The Homogeneous Multiprocessor: A Simulation Study and an Operating System Design, Ph. D. Dissertion, Concordia University, 1987.

12 Miklosko, J. and V.E. Kotov (Editors), *Algorithms, Software and Hardware of Parallel Computers*, pp. 300 - 302, Veda Publishing House, 1984.

13 Mitsubishi, *The Memory Data Book*, 1985.

14 Motorola Inc., *The MC68000 Educational Computer Board User's Manual*, Austin, Texas, 1982.

15 Motorola Inc., *The Motorola Microprocessor Data Manual*, Austin, Texas, 1981.

16 Peterson, W.W. and E.J. Weldon Jr., *Error-Correcting Codes*, pp. 2 - 3, The MIT Press, 1972.

17 Segal, T.L., The TLS-2 Switch Controller, Technical Report, Concordia University, Montreal, Canada, 1986.

18 Seitz, C.L., The Cosmic Cube, *Communications of the ACM*, Vol. 28, No. 1, pp. 22 - 33, January 1985.

19 Swan, R.J., et. al., Cm* – A Modular Multimicroprocessor, *Proceedings AFIPS Conference 1977*, Vol. 46, pp. 645-655, 1977.

20 Texas Instruments, *The TTL Data Book for Design Engineers*, 1981.

21 Wulf, W.A., et. al., C.mmp – An Experimental Computer System, McGraw-Hill, New York, 1981.

Appendix A

A. The Dynamic Memory Subsystem

In this section, the dynamic memory that has been designed and built for the Homogeneous Multiprocessor will be discussed. The dynamic memory sub-system is composed of various integrated circuits from Advanced Micro Devices which are special controllers for the sole purpose of controlling dynamic memories. The dynamic memory integrated circuits are manufactured by Mitsubishi and their size is 256 Kilobits. The dynamic memory system is 1 Megabyte and incorporates error detection and correction. Also made by AMD is an asynchronous bus interface circuit which has been configured and used with the Motorola MC68000.

A.1 General Principles of Dynamic RAMs

Until very recently, the cost of producing a dynamic memory system was much lower compared to a static memory system. It has always been a fact that dynamic RAMs have been manufactured with higher densities than static RAMs due to the smaller size of the individual memory cells. Higher densities imply savings in space which can be a critical factor during the design process. Currently, the 1 Megabit dynamic RAM is the largest integrated memory circuit that is currently (1987) manufactured in a DIP package. As far as the static RAMs are concerned, the largest circuit that can be manufactured in a DIP package is 32 Kilobytes (which is 256 Kilobits). Thus, with a 400 percent more capacity, the DRAM seems to be the best candidate for making large memory arrays. The block diagram of the Mitsubishi M5M4256S-15 dynamic RAM is shown in figure A.1 below.

The only undesirable features of dynamic RAMs are that they need to be refreshed and the possibility of errors caused by alpha particles. Refreshing requires external circuitry, and to reduce the possibilities of errors, error detection and correction or parity checking can be used.

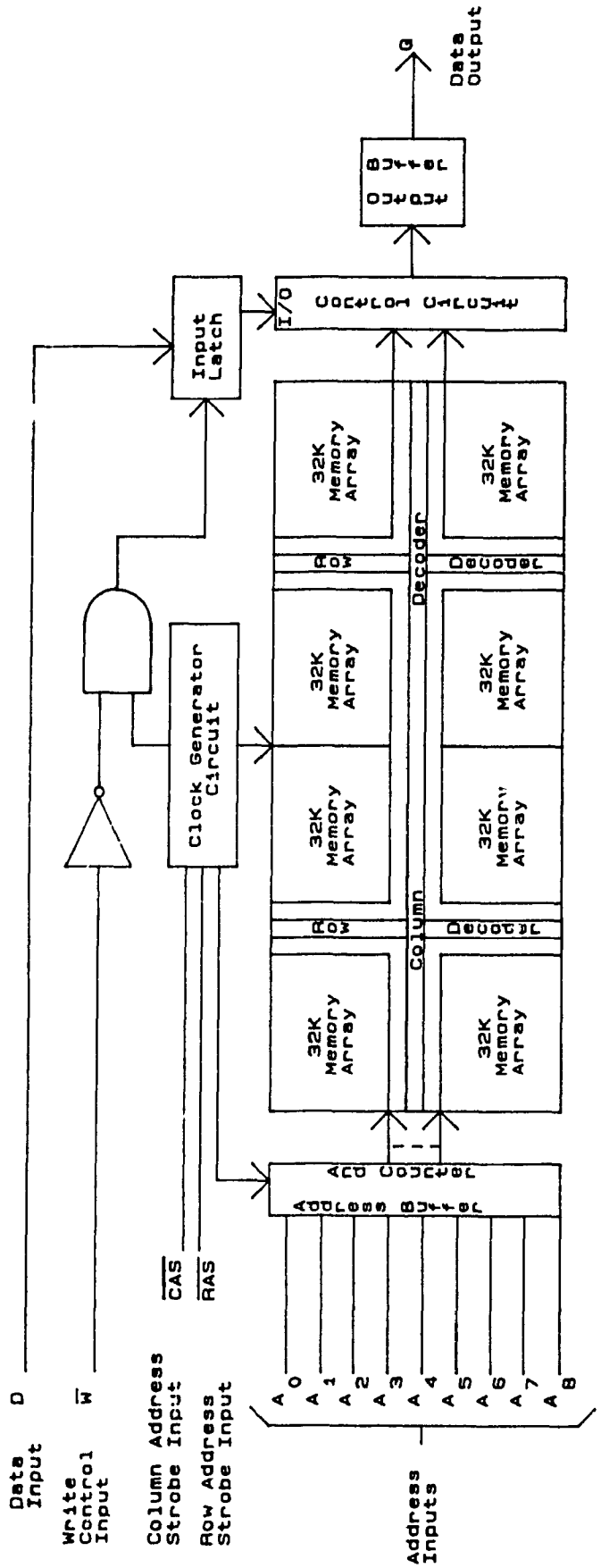


Figure A.1: The Mitsubishi M5M4256S-15 Dynamic RAM.[12]

In the case of refreshing, which must be done otherwise cells start loosing their information after 4 mS., a row of memory cells in the DRAM must be refreshed every 15.625 μS, and the entire chip must be refreshed every 4.0 mS.[12]

The dynamic memory cells in an integrated memory circuit are organized in the form of a matrix. In the case of larger memory circuits, the actual array is composed of many smaller *matrices*, usually more than are required, the reason being that if one of the *matrices* is flawed, the integrated circuit does not have to be discarded. Even though the dynamic RAM consists of more than one matrix, as far as the user is concerned, it is organized as a single matrix with *rows* and *columns*. When accessing a memory cell in a dynamic RAM, a *row address* and a *column address* are generated and presented to the dynamic RAM in a multiplexed fashion. In a 256 Kilobit memory array, 18 address lines are required, however, in order to save space, these are multiplexed over the same pins as row and column addresses. The row address and column address strobes are used to latch the row and column addresses internally. These are then used to select the corresponding row and column from the matrix of RAM cells; their intersection defining the currently active (read/written) cell. The row and column address strobes are commonly denoted by $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ respectively.

In the process of a read or write operation, the following must occur. For a read operation, the output enable is activated, then the row address and the row address strobe ($\overline{\text{RAS}}$) are presented to the dynamic RAM, on the rising edge of $\overline{\text{RAS}}$, the address is latched. Then the column address is presented with the column address strobe ($\overline{\text{CAS}}$); on the rising edge of $\overline{\text{CAS}}$, the address is latched, and the data appears on the Data Output line (DO) (after a $\overline{\text{CAS}} \rightarrow$ DO delay).

For a write operation, the data is first presented to the Data Input latch, (DI) while keeping the Output Enable ($\overline{\text{OE}}$) inactive for the entire cycle. Then, the address is presented to the dynamic RAM in the same manner as described above; on the rising edge of $\overline{\text{CAS}}$ the data is latched and stored in the dynamic RAM.

A.2 Error Detection and Correction Requirements

Reliable DRAM memory systems are very important for the proper operation of any computer system. With every advance in computer technology, the number of bytes of memory is increasing with the same rate as the density of the MOS DRAMs is increasing. Because of the reduced size of the memory cells and the smaller stored charge of the cell in the 256 Kilobit DRAM, the chips are more sensitive to alpha particles which can destroy the information in the cells. By using an error detection and correction unit, dramatic increases in memory reliability are achieved.

Referring to figure A.2, a simple storage system is shown. In this system, there is a source of information, an encoder which transforms the source information into information acceptable to the storage medium, a decoder which is used to transform the information into a form acceptable to the sink, and the sink which is the entity requiring the stored information. In such a system, erroneous data may occur while the information resides in the storage medium.[16]

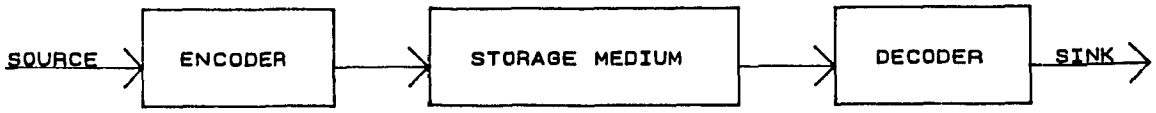


Figure A.2: A Simple Storage System.[16]

If we modify the storage system in the above system to include error detection and correction, it becomes more complicated as depicted in figure A.3. Again we have the source and sink, and there are various encoders and decoders along the data path. There are two encoders along the data path entering the storage medium, the first one is used to transform the source information into binary, the second encoder is used to incorporate the additional information required for error detection and correction (the check bits). The last encoder is the same as in the previous diagram, it is used to transform the information into a form acceptable to the storage medium.

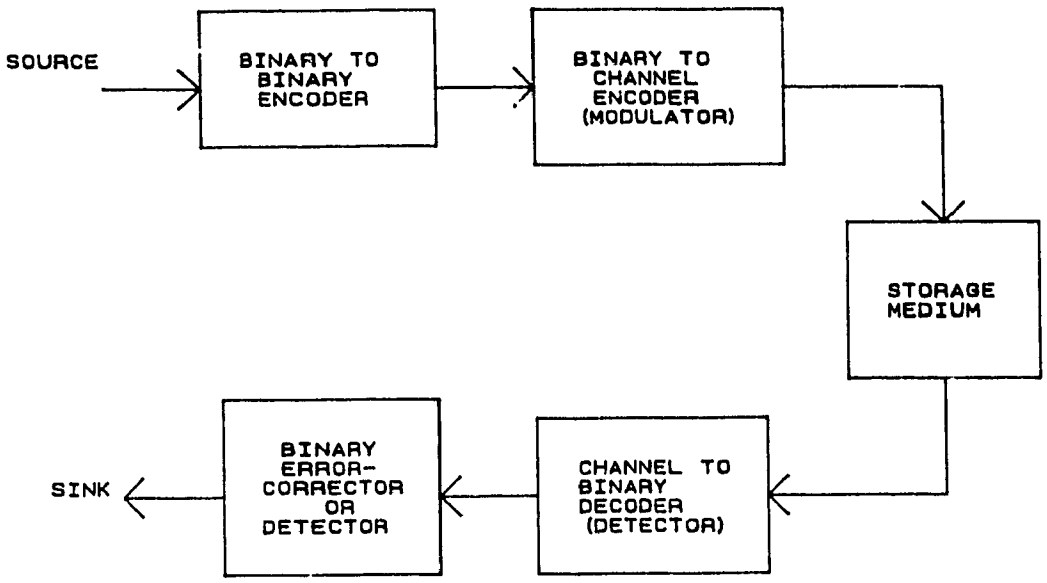


Figure A.3: A Typical Data Storage System Employing Error Detection and Correction.[16]

Along the output stream of the storage medium, the data first encounters a decoder that transforms the data to binary. Then the data is passed through the error detection and correction unit which verifies the information and finally, the decoder that prepares the data for the sink.

Hamming codes are widely used for error detection and correction in memory systems. The following is a simple example of how Hamming codes can be used for error detection and correction.[8] This example will use a 4 bit data word. To begin with, we will require m parity checks for the n bits in the message. It should be noted that n is the total comprising of the original number of data bits (4) and the number of parity check bits (m). Inequality A.1 below, provides a relation of the number of check bits in terms of total number of bits in the message for a single bit error correction code.

$$2^m \geq n + 1 \qquad \text{(A.1)}$$

It can be determined that the number of parity checks required is, m=3. Including the 4 data bits, a total of 7 bits are required for a single bit error detection and correc-

tion. By examining the binary representations of the message bits, it can be determined which data bits will be included in which parity bits. The message bits can be represented as N_2, N_1, and N_0. For example, the first check bit, denoted as C_0 will be produced by the combination of all the message bits that contain a "1" as their N_0. C_1 is produced from the message bits that contain a "1" as their N_1, and finally, C_2 is produced from the bits that contain a "1" as their N_2 bit.

Table A.1: Check Positions.

Position Number	Binary Representation			Included in Parity Check Bit
	N_2	N_1	N_0	
1	0	0	1	1
2	0	1	0	2
3	0	1	1	1,2
4	1	0	0	3
5	1	0	1	1,3
6	1	1	0	2,3
7	1	1	1	1,2,3

Thus we see that the first parity check covers positions 1,3,5,7. The second covers 2,3,6,7. And the third covers 4,5,6,7. By examining the binary representation of the message bits, it is observed that positions 1,2, and 4 contain only one "1". These locations will be used to store the parity check bits. The other locations, 3,5,6, and 7 will be used to store the four data bits. Note that since the message bits in locations 1,2, and 4 are used to store the check bits, they are not taken into consideration during the creation of the parity check bits. They are only used during the detection process to determine if an error occurred.

Note that a parity check will do the following: if there are an even number of 1's in the specified locations, then the parity check will be a "0"; If there is an odd number of 1's in the specified locations, then the parity check will be a "1". Table A.2 will show the flow of events that occurs through the system.

When an error occurs in the message, it is located by performing a new set of parity checks. During this second set of parity checks, if the results are all "0", then there is

no error. However, if there is at least one "1" as a result, then an error exists and it must be corrected. The results obtained from the parity checks, when combined will produce a binary number called the syndrome which is the position number of the bit in error. To correct a bit in error, it is simply complemented.

Table A.2: Encoding a 4-Bit Message and Locating the Error.[16]

ENCODE DATA							
Positions	1	2	3	4	5	6	7
Message	-	-	1	-	0	1	1
Encode	0	1	1	0	0	1	1
Error			X				
Receive	0	1	0	0	0	1	1
LOCATE THE ERROR							
Check 1:		1 3 5 7					
		0 0 0 1			fails → 1		
Check 2:		2 3 6 7					
		1 0 1 1			fails → 1		
Check 3:		4 5 6 7					
		0 0 1 1			correct → 0		
syndrome =		0 1 1 =			3 → position of error		
CORRECT							
Correct error			1				
Corrected message	0	1	1	0	0	1	1

A.3 Error Detection and Correction

In a word oriented memory system such as the one incorporated in the Homogeneous Multiprocessor Proper, the 16 bit format consists of 16 data bits and 6 check bits.

There are two basic modes of operation of the Am2960 Error Detection and Correction Unit. The first is the Generate Mode, which is used during a write cycle and the second is the Detect and Correct Mode, which is used during a read cycle.

During a write cycle, the generate mode will check the contents of the Data Input Latch (which contains the data to be written to memory), and generate 6 check bits according to a modified Hamming code. In general, each check bit is generated as either an exclusive OR or an exclusive NOR function consisting of 8 of the 16 data bits. The XOR and XNOR functions result in even or odd parity checks, respectively. Shown in table A 1 is the check bit encoding chart. Once the check bits are generated, they are stored in the check bit memory at the same time as the data bits are stored in the data memory.

During a read cycle, the data bits are verified correct, and if an error is found, the system will attempt to correct it. This is accomplished by reading the data and check bits from memory, from the data bits, a new set of check bits is created by the same method as during a write cycle. Then the newly created check bits and the check bits retrieved from the check-bit memory are compared. If the two sets are identical, then the retrieved data are correct. If they differ, one of four types of errors may occur. (1) the first and most obvious type is a single bit error in the 16 bit data. If this occurs, then the incorrect bit can be corrected by complementing it; At the same time, the corrected data is rewritten to the memory.

The second type of error which has no effect on the data is a check bit error; this can be corrected by complementing and rewriting the check bits to the memory.

The last two types of errors concern multiple errors. Such errors cannot be

corrected. All double bit errors can be detected, while only some of triple bit errors are detected. For such multiple errors there is not enough information in the check bits to discern which bits are in error, therefore the system is unable to make the appropriate corrections. An interrupt is generated (the $\overline{\text{MULT ERROR}}$) which invokes the appropriate handler.

During check bit generation, in our case, 6 check bits are produced, each corresponding to 8 data bits, of which each data bit contributes in three check bits. For example, data bit D_0 contributes in check bits C_0, C_1, and C_2. At the same time that the data bits are written to the memory, the check bits are also written to the check bit memory. During retrieval of the data, a new set of check bits are produced. If by chance, D_0 gets corrupted, then the newly generated set of check bits will have different values for check bits C_0, C_1, and C_2. The newly created check bits will be compared to the original set by exclusive ORing the two sets, the results of this comparison is called the syndrome bits. In this example, the syndrome bits S_0, S_1, and S_2 will have the value 1, while the other syndrome bits will all be 0, the syndrome decoder will then determine that data bit D_0 is in error and it will complement it. The complete set of syndrome bits and their meaning is shown in table A.3 below. If the corresponding definition for a syndrome number is either C_i or D_i, then this implies that either the check bit C_i or the data bit D_i is in error and should be complemented for correctness. If the definition is double, then there exists a double error, and two bits are in error and no correction can be made. As well, the definition $\geq$ triple refers to three or more bits in error and again no correction can be made.

If only one of the syndrome bits has a value of one, then this implies that the corresponding check bit is in error and should be corrected by complementing it and rewriting it to memory.

If any other combination of syndrome bits are set, this implies that a multiple error has been detected and cannot be corrected.

Table A.3: Check Bit Definition.

$S_XS_0S_1S_2S_4S_8$	Definition	$S_XS_0S_1S_2S_4S_8$	Definition	$S_XS_0S_1S_2S_4S_8$	Definition
0 0 0 0 0 0	No Error	0 1 0 1 1 0	D_6	1 0 1 0 1 1	Double
0 0 0 0 0 1	C_8	0 1 0 1 1 1	Double	1 0 1 1 0 0	$\geq$ Triple
0 0 0 0 1 0	C_4	0 1 1 0 0 0	Double	1 0 1 1 0 1	Double
0 0 0 0 1 1	Double	0 1 1 0 0 1	D_{10}	1 0 1 1 1 0	Double
0 0 0 1 0 0	C_2	0 1 1 0 1 0	D_4	1 0 1 1 1 1	$\geq$ Triple
0 0 0 1 0 1	Double	0 1 1 0 1 1	Double	1 1 0 0 0 0	Double
0 0 0 1 1 0	Double	0 1 1 1 0 0	D_0	1 1 0 0 0 1	D_3
0 0 0 1 1 1	$\geq$ Triple	0 1 1 1 0 1	Double	1 1 0 0 1 0	D_2
0 0 1 0 0 0	C_1	0 1 1 1 1 0	Double	1 1 0 0 1 1	Double
0 0 1 0 0 1	Double	0 1 1 1 1 1	$\geq$ Triple	1 1 0 1 0 0	D_1
0 0 1 0 1 0	Double	1 0 0 0 0 0	C_X	1 1 0 1 0 1	Double
0 0 1 0 1 1	D_{15}	1 0 0 0 0 1	Double	1 1 0 1 1 0	Double
0 0 1 1 0 0	Double	1 0 0 0 1 0	Double	1 1 0 1 1 1	$\geq$ Triple
0 0 1 1 0 1	D_{13}	1 0 0 0 1 1	D_{14}	1 1 1 0 0 0	$\geq$ Triple
0 0 1 1 1 0	D_7	1 0 0 1 0 0	Double	1 1 1 0 0 1	Double
0 0 1 1 1 1	Double	1 0 0 1 0 1	D_{11}	1 1 1 0 1 0	Double
0 1 0 0 0 0	C_0	1 0 0 1 1 0	D_5	1 1 1 0 1 1	$\geq$ Triple
0 1 0 0 0 1	Double	1 0 0 1 1 1	Double	1 1 1 1 0 0	Double
0 1 0 0 1 0	Double	1 0 1 0 0 0	Double	1 1 1 1 0 1	$\geq$ Triple
0 1 0 0 1 1	$\geq$ Triple	1 0 1 0 0 1	D_9	1 1 1 1 1 0	$\geq$ Triple
0 1 0 1 0 0	Double	1 0 1 0 1 0	D_3	1 1 1 1 1 1	Double
0 1 0 1 0 1	D_{12}				

A.4 Incorporation of the Error Detection and Correction Unit

In the design of the dynamic memory system, the requirements were as follows. We desired a system that could handle at least 1 Megabyte of dynamic memory incorporating 256 Kilobit dynamic RAM chips. We also required integrated circuits that are manufactured in DIP packages since our prototypes are wire wrapped boards and they do not support surface mount or pin grid arrays. It was imperative that our dynamic memory system have error detection and correction. We also preferred having a family of memory support products whereby system integration could be done with hardware kept to a minimum. Our final requirement was that these memory support products should be easily integrated into an existing MC68000 based processor and that they operate asynchronously.

As it turned out, only the family of DRAM support devices produced by Advanced Micro Devices (AMD) met our specifications.[1]

The Am2968 Dynamic Memory Controller is capable of handling 1 Megabyte of memory composed of 256 Kilobit memory chips. It is capable of driving 88 DRAMs without any external drivers, and, like all of the AMD integrated circuits, is available in DIP packaging.

The Am2960 Error Detection and Correction Unit provides us with error detection and correction. It is also capable of handling both byte and word oriented read and write operations

The Am8167 is an interface chip that is used for timing all the events that occur in the system, it also provides the refresh timers required by the DRAMs. It is the interface for the Error Detection and Correction Unit, the Dynamic Memory Controller and the Motorola MC68000.

The AMD memory support line also contains one other special function integrated circuit that provides the data path to and from the EDC, DRAMs, and system bus, the Am2962.

Finally, the implementation of this system would require a minimal amount of decoding and support logic. The integrated circuits that are supplied by AMD are from a "chip set". All of the chips that are used are compatible (with the exception of the Am8167 System and Timing Controller).

A.5 Selection of Components for the Dynamic Memory Subsystem

The AMD chips are from the Am2960-70 Memory Support Family. The chips that are incorporated in this design consist of the: Am2960 Error Detection and Correction Unit; Am2962 EDC Bus Buffers; Am2968 Dynamic Memory Controller.

The data interface between the dynamic memories, the Am2960 EDC chip, and the system data bus is accomplished with the Am2962 bus buffers. Shown in figure A.4 is the block diagram of the memory system.

The Am2962 contains two internal latches, a multiplexer, and a RAM driver output

buffer (note that the Am2962 has noninverting buffers). Each of these devices has a 4-bit wide data bus to and from the dynamic memories, the EDC and the system data bus. For a 16 bit data bus, four of the Am2962 Bus Buffers are required. The Bus Input latch is used to latch data during byte write operations. An incoming byte from the system bus is stored in the Bus Input Latch while the memory is being read, and any necessary corrections are being made in the byte not being changed. The Bus Output latch is used when reading from the memory, it latches the data from the memory until the processor reads it, and, at the same time, the memory can be refreshed.

The Am2960 Error Detection and Correction Unit contains all the logic necessary to generate check bits on a 16-bit data bus using the modified Hamming code and to correct the data word by using the check bits. The Am2960 can correct all single bit errors, detect all double bit errors, and detect some triple bit errors. According to the modified Hamming code, for a 16 bit data bus, an additional six bits are required producing a total of 22 bits.

The Am2968 Dynamic Memory Controller is capable of supplying the Row address Strobe, $\overline{RAS}$ and the Column Address Strobe, $\overline{CAS}$ for up to four banks of 256 Kilobit memories making a total memory size of 1 Megabyte. The Am2968 can drive up to 88 DRAM chips without external drivers.

The Am8167 System and Timing Controller is an integral part of the dynamic memory system. The Am8167 is used in an Am2964 Dynamic Memory Controller based design to interface this controller to an MC68000 processor. Nevertheless, we were able to adapt this particular device to our design that uses the Am2968 controller (since no comparable device for the Am2968 existed). Our design can be found at the end of the appendix.

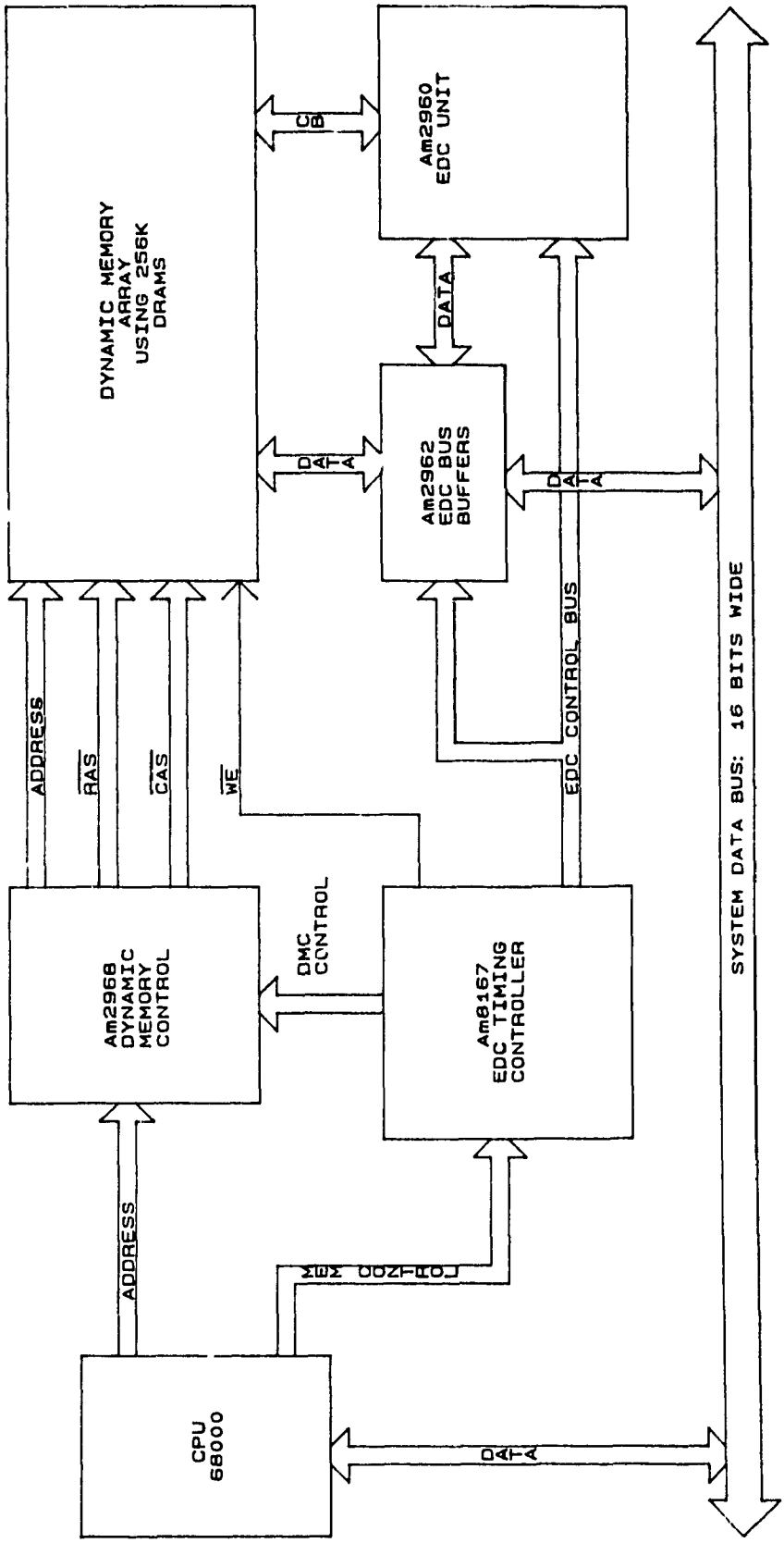


Figure A.4: The Dynamic Memory Subsystem.

A.6 The Memory Subsystem Design

The Am2960 Error Detection and Correction Unit (EDC) contains all the logic necessary to generate check bits for a 16-bit data bus using a modified Hamming Code, and to correct a data word when check bits are present. The Am2960 will correct any single-bit error on a data read cycle and it is capable of detecting all double and some triple-bit errors. The Am2960 is expandable to operate on a 64-bit data bus but in our case the data field in only 16 bits wide therefore only requiring 6 check bits. Data error logging is possible using the error syndrome which is available to the user on a separate output bus.

A block diagram of the Am2960 Unit is shown in figure A.5. Referring to the block diagram, the identifiable parts include the data input latch, check bit input latch, check bit generation logic, syndrome generation logic, error detection logic, error correction logic, data output latch, diagnostic latch, and the control logic.

A.6.1 The Am2960 EDC Architecture

The **Data Input Latch** is used to store 16-bits of data from the bidirectional DATA lines under the control of the Latch Enable input, LE IN. The input data can be used for either check bit generation or error detection/ generation depending on the control mode.

The **Check Bit Input Latch** is used to store up to seven bits from the Check Bit memory under control of the Latch Enable input, LE IN. The Check Bits are used in the Error Detection and Error Correction modes to determine if there are any errors in the 16 bit data.

The **Check Bit Generation Logic** as its name implies, is used to generate the appropriate check bits for the 16-bits of data stored in the Data Input Latch. The Check Bits are generated according to a modified Hamming code which has been described in a previous section.

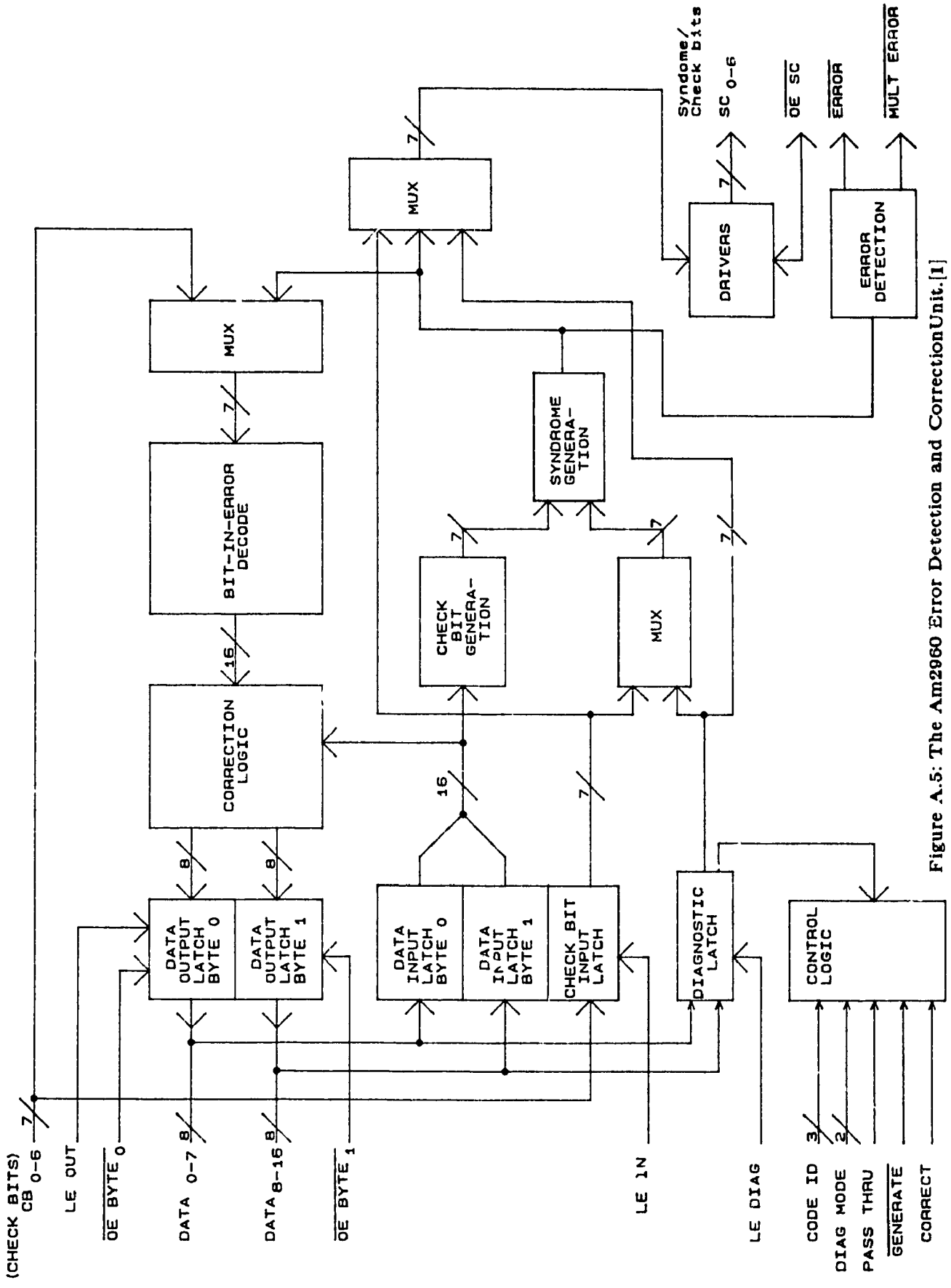


Figure A.5: The Am2960 Error Detection and CorrectionUnit.[1]

The **Syndrome Generation Logic** is used in the Error Detection and Error Correction modes. This hardware compares the check bits read from memory with a newly generated set which is produced from the 16-bit data read from memory, if both sets are identical then there are no errors. If the two sets are not identical, then one or more of the data or check bits is incorrect. The syndrome bits are produced by exclusive-ORing the two sets of check bits. The syndrome bits will be all zeros if the two sets are identical and thus, are no errors. If one or more of the syndrome bits are not zero, then they can be decoded to determine the number of errors and the bit(s) that are incorrect.

The **Error Detection Logic** decodes the syndrome bits generated by the Syndrome Generation Logic. The outputs $\overline{\text{ERROR}}$ and $\overline{\text{MULT ERROR}}$ will remain inactive if there are no errors. $\overline{\text{ERROR}}$ will become active if there are one or more errors, $\overline{\text{MULT ERROR}}$ will become active if there are two or more errors.

The **Error Correction Logic** can correct all single bit errors by complementing the bit that is incorrect. Once the error is corrected, the new data is presented to the Data Output Latch, and from there put on the system data bus.

The **Data Output Latch** is used to store a new result when an error correction is required. The LE OUT signal is used to load the latch the data so that from there, the data is presented to the system data bus. The Data Output Latch is split into two 8-bit latches which can be enabled independently, one for the upper byte and the other for the lower byte.

There are four basic modes of operation for the EDC Unit: Generate, Detect, Correct, and Pass Thru. Each of these modes of operation of the Error Detection and Correction Unit will be discussed in greater detail below.

The generate mode is activated when a write cycle to memory is required. Once the 16-bit data is loaded into the Data Input Latch, check bits are generated based on the

data, then the check bits are placed on the Check Bit bus for placement in the Check Bit memory (only 6 of the 7 check bits are used, the 7^{TH} bit is used for wider data words).

A modified Hamming Code is used to generate these Check Bits, was discussed in section A.2. Shown in table A.4 is the check bit encoding used by the system.

Table A.4: Check Bit Encoding.[1]

Generated Check Bits	Parity	Participating Data Bits															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CX	Even (XOR)		√	√	√		√			√	√		√			√	
C0	Even (XOR)	√	√	√		√		√		√		√		√			
C1	Odd (XNOR)	√			√	√			√		√	√			√		√
C2	Odd (XNOR)	√	√				√	√	√				√	√	√		
C4	Even (XOR)			√	√	√	√	√	√							√	√
C8	Even (XOR)									√	√	√	√	√	√	√	√

Note: The check bit generated as either an XOR or XNOR of the eight data bits noted by an "√" in the table.

The detect mode is active during a read cycle to determine if there are any errors in the data that is being read. The data and check bits are temporarily stored in the Data Input Latch and Check Bit Input Latch respectively. As stated previously, a newly generated set of check bits are generated from the data and compared to the check bits read in from the check bit memory. All single bit and double bit errors are detected while only some of the triple bit errors are detected.

During this process, syndrome bits are generated and placed on the syndrome output bus. They may be decoded to determine if there was a bit error detected. If the error is only a single bit, it is corrected, and then the data is output to the system data bus for the MC68000 to latch.

A.6.2 The Am2962 4-Bit Error Correction Multiple Bus Buffer

The Am2962 bus buffers are high performance, low power Schottky multiple bus buffers that provide a complete data path interface between the Am2960 EDC, the dynamic RAM, and the system data bus. In our system the data bus is 16 bits wide, therefore requiring 4 of these units. Each unit can handle 4 bits of data. By providing separate enables for groups of buffers, one may achieve memory access on nibbles, bytes or 16-bit words. In our design, we provide separate enables in groups of two, providing us with byte and word access. Shown in figure A.6 is the block diagram of the Am2962 Bus Buffer.

A.6.3 The Am2968 Dynamic Memory Controller

The Am2968 DMC is capable of supporting 256Kbit x 1 as well as smaller dynamic memories. The Am2968 uses two 9-bit address latches to hold the row and column addresses required for the DRAMs. There are also two refresh counters (row and column). The two upper address lines will inform the DMC which of the four banks of DRAM will be accessed during a cycle.

The Am2968 DMC has two modes of operation: refresh and read/write. In the refresh mode, the two 9-bit counters cycle through the refresh addresses in order to refresh the DRAMS. During the read/write mode, the address presented to the DMC from the CPU is passed to the address MUX and then split up according to the requirements of the DRAMs for a memory access. Refer to figure A.7 for the block diagram of the Am2968 DMC.

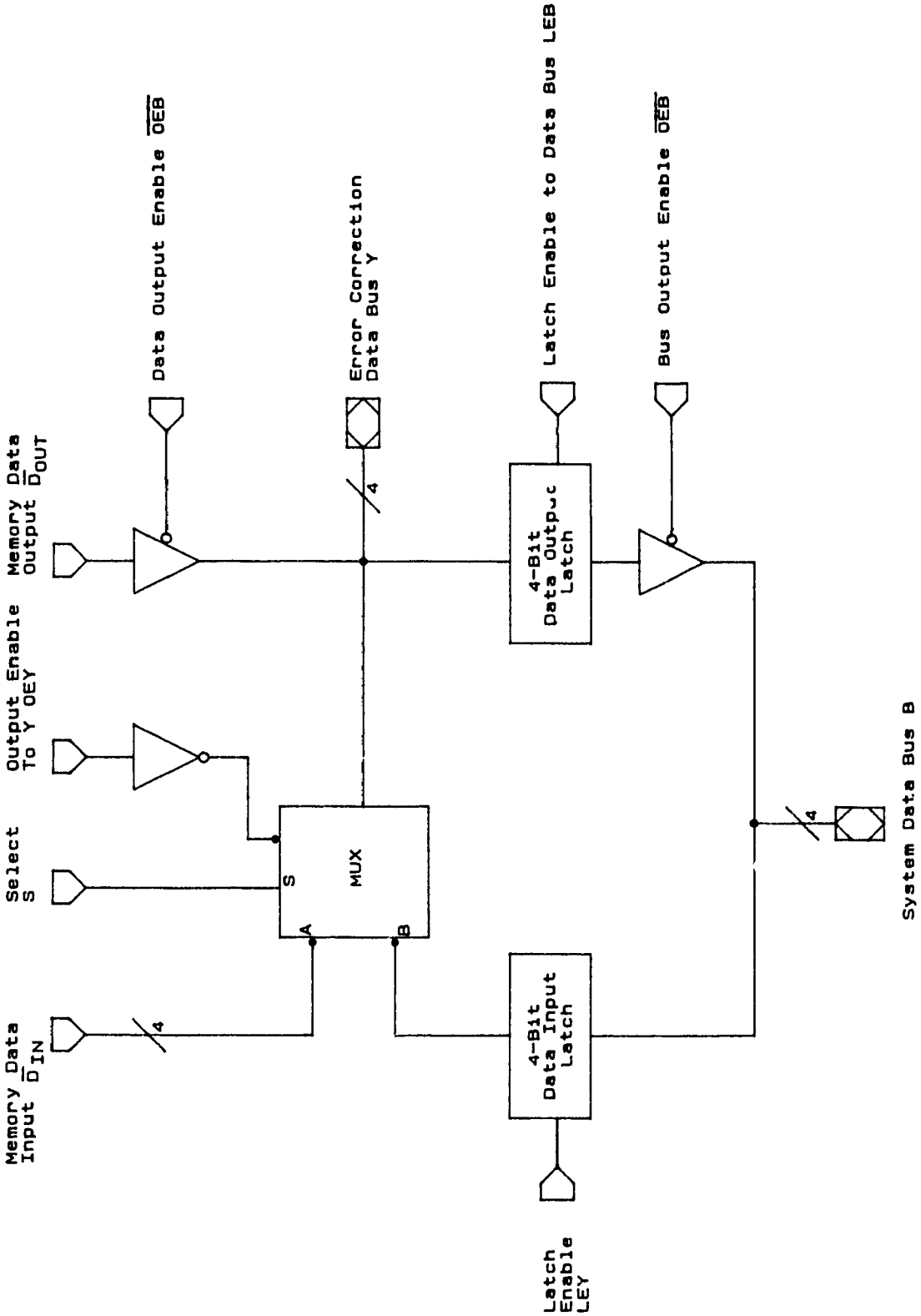


Figure A.6: The Am2962 Bus Buffer.[1]

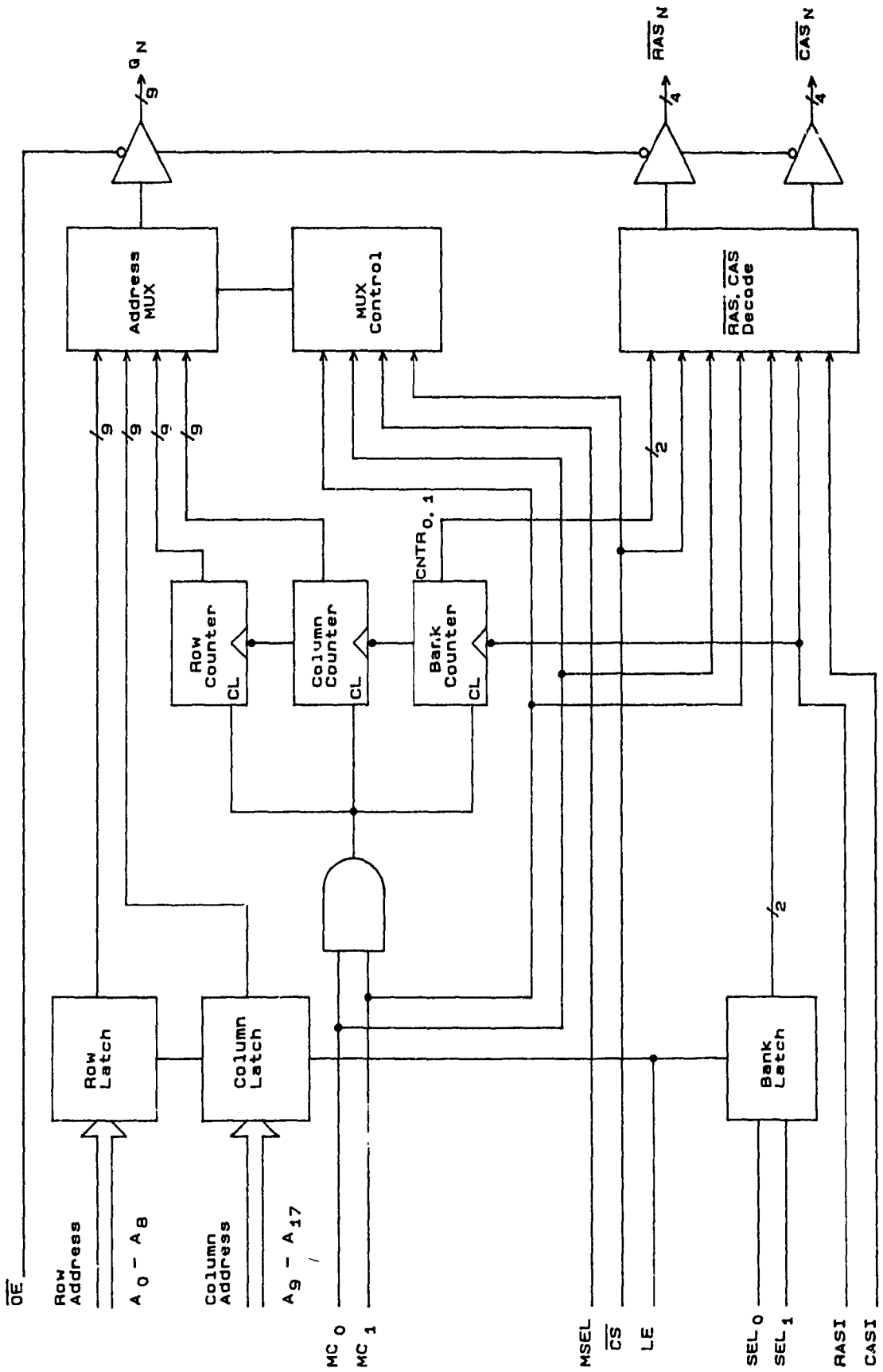


Figure A.7: The Am2968 Dynamic Memory Controller.[1]

A.6.4 The Am8167 Dynamic Memory Timing, Refresh and EDC Controller

The Am8167 is a high speed interface controller that can be used with a variety of microprocessors, one of which is the Motorola MC68000. The Am8167 provides all of the control interface functions including $\overline{RAS}$/Address-MUX/$\overline{CAS}$ timing (the specific timing required for the sequence of the Row Address Strobe, the Row Address, the Column Address, and the Column Address Strobe, respectively), memory request/refresh arbitration and all EDC (Am2960) enables and controls. It should be noted that the Am8167 is not directly compatible with the Am2968 Dynamic Memory Controller. In our design some special circuitry was added to enable proper operation of the system.

The Am8167 has several input control signals such as BYTE/$\overline{WORD}$, READ/$\overline{WRITE}$, Address Strobe, Data Strobe and a refresh clock. From these input, the Am8167 generates control signals for the Am2968 $\overline{RAS}$/ $\overline{CAS}$ and Refresh multiplexer. Table A.5 below shows the corresponding signals used from the MC68000 to the Am8167.

Table A.5: MC68000 to Am8167 Interface.

MC68000 Signal	Am8167 Signal
$\overline{UDS} + \overline{LDS}$	BYTE/$\overline{WORD}$
$\overline{UDS} \cdot \overline{LDS}$	$\overline{DS}$ (Data Strobe)
$\overline{UDS}$	A_0

A.6.4.1 Dynamic Memory Refresh

Refresh operations can be performed either by the CPU, known as transparent refresh, or by the memory controller known as stand-alone refresh. Our system uses stand-alone refresh which puts the responsibility of refresh address generation and timing on the memory controller. The Am8167 will perform the necessary timing and access arbitration. A refresh request is generated using the refresh timer on the Am8167. The refresh timer is known as RCLK and it is connected to a 16 MHz crystal. Every 16 clock pulses, a refresh request is generated for the Am2968 Dynamic Memory Controller to perform a refresh cycle.

Potential conflicts between refresh request and memory requests are resolved by the arbiter inside the Am8167. If a refresh request occurs during a memory operation or after a memory request, it will be honored after the completion of the memory transaction and the necessary precharge time has elapsed. If on the other hand, a memory request occurs during a refresh operation or after a refresh request, it will be acknowledged only after the refresh operation has been completed and the necessary precharge time has elapsed. Finally, if both a memory request and a refresh request occur at the same time, the arbiter will honor the memory request first.

A.6.4.2 Error Detection and Correction

Another function of the Am8167 is the timing and control required for the Error Detection and Correction. There are two methods of error correction that can be implemented, they are *Correct On Error Only* and *Correct Always*.

Correct On Error Only relies on the fact that error detection is faster than correction. At reasonably low rates of errors, this method achieves the highest possible throughput, however, it is incompatible with all present microprocessors because they sample their WAIT input too early in the cycle. This method will insert a wait state on a read cycle, in order for the error to be corrected and to be rewritten to the memory. The alternate method known as Correct Always allows time after each memory read to write the corrected data back to the memory. This is done only in the case of an error. If there is no error, then the time is wasted. Our system uses the Correct Always method due to the incompatibility of the Correct On Error Only method with the MC68000.

There are two error signals available, they are: $\overline{\text{ERROR}}$ and $\overline{\text{MULTIPLE ERROR}}$. As their names imply, if a single bit error occurs, the EDC will correct the error and processing will continue, however, if a multiple bit error occurs, then the error cannot be corrected therefore if processing were to continue, invalid information would be passed on to the processor. Processing must be stopped in order to invoke the appropriate action to be taken by the processor.

A.7 Flow Of Data Through The Memory Subsystem

In this section, the flow of data is discussed. In conjunction with the block diagram of the memory subsystem, shown in figure A.1, a detailed explanation can be given to show the operation and sequence of events that occur during a read or a write operation. In the following sections, a description of a read cycle and a write cycle are given. For the purpose of simplicity, we will skip over the operations involved in the interbus switch and assume that the CPU is directly connected to the memory subsystem. There are byte and word operations, there is no difference between a byte read and a word read. However, a word write is known as a write cycle and a byte write is known as a read-modify-write cycle which will also be described.

A.7.1 A Write Cycle

The first thing that happens during a write cycle is that the CPU outputs an address, an address strobe, a write signal, and the data to be written to that address (not necessarily in this order). The first chip to obtain some information is the Am8167 Interface Controller. This chip receives the address, address strobe, and write signal, it then sends out to the Am2962 Bus buffers a signal for them to latch the 16 bit data. The Bus Buffers will hold the data so that the Am2960 Error and Detection Unit can compute the Check Bits. After a predetermined amount of time, based on the memory subsystem clock, the Am8167 Interface Controller sends out the $\overline{RAS}/\overline{CAS}$, address and strobe and associated signals to the Bus Buffers and Error Detection and Correction unit so that they should each send their data to the respective memories (Bus Buffers to 16 bit RAM, and EDC to 6 bit check RAM). Again after a predetermined amount of time, the data is latched into the memories and the $\overline{DTACK}$ is sent to the CPU to inform it that the memory cycle is finished.

A.7.2 A Read Cycle

A read cycle is very similar to a write cycle. The basic difference is the direction of information flow. As with a write cycle, the CPU outputs an address, an address strobe

and a read signal. This information is passed to the Am8167 Interface Controller. The Interface Controller then sends this information to the Am2968 Dynamic Memory Controller and some signals to the Am2960 Error and Detection Unit and to the Am2962 Bus Buffers. The Dynamic Memory Controller sends out the $\overline{RAS}/\overline{CAS}$ and a, propriate address to the memory. The memory then outputs the 16 bit data which is then latched into the Am2962 Bus Buffers, and the 6 Check Bits are latched into the Error Detection and Correction Unit. The EDC then reads the data from the Bus Buffers, recreates the Check Bits, and finally compares the newly generated check bits with those obtained from memory. If they are the same, everything is correct and the EDC instructs the Bus Buffers to pass the data on to the CPU. Then it is up to the CPU ,if it wants to acquire a byte or the full 16-bit word. If there is a discrepancy between the check bits, the EDC has to do some work that will extend the memory cycle time. If there is only one check bit that is different then the EDC can correct the error and pass the corrected data on to the CPU. At the same time, it writes the corrected data back to the memory. However, if there are two or more check bits that are not the same, a multiple error has occurred and the EDC cannot correct the data. This causes the Memory Subsystem to generate an interrupt to inform the system of the error.

A.7.3 A Read-Modify-Write Cycle

A Read-Modify-Write cycle is used when the CPU want to write only one byte to the memory. Basically the sequence of steps needed is as follows: The memory system reads the 16 bit word that is stored in the memory location in question, verifies the check bits for a correct word, substitutes in the new byte, generates new check bits, and finally stores the new 16 bit word and Check Bits.

The flow of data is as follows. When the CPU generates the write cycle, along with the standard signals, it also generates a byte write signal. This is sent to the Am8167 with the other signals and then the Interface Controller coordinates the system to take the appropriate actions. First, the Interface controller instructs the Dynamic Memory

Controller to read the memory location and then it tells the Bus Buffers to latch the word. At the same time it instructs the Error Detection and Correction Unit to latch the check bits. The Interface Controller then instructs the EDC to verify the 16 word stored in Bus Buffers by regenerating check bits and comparing them with the ones it obtained from the Check Bit Memory. If the check bits are identical it then tells the appropriate Bus Buffers to substitute in the new byte. Then the EDC reads the new 16 bit word and again generates a new set of Check Bits. Finally it instructs the Dynamic Memory Controller to save this new word and its Check Bits. If there is a sigle bit error, it is corrected and the process continues. If there is a multiple error, an interrupt is generated to inform the system.

A.8 Implementation and Experimental Results

The dynamic memory subsystem is built on a Motorola VME wire wrap board. During the design procedure there were various initial restrictions made that had to be strictly followed. The processor section had already been implemented on the board and it utilized approximately 67%. This left a restriction on the amount of area that could be used for the memory since it had to exist on the same board as the processor. Many of the dynamic memory chip sets were available only in leadless chip carriers that cannot be placed on the wire wrap board used. Some of the chip sets were still in the preproduction phases and only their data sheets were available at the time. Our memory systems must be asynchronous which adds other restrictions. We finally were left with AMD to supply us with a chip set that was usable and meet our needs. It should be noted that even the AMD chip set had not yet been completed. The Am8167 Dynamic Memory Timing Controller which is incorporated in our system is not really appropriate. Additional hardware had to be incorporated in the design to ensure the proper operation of the dynamic memory. The appropriate controller was not available at the time, the Am2969 Dynamic Memory Timing Controller which is said to be 100% compatible with the Am2968 Dynamic Memory Controller was still in the preproduction phases and

would not be available for an undetermined amount of time. We were forced to use the non-100% compatible chip, the Am8167.

The design of the dynamic memory system, according to the data book, is quite simple. Many design application notes are incorporated in the data book. During the summer of 1985 the dynamic memory subsystem was built. There were various problems that were discovered when the system did not work. The most prominent problem to this day is decoupling of the memory chips. At the time of writing, the memory will operate correctly for a Block Fill operation but, when inputting assembly code (via the Memory Modify command), it operates erratically. Sometimes the instruction is stored, other times it is not. Initially 0.1uF capacitors were placed on the 256 Kilobit dynamic memory chips. When it was determined that the capacitance may not have been high enough, an additional 0.1uF was added for a total of 0.2uF. After the addition, the board still did not operate correctly 100% of the time. Finally one last test was done. An additional 0.33uF capacitance was added, the the board now operates as described above. We suspect that the $\overline{\text{UDS}}$ and $\overline{\text{LDS}}$ signals may be crosscoupled. This can explain the failure of the Memory Modify command which operates on bytes rather than on words.

Currently, the dynamic memory system comprises of 32 256 Kilobit dynamic memory chips to store the data and 12 256 Kilobit chips to store the error correction code. The other major chips are from the AMD chip set: the Am8167 Dynamic Memory Timing Controller; the Am2968 Dynamic Memory Controller; the Am2960 Error Detection and Correction Unit; and four of the Am2962 Multiple Bus Buffers. Finally there are a few TTL chips for interface purposes. The average memory access cycle time is 700 uS Refer to figure A.8 for the layout of the board. And finally in figure A.9, the complete schematic for the dynamic memory subsystem incorporating error detection and correction.

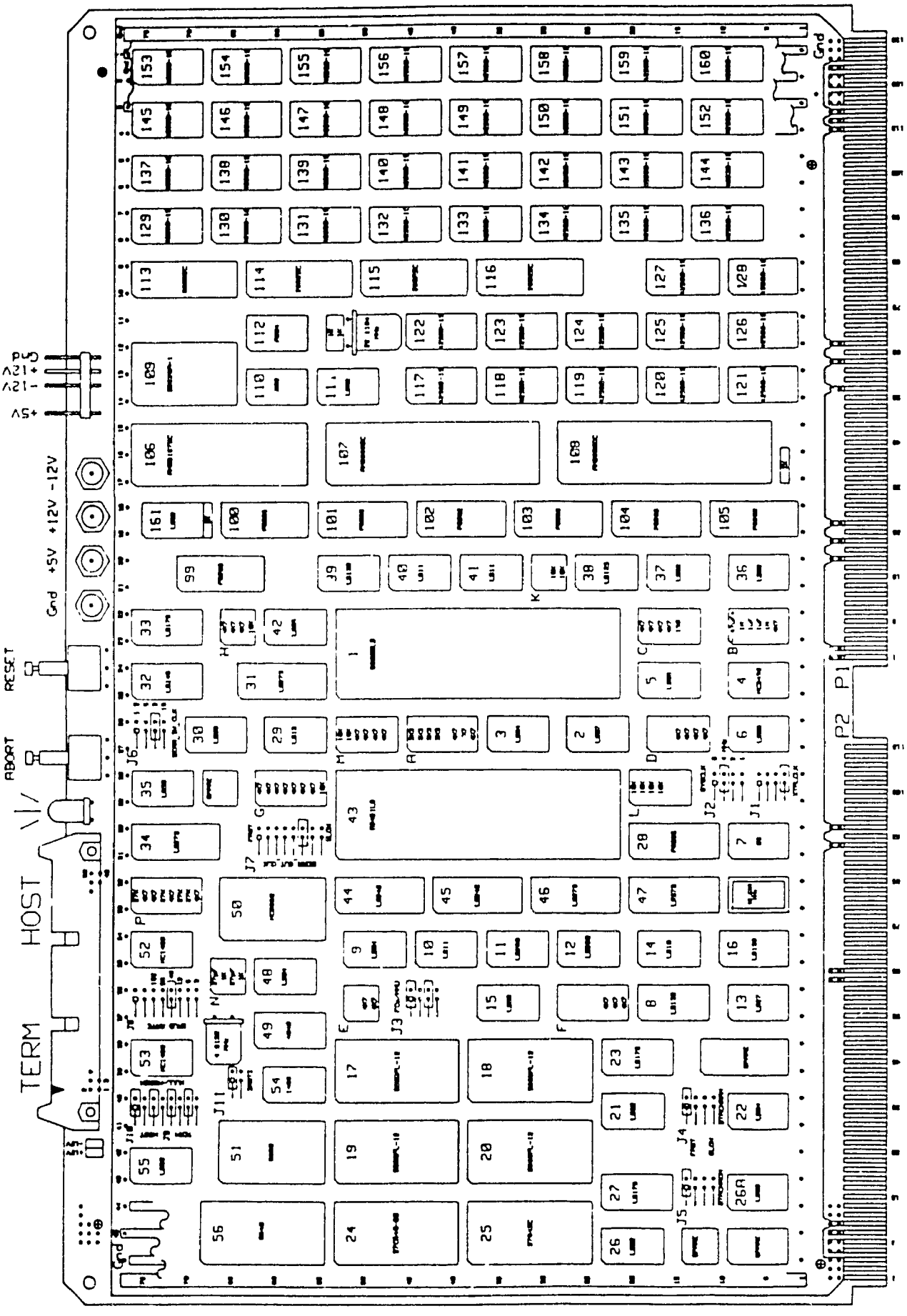


Figure A.8: The Layout of a Processor Node Incorporating the Dynamic Memory

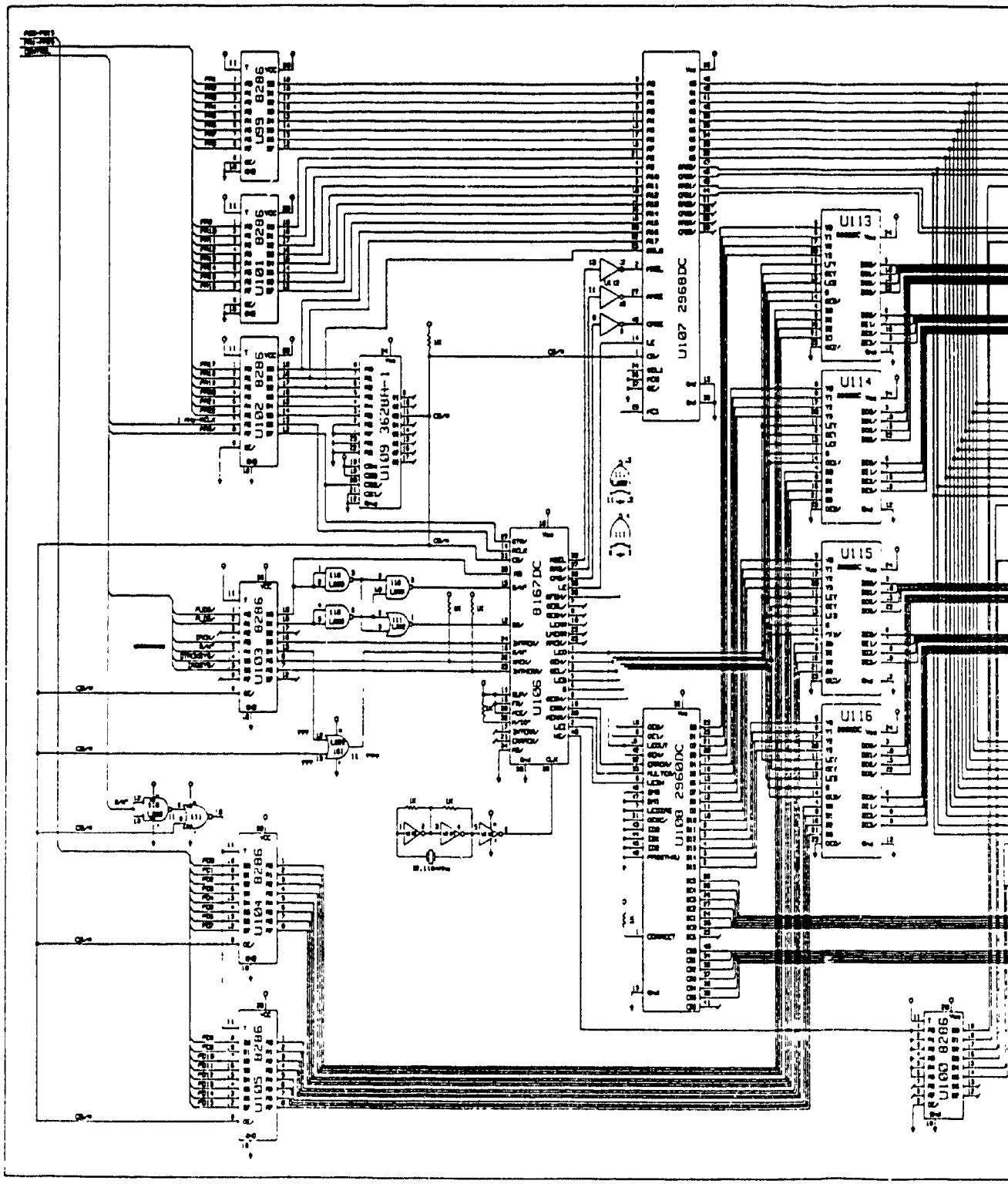
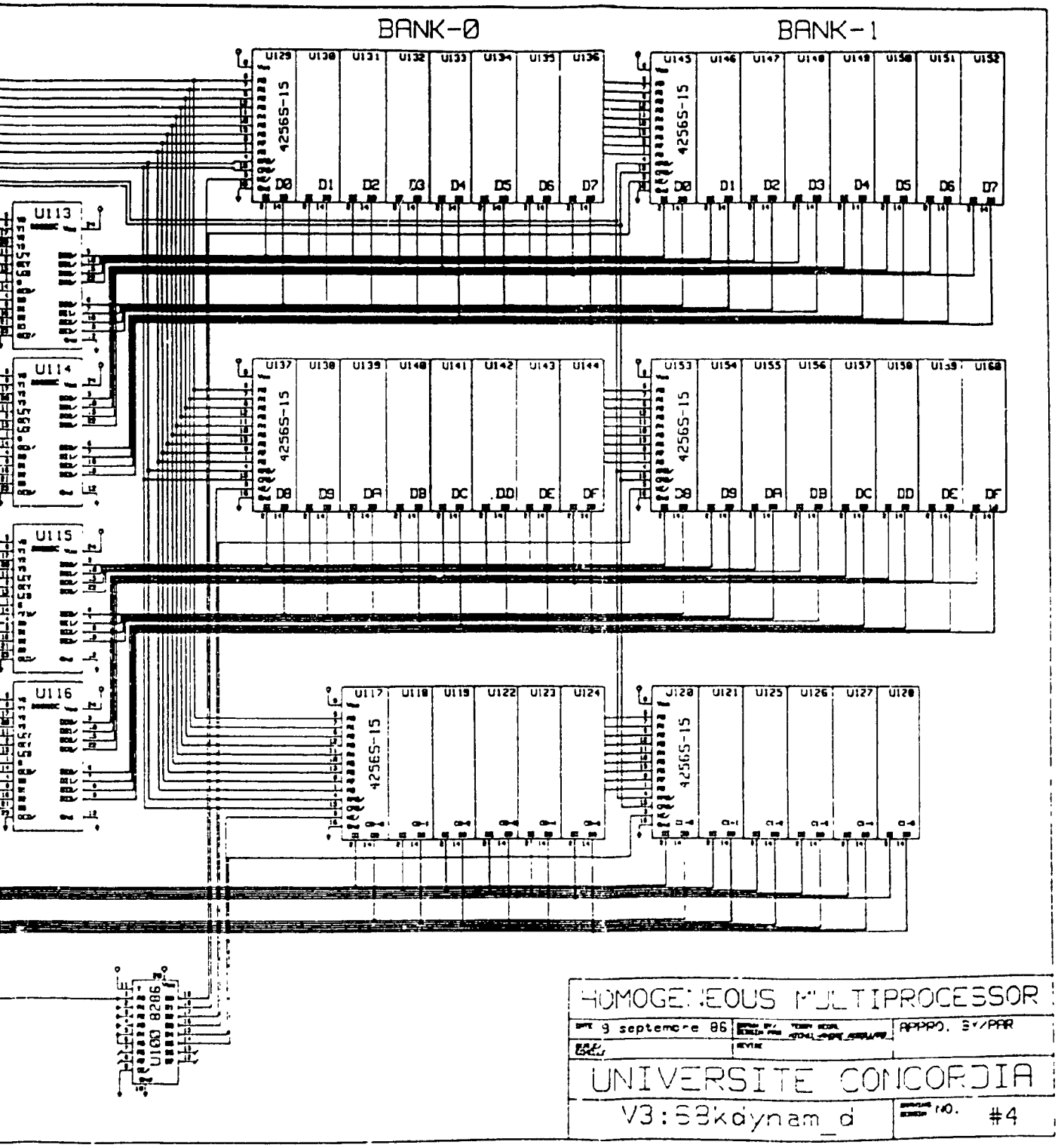


Figure A.9: The Schematic Design of the Dynamic Memory



HOMOGENEOUS MULTIPROCESSOR

DATE 9 septembre 86		APPRO. BY/PAR
	REVISE	

UNIVERSITE CONCORDIA

V3:S3kdynam_d	DRAWING NO. #4

Appendix B

B. The Static Memory Subsystem

The static memory system design is presented in this chapter. The static memory system incorporates the latest static memory integrated circuits, the 32 Kilobyte chip. In order to achieve our 1 Megabyte of memory, 32 of these integrated circuits are required. Apart from these 32 memory chips, only a handful of support chips from the 74LSXX series are required This leads to minimal amount of time required for the design, assembly, and debugging of the system.

When designing a memory system, all aspects of the procedure are simpler and less time consuming for a static memory system versus a dynamic memory system. Unlike the dynamic memory design, incorporation of special controllers and/or timers are not required, this facilitates a simple design procedure that will require a minimal amount of debugging. Due to the nature of static memory integrated circuits, their operation is asynchronous.

B.1 Selection of Components for the Static Memory Subsystem

The most prominent part of this design are the memory chips. At the time of design, the largest available static memory chips were 32 Kilobytes on a single DIP package. Originally, a 1 Megabyte static memory system was desired, however, due to lack of funding only a 0.5 Megabyte of memory was implemented. This was due to the high cost of the memory chips. The brand that was purchased for no other reason than that of availability was the Hitachi HM62256P. This chip contains 256 Kilobits of RAM which is organized into 32 Kilobytes. In order to achieve our 0.5 Megabyte, 16 of these chips were required. Aside from the memory chips, 3 standard decoders were used for individual chip selection, a buffer for *amplifying* signals that must be used by many chips and finally a flip-flop for the $\overline{\text{DTACK}}$.

B.2 The Hitachi HM62256P Architecture

Shown in figure B.1 is a block diagram of the Hitachi memory chip. As specified earlier, it is organized as 32 K x 8 bits. For 32 Kilobytes there are 15 address lines and 8 data lines. The data lines are bidirectional and can be tri-stated. Also included in the RAM chip there are $\overline{\text{WRITE ENABLE}}$, $\overline{\text{OUTPUT ENABLE}}$, and $\overline{\text{CHIP SELECT}}$.

The operation of the memory chip is very simple. An address is presented along with the $\overline{CS}$ (chip select), and either an active $\overline{WE}$ (write enable) or an active $\overline{OE}$ (output enable). The memory chips are rated at 150 ηS, therefore during a read cycle after the address, $\overline{CS}$, and $\overline{OE}$ are presented, it may take up to 150 ηS for the data to appear on the output latch of the memory chip. For a write cycle, the data must be present with the address, $\overline{CS}$, and $\overline{WE}$ for latching purposes and must be stable for at least 150 ηS.

B.3 Static Memory System Architecture

As stated previously, the static memory design is very simple. A 3-to-8 decoder is used to determine if the *requested* memory location is within the memory subsystem. If the address is within, then an output becomes active. This output then selects two 4-to-16 decoders which are used to determine which one of the 8 *banks* (each bank is 32 Kilowords) is to be selected. Two of these decoders are required, one for the high byte, and one for the low byte (required for byte read and byte write). This is achieved via the $\overline{UDS}$ and $\overline{LDS}$ from the MC68000. Our design can be found at the end of the appendix.

The Upper four address lines, A_{20} through A_{23} along with the address strobe, $\overline{AS}$, are used by the 74LS138 3-to-8 decoder to determine if the static memory is to be accessed. The address strobe and A_{23} are used as chip selects for the decoder, while A_{20} - A_{22} are used to partition the memory at the 1 Megabyte boundaries. Shown in table B.1 is the truth table for the decoder. By using A_{23} as one of the chip selects, the decoder is enabled only when an address in the upper half of possible 16 Megabytes is produced. The address strobe is used to only enable the decoder during a memory access.

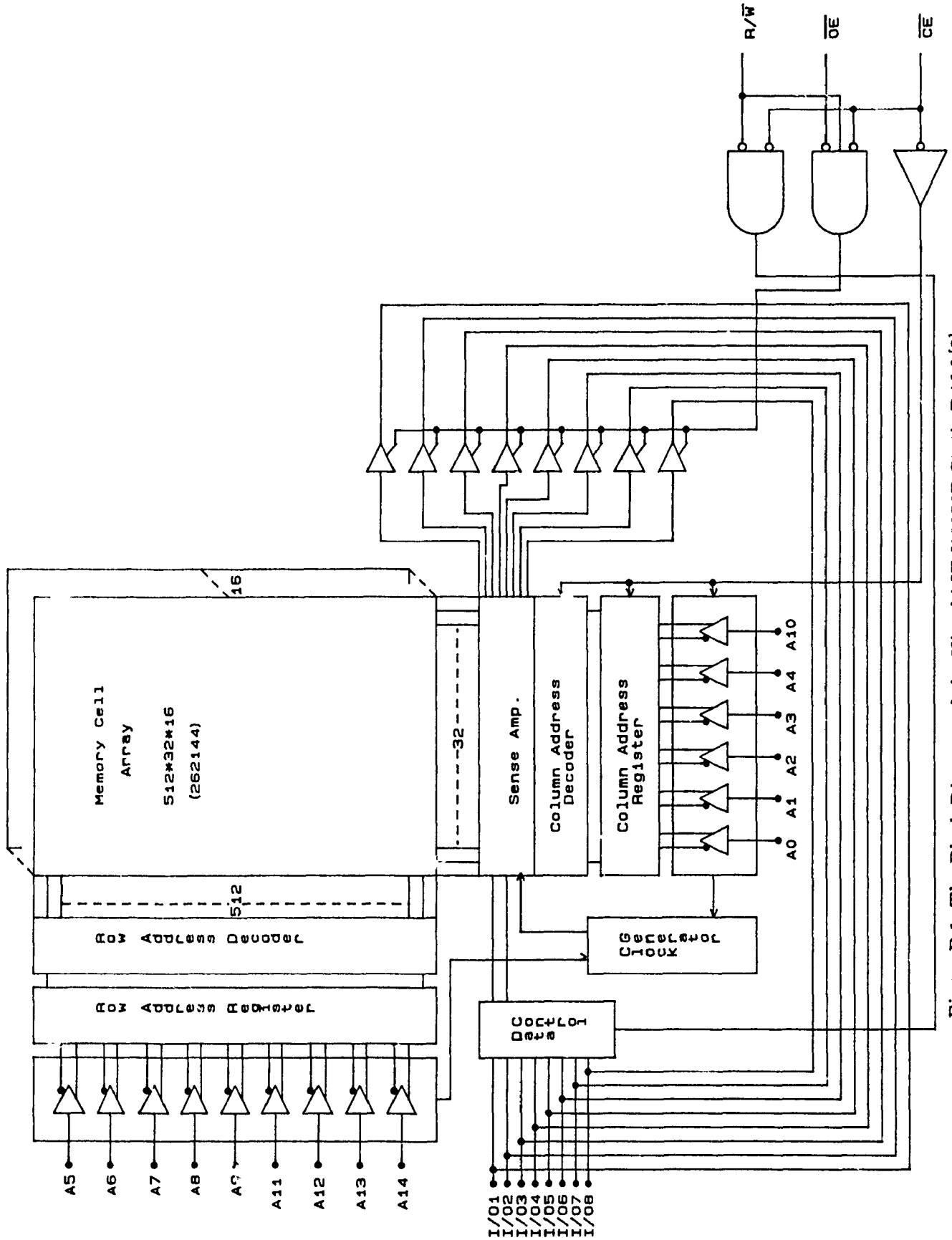


Figure B.1: The Block Diagram of the Hitachi HM62256P Static RAM.[9]

Table B.1: Memory Partition at the First Level.

INPUTS					OUTPUTS							
A	B	C	G_1	G_{2A}	$\overline{Y_0}$	$\overline{Y_1}$	$\overline{Y_2}$	$\overline{Y_3}$	$\overline{Y_4}$	$\overline{Y_5}$	$\overline{Y_6}$	$\overline{Y_7}$
A_{20}	A_{21}	A_{22}	A_{23}	$\overline{AS}$								
X	X	X	L	X	H	H	H	H	H	H	H	H
X	X	X	X	H	H	H	H	H	H	H	H	H
L	L	L	H	L	L	H	H	H	H	H	H	H
L	L	H	H	L	H	L	H	H	H	H	H	H
L	H	L	H	L	H	H	L	H	H	H	H	H
L	H	H	H	L	H	H	H	L	H	H	H	H
H	L	L	H	L	H	H	H	H	L	H	H	H
H	L	H	H	L	H	H	H	H	H	L	H	H
H	H	L	H	L	H	H	H	H	H	H	L	H
H	H	H	H	L	H	H	H	H	H	H	H	L

Note that input G_{2A} is pulled low.

Referring to the memory map in table 2.1, the local memory appears at A_{23}, A_{22} = 1,1. This limits the valid outputs from the 4-to-16 decoder to the lower four (i.e. $\overline{Y_{12}}$ - $\overline{Y_{15}}$). These four outputs will enable the last fourth of the memory range, (i.e. 12 Megabyte - 16 Megabyte). Therefore, the 1 Megabyte of static memory should be enabled from $\overline{Y_4}$. The result of this gives us 1 Megabyte of memory range from the 12 Megabyte to the 13 Megabyte boundaries.

The next step in decoding the address is done using 74ALS154 4-to-16 decoders. By using this decoder, the 1 Megabyte range can be expanded into 16 - 32 Kilobyte partitions. Two of these decoders are used, one for the upper byte and the other for the lower byte. This is achieved by using the next lower address lines, A_{16} through A_{19} as the inputs to the decoder. The other chip select is used for the byte address strobe. On one of the decoders, the upper address strobe ($\overline{UDS}$) is a chip select, and on the other, the lower address strobe ($\overline{LDS}$) is used. Obviously, the static memory chips that will be enabled by this decoder using the upper data strobe will produce the upper byte, while the lower byte is produced by the memory chips connected to the other decoder that utilizes the lower data strobe. Table B.2 depicts the truth table for the 4-to-16 decoders.

Table B.2: Truth table for the 4-to-16 decoders.

INPUTS						OUTPUTS															
A	B	C	D	G_2	G_1																
A_{16}	A_{17}	A_{18}	A_{19}	$\overline{Y_4}$	$\overline{UDS}$ $\overline{LDS}$	$\overline{Y_0}$	$\overline{Y_1}$	$\overline{Y_2}$	$\overline{Y_3}$	$\overline{Y_4}$	$\overline{Y_5}$	$\overline{Y_6}$	$\overline{Y_7}$	$\overline{Y_8}$	$\overline{Y_9}$	$\overline{Y_{10}}$	$\overline{Y_{11}}$	$\overline{Y_{12}}$	$\overline{Y_{13}}$	$\overline{Y_{14}}$	$\overline{Y_{15}}$
X	X	X	X	H	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
X	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	L	L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	L	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	H	L	L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	H	H	L	L	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H
L	H	L	L	L	L	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H
L	H	L	H	L	L	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H
L	H	H	L	L	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H
L	H	H	H	L	L	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H
H	L	L	H	L	L	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H
H	L	H	L	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H
H	L	H	H	L	L	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H
H	H	L	L	L	L	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H
H	H	L	H	L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H
H	H	H	L	L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H
H	H	H	H	L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L

From the two 4-to-16 decoders, 32 distinct chip selects are produced. Each of the outputs is connected to the chip select on each of the static memory chips. Two decoders are necessary to accommodate the MC68000 CPU so that byte writes can be fulfilled. The balance of the address lines, A_1 through A_{15} are connected to the static memory chip address inputs, A_0 through A_{14}, respectively. Also, each of the static memory chips required a write enable and an output enable, these signals are derived from the $R/\overline{W}$ signal.

The propagation delays associated with each component of the static memory system can be found in table B.3 below. The following information does not include the buffers that drive the system (8286 switches). It is not necessary to include the 8286 buffers because they are common to all of the signals entering the system, and all of the

signals exiting the system. Therefore all of the signals exhibit an equivalent propagation delay that enter the system which is 50 ηS and another 50 ηS when exiting the system. This informs us to add an extra 100 ηS to the amount of time required to perform a read or write cycle. From the table below, the total amount of time is 195 ηS, add the 100 ηS delay for the switch, and the total amount of time required for a cycle is 295 ηS.

Table B.3: Propagation Delays in the Static Memory System.

Component	Average Propagation Delay
74LS138	22 ηS
74154	23 ηS
SRM20256C 32 K x 8	150 ηS
Total	195 ηS

Finally, the data bus of all of the memory chips that are connected to the 4-to-16 decoder that uses the upper data strobe as a chip select are *tied* together to produce the upper portion of the data bus, D_8 through D_{15}. And of course, all of the memory chips that are connected to the decoder that uses the lower data strobe as a chip select are tied together to produce the lower data bus, D_0 through D_7. There will never be a collision on the data bus between any two or more memory chips due to the nature of the 4-to-16 decoders, it is impossible for more than one output to be active at any given time.

It has been determined that the memory system requires 195 ηS to perform a read or write operation. From this we can conclude that the $\overline{\text{DTACK}}$ should become active (low) some time (no less than 90 ηS) before the 195 ηS mark. Then it should become inactive (high) for latching into the CPU some time after the 195 ηS mark. By using our 8 MHz clock, the $\overline{\text{DTACK}}$ will become active after 125 ηS and then inactive again after 250 ηS. Including the propagation delays of the switch, the actual time that it takes for a cycle to complete is 350 ηS.

The $\overline{\text{DTACK}}$ is the final signal that must be generated. In our system, the $\overline{\text{DTACK}}$ is generated by using a 74LS74 D-type flip-flop. The output of the 74LS138 ($\overline{Y_4}$) is

inverted and connected to the $\overline{CLR}$ of the flip-flop. The Preset (PR) and input D are pulled to 5 volts through 4.7 Kilo Ω resistors. The output $\overline{Q}$ is used as the $\overline{DTACK}$ for the memory and the clock (CLK) is connected to an 8 MHz source.

Before a memory request begins, the output of the 74LS138 decoder, ($\overline{Y_4}$) is inactive (high). This signal is then passed through an inverter which is then connected to the $\overline{CLR}$. This causes the output $\overline{Q}$ ($\overline{DTACK}$) to be inactive (high). Once the memory request begins, the output $\overline{Y_4}$ from the 74LS138 decoder changes state to low causing the $\overline{CLR}$ to be inactive (high). This allows the flip-flop to operate, and upon the first rising edge of the clock, the $\overline{DTACK}$ ($\overline{Q}$) becomes active (low), due to latching the 5 volts on the D input and passing in to the Q output. After a predetermined amount of time the data will enter/exit the memory and the $\overline{AS}$ will become inactive (high) and the $\overline{CLR}$ will become active (low). This clears the flip-flop, causing the $\overline{Q}$ ($\overline{DTACK}$) to become inactive (high) and the memory request has completed. This can be clearly understood by referring to the timing diagram shown in figure B.2.

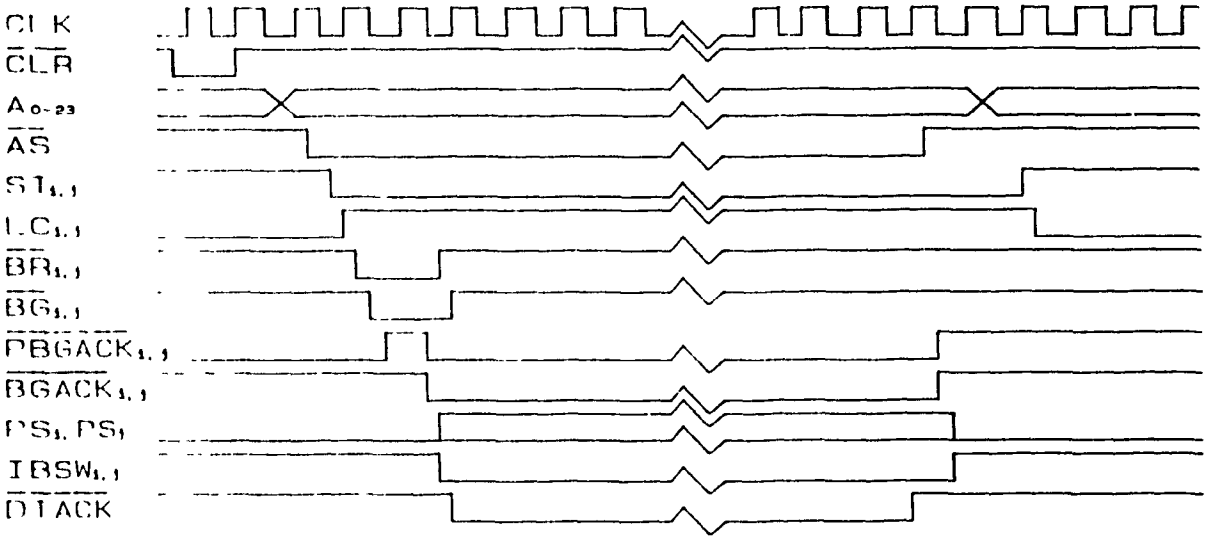


Figure B.2: Timing Diagram for the $\overline{DTACK}$.

B.4 Implementation and Experimental Results

The design and implementation of the static memory subsystem was less complicated than the dynamic memory subsystem. Static memory was chosen for the second Processing Element due to the fall in prices at the time. Therefore we were able to purchase all of the required parts for approximately the same cost as those required for a dynamic memory system. The other reason for choosing a static system is its simpler design and debugging phases. This static memory system is built on a *daughter board* that resides above the Versabus board that contains the processor. There is a connector for the two subsystems. A *daughter board* was chosen so that it might be possible to incorporate the other subsystems that have yet to be implemented on the board with the processor. Another reason is the possibility of testing other processors with this memory board.

Due to lack of funds, the static memory systems is only 0.5 Megabytes. The system is implemented so that as funding becomes available, the additional memory chips can be added; the control circuitry already exists.

During the testing and debugging stages, only one error existed in the system. One of the control signals should have been complemented and it was not. The inverter was added and the static memory subsystem operated perfectly. Since then, the board has been shipped to Victoria, B. C., and a new bug has appeared. Again, crosstalk is suspected and most of them were alleviated by shortening the ribbon cable that connects the daughter board which contains the static memory to the processor. The average memory access cycle time is 300 ηS., less than half of that of the dynamic memory subsystem. Shown in figure B.3 is the layout for the static memory subsystem. And finally in figure B.4. the schematic design for the static memory subsystem.

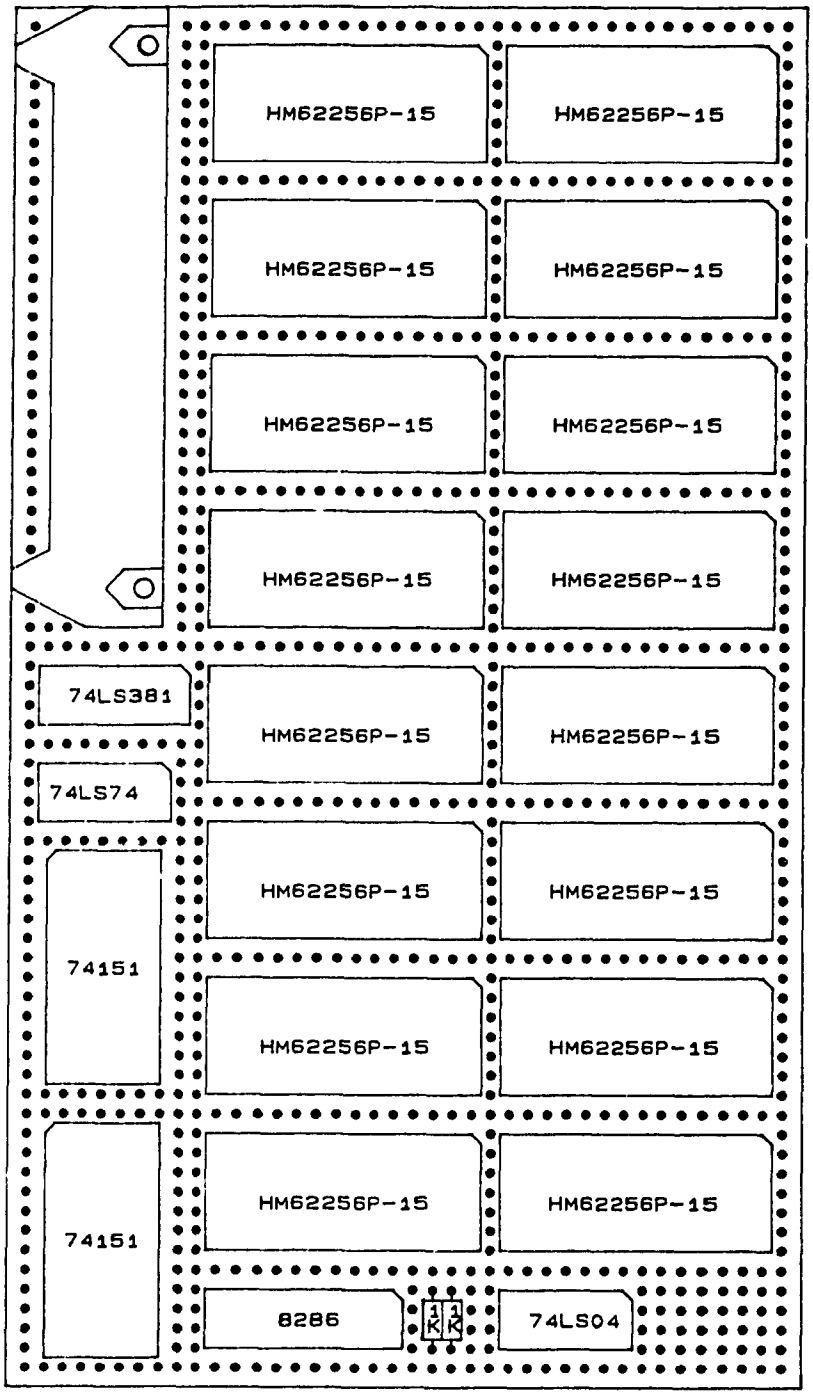


Figure B.3: The Layout of the Static Memory Subsystem.

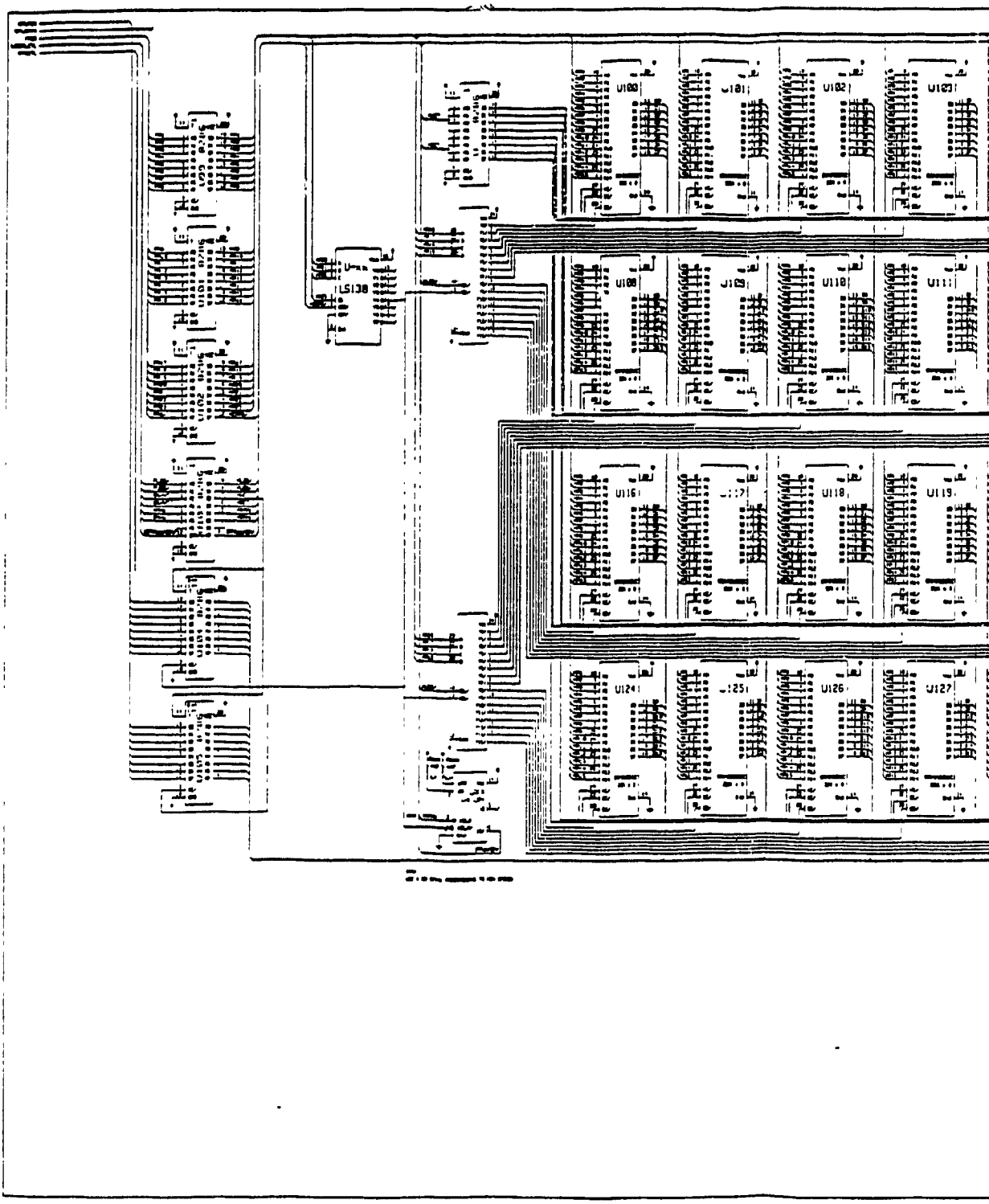


Figure B.4: The Schematic Design of the Static Memory

HOMOGENEOUS MULTIPROCESSOR

09 AVRIL 198?

APPRO. BY/PAR

UNIVERSITE CONCORDIA

V3:68K.mem_d

NO #1

Appendix C

The following is the derivation of the equations that describe Phase I. Equations 2.1 through 2.4 represent Phase I.

From the state diagram shown in figure 2.2 and Algorithm 1.2, the following next state table is produced.

Q^v	$Q^{v+1} = f(R,ST_{I,J},ST_{J,J+1})$				
	0XX	100	101	110	111
OPEN	OPEN	OPEN	OPEN	GREY	GREY
GREY	OPEN	GREY	CLOSED	GREY	CLOSED
CLOSED	OPEN	CLOSED	CLOSED	CLOSED	CLOSED

There are three possible combinations for the state assignment which are shown below. All other possibilities will lead to one of the following outcomes.

State	Ass. 1	Ass. 2	Ass. 3
OPEN	00	00	00
GREY	01	11	10
CLOSED	11	01	01
N.U.	10	10	11

From the three state assignments above, we can produce the next state tables for each assignment substituting the logical values for the states that will then be used for the karnaugh maps

Assignment 1						
		R, $ST_{I,J}$, $ST_{J,J+1}$				
STATE	Q_1,Q_0	0XX	100	101	111	110
OPEN	00	00	00	00	01	01
GREY	01	00	01	11	11	01
CLOSED	11	00	11	11	11	11
N.U.	10	00	00	00	00	00

Assignment 2						
		R, $ST_{I,J}$, $ST_{J,J+1}$				
STATE	Q_1,Q_0	0XX	100	101	111	110
OPEN	00	00	00	00	11	11
GREY	11	00	11	01	01	11
CLOSED	01	00	01	01	01	01
N.U.	10	00	00	00	00	00

Assignment 3						
		R, $ST_{I,J}$, $ST_{J,J+1}$				
STATE	Q_1,Q_0	0XX	100	101	111	110
OPEN	00	00	00	00	10	10
GREY	10	00	10	01	01	10
CLOSED	01	00	01	01	01	01
N.U.	11	00	00	00	00	00

It is possible to use many different type of flip flops to design the circuit. In this design we consider the use of D type and J-K type. Each assignment will be designed using both D and J-K type flip flops. This is done to determine the most efficient design.

This is the design for assignment 1 using D type flip flops.

Q_1,Q_0	R, $ST_{I,J}$, $ST_{J,J+1}$				
	0XX	100	101	111	110
O 00	0	0	0	1	1
G 01	0	1	1	1	1
C 11	0	1	1	1	1
N 10	0	0	0	0	0

$$D_0 = Q_0\ R + \overline{Q_1} \cdot R \cdot ST_{I,J}$$

$Q_1.Q_0$	R, $ST_{I,J}$, $ST_{J,J+1}$				
	0XX	100	101	111	110
O 00	0	0	0	0	0
G 01	0	0	1	1	0
C 11	0	1	1	1	1
N 10	0	0	0	0	0

$$D_1 = Q_0 \cdot R \cdot ST_{J,J+1} + Q_1 \cdot Q_0 \cdot R$$

This is the design for assignment 1 using J-K type flip flops.

$Q_1.Q_0$	R, $ST_{I,J}$, $ST_{J,J+1}$				
	0XX	100	101	111	110
O 00	0	0	0	1	1
G 01	X	X	X	X	X
C 11	X	X	X	X	X
N 10	0	0	0	0	0

$$J_0 = \overline{Q_1} \cdot R \cdot ST_{I,J}$$

$Q_1.Q_0$	R, $ST_{I,J}$, $ST_{J,J+1}$				
	0XX	100	101	111	110
O 00	X	X	X	X	X
G 01	1	0	0	0	0
C 11	1	0	0	0	0
N 10	X	X	X	X	X

$$K_0 = \overline{R}$$

$Q_1.Q_0$	R, $ST_{I,J}$, $ST_{J,J+1}$				
	0XX	100	101	111	110
O 00	0	0	0	0	0
G 01	0	0	1	1	0
C 11	X	X	X	X	X
N 10	X	X	X	X	X

$$J_1 = Q_0 \cdot R \cdot ST_{J,J+1}$$

Q_1,Q_0	R, $ST_{I,J}$, $ST_{J,J+1}$				
	0XX	100	101	111	110
O 00	X	X	X	X	X
G 01	X	X	X	X	X
C 11	1	0	0	0	0
N 10	1	1	1	1	1

$$K_1 = \overline{Q_0} \cdot \overline{R}$$

The procedure for assignment 2 and assignment 3 are similar, they will not be presented, only the resulting equations.

Assignment 2, using D type flip flops.

$$D_0 = Q_0 \cdot R + \overline{Q_1} \cdot R \cdot ST_{I,J}$$

$$D_1 = Q_1 \cdot Q_0 \cdot R \cdot ST_{J,J+1} + \overline{Q_1} \cdot \overline{Q_0} \cdot R \cdot ST_{I,J}$$

Assignment 2, using J-K type flip flop.

$$J_0 = \overline{Q_1} \cdot R \cdot ST_{I,J}$$

$$K_0 = \overline{R}$$

$$J_1 = \overline{Q_0} \cdot R \cdot ST_{I,J}$$

$$K_1 = \overline{Q_0} + \overline{R} + R \cdot ST_{J,J+1}$$

Assignment 3, using D type flip flops.

$$D_0 = \overline{Q_1} \cdot Q_0 \cdot R + Q_1 \cdot \overline{Q_0} \cdot R \cdot ST_{J,J+1}$$

$$D_1 = Q_1 \cdot \overline{Q_0} \cdot R \cdot \overline{ST_{J,J+1}} + \overline{Q_1} \cdot \overline{Q_0} \cdot R \cdot ST_{I,J}$$

Assignment 3, using J-K type flip flops.

$$J_0 = Q_1 \quad R \cdot ST_{J,J+1}$$

$$K_0 = Q_1 + \overline{R}$$

$$J_1 = \overline{Q_0} \cdot R \cdot ST_{I,J}$$

$$K_1 = Q_0 + \overline{R} + R \cdot ST_{J,J+1}$$

By examining the six different sets of equations produced by the three assignments using two different types of flip flops, it is obvious that assignment 1 using J-K flip flops is the most efficient.

$$J_0 = \overline{Q_1} \cdot R \cdot ST_{I,J}$$

$$K_0 = \overline{R}$$

$$J_1 = Q_0 \cdot R \cdot ST_{J,J+1}$$

$$K_1 = \overline{Q_0} \cdot \overline{R}$$

And finally, for Logical Closure and the Status, we obtain.

$$LC_{I,J} = Q_1 \cdot Q_0$$

$$ST_{I,J} = \overline{Q_1} \cdot \overline{Q_0}$$

The following design procedure is shown for Phase II. This design requires three states, therefore two flip flops are required. The design is attempted using both D type flip flops and J-K type flip flops. First, from the state diagram we obtain the following next state table.

Note that the term 5AND is the following:

$$5AND = \overline{PAS} \cdot \overline{PUDS} \cdot \overline{PLDS} \cdot \overline{PDTACK} \cdot \overline{PBGACK}$$

Q^v	$Q^{v+1} = f(LC_{i,j}, \overline{BG}, 5AND)$				
	0XX	100	101	110	111
NO REQ	NO REQ	BR	BR	BR	BR
BR	NO REQ	BR	BGACK	BR	BR
BGACK	NO REQ	BGACK	BGACK	BGACK	BGACK

There are three possible combinations for the state assignment which are shown below. All other possibilities will lead to one of the following outcomes.

State	Ass. 1	Ass. 2	Ass. 3
NO REQ	00	00	00
BR	01	01	11
BGACK	10	11	10
N.U.	11	10	01

From the three state assignments, we can produce the next state tables for each assignment substituting the logical values for the states that will then be used for the karnaugh maps.

Assignment 1						
		$LC_{i,j}, \overline{BG}, 5AND$				
STATE	Q_1, Q_0	0XX	100	101	111	110
NO REQ	00	00	01	01	01	01
BR	01	00	01	10	01	01
BGACK	10	00	10	10	10	10
N.U.	11	00	00	00	00	00

Assignment 2						
		$LC_{I,J},\overline{BG},5AND$				
STATE	Q_1,Q_0	0XX	100	101	111	110
NO REQ	00	00	01	01	01	01
BR	01	00	01	11	01	01
BGACK	11	00	11	11	11	11
N.U.	10	00	00	00	00	00

Assignment 3						
		$LC_{I,J},\overline{BG},5AND$				
STATE	Q_1,Q_0	0XX	100	101	111	110
NO REQ	00	00	11	11	11	11
BR	11	00	11	10	11	11
BGACK	10	00	10	10	10	10
N.U.	01	00	00	00	00	00

It is possible to use many different types of flip flops to design this circuit. In this design we consider the use of D type and J-K type. Each assignment will be designed using both D and J-K type flip flops. This is done to determine the most efficient design.

This is the design for assignment 2 using D type flip flops.

Q_1,Q_0	$LC_{I,J},\overline{BG},5AND$				
	0XX	100	101	111	110
NR 00	0	1	1	1	1
BR 01	0	1	1	1	1
BG 11	0	1	1	1	1
NU 10	0	0	0	0	0

$$D_0 = \overline{Q_1} \cdot LC_{I,J} + Q_0 \cdot LC_{I,J}$$

Q_1,Q_0	$LC_{1,J},\overline{BG},5AND$				
	0XX	100	101	111	110
NR 00	0	0	0	0	0
BR 01	0	0	1	0	0
BG 11	0	1	1	1	1
NU 10	0	0	0	0	0

$$D_1 = Q_1 \cdot Q_0 \cdot LC_{1,J} + Q_0 \cdot LC \cdot \overline{\overline{BG}} \cdot 5AND$$

This is the design for assignment 2 using J-K flip flops.

Q_1,Q_0	$LC_{1,J},\overline{BG},5AND$				
	0XX	100	101	111	110
NR 00	0	1	1	1	1
BR 01	X	X	X	X	X
BG 11	X	X	X	X	X
NU 10	0	0	0	0	0

$$J_0 = \overline{Q_1} \cdot LC_{1,J}$$

Q_1,Q_0	$LC_{1,J},\overline{BG},5AND$				
	0XX	100	101	111	110
NR 00	X	X	X	X	X
BR 01	1	0	0	0	0
BG 11	1	0	0	0	0
NU 10	X	X	X	X	X

$$K_0 = \overline{LC_{1,J}}$$

Q_1,Q_0	$LC_{1,J},\overline{BG},5AND$				
	0XX	100	101	111	110
NR 00	0	0	0	0	0
BR 01	0	0	1	0	0
BG 11	X	X	X	X	X
NU 10	X	X	X	X	X

$$J_1 = Q_0 \cdot LC_{1,J} \cdot \overline{\overline{BG}} \cdot 5AND$$

Q_1,Q_0	$LC_{I,J},\overline{BG},5AND$				
	0XX	100	101	111	110
NR 00	X	X	X	X	X
BR 01	X	X	X	X	X
BG 11	1	0	0	0	0
NU 10	1	1	1	1	1

$$K_1 = \overline{LC_{I,J}} + \overline{Q_0}$$

And finally,

$$\overline{BR} = \overline{Q_1 \cdot Q_0}$$

$$\overline{BGACK} = \overline{Q_1 \cdot Q_0}$$

The procedure for assignment 1 and assignment 3 are similar, they will not be presented, only the resulting equations.

Assignment 1, using D type flip flops.

$$D_0 = \overline{Q_1} \cdot \overline{Q_0} \cdot LC_{I,J} + \overline{Q_1} \cdot LC_{I,J}\,\overline{BG} + \overline{Q_1} \cdot LC_{I,J} \cdot \overline{5AND}$$

$$D_1 = Q_1 \cdot \overline{Q_0} \cdot LC_{I,J} + \overline{Q_1} \cdot Q_0 \cdot LC_{I,J} \cdot \overline{BG} \cdot 5AND$$

Assignment 1, using J-K type flip flops.

$$J_0 = \overline{Q_1} \cdot LC_{I,J}$$

$$K_0 = Q_1 + \overline{LC_{I,J}} + \overline{BG} \cdot 5AND$$

$$J_1 = Q_0 \cdot LC_{I,J} \cdot \overline{BG} \cdot 5AND$$

$$K_1 = Q_0 + \overline{LC_{I,J}}$$

And for assignment 1:

$$\overline{BR} = \overline{\overline{Q_1} \cdot Q_0}$$

$$\overline{BGACK} = \overline{Q_1 \cdot \overline{Q_0}}$$

Assignment 3, using D type flip flops.

$$D_0 = \overline{Q_1} \cdot \overline{Q_0} \cdot LC_{1,J} + Q_1 \cdot Q_0 \cdot LC_{1,J} \cdot \overline{BG} + Q_1 \cdot Q_0 \cdot LC_{1,J} \cdot \overline{5AND}$$

$$D_1 = \overline{Q_0} \cdot LC_{1,J} + Q_1 \cdot LC_{1,J}$$

Assignment 3, using J-K type flip flops.

$$J_0 = \overline{Q_1} \cdot LC_{1,J}$$

$$K_0 = \overline{Q_1} + \overline{LC_{1,J}} + \overline{\overline{BG}} \cdot 5AND$$

$$J_1 = \overline{Q_0} \cdot LC_{1,J}$$

$$K_1 = \overline{LC_{1,J}}$$

And for assignment 3:

$$\overline{BR} = \overline{Q_1 \cdot Q_0}$$

$$\overline{BGACK} = \overline{Q_1 \cdot \overline{Q_0}}$$

By examining the six different sets of equations produced by the three assignments using two different types of flip flops, it is obvious that assignment 2 using J-K flip flops is the most efficient.

$J_0 = \overline{Q_1} \cdot LC_{I,J}$

$K_0 = \overline{LC_{I,J}}$

$J_1 = Q_0 \cdot LC_{I,J} \cdot \overline{BG} \cdot 5AND$

$K_1 = \overline{LC_{I,J}} + \overline{Q_0}$

$\overline{BR} = \overline{Q_1 \cdot Q_0}$

$\overline{BGACK} = \overline{Q_1 \cdot Q_0}$

Appendix D

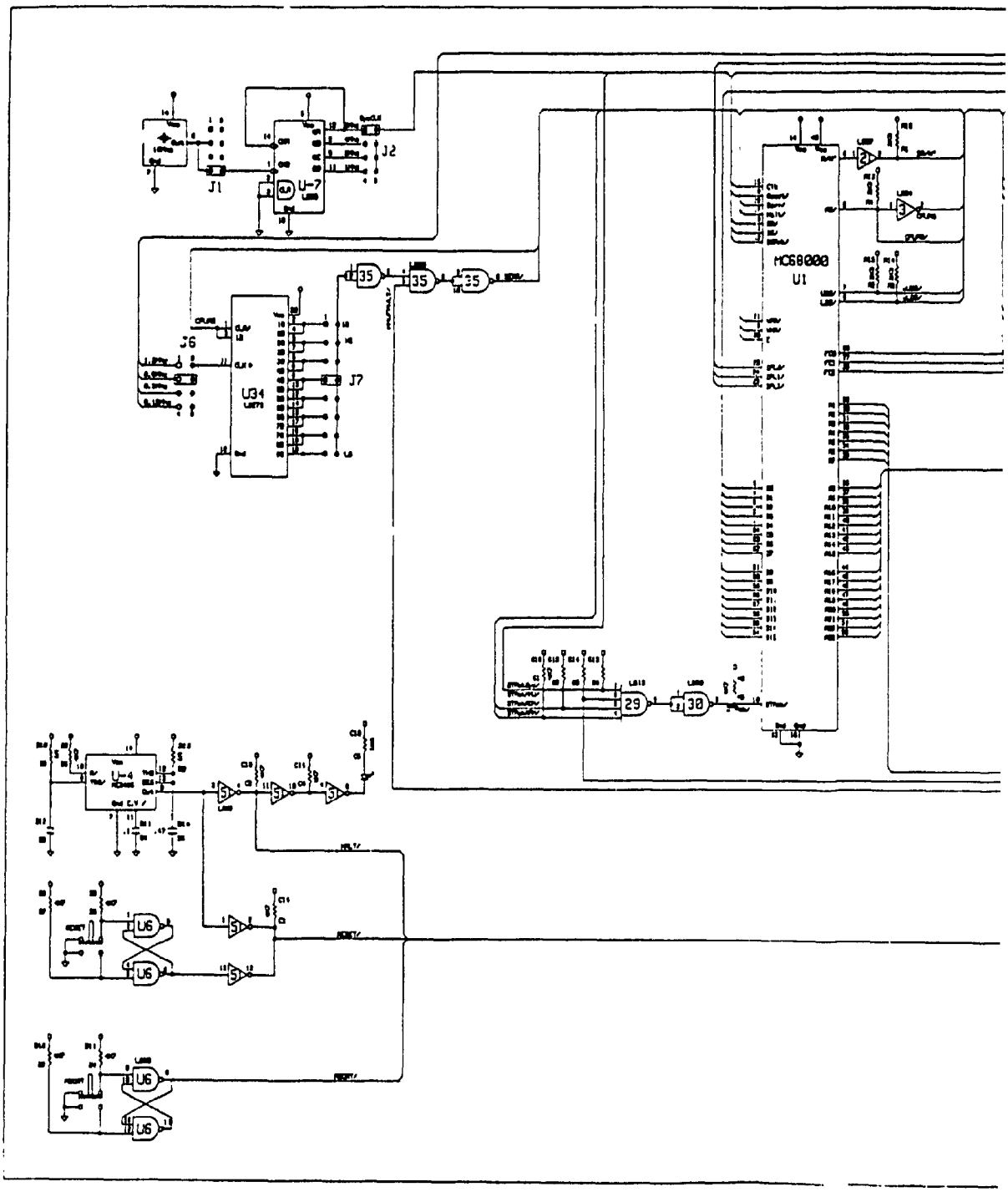
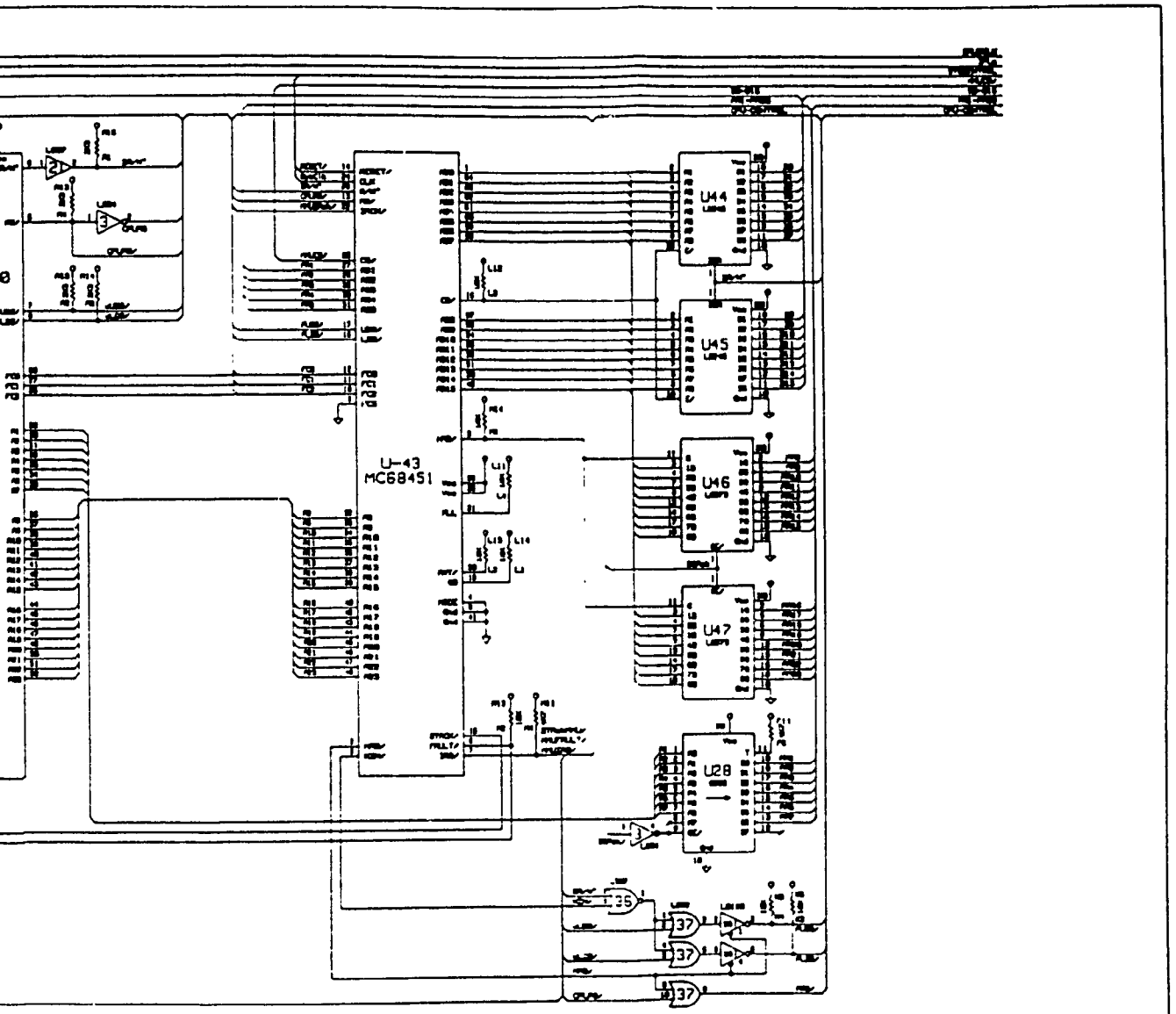


Figure D.1: Processor Schematics (Sheet 1 of 3)



HOMOGENEOUS MULTIPROCESSOR

FEVRIER 1986		APPRO. BY/PAR

UNIVERSITE CONCORDIA

MICHEL:68kpcpu_d; NO. #1

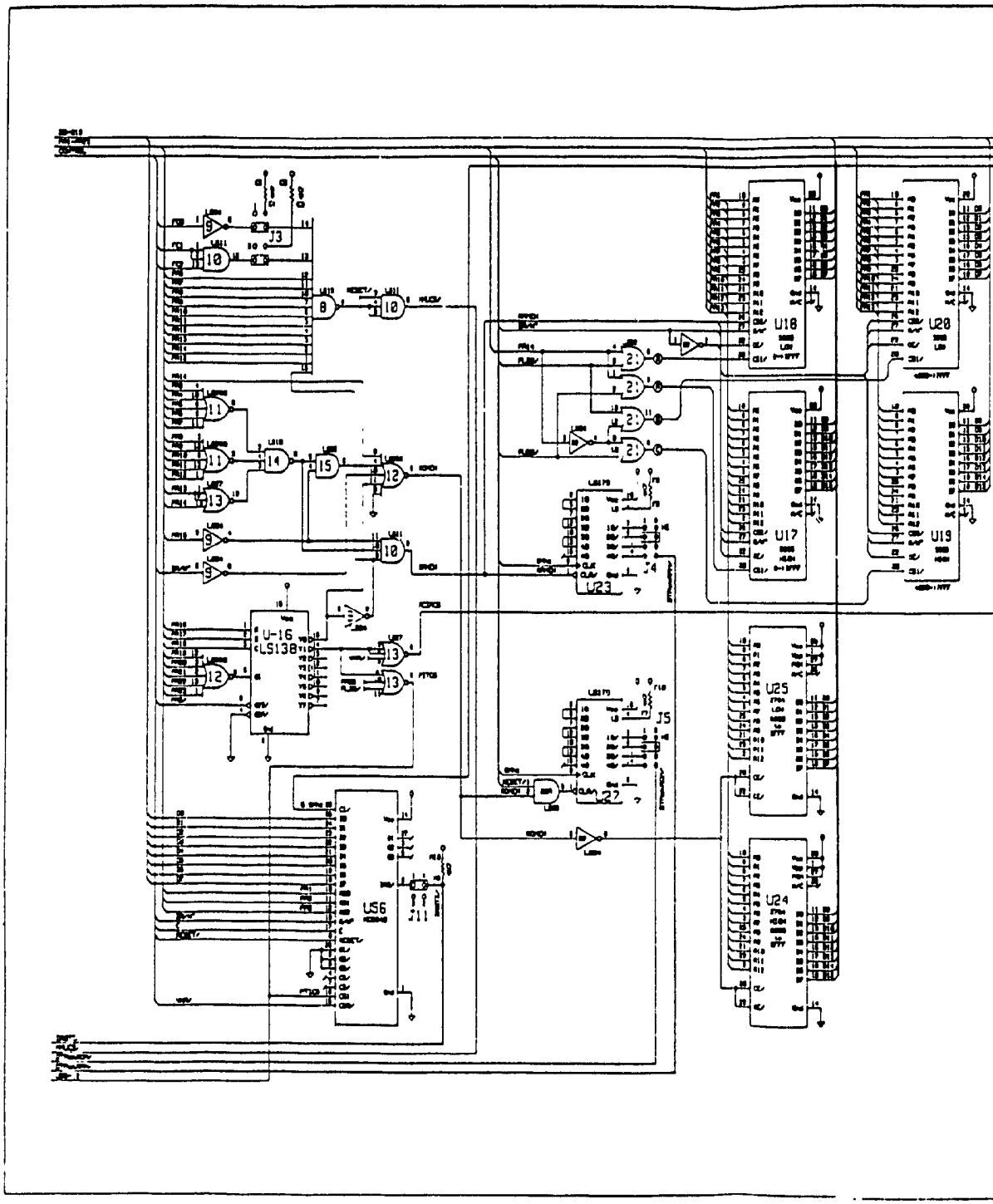


Figure D.2: Processor Schematics (Sheet 2 of 3)

HOMOGENECUS MULTIPROCESSOR

DATE FEVRIER 1986 APPRO. BY/PAR

UNIVERSITE CONCORDIA

NO. #2

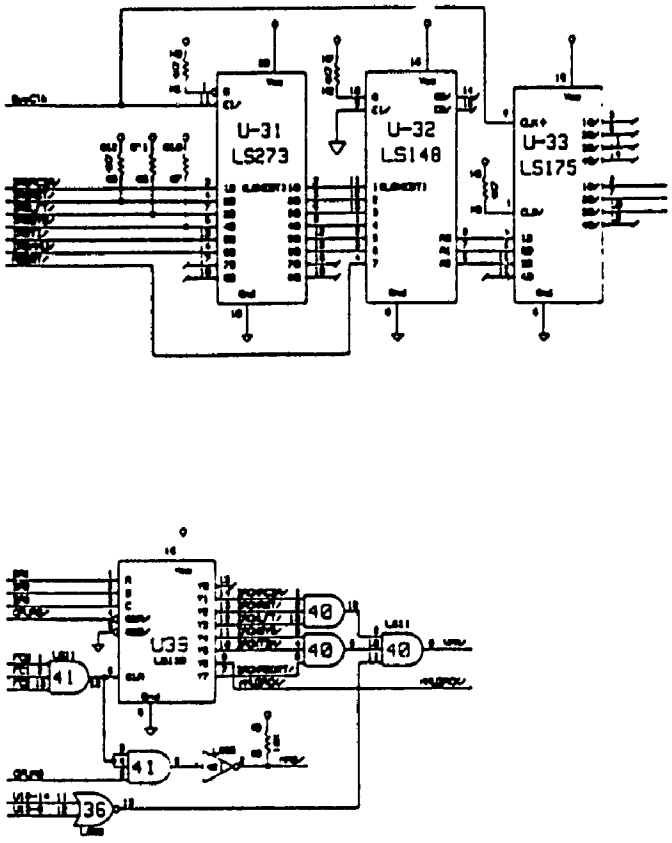
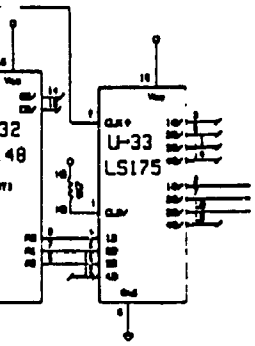


Figure D.3: Processor Schematics (Sheet 3 of 3)



HOMOGENEOUS MULTIPROCESSOR		
DATE FEVRIER 1986		APPRO. BY/PAR
UNIVERSITE CONCORDIA		
MICHEL:68ksys	DRAWING NO.	#3