A parallel microprocessor architecture for a task flow programming enviornment [sic] /

John C. Pagano
Lehigh University

A PARALLEL MICROPROCESSOR ARCHITECTURE FOR A TASK FLOW PROGRAMMING ENVIORNMENT

by

JOHN C. PAGANO

Thesis

Presented to the Graduate Committee

of Lehigh University

in candidacy for the degree of

Master of Science in Electrical Engineering

Lehigh University

1987

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

date: Dec 15, 1987

[illegible]
Advisor in charge

[illegible]
EE Division Chairperson

[illegible]
CSEE Department Chairperson

ACKNOWLEDGMENT

I would like to acknowledge the advice and support of Professor M. D. Wagh. This work was possible because of his great knowledge and open mind, and his constant desire to teach. He has taught me to always be constructive, and that curiosity is a gift.

Table of Contents

List of Figures

List of Tables

Abstract

This thesis describes the design and construction of a parallel computer with four nodes. The system architecture is robust enough to withstand a complete loss of all but one processor nodes at any time during execution. To achieve this, the architecture departs from the conventional parallel architecture ideas in that there is no master and slave relation between processors. Each processor is independent and can perform the system control through software. This machine uses coarse grained parallelism and a shared bus architecture and relies on task flow ideas developed here. The system is expandable and is easily adaptable to future processor technology. This parallel computer could be used in any application requiring high throughput coupled with high reliability.

Chapter 1
INTRODUCTION

The computer industry has long relied on rapid advances in technology to produce faster and more powerful computers. To take advantage of technological breakthroughs such as cheaper memory and large scale integration the architectures of computers must constantly be modified. These modifications are sometimes severe, but none has had the ramifications of the one currently threatening to take place -- Parallel architectures.

1.1 Parallel Architectures

There are currently three approaches to making computers faster and yet cost effective. The first approach concentrates on improving the instructions executed by the processor and there are two divergent idioms, one which says instructions should be made simpler, the other saying they should be made more complex. Interestingly, most practical success has come from taking aspects of both idioms. Reduced instruction set computers (RISC) are an attempt to make the processor execute each instruction as quickly as possible reducing bus idle time and increasing system efficiency. The goal is to have all instructions execute in one clock cycle, and this is accomplished by eliminating complex instructions and streamlining the instruction set. RISC techniques will have a definite impact on computers, but it is a one time gain, so it does not eliminate the need for other improvements. Complex instruction set computers (CISC) are an attempt at having high level instructions directly executed by the processor instead of being interpreted by compilers and assembly language macros. The gain in speed comes because the processor can execute the instruction via firmware without having to do consecutive instruction fetches which would be the case if the compiler replaced the high level command with many simple instructions. The problem that CISC architectures face is that they lose flexibility and must spend a great amount of time decoding the many instructions they support.

Most success with instruction sets has been a combination of what may appear to be the contradictory goals of RISC and CISC. Instruction sets are designed as streamlined as possible but include some of the more common high level commands.

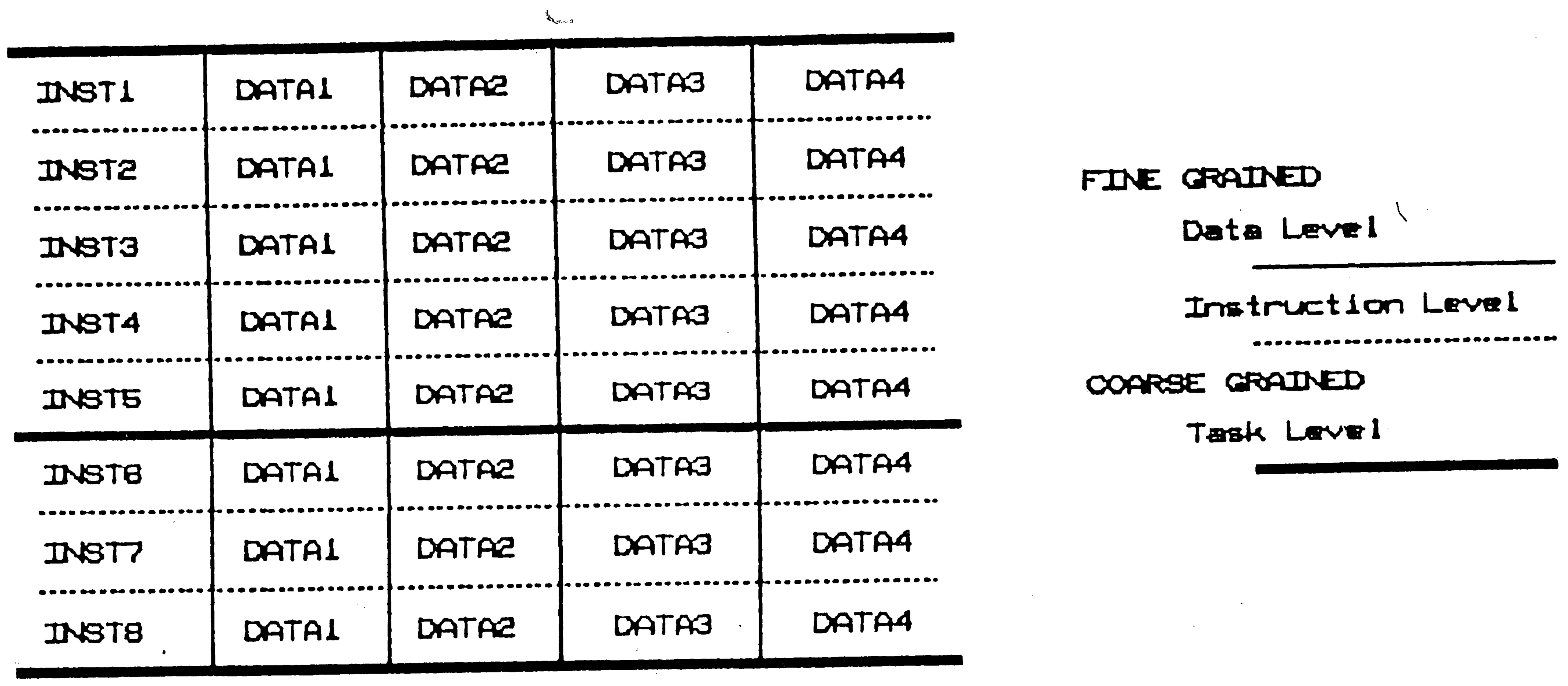
Another approach to improving computers is to simply take what exists and make it bigger and faster. Bigger means more data bits on the bus and bigger memory. Faster means increased clock speed of the processor and faster memory. Bigger has proved very effective in all computers and is currently making its final inroads in microcomputers with the introduction of 32 bit machines. This success, however, has limit. Since the largest data primitive operated on by a single instruction is the floating point number, and since few applications require precision greater than 10 places with an exponent of +/- 512 there is little gained in going beyond 64 bits. And even with 64 bits, most of the precision is wasted in all but iterative finite element analysis and other scientific applications. Faster has also been very effective in making machines more powerful but faster processors only come with much effort, time and money. Since faster circuits can be taken advantage of by any architecture, they have not reduced interest in developing other methods of improving computers.

The third approach takes advantage of the decreasing relative cost of the processing power in a system. Of the three components, only the cost of the processing power has been decreasing at an accelerating rate. I/O cost has been relatively constant, and the cost of storage has only kept pace with the increased needs caused by the increased processing power. Because of these economic facts, designers are attempting to use multiple processing units in one system by exploiting parallelism in the applications. Simply defined, parallelism is a situation where more than one part of the program (a single instruction or a group of instructions) can be executed simultaneously. In general, there are four levels of parallelism shown in Fig. 1-1 which can be exploited.

1. Data level. Executing an instruction on more than one piece of data simultaneously. This can be applied only to problems which are by their very nature, parallel.
2. Instruction level. Executing multiple instructions simultaneously on the same data. This is difficult to achieve since all the instructions must be synchronized to guarantee the proper order of execution.
3. Task level. Executing groups of instructions simultaneously on related data. The efficiency of this type is dependent upon the architecture, and the programmer must define parallel tasks.
4. Job level. Simultaneous execution of unrelated jobs. This type of parallelism is employed in multiuser mainframes. Unfortunately turn around time of each job is not improved by this technique.

1.2 Motivations

Research in parallel architectures has concentrated on three types of machines, with some success in two areas. The first, and most popular motivation is to build a supercomputer. Current architectures suffer from the Von Neumann bottleneck which defines a ceiling on their performance. Parallel architectures are pursued to eliminate the bottleneck and improve the overall performance The second motivation is that some applications are inherently very parallel. This has met with the most success and is common in DSP and graphics applications, but hasn't helped advance the field of general purpose computing. The third and least successful motivation is to decrease the cost of



INST1	DATA1	DATA2	DATA3	DATA4
INST2	DATA1	DATA2	DATA3	DATA4
INST3	DATA1	DATA2	DATA3	DATA4
INST4	DATA1	DATA2	DATA3	DATA4
INST5	DATA1	DATA2	DATA3	DATA4
INST6	DATA1	DATA2	DATA3	DATA4
INST7	DATA1	DATA2	DATA3	DATA4
INST8	DATA1	DATA2	DATA3	DATA4

Figure 1-1:
Three ways of dividing serial code into parallel code.
Horizontally by instructions, or vertically by data.

computing power. This has been pursued with little success because the decreased cost of the computing power is offset by the increased cost of software development.

In this thesis a parallel architecture will be examined which attempts to satisfy three criteria.

- Take advantage of advancing VLSI technology.
- Easy to program for general purpose applications.
- Fault tolerant.
- Flexible enough in design and implementation to eliminate obsolescence with each advance in VLSI technology.

Chapter two of this thesis is an examination of the problems and major decisions that affect parallel processing. It presents a methodic study of the basics with the purpose of showing what is necessary for a successful parallel architecture. Chapter three describes our implementation of the architecture. Chapter four is a summary of current and future work.

Chapter 2
THEORY of PARALLEL ARCHITECTURES

SOFTWARE

2.1 Description of Parallelism

The most difficult aspect of software design of a parallel processor is the description of the parallelism in the algorithm. As mentioned in Chapter 1, there are four levels of parallelism, three of which, data, instruction, and task level, are well suited for execution on a parallel architecture; and the forth, job level, requiring interprocess communication is suited for loosely coupled conventional computers such as the VAX 8900 [1].

Several approaches have been tried to describe parallelism. One approach is to leave the high level language software development unchanged and to use a compiler which finds and defines parallelism. This method is most desirable because of the easy transition from serial to parallel programming but has several limitations. The success of this approach for scientific applications is illustrated by Allient computer's Fortran compiler which parallelizes DO loops. Unfortunately, this is unlikely to affect general purpose programming great deal since the compilation becomes expensive for more sophisticated automatic parallel translation, and designing an efficient compiler is very difficult. The fundamental drawback of this approach is that it relies on software (the compiler) to replace the programmer's intelligence to in recognize parallelism.

The coarse grained parallelism may be best defined by the programmer through special constructs in the programming language. This complicates the programming, but significantly reduces the compilation time and run time overhead [2].

2.2 Programmability of Parallel Architectures

Parallel architectures have been intensely researched with very little success in the area of general purpose computing. This is primarily because of the incompatibility of current architectures with current serial programming techniques. As a result, there are are a lot of parallel architectures available, without appropriate software support. Thus, the primary objective in developing a new architecture should be programmability, rather than merely the most efficient use of the largest number of processing units. Efficiency improvements are traditionally an evolutionary process, and therefore can be sacrificed initially for an easier transition to parallel programming.

Software for a parallel architecture must conform to new constraints. The job must be partitioned into instructions that can execute in parallel (as already discussed); the instructions must be scheduled for execution on a processing unit; and the data flow must be accounted for. Scheduling implies keeping track of the progress of a task (instruction or chunk of instructions) with respect to time. Thus it is responsible for ensuring that the various portions of a program begin execution at the appropriate times so that the data requirements of each portion are correctly satisfied [3] [4]. To solve the problem of scheduling, most often special parallel languages are used. This increases the efficiency of the program, but requires the programmer to abandon the languages and techniques he is comfortable with.

In parallel architectures having more than one memory (which is required to solve the Von Neumann bottleneck as explained below), one has to deal with the problem of keeping all the data residing in different memories consistent. This is referred to as the problem of data coherency [2]. One way of understanding the problem is to look at the structure of memory at two architectural extremes. One extreme is only one memory and more than one processor. This configuration creates a bottleneck because most of a processors time is spent awaiting for the bus. The other extreme is to have a separate memory for each processor, with each data element in one and only one memory. This configurations suffers from wastage of processing time spent in transferring data so that it is in the correct memory to execute. The only solution to this dilemma is to allow multiple copies of data elements to reside in more than one memory or cache simultaneously. In this case, there has to be a way to ensure that data in a memory is correct when it is used. If all the data is corrected continuously, it would be no different than a single memory albeit multi-ported. Thus the improvement in the execution performance forces one to deal with the problem of data coherency.

These last two problems are often grouped together as the synchronization problem, and are responsible for holding back the realization of parallel processing in all but special purpose machines. There are several approaches to handling the synchronization problem. One is to use special constructs within a conventional language such as TEST AND SET [5] [6]. This instruction allows one process to access shared memory and check for other processes using a data elements it is interested in. It also allows the processor to set a semaphore saying that it is using the data [data mov. prim. article]. Another approach is to use special language structures such as Guarded Horn Clauses which have

Test and Set functions Implicitly built in [7] . Both these approaches require the programmer to specify the parallelism and coordinate the execution of instructions. A third approach is to use a whole new model for computing such as data flow where the programmer implicitly describes the parallelism and the operating system coordinates the execution dynamically.

2.3 Data and Task Flow

Because of the limitations on the Von Neumann architecture, a new architecture has to be developed, and many researchers believe that first a whole new paradigm for computation has to be developed [8]. Several models have been proposed and one of the more promising ones is the data flow paradigm. As opposed to a Von Neumann algorithm, where instructions are defined sequentially and executed in order, data flow programs define a group of instructions with no explicit order of execution. Instead, each instruction specifies data elements and is executed as soon as the data becomes available (a more sophisticated and potentially more efficient variation is demand-driven data flow, where execution starts at the final data specifying or tagging instructions which need to be executed and once all necessary data is at the input level, actual execution begins.) An example of the data flow execution of the quadratic formula is shown in Fig. 2-1 because each instruction is available to execute as soon as it's data is ready, data flow algorithms are ideally suited for parallel execution [9] [10].

In numerical computations, where the data is structured in arrays data flow is very attractive, but in general purpose computing the fine grain of the algorithms is impractical. The advantages to data flow are that it can extract all possible parallelism from any problem and can execute independent of the

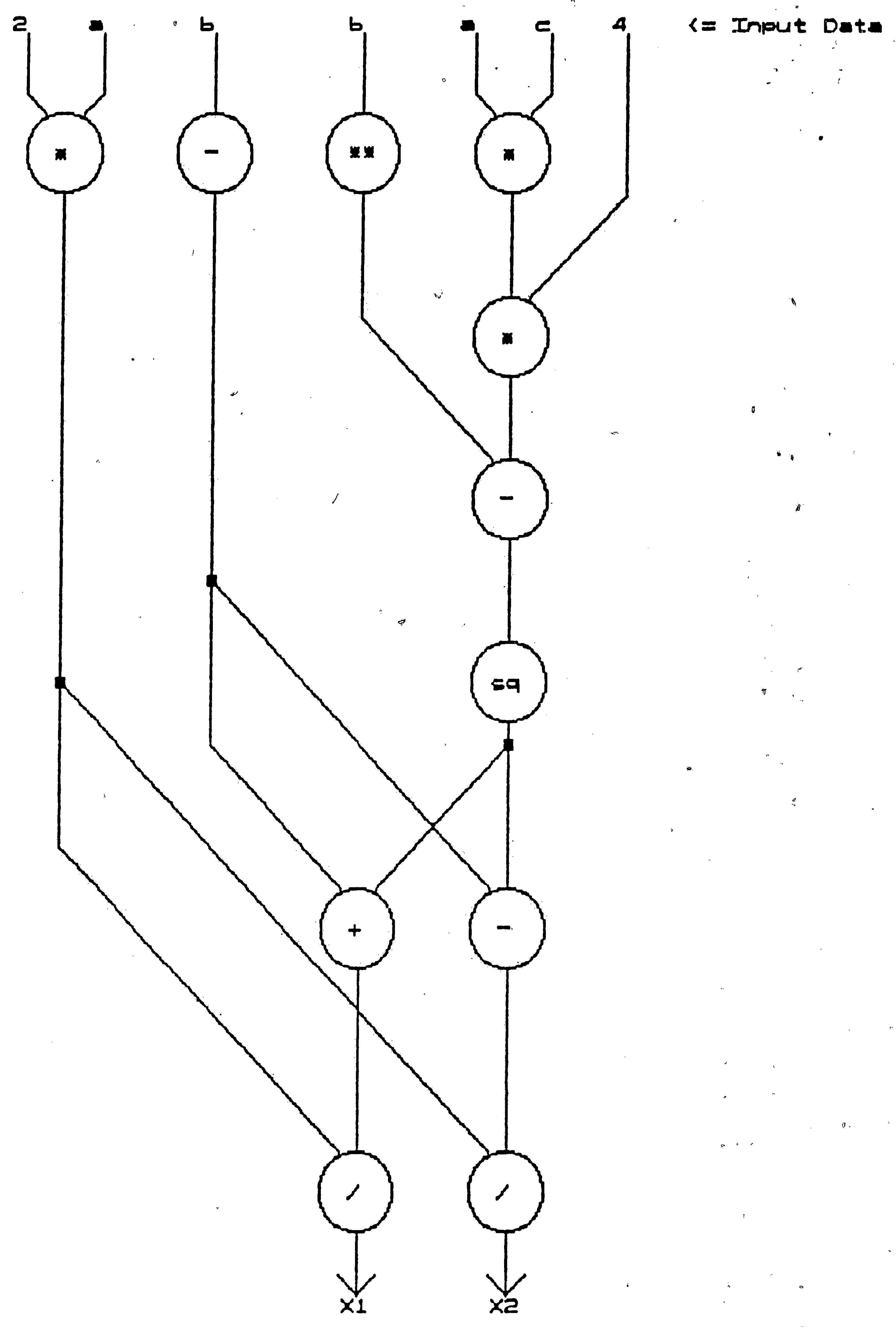


Figure 2-1:
Data flow execution of the quadratic formula.

number of processing units in a system. The disadvantage, however, is that it develops a fine grained parallel algorithm which is not suited for general purpose computing. As can be seen in the above example, if a problem is subdivided with each instruction executing in a different processing unit there will be a large amount of inter-processor communication to transfer data elements. Even with a very sophisticated operating system which has instructions execute in the processing unit where their data already resides, their will be more than one data or instruction transfer for each instruction executed (see Fig. 2-2)

In the Von Neumann model of computing control is specified by the location of the instruction with respect to other instructions. This is called a control flow paradigm because each instruction executes when control is passed to it from the previous instruction. In the Data flow model of computing, control is specified by the data elements required by the instruction to execute. This has the disadvantage of being difficult for humans to visualize because of its complexity. By its very nature, there are many branches of executing code and although the parallelism does not have to be explicitly specified by the programmer, it is very difficult to program and debug.

A golden mean between sequential computing and data flow is what is referred to as the task flow. Task flow is a model where a program is broken into tasks containing tens of instructions. within a task the program executes with a normal flow of control by location. Each task however, specifies what data is necessary for it's execution and the tasks are heaped and execute when the data becomes available. This makes the tasks easy to develop and at the same time decreases the communication between executing tasks.

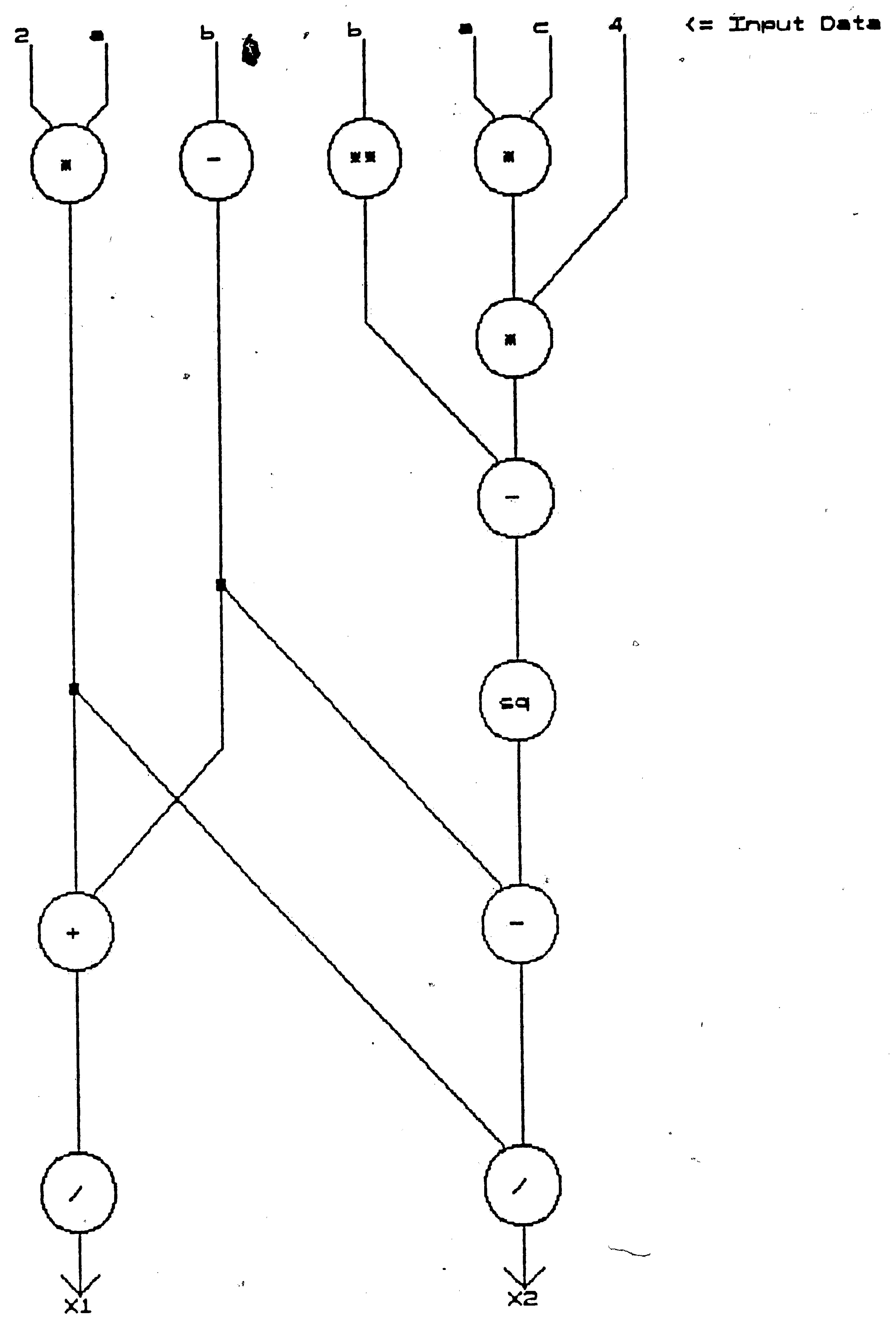


Figure 2-2:
Data flow execution with data reused.

2.4 Overhead of a Parallel System

Overhead is defined as any time during the execution of a job that a processor is not executing instructions specifically defined by the programmer. In a conventional computer, this includes performing operating system functions, I/O, and even the instruction fetch cycle if the processor is not simultaneously executing an instruction. In a parallel system there are several additional operations which add to the overhead burden, these include [11]:

- transferring data between processors or from main memory to local memory.
- Transferring instructions to the processor in which they are to execute.
- communication between processors including synchronization.
- Other operating system functions such as heap management and I/O which are complicated by the parallel environment.

Conventional microcomputers have reduced the overhead by using pipelining and cache memories and in developing a parallel system, overhead reduction must be considered at the outset. In a data flow architecture, techniques such as pipelining can be used on the local level since each task will execute as a serial program. There will be three major causes of overhead, specifically:

1. Data transferring. Data will need to be transferred from the main heap to local memory at the beginning of each task, likewise, data will have to be transferred back to the main heap at the end of the task taking care not to overwrite any new data with an older version. This portion of overhead can be reduced by reusing the data in local memory in some instances.
2. Instruction transferring. Instructions will have to be transferred to the local memory at the beginning of each task. A way of reducing this portion is to have a task execute in a processing unit where its instructions already reside if possible,or having local memories large enough to contain the entire program.
3. Heap management. Before a local processor can start executing a task it must update the problem heap, search for a new task to

execute, and spawn new tasks to add to the heap. This requires some searching which can be very time consuming.

2.5 Task Execution Schemes

A task is a group of several instructions, and because of this, a task can be partially executed when the program control goes to another task. This is unlike Data flow in which by definition an instruction must be ready to execute completely before execution is allowed to begin. A form of task flow could be defined which requires the same constraints as data flow, but this would inhibit such programming tools as functions and nesting, and would cause more overhead by causing tasks to be smaller, self contained units. Because of these complexities, the basic form of a task, and allowable paths of control must be defined.

A task must have a beginning and an end. Execution must start at the beginning and must stop at the end. This is similar to the definition of a subroutine or procedure in a structured language such as Pascal. A task can call another task at any time during its execution. This will complicate the operating system considerably; but as explained above, it is an essential complication. The beginning of a task must define which data is required to execute the task, and the end must define which data needs to be updated in the main heap. Other aspects of task flow control will be considered in chapter 3, but for now enough is known about task execution to develop suitable hardware.

HARDWARE

2.6 Bus Structure And Connectivity

The problem of determining the best architecture for a parallel processor is really the problem of determining the best interconnection scheme between processors. This decision affects every aspect of the machines performance, from cost and reliability to programmability and overhead. Most of the connection schemes currently being researched can be divided into two categories although there are almost as many variations as there are people working in the field. The first category is a simple extension of the standard Von Neumann architecture with a main bus which all processors must use to communicate with each other and access main memory. This suffers from an aggravation of what is known as the Von Neumann bottleneck; because all processors are using one bus, they will often have to wait to be granted access. The second category requires many independent busses connecting a small subset of processing units. This is considered the best scheme for designs with many processors because the addition of processors is accompanied by the addition of busses and communication paths between processors. Although a networked architecture appears to be the most sensible way to efficiently connect many processing units, it has several serious problems with the most serious being its difficulty to program. The operating system for these machines must contend with a large amount of message passing because of their distributed nature, and a program running on these machine must be divisible into hundreds of small tasks, all concurrently executing and communicating through the network. Several companies have developed special processors with instructions which are designed to deal with the communication problem and this has helped reduce the cost of the processing units, but because they are still difficult to program, most success has been in dedicated applications where the problem is easily distributed and the communication between tasks can be matched to the bus network. Unfortunately, this type of fixed network does not serve the needs of a general purpose machine [12] [13].

Shared bus architectures are generally easier to program than networked architectures because they have fewer processing units, and because each processing unit is more independent. This implies task level parallelism rather than data or instruction level. The processing units in such an architecture should be more powerful and self contained. There is less of a bottleneck when the running job has a large portion of serial code -- code which cannot run in parallel with any other code. This is the most promising approach for a computer which is easy to program since one is not bothered by serial portions of code. However, an effort must be made to prevent common bus access from slowing down the processors and a clever programming strategy should distribute an equal amount of work to all processors.

2.7 Memory Structures

In a parallel processor the bus is used primarily to access memory for data and instructions and therefore one way to reduce bus use is to modify the structure of memory. The two most common structures are the shared local memory architecture and the shared global memory with cache [14].

Shared local memory architectures divide up memory into partitions with each processor having direct access to one partition and access to other partitions through the bus (see Fig. 2-3.) This is very similar to networked architectures, and suffers from many of the same problems. Specifically, if a processor is running a task requiring access to memory locations through the bus, it slows down the complete execution. However, these architectures have the advantage that since all processors are working on the same information, the data coherency problem is greatly simplified.

Shared global memory architectures have a single memory, which all processors have equal access to (see Fig. 2-4.) To reduce the strain on the bus, they also have a cache for each processor where information that a processor is using is temporarily stored. The drawback of this scheme is that there can be more than one copy of each datum. If any data is changed in the cache, the data must also be changed in main memory and in any other caches that have a copy of that data. Several people have used this approach, but they have had to develop complex hardware to solve the problem of cache coherency [2] and there is still no guarantee that this type of architecture will be easy to program. For an architecture to be easy to program, the data structures within the software must easily map to the hardware -- either because of the highly limited software, or because of very flexible hardware.

Certain dedicated signal processing and graphics problems are more suited to parallel processors because their data structures match the structure of the processors. General purpose computing, however, has no predefined structure to its data at the instruction level. The most structure that can be guaranteed is at the subroutine level in a structured language such as Pascal or C. In these languages, all data defined in a procedure is global in that procedure, and in all procedures called by it. No data (with minor exceptions) is global at any higher level, and all data defined in a procedure is lost when that procedure reaches its END statement. Since this amount of structure already

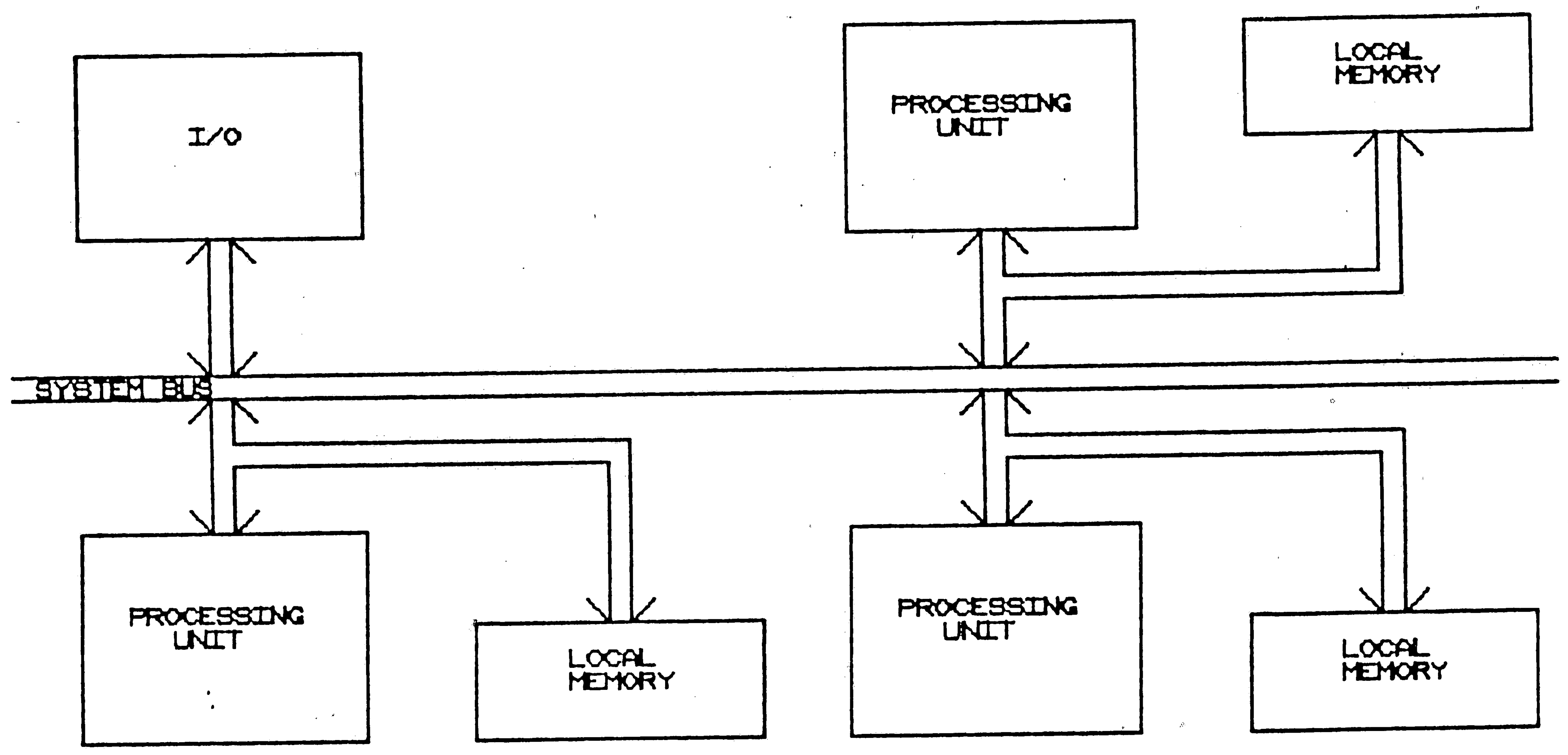


Figure 2-3:
Shared local memory architecture.

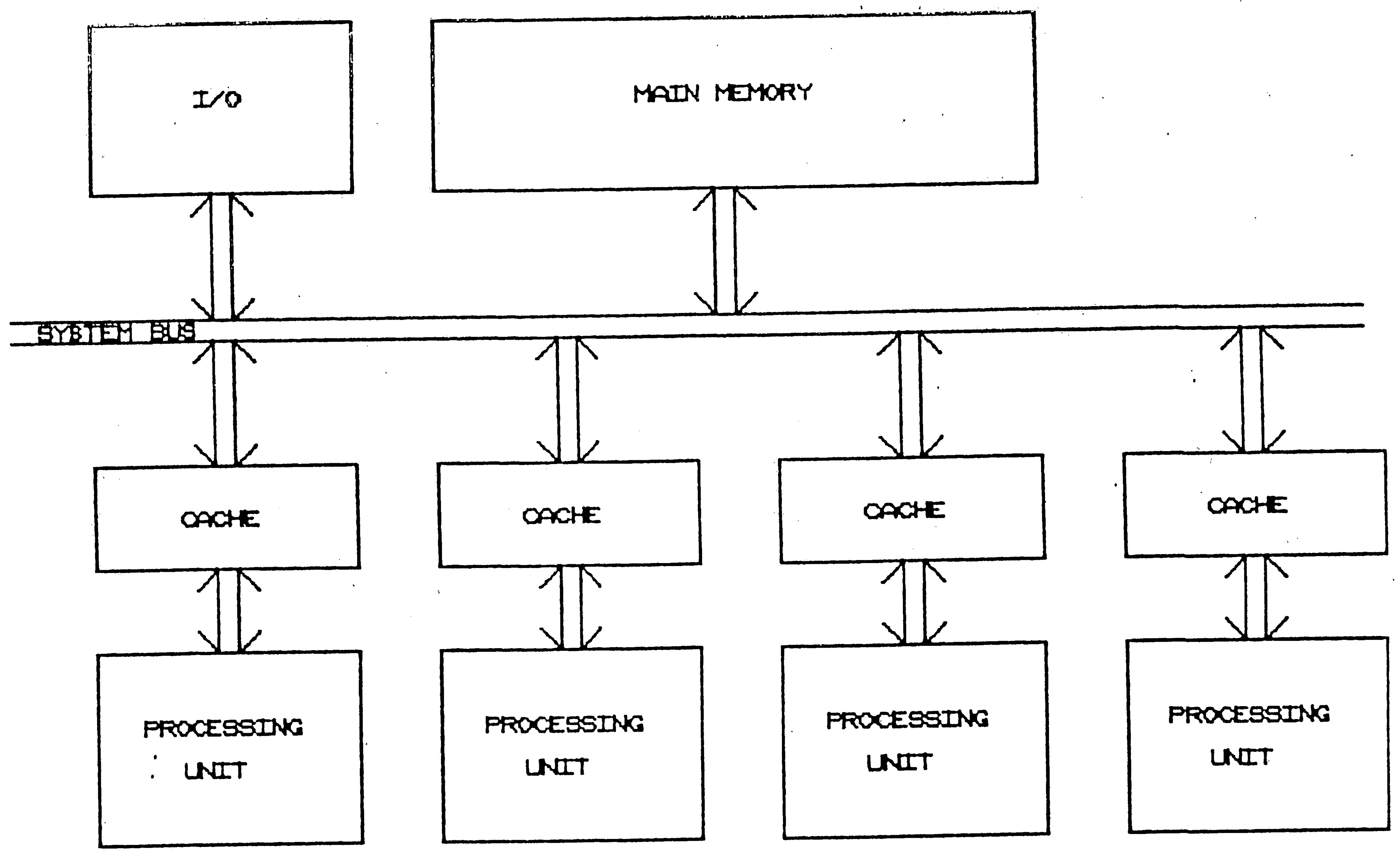


Figure 2-4:
Shared main memory architecture.

exists in common programming practice, it would be ideal to develop an architecture which took advantage of this structure.

Such an architecture should have a main memory where all the code and global data is stored and a local memory where an executing task is stored along with the data (see Fig. 2-5.) Since main memory only contains global data, is is only updated upon completion of a task; and since local memory contains primarily local data it's contents are no longer relevant when a task reaches its END statement. This architecture would take the fullest advantage of parallelism already identified with current programming techniques and languages.

2.8 Supervising and Control

There must be a way of controlling all the interactions between processors, and there must be a way -- either through software or hardware -- of knowing the exact status of every processor and task in the system. This supervisory function includes communication with the outside world, initialization of the system, bus arbitration, job scheduling, and keeping track of data. In a shared bus architecture, all of these functions can be either distributed or centralized and in either case, they can be performed in hardware or software. One common approach is to specify a single processing unit (PU) as "supervisor." Such a PU has a special architecture and is dedicated to all the above mentioned tasks with the exception of those that require excessive speed such as bus arbitration.

To implement the supervisor, an analysis must be made of each function, examining each of the three alternatives: software, dedicated processor, and

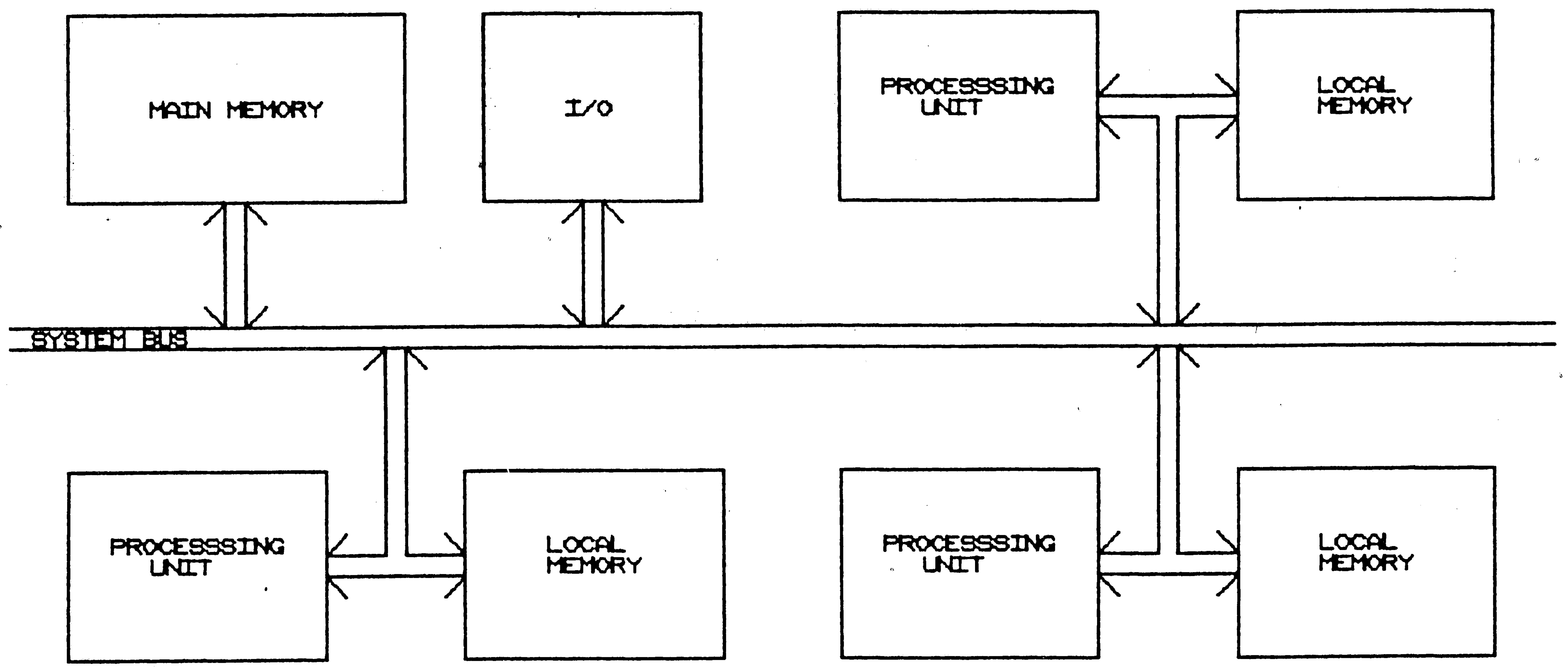


Figure 2-5:
Architecture for task flow machine.

special circuit. Communications to the outside world will obviously require a dedicated interface to a high speed communication link, but the control is probably best performed through software running on a processor -- not necessarily a dedicated one. The system initialization can be performed by any processor as long as there is a circuit which gives it preference over the other processors, and as long as it has an access to the initialization software which resides in ROM. Bus arbitration is the most critical function and therefore must be performed by a special circuit to reduce the access delay time and increase the bus efficiency.

2.9 Bus Arbitration

Bus arbitration in a shared bus system is a critical function because the efficiency of the bus affects the number of processing units which can effectively share it. A shared bus architecture must have a high bandwidth bus, which is used efficiently and equitably. The arbitration scheme must be optimized for the type of access which the memory structure and operating system require. In a shared global memory system the average packet size is proportional to the size of the caches. On the other hand, a dual memory system, as is suited for a task flow environment, the packet size is very large and the access frequency is small. This is desirable because high speed transfer techniques such as direct memory access can be employed in this situation to improve the bus performance.

There are two important considerations in determining the arbitration scheme for a system. the first is prioritization of processes and/or processors. In a task flow environment, running on a shared bus system where all processors are interchangeable, and where all processes are considered equal, One may use arbitrary prioritazition. The simplest scheme would be to assign a fixed priority level to each processor and more sophisticated realizations include rotating priority and software assignable priority. These more sophisticated schemes reduce situations where several processors are idle because they are awaiting for data from a process running on a low priority processor which is being denied access to the bus. These bottlenecks can also be reduced by careful operating system design, provided some hardware support is available.

The second consideration in the design of an arbitration logic is the implementation of the logic as a centralized or a distributed architecture. A distributed arbitration circuit is generally slower and expensive but is more flexible, because it allows the addition of an indeterminate number of processors. A centralized arbitrator, on the other hand, is better equipped to handle complicated arbitration schemes, but increases the number of bus lines needed for arbitration.

To simplify the hardware and to obtain maximum bus efficiency in a task flow environment, a fixed priority centralized arbitrator is chosen for the architecture described in this thesis. This gives the added benefit of greatly simplify the processing units by allowing a centralized bootstrapping as will be shown in chapter three.

2.10 Fault Tolerance

In a system with many processing nodes, there has to be many interconnections between nodes and interconnections are inherently very unreliable. Because many applications are totally dependent upon computers, systems are currently being designed more fault tolerant; often using networked systems, or redundant processors. Parallel systems have a great potential for fault tolerance because of their built in redundancy, but this can only be realized by careful design. For a system to be fault tolerant, it must be able to function after a communication link or processor fails.

In a networked architecture this seems easy because of their multiple communication links, but the difficulty is that there is no way for the other processors in a system to know when a fault occurs. Fault detection is necessary for the other processors to start performing the functions of the missing node, and for proper rerouting of information. Current networked systems are less reliable because of their multiple nodes. Shared bus architectures have an advantage because most can function with a variable amount of nodes. The area where work is need is in fault detection. A task flow architecture is perfectly configured for a fault tolerant system since any task can run on any node. The only provision which is necessary is a way of removing the effects on the problem heap by a processor which starts a task and cannot finish it because of a fault.

2.11 VLSI Compatibility

Advancing VLSI technology is providing microprocessors have functions and sophistication developed for mainframe computer systems. Because of these advances, a parallel microprocessor system must be very flexible and ideally, would have very independent processing units which interacted with a simple protocol and therefore the architecture would be flexible enough to take advantage of future technology. An architecture which relies on special functions for communications in a networked bus environment is not workable because it can not take advantage of microprocessors developed for non parallel systems [15].

Chapter 3
IMPLEMENTATION of FAULT TOLERANT PARALLEL ARCHITECTURE

HARDWARE

The basic structure a computer is determined by the type of application it is intended for, but once the basic structure is determined, there are still many aspects of the computer which need to be decided. These factors largely determine the cost, efficiency, flexibility and expandablility of an architecture. For a general purpose parallel processor, the best architecture is the shared bus design because its simple structure makes it easiest to program and because a smaller number of powerful processing units makes it more robust regarding software design. The chosen architecture as implemented is shown in figure 3-1. Following sections describe details of this architecture.

3.1 Processor Considerations

Current general purpose microprocessor families are very similar in computational power and type of instructions, so the choice of a processor is relegated to other areas including basic architecture, upward compatibility, software availability, higher operating system functions, and the availability of support hardware and software. A general purpose machine which is easy to develop software for should be based on a microprocessor family which already has a large user base. There are two reasons for this, one, because the transition from programming an ordinary computer to programming in parallel will be a big enough step without having to change processors. Two, because it is intended that only minor modifications will need to be made to existing

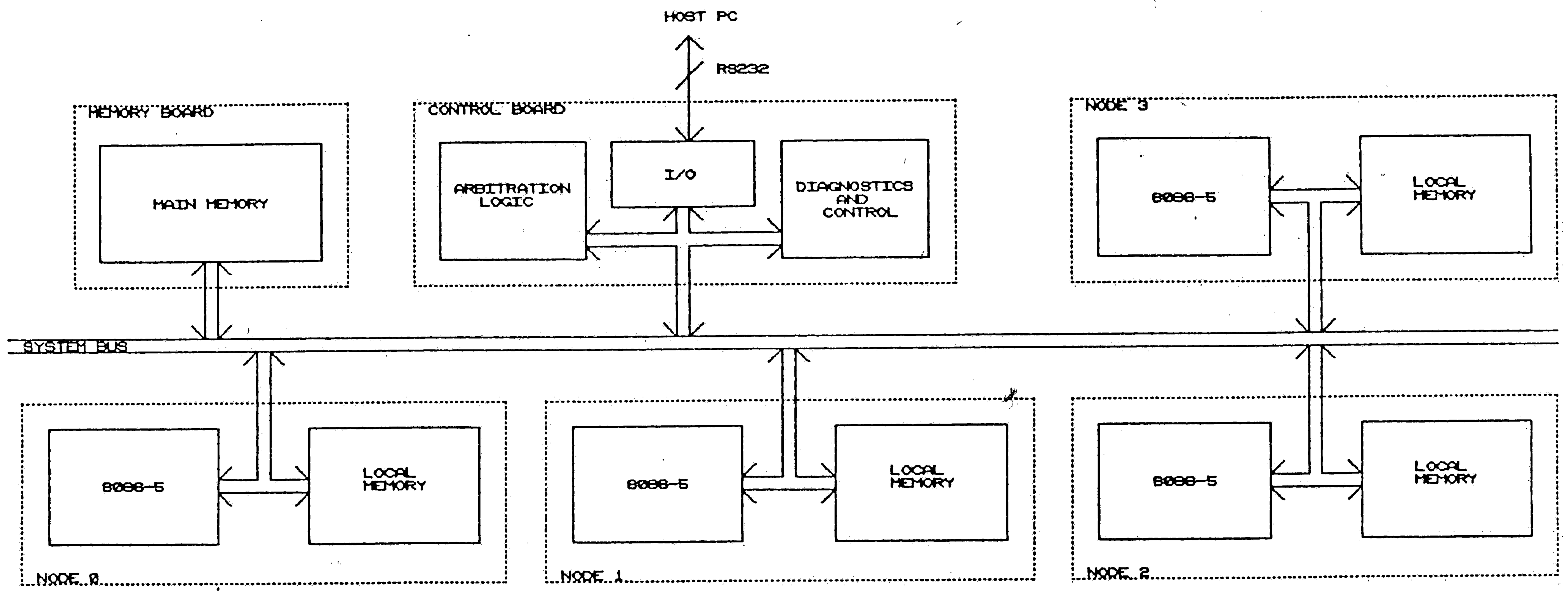


Figure 3-1:
Block diagram of system.

software to make it compatible with the task flow environment, although major modifications may be necessary to make it run efficiently.

Three microprocessor families were considered in this study -- Intel, Motorola, and NEC. Intels iapx series is notable for its popularity which guarantees low price, abundant support software, and future architectures which are compatible with current software and bus structures. Motorola has a somewhat less popular line, but it is a family dedicated to future compatibility to every possible degree. NEC's line of processors is innovative, cheap, and powerful, but as it is a parasite family relying on a petulant host, future compatibility is questionable. Upward compatibility is a very important consideration, with clock speeds increasing steadily (5M in 1981, 16M in 87) and new processors coming out regularly, any new design should be easily modified (by changing processor boards, memory) to prevent obsolescence before it has a chance to be developed and accepted. To this end, Motorola' 68000 series of processors is strongest. It is designed for complete software compatibility and bus compatibility in steps (8 bits, 16 bits,..) to encourage it's use in a modular family of computers all software compatible, and bus size dictated by specific requirements. Furthermore, Motorola has stated a commitment to future chips with higher clock speeds as the technology becomes available. Intels family, although upward compatible is not downward compatible although this seems irrelevant, mutual compatibility is desirable for three reasons.

1. Without downward compatibility, everything about a microcomputer becomes obsolete as soon as a new processor comes out (software, bus structure ...) This means that the only right time to develop a microcomputer is as soon as a new processor is introduced, the exact time when development support is least available.
2. No improvements in software can be used on machines other than the most current ones because of incompatibility. This drastically

reduces a machine's usable life by making it prematurely obsolete.

3. Everything has to be developed all at once (operating system, software, new hardware) increasing the development time and decreasing the window of opportunity.

Considerations such as architecture and operating system functions are not important in a task flow machine because as a multiprocessor, software will handle multitasking and the host computer will handle most I/O. Although obsolete, the 8086 was chosen for the first computer for ease of design and testing against known performance as a uniprocessor. Also, Lehigh is fully equipped with In Circuit Emulators for the Intel iapx series. Future work may use the Motorola Family, as it appears to be far superior to It's competitors for such an application. It should be noted here that the chosen architecture allows for the possibility of computational nodes that are very different from each other, as long as they follow the same basic structure and interface rules.

3.2 Floating Point Unit

Since most floating point coprocessors require very little interface circuitry if any, it is common practice to design the coprocessor into the processor board even if the majority of applications will not warrant the added expense of the processor. These systems can be left with an empty socket on the processor board which can always be upgraded by inserting the coprocessor. This is the best approach also for a multiprocessor system because although the coprocessor alone will cost more than the entire processor board, Floating point intensive programs will run five to twenty times faster with the coprocessor. It would not be logical to design a system with 10 processors which is slower than a conventional machine with a coprocessor for this class of applications.

The decision of which applications warrant the expense of coprocessors is however somewhat different than for the case of a uniprocessor. The speedup in execution time will be proportionally the same for a multiprocessor, but the the cost of adding coprocessors will be n times greater, where n is the number of processors. This assumes that the amount of parallelism doesn't change with the change in software needed to support a coprocessor -- which will not necessarily be the case, and in fact will have to be researched for specific applications to even get a qualitative feel for the effect.

It is possible to add coprocessors to only a limited number of processors nodes in a parallel machine. However, in this event, the operating system would have to try to route floating point intensive tasks to processors equipped with floating point units. Even though this is the best way to incorporate more floating point power, this would increase the software complexity and the job overhead. Floating point coprocessors are not incorporated into the first prototype machine because the additional expense and development time will do little to prove the effectiveness of a task flow architecture, but treatment of special processing units will be examined in the operating system design later in this chapter.

3.3 Memory Structure and Bandwidth

Since the number of processors which can efficiently use a single shared bus depends on the bandwidth of that bus, it is important to keep that bandwidth as high as possible. There are three basic limitations to bus bandwidth with only one of them inherent to the bus itself. That limitation is the delay caused by the tranceivers and wires of the bus but this theoretical maximum bandwidth is rarely reached. The second limitation is the access times of the memory or I/O device receiving or transmitting the data across the bus and the third limitation is the cycle time of the processor controlling the transfer.

The number of processors should be determined by the amount of parallelism which can be easily extracted from the average general purpose program. This cannot be calculated with any accuracy given the range of software run on a general purpose machine, Therefore the only effective procedure is to predict a range with an upper bound determined by the overhead and then tailor the system to specific applications by adding processor nodes where needed, and supplying overhead reducing techniques such as direct memory access. Current version of the architecture has four nodes based on Intel 8086.

One way of increasing the effective bus bandwidth is to use the Direct Memory Access (DMA) technique to transfer data along the main bus, usually between main memory and local memory. DMA is essentially a circuit which performs the block move function which is an instructions in most microprocessors; but the dedicated hardware of a DMA circuit can utilize the maximum bus bandwidth limited only by the memory access times and the bus delays. This avoids the repeated instruction fetch and cycles of the microprocessor, and can increase the transfer rate by two to four times. The transfer rate can be doubled again by incorporating the DMA into main memory and separating the address bus under DMA control as shown in fig. 3-2. This enables the DMA to simultaneously perform a read and a write to a different address in each memory. This technique is not used in uniprocessor systems because they only have one memory and therefore cannot separate the address busses.

3.4 Fault Detection and Tolerance

A system is considered fault tolerant if it can continue functioning after there is a hardware or software failure somewhere in the system. Since it is impossible to guarantee that a fatal fault will not happen, there is no such thing as a "fault proof" system. It is not practical to anticipate every possible failure, and high reliability can be obtained by simply preparing for the most common ones. For a system to be fault tolerant it must be designed with that in mind from the outset. Such considerations often include:

1. Design to reduce low reliability components such as mechanical interconnects, and simplify critical functions to increase reliability.
2. Incorporate redundancy for critical functions.
3. Provide fault detection circuitry which is capable of detecting faults as they occur, and correcting any damage that may have been done to the operating environment.

The shared bus data flow architecture is inherently fault tolerant because of the problem heap which provides an easy way to bypass faulty nodes. To further increase the fault tolerance, the system was designed without designating any single processor as a master processor in charge of the system critical functions. Instead, the controlling functions were implemented either through dedicated circuits (to whom normal fault detection techniques are applicable), or through software functions which run on any processor when it has access to the main bus and memory. All scheduling and control tables are stored in main memory where any processor can access it. Thus if any node encounters a fault, it would not stop the operation of the entire parallel system but would only affect the speed of the system.

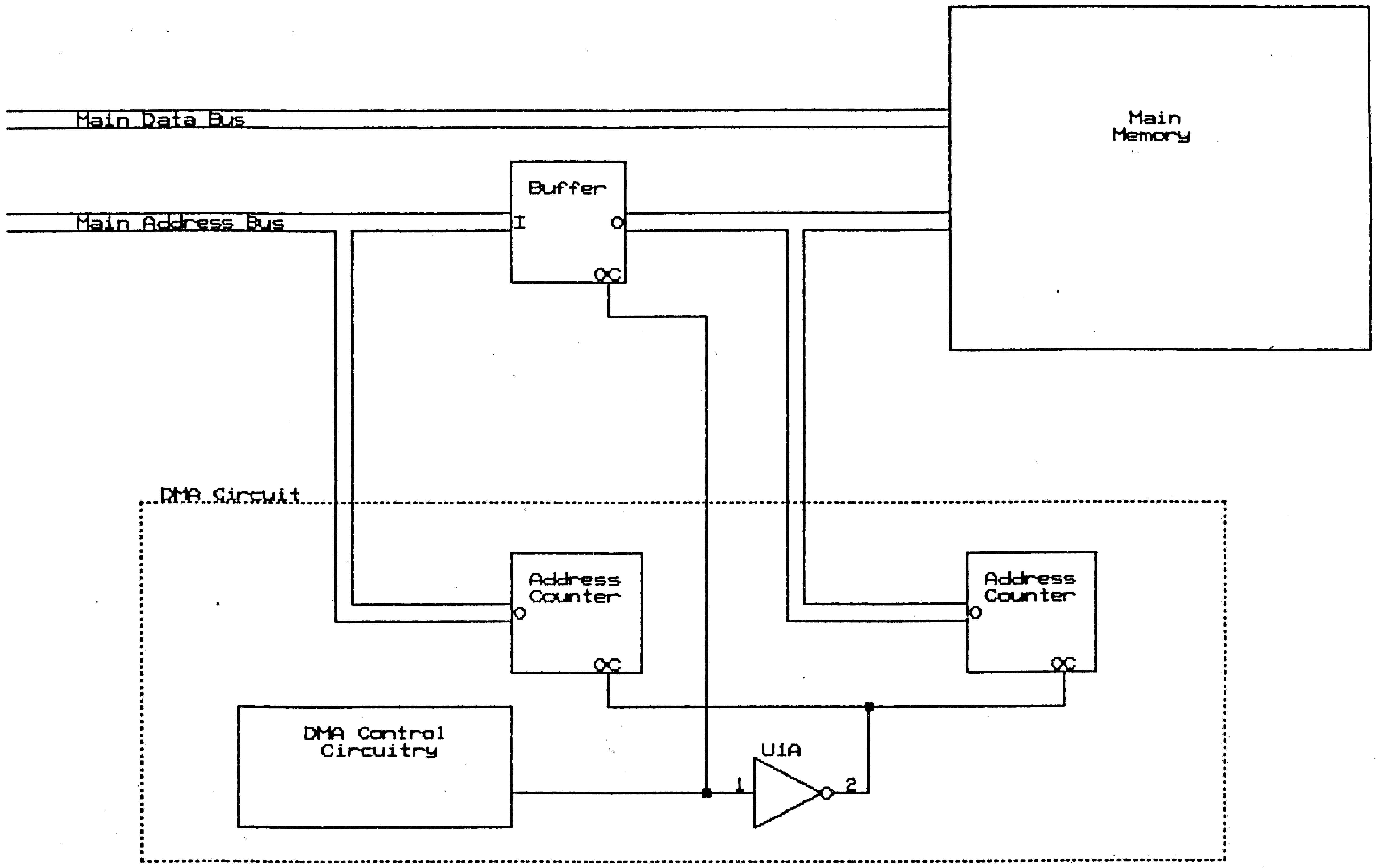


Figure 3-2:
DMA circuit for direct memory to memory transfers.

3.5 Host Interface

To prevent parallel machines from becoming I/O bound, there must be a high speed interface between it and the host computer. The speed of the interface is related to the task execution speed of the parallel machine so this is another parameter which could determine the upper bound on the number of processors. Thus, the interface should be upgradable when the need arises.

Our system has been built with an RS232 serial interface to reduce design time and cost. This is not an effective interface because of its slow speed, but operating at up to 19.2K baud, it is sufficient for testing a prototype.

3.6 Arbitration Logic

The arbitration circuit used in the system is shown in schematic C2. It was built for eight processors, but the hardware is easily expanded to handle more. The bus requests (BRQ\) come in to the control board, are prioritized and then the output bus grants (BG\) are latched until the single processor granted access is no longer requesting use of the bus. This is done with the feedback in the lower portion of the circuit. U19 receives the address of the processor using the bus, and that BRQ\ input is piped to enable the clock to the latch. This arbitration scheme can be overridden by the control signals Boot and Real. The Boot signal disables the outputs of all the latches and gives control to pullup resistors which are preset so only one processor is given access during boot-up. The Real signal enables latch U18 to control the bus with a byte written to it by the initial bus master. The use of these signals will be explained in the next section.

To reduce unnecessary bus accesses, there is also a signal called Token generated by the arbitration circuit. This signal is a positive pulse whenever a processor relinquishes the bus. This signals idle processors that there may be new tasks available to them so that they don't have to continuously access the bus looking for work. Token is initially given to the highest priority processor and is passed along to successively lower priority processors if the higher priority ones are in use. Using this scheme, only the one highest priority idle processor checks for work, and only after each access by a working processor.

3.7 System Initialization

When the system is first powered up, the individual processing units have no operating system kernel and therefore, they all must be disabled except one which will be given access to the ROM on the central memory board. This is done by U7 on the control board, which makes Real low, and Boot high upon reset. With this set of inputs, the arbitration circuit gives control to to the Resistors on the BG\ output signals. Processor 0 is given access to the ROM and initiates communication to the host. Once the host loads down the operating system to main memory, Processor 0 raises Real, and gives bus access to all the processors in the system. Because Boot is still high, all the processors except 0 are in the hold state. Processor 0 then performs a block move of the operating system kernel from main memory to its own local memory, and because of the tranceiver control logic, the information is simultaneously written to all the local memories in the system. Upon completion of this task, Processor 0 reverts access back to itself exclusively, and lowers Boot. At this time, the system is operating normally, and control is given to the highest priority processor (0) to start execution of the job.

SOFTWARE

The primary function of the operating system in a task flow system is managing the problem heap. This combines techniques common to data heap management in a serial machine, and data flow control used in data flow machines. There is also the added problem of accounting for data and transferring information to and from the local processors. To understand the structure of the operating system, it is first necessary to understand the functions associated with handling the tasks, including possible bottlenecks and overhead. For the purpose of the operating system, each task has a structure characterized in table 3-1. The multiprocessor built during this project exploits this structure to provide the task execution sequence, as explained in the following sections

3.8 Task Management

Any structured language program with Minor modifications can efficiently run the task flow machine explained in this chapter. Programming techniques will have to optimize parallelism; other than this, only a definition of dependencies and the task structure outlined in table 3-1 will be necessary for source code to be recompiled for the task flow architecture. Compilation will perform the standard functions such as parsing, translation, relocation, and linking but also should generate a physical task list. This task list typically consists of following information.

- The location of each task for loading into local memory.
- Specification of data that is used by the task for loading into local memory.
- Specification of data that is produced by the task for loading into main memory upon completion.
- Identification of the mother task and any daughter tasks.

Table 3-1: Defining task structure

1. A task has a beginning and an end. Program control must start at the beginning, and finish at the end.
2. Each task (except the main task) is contained within another task called its mother task And the contained task is labeled the daughter task.
3. Tasks with the same mother are sister tasks.
4. Tasks are born when their mother task starts execution and they can then be listed on the problem heap.
5. Once born, a task can only execute if all its dependent tasks are completed and off the problem heap.
6. A task can only be dependent upon its sister tasks (It cannot be dependent upon its daughters.)
7. A task can contain (or make a call to) any other task, including itself.
8. A function is similar to a daughter task, but it can be dependent, and therefore must be attached to its mother so as not to delay execution.
9. All data defined in a task is global only to its decendents.

- List of all the functions required by the task.
- Specification of any special requirements such as extra memory, or a floating point unit for systems which are not homogeneous.

In addition, the physical task list also contains the current count, or the number of the most recent logical task. The task list governs the task execution in our multiprocessor architecture.

At the start of execution, main memory contains the program, the task list, the data heap, and a task heap which contains one element -- the main or initial task. As execution progresses, logical tasks are added to the task heap, executed, and removed. To understand this process we must examine the life of a task.[1]

When a processor is idle, it searches the task heap for an available task. Upon finding a task which is not being executed, it sets a flag indicating that it is executing that task, loads the physical task into its local memory, loads all necessary global data, and begins execution. The first step in execution is the creation of new logical tasks to be added to the problem heap. Once this is done, the local processor can relinquish the bus. After execution the processor regains access to the bus and has to update the data heap and remove the task from the heap, but this can only be done if the daughter tasks spawned at the beginning of execution have been completed. To eliminate processors from being held idle because their task cannot be removed from the task heap, there is a morgue where all tasks are listed before they can be eliminated from the heap. Processors search the morgue every time they remove a task from the heap to see if its mother is awaiting to be removed.

Removal from the heap means that a task is completed, and its resulting data has been added to the data heap. When a processor is checking to see if dependent tasks are completed to determine if a task is executable, it needs only check to see if those tasks are on the heap. If they are, then they are in progress and the task dependent on them must wait. This is a consequence of the sixth property of a task as listed. Because of this, the time spent searching is kept at

[1]Throughout this chapter, it is assumed that the local memories are not large enough to contain the entire program as is the case with the prototype machine. For systems with larger memories, physical tasks need not be transferred.

a minimum. To start a task and to end a task, the entire problem heap must be searched, but the problem heap is never very large.

3.9 Dynamic Task Spawning

A useful tool for coding repetitive sequences such as DO loops is the ability to create a logical task dynamically. After a task is loaded into local memory to execute, the system must check if it is to be repeated, and if so, its name, with an incremented count, must be added to the task heap. The new addition to the heap will be identical to the first, with the exception of the count. To distinguish daughter tasks created by these repeated tasks, each task must also contain the count of its mother task. A task is not dependent upon the completion of sister tasks created dynamically.

The second way that tasks will be added to the heap is when their mother commences to execute. At the beginning of each task there will be a list of daughter tasks which are to be added to the task heap. These daughter tasks can be dependent upon each other, but their mother cannot require any of the data produced by them. The mother however, must wait for their execution before it can be removed from the task heap. This interaction of dependencies is displayed in the example program shown in Fig. 3-3.

Execution in this case will proceed as follows:

- Start up -- Task table is initialized with task A as the only element.
- First processor starts execution of task A by listing its daughter tasks -- AA, AB, and AC in the task table.
- The second processor then proceeds by checking the task table, finding Task AA is available and not dependent upon anything and

```
 A Task                        ; Main task
    AA Task                    ; When AA begins, it puts AAA, AAB,
       AAA Task                ;      and AAC on the heap.
          ...
       AAA End

       AAB Task
          ...
       AAB End
       ...
    AA End

    AB Task (AA)
       Call AAA                ; When AB begins, it puts AAA on the
       ...                     ;      heap, but AB does not execute
    AB End                     ;      until AA is removed from heap

    AC Task (AA)               ; Dependent upon AA
       ...
    AC End
A End
```

Figure 3-3: Typical execution of a program in a task flow architecture.

therefore executable. Its daughter tasks are listed in the task table.

- The third processor will now attempt to load a task, and find that AA's daughter tasks are available.
- When task AB starts execution, it continues until it reaches the "CALL AAA" statement. At this time the processor must access the bus to load task AAA while task AB is put aside in the local memory. In this situation one processor is executing more than one task, and careful programming is required to prevent a software bottleneck.
- As processors finish tasks, they remove them from the heap and continue the process.

3.10 Data Management

With a task flow structure, management of the data heap is similar to management in a standard processor. Access to the heap is controlled by the problem heap, so that Scheduling and data coherency are not a problem for the operating system. If a process has access to the data heap, then it is accessing the correct data. Two processors do not have simultaneous access to the heap, so synchronization is not necessary. The only consideration that is not taken care of inherently, is reloading data from local memory to the heap. Care must be taken to insure that all updated data is the most current. An example being the case of an array which is loaded into several local memories, each processor updating certain elements. Upon reloading, only the updated elements can be changed in main memory [16].

3.11 System Overhead and Bottlenecks

For a system with n processors, the best possible performance would be n times that of a uniprocessor. This speed-up ratio however, is not obtained in practice. There are two fundamental reasons for this. Firstly, most problems have sections of code which must be run in strictly serial fashion independent of the machine architecture. For a problem with s% serial code, the maximum speedup possible for an n processor machine is T/n+T(s%) where T is the execution time on a uniprocessor. Sections of code with less than n parallel tasks capable of executing simultaneously will causes similar inefficiencies, although not as severe. These losses are intimately related to the application, algorithm, and skill of the programmer; so we will not go into details except to say that research in this area has shown that most general purpose programs have a sufficient amount of parallelism to be worthwhile, and as programmers become more experienced with parallel machines, this amount will probably rise.

The second type of inefficiency is caused by the architecture or operating system used in a particular machine. These losses can be further divided into Isolation losses, Parallelizing overhead, and Bottlenecks.

Isolation losses are caused by processors not having the most current information on the progress of other tasks in the system. This includes braking loss caused by a process continuing to solve a problem which is already completed by another processor [11] This loss can be reduced by programming technique, or by providing an interrupt system in the hardware.

Parallelizing losses are inherent to any parallel algorithm. This category includes time spent transferring data between processors, searching heaps, and anything else that is necessary whenever there is more than one processor executing. These losses can sometimes be reduced by careful software design, but they are generally inherent to the architecture and operating system.

Bottlenecks are the classification of losses caused by too many processors trying to do something at the same time. The most common bottleneck for a shared bus architecture is the access to the bus for a task to execute, although in more complex architectures this can include other functions. One way to reduce bottlenecks is to eliminate any unnecessary bus use. This is the purpose of the Token signal which allows only one idle processor to access the bus at a time.

The only other way to reduce bottlenecks is to lengthen the average task length so that there are fewer tasks for a given application. This is the best way for a programmer or compiler to optimize a task for a specific number of processors or a specific system architecture. As the number of tasks increases, the isolation losses decrease and the parallelizing and bottleneck losses increase. Some application require a large amount of data transfers to execute a task. for these applications, it may be advisable to decrease the task size to reduce parallelizing losses and bottlenecks at the expense of Isolation losses.

Chapter 4
CONCLUSION

4.1 discussion

In this thesis, the goals were to build a parallel microcomputer which was fault tolerant, easy to program for general purpose applications, and adaptable to new VLSI technology. To a large extent, these goals were met. The machine which was built is a shared bus architecture for course grained parallelism which makes it simpler to program than other configurations. The Processing nodes are independent, and conform to a simple interface standard, and therefore they can be upgraded independently and adapted for specific applications as new microprocessors are introduced. This allows them to realize advances from technology developed for serial microcomputers such as increased clock speed, advanced instruction sets and even sophisticated pipelining when VLSI technology advances to that level.

Since the nodes are independent and the architecture is flexible, if one processor fails the other nodes can continue functioning with only a relative loss in overall system speed. This is facilitated by the processors independently choosing tasks from the problem heap. Because of this, any processor can execute any task, and there is a record of which tasks are currently being executed by each processor.

Because of the economics of VLSI, it is inevitable that computers evolve to systems which are optimized for the limitations of current VLSI technology, yet are adaptable to the future technology. Uni-processor technology requires either that the processor be limited to the complexity that can be integrated onto one circuit, or that it require many different circuits -- usually in the form of application specific integrated circuits. Although Von Neumann architectures do benefit from the decreasing cost of integrated circuits, they are not optimized to get the greatest benefit. Most parallel machines currently being researched are optimized by having many identical processors working in parallel, however, they suffer from one or more of the following drawbacks.

- They are difficult to program which eliminates the cost savings in the hardware. This also increases the time required to develop a substantial user base -- decreasing the effective life of the system.
- They are not adaptable to new processor technology which in the near future will be developed to increase the power of serial programs.
- They have decreased reliability due to increased chip count and interconnections, without sufficient provisions for fault critical applications.
- They are not robust enough when executing serial code. This severely limits their suitability for many applications.

The task flow architecture described in this thesis was designed to overcome these drawbacks.

4.2 Future Directions

Future work on this architecture should concentrate on overcoming the short comings in the hardware, and developing more software for the machine.

The greatest limitation to the the machine is that it has only one bus and therefore cannot be expanded much beyond eight processing nodes before the bus bandwidth limits performance. A DMA circuit will help increase the bandwidth but more testing is needed to determine if typical applications will be able to use the additional processors. An expanded architecture with a complete task flow architecture at each node might be a desirable configuration to achieve more processing power without excessive bus conflicts.

Currently the machine communicates with the host computer through a serial link. for certain applications it might be desirable to upgrade this to a high speed interface. Such an interface could also provide interrupt capabilities for interactive computing.

The software developed for the system includes the system initialization, diagnostics, task flow kernel, and communication modules. Of these, the task flow kernel is a rudimentary version developed only to test the task flow ideas. Since the task flow operating system is fundamental to the concept of this architecture, further work in this direction is warranted. Work would also be required in developing compilation post-processing to organize the object code for the task flow architecture.

References

[1] J. Bond, "Parallel-processing finally come together in real systems" *Computer Design,* pp.51-74, June 1987.

[2] D. D. Gajski and J. Peir, "Essential Issues in Multiprocessor Systems," *Computer,* vol. 18, no. 6, pp. 9-27, June 1985.

[3] H. F. Jordan, "Structuring Parallel Algorithms in an MIMD, Shared Memory Environment" *Parallel. Computing 3,* vol. 3, no. 2, pp. 93-110, 1986.

[4] A. H. Karp, "Programming for Parallelism," *Computer,* vol. 20, no. 4, pp. 43-57, May 1987.

[5] J. E. Roskos and C. Hsieh, "Data-movement Primitives," *Byte,* pp. 239-252, May 1985.

[6] A. Gottlieb 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.

[7] J. Tanaka et al., "Guarded Horn Clauses and Experiences with Parallel Logic Programming," *Joint Computer Conference Proceedings,* pp. 948-954, 1986.

[8] R. Krajewski, "Multiprocessing: an Overview," *Byte,* pp. 171-181, May 1985.

[9] W. G. Paseman, "Applying Data Flow in The Real World," *Byte,* pp. 201-214, May 1985.

[10] J. L. Gaudiot and W. H. Wei, "Token Relabeling in a Tagged Data-Flow Architecture," *Proceedings of the 1986 International Conference on Parallel Processing,* August 1986, Washington, pp. 592-599.

[11] P. Moller-Nielsen and J. Staunstrup, "Problem-heap: A Paradigm for multiprocessor Algorithms," *Parallel Computing,* vol. 4, no. 1, pp. 63-74, 1987.

[12] J. P. Hayes and Q. F. Stout, "A Microcomputer-based Hypercube Supercomputer," IEEE Micro,] vol.6, no. 5, pp. 6-17, October 1986.

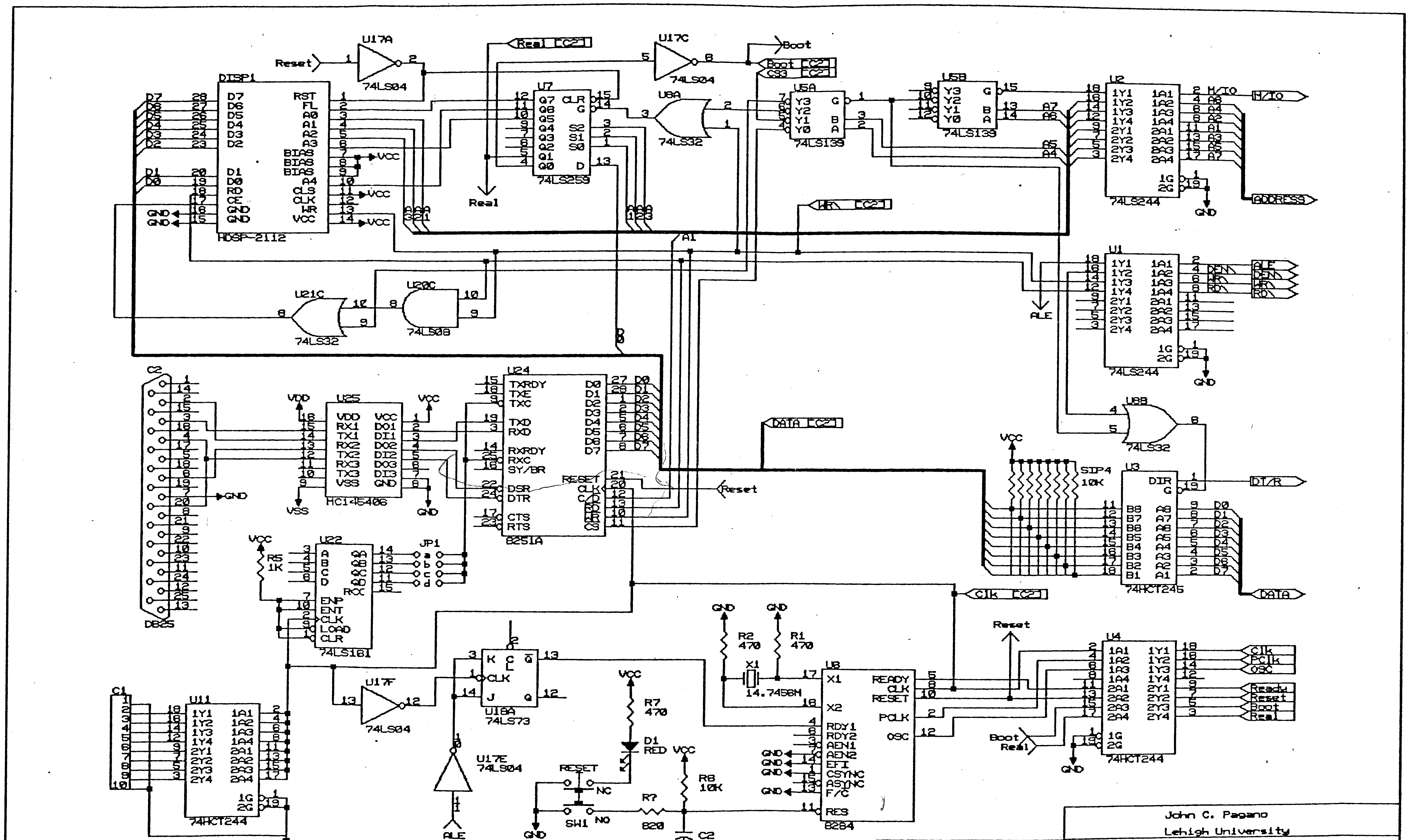
[13] J. Gaudiot et al., "The TX16: A Highly Programmable Multi-microprocessor Architecture," *IEEE Micro,* vol. 6, no. 5, pp. 18-31, October 1986.

[14] J. K. Peacock, "Application DIctates Your Choice of a Multiprocessor Model," *EDN,* vol. 32, no. 13, pp. 241-248, June 25, 1987.

[15] P. Walker, "The Transputer" *Byte,* pp. 219-235, May 1985.

[16] S. H. Bokhari, "Multiprocessing the Serve of Eratosthenes," *Computer,* vol. 20, no. 4, pp. 50-58, April 1987.

Appendix A
SCHEMATICS



JP1 settings

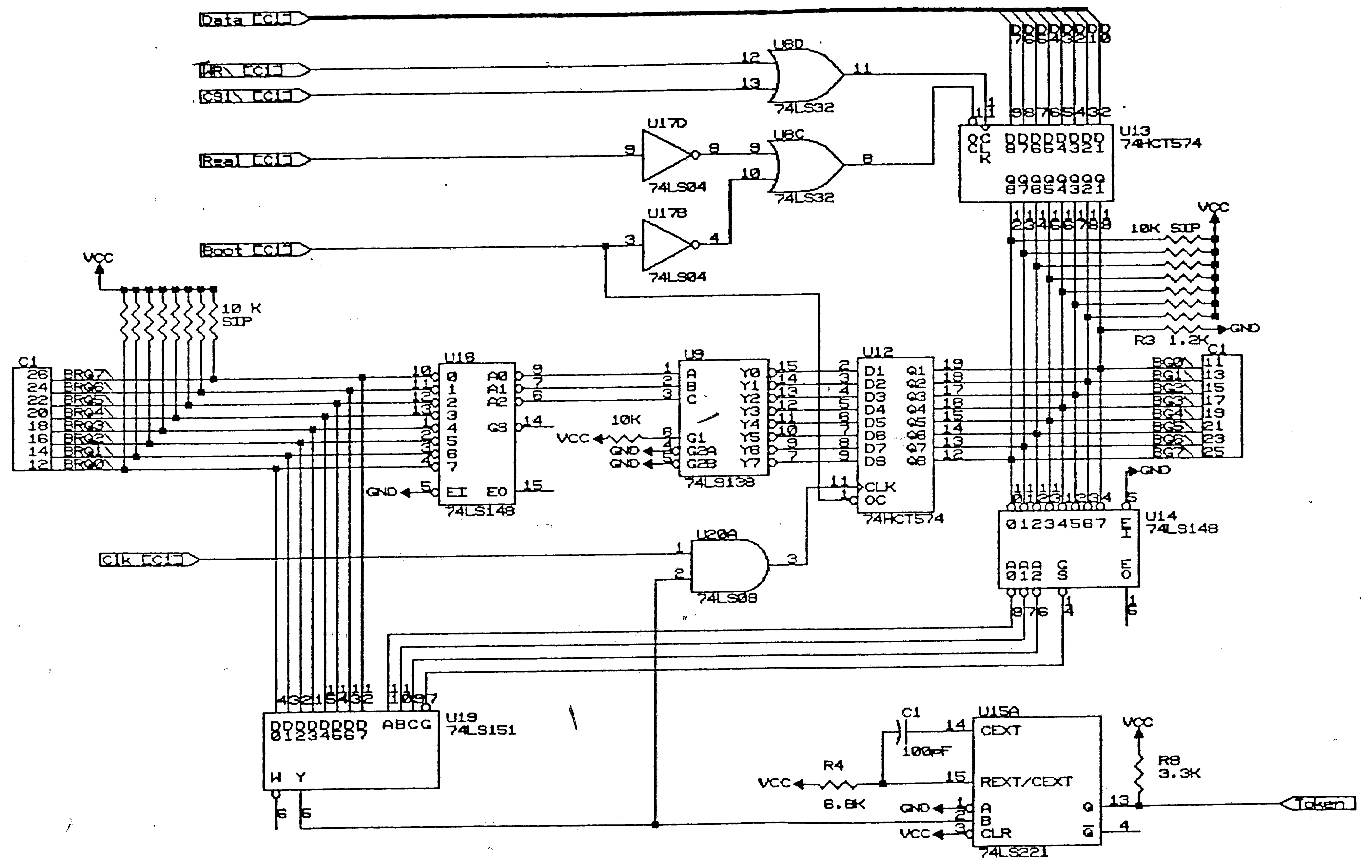
a - 19.2K baud
b - 9.6K baud
c - 4.8K baud
d - 2.4K baud

John C. Pagano
Lehigh University

Title: Control Board — Clock & I/O

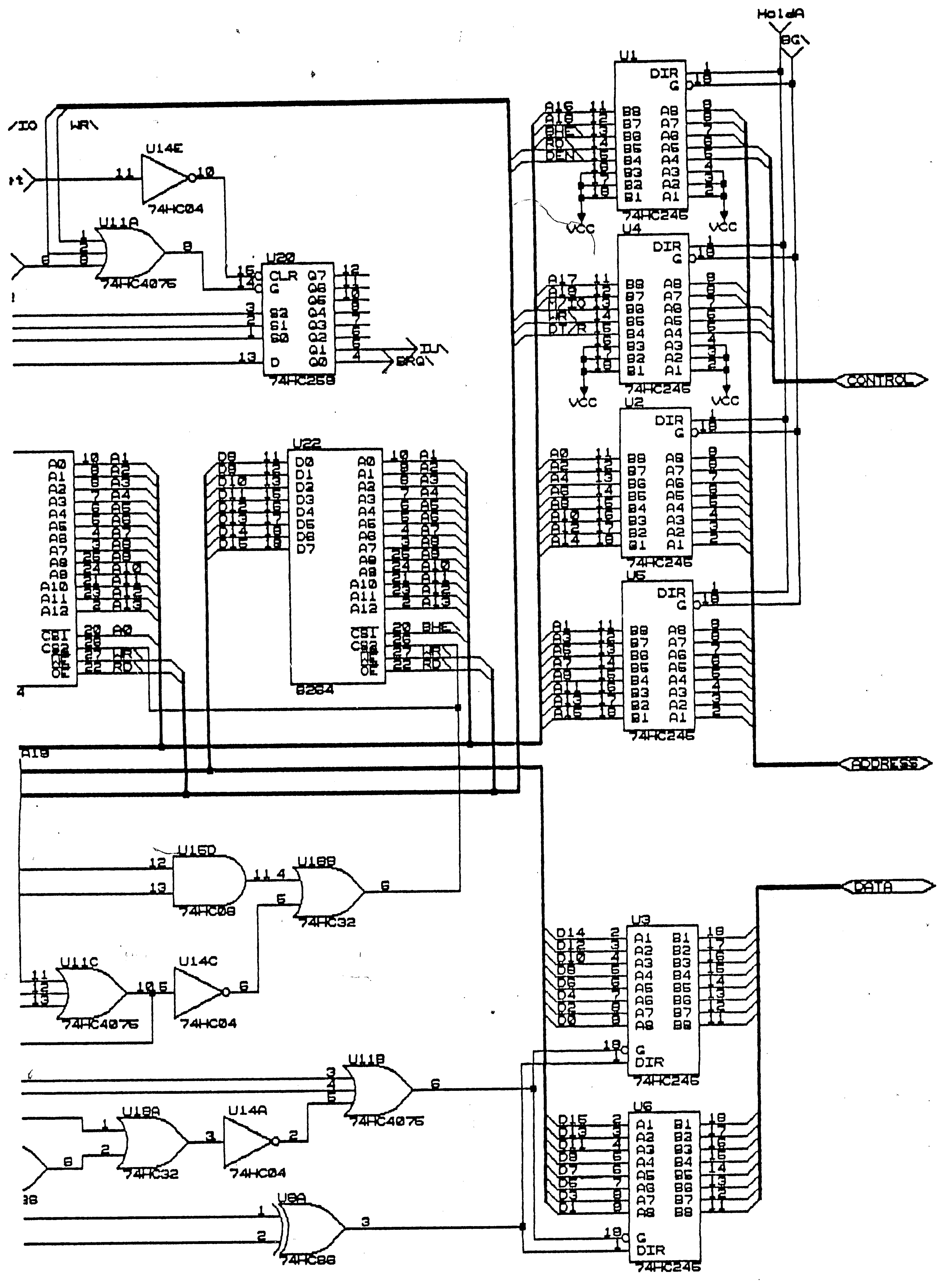
Size	Document Number	REV
B	C1	A

Date: January 1, 1980 Sheet 1 of 2

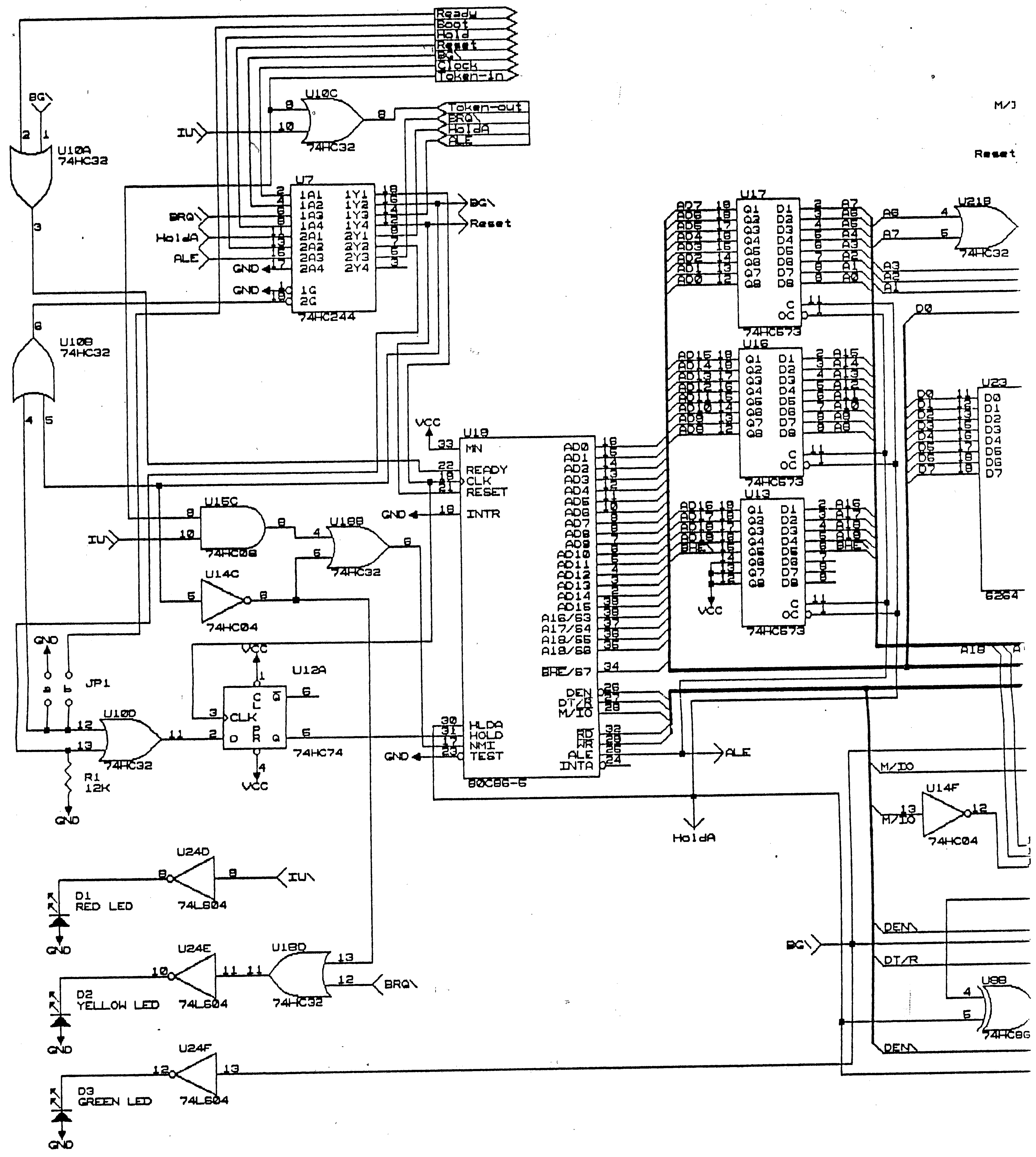


John C. Pagano
Lehigh University

Title	Control Board — Arbitration Logic	
Size B	Document Number C2	REV A
Date: January 1, 1980	Sheet 2 of 2	

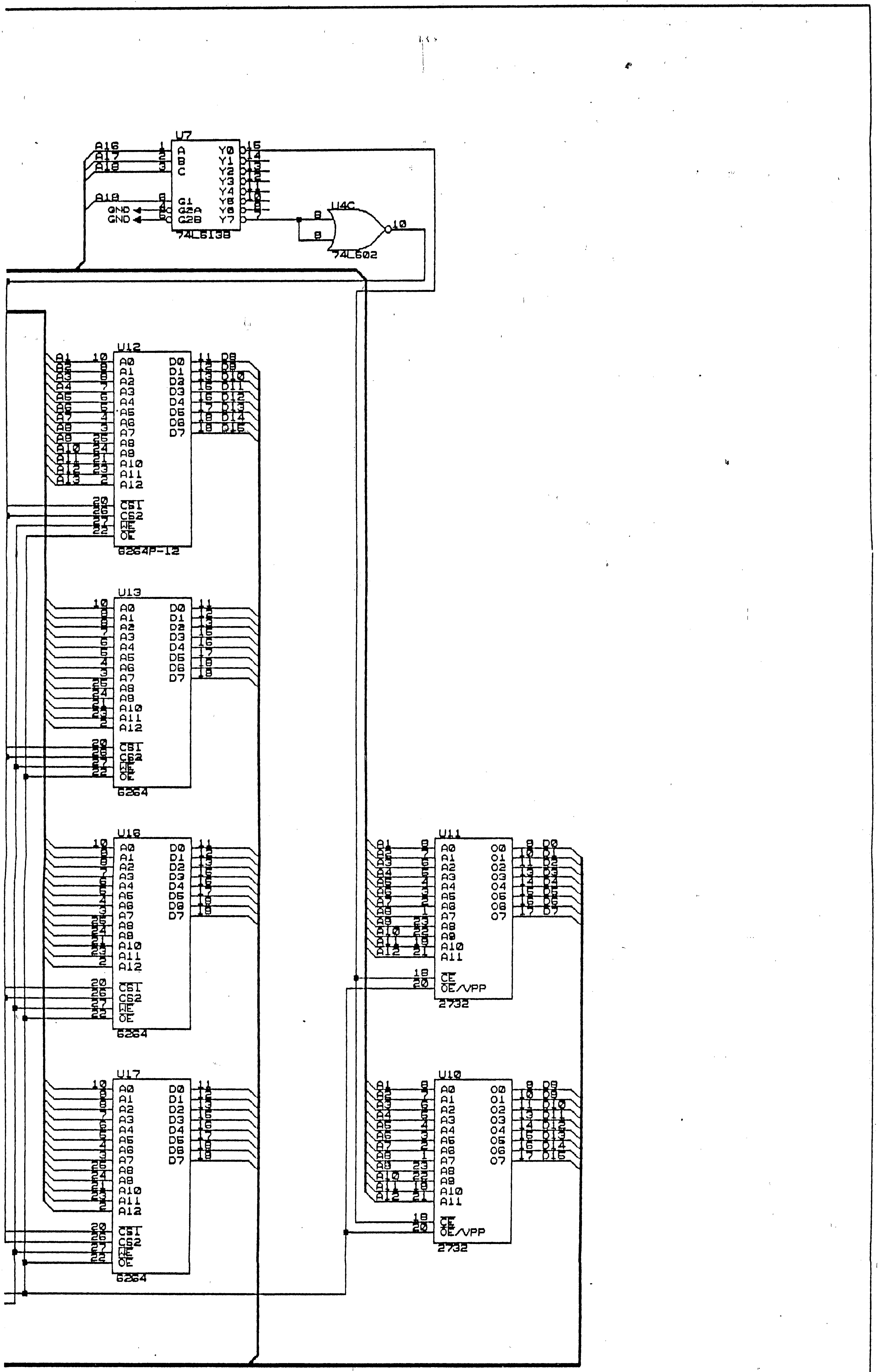


John C. Pagano		
Lehigh University		
Title	Processing Unit	
Size C	Document Number P1	REV B
Date: January 1, 1980	Sheet 1 of 1	

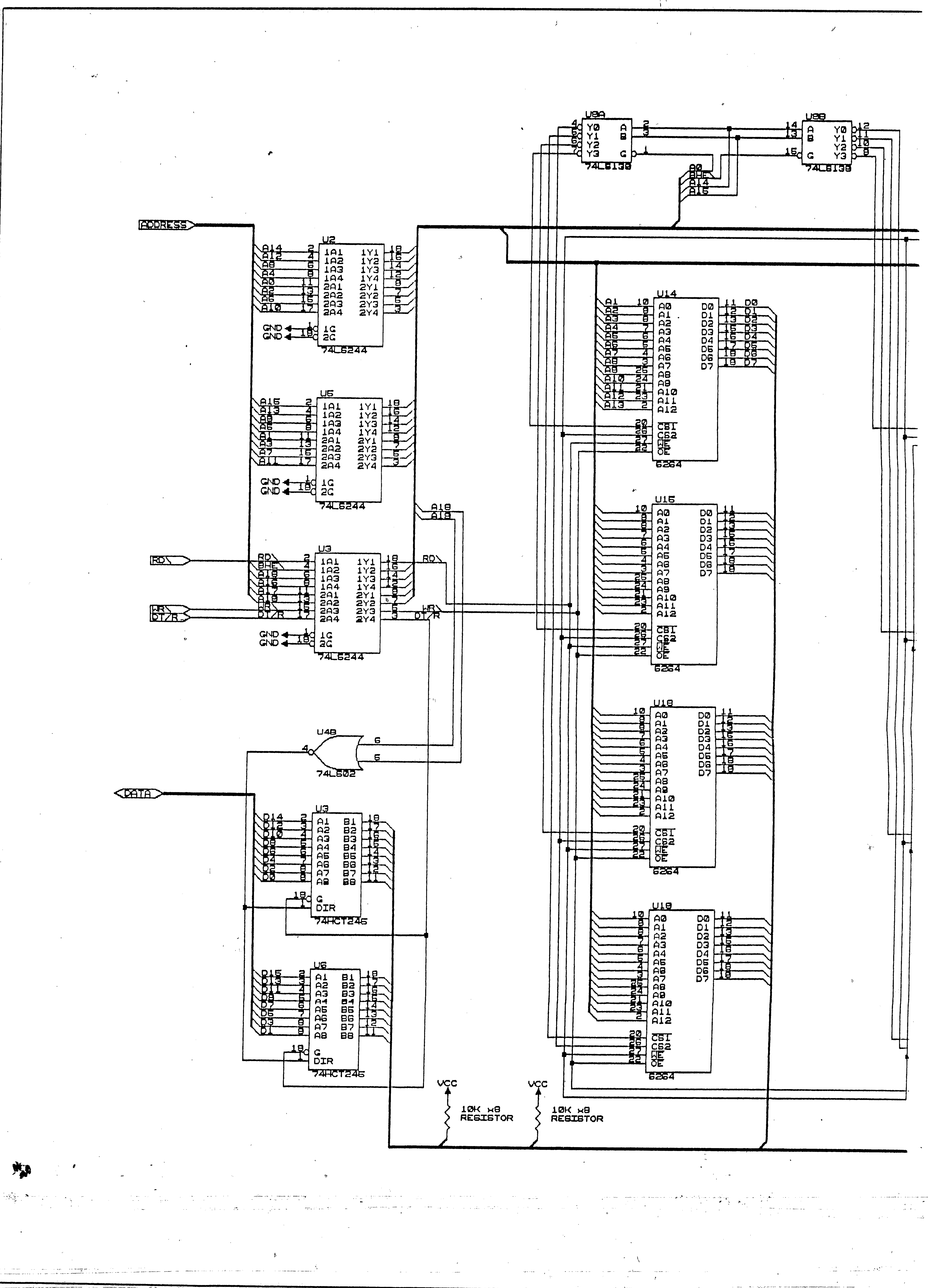


JP1 settings

a – processor #1
b – all others



John C. Pagano Lehigh University		
Title Main Memory Board		
Size C	Document Number M1	REV A
Date: January 1, 1980	Sheet 1 of 1	



ADDRESS
DATA
RD\
WR\
DT/R
U2
U6
U3
74LS244
U4B
74LS02
74HCT245
U9A
U9B
74LS139
U14
U15
U16
U18
6264
VCC
10K x8 RESISTOR
GND
A0
BHE
A14
A15
A18
A19

VITA

John C. Pagano was born to Dominick and Anne Pagano on October 12, 1962 in Paterson New Jersey. He graduated from Lehigh University in 1985 with a B.S. degree in Electrical Engineering. He was coauthor of a paper presented at the 1985 IEEE International Test Conference -- Backdrive Stress Testing of CMOS Integrated Circuits.