51079

0-315-03183-2

National Library of Canada | Bibliothèque nationale du Canada

CANADIAN THESES ON MICROFICHE | THÈSES CANADIENNES SUR MICROFICHE

NAME OF AUTHOR/*NOM DE L'AUTEUR* C. Roger Toren

TITLE OF THESIS/*TITRE DE LA THÈSE* A MULTI-PROCESSOR DESIGN FOR A SERIAL PROBLEM

UNIVERSITY/*UNIVERSITÉ* Simon Fraser University

DEGREE FOR WHICH THESIS WAS PRESENTED/
GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE M. Sc.

YEAR THIS DEGREE CONFERRED/*ANNÉE D'OBTENTION DE CE GRADE* 1981

NAME OF SUPERVISOR/*NOM DU DIRECTEUR DE THÈSE* Thomas W. Calvert

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

DATED/*DATÉ* 19 January 1981 SIGNED/*SIGNÉ*

PERMANENT ADDRESS/*RÉSIDENCE FIXE* [redacted]

NOTICE

The quality of this microfiche 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 a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche 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 mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

A MULTI-PROCESSOR DESIGN FOR A SERIAL PROBLEM

by

C. Roger Toren

B. Sc., Simon Fraser University, 1972

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

by

Special Arrangements

SIMON FRASER UNIVERSITY

September 1980

APPROVAL

Name: C. Roger Toren

Degree: Master of Science

Title of thesis: A Multi-processor Design for a Serial Problem

Examining Committee:

Chairperson: Thomas K. Peucker

Thomas W. Calvert
Senior Supervisor

D. R. Seeley

James J. Weinkam

Peter Lawrence
External Examiner
Associate Professor
Department of Electrical Engineering
University of British Columbia

Date Approved: September 12, 1980

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/~~Project/Extended Essay~~

A MULTI-PROCESSOR DESIGN FOR A SERIAL PROBLEM

Author: _____
(signature)

C. Roger Toren
(name)

19 January 1981
(date)

ABSTRACT

In the past, new computer designs which provided large increases in speed have been based largely on faster circuitry. As the future gains to be made on this basis appear to be smaller and more difficult to achieve, more attention is being paid to multiple processor configurations for increases in capacity. Some early impediments to parallel systems were lack of suitable software and the cost of processors themselves. Most programming tools are designed for problems which are serial in nature, and many people design algorithms in a serial way. The cost of even small processors ranged from thirty thousand to several hundred thousand dollars.

Presently, the cost of some processors has dropped to the level of a few hundred dollars. It is now economically feasible to consider development of multiple processor systems, even in spite of the availability of faster machines. Software for parallel processing is still not well defined, nor are multiple processors in wide spread use. Several basic configurations are described in the literature.

One method of parallel operation on so-called serial problems is to break the problem into n tasks and have a series of n processors each assigned to a task. Such an approach is used on a microscopic scale in pipelined computers where instructions are broken into several smaller steps. This thesis

begins by examining some of the classical multiple processor configurations, and illustrates how "traditional" parallel architecture may be extended to subtasks of larger problems, as opposed to parallelism at the level of single instructions. A dual processor system was implemented in an application that had been served by a single processor.

The application studied was a data acquisition and monitoring problem at the TRIUMF nuclear research facility. The major tasks include acquisition of multiple analogue signals, data logging, data sorting, graphical display generation, and user interfacing. A flow diagram depicting the tasks and the data flow between them provides a basis for determining the number of processors that could be used, and a possible task distribution. In this application, an existing minicomputer was enhanced by the addition of a micro-processor to perform some of the major tasks. The two processors cooperate in a manner similar to that of a pipelined single-processor system. Performance evaluations on the actual system which was built confirm the design calculations and the feasibility of this architecture.

QUOTATION

... For although in a certain sense and for light-minded persons non-existent things can be more easily and irresponsibly represented in words than existing things, for the serious and conscientious historian it is just the reverse. Nothing is harder, yet nothing is more necessary, than to speak of certain things whose existence is neither demonstrable nor probable. The very fact that serious and conscientious men treat them as existing things brings them a step closer to existence and to the possibility of being born.

Herman Hesse,

Magister Ludi

ACKNOWLEDGEMENT

I wish to thank Dr. Tom Calvert, my supervisor, for his guidance and patience throughout the course of this work. I am also grateful to Drs. Jay Weinkam and Doug Seeley for their thoughtful contributions. I am particularly grateful for the financial support provided by Dr. Ralph Korteling of the TRIUMF project, where this system was implemented. I deeply appreciate the help and advice regarding hardware design which I received from Wendell Bishop, without which this design may never have been implemented.

TABLE OF CONTENTS

LIST OF FIGURES

I. Introduction

The rate at which a computer performs instructions has always been one of the common measures of its power. A more meaningful measure, and one of ultimate significance to a computer user, is the time required to solve a given problem. This time is important both practically as well as psychologically. Some problems require a solution within a limited time. For example, it is imperative that a weather prediction problem be solved before the actual event. In applications involving direct communication with the user, a primary goal is to respond to the user's request within a time that the user perceives to be reasonably required to solve his problem.

For a given set of instructions which will solve a given problem, it is obvious that the computer which processes these instructions at a higher rate will solve the problem in less time than a slower computer. The question can be asked: Is processor speed the only factor to be considered in the solution of time-limited problems? Of all the advances made in computer design, the developments in basic component design have had the most impact on the speed of computation. From relays and vacuum tubes, through TTL to ECL integrated circuit technology, the rate of instruction processing has increased roughly by a factor

of one million. Other advances in architecture or algorithms, with few exceptions, have provided only a few orders of magnitude improvement. This is not surprising, as there has been little incentive to explore other possibilities for general applications given the impressive advances of technology.

The rate at which new component technologies are appearing, and their impact on processing speeds, is currently decreasing as researchers deal with such fundamental barriers as the speed of light and problems as mundane as disposing of the vast amounts of heat generated by fast circuitry. Such developments as the Josephson junction[1] may be significant hardware breakthroughs, but their implementation is at least several years off. More attention is now being paid to computer architecture and algorithms to reduce problem solving time. The few orders of magnitude speed improvement that they can provide have become a major contribution to new computer systems.

One active area of research into computer architecture is that of multiprocessors[2] - the coupling of two or more processors (sometimes hundreds) for the purpose of solving, in parallel, the subproblems that make up some larger problem. The parallelism in the hardware may be within instructions, within the data, or between sub-tasks. This has led to the development of "pipelined" processors, vector processors, and multi-processors, respectively. The different forms of parallelism require different treatment in almost all aspects of

systems design.

Often, these new architectures are difficult to use as general purpose machines[3]. Software must be developed with the particular architecture in mind. To examine the implications of this on a typical problem, an existing application was sought for study. One of the experiment groups at the TRIUMF nuclear research facility identified a need for more complete facilities to monitor and control their experiments, and at the same time, to increase the data sampling rate. For a given set of resources, these two objectives are generally mutually exclusive, since they both make demands on the same resource (processor time). This particular application seemed to be composed of several sub-problems, many of which were independent of each other and therefore candidates for parallel processing.

Previously, it was the cost of processors, which hindered the developement of parallel systems. Now, with processors costing of the order of ten to one hundred dollars each, other aspects of the problem are more prominent. One of these new aspects is that of coupling, or communcication, between the processors in a multiple processor configuration. The techniques of programming parallel systems are sufficiently new and complex that they too limit the application of these systems into general computing.

One aspect of converting a serial problem to a parallel one is to locate the sources of parallelism. Following this, a

balance must be found so that the parallel tasks are distributed as evenly as possible between the processors. The objectives of this work were to implement an existing serial problem on a multiple-processor system. The intent is to show that pipelining is not only useful within instructions, as it is classically implemented, but may also be applied to problems consisting of large subtasks and a fixed number of processors. A method of determing the parallel aspects of a problem is sought, so that they may be implemented on individual processors. For the case where there are more parallel subtasks than processors, the problem of balancing the load on each processor is examined.

This work demonstrates, by way of a functioning system, that multi-processor systems can be implemented even for small applications where some parallelism exists in the problem. In the second chapter, the various configurations of processors are reviewed in terms of the relationships both between the processors and between the processors and the data upon which they operate. Those aspects of software and communication which relate to multiple processor systems will be discussed in the third chapter. Chapter four focuses on a problem which was particularly amenable to parallel processing. The environment, input and output, processing requirements, and sources of parallelism are defined here. The fifth chapter describes the decisions faced in the design of a system to satisfy the problem definition give in chapter four. Chapter six deals with the

responses to the design decisions. It describes the actual system which was built, and why certain decisions were made. In conclusion, the system's performance is discussed both from the point of view of designing such a system, and from the operation of the system.

II. Classes of Multiprocessors

This chapter provides a brief overview of parallelism in computer architecture. It is important to be aware of the sources of parallelism of the various architectures in order to choose the most appropriate one for a given problem. This will also give the reader a perspective from which to view the system which was implemented as part of this work. In 1966, Flynn[4] defined four classes of computer architecture, and all computers fall into at least one of these. His classes were delineated by whether or not there was multiplicity in the instruction or data streams. They are termed "single-instruction, single-data", "single-instruction, multiple-data", "multiple-instruction, single-data", and "multiple-instruction, multiple-data".

Before examining these classes, the meaning of the terms "instruction", "datum", and "cycle" as they are used here should be stated. In a 16 bit machine, for example, we consider one word to be a single datum, but in fact, 16 bits are operated on in parallel. I will consider this still to be a single-datum operation because it is an operation on the smallest addressable unit of data, i.e., 16 bits must be operated on, whether they are all needed or not. It is clear, however, that an argument could be made that this is a vector operation, with sixteen processors operating in parallel, especially with respect to

boolean operations.

Single Instruction, Single Data Machines

Most early computers were, and many still are, single-instruction, single data (SISD), according to the Flynn categorizations. An SISD machine executes one instruction per cycle, which operates on one datum per cycle. There are two clear phases in this type of computer: the (instruction) FETCH phase, and the EXECUTE phase. An instruction is fetched, then it is executed after the fetch is complete. When the execute phase completes, the fetch phase begins again. A program is then a series of sequential fetch and execute phases, as depicted in figure 1. Note that while one phase is active, the other is doing no useful work. Also synchronization is required between every phase change. The hardware provides this, by either a regular clock, or some pulse indicating completion of the previous phase. Chen[5] gives three precedence rules for an SISD fetch-execute cycle. Let Fi be the i-th instruction fetch, and Ei be the i-th instruction execution. For all meaningful i,

1. Fi precedes Ei
2. Fi precedes Fi+1
3. Ei precedes Fi+1

In general, the location of intstruction i+1 is known at time i: it is simply the location immediately following instruction i. Thus, since the fetch hardware is idle when the

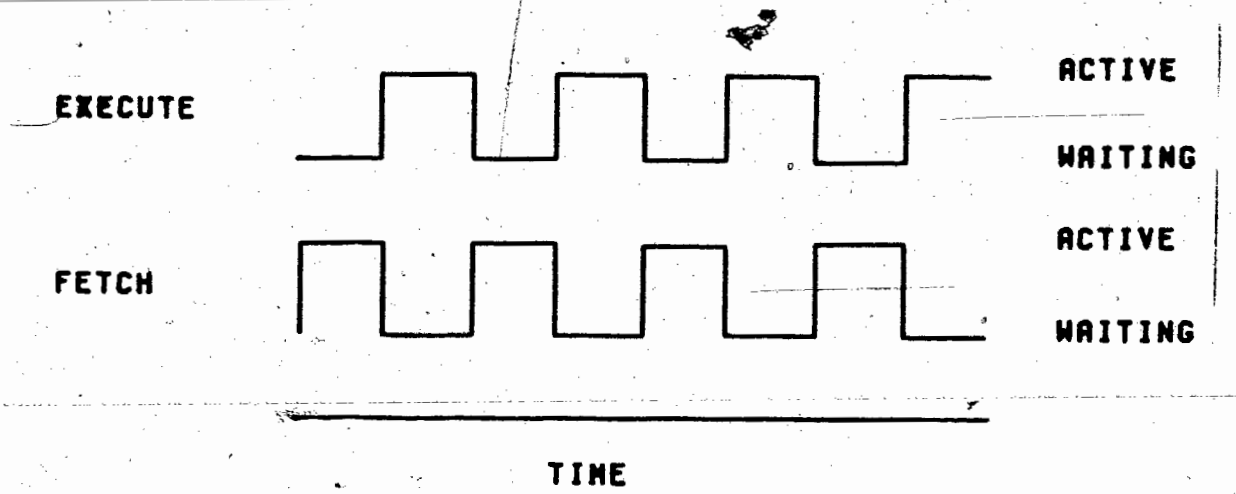

Figure 1. SISD non-overlapped fetch and execute cycle

execute hardware is operating, it should be possible to "look ahead", or pre-fetch the next instruction during the execute phase, thereby overlapping the phase Fi+1 with Ei (figure 2). The precedence rules are modified so that (3) becomes

3) Ei precedes Fi+2

We must now examine the assumptions made to allow the concurrent operation of the fetch and execute phases. First, it was assumed that the location of instruction i+1 is known. This is not true on a branch, whose purpose of course is to alter the sequential fetching of instructions. While the pre-fetch mechanism has fetched the next location in the physical sequence, the execution of a branch must prevent that instruction from reaching the execute phase, direct the fetch hardware to obtain a different instruction, and then it must skip one execute cycle while the fetch is performed (figure 3).

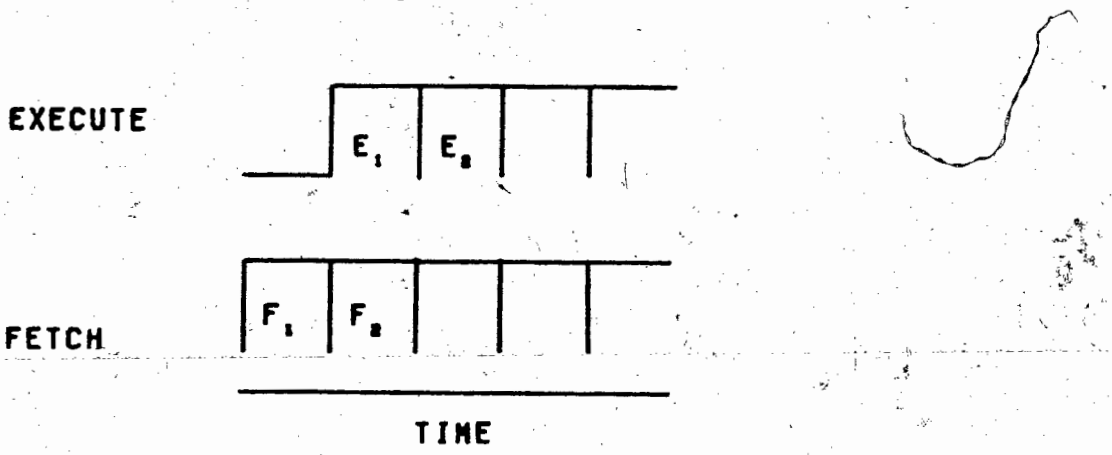

Figure 2. SISD overlapped fetch and execute cycle.

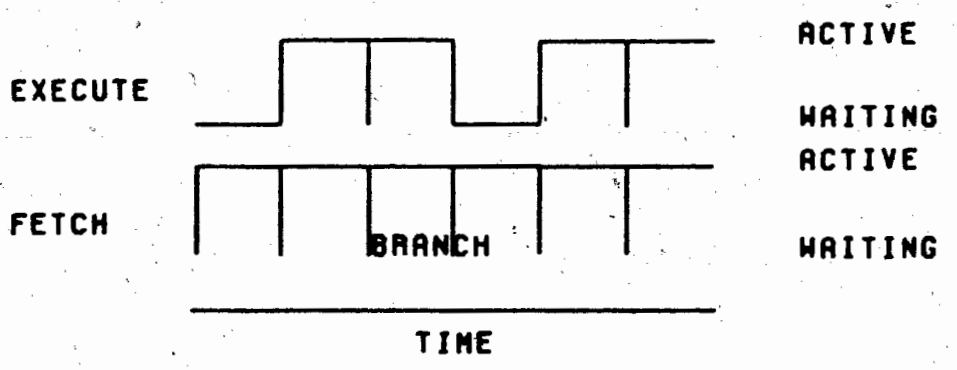

Figure 3. Effect of a branch on overlapped fetch-execute cycle

The other assumption that was implictitly made is that the execute phase does not interfere with the operation of the fetch phase. Interference could occur in two ways. First, the memory may not be able to be accessed by both the fetch and execute units simultaneously. If this was the case, one unit would have

to wait for the other, yielding effectively to non-overlapped operation. At least two solutions to this problem exist. The width of the path from memory may be increased, allowing two or more instructions or data words to be retrieved for the same cost in time. Another solution widely employed is to use interleaved memory units. In two-way interleaving, every second word is controlled by a different memory unit, so the chances of interference are cut roughly in half.

The second way in which interference occurs is if the execute phase Ei modifies the location specified in Fi+1. This effect is not as serious as the first effect because it would be rare in most programs, and can be minimized by programming. Solutions to this problem range from merely documenting the restriction to having "intelligent" buffering units which are able to detect changes to the locations which they are currently reflecting and upon detection, load the new contents.

Below, the conditions where overlap is an effective means of increasing the speed of a processor are examined. If T(Fi) and T(Ei) are the times required to fetch and execute, respectively, the i-th instruction. The total time T required to process a series of n instructions on a non-overlapped machine is

$$T = \sum_{i=1}^{n} (T(Fi) + T(Ei)) \qquad 2.1$$

It is simply the sum of all the times required for each fetch and execute.

In an overlapped machine, we must impose restrictions on the type of instructions in the series of n instructins used above. Any branch instruction will increase the time required by an overlapped machine, so for comparison purposes, we will assume there are none in the stream. Also, we will assume that no instruction execution Ei modifies the instruction at i+1. The total time required for this restricted series of n operations is

$$T = \sum_{i=0}^{n} \max(T(Fi+1) + T(Ei)) \qquad 2.2$$

where $T(Fn+1)=T(E0)=0$.

This time is the sum of the times of the longest time of each fetch and execute pair. Note that the pair is made up of the execution of one instruction, and the fetch of the next. This is an important consideration, since in many machines, T(Ei) is not constant for all i. Also, it takes time T(F1) before true overlap begins, and time T(En) is also not overlapped. If n is large, these times are not significant, but if n is small, as necessitated by a series of instructions containing many branches, then as n -> 1, equation (2.2) degenerates into that for the non-overlapped machine. The time

saved Tv by an overlapped machine is given by

$$Tv = \sum_{i=1}^{n-1} \min(T(Fi+1) + T(Ei)) \qquad 2.3$$

When fetch and execute times are approximately equal, the speed increase is close to a factor of two, but as the ratio of fetch and execute times diverge from one, the savings diminish rapidly. In general, these times are comparible. Also, most programs do not consist of a series of branches.

The concept of overlapped operations is not in any way restricted to the overlap of fetch and execute phases. Both of these phases in themselves have several micro-operations which they perform serially. A fetch may involve a phase to access memory for the instruction, and a phase to decode the instruction before it is passed to the execute phase. The execute phase may be divided into a phase for operand fetch and a phase for the actual operation to be carried out. All these phases may occur in parallel under the same scheme described previously. As more phases are split into smaller micro-operations, and their parts overlapped, the time of one micro-operation becomes very short. While the time for one complete cycle from fetch to the end of execution remains the same, operations are commenced, and thus completed, at the same rate as the micro-operations are completed. Extreme cases of

overlapped operation are known as pipelined operation.

Under appropriate circumstances, pipelining is a very effective means of increasing the speed of processing. It has, of course, the same set of problems as simple fetch/execute overlap, but their effects can be more severe. For example, it takes n cycles to fill a pipeline involving n steps, so the cost of a branch which drains the pipeline is much higher. The advantages of pipelining over other forms of parallel operation lie in the fact that little or no change in a given algorithm is required in order to realize the benefits.

Parallel Processors

In the pipelined approach to computer design advantage was taken of the fact that some units of the hardware were idle during succeeding portions of an instruction cycle. Since the results of the succeeding units in general would not influence the operation of preceeding units, the pipeline could start a new operation as soon as the first unit had passed control to the next unit. If we now examine which blocks of logically contiguous instructions in a program are idle, as opposed to which hardware units are idle, we may determine that some blocks of a program are independent of each other, and thus a number of blocks may be executed in parallel by several processors operating in parallel.

Parallel computers fall into the last three Flynn classes, according to the type of concurrency in the computer's operation. These are: the single-instruction, multi-data (SIMD), the multi-instruction, single-data (MISD), and the multi-instruction, multi-data (MIMD). These machines perform parallel operations on either multiple data streams, multiple instruction streams, or both, respectively.

Single-Instruction, Multiple-Data Machines

An SIMD computer consists of a single control processor similar in many ways to a serial computer, and for example, a vector of arithmetic processors all connected to the central processor, as in the Illiac IV[6]. Each arithmetic processor has some local memory, and a set of registers for indexing and computation. Instructions are divided into either control instructions or vector instructions. Control instructions, such as branches, are executed only by the control processor, for the same purposes as in a serial computer. Vector instructions are fetched by the control processor, then broadcast to all the arithmetic processors for execution. Typically, a base operand address is broadcast along with each instruction. Each arithmetic processor, with its unique index register, can access a different operand by offsetting the broadcast base address by its index register.

We noted in the pipelined SISD computer that a branch can cause considerable disruption of the pipeline. Processes entering the pipeline after the branch has been fetched may have to be discarded. In an SIMD computer, a branch is executed only by the control processor, thus all the arithmetic processors sit idle during this cycle. If there are n arithmetic processors, then effectively n operations are lost due to one branch (or any control operation). This is on the same order as the time required to fill an n-step pipeline. However, the length of a cycle in a parallel computer is likely to be shorter than the length of n micro-cycles in a pipeline machine, therefore the parallel computer will be able to "recover" from the disruption more quickly than a pipelined machine.

It is interesting to note how a conditional branch is handled by a parallel machine. Since each arithmetic processor is operating on different data, the condition codes in each processor are not likely to be the same. Some processors should be issued with one set of instructions following a conditional branch, while the others should be issued with the alternate set. The solution used in the Illiac IV[6] is to selectively set a mask to disable processors while one path of the conditional branch is taken, then the mask is complemented for the duration of the alternate path. The control processor executes both alternatives of the conditional branch.

Little research has actually been done to completely define the types of algorithms suitable for parallel computation. Some areas have been explored, however. Winograd [7], for example, has reported on arithmetic expression evaluation and iterative methods as they relate to parallel processing. He showed that the speed gained in an arithmetic evaluation, such as the sum of a series, is slightly less than linear with the number of processors, while iterative methods were enhanced by a factor that grew only as the log of the number of processors. It would be unrealistic to expect order-of-magnitude gains as a function of the number of processors.

Multiple-instruction, Single-data Machines

The SIMD computer has multiple data streams operated on by a single instruction stream as a means of increasing computational power. One can alternatively conceive of multiple instruction streams operating on a single data stream (an MISD computer). There are no true MISD computers in existence, due in no small way to the difficulty in defining a problem in terms of multiple operations occuring simultaneously on a single piece of data. Clearly, the data stream of an MISD computer could only be a source, not a destination, of data. One can imagine some specific uses, where the data is used to spawn several related processes, such as in a payroll system where the data is the gross pay, and four special processors calculate the tax,

pension, UIC, and social insurance deductions simultaneously. It is not clear that more than a small number of parallel instruction streams would ever be useful or manageable, as opposed to the SIMD machine where large multiplicity in the data stream is realizable and applicable to current problems.

Multiple-instruction, Multiple-data Machines

While the MISD computer was not realistically functional because of its single data stream, if each processor is provided with a data stream, giving rise to the multi-instruction, multi-data computer (MIMD), practical applications again can be considered.

MIMD computers are generally considered to be tightly coupled networks of two or more processors. There is no requirement that the processors be similar, although the interchangability of software is one advantage if they are. Unlike SIMD computers, the instruction set is not confined to vector instructions, nor does it necessarily include them. There are m separate instruction streams operating independently on m different data streams (when m different processors are coupled). The task in implementing programs for MIMD machines is to define the blocks in a program which are independent and thus are able to be executed in parallel. Under program control, the statements FORK and JOIN, proposed by Conway[8] allow for the spawning and eventual synchronization of parallel tasks.

Automatic separation of blocks may be possible using techniques similar to those used in code reordering in optimizing compilers. MIMD software considerations will be discussed more fully in the chapter on "Software and Communications".

In addition to blocks of code executed in parallel, many instructions, particularly I/O instructions, require so much time to execute that it is desirable to use independent processors to complete their execution once it is initiated by a central processor. The use of channel processors in the IBM 360 series is an example, as is the proliferation of micro processors in device control units.

III. Software and Communication

This chapter deals with those aspects of communication and software which are peculiar to multiple processor systems. Communication covers the area of sharing data streams or instructions streams between processors. If sharing is non-existent, it is unrealistic to classify the configuration as a multiprocessor system. The normal input and output operations to devices such as printers and mass storage units associated with most programs are not considered here to be part of the same area of communication. While those functions do generally take place in parallel with other processing, their purpose is not in support of parallelism between processors. They are therefore of interest only as examples of parallelism in current computers.

Software and inter-processor communication overhead reduces the effectiveness of multi-processor systems. Winograd[7] underscores the difference between the theoretical multi-processor system and current reality:

> "...maximizing the number of numerical operations done concurrently assuming that the data is available when needed. We make this assumption knowing full well that in many actual situations it is not valid, and that data movement and sequencing restrictions may slow execution down."

Methods of data movement and task sequencing will be examined,

as well as their motivations.

The communications medium is used by software as a means to share resources, to distribute tasks amoung processors, and to synchronize cooperating tasks. The two areas of software and communication are treated together because of the special nature and purpose of interprocessor communications. Whereas most features of machine architecture are invoked automatically (e.g., pipelining), or are forced upon the programmer (as in the case of array processors), communication is invoked by a conscious effort from the programmer. Additionally, it is more global in its effect. Communication is not one instruction, but virtually, a subsystem within the total system. The software defines both the means and the scope of the communication.

Cooperating Sequential Processes

Terms such as "parallel processing" or "multiprocessing" imply that several distinct processors (but not necessarily individually complete computer systems) are operating together to produce a result which is dependent upon the results of all the individual processors. These individual processors perform sequential operations on the data, possibly different from those operations on the other processors. At some point, however, the results from each processor are collated or merged to produce a "final" result. Due to the sequential nature of each procesor, and the ultimate merging of the results of these processors to

form a composite result, these types of processes are often referred to as cooperating sequential processes.

Ollongren[9] discusses cooperating sequential processes from the point of view of language specification. Referring to a theoretical machine developed in his article, he cited three conditions to which the processors were subject:

1. The set of states of the individual processors are mutually disjoint
2. The sets of pointer stacks of the processors are mutually disjoint
3. Not all sets of memory elements of the processors are mutually disjoint

The first two conditions dictate that the processors be in fact distinct. The third condition provides for the communication between the processors, without which cooperation would be impossible.

He makes the remark: "As a result of condition (3) we are faced with the problem of cooperation between the sequential processors". It is interesting that he views cooperation as a problem, and that it is precipitated by shared memory. It is of equal interest to investigate the methods by which (3) can be implemented, not just through shared memory, but through other means of communication as well. The problem is not so much that cooperation is required, but, more positively, how to achieve cooperation having determined that cooperation is desirable.

Cooperation has two components: the synchronization of the cooperating tasks, and the sharing of information between the tasks.

The ease of synchronizing tasks is related in part to the degree to which the processors are autonomous. If, for example, two cooperating processors are each executing their own independent instruction streams, then they are highly autonomous. Synchronization is accomplished by each processor signalling the completion of its task to the other, each waiting on the other processor before proceeding past a pre-determined synchronization point. This signalling sequence is generally termed a semaphore. Dijkstra[10] developed some primitive operators for such purposes: the P- and V- operators, which signal completion and wait for completion, respectively. This signalling occurs generally at the completion of whole tasks, and is therefore relatively infrequent compared to the rate of execution of instructions.

Other systems, such as array processors, contain sets of usually identical processors executing an instruction stream broadcast by a central unit. The only autonomy they have is that they can choose to execute or not execute the broadcast instruction based upon some condition code generated by the previous instructions. They cannot otherwise determine their own instruction stream. The need for synchronization arises from the fact that different elements of the array being processed may

require different processing. Some may satisfy a conditional statement, others not. The central unit must then broadcast both alternatives of a conditional statement, and the individual processors must execute only those statements of the appropriate alternative. The process becomes complex if the alternatives contain still more conditional statements, where provision must be made to stack (save) the machine states so that a return to the previous level of conditions is possible (i.e., so that those processors which were active before the conditional statement are reactivated). Wettstein[11] expands upon Dijkstra's work by examining synchronization primitives in various environments.

The communication of shared information between tasks is what allows tasks to be called "cooperating". Considerable attention is paid to both the logical (protocol) and physical links between the processors. These two items control the power and effectiveness of interprocessor communication. Protocols are sought which will require a minimum of time to communicate between tasks, and also require a minimum of complexity in the design of the physical link. These links are referred to as coupling between processors, often qualified as loose coupling or tight coupling, depending upon the volume and frequency of communication allowed.

Machine Coupling

The nature and multiplicity of the coupling between processors determines the features of multiprocessing that may be supported, the reliability of the system, and the cost of the system. The number of processors involved also influences what the coupling may be. For example, if there are only two processors, all modes of interconnection are degenerate; only redundancy may be added. Some other interconnection schemes grow complex, and costly, as the square of the number of processors, which limits the practical size of the configuration to a small number of processors, on the order of perhaps ten to one hundred. Still others grow linearly in complexity and cost, at the expense of speed. A few of the classic configurations are examined below. The list of configurations is not exhaustive, but serves to illustrate the range of possible schemes.

Types of Interconnections

Three of the most common modes of interconnection are differentiated by the number of other processors to which each processor is directly connected. The simplest is a form of a master/slave configuration, with one processor connected to n slaves (Figure 4). All communication is through the master processor. Each pair of nodes is separated by at most two links. If one link is broken, only that slave is disabled. The remainder could continue to operate. The cost of adding slaves

is fixed: every slave requires only one link to be connected to the master. A disadvantage arises in large systems, where the master may become a bottle-neck to communication. Also, if the master fails, the entire system collapses.

Another basic configuration has the processors arranged in a ring structure. There is no master; each processors is connected to its two neighbours. Communication to other processors is passed along the ring by each processor until it reaches its destination. Unlike the first configuration, there can be two paths available from each processor to any other processor, thereby increasing the reliabilty of the system. Two paths require, however, that the links be bi-directional. (It could be the case that the links are unidirectional, clockwise for example, yielding only one path to each processor, and one from each processor). In configurations with up to five processors, the speed of the system is at least as good as in the master/slave configuration, while the reliability is greater. There is a maximum of two links between any two processors in a five processor system, but for large rings, the maximum number of intervening links is more than two, thus communication is slower than the master/slave configuration.

A third scheme has each processor directly connected to all other processors. Only two processors ever need be involved in any communication, but the number of connections to support this grows more than linearly with the number of processors.

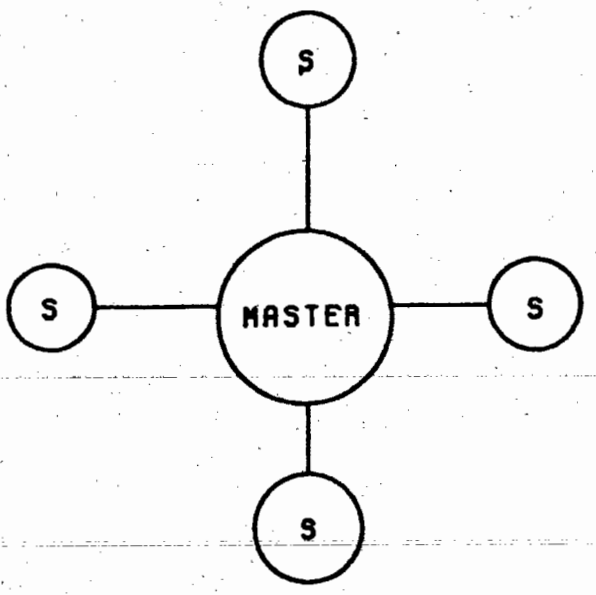

Figure 4. Star (master/slave) processor interconnection

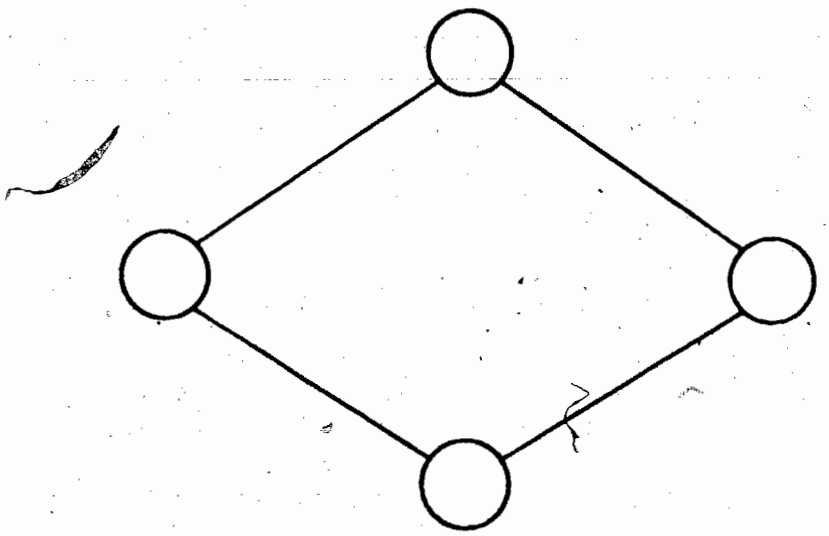

Figure 5. Ring processor interconnection scheme

Reliability of such a system is excellent, because n-1 links must be broken before any processor is inaccessable.

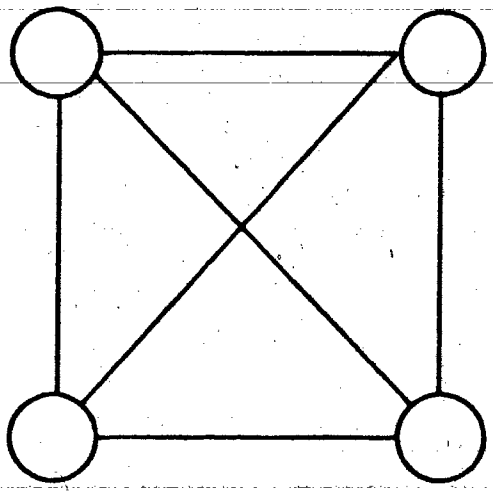

Figure 6. Multiple-master processor interconnection

There is a scheme which combines some of the advantage of complete interconnection, with the low cost of the master/slave scheme. It is a common-bus, or party-line approach. All processors communicate on the same bus, but must of course wait until the bus is free to do so. The delay in waiting for the bus is a disadvantage in large systems, as is the lower reliablity. If the bus fails, the entire system is disabled. Many computer system busses, such as the PDP-11 UNIBUS (Digital Equipment Corporation), use this system for their processor-peripheral interconnections, because of its design simplicity.

The purpose of interconnecting processors is generally to take advantage of either

1. Parallelism in a problem
2. Specialized facilities on one or more processors

If there is parallelism of a local nature, such as a vector

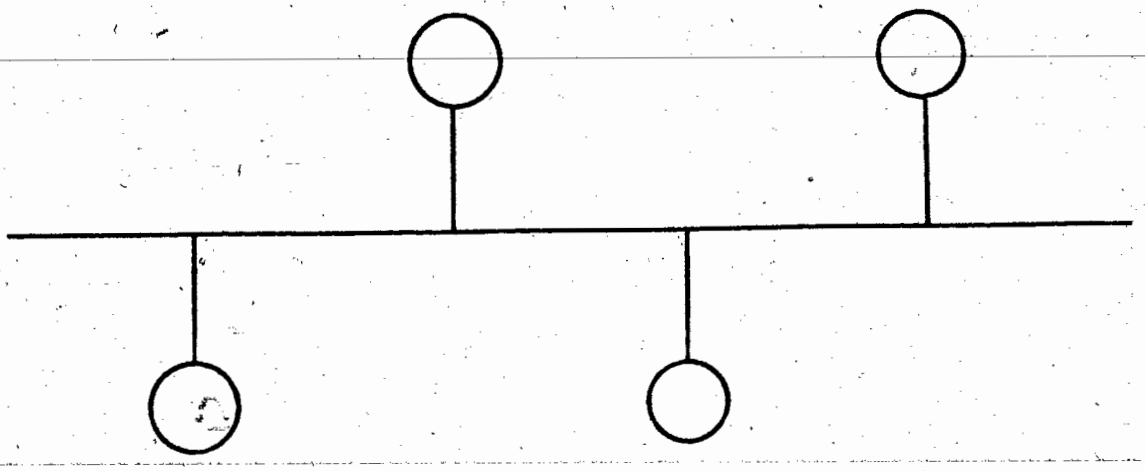

Figure 7. Bus method of processor interconnection

operation, communication paths with speeds on the order of instruction times are desirable. When making use of specialized resources, possibly several of them in parallel, slower paths may suffice, as these resources typically operate on a larger instruction or data stream. Normal techniques of buffering and block I/O compensate for some portion of a slower communication path. I have somewhat loosly termed these two categories "instruction communication" and "data communication".

Instruction Communication

Some systems are so tightly coupled that the processors interact at the instruction level. By this it is meant that individual instructions to be executed are communicated on demand, as well as data to be operated upon. The Illiac IV is an example of this type of coupling. Instructions are broadcast to

sixty-four processors by one central processor. The processor links must be at least as fast as the slowest processor, otherwise processing power is degraded. Paths handling instruction communication are usually highly parallel, and logically transparent to the programmer. The use of networks, such as rings, in instruction communication is generally not effective because of transmission delays which are much longer than an instruction cycle.

Another use of instruction communication occurs in multi-processor systems which attempt to dynamically distribute tasks between processors. A waiting task is dispatched to any available processor, which may or may not be the processor which originated the task. The ability to do this over a large number of processors requires rather sophisticated hardware and software for communications. This use of instruction communication is very similar to that of data communication, and in fact could easily be justified as such due to its block-structured nature.

Data Communication

Data can usually be aggregated into a block considerably longer than one instruction. Similarly, entire procedures may be considered as a block of "data" for the purpose of dynamically distributing tasks among processors. Delays encountered in passing data through many nodes (as in a ring structure) are

tolerable as long as they are kept to a small fraction of the time required to transmit the message. Most networks are designed to handle data communication as opposed to instruction communication.

Data that is an operand of an instruction is best treated as an instruction. It is normally of short length, and required immediately by an instruction. Delays are less acceptable here than for blocks of data, for the same reasons as for instruction communication.

Shared Memory

Shared memory is a physical feature of some multiprocessor systems. In some instances, only a small block of memory (on the order of a few thousand words) will have not one but two or more access paths. Each processor sharing this memory will be connected to one of these paths. The shared memory is generally used as a communications area. Where large amounts of memory are shared, instructions or operands may be fetched at random by any processor - the memory is used as if it were part of each processors main memory. In nearly all cases, each process also has a certain amount of private storage.

The sharing of a portion of memory between processors permits their tightest form of coupling. Since memory is the source or destination of most operands, transactions take place without the overhead of setting up communications devices.

Often, this requires only the setting of a pointer to the data rather than actual movement of data. Instruction streams which are re-entrant may be executed by more than one processor without copying the stream. These features encourage the frequent exchange of information in volumes which might cause excessive overhead or delays in other, less direct, media. Array processors make extensive use of shared memory for concurrent processing of multiple data elements.

While shared memory does have merit, it is not without some serious limitations. Multiple access by more than a few processors is not easily attained. a memory request will lock a block of memory, not just the word requested. While there exist mechanisms to reduce these "collisions"[12], most notably interleaved memory and to a lesser extent cache memory, it is unlikely that two processors whose memories are totally common will operate at full speed. For this reason, usually only a small portion of memory is actually shared between processors, while tasks are executed out of private storage.

Shared memory is most suitable in configurations where only three processors would be contending for access: a node and its two neighbours. In configurations where a processor is linked with more than two or three others, shared memory may be made viable by assigning n blocks of memory in each processor to be shared with each of the n other processors. In this way, there can only be two contenders within each block. Another

consideration concerns the proximity of the processors. All processors sharing the same memory must be physically close to each other to conform to the timing constraints inherent in any two-way high speed bus.

Direct Memory Access

Direct memory access (DMA) between processors provides an efficient link between the private memories of each processor. Using input/output commands, blocks of memory, rather than individual words, are transferred between processors. The transferred contents, perhaps containing both data and instructions, will be operated upon at the destination. It is thus not a direct sharing of memory, since there is no provision to operate within another processors memory.

The key feature of DMA over other forms of interprocessor I/O is that either processor may control the transmission, rather than requiring one processor to be the master at all times. DMA is in effect an extension of the main processor bus. Unlike a memory port, many processors can be accommodated concurrently. The address space available to the processors is generally larger than for shared memory, often comprising all of memory. The difference between DMA and shared memory is that when using DMA one processor must specify which portion of memory it wishes to share, and then obtain a copy before any processing may be done.

Since the DMA bus acts like a processor bus (and in practice often is the processor bus also), it is also limited to processors in close proximity to each other. To overcome this limitation, devices are attached to the bus which will buffer the DMA transfers while driving a data line over a longer distance. As the distance increases, the transmission rate must be lowered to maintain signal integrity. These unidirectional drivers also allow only one processor to control the memory addresses. The more remote processor does not share the memory on the same basis as the processor with the true DMA interface.

IV. Problem Definition

This chapter focuses on a specific application selected for study. It was chosen because it was apparently amenable to a multiple-processor configuration. In order to convert this serial problem to a parallel one, the subtasks must first be identified. This is the primary intent of this chapter. The application is similar to a process-control problem, and therefore a brief general comment is made relating to the role of computers in such an environment. Thereafter, the problem is introduced from the point of view of the user. A framework is being built in order to discuss the design decisions that were faced, and which are discussed in the chapter following. To this end, an overview of the research for which the system was developed is provided. The actual process being monitored is described, the sources of input are introduced, and the processing tasks are defined.

The Process Control Environment

There are many reasons why computers are being employed for the purposes of process control. A suitable computer system can handle more events, more reliably, than a manual system. In some processes, events occur too frequently for manual control to be effective, unless the process is slowed when manual

intervention is required. If the process is not slowed, events which may be critical to the process might be ignored with catastrophic results. Other processes may be relatively slow, but require a high degree of accuracy or repeatability. Digital systems can fulfil these requirements to almost any extent, given the appropriate resources. Perhaps the process is simply too boring to humans; this too is a very valid reason for automating some processes.

The characteristics which make a computer application "process control" should be examined. It is similar to any program in the respect that it has input, computational, and output requirements. However, the substance of these requirements is where the distinctions lie. The input is invariably from machines rather than human operators, and often from a large number of sources, perhaps over a hundred. The rate of input is beyond the control of the program. The computational requirements usually consist of validating the input from each source, such as checking that it is within given tolerances. Much of the computation is therefore taken up by conditional statements. The output can be of two types. One is the normal reports to a user, consisting of messages and results. The other is output which is fed back into the system to control some aspect of the process. The effect of the output will be seen in subsequent input, which in turn may cause new controlling output to be generated.

Problem Origin

The TRIUMF meson facility, located at the University of British Columbia, is a cyclotron used for research into many areas of nuclear physics and medicine. One of the experiments consists of monitoring the products of nuclear collisions, which result from a beam of protons colliding with one of several targets. The products are detected by a number of silicon detectors mounted on movable arms inside a container called a "scattering chamber". The angle of incidence of the target relative to the beam is controllable, as are the positions of the detectors relative to the target. In addition, the target may be changed to one of several installed in a sample changer. Each product that is detected is called an "event". One purpose of the experiment is to create a profile of the products in as many parameters as possible, such as angular distribution, energy, and mass. Other experiments, while differing in intent, make use of the same facilities and thus present similar sources of input. The next few sections briefly describe these sources of input from the experimental equipment, and the monitoring and control facilities necessary for the optimum use of the equipment by the experimenters. The flow of information and the processing requirements are shown in figure 8.

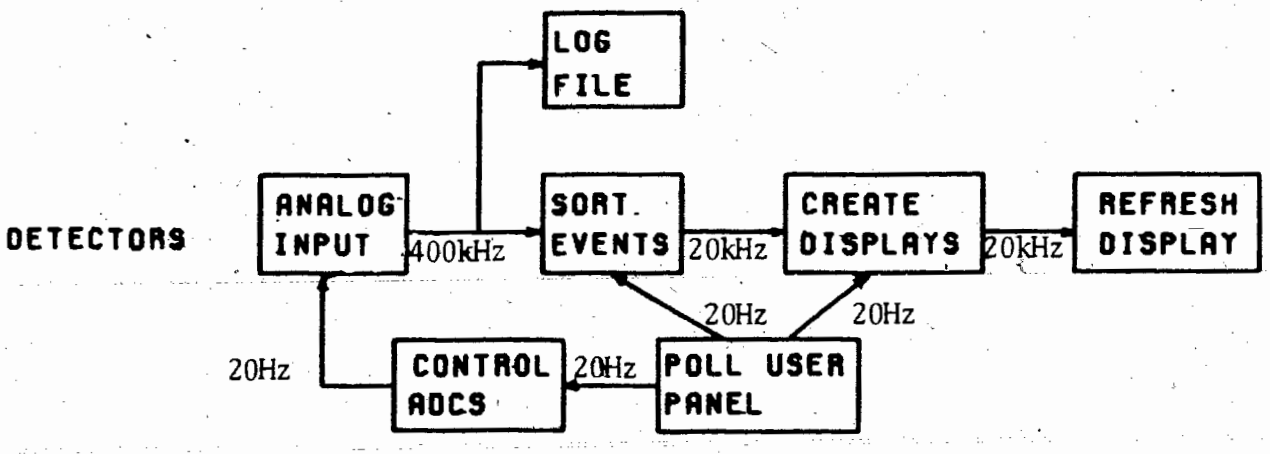

Figure 8. Data Flow and Processing Requirements

Input Sources

Information from the experiment can be loosely grouped into three areas: primary analog input from the event itself, digital and analog environmental information, and digital control information from the user.

The primary analog input sources are the parameters of the experiment which define the characteristics of the events measured by the experiment. Each source defines some independent parameter of the experiment, and each may by itself completely define an event. More often, however, one event is described by a coincidence of two or more parameters: some specific inputs must be simultaneously active, otherwise the event is considered

not to have occurred. Each parameter is digitized into a 12 bit word, which can be presented to a computer every 20 micro seconds. There are up to sixteen possible primary inputs, resulting in a potential data rate of one word every 1.25 micro-seconds, or 800 kHz. The actual events occur randomly in time, although the average rate over time is constant. The occurrence of an event is not under the control of the computer. This is important when considering the effective data rate and the method to be used in dispatching tasks to handle this input. In practice, average data rates of from 100Hz to 5000Hz are expected.

The environmental information originates from any number of measurements that the experimenter deems pertinent to the overall experiment. This may include, for example, the time of day, beam current, various counters, etc. Its use is generally for the normalization of results of the primary input. These are parameters not usually determined by the experimenter, but whose behaviour is well known, and not subject to frequent fluctuations once stablized. As such, it does not need to be sampled as frequently as the primary input. Sampling rates of once every second, or only upon command, are typical.

The control feedback input consists of parameters which can be set by the experimenter, either manually, or under computer control. This would include positions of detectors, sample positions, and status of various other systems. This information

changes only upon command, thus need never be monitored except when a change is requested. The data input rate is negligible. Another form of control input is that from the interface with the user himself. This is chiefly through a panel of sense switches and a light pen. These inputs direct which of the various functions are to be performed . Again, the data rate is negligible, but may occur at any time. The important aspect of this input is that is is not mechanical in origin, but originates from a person. The response time is thus more critical than for other inputs.

Monitor and Control Requirements

The ultimate purpose of monitoring is the verification of the primary input data. This is data whose behaviour is least predictable. Since it has been conditioned by a number of electronic circuits prior to being digitized for input, there is a significant chance for distortion of one form or another to enter the system. The exact effect of fine adjustments of the process cannot be determined beforehand. Many of the coincidence requirements are of a complex nature, or require the coincidence of two or more relatively rare events, which produces an even lower occurrence of these events. At the same time, other events may be occurring which are valid and should be recorded, even while awaiting the verification of the proper specification of rare coincidence requirements. Changes in conditions which occur

beyond the control of the experimenter may adversely affect the nature of the data being collected. It is therefore imperative that the data be available for examination as it is collected. The considerations for this mode of operation are similar to those for the use of online services versus batch for program debugging, in the sense that batch is essentially an unmonitored operation.

A graphical display provides one of the most useful ways to represent data describing complex, or even simple, phenomena. Preliminary visual analysis very often determines the methods which one will use in the detailed analysis. For processes which involve extremely large volumes of data, it is often the only way to represent the data in a comprehensible form. The eye can perform many relatively complicated analyses much more quickly than most computer analysis can be done, and perhaps more importantly, decide which analysis is more relevant to the data presented. For these reasons, a well-supported graphics system is considered an essential aspect of the problem.

Large graphic displays present a significant load to any processor. To prepare a histogram for display requires a certain amount of processing. This includes scaling the histogram to fit on both axes of the screen, selecting subsections of histograms (to increase the resolution over a portion of the histogram), and logarithmic transformations. A cathode ray display must be refreshed at least 20 times per second to eliminate visual

flicker. The minimum acceptable number of points displayed is 256, so there are 195 micro-seconds available to process each point. The rate at which data can be supplied depends on the amount of preprocessing required, and to a lesser extent, the speed of the memory if DMA is used. The preprocessing may involve such operations as windowing, rotating, non-linear scaling, or fetching data from a host for processing. For some of the more time consuming operations, such as log scaling, it is expected that more than 195uS per point will be required. It was evident that the graphics problem alone could require the complete resources of one processor. A scheme to reduce the effective time to this value must be devised.

User Interface

The user interface consists primarily of the graphical display and the associated controls necessary to select the type of display desired. This would include specifying which histogram to display, scaling, and windowing. Also necessary is some method of controlling the process of data collection, such as starting, stopping, and timing. A panel of of buttons, dials, and lamps has been found suitable in previous situations. These inputs must be polled at a rate sufficient to appear continuous to the user. It is expected that polling at the same rate as the display is refreshed will be appropriate.

Data Rates

The intent of the experiments is to determine the relative frequency of occurrence of the various events taking place. There is, therefore, no prescribed data rate, as any portion of the available data should yield results similar to any other portion. One may indicate a rate at which data is available to be sampled, however. This would correspond, for example, to the population of a town where a survey is being conducted. The data rate would correspond to the number of persons stopped to respond to the survey.

Depending upon the experiment, events may occur extremely rarely, for example at less than 1 Hz, or may appear to be continuous, which effectively is anything over 400 kHz. As the events occur randomly, sampling the input randomly, as opposed to capturing every event, is permissible. The accuracy of the results is, however, improved by a larger sample. This may be obtained by either sampling faster or sampling over a longer period of time. It is more useful to be capable of a higher sampling rate, due to the cost of running the cyclotron and of peoples' time. Any improvements in the throughput of the system is therefore useful.

Parallelism in the Problem

A flow diagram is a useful aid in locating parallel tasks. Such a diagram for this application is shown in figure 8. One may be tempted to conclude that the problem is serial in nature, the only major fork apparently being in the logging of data to tape. The parallelism sought is among the tasks which can operate concurrently on different sets of data. Parallelism cannot occur in the data stream because the data is only available as a serial stream from a source independent of either processor. This aspect of the problem dictates the organization of the processors in the categories outlined by Flynn. MIMD systems allow several distinct tasks to each process distinct data items simultaneously. SIMD systems allow only the same operation to be performed concurrently on similar data items.

Five tasks have been identified that can be performed in parallel. These are data acquisition, sorting, recording, display, and the user interface. The criteria used to determine these divisions were based on the precedence requirements of the input to each task. (The division of the tasks across the resources is discussed in the next chapter.) Data that has been collected must be passed to several tasks for processing. Many of these tasks do not pass results to other tasks, or otherwise modify the original data. It is therefore possible to carry on some of these tasks simultaneously, given appropriate system resources.

For example, it has long been realized that slow I/O devices could be effectively speeded up by the technique of buffering. This simply involves initiating input operations for the next record as soon as the device is ready, rather than awaiting a program request. On output, control is returned to the program as soon as the record to be written has been moved to another area of memory (the buffer), while the actual operation is completed at a later time. In the system that was developed, these simple concepts have been used not only in the I/O portions of this system, but have also been extended to other tasks which could execute on other processors while operating on the same data.

In principle, any of the five tasks noted previously could be run in parallel with any other, but some combinations are more optimal than others. It is desirable to have as little communication between tasks residing on different processors to make more time available for useful work. It is also desirable to have a balanced work load between the processors, to reduce the likelihood that one processor must wait for another. Tasks which require frequent synchronization also contribute to high waiting times, and are not good candidates for parallel operation.

In order to determine just which tasks should be grouped together on one processor, a data flow diagram, such as that shown in figure 8, is employed. From such a diagram, the data

rates and task dependencies are clearly visible. One could also label each path with synchronization requirements, and each task with its running cost, to aid in balancing the load.

V. Design Decisions

Chapter four outlined the objectives of this application. In this chapter, various options to be considered are presented for each objective. These are primarily concerned with specifying task distribution, means of communication, and the supporting hardware. The next chapter describes the actual system that was implemented.

Task Distribution Between Processors

Stone [13] in his article on the control of distributed processes, deals with the problem of the distribution of modules (tasks) between processors. Also of interest is the delineation of instruction groups which will comprise the modules, or determination of the best definition of each module. One of the reasons why it is advantageous to selectively distribute the processing load is that some processors perform certain functions faster than others. The pipelined processor is an extreme example of such a situation, where each processor in the pipeline is highly optimized for a limited number of operations.

Another primary consideration noted by Stone is the cost of communication between the various tasks. Tasks which operate on remote processors incur a higher cost per item of information communicated for processing. By carefully choosing the task

distribution, one might reduce the cost of interprocessor communication.

Stone discusses how to optimize the execution of a given set of distributed modules by minimizing the sum of communication and running costs. This may not produce the optimum solution of the problem as a whole if the instruction mix is such that no processor has a clear advantage in processing a given module. If, on the other hand, each module is optimized in such a way that most instructions in it are best suited to one processor, then a further improvement can be made in Stone's optimum configuration. In the general case, however, little distinction may be made between the two processors' capabilities. Communication and processing costs then become the dominant factors in determining optimum task distribution, as Stone has noted. It is this general case with which we shall be concerned. The task definitions very nearly fall out of the problem definition of chapter four. The decision for the most part concerns the task distribution.

The means of communication between multiple tasks, and its costs, should now be examined, comparing the conventional flow of data between two subroutines in a single-processor system with that in which cooperating sequential are executing on a dual processor system.

As stated earlier, the primary goal of this application is to transfer a continuous, randomly presented stream of data from

multiple input sources, in a real-time environment, to a file where it can be analyzed without time constraints. Peripheral to this we would like the ability to pass the data through a series of arbitrary processes. In a conventional system, the data would be passed sequentially from one process to the next, until the last process was finished. A new group of data could then be acquired and the procedure repeated. Only one process at a time would be active. In a dual processor system, one can identify sets of tasks whose input or arguments are not predicated on the results of any other tasks in the same set. The tasks within such a set may be run concurrently on independent processors.

Processors

There are many attributes which are considered when selecting a processor for a uniprocessor system, such as instruction set, speed, and cost. These are still considerations when designing a multiprocessor system, but perhaps with different emphasis. The other considerations which are specific to multiprocessor systems include the purpose and configuration of the multiprocessors, and in light of that, the need for instruction set compatibility, multiprocessing intructions, and communication options.

The multiprocessor configuration under consideration will fall into one of the Flynn categories described in chapter I. Only two of these categories, single-instruction multiple-data,

and multiple-instruction multiple-data are truly practical categories for a multi-processor system. The case of a single-instruction single-data system being composed of two or more redundant processors, while interesting, is not considered here.

In a single-instruction configuration, it is likely that all processors are to interpret the instructions in a similar manner. All processors should therefore interpret instructions indentically. The concept of differing interpretations for a single instruction would almost certainly lead one to classify the configuration as multiple-instruction.

A multiple-instruction configuration allows different instruction sets by use of multiple instruction streams. Once the appropriate streams have been generated, they need not be compatible with other processors. This is particularly true where special purpose processors may be selected to perform specific tasks. There is little desire to have mobility of these modules between processors. It is more general, however, to have identical processors. System reliability is increased through redundancy, as well as reducing the usual encumbrances such as compilers and other system utilities. These items would otherwise be duplicated in the other instruction sets or be in cross-compilers capable of generating all object code in any instruction set.

With respect to instructions specific to multiprocessing, there is one primary instruction which is useful. The use of it is generally for the protection the so-called "critical regions" of a program, where execution by more than one task at once would corrupt the state of some data or status fields. The operation is characterized by the setting of a status bit according to the state of a word, and in the same cycle, writing a pre-defined value into the word. This is commonly referred to as a test-and-set instruction. No interference can occur between modules because both the test and the set occur within one indivisible operation. Wettstein[11] points out that while algorithms exist to assure the same effect without this instruction, no solution is known which is independent of the number of processes involved.

Memory

Memory must be considered as a shared element, rather than a private, instantly accessible resource. It often serves as the communication medium between two processors. The memory requires a finite time to complete a read or a write cycle. When two or more processors, including I/O processors, access memory, there is contention for memory cycles just as there is contention for other shareable elements. Since memory is the most used element in any system (it is the source of all instructions and most

data), the contention problem is one which merits close consideration.

The contention problem arises out of the large block structures of most memories. When one word is accessed, the entire block is considered busy. Thus, if there are two or more processors concurrently requiring memory, one must wait for the other. The problem has been reduced somewhat by creating more blocks of memory, which are either interleaved every n-th word (n-way interleaving), or simply independent adjacent blocks. Interleaving is useful in any processor when the instruction cycle times are faster than memory cycle times. If the instruction sequence is such that consecutive memory words are accessed, one memory block can settle while the next instruction is fetched from an other block.

Although concurrent processes would occupy different areas in memory, because interleaving is evenly distributed throughout memory space there is a 1/n chance of a block being busy in n-way interleaving. Adjacent blocks are more attractive in this case, since there is not likely to be any interference between two processes occupying electronically distinct blocks.

Nova Interface

In spite of the fact that there are only two processors in the network, many decisions must still be made regarding the nature of their physical (electronic) link. These decisions affect the complexity of the interface circuitry, and the software of both systems. A simple hardware design usually shifts some of the burden of data transfer to the software. A complex design can be efficient in terms of lower required processor intervention. It may also tempt one to control the interface with another processor.

The simplest hardware design is one commonly termed "programmed I/O". It is found in "teletype"-like interfaces, where transfer rates are slow, and there are no critical timing constraints. Its simplicity arises out of the fact that a predetermined amount of data is always passed through a fixed location in the computer (an accumulator or a register, for example), and the only protocol necessary in the interface is a flag indicating "ready" or "done". The program must deal with determining the next data item to be sent, and keeping track of the number of items sent to determine the end of the record. There is also considerable overhead in servicing the interupt which usually accompanies the ready signal.

In contrast, a direct memory access (DMA) interface performs most of the bookkeeping functions of the I/O itself. The program identifies a starting memory address and a count of

the number of items to be sent from this area. A DMA interface transmits the data without interfering with the program until the specified number of items has been sent. Many DMA interfaces make use of micro-processors to handle the bookkeeping and protocol.

The choice is essentially between a hardware intensive, high speed interface and a software intensive, lower speed interface. The electronic protocol may also be considered, presenting several options which complement those two main characteristics.

Three general protocols are common. They are direct bus to bus adapters, n-bit parallel interfaces, and bit-serial interfaces. The latter two often are available off-the-shelf, which is a considerable advantage for small systems. Direct bus interconnections are more complex to design, but offer slightly better transfer rates and more flexibility and control than either parallel or serial interfaces. This is because both memory address and data information are available directly on the bus while only data is present in the others. A direct bus connection also accomodates more than two processors, because some device addressing information is also presented. Serial interfaces are typically one or two orders of magnitude slower than the other two modes.

Graphics Module

Two main attributes must be considered in the graphics area: the speed of the display (number of points displayable without flicker), and the resolution of the display. These attributes are not independent of each other, nor are they independent of other system attributes.

The number of displayable points is of course limited by the resolution of the physical display device, but this is not necessarily the upper limit of the system. Several other factors compete to lower the number of displayable points. A minimal graphics system requires a source of data, a digital to analoque converter, and a display device such as a cathode ray tube. Each has a limit of its own which may be the weak link in the path to the display.

Remote Communication

There are a number of considerations regarding the position of the computer related equipment (e.g., processors, mass storage, terminals) with respect to the user and to the data sources. Often, the data sources are in a hostile environment, such as a mill, or an assembly line. In this study, the environment is a high radiation area, which can destroy semiconductors. Most computer equipment will not function reliably under these conditions, but must be moved to a more remote position.

Computers themselves can produce an environment that is rather hostile to its users. Invariably the most offensive noise is produced by fans and other cooling equipment. The noise level is often high enough to be beyond recommended limits for prevention of hearing loss. We would thus like to ensure separation of the user and the equipment. Additionally, the user should be able to position the controls he requires, whether they be keyboards or buttons or other devices, at any place felt appropriate. Most computing systems are not truly portable, so some method of remotely connecting user controls is necessary.

For a small number of bits involved in device control, one is tempted to run a cable with a wire for each bit. As the number of devices increases, the cable necessarily becomes larger, and soon becomes unmanagable. One option is an extended version of the communication mode used for terminal-like devices, a serializer for a large number of bits. Two-wire connections would then be possible for any number of co-located devices.

Packaging

The packaging of any electronic hardware is a non-trivial consideration. Inappropriate methods can raise problems involving intermittent faults, long-term reliability and stability, and outright malfunctioning of the circuits involved. The source of these problems can be traced to one or more of the

following factors: overheating, structural stress on printed circuit boards or components, vibration, and inadequate power distribution. An examination of this aspect of micro-processors (and computers in general) is beyond the scope of this thesis.

VI. Actual System

Since the central processor already in use in the application is a NOVA 3/12 mini-computer, there is a good rationale for using another NOVA for the second processor. The obvious advantage would be the compatibility of the instruction sets. The cost of a second mini was, however, high in comparison with a micro processor. It was felt that a micro processor could do the work envisioned without undue effort in development, at a much lower cost. This work was initiated in part to determine what contribution new, low-cost micro-processors could make to the field of multiprocessing, in addition to how multiprocessing in general could be used in a relatively normal application. We were thus guided by the financial as well as academic interests in choosing to use a micro-processor.

Hardware Configuration

The final configuration fits loosely into the Flynn category of the multiple-instruction, multiple-data processor. Two processors, a 16-bit NOVA 3/12 minicomputer and an 8-bit Motorola MC6800 were linked using a direct memory access path in the NOVA. In 1976 when this decision was made, the MC6800 was typical of those available. There are currently many more designs to chose from, which are both faster and more complex in

architecture. Many are also 16-bit processors.

Since the two processors do not share similar instruction sets, it is not possible to have dynamic selection of tasks and processors, such as Stone[13] describes. Given the distribution of unshared peripheral interfaces between the two processors, the loss of instruction stream sharing is not considered a major item. A clear assignment of tasks to each processor was one result of this fixed distribution.

The peripherals were distributed to each processor on the basis of their importance to the primary purpose of the system. All those directly concerned with collection or recording of data were assigned to the NOVA. Those concerned with monitoring and casual user interaction were assigned to the 6800. The NOVA peripherals, then, consisted of an analogue to digital converter, a magnetic tape unit, a system disk, and a teletype. The 6800 controlled the graphic display and light pen, and the "button box" - an array of buttons, dials and lights for user interaction. The 6800 has 8K bytes of local memory, while the NOVA shares 48K words of memory with the 6800.

Machine Coupling

In an effort to allow the tightest possible coupling for the least amount of effort, the NOVA I/O bus was extended into the 6800, which then acted like a device controller to the NOVA. This allowed the bi-directional transfer of data using the NOVA

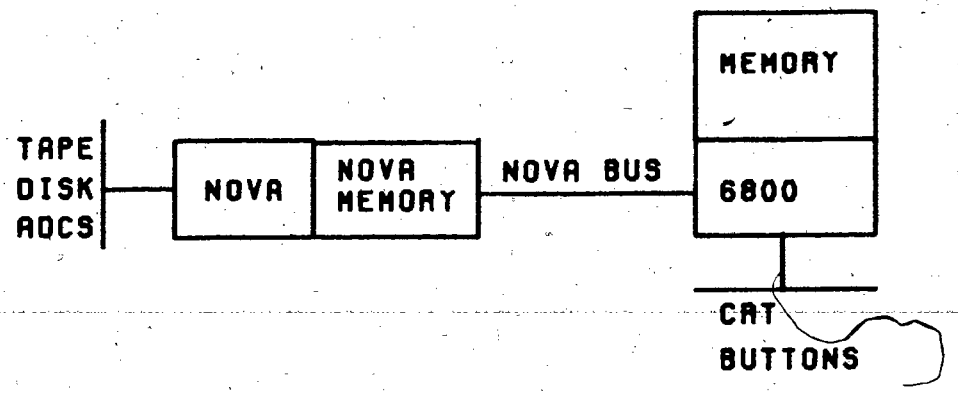

Figure 9. System Configuration

memory as a communications area. These transfers used the DMA feature of the NOVA, which minimizes the impact of communication on the NOVA. The 6800 controlled the bus using programmed I/O, which used a considerable portion of the 6800's time. This is consistent with the intent to reduce the load on the processor which is collecting data, although it was not an optimal solution with respect to the 6800. DMA on the 6800 was not, however, a simple addition to the hardware at the time of design.

Communication

In this configuration, the 6800 was designated to be the logical master of the bus. While the NOVA necessarily retained electronic control, all transactions were initiated by the 6800. The NOVA's contribution to the communications protocol consisted

of setting up a table of pointers to the items to be exchanged with the 6800. These items consisted mostly of histograms for graphical display, and two bit-vectors to contain the status of lamps and switches. The table of pointers was called the "display list" (Figure 10). The 6800 selected a particular item for processing according to switches set by the user.

A single, fixed location in NOVA memory was agreed upon, in advance of compiling software for either machine, to be a pointer to the display list. Each item of the list has various parameters, such as the length and address of a display. It is these parameter tables to which the display list points.

Once the tables have been set up, the only communication tasks which must execute on the NOVA are one to update the display data, and one to take action when switch settings are changed by the user. Neither of these functions requires access beyond shared NOVA memory. The direct bus connection allows the 6800 to chain through the tables at will. No synchronization is necessary between the processors for these tasks, although nothing in the design of the machine coupling prevents this, should it ever be required.

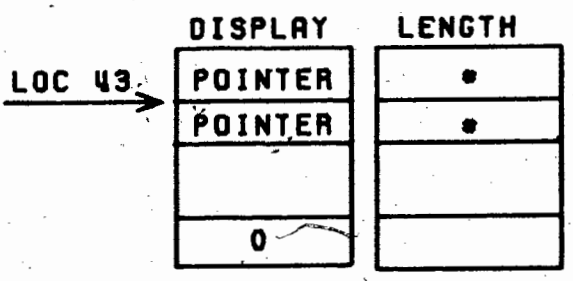

Figure 10. Pointer Chain used to locate areas for information exchange. All pointers and tables are in NOVA memory

Task Partitioning

In the project under study, two groups of tasks can be identified:

1. a group consisting of tasks which are critical to the intent of the experiment itself, principally the acquisition and logging of data, and
2. a group consisting of tasks involved in the interaction with the user.

The tasks in the second group depend on the first group for their data. While they are critical to the peace of mind of the experimenter, the data collected are unaffected by their actions. The first group of tasks, therefore, are given priority for resources over the second group. In practical terms, this

structure may lead to certain low priority tasks being harmlessly bypassed when the critical tasks are heavily loaded.

In partitioning the tasks between the two processors, one fundamental rule followed was that all critical tasks must be executed on the same processor. Additionally, all paths between these tasks must not depend upon the other processor. The effect of this is two-fold. Communication costs are minimized for the critical data, and systems reliability is increased over any other arrangment requiring more than one processor. The rate of failure of two processors is twice as high as the rate of one.

Another consideration in determining the task distribution was dividing the tasks at a point in the data flow where the flow is near a minimum. In this particular instance, for example, the division might have been made between the "analogue input" and "sort events" tasks (refer to figure 8), but the data rate is considerably lower between "sort events" and "create displays". It is also known that there is a great deal of synchronization required at the earlier point, but virtually none at the latter. It is therefore more advantageous to divide the tasks at the latter point.

The second processor was assigned tasks which can be described as terminal tasks. These tasks are ones where the data operated on will cross the interface between the processors no more than once. This is usually some output function, such as the graphical display, or an input function such as the user's

control panel. In any event, the data on these paths is at this time non-critical.

In our system, the sequential flow of data has been broken between the primary and secondary tasks. The secondary tasks are essentially free-running. They are initialized with pointers to all the data buffers, processing what is available in parallel with the primary tasks' filling and emptying of the buffers. Because of the priority structure, secondary tasks in the first processor use up only the resources available after the primary requests are satisfied. The second processor does not load the first other than through memory contention when using the DMA channel. In normal operation, the second processor supervises its own I/O with the first processor. It should be re-emphasized that there is no enforced synchronization between the two groups of tasks, ensuring that the primary tasks never wait on a secondary task.

Communication Costs

One major concern was that communication overhead would be so high that the net gain in power may not be significant. Three test programs were written to determine the overhead of sharing data with another processor. The respective function of each can be briefly stated:

1. A 6800 program to display a 256 word buffer residing in the 6800 memory. Communication is via subroutine arguments.

2. A 6800 program to display a 256 word buffer residing in NOVA memory. Communication is via 6800 programmed I/O on the NOVA bus.

3. A NOVA program to interfere with (2) by concurrently accessing of the same buffer.

Program (1) yields the minimum time to perform the task (on that particular processor). This gives some perspective to the communication costs. Program (2) yields the extra cost of obtaining the data from shared memory for the same task performed by program (1). Program (3) provided information on the cost attributable to the interference of the two machines.

The times were obtained by running the program for a sufficiently long time such that both start-up costs in the program and reaction time of the observer were insignificant. (Timing was measured simply by a stopwatch.) These times ranged from 25 seconds to 91.5 seconds, with estimated errors of ± .5 seconds.

Test Program Number	Time /frame	Comm. overhead	Interference on NOVA
1	6.25mS	1.28mS	n/a
2	23 mS	6.14mS	0.0 mS
3	23 mS	6.14mS	.256 mS

Table I. Communication Costs

The base time, as established by program (1), was 25 seconds to display 4000 frames of 256 points each, or 6.25mS per frame. This is well under the visual flicker limit of 50mS per

frame. As the data is located in local memory, the access operation would typically be an indexed "load accumulator" operation. The time for this instruction is 5.0uS.

When the data was located in NOVA memory, execution time increased from 25 seconds to 91.5 seconds for the same 4000 frames. An examination of the access operation yields the source of the increase. To access one word in NOVA memory required 24uS, nearly a 5-fold increase over that of local access. This alone added 16.6 mS to each frame, for a total of almost 23mS to fetch and display one frame. This is still acceptable in preventing flicker.

The measurements taken above were done with the NOVA halted, so that there was no competition for the I/O bus or memory. Program (3) was designed to check the effects of competition that might arise when the NOVA was active.

It had been safely predicted that the times would not measureable change since the 6800 had priority on the DMA channel. This was in fact the result: it required the same 91.5 seconds to execute program (2) when program (3) competed with it. The overhead was entirely absorbed by the NOVA program, as its architecture requires it to relinquish control when a DMA operation occurres. This time amounted to only 1 second over the total 91.5 seconds of execution. This value can also be computed from the number of accesses, 1,024,000 (4000 frames x 256 points), and the cycle time of a DMA operation of 1.0uS. It was

apparent from this figure that the interference cost to the NOVA was not going to be significant.

Sources of Performance Limitation

These tests demonstrated that it is the protocol, whether imposed by hardware or software, that limits the aggregate data rate, rather than interference or other hardware delays. The addition of communication protocol, in the form of programmed I/O, limited the transfer rate to 40kHz. The maximum bus speed is 1MHz. The time required for the protocol is of course governed partly by the processor speed. To increase the transfer rate, one can either refine the protocol, or use a faster processor. However, as figure 11 indicates, any more than a five-fold increase in processor speed would shift the limiting factors to the the bus speed and the protocol.

	Limit
Bus	1MHz
Processor	200kHz
Protocol	40kHz

Figure 11. Limiting Factors in Hardware

Some Tricks

There is almost 27mS left per frame where useful work could be done, and still avoid flicker of the display. After the final version of the display task was completed, more than 27mS of work had accrued from such tasks as logarithmic scaling and console monitoring. The display program used in the tests was simply too elementary. Rather than find a faster processor, or resort to obscure programming tricks, a long-persistence phosphor was used in the screen. This allowed 300mS of work to be performed for each frame, rather than 50mS, without any real design modifications. This proved to be enough time to prepare each buffer. It is useful to remember that not all solutions require a new program or processor!

VII. Discussion and Conclusions

While gains in performance of orders of magnitude are neither predicted, nor likely, when two processors are cooperating, the relatively low cost of microprocessors makes it economically realistic to contemplate arrays of tens or even thousands of them cooperating to form a single multiple processor system. Such a system could have a large increase in performance over conventional high-speed computers. The problem of controlling these processors, assigning tasks to them, and just physically linking them, in a generalized manner, is still very significant.

In the system discussed, tasks were permanently assigned to each processor. High-speed inter-processor communication was provided through shared memory. The configuration operates as a two-stage pipelined processor operating at the task level. Each processor was free to begin processing new data once its task had passed the results on to the next processor. This is a departure from the conventional implementations of multiprocessor systems, where either an SIMD array type of configuration performs one machine operation on many similar data, or processors in an MIMD configuration are dynamically assigned tasks or portions of tasks which operate on possibly unrelated data. Both of these configurations have been

notoriously difficult to program. For the SIMD system, the data must be structured to fit into the number of processors in the array. For an MIMD system, there is a large overhead in switching tasks between processors, and little way of automatically specifying parallelism in the algorithm.

The configuration chosen for this project was not without its own software problems, the largest of which was the nearly complete lack of development software for the MC6800. This of course is not directly related to parallel processors, but in as much as low-cost micro-processors are candidates for large arrays of processors, the problem is significant. Also, a related encumbrance was the difference in instruction sets. Had they been similar, different combinations of task distributions could have been tried with greater ease. A combination such as a PDP 11/34 and an LSI-11/2 would be more suitable at this time.

The pipelined task approach allows a problem to be segmented into several parts. These parts are programmed on different processors in the pipeline. The programmer minimizes the amount of data that must be exchanged between the processors by carefully choosing the boundaries between the tasks. Additionally, the inherent sequential nature of most peoples' solutions to problems is preserved. Parallelism comes into play for repeated processes on new data.

There is no limit to the number of processors which could be linked in this fashion. Each one could share a portion of its

neighbour's memory for communication. On a system with n pipelined processors, problems need only be segmented into n parts for implementation.

The communication cost borne by the 6800 in this work clearly can be significantly reduced by using DMA to communicate with the NOVA. If the DMA costs were on the same order as those for the NOVA, 1 second per million accesses, the overhead would be under 5 percent of the execute cost of the basic display task. This is quite acceptable. If the machines are equally matched in speed, the amount of work accomplished by them together would be 1.9 times the single machine rate. The implementation of DMA facilities on both processors would provide significant improvement in performance of the 6800.

Shared memory made a very convenient medium for communication. It has the features of high speed, random access, and minimal overhead in hardware.

The intent of this project was to demonstrate that it is in fact feasible to interconnect, without much trouble, two processors for use in a relatively normal application, if certain ground rules were followed. Some of these rules may appear to be in contradiction to the full spirit of multiprocessor systems, but it should be noted that many of those systems have not been entirely successful in their implementation.

The basic guidelines found to be useful were:

- minimize the number of synchronization points (Conway's JOINs) between processors, thereby minimizing costs of waiting for other tasks,
- eliminate dynamic allocation of tasks to processors, as this is a source of communication overhead,
- Locate those tasks with the lowest intertask communication requirements, because communication between different processors is where the greatest losses in a coupled system occur. Both directions of transmission should be examined, i.e, the number of values returned by a task should be considered as well as the number of parameters passed to it. It may be found that some tasks are terminal in this respect: no values are returned. There are several such tasks in this application, including display processing and event recording.

The first two guidelines have been followed, and the third at least partially satisfied.

One of the major problems with nearly all of the current multiprocessor systems consisting of more than a few processors is the lack of viable software with which to easily implement useful programs. I believe one cause of this to be the attempt by the designers to dynamically distribute the load at or near the instruction level. While this may be a useful feature, I have found that a static distribution of whole tasks to be

effective and relatively easy to implement. It would be interesting to determine the cumulative overhead in dynamic distribution of tasks versus the cost of possibly idling processors in a static configuration.

The system that was implemented is extendible to many processors per system by virtue of the limited communication between tasks. In systems requiring more intensive interprocessor communication, one can envision the processors participating in a tree-structured network or graph. Clusters of processors would work on subproblems of the problem presented to the network as a whole. This scheme corresponds to Jackson's description [14] of problem solving using the method of problem reduction.

A two-processor system was fully implemented, and operated for over four years in the application described in chapter IV. It was then superceded by a new commercial unit, which gratifyingly was of a similar design to this system.

Bibliography

[1] Parmentier, R. D., "Josephson Junction Devices for Large Computers", New Concepts and Technologies in Parallel Information Processing, E. R. Caianiello, Ed., Noordhoff, Leyden, The Netherlands, 1975

[2] Baer, Jean-Loup, "A Survey of Some Theoretical Aspects of Multiprocessing", Computing Surveys, Vol. 5 No. 1, March 1973

[3] Flynn, M. J., "Very High-speed Computing Systems", Proc. of the IEEE, 54:1901-1909, 1966

[4] Enslow, Philip H. Jr., "Multiprocessors and Parallel Processing", John Wiley & Sons, New York, 1974, pp. 46-47

[5] Chen, T. C., "Overlap and Pipeline Processing", Introduction to Computer Architecture, Harold Stone, Ed., SRA, Chicago, Ill, 1975, Chapter 9

[6] Barnes et al, "The Illiac IV", IEEE Trans. Comp., C-17:746-757 August 1968

[7] Winograd, S., "On the Speed Gained in Parallel Methods", New Concepts and Technologies in Parallel Information Processing, E. R. Caianiello, Ed., Noordhoff, Leyden, The Netherlands, 1975

[8] Conway, M. E., "A Multiprocessor System Design", AFIPS Conf. Proc., FJCC 24:139-46, Spartan Books, 1963

[9] Ollongren, Alexander, "Definition of Programming Baltimore, 1963 Languages by Interpreting Automata", Academic Press, London, 1974, Chapter 9

[10] Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes", Acta Informatica, 1, 115-138, 1971

[11] Wettstein, H., "The Implementation of Synchronizing Operations in Various Environments", Software - Practice and Experience, Vol. 7, 115-126, 1977

[12] Fung, Kwok-Tung, and Torng, H. C., "On the Analysis of Memory Conflicts and Bus Contentions in a Multiple-Microprocessor system", IEEE Trans. Comp., C-27:28-37 No.1 Jan 1979

[13] Harold Stone and Shahaid H. Bokhari, "The Control of Distributed Processes", IEEE Computer, July 1978, pp. 97-106

[14] Jackson, Philip C., "Introduction to Artificial Intelligence", Petrocelli Books, New York, 1974