National Library of Canada

Cataloguing Branch
Canadian Theses Division

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada

Direction du catalogage
Division des thèses canadiennes

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

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

VARIABLE STRUCTURED FAULT TOLERANT PROCESSOR

by

W. E. Davidson

A thesis submitted to the School of Graduate Studies

in partial fulfillment for the degree

of

Master of Applied Science

Department of Electrical Engineering

Faculty of Science and Engineering

University of Ottawa

Ottawa, Ontario

July 1975

TABLE of CONTENTS

ABSTRACT

With the extended use of control computers the need has grown for fail soft computing. In many of these cases, the control system, even in a fault condition, must maintain a certain degree of control, eg. life support systems, air traffic control, steel making, oil refining, utility control, etc. This thesis will suggest a computer architecture especially suited for high reliability control situations.

The system is designed around two building blocks, EPUs (Elementary Processing Units) and loaders. Each EPU is identical to every other EPU as is each loader identical to every other loader. To the programmer they constitute a flexible hardware structure which he can manipulate according to his program requirements, or which are automatically manipulated in the case of a fault condition. Minimum operating conditions would be one loader (control) and on EPU (processing). The degree of reliability of the system is user defined by the number of loaders and EPUs he invests in his system.

LIST of FIGURES

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to his supervisor, Professor M. Krieger, for his guidance and suggestions, making this thesis possible. He would also like to thank Professors S. Shiva and P. Thompson for their helpful assistance during Dr. Krieger's absence. Special thanks also goes to Paul St-Arnaud* for his work in this field leading to the authors subsequent interest in this field.

* deceased

CHAPTER I - INTRODUCTION

I-1 Brief Description of Control Computers

A control computer, for the purposes of this discussion, will be defined as a system which dynamically commands, directs, or regulates itself, or some other system.

Control systems have become very popular since the advent of mass production techniques. This is because they relieve the need for constant human supervision and their speed allows for closer control of the process. In many of todays control systems applications, the process is very nearly completely controlled by the control system. In cases like this if the control system loses control due to some fault in the system, a shut down sequence must be initiated. Here a human supervisor could spot the error, and initiate the shutdown. However, in many of these systems, the process cannot be shut down without large loss of money, or possible danger to those involved. Added to this problem is the fact that the error may not be obvious until it is too late to regain control. In most cases it is impossible to have a complete human backup to the control system to take over even if they could spot the fault in

time. In cases like this reliability is the most important criterion, and even in a fault condition the control system must be able to maintain a certain degree of control.

Examples requiring this sort of control system would be oil refining, steel making, air traffic control, utility control, etc...

The heart of any control system is its feedback capability. That is, its ability to monitor conditions in the system's performance and along with a set of desired operating conditions acted upon by some algorithm, modify the operation of the system. Basically there are two methods of representing this information, and executing the algorithm; analog and digital, or of course some combination of the two.

In an analog system the information is coded according to a proportionality scheme. That is, in an electronic control system the voltage or current is proportional to the physical quantity it represents. This has the advantage that in a single unit of time, a single line can carry a large amount of information. The accuracy of the system is dependent on how small a change in voltage or current the components can accurately discern. In all but the most expensive analog equipment, this limits the accuracy to two or three significant digits.

In a digital system, the information is converted to a discrete number system. Today all practical electronic digital systems use a binary number scheme. That is, a single line in a single unit of time can represent a weighted bit of information which is either true or false (on or off). This means it requires n units of time, or n lines, or a combination of units of time and lines totalling n, to give 2'n discrete levels of information.

Digital systems have the advantage over analog in logic and decision making situations because of their simplicity, speed, and their direct handling of devices such as relays, solenoids, switches, fuses, locks, counters, annunciators, panel lights and panic buttons, all of which are digital devices. Analog devices have the advantage in situations where linear calculations must be made and where input-output devices are potentiometer and panel meter types.

In many systems, the above advantages dictate the type of electronic system used. However, when reliability and accuracy become important considerations, other characteristics of analog and digital systems must be taken into account. The most important consideration here, [redacted]noise. In analog systems, noise added to the signal changes the signal content, since information is coded according to a proportionality scheme with signal level. Analog devices also have a tendancy to drift with

time, temperature, humidity and various other enviromental conditions. In medium or large complexity calculations, small amounts of noise contributed by each analog input or computing element add to degrade to a large extent the accuracy of the solution. In digital systems a signal is either true or false and there are threshold levels for representing these two conditions. These threshold levels are either actual voltage levels(bipolar trans.) or percentages of supply voltages(cosmos trans.). This means that noise signals smaller than these threshold levels can be completely ignored as they do not add or propogate through the system, since each digital component reshapes the signal, as it processes it, to well within these threshold limits. Noise is a major reason for the dominance of digital computers over analog computers where complex control systems are required.(1)

Analog controllers, servo systems, chart recorders, panel meters and small analog computers are often simpler and cheaper than their digital counterparts. However this advantage is decreasing since the advent of integrated circuits and large scale integration, where complex functions can be realized relatively simply and cheaply using pre-made packages and functions. For complex control situations digital methods can deliver accuracy and perform types of control beyond the ability of an analog system at any cost. Using solid state digital control, analog and digital devices can work together using A/D,

D/A converters. Better yet, direct digital sensors and actuators can be used completely in the control system. For these reasons, the rest of this thesis will discuss solely digital control systems.

In digital control systems, there are two main types of hardware implementations in use today; relays, and solid state devices. Relays have the advantages of being cheap, easy to service and maintain, and extremely noise resistant. The disadvantages of relays are that they are slow, bulky, limited to simple function per device, have high power consumption compared to solid state devices, and have a limited lifetime. Solid state devices on the other hand, are extremely fast, low power consumption, very small, and a nearly unlimited lifetime. With large scale integration size is even further reduced. However because of the high speed and sensitivity, solid state devices can respond to noise that relays would safely ignore. This thesis will be discussing a general control system computer for use in complex and high reliability situations, where the relay's bulk, slow speed and limited lifetime make it an unacceptable form of implementation. Solid state and in particular medium and large scale integration devices are used in these complex control applications.

Ultimately, the digital computer interfaced with the control system gives the greatest degrees of freedom and

flexibility to the system designer and user. The most important feature of using a computer is its ability to modify the control scheme merely by changing the program. To the designer, this means simplified hardware design and the ability to try out different control algorithms on the actual system. To the user, it means he is not limited with a fixed control system, but a flexible one which allows him to change his system and still be able to adapt the control to the new system. Using a digital computer as the main component in a control system does have a few disadvantages. It is slower than dedicated hardware doing the same job. It means that the computer must be extremely reliable, as the whole control system is dependent on this one component. General purpose computers can be used, and in many cases are used, however the user may be paying for more computing power than he needs and still not have the I/O handling capability he wants without paying for a complex interfacing system. To meet this demand for computers specially suited for control system applications, many computer manufacturers have designed powerful mini-computers which can be ordered with many different configurations and options to match the user's needs.

This thesis will present a computer architecture whose configuration varies dynamically with program requirements and fault conditions of the machine. It can be repaired or expanded without machine shutdown. It can be expanded from mini-computer size to large computer system capability, yet still maintain

program compatibility, and it also allows for parrallel processing under programmer control. In short, this design is hopefully an ideal control computer for high reliability control system applications.

I-2 Current Concepts in Control Computers

A control system must be able to sample the vital conditions of the system to be controlled. Along with a set of desired conditions, which can be selected via the operators interface, the control system must make decisions according to the control algorithms, and output the appropriate inputs to the system. Unless the system is very simple, there will be more than one control algorithm.

There are basically two ways of approaching this problem of multiple algorithms. The first method is by breaking the control problem down into its individual control algorithms, then designing small individual control systems for each of the algorithms. Since most control problems cannot be broken down into completely independent algorithms, there must be some means of intercommucation between the individual control systems. In many cases the intercommunications are control systems in their

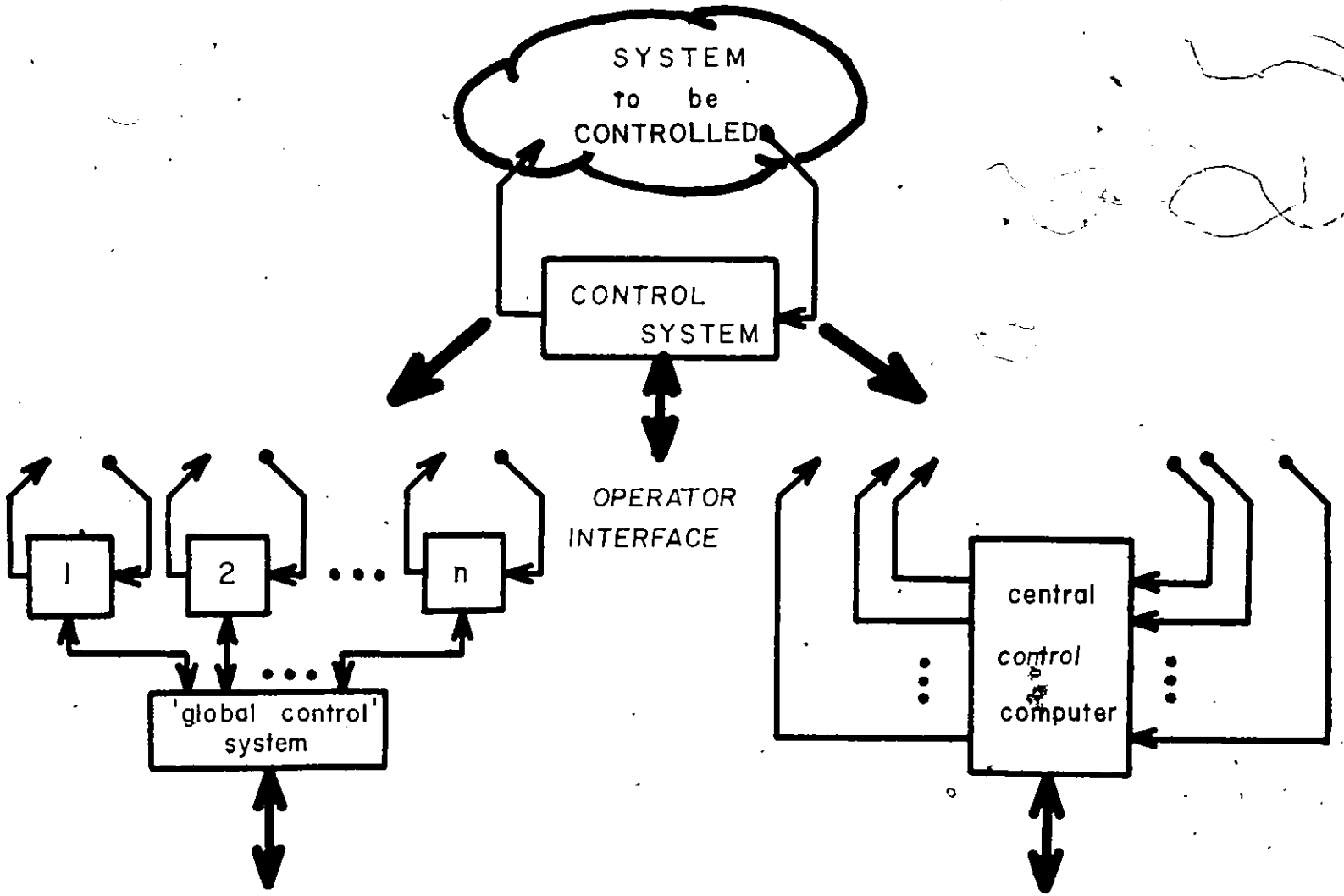


Fig 1 - 1

'multi-algorithm system' approaches

own right. These provide 'global' control and allow for a centralized operator interface.

This type of appraoch is generally called a 'distributed system', or 'hardware approach' and generally yields a faster and more efficient control system.

The other approach, generally referred to as a 'centralized system' or 'software approach', is to use a central computer and by means of software, time share the various algorithms in the control problem. This means the control computer must have a sophisticated I/O system with good interrupt capabilities. The advantage here is that the control computer can be a general purpose machine with special I/O options. In this case much less hardware need be designed and faster installation can be expected. The operator interface being connected to the central computer which executes all control algorithms means this one interface can modify any or all of the algorithms.

This second type of approach is generally slower requiring a great deal of sophisticated programming, and in many cases the user buys more computing power than he needs simply to cut down on this programming burden.

I-3 Common Failures in Computing Systems

Even with the advent of high reliability and long life components, modern day computers are subject to hardware failures. In addition to hardware failures, there are two types of error which must be considered; design and wiring errors, and program errors. A lot of research and work has been done in the last decade to correct, or at least reduce these three sources of error and certain basic techniques have arisen from this work.[2,3,4,5]

Hardware failure, the most damaging of these, since it occurs after the system has been debugged, is handled by means of redundancy in hardware, software, or time. Hardware redundancy can be, parrallel hardware with majority vote, standby hardware with automatic switching, or hardware using coding techniques. Software redundancy consists of diagnostic programs intrinsic in the operating system program, with recovery action subroutines. Time redundancy constitutes repetition of programs or segments, with checks made on 'reasonableness' and duplication of important steps.

Initial design and wiring errors are reduced by using modular units. These are much easier to perform diagnostics on and where many modules are of the same type, this greatly simplifies testing techniques.

Program errors are difficult to eliminate. Higher level languages help, but there is still the problem of program correctness. Some work has been done on 'program correctness' programs, but the main method of verification is by executing test cases.

The hardware approach is more subject to hardware faults, however, due to its distributed nature, a hardware fault need not effect a whole system, but only that part in which it occurred . This allows the designer to build redundancy in areas with the highest failure rates, and not in other areas. The main drawback of a hardware approach results from its distributed nature and specialized parts. This gives the system an inherent rigidity, and great care must be taken in its initial design and programming.

In a software approach, the computer used is not necessarily a 'one of a kind' machine and is therefore more apt to be completely debugged and working properly. Its disadvantage is that it requires a sophisticated time sharing scheme along with an interrupt priority handling of I/O and emergencies. To maintain adequate control, the control computer must be large enough and fast enough to support and execute all the control algorithms within the specified time restraints of the system to be controlled. This means that it is highly succeptible to program faults. Also, any program or hardware faults affect the

whole system in most cases and not just an isolated part of the system. The structure suggested by this thesis will take advantage of both these approaches.

The computer architecture described by this thesis is mainly concerned with the first type of error: hardware failure. Its structure allows for failure detection and recovery, with an equal or lower level of efficiency. Furthermore, this computer architecture consists mainly of only two types of structure. Both are simple and easily modularized, thereby reducing design and wiring errors. Software errors are not specifically covered by this design, although high level languages can be used and the programs can be easily changed.

I-4 Variable Structured Solution

As mentioned earlier, this architecture will be designed to deal mainly with hardware failures. This can be divided into three basic problems;

1. Recognizing that an error has occurred.
2. Saving correct data and program flow.
3. Effecting a repair or replacement of the faulty part.

Redundancy Approaches

Hardware	Standby	effects a replacement of the faulty part however does not detect errors by itself.
	MajorityVote	automatically detects errors and maintains correct data and program flow. It does not, however, replace or repair the faulty part.
	Coding	can be designed to simply detect and/or correct errors, however it does not isolate the faulty component, it works solely on the data stream.

Software & Time — can be designed to detect certain errors providing a sufficient amount of the machine is still working. It can also be designed to isolate the faulty hardware, but cannot by itself replace or repair it.

Standby hardware is the only approach which actually replaces the faulty component. It disconnects the faulty component from the system thereby allowing it to be removed and repaired. Standby hardware must be combined with some other approach in order to recognize the error and maintain data and program integrity. In a standby system, one unit is idle until the other unit fails, this being the redundant unit.

In a majority vote system, at least three identical units must be used. Each capable of handling the complete problem. These units work in parallel, and at every decision point or output, a comparison is made to ensure that all three agree. In the case that all three do not agree, the dissenting unit is disconnected, and the output is taken from the agreeing majority.

In a three unit majority vote system, two units are redundant and the error correcting capability fails after one of

the units fail. However it retains error detecting capabilities. More parallel units can be added to the system to improve error correcting capability, however this represents a large investment in additional units, each as complex as a single unit system.

Coding does not correct errors in its own hardware, but can be effectively used around fault prone hardware.(6)

CODING SYSTEM

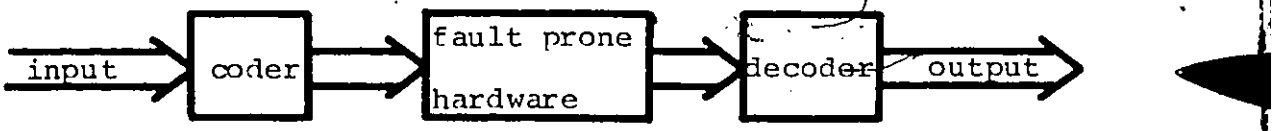


fig. 1-2

In effect, coding provides data redundancy in the fault prone hardware. It will detect and/or correct errors in this hardware within the limits of the coding scheme.

Let us consider a standby system with software or time redundancy used to detect errors.(7)

This means we have one complete unit idle as well as extra software not contributing to the actual program flow. Let us now add another feature to the system. Let the idle unit be continuously executing a self testing and evaluation routine.

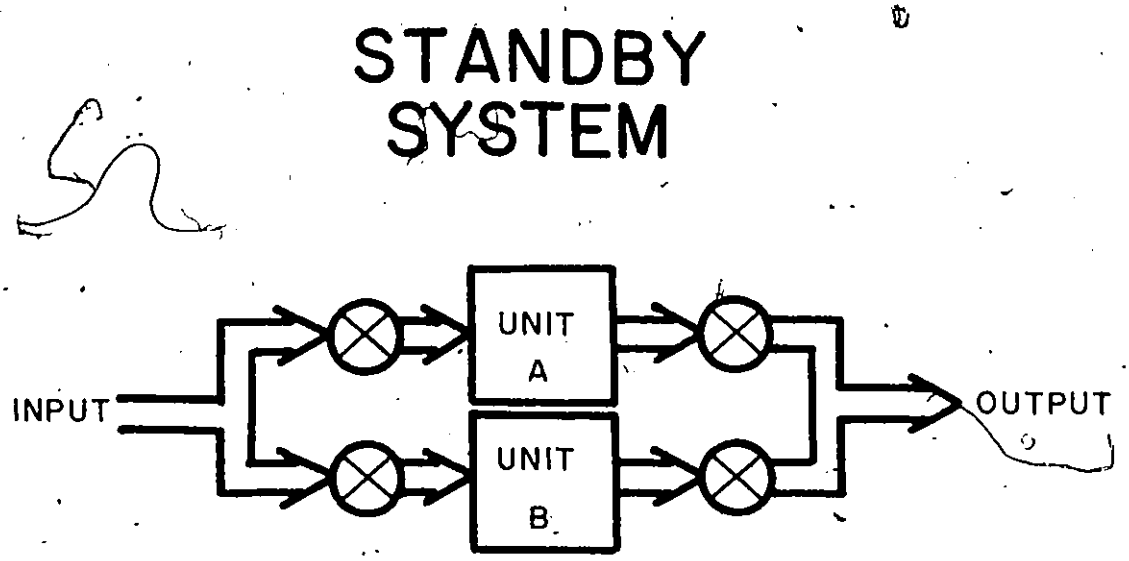


fig 1-3

This means that if and when the idle unit is switched into the Program flow, we can be assured that it is operating properly. This feature has represented no sacrifice of time or efficiency, since it is being executed on the idle unit, yet it adds a degree of confidence in the integrity of the system.

Let us now consider an interleaving of the parallel units in this standby system.

Interleaving is a common technique, often used in memory systems to increase speed. This technique uses parallel units working on the same instruction flow, but alternate instructions. Speedup is achieved by overlapping certain cycles of the instructions. The degree of interleaving refers to the number of parallel units. Any process which has a setup time before execution can be speeded up by this technique.

eg. suppose an instruction cycle consists of

SINGLE INSTRUCTION CYCLE

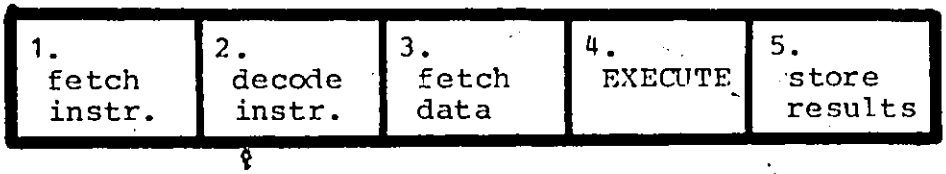


fig. I-4

we could interleave as follows

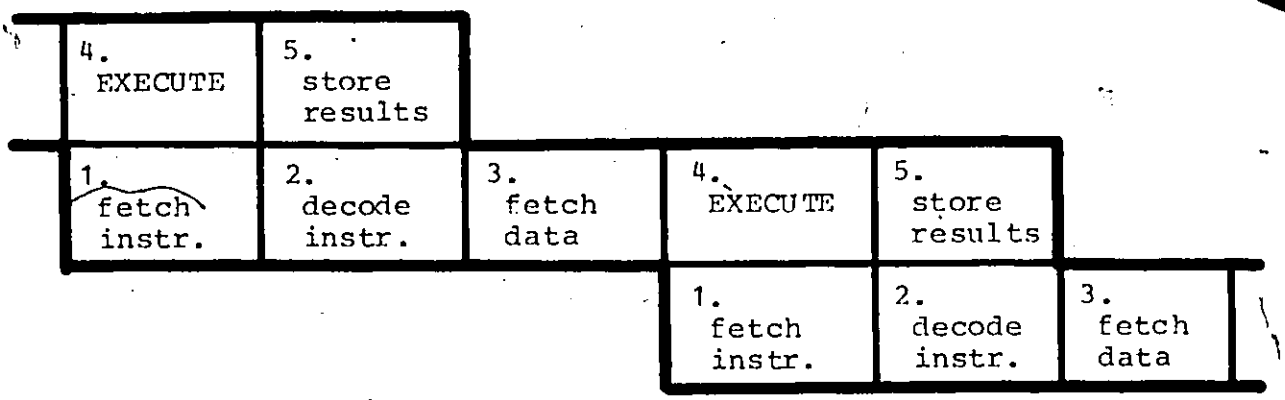
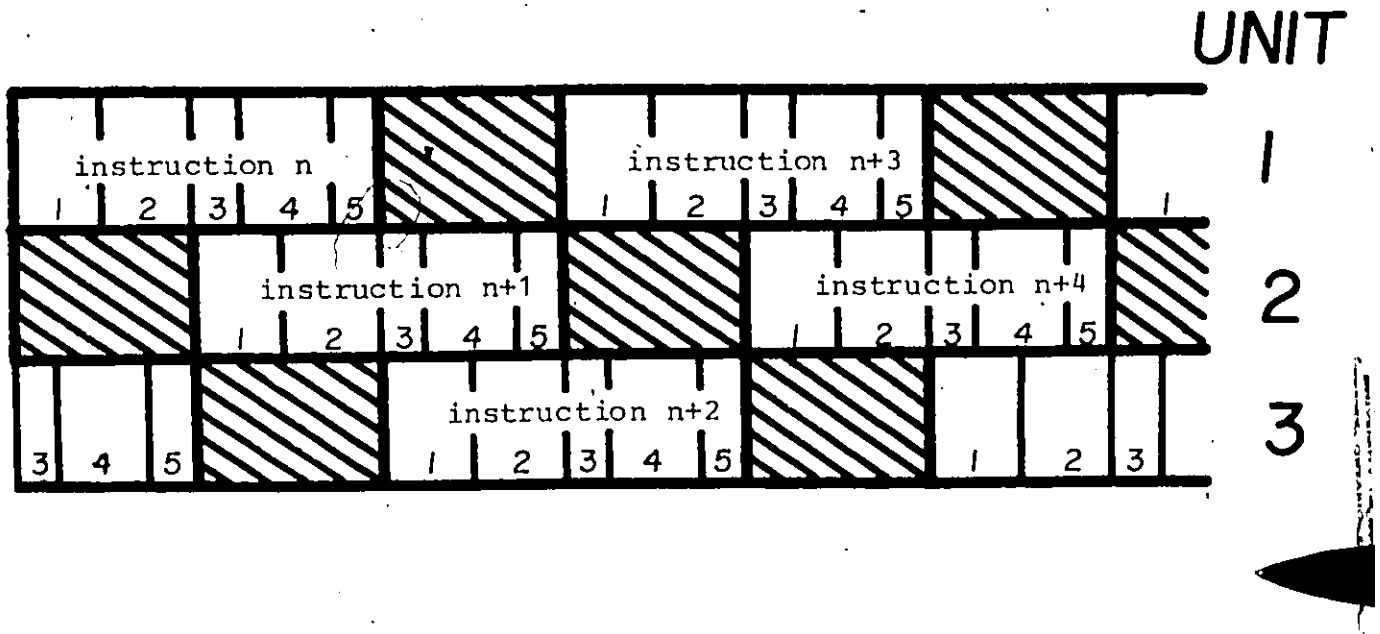


fig. I-5

In a register oriented machine, the fetch data and store results cycles would be very short compared with the other three cycle times. Thus a three level interleave would look like:



cycles

1. fetch instruction
2. decode instruction
3. fetch data
4. EXECUTE
5. store results

unit idle during this time

fig. I-6

It can be seen, that by interleaving we have greatly increased the speed of execution. It also means, that each unit will have some idle time. This idle time will increase with the

degree of interleaving. Let us consider that during the idle time the unit is performing diagnostics on itself.

Since the system as a whole now operates much faster than a single unit, each individual unit can be made much simpler than a single unit doing the same job as the system. Because of all this distributed hardware, most of the register storage connected to a unit can be considered as common to the system, thus each unit can afford to have fewer facilities.

Our system now has greatly reduced hardware investment compared with our original standby system. The software and time redundancy can be substantially reduced since much of the diagnostics or error detecting is done during the unit's idle time. When a unit fails, the level of interleaving is simply reduced.

The architecture suggested by this thesis will be a standby system with interleaving. The degree of interleaving will be flexible and dependent on the routine being executed as well as the number of units in operating condition. The number of units, hereon referred to as 'EPUs' (Elementary Processing Units) will be dependent on the investment and degree of reliability restraints for the overall system. When only a critical number of EPUs remain in operating condition in the system, data and program flow integrity is maintained as long as one EPU is

operating, however speed of operation is reduced with every failing EPU due to reduction in degree of interleaving.

This thesis will discuss the hardware aspects of a system for handling a program flow in the above manner. This system, called a 'String', has the responsibility of setting up software routines in an interleaving scheme, assigning EPUs in or out of the program flow, maintaining data and program integrety. The thesis will also discuss a complete control system using several strings and give program examples to show operation as a complete system.

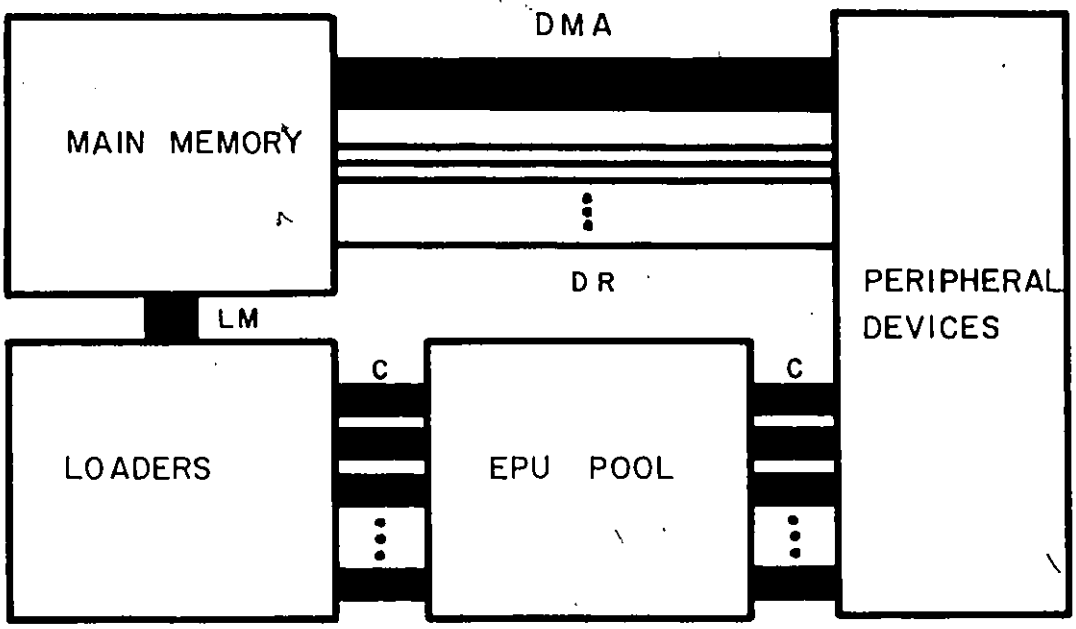
CHAPTER II - THE SYSTEM

II-1 Overview of the System

The architecture suggested by this thesis can best be described as a two instruction level machine. The lowest level instruction is the 'instruction set'. One instruction set can be handled by one EPU (Elementary Processing Unit). The upper level of instructions are actually routines made up of instruction sets, either from main memory, or pre-packaged routines in the Operating Functions Library. These routines are variable size subroutines or algorithms executable by a 'String'. A string being a small processing system made up of one loader and several EPUs.

A program in main memory is made up of these macro-instructions. The macro-instruction routines are loaded into a string which then distributes the instruction sets to the EPUs. In the strings, hardware redundancy in the form of standby units is used. However instead of one unit operating until it fails, the program flow is actually interleaved through many EPUs. The degree of interleaving is dependent on the size of the routine, and the availability of EPUs.

THE SYSTEM



BUS NAMES:

C	Communications	Buses
DMA	Direct	Memory
DR	Direct	Registers
LM	Loader / Memory	Bus

fig II - I

Interleaving allows much faster operation plus gives each EPU idle time during the program flow to allow diagnostics to be preformed. This faster operation means each EPU can be much simpler than a single CPU executing the same problem under the same time restraints. In fact, each EPU can be thought of as an accumulator with its own instruction register, and simple arithmetic logic unit. Diagnostics being performed during idle times means there is a speedup again in operation since program flow need not be interrupted to maintain a diagnostics routine.

The system, in its fully expanded and working state, resembles the hardware approach, as more and more parts fail, it

takes on the aspects of a software approach. The heart of this computer architecture is a structure called a string. The whole computer system is made up of several of these strings connected to a common memory. The system may have a large number of components, however, they fall into only three classifications; the main memory, the loaders, and the EPUs.

The main memory is by far the largest of these components. The loaders and EPUs are simple and identical within their classifications.

Operation

Operation is straight forward. Each loader contains a Program counter for main memory, and a program counter for the string maintaining the state of execution of the subroutine operating. The executive and overall system programs are kept in main memory, but organized as a series of subsets, or subroutines which can fit and be executed by a string. The loader's main memory program counter keeps track of the strings executing position in terms of these main system programs.

The first instruction in the subroutines stored in main memory is an instruction to the loader. This informs the loader of the routines size, and whether or not it requires pre-packaged

routines from the operating functions library. The operating functions library is a protected section of main memory, or a PROM/ROM (read only memory) device which contains the common subroutines used in programming. As soon as this instruction is received, the loader starts generating requests for EPUs from the EPU pool. The complete subroutine and initial data are loaded into the loader's backup memory and distributed to the captured EPUs. When the routine has been completely loaded and distributed to the EPUs, communication with the main memory is terminated and the routine initiated.

In this way, each loader acts independently of the other loaders, and can work on independent sections of the main system programs simultaneously. This is important in many control applications where several separate control algorithms are required. The main memory provides any intercommunications required between these algorithms.

Main Memory

Since multi-port memories are not practical, a queuing scheme must be set up for the main memory. This could be set up as a first come first served, or a first come last served, or even a priority scheme, whatever the customer requires. Since most control algorithms contain many largely repeated loops, and

the main memory is not used in the instruction level transfers, but in only subroutine transfers, this means each loader spends only a small percentage of its time in communication with the main memory.

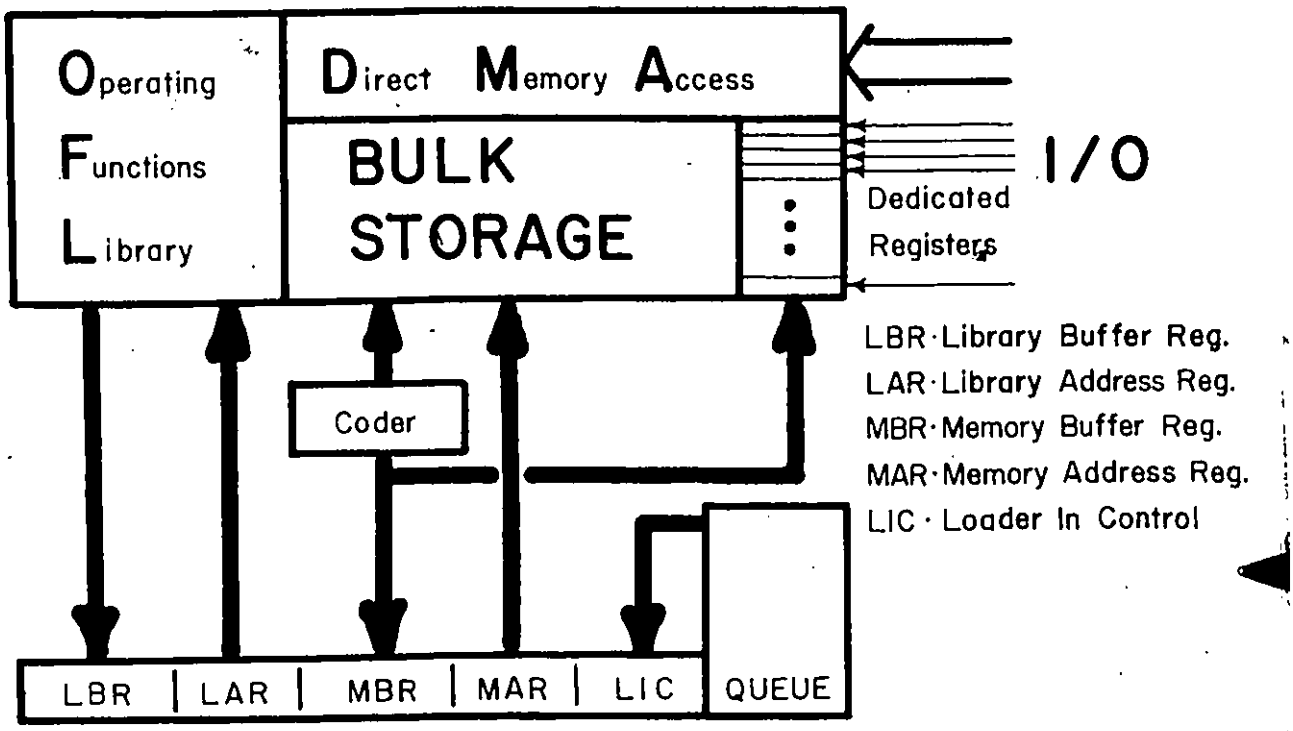


fig II - 2

The main memory would also have a direct memory access port for handling high speed bulk data peripheral devices. This could be realized as a separate circuit, or made to resemble a loader with a very high priority on the memory/loader bus. The memory would also contain dedicated registers which are directly connected to external devices. These registers would be addressed

like any other word in main memory, but their value would directly control, or be controlled by some external device.

So far all peripheral devices have been routed through the main memory. These can also be handled by the strings themselves, as will be described in the section dealing with strings. This would remove a large share of the responsibility for system integrity from the main memory.

System Integrety

The memory/loader bus, since it is common to all loaders, also provides for direct communications between the loaders. These communications would not be in the form of data transfers, but in the form of status checks. When a string has finished its assigned routine, it checks to see if any loader is in a failed state before picking up its next assignment from main memory. If another loader is in a failed state, it picks up the failed loader's program counter and stores it along with its own. The two program streams are then alternately executed according to a priority scheme, until one of the program streams is picked up by a free loader. The concept of a failed loader could be extended to include loaders in a critical state. That is, loaders which are operating a high priority routine while having only a few EPUs committed, thereby making operation slow.

The main memory's integrity can be maintained at a higher level by means of an error correcting code.[8] This is feasible since the main memory is simply a bulk memory and not directly used in execution. This means the memory can be organized into long words thereby allowing large codes with reasonable data rates,[9] and slower simpler routines can be considered in the encoding/decoding procedures. The main memory still remains the weak link since it has no immediate backup, however, under ideal program conditions, it is little used, and certainly less used than in a normal computer architecture.

With careful programming, the number of calls to main memory can be greatly reduced. If a particular routine is used very often, then a string (a loader with its assigned EPUs) could be dedicated to that routine. Then instead of calls to software routines in main memory, there would be calls to that particular string. This eliminates the need to reload a routine every time it is called. Another way to reduce errors is to make use of the pre-packaged routines in the operating functions library. If this library is kept separate from main memory in a PROM/ROM device, the errors in reading can be reduced. This is because a PROM/ROM device once programmed is very reliable, it cannot be written over, it is non-volatile, and has the added attraction of being fast compared with core memories.

II-2 The Processing 'String'

The job of the string is to execute small algorithms assigned to it by the system programs in main memory. Each string consists of a loader and one or more EPUs connected to a common communications bus. The number of EPUs connected to a loader forming the string is dependent on the complexity of the job it is performing, and the availability of EPUs. If only one EPU is available, the loader acts as an instruction and data memory, and the EPU must fetch a new instruction set from the loader each time it has completed its function. In the ideal case, enough EPUs are available, such that the complete algorithm assigned to that processing string can be distributed to the EPUs. In that way a fetch instruction cycle is not required as each instruction set is executed, control is simply passed on to the appropriate EPU. This represents time savings in fetch cycles where the algorithm contains program loops. It also represents time savings in that all instruction and addressing information is decoded when the EPUs are loaded, before control has even been passed to one of these units.

Forming a String

The important features of these strings are the interconnections between loaders and EPUs, and the string formation once a loader has picked up a routine from main memory.

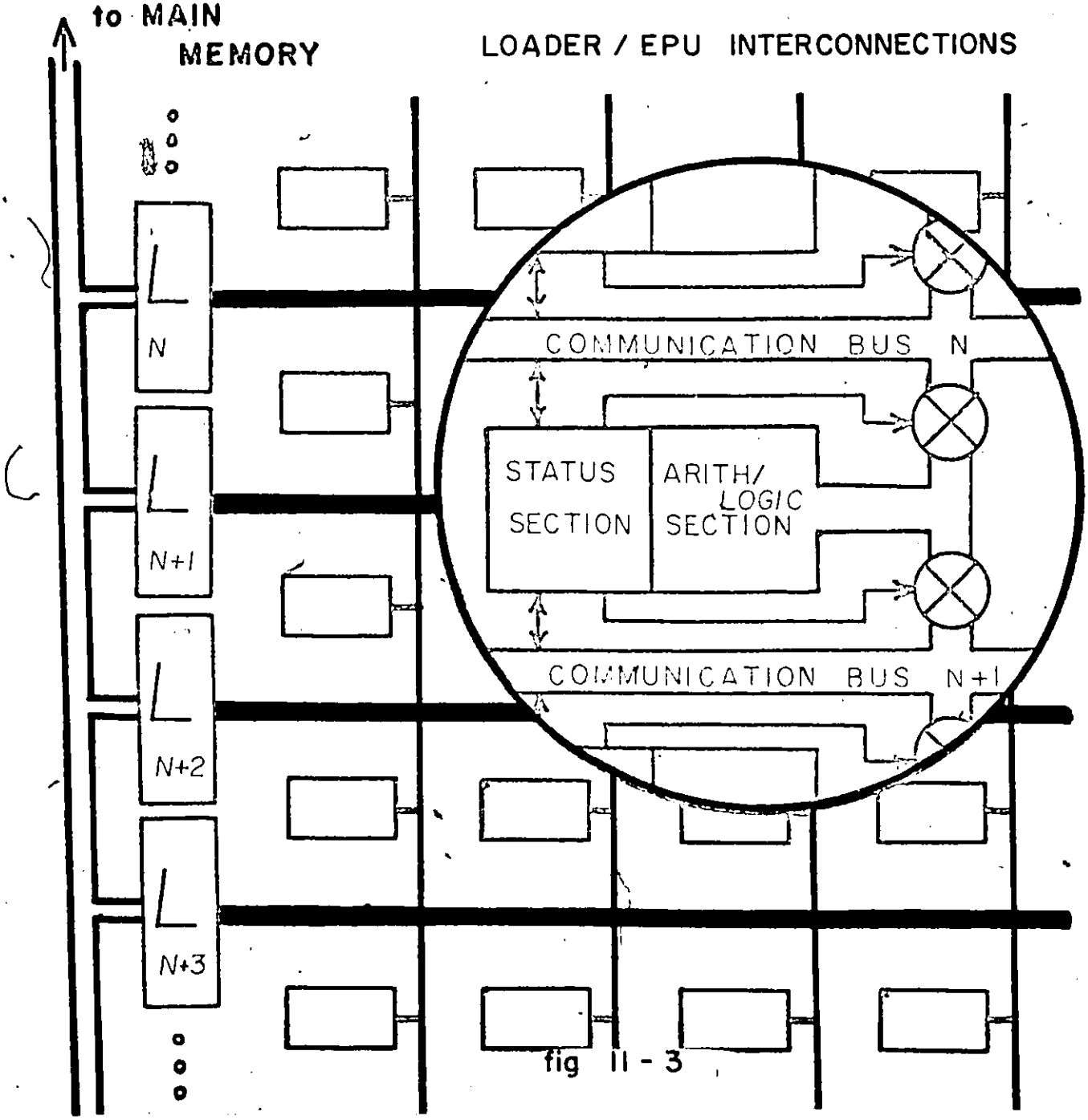


fig II - 3

Each loader has associated with it a communications bus. This communications bus allows for all communications between the loader and the EPUs. Ideally, each EPU in the EPU pool would be connected to every loaders communications bus, thereby giving each loader access to every EPU. This would give a loader searching for a free EPU the best chance of finding one. This would mean fewer EPUs would be required in the EPU pool, since the maximum number of EPUs for each loader would not be required but only a total of (number of Loaders)x(average number of EPUs per routine) would be required.

A more practical solution would be to have each EPU connected to two communications buses. In that way if n EPUs are connected to to the communications bus of loader i, n/2 are shared with loader i-1, and n/2 are shared with loader i+1. This means each loader has access to n EPUs yet the total number of EPUs in the EPU pool is only (number of loaders)x(n/2) . As long as n/2 is equal to or greater than the average number of EPUs required by a routine, and n is equal to or greater than the maximum number of EPUs required by any routine, then ideal string conditions exist.

The loader and its communications bus form the initial structure of the string. The loader must obtain the algorithm and data from the main memory, and re-store it in its own memory. In the model we are constructing using this structure, a two instruction three address format is used. This format along with

a data word form an instruction set. Each instruction set is a function for one EPU. Therefore a routine requiring 14 instruction sets would ideally form a string of 14 EPUs.

Instruction one acts on addresses one and three, whereas instruction two acts on addresses two and three. These addresses can be origins or destinations for data handling instructions, jump instructions or as qualifiers for the instruction. As these instruction sets are loaded in from main memory, they are assigned a self address according to their position in the routine. The addresses used in the instruction sets refer to the instruction sets associated with these self addresses. The instruction sets are also stored in the loaders memory under these self addresses.

When an instruction set is loaded from main memory, a request is also put out on the communications bus for a free EPU. A free EPU being one in operating condition and not already assigned to a loader. This EPU request line on the communications bus has built in delays to avoid simultaneous answering. When the status section of an EPU responds to the request, it deactivates the request and gates the EPU onto the correct communications bus. This EPU then picks up the instruction set and its self address. The EPU then changes its status to busy, and will only respond when addressed by the appropriate self address, until it fails or is released by that loader. If there was no response to the EPU request, the loader simply continues loading instruction

sets fron the main memory as if the request had been answered and the EPU loaded. Remember, no information is lost since the instruction sets are also stored in the loaders memory.

Operation

Once the complete algorithm has been loaded from main memory, the loader initializes the strings program counter, and starts the routine. This program counter is separate from the main memory program counter and is only responsible for the execution state of the routine in the string. It is connected to the communication bus and identifies the EPU which has control. This program counter is automatically incremented each time and EPU has completed its instruction set, unless modified by a jump instruction in one of the EPUs.

If there is no response from any EPU to the address in the program counter it means, either there were too few EPUs, or that EPU has failed. The loader then puts out a request for a free EPU. If there is a reply, then that EPU is captured and loaded with the self address and instruction set from the loaders memory associated with the address in the program counter. Execution is then continued. If there was no response, the loader searches until it finds an EPU on its communication bus. When an

EPU is found, it is reloaded with the self address and instruction set according to the address in the program counter. Execution is then continued. In this way failures occurring during program execution can be tolerated, since the failing EPU simply goes off-line, and another EPU is assigned its task. When too many EPUs fail, such that there are not enough to hold the complete routine, execution can still continue. However, it is slowed down by the fetch EPU cycle. In worst case conditions, operation can be maintained by one EPU, however, the EPU must be reloaded after completing each instruction set.

The Loader

The loader also monitors all load and store cycles. Since the destination and origin addresses are on the communication bus, the loader can update its data store copy of the EPUs. Also in the case of a data load cycle, where no EPU responds, the loader can supply this data from its storage. The other main responsibilities of the loader to the string are; to load or store blocks of data in main memory at the request of an EPU, to store data in main memory and release all EPUs at the termination of the routine, and to provide timing signals and checks on execution times in case an EPU has failed during its execution of an instruction set.

The EPU

The EPU is made up basically of two sections, a status section and an arithmetic logic section. The status section maintains the state of the EPU, either busy or free. It controls the gating of the EPU onto the communications buses. It keeps track of the operating conditions; idle, loading, slave, executing, or failed. Whenever the EPU is in an idle condition, or free state, the status section continuously exercises the arithmetic logic section checking for failures. If a failure occurs, the EPU goes off-line in a fail state.

The arithmetic section responds whenever the status section indicates its address for a data transfer, slave operation, or if passed control by the program counter. This arithmetic logic section can be fairly simple, and have a limited instruction repertoire.

II-3 Input/Output handling

In addition to its other functions, the string can also handle I/O devices. This means that where there is more than one string operating, more than one peripheral device can be handled at one time.

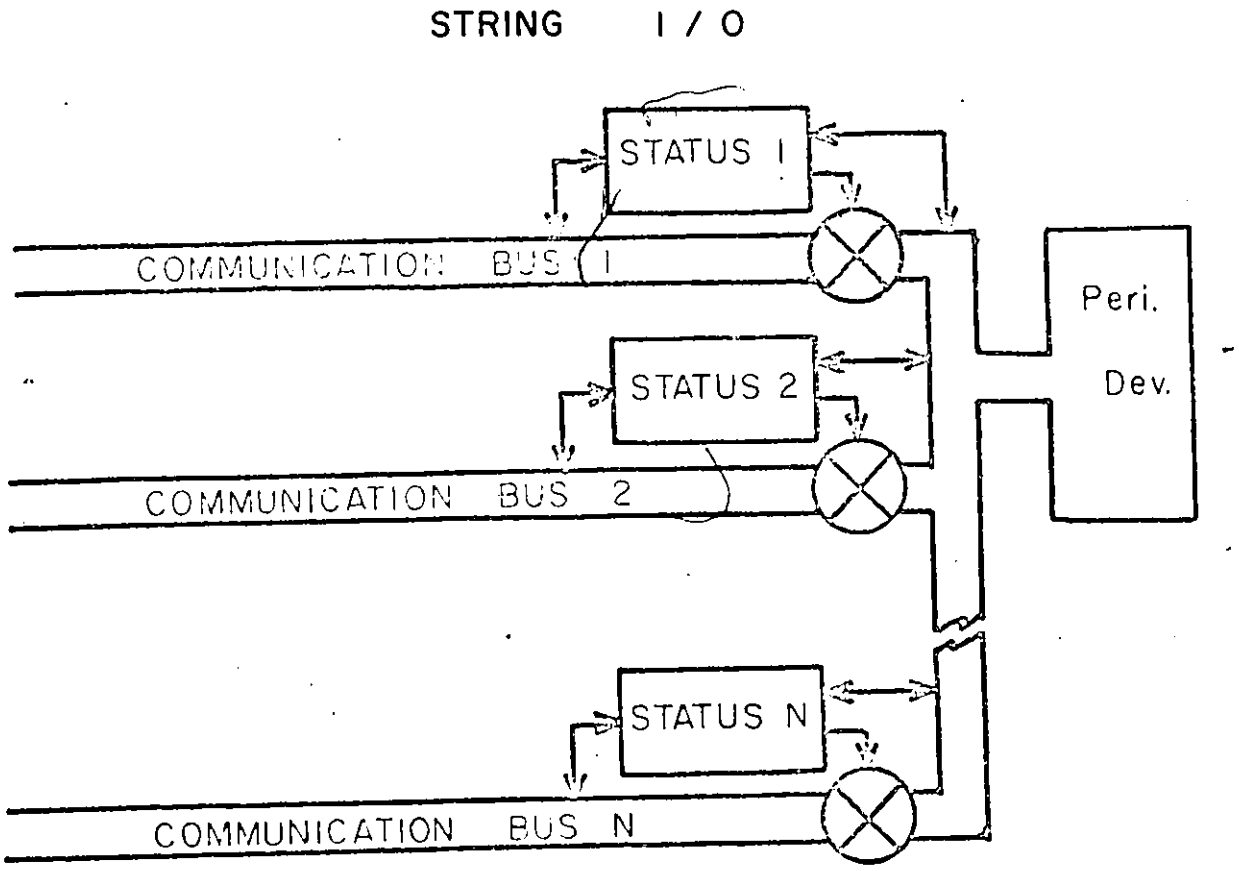


fig II - 4

In control applications where there are many low data rate devices with information belonging to different algorithms, this proves to be a great advantage. These peripheral devices can be considered as EPUs, the only difference being that they

respond to a different set of instructions. The loader when receiving the algorithm must be able to recognize I/O instructions, and the make the request to the pool of peripheral devices instead of the EPU pool. The status section of the peripheral device for that communications bus picks up a self address exactly like an EPU, and the peripheral device is loaded with its instruction set. The first instruction in the set is an initialization which is executed immediately the device is loaded. This could be to lock-out other communications buses, or initialize data transfers. The second instruction is the one executed during the program flow. Data is passed through the data word of the instruction set and is accessed exactly like the data word of any EPU.

Chap III 'System' Realization

This chapter will discuss an exact realization of the system as described in chapter II. The system described here will be a 12 bit data word and a maximum subset size of 16 instruction sets. This means that all addresses referring to instructions or data within the subsets will be 4 bits wide. The instruction width will also be 4 bits allowing 16 instruction types. An index register of 4 bits will also be included with each instruction set. This means the width of an instruction set will be 36 bits, therefore defining the width of the main memory bus at 36 bits.

The Loader, EPU, and Bussing system will be discussed in detail for a system as described above. The style used for describing these units will be as follows;

1: list; defining the responsibilities of the unit

2: flowchart*; to aid in following the unit's control

3: suggestions; for hardware implementation

Instruction sets and programming considerations will not be considered in this chapter. Program operation and examples of typical algorithm implementations will be dealt with in chapter IV.

* Flowcharting is basically a programming aid used to describe a sequential proceedure. It is used here because it is well known and easy to follow. It's drawback is that the units will all be using parallel logic. However the flowcharts will give an idea of the control's decision making process, and where parallel logic is involved, verbal descriptions and multiple flowcharts will be used to aid the reader follow the essence of the control's proceedures.

III-1 The Loader

The main responsibilities of the loader are as follows;

1. To obtain from main memory instructions and data for the subset assigned to its operating string.

2. To output requests on it's communications bus for the required number of EPUs.

3. To load each EPU with the appropriate instructions and data.

4. To initiate the operation of a subset.

5. To monitor and store all data transfers and their destinations during the operations of a subset.

6. To reload an EPU with the appropriate instruction set if control is passed on to an EPU not on the communication bus.

7. To dump in main memory the complete data at the termination of the algorithm, or at the request of the algorithm.

8. To reload new data upon request.

9. To allow communications with other processing strings via their loader's data bus.

The first word received by the loader from main memory is in fact an instruction to the loader. This word as discussed earlier is 36 bits wide.

1 bit is required to determine if it is an OFL subroutine call, or the following is a machine language instruction listing.

4 bits are required to determine the size of the routine, since a limit of 16 instruction sets was put on the size of a string.

16 bits are required to determine if the data is a constant or a variable address for each of the instruction sets.

These 21 bits are always defined as above for the first word of a subset algorithm stored in main memory.

The two types of subset algorithms as stored in main memory.

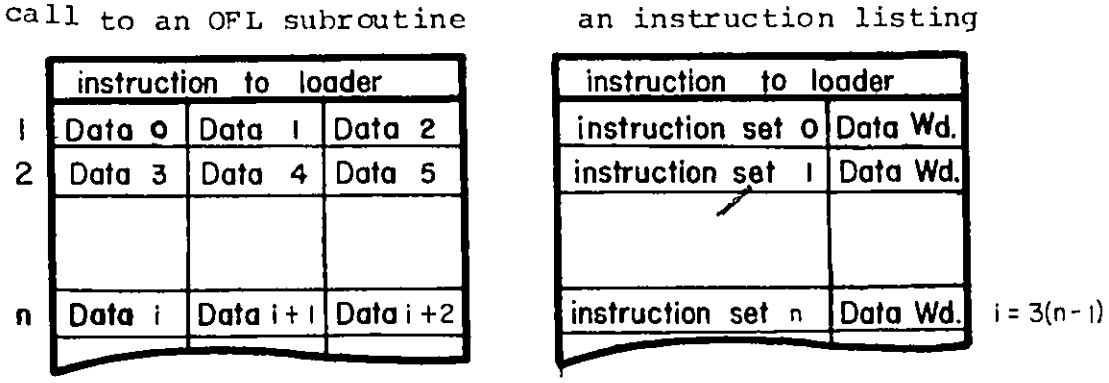


fig III - 1

Data i, or data word belonging to the instruction set i is interpreted as follows:

The ith bit of the sixteen bits in the first word instruction to the loader defines it as a twelve bit constant or a twelve bit address refering to a variable stored in main memory.

In the avent of an OFL subroutine call, the loader must assemble the instruction set. The instructions coming from the OFL memory, and the data word coming from the corresponding 'data i' in main memory. Since main memory is 36 bits wide, and the data word of an instruction set is 12 bits, three data words can be packed in a main memory word. This means some sort of multiplex scheme must be set up.

address of Data Word

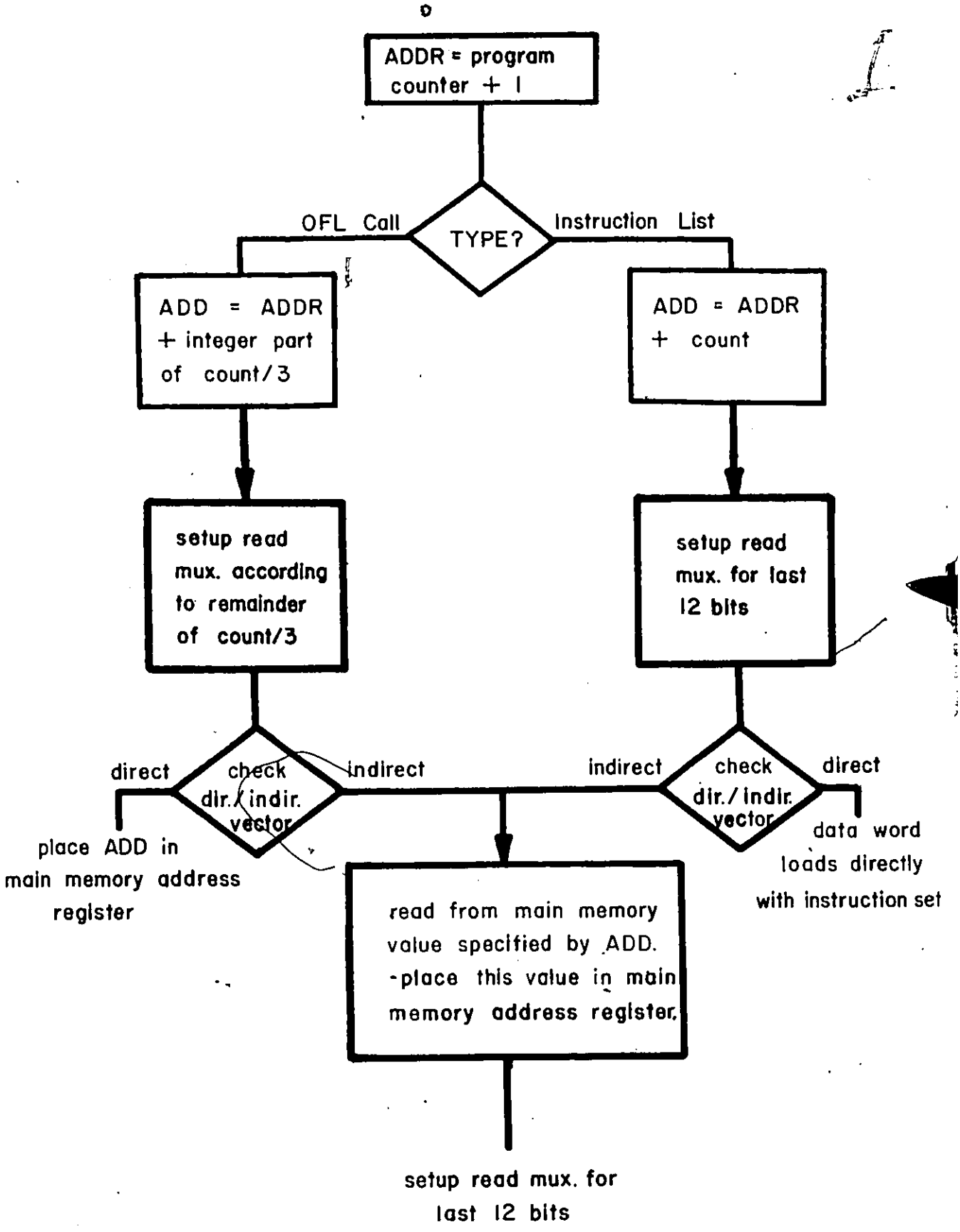


fig III - 2

Loading Routines from Main Memory

Direct 'LOAD' of Data Word

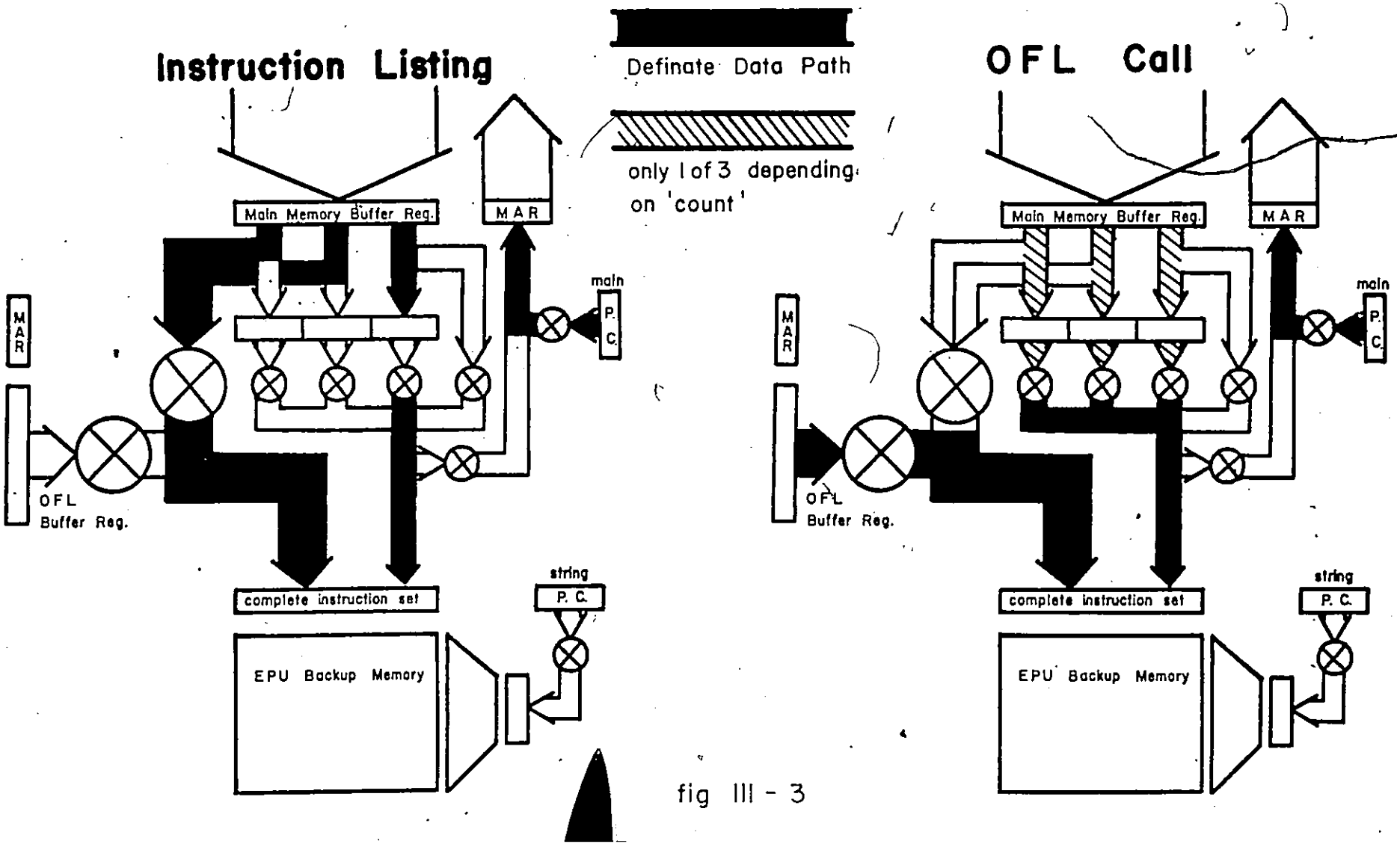


fig III - 3

Indirect 'LOAD' of Data Word Instruction Listing

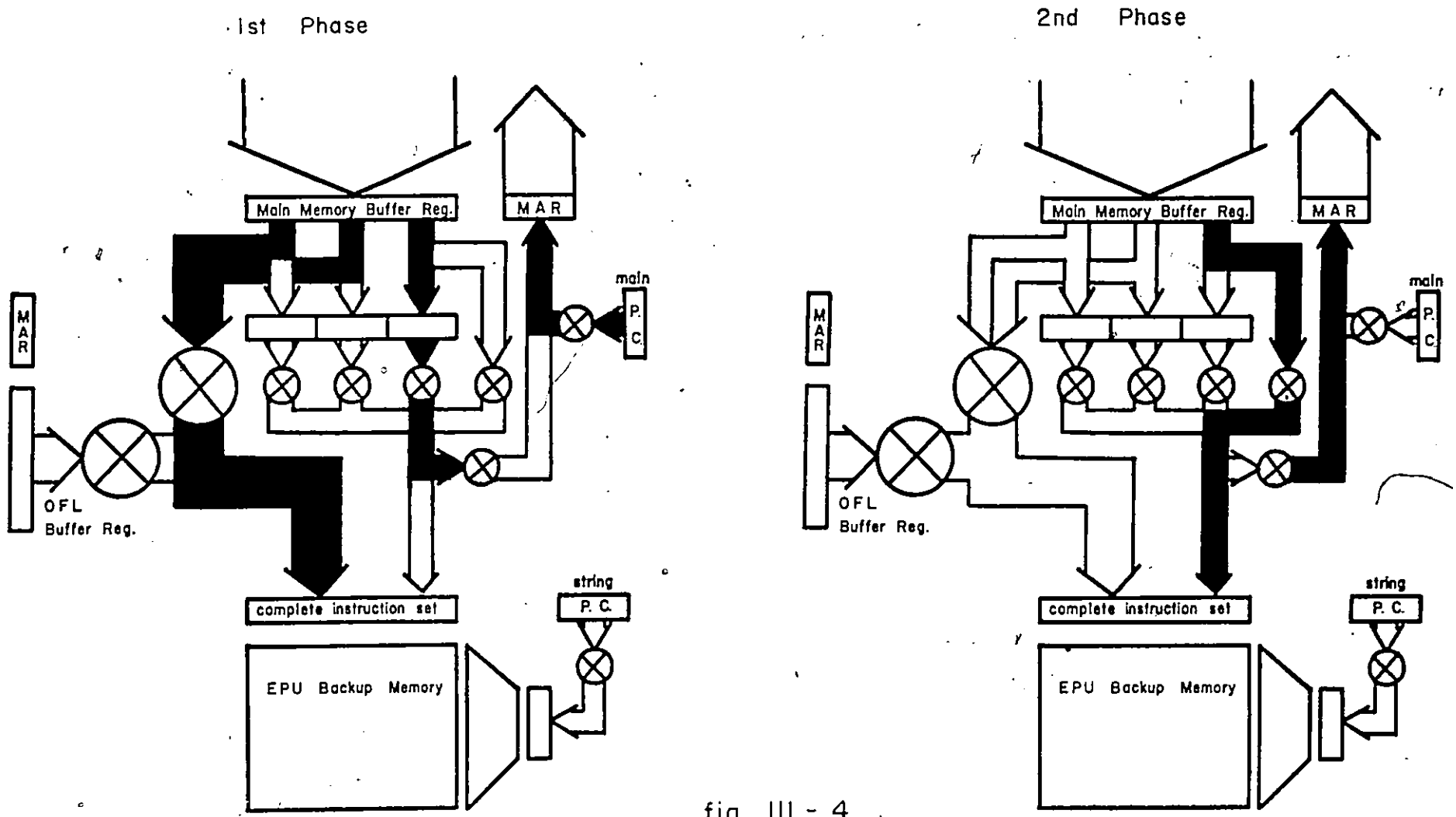


fig III - 4

Indirect 'LOAD' of Data Word
O F L Call

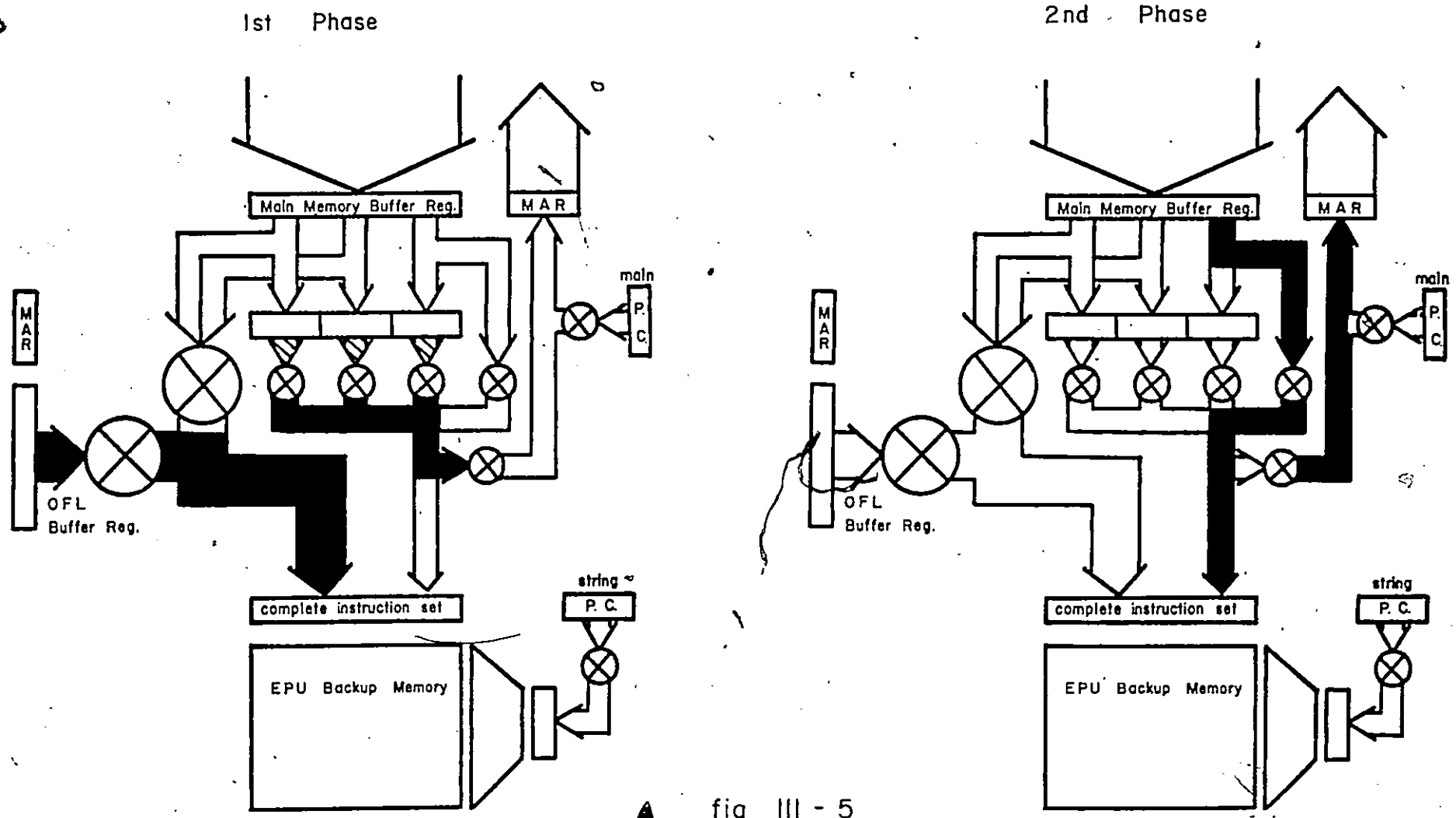


fig III - 5

If the routine is a straight instruction listing in main memory, the operation is then simpler.

This is basically a subset of the OFL previously described control routine, except the data word always comes from the last 12 bits of the main memory buffer register, and instead of the 24 bit instruction set coming from the OFL buffer register, it comes from the first 24 bits of the main memory buffer register. Indirect data is handled in exactly the same way as mentioned before.

During this period, the EPUs must also be assigned and loaded. To accomplish this, the loader connects the instruction count register onto the communication bus. It then puts out an EPU request signal.

The loader must be able to decode the first four bits of the instruction set. These four bits are the first instruction of the instruction set. If it is an Input/Output instruction, the request is put out to the peripheral pool. If not, the request is put out to the general purpose EPU pool.

At the same time the EPU backup memory's buffer register is connected via a three way gating system to the 12 bit wide data lines on the communications bus. The first 12 bits are already connected to the data lines at the time of the EPU request signal. If the request was put out to the peripheral

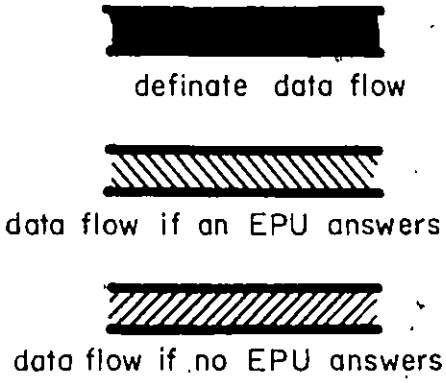
pool, these 12 bits will identify which type of peripheral device is required. Should the request be answered, the EPU reset and the remaining 24 bits of the instruction set are put out on the data lines 12 bits at a time. Therefore instruction sets are loaded into EPUs via a three phase transmission over the data lines. The complete cycle must be completed before another instruction set can be brought in from main memory or the OFL. If at the time of the request there was no reply from any EPU, this section simply signals 'cycle complete'.

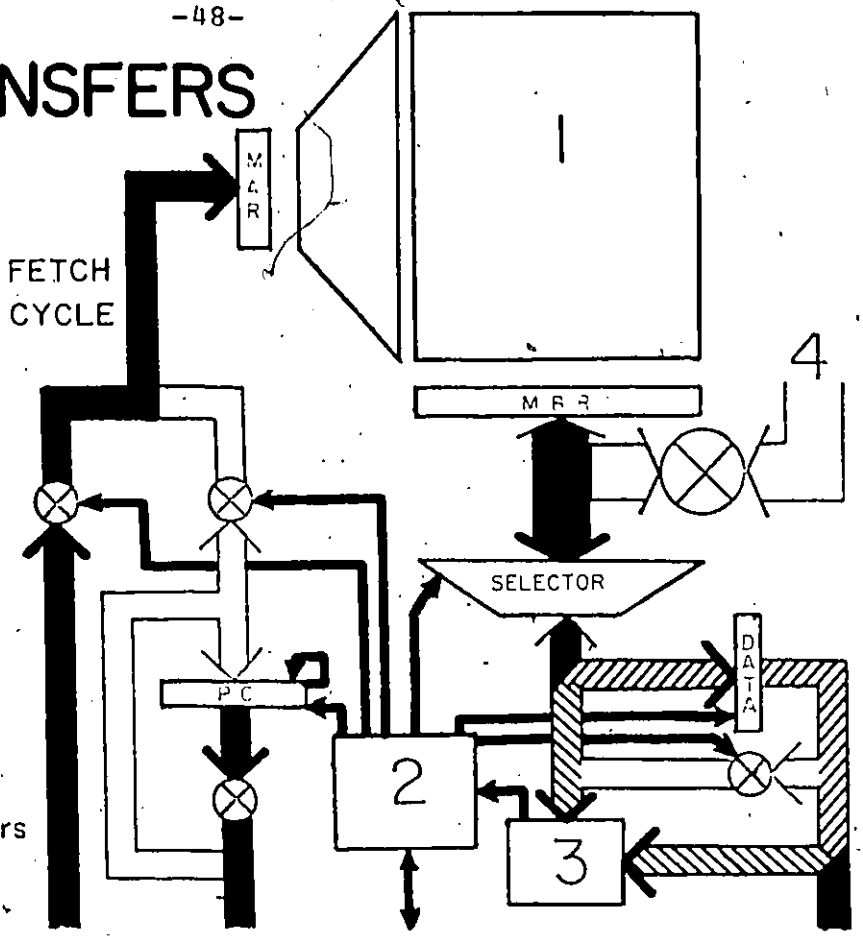
EXECUTION

Once the instruction count I is equal to the size of the routine, loading stops and execution begins. The loader's responsibilities are now to follow the execution of the routine and maintain current data and to reload EPUs in the event of failures or shortages.

Maintaining updated data is simple since all data transfers occur over the communications bus. During an instruction cycle there are basically two types of data transfers; fetch and store. Each data transfer must also be accompanied by a source or destination address depending on whether it is a fetch or store.

DATA TRANSFERS





1- EPU Backup memory
2- Control
3- Comparator
4- to main memory registers

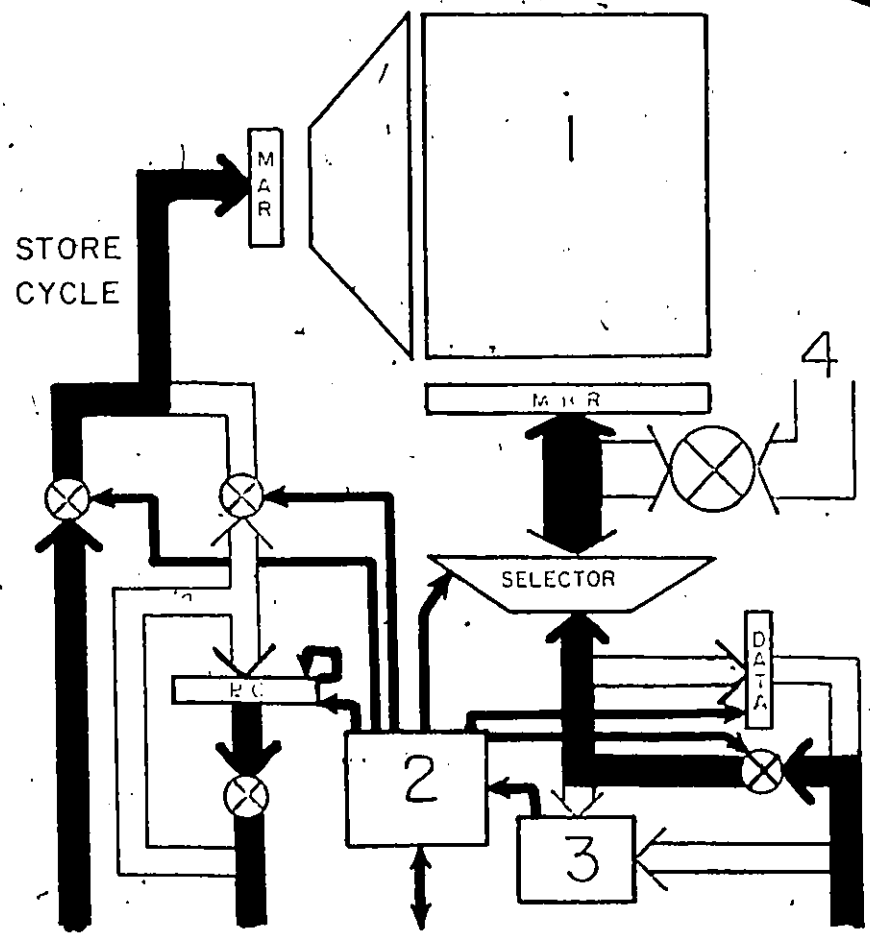


fig III - 6

During a fetch data cycle the loader uses the source address to lookup the appropriate data word in its EPU backup memory. If no EPU answers the the fetch data request the loader supplies the data word from its EPU backup memory. This case would occur when there are too few EPUs to fully distribute the routine or the EPU with the source address has failed. The other possibility is that an EPU does answer the fetch data request. The loader will now compare its data word corresponding to that source address with the data word supplied by the answering EPU. There should be an agreement, if not, one must assume a failure of the loader or communications bus and that string is shut down.

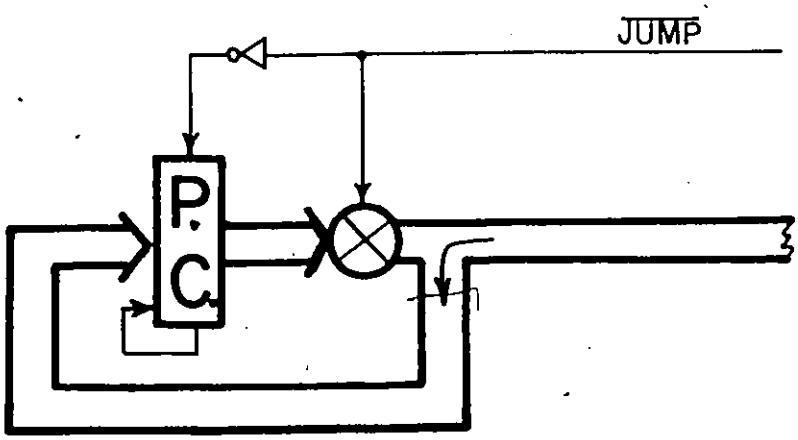
During a store data cycle, the loader uses the destination address to store the data word appearing on the communications bus. This insures that the loader's EPU backup memory always contains the latest data. It also means that the EPU with the destination address need not be functioning or even on the string at all since when the loader reloads that EPU instruction set in a new EPU, it will have the correct data word.

The program counter for the algorithm executing in the string is maintained in the loader. This program counter is a 4 bit two phase count. That is, the four bits identify which one of 16 possible EPUs is executing and the phase identifies which instruction in the instruction set is executing. The program counter is automatically incremented at the completion of an instruction unless modified by a jump instruction. The program

counter defines the operating EPU which can only be executing as long as its address appears on the program counter section of the communications bus.

The only exception is the jump instruction. When the jump line is activated, EPUs ignore the program counter lines since they will be in a transition state. After a sufficient settling time the EPU initiating the jump instruction releases the jump line, loses control since its address no longer appears on the program counter, and the new EPU assumes control.

JUMP INSTRUCTION



Jump line & Address generated by EPU executing 'JUMP' instruction.

fig III - 7

The loader must also monitor responses to the strings program counter. There must always be an EPU on-line for any count on the program counter. If there is no response from any EPU for a particular address on the program counter, the loader must find an EPU to take on the instruction set corresponding to that particular address. The program counter also has a timer assigned to it which is set for the longest instruction execution time. Should the EPU executing fail to signal job complete within this time, the loader follows the same proceedure as is no EPU had responded to the address in the program counter.

The first attempt is to find a free EPU. That is one which is not connected to any processing string at that time. This proceedure is identical to the initial loading techniques described earlier. Should there be no response to the request for a free EPU, then an EPU already connected to the operating string must be reassigned.

This reassignment can be made in several ways. For example the loader could search out the EPU with the lowest address and reassign it with the appropriate address and instruction set. This might involve a search proceedure since if there are not enough EPUs for the complete routine, the lowest address on the string need not be zero. A more direct approach both in implementation as well as execution time would be to reassign the last used EPU. This would simply require a four bit storage register to retain the old program counter address. This

would have the best chance of being on-line in an operating condition, since it was the last EPU used. If this fails a search must then be made.

Communications on Loader/Memory Bus

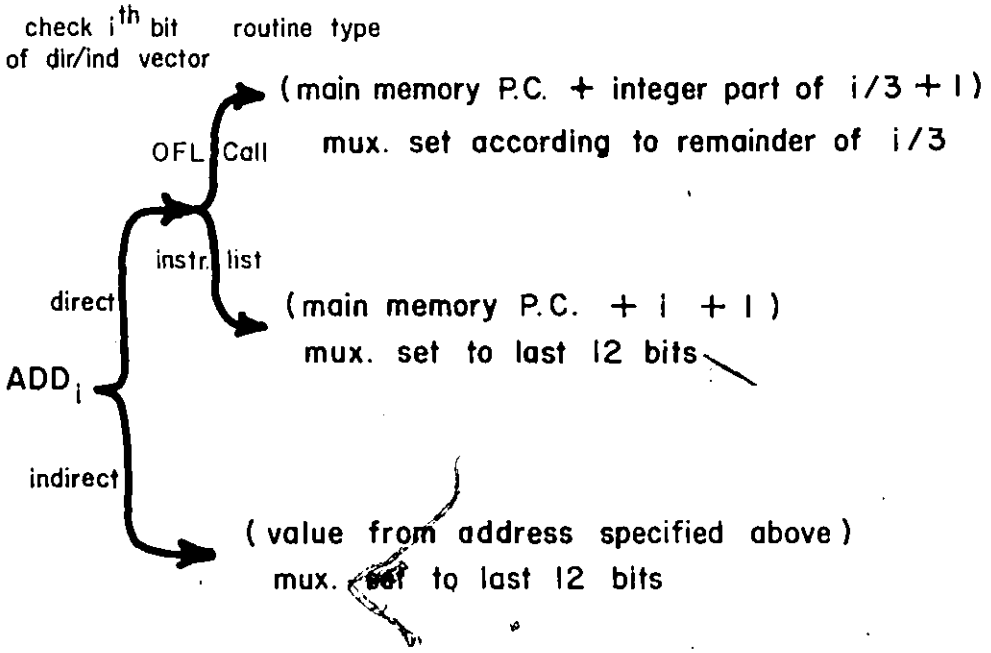
Aside from the initial loading proceedure, the loader may be called upon to make main memory data transfers by the routine operating in that string. The loader must also store all data at the completion of the routine. The storing of data can be of two forms, replacing or storing. 'Replacing', which is done at the termination of a routine, involves storing the data word in the same location where it was initially loaded from. 'Storing' simply involves supplying an address by the routine, and storing the data word at that address. Loading can only be done in one way, by supplying an address and reading the value stored there.

Replacing

This function is carried out completely by the loader and does not require EPU interaction except for the initial argument supplied by the initiating EPU over the data lines. In the case of replacing at the completion of a routine, this argument is simply the size of the routine. The argument is composed of a initial and final string address for the data words to be replaced. Since string addresses are only four bits, the

is only eight bits wide, well within the limits of the twelve bit wide data line on the communications bus.

Actual address in main memory:



Storing

The main memory actual address is supplied on the data lines. The string address of the data word to be stored there is also supplied by the initiating EPU on the four bit wide source/destination lines used for internal string data transfers. The data word is again supplied by the EPU backup memory, but as in internal data transfers, a comparison is made with the data word from the EPU of that address, should it happen to be on-line.

Loading

Again the main memory actual address is supplied by the initiating EPU. The data word is transferred to the EPU backup memory as well as the destination EPU if it is on-line, via a store cycle, as if it was the store cycle of an instruction internal to the string.

Loader Checking

Since the loader/memory bus is common to all loaders, it can be used for loader-loader communications. Data transfers between routines operating in different strings must be made through the main memory to maintain loader and program simplicity. The only inter-loader communications made are checks on operating condition. Should the loader fail line be activated, a loader just completed a routine will interigate the other loaders to find the failed unit. This loader will then pickup the old program count from the failed unit, release the fail line, and then alternate program execution between the two program counters. Should the failed loader have had a load of two program counters, the fail line remains activated. It will be up to another loader to pick up the second program counter.

A loader already operating two program counters does not check the fail line at the completion of a routine, it simply switches program counters and continues operation. Thus one loader can only be operating at most on two program streams at on time. A loader operating on two program counters does however activate the help request line. Any free loader will answer the help request line or fail line and pickup the extra program counter. A free loader is one not operating on any routine and is simply sitting idle.

III-2 EPU - Elementary Processing Unit

The Epu has two basic sections, the status section and the arithmetic-logic section.

The responsibilities of the status section are:

1. maintain operating status of the EPU
2. control gating of EPU onto appropriate communications bus
3. run diagnostics on the EPU
4. control loading of instruction sets

The responsibilities of the arithmetic-logic section are:

1. execute operations specified by its instruction set when its address appears on the program counter
2. output its data word on the data lines when its address appears during a fetch cycle
3. store the data appearing on the data lines in its data word register when its address appears during a store cycle

The main responsibility of the status section is to assure the EPU is operating properly. There are two methods of checking for error used here. The first is the error line which is a single line connected to all major bus drivers and storage registers. This line should be high at all times. Should a driver or register fail, and bring the line low, the status section puts the EPU in a fail state. The second method is a diagnostics routine run while the EPU is idle. The diagnostics are run on a table lookup basis. This table is supplied via a read only memory on board the EPU.

The diagnostics are designed to test the instruction decoder, arithmetic-logic unit and the logic comparator used in the test routine itself.

Output from the diagnostics test table:

1. 12 bits data word replacement
2. 12 bits bus register replacement
3. 12 bits desired ALU output, for comparison only
4. 4 bits instruction register replacement
5. desired instruction decoder output for comparison only
6. agree/disagree bit

The agree disagree bit is used to test the comparators, since purposely the desired outputs and actual outputs will

sometimes disagree to make sure the comparators can detect this difference. The agree/disagree bit specifies if there is an error when the comparator agrees or disagrees.

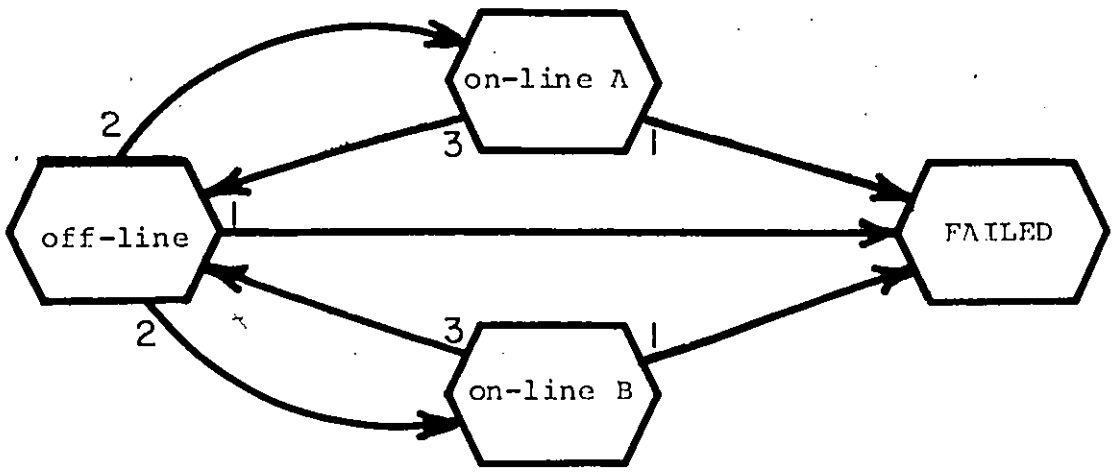
The diagnostics provides error checking while the EPU is off-line, that is not connected to any communications bus, as well as while the EPU is on-line but not busy executing, fetching or storing data. The diagnostics table is simply connected to a counter which when preforming diagnostics simply increments from on test to another. Should the EPU change to a busy state, the counter for the table stops, the replacement lines for the data word, bus register and instruction register are disconnected and the EPU is in an operating state. When the EPU returns to an idle state, the diagnostics are simply continued from where they were last stopped.

Should an error be detected, either by the error line, which could interrupt the EPU even if it were executing, or by the diagnostics routines, the EPU is completely disconnected from both communications buses. This includes the status section which is also disconnected. The loader recognizes that an EPU has failed not by a signal, but by a lack of response from that EPU when it is called upon for execution or data transfer. A fail line could be connected from each EPU to a panel display to show the operator that that particular EPU has failed. Otherwise he would not even know that an EPU had failed.

If the EPU is not in an error state, the status section must maintain the EPU's operating status. These are either 'off-line', that is, not presently connected in any processing string, or 'on-line', that is, connected to a processing string via one of the two communications buses to which the EPU has access. Thus the on-line state must be further broken down into on-line A and on-line B states in order to specify which communication bus the EPU is connected to.

Basic States of the EPU

off-line	-operational but idle
	-failed (no exit from this state)
on-line	-on-line A
	-on-line B



1- a diagnostics routine failure or the error line activated by one of the drivers or storage registers

2- request for free EPU by loader on that communications bus

3- release of EPU by loader on that communications bus

fig III - 8

EPU: 'on-line'

An EPU is brough on-line by a request for a free EPU by one of the two loaders it is connected to via their communications buses. There are actually two lines used in the loading phase of the EPU. They represent four distinct phases of the loading and are generated by the loader.

1. non loading state or operating state
2. request for free EPU and load data word and address
3. load second 12 bits of instruction set
4. load last 12 bits of instruction set

The EPU will only respond if its address is present on either the program counter lines or the destination/origin lines, except for loading state '2' (request for free EPU and load data word and self address). This request can only be answered by an off-line EPU. In answering the request, a signal is sent back to the loader shutting off the request and the communications bus carrying the request is gated onto the EPU (see fig.II-3) . The data lines are loaded into the data word register of the EPU, and the address on the program counter lines is loaded into the self address register of the EPU. The EPU is now in one of the on-line states and is ready to load the remaining 24 bits of the instruction set which are identified as belonging to that EPU by the address on the program counter lines.

The operation of the EPU is best described by the following flow chart. (see fig 9) It describes the control of the EPU. Exception to the flow chart are signals requiring immediate service. These are EPU fail, force load request, and EPU release. The EPU fail has been covered earlier. It disconnects the EPU from both communications buses.

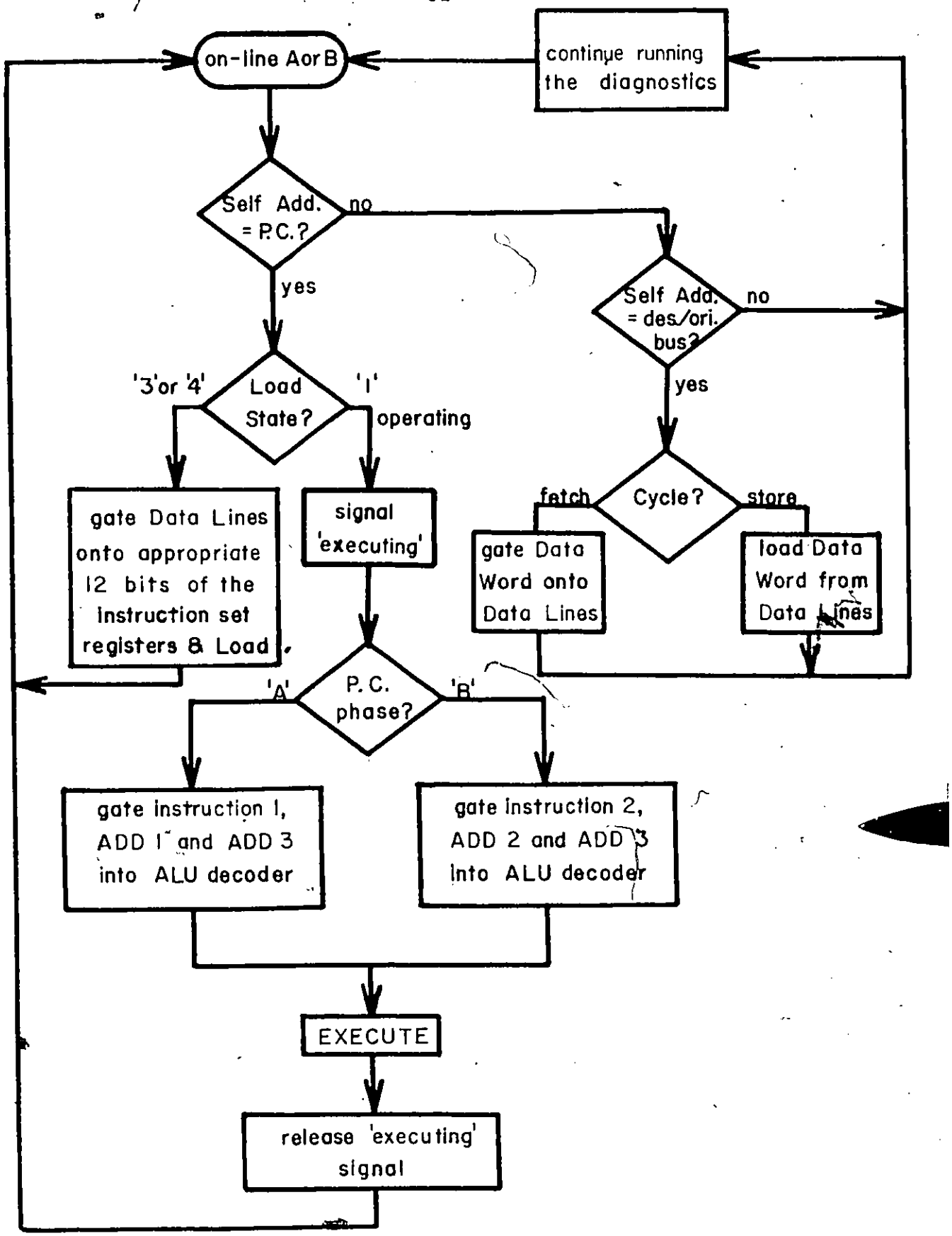

on-line AorB
continue running the diagnostics
Self Add. = P.C.?
no
yes
Self Add. = des./ori. bus?
no
yes
'3'or '4'
Load State?
'1'
operating
fetch
Cycle?
store
gate Data Lines onto appropriate 12 bits of the instruction set registers & Load
signal 'executing'
gate Data Word onto Data Lines
load Data Word from Data Lines
'A'
P. C. phase?
'B'
gate instruction 1, ADD 1 and ADD 3 into ALU decoder
gate instruction 2, ADD 2 and ADD 3 into ALU decoder
EXECUTE
release 'executing' signal

fig III - 9

The force load signal is issued by the loader when it cannot find an off-line EPU to pickup an instruction set during the execution of a routine. The force load signal releases the EPU with the same address as that appearing on the program counter. This means that that EPU is now in an off-line state and can be reloaded exactly like a free EPU.

The EPU release signal is issued by the loader at the termination of a routine, or upon failure of the loader. This signal will put in an off-line state all EPUs connected to that loader's communications bus.

Instruction Execution

The arithmetic-logic unit and decoder must be able to handle three types of instructions; no argument, one argument and two argument instructions, and could be designed using arithmetic codes.(10)

no argument instructions: These are register operations where the register is its own data word. This type of instruction does not require a fetch cycle but does have a store cycle. The destination address is the self address of that instruction set. The reason for the store cycle is simply to maintain data integrety within the loader's EPU backup memory.

DATA TRANSFERS

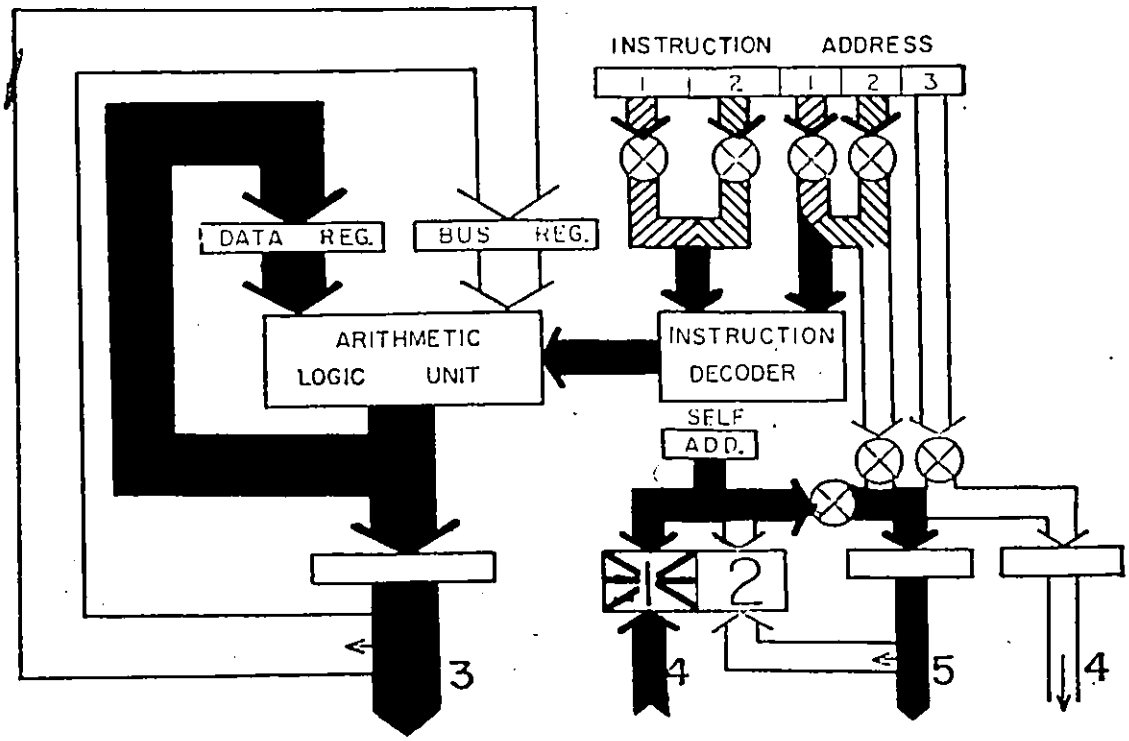
1 P.C. Comparator
2 Destination / Origin Comparator
3 Data Lines (C.B.)
4 Program Counter (C.B.)
5 Des. / Origin Address (C.B.)
C.B. - Communications Bus

Definite Data Route

P.C. is Phase A

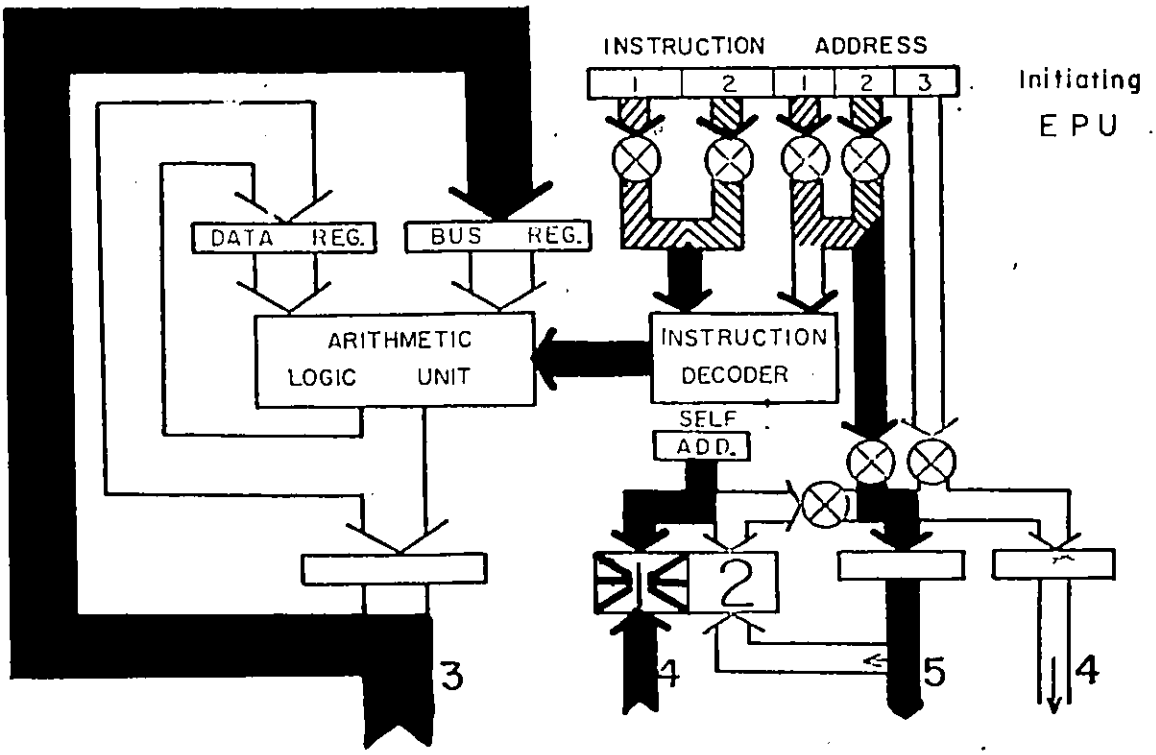
P.C. is Phase B

2 Argument Instr.



REGISTER OPERATION

fig III - 10



FETCH CYCLE

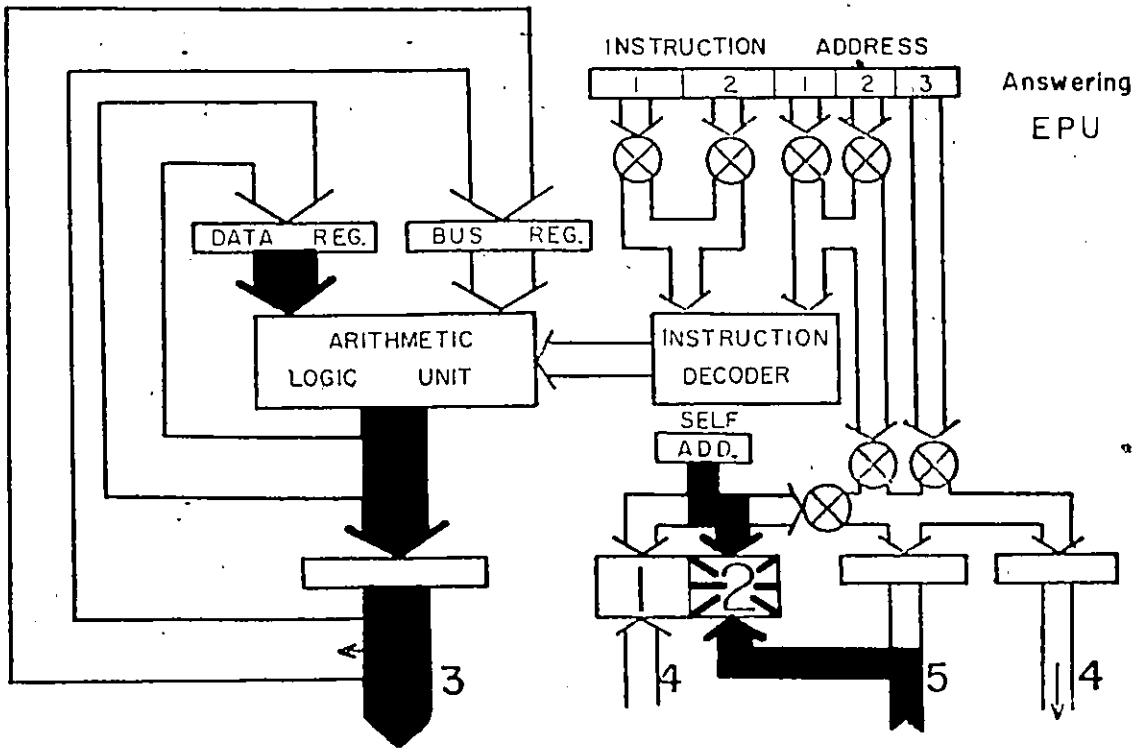
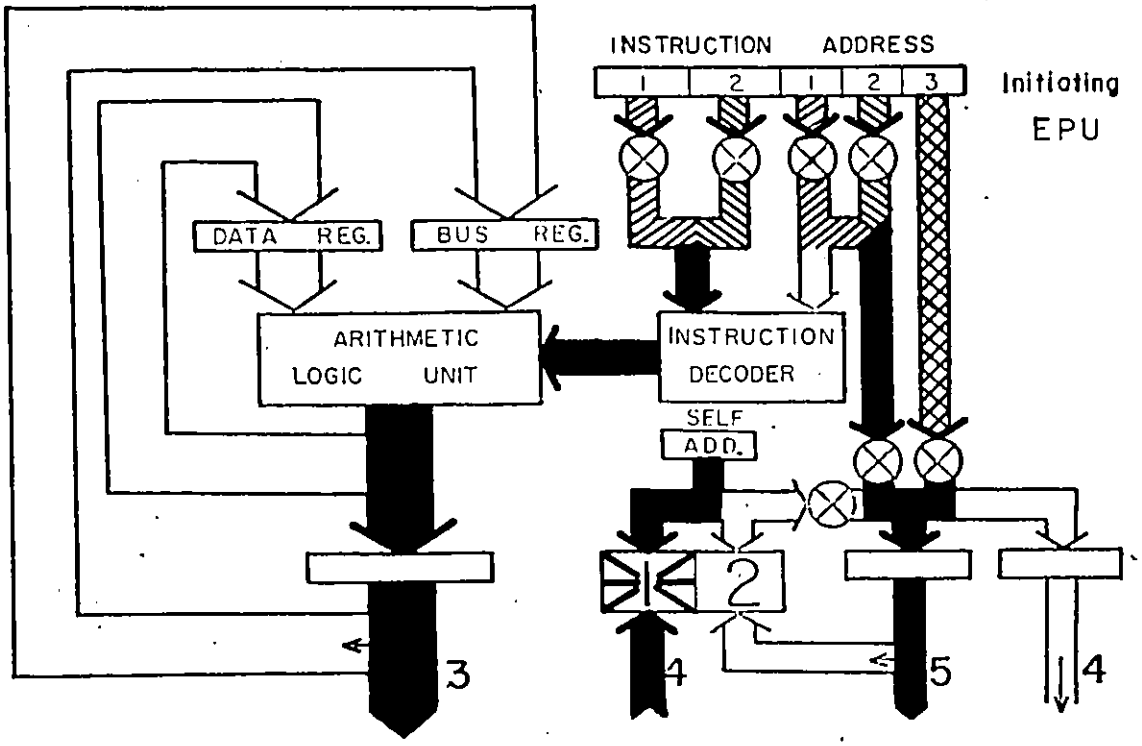


fig III - 11



STORE CYCLE

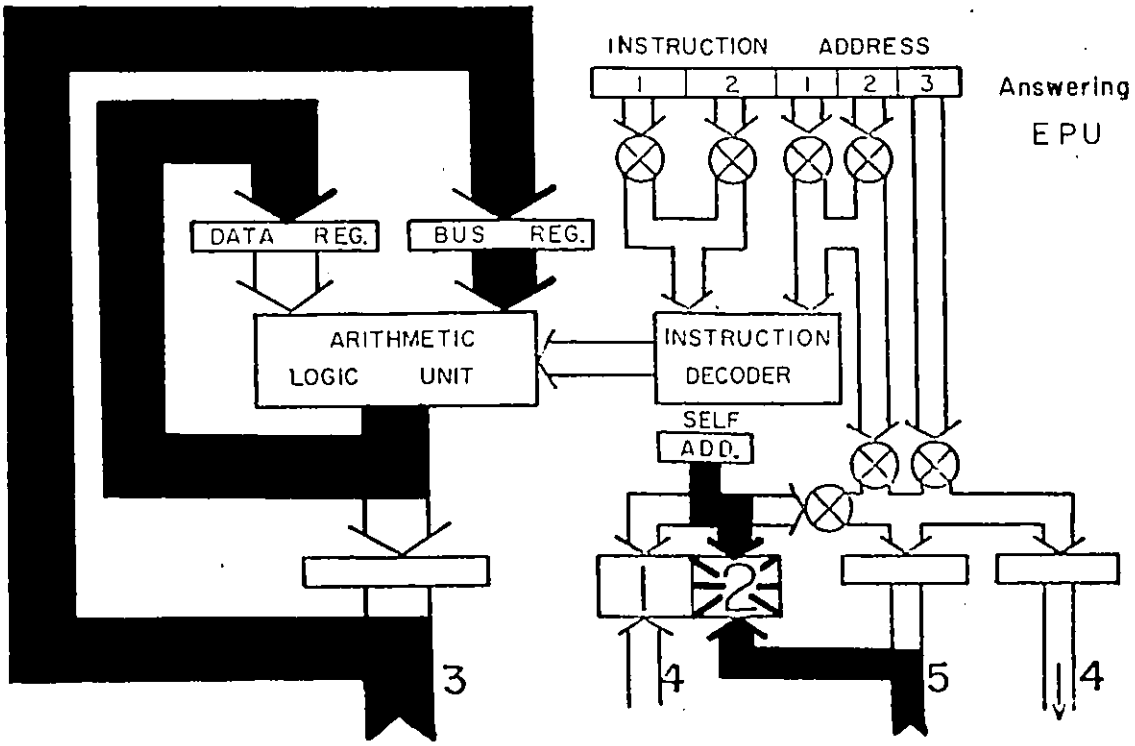


fig III -12

single argument instructions: These are instructions requiring either a fetch data or store data cycle from or to an address other than the self address of the instruction set. Instructions only requiring a fetch cycle also have a store cycle with the destination address being the same as the self address of the instruction set. Again this is for the purpose of maintaining data integrity within the loader's EPU backup memory.

double argument instructions: These are instructions requiring both a fetch data and store data cycle with addresses other than its own self address. The other possibility is a jump instruction where one argument is the destination address of the jump, and the second argument is the condition on which the jump occurs. A jump instruction does not require a store or fetch cycle at all.

III-3 The Communications Bus

The communications bus is a wide bi-directional bus. There is one communications bus associated with each loader. EPUs may be attached to or removed from the communications bus without any modifications to the bus.

The responsibilities of the communications bus are;

1. address lines for program counter and all string data transfers
2. data lines for all data transfers within the string
3. status and control lines for loading and executing routines in the string

types of transfers

EPU to EPU

EPU to Loader

Loader to EPU

Where the EPU can be a general purpose EPU or a specialized EPU dedicated to a peripheral device.

Signals on the communications bus are activated by being pulled to ground. This means that several devices can drive the same line by open collector drives with the line acting as an

inclusive or for ground levels from all drivers connected to the line.

1 address lines

program counter:

-four lines address plus one line for phase

-direction: loader to EPU

EPU to loader (during jump instruction)

destination/origin address:

-four lines address

-direction: EPU to EPU

EPU to loader (maintaining data in EPU backup memory)

2 data lines

data word:

-twelve lines

-direction: EPU to EPU

EPU to loader

loader to EPU

3 status and control lines

loading control;

-four lines

-direction: loader to EPU

L1	L2	L3	L4	
0	0	0	0	-non loading or operating state
0	0	0	1	-request for free EPU
0	0	1	1	-load 2nd 12 bits of instruction set on address match
0	0	1	0	-load 3rd 12 bits of instruction set on address match
0	1	0	0	-request for EPU from peripheral pool
0	1	1	0	-load 2nd 12 bits of instruction set on address match
1	x	1	x	-release EPU on address match
1	x	0	x	-release all EPUs on communications bus

x-don't care

Lines L2 and L4 are the only non passive lines in the communications bus. L4 connects only to the general purpose EPUs and L2 connects only to the EPUs of the peripheral pool.

These lines are in the form of shift registers:

The purpose is to provide a delay between each EPU on the line to prevent simultaneous answering.

operating control and status:

-1 line	EPU to loader	EPU executing
-1 line	EPU to loader	address match on fetch
-1 line	EPU to loader & EPUs	jump
-1 line	EPU to EPU & loader	fetch/store
-1 line	Overflow bit, stored in loader	
-2 lines	EPU to loader	replace, store and load commands

-miscellaneous control lines

timing:

-1 line	EPU to EPU & loader or loader to EPU strobe for data transfers
-clock	loader to EPUs

Chapter IV Programming the System

This chapter will not specify an exact instruction set for the EPUs, but will suggest classes of instructions which will supply adequate programming capability. The actual instruction set is a function of the type of ALU one wishes to use in the EPU.

After the instruction classes have been discussed, small examples will be used to demonstrate the operation of routines in a string. These examples will be extended to a small program to show how they would appear in main memory.

Finally examples will be given of parallel program flows, and the use of strings dedicated to handling one commonly used subroutine.

IV - 1 Instruction Classes

Each instruction in the instruction set has four bits not counting the addresses or modifiers. This limits us to 16 instructions types.

type	description
0	I/O instruction.
1	self register operations, addresses are modifiers.
2	conditional jump, 2 arguments. one address is the destination, the second is the condition for the jump.
3	replacing in main memory.
4	storing in main meomry.
5	loading from main memory.

This leaves nine instruction types to be used for arithmetic and logic operations.

examples of type 1, self register operation, SRO(A)

type 1: modifier

SRO(A)	0	NOP	no operation (index specifies delay)
	1	EXIT	exit from string
	2	CLR	set register to 0
	3	OCR	one's complement register
	4	TCR	two's complement register
	5	INC	increment register and skip on zero

6	DCR	decrement register and skip on zero
7	COF	clear overflow bit
8	SOF	set overflow bit
9	LLS	logical left shift.(index specifies repetition)
10	LRS	logical right shift (index specifies repetition)
11	ALS	arithmetic left shift (index specifies repetition)
12	ARS	arithmetic right shift (index specifies repetition)
13	ROT	rotate (index specifies repetition)
14	RWO	rotate with overflow (index specifies repetition)
15	RST	reset to zero register and overflow bit.

note - index register is used here to specify the number of repetitions of an instruction for certain instruction types.

- on shifts, the overflow bit specifies what is shifted in, zeros or ones.

replacing storing and loading: are all main memory instructions. They were discussed in detail in the loader discussion. (page 52-54

So far the instructions discussed used only one address, and possibly the index register. There are also instructions using two addresses.

jump instruction, type 2 instruction

-address B is destination of the jump

-address A is a modifier, and specifies the condition for the jump. There are 16 possible modifiers for the jump.

JMP

AUNC -jump unconditional to PC phase A

AZER -jump on zero data work to PC phase A

ANZR -jump on non-zero data work to PC phase A

AOFL -jump on overflow bit = 1 to PC phase A

AZOF -jump on overflow bit = 0 to PC phase A

APOS -jump on positive data work to PC phase A

ANEG -jump on negative data word to PC phase A

AZR0 -jump on zero data word and overflow to PC phase A

BUNC -jump unconditional to PC phase B

BZER -jump on zero data work to PC phase B

BNZR -jump on non-zero data work to PC phase B

BOFL -jump on overflow bit = 1 to PC phase B

BZOF -jump on overflow bit = 0 to PC phase B

BPOS -jump on positive data work to PC phase B

BNEG -jump on negative data word to PC phase B

BZR0 -jump on zero data work and overflow to PC phase B

skip instruction, tvpe 6 instruction

-single address instruction. The A address specifies the condition for skip as above. If skip condition occurs, the program counter is incremented, and the phase is set as stated above.

logical operations, two addresses

type					
7	B	=	A	AND	self address data word
8	B	=	A	OR	self address data word
9	B	=	A	XOR	self address data word
10	B	=	OPR(A)		register operation on other than self data word

arithmetic operations, two addresses

type					
11	B	=	A	ADD	self address data word
12	B	=	A	SUB	self address data word
13	B	=	A	MULT	self address data word
14	B	=	A	DIV	self address data word

data transfer operation, two addresses.

type				
15	A	=	B,	a MOVE operation

The arithmetic operations can be done with twos complement arithmetic.[11] This facilitates maintaining correct signs. The multiply and divide operations need not be a complete multiply or divide operation within itself. It can be simply a combination of shift and conditional add or subtract with number of repetitions determined by the index register.

Program jumps and subroutine jumps can be made in main memory by accessing the main memory program counter through the load and store commands. A specific address, such as address zero, could be used to specify the program counter. It would be up to the calling routine to supply the return address for the subroutine jumps.

Since two argument instructions have definite limitations when considering two instructions per instruction set, we could have duplicated some two argument instructions as single argument instructions.

However, since in two argument instructions the second address is always the destination, in many cases second arguments will be the same, since it will be the address of the data word acting as the accumulator for the routine.

IV - 2 Routine examples

example 1; table look up

- a mask word is used, to specify which bits must match
- the key word is supplied which must match in the table
- table is in main memory, pointer to bottom address is given
- a new table is produced of only matching words
- a zero in table defines end of table
- on exit, the number of matches is given

Coding Form

Program name example 1 Routine name table lookup Size 8

S.A.	instr. 1	instr. 2	index	A·1	A·2	B·1 or 2	(Before) Data Word (After)		D
00	OPR	LOAD		CLR	01	04	ptr1	end1	
01	JMP	XOR		AZER	07	06	-	table	
02	AND	SRO		06	NOP	03	mask	mask	
03	OPR	JMP		INC	BNZR	00	-	scratch	
04	SRO	OPR		INC	INC	05	-	match #	
05	STORE	JMP		01	BUNC	00	ptr2	end2	
06	SRO			EXIT			-	scratch	
07							key	key	
08									
09									
10									
11									
12									
13									
14									
15									

S.A. Self Address
D Direct / Indirect Vector
instr. 1 works on A·1 & B·1
instr. 2 works on A·2 & B·2

example 2; find minimum and maximum table values

- table is in main memory, pointer to bottom address is given
- on exit maximum and minimum values from table are given

Coding Form

Program name example 2 Routine name max./min. Size 7

S.A.	instr. 1	instr. 2	index	A·1	A·2	B·1 or 2	(Before) Data Word (After)		D
00	LOAD	SRO		06	INC	-	pointer	end	
01	SUB	SRO		06	NOP	02	0	maximum	
02	SKIP	MOVE		ANEG	01	06	-	scratch	
03	SUB	SRO		06	NOP	04	max val	minimum	
04	SKIP	MOVE		APOS	03	06	-	scratch	
05	SRO	JMP		DCX	AUNC	00	size	0	
06	SRO			EXIT			-	table	
07									
08									
09									
10									
11									
12									
13									
14									
15									

S.A. Self Address
D Direct / Indirect Vector
instr. 1 works on A·1 & B·1
instr. 2 works on A·2 & B·2

example 3; modulo 2 multiply of two polynomials

-the two polynomials (degree less than or equal to 10 are loaded with the routine

-the product must have degree less than or equal to 11

-the product is returned at the end of the routine R=P1*P2

-number of repetitions of program loop is equal to degree of P1

example 4; modulo 2 multiply double precision

-Same features as example 3 except product can have degree 23

Coding Form

Program name example 3 Routine name modulo 2 mult. Size 5

S.A.	instr. 1	instr. 2	index	A·1	A·2	B·1 or 2	(Before) Data Word (After)		D
00	SRO	OPR	1	COF	LRS	01	0	result	
01	JMP	SKIP		BZRO	AZOF	04	P1	0	
02	XOR	SRO		00	COF	00	P2		
03	OPR	SRO	1	LLS	NOP	02			
04	JMP	OPR		AUNC	EXIT	00			
05									
06									
07									
08									
09									
10									
11									
12									
13									
14									
15									

S.A. Self Address
D Direct / Indirect Vector
instr. 1 works on A·1 & B·1
instr. 2 works on A·2 & B·2

Coding Form

Program name example 4 Routine name modulo 2 mult. Size 8

S.A.	instr. 1	instr. 2	index	A·1	A·2	B·for2	(Before) Data Word (After)		D
00	SRO	OPR	1	COF	LRS	01	0	R msb	
01	JMP	JMP		BZRO	AZOF	06	P1	0	
02	SRO	XOR		COF	04	02	0	R lsb	
03	XOR	SRO	1	00	LLS	00	0	P2	
04	SRO	JMP	1	LLS	AZOF	00	P2	0	
05	OPR	SKIP		INC	AUNC	03			
06	JMP	SRO		BUNC	EXIT	03			
07	JMP			AUNC		00			
08									
09									
10									
11									
12									
13									
14									
15									

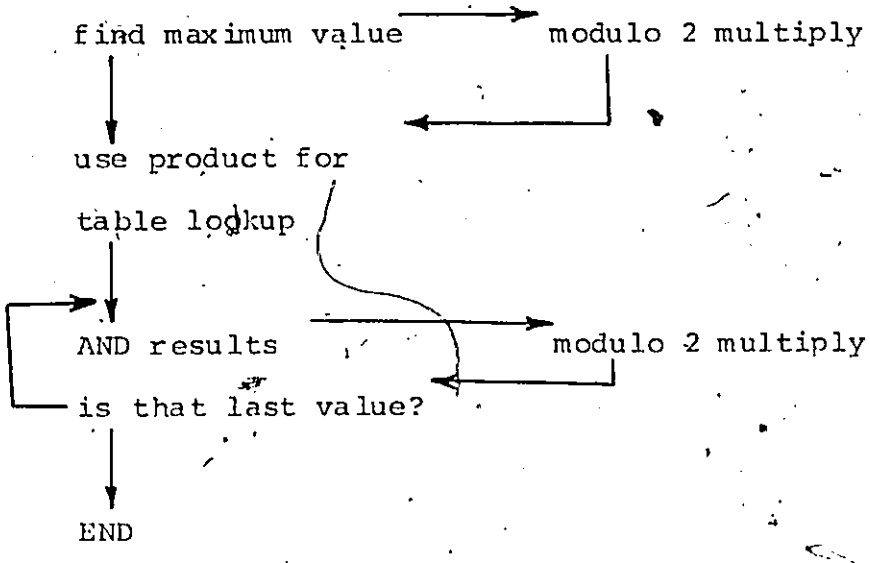
S.A. Self Address
D Direct / Indirect Vector
instr. 1 works on A·1 & B·1
instr. 2 works on A·2 & B·2

IV - 3 Program example

The program example will make use of the routines described in the previous section. The problem will be to perform a modulo 2 multiply of two polynomials, use the result as a key to perform a table lookup. Meanwhile find maximum value from the table and perform a modulo 2 multiply of the maximum value with all the matching values from the table and AND the results.

The modulo 2 routine will be setup in a dedicated string which can be called as a subroutine. An excellent way of initiating a call to such a subroutine would be through a specialized peripheral device which could be connected to several strings at a time. The subroutine call would then appear as a call to a peripheral device with that particular function. This example however, will only use features previously described and therefore calls to the subroutine will be made through main memory.

program flow:



The subroutine call is made via a flag word. The calling routine sets the variables, then turns on the busy flag. The subroutine continuously monitors this flag. A calling program must assure this flag is zero before initiating the call. When the subroutine has finished, it outputs the results then compliments the flag word. The calling routine, when ready, can pickup the results after the flag has been complimented, then sets the flag back to zero.

The maximum minimum and table lookup routines could be combined into one routine. The maximum minimum section being executed first, with a conditional statement placed inbetween such that the routine only continues onto the table lookup if the flag is set to all ones (has been complimented). However if these

Coding Form

Program name IV - 3 Routine name subroutine Size 15

S.A.	instr. 1	instr. 2	index	A·1	A·2	B·1 or 2	(Before) Data Word (After)		D
00	LQAD	OPR		09	DCX	09	Flag add.		
01	JMP	LOAD		AUNC	04	00	P1 add.		
02	LOAD	OPR		07	OCR	09	P2 add.		
03	SRO	OPR	1	COF	LRS	04	0	R msb	
04	JMP	JMP		BZR0	AZOF	09	P1		
05	SRO	XOR		COF	07	05	0	R lsb	
06	XOR	SRO	1	03	LLS	03	0	P2	
07	SRO	JMP	1	LLS	AZOF	03	P2		
08	OPR	SKIP		INC	AUNC	06			
09	JMP	SKIP		BUNC	BUNC	06	Flag		
10	JMP	STORE		AUNC	03	03	Rm add.		
11	STORE	OPR		05	CLR	03	R1 add.		
12	OPR	STORE		CLR	09	05	Flag add.		
13	OPR	SRO		CLR	NOP	06			
14	JMP			AUNC		00			
15									

S.A. Self Address
D Direct / Indirect Vector
instr. 1 works on A·1 & B·1
instr. 2 works on A·2 & B·2

two routines were pre-packaged routines in the Operating Functions Library, the program would look like this:

```
instruction listing          setup and call subroutine

call max/min routine         (OFL)
 pointer ,    0    ,    -
max value,    -    ,   size

instruction listing          loop until subroutine has
                             finished, then continue
                             setting flag = 0

call table lookup routine (OFL)
  ptr 1  ,    -    ,   mask
    -    ,    -    ,  ptr 2
    -    ,   key   ,    -

instruction listing          loop through condensed table
                             calling multiply subroutine
                             and ANDing results
```

Coding Form

Program name IV - 3 Routine name loop through table mult and AND Size 11

S.A.	instr. 1	instr. 2	index	A·1	A·2	B·1 or 2	(Before) Data Word (After)		D
00	SRO	LOAD		NOP	01		Flag add.		
01	JMP	SRO		BNZR	INC	00	0	Flag	
02	LOAD	SRO		10	INC		pointer		
03	STORE	SRO		10	-NOP		P1 add.		
04	STORE	LOAD		01	05		Flag add.		
05	SRO	JMP		INC	BUNC	04	0	Flag	
06	LOAD	SRO		10	NOP		Rm add.		
07	AND	SRO		10	NOP	07	0	Result	
08	SRO	JMP		DCR	AUNC	02	size	0	
09	OPR	STORE		CLR	01	01	Flag add.		
10	SRO			EXIT			scrotch		
11									
12									
13									
14									
15									

S.A. Self Address
D Direct / Indirect Vector
instr. 1 works on A·1 & B·1
instr. 2 works on A·2 & B·2

Chapter V - Conclusions and Future Topics

V-I Summary

The computer architecture suggested in this thesis differs from a standard computer architecture in that it operates on routines rather than single instructions. In a standard computer architecture the CPU loads an instruction from main memory, decodes the instruction sets up addressing for any data transfers required by the instruction. Then the CPU executes the instruction, completes any data transfers specified by the instruction, updates the program counter and repeats the cycle.

In the architecture described by this thesis, the CPU loads a 'macro-instruction' and data from main memory. In an ideal case, all data required is loaded at this time, however, data can be transferred to or from memory during the execution of the 'macro-instruction'. The macro-instruction can be a standard routine, that is, one from the Operating Functions Library, or it can be a variable length instruction set listing. A listing of this type is in effect a macro-instruction which is user defined. At the termination of this macro-instruction data is returned to main memory the program counter is updated, and the cycle is repeated.

The CPU (Central Processing Unit) differs greatly from that of a standard computer architecture. It was referred to in this thesis as a string. It is this string's operational configuration which is variable structured. The macro-instruction is composed of a listing of instruction sets which form a program executing the function of the macro-instruction. When the string loads a macro-instruction from main memory, it distributes these instruction sets to EPUs (Elementary Processing Units).

Each EPU consists of a data register referred to as a data word. This data word also has an instruction set register, instruction decoder and simple arithmetic logic unit associated with it. The macro-instruction is executed within the string by passing on control to the EPU with the appropriate instruction set in it.

The reason for completely distributing the routine is as follows. Program flow is interleaved through the EPUs. This means each EPU is only busy for the time it is executing its instruction set, which is a small part of the routine, or macro-instruction. This means that it has a lot of idle time in which to perform extensive diagnostics. This means diagnostics are run continuously along with the program flow.

Most algorithms or routines have parts which are repeated, that is, decision making and branching which form

program loops. When a macro-instruction has a program loop in it, the advantages of a distributed routine becomes much more evident. There are much fewer information transfers, the fetch instruction cycles are virtually eliminated. This means an increase in speed and less chance of missed bits due to information transfers since there are fewer transfers.

Interleaving of program flow also means a speed up of program execution. This means the logic used in the EPUs can be simpler than that used in a standard CPU to execute within the same time restraints. Simpler logic used in the EPU means bipolar technology such as ttl or ecl can be used and still give small packaging. The advantage of bipolar circuits is it's high speed, however it has a relatively low density of function per package. There are ALUs and even micro-computers on single chips available today using this circuit technique. On the other hand more complex circuits can be used with slower circuit technologies such as MOS. MOS has the advantage of allowing high density of logical functions per package.[12] There are large number of MOS ALUs and micro-computers on single chips available on the market today. In fact the relatively new COS/MOS, which is faster than MOS and very noise resistent,[13] has ALUs on a chip available.

It is within today's technology to produce the loader and the EPU on single chips. In fact with large ceramic package techniques, several chips can be put in one package. This reduces mechanical stresses, environment restraints, socket

contacts and cold solder joints which plague attempts at reliability where many technologies are used. That is combining integrated circuits, discrete components, printed circuits, etc.

Ideally one loader and eight EPUs should be placed on a single package for the realization described in chapter III. This means each package would have one loader/main memory bus and two communication bus ports. One communication bus belongs to the loader on board the chip, the second communication bus is to connect to another package's loader in order to allow sharing of EPUs. Each EPU on the package is connected to both communication buses.

V-II Features

The CPU of this architure has complete redundancy of executing components. The system features graceful degredation in the computing units or strings. As more parts fail, the execution times for the macro-instructions increases. In the worst possible case one loader and one EPU can maintain operation. In this case the architecture resembles a typical computer architecture with the loader acting as a cache memory.

Since the memory has no immediate backup, it is the weak link. However under ideal operating conditions it is seldom used, and if the strings are operating with self contained algorithms, it is never used.

In a system configuration with several loaders, complete strings including loader can be replaced by free loaders, if available with no increase in executing time or by a loader alternating between two program streams which means an increase in execution times. This varying of execution speed depending on operating condition means that responses requiring exact time delays must be strobed against a real time clock. This is because predetermined repetitions of a program loop does not guarantee an exact time delay.

In normal operating condition, the distributed nature means once a string is loaded, there are fewer information transfers, both instruction and data. Therefore there are fewer steps in executing the instruction and thus less chance of missed bits. Each loader being identical as well as each EPU being identical means less design problems from 'one of a kind' components, as well as lower assembly and debugging costs.

The modular design of the system allows for expansion from mini-computer size to large computer system. Allowing instruction set listings to be stored in main memory means flexibility in designing and clecking routines, or macro-instructions, before they are finalized in the Operating Functions Library. The design also allows EPUs to be added or removed from the system during operation, without disturbing program flow, thus allowing for 'on the spot' maintainence without shutting down the system. In addition to this, the system can also tolerate several types of failures in its intercommunications network without loss of integrety. Thus the system using simple components and structures can realize difficult control problems and maintain a very high degree of reliability.

V-III Future Topics

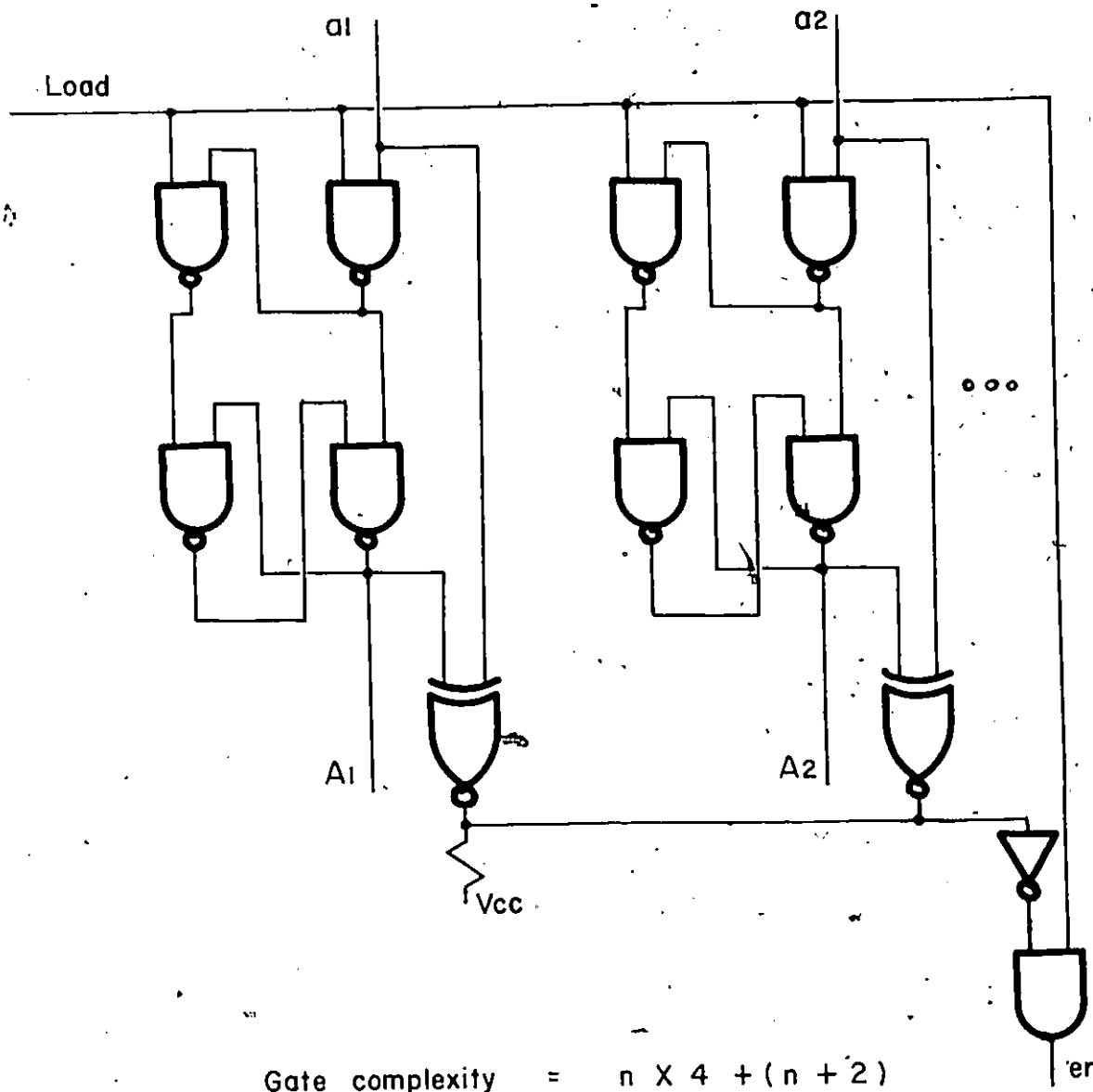
Many possible features of this type of structure were not discussed in this thesis, such as backups for the main memory, direct string communications bypassing main memory, and multiple data word length instruction. This latter concept is possible since it would only mean slaving EPUs together. That is, a data word length operation of three times normal data word length would require two EPUs to be slaved to the executing EPU. Main memory's integrity can be improved by using redundant memories in a majority vote scheme. Direct string communications can be achieved at the expense of slowing down the bus by multiplexing or widening the memory/loader bus.

There are several areas for expanding the concept as well as optimizing hardware design and basic instruction set for improvement in speed and ease of programming. One can also set up guidelines for trade-offs in circuit complexity and optimum instruction type. This can be done by setting up a simulation of the system, and obtaining some programming experience using different instruction sets before finalizing the hardware design. This allows the designer to appreciate problems of the programmer and attempt to ease some of them.

Perhaps the most exciting area for development is the concept of the macro-instructions. Operating Functions Libraries

could be designed to allow for straight forward operation of higher level languages. For example, a system which has directly executable fortan instructions. Thus reducing software overhead to simple assemblers rather than complex complilers, in order to use higher level languages.

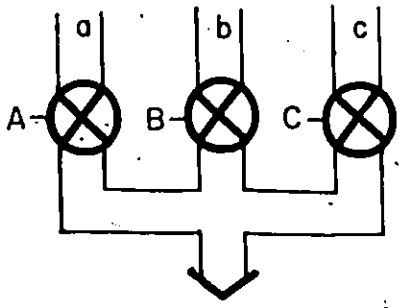
Storage Register

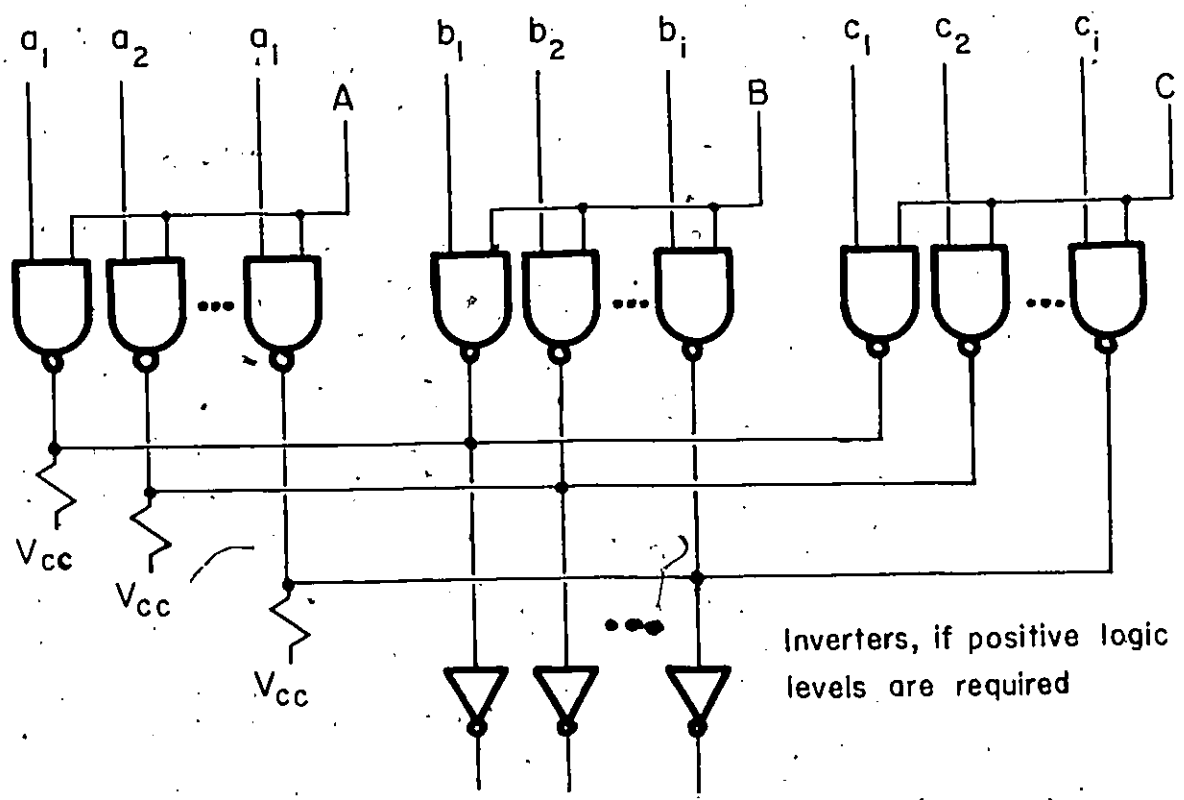


Gate complexity = n X 4 + (n + 2)

Wire complexity = n X 4 X 2 + (n X 2 + 2 + 1)

n = width of register ( ) = error detect circuit ~ 20 %





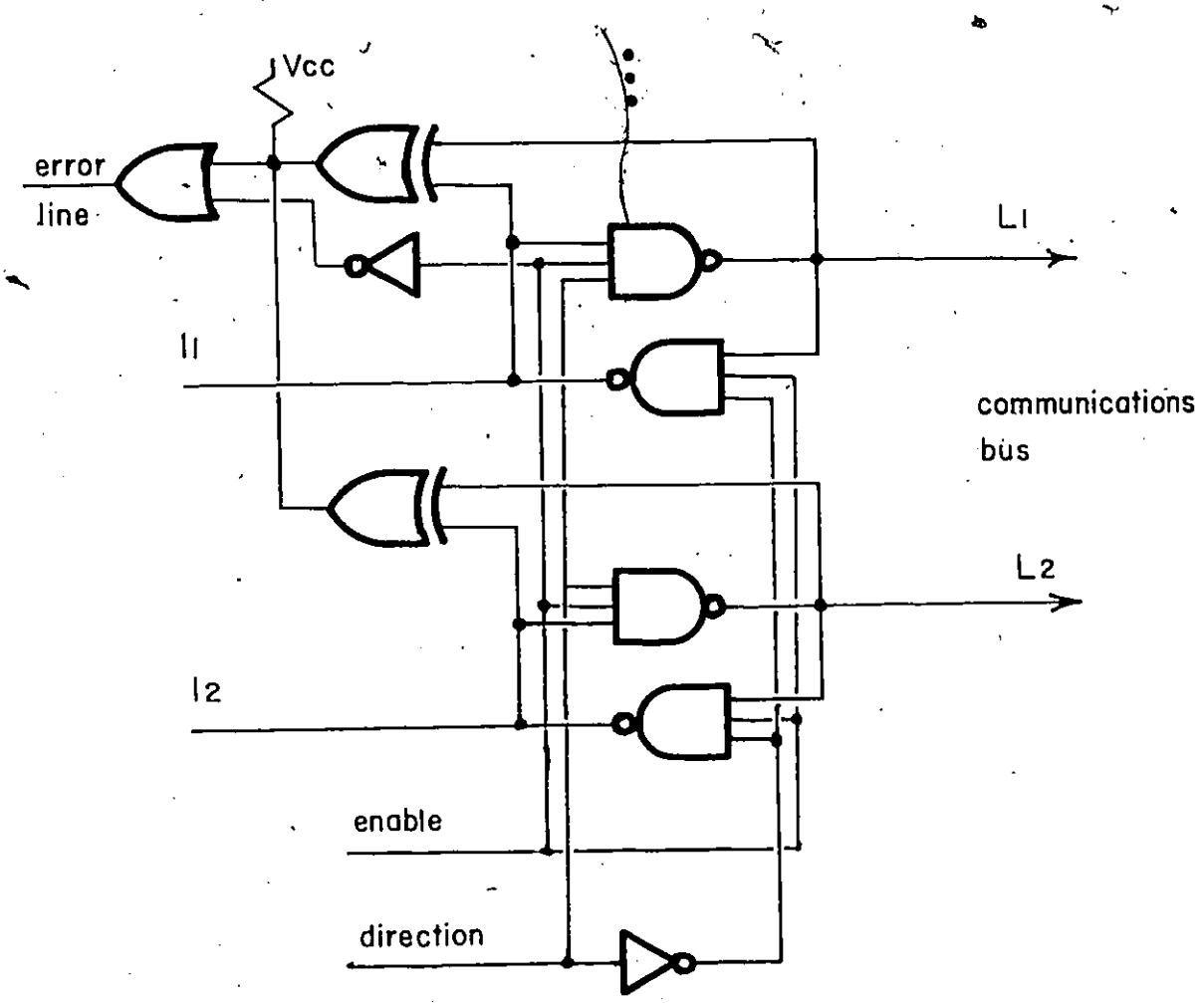
$$A \cdot (a_1, a_2, \cdots a_i) + B \cdot (b_1, b_2, \cdots b_i) + C \cdot (c_1, c_2, \cdots c_i)$$

Gate complexity = $i \times n + i$
$= i(n + 1)$

Wire complexity = $i(2n + 1)$

i = width of gates
n = no. of gates

Bus Driver / Receiver



Gate complexity = n X 2 + 1 + (n + 2)

Wire complexity = n X 2 X 3 + 1 + (2 X n + 2 + 1)

n = width ( ) = error detection circuit

Alphabetic Listing of Mnemonics

Instruction types:

ADD	addition	two arguments
AND	logical 'and'	two arguments
DIV	divide	two arguments
JMP	conditional jump	two arguments
LOAD	load value from main memory	one argument
MOVE	transfer data word in string	two arguments
MULT	multiply	two arguments
OPR	register operation	two arguments
OR	logical inclusive 'or'	two arguments
REPLACE	replace Data Words in memory	one argument
SKIP	conditional skip	one argument
SRO	self register operation	one argument
STORE	store Data Word in memory	one argument
SUB	subtract	two arguments
XOR	logical exclusive 'or'	two arguments

Modifiers for JMP and SKIP instructions:

ANEG	jump on negative Data Word	set P.C. to phase A
ANZR	jump on non-zero Data Word	set P.C. to phase A
AOFL	jump on overflow bit = 1	set P.C. to phase A
APOS	jump on positive Data Word	set P.C. to phase A
AUNC	jump unconditional	set P.C. to phase A
AZER	jump on zero Data Word	set P.C. to phase A
AZOF	jump on overflow bit = 0	set P.C. to phase A
AZRO	jump on zero DW and ovfl	set P.C. to phase A
BNEG	jump on negative Data Word	set P.C. to phase B
BNZR	jump on non-zero Data Word	set P.C. to phase B
BOFL	jump on overflow bit = 1	set P.C. to phase B
BPOS	jump on positive Data Word	set P.C. to phase B
BUNC	jump unconditional	set P.C. to phase B
BZER	jump on zero Data Word	set P.C. to phase B
BZOF	jump on overflow bit = 0	set P.C. to phase B
BZRO	jump on zero DW and ovfl	set P.C. to phase B

Modifiers for SRO and OPR instructions:

ALS	arithmetic left shift
ARS	arithmetic right shift
CLR	clear Data Word
COF	clear overflow bit
DCR	decrement Data Word and skip on result = 0
EXIT	exit from routine
INC	increment Data Word and skip on result = 0
LLS	logical left shift
LRS	logical right shift
NOP	no operation
OCR	one's complement Data Word
ROT	rotate Data Word
RST	reset, clear Data Word and overflow bit
RWO	rotate with overflow
SOF	set overflow bit to 1
TCR	two's complement the Data Word

MICROPROCESSOR SCORECARD

MICROPROCESSOR SCORECARD	CLASSIFICATION	TECHNOLOGY	PARIS FAMILY: Clock Driver	PARIS FAMILY: I/O Controller	PARIS FAMILY: UART/USRT	PARIS FAMILY: RAM	PARIS FAMILY: ROM/PROM	PARIS FAMILY: Interfaces	FEATURES: One Chip CPU	FEATURES: Microprogrammed	FEATURES: Accumulator Slice	FEATURES: DMA Ability	FEATURES: BCD Arithmetic	WORD SIZE (Data Increments)	ADDRESS CAPACITY (Program Words)	CLOCK (MHz Phases)	REGISTER ADD TIME (usec per Data Word)	NUMBER OF CPU REGISTERS: ALU	NUMBER OF CPU REGISTERS: XR	NUMBER OF CPU REGISTERS: GP	RETURN STACK SIZE (No. x Bits)	VOLTAGES REQUIRED	POWER DISSIPATION	OPERATING TEMPERATURE RANGE (°C)	PACKAGE SIZES (DIP Pins)	PRICE RANGE (Approx. 100 Qty CPU)	STATUS: First Samples	STATUS: First Deliveries	REMARKS
BURROUGHS MINI-D	8 Bit CPU	PMOS							✓					8 12	256	1000 1	5	3		1	8 x 8	12 -5		0 – 70	16	5 65	2975	3975	ROM on CPU
ELECTRONIC ARRAYS	8 Bit CPU	NMOS		✓		✓	✓		✓		✓	✓	✓	8 8	64K	1000 3		1	4	16	8 x 16				40		2975		16 Byte Stack Operations
FAIRCHILD F-8	8 Bit CPU	NMOS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	8 8	64K	2000 0	2	55			(RAM)	-5 +12	6	0 – 70	40	5 75	4974	1075	Clock on Chip
FAIRCHILD PPS 25	4 Bit CPU	PMOS		✓		✓	✓						✓	4 x 25 12	4096	400 2	12 5	1			4 x 12	5 -5	5	0 – 70	16 18 24 40	5 69		2971	
INTEL 4004	4 Bit CPU	PMOS	✓	✓		✓	✓	✓	✓				✓	4 8	4K	740 2	10 8	1		16	3 x 12	15 or 1 10 -5	10	0 – 70	16	5 10	2971	4971	
INTEL 4040	4 Bit CPU	PMOS	✓	✓		✓	✓	✓	✓	✓			✓	4 8	4K	740 2	10 8	1		24	7 x 12	15 or 1 10 +5	10	0 – 70	16 24	5 60	4974	4974	8K Program Space in Two Banks
INTEL 8008-1	8 Bit CPU	PMOS	✓	✓	✓	✓	✓	✓	✓	✓				8 8	16K	500 2	12 5	1		6	7 x 14	-9 -5	10	0 – 70	18	5 50	4071	1072	Second Source MIL
INTEL 8080	8 Bit CPU	NMOS	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	8 8	64K	2000 2	2	1		6	(RAM)	-5 +5 +12	10	0 – 70	40	5 290	4075	2974	
INTEL 3001	2 Bit Slice	Bipolar				✓	✓	✓	✓		✓		✓	24 16	512	5000 1	16 5	2		16	(ROM)	5	6 5	0 – 70	28 40	5 550	3074	3074	
INTERSIL 6100	12 Bit CPU	CMOS	✓			✓		✓	✓			✓		12 12	4K	4000 0	5	1		1	Modified Program	5	81	-55 – +125	28 40	5 175	4074	1075	PDP 8 Code Clock on Chip
MONOLITHIC MEMORIES 6701	4 Bit ALU	Bipolar								✓				4 17		5000 1	7	3		16	(ROM)	+5	3	0 – 75	40	5 85	1074	2974	
MOSTEK 5065	8 Bit CPU	PMOS							✓	✓		✓	✓	8 8	32K	1400 2	16	3			(RAM)	12 5 +5	3	0 – 50	40	5 100	1074	3074	3 State CPU
MOTOROLA 6800	8 Bit CPU	NMOS		✓	✓	✓	✓		✓	✓		✓	✓	8 8	64K	1000 2	2	2	1		(RAM)	5	25	0 – 70	24 40	5 150	2974	4974	Second Source AMI
NATIONAL CMP-8	8 Bit CPU	NMOS		✓	✓	✓	✓	✓	✓	✓		✓	✓	8 8	64K	2	1 5	2	2		(RAM)				40		1075	2975	
NATIONAL GPC P	4 Bit Slice	PMOS								✓	✓	✓		4 8 12	16K	715 4	14	8			16 x 4 8	-12 +5	3	0 – 70	22 24	5 160	1075	3075	16 x 4 R x B
NATIONAL IMP-4	4 Bit CPU	PMOS						✓	✓		✓		✓	4 4	4096	500 4	12	4			7 x 12	-12 +5	10	0 – 70	24 40	5 150	3074	4074	16 x 4 Data Stack
NATIONAL IMP-8	8 Bit CPU	PMOS							✓		✓	✓	✓	8 8	64K	715 4	4 5	3	1		16 x 8	-12 +5	10	0 – 70	22 24	5 230	4075	1074	
NATIONAL IMP-16	16 Bit CPU	PMOS							✓		✓	✓	✓	16 16	64K	715 4	4 5	2	2		16 x 16	-12 +5	14	0 – 70	22 24	5 290	1075	3075	
RAYTHEON RP-16	4 Bit Slice	Bipolar							✓		✓			4 8 16	64K	5000 3	3	1	1	2	(RAM)	5		-55 – +125	40		4074		
RCA COSMAC	8 Bit CPU	CMOS							✓	✓		✓	✓	8 8	64K	2175 1	8	1		2	1 x 16	5 -12	81	55 – +125	28 40	5 395	4074	2975	Up 2 to 20
ROCKWELL PPS-4	4 Bit CPU	PMOS	✓	✓	✓	✓	✓		✓		✓		✓	4 8	4K	200 2	5	1		1	2 x 12	17	225	0 – 70	42	5 45	1072	2972	8K Program Space in Two Banks
ROCKWELL PPS-8	8 Bit CPU	PMOS	✓	✓	✓	✓	✓		✓	✓		✓	✓	8 8	32K	215 2	4	1	1	2	(RAM)	17	3	0 – 70	42	5 67	4074	1075	
SIGNETICS 2650	8 Bit CPU	NMOS							✓	✓			✓	8 8	32K	1200 5	4 8	3		5	8 x 15	5	5	0 – 70	40	5 120	4074	1075	
TOSHIBA TLCS-12	12 Bit CPU	NMOS		✓		✓	✓		✓	✓	✓			12 12	4K	1000 3	12	4		5	(RAM)	-5 +5	8	-20 – +50	16 24 28 42	5 275	2874	3074	
TRANSITRON TMC 1601	4 Bit Slice	Bipolar							✓		✓	✓	✓	16 16	32K		4				(RAM)				40		2975	3075	Level B

References

1 Digital Equipment Corporation, "Control Handbook", pp.1-5; 1971.

2 Grimm,R.A., "Hewlett-Packard Automatic Test Systems", Hewlett-Packard Journal, Vol. 20, No. 12, pp.15-20; 1969.

3 McAleer,H.T., "A Look at Automated Testing", IEEE Spectrum, Vol. 8, No. 5, pp. 63-78; May, 1971.

4 Susskind,A.K., "Diagnostics for Logic Networks", IEEE Spectrum, Vol. 10,No. 10,pp. 40-47; October, 1973.

5 Eleccio,M., "Automatic Testing: Quality Raiser, Dollar Saver", IEEE Spectrum,Vol.11,No.5,pp.38-43;August 1974.

6 Peterson and Weldon, "Error Correcting Codes", The MIT Press, 1972.

7 Kneale, S.G., "Reliability of Parallel Systems with Repair and Switching", Proc. 7th Natl. Symp. Rel.and Q.C. pp. 1929-1933; 1961.

8 Chien,R.T., "Memory Error Control: Beyond Parity", IEEE Spectrum, pp. 18-23; July 1973.

9 Berlekamp,E.R., "Algebraic Coding Theory", pp.298-324, McGraw Hill, 1968.

10 James L. Massey and Oscar N. Garcia, "Error-Correcting Codes in Computer Arithmetic", from "Advances in Information Systems Science", Vol. 4, pp. 273-324; 1972.

11 Sklar S., "2's Complement Arithmetic Operations", Computer Design, pp. 115-121; May 1972.

12 Millman and Halkias, "Integrated Electronics", pp. 647; McGraw Hill, 1972.

13 RCA, "COS/MOS Digital", RCA Solid State 74 Data Book Series, SSD-203B; 1974.