

A COMPUTER ARCHITECTURE FOR  
DISCRETE MANUFACTURING

A THESIS

Presented to

The Faculty of the Division of Graduate  
Studies and Research

By

Robert Baugh Sledge, Jr.

In Partial Fulfillment

of the Requirements for the Degree  
Master of Science in Electrical Engineering

Georgia Institute of Technology

August, 1974

A COMPUTER ARCHITECTURE FOR  
DISCRETE MANUFACTURING

Approved:

E. O. Alford, Chairman

E. B. Wagstaff

J. H. Schlag

Date approved by Chairman: 9/20/74

## ACKNOWLEDGMENTS

I wish to take this opportunity to thank my advisor, Dr. Cecil O. Alford, for his help in identifying the problem and for his aid and advice during the research. Thanks are due also to Dr. E. B. Wagstaff for his numerous critical suggestions which resulted in improving the final copy.

I would also like to recognize the help of my lord Jesus Christ, without whose help this thesis would not have been completed within the allocated time constraints.

## TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS . . . . .	ii
LIST OF TABLES . . . . .	v
LIST OF ILLUSTRATIONS . . . . .	vi
SUMMARY . . . . .	viii
Chapter	
I. INTRODUCTION AND PAST APPROACHES . . . . .	1
Introduction	
II. IDENTIFICATION OF SYSTEM REQUIREMENTS . . . . .	17
Microprocessor System Requirements	
III. PROPOSED SYSTEM ARCHITECTURE . . . . .	31
Level (1) System Architecture	
Level (1) Microprocessor Architecture	
IV. COMPARISON AND EVALUATION OF PERFORMANCE . . . . .	72
Performance Evaluation	
Conclusions	
APPENDICES	
A. INPUT MODULE CIRCUITRY . . . . .	91
B. INTERRUPT MODULE . . . . .	94
C. OUTPUT MODULE CIRCUITRY . . . . .	97
D. MEMORY MODULE . . . . .	99
E. BASIC INSTRUCTION SET FOR THE GT 1248 . . . . .	105
F. INTEL 8080 PROCESS CONTROL ROUTINES . . . . .	134
G. ROBOT CONTROL PROGRAMS . . . . .	140

	Page
BIBLIOGRAPHY . . . . .	146

## LIST OF TABLES

Table	Page
1. Computer Usage in Manufacturing Operations . . . . .	8
2. Computer Functional Requirements . . . . .	12
3. Characteristics of the Level (1) System . . . . .	30
4. Comparison of Microprocessor Capabilities . . . . .	47
5. Flag Flip Flop Functions . . . . .	51
6. ALU Operation Set . . . . .	53
7. Field Specifications for the Register Control Instruction .	60
8. Double Register Field Specifications . . . . .	61
9. Fetch Instruction Functions . . . . .	62
10. Jump Instruction Functions . . . . .	66
11. Pulse Instruction Functions . . . . .	69
12. Characteristics of the GT 1248 Instruction Set . . . . .	73
13. Relative Performance for Bit Oriented Instructions . . . . .	79
14. Evaluation of Test Program Results . . . . .	86
15. Programming Symbols . . . . .	106

## LIST OF ILLUSTRATIONS

Figure	Page
1. IBM Comats Computer System . . . . .	4
2. Diagram of Computer/Process Interfacing . . . . .	6
3. Characteristics of Manufacturing Operations . . . . .	7
4. Four-Level Hierarchical Control System . . . . .	10
5. Centralized Computer Wiring Costs . . . . .	14
6. Interlevel Communication . . . . .	27
7. Level (1) Computer System Architecture . . . . .	32
8. Partitioning of CPU Address Space . . . . .	34
9. Block Diagram of Input System . . . . .	37
10. Output System Block Diagram . . . . .	40
11. Memory System Block Diagram . . . . .	43
12. Flowchart for Vertical Information Transfer . . . . .	45
13. Microprocessor CPU Architecture . . . . .	48
14. Two Phase Clock Used for System Timing . . . . .	54
15. Control Section Architecture . . . . .	56
16. Format for the RC, DR and Fetch Microinstructions . . . . .	59
17. Control Store Address Mapping . . . . .	64
18. Format for the Jump, Pulse and Emit Microinstructions . . . . .	65
19. Robot Control Configuration . . . . .	81
20. Memory and I/O Map for the Robot Control Problem . . . . .	83
21. Robot Control Flowchart . . . . .	84
22. Input Module Circuit Diagram . . . . .	92

Figure	Page
23. Interrupt Module Circuitry . . . . .	96
24. Output Module Logic Circuitry . . . . .	98
25. Memory Module Circuitry . . . . .	100
26. Memory Read Timing Sequence . . . . .	102
27. Memory Write Timing Sequence . . . . .	103



## SUMMARY

Over the last decade, industrial control systems utilizing digital computers have typically implemented large centralized computer facilities. Two major drawbacks to this approach have been noted. First, the reliability of the central computer facility must be maintained near 100 percent since a system failure at this level can cause a cessation of all manufacturing operations. The second objection stems from the fact that excessively long control loops are necessary when using the centralized facility. Economic and noise shielding problems accompany the use of these control lines.

A four-level distributed computer hierarchy has been proposed for industrial control applications. Different segments of the overall control problem are assigned to dedicated computers at each of the four levels. The levels are: 1) Material Flow Level, 2) Process Control Level, 3) Production Control Level and 4) Management Control Level. At the Material and Process Control levels, the emphasis is on real time direct control of machines and processes while the Production and Management Control levels are concerned with matters such as scheduling and forecasting.

It has been proposed that real time control of machines and processes at the Level (1) Material Flow Level may be efficiently implemented using microprocessor CPU chips. In this scheme, each machine or process on the plant floor would have a dedicated real time computer system effecting control over that particular process. Such a system

promises to offer benefits over the centralized concept with regards to both reliability and economic factors.

The object of this research was to identify the specific requirements placed on a microprocessor based computer control system capable of operating as a Level (1) processor. Using these requirements to specify the system, an architecture for the overall Level (1) system was developed. This architecture covered both a theoretical microprocessor architecture as well as the computer system hardware architecture.

It was concluded that the microprocessor CPU should have an instruction set that directly supports the control of machines and processes. This means that the microprocessor should be able to manipulate both single and double word data as well as single bit data. The microprocessor system must have an expandable multilevel interrupt structure to receive interrupts from the Level (2) supervisory computer as well as from the plant floor. A writeable control store control section was shown to offer an increase in system throughput and flexibility.

In comparing the performance of the Intel 8080 microprocessor CPU with that of the theoretical microprocessor developed in the paper, it was found that performance was comparable in operations working with eight bit (single word) data. The 8080 proved to be moderately deficient in its ability to manipulate double word data due to its reduced double word instruction set. In manipulating single bit data, the 8080 suffered its worst performance degradation in comparison to the theoretical microprocessor with execution times on the order of six times as large.

It was concluded that the Intel 8080 is capable of operating as

a Level (1) microprocessor within applications compatible with its execution speed. A modification of the instruction set to better manipulate double word data and the addition of a bit processing feature would significantly increase the process control capabilities of the Intel 8080.

## CHAPTER I

### INTRODUCTION AND PAST APPROACHES

#### Introduction

##### Definition of the Problem

Manufacturing control systems over the past ten years have tended toward digital control systems utilizing one or two large central computer systems exercising control over all processes within an entire plant. Typically the entire spectrum of applications programs ranging from management to process control are run in a multiprogramming environment within the computer system, necessitating a large central facility.

Major drawbacks resulting from this approach to manufacturing control have been noted. The centralized facility concept places an extreme emphasis on central system reliability in that a major system failure results in the termination of all control functions, resulting in a manufacturing halt. It has been reported that system availability (uptime divided by uptime plus downtime) in some cases has been less than 99.5 percent [1]. In order to increase system availability, some systems have utilized the principle of dynamic backup in which a second backup central processor is incorporated into the computer system. In the event of a failure in the main central processing unit (CPU), the secondary unit is automatically switched into the system, yielding availabilities as high as 99.95 percent. In addition to the reliability problem, the centralized concept has problems associated with program

interaction in the multiprogramming environment and high cabling costs involved in bringing large amounts of data from the plant floor to the computer facility.

The problems associated with large centralized control facilities have led to a breaking apart of the control function into smaller subsets that may be more efficiently handled by dedicated computers in a hierarchical computer system.

### Background Information

The evolution of computerized process control stemmed from two directions, the automobile industry and the aerospace industry. With the advent of numerically controlled machine tools in the early 1950's, the trend was set for direct digital control of machines in on-line operation [2]. Computer control systems typically cover a broad range of control functions in the total plant automation role, incorporating management, production and equipment control functions. Management information systems and production support systems are typically not real time critical activities, although they may account for a substantial portion of the computer workload in the total system. Equipment control functions are in most applications real time critical, demanding relatively frequent interaction between the control system and the controlled process. Machines and processes under the equipment control function usually fall into one of three main task categories referred to as Make, Move or Test [3].

Early attempts at decentralizing the computer control facility often resulted in a nonintegrated system structure. Such a system is characterized by computers, totally unrelated to each other, spread out

over the plant. Each computer provides equipment control over Make, Move and Test operations occurring simultaneously in from one to a number of separate processes or operations on the plant floor [4].

In an effort to overcome the shortcomings of the nonintegrated system, the trend has been toward a hierarchy of computers. Each computer in the hierarchy is associated with a particular level of the control function. The IBM COMATS computer system [5] shown in Figure 1 is an example of one of the first large scale hierarchical manufacturing control systems. The hierarchical nature of this system stems from the fact that the 1460 computers act as supervisor over the terminals which in turn provide the direct interfacing and control at the plant floor. Three types of terminals are used with this system. A universal tester, a process control terminal built around the IBM 1441 CPU, or a data acquisition terminal which acts as an Input/Output (I/O) buffer can be used to interface the computer to the plant. Up to 99 such terminals can be multiplexed to either of the two IBM 1460 computers. One 1460 is normally used to handle the real time interrupts generated by the terminals while the other is used for time shared processing and dynamic backup capability.

COMATS was the predecessor to the IBM Manufacturing Process Control System [6]. This architecture is similar to the COMATS architecture but was designed to be more versatile. Two system /360 computers provide data analysis and large data banks for the satellite computers. The satellite computers interface to the process and test equipment through sensor based Input/Output typically employing standard sensors such as input and output contact interfaces, analog-to-digital (A/D) interfaces

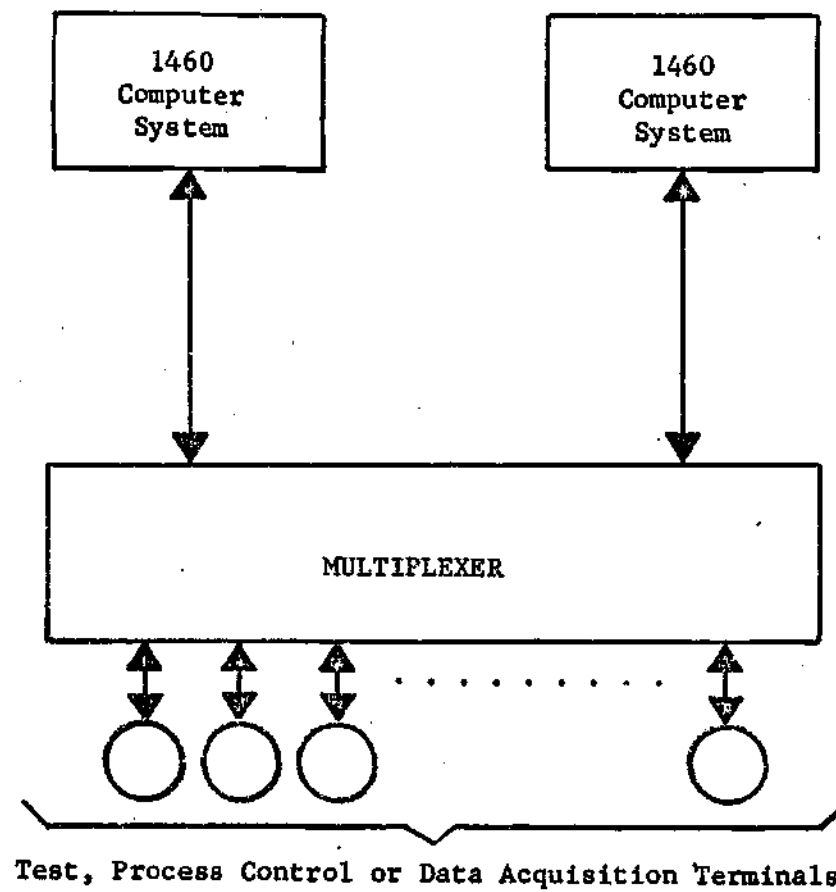


Figure 1. IBM COMATS Computer System

and digital-to-analog (D/A) interfaces [7] as shown in Figure 2. This IBM system placed most of the data processing requirements at the higher level /360 computers in an effort to minimize satellite computer requirements. Satellite computers were typically IBM 1130, 1800 or System /360 series computers and each satellite computer typically controlled more than one process. A special purpose operating system called PCOS (Process Control Operating System) was developed for the main /360 computers to permit them to act in a supervisory and data processing mode for the satellite computers [8]. A high data rate intelligent multiplexing system known as the TCU (Transmission Control Unit) was developed to handle data requests and multiplexing between the supervisory computers and satellite computers [9]. This IBM process control system represented a significant advance in integrated hierarchical control, but the total plant automation function encompasses more than just equipment control. The management information system function must be incorporated into the computer hierarchy. Additional integrated computer control systems similar to the two mentioned here are described in the literature [10], [11], [12].

Hammond and Oh have characterized manufacturing operations as being divided into four levels as shown in Figure 3. These levels are: 1) the Material Flow Level, 2) the Process Control Level, 3) the Production Control Level and 4) the Management Control Level [2]. Computer usage at the different levels is cited in Table 1.

The most critical response times occur at the Process Control Level where real time interaction between computer and process are crucial. At the Production and Management Control Levels the shift is



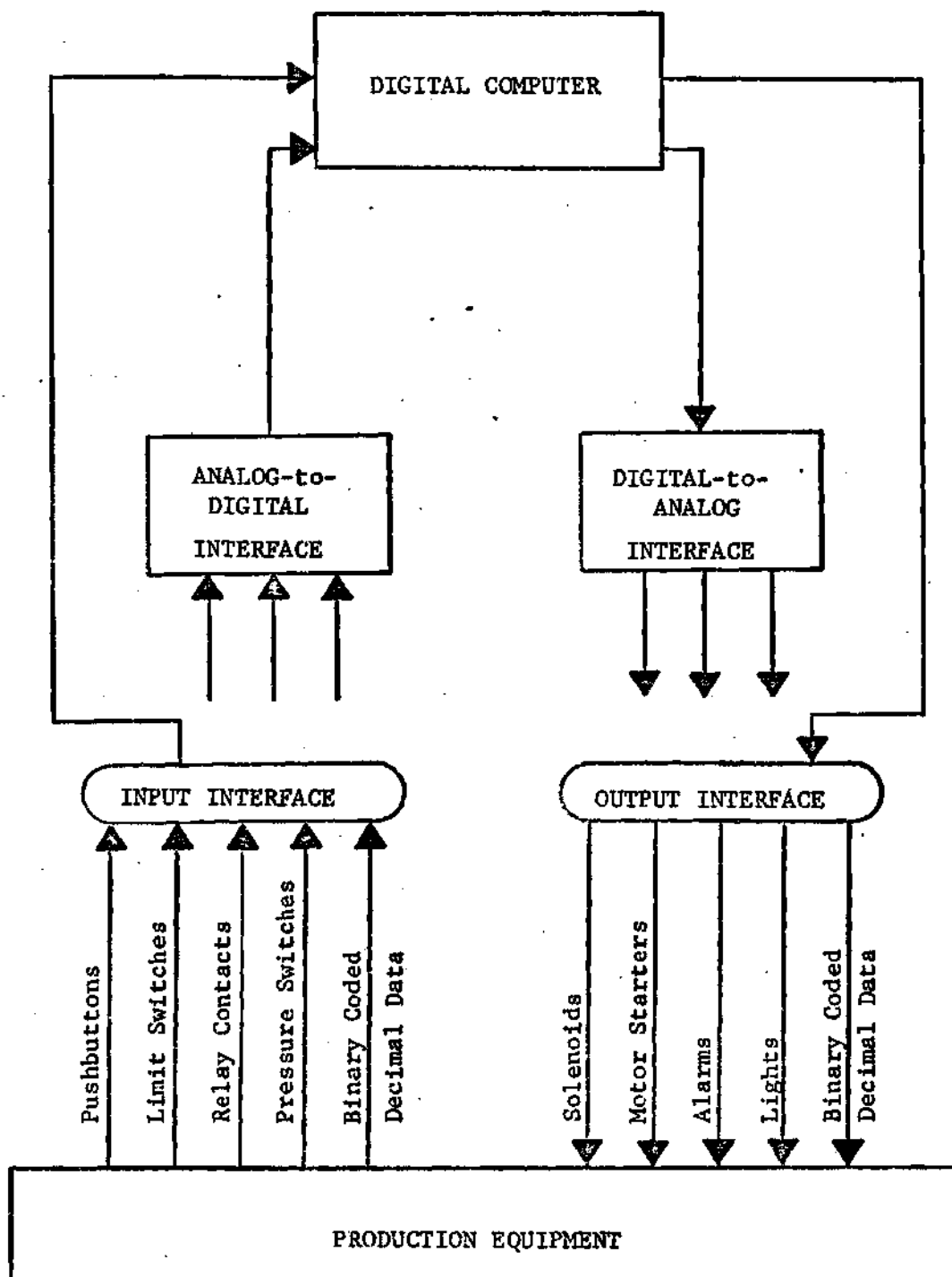


Figure 2. Diagram of Computer/Process Interfacing

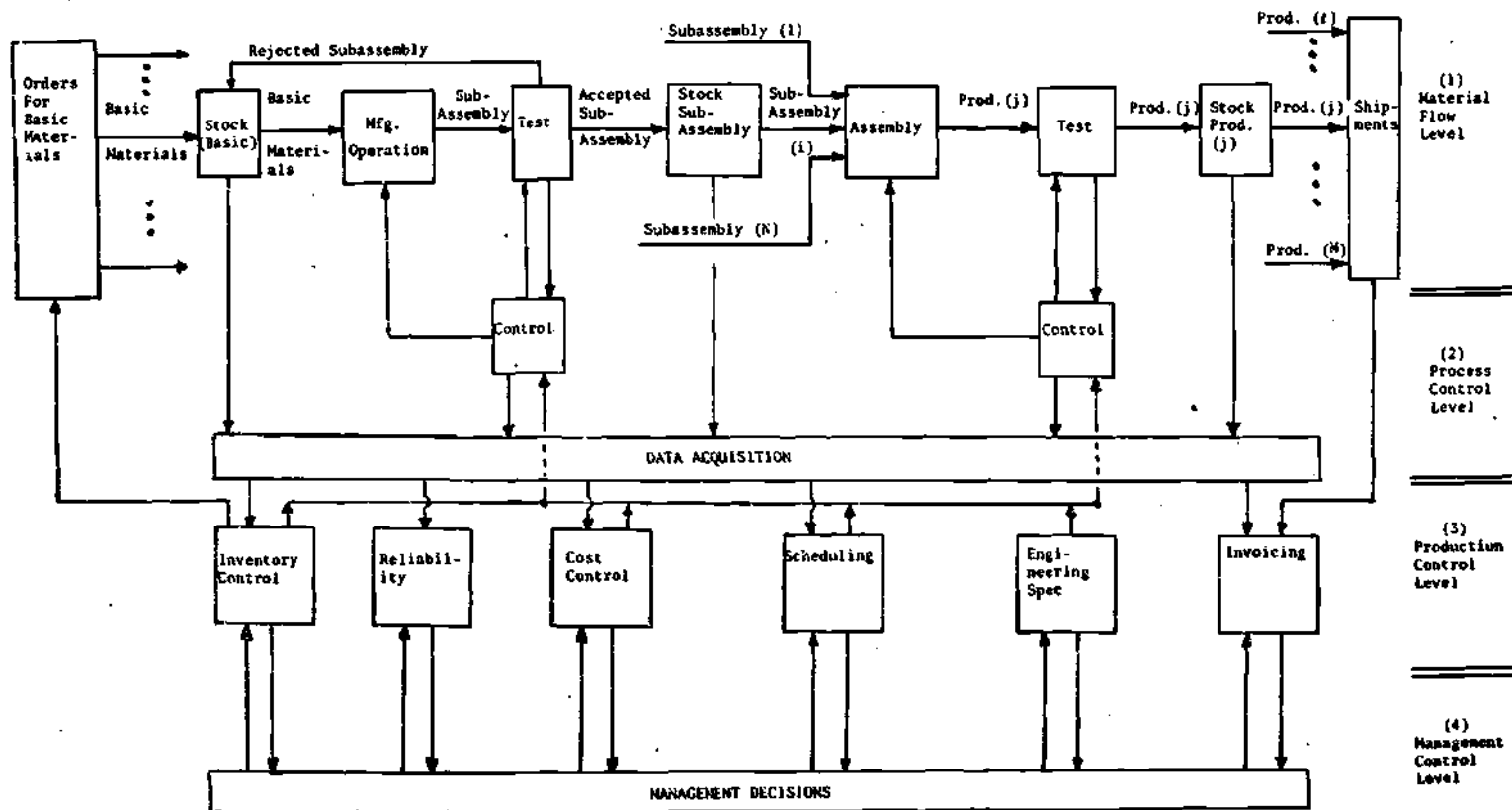


Figure 3. Characterization of Manufacturing Operations

Table 1. Computer Usage In Manufacturing Operations

	ON-LINE	OFF-LINE
Process Control	A.1 Data Acquisition	A.5 Manufacturing Support Operation (e.g., N/C tape production, IC mask generation, etc.)
	A.2 Test Control	
	A.3 Manufacturing Operation Control	
	A.4 Assembly Operation	
Production Control	B.1 Accounting	B.5 Cost Control
	B.2 Daily Production Scheduling	B.6 Inventory Control
	B.3 Inventory of Products and Parts	B.7 Quality and Reliability Parameters
	B.4 Quality Control	B.8 Computation of Production Parameters from Engineering Data
Management Control	C.1 Generation of Management Information (e.g., Profit, Resource Utilization, Payroll, Personnel Data)	C.2 Simulation Studies for Forecasting Economic Environment, Product Demand, Rates of Return, etc.
		C.3 Maintenance of Management and Engineering Information Files

away from real time response and toward more conventional batch type processing. At these two upper levels information is the basic flow quantity.

What is needed is a hierarchical computer system capable of exercising the proper control at all four levels of manufacturing operations. A computer control system architecture structured as in Figure 4 has been proposed to meet this need [1], [13]. At the lowest level of the hierarchy, Level (1), the emphasis is on real time control of a single machine or process. The computer at this level contains the program and data necessary for such control. Since it is desirable to minimize the storage requirements of the Level (1) system, the Process Control Computer System, Level (2), will contain in its mass memory files a copy of all applications programs used at Level (1). Level (2) memory may also contain those parameters that influence initial machine setup, assembly operations and required product output. Since there is no direct horizontal communication between Level (1) computers, the Level (2) Process Control Computer acts as a transmission link as well as a supervisor. The real time requirements at Level (2) are much less critical than those at Level (1) since no direct control of a given machine takes place at this level. Interlevel communication may be initiated by either Level (1) or Level (2) via interrupts to the called processor. In this hierarchical architecture, Level (2) will be used as a coordinating and supervisory control for all the machines, processes and operations at Level (1) [13].

The task of setting up primary production schedules is handled by the Production Control Computer System at Level (3). This function

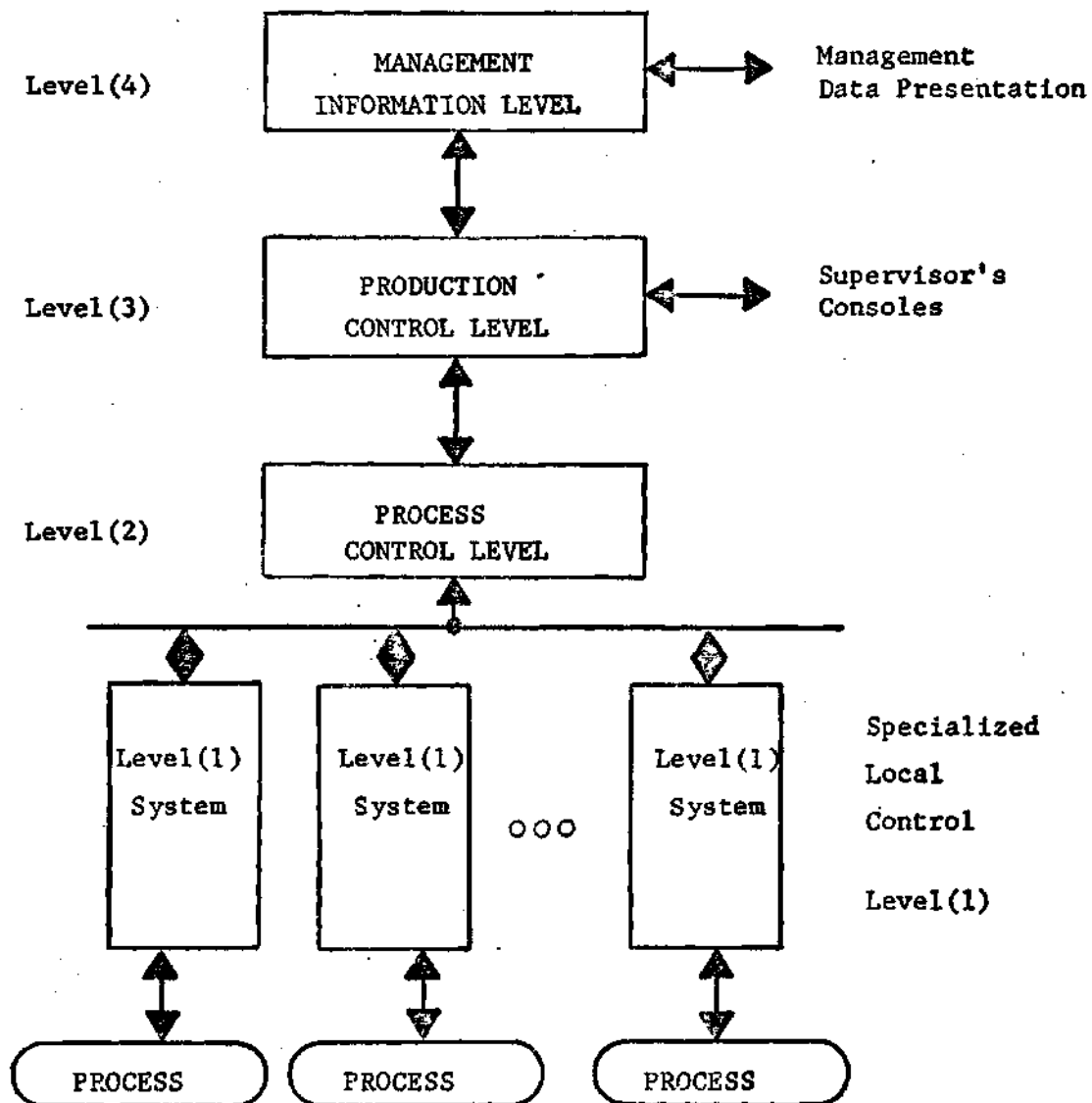


Figure 4. Four-Level Hierarchical Control System

requires input from a variety of sources associated with scheduling as detailed in Figure 3. The computer system at this level is a general purpose data processor with large memory, mass storage, high speed input/output and the capability to support multiple terminals for message requests. The system should support a number of high level languages, offer a high speed interrupt capability and feature a multiprogramming operating system [14]. Included at this level should be the necessary software to support program development in a suitable process control language for Level (1) process control programs.

The Management Control Computer System at Level (4) has the large task of generating the decisions for future plant directions and plans. Computer architecture at this level must be data base oriented and capable of the time shared support of a large number of terminals for managers. Functional requirements [13] of the various computer systems may be found in Table 2.

A four level hierarchy of control represents a shift away from the centralized control concepts of the last decade toward the distributed computer systems [14] mentioned in the literature today. This distributed system shifts the bulk of the real time process control away from a relatively large computer controlling many processes to many dedicated computers at Level (1), each controlling a single process or machine. Three advantages of using a distributed hierarchical architecture as opposed to a centralized architecture immediately suggest themselves.

Modularity. The modular system structure at Level (1) will tend to increase system availability. In a centralized system, a processor

Table 2. Computer Functional Requirements

	INFORMATION SYSTEMS FUNCTIONS	EQUIPMENT CONTROL SYSTEM FUNCTIONS
Level 4	Management Information System Planning Tools for Management Decisions General Support Functions Payments Cost Reporting Cost Estimating Pricing Tax Reporting Personnel Simulation Models	
Level 3	Production Support Functions Product Data for each item Specifications Test & Reliability Scheduling Information Resources Required Inventory Product Changes Unit Cost Sales History Critical Resources	Production Control Inventory Control Optimal Resource Allocation
Level 2 & Level 1		Data Acquisition and Control

failure can lead to many machines and processes coming to a halt for lack of control. A system failure in a Level (1) processor would only have the immediate effect of stopping one machine or process, although other processes could presumably be affected. In this case the Level (2) Supervisory Computer would rearrange the workload among the remaining machines by altering the programs in their controlling processors.

Partial System Installation. The total control system could be built up in parts. A starting system could consist of only the first two levels of the control hierarchy. As plant equipment requirements expanded, additional Level (1) systems could easily be added without having to expand Level (2) hardware requirements. The ability of one Process Control Computer to supervise many subordinate processors is due to the fact that on a time scale, communication between any Level (1) processor and the Level (2) Supervisor is envisioned as being relatively infrequent and short. Each Level (1) processor is a complete computer system capable of independent operation to the extent that the controlled process will allow. This suggests a fallback capability to Level (1) operation exclusively in the event of a major Level (2) system failure.

Wiring Economics. A centralized control facility requires that large amounts of information must be brought from the plant floor to the control computer. Figure 5 shows an analysis of the cost of the cabling necessary to run control loops from the plant floor to a centralized facility [1]. From this graph it is apparent that the cost of installing this wiring network is not insignificant, especially under the consideration that a single process on the plant floor may require



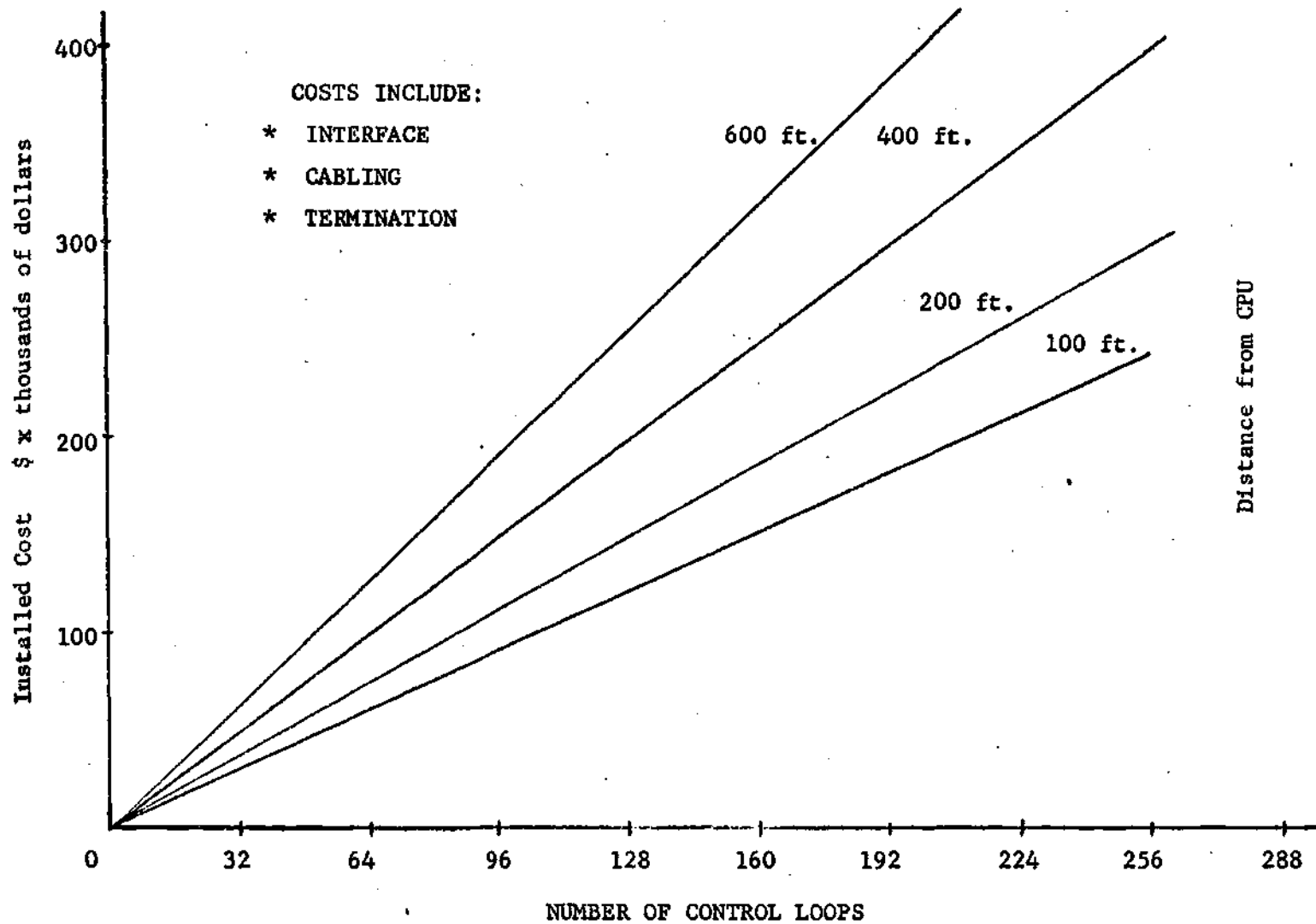


Figure 5. Centralized Computer Wiring Costs

tens of control loops. The distributed hierarchical control architecture would eliminate the majority of this cable network. All control loops for a process would terminate at the Level (1) processor controlling that particular process. One data link from each Level (1) processor to Level (2) Supervisor is all that is required, thus eliminating the need for many long control loops. Other problems associated with noise shielding and data multiplexing are also eliminated using on-site computers.

#### Purpose of the Research

In proposing a four level control hierarchy, both Alford and Keyes have suggested that the Level (1) computer system may be efficiently implemented using microprocessors as the central processing unit. This research will seek to determine the requirements placed on a Level (1) microprocessor system under the constraints of the distributed hierarchical control system. Various methods of satisfying these requirements will be examined. A theoretical microprocessor system architecture will be investigated that will efficiently satisfy Level (1) control requirements. Structure for the memory system and the input/output system will be examined as well as the internal architecture of a Read Only Control Store (microprogrammable) microprocessor.

In addition, the performance of the Level (1) microprocessor control system will be evaluated using two different CPU's. The performance using the theoretical microprogrammable CPU will be compared to that using the Intel 8080 microprocessor CPU chip. The test control problem will be oriented toward control of a manufacturing robot. Relative advantages and disadvantages of each system will be discussed in light

of the findings.

## CHAPTER II

### IDENTIFICATION OF SYSTEM REQUIREMENTS

#### Microprocessor System Requirements

In this section, the specific requirements imposed on the Level (1) microprocessor system will be identified under the constraints of the four level hierarchical control system. As these requirements are identified and defined, they will specify the system structure required for Level (1) control. In the past, commercial control systems have often combined Level (1) and Level (2) into a single computer system used to control many separate processes. A number of such minicomputer systems were originally designed as data processing machines, resulting in a bulky software package to allow them to function as a real time control processing system.

In an effort to identify purely Level (1) control requirements, the structure of a purely process control minicomputer system has been examined with regard to its Level (1) functions. The Texas Instruments 960 is a minicomputer control system designed exclusively for process control utilizing an I/O system, operating system and instruction set tailored to the unique control environment [15], [16], [17]. Those features of the TI 960 that are directly applicable to a Level (1) control system, as well as others unique to the hierarchical structure are reflected in the following Level (1) system requirements.

### Instruction Set

The Level (1) microprocessor CPU should have an instruction set designed especially for the control of machines and processes [16]. Currently available microprocessors have a machine language instruction set, that is, the set of elementary operations specified by software and executed by hardware, designed for general purpose data manipulation. Such an instruction set is suitable for many applications, but exhibits coding inefficiencies when used in implementing process control instructions. For instance, a common process control instruction involves setting an output control line to a high or low state based on the test of a specified input line. Normally, this entire test and set sequence would be specified by a single command in an arbitrary high level process control language such as PROSPRO, which is used on the IBM 1800 system [3]. The actual sequence of machine instructions (object code) necessary to perform this test and set command would then be generated by the process control language compiler. But whether this object code is generated by hand or by a compiler, a relatively large number of machine language instructions will be necessary in order to implement the test and set command. Two approaches to this problem that have been used extensively are inline programming and subroutine programming.

Inline Programming. Each time a high level control instruction must be executed, the machine language instructions necessary for its execution are written sequentially into memory. This approach has the advantage of simplicity but suffers in reduced speed due to the multiple memory accesses needed to fetch the object code. It is felt that inline coding would also be prohibitive from a memory space standpoint due to

economic considerations and limited memory capabilities of microprocessor systems.

Subroutine Programming. In the utilization of this method the section of object code that executes each process control command is written as a subroutine. Thus, instead of writing object code inline, a subroutine call is used to reach a desired section of machine language instructions. Memory must be either permanently allocated to contain the entire set of subroutines corresponding to all process control commands or only those subroutines actually used in a particular program must be passed to the Level (1) memory along with the control program. Subroutine programming has the advantage of requiring less memory than inline programming since each segment of object code is written only once, but a loss of speed due to subroutine linkages is encountered.

These considerations lead to requiring an instruction set designed specifically for process control. In the example given above, there would be a "test and set" machine language instruction included in the CPU's repertoire of instructions. A not necessarily complete listing of the types of instructions to be included in the instruction set is given in Appendix E.

A substantial savings in speed is realized by storing in the control section of the CPU those sequences of operations that implement a process control command rather than storing those sequences as object code in the memory of the Level (1) system. Substantial memory savings are also realized by reducing inline programs or process control subroutines into instructions included in the CPU's instruction set.

### Bit and Byte Oriented System

Industrial sensors are many times bit oriented, i.e., the valve is either open or closed, the switch on or off. Analog-to-digital and digital-to-analog converters are primarily byte oriented devices, frequently requiring additional bytes of information to be used for multiplexing purposes. The overall architecture of the Level (1) computer system should be designed to facilitate both bit and byte I/O operations efficiently.

### Bit Manipulation at CPU Level

Closely associated with the notion of a bit and byte oriented system is the ability to easily test and set or reset individual bits within a byte at the CPU register level. This ability to easily manipulate individual bits within a register is a feature not found in the present generation of microprocessors. With this feature status words could easily be set up in a prescribed location in memory. This would allow the supervisory computer at Level (2) to check on the status of the Level (1) processor and associated machine or process by performing a direct memory access (DMA) on the status words. This concept is particularly important in assembly line control where the status of adjacent machines on the line must be known for proper overall control [16]. Also, since many of the instructions of a process control computer are bit oriented, this feature will allow for an easier implementation of these instructions by the control section of the CPU.

### Flexible Input/Output System

As previously stated, the I/O system of a Level (1) processor should possess an architecture that facilitates both bit and byte

operations. The total integration of a flexible I/O system must span I/O hardware design, CPU hardware considerations and firmware/software.

Hardware design includes all hardware considerations of the I/O system that are external to the CPU. The hardware implementing the input function must be capable of accepting addressable inputs from the plant floor. As information about specific inputs is needed by the CPU, this information must be transferred under CPU control. The input system should have the capability to gather information from a single line or a group of lines, i.e., both bit and byte input capability.

Output system considerations are likewise similar. The CPU should be able to set the state of a single output line or a group of lines simultaneously. The hardware implementing this function must contain memory to hold the state of the output line and each output line must be addressable by the CPU.

I/O architecture should be modular in nature, with the ability to arbitrarily change the number of input and output lines. It should be possible to build the I/O system up modularly, adding additional input or output modules as needs dictate. The ability to develop or use special purpose modules such as D/A or A/D converter modules and interrupt modules is also necessary. These requirements strongly suggest a bus oriented I/O architecture.

#### Microprogrammable Control Section

The requirement that the Level (1) processor have an instruction set specifically designed for process control points to the need for a microprogrammable control section. The sequence of steps necessary to carry out a process control command, such as "test and set," could then



be stored in the control memory as a microprogram and executed whenever the "test and set" instruction was fetched from memory. This would give the system designer the flexibility to create special purpose process control instructions designed to fit the particular plant environment. The feasibility of building a microprocessor utilizing a microprogrammable control section has already been demonstrated by National Semiconductor in their IMP series microprocessors [18].

It is the control section that sends and receives the control signals that synchronize the memory and I/O modules with the timing of the CPU. Thus, the control section should have input and output control lines under microprogram control. These lines would provide flexibility in interfacing the CPU with other components of the computer system.

The control section is responsible for handling primary interrupt signals. A primary interrupt indicates to the CPU that an interrupt has been received, but does not identify the interrupting device. After receipt of a primary interrupt, the computer must save the processor state, identify the device causing the interrupt and service the interrupt. Interrupt routines can be handled either by a machine language subroutine (software) or a microprogram routine (firmware). An interrupt routine stored in microprogram memory would provide for faster interrupt servicing than that of a machine language routine, but would be difficult to modify once stored. For this reason, it is felt that the processor state should be saved under microprogram control, and subsequent device identification and servicing be done by a machine language subroutine contained in the operating system. Restoring processor state after interrupt servicing could be done under microprogram control

if desired.

While there are certainly benefits to be gained from implementing a microprogrammable control section, these benefits do not come without their corresponding problems. Notably, two major problems present themselves: pin count and address translation.

If the control ROM is implemented outside the CPU chip, as it must be for a truly microprogrammable control section, then provision must be made for both address and data lines to communicate with the ROM. Consider an 8K control ROM composed of 21 bit microcode words. Without any multiplexing of address and data, 34 extra pins must be added to the microprocessor package in order to communicate with the ROM. If address and data were multiplexed over a single bus, then only 21 extra pins would be required. Even if the control section were implemented on a separate chip, the pin count per package would not be changed significantly and the total system pin count would almost double.

The problem of generating the starting address of a microprogram routine from its corresponding op code becomes especially difficult when the set of microprogram routines is subject to user change. Modifying the address of a low ROM routine would tend to change the address of all subsequent routines in higher ROM. Assigning each routine a fixed length block of ROM would tend to eliminate this problem but ROM fragmentation would result. Routines requiring more than one block would eliminate an op code for each additional block used. In an effort to eliminate unused ROM memory, an address translation mechanism could be used to translate the op code to the segment address of the corresponding microprogram routine. But this approach brings us back to the problem of modifying

routines. One solution to this problem is to permanently assign a basic instruction set in low ROM so that corresponding op codes would translate to routine segment addresses. A block or page could then be assigned to remaining unused high address op codes. These high op codes could subsequently be user microprogrammed with special purpose microprogram routines.

### Interrupt Structure

The interrupt structure of the Level (1) computer system should be modularly expandable and capable of priority interrupt servicing. A modularly expandable structure is necessary because of the multiplicity of different types of industrial machines and processes that will fall under Level (1) control. One machine may be relatively simple, with few possible interrupt conditions and therefore requiring few interrupt lines. The computer system controlling this machine would need only enough interrupt lines to service its machine plus provision for standard interrupts common to all Level (1) systems. In like manner, some Level (1) systems may control relatively complex processes exhibiting a higher number of possible interrupt conditions. With interrupt circuitry packaged modularly, the system needs may be met by using only enough modules to satisfy the particular requirement. The priority nature of the interrupt system is necessary because of the relative importance of the different events that cause interrupts. An important event should be serviced before a less important event, thus the need for a priority interrupt structure.

It should be possible to store interrupts until they are serviced so that an interrupting device need only signal its interrupt once. Each

interrupt line should have an associated mask to disable that line. Interrupts received during the execution of an instruction should not be acknowledged until that instruction has completed its execution. Upon receipt of multiple interrupts, it should be possible to nest interrupt service routines if desired so that the highest priority interrupt is always being serviced.

### Vertical Communication

An essential requirement placed upon a Level (1) computer system is that it be able to efficiently communicate with the Level (2) Supervisory computer. Why is this necessary? As stated previously, the memory requirements at Level (1) should be minimized as much as possible since a large number of Level (1) computer systems, i.e., one for each machine or process, are necessary. For instance, a particular process may operate under varying conditions over a given time period. Under each set of conditions, a different control program may be necessary for proper operation. If the complete set of control programs for this process were stored at Level (1), a great deal of memory would be necessary to store these temporarily unused programs. On the other hand, with temporarily unused programs stored in mass storage at Level (2), memory requirements at Level (1) are reduced.

Interlevel communication will be necessary to pass other types of data between Level (1) and Level (2). Transferred information may relate to initial machine setup, assembly operations, machine parameters or any other information deemed necessary. Basically, there are two modes of communication possible between the Level (2) and Level (1) systems, active and passive.

Passive Communication. The manner in which data transfers are initiated determines whether a system is operating in the active or passive mode. The situation is depicted in Figure 6. In a passive communication system, all data transfers are initiated by the Level (2) computer. This means that the Level (1) computer must keep current information needed by the Level (2) computer present in its memory at all times since a direct memory access may be initiated by the supervisory computer at any time. This method somewhat simplifies the operating system required at Level (1) but has the disadvantage of limiting overall control flexibility.

Active Communication. A system structure in which data transfers may be initiated either by Level (1) or Level (2) is defined as an active mode system. Interrupts are generated to signal the called processor that information is ready to be received or ready to be transmitted. Such an approach offers the systems designer a much greater flexibility in his design and reduces the need for a careful timing sequence for data transfer between levels. For this reason, the Level (1) microprocessor system should be capable of active mode communication.

Additional Requirements. During the time that a data transfer is taking place between Level (1) and Level (2), the Level (1) CPU should not be diverted from normal processing tasks. This restriction is necessary because of the critical real time nature of process control. If the CPU were involved in the actual data transfer, then its attention would have to be diverted from its primary task of process control. As shown in Figure 6, direct memory access is one possible solution to this

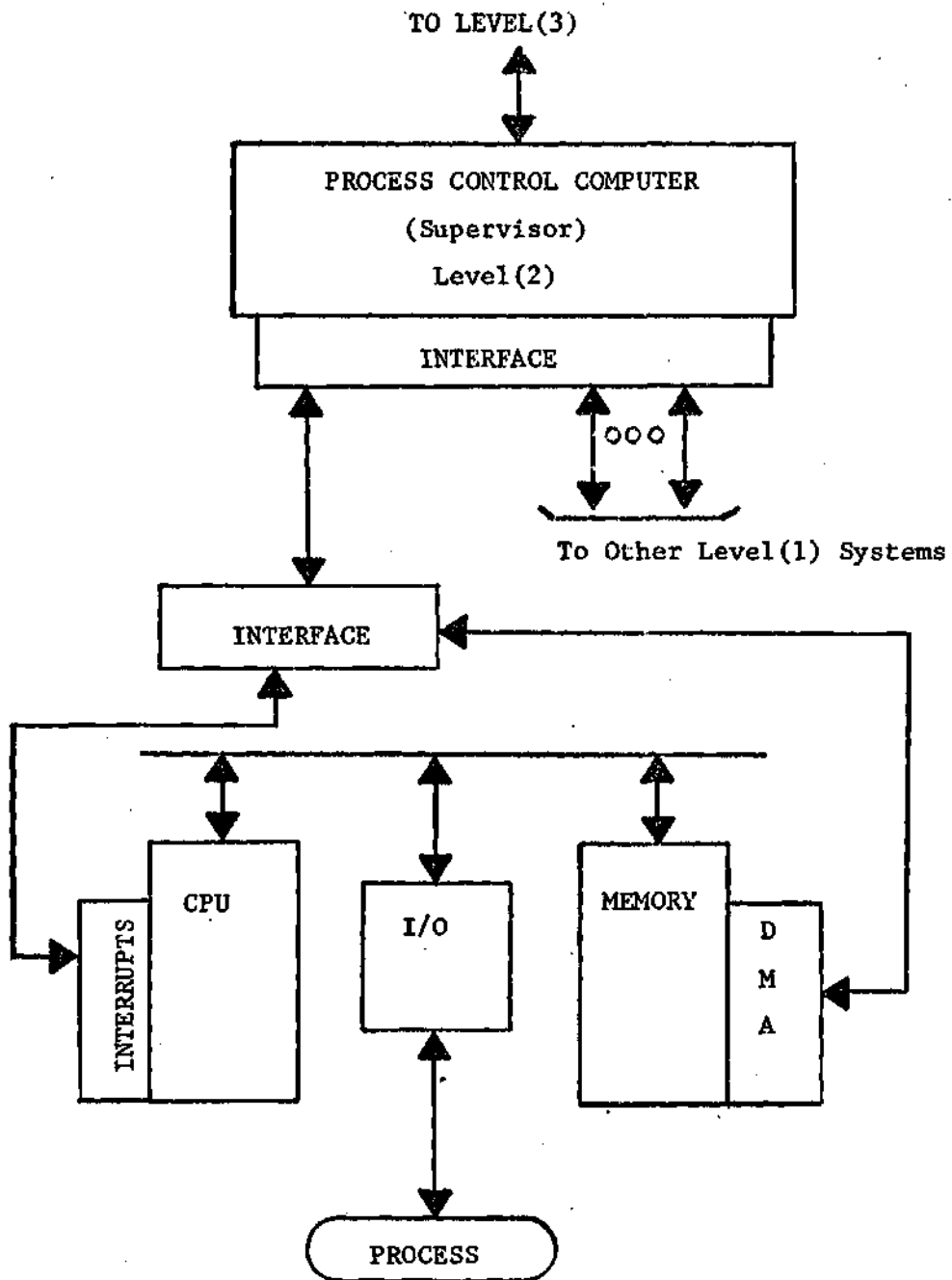


Figure 6. Interlevel Communication

problem, but provision must be made so that Level (1) memory can be accessed both by the Level (1) CPU and the Level (2) computer system at the same time. Otherwise, the Level (1) CPU will end up waiting for its own memory while a data transfer is taking place.

#### Independant Operation

In the event of a supervisory computer failure at Level (2), the interprocessor communication link for Level (1) computers would be lost along with the overall control and supervision functions performed by the Level (2) computer. Obviously, every possible effort should be taken at Level (2) to prevent a major system failure. If such a failure were to occur though, it would be desirable to continue plant operation to the fullest possible extent allowed by the situation. Some plant situations exhibit a substantial interdependence among individual processes, as in an assembly line. Unless some provision were made for limited direct communication between Level (1) computers in this particular case, a Level (2) failure would probably mean a cessation of control activities. Other process control situations are relatively independent of any other process and would be largely unaffected by a Level (2) failure.

In any case, provision should be made to load control programs into Level (1) systems from a secondary source. If this source is linked into the overall architecture at the supervisory computer level, then Level (1) architecture will be unaffected. Nevertheless, the Level (1) architecture should support a direct link from a remote source such as paper tape in case this approach were taken.

Actually, the hardware necessary to implement this requirement has already been specified in the input, output and interrupt systems. The only additions necessary are software routines for input device loading and these may be kept permanently in the operating system. Any type of input device could easily be interfaced to the Level (1) system through the I/O system.

#### Environmental Considerations

The Level (1) processor system should be able to operate in a hostile plant environment. Since this requirement is not directly related to the Level (1) architecture problem, only a cursory view of the factors involved in this problem will be examined. Factors of a plant environment usually fall into four categories: atmospheric contamination, thermal factors, mechanical factors and electromagnetic factors. Atmospheric contaminants are usually eliminated with proper system enclosures, but this can intensify the thermal problems. Mechanical factors are usually present in the form of vibration that can degrade circuit connections. Some applications may require a high acceleration resistance. Electromagnetic shielding is essential when operating in a plant environment due to the numerous sources of this interference. Provision may be a backup power supply circuit in each Level (1) computer system. Without this, a short power disturbance could destroy the contents of all Level (1) semiconductor memories.

The essential characteristics of a Level (1) system are given in Table 3.



Table 3. Characteristics of the Level (1) System

Function	Requirements
Instruction Set	Oriented toward the process control situation. Must include instructions capable of operating on individual bits and bytes in the range 12-16 bits.
Data Types	Bit, Byte, Word, Fixed Point Binary
I/O System	Capable of sending and receiving all data types to and from the plant floor.
Interrupts	Priority system capable of storing interrupts and nesting to service the highest interrupt.
Interlevel Communication	Capability to communicate with the Level (2) supervisory computer. During data transfer the Level (1) CPU should not be inhibited.

## CHAPTER III

### PROPOSED SYSTEM ARCHITECTURE

#### Level (1) System Architecture

In Chapter II, the specific requirements imposed on a Level (1) microprocessor control system were examined. In order to meet these requirements, what other system components will be necessary in addition to a microprocessor CPU chip? Basically, in addition to the CPU, there are three main system areas that must be covered: the memory system, the output system and the input system which contains the interrupt lines as well as input lines. The overall system architecture proposed for the Level (1) control system is shown in Figure 7.

#### System Structure

The major building block of this architecture is the microprogrammable microprocessor CPU and its control section. Contained within the CPU are the arithmetic logic unit, the registers and the internal busses common to all computer CPU's. The control section contains the system timing circuits, the control store memory to hold the microprograms and assorted registers, busses and decoders to carry out the CPU control function [19]. The I/O bus handles all data transfers between the CPU and input, output or memory systems. All three systems are accessed by the CPU as if they were a single large memory system composed of a mix of random access read/write and read only memory elements. Essentially this means that memory space, input system space and output

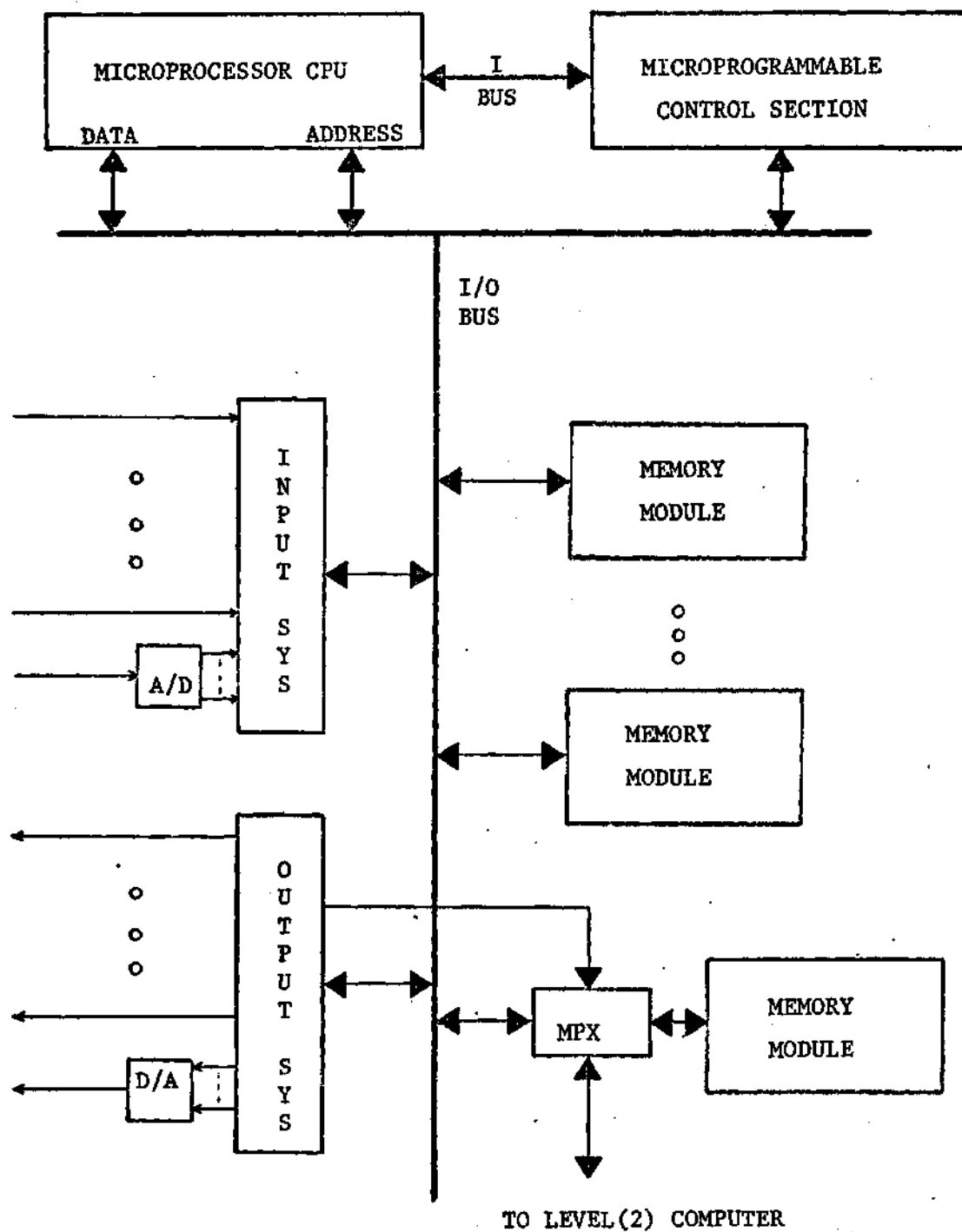


Figure 7. Level(1) Computer System Architecture

system space must all be disjoint subsets of the CPU address space. That is not to say that the CPU cannot, for example, perform a read operation on the output system. As it turns out just such an operation will have to be done. The essential point to be made here is that any word in any of the three systems must not have the same address as any other word in any other system.

Figure 8 shows one possible partitioning of the CPU address space. Essentially, the address space is subdivided into equal length pages of  $L = 2^p$  words each,  $p$  an integer. Assuming that the address bus is capable of handling an  $n$  bit address, this results in a total of  $N = 2^{(n-p)}$  pages for the address space.

It is expected that the majority of the CPU address space will be allocated to the memory system. In order to achieve the modular structure discussed in Chapter II, the memory system could be divided into modular memory modules of  $L$  words each. Each memory module used in a Level (1) system would occupy one page of memory space. The modular nature of this memory system allows memory capacity to be easily changed in order to meet changing control requirements. Serviceability is enhanced as a result of direct replacement of faulty modules. Each memory module could be composed of either random access read/write semiconductor (RAM), core or read only (ROM) memory elements.

Figure 8 shows that the last page of the address space contains the addresses for both the input and output systems. These two systems utilize a different addressing scheme from the memory system in that absolute addresses are bit addresses, although to the CPU these addresses look like word addresses. For instance, if the CPU were to load a

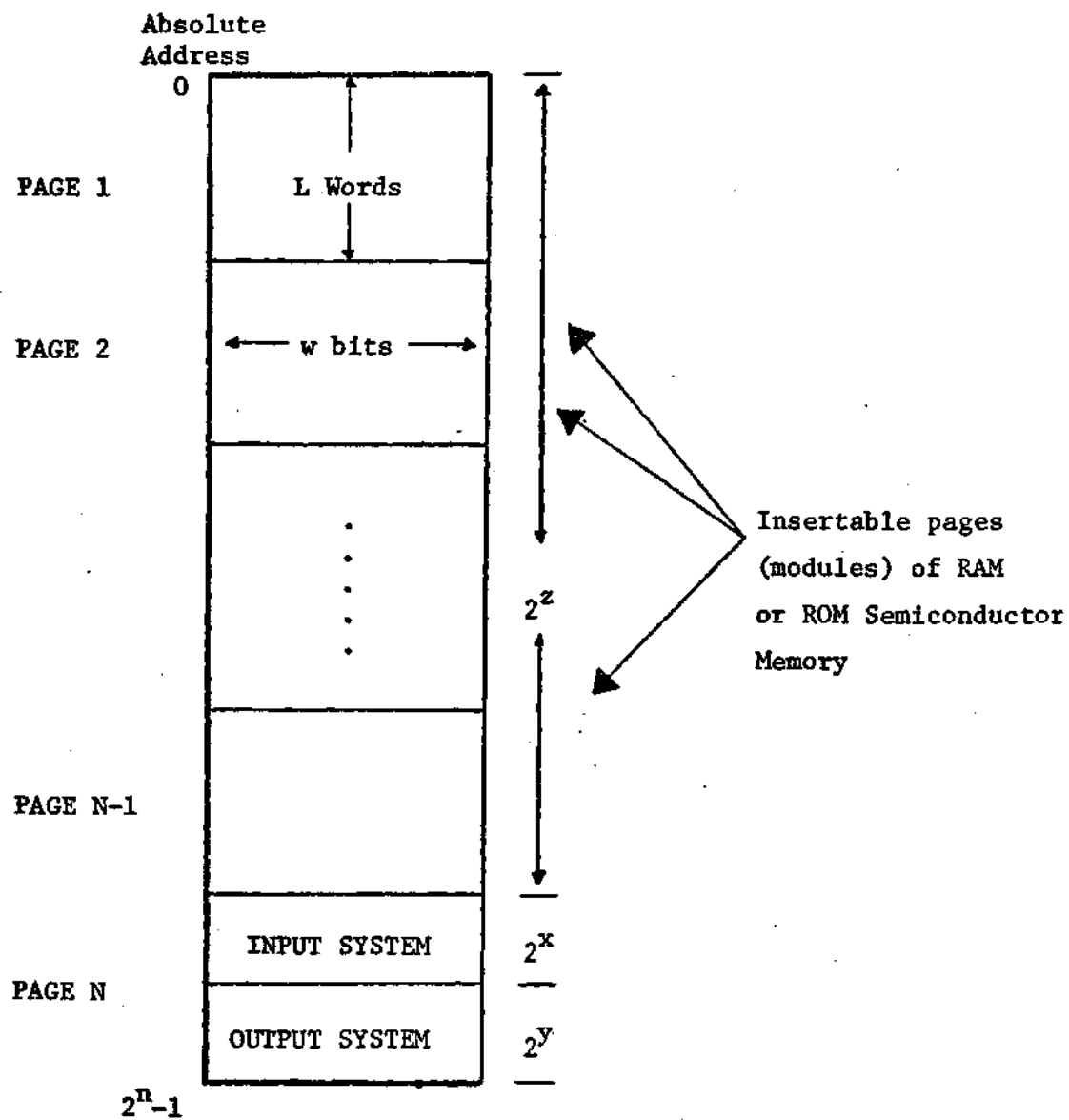


Figure 8. Partitioning of CPU Address Space

register with the contents of an input system address, the register would be loaded with a  $w$  bit word from the input system in which the bit that was originally addressed is located in a predetermined position within the word. The mechanics of the situation will be discussed in the next section. A bit addressed input and output system is necessary to meet the requirements set for these systems in Chapter II. It must be possible to deal with these systems on either a bit or byte basis.

In light of the preceeding discussion, the following relation must hold.

$$2^n \geq 2^x + 2^y + 2^z \quad x, y, z \text{ integers} \quad (1)$$

where

$$\text{Input Address space, } I = 2^x \quad (2)$$

$$\text{Output Address space, } O = 2^y \quad (3)$$

$$\text{Memory Address space, } M = 2^z \quad (4)$$

This means that the input system will be able to handle a maximum of  $I$  input lines and the output system a maximum of  $O$  output lines. The memory system will be able to accomodate a total of  $M$  words of  $w$  bits each.

The choice of page size  $L$  is governed by two basic factors. On one hand, the value of  $L$  should not be so small as to force the number of pages  $N$  to an excessively large number. Restraints on the physical container size of the Level (1) memory system as well as economic factors regarding the number of memory modules in the system dictate that  $N$  be

less than some least upper bound. On the other hand, increasing values of  $L$  tend to lead toward a large unused memory fragment within the last memory module as well as unused address space within the page assigned to the input and output systems. The value of  $L$  should be an integer power of two to aid in address decoding.

### Input System

Basically the input system is composed of an input bank to which the various input lines from the plant floor or other sources are connected, as shown in Figure 9. Logic level inputs are received at the input ports on the bank. Each port can accept  $w$  input signals.

Each input line of each port is addressable by the CPU address register via the address bus. The lower order  $q$  bits of the address specify the particular input line within the port and the remaining higher order bits select the correct port, with  $q$  given in equation five,

$$q = \lceil \log_2(w) \rceil \quad (5)$$

where  $\lceil x \rceil$  denotes the least integer greater than  $x$ . Both bit and byte input addressing is afforded using this scheme. During an input system read operation all  $w$  bits of the addressed input port are transferred to the CPU. Further, if the condition of a single input line is needed, the bit corresponding to that line may be tested by the CPU using a software or firmware routine. The particular bit within the  $w$  bit word is addressed by the low order  $q$  bits of the address register. A maximum of  $I_p = 2^{(x-q)}$  ports may be accommodated by the input system.

Physically, the input system could be constructed using printed circuit cards which plug into the I/O bus. Each card could contain

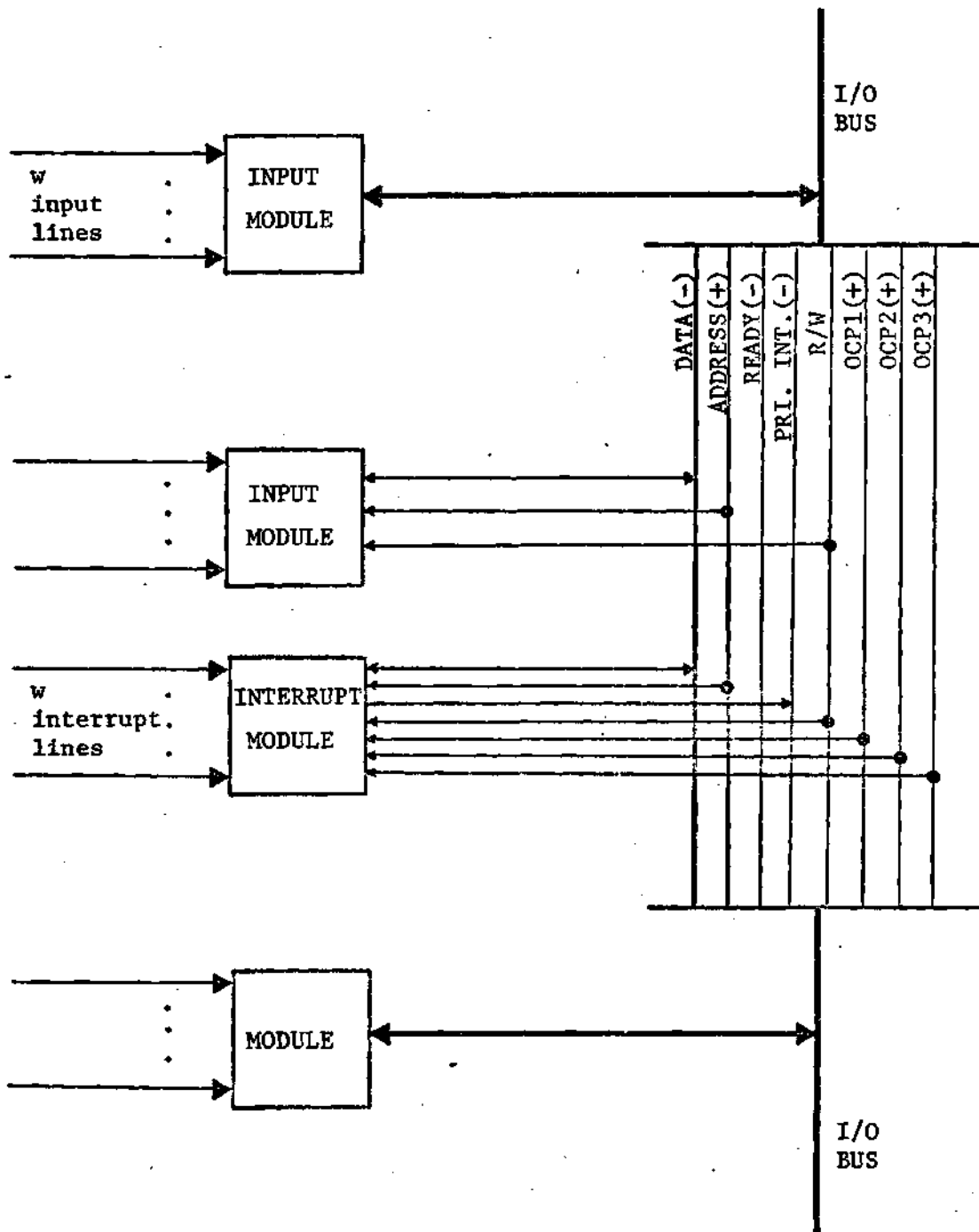


Figure 9. Block Diagram of Input System



several input ports if necessary due to space limitations. Appendix A contains a more detailed discussion of one possible structure of the input port module.

Special purpose input modules may be incorporated into this input system. Modules containing A/D converters and interrupt circuitry are two that immediately suggest themselves.

Interrupt Modules. The need for an expandable priority interrupt structure has already been established. This requirement may be met by using modified input modules to receive incoming interrupt signals. The interrupt module must be capable of accepting either pulse or level interrupt signals. Each interrupt module is plugged into the I/O bus like a standard input module as shown in Figure 9. Internal mask circuitry is contained in each module and mask buffers are set on either a bit or byte basis. Each module contains R interrupt inputs up to a maximum of W inputs, each distinctly addressable.

Four basic operations control the interrupt module. First, a read interrupt buffer (RIB) operation transfers the state of the interrupt flip flops within the interrupt module to the CPU via the data bus. The specified interrupt module is addressed by the high order (n-q) bits of the address bus. In the RIB operation, the low order q bits of the address are meaningless to the interrupt module but can be used by the CPU in identifying the interrupting device. Second, a set mask (SMASK) operation sets the state of the mask flip flops to the state contained on the data bus. Again, only the high order (n-q) bits of the address bus are used to decode the correct module. The third and fourth operations are set individual interrupt mask (SIIM) and reset individual interrupt

mask (RIIM). In each of these operations an individual flip flop and mask associated with an interrupt input are addressed by the address bus. The SIIM operation enables the specified interrupt line while the RIIM operation disables the specified individual interrupt line and resets its interrupt flip flop. The application of an interrupt signal to any enabled interrupt line both sets that interrupt flip flop and signals the CPU via the primary interrupt line that an interrupt has occurred. Logic requirements and timing considerations for the interrupt module are discussed in Appendix B.

An interrupt structure like the one described above is needed to meet the requirements placed on interrupt handling set forth in Chapter II. Enabling and disabling of interrupt lines is accomplished by the SIIM and RIIM operations. Each operation could correspond to a machine language instruction. Interrupts are stored (until they are serviced) within the interrupt flip flops contained in each interrupt module. The state of the interrupt flip flops of any module can be read by the CPU via the RIB operation which is simply a memory read operation with the address of an interrupt module. The SMSK operation corresponds to a memory write operation in which the mask flip flops of an addressed interrupt module may be written into as a group.

### Output System

The output system provides the link for data output from the CPU to the external world, that is, the plant. A diagram of the overall output system is given in Figure 10. System structure is similar to that of the input system. Output modules containing  $w$  output lines per module are plugged into the I/O bus. Individual modules are addressed

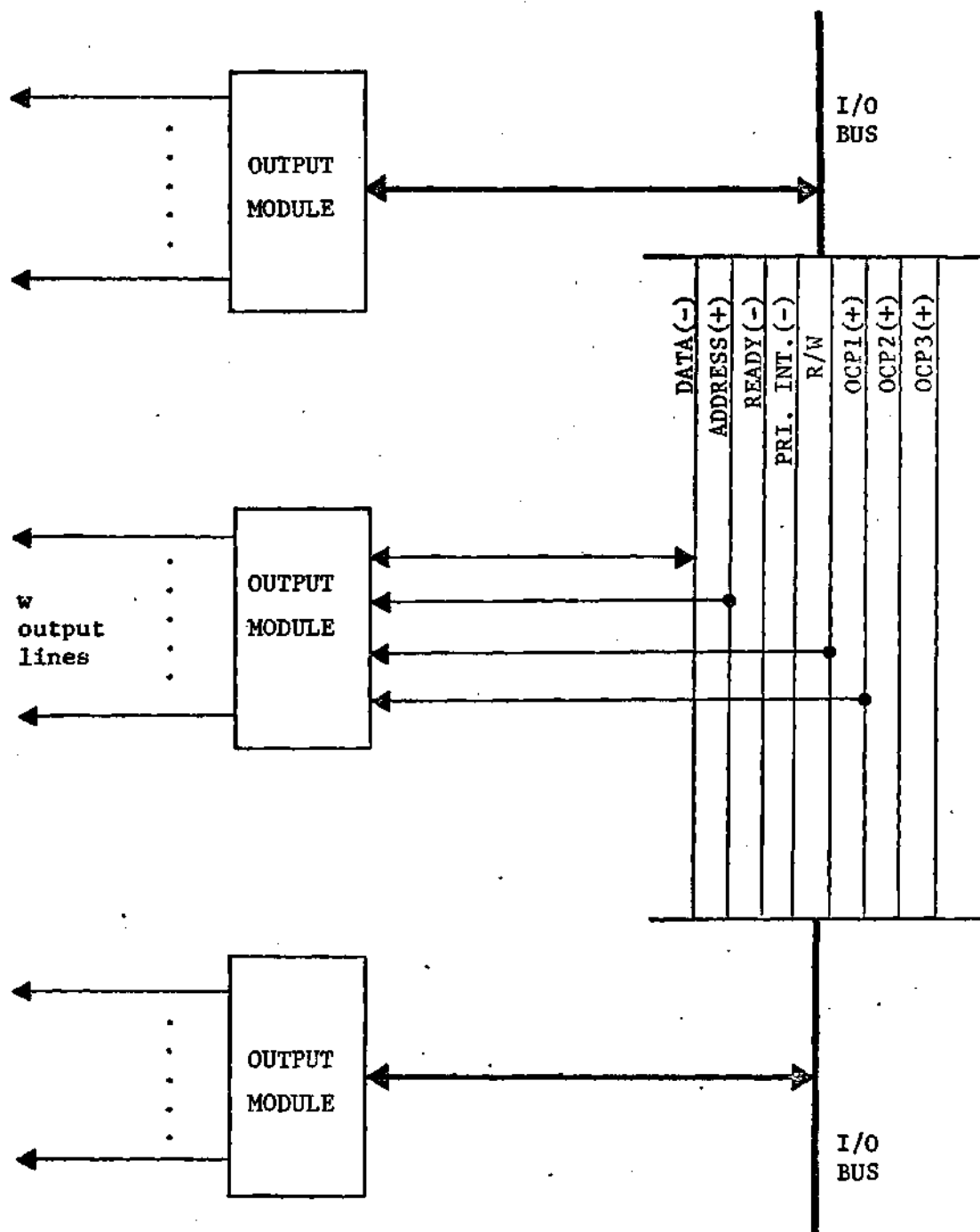


Figure 10. Output System Block Diagram

by the high order  $(n-q)$  bits of the address word. The low order  $q$  bits of the address specify a particular output line within the module. A maximum of  $O_p = 2^{(y-q)}$  output modules may be contained within the output system.

Conceptually, the output system may be thought of as a random access memory composed of  $w$  bit words. Each word corresponds to an output module. Furthermore, all bits of the memory are simultaneously externally accessible through the output lines. This architecture results in a modular structure that is both bit and byte oriented.

Two basic output system operations are necessary using this structure. The write full word (WFW) operation writes a  $w$  bit word from the data bus into the output flip flops driving the addressed output port. The read full word (RFW) operation transfers the state of the addressed output module to the CPU via the data bus. In order to set the state of a single output line, it is necessary to perform a RFW operation on the module housing that line. The CPU then sets the state of the correct bit within the output word using the low order  $q$  bits of the address register to identify the correct bit. The output word is sent back to the output module via the WFW operation and the sequence is completed.

A discussion of the logic implementation of the output module may be found in Appendix C.

#### Memory System

Figures 7 and 8 show that the memory system may be composed of  $(N-1)$  memory modules. Each module has an address space of  $L$  words composed of  $w$  bits per word. Modules may be made up of either core or RAM

or ROM type semiconductor memories. Core memories have the advantage of non-volatility which could prove helpful in case of a power failure, but due to the declining cost per bit of semiconductor memories these will probably be used in the majority of Level (1) memory systems. Those segments of the operating system that are permanent and re-entrant could be stored in ROM for security. Permanent process control programs could also be kept in ROM. Core or RAM semiconductor memories could be used for scratch pad applications and must be used in the memory module shown connected to the multiplex (MPX) circuit in Figure 7. Read/write memory must be used in this transfer module because it is the two way communication link between Level (1) and Level (2). During an information transfer from Level (1) to Level (2), the Level (1) CPU writes the information into the transfer module where it is subsequently read by the Level (2) supervisory computer. The procedure is reversed for Level (2) to Level (1) communication.

Figure 11 gives a block diagram of the memory system. The function of the data and address busses is the same as in the input and output systems. The R/W control line indicates whether a read or write cycle is about to be initiated and the OCPI control line is used to indicate to the memory during a write cycle that data is stable on the data bus. The Ready line is a wired-OR common connection among the memory modules used to indicate module status to the CPU. This line allows slow cycle time memories to be used in the system by synchronizing the CPU to the memory. Appendix D gives a further discussion of memory module hardware and timing requirements.

Multiplex Circuit. The multiplex circuit connecting the memory

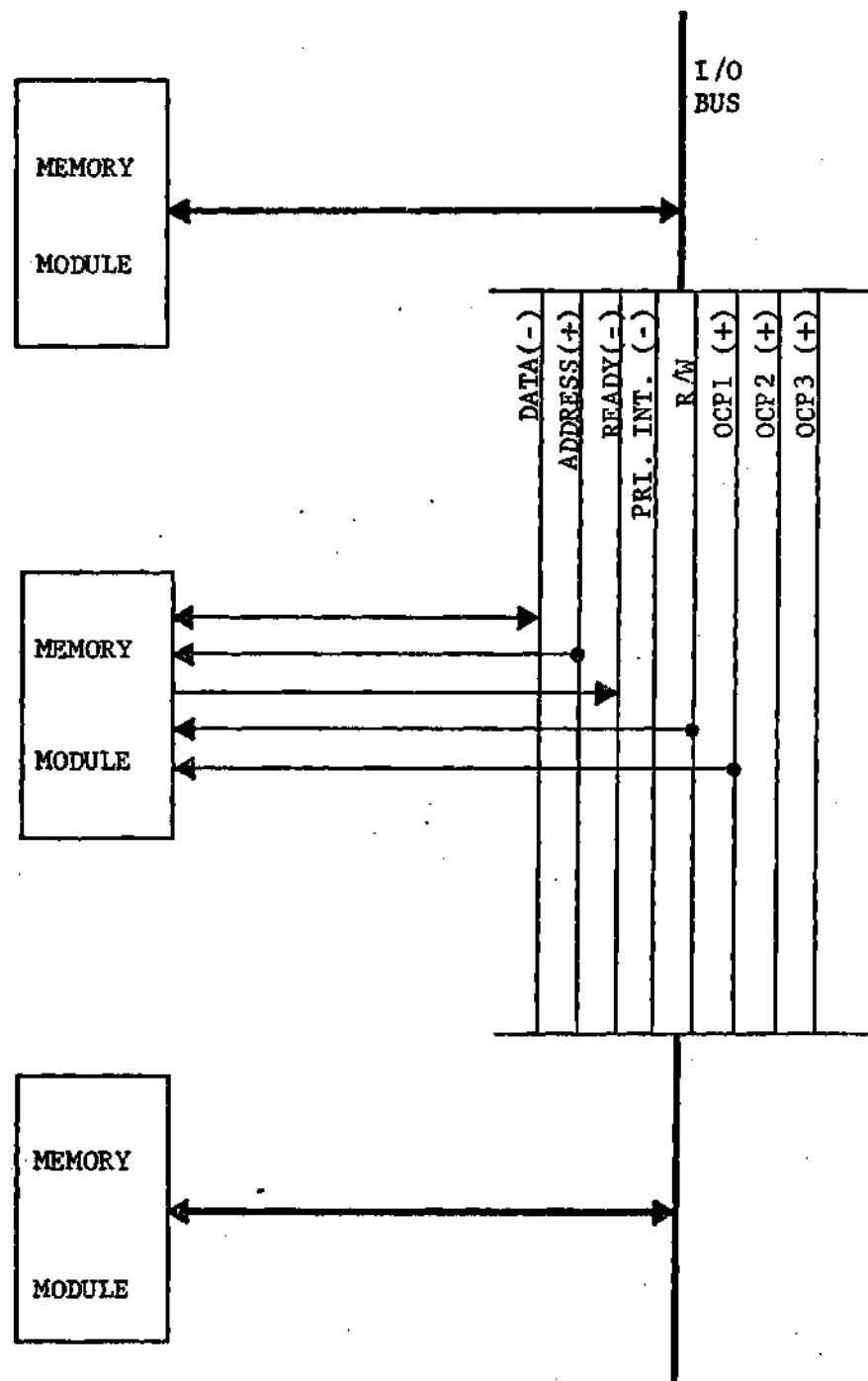


Figure 11. Memory System Block Diagram

module to the I/O bus in Figure 7 serves the purpose of a multi-pole double throw switch. Depending on the state of the output line coming from the output system, the memory module is connected to either the Level (1) I/O bus or an interface to the Level (2) computer system. In this way, verticle communication to and from the Level (2) computer system is possible. Yet, while the actual information transfer is taking place, the Level (1) CPU is free to execute its process control program from any other memory module in the Level (1) system. Allocating and deallocating the transfer memory module is accomplished by the Level (1) CPU through interrupt routines. The normal state of the multiplex circuit is with the transfer module connected to the I/O bus. Figure 12 gives a flow chart of the steps necessary for any information transfers. The format of the transferred information could be either executable statements or binary data.

At this point it may be argued that the multiplex scheme is just a cumbersome method of implementing a dual port memory, so why not use a dual port memory instead? The multiplex scheme is best for several reasons.

First, the cost of building dual port memory modules would be greater than that of single port modules. There is no need for the entire memory system to be dual port. The only time that the Level (2) computer would need to modify the contents of an entire Level (1) memory system would be during a setup operation when new control programs were being sent to the Level (1) memory. During setup, time is not a critical element and the Level (1) CPU is free to move programs and data through the transfer module. Still, it might be argued that the transfer module

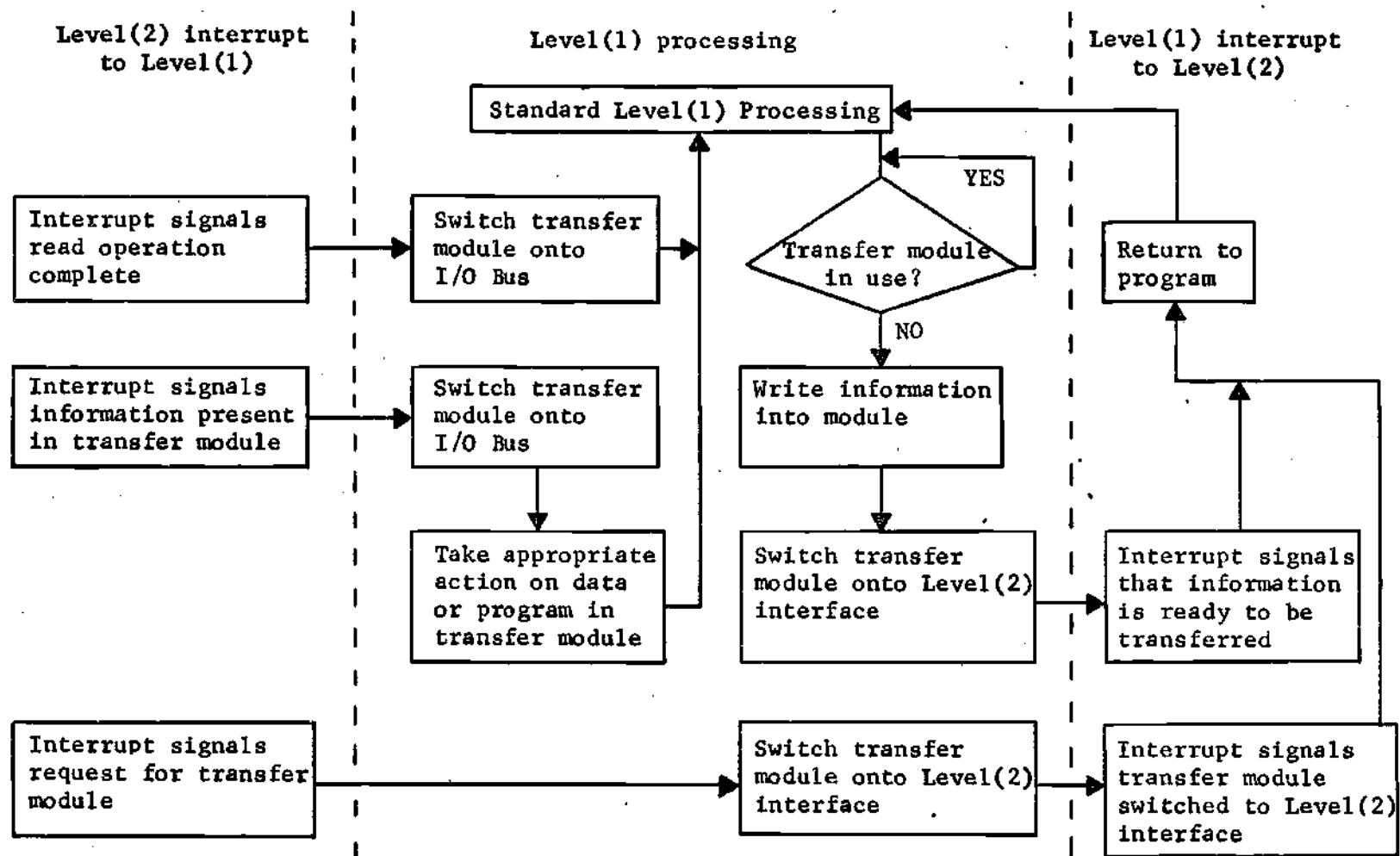


Figure 12. Flowchart for Vertical Information Transfer



should be dual port. But this would still require that two separate types of memory modules be built. The multiplex scheme only requires the single port memory module which is used throughout all Level (1) systems. Controlling the state of the multiplex circuit with an output line from an output module provides the CPU with a convenient means of controlling the switching of the transfer module.

The second reason for favoring the multiplex scheme stems from an observation that if dual port memory modules were used in the memory system, then possible memory conflicts could result [23]. Essentially, the problem is that of two processors sharing a common memory. The dual port approach would offer a costly flexibility that is not essential in this application, while the simpler multiplexing of a single port transfer module is economically beneficial and adequate for this application.

#### Summary

In the preceeding sections an architecture for the input, output and memory systems has been proposed. This architecture was developed under two primary constraints. First, it should be modular so that each Level (1) computer system can be built up to the level required by its unique control situation. Secondly, the architecture should support the system requirements identified in Chapter II. The major requirement affecting the design of the input and output systems was the need for a bit and byte oriented system. It is felt that the proposed architecture meets these requirements in a manner that affords easy interface with a microprocessor CPU.

### Level (1) Microprocessor Architecture

In this section, the architecture of an eight bit microprogrammable microprocessor called the GT 1248 will be described. A machine width of eight bits has been chosen so that a valid comparison with the performance of the eight bit Intel 8080 can later be made. In Chapter II the requirements for the overall Level (1) system were set forth. Table 4 compares the projected capabilities of the GT 1248 with those of the Intel 8080 in satisfying the major requirements imposed upon the Level (1) processor.

Table 4. Comparison of Microprocessor Capabilities

Operation	GT 1248	Intel 8080
Bit manipulation at the Register level.	Available through firmware and supported in the instruction set.	Available indirectly through software routines.
Memory operations at the bit level.	Supported in the instruction set.	Available indirectly through software routines.
Double word operations such as load, store, compare.	Supported in the instruction set.	Reduced capability in the instruction set.
Single word operations	Supported in the instruction set.	Supported in the instruction set.
Special purpose instructions	Available as machine language instructions through microprogramming.	Available through software routines.

### GT 1248 Microprocessor CPU Architecture

Figure 13 gives the internal CPU architecture of the GT 1248.

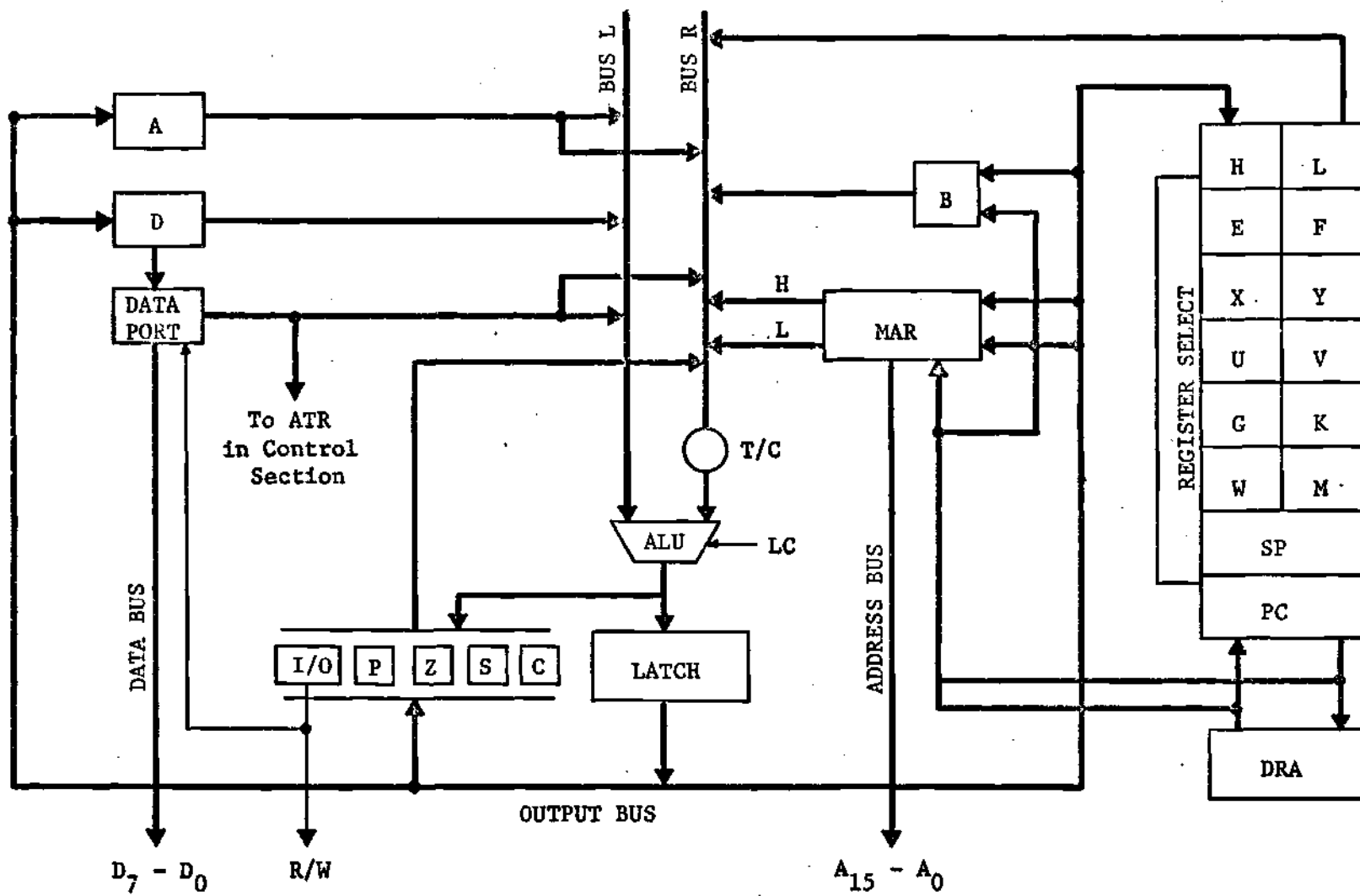


Figure 13. Microprocessor CPU Architecture

Three major internal eight bit busses route data between the arithmetic and logic unit (ALU) and the data registers. Data is transferred to and from the CPU via data bus lines  $D_7 - D_0$  connecting the data port (DP) to external system components. Depending on the state of the I/O flip flop, data is routed from the data bus to Bus R (input mode) or from register D to the data bus (output mode). The address for all memory, input or output system references is carried from the memory address register (MAR) to external components via the address bus,  $A_{15} - A_0$ . The function of the double register arithmetic (DRA) unit is to provide increment, decrement and complement operations on 16 bit double words from the register stack.

CPU Registers. Connected to Bus L are the A register, D register and the data port (DP). Each register is eight bits wide and data flow into and out of all registers is under the control of the microprogram control section. Bit manipulation is accomplished in the A register. Bit address within the A register can be supplied either by the B register or by the microprogram control store. The D register is used for data output in conjunction with the data port and may also be used as a temporary store for intermediate results.

Connected to Bus R are the register stack, the B and A registers, the memory address register (MAR) and the DP. The register stack is made up of 16 eight bit registers. These registers may be accessed as either a single eight bit register via connections to Bus R and the output bus or as a double register composed of 16 bits via connections to the DRA and MAR. Double registers HL, EF, XY, UV and GK are available at the machine language level as general purpose registers. The WM

register is reserved for control section use to store intermediate results. The stack pointer (SP) and program counter (PC) are also available at the machine language level but are reserved for their dedicated functions. The memory address register (MAR) is a 16 bit register made up of two concatenated eight bit registers. The register containing the high order bits,  $A_{15} - A_8$ , of the address is referred to as register MARH. Likewise, the low order bits of the address,  $A_7 - A_0$ , are stored in the lower register, MARL. Register B is a three bit special register used to implement addressing for bit manipulation. Only the low order three bits of data from the output bus are loaded into B during its register load operation. When B is gated onto Bus R, high order bits are gated as logic zeroes while the three low order bits come from B. Register B is always loaded whenever register MARL is loaded from the output bus or whenever the MAR is loaded from the DRA or stack. In this manner B always contains the three low order bits of any MAR address.

ALU Components. Associated with the arithmetic and logic unit (ALU) are: a true/complement (T/C) gate array for Bus R, a group of five flag flip flops and an output latch. The T/C array can be used to gate into the ALU the complement of the data contained on Bus R. This is done under microprogram control and is useful in performing one's and two's complement subtraction schemes.

The functions of the five flag flip flops comprising the processor state word (PSW) are given in Table 5. The I/O flip flop is under microprogram control exclusively, while flags P, Z, S and C are loaded during certain ALU operations. The five flags may be thought of

Table 5. Flag Flip Flop Functions

Flip Flop	Function
I/O	Determines the state of the data port. Logic 1 = data port in the input mode. Logic 0 = data port in the output mode. Normally this flip flop is kept in the logic 0 state.
P	Reflects whether the last word gated from the ALU contained even or odd parity. P = 1 denotes odd parity, P = 0 denotes even parity.
S	Reflects whether the last word gated from the ALU was positive or negative depending on the state of the MSB. S = 1 indicates that the word was negative. S = 0 indicates that the word was positive.
Z	Reflects whether the last word gated from the ALU was equal to zero. Z = 0 denotes that the last word was nonzero. Z = 1 denotes that the word was zero.
C	During certain ALU add operations, the carry generated from the MSB additions is gated into C. During a right or left rotate, the bit rotated is loaded into C.

as a separate register that may be gated onto Bus R or receive data from the output bus. When gated as a register, the five flags are right justified in the field of the word. This feature permits the flags to be stored in memory prior to interrupt servicing.

Table 6 gives a description of the operation set of the ALU. During an ADD operation, a binary one may be gated into the carry of the l.s.b. addition by applying a logic one to the LC terminal of the ALU. During an ADD (with carry) operation, the LC terminal is connected to the C flip flop, otherwise it is under microprogram control and is useful in two's complement schemes.

The eight bit output of the ALU is gated into a temporary latch that holds the result long enough to allow it to be gated via the output bus into a register.

CPU Timing. Figure 14 shows the two phase clock used to control all register transfers. The same clock is used for control section timing. During time  $t_1$  data is gated onto Busses L and R, through the ALU and into the latch. Time  $t_1$  must be greater than or equal to the worst case propagation delay through the ALU and into the latch. During time  $t_{f1}$  the data present at the latch is locked in, in much the same way as in the TTL 74100 latch [21]. Flag flip flops are gated and locked according to the same timing as the latch.

The contents of the latch are gated into a selected register via the output bus during time  $t_2$ . Time  $t_2$  must be greater than or equal to the worst case propagation delay through the output bus into the register. During time  $t_{f2}$  the contents of the register are locked in, so that CPU registers may be of the same structure as the latch.

Table 6. ALU Operation Set

Operation	Description
ADD	$OUTPUT \leftarrow Bus\ L + Bus\ R$ (arithmetic sum) $(C) \leftarrow$ carry, all flags affected
ADD (with carry)	$OUTPUT \leftarrow Bus\ L + Bus\ R + (C)$ $(C) \leftarrow$ carry, all flags affected
ADD (no flags)	$OUTPUT \leftarrow Bus\ L + Bus\ R$ no flags affected
AND	$OUTPUT \leftarrow Bus\ L \wedge Bus\ R$ $(C)$ unaffected
OR	$OUTPUT \leftarrow Bus\ L \vee Bus\ R$ $(C)$ unaffected
EXCLUSIVE OR	$OUTPUT \leftarrow Bus\ L \nabla Bus\ R$ $(C)$ unaffected
ROTATE L	$OUTPUT \leftarrow$ rotate left $(Bus\ L + Bus\ R)$ $(C) \leftarrow$ m.s.b. before rotate operation
ROTATE R	$OUTPUT \leftarrow$ rotate right $(Bus\ L + Bus\ R)$ $(C) \leftarrow$ l.s.b. before rotate operation



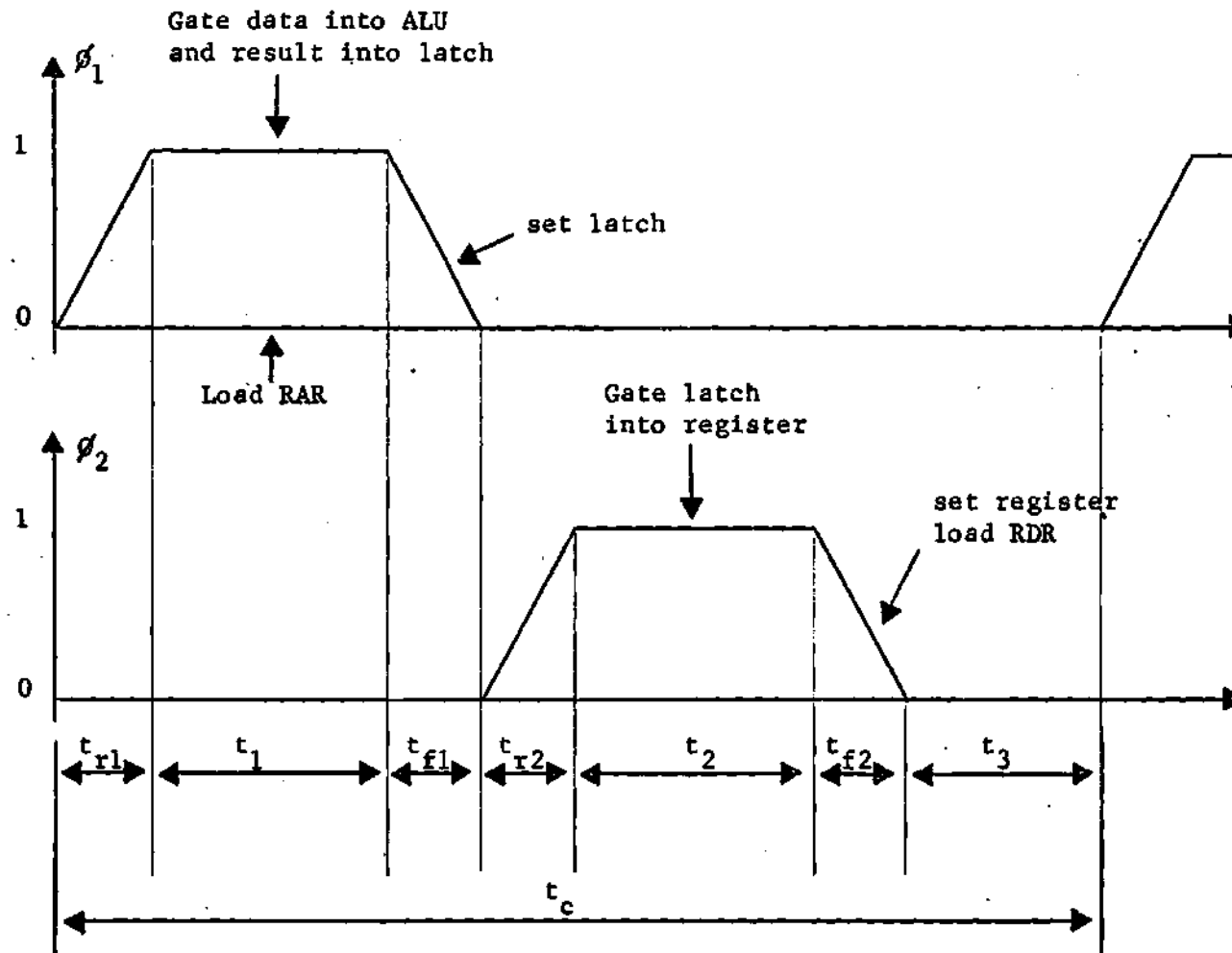


Figure 14. Two Phase Clock Used For System Timing

Registers whose data transfer is accomplished on the positive edge of the clock cannot be used with this timing sequence. Time  $t_3$  is used to allow the decoder signals to settle prior to  $\phi_1$  of the system clock.

#### GT 1248 Control Section Architecture

Figure 15 gives the basic architecture of the control section. The 13 bit ROM address register (RAR) contains the address of the next microinstruction to be executed and is similar in function to the standard program counter. Normal control section operation assumes that microinstructions are fetched sequentially from the ROM, so it is the function of the increment logic (INC) to increment the RAR by one during each microinstruction cycle [22]. Microinstruction words fetched from the ROM control store are stored in the 20 bit R register. The various decoders are attached to the R register so that the proper control signals can be generated from the current microinstruction word. An interrupt flip flop with its associated mask is provided to receive the primary interrupt signal. The OP1 control line is driven by the OP1 flip flop and is used to provide a level control signal to external circuits. Similarly, the OP2 line is used to provide a control pulse to external circuits. Used in conjunction with these two control lines are bits 13 - 0 of the microcode word (see Figure 18) which are available externally at the ROM to provide an address for the multiplexing of OP1 and OP2.

The interrupt test flip flop (I) can be set to the one state during an unconditional microprogram jump. If  $(I) \wedge (PRI\ INT) = 1$ , then instead of the RAR being loaded with the jump address, the RAR is cleared and an interrupt microprogram routine entered at control store location

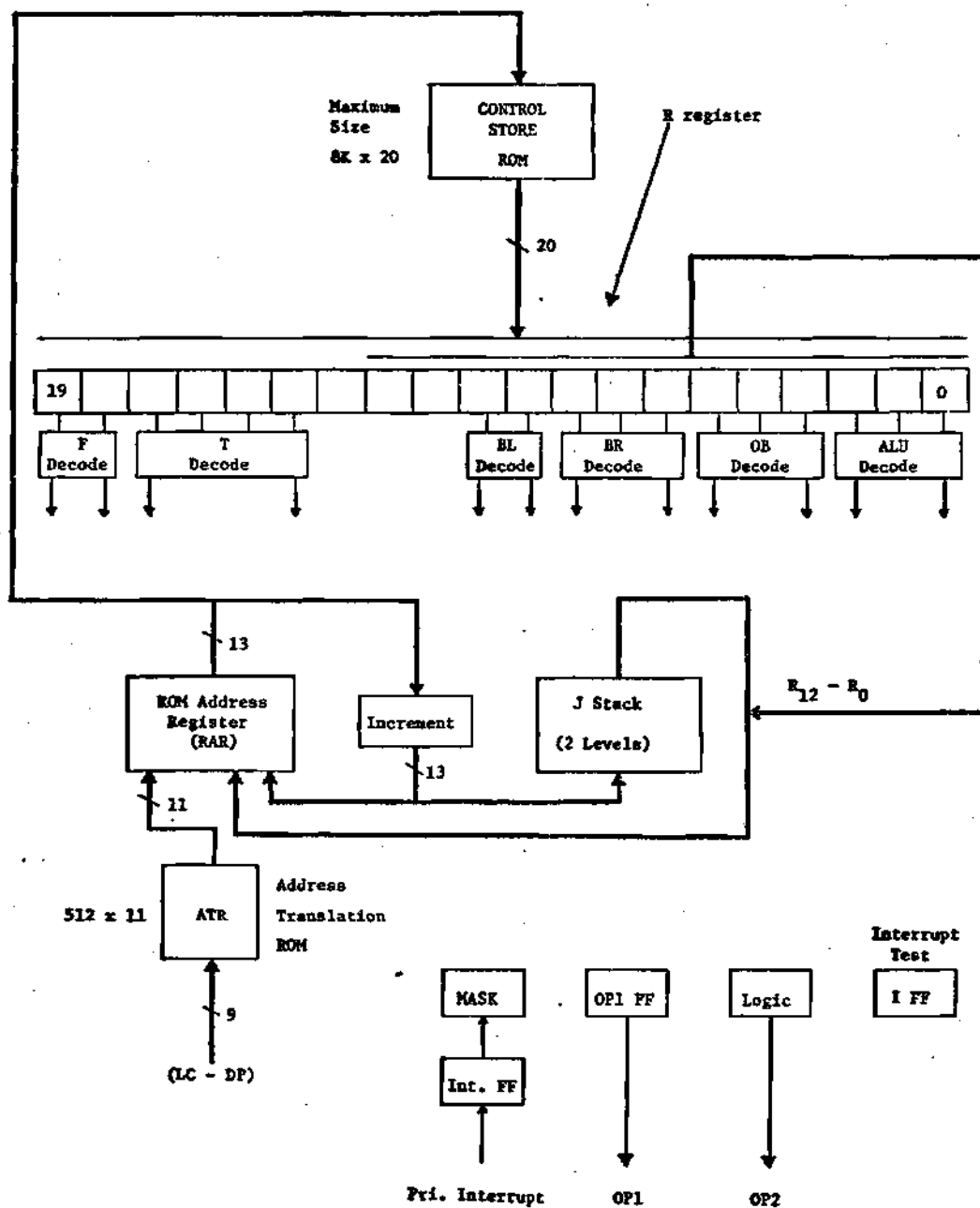


Figure 15. Control Section Architecture

zero. This interrupt detection and vectoring will be discussed more fully under the section on the jump microinstruction.

Up to two levels of microprogram subroutines can be handled with the help of the two level first in last out J stack. The address translation ROM (ATR) is a 512 word by 11 bit read only memory contained within the control section used to convert the eight bit op code used in the GT 1248 to a ROM control store segment address. Each segment contains the microprogram routine for a single machine language instruction. A more detailed discussion of the ATR will be presented later.

Control Timing. Referring to Figure 14, it is seen that during time  $t_1$ , the ROM address of the next microinstruction is loaded into the RAR. ROM's are static devices so that within the access time after the RAR is loaded and settled, the next microinstruction will appear at the output port of the control store ROM. During time  $t_{f2}$ , the next microinstruction is loaded into the R register, dictating that register R input a new microcode word on the negative edge of its gating or clock signal. Each instruction in the microprogram corresponds to a single microcode word fetched from the control ROM. The clock time,  $t_c$ , used to execute one microinstruction is referred to as one machine state or cycle.

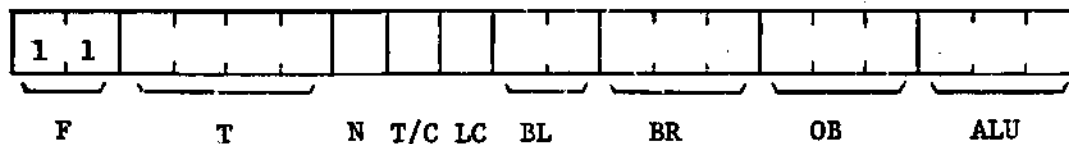
If the control section is implemented on a separate chip from the CPU, then the microinstruction word will have to be decoded both on the control chip and on the CPU chip. It is felt that this approach would lead to an unnecessary duplication of logic. Most current microprocessors have both their CPU and control section integrated on a single chip. The elimination of the large control store ROM from the chip, as

is the case with the GT 1248, will free needed chip area for the GT 1248's large register stack, while at the same time leaving room for the circuitry of Figure 15. In an effort to reduce the pin count of the GT 1248, control store address and data could be multiplexed through a single 20 pin ROM port synchronized by the system clock. This approach would yield a microprocessor pin count of about 51, which is only 11 above that of the Intel 8080.

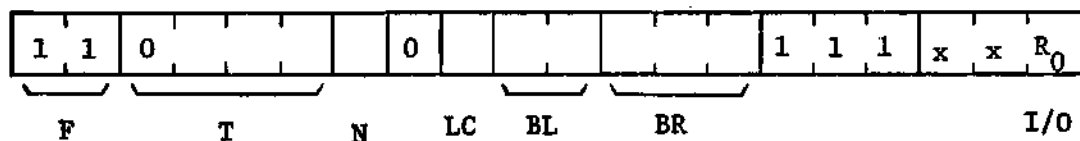
The GT 1248 microprogrammable control section recognizes six basic types of microcode words: register control, double register, fetch, jump, pulse and emit.

Register Control Instruction. The register control instruction is used to control the ALU and route eight bit data through the CPU. Figure 16 gives the format of this microcode word. The T field is used to select one of 16 eight bit registers from the register stack. The T/C field is used to control the T/C gate array on Bus R. If  $(T/C = 1)$  then  $(ALU \leftarrow Bus R)$ . The N field is used to control the I/O and OPl flip flops which are used to synchronize the CPU to the memory during a read or write cycle. If  $(OB = 010)$  then  $((OPl) \leftarrow N \text{ during } t_2)$  else  $((I/O) \leftarrow N \text{ during } t_2)$ . The LC field is used to control the LC terminal on the ALU. If  $(LC = 1)$  then (carry into l.s.b. addition) else (no carry into l.s.b. addition). The BL field controls register gating onto Bus L as defined in Table 7. The BR field controls register gating onto Bus R. Register gating from the output bus into CPU registers is controlled by the OB field. The ALU field selects the ALU function for the current microinstruction cycle. The function codes for these fields are given in Table 7. The mnemonic for the register control microinstruction is

## REGISTER CONTROL INSTRUCTION



## DOUBLE REGISTER INSTRUCTION



## FETCH INSTRUCTION

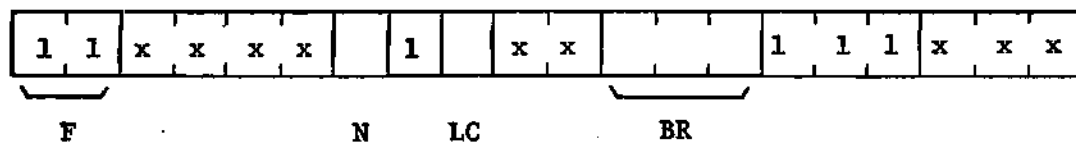


Figure 16. Format for the RC, DR and Fetch Microinstructions

Table 7. Field Specification for the Register Control Instruction

Field	BL	BR	OB	ALU
0 0 0	zero	zero	A	ADD
0 0 1	D	stack	stack	ADD (with carry)
0 1 0	A	A	D	ADD (no flags)
0 1 1	DP	B	B	AND
1 0 0	X	MARL	MARL	OR
1 0 1	X	MARH	MARH	EX-OR
1 1 0	X	PSW	PSW	ROTATE L
1 1 1	X	DP	Restricted	ROTATE R

as follows:

#### RC; Operational Function

For example, the mnemonic  $RC; (A) \leftarrow (\overline{DP}) + (A) + 1, (I/O) \leftarrow N$  indicates that register A is to be loaded with the results of the addition, which is actually a two's complement subtraction of  $(A) - (DP)$ . The I/O flip flop is loaded with the contents of the N field. If the I/O specification is omitted, then by default  $(I/O) \leftarrow 0$ .

Double Register Instruction. This microinstruction is used to control all 16 bit data transfers and arithmetic operations done in the double register arithmetic (DRA) unit. The format of this instruction is given in Figure 16. Selection of one of eight double registers to output from the stack into the DRA during  $t_1$  is accomplished by the T

field. Selection of one of eight double registers to receive data from the DRA during time  $t_2$  is accomplished by the BR field.

The N and LC fields are used to control loading of the MAR in parallel with a DRA operation. IF (LC = 1) then ((MAR) ← data selected by the N field) else (MAR is not loaded). If (N = 0) then (data = DRA input) else (data = DRA output). The BL field controls the function of the DRA as shown in Table 8. During time  $t_2$ , the I/O flip flop is loaded with the contents of the low order bit,  $R_0$ , of the R register.

Table 8. Double Register Field Specifications

Field	T or BR	BL
0 0 0 0	(H)(L)	Increment
0 0 0 1	(E)(F)	Decrement
0 0 1 0	(X)(Y)	Complement
0 0 1 1	(U)(V)	No Change
0 1 0 0	(G)(K)	X
0 1 0 1	(W)(M)	X
0 1 1 0	(SP)	X
0 1 1 1	(PC)	X

The mnemonic for the double register microinstruction follows.

#### DR; Operational Function

For example, the mnemonic DR; (MAR) ← (PC), (PC) ← (PC) + 1 indicates that during time  $t_1$  the MAR is loaded with the contents of the PC double register and during time  $t_2$  the PC register is incremented. If the I/O



specification is omitted, then  $(I/O) \leftarrow 0$  by default.

Fetch Instruction. Figure 16 gives the format for the fetch instruction. This instruction is used in the fetch sequence to load the RAR with the segment address for the current machine language instruction microprogram routine. During time  $t_2$ , the I/O flip flop is loaded with the contents of the N field. The LC field is used to denote the type of fetch instruction being executed as shown in Table 9.

Table 9. Fetch Instruction Functions

Mnemonic	LC Field	Function
Fetch S	0	$(RAR) \leftarrow ATR(0-DP), (I/O) \leftarrow N$
Fetch D	1	$(RAR) \leftarrow ATR(1-DP), (I/O) \leftarrow N$

The operation of Fetch S is as follows. During time  $t_1$  the op code, which is present at the data port (DP), is applied to the lower eight bits of the address translation ROM (ATR) while the contents of the LC field is applied to the high order input of the ATR (see Figure 15). The segment address corresponding to that op code is then locked into the RAR during time  $t_{f1}$ . The sequencing of Fetch D is exactly the same except that  $LC = 1$ .

Two types of fetch instructions are used so that the control section of the GT 1248 is capable of decoding a total of 511 op codes as opposed to the 256 found in the Intel 8080. The extra op codes are needed to handle the process control oriented machine language

instructions that will be added to the GT 1248 instruction set. Some of these additions will be discussed in a later section.

Figure 17 gives a pictorial representation of the use of these two fetch commands. Immediately after the current machine language instruction has completed execution, an unconditional microprogram jump statement directs microprogram execution to the "single op code fetch" routine. This routine loads the first eight bit word of op code for the next machine language instruction from the system memory. This op code is applied to the input of the address translation ROM (ATR). The ATR directs microprogram execution to the segment address of the microprogram routine corresponding to that op code unless the first word of op code equals binary zero. In that case, microprogram execution is directed to the "double op code fetch" routine which loads the second word of op code from system memory. This op code is then used in conjunction with the Fetch D microinstruction to direct microprogram execution to the segment address of the microprogram routine corresponding to that double op code. Fetch time for a double op code instruction will be twice as long as for a single op code instruction.

Jump Instruction. The format of the jump microinstruction is in Figure 18. This instruction is used to provide conditional and unconditional jumps within the microprogram. It can also be used to call a microprogram subroutine and return upon completion. As shown in Table 10, the T field determines the type of jump instruction under execution. All actions specified by the various jump commands must take place during the  $\phi_1$  phase of the system clock.

At this point it should be instructive to describe the JMP Write

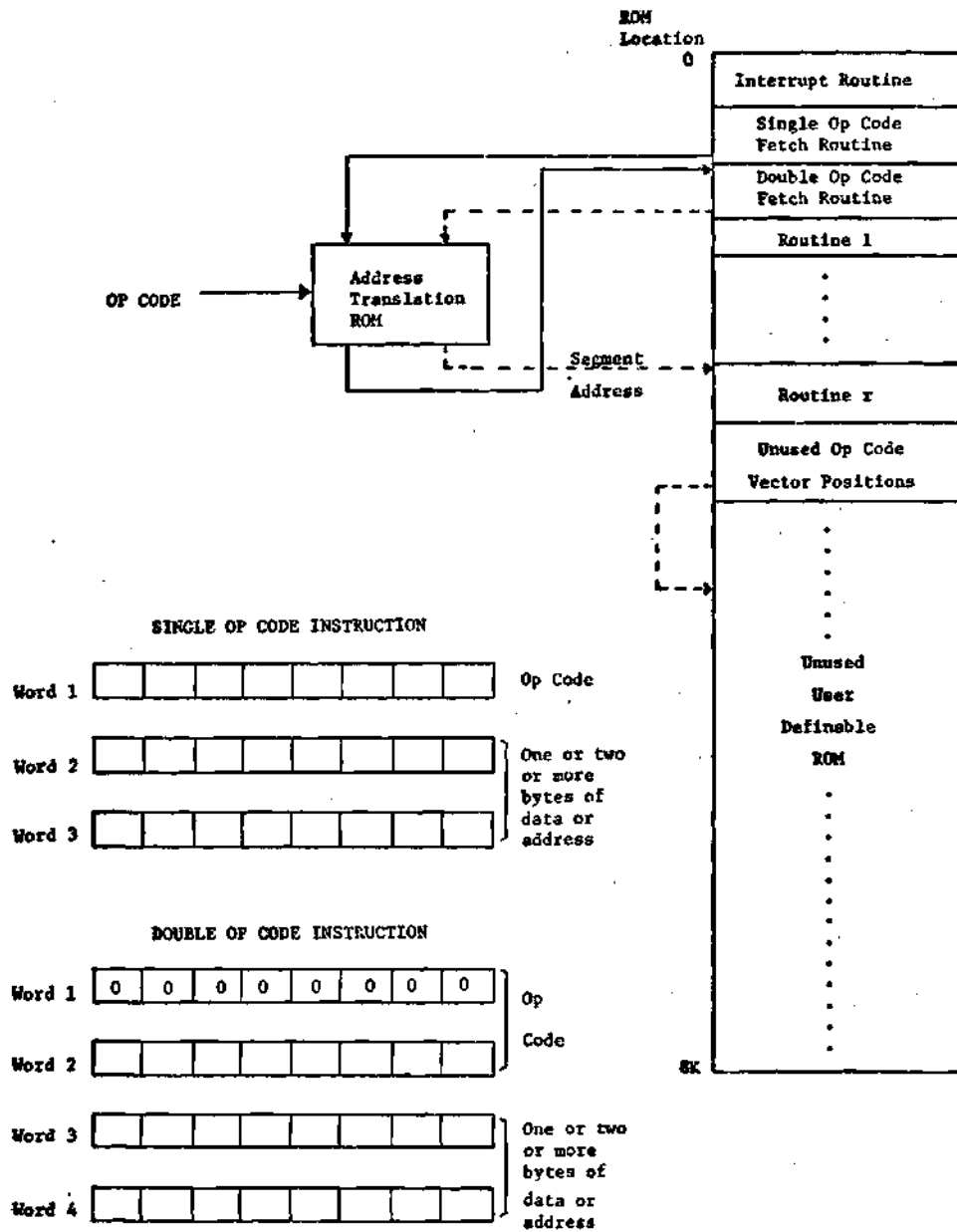
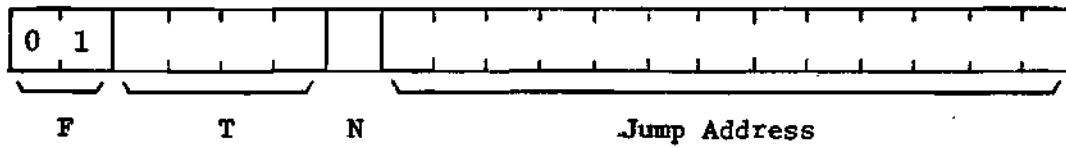
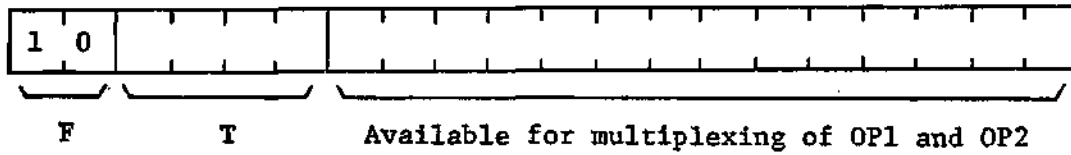


Figure 17. Control Store Address Mapping

## JUMP INSTRUCTION



## PULSE INSTRUCTION



## EMIT INSTRUCTION

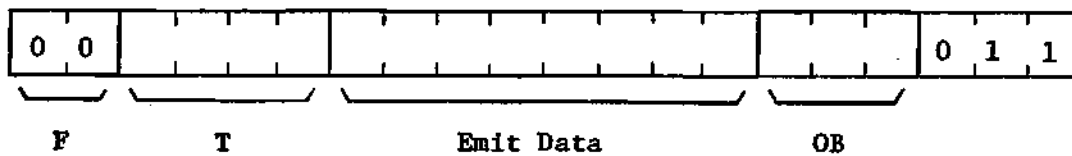


Figure 18. Format for the Jump, Pulse and Emit Microinstructions

Table 10. Jump Instruction Functions

T Field	Mnemonic	Action	Conditional?
0 0 0 0	JMP(N) Address	$(RAR) \leftarrow \text{Address}, (I) \leftarrow N$	No
0 0 0 1	JMP Sub, Address	$(J) \leftarrow (RAR) + 1, (RAR) \leftarrow \text{Address}$	No
0 0 1 0	JMP Return	$(RAR) \leftarrow (J)$	No
0 0 1 1	JMP(N) $A_B$ , Address	If (bit $A_B = N$ ) then $((RAR) \leftarrow \text{Address})$ Else $((RAR) \leftarrow ((RAR) + 1))$	Yes
0 1 0 0	JMP(N) P, Address	If $((P) = N)$ then $((RAR) \leftarrow \text{Address})$	Yes
0 1 0 1	JMP(N) Z, Address	If $((Z) = N)$ then $((RAR) \leftarrow \text{Address})$	Yes
0 1 1 0	JMP(N) S, Address	If $((S) = N)$ then $((RAR) \leftarrow \text{Address})$	Yes
0 1 1 1	JMP(N) C, Address	If $((C) = N)$ then $((RAR) \leftarrow \text{Address})$	Yes
1 0 0 0	JMP(N) PI, Address	If (Primary Interrupt = W) then $((RAR) \leftarrow \text{Address})$	Yes
1 0 0 1	JMP(N) Ready, Address	If (Ready = N) then $((RAR) \leftarrow \text{Address})$	Yes
1 0 1 0	JMP Write, Address	If (Ready = 0) then $((RAR) \leftarrow \text{Address})$ Else $((RAR) \leftarrow (RAR) + 1, (OPl) \leftarrow 0)$	Yes
1 0 1 1	JMP Read, Address	$(I/O) \leftarrow 1$ , If (Ready = 0) then $((RAR) \leftarrow \text{Address})$ Else $((RAR) \leftarrow (RAR) + 1)$	Yes
1 1 0 0	unused		
1 1 1 1	unused		

and JMP Read instructions as they apply to the memory read and write cycles. The JMP Read microinstruction is used in the following manner during a memory read.

DR; (MAR)  $\leftarrow$  (PC), (PC)  $\leftarrow$  (PC) + 1, (I/O)  $\leftarrow$  1

JMP Read, \*

RC; (A)  $\leftarrow$  (DP), (I/O)  $\leftarrow$  0

This section of microcode loads the A register with the contents of memory specified by the program counter (PC) and increments the PC. During  $\phi_2$  of the double register instruction, the MAR is loaded and the R/W control line goes high indicating to memory to start a read cycle. The "JMP Read, \*" instruction jumps to itself until ready is indicated by the memory. At this point the RC instruction is executed, the A register is loaded and the R/W control line reset.

In order to store register A in the memory location pointed to by the PC, the following code could be used.

DR; (MAR)  $\leftarrow$  (PC)

RC; (D)  $\leftarrow$  (A), (OPl)  $\leftarrow$  1

JMP Write, \*

The RC instruction loads register D and sets OPl to the one state. The OPl control line can be used to drive the OCPl control line to the memory that indicates the start of a write cycle. When the memory indicates the data has been written or accepted, the "JMP Write, \*" instruction

allows the next sequential microinstruction to be executed.

During a "JMP(N) Address" jump instruction, the I flip flop is set to N. The "JMP(1) Fetch" jump instruction is used at the end of each microprogram routine to go back to the fetch routine, at the same time setting  $(I) = 1$  during time  $t_{r1}$ . If during time  $t_1$ ,  $(I) \wedge (PRI. INT) = 1$ , then the RAR is cleared and the next microinstruction to be executed will not be the first line of the fetch routine but the microinstruction at control store location zero. This microinstruction can be either a jump to an interrupt routine or the first line of an interrupt routine segment. During time  $t_2$  the I flip flop is cleared.

Pulse Instruction. The format of the pulse microinstruction is shown in Figure 18. This instruction is basically used by the control section for individual bit manipulation by applying pulses to set, reset and toggle inputs on appropriate flip flops. As shown in Table 11, the T field is used to determine the function of the pulse instruction.

All pulse operations occur during phase  $\phi_2$  of the system clock. Bit positions 13-0 of the microinstruction can be used for multiplexing of control signals OP1 and OP2 since these bits are available at the control store ROM output port during  $\phi_2$ .

Emit Instruction. The format of the emit microinstruction is shown in Figure 18. With this instruction, any CPU register can be loaded with eight bit data stored in the emit microcode word. During time  $t_1$ , the emit data is loaded into the temporary latch instead of data from the ALU. All flags remain unaffected. During the latter half of the system clock cycle the contents of the latch are transferred to the CPU register specified by the OB field, just as in the register

Table 11. Pulse Instruction Functions

Mnemonic	T Field	Function
Pulse; Set $A_B$	0 0 0 0	bit $A_B \leftarrow 1$
Pulse; Reset $A_B$	0 0 0 1	bit $A_B \leftarrow 0$
Pulse; Comp. $A_B$	0 0 1 0	bit $A_B \leftarrow \bar{A}_B$
Pulse; Set $A_{OB}$	0 0 1 1	bit $A_{OB} \leftarrow 1$
Pulse; Reset $A_{OB}$	0 1 0 0	bit $A_{OB} \leftarrow 0$
Pulse; Comp. $A_{OB}$	0 1 0 1	bit $A_{OB} \leftarrow \bar{A}_{OB}$
Pulse; Set I/O	0 1 1 0	(I/O) $\leftarrow 1$
Pulse; Reset I/O	0 1 1 1	(I/O) $\leftarrow 0$
Pulse; EI	1 0 0 0	(MASK) $\leftarrow 1$
Pulse; DI	1 0 0 1	(MASK) $\leftarrow 0$ , (IWT) $\leftarrow 0$
Pulse; Set OP1	1 0 1 0	(OP1) $\leftarrow 1$
Pulse; Reset OP1	1 0 1 1	(OP1) $\leftarrow 0$
Pulse; OP2	1 1 0 0	line OP2 is pulsed during $t_2$
Pulse; Set C	1 1 0 1	(C) $\leftarrow 1$
Pulse; Reset C	1 1 1 0	(C) $\leftarrow 0$
unused	1 1 1 1	



control instruction. The T field is used to select one of 16 stack registers if the stack is being loaded. The mnemonic for this instruction is as follows.

EMIT; (r) ← Quantity

For example, the instruction EMIT; (X) ← 372<sub>8</sub> means that the X register is to be loaded with an octal 372.

Control ROM Partitioning. A detailed analysis of the instruction set necessary for a process control microprocessor is not the subject of this paper, but in any architectural consideration the size of an expected instruction set should be approximately known so that the hardware can support it. While an analysis of the basic GT 1248 instruction set will be reserved for a later section, the results of that analysis with respect to size and space requirements will be briefly presented.

The basic GT 1248 instruction set consists of approximately 98 machine language instructions, as found in Appendix E. This instruction set utilizes approximately 316 different op codes, dictating that approximately 61 of these op codes will have to be dual op code instructions. These 316 different op codes require approximately 1324 words of control store to hold the necessary microprogram routines and subroutines along with the two fetch routines.

In Figure 17, the control store address mapping procedure was depicted. The ATR is capable of specifying a segment address within the first 2K of the control store ROM. Within this region the basic instruction set microprograms should be permanently stored. Assuming that the approximately 1270 words required for this storage is reasonable, then

approximately 720 locations are left free within the lower 2K of ROM. The ATR would be permanently programmed to direct unused double op codes to sequential locations within the lower part of this free area. When the user wanted to microprogram a new machine language instruction, he could insert a "JMP(0) Segment Address" instruction in the location corresponding to a chosen double op code. This jump would then vector the control section to the correct user defined microprogram routine in higher ROM.

The permanent section of the control store should be stored in mask programmed ROM for economic and security reasons, while the remainder could be programmable (PROM) or erasable (EPROM) memory for user defined microprograms. The call for a user microprogrammable microprocessor has already been acknowledged in the literature [18]. It has been suggested that microprocessor programming be done more at the microprogram level in an effort to increase efficiency and throughput.

## CHAPTER IV

### COMPARISON AND EVALUATION OF PERFORMANCE

#### Performance Evaluation

Up to this point, the overall architecture of a microprocessor based computer control system for discrete manufacturing control has been presented. In particular, an architecture for the GT 1248 has been presented that was developed with the express purpose of satisfying the microprocessor requirements developed in Chapter II. In this chapter, a comparison will be made between the performance of the GT 1248 and that of the Intel 8080. Performance will be measured on the basis of speed of execution of a given task and the number of words of program needed to accomplish that task, as given in equation six,

$$\text{Performance (P)} = \frac{1}{NT} \quad (6)$$

where T is the execution time for the task and N is the number of eight bit words of source code needed to program the task.

#### Microprocessor Instruction Set

Appendix E gives a detailed description of the basis instruction set of the GT 1248. A condensed summary of the characteristics of this instruction set is given in Table 12, where those instructions that essentially are not found in the instruction set of the Intel 8080 are preceeded by an asterisk. In evaluating the execution speeds of the instructions, it was assumed that each is a single op code instruction

Table 12. Characteristics of the GT 1248 Instruction Set

Mnemonic	Number of Words		Execution Time (US)	Equivalent Intel 8080 Execution Time (US)
	Double Op Code	Single Op Code		
- Single Register and Memory Instructions -				
MOV $r_{1a}, r_{1b}$	2		4-5.5	2.5
MOV M, $r_1$		1	3.5	3.5
MOV $r_1$ , M		1	3.5	3.5
MVI $r_1$		2	3.5	3.5
MVI M		2	5	5
*MVII M		4	8	10 **
INR $r_1$		1	2.5	2.5
DCR $r_1$		1	2.5	2.5
INR M		1	4.5	5
DCR M		1	4.5	5
*INRI M		3	7.5	10 **
*DCRI M		3	7.5	10 **
- Double Data Word Instructions -				
INX dd		1	2.5	2.5
DCX dd		1	2.5	2.5
*INX M		1	7.5	21.5 **
*DCX M		1	7.5	21.5 **
*INXI M		3	10.5	26.5 **
*DCXI M		3	10.5	26.5 **

Table 12. (Continued)

Mnemonic	Number of Words		Execution Time (US)	Equivalent Intel 8080 Execution Time (US)
	Double Op Code	Single Op Code		
LXJ dd		3	5	5
MOVX dd <sub>a</sub> , dd <sub>b</sub>		1	2.5	2.5
STDD dd <sub>1</sub>		3	8	8
LDDD dd <sub>1</sub>		3	8	8
*STID dd <sub>2</sub>		1	5	12 **
*LDID dd <sub>2</sub>		1	5	12 **
DAD dd <sub>1</sub>		1	4	5
*DADI M		3	10	15 **
*CMD H		1	2.5	14 **
*TCMP H		1	3	16.5 **
*TCSM		1	3-5.5	27.5 **
*RLCH		1	5.5	12.5 **
*RRCH		1	6.5-7.5	40 **
*RALH		1	5.5-6	12.5-15 **
*RARH		1	6.5-7.5	40 **
*CMHDX		3	10.5-15.5	
*CMHD dd		1	7.5-13	
*CMHD M		1	10.5-15.5	
- Accumulator Group Instructions -				
LDA		3	6.5	6.5
STA		3	6.5	6.5

Table 12. (Continued)

Mnemonic	Number of Words		Execution Time (US)	Equivalent Intel 8080 Execution Time (US)
	Double Op Code	Single Op Code		
CMA		1	2.5	2
CLA		1	2.5	2
STAX EF		1	3.5	3.5
LDAX EF		1	3.5	3.5
RLC		1	2.5	2
RRC		1	2.5	2
RAL		1	3.5	2
RAR		1	3.5	2
ADD $r_2$		1	2.5	2
SUB $r_2$		1	2.5	2
ANA $r_2$		1	2.5	2
XRA $r_2$		1	2.5	2
ORA $r_2$		1	2.5	2
CMP $r_2$		1	2.5	2
ADD M, ADI		1,2	3.5	3.5
SUB M, SBI		1,2	3.5	3.5
ANA M, ANI		1,2	3.5	3.5
XRA M, XRI		1,2	3.5	3.5
ORA M, ORI		1,2	3.5	3.5
CMP M, CPI		1,2	3.5	3.5

Table 12. (Continued)

Mnemonic	Number of Words		Execution Time (US)	Equivalent Intel 8080 Execution Time (US)
	Double	Single		
	Op Code	Op Code		

---

- Program Counter and Stack Instructions -				
CALL		3	8.5	8.5
RET		1	5	5
*PUSH A		1	3.5	
*POP A		1	3.5	
JC		3	3.5-5.5	5
JNC		3	3.5-5.5	5
JZ		3	3.5-5.5	5
JNZ		3	3.5-5.5	5
JP		3	3.5-5.5	5
JM		3	3.5-5.5	5
JPE		3	3.5-5.5	5
JPO		3	3.5-5.5	5
PUSH dd <sub>1</sub>		1	5	5.5
POP dd <sub>1</sub>		1	5	5
XTHL		1	7.5	9
- Bit Oriented Instructions -				
*SMB		1	5	63.5 **
*RMB		1	5	63.5 **
*SMBI		3	8	68.5 **
*RMBI		3	8	68.5 **

Table 12. (Continued)

Mnemonic	Number of Words		Execution Time (US)	Equivalent Intel 8080 Execution Time (US)
	Double Op Code	Single Op Code		
*TMB		1	4.5	50.5 **
*TMBI		3	7.5	55.5 **
*SBA		1	2.5	34.5 **
*RBA		1	2.5	34.5 **
*SIIM		1	3	16.5 **
*RIIM		1	3	16.5 **
- Miscellaneous Instructions -				
HLT	2		Variable	Variable
NOP	2		3.5	2
EI	2		4	2
DI		1	2.5	2
STC	2		4	2
CFL	2		6	2

\*\* Time given represent approximate execution time for an 8080 machine language routine.



unless otherwise noted. Double op codes were assigned to 61 of the total 316 possible op codes needed for the basic instruction set. Also, it was assumed that a 2.0 MHz system clock was used to drive the GT 1248, since this is the maximum clock frequency used with the Intel 8080.

General Purpose Instructions. The single data word oriented instructions in the 8080 instruction set appear to be adequate for handling eight bit process control operations. This is to be expected since the 8080 is primarily an eight bit data oriented machine. For this reason, the single word instructions included in the GT 1248 instruction set are essentially mirror images of their 8080 counterparts.

On the other hand, the 8080 does not support double data word operations to the extent needed for efficient process control applications. A great deal of the programming used in process control is used in conjunction with A/D and D/A converters which normally require more than eight bits of data. The instruction set of the Level (1) microprocessor should then support these double word operations to much the same extent that it supports its single word operations. The 8080 does not do this so the GT 1248 features a number of double data word instructions designed to supplement those found in the 8080.

Special Purpose Instructions. In addition to being both eight and 16 bit data oriented, the Level (1) microprocessor must also be capable of manipulating individual bits at the register level. This is the area in which the 8080 runs into the most trouble, as is evident from Table 6. In order to compare the relative performance of each microprocessor in this area, the relative performance,  $P_r = P_{1248}/P_{8080}$ , is given in Table 13 for the various bit oriented instructions.

Table 13. Relative Performance for Bit Oriented Instructions

Mnemonic	GT 1248		Intel 8080		$P_r$	Speedup
	T(US)	N	T(US)	N		
SMB	5	1	63.5	3	38.1	12.7
SMBI	8	3	68.5	6	17.12	8.56
TMB	4.5	1	50.5	3	33.66	11.22
TMBI	7.5	3	55.5	6	14.8	7.4
SBA	2.5	1	34.5	3	41.4	13.8
SIIM	3	1	16.5	7	38.5	5.5

It was assumed in calculating the values for Table 13 that the 8080 routines were written as subroutines called by the control program. This would certainly be the case since each subroutine is relatively long, prohibiting inline programming. The only exception was taken with the SIIM operation, which was assumed to be written inline due to its brevity. These 8080 subroutines are listed in Appendix F.

#### Robot Control Evaluation Problem

At this point both microprocessors will be evaluated regarding direct computer control of an industrial robot. Parameters for the test situation are taken from the operation of the Unimate industrial robot [24].

Test Situation. The Unimate industrial robot is a multiple articulation point to point machine. Physically, the Unimate resembles a

single mechanical arm extending from its control apparatus. The arm has six motion axes which define its position. A commanded position for the robot is given by a 128 bit program step which is stored in memory. This program step is divided into eight 16 bit groups. Of these eight 16 bit groups, six are used to control the six motion axes of the Unimate, while the remaining two are devoted to auxiliary or ancillary information such as operate external, wait for external, clamp, weld, time delay, etc. The group assignments are:

- Group 0: Auxiliary Commands
- Group 1: Auxiliary Commands
- Group 2: Swivel Servo
- Group 3: Out-in (radial motion) servo
- Group 4: Yaw servo (wrist)
- Group 5: Down-up (shoulder motion) servo
- Group 6: Bend servo (wrist)
- Group 7: Rotary (waist motion) servo

Figure 19 gives a description of the robot control configuration. The basic operation of the control system is as follows. Two milliseconds are typically allowed for the scan of a complete program step. Therefore the scan time for each articulation or auxiliary command is 250  $\mu$ sec. During the first 250  $\mu$ sec of a program step, the negative commanded position for the first articulation is read from group one of the program step and subtracted from the present position shaft encoder corresponding to that group. The results of this subtraction form an error quantity that is applied to the D/A converter. An octal decoder enables the sample and hold circuit (S/H) that corresponds to

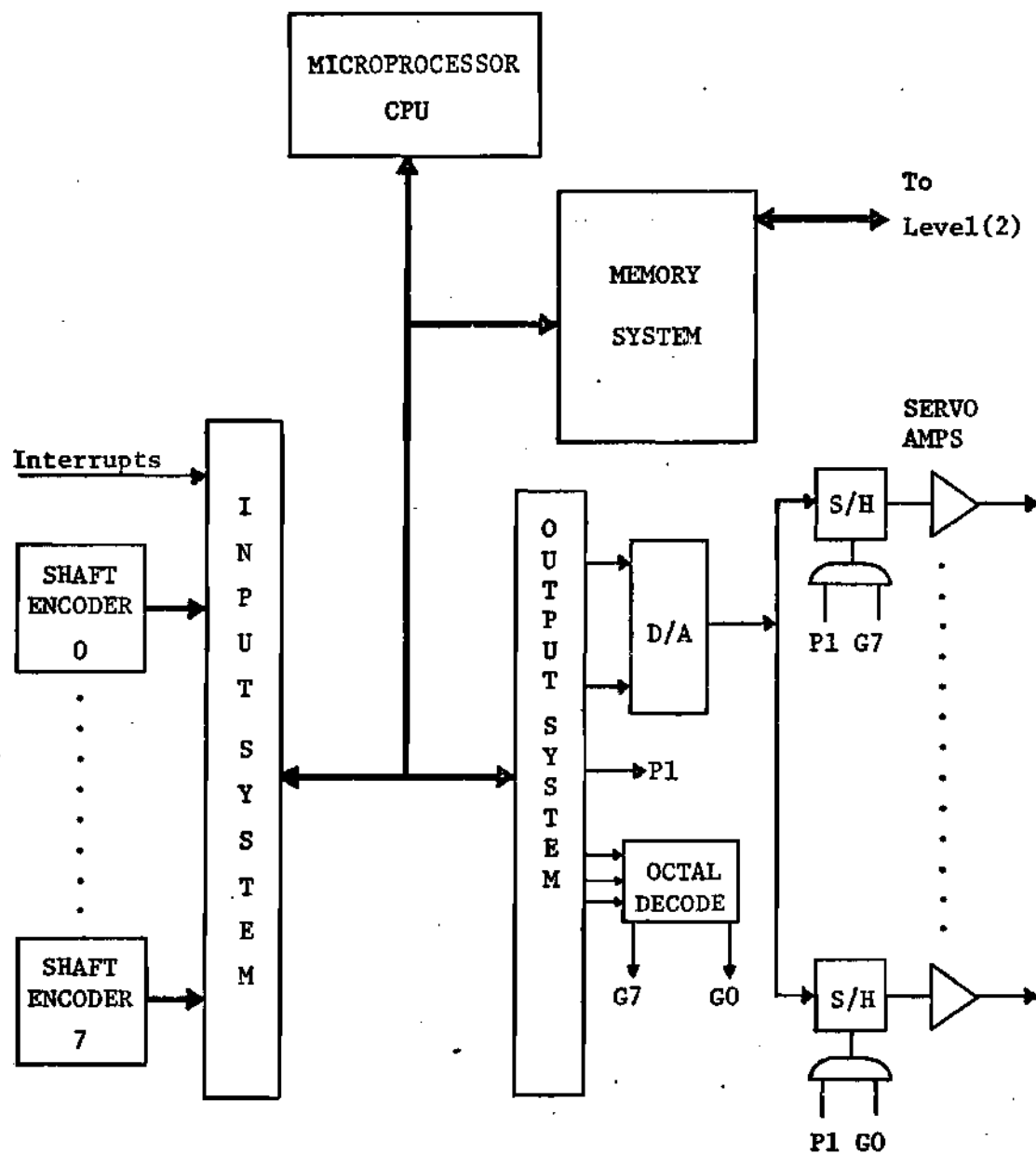


Figure 19. Robot Control Configuration

the present group and the analog error signal is applied to the proper servo amplifier when line P1 is pulsed. This procedure is repeated another seven times for the remaining seven groups within the program step, thus completing the two millisecond program step cycle. A particular program step is cycled through until all error signals are reduced to zero, and then the next program step is acted upon.

A map of the memory, input and output systems as they apply to this problem is given in Figure 20. Each group is made up of two eight bit words of memory, so that a program step occupies 16 sequential memory locations. Successive program steps are arranged sequentially in memory. The "Program Base Address" is a pointer to the base of the program step sequence and the "Current Program Step Address" is a pointer to the address of the program step under current execution. Selection of the correct group within the program step is accomplished through the use of the "Current Field" pointer. The "Program Step Counter" keeps track of the number of program steps that have been executed and is used to indicate the end of a program step sequence.

The flag word is tested at the end of each program step cycle (2 ms. cycle) to determine if conditions have been met for sequencing to the next program step. An interrupt routine initiated by analog comparators monitoring the error voltages could be used to set the flag, but for this particular test it will be assumed that the flag is set by some undefined routine.

A flowchart depicting the steps which the microprocessor must perform is found in Figure 21. This flowchart assumes that the microprocessor will perform the conversion from gray code (which is the code

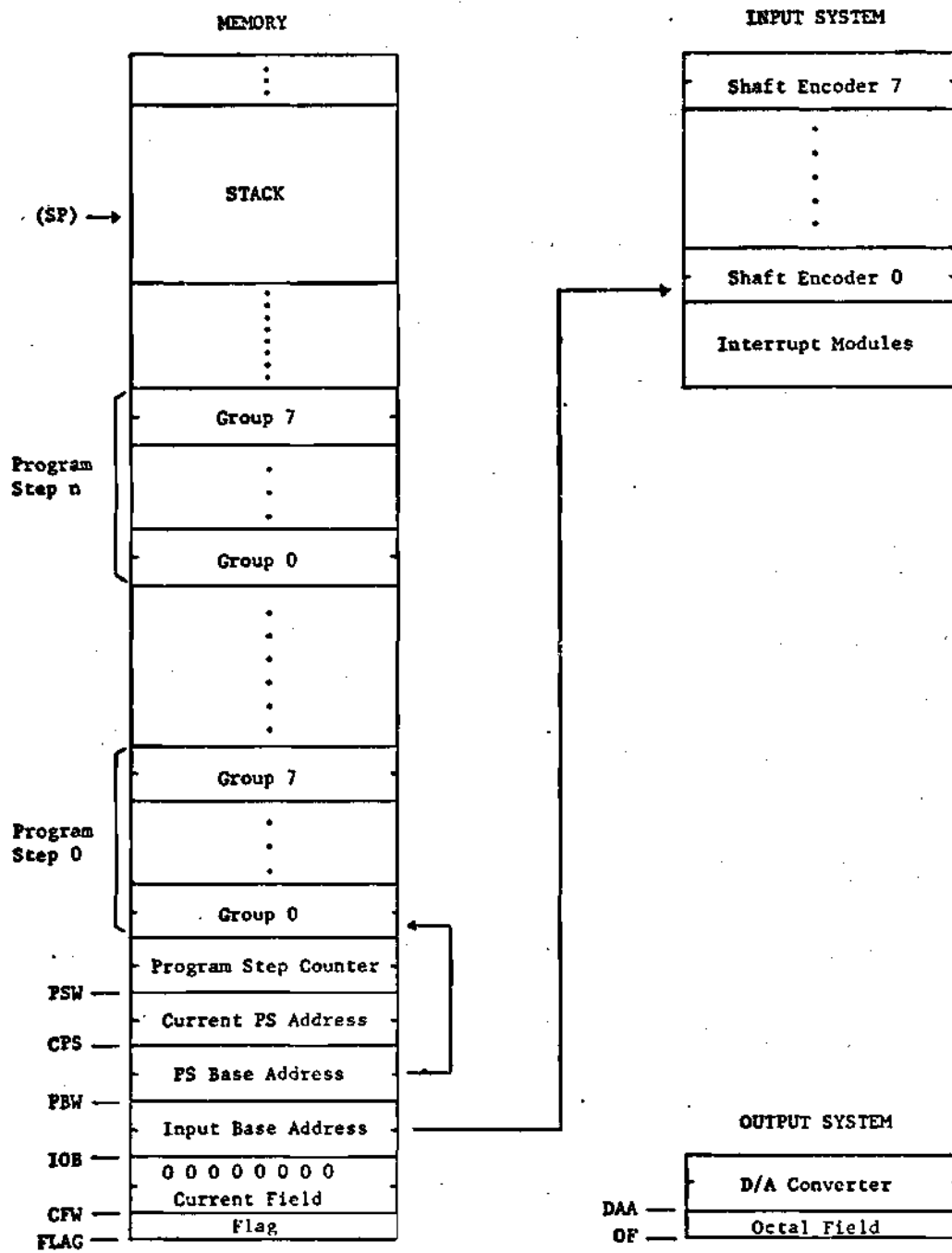


Figure 20. Memory and I/O Map for Robot Control Problem

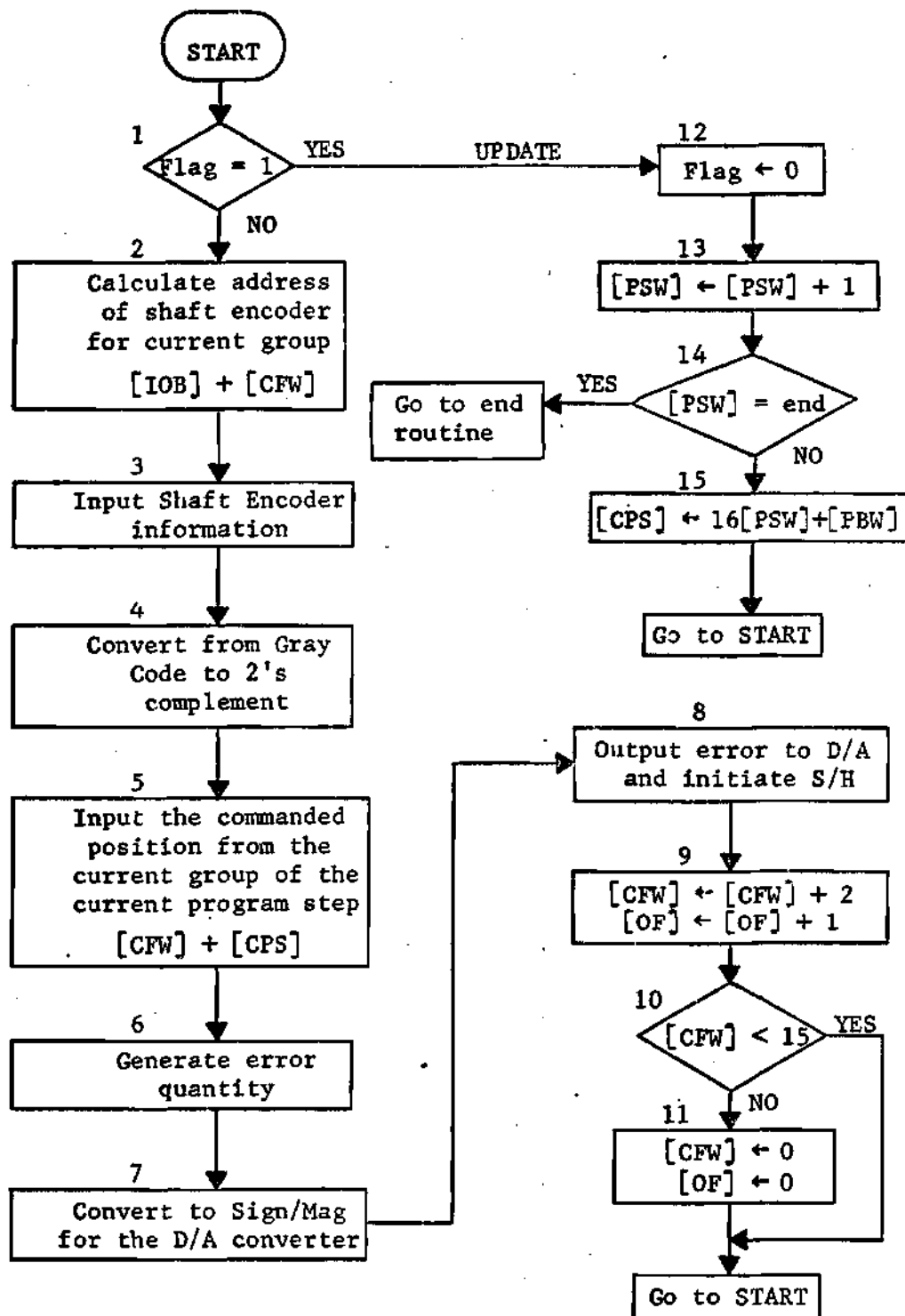


Figure 21. Robot Control Flowchart

used on the shaft encoders) to binary itself instead of using an exclusive-or gate array. This assumption was made so that the benefits of being able to create special purpose process control instructions on the GT 1248 could be compared to the machine language routine approach that must be taken on the Intel 8080.

Table 14 gives a summary of the results of using each micro-processor to perform the algorithm of Figure 21. The programs used in this test may be found for both processors in Appendix G. Relative performance ( $P_r$ ) is defined as performance for the GT 1248 divided by performance for the Intel 8080. The speedup is the execution time for the GT 1248 divided by the execution time for the 8080.

Discussion of Results. Several important limitations of the Intel 8080 are clearly brought out by the data of Table 13. Referring to step four, the advantages of being able to create a custom machine language instruction through user microprogramming is obvious. The 8080 suffers performance degradation here because the routine must be programmed in assembly language and because the 8080 does not support bit manipulation in its hardware or software. Relative performance is high in step four largely due to the numerous words of source code needed to program the routine in the 8080. Since the cost of a 2K RAM memory is over half the cost of the Intel 8080 CPU at this time, the need for keeping the process control programs as short as possible is obvious on an economic basis.

Step seven displays similar results. Again the 8080 gets tied up in a relatively long assembly language routine instead of having a conversion routine included in the instruction set.



Table 14. Evaluation of Test Program Results

Step Number in Figure 21	Execution Time		Relative Performance $P_r$	Speedup
	GT 1248 (US)	Intel 8080 (US)		
1	12	13.5	1.31	1.12
2	18	23	1.7	1.28
3	5	9.5	5.7	1.9
4	42.5	164.5	290.3	3.87
5	25.5	34.5	2.03	1.35
6	4	5	1.25	1.25
7	5.5	27.5	65	5
8	23	140	9.13	6.08
9	19	21.5	1.27	1.13
10	9	9	1	1
11	20.5	20	0.975	0.975
12	9	8.5	0.944	0.944
13	10.5	18.5	4.11	1.76
14	21	22	2.09	1.05
15	36	43	1.3	1.2
Major Cycle Steps 1-15	260.5	560	4.03	2.15
Minor Cycle Steps 1-11	184	468	5.22	2.54

A critical deficiency in the 8080 is found in the execution of step eight. This step outputs the error data to the D/A converter and then initiates the S/H operation by setting and then resetting a single output line. It is in the manipulation of the single output line that the 8080 runs into its most trouble. The GT 1248 can set a single output bit in 5 usec., as opposed to the 63.5 usec. taken by the 8080 subroutine. Process control applications utilizing a microprocessor without a hardware bit manipulation feature will suffer an execution speed degradation most severely in the bit oriented area.

The other flowchart steps show varying degrees of difference between the two microprocessors, depending to a large extent on the amount of 16 bit data being handled. The 8080 is capable of handling most double word operations without too much difficulty, although at a somewhat reduced execution speed compared to the GT 1248. Architecturally the 8080 is equipped to handle 16 bit data but the instruction set just does not support this feature to the extent that it does single word data. The double word compare operation of step 14 would be much slower in the 8080 if negative operands were allowed as in the GT 1248.

### Conclusions

This research has investigated the design of a microprocessor for process control application in a discrete manufacturing environment. The design encompasses five major areas: 1) input and output system architecture, 2) instruction set requirements, 3) hardware bit manipulation requirements, 4) processor architecture and 5) microprogrammable control features. This microprocessor design has been compared to a

typical second generation microprocessor (Intel 8080) in the control of an industrial robot.

The input and output system architecture developed in this paper is similar to architectural approaches taken in minicomputer systems but was developed to directly interface with microprocessor capabilities. The I/O system presented in this paper is oriented to both bit and byte digital input and output. Using this basic architecture, the overall I/O system may be easily upgraded to handle analog signals through A/D and D/A converters. Additionally, other special purpose subsystems such as event sense or relay register options are directly compatible to the basic bit and byte I/O architecture. With regard to the memory system, memory modules are commercially available now that parallel the blocked memory system presented in Chapter III. Vertical communication with the Level (2) Supervisory computer can be implemented using a simple multiplex circuit that switches a "transfer memory module" between the Level (1) and Level (2) computers.

Present generation eight bit microprocessors primarily support eight bit data through their instruction set. The Intel 8080 at present is the only available machine supporting both eight and 16 bit data manipulation directly through its instruction set. A process control microprocessor should support at least 12 bit data formats since this is a common A/D converter width. The 8080 can handle this type of programming but the limited number of double word instructions included in its instruction set limits execution speed due to software routines. The GT 1248 overcomes this difficulty by including an expanded double

word instruction group within its basic instruction set.

No present microprocessor directly supports bit manipulation at the register level. In order to achieve the execution speed of bit oriented instructions enjoyed by minicomputer process control systems, a microprocessor based system must have hardware support of bit manipulation. The Intel 8080 must implement these instructions through software routines that are costly both in memory space and execution speed. As an example, the TI 960 minicomputer can set a single bit in its output system in 7 usec. Using a software routine the Intel 8080 takes approximately 63 usec. for an equivalent operation while the GT 1248 takes only 5 usec.

The addition of bit manipulation to a microprocessor should require no major architectural changes. When compared with the Intel 8080, the only additional bit manipulation hardware used on the GT 1248 included the bit manipulation A register, addition of the B register and corresponding additions to the control section. With respect to an increase in chip area, the changes should not be significant. On the 8080, the decimal arithmetic hardware could be replaced with the bit manipulation hardware.

The advantages of using a microprocessor with a user microprogrammable control section have already been voiced [18]. Specific advantages of this feature have been examined in this paper with respect to the process control problem. Considerable savings in execution time and memory required can be gained through this approach since the object code necessary for a given task is reduced. It is felt that the added

cost of implementing the external control store on the GT 1248 will be offset by subsequent reductions in the amount of program memory required. In microprocessor based systems, memory cost can easily outweigh the cost of the microprocessor itself.

In summary, microprocessors should make a significant impact upon the process control computer market. Present second generation microprocessors are capable of operating as Level (1) controllers in operations compatible with their execution speed. In the future there should be a shift toward an instruction set more oriented toward 16 bit data for process control microprocessors. A bit manipulation capability should also be added to the architecture. As third generation microprocessors are introduced with lower execution times, the higher order functions such as multiply and divide will become feasible within the operation set of the process control microprocessor.

## APPENDIX A

## INPUT MODULE CIRCUITRY

The input system may be composed of different types of modules for varying input requirements. The standard module is one that accepts w binary valued input lines. Figure 22 shows the circuitry necessary for the implementation of this type input module.

Input module circuitry consists mainly of address decoding logic to select the addressed input module and the necessary logic to allow the wired-OR connection to the data bus. Since wired-OR connection is being used, negative logic will be employed on the data bus. That is, zero volts will represent a logic one and the positive voltage level will represent logic zero.

Output buffer NAND gates are open collector type. When a logic zero (positive logic) is applied to the Module Select Line, the data bus is free to float with respect to the input module. A logic one applied to the Module Select Line allows the information present at the input lines to be transferred to the data bus. The input buffer is optional and could be used for level shifting and input protection. The address decode logic is used to enable the addressed input module. Module address is set by using jumper wires between the outputs of the true/invert gates and an AND gate. No provision is made for a ready signal to the CPU since the speed of the logic used in the input module is much faster than the sequential operation of the CPU. The read sequence for the input module follows.

1. CPU gates the address of the input module onto the address bus.

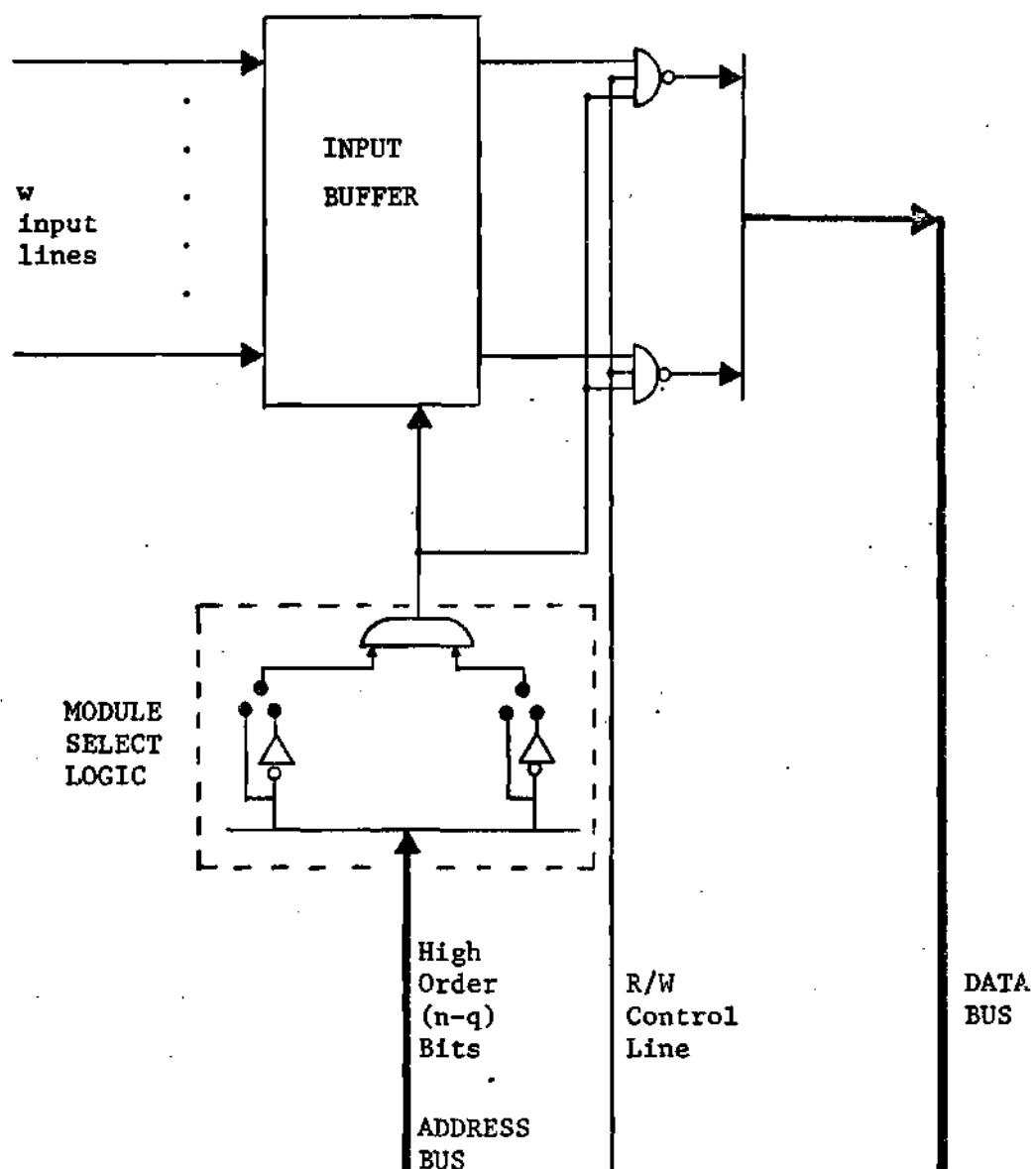


Figure 22. Input Module Circuit Diagram

2. Address decode logic enables the specified input module and the state of the input lines is gated onto the data bus.

3. CPU gates the state of the data bus, inverted, into a register within the CPU.

Operation complete.



## APPENDIX B

## INTERRUPT MODULE

Timing and Logic

The logic circuitry necessary to implement the interrupt module is shown in Figure 23. For each of the four basic interrupt operations a timing sequence and explanation of operation follows.

## A. Read Interrupt Buffer (RIB) Operation.

1. CPU gates n bit address onto the address bus.
2. a. R/W control line is set to the logic one (read) state.  
b. Module select logic enables specified interrupt module.
3. The state of the interrupt flip flops is gated onto the data bus.
4. CPU gates data bus state into an internal register.

This sequence completes the actual RIB operation. Once the interrupt state is gated into the CPU, the low order q bits of the address register can be used to sequentially test the interrupt word until a logic one is found. The corresponding address of the address register would specify the interrupting device.

## B. Set Mask (SMASK) Operation.

1. CPU gates n bit address onto the address bus.
2. a. R/W control line is set to the logic zero (write) state.  
b. Module select logic enables specified interrupt module.
3. CPU gates desired mask state onto the data bus.

4. OCP1 control line is pulsed by the CPU, resulting in the mask state being transferred into the mask flip flops.

C. Set Individual Interrupt Mask (SIIM) Operation.

1. CPU gates n bit address onto the address bus.
2. a. R/W control line set to logic zero (write) state.  
b. Module select logic enables specified interrupt module.  
c. Decoder enables specific mask flip flop.
3. OCP3 control line is pulsed by CPU, resulting in the specified interrupt mask flip flop being set to the interrupt enable state.

D. Reset Individual Interrupt Mask (RIIM) Operation.

1. CPU gates n bit address onto the address bus.
2. a. R/W control line set to logic zero (write) state.  
b. Module select logic enables specified interrupt module.  
c. Decoder enables specific mask flip flop.
3. OCP2 control line is pulsed by CPU, resulting in the specified interrupt flip flop being set to the interrupt disable state and the interrupt flip flop being set to the reset state.

For a possible implementation of the module select logic section, refer to the circuit diagram of Figure 22 in Appendix A.

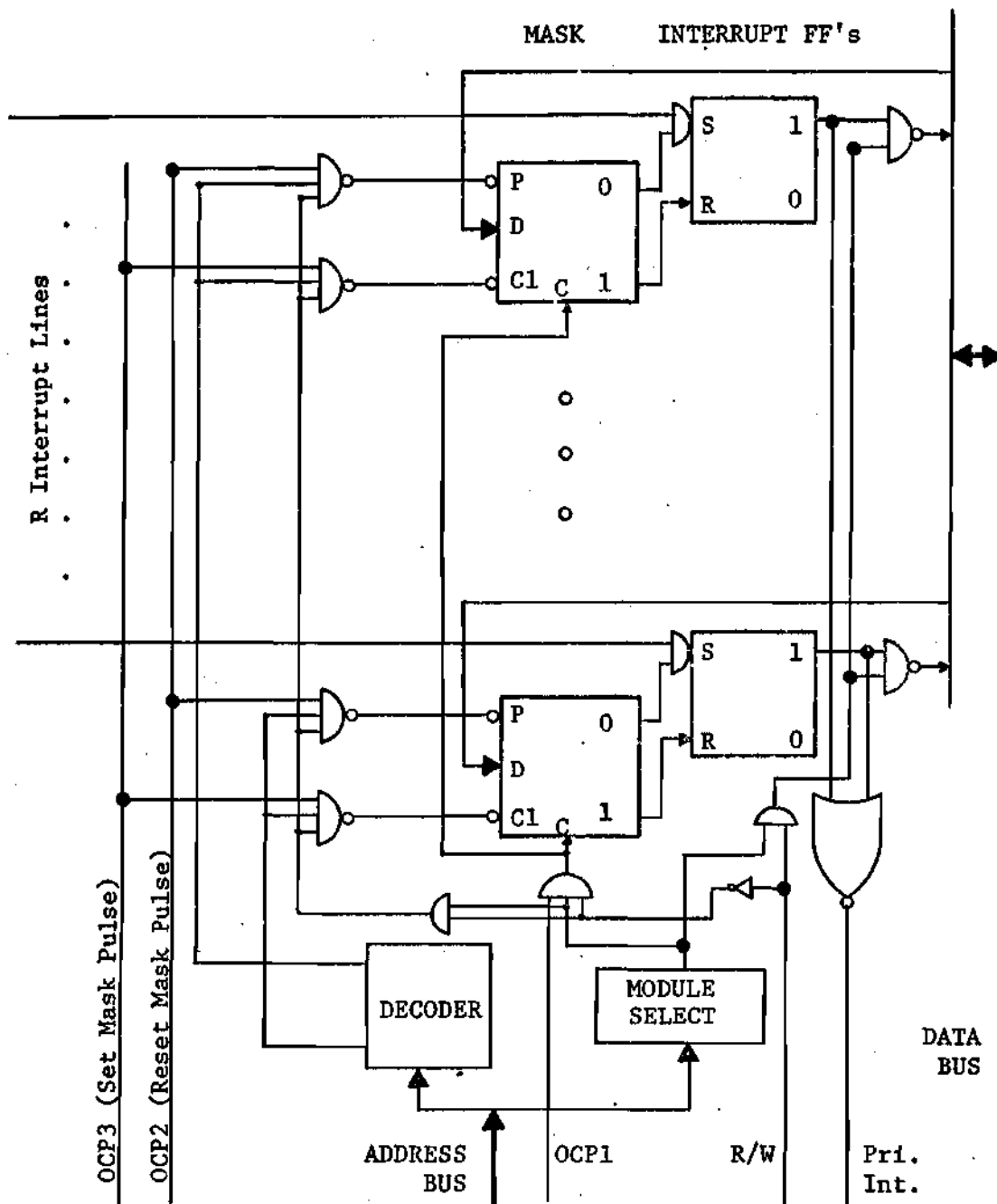


Figure 23. Interrupt Module Circuitry

## APPENDIX C

## OUTPUT MODULE CIRCUITRY

Logic necessary for the implementation of the output module is shown in Figure 24. The module select logic is the same as that found on the input system modules. The sequence of operation for the two output system operations follows.

## A. Read Full Word (RFW) Operation.

1. CPU gates n bit address onto the address bus.
2. a. R/W control line set to the logic one (read) state.  
b. Module select logic enables specified output module.
3. The state of the output flip flops is gated onto the data bus.
4. CPU gates the data bus state, inverted, into an internal register.

## B. Write Full Word (WFW) Operation.

1. CPU gates n bit address onto the address bus.
2. a. R/W control line set to the logic zero (write) state.  
b. Module select logic enables specified output module.
3. CPU gates output word onto the data bus.
4. OCP1 control line is pulsed, transferring output word to the output flip flops driving the output lines.

Physically, several output modules may be packaged on one PC board in order to save space.

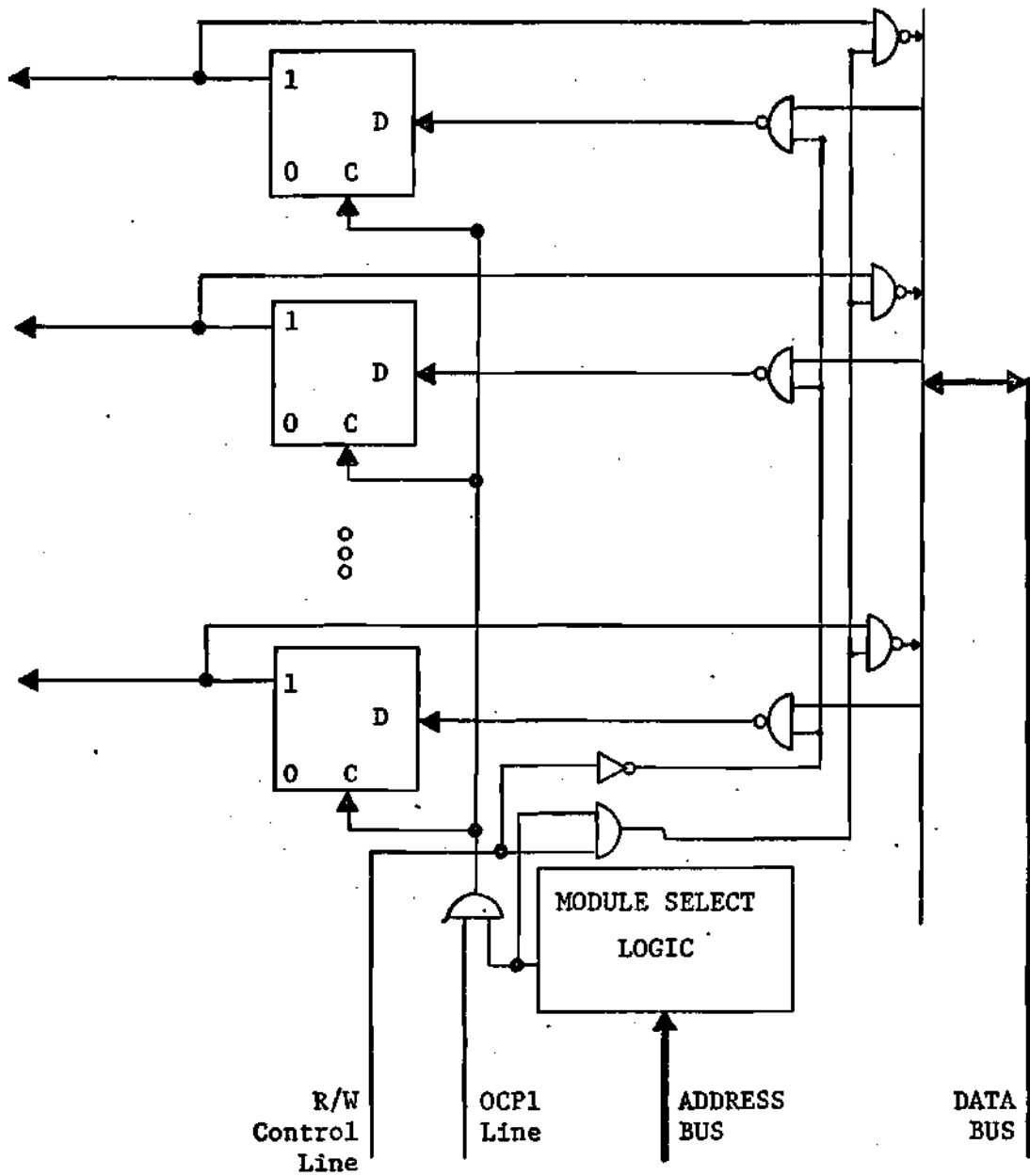


Figure 24. Output Module Logic Circuitry

## APPENDIX D

## MEMORY MODULE

The memory module must contain circuits for address decoding, data storage and memory cycle sequencing. If semiconductor data storage is used, either dynamic or static memory elements may be employed. Dynamic memory requires circuitry to periodically refresh the binary storage elements, while static memory requires no refresh circuitry. Higher bit densities are possible with dynamic as opposed to static memory, although static memory modules are easier to design since they require no refresh circuits. Figure 25 shows the organization of a  $2^p$  word by  $w$  bit static memory using  $2^k$  word by  $m$  bit memory chips. In general, the rules [20] for wiring the array of Figure 25 are as follows:

1. All corresponding power supply leads are made common throughout the array.
2. The write enable signal is made common throughout the array.
3. All corresponding addresses are made common throughout the array.
4. Corresponding data input and data output leads are made common within array columns.
5. Corresponding chip select leads are made common within each row. The function of the chip select leads is to permit the array interconnection. When conditions for chip, i.e., row, selection are not met, no input signal can affect the contents of that row. Nor does any

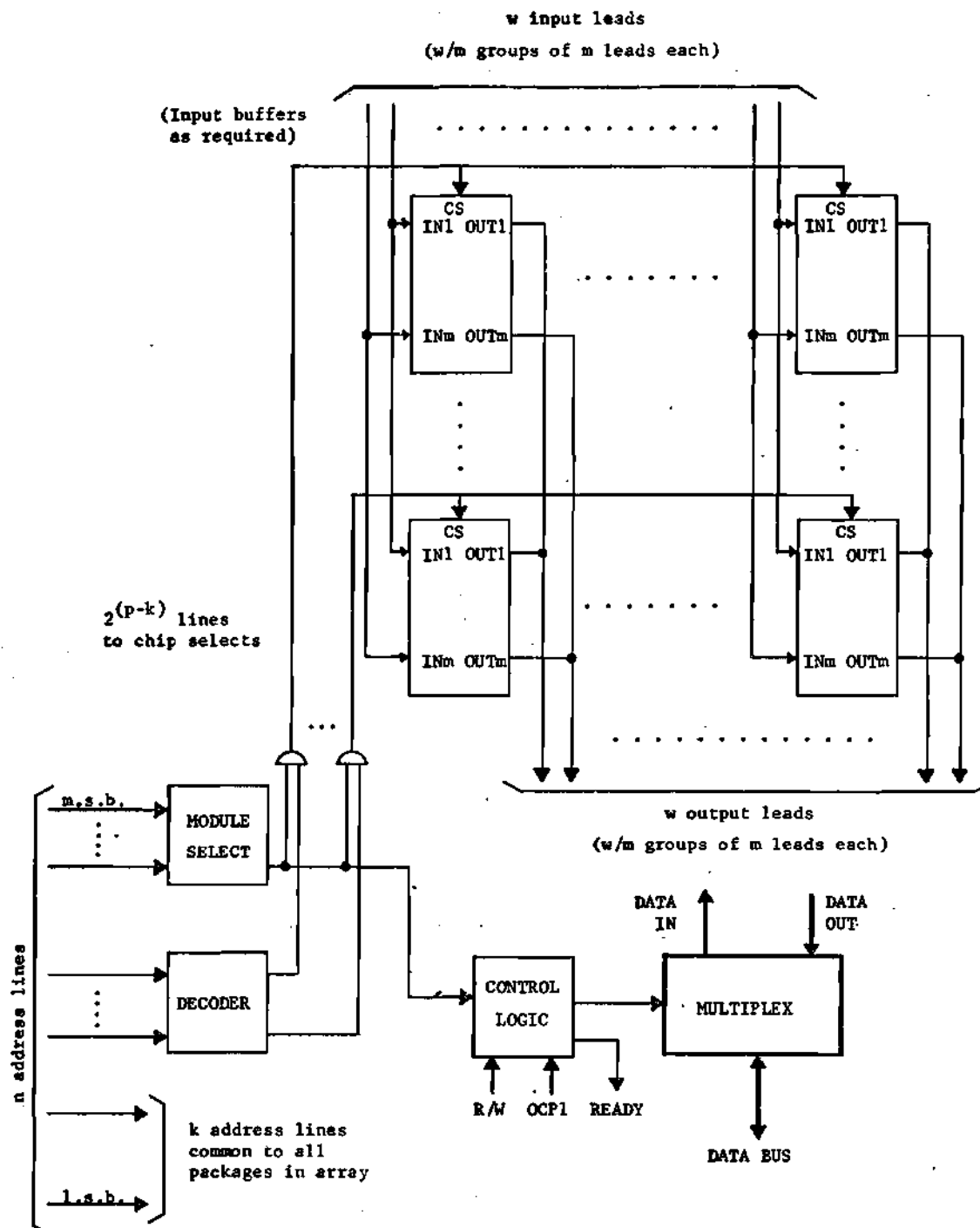


Figure 25. Memory Module Circuitry

unselected chip affect the signals on the data output lines to which it is connected. Thus the chip select lead permits output leads to be OR-tied and eliminates the necessity to decode the write pulse signal.

Further, the module select logic decodes the high order (n-p) bits of the address to enable the correct memory module. The control logic gates data onto the data bus through the multiplex circuit (MPX) whenever the R/W control line is in the read (logic 1) state and the module is enabled. The Ready line indicates to the CPU that valid data is present on the data bus. When the R/W line is in the write (logic 0) state and the module is enabled, data is gated from the data bus into the input leads of the memory chips. The normal state of the R/W control line should be in the logic zero (write) state.

Figure 26 shows the timing sequence for the read operation. First, the address is sent out over lines  $A_{15} - A_0$  of the address bus. This address enables the correct memory module, but at this point the ready line is unaffected. Upon receiving the read signal from the R/W line, the memory module recognizes that a read operation upon the specified address is being performed. The ready line is then pulled to the low state so that the CPU will not erroneously input bad data during the memory access time. Once valid data has been loaded into an output buffer within the memory module, the module indicates ready and the CPU subsequently inputs this data and sets the R/W line to the zero state. This signals the memory module to complete the memory cycle time in the case of dynamic semiconductor or core type memories.

Figure 27 shows the timing sequence for the write operation. The



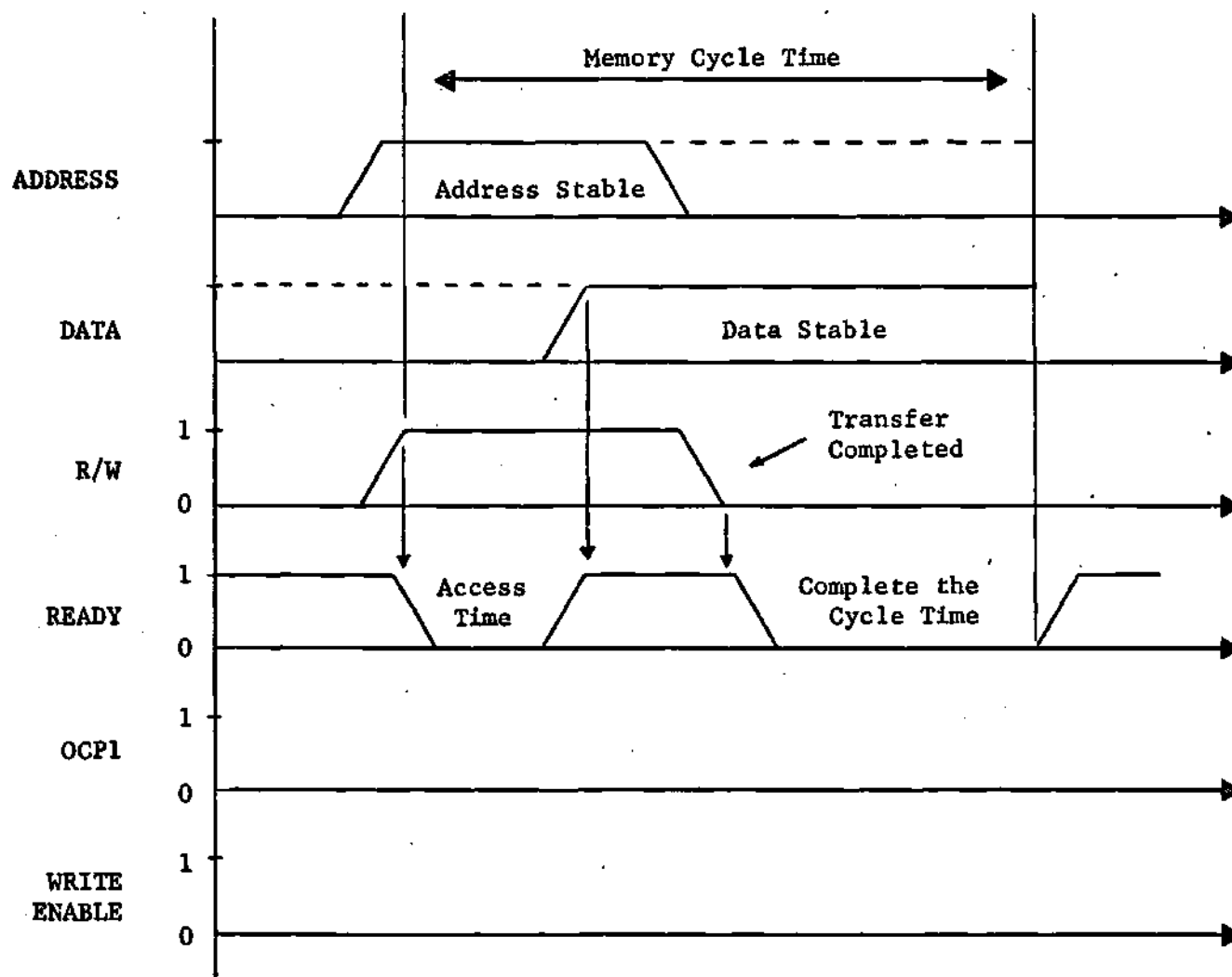


Figure 26. Memory Read Timing Sequence

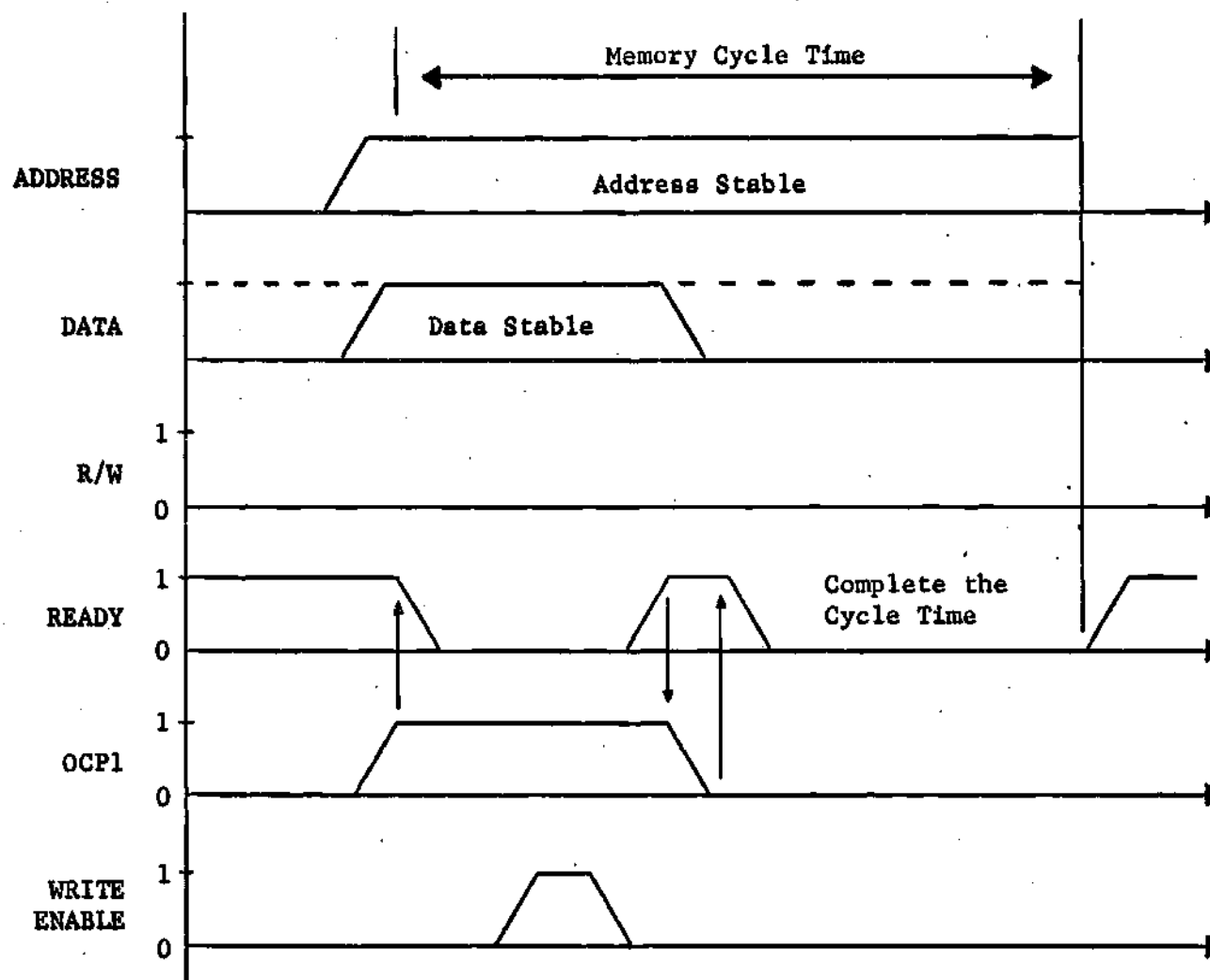


Figure 27. Memory Write Timing Sequence

address is first sent out as in the read operation. Next data is sent out over the data bus,  $D_7 - D_0$ , and the OCPl line indicates to the memory module that valid data is present on the data bus. This signals the memory module to start a write cycle and the ready line is pulled down to the low state until the data has been written. When the ready line again indicates ready the CPU can set the OCPl line to the zero state. This action indicates to the memory module to signal not ready if the memory cycle time must be completed so that a subsequent memory access cannot be made until the memory is ready.

Memory module timing as shown here is compatible both with the Intel 8080 and GT 1248 theoretical microprocessor.

## APPENDIX E

## BASIC INSTRUCTION SET FOR THE GT 1248

Table 15 gives a list of the symbols and their meanings as they are used throughout this appendix.

In order to differentiate between those members of the GT 1248 instruction set that are essentially the same as those found in the Intel 8080 and those members that have been added in an effort to support the process control environment, new instructions will be preceded by an asterisk.

Table 15. Programming Symbols

Symbol	Meaning
$\langle B_n \rangle$	The $n^{\text{th}}$ byte of the instruction
$r_1$	One of the CPU registers A,E,F,X, Y,U,V or B
$r_2$	One of the CPU registers H,L,E,F, X or Y
dd	One of the double registers HL, EF, XY, UV, GK, SP, PL or MAR.
$dd_1$	One of the double registers HL, EF, XY or UV
$dd_2$	One of the double registers EF, XY, UV or GK
( )	Contents of register
[ ]	Contents of memory
$A_B$	Bit B of the A register
$\rightarrow$	Is transferred to
$\leftrightarrow$	Is exchanged with
$b_{\text{adr.}}$	Single bit addressed by [adr.]
ddH	High eight bits of the double register
ddL	Low eight bits of the double register

Single Register and Memory Instructions

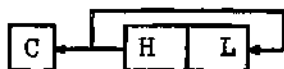
Mnemonic	Flags Affected	Description of Operation
MOV $r_{1a}, r_{1b}$	All	$(r_{1a}) \leftarrow (r_{1b})$
MOV M, $r_1$	All	$[(HL)] \leftarrow (r_1)$
MOV $r_1$ , M	All	$(r_1) \leftarrow [(HL)]$
MVI $r_1$	All	$(r_1) \leftarrow \langle B_2 \rangle$
MVI M	All	$[(HL)] \leftarrow \langle B_2 \rangle$
*MVII M	All	$[\langle B_2 \rangle \times \langle B_3 \rangle] \leftarrow \langle B_4 \rangle$
INR $r_1$	All	$(r_1) \leftarrow (r_1) + 1$
DCR $r_1$	All	$(r_1) \leftarrow (r_1) - 1$
INR M	All	$[(HL)] \leftarrow [(H)(L)] + 1$
DCR M	All	$[(HL)] \leftarrow [(HL)] - 1$
*INRI M	All	$[\langle B_2 \rangle \times \langle B_3 \rangle] \leftarrow [\langle B_2 \rangle \times \langle B_3 \rangle] + 1$
*DCRI M	All	$[\langle B_2 \rangle \times \langle B_3 \rangle] \leftarrow [\langle B_2 \rangle \times \langle B_3 \rangle] - 1$

Double Register and Memory Instructions

Mnemonic	Flags Affected	Description of Operation
INX dd	None	$(dd) \leftarrow (dd) + 1$
DCX dd	None	$(dd) \leftarrow (dd) - 1$
*IWX M	C,S	$[(HL) + 1][(HL)] \leftarrow [(HL) + 1][(HL)] + 1$
*DCX M	C,S	$[(HL) + 1][(HL)] \leftarrow [(HL)] + 1][(HL)] - 1$
*INXI M	C,S	$[\langle B_2 \rangle \times \langle B_2 \rangle + 1][\langle B_2 \rangle \times \langle B_3 \rangle] \leftarrow [\langle B_2 \rangle \times \langle B_3 \rangle + 1]$ $[\langle B_2 \rangle \times \langle B_3 \rangle] + 1$

*DCXI M	C,S	$[\langle B_2 \times B_3 \rangle + 1][\langle B_2 \times B_3 \rangle] \leftarrow [\langle B_2 \times B_3 \rangle + 1]$ $[\langle B_2 \times B_3 \rangle] - 1$
LXI dd	S	$(dd) \leftarrow \langle B_2 \times B_3 \rangle$
MOVX $dd_a, dd_b$	None	$(dd_a) \leftarrow (dd_b)$
STDD $dd_1$	None	$[\langle B_2 \times B_3 \rangle + 1][\langle B_2 \times B_3 \rangle] \leftarrow (dd_1)$
LDDD $dd_1$	None	$(dd_1) \leftarrow [\langle B_2 \times B_3 \rangle + 1][\langle B_2 \times B_3 \rangle]$
*STID $dd_2$	None	$[(HL) + 1][(HL)] \leftarrow (dd_2)$
*LDID $dd_2$	None	$(dd_2) \leftarrow [(HL) + 1][(HL)]$
DAD $dd_1$	S,C	$(HL) \leftarrow (HL) + (dd_1)$
*DADI M	S,C	$(HL) \leftarrow (HL) + [\langle B_2 \times B_3 \rangle + 1][\langle B_2 \times B_3 \rangle]$
*CMD H	None	$(HL) \leftarrow (\overline{HL})$
*TCMP H	None	$(HL) \leftarrow \text{two's complement } (HL)$
*TCSM H	S	convert two's complement (HL) to sign/ magnitude

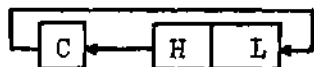
\*RLCH



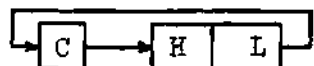
\*RRCH



\*RALH



\*RARH



\*CMHDX

If  $(HL) = \langle B_2 \times B_3 \rangle$  then  $(Z) \leftarrow 1$  $(HL) > \langle B_2 \times B_3 \rangle$  then  $(C) \leftarrow 1$ 

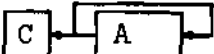

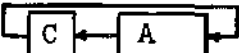

\*CMHD dd

If  $(HL) = (dd)$  then  $(Z) \leftarrow 1$  $(HL) > (dd)$  then  $(C) \leftarrow 1$ 

\*CMHD M

If  $(HL) = [(XY)+1][(XY)]$  then  $(Z) \leftarrow 1$  $(HL) > [(XY)+1][(XY)]$  then  $(C) \leftarrow 1$ Data in two's  
complement form

Accumulator Group Instructions

Mnemonic	Flags Affected	Description of Operation
LDA	All	$(A) \leftarrow [\langle B_2 \rangle \langle B_3 \rangle]$
STA	All	$[\langle B_2 \rangle \langle B_3 \rangle] \leftarrow (A)$
CMA	All	$(A) \leftarrow (\bar{A})$
CIA	All	$(A) \leftarrow 0$
STAX EF	All	$[(EF)] \leftarrow (A)$
STAX HL	All	$[(HL)] \leftarrow (A)$
LDAX EF	All	$(A) \leftarrow [(EF)]$
LDAX HL	All	$(A) \leftarrow [(HL)]$
RLC		
RRC		
RAL		
RAR		
ADD $r_2$	All	$(A) \leftarrow (A) + (r_2)$
SUB $r_2$	All	$(A) \leftarrow (A) - (r_2)$ two's complement
ANA $r_2$	All	$(A) \leftarrow (A) \wedge (r_2)$
XRA $r_2$	All	$(A) \leftarrow (A) \oplus (r_2)$
ORA $r_2$	All	$(A) \leftarrow (A) \vee (r_2)$
CMP $r_2$		If $(A) = (r_2)$ then $(Z) \leftarrow 1$ Both quantities positive If $(A) > (r_2)$ then $(C) \leftarrow 1$
ADD M	All	$(A) \leftarrow (A) + [(HL)]$
SUB M	All	$(A) \leftarrow (A) - [(HL)]$



ANA M	All	$(A) \leftarrow (A) \wedge [(HL)]$
XRA M	All	$(A) \leftarrow (A) \vee [(HL)]$
ORA M	All	$(A) \leftarrow (A) \vee [(HL)]$
CMP M		If $(A) = [(HL)]$ then $(Z) \leftarrow 1$ $(A) > [(HL)]$ then $(C) \leftarrow 1$
ADI	All	$(A) \leftarrow (A) + \langle B_2 \rangle$
SBI	All	$(A) \leftarrow (A) - \langle B_2 \rangle$
ANI	All	$(A) \leftarrow (A) \wedge \langle B_2 \rangle$
XRI	All	$(A) \leftarrow (A) \vee \langle B_2 \rangle$
ORI	All	$(A) \leftarrow (A) \vee \langle B_2 \rangle$
CPI		If $(A) = \langle B_2 \rangle$ then $(Z) \leftarrow 1$ $(A) > \langle B_2 \rangle$ then $(C) \leftarrow 1$

#### Program Counter and Stack Instructions

Mnemonic	Flags Affected	Description of Operation
CALL	None	$[(SP)] - 1][(SP) - 2] \leftarrow (PC), (SP) \leftarrow (SP) - 2$ $(PC) \leftarrow \langle B_2 \rangle \langle B_3 \rangle$
RET	None	$(PC) \leftarrow [(SP) + 1][(SP)], (SP) \leftarrow (SP) + 2$
*PUSH A	None	$[(SP) - 1] \leftarrow (A), (SP) \leftarrow (SP) - 1$
*POP A	None	$(A) \leftarrow [(SP)], (SP) \leftarrow (SP) + 1$
JMP	None	$(PC) \leftarrow \langle B_2 \rangle \langle B_3 \rangle$ (equivalent to LXI PC)
JC	None	If $(C) = 1$ then $(PC) \leftarrow \langle B_2 \rangle \langle B_3 \rangle$
JNC	None	If $(C) = 0$ then $(PC) \leftarrow \langle B_2 \rangle \langle B_3 \rangle$
JZ	None	If $(Z) = 1$ then $(PC) \leftarrow \langle B_2 \rangle \langle B_3 \rangle$
JNZ	None	If $(Z) = 0$ then $(PC) \leftarrow \langle B_2 \rangle \langle B_3 \rangle$

JP	None	If (S) = 1 then (PC) $\leftarrow$ $\langle B_2 \times B_3 \rangle$
JM	None	If (S) = 0 then (PC) $\leftarrow$ $\langle B_2 \times B_3 \rangle$
JPE	None	If (P) = 0 then (PC) $\leftarrow$ $\langle B_2 \times B_3 \rangle$
JPO	None	If (P) = 1 then (PC) $\leftarrow$ $\langle B_2 \times B_3 \rangle$
PUSH $dd_1$	None	$[(SP) - 1][(SP) - 2] \leftarrow (dd_1)$ , (SP) $\leftarrow$ (SP) - 2
POP $dd_1$	None	$(dd_1) \leftarrow [(SP) + 1][(SP)]$ , (SP) $\leftarrow$ (SP) + 2
XTHL	None	(L) $\leftrightarrow$ [SP] $\leftrightarrow$ [(SP) + 1], (SP) unchanged

### Bit Oriented Instructions

---

Mnemonic	Flags Affected	Description of Operation
*SMB	None	$b_{HL} \leftarrow 1$
*RMB	None	$b_{HL} \leftarrow 0$
*SMBI	None	$b_{\langle B_2 \times B_3 \rangle} \leftarrow 1$
*RMBI	None	$b_{\langle B_2 \times B_3 \rangle} \leftarrow 0$
*TMB		If $b_{HL} = 1$ then (C) $\leftarrow 1$ else (C) $\leftarrow 0$
*TMBI		If $b_{\langle B_2 \times B_3 \rangle} = 1$ then (C) $\leftarrow 1$ else (C) $\leftarrow 0$
*SBA	None	$A_B \leftarrow 1$
*RBA	None	$A_B \leftarrow 0$
*SIIM	None	The mask flip flop addressed by (HL) is set
*RIIM	None	The mask flip flop addressed by (HL) is reset
HLT	None	CPU is stopped. Upon receipt of an interrupt the next sequential instruction is executed

NOP	None	No operation
EI	None	Enable primary interrupt
DI	None	Disable primary interrupt
STC		$(C) \leftarrow 1$
CFL		$(C) \leftarrow (\overline{C})$

### Microprogram Routines

Symbolic Location	Microcode	States Required (less fetch)
FETCH	DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1, (I/O) $\leftarrow$ 1 JMP Read, * FETCH S, (I/O) $\leftarrow$ 0	3
DFETCH	DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1, (I/O) $\leftarrow$ 1 JMP Read, * FETCH D, (I/O) $\leftarrow$ 0	3
MOV rr	RC; (D) $\leftarrow$ (r <sub>1b</sub> ) Single microcode RC; (r <sub>1a</sub> ) $\leftarrow$ (D) instruction if transfer is not stack to stack. JMP(1) Fetch	2-3
MOVMr	DR; (MAR) $\leftarrow$ (HL) RC; (D) $\leftarrow$ (r <sub>1</sub> ), (OPl) $\leftarrow$ 1 JMP Write, * JMP(1) Fetch	4
MOVrM	DR; (MAR) $\leftarrow$ (HL), (I/O) $\leftarrow$ 1 JMP Read, *	4

	RC; $(r_1) \leftarrow (DP), (I/O) \leftarrow 0$	
	JMP(1) Fetch	
MVIR	DR; $(MAR) \leftarrow (PC), (PC) \leftarrow (PC) + 1,$	4
	$(I/O) \leftarrow 1$	
	JMP Read, *	
	RC; $(r_1) \leftarrow (DP), (I/O) \leftarrow 0$	
	JMP(1) Fetch	
MVIM	DR; $(MAR) \leftarrow (PC), (PC) \leftarrow (PC) + 1,$	7
	$(I/O) \leftarrow 1$	
	JMP Read, *	
	RC; $(D) \leftarrow (DP), (I/O) \leftarrow 0$	
	DR; $(MAR) \leftarrow (HL)$	
	PULSE SET OPL	
	JMP Write *	
	JMP(1) Fetch	
MVIIM	DR; $(MAR) \leftarrow (PC), (PC) \leftarrow (PC) + 1,$	13
	$(I/O) \leftarrow 1$	
	JMP Read, *	
	RC; $(W) \leftarrow (DP), (I/O) \leftarrow 0$	
	DR; $(MAR) \leftarrow (PC), (PC) \leftarrow (PC) + 1,$	
	$(I/O) \leftarrow 1$	
	JMP Read, *	
	RC; $(M) \leftarrow (DP), (I/O) \leftarrow 0$	
	DR; $(MAR) \leftarrow (PC), (PC) \leftarrow (PC) + 1,$	
	$(I/O) \leftarrow 1$	
	JMP Read, *	

	RC; (D) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0	
	DR; (MAR) $\leftarrow$ (WM)	
	PULSE SET OPI	
	JMP Write, *	
	JMP(1) Fetch	
INRr	RC; ( $r_1$ ) $\leftarrow$ ( $r_1$ ) + 1	2
	JMP(1) Fetch	
DCRr	RC; (D) $\leftarrow$ ( $r_1$ )	3
	RC; ( $r_1$ ) $\leftarrow$ (D) + $\overline{\text{zero}}$	
	JMP(1) Fetch	
INRM	DR; (MAR) $\leftarrow$ (HL), (I/O) $\leftarrow$ 1	6
	JMP Read, *	
	RC; (D) $\leftarrow$ (DP) + 1, (I/O) $\leftarrow$ 0	
	PULSE SET OPI	
	JMP Write, *	
	JMP(1) Fetch	
DCRM	DR; (MAR) $\leftarrow$ (HL), (I/O) $\leftarrow$ 1	6
	JMP Read, *	
	RC; (D) $\leftarrow$ (DP) + $\overline{\text{zero}}$ , (I/O) $\leftarrow$ 0	
	PULSE SET OPI	
	JMP Write, *	
	JMP(1) Fetch	
INRIM	DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1,	12
	(I/O) $\leftarrow$ 1	
	JMP Read, *	
	RC; (W) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0	

```

DR; (MAR) ← (PC), (PC) ← (PC) + 1
(I/O) ← 1
JMP Read, *
RC; (M) ← (DP), (I/O) ← 0
DR; (MAR) ← (WM), (I/O) ← 1
JMP Read, *
RC; (D) ← (DP) + 1, (I/O) ← 0
PULSE SET OPl
JMP Write, *
JMP(1) Fetch
DCRIM      Same microcode as above except          12
           that line DCRIM + 8 is as follows:
RC; (D) ← (DP) + zero, (I/O) ← 0
INXdd      DR; (dd) ← (dd) + 1                      2
           JMP(1) Fetch
DCXdd      DR; (dd) ← (dd) - 1                      2
           JMP(1) Fetch
INXM       DR; (MAR) ← (HL), (HL) ← (HL) + 1,      12
           JMP Read, *
           RC; (M) ← (DP) + 1, (I/O) ← 0
           DR; (MAR) ← (HL), (HL) ← (HL) - 1,
           (I/O) ← 0
           JMP Read, *
           RC; (D) ← (DP) + (C), (I/O) ← 0
           PULSE SET OPl
           JMP Write, *

```

	DR; (MAR) ← (HL)	
	RC; (D) ← (M), (OPl) ← 1, no flags set	
	JMP Write, *	
	JMP(1) Fetch	
DCXM	Same microcode as above except that	12
	line DCXM + 2 is as follows:	
	RC; (M) ← (DP) + $\overline{\text{zero}}$ , (I/O) ← 0	
	and line DCXM + 5 is as follows:	
	RC; (D) ← (DP) + zero + (C), (I/O) ← 0	
INXIM	DR; (MAR) ← (PC), (PC) ← (PC) + 1,	18
	(I/O) ← 1	
	JMP Read, *	
	RC; (W) ← (DP), (I/O) ← 0	
	DR; (MAR) ← (PC), (PC) ← (PC) + 1,	
	(I/O) ← 1	
	JMP Read, *	
	RC; (M) ← (DP), (I/O) ← 0	
	remainder of code same as that of INXM	
	except substitute (WM) for (HL)	
DCIM	Same microcode as INXIM but change	18
	line DCXIM + 8 to:	
	RC; (D) ← (DP) + $\overline{\text{zero}}$ , (I/O) ← 0	
	and line DCXIM + 11 to:	
	RC; (D) ← (DP) + $\overline{\text{zero}}$ + (C), (I/O) ← 0	
LXidd	DR; (MAR) ← (PC), (PC) ← (PC) + 1,	7

```

(I/O) ← 1
JMP Read, *
RC; (ddH) ← (DP), (I/O) ← 0
DR; (MAR) ← (PC), (PC) ← (PC) + 1,
(I/O) ← 1
JMP Read, *
RC; (ddL) ← (DP), (I/O) ← 0,
no flags set
JMP (1) Fetch
MOVXdd DR; (dda) ← (ddb) 2
JMP(1) Fetch
STDDdd DR; (MAR) ← (PC), (PC) ← (PC) + 1 13
JMP Read, *
RC; (W) ← (DP), (I/O) ← 0, no flags set
DR; (MAR) ← (PC), (PC) ← (PC) + 1,
(I/O) ← 1
JMP Read, *
RC; (M) ← (DP), (I/O) ← 0, no flags set
DR; (MAR) ← (WM), (WM) ← (WM) + 1
RC; (D) ← (dd1L), (OPL) ← 1, no flags set
JMP Write, *
DR; (MAR) ← (WM)
RC; (D) ← (dd1H), (OPL) ← 1, no flags set
JMP Write, *
JMP(1) Fetch

```



LDDDDd	<p>Same microcode as for STDDdd but change</p> <p>LDDDDd + 6 through LDDDDd + 12 to:</p> <p>DR; (MAR) <math>\leftarrow</math> (WM), (WM) <math>\leftarrow</math> (WM) + 1,</p> <p>(I/O) <math>\leftarrow</math> 1</p> <p>JMP Read, *</p> <p>RC; (dd<sub>1</sub>L) <math>\leftarrow</math> (DP), (I/O) <math>\leftarrow</math> 0, no flags set</p> <p>DR; (MAR) <math>\leftarrow</math> (WM), (I/O) <math>\leftarrow</math> 1</p> <p>JMP Read, *</p> <p>RC; (dd<sub>1</sub>H) <math>\leftarrow</math> (DP), (I/O) <math>\leftarrow</math> 0, no flags set</p> <p>JMP(1) Fetch</p>	13
STIDdd	<p>DR; (MAR) <math>\leftarrow</math> (HL), (HL) <math>\leftarrow</math> (HL) + 1</p> <p>RC; (D) <math>\leftarrow</math> (dd<sub>2</sub>L), (OPl) <math>\leftarrow</math> 1, no flags set</p> <p>JMP Write, *</p> <p>DR; (MAR) <math>\leftarrow</math> (HL), (HL) <math>\leftarrow</math> (HL) - 1</p> <p>RC; (D) <math>\leftarrow</math> (dd<sub>2</sub>H), (OPl) <math>\leftarrow</math> 1, no flags set</p> <p>JMP Write, *</p> <p>JMP(1) Fetch</p>	7
LDIDdd	<p>DR: (MAR) <math>\leftarrow</math> (HL), (HL) <math>\leftarrow</math> (HL) + 1,</p> <p>(I/O) <math>\leftarrow</math> 1</p> <p>JMP Read, *</p> <p>RC; (dd<sub>2</sub>L) <math>\leftarrow</math> (DP), (I/O) <math>\leftarrow</math> 0, no flags set</p> <p>DR; (MAR) <math>\leftarrow</math> (HL), (HL) <math>\leftarrow</math> (HL) - 1, (I/O) <math>\leftarrow</math> 1</p> <p>JMP Read, *</p> <p>RC; (dd<sub>2</sub>H) <math>\leftarrow</math> (DP), (I/O) <math>\leftarrow</math> 0, no flags set</p> <p>JMP (1) Fetch</p>	7

DADdd	RC; (D) $\leftarrow$ (dd <sub>1</sub> L) RC; (L) $\leftarrow$ (D) + (L) RC; (D) $\leftarrow$ (dd <sub>1</sub> H), no flags set RC; (H) $\leftarrow$ (D) + (H) + (C) JMP(1) Fetch	5
DADIM	DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1, (I/O) $\leftarrow$ 1 JMP Read, * RC; (W) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0 DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1, (I/O) $\leftarrow$ 1 JMP Read, * RC; (M) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0 DR; (MAR) $\leftarrow$ (WM), (WM) $\leftarrow$ (WM) + 1, (I/O) $\leftarrow$ 1 JMP Read, * RC: (D) $\leftarrow$ (DP) + (L), (I/O) $\leftarrow$ 0 PULSE SET OPl JMP Write, * DR; (MAR) $\leftarrow$ (WM), (I/O) $\leftarrow$ 1 JMP Read, * RC; (D) $\leftarrow$ (DP) + (H) + (C), (I/O) $\leftarrow$ 0 PULSE SET OPl JMP Write, * JMP(1) Fetch	17

CMDH	DR; (HL) ← ( $\overline{\text{HL}}$ ) JMP (1) Fetch	2
TCMPH	DR; (HL) ← ( $\overline{\text{HL}}$ ) DR; (HL) ← (HL) + 1 JMP(1) Fetch	3
TSCM	RC; (H) ← (H) (flags set) JMP(0) S, TCSMI DR; (HL) ← ( $\overline{\text{HL}}$ ) DR; (HL) ← (HL) + 1 RC; (A) ← (H) PULSE SET A <sub>7</sub> RC; (H) ← A	3-8
TCSMI	JMP(1) Fetch	
Subroutine for left shift (HL), (C) ← m.s.b.		
LSHL	RC; (D) ← (L) RC; (L) ← (D) + (L) RC; (D) ← (H), no flags RC; (H) ← (D) + (H) + (C) JMP Return	5
RLCH	JMP Subr., LSHL RC; (L) ← (L) + (C), no flags JMP(1) Fetch	8
RALH	JMP(1) C, RBS3 JMP Subr., LSHL JMP(1) Fetch	8-9

```

RBS3      JMP Subr., LSHL
          RC; (L) ← (L) + 1
          JMP(1) Fetch

Subroutine for right shift (HL), (C) ← l.s.b.

SRHL      RC; (H) ← rotate right (H)          7-8
          JMP(1) C, RBS4
          RC; (L) ← rotate right (L)
          EMIT; (D) ← 011111112
          RC; (L) ← (D) ∧ (L)

RBS5      RC; (H) ← (D) ∧ (H)
          JMP Return

RBS4      RC; (L) ← rotate right (L)
          EMIT; (A) ← 100000002
          RC; (L) ← (A) ∨ (L)
          JMP(0) RBS5

RRCH      JMP Subr., SRHL                    10-12
          JMP(0) C, ADRI
          RC; (H) ← (A) ∨ (H)

ADRI      JMP(1) Fetch

RARH      JMP(0) C, RARH                    10-12
          JMP Subr., SRHL
          RC; (H) ← (A) ∨ (H)
          JMP(1) Fetch

RARHL     JMP Subr., SRHL
          JMP(1) Fetch

```

Microprogram Routine to perform the double word compare between (HL)  
and (WM)

CMHW	RC; (H) $\leftarrow$ (H)	11-21
	JMP(O) S, BOB2	
	RC; (W) $\leftarrow$ (W)	
	JMP(O) S, BOB1	
	JMP(O) CMPROU	
BOB1	Pulse Reset C	
	JMP(1) Fetch	
BOB2	RC; (W) $\leftarrow$ (W)	
	JMP(O) S, SMPROU	
	PULSE SET C	
	JMP(1) Fetch	
CMPROU	RC; (D) $\leftarrow$ (L)	
	RC; (M) $\leftarrow$ (D) - (M)	
	JMP(1) Z, CMPROUI	
	RC; (D) $\leftarrow$ (H), no flags	
	RC; (W) $\leftarrow$ - (W), no flags	
	RC; (W) $\leftarrow$ (D) + (W) + (C)	
	RC; (M) $\leftarrow$ Zero V (M)	
	JMP(1) Fetch	
CMPROUI	RC; (D) $\leftarrow$ (H), no flags	
	RC; (W) $\leftarrow$ -(W), no flags	
	RC; (W) $\leftarrow$ (D) + (W) + (C)	
	JMP(1) Fetch	

CMHDX	DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1, (I/O) $\leftarrow$ 1 JMP Read, * RC; (W) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0 DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1, (I/O) $\leftarrow$ 1 JMP Read, * RC; (M) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0 JMP(O) CMHW	18-28
CMHdd	DR; (WM) $\leftarrow$ (dd) JMP(O) CMHW	13-23
CMHDM	DR; (MAR) $\leftarrow$ (XY), (XY) $\leftarrow$ (XY) + 1, (I/O) $\leftarrow$ 1 JMP Read, * RC; (M) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0 DR; (MAR) $\leftarrow$ (XY), (XY) $\leftarrow$ (XY) - 1, (I/O) $\leftarrow$ 1 JMP Read, * RC; (W) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0 JMP(O) CMHW	18-28
LDA	DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1, (I/O) $\leftarrow$ 1 JMP Read, * RC; (W) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0 DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1, (I/O) $\leftarrow$ 1	10

```

        JMP Read, *
RC; (M) ← (DP), (I/O) ← 0
DR; (MAR) ← (WM), (I/O) ← 1
        JMP Read, *
RC; (A) ← (DP), (I/O) ← 0
        JMP(1) Fetch
STA     DR; (MAR) ← (PC), (PC) ← (PC) + 1,          10
        (I/O) ← 1
        JMP Read, *
RC; (W) ← (DP), (I/O) ← 0
DR; (MAR) ← (PC), (PC) ← (PC) + 1,
        (I/O) ← 1
        JMP Reat, *
RC; (M) ← (DP), (I/O) ← 0
DR; (MAR) ← (WM)
RC; (D) ← (A), (OPl) ← 1
        JMP Write, *
        JMP(1) Fetch
CMA     RC; (A) ← ( $\bar{A}$ )                                2
        JMP(1) Fetch
CIA     RC; (A) ← zero + zero                          2
        JMP(1) Fetch
STAXEF DR; (MAR) ← (EF)                                4
        RC; (D) ← (A), (OPl) ← 1
        JMP Write, *
        JMP(1) Fetch

```

LDAXEF	DR; (MAR) $\leftarrow$ (EF), (I/O) $\leftarrow$ 1 JMP Read, * RC; (A) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0 JMP(1) Fetch	4
RLC	RC; (A) $\leftarrow$ rotate right (A) JMP(1) Fetch	2
RRC	RC; (A) $\leftarrow$ rotate left (A) JMP(1) Fetch	
RAL	JMP(0) C, RALL RC; (A) $\leftarrow$ rotate left (A) PULSE SET $A_0$ JMP(1) Fetch	4
RALL	RC; (A) $\leftarrow$ rotate left (A) PULSE Reset $A_0$ JMP(1) Fetch	
RAR	JMP(0) C, RARL RC; (A) $\leftarrow$ rotate right (A) PULSE SET $A_7$ JMP(1) Fetch	4
RARL	RC; (A) $\leftarrow$ rotate right (A) PULSE Reset $A_7$ JMP(1) Fetch	
ADD <sub>r</sub>	RC; (A) $\leftarrow$ (A) + ( $r_2$ ) JMP(1) Fetch	2
SUB <sub>r</sub>	RC; (A) $\leftarrow$ (A) + ( $\overline{r_2}$ ) + 1 JMP(1) Fetch	2



ANAr	RC; $(A) \leftarrow (A) \wedge (r_2)$ JMP(1) Fetch	2
XRAr	RC; $(A) \leftarrow (A) \nabla (r_2)$ JMP(1) Fetch	2
ORAr	RC; $(A) \leftarrow (A) \vee (r_2)$ JMP(1) Fetch	2
CMPr	RC; $(W) \leftarrow (A) + (\overline{r_2}) + 1$ JMP(1) Fetch	2
ADD M	DR; $(MAR) \leftarrow (HL), (I/O) \leftarrow 1$ JMP Read, * RC; $(A) \leftarrow (A) + (DP), (I/O) \leftarrow 0$ JMP(1) Fetch	4
SUBM	DR; $(MAR) \leftarrow (HL), (I/O) \leftarrow 1$ JMP Read, * RC; $(A) \leftarrow (A) + (\overline{DP}) + 1, (I/O) \leftarrow 0$ JMP(1) Fetch	4
ANA M	DR; $(MAR) \leftarrow (HL), (I/O) \leftarrow 1$ JMP Read, * RC; $(A) \leftarrow (A) \wedge (DP), (I/O) \leftarrow 0$ JMP(1) Fetch	4
XRA M	DR; $(MAR) \leftarrow (HL), (I/O) \leftarrow 1$ JMP Read, * RC; $(A) \leftarrow (A) + (DP), (I/O) \leftarrow 0$ JMP(1) Fetch	4
ORA M	DR; $(MAR) \leftarrow (HL), (I/O) \leftarrow 1$ JMP Read, *	4

RC; (A)  $\leftarrow$  (A) V (DP), (I/O)  $\leftarrow$  0

JMP(1) Fetch

CMPM

DR; (MAR)  $\leftarrow$  (HL), (I/O)  $\leftarrow$  1

4

JMP Read, \*

RC; (W)  $\leftarrow$  (A) + ( $\overline{\text{DP}}$ ) + 1,

(I/O)  $\leftarrow$  0

JMP(1) Fetch

ADI, SBI, ANI, XRI, ORI and CPI have the same microcode as ADD M, SUB M, ANA M, XRA M, ORA M, CMP M except that the first line of each should be changed to:

DR; (MAR)  $\leftarrow$  (PC), (PC)  $\leftarrow$  (PC) + 1,

(I/O)  $\leftarrow$  1

CALL

DR; (MAR)  $\leftarrow$  (SP) - 1, (SP)  $\leftarrow$  (SP) - 1

14

RC; (D)  $\leftarrow$  (PCH), (OFl)  $\leftarrow$  1, no flags

JMP Write, \*

DR; (MAR)  $\leftarrow$  (SP) - 1, (SP)  $\leftarrow$  (SP) - 1

RC; (D)  $\leftarrow$  (PCL), (OFl)  $\leftarrow$  1, no flags

JMP Write, \*

DR; (MAR)  $\leftarrow$  (PC), (PC)  $\leftarrow$  (PC) + 1,

(I/O)  $\leftarrow$  1, no flags

JMP Read, I

RC; (W)  $\leftarrow$  (DP), (I/O)  $\leftarrow$  0, no flags

DR; (MAR)  $\leftarrow$  (PC), (I/O)  $\leftarrow$  1

JMP Read, \*

RC; (M)  $\leftarrow$  (DP), (I/O)  $\leftarrow$  0, no flags

	DR; (PC) ← (WM)	
	JMP(1) Fetch	
RET	DR/ (MAR) ← (SP), (SP) ← (SP) + 1,	7
	(I/O) ← 1	
	JMP Read, *	
	RC; (PCL) ← (DP), (I/O) ← 0, no flags	
	DR; (MAR) ← (SP), (SP) ← (SP) + 1,	
	(I/O) ← 1	
	JMP Read, *	
	RC; (PCH) ← (DP), (I/O) ← 0, no flags	
	JMP(1) Fetch	
PUSH A	DR; (MAR) ← (SP) - 1, (SP) ← (SP) - 1	4
	RC; (D) ← (A), (OPl) ← 1, no flags	
	JMP Write, *	
	JMP(1) Fetch	
POP A	DR; (MAR) ← (SP), (SP) ← (SP) + 1,	4
	(I/O) ← 1	
	RC; (A) ← (DP), (I/O) ← 0, no flags	
	JMP(1) Fetch	

The following microcode segment is common to all conditional jump routines.

JPl	DR; (PC) ← (PC) + 1
	DR; (PC) ← (PC) + 1
	JMP(1) Fetch

The unconditional JMP  $\langle B_2 \times B_3 \rangle$  is equivalent to the LXI PC  $\langle B_2 \times B_3 \rangle$

machine language instruction which has already been microprogrammed.

JC	JMP(0) C, JPL	4-8
	DR; (MAR) $\leftarrow$ (PC), (WM) $\leftarrow$ (PC) + 1,	
	(I/O) $\leftarrow$ 1	
	JMP Read, *	
	RC; (PCH) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0, no flags	
	DR; (MAR) $\leftarrow$ (WM), (I/O) $\leftarrow$ 1	
	JMP Read, *	
	RC; (PCL) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0, no flags	
	JMP(1) Fetch	

The remainder of the conditional jump routines have the same microcode as in JC, except for the first line of code in each.

PUSHdd	DR; (MAR) $\leftarrow$ (SP) - 1, (SP) $\leftarrow$ (SP) - 1	7
	RD; (D) $\leftarrow$ (dd <sub>1</sub> H), (OPL) $\leftarrow$ 1, no flags	
	JMP Write, *	
	DR; (MAR) $\leftarrow$ (SP) - 1, (SP) $\leftarrow$ (SP) - 1	
	RC; (D) $\leftarrow$ (dd <sub>1</sub> L), (OPL) $\leftarrow$ 1, no flags	
	JMP Write, *	
	JMP(1) Fetch	
POPdd	DR; (MAR) $\leftarrow$ (SP), (SP) $\leftarrow$ (SP) + 1,	7
	(I/O) $\leftarrow$ 1	
	JMP Read, *	
	RC; (dd <sub>1</sub> L) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0, no flags	
	DR; (MAR) $\leftarrow$ (SP), (SP) $\leftarrow$ (SP) + 1, (I/O) $\leftarrow$ 1	
	JMP Read, *	

	RC; (dd <sub>1</sub> H) ← (DP), (I/O) ← 0, no flags	
	JMP(1) Fetch	
XTHL	DR; (WM) ← (HL)	12
	DR; (MAR) ← (SP), (SP) ← (SP) + 1,	
	(I/O) ← 1	
	JMP Read, *	
	RC; (L) ← (DP), (I/O) ← 0, no flags	
	RC; (D) ← (M), (OPl) ← 1, no flags	
	JMP Write, *	
	DR; (MAR) ← (SP), (SP) ← (SP) - 1,	
	(I/O) ← 1	
	JMP Read, *	
	RC; (H) ← (DP), (I/O) ← 0, no flags	
	RC; (D) ← (W), (OPl) ← 1, no flags	
	JMP Write, *	
	JMP(1) Fetch	
SMB	DR; (MAR) ← (HL), (I/O) ← 1	7
	JMP Read, *	
	RC; (A) ← (DP), (I/O) ← 0, no flags	
	PULSE SET A <sub>B</sub>	
	RC; (D) ← (A), (OPl) ← 1, no flags	
	JMP Write, *	
	JMP(1) Fetch	
RMB	The RMB routine has the same microcode as	7
	the SMB routine except that line RMB + 3	

should be: PULSE Reset  $A_B$

SMBI      DR; (MAR)  $\leftarrow$  (PC), (PC)  $\leftarrow$  (PC) + 1,      13  
           (I/O)  $\leftarrow$  1  
           JMP Read, \*  
           RC; (W)  $\leftarrow$  (DP), (I/O)  $\leftarrow$  0, no flags  
           DR; (MAR)  $\leftarrow$  (PC), (PC)  $\leftarrow$  (PC) + 1,  
           (I/O)  $\leftarrow$  1  
           JMP Read, \*  
           RC; (M)  $\leftarrow$  (DP), (I/O)  $\leftarrow$  0, no flags  
           DR; (MAR)  $\leftarrow$  (WM), (I/O)  $\leftarrow$  1  
           JMP Read, \*  
           RC; (A)  $\leftarrow$  (DP), (I/O)  $\leftarrow$  0, no flags  
           PULSE SET  $A_B$   
           RC; (D)  $\leftarrow$  (A), (OPl)  $\leftarrow$  1, no flags  
           JMP Write, \*  
           JMP(1) Fetch

RMBI      The RMBI routine has the same microcode      13  
           as the SMBI routine except that line RMBI  
           + 9 should be: PULSE Reset  $A_B$

TMB      DR; (MAR)  $\leftarrow$  (HL), (I/O)  $\leftarrow$  1      6  
           JMP Read, \*  
           RC; (A)  $\leftarrow$  (DP), (I/O)  $\leftarrow$  0, no flags  
           JMP(1)  $A_B$ , TMB2  
           PULSE Reset C  
           JMP(1) Fetch

TMB2	PULSE Set C	
	JMP(1) Fetch	
TMBI	DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1, (I/O) $\leftarrow$ 1 JMP Read, * RC; (W) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0, no flags DR; (MAR) $\leftarrow$ (PC), (PC) $\leftarrow$ (PC) + 1, (I/O) $\leftarrow$ 1 JMP Read, * RC; (M) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0, no flags DR; (MAR) $\leftarrow$ (WM), (I/O) $\leftarrow$ 1 JMP Read, * RC; (A) $\leftarrow$ (DP), (I/O) $\leftarrow$ 0, no flags JMP(1) A <sub>B</sub> , TMBI2 PULSE Reset C JMP(1) Fetch	12
TMBI2	PULSE Set C JMP(1) Fetch	
SBA	PULSE Set A <sub>B</sub> JMP(1) Fetch	2
RBA	PULSE Reset A <sub>B</sub> JMP(1) Fetch	2
SIIM	DR; (MAR) $\leftarrow$ (HL) PULSE OP2 $\leftarrow$ pulse (multiplexed to OCP3) JMP(1) Fetch	3

RIIM	DR; (MAR) ← (HL) PULSE OP2 ← pulse (multiplexed to OCP2) JMP(1) Fetch	3
HLT	JMP(0) Pri. Int., * PULSE DI PULSE EI JMP(1) Fetch	4
NOP	JMP(1) Fetch	1
EI	PULSE EI JMP(1) Fetch	2
DI	PULSE DI JMP(1) Fetch	2
STC	PULSE Set C JMP(1) Fetch	2
CFL	RC; (W) ← (A), no flags RC; (A) ← (PSW), no flags PULSE Comp. A <sub>0</sub> RC; (PSW) ← (A), no flags RC; (A) ← (W), no flags JMP(1) Fetch	6



## APPENDIX F

## INTEL 8080 PROCESS CONTROL ROUTINES

The following 8080 machine language routines are the approximate equivalent of corresponding instructions found in the GT 1248 instruction set.

Set Memory Bit (SMB) Instruction

The bit addressed by the contents of double register HL is set to a logic one.

Memory Location	Mnemonic	Action
SMB	MOV A,M	(A) ← word containing the bit
	XCHG	(DE) ← word address
	MOV A,E	
	ANI<00000111>	
	MOV L,A	(HL) ← bit position
	XRA A	
	MOV H,A	
	LXIB<Base>	(BC) ← Base
	DAD B	(HL) ← vector address
	PCHL	Jump to vector
	JMP ABITO	
	JMP ABITI	
Base		

	JMP ABIT2	
	JMP ABIT3	
	JMP ABIT4	vectors
	JMP ABIT5	
	JMP ABIT6	
	JMP ABIT7	
ABIT0	ORI<00000001>	
	JMP BOB	
ABIT1	ORI<00000010>	
	JMP BOB	set desired bit
ABIT2	ORI<00000100>	
	JMP BOB	
ABIT3	ORI<00001000>	
	JMP BOB	
ABIT4	ORI<00010000>	
	JMP BOB	
ABIT5	ORI<00100000>	
	JMP BOB	
ABIT6	ORI<01000000>	
	JMP BOB	
ABIT7	ORI<10000000>	
BOB	XCHG	(HL) ← word address
	MOV M,A	[(HL)] ← (A)
	RET	Return to main program

Set Memory Bit Immediate (SMBI) Instruction

The bit addressed by an immediate address is set to a logic one.

The following code could be written into the control program.

Memory Location	Mnemonic	Action
	LXI H <Adr.>	(HL) is loaded with the immediate address
	CALL SMB	Call the SMB Subroutine

Test Memory Bit (TMB) Instruction

The bit addressed by the contents of double register HL is tested.

If  $b_{HL} = 0$ , then  $(Z) \leftarrow 1$ .

Memory Location	Mnemonic	Action
TMB	MOV A,M	(A) ← word containing the bit
	MOV A,L	
	ANI<00000111>	
	MOV L,A	(HL) ← bit position
	XRA A	
	MOV, H,A	
	LXI B <Basel>	(BC) ← Basel
	DAD B	(HL) ← vector address
	PCHL	Jump to vector
Basel	JMP BBIT0	
	JMP BBIT1	
	JMP BBIT2	
	JMP BBIT3	

```

                                JMP BBIT4                vectors
                                JMP BBIT5
                                JMP BBIT6
                                JMP BBIT7
BBIT0                          ANI<00000001>
                                RET
BBIT1                          ANI<00000010>
                                RET                set (Z) and return
                                .
                                .
                                .
                                .
BBIT7                          ANI<10000000>
                                RET

```

#### Test Memory Bit Immediate (TMBI) Instruction

The bit addressed by an immediate address is tested as in the TMB instruction.

Memory Location	Mnemonic	Action
	LXI H <Adr.>	(HL) is loaded with the immediate address
	CALL TMB	Call the TMB Subroutine

#### Set A Register Bit (SBA) Instruction

The desired bit of the A register, specified by the contents of double register HL, is set to logic one. All other bits are unchanged.

Memory Location	Mnemonic	Action
SBA	LXI D <Base2>	(DE) ← Base2
	DAD D	(HL) ← vector address
	PCHL	Jump to vector
Base2	JMP BIT0	
	JMP BIT1	
	JMP BIT2	
	JMP BIT3	vectors
	JMP BIT4	
	JMP BIT5	
	JMP BIT6	
	JMP BIT7	
BIT1	ORI<00000001>	
	RET	
	.	
	.	set desired bit and return
BIT7	.	
	.	
	ORI<10000000>	
	RET	

#### Set Individual Interrupt Mask (SIIM) Instruction

The address of a specified interrupt mask is sent out over the address bus and the OCP3 control line is pulsed. The 8080 pulses the  $\overline{\text{WR}}$  line during a write operation and this line is normally connected to the OCP1 control line. The SIIM instruction must temporarily connect

$\overline{WR}$  to OCP3 and pulse this line while the address of the mask is valid.

This short section of code could probably best be written inline in the main program.

Memory Location	Mnemonic	Action
	OUT <OCP3>	$\overline{WR}$ is connected to OCP3
	STA, Address	Mask is set
	OUT <OCP1>	$\overline{WR}$ is connected to OCP1

## APPENDIX G

## ROBOT CONTROL PROGRAMS

Main Control Routine

The following routine is divided into the steps presented in Figure 21.

Number	GT 1248		Intel 8080	
	Location	Mnemonic	Location	Mnemonic
12	UPDATE	CIA	UPDATE	XRA A
		STA,FLAG		STA,FLAG
13		IWXI,PSW		LHLD,PSW
				IWX H
				SHLD,PSW
14		CMHDX<end> JZ, End Routine		MOV A,H
				SUI<end H>
				JNZ, Compute
				MOV A,L
				SUI<end L>
15		DAD HL	COMPUTE	JZ, End Routine
				DAD H
				DAD H
				DAD H
		DAD HL		DAD H

		LDDD EF,PEW		XCHG	
		DAD EF		LHLD,PEW	
		STDD HL,CPS		DAD D	
				SHLD,CPS	
1	START	LDA,FLAG	START	LDA,FLAG	
		JZ, UPDATE		ORA A	
				JZ, UPDATE	
2		LDDD HL,IOB		LHLD,IOB	
		DADI,CFW		XCHG	
				LHLD,CFW	
				DAD D	
3		LDID EF		MOV C,M	
				INX H	
				MOV A,M	
4	CONVERT	CONV (see	CONVERT	MVI D,<10000000>	
		microprogram		RLC	
		for the convert		RLC	Written
		instruction follow-		JNC,*+2	Inline Six
		ing this section)		ADD D	Times
				MOV B,A	
				ANA D	
				ADD C	
				RLC	Written
				JNC,*+2	Inline Six
				ADD D	Times



			RLC
			MOV C,A
			MOV A,B
			ANI<01000000>
			JZ, WWW
			MOV A, B
			ANI<10111111>
			CMA
			MOV B,A
			MOV A,C
			CMA
			MOV C,A
			INX B
5	LDDD HL,CFW	WWW	LHLD,CPS
	DADI,CPS		XCHG
	LDID XY		LHLD,CFW
	MOVX HL,XY		DAD D
			MOV E,M
			INX H
			MOV D,M
			XCHG
6	DAD EF		DAD B
7	TCSM		MOV A,H
			ANI<01000000>
			JZ, POSITIVE
			MOV A,H

			CMA
			MOV H,A
			MOV A,L
			CMA
			MOV L,A
			INX H
8	STDD HL, DAA	POSITIVE	SHLD,DAA
	LXI HL, P1		LXI H, P1
	SMB		CALL SMB
	RMB		CALL RMB
9	LDA,CFW		LDA,CFW
	INR A		INR A
	IWR A		IWR A
	IWRI M, OF		LXI H, OF
			IWR M
10	CPI <15>		CPI <15>
	JNC *+2		JC *+2
11	CLA		XRA A
	STA,CFW		STA,CFW
	STA,OF		STA,OF
	JMP,START		JMP,START

Microprogram for the CONV Instruction

---

Symbolic Location

Microcode

---

CONV

EMIT; (B) ← 110<sub>2</sub>

[illegible]

```

                                DR; (EF) ← (EF)
                                DR; (EF) ← (EF) + 1
                                JMP(1), Fetch
COWV1                          JMP(1) AB, *+3
                                RC; (B) ← (B) - 1
                                JMP(0), *+3
                                RC; (B) ← (B) - 1
                                PULSE; Comp. AB
                                JMP Return

```

#### Starting Conditions for the 8080 Convert Routine

The conversion routine of step four is capable of converting a 14 bit signed Gray code to its equivalent two's complement representation so that the error quantity may be calculated. Starting conditions for the routine are with the following data in registers A and C:

	A								C						
S	G <sub>13</sub>	G <sub>12</sub>	G <sub>11</sub>	G <sub>10</sub>	G <sub>9</sub>	G <sub>8</sub>	G <sub>7</sub>	0	G <sub>6</sub>	G <sub>5</sub>	G <sub>4</sub>	G <sub>3</sub>	G <sub>2</sub>	G <sub>1</sub>	G <sub>0</sub>

and the routine ends with the two's complement answer in registers B and C as follows:

		B												C										
X	S	B <sub>13</sub>	B <sub>12</sub>	B <sub>11</sub>	B <sub>10</sub>	B <sub>9</sub>	B <sub>8</sub>	B <sub>7</sub>	B <sub>6</sub>	B <sub>5</sub>	B <sub>5</sub>	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>									

## BIBLIOGRAPHY

1. Keyes, M. A., "Distributed Digital Control," Control Engineering, September 1973, pp. 77-80.
2. Hammond, J. L., Oh, S. J., "Evolution of Systems Approaches to Computer Control in Discrete Manufacturing: A Survey," IEEE Trans. Mfg. Tech., Vol. 2, June 1973, pp. 4-11.
3. Kinberg, C., Landeck, B. W., "Integrated Manufacturing Systems: Architectural Considerations," IBM Journal Res. and Dev., Vol. 14, No. 6, November 1970, pp. 589-604.
4. Howell, R. S., "Computers In - A Look," Automation, May 1972.
5. Stuehler, J. E., Watkins, R. V., "A Computer Operated Manufacturing and Test System," IBM Journal Res. and Dev., Vol. 11, July 1967, pp. 452-460.
6. Stuehler, J. E., "An Integrated Manufacturing Process Control System: Implementation in IBM Manufacturing," IBM Journal Res. and Dev., Vol. 14, No. 6, November 1970, pp. 605-613.
7. Poisson, N. A., "Interfacing Computers to Production Equipment," Automation, May 1972.
8. Calva, J. R., "PCOS: A Process Control Extension to Operating System /360," IBM Journal Res. and Dev., Vol. 14, No. 6, November 1970, pp. 620-632.
9. Thoburn, F. W., "A Transmission Control Unit for High Speed Computer-to-computer Communication," IBM Journal Res. and Dev., Vol. 14, No. 6, November 1970, pp. 614-619.
10. Harrison, T. J., Homiak, R. L., Merckel, G. U., "IBM System /7 and Plant Automation," IBM Journal Res. and Dev., Vol. 14, No. 6, November 1970, pp. 652-660.
11. Gaines, N. W., Greenacre, G. R., Harris, B. J., O'Neil, W. E., Banks, R. S., Olson, W. R., Rammell, G. A., Romeu, F. J., "Union Carbide Integrates Multi-Computer Process Control," Instrumentation Technology, March 1967, pp. 49-51.
12. Rispole, L. M., "Hierarchical Computer Control Systems," Instruments and Control Systems, Part 1, October 1970, pp. 117-119, Part 2, November 1970, pp. 116-119.

13. Alford, C. O., "Design Aspects of Computer Control in Discrete Manufacturing," IEEE Trans. Mfg. Tech., Vol. 2, December 1973, pp. 26-36.
14. Nilsen, R. N., "Distributed Function Computer Architectures," Computer, Vol. 7, No. 3, March 1974, pp. 15-16.
15. Programmers Reference Manual, Model 960 Process Control Computer, Texas Instruments Inc. manual, Part. No. 214091-9701.
16. Shuraym, G., "Why the 960 Computer for Industrial Automation," Texas Instruments Inc. manual.
17. "Industrial Automation Simplified with the Communications Register Unit," Texas Instruments Inc. brochure.
18. Reyling, G., "Considerations in Choosing a Microprogrammable Bit-Sliced Architecture," Computer, Vol. 7, No. 7, July 1974, pp. 26-29.
19. Husson, S. S., Microprogramming: Principles and Practices, Prentice Hall, Inc., 1970, pp. 13-38.
20. The Intel Memory Design Handbook, Intel Corporation, 1973, pp. 1-9.
21. Signetics Digital 54/7400 Data Book, Signetics Corporation, 1972, pp. 111-112.
22. "Design of a General Purpose Microcontroller," Application Note, Scientific Micro Systems, Inc., July 1973.
23. Donovan, J. L., Systems Programming, McGraw-Hill Inc., 1972, pp. 395-400.
24. Unimate information gained through personal correspondence with Mr. M. J. Dunne, Chief Engineer, Unimation Inc.