1978

A Virtualizable MACHine for Multiprogrammed Operation Based on Non-Virtualizable Microprocessors

William D. Armitage
University of Rhode Island

Follow this and additional works at: <https://digitalcommons.uri.edu/theses>

Recommended Citation

Armitage, William D., "A Virtualizable MACHine for Multiprogrammed Operation Based on Non-Virtualizable Microprocessors" (1978). *Open Access Master's Theses*. Paper 1390.
<https://digitalcommons.uri.edu/theses/1390>

This Thesis is brought to you for free and open access by DigitalCommons@URI. It has been accepted for inclusion in Open Access Master's Theses by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu.

A VIRTUALIZABLE MACHINE FOR MULTIPROGRAMMED
OPERATION BASED ON NON-VIRTUALIZABLE
MICROPROCESSORS

BY

WILLIAM D. ARMITAGE

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

Approved:

Thesis Committee

Major Professor

UNIVERSITY OF RHODE ISLAND

1978

UNIVERSITY OF RHODE ISLAND

1978

MASTER OF SCIENCE THESIS

OF

WILLIAM D. ARMITAGE

Approved:

Thesis Committee

Major Professor

Nelson Weidman

Edmund A. Lamagna

Robert B. Kelly

R. A. Michel

Dean of the Graduate School

UNIVERSITY OF RHODE ISLAND

1978

ABSTRACT

Microcomputers are proliferating in dedicated applications and as single-user general-purpose digital computers. Many common applications on larger machines are inherently multi-user and require a multiprogrammed mode of operation. Multiprogrammed operating systems, although desirable for this reason and to maximize utilization of expensive system components, have not yet been satisfactorily implemented on microcomputers. It is shown that a typical microprocessor -- the Intel 8080 -- is inherently incapable of supporting a multiprogrammed operating system due to a lack of any privileged instruction set whatsoever. Other disadvantages of microprocessor-based systems that affect their capability for multiprogramming are discussed, including the limited memory address space, lack of relocation aids and lack of a "test and set" instruction for synchronization purposes. A machine architecture is proposed that utilizes two or more 8080s in a master/slave relationship to effectively implement a privileged instruction set. The architecture is shown to be virtualizable -- that is, capable of supporting a virtual machine monitor -- and to have good storage protection and fault-tolerance characteristics. A "Dynamic Memory Banking" system is included in the architecture that relieves the 64K limitation on memory

resources, makes program relocation unnecessary and allows the assignment of memory to whatever process requires it at whatever address. This memory system simplifies problems involved in implementation of virtual storage; the central concepts are applicable to larger machines as well. Required and optional aspects of operating system software for the proposed architecture are discussed and specific suggestions for implementation are made.

1. Computer architecture summary	1
1.1 Intent of the study	1
2. MULTIPROGRAMMING AND MULTIPROCESSING	10
2.1 Multiprogramming	10
2.2 Multiprocessing	10
3. THE CONCEPT OF VIRTUAL STORAGE	19
3.1 Support of virtual storage monitoring	19
3.2 Hardware requirements	20
3.3 Summary	21
4. A TYPICAL MICROPROCESSOR: INTEL 8080	23
4.1 Architecture and instruction set	24
4.2 The S-100 bus	27
4.3 Multiprogramming and the 8080	27
4.4 Current multi-user systems	28
5. PROPOSED MACHINE ARCHITECTURE	30
5.1 Supervisory hardware changes	32
5.2 Dynamic memory location system	32
5.3 Architectural modifications	33
5.4 Characterization of the machine	33
6. OPERATING SYSTEMS SOFTWARE	34
6.1 Required functions	35
6.2 Optional functions	35
6.3 Summary	36
7. CONCLUSIONS AND EXTENSIONS	38
7.1 Conclusions	38
7.2 Recommendations for future work	39
7.3 Closing remarks	39

TABLE OF CONTENTS

ABSTRACT	ii
LIST OF ILLUSTRATIONS	vi
Chapter	
1. INTRODUCTION	1
1.1 Machine evolution	2
1.2 Computer architecture economics	5
1.3 Intent of the study	8
2. MULTIPROGRAMMING AND MULTIPROCESSING	10
2.1 Multiprogramming	10
2.2 Multiprocessing	24
3. THE CONCEPT OF VIRTUALIZABILITY	29
3.1 Support of Virtual Machine Monitor (VMM)	32
3.2 Hardware requirements	39
3.3 Summary	43
4. A TYPICAL MICROPROCESSOR: INTEL 8080	45
4.1 Architecture and instruction set	46
4.2 The S-100 bus	51
4.3 Multiprogramming and the 8080	52
4.4 Current multi-user systems	55
5. PROPOSED MACHINE ARCHITECTURE	62
5.1 Supervisory computer concept	62
5.2 Dynamic Memory Banking system	66
5.3 Architectural specifications	72
5.4 Characteristics of this architecture	98
6. OPERATING SYSTEMS SOFTWARE	118
6.1 Required functions	119
6.2 Optional functions	125
6.3 Summary	129
7. CONCLUSIONS AND EXTENSIONS	130
7.1 Conclusions	130
7.2 Recommendations for future work	131
7.3 Closing remarks	133

REFERENCES	135
APPENDIX A. INTEL 8080 INSTRUCTION SET SUMMARY	138
APPENDIX B. S-100 BUS LINE DESIGNATIONS	141

Figure	
1. Crayon Multi-user System	57
2. Example of Storage Fragmentation	67
3. Module Interrelationships and Busses	71
4. Intuitive "Snapshot" of Dynamic Resource Allocation System	81
5. Input Driver Flowchart	97

LIST OF ILLUSTRATIONS

Figure		
1.	Cromemco Multi-user System	57
2.	Example of Storage Fragmentation	67
3.	Module Interrelationships and Busses	73
4.	Intuitive "Snapshot" of Dynamic Memory Banking System	83
5.	Input Driver Flowchart	127

CHAPTER 1

INTRODUCTION

For several years, the coming of the "age of the personal computer" has been heralded. Due to rapidly dropping prices for computer components brought on by large scale integration (LSI) of circuitry on miniature silicon wafers, forecasts include continued reductions in the cost of general purpose computers as well as a vast range of new, dedicated applications [Hilburn 1976]. In three years, the price of an INTEL 8080 -- a typical LSI Central Processing Unit (CPU) -- has dropped from over \$200 to less than \$12 in quantities of one. Although future reductions in price of a single CPU of the same order of magnitude as the \$200 to \$12 drop are not likely, it is expected that an effective reduction of the same or greater order is almost inevitable through the inclusion of more and more functions (such as I/O and main memory) on the single chip containing the CPU. This will greatly reduce the costs involved in implementing the support circuitry required by the CPU.

It is natural for persons either not particularly involved with computers or involved from a user point of view to assume that microcomputers, having been developed after large third generation machines, have all their capabilities excepting, perhaps, raw speed. This would be a

mistake. In some ways they are a "throwback" to earlier machines. To understand their limitations and the reasons for them, a brief review of pertinent aspects of computer evolution is useful.

1.1 Machine evolution

The primary "evolutionary force" in computer systems design has been the advancement in solid state physics [Osborne 1975]. The earliest stored-program computers were essentially one-of-a-kind designs based on vacuum tube circuitry -- the "state of the art" of the day. They were slow and consumed impressive amounts of electrical power. Programs were written strictly in machine language, usually "toggled in" through the front panel switches or read in from punched cards. They were successes nonetheless, as no combination of man and desk calculator could approach their raw "number-crunching" power. Due to their construction, of course, they were extremely expensive. For primarily this reason, their use was limited to applications where a high premium was put on speed of computation, or where the sheer mass of computation required made manual calculation totally impractical [Rosen 1969]. Several manufacturers introduced vacuum tube based commercial machines such as the IBM 709. These machines gradually gained acceptance, but their high cost and unreliability limited their range of practical applications.

The development of the semiconductor made the use of the computer practical for tasks without massive

computational and data-handling needs. Discrete semiconductors (transistors) and other discrete components were used to construct computers like the IBM 7070 and IBM 1401 that brought the capabilities of data processing within the economic reach of most larger businesses. Development of programming aids such as assemblers and high-level languages received much emphasis. Computer systems were still, like their ancestors, single-job machines - they ran one program at a time.

Solid-state technology continued to advance. Methods for "microminiaturizing" components were developed; these methods had great impact in the computer design field. Small and Medium Scale Integration (tens or hundreds of devices on a single "chip") made two kinds of computers possible. It has been observed that each technological advance in solid-state devices results in two directions of development - systems comparable in performance to already existing machines can be made smaller and less expensive, and machines of about the same size and cost can be developed with greater capabilities [Osborne 1975]. Microminiaturized circuitry resulted in machines like the IBM 360 - with capabilities significantly advanced from earlier machines - and, eventually, a new class of machines called minicomputers, which had the same characteristics of earlier machines but which, as implied by their name, were much smaller in physical size and also cost significantly less.

Large Scale Integration (LSI), or the combining of

the equivalent of literally thousands of components onto a single chip, brought the next stage in computer hardware development. Using LSI, it was possible to develop a complete CPU in one package. The essential features of a CPU - an Arithmetic & Logical Unit (ALU), Accumulator, Registers and Control Unit - were all included, although the chip still required significant support circuitry depending on the particular application involved. Microprocessors will undoubtedly find their most common application as intelligent controllers of devices heretofore limited to control by logic circuits designed specifically for the application or even by mechanical switching assemblies. Designers find that it is often easier to adapt a microprocessor to the task with software and a minimum of hardware than it is to design, debug and implement essentially new control circuitry. "It has been suggested that any digital system employing more than fifty gates is a candidate for application of a microcomputer" [Hilburn 1976]. Examples of these applications include microwave ovens and sewing machines, to name two applications already converted to CPU control. Heating systems, traffic signals, washing machines and many automotive systems are among those which will be seeing control by microprocessors before long.

In addition to its advantages as an inexpensive alternative to custom logic designs for controlling common devices, the microprocessor has always held promise as a general-purpose digital computer. In fact, the first 8-bit

microprocessor -- the Intel 8008 -- was contracted for by the Datapoint Corporation, a manufacturer of intelligent terminals and small computer systems [Osborn 1975]. Since 1975, when MITS, Inc., of Albuquerque, New Mexico, introduced the first microcomputer system in kit form, general purpose microcomputers have proliferated. As software -- both systems and applications -- has been developed, these new entries at the low end of the computer spectrum have enabled businesses which could not before afford a computer to automate. In addition, they have been making inroads in what used to be minicomputer areas, forcing minicomputer manufacturers to upgrade the performance and versatility of their machines to that of former medium-scale digital computers [Rao 1978].

1.2 Computer architecture economics

Large scale integration of computer CPUs and memory has resulted in a significant reversal of economic considerations in computer use.

Earlier Economics

Until rather recently, the central electronics (CPU circuitry and main memory) of a computer system accounted for most of the system's cost. On-line disk storage accounted for a significant proportion of the remainder, with mechanical peripherals making up the difference. Operating systems were written to maximize usage of the most expensive parts of the system, primarily the CPU and main memory. In earlier systems -- the IBM 1401, for example -- the CPU was

waiting for a mechanical peripheral much of the time. In heavily "I/O-bound" jobs, the CPU was idle for a very high percentage of the time. In the next generation of computers -- the IBM System 360, for example -- features were included in the design that made it possible for more than one program to be in memory and executing at the same time. In this way, one program could be waiting for a card to be read or a line to be printed while the other program was receiving CPU time. This process is called "multiprogramming"; it shall be discussed in more detail later.

Microcomputers

Large scale integration has resulted in extremely inexpensive CPUs. In addition, it has had great effect on main memory costs. Today, a 16K (bytes) memory board for a microcomputer can be purchased for less than four hundred dollars. Mechanical peripherals, on the other hand, cost close to what they did several years ago. The result is a real inversion of relative costs. It is not uncommon to find a complete microcomputer system with 64K of main memory attached to a printer that costs more than the system itself! Disk systems, with their mechanical aspects, remain expensive; a microcomputer system with a single hard disk drive may cost several times the amount of the same system without the disk.

Complicating the economic comparison of microcomputer systems with earlier computers is the fact that microcomputers, as they exist today, are primarily

single-user machines. In most cases, the user's software has complete control of the machine; no "supervisory program" exists in memory that is able to, for example, distribute CPU time between two or more programs in memory at the same time. There are two primary reasons for this. First, systems software in the form of good high-level language processors and single-user disk operating systems have only become available during 1977. It is a simple fact that good software for a new machine takes significant time and effort to develop. Multi-user systems software is more complex than single-user software, and will take more time to develop. Secondly -- and most importantly -- microprocessors in common use today do not readily support multi-user systems. This will later be shown in detail.

Multi-user capabilities desirable

The current inability of microcomputers to support multiprogramming blunts the sharp cost advantages of such a system. Multiprogramming is still desirable for the same reason it was many years ago -- maximum utilization of expensive system components. Merely the identity of the expensive components has changed. It would be advantageous, for example, if several users could share access to an expensive hard disk system, as an alternative to each user requiring his own, dedicated disk system. The same comment holds for mechanical "unit record" devices like card readers, punches and line printers. Although main memory has come down in price due to large scale integration, it is not

cheap; 64K of memory still costs approximately \$1500.00. As mentioned above, a CPU can be obtained for less than twelve dollars. Utilization of main memory is still a consideration that urges implementation of multiprogramming.

As multiprogramming has been commonly available on large machines since the mid 1960's, some applications have been developed which take advantage of shared auxiliary storage resources. Instead of merely sharing the hardware, these applications share the use of information available from auxiliary storage. This process is referred to as sharing a "common data base."

There are two ways of accomodating this class of applications on microcomputers. One method is to provide a machine controlling the auxiliary storage devices. The sole purpose of this machine would be to service requests from other machines for auxiliary storage operations. This approach involves an additional system and additional I/O interfaces (for communications between the data base machine and the machines being served). The other method, of course, is to implement a multiprogramming capability on microcomputers.

1.3 Intent of the study

Overcome limitations of microprocessors

The intent of this study, therefore, is to discuss the difficulties that would be encountered in the implementation of a multiprogramming capability on a microprocessor-based system. The concepts of multiprogram-

ming and multiprocessing will be defined and described in detail. The concept of machine "virtualizability" will be described, and the implications of the concept for multiprogramming will be discussed. The Intel 8080, a typical microprocessor, will be closely examined as to its virtualizability and inferences drawn regarding its ability to support true multiprogramming. A computer system architecture using the 8080 will be introduced and defined, and it will be shown that it meets the requirements for virtualizability and can therefore support multiprogramming. After the architecture has been defined, basic requirements for multiprogrammed operating systems software will be specified and discussed.

Main memory architecture

In the course of specifying the machine architecture, a main memory organization will have been described. It will be shown that this organization, referred to as "Dynamic Memory Banking," makes several "classical" computer science problems trivial, as well as reducing the complexity of solutions to other problems as well. Although it will have been structured for microcomputer use, it will be apparent that the basic concept is adaptable to large machines as well, providing the same benefits as those provided to a microcomputer system.

CHAPTER 2

MULTIPROGRAMMING AND MULTIPROCESSING

Before attempting to discuss multiprogramming on a microcomputer, it is essential that it be understood exactly what multiprogramming is. True multiprogramming implies certain capabilities on the part of the system -- some implemented in hardware and others in software.

As the proposed machine architecture to be described in Chapter 5 includes a provision for a "multiprocessing" capability -- in addition to multiprogramming -- it is necessary that this concept be described as well.

2.1 Multiprogramming

It is a natural tendency for users to see a computer from their point of view -- as a machine to do their job. The user may well think of the system as a strictly sequential device; it processes one job after another the same way it processes instructions in the user's program -- in consecutive fashion. Indeed it is possible for a computer to process jobs in exactly this way -- early computers were limited to this mode.

One very noticeable result of this method of processing jobs was a waste of CPU time. Mechanical peripherals, for example, were slow. When a program directed

that a card be read, the CPU's processing power was suspended while the input operation took place. For jobs requiring a large number of unit record operations, it was not unusual for the CPU to be idle well over ninety percent of the total job time. "Even when processors are kept busy most of the time, the utilization of other computer resources is often poor; for example, any main storage not occupied by the current job (and some minimal part of the operating system) is essentially a wasted resource" [Shaw 1974]. The search for a method to make use of the wasted resources of CPU time and main storage -- both very expensive commodities in those days -- was what resulted in multiprogramming.

More than one independent process

The basic concept underlying multiprogramming is maintaining more than one independent sequential process in an active state in main storage [Shaw 1974]. For the purposes of understanding, the informal definition of "process" given in Shaw is acceptable: "A sequential process (sometimes called 'task') is the activity resulting from the execution of a program with its data by a sequential processor." Examples of sequential processes (hereafter referred to as simply "processes") are jobs, programs, or even special-purpose routines in the operating system itself.

A key characteristic of a multiprogrammed mode of operation is an appearance of simultaneity in running of the processes in main memory. All jobs or programs seem to be executing at the same time. Depending on the time scale an

observer wishes to use, this simultaneity may disappear. The single-processor multiprogrammed system can still execute only one instruction at a time. The illusion of simultaneous execution of all programs is due to the rapid "multiplexing" of the CPU between the various active processes awaiting processing. The CPU may execute instructions from one process for a few milliseconds, stop and save registers and status from that process, load registers and status from a second process, and then commence executing the instructions of that second process. A few milliseconds later, a "process switch" to a third process may occur, and so on. Depending on the design philosophy of the operating system, the process switches may occur only when a process that is executing requests an I/O operation or an I/O operation requested by a higher priority process is completed. Alternately, these switches may occur whenever a timer preset by the operating system signals. The latter case is referred to as a time-slice system.

Other benefits

Although the increased utilization of expensive CPU time and main memory was by itself sufficient motivation for the implementation of multiprogramming, other benefits are also realized. Since more than one job can be entered into main memory at one time, multiple "entry points" are possible - users can enter jobs at remote job entry stations or from their own individual terminals [Shaw 1974].

In addition, hardware is not the only "resource"

that can be shared - principal software resources, such as language compilers, I/O routines and system utilities, can be utilized by more than one process in main storage. Another benefit of multiprogramming is that processing time can be scheduled; more CPU time can be allocated to higher-priority programs by the multiprogramming operating system [Shaw 1974].

Operating systems software

In the preceeding discussion, we have mentioned an "operating system" several times. Processes occupying main storage simultaneously cannot be expected to allocate CPU time and main memory resources to themselves. A usually complex piece of software is written for this purpose and essential parts of it remain present, or "resident," in main storage at all times. This software is referred to as the "operating system," and more specifically as the "multiprogrammed operating system."

For the purposes of this thesis, we shall use the term "operating system" to refer to control programs having responsibility for task (process), job and data management, and shall exclude processing programs like language translators, service programs and user programs, although a complete definition of the term may include them [Katzan 1973].

Managing resouces of the machine. As "multiprogramming involves the sharing of time on processors and the sharing of space in main storage, as well as the potential

sharing of other resources" [Shaw 1974], the prime purpose of the operating system must be to manage the resources of the system.

CPU time must be distributed among the active processes in main storage in accordance with some allocation strategy; we shall refer to the collection of operating system routines which perform this function as the "processor scheduler."

The main storage resources belonging to the system must be allocated to jobs requesting memory. Operating system routines for this purpose can be quite simple or very complex, depending on whether the allocation is to be done in one step at the beginning of a job (static) or continuously adjusted to the current needs of the jobs during their execution (dynamic).

Another group of resources that must be shared among processes on a multiprogrammed machine is the set of I/O devices attached to the system. Unit record devices, such as card readers, punches and line printers, are included in this group, as are individual user terminals. Auxiliary storage, such as disk and drum systems and magnetic tape drives, are also considered I/O devices.

Management of disk and drum systems is a large area in itself. A given disk drive or drum is often being used by several processes simultaneously; there may be literally dozens of requests for disk service queued up at one point in time. This may develop into a major system bottleneck.

It has been recognized that poor use of direct access storage is a common cause of inefficiency in multiprogrammed systems, and much work has been done in developing algorithms to make optimal use of these devices [Teorey 1972].

Isolation of users from each other. An essential aspect of a multiprogrammed system is the measure of isolation it provides between processes. Remember that the user still views the system as a sequential device, and the operation of his program must be identical to what would be expected were it to have exclusive use of the machine, with the exception of running time. If other processes that happen to be in main storage at the same time can affect in any way the operation of a user's job, the above condition is not met.

Since several processes coexist in storage, one obvious function of isolation is to prevent one process from inadvertently or willfully altering the memory belonging to any other process or the operating system itself. This function is referred to as "memory protection." It may take the form of store protection, the most common form, in which a process may "see" what is in the storage allocated to another process but is prevented from changing it. This protection is sufficient to ensure that a process is not destroyed by another process, but cannot satisfy privacy considerations. As confidential data (business and personnel records) are often temporarily in storage for processing, another process could continuously read this data from memory

and store it on its own auxiliary storage for later perusal. When memory protection is implemented such that it prevents reading of other processes' memory, it is known as "store-and-fetch protection."

Another readily understood requirement for isolation deals with I/O devices. Peripherals allocated to one process must not be used by another process until the first process has released them. One can easily imagine the irritation that would ensue if User A was to cause a printed message to appear in the middle of a long printed report being produced by User B's job!

User files on direct access auxiliary storage must also be protected in much the same way that main storage is. The privacy concerns referred to above are even more important in preventing unauthorized access to files; the owner of the file must have control over which, if any, classes of users other than himself may use or modify it [Shaw 1974].

To summarize the "management" and "isolation" functions of the multiprogramming operating system, we can say that the system must first allocate resources to processes and then enforce those allocations.

Can be simple or sophisticated in concept. It was mentioned above that storage allocation routines can be simple or very complex. This "continuum of complexity" can be carried throughout most of the operating system structure. The more sophisticated the desired allocation strategies, or the more flexible the enforcement of those allocations, the

more complex must be the operation system. The advantages of multiprogramming -- even in its simplest realization -- is not without its price, and that price increases with sophistication.

A processor scheduler, for example, can use a simple or very complex algorithm to determine which process will receive CPU time next, and how much it will receive. If it is desirable to cancel a process automatically if it exceeds some prior estimate of processing time, the scheduler must also check for this condition and be capable of taking appropriate action.

Disk schedulers perform the function of determining what disk service request is to be serviced next. In its simplest form, the scheduler will process requests on a first-come first-served (FCFS) basis. Under heavy disk request loading, however, FCFS can result in the type of system bottleneck mentioned earlier. More sophisticated policies that take into account the physical structure of the disk and even its rotational characteristics have been investigated and implemented in various systems to good advantage [Teorey 1972].

It is necessary to bear in mind that CPU time and other resources devoted to "housekeeping chores" of multiprogramming are lost to user processes. The time the CPU spends on these functions is called "overhead" [McKinney 1969]. Overhead caused by the operating system can be significant during execution [Shaw 1974]. It is well to

remember that increased sophistication in operating system capabilities requires a "trade off" in the form of increased overhead.

"Perfect" processes. It is certainly possible for any computer system to support a multiprogramming system, provided some very stringent conditions are enforced. The first condition is that all processes actively cooperate. A process must not attempt to destroy other processes nor interfere with them in any way. After some short amount of processing, the process must call an operating system routine that will determine the next process to be run. All requests for I/O should be forwarded to the operating system via a call. There must be no attempts to "monopolize" the CPU or any other system resource.

The second condition is that all processes coexisting in main storage be fully debugged. The need for this is obvious, as it is apparent that if this condition is not fulfilled, there is no way to guarantee that the first condition will be, despite the best of intentions. It is difficult at best to ensure that these conditions -- particularly the second -- exist.

"Imperfect" processes. Imperfect programs could, for example, enter a hard loop. In this situation, the "scheduler routine" would never be called; the imperfect process would have halted the other processes. An imperfect process, of course, can attempt to write into memory not allocated to it, as any programmer whose job cancelled due

to a "protection exception" can attest!

Systems programmers have long realized that there are occasionally presented to the system, processes that can only be described as "hostile." The primary intent of such processes seems to be to "break" the system -- to circumvent its protection and allocation mechanisms. These processes often severely test the capabilities of the best operating systems; not all computers have the hardware features necessary to support an operating system that can repel any of them.

Hardware enhancements required

In practice, multiprogramming is not often attempted on hardware lacking certain essential features; the stringent conditions imposed on processes admissible to such machines make the effort impractical except for occasional dedicated applications.

Shaw [1974] specifies some hardware features that are required for or that simplify multiprogrammed operation. We will briefly discuss each.

Priority Interrupt Facilities. This capability enables the hardware to interrupt the normal sequential processing of instructions by the CPU. An I/O interface, for example, can interrupt the CPU when it receives a character or complete message from a terminal. Disk storage circuitry can interrupt the CPU when a disk operation is complete. The proper operation of the multiprogrammed system requires this for two reasons: first, the next disk operation

in line must be started and, second, the processor scheduler must be "notified" that the process that was waiting for the completed operation is no longer waiting for I/O. Interrupts can occur from "external sources" -- the operator pressing the "interrupt key" on the front panel, for example. It is not essential, but is advantageous, for the interrupts to be "prioritized." This allows interrupts to be assigned to different priority classes; an interrupt that requests a service that can be delayed can be assigned to a lower priority class than one that deals with a distinctly time-sensitive one. An example of the former class would be an interrupt signifying that a line printer is ready to accept more output. An interrupt caused by the receipt of a character from a console keyboard might belong to the latter class; it is possible that the character might be lost if another arrives before the interrupt is "serviced."

Storage and Instruction Protection. We have already discussed the need for memory not allocated to a process to be protected from the actions of that process. There must be hardware features, controllable by the operating system, to protect and unprotect areas of main storage. "Instruction protection" refers to the need to limit the use of certain machine instructions to operating system use only. As an obvious example, consider the instructions used by the operating system to protect other users' memory, as described above. If any user can use these particular instructions, the protection mechanism is rendered ineffective. From

previous discussion it can be deduced that instructions that inhibit the interrupt system of the machine or directly affect I/O devices should also be placed in this class of instructions, commonly called "privileged instructions."

Dynamic Address Relocation. Address relocation in itself can certainly be termed a "classical" problem in operating systems design. In early computers, machine code was produced that would work only at one location in memory. This situation persisted for some time - even into the multiprogramming era. Early versions of IBM's Disk Operating System (DOS) required that several copies of the same program be available -- one copy assembled or compiled for each of the possible memory areas in which it might be required to run. This was clearly an unacceptable situation. Language processors and loaders were developed that permitted the "binding" of a process to a particular address to be delayed until the time when it was to be loaded into memory for execution. We can reasonably say that this much capability is a necessity for a practical implementation of a multiprogramming system. For reasons that will be discussed later, there are further advantages to be gained by delaying the binding time until the address is actually referenced by the CPU in the execution of the process. This capability is known as "dynamic address relocation" and is not strictly necessary but may be very convenient for implementation of a multiprogrammed system.

Timer. The existence of a timer capable of generating

interrupts to the CPU is convenient in the implementation of multiprogramming, and is essential for a time-slice system.

Base Registers. The presence of a base register capability in the hardware facilitates relocation of processes. A base register for a section of code or data is a register usually containing the main storage address of the beginning of that section of code or data. Storage addresses falling within that section are referred to by instructions in the process by their difference from the address that is in the base register. This has two important effects. First, it reduces the required size of the address field(s) in the instruction format(s); for example, if the size of a section is limited to 4096 bytes (as it is in the IBM 360 and 370), only twelve bits are required in the instruction format to provide for the difference or "displacement" in the address. Naturally, the instruction must also provide a means to indicate the identity of the particular base register to be used, but even then the space savings in the instruction format is significant; in the IBM 360 and 370, the base register and displacement together require 16 bits in the instruction, as compared to 24 bits if the address was to be specified in its entirety. Secondly, a program ready for execution can be loaded into main storage at any address and, provided the base registers are loaded properly, all memory-addressing instructions will operate properly. It should be noted that base registers by themselves are not sufficient for "dynamic address

relocation" as defined above. They do, however, greatly simplify program relocation at program load time.

Direct Access Auxiliary Storage. Disk storage or its functional equivalent is of great use in a multiprogramming system. User jobs can be held on disk storage until memory or other required resources are available. Jobs can be ranked by priority by the system rather than by the operator and processed in that order. Printed and punched output can be stored temporarily on disk until the appropriate peripheral is available for use - this is referred to as "output spooling." Although direct access auxiliary storage is not strictly necessary for multiprogramming, it is certainly extremely useful, and maximizes efficiency of the peripheral devices.

Summary

To summarize, multiprogramming is that condition that exists when several independent processes occupy main storage simultaneously and receive "multiplexed" service from the CPU to achieve an appearance of simultaneity in execution. An operating system is required to allocate the resources of the machine to processes competing for them and to enforce those allocations; the more sophisticated the allocation strategies and flexible the enforcement of allocations, the more overhead can be expected. Provided that processes meet certain stringent requirements, any computer can be multiprogrammed; in practice, however, several hardware features are required or advantageous for

implementation of multiprogramming.

2.2 Multiprocessing

The first computers were slow; vacuum tube technology significantly limited the speed at which CPUs and memory could operate. Transistor technology increased this speed significantly, and integrated circuitry, with its improved composition and decreased size, provided another boost. Advances in technology are not, however, the only ways a computer can be made to execute more instructions in a second.

"Increases in effective computer speeds can be achieved by improvements in either component technology or machine architecture. Given a fixed technology, parallel execution of hardware units can, in principle, dramatically improve system performance as compared with sequential operation. Several independent processors are often connected to common storage and control circuitry; these include central processors, I/O processors, such as data channels, and special purpose processors, such as arithmetic units." [Shaw 1974]

More than one CPU for user processes .

For the purposes of this thesis, we shall define multiprocessing as the use of two or more CPUs to operate on user processes simultaneously. This may or may not, depending on the degree of sophistication of the operating system, include the operation of two or more CPUs on the same user job.

More processing power. An expected consequence of multiprocessing is that the raw processing power of the system is multiplied by approximately c , where c is the

number of CPUs. The factor cannot be exactly c , as there will be occasions when both CPUs wish to fetch from the same "memory module"; this will result in a momentary delay for one of the CPUs due to "memory contention."

True simultaneity of execution. Another consequence is, of course, true simultaneity of process execution. Two processes, each being executed by its own processor, are executing literally simultaneously. This true simultaneity can actually cause difficulties, especially as regards "critical sections" of code in the operating system. Critical sections are areas of code usually dealing with resource allocation or enforcement that, due to their function, should only be "occupied" by one process at a time. Although a discussion of critical sections sufficient to impart understanding would be too lengthy to include here, a simple example may help to give an intuitive feeling for the problem. Suppose that both CPUs in a dual CPU system remove themselves from their respective processes simultaneously for some reason (just a coincidence, for example) and enter the CPU scheduling routine simultaneously. If one or the other is not prevented from proceeding, it is inevitable that both CPUs will assign themselves to the same user process simultaneously. The result of this will, in general, be disastrous to the user process -- for example, any additions of one area in memory to another will be done twice. Just this example alone should convince one of the need for exclusion from "critical sections" like the processor

scheduler!

Usually includes multiprogramming

As in the example above, multiprocessing can and usually does include multiprogramming. There is no logical difference as far as the user is concerned -- his program still produces the same result it would if it were run on a machine dedicated to his job alone. In a uniprocessor multiprogramming system, there might be four or five jobs, for example, being serviced by one processor. In a multiprocessor configuration, there might be eight or nine jobs being serviced by two processors. A process might be serviced by CPU A on one time slice and by CPU B on the next. For the purposes of the remainder of this paper, the term multiprocessing shall mean multiprocessing with multiprogramming.

As we might expect, multiprocessing does result in a more complex operating system with a resultant increase in overhead. Most of this overhead is due to implementation of the integrated control within the operating system; synchronization (as mentioned above) and scheduling in general are areas where the operating system must be significantly more complex [Baer 1976].

Hardware enhancements required

Multiprocessing requires some additional hardware features over and above those required for multiprogramming.

Control of processors. Since there are two or more

CPUs, the relationship between them must be defined and arranged in hardware. CPUs may be arranged in a master/slave relationship, where one serves as a processing peripheral to the other, or may be arranged as equals in what is referred to as a "symmetrical" multiprocessing system.

Memory access. There must be some hardware provision made to enable both CPUs to access a common memory. The memory system must be able to resolve contention conflicts.

Synchronization. Exclusion from critical sections can be done via software mechanisms. This, however, involves complex and confusing routines, as well as an increase in overhead [Shaw 1974]. A hardware solution is much more desirable. To enable exclusion from critical areas as defined above, an instruction must exist that tests a byte and sets it in one instruction. This is necessary even in uniprocessor systems, but the multiprocessor environment puts even tighter requirements on this instruction. In a uniprocessor system it was sufficient to ensure that this operation could be completed within one machine instruction, as interrupts (and, therefore, process switches) could occur only between instructions. In a multiprocessor environment, however, it is conceivable that two CPUs could be executing a "test-and-set" at the same time on the same byte in memory. The hardware must therefore ensure that the entire test-and-set is performed without any possibility of a memory access from the other processor(s) during it.

Summary

To summarize, multiprocessing differs from uniprocessor multiprocessing in one primary way -- more than one CPU is in use working on user jobs. The amount of raw processing power is therefore multiplied without having to include the peripheral and memory resources that would be required in simply obtaining a second independent computer system.

CHAPTER 3

THE CONCEPT OF VIRTUALIZABILITY

Throughout the previous chapters, it has been noted that the user's view of the system has not really been changed by multiprogramming. His jobs are still processed sequentially insofar as he can see. There are, of course, differences he has been able to notice or has had to accept. He has probably noticed that the begin and end times for his job indicate a greater total duration (caused by the fact that he is sharing the CPU with other jobs). This, however, has been offset, to the user's pleasure, by a significant decrease in the time he has had to wait for his job to be run (due to the greater throughput enjoyed by multiprogrammed systems). If the user is knowledgeable of the machine instruction set, he also realizes that there are several instructions he may not use. These are the "privileged instructions" mentioned briefly in the previous chapter.

Privileged instructions are, intuitively, those which can directly affect allocation of resources or the enforcement of the allocations, or directly affect the operating system itself. To allow a user access to these instructions would enable an imperfect or hostile process to catastrophically affect the system operation. Examples of privileged instructions include those which affect the interrupt system,

protect and unprotect memory, and do I/O. The operating system, as it has been defined earlier, is alone permitted use of these instructions.

The computer that is presented to the user, then, could be thought of as a different machine than the "bare" hardware he is actually using -- one possessing a subset of the instructions available on the bare machine. Some instructions are provided to alert the operating system to the fact that service is desired that only the operating system, with its access to the privileged instruction set, can provide -- I/O, for example. The machine that the user "sees" is sometimes referred to as a "virtual machine" [Shaw 1974] and sometimes as an "extended machine" [Goldberg, R. 1974]. This paper shall adopt the convention of referring to this machine as an "extended machine," as the term "virtual machine" will be given a more stringent definition later.

In most cases, the user should be quite satisfied with his extended machine; he has access to almost the entire instruction set and can "call" the operating system to perform those functions for him that he is prohibited from performing for himself. The lack of a few machine instructions would seem to be a small price to pay for the advantages to all users provided by multiprogramming.

What of the user, however, who is writing an operating system, or other software that requires use of the privileged instruction set? How is he to test his system? Under a

multiprogrammed operating system, only one body of privileged software can be run [Goldberg, R. 1974]. The user who finds himself in this situation must, it seems, revert to the original method of using the computer; he can only test his software when the entire machine can be dedicated to his job.

Another very practical problem relates to the fact that for some machines (IBM 360/370, for example), there are several operating systems available. Each system offers different degrees of sophistication, and each presents a different "extended machine" to the user. Often, programs that were written and translated to run "under" one system will not operate under another. Computer installations wishing to convert from one system to another to gain the advantages of a more sophisticated extended machine are usually faced with a monumental task in rewriting and/or retranslating their entire program library. The process sometimes takes months, during which one operating system must be "up" some of the time, and the other the rest of the time. As the time at which it is desirable to run a particular program does not always coincide with the time at which the operating system it can run under is up, practical difficulties of great magnitude can appear, especially in a heavily utilized facility. The root of this problem is also the inability to run more than one privileged software "nucleus" at one time.

Obviously, what would solve this problem is a special

operating system capable of supporting multiple extended machines that would each look like the real machine, complete with the full instruction set. The operating systems writer could then test his software during normal operating hours. The installation converting from one operating system to another could run both, each on a different extended machine provided by this special operating system.

This type of extended machine - appearing to the user to be identical to the real machine - will be termed a "virtual machine" (VM). The special operating system that creates these virtual machines is called a "virtual machine monitor" (VMM) [Popek 1974].

The concept of machine "virtualizability" can now be introduced. One can readily see that a VMM -- an operating system that allows users access to the full instruction set and still maintains control of the machine resources -- must be different in concept from a simple multiprogramming system. It should not be surprising to learn that this operating system requires a greater level of hardware support than is required for a multiprogrammed system.

3.1 Support of Virtual Machine Monitor (VMM)

Simply put, a machine is "virtualizable" if it is capable of supporting a VMM [Popek 1974]. This might appear to be too simple a definition, as it would seem to admit a "loophole" -- a software simulator.

Simulation as a method

It is possible for any machine to run a "simulator." A simulator is software that can run as a user program that simulates a complete computer system. In fact, such a method is often used in software development for a machine that has not yet been physically constructed. A simulator for the new machine is written to run on a current machine; each machine instruction for the new machine is interpreted and "executed" on the current machine by routines that simulate the effect of the new machine's instructions. The effect of privileged instructions is also simulated. The simulator may be written such that it can simulate several machines at once, or, since it runs as a user program, several copies of the simulator can run on the current machine under an ordinary multiprogramming system. Either way, the current system is able to support several "virtual machines" identical to the bare hardware comprising the new machine, and isolate them from each other.

The factor that will prevent us from calling the simulator a VMM is speed -- or lack of it. When a new machine is being simulated by a different machine, it is not uncommon to find the virtual machines thus provided slowed down by as much as 1000 to 1 [Goldberg, R. 1974]. The same situation still applies if we have a current machine simulate multiple copies of itself. Due to the fact that the instruction set of the "host" extended machine is identical (except for the privileged instructions) to the instruction

set of the machine being simulated, routines to simulate operation can be much more efficient. It may be possible to construct a simulator that exhibits only a 20 to 1 slowdown [Goldberg, R. 1974]. This may be a significant improvement over a 1000 to 1 slowdown, but it is still excessive. Purely software-based simulators, then, are inappropriate as VMMs, and properties of a virtual machine can be specified that close the "loophole" of software simulators.

Three properties for desirable operation

Gerald J. Popek and Robert P. Goldberg, in a paper titled Formal Requirements for Virtualizable Third Generation Architectures [Popek 1974], define three properties describing the operation of any arbitrary program if run on a virtual machine: the efficiency property, the resource control property, and the equivalence property.

Efficiency. In order for the efficiency property to hold, all "innocuous" instructions must be executed directly on the real machine's hardware, with no intervention or simulation on the part of the VMM. Innocuous instructions are intuitively, at this point, all machine instructions that are not "privileged." This requirement effectively eliminates software-based simulators from consideration as VMMs.

Resource control. The resource control property mandates that the program running on the virtual machine cannot, by itself, affect system resources; it must not be able to affect the amount of memory allocated to it, I/O

resources assigned to it, or CPU time allocated to it.

Equivalence. The equivalence property requires that the arbitrary program will perform indistinguishably from how it would were it to be run on the bare hardware, with two exceptions. The first exception concerns timing. As the VMM must intervene for privileged instructions, program sections which are time sensitive may not operate identically. In addition, it is assumed that there may be other virtual machines being supported on the same hardware in a multiprogramming arrangement; this will also affect the operation of time sensitive sections of code. The second exception concerns resource allocation. If the arbitrary program was running on the bare machine, it would have the entire memory, for example, available to it. When running under a VMM, it is obvious that this cannot be the case (unless virtual memory is also provided). Even if the program and the VMM were the only processes on the machine, the program would not have access to the entire memory; the VMM itself occupies a certain amount of storage. This second exception allows one to view the virtual machine as

"a 'smaller' version of the actual hardware: logically the same, but with a lesser quantity of certain resources. Then the equivalence to be guaranteed is that between running on an actual smaller hardware machine and the environment we have created." [Popek 1974]

Definitions

It is now possible to present better definitions for

"virtual machine monitor" and "virtual machine."

"We say that a virtual machine monitor (VMM) is any control program that satisfies the three properties of efficiency, resource control, and equivalence. Then functionally, the environment which any program sees when running with a virtual machine monitor present is called a virtual machine."
[Popek 1974]

Overhead considerations

Just as multiprogramming exacts a price, in the form of "overhead," so too does a VMM. The VMM is, of course, software. CPU time is required to run it, and main memory to hold it and its data. A collection of jobs, or "jobstream," requires additional time to run under a VMM as compared to the bare machine.

Robert P. Goldberg, in his paper titled Survey of Virtual Machine Research, describes some principal sources of overhead concerned with VMMs:

"Maintaining the status of the virtual processor. The complete integrity of all visible registers, status bits, and reserved memory (interrupt control) locations must be preserved.

Support of privileged instructions. Third-generation virtual machine systems have expended processor overhead to trap and simulate privileged instructions.

Support of Paging Within Virtual Machines. Software techniques are currently used to transform a paged address in a VM into an address in the VM and finally into a real memory address.

Console Functions. The operator's panel and lights are simulated in software. This overhead is not invoked as frequently as the others cited above.

Additional sources of overhead include the reflection of exceptions and I/O interrupts to the virtual machines, support of virtual timers and clocks, and the translation of I/O channel programs before the VMM initiates I/O." [Goldberg, R. 1974]

A few areas in the above quotation may require some

clarification, as they involve concepts not already discussed in this thesis. In regard to "paging," the term refers to a capability, implemented in software, hardware, or a combination of both, that effectively allows several virtual machines to address different physical memory locations via the same address. An example may help. There are often reserved areas of the physical memory that are used in a very specific way by the hardware or by privileged instructions. In the IBM 370, for example, memory locations 0 through 511 (decimal) are used to maintain machine status, including program status words, timers, and many other fields which are implicitly referred to by certain machine instructions. If a program running on a virtual machine is to have full use of the instruction set, it follows that it must have access to, and control of, that area of memory. Yet it is also required that the actions of a program running on a virtual machine only have an effect on that virtual machine; therefore, access to that real area of memory cannot be permitted. The solution is to "map" an address referenced within a VM to an address in real, physical storage. It is therefore possible for a system to provide an address range starting at zero to each virtual machine it supports. The responsibility for this mapping, in the IBM 370, falls to both the hardware and the VMM. A hardware feature referred to as "Dynamic Address Translation (DAT)" maps memory addresses specified by processes to physical memory addresses in main storage by reference to

tables in main storage or very fast memory buffers. The responsibility of the VMM in regard to DAT is to keep these translation tables updated with the proper correspondence between VM addresses and physical addresses.

The term "page" itself refers to a subdivision of main memory for which this address translation is indivisible; that is, an entire page range of addresses in a VM is translated to a corresponding page range of addresses in physical storage. The size of a page in the IBM 370 can be selected to be either 2,048 bytes (2K) or 4,096 bytes (4K) [IBM 1973].

"Translation of I/O channel programs" was also mentioned as a source of overhead. I/O channels are limited processors attached to main storage and I/O devices; through channel commands, they are directed by the CPU to perform I/O operations. The VMM must translate channel commands from the virtual machines before passing them on to the channel for primarily two reasons. First, when the VM attempts to use a system I/O device -- the operator's console, for example -- the VMM must redirect that output or input to the device that has been designated the operator's console for that virtual machine -- a user's terminal, for example. Secondly, some machines, like the IBM 370, do not extend address translation to the I/O channels. This implies that I/O channels must use physical addresses to refer to memory. The VMM must therefore translate addresses in the commands to the channels into physical addresses. "For

virtual machines supported with paged memory mapping, channel program translation can be a significant source of overhead" [Goldberg, R. 1974].

Implications for multiprogramming

Virtual machine monitors and the virtual machines they create go rather beyond the modest requirements of a simple multiprogramming system. It is clear, however, that if it can be shown that a machine architecture can support a virtual machine monitor, it can support a multiprogrammed system. At the least, each job could be "run" on a separate virtual machine. Practically that would not be necessary; this point, however, emphasizes that machine virtualizability is a sufficient condition for support of a multiprogrammed system.

3.2 Hardware requirements

Popek and Goldberg's paper goes on to define precisely the requirements a machine must fulfill in order to be capable of supporting a VMM.

Definition: third generation machine

Popek and Goldberg's theorems regarding virtualizability apply specifically to third generation machines, so it would be wise to review their definition of this class of computer systems:

"The processor is a conventional one with two modes of operation, supervisor and user. In supervisor mode, the complete instruction repertoire is available to the processor. In user mode, it is not. Memory addressing is done relative to the

contents of a relocation register. The instruction set consists of the usual complement of instructions for doing arithmetic, testing, branching, moving data in memory, and the like. . . . After superficial complexities in such systems are removed, what remains is generally a primitive protection system built around a supervisor/user mode concept, and a simple memory allocation system built around a relocation-bounds system." [Popek 1974]

A brief explanation of the concept of a "relocation register" is required. In a machine with a single relocation register, all memory accesses are done relative to the contents of that register. In other words, the address specified by the CPU is added to the contents of the relocation register in order to obtain the physical address. Different virtual machines could each refer to address zero, but, provided that the relocation register was loaded with the beginning address of the memory block allocated to each virtual machine before it was given control of the CPU (via a process switch), they would each be referencing different areas of physical storage.

Instruction classification

Popek and Goldberg define three classes of instructions on the basis of their behavior. The relationship between these classes determines whether or not the machine is virtualizable.

Privileged. An instruction is privileged if and only if it "traps" when executed in the user mode and does not "trap" when executed in the supervisor mode. Briefly, a trap is an interrupt generated by the CPU when it attempts to access out-of-range memory addresses or, as in this case,

attempts to execute an instruction reserved for operating system use while it is in the user mode [Popek 1974].

Sensitive. The next class of instructions are called "sensitive" instructions. There are two types of sensitivity: "control sensitivity" and "behavior sensitivity." An instruction is control sensitive if it can affect the amount of resources allocated or can change the processor mode. Control sensitive instructions are those that can affect the control that a VMM must have over the resources of the system. An example of a control sensitive instruction would be "Mask Timer Interrupt." Execution of this instruction would enable a process to continue indefinitely, monopolizing a resource -- CPU time -- the VMM must be able to control. Behavior sensitive instructions are those whose effect depends on the value in the relocation-bounds register or the mode (supervisor or user). In short, they do not always do the same thing, depending on their location in physical storage or the mode [Popek 1974]. An example of a behavior sensitive instruction would be "Load Physical Address." The value loaded would depend on the value in the relocation-bounds register (i.e., where in physical memory the program is located).

Innocuous. The last class of instructions are the innocuous instructions. Very simply, if an instruction is not sensitive (control or behavior sensitive), it is innocuous. In other words, sensitive instructions and innocuous instructions are disjoint sets the union of which

is the entire instruction set of the machine. Privileged instructions, on the other hand, may include some that are sensitive and some that are innocuous [Popek 1974].

Virtualizability theorem

Theorem 1 in Popek and Goldberg's paper provides a criterion, based upon the above instruction classification, for determining whether a specific machine is virtualizable:

"For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions." [Popek 1974]

It should be pointed out that the converse -- if any sensitive instruction is not privileged, a VMM can not be constructed -- does not necessarily hold. Popek and Goldberg indicate that it may be possible to work around certain types of deficiencies in an ad hoc manner to implement a VMM on a machine not quite satisfying the requirements of the theorem.

Recursive virtualizability theorem

Since an important purpose of the VMM is to permit more than one body of privileged code to run on a single machine simultaneously, an interesting question can be asked: Is it possible to run a virtual machine monitor as a user program under a virtual machine monitor; put another way, since a virtual machine generated by a VMM is supposed to be an efficient duplicate of the real machine, can another (or a copy of the same) VMM run on that virtual machine? A

computer whose hardware permits this operation is called recursively virtualizable. Theorem 2 in Popek and Goldberg specifies requirements for recursive virtualizability:

"A conventional third generation computer is recursively virtualizable if it is: (a) virtualizable, and (b) a VMM without any timing dependencies can be constructed for it." [Popek 1974]

The restriction on timing dependencies arises from the equivalence property described earlier. The virtual machine is equivalent to the real machine with two exceptions: timing and resource availability. If a VMM includes time-sensitive code, it almost certainly will not perform on a VM as if it were running on the bare machine; therefore, the timing restriction must be included in Theorem 2 above. The consequence of the resource availability exception, incidentally, implies that indefinite recursive virtualizability will result in the virtual machines at each successive level being smaller and smaller, until there is insufficient main storage available to continue the recursion.

Recursive virtualization, as might be intuitively realized, can grossly increase the overhead on the machine, and is of little practical value.

3.3 Summary

The concept of virtualizability is a useful one, not only for its primary benefits, but also because it is a sufficient condition for support of a multiprogrammed operating system. A theorem has been discussed that enables one to determine whether a machine is virtualizable on the

basis of a defined classification of the instruction set. This will be of significant use in the following chapters.

CHAPTER 4

4.1 THE INTEL MICROPROCESSOR: INTEL 8086

The Intel 8086 is a typical microprocessor to which supply. It is a 16-bit microprocessor designed, designed, and has been used in a wide range of applications brought the price down to under \$100. It is a single microprocessor.

Most other microprocessors, however, designed after the 8086, have at least 16 bits of data bus. It would be reasonable to suppose that if a virtual machine architecture can be constructed using the 8086, it should be possible to do so with most other microprocessors as well. The Intel 8086, therefore, is the microprocessor that will be examined as to its virtualizability, and shall be the device upon which the architecture in Chapter 5 will be based.

The purpose of this chapter is to introduce the 8086, including basic architecture and programming. It is the detail required to understand the architecture in Chapter 5. It discusses a number of features often used in conjunction with the 8086. It discusses architecture problems with the 8086 that affect its capacity to support a multiprogramming operating system, including problems in regard to virtualizability, and (d) look at current multi-processor

CHAPTER 4

A TYPICAL MICROPROCESSOR: INTEL 8080

The Intel 8080 is a typical microprocessor in good supply. It is among the earliest microprocessors to be designed, and has been sold in sufficient quantities to have brought the price down to below twelve dollars for a single microprocessor.

Most other microprocessors, having been designed after the 8080, have at least its capabilities, so it would be reasonable to suppose that if a virtualizable machine architecture can be constructed using the 8080, it should be possible to do so with most other microprocessors as well. The Intel 8080, therefore, is the microprocessor that will be examined as to its virtualizability, and shall be the device upon which the architecture in Chapter 5 will be based.

The purposes of this chapter are to (a) introduce the 8080, including basic architecture and instruction set, in the detail required to understand the architecture in Chapter 5, (b) discuss a standard "bus" structure often used in conjunction with the 8080, (c) examine architectural problems with the 8080 that affect its capacity to support a multiprogramming operating system, including problems in regard to virtualizability, and (d) look at current multi-user

systems based on the 8080 and a similar microprocessor, noting their limitations.

4.1 Architecture and instruction set

The 8080 itself is a single-chip microprocessor, fabricated using n-channel metal-oxide semiconductor (NMOS) technology. A basic clock period is .5 microseconds (500 nanoseconds) in length. From three to five clock periods constitute a "machine cycle", and one or more machine cycles are required to execute an instruction. Provided the components used in main storage can respond in a clock period or less, instruction length will vary from four to eighteen clock periods (2.0 to 9.0 microseconds); slow memory will increase the number of clock periods required for a machine cycle and, therefore, the instruction execution time [Osborne 1975]. The speed of the 8080 is comparable to most other current microprocessors.

Registers

There are seven 8-bit registers within the 8080 that are usable by a programmer. One is the accumulator (A); it has many capabilities not common to all registers, such as arithmetic/logical operations and I/O. The other six registers -- B, C, D, E, H and L -- can be used as three 16-bit "register pairs" for address computation or access, or as individual 8-bit general purpose registers. There is also an 8-bit status register, only five bits of which are used. These five bits contain the current value of the

status flags, which are Zero, Sign, Parity, Carry and Auxiliary Carry. The status register together with the accumulator make up a 16-bit Program Status Word (PSW).

There are two 16-bit registers in the 8080 with which the programmer must be concerned. The Program Counter contains the address of the next instruction to be executed. The Stack Pointer implements a single pushdown LIFO stack in main memory. The Stack Pointer contains the address of the "top" entry on the stack. "Top" is actually a misnomer, as the Stack Pointer is normally initially set at the top of its available memory range; each "push" operation actually decreases the value in the stack pointer by two (stack entries are two bytes in length).

Memory access

Sixteen address bits are provided by the 8080 for use in accessing memory. The logical address space is therefore limited to 65,536 bytes. During a memory write operation, the 8080 outputs the address on its sixteen address lines, outputs status to indicate that the operation in progress is a memory write and, during the second half of the current clock period, outputs the data to be stored in the addressed byte on its eight data lines. A memory read is performed in precisely the same way, except that the status indicates a memory read and the data lines are used to input the data from the memory.

Input/Output operations

Input/Output operations resemble memory operations in many ways. The data lines are also used to output data from the CPU or input it from the device interface. The same lines used to specify a memory address are also used to specify an "I/O Port" number; these I/O Port numbers are eight bits wide (a range of 0-255 decimal) and are duplicated on the high-order and low-order 8-bits of the address lines. Naturally, the status during these operations differs from memory operations; there are status lines that indicate an input or output operation.

Interrupt system

The 8080 may be interrupted between instructions, and there is a line into the CPU that indicates whether an interrupt is pending (INT). Provided interrupts have not been disabled within the CPU (by execution of a disable interrupt (DI) instruction), this line is checked by the CPU between each instruction and, if the line is "high" (binary value 1), the CPU will cease obtaining instructions from memory and output a status indication that the interrupt has been acknowledged. It is then the responsibility of the interrupting I/O interface to "jam" an instruction operation code onto the data lines, where the CPU expects to find it. This instruction will in most cases be a "restart" instruction -- a one-byte "call" to an implicit address. There are eight restart instructions in the 8080's repertoire, and they cause a call (push the current value of the program

counter onto the stack) to routines at (hexadecimal) memory addresses 0, 8, 10, 18, 20, 28, 30 and 38. It is the responsibility of these routines to take such actions as saving PSW and other register contents and servicing the interrupt. The 8080, incidently, continually outputs a status signal (INTE) to indicate whether interrupts are enabled.

Status and control signals

In the discussion thus far we have referred to some "status indications" the 8080 receives or outputs. There are other status signals which should be briefly described, as they will be of use in understanding the architecture to be discussed in Chapter 5.

Input (to the CPU) status signals that are of interest include Reset (RESET) and Ready (READY). RESET is a signal that, if held high for at least three clock periods, will zero all registers except the status flags, thus causing (as soon as the Reset signal is removed) program execution to start with location zero in memory. READY is a status signal to inform the CPU that signals on the data line are stable and can be used. This is the means by which slow memory or input interfaces can prevent the CPU from moving ahead until they are ready.

Output (from the CPU) status signals that should be understood are Interrupt Acknowledge (INTA), I/O Input (INP), I/O Output (OUT), Memory Read (MEMR), Memory Write (WO), Interrupt Enabled (INTE), Halt Acknowledge (HLTA) and First

Instruction Byte Fetch (M1). INTA, INP, OUT, MEMR, WO and INTE have already been discussed in sufficient detail. Halt Acknowledge (HLTA) indicates that the CPU has halted in response to a HALT instruction in software. An interrupt will be required to restart execution. If a CPU is halted with interrupts disabled, a RESET is the only way to restart it. Signal M1 indicates that the byte being fetched from memory is the first byte of a new instruction.

Instruction set

The instruction set of the 8080 is listed in Appendix A. It is a typical instruction set in relation to other microprocessors, containing the usual mix of instructions to move data between registers and/or memory, perform arithmetic and logical operations and do input/output.

Persons primarily familiar with a large machine such as the IBM 370 may find it easier to comprehend the relative powers of the instruction sets with an example. Many instructions in the repertoire of the IBM 370 allow a programmer to accomplish a great deal with one instruction; for example, a common instruction -- MVC -- can move up to 256 bytes from one location in memory to another. The 8080 instruction set allows only one byte at a time to be moved, and even then it is a two instruction operation for each byte! On the other hand, the instructions of the IBM 370 are two, four or six bytes in length; 8080 instructions are only one, two or three bytes long.

4.2 The S-100 bus

A microprocessor is of little use alone. In fact, the 8080 can do literally nothing by itself. A certain minimal amount of circuitry is essential just to make the address, data and status lines available to memory and I/O interfaces. Ideally, a standard "bus" should be established into which all memory and I/O boards could "plug." The bus would have to have at least enough lines to carry the address, data and status needed, and leave some free lines for later expansion or special purposes. As mentioned in Chapter 1, the MITS company was the first to introduce a complete microcomputer based on the 8080 in kit form -- the Altair. The bus they used on their machine had one hundred lines -- well more than needed. As MITS memory and I/O boards were rather highly priced, a number of other manufacturers sprang up to offer "Altair bus compatible" memory and interface boards.

In a short time, another firm, IMS Associates, marketed a new microcomputer system based on the Intel 8080. This was a very important point in industry development, for the IMSAI 8080, as the IMS machine was called, was also based on the "Altair bus." The idea of a standard bus structure was reinforced, and today there are more than a dozen microcomputers based on that bus, and many times that number of manufacturers supplying compatible boards. New microcomputer manufacturers were naturally reluctant to use the term "Altair bus"; the standard 100-pin bus originated

by MITS is now known as the S-100 bus. Designations for the 100 lines on the bus are given in Appendix B.

4.3 Multiprogramming and the 8080

In determining the feasibility of implementing a multiprogrammed operating system on the 8080, a logical starting question might be "Is the 8080 virtualizable?" If it is, then it can certainly support a multiprogramming system, and we can proceed immediately to design the operating system software. This, however, is not the case.

No privileged instruction set

The first problem one can see regarding the 8080's instruction set is that there are no privileged instructions. There is only one mode of operation. Any program running on the CPU has access to the entire instruction set. Unfortunately, an examination of the instruction set reveals several instructions that appear to be sensitive.

DI (Disable Interrupts). The Disable Interrupts instruction (DI) masks out all interrupts from the CPU. As this instruction could be used by a process to eliminate timer interrupts being used to time-slice, for example, it could be used to "grab" control of the machine, effectively allocating all CPU time -- a resource -- to that one process. The DI instruction is obviously control sensitive.

OUT (Output). The Output instructions (OUT) is also sensitive because of two normal uses. The obvious use of an OUT instruction is to output data to an output device.

There is nothing to prevent a process from printing on any terminal it wishes -- any I/O interface could be addressed. As I/O devices are system resources, OUT is control-sensitive. The OUT instruction has another function in many microcomputer systems -- it is used to write-protect blocks of memory. An I/O port is implemented on each memory board, and an OUT instruction addressing that port is used to set a block of memory to write-protect status (read but no write) or to unprotected status (reads and writes permitted). An imperfect or hostile process could use this function of the OUT instruction to unprotect other processes' assigned memory and destroy its contents. This is another function that qualifies the OUT instruction as control-sensitive.

IN (Input). While the reason is not as obvious as for the OUT instruction, the IN instruction is also control sensitive. The status indication made available by an input interface to indicate that a character has been received and is available to be read from the data port lasts only until an IN instruction reads the character from that data port. If a process performs an IN instruction from a data port assigned to a second process, that second process may miss a character.

Other possible problems

Obviously, on the basis on non-privileged sensitive instructions, the 8080 does not satisfy the requirements of Popek and Goldberg's theorem on virtualizability. The same problem indicates that it will be practically impossible to

implement a multiprogramming operating system on an 8080. Although it may at this point appear to be merely an academic exercise, other characteristics of the 8080 that affect its capability for multiprogramming will now be examined.

Processor speed. The speed of the 8080, while not near that of large computers, is sufficient for many types of applications that could be multiprogrammed -- especially ones that are I/O-bound or have a large percentage of user "think" time, as in educational time-sharing systems. So speed does not appear to be a serious problem for multiprogramming, although implementation of a multiprocessing capability would be valuable in adapting the machine for processes that use significant CPU time or to reduce the variance in response time due to process CPU time loading.

Address space. As discussed above, the address range for memory is only 65,536 bytes (64K). This is because memory is addressed via 16 lines from the CPU and on the bus. This quantity is sufficient for most single processes, but severely restricts the number of processes that could be in main storage simultaneously. For example, if only 16K were allocated for each process, only 4 processes could be in memory simultaneously. It can be said that the address space is a problem that would have to be dealt with in implementing a multiprogrammed operating system.

Relocation aids. Relocation aids -- base registers, for example -- are considered very advantageous for

multiprogramming, facilitating as they do the loading of a process at any address in main storage. The 8080 has no facility even remotely resembling this concept. Memory is accessed via either an explicit address in a 16-bit register pair or via a 16 bit address in the second and third bytes of a memory referencing instruction. Any relocation must therefore be done by a relocating loader. Much of the packaged software available for the 8080 is available in object code only -- object code that will work at only one address. Either this software must be abandoned, or multiple copies would have to be compiled or assembled to run at several addresses, and the appropriate copy selected by the operating system depending on the memory range that was available, a not particularly satisfactory solution.

No "Test and Set" instruction. As discussed in Chapter 2, a "Test and Set" instruction is generally considered necessary for synchronization. An examination of the 8080's instruction set reveals no instruction similar to Test and Set.

4.4 Current multi-user systems

It would therefore appear that the Intel 8080 is badly suited for a multiprogrammed operating system. There are, however, some "time-sharing" systems based on the 8080 or a similar processor available. These systems were investigated to determine the method used to implement multi-user capabilities.

Cromemco system

The time-sharing system vended by Cromemco, Inc., a manufacturer of an S-100 bus system based on the Zilog Z-80 microprocessor (a CPU upwardly software compatible with the 8080), is constructed as a typical multiprogrammed system. An operating system is resident and controls CPU time allocated to each process; a timer-generated interrupt initiates the process switches. Mr. Brian G. Job, Sales Manager at Cromemco, provided the following information regarding their system.

The Cromemco system uses a memory bank system to alleviate the problem of the limited address range. As is the case in essentially all S-100 bus memory boards, a Cromemco memory board has a switch bank to set the address range to which the board will respond. Unlike other boards, however, a Cromemco board also has a set of eight switches that allow the board to be assigned to one or more of eight "memory banks." If only switch 6 is on, for example, that board is assigned to bank 6 only. If both switches 2 and 4 are on, that board is shared by banks 2 and 4 -- it "belongs" to both banks. Figure 1 illustrates the physical structure of the Cromemco system and also presents a logically equivalent intuitive structure. Each board shown in the physical structure has an address range (illustrated below the block) and a "bank membership" (illustrated above the block) set via switches before the system is powered up. It is inadvisable to change the setting of these switches

Figure 1. Cromemco Multi-user System

during operation. A given board will respond to a memory operation initiated by the CPU only if the specified address is within the set address range of the board and the board is "bank enabled." An output port is implemented on each board -- all boards have the same port number. In order to enable the boards belonging to bank 2, for example, the CPU must execute an OUT instruction to that port number with bit 2 in the accumulator on (1).

Each user has his own bank of memory, which can coexist with other banks in the same address range, since only one bank will be enabled at any one time. Each user can therefore have up to 64K of memory (if 64K of memory was assigned to that bank when the memory boards were set up). This would seem to be a good way of protecting the memory allocated to one process from disruption by another process.

The Z-80 has an instruction set somewhat expanded from that of the 8080, but there are still no privileged instructions. The Z-80 is therefore no better than the 8080 in regard to its capability to support a true multiprogramming system.

How, then, is the Cromemco time-sharing system supported by the hardware? The answer is that it is not. The problems have been partially ignored and partially retreated from. Cromemco supplies with the time-sharing system a special BASIC language interpreter. This high-level language specifically avoids implementing the most troublesome BASIC instructions; the abilities to do direct

output and to store into a certain memory address, normally capabilities included on the better microcomputer BASICs, have been omitted. They are purposeful omissions -- the BASIC instructions to read a specific memory byte and to do direct input have been included. In addition, the ability to call machine language subroutines has been eliminated; a user who could do this might, in error, execute a Disable Interrupts instruction or, perhaps, turn other banks of memory on and alter their contents.

Cromemco does not limit the user to use of this time-sharing BASIC. As the operating system has been designed as a multiprogramming system, the "owner" of the system can allow anything, including machine language programs, to be run. In this case, Cromemco warns, there is no way to guarantee that a malfunctioning process will not disrupt other users and the operating system itself. In short, this is a return to many of the stringent conditions mentioned in Chapter 1 that guaranteed that a multiprogramming operating system could run on any machine. Unfortunately, they were impractical when they were discussed, and it must be concluded that the Cromemco time-sharing system is largely impractical for the same reason. Cromemco made a step forward with the memory bank system; it did not, however, solve the other serious problems.

Altair system

MITS, manufacturer of the Altair line of microcomputers, has made available a time-sharing system

that runs on the 8080-based Altair 8800. Mr. Peter Connor, owner of the Computer Shack store in Albuquerque, New Mexico, a MITS dealer, provided information concerning Altair Timesharing BASIC.

The Altair software does not include even an attempt to implement a multiprogramming capability. Instead, a special BASIC interpreter is provided that can provide multiplexed service to several user programs existing in memory simultaneously. It should be noted that no banking is used; all users' programs must fit in what is left of the 64K memory after the BASIC interpreter is loaded. This is a severe limit. Users are prevented from affecting other users by the elimination of BASIC statement types that could cause trouble.

ShackShare system

Mr. Connor also provided information on his own timesharing system, called "ShackShare." He indicated that the system was developed primarily for use by his own programmers. The need for the system was dictated by a demand for multi-user business applications.

ShackShare uses a memory bank system much like the Cromemco system, except that each bank is composed of 60K of memory. The top 4K of memory (F000-FFFF) is reserved for the "operating system."

A BASIC interpreter is also provided, but, unlike the Cromemco time-sharing BASIC, it allows the use of instructions that could cause interference with other

processes. Mr. Connor reports that this is known and accepted; all applications are fully debugged before they are sold to customers. The author's own experience in programming microcomputers and large machines raises doubts in this area.

Summary: current systems

It is clear from the above information that a true multiprogrammed system has yet to be developed for current microcomputers. Current attempts either ignore the problems or limit the user to use of a single, limited high-level language.

3.1 Supertwo - computer address

There is a way, however, in which this distinction can be made. This method will be discussed now.

In the design of operating systems, it has been suggested that maintaining a multiprogrammed system is more difficult than maintaining a single program system. This is because the system must be able to handle a large number of programs at the same time. This is a significant challenge in the design of operating systems. It has also been suggested that the system must be able to handle a large number of programs at the same time. This is a significant challenge in the design of operating systems.

CHAPTER 5

PROPOSED MACHINE ARCHITECTURE

The material presented in Chapter 4 would seem to make an 8080-based true multiprogramming system an impracticality. The prime difficulty is apparently a lack of any privileged instruction set whatsoever; "instruction protection," as described earlier, is non-existent. As there is no "supervisor mode" of operation (or perhaps it would be better to say no "user mode"), it is obvious that there cannot be any instructions whose use is limited to the operating system. In short, there is no hardware distinction between the environment in which the operating system runs and the environment in which the user processes run.

5.1 Supervisory computer concept

There is a way, however, in which this distinction can be created. This method will be discussed now.

In the design of operating systems, it has been suggested that visualizing a multiprogrammed operating system and the user programs as a collection of processes being run on logically separate computers can be useful in clarifying the design of the operating system software [Gaines 1972]. It has also been often expressed that the software/hardware boundary, especially in operating systems, has been shifting

in the direction of implementing more of the central operating system as hardware functions [Tanenbaum 1976] [Shaw 1974], and that, in fact, "hardware and software are logically equivalent" [Tanenbaum 1976].

It has already been determined that the 8080 can provide not even the minimal support required for a multiprogrammed system; the "software/hardware boundary" has been moved as far as possible toward software, and yet it is not enough. If one 8080 will not support a multiprogrammed system, perhaps two should be tried.

Dedicated CPU for operating system

The concept of a "supervisory" computer has been discussed in relation to multiprocessing systems for some time [Shaw 1974] [Gagliardi 1975] [Baer 1976]. An architecture which employs this concept where one processor has control over the others is called an "asymmetric multiprocessor"; the type of processor control is referred to as "fixed." This distinguishes it from the architecture in which all processors are equal, have access to the operating system code and schedule themselves; this architecture is called a "symmetric multiprocessor" and the type of control referred to as "floating" [Baer 1976].

A recent book on microprocessors describes a cellular computer in which each node consists of two microprocessors -- one handles a single user's program, and the other oversees communications with other nodes and operating system routines [Rao 1978].

The effect on virtualizability, however, has apparently not been addressed. By placing two microprocessors in a master-slave relationship, two hardware "modes" have effectively been created. The mode is either "master" or "slave," depending on which processor the software is running. This architecture moves a distinction normally made in the instruction set of a single CPU into the structure of the machine.

This master/slave arrangement of processors is a central concept of the architecture being proposed. The master processor executes only operating system code. As the master processor is in control of the real resources of the machine, this provision is essential.

One or more CPUs for user processes

The slave processor executes, generally, user programs. It is probable that many areas of the operating system could be executed by the slave processor, but, since there is a processor dedicated to operating system functions, it is likely that very little of this will be necessary. This is a boundary that can be moved if necessary to maximize throughput of the system.

There is no reason why there cannot be more than one slave processor. Increasing this number would make the system fit the definition of a multiprocessing system introduced in Chapter 2, which required that more than one processor be used on user processes.

The master-slave concept implies that one processor

is strongly under the control of the other. Means must be provided in the architecture to implement this control. For example, the master must be able to interrupt the slave. Since we know that a user could mask off interrupts, the master must be able to interrupt the slave even when the slave's interrupts are masked! This effectively would make Disable Interrupts (DI) a privileged instruction, since only the master processor would have the ability to truly mask interrupts.

As it is desirable for the multiprocessing system to multiprogram, all slave CPUs should have access to a common main storage. Otherwise, each process would have to be moved into a different processor's "local" memory for each slice of its execution time, producing excessive overhead.

The master processor should have access to the common main storage as well, as it will occasionally have to examine register save areas, etc., as well as perform I/O from common memory.

Input/Output

The master processor, having control over real machine resources, must have control over I/O devices. All I/O interfaces will be on the master processor's bus. IN and OUT instructions in user processes will have to be trapped and performed in some way by the master processor. This effectively makes IN and OUT privileged instructions, as user processes must go through an operating system trap procedure for them to have any effect. Slave processors

would not normally have any I/O interfaces on their busses.

The master processor need not actually perform the I/O operations, although it could. Although not a part of the architecture being proposed, a microprocessor-based I/O channel could be installed either on the master processor's bus or as an independent slave processor executing only I/O routines under direction of the master processor.

5.2 Dynamic Memory Banking system

It appears that a concept is taking form that might well be able to support multiprogramming or perhaps even a VMM.

Memory-related difficulties

But other problems remain, among them the limited main storage address space and the lack of program relocation aids. One method used in some large machine time-sharing systems to compensate for a limited amount of memory consists of keeping only a few of the active processes in memory at one time, "swapping" processes continuously between main storage and an external medium like magnetic disks. Needless to say, the overhead in this arrangement is excessive [McKinney 1969].

Moreover, what memory there is is subject to "fragmentation." This is a phenomenon that is common to multiprogrammed systems using static memory allocation. Figure 2 illustrates a simple example of storage fragmentation. At A, the memory map appears as it would

Figure 2. Example of Storage Fragmentation

after system initialization. The operating system is occupying the bottom 4K of memory, leaving 60K unused and available for user programs. At B, the first three programs have been loaded into memory and are active (in execution). The first program to be loaded required 16K and was loaded from the 4K point in memory (just above the operating system) to the 20K point. The second program, which required 12K, has been loaded from 20K to 32K. Program 3, 24K in length, was loaded from 32K to 56K. Program 4, which requires 16K, could not be loaded because there was only 8K of storage left unused. Some time later, at C, Program 2 terminates, freeing its 12K of storage. Despite the fact that there is now a total of 20K of storage unused, Program 4 still may not be loaded! User programs require blocks of contiguous storage, and the 20K of unused storage is fragmented into two non-contiguous blocks of 12K and 8K respectively. Over a period of time, storage can become fragmented into a "checkerboard of unused (and often unusable) 'holes'" [Shaw 1974]. In a static allocation system, there are really only two ways out of this problem. One is to cease accepting jobs and run until all jobs in memory have terminated; the memory is then as it was after initialization and the process of fragmentation can begin again. The other method consists of moving active programs around in memory to "compact" the unused memory into one block. Compaction generates significant overhead; in addition, instruction formats, addressing modes and register usage must allow

moving of code after execution has commenced. Intel 8080 machine code is not amenable to this activity, partially because of the lack of relocation aids.

Memory system development

What is required is a memory system architecture which relieves the limitation of 64K for the entire system, makes relocation unnecessary, and allows the assignment of whatever memory is available to any process that needs memory at any address. A starting point for this effort is the memory bank concept in the Cromemco time-sharing system discussed at the end of Chapter 4.

Review of Cromemco system. In the Cromemco memory bank concept, all memory boards (4K, 8K or 16K each) include two rows of hand set switches. One row defines the address range of the board. The other row consists of eight switches which assign the board to any combination of eight banks of memory. (See Figure 1 in Section 4.4.) Each bank can contain up to 64K of memory, so the entire system can contain up to 512K, a respectable amount for a multi-user system. An output port is implemented on the board -- the port number is the same for every board in the system. To make a bank active (enable it to respond when an address in its assigned range appears on the address bus), it is only necessary to perform an OUT instruction to that port, specifying (on the data lines) the bank number. Boards not assigned to that bank will be disabled and boards assigned to that bank will be enabled.

Bank assignment. Several alterations will be made to the Cromemco concept. For reasons that will become apparent later, there will be only four banks. The bank "membership" will be set via an OUT instruction to the port on the board containing the memory block, rather than by means of switches. The port number will not be the same for all boards, but will be different for each. A new row of eight switches will be used to set the "block number" -- this will be identical to the port number of the I/O port on the board.

To summarize the operation of the banking, an OUT instruction to a particular board will be able to set the bank membership of that board only. Note that nothing has been mentioned about turning banks "on" and "off," as in the Cromemco system. Four bus lines will be indicators of which bank is being accessed for each memory operation. If, for example, a memory read is being performed from location 5BA6 (hexadecimal) and there is a board assigned to the address range 5000 to 5FFF whose bank membership is banks 0 and 2, the board will respond, putting the contents of that memory location on the data bus, if either bank line 0 or 2 is high. The motivation for this operation will become clear soon.

Addressing.

The mechanism for addressing a block of memory will also be altered. At present, the address range of a typical microcomputer memory board is set via a row of switches. If

it is desired to be able to address the board on 4K boundaries, four switches (to set the first hexadecimal digit of the address) are required. If the switches are set for a hexadecimal C (binary 1100) on a 4K board, for example, the board will respond to addresses in the range C000 through CFFF. This arrangement is more flexible than in early machines, where the address was designed right into the basic memory structure.

It is necessary, however, to make the addressing mechanism a good deal more flexible. There is no reason why the address range of a board has to be manually fixed in any way. If the operating system software can at any time specify the address range of a given memory board, it will facilitate memory allocation and use in ways that will be discussed later. This concept is not unprecedented; the idea of including a "page comparator" on a memory module has been advanced to serve the needs of advanced distributed computer architectures [Anderson 1975]. The application of the concept in conjunction with a memory banking system is, however, believed to be novel.

The combination of a dynamically reconfigurable memory bank system with dynamically readdressable memory blocks forms the basis of the "Dynamic Memory Banking" concept. As shall be shown later, this memory architecture has characteristics that will enable it to fulfill the desired requirements; it will relieve the 64K limitation on memory resources, make relocation unnecessary and allow the

assignment of memory to whatever process requires it at whatever address.

5.3 Architectural specifications

Little real detail has been provided in the preliminary discussion above. Moreover, the necessary interconnection of master and slave processors with the dynamic memory banking system has not been even initially addressed.

This section will define the functions of each of the major components, or "modules," of the system. Following this, examples sufficient to provide an intuitive understanding of system operation will be presented. It is believed that this approach will result in understanding superior to that which would result if operation of the system was discussed prior to adequately defining hardware functions.

The specifications which follow are not meant to completely define the hardware involved. Problems such as providing proper circuit timing are left to the implementor.

Figure 3 depicts the interrelationship of the four major areas of the system. The Master CPU and Bus area contains all I/O interfaces, the Master Processor itself, and a connection to the Memory & Processor Control Module. The bus structure in this area is S-100. Each Slave CPU Module contains the slave processor and a connection to the Memory & Processor Control Module. The Memory Module is a block of memory "sitting" on a bus controlled by the Memory

Figure 3. Module Interrelationships and Busses

& Processor Control Module. The Memory & Processor Control Module is circuitry designed to allow access to the memory system by all processors and to implement the control the master processor must have over the slave processor(s) and memory bank configuration and addressing. Where the detail is appropriate, the identity of signals being passed between modules is specified. Although omitted from the diagram for purposes of clarity, a common internal clock is used by all components of the system.

Master Processor and bus

The Master Processor and its bus greatly resembles a standard 8080-based S-100 bus microcomputer available from several manufacturers today. All I/O interfaces, including the operator's console, terminal interfaces, printers, readers, tape and disk controllers, and any other I/O interface desired, are "on" the Master Processor's bus. Also on the bus is the Memory & Processor Control Module (M&PCM), which is accessed by the Master Processor via both I/O and memory-accessing instructions. Precisely what the M&PCM "looks like" to the Master Processor will be covered in detail when the M&PCM is described below.

If a front panel is to be installed on the machine, the most logical place for it to reside would be the Master Processor's bus, perhaps augmented with some status signals directly from the M&PCM. Such a panel is not really necessary; most recent microcomputers omit them. During construction of a prototype, however, a front panel can be

useful for hardware debugging.

Slave Processor Modules

In Figure 3, only one Slave Processor Module (SPM) is shown. The architecture being described, however, provides for three SPMs; the two not shown are connected to the M&PCM in the same manner as the one shown. The following description of an SPM therefore applies to all SPMs in the system.

The SPM is a circuit board containing an 8080 and additional circuitry necessary to provide status signals, addresses and data to the M&PCM and to receive command signals and data from the M&PCM.

The sixteen address lines from the 8080 are buffered (isolated and strengthened) and "sent" to the M&PCM. The eight data lines are tri-state buffered and "sent" to the M&PCM. A "tri-state buffer" is intuitively a device that allows a signal to either control a line with a high (1) or low (0) value, or have no effect on the line. There are therefore three possible functions of the buffer - make the line high, make the line low, or let the line "float" -- hence the term "tri-state." This is required for the data lines since for certain instructions (OUT and memory write instructions) the 8080 must control what is on the data bus, and for others (IN and memory read instructions) the 8080 must allow an external source to control the contents of the data bus. Circuitry must also be provided to receive data bus contents from the M&PCM (on IN and memory read

instructions) and buffer it to the 8080. During implementation of this architecture, the designer may wish to exercise the option of providing separate data input and output busses, as it may well simplify circuitry in both the SPMs and the M&PCM.

Status signals from the 8080 are buffered and sent to the M&PCM, and command signals from the M&PCM are buffered for input to the 8080. The identity of these signals may provide a hint of their use in the operation of the architecture.

The status signals from the 8080 to the M&PCM include First Instruction Byte Fetch (M1), Interrupt Enabled (INTE), I/O Output (OUT), I/O Input (INP), Memory Read (MEMR), Memory Write (WO), and Halt Acknowledge (HLTA). M1 indicates that the current memory read operation taking place is for the purpose of fetching the first byte of a new instruction. This indication will provide the timing necessary for the M&PCM to "jam" in an instruction other than the one that would normally be fetched from the program code in memory. As shall be discussed under M&PCM design, this capability is essential to making the architecture virtualizable. The INTE signal will signal the M&PCM (and, through it, the Master Processor) that, depending on the value, interrupts are either enabled or masked off by the SPM. OUT and INP signals indicate to the M&PCM that an IN or OUT instruction is being executed. This will be used to initiate the "trap" operation for these instructions. MEMR and WO are used to

indicate that a memory operation is in progress; the M&PCM's function will then be to perform the desired operation and, if a memory read, "feed" the data back to the SPM. HLTA notifies the M&PCM (which passes the information on to the Master Processor) that the process running on the SPM has halted as a result of executing a HALT instruction. The Operating System running on the Master Processor may use this information to initiate a process switch sooner than it normally would have.

Command signals from the M&PCM to the SPM include RESET and READY. By use of the RESET line, the M&PCM can cause the SPM's 8080 to be reset; all registers except the status register will be zeroed. This will cause the SPM to start executing instructions at memory location zero as soon as the RESET is lifted. The READY signal enables the M&PCM to stall the SPM indefinitely right in the middle of an operation. This capability will prove useful in implementing the I/O trap, as well as being essential in delaying an SPM during a memory operation when a memory operation for another SPM or the Master Processor is taking place.

The SPM is a relatively simple piece of hardware; the buffering functions described above are commonly done on the CPU cards of most current microcomputer systems. If it were desired to save time in implementation, a standard S-100 bus CPU card could be used as an SPM.

Memory Modules

For the purpose of simplifying the architectural

concept, certain assumptions will be made concerning each Memory Module (MM). Limiting the boundaries on which a given MM may be addressed is particularly useful in making the architecture efficient. The assumption in this architecture is that modules will be addressable on 4K boundaries. This implies that the address assignment of a MM requires only four bits (the most significant hexadecimal digit in the address). This four bit "page number" will later be shown to have advantages in a four processor system. A consequence of this addressing assumption is the requirement that a MM have at least 4K of memory. There is no reason why a MM could not have more than 4K; any integer multiple of 4K would be satisfactory and, in certain cases such as large compilers stored in read only memory (ROM), would have some significant advantages which will be discussed later. In describing the machine architecture, an assumption shall be that all MMs possess 4K of memory.

In preceeding descriptions of the memory architecture, the terms memory module and memory board were used interchangeably. It should be recognized that this need not be the case. A few manufacturers today produce 16K memory boards that are actually four independently addressable 4K blocks of memory. There is no reason why several MMs could not be implemented on one board. This approach, in fact, would have significant cost advantages over putting only one MM on each board, since it would involve only one circuit board and a significant number of components -- including

both power and logic -- should be able to be shared by the MMs on the board.

Each MM "sits" on a bus controlled by the M&PCM. It sits in parallel with every other MM in the system; each MM receives the same signals at the same time. As shown in Figure 3, this bus (which could be in S-100 bus format) carries the 16-bit memory address or the "duplicated" I/O port number (for I/O output operations), the 8-bit data lines, and four discrete signals. These discrete signals are READY, OUT-SYNC, I/O OUTPUT and MEMWR. READY is a signal that is made high (1) when the selected MM has performed the requested memory operation. OUT-SYNC is a signal provided by the M&PCM to indicate that the values on the address lines, data lines (for memory write or I/O output), I/O OUTPUT and MEMWR are stable and can be used by the MMs. I/O OUTPUT and MEMWR are signals generated by the M&PCM to indicate to the MMs that the operations being performed are I/O output (to the MM ports) and memory write respectively. If neither I/O OUTPUT nor MEMWR are high when OUT-SYNC goes high, the operation is a memory read.

A part of each memory module is a row of eight switches. They are used to set, in binary, the "module number." This module number is equivalent to the I/O port number that the module will respond to when signal I/O OUTPUT from the M&PCM is high. It might be noted at this point that multiple MMs on a single board could effectively share this switch; a 4-module board could have one switch,

the setting of which would be taken as the module number of the "first" MM on the board. The other three MMs would then be assumed to have the next three consecutive module numbers. In this architecture, incidently, all MMs are intended to have unique module numbers -- an I/O output to a given port should affect no more than one MM.

Each MM has associated with it an 8-bit "address/bank latch"; a latch can be visualized as a very small memory. The latch associated with the MM is used to hold the current address to which the MM will respond (as a 4-bit value equal to the high-order hexadecimal digit (four bits) of the address) and also the current bank membership (as four bits). The value is set by means of an I/O output operation from the M&PCM. The high-order four bits in the latch will represent the address and the low-order four bits of the bank membership. The contents of the latch is used to determine whether the MM is "selected," or active, for the memory operation taking place.

The behavior of the MM is as follows:

1. When the low-order eight bits of the address lines equal the value set on the module number switches, I/O OUTPUT is high, and OUTPUT-SYNC goes high, the contents of the address/bank latch on the MM is set from the value on the data bus, bit for bit.
2. When the high-order four bits of the address bus are equal to the high-order four bits in the address/bank latch, the result of a logical AND between the bank bus and

the low-order four bits in the address/bank latch is non-zero (there is a "match" on the bank), and OUTPUT-SYNC goes high, the MM performs one of the following operations. If MEMORY WRITE is high, the MM stores the contents of the data bus in the MM's relative memory location addressed by the low-order twelve bits on the address bus. If MEMORY WRITE is low, the MM obtains the contents of the data byte in its relative memory location addressed by the low-order twelve bits of the address bus and puts it on the data bus. As soon as the operation is concluded, the MM sets READY high for as long as OUTPUT-SYNC remains high.

It should now be easy to see why the choice of four address bits is appropriate for a four bank system. This results in an eight bit latch which can be set in one operation from the data bus. This implies that the M&PCM can set both the response address and the bank membership in a single operation. It should be noted that the same efficiency can be obtained in a system designed for fewer banks and smaller MMs. If, for example, a system requiring only two banks was desired, six bits could be used for the address of the memory module. This would mean that MMs could be set to 1K boundaries; consequently, the memory contained in a MM could be as little as 1K.

The design of the MM does not differ in many respects from the design of a typical S-100 bus memory board on the market today. What is novel in the MM concept is making the bank membership and, particularly, the addressing dynamically

alterable. This frees main storage from many of the constraints of a "uniquely addressed" memory architecture. Different processes can literally occupy the same address space at the same time. The advantages of this architecture shall be discussed later in this chapter.

The correspondence between the number of processors and the number of physical memory banks (four of each) is no coincidence. Each memory bank serves a specific processor. MMs to be accessed by SPMs 1, 2 and 3 must be assigned to banks 1, 2 and 3 respectively. MMs assigned to bank 0 can be accessed by the Master Processor. A given MM may be assigned to one or more processes at a given point in time. In fact, a MM may be assigned to no physical bank for most of the time.

In order to clarify the effect of the Dynamic Memory Banking system, Figure 4 is provided. This figure in no way represents the physical structure of the system, but attempts to assist an intuitive understanding of logical and physical banking as maintained by both the hardware and the operating system. Fourteen memory modules are assigned to the seven logical banks shown. Logical banks are approximately equivalent to user processes; logical bank "a" in this snapshot is apparently the operating system, since it is assigned to physical bank 0 (accessed by the Master Processor).

Module FF, a 16K ROM module (possibly a BASIC language interpreter), is being shared by four users -- those occupying

Logical Banks

Module Number	Physical Bank(s)	Logical Bank(s)	Assigned Address (high-order digit)
00	0	a	0
01	none	d	4
0B	1	f	6
2F	0	a,b	3
35	none	g	5
74	1	f	5
79	0	a	1
7A	2	c	4
A1	none	g	4
B2	0	a,b	2
B3	2	c	5
DE	1	f	4
F3 (4K ROM)	3	b,e	0
FF (16K ROM)	1,2	c,d,f,g	0-3

Figure 4. Intuitive "Snapshot" of Dynamic Memory Banking System

logical banks c, d, f and g. As users c and f are both currently running (on SPMs 2 and 1 respectively), module FF is assigned to both physical banks 1 and 2. You may note that modules 2F and B2 are shared by logical banks a (the operating system) and b (a user process). This situation may have occurred as a result of process b requesting the operating system to write out a large block of data to disk storage. At the point at which this "snapshot" was taken, the operating system has assigned the two modules containing this block of data to physical bank 0 in order that the Master Processor can access them.

The assignment of modules to physical banks is shown both in the list of modules and by the arrows extending from the M&PCM in the figure to the appropriate logical banks. It must be emphasized that Figure 4, unlike Figure 1 depicting the Cromemco system, is only a "snapshot"; a few milliseconds later, the assignment of modules to physical and even logical banks may be radically different.

Memory & Processor Control Module

The Memory & Processor Control Module (M&PCM) can be considered to be the "center" of the architecture. As implied by its name, it is the means by which the Master Processor exercises its control over the dynamic memory banking system and over the SPMs in the system. If it is to avoid degrading the system performance, it must perform its functions -- particularly its memory operations -- efficiently. Ideally, memory operations should be performed

in one clock period -- the same amount of time normally required to access memory in a typical uniprocessor microcomputer system.

The M&PCM will be a reasonably complex collection of digital logic circuits. A designer would be presented with a number of options relating to additional capabilities of the M&PCM, or other options relating to its efficiency. The operational behavior of the M&PCM as presented below is believed to represent a reasonable compromise between the "bare minimum" and the ideal.

It is recognized that some of the operations discussed below will require more than one clock period to accomplish, both because of their complexity and because of the occasional necessity to "wait" until specific conditions are fulfilled. No attempt will be made to specify precise timing requirements; these are left to the implementor.

Capabilities. The capabilities of the M&PCM to affect SPMs and MMs should first be reviewed. The M&PCM has control over the RESET and READY lines to each SPM. This implies that it possesses the capability to completely reset the SPM, causing it to "start over" at address zero. Control over the READY line enables it to suspend the operation of the SPM in the middle of a machine cycle. There are several possible justifications for such a suspension: an I/O instruction may have to be trapped, a memory write or read may have to be delayed due to memory contention, or the Master Processor could direct the suspension. The data bus

to each SPM is also controlled by the M&PCM during memory read and I/O input operations. The M&PCM has complete control over the memory bus (except for the data bus during memory read operations). It can change the address range and bank membership of any MM. It can perform a memory read or write to any MM whose bank membership is non-zero. As in the case of control over the SPMs, some of the control over the memory system is invoked automatically within the M&PCM (as in the case of normal memory reads and writes) and some is at the direction of the Master Processor (resetting of MM address/bank latches via I/O OUTPUT operations from the M&PCM).

The M&PCM obtains certain information from SPMs and MMs, some of which is passed on to the Master Processor and some of which is used directly by the M&PCM. Signals obtained from the SPMs for relay to the Master Processor include Interrupt Enabled (INTE), I/O Output (OUT), I/O Input (INP) and Halt Acknowledge (HLTA). In addition, the low-order eight bits of the address bus (containing the I/O port number) and the contents of the data bus (for I/O output operations) are passed on to the Master Processor during an I/O trap. Information obtained from the SPMs for use within the M&PCM includes address bus and (for memory write operations) data bus contents, as well as the status lines for First Instruction Byte Fetch (M1), I/O Output (OUT), I/O Input (INP), Memory Read (MEMR), and Memory Write (WO). Information obtained by the M&PCM from the MMs

consists of the READY line and the data bus (for memory reads). The READY line is used by the internal circuitry of the M&PCM, and the contents of the data bus is passed on to the processor requesting the memory read.

M&PCM/Master Processor relationship. It is necessary to discuss the relationship of the M&PCM and the Master Processor. The M&PCM services the memory reads and writes from the Master Processor bus as it does those from the SPMs. The Master Processor/M&PCM interface does not, however, trap I/O instructions on the Master Processor's bus. In fact, the M&PCM is an I/O "device" on the Master Processor's bus. By performing input operations from the ports implemented on the M&PCM, the Master Processor can obtain information regarding the operation of the SPMs. By performing output operations to the ports implemented on the M&PCM, the Master Processor can "command" the M&PCM to take certain control actions in regard to the SPMs or MMs. The following discussion of I/O port assignments on the M&PCM is merely a suggestion; there are many ways in which these ports can be arranged.

Input ports (from the M&PCM to the Master Processor bus) are used by the Master Processor to obtain information regarding the status of SPMs. For each SPM, one "status port" and two "data ports" are implemented in the M&PCM. The status port provides, in bit form, the following:

Bit 0:	INTE
Bit 1:	IN operation pending
Bit 2:	OUT operation pending
Bit 3:	HLTA

Bit 4: Memory error
 Bit 5: SPM suspended
 Bits 6-7: Unused

One input data port will contain the "port number" for an I/O operation initiated by the SPM that has been trapped by the M&PCM. Obviously, this data port will contain valid information only when Bit 1 or Bit 2 of the input status port is on. The other data port will contain (for OUT instructions executed by the SPM) the data to be output.

The generation of "IN operation pending," "OUT operation pending," "Memory error" and "SPM suspended" signals will be described later. "INTE" and "HLTA" are simply the current values of those signals as received directly from the SPM. In addition to turning on the appropriate bits in the status port, "IN operation pending," "OUT operation pending," "HLTA," and "Memory error" also generate an interrupt to the Master Processor.

Output ports (from the Master Processor to the M&PCM) are used by the Master Processor to command the M&PCM. For each SPM in the system, one "command port" and one data port are implemented in the M&PCM). The values sent to the command port will have the following meanings to the M&PCM:

Bit 0: Suspend SPM at instruction boundary
 Bit 1: Release SPM
 Bit 2: Jam Data
 Bit 3: Reset SPM
 Bits 4-7: Unused

The operations initiated by the above signals will be described soon. The data port is used by the Master Processor to "send" eight bits of data to the M&PCM. Its

use is required during Release SPM (after IN instructions) and Jam Data operations.

In addition to the ports used for control of the SPMs, there are two data ports in the M&PCM used by the Master Processor to exercise control over the memory resources. These ports are always used in conjunction with each other. One is used to receive the address/bank data to be sent to a MM, and the other is used to receive the MM number.

Operational behavior

The purpose of the following discussion is to describe the actions taken by the M&PCM under various conditions, noting in what ways the Master Processor, SPMs and MMs are affected and respond.

Memory reconfiguration. The Master Processor initiates this operation by performing an OUT instruction to the address/bank data port on the M&PCM with the desired contents of the target address/bank latch in the accumulator. It then loads the accumulator with the module number of the target MM and performs an OUT instruction to the module number data port on the M&PCM. At any time the M&PCM receives a module number from the Master Processor, it puts the number on the address bus, the contents of the address/bank data port on the data bus, sets I/O OUTPUT high, and, after a short pause to ensure stable data, sets OUTPUT-SYNC high for the remainder of the current clock period. Any pending memory operations from SPMs are delayed

for the clock period. The addressed MM, on recognizing the operation, loads its address/bank latch from the data bus. This completes the assignment of the MM to a particular address range and bank(s).

Memory read from SPM. It should be noted that the functioning of this operation is identical to a memory read from the Master Processor except that, as the Master Processor has priority, it will not be delayed due to memory contention.

The M&PCM recognizes a memory read operation from an SPM by a high status on the MEMR line. The address bus from the SPM is latched by the M&PCM and the READY line to the SPM is made low, indicating not ready. "Tie-breaking" logic in the M&PCM determines if the SPM is the highest priority processor requesting a memory operation during this clock period. The Master Processor (Bank 0) has the highest priority; SPMs (Banks 1, 2, and 3) have descending priorities with increasing bank numbers. If the requesting SPM is not the highest priority processor requesting a memory operation in the current clock period, the operation is delayed until it is. At that time, the latched address is put on the memory system address bus, the bank line corresponding to the processor making the request is made high, and, after a pause to ensure signal stability, OUTPUT-SYNC is made high. The MM whose address/bank latch matches properly with the high-order four bits of the address bus and the bank bus obtains the contents of the required byte of memory and, after OUTPUT-SYNC goes high, puts that byte on the data bus

and makes READY high and keeps it high until OUTPUT-SYNC drops. This completes the operation as far as the MM is concerned. The M&PCM waits until READY goes high, pauses to ensure signal stability, and then passes the contents of the memory data bus through to the data bus of the requesting SPM and makes the READY signal to the SPM high. This is the normal completion of the operation for the system. If, however, the READY signal on the memory bus never goes high an error condition exists. Either the addressed MM has malfunctioned, or the SPM has attempted to read from memory that was not assigned to its bank. If the READY signal does not go high by the end of the clock period, the M&PCM turns on the "Memory Error" (Bit 4) and "SPM suspended" (Bit 5) bits in the input status port on the Master Processor bus, creating an interrupt in the process, and makes the high-order eight bits of the memory address available to the Master Processor at the "port number" data port. The READY line to the SPM is left low pending action by the operating system running on the Master Processor.

Memory write from SPM. This operation is very similar to the memory read described above; therefore, only the differences will be described. The source of the data is, of course, the SPM. The 8080 does not put out the data onto the bus until the second half of the clock cycle. When this finally occurs, the contents will be immediately sent direct to the data bus of the memory system, provided, of course, that higher priority memory operations have not

delayed the operation. If a delay has occurred, the data bus will be latched by the M&PCM for use as soon as the operation can proceed. In any case, the OUTPUT-SYNC line of the memory bus is not made high until the data from the SPM has been put on the data bus of the memory system and has been allowed to stabilize. Needless to say, it is not necessary for any data to be passed back to the SPM. The READY line on the SPM can be made high as soon as the READY line on the memory bus goes high, indicating that a MM has responded. Memory error activities are the same as for the read operation.

IN instruction executed by SPM. The M&PCM recognizes this by a high condition on the SPM INP status line. The contents of the low-order eight bits of the SPM address bus is passed to the latch serving the "port number" port for that SPM on the Master Processor bus. Simultaneously, the "IN operation pending" and "SPM suspended" bits are turned on in the status port representing the appropriate SPM on the Master Processor bus, generating an interrupt to the Master Processor. The READY line to the SPM is left low pending action by the Master Processor.

OUT instruction executed by SPM. This operation is very similar to the IN operation; only the differences will be mentioned. In addition to the port number, the contents of the SPM data bus during the second half of the clock period will be made available to the Master Processor by means of the the data port appropriate to the SPM on the

Master Processor bus. The bit turned on in the status port on the Master Processor bus is Bit 2 (OUT operation pending).

Master Processor command: Suspend SPM at instruction boundary. This command is initiated when the Master Processor performs an OUT instruction to the command port corresponding to the target SPM. The OUT is performed with a 1 in bit position 0 of the accumulator. This sets the M&PCM circuitry for the operation. Operations by the SPM continue normally until signal M1 (First Instruction Byte Fetch) goes high, indicating that a new instruction is being fetched. At this point, the memory read operation being requested is blocked and the "SPM suspended" bit (Bit 5) in the status port on the Master Processor bus is turned on. The READY line to the SPM is left low, suspending operation of the SPM pending further action by the Master Processor.

Master Processor Command: Release SPM. This command bit (Bit 1 of the command port for each SPM) is used to "release" an SPM from a suspension. The suspension may have occurred in response to an earlier Master Processor command, an I/O trap, or a memory error. By performing an OUT instruction to the command port for the SPM with bit 1 on in the accumulator, the Master Processor releases the SPM for normal operation. When receiving this command, the M&PCM waits until the next clock period and either performs the requested memory operation or (if the halt was for an I/O instruction trap) simply releases the SPM by making READY high. In the case of an IN instruction trap, the value last

sent to the appropriate data port on the M&PCM by the Master Processor is placed on the SPM data bus prior to making READY high.

Master Processor command: Jam Data. The Jam Data command bit (Bit 2) is used to force a byte of data onto the data bus of the SPM. Primarily used in conjunction with a "Release SPM" command, its main use is in inserting a restart operation code into an SPM as the first (and only) instruction byte, effectively creating an interrupt, whether or not interrupts are disabled in the SPM. The M&PCM obtains the byte of data to be "jammed" from the data port on the Master Processor bus.

Master Processor command: Reset SPM. This command bit (Bit 3) enables the Master Processor to reset an SPM. The Master Processor initiates it by performing an output operation to the command port with bit 3 of the accumulator turned on. The M&PCM simply passes the value of the command bit through to the RESET line of the SPM.

Simultaneous commands: Release SPM and Suspend SPM at instruction boundary. If simultaneous "Release SPM" and "Suspend SPM at instruction boundary" commands are given to the M&PCM, the "Release SPM" will take precedence until M1 goes low (off). When M1 again goes high for the fetch of the following instruction, the SPM will again be suspended. This "lock-step" mode of SPM operation is required for proper functioning of process switches as described in Chapter 6.

Examples of operation

The preceeding discussion has defined the operational behavior of the system and, in particular, the M&PCM. What has actually been described is a set of hardware "primitive operations" and, although one could justify postponing a discussion of how these primitives are used by operating systems software in creating larger logical operations, it would be wise to present some examples of larger operations that might be used in order to reinforce an understanding of the primitive operations. Two logical operations will be described: the IN instruction trap and the creation of an interrupt to an SPM.

IN instruction trap. It is clear by now that IN and OUT instructions in user processes must be "trapped" and performed by the Master Processor. As IN requires more complex activity, it will be described; the OUT operation is performed in a similar but simpler way.

The operation of the IN instruction trap has been described above. It shall therefore be used as a starting point. When an SPM executes an IN instruction, the operation is trapped and the Master Processor is interrupted by the M&PCM. This interrupt creates a call to one of the restart locations in low memory; the particular purpose of the routine at that location is to service interrupts from the M&PCM. Depending on the design of the M&PCM, the particular restart generated may implicitly provide the identity of the SPM for which service is required; if this is not done, the

interrupt routine must first determine (by inputting from each status port) which status port generated the interrupt -- i.e., which SPM requires service. Once finding the port and determining that it is an "IN operation pending" that requires service, the Master Processor "reads" the I/O port number specified by the SPM from the "Port Number" port on the M&PCM. This port number will probably have to be "mapped" to another number. For example, a user program may be written for use of a terminal that can be addressed at port 3B (hexadecimal). When the program is actually being executed, however, the user may be at a terminal addressed at 6F. It is the responsibility of the operating system running on the Master Processor to maintain a port-to-port correspondence list and to use it to map a port number in a user program to a physical port number. The Master Processor then performs an IN operation from the physical port and an OUT operation to the data port appropriate to the SPM being serviced on the M&PCM. The Master Processor then performs a "Release SPM" primitive operation. As described earlier, the M&PCM then puts the byte from the data port just loaded by the Master Processor on the SPM data bus and makes READY high. The SPM continues with conditions identical to what would exist if it had been able to do the IN operation itself.

Create interrupt to SPM. It is often necessary for the operating system to interrupt a user process running on an SPM. There are two possible reasons. One is that the

process has received its quantum of processor time and must be suspended for a process switch. The other is that the user process itself may be interrupt driven, and requires interrupts for its proper operation.

In general, "simulation" of an interrupt to a user process proceeds as follows. If it is desired not to interrupt a process while interrupts are disabled, the Master Processor loops, continually testing the status port appropriate to the target SPM until it observes that interrupts are enabled (INTE is on). It then issues a "Suspend SPM at Instruction Boundary" primitive command to the M&PCM. If appropriate, the INTE bit is examined to ensure it did not turn off in the intervening time. Assuming it did not, the operation code of the desired restart instruction is sent to the M&PCM data port by the Master Processor, which then issues a combination "Jam Data" and "Release SPM" command. The M&PCM, as described earlier, will then route the restart byte to the SPM data bus and make READY high.

An option in the design of the M&PCM is presented here. It is possible that certain interrupt-driven user processes may be "difficult" to interrupt using the above procedure. If the process operates the great preponderance of the time in interrupts disabled mode, and only enables interrupts for a short period, it is conceivable that the Master Processor could miss the "window" during which interrupts are enabled. Note, of course, that this problem

does not affect the capability of the Master Processor to interrupt an SPM for a process switch; interrupts can be created to an SPM whether or not interrupts are disabled. The problem arises only in regard to interrupts "simulated" to the user process. The designer may wish to implement a new command bit. This command bit, called "Suspend SPM on Instruction Boundary with Interrupts Enabled", would remain pending until both M1 and INTE were high. At that time, the SPM would be suspended and the Master Processor interrupted. This would ensure that the user process operated exactly as if its processor was receiving the interrupt itself.

5.5 Characteristics of the architecture

In light of the behavioral operation of the system, it is possible to discuss the characteristics of the machine whose architecture has been specified.

Virtualizability

One might at first ask whether the machine is virtualizable. In Chapter 4, it was noted that three instructions in the 8080 instruction set are sensitive. Disable Interrupts (DI), I/O Output (OUT) and I/O Input (IN) can each affect the enforcement of operating system allocation decisions.

The proposed architecture has made those instructions "privileged" by restricting their effect on the system. DI may be executed, and may in fact be used for its original purpose by a user testing interrupt-driven software. It

does not, however, prevent the Master Processor from actually interrupting the process to enforce its allocation of CPU time. The DI could therefore be said to be "simulated" by the system with ultimate efficiency -- it is simulated in the same amount of time it would take to execute it! IN and OUT instructions have also been made privileged by ensuring they are trapped by the system and simulated by the Master Processor. The simulation is of course not as efficient as the "simulation" of the DI instruction, nor does it have to be in order to allow the machine to be virtualizable.

As discussed in Chapter 3, a machine is virtualizable if the sensitive instructions are a subset of the privileged instructions. The sensitive instructions are DI, OUT, and IN. The privileged instruction set that has been created through the architecture is composed of the instructions DI, OUT, and IN. The proper set relationship therefore exists, and the machine is virtualizable, provided only that it meets requirements for a "third generation machine."

From Chapter 3, the essential characteristics of a "third generation machine" are a dual (supervisor-user) operation mode and addressing done relative to a "relocation" register. The dual operation mode has effectively been created as described above. The requirement for a "relocation" register must be interpreted. The purpose of the relocation register is to define the address area allocated to the user process and to provide a means for the process to run as if it were running relative to location zero, irrespective of

where in main storage it actually exists. The hardware in the architecture that has been described effectively simulates a relocation register without any degradation in performance whatsoever. The "relocation register" is actually represented in the address/bank latches in the MMs. The "relocation" of memory accesses, instead of taking place in the addressing circuitry of the machine, actually takes place in the MMs. A user has no way of even attempting to access memory assigned to another user, as it would be in another bank of MMs. If he attempts to read from or write to memory not assigned to his process, no MM will respond and a memory error condition will be detected and trapped by the M&PCM; an example of this would be a process allocated memory from 0000 to 5FFF attempting to read from location 804B.

In fact, the "relocation register" simulation employed in this architecture is significantly more powerful than a simple register. For example, it is not necessary to assign a user process a contiguous address space. Some 8080 programs available today use memory in a couple of non-contiguous pieces, 0000-3FFF and D000-FFFF for example. This architecture allows the system to assign real memory to only those areas of address space where memory is actually required.

It would seem that the requirements for virtualizability have been fulfilled. The machine should be able to support a VMM. It can therefore support a

multiprogramming system.

The architecture also presents other advantages, not only to systems implemented via a VMM but also to processes running under a simple multiprogramming operating system.

Relocation unnecessary

The problems of program relocation are effectively bypassed in this architecture. Since there can be any number of independent 64K address spaces in the machine, and the available memory can be set to any address range desired, there is obviously no need to be able to "move" programs to where there is memory available. Intuitively, the memory is being moved to the program. This has two important advantages. First, current 8080 assemblers and compilers -- which do not generate relocatable code -- can be used to generate object programs that will run on this machine under a multiprogrammed operating system. Secondly, currently available proprietary programs (for which the source language is unavailable for recompilation) may be used on this machine. Eventually, of course, programs developed for this system may wish to use relocatable routines linked together as is the custom with large systems; in this case, it will be necessary to write assemblers and compilers that generate relocatable code, as well as a utility that links them together into a single process.

External storage fragmentation eliminated

Another advantage of the architecture is the

elimination of external fragmentation of main storage. Since MMs can be reassigned to any bank and any address range at will, a given 4K block of memory is simply a 4K block of memory. Any five MMs, for example, can be used to form a bank of 20K of memory for a process. No "checkerboarding" of memory occurs.

There is some question as to whether elimination of external fragmentation does anything except change the external fragmentation to "internal fragmentation." If storage is allocated in blocks of fixed size -- a characteristic of the architecture presented above -- some storage will almost always be wasted at the end of the last allocated block. This waste is referred to as internal fragmentation [Doran 1976], and is considered by some to have the potential for exceeding losses due to external fragmentation in a static allocation system [Shaw 1974]. It is only fair to point out, however, that some third generation systems (the IBM 360, for example) suffer from what can only be described as both external and internal fragmentation. The IBM 360, with the exception of a few late models (Model 67, for example), are static allocation machines. They therefore suffer from external fragmentation. In addition, however, memory can only be allocated in multiples of 2K; this is because the memory boundaries for protection purposes are every 2K [IBM 1968]. If a program requires 56.1K, for example, 58K must be allocated. The unused 1.9K has been lost to what can only be called "internal fragmentation."

In our architecture, internal fragmentation is limited to existence in n MMs, where n is the number of contiguously addressed blocks of memory. Since, in most cases, a process will be assigned only one block of contiguous addresses, n is roughly equivalent to the number of processes p . As, on the average, half of the 4K memory block will be wasted for each instance of internal fragmentation, the amount of memory wasted due to internal fragmentation will be roughly 2pK.

Although this waste is not considered serious (and becomes less and less serious with falling memory costs), there are ways to reduce the effect of internal fragmentation. Both methods described below involve using all or some MMs with smaller amounts of memory.

All MMs could have less memory -- 1K for example. The waste due to each instance of internal fragmentation would then average .5K, one quarter of the value in the proposed architecture. This, of course, would require that capabilities for addressing MMs on 1K boundaries be provided, resulting in either a reduction in the number of processors in the system or a more inefficient procedure for altering address/bank membership of MMs. This method would also require many more MMs to make up a block of memory for use by a process.

The other method is to provide, for example, a few MMs of 3K, 2K and 1K capacity, which the operating system would assign as "tail end" MMs for processes. This would

require a more sophisticated operating system, and, it is believed, would not be worth the additional hardware cost -- it may well cost less to allocate a full 4K MM where only 1K is required than to provide the extra special-purpose MMs. In addition, these special-purpose MMs could not be combined into a single contiguous block due to addressing restrictions; they would therefore not satisfy the very desirable concept of being able to consider a block of memory as simply a block of memory.

The conclusion is that it is best to ignore the relatively small waste due to internal fragmentation.

Number of processes

It should be emphasized that the number of processes that may exist simultaneously in main memory is not limited to the number of physical memory banks, as it was in the Cromemco system. While there can only be four physical banks of memory in the machine, there can be any number of logical banks. When a process initially enters the system, the required memory is assigned by the operating system from a "free list" of MMs. A table maintained by the operating system lists the logical banks of MMs assigned to each process with their respective address assignments, much as illustrated in Figure 4 in Section 5.3. When a process is to receive CPU time, the operating system looks up the identity of the MMs in the logical bank and assigns the proper address and bank membership (appropriate to the particular SPM that will be running it for this quantum of

CPU time) to each of them. When a process is not actually running on an SPM, the MMs assigned to it are not assigned to any physical bank.

The number of processes that can be contained within main storage simultaneously is limited only by the total amount of memory available.

Storage protection

Protection of memory assigned to one process from the actions of another process is implicit in the system architecture. As in the Cromemco and ShackShare systems described in Chapter 4, there is no way for a process to even "see" (address) the memory assigned to other processes. Hence, there is no way in which a memory write of any kind can affect memory not assigned to the process. Unlike the Cromemco and ShackShare systems, there is no way for a user process to switch banks by itself and "get" to memory owned by another process. Storage protection is therefore complete.

A byproduct of this complete storage protection is a high degree of privacy and security between processes in the system. Virtualizable architectures have for some time been an area of interest for designers of "secure" systems [Goldberg 1974].

Fault-tolerance

Another area in which this architecture has strength is tolerance to system faults. In contrast to conventional systems, in which any fault effectively renders the entire

system useless, this architecture is fault-tolerant in two areas: main storage and slave processors. Jack Goldberg, Karl N. Levitt and John H. Wensley, in a paper titled "An Organization for a Highly Survivable Memory," stated that "Main memory is typically the most unreliable system unit (except for mechanical peripherals), but is also the system function that benefits most from fault-tolerance techniques" [Goldberg, J. 1974]. In the architecture proposed above, a failing memory board can simply be removed from the list of usable boards and the system operator notified via the console. A process might also be implemented in ROM whose function it would be to test a suspect memory board and print a diagnostic report. In addition, the master processor, in its "spare time," could continuously test MMs in the free list for memory errors. This last possibility must be looked at carefully, however, as allowing the Master Processor to run continuously will have a significant effect on the degree of memory contention encountered by the SPMs.

SPMs. The architecture is also fault-tolerant with regard to SPMs. A malfunctioning SPM could be taken out of service by the Master Processor by issuing a "Suspend SPM" command or a RESET command and leaving the SPM in that status. The problems of SPM fault detection itself are complex, and are left for future work. The capability of the system to run with one or two SPMs suspended, however, remains. Of course, since each SPM would probably be contained on its own circuit board, an SPM could easily be

replaced after the system is powered down.

Complex scheduling algorithms

Another characteristic of this architecture is related to the selection of CPU scheduling algorithms. There are many possible algorithms for ranking processes competing for CPU time, ranging from very simple round-robin methods to sophisticated algorithms for specific aims [Shaw 1974]. A trade-off between algorithm performance and overhead has always influenced the choice of scheduler. As might be expected, the "better" the algorithm at distributing CPU time optimally, the more overhead it generates.

The proposed architecture includes a CPU (the Master Processor) dedicated to operating system functions. Any time this processor spends on more complex scheduling algorithms will not be taken from CPU time used to process user programs. The overhead cost in implementation of sophisticated scheduling algorithms is greatly reduced. It must again be pointed out, however, that running of the Central Processor does result in some degree of degradation of user processes due to memory contention.

Reduced need for synchronization

In Chapter 2, it was mentioned that multiprocessing requires a significant degree of process "synchronization" to prevent more than one processor from entering "critical sections" of code at the same time. This asymmetric multiprocessing architecture provides that only one processor

(the Master Processor) may execute critical sections of operating system code. This greatly reduces the need for synchronization.

Shared memory

It was mentioned above that the complete isolation between processes provided by this architecture is an advantage where privacy and security is required. Some processes, however, may have a valid need to share storage. It is desirable to be able to provide such a capability for those processes requiring it.

Read/Write memory modules. There is no reason why the operating system could not assign an individual MM to more than one logical -- indeed, more than one physical -- bank. Provision of this capability may make the process table and its use more complex, but that is purely an operating system software problem -- the hardware is fully capable.

In this regard, it should be mentioned that in the case of two processes working on the same area of memory it may be necessary to exclude one process from using the memory during a period when the other process is changing data in that area, as the updated area may not always include valid data during the update [Liskov 1972] [Shaw 1974]. This exclusion must be the basic responsibility of the processes involved, but it may be possible to implement primitive functions in the operating system, addressable through I/O operations, to facilitate this exclusion

operation.

Read-only memory modules. Sharing of standard read-write MMs is not, however, the major anticipated application of MM sharing between processes. Often used utilities and language translators can be stored in read-only-memory (ROM) and assigned a MM number. As standard code can only be executed at one address, the address of this ROM MM could not be altered; the bank membership, however, would function in the same way as a standard MM. A BASIC language interpreter, for example, could be a member of several logical banks simultaneously, resulting in a net saving in main storage.

The interrupt routines invoked for the SPMs by the Master Processor during process switches would also probably be ROM MMs, to prevent their accidental alteration by a malfunctioning SPM. In reference to the problem, mentioned above, of detecting SPM malfunctions, this "process switch ROM" MM could include code to perform a fast operational check of the SPM at every process switch.

If a VMM were to be written for this machine without any "timing dependencies," Theorem 2 from Popek and Goldberg implies that the machine would be "recursively virtualizable" [Popek 1974]. In other words, the VMM could be run under the VMM under the VMM, etc. If, for some reason, this mode of operation was often desired, it would be best to provide the VMM code in a ROM MM; it could then be shared among the various levels of the recursion.

virtual storage implementation potential

Virtual storage methods have been briefly mentioned previously. A virtual storage system makes available to user processes main storage space that does not actually exist in the form of physical memory. It does this by writing out to a fast auxiliary storage device (drum or disk) processes that are in the virtual memory. For example, a machine with only 300K of real memory may have a virtual memory of 1000K. All 1000K is represented on the auxiliary storage device. Obviously, an instruction cannot be executed while it is on the auxiliary storage device -- it must be in main memory. So "pages" of memory are read into real memory when they are required. A typical size for a page ranges from 256 bytes to 4K, depending on the system. When a page is required in main storage, a little-used page in main memory is written back to its location on the auxiliary device, and the needed page is read in in its place. Various algorithms are used to determine which page will be eliminated from main storage.

A key characteristic of a virtual memory system is the need for the system to "map" an address in virtual memory to an equivalent address in real memory. In addition, if the required address does not exist in main memory at the time, action must be taken to initiate a page swap. The detection of the fact that the required page is not in memory is termed a "page fault." Once a page is in real memory, an address translation mechanism must be invoked on

every memory operation. The mechanism, called "Dynamic Address Translation" on IBM machines, refers to a page table in memory maintained by the operating system. It is the responsibility of the operating system to service all page faults, and to keep the page table updated. This is all a rather complex process for both hardware and software.

An examination of the proposed architecture shows that it is very amenable to implementation of a virtual memory system. A page fault is already implemented: it was referred to above as a "Memory Error"; it occurs when there is no MM at the address referenced -- exactly the definition of a page fault! When such a fault occurs, the operating system would determine which page was to be eliminated from main storage and write that page out to auxiliary storage. The needed page would then be read into the newly "freed" MM, and the address/bank latch on the MM would be set to reflect the proper address range and bank membership.

No additional address mapping hardware is required -- the capability already exists in the form of the address/bank latch on each MM. From a different perspective, the Dynamic Memory Banking System is actually a virtual memory system with real addresses freed from the restraints of conventional sequential numbering, such that real addresses can be made equal to virtual addresses.

The proposed architecture did not include the concept of paging between main and auxiliary storage. It is clear from the above discussion, however, that such paging could

be easily implemented.

Dynamic Memory Banking on larger machines

The Dynamic Memory Banking System, as described in the proposed architecture, is ideally suited for a microcomputer. The basic concept behind it is, however, fully applicable to larger machines. The central concept is the address/bank latch implemented in each Memory Module.

The preceeding discussions have clearly covered the advantages of this system, including the ease of implementation for a virtual memory system. There is no reason why the system should not be implemented on large machines, providing the same benefits it brings to microcomputers. Due to a wider data bus on minicomputers and large computers, the address/bank latch could be much wider. This would allow more physical banks and smaller MMs with more flexible addressing.

Limitations

There are some limitations inherent in the proposed architecture.

Number of CPUs. One limitation is in regard to the number of processors that can be supported. The limitation of four processors (Master Processor and three SPMs) was not solely due to the four bit availability for bank information in the address/bank latch in MMs. All processors are continually performing memory operations -- to fetch instructions and to read and write data. The existence of

memory contention between processors has already been briefly discussed. The number of processors the memory system can support is related to the ratio of the average number of clock periods that occur for every memory access made by a processor during normal instruction execution. For example, if a processor initiated a memory operation during every clock period (and the memory could handle one memory access every clock period) for the average instruction, the memory would only support that one processor. In a two processor system of this type, the lower priority processor would never get a memory operation -- it would be continually preempted by the continuous memory operations of the higher priority processor! If, on the average, every other clock period was a memory operation by each processor, the system could support two processors before its memory service capacity was exhausted.

An examination of the clock period:memory operation column in Appendix A shows that the ratio varies between 3.2 and 11.0. The majority of instructions have a ratio value between 3.33 and 4.0. Four processors would therefore seem to be the maximum that the memory system can practically support in its present form. In fact, the lowest priority processor can expect significant degradation due to memory contention.

The conclusion is that the system can support four processors; adding more would probably not increase the throughput. The system is "memory speed bound." It might

be noted at this point that any time the Master Processor is halted, awaiting an interrupt, is time that the SPMs will enjoy better memory performance.

Number of Memory Modules. Another limitation in the memory system architecture regards the number of MMs that can be installed. The limitation is solely due to their numbering system. As they are addressed via their module number which is limited to eight bits, only 256 MMs may be addressed. Assuming 4K for each MM, that limits the system to 1024K, or one megabyte. As mentioned earlier, MMs could be 8K in size. This would allow a total of two megabytes of memory.

It should be emphasized here that the I/O port numbers assigned to MMs in no way conflict with I/O port numbers assigned to I/O interfaces on the Master Processor bus. The MM bus and the Master Processor bus are physically and logically separate; use of a particular port number for a MM does not prevent its use as a number for an I/O interface.

Addressing of shared MMs. Another limitation that should be mentioned concerns shared MMs. Since it is conceivable that two processes sharing the MM could be running (on different SPMs) simultaneously, both processes must address the shared MM at the same address.

Multiple MM responses. No provisions are provided in the hardware to prevent the assignment of more than one MM to a given address range and physical bank. This could

occur in two ways. The port number switches could inadvertently be set the same for two different MMs. Alternatively, the operating system routines whose responsibility it is to manage the memory system could be imperfect and actually set two or more different MMs to the same address range and bank(s). Although alterations to the design of the MMs and the M&PCM could be made to detect simultaneous operation of two or more MMs, it is considered more cost-effective to require that module numbers be set carefully and that operating system routines be thoroughly debugged.

Estimated costs

The following estimated costs for the various components of the system have not been precisely computed; to do so would first require that the circuitry be designed, a task being left for future work. These costs are the author's best estimate, based on experience in microcomputer hardware work and retail microcomputer system and component sales.

Master Processor and bus. The Master Processor and its bus can be purchased essentially complete. A system like the IMSAI 8080 microcomputer is fully satisfactory. The IMSAI has a front panel, CPU card, power supply, chassis, and a S-100 bus with slots for twenty-two interface cards. The IMSAI retails, in kit form, for approximately seven hundred dollars, and may be the most cost-effective way to obtain this circuitry.

Memory & Processor Control Module. The M&PCM is the most complex and unusual component in the architecture. It is therefore the most difficult to price. It is estimated that the cost of the components (primarily normal or fast 7400-series TTL integrated logic circuits) to construct this module should not exceed five hundred dollars. The M&PCM should be constructed on two S-100 bus cards, one of which will be inserted in a slot in the Master Processor bus. The other should be inserted in another IMSAI 8080 chassis (without CPU card) that will serve as the memory system base. Cost of an IMSAI without the CPU card should not exceed six hundred dollars. Cabling will connect the two boards of the M&PCM together, as it will the SPMs to the M&PCM.

Slave Processor Modules. The SPMs are simple CPU cards which can be obtained for no more than two hundred dollars in kit form. The SPMs could reside in the memory chassis and obtain their power from the power supply in that chassis. As mentioned above, they are connected to the M&PCM in the memory chassis by cables.

Memory Modules. Considering dropping prices for memory circuits, it is estimated that a four-module board (containing a total of 16K of memory) could be constructed for not more than four hundred dollars.

Terminal interfaces. S-100 bus interfaces for terminals are a standard item. An interface capable of interfacing two serial interface devices is available for

less than two hundred dollars; the cost per terminal to be implemented is therefore approximately one hundred dollars.

Typical system cost. A typical system constructed from this architecture might contain two SPMs, interfaces for six terminals, and 256K of storage. The total cost for such a system is estimated to be no greater than \$8,600. This cost, of course, does not include the terminals themselves nor any other I/O devices like printers or disk systems.

CHAPTER 6

OPERATING SYSTEMS SOFTWARE DESIGN CONSIDERATIONS

Chapter 5 has specified a virtualizable machine architecture and imparted an intuitive knowledge of its operation. Some specifications for an operating system may now be advanced.

Much work has been done in the area of operating system design; there are many concepts upon which this piece of software may be built. This chapter will not attempt to provide detailed specifications for such software, nor will it even attempt to determine which of the many operating system concepts is most appropriate to this architecture.

There are, however, certain aspects of required operating system functions peculiar to this architecture. This chapter will describe these aspects and attempt to suggest methods by which they may be implemented in the operating system design.

An initial decision must be made between a VMM and an "ordinary" multiprogramming system. Due to machine virtualizability and the other hardware characteristics discussed in Chapter 5, either type of software can be supported. A multiprogrammed operating system fulfills the aims of a multi-user system, which was the primary motivation for the study. This thesis will limit itself to examining

the pertinent aspects of a multiprogrammed operating system as they apply to this machine. VMM specifications will be left for future work.

6.1 Required Functions

All multiprogrammed systems must have certain basic functions dealing with resource allocation and enforcement of allocations. Operating system routines that allocate resources are known as "allocators"; "when the resource is an active unit such as a central processor or data channel, the allocator is usually called a scheduler" [Shaw 1974].

Schedule SPMs

The operating system will require a scheduler for the SPMs. This scheduler may be simple or complex in concept, simply rotating around the active processes in main storage or taking into account such things as job priority, the amount of CPU time a process has already received, the ratio of CPU time used to I/O operations performed and other resources committed to the process. The decision is relatively free of overhead considerations, except for memory contention with the SPMs caused by operation of the Master Processor.

Two peculiarities of the system architecture should be considered in this area. First, there is parallel operation of the scheduling routines and the SPMs. This makes it possible (and very desirable) for the determination of the identity of the next process to receive time on the

SPMs to be made before process switch time arrives. Algorithms that provide this continuous ranking among processes are to be preferred, as this will result in an efficient process switch and consequently greater system throughput. Secondly, all SPMs are not equal in our architecture; they differ in their priority for memory operations. This fact should be taken into account in scheduling and, in fact, can be made good use of in systems containing jobs of different priorities.

Allocate other resources

As in any multiprogrammed system, the other resources of the machine, such as main storage and I/O, must be allocated to processes. The problem of main storage allocation has been made particularly simple in this architecture. It is recommended that a "free list" of Memory Modules be maintained. MMs required by processes should be taken from the "front" of this list and MMs released by processes should be added to the "back." Modules under any suspicion of malfunction are removed from the list and placed in a special logical bank for attention by service personnel or, possibly, a diagnostic routine to be run by either the Master Processor or an SPM when system loading is light.

The system must maintain process lists that specify the MMs (module numbers and corresponding address ranges) and I/O resources assigned to each process. These lists may also carry externally generated process information, such as

job priority, and internally generated process history, such as CPU time used. In short, the process list for each process should contain all information required by the operating system to properly service the job.

process switches

If an appearance of simultaneity in process execution is to be maintained, each process must be given quanta of CPU time on an SPM regularly. After a quantum of time has been allowed, the process must be interrupted, register contents stored, and a new process given control. As described earlier, this operation is known as a process switch. A process switch does generate overhead, as the SPM cannot be working on a user process during it. It is desirable, therefore, that the switch be as efficient as possible.

It is recommended that a special-purpose MM containing both ROM and read-write RAM be provided. This MM would be switched into the SPM's bank when the interrupt is generated. A simple routine in ROM would cause the SPM to store the contents of all registers in the 8080 on the MM and then halt. After the process switch was complete, the operating system would transfer this information to an area provided in the process list. A similar MM (conceivably a different address on the same MM) would be used to load the registers of the SPM with the proper values for the new process.

The entire operation might work like this: A timer

interrupt is generated to the Master Processor. The SPM scheduler routine, invoked by the interrupt, takes over. The Master Processor issues a "Suspend SPM on Instruction Boundary" command to the M&PCM, delays a very short time, reads the status to ensure the SPM has been suspended, and "unbanks" the MM at address zero. It then "banks" the special-purpose ROM-RAM MM, outputs the operation code for a Restart 7 instruction (a single-byte call to location 0038) to the data port on the M&PCM, and issues a combination "Release SPM" and "Jam Data" command to the M&PCM. The Master Processor then proceeds to "unbank" all MMs assigned to the old process. Note that the MMs are not removed from their logical bank -- just the physical bank. While the Master Processor is unbanking MMs, the SPM executes the routine in ROM at location 0038, storing all registers in the RAM provided in the MM. It then halts. The Master Processor, meanwhile, has proceeded to assign to the appropriate physical bank all MMs assigned to the new process except the MM that is assigned address zero. The Master Processor then determines that the SPM has halted, and issues a Reset SPM command to the M&PCM and then lifts it. This causes the SPM to begin execution at location zero in ROM. The routine found at that point causes the SPM to load all registers except the program counter from an area in RAM (Not the same area in which the register contents from the "old" process were stored) and halt. The Master Processor then "lock steps" the SPM through the next few instructions

to load the program counter. A lock step effect can be obtained by issuing a combination "Release SPM" and "Suspend SPM on Instruction Boundary" command, as described in Chapter 5. Once the SPM is fully loaded, including the program counter, the Master Processor unbanks the ROM-RAM MM, banks the address zero MM for the process to be run, and issues a "Release SPM" command. Housekeeping chores that must now be done by the operating system include transferring the saved register contents from the old process to the appropriate area in the process list, identifying the next process to be run, and transferring the register contents for that process to the "new process" RAM area on the ROM-RAM MM, completing the setting up for the next process switch.

It is estimated that overhead due to a process switch should never exceed two milliseconds, and should in most cases be less than a millisecond. The primary source of variation in process switch time is the number of MMs that must be unbanked and banked.

Run input/output

A major function of the operating system is running input/output for processes. This includes allocation of the I/O resources and the actual performance of the I/O operations. Port numbers specified by the user process must be mapped to the physical port. In some cases, status bits may differ between physical and "virtual" I/O devices; these must be mapped also. Spooling of output to printers, etc., may also be implemented in the operating system.

Isolation of user processes from physical I/O also enables the operating system to "translate" I/O requests for a particular device to another device with entirely different operational characteristics. This resembles the ability of virtual machines to "retrofit" new features [Goldberg 1974]; a user process can thereby benefit from improved devices that were not available when the process was written.

Maintain input buffer for processes. Input on the 8080 is normally done character by character. An input character must be "serviced" before the next character arrives at the input interface, or the first character will be lost. Two keys can be pressed in rapid succession on a keyboard, and unless sufficiently frequent quanta of processing time can be allocated to each process, some method must be provided in the operating system to handle this problem. It is recommended that an input buffer be maintained for each process in the form of a FIFO queue. This should also result in more efficient use of CPU time by the process.

Detect end-of-process

There must be some way for the operating system to detect the fact that an end-of-process has occurred. This could be implemented via an output operation to a special port that the operating system would recognize as an end-of-process signal. Alternately, the process could halt with interrupts disabled; as there is no way for the process to restart itself in this position, it is a clear indication that it is done. Both of these methods are appropriate for

batch processing. A user on a terminal could indicate termination of his session by turning off his terminal (which the interface could be wired to recognize), or could issue a "signoff" command that the Master Processor would interpret as termination.

When a process terminates, the operating system must return all MMs assigned to it to the free list and delete the process list from its queue of active processes. History information concerning the operation of the process may optionally be posted to a job accounting file.

6.2 Optional functions

Although the functions discussed above are those essential to operation of the system, there are several optional functions which may be implemented.

Simulate interrupts to processes

If it is desirable to allow user processes to be interrupt driven, provision must be made for simulation of interrupts to user processes. As discussed in Chapter 5, the hardware to allow this is already provided. Since the particular restart operation code to be used is under the control of the Master Processor, there is no need for the priority of interrupts simulated to the process to be the same as the priority of interrupts for the Master Processor.

Provide extended machine interface

It was stated in Chapter 2 that a multiprogrammed operating system provides "extended machines" for user

processes. The extended machine the operating system being discussed will provide is actually very much like a virtual machine. In fact, it "looks" to user processes like "bare hardware"; the bare hardware it simulates is a uniprocessor 8080-based microcomputer system. This extended machine is rather inefficient where I/O is concerned. The reason for this concerns the way in which an 8080-based system normally does I/O.

Figure 5 presents a flowchart for a typical input "driver." The status port is continuously interrogated until the bit indicating a character has been received turns on. The character is then read from the data port. This makes it difficult to make use of the time a process is waiting for input. A multiprogrammed operating system on a large machine simply notes the fact that the process is waiting for input, and blocks the process from further CPU time until that input arrives.

There are complications which make this procedure impractical in processing current 8080 software. Some processes test the status port every so often to see if the operator wishes to interrupt processing. If no character has been received, processing continues. So it is impossible to suspend a process until input is received simply because it requests information from a status port.

A solution may be to implement a more extended machine for use by processes that have been written for it. Such processes would signal via an I/O operation to a

Figure 5. Input Driver Flowchart

special port that they wish to be suspended until input is available. Actually, this more extended machine would suffice for running all processes, as it would retain the ability to handle the currently standard I/O methods. Processes that use the more extended features would improve concurrency in use of system resources.

Output buffers

In discussing essential features of the operating system, provision of input buffers for processes was suggested. While not essential for proper operation of the system, output buffers could also be established. As this would eliminate waiting for slow I/O devices, a process could do significantly more work in its quantum of processing time. If a buffer became full, the process could be blocked from further CPU time until the buffer was almost empty, enabling other processes to make use of more CPU time. This feature could be implemented even if the "more extended machine" described above were not.

Multi-bank memory for single processes

Thus far, discussion has been limited to handling of processes requiring no more than 64K of memory. It may be desirable to implement a means by which user processes can request more than one logical bank of storage and initiate (through the operating system) bank switches. This, although simple in concept, would require a more complex process list and, therefore, more overhead in process switches. It

should therefore be looked at carefully to ensure the benefits are sufficient; planned applications will determine the necessity for this feature.

6.3 Summary

This chapter has discussed required and optional aspects of multiprogrammed operating systems for the proposed machine architecture. Other than those aspects discussed above, the specific design concept is not limited, and the implementor may feel free to use any design philosophy he desires.

It has been noted that the proposed architecture is a microprogrammed machine. The machine is designed to produce a machine whose internal architecture is complex. This is the nature of the machine, and it is clearly intended to be a theoretical machine. A conclusion of previous chapters is that true multiprogrammed systems are possible and practical with currently available 8-bit microprocessors. The following system was computed in Chapter 5 to be very reasonably demanding on the processing power and multiprogramming capabilities.

The Dynamic Memory Access System, an integral component of the proposed architecture, is based on concepts available to larger machines as well. A possible topic for future investigation is the alterations to design required to implement the system on a single microprocessor or a larger

CHAPTER 7

CONCLUSIONS AND EXTENSIONS

7.1 Conclusions

This study is merely the initial step in the realization of a microprocessor based virtualizable machine. Much work, including design and construction of the hardware and implementation of an appropriate operating system, remains to be done. Nonetheless, certain conclusions can be drawn at this point.

It has been shown that two distinctly non-virtualizable microprocessors can be combined in such a way as to produce a machine whose total architecture is virtualizable. This is the central conclusion of this thesis, and is clearly interesting from a theoretical viewpoint. A conclusion of practical interest is that true multiprogrammed systems are possible and practical with currently available 8-bit microprocessors. The estimated typical system cost computed in Chapter 5 is very reasonable considering the processing power and multi-user capabilities provided.

The Dynamic Memory Banking System, an integral component of the proposed architecture, is based on concepts applicable to larger machines as well. A possible topic for future investigation is the alterations in design required to implement the system on a 16-bit minicomputer or a larger

"full-scale" machine.

7.2 Recommendations for future work

An initial study of this type naturally leaves many actions and additional investigations for the future. Some of this further work is obvious, and some is not.

Much work remains to be done before a working prototype is realized. Working from the rough specifications in this study, the implementor must finalize the design of the functional components. The machine must then be physically constructed and debugged. It should be mentioned that several components of this architecture -- the Master Processor and bus, Slave Processor Modules and Memory Modules -- are identical or very similar in design to current S-100 bus microcomputer products. Much design and debugging time can be avoided if advantage is taken of this similarity by simply modifying current product designs. This similarity is actually an advantage of the architecture, as it should reduce significantly the implementation time.

Operating system software must be designed and tested. It may be worthwhile to write a simulator for the machine that will run on an 8080-based system or even a large system. This would facilitate testing of the software prior to completion of the hardware.

As discussed briefly in Chapter 5, the system architecture is well suited for implementation of virtual memory. This area would be primarily a software task, as the essential hardware is already provided for in the basic

architecture.

As described in Chapter 5, memory contention is the limiting factor in the processing power of the system. There are at least two ways in which memory contention can be reduced. The operating system code, since it is executed by the Master Processor, could reside in memory on the Master Processor's bus, rather than in the Dynamic Memory Banking system. The great preponderance of memory accesses by the Master Processor would therefore not go through the M&PCM, resulting in significantly less contention being experienced by the SPMs. This method would require that the Master Processor have the capability to "turn off" the memory access "connection" from its bus to the M&PCM.

The second method involves the use of faster memory. If Memory Modules capable of responding in one half of a clock period were used, the memory system could perform two memory operations in the time it formerly took to do one, and two processors could therefore access the memory system during each clock period. This would also result in significantly reduced memory contention; in fact, the number of SPMs in the system could easily be increased to four or five, if an increase in the size of MMs to 8K or 16K was permissible. It should be noted that this method would require a more sophisticated M&PCM design, as well as more expensive memory components. One limitation should be discussed. As mentioned earlier, an 8080 performing a memory write does not make the data to be written available

until the second half of the clock period. A memory write would therefore not compete for a memory operation until the second half of the clock period during which it was initiated.

A potentially valuable extension to the proposed architecture for certain applications would be the inclusion of more than one variety of microprocessor on the SPMs. For example, each SPM could contain an 8080, a Motorola 6800 and a MOS 6502. A process switch would then also have to select the processor to be used during the next quantum of processing time. As microprocessors are comparatively inexpensive, this would not increase the cost of the system significantly. Such a system would be capable of running object code for any of the installed microprocessors.

Although not at all addressed in this study, it is conceivable that a more sophisticated Memory & Processor Control Module could implement a virtualizable system where all processing is performed by the Master Processor, user processes included. This would require an extremely more complex M&PCM and is believed to be impractical from a cost standpoint at this time. Future developments in LSI circuitry may, however, prompt a reexamination of this matter.

7.3 Closing remarks

The advent of the microprocessor prompted many predictions of rapid development of sophisticated distributed, cooperating and dedicated applications -- in short, the imminent arrival of a new era. Problems in developing architectures and operating systems that could

make use of currently available microprocessors later influenced some persons in the field to observe that perhaps the new era was not as close as it seemed. For example, in 1976 Jean-Loup Baer, in his paper Multiprocessing Systems, noted that "although the trend to distribute processing and to further departure from the von Nueumann concept of a monolithic stored program computer is irreversible, realizations might not be so close as hand as the current hardware technology permits" [Baer 1976].

It is the author's opinion that the architecture proposed in this thesis represents an intermediate step in this transition to fully distributed microprocessor-based systems -- a step which may be taken with current hardware and software technology. It is a step that can be taken now.

REFERENCES

- Anderson, Judith A. and G. J. Lipovski. "A Virtual Memory for Microprocessors." Proceedings of the 2nd Annual Symposium on Computer Architecture. (January 1975), 80-84.
- Baer, Jean-Loup. "Multiprocessing Systems." IEEE Transactions on Computers C-25 (December 1976), 1271-1277.
- Davis, Ruth M. "Evolution of Computers and Computing." Science 195 (March 1977), 1096-1102.
- Denning, Peter J. and G. Scott Graham. "Multiprogrammed Memory Management." Proceedings of the IEEE 63 (June 1975), 924-939.
- Doran, R. W. "Virtual Memory." Computer 9 (October 1976), 27-37.
- Foster, Caxton C. Computer Architecture. New York, Van Nostrand Reinhold, 1976.
- Gaines, R. Stockton. "An Operating System Based on the Concept of a Supervisory Computer." Communications of the ACM 15 (March 1972), 150-156.
- Gagliardi, U. O. "Trends in Computing-System Architecture." Proceedings of the IEEE 63 (June 1975), 858-862.
- Gear, C. William. Computer Organization and Programming. 2nd ed. New York, McGraw-Hill, 1974.
- Giloi, W. K. and H. K. Berg. "Data Structure Architectures -- A Major Operational Principle." Proceedings of the Fifth Annual Symposium on Computer Architecture. (April 1978), 175-181.
- Goldberg, Jack, Karl N. Levitt, and John H. Wensley. "An Organization for a Highly Survivable Memory." IEEE Transactions on Computers C-23 (July 1974), 693-708.
- Goldberg, Robert P. "Survey of Virtual Machine Research." Computer 7 (June 1974), 34-45.
- Gschwind, Hans W. and Edward J. M. McCluskey. Design of Digital Computers. 2nd ed. New York, Springer-Verlag, 1975.

- Hilburn, John L. and Paul M. Julich. Microcomputers/Microprocessors: Hardware, Software, and Applications. Englewood Cliffs, NJ, Prentice-Hall, Inc., 1976.
- Hoare, C. A. R. "Monitors: An Operating System Structuring Concept." Communications of the ACM 17 (October 1976), 549-557.
- Intel Corporation. 8080/8085 Assembly Language Programming Manual. Santa Clara, CA, Intel Corporation, 1977.
- International Business Machines Corporation. IBM System/360 Principles of Operation. 8th ed. Poughkeepsie, NY, International Business Machines, 1968.
- _____. IBM System/370 Principles of Operations. 4th ed. Poughkeepsie, NY, International Business Machines Corporation, 1973.
- Katzan, Harry Jr. Operating Systems -- A Pragmatic Approach. New York, Van Nostrand Reinhold Company, 1973.
- Lipovski, G. Jack. "On a Varistructured Array of Microprocessors." IEEE Transactions on Computers C-26 (February 1977), 125-138.
- Liskov, Barbara H. "The Design of the Venus Operating System." Communications of the ACM 15 (March 1972), 144-149.
- McGlynn, Daniel R. Microprocessors -- Technology, Architecture, and Applications. New York, John Wiley & Sons, 1976.
- McKinney, J. M. "A Survey of Analytical Time-Sharing Models." Computing Surveys 1 (June 1969), 105-116.
- Mowle, Frederic J. A Systematic Approach to Digital Logic Design. Reading, MA, Addison-Wesley, 1976.
- Osborne, Adam and Associates. An Introduction to Microcomputers. Berkeley, CA, Adam Osborne and Associates, 1975.
- Popek, Gerald J. and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures." Communications of the ACM 17 (July 1974), 412-421.
- Processor Technology Corporation. Sol Systems Manual. Emeryville, CA, Processor Technology Corporation, 1977.

Randall, B. and C. J. Kuehner. "Dynamic Storage Allocation Systems." Communications of the ACM 11 (May 1968), 297-306.

Rao, Guthikonda V. Microprocessors and Microcomputer Systems. New York, Van Nostrand Reinhold Company, 1978.

Rosen, Saul. "Electronic Computers: A Historical Survey." Computing Surveys 1 (March 1969), 7-36.

Sippl, Charles J. Microcomputer Handbook. New York, Mason/Charter, 1977.

Shaw, Alan C. The Logical Design of Operating Systems. Englewood Cliffs, NJ, Prentice-Hall, Inc., 1974.

Soucek, Branko. Microprocessors & Microcomputers. New York, John Wiley & Sons, 1976.

Tanenbaum, Andrew S. Structured Computer Organization. Englewood Cliffs, NJ, Prentice-Hall, Inc., 1976.

Teorey, Toby J. and Tad B. Pinkerton. "A Comparative Analysis of Disk Scheduling Policies." Communications of the ACM 15 (March 1972), 177-184.

APPENDIX A

INTEL 8080 INSTRUCTION SET SUMMARY

<u>Instruction</u>	<u>Mnemonic</u>	<u>Op Code</u>	Clock Cycle: Memory Access
			<u>Ratio</u>
Call (Unconditional)	CALL	CD	3.40
Call (on condition)	various	various	
condition not met:			3.67
condition met:			3.40
Return (Unconditional)	RET	C9	3.33
Return (on condition)	various	various	
condition not met:			5.00
condition met:			3.67
Input	IN	DB	5.00
Output	OUT	D3	5.00
Load Register Pair Immediate	LXI	various	3.33
Push Register Pair	PUSH	various	3.67
Pop Register Pair	POP	various	3.33
Store Accumulator Direct	STA	32	3.25
Load Accumulator Direct	LDA	3A	3.25
Exchange DE and HL	XCHG	EB	4.00
Exchange HL and top stack entry	XTHL	E3	3.60
Load Stack Ptr from HL Pair	SPHL	F9	5.00
Load Program Counter from HL Pair	PCHL	E9	5.00
Double Add Register Pair to HL Pair	DAD	various	10.00

Clock Cycle:
Memory Access
Ratio

<u>Instruction</u>	<u>Mnemonic</u>	<u>Op Code</u>	
Store Accumulator Indirect	STAX	various	3.50
Load Accumulator Indirect	LDAX	various	3.50
Increment Register Pair	INX	various	5.00
Decrement Register Pair	DCX	various	5.00
Move (register to register)	MOV	various	5.00
Move (to or from memory)	MOV	various	3.50
Move Immediate (to register)	MVI	various	3.50
Move Immediate (to memory)	MVI	36	3.33
Halt	HLT	76	7.00
Increment Register	INR	various	5.00
Decrement Register	DCR	various	5.00
Increment Memory	INR	34	3.33
Decrement Memory	DCR	35	3.33
Arithmetic/Logical Operations (register to accumulator)	various	various	4.00
Arithmetic/Logical Operations (memory to accumulator)	various	various	3.50
Arithmetic/Logical Operations (immediate)	various	various	3.50
Rotate Accumulator	various	various	4.00
Jump (unconditional)	JMP	C3	3.33
Jump (on condition)	various	various	3.33
Complement Accumulator	CMA	2F	4.00
Set Carry Bit	STC	37	4.00
Complement Carry Bit	CMC	3F	4.00
Decimal Adjust Accumulator	DAA	27	4.00

<u>Instruction</u>	<u>Mnemonic</u>	<u>Op Code</u>	Clock Cycle:
			Memory Access Ratio
Store HL Direct	SHLD	22	3.20
Load HL Direct	LHLD	2A	3.20
Enable Interrupts	EI	FB	4.00
Disable Interrupts	DI	F3	4.00
No Operation	NOP	00	4.00

Note: The above table was compiled partially from information given in the 8080/8085 Assembly Language Programming Manual, Intel Corporation, Santa Clara, CA, 1977.

5	VI1	Reserved Interrupt Line # 1
6	VI2	Reserved Interrupt Line # 2
7	VI3	Reserved Interrupt Line # 3
8	VI4	Reserved Interrupt Line # 4
9	VI5	Reserved Interrupt Line # 5
10	VI6	Reserved Interrupt Line # 6
11	VI7	Reserved Interrupt Line # 7
12	READY	READY # 1
13-17	TO BE DEFINED	
18	STAT DSB	STATUS DISABLE
19	EXT DSB	EXTERNAL CONTROL DISABLE
20	UNPROT	UNPROTECTED
21	SS	STACK STEP
22	ADD DSB	ADDRESS DISABLE
23	EO DSB	DATA OUT DISABLE
24	02	PHASE 2 CLOCK

APPENDIX B

S-100 BUS LINE DESIGNATIONS

<u>LINE</u>	<u>SYMBOL</u>	<u>SIGNAL NAME</u>
1	+8V	+8 Volts
2	-16V	-16 Volts
3	XRDY	EXTERNAL READY
4	VIO	Vectored Interrupt Line # 0
5	VI1	Vectored Interrupt Line # 1
6	VI2	Vectored Interrupt Line # 2
7	VI3	Vectored Interrupt Line # 3
8	VI4	Vectored Interrupt Line # 4
9	VI5	Vectored Interrupt Line # 5
10	VI6	Vectored Interrupt Line # 6
11	VI7	Vectored Interrupt Line # 7
12	XRDY2	EXTERNAL READY # 2
13-17	TO BE DEFINED	
18	<u>STAT DSB</u>	STATUS DISABLE
19	<u>C/C DSB</u>	COMMAND/CONTROL DISABLE
20	UNPROT	UNPROTECT
21	SS	SINGLE STEP
22	<u>ADD DSB</u>	ADDRESS DISABLE
23	<u>DO DSB</u>	DATA OUT DISABLE
24	O2	PHASE 2 CLOCK

<u>LINE</u>	<u>SYMBOL</u>	<u>SIGNAL NAME</u>
25	O1	PHASE 1 CLOCK
26	PHLDA	HOLD ACKNOWLEDGE
27	PWAIT	WAIT
28	PINTE	INTERRUPT ENABLE
29	A5	Address Line # 5
30	A4	Address Line # 4
31	A3	Address Line # 3
32	A15	Address Line # 15 (MSB)
33	A12	Address Line # 12
34	A9	Address Line # 9
35	DI01	Data In/Out line # 1
36	DI00	Data In/Out line # 0
37	A10	Address Line # 10
38	DI04	Data In/Out Line # 4
39	DI05	Data In/Out Line # 5
40	DI06	Data In/Out Line # 6
41	DI02	Data In/Out Line # 2
42	DI03	Data In/Out Line # 3
43	DI07	Data In/Out Line # 7
44	SM1	MACHINE CYCLE 1
45	SOUT	OUTPUT
46	SINP	INPUT
47	SMEMR	MEMORY READ
48	SHLTA	HALT ACKNOWLEDGE
49	<u>CLOCK</u>	CLOCK
50	GND	GROUND

<u>LINE</u>	<u>SYMBOL</u>	<u>SIGNAL NAME</u>
51	+8V	+8 Volts
52	-16V	-16 Volts
53	SSWI	SENSE SWITCH INPUT
54	<u>EXT CLR</u>	EXTERNAL CLEAR
55	RTC	REAL TIME CLOCK
56	<u>STSTB</u>	STATUS STROBE
57	<u>DIG1</u>	DATA INPUT GATE # 1
58	FRDY	FRONT PANEL READY
59-64	TO BE DEFINED	
65	<u>MREQ</u>	MEMORY REQUEST
66	<u>REF</u>	REFRESH
67	<u>PHANTOM</u>	PHANTOM DISABLE
68	MWRITE	MEMORY WRITE
69	PS	PROJECT STATUS
70	PROT	PROTECT
71	RUN	RUN
72	PRDY	PROCESSOR READY
73	<u>PINT</u>	INTERRUPT REQUEST
74	<u>PHOLD</u>	HOLD
75	<u>PRESET</u>	RESET
76	PSYNC	SYNC
77	<u>PWR</u>	WRITE
78	PDBIN	DATA BUS IN
79	A0	Address Line # 0
80	A1	Address Line # 1
81	A2	Address Line # 2

<u>LINE</u>	<u>SYMBOL</u>	<u>SIGNAL NAME</u>
82	A6	Address Line # 6
83	A7	Address Line # 7
84	A8	Address Line # 8
85	A13	Address Line # 13
86	A14	Address Line # 14
87	A11	Address Line # 11
88	DIO2	Data In/Out Line # 2
89	DIO3	Data In/Out Line # 3
90	DIO7	Data In/Out Line # 7
91	DIO4	Data In/Out Line # 4
92	DIO5	Data In/Out Line # 5
93	DIO6	Data In/Out Line # 6
94	DIO1	Data In/Out Line # 1
95	DIO0	Data In/Out Line # 0
96	SINTA	INTERRUPT ACKNOWLEDGE
97	<u>SWO</u>	WRITE OUT
98	SSTACK	STACK
99	<u>POC</u>	POWER-ON CLEAR
100	GND	GROUND

Note: The above was extracted from Appendix AVII, Sol SYSTEMS MANUAL, Processor Technology Corporation, Emeryville, CA, 1977.