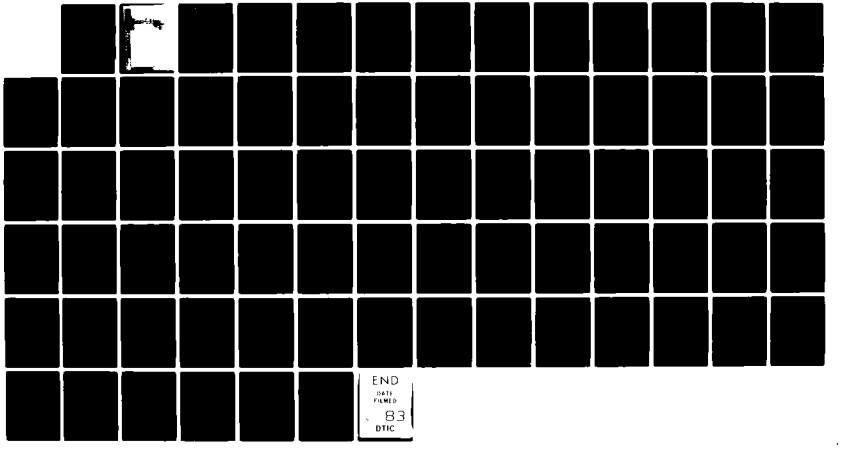
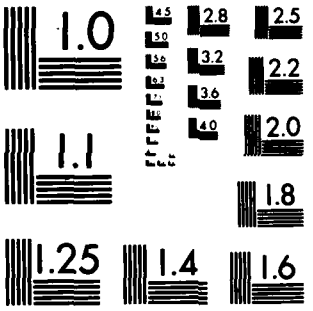
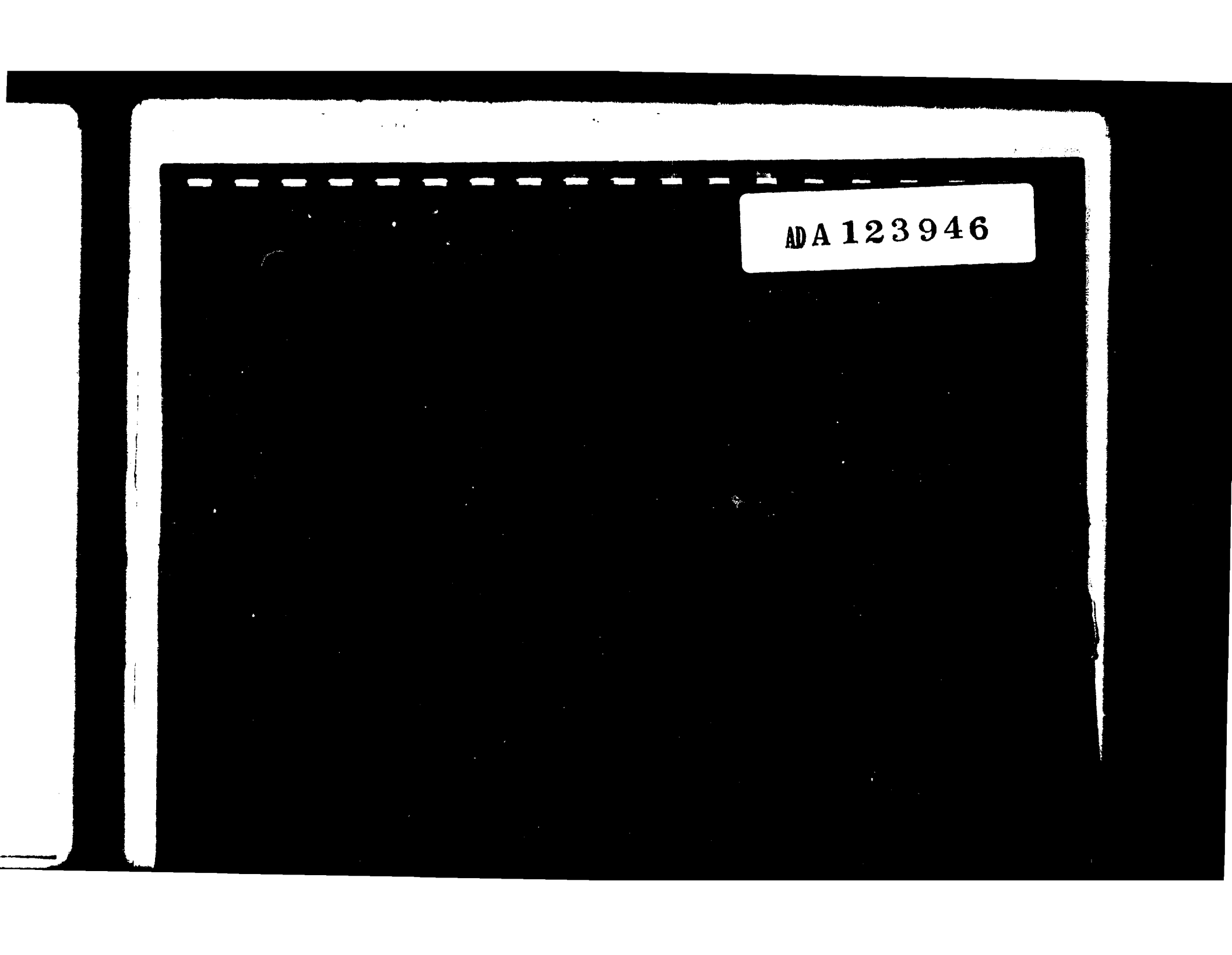
AD-A123 946 REALIZING A LARGE MEMORY SPACE ON A MICROPROCESSOR SYSTEM(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB E F PFLUG JUL 80 R-887 N00014-79-C-0425 1/1

UNCLASSIFIED F/G 9/2 NL



END
DATE FILMED
83
DTIC





UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A123946	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) REALIZING A LARGE MEMORY SPACE ON A MICROPROCESSOR SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER R-887; UILU-ENG 80 2219
7. AUTHOR(s) Elmer Frederick Pflug, III		8. CONTRACT OR GRANT NUMBER(s) N00014-79-C-0425
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		12. REPORT DATE July, 1980
		13. NUMBER OF PAGES 66
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Microprocessor System
Memory Addressing
Virtual Memory

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The memory space of current microprocessors is limited by the number of address bits inherent in the architecture. Several applications of microprocessors require large addressing space, for example, in multiple microprocessor systems with a single shared memory.

This research is focussed on obtaining cost-effective ways to achieve a large addressing space. Several different ways are suggested for this

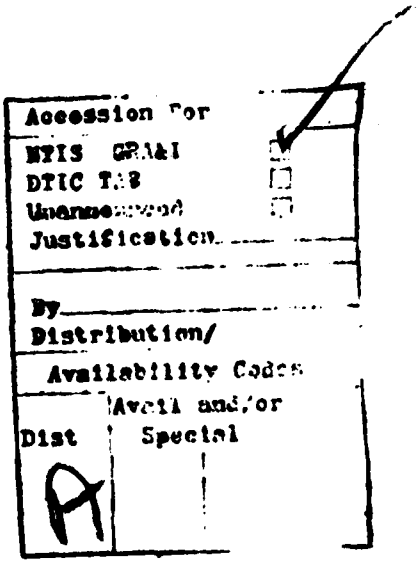
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (continued)

purpose. Applicability of these results to three real microprocessors is discussed. The microprocessors discussed are Intel 8080 Zilog Z80 and Motorola 6800. It is concluded that Intel 8080 and Z80 can address a large memory space with very little additional hardware, while Motorola 6800 requires much more hardware.

A large memory is usually implemented as two-level hierarchy for reasons of cost-effectiveness. This research also focusses on the ways to organize a two-level virtual memory system for microprocessors. It is concluded that current microprocessors require a large amount of external hardware to support a virtual memory system. Even with this additional hardware, the virtual memory system is not as general as in large computer systems. The restrictions imposed on the system to correctly support a virtual memory system on a microprocessor are also discussed.



UILU-ENG 80-2219

REALIZING A LARGE MEMORY SPACE ON A MICROPROCESSOR SYSTEM

BY

Elmer Frederick Pflug, III

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract N00014-79-C-0424.

REALIZING A LARGE MEMORY SPACE ON A MICROPROCESSOR SYSTEM

By

Elmer Frederick Pflug, III

Abstract

The memory space of current microprocessors is limited by the number of address bits inherent in the architecture. Several applications of microprocessors require large addressing space, for example, in multiple microprocessor systems with a single shared memory.

This research is focussed on obtaining cost-effective ways to achieve a large addressing space Several different ways are suggested for this purpose. Applicability of these results to three real microprocessors is discussed. The microprocessors discussed are Intel 8080 Zilog Z80 and Motorola 6800. It is concluded that Intel 8080 and Z80 can address a large memory space with very little additional hardware, while Motorola 6800 requires much more hardware.

A large memory is usually implemented as two-level hierarchy for reasons of cost-effectiveness. This research also focusses on the ways to organize a two-level virtual memory system for microprocessors. It is concluded that current microprocessors require a large amount of external hardware to support a virtual memory system. Even with this additional hardware, the virtual memory system is not as general as in large computer systems. The restrictions imposed on the system to correctly support a virtual memory system on a microprocessor are also discussed.

ACKNOWLEDGEMENT

I would like to express my appreciation to Professor Ed Davidson and especially to Professor Janak Patel for the generous amount of time they spent advising me on this paper. I would also like to thank my wife Julie for her assistance in typing and proof reading this paper. I would especially like to thank my parents, Marion and Elmer F. Pflug Jr. for their support and encouragement throughout my college career.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER 1

Introduction

1.1 Background and Motivation

The power of the microprocessor has been increasing greatly since its conception. Today microprocessors are being used in applications requiring large address spaces to handle larger user memory requirements and protection. Multiple microprocessor systems require large shared memory spaces to be effective.

The current widely used generation of 8 bit microprocessors is inherently limited to an address length of 16 bits and thus an address space of 2^{16} or 64K words. This has become inadequate for the increasingly complex applications for which these microprocessors are being used. Therefore one objective is to increase the addressing capabilities of such microprocessors beyond 64K words.

The large memory requirements of some applications force the use of a two-level memory hierarchy for cost effectiveness. Normally a paging scheme is used in implementing this two-level hierarchy because of the ease with which it is applied to the two-level memory system and because of the structure that it brings to the design.

At the present time there are no inherent capabilities for implementing a virtual memory system on current microprocessors. A second objective is to study the constraints of implementing a virtual memory system for current microprocessors and to suggest ways to make new microprocessors more compatible with virtual memory.

1.2 Overview of Research

In chapter 2 of this paper the objective of extending the addressing capabilities of current microprocessors is explored. Some general statements are made and some real microprocessors are examined. Chapters 3 and 4 deal with the compatibility of microprocessors with virtual memory. Chapter 3 covers the topic of page faults and how they are handled. Chapter 4 investigates the translation of virtual addresses to physical addresses. Chapter 5 ends the paper with some conclusions and comments.

CHAPTER 2

Increasing The Logical Memory Space

2.1 Introduction

The address space of any microprocessor is architecturally limited by the number of bits in that microprocessor's address. The microprocessor's address is a loose term which can be interpreted as the maximum number of address bits that an instruction can specify, the width of an index register or program counter (PC), or the width of the external address bus. On many microprocessors all of these are equal. In this discussion the number of bits in the microprocessor's address will be considered to be the width of the address bus. A microprocessor with m lines in its external address bus, i.e., an m bit microprocessor address, has an address space of 2^m words. For example, a microprocessor with 16 address bits has an address space of 2^{16} words or 64K words (1K words is equal to 1024 words).

The address length of the microprocessor limits the total memory space that can be directly addressed by that microprocessor, without any external help, to one address space. However, with additional hardware the total memory space accessible by that microprocessor can be increased. For example, multiple copies of the entire address space can be created to make the total accessible memory space much larger than one address space. This is done by creating or obtaining information independent of the microprocessor address which can be used to differentiate between multiple address spaces. In many cases the cost of the external hardware necessary to do this will be small compared to the gain in the total logical memory space accessible to the microprocessor.

The design of such hardware is the subject of this chapter. The techniques for increasing the accessible memory space can be classified as:

1. modes: distinguisning between the type of memory access
2. segmentation: explicitly keeping track of multiple address spaces by the use of an external register.

Section 2.2 and 2.3 investigate access modes, while section 2.4 discusses segmentation.

Manufacturers of microprocessors are acknowledging the need for greater addressing capabilities and through the ever increasing power of technology are starting to meet these needs. Sections 2.2, 2.3, and 2.4 then are directed toward the most commonly used 8 bit generation of microprocessors which are limited to 16 address bits. Section 2.5 examines how the new generation microprocessors are internally using the

topics of these previous sections to extend their addressing capabilities. Section 2.6 discusses the need of memory expansion on multiple microprocessor systems.

2.2 Access Modes

One way of extending the logical memory space of a system is to create modes from microprocessor status information, providing that this information can be obtained. A mode is a state of the microprocessor which is independent of the microprocessor address and can be used to identify the type of memory reference. For example, the internal state of a microprocessor has sufficient information to distinguish between an instruction word fetch and a data word fetch. If this information is available externally, then two modes can be obtained, the instruction mode and the data mode. Then it is possible to have two mode areas, each an entire address space of length 2^m words, where m is the length of the microprocessor address. One or the other of these two mode areas is accessed depending on the mode of the microprocessor. This is like adding an extra bit to the microprocessor address, creating a new memory space of length $2^{(m+1)}$ words. Other potentially useful modes are the stack mode, the interrupt service routine mode, and possibly the input/ouput mode.

The stack mode is defined as an access done directly using the stack pointer (SP). This includes such common instructions as PUSH, POP, CALL, and RET. The stack mode is a potentially useful mode.

Including a stack mode allows a stack to have up to 2^m words. This relieves the user from having to worry about the stack running into other data or code. However, there are some drawbacks to the stack mode. For example, in some cases it is convenient to reference data within the stack without using a stack instruction such as PUSH or POP, but rather by doing a normal memory reference using the stack pointer contents with an offset as an operand address. This type of reference to the stack cannot be done with a system using a stack mode unless an instruction such as EXAMINE STACK existed which would reference within the stack and also cause the microprocessor to indicate a stack reference.

The interrupt service routine (ISR) mode is defined as accesses made while running an ISR. This mode would be entered when an interrupt is first acknowledged and would continue until the return at the end of the ISR is executed. The mechanisms for determining these entrance and exit points will be discussed later. The input/output mode would be defined as an input or output data reference. Many microprocessors have this capability (eg. the Intel 8080). The usefulness of this mode is questionable from the point of view of extending the addressing space, and depends on the number of input/output devices on the system. The input/output mode will not be examined in this discussion since it is just an application of the concepts to be discussed and since its usefulness is limited.

Some modes may be nested to create even more modes. For example, the instruction, data, and stack modes could be nested within the ISR mode to create new modes: the ISR instruction mode, the ISR data mode,

and the ISR stack mode. It is important to examine these modes to see if they are useful. It is useful to have an ISR mode area which holds interrupt service routines to free the user from having to worry about where they reside in the user area. However it is not useful, and probably detrimental, to have the data for these ISR's reside in the ISR mode area, since this would necessitate a large amount of communication with the user; for example with interrupt driven input/output. It is equally detrimental to have the data for the ISR's be in an ISR data or stack area, since these too would be unaccessible to the user programs. It is much more useful to have the instructions of the ISR's reside in a private area and have the data for these routines reside in the user's data area. This can be done by creating a normal instruction mode, an ISR instruction mode and a common data mode. The common data mode can further be broken into the normal data mode and the stack mode.

There are then four modes considered to be useful; the normal instruction mode, the ISR instruction mode, the data mode and the stack mode. Reasonable combinations of these modes can be used to create a system with two, three, or four modes. Not all modes can be created with every microprocessor. Also the amount of external hardware required for identifying a particular mode varies a great deal with the different microprocessors. By creating n modes from microprocessor status information, n mode areas, each an address space of size 2^m, may be differentiated. If $n=2^x$, then this is like adding x bits to the microprocessor address, giving a total effective memory space of $2^{(m+x)}$ words. If a memory expansion factor, MEF, is defined as the total expanded memory space using modes divided by the original address

space, then using n modes yields a memory expansion factor of n. Even with only two modes this is an appreciable gain of logical memory space.

In some cases however it may not be advantageous to use modes to differentiate between mode areas which are entire address spaces. Consider a system with a single microprocessor which must assemble and load its own programs. A problem arises in that this microprocessor cannot write in its logical instruction mode area. If it is assumed that the assembler program is in the instruction mode area and the source program is in the data mode area, then the microprocessor can assemble the source program and store the object program in the data mode area. This presents no problems until it is time to run the object program. Now the object program should be in the instruction mode area in order to run. It is however in the data mode area. One solution would be to simply have a way to exchange the physical instruction mode and data mode areas, but this is not a complete solution. During assembly some constants may be generated and these are data and must reside in the data mode area when the object is run. However after the instruction mode and data mode areas are exchanged, they will reside in the instruction mode area and therefore cannot be accessed as operands.

In such a case the mode information can be used to create separate mode areas whose sizes are a subset of one address space. The two mode areas are now of size 2^m-s, where $0<s<2^m$. There then exists a remaining space of size s, which can be shared by all modes. This space is called the common area. Now the assembler can reside in the present instruction mode area, and the source program in the data mode area. As the source is assembled, the object program will be stored in the data

mode area, and all constants can be stored in the common area. When the exchange of instruction mode and data mode is made, the constants are still accessible in the common area.

The method by which the common area is distinguished from the mode areas is to use high order address bit(s) to determine whether or not mode information is to be enabled. For example, assume that the high order address bit A_{m-1} is used to determine whether or not to enable modes. Then three areas are created, each of size 2^{m-1} words. One area is used when $A_{m-1}=1$ and mode = instruction mode. Another area is used when $A_{m-1}=1$ and mode = data mode. These are the mode areas. The remaining area, the common area, is used when $A_{m-1}=0$, no matter which mode is in effect. Figure 2.1 shows the memory map for this particular example, assuming m=16.

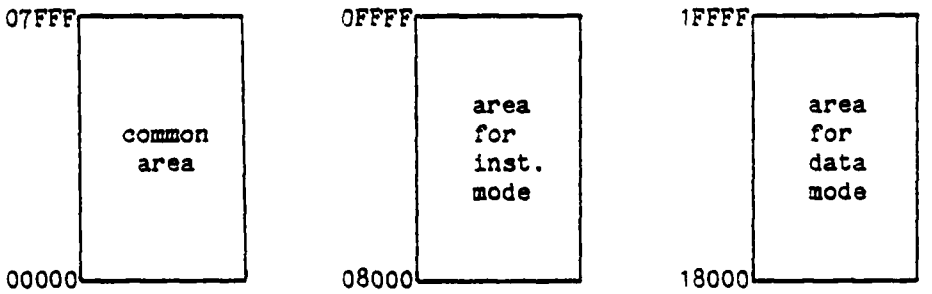


Figure 2.1 Logical memory space of two mode system with m=16 (addresses in hexadecimal).

The size of the common and mode areas can be made larger or smaller by changing the number and the use of the high order address bits. When the common area size is increased, the mode area sizes are decreased and vice versa. Assume that the three high order address bits $A_{m-1}A_{m-2}A_{m-3}$ are used to determine mode usage. In this case many different common area and mode area sizes can be obtained. If the common area is used only when $A_{m-1}A_{m-2}A_{m-3}$=111, then the common area will be of size 2^{m-3} and each of the mode areas will be of size $2^m-2^{m-3}=(7/8)*2^m$. If however the common area is used when $A_{m-1}A_{m-2}A_{m-3}$=111, 110 or 101, then the common area will be of size $3*2^{m-3}$, and the mode areas will each be of size $2^m-(3*2^{m-3})$. Using more or less of the combinations of $A_{m-1}A_{m-2}A_{m-3}$ to enable the modes effects the sizes of the common area and the mode areas.

Figure 2.2 shows a logical implementation of the mode enabling circuitry for a system with four modes, using three bits with the combinations 000, 001, and 010 to designate the common area. When any of these three combinations appear, the added address lines $A_{m+1}A_m$ are forced to 0 and the common area is referenced. When any other combination of bits $A_{m-1}A_{m-2}A_{m-3}$ appear the outputs of the encoder are gated to $A_{m+1}A_m$. Figure 2.3 shows where in the logical memory space the common area and the specific mode areas reside, assuming m=16.

The exchanging of modes can be done by using a small programmable random access memory to map the outputs of the decoder into the added address lines $A_{m+1}A_m$. Figure 2.4 a) shows the logic for a two mode system, and figure 2.4 b) shows the logic for a four mode system. This logic is inserted in the dashed box labeled mapping in figure 2.2. The

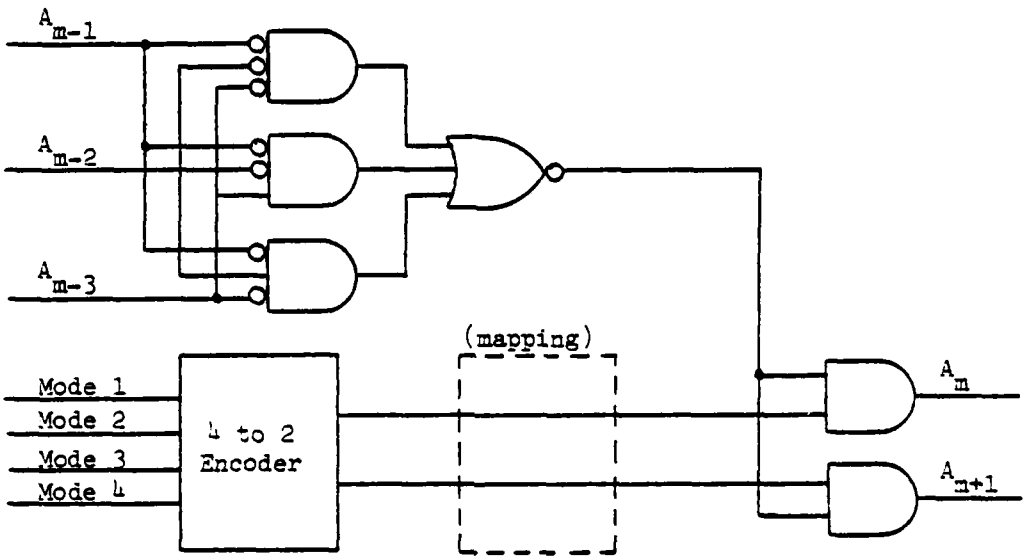


Figure 2.2 Mode enabling circuitry for four modes with combinations 000, 001, 010 signifying common area.

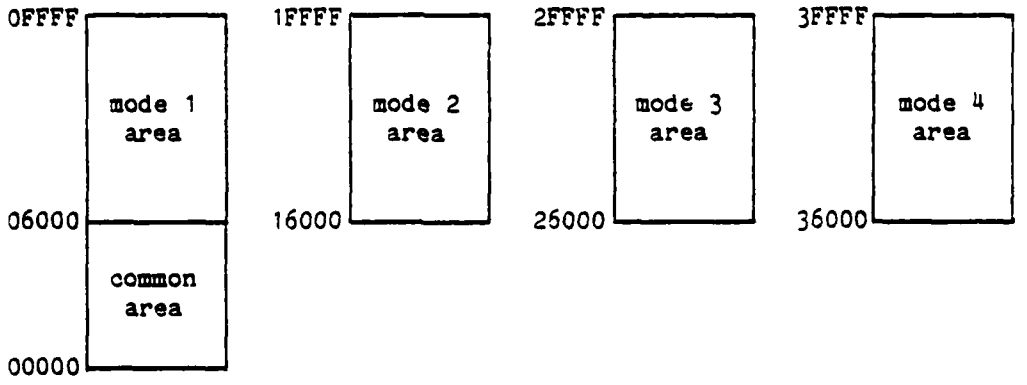


Figure 2.3 Logical memory space of above mode system (address in hexadecimal).

map is written as an output device and the system is responsible for changing the map contents. By changing the map, each mode area can reside in any logical mode frame. This flexibility is necessary on a system which must assemble its own programs and/or load its own memory.

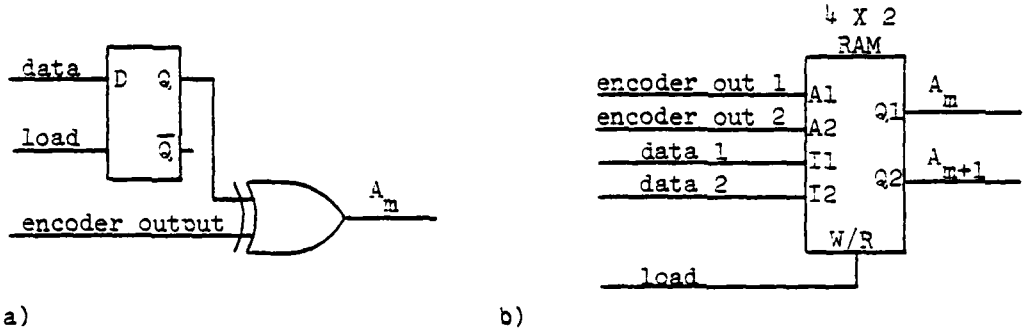


Figure 2.4 Mapping hardware for systems with a) two modes and b) four modes.

It is possible to derive generalized equations for the sizes of the common area and the modes areas. Using these equations a generalized memory expansion factor can be obtained. Let *n* be the number of modes that can be obtained from microprocessor information. Also let b be the number of high order address bits used, with c combinations of these b bits signifying a reference to the common area. Then the common area is of size $c*2^{m-b}$ words, and each mode area is of size $2^m-(c*2^{m-b})$ words. The total memory space accessible under these constraints is $(c*2^{m-b})+(n*(2^m-(c*2^{m-b})))$ words. Simplifying this equation and

dividing by 2^m gives a memory expansion factor of

$$MEF=n-[(n-1)*c*2^{-b}]. \qquad (2.1)$$

In the above example of two modes and one combination of one bit, namely $A_{m-1}=0$, used to reference the common area, the values of the variables are n=2, b=1, and c=1. Substituting these values into equation 2.1 gives an MEF of 1.5. This can easily be seen to be true in figure 2.1. Table 2.1 lists some representative values of n, b, and c, and shows the resulting common area size, mode area sizes, and memory expansion factor.

2.3 Implementing Modes on Microprocessor Systems

The concepts discussed in section 2.2 are applicable to a large set of microprocessors. The specific microprocessors examined however will be limited to the Intel 8080, the Motorola 6800, and the Zilog Z80 microprocessors, all commonly used in industry today. They all have a limited address length of 16 bits, resulting in an address space of 64K bytes. In most systems using these microprocessors, the total logical memory space is equal to the address space. In many applications this may be inadequate and thus there exits a need to increase the logical memory space of these microprocessors.

In the preceding section some useful modes were discussed; the normal instruction mode, the interrupt service routine (ISR) instruction mode, the data mode, and the stack mode. To obtain these modes certain processor information must be present. It must be possible to

TABLE 2.1 Some values of n, b, and c and the corresponding memory area sizes and MEF (m=16).

n modes	b	c	mode area sizes (K words)	common area sizes (K words)	MEF
2	1	1	32	32	1.500
	2	1	48	16	1.750
		3	16	48	1.250
	3	1	56	08	1.875
		3	40	24	1.625
		5	24	40	1.375
		7	08	56	1.125
3	1	1	32	32	2.000
	2	1	48	16	2.500
		3	16	48	1.500
	3	1	56	08	2.750
		3	40	24	2.250
		5	24	40	1.750
		7	08	56	1.250
4	1	1	32	32	2.500
	2	1	48	16	3.250
		3	16	48	1.750
	3	1	56	08	3.625
		3	40	24	2.875
		5	24	40	2.125
		7	08	56	1.375

differentiate between an instruction fetch and a data reference, between a normal data reference and a stack reference, and between a normal program instruction fetch and an instruction fetch while running an ISR. It will not be possible to get all this information out of every microprocessor. Therefore some microprocessors are not capable of supporting four modes, but may support less than four modes.

2.3.1 Modes in an Intel 8080 System.

One microprocessor capable of supporting all four of the previously defined modes is the Intel 8080. Each 8080 memory reference takes one machine cycle which is broken up into three to five states [1]. During the first state of every machine cycle a processor status word is output on the data bus. This status word can be latched and used to determine the proper mode for each reference. Some of the information included in the status word is an interrupt acknowledge signal (INTA), a stack reference signal (STACK), and a signal called M1 which indicates the fetch of the first byte of an instruction. STACK can be used directly as the stack mode signal. The rest of the mode signals however must be generated using external hardware.

To determine when an instruction fetch is being done, M1 must be used. Whenever M1 is present an instruction fetch is being done, but M1 is not present during the fetch of succeeding instruction bytes of a multiple-byte instruction. The opcode of the instruction will be present on the data lines during state T3 of cycle M1. This opcode can

be externally decoded by a 256 X 2 read only memory (ROM) to determine the length of the instruction. A counter can then count the number of memory references after each M1 cycle. Using this counter, M1, and the ROM outputs, an instruction byte fetch can be identified. A data mode signal can now be generated by (not instruction fetch) AND (not STACK).

To determine whether an instruction byte fetch is a normal instruction fetch or an ISR instruction fetch, a way must be found to tell when an ISR is being run. When the first byte of an ISR is accessed, the INTA bit of the status word is one. This can be used to set a flip-flop which holds the ISR information. When the flip-flop output is one an ISR is being run. The flip-flop needs to be reset after the ISR has completed running. This can be done by creating a special location somewhere within the ISR instruction mode area which when referenced resets the flip-flop and returns the opcode for the RET (return) instruction on the data bus. To exit the ISR instruction mode, a JMP is done to this location. The RET causes the program which was running before interrupt to resume. ISR instruction mode can now be generated by (instruction fetch) AND (ISR flip-flop set), while normal instruction mode is indicated by (instruction fetch) AND (ISR flip-flop not set).

All four mode signals are now generated and two extra address bits can be formed as in section 2.2. Again, unless the system's programs and data are down loaded from a host system, there should be hardware to create a common area. This hardware is straight forward as in section 2.2 and will not be detailed in this section.

2.3.2 Modes in a Zilog Z80 System.

The Z80 microprocessor is to a large extent an enhancement of the 8080, and therefore the generation of mode signals with a Z80 system is very similar to that of an 8080 system [2]. However only three modes of the four considered can be obtained in a Z80 system. The Z80 does not output a processor status word. Instead the status is indicated by certain output signals on the microprocessor's pins. The 8080 implementation of four modes required the STACK, INTA, and M1 bits of the processor status word to generate the mode signals. No indication of a stack reference can be obtained from the Z80 and therefore the stack mode cannot be created. However M1 is an output pin of the Z80 and INTA can be generated by ANDing the IORQ (input/output request) and M1 output pins, so ISR instruction mode and normal instruction mode can be generated as in the 8080 case.

The third mode created with the Z80, namely the data mode, contains the stack mode and normal data mode of the 8080 system. The data mode signal is simply generated when an instruction byte is not being fetched.

2.3.3 Modes in a Motorola 6800 System.

The Motorola 6800 microprocessor does not externally supply any processor status information necessary to create modes [3]. Therefore this microprocessor is not readily applicable to the mode method of increasing the logical memory space, for it would require an extreme

amount of external hardware.

2.4 Segmentation

A second way of increasing the logical memory space is to define segments which are copies of the address space or subsets of the address space, just as modes were. However moving from one segment to another is more difficult and must be better defined than moving between mode areas. To differentiate between segments, a segment register exists. This register holds a pointer to a segment frame in logical memory. The number of segments in logical memory is determined during system design.

Segments differ from modes in that the user must be very conscious of segments. In the mode method of section 2.2, the user only had to be sure his program would fit in the instruction mode area, his data would fit in the data mode area, etc. Switching between modes was done transparent to the user. But with segments, the user must be aware of when a segment switch should be made, and in fact he should request the switch. Therefore he must structure his programs accordingly. This is most easily done by breaking up the program into subroutines. Each subroutine can reside in a separate segment or many subroutines can reside in one segment. Ideally each subroutine should reside entirely within one segment to minimize the overhead of switching between segments.

The segment size can be as large as the address space of the processor, but as with modes this may create some problems. One of the problems with segmentation is how to switch execution between segments. What is desired to be done is to first change the segment register, and then jump to a particular entry point in the new segment. However if this is to be done with code in the old segment, then after the segment register is changed, the new segment is already the current segment and the jump instruction in the old segment cannot be executed. This is not a problem if the program counter of the microprocessor after executing the change segment instruction is equal to the desired entry point of the new segment, but requiring this would create extremely awkward segment switching with high overhead. Even if this is tolerated, there is still a problem with parameter passing.

The solution to these problems is to have a segment size which is a fraction of the total address space, and allow more than one segment to be accessible at once. This is typically done by having a base segment and a current segment concurrently accessible. The base segment is always accessible and contains the necessary routines and storage for parameter passing, current segment switching, system management, and possibly the user's global variables. The current segment is the user segment which contains the subroutine(s) and local variables for the subroutine(s). The sizes of the base and current segments can be fixed or dynamic depending on the implementation.

There are two ways to define segments. One way is to require that all segments are disjoint and the other way is to allow overlapping segments. The implementation and concepts of the two are different.

2.4.1 Overlapping Segments

There are two possible reasons for allowing overlapping segments: a) to allow sharing between two adjacent segments in logical memory, and b) to minimize fragmentation in a fixed segment size system. Sharing is not a valid reason because the sharing due to overlapping is only between adjacent segments. Any sharing or switching between nonadjacent segments requires a base-current segment system. This base-current segment system already can share and switch adjacent segments. Therefore sharing between segments is not a good reason for allowing overlapping segments.

Minimizing fragmentation is a marginally acceptable reason for having overlapping segments only in a system which has fixed size segments. If a subroutine requires only one half of a segment and no other subroutines exist which have size less than or equal to one half of a segment, then allowing overlapping segments would permit the next segment to begin right after the one half segment subroutine, thus saving one half of a segment from being wasted. However the price paid for saving this otherwise fragmented memory may be too high to pay.

To implement overlapping segments, the segment register must be able to reference any location in the logical memory space. This requires a long register. Also the contents of the segment register must be added to the processor address for every memory reference. This requires a large adder and may introduce a significant amount of time overhead for each reference. This reduced system performance, and the extra hardware necessary may be too high a price to pay for reducing the logical memory fragmentation.

It is important to note that any segmentation system which is used with a two-level virtual memory system using paging will only have this large fragmentation problem in the virtual space. The largest fragment in physical space is only one page. In most real applications the large virtual space created with segments will be used with a two-level hierarchy memory system with paging. Therefore the fragmentation reason for overlapped segments is not a strong one.

2.4.2 Disjoint Segments

Overlapping segments were justifiable only in a fixed segment size system. However disjoint segments can be either fixed size or variable size. Fixed size segments on a system with only a one-level physical memory can result in a large amount of fragmentation of the logical memory space. Variable size segments can prevent this fragmentation, but are costly to implement.

To implement variable size segments, the system must be capable of storing the starting address of a segment (which could be any location in memory) and the length of a segment in a hardware register. Each memory reference must generate an address by adding the processor address to the segment's starting address as in the overlapping segment case, and also check to see if this processor address is within the segment. These two operations can be done in parallel, but some time overhead still results and a significant amount of hardware is necessary. This extra time and hardware is probably not justified by the saving of otherwise fragmented logical memory.

The most straightforward and easy to implement segment system is one that uses fixed size, nonoverlapping segments. If the size of the segments is a power of two, then no addition is necessary and the contents of the segment register can simply be concatenated with a part of the microprocessor address. Figure 2.5 shows how a system of this type might be implemented. The segment register is addressable as an I/O port. Thus it can be changed under program control.

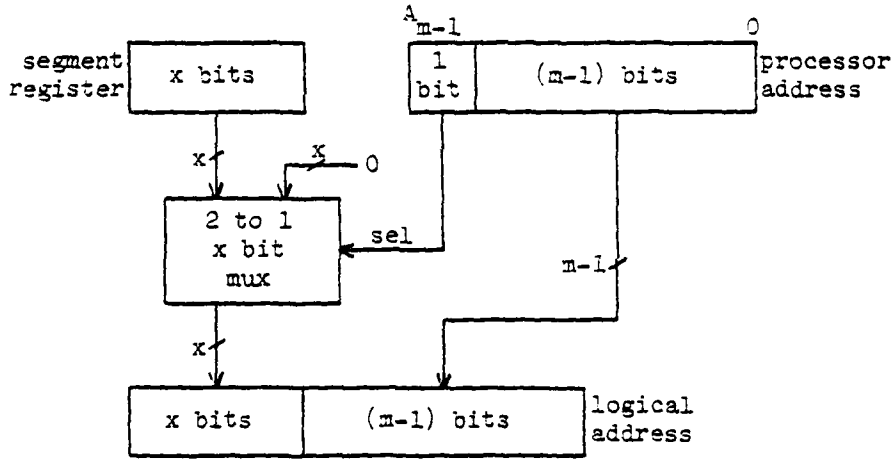


Figure 2.5 Basic segment system with 2^x segments of size 2^{m-1} words each.

The microprocessor address length is m bits and the high order bit A_{m-1} is used to determine whether the base segment or the current segment is being referenced. The segment register is x bits long

permitting 2^x possible segments. Segment 0 is the base segment and the next 2^x-1 segments are user segments. All segments are of size 2^{m-1} words. There is only the small multiplexer delay between the time the processor address is valid and the time the logical address is valid. This will probably not result in any time overhead.

As in the mode method of increasing the logical memory space, the sizes of the base segment and user segments can be different by using more high order address bits and different combinations of the bits to determine whether the base or current segment is referenced. Again as the size of the base segment decreases the user segment sizes increase and vice versa. The memory expansion factor (MEF) as previously defined is for this system

$$MEF=[c*2^{(m-b)}+((2^x-1)*(2^m-(c*2^{(m-b)})))]/2^m.$$

This simplifies to

$$MEF=2^x-[(2^x-2)*c*2^{-b}]-1$$

where b and c are defined as they were in the mode system with b being the number of high order address bits used to determine base or current segment and c being the number of combinations of these bits signifying a reference to the base segment. Table 2.2 gives some segment sizes and MEF's for representative values of b, c, and x.

TABLE 2.2 Some values of x, b, and c and the corresponding segment sizes and MEF (m=16).

x	b	c	user seg. sizes (K words)	base seg. sizes (K wo s)	MEF
2	1	1	32	32	2.00
	2	1	48	16	2.50
		3	16	48	1.50
	3	1	56	08	2.75
		3	40	24	2.25
		5	24	40	1.75
		7	08	56	1.25
3	1	1	32	32	4.00
	2	1	48	16	5.50
		3	16	48	2.50
	3	1	56	08	6.25
		3	40	24	4.75
		5	24	40	3.25
		7	08	56	1.75
4	1	1	32	32	8.00
	2	1	48	16	11.50
		3	16	48	4.50
	3	1	56	08	13.25
		3	40	24	9.75
		5	24	40	6.25
		7	08	56	2.75
5	1	1	32	32	16.00
	2	1	48	16	23.50
		3	16	48	9.50
	3	1	56	08	27.25
		3	40	24	19.75
		5	24	40	12.25
		7	08	56	4.75

2.5 Uses of Modes and Segmentation in New Microprocessors

Currently there are three major manufacturers of advanced microprocessors; the same three who manufacture the "standard" eight bit microprocessors: Zilog, Motorola, and Intel. These advanced microprocessors have at least 16 bit data paths and increased addressing capabilities. The new generation microprocessors are the Zilog Z8000, the Motorola 68000, and the Intel 8086.

Of the three the one most directly utilizing modes and segmentation is the Zilog Z8000 [4]. In the Z8000 references to input/output and memory are references to separate memory spaces. The input/output space is 64K words long, while the memory space is broken up into as many as six mode areas, each an address space of length 64K words on the 40 pin version, and each a segmented memory space of up to 8 Megawords on the 48 pin version (the Z8000 comes in two versions, one for small systems and one for medium to large systems). These address spaces are referenced under the following modes: user instruction, user data, user stack, system instruction, system data, and system stack modes. As was pointed out in section 2.2 for interrupts, the system data and system stack modes are not necessarily useful.

The Z8000 uses segmentation on the 48 pin version. There are seven segment bits in the processor address which means that there are possibly 128 64K word segments. The address space as previously defined is, for the 48 pin version of the Z8000, the addressable space after segmentation since segmentation is done internal to the microprocessor. Therefore each segment is not equal to an address space as previously

defined. However segmentation internal to the Z8000 is implemented as nonoverlapping fixed size segments of length 64K words. All address arithmetic is done in 16 bit quantities so that the same arithmetic unit that is used for data can be used for addresses. The segment register is not affected by the address arithmetic. Therefore the internal segment implementation of the Z8000 is very much like that discussed in section 2.4. No base segment is necessary, however, since each instruction can specify an entire address including the segment number and thus switching segments and passing parameters can be done directly.

The Intel 8086 implements segments slightly different than the Z8000 [5]. Like the Z8000, 16 bit arithmetic is done and therefore segments are 64K words long. However unlike the Z8000, the 8086 uses a 16 bit segment register in address calculations. The most significant 16 bits of a 20 bit address are represented in this segment register. This is equivalent to having a 20 bit segment register with the 4 least significant bits equal to zero. The 8086 thus uses 64K word segments which may overlap, but which begin on boundries that are multiples of 16. Or another way to interpret this is that there are 2^{16} disjoint segments each of size 16 words. Figure 2.6 shows the microprocessor address generation of the 8086.

The 8086 also allows four modes; the current code, current data, current stack, and current extra modes. These modes are used internally to pick one of four segment registers which is then used to generate the processor address.

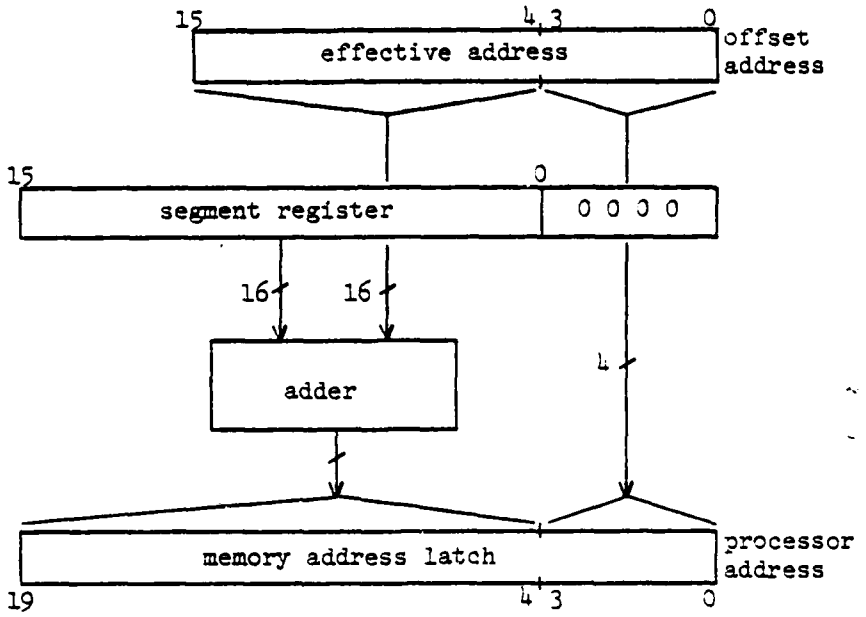


Figure 2.6 Memory address generation of the Intel 8086.

The Motorola 68000, as understood from available information, seems to have 2 modes: a user mode and a supervisory or system mode [6]. The mode is externally indicated so that separate address spaces can be created, or address translation can be disabled in system mode. Variable length segmentation is utilized in an architecture specified memory management scheme. Not much specific literature is available on the 68000, so the details of this segmentation are not known.

2.6 Multiple Microprocessor Systems

When more than one microprocessor is used in a system, each with its own instruction and data streams, there are a couple of ways in which the total logical memory space of the system can be defined. Each microprocessor can address the same memory space. Then two microprocessors with the same address will access the same memory location. In this type of system sharing between microprocessors is trivial, but protection and contention become significant problems. The more microprocessors in the system, the less memory space each microprocessor can privately access.

The other way of defining the total logical memory space is to have each microprocessor's memory space be a separate space. If there are y identical microprocessors in the system, then the total logical memory space of the system becomes y times the total memory space per microprocessor. The separate spaces are accessed by appending a processor number to the microprocessor address, thereby extending the system address length. Protection is enhanced in this type of system. Sharing can be accomplished by mapping the logical processor address of two microprocessors into the same physical address. This also allows more flexible sharing than having the microprocessors access the same logical address because the two logical microprocessor addresses need not be equal to be mapped to the same physical address.

2.7 Concluding Remarks

The use of external hardware is a real solution to the memory addressing problems of the current microprocessors with 16 bit addresses. Segmentation can be used by any microprocessor, but its use is not transparent to the user. Access modes are transparent to the user, but cannot be used by all microprocessors. Table 2.3 shows the relative difficulty of implementing the four modes presented on the three popular microprocessors.

Even with the advances in technology that have been experienced recently, the amount of logic that can be put on a chip is still not limitless. Therefore manufactures are using techniques similar to those presented in sections 2.2 and 2.4 to minimize the internal logic of new microprocessors, especially in respect to busses and address arithmetic.

TABLE 2.3 Relative difficulty of using each of the four modes on the three microprocessors examined.

microprocessor	data mode	inst. mode	stack mode	ISR mode
Intel 8080	moderate	moderate	trivial	moderate
Zilog Z80	moderate	moderate	extreme	moderate
Motorola 6800	extreme	extreme	extreme	extreme

CHAPTER 3

Handling Page Faults On Microprocessor Systems

3.1 Introduction

Virtual memory is a well known and proven technique of implementing a large logical memory space with a good speed, cost per word compromise [7,8]. Implementation on a single large processor system has been well studied and is fairly straightforward. However, microprocessor systems present some significant problems for virtual memory implementation. This chapter investigates the problem of handling page faults on microprocessor systems, and goes on to discuss ways to handle interrupts on microprocessors designed to be used in virtual memory systems.

Ideally it is desired to immediately interrupt the microprocessor on the occurrence of a page fault. There are two levels on which interrupts may be handled [15]. The first and most common level is the

machine level interrupt. In this level interrupt requests are tested for only before the machine instruction fetch cycle. This method guarantees that an interrupt can occur only after the machine instruction is completed and before a new instruction starts.

The second level of handling interrupts is the control (or microprogram) level. In this level interrupts are tested for as often as once every microprocessor clock cycle. This method allows interrupt requests to be handled quickly before the CPU state changes, but requires that restrictions be placed on the control and interrupt mechanisms. These restrictions are due to the extra CPU information which must be saved, and the timing of the microprocessor.

The current generation of microprocessors can only be interrupted on the first level; the machine level. This is a severe inadequacy when the microprocessor is to be used in a virtual memory system.

Section 3.2 investigates the handling of page faults on current microprocessor systems. 3.2.1 looks at a method which does not use the interrupt capability of the microprocessor at all, and 3.2.2 looks at the constraints placed on a virtual memory system by a current microprocessor with only machine level interrupts. The methods suggested are very restrictive and certainly a microprocessor designed for use in a virtual memory system could do better. Section 3.3 explores the internal requirements of the control level interrupt capability necessary on a microprocessor suited to supporting virtual memory.

3.2 Handling Page Faults with Current Microprocessors

3.2.1 Without Interrupts - Waiting

One method of handling a page fault on microprocessor systems is to simply have the microprocessor go into a wait state until the needed page is brought into memory [9]. The concept is to, on a page fault, have the page checking hardware force the microprocessor into a wait state and inform the system of the fault. After the page fault is serviced, the microprocessor is allowed to exit its wait state and continue as if nothing happened. Therefore to the microprocessor a page fault is nothing more than a very slow memory reference. This technique can only be used when two important requirements are met: 1) the microprocessor is capable of waiting an indefinite amount of time before the memory reference is completed, and 2) the system has the capability to handle the page fault independent of the main microprocessor.

Only some microprocessors are capable of waiting indefinitely for the completion of a memory reference. This wait is normally initiated by an external processor signal which is controlled by the referenced memory. Two microprocessors with the wait state capability are the Intel 8080 and the Zilog Z80 [1],[2]. The external processor signal on the 8080 is the READY line while on the Z80 it is called the WAIT line. Pulling this line low before a specific time in state T2 of a machine cycle will cause the microprocessor to go into a wait state. Returning the line to high after a particular time in the wait state allows the microprocessor to complete the machine cycle.

One possible problem with waiting for a page fault recovery on these microprocessors is that the decision of whether or not to wait must be made in a relatively short amount of time. On both the 8080 and the Z80 this decision time is about one clock period from the time the address bus is stable. This may be as short as 320 ns which is not sufficient to check maps for page availability unless high speed logic is used. One way to extend the decision time is to automatically insert one or more wait states after the T1 state of each machine cycle, which would add extra clock periods to the decision time. This however would also degrade the performance of the system and should be avoided if possible.

There are a couple of reasons for not handling page faults on a microprocessor system by going into a wait state. One reason is that many microprocessors are not capable of waiting indefinitely for the completion of a memory reference. One such microprocessor is the Motorola 6800, which cannot go into a wait or idle state in the middle of an instruction [3]. The memory response time can be extended slightly by stretching the clock signals, but only to a maximum of 4 us before internal processor information is in danger of being lost. This is clearly too short a time period for a page swap to be made. Thus the Motorola 6800 and all other microprocessors not capable of going into an indefinite wait state in the middle of a memory reference are not capable of handling page faults by waiting.

A second reason for not handling a page fault by waiting is the second requirement that the system be capable of doing the page fault servicing independent of the main microprocessor. To service a page

fault some processing is required to determine what page to replace, whether the replaced page should be written back to secondary memory, where the desired page resides in secondary memory, and also to update the virtual memory maps. The system must be capable of processing a page fault independent of the main processor. This means that there must either be a host system to do page swapping and map updating, or there must be a separate processing unit dedicated to virtual memory page fault routines and map management. In a single processor microcomputer system it may not be desirable to add a separate microprocessor dedicated to handling system management.

The reason the main processor is not capable of processing the page fault is that it must be in the wait state the entire time the page fault is being processed. It cannot execute any instructions until the current instruction is completed, but the current instruction cannot be completed until the page fault is resolved. So processing time is wasted while the microprocessor is in this wait state.

3.2.2 Handling Page Faults with Machine Level Interrupts

A second way of handling page faults on current microprocessor systems is to have the page fault generate a machine level interrupt. Since the instruction causing the page fault must complete before the interrupt request is acknowledged, its result is computed with invalid data and it must be re-executed. It will be shown that this completion with false data and re-execution of the instruction is costly.

In order to re-execute the instruction, certain information about the state of the machine must be saved. The address of the first word of the instruction and the page number of the missing page must be saved. The quickest and easiest way of saving this information is to use hardware registers. The instruction address is stored during the fetch of the first word of every instruction until a page fault occurs. Then the storing of this register is disabled and the register contains the desired restarting point. After the page fault is processed the restart adress is obtained and the loading of this register is again enabled. The page number is similarly stored in a hardware register which is loaded with the proper address bits when a page fault occurs.

When a page fault occurs the completion of the current instruction must be correctable. This means that the completion and re-execution of this instruction cannot affect its result. This requirement results in the following restrictions [10].

First, multiple word instructions should not cross page boundaries. The first word of an instruction contains the opcode, while the following words normally contain either immediate data or an operand address. Faulty immediate data can cause the alteration of an internal register in a way which makes recovery impossible. A faulty operand address could result in the modification of memory and this cannot be allowed.

A second restriction is that the data returned by the faulty memory reference must be forced to a value which allows the largest number of instructions to be re-executed. Any instructions which cannot be correctly re-executed must be prohibited. The prohibited instructions

can probably be converted by macros in the assembler or compiler into a sequence of allowable instructions and thus this prohibiting of instructions can be kept transparent to the user. With microprocessor instruction sets, normally only a small number of instructions will have to be prohibited. Anderson and Lipovski show that if the data lines of an Intel 8080 based system are forced to zero when a page fault occurs, only three instructions need be prohibited [10].

The third restriction is that whenever the microprocessor is running a program which resides in virtual space, interrupts must be enabled. This is an obvious restriction since if it is not possible to inform the microprocessor of the page fault, the microprocessor will continue executing using false data, and it will generate false results.

A final restriction is that a normal stack reference cannot be allowed to cause a page fault interrupt. Most microprocessors react to an interrupt by PUSHing at least the program counter (PC) onto the stack. Following a stack reference page fault this would also cause a page fault resulting in an endless loop. Anderson and Lipovski suggest solutions to this problem [10]. The preferred solution for virtual memory systems implemented with current microprocessors is to have the current and next pages of the stack resident in main memory at all times. This does dedicate two pages of the main memory to the stack, but this is much more attractive than limiting the stack to one page.

From the above discussion it is apparent that with current microprocessors it is not very easy to handle page fault interrupts. This makes implementing any sort of virtual memory system with these

microprocessors awkward. Depending on the system and the microprocessor used, it may be more cost effective to increase the main memory size of the system rather than implement virtual memory.

3.3 Interrupts on Microprocessors Compatible with Virtual Memory

3.3.1 Concepts

One of the more important requirements of any computing system is that the results of a program should not be dependent on the system configuration or state at the time the program is executed. Thus in order for a microprocessor to be compatible with virtual memory it must be able to guarantee that pseudo random occurrences of page faults do not result in invalid output. A page fault signals the system and the microprocessor that the memory location referenced is not presently accessible and if any dummy data is returned in response to this reference then it is not valid. Completing the execution of the instruction using this dummy data can result in a change of microprocessor state which cannot be recovered from. This of course will cause invalid program results and must be prohibited. It is therefore necessary to have a page fault result in the current instruction being canceled (suspended) and re-executed (completed) after the page fault has been serviced.

The most logical way to have the page fault affect the microprocessor is through the interrupt mechanism. For the purpose of this discussion an interrupt request will be the act of a device external to the control unit of a microprocessor signalling its desire to have normal program execution altered, while an interrupt will be the act of the control unit acknowledging the interrupt request and altering the program execution. There are two types of interrupt requests: internal and external [11]. Internal requests are due to arithmetic results (overflow, divide by zero, etc.), control conditions (illegal opcode, stack overflow), or memory conditions (parity error, protection error, page fault). External interrupts are generated by I/O devices or other devices such as another microprocessor in a multiple microprocessor system or an operator console. The page fault interrupt will be treated as a special case, because these other types of interrputs can be adequately handled with present interrupt strategies.

Most small and medium scale computers today do not allow control level interrupts. As was seen in the previous section, implementing virtual memory on these systems is difficult and restrictive. To understand why the choice of interrupting only between instructions is made, the concepts of CPU state and instruction cycle must first be defined. CPU state is an important concept because in order to continue execution of an interrupted program after servicing the interrupt request, it is necessary to restore the state of the processor to what it was when the interrupt occurred. To do this the state of the microprocessor at the time of this interrupt must obviously be saved. The instruction cycle is important to define because a big issue with

interrupts is when they occur and the instruction cycle has been used as the determining factor.

The term state is very loosely used with a variety of different meanings. However the state of a CPU is normally defined by

1. the contents of all the registers within the CPU
2. the state of all control signals in the CPU hardware [12]

and will be used as such in this discussion. There are basically two types of elements in CPU hardware. One is combinational logic and the other is the memory element. The CPU state is made up of all of these memory elements each of which falls into one of the above two categories. The state of the microprocessor is therefore a large amount of information and saving it is not a trivial matter.

The instruction cycle of a microprocessor is composed of the actions required to fetch and execute the instruction. These actions vary with different instructions. The composition of an instruction cycle is typically not well defined. However there are certain divisions which can be made in instruction cycles. One such division is between the instruction fetch and the execution. Every instruction contains these two divisions. However these are fairly broad divisions and the instruction cycle can be futher broken down into: [13]

1. fetch instruction
2. increment PC
3. decode instruction
4. fetch operands
5. execute

Numbers 1 and 2 above constitute the previous instruction fetch division and numbers 3, 4, and 5 constitute the execution division. Zero address instructions do not require any operand fetch (memory reference) while some instructions may fetch several operands.

There are of course some exceptions to this general instruction structure. There are some newer microprocessors which include instructions that do string operations. These instructions have multiple operand fetch, execute cycles resulting in an instruction cycle such as

1. fetch instruction
2. increment PC
3. decode instruction
4. fetch operands
5. execute

.
.
.

n-1. fetch operands

n. execute

This type of instruction will be ignored since small systems are being addressed here and these string operations are generally found on larger systems. Instructions in many microprocessor instruction sets are more than one word in length. In these cases there are one or more fetch instruction word, increment PC cycles inserted after the instruction decode. Some microprocessors may also have auto increment/decrement features. An instruction with one of these features would have an increment/decrement register before or after the fetch operand cycle.

All instruction cycles do have the fetch instruction cycle. This is the first thing done in an instruction. It consists of outputting the PC onto the address bus and after a period of time inputting the data into the instruction register. This is the one thing that has made interrupting only between instructions such an attractive method. Because of this common first cycle, all instructions start out with almost the same control state. Therefore by interrupting only between instructions, very little control state information has to be saved. The return to a program after an interrupt routine can be, and often is, handled in exactly the same way as a return from a subroutine call.

This is not only the easiest way of handling interrupts, but it is also all that is really necessary for interrupts as they are commonly used. Harold S. Stone puts it like this:

> "The purpose of the interrupt system is to provide for useful computation in place of wait loops (especially while waiting for the completion of an I/O operation)." [14]

Page fault interrupts obviously do not fall into this limited purpose and therefore any system compatible with virtual memory needs expanded interrupt capabilities.

There are two basic issues to be resolved on page fault interrupts. The first is <u>when</u> should the interrupt be handled and the second is <u>how</u> should the interrupt be handled. These two issues are fairly independent. The issue of when the page fault interrupt should occur has one preferred solution over the total range of possible instruction sets, while the issue of how the page fault interrupt is to be handled

has two basic solutions, with the preferred one being dependent on the instruction set complexity and the implementation of the CPU's control unit.

To answer the question of when the interrupt should be handled, it is first necessary to further explore the composition of an instruction cycle. The CPU basically consists of a functional unit to do operations, some registers to store operands and results, buses on which data is transferred, and a control unit to determine what data is transferred where and when, and how this data is operated on. The instruction cycle is then made up of CPU clock cycles, each defining one or more concurrent register transfers and operations.

A memory reference consists of a register transfer from some register holding the desired address (eg. the PC) to a buffer register commonly called the memory address register (MAR). After a predetermined amount of time (typically one or two CPU clock cycles) the reference is completed by loading the contents of the data bus into a register. If a page fault occurs during this memory reference this data is invalid and it is therefore desirable to have a page fault interrupt prohibit the transfer of the data bus contents to the selected register. Therefore the interrupt must be allowed to occur before this transfer for every memory reference which is done. Typically this could be done by allowing the interrupt to occur at a particular time during every clock cycle, or possibly only during the last clock cycle of each memory reference.

Allowing interrupts at these points also puts a constraint on the virtual memory hardware. If the interrupt is to be allowed before any register gets loaded with data due to a memory reference, the interrupt request must be generated prior to this time. This constraint is easily met since the information which causes a page fault is normally stored and retrieved from the same map that is used to determine the physical address which references the data.

As stated, there are two different methods of handling the page fault interrupt when it does occur. The big difference between the two is the place from where execution is continued after the page fault has been processed. One method allows execution to resume at the beginning of the memory reference causing the page fault interrupt, while the other requires that the entire instruction be re-executed. The preferred solution depends on the complexity of the CPU.

3.3.2 Re-executing the Entire Instruction

The method of re-executing the entire instruction is attractive because it does not require the preservation of all the control information. Preserving the control state is especially difficult when the CPU control unit is hardwired since the logic is fairly random and many of the memory elements of the control are often hard to access cleanly. What must be preserved for the re-execution method is all CPU state information, except the control state which is repeatable at the beginning of each instruction, which can change during the course of any

instruction of the instruction set prior to the last memory reference data transfer of that instruction. This typically includes all of the user accessible registers; all condition registers; virtually all information which must be protected while calling and running a subroutine.

This information must be preserved such that the re-execution of the current instruction due to a page fault will not alter its results. Generally this may require a temporary duplicate set of all of these registers, which are loaded with the contents of their corresponding user accessible registers at the start of each instruction. However only those registers that are changed in a way which would cause false results upon re-execution must be saved in temporary duplicate registers. For example, any register which is only irrevocably changed after the last memory reference of any instruction need not be saved since a page fault cannot occur between the time the register is changed and the time the instruction ends. Also any register which is only changed by being loaded from memory need not be saved since re-execution would simply reload it.

It also may not be necessary to save all registers whose contents can be irrevocably changed by any instruction. Assume that there are m registers whose contents are modified in one or more instructions of the instruction set. Now suppose that at most, n of these m registers are modified during any single instruction. Then only n additional registers are needed to preserve the state of the microprocessor while executing an instruction. Some obvious registers which must be stored for every instruction include the PC (program counter) and the SP (stack

pointer).

One drawback to this method is that it is limited in the complexity of the instruction set of the CPU with which it is implemented. This is because certain more powerful instructions not only alter the CPU state but also the main memory in a way in which re-execution would cause false results. Consider an instuction which could have a page fault occur after loading a register from memory and writing this same memory location. Proper re-execution is impossible by this method since after the register is restored to what it was prior to this instruction, the correct contents of the memory location is lost. This is typical of string operations. Thus the instruction set of a CPU with such an interrupt handling method is restricted. This restriction may not be serious for some microprocessors however.

Consider the Intel 8080 microprocessor [1]. Its instruction cycle is broken up into machine cycles which may or may not include a memory reference (only one instruction in the 8080 instruction set has machine cycles which do not include memory references). Each machine cycle is broken up into states (Ti's). The memory reference of a machine cycle is initiated during T1 and the data transfered to a register during T3. The preferred time to allow interruption is during state T2 of every memory referencing machine cycle. Then by saving the PC and SP at the beginning of every instruction, all instructions of the instruction set could validly be re-executed. Thus it is seen that for less powerful instruction sets, the re-execution of the entire instruction is an attractive method of handling these page fault interrupts.

3.3.3 Continuing Execution with the Interrupted Memory Reference

The second method of handling interrupts is to resume execution of the interrupted instruction at the last memory reference before the interrupt. This allows a more powerful instruction set, but also requires the saving of all CPU control state information at the beginning of each memory reference. This requires temporary control registers to save the state of the control memory elements. The main advantage of this method is that any instruction can be treated in this manner, allowing arbitrarily powerful and complex instruction sets.

The implementation of this method may or may not be realistic, depending on the complexity of the control unit. In a CPU with a straightforwardly implemented microprogrammed control unit, all that may be necessary is to additionally store the microprogram counter. In more complex microprogrammed control units it may be necessary to save a microprogram scratchpad register set and/or stack. On microprogrammed control units with subroutines, it may be possible to just PUSH the current microprogram counter onto a return stack. The implementation varies largely with the control unit realization.

Due to the complexity and overhead of interrupting in the middle of an instruction, it may be advantageous to have two distinct interrupt types. One would interrupt only between instructions, while the other could interrupt in the middle of instructions. In this way interrupt requests whose sources do not affect the result of the current instruction could be handled between instructions, thereby avoiding some costly overhead.

3.3.4 Stack Issues

One problem which must be resolved under either of these methods is where to save the CPU state information which must be preserved. In a system with only one stack, the information could be PUSHed onto the stack. Then however stack references could not be allowed to cause page faults. This is for the same reasons as were discussed in section 3.3. On systems with more than one register allowed as the stack pointer, or even a memory location allowed as the stack pointer, the solution of section 3.3 of keeping the current and next closest stack pages in main memory would be unrealistically complex and time consuming. In these cases, and generally, it may be preferrable to reserve a resident part of main memory for storing CPU state registers during a page fault interrupt. Some resident memory is already necessary to store the page fault routine since servicing a page fault must not cause another page fault. Since this must be a fixed size portion of memory only a few concurrent page faults (from different tasks) should be allowed.

3.4 Concluding Remarks

In this chapter it has been shown that implementing a virtual memory system on current microprocessors is extremely difficult and costly. It can be done for some 8 bit microprocessors such as the Intel 8080, but it is very restrictive. The more powerful instruction sets of the newer microprocessors are going to make it much harder and

more costly to implement. Therefore new microprocessors should be designed with extended interrupt capabilities in order that they may better support virtual memory systems.

The large addressing capabilities of new microprocessors dictate that systems of the future will use virtual memory. This will require control level interrupt capabilities. Designing a microprocessor with control level interrupts has been shown to be a fairly reasonable task. With the wide spread and almost complete use of microprogrammed control units, this task is made even easier.

CHAPTER 4

Address Mapping On Microprocessor Systems

4.1 Introduction

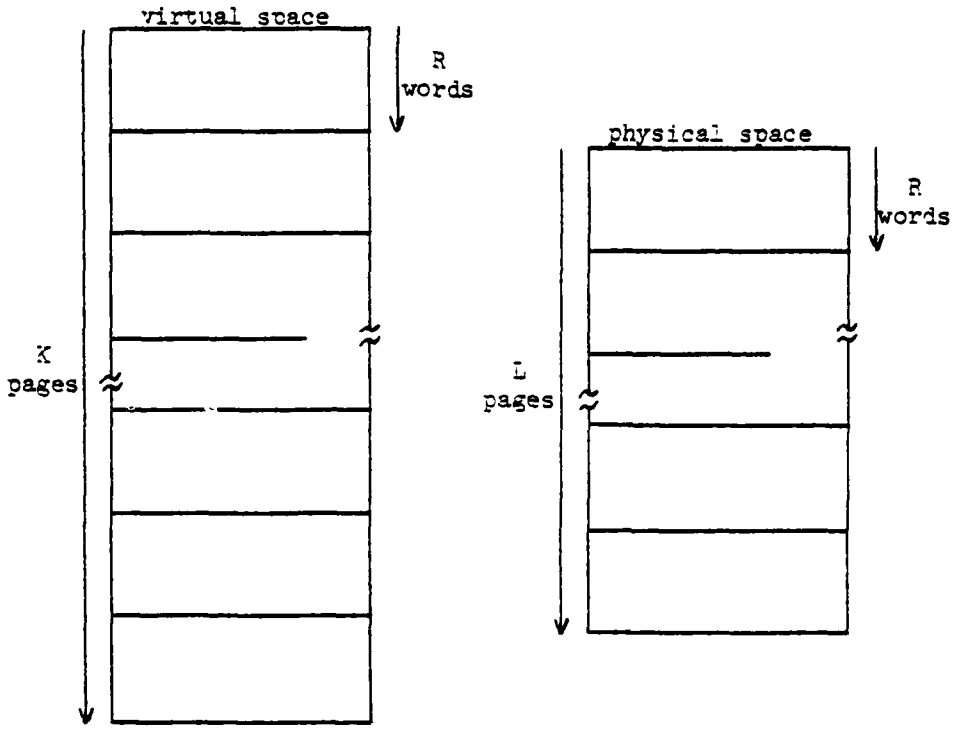
One of the key components of any virtual memory system is the translation of the processor generated virtual address into the physical address sent to the main memory. The hardware which accomplishes this is called the address mapping hardware due to the fact that this translation is a mapping from virtual space into physical space. This address mapping has been thoroughly studied with respect to large microprocessor systems [7],[8]. However the cost and implementation of microprocessor systems affect some of the design tradeoffs of address mapping. Section 4.2 will present the basic principle of address mapping.

Section 4.3 will develop the basis for desiring to have this mapping done by a single chip called the memory management unit (MMU). Section 4.4 will develop the MMU concept and try to investigate what features should be included in this chip. Section 4.5 will discuss some issues of mapping in a multiple microprocessor system.

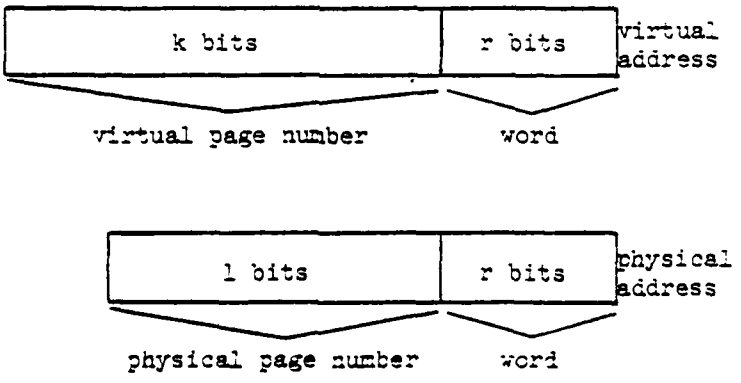
4.2 The Basic Principles of Address Mapping

The virtual and physical memory spaces of a virtual memory system are normally divided into fixed size blocks, called pages. Assume that each virtual and physical page is of length 2^r words, that the virtual space is composed of $J=2^j$ pages, and that the physical space is composed of $K=2^k$ pages. For virtual memory to be effective $J>>K$ must be true. Each page in physical memory potentially holds a virtual page and is thus called a page frame. Figure 4.1 a) shows the virtual and physical memory spaces and figure 4.1 b) shows the corresponding addresses. Some of the obvious design considerations involved with paging are page size, number of virtual pages, and number of physical pages. These are application dependent and will not be discussed in this paper. Matick [7] and Denning [8] explore some representative cases.

The purpose of address mapping is to provide the translation of the virtual address of the microprocessor to the physical address needed by the main memory, and also to determine if the referenced page resides in main memory. The basic scheme for this is shown in figure 4.2. The map can either be a lookup table called an indexed page table, with J



a)



b)

Figure 4.1 The memory spaces (a) and addresses (b) of a virtual memory system.

entries, one for each virtual page, or it can be an associative (content addressable) table with K entries, one for each physical page frame. The entries in the indexed table would contain the physical page number, a presence bit, and some other control information while the associative table entries would contain a virtual page number, a physical page number, and some other control bits.

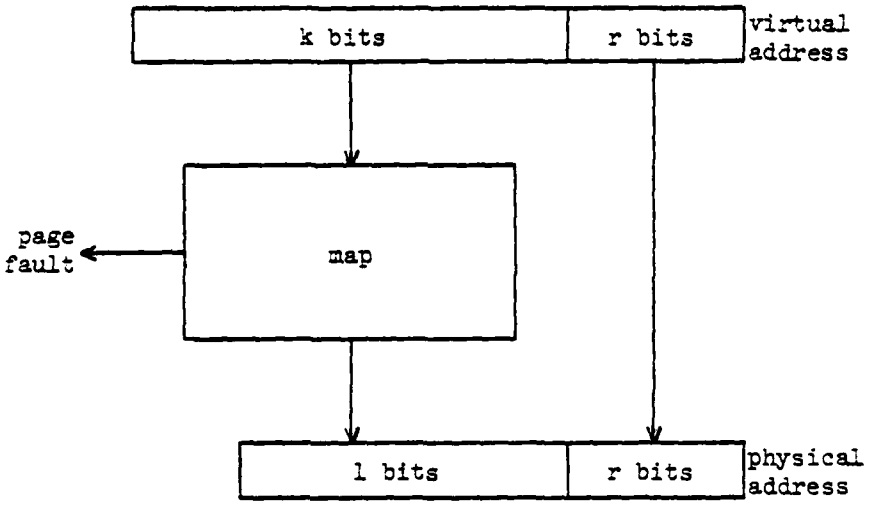


Figure 4.2 Address translation of a virtual memory system.

The presence of a virtual page in physical memory is simply determined by testing the presence bit in the indexed table implementation of the

map, while it is determined by an associative match of the virtual page number in the associative table.

Each of these two methods has its drawbacks. The indexed table method requires a large amount of memory to store all the entries. It can be stored in fast, dedicated memory, but this is very expensive. It can be stored in the main memory, but this requires an extra main memory cycle for each microprocessor memory reference which results in extreme performance degradation. The associative table requires less storage for its entries, but it requires an associative compare of the virtual page number portion of each entry. This can be extremely expensive with a moderate number of physical pages. Thus both methods by themselves are not adequate.

The typical solution is to use both an indexed page table and a partial associative table, or CAM (content addressable memory). The indexed page table is stored in main memory and the CAM is a small, fast memory with typically 8 or 16 entries. The CAM is of course dedicated hardware.

A memory reference first initiates a test of the CAM. If the virtual page number is in the CAM then the translation occurs quickly, and the reference is completed in one memory cycle. If the virtual page number is not in the CAM, then the indexed page table is referenced to see if the virtual page is in fact in physical memory. If it is, then this virtual page entry of the page table replaces the least recently used (LRU) entry of the CAM and the reference is completed in two memory cycles. By keeping the most recently used page table entries in the CAM, most references are completed in one memory cycle due to the

locality of references [7]. If the virtual page is not in main memory, i.e., if both these tests fail, then a page fault is generated.

This scheme of having both an indexed page table and a partial associative table is a very effective compromise between the two methods. It is based on the same principle as virtual memory, which is the phenomenon of locality of reference. It is also a very general scheme in that any virtual page may reside in any physical page frame.

4.3 The Basis for a Single Chip MMU

The implementation of virtual memory on a microprocessor system must be done under some serious constraints. The microprocessor is a device which is relatively inexpensive, even with its increasing computing power. Thus the designer of a system built around a microprocessor must especially be aware of the cost of this system. It does not make sense to use a microprocessor as the central processing unit of a system with with extensive support hardware.

The microprocessor has been kept a relatively inexpensive component through the use of circuit integration, i.e., putting the entire logic of the microprocessor onto a single chip. The cost of developing such a chip must be absorbed by mass producing it to keep the individual chip costs down. Thus the design of a single chip should be general enough to cover a wide range of applications. This integration concept can be applied to the address mapping of a virtual memory system.

It is therefore desirable to include all of the address mapping hardware on a single chip. This chip will be called the MMU (memory management unit). This integration and the required generality can impose some constraints on the mapping design. It also gives powerful possibilities since a large amount of logic can be inexpensively put on a single chip.

4.4 Features of an MMU

4.4.1 The Page Table Register

Normally a system utilizing virtual memory is a fairly powerful system. Usually multiple tasks or users are allowed on such systems to keep the CPU utilization high. Each task should be given its own page table to enhance the flexibility of the system and the relocatability of shared code such as system routines. When a task is running its page table is used. When this task is switched to another task, the page table used to generate physical memory addresses is also switched.

The switching of tasks can occur under several cases. One case is when the first task finishes and a second task is then begun. Another case is when a page fault occurs. The normal course of action on a page fault is to have the microprocessor do some system management including determining which page to swap out of main memory and changing maps, initiating a page transfer, and then continuing to process another task

whose working set of pages resides in main memory. An important factor in system performance is task switching, which includes page table switching.

Two methods for implementing page tables exist. One method uses a fixed page table area. The page table is then switched by writing out the current table and then writing in the new one. This is very time consuming and degrades the task switching efficiency. The second way is to make page tables relocatable. A pointer is used to point to the currently used page table. The current page table entries are then accessed by adding the virtual page number to the page table pointer and using the result to reference the page table. Changing page tables becomes as easy as changing the page table pointer. This pointer can be stored in a register called the page table register (PTR).

Typically there are a predetermined number of tasks whose working sets can be resident in main memory. The pages of main memory are then divided between these tasks with a few extra pages remaining uncommitted for the handling of page faults.

4.4.2 The Associative Array

As discussed in section 4.3, the most desirable implementation of address mapping is to have both the indexed page table and the associative table to achieve a compromise between mapping speed and cost. The indexed page tables reside in main memory while the associative table resides in the MMU. Two important issues are the

number of entries or the size of the partial associative table (which determines the number of associative compares done) and the contents of the table.

The size of the partial associative table, or CAM, is really a compromise between the time and complexity of doing the associative compares and the hit ratio of the CAM. The hit ratio of the CAM is the number of address translations done through the CAM divided by the total number of address translations (done through both the CAM and the indexed table). On most large computers this tradeoff has resulted in CAM's of 8 to 32 entries. However some study should be done to determine a good CAM size for microprocessor systems. This size will probably be dependent on the system and its application, but some general size should be chosen for the MMU to keep it flexible.

The entries of the CAM should include a virtual page number which is associatively compared, a corresponding physical page number and some control bits. The physical page number and control bits for an entry are output when the virtual page number of that entry matches the microprocessor generated virtual page number. Some control bits which are necessary are a clean/dirty bit used to determine whether a page to be replaced in main memory needs to be written out to secondary memory, protection bits which are used if the page table contains protection information, and some bits to keep track of the least recently used entry, so that this entry can be replaced when a referenced virtual page number is not in the CAM.

4.4.3 Control Logic

The MMU must be capable of performing some dedicated functions. It must be capable of accepting a virtual address and doing the mapping function. Thus it must be able to first initiate the associative compare. If this fails it must be able to use the PTR to generate a reference to the indexed page table in main memory. If this is successful, the MMU must generate a second reference to main memory to complete the requested reference and also update the array. If the indexed page table reference returns a false presence bit the MMU must send a page fault signal to the microprocessor.

The PTR and CAM of the MMU must be readable and writeable by the microprocessor to allow the microprocessor to do system management. Also when the system is servicing the page fault in a supervisory mode, the MMU must be able to inhibit the address mapping and pass addresses directly to main memory.

4.4.4 Other Features Which May Be Included in the MMU

The previous three sections discussed some of the required features of the MMU. This is by no means all of the possible features which may be included in the MMU. Some other features which may fit nicely into the function of the MMU include memory protection checking and segmentation. Since memory protection information is often kept in page tables, the MMU, which uses page table information to generate addresses, would be a natural place to include the memory protection

checking hardware. In segmentation, microprocessor addresses are used with a segment register to generate a full virtual address. This address is then sent to the address mapping hardware which in this case is the MMU. It would be convenient to include both the segmentation and mapping hardware in the MMU. Then the MMU would be the only hardware involved in taking the microprocessor address and generating the physical address.

4.5 Address Mapping on Multiple Microprocessor Systems

There are some critical design tradeoffs in implementing the address translation hardware of a multiple microprocessor system with a shared virtual memory. One important decision is whether to give each microprocessor its own map, or whether to have a global map. This section does not attempt to solve this problem, but rather to discuss how the MMU can be used to enhance either method. Obviously if one central map were used, the MMU would be a natural choice since it could maintain the map and do the translation function. A problem of contention could arise, but it would be no more serious than the contention problems which already exist for the global table and the memory itself.

If individual maps were used, each microprocessor would require its own page table to reside in the virtual memory. The address translation for each microprocessor could be done separately, but there must be some global system unit which would oversee the individual translations.

This is due to the sharing among microprocessors which is desired to enhance system efficiency. The MMU could be such a unit. In such a case the MMU would not have to have any address translation duties. It would rather be called upon to handle page faults to keep all of the individual maps of each microprocessor valid. This function could be done independently and in parallel with the normal address translation of each microprocessor, increasing the efficiency of the system.

The role of the MMU in a multiple microprocessor system may change some from its role in a single microprocessor system. However it still would utilize the powerful concept of integration and cost reduction, thus enhancing the implementation of virtual memory on microprocessor systems.

4.6 Concluding Remarks

This chapter was in no way intended to be a complete treatment of a single chip MMU. It was however intended to generate some thought on the possiblilities of creating a single chip which could do the required functions of address translation. Hopefully it has been shown that this MMU could be an attractive way of bringing virtual memory capabilities to microprocessor systems of the future.

CHAPTER 5

Conclusions

5.1 Summary

In some applications, current microprocessors are limited by the size of the address space which they can access. This is due to the number of address bits which they use. This limitation can be overcome to a certain extent by adding external hardware to create more address bits. One method of doing this is to use processor state information to create memory access modes such as instruction mode or data mode. The switching of these modes can easily be made transparent to the user. A second method is to form segments which are independent of the microprocessor state. Arbitrarily large numbers of segments can be created, but segment switching is not transparent to the user.

Large logical memory spaces are typically not implemented entirely in main memory. It is much more cost effective to use at least a two-level virtual memory scheme. Microprocessors do create some problems with such an implementation.

One problem is that a page fault must interrupt program execution in a way such that correct results are insured. The limited interrupt capabilities of microprocessors make program interruption and continuation difficult and restrictive.

The second problem which arises is created by the cost constraints of microprocessor systems. One reason for using a microprocessor (and extending its addressing capabilities) is that it is a relatively inexpensive component. The system which utilizes a microprocessor is also under this cost constraint. It is necessary to make the implementation of virtual memory more cost effective so that it can be used in microprocessor systems.

5.2 Suggestions

The limited address space problem of the microprocessor is being avoided in new generation microprocessors through the use of greater address lengths in both instructions and in the address bus. However the virtual memory implementation of these large memory spaces is still difficult and expensive. Microprocessor designers should now investigate ways of making future microprocessors more compatible with virtual memory.

Two ways of doing this were discussed in this paper. The first way is to enhance the interrupt capabilities of future microprocessors, thereby making page fault handling easier. The second way is to create support chips dedicated to virtual memory implementation, thus making this implementation more cost effective.

Meeting the needs of future microprocessor users will require research and development on the part of the microprocessor manufacturers. However the technology of today certainly makes a two-level virtual memory implementation a realistic goal of future microprocessor systems.

REFERENCES

1. Intel Corporation, Intel 8080 Microcomputer Systems Users Manual, September 1975, pp. 2-1 - 2-20.
2. Adam Osborne, An Introduction to Microprocessors. Volume II. Some Real Products. Adam Osborne and Associates, 1976, pp. 5-1 - 5-65.
3. Motorola Corporation, M6800 Microcomputer Systems Design Manual. Motorola Corporation, 1975.
4. Bernard L. Pueto, "Architecture of a New Microprocessor," Computer, pp. 10-21, February 1979.
5. S. Morse, W. Pohlman, B. Ravenel, "The Intel 8086 Microprocessor: A 16-Bit Evolution of the 8080," Computer, pp. 18-27, June 1978.
6. E. Stritter, T. Gunter, "A Microprocessor Architecture for a Changing World: The Motorola 68000," Computer, pp. 43-51, February 1979.
7. R. E. Matick, Computer Storage Systems and Technology. New York: Wiley and Sons, 1977, pp. 532-644.
8. P. J. Denning, "Virtual Memory," Computing Surveys, vol 2, no 3, p. 153, September 1970.
9. M. D. Ruggiero, S. G. Zaky, "A Microprocessor-Based Virtual Memory System".

10. J. A. Anderson, G. J. Lipovski, "A Virtual Memory for Microprocessors," Proc. Second Annual Symp. on Comp. Arch., pp. 80-84, January 1975.

11. D. J. Kuck, The Structure of Computers and Computations. New York: Wiley and Sons, 1978, p.348.

12. A. G. Lippiatt, The Architecture of Small Computer Systems. London: Prentice-Hall International, Inc. 1978, p.98.

13. J. L. Peterson, Computer Organization and Assembly Language Programming. New York: Academic Press, 1978, p.55.

14. H. S. Stone, Introduction to Computer Organizations and Data Structures. New York: McGraw-Hill, 1972, p.174.

15. Advanced Micro Devices, Build a Microcomputer, Chapter VI. Advanced Micro Devices, 1979, pp. 1-29.