A Simulator for the Intel 8086 microprocessor

William A. Chapman

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritsscholarworks@rit.edu.
A SIMULATOR FOR THE INTEL 8086 MICROPROCESSOR

by

William A. Chapman

A Thesis Submitted
in
Partial Fulfillment
of the
Requirements for the Degree of
Master of Science
in
Electrical Engineering

Approved by:

Professor Ken Hsu
(Thesis Advisor)

Professor James R. Schueckler

Professor

Professor

(Department Head)

DEPARTMENT OF ELECTRICAL ENGINEERING
COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
MAY, 1988
Abstract

This project was originally suggested by J. Schueckler as an aid to teaching students the Intel 8086 Assembly Language. The need for such a tool becomes apparent when one considers the expense of providing students with dedicated hardware that rapidly becomes obsolete, but a Simulator which could be easily updated and runs on a general purpose or timesharing computer system would be accessible to many students for a fraction of the cost. The intended use of the Simulator therefore dictated that it precisely model the hardware, be available on a multiuser system and run as efficiently as possible. An additional design goal was to develop the Simulator in a structured, high level language so far as was practical, to make it easier for others to understand, maintain and modify when necessary. It was therefore in this framework that this Intel 8086 Simulator was developed in Pascal to run on a Digital Equipment Corporation VAXcluster.

The following are trademarks of Digital Equipment Corporation:
DEC DEC/MMS VAX VAXcluster VMS VAX/VMS VAX 8600

The following are trademarks of Intel Corporation:
DEBUG-88 iAPX86/88 Intel iSDM86 MCS-86
## TABLE OF CONTENTS

### Chapter 1  Introduction

1.1 Preface ........................................... 1
1.2 Overview ......................................... 1
1.3 Literature Review ................................. 5
1.4 Summary ........................................... 8

### Chapter 2  Simulator Architecture

2.1 Preface ........................................... 9
2.2 Overview ......................................... 9
2.3 Central Processing Unit Model .................... 15
2.4 Memory Model .................................... 20
2.5 Instruction Formats ............................... 24
2.6 Summary .......................................... 27

### Chapter 3  Input/Output Architecture

3.1 Preface ........................................... 28
3.2 Overview ......................................... 29
3.3 Map Description ................................... 31
3.4 Map Generation And Viewing .................... 36
3.5 Data Description .................................. 39
3.6 Data Generation And Review ..................... 40
3.7 User Communications .............................. 46
3.8 Summary .......................................... 48

### Chapter 4  Debugger Architecture

4.1 Preface ........................................... 49
4.2 Overview ......................................... 49
4.3 Command Types .................................... 53
4.3.1 Utility Commands ............................... 54
4.3.2 Execution Commands ............................ 55
4.3.3 Examine/Modify Commands .................... 59
4.4 Limitations ....................................... 64
4.5 Summary .......................................... 67

### Chapter 5  Maintenance and Support Features

5.1 Preface ........................................... 68
5.2 Opcode Look Up Table Maintenance ............... 69
5.3 Keyword Look Up Table Maintenance .............. 70
5.4 Command Procedures .............................. 71
5.5 Expansion And Modification Capabilities ....... 72
5.6 Language Extensions And Deviations ............. 75
5.7 Summary .......................................... 79
<table>
<thead>
<tr>
<th>Chapter 6</th>
<th>Summary and Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Preface</td>
</tr>
<tr>
<td>6.2</td>
<td>Observations On The Simulator</td>
</tr>
<tr>
<td>6.3</td>
<td>Observations On The Debugger</td>
</tr>
<tr>
<td>6.4</td>
<td>Summary</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 7</th>
<th>Enhancements</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Preface</td>
</tr>
<tr>
<td>7.2</td>
<td>Simulator Enhancements</td>
</tr>
<tr>
<td>7.3</td>
<td>Debugger Enhancements</td>
</tr>
<tr>
<td>7.4</td>
<td>Support Program Enhancements</td>
</tr>
<tr>
<td>7.5</td>
<td>Library Routines And Operating Systems Enhancements</td>
</tr>
<tr>
<td>7.6</td>
<td>Summary</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appendix A</th>
<th>Sample Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>DAC Program Listing</td>
</tr>
<tr>
<td>8.2</td>
<td>DAC Debug Session</td>
</tr>
<tr>
<td>8.3</td>
<td>DAC IOMap86 Session</td>
</tr>
<tr>
<td>8.4</td>
<td>DAC IOData86 Session</td>
</tr>
<tr>
<td>8.5</td>
<td>DAC I/O Map And Data</td>
</tr>
<tr>
<td>8.6</td>
<td>ADC Program Listing</td>
</tr>
<tr>
<td>8.7</td>
<td>ADC Debug Session</td>
</tr>
<tr>
<td>8.8</td>
<td>ADC IOMap86 Session</td>
</tr>
<tr>
<td>8.9</td>
<td>ADC IOData86 Session</td>
</tr>
<tr>
<td>8.10</td>
<td>ADC I/O Map And Data</td>
</tr>
</tbody>
</table>

| Appendix B | Ordered Opcodes for the Intel 8086 Processor |

| Appendix C | Opcode Look Up Table |

| Appendix D | Data Dictionary |

| Appendix E | Keyword Look Up Table |
## Appendix F  User's Guide for the Intel 8086 Simulator

13.1 General .......................................................... 158  
13.2 Input And Output ............................................. 159  
13.3 Debugger ....................................................... 159  
13.4 Attachment I - An Alphabetical Listing Of  
  Debugger Keywords ........................................... 163  
13.5 Attachment II - Debugger Keywords Listed By  
  Class ............................................................ 165  
13.6 Support Policy .................................................. 167  
13.7 Maintenance Capabilities .............................. 168  
13.8 Release Notes For The Intel 8086 Simulator ... 170  

## Appendix G  Unsupported DEBUG-88 Commands ............ 171  

## Bibliography ..................................................... 172
Chapter 1
Introduction

1.1 Preface

This Simulator was designed primarily for the academic environment as an aid to teaching students the Intel 8086 Assembly Language. It was therefore of primary concern that the Simulator model the operation of the Intel 8086 microcomputer hardware with as few compromises as possible. The Simulator's host machine was chosen to be a Digital Equipment Corporation VAXcluster which is a multinode system that can be made available to many students simultaneously. Although the architectures of the Intel 8086 and the Digital Equipment Corporation VAX computers are different, it was possible to create an operational model of the Intel 8086 processor through the use of a variety of data structures and extensions to DEC VAX Pascal.

1.2 Overview

Some compromises were necessary however, especially regarding the instructions that dealt directly with the hardware (e.g. input and output, interrupts, bus lock and escape) since this is where the two architectures differ most significantly. Two types of input and output operations were designed for the Simulator. User communications for interacting with the program
are supported for such operations as displaying text or prompts and entering values to be assigned to input variables, while other hardware devices are simulated by mapping the I/O operations to a data file. The multiuser environment of the Simulator precluded the direct support of other hardware operations such as interrupts and direct memory access (DMA) operations. Other hardware related instructions which could not be supported include those which manipulate the bus control signals for a multiple processor configuration and those which communicate with an attached coprocessor such as the Intel 8087 Numeric Processor Extension. These unsupported instructions require actual hardware to be connected to the system which is not practical for the target environment.

In an academic environment, students who are learning to use a new computer frequently make a plethora of errors which is characteristic of the learning process. The Simulator must tolerate these mistakes or the learning process will be inhibited. In order to be useful here, the user interface must be friendly and forgiving of as many blunders as practical. The Debugger, which is the user interface for the Simulator, was therefore designed to be as tolerant as possible of incorrect input and to provide as much recovery capability as practical.
The Debugger provides the user with a variety of ways to access memory (i.e. using either numeric values or register contents for segment and displacement parameters) and a variety of ways to interpret the information found there (e.g. boolean, byte, word, pointer etc.). The Debugger also provides an expanded number of breakpoints (up to sixteen) for the user to investigate the program's operation and a variety of methods to control program execution (e.g. single step, resume, program step over subroutines etc.). Also included is a method to regain control (i.e. Control D) over a malfunctioning program without aborting the Simulator. These features are intended to assist users in debugging their programs and gain a good understanding of how the Intel microcomputer functions.

The Debugger was also designed to encourage the user to become familiar with what actually goes on in a microcomputer as a program is executed. As a result of this assumption, the Debugger provides a minimum set of commands for the user and does not support symbolic debugging, or make any assumptions about the initial register contents or flag states.

It was desirable to implement the Simulator on a computing system which provided simultaneous access to a large number of students. This implied that the Simulator, all of the support programs and look up tables had to be sharable, but the programs
under test, the I/O maps and the I/O data files had to reside in the students' accounts. This would then permit one copy of the Simulator to reside on the system and be shared by many students at the same time. The host machine chosen to support the Simulator was the Digital Equipment Corporation VAXcluster since this provided access by the largest audience of users at this facility. Other support programs such as the cross assembler, cross linker and cross locator were also available on the system, as well as a variety of DEC supported utilities to assist in the program development and debug process.

It was desirable to implement the Simulator in a modern high level language so that good structured programming techniques could be used in the development process. It was also necessary to provide good documentation for the Simulator so that any changes or enhancements could be readily incorporated by others in the future. It was recognized that Pascal possessed the necessary qualifications and so it was chosen as the implementation language. However certain Digital Equipment Corporation extensions to Pascal were expedient to model the microprocessor's operation and these are discussed in Chapter 5, MAINTENANCE AND SUPPORT FEATURES, Language Extensions and Deviations. The interactive communications between the user and the program under test also required further deviations from the extensions of VAX Pascal and it was necessary to rely directly upon the VMS I/O capabilities. The choice of Pascal as the
implementation language should minimize the future maintenance efforts.

1.3 Literature Review

Many sources of information were reviewed to acquire the necessary detailed knowledge to develop this Simulator. Several user's guides, reference manuals and user's manuals published by Intel provided insight into the architecture of the Intel 8086 Processor as did several texts from independent sources. In addition, several references on Pascal also helped formulate the necessary data structures to support the Intel 8086 Architecture. No references were found however, on a generalized approach for the development of software to simulate the operation of non-native computer opcodes.

The examination of Intel documentation provided considerable insight into the organization of the Intel 8086 processor and its opcodes. This analysis initially focused on how to interpret the MOD and R/M fields of the opcode, and influenced the design of the Effective Address Type record, the RMDecoder module and the ComputeEA (Compute Effective Address) software modules. The organization of the opcodes affected the design of the Opcode Look Up Table, the OpcodeLUTEntry record and the Opcode Type Record. Intel, in their literature, had already assigned opcodes to various classes and this provided a useful
way to segregate the execution of the instructions. It was not sufficient however, to develop the look up table because the opcodes were not all resolved in the first byte. This led to the development and inclusion of an extended class to fully resolve the opcodes.

The User's Manual Programmer's Reference proved to be the major reference for the specific details of implementation for each opcode. However, other sources were also used to clarify or simplify the implementation of each opcode. The Motorola Programmers Handbook for the Motorola 6800 processor provided a more concise, easier to implement method for setting condition codes as a result of performing arithmetic operations. The Rochester Public Library, Rundel Branch, Science and Technology Division also provided precise definitions of mathematical terms which were used to properly identify the variables used in some operations.

Several Intel manuals were consulted to determine the structure and content of the object files created by the cross product assembler and compiler. Since these files are the source of the information on which the Simulator operates, a detailed review of the file formats was essential. This review provided the necessary algorithm for computing the file checksum which is used to verify proper loading of the located object module.
This investigation also revealed the structure of the header records, physically enumerated data records and physically iterated data records which contain the necessary information to load into memory for simulation.

The Intel manual for DEBUG-88 was reviewed and formed the basis for the design of the Debugger. The various command format notations were modeled and implemented as described in Chapter 4 Debugger Architecture. The utility commands, execution commands and examine/modify commands are all patterned after their DEBUG-88 counterparts.

The DEC literature on Pascal and the supported extensions was referenced to select the appropriate data type for manipulating data in the Simulator. The unsigned data type and the associated operators UINT, INT, UAND, UNOT, UOR, UXOR and HEX proved most useful when manipulating the internal bits of information. The VAX/VMS literature was also perused for the information necessary to support the queued I/O operations and the asynchronous trap for the Control D function. Several reference manuals, a course lecture guide and workbook were also needed to successfully implement these features.
While an extensive literature search did not prove fruitful, a thorough review of Intel and DEC documentation did provide the detailed knowledge to develop an efficient simulator for the Intel 8086 processor.

1.4 Summary

The Simulator and auxiliary programs provide a cost effective method for users to test and debug Intel 8086 assembly language programs. Most of the Intel instructions (including I/O instructions) are supported. Bidirectional communications with the user's terminal are provided and a simple, user friendly Debugger controls the program's execution and provides interaction with the program variables. The Simulator and auxiliary programs have been developed in Pascal, a modern high level language, to permit structured programming techniques, self documenting code and ease of future modifications.
Chapter 2
Simulator Architecture

2.1 Preface

This Simulator was designed to operate solely on the output of the cross products assembler, linker and locator. After creating and successfully assembling the source code, the user must link the object modules together using the cross products linker and then position the executable code within the memory map of the Simulator using the cross products locator. The Simulator currently has 8192 (8K) bytes of memory which start at hexadecimal '0000'. A larger memory map can be provided using the maintenance capabilities defined in Chapter 5, MAINTENANCE AND SUPPORT FEATURES, Expansion and Modification Capabilities. Programs that are intended to be executed by the Simulator must have been successfully assembled, linked and located by the appropriate cross products and confine their memory references to the above locations defined in the Simulator's memory map.

2.2 Overview

After a program has been created, assembled and linked without errors, the locate operation produces a file referred to as an MCS-86 Absolute Object File. This file consists of Physically Enumerated Data Records which contain the information to be "loaded" into memory for the Simulator. These records contain
the data and define the physical addresses where those data blocks will reside in memory. A checksum is also included to verify that the data are loaded correctly. The MCS-86 Absolute Object File may also contain a Module End Record that defines the starting address of the main module which is loaded into the Code Segment Register and the Instruction Pointer. The Data Segment, Extra Segment and Stack Segment registers and the Stack Pointer are not initialized, nor are any of the general registers or status flags.

If these steps have been properly executed, the located object file will contain the exact bit patterns as they would appear after being loaded into the memory of an Intel computer. This file can also be read by the Simulator into an array in a Pascal program which models the memory of an Intel machine. The information in this array can then be interpreted as opcodes, operands or data and manipulated by the Pascal program as needed to simulate the operation of an Intel 8086 microprocessor. If the user attempts to "load" a file which has not been located, the record types are incorrect, the information is not stored in memory and an error message is returned to the user.

Intel provides a variety of system services to its users through previously defined system utilities. These include such operations as allocating and deallocating memory buffers,
communications with disk and tape drives, file management services and a host of other functions normally expected in a sophisticated user environment. Many of these operations vary from system to system and all are dependent upon the actual hardware characteristics and their addresses to execute properly. The DEC VMS environment in which the Simulator operates makes the support for these system services much more complicated. It was determined that support for these system services was not essential for an introductory teaching tool and therefore it was considered outside the scope of this project to provide support for these system calls.

In order for the Simulator to be an acceptable teaching tool, it was necessary to provide some support for input and output functions but the host computer and its environment placed constraints on the extent to which hardware could be simulated. The most practical compromise was to provide a fixed number of simulated I/O ports where the supporting software would read previously defined data from an array or store data "generated" by the program in an array. The array would be loaded from a disk file when the simulation session began and would be saved in a disk file when the session ended. Ancillary programs were also developed to create and maintain the input data files and review the output data files. It was not practical to attempt any support for interrupt driven I/O or direct memory access (DMA) operations. These restrictions do not detract from the
usefulness of the Simulator as an introductory teaching tool however, since interrupts and DMA operations are advanced topics that are closely allied with hardware. Likewise, support for the Intel 8089 I/O Coprocessor was not provided. The design of the Simulator does not preclude support for any advanced I/O devices, but modeling asynchronous interrupts and their impact on bus signals internal to the Intel 8086 CPU was considered beyond the scope of this project. This may be an enhancement that could be considered at some future time.

It was also necessary to provide support for the input and output operations to the user's terminal and this is described in Chapter 3, INPUT/OUTPUT ARCHITECTURE, User Communications.

There are three instructions in the Intel 8086 instruction set that are tied very closely to the hardware and cannot be fully simulated. These are the WAIT, LOCK and ESCAPE instructions. The WAIT instruction causes the processor to do nothing until an interrupt occurs. Since interrupts are not supported by the Simulator, this instruction cannot be properly emulated. If this opcode is encountered, a message is sent to the user's terminal indicating that the "WAIT instruction (is) not supported", and execution of the user's program will continue with the next opcode as though an interrupt had been received immediately. The second instruction which cannot be fully
supported is the LOCK opcode. The LOCK instruction is used to assert the bus lock signal in the Intel 8088 maximum mode configuration. Since the hardware signals are not directly simulated, this instruction has no effect on the execution of the Simulator. If this opcode is encountered, a message will be sent to the user's terminal indicating that the "Bus lock instruction (is) not supported", and the execution will continue with the next opcode. The third instruction that is not fully supported is the ESCAPE instruction. This opcode is used to communicate with other coprocessors that may be associated with the Intel 8086. This instruction will access the memory operand as required but there is no bus on which to place that operand. This instruction is therefore partially supported so no message is issued to the user's terminal, and execution will continue with the next opcode.

The other group of instructions whose implementation deviates from a strict interpretation of the Intel 8086 processor are those which cause software interrupts for system services. These software interrupts include the breakpoint interrupt, the divide by zero interrupt, the single step interrupt, and the nonmaskable interrupt. The response to the breakpoint interrupt is built into the Debugger and may not be modified by the user. All of the other software interrupts will cause the appropriate information to be placed on the stack and the interrupt to be processed correctly if the user has defined a vector in the
interrupt pointer table. Since catastrophic things may happen if the user has not defined the required interrupt vector, a message is sent to the user's terminal indicating that "Interrupt Type xxx" has occurred before resuming execution at the location specified by the vector in the interrupt pointer table. The user may then choose to ignore this additional piece of information, or it could be an invaluable debugging message if the interrupt was unexpected (i.e. a divide by zero interrupt) and no vector had been defined. All of the other instructions perform the appropriate operations as described by the available Intel documentation.

Support for other Intel microcomputer components such as the Intel 8089 Numeric Processor Extension and the Intel 80130 Operating System Firmware are not provided. They were likewise considered beyond the scope of this project, but nothing in the design of the Simulator precludes supporting them at some future time.

The assumptions and restrictions indicated above are not believed to be serious impediments to the intentions of this Simulator. It is the conviction of the author that these assumptions and restrictions will focus the student's attention on the task of learning and understanding the operation of the Intel 8086 Microcomputer.
2.3 Central Processing Unit Model

The Central Processing Unit (CPU) Model is composed of two pieces, the Register Model and the Arithmetic Logic Unit Model. The Register Model consists of three "Register Files" and the software modules needed to access and update the register files. The register files include the General Register File, the Segment Register File and the Control Register File which contain the contents of the Intel 8086 working registers. The Arithmetic Logic Unit Model consists of software developed by the author to manipulate the contents of the register files, memory array and I/O ports in a manner consistent with the operation of the Intel 8086 microprocessor. This section will discuss the implementation of the Register Model and the Arithmetic Logic Unit Model.

The Intel 8086 CPU supports four general purpose registers, two pointer registers and two index registers in the General Register File. The registers are all sixteen bits wide, but the upper and lower eight bits of each of the four general purpose registers can also be accessed independently. The General Register File is modeled by a collection of eight uniquely identified variables where each variable is thirty two bits wide, but only the lower sixteen bits of each are used. The software modules which access the register files imitate the operation of the Intel CPU so that the upper and lower eight
bits of the four general purpose registers can be accessed independently. The method of accessing the General Register File consists of forming a unique identifier from a width qualifier and a three bit register, pointer or index designator. The width qualifier specifies whether an eight or sixteen bit register is to be accessed and the designator then uniquely selects the appropriate register. This approach was chosen because it permitted a direct implementation from the information contained in the opcode format. The width and designator parameters are used by the software modules FETCHREGPTR and STOREREGPTR to access or update the contents of the General Register File.

The Intel CPU supports four segment registers in the Segment Register File, each of which is sixteen bits wide and are used in calculating physical addresses. The Segment Register File is modeled by a collection of four uniquely identified variables where each variable is thirty two bits wide but only the lower sixteen bits of each are used. The method of accessing the Segment Register File consists of forming a unique identifier from a fictitious width qualifier and a two bit register designator. The Segment Registers are then accessed and updated by using the General Register File routines but with the fictitious width qualifier selecting only the segment registers and the two bit designator mapped onto the three bit designator for the FETCHREGPTR and STOREREGPTR modules. Thus all accesses
to the Segment Register File are made through the same software modules that are used to access the General Register File.

The Intel 8086 CPU supports a Status Flags word and an Instruction Pointer in the Control Register File. The Status Flags word contains nine processor status flags which reflect the results of various operations in the arithmetic logic unit. The flags are all boolean storage devices, having a value of either true or false, and can be tested or updated independently or treated as a collection called the Flags Register. The model of the Control Register Files organizes the status flags as nine separate boolean variables which can each be accessed or updated without effecting the others. For those operations where the flags must be treated as a sixteen bit register, dedicated software modules either assemble the nine boolean variables and map them into a sixteen bit register or disassemble a sixteen bit register and map them into the appropriate nine boolean variables, permitting proper execution of those operations.

The Instruction Pointer is used by the CPU to determine the address of the next memory resident datum needed by the arithmetic logic unit, which may represent an opcode, an address, an address offset, or datum for manipulation. Once the information is retrieved, the Instruction Pointer is incremented to point to the next datum. Some of the decisions made in the
course of executing an instruction however, may further change the value of the Instruction Pointer. Since there are many modules and procedures in the Simulator which can effect the Instruction Pointer, it is implemented as a global variable accessible by all of the software modules.

The arithmetic logic unit is the "brain" of any computer system and the Simulator models the arithmetic logic unit's implementation of each Intel 8086 opcode with software on the host computer. The opcodes are grouped into six classes: Data Transfer, Arithmetic, Logic, String Manipulation, Control Transfer and Processor Control and a software module has been developed for each of the six classes. Each module contains routines for the opcodes in the specified class and manipulates the flags, registers, memory, I/O ports or Instruction Pointer as needed to precisely reflect the operation of the Intel 8086 Central Processing Unit.

In a "conventional microprocessor" the arithmetic logic unit alternates its operations between fetching information and executing opcodes. The Intel 8086 however is an "advanced microprocessor" which speeds up this process by pipelining or queuing data from memory in parallel with the execution of the current opcode. The primary purpose of pipelining is to speed up the execution of opcodes by anticipating the next datum to be
used and having it waiting in the internal instruction queue without causing the processor to wait for it to be fetched from memory. The Simulator does not model the pipelining done in the arithmetic logic unit for several reasons. First of all, as a teaching tool, the Simulator's primary purpose is to have the correct net effect on the register files, memory array and I/O ports and pipelining is not necessary to achieve this goal. Secondly, pipelining is done primarily to increase the speed of execution in the target environment, but speed of execution is not of primary concern for the Simulator. In fact since the Simulator is executing on a different processor in a high level language, each instruction could take several thousand times longer to execute than it would on a native machine. Thirdly, there is no difference in the results produced by the arithmetic logic unit if it had to wait for the next datum to be fetched from memory. For these reasons, pipelining has been omitted from the model of the arithmetic logic unit.

In an actual Central Processing Unit, Control Circuitry is also necessary to complete the processor system. Control Circuitry provides the necessary clocking and sequencing operations to insure that everything progresses smoothly within the Central Processing Unit. However, none of control circuit functions are visible to the end user unless they malfunction. The purpose of the Simulator is to have the same net effect on the register files, memory array, and I/O ports that would result from an
actual Intel 8086 processor executing the same program. Therefore it was not deemed necessary to simulate the control circuitry functions, and no attempt was intended or made to model them.

The Simulator has modeled the internal Intel 8086 registers as variables and has support routines which provide access to the registers in a manner consistent with the operation of the Central Processing Unit. The arithmetic logic unit has been simulated by software modules to effect an accurate manipulation of the flags, registers, memory, I/O ports and Instruction Pointer which is consistent with the Intel 8086 specifications. However, pipelining and control circuitry functions of the Intel 8086 processor have not been modeled since these hardware operations are not immediately visible to the user.

2.4 Memory Model

The Intel 8086 CPU is capable of addressing 1,048,576 bytes (one megabyte) of memory for programs and data. In order to address this large of a memory space, a twenty bit address is required, but each of the registers which point to memory is only sixteen bits wide. The Intel 8086 CPU forms physical memory addresses by multiplying the contents of the desired segment register by sixteen to form the segment address and then adding the offset
or logical address to form a twenty bit physical address, which points to the desired memory location. The Simulator model has been constructed to follow this process in creating addresses to reference memory and no access can be made to memory without going through a module which computes the appropriate physical address.

The Intel 8086 CPU requires two memory references to access a word which is not aligned on a word boundary. Since instructions can vary in length from one to six bytes, inefficiencies in fetching instructions from memory would occur if this were done serially. The pipeline architecture alleviates this situation by fetching data from memory as words and placing them in the internal instruction queue. The arithmetic logic unit then removes a datum from the queue a byte or word at a time, as necessary, to complete the execution of each instruction, thus reducing the amount of time the arithmetic logic unit has to wait for the next datum and improving the efficiency of the processor. The Simulator however, does not support the internal instruction queue and has therefore taken a different approach to the problem. The Simulator accesses memory a single byte at a time and if a word access is requested, a higher level software module makes two sequential one byte requests, and assembles them into a word for the calling module. In this manner, greater control is exercised over memory reference operations with no loss of
generality. The only negative aspect of this process is that "speed" of execution is reduced but the execution speed is not a design criteria to be modeled by the Simulator.

The Intel memory space is simulated by a one dimensional array with four - eight bit bytes of memory per address in the array, thus providing the most efficient use of the thirty two bit word size in the VAX host computer. Support routines were developed to select the requested byte from the array and provide the proper boundary alignments for the calling module. As indicated previously, word accesses of sixteen bits are accomplished by assembling two adjacent bytes in a higher level software module. This approach of packing four bytes per array address provides the most economical use of the VAX host memory and a minimum image (page) size for execution.

The size of the memory array for the Simulator is defined by two parameters, "Low Memory Limit" and "High Memory Limit". The "Low Memory Limit" is defined as having a value of zero to provide address space for interrupt vectors, so the size of memory is proportional to the value of the "High Memory Limit" parameter. The number of bytes of memory in the Simulator model is therefore four times the "High Memory Limit" since there are four bytes of memory per address in the array. All of the memory addressed by a program must reside within this address.
space,* including code, data and stack requirements. Any attempt to load a program which is not within these defined boundaries of the Simulator's memory will result in an error message and cause the loader to abort. Likewise any attempt to access the stack or data outside these memory limits will result in an error message during program execution.

A one dimensional array is used to model memory for the Intel 8086 Simulator, where each entry in the array contains four-eight bit bytes. The physical address is assembled from a segment address and an offset, and is used as the index into the array to access the desired information. Two sequential eight bit byte accesses are used to assemble sixteen bit words. The size of memory is defined before the Simulator is compiled and all programs, data and stack references must be confined to this space. Attempts to access memory outside the bounds defined in the Simulator will result in an error message for the user, but the memory space can be expanded and the Simulator rebuilt if more space is needed.

* Refer to Chapter 5, MAINTENANCE AND SUPPORT FEATURES, Expansion and Modification Capabilities regarding instructions and considerations for expanding memory space.
2.5 Instruction Formats

Each Intel 8086 instruction is from one to six bytes long, where the first byte contains the opcode and subsequent bytes contain additional information, data or addresses for the arithmetic logic unit. Each opcode belongs to one of six classes: Data Transfer, Arithmetic, Logic, String Manipulation, Control Transfer and Processor Control. There are eight opcodes which are not uniquely identified in the first byte and require a second byte, so they fall into a special "extended class" for this implementation of the Simulator. The class for each opcode is determined by using the value of the opcode as an index into an Opcode Look Up Table which has two hundred fifty six entries (all possible permutations of eight bits or 2**8) of which two hundred thirty four entries are presently used, leaving twenty two "spare" entries. The spare entries are filled with an invalid opcode class to increase the robustness of the Simulator.

The Extended class causes the eight non-unique opcodes to use three bits from the second byte to resolve their class. The resolution of the Extended class is not included in the Opcode Look Up Table because it would consume extra table space for only eight of the two hundred fifty six entries. A second (and smaller) table could have been created to resolve the eight non-unique opcodes into the necessary twenty two unique opcodes.
but that would have required an additional maintenance program. Instead, the resolution of these eight opcodes was built into the Simulator for sake of compactness.

In addition to the opcode class, the look up table contains three additional fields: a "key" field, a "direction bit present" field and a "width bit present" field. The opcode "key" field has nine possible values (zero through eight) and it provides the necessary information to interpret the opcode and subsequent byte(s), to identify the registers, addressing modes, displacements or immediate data necessary for the proper execution of this instruction. The "direction bit present" field indicates if bit one of the opcode should be interpreted as a direction bit (i.e. to or from the arithmetic logic unit), and the "width bit present" field indicates if bit zero should be interpreted as a width bit for byte or word operations.

The "direction bit present" and the "width bit present" fields imply that a single instruction might have more than one entry in the opcode table. For example, the ADD instruction supports both a direction bit and a width bit, so the ADD instruction has four different hexadecimal values (i.e. from 00 to 04) depending upon whether it was to or from a register or memory and if it was a byte or word operation. All four of these entries (00, 01, 10, 11) must have identical characteristics in
the Opcode Look Up Table for the instruction to be executed properly. Care must be taken when performing any modifications to take this into consideration.

Once the extended opcode has been resolved, the appropriate addressing mode selected, the effective address of the operand computed, the register designators selected, and the proper segment register selected, the Simulator invokes the appropriate module for this instruction. The selected Simulator software performs the appropriate manipulation of the status flags, Instruction Pointer, register files or memory array as required to complete the execution of this instruction.

A look up table is used to decompose each opcode into one of six classes where software procedures perform the appropriate manipulation of data. The look up table also provides information on how to decode addressing modes, displacements, registers, immediate data, the operand direction and the instruction width. A support program is also provided to perform any required maintenance on the look up table. (See Chapter 5, MAINTENANCE AND SUPPORT FEATURES, Opcode Look Up Table Maintenance.)
2.6 Summary

A Simulator has been developed to execute Intel 8086 instructions on a DEC VAX computer. The Simulator supports all of the Intel 8086 instruction set except three instructions which are dependent upon Intel hardware. A model for the central processing unit has been created which is functionally equivalent to the Intel 8086 hardware, but it does not support pipelining or simulate the supporting control circuitry. A one dimensional array is used to model Intel memory for storage of user programs and data. A look up table is used to interpret Intel 8086 opcodes and select the appropriate software procedures for the simulation.
Chapter 3
Input/Output Architecture

3.1 Preface

A computer is useful only if it can communicate with its environment, so the Simulator must have some facility for modeling input and output operations in order to assist the user in understanding how his program will interact with the outside world. The Intel 8086 CPU supports two methods for addressing I/O ports, fixed port addressing which includes the port number as the operand and variable port addressing which uses the contents of the DX register as the port address. Fixed port addressing is limited to eight bits of address information as immediate data or 256 ports, but the DX register is a sixteen bit register, so it is possible to have 65,536 unique port addresses, each having some data associated with them. In reality, an application would use only a few ports but port addresses are rarely contiguous except over a very short range, and they are usually not fully decoded. This suggests that if all 65,536 port addresses were to be provided, the I/O space would be sparsely populated. It was also not desirable to restrict the user to a particular range of port addresses if they were to accurately model their desired application.
The Simulator must also provide a mechanism for the program under test to communicate interactively with the user for a variety of reasons. A program may need to issue a prompt to the user, it may need data from the user or it may be necessary to deliver a status message to the user. To support this, a feature has been built in to the Simulator to allow the program under test to communicate directly with the user's keyboard and CRT even though they are being used to run the Simulator.

3.2 Overview

For these reasons, it was decided that the I/O model would allow the user to describe the key characteristics of the I/O ports needed to support their application and then the Simulator would translate references to those ports into operations on the simulated I/O ports. In this manner, the I/O model would support a limited number of ports, having any address the user desired within the following guidelines:

- Eight ports in total are provided, each eight bits wide.

- Sixteen bit wide ports must consist of two physically adjacent eight bit wide ports (addresses of n and n + 1).

- Any port can be either a read, write or bidirectional port.

- Data that is input to the program will be retrieved from an array associated with that port. The array is loaded from a disk file when the Simulator is started.
- Data that is output from the program will be stored in an array associated with that port. The array is saved in a disk file when the Simulator ends in a normal fashion.

- The port address "FF" (hexadecimal) is reserved for communications with the users' terminal.

In order to support these features, an auxiliary program is used to define an I/O Map which will contain the key characteristics for each I/O port and store those characteristics on the host computer's disk. Likewise a second auxiliary program must be used to create and review the input data for the program and to review the output data generated by the Simulator. This data file is also stored on the host computer's disk.

When the Simulator is started, the user is asked for the names of the I/O Map File and the I/O Data File if any I/O (other than with the user's terminal) is to be performed by his program. These files are then read from the host computer's disk into working arrays in the Simulator and the necessary pointers initialized. If the simulation session ends in a normal manner, these two files are updated and can be reviewed with the programs that created them. If the simulation session ends in an abnormal manner, no updates are made to the disk files and none of the intermediate I/O results are saved.
In order to perform input or output operations, the Simulator searches the I/O Map for the port address and retrieves the key characteristics. The Simulator then uses the Port Index and the Data Index to access the I/O Data array for the appropriate read or write operations. The Data Index is then incremented and stored back into the map for the next operation. It is possible however, to write to an input port or read from an output port, but these operations do not advance the Data Indices.

Limited I/O support is provided by the Simulator for testing and debugging programs. In order to provide a reasonably cost effective model, the number of I/O ports is limited but to preserve flexibility the user's addresses are mapped onto the Simulator's working I/O Data array. This map and input data are created by auxiliary programs which are also used to review the output from running the Simulator. The map and input data are loaded into working arrays in the Simulator when it is invoked, and the disk files are updated upon normal termination of the Simulator. The Simulator also supports I/O to the user's terminal.

3.3 Map Description

The I/O Map has seven fields which contain the necessary information to access and review I/O ports. These fields are described as follows:
Port Address: This is the address (in hexadecimal) of the I/O port that is referenced in the user's program. Any address can be used from "0" to "FFFF" (hexadecimal) except for the address "FF" (hexadecimal) which is reserved for the user's keyboard and CRT interface. However, the total number of addresses specified (excluding the keyboard and CRT ports) must be less than or equal to the "Maximum Number of I/O Ports" as defined at compile time of the Simulator (presently eight). All eight bit ports require the allocation of one address and all sixteen bit ports require the allocation of two adjacent addresses.

Port Width: This is the number of bits (either eight or sixteen) in the I/O port. The lower eight bits of a sixteen bit port are located at the designated address and the upper eight bits are located at the address plus one.

Upper/Lower Flag: This flag is used in conjunction with the port width to indicate if this is the lower or upper eight bits of a sixteen bit port. (For an eight bit port the flag is set to the "lower" state.) If the user accesses the upper eight bits of a sixteen bit port with a sixteen bit instruction, then the operation is adjusted to be aligned with the port's correct address. If an eight bit instruction is used to access the upper eight bits of a
sixteen bit port then the operation is executed only on the requested eight bits. This field was included in the map rather than force the user to adopt some other alignment convention for sixteen bit ports.

Auxiliary Index: This is the corollary index (i.e. port index) for the other eight bits of a sixteen bit port. This field is included so that the actual entries in the map do not have to be adjacent for sixteen bit ports. This field contains an invalid entry for an eight bit port.

Data Index: This index indicates how many entries have been read from the I/O Data array or written to the I/O Data array by the user's program. Each time an input or output operation is performed, this field is manipulated depending upon the value of the I/O State flag. (Please refer to the discussion in the I/O Data Description section for more details.) This field of the map is reset each time the Simulator begins execution and the map is loaded.

Input Index: This index indicates how many entries the user created in the data file for use by the program. It is adjusted only when entries are Input, Appended, Inserted or Removed from the data file by the support program.
I/O State: This flag indicates whether the port is an input, output or bidirectional port. Data can be input from an input or bidirectional port and written to an output or bidirectional port. Each of these operations has the potential for modifying the Data Index per the discussion in the I/O Data Description section.

The total number of I/O ports is specified by a constant "Max Number of IO Ports" and the maximum number of data points for each port is specified by the constant "Max Number of IO Bytes". The maximum number of data points is the same for all I/O ports since the data are stored in the Simulator in a rectangular array. The actual number of data points used can vary from port to port, but if any attempt is made to exceed the maximum value, the Simulator will crash. The "Max Number of IO Ports" and the "Max Number of IO Bytes" determine the size of the map files and data files on the host computer's disk and the size of the arrays that need to be allocated in the Simulator when it is compiled.

Each of these parameters can be modified, the modules recompiled and relinked to change the number of I/O ports or the amount of data for all ports as needed. It should be noted however, that the sizes of these fields have a direct impact on the response time to start and exit from the Simulator if I/O is being used.
Each time the Simulator starts and the user indicates that I/O is to be performed, the specified I/O Map and Data files are loaded into memory to support the simulation session. As a general rule of thumb, if the "Max Number of IO Ports" is kept the same, doubling the number of data points will cause the Simulator to take twice as long to initialize the I/O block before starting the Simulation session and twice as long to write the files out to disk before exiting and returning control to VMS. The same rule of thumb applies if the number of data points is kept the same and the "Max Number of IO Ports" is doubled.

It is for the benefit of the user community to keep these maps as small as is practical, not only to optimize initialization and exit times from the Simulator but also to optimize execution speed and page swapping in the host computer. If the maps are made excessively large then the program will take up more resident memory space and perhaps exceed what has been allocated to the user by the host computer. If this happens then the VMS Operating System stores part of the simulation session on disk, swapping pages in and out of memory as needed, further slowing the response time of the Simulator to the end user.
It should also be noted that any map or data file generated for use with a particular version of the Simulator will not be compatible with a different version of the Simulator where the "Max Number of IO Ports" or the "Max Number of IO Bytes" have been changed. The entire map and all data will have to be re-entered into new files before they can be used.

An I/O Map permits the user to have the I/O port addresses at the desired locations for this application without consuming excessive amounts of I/O real estate which would then be sparsely populated. Certain parameters have been built into the Simulator and support programs to permit modifications of the number of ports and amount of storage for data if necessary.

3.4 Map Generation And Viewing

Before running the Simulator to execute a program which reads an input port or writes to an output port, the user must create a map defining the necessary key characteristics of the I/O ports. A support program (IOMAP86) has been developed to perform this task as well as review an already existing map or to generate a hard copy of any map created with this version of the program. The IOMAP86 program supports eight commands as discussed below:
Create: Creates a new I/O Map with the file name supplied by the user. All of the map data is input by the user for the desired number of I/O ports. All unused ports are initialized with invalid values for increased robustness.

Read: Reads into memory a previously created I/O Map with the file name supplied by the user. This must be done before the map can be displayed, printed or modified. An invalid file name will cause the IOMAP86 program to display an error message and request a new command.

Modify: Allows the user to change an existing I/O Map by adding (A) a new port, deleting (D) an existing port, resetting (R) a datum (D) index, resetting (R) an input (I) index or resetting (R) both (B) data and input indices, or any combination of these operations until the user is finished (F) and wants to exit this function.

Display: Displays a port index plus the seven port parameters on the user's terminal.

Print: Creates a formatted, printable disk file from the current version of the I/O Map. This file contains descriptive headers for each of the fields, the port indices
and the seven parameters for each of the ports. After exiting from the map utility, the user can print this file at the desired printer.

Write: Saves the current version of the I/O Map in the disk file designated by the Create or Read operation without exiting the map generation and review program (IOMAP86).

Quit: Exit the map generation program without saving the current state of the map. An auxiliary check is made to be sure the user is intentionally leaving the program without saving any changes.

Exit: This is the normal exit from the program which saves the map in the disk file specified by the Create or Read operation. This operation is equivalent to a Write operation followed by a Quit operation.

These commands provide the user with adequate capabilities to create an I/O Map or make any necessary changes. The Simulator assumes that the necessary I/O Map and Data files have been created prior to its invocation if the user program is to perform any I/O other than to the user's terminal.
3.5 Data Description

The actual I/O data are stored in a rectangular array whose dimensions are the "Max Number of I/O Ports" by the "Max Number of I/O Words" as defined at compilation time of the Simulator. Data are read from the array or stored in the array as eight bit bytes based on the current parameters of the I/O Map.

The Port Address is located in the I/O Map and the necessary key characteristics are used to access the stored data. The Port Index identifies the appropriate linear array containing the data and the "Data Index" modulo "Bytes Per Word" is used to select the current word within the linear array and the remainder of the "Data Index" modulo "Bytes Per Word" is then used to select the correct byte of data. A word datum is accessed as two eight bit bytes of data, stored independently in two separate linear arrays as designated by the current parameters of the I/O Map and using the Auxiliary Index for the corollary Port Index.

Each datum is stored as an eight bit unsigned number in the same manner as a datum is stored in the memory array. Interpretation of the value of the datum (i.e. positive or negative integer, character etc.) is left to the user of the program being simulated.
As each datum is accessed, the Data Index is adjusted based upon the type of access and the port direction. Operations which fetch a datum from an I/O port cause the Data Index to be incremented if the port is designated as an input port. If the port is designated as an output port or a bidirectional port, the fetch operation proceeds to return the current value of the port, but the Data Index is not incremented. Operations which store a datum to an I/O port cause the Data Index to be incremented if the port is designated as an output port or a bidirectional port. If the port is designated as an input port then the current value of the datum in the port is overwritten by the store operation and the Data Index is not incremented. If the next operation on that port is to fetch a datum, then the value that was stored there will be the value that is retrieved.

Input and output data are stored in a rectangular array and accessed by Port Number and Data Index for compact storage. The Data Index is adjusted based on the type of access (i.e. input or output) and the type of port.

3.6 Data Generation And Review

Prior to running the Simulator to execute a program which requires or generates I/O data, the user must create a map defining the necessary characteristics of the I/O ports. Furthermore, if the program requires input data, the user must
also supply the data which will be read by the program during the simulation session. An additional support program (IODATA86) has been developed to create the input data set and review the I/O data generated by the program. The IODATA86 program supports nine commands as discussed below:

Create: Creates a new I/O Data file with the file name supplied by the user. Any valid file name supported by the VMS environment (including subdirectory paths) is acceptable. An invalid file name will cause VMS to abnormally terminate the IODATA86 program. All data points in the newly created file are presumed to be in an indeterminate state, i.e. the values of the data points as a result of the create operation are not specified.

Read: Reads into memory a previously created I/O Data array from the file specified by the user. This must be done before the data can be displayed, printed or modified. An invalid file name will cause the IODATA86 program display an error message and request a new command.

Modify: Allows the user to manipulate the existing data set for the specified port, but only if the port is an input port or a bidirectional port. Supported operations include appending (A) data, inserting (I) data, removing data (R),
changing a datum (C), displaying (D) data with index or any combination of these operations on the specified port until the user is finished (F) and wants to exit this function. Since free form editing of the data is not supported and most of these operations require knowledge of the index associated with each datum point, it is recommended that the user issue the command "Display with Index" to provide the required information before performing other operations.

Append: Allows the user to add a specified number of data points (up to the maximum number allocated) at the end of the existing data set.

Insert: Allows the user to insert a specified number of data points (up to the maximum number allocated) between two specified data points. Existing data after the specified point of insertion are shifted to make room for the new data.

Remove: Allows the user to remove a specified number of data points from the data set. If the points are removed from the middle of the data set, the retained values at the end of the data set are moved up to the deletion starting point so that the new data set is contiguous.
Change: Allows the user to provide a new eight or sixteen bit value for an existing datum point. Freeform editing of the data set such as changing a single character or moving around within the data set (i.e. such as with a screen editor) is not supported. Only one datum point (eight or sixteen bit) can be modified each time this command is executed.

Display With Index: Presents information to the user's terminal about the I/O port under modification. The information includes the port address, width and direction, the number of data points entered by the user, the number of data points accessed by the Simulator during it's most recent execution and the data for this port. Most of the other Modify operations require the index for the datum point or points before other the operations can be performed. The Display With Index function provides this information for the user so that other operations can be successfully completed.

Finished: Indicates that the user has completed the desired Modify operations.
Input: Allows the user to input a specified number of data points for an input or bidirectional port only. The stream of data is stored in the I/O Data array based on the specified port address.

Display: Displays the I/O port information and the data for that port on the user's terminal. In addition to the port address, width and direction, the Display function also indicates the number of data points entered by the user and the number of data points accessed by the Simulator during its most recent execution and the data for this port. While the data are displayed in a sequential manner, no index information is provided by this command.

Print: Creates a formatted, printable disk file of the current version of the data for all ports. In addition to the port address, width, direction and data, the Print function also indicates the number of data points entered by the user and the number of data points accessed by the Simulator during its most recent execution. After exiting from IODATA86, the user can queue the file to be printed.
Write: Saves the current version of the I/O Data array in the disk file designated by the Create or Read operation without exiting the IODATA86 program. The I/O Map file is also updated if the user has Appended, Inserted or Removed any data points causing the Index parameter to change.

Quit: Exit the data generation and reviewing program without saving the current values in the I/O Data array. An auxiliary check is made to be sure the user is intentionally leaving the program without saving any of the modifications.

Exit: This is the normal exit from the program which saves the data in the I/O Data array. Data are stored in the disk file designated by the Create or Read operation. The I/O Map file is also updated if the user has Appended, Inserted or Removed any data points causing the Index parameter to change. The Exit function is equivalent to the Write operation followed by the Quit operation.

These commands provide the user with the ability to review both the input and output data which has been directed to or received from the I/O ports as well as to create input data for use by the Simulator. The Simulator assumes that the necessary I/O Map and Data files exist prior to its invocation.
3.7 User Communications

The Simulator must also support interactive communications between the user and the program under test. This feature required the use of QIO calls to communicate directly with the VMS environment.

Any input or output operations to or from the port addresses designated for the keyboard and CRT are treated differently from the normal I/O operations. These port addresses are not built into the I/O Map or Data files, so that communications can occur interactively with the user without the overhead of having to specify and load an I/O Map and a previously defined data set. Communications with the user's keyboard and CRT also take precedence over other I/O operations. This implies that if a port is defined in the I/O Map with the same address as the keyboard and CRT, that port in the I/O map is unreachable.

In order to support the interaction with the user while running the Simulator, it was necessary to implement these communications directly with the DEC VMS environment using QIO calls. This approach was necessary in order for the Simulator to process each input keystroke or character to be displayed as a separate I/O step, which is the desired implementation at this "primitive" level of computing. This implementation is compatible with the extended VAX Pascal provided by DEC.
The keyboard and CRT ports are eight bits wide but the CRT port supports only seven bit wide data, ignoring the most significant bit of the byte. This was done to insure that the messages sent to the user would be composed of those characters which are in the printable ASCII character set.

Addresses have been chosen for the user's keyboard and CRT which are fixed at compile time for the Simulator. The selected addresses are "FF" hexadecimal for both the keyboard port and the CRT port, thus making this a bidirectional port. These particular addresses were chosen so they can be accessed with the simpler fixed port I/O instructions in the 8086 repertoire, as well as located out of the most frequently or most likely to be used part of the I/O address space. The port addresses can be modified if necessary, and the Simulator recompiled if it becomes essential that these addresses be used for another purpose.

Communications are supported for the user to interact via the user's keyboard and CRT with the program being simulated. The keyboard and CRT addresses are predetermined as the result of building in this feature, but they can be modified if needed and the Simulator recompiled.
3.8 Summary

Two types of I/O are supported by the Simulator, mapped I/O and terminal I/O. For mapped I/O, the user specifies the key I/O port characteristics using the program IOMAP86 and the input data using the program IODATA86 prior to the simulation session. The user can then examine the output data as desired after the simulation session, also using the program IODATA86. The maximum number of ports and the number of data points per port can be modified and the Simulator and support programs recompiled if changes are necessary. Interactive communications between the program under test and the user are possible using his or her terminal. Likewise, the port address for the keyboard and CRT can be changed and the Simulator and support programs recompiled if necessary.
4.1 Preface

A Simulator is a useful tool for verifying the proper operation of a program under development, but other tools must also be available in order to control the execution of that program and to examine or modify the program or the parameters that are being used by the program. These features have been included in the Debugger in order to facilitate the use of the Simulator. There are many features that could be included in the Debugger to make life easier for the user, but they can also cause the user to lose touch with the environment in which he or she is working. Since this project was developed as a teaching tool, a minimum number of features have been provided to encourage the user to understand what the program is doing. This does not mean that the Debugger has to be difficult to use. In fact, user friendliness is enhanced by the simplicity of the tool set.

4.2 Overview

Consistency is a major feature to emphasize in order to make a tool set easier to use. All of the commands that perform similar operations should do them in a similar manner. For example, all of the commands which access VMS files should have the same format, all of the commands that display memory,
registers or the flags should have a common user interface and all of the commands that modify memory, registers or flags should be consistent both among themselves and with the display commands. This makes it easier for the user to learn how to use the features and easier to remember them later. Consistency also makes the Debugger easier to develop and maintain since a common routine would be called to implement similar functions.

The Debugger was designed to support as many of the Intel DEBUG-88 features as was practical. There are many features which are identical, including register and flag names, the commands used to examine the registers, memory, flags and I/O ports and some of the commands to control program execution. However, there are also substantial departures from DEBUG-88. It was decided that in order to keep the user closer to the operation of the hardware, symbolic debugging would not be supported. In the interest of simplifying the Debugger, the on-line evaluation feature was not included, it is not possible to change the radix used for the presentation of information and compound commands are not available to the user.

Some of the available commands were also enhanced. It was felt that two breakpoints were inadequate to control the program execution, so the number was increased to sixteen. Each breakpoint can be set, displayed or cleared individually, or all
sixteen breakpoints can be treated as a group to be displayed or cleared. Additional commands were added to control program execution such as START, CONTINUE and RESUME. As a result of these changes the concepts of the "break register" and the "go register" are no longer supported.

In order to provide sufficient flexibility for implementing the Debugger commands, they are contained in external disk file. The commands are then read from the disk file and loaded into a look up table each time the Debugger is invoked by the Simulator. There are four active fields in the Debugger Keyword Look Up Table. They include the keyword name, a class code, an identifier and a width code. Each keyword name is limited to eight characters, but only the first three characters are used in identifying the keyword. When parsing the command line, a keyword is isolated and compared against each keyword name entry in the table until a match is found. The remaining characteristics in the table tell the Debugger how to interpret the command. The class code indicates in which of the thirteen classes (i.e. Utility, Qualifier, Operand, I/O, blank, Logic, Stack, Register, Flag, Memory, Breakpoint, Execute or unused) the command belongs and is the major discriminator for all of the Debugger commands.
Once the class for a given command has been determined, the identifier indicates the how that specific command should be interpreted. It is the identifier value for example that causes the BYTE command to display eight bits of information in hexadecimal and the SINteger command to display the same eight bits as a signed, decimal number. The fourth field in the keyword look up table is the width field and it indicates the width parameter which is used primarily for the register and I/O port classes. The width field provides the necessary information to distinguish the AX register from the AL register in the subsequent Debugger manipulations since both have the same register identifier.

As previously indicated, the keyword name is a maximum of eight characters long, and must be unique within the first three characters. However, there are some execution class commands which are used much more frequently than others, so special one letter commands are supported that duplicate the entries for the corresponding full commands (e.g. R for RESume). There is an additional constraint in selecting keyword names since none of the command names can be composed entirely of hexadecimal (i.e. letters A through F) and numeric characters. If this were done, the Debugger would not be able to distinguish between a hexadecimal number and a Debugger command. This implies for example that "ABE", "AB2" and "C" are not valid Debugger commands.
Storing the Debugger commands in an external disk file required the development of an additional support program (KEYWORDLUTGEN) to create and perform maintenance on the keyword table. See Chapter 5, MAINTENANCE AND SUPPORT FEATURES, Keyword Look Up Table Maintenance for more details.

A Debugger has been developed to provide the necessary commands to control the execution of a program under simulation. The commands have been developed to be easy to learn and use and consistency among commands has been emphasized. Many of the commands are identical to Intel's DEBUG-88, but some departures were necessary and some enhancements were also implemented.

4.3 Command Types

There are three different types of commands in the Debugger. The first type are utility commands and they allow the user to communicate with the outside world, the second type are execution commands and they allow the user to control the execution of the program under test and the third type are examine/modify commands and they allow the user to examine or modify the memory, registers, flags etc. that are used by the Simulator. All of these commands however, have a similar format to provide consistency and ease of use. If a command is intended to display or modify an object, then that command followed by a carriage return will display the object and that
command followed by an equal sign, a value (even if it is the null value) and a carriage return will modify the object, setting it to the new value. This is true if the object is a memory location, a register, a flag, a breakpoint or a file name. This similarity between commands makes the Debugger easier to learn and use.

4.3.1 Utility Commands

There are three Utility commands available to the user:

* Load

  + LOAd <cr>: Displays on the user's terminal the name of the file which has been loaded.

  + LOAd = Filename <cr>: Loads the specified linked and located Intel executable file into the Simulator's memory for execution.

* Log

  + LOG <cr>: Displays on the user's terminal the name of the file which contains the commands the user has logged for this simulation session.

Note: "<cr>" is a carriage return and the colon is not part of the command.
+ LOG = <cr>: Disables logging of Debugger commands which the user is entering.

+ LOG = Filename <cr>: Enables logging of the Debugger commands which the user is entering. Commands are written to the file specified.

* EXIt: Leave the Debugger and Simulator, write the I/O Map and I/O Data files to disk if they were used, close the log file if it is open and return to the VMS environment.

Where a file name is specified, a full path name (node::disk:[user]filename.extension) can be entered, but it must be less than forty characters in length including all punctuation. An invalid path or file name will cause the Simulator to crash.

4.3.2 Execution Commands

Execution commands are provided to enable the user to start and stop the Simulator in a controlled manner. There are six basic commands which cause the Simulator to begin execution (i.e. GO, STArt, CONtinue, RESume, STEp and PSTep).

* GO or STArt will begin the execution of the program at the address specified by the Code Segment Register and Instruction Pointer values. If numeric values are used in
the GO or STArt command, these values will replace the contents of the Code Segment Register and/or the Instruction Pointer before execution of the program begins.

* RESume or CONtinue will cause the program to resume execution at the address specified by the current values of the Code Segment Register and Instruction Pointer. All of the registers, stack, memory, flags etc. will be unchanged from where the program was interrupted unless they were modified by the user.

* STEp will cause the Simulator to execute one instruction and then return control to the Debugger.

* PSTep will cause the Simulator to execute one instruction and then return control to the Debugger unless that instruction is a subroutine call. When a subroutine call is encountered, the Simulator will execute the entire subroutine before returning control to the Debugger at the first instruction past the subroutine call unless a breakpoint is encountered.

These commands can be abbreviated in the usual manner, but since they are the most frequently used commands, special one letter abbreviations (i.e. G for GO or STArt, R for RESume or CONtinue, S for STEp and P for PSTep) have also been included for the user.
It was generally perceived that the two breakpoints specified in DEBUG-88 were not sufficient for debugging programs, so the breakpoint structure was modified to accommodate sixteen breakpoints. The command syntax for displaying the location of a breakpoint is the breakpoint name followed by a carriage return (e.g. BP0 <cr>). If the breakpoint name is followed by an equal sign and a carriage return (e.g. BP0 = <cr>), the "null" value is assigned to that breakpoint, or that breakpoint is deactivated. If the breakpoint name is followed by an equal sign, a value and a carriage return (e.g. BP0 = CS:A3 <cr>), then that value is interpreted as the address at which the breakpoint is to be set. All valid addressing modes that can be used in the Debugger are acceptable for specifying the breakpoint address. If a segment register is not specified, then the Code Segment Register is used as a default.

Breakpoints are dynamically installed and removed from the source code during simulation. This is done so that when a user examines a memory location on which a breakpoint has been set, the expected value of memory is found rather than the breakpoint opcode. Just prior to starting or resuming execution, all breakpoints are installed and the opcodes saved. When a breakpoint is encountered, the breakpoint opcodes are removed, the original opcode is restored and control is returned to the Debugger. It is very important for the correct operation of the program, the Simulator and the Debugger that the user place
breakpoints only on the first byte of an instruction but there is nothing in the Simulator or Debugger which would guarantee this. If a breakpoint is not placed on an instruction boundary, this would most likely cause the incorrect operation of the program rather than executing the breakpoint as expected.

All sixteen breakpoints can be treated collectively by using the command ALLbp. This allows the user to see all of the addresses on which breakpoints have been set. When the command is followed by an equal sign and a carriage return all breakpoints are removed from the program under test.

Any time a program is running and a breakpoint is encountered, the user is notified and control is returned to the Debugger. The user then has control of the debugging session and can use any of the commands to examine or manipulate the registers, flags, memory or stack as needed to modify the program's operation.

In order to provide better control over a malfunctioning program under test, it is possible to abort the Simulator and return control to the Debugger at the next instruction by entering a Control D. Thus if a user program is caught in an infinite loop, or executing where no breakpoints were entered, control can be returned to the Debugger in an orderly manner, the user
can set the appropriate breakpoints and then resume execution.
If the Debugger or Simulator become corrupted, a Control C or Control Y will return the user to the VMS environment, but no recovery of the simulation session is possible when this action is taken.

4.3.3 Examine/Modify Commands

Examine commands are provided to enable the user to verify that memory, registers, flags etc. contain the expected intermediate values during program execution. The modify commands enable the user to change the intermediate values if they are not correct or to alter the program so that correct values are produced. There are six commands which can be used to examine memory, and they all require a full twenty bit physical address (i.e. a segment register value and a displacement) in order to access the desired memory location. However, these commands have been simplified by assuming that the user desires to examine the memory location pointed to by the current values of the Code Segment Register and Instruction Pointer. These assumptions are easily modified by providing more information with the desired examine memory command. If no segment register information is included, then the Code Segment Register value is used and the desired memory address can then be entered directly in hexadecimal or the contents of any register can be used as the displacement. Alternatively, the Code Segment Register value
can be replaced by entering a hexadecimal value for the segment or the value of any register can be used by specifying its name. Some of these combinations may be meaningless in terms of the utilized memory space, but the modes of addressing are available.

There are also two qualifiers that can be used when displaying memory, the LENgth and the TO parameters. The LENgth parameter allows the user to specify how many times the command should be applied, and it displays different amounts of memory for the various commands. For example, the BYTe command displays one byte of memory and the POInter command displays four bytes of memory. The command sequence "BYTe LENgth 2" will display two bytes of memory, but "POInter LENgth 2" will display two pointers each of which is four bytes long, for a total of eight bytes displayed. The TO qualifier specifies the address where the display process will stop and therefore has a different effect. To display three bytes of memory using the BYTe command, the user would enter "BYTe" nnnn "TO" nnnn + 3. Since the WORD command displays two bytes of memory at once, the number of bytes is increased to insure that the desired information is displayed. Therefore the command sequence "WORD" nnnn "TO" nnnn + 3 will display four bytes or two words of memory.
The available commands differ only in the way the contents of memory are displayed to the user. The six commands are as follows:

* **BOOLean:** Interprets the information in memory eight bits at a time and displays it as TRUe or FALse. If the eight bit value in memory is an odd number, then it is interpreted as TRUe.

* **BYTe:** Displays the information in memory eight bits at a time, as a hexadecimal number.

* **INTeger:** Displays the information in memory sixteen bits at a time as a signed, decimal number.

* **POInter:** Displays the information in memory thirty two bits at a time as a sixteen bit segment value followed by a sixteen bit displacement value.

* **SINteger:** Displays the information in memory eight bits at a time as a signed, decimal number.

* **WORd:** Displays the information in memory sixteen bits at a time as a hexadecimal number.

All of these commands can be used to modify memory as well as display it. If the command is followed by just a carriage return, then the information will be displayed but if the
command is followed by an equal sign and a value, then the specified memory location is updated to the new value.

The STAck command is a special form of the memory display command which simplifies the process of displaying the contents of the stack. The STAck command uses the Stack Segment Register and the Stack Pointer for the segment register value and displacement respectively for accessing memory rather than the Code Segment Register and Instruction Pointer assumed for other memory accesses. This command can also have a LENGTH qualifier to indicate how many of the stack entries should be displayed. No changes to the stack entries are possible using this command.

Registers are less complicated to address than memory, but the same command syntax is used. To display a register's value, the register name is followed by a carriage return (e.g. AX <cr>) and to modify the contents of the register, the register name is followed by an equal sign, the desired new value for the register and a carriage return. Both the short and long forms of the register names are supported (e.g. AL and RAL). The flags can also be assembled and displayed as a register using the RF command. It is possible to display the contents of all registers including the assembled Flags in the RF register by using the REGister command.
In a similar manner, the flags can be examined and modified. To display a flag's state, the flag name is followed by a carriage return (e.g. ZF <cr>) and to modify the state of the flag, the flag name is followed by an equal sign, the desired new state of the flag and a carriage return. Although any hexadecimal value can be entered for a new value, all bits off will be interpreted as "clear" or "false". This implies that any other value will result in the flag being set or interpreted as "true". All nine flags can be displayed at once using the FLAg command.

The Debugger also allows the user to examine the I/O ports without altering the Data Index of that port. This means for example that the user can look at an input port and examine the next value the program will retrieve without causing the Data Index to advance. There are two commands to examine or modify the I/O ports, PORT and WPOrt. The PORT command operates on eight bits of data in either an eight or a sixteen bit port, and the port can be an input or an output port. Likewise the WPOrt command operates on sixteen bits of data for an input or an output port. However, the Debugger will correctly display or modify sixteen bits of data if the upper address of a sixteen bit port is specified for the WPOrt command. The commands also allow the datum stored in the ports to be modified in the same manner as the memory commands. Writing a datum to an output port will overwrite the last value stored and writing a new datum to an input port will modify the datum that is about to be
read from that port. It is not possible however, to move "forward" or "backward" in time for a given I/O port and examine or modify other data values. The user only has access to the data that currently resides in the ports. The LENgth and TO qualifiers are also supported, and cause multiple ports to be examined if they are present in the I/O Map.

Three types of commands are provided in the Debugger, utility commands, execution commands and examine/modify commands. All of the commands operate in a similar fashion, so that the command followed by any operands or qualifiers and a carriage return will display the requested information and the command followed by an equal sign, a value and a carriage return will modify the object and set it to the new value.

4.4 Limitations

As previously indicated, the Debugger was intentionally developed as a primitive tool to encourage the user to remain closer to the actual operation of the hardware. This means that only the essential commands for debugging were developed and some features were intentionally not supported. These unsupported features include symbolic debugging, evaluation of expressions, manipulation of the display radix, disassembly of instructions, support for displaying line numbers and module
names and support for compound commands. See Appendix G for a complete list of DEBUG-88 commands that are not supported.

Since symbolic debugging is not supported, it was not necessary to maintain a symbol table in the Debugger and therefore many of the DEBUG-88 commands that reference or manipulate the symbol table are not supported. Symbolic debugging is not supported because it discourages the user from thinking about how the processor is interpreting the segment register and displacement values. The evaluate expression command also relies upon the symbol table for the correct interpretation of expressions, and since the symbol table is not maintained the evaluate expression function is not supported.

The option to change the radix of the information displayed is not supported. Although there is nothing precluding the support of this feature, most of the literature on the Intel 8086 processor is in hexadecimal and support for binary, octal or decimal display formats seemed counterproductive. When learning a foreign language, it is easier if the user "thinks" in the new language rather than think in the old and translate into the new language, so most of the information displayed by the Debugger or entered for commands is in hexadecimal notation. The use of decimal notation is generally confined to those instances where integer counts are required by the VMS environment and where
errors would cause the Simulator or Debugger to malfunction. Anywhere a decimal value is required, the communications with the user clearly indicates this.

The disassembly of instructions is likewise not supported. While it is a feature found in many of the more sophisticated debuggers, it also takes the user a step further away from the reality of the machine code world. The display of line numbers and module names is not supported since this feature would add a level of complexity to the Debugger which is not essential to its proper operation. The use of compound commands is also not supported since the typical user will be a first time student and most user programs will not be complex enough to warrant the inclusion of compound commands. Instead of supporting compound commands, additional breakpoints were included to increase the power of the Debugger.

These limitations were carefully evaluated to insure that the essential commands were provided for the user to debug a program, and the complexity of the Debugger was minimized.
4.5 Summary

A Debugger has been developed to provide the necessary commands to control, monitor and modify a program under test. Consistency and ease of use were major considerations in developing these commands as well as providing a smooth transition to the Intel program, DEBUG-88. Commands have been provided to interface with the VMS environment, start and stop the Simulator and examine and modify registers, flags, memory etc. as required to debug a program.
Chapter 5

Maintenance and Support Features

5.1 Preface

The Simulator uses an Opcode Look Up Table to obtain information on how the opcodes should be interpreted. In order to develop the Opcode Look Up Table a program, OPCODELUTGEN, was developed which can also be used for any necessary maintenance. Likewise, the Debugger uses a Keyword Look Up Table to interpret commands, and the maintenance program KEYWORDLUTGEN has been developed to create the look up table and can be used to make any necessary changes.

There are several command procedures and support programs which were developed during the course of designing and building the Simulator. These tools can also be useful in performing maintenance such as expanding user memory, changing the number of I/O ports or changing the amount of data allowed per port. The Simulator response time must also be taken into consideration when modifications are made so that it remains acceptable to the user community. Some installations may not support all of tools (such as the DEC/MMS build commands) but other alternatives can be developed to provide the equivalent functions. Some of the extensions to VAX Pascal were very useful in the development of the Simulator. The global and
external attributes for variables and procedures, the unsigned data type and associated operators and the include feature were some of the fundamental building blocks for the Simulator. The asynchronous trap also required a variety of extensions to support the Control D function.

5.2 Opcode Look Up Table Maintenance

The Simulator uses an Opcode Look Up Table to obtain the necessary information for the interpretation and execution of each opcode. The look up table is stored as an external file so that maintenance of the table can be done by a qualified individual without recompiling the Simulator. A program was developed (OPCODELUTGEN) to create the table and provide the necessary support functions for maintaining and enhancing the active opcode set. This program also provides a print function to generate a hard copy of the table.

The OPCODELUTGEN program allows the user to modify the Class, Key, Direction Bit Present and Width Bit Present fields of the table. The Class field is limited to a one letter abbreviation for each of the six valid classes plus the Extended and Invalid classes. The Key field inputs are limited to values from zero to eight, although all values might not be valid for all classes. The Direction Bit Present and Width Bit Present fields are filled with true/false responses depending upon whether the
opcode requires a direction bit and/or a width bit. The table is accessed by using the hexadecimal value of the opcode as an index for the information.

Although this program is provided to perform maintenance on the Opcode Look Up Table, it requires a knowledgeable and extremely careful maintenance programmer to make any changes.

5.3 Keyword Look Up Table Maintenance

To simplify the operation and maintenance of the Debugger, its commands are also stored in a separate file which is loaded into a look up table at execution time. This makes the Debugger less sensitive to maintenance changes of the keywords, but it also requires an additional program (KEYWORDLUTGEN) to perform any required maintenance on the keywords.

The KEYWORDLUTGEN program provides the maintenance programmer with the ability to change the spelling of a keyword name or add keyword entries as needed. There are one hundred one (101) entries in the keyword table, including six spare or unused entries for future commands or abbreviations as deemed necessary. It is the responsibility of the individual making any modifications to insure that the change or new entry in the keyword table does not conflict with an existing entry and cannot be interpreted as a hexadecimal number. The maintenance
program does not perform any checks for collisions or prevent the entry of abbreviations which would render the Debugger inoperative. The maintenance program is provided primarily for the simple expansion of commands, changes in spelling (such as renaming the command "LOG" to "JOURNAL") and the addition of other abbreviations as deemed expedient by the user community.

5.4 Command Procedures

One of the optional DEC utilities is the Module Management System (DEC/MMS) which minimizes the amount of work to be done to build a large, complex software system. DEC/MMS provides the user with the capability to describe the relationships between software modules and then when one of the modules has been modified, these relationships are reviewed and a user defined sequence of operations is invoked as appropriate. This provides the ability to compile, update an object library and link only those modules which must be processed to properly reflect the recent change. Four such DEC/MMS command files have been created to support the generation of the Simulator, the I/O Map and Data generation and viewing programs, the Opcode Look Up Table maintenance program and the Debugger's Keyword Look Up Table maintenance program. If the user's installation does not support DEC/MMS, command file (COM) procedures have been developed to build the programs, but they do not have the level of sophistication of DEC/MMS and cause all of the support modules to be compiled and updated in the library, even for a
small change which might otherwise affect only one module. If these COM procedures are used, excess work may be performed, and the user should consider issuing the necessary commands manually for small changes.

5.5 Expansion And Modification Capabilities

Various expansion features were reviewed in the previous sections. Those with a significant impact on system performance or response time are summarized here.

It is possible to expand the memory space for the user program beyond the present eight kilobytes in the Simulator. The file CONST.DEFN must be edited, and the constant "HighMemoryLimit" given a new value. (It is recommended that no changes be made to the value of the constant "LowMemoryLimit".) Since this is the number of VAX words that are reserved and each word contains four bytes of Intel information, this value should be twenty five percent of the total desired memory including interrupt vectors, stack, data and code space. Once this modification is made, the Simulator, IOMAP86, IODATA86, KEYWORDLUTGEN, and OPCODELUTGEN must be recompiled and relinked. If DEC/MMS is used, the minimum number of modules needed to support the change will be recompiled and relinked. If DEC/MMS is not available, all programs should be recompiled and relinked to ensure that all references to the altered variables are correct. Since the
file CONST.DEFN is used in many modules, this modification is not a candidate for performing the updates manually.

As the user memory space is increased, care must be taken to monitor the Working Set Size allocated to the user by the VMS Operating System. When the size of the Simulator image exceeds the allocated Working Set Size, the VMS Operating System will start page swapping to keep the active portion of the user program in memory. This may appear as a severe degradation in response time and new user parameters should be negotiated with the computer center.

The maintenance person also has the ability to modify the number of I/O ports available to the user community. It should be noted that modifying the number of I/O ports or the number of data points per I/O port will invalidate all existing maps and data sets for all users, and all of that information will have to be re-entered. No provision has been provided to translate from one map size to another or from one data set size to another. The users should also be made aware that increasing the map size with the same number of data points or increasing the number of data points with the same map size will make the Simulator start up time longer, as additional data points must now be loaded.
To modify the number of I/O ports or the number of data points per I/O port, the file IOCONST.DEFN must be edited. The constant "Max Number Of I/O Ports" will modify the number of I/O ports while "Max Number of I/O Words" and "Max Number Of I/O Bytes" must both be modified to increase the number of I/O data points. The constant "Max Number Of I/O Bytes" must always remain four times the "Max Number Of I/O Words" since there are four bytes of I/O data per word reserved on the VAX. Once the modifications are made, the Simulator, IOMAP86, IODATA86, KEYWORDLUTGEN, and OPCODELUTGEN must be recompiled and relinked. If DEC/MMS is used, the minimum number of modules needed to support the change will be recompiled and relinked. If DEC/MMS is not available, all programs should be recompiled and relinked to ensure that all references to the altered variables are correct. Since the file IOCONST.DEFN is used in many modules, this modification is not a candidate for performing the updates manually.

During the development of the Simulator, a maximum VMS path and file name length was established at forty characters. This can be changed by editing the file CONST.DEFN and selecting a new value for the constant "FilenameLength". Once the modifications have been completed, the necessary program modules must be recompiled and relinked for the Simulator, IOMAP86, IODATA86, KEYWORDLUTGEN, and OPCODELUTGEN.

- 74 -
Provisions are available for modifying the amount of memory, the number of I/O ports, the number of data points per I/O port and the length of the VMS path name and file name. Once these modifications are complete, the Simulator and support programs must be recompiled and relinked to incorporate the changes. Some of the response time implications and precautions surrounding these changes were also discussed.

5.6 Language Extensions And Deviations

One of the goals in the development of this Simulator was to use a structured, high level language not only for ease of development but also for ease of maintenance. The language chosen was Digital Equipment Corporation's implementation of Pascal and the choice was based in part on the availability of the compiler on teaching systems and the supported extensions to the language. There were only three Simulator features that required the author to digress from Pascal. These were the need to interrupt the Simulator at the next instruction (Control D), the need to support interactive keyboard input and output and the need to check for the existence of a file prior to opening it to prevent the Simulator from crashing as the result of an incorrect user input.
Many of the DEC extensions to Pascal were used in the development of the Simulator. These included the following:

- The ability to assign a procedure the global attribute and reference it as an external procedure from another module was essential in order to segment the development and debugging of the Simulator.

- The ability to "include" another file at compile time was essential to maintaining version control. This guaranteed that constants, data types and variables were only defined in one place for all modules and support programs.

- The ability to define global variables in one procedure and reference them as external variables from another procedure simplified the module interfaces. For example, the Memory space is declared as a global variable, and defined at the highest level module even though it is only accessed by two of the lowest level modules. Since the variable and its associated type are defined in "include" files, they are only available to procedures that explicitly require them. However, if they were passed as part of the procedure call from the highest to the lowest module, all modules would have access to Memory in an uncontrolled fashion whether or not they needed the access.

- 76 -
- The ability to define constants in binary and hexadecimal formats increased the accuracy and readability of their definitions.

- The data type UNSIGNED and the unsigned operators UTRUNC, UAND, UOR, UNOT, UXOR, UINT and INT allowed the Simulator to operate in an easily understood manner on bit patterns.

- The use of a VOLATILE variable and ASYNCHRONOUS and UNBOUND procedure attributes were required to support the Control D function.

- The ability to write variables to an output device using the HEX macro, including parameters to control the field widths, was of great value in maintaining consistent entry and display formats.

- The ability to OPEN files as sharable made the use of look up tables practical.

The reader is referred to the DEC documentation on these Pascal extensions for further definition, but the Simulator would have been more difficult if not impossible to develop without them.
As previously indicated, the development of the user I/O interface and the support of the Control D operation required the use of non-Pascal features. When a user is performing I/O with a Pascal program, Pascal requires complete lines to be entered before they are processed. The Simulator needed the ability to accept and process a single character without a carriage return in order to support I/O to the user terminal with the IN and OUT instructions. This support was provided by linking in the VMS Library routines and performing $QIOW calls to the assigned terminal. This routine is essentially a "queued-I/O-wait" call to the specified device and it was established to accept one character at a time. Implicit in the support of these procedures was the facility in Pascal that allowed them to be included in the compilation. Without the LIB$STOP, $CANCEL, $ASSIGN and $QIOW procedures, terminal I/O on a per character basis would not have been possible.

The support for Control D required the ability to accept input from an asynchronous trap, or an AST with the ASYNCHRONOUS and UNBOUND attributes using the previously mentioned VOLATILE variable. This allowed the program to "accept" a Control D whenever it was typed, record its receipt and continue with the normal processing. The Simulator then looked at the variable upon the completion of an instruction and return to the Debugger if a Control D had been entered. This provided the user with
the ability interrupt the simulation session and return control to the Debugger without aborting the program.

A FORTRAN subroutine was also acquired to check for the existence of a file prior to opening it. This provided the user with a greater degree of fault tolerance to misspelled names, invalid directories, subdirectories or path names and allowed them to recover from these problems rather than causing the Simulator or Debugger to crash when they attempted to reference "non-existent" files. This is a blessing to those users who are prone to typographical errors.

Pascal was chosen as the high level language in which to implement this Simulator based in part on its availability and its extensions which made this project practical. Input and output to the user's terminal required "queued I/O wait" calls and support for Control D required additional features of VMS. A FORTRAN subroutine was also included to improve the fault tolerance for the user file names.

5.7 Summary

During the design of the Simulator and Debugger, maintenance programs were also developed to support the external Opcode Look Up Table (OPCODELUTGEN) and Keyword Look Up Table (KEYWORDLUTGEN). Additional command procedures and support
programs were created to assist in the development process. The Module Management System provided the capability to minimize the amount of compilations required, while providing an up-to-date version for debugging. This minimized confusion and provided a certain amount of configuration control over the process. Certain expansion capabilities were provided by changing any of four key constants to add user memory space, I/O ports, I/O data or longer path and file names for VMS. The Simulator was implemented in Pascal for ease of development and maintenance and the DEC VMS extensions proved essential in the implementation process.
Chapter 6

Summary and Conclusions

6.1 Preface

The primary reason for undertaking this project was to develop a Simulator for the Intel 8086 microprocessor which would permit students to have "hands-on" experience without the expense of the processor hardware. In a teaching environment, it is difficult to justify the cost of a set of dedicated hardware for each student, so it was the goal of this project to provide a Simulator on a multiuser minicomputer for students to debug their programs to some level of proficiency prior to using the hardware. As a result, students require less time with the hardware and it does not become the limiting factor in the learning process. It is the author's opinion that this goal has been achieved.

A secondary goal was to demonstrate that a Simulator could be developed in a modern, high level language which simplifies the development process and aids in the maintenance of the Simulator after the project is completed. Pascal was chosen as the high level language and the minicomputer of choice for the implementation was the VAXcluster from Digital Equipment Corporation. DEC's version of Pascal provided several extensions which proved to be essential to the development of
the Simulator, and it has been demonstrated that building a Simulator with a modern, high level language is indeed practical.

6.2 Observations On The Simulator

The following observations have been made regarding the development of the Simulator and its subsequent performance:

- The Simulator has a "reasonable response time" in the student environment. When the Simulator is executing on a VAX 8600 with approximately 100 other student users, the response time for communicating with the user is about the same as the response time when using the editor.

- The size of the complete Simulator and Debugger with all look up tables loaded, eight kilobytes of Simulator memory allocated and eight I/O ports each having 256 bytes of data is less than two hundred blocks of DEC memory. This compact size also contributes to the speed of execution.

- Although minimal user feedback has been obtained, the Simulator appears to satisfy the need for a learning tool.
Initial user feedback however, indicated that default values were expected for the Stack Segment Register and the Stack Pointer and that they should be consistent with the Intel Debugger. This was despite the fact that the User's Guide provided with the Simulator and Debugger clearly stated that no initial values would be assumed for any registers except the Code Segment Register and the Instruction Pointer when a program start directive was included in the source code. This reinforced the author's perception that users are frequently unaware of the extent of the services provided to them with more sophisticated debuggers and that first time students need to understand that the user is responsible for everything when they work with a standalone microcomputer.

The Simulator was not intended for a professional or industrial environment and does not appear suited to these applications. The simulation of input and output is too limited for "real world" applications and the Simulator does not support any calls to library utilities or interaction with an operating system to take advantage of already existing software. The cost of developing improvements to make this system suitable for the professional or industrial environment does not seem justifiable when applications for these environments ultimately need the hardware anyway.
As a learning tool, the Simulator supports most of the instruction set of the Intel 8086 processor, including the input and output instructions. However, the use of external programs to establish I/O Maps and to edit input data and review output data makes it awkward to use I/O operations. The user must construct an I/O Map, key in data, execute the test program, and then review the output all of which requires running four different programs before the user can review any I/O results. A minor change in the input data set then requires three of the programs to be run again and this quickly becomes confusing and time consuming for the user, but it was the most practical method to provide this capability.

Conversely, interactive I/O with the user was implemented in a more elegant fashion than expected. Using the $QIOW operation directly from Pascal on a per character basis provided the necessary support for the user program to communicate directly with the terminal.

Likewise the use of the asynchronous trap or AST to implement the Control D function was also a very elegant solution for this feature. The ability to provide support for the Control D function also increased the user friendliness of the system.
6.3 Observations On The Debugger

The following observations have been made regarding the development of the Debugger and its subsequent performance:

- The Debugger turned out to be much more complicated than initially anticipated. The Debugger design went through three major revisions before an acceptable set of commands and implementation techniques were identified. In retrospect, the Debugger took almost as much time to develop and bring to an operational state as the Simulator itself.

- The limited functionality of the Debugger is a result of the author's personal experience in teaching others how to use minicomputers and microcomputers. Many users are unaware of the extent of the services provided by an operating system or debugger during program development and they don't realize that their program is responsible for the initial values of various registers when their microcomputer and program are buried inside a product. The nature of a teaching tool is to encourage students to develop good habits and this philosophy influenced the assumptions implemented in the Debugger.
Likewise, the author chose not to support symbolic debugging to encourage the user to develop a better understanding of how the system operates. Symbolic debuggers are productivity aids and are as essential in program development as high level languages, but students need to learn what the symbolic debugger is doing for them and to them. If for example, the segment registers are not established correctly by the program under test, addressing problems can be difficult to isolate. If the user manually traces through the address computation the problems are usually more apparent than if a symbolic debugger is used. This is especially true when segment registers are used, since a single memory location can be accessed by any one of several addressing schemes and a variety of addressing modes.

6.4 Summary

The development of the Simulator and the Debugger represent a significant tool set for enhancing the process of learning the Intel 8086 instruction set.
7.1 Preface

During the development of the Simulator, a number of enhancements were suggested to broaden the scope of the project. Some of those enhancements have been implemented and many of them improve the user friendliness of the Simulator, the Debugger and the other support programs. As with any project however, improvements can be made forever but there is a point when the incremental cost of the improvement exceeds its benefit. While there are still valuable enhancements that can be made to this Simulator, the primary objective of developing a usable Simulator for the Intel 8086 microprocessor written in a modern high level language has been achieved.

There remains a list of improvements and enhancements which could expand the usefulness of the Simulator. They can be divided into four categories: enhancements for the Simulator, the Debugger, the support programs and those which would expand the scope of the Simulator to provide support for Intel library routines and operating system functions.
7.2 Simulator Enhancements

+ The Simulator was designed to support the Intel 8086 microprocessor, but the architecture of the Simulator was designed to permit limited expansion of the instruction set. A minimal amount of effort would be required to support the Intel 80186 processor with its expanded instruction set. There are an additional seventeen instructions for which code must be developed and some of them will require using a few of the "spare" slots in the Opcode Look Up Table, while others will require further resolution in the extended class. However, the architecture of the Intel 80286 and Intel 80386 processors would require a major rewrite of the Simulator and would best be developed separately, using this Simulator as a model.

+ As an enhancement of the teaching tool, the Simulator could be expanded to support the Intel 8087 math coprocessor. Such an expansion could be most useful, both to add new capability to the Simulator and as an exercise in developing VAX applications which must be coordinated as true coprocesses.
A better method could be developed to support input and output operations. This could take several phases, one being the integration of the creating, editing and viewing of maps and data as part of the Debugger.

A second phase could be to provide real hardware interfaces for the user to manipulate. This would require extreme care on a multiuser system and might not be cost effective. The real benefit of this phase would be the experience of encountering the unexpected when performing I/O. The current configuration does not provide any real opportunity to experience the impact of noise on a signal, or the response of a system when a circuit design is incorrect or improperly implemented.

It has also been suggested that the Simulator monitor, record and provide to the user the number of clock cycles that were required to execute the program. This was not originally included because of the magnitude of the Simulator project, but it could be a useful measure of program efficiency. It would however, require extensive modifications to almost every module in the Simulator and hence would be a project unto itself.
The Intel 8086 uses a bit in the Processor Status Word, the Trap Flag, to permit single step execution of the program. The Simulator does not support this flag, other than to permit its state to be modified. The function of single stepping through the program is provided by the Debugger and hence support for the Trap Flag was not considered necessary. To be an accurate representation of the processor, support for the Trap Flag should be provided and linked to the Debugger in a coherent fashion.

7.3 Debugger Enhancements

During the development of the Debugger, several problems were encountered which are minor inconveniences but represent flaws in the implementation of the Debugger. One of these problems occurs when a breakpoint is set on an instruction and the instruction is executed in single step mode. The Debugger does not give any indication that a breakpoint had been encountered. As a user, this notification could have been a reminder that other operations needed to be performed at this point in the program.
The user friendliness of the Debugger might be improved if the Debugger could check for the possibility that the user placed a breakpoint in the middle of a multi-byte instruction. This could be accomplished by scanning the addresses in the breakpoint table as each instruction is being executed. If a breakpoint was encountered in the middle of a multi-byte instruction, this could then cause an unexpected breakpoint interrupt and notification to the user of the abnormal situation. It would require a major revision in the way breakpoint information is stored and in the way memory is accessed to assemble all of the bytes needed to complete an instruction.

Presently, there is nothing to prevent the user from inadvertently placing two breakpoints at the same location and causing the Simulator to be unable to execute the original instruction on which the breakpoint had been set. A simple check of the breakpoint table could be made prior to adding the breakpoint to see if that address has already had a breakpoint set on it.

Improvements could be made to the logging of user commands in the Debugger. Presently, the user can create a file of the commands which are entered during the debugging session, but there is no process by which the user can automatically
recover from the log file. This feature would be particularly useful when accessing the host system remotely and communications are lost or in resuming a particularly long and complicated debugging session.

+ An additional feature that would benefit students and instructor alike is to be able to log the results of the Debugger commands. Presently, the user can examine a register or a memory location and the information is displayed on the user's terminal. Unless that is a hard copy terminal, or the terminal has a printer port and a printer attached, it is not possible to obtain a hard copy of the information for discussion. The log command of the Debugger could be expanded to write anything displayed at the terminal to a log file, or it could be implemented to separate the commands to one file and the results to another file so that the previous feature, recovery from a log file, would still work.

+ Although symbolic debugging is not supported and the author does not believe it should be, others may not be of the same opinion. An addition to the Debugger which may be deemed appropriate could be the addition of symbolic debugging.
The Debugger could also be expanded to include support for compound commands. Commands which permit loops of instructions to be executed or which support logical operations would significantly increase the power of the Debugger.

An additional feature that could be supported in the Debugger is to allow the user to control the radix of the information which is displayed. Presently, the radix information is "hard coded" into the Debugger and the base used for displaying information is what the author felt was most useful. Other users may have different needs and desire to dynamically alter the radix of the displayed information.

7.4 Support Program Enhancements

As previously indicated, improvements could be made to permit easier use of the I/O operations. An additional feature that should be considered is free form editing of input data rather than forcing the user to specify which byte is to be changed. The current approach is admittedly primitive, but it was necessary to provide a capability for
editing I/O Data, not in making everything elegant. There is substantial room for improvement in this area.

+ The likelihood of future users modifying the Debugger's keyword table is very high. There are substantial risks in making modifications to the keyword table and having a resultant table which is flawed. It is possible to add more robustness to the editing process and perform certain checks to see if the desired change conflicts with an existing entry or could be misinterpreted as a hexadecimal value before accepting the input. Also the identifier value is not specified in a very robust manner and unsupported values can be created which would result in confusion.

+ Likewise the Opcode Look Up Table maintenance program could be made more robust. The maintenance programmer modifying this table must be excruciatingly careful when making changes so that the Simulator works correctly. While changes in this table are much less likely, the expansion of the Simulator to support the Intel 80186 processor will result in a need to modify the Opcode Look Up Table. Any modifications to this table should be thoroughly understood before they are attempted.
The power of the Simulator could be greatly enhanced by including support for existing Intel library routines. This could be accomplished in two ways: the routines could be interpreted, rewritten in Pascal and assembler calls directed to the appropriate Pascal procedures, or the necessary library routines could be physically linked, located and then loaded with the program under test, thus causing the Simulator to execute the library routine. The first approach requires a complete and exact understanding of each supported library routine as implemented by Intel and some of these routines may be "trade secrets". Implementation at this level would also have to replicate any flaws in the original code. The second approach assumes that the library routines are self-contained, or at least that all of the pieces are included in the package that is loaded with the program under test. This would also require a much larger memory space and could significantly impact the Simulator response time.
Likewise, support for any operating system functions could be implemented in either of the two above mentioned approaches. Support for an operating system would facilitate the transition from a simulation environment to operation on the target hardware.

As a subset of these features, a selected peripheral or class of peripherals could be supported. It might be useful to support a time of day clock, a timer, disk I/O or a printer. The effort expended to implement these features needs to be evaluated carefully against the cost and availability of hardware to perform these functions on the target system.

7.6 Summary

There are a number of enhancements which would increase the usefulness of the Simulator. A careful and thorough evaluation of the impact of any change should be done before it is implemented to minimize the undesirable side effects.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>NAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>DAC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Output stored data to a digital to analog converter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>8255 IS USED TO GENERATE A WAVEFORM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>000B</td>
<td>7</td>
<td>DATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>DATA_PNTS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0B00H, 0B00H, 0940H, 09DBH, 0A75H, 0B04H, 0BBFH, 0C15H</td>
</tr>
<tr>
<td>0004</td>
<td>000B</td>
<td>9</td>
<td>DW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0CB6H, 0CF2H, 0D5AH, 0DACH, 0DEAH, 0E20H, 0E47H, 0E5EH</td>
</tr>
<tr>
<td>0006</td>
<td>000B</td>
<td>10</td>
<td>DW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0EB6H, 0E5EH, 0E47H, 0E20H, 0DEAH, 0DACH, 0D5AH, 0CF2H</td>
</tr>
<tr>
<td>0008</td>
<td>000B</td>
<td>11</td>
<td>DW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0CB6H, 0C15H, 0BBFH, 0B04H, 0A73H, 09DBH, 0940H, 0BA0H</td>
</tr>
<tr>
<td>000C</td>
<td>000B</td>
<td>12</td>
<td>DW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0B00H, 0760H, 06C0H, 0625H, 05BDH, 04FCH, 0471H, 03EBH</td>
</tr>
<tr>
<td>000E</td>
<td>000B</td>
<td>13</td>
<td>DW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>037AH, 030EH, 02A5H, 0254H, 0226H, 01E0H, 01B9H, 0102H</td>
</tr>
</tbody>
</table>

DAC Program Listing

Appendix A
Sample programs
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0056</td>
<td>5402</td>
<td>14</td>
<td>DW 017AH, 0102H, 0189H, 01E0H, 0226H, 0254H, 02A5H, 030EH</td>
</tr>
<tr>
<td>0058</td>
<td>2602</td>
<td>15</td>
<td>DW 037AH, 03EBH, 0471H, 04FCH, 05BDH, 0625H, 06COH, 0760H</td>
</tr>
<tr>
<td>005A</td>
<td>E901</td>
<td>16</td>
<td>CLOCK DW 49152</td>
</tr>
<tr>
<td>005C</td>
<td>B901</td>
<td>17</td>
<td>FREG DW 1</td>
</tr>
<tr>
<td>005E</td>
<td>0201</td>
<td>18</td>
<td>COUNTER DW 1 DUP (?)</td>
</tr>
<tr>
<td>0060</td>
<td>7A01</td>
<td>19</td>
<td>OVERHEAD DW 164</td>
</tr>
<tr>
<td>0062</td>
<td>0201</td>
<td>20</td>
<td>LOOP_STS DW 17</td>
</tr>
<tr>
<td>0064</td>
<td>B901</td>
<td>21</td>
<td>LEN_DATA DW 64</td>
</tr>
<tr>
<td>0066</td>
<td>E001</td>
<td>22</td>
<td>DATA ENDS</td>
</tr>
<tr>
<td>0068</td>
<td>2602</td>
<td>23</td>
<td>STACK SEGMENT PUBLIC</td>
</tr>
<tr>
<td>006A</td>
<td>5402</td>
<td>24</td>
<td>DW 10 DUP (?)</td>
</tr>
<tr>
<td>006C</td>
<td>A502</td>
<td>25</td>
<td>STK_TOP LABEL WORD</td>
</tr>
<tr>
<td>006E</td>
<td>0E03</td>
<td>26</td>
<td>STACK_TOP DW 0</td>
</tr>
<tr>
<td>0070</td>
<td>7A03</td>
<td>27</td>
<td>STACK ENDS</td>
</tr>
<tr>
<td>0072</td>
<td>EB03</td>
<td>28</td>
<td>ASSUME CS:DAC_CODE, DS:DATA, SS:STACK</td>
</tr>
<tr>
<td>0074</td>
<td>7104</td>
<td>29</td>
<td>DAC_CODE SEGMENT PUBLIC</td>
</tr>
<tr>
<td>0076</td>
<td>FC04</td>
<td>30</td>
<td>START: MDV AX, DATA  ; Set up DB to point to data segment</td>
</tr>
<tr>
<td>0078</td>
<td>BD05</td>
<td>31</td>
<td>MDV DS, AX</td>
</tr>
<tr>
<td>007A</td>
<td>2506</td>
<td>32</td>
<td>MDV AX, STACK  ; Set up BB to point to stack segment</td>
</tr>
<tr>
<td>007C</td>
<td>C006</td>
<td>33</td>
<td>MDV SS, AX</td>
</tr>
<tr>
<td>007E</td>
<td>6007</td>
<td>34</td>
<td>MDV SP, offset STK_TOP  ; Set Stack Pointer to top of stack</td>
</tr>
<tr>
<td>0080</td>
<td>00C0</td>
<td>35</td>
<td>MDV SP, 14H</td>
</tr>
<tr>
<td>0082</td>
<td>0100</td>
<td>36</td>
<td>MDV AX, 9990H  ; Initialize I/O port at OFFEH and OFFFH</td>
</tr>
<tr>
<td>0084</td>
<td>41</td>
<td>37</td>
<td>MDV DX, OFFEH  ; P1A, P2A as inputs</td>
</tr>
<tr>
<td>0086</td>
<td>A400</td>
<td>38</td>
<td>MDV PIB, P2B as outputs</td>
</tr>
<tr>
<td>0088</td>
<td>1100</td>
<td>39</td>
<td>MDV PIC as input</td>
</tr>
<tr>
<td>008A</td>
<td>4000</td>
<td>40</td>
<td>MDV P2C as output</td>
</tr>
<tr>
<td>008C</td>
<td>10</td>
<td>41</td>
<td>OUT DX, AX</td>
</tr>
<tr>
<td>008E</td>
<td>BB----</td>
<td>42</td>
<td>OUT DX, AX</td>
</tr>
<tr>
<td>0090</td>
<td>33</td>
<td>43</td>
<td>MDV DX, 0</td>
</tr>
<tr>
<td>0092</td>
<td>0B0000</td>
<td>44</td>
<td>MDV AX, CLOCK</td>
</tr>
<tr>
<td>0094</td>
<td>A1000</td>
<td>45</td>
<td>DIV FREG  ; Determine clock frequency</td>
</tr>
</tbody>
</table>
LOC    OBJ    LINE    SOURCE

0021  2B06B600  R   46    SUB   AX, OVERHEAD  ; Remove overhead
0025  BA0000   47    MOV   DX.0
0028  F736B800  R   48    DIV   LOOP_STS  ; 17/5
002C  A3B400   49    MOV   COUNTER,AX  ; Save it in counter
002F  50    MORE:
002F  BBOEBA00  R   51    MOV   CX, LEN_DATA  ; Get number of data points
0033  BD360000  R   52    LEA   SI, DATA_PNTS  ; Point to data
0037  54    WAVE_GEN:
0037  BB04   55    MDV   AX, [BX]  ; Get data point
0039  46    56    INC   BI  ; Point to next one
003A  46    57    INC   SI  ;
003B  51    58    PUSH   CX  ; Save CX
003C  EB8000   59    CALL   DELAY  ;
003F  B104   60    MDV   CL,4  ; Shift the data by 4 bits
0041  D3E0   61    BHL   AX, CL  ;
0043  B0FAFF  62    MDV   DX, OFFFAH  ; Point to the I/O port
0046  EF   63    OUT   DX, AX  ; Output the data
0047  59    68    POP   CX  ; Restore CX
004B  E2ED   69    LOOP   WAVE_GEN  ;
004C  F4    70    MHT
004D  EBE2   71    JMP   MORE
004D  9C    72    DELAY:
004D  9C    73    PUSHF  ; Save registers
004E  50    74    PUSH   AX
004F  51    75    PUSH   CX
0050  BB0EBA00  R   76    MDV   CX, COUNTER  ; Initialize delay value
0054  77    TIME_DLY:
0054  E2FE   78    LOOP   TIME_DLY  ; Delay
0056  59    79    POP   CX  ; Restore registers
0057  59    80    POP   AX
0058  9D    81    POPF
0059  C3    82    RET
-----  83    DAC_CODE ENDS
0084  85    END   START

ASSEMBLY COMPLETE, NO ERRORS FOUND
8.2 DAC Debug Session

wac> em86

Intel 8086 Simulator
Version 1.4

Will this program access any IO ports [Y/N] ? y
Enter name of file containing IO port map: wacdac.map
Enter name of file containing IO port data: wacdac.data
* loa = wacdac.dat

* reg
AX = 0000 BX = 0000 CX = 0000 DX = 0000
CS = 002B SS = 0000 DS = 0000 ES = 0000
IP = 0000 SP = 0000 SI = 0000 DI = 0000
RF = 0000 BP = 0000

* byte cs:0 len a
002B:0000 00
002B:0001 00
002B:0002 00
002B:0003 00
002B:0004 00
002B:0005 00
002B:0006 00
002B:0007 00
002B:0008 00
002B:0009 00

* wport fffe
Port address: FFFE / FFFF: 0000
* bp0=17
* g
Breakpoint BP0 encountered at: 002B:0017

* reg
AX = 9990 BX = 0000 CX = 0000 DX = FFFE
CS = 002B SS = 0029 DS = 0020 ES = 0000
IP = 0017 SP = 0014 SI = 0000 DI = 0000
RF = 0000 BP = 0000

* wport fffe
Port address: FFFE / FFFF: 9990
* bp1=2f
* all
Breakpoint 0 = 002B:0017
Breakpoint 1 = 002B:002F
* r
Breakpoint BP1 encountered at: 002B:002F

* reg
AX = 0B41 BX = 0000 CX = 0000 DX = 0000
CS = 002B SS = 0029 DS = 0020 ES = 000C
IP = 002F SP = 0014 SI = 0000 DI = 0000
RF = 0094 BP = 0000
* s
This instruction at 002B:002F
* cx
CX = 0040
* s
This instruction at 002B:0033
* si
SI = 0000
* ds
DS = 0020
* word ds:si
0020:0000 0800
* s
This instruction at 002B:0037
* ax
AX = 0800
* s
This instruction at 002B:0039
* s
This instruction at 002B:003A
* si
SI = 0002
* s
This instruction at 002B:003B
* s
This instruction at 002B:003C
* stack 4
  0029:0010 003F
  0029:0012 0040
  0029:0014 0000
  0029:0016 0000
* cx
CX = 0040
* sp
SP = 0010
* bp2=54
* all
Breakpoint 0 = 002B:0017
Breakpoint 1 = 002B:002F
Breakpoint 2 = 002B:0054
* r
Breakpoint BF2 encountered at: 002B:0054
* sp
SP = 000A
* stack 6
  0029:0000A 0040
  0029:0000C 0B00
  0029:0000E 0000
  0029:00010 003F
  0029:00012 0040
  0029:00014 0000
* bp2=
* all
Breakpoint 0 = 002B:0017
Breakpoint 1 = 002B:002F
* r

Aborted by a CtrlD encountered at: 002B:0054
* reg
  AX = 0B8F BX = 0000 CX = 0730 DX = 0FFA
  CS = 002B SS = 0029 DS = 0020 ES = 0000
  IP = 0054 SP = 0000 SI = 0002 DI = 0000
  RF = 0000 BP = 0000
* stack 6
  0029:0000A 0040
  0029:0000C 0B00
  0029:0000E 0000
  0029:00010 003F
  0029:00012 0040
  0029:00014 0000
* r

Aborted by a CtrlD encountered at: 002B:0054
* reg
  AX = 0BBF BX = 0000 CX = 0730 DX = 0FFA
  CS = 002B SS = 0029 DS = 0020 ES = 0000
  IP = 0054 SP = 0000 SI = 0002 DI = 0000
  RF = 0000 BP = 0000
* stack 6
  0029:0000A 003A
  0029:0000C 0BBF
  0029:0000E 0000
  0029:00010 003F
  0029:00012 003A
  0029:00014 0000
* word B4
  002B:00B4 0000
* word ds:B4
  0020:00B4 0B41
* word ds:B4 = 35
* r
Breakpoint BP2 encountered at: 002B:0050
Aborted by a CtrlD encountered at: 002B:0050
* all
Breakpoint 0 = 002B:0017
Breakpoint 1 = 002B:002F
Breakpoint 2 = 002B:0050
* bp2=
* all
Breakpoint 0 = 002B:0017
Breakpoint 1 = 002B:002F
* cx:
CX = 003B
* s
This instruction at 002B:0050
* cx:
CX = 0035
* r

Aborted by a CtrlD encountered at: 002B:0054
* cx:
CX = 0029
* stack 6
   0029:000A 000E
   0029:000C 0226
   0029:000E 0004
   0029:0010 003F
   0029:0012 000C
   0029:0014 0000
* wport ffffa
Fort address: FFFA / FFFB: 1E00
* r
Halt encountered at: 002B:004A
* wport ffffa
Fort address: FFFA / FFFB: 7600
* reg
AX = 7600 BX = 0000 CX = 0000 DX = FFFA
CS = 002B SS = 0029 DS = 0020 ES = 0000
IP = 004B SP = 0014 SI = 00B0 DI = 0000
RF = 0014 BP = 0000
* ex:
wac> f
8.3 DAC IOMap86 Session

wac> map86

Intel 8086 Simulator - I/O Map Generation Utility
Version 1.2

Generate or Modify I/O Map

Enter desired operation:
Create (C), Read (R), Modify (M), Display (D)
Print (P), Write (W), Quit (Q), Exit (E):

Enter name of file containing map: wacdac.map

Enter desired operation:
Create (C), Read (R), Modify (M), Display (D)
Print (P), Write (W), Quit (Q), Exit (E):

Description of I/O Port Map

<table>
<thead>
<tr>
<th>Port Index</th>
<th>Port Address</th>
<th>Port Width</th>
<th>Aux Upper Index</th>
<th>Aux Data Index</th>
<th>Input I/O Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FFFE</td>
<td>16</td>
<td>L</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>FFFF</td>
<td>16</td>
<td>U</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>FFFA</td>
<td>16</td>
<td>L</td>
<td>4</td>
<td>65</td>
</tr>
<tr>
<td>4</td>
<td>FFFB</td>
<td>16</td>
<td>U</td>
<td>3</td>
<td>65</td>
</tr>
<tr>
<td>5</td>
<td>FFFF</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>FFFF</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>FFFF</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>FFFF</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>1</td>
</tr>
</tbody>
</table>

Enter desired operation:
Create (C), Read (R), Modify (M), Display (D)
Print (P), Write (W), Quit (Q), Exit (E): ex
8.4 DAC IOData86 Session

wac> data86

Intel 8086 Simulator - Review I/O Data Utility
Version 1.2

Enter name of file containing map: wacdac.map

Enter desired operation to perform on data file:
Create file (C), Read file (R), Modify data (M), Input data (I),
Display data (D), Print data (P), Write (W), Quit (Q), Exit (E): r

Enter name of file containing data: wacdac.data

Enter desired operation to perform on data file:
Create file (C), Read file (R), Modify data (M), Input data (I),
Display data (D), Print data (P), Write (W), Quit (Q), Exit (E): d

Enter port address : fffe
Index: 1
Address: FFFE and FFFF, Width: 16 bits, Entries Accessed: 1,
Number of Entries: 0, Direction: Output
Data: 9990

Enter desired operation to perform on data file:
Create file (C), Read file (R), Modify data (M), Input data (I),
Display data (D), Print data (P), Write (W), Quit (Q), Exit (E): d

Enter port address : fffa
Index: 3
Address: FFFA and FFFB, Width: 16 bits, Entries Accessed: 64,
Number of Entries: 0, Direction: Output
Data: B000 BA00 9400 9DB0 A730 B040 BBFO C150 CB60 CF20 D5A0 DAC0 DEA0 E200
E470 E5E0 EB60 E5E0 E470 E200 DEA0 DAC0 D5A0 CB60 C150 BBFO B040
A730 9DB0 9400 BA00 B000 7600 6200 6250 58D0 4FC0 4710 3EB0 37A0 30E0
2A50 2540 2260 1E00 1890 1020 17A0 1020 1890 1E00 2260 2540 2A50 30E0
37A0 3EB0 4710 4FC0 58D0 6250 6C00 7600

Enter desired operation to perform on data file:
Create file (C), Read file (R), Modify data (M), Input data (I),
Display data (D), Print data (P), Write (W), Quit (Q), Exit (E): ex
# 8.5 DAC I/O Map And Data

**wac> list wacdacmap.list**

**Description of 10 Port Map**

<table>
<thead>
<tr>
<th>Port Index</th>
<th>Port Address</th>
<th>Port Width</th>
<th>Upper Limit</th>
<th>Aux Index</th>
<th>Data Index</th>
<th>Input I/O State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FFFE</td>
<td>16</td>
<td>L</td>
<td>2</td>
<td>1</td>
<td>1 Output</td>
</tr>
<tr>
<td>2</td>
<td>FFFF</td>
<td>16</td>
<td>U</td>
<td>1</td>
<td>1</td>
<td>1 Output</td>
</tr>
<tr>
<td>3</td>
<td>FFFA</td>
<td>16</td>
<td>L</td>
<td>4</td>
<td>1</td>
<td>1 Output</td>
</tr>
<tr>
<td>4</td>
<td>FFFB</td>
<td>16</td>
<td>U</td>
<td>3</td>
<td>1</td>
<td>1 Output</td>
</tr>
<tr>
<td>5</td>
<td>FFFF</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>1</td>
<td>1 Input</td>
</tr>
<tr>
<td>6</td>
<td>FFFF</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>1</td>
<td>1 Input</td>
</tr>
<tr>
<td>7</td>
<td>FFFF</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>1</td>
<td>1 Input</td>
</tr>
<tr>
<td>8</td>
<td>FFFF</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>1</td>
<td>1 Input</td>
</tr>
</tbody>
</table>

**wac> list wacdacadata.list**

**Index: 1**

Address: FFFE and FFFF, Width: 16 bits, Entries Accessed: 1,
Number of Entries: 0, Direction: Output
Data: 9990

**Index: 3**

Address: FFFA and FFFB, Width: 16 bits, Entries Accessed: 64,
Number of Entries: 0, Direction: Output
Data: 8000 BA00 9400 9DB0 A730 B040 BBFO C150 CB60 CF20 D5A0 DAC0 DEA0 E200 E470 E5E0 EB60 E5E0 E470 E200 DEA0 DAC0 D5A0 CF20 CB60 C150 BBFO B040 A730 9DB0 9400 BA00 B000 7600 6C00 6250 5BD0 4FC0 4710 3EB0 37A0 30E0 2A50 2540 2260 1E00 1890 1020 17A0 1020 1B90 1E00 2260 2540 2A50 30E0 37A0 3EB0 4710 4FC0 5BD0 6250 6C00 7600
ADC Program Listing

LOC OBJ LINE SOURCE

0000 456746572206D

Prompt db 'Enter maximum number of iterations: '

0024 2400 21 PromptLength dw 36

0026 0A 22 TEN db 10

0000 (10 28 Stack dw 10 dup(?)

0014 29 StackTop label word

0024 2400 30 Stack ends

8.6 ADC Program Listing
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0B-----</td>
<td>37</td>
<td>START:</td>
</tr>
<tr>
<td>0000</td>
<td>BB-----</td>
<td>38</td>
<td>MOV AX, DataSegment</td>
</tr>
<tr>
<td>0003</td>
<td>B0DB</td>
<td>39</td>
<td>MOV DS, AX</td>
</tr>
<tr>
<td>0005</td>
<td>BB-----</td>
<td>40</td>
<td>MOV AX, Stack</td>
</tr>
<tr>
<td>000B</td>
<td>BED0</td>
<td>41</td>
<td>MOV SS, AX</td>
</tr>
<tr>
<td>000A</td>
<td>BC1400</td>
<td>42</td>
<td>MOV SP, offset StackTop</td>
</tr>
<tr>
<td>000D</td>
<td>BBDABA</td>
<td>43</td>
<td>MOV AX, BABAH</td>
</tr>
<tr>
<td>0010</td>
<td>BAFEFF</td>
<td>44</td>
<td>MOV DX, OFFFEH</td>
</tr>
<tr>
<td>0013</td>
<td>EF</td>
<td>49</td>
<td>OUT DX, AX</td>
</tr>
<tr>
<td>0014</td>
<td>BD1E0000</td>
<td>50</td>
<td>Out DX, AX</td>
</tr>
<tr>
<td>0019</td>
<td>B0BE2400</td>
<td>51</td>
<td>OUT CX, PromptLength</td>
</tr>
<tr>
<td>001C</td>
<td>BOOD</td>
<td>52</td>
<td>MOV AL, CR</td>
</tr>
<tr>
<td>001E</td>
<td>E6FF</td>
<td>53</td>
<td>MOV AL, LF</td>
</tr>
<tr>
<td>0020</td>
<td>B00A</td>
<td>54</td>
<td>OUT CRTOut, AL</td>
</tr>
<tr>
<td>0022</td>
<td>E6FF</td>
<td>55</td>
<td>OUT CRTOut, AL</td>
</tr>
<tr>
<td>0024</td>
<td>B007</td>
<td>56</td>
<td>GET a character</td>
</tr>
<tr>
<td>0026</td>
<td>E6FF</td>
<td>57</td>
<td>OUT CRTOut, AL</td>
</tr>
<tr>
<td>0027</td>
<td>43</td>
<td>58</td>
<td>INC BX</td>
</tr>
<tr>
<td>0029</td>
<td>E2F9</td>
<td>59</td>
<td>LOOP IssuePrompt</td>
</tr>
<tr>
<td>002B</td>
<td>E4FF</td>
<td>60</td>
<td>MOV AL, [BX]</td>
</tr>
<tr>
<td>002D</td>
<td>2C30</td>
<td>61</td>
<td>MOV AL, KBDIn</td>
</tr>
<tr>
<td>002F</td>
<td>F6262600</td>
<td>62</td>
<td>SUB AL, ASCIIBase</td>
</tr>
<tr>
<td>0033</td>
<td>BADB</td>
<td>63</td>
<td>MUL TEN</td>
</tr>
<tr>
<td>0035</td>
<td>E4FF</td>
<td>64</td>
<td>MOV BL, AL</td>
</tr>
<tr>
<td>0037</td>
<td>2C30</td>
<td>65</td>
<td>IN AL, KBDIn</td>
</tr>
<tr>
<td>0039</td>
<td>02C3</td>
<td>66</td>
<td>SUB AL, ASCIIBase</td>
</tr>
<tr>
<td>003B</td>
<td>32E4</td>
<td>67</td>
<td>ADD AL, BL</td>
</tr>
<tr>
<td>003D</td>
<td>BDC8</td>
<td>68</td>
<td>XOR AH, AH</td>
</tr>
<tr>
<td>003F</td>
<td>BOOD</td>
<td>69</td>
<td>MOV AL, CR</td>
</tr>
<tr>
<td>0041</td>
<td>E6FF</td>
<td>70</td>
<td>OUT CRTOut, AL</td>
</tr>
<tr>
<td>0043</td>
<td>B00A</td>
<td>71</td>
<td>MOV AL, LF</td>
</tr>
<tr>
<td>0045</td>
<td>E6FF</td>
<td>72</td>
<td>OUT CRTOut, AL</td>
</tr>
<tr>
<td>0047</td>
<td>51</td>
<td>73</td>
<td>PUSH CX</td>
</tr>
<tr>
<td>004B</td>
<td>BAFEFF</td>
<td>74</td>
<td>MOV DX, OFFFH</td>
</tr>
<tr>
<td>004B</td>
<td>B001</td>
<td>75</td>
<td>MOV AL, 01H</td>
</tr>
<tr>
<td>004D</td>
<td>EE</td>
<td>76</td>
<td>OUT DX, AL</td>
</tr>
<tr>
<td>004E</td>
<td>F6D0</td>
<td>77</td>
<td>NOT AL</td>
</tr>
<tr>
<td>004F</td>
<td>EE</td>
<td>78</td>
<td>OUT DX, AL</td>
</tr>
<tr>
<td>0051</td>
<td>B90500</td>
<td>79</td>
<td>MOV CX, 5</td>
</tr>
<tr>
<td>0054</td>
<td>89</td>
<td>80</td>
<td>PULSE:</td>
</tr>
<tr>
<td>0054</td>
<td>E2FE</td>
<td>81</td>
<td>LOOP PULSE</td>
</tr>
<tr>
<td>LOC</td>
<td>OBJ</td>
<td>LINE</td>
<td>SOURCE</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>0056</td>
<td>F6D0</td>
<td>92</td>
<td>NOT AL</td>
</tr>
<tr>
<td>0058</td>
<td>EE</td>
<td>93</td>
<td>OUT DX,AL</td>
</tr>
<tr>
<td>0059</td>
<td>BAFDF</td>
<td>95</td>
<td>MOV DX,OFFFH</td>
</tr>
<tr>
<td>005C</td>
<td>EC</td>
<td>96</td>
<td>AGAIN: IN AL,DX</td>
</tr>
<tr>
<td>005D</td>
<td>0A0C</td>
<td>97</td>
<td>OR AL,AL</td>
</tr>
<tr>
<td>005F</td>
<td>79FB</td>
<td>98</td>
<td>JNS AGAIN</td>
</tr>
<tr>
<td>0061</td>
<td>BAFAFF</td>
<td>100</td>
<td>MOV DX,OFFFAH</td>
</tr>
<tr>
<td>0064</td>
<td>ED</td>
<td>101</td>
<td>IN AX,DX</td>
</tr>
<tr>
<td>0065</td>
<td>25FOF</td>
<td>102</td>
<td>AND AX,OFFFH</td>
</tr>
<tr>
<td>006B</td>
<td>050008</td>
<td>103</td>
<td>ADD AX,0800H</td>
</tr>
<tr>
<td>006B</td>
<td>DICB</td>
<td>104</td>
<td>ROR AX,1</td>
</tr>
<tr>
<td>006D</td>
<td>25FOF</td>
<td>105</td>
<td>AND AX,OFFFH</td>
</tr>
<tr>
<td>0070</td>
<td>BAFBFF</td>
<td>106</td>
<td>MOV DX,OFFFBH</td>
</tr>
<tr>
<td>0073</td>
<td>EF</td>
<td>107</td>
<td>OUT DX,AX</td>
</tr>
<tr>
<td>109</td>
<td></td>
<td>110</td>
<td>MAY CALL THE DELAY ROUTINE HERE FOR A CERTAIN SAMPLING TIME,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111</td>
<td>with the delay in the CX Register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>0074</td>
<td>59</td>
<td>113</td>
<td>POP CX</td>
</tr>
<tr>
<td>0075</td>
<td>E200</td>
<td>114</td>
<td>LOOP MORE</td>
</tr>
<tr>
<td>0077</td>
<td>F4</td>
<td>115</td>
<td>HLT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>116</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>117</td>
<td>( \text{DELAY: LOOP DELAY} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>118</td>
<td>RET</td>
</tr>
<tr>
<td></td>
<td></td>
<td>119</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>120</td>
<td>ADDCODE ENDS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>121</td>
<td>END START</td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETE, NO ERRORS FOUND

wac> ff
8.7 ADC Debug Session

Intel 8086 Simulator
Version 1.4

Will this program access any I/O ports [Y/N]? y
Enter name of file containing I/O port map: wacadc.map
Enter name of file containing I/O port data: wacadc.data
* loa=wacadc.dat
* bp0=1c
* bp1=47
* g
Breakpoint B0 encountered at: 0020:001c
* reg
AX = 0004 BX = 0024 CX = 0024 DX = FFFE
CS = 0020 SS = 0028 DS = 0029 ES = 0000
IP = 001c SP = 0014 SI = 0000 DI = 0000
RF = 0000 BP = 0000
* wport fffe
Port address: FFFE / FFFF: BABA
* word ds:bx len 4
0029:0004 6E45
0029:0006 6574
0029:0008 2072
0029:000a 616d
* r
Enter maximum number of iterations: 12
Breakpoint B1 encountered at: 0020:0047
* reg
AX = 000A BX = 000A CX = 000c DX = FFFE
CS = 0020 SS = 0028 DS = 0029 ES = 0000
IP = 0047 SP = 0014 SI = 0000 DI = 0000
RF = 0044 BP = 0000
* stack 2
0028:0004 6E45
0028:0006 6574
* s
This instruction at 0020:0047
* stack 3
0028:0012 000c
0028:0014 6E45
0028:0016 6574
* sp
SP = 0012
* a1
Breakpoint 0 = 0020:001c
Breakpoint 1 = 0020:0047
* bp1=
* s
This instruction at 0020:0048
* s
This instruction at 0020:004B
* s
This instruction at 0020:004D
* al
AL = 01
* dx
DX = FFFC
* port fffc
Port address: FFFC: 01
* s
This instruction at 0020:004E
* al
AL = FE
* port fffc
Port address: FFFC: 01
* s
This instruction at 0020:0050
* port fffc
Port address: FFFC: FE
* e
This instruction at 0020:0051
* cx
DX = 0005
* e
This instruction at 0020:0054
* e
This instruction at 0020:0054
* s
This instruction at 0020:0054
* e
This instruction at 0020:0054
* s
This instruction at 0020:0054
* s
This instruction at 0020:0054
* s
This instruction at 0020:0054
* s
This instruction at 0020:0054
* s
This instruction at 0020:0054
* s
This instruction at 0020:0054
* port fffc
Port address: FFFC: FE
* s
This instruction at 0020:0058
* al
AL = 01
* port fffc
Port address: FFFC: 01
* bp2=5f
* r
Breakpoint BP2 encountered at: 0020:005F
* al
AL = 40
* sfl
SFL = 0
* port fffd
Port address: FFFD: 40
* r
Breakpoint BP2 encountered at: 0020:005F
* sfl
SFL = 0
* port fffd
Port address: FFFD: CO
* s
This instruction at 0020:005F
* s
This instruction at 0020:005C
* al
AL = CO
* sfl
SFL = 0
* s
This instruction at 0020:005D
* al
AL = CO
* sfl
SFL = 1
* s
This instruction at 0020:005F
* s
This instruction at 0020:0061
* s
This instruction at 0020:0064
* s
This instruction at 0020:0065
* ax
AX = 0A54
* s
This instruction at 0020:0068
* ax
AX = 1254
* s
This instruction at 0020:006B
* ax
AX = 092A
* wprot fff8
Bad keyword class: "N"
Bad keyword class: "N"
* wport fff8
Port address: FFF8 / FFF9: 0000
This instruction at 0020:006D
This instruction at 0020:0070
* wport fffB
Port address: FFF8 / FFF9: 0000
This instruction at 0020:0073
* wport fffB
Port address: FFF8 / FFF9: 092A
* ~r1EC/
This instruction at 0020:0074
* cx
CX = 000C
This instruction at 0020:0075
This instruction at 0020:0047
* all
Breakpoint 0 = 0020:001C
Breakpoint 2 = 0020:005F
* all=
* bpf=75
* r
Breakpoint BPF encountered at: 0020:0075
* cx
CX = 000B
* wport fffB
Port address: FFF8 / FFF9: 05BB
* r
Breakpoint BPF encountered at: 0020:0075
* cx
CX = 000A
* wport fffB
Port address: FFF8 / FFF9: 0A06
* r
Breakpoint BPF encountered at: 0020:0075
* cx
CX = 0009
* wport fffB
Port address: FFF8 / FFF9: 0BA1
* r
Breakpoint BPF encountered at: 0020:0075
* cx
CX = 0008
* wport fffB
Port address: FFF8 / FFF9: 0752
* r
Breakpoint BPF encountered at: 0020:0075
* cx
 CX = 0007
 * wport fff8
 Port address: FFF8 / FFF9: 0B54
 * r
Breakpoint BPF encountered at: 0020:0075
* cx
 CX = 0006
 * wport fff8
 Port address: FFF8 / FFF9: 05FE
 * r
Breakpoint BPF encountered at: 0020:0075
* cx
 CX = 0005
 * wport fff8
 Port address: FFF8 / FFF9: 092B
 * r
Breakpoint BPF encountered at: 0020:0075
* cx
 CX = 0004
 * wport fff8
 Port address: FFF8 / FFF9: 0A51
 * r
Breakpoint BPF encountered at: 0020:0075
* cx
 CX = 0003
 * wport fff8
 Port address: FFF8 / FFF9: 0412
 * r
Breakpoint BPF encountered at: 0020:0075
* cx
 CX = 0002
 * wport fff8
 Port address: FFF8 / FFF9: 0BAB
 * r
Breakpoint BPF encountered at: 0020:0075
* cx
 CX = 0001
 * wport fff8
 Port address: FFF8 / FFF9: 0962
 * cx
 CX = 0001
 * s
This instruction at 0020:0075
 * s
This instruction at 0020:0077
Halt encountered at: 0020:0077
 * e
Bad keyword class: "N"
 * ex

- 114 -
8.8 ADC IOMap86 Session

wac> map86

Intel 8086 Simulator - I/O Map Generation Utility
Version 1.2

Generate or Modify I/O Map

Enter desired operation:
Create (C), Read (R), Modify (M), Display (D)
Print (P), Write (W), Quit (D), Exit (E): r

Enter name of file containing map: wacadc.map

Enter desired operation:
Create (C), Read (R), Modify (M), Display (D)
Print (P), Write (W), Quit (D), Exit (E): d

Description of I/O Port Map

<table>
<thead>
<tr>
<th>Port Index</th>
<th>Port Address</th>
<th>Port Width</th>
<th>Port Lower</th>
<th>Aux Index</th>
<th>Data Index</th>
<th>Input I/O</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FFFE</td>
<td>16</td>
<td>L</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>Output</td>
</tr>
<tr>
<td>2</td>
<td>FFFF</td>
<td>16</td>
<td>U</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>Output</td>
</tr>
<tr>
<td>3</td>
<td>FFFC</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>37</td>
<td>1</td>
<td>Output</td>
</tr>
<tr>
<td>4</td>
<td>FFFD</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>32</td>
<td>46</td>
<td>Input</td>
</tr>
<tr>
<td>5</td>
<td>FFFA</td>
<td>16</td>
<td>L</td>
<td>6</td>
<td>13</td>
<td>31</td>
<td>Input</td>
</tr>
<tr>
<td>6</td>
<td>FFFB</td>
<td>16</td>
<td>U</td>
<td>5</td>
<td>13</td>
<td>31</td>
<td>Input</td>
</tr>
<tr>
<td>7</td>
<td>FFFB</td>
<td>16</td>
<td>L</td>
<td>8</td>
<td>13</td>
<td>1</td>
<td>Output</td>
</tr>
<tr>
<td>8</td>
<td>FFF9</td>
<td>16</td>
<td>U</td>
<td>7</td>
<td>13</td>
<td>1</td>
<td>Output</td>
</tr>
</tbody>
</table>

Enter desired operation:
Create (C), Read (R), Modify (M), Display (D)
Print (P), Write (W), Quit (D), Exit (E): e
8.9 ADC IOData86 Session

wac> data86

Intel 8086 Simulator - Review I/O Data Utility
Version 1.2

Enter name of file containing map: wacadc.map

Enter desired operation to perform on data file:
Create file (C), Read file (R), Modify data (M), Input data (I),
Display data (D), Print data (P), Write (W), Quit (Q), Exit (E):

Enter name of file containing data: wacadc.data

Enter desired operation to perform on data file:
Create file (C), Read file (R), Modify data (M), Input data (I),
Display data (D), Print data (P), Write (W), Quit (Q), Exit (E):

Enter port address: fffe
Index: 1
Address: FFFE and FFFF, Width: 16 bits, Entries Accessed: 1,
Number of Entries: 0, Direction: Output
Data: BABA

Enter desired operation to perform on data file:
Create file (C), Read file (R), Modify data (M), Input data (I),
Display data (D), Print data (P), Write (W), Quit (Q), Exit (E):

Enter port address: fffe
Index: 3
Address: FFFC, Width: 8 bits, Entries Accessed: 36,
Number of Entries: 0, Direction: Output
Data: 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01

- 116 -
Enter port address: fffd

Index: 4
Address: FFFD, Width: 8 bits, Entries Accessed: 31,
Number of Entries: 45, Direction: Input
Data: 40 40 40 40 40 C0 40 40 C0 40 40 C0 40 40 C0 40 40 40 40 C0
     40 40 40 40 40 C0 40 40 C0 40 40 C0 40 40 C0 40 40 C0 40 40 C0

Enter desired operation to perform on data file:
Create file (C), Read file (R), Modify data (M), Input data (I),
Display data (D), Print data (P), Write (W), Quit (Q), Exit (E): d

Enter port address: fffa
Index: 5
Address: FFFA and FFFB, Width: 16 bits, Entries Accessed: 12,
Number of Entries: 30, Direction: Input
Data: OA54 A370 5C0D EF42 06A5 9EAB 03FC 8A51 7CA3 5024 3957 0AC5 19AF 9D24
     0ACD EB3F 9AFC 03E7 9BA2 37AE F023 D5CA 9A58 C247 349A 1B3C 32A0 862C
     A034 2B46

Enter desired operation to perform on data file:
Create file (C), Read file (R), Modify data (M), Input data (I),
Display data (D), Print data (P), Write (W), Quit (Q), Exit (E): d

Enter port address: fff8
Index: 7
Address: FFF8 and FFF9, Width: 16 bits, Entries Accessed: 12,
Number of Entries: 0, Direction: Output
Data: 092A 05B8 0A06 0BA1 0752 0B54 05FE 092B 0A51 0412 08AB 0962

Enter desired operation to perform on data file:
Create file (C), Read file (R), Modify data (M), Input data (I),
Display data (D), Print data (P), Write (W), Quit (Q), Exit (E): e
8.10 ADC I/O Map And Data

```plaintext
wac> list wacadcmapstart.list
Description of IO Port Map

<table>
<thead>
<tr>
<th>Port Index</th>
<th>Port Address</th>
<th>Port Width</th>
<th>Upper</th>
<th>Aux Index</th>
<th>Data Index</th>
<th>Input I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FFFE</td>
<td>16</td>
<td>L</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>FFFF</td>
<td>16</td>
<td>U</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>FFFC</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>FFFD</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>1</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>FFFA</td>
<td>16</td>
<td>L</td>
<td>6</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>FFFB</td>
<td>16</td>
<td>U</td>
<td>5</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>7</td>
<td>FFFB</td>
<td>16</td>
<td>L</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>FFF9</td>
<td>16</td>
<td>U</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

wac> list wacadcdatastart.list

Index: 1
Address: FFFE and FFFF, Width: 16 bits, Entries Accessed: 0,
Number of Entries: 0, Direction: Output

Index: 3
Address: FFFC, Width: 8 bits, Entries Accessed: 0,
Number of Entries: 0, Direction: Output

Index: 4
Address: FFFD, Width: 8 bits, Entries Accessed: 0,
Number of Entries: 45, Direction: Input
Data: 40 40 C0 40 40 C0 C0 C0 40 40 C0 40 40 C0 C0 C0 40 40 C0 40 40 C0 C0 C0 40 40 C0 40 40 C0 C0 C0 40 40 C0 40 40 C0 C0 C0 C0

Index: 5
Address: FFFA and FFFB, Width: 16 bits, Entries Accessed: 0,
Number of Entries: 30, Direction: Input
Data: 0A54 A370 5C0D EF42 06A5 9EAB 03FC BA51 7CA3 5024 3957 0AC5 19AF 9D24 0ACD E83F 9AFC 03E7 9B2A 37AE F023 D5CA 9A3B C247 349A 1B3C 32A0 862C A034 2846

Index: 7
Address: FFFB and FFF9, Width: 16 bits, Entries Accessed: 0,
Number of Entries: 0, Direction: Output

wac> ff
```
### Description of IO Port Map

<table>
<thead>
<tr>
<th>Port Index</th>
<th>Port Address</th>
<th>Port Width</th>
<th>Port Upper</th>
<th>Port Lower</th>
<th>Aux Upper</th>
<th>Aux Lower</th>
<th>Data Upper</th>
<th>Data Lower</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FFFE</td>
<td>16</td>
<td>L</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>Output</td>
</tr>
<tr>
<td>2</td>
<td>FFFF</td>
<td>16</td>
<td>U</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>Output</td>
</tr>
<tr>
<td>3</td>
<td>FFFC</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>37</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>Output</td>
</tr>
<tr>
<td>4</td>
<td>FFFD</td>
<td>8</td>
<td>L</td>
<td>-1</td>
<td>32</td>
<td>46</td>
<td></td>
<td></td>
<td></td>
<td>Input</td>
</tr>
<tr>
<td>5</td>
<td>FFFA</td>
<td>16</td>
<td>L</td>
<td>6</td>
<td>13</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td>Input</td>
</tr>
<tr>
<td>6</td>
<td>FFFB</td>
<td>16</td>
<td>U</td>
<td>5</td>
<td>13</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td>Input</td>
</tr>
<tr>
<td>7</td>
<td>FFFB</td>
<td>16</td>
<td>L</td>
<td>8</td>
<td>13</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>Output</td>
</tr>
<tr>
<td>8</td>
<td>FFF9</td>
<td>16</td>
<td>U</td>
<td>7</td>
<td>13</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>Output</td>
</tr>
</tbody>
</table>

### Data Access

- **Index 1**: Address: FFFE and FFFF. Width: 16 bits, Entries Accessed: 1, Number of Entries: 0, Direction: Output
  - Data: 8A8A

- **Index 3**: Address: FFFC. Width: 8 bits, Entries Accessed: 36, Number of Entries: 0, Direction: Output
  - Data: 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01 01 FE 01

- **Index 4**: Address: FFFD. Width: 8 bits, Entries Accessed: 31, Number of Entries: 45, Direction: Input
  - Data: 40 40 C0 40 40 C0 C0 C0 40 40 C0 40 40 C0 C0 C0 40 40 C0 40 40 C0 40 40 C0 C0 C0 40 40 C0 40 40 C0 C0 C0

- **Index 5**: Address: FFFA and FFFB. Width: 16 bits, Entries Accessed: 12, Number of Entries: 30, Direction: Input
  - Data: 0A54 A370 5C0D EF42 06A5 9EAB 03FC B051 7CA3 5024 3957 0AC5 19AF 9D24 0ACD EB3F 9AFC 03E7 9BA2 37AE F023 D5CA 9A3B C247 349A 181C 32A0 B62C A034 2846

- **Index 7**: Address: FFFB and FFF9. Width: 16 bits, Entries Accessed: 12, Number of Entries: 0, Direction: Output
  - Data: 092A 05B8 0A06 0BA1 0752 0B54 05FE 092B 0A51 0412 0B8B 0962

**wac> ff**
Ordered Opcodes for the Intel 8086 Processor

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Class Key</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000d w md reg m (displo) (disphi)</td>
<td>A 1</td>
<td>ADD</td>
<td>add r/m with register to either</td>
</tr>
<tr>
<td>0000010w data (data w = 1)</td>
<td>A 0</td>
<td>ADD</td>
<td>add immediate to accumulator</td>
</tr>
<tr>
<td>000010d w md reg r/m (displo) (disphi)</td>
<td>L 1</td>
<td>OR</td>
<td>or r/m and register to either</td>
</tr>
<tr>
<td>0000110w data (data w = 1)</td>
<td>L 0</td>
<td>OR</td>
<td>or immediate to accumulator</td>
</tr>
<tr>
<td>000100d w md reg r/m (displo) (disphi)</td>
<td>A 1</td>
<td>ADC</td>
<td>add with carry r/m with register to either</td>
</tr>
<tr>
<td>0001010w data (data w = 1)</td>
<td>A 0</td>
<td>ADC</td>
<td>add with carry immediate to accumulator</td>
</tr>
<tr>
<td>000110d w md reg r/m (displo) (disphi)</td>
<td>A 1</td>
<td>SBB</td>
<td>subtract with borrow r/m and register to either</td>
</tr>
<tr>
<td>0001110w data (data w = 1)</td>
<td>A 0</td>
<td>SBB</td>
<td>subtract with borrow immediate from accumulator</td>
</tr>
<tr>
<td>000sr110</td>
<td>D 0</td>
<td>PUSH</td>
<td>push segment register</td>
</tr>
<tr>
<td>000sr111</td>
<td>D 0</td>
<td>POP</td>
<td>pop segment register</td>
</tr>
<tr>
<td>001000d w md reg r/m (displo) (disphi)</td>
<td>L 1</td>
<td>AND</td>
<td>and r/m with register to either</td>
</tr>
<tr>
<td>0010010w data (data w = 1)</td>
<td>L 0</td>
<td>AND</td>
<td>and immediate to accumulator</td>
</tr>
<tr>
<td>00100111</td>
<td>A 0</td>
<td>AAA</td>
<td>ASCII adjust for add</td>
</tr>
<tr>
<td>0010100d w md reg r/m (displo) (disphi)</td>
<td>A 1</td>
<td>DAA</td>
<td>decimal adjust for add</td>
</tr>
<tr>
<td>0010110w data (data w = 1)</td>
<td>A 0</td>
<td>DAA</td>
<td>decimal adjust for subtract</td>
</tr>
<tr>
<td>001100d w md reg r/m (displo) (disphi)</td>
<td>L 1</td>
<td>XOR</td>
<td>exclusive or r/m and register to either</td>
</tr>
<tr>
<td>0011011w data (data w = 1)</td>
<td>L 0</td>
<td>XOR</td>
<td>exclusive or immediate to accumulator</td>
</tr>
<tr>
<td>00110111</td>
<td>A 0</td>
<td>AAA</td>
<td>ASCII adjust for add</td>
</tr>
<tr>
<td>001110d w md reg r/m (displo) (disphi)</td>
<td>A 1</td>
<td>CMP</td>
<td>compare r/m and register</td>
</tr>
<tr>
<td>0011110w data</td>
<td>A 0</td>
<td>CMP</td>
<td>compare immediate with accumulator</td>
</tr>
<tr>
<td>00111111</td>
<td>A 0</td>
<td>AAS</td>
<td>ASCII adjust for subtract</td>
</tr>
<tr>
<td>01000reg</td>
<td>A 5</td>
<td>INC</td>
<td>increment 16 bit register</td>
</tr>
<tr>
<td>01001reg</td>
<td>A 5</td>
<td>DEC</td>
<td>decrement 16 bit register</td>
</tr>
<tr>
<td>01010reg</td>
<td>D 5</td>
<td>PUSH</td>
<td>push 16 bit register</td>
</tr>
<tr>
<td>01011reg</td>
<td>D 5</td>
<td>POP</td>
<td>pop 16 bit register</td>
</tr>
<tr>
<td>Opcode</td>
<td>Class Key Mnemonic</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>--------------------</td>
<td>-------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>01110000</td>
<td>ip-inc-8</td>
<td>jump on overflow</td>
<td></td>
</tr>
<tr>
<td>01110001</td>
<td>ip-inc-8</td>
<td>jump on not overflow</td>
<td></td>
</tr>
<tr>
<td>01110010</td>
<td>ip-inc-8</td>
<td>jump on below / not above or equal</td>
<td></td>
</tr>
<tr>
<td>01110011</td>
<td>ip-inc-8</td>
<td>jump on not below / above or equal</td>
<td></td>
</tr>
<tr>
<td>01110100</td>
<td>ip-inc-8</td>
<td>jump on equal / zero</td>
<td></td>
</tr>
<tr>
<td>01110101</td>
<td>ip-inc-8</td>
<td>jump on not equal / not zero</td>
<td></td>
</tr>
<tr>
<td>01110110</td>
<td>ip-inc-8</td>
<td>jump on below or equal / not above</td>
<td></td>
</tr>
<tr>
<td>01110111</td>
<td>ip-inc-8</td>
<td>jump on not below or equal / above</td>
<td></td>
</tr>
<tr>
<td>01111000</td>
<td>ip-inc-8</td>
<td>jump on sign</td>
<td></td>
</tr>
<tr>
<td>01111001</td>
<td>ip-inc-8</td>
<td>jump on not sign</td>
<td></td>
</tr>
<tr>
<td>01111010</td>
<td>ip-inc-8</td>
<td>jump on parity / parity even</td>
<td></td>
</tr>
<tr>
<td>01111011</td>
<td>ip-inc-8</td>
<td>jump on not parity / parity odd</td>
<td></td>
</tr>
<tr>
<td>01111100</td>
<td>ip-inc-8</td>
<td>jump on less / not greater or equal</td>
<td></td>
</tr>
<tr>
<td>01111101</td>
<td>ip-inc-8</td>
<td>jump on less / greater or equal</td>
<td></td>
</tr>
<tr>
<td>01111110</td>
<td>ip-inc-8</td>
<td>jump on not less or equal / greater</td>
<td></td>
</tr>
<tr>
<td>01111111</td>
<td>ip-inc-8</td>
<td>jump on not less or equal / greater</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000sw</td>
<td>md 000 r/m (displo) (dispshi) (data (s:w = 01)</td>
</tr>
<tr>
<td>1000000w</td>
<td>md 001 r/m (displo) (dispshi) (data (s:w = 01)</td>
</tr>
<tr>
<td>100000sw</td>
<td>md 010 r/m (displo) (dispshi) (data (s:w = 01)</td>
</tr>
<tr>
<td>100000sw</td>
<td>md 011 r/m (displo) (dispshi) (data (s:w = 01)</td>
</tr>
<tr>
<td>1000000w</td>
<td>md 100 r/m (displo) (dispshi) (data (s:w = 01)</td>
</tr>
<tr>
<td>1000000w</td>
<td>md 101 r/m (displo) (dispshi) (data (s:w = 01)</td>
</tr>
<tr>
<td>1000000w</td>
<td>md 110 r/m (displo) (dispshi) (data (s:w = 01)</td>
</tr>
<tr>
<td>1000000w</td>
<td>md 111 r/m (displo) (dispshi) (data (s:w = 01)</td>
</tr>
<tr>
<td>1000010w</td>
<td>md reg r/m (displo) (dispshi)</td>
</tr>
<tr>
<td>1000011w</td>
<td>md reg r/m (displo) (dispshi)</td>
</tr>
<tr>
<td>1000100w</td>
<td>md reg r/m (displo) (dispshi)</td>
</tr>
<tr>
<td>10001100</td>
<td>md o sr r/m</td>
</tr>
<tr>
<td>10001110</td>
<td>md o sr r/m</td>
</tr>
<tr>
<td>10001111</td>
<td>md 000 r/m</td>
</tr>
<tr>
<td>10010reg</td>
<td></td>
</tr>
<tr>
<td>10011000</td>
<td></td>
</tr>
<tr>
<td>10011001</td>
<td></td>
</tr>
<tr>
<td>10011010</td>
<td>ip-lo ip-hi cs-lo cs-hi</td>
</tr>
<tr>
<td>10011011</td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>Class</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>10011100</td>
<td>D</td>
</tr>
<tr>
<td>10011101</td>
<td>D</td>
</tr>
<tr>
<td>10011110</td>
<td>D</td>
</tr>
<tr>
<td>10011111</td>
<td>D</td>
</tr>
<tr>
<td>1010000w addrlo addrhl</td>
<td>D</td>
</tr>
<tr>
<td>1010001w addrlo addrhl</td>
<td>D</td>
</tr>
<tr>
<td>1010100w data (data w = 1)</td>
<td>S</td>
</tr>
<tr>
<td>1010101w</td>
<td>S</td>
</tr>
<tr>
<td>1010110w</td>
<td>L</td>
</tr>
<tr>
<td>1010111w</td>
<td>S</td>
</tr>
<tr>
<td>1111wreg data (data)</td>
<td>S</td>
</tr>
<tr>
<td>11000010 data-lo data-hi</td>
<td>C</td>
</tr>
<tr>
<td>11000011</td>
<td>C</td>
</tr>
<tr>
<td>11000100 md reg r/m (displo) (disphi)</td>
<td>D</td>
</tr>
<tr>
<td>11000101 md reg r/m (displo) (disphi)</td>
<td>D</td>
</tr>
<tr>
<td>1100011w md 000 r/m (displo) (disphi) data (data w = 1)</td>
<td>D</td>
</tr>
<tr>
<td>11001010 data-lo data-hi</td>
<td>C</td>
</tr>
<tr>
<td>1100111</td>
<td>C</td>
</tr>
<tr>
<td>11001100</td>
<td>C</td>
</tr>
<tr>
<td>11001111</td>
<td>C</td>
</tr>
<tr>
<td>1111000w md 000 r/m (displo) (disphi)</td>
<td>L</td>
</tr>
<tr>
<td>1111000w md 001 r/m (displo) (disphi)</td>
<td>L</td>
</tr>
<tr>
<td>1111000w md 010 r/m (displo) (disphi)</td>
<td>L</td>
</tr>
<tr>
<td>1111000w md 011 r/m (displo) (disphi)</td>
<td>L</td>
</tr>
<tr>
<td>1111000w md 100 r/m (displo) (disphi)</td>
<td>L</td>
</tr>
<tr>
<td>1111000w md 101 r/m (displo) (disphi)</td>
<td>L</td>
</tr>
<tr>
<td>1111000w md 111 r/m (displo) (disphi)</td>
<td>L</td>
</tr>
<tr>
<td>11010010 000001010</td>
<td>A</td>
</tr>
<tr>
<td>11010101 000001010</td>
<td>A</td>
</tr>
<tr>
<td>11010111</td>
<td>D</td>
</tr>
<tr>
<td>11011xxx md xxx r/m (displo) (disphi)</td>
<td>P</td>
</tr>
<tr>
<td>Opcode</td>
<td>Class</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>11100000 ip-inc-8</td>
<td>C 0</td>
</tr>
<tr>
<td>11100001 ip-inc-8</td>
<td>C 0</td>
</tr>
<tr>
<td>11100010 ip-inc-8</td>
<td>C 0</td>
</tr>
<tr>
<td>11100011 ip-inc-8</td>
<td>C 0</td>
</tr>
<tr>
<td>1110010w port</td>
<td>D 0</td>
</tr>
<tr>
<td>1110011w port</td>
<td>D 0</td>
</tr>
<tr>
<td>11101000 ip-inc-lo ip-inc-hi</td>
<td>C 0</td>
</tr>
<tr>
<td>11101001 ip-inc-lo ip-inc-hi</td>
<td>C 0</td>
</tr>
<tr>
<td>11101010 ip-lo ip-hi cs-lo cs-hi</td>
<td>C 0</td>
</tr>
<tr>
<td>11101011 ip-inc8</td>
<td>C 0</td>
</tr>
<tr>
<td>1110110w</td>
<td>D 0</td>
</tr>
<tr>
<td>1110111w</td>
<td>D 0</td>
</tr>
<tr>
<td>11110000</td>
<td>P 0</td>
</tr>
<tr>
<td>1111001z</td>
<td>S 0</td>
</tr>
<tr>
<td>11110100</td>
<td>P 0</td>
</tr>
<tr>
<td>11110101</td>
<td>P 0</td>
</tr>
<tr>
<td>1111011w md 000 r/m (displo) (disphi) data (data w = 1)</td>
<td>E 7</td>
</tr>
<tr>
<td>1111011w md 010 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>1111011w md 011 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>1111011w md 100 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>1111011w md 101 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>1111011w md 110 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>1111011w md 111 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>11111000</td>
<td>P 0</td>
</tr>
<tr>
<td>11111001</td>
<td>P 0</td>
</tr>
<tr>
<td>11111010</td>
<td>P 0</td>
</tr>
<tr>
<td>11111011</td>
<td>P 0</td>
</tr>
<tr>
<td>11111100</td>
<td>P 0</td>
</tr>
<tr>
<td>11111101</td>
<td>P 0</td>
</tr>
<tr>
<td>1111111w md 000 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>1111111w md 001 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>11111111 md 010 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>11111111 md 011 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>11111111 md 100 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>11111111 md 101 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
<tr>
<td>11111111 md 110 r/m (displo) (disphi)</td>
<td>E 7</td>
</tr>
</tbody>
</table>
Definition of Class codes:

A = Arithmetic
D = Data Transfer
L = Logic
E = Extended, must be resolved into one of the other six classes

C = Control Transfer
S = String
P = Processor Control

Definition of abbreviations:

d = direction, from (0) or to (1) CPU
w = width, eight (0) or sixteen (1) bits
md = "mod", used to compute the Effective Address
r/m = register or memory indicator
   for register operations, this is the register ID
   for memory operations, determines the addressing mode
displo = low order eight bits of a displacement
dispdi = high order eight bits of a displacement
ip-inc-8 = eight bit offset
to be added to the Instruction Pointer
xxx = "don't care" bits
port = eight bit address for a fixed port
data = eight bits of data
z = zero flag indicator for REPEAT instruction
sr = segment register
reg = general register designator
s:w = sign extension indicator
   s:w = 01 then sixteen bits of immediate data follow
   s:w = 11 then sign extend eight bits of data to sixteen
ip-lo = low order eight bits of the Instruction Pointer
ip-hi = high order eight bits of the Instruction Pointer
cs-lo = low order eight bits of the Code Segment Register
cs-hi = high order eight bits of the Code Segment Register
addrllo = low order eight bits of an address
addrhi = high order eight bits of an address
ip-inc-lo = low order eight bits to be added
to the Instruction Pointer
ip-inc-hi = high order eight bits to be added
to the Instruction Pointer
int-type = interrupt type
cs = register to be used for segment override
v = flag indicating a single shift (0) to be performed
   or a variable count (1) should be used for shifts
Appendix C
Opcode Look Up Table

<table>
<thead>
<tr>
<th>Index</th>
<th>Hex</th>
<th>Class</th>
<th>Key</th>
<th>Dir</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>A</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>A</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>D</td>
<td>4</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>D</td>
<td>4</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>L</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>L</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>L</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>L</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>L</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>L</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>D</td>
<td>4</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>D</td>
<td>4</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>A</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>A</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>D</td>
<td>4</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>D</td>
<td>4</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>A</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>A</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>A</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>D</td>
<td>4</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>D</td>
<td>4</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>L</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>L</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>L</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>L</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>L</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>L</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>P</td>
<td>4</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Index</td>
<td>Hex</td>
<td>Class</td>
<td>Key</td>
<td>Dir</td>
<td>Width</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>39</td>
<td>27 A</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>28 A</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>29 A</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2A A</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>2B A</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>2C A</td>
<td>0</td>
<td>N</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>2D A</td>
<td>0</td>
<td>N</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>2E P</td>
<td>4</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>2F A</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>30 L</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>31 L</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>32 L</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>33 L</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>34 L</td>
<td>0</td>
<td>N</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>35 L</td>
<td>0</td>
<td>N</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>36 P</td>
<td>4</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>37 A</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>38 A</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>39 A</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>3A A</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>3B A</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>3C A</td>
<td>0</td>
<td>N</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>3D A</td>
<td>0</td>
<td>N</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>3E P</td>
<td>4</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>3F A</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>40 A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>41 A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>42 A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>43 A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>44 A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>45 A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>46 A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>47 A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>48 A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>49 A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>4A A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>4B A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>4C A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>4D A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>4E A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>4F A</td>
<td>5</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>Hex</td>
<td>Class</td>
<td>Key</td>
<td>Dir</td>
<td>Width</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>81</td>
<td>51</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>82</td>
<td>52</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>83</td>
<td>53</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>84</td>
<td>54</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>85</td>
<td>55</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>86</td>
<td>56</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>87</td>
<td>57</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>88</td>
<td>58</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>89</td>
<td>59</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>90</td>
<td>5A</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>91</td>
<td>5B</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>92</td>
<td>5C</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>93</td>
<td>5D</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>94</td>
<td>5E</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>95</td>
<td>5F</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>96</td>
<td>60</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>97</td>
<td>61</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>98</td>
<td>62</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>99</td>
<td>63</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>100</td>
<td>64</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>101</td>
<td>65</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>102</td>
<td>66</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>103</td>
<td>67</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>104</td>
<td>68</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>105</td>
<td>69</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>106</td>
<td>6A</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>107</td>
<td>6B</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>108</td>
<td>6C</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>109</td>
<td>6D</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>110</td>
<td>6E</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>111</td>
<td>6F</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>112</td>
<td>70</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>113</td>
<td>71</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>114</td>
<td>72</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>115</td>
<td>73</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>116</td>
<td>74</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>117</td>
<td>75</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>118</td>
<td>76</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>119</td>
<td>77</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>120</td>
<td>78</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Index</td>
<td>Hex</td>
<td>Class</td>
<td>Key</td>
<td>Dir</td>
<td>Width</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>121</td>
<td>79</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>122</td>
<td>7A</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>123</td>
<td>7B</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>124</td>
<td>7C</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>125</td>
<td>7D</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>126</td>
<td>7E</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>127</td>
<td>7F</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>128</td>
<td>80</td>
<td>E</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>129</td>
<td>81</td>
<td>E</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>130</td>
<td>82</td>
<td>E</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>131</td>
<td>83</td>
<td>E</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>132</td>
<td>84</td>
<td>L</td>
<td>1</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>133</td>
<td>85</td>
<td>L</td>
<td>1</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>134</td>
<td>86</td>
<td>D</td>
<td>1</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>135</td>
<td>87</td>
<td>D</td>
<td>1</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>136</td>
<td>88</td>
<td>D</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>137</td>
<td>89</td>
<td>D</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>138</td>
<td>8A</td>
<td>D</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>139</td>
<td>8B</td>
<td>D</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>140</td>
<td>8C</td>
<td>D</td>
<td>3</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>141</td>
<td>8D</td>
<td>D</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>142</td>
<td>8E</td>
<td>D</td>
<td>3</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>143</td>
<td>8F</td>
<td>D</td>
<td>2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>144</td>
<td>90</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>145</td>
<td>91</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>146</td>
<td>92</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>147</td>
<td>93</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>148</td>
<td>94</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>149</td>
<td>95</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>150</td>
<td>96</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>151</td>
<td>97</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>152</td>
<td>98</td>
<td>A</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>153</td>
<td>99</td>
<td>A</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>154</td>
<td>9A</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>155</td>
<td>9B</td>
<td>P</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>156</td>
<td>9C</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>157</td>
<td>9D</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>158</td>
<td>9E</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>159</td>
<td>9F</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>160</td>
<td>A0</td>
<td>D</td>
<td>8</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>161</td>
<td>A1</td>
<td>D</td>
<td>8</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>162</td>
<td>A2</td>
<td>D</td>
<td>8</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>163</td>
<td>A3</td>
<td>D</td>
<td>8</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>Index</td>
<td>Hex</td>
<td>Class</td>
<td>Key</td>
<td>Dir</td>
<td>Width</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>164</td>
<td>A4</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>165</td>
<td>A5</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>166</td>
<td>A6</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>167</td>
<td>A7</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>168</td>
<td>A8</td>
<td>L</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>169</td>
<td>A9</td>
<td>L</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>170</td>
<td>AA</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>171</td>
<td>AB</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>172</td>
<td>AC</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>173</td>
<td>AD</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>174</td>
<td>AE</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>175</td>
<td>AF</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>176</td>
<td>B0</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>177</td>
<td>B1</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>178</td>
<td>B2</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>179</td>
<td>B3</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>180</td>
<td>B4</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>181</td>
<td>B5</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>182</td>
<td>B6</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>183</td>
<td>B7</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>184</td>
<td>B8</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>185</td>
<td>B9</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>186</td>
<td>BA</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>187</td>
<td>BB</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>188</td>
<td>BC</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>189</td>
<td>BD</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>190</td>
<td>BE</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>191</td>
<td>BF</td>
<td>D</td>
<td>5</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>192</td>
<td>C0</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>193</td>
<td>C1</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>194</td>
<td>C2</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>195</td>
<td>C3</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>196</td>
<td>C4</td>
<td>D</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>197</td>
<td>C5</td>
<td>D</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>198</td>
<td>C6</td>
<td>D</td>
<td>2</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>199</td>
<td>C7</td>
<td>D</td>
<td>2</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>200</td>
<td>C8</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>201</td>
<td>C9</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>202</td>
<td>CA</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>203</td>
<td>CB</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>204</td>
<td>CC</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>205</td>
<td>CD</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>206</td>
<td>CE</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>207</td>
<td>CF</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Index</td>
<td>Hex</td>
<td>Class</td>
<td>Key</td>
<td>Dir</td>
<td>Width</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>208</td>
<td>D0</td>
<td>L</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>209</td>
<td>D1</td>
<td>L</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>210</td>
<td>D2</td>
<td>L</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>211</td>
<td>D3</td>
<td>L</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>212</td>
<td>D4</td>
<td>A</td>
<td>6</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>213</td>
<td>D5</td>
<td>A</td>
<td>6</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>214</td>
<td>D6</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>215</td>
<td>D7</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>216</td>
<td>D8</td>
<td>P</td>
<td>2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>217</td>
<td>D9</td>
<td>P</td>
<td>2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>218</td>
<td>DA</td>
<td>P</td>
<td>2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>219</td>
<td>DB</td>
<td>P</td>
<td>2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>220</td>
<td>DC</td>
<td>P</td>
<td>2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>221</td>
<td>DD</td>
<td>P</td>
<td>2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>222</td>
<td>DE</td>
<td>P</td>
<td>2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>223</td>
<td>DF</td>
<td>P</td>
<td>2</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>224</td>
<td>E0</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>225</td>
<td>E1</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>226</td>
<td>E2</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>227</td>
<td>E3</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>228</td>
<td>E4</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>229</td>
<td>E5</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>230</td>
<td>E6</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>231</td>
<td>E7</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>232</td>
<td>E8</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>233</td>
<td>E9</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>234</td>
<td>EA</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>235</td>
<td>EB</td>
<td>C</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>236</td>
<td>EC</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>237</td>
<td>ED</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>238</td>
<td>EE</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>239</td>
<td>EF</td>
<td>D</td>
<td>0</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>240</td>
<td>F0</td>
<td>P</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>241</td>
<td>F1</td>
<td>Q</td>
<td>-1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>242</td>
<td>F2</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>243</td>
<td>F3</td>
<td>S</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>244</td>
<td>F4</td>
<td>P</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>245</td>
<td>F5</td>
<td>P</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>246</td>
<td>F6</td>
<td>E</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>247</td>
<td>F7</td>
<td>E</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>248</td>
<td>F8</td>
<td>P</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>249</td>
<td>F9</td>
<td>P</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>250</td>
<td>FA</td>
<td>P</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>251</td>
<td>FB</td>
<td>P</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>252</td>
<td>FC</td>
<td>P</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>253</td>
<td>FD</td>
<td>P</td>
<td>0</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>254</td>
<td>FE</td>
<td>E</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>255</td>
<td>FF</td>
<td>E</td>
<td>7</td>
<td>N</td>
<td>P</td>
</tr>
</tbody>
</table>

- 130 -
Appendix D
Data Dictionary

This is a Data Dictionary for the global/external constants, data structures and variables used in the Simulator.

- Activate: constant - defined as "true" for use in the Debugger to indicate whether or not a given breakpoint should be used.

- AddressState: constant - defines the state (in a finite state machine sense) where the address is assembled for entering data into memory using the Debugger assignment function.

- AHId: constant - identifies the AH register, defining it in binary as '100'.

- AHWidth: constant - defines AH as an eight bit register.

- ALId: constant - identifies the AL register, defining it in binary as '000'.

- All16Bits: constant - defines a mask for all 16 bits as a hexadecimal 'FFFF'.

- AllBPIDs: constant - defines the collection of all sixteen breakpoints.

- ALorAXId: constant - identifies the AL or AX register or accumulator (as eight or sixteen bits) where the width has not yet been determined. The constant has a value in binary of '000'.

- ALWidth: constant - defines AL as an eight bit register.

- ArithmeticClass: constant - identifies the entries in the Opcode Look Up Table as belonging to the Arithmetic Class and having the value 'A'.

- 131 -
- AssignmentID: constant - identifies the operator encountered when parsing the Debug command line as an assignment operator, and assigning it a value of one.

- AuxCarryFlag: constant - defines the bit position of the Auxiliary Carry Flag in binary as '000000010000'. This constant also serves as a mask to assemble or isolate the flag.

- AX: variable - a global, unsigned variable containing the value of the AX (and consequently the AL and AH) register.

- AXId: constant - identifies the AX register or the accumulator, defining it in binary as '000'.

- AXWidth: constant - defines AX as a sixteen bit register.

- BadCEAModeValue: error message - indicates that an Effective Address could not be computed successfully.

- BadCheckSum: error message - indicates that a checksum error was encountered attempting to load the executable image into the Simulator's memory.

- BadMemoryAddress: error message - indicates an attempt was made to access an invalid memory location. This most likely occurred as a result of an address outside of the supported memory bounds.

- BadMemoryWidth: error message - indicates the Simulator was unable to interpret a width parameter.

- BadOpcode: error message - indicates an invalid or unsupported opcode was encountered.

- BadOpcodeClass: error message - indicates an invalid opcode class (in the Opcode Look Up Table) was encountered.
- **BadOpcodeExtension**: error message – indicates the opcode extension could not be resolved.

- **BadOpcodeKey**: error message – indicates the opcode key (from the Opcode Look Up Table) is not supported.

- **BadOperandType**: error message – indicates an invalid operand type was encountered.

- **BadPortAddress**: error message – indicates an invalid or undefined I/O port access was attempted.

- **BadRegisterID**: error message – indicates an invalid register identifier was used in the current operation.

- **BadRMDModeValue**: error message – indicates an invalid register/memory code was encountered.

- **BHId**: constant – identifies the BH register, defining it in binary as '111'.

- **BHWidth**: constant – defines BH as an eight bit register.

- **Bit8Divisor**: constant – defines the bit eight divisor in hexadecimal as '100'. The constant is used to shift data from the high eight bits to the low eight bits.

- **Bit8Multiplier**: constant – defines the bit eight multiplier in hexadecimal as '100'. The constant is used to shift data from the low eight bits to the high eight bits.

- **Blank**: constant – defines a space or blank character.

- **BlankClass**: Debugger Command Class – This class is used in the Debugger to eliminate extra spaces input by the user to improve the clarity of their commands.
- **BLId:** constant - identifies the BL register, defining it in binary as '011'.

- **BLWidth:** constant - defines BL as an eight bit register.

- **BooleanID:** constant - indicates that the contents of memory being displayed by the Debugger should be displayed in the boolean format.

- **BooleanState:** constant - defines the state (in a finite state machine sense) where the boolean data are assembled for entering data into memory using the Debugger assignment function.

- **BP:** variable - a global, unsigned variable containing the value of the BP (Base Pointer) register.

- **BPId:** constant - identifies the BP register, defining it in binary as '101'.

- **BPWidth:** constant - defines BP as a sixteen bit register.

- **Breakpoint:** constant - indicates that the simulation has ceased because a breakpoint was encountered.

- **BreakpointClass:** Debugger Command Class - processes the operator commands related to breakpoint operations.

- **BreakpointOpcode:** constant - defines the opcode to be substituted in memory to cause control to return to the Debugger.

- **BreakPointTableType:** array of records - provides the storage locations for the information stored about each breakpoint.
- **BreakRecord**: record - defines the information stored for each breakpoint as consisting of the following variables:

  + **Encountered**: boolean variable - indicates if the breakpoint has been encountered.

  + **Activated**: boolean variable - indicates if the breakpoint should be activated when execution resumes.

  + **Segment**: unsigned variable - contains the segment register portion of the desired breakpoint location.

  + **Address**: unsigned variable - contains the address portion of the desired breakpoint location.

  + **PhysicalAddress**: unsigned variable - used to verify that the proposed breakpoint location is within the bounds of defined memory.

  + **Code**: unsigned variable - contains the opcode which was removed to insert the breakpoint opcode.

- **BX**: variable - a global, unsigned variable containing the value of the BX (and consequently the BL and BH) register.

- **BXId**: constant - identifies the BX register, defining it in binary as '011'.

- **BXWidth**: constant - defines BX as a sixteen bit register.

- **ByteODivisor**: constant - defines the divisor to separate off the low eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000001'.

- **ByteOMask**: constant - defines the mask to separate off the low eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '000000FF'.

- 135 -
- **Byte0Multiplier**: constant - defines the multiplier to pack data into the low eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000001'.

- **Byte1Divisor**: constant - defines the divisor to separate off the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000100'.

- **Byte1Mask**: constant - defines the mask to separate off the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '0000FF00'.

- **Byte1Multiplier**: constant - defines the multiplier to pack data into the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000100'.

- **Byte2Divisor**: constant - defines the divisor to separate off the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00010000'.

- **Byte2Mask**: constant - defines the mask to separate off the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00FF0000'.

- **Byte2Multiplier**: constant - defines the multiplier to pack data into the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00010000'.

- **Byte3Divisor**: constant - defines the divisor to separate off the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '01000000'.

- **Byte3Mask**: constant - defines the mask to separate off the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of 'FF000000'.

- **Byte3Multiplier**: constant - defines the multiplier to pack data into the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '01000000'.

- **ByteOMultiplier**: constant - defines the multiplier to pack data into the low eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000001'.

- **ByteDivisor**: constant - defines the divisor to separate off the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000100'.

- **ByteMask**: constant - defines the mask to separate off the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '0000FF00'.

- **ByteMultiplier**: constant - defines the multiplier to pack data into the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000100'.

- **Byte2Divisor**: constant - defines the divisor to separate off the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00010000'.

- **Byte2Mask**: constant - defines the mask to separate off the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00FF0000'.

- **Byte2Multiplier**: constant - defines the multiplier to pack data into the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00010000'.

- **Byte3Divisor**: constant - defines the divisor to separate off the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '01000000'.

- **Byte3Mask**: constant - defines the mask to separate off the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of 'FF000000'.

- **Byte3Multiplier**: constant - defines the multiplier to pack data into the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '01000000'.

- **ByteOMultiplier**: constant - defines the multiplier to pack data into the low eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000001'.

- **ByteDivisor**: constant - defines the divisor to separate off the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000100'.

- **ByteMask**: constant - defines the mask to separate off the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '0000FF00'.

- **ByteMultiplier**: constant - defines the multiplier to pack data into the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000100'.

- **Byte2Divisor**: constant - defines the divisor to separate off the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00010000'.

- **Byte2Mask**: constant - defines the mask to separate off the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00FF0000'.

- **Byte2Multiplier**: constant - defines the multiplier to pack data into the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00010000'.

- **Byte3Divisor**: constant - defines the divisor to separate off the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '01000000'.

- **Byte3Mask**: constant - defines the mask to separate off the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of 'FF000000'.

- **Byte3Multiplier**: constant - defines the multiplier to pack data into the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '01000000'.

- **ByteOMultiplier**: constant - defines the multiplier to pack data into the low eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000001'.

- **ByteDivisor**: constant - defines the divisor to separate off the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000100'.

- **ByteMask**: constant - defines the mask to separate off the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '0000FF00'.

- **ByteMultiplier**: constant - defines the multiplier to pack data into the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000100'.

- **Byte2Divisor**: constant - defines the divisor to separate off the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00010000'.

- **Byte2Mask**: constant - defines the mask to separate off the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00FF0000'.

- **Byte2Multiplier**: constant - defines the multiplier to pack data into the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00010000'.

- **Byte3Divisor**: constant - defines the divisor to separate off the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '01000000'.

- **Byte3Mask**: constant - defines the mask to separate off the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of 'FF000000'.

- **Byte3Multiplier**: constant - defines the multiplier to pack data into the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '01000000'.

- **ByteOMultiplier**: constant - defines the multiplier to pack data into the low eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000001'.

- **ByteDivisor**: constant - defines the divisor to separate off the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000100'.

- **ByteMask**: constant - defines the mask to separate off the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '0000FF00'.

- **ByteMultiplier**: constant - defines the multiplier to pack data into the second eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00000100'.

- **Byte2Divisor**: constant - defines the divisor to separate off the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00010000'.

- **Byte2Mask**: constant - defines the mask to separate off the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00FF0000'.

- **Byte2Multiplier**: constant - defines the multiplier to pack data into the third eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '00010000'.

- **Byte3Divisor**: constant - defines the divisor to separate off the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '01000000'.

- **Byte3Mask**: constant - defines the mask to separate off the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of 'FF000000'.

- **Byte3Multiplier**: constant - defines the multiplier to pack data into the high eight bits in a thirty two bit VAX word. The constant has a value in hexadecimal of '01000000'.
- **ByteID**: constant - indicates that the contents of memory being displayed by the Debugger should be displayed in the byte format.

- **BytesPerWord**: constant - defines the number of Intel bytes of memory that fit in a DEC VAX word.

- **Call**: constant - indicates that the Simulator has encountered a subroutine call, and it may be appropriate to being uninterrupted simulation until a "Return" instruction is encountered.

- **CarryFlag**: constant - defines the bit position of the Carry Flag in binary as '000000000001'. This constant also serves as a mask to assemble or isolate the flag.

- **Characters**: set - a grouping of all characters that the user can enter at a keyboard.

- **CHId**: constant - identifies the CH register, defining it in binary as '101'.

- **CHWidth**: constant - defines CH as an eight bit register.

- **Clear**: constant - defines the state of a boolean constant or variable which is to have the value of false or no bits set.

- **CLId**: constant - identifies the CL register, defining it in binary as '001'.

- **CLWidth**: constant - defines CL as an eight bit register.

- **ColonID**: constant - identifies the operator encountered when parsing the Debug command line as a colon operator, and assigning it a value of two.

- **ColonState**: constant - defines the state (in a finite state machine sense) where a colon has been encountered while assembling an address for entering data into memory using the Debugger assignment function.
- **CommaID**: constant - identifies the operator encountered when parsing the Debug command line as a comma operator, and assigning it a value of five.

- **ControlTransferClass**: constant - identifies the entries in the Opcode Look Up Table as belonging to the Control Transfer Class and having the value 'C'.

- **ControlD**: constant - indicates that the simulation has ceased because the user entered a Control D code.

- **CRTOutPortNumber**: constant - defines the port address for output to the user's terminal.

- **CS**: variable - a global, unsigned variable containing the value of the CS (Code Segment) register.

- **CSId**: constant - identifies the CS register, defining it in binary as '01'.

- **CSWidth**: constant - defines CS as a sixteen bit segment register.

- **CX**: variable - a global, unsigned variable containing the value of the CX (and consequently the CL and CH) register.

- **CXId**: constant - identifies the CX register, defining it in binary as '001'.

- **CXWidth**: constant - defines CX as a sixteen bit register.

- **DataArraySize**: constant - defines the maximum amount of numeric data that a user can enter per Debugger command.

- **DataArrayType**: type - defines how numeric keyboard data are temporarily stored until used by the Debugger.
- **DataTransferClass**: constant - identifies the entries in the Opcode Look Up Table as belonging to the Data Transfer Class and having the value 'D'.

- **DeActivate**: constant - defined as "false" for use in the Debugger to indicate whether or not a given breakpoint should be used.

- **DHId**: constant - identifies the DH register, defining it in binary as '110'.

- **DHWidth**: constant - defines DH as an eight bit register.

- **DI**: variable - a global, unsigned variable containing the value of the DI (Destination Index) register.

- **DIId**: constant - identifies the DI register, defining it in binary as '111'.

- **DirectionFlag**: constant - defines the bit position of the Direction Flag in binary as '010000000000'. This constant also serves as a mask to assemble or isolate the flag.

- **DIWidth**: constant - defines DI as a sixteen bit register.

- **DLId**: constant - identifies the DL register, defining it in binary as '010'.

- **DLWidth**: constant - defines DL as an eight bit register.

- **DS**: variable - a global, unsigned variable containing the value of the DS (Data Segment) register.

- **DSId**: constant - identifies the DS register, defining it in binary as '11'.

- **DSSWidth**: constant - defines DS as a sixteen bit segment register.
- DX: variable - a global, unsigned variable containing the value of the DX (and consequently the DL and DH) register.

- DXId: constant - identifies the DX register, defining it in binary as '010'.

- DXWidth: constant - defines DX as a sixteen bit register.

- EffectiveAddressType: record - defines the information stored for an effective address as consisting of the following variables:

  + Mode: character variable - defines whether this address should be considered as a register reference or a memory reference.

  + Width: integer variable - defines the width (eight or sixteen bits) of the variable to be accessed.

  + Address: unsigned variable - contains the register designation or the memory address to be accessed.

  + Segment: integer variable - contains the segment register to be used when accessing memory.

- EightBits: constant - defines the variable or register as eight bits wide, having a width value of zero.

- Elipse: constant - defines the period character.

- EndofLine: constant - defines the end of line when parsing user input to the Debugger.

- ErrorClass: Debugger Command Class - processes the invalid operator commands.
- ErrorState: constant - defines the state (in a finite state machine sense) that is entered when an error has been encountered entering a datum for the Debugger.

- ES: variable - a global, unsigned variable containing the value of the ES (Extra Segment) register.

- ESId: constant - identifies the ES register, defining it in binary as '00'.

- ESWidth: constant - defines ES as a sixteen bit segment register.

- ExecuteClass: Debugger Command Class - processes the operator commands related to program execution.

- ExtendedClass: constant - identifies the entries in the Opcode Look Up Table as belonging to the Extended Class (those opcodes whose class is not yet fully resolved) and having the value 'E'.

- FF: constant - defines a form feed character to be used in formatting output for the user.

- FileName: variable - contains the path and file name to access VMS files.

- FilenameLength: constant - defines the maximum permissible length of file names.

- FirstBreakpoint: constant - defines the starting entry in the breakpoint table.

- FirstIODataPoint: constant - defines the starting entry in the I/O data table.

- FirstIOPortIndex: constant - defines the starting entry in the I/O port table.
- FirstOpcodeValue: constant - defines the starting entry in the Opcode Look Up Table.

- FirstKeywordEntry: constant - defines the starting entry for the Debugger's Keyword Look Up Table.

- FlagClass: Debugger Command Class - processes the operator commands related to flag (processor status word) operations.

- Flags: variable - a global, record structured variable containing the values for the flags or processor status word.

- FlagType: record - defines the structure used to store the flags as consisting of the following variables:

  + Carry: boolean variable - contains the value of the Carry flag.

  + Parity: boolean variable - contains the value of the Parity flag.

  + AuxCarry: boolean variable - contains the value of the Auxiliary Carry flag.

  + Zero: boolean variable - contains the value of the Zero flag.

  + Sign: boolean variable - contains the value of the Sign flag.

  + Trap: boolean variable - contains the value of the Trap flag.

  + Interrupt: boolean variable - contains the value of the Interrupt flag.
+ Direction: boolean variable - contains the value of the Direction flag.

+ Overflow: boolean variable - contains the value of the Overflow flag.

- Halt: constant - indicates that the simulation has ceased because a halt instruction was encountered.

- HexState: constant - defines the state (in a finite state machine sense) where hexadecimal data are assembled before being stored in memory using the Debugger assignment function.

- High8Bits: constant - defines a mask for the high eight bits of a word and has a value in hexadecimal of 'FF00'.

- HighMemoryLimit: constant - the upper limit of the memory array, having a default value in decimal of 2048. This provides eight kilobytes of user memory.

- InputIndicator: constant - defines the direction of a port as being an input port.

- InputLine: string variable - contains the current user input to the Debugger.

- InputLineLength: constant - defines the maximum length of the user input for the Debugger as 120 decimal characters.

- InputOutputClass: Debugger Command Class - processes the operator commands related to input and output operations.

- InputPointerRange: constant - defines the limits of the index to the user input for the Debugger.

- IntegerID: constant - indicates that the contents of memory being displayed by the Debugger should be displayed in the decimal integer format.
- IntegerState: constant - defines the state (in a finite state machine sense) where integer data are assembled for entering into memory using the Debugger assignment function.

- InterruptFlag: constant - defines the bit position of the Interrupt Flag in binary as '001000000000'. This constant also serves as a mask to assemble or isolate the flag.

- InvalidAddress: constant - defines an invalid or unused address in the I/O port map.

- InvalidClass: constant - identifies the unused and therefore invalid entries in the Opcode Look Up Table as belonging to the Invalid Class and having the value 'I'. This class was created to improve the robustness of decoding opcodes.

- InvalidIndex: constant - defines an invalid or unused index in the I/O port map.

- InvalidLocation: constant - defines an invalid or unused location in the I/O port map.

- IOData: variable - a global, two dimensional array of unsigned variables containing the I/O data for all of the I/O ports.

- IOIndicator: constant - defines this I/O port as a bidirectional port.

- IOPortMap: array of records - provides the storage locations for the information stored about each I/O port.

- IP: variable - a global, unsigned variable containing the value of the Instruction Pointer.
- **KeywordCharacteristics**: record - defines the information stored for each Debugger keyword as consisting of the following variables:

  + **Name**: variable - the name of the keyword.

  + **Class**: variable - the keyword class.

  + **Width**: variable - defines the operation as affecting eight or sixteen bits.

  + **ID**: variable - the identifying value of the keyword within the designated class.

  + **Value**: unsigned variable - used to get numeric values from the command line into the Debugger.

- **KeyboardInPortNumber**: constant - defines the port address for input from the user's terminal.

- **LastBreakpoint**: constant - defines the end of the breakpoint table.

- **LastKeywordEntry**: constant - defines the end of the keyword table.

- **LastOpcodeValue**: constant - defines the last entry in the Opcode Look Up Table.

- **LengthID**: constant - indicates that the specified command should be repeated by the Debugger the indicated number of times.

- **Letters**: set - a grouping of all letters from 'A' to 'Z', both upper and lower case.
- **LettersAndNumbers**: set - a grouping of all letters from 'A' to 'Z' both upper and lower case and all digits '0' to '9'.

- **LogicClass**: constant - identifies the entries in the Opcode Look Up Table as belonging to the Logic Class and having the value 'L'.

- **Low8Bits**: constant - defines a mask for the lower eight bits of a word and has a value in hexadecimal of 'FF'.

- **LowerPortIndicator**: constant - indicates the associated information describes an eight bit I/O port or the lower eight bits of a sixteen bit port.

- **LowMemoryLimit**: constant - defines the lower bound of the memory array having a value in decimal of zero.

- **MaxID**: constant - defines the maximum value of the identifier allowed for a class of Debugger commands.

- **MaxNumberOfIOBytes**: constant - defines the maximum number of data bytes that can be entered for each I/O port, having a decimal default value of 256 or four times the "Max Number Of IO Words" allocated.

- **MaxNumberOfIOPorts**: constant - defines the maximum number of I/O ports for the Simulator and support programs having a decimal default value of eight.

- **MaxNumberOfIOWords**: constant - defines the maximum number of data words that can be allocated for each I/O port, having a decimal default value of 64.

- **MaxWidth**: constant - defines the maximum width used to verify keyword names for the Debugger.

- **Memory**: variable - a global array which contains the user program and data.
- **MemoryArray**: type - defines the limits and data type for the memory array.

- **MemoryClass**: Debugger Command Class - processes the operator commands related to memory access operations.

- **MinID**: constant - defines the minimum value of the identifier for a class of Debugger commands.

- **MinusID**: constant - identifies the operator encountered when parsing the Debug command line as a minus operator, and assigning it a value of four.

- **MinWidth**: constant - defines the minimum width for names in the Debugger keyword table.

- **NameString**: string variable - defines a string to contain the commands parsed off the user input line for the Debugger.

- **NameType**: type - defines the characteristics of the name string.

- **NegativeOperator**: constant - defines a unary operator for negating values input by the user.

- **NoMemoryAccess**: error message - indicates that memory was not accessed during the loading of a program (i.e. the program was not loaded into memory).

- **Normal**: constant - indicates that the simulation of this opcode was completed in a normal fashion.

- **NullAddressValue**: constant - defines a null address value for default operations in the Debugger.

- **NullEntries**: constant - defines a null default value for the number of entries in the Debugger.
- NullSegmentValue: constant - defines a null segment value for default operations in the Debugger.

- Numbers: set - a grouping of all digits '0' to '9'.

- NumericClass: Debugger Command Class - processes the operator commands related to entering numeric data.

- OpcodeLUT: array of records - provides the storage locations for the information stored about each opcode entry in the Opcode Look Up Table.

- OpcodeLUTEntry: record - defines the entries in the Opcode Look Up Table as consisting of the following variables:

  + OpcodeClass: variable - contains the opcode class (i.e. Data Transfer, Arithmetic etc.) for this entry.

  + OpcodeKey: variable - contains the opcode key for this entry.

  + DirectionBitPresent: variable - indicates in a boolean sense if a direction bit is present in this opcode.

  + WidthBitPresent: variable - indicates in a boolean sense if a width bit is present in this opcode.

- OpcodeType: record - defines the information stored for each opcode as consisting of the following variables:

  + Direction: variable - defines whether the information moves to or from the CPU.

  + Full: variable - the value of the opcode.

  + Width: variable - defines this operation as an eight or sixteen bit process.
- OperatorClass: Debugger Command Class - processes the punctuation (i.e. equal sign, plus sign, minus sign, colon and comma) encountered in operator commands.

- OutputIndicator: constant - defines this I/O port as an output port.

- OverflowFlag: constant - defines the bit position of the Overflow Flag in binary as '100000000000'. This constant also serves as a mask to assemble or isolate the flag.

- PageSize: constant - defines the number of lines to be placed on the printed page output for the user.

- ParityFlag: constant - defines the bit position of the Parity Flag in binary as '000000000100'. This constant also serves as a mask to assemble or isolate the flag.

- PlusID: constant - identifies the operator encountered when parsing the Debug command line as a plus operator, and assigning it a value of three.

- PointerID: constant - indicates that the contents of memory being displayed by the Debugger should be displayed in the pointer format.

- PortEntry: record - defines the information stored for each I/O port as consisting of the following variables:

  + Address: variable - contains the address of the I/O port.

  + AuxIndex: variable - contains the index for the remaining eight bits of information if this is a sixteen bit port.

  + DataIndex: variable - contains the user's program index into the data array for this port.
+ InputIndex: variable - contains the number of data entries made by the user when creating input data.

+ PortWidth: variable - indicates if this is an eight or sixteen bit wide port.

+ LowerUpperIndicator: variable - indicates if this is the lower or upper eight bits of information for a sixteen bit port.

+ InOutIndicator: variable - indicates if this is an input port, an output port or a bidirectional port.

- PortMap: array of records - defines the characteristics of the I/O port map.

- PositiveOperator: constant - defines a unary positive operator.

- ProcessorControlClass: constant - identifies the entries in the Opcode Look Up Table as belonging to the Processor Control Class and having the value 'P'.

- QualifierClass: Debugger Command Class - processes the memory operation qualifiers (i.e. LENGTH and TO).

- RegisterClass: Debugger Command Class - processes the operator commands related to register operations.

- RegisterType: record - defines the information stored for each register as consisting of the following variables:

  + Id: variable - contains the register identifier.

  + Width: variable - contains the width parameter for the register (eight or sixteen bits).
- Return: constant - indicates that the Simulator has encountered a "Return" instruction and the Debugger may need to cease uninterrupted simulation and return control to the user.

- SegmentOffset: constant - defines the multiplier to be used when computing a physical address.

- SegmentOverRideCount: variable - contains the necessary information to determine if a segment override should be applied on the next operation.

- SegmentOverRideValue: variable - contains the register identifier that should be used in the segment override operation.

- SegmentState: constant - defines the state (in a finite state machine sense) where the segment portion of the address is assembled for entering data into memory using the Debugger assignment function.

- SeparatorState: constant - defines the state (in a finite state machine sense) which is entered when a datum separator (comma) is encountered while entering data into memory using the Debugger assignment function.

- SetHigh: constant - defines the state of a boolean constant or variable which is to have the value of true or all bits set.

- ShortString: string variable - contains a short string for manipulation by the Simulator.

- ShortStringLength: constant - defines the length of the short string.

- SI: variable - a global, unsigned variable containing the value of the SI (Source Index) register.
- **SignFlag**: constant - defines the bit position of the Sign Flag in binary as '000010000000'. This constant also serves as a mask to assemble or isolate the flag.

- **SIId**: constant - identifies the SI register, defining it in binary as '110'.

- **SimpleMode**: constant - defines the "mod" bits for computing the effective address of the direct memory to accumulator and accumulator to memory data transfer operations, and has the binary value of '0'.

- **SimpleRM**: constant - defines the "r/m" bits for computing the effective address of the direct memory to accumulator and accumulator to memory data transfer operations, and has the binary value of '110'.

- **SIntegerID**: constant - indicates that the contents of memory being displayed by the Debugger should be displayed in the signed eight bit decimal integer format.

- **SIWidth**: constant - defines SI as a sixteen bit register.

- **SixteenBits**: constant - defines the variable or register as sixteen bits wide, having a width value of one.

- **SP**: variable - a global, unsigned variable containing the value of the SP (Stack Pointer) register.

- **SPId**: constant - identifies the SP register, defining it in binary as '100'.

- **SPWidth**: constant - defines SP as a sixteen bit register.

- **SRWidth**: constant - defines the segment register as sixteen bits wide, having a width value of two.

- **SS**: variable - a global, unsigned variable containing the value of the SS (Stack Segment) register.
- SSId: constant - identifies the SS register, defining it in binary as '10'.

- SSWidth: constant - defines SS as a sixteen bit segment register.

- StackClass: Debugger Command Class - processes the operator commands related to the stack operations.

- StartOfData: constant - defines the initial index into the data entered by the user for the Debugger.

- StartofLine: constant - defines the initial index into the InputLine for the Debugger.

- StringClass: constant - identifies the entries in the Opcode Look Up Table as belonging to the String Class and having the value 'S'.

- ToID: constant - indicates that the contents of memory should be displayed by the Debugger from the specified starting address to the subsequent terminating address.

- TrapFlag: constant - defines the bit position of the Trap Flag in binary as '000100000000'. This constant also serves as a mask to assemble or isolate the flag.

- TrueFalseClass: Debugger Command Class - processes the operator commands related to entering boolean data.

- UnaryState: constant - defines the state (in a finite state machine sense) which is entered when it is desired to store the two's complement of the datum into memory using the Debugger assignment function.

- UndefinedClass: constant - defines an invalid class for the Debugger keyword table to improve the Debugger's robustness.
- **UndefinedID**: constant - defines an invalid identifier for the Debugger keyword table to improve the Debugger's robustness.

- **UndefinedName**: constant - defines an invalid name for the Debugger keyword table to improve the Debugger's robustness.

- **UndefinedWidth**: constant - defines an invalid name width for the Debugger keyword table to improve the Debugger's robustness.

- **UpperPortIndicator**: constant - defines this I/O port information as belonging to the upper eight bits of a sixteen bit I/O port.

- **UtilityClass**: Debugger Command Class - processes the operator commands related to utility operations (e.g. load and log).

- **WordID**: constant - indicates that the contents of memory being displayed by the Debugger should be displayed in the word format.

- **WordMultiplier**: constant - defines a multiplier to isolate an Intel word and has a hexadecimal value of '100'.

- **ZeroFlag**: constant - defines the bit position of the Zero Flag in binary as '000001000000'. This constant also serves as a mask to assemble or isolate the flag.

- **ZeroOne**: type - defines a range of integers from minus one to one, with an acceptable valid range of zero to one and with an acceptable invalid state of minus one. This is used to perform range checking on the width variable.
Appendix E  
Keyword Look Up Table

The following list contains the current definitions of the keywords used in the Debugger.

<table>
<thead>
<tr>
<th>Index</th>
<th>Keyword</th>
<th>ID</th>
<th>Class</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EXIT</td>
<td>1</td>
<td>U</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>LOG</td>
<td>2</td>
<td>U</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>LOAD</td>
<td>3</td>
<td>U</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>LENGTH</td>
<td>3</td>
<td>Q</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>TO</td>
<td>5</td>
<td>Q</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>=</td>
<td>1</td>
<td>O</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>:</td>
<td>2</td>
<td>O</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>+</td>
<td>3</td>
<td>O</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>4</td>
<td>O</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>,</td>
<td>5</td>
<td>O</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>PORT</td>
<td>2</td>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>WPORT</td>
<td>6</td>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>(blank)</td>
<td>1</td>
<td>(blank)</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>FALSE</td>
<td>0</td>
<td>L</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>TRUE</td>
<td>-1</td>
<td>L</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>STACK</td>
<td>1</td>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>REGISTER</td>
<td>8</td>
<td>R</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>AL</td>
<td>0</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>CL</td>
<td>1</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>DL</td>
<td>2</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>BL</td>
<td>3</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>AH</td>
<td>4</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>CH</td>
<td>5</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>DH</td>
<td>6</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>BH</td>
<td>7</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>AX</td>
<td>0</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>CX</td>
<td>1</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>DX</td>
<td>2</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>BX</td>
<td>3</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>SP</td>
<td>4</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>BP</td>
<td>5</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>SI</td>
<td>6</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>DI</td>
<td>7</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>Index</td>
<td>Keyword</td>
<td>ID</td>
<td>Class</td>
<td>Width</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>----</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>33</td>
<td>ES</td>
<td>0</td>
<td>R</td>
<td>2</td>
</tr>
<tr>
<td>34</td>
<td>CS</td>
<td>1</td>
<td>R</td>
<td>2</td>
</tr>
<tr>
<td>35</td>
<td>SS</td>
<td>2</td>
<td>R</td>
<td>2</td>
</tr>
<tr>
<td>36</td>
<td>DS</td>
<td>3</td>
<td>R</td>
<td>2</td>
</tr>
<tr>
<td>37</td>
<td>IP</td>
<td>4</td>
<td>R</td>
<td>2</td>
</tr>
<tr>
<td>38</td>
<td>RF</td>
<td>5</td>
<td>R</td>
<td>2</td>
</tr>
<tr>
<td>39</td>
<td>RAL</td>
<td>0</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>RCL</td>
<td>1</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>41</td>
<td>RDL</td>
<td>3</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>RBL</td>
<td>3</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>43</td>
<td>RAH</td>
<td>4</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>44</td>
<td>RCH</td>
<td>5</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>45</td>
<td>RDH</td>
<td>6</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>46</td>
<td>RBH</td>
<td>7</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>RAX</td>
<td>0</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>RCX</td>
<td>1</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>RDX</td>
<td>2</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>RBX</td>
<td>3</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>51</td>
<td>CFL</td>
<td>0</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>52</td>
<td>PFL</td>
<td>1</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>53</td>
<td>AFL</td>
<td>2</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>54</td>
<td>ZFL</td>
<td>3</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>55</td>
<td>SFL</td>
<td>4</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>56</td>
<td>TFL</td>
<td>5</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>57</td>
<td>IFL</td>
<td>6</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>58</td>
<td>DFL</td>
<td>7</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>59</td>
<td>OFL</td>
<td>8</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>FLAG</td>
<td>9</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>61</td>
<td>BOOLEAN</td>
<td>1</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>62</td>
<td>BYTE</td>
<td>2</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>63</td>
<td>INTEGER</td>
<td>3</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>POINTER</td>
<td>4</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>65</td>
<td>SINTEGER</td>
<td>5</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>66</td>
<td>WORD</td>
<td>6</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>67</td>
<td>Qqqqqqqqq</td>
<td>-2</td>
<td>z</td>
<td>-1</td>
</tr>
<tr>
<td>68</td>
<td>Qqqqqqqqq</td>
<td>-2</td>
<td>z</td>
<td>-1</td>
</tr>
<tr>
<td>69</td>
<td>Qqqqqqqqq</td>
<td>-2</td>
<td>z</td>
<td>-1</td>
</tr>
<tr>
<td>Index</td>
<td>Keyword</td>
<td>ID</td>
<td>Class</td>
<td>Width</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>----</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>70</td>
<td>BP0</td>
<td>0</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>71</td>
<td>BP1</td>
<td>1</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>72</td>
<td>BP2</td>
<td>2</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>73</td>
<td>BP3</td>
<td>3</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>74</td>
<td>BP4</td>
<td>4</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>75</td>
<td>BP5</td>
<td>5</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>76</td>
<td>BP6</td>
<td>6</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>77</td>
<td>BP7</td>
<td>7</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>78</td>
<td>BP8</td>
<td>8</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>79</td>
<td>BP9</td>
<td>9</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>80</td>
<td>BPA</td>
<td>10</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>81</td>
<td>BPB</td>
<td>11</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>82</td>
<td>BPC</td>
<td>12</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>83</td>
<td>BPD</td>
<td>13</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>84</td>
<td>BPE</td>
<td>14</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>85</td>
<td>BPF</td>
<td>15</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>86</td>
<td>ALLBP</td>
<td>16</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>87</td>
<td>GO</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>88</td>
<td>G</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>89</td>
<td>START</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>90</td>
<td>CONTINUE</td>
<td>2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>91</td>
<td>RESUME</td>
<td>2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>92</td>
<td>R</td>
<td>2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>93</td>
<td>STEP</td>
<td>3</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>94</td>
<td>S</td>
<td>3</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>95</td>
<td>PSTEP</td>
<td>4</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>96</td>
<td>P</td>
<td>4</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>97</td>
<td>Qqqqqqqqq</td>
<td>-2</td>
<td>z</td>
<td>-1</td>
</tr>
<tr>
<td>98</td>
<td>Qqqqqqqqq</td>
<td>-2</td>
<td>z</td>
<td>-1</td>
</tr>
<tr>
<td>99</td>
<td>Qqqqqqqqq</td>
<td>-2</td>
<td>z</td>
<td>-1</td>
</tr>
<tr>
<td>100</td>
<td>EX</td>
<td>1</td>
<td>U</td>
<td>1</td>
</tr>
</tbody>
</table>
13.1 General

- Three main programs are available to the general user:
  + Simulator - for executing programs (SIM86).
  + I/O Map Generation - for creating and reviewing the I/O map (IOMAP86).
  + Review I/O Data - for creating input data and reviewing input and output data (IODATA86).

- All numbers in the Simulator and support programs are in hexadecimal unless otherwise specified.

- Any place a file name is requested, a full VMS path name can be used, but the total path name specified must be less than 40 characters.

- An invalid file name anywhere a file name is requested will crash the Simulator.

- The Simulator should be insensitive to upper/lower case for letters.

- Programs must be assembled, linked and located by the cross products tools.

- Programs must reside between the addresses of 0 and 8192 (8K) (bytes) memory, including all stack and data requirements. Attempts to access outside these boundaries should produce soft errors.

- Each program is responsible for setting up all Segment Registers, Stack Pointer and Instruction Pointer prior to use. (Default values are zero.)
- No Intel system services are provided.

- All trap definitions are left to the user (except for breakpoint, which cannot be redefined).

- The Simulator will display the fact that a trap was encountered.

- Input and output port addresses and data are mapped to save disk space and execution time.

13.2 Input And Output

- Input / Output space is limited to 8 I/O ports. (A 16 bit wide port counts as two ports).

- I/O data is limited to 256 bytes for an 8 bit port or 256 words for a 16 bit port.

- I/O can be directed to the user's terminal by using the port address "FF" for both input and output.

13.3 Debugger

- Anywhere a punctuation or terminator is used, a space can be added for legibility (i.e. byte AX : BX <cr> is the same as byte AX:BX<cr>, or BP0=<cr> is the same as BP0 = <cr>).

- Logging of Debugger commands is available, but recovery from or execution of commands from the log file is not supported.

- All commands can be shortened to 3 letters if their original form is longer. Spelling past 3 letters is not checked.
Available commands - (an alphabetical listing of all commands can be found in Attachment I, and a listing of commands by class can be found in Attachment II):

+ LOG turns on logging of commands.
+ EXIT exits from the Debugger and the Simulator.
+ LOAD loads a file.
+ All registers and flags are available using their standard naming conventions.

Examples:

* RAX <cr> will display the contents of the AX register.

* AX = FdCb <cr> will set the contents of the AX register to the hexadecimal value FDCB.

+ REGISTER will display the contents of all registers.
+ FLAG will display the contents of all flags.
+ Sixteen breakpoints are available: BP0 through BPF.
  - BP0 <cr> displays breakpoint 0.
  - BP0 = yyyy <cr> sets breakpoint 0 at memory location CS : YYYY.
  - BP0 = xxxx : yyyy <cr> sets breakpoint 0 at memory location xxxx : yyyy where xxxx is the segment register value and yyyy is the displacement.
  - BP0 = <cr> clears breakpoint 0.

+ ALLbp <cr> displays all breakpoints.
+ ALLbp = <cr> clears all breakpoints.
+ Starting program execution:
  - GO <cr>, G <cr>, or STArt <cr> will start a program at CS : IP.
  - GO = xxxx : yyyy <cr> will load CS : IP with xxxx and yyyy respectively and start the program.
  - GO = CS : IP <cr> will start the program at CS : IP.
  - GO = yyyy <cr> will assume the current value for CS, load IP with yyyy and start the program at CS : IP.
  - GO = IP <cr> will assume the current value for CS and start the program at CS : IP.

+ RESume, R, CONtinue are used to resume execution.
+ STEp, S will single step through the program.
+ PSTep, P will step over subroutines unless breakpoints are encountered.

+ The following memory access commands are supported:
  - BOOlean - the least significant bit of the byte is interpreted as FALSE if its value is 0 and TRUE if its value is 1.
  - BYTE - 8 bits of data in hexadecimal.
  - WORd - 16 bits of data in hexadecimal.
  - INTeger - 8 bits of data as a signed decimal number.
  - SINteger - 16 bits of data as a signed decimal number.
  - POInter - 32 bits of data interpreted as a segment address pair (e.g. Segment : Address).
All memory segment and address values are specified in hexadecimal.

All addresses for memory can be specified as follows:

- Register1 : Register2 – where the contents of Register1 will be used as the segment value and the contents of Register2 will be used as the address value.

- xxxx : yyyy – where xxxx will be used as the segment value and yyyy will be used as the address value.

- yyyy – where CS will be used as the segment value and yyyy will be used as the address value.

- I/O ports can be accessed by port number or the contents of the DX register.

- If the low address of a 16 bit port is accessed with the WPOrt command then all 16 bits are displayed or modified as requested.

- If the high address of a 16 bit port is accessed with the WPOrt command then all 16 bits are displayed or modified as requested.

- If the low address of a 16 bit port is accessed with the PORt command then only the specified 8 bits are displayed or modified as requested.

- If the high address of a 16 bit port is accessed with the PORt command then only the specified 8 bits are displayed or modified as requested.

- Control D will abort a run away program at the next instruction.

- Control Y or Control C will abort a run away Simulator.
13.4 Attachment I - An Alphabetical Listing Of Debugger Keywords

- (blank)
- AFL
- AH
- AL
- ALLBP, ALLB, ALL
- AX
- BH
- BL
- BOOLEAN, BOOLEA, BOOLE, BOOL, BOO
- BP
- BP0
- BP1
- BP2
- BP3
- BP4
- BP5
- BP6
- BP7
- BP8
- BP9
- BPA
- BPB
- BPC
- BPD
- BPE
- BPF
- BX
- BYTE, BYT
- CFL
- CH
- CL
- CONTINUE, CONTINU, CONTIN, CONTI, CONT, CON
- CS
- CX
- DFL
- DH
- DI
- DL
- DS
- DX
- ES
- EXIT, EXI
- FALSE, FALS, FAL
- FLAG, FLA
- G
- GO, G
- IFL
- INTEGER, INTEG, INTE, INT
- IP
- LENGTH, LENG, LEN
- LOAD, LOA
- LOG
- OFL
- PFL
- P
- POINTER, POINTE, POINT, POIN, POI
- PORT, POR
- PSTEP, PST, PST, P
- R
- RAH
- RAL
- RAX
- RBH
- RBL
- RBX
- RCH
- RCL
- RCX
- RDH
- RDL
- RDX
- REGISTER, REGISTE, REGIST, REGIS, REGI, REG
- RESUME, RESUM, RESU, RES, R
- RF
- S
- SFL
- SI
- SINTEGER, SINTEGE, SINTEG, SINTE, SINT, SIN
- SP
- SS
- STACK, STAC, STA
- STEP, STE, S
- TFL
- TO
- TRUE, TRU
- WORD, WOR
- WPORT, WPOR, WPO
- ZFL
- =
- :
- + (Unary operator)
- - (Unary operator)
- ,
13.5 Attachment II - Debugger Keywords Listed By Class

- Blank
  (blank)

- Breakpoints
  BP0
  BP1
  BP2
  BP3
  BP4
  BP5
  BP6
  BP7
  BP8
  BP9
  BPA
  BPB
  BPC
  BPD
  BPE
  BPF
  ALLbp

- Execution
  GO, G
  CONTinue
  RESume, R
  STEp, S
  PSTep, P

- Flags
  CFL
  PFL
  AFL
  ZFL
  SFL
  TFL
  IFL
  DFL
  OFL
  FLAg

- Input and Output
  PORt
  WPOrt

- Logic Values
  FALse
  TRUe
- Memory
  BOOlean
  BYTE
  INTeger
  POInter
  SINteger
  WORd

- Operands
  =
  :
  + (Unary operator)
  - (Unary operator)

- Qualifiers
  LENgth
  TO

- Register
  REGister
  AL, RAL
  CL, RCL
  DL, RDL
  BL, RBL
  AH, RAH
  CH, RCH
  DH, RDH
  BH, RBH
  AX, RAX
  CX, RCX
  DX, RDX
  BX, RBX
  SP
  BP
  SI
  DI
  ES
  CS
  SS
  DS
  IP
  RF

- Stack
  STAck

- Utility
  EXIT
  LOAd
  LOG

- 166 -
13.6 Support Policy

I will investigate problems reported to me under the following conditions:

- Notification should be made via the system mail utility to user WAC1751 describing the problem.

- Mail will be examined at least once per week for reports of malfunctions.

- The author of the Simulator86 program is not responsible for any student failing to complete an assignment as a result of a malfunction of the Simulator.

- Include all symptoms in the problem description.

- A short test program which demonstrates the problem (typically 5 to 10 lines of source code) is required for a rigorous investigation. The test program should be mailed to WAC1751 along with a log file of the Debugger commands used to demonstrate the problem.

- Resolution of bonafide problems is not guaranteed within a specified time frame.

- Comments regarding improvements to the user interface will be taken under consideration.
13.7 Maintenance Capabilities

- Memory space, number of I/O ports, amount of I/O data, file name length and terminal port designator can be varied by editing the appropriate definition files, recompiling and relinking the Simulator programs.

- The Simulator executable image is about 160 blocks in size.

- Three utilities are available for maintenance:

  A Command build file - walks the maintenance person through the building of the three main programs and two support utilities.

  Opcode Look Up Table Generator - for Opcode Look Up Table generation and maintenance.

  Debugger Keyword Look Up Table Generator - for Debugger Keyword Look Up Table maintenance.

- The Build Command procedures assumes all definition files in the parent directory.

- The maintenance person can modify the spelling of any keyword, provided it doesn't conflict with other keywords or a single, double or triple hexadecimal character number (i.e. ACE is not a valid keyword). Note that it is the responsibility of the maintenance person to insure that conflicts do not exist. No checks are done by the program to inhibit invalid or duplicate entries.
The Simulator is limited to VAX Pascal and its extensions except for:

Three features which use the VAX Utilities and $QIOW functions.
- Support of Control D
- Keyboard input
- CRT output

Checking for the existence of a file. This feature uses a FORTRAN Subroutine.

The Simulator assumes the Opcode Look Up Table File and the Debugger Keyword Look Up Table File in the same directory as the executable module.
13.8 Release Notes For The Intel 8086 Simulator

The following problems have been corrected:

- The implementation of CBW has been corrected. The investigation has surfaced other "bugs" associated with storing 8 bit registers, which have also been corrected.

The following changes have been made:

- The format of the LOG instruction has changed.
  * LOG <CR> displays the name of the log file
  * LOG = <CR> turns off logging of commands
  * LOG = filename <CR> writes Debugger commands to the file called "filename"

- The format of the LOAd instruction has changed.
  * LOAd <CR> displays the name of the file loaded
  * LOAd = filename <CR> loads the specified file into memory

- An invalid file name or VMS path name should no longer crash the Simulator. Invalid file names should instead generate an error message and allow you to input a correct file name or path name. Invalid file formats (such as reading a data file instead of a map file) will still crash the Simulator.
Appendix G
Unsupported DEBUG-88 Commands

The following is an alphabetical list of the DEBUG-88 commands which are not supported:

* ASM
* COUNT, WHILE, UNTIL
* DEFINE
* DOMAIN
* EVALUATE
* EVALUATE LINE
* EVALUATE SYMBOL
* IF, THEN, ORIF, ELSE
* LINE
* MODULE
* RADIX
* REMOVE
* REPEAT, WHILE, UNTIL
* SYMBOL
Bibliography


3. HP-16C Computer Scientist Owner's Handbook April 1984


5. MCS-86 Assembly Language Reference Guide 9800749A October 1978


7. Introduction to Switching Theory and Logical Design, by Fredrick J. Hill, Gerald R. Peterson

8. VAX-11 Pascal Language Reference Manual (Order No. AA-H484C-TE)


10. Intel Debug-88 User's Manual (Order Number 121758-001)

11. Intel iAPX86,88 Family Utilities User's Guide (Order Number 121616-004)

12. Intel iSDM86 System Debug Monitor Reference Manual (Order Number 146165-001)

13. Intel MCS-86 Absolute Object File Formats (Order Number 9800821A)
14. Intel 8086 Relocatable Object Module Formats (Order Number 121748-001)

15. Utilizing VMS Features from VAX Pascal, Lecture Guide EY2321E-LG0001

16. Utilizing VMS Features from VAX Pascal, Language Specific Workbook EY-2320E-WB-0001

17. Motorola M6800 Programming Reference Manual M68PRM(D)

18. VAX/VMS Internals and Data Structures – by Lawrence J. Kenah & Simon F. Bate (Order No. EY-00014-DD)


22. Rochester Public Library, Rundel Branch, Science & Technology Division