INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 1336555

Meta assembler and emulator for the Intel 8086 microprocessor

Shoaib, Rao Mohammad, M.S.

The University of Arizona, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

META ASSEMBLER AND EMULATOR
FOR THE INTEL 8086 MICROPROCESSOR

BY

RAO MOHAMMAD SHOAIB

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING

In The Graduate College

THE UNIVERSITY OF ARIZONA

1989

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permissions, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the dean of the Graduate college when in his or her judgement the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: _____

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below :

Theodore L Williams — April 14, 1989

Theodore L Williams — DATE

professor

Electrical & Computer Engineering.

ACKNOWLEDGMENTS

The author wishes to express his appreciation to his advisor and committee chairman, Professor Theodore L Williams, for his guidance, helpful comments and support and Professors F.J.Hill and J.V.Wait, for their suggestions and evaluation during the course of this study. I would also like to extend my appreciation to all my family members for their constant support, and to my friends for their constructive comments during the preparation of this thesis.

TABLE OF CONTENTS

ABSTRACT

The thesis describes a Universal meta cross assembler and an emulator for the Intel 8086 microprocessor. The utility is designed to be used as an instructional tool to teach assembly language programming to students. One implementation is available to allow students to run Intel 8086 programs on the university's vax mainframe, so that students can test their programs at their convenience. This setup also results in low operating costs with no additional equipment requirements. Several options are provided in the emulator to debug the 8086 assembly language programs composed by students. The assembler, besides generating Intel 8086 machine code, has the capability to generate machine code for a number of microprocessors or microcontrollers. The machine code file generated by the assembler is the input to the emulator. Both the

assembler and the emulator are completely portable and can be recompiled to run on any system with a standard C compiler.

CHAPTER 1.

INTRODUCTION

The thesis describes a software instructional utility for teaching assembly language programming to students. The utility consists of a universal meta cross assembler and an emulator for the Intel 8086 microprocessor. The utility is implemented on the University of Arizona's computer facility.

Every educational institution requires a lot of equipment and man power to run a lab course. A typical setup for such a course uses a computer built around a specific microprocessor (eg IBM PC with Intel 8086) or to use a hardware emulator. The department must maintain this equipment at additional expense. Further, limitations in manpower make it difficult to allow access

to the laboratory 24 hours each day. Extensive resources are required to replace this equipment as the current technology advances.

The possibility of using a software emulator (a program which performs the instructions of the specific microprocessor) was explored. As most universities have a central computing facility which the students can use 24 hours a day, the students can access the emulator through this facility without any additional maintenance. The multiuser environment of the computing facility allows multiple users to access the same emulator program. The file protection facility allows the instructor to keep his solutions in a private area. It also allows the instructor to provide feedback and grades to each student privately.

However, a software emulator cannot be used to study peripheral devices and interfacing techniques. A separate facility is required to study these concepts in the laboratory. The software emulator is most useful in the early stages of the design process. Later, the

design can be tested in a more realistic setting, where peripheral devices, including those with hardware interrupts, can be added to the system.

On a mainframe, if the technology changes, the cost to update the software emulator will be very low. The major effort would be devoted to modifications of a small portion of the emulator.

The emulator, developed in C, is very modular and simple so that changes to emulate other microprocessors can be easily incorporated.

Based on the above facts, a software emulator for Intels' 8086 microprocessor was selected since the IBM PC is built around the Intel 8086. Thus, the students gain practical experience while learning assembly language programming.

A meta assembler was also written in C . It generates machine code from the assembly language source code written by the students. This machine code is the

input to the emulator. As it is a meta assembler, it has the capability to develop machine code for nearly any microprocessor.

The emulator has the following features :

1. A 64K portion of the host machine's memory is reserved for the emulator. The target machine's register values are also kept in the memory of the host machine.

2. All registers of the microprocessor may be initalized to any value before emulation begins.

3. The user may (1) single step the program (2) examine or change the contents of a memory location (3) examine or change the contents of a register.

4. The user can also set a breakpoint. The breakpoint stays enforced until the user specifically removes it.

5. The last instruction executed is printed on screen.

6. The microprocessor registers and flags are always displayed on the screen as the emulation proceeds.

7. The user can set the instruction pointer to any value, to repeat instructions with a different set of data or to start at any point in the program.

The Meta Assembler has the following features :

1. It produces machine code for numerous different target processors.

2. It is a two pass assembler and hence allows forward references.

3. It supports conditional assembly to reconfigure a single assembly language program for different hardware environments.

4. It supports numeric expressions made up of labels, constants and brackets combined with logical operators or arithmetic operators.

The rest of this document is organized in the following manner :

Emulator : Chapter 2 describes the function and working of the emulator, the implementation is also discussed.

Meta Cross Assembler : Chapter 3 describes the function and working of the Meta Cross Assembler. Data structure and implementation issues are also discussed at the end of chapter 3.

Emulator Manual : Chapter 4 describes (1) how to run the emulator, (2) what the different fields on the screen mean, (3) the available options.

Assembler Manual : Chapter 5 describes the syntax of the assembly language source file and the various assembler directives. It also explains the procedure required to create a machine code file. This section can serve as a student tutorial.

Instruction Table : Chapter 6 describes how to modify the assembler for a different instruction set. This section can serve as a manual for students who wish to modify the assembler for a new instruction set. Notice that the assembler itself does not have to be changed.

CHAPTER 2.

EMULATOR

2.1. INTRODUCTION :

A software emulator is a program that generates the environment of another microprocessor on the host machine. It is used to run programs written for a specific target processor on a different host microprocessor. For example, by using this emulator, an Intel 8086 program can be run on any computer which supports standard C, such as the Digital Equipment VAX or the Apple Macintosh.

The 8086 emulator written in C is the heart of this instructional utility. It reads the machine code file (generated by an assembler) into a 64K memory area, decodes each instruction, and executes the instructions, one by one, updating the microprocessor registers and memory contents.

The emulator is very modular and sufficiently simple so that it can be easily modified to emulate other microprocessors and microcontrollers.

2.2. INPUT REQUIREMENTS :

Input to the emulator is the machine code file (assm.res) generated by the assembler (described later). The student writes his program in assembly language which is translated into machine code by the assembler. For example, if the user writes the following assembly language source program named foo.asm:

```
Mov  ax, #4        ;# is for immediate data.
Add al,  #2
NOP
```

The machine code will be stored in the ASCII text file "assm.res". The file will contain :

```
b8
04
00
```

```
04
02
90
```

If the user has the machine code available, he does not need to run the assembler. He can simply use a text editor to create a file in the above format and pass the file on to the emulator. The format of the machine code file is one byte per line. No spaces are allowed.

2.3. STRENGTHS & LIMITATIONS:

There are several options available to the user while he is emulating the execution of his assembly language program such as single stepping, breakpoints, setup, etc. (see the emulator manual for details). These options are helpful in debugging programs.

Most students are accustomed to writing 8086 programs using the IBM PC, with software interrupts to access the operating system functions (DOS). As the emulator does not support MSDOS function calls, these calls should not be included in a program written to run under this emulator.

Since this is a software emulation of the Intel 8086, no hardware interrupts can occur and multiple input and output devices cannot be supported.

The only input is from the keyboard and the only output is to the user's terminal. For an IN instruction, the input is taken from the keyboard regardless of the input port number. For an OUT instruction, the output is sent to the users terminal regardless of the output port number. However, in both cases the emulator displays the port number so that the student can check the operation of the program.

The INT (software interrupt) instruction of the intel 8086 is supported by the emulator; however, an interrupt handling routine must be included and its address should be stored in the vector table as explained below.

For example, the INT 12h (h is for hexadecimal) instruction represents a type 12 interrupt. For this type of interrupt, the offset of the interrupt service routine should be stored at absolute word memory location

0048h (Int type *4) and the segment value at absolute word memory location 004Ah (Int type *4 + 2). When an INT 12h instruction is encountered, the processor flag, instruction pointer and the code segment register is loaded with the value stored at word memory location 4Ah and the instruction pointer with the value stored at word memory location 48h and the next instruction is fetched from this location.

2.4. UTILITY & PORTABILITY:

The emulator is written in the C language and is easily portable to other systems with a standard C compiler.

Most microprocessors have some common set of instructions with different opcodes and mnemonics. Since each instruction of the Intel 8086 is implemented as a separate routine, the old routines can be used to handle the operation by changing the decoding routine. Some microprocessors may set the flags differently then the 8086, however the flag setting can be easily changed as separate routines are used for setting each flag.

There are separate routines to fetch the operands and store the result. These routines can be changed if the addressing scheme changes. In the Intel 8086, a word is stored with the low byte first. If this is not the case for the new microprocessor, then the routines which fetch each byte and assemble the word must be changed accordingly. The storing routine must be similarly changed.

2.5. STRUCTURE & WORKING OF PROGRAM :

The structure developed to write the program is patterned after Intel 8086 microprocessor structure. Only the prefetching of the instructions and the arithmetic logic unit are omitted from the emulator. For every register of the 8086, there are three variables, i.e., for AX there is AX, AL and AH. AL and AH contain the lower and higher bytes respectively. Besides the register variables, two variables are declared for the

instruction pointer (IP) and the processor flag (Proc_flag). Each variable is updated as each instruction is executed.

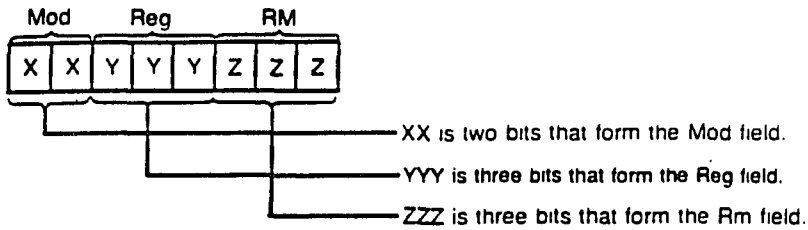
After the initializations and option selections are complete, the G command starts executing instructions. The byte stored at the memory location pointed to by the IP is fetched as the opcode of the first instruction and IP is incremented.

The opcode is used as an index to get a function pointer from an array which points to a routine handling this instruction. Since the Intel 8086 supports multiple addressing modes, depending upon the instruction, it may be necessary to fetch another byte to determine the addressing mode. More bytes may be fetched as operands. For example, if the first byte is 90h, then it is simply a NOP instruction. But if the opcode is 88h then all that is clear is that it is a MOV instruction. The next byte has to be fetched to find out the addressing mode. This byte is broken into three parts to specify the operands. Table 1 describes how this byte determines the operands. Once the operands are fetched, the memory contents and microprocessor flags are updated. If single

step mode is set or a breakpoint is set, control is returned to the user. Otherwise , the next byte is fetched from where the address found in the IP register, and emulation continues.

2.6. PROBLEMS ENCOUNTERED :

Only one major problem was encountered after the structure of the program was selected. The problem involved setting the auxiliary carry flag and the overflow flag. The problem was resolved by resorting to bit-by-bit twos complement addition.



Mod= 00 No displacement bytes.
01 One signed displacement byte.

Mod Reg Rm	Displacement

10 Two displacement bytes (unsigned 16-bit).

Mod Reg Rm	Displacement	Displacement
	Low Byte	High Byte

Reg=

	Byte	Word
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Rm=

	Mod=00	Mod=01 or Mod=10	Mod=11 Byte	Mod=11 Word
000	BX+SI	BX+SI+Displacement	AL	AX
001	BX+DI	BX+DI+Displacement	CL	CX
010	BP+SI	BP+SI+Displacement	DL	DX
011	BP+DI	BP+DI+Displacement	BL	BX
100	SI	SI +Displacement	AH	SP
101	DI	DI +Displacement	CH	BP
110	Direct	BP+Displacement	DH	SI
111	BX	BX+Displacement	BH	DI

CHAPTER 3.

THE META ASSEMBLER

3.1. INTRODUCTION :

The input to the emulator is a file containing the machine code in ASCII format. In order to develop this machine code file from assembly language source code, a meta cross assembler was developed.

An assembler reads the assembly language source file and generates the machine code for the target machine. Normally, an assembler develops machine code for just one microprocessor, but a meta assembler develops machine code for a set of machines. The input to the assembler consists of an assembly source file and an assembler instruction table (described later). The output from the assembler is a file (assm.res) containing the machine code in ASCII text format, one byte per line.

3.2. INPUT REQUIREMENTS :

The input to the assembler is the assembly language source file written by the user. The source file can be made using any text editor. If the file does not have the right format, an error will be printed and the assembler will stop.

The assembly source file is free format with just one instruction per line. Comments start with a semicolon and continue to the end of the line. Blank lines are also permitted.

The assembler has several directives which are described in the assembler manual. As noted in the assembler manual, some instructions are modified so that they indicate the size of the operands. For example, MULB is used for byte operands and MULW for word operands. Immediate data should be preceded by "#" for example :

```
MOV AL, #8          ;put eight in AL.
```

3.3. STRENGTHS & LIMITATIONS :

The assembler is based upon the cross 16 meta assembler [3]. The assembler is very powerful in the sense that a different microprocessor may be accomodated by just changing the instruction table. The user can even make changes in the instruction table to change the mnemonics to suit his needs. If the user decides to use another microprocessor, he does not need to modify the assembler source code.

This is a two pass assembler which allows forward references. That is, the user can use labels in an operand before they are defined. This feature is essential for forward branches in the program. All labels, symbols, and their corresponding values are placed in a hash table as they are encountered on the first pass. All symbols and labels not resolved after the first pass are assigned a zero value and a warning

message is printed. If a label or symbol is multiply defined, the present value overrides the last value. An error message is printed but the assembly continues.

The assembler accepts numeric expressions made up of labels, constants and brackets, combined with logical and arithmetic operators. Any expression which results in an overflow is indicated and the assembler stops.

```
MOV AX, #(4 * 2) / (2 + 2) ;# is immediate data
```

The assembler does not support relocation, linking, macros or libraries.

3.4. UTILITY & PORTABILITY :

The assembler is written in the C language and may be transferred to another system that has a standard C compiler.

Software development is flexible in that the user can develop the software for several target machines on a single host with powerful resources such as editors, printers, debuggers and an elaborate operating system. These facilities aid both the student and especially the instructor who must manage the course administration and respond to questions.

3.5. INSTRUCTION TABLE :

The assembler uses the instruction table as a look-up table for instructions.

The instruction table contains all the opcode mnemonics of the particular microprocessor sorted alphabetically. In case of multiple byte instructions, the shorter instruction is at the beginning of the table, i.e., the Intel 8086 has two types of ADD instructions, namely 8-bit and 16-bit with different opcodes. when written in assembly language, the same mnemonic is used for both. When an instruction is found in the table, its operands are checked to see if they are 8-bits or 16-bits

wide. The assembler then selects the appropriate width or generates an error message if the operands are inconsistent.

3.6. STRUCTURE & WORKING OF THE PROGRAM:

Since the assembler has to search the instruction table for the opcode mnemonic of each instruction, a fast search is essential. Due to the format of the instruction table, many search strategies cannot be used. In the present implementation, instructions are put in a hash table. The hash value of a mnemonic computed over the first three characters of the mnemonics is divided by three. Computing the hash value over the whole mnemonic will not work. For instructions like :

```
ADD AL, BL
```

the assembler searches for the hash value of ADD. However in the instruction table, the mnemonic is stored as ADDAL,{...}^...: and the hash values will be

different if more than three characters are included, division of the hash value by three was found to be sufficient to spread the mnemonics throughout the table.

The assembler reads the assembly language source file twice. After each line is read from the file, the line is stored in a character array. While storing the line (1) comments are removed, (2) all characters are changed to upper case unless they are in quotes, (3) an error is flagged if the line is more then 128 characters long.

If a line is a comment or a blank, it is discarded, the line number is incremented, and the next line is fetched. Otherwise, the first word in the line is tested to see if it is a label, an assembler directive, or an instruction. If it is any one of the above, the appropriate routine is called. Statements may contain a numeric expression which is evaluated using the algorithm on page 109 of [1].

This algorithm is very flexible and the precedence of operators can be easily changed by changing the weight assigned to them. The evaluation of the expressions

could be done by parsing as described in chapter 2 of [3]. But this would not have been as flexible as the scheme used here.

While evaluating expressions, the hash table is searched when a label is encountered. If the label is not found, it is assigned a zero value (it might be a forward reference). In the first pass no error message is printed; however, in the second pass, if the label is not found, then an error message is printed. The assembly continues assuming a zero value.

If the line is not an assembler directive, but an instruction, the line is put in an array by removing all the spaces between words. The mnemonic of the instruction is hashed (first three characters only). Based on this value, the instructions are fetched from the hash table. The instruction from the source file is compared with the instructions in the hash table. If no match is found, an error message is printed and the assembly stops. if a match is found, then the operands are checked to see if they can be represented within the number of bits allowed (as explained earlier in the instruction table section). If not, the search proceeds

further until an instruction is found which satisfies the condition. At this point, the opcode is generated from the instruction table and the operand values as required by the type of operand. The final opcode is written in the assm.res file and the byte count is incremented. If an operand is found which does not fit within the allowable number of bits, an error message is generated.

It is important that the instruction table be ordered with the narrow instructions first, or else, even for 8-bit operands, 16-bit instructions will be selected.

CHAPTER 4.

CONCLUSION :

This thesis describes a Universal meta-cross assembler and an emulator for the Intel 8086 and other microprocessors. These instructional tools are an aid in teaching assembly language programming to engineering students. One implementation allows students to exercise their Intel 8086 programs on the university's VAX mainframe.

This facility allows students to work at any time yet it requires no additional departmental funding. In addition, the full facility of a mainframe computer are available to the instructor to administer the course.

The meta-assembler is capable of generating assembly code for a variety of microprocessors. Only a single lookup table must be modified, no changes in the assembler code is required. Both the assembler and emulator are easily moved to any system with a standard C compiler.

CHAPTER 5.

EMULATOR MANUAL :

The emulator can be invoked by typing

```
r 8086.emu
```

The screen will be cleared and another screen appears that is divided into sections with headings showing the options available to the user.

The different sections of the screen are explained below:

HELP MESSAGES :

This section of the screen always shows the options available to the user. Options can be selected by typing the letter shown in front of that option. The program does not distinguish between upper and lower case. In

the "S" (examine variable) and "A" (alter variable) options, the variable names are the same as their register names in the 8086 except for the following:

PF for Processor Flag.

M for Memory.

B for Breakpoint.

It should be noted that the breakpoint is set using the B command. In order to examine the value of a breakpoint, select the S mode, then select the variable named B.

When examining or changing memory contents, the user is required to give the address of the memory location. The return key selects the memory location zero. While examining a location, the return key will just increase the memory location by one. The "S" and "A" options can be terminated by typing E or e. All data and memory locations are represented in hexadecimal format. The "H"

suffix is not required. In case the first digit is a letter, a leading zero is required in order to distinguish it from a symbol.

In case the screen becomes garbled, the R command refreshes the display.

ERROR / INFORMATION MESSAGES :

This section of the screen displays error messages. For example, if the user specifies a file to be loaded using the "L" option and the file does not exist, then an error message will be printed in this section. Similarly, if the user selects the single step mode, then a message saying that the single step mode has been selected is printed here.

MICROPROCESSOR REGISTERS :

This section of the screen displays the various microprocessor register values at any time.

DISPLAY VALUES :

This section of the screen displays the register values and memory locations. When running the program, the last instruction executed is printed in this section.

STATUS :

This section of the screen displays the current options. If single step mode is set, a "C" will be displayed. Similarly, a "B" is displayed if a breakpoint is set.

INPUT VALUES :

Values are entered by the user in this section. Once the user has selected an option from the screen, he will be prompted for a command.

The registers are initalized using the "A" option, and files are loaded using the "L" option. The single step mode or a breakpoint may be set before the "G" option starts the execution of the program. In single step mode, control is returned to the user after one

instruction. Changes in register contents or memory locations may be made before restarting the program. In single step mode, the user must type "G" after every instruction.

If a breakpoint is set, control is returned to the user when the break point is reached. The user must type a "G" to resume execution of the program.

If the emulator is not in single step mode and if no breakpoint is set, execution proceeds until a NOP instruction is reached or until an illegal instruction is encountered. It is suggested that the program be ended with a NOP instruction. Otherwise, the execution may continue past the last instruction destroying register and memory values.

CHAPTER 6.

ASSEMBLER MANUAL :

6.1. RUNNING THE ASSEMBLER :

The assembler can be invoked by typing :

```
$ r assm
```

The assembler then asks the user for the assembly language source file name.

```
$ Enter assembly language file name.
```

If the file does not exist, an error message is printed and the assembler stops. otherwise, the assembly begins. The assembler indicates the current pass on the display. If the assembler cannot find an instruction, or if there is an error in the format of a directive, an error message is printed. The line where the error occurred is displayed preceded by the line number and

assembly stops. If the value of a label or constant cannot be found during the first pass, an error message displays the name of the variable. The value of the label is arbitrarily set to zero.

The remainder of this section describes the various directives and their format. At the end of this section a assembly language file is included as an example.

6.2. ASSEMBLY SOURCE FILE :

An ASCII text file containing the source code serves as the assembler input. It contains one instruction per line. It also tells the assembler which instruction table to use, while constructing the machine code.

6.3. ASSEMBLY LINE FORMAT :

Only one assembly instruction or assembler directive is permitted per line. The assembly line is free format. For example, labels need not start in column1. Each line may contain some or all of the following elements:

```
Labels:    operation    operand(s)    ;comment
```

6.3.1. **LABEL:**

A label is a word starting with an alphabetic character or an underscore and ending with a colon, which is assigned the present value of the program counter or other user specified value in the range -2,147,483,648 to 2,147,483,647. Characters within the label must be alphanumeric or an underline and must end with a colon. They can be of any length not exceeding 128 characters. The assembler does not distinguish between upper and lower case. A label may stand alone on a line, in which case it will be assigned the current value of the program counter.

6.3.2. **OPERATION:**

An operation is an assembler directive, or processor assembly instruction defined in the instruction table. All operations must start with two alphabetic characters.

6.3.3. **OPERANDS :**

Operands are labels, constants or expressions representing integer values in the range -2,147,483,648 to 2,147,483,647 and / or character strings. They can be used as operands in an assembly language instruction.

6.3.3.1. **NUMERIC CONSTANTS :**

A numeric constant is an ASCII representation of a 32-bit signed integer in decimal or hexadecimal format. All numeric constants must begin with a number, hexadecimal representation is indicated by a trailing H, the default base is 10.

Some examples of numeric constants are

```
DEC1:     EQU -1          ;DECIMAL
HXE1:     EQU 0ffh        ;Hexadecimal
```

6.3.3.2. **STRING CONSTANTS :**

String constants consist of a series of ASCII characters between two quotation marks. The assembler will convert these constants to a hexadecimal representation of their ASCII values. Lower case characters within string constants will be represented as such. A quotation mark cannot appear within a character string, as it will be interpreted as the end of that string. A string constant may also be used as an operand where applicable. In a DFB statement, a string constant may be of any length, bearing in mind that an assembly source file must not exceed 128 characters in length. When a string constant is used as an operand in the DWM, DWL, DFL or EQU directives, an error will be flagged if the string constant exceeds the length of the operand specified by the instruction. A string constant cannot be extended beyond its present line.

```
DFB       "YEA , YEA , YEA!"
```

6.4. COMMENTS :

A comment is a sequence of characters starting with a semicolon upto and including the end of the line. Comments are ignored by the assembler. A comment may start anywhere on a line.

```
ADD al , bl        ;comment following instruction.

;  This is a comment on a separate line.
```

6.5. EXPRESSIONS :

The assembler will accept numeric expressions made up of labels, constants and brackets combined with logical and arithmetic operators. A list of recognized logical and arithmetic operators in their relative precedence is shown below , where X and Y represent integer values:

```
Express1:   EQU  40 - 20          ;subtraction
Express2:   EQU  10 MOD  3        ;remainder
Express3:   EQU  (10*3)/(30/3)    ;little complex
```

The assembler recognizes the following logical and arithmetic operators. Operators are listed in order of descending precedence :

()	parentheses
~Y	Ones complement of Y.
X * Y	Integer multiplication of X and Y.
X / Y	Integer division of X by Y.
X mod Y	Integer remainder after division of X by Y.
X < Y	Logical shift left of X by Y bits and filled from the right with zeros.

X > Y	Logical shift right of X by Y bits and filled from the left with zeros.
X + Y	Integer addition of X and Y.
X - Y	Integer subtraction of Y from X.
X & Y	Logical AND of X and Y .
X | Y	Logical OR of X and Y.
X ^ Y	Exclusive OR of X and Y.

6.6. ASSEMBLER DIRECTIVES :

The following directives described in the following pages are recognized by the assembler :

CPU	CPU table declaration.
DFB	Define Byte.

DFL	Define long integer.
DWM	Define Word (high byte first).
DWL	Define Word (low byte first).
END	End of program.
EQU	Equate label to permanent value.
IF, ELSE, ENDI	Conditional Assembly.

All the define type directives reserve and initalize the memory contents, except for the DFS directive, Which only reserves memory and initially clears this area of the memory.

6.6.1. CENTRAL PROCESSING UNIT TABLE DECLARATION

The CPU directive tells the assembler which processor instruction table is to be used during assembly.

The CPU directive has the following syntax :

```
Label:  CPU        "cpu_file_name"     ;comment
```

Only the instruction table file name may be specified with the CPU directive. A label may be placed before the CPU directive, which will be assigned the current value of the program counter. Multiple CPU directives will result in successive changes in the instruction table. A nonexistent file name will stop execution of the program after printing an error message.

```
CPU                "8086.tbl"

;  use 8086 instruction table
```

6.6.2. DFB : DEFINE BYTE

The DFB directive allows the user to define the value of storage areas on a byte-by-byte basis.

The DFB directive has the following syntax:

```
Label:  DFB expr1,expr2,....,expr(n) ;comment
```

Except for a string constant, the result of each expression must represent an 8-bit value or an error will be flagged. An expression may consist of a numeric constant, a string constant, a label, or a formula. There is no limit on the number of bytes that may be defined using a single DFB directive, with the proviso that the length of the input line must not exceed 128 characters.

```
DFB       "HELLO HOW ARE YOU"      ;ASCII string

DFB       0 , 1 , 2                ;Integers

DFB       "HI THERE", 1 , 2    ;string and integers
```

6.6.3. DFL : DEFINE LONG INTEGER

The DFL directive allows the user to define the value of storage areas on a long integer or long word basis (a long integer is 32-bits wide). There is no

limit on the number of long integers that may be defined using a single DFL directive, with a line length of less than 128 characters.

The directive syntax is as follows :

```
Label:  DFL expr1,expr2,.....expr(n)     ;comment
```

The value of each expression must be representable using a 32-bit signed integer or an error will be flagged and the program will terminate. An expression may consist of a numeric constant, a string constant, a label, or a formula. Although ASCII constants may be used, an error will be flagged if they exceed 4 characters. The DFL directive initalizes and reserves memory for 32-bit signed integers.

```
CONST1:   DFL     -2        ;Numeric constant

          DFL     CONST1    ;Label
```

6.6.4. DFS : DEFINE STORAGE

The define storage (DFS) directive may be used to reserve a section of memory with unspecified contents during assembly.

The DFS directive has the following syntax:

```
Label:    DFS     expression      ;comment
```

The expression can be of any form which represents a 32-bit positive integer value.

```
STOR_B    DFS     20*1            ;Reserve 20 bytes

STOR_C    DFS     20*2            ;Reserve 20 words
```

6.6.5. DWL : DEFINE WORD (Least significant byte first)

The DWL directive allows the user to define the value of storage areas on a word by word basis (one word represents two bytes). There is no limit on the number of words that may be defined using the DWL directive, except that the length of line must not exceed 128 characters.

The DWL directive has the following syntax :

```
Label:   DWL  expr1,expr2,...,expr(n)
```

The result of each expression must represent a 16-bit integer or else an error message will be printed and the program will terminate. An expression may consist of a numeric constant, a string constant, a label or a formula. Although ASCII string constants may be used, an error will be caused if they exceed two characters in length. The DWL directive will store the least significant byte of the 16-bit value before the most significant byte. By using either the DWM or DWL

directives, words may be placed in memory in a format which corresponds to the format used by the target processor.

```
DWL       0 , 1 , 2        ;Numeric constant

DWL       3FH , 44H        ;Hexadecimal
```

6.6.6. DWM : DEFINE WORD (Most Significant Byte First)

The DWM directive allows the user to define the value of storage areas on a word-by-word basis. There is no limit on the number of words that may be defined using a single DWL directive, except that the length of a line must not exceed 128 characters.

The DWM directive has the following syntax:

```
Label:  DWM expr1,expr2,...,expr(n) ;comment
```

The result of each expression must represent a 16-bit integer value or an error will be flagged. An expression may consist of a numeric constant, a string constant, a label or a formula. Although ASCII string constants may be used, an error will be flagged if they exceed 2 characters in length. The DWM directive will store the most significant byte of the 16-bit value before the least significant byte. By using either the DWM or DWL directives, words may be placed in memory in a format that corresponds to the format used by the target processor.

```
DWM       0 , 1          ;Numeric constant

DWM       0FFFFH         ;up to 16-bits
```

6.6.7. **END : End Of Program**

The end of assembly (END) directive is optional, but when used, it will be the last line of the assembly source file assembled.

This directive has the following syntax :

```
Label:          END          ;comment
```

6.6.8. **EQU : EQUATE**

The equate (EQU) directive assigns an integer value to a label.

The EQU directive has the following syntax :

```
Label:    EQU     expression      ;comment
```

Neither the label nor the expression are optional in this directive. The expression may consist of any numeric constant, character string or formula whose value can be represented in 32-bits. Defining the label more then once will change the value of the label.

```
CR:       EQU     13h     ;ASCII Carriage return

LF:       EQU     10      ;ASCII Line Feed
```

6.6.9. IF , ELSE , and IF : Conditional Assembly

The assembler supports conditional assembly using the IF, ELSE and ENDI directives to define areas of the source file which are not to be assembled. This feature is usually used to reconfigure a single assembly language program for different hardware environments.

Conditional assembly has the following syntax:

```
IF        expression      ;comment

          line 1
          line 2
            .
            .
            .
          line n

ELSE                      ;comment

          line 1
          line 2
            .
            .
          line n

ENDI                      ;comment
```

Upon encountering an IF statement the assembler evaluates the single expression following it. All labels used in this expression must be defined prior to the IF.

if the expression evaluates to zero, the statement between the IF and either an ELSE or an ENDI are not assembled. The else is an optional directive, allowing just one of the two sections of the source file within the IF block to be assembled. All conditional blocks must have an IF directive and an ENDI directive, the else directive being optional. IF blocks may be nested.

6.7. SAMPLE ASSEMBLY FILE

Following is a sample assembly language file foo.assm

```
;This program reads byte values stored at memory
;labeled "data" and then stores them at memory
;labeled "store"

CPU "8086.tbl"          ;use 8086 Instruction table
jmp start
data:   dfb  1 , 3 , 6
```

```
store:  dfs 2*4
start:  mov bx , #data
        mov di , #store
        mov cx , #3
shift:  mov dl ,  [bx]
        mov [di] , dl
        incw bx
        incw di
        decw cx
        jnz shift
        nop
```

CHAPTER 7.

THE INSTRUCTION TABLE

The following section describes the formation of an Instruction table for use by the assembler. This section may be skipped by the user if he or she wants to write assembly language programs for an existing instruction table.

7.1. ASSEMBLER INSTRUCTION TABLE:

The instruction table defines the instructions and registers of the individual microprocessor. The instruction table consists of an alphabetically ordered listing of all the instructions. The table specifies the format, placement of operands, and the corresponding hexadecimal machine language or opcode. Each instruction must be on a separate line and followed by a carriage return. An operand may be placed anywhere within the instruction, with its characteristics enclosed in a set

of script "{}" brackets. The ASCII hexadecimal opcode for the instruction is placed between the carat "^" and colon ":". from this field the assembler determines both the value and length of the instruction's opcode. There are no limits on the length of the instruction or its opcode, except that an input line cannot exceed 128 characters. During assembly, the supplied opcode is converted to a binary value and logically ored with any operands defined within the script brackets "{}". Following the last instruction, an optional asterisk "*" may be used to signify the start of a section of equate assembler directives, which are normally used to define register values. During assembly, these are treated in an identical fashion to an EQU directive in the assembly source file. The syntax of the instruction table is as follows :

```
Instruction                    {operand(S)}^Hexstring :
    .                                  .
    .                                  .

    *
```

```
Label:              EQU              Expression
  .                  .                  .
  .                  .                  .
```

The 8086 Instruction table looks like this :

```
AAA^37:
AAD^D50A:
AAM^D40A:
AAS^3F:
ADCAL,#{8B8}^1400:
ADCAX,#{8I16}^150000:
.
.
.
.
*
AL:   EQU  0
CL:   EQU  1
DL:   EQU  2
BL:   EQU  3
 .
 .
```

```
.
.
AX:  EQU  -8
CX:  EQU  -7
DX:  EQU  -6
```

7.2. OPERAND TYPES

The instructions and operands are combined into a single expression on the left side of the "^". There must not be blanks anywhere in this field. Any blanks that may be specified in the processors assembly language programming manual must be excluded. This will not prevent the user from placing blanks or tabs in the assembly source file. operands within an instruction are optional, and there is no limit to their number.

As stated previously, an operand is defined by placing it in script brackets in its proper location within the instruction. The operand definition, within the brackets has the following syntax:

{starting location operand type operand length}

Starting location is the position of the first bit of the operand within the instruction opcode. This value must be specified as a positive decimal number. The bit positions are numbered from left to right (most significant bit first) starting at 0 and preceding to a maximum of 511.

Operand type is a single character which represents any changes that may be made to the opcode or checks that may be performed before it is placed there.

Operand length is the number of significant bits of the operand which will be placed in the opcode, beginning with the least significant bit. The operand length must be a positive decimal number ranging from 1 to a maximum of 32.

7.2.1. The A operand type

The operand is ored with the opcode as specified by the starting-location and operand length. A range check is performed on the operand during assembly, to insure that it can be represented in a signed twos complement format in the number of bits specified by the operand length. if it cannot, the assembler will proceed further in the instruction table looking for an instruction which will handle the operand. If a matching instruction with a sufficiently long bit length is found this new instruction will be used or else an error will be flagged.

7.2.2. The B operand type

The operand is ored with the opcode as specified by the starting location and operand length. A range check is performed on the value of the operand during assembly, to insure that it can be fully represented in an unsigned binary format in the number of bits specified by the operand length. if it cannot, the assembler will proceed

further in the instruction table looking for an instruction which will handle the operand, else, an error is flagged.

7.2.3. The C operand type

The operand is ored with the opcode as specified by the starting location and operand length. This operand type is identical to the B type except that a range check is not performed. It is necessary to assemble instructions that only make use of less significant bits of the operand.

7.2.4. The D operand type

The operand is ored with the opcode as specified by the starting location and operand length. This operand type is identical to the C type except that a range check is performed to insure that its value falls between 0F0H and 0FFH. It is necessary to assemble only the DRSZ REG and LD REG, # instructions of the cop800 family.

7.2.5. The I operand type

The high and low bytes are exchanged in a similar fashion to the DWL directive, and the result is ored with the opcode as specified by the starting location and operand length. Since this operand type is used primarily with instructions that utilize the full sixteen bits of the operand, a range check is not performed.

7.2.6. The L operand type

A relative operand is calculated by the assembler during assembly and its high and low bytes are exchanged in the manner as DWL directive and the I operand type. The result is ored with the opcode as specified by the starting location and operand length. This relative value is the value of the operand specified in the assembly language source file, minus the memory address of the byte immediately following the opcode. A range check is performed during assembly, to ensure that the calculated operand can be fully represented as a singed twos complement number in the number of bits specified in

the operand length. If this condition is not true, the assembler will look for a matching instruction with a longer operand length, else an error will be flagged.

7.2.7. The P operand type

The page operand type is used primarily by the 8048 and 8051 family of microcontrollers for their 11-bit call and jump instructions. The page operand is similar to the C operand type except that bit 10 through 8 of the operand specified in the assembly language source file are ored with bits 0 through 2 of the opcode, respectively. The operand is also ored with the opcode as specified by the starting location and operand length. Unlike the C operand type, a range check is performed to insure that the operand is on the same 2K page boundary as the program counter. If not , an error will be flagged.

7.2.8. The R operand type

A relative operand is calculated by the assembler during assembly, and the result ored with the opcode as specified by the starting location and operand length. This relative value is the value of the operand specified in the assembly language source file, minus the address of the byte immediately following the opcode. A range check is performed during assembly, to ensure that value of the calculated operand can be fully represented in a signed twos complement format in the number of bits specified in the operand length. If this cannot be done assembler will look for a matching instruction with a longer operand length. This procedure enables assembler to automatically select the shortest opcode for an instruction. Many processors use different mnemonics for the relative and absolute branching instructions. The user may wish to make these different mnemonics similar so that the assembler will select the shortest instruction.

7.2.9. The S operand type

This operand type is identical to the R type except that the relative operand is calculated as the operand value specified in the assembly source file minus the address of the last byte of the instruction. The relative operand is calculated during assembly and the result ored with the opcode as specified by the starting location and operand length. A range check is performed to ensure that value of the calculated operand can be fully represented in a signed twos complement format in the number of bits specified by the operand length. If this cannot be done, the assembler will look for a matching instruction with a longer operand length, else an error will be flagged.

7.2.10. The T operand type

This operand type is identical to the R type except then the relative operand is calculated as the operand value specified in the assembly source file minus the

address of the second to last byte of the instruction. The relative operand is calculated by the assembler during assembly, and the result ored with the opcode as specified by the starting location and operand length. A range check is performed to ensure that value of the calculated operand can be fully represented in a signed twos complement format in the number of bits specified by the operand length. If this cannot be done the assembler will look for a matching instruction with a longer operand length, or else an error will be flagged.

7.2.11. Register definition

An asterisk "*" may be placed on the line following the last assembly instruction in the instruction table. On subsequent lines, standard equate (EQU) directives may be used to define register values or whatever other labels the user may wish. This definition section is optional and may be placed in the assembly source file instead.

REFERENCES

1. Thomas L. Naps Bhagat Singh, "Introduction to Data Structures with Pascal", West Publishing Company 1986.

2. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers, Principles, Techniques and Tools", Addison Wesley, 1978 .

3. Cross - 16 Meta Assembler User's Guide, Universal Cross Assembler Canada. 1988