DECsystem ASSEMBLER LANGUAGE

An Honors Thesis (ID 499)

By

Beth Moore

THESIS DIRECTOR

[Signature]

Ball State University
Muncie, IN
November 1981
Graduating Fall 1981
In order to take full advantage of the power of assembler language one should know how the information of the computer is held in its words. A number holding unit is called a word and is 36 individual binary digits. Each binary digit (bit) represents one of two states, ON or OFF. These two states, and the many combinations that each word can utilize of these states, is the communication between the user and the machine.

In this case, though the computer is readily able to deal with numbers in binary form, it also can easily deal with numbers in octal form, in which the base is eight. To convert binary representations to octal representations is quite simple. Take for example the binary number 110100 which is equivalent to the octal number 64. This is true because we group the binary representation in three-digit groups starting from the right side of the number (110100) = (110)(100). Then the conversion is made by replacing the grouped numbers with their decimal equivalent (110) = 6 and (100) = 4.
The various symbols that a user can type on the terminal are encoded by the computer as a number. These numbers, unique to each symbol, are designated as a code...in this case it is the ASCII code (American Standard Code for Information Interchange). On a full standard terminal this will be equal to 127 distinct symbols.

For example, the ASCII code for A is 101 with successive letters one more than the previous letter, but the ASCII representation is in octal notation. The numerals have ASCII code 60 for 0, incrementing this code by one for successive numerals. To output an ASCII character on the terminal the OUTCHR command is used. This command instructs the monitor to send contents of the appropriate word to the terminal as an ASCII character. For example OUTCHR (101) would print the letter A.

Every program written in this language must have an EXIT instruction which informs the monitor to perform certain routine functions to stop the program. This instruction must come before the END statement of the program. Using the OUTCHR com-
mand I will demonstrate a very simple program in DEC assembler:

```
START:       OUTCHR   (101)
            EXIT
            END      START
```

START is only a label which lets the computer know where to commence operations when the program is executed. The END statement points to START...the beginning of the program. EXIT and OUTCHR have already been discussed, so execution of this program will type out the letter A.

When a user creates a program, the program must be given a name. This name may be any collection of up to six letters and numbers as long as the first character is a letter. The extension of the program name should be MAC since this is a macro (assembler language) programming language. So the above program could be named OUTPUT.MAC, and to get the letter A to be printed the command for execution is EX OUTPUT.

There are, of course, several instructions that this language uses, but before I introduce these I would like to talk about the editing feature of the language. It is known as TECO.

To summon the editor to make a new file, a user should type after the period issued by the monitor the command MAKE OUTPUT
(where OUTPUT is used because this is the name chosen for the example program.) The machine will return an * which lets the user know that he is in contact with TECO, the editor.

Now the user is able to insert, amend, delete, etc. code he has written into his program. Before any commands of TECO can be performed, they must be followed by $$, which is the escape key pressed twice, not the dollar sign. To get back into monitor mode, the user should type EX followed by two depressions of the escape key. If the user wishes to return to TECO after exiting from it, he needs to only type TECO OUTPUT (again where OUTPUT is the name of the user's program).

There are several commands that TECO utilizes, so I will only introduce those that are most commonly used. FS is used to change text; such as: FSJUMPS$JUMPed$$ will cause the word JUMP to be changed to JUMPed. One important thing that must be remembered when using TECO is that the user is dealing with a pointer while amending his text. The pointer must be followed in order to know how to reach the required text that needs to be edited.
Whenever a user calls into TECO the pointer is automatically set to the beginning of the already existing file. So if the user wants to search his text, with the S command, he must be made aware of the fact that his text will only be searched from the location of the pointer at that moment to the end of his program. Nothing prior to the pointer will be searched.

To type out a line of text while in TECO, the user can implement the T command, and this command will not move the position of the pointer. The D instruction deletes the next character after the pointer, and K deletes the current line from the position of the pointer to the end of the line, including the carriage return.

I is the command to insert text into the program. For example: *IBALL STATE UNIVERSITY$EX$$ would cause BALL STATE UNIVERSITY to be inserted in the user's program in regard to the position of the pointer when the insertion was made. One must remember that the first $ (escape) is needed to denote that the instruction is what is previous to this $ and for the command to be executed it must be followed by EX$$ (Itext$$EX$$).
To advance the pointer or move the pointer back a specified number of lines the \texttt{nL} command is appropriate to use. One must keep in mind that the end of the line is recognized by \texttt{TECO} as a line feed and not by a carriage return.

After the user completes his editing, he can find out what files he has in his directory and discover that he now not only has an updated file, but he also has a back-up file.

(\textit{in our example it would be OUTPUT.BAK})

Another feature used by the \texttt{DECsystem-10} is \texttt{DDT} (Dynamic Debugging Technique). \texttt{DDT} \texttt{OUTPUT} gets the user in contact with \texttt{DDT} to "debug" his program. \texttt{Control C} gets the user out of \texttt{DDT}.

To see the contents of any location in one's program, one needs only to type the label of the specified line followed by an \texttt{/ (label/)}. When a location has been referenced this location is considered to be opened and its contents can be changed. \"\texttt{/text/} will enter the designated text into whichever word has been opened. Words must be closed after the proper corrections have been made.
A word can be closed with a line feed, a carriage return, an up-arrow, a tab, or a backslash. Each has its own advantages, but I preferred the up-arrow because it goes through one's text of his program line by line and closed the word, moved the pointer on to get the next location, opened that location, and types out that location's contents.

DDT has several features to it, as did TeCO, so I will point out one final aspect of the device before closing my brief discussion on DDT. This feature is the setting of what is known as "breakpoints" in a user's program. This is a very useful tool because it allows execution of one's program to stop at each breakpoint so the user can see what sections of his program are running correctly and which parts need to be corrected.

A breakpoint is set at a location by typing the address of the location (symbolically or as an octal number), then the two characters $B$ (again where $\$ is the escape key). One can assign a total of eight breakpoints in his program. Just typing $B$ removes all of the breakpoints in one's program, and to remove a single breakpoint one needs to type $0\$n$$ (where n
is the number of the breakpoint 1-8 and the $ in each case is the escape key).

Back to the basic fundamentals of the language itself we find that there are sixteen memory locations available to the user. These memory locations are known as accumulators. It is important to note that arithmetic operations can only be performed on a number if it is held in an accumulator.

Using accumulators, one must declare them at the beginning of his program with the number of the accumulator he intends to use assigned to a name. For example ONE=1 would assign the accumulator one the name ONE, and whenever ONE was used in his program it would be understood by the user and the machine that accumulator one is intended to be used in the calculation.

Several useful commands that this language incorporates is INCHWL which instructs the machine to get from the terminal a character typed by the user at the right time; ADDI (add immediate) adds the octal number designated at the end of the line (there is also SUBI...subtract immediate); IMULI and IMUL which multiplies the first operand by the number of the second operand and by the contents of the second operand respectively (with matching divide instructions, too).
This language, like other assembler languages incorporates the basic skip, jump and move instructions, too. For future reference of programs, the ; is the character that is typed before the line of comments used in a programmer's documentation.

When one becomes familiar with the instructions, and begins to write his own programs he may encounter the difficulty that the sixteen accumulators available to him are not enough space to allow him to do his intended processing. This is when one can declare memory space needed by # preceding memory words declared by the user for his use.

Word format is imperative to the user's knowledge also before he can begin to do any serious programming. The instruction code occupies bits 0 through 8, bits 9 through 12 contain the accumulator number, bits 13-17 are considered when we start to look at addressing, and bits 18 through 35 contain the address of the second operand in any instruction of the form:

\[
\text{instr} \quad \text{operand 1, operand 2}
\]

Effective addressing uses the bits 14-17 to designate an
index register. This effective address calculation is carried out for every instruction whose format specifies a memory reference. This effective address calculation is just the same for an instruction in immediate mode. This calculation of the effective address is done by the central processor. It is accomplished by retrieving the right half of the word, if bits 14-17 are zero there is no indexing by the index register, otherwise add to the contents of the accumulator given in bits 14-17 the right half of the word that was retrieved.

When the user wishes to use indirect addressing in his program, the symbol @ before a symbol (@NUM) to indicate indirect addressing will be used at this location. When bit 13 is set to 0, this shows completion of the effective address calculation, but as long as bit 13 is 1, the effective address calculation has not been completed.

Halfword instructions are also found in this language. They are HRR, HRL, HLR, and HLL where the first letter stands for Half, the second letter specifies which half of the source word is to be moved, and the third letter specifies which half of the destination word is to receive it.
The process of preparing a user's programs for execution is accomplished by the system program known as the Linking loader, known as LINK-10. The first thing that one does at the completion of writing a program is to compile it. If a subroutine or macro will be used in several different programs, one can write it as a separate file, and join it with each program at compilation time.

Many programming tasks involve the storage, manipulation, and output of text, which will be handled within the computer in the form of ASCII code. Each ASCII code requires seven bits; to reduce memory allocation one should pack in ASCII characters five to a word. This collection of adjacent bits within a single word is called a byte. To manipulate these packed characters one follows the byte pointer to reference the particular piece of information he wants to use.

The location chosen to house the byte pointer uses bits 13 through 35 to store the memory reference, bits 6 through 11 give the size of the byte in bits, and bits 0 through 5 of the byte pointer give the number of bits to the right of the byte in the word.
One aspect of MACRO-10 that I found to be very interesting
and useful was the fact that the whole FORTRAN library of sub-
routines and functions is available for its use. Conversely,
a FORTRAN program may reference a MACRO-10 subroutine. Input
and output using MACRO-10 can be utilized on a disk file, too.
All of these features show that MACRO-10 is a very versatile
language in itself.

Though most computers have a great deal in common, and
much of what I have said about MACRO-10 can apply equally well
in other places, with only a few minor changes, but this lang-
ue is too rich to be overlooked in what it can do.

MACRO-10 (DECE-10) gives a user a good knowledge of how
assembler language works, and in conjunction gives him a better
awareness of how the computer operates itself. This higher
level language should not be ignored. Programming with it is
very enjoyable and relatively easy, as I conclude this paper
with a demonstration of two programs that I conjured up using
MACRO-10.
REFERENCE

INTRODUCTION TO DECsystem ASSEMBLER LANGUAGE PROGRAMMING

TITLE INCHR

FUNCTION SUBPROGRAM TO IDENTIFY A CHARACTER
K=INCHR(CHR)

1. space
2. space on tab
3. letter, digit, n, ?

ENTRY INCHR
G=16
CHAR=1

INCHR:

MOVE 1
LIT CHAR,(POINT 7:R(8),6)
CAIE CHAR,*
Jrst TAB
CAIE CHAR,*
Jrst CHAR,*
CAIE SYM
Jrst CHAR,*
CAIE SYM
CAIE CHAR,*
CAIE CHAR,*
Jra G-1(S)
CAIE CHAR,*
Jra CHAR,*
Jra G+1(S)
SYN: MOVEI G-1(S)
TAP: MOVEI 2
Jra G+1(S)
END
CALL EXIT (BY-LONG ERROR) WHERE 1, NO ECHO, ELSE, ECHO.

MOVE 3,0

MOVE (3+1)

MOVE (3+1)

MOVE 1+2222

MOVE 3+1

MOVE 3+1

MOVE 3+1

MOVE 3+1

AGAIN:

MOVE 7+0

GAIN 7+0

JUMP Null

JUMP 3+1

JUMP AGAIN

NULL:

MOVE 16+(X40+4+600+0)

CLEAR 16+5(2)

CLEAR

THROB.

HALT

EXIT

EXIT

EXIT