1-1-1981

Assembler and interpreter for a mix-like assembly language

Shiang-Shyoung Shen
Atlanta University

Follow this and additional works at: http://digitalcommons.auctr.edu/dissertations
Part of the Computer Sciences Commons, and the Mathematics Commons

Recommended Citation

This Thesis is brought to you for free and open access by DigitalCommons@Robert W. Woodruff Library, Atlanta University Center. It has been accepted for inclusion in ETD Collection for AUC Robert W. Woodruff Library by an authorized administrator of DigitalCommons@Robert W. Woodruff Library, Atlanta University Center. For more information, please contact cwiseman@auctr.edu.
ASSEMBLER AND INTERPRETER
FOR A MIX-LIKE ASSEMBLY LANGUAGE

A THESIS

SUBMITTED TO THE FACULTY OF ATLANTA UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE

BY
SHIANG-SHYOUNG SHEN

COMPUTER SCIENCE
DEPARTMENT OF MATHEMATICS

ATLANTA, GEORGIA
JANUARY 1981
ACKNOWLEDGEMENTS

Many thanks and humble appreciations are extended to Dr. Nazir Warsi, my committee chairman and advisor, for his undying patience and assistance in the completion of this thesis.

I also would like to thank the other committee members, Dr. Johnny Houston and Dr. Benjamin Martin for their assistance.
# TABLE OF CONTENTS

**CHAPTER I  MIX-LIKE ASSEMBLY LANGUAGE**

1.1 HARDWARE ................................................. 1  
1.2 SOFTWARE ............................................... 2

**CHAPTER II  PROGRAMMING CONSIDERATIONS**

2.1 HARDWARE CONSTRAINTS .................................. 12 
2.2 LANGUAGE USED ........................................... 13 
2.3 PROGRAM STRUCTURE AND IMPLEMENTATION DEFINITION ... 14  
2.4 COMPILATION AND RUN-TIME PROBLEMS .................. 18

**CHAPTER III  COMPUTER PROGRAMMING ALGORITHMS**

3.1 TREE STRUCTURE SCHEMATIC ............................. 19 
3.2 A SIMPLE ALGORITHM ..................................... 20 
3.3 FLOWCHART ................................................ 21

**CHAPTER IV  ACTUAL COMPUTER PROGRAM**

4.1 NESTED PROGRAM STRUCTURE ............................ 28  
4.2 ACTUAL PROGRAM .......................................... 29

**CHAPTER V  EXAMPLE PROGRAMS**

74
CHAPTER I

MIX-LIKE ASSEMBLY LANGUAGE

This Mix-Like assembly language is based on Knuth's MIX assembly language. In this chapter, we present a brief description of Mix-Like assembly language and the hardware it needs.

1.1 HARDWARE

The hardware we use is a little unusual. The reason will be discussed in Chapter II.
REGISTERS

There are ten registers in our hardware.
The A-register (accumulator) called ACC in the program.
The X-register (extension) called EXT in the program.
The I-registers (index registers) I1, I2, I3, I4, I5, I6. We use REG1, REG2, ..., REG6 in the program.
The J-register (jump address) called REGJ in the program.
The Program Counter. We use PC in the program.

Each of these registers use one word of PDP-11/40. The ACC has many uses, especially for arithmetic and other operations on data. The EXT is mainly used to hold the remainder of a division. It also assists the ACC in various operations. The Index registers are used primarily for counting and referencing variable memory addresses. The REGJ always holds the address of the instruction to be executed following the "JUMP" instruction, and is primarily used in connection with subroutines. The PC always holds the address of the next instruction.

MEMORY

There are 2000 cells of memory storage, numbered from 0 to 1999. Each cell contains 4 words of PDP-11/40. Besides these, an overflow toggle (OV in the program) and a comparison indicator (CI in the program) each with one word of PDP-11/40 are also used.

1.2 SOFTWARE

PARTIAL FIELDS OF WORDS

Most of the instructions allow the programmer to use only a part of a word if he so chooses. In such a case a field specification is
given. The allowable fields are:

In Register \(-21345\) In Memory \(00-21345\)

Note that if we store data in memory we always use the fourth word.

The field specification is represented by \((L:R)\), where \(L\) is the left-hand part and \(R\) is the right-hand part of the field. Some examples of field specifications are given below.

- \((0:0)\) the sign only.
- \((0:2)\) the sign and the first two digits (left to right).
- \((0:5)\) the whole word. This is most common field specification.
- \((1:5)\) the whole word except for the sign.
- \((4:4)\) the fourth digit only.
- \((4:5)\) the two last digits.

The field \((L:R)\) is denoted in the machine by the single number \(8L+R\).

INSTRUCTION FORMAT

A computer word used for instructions has 4 bytes and the following form.

\[
\begin{array}{|c|c|c|c|}
\hline
1 & 2 & 3 & 4 \\
\hline
\pm A & I & F & C \\
\hline
\end{array}
\]

In this representation \(C\) is the operation code, and \(F\) holds a modification of the operation code. \(F\) is usually a field specification. For example, if \(C=8\) and \(F=11\), the operation is "load the A-register with the \((1:3)\) field". Sometimes \(F\) is used for other purposes. The left-most portion of the instruction, \(\pm A\), is the address. \(I\) is used to modify the address of an instruction. If \(I=0\), the address \(\pm A\) is used without any change, otherwise \(I\) contains a number between 1 and 6. The contents of index register \(\text{REGi}\) are added algebraically to
±A and the result is used as the address of the instruction. This indexing process takes place for every instruction. We will use the letter M to indicate the address after any specified indexing has occurred. In most instructions, M refers to a memory cell which ranges between 0 and 1999. In this case we write CONT(M) to denote the value stored in memory location M. In certain instructions, the address M has another significance, and it may be negative.

**NOTATION**

In symbolic form an instruction is represented as

\[ \text{OP} \quad \text{ADDRESS}, \text{I(F)} \]

Here \( \text{OP} \) is a symbolic name given to the operation code (the C-field) of the instruction, \( \text{ADDRESS} \) is the ±A portion, and \( \text{I,F} \) represent the I-field and F-field respectively.

If \( \text{I} \) is zero, then \( \text{I} \) is omitted. If \( \text{F} \) is a normal F-specification for this particular operator, then \( \text{F} \) need not be written. If \( \text{F} \) is standard, it will be mentioned explicitly when we discuss a particular operator.

In the following, some examples are discussed. The instruction to load a number into the accumulator is LDA with operation code 8.

<table>
<thead>
<tr>
<th>Assembly Language Code</th>
<th>Actual Numeric Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA 2000, 2(0:3)</td>
<td>2000 2 3 8</td>
</tr>
<tr>
<td>LDA 2000, 2(1:3)</td>
<td>2000 2 11 8</td>
</tr>
<tr>
<td>LDA 2000(1:3)</td>
<td>2000 0 11 8</td>
</tr>
<tr>
<td>LDA 2000</td>
<td>2000 * 5 8</td>
</tr>
<tr>
<td>LDA 2000, 4</td>
<td>2000 4 5 8</td>
</tr>
</tbody>
</table>

We will now define the action corresponding to each instruction.
LOADING OPERATORS

*LDA(load A). C=8; F=field.

The specified field of CONT(M) replace the previous contents of ACC. As in all operations where a partial field is used as an input, the sign is used if it's a part of the field. Otherwise the absolute value is understood. The field is shifted over to the right-hand part of the register as it is loaded. Suppose location 1000 contains the word \[0 \ 0 \ 0 \ -12345\], then the following are the results of loading various partial fields.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Contents of ACC afterwards</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA 1000</td>
<td>-12345</td>
</tr>
<tr>
<td>LDA 1000(1:5)</td>
<td>12345</td>
</tr>
<tr>
<td>LDA 1000(3:5)</td>
<td>345</td>
</tr>
<tr>
<td>LDA 1000(0:3)</td>
<td>-123</td>
</tr>
<tr>
<td>LDA 1000(4:4)</td>
<td>4</td>
</tr>
<tr>
<td>LDA 1000(1:1)</td>
<td>1</td>
</tr>
</tbody>
</table>

*LDX(load X). C=15 ; F=field.

The same as LDA except that EXT is loaded.

*LDi(load i). C=8+i; F=field.

The same as LDA except that REGi is loaded. REGi is constrained to hold a number not greater than 1999. Otherwise, the value is undefined. In the description of all instructions, "i" stands for an integer, 1≤i≤6. Thus LDi stands for 6 different instructions: LD1, LD2,...,LD6.

*LDAN(load A negative). C=16; F=field.

*LDXN(load X negative). C=23; F=field.
*LDIN (load i negative). \( C=16+i \); \( F=\text{field} \).

These eight instructions are the same as LDA, LDX, LDi except that the opposite signs are loaded.

STORING OPERATORS

*STA (store A). \( C=24 \); \( F=\text{field} \).

The contents of ACC replaces the field of \( \text{CONT}(M) \) specified by \( F \). The other parts of \( \text{CONT}(M) \) are unchanged.

On a store operation the field \( F \) has the opposite significance of the load operation. The number of digits in the field is taken from the right-hand side of the register and shifted left if necessary to be inserted in the proper field of \( \text{CONT}(M) \). The sign is not altered unless it is a part of the field. The contents of the register is not affected.

Example:

Suppose that location 1000 contains \( 0 \ 0 \ 0 \ -12345 \) and ACC contains \( 27890 \), then the following is the state of the memory segment related to store operation.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Contents of location 1000 afterwards</th>
</tr>
</thead>
<tbody>
<tr>
<td>STA 1000</td>
<td>0  0  0  27890</td>
</tr>
<tr>
<td>STA 1000(1:5)</td>
<td>0  0  0 -27890</td>
</tr>
<tr>
<td>STA 1000(5:5)</td>
<td>0  0  0 -12340</td>
</tr>
<tr>
<td>STA 1000(2:2)</td>
<td>0  0  0 -10345</td>
</tr>
<tr>
<td>STA 1000(2:3)</td>
<td>0  0  0 -19045</td>
</tr>
<tr>
<td>STA 1000(0:1)</td>
<td>0  0  0  2345</td>
</tr>
</tbody>
</table>

*STX (store X). \( C=31 \); \( F=\text{field} \).

The same as STA except EXT is stored rather than ACC.
*STi(store i). C=24+1; F=field.

The same as STA except REGi is stored rather than ACC.

*STJ(store J). C=32; F=field.

The same as STi except REGJ is stored, and its sign is always +.

*STZ(store zero). C=33; F=field.

The same as STA except zero is stored. In other words, the specified field of CONT(M) is cleared.

ARITHMETIC OPERATORS

For the add, subtract, multiply, and divide operations, a field specification is allowed. The standard field specification is (0:5). Other fields are treated as in LDA. We will use the letter V to indicate the specified field of CONT(M). Thus, V is the value which would have been loaded into ACC if the operation code were LDA.

*ADD. C=1; F=field.

V is added to ACC. If the magnitude of the result is too large for ACC, the overflow toggle is set on.

*SUB(subtract). C=2; F=field.

V is subtracted from ACC. Overflow may occur as in ADD.

*MUL(multiply). C=3; F=field.

V times (ACC) replaces ACC. Overflow may occur.

*DIV(divide). C=4; F=field.

The value of ACC is divided by V. The quotient is placed in ACC and the remainder in EXT.

ADDRESS TRANSFER OPERATORS

In the following operations, the "address" M is used as a signed number, not as the address of a cell in memory.
*ENTA(enter A). C=48; F=2.

The quantity M is loaded into ACC. The action is equivalent to LDA from a memory word containing the signed value of M.

Example: "ENTA 0,1" sets ACC to the current contents of REG1.

*ENTi(enter i). C=48+i; F=2.

*ENTX(enter X). C=55; F=2.

They are analogous to ENTA, loading the appropriate register.

*ENNA(enter negative A). C=48; F=3.

*ENNi(enter negative i). C=48+i; F=3.

*ENNX(enter negative X). C=55; F=3.

The same as ENTA, ENTi, and ENTX, except that the opposite sign is loaded.

*INCA(increase A). C=48; F=0.

The quantity M is added to ACC. Overflow is possible.

*INCi(increase i). C=48+i; F=0.

Add M to REGi. It is undefined if overflow occurs.

*INCX(increase X). C=55; F=0.

M is added to EXT. Overflow may occur.

*DECA(decrease A). C=48; F=1.

*DECi(decrease i). C=48+i; F=1.

*DECX(decrease X). C=55; F=1.

These eight instructions are the same as INCA, INCi, and INCX, respectively, except that M is subtracted from the register.

Note that the operation code C is the same for ENTA, ENNA, INCA, and DECA. The F-field is used to distinguish the various operations in each case.
COMPARISON OPERATORS

Each one of the comparison operators compares the value contained in a register with a value contained in the memory. The CI is then set to less than, equal to or greater than according to whether the value of the register is less than, equal to or greater than the value of the memory cell.

*CMPA(compare A). C=56; F=field.

The specified field of ACC is compared with the same field of CONT(M). If the field F does not include the sign position, the fields are both thought of as positive. Otherwise the sign is taken into account in the comparison.

*CMPX(compare X). C=63; F=field.

This is analogous to CMPA.

*CMPl(compare i). C=56+i; F=field.

This is analogous to CMPA.

JUMP OPERATORS

Ordinarily, instructions are executed in sequential order. In other words, instruction executed after the one in location P is the instruction found in location P+1. Several "JUMP" instructions allow this sequence to be interrupted. Whenever a jump of any kind takes place, the J-register is set to the address of the next instruction, i.e. the address of the instruction which would have been next if we hadn't jumped. A "store J" instruction then can be used by the programmer to set the address field of another command which will later be used to return to the original place in the program. The REGJ is changed whenever a jump actually occur in a program (except JSJ) and
it is never changed except when a jump occurs.

*JMP(jump). C=39; F=0.

This is an unconditional jump and the next instruction is taken from location M.

*JSJ(jump, save J). C=39; F=1.

The same as JMP except that the contents of REGJ are unchanged.

*JOV(jump on overflow). C=39; F=2.

If the overflow toggle is on, it is turned off and a JMP occurs. Otherwise, nothing happens.

*JNOV(jump on no overflow). C=39; F=3.

If the OV is off, a JMP occurs. Otherwise, it is turned off.

*JL,JE,JG,JGE,JNE,JLE(jump on less, equal, greater, greater-or-equal, unequal, less-or-equal). C=39; F=4,5,6,7,8,9, respectively.

Jump if the comparison indicator is set to the condition.

*JAN,JAZ,JAP,JANN,JANZ,JANP(jump A negative, zero, positive, nonnegative, nonzero, nonpositive). C=40; F=0,1,2,3,4,5, respectively.

If the contents of ACC satisfies the stated condition, a JMP occurs. Otherwise, nothing happens.

*JXN,JXZ,JXP,JXNN,JXNZ,JXNP(jump X negative, zero, positive, nonnegative, nonzero, nonpositive). C=47; F=0,1,2,3,4,5, respectively.

*JiN,JiZ,JiP,JiNN,JiNZ,JiNP(jump i negative, zero, positive, nonnegative, nonzero, nonpositive). C=40+i; F=0,1,2,3,4,5, respectively.

These are analogous to the corresponding operations for ACC.

MISCELLANEOUS OPERATORS

*MOVE. C=7; F=number.

The number of words specified by F is moved, starting from the
location \( M \) to the location specified by the contents of \( \text{REG}1 \). The transfer occurs one word at a time, and \( \text{REG}1 \) is incremented by the value of \( F \) at the end of the operation. If \( F=0 \), nothing happens.

Care must be taken when the group of locations involved overlap. For example, suppose that \( F=3 \) and \( M=1000 \). Then if \( (\text{REG}1)=999 \), we transfer \((1000)\) to \((999)\), \((1001)\) to \((1000)\), and \((1002)\) to \((1001)\). Nothing unusual occurred here. But if \( (\text{REG}1) \) were 1001 instead, we would move \((1000)\) to \((1001)\), then \((1001)\) to \((1002)\), and finally \((1002)\) to \((1003)\). So the same word \((1000)\) would have moved into three places.

*SLA,SRA,SLC,SRC(shift left A, shift right A, shift left A circularly, shift right A circularly). \( C=6; \ F=0,1,2,3 \), respectively.

These are the "shift" commands. Signs of ACC are not affected in any way. \( M \) specifies the number of digits to be shifted left or right. \( M \) must be non-negative.

*NOP(no operation). \( C=0; F \) is ignored.

No operation occurs, and this instruction is by-passed.

*HLT(halt). \( C=5; F=2 \).

The machine stops and the execution is terminated.

INPUT-OUTPUT OPERATORS

*IN(input). \( C=36; F=\text{number} \).

This instruction initiates the transfer of data specified by the number into sequential locations starting with \( M \).

*OUT(output). \( C=37; F=\text{number} \).

This instruction starts the output of information from memory locations starting at \( M \)(the number of output contents of memory locations are specified by the \( F \)-field).
CHAPTER II
PROGRAMMING CONSIDERATIONS

In this chapter, we will discuss how this program MAL.PAS is implemented. First of all, this program is written in PASCAL. In spite of the power of PASCAL, the hardware constraints of PDP-11/40 has forced various modifications.

2.1 HARDWARE CONSTRAINTS

In Knuth's MIX computer, the basic unit byte consists of six bits. A byte is capable of holding 64 distinct values. That is, we know that any number between 0 and 63, inclusive, can be contained in one byte. A word is 5 bytes plus a sign. Adjacent 2 to 5 bytes can express the number 0 through 1073741823. This is very powerful, especially when the programmer tries to use the Field Specification to manipulate the data within one word.

In MAL.PAS we use one-dimensional array of 4 words to simulate a MIX word, the first word holding the address part, the second word holding the index part, the 3rd word holding the field-specification part and the 4th word holding the operation code of the instruction. For storing data, we only use the 4th word and the other words are filled with zeros. All the registers use one word.
In MIX, the field-specification is manipulated with bytes. In MAL, the field-specification is manipulated with digits. The first component can hold a number ranging from 0 to 3. One word can hold a five-digit number, each digit ranging from 0 to 9. Hence, the field-specification feature can be implemented by a word which logically consists of five components.

2.2 LANGUAGE USED

The programming language used here is PASCAL, a high-level language which is devised by Niklaus Wirth. Some of the special features exploited in this thesis are discussed below.

WELL-STRUCTURED PROGRAMMING LANGUAGE

The size of a large program makes it complicated and thus difficult to understand. By using the program modularity, and top-down programming with PASCAL we can achieve the structured programming goals. PASCAL is designed to be a general purpose language. It can handle character data as well as numeric data in very clear and elegant way.

RECORD AND CASE FEATURES

Record is a collection of several fields. It can combine various data types. Case structure enables us to branch to one of several alternative statements. The case statement also incorporates the inclusion of ELSE clause. These two features are very important tool for implementing MAL.PAS.

Suppose in MAL.PAS we created a table for all the instructions with different types of data needed. The record structure make this very easy to handle.
2.3 PROGRAM STRUCTURE AND IMPLEMENTATION

As we know most of assemblers take two passes through the input program to produce the object. The first pass is used to produce a table of all symbolic addresses used and their address value. The second is used to substitute these values into the original symbolic form to get the binary form.

In this implementation, we do not quite follow this two-pass method. Our method can be viewed as the combination of both passes.

The assembly language has four well-defined components.
1. The location name which may be blank.
2. The mnemonic instruction or pseudo instruction.
3. The address field.
4. The comments field.

Two-pass assembler scans the location field first and produces a table to associate a memory address with each name in the table. Later it uses a chaining method to link the operation code and the address.

In our program, we scan one line at a time. The location name is put into a symbol table. Then the scanned instruction or the pseudo instruction is referenced to determine the value of the location name. The address field follows the instruction or pseudo. We compute
the address field and assign the machine codes to the correct memory cell. If a future reference location appears in the address field we reserve a word to store a key information for later use. Its value is assigned later when it is available. This is the first part of our program which we call TRANSLATOR. The second part of our program which we call EXECUTION is interpreter part.

FORMATTED ASSEMBLY LANGUAGE

The input data of MIX-LIKE ASSEMBLY LANGUAGE has the following format:

<table>
<thead>
<tr>
<th>Columns 1 - 6.</th>
<th>Location field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Columns 7</td>
<td>Blank</td>
</tr>
<tr>
<td>Columns 8 - 11</td>
<td>OP field</td>
</tr>
<tr>
<td>Columns 12</td>
<td>Blank</td>
</tr>
<tr>
<td>Columns 13 - 72</td>
<td>Address field and optional remarks</td>
</tr>
</tbody>
</table>

If column 1 contains an asterisk, the entire card(line) is treated as a comment. The optional remarks field is followed by address field and must have at least one blank between them.

The following few rules define the language.

1. A symbol is a string beginning with a letter followed by zero to five letters and/or digits.

2. A number x has the range -32600 ≤ x ≤ 32600.

3. Each appearance of a symbol is either a "defined symbol" or a "future reference". A defined symbol is a symbol which has appeared in the LOC field of a preceding line of the program. A future reference is a symbol which has not yet been defined this way.

4. An atomic expression is either
a) a number, or
b) a defined symbol, or
c) an asterisk.

5. An expression is either
   a) an atomic expression, or
   b) a plus or minus sign followed by an atomic expression, or
c) an expression followed by a binary operation followed by an atomic expression. Only three binary operations, +, -, \:, have been implemented to this point.

6. An A-part is either
   a) vacuous, denoting the value zero, or
   b) an expression, or
c) a future reference.

7. An index-part is either
   a) vacuous, denoting the value zero, or
   b) a comma followed by a digit from 1 to 7.

8. A F-part is either
   a) vacuous, denoting the standard F-setting, or
   b) a left parenthesis followed by an expression followed by a right parenthesis.

9. A W-part is an expression.

10. The assembly process makes use of the value denoted by * called the location counter and is initially zero. The value of * should always be a nonnegative number. When the location field of a line is not blank, it must contain a symbol which has not been previously defined. An equivalent of the symbol is then defined to be
the current value of *.

11. After processing the LOC field as described in 10, the assembly process depends on the value of the OP field. There are six possibilities for OP.

a) OP is symbolic MIX-LIKE operator. In this case, the ADDRESS should be an A-part, followed by an index part and then followed by a F-part.

b) OP is "EQU". The ADDRESS should be a W-part. The equivalent of the symbol is set equal to the value specified by ADDRESS. This rule takes over rule 10. The value of * is unchanged.

c) OP is "ORIG". The ADDRESS should be a W-part. The location counter * is set to this value.

d) OP is "ALF". The effect is to assemble the word of character codes in columns 13 - 16 of the card or line.

e) OP is "CON". The ADDRESS should be a W-part. The effect is to assemble a word having this value into the location specified by * and to advance * by one.

f) OP is "END". The ADDRESS should be a W-part, which specifies the location of the instruction at which the program ends.

12. Every symbol has one and only one equivalent value.

The most significant consequence of the above rules is the restriction on future reference. A symbol which hasn't yet been defined in the LOC field of a previous card(line) may not be used except as the A-part of an instruction. In particular, it may not be used in connection with arithmetic operation or in the ADDRESS field of EQU, ORIG, or CON.
2.4 COMPILATION AND RUN-TIME PROBLEMS

ERROR CHECKING SWITCHES ($A$, $T$)

The $A$ switch controls the generation of code to check array references and ensure that the index is within the bounds of the array. Bounds checking is initially enabled. The $A$- switch will disable checking. If enabled, each bounds check requires 8 words.

The $T$ switch controls stack overflow checking, and is initially enabled. Stack overflow is possible upon entry to any procedure or function block. This switch can be disabled with $T-$, resulting in savings of memory (2 words per procedure).

Because of the large memory requirement of the program, we use both disable switches /*$A-$, $T-$*/ to have some memory saving.

MEMORY ALLOCATION SWITCH (/B)

The /B switch selects the big compiler, which runs in 28K words (instead of 20K for the small compiler). The /B switch also uses the MACRO assembler, rather than the MAC assembler linker which is limited in terms of symbol table space.

Although these switches have been used, due to the memory limitations the compiler still can't compile the program. Therefore, the memory size implementation has been restricted to only 2000 words.

CASE STATEMENT

Consider the following PASCAL program segment.

Declaration: VAR O: ARRAY[1..4] OF CHAR; CH: CHAR;

a) CASE CH OF '(':...... b) CASE O OF 'EQU ':'......
   '-'......
   'ORIG':.......

The PASCAL handles the type (a) but not (b).
3.1 TREE STRUCTURE SCHEMATIC

In the beginning, we must realize that this program is a complex problem. Therefore, for this kind of problem we must first express the algorithm in some preliminary form and then in the form of flowcharts. However, before we describe the algorithm and draw the flowcharts, we construct a "TREE STRUCTURE SCHEME" for it. A tree structure is unlike the algorithm description or flowchart specification. Between sub-structures, it has no flowlines indicating directions. Its only purpose is to depict the functions to be performed within the various sub-structures.

The tree structure scheme can be stated as follow.
3.2 A SIMPLE ALGORITHM

After the functional chart is completed, we write a simple algorithm connecting all sub-structures together. A simple algorithm for this can be stated as follow.

STEP1: INPUT. Read one line at a time from an external file, written in MIX-LIKE ASSEMBLY LANGUAGE, and load it into a temporary buffer(array LINE). During the input(reading one character at a time within a line), print out the character until "END OF LINE" is detected.

STEP2: Scan through the current input line. (1)Scan the LOC-field. If there exist a symbol then assign the symbol to the symbol table. (2)Scan the OP-field (3)corresponding to the OPERATOR or PSEUDO OPERATOR scanned, invoke different procedures to scan the ADDRESS-field. During these three scan processes, we load all the codes into the memory. If there exist syntax errors then the ERROR routine is invoked to assign the error code and print the error messages immediately following the current printed line.

STEP3: END OF FILE? This is done by detecting the pseudo operator "END". If "END" is not detected, go to STEP1; otherwise, go to next STEP.

STEP4: ERROR = 0? If not, then go to STEP3. If yes, then go to next STEP.

STEP5: Based on the OP-codes, Field-codes, Index-codes, and Address-codes invoke the corresponding PROCEDUREs to process the information using the memory and registers.
PROCEDURE TRANSLATOR

Initialize instruction table, including its name, code, and standard field-specification.

Input an external file. The compiler will search users existing files until the specified file is found. If not, the variable "SIZE" will be set to the value "-1".

Initialization: COUNTER is program counter. LININDEX is label table index. LINENUMBER will assign line number to input file. When CHECK =1, it indicates the executable statements location in memory. ERR is error counter. I is character counter. CC is line length counter. CH is the current character read.

Invoke PROCEDURE SCANLINE (see next page).

Advance LINENUMBER by one.

END OF FILE? If not, scan next line.

ERR=0? If not, end of the program. If yes, pass the control to PROCEDURE EXECUTION.
PROCEDURE SCANLINE

(g: GETCHAR, an inner procedure.)
Ignore the leading blank.

I and CC are set to zero in the beginning. When starting to scan one line, this condition will invoke PROCEDURE GETCHAR to read one character of current input line at a time, print it immediately and store the characters read into a temporary buffer.

END OF LINE? If not, scan next character.

After EOLN, advance CC by one to prevent the condition I=CC from occurring again.

PROCEDURE GETCHAR's second purpose is to get one character at a time from the temporary line buffer.

If * is the first character in the line buffer, then ignore the whole line of comments and go to next line.

If LINE[1] through [7] are all blanks, then we don't have label in location field. Set the flag CB=1. Otherwise, set CB=0.
PROCEDURE SCANLABEL

SCANLABEL is an inner procedure of SCANLINE

If label does not begin at the first column and with a letter or the length of label is greater than six, then the ERROR routine is invoked. It assigns the error code, advances the error counter, and prints out the error messages after this error line.

Store the label into a temporary storage L.

Search the label table. If the label is not in the table then set LPTR = 0, assign label to the label table, and advance LINDEX by one. If the label is found, then set LPTR = J (i.e. J is the position in the label table) and assign this label value to label table.
The label table contains three fields: LABL.NAM, LABL.VALUE, LABL.ID. When we scan location field, if there exist a label then we assign that label to LABL.NAM. Also, after we have scanned the whole line, we can determine the label value which will be assigned to LABL.VALUE. If a label is found in address-field, first we search the table. If we find the label, LABL.VALUE can be determined; else, this label is a future reference. We assign it to LABL.NAM, leave the LABL.VALUE blank, and assign the value of current counter to LABL.ID.

SKIPSACES is an inner procedure. It will skips all the blanks including tab, return, line-feed, ....... etc.

GETCHAR is an inner procedure (see P23). If OP does not start at LINE[8] or the length of OP is greater than four, then the ERROR routine is invoked; else, PROCEDURE SCANOP picks up the OP and assign it to a temporary storage 0.

If ADDRESS does not start at LINE[13], then ERROR routine is invoked again.
PROCEDURE SCANADDRESS depends upon the operator or pseudo to invoke procedures to compute the address.

During the scan process, some of the codes are already assigned to the suitable memory locations, and after address is scanned all the codes are assigned to the memory except the future reference label. These are updated when values are found in future location field.

The pseudo operators "EQU", "ORIG", and "CON" use PROCEDURE GET1 to compute the addresses.

When the "OPTION REMARK FIELD" is presented, the remarks are simply ignored. For this, we insist that there be at least one blank between address field and the remark field.
PROCEDURE EXECUTION

Four parts of the instruction are renamed. It is easy to understand this way.

If the index-field is presented, compute the effective address for each instruction.

The FIELD-SPECIFICATION has the form (L:R). It is stored in the memory as a single number 8L+R.

Call PROCEDURE OPERATION to check the opcodes and invokes various procedures to perform various functions.

Is it a JUMP operator? Is the condition satisfied? If so, set PC to ADDR value, else advance PC by one.

END OF EXECUTABLE STATEMENTS?
If not, executes next statement; else, terminates the EXECUTION.
CHAPTER IV

ACTUAL COMPUTER PROGRAM

4.1 NESTED PROGRAM STRUCTURE

4.2 ACTUAL PROGRAM
DOCUMENTATION

THIS PROGRAM, MAL.PAS, SIMULATES BOTH AN ASSEMBLER AND AN INTERPRETER FOR A MIX-LIKE ASSEMBLY LANGUAGE.

Both the hardware (simulated computer) and the software (mix-like assembly language) are described in Chapter One of this project.

The first portion of this program, e.g., assembler, is implemented as a procedure "translator," and the second part, e.g., interpreter, is implemented as a procedure "execution." Both procedures are the outermost procedures of a nested procedures group.

There are sixteen procedures nested in the translator. They are scanline, error, skipspace, getchar, coplabel, labeltable, scanlabel, compnum, compfield, cases, scanop, assign, geta1, geta2, getfield, and scanaddress.

"Translator" takes an assembly program as an input, generates a series of decimal codes and through the main program passes all the codes to procedure "execution." Procedure "execution" plays the "interpreter" role in this program. It uses procedure operation to interpret the opcodes and invokes 35 different procedures to achieve the objectiveness of the input assembly program.

Also, MAL uses a small procedure called printtime to print the current time before the program prints out the results.

The simple algorithm for this program can be stated as follows:

**STEP1:** Read one line at a time from an external file (written in mix-like assembly language), and load it into a temporary buffer (array line [1..72]).

**STEP2:** During the input (reading one character at a time within a line), also print out the characters until end of line is detected.

**STEP3:** Scan through the current line, assign the symbol to symbol table; also assign the value to that symbol. After generating all the information, load it into memory. The information includes address codes, index codes, field codes, and opcodes. If there exists syntax errors, then print the error code immediately following the current line printed.

**STEP4:** End of file? (by detecting the pseudo operator "end") If not, go to step1. If yes, goto next step.

**STEP5:** If error > 0 then goto step 10, else next step.

**STEP6:** Based on the opcodes, field codes, index codes and address codes invoke all the correspondi-
NG PROCEDURES TO CALCULATE ALL KINDS OF INFORMATION.
STEP7: END OF EXECUTABLE INSTRUCTION? (BY DETECTING
THE OPERATOR "HLT") IF YES, GOTO NEXT STEP;
ELSE GOTO STEP6.
STEP8: PRINT THE CURRENT TIME.
STEP9: PRINT THE RESULTS.
STEP10: STOP.
The detail description of each procedure is written in
comment box before the procedure heading.

PROGRAM MAL;
LABEL 100;
CONST AL=10;
MAX1=32600;
MAX2=1999;
TYPE ALFA=PACKED ARRAY[1..AL] OF CHARACTER;
CELL=ARRAY[1..4] OF INTEGER;
VAR MEM:ARRAY[0..MAX2] OF CELL;
F:TEXT;
FNAME:ALFA;
PC;
REG1,
REG2,
REG3,
REG4,
REG5,
REG6,
REGJ;
ACC;
EXT;
OV;
CI;
J;
INTEGER;

PROCEDURE "TRANSLATOR" DOES THE JOB OF AN ASSEMBLER FOR
MAL.PAS. IT TAKES AN EXTERNAL FILE WRITTEN IN MIX-LIKE
ASSEMBLY LANGUAGE AS AN INPUT, THEN INVOKE THE PROCEDURE
"SCANLINE" TO SCAN ONE LINE OF THE INPUT FILE AT A TIME,
AND GENERATE THE DECIMAL CODES.
PROCEDURE TRANSLATOR;
CONST NOINST=139; (*NUMBER OF INSTRUCTION*)
LENGTH=50; (*LENGTH OF LABEL TABLE*)
LINELEN=72; (*LINE LENGTH*)
TYPE INSTRUCTION=RECORD
  NAM:PACKED ARRAY[1..4] OF CHAR;
  CODE:INTEGER;
  FIELD:INTEGER
END;
TABLE=RECORD
  NAM:PACKED ARRAY[1..6] OF CHAR;
  VALUE:INTEGER;
  ID:INTEGER
END;
VAR INST:ARRAY[1..NOINST] OF INSTRUCTION;
LABL:ARRAY[1..LENGTH] OF TABLE;
LINE:ARRAY[1..LINELEN] OF CHAR;
CH:CHAR; (*LAST CHARACTER READ*)
I, (*LINE LENGTH*)
INDEX, (*LABEL TABLE INDEX*)
CC, (*CHARACTER COUNTER*)
ERR, (*ERROR COUNTER*)
LINENUMBER, (*LINE NUMBER*)
SIZE, (*INPUT FILE SIZE*)
CHECK, (*PROGRAM COUNTER FLAG*)
COUNTER; (*MEMORY LOCATION INDEX*)
INTEGER;

/* SCAN ONE LINE OF INPUT FILE AT A TIME, ASSIGN THE SYMBOL TO SYMBOL TABLE AND ASSIGN THE OPCODE, INDEX CODE, FIELD CODE, AND ADDRESS CODE TO SUITABLE MEMORY CELLS. AFTER THAT, WE CAN CALCULATE THE SYMBOL VALUE, ASSIGN THEM TO SYMBOL TABLE AND MEMORY CELLS. */
PROCEDURE SCANLINE;
LABEL 99;
CONST E='EQU '; G='ORIG';
C='CON '; A='ALF '; N='END '; (*FIVE PSEUDO OPERATORS*)
VAR L:PACKED ARRAY[1..6] OF CHAR;
O:PACKED ARRAY[1..4] OF CHAR;
CB, (*LABEL FLAG*)
FP, (*1ST POINTER TO SYMBOL*)
LP, (*2ND POINTER TO SYMBOL*)
LEN, (*SYMBOL LENGTH*)
LPTR, (*LABEL TABLE POINTER*)
LPTR2, (*LABEL TABLE POINTER*)
NUM, (*NUMBER CONVERT FROM CHARACTERS*)
K, (*INSTRUCTION INDEX*)
INTEGER;
/* DETECT ERRORS AND ASSIGN ERROR CODE */
PROCEDURE ERROR(N:INTEGER);
BEGIN
  WRITELN('****','I-1','~',N:2);
  ERR:=ERR+1
END; (*ERROR*)

/* SKIP BLANK WITHIN LINE LENGTH */
PROCEDURE SKIPSPACES;
BEGIN
  WHILE (I<=LINELENG) AND (ORD(LINE[I])<=40) DO
    BEGIN (*40 IS OCTAL REPRESENTATION FOR BLANK*)
      I:=I+1;
      CH:=LINE[I]
    END;
    I:=I-1
  END; (*SKIPSPACES*)

/* GET ONE CHARACTER AT A TIME PER INPUT FILE LINE, AND
PRINT THE INPUT FILE LINE AFTER LINE. */
PROCEDURE GETCHAR;
BEGIN
  IF I=CC THEN
  BEGIN
    IF EOF(F) THEN
    BEGIN
      WRITE('**PROGRAM INCOMPLETE**')
      GOTO 100
    END;
    CC:=0;
    I:=0;
    WRITE(LINE NUMBER:4,' ')$
    WHILE NOT EOLN(F) DO
    BEGIN (*COPY THE INPUT FILE TO KEYBOARD USED*)
      CC:=CC+1;
      READ(F,CH);$
      WRITE(CH)
      LINE[CC]:=CH
    END;
    WRITELN;
    CC:=CC+1;
    READ(F,LINES[CC])
  END;
  IF I<LINELENG THEN
  BEGIN
    I:=I+1;
    CH:=LINE[I]
  END
END; (*GETCHAR*)
/* GET LABEL FROM LABEL FIELD OR ADDRESS FIELD, INVOKE INNER
PROCEDURE LABELTABLE TO ASSIGN THE LABEL TO LABEL TABLE OR
SEARCH THE LABEL FROM LABEL TABLE. */

PROCEDURE COMPLABEL;
PROCEDURE LABELTABLE;
LABEL 20+30;
VAR J: INTEGER;
BEGIN
FOR J:=FP TO LP DO
  LCJ-FP+1:=LINE[CJ];
IF LEN<6 THEN
  FOR J:=LEN+1 TO 6 DO
    FOR J:=1 TO LENGTH DO
      BEGIN
        IF L=LABLCJ+NAM THEN
          BEGIN
            LPTR:=J;
            GOTO 20
          END
        ELSE LPTR:=0
      END;
  20:
  IF LPTR<>0 THEN
    FOR J:=LPTR TO LENGTH DO
      BEGIN
        IF L=LABLCJ+NAM THEN
          BEGIN
            LPTR2:=J;
            GOTO 30
          END
        ELSE LPTR2:=0
      END;
  30:
END;

BEGIN
WHILE CH IN ['A'..'Z','O'..'9'] DO
  GETCHAR;
  LP:=I-1;
  LEN:=LP-FP+1;
  IF LEN>6 THEN (*MAXIMUM OF LABEL LENGTH IS SIX*)
    ERROR(3)
  ELSE
    LABELTABLE
END; (*COMPLABEL*)

/* SCAN THE LABEL FIELD AND INVOKE PROCEDURE COMPLABEL
TO DO THE REST JOBS */
PROCEDURE SCANLABEL;
BEGIN
  IF FP=1 THEN (*START AT 1ST COLUMN*)
BEGIN
  IF CH IN ['A'..'Z'] THEN
    COMPLABEL
  ELSE
    ERROR(1)
  END
ELSE
  IF (FP>1) AND (FP<7) THEN
    ERROR(2)
  END;
END; /*SCANLABEL*/

/* CONVERT THE DIGIT CHARACTER SCANNED TO ITS VALUE */
PROCEDURE COMPNUM;
BEGIN
  NUM:=0;
  REPEAT
    NUM:=10*NUM+(ORD(CH)-ORD('0'));
    GETCHAR
  UNTIL NOT (CH IN ['0'..'9'])
END;

/* COMPUTE THE FIELD SPECIFICATION IN ADDRESS PART */
PROCEDURE COMPFIELD;
BEGIN
  IF CH=':' THEN
    BEGIN
      GETCHAR;
      NUM:=NUM*8+(ORD(CH)-ORD('0'))
    END
  END;

PROCEDURE SCANOP;
PROCEDURE ASSIGN;
VAR J:INTEGER;
BEGIN
  FOR J:=FP TO LP DO
    OCJ-FP+1]:=LINE[J];
  IF LEN<4 THEN
    FOR J:=LEN+1 TO 4 DO
      OCJ]:=''
  END;
BEGIN
  IF CH IN ['A'..'Z'] THEN
    BEGIN
      WHILE CH IN ['A'..'Z','1'..'6'] DO

GETCHAR;
LP:=I-1;
LEN:=LP-FP+1;
IF (LEN>4) OR (LP>11) THEN
  ERROR(5)
ELSE
  ASSIGN
END
ELSE
  ERROR(6)
END
ELSE
  ERROR(4)
END;

/* ACCORDING TO THE PSEUDO OPERATOR SCANNED, GET THE
ADDRESS FROM ADDRESS PART AND ASSIGN THE VALUE TO
MEMORY LOCATION. */
PROCEDURE GETA1;
BEGIN
  SKIPSPACES;
  GETCHAR;
  FP:=I;
  IF FP=13 THEN (*START AT 13TH COLUMN*)
  BEGIN
    IF CH IN ['0'..'9'] THEN
    BEGIN
      COMPNUM;
      IF CH<>'' THEN
        COMPFIELD
    END;
    IF CH='-' THEN
    BEGIN
      GETCHAR;
      COMPNUM;
      NUM:=-NUM
    END;
    IF CH IN ['A'..'Z'] THEN
    BEGIN
      FP:=I;
      COMPLABEL;
      CASE CH OF
        ' ': NUM:=LABL[LPTR].VALUE;
        '+' BEGIN
          GETCHAR;
          COMPNUM;
          NUM:=LABL[LPTR].VALUE+NUM
        END;
        '-' BEGIN
          GETCHAR;
          COMPNUM;
      END;
    ELSE
      ERROR(4)
NUM: = LABL[LPTR].VALUE- NUM 
END

END
END;
IF CH = '*' THEN
BEGIN
GETCHAR;
CASE CH OF
' ': NUM: = COUNTER;
'+': BEGIN
GETCHAR;
COMPNUM;
NUM: = COUNTER + NUM
END;
'-': BEGIN
GETCHAR;
COMPNUM;
NUM: = COUNTER - NUM
END;
ELSE
ERROR(8)
END
CASE CH OF
END
/* ACCORDING TO THE OPERATOR SCANNED, IF ADDRESS-PART IS A EXPRESSION THEN SCAN REST OF ADDRESSES AND COMPUTE THEM & I-PART, ASSIGN BOTH VALUES TO MEMORY CELL. */
PROCEDURE CASES;
BEGIN
CASE CH OF
' ': MEM[COUNTER, 2]: = 0;
'+': BEGIN
GETCHAR;
IF CH IN [ '1', '7' ] THEN
BEGIN
COMPNUM;
MEM[COUNTER, 2]: = NUM
END
ELSE
ERROR(4)
END;
END;
'+': BEGIN
GETCHAR;
IF CH IN [ '0', '9' ] THEN
BEGIN
COMPNUM;
MEM[COUNTER, 1]: = MEM[COUNTER, 1] + NUM
END;
END;
IF CH IN ['A'..'Z'] THEN
BEGIN
    FP:=I;
    COMPLABEL;
    MEM[COUNTER,1]:=MEM[COUNTER,1]+LABL[LPTR],VALUE
END;
CASE CH OF
    '/': MEM[COUNTER,2]:=0;
    '/': BEGIN
    GETCHAR;
    IF CH IN ['0'..'9'] THEN
        BEGIN
            COMPNUM;
            MEM[COUNTER,2]:=NUM
        END;
    ELSE
        ERROR(4)
    END
END
END;
'-' BEGIN
    GETCHAR;
    IF CH IN ['0'..'9'] THEN
        BEGIN
            COMPNUM;
            MEM[COUNTER,1]:=MEM[COUNTER,1]-NUM
        END;
    IF CH IN ['A'..'Z'] THEN
        BEGIN
            FP:=I;
            COMPLABEL;
            MEM[COUNTER,1]:=MEM[COUNTER,1]-LABL[LPTR],VALUE
        END;
    CASE CH OF
        '/': MEM[COUNTER,2]:=0;
        '/': BEGIN
            GETCHAR;
            IF CH IN ['0'..'7'] THEN
                BEGIN
                    COMPNUM;
                    MEM[COUNTER,2]:=NUM
                END;
            ELSE
                ERROR(4)
            END
        END
    END
END
END; (*CASES*)
/* ACCORDING TO THE OPERATOR SCANNED, GET THE ADDRESS
  OF THE ADDRESS PART. */
PROCEDURE GETA2;
BEGIN
  SKIPSACES;
  GETCHAR;
  IF LINE[13]==' ' THEN
    BEGIN
      MEMCOUNTER,1]:=0;
      MEMCOUNTER,2]:=0
    END;
  CASE CH OF
    'O'..'2'..'9';
    BEGIN
      COMPNUM;
      MEMCOUNTER,1]:=NUM;
      MEMCOUNTER,3]:=INST[K].FIELD;
      CASES
    END;
    '-'
    BEGIN
      GETCHAR;
      IF CH IN ['O'..'9'] THEN
        BEGIN
          COMPNUM;
          MEMCOUNTER,1]:=NUM;
          MEMCOUNTER,3]:=INST[K].FIELD;
          CASES
        END;
      IF CH IN ['A'..'Z'] THEN
        BEGIN
          FP]:=I;
          COMPLABEL;
          MEMCOUNTER,1]:=NUM;
          MEMCOUNTER,3]:=INST[K].FIELD;
          CASES
        END;
    END;
    'A'..'Z';
    BEGIN
      IF LPT<0 THEN
        BEGIN
          IF LABEL[LPT].ID=0 THEN
            BEGIN
              MEMCOUNTER,1]:=LABEL[LPT].VALUE;
              MEMCOUNTER,3]:=INST[K].FIELD;
            END
          ELSE
            BEGIN
            END
        END
    END
END.
LABL[INDEX-1].NAM:=L;
LABL[INDEX-1].ID:=COUNTER
END
ELSE
BEGIN
WITH LABL[INDEX] DO
BEGIN
NAM:=L;
ID:=COUNTER
END;
INDEX:=INDEX+1
END;
CASES
END;
'*':
BEGIN
MEMCOUNTERJ1:=COUNTER;
MEMCOUNTERJ3:=INSTKJ.FIELD;
GETCHAR;
CASES
END
END;  (*GETA2*)

/* COMPUTE THE F-PART OF THE ADDRESS-FIELD */
PROCEDURE GETFIELD;
BEGIN
IF CH='<' THEN
BEGIN
GETCHAR;
IF CH IN ['0'..'9'] THEN
BEGIN
COMPNUM;
COMPFIELD;
MEMCOUNTERJ3:=NUM
END;
IF CH IN ['A'..'Z'] THEN
BEGIN
FP:=I;
COMPLABEL;
MEMCOUNTERJ3:=LABL[LPTR].VALUE
END
END
END;  (*GETFIELD*)

/* ACCORDING TO THE OPERATORS OR PSEUDO OPERATORS SCANNED, 
DECIDE HOW TO COMPUTE THE ADDRESSES. */
PROCEDURE SCANADDRESS;
BEGIN
IF 0=E THEN
BEGIN
GETA1;
LABLINDEX-1].VALUE:=NUM
END;
IF 0=G THEN
BEGIN
GETA1;
IF CB=0 THEN
BEGIN
IF LABLINDEX-1].NAM<>"' THEN
LABLINDEX-1].VALUE:=NUM
END;
COUNTER:=NUM;
CHECK:=CHECK+1;
IF CHECK=1 THEN
PC:=COUNTER
END;
IF 0=C THEN
BEGIN
GETA1;
IF CB=0 THEN
BEGIN
IF LABLINDEX-1].NAM<>"' THEN
LABLINDEX-1].VALUE:=NUM
END;
MEMCOUNTER,4]=NUM;
COUNTER:=COUNTER+1
END;
IF 0=A THEN
BEGIN
REPEAT
GETCHAR
UNTIL I=13;
MEMCOUNTER,1]:=ORD(CH);;
GETCHAR;
MEMCOUNTER,2]:=ORD(CH);;
GETCHAR;
MEMCOUNTER,3]:=ORD(CH);;
GETCHAR;
MEMCOUNTER,4]:=ORD(CH);;
COUNTER:=COUNTER+1
END;
IF \(0=N\) THEN
BEGIN
\(*\) DO NOTHING *)
END;

FOR \(K:=1\) TO NOINST DO
IF \(O=\text{INST}[K].\text{NAM}\) THEN
BEGIN
IF \(CB=0\) THEN
BEGIN
IF \(\text{LABL}[\text{INDEX}-1].\text{NAM}<>'\) THEN
\text{LABL}[\text{INDEX}-1].\text{VALUE}:=\text{COUNTER}
END;
ELSE
BEGIN
IF \(\text{LABL}[\text{INDEX}].\text{NAM}<>'\) THEN
\text{LABL}[\text{INDEX}].\text{VALUE}:=\text{COUNTER}
END;
\text{MEM}[\text{COUNTER}+3]=\text{INST}[K].\text{FIELD};
\text{MEM}[\text{COUNTER}+4]=\text{INST}[K].\text{CODE}.
IF \((K=0)\) OR \((K<5)\) AND \((K<13)\) OR \((K>41)\) AND \((K<132)\)
THEN
BEGIN
\text{GETA2}
END;
IF \((K>0)\) AND \((K<6)\) OR \((K>12)\) AND \((K<42)\) OR \((K>131)\)
THEN
BEGIN
\text{GETA2};
\text{GETFIELD}
END;
\text{COUNTER}:=\text{COUNTER}+1
END;

BEGIN
\(*\text{SCANADDRESS*}\)
\(*\text{SCANLINE*}\)
BEGIN
\(*\text{SCANADDRESS*}\)
\(*\text{SCANLINE*}\)
\(*\text{ignore the leading blank*}\)
\text{GETCHAR};
\text{IF LINE[1]<<'*' THEN}
\(*\text{else ignore comment line*}\)
BEGIN
\text{FP:=I};
\text{CB:=0};
\text{IF FP=8 THEN}
\(*\text{HAVE LABEL?*}\)
\text{BEGIN}
\text{CB:=1};
\text{GOTO 99}
\text{END};
\text{SCANLABEL};
\(*\text{check label table*}\)
\text{IF LPTR=0 THEN}
\(*\text{not found*}\)
\text{BEGIN}
\text{LABL}[\text{INDEX}].\text{NAM}:=L;
\(*\text{assign to label table*}\)
\text{INDEX}:=\text{INDEX}+1
\text{END};
ELSE
BEGIN
(*FOUND*)
IF LABL[L PTR].ID<>0 THEN (*FUTURE REFERENCE*)
BEGIN
MEM[LABL[L PTR].ID,1]:=MEM[LABL[L PTR].ID,1]+COUNTER;
(*REASSIGN LABEL VALUE*)
LABL[L PTR].VALUE:=COUNTER;
LABL[L PTR].ID:=0
END;
IF LPTR2<>0 THEN (*2ND FUTURE REFERENCE*)
MEM[LABL[L PTR2].ID,1]:=MEM[LABL[L PTR2].ID,1]+COUNTER
(*REASSIGN LABEL VALUE*)
END;
SKIPSPACES;
GETCHAR;
FP:=I;
??;
SCANOP;
(*SCAN OPERATOR AND ASSIGN CODE TO MEM*)
SCANADDRESS
(*COMPUTE THE ADDRESS*)
END

BEGIN
(*TRANSLATOR*)
/*INSTRUCTIONS TABLE CREATION */
INST[1].NAM:=‘NOP’;
INST[2].NAM:=‘ADD’;
INST[3].NAM:=‘SUB’;
INST[4].NAM:=‘MUL’;
INST[5].NAM:=‘DIV’;
INST[6].NAM:=‘NUM’;
INST[7].NAM:=‘CHAR’;
INST[8].NAM:=‘HLT’;
INST[9].NAM:=‘SLA’;
INST[10].NAM:=‘SRA’;
INST[12].NAM:=‘SRC’;
INST[13].NAM:=‘MOVE’;
INST[14].NAM:=‘LDA’;
INST[15].NAM:=‘LD1’;
INST[16].NAM:=‘LD2’;
INST[17].NAM:=‘LD3’;
INST[18].NAM:=‘LD4’;
INST[19].NAM:=‘LD5’;
INST[20].NAM:=‘LD6’;
INST[21].NAM:=‘LDX’;
INST[22].NAM:=‘LDAN’;
INST[23].NAM:=‘LD1N’;
INST[24].NAM:=‘LD2N’;
INST[25].NAM:=‘LD3N’;
INST[1].CODE:=0;
INST[2].CODE:=1;
INST[3].CODE:=2;
INST[4].CODE:=3;
INST[5].CODE:=4;
INST[6].CODE:=5;
INST[7].CODE:=6;
INST[8].CODE:=5;
INST[9].CODE:=6;
INST[10].CODE:=6;
INST[12].CODE:=6;
INST[13].CODE:=7;
INST[14].CODE:=8;
INST[15].CODE:=9;
INST[16].CODE:=10;
INST[17].CODE:=11;
INST[18].CODE:=12;
INST[19].CODE:=13;
INST[20].CODE:=14;
INST[21].CODE:=15;
INST[22].CODE:=16;
INST[23].CODE:=17;
INST[24].CODE:=18;
INST[25].CODE:=19;
INST[1].FIELD:=0;
INST[2].FIELD:=5;
INST[3].FIELD:=5;
INST[4].FIELD:=5;
INST[5].FIELD:=5;
INST[6].FIELD:=5;
INST[7].FIELD:=1;
INST[8].FIELD:=2;
INST[9].FIELD:=0;
INST[10].FIELD:=1;
INST[11].FIELD:=2;
INST[12].FIELD:=3;
INST[13].FIELD:=0;
INST[14].FIELD:=5;
INST[15].FIELD:=5;
INST[16].FIELD:=5;
INST[17].FIELD:=5;
INST[18].FIELD:=5;
INST[19].FIELD:=5;
INST[20].FIELD:=5;
INST[21].FIELD:=5;
INST[22].FIELD:=5;
INST[23].FIELD:=5;
INST[24].FIELD:=5;
INST[25].FIELD:=5;
PROCEDURE "EXECUTION" DOING THE INTERPRESTER JOB FOR THIS PROGRAM. IT TAKES ALL THE CODES GENERATED BY TRANSLATOR TO PERFORM VARIOUS ACTIONS AND GET THE RESULTS.*

PROCEDURE EXECUTION;
VAR OPCODE, (*OPERATOR CODE*)
FIELD, (*FIELD-SPECIFICATION*)
INDEX, (*INDEX FIELD*)
ADDR, (*ADDRESS PART*)
EFA, (*EFFECTIVE ADDRESS*)
RFIELD, (*RIGHT-FIELD OF FIELD SPECIFICATION*)
LFIELD, (*LEFT-FIELD OF FIELD SPECIFICATION*)
C1,C2,C3,C4,C5:INTEGER;

PROCEDURE NOP;
BEGIN
PC:=PC+1
END; (*NOP*)

PROCEDURE ADD;
VAR C1,C2,C3,C4,C5:INTEGER;
BEGIN
C1:=MEM[EFA,4];
C2:=MEM[EFA,4] DIV 10;
C3:=MEM[EFA,4] DIV 100;
C4:=MEM[EFA,4] DIV 1000;
C5:=MEM[EFA,4] DIV 10000;
CASE LFIELD OF
  0: BEGIN
    CASE RFIELD OF
      1: ACC:=ACC+C5;
      2: ACC:=ACC+C4;
      3: ACC:=ACC+C3;
      4: ACC:=ACC+C2;
      5: ACC:=ACC+C1
    END
  END;
  1: BEGIN
    CASE RFIELD OF
      1: ACC:=ACC+ABS(C5);
      2: ACC:=ACC+ABS(C4);
      3: ACC:=ACC+ABS(C3);
      4: ACC:=ACC+ABS(C2);
      5: ACC:=ACC+ABS(C1)
    END
  END;
  2: BEGIN
    CASE RFIELD OF
      2: ACC:=ACC+ABS(C4-C5*10);
      3: ACC:=ACC+ABS(C3-C5*100);
      4: ACC:=ACC+ABS(C2-C5*1000)
    END
  END;
*/
/* NO OPERATION, ONLY ADVANCE PC BY ONE */
/* THE CONTENT OF EFFECTIVE MEMORY LOCATION'S SPECIFIED FIELD IS ADDED TO ACC. IF THE MAGNITUDE OF THE RESULT IS TOO LARGE FOR ACC, THE OVERFLOW TOGGLE IS SET ON. */
5: \text{ACC}:=\text{ACC}+\text{ABS}(\text{C1}-\text{C2} \times 10)
\text{END}
\text{END}\
3: \text{BEGIN}
\text{CASE RFIELD OF}
3: \text{ACC}:=\text{ACC}+\text{ABS}(\text{C3}-\text{C4} \times 10)
4: \text{ACC}:=\text{ACC}+\text{ABS}(\text{C2}-\text{C4} \times 100)
5: \text{ACC}:=\text{ACC}+\text{ABS}(\text{C1}-\text{C4} \times 1000)
\text{END}
\text{END}\
4: \text{BEGIN}
\text{CASE RFIELD OF}
4: \text{ACC}:=\text{ACC}+\text{ABS}(\text{C2}-\text{C3} \times 10)
5: \text{ACC}:=\text{ACC}+\text{ABS}(\text{C1}-\text{C3} \times 100)
\text{END}
\text{END}\
5: \text{ACC}:=\text{ACC}+\text{ABS}(\text{C1}-\text{C2} \times 10)
\text{END}\
\text{IF ACOMAX}1 \text{ THEN}
\text{OV}:=1$
\text{PC}:=\text{PC}+1$
\text{END} (\text{*ADD*})

\begin{verbatim}
/* THE CONTENT OF EFFECTIVE MEMORY LOCATION’S SPECIFIED
FIELD IS SUBTRACTED FROM ACC. OVERFLOW MAY OCCURED. */
PROCEDURE SUB;
VAR C1,C2,C3,C4,C5:INTEGER;
BEGIN
C1:=MEM[4A+43 DIV 10]
C2:=MEM[4A+43 DIV 100]
C3:=MEM[4A+43 DIV 1000]
C4:=MEM[4A+43 DIV 10000]
CASE LFIELD OF
0: BEGIN
\text{CASE RFIELD OF}
1: \text{ACC}:=\text{ACC}-\text{C5}\
2: \text{ACC}:=\text{ACC}-\text{C4}\
3: \text{ACC}:=\text{ACC}-\text{C3}\
4: \text{ACC}:=\text{ACC}-\text{C2}\
5: \text{ACC}:=\text{ACC}-\text{C1}\
\text{END}
\text{END}\
1: BEGIN
\text{CASE RFIELD OF}
1: \text{ACC}:=\text{ACC}-\text{ABS}(\text{C5})\
2: \text{ACC}:=\text{ACC}-\text{ABS}(\text{C4})\
3: \text{ACC}:=\text{ACC}-\text{ABS}(\text{C3})\
4: \text{ACC}:=\text{ACC}-\text{ABS}(\text{C2})\
5: \text{ACC}:=\text{ACC}-\text{ABS}(\text{C1})\
\text{END}
\end{verbatim}
BEGIN
CASE RFIELD OF
  2: ACC:=ACC-ABS(C4-C5*10);
  3: ACC:=ACC-ABS(C3-C5*100);
  4: ACC:=ACC-ABS(C2-C5*1000);
  5: ACC:=ACC-ABS(C1-C5*10000)
END
END;
3: BEGIN
CASE RFIELD OF
  3: ACC:=ACC-ABS(C3-C4*10);
  4: ACC:=ACC-ABS(C2-C4*100);
  5: ACC:=ACC-ABS(C1-C4*1000)
END
END;
4: BEGIN
CASE RFIELD OF
  4: ACC:=ACC-ABS(C2-C3*10);
  5: ACC:=ACC-ABS(C1-C3*100)
END
END;
5: ACC:=ACC-ABS(C1-C2*10)
END;
IF ACC>MAX1 THEN
  OV:=1;
  PC:=PC+1
END;

PROCEDURE MUL;
VAR C1,C2,C3,C4,C5:INTEGER;
BEGIN
  C1:=MEMCEFA,4J;
  C2:=MEMCEFA,4J DIV 10;
  C3:=MEMCEFA,4J DIV 100;
  C4:=MEMCEFA,4J DIV 1000;
  C5:=MEMCEFA,4J DIV 10000;
CASE LFIELD OF
  0: BEGIN
    CASE RFIELD OF
      1: ACC:=ACC*C5;
      2: ACC:=ACC*C4;
      3: ACC:=ACC*C3;
      4: ACC:=ACC*C2;
      5: ACC:=ACC*C1
    END
END
  1: BEGIN
    CASE RFIELD OF

/* THE CONTENT OF EFFECTIVE MEMORY LOCATION'S SPECIFIED FIELD TIMES (ACC) REPLACES ACC, OVERFLOW MAY OCCURRED */
/* THE VALUE OF ACC IS DIVIDED BY THE CONTENT OF EFFECTIVE
   MEMORY LOCATION'S SPECIFIED FIELD. AFTERWARD THE RESULT
   QUOTIENT IS PLACED IN ACC AND REMAINDER IS PLACED IN
   EXT. */

PROCEDURE DIVD;
VAR C1,C2,C3,C4,C5:INTEGER;
BEGIN
  C1:=MEM#EFA4;
  C2:=MEM#EFA4 DIV 10;
  C3:=MEM#EFA4 DIV 100;
  C4:=MEM#EFA4 DIV 1000;
  C5:=MEM#EFA4 DIV 10000;
  CASE LFIELD OF
  0: BEGIN
    CASE RFIELD OF
    END;
  1: BEGIN
    CASE RFIELD OF
    END;
  END;
  IF ACC>MAX1 THEN
    OV:=1;
    PC:=PC+1
END;

1: ACC:=ACC*ABS(C5);
2: ACC:=ACC*ABS(C4);
3: ACC:=ACC*ABS(C3);
4: ACC:=ACC*ABS(C2);
5: ACC:=ACC*ABS(C1)
END
END;
2: BEGIN
  CASE RFIELD OF
  2: ACC:=ACC*ABS(C4-C5*10);
  3: ACC:=ACC*ABS(C3-C5*100);
  4: ACC:=ACC*ABS(C2-C5*1000);
  5: ACC:=ACC*ABS(C1-C5*10000)
END
END;
3: BEGIN
  CASE RFIELD OF
  3: ACC:=ACC*ABS(C3-C4*10);
  4: ACC:=ACC*ABS(C2-C4*100);
  5: ACC:=ACC*ABS(C1-C4*1000)
END
END;
4: BEGIN
  CASE RFIELD OF
  4: ACC:=ACC*ABS(C2-C3*10);
  5: ACC:=ACC*ABS(C1-C3*100)
END
END;
5: ACC:=ACC*ABS(C1-C2*10)
END;
IF ACC>MAX1 THEN
  OV:=1;
  PC:=PC+1
END;

(*MUL*)
EXTJ:=ACC MOD C5;
ACCJ:=ACC DIV C5
END;

2: BEGIN
EXTJ:=ACC MOD C4;
ACCJ:=ACC DIV C4
END;

3: BEGIN
EXTJ:=ACC MOD C3;
ACCJ:=ACC DIV C3
END;

4: BEGIN
EXTJ:=ACC MOD C2;
ACCJ:=ACC DIV C2
END;

5: BEGIN
EXTJ:=ACC MOD C1;
ACCJ:=ACC DIV C1
END
END

1: BEGIN
CASE RFIELD OF
1: BEGIN
EXTJ:=ACC MOD ABS(C5);
ACCJ:=ACC DIV ABS(C5)
END;

2: BEGIN
EXTJ:=ACC MOD ABS(C4);
ACCJ:=ACC DIV ABS(C4)
END;

3: BEGIN
EXTJ:=ACC MOD ABS(C3);
ACCJ:=ACC DIV ABS(C3)
END;

4: BEGIN
EXTJ:=ACC MOD ABS(C2);
ACCJ:=ACC DIV ABS(C2)
END;

5: BEGIN
EXTJ:=ACC MOD ABS(C1);
ACCJ:=ACC DIV ABS(C1)
END
END

END
END;

2: BEGIN
CASE RFIELD OF
2: BEGIN
EXTJ:=ACC MOD ABS(C4-C5*10);
ACCJ:=ACC DIV ABS(C4-C5*10)
END;

3: BEGIN
EXTJ:=ACC MOD ABS(C3-C5*100);  
ACCJ:=ACC DIV ABS(C3-C5*100)  
END;

4: BEGIN  
EXTJ:=ACC MOD ABS(C2-C5*1000);  
ACCJ:=ACC DIV ABS(C2-C5*1000)  
END;

5: BEGIN  
EXTJ:=ACC MOD ABS(C1-C5*10000);  
ACCJ:=ACC DIV ABS(C1-C5*10000)  
END

END

3: BEGIN
CASE RFIELD OF
  3: BEGIN  
  EXTJ:=ACC MOD ABS(C3-C4*10);  
  ACCJ:=ACC DIV ABS(C3-C4*10)  
  END;
  4: BEGIN  
  EXTJ:=ACC MOD ABS(C2-C4*100);  
  ACCJ:=ACC DIV ABS(C2-C4*100)  
  END;
  5: BEGIN  
  EXTJ:=ACC MOD ABS(C1-C4*1000);  
  ACCJ:=ACC DIV ABS(C1-C4*1000)  
  END

END

END;

4: BEGIN
CASE RFIELD OF
  4: BEGIN  
  EXTJ:=ACC MOD ABS(C2-C3*10);  
  ACCJ:=ACC DIV ABS(C2-C3*10)  
  END;
  5: BEGIN  
  EXTJ:=ACC MOD ABS(C1-C3*100);  
  ACCJ:=ACC DIV ABS(C1-C3*100)  
  END

END

END;

5: BEGIN  
EXTJ:=ACC MOD ABS(C1-C2*10);  
ACCJ:=ACC DIV ABS(C1-C2*10)

END;

PC:=PC+1
END  

PROCEDURE NUMR;
BEGIN  
END  

(*DIVD*)

(*NUMR*)
PROCEDURE CHA;
BEGIN
END; /*CHA*/
/* STOP THE MACHINE */
PROCEDURE HALT;
BEGIN
END; /*HALT*/
/* THE VALUE OF ADDRESS-PART SPECIFIES THE NUMBER OF
DIGITS TO BE SHIFTED LEFT OUT OF ACC. THE SIGN OF
THE VALUE IS NOT AFFECTED. */
PROCEDURE SLA;
VAR M1,M2,M3,M4:INTEGER;
BEGIN
M1:=ACC MOD 10000;
M2:=ACC MOD 1000;
M3:=ACC MOD 100;
M4:=ACC MOD 10;
CASE ADDR OF
  1: ACC:=M1*10;
  2: ACC:=M2*100;
  3: ACC:=M3*1000;
  4: ACC:=M4*10000
END;
PC:=PC+1
END; /*SLA*/
/* SAME AS SLA, EXCEPT SHIFTED RIGHT OUT OF ACC. */
PROCEDURE SRA;
BEGIN
CASE ADDR OF
  1: ACC:=ACC DIV 10;
  2: ACC:=ACC DIV 100;
  3: ACC:=ACC DIV 1000;
  4: ACC:=ACC DIV 10000;
  5: ACC:=0
END;
PC:=PC+1
END; /*SRA*/
/* "CIRCULATING" SHIFT. IN WHICH DIGITS THAT LEAVE LEFT
END ENTER IN AT THE RIGHT END OF ACC. */
PROCEDURE SLC;
VAR A1,A2,A3,A4,M1,M2,M3,M4:INTEGER;
BEGIN
A1:=ACC DIV 10000; M1:=ACC MOD 10000;
A2:=ACC DIV 1000; M2:=ACC MOD 1000;
A3:=ACC DIV 100; M3:=ACC MOD 100;
A4:=ACC DIV 10; M4:=ACC MOD 10;
CASE ADDR OF
   1: BEGIN
      IF A1 <> 0 THEN
         ACC:=M1*10+A1
      ELSE
         ACC:=ACC*10
      END;
   2: BEGIN
      IF A2=0 THEN
         ACC:=ACC*100
      ELSE
         ACC:=M2*100+A2
      END;
   3: BEGIN
      IF A3=0 THEN
         ACC:=ACC*1000
      ELSE
         ACC:=M3*1000+A3
      END;
   4: BEGIN
      IF A4=0 THEN
         ACC:=ACC*10000
      ELSE
         ACC:=M4*10000+A4
      END
   END;
IF ACC>MAX1 THEN
   OV:=1;
   PC:=PC+1
END; (*SLC*)

/* "CIRCULATING" SHIFTr IN WHICH DIGITS THAT LEAVE RIGHT
   END ENTER IN AT THE LEFT END OF ACC. */
PROCEDURE SRC;
BEGIN
CASE ADDR OF
   1: ACC:=(ACC MOD 10)*10000+ACC DIV 10;
   2: ACC:=(ACC MOD 100)*1000+ACC DIV 100;
   3: ACC:=(ACC MOD 1000)*100+ACC DIV 1000;
   4: ACC:=(ACC MOD 10000)*10+ACC DIV 10000
END;
IF ACC>MAX1 THEN
   OV:=1;
   PC:=PC+1
END; (*SRC*)

/* THE NUMBER OF WORDS SPECIFIED BY F-PART IS MOVED,
   STARTING FROM LOCATION WHICH A-PART DENOTED TO
THE LOCATION SPECIFIED BY THE CONTENTS OF INDEX-REGISTER 1

PROCEDURE MOVE;
VAR J:INTEGER;
BEGIN
  IF FIELD>0 THEN
  BEGIN
    FOR J:=0 TO (FIELD-1) DO
      MEM[REG1+J]:=MEM[MEM[PC,1]+J];
    REG1:=REG1+FIELD
  END;
  PC:=PC+1
END;

/* THE SPECIFIED FIELD OF CONTENTS(M) REPLACED THE PREVIOUS CONTENTS OF REGISTER. */

PROCEDURE LOAD(VAR REG:INTEGER);
VAR C1,C2,C3,C4,C5:INTEGER;
BEGIN
  C1:=MEM[IFA,4];
  C2:=MEM[IFA,4] DIV 10;
  C3:=MEM[IFA,4] DIV 100;
  C4:=MEM[IFA,4] DIV 1000;
  C5:=MEM[IFA,4] DIV 10000;
CASE LFIELD OF
  0: BEGIN
    CASE RFIELD OF
      1: REG:=C5;
      2: REG:=C4;
      3: REG:=C3;
      4: REG:=C2;
      5: REG:=C1
    END
  END;
  1: BEGIN
    CASE RFIELD OF
      1: REG:=ABS(C5);
      2: REG:=ABS(C4);
      3: REG:=ABS(C3);
      4: REG:=ABS(C2);
      5: REG:=ABS(C1)
    END
  END;
  2: BEGIN
    CASE RFIELD OF
      2: REG:=ABS(C4-C5*10);
      3: REG:=ABS(C3-C5*100);
      4: REG:=ABS(C2-C5*1000);
      5: REG:=ABS(C1-C5*10000)
    END
  END;
END;
3: BEGIN
   CASE RFIELD OF
      3: REGt=ABS(C3-C4*10);
      4: REGt=ABS(C2-C4*100);
      5: REGt=ABS(C1-C4*1000)
   END
   END;
4: BEGIN
   CASE RFIELD OF
      4: REGt=ABS(C2-C3*10);
      5: REGt=ABS(C1-C3*100)
   END
   END;
5: REGt=ABS(C1-C2*10)
END;
PCt=PC+1
END; /*LOAD*/

/* SAME AS LOAD, EXCEPT THAT OPPOSITE SIGN IS LOADED */
PROCEDURE LOADN(VAR REG INTEGER);
VAR C1,C2,C3,C4,C5:INTEGER;
BEGIN
   C2:=MEM[4:4] DIV 10;
   C3:=MEM[4:4] DIV 100;
   C4:=MEM[4:4] DIV 1000;
   C5:=MEM[4:4] DIV 10000;
   CASE LFIELD OF
      0: BEGIN
         CASE RFIELD OF
            0: REGt=-C5;
            1: REGt=-C4;
            2: REGt=-C3;
            3: REGt=-C2;
            4: REGt=-C1
         END
      END;
      1: BEGIN
         CASE RFIELD OF
            1: REGt=-C5;
            2: REGt=-C4;
            3: REGt=-C3;
            4: REGt=-C2;
            5: REGt=-C1
         END
      END;
      2: BEGIN
         CASE RFIELD OF
            2: REGt=C5*10-C4;
            3: REGt=C5*100-C3;
            4: REGt=C5*1000-C2;
         END
      END;
   END
END;
CASE RFIELD OF
  3: REG := C4*10 - C3;
  4: REG := C4*100 - C2;
  5: REG := C4*1000 - C1
END
END;
4: BEGIN
CASE RFIELD OF
  4: REG := C3*10 - C2;
  5: REG := C2*10 - C1
END;
PC := PC + 1
END

PROCEDURE STORE(VAR REG: INTEGER);
VAR A1, A2, A3, A4, M1, M2, M3, M4: INTEGER;
BEGIN
  A1 := REG MOD 10;
  A2 := REG MOD 100;
  A3 := REG MOD 1000;
  A4 := REG MOD 10000;
  M1 := MEM[REG,4] MOD 10;
  M2 := MEM[REG,4] MOD 100;
  M3 := MEM[REG,4] MOD 1000;
  M4 := MEM[REG,4] MOD 10000;
CASE LFIELD OF
  0: BEGIN
    IF (REG > 0) AND (MEM[REG,4] < 0) THEN
      BEGIN
        MEM[REG,4] := ABS(MEM[REG,4]);
        M1 := ABS(M1);
        M2 := ABS(M2);
        M3 := ABS(M3);
        M4 := ABS(M4)
      END;
    IF (REG < 0) AND (MEM[REG,4] > 0) THEN
      BEGIN
        MEM[REG,4] := -MEM[REG,4];
        M1 := -M1;
        M2 := -M2;
        M3 := -M3;
        M4 := -M4
      END;
    END;
  END;
END$
CASE RFIELD OF
3: MEM[CEFA,4]J=M2+A3*100;
4: MEM[CEFA,4]J=M1+A4*10;
5: MEM[CEFA,4]J=REG
END$
END$
1: BEGIN
IF ((REG>0) AND (MEM[CEFA,4]<0)) OR
((REG<0) AND (MEM[CEFA,4]>0)) THEN
BEGIN
REG:=-REG;
A1:=-A1;
A2:=-A2;
A3:=-A3;
A4:=-A4
END$
CASE RFIELD OF
3: MEM[CEFA,4]J=MEM[CEFA,4]-M4+M1+A3*100;
4: MEM[CEFA,4]J=MEM[CEFA,4]-M4+M1+A4*100;
5: MEM[CEFA,4]J=MEM[CEFA,4]-M4+A4
END$
END$
2: BEGIN
IF ((REG>0) AND (MEM[CEFA,4]<0)) OR
((REG<0) AND (MEM[CEFA,4]>0)) THEN
BEGIN
REG:=-REG;
A1:=-A1;
A2:=-A2;
A3:=-A3;
A4:=-A4
END$
CASE RFIELD OF
4: MEM[CEFA,4]J=MEM[CEFA,4]-M4+M1+A3*100;
5: MEM[CEFA,4]J=MEM[CEFA,4]-M4+A4
END$
END$
3: BEGIN
IF ((REG>0) AND (MEM[CEFA,4]<0)) OR
((REG<0) AND (MEM[CEFA,4]>0)) THEN
BEGIN
REG:=-REG;
A1:=-A1;
A2:=-A2;
A3:=-A3;
BEGIN
  IF ((REG>0) AND (MEM[CEFA,4]<0)) OR
     ((REG<0) AND (MEM[CEFA,4]>0)) THEN
  BEGIN
    REGJ=-REG;
    A1J=-A1;
    A2J=-A2;
    A3J=-A3;
    A4J=-A4
  END;
  CASE RFIELD OF
    4: MEM[CEFA,4]=MEM[CEFA,4]-M2+M1+A1*100;
    5: MEM[CEFA,4]=MEM[CEFA,4]-M3+A3
  END
  END;

5: BEGIN
  IF ((REG>0) AND (MEM[CEFA,4]<0)) OR
     ((REG<0) AND (MEM[CEFA,4]>0)) THEN
  BEGIN
    REGJ=-REG;
    A1J=-A1;
    A2J=-A2;
    A3J=-A3;
    A4J=-A4
  END;
  MEM[CEFA,4]=MEM[CEFA,4]-M1+A1
  END
END;
PC:=PC+1
END;

/* SAME AS STORE, EXCEPT ZERO IS STORED( THE SPECIFIED
   FIELD OF CONTENTS(M) IS CLEARED TO ZERO). */

PROCEDURE STZ;
VAR A1, A2, A3, A4, M1, M2, M3, M4: INTEGER;
BEGIN
  A1:=MEM[CEFA,4] DIV 10*10;
  A2:=MEM[CEFA,4] DIV 100*100;
  A3:=MEM[CEFA,4] DIV 1000*1000;
  A4:=MEM[CEFA,4] DIV 10000*10000;
  M1:=MEM[CEFA,4] MOD 10;
  M2:=MEM[CEFA,4] MOD 100;
  M3:=MEM[CEFA,4] MOD 1000;
M4: = MEM[CEFA, 4] MOD 10000;
CASE LFIELD OF
  0: BEGIN
    CASE RFIELD OF
      1: MEM[CEFA, 4] = M4;
      2: MEM[CEFA, 4] = M3;
      3: MEM[CEFA, 4] = M2;
      4: MEM[CEFA, 4] = M1;
      5: MEM[CEFA, 4] = 0
    END
  END
  1: BEGIN
    CASE RFIELD OF
      1: MEM[CEFA, 4] = M4;
      2: MEM[CEFA, 4] = M3;
      3: MEM[CEFA, 4] = M2;
      4: MEM[CEFA, 4] = M1;
      5: MEM[CEFA, 4] = 0
    END
  END
  2: BEGIN
    CASE RFIELD OF
      2: MEM[CEFA, 4] = M3 + A4;
      3: MEM[CEFA, 4] = M2 + A4;
      4: MEM[CEFA, 4] = M1 + A4;
      5: MEM[CEFA, 4] = A4
    END
  END
  3: BEGIN
    CASE RFIELD OF
      3: MEM[CEFA, 4] = M2 + A3;
      4: MEM[CEFA, 4] = M1 + A3;
      5: MEM[CEFA, 4] = A3
    END
  END
  4: BEGIN
    CASE RFIELD OF
      4: MEM[CEFA, 4] = M1 + A2;
      5: MEM[CEFA, 4] = A2
    END
  END
  5: MEM[CEFA, 4] = A1
END;
PC: = PC + 1
END

/* INPUT, THE NUMBER OF F-PART SPECIFIED HOW MANY DATA WE WANT TO READ. */

PROCEDURE INP:
VAR J: INTEGER;
BEGIN
CLOSE(F);
FOR J:=0 TO (FIELD-1) DO
  READLN(MEM[MEM[PC,1]+J,4]);
  PC:=PC+1
END;

PROCEDURE OUT;
VAR J:INTEGER;
BEGIN
  IF INDEX=7 THEN
    BEGIN
      FOR J:=0 TO (FIELD-1) DO
        WRITE(MEM[MEM[PC,1]+J,4]);
      WRITELN
    END
  ELSE
    BEGIN
      FOR J:=0 TO (FIELD-1) DO
        WRITE(MEM[FA+J,4]);
      WRITELN
    END
  PC:=PC+1
END;

PROCEDURE JMP;
BEGIN
  REGJ:=PC+1;
  PC:=MEM[PC,1]
END;

PROCEDURE JSJ;
BEGIN
  PC:=MEM[PC,1]
END;

PROCEDURE JOV;
BEGIN
  REGJ:=PC+1;
  IF OV=1 THEN
    BEGIN
      PC:=MEM[PC,1]
    END
END;

/* UNCONDITIONAL JUMP */
PROCEDURE JMP;
BEGIN
REGJ:=PC+1;
PC:=MEM[PC,1]
END;

/* SAME AS JMP, BUT THE CONTENTS OF REGJ ARE UNCHANGED */
PROCEDURE JSJ;
BEGIN
PC:=MEM[PC,1]
END;

/* JUMP ON OVERFLOW */
PROCEDURE JOV;
BEGIN
  REGJ:=PC+1;
  IF OV=1 THEN
    BEGIN
      PC:=MEM[PC,1]
    END
END;

/* OUTPUT, THE NUMBER OF F-PART SPECIFIED HOW MANY DATA
   WE WANT TO PRINT. IF I-PART IS 7 THEN THE SPECIFIED
   CHARACTERS IS PRINT. */

PROCEDURE OUT;
VAR J:INTEGER;
BEGIN
  IF INDEX=7 THEN
    BEGIN
      FOR J:=0 TO (FIELD-1) DO
        WRITE(MEM[MEM[PC,1]+J,4]);
      WRITELN
    END
  ELSE
    BEGIN
      FOR J:=0 TO (FIELD-1) DO
        WRITE(MEM[FA+J,4]);
      WRITELN
    END
  PC:=PC+1
END;

PROCEDURE JMP;
BEGIN
  REGJ:=PC+1;
  PC:=MEM[PC,1]
END;

PROCEDURE JSJ;
BEGIN
  PC:=MEM[PC,1]
END;

PROCEDURE JOV;
BEGIN
  REGJ:=PC+1;
  IF OV=1 THEN
    BEGIN
      PC:=MEM[PC,1]
    END
END;

/* UNCONDITIONAL JUMP */
PROCEDURE JMP;
BEGIN
REGJ:=PC+1;
PC:=MEM[PC,1]
END;

/* SAME AS JMP, BUT THE CONTENTS OF REGJ ARE UNCHANGED */
PROCEDURE JSJ;
BEGIN
PC:=MEM[PC,1]
END;

/* JUMP ON OVERFLOW */
PROCEDURE JOV;
BEGIN
  REGJ:=PC+1;
  IF OV=1 THEN
    BEGIN
      PC:=MEM[PC,1]
    END
END;

/* UNCONDITIONAL JUMP */
PROCEDURE JMP;
BEGIN
REGJ:=PC+1;
PC:=MEM[PC,1]
END;

/* SAME AS JMP, BUT THE CONTENTS OF REGJ ARE UNCHANGED */
PROCEDURE JSJ;
BEGIN
PC:=MEM[PC,1]
END;

/* JUMP ON OVERFLOW */
PROCEDURE JOV;
BEGIN
  REGJ:=PC+1;
  IF OV=1 THEN
    BEGIN
      PC:=MEM[PC,1]
    END
END;
OV:=0
END
ELSE
PC:=PC+1
END; /*JOV*/

/* JUMP ON NO OVERFLOW */
PROCEDURE JNOV;
BEGIN
REGJ:=PC+1;
IF OV=0 THEN
PC:=MEM[PC+1]
ELSE
BEGIN
OV:=0;
PC:=PC+1
END
END; /*JNOV*/

/* JUMP ON CONDITION LESS */
PROCEDURE JL;
BEGIN
REGJ:=PC+1;
IF CI=3 THEN
PC:=MEM[PC+1]
ELSE
PC:=PC+1
END; /*JL*/

/* JUMP ON CONDITION EQUAL */
PROCEDURE JE;
BEGIN
REGJ:=PC+1;
IF CI=1 THEN
PC:=MEM[PC+1]
ELSE
PC:=PC+1
END; /*JE*/

/* JUMP ON CONDITION GREATER */
PROCEDURE JG;
BEGIN
REGJ:=PC+1;
IF CI=2 THEN
PC:=MEM[PC+1]
ELSE
PC:=PC+1
PROCEDURE JGE;
BEGIN
  REGJJ:=PC+1;
  IF (CI=1) OR (CI=2) THEN
    PC:=MEM[PC,1]
  ELSE
    PC:=PC+1
END;

PROCEDURE JNE;
BEGIN
  REGJJ:=PC+1;
  IF (CI=2) OR (CI=3) THEN
    PC:=MEM[PC,1]
  ELSE
    PC:=PC+1
END;

PROCEDURE JLE;
BEGIN
  REGJJ:=PC+1;
  IF (CI=1) OR (CI=3) THEN
    PC:=MEM[PC,1]
  ELSE
    PC:=PC+1
END;

PROCEDURE JRN(VAR REG:INTEGER);
BEGIN
  REGJJ:=PC+1;
  IF REG<0 THEN
    PC:=MEM[PC,1]
  ELSE
    PC:=PC+1
END;

PROCEDURE JRZ(VAR REG:INTEGER);
BEGIN
  REGJJ:=PC+1;
  IF REG=0 THEN

PC:=MEM[PC,1] ELSE PC:=PC+1 END; (*JRZ*)

/* JUMP IF CONTENTS OF REGISTER IS POSITIVE */
PROCEDURE JRP(VAR REG:INTEGER); BEGIN REGJ:=PC+1; IF REG>0 THEN PC:=MEM[PC,1] ELSE PC:=PC+1 END; (*JRP*)

/* JUMP IF CONTENTS OF REGISTER IS NON-NEGATIVE */
PROCEDURE JRNN(VAR REG:INTEGER); BEGIN REGJ:=PC+1; IF REG>=0 THEN PC:=MEM[PC,1] ELSE PC:=PC+1 END; (*JRNN*)

/* JUMP IF CONTENTS OF REGISTER IS NON-POSITIVE */
PROCEDURE JRNP(VAR REG:INTEGER); BEGIN REGJ:=PC+1; IF REG<=0 THEN PC:=MEM[PC,1] ELSE PC:=PC+1 END; (*JRNP*)

/* JUMP IF CONTENTS OF REGISTER IS NON-ZERO */
PROCEDURE JRNZ(VAR REG:INTEGER); BEGIN REGJ:=PC+1; IF REG<>0 THEN PC:=MEM[PC,1] ELSE PC:=PC+1 END; (*JRNZ*)

/* THE QUANTITY OF A-PART IS ADDED TO REGISTER */
PROCEDURE INCR(VAR REG:INTEGER); BEGIN REG:=REG+MEM[PC,1]
IF \text{REG} > \text{MAX1} \text{ THEN}
\begin{align*}
\text{OV} & = 1; \\
\text{PC} & = \text{PC} + 1 \\
\text{END} & \quad (\ast \text{INCR}\ast)
\end{align*}

\text{/* THE QUANTITY OF A-PART IS SUBTRACTED FROM REGISTER */}

\text{PROCEDURE DECR(VAR \text{REGJ}\text{INTEGER})\;}
\begin{align*}
\text{BEGIN} \\
\text{REG} & = \text{REG} - \text{MEM}[\text{PC}, 1]; \\
\text{PC} & = \text{PC} + 1; \\
\text{END} & \quad (\ast \text{DECR}\ast)
\end{align*}

\text{/* THE QUANTITY OF A-PART IS LOADED INTO REGISTER */}

\text{PROCEDURE ENTR(VAR \text{REGJ}\text{INTEGER})\;}
\begin{align*}
\text{BEGIN} \\
\text{CASE INDEX OF} \\
0: & \text{REG} = \text{MEM}[\text{PC}, 1]; \\
1: & \text{REG} = \text{MEM}[\text{PC}, 1] + \text{REG1}; \\
2: & \text{REG} = \text{MEM}[\text{PC}, 1] + \text{REG2}; \\
3: & \text{REG} = \text{MEM}[\text{PC}, 1] + \text{REG3}; \\
4: & \text{REG} = \text{MEM}[\text{PC}, 1] + \text{REG4}; \\
5: & \text{REG} = \text{MEM}[\text{PC}, 1] + \text{REG5}; \\
6: & \text{REG} = \text{MEM}[\text{PC}, 1] + \text{REG6}; \\
\text{END} \\
\text{PC} & = \text{PC} + 1 \\
\text{END} & \quad (\ast \text{ENTR}\ast)
\end{align*}

\text{/* SAME AS ENTR, EXCEPT THAT OPPOSITE SIGN IS LOADED */}

\text{PROCEDURE ENNR(VAR \text{REGJ}\text{INTEGER})\;}
\begin{align*}
\text{BEGIN} \\
\text{CASE INDEX OF} \\
0: & \text{REG} = -\text{MEM}[\text{PC}, 1]; \\
1: & \text{REG} = -\text{MEM}[\text{PC}, 1] - \text{REG1}; \\
2: & \text{REG} = -\text{MEM}[\text{PC}, 1] - \text{REG2}; \\
3: & \text{REG} = -\text{MEM}[\text{PC}, 1] - \text{REG3}; \\
4: & \text{REG} = -\text{MEM}[\text{PC}, 1] - \text{REG4}; \\
5: & \text{REG} = -\text{MEM}[\text{PC}, 1] - \text{REG5}; \\
6: & \text{REG} = -\text{MEM}[\text{PC}, 1] - \text{REG6}; \\
\text{END} \\
\text{PC} & = \text{PC} + 1 \\
\text{END} & \quad (\ast \text{ENNR}\ast)
\end{align*}

\text{/* THE SPECIFIED FIELD OF REGISTER IS COMPARED WITH THE \;}
\text{SAME FIELD OF CONTENTS, THE COMPARISON INDICATOR \;}
\text{IS THEN SET TO LESS, EQUAL, OR GREATER ACCORDING TO \;}
\text{THE COMPARED RESULT. */}

\text{PROCEDURE CMPR(VAR \text{REGJ}\text{INTEGER})\;}
\text{VAR R1, R2, R3, R4, R5;}

\text{PROCEDURE CMPR(VAR \text{REGJ}\text{INTEGER})\;}
\text{VAR R1, R2, R3, R4, R5;
C1, C2, C3, C4, C5: INTEGER;
BEGIN
C1 := MEM[04A, 43] DIV 10;
C2 := MEM[04A, 43] DIV 100;
C3 := MEM[04A, 43] DIV 1000;
C4 := MEM[04A, 43] DIV 10000;
R1 := REG;
R2 := REG DIV 10;
R3 := REG DIV 100;
R4 := REG DIV 1000;
R5 := REG DIV 10000;
CASE LFIELD OF
0 BEGIN
CASE RFIELD OF
1 BEGIN
IF R5 = C5 THEN CI := 1;
IF R5 > C5 THEN CI := 2;
IF R5 < C5 THEN CI := 3
END;
2 BEGIN
IF R4 = C4 THEN CI := 1;
IF R4 > C4 THEN CI := 2;
IF R4 < C4 THEN CI := 3
END;
3 BEGIN
IF R3 = C3 THEN CI := 1;
IF R3 > C3 THEN CI := 2;
IF R3 < C3 THEN CI := 3
END;
4 BEGIN
IF R2 = C2 THEN CI := 1;
IF R2 > C2 THEN CI := 2;
IF R2 < C2 THEN CI := 3
END;
5 BEGIN
IF R1 = C1 THEN CI := 1;
IF R1 > C1 THEN CI := 2;
IF R1 < C1 THEN CI := 3
END
END
1 BEGIN
CASE RFIELD OF
1 BEGIN
IF ABS(R5) = ABS(C5) THEN CI := 1;
IF ABS(R5) > ABS(C5) THEN CI := 2;
IF ABS(R5) < ABS(C5) THEN CI := 3
END;
2 BEGIN
IF ABS(R4) = ABS(C4) THEN CI := 1;
IF ABS(R4) > ABS(C4) THEN CI := 2;
IF ABS(R4)<ABS(C4) THEN CI:=3
END;
3: BEGIN
  IF ABS(R3)=ABS(C3) THEN CI:=1;
  IF ABS(R3)>ABS(C3) THEN CI:=2;
  IF ABS(R3)<ABS(C3) THEN CI:=3
END;
4: BEGIN
  IF ABS(R2)=ABS(C2) THEN CI:=1;
  IF ABS(R2)>ABS(C2) THEN CI:=2;
  IF ABS(R2)<ABS(C2) THEN CI:=3
END;
5: BEGIN
  IF ABS(R1)=ABS(C1) THEN CI:=1;
  IF ABS(R1)>ABS(C1) THEN CI:=2;
  IF ABS(R1)<ABS(C1) THEN CI:=3
END
END;
2: BEGIN
CASE RFIELD
  2: BEGIN
    IF ABS(R4-R5*10)=ABS(C4-C5*10) THEN CI:=1;
    IF ABS(R4-R5*10)>ABS(C4-C5*10) THEN CI:=2;
    IF ABS(R4-R5*10)<ABS(C4-C5*10) THEN CI:=3
END;
  3: BEGIN
    IF ABS(R3-R5*100)=ABS(C3-C5*100) THEN CI:=1;
    IF ABS(R3-R5*100)>ABS(C3-C5*100) THEN CI:=2;
    IF ABS(R3-R5*100)<ABS(C3-C5*100) THEN CI:=3
END;
  4: BEGIN
    IF ABS(R2-R5*1000)=ABS(C2-C5*1000) THEN CI:=1;
    IF ABS(R2-R5*1000)>ABS(C2-C5*1000) THEN CI:=2;
    IF ABS(R2-R5*1000)<ABS(C2-C5*1000) THEN CI:=3
END;
  5: BEGIN
    IF ABS(R1-R5*10000)=ABS(C1-C5*10000) THEN CI:=1;
    IF ABS(R1-R5*10000)>ABS(C1-C5*10000) THEN CI:=2;
    IF ABS(R1-R5*10000)<ABS(C1-C5*10000) THEN CI:=3
END
END;
3: BEGIN
  CASE RFIELD OF
    3: BEGIN
      IF ABS(R3-R4*10)=ABS(C3-C4*10) THEN CI:=1;
      IF ABS(R3-R4*10)>ABS(C3-C4*10) THEN CI:=2;
      IF ABS(R3-R4*10)<ABS(C3-C4*10) THEN CI:=3
END;
    4: BEGIN
      IF ABS(R2-R4*100)=ABS(C2-C4*100) THEN CI:=1;
IF \( \text{ABS}(R2-R4*100) > \text{ABS}(C2-C4*100) \) THEN \( CI := 2 \);
IF \( \text{ABS}(R2-R4*100) < \text{ABS}(C2-C4*100) \) THEN \( CI := 3 \)
END;

5: BEGIN
IF \( \text{ABS}(R1-R4*1000) = \text{ABS}(C1-C4*1000) \) THEN \( CI := 1 \);
IF \( \text{ABS}(R1-R4*1000) > \text{ABS}(C1-C4*1000) \) THEN \( CI := 2 \);
IF \( \text{ABS}(R1-R4*1000) < \text{ABS}(C1-C4*1000) \) THEN \( CI := 3 \)
END;
END

4: BEGIN
CASE RFIELD OF
4: BEGIN
IF \( \text{ABS}(R2-R3*10) = \text{ABS}(C2-C3*10) \) THEN \( CI := 1 \);
IF \( \text{ABS}(R2-R3*10) > \text{ABS}(C2-C3*10) \) THEN \( CI := 2 \);
IF \( \text{ABS}(R2-R3*10) < \text{ABS}(C2-C3*10) \) THEN \( CI := 3 \)
END;
5: BEGIN
IF \( \text{ABS}(R1-R3*100) = \text{ABS}(C1-C3*100) \) THEN \( CI := 1 \);
IF \( \text{ABS}(R1-R3*100) > \text{ABS}(C1-C3*100) \) THEN \( CI := 2 \);
IF \( \text{ABS}(R1-R3*100) < \text{ABS}(C1-C3*100) \) THEN \( CI := 3 \)
END
END

5: BEGIN
IF \( \text{ABS}(R1-R2*10) = \text{ABS}(C1-C2*10) \) THEN \( CI := 1 \);
IF \( \text{ABS}(R1-R2*10) > \text{ABS}(C1-C2*10) \) THEN \( CI := 2 \);
IF \( \text{ABS}(R1-R2*10) < \text{ABS}(C1-C2*10) \) THEN \( CI := 3 \)
END
PC:=PC+1
END

(*CMPR*)

/* GET THE_OPCODE AND FIELD VALUE FROM CURRENT INSTRUCTION LOCATION AND INVOKE VARIOUS PROCEDURE TO PERFORM ACTION */
PROCEDURE OPERATION;
BEGIN
CASE OPCODE OF
0: NOP;
1: ADD;
2: SUB;
3: MUL;
4: DIV;
5: BEGIN
CASE FIELD OF
0: NUMR;
1: CHA;
2: HALT
END
END;
6: BEGIN
CASE FIELD OF
0: SLA;
1: SRA;
2: SLC;
3: SRC
END
END;
7: MOVE;
8: LOAD(ACC);
9: LOAD(REG1);
10: LOAD(REG2);
11: LOAD(REG3);
12: LOAD(REG4);
13: LOAD(REG5);
14: LOAD(REG6);
15: LOAD(EXT);
16: LOADN(ACC);
17: LOADN(REG1);
18: LOADN(REG2);
19: LOADN(REG3);
20: LOADN(REG4);
21: LOADN(REG5);
22: LOADN(REG6);
23: LOADN(EXT);
24: STORE(ACC);
25: STORE(REG1);
26: STORE(REG2);
27: STORE(REG3);
28: STORE(REG4);
29: STORE(REG5);
30: STORE(REG6);
31: STORE(EXT);
32: STORE(REGJ);
33: STZ;
36: INP;
37: OUT;
39: BEGIN
CASE FIELD OF
0: JMP;
1: JSJ;
2: JOV;
3: JNOV;
4: JL;
5: JE;
6: JG;
7: JGE;
8: JNE;
9: JLE
END
END;
40: BEGIN
CASE FIELD OF
CASE FIELD OF
0: JRN(REG1)
1: JRZ(REG1)
2: JRP(REG1)
3: JRNN(REG1)
4: JRNZ(REG1)
5: JRNP(REG1)
END
END;

CASE FIELD OF
0: JRN(REG2)
1: JRZ(REG2)
2: JRP(REG2)
3: JRNN(REG2)
4: JRNZ(REG2)
5: JRNP(REG2)
END
END;

CASE FIELD OF
0: JRN(REG3)
1: JRZ(REG3)
2: JRP(REG3)
3: JRNN(REG3)
4: JRNZ(REG3)
5: JRNP(REG3)
END
END;

CASE FIELD OF
0: JRN(REG4)
1: JRZ(REG4)
2: JRP(REG4)
3: JRNN(REG4)
4: JRNZ(REG4)
5: JRNP(REG4)
END
END;

CASE FIELD OF
0: JRN(REG5)
1: JRZ(REG5)
2: JRP(REG5);
3: JRNN(REG5);
4: JRNZ(REG5);
5: JRNP(REG5)
END
END;
46: BEGIN
CASE FIELD OF
  0: JRN(REG6);
  1: JRZ(REG6);
  2: JRP(REG6);
  3: JRNN(REG6);
  4: JRNZ(REG6);
  5: JRNP(REG6)
END
END;
47: BEGIN
CASE FIELD OF
  0: JRN(EXT);
  1: JRZ(EXT);
  2: JRP(EXT);
  3: JRNN(EXT);
  4: JRNZ(EXT);
  5: JRNP(EXT)
END
END;
48: BEGIN
CASE FIELD OF
  0: INCR(ACC);
  1: DECR(ACC);
  2: ENTR(ACC);
  3: ENNR(ACC)
END
END;
49: BEGIN
CASE FIELD OF
  0: INCR(REG1);
  1: DECR(REG1);
  2: ENTR(REG1);
  3: ENNR(REG1)
END
END;
50: BEGIN
CASE FIELD OF
  0: INCR(REG2);
  1: DECR(REG2);
  2: ENTR(REG2);
  3: ENNR(REG2)
END
END;
51: BEGIN
CASE FIELD OF
BEGIN

0: INCR(REG3);
1: DECR(REG3);
2: ENTR(REG3);
3: ENNR(REG3)
END

52: BEGIN
CASE FIELD OF
0: INCR(REG4);
1: DECR(REG4);
2: ENTR(REG4);
3: ENNR(REG4)
END
END;

53: BEGIN
CASE FIELD OF
0: INCR(REG5);
1: DECR(REG5);
2: ENTR(REG5);
3: ENNR(REG5)
END
END;

54: BEGIN
CASE FIELD OF
0: INCR(REG6);
1: DECR(REG6);
2: ENTR(REG6);
3: ENNR(REG6)
END
END;

55: BEGIN
CASE FIELD OF
0: INCR(EXT);
1: DECR(EXT);
2: ENTR(EXT);
3: ENNR(EXT)
END
END;

56: CMPR(ACC);
57: CMPR(REG1);
58: CMPR(REG2);
59: CMPR(REG3);
60: CMPR(REG4);
61: CMPR(REG5);
62: CMPR(REG6);
63: CMPR(EXT)
END
END;

(*OPERATION*)

BEGIN

(*EXECUTION*)
REPEAT
  OPCODE:=MEM[PC,4];
  FIELD:=MEM[PC,3];
  INDEX:=MEM[PC,2];
  ADDR:=MEM[PC,1];
  CASE INDEX OF
    0: EFA:=ADDR;
    1: EFA:=ADDR+REG1;
    2: EFA:=ADDR+REG2;
    3: EFA:=ADDR+REG3;
    4: EFA:=ADDR+REG4;
    5: EFA:=ADDR+REG5;
    6: EFA:=ADDR+REG6
  END;
  RFIELD:=FIELD MOD 8;
  LFIELD:=FIELD DIV 8;
  OPERATION
  UNTIL (MEM[PC,4]=5) AND (MEM[PC,3]=2)
END;

/* PRINT THE CURRENT TIME BEFORE THE RESULTS ARE PRINTED */
PROCEDURE PRINTTIME;
VAR HRS,MINS:INTEGER;
  AP: ARRAY[1..2] OF CHAR;
BEGIN
  MINS:=ROUND(TIME*60);
  HRS:=MINS DIV 60;
  MINS:=MINS MOD 60;
  IF HRS<12 THEN
    AP:=‘AM’
  ELSE
    IF (HRS=12) AND (MINS=0) THEN
      AP:=‘AM’
    ELSE
      AP:=‘PM’;
    WRITE(('(',HRS+11) MOD 12+1):2);
    WRITE(‘’,MINS DIV 10:1,MINS MOD 10:1,AP:3);
    Writeln(chr(7))
END;

/****MAIN PROGRAM****/
BEGIN
  FOR J:=0 TO MAX2 DO
    BEGIN
      (*INITIATED MEMORY*)
MEM[1] = 0;
MEM[2] = 0;
MEM[3] = 0;
MEM[4] = 0;

END;

(*INITIATED ALL REGISTERS*)

ACC := 0;
EXT := 0;
REG1 := 0;
REG2 := 0;
REG3 := 0;
REG4 := 0;
REG5 := 0;
REG6 := 0;
REG7 := 0;
OU := 0;
CI := 0;
PC := 0;
WRITELN;
TRANSLATOR;
WRITELN;
WRITELN(’*ERRORS DETECTED: 0’);
WRITE(’*’, FNAME,’ ’);
PRINTTIME;
WRITELN;
EXECUTION;
WRITELN;
100:  
END.  

(*MAL*)
There are four example programs in this chapter. These example programs will demonstrate how to use the program - MAL.PAS.

First of all, there must exist a file which was written in MIX-LIKE ASSEMBLY LANGUAGE in users account. When MAL asks for input file, the user must give a correct file name. In example program 4, because the user gave the wrong name for the input file (i.e. It does not exist in users account), MAL will keeps on asking for input file until the given name is found. After this, the file will be printed out with the line number in front of each line.

If users assembly language has no errors (see previous chapters for the rules and formatted forms), MAL will prints out two lines of messages (i.e. first, *ERRORS DETECTED: 0; second, the file name followed by the exact time when computer prints out these messages), and then the results.

If there exist some errors, MAL will prints out the error codes immediately follow the error line (see example program 4), and MAL will tells user how many errors the file have. If there do exist errors, MAL will not do anything further.
EXAMPLE PROGRAM 1

INPUT FILE? PRIM

1 * EXAMPLE PROGRAM.....FIRST FIVE HUNDRED PRIMES.
2 * ALGORITHM OF THE PROGRAM:
4 * N WILL RUN THROUGH THE ODD NUMBERS WHICH ARE
5 * CANDIDATES FOR PRIMES; J KEEPS TRACK OF HOW
6 * MANY PRIMES HAVE BEEN FOUND SO FAR.
7 * STEP2: SET J <-- J+1, PRIME[J] <-- N.
8 * STEP3: IF J=500, GO TO STEP9.
9 * STEP4: SET N <-- N+2.
10 * STEP5: SET K <-- 2.
11 * STEP6: DIVIDE N BY PRIME K ; LET Q BE THE QUOTIENT AND
12 * R THE REMAINDER. IF R=0, N IS NOT PRIME, GOTO 4.
13 * STEP7: IF Q=PRIME[K], GO TO STEP2.
14 * STEP8: INCREASE K BY ONE, AND GO TO STEP6.
15 * STEP9: PRINT THE TITLE AND THE TABLE.

REGISTER ASSIGNMENT:
17 * REG1=J-500, REG2=N, REG3=K, REG4 INDICATES B.
18 L EQU 500 NUMBER OF PRIMES TO FIND
19 PRIME EQU -1 MEMORY AREA FOR TABLE OF PRIMES
20 BUF0 EQU 1000 MEMORY AREA FOR THE TITLE
21 ORIG 1500 START LOCATION FOR INSTRUCTIONS
22 START ENT1 1-L START TABLE, J <-- J+1
23 ENT2 3 N <-- 3
24 LOOP1 INC1 1 N IS PRIME, J <-- J+1
25 ST2 PRIME+L,1
26 J1Z PRINT
27 LOOP2 INC2 2
28 ENT3 2
29 LOOP3 ENTA 0,2
30 DIV PRIME,3
31 JXZ LOOP2
32 CMPA PRIME,3
33 INC3 1
34 JG LOOP3
35 JMP LOOP1
36 PRINT OUT TITLE,7(7)
37 OUT PRIME+1(500)
38 HLT
39 ORIG PRIME+1
40 CON 2
41 ORIG BUFO
42 TITLE ALF FIRS
43 ALF T FI
44 ALF VE H
45 ALF UNDR
46 ALF ED P
47 ALF PRIME
48 ALF S
49 END START

PRIME[J] <-- N
500 PRIMES FOUND?
ADVANCE N
K <-- 2
PRIME[K]/N?
R=0?
PRIME[K] LARGE?
ADVANCE K
JUMP IF Q > PRIME[K]
OTHERWISE N IS PRIME
PRINT TITLE
PRINT 500 PRIMES
STOP
FIRST PRIME IS 2
ALPHABETIC INFORMATION FOR TITLE LINE
END OF THE PROGRAM

*ERRORS DETECTED: 0
FIRST FIVE HUNDRED PRIMES

2  3  5  7 11 13 . . . 23
29 31 37 41 43 47 . . . 61
. . . . . . . . . . .
3469 3491 3499 3511 3517 3527 . . . 3539
3541 3547 3557 3559 3571

EXAMPLE PROGRAM 2

INPUT FILE? MAXI

1 * EXAMPLE PROGRAM.....FIND THE MAXIMUM

2 * ALGORITHM OF THE PROGRAM:

3 * STEP1: SET J <-- N, K <-- N-1, M <-- X(N).

4 * STEP2: IF K=0, GO TO STEP6.

5 * STEP3: IF X(K)=M, GO TO STEP5.

6 * STEP4: SET J <-- K, M <-- X[N].

7 * STEP5: DECREASE K BY ONE, RETURN TO STEP2.

8 * STEP6: STOP.

9 * REGISTER ASSIGNMENT:

10 * ACC=M, REG1=N, REG2=J, REG3=K.

11 X EQU 1000

12 ORIG 1500

13 OUT T1,7(4) PRINT THE INPUT TITLE

14 IN X+1(20) INPUT

15 ENT1 20

16 ENT3 0,1

17 JMP CHANGE J <-- N, M <-- X[N], K <-- N-1
18 LOOP CMPA X,3 COMPARE
19 JGE DECR
20 CHANGE ENT2 O,3 CHANGE M, J <-- k
21 LDA X,3 M <-- X[K]
22 DECR DEC3 1 DECREASE K
23 J3P LOOP ALL TESTED?
24 STA 1
25 OUT T2,7(4) PRINT THE OUTPUT TITLE
26 OUT 1(1) PRINT THE MAXIMUM
27 HLT
28 ORIG 500
29 T1 ALF THE INPUT TITLE
30 ALF NUMB
31 ALF ERS
32 ALF ARE:
33 ORIG 600 OUTPUT TITLE
34 T2 ALF THE
35 ALF MAXI
36 ALF MUM
37 ALF IS:
38 END END OF THE PROGRAM
*ERRORS DETECTED: 0
*MAXI 10:14 AM
THE NUMBERS ARE:
234 367 9 20 4000 102 5 92 702 1090 233 5555 7 0 47 29 1 2 3 5551
THE MAXIMUM IS: 5555
EXAMPLE PROGRAM 3

INPUT FILE? SIDE

1 * EXAMPLE PROGRAM.....SIDWAYS ADDITION
2 * THIS SIMPLE PROGRAM WILL GIVES THE SUM OF THE
3 * DIGITS OF YOUR INPUT NUMBER.

        ORIG 100
4
5        IN  20(1)
6
7        LDA 20(5:5)
8        ADD 20(4:4)
9        ADD 20(3:3)
10       ADD 20(2:2)
11       ADD 20(1:1)
12
13       STA 20
14
15       OUT 20(1)
16
17       HLT
18
19       END

*ERRORS DETECTED:  0

*SIDE

10:17 AM

31987

28

EXAMPLE PROGRAM 4

INPUT FILE? A

INPUT FILE? B

INPUT FILE? C

INPUT FILE? TEST

1 * EXAMPLE PROGRAM.....ERRORS TEST
2 *
3 ORIG 1000
**** ^ 7
4 IN 20(20)
**** ^ 6
**** ^ 7
5 LABEL12 ENT1 0,3
**** ^ 3
**** ^ 4
**** ^ 7
6 ENT2 0,1
7 LOOP DEC 1
**** ^ 2
**** ^ 4
8 OUT 20,7(20)
**** ^ 7
9 23L ADD 100
**** ^ 1
10 SUB 100
11 MUL 230
**** ^ 4
12 HLT
13 END
*ERRORS DETECTED: 11