The Essence of Computer Science

by

J. E. L. Peck

Technical Manual 75-7

October 1975

Department of Computer Science
University of British Columbia
Vancouver, B. C.

cardset ✓

subj – Computers-general

The Essence of Computer Science

J. E. L. Peck

Preface

The aim of this little booklet is to explore the possibility, for the teaching of computer science, of the language BCPL and its associated ideal machine, which was originally developed for portability.

All too often it happens that we introduce computer science by teaching a high level language, and an assembler language, where the two are unrelated. How many instructors, for example, have ever seen (let alone understood) the compiler for their favourite language. Sometimes we use a fictitious machine to teach about machine structure because the actual machine is too complicated. The language MIX is an example. But how many compilers are written in MIX, and is there a translator from some high level language into MIX? In short, our complaint is that the vehicles for the teaching of computer science are disjointed.

The BCPL - MINICODE system seems to offer a unique answer to the usual dilemma of what to teach. Here we have one high level language in which a whole system is written. There is a translator, written in BCPL, from BCPL to an assembler language for a simple machine. We have also an assembler and a loader and interpreter for this machine written in BCPL. Thus, in theory, with a small loader and interpreter, we have a complete portable system which we can use for instruction. Another point is that BCPL, despite some criticisms we may have such as lack of types, contains control structures well suited to the teaching of safe programming techniques. Those who have some belief that a unified complete system is the way to go, may find these few pages of interest.

Vancouver
1975 July.

Chapter I

The Computer

Chapter one is a description of an ideal computer, its construction, and the operations that it may perform.

1.1. Introduction

In this exposition we shall describe the essence of computer science by using one high level language and a related ideal machine on which it may be implemented. As we progress we shall discover that this is all that we need in order to explain the basic concepts of machine architecture, instruction sets, assembler language, assemblers, loaders, interpreters, compilers, subroutines, functions, parameter passing, recursion and the like.

1.2. Number systems

In the system which we shall describe, it will often be necessary to write integers in more than one base. In fact, the three ways of writing integers that will frequently be used are the familiar decimal representation (base 10), the octal representation (base 8) and the binary representation (base 2). Since integers in each of these three ways will be scattered throughout the text, we need some convenient way in which to distinguish some of them. Accordingly, we shall adopt the convention that an octal integer (base 8) is preceded by an octothorp, "#". Thus we may write 33 = #41 and understand that it means 33 in base 10 equals 41 in base 8. Naturally 7 = #7, so that for some small integers it will not matter how we write them. For binary integers (base 2) we shall not need a particular convention since it is usually clear from the context what is meant. An integer in which the digits are only 0 and 1 is often in binary (base 2).

1.3. An ideal computer

The computer[1] that we describe here does not need to exist as an actual piece of hardware, for it can always be simulated or emulated. However, for the purpose of describing the essence of computing, we may think of it as a piece of hardware. Its two main components are:

a) a memory consisting of a sequence of storage cells, and
b) some registers (or special storage cells) where most of the work is done.

Each of these storage cells, both in the memory and in the reg-

--- [1] This computer was first described by Martin Richards [R1].

isters, consists of a number of bits each of which is either "on" (1) or "off" (0). Each bit may be considered as a binary digit, and in this way an integer can be represented in a cell in binary. The actual number of bits in each storage cell, the "cell size", is a matter of taste, and we shall not specify it here. You will find that cell sizes on some typical machines are 64, 32, 24, 16 and 12. In many of the illustrations in this text we shall use a cell size of 16 bits, since this is the size appropriate to many of today's minicomputers, but we are seldom concerned with cell size.

A fact of importance is that our ideal computer stores integers in what is known as the "two's complement" form. In this form, the first bit is taken to be the sign, 0 for + and 1 for −, in fact, the bits arise from representing an integer k (positive or negative) by the rightmost n bits in ((2 to the power (n+1)) + k), where n is the cell size. To understand this it is easiest to suppose, for the moment, that the cell size is three, in which case the integers are as follows:

cell content	signed integer
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

Notice a curiosity in this system, viz., that the largest negative number is always one less than the negative of the largest positive number.

Another fact of importance is that our ideal computer can communicate with the outside world, i.e., it may read one character at a time and it may write one character at a time. This may appear too primitive a capability to be really useful, but it has conceptual simplicity and the power that we need, especially since newline and newpage are considered as characters. In fact, a rather versatile set of input and output routines uses only these facilities. A character is represented within a computer cell as an integer, e.g., in the ASCII[1] system, the character "B" is represented by #102, "7" by #67 and "+" by #53.

1.3.1. The memory

The memory consists of a sequence of consecutive storage cells. The number of storage cells in the memory is of no concern here. We may always assume, in what follows, that there will be enough. Of course, on any practical computer the memory

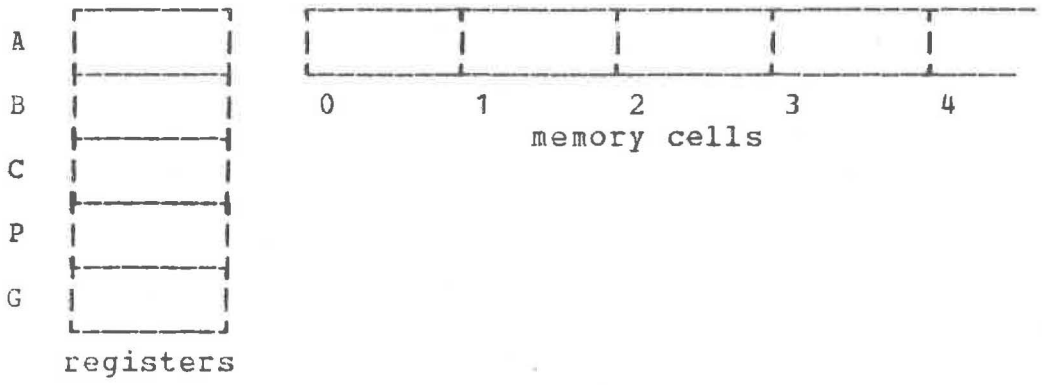
--- [1] ASCII stands for American Standard Code for Information Interchange.

is limited, but this limitation need not bother us now. We think of the cells as being numbered consecutively from zero upwards. A mental picture of the first few cells of memory may be something like the following.

The consecutive numbers assigned to the cells are known as their "addresses". Thus, the address of the first cell is 0, that of the second is 1 and so on.

1.3.2. The registers

Our ideal computer has five registers (or specialized storage cells). Two of these cells are accumulators, i.e., they participate in arithmetic and other operations. Another two are index registers, i.e., their content is always interpreted as an address. The fifth register is the instruction counter. This register contains the address of the next instruction. The two accumulators are known as the A- and the B-registers, the index registers are the P (program) and the G (global) registers and the fifth is the C (instruction counter) register. Our mental picture of the ideal computer with memory and registers is now something like the following.



1.3.3. The operations

An operation which the computer performs may copy information (bits) from a memory cell to a register, from a register to a memory cell or may manipulate the content of registers particularly the A- and B-registers. Each operation is performed as a result of the analysis of an "instruction". Each instruction is a sequence of bits which have been fetched from a memory cell. Our ideal computer has only eight basic operations. These are: load (0), store (1), add (2), jump (3), jump if true (4), jump if false (5), call a routine (6) and execute an accumulator operation (7). We shall examine each of these in detail when we look at the instructions. For the moment, it is enough to know that the store operation (1) copies the content of register A to

some memory cell. It is the only way of moving information into the memory. The load operation (0) first copies the content of register A to register B, then moves a new value to register A. This value may, or may not, come from memory. The execute instruction (7) may perform a specified operation (possibly arithmetic) on the content of the two accumulators, for example, if its operand is 8, it adds the content of the A- and the B-registers, leaving the result in the A-register.

1.3.4. The instructions

As mentioned above, an instruction is a sequence of bits, the content of some memory cell which has been fetched for analysis. These bits specify three things:

a) the operation,
b) the modifiers, and
c) the raw operand.

The operation is specified by three bits, the modifiers by another three and some of the remaining bits specify the operand. We may then picture an instruction as follows

where F is the operation, I, P and G are the modifiers and D is the raw operand.

Observe that the three F bits are sufficient to represent the numbers from 0 to 7 and therefore can specify eight different operations. The modifiers consist of three bits known as the I-flag, the P-flag and the G-flag. Their effect is to cause further calculation to be done on the operand.

If we use a sixteen bit cell and do not, for the moment, use the first bit, then some examples of instructions, in octal and in binary are:

instruction	octal	binary
load 1	#000001	0 000 000 000 000 001
store 3	#010003	0 001 000 000 000 011
add 2	#020002	0 010 000 000 000 010

in which none of the modifiers is on.

1.3.5. Operand modification

When an instruction is analyzed, its D part is extracted, as a non-negative integer, and is considered as the raw operand. This raw operand is subject to modification determined by the modifiers. As mentioned above, the modifier field of the instruction consists of three bits, the I-, P- and G-flags. Operand modification is done as follows:

a) if the P-flag is on, then the content of the P-register is added to the raw operand to give a modified operand;

b) if the G-flag is on, then the content of the G-register is added to the raw operand to give a modified operand.

It is convenient to know that in any one instruction the P- and G-flags cannot both be on. After steps a) and b) above, then

c) if the I-flag is on, then the operand (possibly modified as above), is considered as the address of a cell in memory, the content of which will be used as the modified operand.

Some examples might be in order here. Assuming a cell size of 16, the instruction #001002 (in binary 0 000 001 000 000 010) has its G-flag on; consequently its modified operand is 2 plus the content of the G-register. The instruction #002002 has its P-flag on; consequently its modified operand is 2 plus the content of the P-register. The instruction #006002 (in binary 0 000 110 000 000 010) has both the I-flag and the P-flag on. Its modified operand is therefore obtained by adding 2 to the content of the P-register and using this as the address of a memory cell from which the final modified operand is fetched. In each of these instructions the operation is load. The instruction

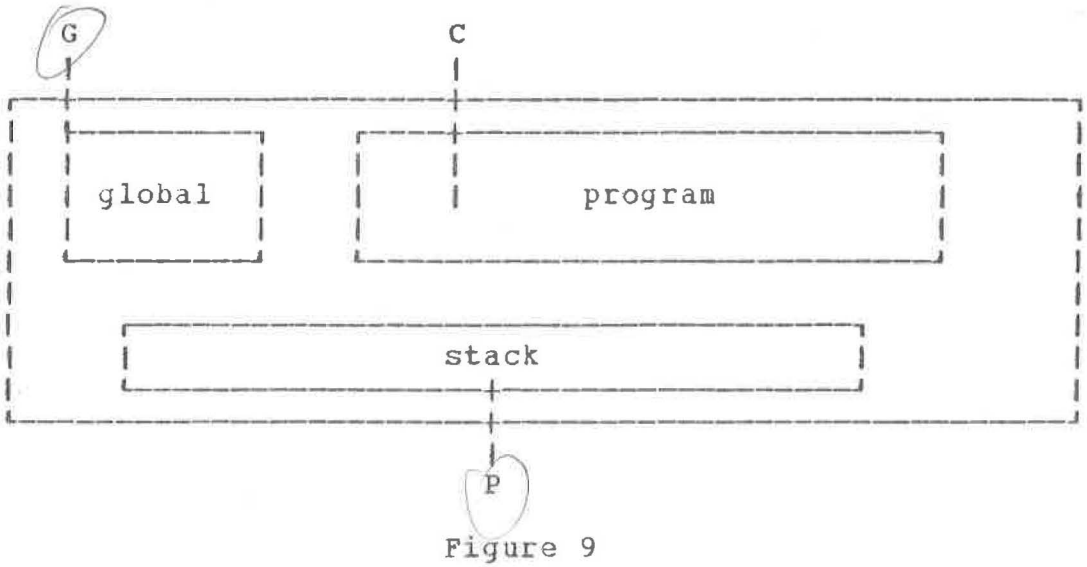


Figure 9

#006002 therefore loads the content of some memory cell into the A-register.

1.3.6. Memory layout and machine operation cycle

During the execution of a program the memory of the computer may be considered as having three independent areas: the program, containing the constants and the sequence of instructions to be executed; the global array, which allows for communication between program segments; and the stack, which has the property that it may grow and shrink during program execution. The three registers P, G and C are associated with these areas in the following way.

a) The P-register points to (i.e., contains the address of) some cell in the stack.

b) The G-register points to the first cell of the global array.

c) The C-register points to an instruction in the program.

A mental picture of the memory layout might therefore be as represented in figure 9.

The basic cycle of operation of the computer is as follows:
a) the instruction pointed to by the C-register is fetched and analyzed and the content of the C-register is incremented by one, and
b) the operation specified by the instruction is executed.

The basic cycle consists then of two parts -- fetch and execute. This cycle is repeated as often as is necessary, i.e., fetch, execute, fetch, execute, fetch, execute, and so on.

Chapter II

The assembler language

This chapter describes an assembler language which makes it easier to study the operation of the ideal computer more closely and allows us to construct some simple illustrative programs.

2.1. Mnemonics

The discussion of computer instructions and their illustration using octal or binary integers is tedious. It is not easy to remember, e.g., that the jump operation is #03xxxx (here x indicates an unspecified octal digit), or that a combination of the I- and P-flags is #0x6xxx. Instead, we use letters of the alphabet to represent these things.

The mnemonics for a computer instruction set can be a matter of choice. Those which we use for our ideal machine are as follows:
a) the representation of each instruction begins with one of the letters L, S, A, J, T, F, K and X,
b) then come the optional modifiers, first the letter I, then P or G, and
c) then follows the operand, which is either a decimal integer, a character between apostrophes or a label reference. A label reference is the letter L followed by a small non-negative decimal integer.

To make this more readable, we also allow some letters or special characters after an X, e.g., XR is equivalent to X4 and X+ is equivalent to X8.

A little piece of program using these mnemonics might be

```
$ 1:LL499 SP4 LIG60 K2 XR
499:C"HELLO"
```

where L499 is a label reference and XR is the operation "Return to caller" which is equivalent to X4.

Although we are not yet ready to follow what it does, it is

instructive to compare this with the instructions for a sixteen bit machine in octal which it represents

```
 003405, 012004, 005074, 060002, 070004, 002510, 042514, 046117
1:LL499     SP4   LIG60      K2      XR  499:"H    E  L    L O"
```

and observe that it is probably worthwhile learning these mnemonics before going any further, especially since mnemonics are more concise.

2.1.1. The assembler

A set of operation mnemonics, as described above, together with a few other aids is usually called an "assembler language". Very little programming is now done by writing instructions in binary, or even in octal, although this was the way it was done in the early days of computing. A better way is to use an assembler language, although, even this has now fallen into disfavour. The modern way to program is to use a high level language, i.e., a language more suitable for problem solving. Our eventual aim is to become familiar with the high level language BCPL(1), but first we need to understand a machine on which it runs and an assembler language for that machine.

Of course, the machine itself does not operate directly from the assembler language: for example, the machine does not know how to interpret LIG60 directly. But machines are very good at tedious clerical tasks, and the translation of a program from assembler language to its equivalent in binary is just such a task. This process is known as "assembly" and the program which does it is called the "assembler". Before long we shall be studying the details of just such an assembler. To do this, and to understand the details of machine instructions, we shall soon concentrate our attention upon an assembler language.

2.1.2. A loader

While the assembler is translating the mnemonics to machine instructions, it does not execute them. It may either store them somewhere in the memory for later execution, or, more frequently, it may write them out for later use. If it does the latter, then we need some way in which to load the machine instructions back into the memory. This job is done by a program known as the "loader". At some point we shall study how the loader works, but for the moment we need only know that it exists, that it knows how to load a set of instructions into the memory and how to transfer control to the first instruction of the program.

--- (1) The language BCPL was developed and implemented by Martin Richards [R2].

2.2 MINICODE(1)

The assembler language which we choose to use on our ideal machine is called MINICODE. First let us get an overall view of MINICODE by glancing at the following lines.

```
JL3
$ 1:LIP2 L0 X<= FL4 XR 4:LIP2 L1 X- SP8 LIP3 SP9 LIP5 SP10
 LIP4 SP11 LIL2 K6 LL499 SP8 LIP2 SP9 LIP3 SP10 LIP5 SP11
 LIG76 K6 LIP2 L1 X- SP8 LIP4 SP9 LIP3 SP10 LIP5 SP11 LIL2
 K6 XR 3:JL6
$ 5:7:LIG70 K2 SP2 LIP2 L0 X<= FL8 XF 8:LIP2 SP5 L'S' SP6
 L'I' SP7 L'D' SP8 LIL2 K3 JL7 XR 6:XF
 2:DL1 499:C"MOVE %N FROM %C TO %C*N"
G1L5
Z
```

We see in this that a sequence of instructions is represented rather compactly and in a form in which we might read it more easily than in octal (once we get used to it!).

The language MINICODE is made up from:

a) executable instructions,
b) storage reservation instructions,
c) labels,
d) pseudo-instructions (messages to the assembler), and
e) comments.

In the example above LIP2 is an executable instruction, DL1 is a storage reservation instruction, 499: is a label and Z is a pseudo-instruction. There are no comments in the example, since this MINICODE was produced by machine from a program in BCPL.

2.2.1. Executable Instructions

The precise form (syntax) of an executable instruction is as follows.

```
( L )
( S )
( A )          ( P )
( J )    I     ( G )    n
( T )   +-+    ( L )
( F )          +---+
( K )
( X )
```

Here, ( ) indicates a selection of alternatives, +-+ indicates that what stands above it is optional, i.e., may be left out, and n stands for a decimal integer (usually non-negative) or a character within apostrophes. After the letter X, the integer n may be replaced by an equivalent suggestive symbol (see below). Examples of executable instructions are LIG70 K2 SP2. We should take these and examine them together with the corresponding in-

--- (1) MINICODE is a derivation of the language INTCODE first described by Martin Richards [R3].

struction in octal and binary. Thus

```
LIG70          #005106          0 000 101 001 000 110
```

means that the operation is 'load', the I- and G-flags are on and the raw operand is 70 or #106. The instruction

```
K2             #060002          0 110 000 000 000 010
```

means that the operation is 'call' and the raw operand is 2. There are no flags on. The instruction

```
SP2            #012002          0 001 010 000 000 010
```

means that the operation is 'store' the P-flag is on and the raw operand is 2.

A more systematic description of the operations of the ideal computer is now in order. If some of the details are not clear, then we should not worry for the moment but should remember this list for later reference.

Here it is assumed that d is the value of the operand after index modification and indirect address calculation, if any, has been completed. When an instruction is fetched, then the C-register is first incremented so that it points to the next instruction.

0) Ld (Load d) copies the content of the A-register to the B-register and then loads the operand d into the A-register.
1) Sd (Store d) copies the content of the A-register to the cell at address d.
2) Ad (Add d) adds d to the content of the A-register leaving the sum in the A-register.
3) Jd (Jump to d) places d in the control register C.
4) Td (jump to d if True) places d in the control register C if the content of the A-register is -1.
5) Fd (jump to d if False) places d in the control register C if the content of the A-register is 0.
6) Kd (call a routine, with stack length d) places the content[1] of P at address (P)+d (stack link), places (C) at (P)+d+1 (return address), places (P)+d in P (new environment) and places the (A) in C (address of routine). A detailed explanation of the meaning of this instruction will be given later. The curious may be interested to know that the address of the routine to be called is already in the A-register.
7) Xd (eXecute the operation number d). The operation specified by d is executed using registers A and B. Usually the result is placed in A. For example, X8 (or X+) adds the content of B to the content of A, and X5 (or X*) multiplies the content of A by the content of B. It is important to notice that none of the execute instructions involves a

--- [1] We shall often abbreviate "content of P" to "(P)".

memory address directly.

Note that Ld and Sd differ in that Ld loads the operand itself, whereas Sd stores a value at the address d. Thus L treats d as a number and S treats d as an address. It may be useful, at this stage, to list all the standard execute instructions. They are:

instruction		meaning	in BCPL-like notation
X0		no operation	
X1	X!	dereference register A	A := !A
X2	XN	arithmetic negation	A := -A
X3	X¬	logical negation	A := ¬A
X4	XR	return to caller	C := P!1; P := P!0
X5	X*	multiplication	A := B * A
X6	X/	division	A := B / A
X7	X/*	remainder	A := B REM A
X8	X+	addition	A := B + A
X9	X-	subtraction	A := B - A
X10	X=	equality	A := B = A
X11	X¬=	inequality	A := B ¬= A
X12	X<	less than	A := B < A
X13	X>=	not less than	A := B >= A
X14	X>	greater than	A := B > A
X15	X<=	not greater than	A := B <= A
X16	X<<	shift left	A := B << A
X17	X>>	shift right	A := B >> A
X18	X/|	and	A := B /| A
X19	X|/	or	A := B |/ A
X20	X¬E	not equivalent	A := B NEQV A
X21	XE	equivalent	A := B EQV A
X22	XF	finish	FINISH
X23	X?	case (sequential search)	
X24		case (indexed jump)	
X25		read a character	A := RDCH()
X26		write a character	WRCH(A)

The operation X! takes the content of the A-register as an address of a cell and fetches the content of that cell into the A-register. The operation X¬ reverses every bit in the A-register, i.e., every 0 bit becomes a 1 bit and vice versa. The operations X/|, X|/, X¬E and XE are bitwise logical operations on the contents of the A- and B-registers which, for each pair of bits, delivers a corresponding bit in the A-register, according to the following table

A	B	/|	|/	¬E	E
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	1

An explanation of other operations will be given in the subsequent text.

2.2.2. The storage reservation instructions

It is important to remember that a storage reservation instruction is not executed. Its purpose is to help in storing data items (constants) needed by the program. There are storage reservation instructions for

a) cell storage,
b) character storage, and
c) global storage.

The first two reserve storage in the program section of memory and the third involves storage in the global array. An example of a cell storage instruction is D39, which sets aside the next cell with the value 39 in it, or DL499, which sets aside the next cell with the value (address) of label number 499 in it. Since characters may possibly be stored more than one per cell, there is a special storage instruction for them. An example is C65, where 65 is the ASCII value of the character A. It is also possible to use the characters themselves, within quotes, e.g., C"HELLO" is equivalent, in ASCII, to the sequence of instructions C5 C72 C69 C76 C76 C79 , the first instruction, C5, gives the length of the character string.

2.2.3. Labels

A label is of the form n: , where n is some small positive decimal integer, e.g., 5: . It is important to note that the number which appears in the label is not necessarily the actual address of the cell containing the instruction which it precedes. For example, 5: does <u>not</u> mean that the next instruction is stored in cell number 5. The relation between the label number and the actual cell at which its instruction is stored is usually a well-kept secret, known only to the assembler and the loader.

2.2.4. Pseudo-instruction

There are two pseudo-instructions in the MINICODE assembler language. One is the letter Z which indicates the end of a section. A section is a piece of code across which labels may be referenced. This means that if a label appears in one section of code, then it may not be referenced from another section. In the following example

```
4:LIP2 SP5 LIG143 K3 JL4 Z
JL4 Z
```

the second occurrence of "JL4" will reference label 4 which does not exist in its section. An error will therefore result. All communication between sections of a program is done via the addresses in the global array.

The other pseudo-instruction is a dollar symbol. It marks the entry to a routine or a function, and is mostly used to aid readability.

2.2.5. Comments

Comments are remarks which might be useful to the human reader in understanding the code. In MINICODE, a comment consists of a solidus, "/", together with all characters to its right and up to the end of the line. An example of a line of MINICODE with a comment is:

```
LIP3 SP7 LIG14 K5 / This writes one character
```

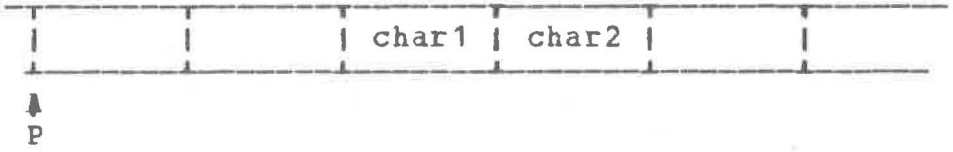
Note that there are four occurrences of a solidus which do not begin a comment. They are in X/, X/*, X/| and X|/, which are equivalent to X6, X7, X18 and X19 respectively.

2.3. A small example

We are now perhaps ready to study a small program written in MINICODE. Do not be disturbed by the fact that what it does is trivial. We need to start with the simplest of things so that the basic principles can be well established. The program will read two characters and then print one of them, the larger. The program is as follows.

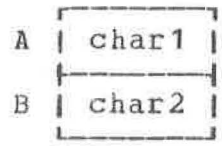
```
 X25 SP2 X25 SP3        / read two characters
 LIP2 X> TL4            / compare them
 LIP2 JL5               / select the first
4: LIP3                 / select the second
5: X26 XF               / print a character
 Z
```

We read it in the following way. The instruction X25 reads one character from the outside world and leaves it, as a binary number, in the A-register. For example, if the character read were 'B', then the number in the A-register would be 66 or #102 (in binary 0 000 000 001 000 010). The instruction SP2 stores the content of the A-register in the stack at the cell pointed to by the content of the P-register plus 2 (1). The instruction X25 reads one more character from the outside world into the A-register and SP3 stores it in the next consecutive cell, P3. A picture of the stack at this stage is



The next instruction, LIP2, loads the content of stack position P2 into the A-register after pushing the previous content of the A-register into the B-register. The two registers now have the content as shown below.

--- (1) We shall often shorten "the cell pointed to by the content of the P register plus n" to "Pn".



The instruction X> compares the content of the two registers (as integers), i.e., it determines whether the relation "char2 > char1" is true. If it is true it puts -1 (true) in the A-register, otherwise it puts 0 (false) in the A-register.

The instruction TL4 performs a jump to the label 4 if the content of the A-register is true (-1). The instruction LIP2 first loads the content of stack position P2 (the first character) into the A-register and JL5 jumps to the label 5. At label 4 the content of stack position P3 (the second character) is loaded into the A-register by the instruction LIP3. At label 5, the content of the A-register is written to the outside world, by x26, as a character. The program finishes with the instruction XF.

Having seen the details of this program, it might now be instructive to examine the MINICODE for the same program generated automatically from the high level language BCPL. The instructions in that language are

```
GLOBAL $( START:1; RDCH:13; WRCH:14 $)
LET START() BE
   $( LET A = RDCH()
      LET B = RDCH()
      WRCH(A > B -> A, B) $)
```

The strange numbers, 13 and 14, are there because the BCPL system happens to store the entry address to the read-a-character routine, RDCH, in global cell number 13. Similarly the entry to WRCH is stored in global cell 14, and the entry to the START routine is always in global cell 1. The expression A > B -> A, B , means "if A is greater than B, then the value is that of A; otherwise, the value is that of B".

The MINICODE generated automatically by a compiler is as follows.

```
$ 1:LIG13 K2 SP2 LIG13 K3 SP3 LIP2 LIP3 X> FL4
 LIP2 SP6 JL3 4:LIP3 SP6
3:LIG14 K4 XR
G1L1
Z
```

The dollar symbol marks the entry to the START routine. The instruction G1L1 ensures that the entry address to the START routine is loaded into global cell 1. Upon comparing this code with that developed above, it will be clear that code produced by a compiler is usually not as concise as that which might be produced by hand, since the compiler must try to treat many dif-

ferent things in a uniform manner. At the present time we shall not analyze further either the program in BCPL or the code produced automatically, since our immediate aim is the understanding of MINICODE.

2.3.1. Another example

The next example program prints the content, n, of some memory cell as a non-negative octal number, in a given number, d, of print positions. For example, the 16 bit binary integer

0 101 010 110 011 111

would be printed in 6 print positions as

052637 .

Observe how easy it is for a human to do this translation! It is only necessary to split the binary integer into groups of three bits each, starting from the right. Each group then is represented by some octal digit. For example, the binary number 01011110 should be viewed as 01 011 110 and then printed as 136.

An interesting solution arises by arguing as follows. If the value of d, the number of print positions, is one, then we can print only one digit, which is the octal digit representing the rightmost three bits. If the value of d is greater than one, then detach the rightmost three binary digits, print the left hand part in d-1 print positions and then print the digit representing the rightmost three bits. Since there is a subtlety here, we shall say this again in terms of the example given above. Thus, to print the binary integer

0 101 010 110 011 111

in 6 print positions, all we need do is to print the binary integer

0 101 010 110 011

in 5 print positions, i.e., 05263, and then to print the octal digit representing the binary integer 111, i.e., 7.

What we have shown is that the problem of printing a non-negative integer in octal with d print positions can be reduced to that of printing an integer, in octal, in d-1 print positions, and therefore, eventually to that of printing an integer, in octal, in one print position.

The principle involved in this solution is known as recursion, and the program is expressed in BCPL as follows:

```
GLOBAL $( WRCH:14 $)
LET WRITEOCT(N, D) BE
  $( IF D > 1 THEN WRITEOCT(N>>3, D-1)
     WRCH(N /| 7 + '0') $)
```

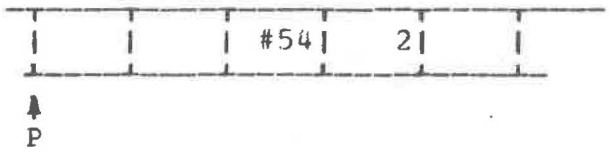
We do not intend to make a systematic study of BCPL yet. That will come later. For the moment the display, in BCPL, of some programs under discussion, may be helpful, and occasional remarks will be made to aid the understanding of them. Since, in this chapter, every BCPL program will be accompanied by its translation to MINICODE, the meaning of any construct can be determined from the translation. In the above, the expression N >> 3 yields the value of N shifted right by 3 bits, the vacated positions on the left being filled with zeros, and N /| 7 yields the rightmost three bits of N, all the other bits being set to 0. In MINICODE, for example, if the content of the B-register is #0325 (in binary 0 000 000 011 010 101), and that of the A-register is 7, then after the instruction X>>, the content of the A-register will be 0 000 000 000 000 001 in binary. With the same initial conditions, the content of the A-register, in binary, after the instruction X/|, will be 0 000 000 000 000 101.

But we should write this program in MINICODE. Assuming that n is in P2 and d is in P3, this might be:

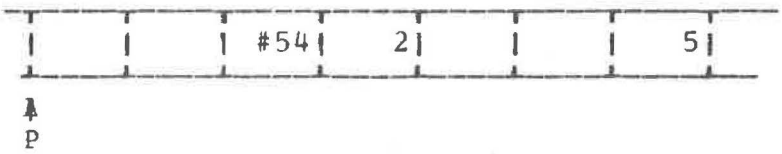
```
3:LIP3 L1 X> FL5        / is d > 1?
  LIP2 L3 X>> SP6       / store n>>3 in P6
  LIP3 L1 X- SP7        / store d-1 in P7
  LL3 K4                / call the routine starting at label 3
5:L7 LIP2 X/| A'0'      / n /| 7 + '0' in register A
  X26 XR                / write and return                    .
```

2.3.2. The stack

In order to follow what is happening here, we should examine the content of the stack at each stage. Initially, supposing n=#54 and d=2, we have

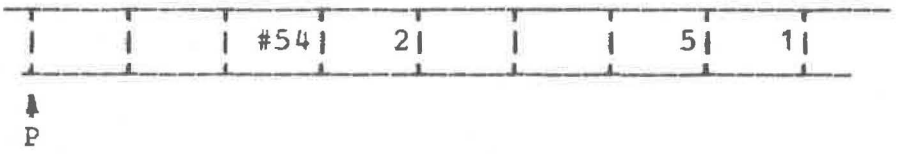


After the instructions LIP3 L1 X>, and because 2>1 is true, we have -1 (true) in the A-register, and the stack is unchanged. The instruction FL5 therefore does not result in a jump. After the instructions LIP2 L3 X>> SP6, the stack is

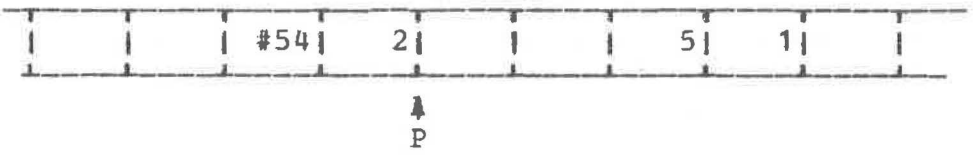


since X>> shifts the content of the B-register right by a, where a is the content of the A-register and with the result deposited in the A-register. After the instructions LIP3 L1 X- SP7, the

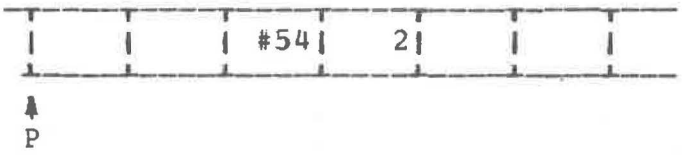
stack is



The instruction LL3 puts the address of label 3 in the A-register ready for a call, and the instruction K4 executes that call. This has the effect of changing the P-register so that it points four cells beyond where it pointed before. The stack is now



The next instructions to be executed are at the label 3, viz., LIP3 L1 X> . This will result in a 0 (false) in the A-register, since 1>1 is false, and FL5 causes the instruction counter to be changed so that control jumps to label 5. The next instructions are L7 LIP2 X/| A'0', which calculate the ASCII number representing the rightmost three binary digits of the value in P2. In more detail, after L7 and LIP2, the A- and B-registers contain 5 and 7 respectively. In binary these are 101 and 111. The operation X/| now takes the logical-and of these two binary numbers, which is 101 and deposits this in the A-register (redundant in this case). The instruction A'0' will convert this to the internal coding of a digit as a character, since we know that the decimal digits are encoded consecutively. The instruction X26 prints the character 5. The instruction XR then returns to the caller. It moves the stack pointer back and execution resumes at the point after the last call, i.e., at label 5. The stack now appears as



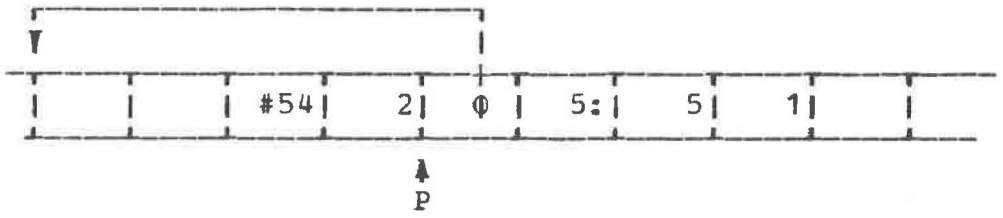
and the instructions 5:L7 LIP2 X/| A'0' calculate, in the A-register, tne ASCII number corresponding to the digit 4. The instructions X26 XR now print the character 4 and return to the caller. This shows, for example, how the content of some cell will be printed in two print positions.

2.3.3. The stack linkage

In the example just given we discover that a routine is called. The effect of a call is to "move" the stack pointer P forward by n cells, were the call is Kn. The return "moves" the stack pointer back to where it was before the call. The mechanism used involves the first two stack cells, P0 and P1, to help

in this process. Cell P0 is used to store the previous content of P, and P1 is used to store the return address, i.e., the address of the next instruction after the call.

A more complete picture of the stack, after the call, K4, in the above example, is



where "5:" indicates that the address of label 5 has been stored in P1.

We may see now how the intruction XR (return to caller) can work. Our machine accomplishes this by copying the content of stack cell P1 to the C-register and by copying the content of the stack cell P0 to the P-register. Thus the return-to-caller is accomplished easily.

The mechanism of a call is a little more complicated. It must first set up two cells ready for the return. The steps, with the call Kn, are as follows:

a) copy the content of P to Pn, this sets up the backward stack link,
b) copy the content of C to P(n+1), this stores the return address,
c) add n to P, this moves the stack pointer forward n cells, and
d) copy the content of A to C, this loads the address of the entry to the called routine into the C-register.

Since at the time of a call, register A contains the address of the routine to be called, this means that the next instruction will be taken from the entry to that routine.

2.3.4. A variation

It is easy to divide a non-negative integer by eight using a shift instruction. One must shift right by 3, since each right shift by one bit divides the content by 2. On most computers a shift instruction is faster than the divide instruction. This is why we used it in the example above. If we were to print a non-negative number in decimal in a given number of print positions, then the routine needs to be altered a little. In BCPL it would be

```
GLOBAL $( WRCH:14 $)
LET WRD(N, D) BE
  $( IF D > 1 THEN WRD(N / 10, D - 1)
     WRD(N REM 10 + '0') $)
```

In this, the shifting is replaced by division, i.e., we have

N / 10 instead of N >> 3. Also N REM 10 delivers the remainder after division of N by 10, and replaces N /| 7, which for octal is usually faster. The code produced by the compiler is:

```
$ 1:                                  / LET WRD(N, D) BE
 LIP3 L1 X> FL4                       / $( IF D > 1 THEN
 LIP2 L10 X/SP6                      /     N/10
 LIP3 L1 X- SP7                       /          D-1
 LIL2 K4                              / WRD(...,...)
4:LIP2 L10 X/* A'0' SP6 LIG14 K4      / WRCH(N REM 10 + '0')
 XR                                   /  $)
2:DL1                                                          .
```

Compare this code with the hand coded version in section 16 above. Apart from replacing X>> by X/ and X/| by X/*, the compiler-produced code treats WRCH as a routine like any other routine, and it is called in a standard way, viz., SP6 LIG14 K4. The 14-th global cell contains the address of the routine WRCH. Note that the 2:DL1 is the compiler's way of remembering the entry to the routine WRD. It uses this on the line LIL2 K4 when the routine is called.

2.3.5. Printing in decimal

In general, the routine defined in the above section is not too useful because a) it does not work when the number is negative and b) it does not suppress leading zeros, as is the usual custom. Instead, we shall now consider a routine which will print a number, in decimal, in the minimum number of print positions. For example, the number 237 should take three print positions and the number -7 should take two. A solution, if the number is non-negative, might be

```
GLOBAL $( WRCH:14 $)
LET WRPD(N) BE
   $( IF N > 10 THEN WRPD(N/10)
      WRCH(N REM 10 + '0') $)                                  .
```

The difference between this and the preceding routine is that it stops calling itself when N <= 10 rather than when D = 1. Its translation to MINICODE is

```
$ 1:                                  / LET WRPD(N) BE
 LIP2 L10 X> FL4                      / $( IF N > 10 THEN
 LIP2 L10 X/ SP5 LIL2 K3              /     WRPD(N / 10)
4:LIP2 L10 X/* A'0' SP5               /     N REM 10 + '0'
 LIG14 K3                             /     WRCH(...)
 XR                                   /     $)
2:DL1                                                          .
```

Again, this does not work if the number is negative, so we might consider the following additional routine

```
GLOBAL $( WRCH:14 $)
LET WRD(N) BE
  TEST N < 0 THEN
```

```
        $( WRCH('-'); WRPD(-N) $)
      OR WRPD(N)
```

Here the command TEST b THEN c1 OR c2 tests the condition b; if it is true then the command c1 is executed; otherwise, the command c2 is executed. The translation to MINICODE is

```
$ 4:                              / LET WRD(N) BE
 LIP2 L0 X< FL7                   / TEST N < 0 THEN
 L'-' SP5 LIG14 K3                /   $( WRCH('-')
 LIP2 XN SP5 LIL2 K3              /      WRPD(-N) $)
 JL8
7:LIP2 SP5 LIL2 K3                / OR WRPD(N)
8:XR
2:DL1 5:DL4
```

The reader should now experiment with this to see whether it really works. He will find, in fact, that it will work correctly on all numbers except one. The one on which it does not work is the largest negative integer, i.e., the integer -(2 to the power(n-1)), where n is the cell size. The problem here is that the negative of such a number is too large for a memory cell, in two's complement form, so the results will be unpredictable. But there is a way out. We may program it instead as follows

```
GLOBAL $( WRCH:14 $)

LET WRD(N) BE
  TEST N < 0 THEN
    $( WRCH('-'); WRPD(N) $)
  OR WRPD(-N)

AND WRPD(N) BE
  $( IF N < -10 THEN WRPD(N/10)
     WRCH(-(N REM 10) + '0') $)
```

This solution depends upon feeding only negative numbers to the routine WRPD, and in changing that routine to manipulate only negative numbers. It also depends upon the knowledge that the expression

A REM B

is always equal to

A - ((A / B) * B)

for all values of A and B, whether positive or negative. For example, -7 / 5 is -1 and -7 REM 5 is -2. This fact may disturb some mathematicians interested in number theory, but it turns out to be convenient here.

2.3.6. More printing of decimals

Often our requirement is not to print an integer in the minimum of print positions, but to print it in a given number of print positions, but with leading zeros replaced by spaces and with the negative sign, if appropriate, appearing in front of

the first significant digit. This is the most helpful way if integers are to be printed in columns. A solution to this along the lines given above is not easy. Instead, we display now the routine from the BCPL library.

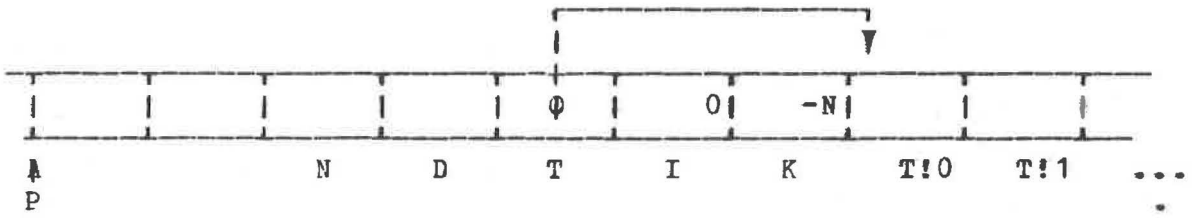
```
LET WRITED(N, D) BE
$(1 LET T = VEC 20 // create an array of 21 contiguous cells
    AND I, K = 0, -N
    IF N<0 THEN D, K := D-1, N
    T!I, K, I := -(K REM 10), K/10, I+1  REPEATUNTIL K=0
    FOR J = I+1 TO D DO WRCH(' ')
    IF N<0 DO WRCH('-')
    FOR J = I-1 TO 0 BY -1 DO WRCH(T!J+'0') $)1
```

Its translation to MINICODE is

```
$ 8:                                    / LET WRITED(N, D) BE
 LP7 SP4                                / $(1 LET T = VEC 20
 L0 SP5 LIP2 XN SP6                     / AND I, K = 0, -N
 LIP2 L0 X< FL60                        / IF N < 0 THEN
 LIP3 L1 X- SP3 LIP2 SP6                / D, K := D-1, N
60:61:LIP6 L10 X/* XN SP28              / -(K REM 10)
 LIP5 AIP4 SP29 LIP28 SIP29             / T!I := ...
 LIP6 L10 X/ SP6                        / K := K / 10
 L1 AIP5 SP5                            / I := I + 1
 L0 LIP6 X= FL61                        / REPEATUNTIL K = 0
 L1 AIP5 SP28                           / FOR J = I + 1
 LIP3 SP29 JL62                         / TO D
 63:L' ' SP32 LIG14 K30                 / WRCH(' ')
 LIP28 A1 SP28                          / // increment I
62:LIP28 LIP29 X<= TL63                 /      ... DO
 LIP2 L0 X< FL64                        / IF N < 0 THEN
 L'-' SP30 LIG14 K28                    / WRCH('-')
64:LIP5 L1 X- SP28 JL65                 / FOR J = I-1
66:LIP28 AIP4 X! A'0' SP31 LIG14 K29    / WRCH(T!J+'0')
 LIP28 AIL499 SP28                      / BY -1
65:LIP28 L0 X>= TL66                    / TO 0 DO
 XR                                     /   $)1
499:D-1
```

2.3.7 Arrays

It is worth taking a close look at this routine, for it introduces us to the concept of an array, i.e., a set of contiguous storage cells reached by the same name. The declaration LET T = VEC 20 reserves 22 cells on the stack. Note that its translation is LP7 (not LIP7) SP4, which places the address of P7 into the cell P4. The 21 cells from P7 to P27 may now be used as an array and accessed as T!0 to T!20. After the declaration AND I, K = 0, -N , two more cells P5 and P6 are reserved with values placed in them. The stack now looks as follows



The next point of importance is the manner in which the elements of the array are accessed. If the value of I satisfies $0 \le I \le 20$, then the value of the expression T+I is the address of the I-th element of the array T. Notice that this is computed by the instructions

```
LIP5 AIP4 SP29 ,
```

where I is in P5 and T is in P4. The value of T+I is then stored in P29, ready to be used as an address.

The routine WRITED works by dividing the given integer successively by 10 until the quotient is zero, storing the successive remainders in the array T and counting how often this is done. When this is complete, then the j-th element of the array T holds the j-th decimal digit, from the right, of the integer to be printed, and I contains the number of its significant digits. If the integer is non-negative, then we must first print D-I spaces and then the significant decimal digits in the reverse order from that in which they were found. If the integer is negative, then we must also allow space for the minus sign.

Observe that the basic calculation of the digits to be printed is made in the command

```
T!I, K, I := -(K REM 10), K/10, I+1 REPEATUNTIL K=0
```

Here T!I should be read as "T subscript I", indicating that we want the I-th element of the array T. Its address is the value of I + T. Note also that the significant digits of the given integer are printed by the command

```
FOR J = I-1 TO 0 BY -1 DO WRCH(T!I+'0')
```

in which the value of D is not used! The allowance for the negative sign is accomplished by the command

```
IF N < 0 THEN D, K := D-1, -N .
```

A rather good feature of the routine WRITED is that, if the value of D is too small, then the complete integer is nevertheless printed, with no information lost. Thus the effect of the call WRITE(N, 0) is equivalent to the call WRD(N) as discused in the preceding section.

2.3.8. Printing in hexadecimal

It is sometimes convenient to be able to print numbers in hexadecimal (base 16). The usual convention is to allow the first six letters of the alphabet, A to F, to represent the digit values ten to fifteen. Thus a hexadecimal integer FF is 255 in decimal and hexadecimal 100 is 256 in decimal.

A routine for writing a non-negative integer in hexadecimal, in a given number of print positions is

```
GLOBAL $( WRCH:14; WRITEHEX:75 $)
LET WRITEHEX(N, D) BE
    $( IF D>1 DO WRITEHEX(N>>4, D-1)
       WRCH((N/|15)!TABLE
            '0','1','2','3','4','5','6','7',
            '8','9','A','B','C','D','E','F' )  $)     .
```

This routine displays an interesting new feature. The value of

```
TABLE 3, 1, 2                                          ,
```

for example, is the address of a set of contiguous cells containing the constants which follow TABLE. It is thus an array and can be treated as such. The expression N/|15, which yields the rightmost four bits of N, is then used to subscript the table in WRITEHEX. The translation to MINICODE is

```
$ 14:                                  / LET WRITEHEX(N, D) BE
 LIP3 L1 X> FL82                       / $( IF D > 1 THEN
 LIP2 L4 X>> SP6                       /          N>>4
 LIP3 L1 X- SP7                        /                D-1
 LIG75 K4                              /  WRITEHEX(...,...)
82:L15 LIP2 X/| AL83 X! SP6            /   (N/|15) ! TABLE ...
 LIG14 K4 XR                           /   WRCH(...) $)
83:D'0' D'1' D'2' D'3' D'4' D'5' D'6' D'7'
 D'8' D'9' D'A' D'B' D'C' D'D' D'E' D'F'
 G75L14
```

The reader should consider now the possibility of generalizing the routines given here to print numbers in any base.

2.4. Character strings

A character is represented within a computer cell by a non-negative integer. In the ASCII system this integer lies in the range 0 to 127 and in the EBCDIC(1) system in the range 0 to 255. Thus, the ASCII system needs only seven bits and the EBCDIC system requires eight. If our ideal computer allows eight bit characters, then it may accommodate both systems, so this is what we shall do. Moreover, since a common cell size is

--- (1) EBCDIC stands for Extended Binary-Coded-Decimal Interchange Code.

sixteen bits, the choice of eight bits per character means that we may conveniently store two characters per cell. If we were to store strings of characters, one per cell, then this would be wasteful of space, although occasionally it is useful to do this. The standard way for storing strings in the BCPL language is to use an array of cells. There is thus one cell containing an address of a set of contiguous cells. Into these cells is placed, in successive eight bit "bytes", first the number of characters in the string, followed by the integers which represent the characters. For example, the string "HELLO" would be stored as follows.

Observe that this limits the number of characters in a string to 255 (or 2^8-1) at most, but this is not a severe limitation. On a computer with a cell size of 32 bits one may store four characters per cell.

The routine for printing a string in the BCPL library is

```
GLOBAL $( WRCH:13; WRITES:60; GETBYTE:85 $)
LET WRITES(S) BE
  FOR I = 1 TO GETBYTE(S, 0) DO WRCH(GETBYTE(S, I))     ,
```

which depends upon another more primitive routine GETBYTE. The call GETBYTE(S, I) yields the I-th byte of the string S. For example the value of GETBYTE("HELLO", 0) is 5 (the length of the string) and the value of GETBYTE("HELLO", 2) is 'E'. The translation of WRITES into MINICODE is

```
$ 4:                                  / LET WRITES(S) BE
 L1 SP3 LIP2 SP6 L0 SP7 LIG85 K4      / FOR I=1 ... GETBYTE(S,0)
 SP4 JL32                             / // upper limit in P4
33:LIP2 SP9 LIP3 SP10 LIG85 K7        / GETBYTE(S, I)
 SP7 LIG14 K5                         / WRCH(...)
 LIP3 A1 SP3                          / // increment I
32:LIP3 LIP4 X<= TL33 XR              /  TO ... DO              .
```

It is also interesting to observe that the translation of

```
LET START() BE WRITES("HELLO")
```

is as follows

```
$ 1:LL499 SP4 LIG60 K2 XR
499:C"HELLO"                                                    .
```

We have seen this before, but now we may understand that the instruction LL499 loads the address of the string into the A-register, SP4 stores that address at P4, LIG60 loads the address of the WRITES routine into the A-register and K4 calls the rou-

tine WRITES.

If we assume that CH.P.C is the number of characters per cell, CELL.SIZE is the number of bits per cell, CHAR.SIZE is the number of bits per character, and CHAR.MASK is (2 raised to the power CHAR.SIZE)-1, then a completely general version of GETBYTE for any machine might be

```
GLOBAL $( GETBYTE:85 $)
LET GETBYTE(S, I) =
  (S!(I/CH.P.C) >>
     ((CH.P.C-1) - I REM CH.P.C) * CHAR.SIZE) /| CHAR.MASK .
```

If the cell size is 16 and the CHAR.SIZE is 8, then this may be written, for faster execution, in the form

```
GLOBAL $( GETBYTE:85 $)
LET GETBYTE(S, I) =
  (S!(I >> 1) >> ((¬I /| 1) << 2) ) /| 255        .
```

It would be a useful exercise for the reader to determine that this does indeed deliver what is expected. Its translation to MINICODE is

```
$ 1:                                  / LET GETBYTE(S, I) =
 LIP3 L1 X>> AIP2 X! SP4              /  S ! (I >> 1)
 LIP3 X¬ L1 X/| L2 X<< SP5            /  ((¬I /| 1) << 2)
 LIP4 LIP5 X>> L255 X/|               /  (... >> ...) /| 255
 XR
 G85L1                                                        .
```

Actually, some machines might be able to treat this function as a basic machine instruction, in which case further efficiency in execution is to be expected. There is an associated routine PUTBYTE which has the opposite effect from GETBYTE. Thus the call PUTBYTE(S, I, C) will store the character C in the I-th byte of the string S. For the details of PUTBYTE the reader should consult the source listing of the BCPL library.

2.4.1. The remaining library routines

This is almost the place for us to abandon our discussion of MINICODE, in favour of BCPL instead and the reader might wish to heave a sigh of relief! Up to the present our intention was to understand the machine and its assembler language. Our knowledge of BCPL has been derived by some kind of "osmosis". We shall soon need to be more systematic. But before we make this change, we shall look at two more of the BCPL library routines as they appear in MINICODE, for these are an excellent source of well coded examples to use for illustration. The two are: that function which reads a decimal (base 10) integer, READN, and the formatted output routine WRITEF.

A simple minded-function for reading a decimal (base 10) integer might be

```
GLOBAL $( RDCH:13; CH:71 $)
LET RDN() = VALOF
  $(1 LET N, B = 0, FALSE
      IF CH = '-' THEN
        $( B := TRUE; CH := RDCH() $)
      WHILE '0' <= CH <= '9' DO
        $( N := N * 10 + CH - '0'; CH := RDCH() $)
      RESULTIS B -> -N, N $)1
```

In BCPL, the construction VALOF c, where c is a command, becomes an expression which yields a result. The command c must somewhere contain the command RESULTIS e, where e is an expression. Note that the function RDN expects that CH already contains the first character of the integer to be read. It depends upon the primitive function RDCH whose call, RDCH(), yields the next character from the input stream. The expression

b -> e1, e2

in BCPL, is interpreted as follows: if the value of b is true, then the value of the expression is that of e1; otherwise, it is that of e2. Observe carefully the code generated by B -> -N, N , in tne translation of RDN to MINICODE.

```
$ 1:                                / LET RDN() = VALOF
 L0 SP2 L0 SP3                      / $( LET N, B = 0, FALSE
 L'-' LIG71 X= FL5                  / IF CH = '-' THEN
 LIL499 SP3 LIG13 K4 SG71           / $( B := TRUE; CH := RDCH() $)
5:JL7                               / WHILE ...
6:L10 LIP2 X* AIG71 L'0' X- SP2     / $( N := N * 10 + CH - '0'
 LIG13 K4 SG71                      / CH := RDCH() $)
7:L'0' LIG71 X<= FL8                / '0' <= CH
 LIG71 L'9' X<= TL6                 / ... <= '9' DO
8:LIP3 FL10 LIP2 XN SP4 JL9         /          B -> -N,
10:LIP2 SP4                         /                N
9:LIP4 JL4 4:XR                     /  RESULTIS ...
2:DL1 499:D-1                       /             TRUE
```

The reader is expected to work through this MINICODE with a trivial example to determine how it works. Suppose for example that the variable CH contains the character '2' and that the next two characters on the input stream are '1' followed by a space. The result delivered must be the value of the decimal integer 21.

Although the function given above may be easy to follow, it suffers from the disadvantage already mentioned: one must assume that the first character of the integer to be read has already been swallowed by RDCH. The function from the BCPL library, READN, which is reproduced below, overcomes this and other disadvantages. It is:

```
GLOBAL $( RDCH:13; TERMINATOR:71 $)

LET READN() = VALOF
```

```
$(1 LET SUM, NEG = 0, FALSE
    $(L TERMINATOR := RDCH()
        SWITCHON TERMINATOR INTO
        $( CASE ' ': CASE '*T': CASE '*N': LOOP
           CASE '-': NEG := TRUE
           CASE '+': TERMINATOR := RDCH() $)
        BREAK                              $)L REPEAT
    WHILE '0'<=TERMINATOR<='9' DO
        $( SUM := 10*SUM + TERMINATOR - '0'
           TERMINATOR := RDCH()  $)
    RESULTIS NEG -> -SUM, SUM $)1                          .
```

This function is rather similar to RDN as defined above, but it also allows for the possibility of reading over blanks, new lines ('*N') and tabs ('*T') before reaching the integer. Also, it allows that the integer may be preceded by a plus sign ('+'). Another feature to notice is that when this function has yielded the integer, then the value of the variable TERMINATOR is the next character from the input stream beyond that integer. The translation to MINICODE is:

```
$ 42:                          / LET READN() = VALOF
 L0 SP2 L0 SP3                 / $(1 LET SUM, NEG = 0, FALSE
131:LIG13 K4 SG71              / $(L TERMINATOR := RDCH()
 JL132                         / SWITCHON ...
134:135:136:                   / $( CASE ' ': CASE '*T': CASE '*N':
 JL131                         /    LOOP
137:LIL443 SP3                 / CASE '-': NEG := TRUE
138:LIG13 K4 SG71 JL133        / CASE '+': TERMINATOR := RDCH() $)
132:LIG71 X? D5 DL133          /  ... TERMINATOR INTO
 D' ' DL134 D'*T' DL135 D'*N' DL136 D'-' DL137 D'+' DL138
133:JL139 JL131                /  BREAK $)L REPEAT
139:JL141                      /  WHILE ...
140:LIP2 L10 X* AIG71 L'0' X-  / 10 * SUM + TERMINATOR - '0'
 SP2                           /  $( SUM := ...
 LIG13 K4 SG71                 /  TERMINATOR := RDCH() $)
141:L'0' LIG71 X<= FL142       / '0' <= TERMINATOR
 LIG71 L'9' X<= TL140          / ... <= '9' DO
142:LIP3 FL144                 / RESULTIS NEG ->
 LIP2 XN SP4 JL143             /     -SUM,
144:LIP2 SP4                   /      SUM
143:LIP4 JL130                 /    $)1
130:XR
443:D-1                        / TRUE                      .
```

This example displays many interesting features. One of these is the switchon command, with the form

```
SWITCHON v INTO
$( CASE v1: s1
   CASE v2: s2
   ...
   CASE vn: Sn
   DEFAULT: Sd $)                                          .
```

This command allows for the choice of several different actions

depending upon the value of a variable. For example, if the value of v is v2, then control is transferred to command s2. Note how this is translated to MINICODE

X? DLd Dv1 DL1 Dv2 DL2 ... Dvn DLn

where the value in the A-register is to be compared successively with the n values v1, v2, ..., vn. If it is equal to one of them, say vi, then control is transferred to the instruction at the label Li; otherwise, control is transferred to the instruction at the label Ld (the default label), if that label is present, and otherwise to the command that follows the switchon command.

Another interesting feature is the REPEAT, which modifies the command preceding it by making it into a loop, i.e., a command which is executed repeatedly.

An associated command is LOOP, which transfers control to the end of the loop ready for another repetition. Another associated command is BREAK. This command transfers control to the command that follows the loop.

A better perspective on REPEAT, LOOP and BREAK might be got by knowing that the framework

```
$(    ...
      ...          LOOP
      ...
BREAK
      ...          $) REPEAT
```

is equivalent, in this instance, to the following chaotic mess of jumps

```
L1:   ...
      ...          GOTO L2
      ...
GOTO L3
      ...          L2: GOTO L1
L3:
```

2.4.2. Formatted write

We consider now the formatted output routine WRITEF. Before giving the BCPL source, a few remarks about what it is supposed to do, are in order. The call WRITEF("HELLO") will produce exactly the same output as the similar call WRITES("HELLO"), but it is usually not used for writing strings only.. Another call, WRITEF("VALUE = %N", V), will write

VALUE = 234

supposing the value of V is 234. Thus WRITEF acts like WRITES

on its first parameter except when that string contains a percent symbol. If a percent symbol is present, then following that symbol may be one of two layout symbols. The layout symbols and their purpose are summarized below, where it is assumed that the value of the next unused parameter is v.

symbol	action
C	WRCH(v)
S	WRITES(v)
N	WRITED(v, 0)
Iw	WRITED(v, w)
Ow	WRITEOCT(v, w)
Xw	WRITEHEX(v, w)

Here it is understood that the character used for the layout symbol, w, is one hexadecimal (base 16) digit, i.e., 0, 1, 2, ..., 9, A, B, ..., F. If the character following the percent symbol is not a layout character, then the percent symbol is ignored and the character following it is written in the normal way. This allows the percent symbol itself to be written using "%%". The WRITEF routine in BCPL is:

```
GLOBAL $( WRCH:14; WRITES:60; WRITED:68; WRITEHEX:75
          WRITEOCT:77; GETBYTE:85 $)
LET WRITEF(FORMAT, A, B, C, D, E, F, G, H, I, J, K) BE
$(1 LET T = @A
    FOR  P = 1 TO GETBYTE(FORMAT, 0)  DO
    $(2 LET  CH = GETBYTE(FORMAT, P)
        TEST  CH='%' THEN
        $(3 LET F, ARG, N = WRITED, !T, 0
            P := P + 1
        $(  LET  TYPE = GETBYTE(FORMAT, P)
            SWITCHON  TYPE INTO
            $(4 DEFAULT:    WRCH(TYPE);  LOOP
                CASE 'S': F := WRITES;   GOTO DO.IT
                CASE 'C': F := WRCH
                CASE 'N':                GOTO DO.IT
                CASE 'O': F := WRITEOCT; ENDCASE
                CASE 'X': F := WRITEHEX
                CASE 'I':                ENDCASE     $)4
            P := P+1;   N := GETBYTE(FORMAT, P)
            N := ('0'<=N<='9') -> N-'0', N+10-'A'
        DO.IT: F(ARG, N); T := T+1 $)3
        OR  WRCH(CH)                    $)2 $)1
```

Note that the operator @, on the second line, yields the address of its right operand. Also, it will be useful to know that

TEST b THEN s1 OR s2

executes command s1 if b is true and otherwise executes the command s2. In the routine above, it is used to test whether we have a percent symbol or not. The translation of WRITEF to

MINICODE is:

```
$ 17:                                  / LET WRITEF(FORMAT, A, ..., K) BE
 LP3 SP14                              / $( LET T = @A
 L1 SP15 LIP2 SP18 L0 SP19             / FOR P = 1 ... FORMAT, 0
 LIG85 K16 SP16                        /       GETBYTE(...,...)
 JL42                                  /       DO
43:LIP2 SP19 LIP15 SP20                /                     FORMAT, P
 LIG85 K17 SP17                        / $(2 LET CH = GETBYTE(..,...)
 L'%' LIP17 X= FL44                    / TEST CH = '%' THEN
 LIG68 SP18                            / $(3 LET F, ... = WRITED, ...
 LIP14 X! SP19 L0 SP20                 /   ARG, N = ..., !T, 0
 L1 AIP15 SP15                         /   P := P + 1
 LIP2 SP23 LIP15 SP24                  /                       FORMAT, P
 LIG85 K21 SP21                        / $( LET TYPE = GETBYTE(...,...)
 JL48                                  /  SWITCHON
50:LIP21 SP24 LIG14 K22 JL51           /  $(4 DEFAULT: WRCH(TYPE); LOOP
52:LIG60 SP18 JIL47                    /  CASE 'S':F:=WRITES;GOTO DO.IT
53:LIG14 SP18                          /  CASE 'C':F:=WRCH
54:JIL47                               /  CASE 'N':            GOTO DO.IT
55:LIG77 SP18 JL49                     /  CASE 'O':F:=WRITEOCT;    ENDCASE
56:LIG75 SP18                          /  CASE 'X':F:=WRITEHEX
57:JL49                                /  CASE 'I':                ENDCASE
 JL49                                  /          $)4
48:LIP21 X? D6 DL50                    /  ... CH INTO
 D'S' DL52 D'C' DL53 D'N' DL54 D'O' DL55 D'X' DL56 D'I' DL57
49:L1 AIP15 SP15                       /   P := P + 1
 LIP2 SP24 LIP15 SP25                  /              FORMAT, P
 LIG85 K22 SP20                        /  N := GETBYTE(...,...)
 L'0' LIP20 X<= FL59                   /  ('0'<=N
 LIP20 L'9' X<= FL59                   /      ...<='9') ->
 LIP20 L'0' X- SP22 JL58               /    N - '0'
59:L10 AIP20 L'A' X- SP22              /    N + 10 - 'A'
58:LIP22 SP20                          /  N := ...
46:LIP19 SP24 LIP20 SP25               / DO.IT:  ...     ARG, N
 LIP18 K22                             /                F(...,...)
 L1 AIP14 SP14                         /  T := T + 1 $)3
 JL45                                  /  OR
44:LIP17 SP20 LIG14 K18                /  WRCH(CH) $)2
45:51:LIP15 A1 SP15                    / // increment P
42:LIP15 LIP16 X<= TL43                /    TO  ...
 XR                                    /   $)1
47:DL46
```

In this BCPL example there is a jump, GOTO DO.IT. In general, the use of jumps is poor programming practice and should be avoided. The reason is that source code containing jumps tends to be difficult to follow and even more difficult to establish correct. Moreover, since experience has shown that the major cost of software is in its maintenance, programs should be clear to those who did not create them. Programs with jumps are usually less clear. But despite this, Knuth [K] has shown that there are occasions when the use of a jump can be justified by the efficiency it brings. The routine WRITEF seems to be one of these. Moreover, this routine is probably one of the most frequently used output routines in the BCPL library,

and one should perhaps resist attempts to make it more "structured" and possibly less efficient. All of this is a warning that beginning programmers should never use jumps in a high level language but should leave them only to the most experienced. In the same way, amateur composers should never use discords. Only the masters know just where they can be tolerated.

A feature of the above routine, observable in its translation, is the knowledge that the address of a routine is a value which may be assigned. Thus, in the execution of the call WRITEF("VALUE = %O6", V), since the layout symbol following the percent symbol is the letter O, the variable F is assigned the address of the routine WRITEOCT. Consequently the call F(ARG, N) is equivalent, in this instance, to the call WRITEOCT(V, 6).

Observe that WRITEF is defined as accepting twelve parameters, but that the routine works with any number up to twelve. The parameters, of course, are loaded consecutively onto the stack before the call. The address of parameter A is captured by the variable T in the declaration

```
LET T = @A
```

whose translation to MINICODE is

```
LP3 SP14
```

Subsequently, when we must look for the next parameter, T is incremented by the command

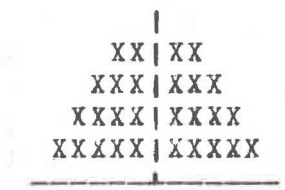
```
T := T + 1
```

on the line following the label DO.IT.

2.5. The towers of Hanoi

We take one more example to illustrate MINICODE. This is the famous puzzle whose solution nicely illustrates the power of recursion in a programming language.

The puzzle assumes that there are three pegs labelled the "source", the "intermediate" and the "destination". On the source peg are a number of discs of increasing size piled in pyramid fashion.

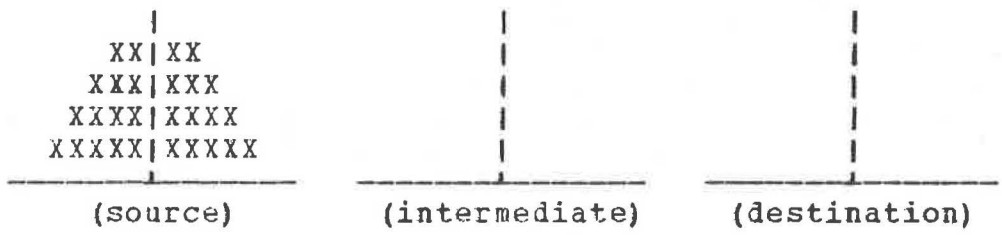


(source)

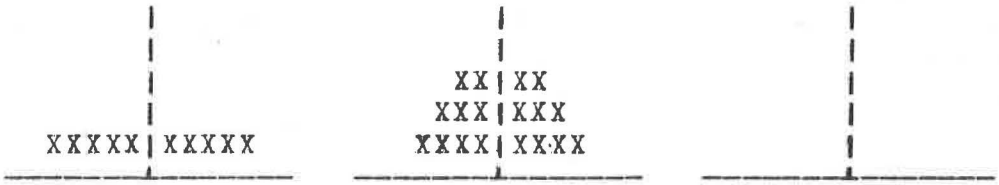
The puzzle is to move this pile to the destination using only acceptable moves. An acceptable move moves only one disc from

the top of one pile to the top of another, but never moves a disc on top of a smaller one.

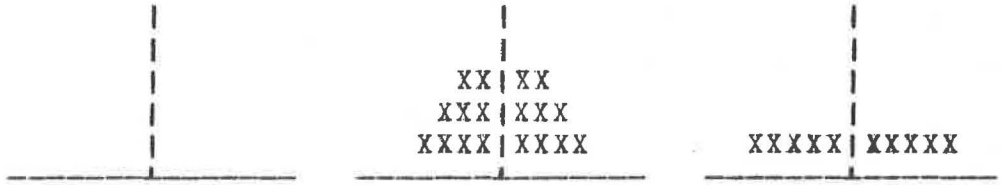
Observing that there is nothing to do if the number, n, of discs is zero, we solve the problem inductively. Assuming then that we know how to move n-1 discs from any one peg to any other, using the third as an intermediate, the solution is as follows.



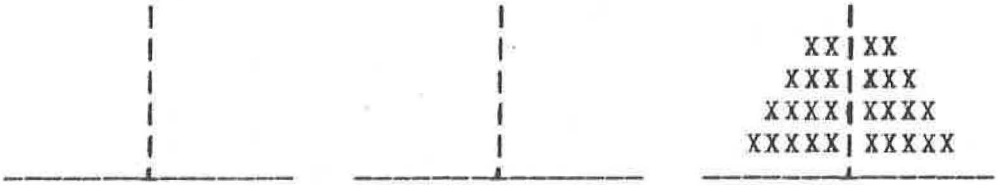
Move n-1 discs from the source to the intermediate.



Then move the bottom disc from the source to the destination,



Then move the n-1 discs from the intermediate to the destination.



The routine, in BCPL, for performing the solution given above is

```
LET HANOI(N, S, I, D) BE
 $( IF N = 0 RETURN
    HANOI(N-1, S, D, I)
    WRITEF("MOVE %N FROM %C TO %C*N", N, S, D)
    HANOI(N-1, I, S, D)   $)
```

and its call is included in the following routine

```
LET START() BE
 $( LET N = READN()
    IF N = 0 FINISH
    HANOI(N, 'S', 'I', 'D') $) REPEAT
```

The solution for the case n = 3, printed by the program is:

```
MOVE 1 FROM S TO D
MOVE 2 FROM S TO I
MOVE 1 FROM D TO I
MOVE 3 FROM S TO D
MOVE 1 FROM I TO S
MOVE 2 FROM I TO D
MOVE 1 FROM S TO D
```

The translation of these two routines to MINICODE is

```
 JL3
$ 1:                              / LET HANOI(N, S, I, D) BE
 L0 LIP2 X= FL4 XR                / $( IF N = 0 RETURN
4:LIP2 L1 X- SP8                  /          N-1
 LIP3 SP9 LIP5 SP10 LIP4 SP11     /                  S,   D,   I
 LIL2 K6                          /   HANOI(..., ..., ..., ...)
 LL499 SP8 LIP2 SP9               /       "...",  N
 LIP3 SP10 LIP5 SP11 LIG76 K6     / WRITEF(..., ..., S, D)
 LIP2 L1 X- SP8                   /          N-1
 LIP4 SP9 LIP3 SP10               /                 I,  S,
 LIP5 SP11 LIL2 K6                /  HANOI(..., ..., ..., D)
 XR                               /    $)
3:JL6
$ 5:7:                            / LET START() BE
 LIG70 K2 SP2                     / $( LET N = READN()
 L0 LIP2 X= FL8 XF                / IF N = 0 FINISH
8:LIP2 SP5 L'S' SP6               /          N, 'S',
 L'I' SP7 L'D' SP8 LIL2 K3        / HANOI(..., ..., 'I', 'D')
 JL7 XR                           /  $) REPEAT
6:XF
2:DL1
499:C"MOVE %N FROM %C TO %C*N"
 G1L5
 Z
```

The loader arranges that the return address from the START routine is always the first cell of the program area. The compiler is therefore careful to place into this cell an instruction which jumps eventually to an XF (in the above example by JL3, 3:JL6, 6:XF). In this particular program this turns out to be unnecessary, since a FINISH command is included explicitly.

To use the assembler, no knowledge of its structure is needed; however, if we wish to understand how it works, then a more detailed study of the software system is necessary. This is accomplished in the next chapter.

Chapter III

The software system

This chapter explains the basic software system associated with the ideal machine. Its purpose is a) to provide further examples of a variety of programming techniques, b) to round out our description of a complete programming system, and c) to clarify the distinction between an assembler, a loader and an interpreter, by a close study of each one. The assembler translates MINICODE programs into instructions, coded in octal, ready for loading. The loader loads the instructions into the computer and the interpreter performs the fetch-execute cycle on the loaded instructions. Each of these will be explained as a BCPL program. It is assumed here that a knowledge of the BCPL language has already been gained from elsewhere [R4].

3.1. The assembler

Up to this point we have regarded the assembler as some kind of "black box" that does the right job. A complete understanding of the system, however, can only come with a detailed study of the construction of the assembler. Its purpose is to translate MINICODE instructions into octal integers representing instructions for execution by the ideal machine. This may at first appear to be a simple task. However, examination of the following sample of MINICODE

```
1:LIP2 TL3 LIP3 LIP4 X< FL3 JL1 3:XR
```

reveals a problem. How can one translate the instruction TL3 without having yet encountered the defining occurrence of the label at 3:XR ? The solution to this problem is one of the important things to learn about assemblers, but first we shall begin with the simple parts.

We shall examine the assembler in stages and then put the pieces together later. The simplest routine, for a start, is

```
GLOBAL $( RDCH:13; CH:101 $)
LET RCH() BE
  $(1 CH := RDCH()
      UNLESS CH = '/' RETURN
      UNTIL CH = '*N' DO CH := RDCH() $)1 REPEAT
```

Thus the purpose of RCH is to assign the next significant character from the input stream to the variable CH, ignoring comments.

3.1.1. Reading constants

The next function is RDCNST. Its purpose is to read a decimal integer, or a character constant from the input stream, on the assumption that CH already contains its first character. The function is:

```
LET RDCNST() = VALOF
$(1 LET A, B = 0, FALSE
    SWITCHON CH INTO
    $(2 CASE '-': B := TRUE
        CASE '+': RCH()
        DEFAULT: WHILE '0' <= CH <= '9' DO
          $( A := 10 * A + CH - '0'; RCH() $)
          ENDCASE
        CASE '*': RCH(); A := CHAR()
          RCH() $)2
    RESULTIS B -> -A, A $)1
```

This function will be familiar since it is similar to functions studied earlier. The difference here is that RDCNST allows for a character constant in place of a decimal integer, so that the assembler may accept, e.g., L'A' as well as L65. The function RDCNST calls another function CHAR to interpret the reading of a character. We now present CHAR.

```
LET CHAR() = VALOF
$(1 LET A = CH
    IF A = '**' THEN
    $(2 CH := RDCH()
        A := CH = 'N' -> '*N',
             CH = 'P' -> '*P',
             CH = 'T' -> '*T',
             CH = 'S' -> '*S',
             CH $)2
    CH := RDCH(); RESULTIS A $)1
```

The purpose of CHAR is to take care of those characters which may not be directly representable and for which the escape character is used. These are *N, *P, *T and *S. Thus, the assembler will be able to read instructions like L'*N' correctly.

3.1.2. The eight operations

We shall now look at the routine OPERATION. This routine assumes that we have already come to an executable instruction, such as JIL2, and have read its first character. We then pass, as a parameter to OPERATION, a value in which the three operation bits of the instruction have already been set. For example, for JIL2, we pass the parameter #020000 (for a sixteen bit cell size). The definition of OPERATION is:

```
MANIFEST $( IBIT:#4000; PBIT=#2000; GBIT=#1000
            ABITS=#0777; DBLBIT=#1000000     $)

LET OPERATION(W) BE
$(1 CH := RDCH()
    IF    CH = 'I'
    THEN $( W := W |/ IBIT; RCH() $)
    IF    CH = 'L'
    THEN $(2 RCH(); STW( W |/ DBLBIT)
       LABREF(RDCNST(), P); RETURN $)2
```

```
TEST  CH = 'P'
THEN $( W := W |/ PBIT; RCH() $)
OR IF CH = 'G'
THEN $( W := W |/ GBIT; RCH() $)
IF W = #070000
THEN $( W := W |/ (('0'<=CH<='9') -> RDCNST(), MINI.X() )
   RETURN $)
$( LET D = RDCNST()
   TEST (D /| ABITS) = D
   THEN STW(W |/ D)
   OR $( STW(W |/ DBLBIT); STW(D) $)1
```

Note that OPERATION results in a two cell instruction in the cases that the operand is a label reference or when the operand is too large to fit into the space available in the cell. The latter occurs when D /| ABITS = D is false. The three new routines used by OPERATION are STW, MINI.X and LABREF. The call STW(W) places the instruction W in the next available cell reserved for the assembled program. The function MINI.X interprets the mnemonics for the execute instruction, e.g., X+, and the routine LABREF takes care of label references such as 20 in LIL20. The definition of STW is:

```
LET STW(W) BE
$(1 !P := W; P, CP := P+1, 0;
    IF P > PROGMAX THEN
    $( WRITEF("*NSEGMENT TOO LARGE")
       FINISH $)1
```

Here it is assumed that the global variable P contains the address of the next available cell in the assembled program area. The variable P is then incremented immediately. The setting of CP, the character phase, to 0 is a precaution concerning the storage of characters. The routine STW has a built-in check, P > PROGMAX, that the program segment may be too large.

3.1.3. The execute instructions

The purpose of the function MINI.X is to decode the special MINICODE execute instruction mnemonics, e.g., X+ is equivalent to X8. Observe that if we were always to write, for example, X8 instead of X+, then the assembler does not need to do this extra work. However, the convenience and readability of MINICODE seem to suggest that it is worthwhile to add this little extra to the assembler. The operation of MINI.X can be learned easily from the following listing

```
LET MINI.X() = VALOF
  SWITCHON CH INTO  // CHAR BEYOND 'X'
    $(1 CASE '!': RCH(); RESULTIS 1
        CASE 'N': RCH(); RESULTIS 2
        CASE '¬': RCH();
          TEST CH='=' THEN $( RCH(); RESULTIS 11 $)
          OR TEST CH='E' THEN $( RCH(); RESULTIS 21 $)
          OR RESULTIS 3
        CASE 'R': RCH(); RESULTIS 4
```

```
CASE '**': RCH(); RESULTIS 5
CASE '/': RCH()
  TEST CH='|' THEN $( RCH(); RESULTIS 18 $)
  OR TEST CH='**' THEN $( RCH(); RESULTIS 7 $)
  OR RESULTIS 6
CASE '+': RCH(); RESULTIS 8
CASE '-': RCH(); RESULTIS 9
CASE '=': RCH(); RESULTIS 10
CASE '<': RCH()
  TEST CH='=' THEN $( RCH(); RESULTIS 15 $)
  OR TEST CH='<' THEN $( RCH(); RESULTIS 16 $)
  OR RESULTIS 12
CASE '>': RCH()
  TEST CH='=' THEN $( RCH(); RESULTIS 13 $)
  OR TEST CH='>' THEN $( RCH(); RESULTIS 17 $)
  OR RESULTIS 14
CASE '|': RCH()
  TEST CH='/' THEN $( RCH(); RESULTIS 19 $)
  OR RESULTIS 19
CASE 'E': RCH(); RESULTIS 20
CASE 'F': RCH(); RESULTIS 22
CASE '?': RCH(); RESULTIS 23
DEFAULT: WRITEF("*NBAD CH %C", CH)
  RCH(); RESULTIS 0 $)1
```

3.1.4. Labels

There is a basic problem with labels in any assembler language. Since we leave it to the assembler where each instruction is to be stored, and since we may often use a label before we have reached its defining occurrence, we may not always know the precise address for a label. The resolution of this problem will now be discussed.

The effect of the call LABREF(N, A) is to determine, if possible from the table of labels, the address of label number N and to store this in the cell whose address is in A. It must also allow for the fact that the label number referred to has not yet been encountered. The definition of LABREF is:

```
LET LABREF(N, A) BE
$(1 LET K = LABV!N
    TEST K < 0 THEN K := -K OR LABV!N := A;
    !A := K
    IF A >= PROGRAMO THEN P := P + 1 $)1
```

From this one may see that the look-up of the address itself from the label number N is accomplished by

```
LET K = LABV!N
```

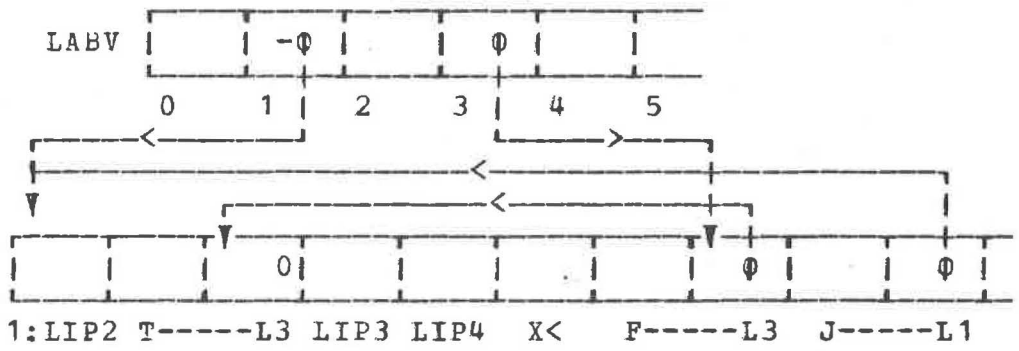
If K is negative, this means that the address of the label is known. It is -K. If K is not negative, then this means that, either LABV!N is zero, i.e., this is the first reference to the

unknown label N, or it is the address of the place where the same unknown label was previously encountered. Note that if A does not point to the global area, A >= PROGRAM0, then we must increment the instruction pointer P.

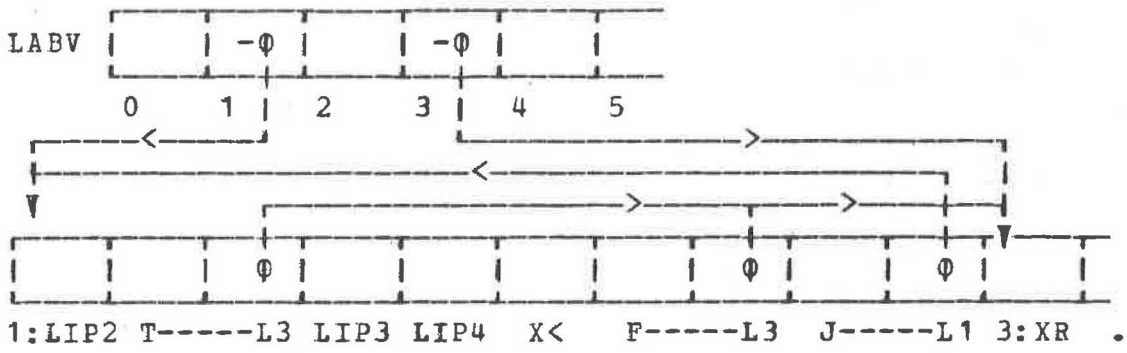
Since the resolution of labels is a vital part of the assembly process, we shall illustrate it by an example. Suppose we are assembling the nonsense piece

```
1:LIP2 TL3 LIP3 LIP4 X< FL3 JL1 3:XR
```

in which there are two forward jumps to the same label number 3. The code which has been generated by the time we have reached JL1 may be represented by the following diagram



Here we see that the address of label 1 has already been resolved, but that of label 3 has not. The effect of the command LABV!N := A, followed by !A := K, is to help create a chain of addresses, ending in 0, connecting together all those places which reference the same unknown label. Thus, LABV!1 points to the position of label 1 in the generated code, but LABV!3 begins a chain of references to label 3, although that label has not yet been seen. After the assembler has encountered label 3, this becomes



This latter change is accomplished by the routine SETLAB which is called when the defining occurrence of a label is encountered. That routine is:

```
LET SETLAB(N) BE
$(1 LET K = LABV!N
    IF K < 0 THEN WRITEF("*NL%N ALREADY SET",N)
```

```
    WHILE K > 0 DO
      $( LET W = !K; !K := P-PROGRAM0; K := W $)
    LABV!N := PROGRAM0-P $)1
```

The reader should work carefully through the loop beginning

```
WHILE K > 0 DO
```

to convince himself that it does follow the chain of references, as described above, and sets the label references correctly. The reason for P-PROGRAM0 instead of P is that the assembler constantly maintains the fiction that it is assembling a segment which begins at cell number zero. This is a fiction because, on many actual machines, the first few cells of memory are usually reserved for the exclusive use of the operating system.

3.1.5. The main part

The next part of the assembler to be described is the heart of it. The major portion is one large case command driven by the phrase

```
UNTIL CH = ENDSTREAMCH DO
SWITCHON CH INTO
```

We shall look at a few of the cases involved. Those concerning labels are

```
CASE '0': CASE '1': CASE '2': CASE '3': CASE '4':
CASE '5': CASE '6': CASE '7': CASE '8': CASE '9':
  SETLAB(RDCNST()); CP := 0; ENDCASE
```

This takes care of the defining occurrence of a label by calling SETLAB, which was described above, whenever one encounters a decimal digit. Note that CP, the character phase, is reset to 0 to ensure that any character string constants which follow will be stored correctly.

Executable instructions are handled, for a sixteen bit machine, as follows

```
CASE 'L': OPERATION(#00000); ENDCASE
CASE 'S': OPERATION(#10000); ENDCASE
CASE 'A': OPERATION(#20000); ENDCASE
CASE 'J': OPERATION(#30000); ENDCASE
CASE 'T': OPERATION(#40000); ENDCASE
CASE 'F': OPERATION(#50000); ENDCASE
CASE 'K': OPERATION(#60000); ENDCASE
CASE 'X': OPERATION(#70000); ENDCASE
```

Each of these cases calls the routine, OPERATION, passing to it a parameter representing an ideal machine instruction in which the operation bits have been set. The routine, OPERATION, then completes the assembly of the appropriate instruction, as described earlier.

3.1.6. Storage reservation

A storage reservation instruction, beginning with the letter D, e.g., D-1, D'H' or DL47, is assembled as follows.

```
CASE 'D': RCH()
  TEST CH = 'L'
  THEN $( RCH(); LABREF(RDCNST(), P) $)
  OR STW(RDCNST())
  ENDCASE
```

If the datum is a label reference, e.g., L47 in DL47, it calls LABREF, which has been described above; otherwise, it stores the value following the letter D using STW.

The storage reservation instruction beginning with the letter G, must be of the form GnLm, where n and m are small non-negative decimal integers, e.g., G101L1. This is handled by

```
CASE 'G': RCH()
  $( LET A = RDCNST() + G
     TEST CH = 'L' THEN RCH()
     OR WRITEF("*NBAD CODE AT %N", P)
     LABREF(RDCNST(), A) $)
  ENDCASE
```

It checks that the letter L is present and then calls LABREF, which has been described above, to allow for the storage of the appropriate address in the global vector. Note that the value of the variable G is the address of the first global cell.

The assembly of the storage reservation instruction beginning with the letter C allows either for the primitive form

```
C5 C69 C76 C76 C79
```

or its equivalent

```
C"HELLO"
```

It is assembled by

```
CASE 'C': RCH()
  TEST CH = '"'
  THEN $(2 LET V = VEC 255 AND I = 0
    CH := RDCH()
    WHILE CH ¬= '"' DO
      $( I := I + 1; V!I = CHAR() $)
    STC(I)
    FOR J = 1 TO I DO STC(V!J); RCH() $)2
  OR STC(RDCNST())
  ENDCASE
```

Note that the assembler allows for a maximum of 255 characters in the string with the form C"...".

The routine STC is machine dependent. For a sixteen bit cell size, storing two characters per cell, it is

```
LET STC(C) BE
$(1 IF CP=0 THEN $( STW(0); CP := 16 $)
    CP := CP - 8;
    !(P-1) := !(P-1) /| (C << CP) $)1
```

A more general routine is obtained by replacing "16" by "CH.P.C*CHAR.SIZE" and "8" by "CHAR.SIZE". This is the place where it is essential that CP, the character phase, has the value zero at the beginning of a string constant.

3.1.7. End of segment

The pseudo-instruction Z, which marks the end of a segment, is recognized by

```
CASE 'Z': OFFSET := WRITE.CODE()
  P, CP := PROGRAM0, 0; RCH()
  ENDCASE
```

This case calls the function WRITE.CODE, which writes out the generated code, in octal, for the segment just concluded. The function WRITE.CODE delivers the length of the code for all segments written out and the assembler uses this information, stored in OFFSET, to decide where to locate the next segment. It also resets P, the instruction pointer, and CP, the character phase.

One might wonder why the code is not written out at the time that each instruction is translated. One reason for not doing this, is that we must wait until all the labels have been resolved, unless we want to complicate the loader. Another is that a more powerful assembler, such as that listed in the Users Guide [P2], compactifies the code by a relative addressing technique [P1] and must have all the code of one section present to do this.

An essential part of the assembler is

```
CASE '$': CASE ':': CASE ' ': CASE '*N': RCH(); ENDCASE
```

which skips over characters which are of no importance.

This concludes our study of a simple version of the assembler.

3.2. The loader

The purpose of the loader is to read a load module, i.e., a set of octal numbers representing ideal machine instructions and data, into the computer. Perhaps the best way to understand the construction of a load module is to examine one which has been generated from a trivial example.

```
GLOBAL $( START:1;  GREETING:101 $)
LET START() BE GREETING()
.
GLOBAL $( WRITES:60; GREETING:101 $)
LET GREETING() BE WRITES("HELLO")
.
```

Here there are two BCPL source program segments each ending with ".". For purpose of illustration, the segments here are very small. In normal BCPL programming a new segment is appropriate after eight or nine pages (about 500 lines) of code. These illustrative segments generate the following MINICODE

```
JL2
$ 1:LIG101 K2 XR 2:XF
G1L1
Z
JL2
$ 1:LL499 SP4 LIG60 K2 XR 2:XF
499:C"HELLO"
G101L1
Z
```

which, in turn, produce the following load module

```
P000000
 070000, 033405, 070000, 005145, 060002, 070004, 070026
 G000001 0+000002
P000007
 033407, 070000, 003406, 012004, 005074, 060002, 070004, 070026
 002510, 042514, 046117
 G000145 0+000010
 .END
```

This load module comes from a more sophisticated assembler than that described above. In particular, it inserts an X0 (no operation) at the entry to each routine or function and reduces some of the two cell instructions to one cell by a relative addressing technique (see [P1]). Also, it is relocatable, i.e., the segments may be loaded into any part of memory. Notice that the load module given above consists of two segments. Each segment begins with Pn, where n is the loading position, relative to zero, of the first instruction of the segment. The first segment is given the position zero and subsequent segments are given loading positions beyond those already loaded. After this follow a number of octal integers separated by spaces and/or commas. Then we have one or more occurrences of Gn m, where n is the octal number (0 for the first) of a global cell and m is the octal content to be loaded into it. Any cell, whether in the global or the program area, containing an address which may be subject to relocation, is preceded by the characters "0+". The end of a load module is signalled by ".END".

3.2.1. Details of the loader

The loader assumes that PROGRAM0 contains the address of the first cell in the actual program area and it uses this as the initial relocation addend. The loader action is governed by a case command, driven by

```
UNTIL CH = ENDSTREAMCH DO
SWITCHON CH INTO                                        .
```

The Pn is accepted by

```
CASE 'P': RCH(); PGM := TRUE
  A := PROGRAM0+OFFSET+RDO()
  ENDCASE                                               ,
```

where RDO is a function which reads an octal integer, OFFSET is the length of previously loaded modules (initially zero) and PGM is a boolean (true or false) variable which is true when A is pointing to the actual program area. The variable A is thus given an appropriate address into which to load subsequent instructions.

An octal instruction is loaded by

```
CASE '0': CASE '1': CASE '2': CASE '3':
CASE '4': CASE '5': CASE '6': CASE '7':
  !A := RDO()
  IF PGM THEN $( A := A + 1; P := A $)
  ENDCASE                                               .
```

The purpose of the command beginning IF PGM THEN is to increment the actual loading position, unless A points to a global cell. The variable P keeps track of the next available position in the program area. A relocatable address is loaded by

```
CASE 'O': RCH(); RCH()
  !A := PROGRAM0+OFFSET+RDO()
  IF PGM THEN $( A := A + 1; P := A $)
  ENDCASE                                               .
```

Note that the character '+' is ignored. A global cell number, i.e., the n in Gn m, is handled by

```
CASE 'G': RCH(); PGM := FALSE
  A := G + RDO()
  ENDCASE                                               .
```

The end of the load module is detected by

```
CASE '.':
  UNTIL CH = '*S' |/ CH = '*N' DO RCH()
  OFFSET := P-PROGRAM0; ENDCASE
```

which skips over the "END" and then calculates the length of code already loaded, which is needed for relocation of the next

module. Of course, we also need the following

```
CASE '*S': CASE '*N': CASE ',': RCH(); ENDCASE
```

to get rid of non-essential characters. Normally the loader and interpreter are combined together in one program. When the loader has finished its job, a state which is determined by reading the ENDSTREAMCH, it hands control to the interpreter.

3.3. The interpreter

Why do we need an interpreter? Well, as we have remarked before, the ideal computer does not necessarily exist, as a piece of hardware, although in these days of microprogramming, it might well be possible to make some computer with this facility look like the ideal computer which we have described. Those not fortunate to have such a facility, will have to make do with what is available. It is therefore necessary to make whatever computer we have at hand act as though it were our ideal computer. This requires a program known as the interpreter.

If our local computer already has a BCPL compiler, then the task is easy, for an interpreter written in BCPL is readily available. But then, of course, we are not going to need an interpreter anyway, except for pedagogic purposes, since we can translate our BCPL programs directly to the machine language of the local computer. If a BCPL compiler is not available, then the interpreter must be written in some other acceptable language. For various versions of the interpreter see the MINICODE Users Guide [P2].

We shall now describe the actions of the interpreter. It is assumed, of course, that the program and the library routines have already been loaded. It is also assumed that the variables A, B, C, P and G represent the five registers, that G contains the address of the first global cell, G!0, that C contains the address of the next instruction in the program area of memory and that P points to a cell on the stack with P!0 and P!1 already loaded with the appropriate contents. It is now necessary to describe the fetch - execute cycle.

3.3.1. The fetch

Assuming that W is a cell used for holding the fetched instruction and that D will later be used to hold its operand, the cycle begins with

```
W := !C; C := C + 1
```

.

This fetches the instruction from memory and increments the instruction counter C. The next step is to find the operand, but first we must determine whether the instruction occupies one cell or two. Assuming a sixteen bit cell and the declarations

```
MANIFEST $( DBLBIT=#1000000; ABITS=#777
  IBIT=#4000; PBIT#2000; GBIT=#1000; FSHIFT=9 $)
```

this is done by

```
TEST W /| DBLBIT = 0
THEN D := W /| ABITS
OR $( D := !C; C := C + 1 $)
```

The raw operand is now in D and the instruction counter is incremented again for a two word instruction. Note that on some computers W /| DBLBIT = 0 can be replaced by W >=0.

3.3.2. Operation modification

The next step is to examine the operand modifiers. This is accomplished by

```
TEST  W /| PBIT ¬= 0 THEN D := D + P
OR IF W /| GBIT ¬= 0 THEN D := D + G
IF    W /| IBIT ¬= 0 THEN D := !D
```

The variable D now contains the modified operand. Observe that P and G modification of the operand is mutually exclusive.

3.3.3. Execute

Now that the modified operand has been determined, the remainder of the interpreter is relatively simple. It consists of two nested case commands as follows.

```
SWITCHON (W >> FSHIFT) /| 7 INTO
$(2 CASE 0: B := A; A := D; ENDCASE
    CASE 1: !D := A;        ENDCASE
    CASE 2: A := A + D;     ENDCASE
    CASE 3: C := D;         ENDCASE
    CASE 4: A := ¬ A
    CASE 5: UNLESS A DO C := D; ENDCASE
    CASE 6: D := P + D; D!0, D!1 := P, C
      P,C:=D,A;             ENDCASE
    CASE 7: SWITCHON D INTO
    $(3 CASE 0:                 ENDCASE
        CASE 1:  A := !A;       ENDCASE
        CASE 2:  A := -A;       ENDCASE
        CASE 3:  A := ¬ A;      ENDCASE
        CASE 4:  C := P!1; P := P!0; ENDCASE
        CASE 5:  A := B * A;    ENDCASE
        CASE 6:  A := B / A;    ENDCASE
        CASE 7:  A := B REM A;  ENDCASE
        CASE 8:  A := B + A;    ENDCASE
        CASE 9:  A := B - A;    ENDCASE
        CASE 10: A := B = A;    ENDCASE
        CASE 11: A := B ¬= A;   ENDCASE
        CASE 12: A := B < A;    ENDCASE
        CASE 12: A := B >= A;   ENDCASE
        CASE 14: A := B > A;    ENDCASE
```

```
CASE 15: A := B <= A;  ENDCASE
CASE 16: A := B << A;  ENDCASE
CASE 17: A := B >> A;  ENDCASE
CASE 18: A := B /| A;  ENDCASE
CASE 19: A := B |/ A;  ENDCASE
CASE 20: a := B NEQV A; ENDCASE
CASE 21: A := B EQV A; ENDCASE
CASE 22: RETURN
CASE 23: B, D := C!0, C!1
  UNTIL B=0 DO
    $(4 B, C := B-1, C+2
    IF A = C!0 DO
      $( D := C!1; BREAK $) $)4
  C := D; ENDCASE
CASE 24: A := A - C!0        // STEP
  TEST 0 <= A <= C!1         // WITHIN RANGE
  THEN C := C!(3+A)          // STEP IN
  OR C := C!2; ENDCASE       // DEFAULT
CASE 25: A:=M.RDCH();  ENDCASE
CASE 26: M.WRCH(A);    ENDCASE
  $)3 )$2
```

After this execution, the interpreter returns to the fetch part of its cycle.

Observe how the construction of this part of the interpreter makes it easy for us to add new X-instructions, if we feel like it. Examination of the listing of the interpreter in the Users Guide [P2] shows many useful additional X-instructions.

Some remarks are in order concerning the translation of Kn (the first CASE 6), which handles the call of a routine or function, whose address is in the A-register. Since the value of D is n, the command

```
D := D + P
```

saves the new value of the P-register temporarily in D. The command

```
D!0, D!1 := P, C
```

inserts the backward stack link and the return address in the first two cells of the new stack frame and the command

```
P, C := D, A
```

sets the P-register to its new value and puts the address of the first instruction of the called function or routine into the instruction counter. Note that the instruction X4 (or XR) easily returns to the status-quo by the commands

```
C := P!1; P := P!0
```

which must be consecutively executed, of course.

In the cases dealing with X25 and X26, it is assumed that M.RDCH and M.WRCH are similar to RDCH and WRCH respectively. They are not necessarily the same because, for example, one may want to load and execute a load module created under the ASCII system on a computer which uses the EBCDIC system. If this is the case, then both M.RDCH and M.WRCH will need translation tables. In addition, these are the two important primitives which communicate with the outside world (usually represented by the operating system).

3.3.4. The case command

Also observe that the execution of the instruction X23 or X? involves the loop

```
UNTIL B = 0 DO
  $(4 B, C := B-1, C+2
  IF A = C!0 THEN
    $( D := C!1; BREAK $) $)4
```

which searches the cells containing the translation of

Dn DLd DV1 DL1 ... Dvn DLn

assuming that C points originally to the first of them. If the BREAK is taken, then the value in the A-register has been matched; otherwise, the original assignment of C!1 to D means that the default label, Ld, is used.

The case dealing with the instruction X24 is included because a more sophisticated assembler can find better ways to emit code for the case command (see [P1,P2]). Thus it is possible for the assembler to digest a set of instructions like

X? Dn DLd Dv1 DL1 Dv2 DL2 ... Dvn DLn

and to generate instructions for the ideal computer as though the original MINICODE were

X24 Dmin Dmax DLd DL1 DL2 DL3 ... DLm

where "min" and "max" are the bounds on the values in the set v1, v2, ... ,vn and the labels L1, L2, ..., Ln are rearranged and possibly expanded so that control is transferred to the correct place by a direct jump governed by the value in the A register, rather than by making a sequential search through the values v1, v2, ..., vn. In some cases this can result in more compact and faster executing code.

(This document is incomplete)

References

[R1] M.Richards, INTCODE -- An interpretive machine code for BCPL, Proceedings of the IFIP Trondheim Conference on Machine Oriented Higher Level Languages, North Holland, 1973.

[R2] M.Richards, BCPL, A tool for compiler writing and system programming, Spring Joint Computer Conference, 1969, pp. 557-566.

[R3] M.Richards, Bootstrapping the BCPL compiler using INTCODE, Cambridge University, August 1973.

[R4] M.Richards, The BCPL Programming Manual, UBC, Computer Science, June 1975.

[P1] J.E.L.Peck, V.S.Manis and W.E.Webb, Code compaction for Minicomputers with INTCODE and MINICODE, Technical Report 75-02, Computer Science, UBC, Vancouver, 1975.

[P2] J.E.L.Peck, The MINICODE users guide, Technical Manual, Computer Science, UBC, Vancouver, 1975.

[K] D.E.Knuth, Structured programming with goto statements, Computing Surveys, V.6, No.4, 1975, pp. 261-301.