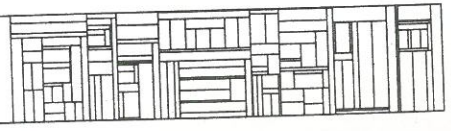
The Emulated OCODE Machine for the Support of BCPL

by

Ole Sørensen

DAIMI PB-45
April 1975

Institute of Mathematics University of Aarhus
DEPARTMENT OF COMPUTER SCIENCE
Ny Munkegade - 8000 Aarhus C - Denmark
Phone 06-12 83 55



CONTENTS

1. Introduction

In the spring of 1973 it was decided to implement the language BCPL [1, 2, 3] on the experimental microprogrammable computer RIKKE-1 [5, 6] being constructed in this department. The language was chosen to be the systems programming language for RIKKE-1, one argument was the possibility of transferring the Oxford Operating system OS 8 [9] to RIKKE-1.

As there existed an intermediate object language OCODE [2] for the translation of BCPL, one way of accompl ishing this goal was to write an emulator for some internal representation of OCODE (or a slightly modified version of OCODE) and an OCODE assembler that would assemble symbolic OCODE into this internal representation.

This paper describes the design process for an internal representation of OCODE, the resulting machine, the emulator, and the assembler, and finally there is a discussion of our experiences of running the OCODE machine during the past 8 months. Some future analysis and possible modifications are mentioned.

The OCODE machine was designed and the OCODE emulator was written during the summer and autumn 1973 by Bjarne Stroustrup and the author. The OCODE machine has been running on RIKKE-1 with a mini-system written in BCPL since July 1974. Some modifications have been made during this period.

2. OCODE

In [2] Martin Richards defines OCODE as an intermediate object language for the translation of BCPL. OCODE is a symbolic language running on an imaginary stack machine which corresponds closely to the structure of BCPL.

As shown in figure 1 the BCPL compiler first translates a program into a tree representation which is then translated into OCODE (this is described in more detail in [3]). At last a code generator generates the code for the target machine. We use the same structure, but our code generator is called an assembler because our machine code is so close to the symbolic OCODE.

The OCODE machine has 3 sorts of data: local, global and static, each with its own accessing method. Local variables are placed on the stack and are addressed relative to the activation pointer, P, which points to the currently active stackframe. Global variables are placed in a vector of consecutive cells and are addressed relative to the start of the vector. Static variables are addressed by symbolic addresses (labels) so their allocation in the OCODE machine is not specified.

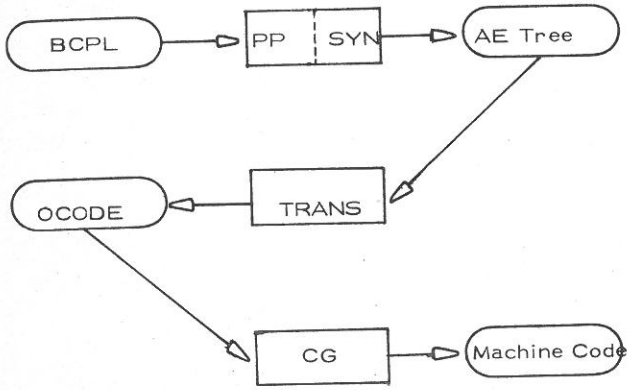


Figure 1 : The structure of the BCPL compiler.

The stack is organized as shown in figure 2. The activation pointer P points to the first cell in the active stackframe, and S gives the number of cells in the stackframe. S is adjusted each time an element is pushed on the stack or popped off the stack. P is adjusted on procedure or function entry or exit: on entry P is set to a new position, the old value of P is filled into the cell that P is now pointing to, and the return address in the code is put into the next cell; on exit P is reset to the old value, S is adjusted, and the return address is used to resume execution of the previous procedure.

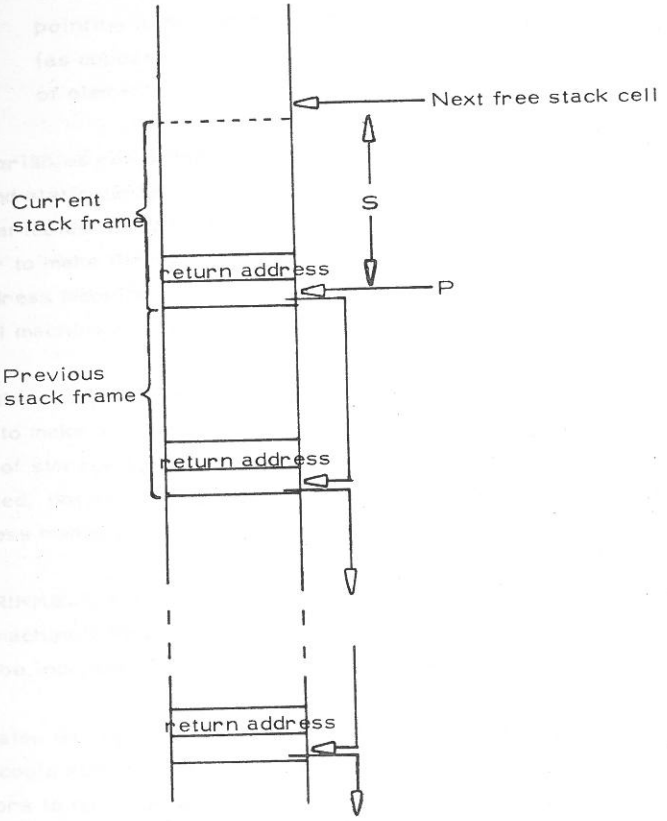


Figure 2 : The stack

3. Design Criteria

It was decided to keep close to the OCODE described by Martin Richards, but on the other hand we wanted to make modifications that would optimize storage utilization and execution time if possible.

The data organization and accessing method of the OCODE machine is well described in [2]. It can be directly transferred to a machine with the following registers:

- G pointing to the global vector
- P containing the activation pointer
- S pointing to the top element of the stack
  (as opposed to [2] where S denotes the number of elements in the active stackframe)

Local variables can be addressed relative to P, global variables relative to G, and static variables by absolute address. (The decision about static variables has later been revised: see section 7.) Furthermore, in order to make the addressing independent of the physical machine, someaddress mapping should be used to transform virtual addresses into physical machine addresses, e.g. BASE-LIMIT registers.

As for the code, no internal representation is indicated in [2] so here we had to make our own decisions. A major design criterion was that the amount of storage space occupied by a "typical" program should be minimized, not only to save space but hopefully also execution time since less memory references would be required.

Since RIKKE-1 is a 16 bit machine our OCODE machine should be a 16 bit machine with a store of up to 64K words. Otherwise the design should be independent of the RIKKE-1 hardware.

It was also decided that the OCODE emulator should be designed such that it could run in some multiprogramming system allowing different emulators to operate concurrently.

4. Statistics

In order to be able to decide on a reasonable internal representation of OCODE which would optimize memory utilization for the code, we picked out some BCPL routines from the runtime library (WRITES, UNPACKSTRING, PACKSTRING, WRITED, WRITEN, NEWLINE, READN, WRITEOCT, WRITED, and WRITEF) and from the BCPL compiler (JUMPCOND, TRANSSWITCH, and TRANSFER), and translated them into OCODE, using a BCPL compiler running on the department's GIER machine. (This was a bootstrap version of the compiler in INTCODE [4], and the INTCODE simulator was written in ALGOL, so it was a very slow translation, taking about 8 hours).

This material was, of course rather small (about 300 lines of BCPL, of which almost 1/3 was manifest and global declarations). The reasons for not taking more material at that time were the slow translation speed and the fact that the results were to be calculated by hand (since we did not know what to look for we could not make an automatic analysis).

Table 1 shows the distribution of the instructions in the translated code (leaving out the assembler/loader directives which are given in table 2).

The directive GLOBAL produced by our version of the compiler seems to be a replacement for the directives INITGL and INITGN described in [2]. It is of the form

GLOBAL k g_1 Ln_1 g_2 Ln_2 .... g_k Ln_k

where k, g_1, g_2, ..., g_k are integers, g_1 ... g_k representing global variable numbers, and Ln_1 ... Ln_k are assembly parameters (labels).

The most important information we get from table 1 is that a few instructions occur very frequently whereas most instructions occur rather infrequently. In our attempts to optimize storage utilization we must therefore pay special attention to the frequently occurring instructions since the total result of the optimization will be heavily influenced by these instructions.

Table 1. Distribution of OCODE instructions.

	Compiler rout.		Runtime rout.		Total	
	no.	%	no.	%	no.	%
LP	64	15.3	97	19.0	161	17.3
LLP			2	0.4	2	0.2
SP	7	1.7	32	6.3	39	4.2
LG	74	17.7	26	5.1	100	10.8
LLG						
SG	18	4.3	3	0.6	21	2.3
LL			6	1.2	6	0.6
LLL						
SL						
LN	50	12.0	88	17.2	138	14.9
TRUE			1	0.2	1	0.1
FALSE						
LSTR						
MULT			1	0.2	1	0.1
DIV			1	0.2	1	0.1
REM			1	0.2	1	0.1
PLUS	22	5.3	38	7.4	60	6.5
MINUS	4	1.0	8	1.6	12	1.3
EQ	5	1.2	9	1.8	14	1.5
NE						
LS	1	0.2	5	1.0	6	0.6
GR			1	0.2	1	0.1
LE	1	0.2	4	0.8	5	0.5
GE			1	0.2	1	0.1
LSHIFT			2	0.4	2	0.2
RSHIFT			7	1.4	7	0.8
LOGAND			4	0.8	4	0.4
LOGOR			2	0.4	2	0.2
EQV						
NEQV						
NEG			2	0.4	2	0.2
NOT	4	1.0			4	0.4
RV	21	5.0	15	2.9	36	3.9
STIND			5	1.0	5	0.5
JT	4	1.0	4	0.8	8	0.9
JF	6	1.4	17	3.3	23	2.5
JUMP	9	2.2	15	2.9	24	2.6
GOTO			6	1.2	6	0.6
FINISH	1	0.2	1	0.2	2	0.2
SWITCHON	1	0.2	2	0.4	3	0.3
STACK	60	14.4	40	7.8	100	10.8
STORE	12	2.9	15	2.9	27	2.9
RES			2	0.4	2	0.2
RSTACK			2	0.4	2	0.2
FNAP	6	1.4	3	0.6	9	1.0
RTAP	37	8.9	18	3.5	55	5.9
SAVE	3	0.7	9	1.8	12	1.3
FNRN			2	0.2	2	0.2
RTRN	8	1.9	14	2.7	22	2.4
	418		511		929	

Before turning to the internal representation of instructions let us note the following:

1) The instructions STACK and SAVE have identical effect, and will therefore be implemented as one instruction denoted STACK.

2) The differences between RTAP and FNAP can be moved to the return, so that we need only one instruction for calling routines or functions, denoted RTFNAP.

3) The LSTR instruction can internally be represented by LLL loading a pointer to the string on the stack.

Table 2. Assembler/loader directives.

	Compiler routines	Runtime routines	Total
LAB	22	45	67
ENTRY	3	9	12
DATALAB		3	3
ITEML		3	3
ITEMN			
GLOBAL	1	1	2

5. Internal representations and optimizations

Our first assumptions are the following:

1) The OCODE machine has a store of (up to) 64K 16 bit words.

2) We must be able to address any location of the store, so addresses that are not limited by some criterion must occupy 16 bits.

3) Any instruction must be addressable, i.e. it must start at a word boundary.

5.1. Initial version

Assumptions 1, 2, and 3 are made. Because of assumption 2 the address part of addressed instructions must occupy 1 word, and because of assumption 3 the operation code must then also occupy 1 word. This does not apply to the instructions accessing the global vector. If we limit the size of the global vector to 1024 words, then these instructions can be contained in 1 word, with a 6-bit opcode and a 10-bit address field. Addressless intructions will also occupy 1 word because of assumption 3. Table 3 shows the storage space required by our example programs under these assumptions.

5.2. First optimization: Byte addressing

Now it seems to be a waste of storage space to use 16 bits for an operation code, remembering that there are about 50 OCODE instructions, so let us replace assumption 3 by

3') Any instruction following a LAB, ENTRY, RTAP or FNAP must be addressable.

This change can be made because we can only branch to these instructions.

Now we can split up each word in 2 8-bit bytes letting each operation code occupy (at most) one byte, so that addressed instructions will occupy 3 bytes (1 byte for the opcode, 2 bytes for the argument), addressless instructions will occupy 1 byte, and instructions accessing the global vector will occupy 2 bytes. However assumption 3' tells us that some instructions must start in the first byte of a word, so we must

Table 3. Storage requirements using word- and byte-addressing

	no.	%	word addressing bytes	%	byte addressing bytes	%
LP	161	17.3	644	20.9	483	20.5
LLP	2	0.2	8	0.3	6	0.3
SP	39	4.2	156	5.1	117	5.0
LG	100	10.8	200	6.5	200	8.5
LLG						
SG	21	2.3	42	1.4	42	1.8
LL	6	0.6	24	0.8	18	0.8
LLL						
SL						
LN	138	14.9	552	17.9	414	17.6
TRUE	1	0.1	2	0.1	1	0.0
FALSE						
MULT	1	0.1	2	0.1	1	0.0
DIV	1	0.1	2	0.1	1	0.0
REM	1	0.1	2	0.1	1	0.0
PLUS	60	6.5	120	3.9	60	2.5
MINUS	12	1.3	24	0.8	12	0.5
EQ	14	1.5	28	0.9	14	0.6
NE						
LS	6	0.6	12	0.4	6	0.3
GR	1	0.1	2	0.1	1	0.0
LE	5	0.5	10	0.3	5	0.2
GE	1	0.1	2	0.1	1	0.0
LSHIFT	2	0.2	4	0.1	2	0.1
RSHIFT	7	0.8	14	0.5	7	0.3
LOGAND	4	0.4	8	0.3	4	0.2
LOGOR	2	0.2	4	0.1	2	0.1
EQV						
NEQV						
NEG	2	0.2	4	0.1	2	0.1
NOT	4	0.4	8	0.3	4	0.2
RV	36	3.9	72	2.3	36	1.5
STIND	5	0.5	10	0.3	5	0.2
JT	8	0.9	32	1.0	24	1.0
JF	23	2.5	92	3.0	69	2.9
JUMP	24	2.6	96	3.1	72	3.1
GOTO	6	0.6	12	0.4	6	0.3
FINISH	2	0.2	4	0.1	2	0.1
SWITCHON	3	0.3	72	2.3	72	3.1
STACK	112	12.1	448	14.5	336	14.3
STORE	27	2.9	54	1.7	27	1.1
RES	2	0.2	8	0.3	6	0.3
RSTACK	2	0.2	8	0.3	6	0.3
RTFNAP	64	6.9	256	8.3	192	8.2
FNRN	2	0.2	4	0.1	2	0.1
RTRN	22	2.4	44	1.4	22	0.9
NOOP					72	3.1
	929		3086		2353	

add a nooperation (NOOP) to the instruction set which can be inserted in the second byte of the previous word if necessary. The number of NOOP's to be inserted is estimated as:

(#LAB + #ENTRY + #RTAP + # FNAP) / 2
= (67 + 12 + 55 + 9) / 2 = 71.5 ≈ 72

Table 3 shows the memory requirement after this optimization, and the gain of storage space amounts to

23.8%

5.3. Second Optimization: Short arguments

We now turn to the restriction of assumption 2 that addresses (and constants) should always occupy 16 bits. Is this reasonable?

If we allow for 2 or 3 versions of the same instruction with different sizes of the argument we can in each case use that version of the instruction in which the actual argument can be contained. We now get this modification of assumption 2:

> 2') We must be able to address any location of the store, so addressed instructions (except those accessing the global vector) must have a version with a 16-bit argument.

In the following it is shown that this flexibility gives a substantial saving of storage space.

First look at the constants. What are constants used for? Probably mainly for the step length in for-loops, and for indexing in (small) vectors, and if this is true, most constants will be "small". Now, we have already on page 9 decided upon an instruction format for accessing the global vector with 6-bit opcode and a 10 bit argument. The same format could be used for instructions with a constant argument in the interval [-512, 511], and in our example it turns out that all constants are within this interval. Of course we still need the ability to have a 16-bit constant, so we must have 2 versions of the LN instruction.

But we can go even further if we note that about half of the constants

are used immediately in an arithemetic, comparison or shift operation. These instructions operate on the 2 top elements on the stack and place their result on the stack. If we add a version of each of them with a 6-bit opcode and a 10-bit argument, we can replace the combination of an LN instruction followed by one of these instructions by the new version.

Example:

if k is in the interval [-512, 511], then

```
LN k
MINUS
```

is replaced by

```
MINUS10  k
```

Thus we save one byte of code, and moreover we save an instruction decoding and a push and a pop of the stack.

If our arithmetic or comparison operation is symmetric, we may have some other load instruction between the LN and the operation.

Example:

if k is in the interval [-512, 511], then

```
LN k
LG n
PLUS
```

is replaced by

```
LG n
PLUS10  k
```

Now we turn to the addresses. As far as global variables are concerned we have already made the optimization (page 9).

For static variables we cannot hope to gain anything since we have decided that they should be addressed by absolute addresses. (This decision has later been revised, see section 7.).

The dynamic (local) variables, however, are addressed relative to the P register and their addresses are known at compile time, so here we can use the shorter format for "small" addresses. But how often do we really need to address up to 1024 local variables? Certainly only if we use vectors. Otherwise the number of local variables is probably in most cases not more than, say, 10 or 12, so we can use an even shorter format for addressing these variables. With a 4-bit opcode and a 4-bit argument we can squeeze an addressed instruction into one byte, and it turns out that about 80-90% of these instructions actually will use this format so there is a substantial saving of space, since the instructions involved (LP, SP, and STACK) are about 30-35% of the total number of instructions. *)

The result of these optimizations is given in table 4, showing the distribution of the instructions with different argument sizes, and the storage requirements after the optimization.

Note that more than 50% of the instructions are 1-byte instructions, and that almost 30% of all instructions use the very short format (4 bit opcode - 4 bit argument). The gain of storage space is a further

36.5%

compared to the previous version, and

51.6%

compared to the first version.

Finally we decided that jump instructions (JUMP, JT, JF, RES) should use relative addressing, and so they can also have a version with a 6-bit opcode and a 10-bit argument. However the assembler can only make the optimization for backward jumps since the size of the argument is not known for forward jumps until the label is defined. The optimization of backward jumps is not included in table 4, so the gain of storage space is a little more than mentioned above.

*) Later results show that the constant arguments as well, can in most cases use this ultrashort format, since 0 and 1 are the most used constants, and other one-digit constants are also used much more than larger numbers. Negative numbers, even -1, do not seem to be of any importance. However, a larger amount of data should be analyzed before final conclusions are drawn.

Table 4. The result of using "short" arguments

	no arg	4-bit arg	10-bit arg	16-bit arg	Total no.	Total bytes	Total %
LP	-	146	15		161	176	11.7
LLP	-	-	2		2	4	0.3
SP	-	29	10		39	49	3.3
LG	-	-	100	-	100	200	13.4
LLG	-	-		-			
SG	-	-	21	-	21	42	2.8
LL	-	-		6	6	18	1.2
LLL	-	-					
SL	-	-					
LN	-	-	63		63	126	8.4
TRUE	1	-	-	-	1	1	0.1
FALSE		-	-	-			
MULT		-	1	-	1	2	0.1
DIV		-	1	-	1	2	0.1
REM		-	1	-	1	2	0.1
PLUS	18	-	42	-	60	102	6.8
MINUS	1	-	11	-	12	23	1.5
EQ	5	-	9	-	14	23	1.5
NE		-		-			
LS	4	-	2	-	6	8	0.5
GR		-	1	-	1	2	0.1
LE	3	-	2	-	5	7	0.5
GE		-	1	-	1	2	0.1
LSHIFT		-	2	-	2	4	0.3
RSHIFT	2	-	5	-	7	12	0.8
LOGAND	4	-	-	-	4	4	0.3
LOGOR	2	-	-	-	2	2	0.1
EQV		-	-	-			
NEQV		-	-	-			
NEG	2	-	-	-	2	2	0.1
NOT	4	-	-	-	4	4	0.3
RV	36	-	-	-	36	36	2.4
STIND	5	-	-	-	5	5	0.3
JT	-	-		8	8	24	1.6
JF	-	-		23	23	69	4.6
JUMP	-	-		24	24	72	4.8
GOTO	6	-	-	-	6	6	0.4
FINISH	2	-	-	-	2	2	0.1
SWITCHON	3	-	-	-	3	72	4.8
STACK	-	96	16		112	128	8.6
STORE	27	-	-	-	27	27	1.8
RES	-	-		2	2	6	0.4
RSTACK	-	-	-	2	2	6	0.4
RTFNAP	-	-	64		64	128	8.6
FNRN	2	-	-	-	2	2	0.1
RTRN	22	-	-	-	22	22	1.5
NOOP	72	-	-	-	72	72	4.8
no.	221	271	369	65	926	1494	
%	23.9	29.3	39.8	7.0			

6. Input/Output and Interface to Environment

No attempt is made here to specify completely a micromonitor allowing multiple emulators. Only a number of restrictions which can be reasonably required of emulators in such a system are mentioned as design goals for the OCODE emulator.

6.1. Input/Output

The emulator must not directly perform I/O on the physical machine but should issue requests to an I/O nucleus which is common to all emulators in the system [10].

The OCODE machine should initiate I/O by writing a pointer to the I/O request block in cell 0 of its store. Whenever the main loop of the emulator finds a nonzero content in cell 0 it will assume that it is a pointer to a request block for the I/O nucleus or for the micromonitor. The emulator must then take proper action to pass the request on to the nucleus (e.g. convert logical (OCODE) addresses to physical (RIKKE) addresses), and then return control to the system.

6.2. Interrupts

In order to allow for I/O to take place concurrently with normal computation, and to prevent a virtual machine from monopolizing the physical processor, some sort of interrupt system must exist. However, in a microprogrammed computer with a large number of control registers and lines it is not appropriate to have a hard interrupt since it must be disabled most of the time. (At least this is true with RIKKE-1, unless we restrict the use of hardware resources very much and guarantee that no error can occur between the occurrence of an interrupt andthe return to the interrupted program.)

A better solution would be to let the "interrupt" be a testable condition, and impose the programming discipline that any emulator in the system should test for this condition at regular intervals, the (maximum) length of which is determined by the system (but it should not be shor-

ter than to allow for all normal virtual machine instructions to be decoded and executed within the interval). If the interval is fixed at 50 μs (in the RIKKE-1 implementation), all OCODE instructions except the SWITCHON instruction can be executed.

Since the RIKKE -1 hardware does not provide a single testable condition indicating change of the status of some I/O device we cannot directly use this approach, but a logically equivalent system can be made by polling the status flags of all devices, and comparing them to the previous status.

6. 3. Emulator entry and exit

Having detected that it has to exit (either because of a request or because of an "interrupt") the emulator must enter a routine saving the state of the virtual machine and resetting the hardware to the system standard before returning to the system. When the emulator is entered the opposite function is performed: a routine initializing the hardware for the emulator and the virtual machine is executed before the emulator starts emulating the instructions of the virtual machine.

Subsections 6. 1. - 6. 3. imply a structure for the emulator as shown in figure 3.

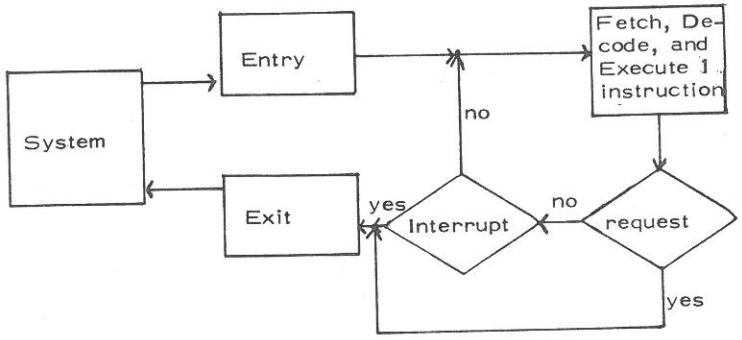


Figure 3. Structure of emulator interface to environment.

6.4. Provisional I/O instructions

Because we could not expect an I/O nucleus to be available at the same time as the OCODE emulator, and in order to be able to test the emulator and the I/O nucleus independently, some primitive I/O instructions were included in the first version of the OCODE machine instruction set. These were:

STARTI , using the contents of the stacktop as device number for selecting input device

STARTO , using the contents of the stacktop as device number for selecting output device

INPUT , reading one character (or word) from the selected input device, and placing it on top of the stack

OUTPUT , writing the contents of the stacktop on the selected output device.

Note that these instructions use physical device numbers, and that they make no interpretation of the binary values they input or output, which implies that all character conversion routines must be written in BCPL.

6.5. "The SWITCHON Problem"

The problem of the SWITCHON instruction is that its execution may take such a long time that we cannot let it complete without allowing for interrupts to be handled. The approach we have taken to solve this problem (which may not be the final solution) is the following:

The SWITCHON instruction produced by the compiler:

SWITCHON k L_d K_1 Ln_1 K_2 Ln_2 ... K_k Ln_k

is assembled into

LN k

SWITCHON K_1 Ln_1 K_2 Ln_2 ... K_k Ln_k L_d

so that our SWITCHON instruction will assume the value k to be on the stack.

The SWITCHON instruction format has one byte for the operation code and one word for each of its arguments (so that there may be an "empty" byte between the operation code and the first argument), and in the following description it is assumed that the program counter (PC) points to the first argument (K_1) when the instruction has been decoded and execution starts.

The instruction will in sequence compare K_1, K_2, ... K_k with the switch variable, which is on the stack and if it finds that K_i is equal to this value it will jump to the label Ln_i (which is assembled to a relative address). If no match is found it will jump to the default label (L_d).

The instruction now works roughly like this:

```
1. pop k off the stack
2. test k > maxno then
      $( k:= k - maxno
         count:= maxno
      $) or
      $( count:= k
         k:= 0
      $)
3. for i = 1 to count do
      compare and branch conditionally
4. test k = 0 then
      use default label
   or
      $( push k on stack
         take care that the next instruction
         to be executed is SWITCHON
      $)
```

This implies a lot of overhead, especially point 4.

Leaving out points 2 and 4, we have the following more exact description of the SWITCHON instruction

```
$(
    pop ( )
    for i = Y to 1 by -1 do
    $(
        x:= rv PC
        PC:= PC + 2
        if x = stacktop goto L              // match
    $)
    PC:= PC + 1                             // default
L:  PC:= PC + rv (PC - 1) - 1
    pop ( )
$)
```

where the routine pop ( ) is defined as

```
let pop ( ) be
        $( Y:= stacktop
           S:= S - 1
           stacktop:= rv S
        $)
```

7. A New Access Method for Statics

The modification described in this section was first proposed by Nigel Derrett, and is being implemented by Eric Kressel and Ib Holm Sørensen.

The idea is that static data items (static variables, labels, routines, functions, strings) should be addressed relative to a register, like local and global variables.

The new register is called

DB , Data Base register

Now this does not give us very much unless we further require that the DB register points to the data area of the (possibly seperately compiled) code segment currently being executed. This means that the DB register must be changed when the locus of execution changes from one segment to another. This can only occur in the following situations:

- routine or function call
- routine or function exit
- goto

so the DB register must be updated exactly in these situations.

Having introduced the DB register it seems natural to have a similar register pointing to the code segment:

CB , Code Base register

which is updated at the same time as the DB register.

We can now see what information is necessary to identify a label (or a routine or a function). First of all we must be able to identify the code- and data-area of the segment containing the label. This is done by the SEGMENT DESCRIPTOR (fig. 4) consisting of the two base register values to be used in the new segment. Secondly we must know the entry point within the code area corresponding to the label. Since there may be several entry points in one segment, this information must be separate from the segment descriptor, so we have a LABEL DESCRIPTOR (fig. 4) containing a pointer to the SEGMENT DESCRIPTOR and the relative address of the entry point within the code segment.

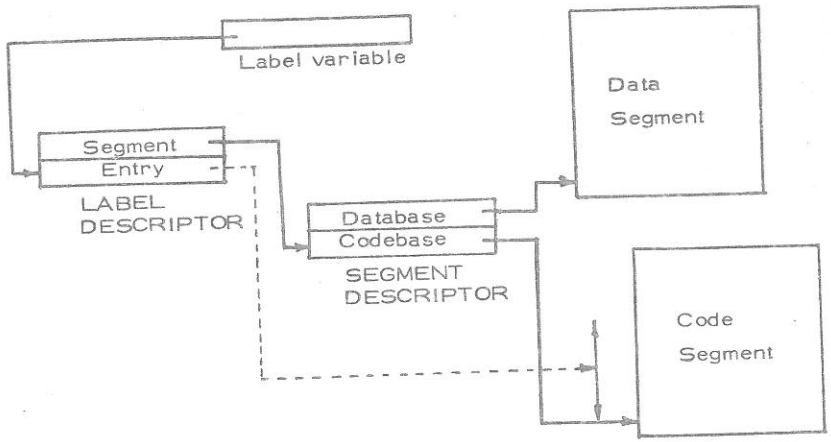


Figure 4. Structure of a Label.

A variable containing a pointer to a LABEL DESCRIPTOR will be denoted a label variable.

Now we can describe what happens when a goto is executed. Assume that the OCODE machine has the top element of the stack in a register called stacktop, and that it contains a pointer to a LABEL DESCRIPTOR.

Using the following routines (described in BCPL)

```
let pop ( ) be
$( Y:= stacktop
   S:= S -1
   stacktop:= rv S
$)
```

```
and goto ( ) be
$( Z:= rv (Y + 1)              ;/ "segment"
   unless Z = IB do
      $( IB:= Z                // update IB
         CB:= rv Z             // update Code Base
         DB:= rv (Z + 1)       // update Data Base
      $)
   PC:= rv Y                   // update Program Counter
$)
```

the OCODE instruction GOTO will now simply do the following:

```
$( pop ( )
   goto ( )
$)
```

Here we have assumed an extra register IB which always points to the SEGMENT DESCRIPTOR of the currently active segment. Thereby we have optimized the goto ( ) routine since we avoid updating CB and DB when it is not necessary. Y and Z are "processor registers" used only for intermediate storage.

Routines and functions are referenced in the same way as labels, but in order to be able to make proper return, the return information in the stack must be extended (fig. 5).

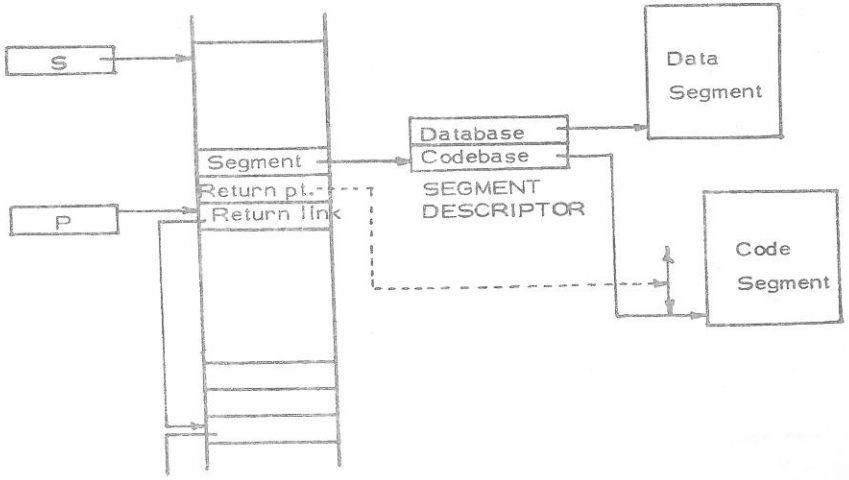


Figure 5. Return information in stack.

Now the routine- and function call instruction (RTFNAP k) becomes:

```
$( rv (P + k) := P          // return link
   rv (P + k + 1) := PC     // return point in code
   rv (P + k + 2) := IB     // return segment
   P := P + k
   Y := stacktop
   goto ( )
$)
```

Routine return (RTRN) becomes:

```
$( S := P - 1
   stacktop := rv S
   Y := P + 1
   P := rv P
   goto ( )
$)
```

Funtion return (FNRN) becomes:

```
$( S := P
   Y := P + 1
   P := rv P
   goto ( )
$)
```

The advantages of this system are:

1) We can use a shorter instruction format for static data references, so we save space in the code.

2) The code is relocatable, since it is only referenced via the SEGMENT DESCRIPTOR.

3) The code can be shared by processes, and interactions can be prevented by giving each process its own data area.

4) Loading will be easier, since the loader has no chains of references to update, but need only establish the descriptors.

Of course we do not get these advantages without paying for them with some disadvantages:

1) The descriptors, and the return information in the stack require extra space.

2) There will be more work on procedure entry and exit, and on goto.
3) 3 new virtual machine registers are introduced.

It would have been nice if advantage 2 was also valid for the data. This is not possible because we can construct pointers in BCPL using the lv operator.

8. The OCODE Machine

In this section the OCODE machine is described as of March 1975 with addition of the modification described in section 7. The machine has been running on RIKKE-1 for about 8 months with a few modifications being made during this period.

8.1. Registers

The OCODE machine has 16 registers:

BASE : Physical (RIKKE) address of the first cell in the OCODE machine store. Any other address is relative to BASE.

LIMIT : The size (number of cells - 1) of the OCODE machine store.

G : Pointer to the global vector.

P_0 : Base of stackarea. Used to detect stackunderflow.

P : Activation pointer (pointer to the currently active stackframe).

S : Stacktop pointer.

T : Stack Limit. Used to detect stack overflow, and to protect the top of the store against overwriting.

IB : Pointer to the SEGMENT DESCRIPTOR of the active segment.

CB : Code Base. Pointer to the code area of the active segment.

DB : Data Base. Pointer to the data area of the active segment.

ARG : The argument (if any) of the instruction being executed.

OP : Bits 8 - 15 contain the operation of the instruction being executed.
Bits 0 - 7 contain the operation code of the next instruction to be executed if it has been fetched.

PC : Program Counter. Contains the address relative to CB of the next word to be read from the code stream.

CR : Condition Register. Used for various software interrupts.

ET : Errortype. If an interrupt has occurred ET contains the reason for interrupt.

DC : Pointers to decoding tables. Used internally by the emulator.

8.2. Address Space

The logical address space (the store) of the OCODE machine is a linear store of 16-bit words. The size of the store is determined by the LIMIT register which contains the highest legal address. The maximum size is 65536 words. The upper part of the store can be protected against overwriting by means of the T register which contains the highest legal address for writing. T must never exceed LIMIT.

The BASE register is used for transforming addresses of the OCODE machine into physical addresses. Any other address mapping could be used instead of the BASE - LIMIT approach used here.

8.3 The Global Vector

An instruction accessing the global vector has a 10-bit address field which is interpreted as a signed integer in the interval [-512, 511]

(2's complement). The maximum size of the global vector is thus 1024 words with the G register pointing to global number 0.

8.4. Static Variables

Static variables are addressed relative to the DB register which is updated when a GOTO, RTFNAP, RTRN or FNRN instruction is executed. See section 7 for details.

8.5. The Stack

The storage area in which the stack can grow and shrink is bounded by the addresses contained in the registers P_0 (stack base) and T (stack limit). The P register contains the activation pointer to the currently active stackframe, and the S register points to the top element of the active stack *).

Details about the stack administration (procedure call and return) are given in section 7 and figure 5.

8.6. The Code

Code is allocated in segments which are accessed by means of labels (or function or routine entry points) as described in section 7 and figure 4. The segment descriptor of the active segment (i.e. the segment in which execution is currently going on) is copied into the CB (codebase) and DB (database) registers, and a pointer to the descriptor is in the IB register. These registers are only modified by the GOTO, RTFNAP, RTRN and FNRN instructions as described in section 7.

The code consists of a sequence of instructions, each of which may occupy 1, 2 or 3 8-bit bytes (except for SWITCHON which occupies 4 bytes for each branch). As the machine has 16-bit words, 3-byte instructions always cross a word boundary and 2-byte instructions may cross a word boundary.

*) In the RIKKE-1 implementation the top element is kept in a machine register (LR [0]) and is only written down to store when the stack is pushed or when a STACK or STORE instruction is executed.

When instructions are executed in sequence, a pseudo-byteaddressing is applied. Bits 0 - 7 of register OP are used as a buffer to hold the second byte of the last word that was read from the instruction sequence, and a condition flag is used to tell whether this is an OPCODE or it is part of the argument of the previous instruction.

However, branch instructions (JUMP, JT, JF, GOTO, RTFNAP, RTRN,and FNRN) only branch to words, so execution is always resumed in the first byte (bits 8 - 15) of the word branched to. This implies that it may be necessary to insert a NOOP before an instruction which one wants to branch to so that the instruction is placed correctly in the first byte of a word.

8.7. Instruction Formats

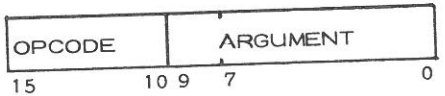
There are 4 different instruction formats in our OCODE machine. Some OCODE instructions as produced by the BCPL compiler have 2 or 3 different internal representations, and the assembler decides (based on the size of the argument) which representation to use in each case.

8.7.1. Instruction Format 4 - 4

4 bit operation code, 4 bit argument.

This format is used for the instructions LP k, SP k, and STACK k, when the argument, k, interpreted as a nonnegative integer, is less than 16.

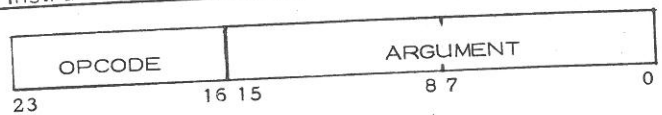
8.7.2. Instruction Format 6 - 10



6 bit operation code, 10 bit argument

This format is used for instructions which interpret their argument as a signed integer in the interval [-512, 511].

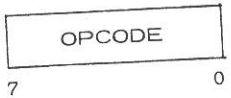
8.7.3. Instruction Format 8 - 16



8 bit operation code, 15 bit argument.

This format is used for instructions with an argument, when the argument cannot fit into format 4 - 4 or 6 - 10.

8.7.4. Instruction Format 8 - 0



8 bit operation code, no argument.

This format is used for instructions which take their arguments (if any) only from the top of the stack, and for the SWITCHON instruction which has a special argument fetch.

8.8. List of OCODE Instructions

In the following list of OCODE instructions, the internal representations are given in decimal. For the 4 bit and 6 bit operation codes the decimal value corresponds to an 8 bit value with 4 or 2 0-bits to the right, thus all decimal values are unique.

Some of the more complicated instructions are described in previous sections, the remaining ones are described briefly in the table, using

"stacktop" to denote the top-of-stack register

"Y" to denote a "processor register" used for intermediate storage

and the routines:

```
let push ( ) be
        $( rv S:= stacktop
           S:= S + 1
        $)
```

```
and pop ( ) be
        $( Y:= stacktop
           S:= S - 1
           stacktop:= rv S
        $)
```

All the unused operation codes (including 236, 237, and 238 which are marked with * in the table, and 240 - 255 which are not in the table) will cause an error if they occur in the code stream at a place where an operation code is expected.

List of OCODE instructions.

mnemonic	Internal representation				short description
	Format 4 – 4	Format 6 – 10	Format 8 – 16	Format 8 – 0	
NOOP	–	–	–	0	// no operation
	–	–	–	1	
RV	–	–	–	2	stacktop := rv stacktop
STORE	–	–	–	3	rv S:= stacktop
FINISH	–	–	–	4	// see section 8.11
TRUE	–	–	–	5	push ( ) stacktop:= true
FALSE	–	–	–	6	push ( ) stacktop:= false
FNRN	–	–	–	7	// see section 7
RTRN	–	–	–	8	// see section 7
NEG	–	–	–	9	stacktop:= – stacktop
NOT	–	–	–	10	stacktop:= ¬ stacktop
	–	–	–	11	
	–	–	–	12	
	–	–	–	13	
	–	–	–	14	
	–	–	–	15	
STIND	–	–	–	16	pop ( ) rv Y:= stacktop pop ( )
GOTO	–	–	–	17	// see section 7
	–	–	–	18	
	–	–	–	19	
LOGAND	–	–	–	20	pop ( ) stacktop:= stacktop ∧ Y
LOGOR	–	–	–	21	pop ( ) stacktop:= stacktop ∨ Y

mnemonic	Internal representation				short description
	Format 4 – 4	Format 6 – 10	Format 8 – 16	Format 8 – 0	
EQV	–	–	–	22	pop ( ) stacktop:= stacktop ≡ Y
NEQV	–	–	–	23	pop ( ) stacktop:= stacktop ≠ Y
SWITCHON	–	–	–	24	// see section 6.5
	–	–	–	25	
	–	–	–	26	
	–	–	–	27	
	–	–	–	28	
	–	–	–	29	
	–	–	–	30	
	–	–	–	31	
PLUS	–	–	–	32	pop ( ) stacktop:= stacktop + Y
MINUS	–	–	–	33	pop ( ) stacktop:= stacktop – Y
EQ	–	–	–	34	pop ( ) stacktop:= stacktop = Y
NE	–	–	–	35	pop ( ) stacktop:= stacktop ≠ Y
LS	–	–	–	36	pop ( ) stacktop:= stacktop < Y
GR	–	–	–	37	pop ( ) stacktop:= stacktop > Y
LE	–	–	–	38	pop ( ) stacktop:= stacktop ≤ Y
GE	–	–	–	39	pop ( ) stacktop:= stacktop ≥ Y
MULT	–	–	–	40	pop ( ) stacktop:= stacktop * Y
DIV	–	–	–	41	pop ( ) stacktop:= stacktop div Y
REM	–	–	–	42	pop ( ) stacktop:= stacktop rem Y

mnemonic	Internal representation				short description
	Format 4 – 4	Format 6 – 10	Format 8 – 16	Format 8 – 0	
	–	–	–	43	
LSHIFT	–	–	–	44	pop ( ) stacktop:= stacktop lshift Y
RSHIFT	–	–	–	45	pop ( ) stacktop:= stacktop rshift Y
	–	–	–	46	
	–	–	–	47	
PLUS10 k	–	128	–	–	stacktop:= stacktop + k
MINUS10 k	–	132	–	–	stacktop:= stacktop – k
EQ10 k	–	136	–	–	stacktop:= stacktop = k
NE10 k	–	140	–	–	stacktop:= stacktop ≠ k
LS10 k	–	144	–	–	stacktop:= stacktop < k
GR10 k	–	148	–	–	stacktop:= stacktop > k
LE10 k	–	152	–	–	stacktop:= stacktop ≤ k
GE10 k	–	156	–	–	stacktop:= stacktop ≥ k
MULT10 k	–	160	–	–	stacktop:= stacktop * k
DIV10 k	–	164	–	–	stacktop:= stacktop div k
REM10 k	–	168	–	–	stacktop:= stacktop rem k
	–	172	–	–	
LSHIFT10 k	–	176	–	–	stacktop:= stacktop lshift k
RSHIFT10 k	–	180	–	–	stacktop:= stacktop rshift k
	–	184	–	–	
	–	188	–	–	
LN k	–	192	224	–	push ( ) stacktop:= k
LLL n	–	196	225	–	push ( ) stacktop:= DB + n
LL n	–	200	226	–	push ( ) stacktop:= rv (DB + n)
SL n	–	204	227	–	rv (DB + n):= stacktop pop ( )

mnemonic	Internal representation				short description
	Format 4 - 4	Format 6 - 10	Format 8 - 16	Format 8 - 0	
RSTACK k	-	208	228	-	S:=P + k // equivalent to //SP k + 1 STACK k + 1
JUMP n	-	212	229	-	PC:= PC + n
JT n	-	216	230	-	pop ( ) if Y < 0 then PC:= PC + n
JF n	-	220	231	-	pop ( ) unless Y < 0 do PC:= PC + n
LP n	48	96	232	-	push ( ) stacktop:= rv (P + n)
LLP n	-	100	233	-	push ( ) stacktop:= P + n
SP n	64	104	234	-	rv (P + n):= stacktop pop ( )
STACK k	80	108	235	-	rv S:= stacktop S:= P + k - 1 stacktop:= rv S
LG n	-	112	* 236	-	push ( ) stacktop:= rv (G + n)
LLG n	-	116	* 237	-	push ( ) stacktop:= G + n
SG n	-	120	* 238	-	rv (G + n):= stacktop pop ( )
RTFNAP k	-	124	239	-	// see section 7

OCODE machine. When an error is detected by the emulator, the type of the error is written in register ET, space for return links is allocated on the stack, and the registers are dumped above this as parameters for the interrupt routine. Lastly a routine call is simulated for the routine pointed to from cell LIMIT - 1. This is supposed to be a general interrupt handling routine.

The following errors are detected by the emulator (other errors could be included, e.g. arithmetic overflow):

- Reading above LIMIT
- Writing above T
- Stack overflow (attempt to set S above T)
- Stack underflow (attempt to set P below P_0, can only occur if the return link is violated)
- Stackframe underflow (attempt to set S below P)
- Bad code (attempt to execute an undefined operation code).

The FINISH instruction is treated in the same way as an error, i.e. it simulates a call of the interrupt routine.

9. The OCODE Emulator

The structure of the OCODE emulator on RIKKE-1 is partly dependent on the structure of the RIKKE-1 hardware so we briefly review some of the facilities of RIKKE-1.

9.1. The RIKKE-1 Hardware

Figure 6 shows the structure of RIKKE-1.

The Control Store is 512 words of 64 bits.

The Microinstruction Sequencer can use a number of sources to select the next address, among which are:

- Current address
- Current address + 1
- Current address - 1
- Current address + data from microinstruction (relative addressing)
- Data from microinstruction (absolute addressing)
- One of 2 return jump stacks, each with 16 words
- Data from Main Data Path.

The address is selected based on the value of a condition (there are a few restrictions on the combinations of address sources in the true and false case in the same microinstruction).

The Main Data Path consists of a Bus Selector and shifting and masking capabilities. It has connected to it a number of registers and functional units, some of which are:

- Arithmetical-Logical Unit (ALU) with 2 inputs of which the one input is one of
- 4 Local Registers (LR[0 : 3]) which can be operated as a stack (but not necessarily)
- Working Registers:
  - WA : 256 registers of 16 bits
  - WB : 256 registers of 16 bits
  - WA (and WB) can be operated as 256 registers or as 16 groups of 16 registers.
  - We only use the latter option.

The Main Store is 32 K words of 16 bits.

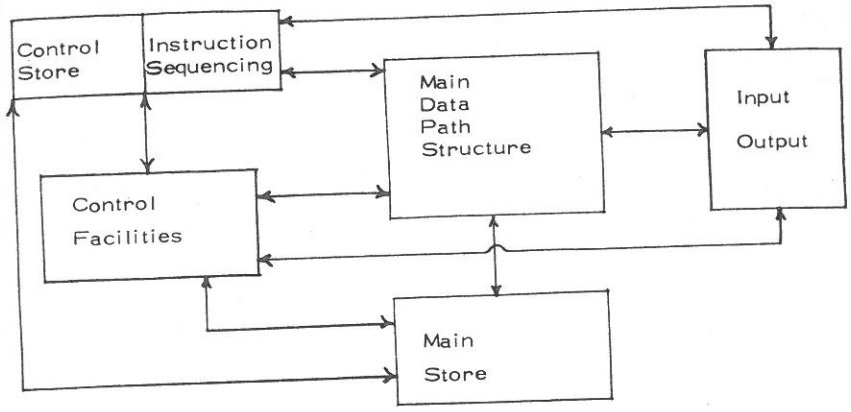


Figure 6. The RIKKE-1 System.

9.2. Resources Used by the OCODE Emulator

The store of the OCODE machine is allocated in the RIKKE-1 Main Store. The registers occupy one group of 16 registers (WA), and the top element of the stack is located in LR [0], but is written down in the store when the emulator gives up control (e.g. in order to call the I/O nucleus).

The emulator itself uses 4 groups of 16 registers (WB) for decoding purposes and 1 group to hold the constants -1 through 14. In addition some of the Control Facilities and one of the return jump stacks are used, and the code of the emulator occupies about 300 words of Control Store.

9.3. Structure of the Emulator

Using the return jump stack it is easy to implement subroutines in RIKKE-1 microcode, and the code emulating each OCODE instruction is actually a subroutine called from the main loop of the emulator. Among other important subroutines are the routines for reading from the code stream and administrating the (pseudo-) byte-addressing, routines for (10-bit and 16-bit) argument fetch, and the following routines:

READ , reading from main store
WRITE , writing to main store
PUSH , pushing the stack
POP , popping the stack

READ and WRITE map logical to physical addresses and check the validity of the addresses.

PUSH and POP check for stack-overflow and stackframe-underflow, respectively.

The emulator interface to the environment was described in section 6 and figure 3.

Figure 7 shows the structure of the main loop of the emulator. It shows that after the instruction fetch the instructions are split up

into 16 groups. This is done by using the first 4 bits of the operation code as index in a table (allocated in a group of WB registers) of entry-points in the emulator.

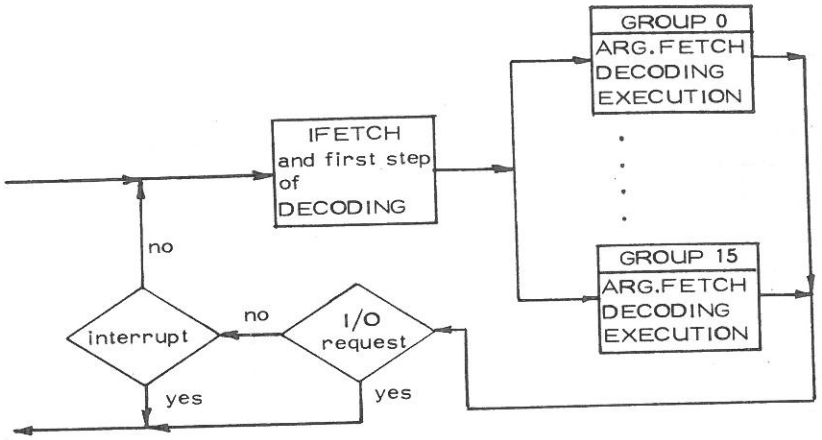


Figure 7. Main Loop of the OCODE Emulator.

Each of the groups then performs an argument fetch before the decoding is finished, and finally execution of the instruction is performed. Since some instructions occur in different versions with different argument fetch the decoding and execution may be common to several groups.

Some of the groups are decoded using a table, as in the first step, but now the last 4 bits of the operation code is used as index in the table (note that the 2 middle bits in the 6-bit operation codes are thus decoded twice, but this does not cost us anything). Other groups are decoded bit-by-bit (by an if - then - else construction). This is rather expensive in control store, so it is only done in the cases where something useful can be done in parallel with the decoding.

As an example figure 8 shows how the arithmetic instructions without arguments (GROUP 2) and those with a 10-bit argument (GROUP 8, 9, 10, 11) are decoded using a common table.

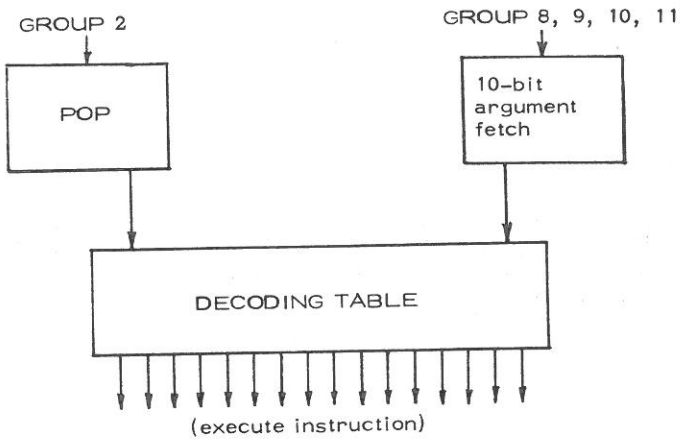


Figure 8. Argument fetch and decoding of arithmetic instructions.

This means that the operation codes must be selected so that they allow for this common decoding. Thus we have slightly violated the decision that the design of the OCODE machine should be machine-independent (at least it is dependent on a decoding strategy which was selected as the most efficient for implementation on RIKKE-1).

10. The OCODE Assembler

The job of the OCODE assembler is to take a segment of symbolic O-CODE and to produce a code segment, a static data segment, a global initialization part, and linkage information for the loader. However we will not go into details about the loader format but concentrate on the optimizations done by the assembler.

For instructions with different argument sizes the assembler will decide the argument size to use based on the value of the argument. In the case of a forward jump the value of the argument is not known so the argument size will always be 16 bits.

With the arithmetic instructions with an argument the situation is a little more complicated. In order to be able to construct these instructions the assembler must always remember the previous instruction. If the assembler finds an arithmetical, comparison or shift instruction in the symbolic OCODE stream and the previous instruction was a LN with a 10-bit argument, then the LN instruction is removed, and instead the arithmetical, comparison, or shift instruction is inserted in the version with a 10-bit argument. Other wise it is inserted in the version without argument. If the operation is symmetric, the assembler will look at the 2 instructions previously inserted and if possible make the optimization.

The BCPL compiler produces some OCODE instructions that are superfluous. These will be removed by our OCODE assembler. Examples are:

1) STACK i
   STACK j
   The first instruction is superfluous, so we get
   STACK j

2) JUMP Ln_1
   // anything except LAB or ENTRY
   LAB Ln_2
   Anything between the unconditional branch instruction and

the first following LAB or ENTRY will never be executed and can be removed, so we get

JUMP Ln_1

LAB Ln_2

The same optimization is made for RTRN, FNRN, GOTO and FINISH

3) STORE

This instruction is unnecessary in our implementation because the stack is always pushed before an item to be loaded onto the stack is read from the store. Thus even if we want to load the top element onto the stack we are always sure that it is in the store when it is read since pushing the stack implies writing the stacktop register down into store.

There are a few other problems that are solved by the OCODE assembler:

1) Our procedure call and return will not work correctly if a function is called as a routine because the FNRN will leave a result on the stacktop. Therefore we must reset the stacktop after returning, so

```
RTAP k
```

becomes

```
RTFNAP k
STACK k
```

(where the STACK k instruction is placed in the next word, so a NOOP is inserted if the RTFNAP k ends in the middle of a word.)

In the same way

```
FNAP k
```

becomes

```
RTFNAP k
STACK k + 1
```

although the interpretation of the result when a routine is called as a function is meaningless.

2) We assume that the RES instruction always jumps to a RSTACK instruction, and thus we can substitute

RES Ln

by

JUMP Ln

since the RSTACK instruction saves the stacktop. Thus we do not need an A register for saving the stacktop.

11. Conclusions

When we started this project we had no experience in this sort of work, and moreover we did not know very much about either the BCPL language or the RIKKE-1 machine (which had just been designed, and the first description [5] was being prepared).

It turned out to be a great advantage in finishing and debugging the OCODE emulator that we were at the same time (autumn 1973 and spring 1974) testing the RIKKE-1 hardware, and we had a very fruitful cooperation with the people writing the RIKKE-1 microassembler and simulator [7] without which it would have been a very hard job to test the emulator.

But still the main problem was deciding the internal representation for the code and the organization of the OCODE machine. The amount of data analyzed was no doubt too small, but on the other hand it has turned out that our results are similar to those obtained at Oxford [11].

It is planned to include in the OCODE assembler a facility to make a (static) analysis of the code being assembled, and it should be easy to add a facility for dynamic analysis into the OCODE emulator.

We now feel (based on the results from Oxford) that there would probably be an advantage in having more instructions with the 4 bit opcode - 4 bit argument format (RTFNAP, LN, PLUS), but there is no room for these in our instruction set without making other changes (probably changing the 6 - 10 format to 8 - 8 except for the instructions acessing the global vector). There is also the possibility that a 3 - 5 or 5 - 3 format would give an advantage over 4 - 4, but this must await further analysis.

About mid-July 1974 we succeeded for the first time bootstrapping the OCODE machine onto RIKKE-1 with a "mini-system" (written in BCPL) and loading and executing a "user program" (a very small program computing factorials). Since then a few errors have been found and corrected at all levels (RIKKE-1 hardware, OCODE e-

mulator, "mini-system", OCODE assembler, RIKKE-1 microassembler, RIKKE-1 simulator).

The "mini-system" has been further developed, and the BCPL compiler and the OCODE assembler have been transferred to the OCODE machine on RIKKE-1, so that today we can edit, compile, assemble, load,and execute BCPL programs on RIKKE-1.

References

[1] M. Richards: The BCPL Reference Manual.
Technical Memorandum no. 69/1 - 2.
The Computer Laboratory, Corn Exchange Street, Cambridge.

[2] M. Richards: The Portability of the BCPL Compiler.
Software - Practice and Experience, Vol 1 (1971).
pp 135 - 146.

[3] M. Richards: BCPL, A Tool for Compiler Writing and System Programming.
AFIPS Spring Joint Computer Conference 1969
pp. 557 - 566.

[4] M. Richards: INTCODE - An Interpretive Machine.
The Computer Laboratory, Corn Exchange Street, Cambridge.

[5] Bruce D. Shriver: A Description of the MATHILDA System.
DAIMI PB-13, Department of Computer Science, University of Aarhus, Denmark, April 1973.

[6] E. Kressel, J. Staunstrup: The RIKKE-1 Reference Manual.
DAIMI MD-7, Department of Computer Science, University of Aarhus, Denmark, April 1974.

[7] E. Lynning, E. Kressel, H.O.S. Andersen, I.H. Sørensen:
A Users Manual for the Simulated RIKKE - MATHILDA System on the CDC6400.
DAIMI MD-12, Department of Computer Science, University of Aarhus, Denmark. December 1974.

[8] P. Kornerup, Bruce D. Shriver: An Overview of the MATHILDA System.
DAIMI PB-34, Department of Computer Science, University of Aarhus, Denmark, August 1974.

[9] C. Strachey, J. Stoy: The Text of OS Pub.
Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1972.

[10] Robert F. Rosin: Proposal for a Nucleus I/O System.
DAIMI PB-23, Department of Computer Science, University of Aarhus, Denmark, January 1974.

[11] N. Derrett: Design of Computing Mechanisms.
University of Oxford, May 1974.