In Search of an Optimal Machine Architecture for BCPL

79-1

by

R.K. Agarwal and S.T. Chanson

Dept. of Computer Science
University of British Columbia

January 1979.

Technical Report 79-1

In Search of an Optimal Machine Architecture for BCPL

by

R.K. Agarwal and S.T. Chanson

Dept. of Computer Science
University of British Columbia

January 1979.

Technical Report 79-1

Table of Contents

ABSTRACT

This paper investigates the problem of generating optimal space-efficient code for the language BCPL. Designing such a code was seen to be a two-phase process. The first phase was to describe an internal representation scheme for BCPL programs which preserved those program features which are salient to translation and at the same time minimize the number of instructions generated. The second phase consisted of the realization of the internal representation as an actual machine taking into account the usage frequencies of instructions and other real world constraints such as word size and addressing space. The intermediate code, called ICE and an encoding scheme (known as ESO, standing for encoding scheme 0) are described. ICE/ESO is seen to reduce code size by an average of about 32% compared to BCODE which is a realization of OCODE, the intermediate language currently used in BCPL program translation.

0. The Meaning of Optimality

One often speaks of the desire to produce "efficient programs". Apart from the criterion of correctness, program efficiency is usually measured in terms of time (the number of CPU cycles used), and space (the total amount of storage used by the process). In the present context, the optimality of generated code will be measured only on the basis of space efficiency. Such a stance is fairly popular and is normally justified by noting that memory is a more critical resource than CPU cycles for, although both are becoming less expensive, word size limitations restrict the convenient access of large areas of store.[1] Since we are discussing the generation of optimal code by an automatic translator it is reasonable to state that space efficiency will lead to time efficiency. This is because the increase of space efficiency by the restructuring of programs (which might present a tradeoff between program speed and size) is not considered here; the elementary optimizations which are discussed in this paper are shown not to degrade the generated code's speed characteristic.

In this paper, the problem of generating optimal space efficient code is investigated for the language BCPL (Basic Combined Programming Language). BCPL [R1] is a typeless language which is particularly suited for the writing of systems programs. It is a good choice for the present study not only because it is a simple language which has been used in practice, but also because most BCPL compiler implementations presently generate an intermediate code called OCODE [R2]. The existence of OCODE facilitates the evaluation of the relative merits of the code developed here, called TCE. Designing such a code was seen to be a two-phase process: the first phase was to describe an internal representation scheme for BCPL programs which preserved those program features which are salient to translation; the second consisted of the realization of the internal representation as an actual machine. The realization would produce an instruction set encoding based on usage frequencies of instructions and other real world constraints on machines such as word size and addressing space. Roughly speaking, therefore, the aim of the former phase was to minimize the number of instructions generated, whereas the latter would ensure their optimal encoding on a target machine.

In some sense, the answer to phase one of the problem is evident: we can make the intuitively reasonable assumption that the optimal representation of a BCPL program is the program itself. This assumes that the algorithm expressed by a BCPL program cannot be expressed more succintly. A common data structure used to represent a program is the tree. A tree has several disadvantages when viewed as an intermediate code for

----

(1) See [T] for a fuller discussion of other considerations.

BCPL.[2] Since a tree is a structure in two-dimensions (sequencing and nesting) it is difficult to realize in terms of the sequential machine architectures prevalent today.[3] If translation were being considered to the native code of some existing machine a tree may be a more reasonable choice. However the task of such a translator could be greatly simplified if the intermediate code were itself one-dimensional. Such a one dimensional code should ideally have the property that its instructions can be expanded into instructions for the target machine in a context-free way (that is, by treating commands in the intermediate code as macros defined in terms of the target machine's instructions). It is the design of such a code that will be discussed in section 2.

The advantages of two-dimensional representation should not be overlooked, however: a tree representation would be ideal in that it would closely reflect program structure and at once remove all unneccessary information such as noise words, and most names. But since we are looking for an intermediate code which can be viewed as an actual machine with a structure common to those in existence today, the one-dimensional alternative will be the only one developed here.

As a corrollary to the previous assumption that the BCPL program being translated has been optimally represented, we have that the introduction of such programming artifices as index registers are unnecessary. This is because an index register may typically be useful in reducing code size if it can be loaded with the address of a frequently referenced vector (say). If the high level language does not allow for the explicit loading of the index register with the vector's starting address, some form of data flow analysis is required for its optimal usage. We have specifically precluded such analysis.

1. A Method for Producing Good Machine Code from BCPL

In this section the design phase of the intermediate code ICE is discussed. As noted previously, two major objectives are to be met: ICE should be a language that is easily encodable as the instruction set for some real machine such that the encoding is efficient and it should be amenable to translation into the host language of some other machine.

----

(2) In an environment where complete syntactic information regarding a program is required at execution time (as in, say, an interactive debugging system) a tree is likely to be the representation scheme of choice.

(3) What is required is a machine capable of executing directly some LISP type language. Even here, the linking scheme would have to be modified to reduce space wastage due to linkage fields.

Although the machines underlying OCODE and ICE are the same, ICE manipulates data objects differently. In BCPL the basic data object is always the <u>word</u>. A word is of unrestricted size and form provided that it can be used to store any address and that consecutive words are numbered consecutively. OCODE manipulates data objects by pushing them onto the runtime stack, then applying the required operator to them. If OCODE is to be viewed as a real machine, the need to explicitly stack all data objects is wasteful in both time and space since the push operation requires a separate instruction. If OCODE is to be translated into a different machine's language, some fairly intricate pattern matching mechanisms are required if reasonable object code is to be generated. This is because a BCPL command such as

```
a := b+c                                                   (E1)
```

will translate into the OCODE commands

```
L b     Push b onto stack
L c     Push c onto stack
PLUS    Replace top two elements of stack with sum
S a     Store the top of stack in location a.
```

Now consider a fairly typical multiple register machine architecture with instructions of the form

```
<op> <reg> <addr>
```

where <op> is the diadic operator applied to the contents of the register <reg> and the memory location <addr> in the form <reg> := <reg> <op> <addr>. To generate the expected

```
LOAD  reg1,B
ADD  reg1,C
STORE  reg1,A
```

sequence for this machine from the above OCODE segment, the 'L c; PLUS' sequence has to be recognized. This is only a simple instance of the pattern matching capabilities needed. A BCPL command such as

```
a!b := c                                                   (E2)
```

may be implemented on many machines as a single instruction; OCODE generates the sequence

```
L c
L a
L b
PLUS    Calculate address of a!b
STIND   Store c in address at top of stack            .
```

Some mechanism has to exist to detect this pattern to generate optimal code from OCODE. Unfortunately, such translation schemes are not straightforward to implement.

ICE(*) views data objects to be of two basic types: cellular and complex. Any data object which can be directly stored in a word without the need for further evaluation is cellular. Complex objects are those which can be stored in a word only if evaluated. Hence all non-trivial expressions are complex. Generally, ICE allows the direct specification of all cellular objects as instruction operands; the runtime stack is used only to store the intermediate results produced in the evaluation of complex objects. A disadvantage of such an intermediate representation is that the number of instructions in the repertoire increases enormously. Whereas OCODE has exactly one operator specifying an operation, ICE in principle requires 2**n operator variants to specify all the cellular and complex operand n-permutations for an n-adic operator. To linearize this exponential growth, a realistic compromise has been made: instead of having instruction variants allowing any operator type permutation, the only ones ICE features from the 2**n possibilities are those whose rightmost operands are all cellular. The remaining operands, be they cellular or complex, are all fetched from the stack. Thus, for an n-adic operator ICE has a zero operand variant (where all operands are on the stack) to an n-operand variant (where all operands are cellular and thus directly specifiable). This produces a total of n+1 instruction variants. By judicious choice of the order of operand specification, such a restricted representation produces almost as good code as would be in the general case. There are several reasons for this. In some cases (e.g., commutative operators) the linearized set of operands are as general as in the exponential case, since the order of operands can be reversed. In some other cases certain operands, such as the selector field in a select expression (see MOVESELECT in A.1.4) can only be cellular objects and so invalidate some of the possible variants in the exponential set. In yet other cases, an operator as well as its inverse is available (e.g., GT and LS for the "greater than" and "less than" relations). This allows BCPL code sequences such as

```
... A<(B+C) ...
```

to be transformed into

```
... (B+C)>A ...
```

with a corresponding increase in code density, for reasons noted below. The existence of an inverse for an operator makes it, in essence, commutative. One should note that the specific order

----

(*) Only the design principles of ICE are discussed here. A complete description of the ICE instruction repertoire is to be found in appendix A.4.

in which operands are allowed to appear is relatively unimportant, for we could specify (for example) that cellular operands could only appear in the leftmost positions, instead of the rightmost. If the operands were themselves reversed the leftmost scheme would be equivalent to the previous one (modulo the notation used). The important point in the linearization scheme is that relatively little representational power is lost by it use.

To illustrate the possible instruction variants for an operator let us consider the operation of division, which takes two arguments (shown as 'x' and 'y' below). If both 'x' and 'y' are cellular, the instruction generated is

```
DIV  2 x y
```

If 'x' is complex, the correct ICE instruction is

```
DIV  1 y
```

where the value of 'x' is now fetched from the top of stack[5]. If both 'x' and 'y' are complex or if 'x' is cellular but 'y' is complex, the instruction to be generated is

```
DIV  0
```

where 'y' is at the top of stack and 'x' at the location immediately below the top.

Note here that we are forced to push 'x' on to the stack if 'y' is not cellular. This is a consequence of the linearization scheme outlined above. However, if the operator is commutative, then the operand order can be reversed to allow the cellular 'x' operand to be directly specified. Since BCPL specifically leaves the order of sub-expression evaluation undefined, operators which are commutative in ordinary mathematics can (and must) be considered commutative by a BCPL to ICE translator. Note that the commuting of operator order wherever advantageous is not an option -- it is a part of the definition of ICE. Similarly, maximizing the number of operands to an instruction is also not optional. Hence if two cellular objects 'a' and 'b' are to be added, the correct ICE instruction is

```
ADD  2 a b
```

An instruction sequence such as

```
PUSH  1 a
PUSH  1 b
ADD 0
```

----

[5] Unless otherwise specified, fetching an item from the stack always implies its deletion.

is incorrect. Furthermore, the BCPL fragment

```
C+A*B
```

is correctly translated into ICE as

```
MULT  2 A B
ADD  1 C
```

Note the transformation of 'C+A*B' into 'A*B+C.'

The advantages of such an intermediate code will now be outlined. If one wishes to reduce the instruction count, it is clear that elimination of unnecessary PUSH instructions (L in OCODE) helps. Although the number of bits required to represent an instruction code has now increased, the overall number of bits needed to represent a program (in comparision to similar encodings for OCODE) is nonetheless reduced.

In comparision to OCODE, ICE is also better suited to translation into the host language for machines which presently exist. This is because most machines allow at least some of the operands of an operator to be explicitly specified. Continuing with our previous examples we note that the ICE codes generated for (E1), if both 'b' and 'c' are cellular is

```
ADD  2 b c
MOVE  1 a
```

More importantly, we note that the definition of ICE requires that the

```
ADD  2 b c
```

be generated, and not (say)

```
PUSH  1 b
ADD  1 c
```

This implies that if a PUSH command is encountered in translating ICE to some other language we are guaranteed that the PUSH is indeed necessary[6]. The major advantage of such a property is that ICE instructions can be transformed into the language of most other machines in a context-free way, and still

----

(6) It is assumed that instructions on real machines allow operands to be specified only in the order that ICE allows; that is, instructions such as

```
DIVIDE  <addr>,<reg>
```

meaning <reg> := <addr>/<reg>, where <addr> is a memory location and <reg> a register, are not allowed. Empirical evidence shows this to be a reasonable assumption.

produce close to optimal code. Such is not the case for OCCDE. A good quality code generator can thus be produced for most target machines by treating ICE commands simply as macros whose expansion is defined using the macro assembler which is usually provided by the machine's vendor. Recognizing that the format of macros accepted by macro assemblers varies considerably, an exact external form for ICE commands has not been defined. For the purposes of description, appendix A.1 does indeed present a representation scheme; the current implementation of the BCPI to ICE translator allows the appearance of ICE commands to be modified readily however. Indeed, it is quite reasonable to perform the "macro expansion" referred to earlier within the routine which emits the ICE code (in the BCPL to ICE translator).

ICE is essentially a generalization cf OCODE. Fixed sequences of commands which frequently occur in OCODE have been combined into one ICE instruction. The coalescing of instructions has not been done in an arbitrary way. The general rule followed has been that every BCPL operator has been assigned a corresponding ICE instruction. In practice such an architecture resembles those of real machines quite closely. In particular, the scheme used to linearize the number of variants of an instruction seems to be employed by real machines also. It should be noted that some machines allow for a greater degree of compression than ICE. For example, the BCPL command of (E1)

    a := b+c   (E1)

can be translated into a single instruction on some existing machines. ICE can, at best, produce

```
ADD  2 b c
MOVE  1 a
```
.

This is because machines which allow (E1) to be expressed as one command are combining the distinct BCPL operations of addition and assignment. ICE does not include such combinations in its instruction set.

From the viewpoint of Flynn's work [F] on the evaluation of machine architectures, ICE's superiority over OCODE results from the reduction of the need for M-type instructions [7] to the point of absolute necessity. ICE also unites several distinct OCODE commands as single commands with variants. For example, the ICE equivalent of the OCODE STIND operation is MOVE 0. Similarly, the operations JUMP and GOTO in OCODE are simply variants of the ICE JUMP command.

----

[7] In Flynn's terminology, M-type instructions are those which move data from one space in the memory hierarchy (e.g., registers) to another (e.g., main memory).

2. An Encoding Scheme for ICE

From the discussion of the previous section it can be inferred that, under the constraints specified, ICE does indeed minimize the number of instructions generated from a program written in BCPL. If we are to view ICE as the instruction set for a real machine, it is not clear however that ICE expresses programs in fewer bits than an alternate scheme such as ECODE (see [M]) for although the number of instructions generated has decreased, the number of bits required to represent them has increased (due to their greater complexity). Indeed, the problem of encoding an instruction set optimally is largely an exercise in the statistical measure of the frequency of instruction usage; an encoding for ICE which is optimal for all conceivable BCPL programs is thus not possible in principle. Here we present a reasonable encoding based upon some measurements of a large sample of BCPL programs and the constraints on encoding schemes which real hardware inevitably provides.

An initial decision was made to have a machine with a word length of sixteen bits. This was done largely because an encoding was being sought which would be suitable for use in a minicomputer environment. From experience with machines of various word sizes, sixteen bit words were also felt to present a reasonable tradeoff point between the information storage capacity of a word and the memory wastage associated with the use of large word sizes. This choice of word size has one disadvantage: floating point operators are unavailable since real values are not conveniently stored in sixteen bits. Since real arithmetic is not a feature of standard BCPL[8], and since such data manipulation is uncommon in BCPL, the lack of this capability was not felt to be serious. Lastly, choosing a sixteen bit word was advantageous since other object machines for BCPL have been devised using the same word size. This allows a method for measuring (by comparision) the relative space efficiency of a particular ICE encoding scheme. Since ICE instructions consist of an operator followed by zero or more operands, the encoding problem can be divided into the problems of encoding the operator and encoding the operands. These two encodings cannot be performed entirely independently however since they both have to meld together well in the environment of the underlying word.

Including all possible command variants, ICE consists of a total of 256 operators. As will be seen shortly, a "no operation" instruction is also needed. Since PUSH 0 is a one byte operator which does nothing (it pops the top of stack, then pushes it back on) it will be used as the no op. The total of 256 operators can be represented by a single byte (8-bits) of

----

[8] ICE is sufficiently powerful to accomodate BCPL-V, an extension of BCPL which permits real arithmetic.

information. The actual mapping of operators to bit patterns is left unspecified; this is done so that the implementor might make best use of any special characteristics of the machine on which ICE is emulated or simulated. Note that the 256 operators include all those which operate on real values; for the present case, there will therefore be fewer than 256 operators. A complete byte is nonetheless assigned to the operator field to simplfy the hardware decoding logic (or microcode). Such considerations will affect the form of the operand field also. In particular, it will be assumed that the basic (indivisible) size of any datum is eight bits. Hence, in this encoding scheme, instructions and data will always be in multiples of bytes. As may be evident, the imposition of such a constraint reduces encoding efficiency.

The representation of operands under this encoding scheme (called ES0) will now be discussed. An operand in BCPL consists of two parts: an addressing mode (admode), and a value for the particular addressing mode. For example, an operand referencing global cell 20 is in the global admode with a value of 20. There are four basic addressing modes. They are <u>absolute</u>, <u>global</u>, <u>local</u>, <u>and</u> <u>relative</u>. In addition, for each admode, ICE has the ability to specify whether the addressing is direct or indirect. Hence three bits are needed to represent the admode. If operands are to be stored in a single byte there are five bits left for the value field. Since five bits are insufficient to represent all value fields, operands are allowed to be either long (three bytes) or short (one byte). One bit is required to represent this length attribute and hence the space for the value field of a short operand is reduced to four bits. Although this may seem restrictive at first, Table A.2.3. in appendix A.2. shows that an average of 71 percent of operands fit into four bits, when represented in two's-complement notation. If an operand is long, it must occupy three bytes. This is because a two byte operand leaves only twelve bits of space for the value field; more are needed to represent addresses in any medium sized program.

The operator and operand encoding schemes having been described, they can now be combined to represent ICE instructions. The ICE/ES0 machine represents instructions by specifying the operator (in one byte) and following it with the (implied) number of operand fields required. Since the basic addressable unit at the BCPL level is a word, a problem arises whenever code is addressed (e.g., by a JUMP) which is at an odd numbered byte (since there exists no corresponding word address). Two methods are apparent which overcome the problem: since all addressing at the BCPL level is accomplished via an indirect branch through a cell, a byte address can be stored within the cell. This limits the word addressing space to fifteen bits on a sixteen bit machine (the freed bit being used to index the byte). The other alternative is to generate a one byte "no operation" command whenever necessary preceeding a label declaration within the code body (i.e., declarations using the LAB command). ICE/ESO adopts the latter solution.

Measuring the space efficiency of any machine architecture is not a straightforward task, for the results are affected by the sampling of programs studied. Both the style of programming present in the sample, as well as the applications being sampled are factors which influence the outcome of the analyses. Nonetheless, Table A.2.4. in appendix A.2. presents a comparision between ICE/ESO and SLIM, a machine devised specially for representing BCPL programs (see [Fox] for a description of the SLIM machine). Code generation for a small sample of programs shows that SLIM compares favourably with EM1, an experimental machine designed by Tanenbaum which attempts to minimize the object code size of programs written in SAI, a language with a BCPL flavour (see [T] and [Fox]). As a further indication of the compactness of ICE/ESO, we note from Table A.2.4 that the average SLIM to ICE/ESO code ratio is 1.18 whereas Fox (see [Fox]) reports BCODE to SLIM code ratio to be 1.12. This means that BCODE, which is a realization of OCODE (the intermediate language currently used in BCPL program translation) takes up an average of 32 percent more space as compared to ICE/ESO.

3. Conclusions

Our objective has been to find a space-efficient way of encoding BCPL programs. A two-phase method has been used in developing this code: the first phase produced an instruction set which minimized the number of instructions generated; in the second phase a space-efficient encoding for this instruction set was derived. The translator section of a BCPL compiler has been modified to generate ICE code and to collect code size statistics. Assuming that no optimizations other than the reordering of operands to commutative operators and constant folding are allowed, ICE minimizes the number of generated instructions, within the constraints imposed by the linearization scheme.

As pointed out in section 2, it is meaningless to talk of an encoding for ICE which is optimal for all programs. The encoding scheme presented, ESO, emerged from an attempt to satisfy the conflicting objectives of uniformity and space efficiency. As an example, note that all operators (including their variants) are entirely encoded in the first byte of an instruction. However, not all operators are used with equal frequency; for example, the ADD operator in any of its variants is far more prevalent than REM. Hence, with a sufficiently tricky encoding, some of the most frequently used operators along with one operand could possibly be represented in a single byte. The (acceptable) tradeoff would be that some of the rarely used operators would now require more than one byte for their representation. Tanenbaum adopts such an approach in [T]. Further gains could likely be made if one were not constrained by byte boundaries when devising an encoding. ESO was designed for such a restricted environment however to conform with word formats prevalent on current machines, and to simplify the decoding logic (or microcode) used in implementing it.

As a final remark, note that an instruction

```
<op> <var(1)> ... <var(n)>
```

is equivalent to

```
PUSH <var(1)>
<op> <var(2)> ... <var(n)>
```

assuming that there is no stack overflow as a result of the PUSH. This provides a way of reducing the number of operand variants, and hence the number of bits required to encode an operator. Note that the above expansion degenerates to OCODE if it is applied recursively to the point that all operators are seen only in their zero-operand variant.

Acknowledgements

We would like to acknowledge the help received from the following people in the course of this project in the form of criticisms, suggestions, and insights from various discussions: Mark Fox, Stephen Ma, and Dave Mielke. The project was supported by the National Sciences and Engineering Research Council of Canada under Grant No. A3554, by the Youth Employment Program for Universities (YEPU) and the Department of Computer Science at UBC.

References

[F] Flynn, M.J., Computer Organization and Architecture, Lecture notes for the advanced course on operating systems. Munich, 1977.

[Fox] Fox, M., Machine Architecture and the Programming Language BCPL. M.Sc. Thesis, Dept. of Computer Science, UBC 1978.

[K] Knuth, D.E., "An Empirical Study of FORTRAN Programs", Software: Practice and Experience Vol. 1, 1971, pages 261 to 301.

[M] Ma, S. and R.K. Agarwal, The BCODE System, Technical Manual TM-24,Dept. Of Computer Science, UBC, April 1978.

[R1] Richards, M., The BCPL Programming Manual revised by J.E.L. Peck and V.S. Manis, Technical Manual 75-10, Department of Computer Science, UBC 1977.

[R2] Richards, M., "The Portability of the BCPL Compiler", Software: Practice and Experience, Volume 1, 1971. Pp135-146.

[T] Tanenbaum, A.S., "Implications of Structured Programming for Machine Architecture", CACM, Vol. 21, Number 3, March 1978. Pp237-246.

A.1. A Description of ICE

The intermediate code ICE is described here in a reference manual format. As discussed in section 1, the properties of the ICE code generated are an integral part of ICE, in addition to the instruction repertoire itself. In particular, recall that one of the major advantages of ICE over OCODE is that one is guaranteed that each ICE operator will maximize the number of arguments passed to it. For example, the BCPL command

```
A := B/C
```

could be translated into ICE as follows

```
PUSH 1 B
PUSH 1 C
DIV 0
PUSH 1 @A
MOVE 0
```

However, the correct ICE code (by the maximization of operands property) is

```
DIV 2 B C
MOVE 1 @A
```

Furthermore, since the order of evaluation of operands is unspecified in BCPL, the correct ICE code for the BCPL code fragment

```
A := B+(C-2)
```

is

```
SUBTRACT 2 C 2
ADD 1 B
MOVE 1 @A
```

Since addition is commutative, the expression 'B+(C-2)' is transformed into '(C-2)+B' which allows the cellular[9] object 'B' to be specified directly as an instruction operand. It should be noted though that if ICE is used as an intermediate code for a language which defines evaluation order, operands cannot be validly commuted.

As presently generated, the external representation of ICE instructions follow a very rigid format. The general form of an ICE instruction is

<op><var><arg(1)> ... <arg(<var>)>

where <op> is the instruction name

<var> is the instruction variant (see below)

<arg(n)> is the _n_th argument to the instruction (see below)

In the descriptions below, instructions are classified by the number of arguments they accept. For example, ADD is a diadic operator since it accepts two arguments. In general, for an _n_-adic operator, the instruction variant number (<var>) specifies the number of arguments which occur directly after the

----

(9) A data object is cellular if it can be directly specified as an argument to an instruction. All words are cellular objects. Non-cellular objects are termed complex. Most expressions are complex. An exception is !A which, if evaluated in Lmode, is cellular.

instruction code. The n-<var> operands which the instruction still needs are fetched from the n-<var> topmost locations of the runtime stack (where fetching an object from the stack implies its deletion). After performing the operation specified by <op> the result, if any, is pushed onto the stack. Since operands to an instruction are stacked only if they are not cellular themselves, for an n-adic operator one requires 2**n instruction variants in general to allow cellular objects to be always specified directly as an instruction's operand. ICE allows for only n+1 variants by allowing only the rightmost cellular operand fields to an instruction to be specified directly. To illustrate by example, consider the ICE instruction 'MOVEBYTE 3 c b a', whose effect in terms of BCPL is 'a%b := c'. The four variants of MOVEBYTE are given below in tabular form along with the conditions under which each is generated. In the table, S is a variable which point to the topmost used element of the stack.

MOVEBYTE 3 c b a	c,b,a are all cellular objects. Effect is a%b := c
MOVEBYTE 2 b a	c is a complex object. Effect is a%b := !S S -:= 1
MOVEBYTE 1 a	b is a complex object; c is complex or cellular. Effect is a%(!S) := !(S-1) S -:= 2
MOVEBYTE 0	a is complex; b,c can be any combination of cellular or complex objects provided both are not cellular. Effect is (!S)%(!(S-1)) := !(S-2) S -:= 2

Note the order in which the elements are fetched from the stack; this scheme is used uniformly by all instructions.

Arguments (cf. <arg(n)> in the general instruction format) can be one of two general types. The most common is the <admode,value> pair. This is used to specify simple objects; for example global cell 100 would be represented as 'G 100'. The valid admode types are

Code	BCPL Equivalent
C	character constant
F	floating point constant
G	global cell reference

```
L       label reference
N       numeric constant
P       local (dynamic) variable
R       field selector constant
S       string constant reference
X       external label reference
```

Each of these codes can be modified by the indirection operator "I". Hence PUSH 1 IL L0002 means push onto the stack the contents of the cell labelled L0002. The value field is an integer, a character, a floating point constant, a string constant, or a label. An example of a value is the "100" in 'G 100'.

Many instructions do not require a generalized <admode,value> notation to specify operands. In general these operands are always constants, as in the constant string argument to the SECTION command, or the constant label argument to the RESULTEXIT command.

In the descriptions which follow, the instructions are listed in order of the number of operands each accepts. From the point of view of a translator, this has the advantage that groups of instructions which have similar argument types can be processed by the same translator segment. In the tables below C refers to the program counter, P points to the stack frame pointer, and S to the top of stack (i.e., the last used cell on the stack). Where required, the instruction is followed by a description of its effect in terms of BCPL commands. Also note that not all of the instructions described are those which generate actual object code; many, such as the NILSTATE operator, are directives required either during assembly, or during code generation. These types of operators are followed by an asterisk (*) below.

A.1.1. Niladic Operators

Niladic operators take no arguments. They are

```
RV :   Indirection
     !S := !(!S)

RTNRTRN :   Return from a routine invocation
     S := P-1 ||  restore stack pointer
     C := P!1 ||  restore program counter
     P := P!0 ||  restore frame pointer

FCNRTRN :   Return from a function invocation
     C := P!1 ||  restore program counter
     P := P!0 ||  restore frame pointer
     !P := !S ||  place result on top of caller's stack
     S := P   ||  restore stack pointer
```

FINISH : Terminate program execution unconditionally

SAVEMARKER : Allocate space on stack for saving program counter and stack pointer
S +:= 2

FALSE : Push **false** onto stack

TRUE : Push **true** onto stack

NILSTATE (*) : Code generator directive forcing generation of code which ensures that the contents of all memory cells in the run-time environment are valid.

END (*) : Code generator directive signifying the end of a compilation section.

STARTBLOCK (*) : Signifies the start of a BCPL block.

A.1.2. Monadic Operators

Monadic operators in ICE are of two types: those which take an <admode,value> argument and those which take a constant argument. The notation used in describing them is

<op> <arg>

where <op> is the instruction being applied to <arg>. If <arg> is denoted by "var", it means that the argument is of the <admode,value> type. Only such instructions are allowed to take their operands from the stack (thus producing the n+1 instruction variants discussed earlier). If not denoted by "var", the argument to the instruction can only be a constant (the type of the constant being denoted by the single letter argument codes listed earlier).

PUSH var : Push var onto the stack.

NEG var : Push -var onto the stack.

FNEG var : Push #-var onto the stack.

NOT var : Push ¬var onto the stack.

ABS var : Push **abs** _var_ onto the stack.

FABS var : Push **fabs** _var_ onto the stack.

FIX var : Push **fix** _var_ onto the stack.

FLOAT var : Push **float** _var_ onto the stack.

STACK N : Set the stack pointer
    S := N

RESULTSTACK N : Take the current top of stack as an expression result.
    P!N := !S
    S := N

JUMP var : Jump to location _var_

RESULTEXIT L : Jump to location L; also states that the top of stack contains the result of an expression (generated by the BCPL _resultis_ command).

LAB L (*) : Define label L within program code.

DATALAB L (*) : Define label L within the data area.

COMMAND N (*) : Start of BCPL command number N.

ENDBLOCK nlist (*)[10] : Denotes end of a BCPL block.

ENDPROC nlist (*)[10] : Denotes end of a BCPL procedure.

ITEMC C : Defines a word with the character C stored right justified in it.

ITEMN N : Defines a word with the value N stored in it.

----

(10) The argument _nlist_ is a list of BCPL source names which can be optionally generated by the ICE translator.

ITEML L : Defines a word with the address of label L stored in it.

ITEMF F : Defines a word initialized to the floating point constant F stored in it.

ITEMS S : Allocates a contiguous block of store with the BCPL representation of the string S stored in them.

BUFFER N : Allocates N contiguous words of store without any initialization.

SECTION S (*) : An assembler directive specifying the start of section S.

NEEDS X (*) : Loader directive specifying that external symbol X is needed by the program.

INCLUDE S (*) : Assembler directive specifying that object file S should be concatenated to the object code generated by the present compilation.

PARAMETER S (*) : Implementation dependent assembler/code generator directive, as specified by the string S.

A.1.3. Diadic Operators

Diadic operators are described using the general form
<op> <arg(1)> <arg(2)>
where <op> is the operator being applied to arguments <arg(1)> and <arg(2)>. The remarks concerning argument types in section A.4.2. apply here as well.

MULT var1, var2: Push var1*var2 onto stack.

DIV var1, var2: Push var1/var2 onto stack.

REM var1, var2: Push var1 **rem** var2 onto stack.

ADD var1, var2: Push var1+var2 onto stack.

SUBTRACT var1, var2: Push var1-var2 onto stack.

EQ var1, var2: Push var1=var2 onto stack.

NE var1, var2: Push var1¬=var2 onto stack.

LS var1, var2: Push var1<var2 onto stack.

GE var1, var2: Push var1>=var2 onto stack.

GR var1, var2: Push var1>var2 onto stack.

LE var1, var2: Push var1<=var2 onto stack.

LSHIFT var1, var2: Push var1<<var2 onto stack.

RSHIFT var1, var2: Push var1>>var2 onto stack.

LOGOR var1, var2: Push var1|var2 onto stack.

LOGAND var1, var2: Push var1&var2 onto stack.

NEQV var1, var2: Push var1 **neqv** var2 onto stack.

EQV var1, var2: Push var1 **eqv** var2 onto stack.

FMULT var1, var2: Push var1#*var2 onto stack.

FDIV var1, var2: Push var1#/var2 onto stack.

FADD var1, var2: Push var1#+var2 onto stack.

FSUBTRACT var1, var2: Push var1#-var2 onto stack.

FEQ var1, var2: Push var1#=var2 onto stack.

FNE var1, var2: Push var1#¬=var2 onto stack.

FLS var1, var2: Push var1#<var2 onto stack.

FGE var1, var2: Push var1#>=var2 onto stack.

FGR var1, var2: Push var1#>var2 onto stack.

FLE var1, var2: Push var1#<=var2 onto stack.

PUSHINDX var1, var2: Push var2!var1 onto stack.

PUSHBYTE var1, var2: Push var2%var1 onto stack.

PUSHSELECT var1, var2: Push var2 **of** var1 onto stack.

MOVE var1, var2 : Store the value of var1 in the location
referenced by var2.
!var2 := var1

MODMULT MODDIV MODREM MODADD MODSUBTRACT MODEQ MODNE MODLS
MODGE MODGR MODLE MODLSHIFT MODRSHIFT MODLOGOR MODLOGAND MODEQV

MODFMULT MODFDIV MODFADD MODFSUBTRACT MODFEQ MODFNE MODFLS MODFGE MODFGR MODFLE] var1, var2

The effect of the MOD operators is similar to their non-modified counterparts described above, with the exception that the destination of the result is not the top of the stack but the location referenced by var2. For example,

```
MODDIV var1, var2
```

means

```
!var2 /:= var1
```

Note that the order of the operands, with respect to the non-modified operator, has reversed.

JUMPF var, L : Jump to the location referenced by L if var is **false.**

JUMPT var, L : Jump to the location referenced by L if var is **true.**

FCNCALL var, N : Function invocation

```
temp := P+N        || temp <- start of new frame
temp!0 := P        || save old frame pointer
temp!1 := C        || set new stack pointer
C := var           || branch to procedure
S := P+2           || set new stack pointer
```

RTNCALL var, N : Routine invocation

The effect of this instruction is the same as FCNCALL with the additional requirement that the result at the top of stack returned is deleted (i.e., popped off the stack). This also requires that RTNRTRN have the same semantics as FCNRTRN.

PROCENT S, L (*) : Specifies the start of the definition of the procedure named S. Also states that L is to be defined as the entry point to the procedure.

GLOBAL n, glist (*) : Defines a list (glist) of n pairs of the form (N L) where N is the global cell which is to be initialized to the address of label L.

ENTRYLIST n, elist : Defines a list (elist) of n pairs of the form (X L) where the entry symbol X is to be initialized to the address of label L. If L=0 then X is an external symbol.

A.1.4. Triadic Operators

Triadic operators are described by the general form
<op> <arg1> <arg2> <arg3>
where <op> is the operator applied to the three arguments <arg1>, <arg2>, and <arg3>.

SWITCHON var, N(0), L(0) : The SWITCHON command expects N(0) pairs of N L values immediately following it. Its effect is to scan the N field of each N L pair until a value equal to var is found. A branch is then made to the corresponding L. If no match is found, a branch is made to label L(0).

MOVEINDX var1, var2, var3 : Store into an indexed cell.
var3!var2 := var1

MOVEBYTE var1, var2, var3 : Store into a byte.
var3%var2 := var1

MOVESELECT var1, var2, var3 : Store into a selector field.
var3 of var2 := var1

[JUMPLS JUMPGR JUMPLE JUMPGE JUMPEQ JUMPNE JUMPFLS JUMPFGR JUMPFLE JUMPFGE JUMPFEQ JUMPFNE] var1, var2, L
The effect of each instruction in this class is to apply the relation following the JUMP to var1 and var2 (see the descriptions of the diadic relationals). If the result is **true**, a branch is made to label L. Note that the result is not stacked.
An example is 'JUMPLE X, Y, L045' whose effect is to jump to label L0045 if X<=Y.

A.2. Some Statistics on the Composition of BCPL Programs

In this appendix some figures are given regaring various aspects of BCPL programs. All measurements presented are based on static analyses of programs. Two major classes of analysis were performed: those from BCPL programs themselves[11] and those from the OCODE generated by the BCPL compiler. For the former class, about thirty-five BCPL sections were analyzed, for the latter about sixty sections. The sixty sections correspond to over 11,000 BCPL commands. In both instances, the programs locked at were largely of the "systems" variety, namely compilers, code generators, run-time support libraries, text editors and the like. Such a sampling was justified in that BCPL is specifically suited for systems applications; indeed other types of programs were unavailable for analysis.

The analysis of BCPL program composition revealed command usage frequency as shown in Table A.2.1. In addition, Table A.2.2. shows the average complexity of BCPL expressions, based on operator counts.

BCPL Command	Frequency (percent)[12]
Assignment	22
Routine application	27
Function application	13
<u>if</u>, <u>unless</u>, <u>test</u>	12
<u>while</u>, <u>until</u>	0
<u>repeat</u>, <u>repeatwhile</u>, <u>repeatuntil</u>	0
<u>for</u>	2
<u>return</u> (explicit)	1
<u>resultis</u>	7
<u>loop</u>	0
<u>break</u>	0
<u>endcase</u>	3
<u>goto</u>	0
<u>finish</u>	0
<u>switchon</u>	0

Table A.2.1. Frequency of BCPL command usage.

The second class of analysis was on the composition of OCODE generated from the BCPL compiler.[13] The data reported here is a set of measurements on the number of bits required to

----

[11] The data for these were gathered at UBC by Mark Fox.
[12] Values of less than 1 percent are shown as 0.
[13] the compiler used to generate the OCODE was the BCPL-V compiler at UBC. The BCPL-V language is a slightly enhanced version of standard BCPL. The compiler used to generate the OCODE did no optimization on the BCPL source.

represent the value of an operand.[14] [15] It should be realized though that the absolute address of a label operand is not easily determined during code generation. Since statistics were gathered during this phase the number of label operands encountered are listed separately in the column labelled <u>nr</u> in Table A.2.3. Following this column is a count of all operands less those which were labelled (<u>nr</u>-sum). Finally a complete sum is shown. Note that two's-complement notation is always assumed. Hence there is always one bit reserved for the sign, even if the operand can never be negative (as, for example, in the STACK command).

Number of Operators	Frequency (percent)[12]
0	76
1	19
2	1
3	0
4	0

Table A.2.2. Typical expression complexity in BCPL.

In table A.2.4., a comparision is made between the overall object size for ICE/ESO and SLIM (see [Fox] for details on SLIM). The data reported is the sum of the space occupied by data and code without any relativization of operands.

----

[14] Other analyses of OCODE command counts, etc. were performed. They are available upon request.
[15] Note that an operand is, in general, the pair <admode,value> where <u>admode</u> is the addressing mode (e.g., global, P-relative, etc.), and <u>value</u> the value for the specified admode. Measurements are made on the space occupied by the value field only since the admode field occupies a fixed number of bits (typically three).

Application	Min operand width in bits (2's compl) 2	3	4	5	6
UNIX Text Editor	434	671	696	153	30
Run-time Library	198	283	305	136	95
MCODE-HP cgen V1	344	692	527	244	47
MCODE-HP cgen V2	704	1257	1385	419	44
MCODE-Minicode	255	637	615	80	210
ALGAE Compiler	390	532	828	709	105
BCPL Compiler	885	1784	1840	923	103
BCPL-/370 cgen	895	1473	1794	762	351
InterLISP Kernel	350	1487	1284	250	25
BCODE cgen	421	416	670	498	160
Parsing Machine	82	167	307	119	9
ISAM Library	83	170	152	134	58
Permutations gen	34	52	105	9	1
Intcode Ldr,Int	118	271	199	18	9
Intcode Assembler	225	374	400	84	17
Towers of Hanoi	8	15	16	0	0
C Parser	100	276	344	140	10
Sum	5560	10643	11514	4723	1279
Cumulative Sum	5560	16203	27717	32440	33719

Table A.2.3a. Typical operand width in BCPL.

Application	Min operand width in bits (2's compl) 7	8	9	10	11
UNIX Text Editor	142	39	72	13	10
Run-time Library	30	2	4	3	0
MCODE-HP cgen V1	51	83	22	108	15
MCODE-HP cgen V2	63	74	6	102	4
MCODE-Minicode	127	120	112	10	22
ALGAE Compiler	100	159	16	40	6
BCPL Compiler	359	425	84	58	18
BCPL-/370 cgen	146	190	29	111	1
InterLISP Kernel	59	33	111	151	49
BCODE cgen	41	37	33	24	110
Parsing Machine	23	26	32	24	0
ISAM Library	11	41	71	0	1
Permutations gen	5	23	0	0	0
Intcode Ldr,Int	27	5	2	32	2
Intcode Assembler	29	19	1	24	113
Towers of Hanoi	1	3	0	0	0
C Parser	13	42	9	0	0
Sum	1230	1321	604	700	351
Cumulative Sum	34949	36270	36874	37574	37925

Table A.2.3b. Typical operand width in BCPL.

Application	Min operand width in bits (2's compl) 12	13	14	15	16
UNIX Text Editor	23	0	0	0	0
Run-time Library	0	0	0	0	0
MCODE-HP cgen V1	359	0	0	0	0
MCODE-HP cgen V2	44	1	0	0	0
MCODE-Minicode	7	8	78	0	0
ALGAE Compiler	9	0	2	19	0
BCPL Compiler	0	11	65	1	0
BCPL-/370 cgen	79	0	16	7	3
InterLISP Kernel	22	2	1	1	1
BCODE cgen	6	71	0	0	1
Parsing Machine	0	0	0	0	0
ISAM Library	0	0	0	0	0
Permutations gen	0	0	0	0	0
Intcode Ldr,Int	19	0	1	0	0
Intcode Assembler	0	0	69	0	0
Towers of Hanoi	0	0	0	0	0
C Parser	0	0	0	0	0
Sum	568	94	232	28	5
Cumulative Sum	38493	38587	38819	38847	38852

Table A.2.3c. Typical operand width in BCPL.

Application	nr (16)	SUM-nr (16)	SUM (16)
UNIX Text Editor	1456	2283	3739
Run-time Library	400	1056	1456
MCODE-HP cgen V1	1708	2492	4200
MCODE-HP cgen V2	2460	4076	6536
MCODE-Minicode	1436	2281	3717
ALGAE Compiler	1761	2915	4676
BCPL Compiler	4187	6556	10743
BCPL-/370 cgen	2681	5857	8538
InterLISP Kernel	2131	3826	5957
BCODE cgen	1337	2488	3825
Parsing Machine	441	789	1230
ISAM Library	184	721	905
Permutations gen	68	229	297
Intcode Ldr,Int	952	703	1655
Intcode Assembler	913	1355	2268
Towers of Hanoi	13	43	56
C Parser	693	934	1627
Sum	22895	38853	61748

Table A.2.3d. Typical operand width in BCPL.

(16) See the text of this appendix for details.

Application	SLIM	ICE	Ratio
Intcode Interpreter Sect. 0	5728	4978	1.15
Intcode Interpreter Sect. 1	1526	1430	1.07
Hanoi	218	194	1.12
BCPL Compiler LEX	6578	5240	1.26
BCPL Compiler SYN	6308	5148	1.23
BCPL Compiler TRNA	5670	4904	1.16
BCPL Compiler TRNB	5002	4180	1.20
BCPL Compiler TRNC	5986	4753	1.26

Table A.2.4. A comparision of ICE/ESO and SLIM code density (in bytes).

A.3. Using the BCPL/ICE Translator

This appendix gives instructions for running the BCPL/ICE translator, as available under MTS at UBC. The translator is invoked by the command

```
$RUN BCDE:BCPL SCARDS=sourcefile 0=icefile -
     SPRINT=listfile PAR=parameters
```

where sourcefile is the file containing the BCPL source,
icefile is the file to which the ICE code will be written,
listfile is the file to which the program listing is to be directed, and
parameters is the normal parameters list used by the standard BCPL compiler.

The compiler automatically generates statistics on the listfile giving the size of the ICE object using the ICE/ESO machine.