Title
AN INTRODUCTION TO BLIMP: A SYSTEMS PROGRAMMING LANGUAGE FOR THE CDC 6000/7000 CPUs

Author
Vardas, Leo S.

Publication Date
1976-04-01

Presented at the VIM 24 Conference,
Jack Tar Hotel, San Francisco, CA,
April 6 - 8, 1976

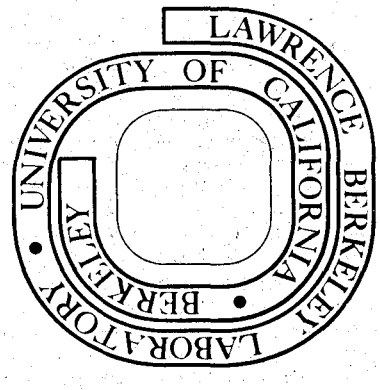
LBL-4828
c.1

AN INTRODUCTION TO BLIMP A SYSTEMS PROGRAMMING LANGUAGE FOR THE CDC 6000/7000 CPUs

Leo S. Vardas

April 1, 1976

Prepared for the U. S. Energy Research and
Development Administration under Contract W-7405-ENG-48



LBL-4828
c.1

DISCLAIMER

Session: 32B
Languages and Processors Committee

An Introduction to BLIMP

A systems programming language for the CDC 6000/7000 CPUs

By Leo S. Vardas

ABSTRACT

BLIMP is a systems programming language for the CDC 6000/7000 CPUs. It was developed primarily for the implementation of procedures which deal with data items, or fields, in tables of the type typically found in operating systems. The language incorporates the concepts of structured programming both in its control constructs and in the techniques of code generation and optimization. This compiler is currently being used at BKY. The 6000 System deadstart loader, the associated HELP utility, a macro-processor, a meta-compiler, the compiler itself, and an assortment of utilities have been implemented in BLIMP.

INTRODUCTION

One of the features of BLIMP is the efficient accessing of fields - data items which are contiguous bit strings and subsets of a 60 bit word. Whereas conventional procedural languages are word oriented and, at best, provide an ad hoc bit or character accessing capability, BLIMP can treat the field as a fundamental addressable unit. This is expecially true of the code generator and optimizor. Field access is not performed by a procedural utility; the compiler generates inline instructions similar to those produced by conventional handcoding techniques.

FIELD TEMPLATE

Consider the following abridgement of the File Name Table (FNT) as it appears in the SCOPE Reference Manual:

```
            59                 18        11           0
Word 0    logical file name          link to
             C.FNAME              optional supplement
                                       C. FLNKAD
Word 1
                52            36
Word 2            FET address
                   C.FETAD
```

The mnemonic C.FNAME is associated with the leftmost 42 bits of the first word of the FNT. Now C.FNAME refers to the logical-file-name field of any FNT. The mnemonic has not been bound to any specific FNT. It is then, in a sense, an element of a based array. We can think of the identifier C.FNAME as the name of a template; by appropriately positioning this template, the logical-file-name field of a specific FNT can then be accessed.

The template C.FNAME in the example implicitly establishes an operation or function defined by the characteristics of word offset and left and right bit positions. As such, the addition of an operand, namely an address of a specific FNT, denotes a particular field access. Syntactically, this is written in the classical functional notation: C.FNAME (A.FNT) where A.FNT is the name of a FNT table or array in the FORTRAN sense. This syntax states that the template C.FNAME is "applied" to the table A.FNT.

This form of the functional notation is adequate for operands which are named tables. However, a typical application will need to access tables which are dynamic or nameless, and referenced only by an address. Such tables are accessed indirectly through a pointer. Borrowing from the practices of mini-computer assembly language, an '*' preceding an operand is used to indicate an indirection operation. For example, let LC.FNT be the name of a cell whose contents is the address of a FNT. Then C.FNAME (*LC.FNT) denotes the application of the template to the FNT.

Examples

C.FNAME (TABLES (I))	accesses a FNT which is located at the I-th word of TABLES.
C.FNAME(*X+100B)	evaluates X+100B. The result is the address of the desired FNT.

One of the advantages of the template mechanism is that structurally identical fields from different tables can be referenced by the same mnemonic. For example, if the logical-file-name fields of two distinct FNTs were to be compared, it could be done by
IF C.FNAME(*FNT.1)>C.FNAME(*FNT.2) etc.

FIELD TEMPLATE DEFINITION

The format for a simple field template definition is FIELD name / W, L, R [,ATT]/

where
W is word offset =0,1,2, --
L is left bit position =59, 58,--- ,ø
R is right bit position =59, 58,--- ,ø

ATT is an optional 3 letter mnemonic whose components are memory, sign-type, and address-type where

memory is S for SCM
L for LCM
sign-type is N for numeric (leading bit is a sign bit)
T for typeless
address-type is W for arbitrary word
A for address implying 17 bit address.

The A address-type is useful for efficiently extracting a 17 bit address pointer without masking. It is accomplished by a SX X. STW is the default ATT.

Examples

```
FIELD C.FNAME/ 0, 59, 18 /
FIELD C.FETAD/ 2, 53, 36, STA /
FIELD C.FLNKAD/ 0, 11, 0 /
FIELD F.FET.NAME/ 0, 59, 18 /
```

CFNAME(*LC.FNT)=F.FET.NAME(*C.FETAD(*LC.FNT))
copies the FET file name field into the file name field of the associated FNT.

It often is the case that a field identifier should be bound to a specific table rather than be a "floating" template. This facility is available in BLIMP through an extended field declarative. For example

FIELD FBOUND/0,43,38/$(* B7) $ binds FBOUND so that the template will be applied to the table whose address is in B7. This definition is, in essence, a macro extension to the field template.

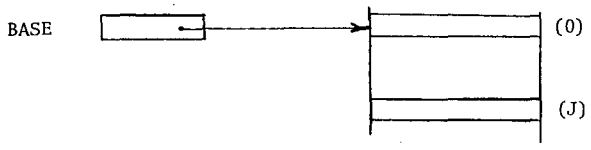
VALUE = BOUND
executes exactly as the following:

FIELD FBOUND /0, 43, 38 /
VALUE = FBOUND (*B7)

The following example defines a parameterized template-macro with one parameter, REL, a relative offset:

```
FIELD FBASED (REL)/1,47,47/$(BASE+REL)$
```

This definition would be used for accessing a field of a table at some to-be-specified offset from a position specified by the address in BASE.



FBASED (J) references a field in the word whose location is the value of BASE+J+1. This mechanism is provided for referencing fields of dynamic tables. Note that the indirection operation allows an arbitrary expression and uses the low order bits of the result as an address.

STATEMENT FORMAT AND ELEMENTS

Most BLIMP statements are free field and text may start in column one. Only one statement per line is permitted. A plus sign in col. 1 indicates a continuation line. Although 90 col. card images are accepted, only the first 72 columns are used.

There are two kinds of comments. An asterisk in col. 1 indicates a comment line. A string starting with a period indicates the end of the statement and the remainder of the line is interpreted as a comment. The compiler will then automatically right justify the second form of the comment when the output listing is created.

A label is a lone identifier appearing anywhere on a line. It must be the only character string on the line except for a comment.

Examples

```
A.LABEL                 .A COMMENT
LBL   X = 1             .incorrect statement
```

Identifiers may be up to 63 alphanumeric characters starting with a letter. Embedded periods are permitted except as the first chr. Embedded blanks are not permitted.

Examples

```
F.TABLE.FIELD
Q..WEIRD..
```

Numeric constants are the usual decimal and octal digit sequence with the restriction that octal constants must be suffixed by a 'B'without any blanks following the last digit.

The hollerith constants are of the type H,L,R,C and Z as in COMPASS. Two forms are available: chr count preceding the type and string (as in FORTRAN); type followed by quoted string.

Examples

```
1249                    37 77 77B
3HABC                   H'ABC'
9CTWO-WORDS             C'TWO-WORDS'
'TWO-WORDS'             .defaults to C type
```

CONTROL STRUCTURES

In many ways BLIMP follows the conventions of FORTRAN. Its storage allocation is static, recursion is not available, and the relocatable subroutine structure is similar. However, despite these similarities, the two languages are really quite different, particularly with respect to the control structures.

Six block oriented control structures are provided. These are block oriented in the sense that a sequence of statements - a code block - is preceded by a keyword statement and delimited by a matching end statement. The body of the block itself may consist of any executable statements including nested blocks.

For brevity, the descriptions of the structures are provided as comments in the following examples.

```
IF X <0 THEN    .If block; no further text after THEN
  X=0           .Set X=0 and call if condition
  CALL SUB(Z)   .is true
ELSE            .ELSE block is optional
  RETURN
ENDIF           .Both paths join at ENDIF

IF (X LT Ø)(Y=X)THEN .Compound/Boolean conditional
  X=0
  IF Z>0 THEN         .Nested IF block
  Y=0
  CALL EXIT
  ENDIF               .W/0
ELSE                  .ELSE block
                      .ELSE block,part of outer block
  IF Z GT 0 THEN      .Another nested IF block
  Z=0
  ELSE                .With a corresponding ELSE
  Z=1
  ENDIF               .Ends inner block
  RETURN
ENDIF                 .Ends outer

IF X=0 THEN RETURN    .A simple statement (non-block)
                      .After THEN
                      .Analogous to FORTRAN IF.

IF CHR = 1RA THEN
  CALL SUBA
ELSEIF CHR = 1RB THEN.Combined ELSE and NEW IF
  CALL SUBB           .Avoids excessive nesting
ELSEIF KEY>'AB'THEN   .And multiple
  CALL STUFF          .ENDIFS.
ELSE                  .ELSEIF is one word
  CALL NONE           .ELSE can only follow
                      .ELSEIF blocks (if any)
ENDIF

WHILE X <100 DO       .Like IF with backward
  X=X+1               .Branch
  CALL PRO

ENDWHILE
REPEAT                .Indefinite loop
  CALL PRO(X)         .Between REPEAT
  IF X=0 THEN RETURN  . And
ENDREPEAT             .ENDREPEAT

REPEAT                .Like above.
  CALL PRO(X)         .UNTIL condition
UNTIL X=0             .Imposes condition and delimits
  RETURN              .Loop. Any statements between
EMDREPEAT             .UNTIL and ENDREPEAT executed
                      .Once
```

LEAVE and RESTART are special forms of branching from within blocks. LEAVE exits the block and transfers control to just beyond the end of the block. It is possible to exit from within nested blocks by associating a name (i.e., a label) with the block. The RESTART is somewhat similar. It is a backward branch and its effect is to re-enter the block.

```
REPEAT
  CALL SUB(X)
  IF X = 0 THEN RESTART     .Start loop over
  CALL POST(X)
  IF X>1 THEN LEAVE         .terminate by exiting loop
ENDREPEAT

LBL                         .a label immediately
                            .preceding a block.
                            .comments,blanks ignored

REPEAT
  N=READC(INPUT,BUFFER,MX)
  IF N>0 THEN
    CALL PROCESS (FLAG)
    IF FLAG = 0 THEN LEAVE LBL.terminate entire loop
    IF FLAG = 1 THEN LEAVE    .exit the IF block
      CALL POST
      RESTART LBL
  ENDIF                        .start entire loop over
*   LEAVE DESTINATION          .ends IF N>0 block
ENDREPEAT
*  LEAVE LBL DESTINATION



  CASE I + J                .Jump table construct,e.g.,
                            .computed GO TO
1, 13                       .a 'label'. Block is
  CALL ONE                  .selected if I+J= 1 or 13
  RESTART

12                          .order is immaterial
  CALL EXIT

<1                          .optional test block. No
 CALL LOW                   .bounds check unless ex-
                             plicitly
                            .requested


6 THRU 10, 4                .6,7,8,9,10 and 4 block
  X = 6
  Z = 10
ELSE                        .2,3,5,11
*OPTIONAL CODE BLOCK HERE
ENDCASE                     .no test for >13
                            .all code blocks of CASE
                            .rejoin at ENDCASE


MATCH KEYWORD(J)            .A search construct against
                            .table of constants
'IF', 'WHILE'               .Any 60 bit constant
                             qualifies
 CALL RELAT                 .Keyword (J) is compared with
                            .each 'label' for a match


'REPEAT'                    .A linear search is performed
 CALL REPEATS               .Starting with 'IF', then
                             'WHILE'
'CASE','MATCH'              .etc.
 CALL TABLE
 MATCH KEYWORD(J)           .nested MATCH Block
  'CASE'
  CALL CASES
 'MATCH'
  CALL MATCHS
 ENDMATCH
3LABC,425,1R*,377B          .ridiculous 'labels' but here
 KEYWORD(J)=0               .to demonstrate constants
 RESTART
ELSE
 CALL NONSENSE              .call a local(not external)
 GO TO FAIL.EXIT             subroutine
                            .GO TO is alive ( and well?)
ENDMATCH
```

```
FOR I = M THRU K+J BY -2    .A DO loop construct
 FOR J = LOW UNTIL LIMIT    .THRU is inclusive
                            .UNTIL is 'up to but
                            .not including'

 ENDFOR
 ENDFOR

FOR I THRU MXTAB            .lower limit
                            .defaults to 0 or 1
                            .depending on whether
                            .subscripting zero or 1
                             base.
 IF TAB (I) =KEY THEN LEAVE
THEN
                            .optional block-ends
                            .loop
 MXTAB=MXTAB+1              .handle exception condition
 TAB(MXTAB) =KEY            .w/o labels
ENDFOR
```

ADDITIONAL FEATURES SUMMARY

BLIMP modules may have embedded local subroutines. These routines may optionally be global - like FORTRAN's ENTRY. The DIMENSION and DATA statements are incorporated into a composite DECLARE statement. COMMON blocks via a GLOBAL block declarative are available in conjunction with the DECLARE. An intrinsic function which permits access to an absolute memory location is provided for both fetching and storing.

The implementation of BLIMP reflects the philosophy that programming languages should, when possible, be complementary, not competetive. Therefore, subroutine linkage is compatible with FTN. More importantly, inline COMPASS statements may be inserted within a subroutine. This feature essentially makes available the whole spectrum of COMPASS system macros for, as an example, specialized I/O, initialization, and system calls. Since BLIMP produces COMPASS modules as its object code, it merely transliterates any embedded COMPASS statements referencing a BLIMP identifier.

A declarative is available for reserving and accessing machine registers. This is especially useful for creating a replacement module for an existing system routine or overlay which employs a special parameter communication convention.

The compile time facility of BLIMP has been modeled after COMPASS. Compile time variables may be defined and given numeric values analogous to the SET operation in COMPASS. Conditional compilation is provided and an extended MACRO capability provides a parameterized MICRO facility as well.

Finally, the compiler edits the original source code; statements are automatically indented, comments right justified and a profile of the nesting of the various blocks and sub-blocks is indicated. Extensive cross referencing of all symbols including macros and system text identifiers is optionally available.

Although BLIMP does not incorporate any special I/O statements, a run-time library has been developed. It is primarily an interface to the many KRONOS and 7600 SCOPE 1 I/O and system macros. In addition a utility for composing formatted output text with automatic line counting, titling and heading control is available.

LAWRENCE BERKELEY LABORATORY
SYSTEMS PROGRAMMING GROUP
Bldg. 50A, Room 1121A
Berkeley, Ca. 94720

Work done under the auspices of the U.S. ERDA.

TECHNICAL INFORMATION DIVISION
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720