International Journal of Computer and Information Sciences, Vol. 7, No. 2, 1978

On the Time Required for Reference Count Management in Retention Block-Structured Languages. Part 2[1]

D. M. Berry,[2] L. M. Chirica,[2] J. B. Johnston,[3] D. F. Martin,[2] and A. Sorkin[2]

Received February 1975; revised June 1976

In this paper, two implementations of generalized block-structured languages are presented and compared for time requirements. One implementation, the Lifetime Stack Model, implements the deletion strategy with lifetime checks; the other, the Partial Reference Count Contour Machine, implements the retention strategy. For a large subset of the lifetime well-stacking programs, those that run correctly on the first model, the two models are shown to require nearly the same order of magnitude of time. The use of full label values is shown to have a detrimental effect on the time efficiency of the latter model. Part 1, in Volume 7, Number 1, of this journal, gives a general description of the machines, some of their definitions, and proof of the results. Part 2, in this issue, serves as an appendix to Part 1 and contains most of the formal definitions of the machines.

KEY WORDS: Block-structured languages; retention vs. deletion; contour model; stack model; reference counts; lifetime checks; time estimates.

This appendix defines the machine instructions for the LSM and the PRCCM, alluded to in Sects. 3.6 and 4.6 in Part 1 of this paper. The definitions are given in parallel with the definition of the LSM in the left column and the definition of the PRCCM in the right column. This was done both to take advantage of similarities and to point out the differences. When the two machines agree completely, the description runs across both columns. When they differ, the description is split into two columns.

[1] Supported (in part) by the National Science Foundation, Grant No. DCR 75-08659.

[2] Computer Science Department, UCLA, Los Angeles, California 90024.

[3] Computer Science Department, New Mexico State University, Las Cruces, New Mexico 88003.

0091-7036/78/0600-0091$05.00/0

3.6 MACHINE INSTRUCTIONS

4.6 MACHINE INSTRUCTIONS

Every GBSL program is translated into Polish code consisting of the instructions listed in Sections 3.6.2 and 4.6.2. There are twenty-two basic instructions. Some of them, i.e., VAL, POP, IND, STO, EXIT, and RETURN, have more than one option, each indicated by a different suffix (no suffix is considered an empty suffix).

For each basic instruction, an English description of the basic function of the instruction is given. If the instruction has several options indicated by different suffixes, each of the variations is listed with an English description of the change from the basic function. Following the English description is commented pseudo Algol microcode to implement the instruction. This is followed by an estimate as to the time required to execute the instruction (e.g., constant, proportional to "X", potentially unbounded, T (to execute something)[§], or a sum of any of these). If the two models have identical times[†] for a given instruction, then the time indication is asterisked in both columns. There are some conventions for giving the microcode:

1) Statements are in free format and are separated by semi-colons.
2) Comments begin with a "¢" on any line and end at the end of the same line.
3) Keywords are words with all uppercase letters.
4) Identifiers are strings of consecutive lowercase letters, digits, and underscores.
5) The conditional is IF...THEN...ELSE...FI. The for loop is FOR...FROM...BY...TO...DO...OD.
6) Arithmetic can be done on pointers (by "pointer" is not meant a pointer value but the contents of a lo or hi field).
7) Indirection of a pointer is indicated by a "*" following the pointer, e.g.,
   ep*
8) Subscripting of a contiguous block of words pointed to by a pointer is defined by
   ptr* [i] = (ptr + i)*
9) A trimmer can be used to select a consecutive list of words, e.g.,
   ptr* [i:j]
   will select the i through jth words of the cell pointed to by ptr.
10) A field of a word can be selected by use of a field name following a ".", e.g.,
   ptr*. hi
   selects the hi field of the word pointed to by ptr.
11) In expressions, the following precedence is used:
   highest: unary +,-
   ×,/
   binary +,-
   <, ≤, =, ≠, ≥, >
   ¬

∧
∨
lowest: indirection, subscripting, trimming, selection.

Within a precedence level, the association is left-to-right.

12) The following synonyms are available for use as selectors:

as substitutes for hi:
ep, sp, life, sl
as substitutes for lo:
ip, dl, ptr, ts
as a substitute for mid:
nh

13) The components of a processor's soa may be selected directly by applying a component name to Π, e.g.,

Π.ip, Π.ep, Π.nh, Π.sp, Π.ts

14) The display is selected by a special abbreviation, DISP.

15) Some abbreviations for use in accessing words of the stack are given:

1st is an abbreviation for
(Π.ts + Π.sp - 1)*
and selects the top word of the stack

2nd is an abbreviation for
(Π.ts + Π.sp - 2)*
and selects the second word of the stack

⋮

nth is an abbreviation for
(Π.ts + Π.sp - n)*
and selects the nth word of the stack

---

† The two models are said to require identical time for an instruction if the microcode implementing the instruction is identical in both models.

§ T(×) is the Time for ×.

There are some basic primitives called by the microcode. Time estimates are given with the primitives. The time for any primitive whose description stretches over both columns is the same in both models.

1) makeint (int): forms a properly tagged integer value with int as the val.

Time: constant*

2) makebool (bool): forms a properly tagged boolean value with bool as the val.

Time: constant*

3) makeptr (ep, ptr): forms a properly tagged pointer value with ep as the ep and ptr as the pointer proper.

Time: constant*

4) makeproc (ep, nh, ip): forms a properly tagged procedure value with ep as the ep, nh as the nh, and ip as the ip.

Time: constant*

5) makelbl (ep,nh,ip): forms a properly tagged label value with ep as the ep, nh as the nh, and ip as the ip.

Time: constant*

6) makestk (sp,ts): forms a properly tagged STK word with sp as the sp and ts as the ts.

Time: constant*

7) makeund(): form a properly tagged undefined value

Time: constant

8) makertl (nh, ip): forms a properly tagged return label with nh as the nh and ip as the ip.

Time: constant*

9) makeARcont (sl, nh, dl): forms a properly tagged AR control word with sl as the sl, nh as the nh, and dl as the dl.

Time: constant*

10)

makertl (ep, nh, ip): forms a properly tagged return label with ep as the ep, nh as the nh, and ip as the ip.

Time: constant*

alloc_contour (n): search the free list of the contour segment from the beginning to find a free block of length greater than or equal to n. If one is not found then initiate garbage collection. Otherwise carve out the first n words of the free block to make a contour. The remainder of the free block is left in the free list in the same place as the original free block. The tag of the first two words of the contour are set to LNK and CT respectively. Search the allocated contour list from the end for the proper place to link the new contour so that the allocated list is ordered by address; link the contour into the list. Set the reference count of the contour to 1. Return a pointer pointing to the 2nd word of the contour.

Time: potentially unbounded

11) free_contour (ptr): search the free list of the contour segment from the beginning to find the proper place in the address ordered free list for inserting the contour pointed to by ptr. The contour is added to the free list, and if it is physically adjacent to either of its neighbors on the list, then it must be merged with its adjacent neighbor(s). If the LNK word of the freed contour is no longer used as a link then it is reset to an undefined value.

Time: potentially unbounded

12) alloc_stack (n): same as alloc_contour except using stack segment. It returns a pointer pointing to a fully tagged ces of length n.

Time: potentially unbounded

13) free_stack (ptr): same as free_contour, except using stack segment and the ces pointed to by ptr.

Time: potentially unbounded

There are some "hardware" procedures called by the microcode: as with basic primitives, time estimates are given.

```
1) push (v) =
    Π.ts ← Π.ts + 1; ¢Increment processor's ts by 1.
    1st ← v; ¢Assign value to new stack top.
```

Time: constant*

```
2) pop =
    Π.ts ← Π.ts - 1; ¢Decrement processor's ts by 1.
```

Time: constant*

```
3) pop (n) =
    Π.ts ← Π.ts - n; ¢Decrement processor's ts by n.
Time: proportional to n*
```

4)

```
incrc(ptr) =     ¢If ptr is not NIL,
 IF ptr ≠ NIL    ¢increment ref count of
   THEN          ¢cell pointed to by ptr
    ptr*.rc ← ptr*.rc+1;    ¢by 1 and
    RETURN ptr*.rc;         ¢return new ref
   ELSE error               ¢count as value.
 FI

Time: constant
```

5)

```
decrc(ptr) =     ¢If ptr is not NIL,
 IF ptr ≠ NIL    ¢decrement ref count of
  THEN           ¢cell pointed to by ptr
   ptr*.rc ← ptr*.rc-1;      ¢by 1 and
   RETURN ptr*.rc;           ¢return new ref
  ELSE error                 ¢count as value.
 FI

Time: constant*
```

```
6) reconstruct_display =
    ptr ← Π.ep;  ¢Set ptr to copy of Π's ep.
    FOR i FROM Π.nh - 1 BY -1 WHILE ptr ≠ NIL  ¢For as long as there is
     DO    ¢a static chain, each successively lower  DISP element is
      DISP[i] ← ptr;  ¢loaded with each successive element of the
      ptr ← ptr*.sl  ¢static chain.
     OD;
Time: proportional to Π.nh - 1*
```

7)

```
is_scoped (tag)=   ¢Used to determine if tag is that
 RETURN( tag=PTR v tag=PRC     ¢of a reference
       v tag=LBL v tag=RTL);   ¢countable word.

Time: constant
```

One pair of suffixes appears quite frequently and is explained once here. For VAL, POP, IND, and RETURN there is a choice of two suffixes: none and P. All of these instructions end up leaving or popping one value in or from the (expression) stack. For exactly what value is left or popped, see the individual instruction descriptions. The suffix gives the type of value being left or popped. If the suffix is P, then the value is of a pointer, label or procedure mode. If the suffix is empty, the value is of an integer or boolean mode.

In LSM, the presence of a P suffix indicates the possibility of a lifetime check being performed as part of the instruction execution.

In PRCCM, the presence of a P suffix indicates the possibility of some reference count updating being done as part of the instruction execution.

3.6.1 Instruction cycle

4.6.1 Instruction cycle

Both machines follow the normal fetch increment and execute cycle as follows

```
WHILE is-instruction (Π.ip*)  ¢Is there a next instruction?
   DO inst ← Π.ip*;  ¢Fetch instruction.
     Π.ip ← Π.ip + length (inst);  ¢Increment ip.
     execute (inst);  ¢Execute the instruction.
   OD
```

where 1)is-instruction (inst) returns TRUE if and only if inst is an instruction.

2)length (inst) is the number of words that inst takes up in the instruction stream.

3)execute (inst) is the execution of the instruction inst as described by the microcode in the following section.

3.6.2 Execution of Instructions§	4.6.2 Execution of Instructions§
1) CONST, i Push i, where i is an integer, TRUE, FALSE, or NIL.	
push (i); ¢Push the constant itself; note that ¢the constant must contain its own tag	push (i); ¢Push the contant itself; note that ¢the constant must contain its own tag.
Time: constant*	Time: constant*
2) VAL, i,j Push the value starting at the jth word of the AR or contour pointed to by DISP [i].	

§ For the sake of brevity in the instruction definitions, both models assume that all variables accessed have been initialized, and therefore, neither model checks whether a used value has the correct tag. Likewise neither model checks whether a pointer value to be indirected is non-NIL. These checks could easily be added to both models at a similar cost.

a) VAL, i,j Push an integer or boolean value.

```
push ((DISP[i] + j)*);
```

Time: constant*

```
push ((DISP[i] + j)*);
```

Time: constant*

b) VALP, i, j Push a pointer, label or procedure value.

No change

```
push ((DISP[i] + j)*);
```

Time: constant

Increment the reference count of the cell referred to by the pushed value.

```
push ((DISP[i] + j)*);    ¢Push value.
incrc (1st.ep);           ¢Increment reference count.
```

3) ADR i,j Push a pointer pointing to the jth word of the AR or contour pointed to by DISP[i].

```
push (makeptr (DISP[i],   ¢Push both the ep and
    DISP[i] + j));        ¢the pointer proper.
```

Time: constant

The reference count of the cell referred to by the pushed pointer is incremented.

```
push (makeptr (DISP[i],   ¢Push both the ep and
    DISP[i] + j));        ¢the pointer proper.
incrc (1st.ep);  ¢Increment ref count.
```

Time: constant

4) ADD Replace the second stack word by the sum of the top two words and pop once.

```
2nd ← makeint (1st.val + 2nd.val);
pop;
```

Time: constant*

```
2nd ← makeint (1st.val + 2nd.val);
pop;
```

Time: constant*

5) EQ Replace the second stack word by the truth value of whether the top two words are equal, and pop once.

```
2nd ← makebool·(1st.val = 2nd. val);
pop;
```

Time: constant*

```
2nd ← makebool (1st.val = 2nd.val);
pop;
```

Time: constant*

6) NEG Replace the first word of the stack by its arithmetic negation.

```
1st ← makeint(-1st.val);
Time: constant*
```

```
1st ← makeint(-1st.val);
Time: constant*
```

7) POP Pop one value from the top of the stack.

a) POP Pop an integer or boolean value.

```
pop;
Time: constant*
```

```
pop;
Time: constant*
```

b) POPP Pop a pointer, label, or procedure value.

No change

```
pop;

Time: constant
```

Decrement the reference count of the cell referred to by the popped value.

```
decrc (1st.ep);   ¢Decrement ref count.
pop;              ¢Pop the value.

Time: constant
```

8) IND Replace the pointer on top of the stack by the value it points to.

a) IND The pointed-to value is an integer or boolean value.

```
1st ← 1st.ptr*;   ¢Pick up pointed-to value.

Time: constant
```

The reference count of the cell referred to by the pointer is decremented.

```
decrc (1st.ep);   ¢Decrement ref count.
1st ← 1st.ptr*;   ¢Pick up pointed to value.

Time: constant
```

b) INDP The pointed-to value is a pointer, label, or procedure value.

No change

In addition to the reference count decrementation of a), the reference count of the cell referred to by the pointed-to value is incremented.

```
1st ← 1st.ptr*;

Time: constant
```

```
decrc (1st.ep);   ¢Decrement ref count of cell ptd-to
                  ¢by ep of pointer.
1st ← 1st.ptr*;   ¢Pick up pointed-to value.
incrc (1st.ep);   ¢Increment ref count of cell ptd to
                  ¢by ep of pointed-to value.

Time: constant
```

9) STO STO has two independent sets of suffixes.

1) The first suffix is either V or I. The V suffix is used when the left hand side of the implemented assignment is a variable. The I suffix is used when the left hand side is an indirection.
2) The second suffix is either none or P. This suffix indicates the type of the value being assigned as with VAL.

In the case of the STOV's, the instruction has the (i,j) pair of the variable being assigned to as its two arguments. In the case of the STOI's, a pointer value pointing to the cell to be assigned to is assumed to be in the second stack position. In either case, the value to be assigned is on top of the stack. The assignment is performed, the pointer value, if any, is cleared from the stack, and the assigned value is left in the stack as the value of the assignment.

a) STOV, i,j Assign an integer or boolean value to a variable.

```
(DISP[i] + j)* ← 1st;  ¢Assign top stack value to
                       ¢word accessed by (i,j).
```

```
(DISP[i] + j)* ← 1st;  ¢Assign top stack value to
                       ¢word accessed by (i,j).
```

Time: constant*	Time: constant*
b) STOVP, i,j Assign a pointer, label, or procedure value to a variable. A lifetime check is performed.	If the to-be-erased value in the to-be-assigned word is a pointer, label or procedure value§, then the reference count of the cell referred to by this value must be decremented, unless the value resides in the cell. After the assignment is done, the reference count of the cell referred to by the just-assigned value must be incremented, unless the assigned-to word is in the referred-to cell.
	§ Upon allocation of a contour, all programmer accessible words are guaranteed to have either an INT, BOO, STK, or UND tag. Therefore, because of the compile time type checking on assignments, if a word that is to be assigned a pointer, label, or procedure value has any reference countable value at all, it must be a pointer, label, or procedure value validly initialized since allocation of the contour. Thus, it is necessary for STOxP, with x = V or I, to check only whether the to-be-erased word has no PTR, LBL, or PRC tag, to be sure that the to-be-erased word has no reference countable value.

```
IF DISP[i] < 1st.ep          ¢Lifetime check.
 THEN lifetime_error;          ¢If not OK then error;
 ELSE (DISP[i] + j)* ← 1st     ¢else do
FI;                            ¢assignment.
```

Time: constant

```
ptr ← DISP[i] + j;    ¢Get address of cell to be
                      ¢assigned to.
IF ptr*.tag=PTR ∨     ¢If to-be-erased value is a
   ptr*.tag=LBL ∨     ¢pointer, label, or procedure
   ptr*.tag=PRC       ¢value, and
 THEN
  IF DISP[i] = ptr*.ep  ¢if value not reside in
   THEN                 ¢referred-to cell then,
    decrc(ptr*.ep) FI FI;  ¢decr ref count.
ptr* ← 1st;     ¢Assign value.
IF DISP[i] ≠ ptr*.ep   ¢If assigned value not reside
 THEN                  ¢in referred-to cell, then
  incrc(ptr*.ep) FI;   ¢increment ref count.
```

Time: constant

c) STOI Assign an integer or boolean value to an indirection. The value to be assigned is on top of the stack, and the pointer value pointing to the cell to be assigned to is below that. After the assignment, the assigned value is slid down the stack one position and the stack is popped by one.

```
2nd.ptr* ← 1st;  ¢Assign value to ptd-to cell.
2nd ← 1st;       ¢Move value down stack.
pop;             ¢Pop stack once.
```

Time: constant

The reference count of the cell referred to by the pointer value must be decremented.

```
2nd.ptr* ← 1st;  ¢Assign value to pted-to cell.
decrc(2nd.ep);   ¢Decrement ref count.
2nd ← 1st;       ¢Move value down stack.
pop;             ¢Pop stack once.
```

Time: constant

d) STOIP Assign a pointer, label, or procedure value to an indirection.
A lifetime check is added to the code for STOI.

In addition to that which is done in STOI, it is necessary to decrement the reference count of the cell referred to by the to-be-erased value and to increment the reference count of the cell referred to by the assigned value.

```
IF 2nd.ep < 1st.ep
 THEN lifetime_error
 ELSE 2nd.ptr* ← 1st;
      2nd ← 1st;
      pop FI;
```

Time: constant

```
IF 2nd.ptr*.tag = PTR ∨ 2nd.ptr*tag = LBL ∨
   2nd.ptr*.tag = PRC
 THEN
  IF 2nd.ep ≠ 2nd.ptr*.ep
   THEN decrc(2nd.ptr*.ep) FI FI;
2nd.ptr* ← 1st;
decrc(2nd.ep);
IF 2nd.ep ≠ 2nd.ptr*.ep
 THEN incrc(2nd.ptr*.ep) FI;
2nd ← 1st;
pop;
```

Time: constant

10) MAKE-LAB-STACK Manufacture and STK word for the 1th word of the AR or contour pointed by the processor's ep.
The STK word consists of a copy of the processor's sp and ts.

Form a ces for the use of all label constants declared in the block just entered. The ces is initialized to a copy of the currently active portion of the ES, and the reference counts of all cells referred to by the reference countable values in the active portion of the ES must be incremented. A pointer to the ces and the processors current ts are used to make the STK word.

```
(Π.ep + 1)* ← makestk(Π.sp,Π.ts);
```

Time: constant

```
m ← Π.ts;   ¢Height of active portion of ES.
ptr ← alloc_stack (m+1)  ¢Allocate ces of size or
                         ¢m+1, and
                         ¢set ptr to point to it.
ptr*[0:m-1]←Π.sp*[0:m-1]; ¢Copy m words of ES into
                         ¢ces.
FOR i FROM 0 BY 1 TO m-1 DO  ¢Incr ref count of
                             ¢each cell
 IF is_scoped(ptr*[i]. tag)  ¢ptd to by a ptr, la-
                             ¢bel, or procedure
  THEN                       ¢in ces.
   incrc(ptr*[i].ep)
 FI OD;
Π.ep*[1] ← makestk(ptr,m);  ¢Set STK word with
                            ¢sp=ptr and ts=m.
```

Time: potentially unbounded

11) MAKELABEL, ip Form and push a label value using the argument ip as the ip, and the processor's nh and ep as the nh and ep.

```
push(makelbl(Π.ep,Π.nh,ip));   ¢Push value.
```

Time: constant

It is necessary to increment the reference count of the contour pointed to by the ep.

```
push(makelbl(Π.ep,Π.nh,ip));   ¢push value.
incrc(1st.ep);     ¢Increment ref count.
```

Time: constant

12) GOTO

Pick up the label value on top of the stack, and reset the processor's ip, ep, and nh to copies of those of the label value. Reset the processor's sp and ts to copies of those of the STK word of the AR now pointed to by the processor's ep. Then, reconstruct the display.

Pick up the label value on top of the stack, and reset the processor's ip, ep, and nh to copies of those of the label value. From the contour now pointed to by the processor's ep an STK word can be picked up. Copy the contents of the ces pointed to by the sp of the STK word into the ES. Then, reset the processor's ts to that of the STK word. Finally, reconstruct the display.

It is necessary to decrement the reference counts of the cells referred to by the values in the ES before the goto, and to increment the reference counts of the cells referred to by the values in the ES after the goto. It is also necessary to ripple reference count decrementation down the chain of contours in the normal return sequence from the processor's ep before the goto. The normal return chain is via the static link in block contours and via the return label ep in procedure contours. This rippling is required to force deallocation of contours when a non-local goto is done.

```
eptr ← Π.ep;    ¢Save processor's old ep.
Π.ep ← 1st.ep;   ¢Reset Π's ep, nh, and ip from
Π.nh ← 1st.nh;   ¢those of label value.
Π.ip ← 1st.ip;
FOR i FROM 0 BY 1 TO Π.ts - 2   ¢Decr ref counts of
 DO                             ¢cells referred to
  IF is_scoped(Π.sp*[i].tag)    ¢by values in ES
   THEN                         ¢except for label
    decrc(Π.sp*[i].ep) FI OD;   ¢on top of ES.
sptr ← Π.ep*[1].sp;   ¢Pick up sp of STK word.
m ← Π.ep*[1].ts;      ¢Pick up ts of STK word.
Π.sp [0:m-1] ← sptr*[0:m-1];   ¢Copy ces to ES.
Π.ts ← m;                      ¢Set Π's ts.
FOR i FROM 0 BY 1 TO m-1       ¢Incr ref counts
 DO                            ¢of cells referred
  IF is_scoped(Π.sp*[i].tag)   ¢by values in new
   THEN incrc(Π.sp*[i].ep)     ¢ES.
 FI OD;
```

```
Π.ep ← 1st.ep;   ¢Reset Π's ep, nh, and ip from
Π.nh ← 1st.nh;   ¢those of the label value.
Π.ip ← 1st.ip;
Π.sp ← (Π.ep + 1)*.sp;   ¢Reset Π's sp and ts from
Π.ts ← (Π.ep + 1)*.ts;   ¢STK word of new top AR.
reconstruct_display;
```

```
decrc(eptr);   ¢Decr ref count of contour pointed
               ¢to by processor's old ep.
fin ← FALSE;   ¢Set loop finish flag.
WHILE eptr ≠ NIL ∧   ¢As long as eptr points to a
      ¬fin           ¢contour and not finished,
 DO                  ¢do:
  IF eptr*[0].rc = 0   ¢If ref count of contour is
   THEN                ¢zero then,
    IF eptr*[1].tag = RTL   ¢if contour has return
     THEN next ← eptr*[1].ep   ¢label, then next
                               ¢is ret label ep else next
     ELSE next ← eptr*[0].sl   ¢is static link.
    FI;
    FOR i FROM 0 BY 1 TO eptr*[-1].len -2   ¢Decr
     DO                     ¢ref count of cells referred
      IF is_scoped(eptr*[i].tag)   ¢to by values in
       THEN decrc(eptr*[i].ep);   ¢contour.
            eptr*[i] ← makeund()
     FI OD;
    IF eptr*[1].tag = STK   ¢If inaccessible con-
     THEN                ¢tour has a ces, then:
      ptr ← eptr*[1].sp   ¢For each value in the
      FOR i FROM 0 BY 1 TO eptr*[1].ts - 1
       DO                       ¢ces, if it is ref coun-
        IF is_scoped(ptr*[i].tag)   ¢table, decr
         THEN decrc(ptr*[i].ep);  ¢ref count of
              ptr*[i] ← makeund()   ¢referred to
       FI OD;                      ¢cells.
      free_ces(ptr)   ¢Now free ces.
    FI;
    free_contour(eptr);  ¢Finally free contour.
    eptr ← next  ¢Prepare for next loop cycle.
   ELSE   ¢Else contour's ref count > 0 and
    fin ← TRUE     ¢halt loop
 FI OD;
reconstruct_display;
```

Time: constant

Time: potentially unbounded

13) MAKEPROC, ip, n — Form and push a procedure value with the argument ip as the ip, n as the nh, and if n > 0 then DISP[n-1] otherwise NIL as the ep.

It is necessary to increment the reference count of the contour pointed to by the procedure value's ep.

```
IF n > 0
 THEN push(makeproc(DISP[n-1],n,ip))
 ELSE push(makeproc(NIL,0,ip))
FI;          ¢Push value.
```

```
IF n > 0
 THEN push(makeproc(DISP[n-1],n,ip));
      incrc(1st.ep)      ¢Increment ref count.
 ELSE push(makeproc(NIL,0,ip))
FI;          ¢Push value.
```

Time: constant

Time: constant

14) RESERVE

Reserve one word that will be filled in later with the return label.

With respect to the PRCCM, this instruction is a NO-OP.

```
push(makeint(0));  ¢Push a zero to reserve a word.
```

```
;   ¢This is an empty statement
```

Time: constant

Time: constant

15) ALLOC-ADJ, n

With respect to the LSM, this instruction is a NO-OP.

;

Time: constant

Allocate a contour with n+2 words (i.e., with n words for the return label and the parameters). Set its static link and nh to a copy of the ep and one less than the nh of the procedure value in the nth word of the stack (underneath all the parameter values). Leave a pointer value pointing to the contour on top of the stack. It is necessary to increment the reference count of the contour pointed to by the procedure's ep.

```
ptr ← alloc_contour (n+2);    ¢Allocate contour; set
                              ¢ptr to point to it.
ptr*.sl ← nth.ep;     ¢Set static link of contour.
ptr*.nh ← nth.nh-1;   ¢Set nh of contour.
push (makeptr (ptr,ptr));     ¢Push pointer to contour.
incrc (nth.ep);               ¢Incr ref count.
```

Time: constant + T(alloc_contour)

16) PASS, n

With respect to the LSM, this instruction is a NO-OP. There is no need to physically pass the parameters because the mes used to evaluate the actual parameters becomes the formal parameter AR for the called procedure.

Pass the n words below the top of the stack to the n consecutive words of the contour pointed to by the pointer value on top of the stack, starting at the word of the contour indexed by 2. Move the pointer value in the top of the stack n words down and pop the stack by n. Since the n parameter values are moved to the contour and are then popped from the stack, there is no net change in reference counts.

```
1st.ep*[2:n+1] ←     ¢Move the n words from the stack
    Π.sp*[Π.ts-n-1:Π.ts-2];   ¢to the contour.
(n+1)th ← 1st;       ¢Slide the pointer value down.
pop(n);              ¢Pop by n.
```

Time: proportional to n

Time: constant

17) CALL, n

Save the processor's current ip (which has already been incremented) and nh in the return label subcell of the formal parameter AR being constructed. Reset the processor's ip and nh from the called procedure value. Save the processor's ep in the dynamic link of the AR and reset the processor's ep to point to the AR. Finally, reconstruct the display. The argument n of the instruction gives the size of the return label and parameter part of the AR being constructed out of the top n+1 words of the topmost mes. Thus, the return label cell is in the nth word of the stack and the procedure value is in the (n+1)th word of the stack. The (n+1)th word will become the static link-dynamic link cell of the AR.

```
nth ← makertl (Π.nh, Π.ip);   ¢Save the processor's
                               ¢ip and nh in return label.
Π.ip←(n+1)th.ip;  ¢Reset the processor's ip from
                  ¢procedure value.
Π.nh←(n+1)th.nh;     ¢The processor's new nh is
                        ¢that of the procedure.
(n+1)th ← makeARcont(    ¢Change the procedure value
    (n+1)th.sl, (n+1)th.nh-1,  ¢into an AR control
    Π.ep );                ¢word.

Π.ep←Π.sp+Π.ts-(n+1) ¢Reset Π's ep to point to
                     ¢AR control word.
reconstruct_display;

Time: constant + T(reconstruct_display)
```

Save the processor's current ip (which has already been incremented), nh, and ep in the return label cell of the formal parameter contour. Reset the processor's ip and nh from those of the called procedure value. The processor's ep is reset to point to the new contour. The contour is pointed to by the top of the stack and the procedure value is in the second stack word. These two values are popped and the display is reconstructed. In the course of the instruction execution, there is no net change to the reference count of the contour, but the reference count of the cell pointed to by the popped procedure value's ep must be decremented. (Note that the argument n is never used in the PROCM.)

```
1st.ep*[1] ← makertl(    ¢Save processor's ep, nh
      Π.ep, Π.nh, Π.ip);  ¢and ip in return label.
Π.ip ← 2nd.ip;    ¢Reset Π's ip from procedure
Π.nh ← 2nd.nh;     ¢Π's nh is
                   ¢nh of procedure.
Π.ep ← 1st.ep;    ¢Reset Π's ep to point to new
                  ¢contour.
decrc(2nd.ep);    ¢Decrement ref count of cell
                  ¢referred to by procedure value.
pop(2); ¢Pop pointer-to-contour and procedure.
reconstruct_display;


Time: constant + T(reconstruct_display)
```

18) RETURN, n, j_1, ..., j_n Return from procedure call.

Pick up the value to be returned and decrement the processor's ts so that the top of the stack is just beneath the topmost AR (i.e., Π.sp + Π.ts = Π.ep). Reset the processor's ip and nh

Reset the processor's ip, nh, and ep from the return label of the contour pointed to by the processor's ep. It is not necessary to move the returned value because it is already in its proper

from the return label of the AR. Then reconstruct the display. Finally push the returned value into the stack. Rather than popping and pushing, the stack will be popped to where it would be after the push and the returned value is merely assigned to the top of the stack. (The arguments of the instruction are ignored in LSM.)

place in the contiguous ES. The exited contour's reference count is decremented and if the reference count goes to zero, the contour must be freed. Before freeing it, it is necessary to decrement the reference count of all cells referred to by any pointers, labels, or procedures in the contour (in this case, it is not necessary to check that the referred-to cell is not the same contour since it is already being returned to the free list). Since the contour is that of a procedure, it has no STK word, so there is no ces to free. It is also necessary to reset to undefined values all words containing pointer, label, or procedure values, the return label, or the static link. The purpose of the argument list is to indicate that there are n words containing reference countable data and that their displacements into the contour are j_1, ..., and, j_n. Finally the display is reconstructed.

a) RETURN, n, j_1, ..., jn The returned value is an integer or boolean value.

```
val ← 1st;    ¢Save returned value.
Π.ts ← Π.ep-Π.sp + 1;  ¢Pop all but one word of AR.
Π.ip ← (Π.ep + 1)*.ip; ¢Reset processor's ip & nh
Π.nh ← (Π.ep + 1)*.nh; ¢from return label.
Π.ep ← Π.ep*.dl;   ¢Reset processor's ep from dl.
reconstruct_display;
1st ← val;  ¢Put returned value in top of stack.
```

Time: constant + T(reconstruct_display)

```
ptr ← Π.ep;   ¢Save processor's current ep pointing
              ¢to exited contour.
Π.ip ← ptr*[1].ip;  ¢Reset processor's ip, ep, nh
Π.nh ← ptr*[1].nh;  ¢from return label
Π.ep ← ptr*[1].ep;
ptr*[1] ← makeund();   ¢Set ret label to undefined
IF decrc(ptr) = 0  ¢Decr contour's ref count and
 THEN   ¢if ref count goes to zero then for
  FOR i FROM 1 BY 1 TO n  ¢each ref countable
   DO                ¢word in the contour: Decr
    decrc(ptr*[ji].ep);  ¢ref count of ptd-to cell.
    ptr*[ji] ← makeund()   ¢Set word to undefined.
   OD;
  decrc(ptr*[0].sl);    ¢Decr ref count of contour
                        ¢ptd to by static link, and
  ptr*[0] ← makeund();  ¢set sl word to undefined,
  free_contour(ptr)   ¢and free exited contour.
 ELSE   ¢Else exited contour has ref count >0...
  incrc(Π.ep)    ¢There is an additional pointer
FI;   ¢pointing to contour that processor's ep now
                ¢points to.
reconstruct_display;
```

Time: constant + proportional to n + T(reconstruct_display) + T(free_contour)

b) RETURNP, n, j_1, ..., j_n The returned value is a pointer, label, or procedure. A lifetime check must be performed.

```
IF 1st.ep ≥ Π.ep*.dl   ¢Lifetime check; if not
 THEN lifetime_error FI;  ¢OK then error.
val ← 1st;              ¢Otherwise as in a)
Π.ts ← Π.ep-Π.sp + 1    ¢above
```

No change.

¢Exactly as for a) above

```
Π.ip ← (Π.ep + 1)*.ip;
Π.nh ← (Π.ep + 1)*.nh;
Π.ep ← Π.ep*.dl;
reconstruct_display;
1st ← val;
```

Time: constant + T(reconstruct_display)

Time: as in a)

19) ALLOC-ADJ-ENTER, n Block entry.

Push an AR with n words for the identifiers. Set its static link and dynamic link to copies of the processor's current ep, and reset the processor's ep to point to the new AR. Increment the processor's new ep into the new top display element.

```
push (makeARcont(          ¢Push an AR control word
  Π.ep, Π.nh, Π.ep));      ¢with Π.ep as sl and dl.
Π.ep ← Π.sp + Π.ts-1;   ¢Set processor's ep to point
                         ¢to AR control word.
Π.ts ← Π.ts + n;         ¢Push in n blank words.
DISP[nh] ← Π.ep;     ¢Copy processor's ep into new
                     ¢display element.
Π.nh ← Π.nh + 1;     ¢Increment processor's nh.
```

Time: constant

Allocate a contour with n words for identifiers. Set its static link to a copy of the processor's current ep and reset the processor's ep to point to the new contour. Increment the processor's nh by 1 and copy its new ep into the new top display element. There is no net change in reference count to either the old or the new contour.

```
ptr ← alloc_contour (n+2);   ¢allocate contour of
                 ¢size n+2;     set ptr to point to it.
ptr*.sl ← Π.ep;    ¢Set sl of contour.
ptr*.nh ← Π.nh;    ¢Set nh of contour.
Π.ep ← ptr;    ¢Reset processor's ep.
DISP[nh] ← ptr;    ¢Set new DISP element to point
                   ¢to new contour.
Π.nh ← Π.nh + 1;    ¢Increment processor's nh.
```

Time: constant + T(alloc_contour)

20) EXIT, n, j_1, ..., j_n Exit from a block.

EXIT has two independent sets of suffixes:

1) The first, either L or none, is used to indicate whether or not the block declares at least one label constant.
2) The second, either P or none, is used to indicate the type of the value returned by the block, as with VAL.

In LSM, only the second suffix is significant, as it indicates whether or not a lifetime check will be needed on the value being returned.

Pick up the value to be returned and decrement the processor's ts so that the top of the stack is where it would be if the top AR were popped and then the returned value were pushed. Reset the processor's ep from the dynamic link of the popped AR. Decrement the processor's nh, thus popping the display by 1. Finally, assign the returned value to the top of the stack (The arguments of the instruction are ignored in LSM.).

In PRCCM, only the first suffix is significant, as it is used to indicate whether or not the contour being exited has an STK word and thus a ces. In PRCCM, the first suffix is not significant because the value being returned is properly in position on top of the ES, and no reference count updating need be performed as a result of this value.

Reset the processor's ep from the static link of the exited contour. The exited contour's reference count is decremented. If the reference count goes to zero, using the arguments as in RETURN, the reference counts of cells referred to by pointer, label, and procedure values in the exited contour are decremented. Also the words containing reference countable values and the static link are reset to undefined. If the contour has a ces, then the reference counts of the cells referred to by its reference countable values are decremented and the ces is freed. Finally the exited contour is freed. If, however, the exited contour's reference count has not gone to zero, then the reference count of the contour the processor's ep now points to is incremented. In either case, the processor's nh is decremented by 1 to pop the display by 1.

a) EXIT, n, j_1, ... j_n The returned value is an integer or boolean value, and the block declares no labels.

```
val ← 1st;   ¢Save returned value.
Π.ts ← Π.ep-Π.sp + 1;   ¢Pop all but one word of AR.
Π.ep ← Π.ep*.dl;   ¢Reset processor's ep from dl.
Π.nh ← Π.nh-1;   ¢Decrement processor's nh.
```

```
ptr ← Π.ep;   ¢Save processor's ep pointing to
              ¢exited contour.
Π.ep ← ptr.sl;   ¢Reset processor's ep from sl.
IF decrc(ptr) = 0   ¢Decr contour's ref count and
```

```
1st ← val;   ¢Move returned value to top of stack.
```

Time: constant

```
 THEN    ¢if ref count goes to zero then for
  FOR i FROM 1 BY 1 TO n   ¢each ref countable
   DO    ¢word in exited contour: Decr ref
    decrc(ptr*[j_i].ep);     ¢count of ptd-to cell.
    ptr*[j_i] ← makeund()    ¢Set word to undefined.
   OD;
   ptr*[0] ← makeund();   ¢Set sl word to undefined.
   free_contour(ptr);     ¢Free contour.
  ELSE    ¢Else exited contour has ref count >0...
   incrc(Π.ep)  ¢There is an additional pointer
FI;        ¢pointing to contour that Π's ep
           ¢now points to.
Π.nh ← Π.nh-1;    ¢Decrement processor's nh.
```

Time: constant + proportional to n + T(free_contour)

b) EXITP, n, j_1, ..., j_n The returned value is a pointer, label, or procedure value, and the block declares no labels.

A lifetime check is required.

```
IF 1st.ep ≥ Π.ep*.dl   ¢Lifetime check; if not
 THEN lifetime_error FI;  ¢OK then error else...
val ← 1st;                ¢Rest as a) above.
Π.ts ← Π.ep-Π.sp + 1;
Π.ep ← Π.ep*.dl;
Π.nh ← Π.nh-1;
1st ← val;
```

Time: constant

No change.

¢Exactly as a) above

Time: as in a)

c) EXITL, n, j , ..., j

The returned value is an integer or boolean value, and the block declares at least one label constant.

If the exited contour is to be freed, its ces must be scanned for reference countable values. After decrementing reference counts and resetting the reference countable values to undefined, the ces is freed.

¢Exactly as in a) above.

```
ptr ← Π.ep;   ¢As in a) above except as noted.
Π.ep ← ptr.sl;
IF decrc(ptr) = 0
 THEN
  FOR i FROM 1 BY 1 TO n
   DO
    decrc(ptr*[j_i].ep);
    ptr*[j_i] ← makeund()
   OD;
  ptr*[0] ← makeund();
  sptr ← ptr*[1].sp;   ¢Get pointer to ces.
  FOR i FROM 0 BY 1 TO ptr*[1].ts - 1  ¢For each
   DO       ¢ref countable value in ces,
    IF is_scoped(sptr*[i].tag)   ¢decrement ref
     THEN                        ¢count and set to
      decrc(sptr*[i].ep);        ¢undefined.
      sptr*[i] ← makeund()
   FI OD;
  free_ces(sptr);   ¢Free ces.
  free_contour(ptr);
 ELSE
  incrc(Π.ep)
FI;
Π.nh ← Π.nh - 1;
```

Time: constant

Time: constant + proportional to n + T(free_contour) + potentially unbounded to scan ces + T(free_ces)

d) EXITLP, n, j_1, ..., j_n The returned value is a pointer, label, or procedure value, and the block declares at least one label constant.

```
¢Exactly as in b) above.

Time: constant
```

```
¢Exactly as in c) above.

Time: as in c)
```

21) HOPTO, ip Reset the processor's ip to the ip argument.

```
Π.ip ← ip

Time: constant*
```

```
Π.ip ← ip

Time: constant*
```

22) HOPTOF, ip Pick up and pop the top word of the stack. If it is FALSE, reset the processor's ip to the ip argument.

```
val ← 1st;     ¢Pick up top word.
pop;           ¢Pop one word.
IF val = FALSE   ¢If word is FALSE, do
 THEN Π.ip ← ip FI;     ¢HOPTO.

Time: constant*
```

```
val ← 1st;     ¢Pick up top word.
pop;           ¢Pop one word.
IF val = FALSE   ¢If word is FALSE, do
 THEN Π.ip ← ip FI;     ¢HOPTO.

Time: constant*
```