1063

Bell Laboratories

Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication (see GEI 13.9-3)

Title- A Description of How UNIX Fortran Works

Date- April 10, 1975

TM-75-8234-3

Other Keywords- UNIX Fortran

Author(s)	Location and Room	Extension
I. A. Winheim	MH 2C-425	MH 2771

Charging Case- 49170-110

Filing Case- 40952-001

ABSTRACT

Interfaces between UNIX Fortran and the operating system cause problems not encountered while interfacing between other languages and the operating system. These problems are caused because UNIX Fortran runs interpretively and is therefore different in many respects. This memorandum will describe the main differences between C and Fortran, and will describe how to interface Fortran with the operating system.

Pages Text 8 Other 14 Total 22

No. Figures 25 No. Tables 0 No. Refs. 0

Address Label

E-1932-C (6-73)

SEE REVERSE SIDE FOR DISTRIBUTION LIST

BELL TELEPHONE LABORATORIES, INC.

TM-75-8234-3

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

10 REFERENCE COPIES

+BALDWIN,GEORGE L
BARTLETT,WADE S
BLINN,JAMES C
CRUM,TED A
+FEDER,J
FREEMAN,K GLENN
HANSEN,MRS G J
+HOLLIDAY,SARETTA
+JENSEN,PAUL D
KEESE,W M
KELLY,L J
KERNIGHAN,BRIAN W
+KLEIN,MISS R L
+LONDON,H S
+MARANZANO,JOSEPH F
MORGAN,S P
+RALEIGH,THOMAS M
+RITACCO,J E
+ROCHKIND,M M
+SPENCE,NORMAN A
+TAGUE,BERKLEY A
+VOGEL,GERALD C
23 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

ALBERTS,BARBARA A
ALT,MISS DOROTHY L
AMOSS,JOHN J
ANDERSON,MS K J
ARCHER,RUSSELL E JR
ARNOLD,S L
BAUER,MS H A
BAYER,DOUGLAS L
BECKER,R A
BELL,R E
BERTH,R P
BIRCHALL,R H
BIREN,MRS IRMA B
BLUM,MRS MARION
BRANDT,RICHARD B

COVER SHEET ONLY TO

BROWN,COLIN W
BURNETTE,W A
BZOWY,D E
CALESSO,GIULIO L
CANADAY,RUDD H
CARDOZA,WAYNE M
CARRAN,J H
CARR,DAVID C
CHIANG,T C
CHRIST,C W JR
COKE,MISS E U
COLE,LOUIS M
COOK,THOMAS J
COPP,DAVID H
COREY,D A
COSTON,WALTER P
COULTER,J REGINALD
CRAGUN,D W
CRANE,RODERICK P
CUNNINGHAM,STEPHEN J
D STEFAN,D J
DAVIS,D R
DE JAGER,D S
DOLAN,MRS MARIE T
DOLOTTA,T A
DONOFRIO,L J
DOWD,PATRICK G
EDELSON,D
EDMUNDS,T W
EITELBACH,DAVID L
ERRICHIELLO,PHILIP M
ESTOCK,R G
FAULKNER,R A
FLEISCHER,HERBERT I
FLYNN,MS M L
FORTNEY,V J
FOUNTOUKIDIS,A
FRANK,H G
FREEDMAN,M I
FREEMAN,R DON
FREIDENREICH,MRS B
GANNON,T F
GATES,G W
GERGOWITZ,E B
GIBSON,H T JR
GLASER,W A
GLASSER,ALAN L
GOLDSMITH,L D
GORMAN,JAMES E
GRAVEMAN,R F
GREENBAUM,H J
GRUCHOWSKI,M P
HAIGHT,R C
HAMILTON,MRS L
HAMILTON,PATRICIA
HARRIS,CHARLES S
HELD,RICHARD W
HOYT,WILLIAM F
HUMCKE,D J
HUPKA,MRS FLORENCE
HYMAN,B

COVER SHEET ONLY TO

IMAGNA,CLYDE P
IVIE,EVAN L
JACKOWSKI,D J
JAKUBEK,RAYMOND J
JOHNSON,STEPHEN C
JONES,B R
KAMINSKY,MICHAEL L
KAPLAN,A E
KAUFMAN,LARRY S
KAYEL,R G
KETCHLEDGE,B A
KEVORKIAN,DOUGLAS E
KNUDSEN,DONALD B
KORNEGAY,R L
LA CAVA,JOSEPH L
LAMBERT,MRS C A
LANDAU,C N
LESSEK,PETER V
LEVINE,P R
LICWINKO,J S
LINDERMAN,J
LIST,WILLIAM H S
LONG,DAVID W
LONG,P F
LORENC,ANTHONY
LUDERER,GOTTFRIED W R
LUDWIG,J J
LYCKLAMA,HEINZ
LYONS,T G
MACHOL,R E JR
MARGOTTA,JUSTINE P
MASHEY,JOHN R
MASHEY,MRS R
MC ILROY,M DOUGLAS
MENZEL,MRS F S
METZLER,MRS HELEN M
MILLSPAUGH,J A JR
MILNE,D C
MOLINELLI,JOHN J
MOLLENAUER,J F
MORGAN,DENNIS J
MORR,PHILLIP L
NETZEL,T L
NOLL,J C
NOWITZ,D A
O CONNELL,T F
O NEILL,DENNIS M
OSSANNA,J F JR
PALUCK,MRS M B
PATEL,C K N
PEARLMAN,MS M E
PEKARICH,S P
PERDUE,R J
PEREZ,MRS CATHERINE D
PERNESKI,A J
PETERSON,RALPH W
PETERSON,T G
PHILLIPS,S J
PILLA,MICHAEL A
POPPER,C
PRYOR,ROGER W

COVER SHEET ONLY TO

PUERLING,BRUCE W
RIDDLE,GUY G
ROBINSON,HERMAN E
ROCHKIND,M J
ROMITO,LETITIA J
ROM,DOUGLAS B
RONKOVITZ,FRANK J JR
ROOME,WILLIAM D
ROSENFELD,PETER E
ROSLER,LAWRENCE
SATZ,L R
SCHLEGEL,C T
SCRIBNER,NEAL M
SIDOR,DONALD J
SILVERSTEIN,STEVEN M
SIX,FREDERICK B
SMITH,D W
SOUTHERN,MS M J
STAMPFEL,JOHN P
STARRICK,
STEVENSON,D E
STROHECKER,CARY A
STURMAN,JOEL N
SWANSON,GEORGE K
TANIS,MRS PATRICIA J
THOMPSON,BERNARD E
THOMPSON,MRS M
TINLEY,DENNIS J
UNDERWOOD,R W
VAN ORNUM,JAMES H
VARNEY,ROBERT C
VIGGIANO,F A
WALKER,MISS E A
WANDZILAK,P D
WATKINS,G T
WEBB,FRANCIS J
WEHR,L A
WEYTHMAN,J E
WHIPPLE,JOHN H
WHITE,RALPH C JR
WILSON,GEOFFREY A
WINCKELMANN,W A
WINHEIM,MISS IRENE A
WOLFE,ROBERT M
WOOD,J L
ZEPP,HILTON E
184 NAMES

+ NAMED BY AUTHOR

> CITED AS REFERENCE SOURCE

MERCURY SPECIFICATION.....

COMPLETE MEMO TO:

COVER SHEET TO:
823-SUP

UNPL = UNIX/PROGRAMMING LANGUAGES

HARDISON,D L
MH 2C121

TM-75-8234-3
TOTAL PAGES 22

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.

PLEASE SEND A COMPLETE COPY TO THE ADDRESS SHOWN ON THE OTHER SIDE
NO ENVELOPE WILL BE NEEDED IF YOU SIMPLY STAPLE THIS COVER SHEET TO THE COMPLETE COPY.
IF COPIES ARE NO LONGER AVAILABLE PLEASE FORWARD THIS REQUEST TO THE CORRESPONDENCE FILES.

Bell Laboratories

subject: A Description of How UNIX Fortran Works

date: June 25, 1975

from: I. A. Winheim
TM-75-8234-3

MEMORANDUM FOR FILE

I. Introduction

Calls from UNIX Fortran programs to the UNIX operating system cause problems not encountered when calling UNIX in other languages. These problems arise because UNIX Fortran runs interpretively. This memorandum will describe the main differences between C and Fortran, and will describe the interface between the UNIX Fortran Interpreter and the UNIX operating system.

II. General Description of how Fortran Works

When a Fortran program is compiled the result is an executable file, a.out, which contains the Fortran Interpreter, an Interpretation List (list of primitive routines to interpretively execute the Fortran statements), and any needed assembly language subroutines. If several Fortran fragments are compiled and loaded together they will each have a separate Interpretation List. A simple Fortran program having one call to the subroutine sum is shown in figure A1. The sum subroutine is shown in figure A2.

The Interpretation List compiled for this Fortran program is shown in Figure A3 and consists of a table of addresses which specify to the Interpreter the subroutines and their arguments to be executed to interpret the Fortran. Subroutines which implement primitive operations such as addition, subtraction, assignment etc. are available to the compiler.

The Interpretation List (called main: in this case because it is compiled as a main program rather than as a subroutine) can be divided into three parts. The first section is the list of primitive operations to be performed to interpret the Fortran (Section A of Figure A3). The next section contains the compile time constants and variable space (Section B of Figure A3). The last section (not shown in this example because a "main" Interpretation List does not return values) consists of space required within a subroutine Interpretation List for linkage and for returning values.

There are three types of entries in the Interpretation List: two types for primitive operations (that do not require calls to other more basic operations), and one type for subroutine calls that require calls to other operations. The two types of entries for primitive operations separate monadic (single operand) from dyadic (double operand) operations. The first type (monadic) involves fetching or storing single arguments on the stack or in memory, and the second type (dyadic) involves binary operations. The Interpretation List format for monadic primitive operations consist of the address of the assembly language subroutine performing the move, and the memory address of either the argument for a fetch or the destination for a store. The Interpretation List format for a binary operation consists of the address of the assembly language subroutine implementing the operation. There are no arguments associated with the binary operations because the stack is used as a group of accumulator registers. The operation is therefore performed on the last two entries in the stack. The subroutine call format is more complex. The first argument is the address of the call primitive. The second argument is the address of the subroutine to be called by the call primitive. The third and fourth arguments give the location and length respectively of an Argument List which contains the parameters to be passed in the call. The final argument indicates the number of bytes to be returned from the called subroutine which is a function of the type and precision of the returned argument.

An assembly language Fortran subroutine must obey the following rules (see Figure A2). The entry point must be the name of the subroutine with a period appended to the end. The first word of the subroutine is the address of an area reserved in the subroutine for returning values. The second word of the subroutine must contain the address of the real entry point of the subroutine (address of the executable code). It is this entry that enables linking to the Interpretation List.

In order to explain in greater detail how Fortran is executed, and to show the linkage conventions, this memorandum will step through the execution of the Fortran program in Figure A1.

III. The Interpretation List

The Interpretation List is a list of addresses, which point to subroutines in the Fortran Library which perform primitive operations, and the addresses of arguments used by these primitive operations. This list of addresses constitutes an executable program generated by the Fortran Compiler in an Interpretation Language.

The Interpretation Language is a zero-address language which uses the stack as the accumulator. In a zero-address language all binary operations (add, subtract, etc) are done on the stack. A zero-address instruction for such an operation does not require

the address of either operand. For example an add would take the contents of the cell pointed to by the SP (Stack Pointer), add it to the contents of the cell above it (these two cells are the latest values pushed on the stack), and would then place the result in the stack cell of the second operand. This cell (containing the result of the add) is now the first operand of the next primitive operation. Since the Fortran program being used as an example does not require any binary operations, an illustration of this type of instruction is shown separately in Figure A4 for the iad4 (integer add for four byte operands) primitive operation.

The Interpretation Language instructions used to execute the Fortran program of Figure A1 are all of the fetch or store type and these must have a memory address associated with them. A load type instruction (i.e. rval4,lval) must provide a source address from which to fetch the argument being pushed on the stack. A store type instruction (i.e. gmv4,stsp,stst) requires a destination address for popping a word off the stack into a memory location. A move from one location in memory to another location (other than the stack) requires two moves in a zero-address language. One pushes the word on the stack and the other pops the word off the stack to the appropriate memory location.

The call primitive differs from any other Interpretation Language primitive in that the code to be executed (i.e. the called subroutine) is not included as one of the Fortran Library. The called subroutine is in the form of another Interpretation List, remembering that Interpretation language instructions are addresses of pieces of code to be executed. This other Interpretation List is in many cases one instruction long, and would consist of the second word of the subroutine (the address of the executable code). The code to be executed may reside anywhere in the addressable memory of the program, since there is no restriction saying the executable code must start at the third entry in the subroutine. (Note that on returning from the subroutine it's location in memory is unimportant since the stack contains addresses within both the main Interpretation List and the executable code of the subroutine.) During the execution of the call primitive, register r4 (the program counter) points to the second word of the called subroutine, at this point the subroutine is linked to the Interpreter to be executed. The Interpreter uses this address as if it were an entry in an Interpretation List and executes the code pointed to by this address. This second Interpretation List may point to even another Interpretation List if the subroutine had been written in Fortran since Fortran statements must be executed via an Interpretation List. Figure A5 illustrates the flow of execution for a subroutine in the Fortran Library, for a user subroutine written in assembly language, and for a user subroutine written in Fortran. The executable code for the above cases is illustrated as residing anywhere in memory.

IV. Execution of Interpretation List

The use of the Interpretation List can be shown by following the execution of the instructions in the list. Figure A5 illustrates the state of memory prior to the execution of rval4, the first entry in the list. At execution time the Fortran Interpreter uses general register r4 to step through the Interpretation List. It is the program counter for the Interpreter. While executing the rval4 instruction, r4 is moved to point to the argument for the instruction, in this case, the address of c0. The rval4 mnemonic comes from the term "right hand value" meaning the contents (not the address) of the argument is important. Therefore, the contents of c0 are pushed on the stack. The operation in this instance pushes four bytes (remember that integers are stored in two words) located at c0 onto the stack. R4 is then incremented to point to the address of the next instruction, gmv4 (See Figure A6). While executing gmv4, r4 points to the associated argument (destination address), in this case sec_. This instruction pops 4 bytes off the stack, using the last one in first one off (LIFO) convention, and places the contents at location sec_. R4 is then incremented to point to the next entry in the Interpretation List. Figure A7 shows the results of executing gmv4. The two primitive operations just completed correspond to statement 1 of the Fortran program:

```
sec = 10
```

The next statement in the Fortran program is the subroutine call. An Argument List must be created before the subroutine call is made. Preparation of the Argument List is started by executing the stsp primitive operation. This saves the contents of the current stack pointer (SP) in the Argument List at,ft+0 (see Figure A8). Before completing the stsp instruction, r4 is incremented to point to the next entry in the Interpretation List. The addresses of the arguments to be passed in the subroutine call are then placed in the Argument List. Figures A9 and A10 show the results of executing the lval and stst instructions respectively to store the address of one of the arguments in the Argument List. Lval treats the argument following it as a left value expression (i.e. treated as an address) and pushes it onto the stack. Stst then pops the stack and stores the value popped at the location specified by the argument to stst. This argument is the address of the next consecutive entry in the Argument List. In this case the Argument List is completed by executing lval and stst four times, once for each argument being passed in the call statement of the Fortran program:

```
call sum(1,2,3,4)
```

Figure A11 illustrates the Argument List prior to executing the subroutine call.

A summary of the execution so far will be made before discussing the execution of the call instruction.

1) An Interpretation List, containing addresses of assembly language subroutines and arguments, is generated when a Fortran program is compiled. This list is actually a Interpretation Language program to be executed by the Fortran Interpreter.

2) General register r4 is used at execution time by the Fortran Interpreter to step through the Interpretation Language program. It is the Interpreter's Program Counter.

3) To execute a Fortran subroutine call, an Argument List containing the addresses of the arguments being passed to the called subroutine is generated at a location labeled ft+0. (This fact is useful when using the debugger since the internal name ft+0 corresponds to a label, argp+2, in the symbol table.) On subsequent calls, Argument Lists are generated starting at the same location, ft+0.

4) The stack pointer is saved at the time the Argument List is generated so that information stored on the stack by the call primitive is accessible to the retrn primitive.

V. Execution of Subroutine Call

Figure A12 shows the start of the call. An Argument List has been established and the subroutine, sum , exists in the format previously described. The steps in executing the call primitive are as follows:

1) The address of the first entry in the called subroutine (the address of the return value) is placed in r0.

2) The contents of r3 are saved on the stack and replaced with the second argument to call , the address of the Argument List(see Figure A14).

3) The address of the argument count, the third argument to call , is placed on the stack (Figure A15).

4) The next step (Figure A16) puts the contents of r0 (the address of the first location in the subroutine) into r4.

5) Figure A17 shows that the contents of the first location in the subroutine (the address of the area reserved in the subroutine for return values) is saved on the stack and r4 is incremented to point to the address of the code to be executed.

6) Figure A18 shows that at the completion of the call prim-

itive the stack points to both the Interpretation List and the called subroutine. The retrn primitive operation uses the information on the stack to get back to the calling Interpretation List via the address of the argument count and to return data from the subroutine via the address of the return values. The subroutine has access to the arguments it was passed through register r3 (the address of the Argument List).

7) Register r3 must be saved before being used since r3 points to the stack pointer stored in the first location of the Argument List. The saved stack pointer directs the retrn primitive to the information it needs.

VI. Execution of Subroutine Return

Upon completion of the subroutine a retrn is executed. This returns execution to the subroutine's caller. It is important to clear the stack of any intermediate data placed there by the subroutine before completing the subroutine since retrn expects to find only the saved r3, the address of the argument count, and the LV of the function. Figures A20 and A21 show how the stack is cleared of these values, and how the resulting contents of registers r0, r1, r3, and r4 are affected. The remaining steps in the retrn are as follows:

1) Clear the stack of anything stored there between the starting of the Argument List and the execution of the call by restoring the stack pointer from the SP saved in the Argument List (See Figure A22).

2) Register r4 is incremented to point to the return byte count. This value is then stored in general register r0. The byte count in this case indicates that nothing is being returned (see Figure A23) .

The return byte count would be four if the Fortran program used the call instruction like:

```
total = sum(1,2,3,4)
```

where the location provided to receive the return values was declared as integer. To demonstrate the method of returning a value to the calling function steps 3 and 4 and Figures A24 and A25 will assume the call made was similar to the call assigning sum to "total" as above.

3) Figure A24 shows that the return byte count, r0, and the address of the return value, r1, are added together making r1 point to the high order byte of the last value to be returned. If one integer value is returned the byte count would be four and the sum of r0 and r1 would make r1 point to the high order byte of the return value.

4) Figure A25 illustrates two is subtracted from the byte count and that r1 is decremented to point to the end of the first word of the value being returned, which is then pushed onto the stack. This process is repeated until the byte count is less than zero.

This completes the retrn and r4 is incremented to point to the next entry in the Interpretation List. In the case where values are returned the Interpretation List would contain the Interpretation Language instructions that pop the values from the stack and place them in the proper memory locations.

VII. Conclusion

A summary of the salient points of the discusion follows.

1) When using constants or variables in Fortran, remember that Fortran stores integers in two words, and uses different precisions for other types of objects then the C language uses.

2) When a Fortran program is compiled a Interpretation List containing zero-address instructions is generated.

3) At execution time the Fortran Interpreter uses general register r4 to step through the Interpretation List. Prior to completing the execution of the present instruction r4 is incremented to point to the next instruction in the Interpretation List.

4) When a subroutine call is executed an Argument List of the addresses of arguments being passed is created. This list always starts at ft+0, which corresponds to the label argp+2 in the symbol table.

5) The format of the call primitive in the Interpretation List has four arguments:
   a) the address of the subroutine
   b) the starting address of the Argument List
   c) the length of the Argument List
   d) the number of bytes to be returned

6) The subroutine being called must have the following format:
   a) the first word contains the address of the ares in the subroutine reserved for returning values
   b) the second word contains the address of the code to be executed

7) When executing the call primitive for an assembly language subroutine, r4 points to the second location in the subroutine. The subroutine is, at this point, linked to the

Interpretation List. The effect of this linking in of the subroutine expands the Interpretation Language instruction repertoire to include a new primitive operation.

8) R3 is used as a pointer to the Argument List during the call and must point to the same location on the return.

9) When a call is executed the following are saved on the stack:

a) the contents of r3 before the call
b) the address of the argument Count
c) the address of the area in the subroutine for returning values

Since the address of the argument count points to a location in the Interpretation List and the address of the return values points to a location in the subroutine the stack enables access to either.

10) If the subroutine is a user supplied assembly language program, SP, r4, and r3 must be saved on entry and exit from the subroutine.

11) Upon executing the retrn primitive, the stack is cleared of the three values placed there by the call by popping

a) the address of the return values into r1
b) the address of the argument count into r4
c) The contents of the saved r3 into r3

VIII. Acknowledgement

I would like to thank T. M. Raleigh for his help in unravelling how UNIX Fortran works.

I. A. Winheim

MH-8234-IAW-nroff

atts.:
Figures A1-A25

```
integer sec
        sec = 10
        call sum(1,2,3,4)
        stop
        end
```

FORTRAN PROGRAM
FIG. A1

```
/       example of UNIX Fortran
/       calling interface to machine code
/       this example is a function that
/       returns the single precision
/       sum of all of its single precision arguments.
/       for example:
/               f = sum(1.,2.,3.)
/       sets f to 6.

.globl  sum.                            / definition of entry
.globl  retrn                           / reference of return

sum.:                                   / entry point
        value                           / location of return value
        .+2                             / pointer to execution code
        setf
        mov     *2(sp),r0               / arg count
        mov     r3,r1                   / r3 points to arg list
        tst     (r1)+                   / zeroth arg is old r3

        clrf    fr0                     / start of actual function
1:
        addf    *(r1)+,fr0              / add in each argument
        sob     r0,1b                   / end loop

        movf    fr0,value               / make returned value available
        jmp     retrn                   / actual return

.bss
value:  .=.+4                           / space for return value

/ synopsis:
/       1. save registers r3, sp
/       2. arg list (pointers to values)
/          begins at 2(r3)
/       3. entry name is name of function
/          followed by "."
/       4. first word after entry point is
/          location of return value. this is
/          true for both functions and subroutines
/       5. second word after entry point is
/          pointer to pdp-11 code body
/       6. return is expedited by a jump to
/          the globl routine "retrn"
```

THE SUM SUBROUTINE
FIG. A2

```
main:
        rval4; c0
        gmv4; sec_
        stsp; ft+0.
        lval; c1
        stst; ft+2.
        lval; c2
        stst; ft+4.
        lval; c3
        stst; ft+6.
        lval; c4
        stst; ft+8.
        call; sum.; ft+0.; 4.; 0
        stop; 0
        stop; 0
        .even
```

sec = 10

call sum(1,2,3,4)

A

Stop
end

```
c0:
        0
        12
c1:
        0
        1
c2:
        0
        2
c3:
        0
        3
c4:
        0
        4
```

Constants (double word integers)

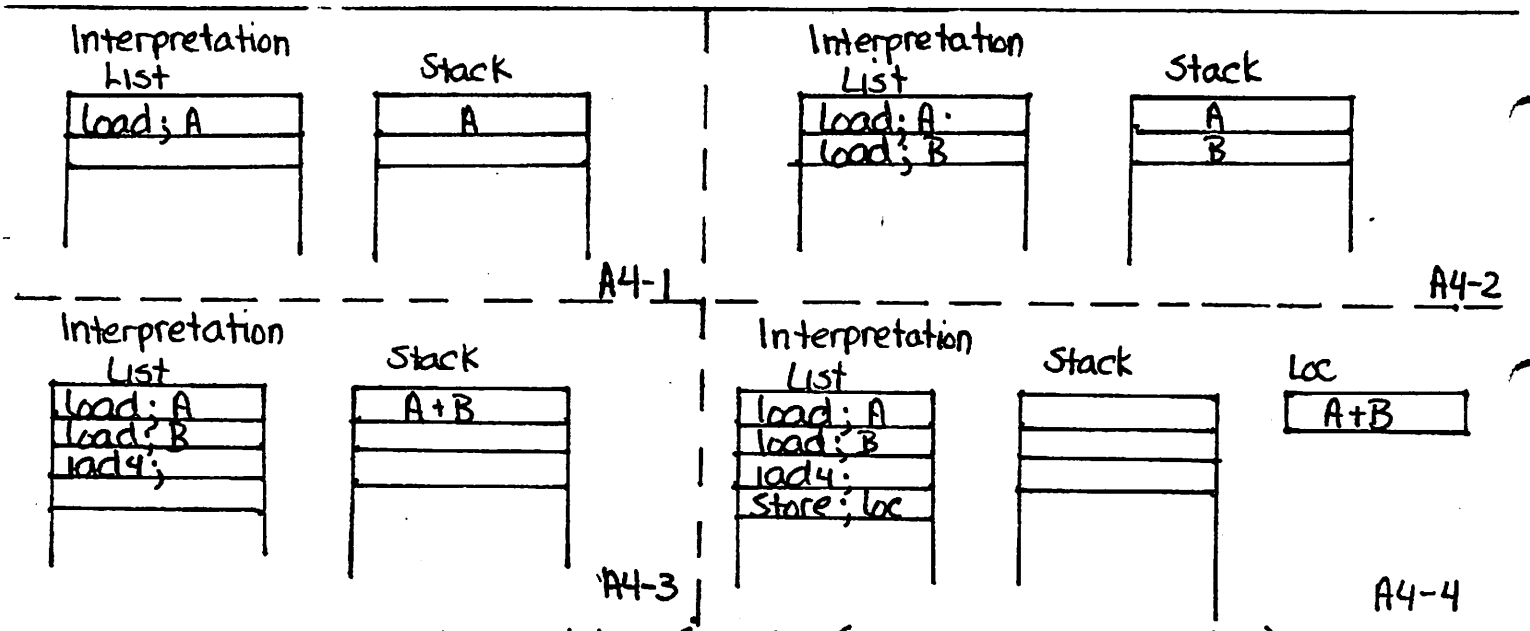
B

```
.bss
ft:     .=.+10.
base:
sec_ = base+2.
.=.+6.
.text
.globl  main
```

space for 5 arguments

2 words for sec_ (variable space)

INTERPRETATION LIST
FIG. A3



Execution of iad4 (integer add 4 bytes)

FIG A4

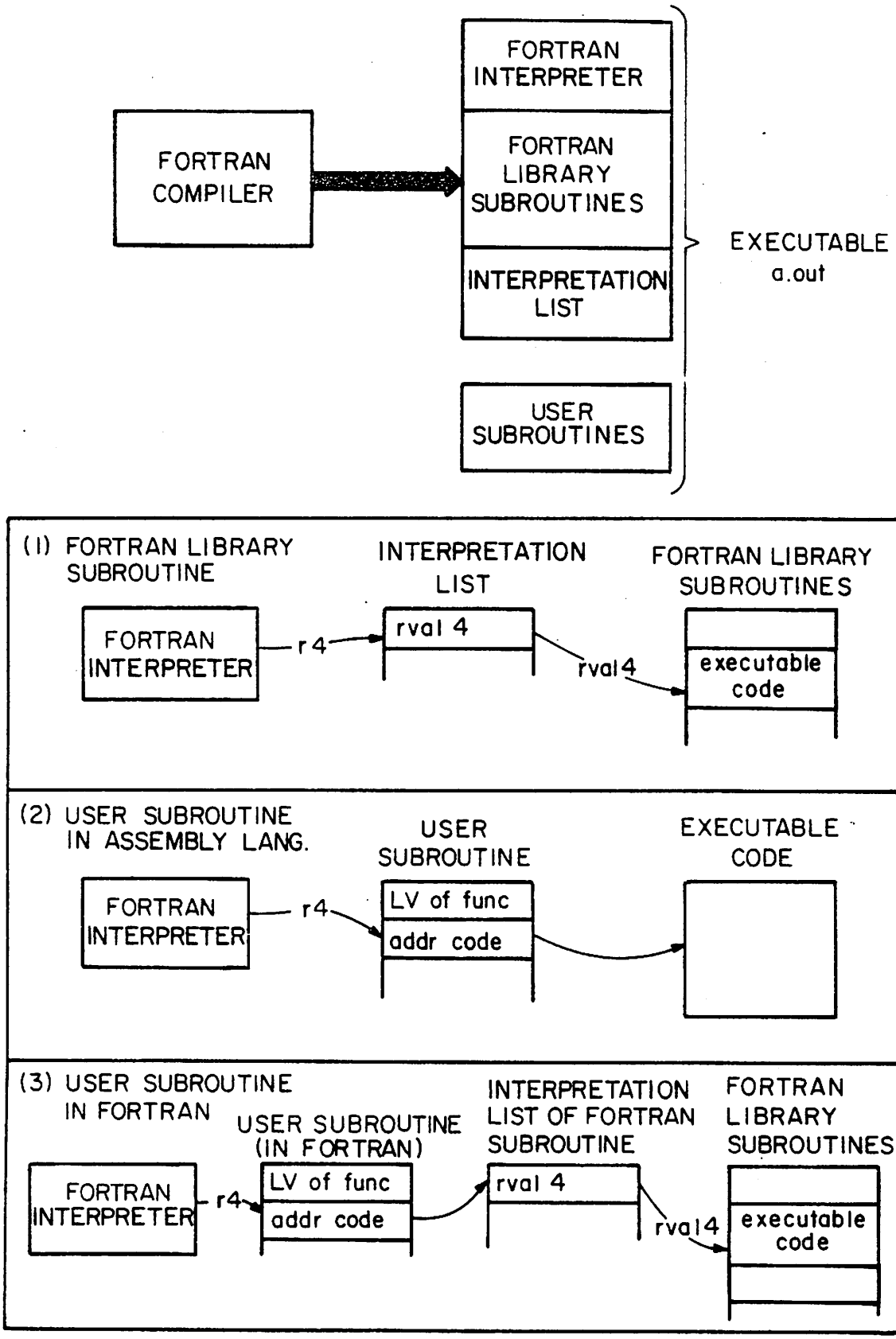
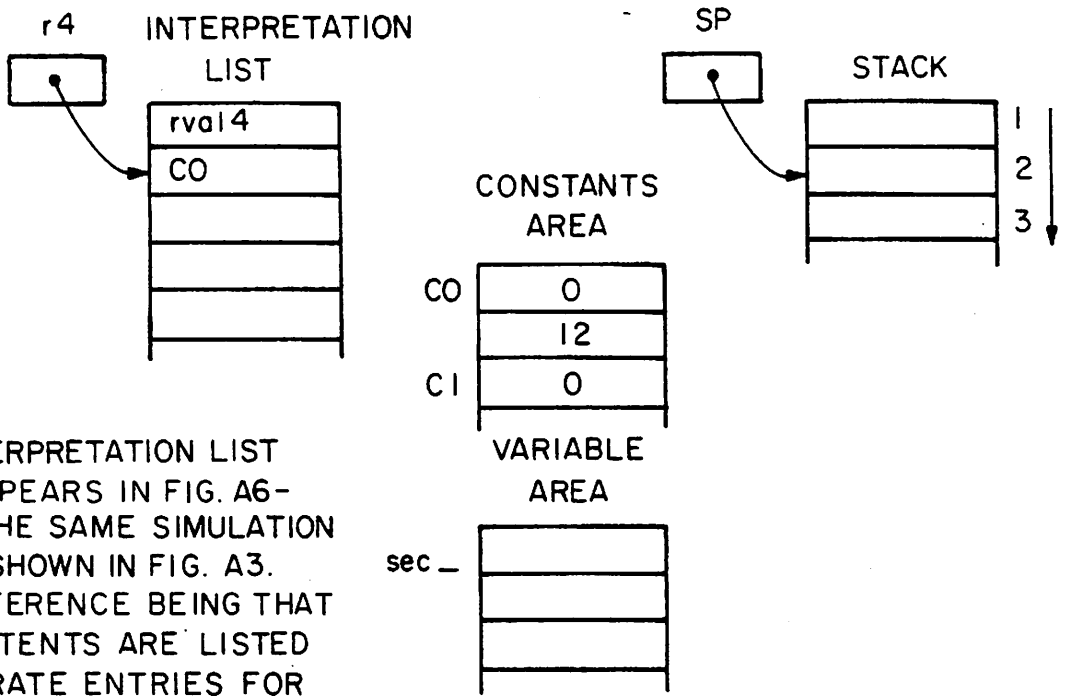


FIG. A5 THE FLOW OF EXECUTION FOR THREE CASES -(r4 IS PROGRAM COUNTER)



NOTE:
THE INTERPRETATION LIST AS IT APPEARS IN FIG. A6-A25 IS THE SAME SIMULATION LIST AS SHOWN IN FIG. A3. THE DIFFERENCE BEING THAT THE CONTENTS ARE LISTED AS SEPARATE ENTRIES FOR EASE OF READING.

FIG. A6 BEFORE EXEC OF rval4

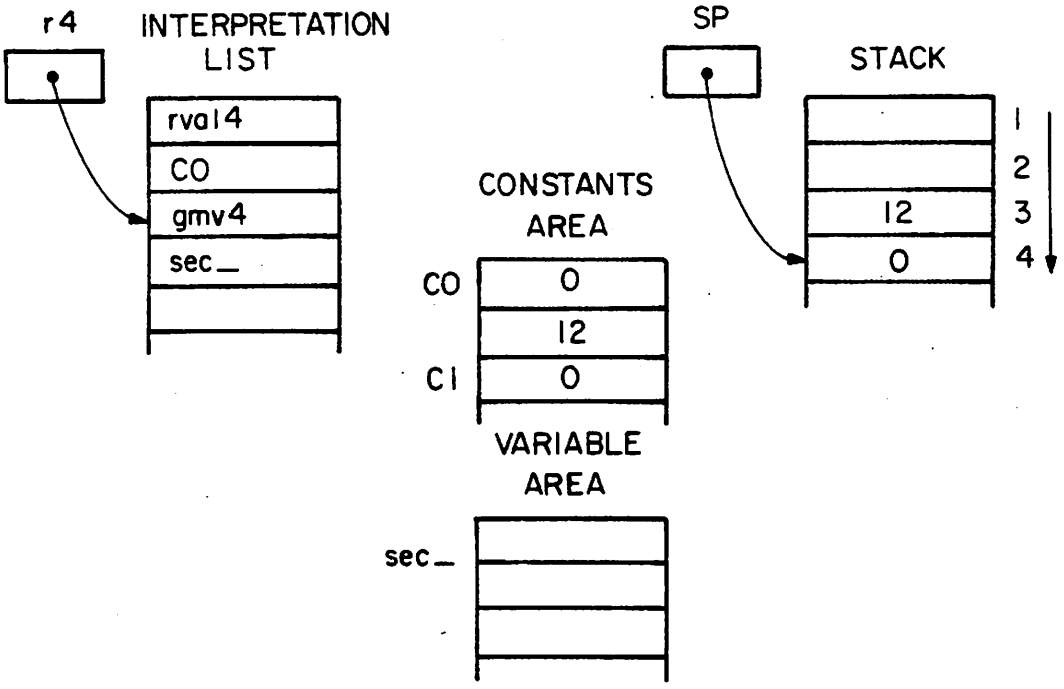


FIG. A7 AFTER EXEC OF rval4
OR
BEFORE EXEC OF gmv4

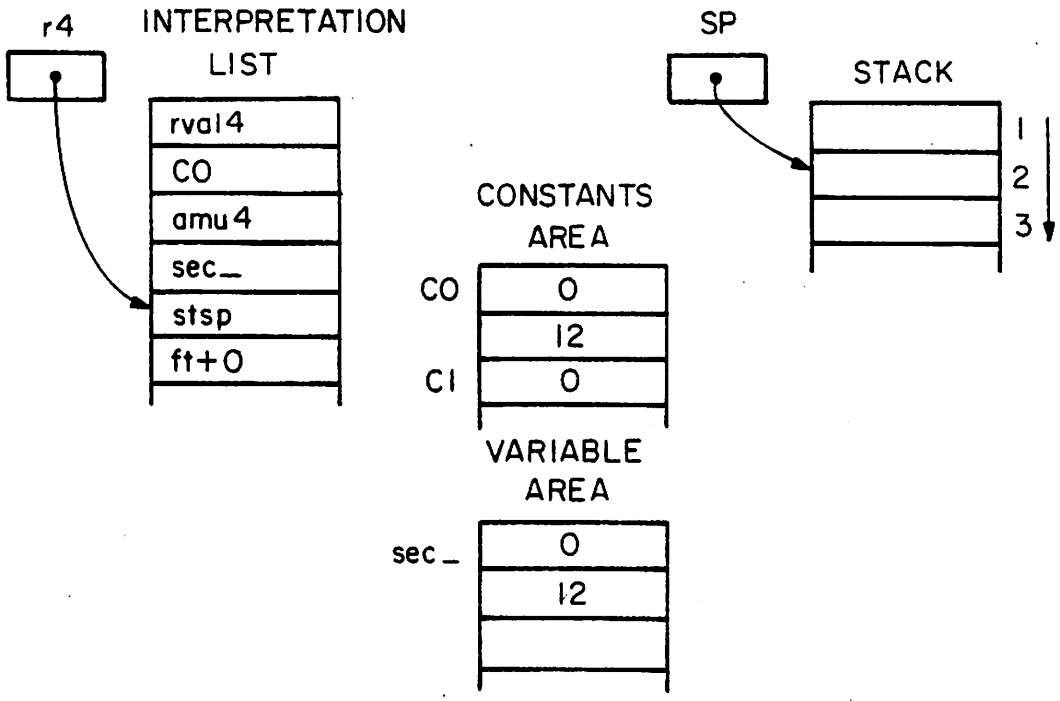


FIG. A8 AFTER EXEC gmv4/BEFORE EXEC stsp

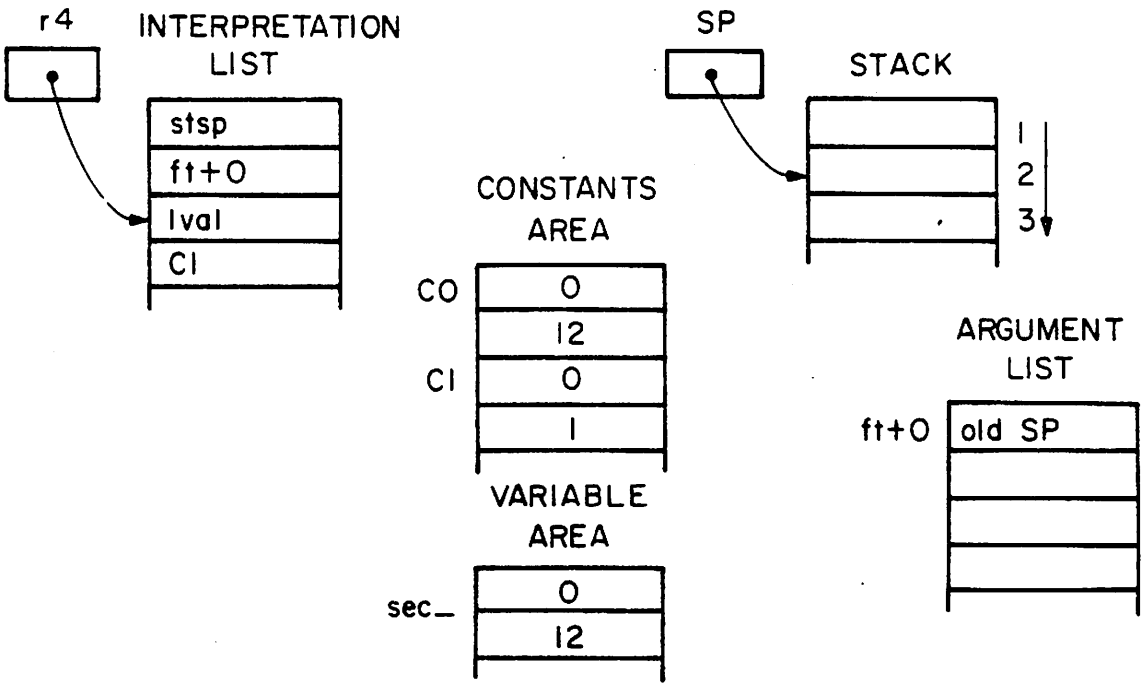


FIG. A9 AFTER EXEC OF stsp/BEFORE EXEC lval

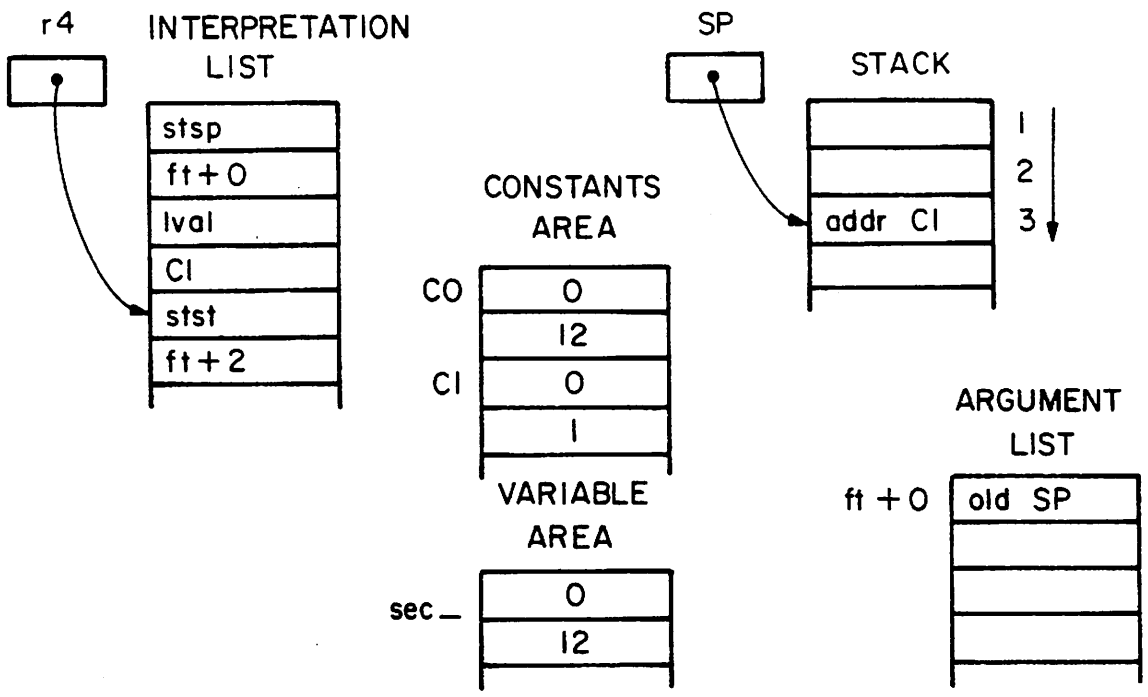


FIG. A10 AFTER EXEC lval / BEFORE EXEC stst

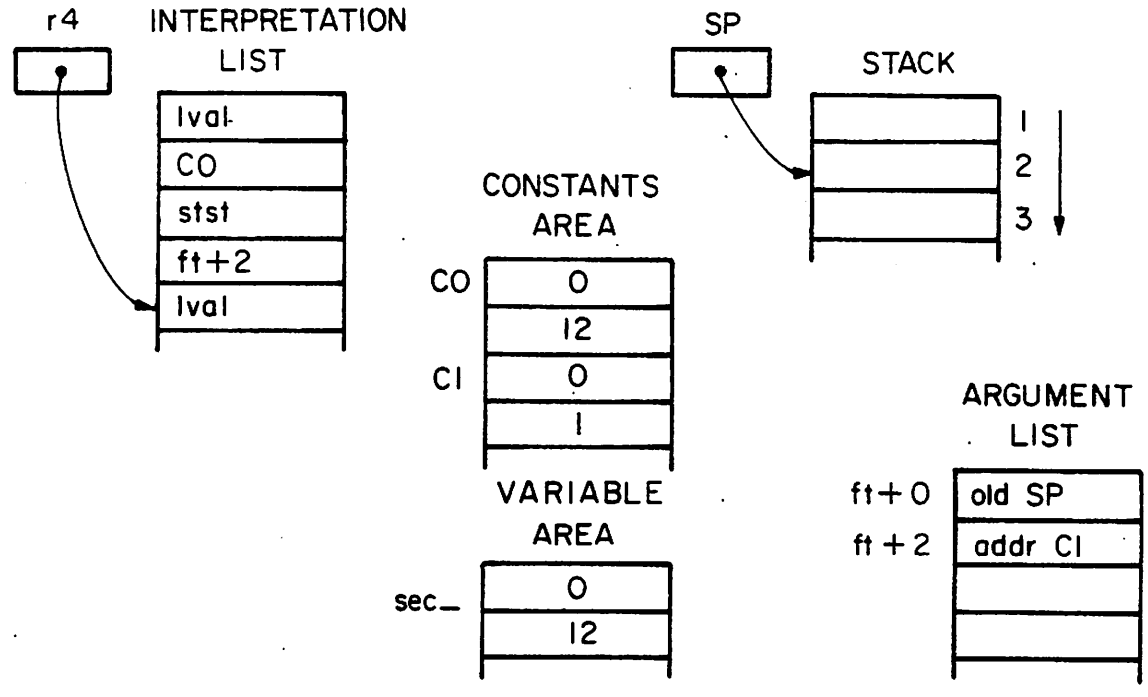


FIG. A11 AFTER EXEC stst

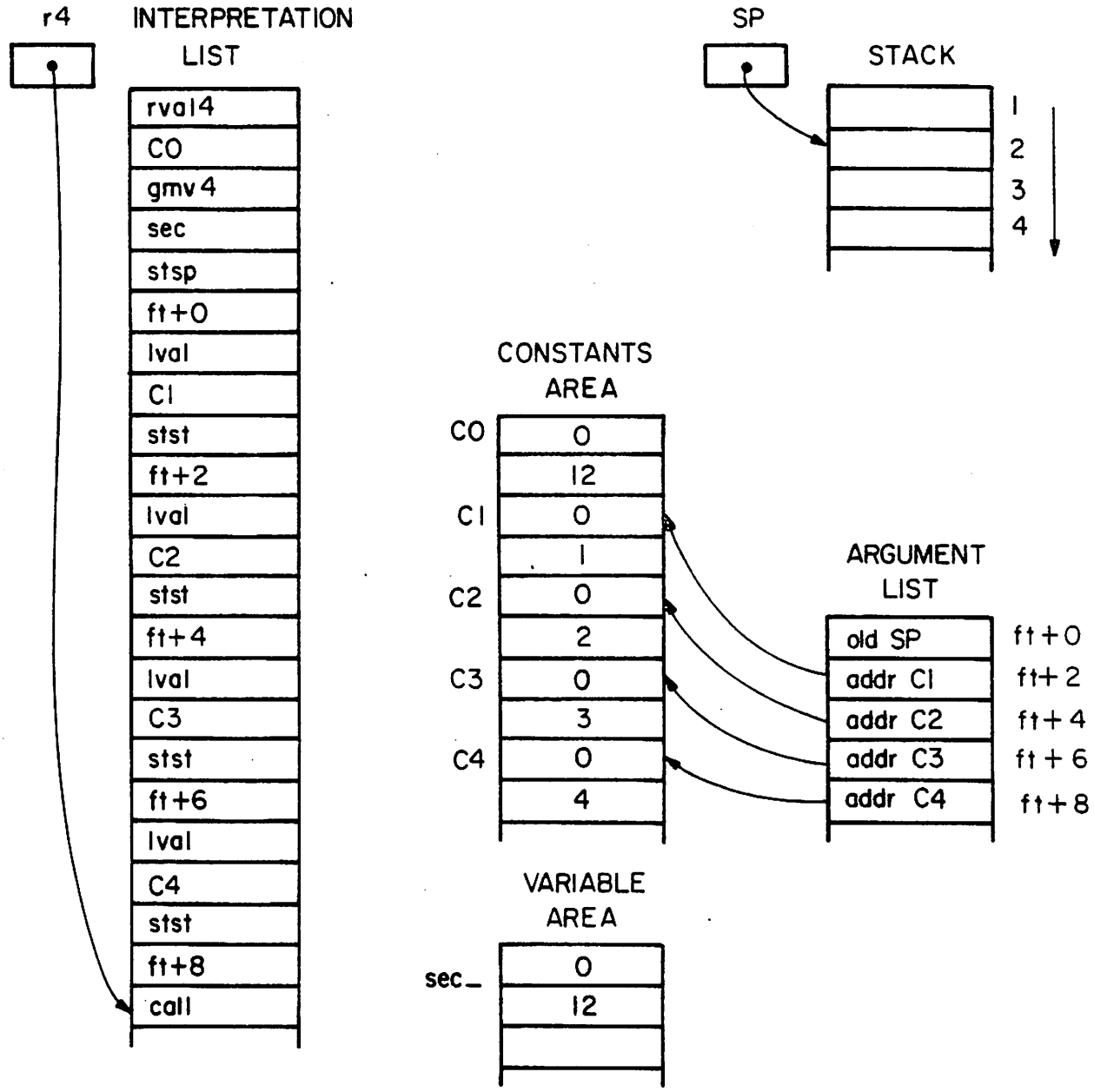


FIG. A12 PRIOR TO CALL

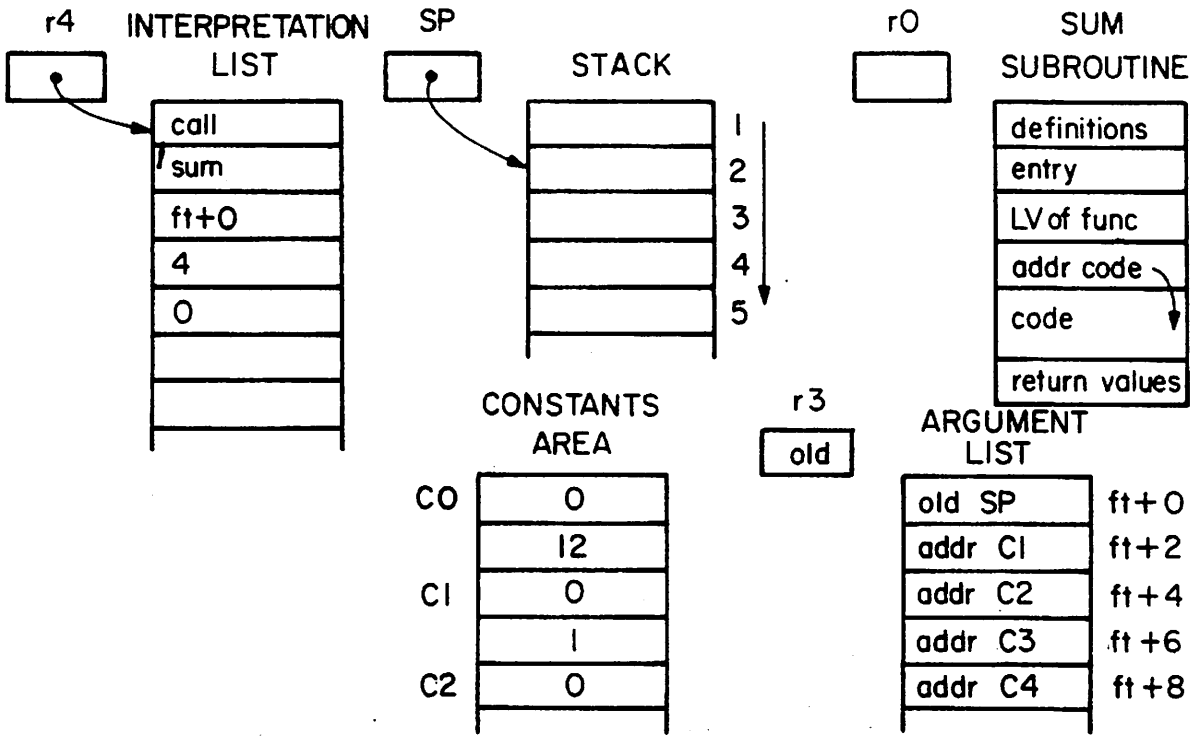


FIG. A13 BEFORE EXEC CALL

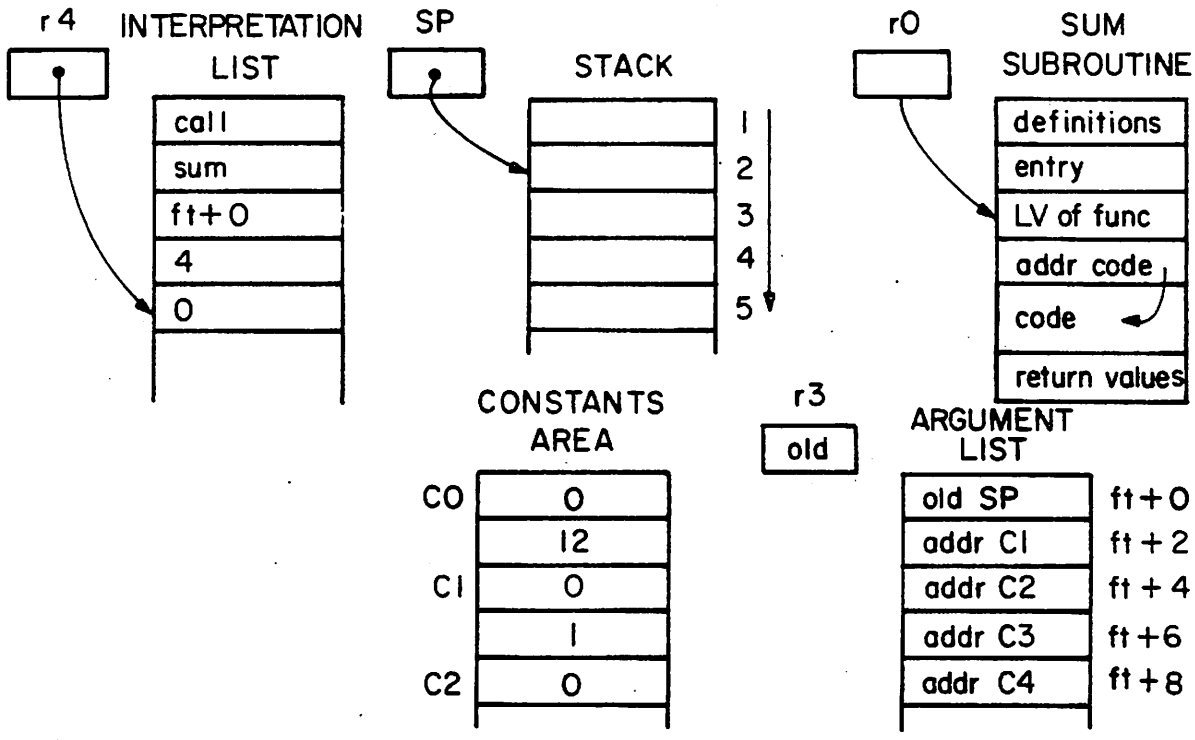


FIG. A14 STEP 1 OF CALL (r0 POINTS TO LV OF FUNCTION)

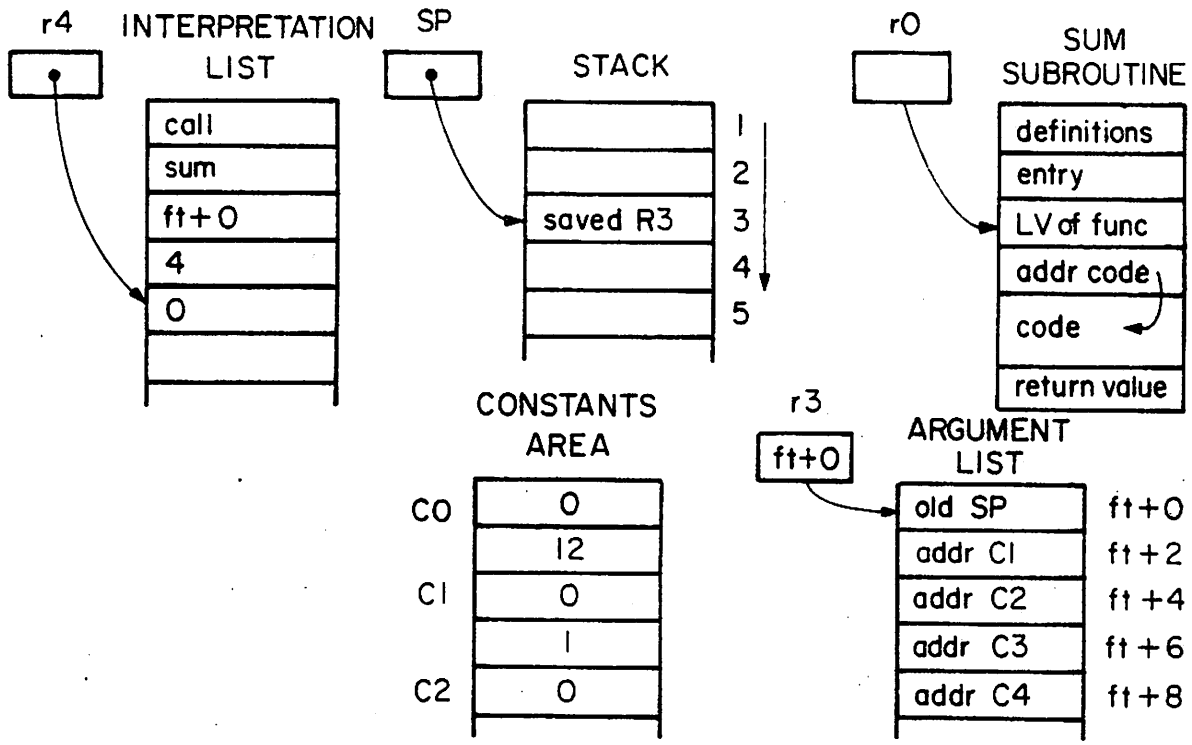


FIG. A15 STEP 2 OF CALL (SAVE OLD R3 ON STACK + PUT addr OF ARG LIST IN r3)

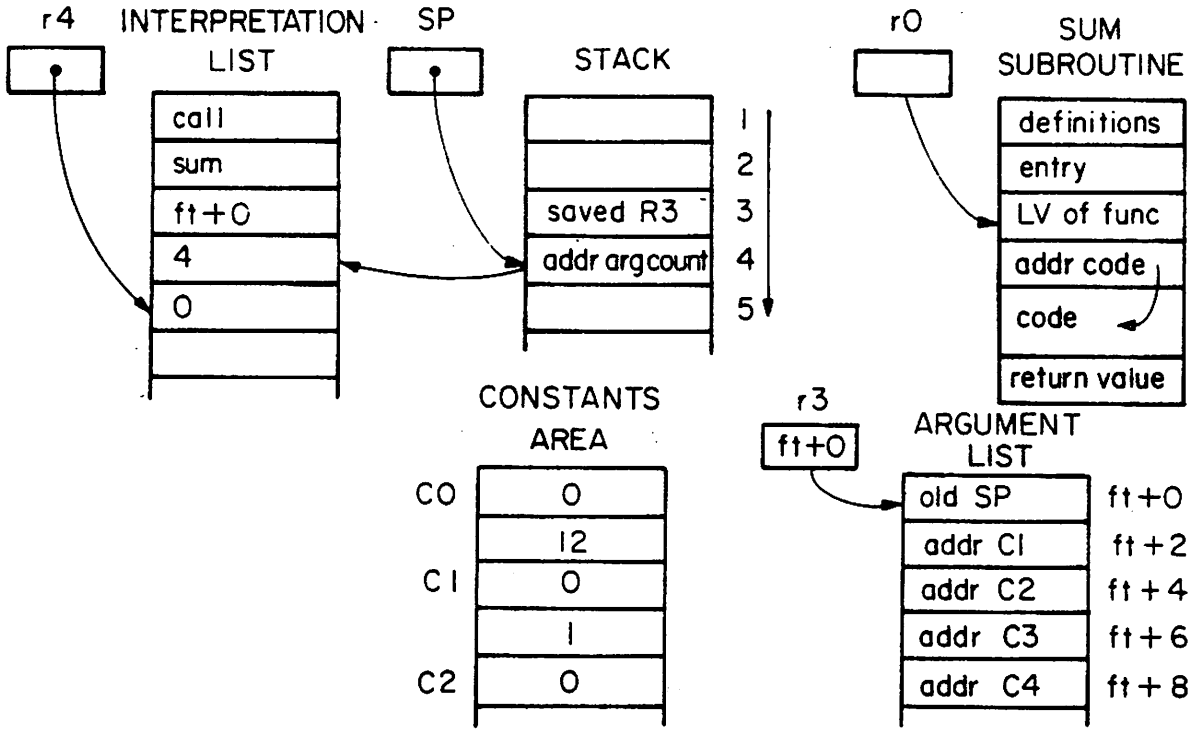


FIG. A16 STEP 3 OF CALL (SAVE ADDRESS OF ARG COUNT ON STACK)

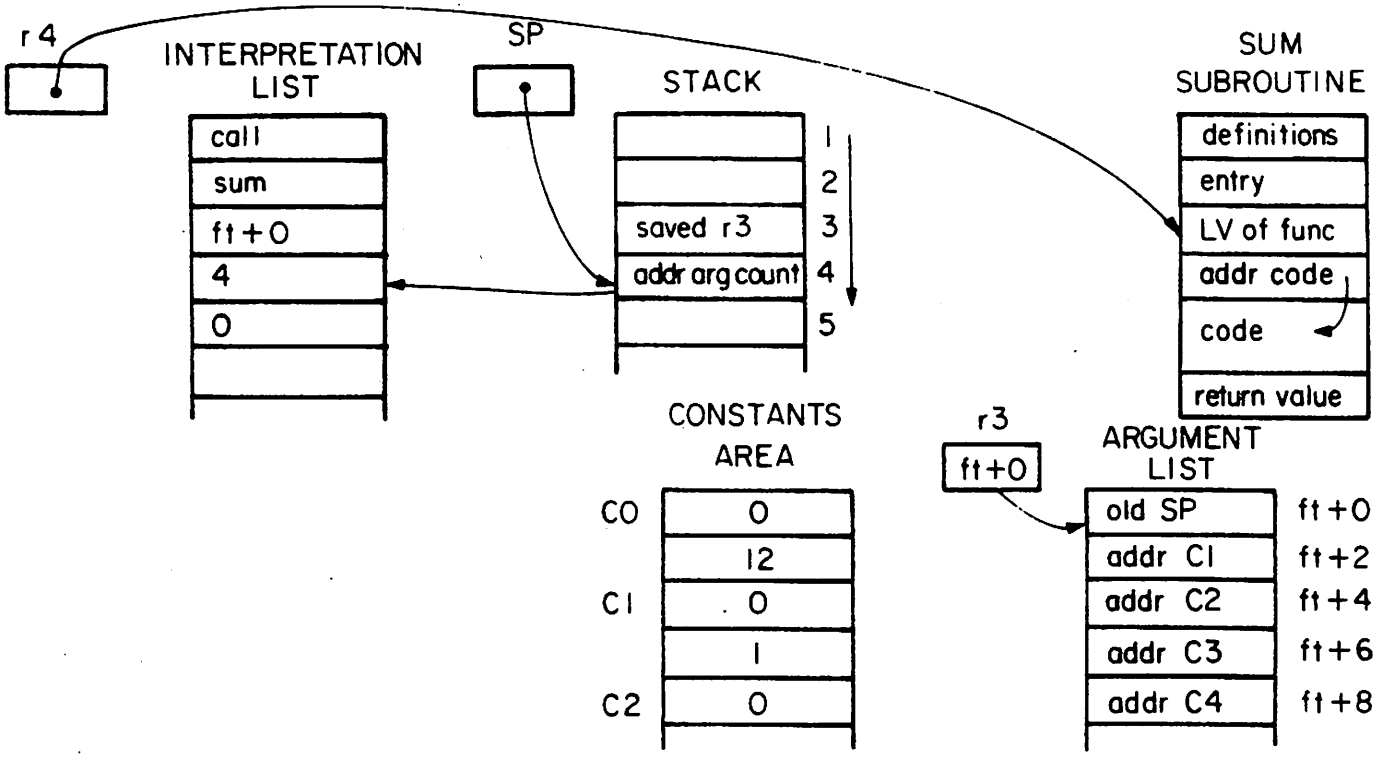


FIG. A17 STEP 4 OF CALL (r4 (FORTRAN INTERPRETER) POINTS TO SUM)

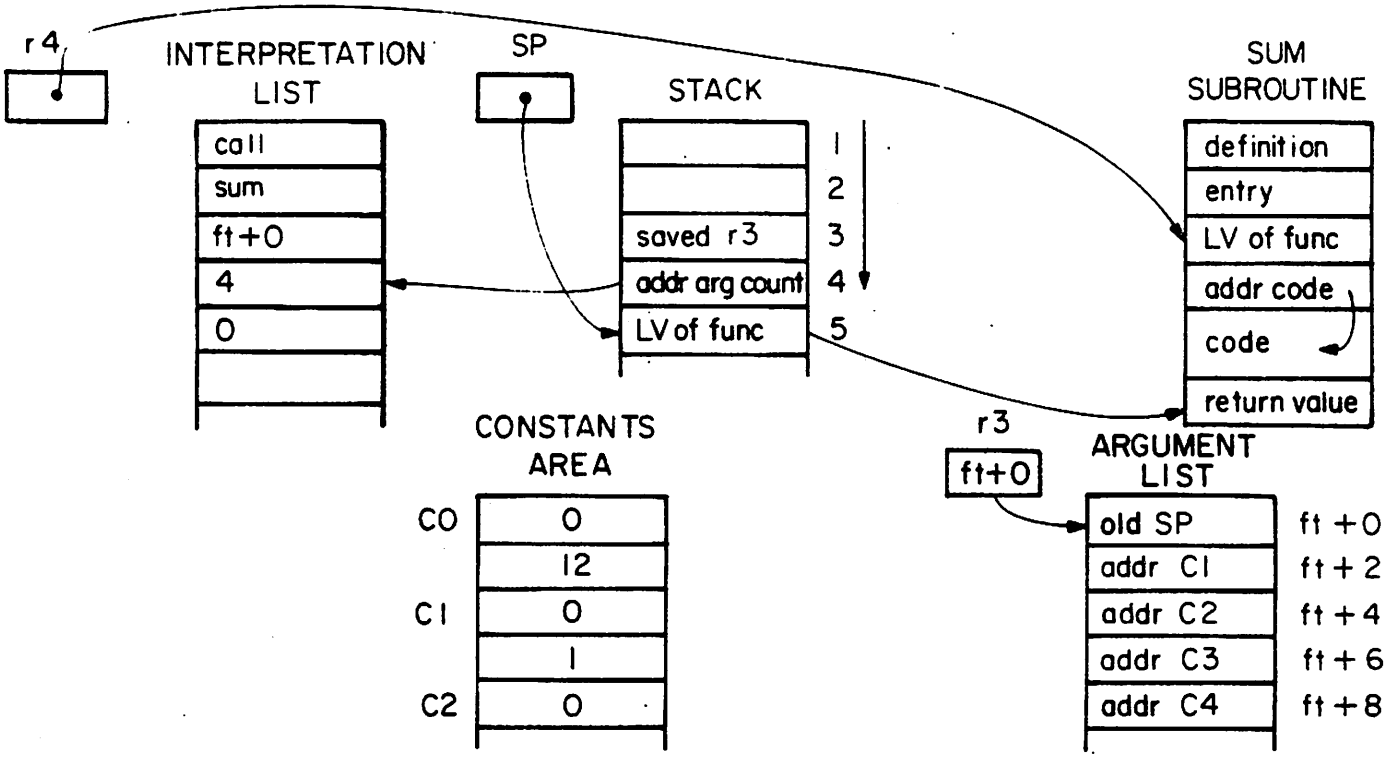


FIG. A18 STEP 5 OF CALL (SAVE LV OF FUNCTION ON STACK AND INCREMENT r4 TO POINT TO ADDRESS EXECUTION CODE)

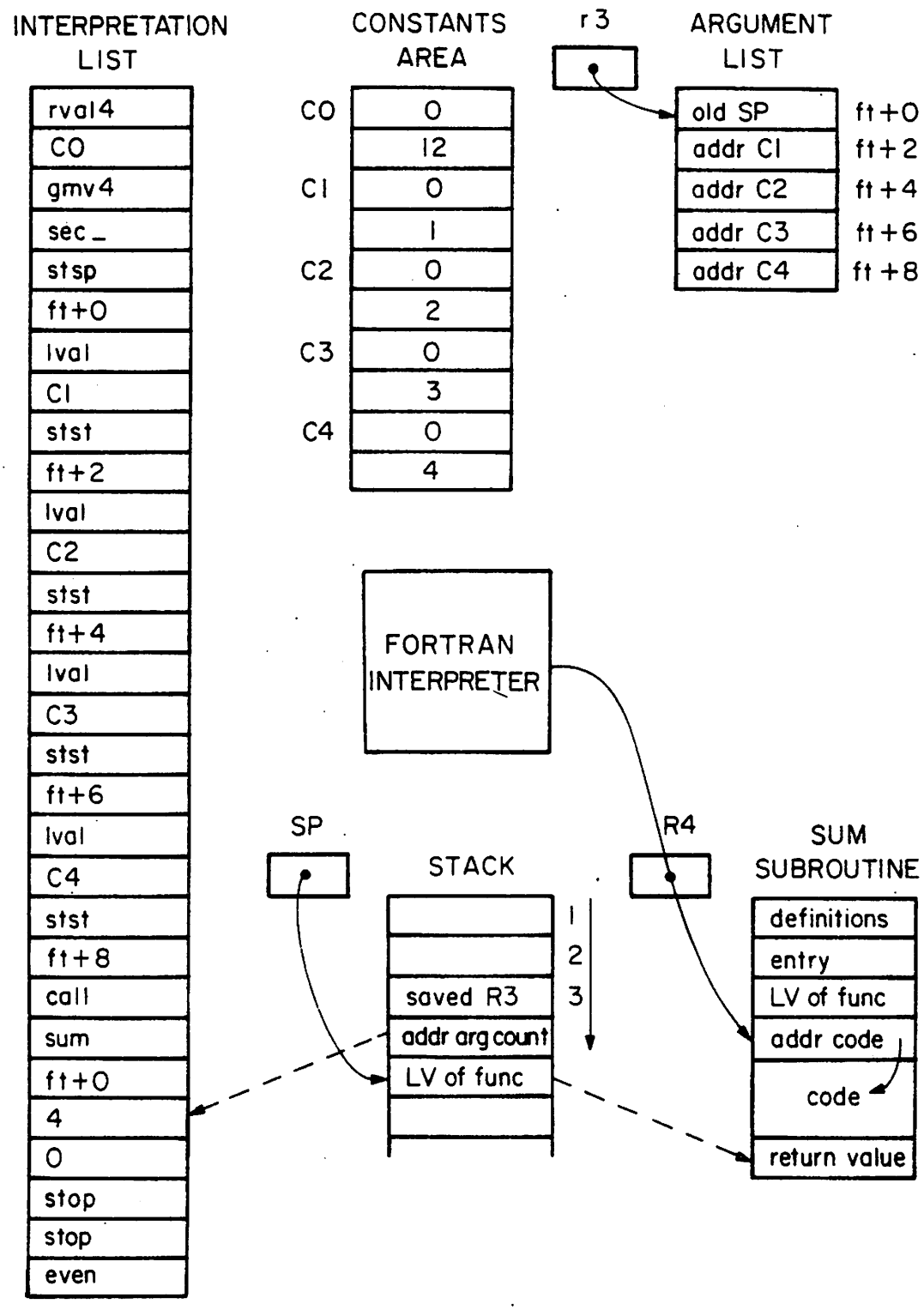


FIG. AI9 CALL PRIMITIVE FINISHED

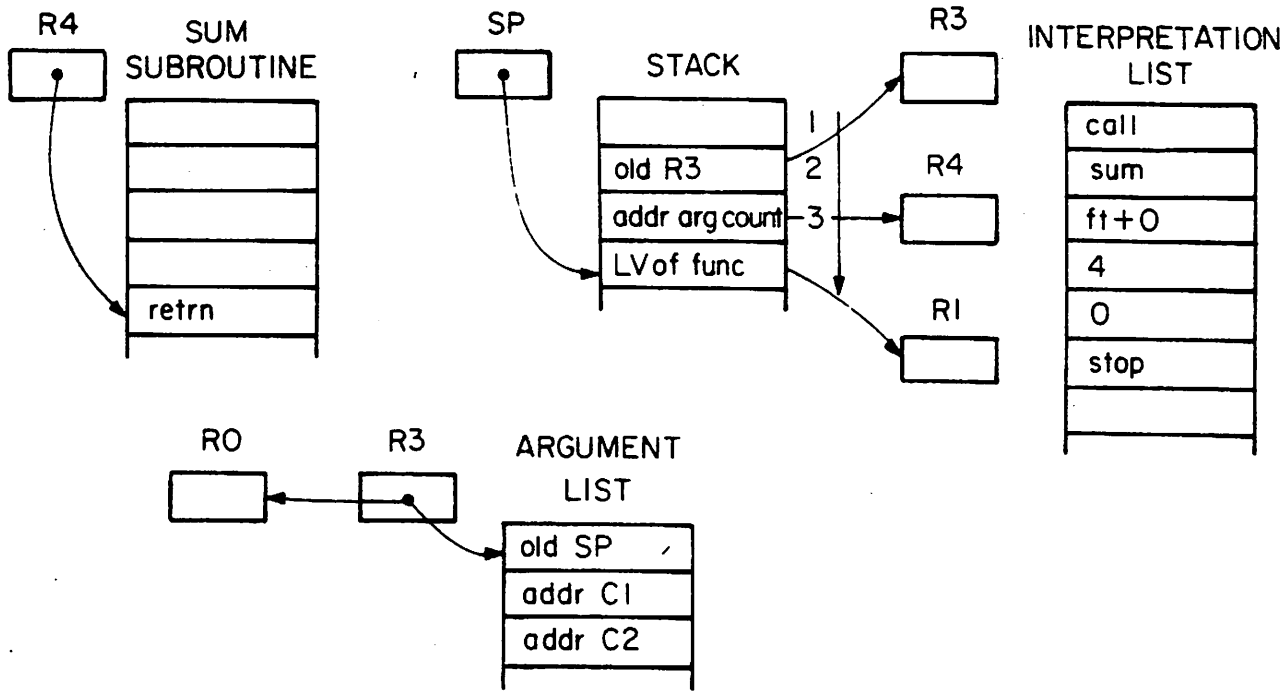


FIG. A20 STEP 1 RETRN (r3 INTO r0 – r0 PTS TO ARGUMENT LIST)

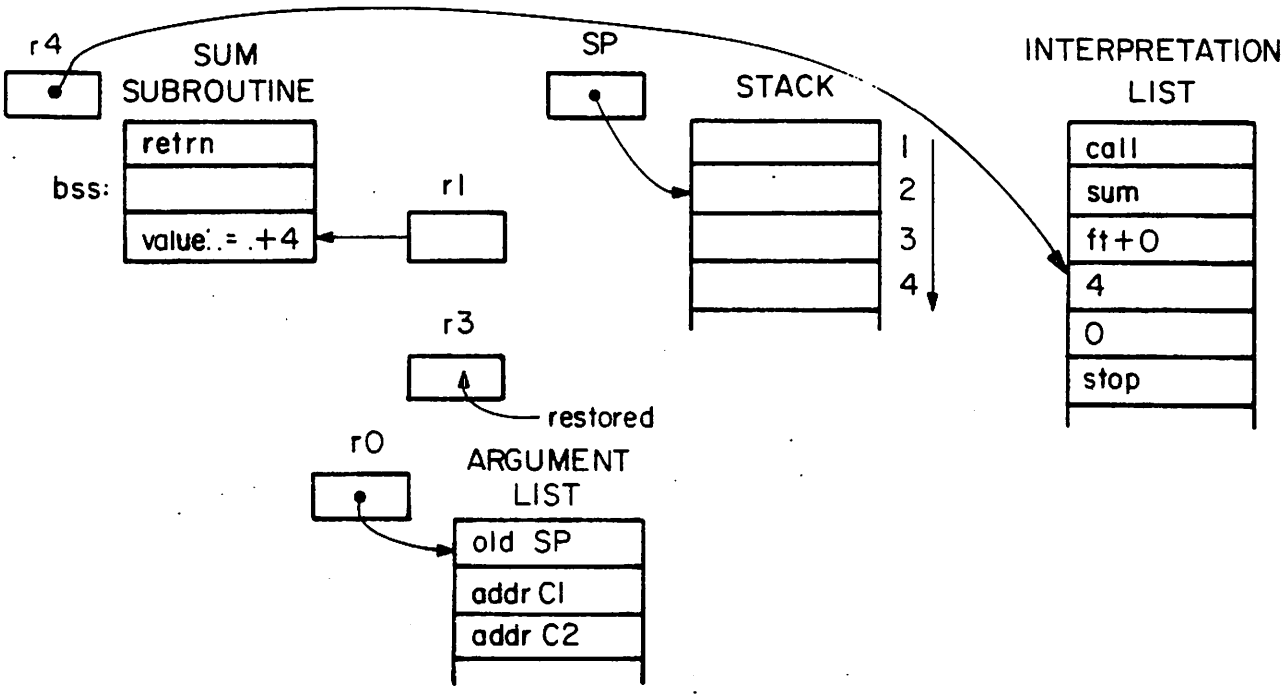


FIG. A21 STEP 2 RETURN (CLEAR STACK r1 – LV OF FUNCTION
r4 – ADDRESS OF ARGUMENT COUNT
r3 – RESTORED)

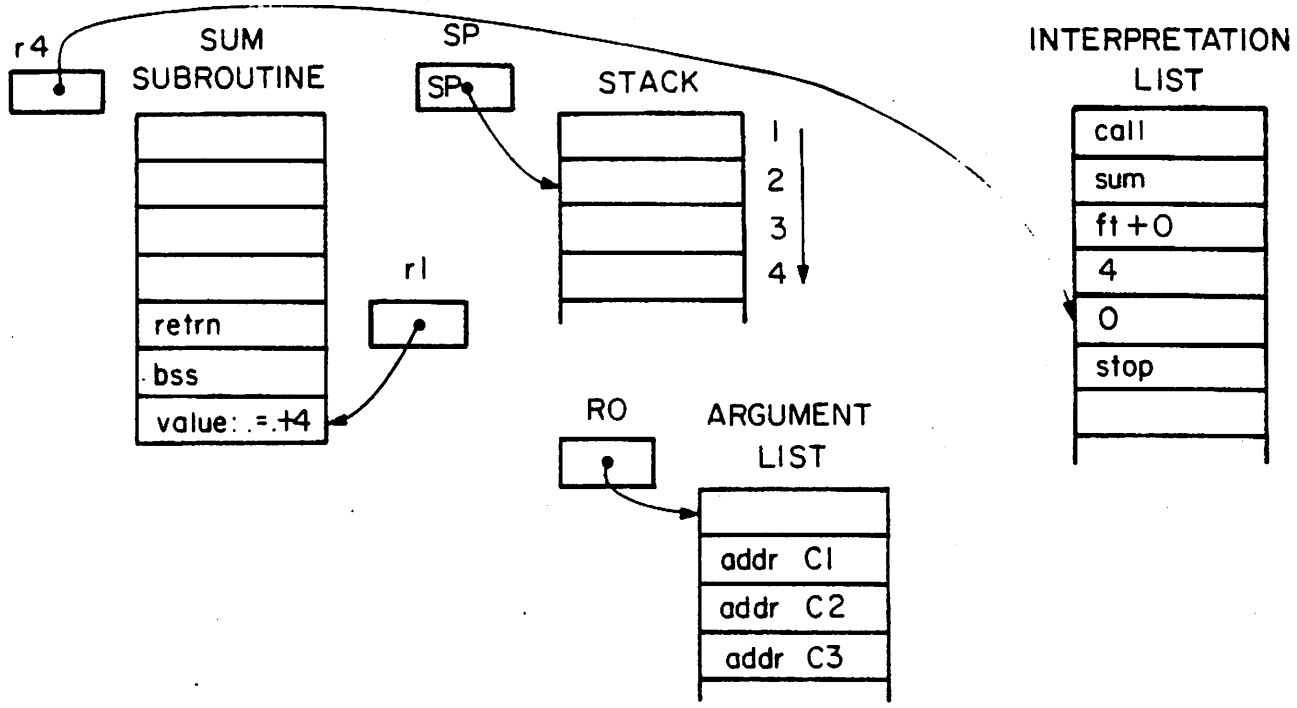


FIG. A22 STEP 3 retrn (RESTORE ORIGINAL SP)

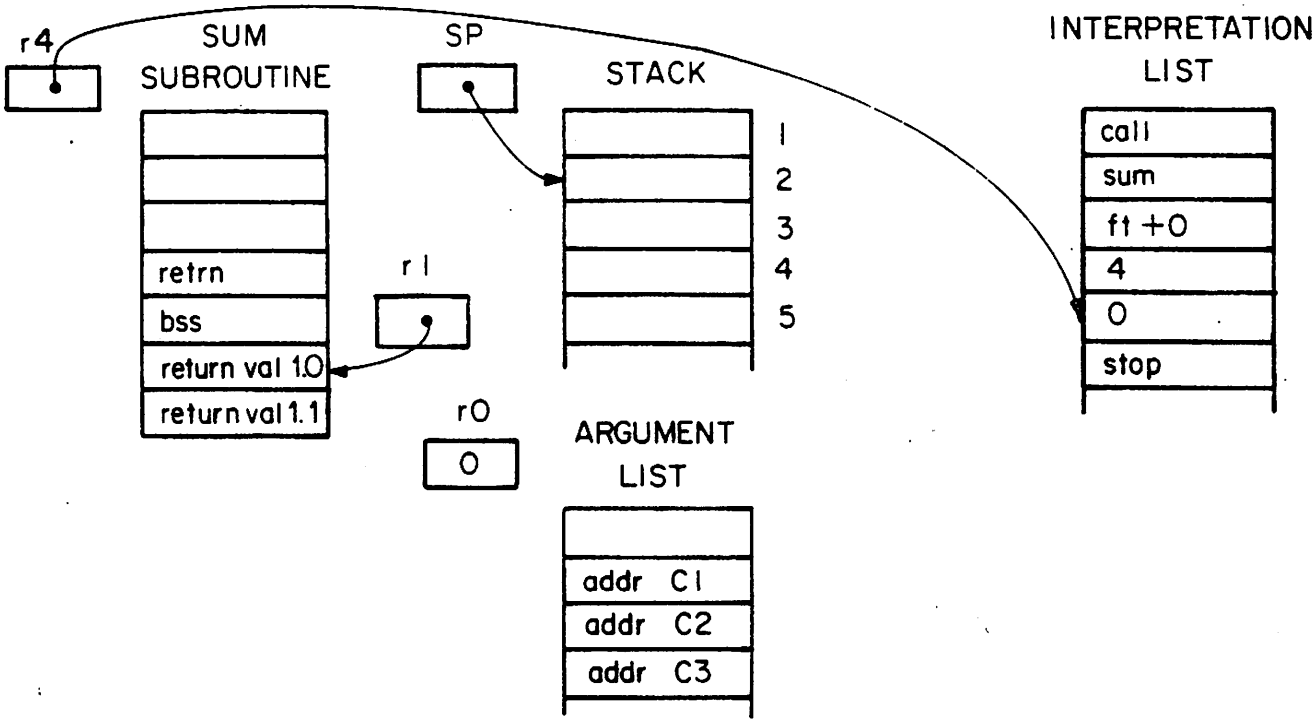


FIG. A23 STEP 4 retrn (PUT RETURN BYTE COUNT IN r0
THEN ROUND UP TO EVEN NUMBER IF NOT)

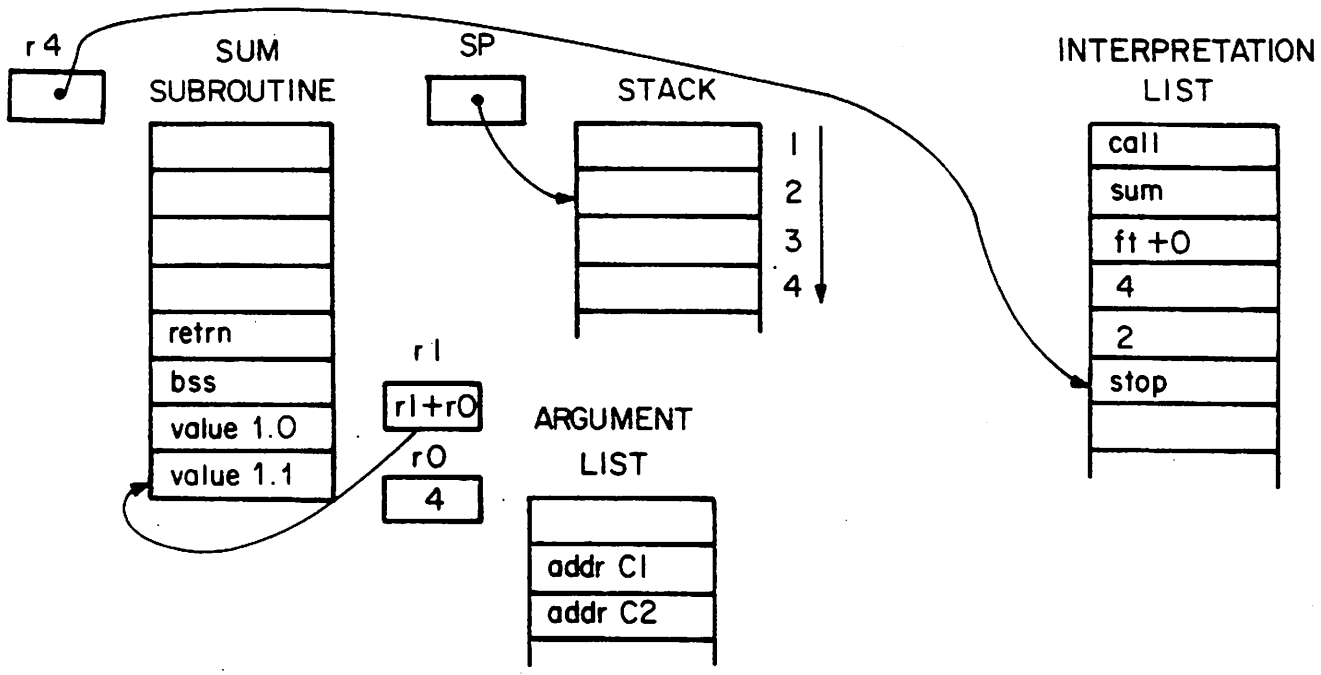


FIG. A24 STEP 5 retrn (ADD BYTE COUNT TO LV OF FUNCTION)

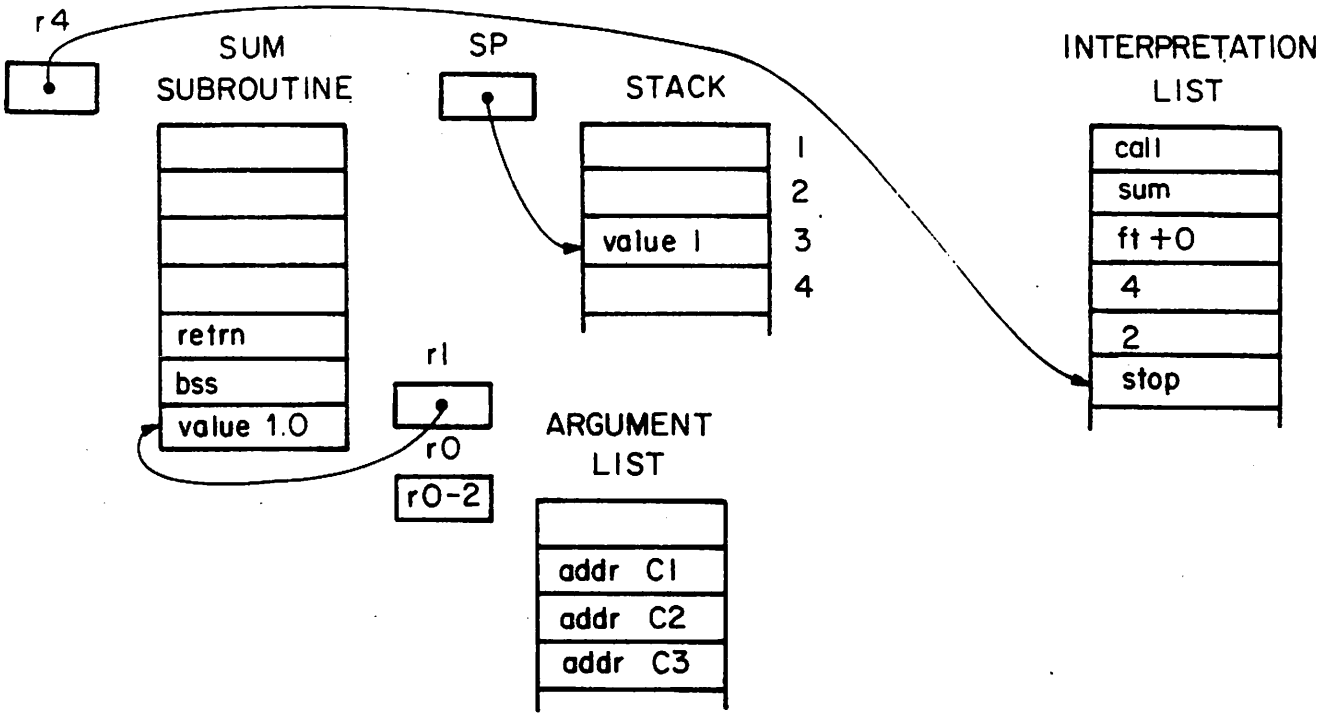


FIG. A25 STEP 6 retrn (SUBTRACT 2 FROM BYTE COUNT IF ≠0
DECREMENT r1/PUSH LAST RETURN VALUE ONTO STACK)