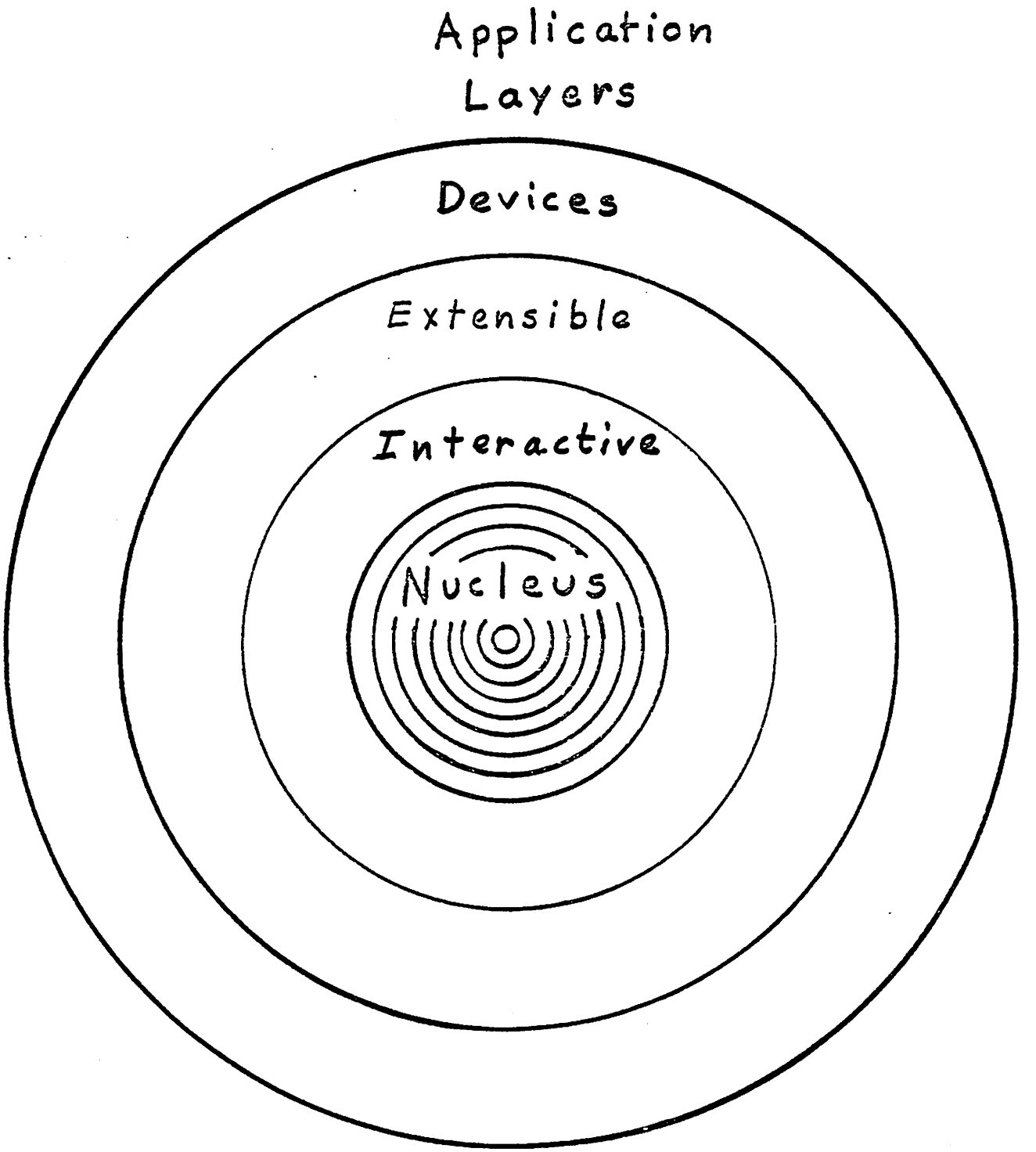
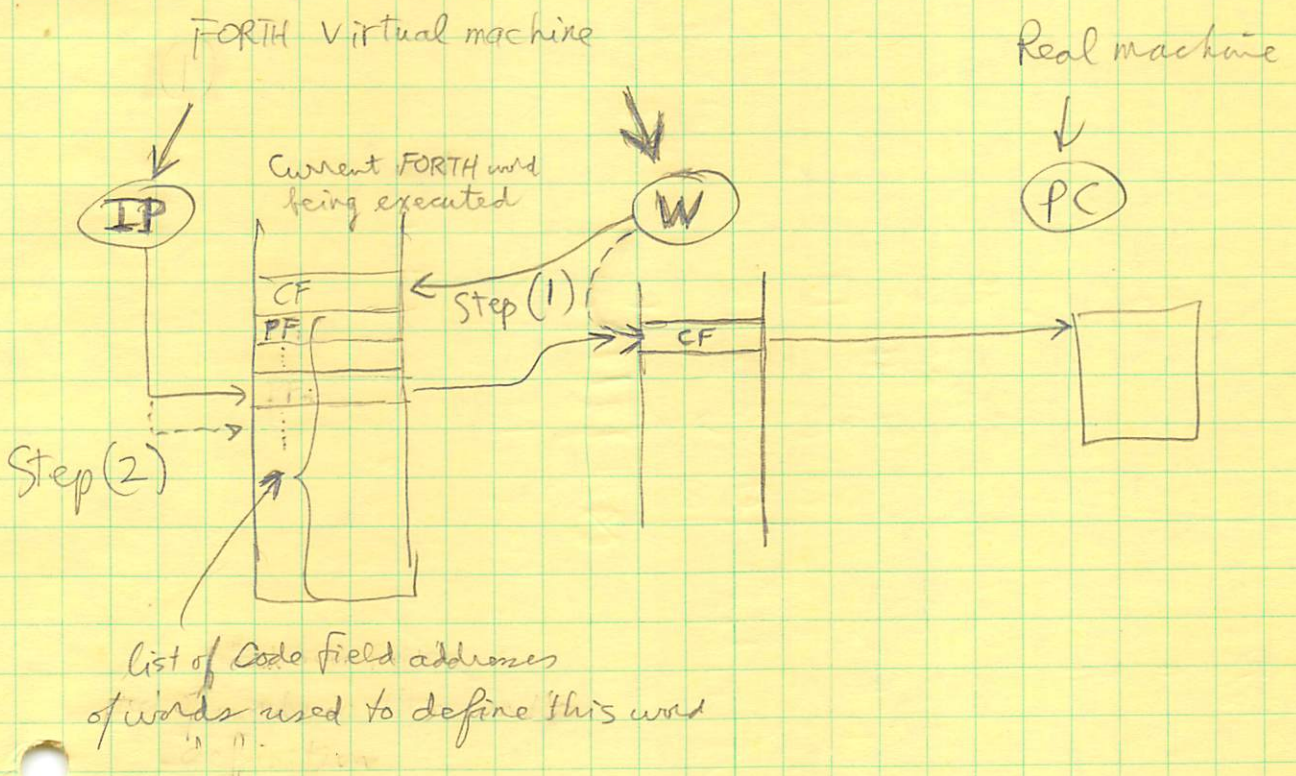
ADDRESS INTERPRETER

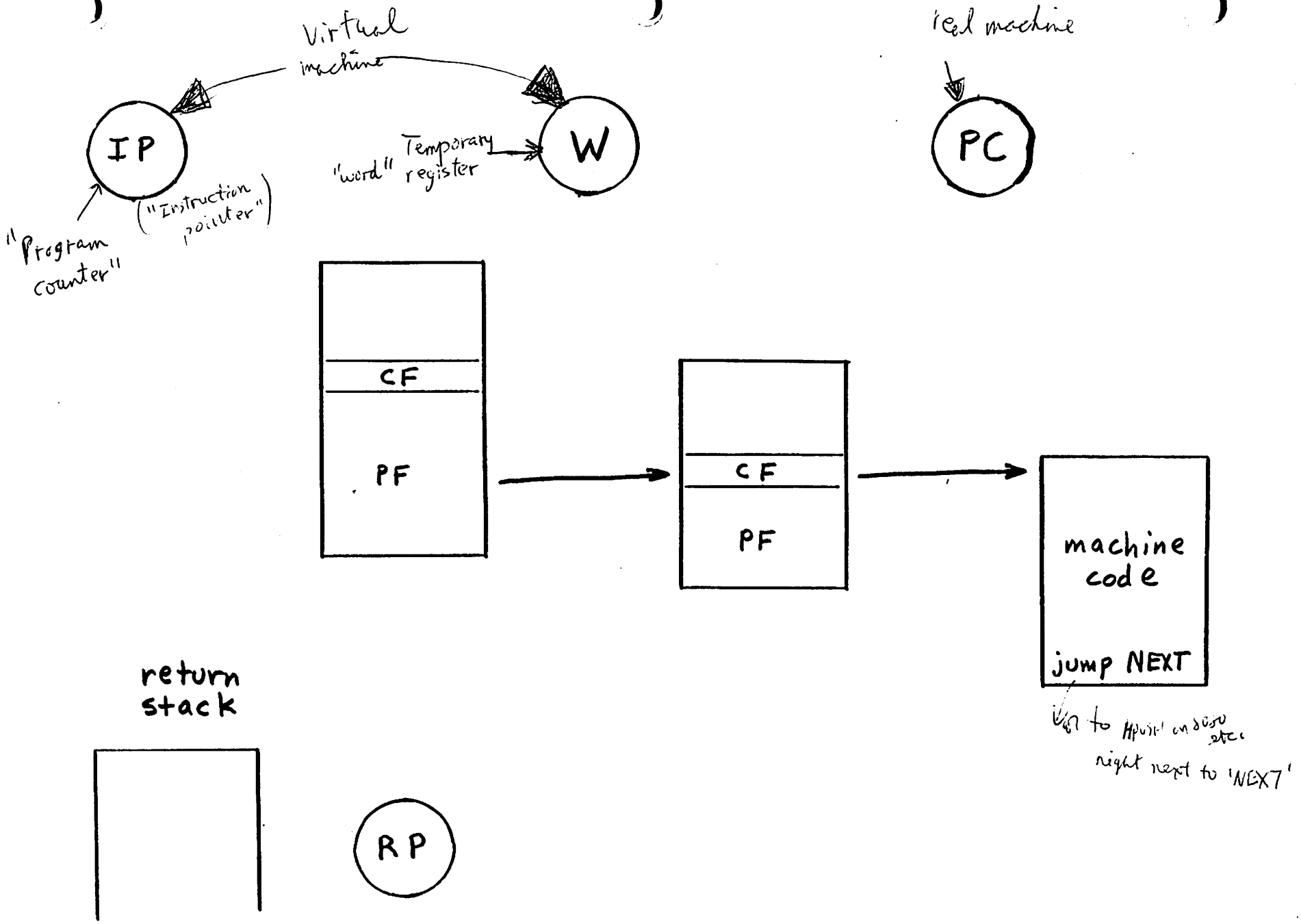


1-12-81

FORTH's address interpreter

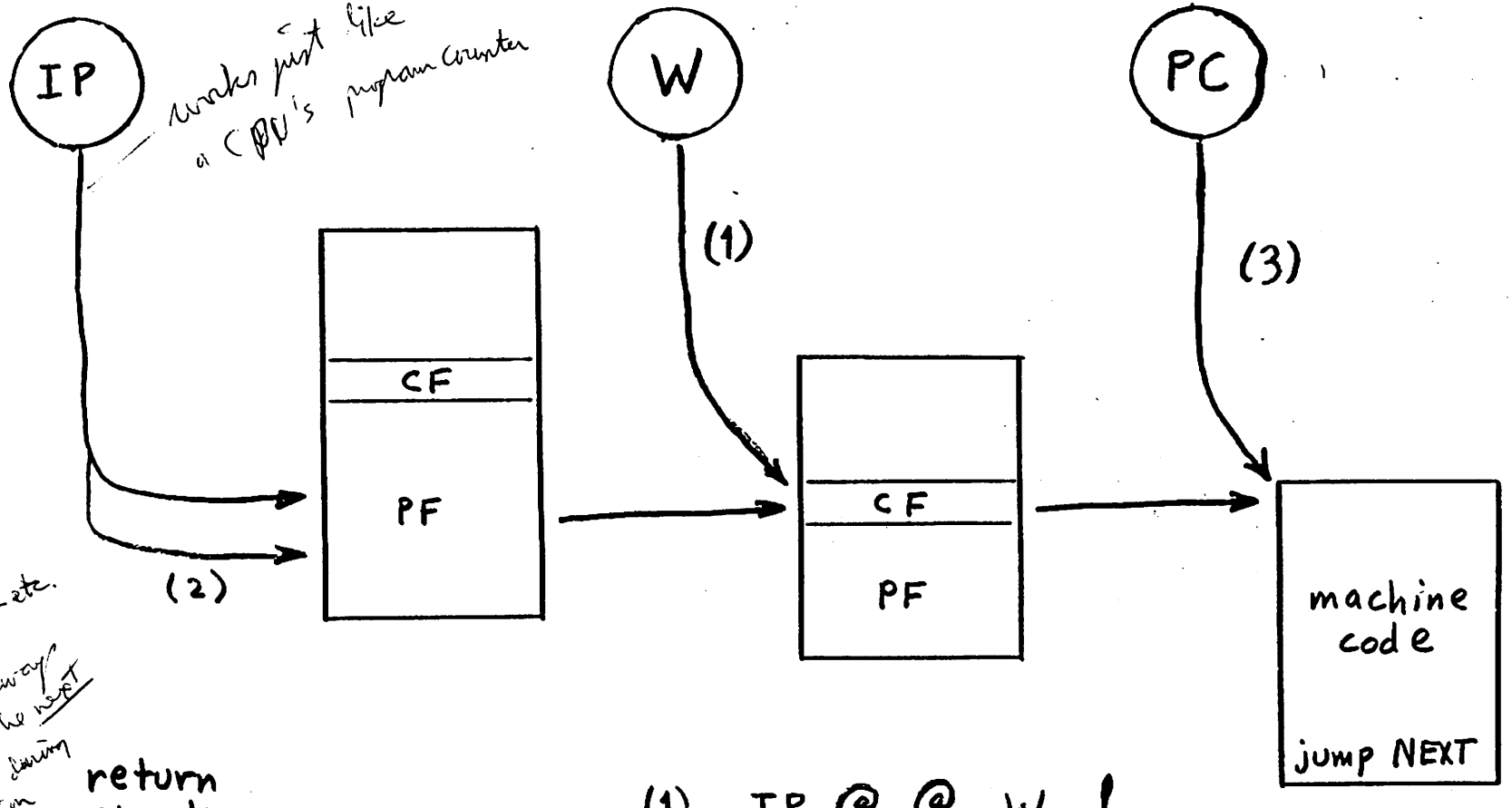
(see Kim Harris course notes p. 108)




Virtual machine
real machine
IP
("Instruction pointer")
"Program counter"
"word"
Temporary register
W
PC
CF
PF
CF
PF
machine code
jump NEXT
return stack
RP

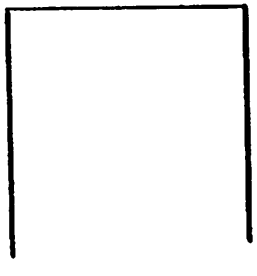
(107)

NEXT FORTH's Address Interpreter



8080, 6502 etc. PACE, PC & IP always point to the next instruction during execution

return stack

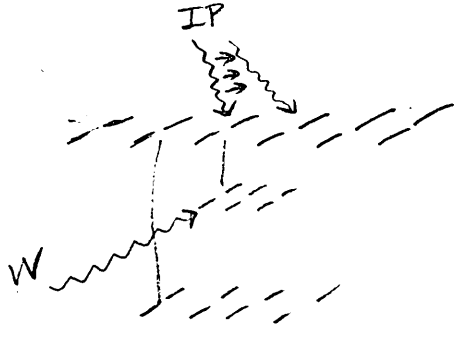


RP

(1) IP @ @ W !

(2) 2 IP +!

(3) W @ @ PC !



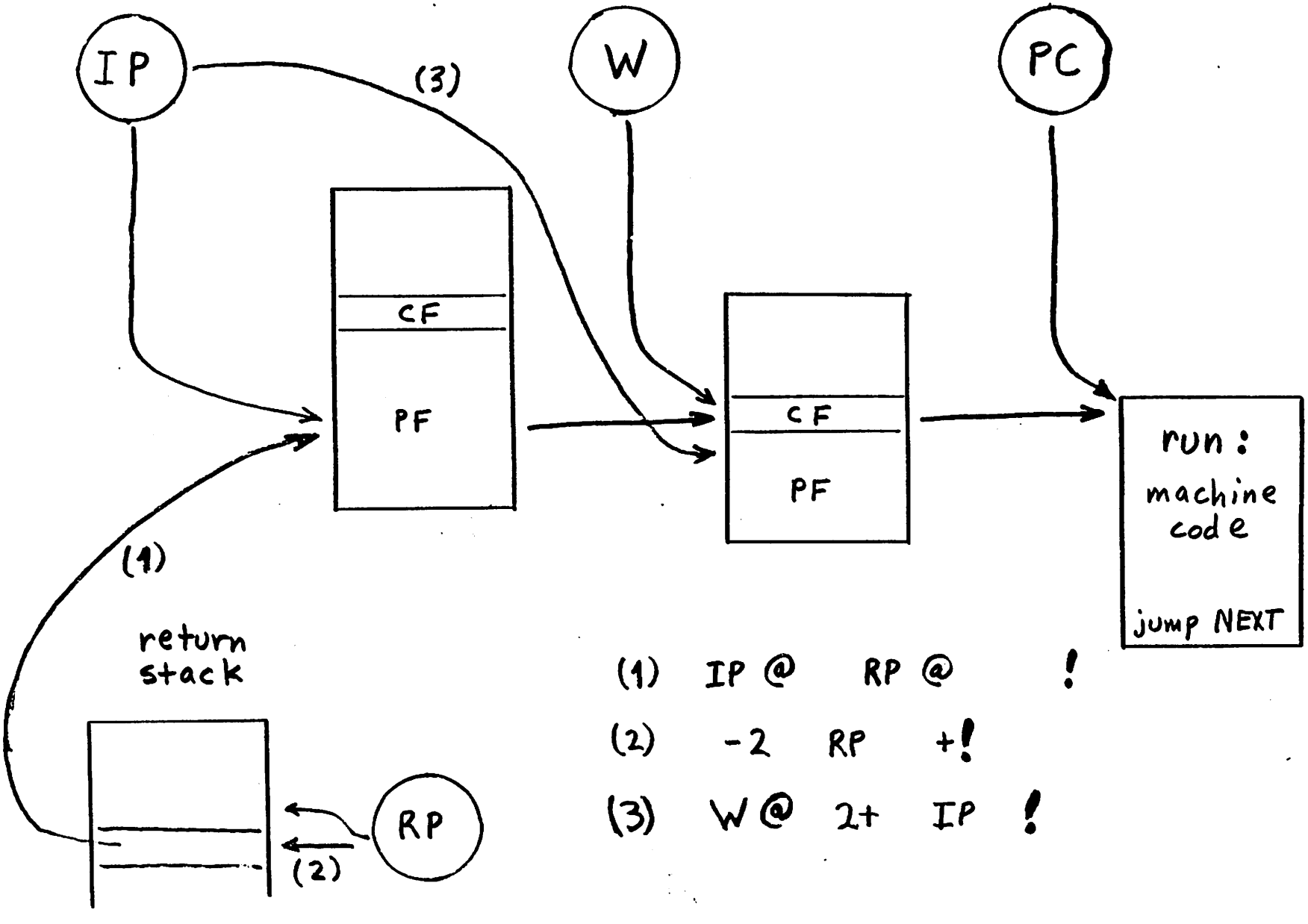
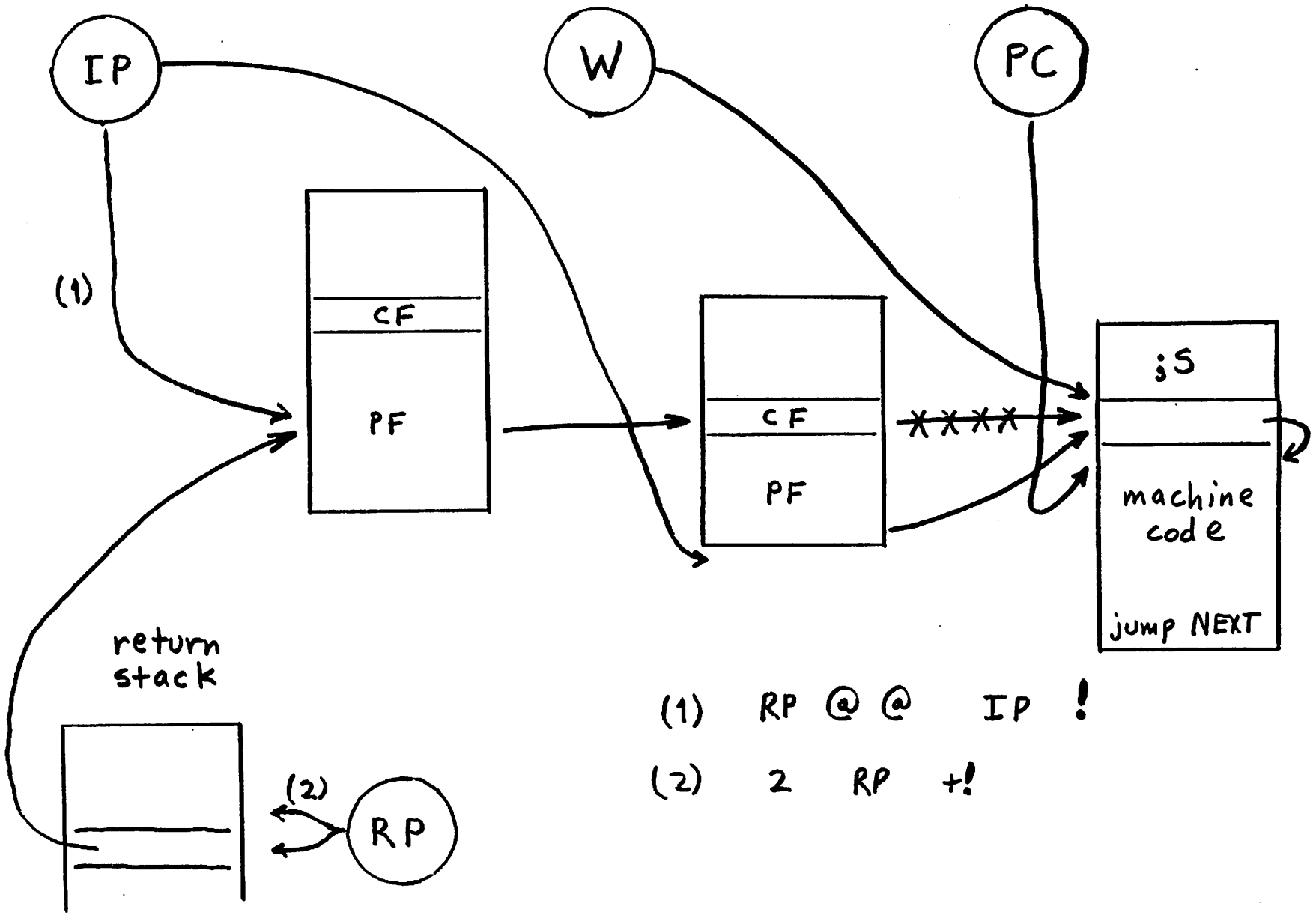

run: "subroutine" nest
IP
W
PC
(3)
CF
PF
CF
PF
run:
machine
code
jump NEXT
(1)
return
stack
RP
(2)
(1) IP @ RP @ !
(2) -2 RP +!
(3) W @ 2+ IP !

fig 2 I4

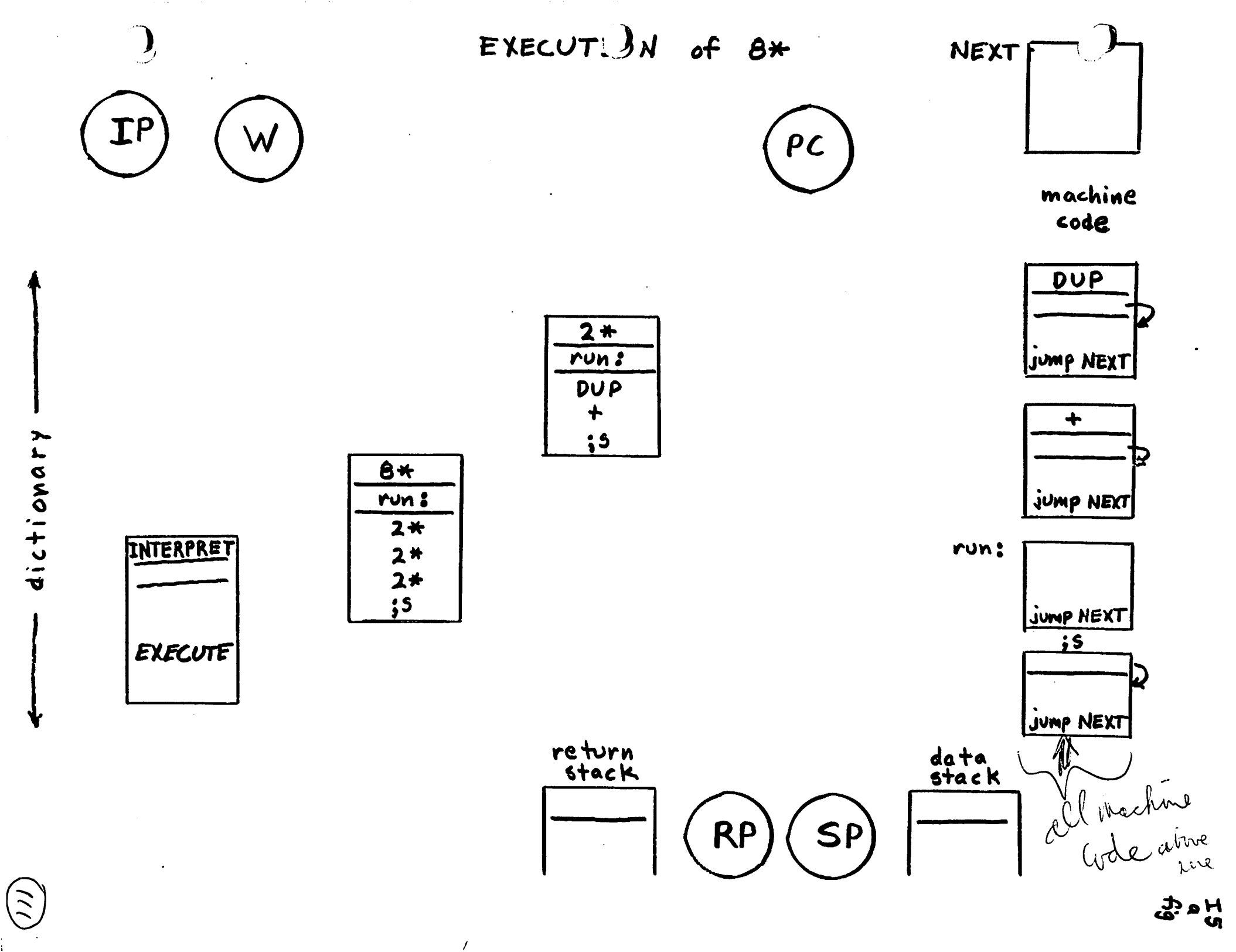
;S "subroutine" unnest



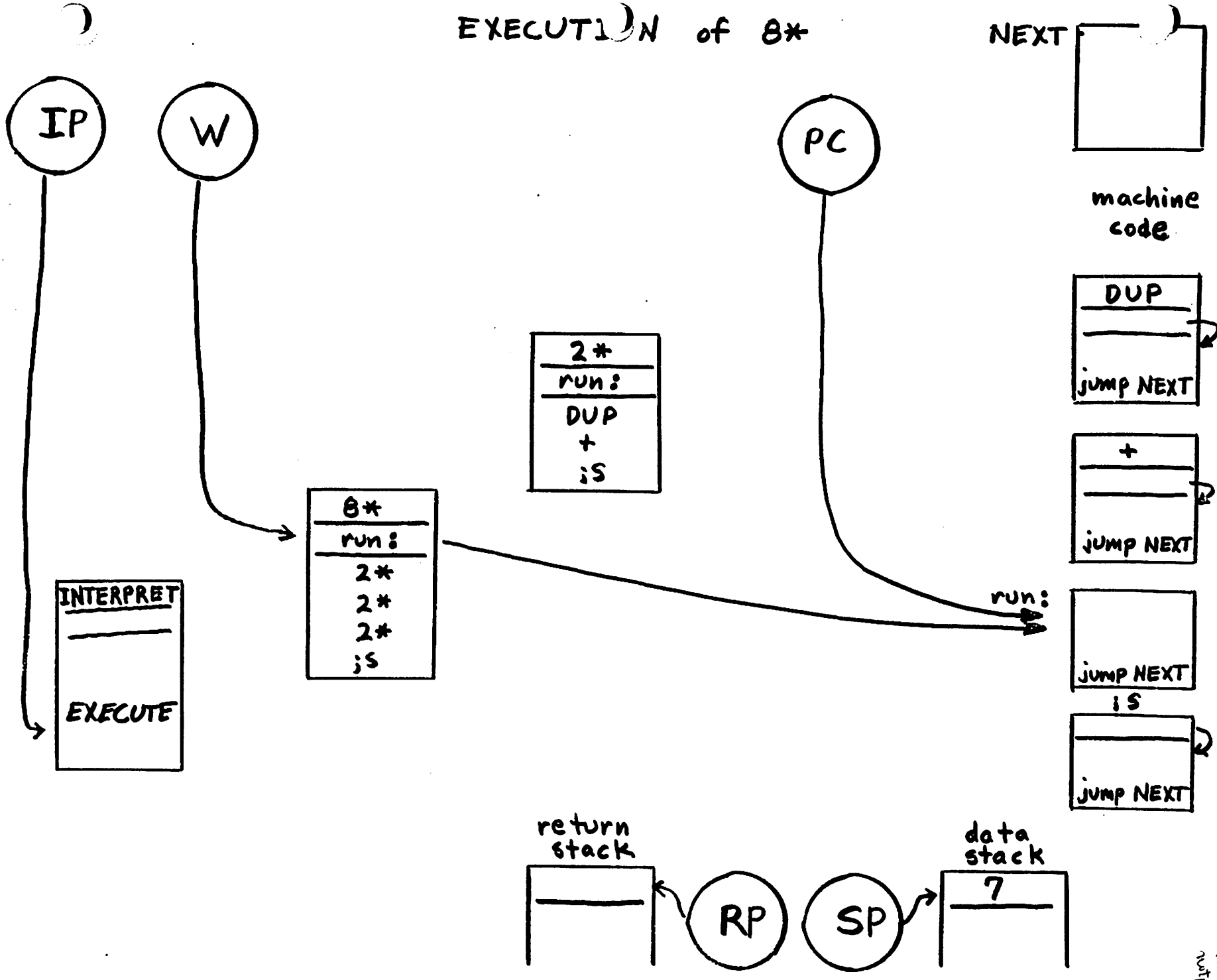
(1) RP @ @ IP !

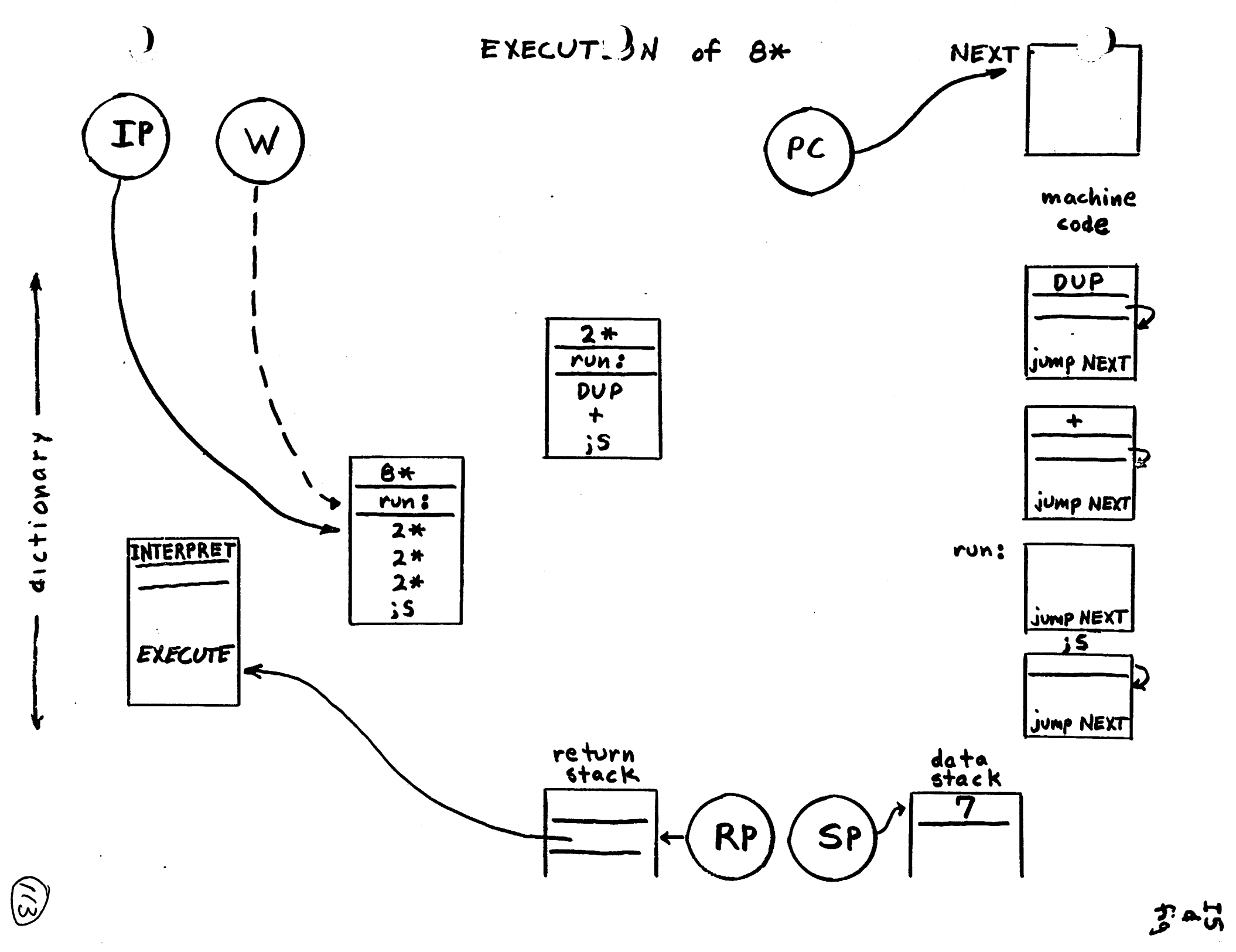
(2) 2 RP +!

EXECUTION of 8*

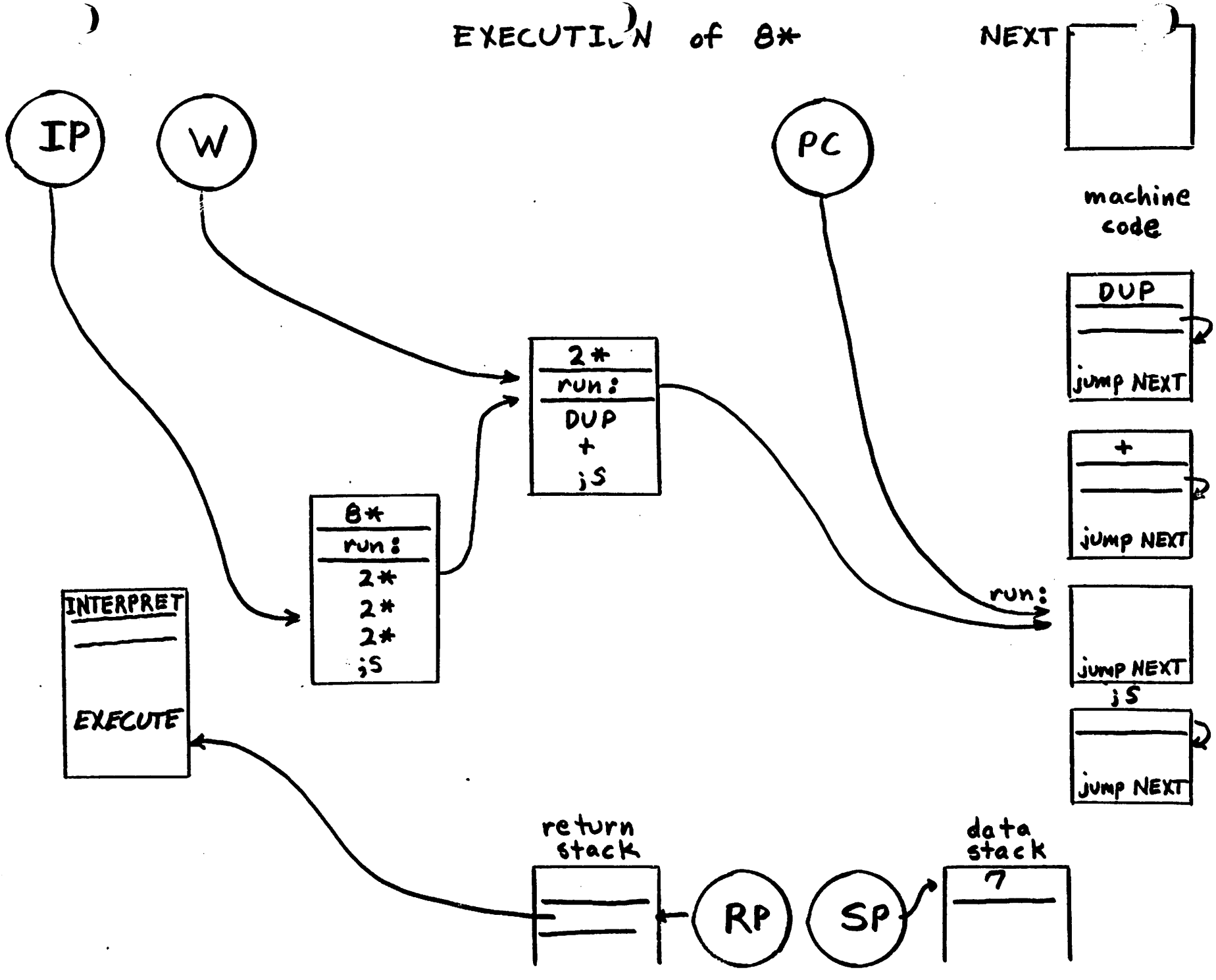


EXECUTION of 8*

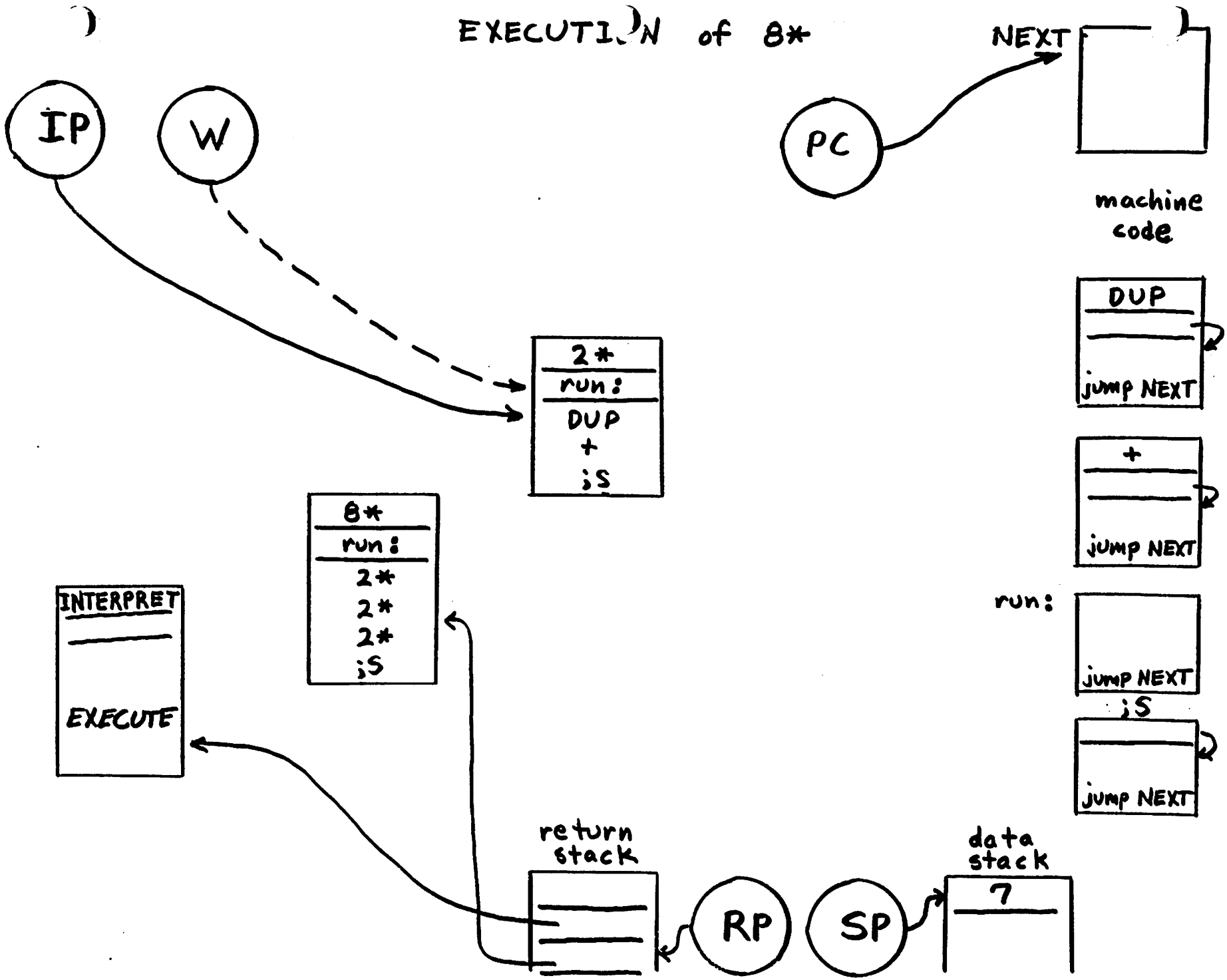


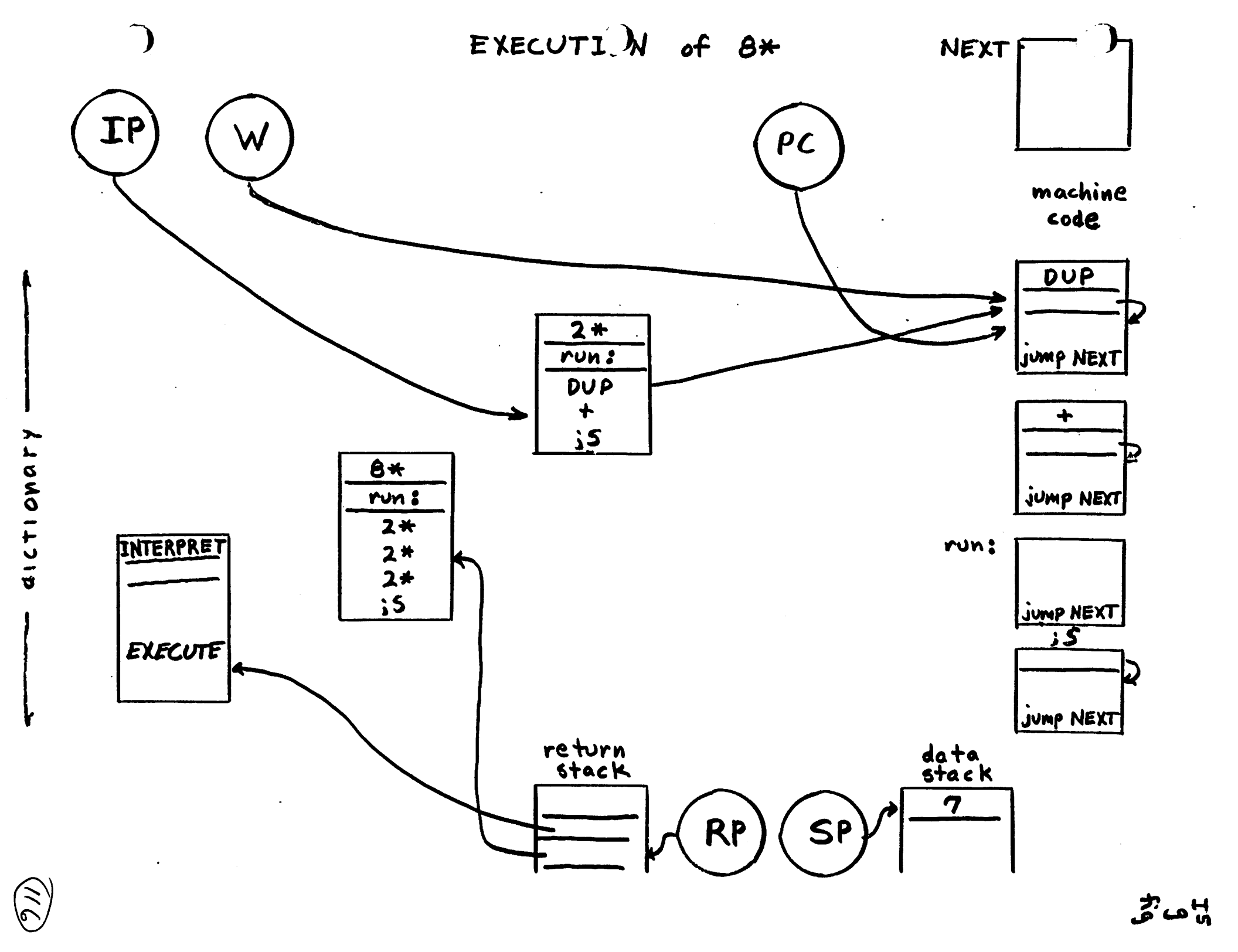

EXECUTION of 8*
NEXT
IP
W
PC
machine code
DUP
jump NEXT
2*
run:
DUP
+
;S
+
jump NEXT
8*
run:
2*
2*
2*
;S
run:
jump NEXT
;S
jump NEXT
INTERPRET
EXECUTE
dictionary
return stack
data stack
7
RP
SP

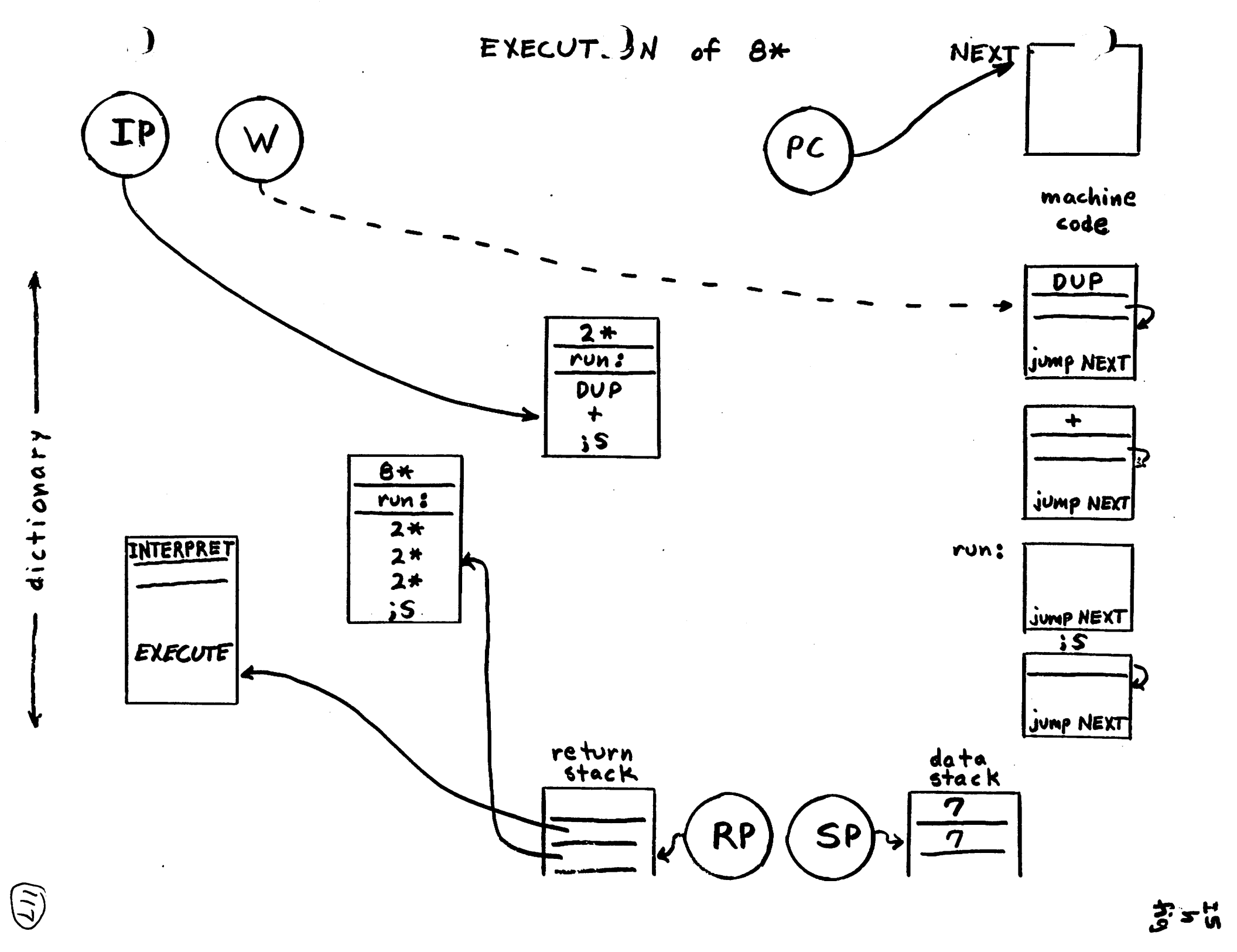
EXECUTION of 8*

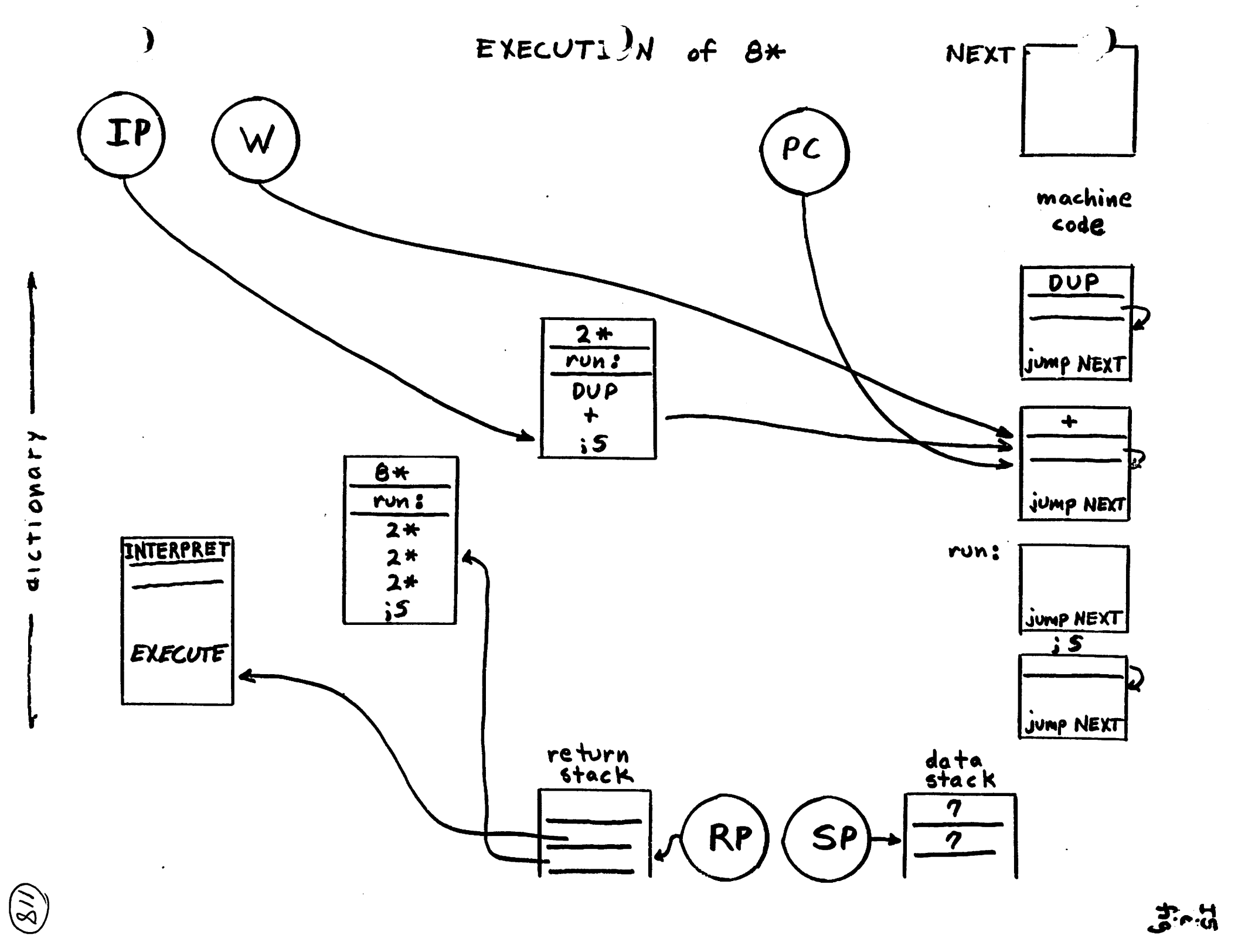


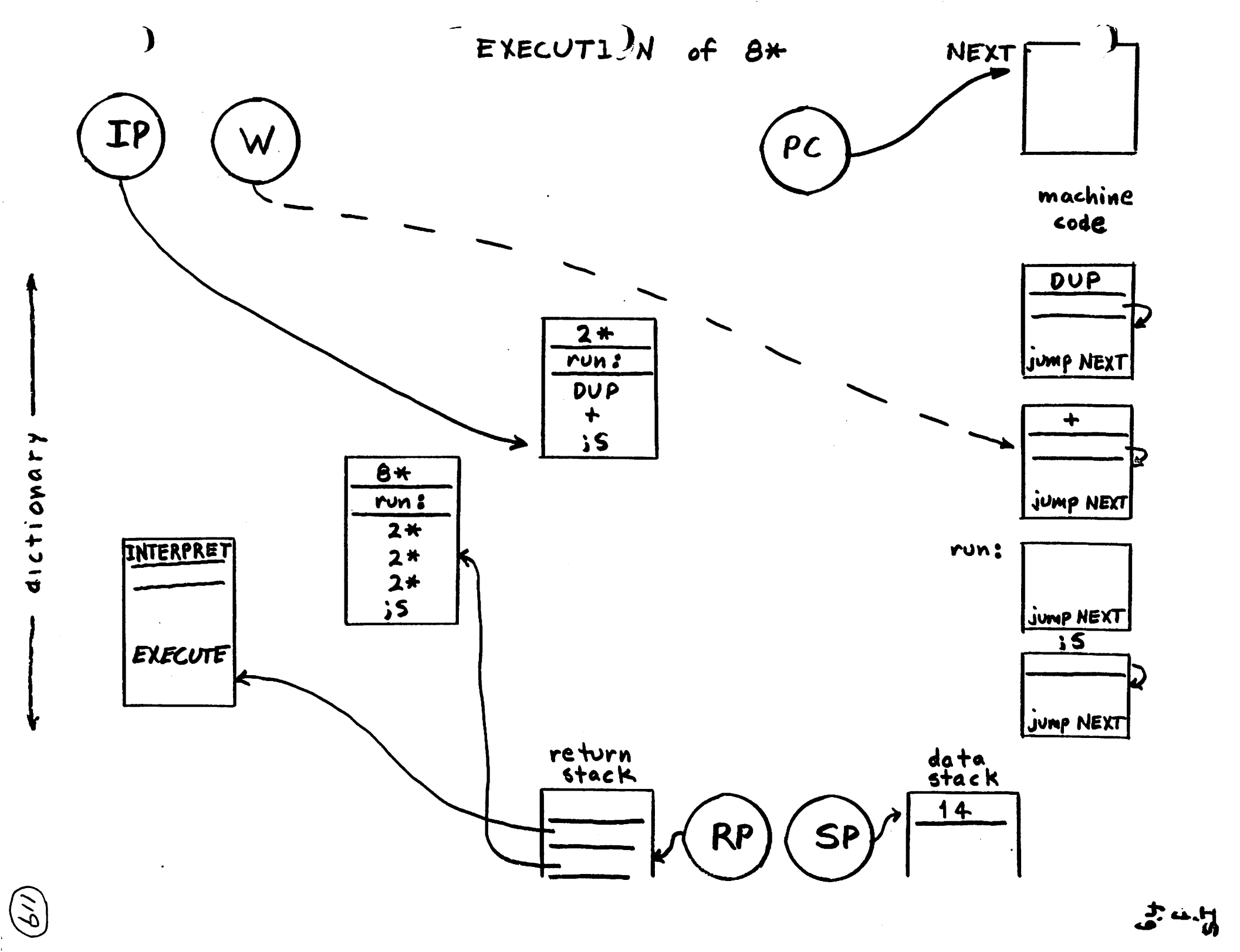
EXECUTION of 8*



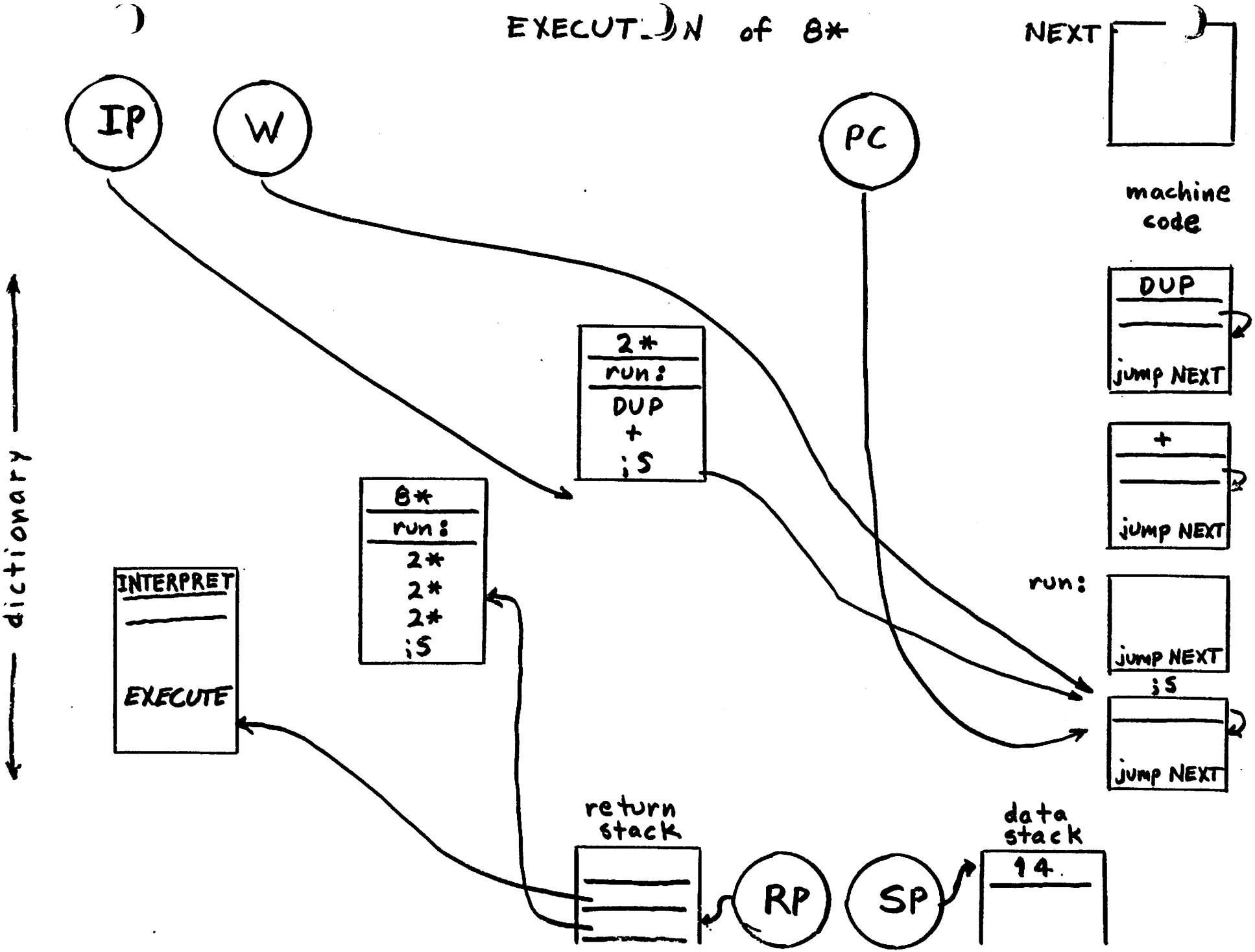

EXECUTION of 8*
NEXT
IP
W
PC
machine code
DUP
jump NEXT
2*
run:
DUP
+
;S
+
jump NEXT
8*
run:
2*
2*
2*
;S
run:
jump NEXT
;S
jump NEXT
INTERPRET
EXECUTE
dictionary
return stack
RP
SP
data stack
7


EXECUTION of 8*
NEXT
IP
W
PC
machine code
DUP
jump NEXT
2*
run:
DUP
+
;S
+
jump NEXT
8*
run:
2*
2*
2*
;S
run:
jump NEXT
;S
jump NEXT
INTERPRET
EXECUTE
dictionary
return stack
data stack
7
7
RP
SP

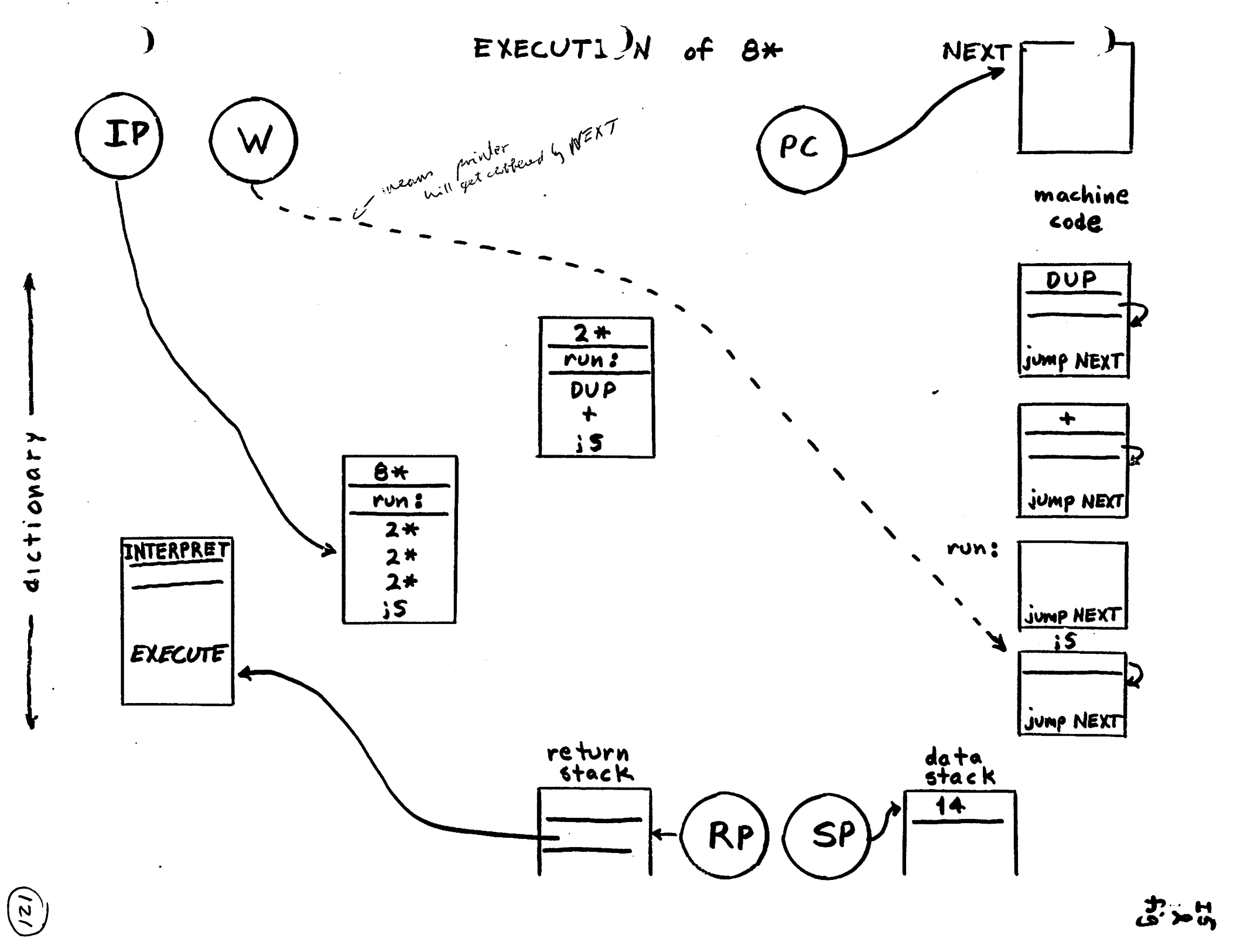

EXECUTION of 8*
NEXT
IP
W
PC
machine code
DUP
jump NEXT
2*
run:
DUP
+
;S
+
jump NEXT
8*
run:
2*
2*
2*
;S
run:
jump NEXT
;S
jump NEXT
INTERPRET
EXECUTE
dictionary
return stack
data stack
RP
SP
7
7


EXECUTION of 8*
NEXT
IP
W
PC
machine code
DUP
jump NEXT
2*
run:
DUP
+
;S
+
jump NEXT
8*
run:
2*
2*
2*
;S
run:
jump NEXT
;S
INTERPRET
EXECUTE
jump NEXT
dictionary
return stack
data stack
14
RP
SP

EXECUTION of 8*



EXECUTION of 8*



Why is this address interpreter fast inspite of the above overhead ? NEXT is 2 instructions on PDP-11

Answer : all we do is fetch addresses

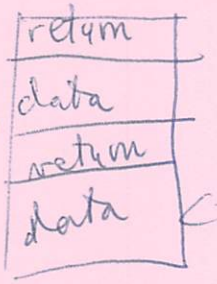
E.g. P-code interpreter of PASCAL has to a detailed set of case statements

microcoded versions of FORTH are running on HP 2100
2109

– may be coming on LXI-11

Why FORTH has 2 stacks

Borroughs :



to access this data, requires multiple indirect fetches

– "frame activation & deactivation"

– turns out to be less efficient

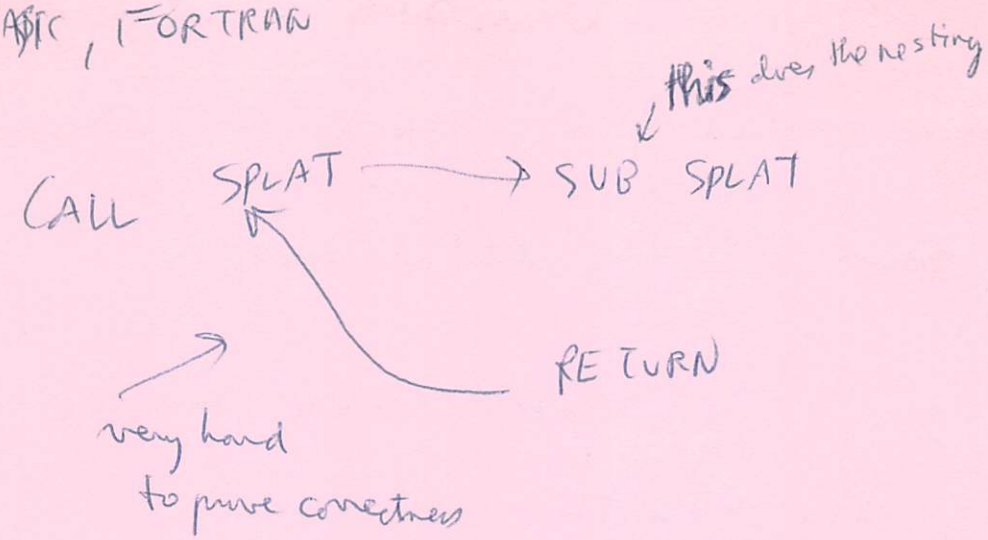
one major improvement over ALGOL

Use of W & indirect threaded code allows the resting operation at the caller

for structured nesting & unnesting

(over)

e.g. in BASIC, FORTRAN



in FORTH, we don't have "CALL" → so we can test SPLAT independently

240 B/SCR

VARIABLE #START

: ELEMENT (index (subscript) --- addr) 2* B/BUF/MOD #START @ + BLOCK + UPDATE;

(Test virtual array)

: INIT-ARRAY 500 0 DO I I ELEMENT ! LOOP;

: .ARRAY 0 DO CR I . SPACE I ELEMENT ? LOOP;

(Make the virtual array into a file)

: AVAILABLE (--- adr) #START @ BLOCK UPDATE;

(e.g.)

0 AVAILABLE !

(storing numbers)

: PUT AVAILABLE +!

AVAILABLE @ ELEMENT !;

eg. 3 PUT

(Inspecting file entries)

: SEE AVAILABLE @ 1+ 1 DO CR I. I ELEMENT ? LOOP;

forth class 6/29/80

p. I26

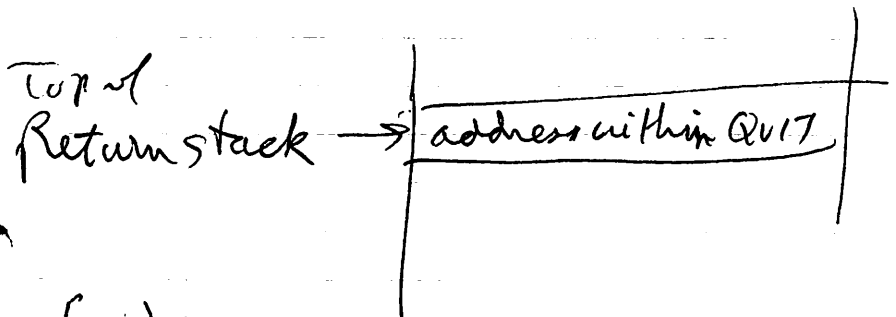
see screen #52

: INTERPRET -FIND

IF STATE @ <

-how to get out (at end of line):

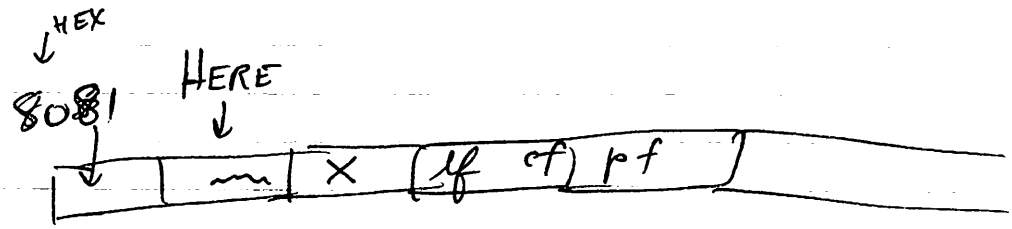
see QUIT



-USES word "null"
screen 45

When screen 45 loaded:

8081 ~~if~~ is found by interpreter & put on data stack
HERE -- executed -- addr it



! puts 8081 in addr of HERE

81

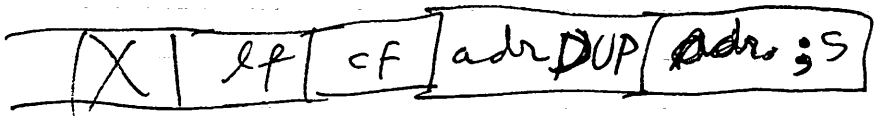
-this puts in a word with name 1 byte long & name is ASCII null

80

- purpose of null: to get us out of whatever we're doing

Example for p. C9 r1

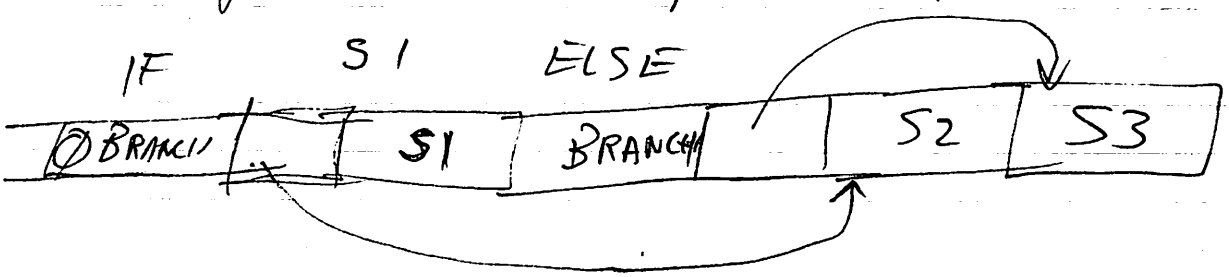
: X DUP ; (2) ← time frame



p. C12 r1 NOTE: Data stack not used by words ~~by compr~~ when compiling so available for

p. C14 r1

Desired dictionary result for "ELSE"



CORRECTIONS revised < & > that work!

: < 2DUP XOR 0< IF DROP ELSE - THEN 0< ;

: U< 2DUP XOR 0< IF DROP 0< 0= ELSE - 0< THEN ;

SCR #56 : S->D DUP 0< MINUS ;

SCR #60 : FLUSH #BUF 1+ 0 DO \
BUFFER DROP LOOP ;