Standard model for procedure activation

Peter J. Denning

July 2019

What is standard model?

- Procedures are active in the interval between call and return
- Storage for variables and computation is allocated only when procedure is active
  - Called an activation record (AR) or frame
- Procedures can call others, or even themselves (recursion)
  - Returns are in reverse order from calls (LIFO order) -- hence ARs can be pushed on a stack at call and popped on return

LOG program example

We present the standard model along with a worked example of the protocols for procedure call and return for a program LOG that computes logarithms. This model is described in the chapter on Machines in *Great Principles of Computing* (Denning and Martel, MIT Press, 2015).

The table on the next slide is the instruction set of a simple stack machine used in the example. (Excerpted from the book chapter.)

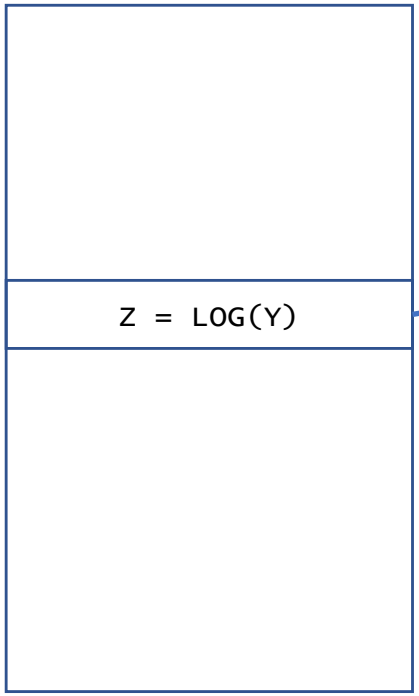
These protocols demonstrate the close cooperation needed between the hardware designer and the software designer so that procedure call and return can be lightning fast.

Table 1 is an instruction set for a stack machine configuration that features a stack pointer (SP) and activation record pointer (AR) in the CPU, in addition to the instruction pointer (IP). The "Op Code" is an abbreviation for the name of the instruction. The effect of executing the instruction is shown in the "Before" and "After" columns, which show the stack configuration just before and just after the instruction is executed. The letter "S" represents the state of the stack prior to the current instruction. Mem[*a*] means the contents stored in memory location *a*. Essential side effects of changing the instruction pointer and changing the contents of a memory location are shown in the "Memory Effects" column.

Table 1: Instruction Set of Stack Machine

Type	Op Code	Name	Before	After	Memory Effects
Arithmetic and logical operators	ADD	Add	S a b	S c	
	SUB	Subtract			
	MUL	Multiply			
	DIV	Divide			
	EQ	Test for equal			
	NE	Test for not equal			
Memory interface	LA a	Load address a	S	S a	
	L	Load	S a	S v	v = Mem[a]
	ST	Store	S a v	S	leaving Mem[a]=v
	EXC	Exchange	S a b	S b a	reverse top two
Sequencing	GO	Go	S a	S	leaving IP=a
	GOF	Go on false	S a v	S	leaving IP=a if v=0
	CALL	Call procedure	S e a	S AR	new frame AR on stack
	RET	Return to caller	S AR	S	restore caller's state
Completion	EXIT	Exit	empty	empty	

Program F contains a call to a LOG function



Ideal compilation if machine had a LOG instruction

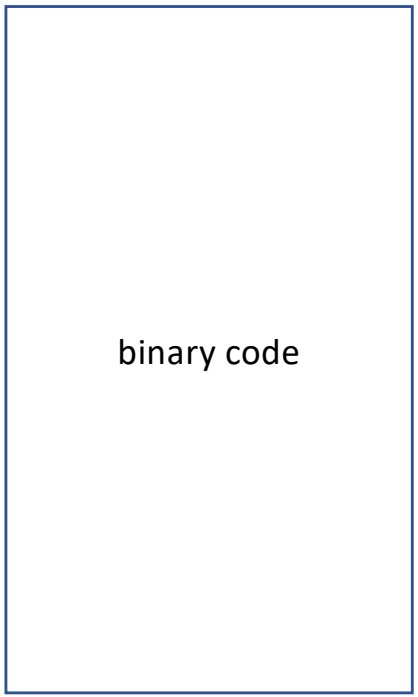
```
LA  Z
L   Y
LOG
ST
```

How to simulate LOG with a function call?

Instructions for stack machine instruction set on previous slide

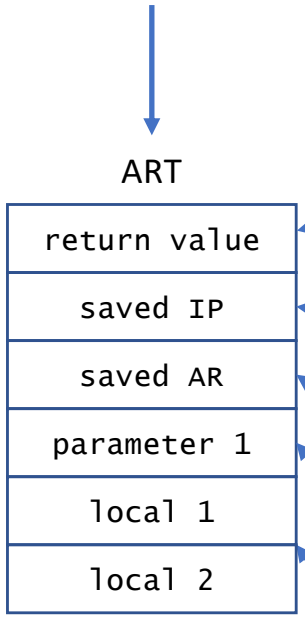
Replace LOG with an instruction sequence that calls a LOG function in software

First, compile the program for LOG into LOG.EXE (binary) and ART (activation record template)



binary code segment of all functions is read-only (execute-only on systems that support it) to protect the trusted software from tampering

ART is the structure for the first few stack slots of the AR of the called procedure (in this case 6 slots)



Convention: all functions leave their final value in the first slot

IP is the CPU register holding the address of the caller's next instruction

AR is the CPU register holding the base address of the caller's activation record

This is the parameter to LOG

Assume binary code for LOG has two local variables

CALL instruction

- Caller uses **calling sequence** to
  - build activation record A on stack
  - place entry point e on stack
  - place AR size a on stack
- Just before CALL, stack configuration is S1 S2 A e a
  - S1 = stack prior to caller's current AR
  - S2 = caller's current AR (register AR points to base S2)
- After CALL, configuration is S1 S2 A, with
  - A = Register AR now points to base of A
  - SP now points to end of A

Operation of CALL and RET

CALL =

```
B = SP-(pop Mem[SP])
Mem[B+1] = IP
Mem[B+2] = AR
IP = pop Mem[SP]
AR = B
```

Changes stack from S1 S2 A e a to S1 S2 A and sets IP=e

Prior to call, AR register points to base of S2; after to base of A

Saves the caller IP and AR registers in slots 1 and 2 of the new AR

RET =

```
SP = AR
AR = Mem[SP+2]
IP = Mem[SP+1]
```

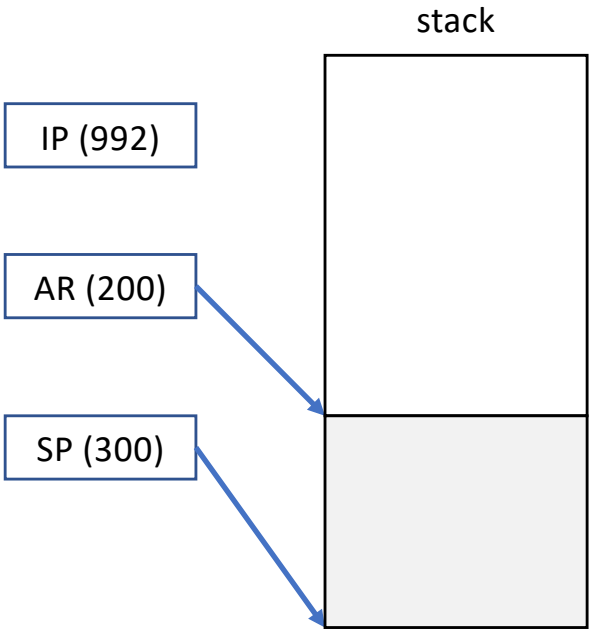
Restores the caller state by reloading the PSW and AR registers

Leaves SP at base of called A, which holds the return value

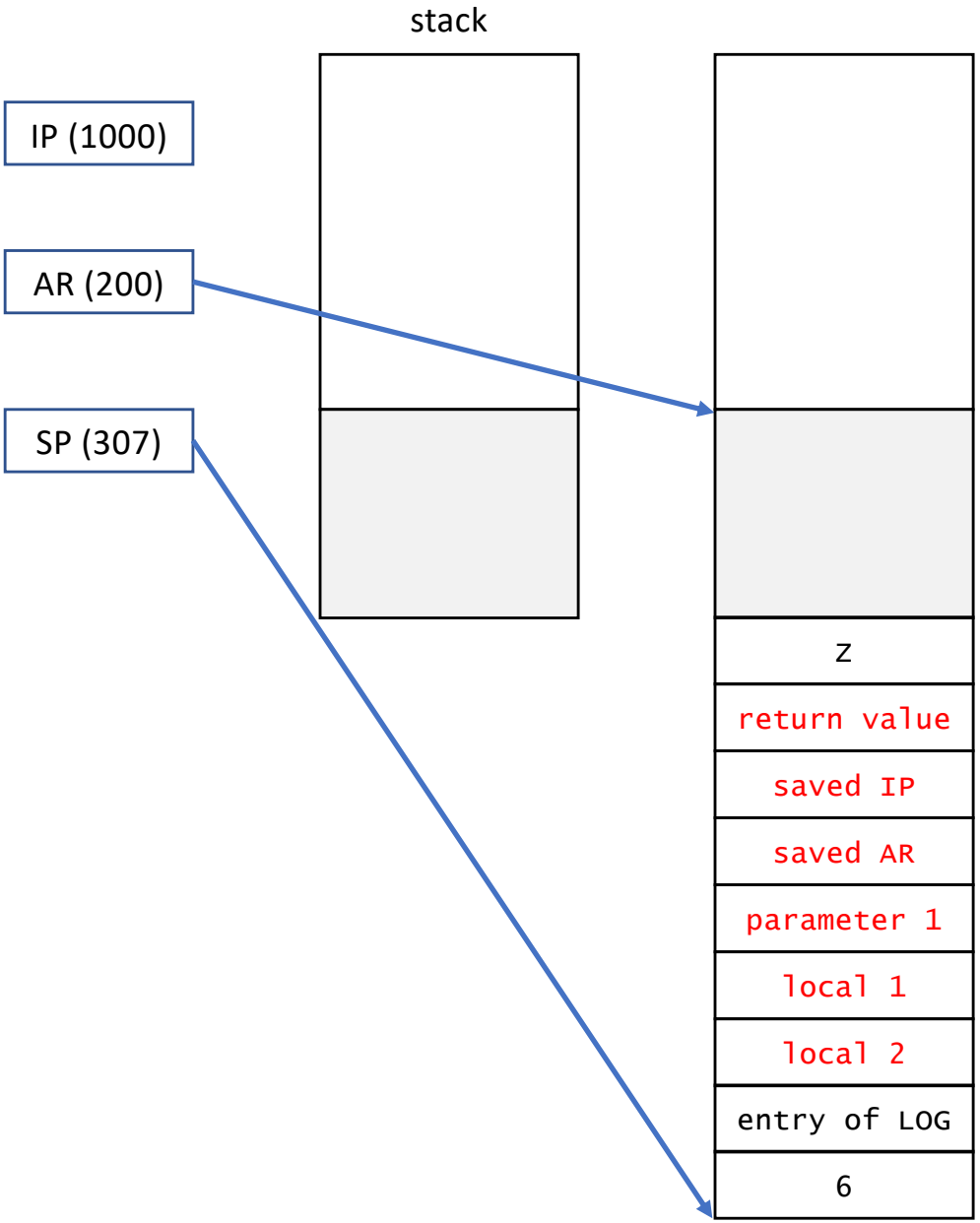
Example calling sequence

The compiler builds the new activation record on the stack according to the template, then calls LOG. Before it starts it puts the target Z address on top because the net effect of calling LOG will be to leave the value of LOG(Y) on top of the stack, in the proper configuration for the store operation.

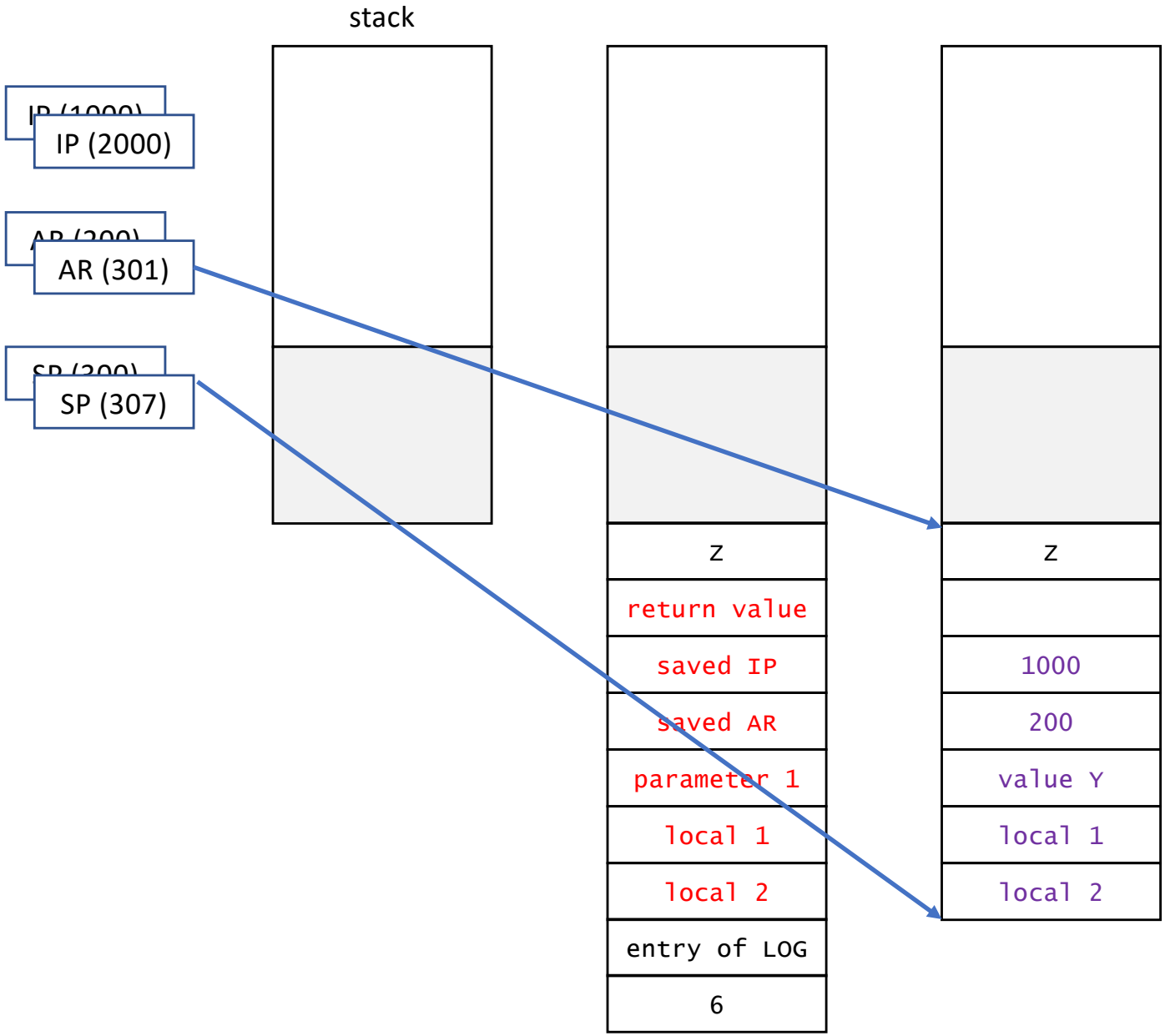
```
LA Z          target address for result of LOG(Y)
SP = SP+3     reserve first 3 slots of AR
L Y           load the value of parameter Y (= LA Y, L)
L loc1        load the value of local 1
L loc2        load the value of local 2
LA LOG        load entry point of LOG.EXE code
LA 6          load size of new AR (6 slots)
CALL          call instruction
ST            store result (target address Z already on stack)
```



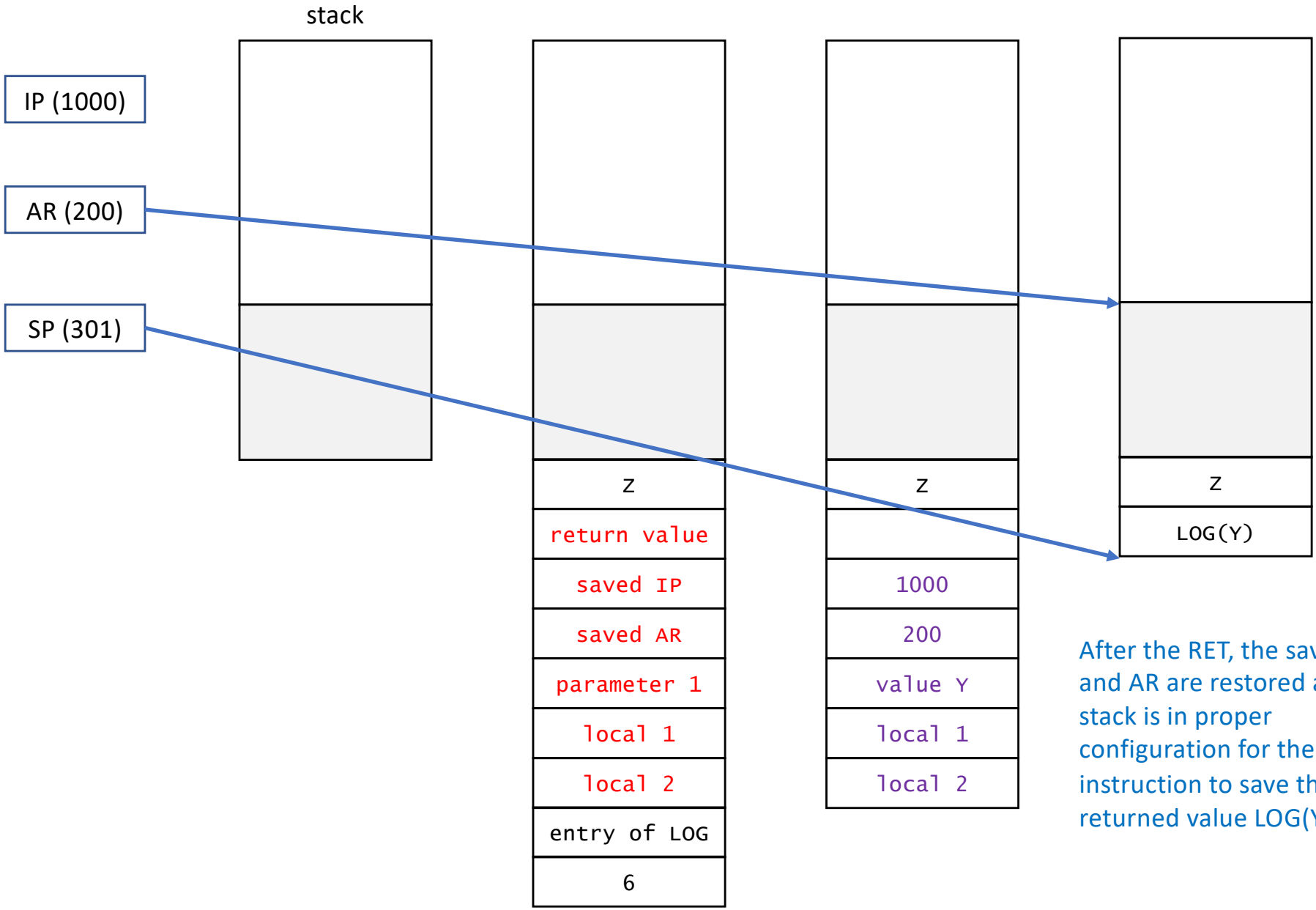
Assume CALL instruction in prior slide is at location 999, the base of the current AR is 200, and the stack pointer SP is 300. The stack configuration just before the calling sequence begins is shown above.



The purpose of the calling sequence is to grow the stack with a copy of the LOG function's AR template, shown here in red. CALL has not been executed yet.

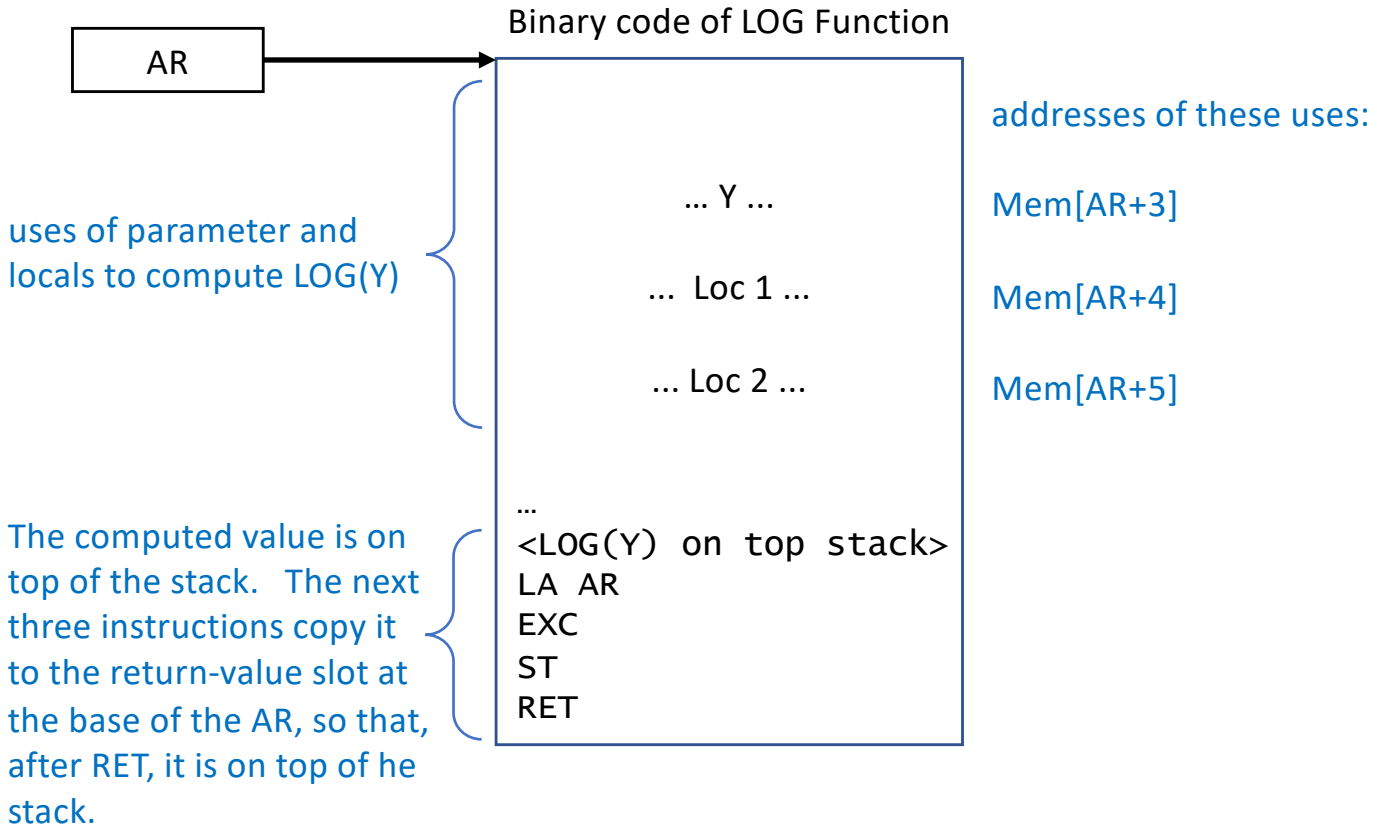


Assume entry point to LOG is at location 2000. The stack has this configuration just after the CALL. The base of the new AR is 6 slots below the SP at the time of call.



After the RET, the saved IP and AR are restored and the stack is in proper configuration for the ST instruction to save the returned value LOG(Y) into Z.

Returning from a call



Summary

- We illustrated the key ideas of subprogram call and return using the LOG procedure
- Called procedure has activation record pushed on stack prior to call
- CALL instruction saves caller's state and starts executing the called procedure's code
- RET instruction restores caller's state and leaves computed value on top of stack ready to be stored in the variable specified by caller