Return address, call instruction, stack, stack pointer, push, pop, LIFO saving and restoring of registers, nesting of subroutines, return instruction.

Most of the older processors have a stack pointer register which is to be initialized to an appropriate value at the beginning of the program. Execution of the CALL instruction makes the CPU hardware (i) decrement stack pointer, (ii) save return address on the stack, (iii) change PC to the subroutine's starting address.

The return instruction at the end of the subroutine (i) picks up the return address from the stack and puts it into the PC (ii) increments the stack pointer. All this is done in HARDWARE.
STACK IS

a) inside the CPU

b) in the memory
SP (Stack Pointer) is in
a) inside the CPU
b) in the memory
STACK GROWS IN THE DIRECTION OF

a) decreasing addresses

b) increasing addresses
STACK IS A

@ LIFO

b FIFO

DATA STRUCTURE
SP (Stack Pointer)
Points to the

a) Top of the Stack

b) Bottom of the Stack
SP POINTS TO THE

(a) Top-most Filled Location

(b) Next Empty Location
Loc \$ \Phi \rightarrow \text{MEMORY}

SP initialization

000-FFF = 2^{12} = 4K\text{Byte}

Allocated to STACK
When you **push**, do you

(a) increment SP

(b) decrement SP
WHEN YOU **PUSH**

do you

(a) first decrement SP
   and then place data

(b) first place data and
    then decrement SP
WHEN YOU POP
first you_____
and then you_____
3-25-K
STACK SHALL BE BALANCED.
To keep **Stack Balanced**, there shall be as many ___ as there are ___.

As many ___ as there are ___.
RETURN ADDR is

(a) The ADDR of CALL INSTR
(b) The NEXT ADDR

Call SUBR
CALL SUBR =

(i) \[ i \rightarrow SP \] (decrement/increment)

(ii) \[ Save/retrieve \] on the stack

(iii) Change (PC/SP) to the starting ADDR
$RTN =$

Return from sub routine

SAVE / RETRIEVE

Return address

From the stack and put it in the

(PC / SP)

(i)

(ii)

INCREMENT / DECREMENT

The SP.
NESTED

SUBROUTINE CALLS

MAKE THE STACK

(GROW / SHRINK)
Recursive function calls make the stack (grow/shrink)
AN EXAMPLE OF A RECURSIVE FUNCTION

\[ n! = n \times [(n-1)!] \]

1! = 1
Suppose we forget about **nested subroutines** and also interrupts & traps, we can do away with stack and stack pointer. There will be only one return address at most at any time and it can be saved in a special register called **link register ($31 in the case of MIPS)**.

Instead of a CALL instruction, MIPS has a **jump and link instruction jal**

The return instruction in MIPS is the **jump register instruction jr, $31**. jr $31 picks up the return address from $31.

If you plan to call another subroutine from within a subroutine, then you have to manually “off-load” the link register $31 on the stack, call the routine, when you return back retrieve the original return address from the stack and put it back in the link register $31.
OPERATION THEATRE

INSTRUMENT TABLE (TEMPORARY STORAGE)
Example of nested routines

A:
  ...
  jal B    # call B & save return address in $31
  ...
B:
  ...
  add $29, $29, $24   # adjust stack to make room for next item
  sw $31, 0($29)    # save the return address
  jal C    # call C & save return address in $31
  lw $31, 0($29)   # restore B's return address
  add $29, $29, $24  # adjust stack to pop B's return address
  ...
  jr $31  # return to routine that called B
C:
  ...
  jr $31  # return to routine that called C

Register saving

1. Caller save. The calling procedure (caller) is responsible for saving and restoring any registers that must be preserved across the call. The called procedure (callee) can then modify any register without constraint.

2. callee save. The callee is responsible for saving and restoring any registers that it might use. The caller uses registers without worrying about restoring them after a call.
A calls B which in-turn calls C

A
Main program
jal B
RA1

B
Subroutine
jal C
RA2 Restore link
Save link reg #31
jr $31

C
Subroutine
jr $31
direction of decreasing addresses