Laboratory 9 Notes
X86 Stack

Stack Operations

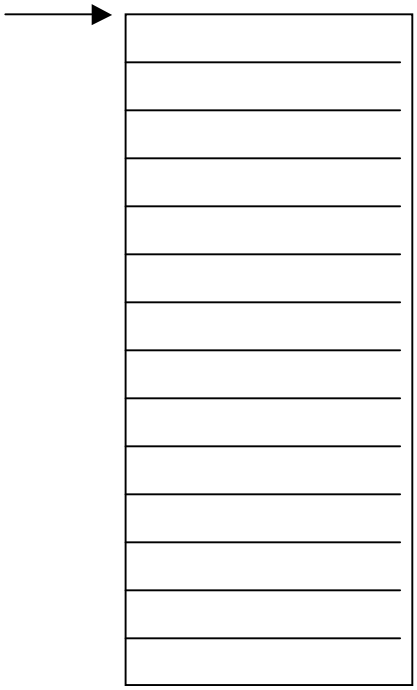
push *src*

1. Make space on the stack by decrementing %esp (stack pointer).
2. Move *src* to the stack

%esp ← %esp - 4
(%esp) ← src

Initial state of the stack

%esp=0xfffffffc

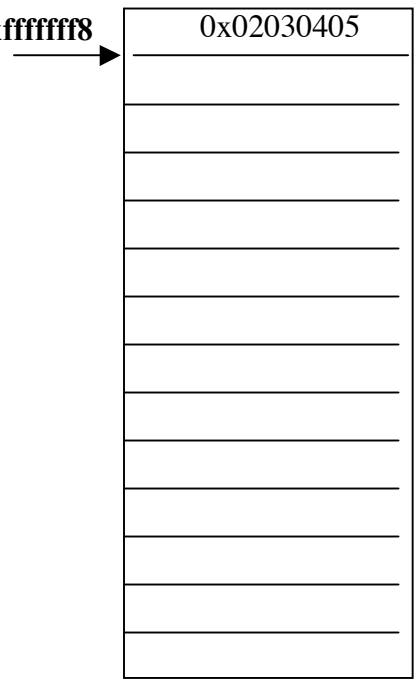


Push a word-size value in %eax on the stack (decrement %esp and move Src to (%esp)

(assume %eax = 0x02030405)

Push %eax

%esp=0x0xfffffff8

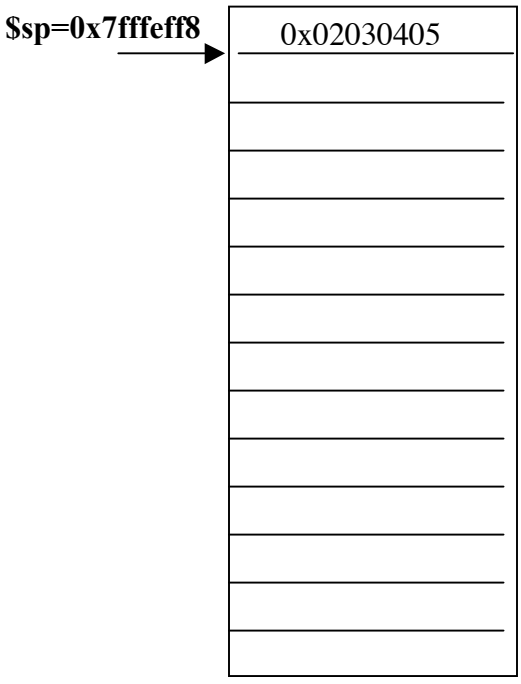


pop *dest*

1. Move contents of top of stack to the *dest*
2. Release space on the stack by incrementing %esp.

dest ← (%esp)
%esp ← %esp + 4

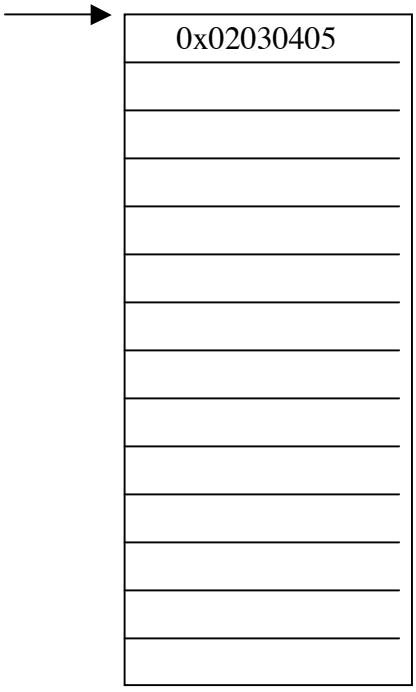
Initial State of Stack



Pop a word-size value from the stack.

Pop %ebx

(%ebx gets 0x02030405)



Instructions used for Function call and return

call *function*
1. Pushes the return address on stack (the address of the instruction *following* the function call)
2. Puts the starting address of the function in %eip:

%esp ← %esp - 4
(%esp) ← %eip (already updated for next instruction)
%eip ← address of function

leave
1. Releases the function stack frame by moving the $esp (top of frame stack) back to the base pointer %ebp (bottom of the frame stack)
2. Reset the %ebp to the old %ebp, which is popped off the stack.

(%esp) ← %ebp
%ebp ← (%esp)
%esp ← %esp + 4

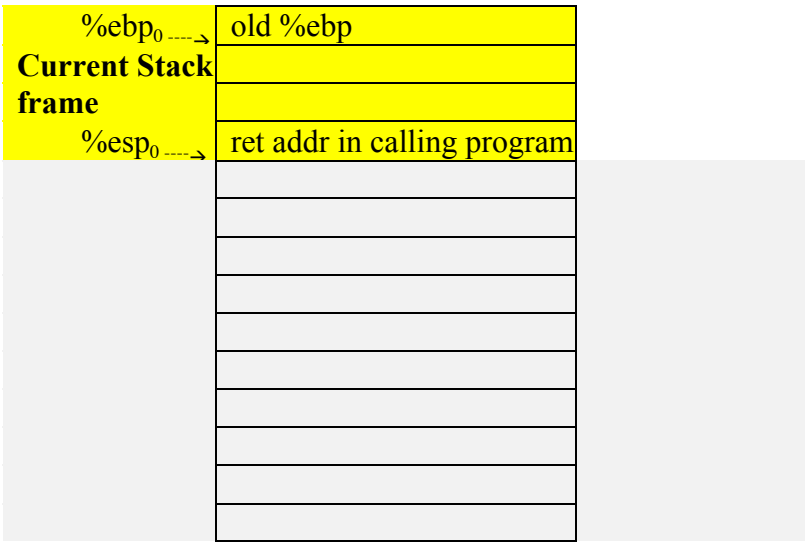
ret
1. Pops the return address off the top of the stack and puts it in %eip (resumes execution of the caller function.

%eip ← (%esp)
%esp ← %esp + 4

Conventions for drawing stack diagrams

To record the contents of the stack to understand how the stack is used, using the following notation:

- We use the model of memory where the stack has low addresses at the bottom and high at the top. Each row in the stack represents a word. The initial **%esp** with a subscript of **0** is pointing to the top of the current stack frame, and the **%ebp** is pointing to the bottom of the current stack frame.



- Trace the effect on the stack of executing each instruction in the program by moving the position of the **%esp** and **%ebp** when they change, (incrementing the subscript for each new value), and by recording new values on the stack as they are stored there.

- When the stack starts to empty, continue with the same notation, except use the right hand side of the stack diagram to indicate the changes.

- Also record changes to relevant registers.

setup for function getAndSumValues:

0x08048414 <+0>: push %ebp
0x08048415 <+1>: mov %esp,%ebp
0x08048417 <+3>: sub $0x28,%esp

print the prompt

0x0804841a <+6>: mov $0x8048554,%eax
0x0804841f <+11>: mov %eax,(%esp)
0x08048422 <+14>: call 0x8048338 <printf@plt>

put parameters for *scanf* on stack and accept input form the user

parameter 1 = addr of formatting string
0x08048427 <+19>: mov $0x8048567,%eax
parameter 2 = addr on stack where input stored (local variable: call it **n**)
0x0804842c <+24>: lea -0xc(%ebp),%edx # addr on stack where input stored
0x0804842f <+27>: mov %edx,0x4(%esp)
0x08048433 <+31>: mov %eax,(%esp)
0x08048436 <+34>: call 0x8048348 <__isoc99_scanf@plt>
0x0804843b <+39>: mov -0xc(%ebp),%eax

*# **n** and %eax contains the value entered by the user*

*# BASE CASE: if user entered a 0, initialize **result** (%eax) to 0 and return it*
0x0804843e <+42>: test %eax,%eax
0x08048440 <+44>: jne 0x8048449 <getAndSumValues+53>
0x08048442 <+46>: mov $0x0,%eax
0x08048447 <+51>: jmp 0x8048453 <getAndSumValues+63>
-
- *#RECURSIVE CASE: get another value and add it to the **result***
0x08048449 <+53>: call 0x8048414 <getAndSumValues>
0x0804844e <+58>: mov -0xc(%ebp),%edx
0x08048451 <+61>: add %edx,%eax
0x08048453 <+63>: leave
0x08048454 <+64>: ret