



## Assembly Language: Function Calls

1



## Goals of this Lecture

- Help you learn:
  - Function call problems:
    - Calling and returning
    - Passing parameters
    - Storing local variables
    - Handling registers without interference
    - Returning values
  - IA-32 solutions to those problems
    - Pertinent instructions and conventions

2



## Function Call Problems

1. Calling and returning
  - How does caller function *jump* to callee function?
  - How does callee function *jump back* to the right place in caller function?
2. Passing parameters
  - How does caller function pass *parameters* to callee function?
3. Storing local variables
  - Where does callee function store its *local variables*?
4. Handling registers
  - How do caller and callee functions use the *same registers* without interference?
5. Returning a value
  - How does callee function send *return value* back to caller function?

3



## Problem 1: Calling and Returning

- How does caller function *jump* to callee function?
- I.e., Jump to the address of the callee's first instruction
- How does the callee function *jump back* to the right place in caller function?
- I.e., Jump to the instruction immediately following the most-recently-executed call instruction

4

### Attempted Solution: Use Jmp Instruction



- Attempted solution: caller and callee use jmp instruction

```
P:          # Function P
...
  jmp R     # Call R
Rtn_point1:
...

R:          # Function R
...
  jmp Rtn_point1 # Return
```

5

### Attempted Solution: Use Jmp Instruction



- Problem: callee may be called by multiple callers

```
P:          # Function P
...
  jmp R     # Call R
Rtn_point1:
...

R:          # Function R
...
  jmp ???  # Return

Q:          # Function Q
...
  jmp R     # Call R
Rtn_point2:
...
```

6

### Attempted Solution: Use Register



- Attempted solution 2: Store return address in register

```
P:          # Function P
...
  movl $Rtn_point1, %eax
  jmp R     # Call R
Rtn_point1:
...

R:          # Function R
...
  jmp *%eax # Return

Q:          # Function Q
...
  movl $Rtn_point2, %eax
  jmp R     # Call R
Rtn_point2:
...
```

Special form of jmp instruction; we will not use

7

### Attempted Solution: Use Register



- Problem: Cannot handle nested function calls

```
P:          # Function P
...
  movl $Rtn_point1, %eax
  jmp Q     # Call Q
Rtn_point1:
...

R:          # Function R
...
  jmp *%eax # Return

Q:          # Function Q
...
  movl $Rtn_point2, %eax
  jmp R     # Call R
Rtn_point2:
...
  jmp %eax  # Return
```

Problem if P calls Q, and Q calls R

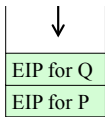
Return address for P to Q call is lost

8

## IA-32 Solution: Use the Stack



- May need to store many return addresses
  - The number of nested functions is not known in advance
  - A return address must be saved for as long as the invocation of this function is live, and discarded thereafter
- Addresses used in reverse order
  - E.g., function P calls Q, which then calls R
  - Then R returns to Q, which then returns to P
- Last-in-first-out data structure (stack)
  - Caller pushes return address on the stack
  - ... and callee pops return address off the stack
- IA 32 solution: call and ret instructions use the stack

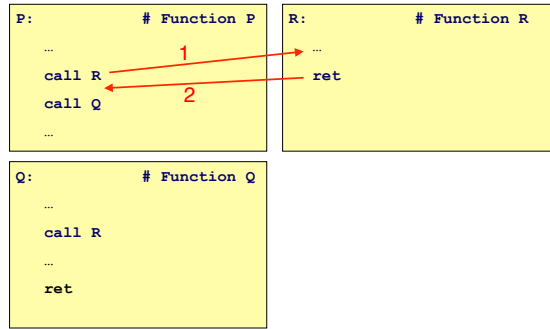


9

## IA-32 Call and Ret Instructions



- Ret instruction “knows” the return address

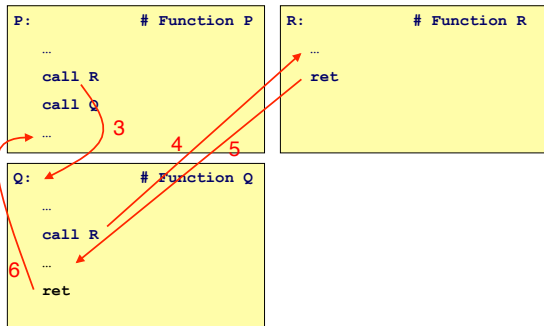


10

## IA-32 Call and Ret Instructions



- Ret instruction “knows” the return address



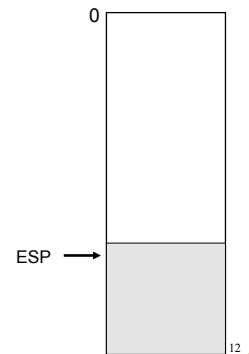
11

## Implementation of Call



- ESP (stack pointer register) points to top of stack

Instruction	Effective Operations
<code>pushl src</code>	<code>subl \$4, %esp</code>
	<code>movl src, (%esp)</code>



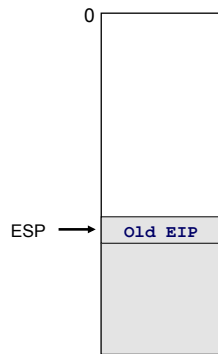
12

## Implementation of Call



- ESP (stack pointer register) points to top of stack

Instruction	Effective Operations
pushl src	subl \$4, %esp movl src, (%esp)



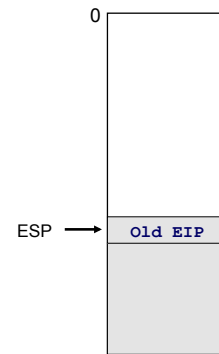
13

## Implementation of Call



- ESP (stack pointer register) points to top of stack

Instruction	Effective Operations
pushl src	subl \$4, %esp movl src, (%esp)
popl dest	movl (%esp), dest addl \$4, %esp



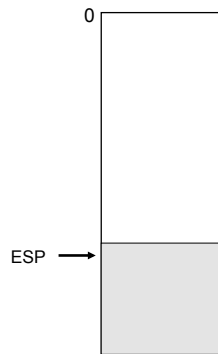
14

## Implementation of Call



- ESP (stack pointer register) points to top of stack

Instruction	Effective Operations
pushl src	subl \$4, %esp movl src, (%esp)
popl dest	movl (%esp), dest addl \$4, %esp



15

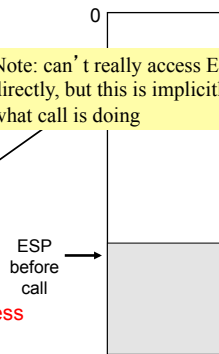
## Implementation of Call



- EIP (instruction pointer register) points to next instruction to be executed

Instruction	Effective Operations
pushl src	subl \$4, %esp movl src, (%esp)
popl dest	movl (%esp), dest addl \$4, %esp
call addr	pushl %eip jmp addr

Note: can't really access EIP directly, but this is implicitly what call is doing



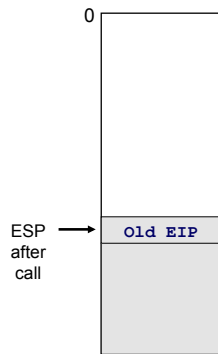
16

Call instruction pushes return address (old EIP) onto stack, then jumps

## Implementation of Call



Instruction	Effective Operations
pushl src	subl \$4, %esp movl src, (%esp)
popl dest	movl (%esp), dest addl \$4, %esp
call addr	pushl %eip jmp addr



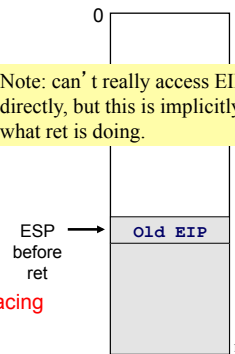
17

## Implementation of Ret



Instruction	Effective Operations
pushl src	subl \$4, %esp movl src, (%esp)
popl dest	movl (%esp), dest addl \$4, %esp
call addr	pushl %eip jmp addr
ret	popl %eip

Note: can't really access EIP directly, but this is implicitly what ret is doing.



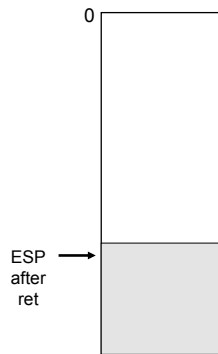
18

Ret instruction pops stack, thus placing return address (old EIP) into EIP

## Implementation of Ret



Instruction	Effective Operations
pushl src	subl \$4, %esp movl src, (%esp)
popl dest	movl (%esp), dest addl \$4, %esp
call addr	pushl %eip jmp addr
ret	popl %eip



19

## Function Call Problems



1. Calling and returning
  - How does caller function *jump* to callee function?
  - How does callee function *jump back* to the right place in caller function?
2. Passing parameters
  - How does caller function pass *parameters* to callee function?
3. Storing local variables
  - Where does callee function store its *local variables*?
4. Handling registers
  - How do caller and callee functions use the *same registers* without interference?
5. Returning a value
  - How does callee function send *return value* back to caller function?

20

## Problem 2: Passing Parameters



- Problem: How does caller function pass *parameters* to callee function?

```
int add3(int a, int b, int c)
{
    int d;
    d = a + b + c;
    return d;
}

int f(void)
{
    return add3(3, 4, 5);
}
```

21

## Attempted Solution: Use Registers



- Attempted solution: Pass parameters in registers

```
f:
    movl $3, %eax
    movl $4, %ebx
    movl $5, %ecx
    call add3
    ...
```

```
add3:
    ...
    # Use EAX, EBX, ECX
    ...
    ret
```

22

## Attempted Solution: Use Registers



- Problem: Cannot handle nested function calls

```
f:
    movl $3, %eax
    movl $4, %ebx
    movl $5, %ecx
    call add3
    ...

add3:
    ...
    movl $6, %eax
    call g
    # Use EAX, EBX, ECX
    # But EAX is corrupted!
    ...
    ret
```

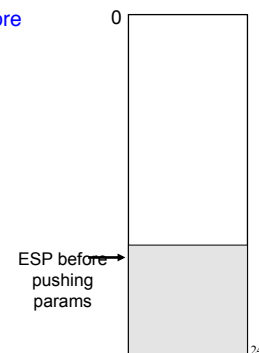
- Also: How to pass parameters that are longer than 4 bytes?

23

## IA-32 Solution: Use the Stack



- Caller pushes parameters before executing the call instruction

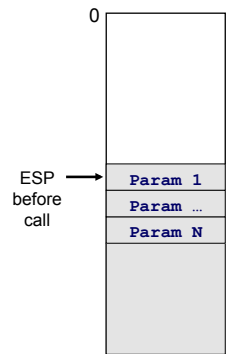


24

## IA-32 Parameter Passing



- Caller pushes parameters in the reverse order
  - Push N<sup>th</sup> param first
  - Push 1<sup>st</sup> param last
  - So first param is at top of the stack at the time of the Call

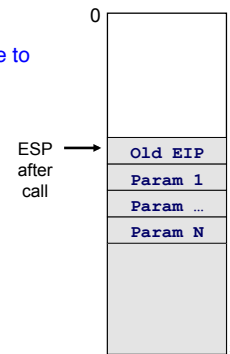


25

## IA-32 Parameter Passing



- Then call the callee
- Callee addresses params relative to ESP: Param 1 as 4(%esp)

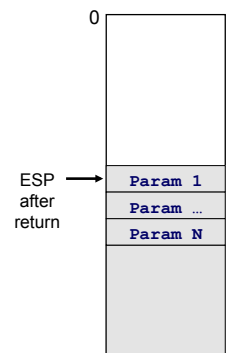


26

## IA-32 Parameter Passing



- After returning to the caller...

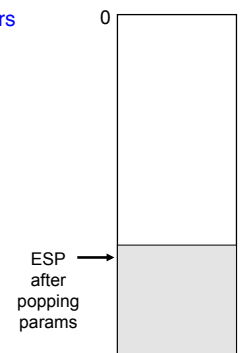


27

## IA-32 Parameter Passing



- ... the caller pops the parameters from the stack



28

## IA-32 Parameter Passing



For example:

```
f:
...
# Push parameters
pushl $5
pushl $4
pushl $3
call add3
# Pop parameters
addl $12, %esp

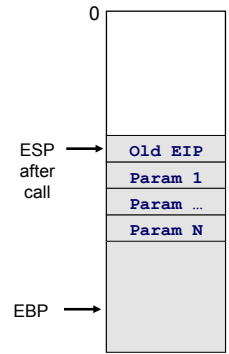
add3:
...
movl 4(%esp), wherever
movl 8(%esp), wherever
movl 12(%esp), wherever
...
ret
```

29

## Base Pointer Register: EBP



- **Problems:**
  - As callee executes, ESP may change
    - E.g., preparing to call another function
  - Error-prone for callee to reference params as offsets relative to ESP
- **Solution:**
  - Use a register called EBP to hold what stack pointer was at beginning of callee's execution
  - EBP doesn't move during callee's execution
  - Use EBP as fixed reference point to access params



30

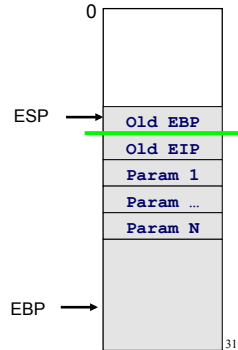
## Using EBP



- **Need to save old value of EBP**
  - Before overwriting EBP register

• Callee executes "prolog"

```
→ pushl %ebp
   movl %esp, %ebp
```



31

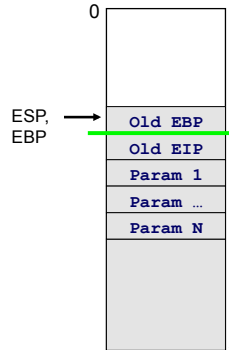
## Base Pointer Register: EBP



- Callee executes "prolog"

```
pushl %ebp
→ movl %esp, %ebp
```

- Regardless of ESP, callee can reference param 1 as 8(%ebp), param 2 as 12(%ebp), etc.



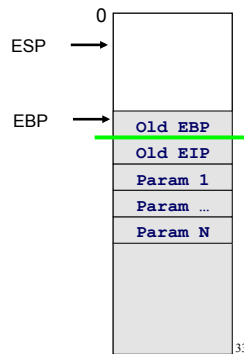
32



## Base Pointer Register: EBP



- Before returning, callee must restore ESP and EBP to their old values
  - Callee executes "epilog"
    - `movl %ebp, %esp`
    - `popl %ebp`
    - `ret`
- %ESP needs to be restored to its "pre-callee" value, which was stored in %EBP



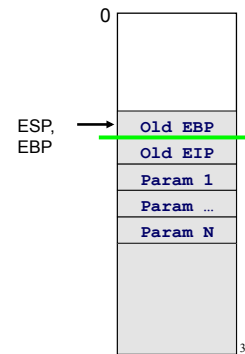
33

## Base Pointer Register: EBP



- Callee executes "epilog"
  - `movl %ebp, %esp`
  - `popl %ebp`
  - `ret`

%EBP needs to be restored to its "pre-callee" value, which was stored on the stack.

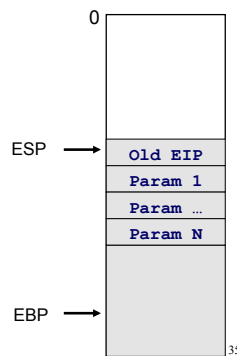


34

## Base Pointer Register: EBP



- Callee executes "epilog"
  - `movl %ebp, %esp`
  - `popl %ebp`
  - `ret`



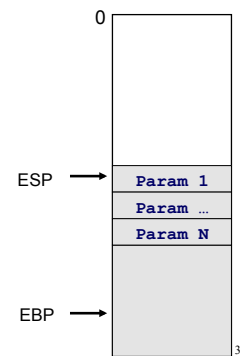
35

## Base Pointer Register: EBP



- Callee executes "epilog"
  - `movl %ebp, %esp`
  - `popl %ebp`
  - `ret`

And we're back to where we were before the call



36

### Problem 3: Storing Local Variables



- Where does callee function store its *local variables*?

```
int add3(int a, int b, int c)
{
    int d;
    d = a + b + c;
    return d;
}

int foo(void)
{
    return add3(3, 4, 5);
}
```

37

### IA-32 Solution: Use the Stack



- Local variables:**
  - Short-lived, so don't need a permanent location in memory
  - Size known in advance, so don't need to allocate on the heap
- So, the function just uses the top of the stack**
  - Store local variables on the top of the stack
  - The local variables disappear after the function returns

```
int add3(int a, int b, int c)
{
    int d;
    d = a + b + c;
    return d;
}

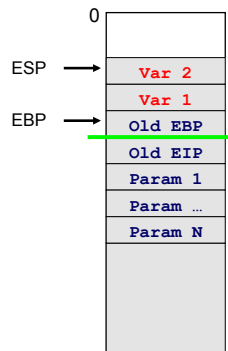
int foo(void)
{
    return add3(3, 4, 5);
}
```

38

### IA-32 Local Variables



- Local variables of the callee are allocated on the stack
- Allocation done by moving the stack pointer
- Example: allocate memory for two integers
  - subl \$4, %esp
  - subl \$4, %esp
  - (or equivalently, subl \$8, %esp)
- Reference local variables as negative offsets relative to EBP
  - 4(%ebp)
  - 8(%ebp)



39

### IA-32 Local Variables



For example:

```
add3:
...
    # Allocate space for d
    subl $4, %esp
...
    # Initialize d
    movl whatever, -4(%ebp)
...
    ret
```

40

## Problem 4: Handling Registers



- **Problem:** How do caller and callee functions use *same registers* without interference?
- **Registers are a finite resource**
  - In principle: Each function should have its own registers
  - In reality: All functions share same small set of registers
- **Callee may use register that the caller also is using**
  - When callee returns control to caller, old register contents may have been lost
  - Caller function cannot continue where it left off

41

## IA-32 Solution: Use the Stack



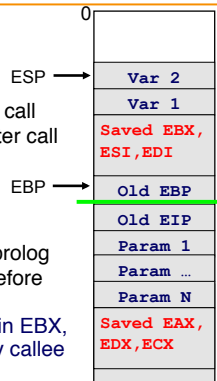
- **Save the registers on the stack**
  - Someone must save old register contents
  - Someone must later restore the register contents
- **Define a convention for who (caller or callee) saves and restores which registers**

42

## IA-32 Register Handling



- **Caller-save registers**
  - **EAX, EDX, ECX**
  - If necessary...
    - Caller saves on stack before call
    - Caller restores from stack after call
- **Callee-save registers**
  - **EBX, ESI, EDI**
  - If necessary...
    - Callee saves on stack after prolog
    - Callee restores from stack before epilog
  - Caller can assume that values in EBX, ESI, EDI will not be changed by callee



43

## Problem 5: Return Values



- **Problem:** How does callee function send return value back to caller function?
- **In principle:**
  - Store return value in stack
  - return value in register
- **Or, for efficiency:**
  - Known small size => store return value in register
  - Other => store return value in stack

```
int add3(int a, int b, int c)
{
    int d;
    d = a + b + c;
    return d;
}

int foo(void)
{
    return add3(3, 4, 5);
}
```

44

## IA-32 Return Values



### IA-32 Convention:

- **Integral type or pointer:**
  - Store return value in EAX
  - char, short, int, long, pointer
- **Floating-point type:**
  - Store return value in floating-point register
  - (Beyond scope of COS 217)
- **Structure:**
  - Store return value on stack
  - (Beyond scope of COS 217)

```
int add3(int a, int b, int c)
{
    int d;
    d = a + b + c;
    return d;
}

int foo(void)
{
    return add3(3, 4, 5);
}
```

45

## Stack Frames



### Summary of IA-32 function handling:

- Stack has one **stack frame** per active function invocation
- ESP points to top (low memory) of current stack frame
- EBP points to bottom (high memory) of current stack frame
- Stack frame contains:
  - Return address (Old EIP)
  - Old EBP
  - Saved register values
  - Local variables
  - Parameters to be passed to callee function

46

## A Simple Example



```
int add3(int a, int b, int c)
{
    int d;
    d = a + b + c;
    return d;
}
```

```
/* In some calling function */
...
x = add3(3, 4, 5);
...
```

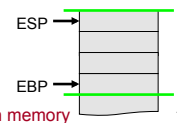
47

## Trace of a Simple Example 1



```
x = add3(3, 4, 5);
```

Low memory



48

## Trace of a Simple Example 2

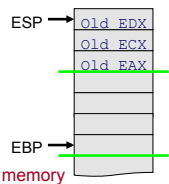


```
x = add3(3, 4, 5);
```

Low memory

# Save caller-save registers if necessary

```
pushl %eax
pushl %ecx
pushl %edx
```



49

## Trace of a Simple Example 3



```
x = add3(3, 4, 5);
```

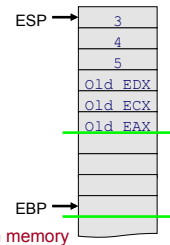
Low memory

# Save caller-save registers if necessary

```
pushl %eax
pushl %ecx
pushl %edx
```

# Push parameters

```
pushl $5
pushl $4
pushl $3
```



50

## Trace of a Simple Example 4



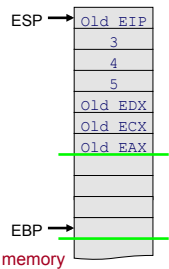
```
x = add3(3, 4, 5);
```

Low memory

# Save caller-save registers if necessary

```
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
```

```
# Call add3
call add3
```



51

## Trace of a Simple Example 5

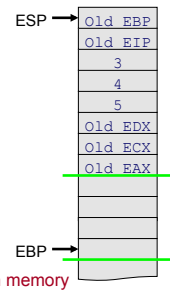


```
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

Low memory

```
# Save old EBP
pushl %ebp
```

} Prolog



52

### Trace of a Simple Example 6

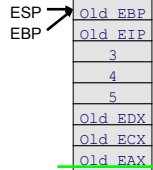


```
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

Low memory

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
```

Prolog



High memory

53

### Trace of a Simple Example 7

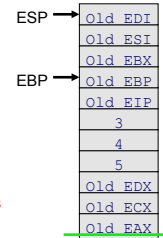


```
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

Low memory

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save callee-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
```

Unnecessary here; add3 will not change the values in these registers



High memory

54

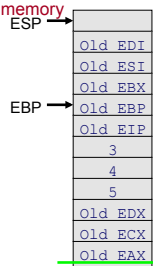
### Trace of a Simple Example 8



```
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

Low memory

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save callee-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
# Allocate space for local variable
subl $4, %esp
```



High memory

55

### Trace of a Simple Example 9

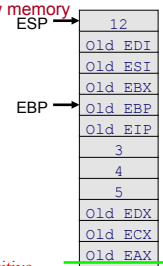


```
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

Low memory

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save callee-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
# Allocate space for local variable
subl $4, %esp
# Perform the addition
movl 8(%ebp), %eax
addl 12(%ebp), %eax
addl 16(%ebp), %eax
movl %eax, -16(%ebp)
```

Access params as positive offsets relative to EBP  
Access local vars as negative offsets relative to EBP



High memory

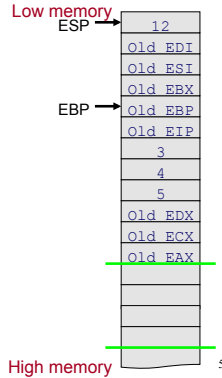
56

## Trace of a Simple Example 10



```
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

# Copy the return value to EAX  
 movl -16(%ebp), %eax  
 # Restore callee-save registers if necessary  
 movl -12(%ebp), %edi  
 movl -8(%ebp), %esi  
 movl -4(%ebp), %ebx



57

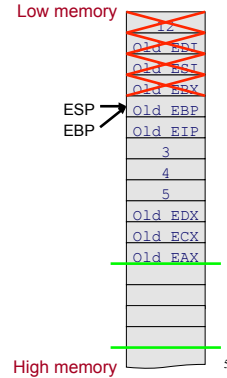
## Trace of a Simple Example 11



```
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

# Copy the return value to EAX  
 movl -16(%ebp), %eax  
 # Restore callee-save registers if necessary  
 movl -12(%ebp), %edi  
 movl -8(%ebp), %esi  
 movl -4(%ebp), %ebx  
 # Restore ESP  
 movl %ebp, %esp

} Epilog



58

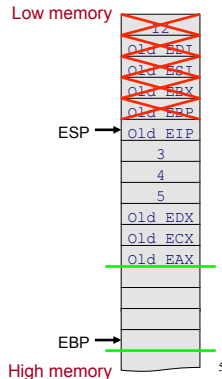
## Trace of a Simple Example 12



```
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

# Copy the return value to EAX  
 movl -16(%ebp), %eax  
 # Restore callee-save registers if necessary  
 movl -12(%ebp), %edi  
 movl -8(%ebp), %esi  
 movl -4(%ebp), %ebx  
 # Restore ESP  
 movl %ebp, %esp  
 # Restore EBP  
 popl %ebp

} Epilog



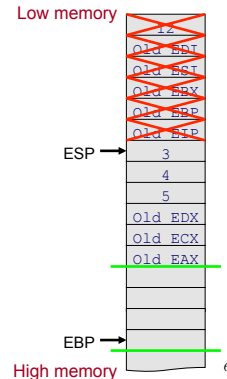
59

## Trace of a Simple Example 13



```
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}
```

# Copy the return value to EAX  
 movl -16(%ebp), %eax  
 # Restore callee-save registers if necessary  
 movl -12(%ebp), %edi  
 movl -8(%ebp), %esi  
 movl -4(%ebp), %ebx  
 # Restore ESP  
 movl %ebp, %esp  
 # Restore EBP  
 popl %ebp  
 # Return to calling function  
 ret



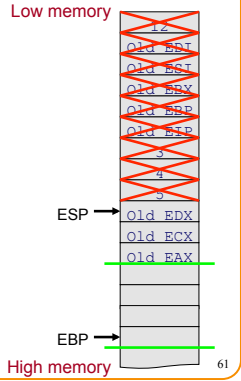
60

### Trace of a Simple Example 14



```
x = add3(3, 4, 5);
```

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl $12, %esp
```

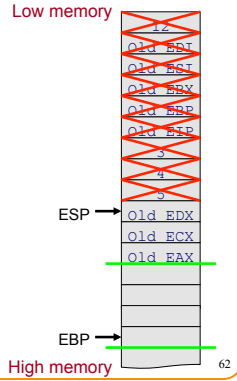


### Trace of a Simple Example 15



```
x = add3(3, 4, 5);
```

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl $12, %esp
# Save return value
movl %eax, wherever
```

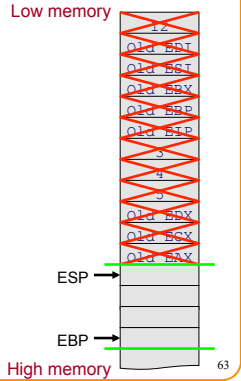


### Trace of a Simple Example 16



```
x = add3(3, 4, 5);
```

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl $12, %esp
# Save return value
movl %eax, wherever
# Restore caller-save registers if necessary
popl %edx
popl %ecx
popl %eax
```

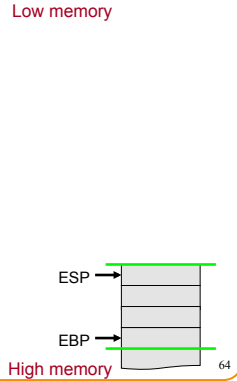


### Trace of a Simple Example 17



```
x = add3(3, 4, 5);
```

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl $12, %esp
# Save return value
movl %eax, wherever
# Restore caller-save registers if necessary
popl %edx
popl %ecx
popl %eax
# Proceed!
```





## Summary



- **Calling and returning**
  - Call instruction: push EIP onto stack and jump
  - Ret instruction: pop stack to EIP
- **Passing parameters**
  - Caller pushes onto stack
  - Callee accesses as positive offsets from EBP
  - Caller pops from stack

65

## Summary (cont.)



- **Storing local variables**
  - Callee pushes on stack
  - Callee accesses as negative offsets from EBP
  - Callee pops from stack
- **Handling registers**
  - Caller saves and restores EAX, ECX, EDX if necessary
  - Callee saves and restores EBX, ESI, EDI if necessary
- **Returning values**
  - Callee returns data of integral types and pointers in EAX

66