Procedure Calls

CS 217

1

Procedure Calls

- Procedure: a piece of code with a well-defined interface, and well-defined entry and exit points
- **Call:** jump to the beginning of an arbitrary procedure
- **Return:** jump back to the caller
- The jump address in the return operation is dynamically determined
  - Jump to the instruction immediately following the Call instruction in the Caller

2

Implementing Procedure Calls

```
P:          # Proc P
  …
  jmp R     # Call R
Rtn_point1:
  …
```

```
R:          # Proc R
  …
  jmp ???   # Return
```

```
Q:          # Proc Q
  …
  jmp R     # Call R
Rtn_point2:
  …
```

What should the return instruction in R jump to?

3

Implementing Procedure Calls

```
P:          # Proc P
  movl $Rtn_point1, %eax
  jmp R     # Call R
Rtn_point1:
  …
```

```
R:          # Proc R
  …
  jmp %eax  # Return
```

```
Q:          # Proc Q
  movl $Rtn_point2, %eax
  jmp R     # Call R
Rtn_point2:
  …
```

Convention: At Call time, store return address in %eax

4

Problem: Nested Procedure Calls

```
P:          # Proc P
   movl $Rtn_point1, %eax
   jmp Q    # Call Q
Rtn_point1:
   …
```

```
R:          # Proc R
   …
   jmp %eax  # Return
```

```
Q:          # Proc 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

Need to use a Stack

- P calls Q, Q calls R, R calls S, S calls P again
- A return address needs to be saved for as long as the procedure invocation continues
- Return addresses are used in Last-In-First-Out order
- Stack is a natural solution

Stack Frames

- Use stack for all temporary data related to each active procedure invocation
- Return address
- Procedure arguments
- Local variables of procedures
- Saving registers across invocations

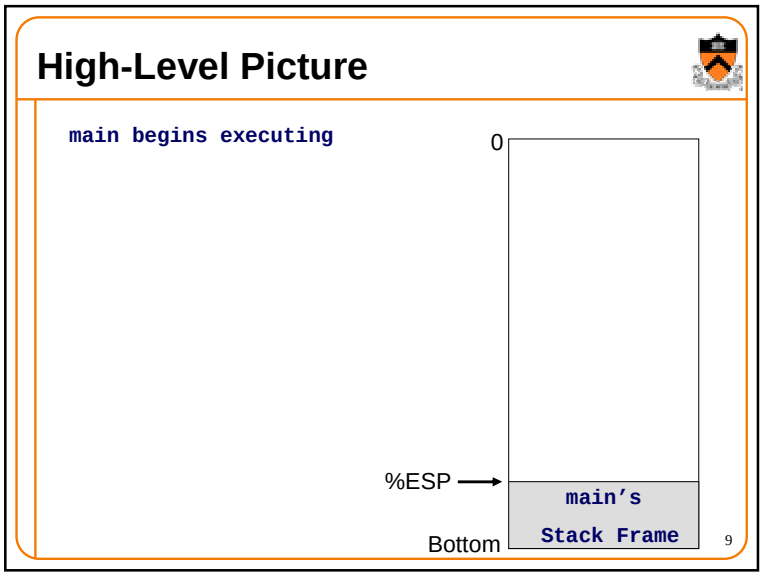
Stack Frame

- Stack has one Stack Frame for each active procedure invocation

High-Level Picture

- At Call time, push a new Stack Frame on top of the stack
- At Return time, pop the top-most Stack Frame

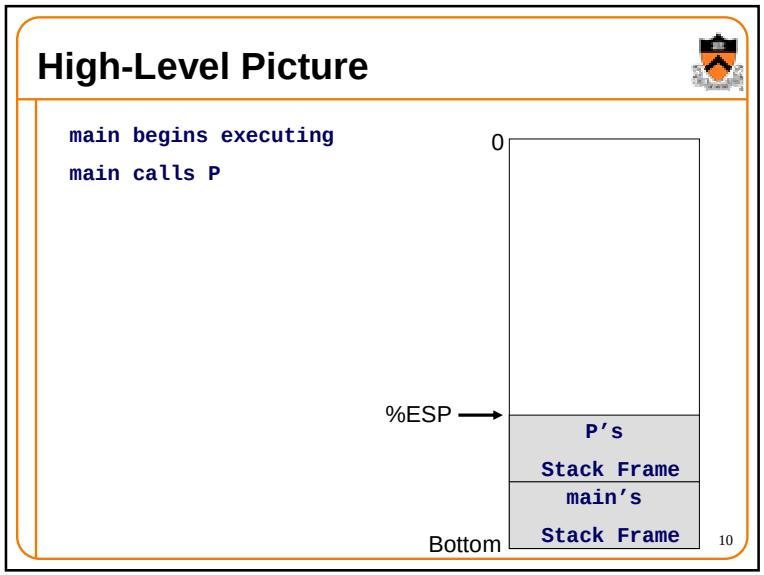
High-Level Picture



```
main begins executing
```


9

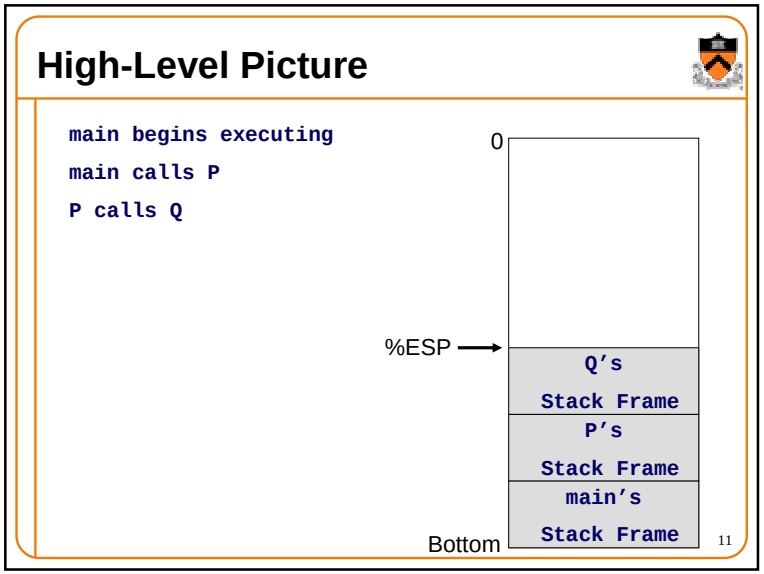
High-Level Picture



```
main begins executing
main calls P
```


10

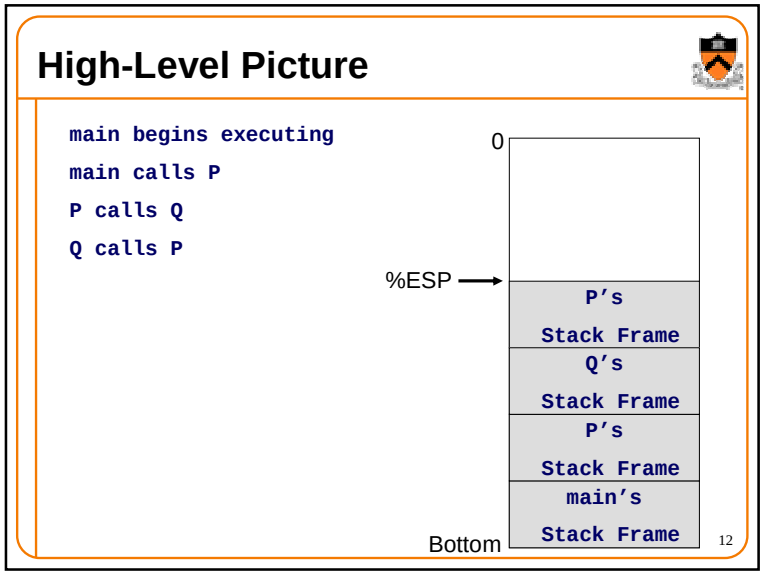
High-Level Picture



```
main begins executing
main calls P
P calls Q
```


11

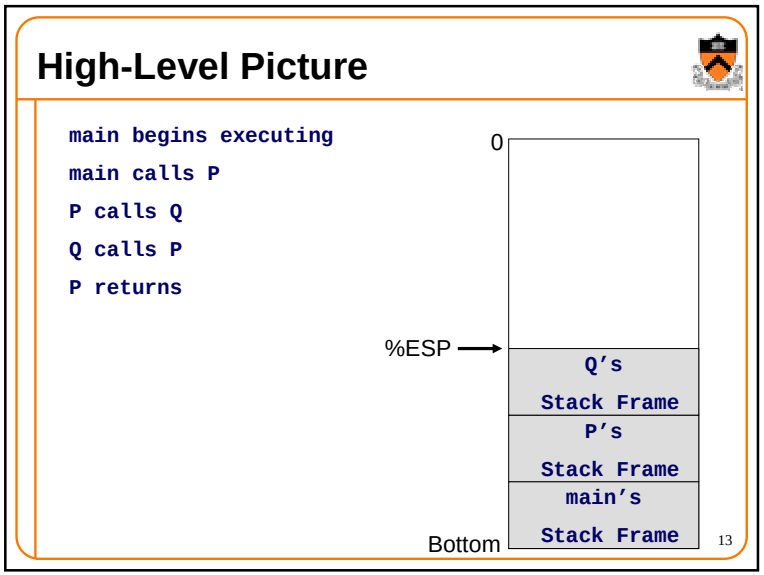
High-Level Picture



```
main begins executing
main calls P
P calls Q
Q calls P
```

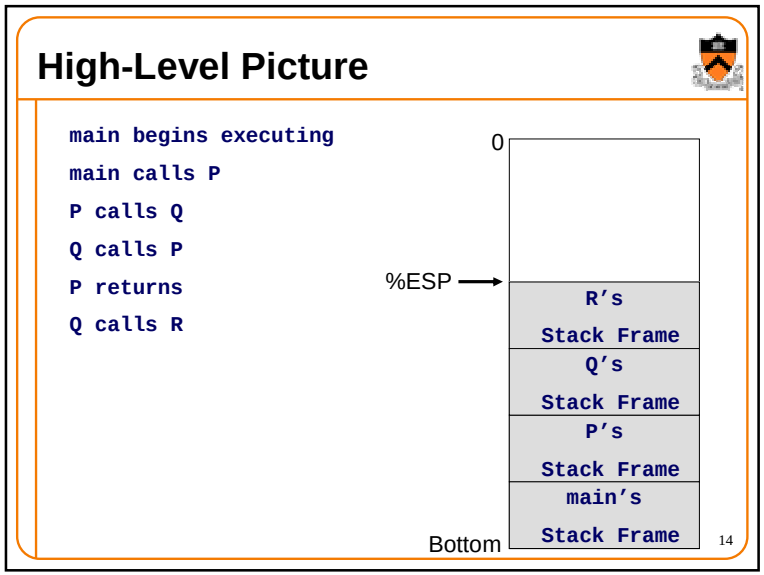

12

High-Level Picture



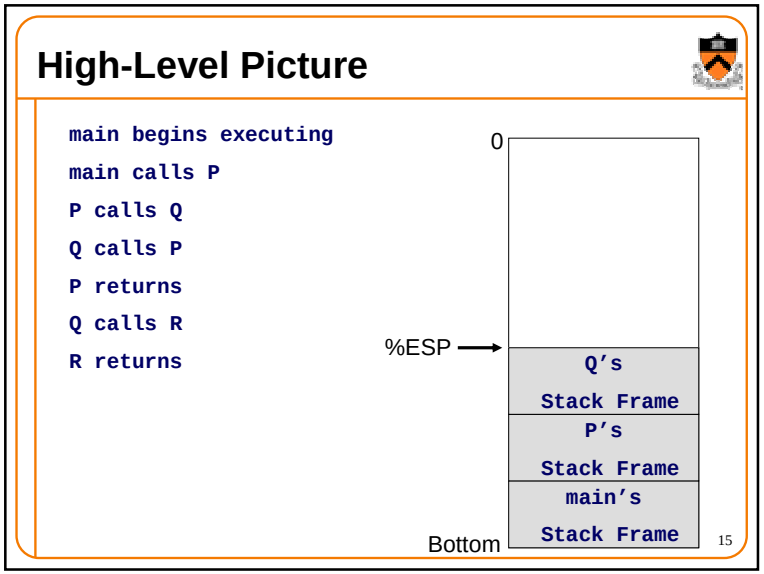
```
main begins executing
main calls P
P calls Q
Q calls P
P returns
```


High-Level Picture



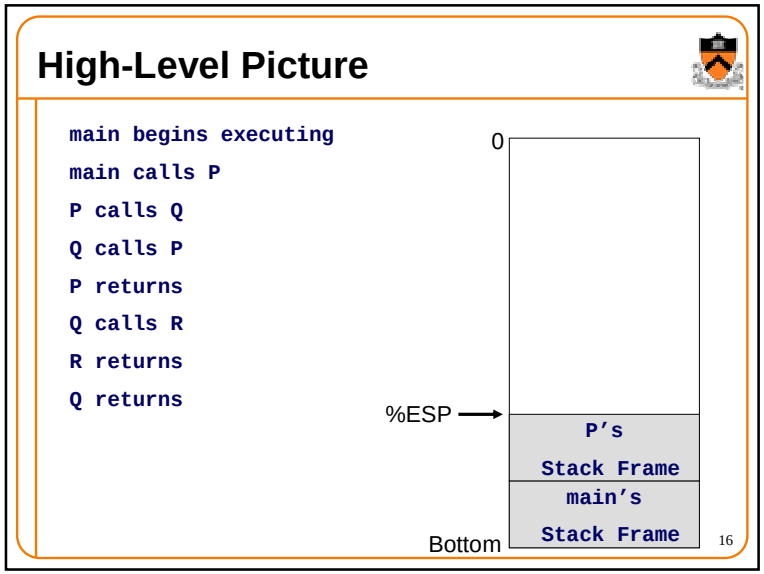
```
main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
```


High-Level Picture



```
main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
R returns
```

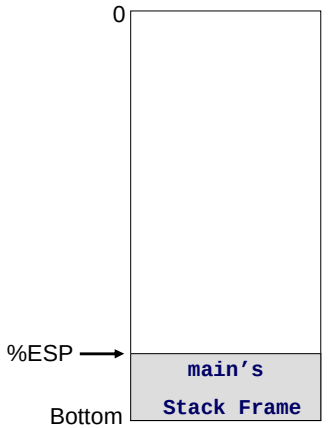

High-Level Picture



```
main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
R returns
Q returns
```

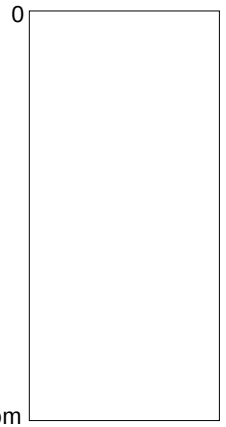
High-Level Picture

```
main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
R returns
Q returns
P returns
```



High-Level Picture

```
main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
R returns
Q returns
P returns
main returns
```

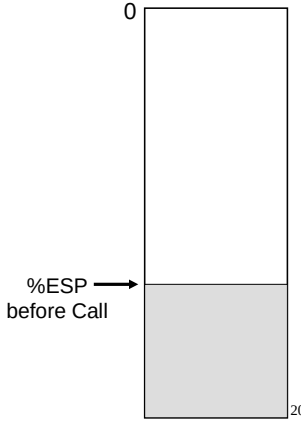


Procedure Call Details

- Call and Return instructions
- Argument passing between procedures
- Base pointer management (EBP)
- Local variables
- Register saving conventions

Call and Return Instructions

Instruction	Description
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`
`ret`	`pop %eip`



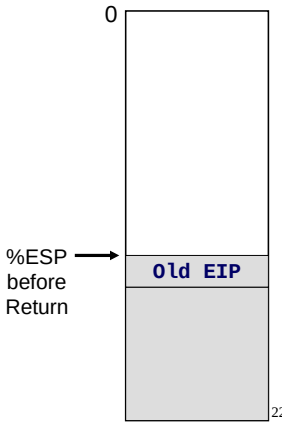
Call and Return Instructions

Instruction	Description
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`
`ret`	`pop %eip`

Call and Return Instructions

Instruction	Description
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`
`ret`	`pop %eip`

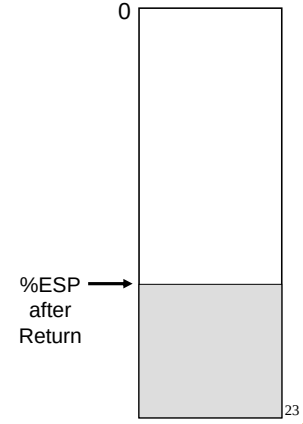
Return instruction assumes that the return address is at the top of the stack



Call and Return Instructions

Instruction	Description
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`
`ret`	`pop %eip`

Return instruction assumes that the return address is at the top of the stack

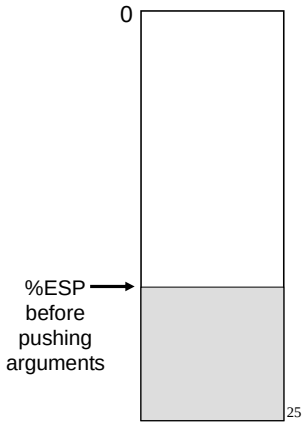


Procedure Call Details

- Call and Return instructions
- Argument passing between procedures
- Base pointer management (EBP)
- Local variables
- Register saving conventions

Input Parameters

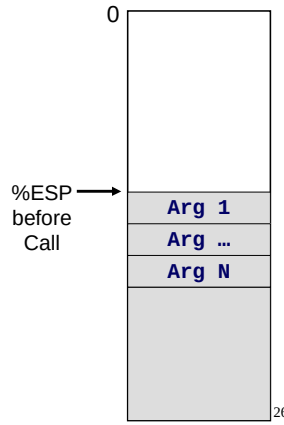
- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push Nth argument first
  - Push 1st argument last
  - So that the first argument is at the top of the stack at the time of the Call



25

Input Parameters

- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push Nth argument first
  - Push 1st argument last
  - So that the first argument is at the top of the stack at the time of the Call

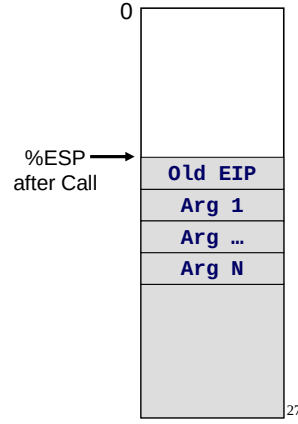


26

Input Parameters

- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push Nth argument first
  - Push 1st argument last
  - So that the first argument is at the top of the stack at the time of the Call

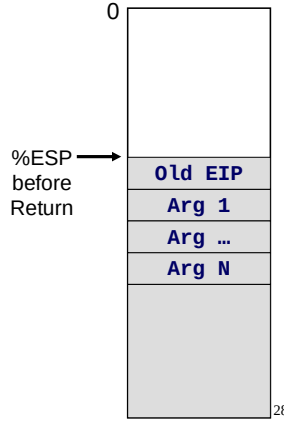
Callee can address arguments relative to ESP: Arg 1 as 4(%esp)



27

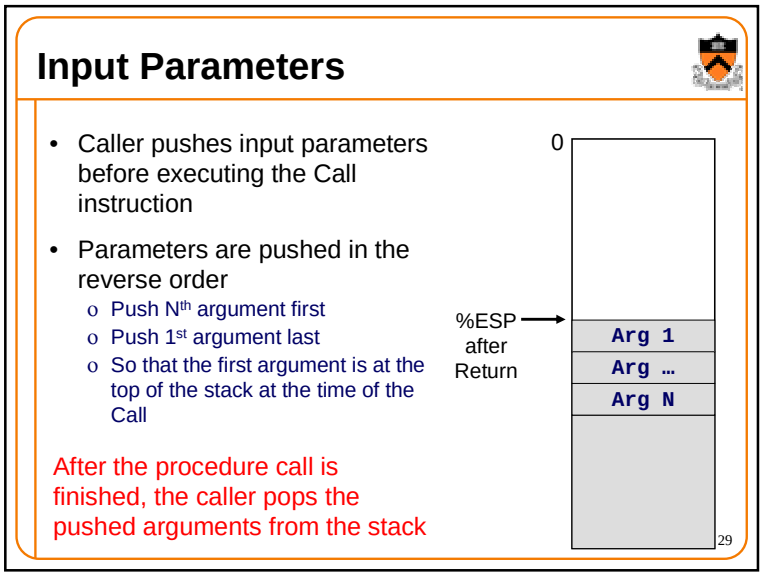
Input Parameters

- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push Nth argument first
  - Push 1st argument last
  - So that the first argument is at the top of the stack at the time of the Call



28

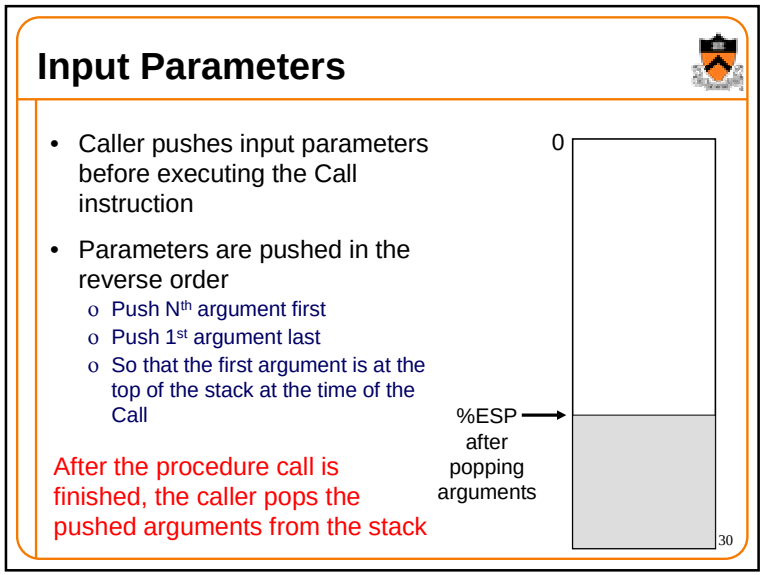
Input Parameters



- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push Nth argument first
  - Push 1st argument last
  - So that the first argument is at the top of the stack at the time of the Call

After the procedure call is finished, the caller pops the pushed arguments from the stack

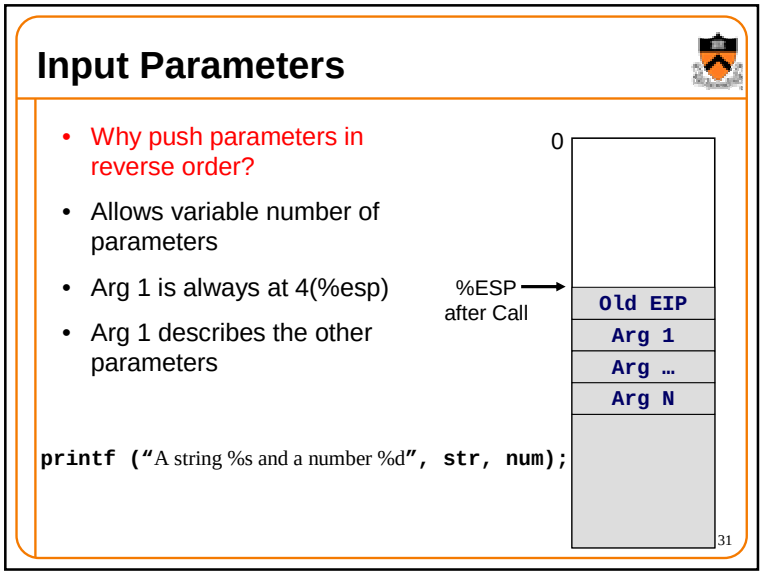
Input Parameters



- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push Nth argument first
  - Push 1st argument last
  - So that the first argument is at the top of the stack at the time of the Call

After the procedure call is finished, the caller pops the pushed arguments from the stack

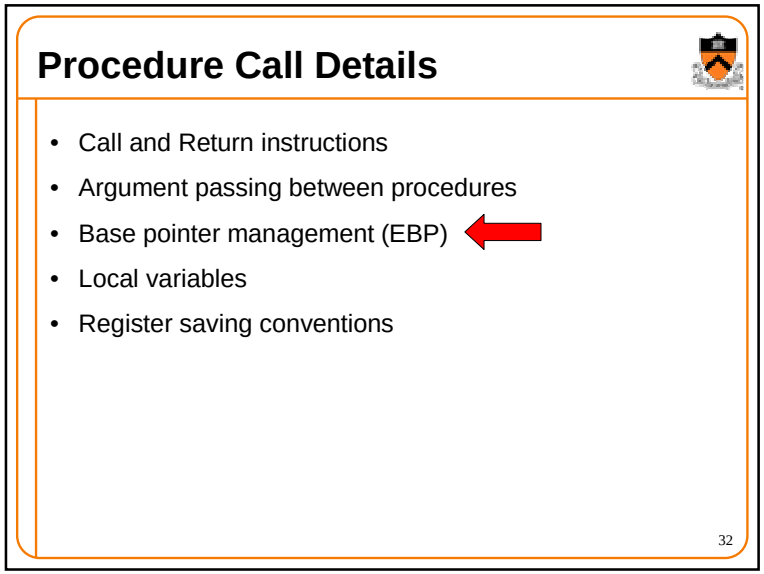
Input Parameters



- Why push parameters in reverse order?
- Allows variable number of parameters
- Arg 1 is always at 4(%esp)
- Arg 1 describes the other parameters

```
printf ("A string %s and a number %d", str, num);
```


Procedure Call Details

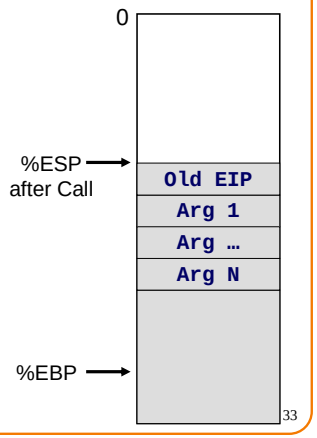


- Call and Return instructions
- Argument passing between procedures
- Base pointer management (EBP)
- Local variables
- Register saving conventions

Base Pointer: EBP

- As Callee executes, ESP may change
- Use EBP as a fixed reference point to access arguments and local variables
- Need to save old value of EBP before using EBP
- Callee begins by executing

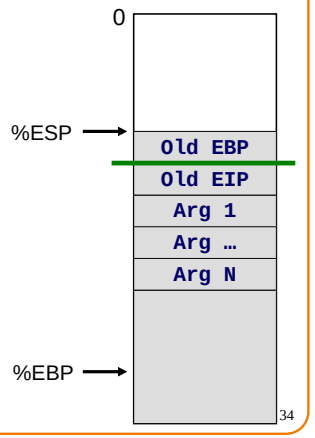
```
pushl %ebp
movl %esp, %ebp
```



Base Pointer: EBP

- As Callee executes, ESP may change
- Use EBP as a fixed reference point to access arguments and local variables
- Need to save old value of EBP before using EBP
- Callee begins by executing

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

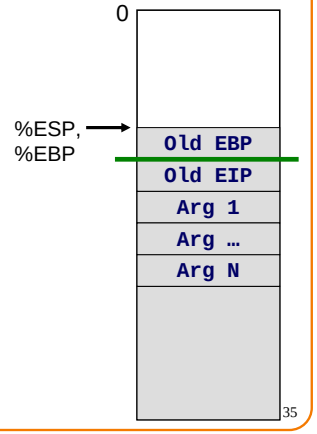


Base Pointer: EBP

- As Callee executes, ESP may change
- Use EBP as a fixed reference point to access arguments and other local variables
- Need to save old value of EBP before using EBP
- Callee begins by executing

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

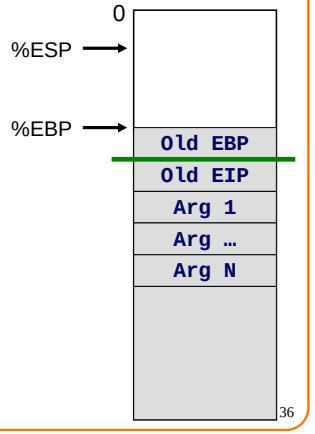
- Regardless of ESP, Callee can address Arg 1 as 8(%ebp)



Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
- Executes

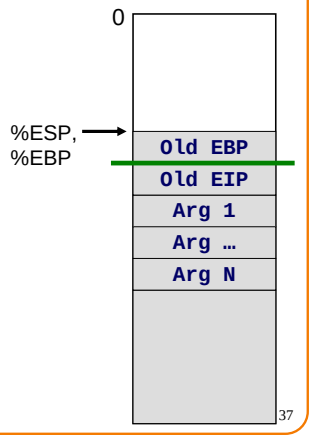
```
movl %ebp, %esp
popl %ebp
ret
```



Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
- Executes

```
movl %ebp, %esp
popl %ebp
ret
```

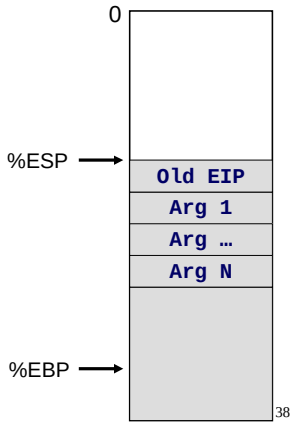


37

Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
- Executes

```
movl %ebp, %esp
popl %ebp
ret
```

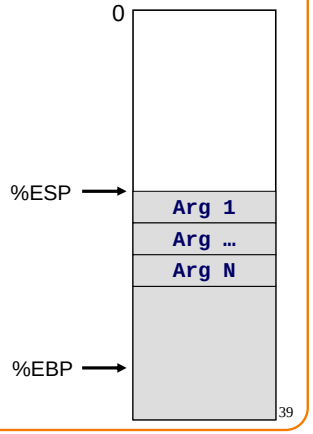


38

Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
- Executes

```
movl %ebp, %esp
popl %ebp
ret
```



39

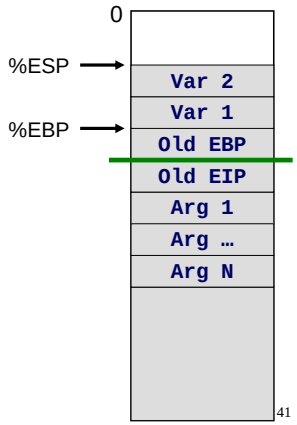
Procedure Call Details

- Call and Return instructions
- Argument passing between procedures
- Base pointer management (EBP)
- Local variables
- Register saving conventions

40

Allocation for Local Variables

- Local variables of the Callee are also allocated on the stack
- Allocation done by moving the stack pointer
- Example: allocate two integers
  - subl $4, %esp
  - subl $4, %esp
  - (or equivalently, subl $8, %esp)
- Reference local variables using the base pointer
  - -4(%ebp)
  - -8(%ebp)



41

Procedure Call Details

- Call and Return instructions
- Argument passing between procedures
- Base pointer management (EBP)
- Local variables
- Register saving conventions

42

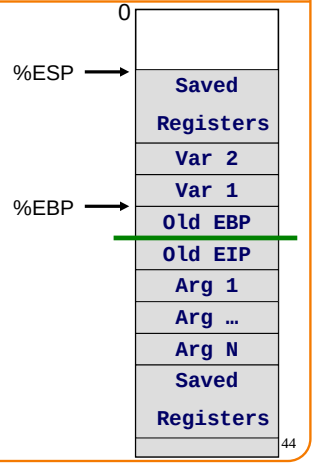
Use of Registers

- Problem: Callee may use a register that the caller is also using
  - When callee returns control to caller, old register contents may be lost. BAD!
  - Someone must save old register contents and later restore
- Need a convention for who saves and restores which registers

43

GCC/Linux Convention

- Caller-save registers
  - **%eax, %edx, %ecx**
  - Save on stack prior to calling
  - Restore after return
- Callee-save registers
  - **%ebx, %esi, %edi**
  - Old values saved on stack prior to using
  - Restore after using
- %esp, %ebp handled as described earlier
- Return value is passed from Callee to Caller in %eax



44

A Simple Example

```
int add3(int a, int b, int c)
{
  int d;

  d = a + b + c;

  return d;
}
```

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


A Simple Example

```
int add3(int a, int b, int c)
{
  int d;

  d = a + b + c;

  return d;
}
```

```
add3:
  # Save old ebp, and set-up
  # new ebp
  pushl %ebp
  movl %esp, %ebp

  # Allocate space for d
  subl $4, $esp

  # In general, one may need
  # to push Callee-save
  # registers onto the stack
```

```
# Add the three arguments
movl 8(%ebp), %eax
addl 12(%ebp), %eax
addl 16(%ebp), eax

# Put the sum into d
movl %eax, -4(%ebp)

# Return value is already
# in eax

# In general, one may need
# to pop Callee-save
# registers

# Restore old ebp and
# discard stack frame
movl %ebp, %esp
popl %ebp

# Return
ret
```


A Simple Example

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

```
foo:
  # Save old ebp, and set-up
  # new ebp
  pushl %ebp
  movl %esp, %ebp

  # No local variables

  # No need to save Callee-
  # save registers as we
  # don't use any registers
```

```
# No need to save Caller-
# save registers either

# Push arguments in reverse
# order
pushl $5
pushl $4
pushl $3

call add3

# Return value is already
# in eax

# Restore old ebp and
# discard stack frame
movl %ebp, %esp
popl %ebp

# Return
ret
```


System Calls

```
.section .data
# pre-initialized variables
# go here

.section .bss
# zero-initialized variables
# go here

.section .rodata
# pre-initialized constants
# go here
```

```
.section .text
.globl _start
_start:
# Program starts executing
# here

# Body of the program goes
# here

# Program ends with an
# "exit()" system call
# to the operating system
movl $1, %eax
int $0x80
```

Summary

- Temporary data related to procedure invocations is stored on the stack
- Stack Frame for a procedure invocation includes return address, procedure arguments, local variables and saved registers
- Call and Ret instructions implement procedure calls
- Base pointer EBP is used as a fixed reference point in the Stack Frame
- Arguments and local variables are addressed relative to the base pointer

49