Assembly Language: Function Calls

Professor Jennifer Rexford

http://www.cs.princeton.edu/~jrex

1

Goals of Today's Lecture

- Challenges of supporting functions
  - Providing information for the called function
    - Function arguments and local variables
  - Allowing the calling function to continue where it left off
    - Return address and contents of registers
- Stack: last-in-first-out data structure
  - Stack frame: args, local vars, return address, registers
  - Stack pointer: pointing to the current top of the stack
- Calling functions
  - Call and ret instructions, to call and return from functions
  - Pushing and popping the stack frame
  - Using the base pointer EBP as a reference point

2

Challenges of Supporting Functions

- Code with a well-defined entry and exit points
  - Call: How does the CPU *go to* that entry point?
  - Return: How does the CPU *go back* to the right place, when "right place" depends on who called the function?
- With arguments and local variables
  - How are the *arguments* passed from the caller?
  - Where should the *local variables* be stored?
- Providing a return value
  - How is the *return value* returned to the calling function?
- Without changing variables in other functions
  - How are the values stored in registers protected?

Call and Return Abstractions

- Call a function
  - Jump to the beginning of an arbitrary procedure
  - I.e., jump to the address of the function's first instruction
- Return from a function
  - Jump to the instruction immediately following the "most-recently-executed" Call instruction
  - But, the same function may be called from many places!

```
P:              # Function P

  …

  jmp R         # Call R

Rtn_point1:

  …
```

```
R:              # Function R

  …

  jmp Rtn_point1   # Return
```

Challenge: Where to Return?

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

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

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

What should the return instruction in R jump to???

5

Store Return Address in Register?

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

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

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

Convention: At Call time, store return address in EAX

6

Problem: Nested Function Calls

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

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

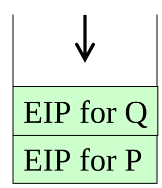
```
Q:            # Function Q
   movl $Rtn_point2, %eax
   jmp R      # Call R
Rtn_point2:
   …
   jmp %eax   # Return
```

- Problem if P calls Q, and Q calls R
- Return address for P to Q call is lost

7

Solution: Put Return Address on a 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 function invocation continues
- 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)
  - Calling function pushes return address on the stack
  - … and called function pops return address off the stack

8

Arguments to the Function

- Calling function needs to pass arguments
  - Cannot simply put arguments in a specific register
  - Because function calls may be nested
- So, put the arguments on the stack, too!
  - Calling function pushes arguments on the stack
  - Called function loads/stores- them on the 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);
}
```

9

Local Variables

- Local variables: called function has 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);
}
```

Registers

- Registers
  - Small, fast memory (e.g., directly on the CPU chip)
  - Used as temporary storage for computations
- Cannot have separate registers per function
  - Could have arbitrary number of nested functions
  - Want to allow each function to use all the registers
- Could write all registers out to memory
  - E.g., save values corresponding to program variables
    - Possible, but a bit of a pain…
  - E.g., find someplace to stash intermediate results
    - Where would we put them?
- Instead, save the registers on the stack, too

11

Stack Frames

- Use stack for all temporary data related to each active function invocation
  - Return address
  - Input parameters
  - Local variables of function
  - Saving registers across invocations

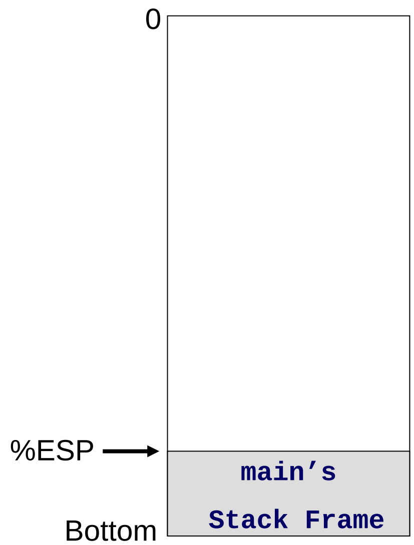
  } **Stack Frame**

- Stack has one Stack Frame per active function invocation

12

High-Level Picture

main begins executing

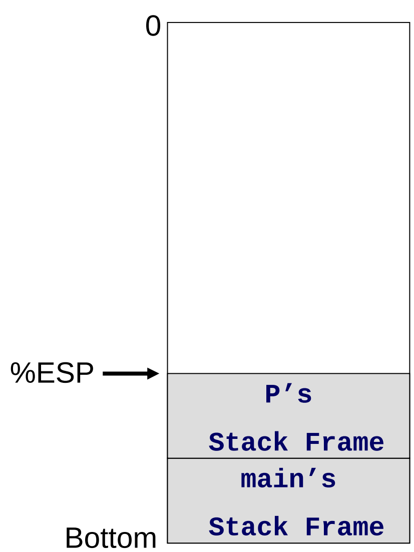


13

High-Level Picture

main begins executing

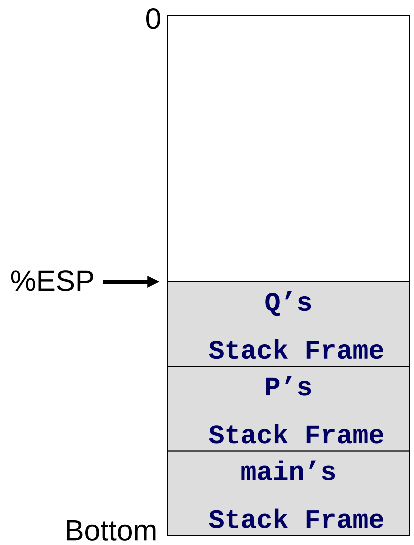
main calls P



14

High-Level Picture

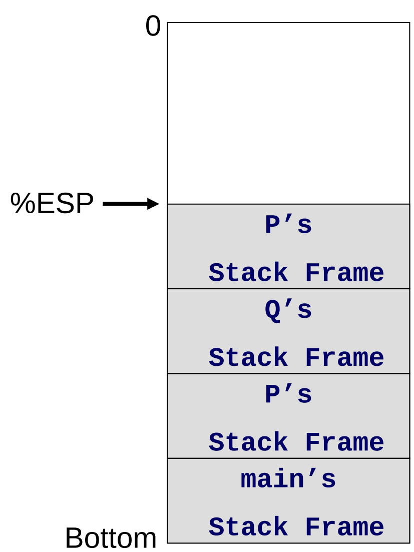
```
main begins executing
main calls P
P calls Q
```



15

High-Level Picture

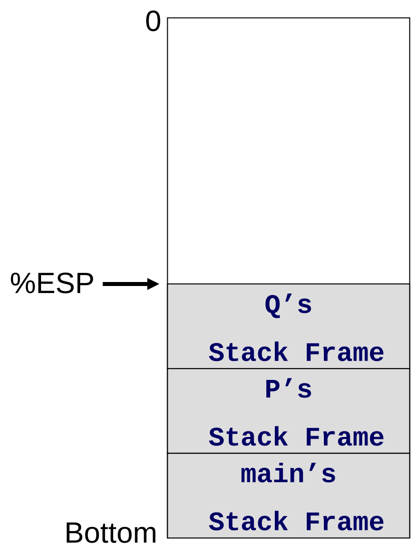
```
main begins executing
main calls P
P calls Q
Q calls P
```



16

High-Level Picture

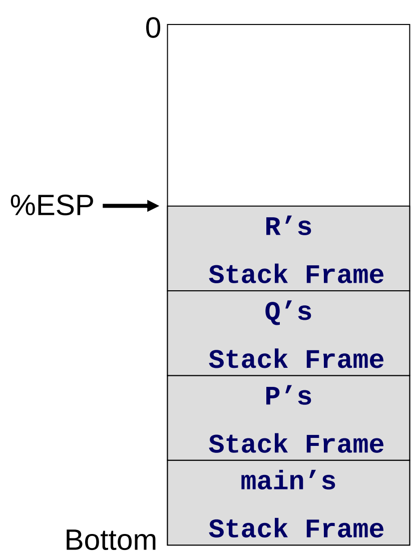
```
main begins executing
main calls P
P calls Q
Q calls P
P returns
```



17

High-Level Picture

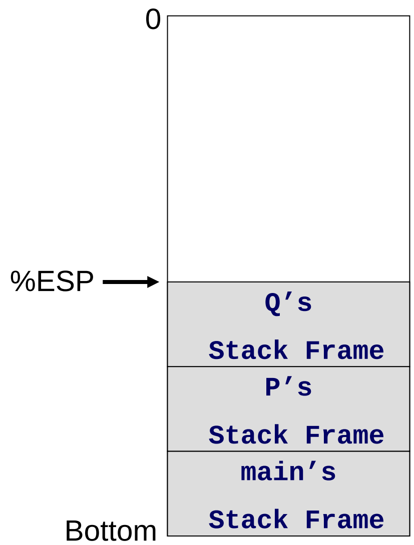
```
main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
```



18

High-Level Picture

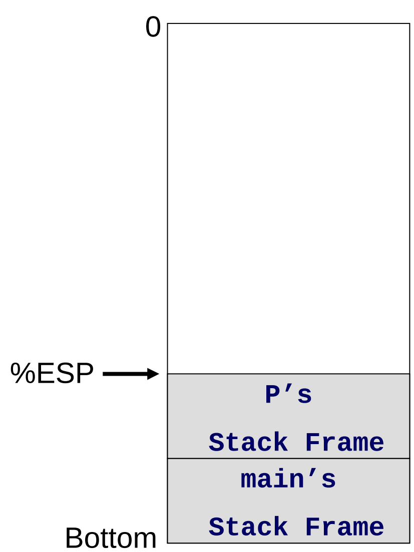
```
main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
R returns
```



19

High-Level Picture

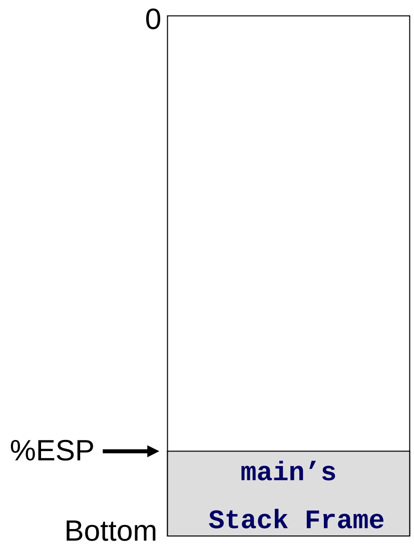
```
main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
R returns
Q returns
```



20

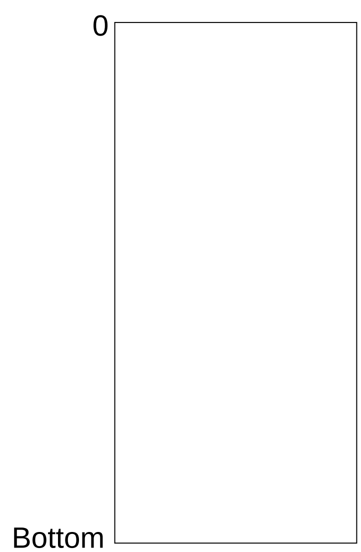
High-Level Picture

```
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
```



Function Call Details

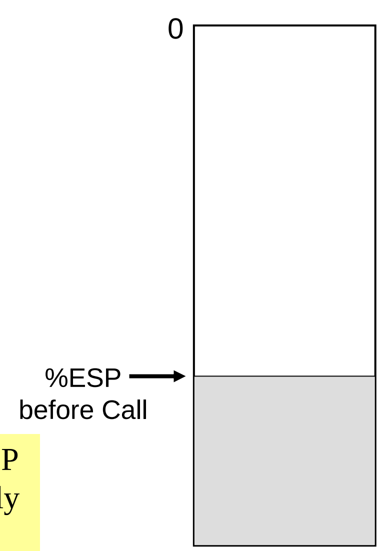
- Call and Return instructions
  - Call: push EIP on the stack, and jump to function
  - Return: pop the stack into the EIP to go back
- Argument passing between procedures
  - Calling function pushes arguments on to the stack
  - Called function reads/writes on the stack
- Local variables
  - Called function creates and manipulates on the stack
- Register saving conventions
  - Either calling or called function saves all of the registers before use

23

Call and Return Instructions

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`	`pop %eip`

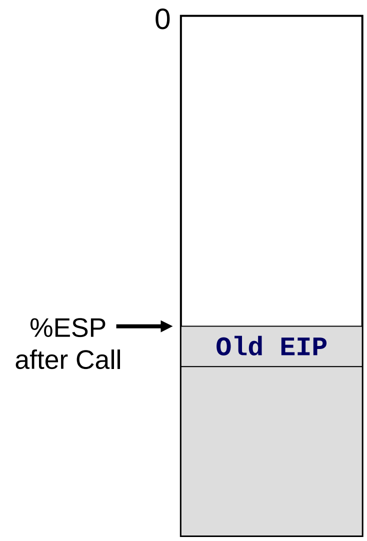
Note: can't really access EIP directly, but this is implicitly what call and ret are doing.



24

Call and Return Instructions

Instruction	Operation
`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`

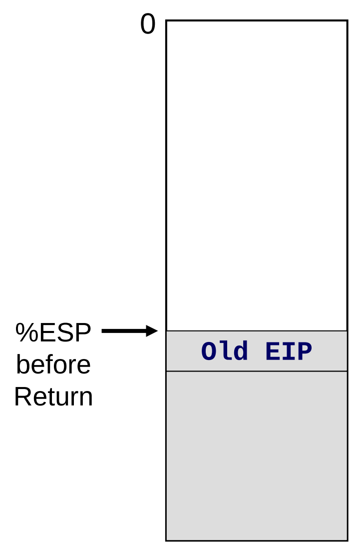


25

Call and Return Instructions

Instruction	Operation
`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

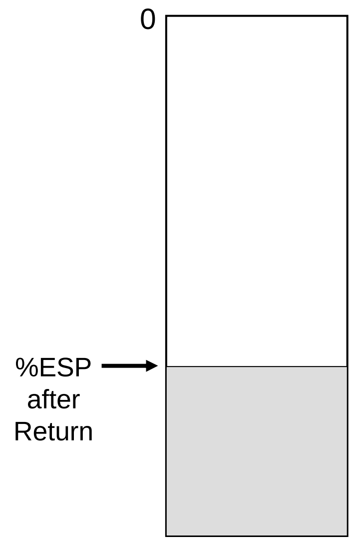


26

Call and Return Instructions

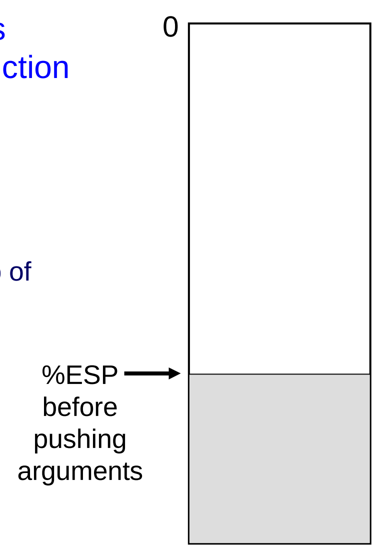
Instruction	Operation
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`
`ret`	`pop %eip`

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



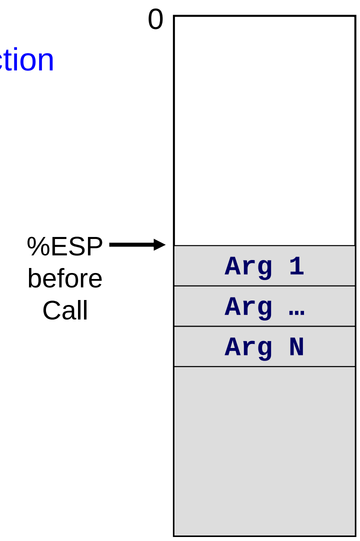
Input Parameters

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



Input Parameters

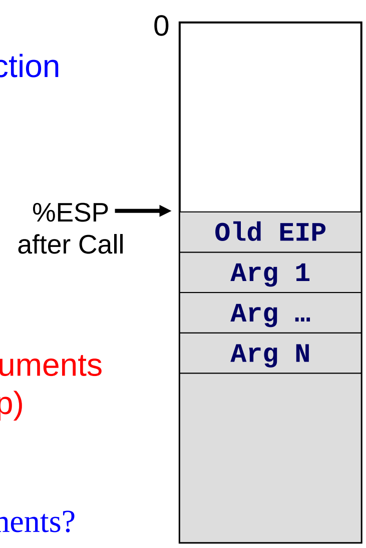
- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push N^{th} argument first
  - Push 1^{st} argument last
  - So that first argument is at top of the stack at the time of the Call



Input Parameters

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

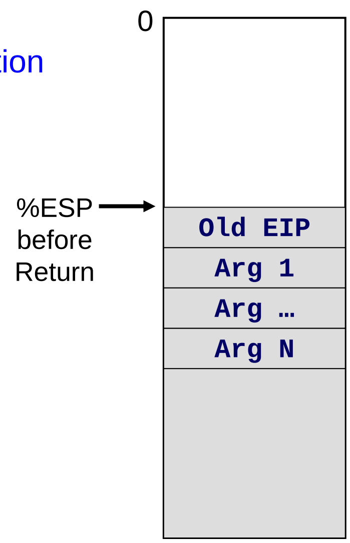
Called function can address arguments relative to ESP: Arg 1 as 4(%esp)



Why is the EIP put on *after* the arguments?

Input Parameters

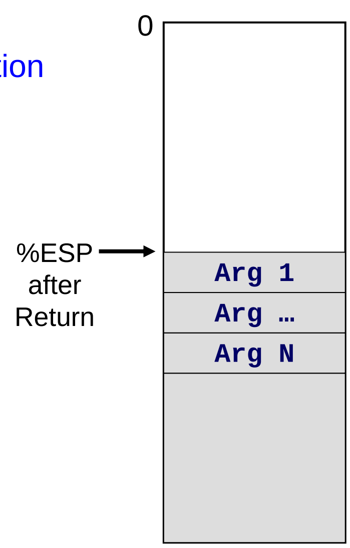
- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push N^{th} argument first
  - Push 1^{st} argument last
  - So that first argument is at top of the stack at the time of the Call



Input Parameters

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

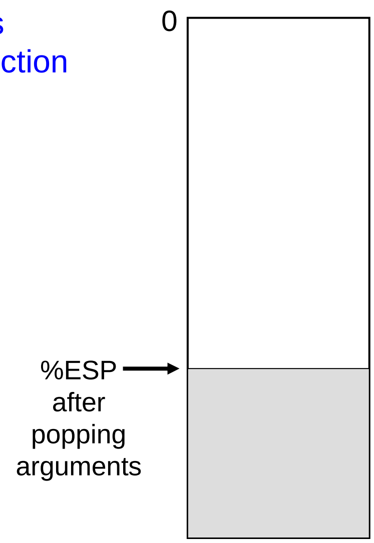
After the function call is finished, the caller pops the pushed arguments from the stack



Input Parameters

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

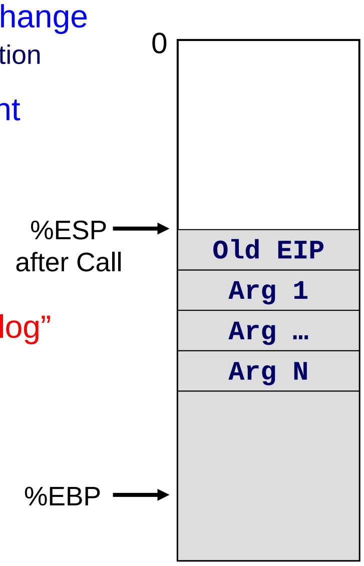
After the function call is finished, the caller pops the pushed arguments from the stack



Base Pointer: EBP

- As Callee executes, ESP may change
  - E.g., preparing to call another function
- Use EBP as fixed reference point
  - E.g., to access arguments and other local variables
- Need to save old value of EBP
  - Before overwriting EBP register
- Callee begins by executing "prolog"

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

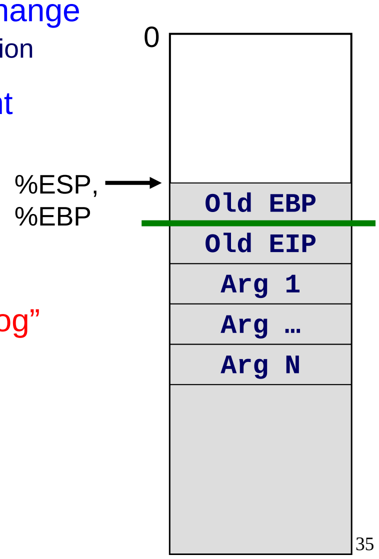


Base Pointer: EBP

- As Callee executes, ESP may change
  - E.g., preparing to call another function
- Use EBP as fixed reference point
  - E.g., to access arguments and other local variables
- Need to save old value of EBP
  - Before overwriting EBP register
- Callee begins by executing "epilog"

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

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

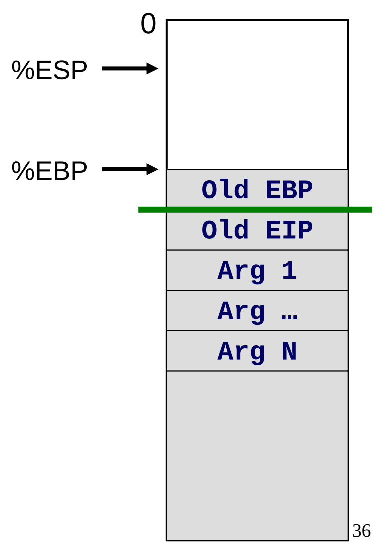


35

Base Pointer: EBP

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

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

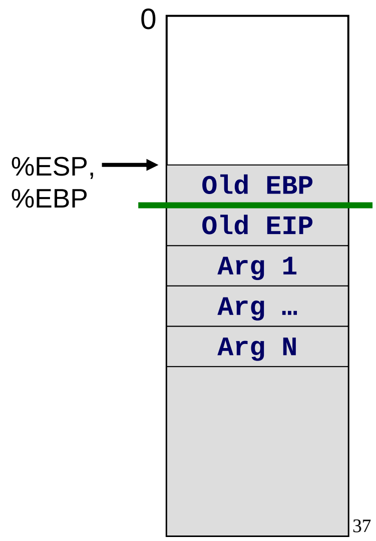


36

Base Pointer: EBP

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

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

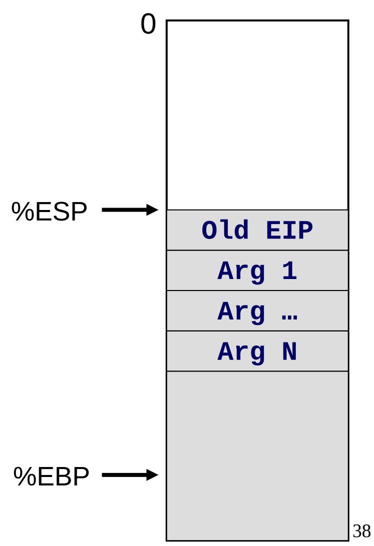


37

Base Pointer: EBP

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

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

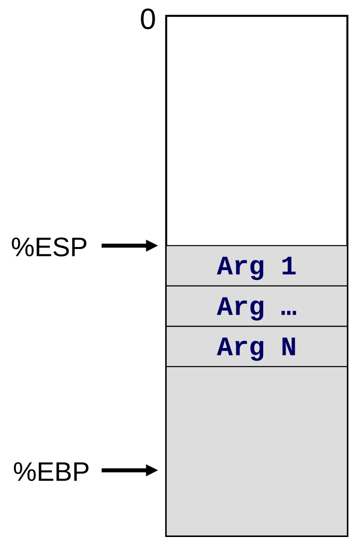


38

Base Pointer: EBP

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

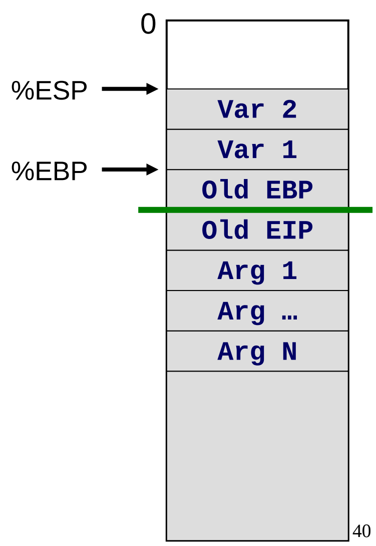
```
movl %ebp, %esp
popl %ebp
ret
```



39

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)



40

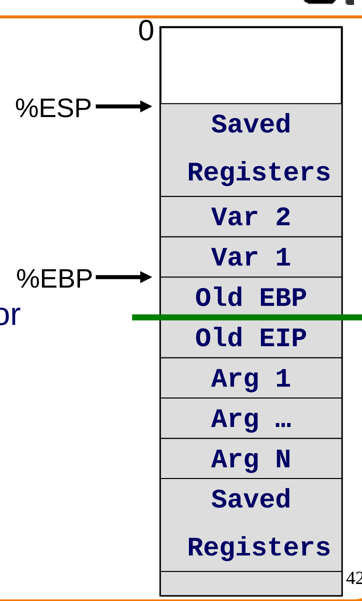
Use of Registers

- Problem: Called function may use a register that the calling function is also using
  - When called function returns control to calling function, old register contents may be lost
  - Calling function cannot continue where it left off
- Solution: save the registers on the stack
  - Someone must save old register contents
  - Someone must later restore the register contents
- Need a convention for who saves and restores which registers

41

GCC/Linux Convention

- Caller-save registers
  - **`%eax, %edx, %ecx`**
  - Save on stack (if necessary) prior to calling
- Callee-save registers
  - **`%ebx, %esi, %edi`**
  - Old values saved on stack prior to using, and restored later
- %esp, %ebp handled as described earlier
- Return value is passed from Callee to Caller in %eax



42

A Simple Example

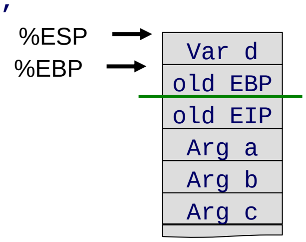
```
int add3(int a, int b, int c)
{
  int d;

  d = a + b + c;

  return d;
}
```

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

43

A Simple Example

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

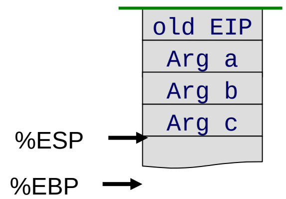



```
add3:
 # Save old ebp and set up new ebp
  pushl %ebp
  movl %esp, %ebp

 # 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, discard stack frame
  movl %ebp, %esp
  popl %ebp

# Return
  ret
```

44

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

# Pop arguments from the stack
 addl $12, %esp

# Return value is already in eax

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

# Return
 ret
```

45

Conclusion

- Invoking a function
  - Call: call the function
  - Ret: return from the instruction
- Stack Frame for a function invocation includes
  - Return address,
  - Procedure arguments,
  - Local variables, and
  - Saved registers
- Base pointer EBP
  - Fixed reference point in the Stack Frame
  - Useful for referencing arguments and local variables

46