Assembly Language: Function Calls
Goals of this Lecture

Help you learn:
• Function call problems
• IA-32 solutions
  • Pertinent instructions and conventions
Function Call Problems

Calling and returning
• How does caller function jump to callee function?
• How does callee function jump back to the right place in caller function?

Passing arguments
• How does caller function pass arguments to callee function?

Storing local variables
• Where does callee function store its local variables?

Returning a value
• How does callee function send return value back to caller function?

Handling registers
• How do caller and callee functions use same registers without interference?
void f(void)
{
    ...  
    n = add3(3, 4, 5);
    ...
}

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

Calling and returning

Passing arguments
Storing local variables
Returning a value
Handling registers
An example
Problem 1: Calling and Returning

How does caller \textit{jump} to callee?
\begin{itemize}
  \item I.e., Jump to the address of the callee’s first instruction
\end{itemize}

How does the callee \textit{jump back} to the right place in caller?
\begin{itemize}
  \item I.e., Jump to the instruction immediately following the most-recently-executed call instruction
\end{itemize}

```c
void f(void)
{
  ... 
  n = add3(3, 4, 5);
  ...
}
```

```c
int add3(int a, int b, int c)
{
  int d;
  d = a + b + c;
  return d;
}
```
Attempted Solution: jmp Instruction

Attempted solution: caller and callee use jmp instruction

f:
...  
jmp g  # Call g
fReturnPoint:
...  

g:
...  
jmp fReturnPoint  # Return
Problem: callee may be called by multiple callers

f1:
    ...
    jmp g      # Call g
f1ReturnPoint:
    ...

f2:
    ...
    jmp g      # Call g
f2ReturnPoint:
    ...

f:
    ...
    jmp ???   # Return

---

Attempted Solution: **jmp** Instruction
Attempted Solution: Use Register

Attempted solution: Store return address in register

```asm
f1:
    movl $f1ReturnPoint, %eax
    jmp g  # Call g
f1ReturnPoint:
    ...

f2:
    movl $f2ReturnPoint, %eax
    jmp g  # Call g
f2ReturnPoint:
    ...
```

```asm
g:
    ...
    jmp *%eax  # Return
```

Special form of `jmp` instruction
Problem: Cannot handle nested function calls

**f:**

```assembly
movl $fReturnPoint, %eax
jmp g  # Call g
```

`fReturnPoint:`

```assembly
...
```

**g:**

```assembly
movl $gReturnPoint, %eax
jmp h  # Call h
```

`gReturnPoint:`

```assembly
...
```

**h:**

```assembly
jmp *%eax  # Return
```

Problem if `f()` calls `g()`, and `g()` calls `h()`

Return address `g() -> f()` is lost
Observations:

- May need to store many return addresses
  - The number of nested function calls 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
- Stored return addresses are destroyed in reverse order of creation
  - f() calls g() => return addr for g is stored
  - g() calls h() => return addr for h is stored
  - h() returns to g() => return addr for h is destroyed
  - g() returns to f() => return addr for g is destroyed
- LIFO data structure (stack) is appropriate

IA-32 solution:

- Use the STACK section of memory
- Via \texttt{call} and \texttt{ret} instructions
call and ret Instructions

ret instruction “knows” the return address

f:
...
call h
...
call g
...

h:
...
ret

1

2

g:
...
call h
...
ret

3

4

5

6
Implementation of `call`

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effective Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl src</td>
<td>subl $4, %esp</td>
</tr>
<tr>
<td></td>
<td>movl src, (%esp)</td>
</tr>
<tr>
<td>popl dest</td>
<td>movl (%esp), dest</td>
</tr>
<tr>
<td></td>
<td>addl $4, %esp</td>
</tr>
</tbody>
</table>
Implementation of **call**

**EIP** (instruction pointer) register points to next instruction to be executed.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effective Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl src</td>
<td>subl $4, %esp</td>
</tr>
<tr>
<td></td>
<td>movl src, (%esp)</td>
</tr>
<tr>
<td>popl dest</td>
<td>movl (%esp), dest</td>
</tr>
<tr>
<td></td>
<td>addl $4, %esp</td>
</tr>
<tr>
<td>call addr</td>
<td>pushl %eip</td>
</tr>
<tr>
<td></td>
<td>jmp addr</td>
</tr>
</tbody>
</table>

**call** instruction pushes return addr (old EIP) onto stack, then jumps.

Note: Can’t really access EIP directly, but this is implicitly what **call** is doing.
# Implementation of \texttt{call}

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effective Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{pushl src}</td>
<td>\texttt{subl $4, %esp}  \texttt{movl src, (%esp)}</td>
</tr>
<tr>
<td>\texttt{popl dest}</td>
<td>\texttt{movl (%esp), dest}       \texttt{addl $4, %esp}</td>
</tr>
<tr>
<td>\texttt{call addr}</td>
<td>\texttt{pushl %eip} \texttt{jmp addr}</td>
</tr>
</tbody>
</table>

ESP after call

Old EIP
Implementation of `ret`

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effective Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl src</td>
<td>subl $4, %esp</td>
</tr>
<tr>
<td></td>
<td>movl src, (%esp)</td>
</tr>
<tr>
<td>popl dest</td>
<td>movl (%esp), dest</td>
</tr>
<tr>
<td></td>
<td>addl $4, %esp</td>
</tr>
<tr>
<td>call addr</td>
<td>pushl %eip</td>
</tr>
<tr>
<td></td>
<td>jmp addr</td>
</tr>
<tr>
<td>ret</td>
<td>popl %eip</td>
</tr>
</tbody>
</table>

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

`ret` instruction pops stack, thus placing return addr (old EIP) into EIP

ESP before `ret`
# Implementation of `ret`

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effective Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pushl src</code></td>
<td><code>subl $4, %esp</code></td>
</tr>
<tr>
<td></td>
<td><code>movl src, (%esp)</code></td>
</tr>
<tr>
<td><code>popl dest</code></td>
<td><code>movl (%esp), dest</code></td>
</tr>
<tr>
<td></td>
<td><code>addl $4, %esp</code></td>
</tr>
<tr>
<td><code>call addr</code></td>
<td><code>pushl %eip</code></td>
</tr>
<tr>
<td></td>
<td><code>jmp addr</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td><code>popl %eip</code></td>
</tr>
</tbody>
</table>

ESP after `ret`
Running Example

Caller

f:
...
  # Call the function
  call add3
  ...

Callee

add3:
  ...
  ret
Agenda

Calling and returning

**Passing arguments**

Storing local variables

Returning a value

Handling registers

An example
Problem 2: Passing Arguments

Problem: How does caller pass arguments to callee?

```c
void f(void)
{   ...
    n = add3(3, 4, 5);
    ...
}
```

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

- May need to store many arg sets
  - The number of arg sets is not known in advance
  - Arg set must be saved for as long as the invocation of this function is live, and discarded thereafter
- Stored arg sets are destroyed in reverse order of creation
  - f() calls g() => arg set for g is created
  - g() calls h() => arg set for h is created
  - h() returns to g() => arg set for h is destroyed
  - g() returns to f() => arg set for g is destroyed
- LIFO data structure (stack) is appropriate

IA 32 solution:

- Use the STACK section of memory
Passing Args on the Stack

Before executing `call` instruction…
Passing Args on the Stack

Caller pushes args in reverse order

- Push Nth arg first
- Push 1st arg last
- So 1st arg is at top of the stack at the time of the call

```
0
ESP
```

```
Arg1
Arg...
ArgN
```
Passing Args on the Stack

Caller executes \texttt{call} instruction
Passing Args on the Stack

Callee accesses args relative to ESP

Arg1 as 4(esp)
Arg2 as 8(esp)
...

ESP →
Old EIP
Arg1
Arg...
ArgN
Passing Args on the Stack

Callee executes \texttt{ret} instruction
Passing Args on the Stack

Caller pops args from the stack

ESP
Running Example

f:

...# Push arguments
pushl $5
pushl $4
pushl $3

# Call the function
call add3

# Pop arguments
addl $12, %esp
...

add3:

...# Use arguments
movl 4(%esp), %eax
addl 8(%esp), %eax
addl 12(%esp), %eax
...
ret
Base Pointer Register: EBP

Problem:
- As callee executes, ESP may change
  - E.g., preparing to call another function
  - Error-prone for callee to reference args as offsets relative to ESP

Solution:
- Use **EBP** (base pointer) register
  - EBP doesn’t change during callee’s execution
  - Use EBP as fixed reference point to access args

```
Old EIP
Arg1
Arg...
ArgN
```

```
ESP
EBP
```
Before executing `call` instruction…
Caller pushes args in reverse order

- Push Nth arg first
- Push 1st arg last
- So 1st arg is at top of the stack at the time of the `call`
Passing Args on the Stack (v2)

Caller executes `call` instruction

```
ESP -> Old EIP
    Arg1
    Arg...
    ArgN
```

EBP
Passing Args on the Stack (v2)

Need to save old value of EBP

- Before overwriting EBP register

Callee executes “prolog”

\[
\text{pushl } \%ebp
\]

\[
\text{movl } \%esp, \%ebp
\]

ESP →

0

Old EBP

Old EIP

Arg1

Arg...

ArgN

EBP →
Callee executes “prolog”

\[
\begin{align*}
\text{pushl } & \%ebp \\
\text{movl } & \%esp, \%ebp
\end{align*}
\]

Regardless of ESP, callee can reference Arg1 as \(8(\%ebp)\), Arg2 as \(12(\%ebp)\), etc.
Passing Args on the Stack (v2)

Before returning, callee must restore ESP and EBP to their old values.

Callee executes “epilog”

\[
\text{movl } \%ebp, \%esp \\
\text{popl } \%ebp
\]

ESP

EBP

Old EBP

Old EIP

Arg1

Arg...

ArgN
Callee executes “epilog”

\[
\begin{align*}
\text{movl} & \ %ebp, \ %esp \\
\text{popl} & \ %ebp
\end{align*}
\]
Callee executes “epilog”

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

ESP →

0

Old EIP
Arg1
Arg...
ArgN

EBP →
Passing Args on the Stack (v2)

Callee executes `ret` instruction

```
ESP  Arg1  Arg...  ArgN  0
  ↓    ↓      ↓      ↓  ↓
  EBP  ↓      ↓      ↓  ↓
```
Passing Args on the Stack (v2)

Caller pops args from the stack
f:
...  
# Push arguments
pushl $5
pushl $4
pushl $3

# Call the function
call add3

# Pop arguments
addl $12, %esp
...

add3:
pushl %ebp
movl %esp, %ebp

... 
# Use arguments
movl 8(%ebp), %eax
addl 12(%ebp), %eax
addl 16(%ebp), %eax

... 
movl %ebp, %esp
popl %ebp
ret
Agenda

Calling and returning
Passing arguments

**Storing local variables**
Returning a value
Handling registers
An example
Problem 3: Storing Local Variables

Where does callee function store its *local variables*?

```c
void f(void)
{
    n = add3(3, 4, 5);
    ...
}

int add3(int a, int b, int c)
{
    int d;
    d = a + b + c;
    return d;
}
```
Observations (déjà vu again!):

- May need to store many local var sets
  - The number of local var sets is not known in advance
  - Local var set must be saved for as long as the invocation of this function is live, and discarded thereafter
- Stored local var sets are destroyed in reverse order of creation
  - f() calls g() => local vars set for g is created
  - g() calls h() => local vars set for h is created
  - h() returns to g() => local vars set for h is destroyed
  - g() returns to f() => local vars set for g is destroyed
- LIFO data structure (stack) is appropriate

**IA 32 solution:**

- Use the STACK section of memory
Callee allocates space for its local variables on the stack
  • Via `pushl` instructions
  • Via `subl $4,%esp` instructions

Example: allocate memory for two integers
  • `subl $4,%esp` # int i;
  • `pushl $5` # int j = 5;

Callee references local variables as negative offsets relative to EBP
  • `-4(%ebp)` # Access i
  • `-8(%ebp)` # Access j
Running Example

f:

```assembly
... # Push arguments
pushl $5
pushl $4
pushl $3

# Call the function
call add3

# Pop arguments
addl $12, %esp
...
```

add3:

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

# Allocate mem for local var
subl $4, %esp

# Use arguments
movl 8(%ebp), %eax
addl 12(%ebp), %eax
addl 16(%ebp), %eax

# Use local variable
movl %eax, -4(%ebp)
...
movl %ebp, %esp
popl %ebp
ret```

...
Agenda

Calling and returning
Passing arguments
Storing local variables
**Returning a value**
Handling registers
An example
Problem 4: Return Values

Problem: How does callee function send return value back to caller function?

```c
void f(void)
{   ...
    n = add3(3, 4, 5);
    ...
}

int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```
IA-32 Solution: Use EAX

In principle
• Store return value in stack frame of caller

Or, for efficiency
• Known small size => store return value in register
• Other => store return value in stack

IA-32 convention
• Integer or pointer:
  • Store return value in EAX
• Floating-point number:
  • Store return value in floating-point register
  • (Beyond scope of COS 217)
• Structure:
  • Store return value on stack
  • (Beyond scope of COS 217)
Running Example

f:

...  
# Push arguments
pushl $5
pushl $4
pushl $3

# Call the function
call add3

# Pop arguments
addl $12, %esp

# Use return value
movl %eax, n
...

add3:

pushl %ebp
movl %esp, %ebp

# Allocate mem for local var
subl $4, %esp

# Use arguments
movl 8(%ebp), %eax
addl 12(%ebp), %eax
addl 16(%ebp), %eax

# Use local variable
movl %eax, -4(%ebp)

# Indicate return value
movl -4(%ebp), %eax

movl %ebp, %esp
popl %ebp
ret
Agenda

- Calling and returning
- Passing arguments
- Storing local variables
- Returning a value
- Handling registers
- An example
Problem 5: Handling Registers

Observation: Registers are a finite resource
- In principle: Each function should have its own registers
- In reality: All functions share same small set of registers

Problem: How do caller and callee use same set of registers without interference?
- 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
Callee-save registers: EBX, ESI, EDI

- If necessary…
  - Callee saves to stack after prolog
  - Callee restores from stack before epilog
- Caller can assume that values in EBX, ESI, EDI will not be changed by callee

Caller-save registers: EAX, ECX, EDX

- If necessary…
  - Caller saves to stack before call
  - Caller restores from stack after call
Running Example

f:
...
# Save EAX, ECX, EDX
pushl %eax
pushl %ecx
pushl %edx

# Push arguments
pushl $5
pushl $4
pushl $3

# Call the function
call add3

# Pop arguments
addl $12, %esp

# Use return value
movl %eax, n

# Restore EAX, ECX, EDX
popl %edx
popl %ecx
popl %eax
...

...
Running Example

add3:
    pushl %ebp
    movl %esp, %ebp

    # Save EBX, ESI, EDI
    pushl %ebx
    pushl %esi
    pushl %edi

    # Allocate mem for local var
    subl $4, %esp

    # Use arguments
    movl 8(%ebp), %eax
    addl 12(%ebp), %eax
    addl 16(%ebp), %eax

    # Use local variable
    movl %eax, -16(%ebp)

    # Indicate return value
    movl -4(%ebp), %eax

    # Restore EBX, ESI, EDI
    movl -12(%ebp), %edi
    movl -8(%ebp), %esi
    movl -4(%ebp), %ebx

    movl %ebp, %esp
    popl %ebp
    ret

Not necessary to save callee-save registers in this particular function
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:

- Values of caller-save registers
- Arguments to be passed to callee function
- Return address (old EIP)
- Old EBP
- Values of callee-save registers
- Local variables
Agenda

- Calling and returning
- Passing arguments
- Storing local variables
- Returning a value
- Handling registers

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

void f(void)
{
    ... 
    n = add3(3, 4, 5);
    ...
}
Trace of Running Example 1

n = add3(3, 4, 5);

Low memory

High memory
Trace of Running Example 2

```c
n = add3(3, 4, 5);
```

# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx

![Stack diagram](image)
Trace of Running Example 3

n = add3(3, 4, 5);

# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push arguments
pushl $5
pushl $4
pushl $3

High memory

Low memory
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx

# Push arguments
pushl $5
pushl $4
pushl $3

# Call add3
call add3

n = add3(3, 4, 5);
int add3(int a, int b, int c)
{ int d;
  d = a + b + c;
  return d;
}
int add3(int a, int b, int c)
{
    int d;
    d = a + b + c;
    return d;
}

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

Prolog

Low memory

High memory

ESP
Old EBP
Old EIP
Old ECX
Old EDX
Old EAX

EBP
Old EAX
Old ECX
Old EDX
Old EIP
int add3(int a, int b, int c) {
    int d;
    d = a + b + c;
    return d;
}

# 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

<table>
<thead>
<tr>
<th>Low memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBP</td>
</tr>
<tr>
<td>Old EBP</td>
</tr>
<tr>
<td>Old EDI</td>
</tr>
<tr>
<td>Old ESI</td>
</tr>
<tr>
<td>Old EBX</td>
</tr>
<tr>
<td>Old EIP</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>Old EDX</td>
</tr>
<tr>
<td>Old ECX</td>
</tr>
<tr>
<td>Old EAX</td>
</tr>
</tbody>
</table>

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

# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save caller-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
# Allocate space for local variable
subl $4, %esp
# Trace of Running Example 9

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

# Save old EBP
pushl %ebp

# Change EBP
movl %esp, %ebp

# Save caller-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 args as positive offsets relative to EBP

Access local vars as neg offsets relative to EBP

High memory

Low memory
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
```c
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

```
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

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
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
n = add3(3, 4, 5);

# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push arguments
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop arguments
addl $12, %esp
n = add3(3, 4, 5);

# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push arguments
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop arguments
addl %12, %esp
# Use return value
movl %eax, n
n = add3(3, 4, 5);

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

n = add3(3, 4, 5);
Function calls in IA-32 assembly language

Calling and returning
- **call** instruction: push EIP onto stack and jump
- **ret** instruction: pop from stack to EIP

Passing arguments
- Caller pushes onto stack
- Callee accesses as positive offsets from EBP
- Caller pops from stack
Summary (cont.)

Storing local variables
• Callee pushes onto 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 places data of integer types and addresses in EAX