Assembly Language: Function Calls

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
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?

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
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

• 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: ...
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: ...
```

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

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

Special form of jmp instruction; we will not use

- Problem: Cannot handle nested function calls

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

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

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

Problem if P calls Q, and Q calls R

Return address for P to Q call is lost
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

IA-32 Call and Ret Instructions

- Ret instruction “knows” the return address

```
P:    # Function P
     ...
     call R
     call Q
     ...

Q:    # Function Q
     ...
     call R
     ...
     ret

R:    # Function R
     ...
     ret
```
IA-32 Call and Ret Instructions

- Ret instruction “knows” the return address

```
P:     # Function P
       ...
       call R
       call Q
       ...

Q:     # Function Q
       ...
       call R
       ...

R:     # Function R
       ...
       ret
```

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>
</tbody>
</table>

```

ESP

0

ESP
```
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>
</tbody>
</table>

ESP

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

• 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>

ESP

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>

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

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

ESP before call
Implementation of Call

<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>

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 address (old EIP) into 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>

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?
Problem 2: Passing Parameters

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

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

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

Attempted Solution: Use Registers

- Attempted solution: Pass parameters in registers

```assembly
f:
   movl $3, %eax
   movl $4, %ebx
   movl $5, %ecx
   call add3
   ...
add3:
   ...
   # Use EAX, EBX, ECX
   ...
   ret
```
Attempted Solution: Use Registers

- Problem: Cannot handle nested function calls

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

```asm
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?

IA-32 Solution: Use the Stack

- Caller pushes parameters before executing the call instruction

```
ESP before pushing params
```

```asm
0
```
IA-32 Parameter Passing

- Caller pushes parameters in the reverse order
  - Push \( N^{th} \) param first
  - Push 1\(^{st} \) param last
  - So first param is at top of the stack at the time of the Call

![Diagram showing ESP before call with parameters Param 1, ..., Param N]

IA-32 Parameter Passing

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

![Diagram showing ESP after call with Old EIP and parameters Param 1, ..., Param N]
IA-32 Parameter Passing

• After returning to the caller…

Esp after return

Param 1
Param ...
Param N

IA-32 Parameter Passing

• … the caller pops the parameters from the stack

Esp after popping params
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
```

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
Using EBP

• Need to save old value of EBP
  • Before overwriting EBP register

• Callee executes "prolog"
  pushl %ebp
  movl %esp, %ebp

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.
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 it’s “pre-callee” value, which was stored in %EBP

Base Pointer Register: EBP

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

  %EBP needs to be restored to it’s “pre-callee” value, which was stored on the stack.
• Callee executes “epilog”
  
  `movl %ebp, %esp`
  `popl %ebp`
  `ret`

And we’re back to where we were before the call
Problem 3: Storing Local Variables

• Where does callee function store its local variables?

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

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

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

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

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

For example:

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

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
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

Problem 5: Return Values

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

- **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

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

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

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

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

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
A Simple Example

```c
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);
...
```

Trace of a Simple Example 1

```c
x = add3(3, 4, 5);
```
Trace of a Simple Example 2

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

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

```
High memory
```

Trace of a Simple Example 3

```c
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

```
High memory
```
Trace of a Simple Example 4

```c
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
```

Trace of a Simple Example 5

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

```
# Save old EBP
pushl %ebp

{ Prolog
```

```
ESP ── Old EIP
    3
    4
    5
    Old EDX
    Old ECX
    Old EAX
```

```
EBP ── High memory
```

```
ESP ── Old EBP
    Old EIP
    Old EDX
    Old ECX
    Old EAX
```

```
EBP ── High memory
```
Trace of a Simple Example 6

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

Trace of a Simple Example 7

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
Unnecessary here; add3 will not
change the values in these registers
Trace of a Simple Example 8

```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

Trace of a Simple 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 params as positive offsets relative to EBP
Access local vars as negative offsets relative to 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

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
# Return to calling function
ret
Trace of a Simple Example 14

\[ x = \text{add3}(3, 4, 5); \]

# 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

\[ x = \text{add3}(3, 4, 5); \]

# 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

```c
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

```c
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

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