Assembly Language: Function Calls

Jennifer Rexford
Goals of this Lecture

• 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
Recall from Last Lecture

Examples of Operands

• Immediate Operand
  • movl $5, ...
    • CPU uses 5 as source operand
  • movl $i, ...
    • CPU uses address denoted by i as source operand

• Register Operand
  • movl %eax, ...
    • CPU uses contents of EAX register as source operand
Recall from Last Lecture (cont.)

- **Memory Operand: Direct Addressing**
  - movl i, ...
  - CPU fetches source operand from memory at address i

- **Memory Operand: Indirect Addressing**
  - movl (%eax), ...
  - CPU considers contents of EAX to be an address
  - Fetches source operand from memory at that address

- **Memory Operand: Base+Displacement Addressing**
  - movl 8(%eax), ...
  - CPU computes address as 8 + [contents of EAX]
  - Fetches source operand from memory at that address
Recall from Last Lecture (cont.)

• Memory Operand: Indexed Addressing
  • movl 8(%eax, %ecx), …
    • Computes address as 8 + [contents of EAX] + [contents of ECX]
    • Fetches source operand from memory at that address

• Memory Operand: Scaled Indexed Addressing
  • movl 8(%eax, %ecx, 4), …
    • Computes address as 8 + [contents of EAX] + ([contents of ECX] * 4)
    • Fetches source operand from memory at that address

• Same for destination operand, except…

• Destination operand cannot be immediate
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 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:
   ...

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

R: # Function R
   ...
   jmp ??? # Return
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
Attempted Solution: Use Register

- 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
• 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, 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: Use the stack via call and 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
• Ret instruction “knows” the return address
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>

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

Call instruction pushes return address (old EIP) onto stack
### 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>

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>pop %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>pop %eip</td>
</tr>
</tbody>
</table>

ESP after ret
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

```
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

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

```assembly
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?
• Caller pushes parameters before executing the call instruction
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
IA-32 Parameter Passing

• Callee addresses params relative to ESP: Param 1 as 4(%esp)
IA-32 Parameter Passing

- After returning to the caller…
IA-32 Parameter Passing

- ... the caller pops the parameters from the stack
IA-32 Parameter Passing

For example:

```plaintext
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

- **Problem:**
  - 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 EBP as fixed reference point to access params
Using EBP

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

• Callee executes “prolog”

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

ESP

0

Old EBP

Old EIP

Param 1

Param ...

Param N

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
  ```

  ![Diagram showing ESP and EBP registers with Stack frame](diagram.png)
Base Pointer Register: EBP

- Callee executes “epilog”
  ```
  movl %ebp, %esp
  popl %ebp
  ret
  ```
Base Pointer Register: EBP

- Callee executes “epilog”
  
  ```
  movl %ebp, %esp
  popl %ebp
  ret
  ```
Base Pointer Register: EBP

• Callee executes “epilog”

  movl %ebp, %esp
  popl %ebp
  ret

ESP
  Param 1
  Param ...
  Param N

EBP
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);
}
```
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;
}
```
```c
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)`
IA-32 Local Variables

For example:

```
add3:
  ...
  # Allocate space for d
  subl $4, %esp
  ...
  # Access 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 set of registers
  • In reality: All functions must use the same small set of registers

• Callee may use a register that the caller also is using
  • When callee returns control to caller, old register contents may be lost
  • Caller function cannot continue where it left off
• IA-32 solution: save the registers on the stack
  • Someone must save old register contents
  • Someone must later restore the register contents

• Define a convention for who 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 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 course)

- Structure:
  - Store return value on stack
  - (Beyond scope of course)

```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);
```

Low memory

High memory
Trace of a Simple Example 2

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

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

![Diagram showing memory layout with ESP and EBP pointers to Old EAX, Old ECX, and Old EDX in high memory.](Image)
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

Low memory

3
4
5
Old EDX
Old ECX
Old EAX
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

High memory

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

# Save old EBP
pushl %ebp

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

<table>
<thead>
<tr>
<th>ESP</th>
<th>Old EBP</th>
<th>Old EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Old EDX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Old ECX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Old EAX</td>
<td></td>
</tr>
</tbody>
</table>

52
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
    } Unnecessary here; add3 will not change the values in these registers

ESPCentESP

EBP Old EBP
Old EBX
Old EIP
3
4
5
Old EDX
Old ECX
Old EAX
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
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

<table>
<thead>
<tr>
<th>ESP</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old EDI</td>
<td></td>
</tr>
<tr>
<td>Old ESI</td>
<td></td>
</tr>
<tr>
<td>Old EBX</td>
<td></td>
</tr>
<tr>
<td>Old EBP</td>
<td></td>
</tr>
<tr>
<td>Old EIP</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Old EDX</td>
<td></td>
</tr>
<tr>
<td>Old ECX</td>
<td></td>
</tr>
<tr>
<td>Old EAX</td>
<td></td>
</tr>
</tbody>
</table>

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
# Restore ESP
movl %ebp, %esp

Epilog
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;
}

# 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

Low memory

High memory

Epilog
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

```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
```

```
12
Old EBP
Old EBP
Old EBP
Old EBP

5
4
3

12
Old EDX
Old ECX
Old EAX
```

Low memory

High 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

# Pop parameters
addl %12, %esp

# Save return value
movl %eax, wherever
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