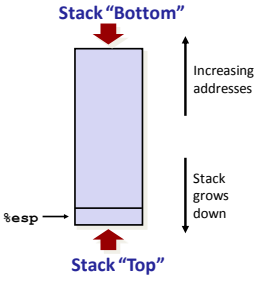
Lecture 14

Procedures

CPSC 275
Introduction to Computer Systems

IA32 Stack

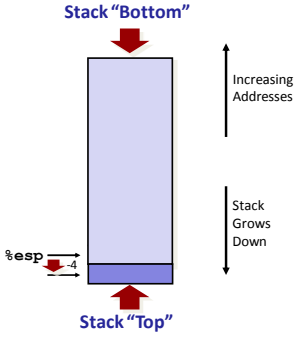
- Grows toward lower addresses
- Register `%esp` contains lowest stack address
  - address of "top" element
- Two operations on stack:
  - **push**: adding a new item on the stack
  - **pop**: removing the top element from the stack



IA32 Stack: **push**

`push src`

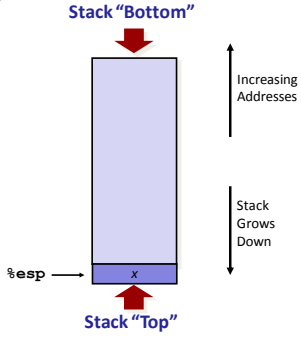
- Fetch operand at *src*
- Decrement `%esp` by 4



IA32 Stack: **push**, cont'd

`push src`

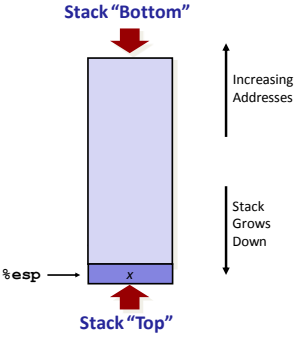
- Fetch operand at *src*
- Decrement `%esp` by 4
- Write operand at address given by `%esp`



IA32 Stack: **pop**

`pop dest`

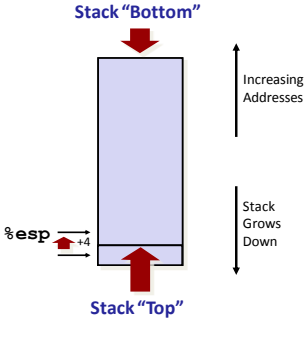
- Copy the stack top item to **dest**.



IA32 Stack: **pop**, cont'd

`pop dest`

- Copy the stack top item to **dest**.
- Increment `%esp` by 4



Procedure Control Flow

- Use stack to support procedure (function) calls and return
- Procedure call: **`call label`**
  - Push return address on stack
  - Jump to ***label***
- Return address: address of the *next* instruction right after call
- Procedure return: **`ret`**
  - Pop return address from stack
  - Jump to return address

Compiling Into Assembly

C Code `code.c`

```
int sum(int x, int y)
{
  int t = x + y;
  return t;
}
```

Generate with command

```
$ gcc -O1 -S code.c
```

produces file `code.s`

IA32 Assembly `code.s`

```
sum:
   pushl %ebp
   movl %esp,%ebp
   movl 12(%ebp),%eax
   addl 8(%ebp),%eax
   popl %ebp
   ret
```

But what does its *object code* look like?

Compiling Into Object Code

C Code `code.c`

```
int sum(int x, int y)
{
  int t = x + y;
  return t;
}
```

Generate object code with command

```
$ gcc -O1 -c code.c
```

produces file `code.o`

Object Code `code.o`

```
080483c4 <sum>:
   0x55
   0x89
   0xe5
   0x8b
   0x45
   0x0c
   0x03
   0x45
   0x08
   0x5d
   0xc3
```

Object Code

`code.o`

```
080483c4 <sum>:
   0x55
   0x89
   0xe5
   0x8b
   0x45
   0x0c
   0x03
   0x45
   0x08
   0x5d
   0xc3
```

- Total of 11 bytes
- Starts at address 0x080483c4

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for **`malloc`**, **`printf`**
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

Disassembling Object Code

`code.o`

```
080483c4 <sum>:
   0x55
   0x89
   0xe5
   0x8b
   0x45
   0x0c
   0x03
   0x45
   0x08
   0x5d
   0xc3
```

- Total of 11 bytes
- Starts at address 0x080483c4
- Each instruction 1, 2, or 3 bytes

But what does it mean?

Disassemble object code with command

```
$ objdump -d code.o
```

produces object code using the following format:

```
080483c4 <sum>:
 80483c4:   55          push   %ebp
 80483c5:   89 e5       mov    %esp,%ebp
 80483c7:   8b 45 0c    mov    0xc(%ebp),%eax
 80483ca:   03 45 08    add    0x8(%ebp),%eax
 80483cd:   5d          pop    %ebp
 80483ce:   c3          ret
```

Machine Instruction Example

```
int t = x + y;
```

```
addl 8(%ebp),%eax
```

Similar to expression:

```
y += x
```

```
0x80483ca:   03 45 08
```

- C Code
  - Add two signed integers
- Assembly
  - Add two 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:
    - **`y`**: Register **`%eax`**
    - **`x`**: Memory **`M[%ebp+8]`**
    - **`t`**: Register **`%eax`**
      - Return function value in **`%eax`**
- Object Code
  - 3-byte instruction
  - Stored at address **`0x80483ca`**

What Can be Disassembled?

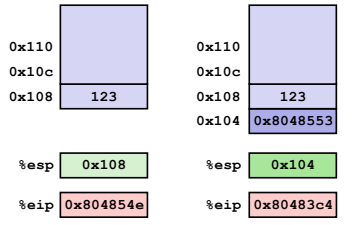
```
$ objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55              push   %ebp
30001001:  8b ec           mov    %esp,%ebp
30001003:  6a ff           push   $0xffffffff
30001005:  68 90 10 00 30  push   $0x30001090
3000100a:  68 91 dc 4c 30  push   $0x304cdc91
```

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Procedure Call Example

```
804854e:    e8 3d 06 00 00      call   80483c4 <sum>
8048553:    50                  pushl  %eax
```

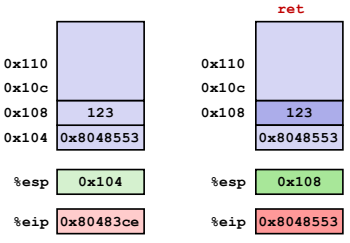
call sum at 0x80483c4



%eip: program counter

Procedure Return Example

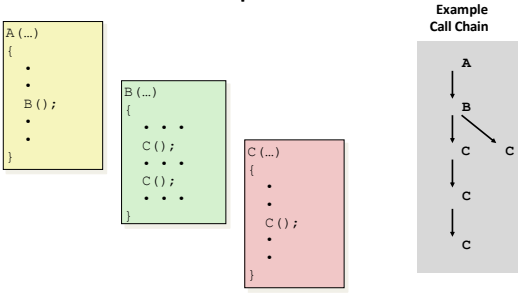
```
80483ce:    c3                  ret
```



Stack-Based Languages

- C, Pascal, Java, etc.
- Need some place to store state of each instantiation (or *activation*)
  - Arguments
  - Local variables
  - Return pointer
- Stack allocated in *frames*.
  - state for single procedure instantiation

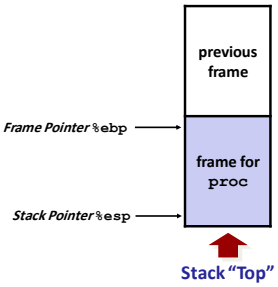
Call Chain Example

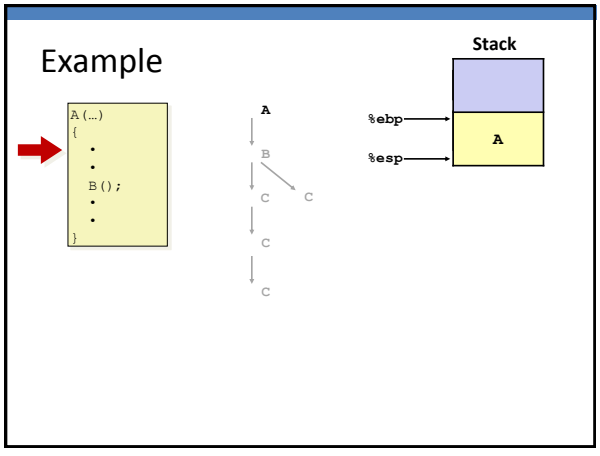


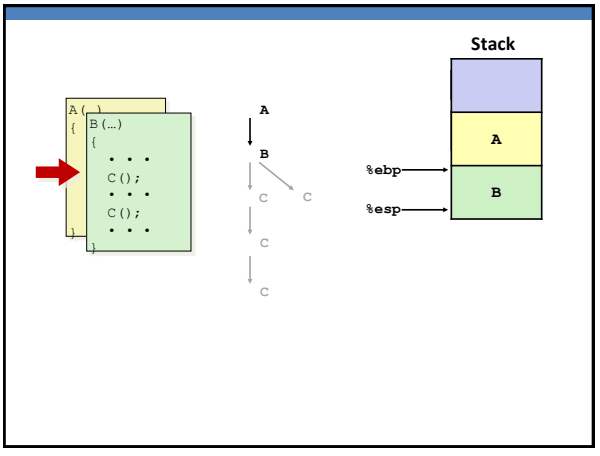
Procedure C() is recursive

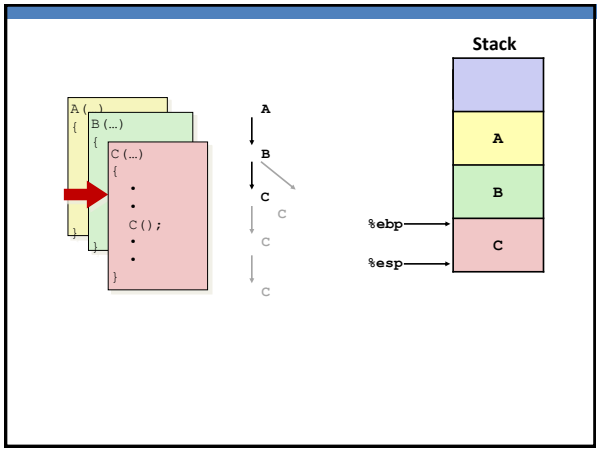
Stack Frames

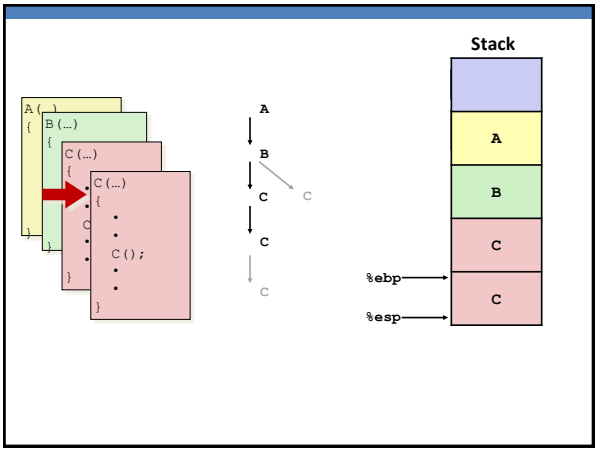
- Contents
  - Local variables
  - Return information
  - Temporary space
- Management
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code

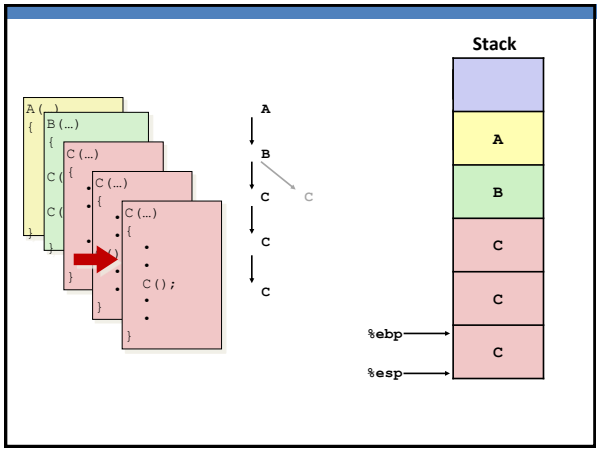


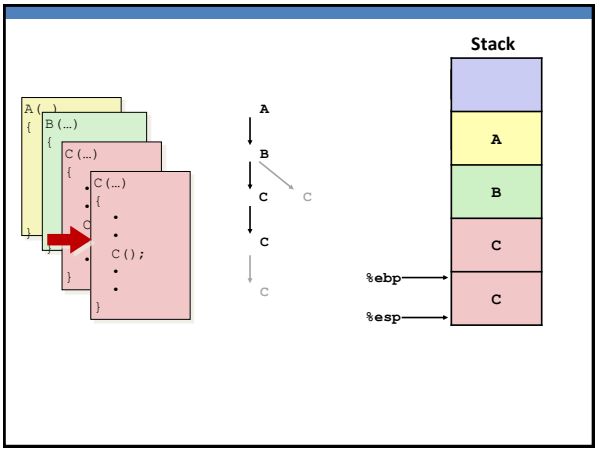

Example
Stack
A (...)
{
B();
}
A
B
C
%ebp
%esp
A

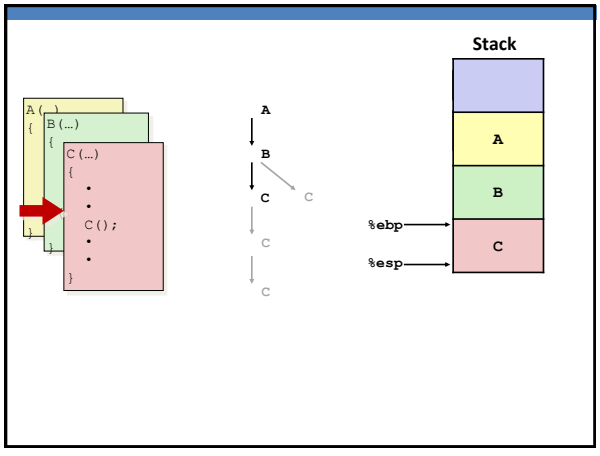

Stack
B (...)
{
C();
C();
}
A
B
C
%ebp
%esp
A
B

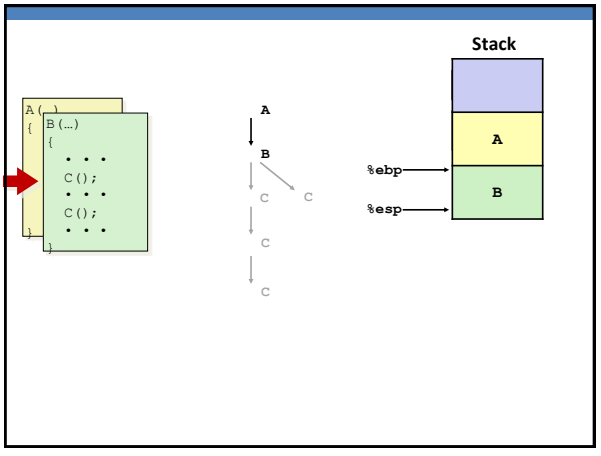

Stack
C (...)
{
C();
}
A
B
C
%ebp
%esp
A
B
C

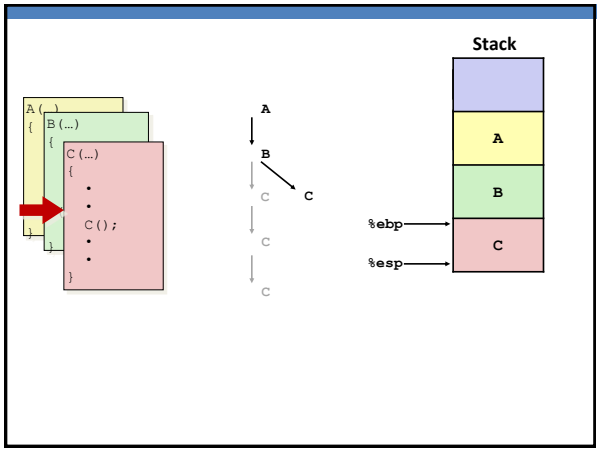

Stack
A
B
C
%ebp
%esp
A
B
C
C

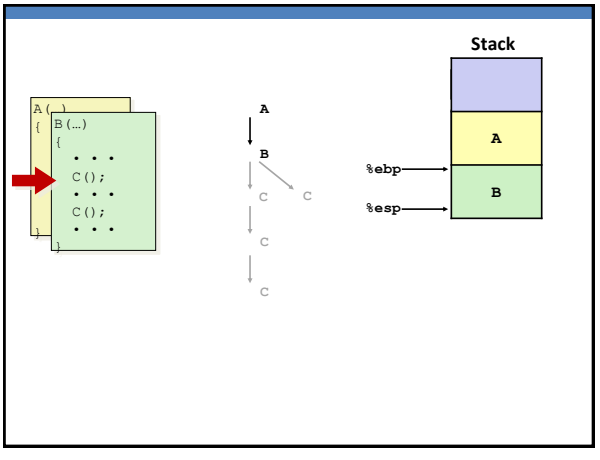

Stack
A
B
C
C
%ebp
%esp
A
B
C
C
C

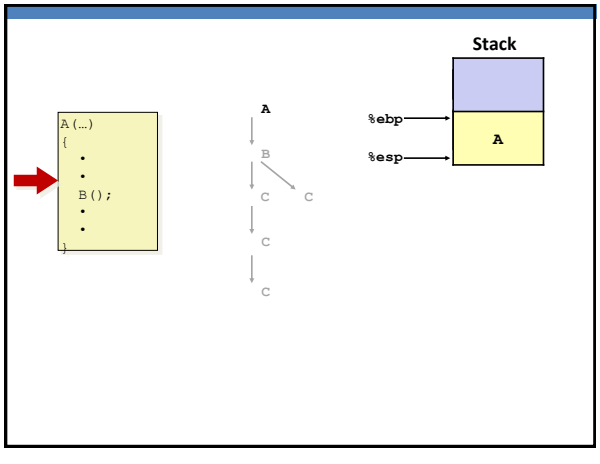

Stack
A
B
C
C
%ebp
%esp
A
B
C
C





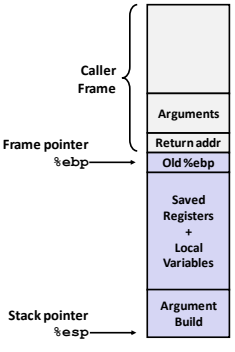






IA32/Linux Stack Frame

- Current Stack Frame (top to bottom)
  - Parameters for function about to call (*argument build*)
  - Local variables (if can't keep in registers)
  - *Saved register* context
  - Old frame pointer
- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



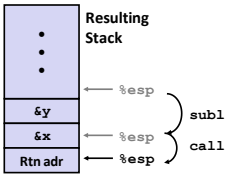
Revisiting **swap**

```
int x = 15213; // global var
int y = 18243;

void call_swap() {
  swap(&x, &y);
}
```

Calling `swap` from `call_swap`

```
call_swap:
    • • •
    subl   $8, %esp
    movl   $y, 4(%esp)
    movl   $x, (%esp)
    call   swap
    • • •
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Revisiting **swap**

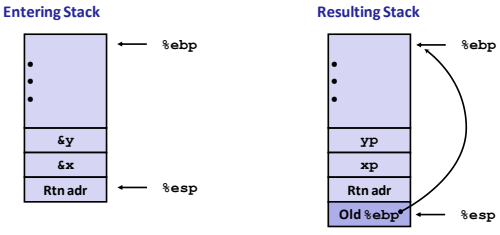
```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx

    movl  8(%ebp), %edx
    movl  12(%ebp), %ecx
    movl  (%edx), %ebx
    movl  (%ecx), %eax
    movl  %eax, (%edx)
    movl  %ebx, (%ecx)

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

Set Up — Body — Finish

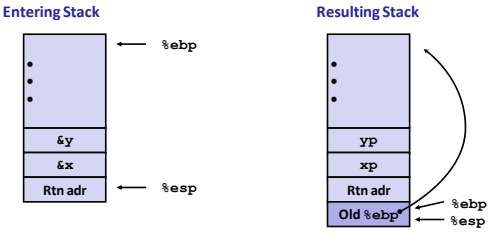
→ Will be explained shortly.

swap Setup #1



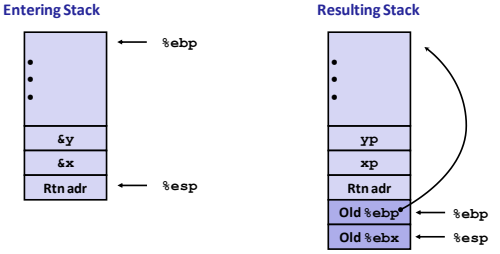

```
swap:
    pushl   %ebp
    movl    %esp,%ebp
    pushl   %ebx
```

swap Setup #2



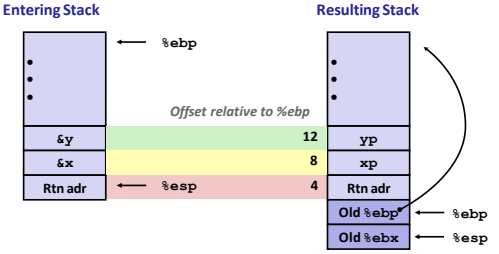

```
swap:
    pushl   %ebp
    movl    %esp,%ebp
    pushl   %ebx
```

swap Setup #3



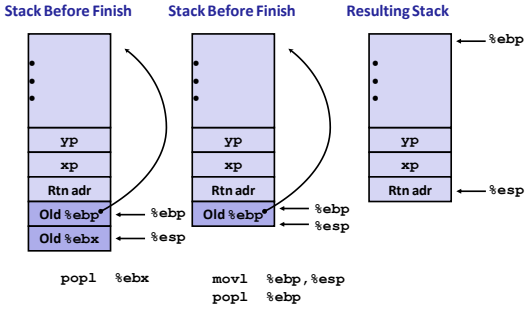

```
swap:
    pushl   %ebp
    movl    %esp,%ebp
    pushl   %ebx
```

swap Body




```
movl  8(%ebp),%edx   # get xp
movl  12(%ebp),%ecx  # get yp
. . .
```

swap Finish



swap Finish

- Observation
  - Saved and restored register **%ebx**
  - Not so for **%eax**, **%ecx**, **%edx**
- **leave** instruction may replace:

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

Register Saving Conventions

- When procedure **A** calls **B**:
  - **A** is the ***caller***
  - **B** is the ***callee***
- Can register be used for temporary storage?

```
A:
   • • •
   movl     $15213, %edx
   call B
   addl     %edx, %eax
   • • •
   ret
```

```
B:
   • • •
   movl     8(%ebp), %edx
   addl     $18243, %edx
   • • •
   ret
```

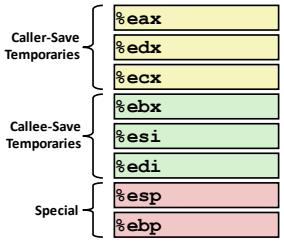
  - Contents of register **%edx** overwritten by **B**
  - This could be trouble → something should be done!

Register Saving Conventions

- Conventions
  - ***"Caller Save"***
    - Caller saves temporary values in its frame before the call
  - ***"Callee Save"***
    - Callee saves temporary values in its frame before using

IA32 Register Usage

- **%eax**, **%edx**, **%ecx**
  - Caller saves prior to call if values are used later
- **%eax**
  - also used to return integer value
- **%ebx**, **%esi**, **%edi**
  - Callee saves if wants to use them
- **%esp**, **%ebp**
  - special form of callee save
  - Restored to original values upon exit from procedure



Practice Problems

- Read CSaPP Sec. 3.7 and try the following problems:

  3.30, 3.31, 3.33, 3.34