

CS 33

Machine Programming (4)

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.

Today

- **IA 32 procedures**
 - **stack structure**
 - **examples of recursion & pointers**
- **x86-64 procedures**

Supplied by CMU.

Stacks and Procedures (1)

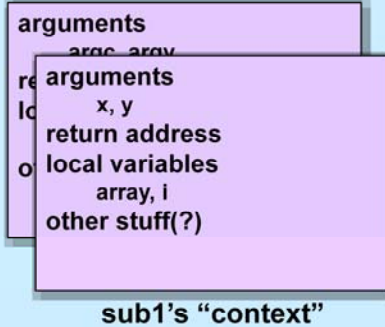
```
int main(int argc, char *argv[]) {  
    char buf[1024];  
    int res, a1, a2, a3, a4;  
  
    ...  
  
    res = sub1(a1, a2);  
    a3 = res + 1;  
  
    ...  
  
    res = sub2(a2, a3, a4);  
    res += 6;  
  
    ...  
  
    return 0;  
}
```

arguments
 argc, argv
return address
local variables
 buf, res, a1, a2, a3, a4
other stuff(?)

main's "context"

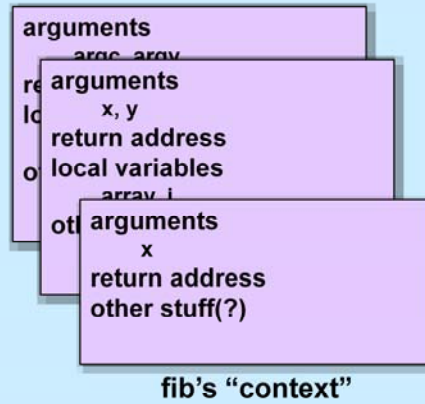
Stacks and Procedures (2)

```
int sub1(int x, int y) {  
    int array[24];  
    int i;  
  
    ...  
  
    for (i=0; i<24; i++)  
        array[i] = fib[i];  
  
    ...  
  
    return sub10(24, array);  
}
```



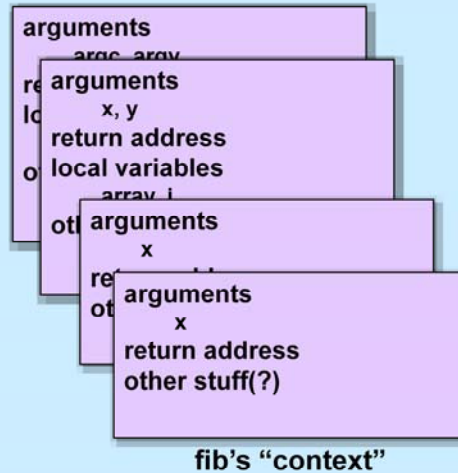
Stacks and Procedures (3)

```
int fib(int x) {  
    if (x < 2)  
        return x;  
    else  
        return fib(x-1)+fib(x-2);  
}
```



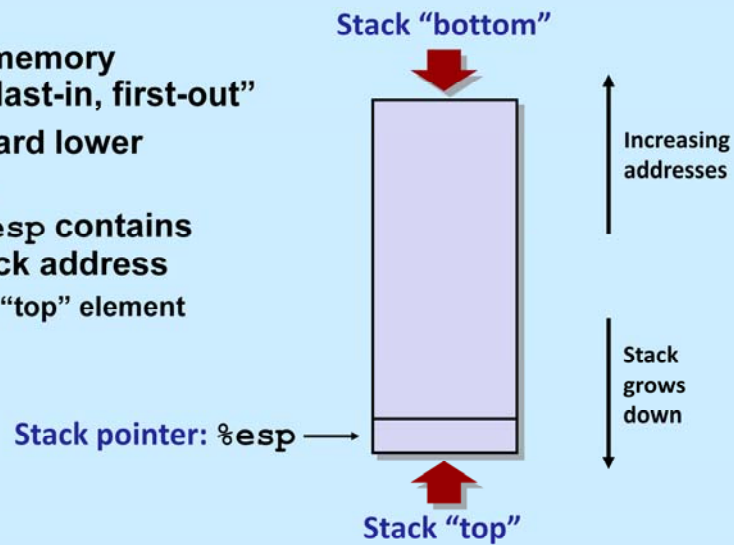
Stacks and Procedures (4)

```
int fib(int x) {  
    if (x < 2)  
        return x;  
    else  
        return fib(x-1)+fib(x-2);  
}
```



IA32 Stack

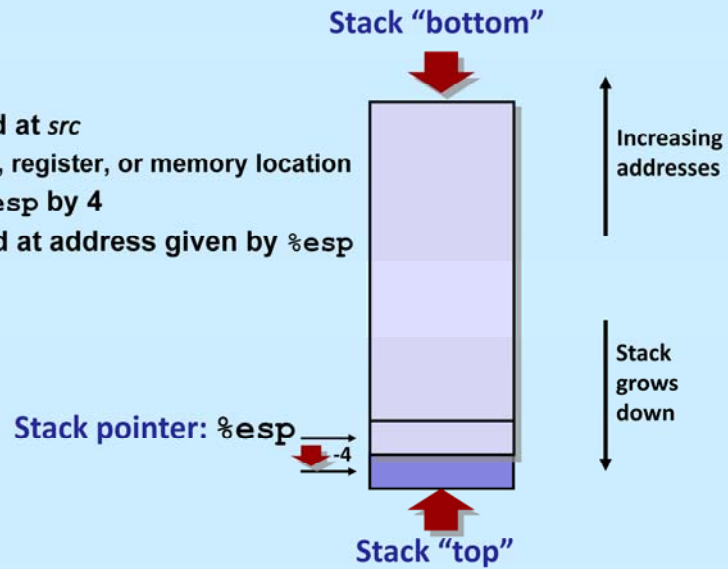
- Region of memory managed “last-in, first-out”
- Grows toward lower addresses
- Register `%esp` contains lowest stack address
 - address of “top” element



Supplied by CMU.

IA32 Stack: Push

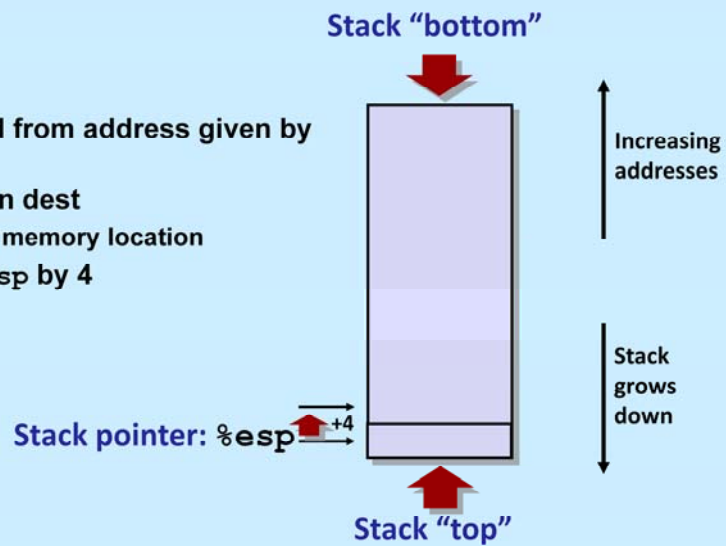
- `pushl src`
 - fetch operand at `src`
 - » immediate, register, or memory location
 - decrement `%esp` by 4
 - store operand at address given by `%esp`



Supplied by CMU.

IA32 Stack: Pop

- `popl dest`
 - fetch operand from address given by `%esp`
 - put operand in `dest`
 - » register or memory location
 - increment `%esp` by 4



Supplied by CMU.

Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call sub`
 - push return address on stack
 - jump to *sub*
- **Return address:**
 - address of the next instruction right after call
 - example from disassembly

```
804854e: e8 3d 06 00 00    call    8048b90 <sub>  
8048553: 50               pushl   %eax
```

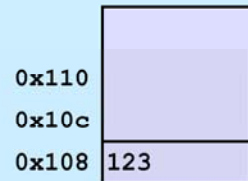
– return address = 0x8048553

- **Procedure return:** `ret`
 - pop address from stack
 - jump to address

Procedure Call

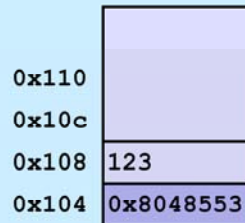
| | | |
|----------|----------------|--------------------|
| 804854e: | e8 3d 06 00 00 | call 8048b90 <sub> |
| 8048553: | 50 | pushl %eax |

call 8048b90



%esp 0x108

%eip 0x804854e



%esp 0x104

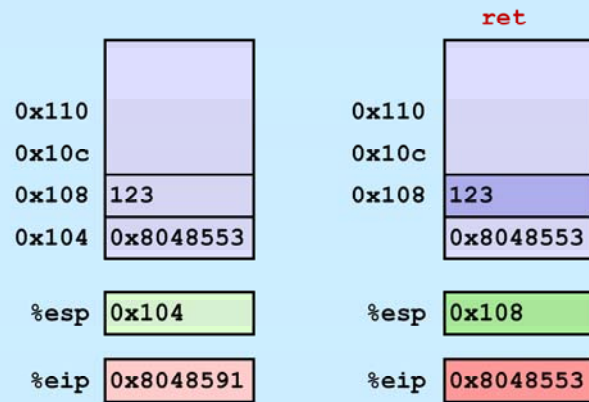
%eip 0x8048b90

%eip: program counter

Supplied by CMU.

Procedure Return

8048591: c3 ret



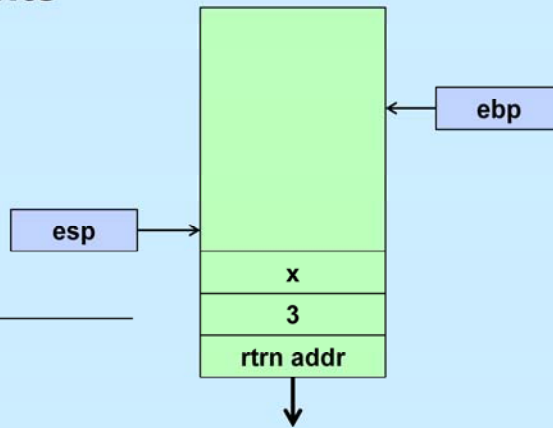
%eip: program counter

Supplied by CMU.

Passing Arguments

```
int x;  
int res;  
int main() {  
    ...  
    res = subr(3, x);  
    ...  
}
```

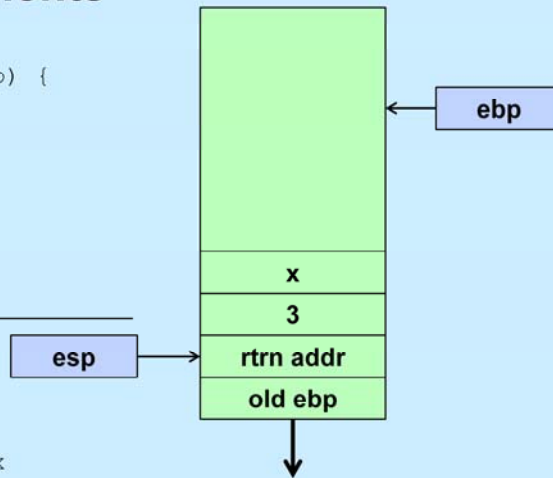
```
main:  
    ...  
    pushl x  
    pushl $3  
    call subr  
    movl %eax, res  
    ...
```



Retrieving Arguments

```
int subr(int a, int b) {  
    return a + b;  
}
```

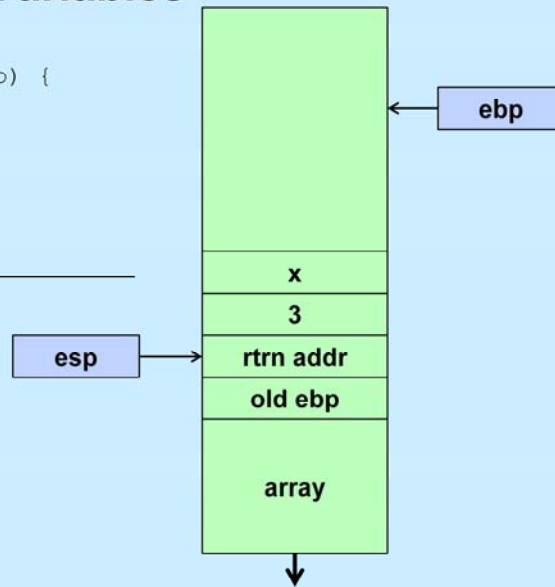
```
subr:  
    pushl %ebp  
    movl %esp, %ebp  
    movl 12(%ebp), %eax  
    addl 8(%ebp), %eax  
    popl %ebp  
    ret
```



Space for Local Variables

```
int subr(int a, int b) {  
    int array[20];  
    ...  
}
```

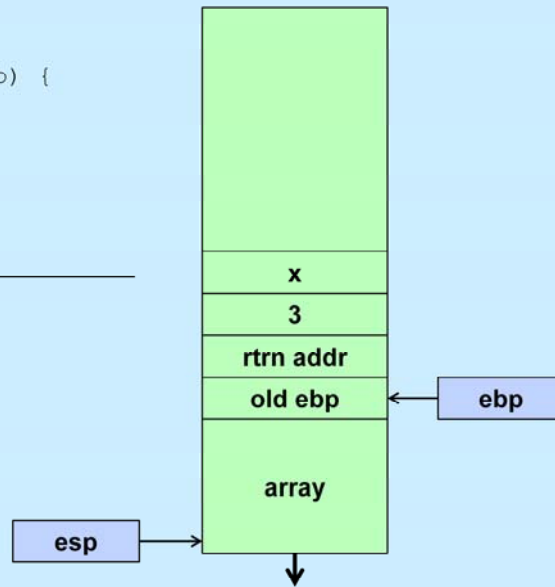
```
subr:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $80, %esp  
    ...  
    addl $80, %esp  
    popl %ebp  
    ret
```



Quick Exit ...

```
int subr(int a, int b) {  
    int array[20];  
    ...  
}
```

```
subr:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $80, %esp  
    ...  
    leave  
    ret
```



The *leave* instruction causes the contents of *ebp* to be copied into *esp*, thereby removing everything from the stack that had been pushed into the frame. It then pops the current stack top (the old *ebp*) into the *ebp* register. The effect of *leave* is thus to return to the caller's stack frame.

There is an *enter* instruction that has the same effect as that of the first three instructions of *subr* combined (it has an operand that indicates how much space for local variables to allocate). However, it's not used by *gcc*, apparently because it's slower than doing it as shown in the slide.

Register-Saving Conventions

- When procedure `yoo` calls `who`:
 - `yoo` is the *caller*
 - `who` is the *callee*
- Can registers be used for temporary storage?

```
yoo:
. . .
movl $33, %edx
call who
addl %edx, %eax
. . .
ret
```

```
who:
. . .
movl 8(%ebp), %edx
addl $32, %edx
. . .
ret
```

- contents of register `%edx` overwritten by `who`
- this could be trouble: something should be done!
 - » need some coordination

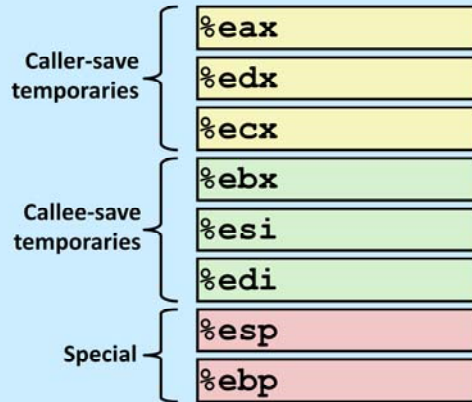
Register-Saving Conventions

- When procedure `yoo` calls `who`:
 - `yoo` is the *caller*
 - `who` is the *callee*
- Can register be used for temporary storage?
- Conventions
 - “*caller save*”
 - » caller saves temporary values on stack before the call
 - » restores them after call
 - “*callee save*”
 - » callee saves temporary values on stack before using
 - » restores them before returning

Supplied by CMU.

IA32/Linux+Windows 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



Supplied by CMU.

Register-Saving Example

```
yoo:
...
movl $33, %edx
pushl %edx
call who
popl %edx
addl %edx, %eax
...
ret
```

```
who:
...
pushl %ebx
...
movl 4(%ebp), %ebx
addl %53, %ebx
movl 8(%ebp), %edx
addl $32, %edx
...
popl %ebx
...
ret
```

Supplied by CMU.

Today

- **IA 32 procedures**
 - stack structure
 - examples of recursion & pointers
- **x86-64 procedures**

Supplied by CMU.

Recursive Function

```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

- **Registers**

- **%eax, %edx** used without first saving
- **%ebx** used, but saved at beginning & restored at end

```
pcount_r:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $4, %esp
    movl 8(%ebp), %ebx
    movl $0, %eax
    testl %ebx, %ebx
    je .L3
    movl %ebx, %eax
    shrl %eax
    movl %eax, (%esp)
    call pcount_r
    movl %ebx, %edx
    andl $1, %edx
    leal (%edx,%eax), %eax
.L3:
    addl $4, %esp
    popl %ebx
    popl %ebp
    ret
```

Supplied by CMU.

Recursive Call #1

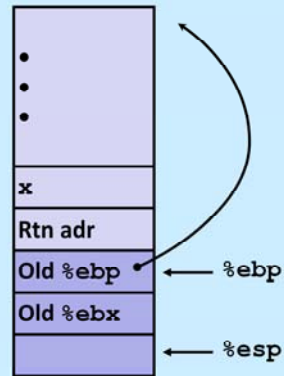
```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

- **Actions**

- save old value of %ebx on stack
- allocate space for argument to recursive call
- store x in %ebx

%ebx x

```
pcount_r:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $4, %esp
    movl 8(%ebp), %ebx
    . . .
```



Supplied by CMU.

Recursive Call #2

```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

```
...
movl $0, %eax
testl %ebx, %ebx
je .L3
...
.L3:
...
ret
```

- **Actions**

- if `x == 0`, return
 - » with `%eax` set to 0

`%ebx` x

Supplied by CMU.

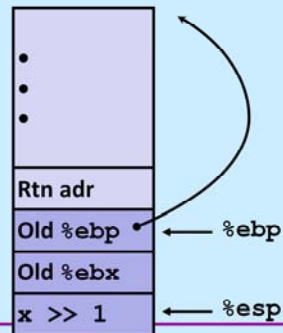
Recursive Call #3

```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

```
...
movl  %ebx, %eax
shrl  %eax
movl  %eax, (%esp)
call  pcount_r
...
```

- **Actions**
 - store $x \gg 1$ on stack
 - make recursive call
- **Effect**
 - $\%eax$ set to function result
 - $\%ebx$ still has value of x

$\%ebx$ x



Supplied by CMU.

Recursive Call #4

```
/* Recursive popcount */  
int pcount_r(unsigned x) {  
    if (x == 0)  
        return 0;  
    else return  
        (x & 1) + pcount_r(x >> 1);  
}
```

```
. . .  
movl    %ebx, %edx  
andl    $1, %edx  
leal    (%edx,%eax), %eax  
. . .
```

- **Assume**
 - %eax holds value from recursive call
 - %ebx holds x
- **Actions**
 - compute $(x \& 1) + \text{computed value}$
- **Effect**
 - %eax set to function result

%ebx

x

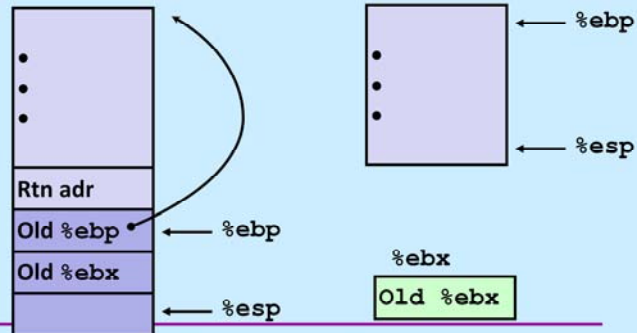
Recursive Call #5

```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

```
...
L3:
    addl$4, %esp
    popl%ebx
    popl%ebp
    ret
```

• Actions

- restore values of %ebx and %ebp
- restore %esp



Supplied by CMU.

Observations About Recursion

- **Handled without special consideration**
 - stack frames mean that each function call has private storage
 - » saved registers & local variables
 - » saved return pointer
 - register-saving conventions prevent one function call from corrupting another's data
 - stack discipline follows call / return pattern
 - » if P calls Q, then Q returns before P
 - » last-in, first-out
- **Also works for mutual recursion**
 - P calls Q; Q calls P

Supplied by CMU.

Pointer Code

Generating Pointer

```
/* Compute x + 3 */  
int add3(int x) {  
    int localx = x;  
    incrk(&localx, 3);  
    return localx;  
}
```

Referencing Pointer

```
/* Increment value by k */  
void incrk(int *ip, int k) {  
    *ip += k;  
}
```

- **add3 creates pointer and passes it to incrk**

Supplied by CMU.

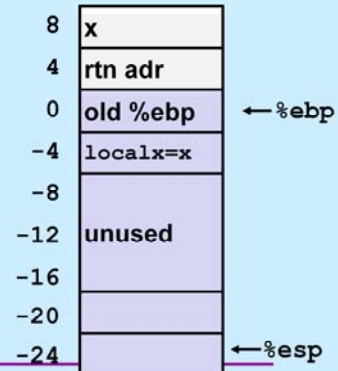
Creating and Initializing Local Variables

```
int add3(int x) {  
    int localx = x;  
    incrk(&localx, 3);  
    return localx;  
}
```

- Variable localx must be stored on stack
 - because: need to create pointer to it
- Compute pointer as -4(%ebp)

First part of add3

```
add3:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $24, %esp      # Alloc. 24 bytes  
    movl 8(%ebp), %eax  
    movl %eax, -4(%ebp) # Set localx to x
```



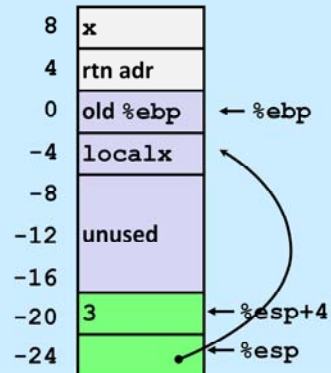
Creating Pointer as Argument

```
int add3(int x) {  
    int localx = x;  
    incrk(&localx, 3);  
    return localx;  
}
```

- Use leal instruction to compute address of localx

Middle part of add3

```
movl $3, 4(%esp)    # 2nd arg = 3  
leal -4(%ebp), %eax # &localx  
movl %eax, (%esp)   # 1st arg = &localx  
call incrk
```



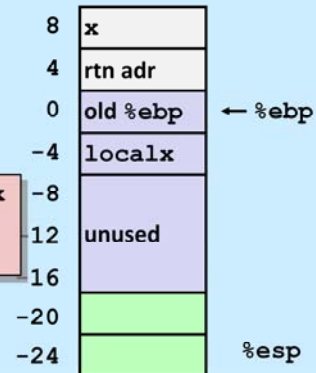
Retrieving local variable

```
int add3(int x) {  
    int localx = x;  
    incrk(&localx, 3);  
    return localx;  
}
```

- Retrieve localx from stack as return value

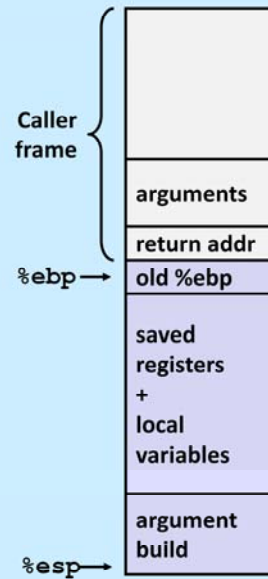
Final part of add3

```
movl -4(%ebp), %eax # Return val= localx  
leave  
ret
```



IA 32 Procedure Summary

- **Important Points**
 - stack is the right data structure for procedure call / return
 - » if P calls Q, then Q returns before P
- **Recursion (& mutual recursion) handled by normal calling conventions**
 - can safely store values in local stack frame and in callee-saved registers
 - put function arguments at top of stack
 - result return in `%eax`
- **Pointers are addresses of values**
 - on stack or global



Supplied by CMU.

Today

- **IA 32 procedures**
 - stack structure
 - examples of recursion & pointers
- **x86-64 procedures**

Supplied by CMU.

x86-64 Integer Registers

| | | | |
|------|------|------|-------|
| %rax | %eax | %r8 | %r8d |
| %rbx | %ebx | %r9 | %r9d |
| %rcx | %ecx | %r10 | %r10d |
| %rdx | %edx | %r11 | %r11d |
| %rsi | %esi | %r12 | %r12d |
| %rdi | %edi | %r13 | %r13d |
| %rsp | %esp | %r14 | %r14d |
| %rbp | %ebp | %r15 | %r15d |

– Twice the number of registers; accessible as 8, 16, 32, 64 bits

Supplied by CMU.

x86-64 Integer Registers: Usage Conventions

| | | | |
|-------------|---------------|-------------|--------------|
| %rax | Return value | %r8 | Argument #5 |
| %rbx | Callee saved | %r9 | Argument #6 |
| %rcx | Argument #4 | %r10 | Caller saved |
| %rdx | Argument #3 | %r11 | Caller Saved |
| %rsi | Argument #2 | %r12 | Callee saved |
| %rdi | Argument #1 | %r13 | Callee saved |
| %rsp | Stack pointer | %r14 | Callee saved |
| %rbp | Callee saved | %r15 | Callee saved |

Supplied by CMU.

x86-64 Registers

- **Arguments passed to functions via registers**
 - if more than 6 integral parameters, then pass rest on stack
 - these registers can be used as caller-saved as well
- **All references to stack frame via stack pointer**
 - eliminates need to update `%ebp/%rbp`
- **Other registers**
 - 6 callee-saved
 - 2 caller-saved
 - 1 return value (also usable as caller-saved)
 - 1 special (stack pointer)

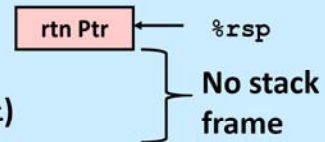
Supplied by CMU.

x86-64 Long Swap

```
void swap_l(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```

- **Operands passed in registers**
 - first (xp) in %rdi, second (yp) in %rsi
 - 64-bit pointers
- **No stack operations required (except ret)**
- **Avoiding stack**
 - can hold all local information in registers

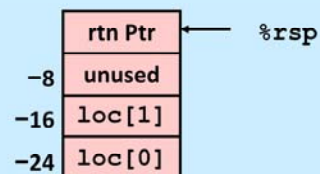


x86-64 Locals in the Red Zone

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
    movq    (%rdi), %rax
    movq    %rax, -24(%rsp)
    movq    (%rsi), %rax
    movq    %rax, -16(%rsp)
    movq    -16(%rsp), %rax
    movq    %rax, (%rdi)
    movq    -24(%rsp), %rax
    movq    %rax, (%rsi)
    ret
```

- **Avoiding stack-pointer change**
 - can hold all information within small window beyond stack pointer
 - » 128 bytes



Supplied by CMU.

The *volatile* keyword tells the compiler that it may not perform optimizations on the associated variable, such as storing it strictly in registers and not in memory. It's used primarily in cases where the variable might be modified via other routines that aren't apparent when the current code is being compiled. We'll see useful examples of its use later. Here it's used simply to ensure that *loc* is allocated on the stack, thus giving us a simple example of using local variables stored on the stack.

The issue here is whether a reference to memory beyond the current stack (as delineated by the stack pointer) is a legal reference. On IA32 it is not, but on x86-64 it is, as long as the reference is not more than 128 bytes beyond the end of the stack.

x86-64 NonLeaf without Stack Frame

```
/* Swap a[i] & a[i+1] */  
void swap_ele(long a[], int i)  
{  
    swap(&a[i], &a[i+1]);  
}
```

- No values held while swap being invoked
- No callee-save registers needed
- `rep` instruction inserted as no-op
 - based on recommendation from AMD
 - » can't handle transfer of control to `ret`

```
swap_ele:  
    movslq %esi,%rsi          # Sign extend i  
    leaq 8(%rdi,%rsi,8), %rax  # &a[i+1]  
    leaq (%rdi,%rsi,8), %rdi   # &a[i] (1st arg)  
    movq %rax, %rsi           # (2nd arg)  
    call swap  
    rep                          # No-op  
    ret
```

Supplied by CMU.

x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
(long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Keeps values of `&a[i]` and `&a[i+1]` in callee-save registers
- Must set up stack frame to save these registers

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq   %esi, %rax
    leaq    8(%rdi, %rax, 8), %rbx
    leaq    (%rdi, %rax, 8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

Supplied by CMU.

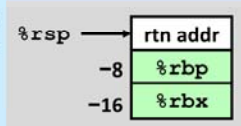
Understanding x86-64 Stack Frame

```
swap_ele_su:
    movq    %rbx, -16(%rsp)    # Save %rbx
    movq    %rbp, -8(%rsp)     # Save %rbp
    subq    $16, %rsp         # Allocate stack frame
    movslq   %esi, %rax        # Extend i
    leaq    8(%rdi,%rax,8), %rbx # &a[i+1] (callee save)
    leaq    (%rdi,%rax,8), %rbp # &a[i]   (callee save)
    movq    %rbx, %rsi        # 2nd argument
    movq    %rbp, %rdi        # 1st argument
    call    swap
    movq    (%rbx), %rax       # Get a[i+1]
    imulq   (%rbp), %rax       # Multiply by a[i]
    addq    %rax, sum(%rip)    # Add to sum
    movq    (%rsp), %rbx      # Restore %rbx
    movq    8(%rsp), %rbp     # Restore %rbp
    addq    $16, %rsp         # Deallocate frame
    ret
```

Supplied by CMU.

Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)    # Save %rbx
movq    %rbp, -8(%rsp)     # Save %rbp
```



```
subq    $16, %rsp         # Allocate stack frame
```

• • •



```
movq    (%rsp), %rbx      # Restore %rbx
movq    8(%rsp), %rbp     # Restore %rbp
```

```
addq    $16, %rsp         # Deallocate frame
```

Supplied by CMU.

Interesting Features of Stack Frame

- **Allocate entire frame at once**
 - all stack accesses can be relative to `%rsp`
 - do by decrementing stack pointer
 - can delay allocation, since safe to temporarily use red zone
- **Simple deallocation**
 - increment stack pointer
 - no base/frame pointer needed

Supplied by CMU.

x86-64 Procedure Summary

- **Heavy use of registers**
 - parameter passing
 - more temporaries since more registers
- **Minimal use of stack**
 - sometimes none
 - allocate/deallocate entire block
- **Many tricky optimizations**
 - what kind of stack frame to use
 - various allocation techniques

Supplied by CMU.