x86 : move, leaq
imitate C in Assembly code

function calls

Passing control
To beginning of procedure code
Back to return point
Passing data
Procedure arguments
Return value
Memory management
Allocate during procedure execution
Deallocate upon return

Accomplished using instructions + conventions

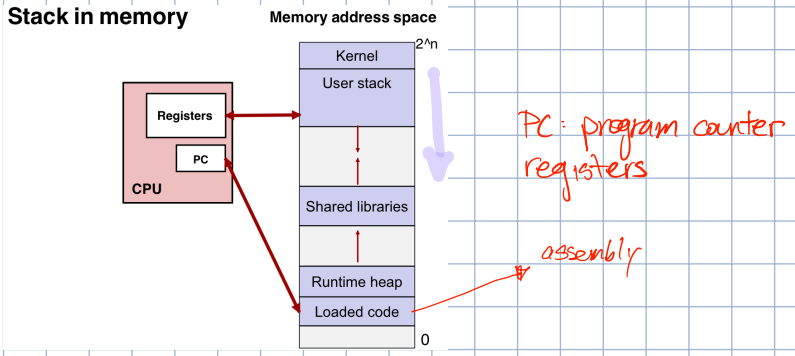
```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

fn call

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

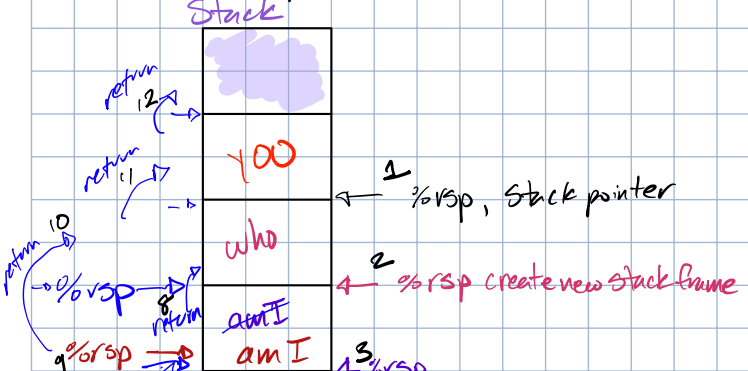
Assembly uses memory & CPU instructions

Stack: part of memory



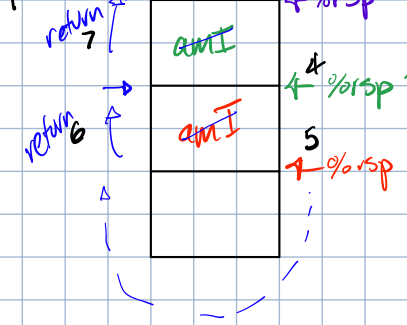
fn local variables go in User stack (lower end → "stack frame"
more memory needed? grow downward

stack grows to bottom
"stack top" visually the bottom




```
foo (…)
{
  …
  who (…)
}
```
call new fn

```
who (…)
{
  …
  amI(…); ★ call amI
  …
```



despite calling itself, different local values


```
amI(...); ★ after all returns call it again
...
}

amI(...)
{
...
amI(...)  ★ call itself ★ call again
...
}
```

%rsp pointing to top of stack frame

What's in stack frame?

Caller Stack Frame

Return address

Pushed by `call` instruction

Arguments for this call (optional)

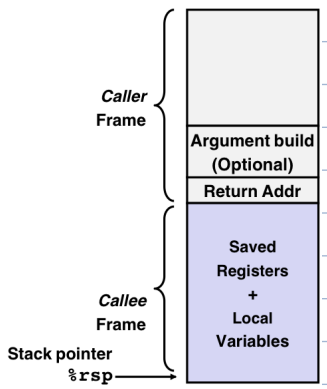
Callee Stack Frame

Saved register context

Local variables

If can't keep in registers

Arguments for function about to call (optional)



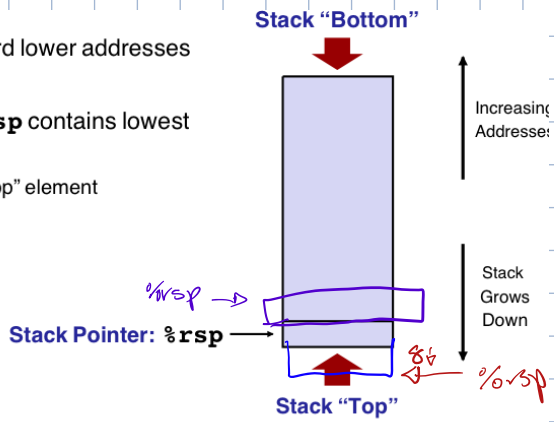
pushq → pt 8 byte buffer

grow stack, more to lower address

Grows toward lower addresses

Register `%rsp` contains lowest stack address

address of "top" element



pushq Src

decrement %rsp by 8

pushq %rsp push 8 byte integer to top

popq Src

saves memory. deallocates 8 bytes from top

popq %rbx

only changes @ top

Passing control

each line has unique address

```
void multstore
 (long x, long y, long *dest){
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
 400540: push   %rbx     # Save %rbx
 400541: mov    %rdx,%rbx    # Save dest
 400544: callq  400550 <mult2>   # mult2(x,y)
 400549: mov    %rax,(%rbx)  # Save at dest
 40054c: pop    %rbx     # Restore %rbx
 40054d: ret             # Return
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax   # a
  400553:  imul   %rsi,%rax   # a * b
  400557:  ret            # Return
```

calls by callq address line

Procedure call

push return address on stack (1)

address of next instruction after call

jump to label

Procedure return

pop address from stack

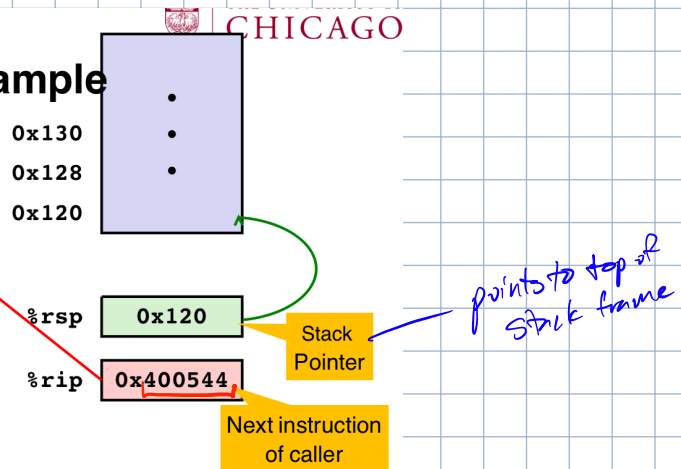
jump to address (2)

CHICAGO

Control Flow Example

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

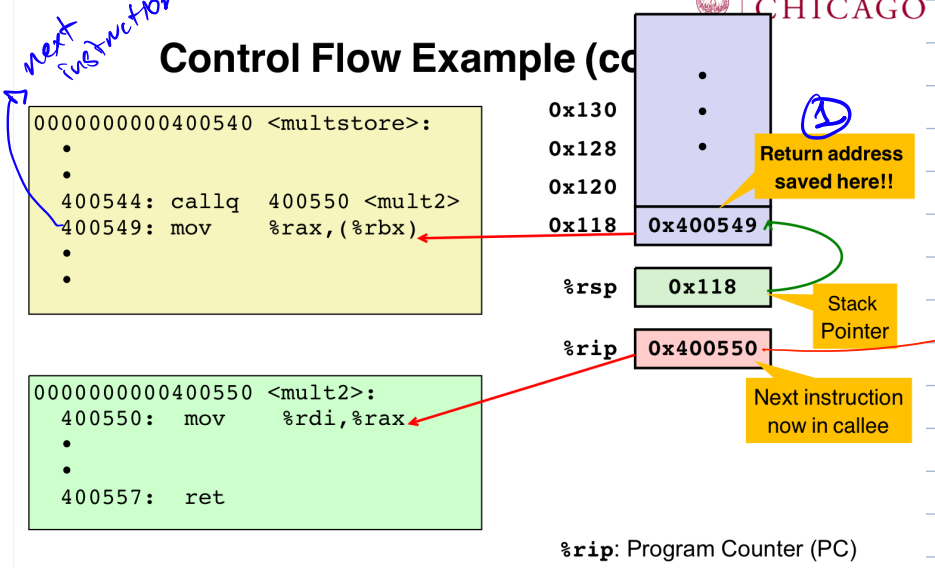
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



points to top of stack frame

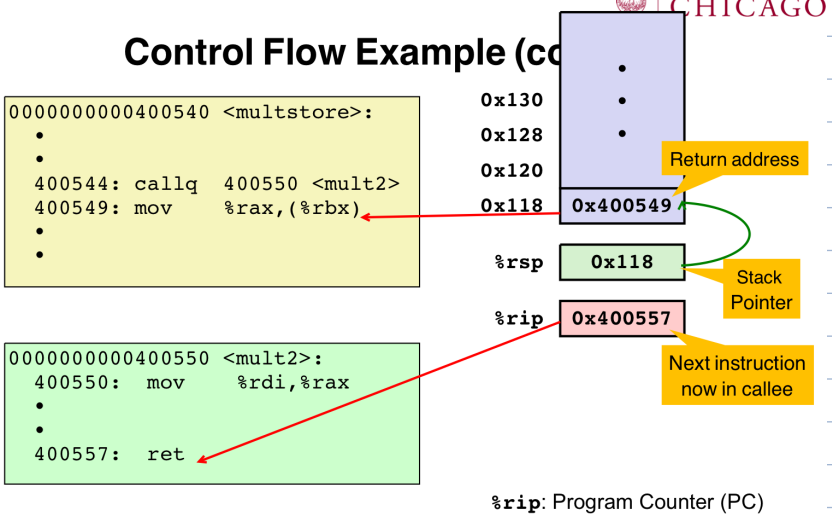
%rip: Program Counter (PC)

Control Flow Example (cc



%rip: Program Counter (PC)

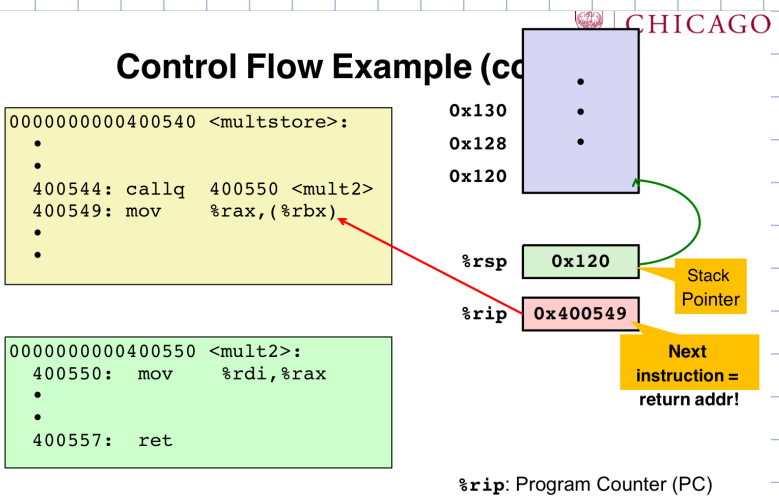
Control Flow Example (cc



%rip: Program Counter (PC)

move back to caller

Control Flow Example (cc



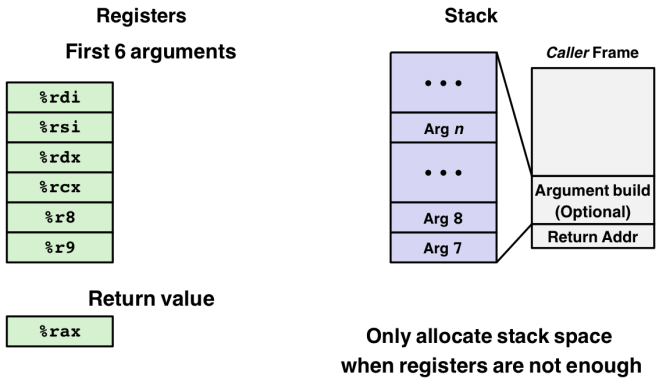
%rip: Program Counter (PC)

arguments? then registers, they don't change

6 args: %rdi %rsi %rdx %rcx %r8 %r9

return value: %rax

Managing local data



register values don't change

can be overwritten

When procedure yoo calls who:

yoo is the *caller*

who is the *callee*

```
yoo:
    • • •
    movq  $60637, %rdx
    call  who
    addq  %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq  $15213, %rdx
    • • •
    ret
```

Contents of register **%rdx** overwritten by **who**

This could be trouble → something should be done!

Need some coordination

When procedure yoo calls who:

yoo is the *caller*

who is the *callee*

Conventions

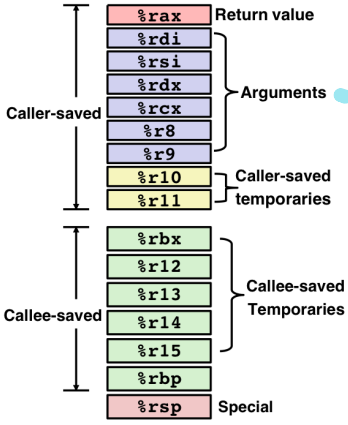
"Caller Saved"

Caller saves temporary values in its frame before the call

"Callee Saved"

Callee saves temporary values in its frame before using

Callee restores them before returning to caller



may change
save elsewhere get back after f'n call

1st arg: rdi

2nd arg: rsi

Review: Stack

Each "function" has its own space on stack, called a *stack* frame

Stack frame management

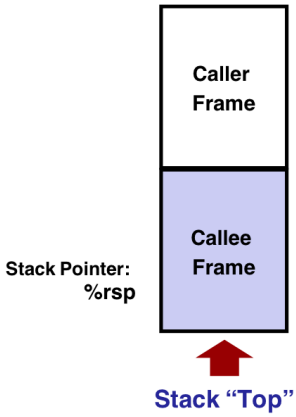
Allocated when enter procedure ("**call**" code)

Deallocated when returns ("**ret**" code)

Last-in first-out" (LIFO) stack discipline matches function call/ret patterns

If P calls Q, then Q returns before P

%rsp must be put back to the end of Caller Frame when Callee returns



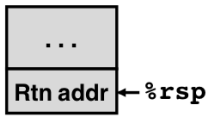
```
long call_incr2(long x) {
    long v1 = 60637;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
    pushq   %rbx
    subq    $16, %rsp
    movq    $60637, 8(%rsp)
    movq    %rdi, %rbx
    movq    $3000, %rsi
    leaq    8(%rsp), %rdi
    call    incr
    addq    %rbx, %rax
    addq    $16, %rsp
    popq    %rbx
    ret
```

Caller-saved: %rdi, %rsi, %rax
Callee-saved: %rbx

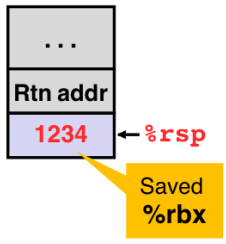
Initial states

Register	Use(s)
%rdi	x (1st arg)
%rsi	
%rbx	1234
%rax	



Resulting states

Register	Use(s)
%rdi	x (1st arg)
%rsi	
%rbx	1234
%rax	



look @ slides for example

Stack frames mean that each function call has private storage

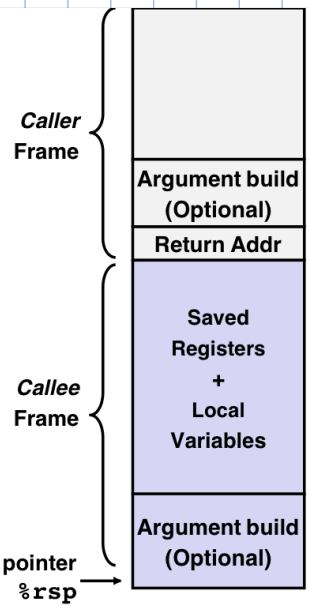
- Saved registers & local variables
- Saved return pointer

Register saving conventions prevent one function call from corrupting another's data

- Unless the C code explicitly does so (e.g., buffer overflow)

Stack discipline follows call / return pattern

- "If P calls Q, then Q returns before P" <==> "Stack: Last-In, First-Out"
- Also works for mutual recursion: P calls Q; Q calls P



Buffer Overflow

allocate buffer/space in C to hold variable, but its bigger than space → comption

Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

get string from terminal until enter

No way to specify limit on number of characters to read

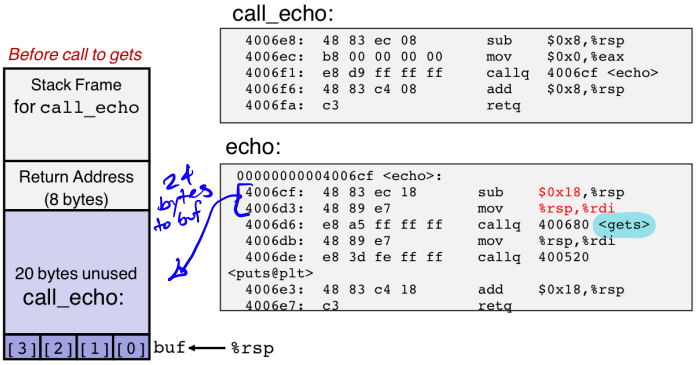
Similar problems with other library functions

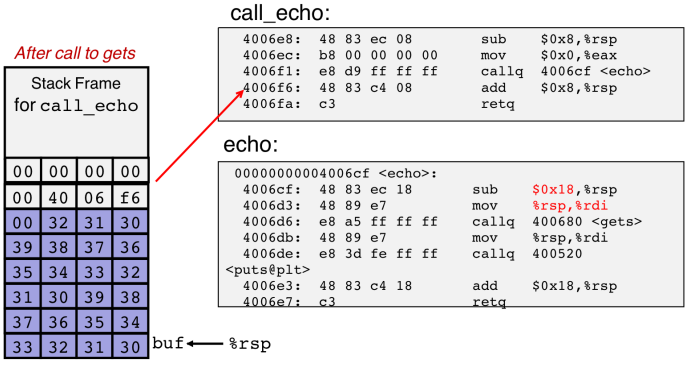
strcpy, **strcat**: Copy strings of arbitrary length

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

need more than 4 bytes

```
void call_echo() {
    echo();
}
```



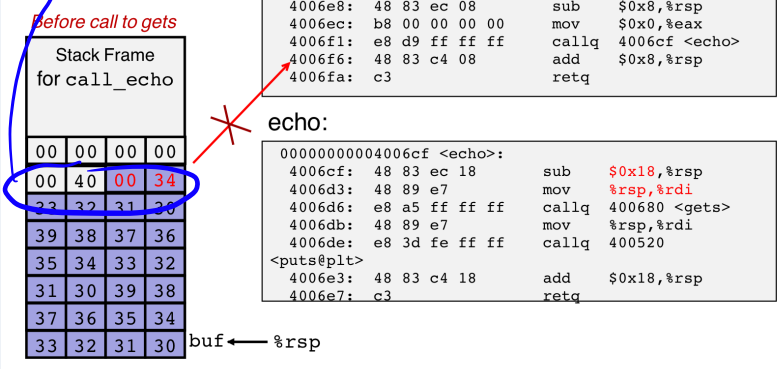


What happens with input "0123456789012456789012"

ASCII of '0' is 0x30, '1' is 0x31, …

return address fucks up

CPU can't read → seg. fault



can inject malicious code

What happens with input "0123456789012456789012345"

ASCII of '0' is 0x30, '1' is 0x31, …