

# 6.035

Fall 2006

## Project 3: Unoptimized Code Generation

Michael Gordon

# Segment III Roadmap

- Design and Checkpoint
  - Due Thursday 10/19
  - Checkpoint
    - Hand-in a tarball of what you have
    - If you get codegen to work, no effect
    - If you have problems at end, we will be very harsh if you haven't done much work by the checkpoint
- Group meeting
  - not mandatory, meet with me if you want.
- Implementation and Report
  - Due on 10/26

# New Machines

- Meet `tyner.csail.mit.edu` and `silver.csail.mit.edu`!
- We received two new AMD64 machines
  - dual processor
  - dual core per processor
  - 8 gigs of RAM
- They are being setup now.

# Unoptimized Code Generation

- Translate all the instructions in the intermediate representation to assembly language
  - Allocate space for the variables.
    - Globals
    - Arrays
  - Adhere to calling conventions
  - Short circuiting
  - Runtime checks

# x86-64 (AMD64)

- Stack values are 64-bit (8-byte)
- Values in decaf are 32-bit or 1-bit
- For this phase, we are not optimizing for space
- Use 64-bits (quadword) for ints and bools.
- Use instructions that operate on 64-bit values for stack and mem operations, e.g. mov
- Arithmetic instructions have 32-bit operands, add, sub, etc

# Registers

Register	Purpose	Saved across calls
<b>%rax</b>	temp register; return value	No
<b>%rbx</b>	callee-saved	Yes
<b>%rcx</b>	used to pass 4th argument to functions	No
<b>%rdx</b>	used to pass 3rd argument to function	No
<b>%rsp</b>	stack pointer	Yes
<b>%rbp</b>	callee-saved; base pointer	Yes
<b>%rsi</b>	used to pass 2nd argument to function	No
<b>%rdi</b>	used to pass 1st argument to functions	No
<b>%r8</b>	used to pass 5th argument to functions	No
<b>%r9</b>	used to pass 6th argument to functions	No
<b>%r10-r11</b>	temporary	No
<b>%r12-r15</b>	callee-saved registers	Yes

# Composition of an Object File

- We use the ELF file format
- The object file has:
  - Multiple Segments
  - Symbol Information
  - Relocation Information
- Segments
  - Global Offset Table
  - Procedure Linkage Table
  - Text (code)
  - Data
  - Read Only Data

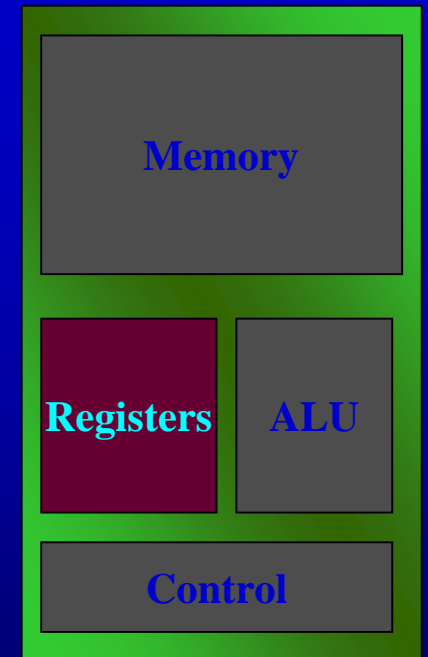
```
.file    "test2.c"
.section .rodata
.LC0:
.string "error %d"

.section .text
.globl fact
fact:
    pushq   %rbp
    movq   %rsp, %rbp
    subq   $16, %rsp
    movl   -8(%rbp), %eax
    leave
    ret
.
.comm    bar,4,4
.comm    a,1,1
.comm    b,1,1

.section .eh_frame,"a",@progbits
.long   .LECIE1-.LSCIE1
.long   0x0
.byte   0x1
.string ""
.uleb128 0x1
```

# Registers

- Instructions allow only limited memory operations
  - ~~add -4(%rbp), -8(%rbp)~~
  - mov -4(%rbp), %r10
  - add %r10, -8(%rbp)
- Important for performance
  - limited in number
- Special registers
  - %rbp                    base pointer
  - %rsp                    stack pointer



# Allocating Read-Only Data

- All Read-Only data in the text segment
- Integers
  - use load immediate
- Strings
  - use the `.string` macro

```
.section .text
.globl main
main:
    enter    $0, $0
    mov     $.msg, %rdi
    mov     $5, %rsi
    mov     $0, %rax
    call    printf
    leave
    ret

.msg:
    .string "Five: %d\n"
```

# Assembly language

- Relocatable machine language (object modules)
  - all locations(addresses) represented by symbols
  - Mapped to memory addresses at link and load time
  - Flexibility of separate compilation

# Global Variables

- Allocation: Use the assembler's `.comm` directive
- Use PC relative addressing
  - `%rip` is the current instruction address
  - `X(%rip)` will add the offset from the current instruction location to the space for `x` in the data segment to `%rip`
  - Creates easily relocatable binaries

```
.section .text
.globl main
main:
    enter $0, $0
    mov     $.msg, %rdi
    mov     x(%rip), %rsi
    mov     $0, %rax
    call   printf
    leave
    reto

.msg:
.string "x: %d\n"

.comm     x, 8, 8
```

## `.comm name, size, alignment`

The `.comm` directive allocates storage in the data section. The storage is referenced by the identifier *name*. *Size* is measured in bytes and must be a positive integer. *Name* cannot be predefined. *Alignment* is optional. If *alignment* is specified, the address of *name* is aligned to a multiple of *alignment*

# What about Arrays

- What code would you write for?

ex: `a[4] = 5;`

...

```
mov $5, %r10
```

```
mov $4, %r11
```

```
???
```

...

```
.comm a, 8 * 10, 8
```

# Array Addressing

- The data segment grows toward larger addresses.
- How to access an array element?
- We want something like
  - $\text{base} + \text{offset} * \text{type\_size}$
  - $(a + \%rip) + (\text{element} * \text{type\_size})$
- Ex:
  - if  $a$  is  $\text{int}[10]$ ,  $a[4]$ 's address is
    - $(a + \%rip) + (4 * 8)$

# Addressing Modes

- x86-64 supports complex addressing modes  
offset(base,index,scale)

where the address is calculated as:

$$\text{offset} + \text{base} + \text{index} * \text{scale}$$

- Forget about offset! Always use 0.

# What about Arrays

- What code would you write for?

ex: `a[4] = 5;`

...

```
mov $5, %r10
```

```
mov $4, %r11
```

```
???
```

...

```
.comm a, 8 * 10, 8
```

# What about Arrays

- What code would you write for?

ex: `a[4] = 5;`

```
...  
mov $5, %r10  
mov $4, %r11  
mov %r10, (a(%r11, 8)(%rip))  
...  
.comm a, 8 * 10, 8
```

# lea

- Load effective address
- load address of source operand

`lea (%rax), %r11` -- put the value of %rax in %r11

- Compare to mov:

`mov (%rax), %r11` -- put the value of memory address pointed to by %rax in %r11

- But what is important is that we can use addressing modes in lea:

`lea a(%rip), %rax`

# What about Arrays

- What code would you write for?

ex: `a[4] = 5;`

```
...  
mov $5, %r10  
mov $4, %r11  
lea a(%rip), %rax  
mov %r10, 0(%rax, %r11, 8)  
...  
.comm a, 8 * 10, 8
```

# Procedure Abstraction

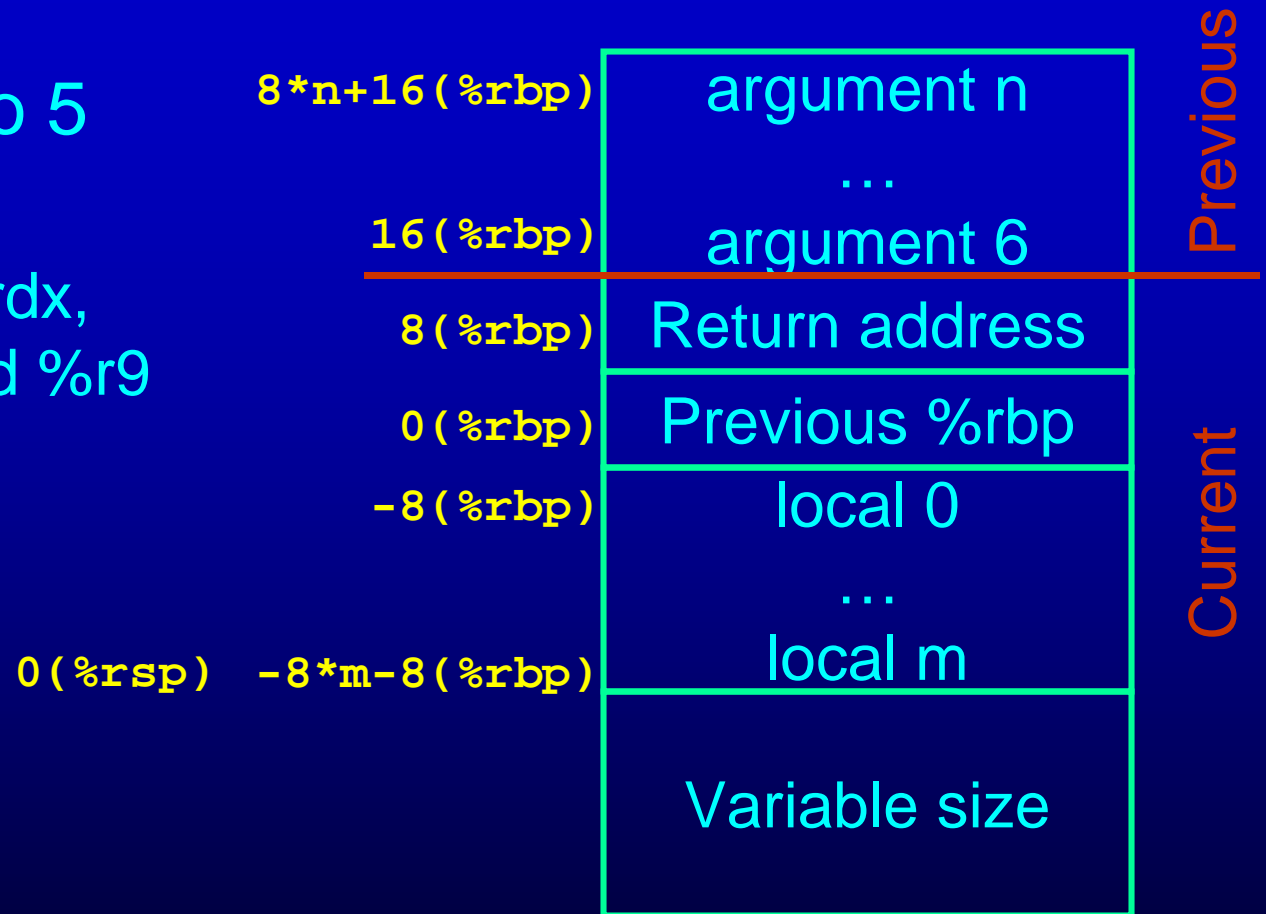
- Stack frames (activation records)
- Calling convention

# Registers

- What to do with live registers across a procedure call?
  - Caller Saved
  - Callee Saved
    - %rsp, %rbp

# The Stack

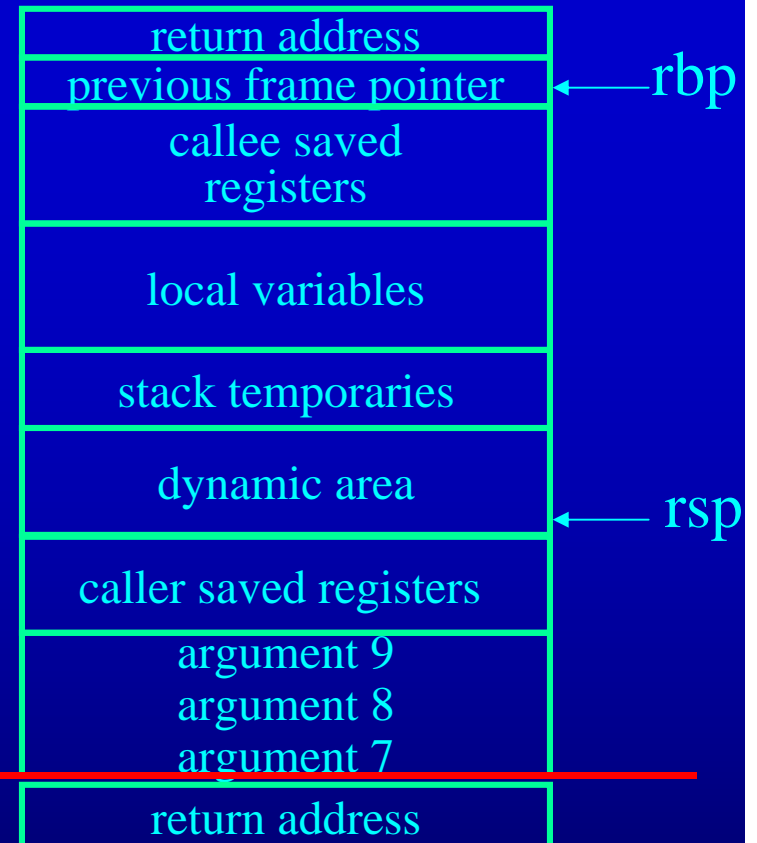
- Arguments 0 to 5 are in:
  - %rdi, %rsi, %rdx, %rcx, %r8 and %r9



# Stack

- Calling: Caller
  - Assume %rcx is live and is caller save
  - Call foo(A, B, C, D, E, F, G, H, I)
    - A to I are at -8(%rbp) to -72(%rbp)

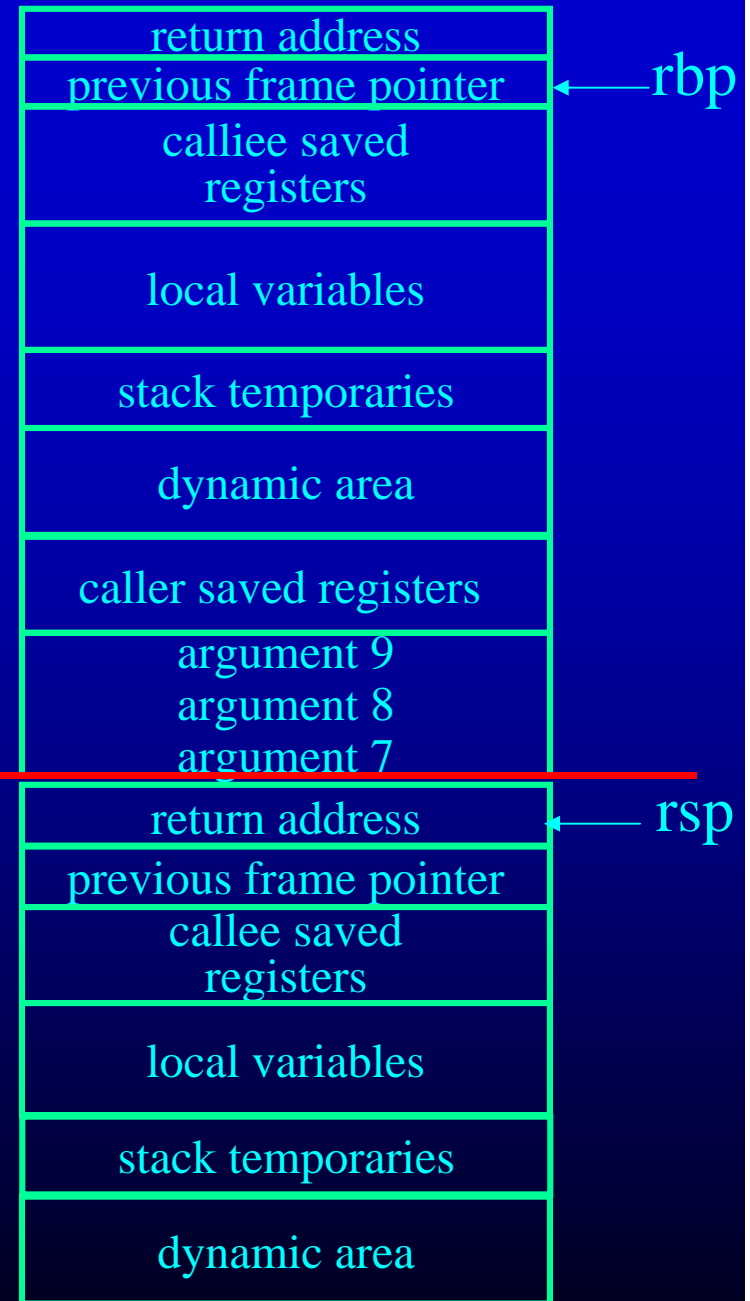
```
push    %rcx
push    -72(%rbp)
push    -64(%rbp)
push    -56(%rbp)
mov     -48(%rbp), %r9
mov     -40(%rbp), %r8
mov     -32(%rbp), %rcx
mov     -24(%rbp), %rdx
mov     -16(%rbp), %rsi
mov     -8(%rbp), %rdi
call   foo
```



# Stack

- Calling: Callee
  - Assume %rbx is used in the function and is callee save
  - Assume 40 bytes are required for locals

```
foo:
    push    %rbp
    enter  $48, $0
    mov     %rsp, %rbp
    sub    $48, %rsp
    mov    %rbx, -8(%rbp)
```

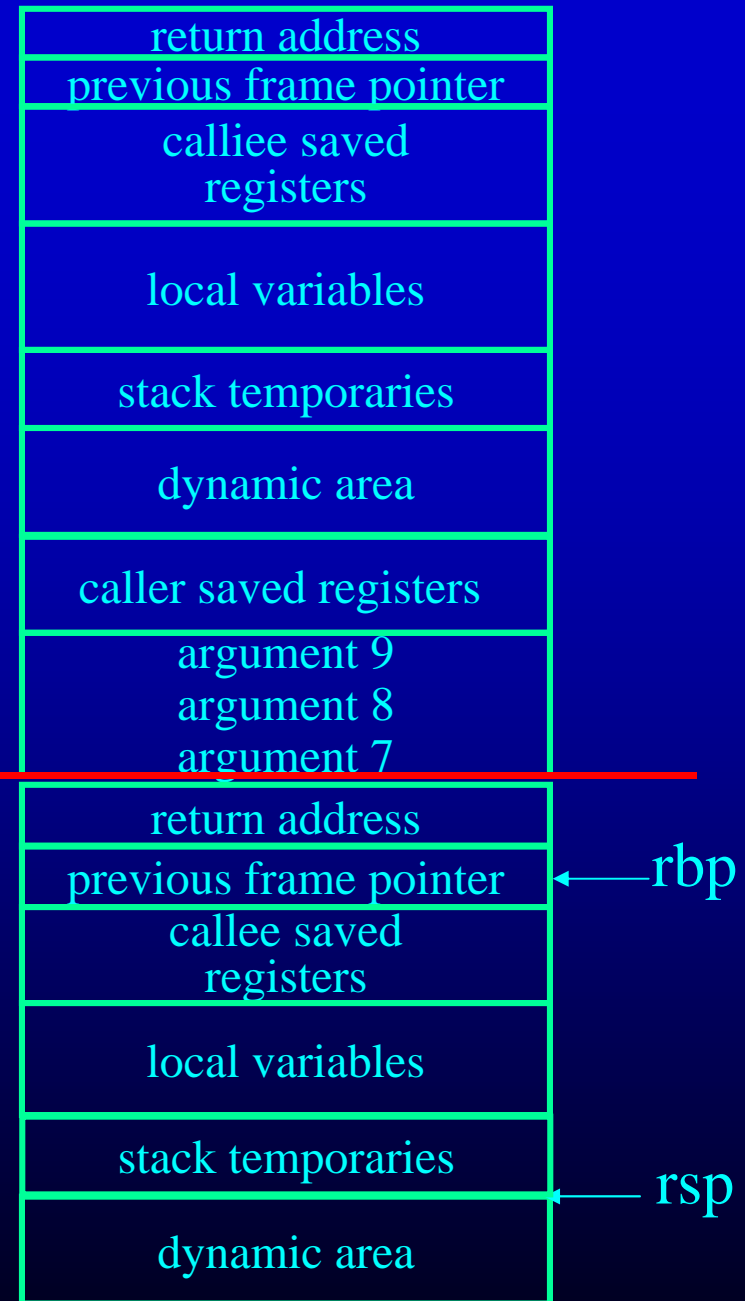


# Stack

- Arguments
- Call foo(A, B, C, D, E, F, G, H, I)
  - Passed in by pushing before the call

```
push    -72(%rbp)
push    -64(%rbp)
push    -56(%rbp)
mov     -48(%rbp), %r9
mov     -40(%rbp), %r8
mov     -32(%rbp), %rcx
mov     -24(%rbp), %rdx
mov     -16(%rbp), %rsi
mov     -8(%rbp), %rdi
call    foo
```

- I would move all args to current stack frame for now.

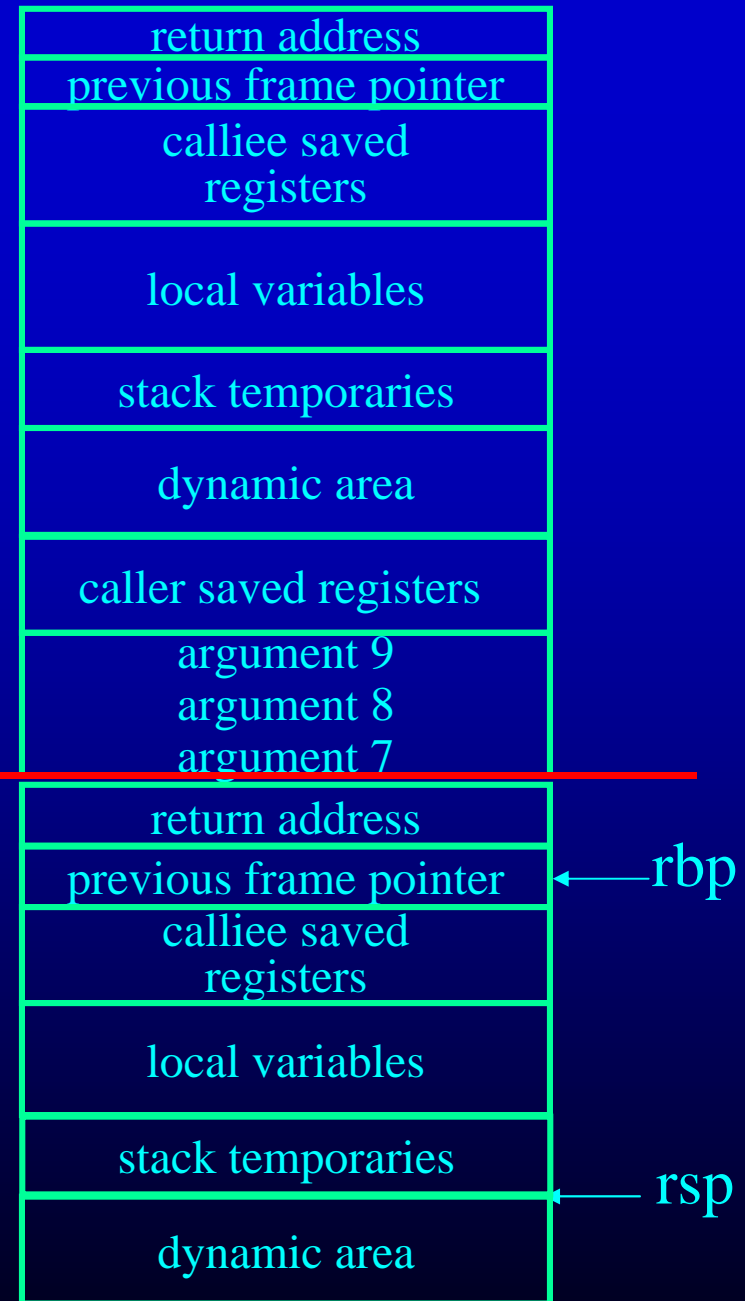


# Stack

- Returning Callee
  - Assume the return value is the first temporary
  - Restore the caller saved register
  - Put the return value in %rax
  - Tear-down the call stack

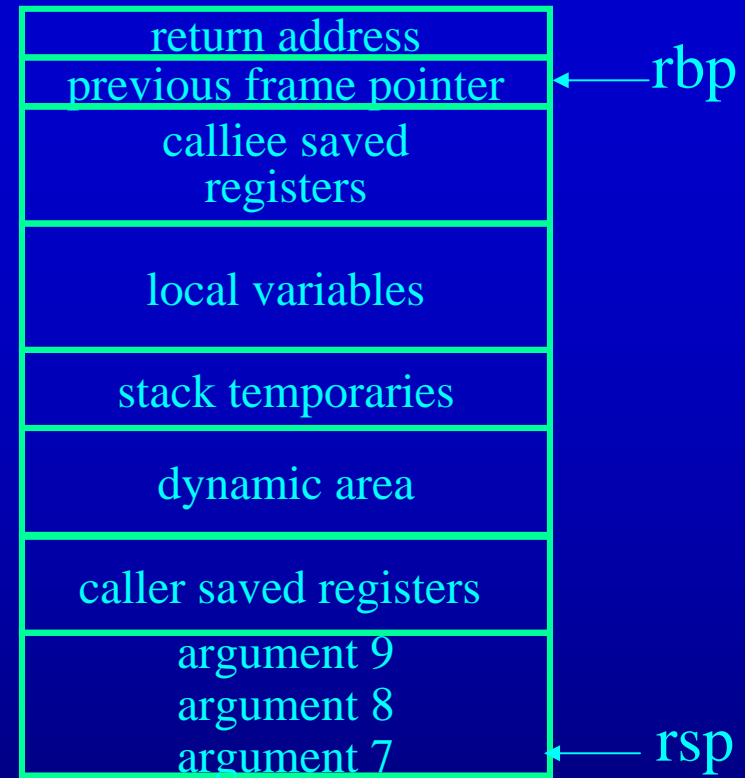
```
mov    -8(%rbp), %rbx
mov    -16(%rbp), %rax
mov    %rbp, %rsp
pop    %rbp
ret
```

**leave**



# Stack

- Returning Caller
  - Assume the return value goes to the first temporary
  - Restore the stack to reclaim the argument space
  - Restore the caller save registers
  - Save the return value



```
call    foo
add     $24, %rsp
pop     %rcx
mov     %rax, 8(%rbp)
...
```

# Reusing Temporaries

- You can allocate a temporary for each expression
- You can reuse temporaries very simply
- Ex:
  - eval E1 into T1
  - eval E2 into T2
  - $T3 = T1 + T2$
- After T3 is assigned, do we need T1 and T2?

# Reusing Temporaries

- Simple stack algorithm:
  - Keep a count for temporaries  $c$  (init to 0)
    - create a temporary location named  $T_c$
    - each  $T_c$  is a different location on the stack
    - $T_c$  is reused!
  - While traversing IR
    - Whenever a temporary name is used as an operand, decrement  $c$  by 1
    - Whenever a temporary name is generated use  $T_c$  and increase  $c$  by 1

# Reusing Temporaries Example

$$x = 1 * 2 + 3 * 4 - 5 * 6$$

Statement	Value of T after statement (0 at start)
$T0 = 1 * 2$	1
$T1 = 3 * 4$	2
$T0 = T0 + T1$	1
$T1 = 5 * 6$	2
$T0 = T0 - T1$	1
$X = T0$	0

# Hints

- Don't worry about machine portability
  - I like flat low-level IRs.
    - 2 address code looks nice+
    - $\text{operand}_1 \text{ operation} = \text{operand}_2$
- Control Flow Analysis (conversion to control flow graph) can be easily done on 2 address code.