# Lesson 4
# Machine Prog 2

ICS Seminar #9
张龄心
Oct 11, 2023

# Content

- Procedure: Explanation
- The Run-Time Stack
- Control Transfer
- Data Transfer
- Local Storage on the Stack
- Local Storage in Registers
- Recursive Procedures
- Conclusion: What happens when P calls Q?

# Procedure: Explanation

**What happens when procedure P calls procedure Q?**

- **Passing Control**

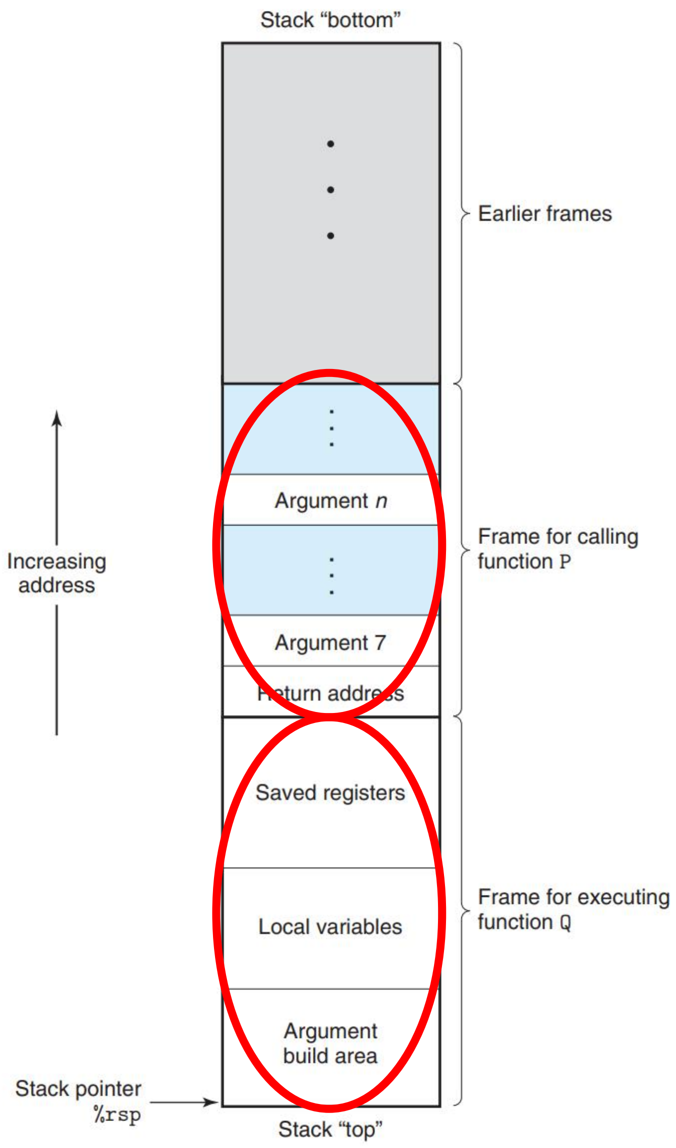
  PC (program counter, %rip) should be set to:

  - the starting address of Q when P calls Q
  - the next instruction of "call Q" in P when Q returns back to P
- **Passing Data**
  - P passes argument(s) to Q, and Q returns a value to P
- **Allocating & Deallocating Memory**
  - Q may allocate space for local variables etc.
  - The storage should be freed before Q returns

# The Run-Time Stack

## Concept & Differentiation

- **Stack**: Part of memory.
    - Every byte on the stack gets an 8-byte address.
- **Registers(Regs)**: Independent parts **outsides memory**.
    - Registers have no addresses.
    - x86 Registers are only ever addressed by name.
- **Stack Frame**: Part of stack. A stack has several frames.
    - **Saved Registers** (Callee-saved registers)
    - **Local Variables**
    - **Argument Build Area**



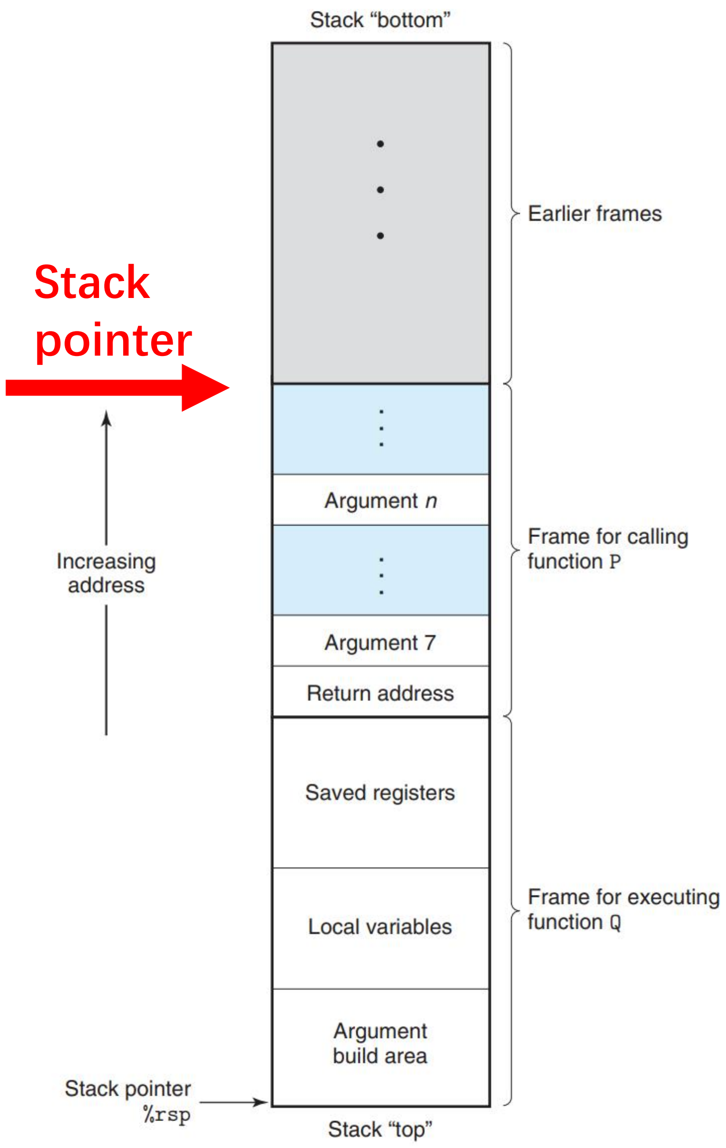
# The Run-Time Stack

## Allocating & deallocating space

- by decreasing & increasing the stack pointer(%rsp)
    - To allocate space for data, %rsp decreases
    - To deallocate space, %rsp increases

## Discipline: Last-in, First-out -> call/return machanism

- When Q executes, **previously called procedures are suspended.**
- When Q returns, %rsp increases, and stack frame of Q is freed.

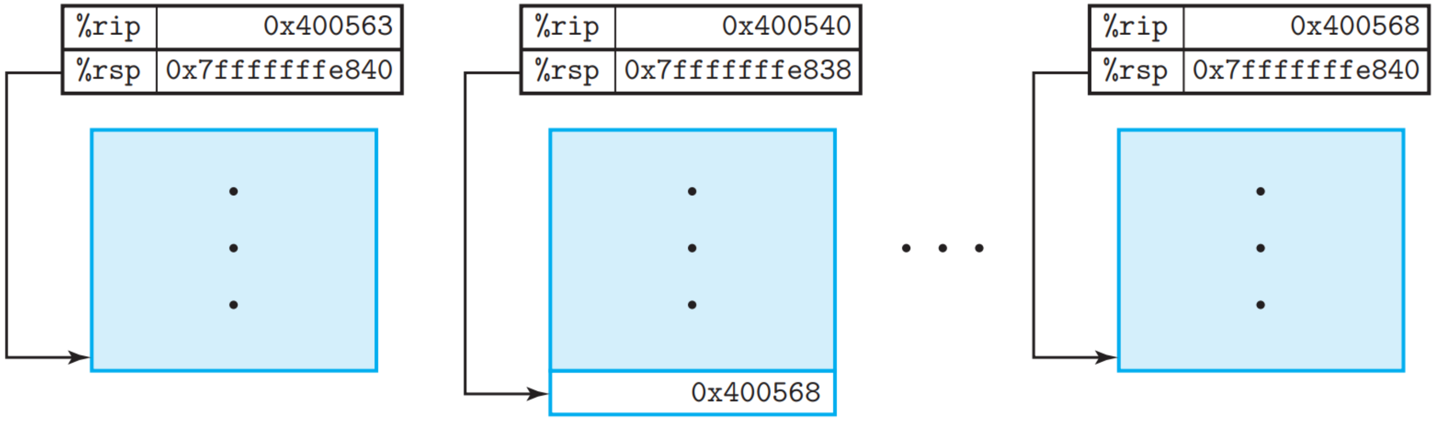


# Control Transfer: How P calls Q?

**When PC reads "call Q" in P, the instruction**

- **pushes the return address onto the stack**
  - Return address: the next instruction of "call Q" in P
- **sets PC to the beginning of Q**

(Note: "callq" &" retq" are used in .asm generated by **OBJDUMP**)




(a) Executing `call` (b) After `call` (c) After `ret`

# Control Transfer: How P calls Q?

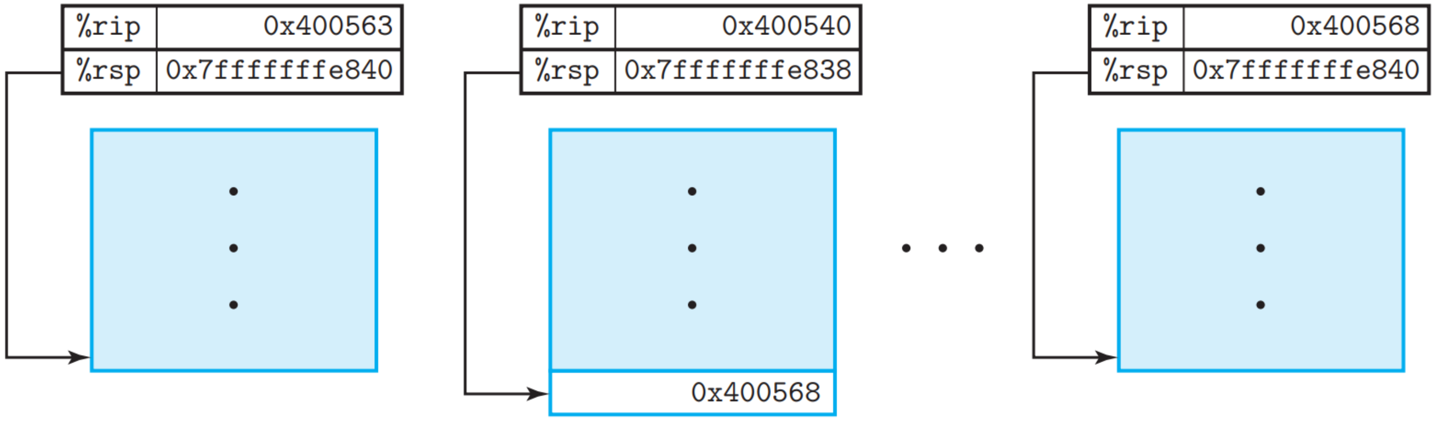
**When PC reads "ret" in Q, the instruction**

- **pops the return address off the stack**
- **sets PC to the return address**

And the execution of P is resumed.

*Note: **direct/indirect calls**

| Instruction | | Description |
|---|---|---|
| `call` | *Label* | Procedure call |
| `call` | **Operand* | Procedure call |
| `ret` | | Return from call |




(a) Executing `call` (b) After `call` (c) After `ret`

# Data Transfer: Arguments & Return Value

**Procedure calls may need data transfer:**

- **Passing data as arguments via**
  - 6 specific registers
  - Argument build area in the frame
- **Returning a value via**
  - Register %rax

| Operand size (bits) | Argument number | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 64 | %rdi | %rsi | %rdx | %rcx | %r8 | %r9 |
| 32 | %edi | %esi | %edx | %ecx | %r8d | %r9d |
| 16 | %di | %si | %dx | %cx | %r8w | %r9w |
| 8 | %dil | %sil | %dl | %cl | %r8b | %r9b |

**Figure 3.28** **Registers for passing function arguments.** The registers are used in a specified order and named according to the argument sizes.


**Figure 3.2** **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

# Data Transfer: Arguments & Return Value

**Registers for arguments: “dsdc89”**

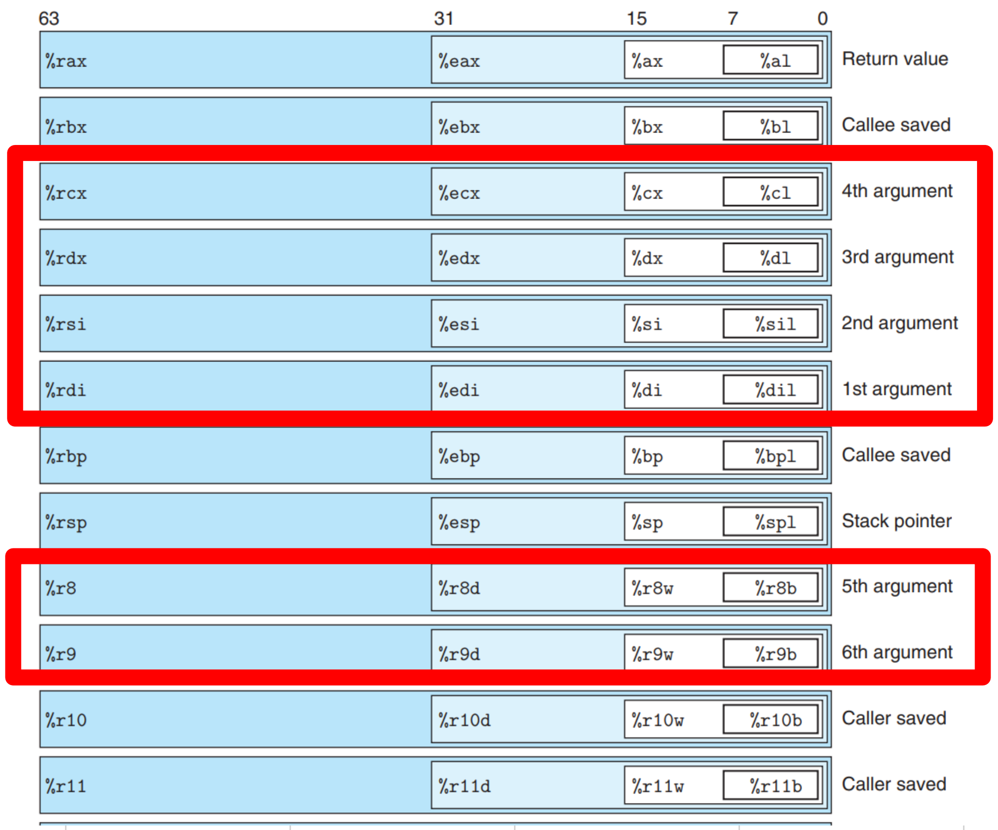
- **%rdi %rsi %rdx %rcx %r8 %r9**

**Note:**

- At most **6 arguments**
- Attention to **%di/%dil/%dl**
- **%r8b<%r8w<%r8d (not %r8l)**

| Operand size (bits) | Argument number | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 64 | %rdi | %rsi | %rdx | %rcx | %r8 | %r9 |
| 32 | %edi | %esi | %edx | %ecx | %r8d | %r9d |
| 16 | %di | %si | %dx | %cx | %r8w | %r9w |
| 8 | %dil | %sil | %dl | %cl | %r8b | %r9b |

Figure 3.28 **Registers for passing function arguments.** The registers are used in a specified order and named according to the argument sizes.




| Registers | %r8b | %r8w | %r8d | %r8 |
|---|---|---|---|---|
| Instructions | movb | movw | movl | movq |

Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

# Data Transfer: Arguments & Return Value

**If theres more than 6 arguments for Q:**

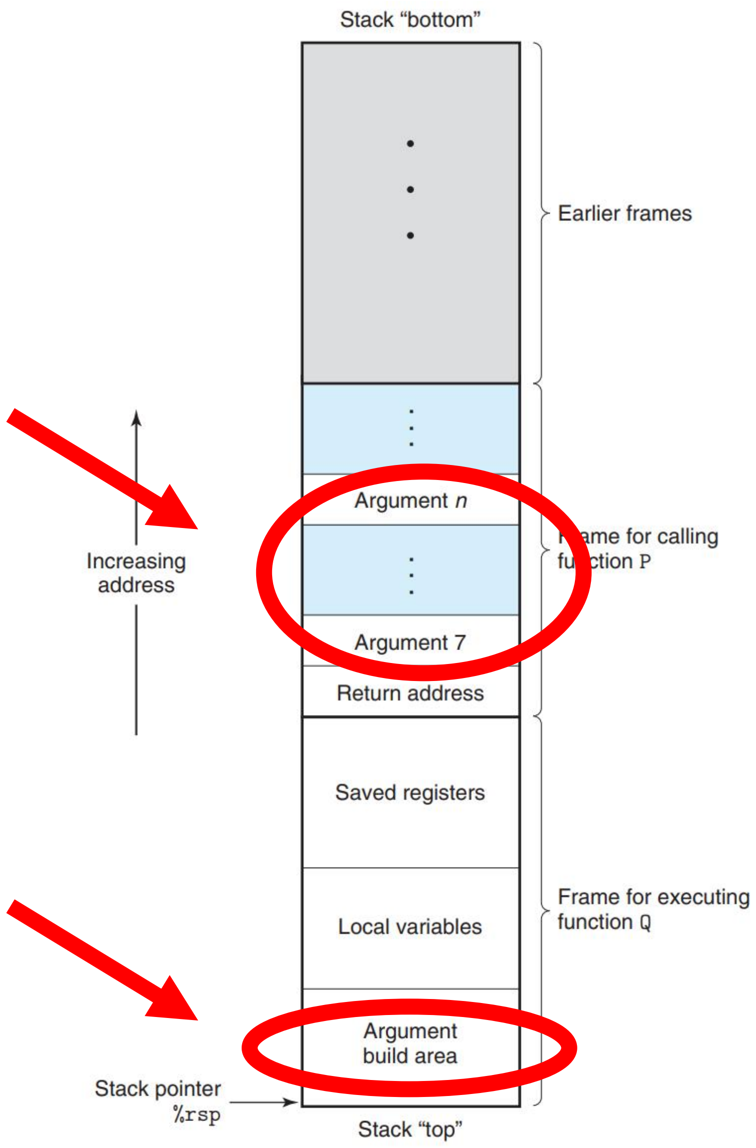
- **Arg 7~Arg n** will be stored in **a specific part in the stack frame of P**

  ->**Argument build area**

- Q can use these arguments in the stack by **k(%rsp),** with k as the offset

**About return value:**

- If a return value is needed, Q puts the value in **%rax (for floating-point: %ymm0/%xmm0)**

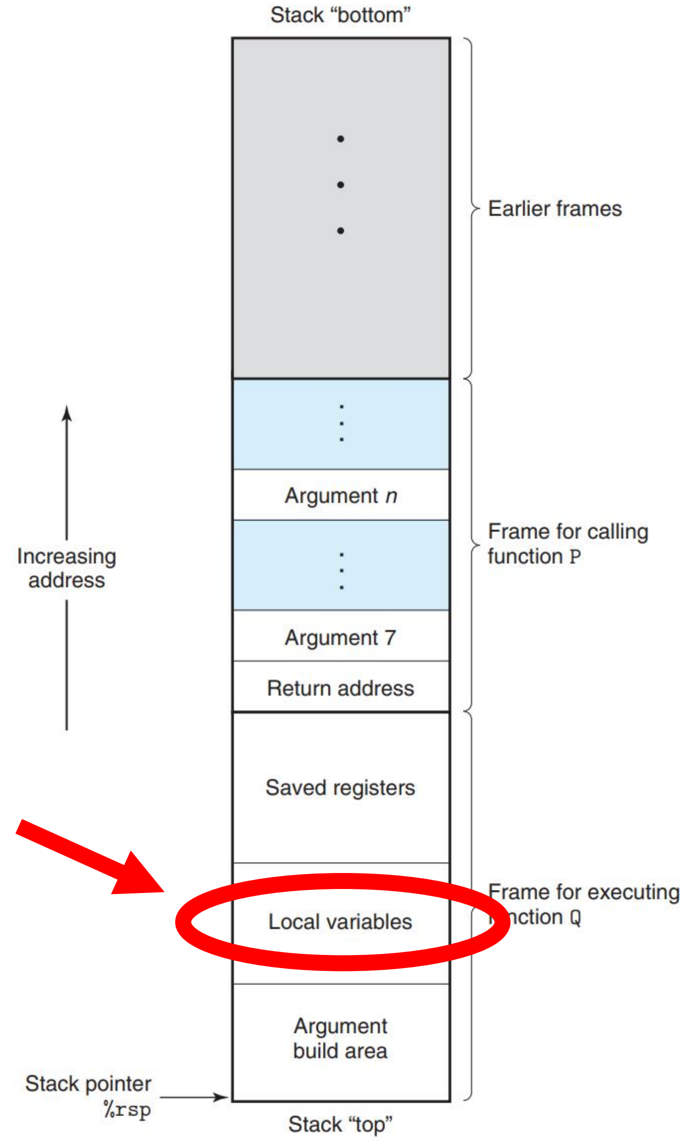


# Local Storage on the Stack

**Sometimes local data must be stored in memory:**

- **No enough registers** for all the local data
- **Address op "&"** is applied to a local variable
  - **Registers has no address**. x86 Registers are only ever addressed by name.
- **Arrays/Structures** as local variables
  - which must be **accessible by reference**

Local data stored on the stack can be visited via **k(%rsp)**, with k as the offset. (*%rbp)



# Local Storage in Registers

**Callee-saved Registers: 6**

- %rbx  %rbp
- %r12 ~ %r15

**Caller-saved Registers: 9**

- All the other registers **except %rsp**

**Exception: 1**

- %rsp (stack pointer)


Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

# Local Storage in Registers

**The Callee & the Caller are a set of relative concepts:**

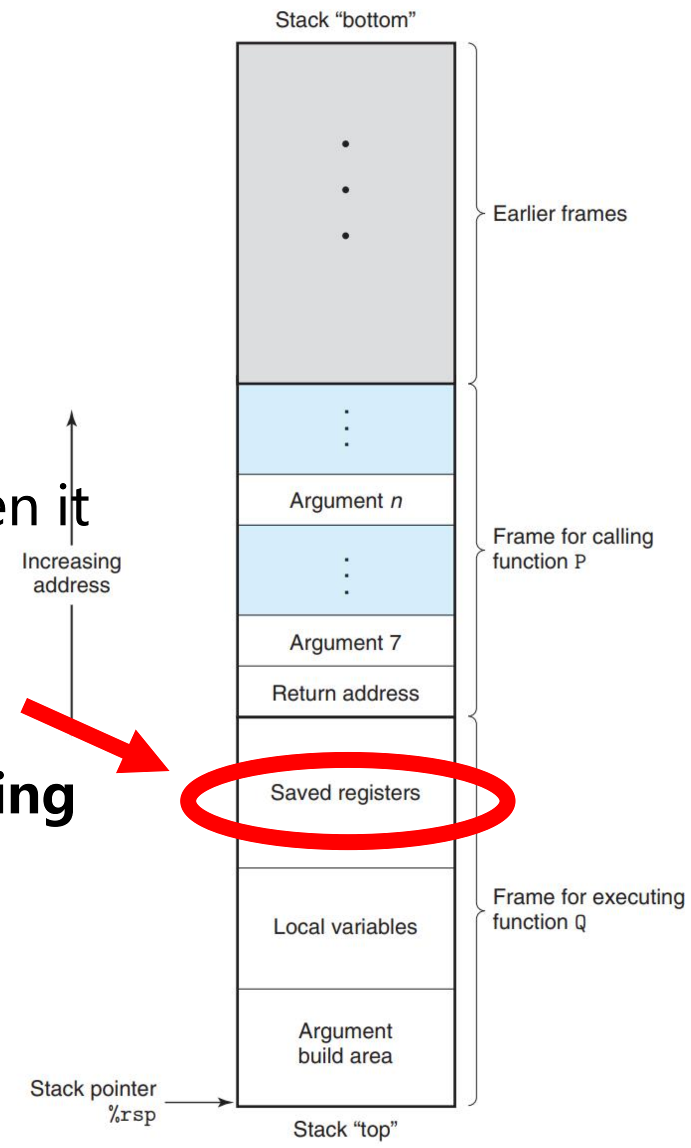
- If P calls Q, P is the Caller and Q is the Callee
- If then Q calls T, Q is the Caller of T as well as the Callee of P

**Callee-saved Regs**

- Q(the Callee) is responsible to **keep the regs as they are** when it returns back to P by
  - **never changing them at all**
  - **pushing the values first and pop them before returning**
- P(the Caller) can save its data safely here

**Caller-saved Regs**

- Q can change them as it wants
- If P wants to keep the values, it should **push them onto its stack**



# Local Storage in Registers

**If P wants to save some local data and use them after it calls Q :**

- Sol 1. put them in **Callee-saved regs**
- Sol 2. push them in **its own stack frame**

For Q, **all the local data of Q will be lost after Q returns back to P.**


Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

# Recursive Procedures

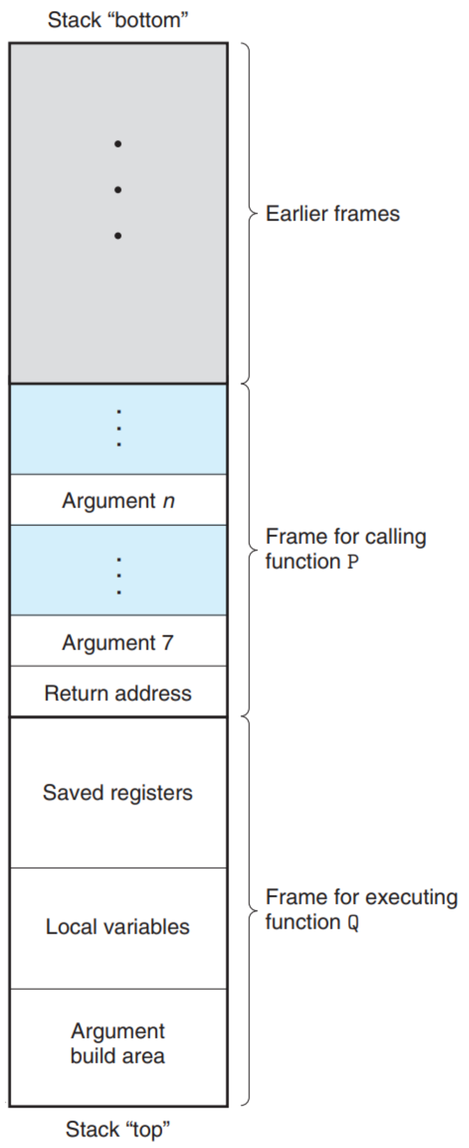
**When P calls itself recursively:**

- PC reads "call P" and **jump to the start address of P repeatedly**
- **New stack frames** are generated for new calls

**Until:**

- **Return condition satisfied**, returns all the way back
- **Stack overflow**

(And you may visit stackoverflow.com to find out why (x))



# Conclusion: What happens when P calls Q?

- **Preparation**: n arguments for Q
  - n≤6 -> stored in regs in order ( “dsdc89” )
  - n>6 -> Arg 1~6 in regs, Arg 7~n in stack (argument build area)

- **Call**: When PC reads “call Q” in P, the instruction
  - pushes the return address onto the stack (the next instruction of “call Q” in P)
  - sets PC to the beginning of Q

# 定长数组vs变长数组

- 定长数组:
  - 编译器能够作出优化
  - 尽可能避免开销较大的乘法

- 变长数组
  - 由于长度不确定，单个索引时使用imulq -> 不可避免的性能损失
  - 但是，规律性的访问仍能被优化

# Conclusion: What happens when P calls Q?

- **Execution**: In procedure Q
  - To use the Callee-saved regs, Q has to push the data onto its stack
  - P is suspended, and what happens in Q has nothing to do with P

    ->Package
  - The return value (if any) is stored in %rax/%ymm0

- **Return**: when PC reads "ret" in Q, the instruction
  - pops the return address off the stack
  - sets PC to the return address

# 复合数据类型和对齐

- struct: 一段连续的区域内, 按顺序存放不同的类型
- union: 同一段数据(字节), 多种解读方式

- 对齐:
  - **所有**K字节大小的对象, 必须k字节对齐(起始地址为k的倍数)
    注意: 记得确保所有struct内部都满足对齐条件
  - 对x86机器，处理未对齐的数据仍可正常运行，只是效率较低
    *对于其它一些机器，处理未对齐的数据可能导致内存错误
  - 栈帧需要8字节对齐
    - *某些标准下, 可能是16字节对齐

# 例题

1、假定静态int型二维数组a和指针数组pa的声明如下：

```
static int a[4][4]={{3, 8, -2, 6}, {2, 1, -5, 3 }, {1, 18, 4, 10},{4, -2, 0, 8}};
static int *pa[4]={a[0], a[1], a[2], a[3]};
```

若a的首地址为0x601080，则&pa[0]和pa[1]分别是:

A. 0x6010c0、0x601090

B. 0x6010e0、0x601090

C. 0x6010c0、0x6010a0

D. 0x6010e0、0x6010a0

# 例题

1、假定静态int型二维数组a和指针数组pa的声明如下:

```
static int a[4][4]={{3, 8, -2, 6}, {2, 1, -5, 3 }, {1, 18, 4, 10},{4, -2, 0, 8}};
static int *pa[4]={a[0], a[1], a[2], a[3]};
```

若a的首地址为0x601080，则&pa[0]和pa[1]分别是:

A. 0x6010c0、0x601090

B. 0x6010e0、0x601090

C. 0x6010c0、0x6010a0

D. 0x6010e0、0x6010a0

A

# 例题

2、假设结构体类型student_info的声明如下：

C

Copy Caption

```c
struct student_info {
char id[8];
char name[16];
unsigned zip;
char address[50];
char phone[20];
}x;
```

若x的首地址在`%rdx`中，则`unsigned xzip=x.zip;`所对应的汇编指令为:

A. `movl 0x24(%rdx), %eax`

B. `movl 0x18(%rdx), %eax`

C. `leaq 0x24(%rdx), %rax`

D. `leaq 0x18(%rdx), %rax`

# 例题

2、假设结构体类型student_info的声明如下：

C

```c
struct student_info {
char id[8];
char name[16];
unsigned zip;
char address[50];
char phone[20];
}x;
```

若x的首地址在`%rdx`中，则`unsigned xzip=x.zip;`所对应的汇编指令为:

A. `movl 0x24(%rdx), %eax`

B. `movl 0x18(%rdx), %eax`

C. `leaq 0x24(%rdx), %rax`

D. `leaq 0x18(%rdx), %rax`

**B**

# 例题

14．在 x86-64、Linux 操作系统下有如下 C 定义：

```
struct A {
    char CC1[6];
    int II1;
    long LL1;
    char CC2[10];
    long LL2;
    int II2;
};
```

（1） sizeof(A) = ________字节。

（2） 将 A 重排后，令结构体尽可能小，那么得到的新的结构体大小为________字节。

# 例题

14．在 x86-64、Linux 操作系统下有如下 C 定义：

```
struct A {
    char CC1[6];
    int II1;
    long LL1;
    char CC2[10];
    long LL2;
    int II2;
};
```

(1) sizeof(A) = ________字节。

(2) 将 A 重排后，令结构体尽可能小，那么得到的新的结构体大小为________字节。

**56 40**

# 例题

3、有如下定义的结构，在x86-64下，下述结论中错误的是?

```
struct {
  char c;
  union {
    char vc;
    double value;
    int vi;
  } u;
  int i;
} sa;
```

A. `sizeof(sa) == 24`

B. `(&sa.i - &sa.u.vi) == 8`

C. `(&sa.u.vc - &sa.c) == 8`

D. 优化成员变量的顺序 可以做到 `sizeof(sa) == 16`

# 例题

3、有如下定义的结构，在x86-64下，下述结论中错误的是?

```
struct {
  char c;
  union {
    char vc;
    double value;
    int vi;
  } u;
  int i;
} sa;
```

A. `sizeof(sa) == 24`

B. `(&sa.i - &sa.u.vi) == 8`

C. `(&sa.u.vc - &sa.c) == 8`

D. 优化成员变量的顺序 可以做到 `sizeof(sa) == 16`

**B (大端/小端法下, union的位置问题)**

# 例题

(　　)4. 在下面的代码中,A和B是用#define定义的常数:

```
typedef struct {int x[A][B]; long y;} str1;
typedef struct {char array[B]; int t; short s[A]; long u;} str2;
void setVal(str1 *p, str2 *q) {
        long v1 = q->t;long v2 = q->u;
        p->y = v1+v2;
}
```

GCC为setVal产生下面的代码:

```
setVal:
movslq 8(%rsi), %rax
addq 32(%rsi), %rax
movq %rax, 184(%rdi)
ret
```

则A=____,B=____.

A. 8,6　　B. 10,8　　C. 10,5　　D. 9.5

# 例题

(　　)4. 在下面的代码中,A和B是用#define定义的常数:

```
typedef struct {int x[A][B]; long y;} str1;
typedef struct {char array[B]; int t; short s[A]; long u;} str2;
void setVal(str1 *p, str2 *q) {
        long v1 = q->t;long v2 = q->u;
        p->y = v1+v2;
}
```

GCC为setVal产生下面的代码:

```
setVal:
movslq 8(%rsi), %rax
addq 32(%rsi), %rax
movq %rax, 184(%rdi)
ret
```

则A=____,B=____.

A. 8,6　　B. 10,8　　C. 10,5　　D. 9.5

4. D(4 < B <= 8,5 < A <= 10,44 < A*B <= 46,解得 A=9 B=5)

# 例题

15．在 x86-64、LINUX 操作系统下，考虑如下的 C 定义：

```
typedef union {
    char c[7];
    short h;
} union_e;

typedef struct {
    char d[3];
    union_e u;
    int i;
} struct_e;

struct_e s;
```

回答如下问题：

56 40

(1) s.u.c 的首地址相对于 s 的首地址的偏移量是________字节。

(2) sizeof(union_e) = ________字节。

(3) s.i 的首地址相对于 s 的首地址的偏移量是________字节。

# Thank you!