# BUFFER OVERFLOW - BACKGROUND

CS-5156/CS-6056: SECURITY VULNERABILITY ASSESSMENT (SPRING 2025)

LECTURE 7

# Outline

- Vulnerability and Exploit
- Program memory structure
- Assembly Review
- Activation Records
- Buffer Overflow
- x86-64

# Outline

- Vulnerability and Exploit
- Program memory structure
- Assembly Review
- Activation Records
- Buffer Overflow
- x86-64

# Definitions

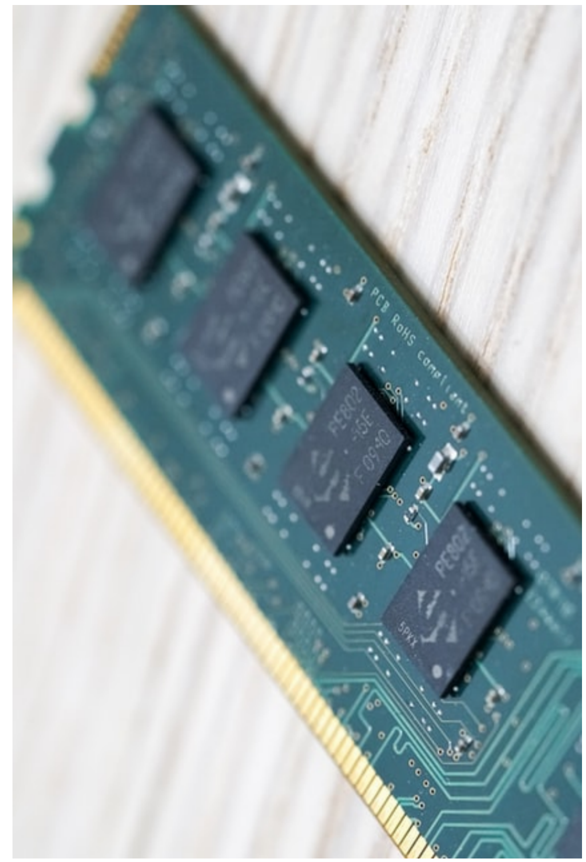
- Vulnerability
  - **Bug**/weakness/flaw in a **software**/system/network
- Exploit
  - **Software** *or* **set of commands** used by a **threat actor** to take advantage of a **vulnerability** in order to perform unauthorized actions within the system/network

# Outline

- Vulnerability and Exploit
- Program memory structure
- Assembly Review
- Activation Records
- Buffer Overflow
- x86-64

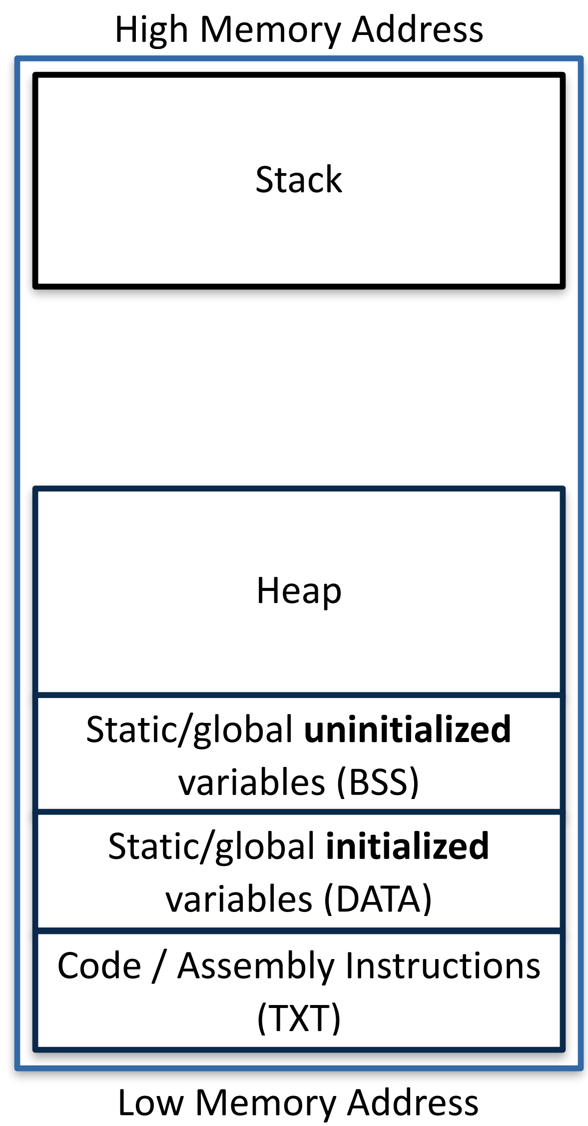
# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
void main() {
        bar();
        int *ptr;
        ptr = malloc(15 * sizeof(*ptr));
}
```



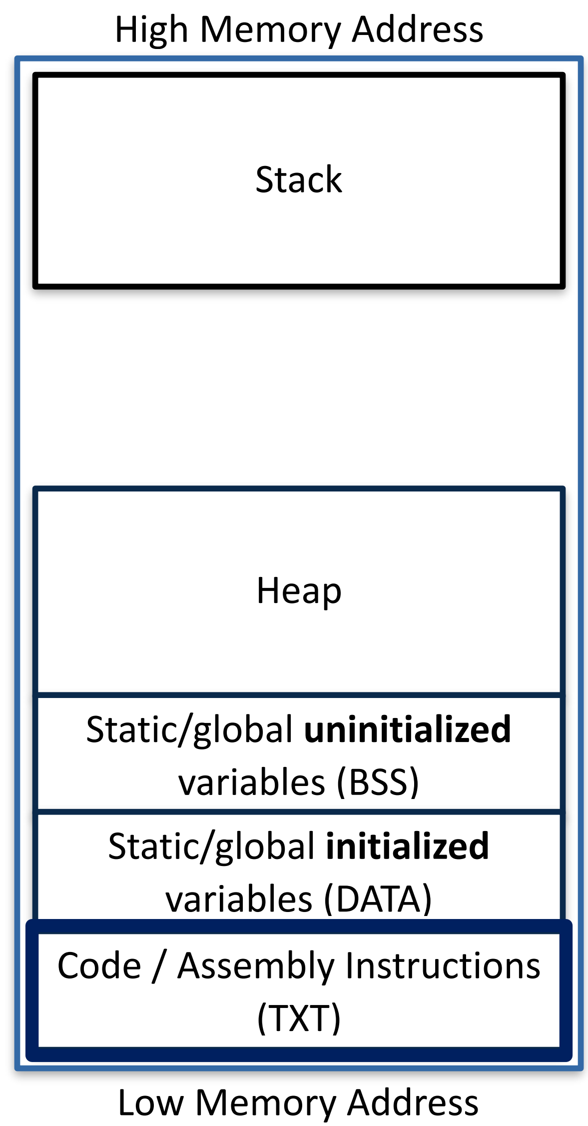
# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
      int localVariable1, 2, ….,N;
      return 0;
}
void bar() {
      foo(1, 5, 10, 20, …., 100);
}
void main() {
      bar();
      int *ptr;
      ptr = malloc(15 * sizeof(*ptr));
}
```



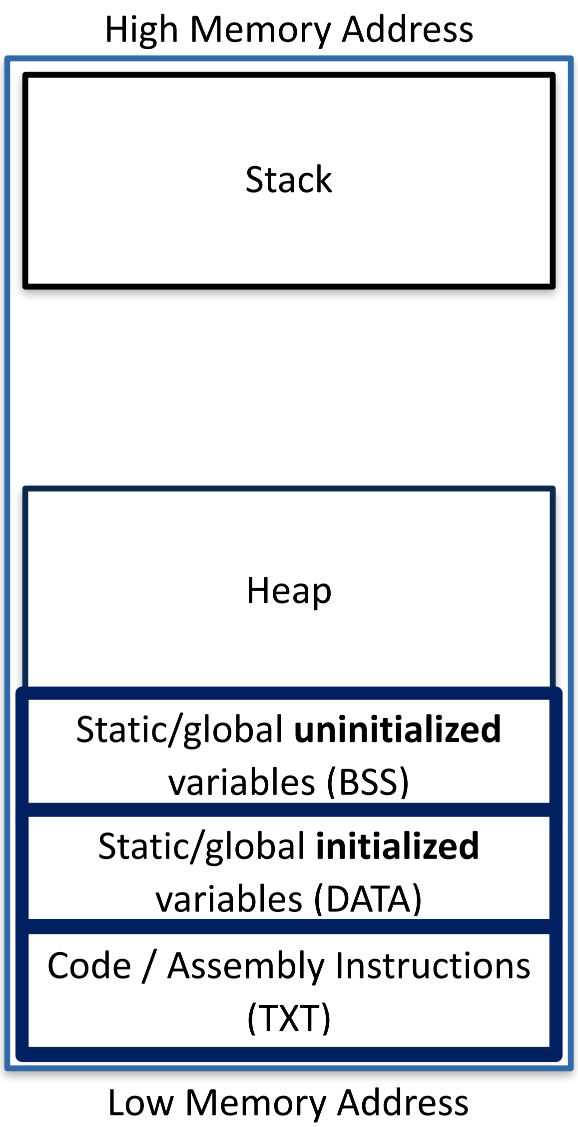
# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
void main() {
        bar();
        int *ptr;
        ptr = malloc(15 * sizeof(*ptr));
}
```



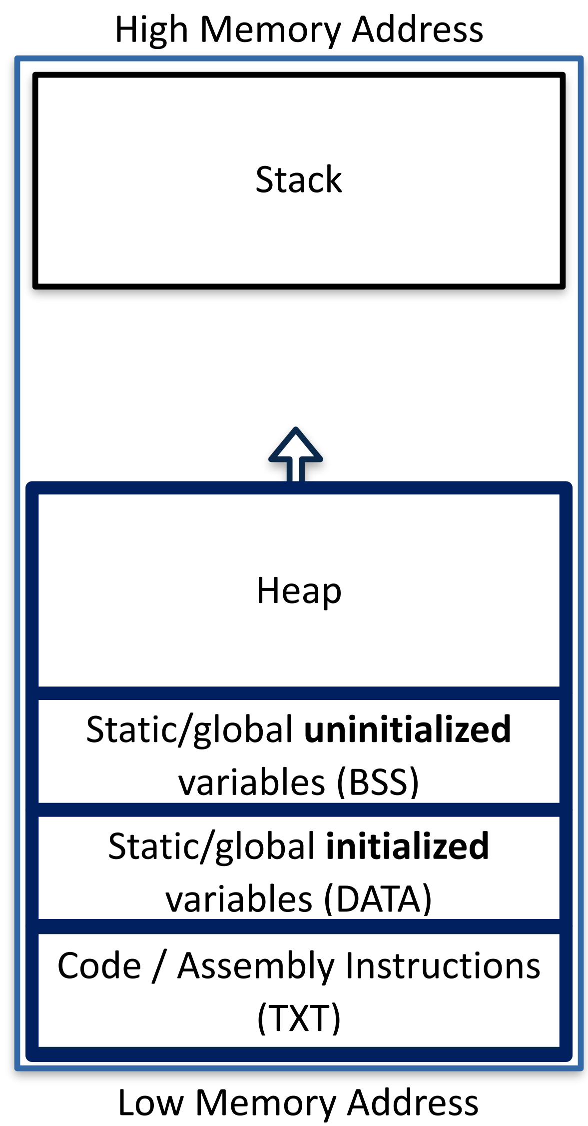
# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
    int localVariable1, 2, ….,N;
    return 0;
}
void bar() {
    foo(1, 5, 10, 20, …., 100);
}
void main() {
    bar();
    int *ptr;
    ptr = malloc(15 * sizeof(*ptr));
}
```



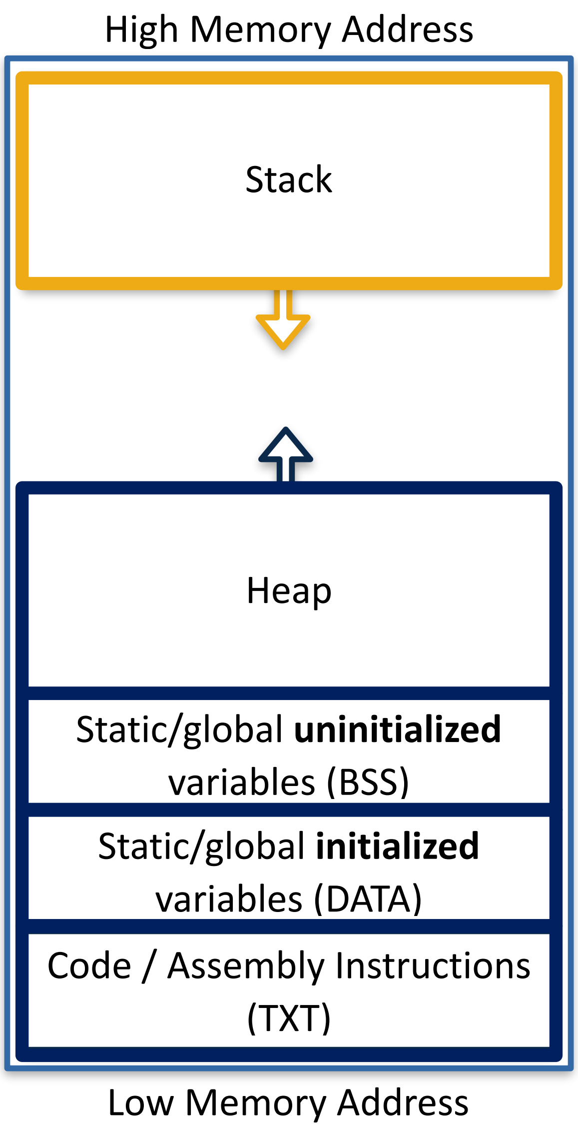
# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
void main() {
        bar();
        int *ptr;
        ptr = malloc(15 * sizeof(*ptr));
}
```

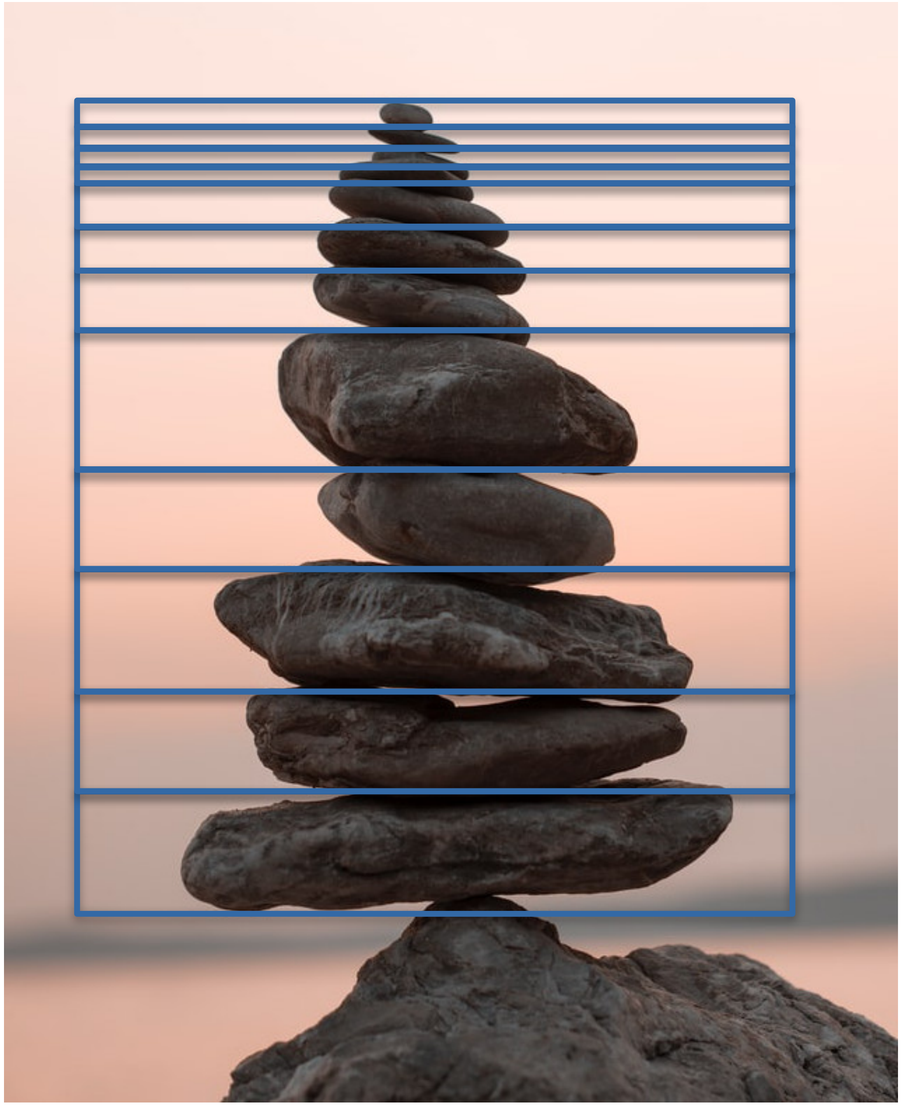


# Program Memory Structure

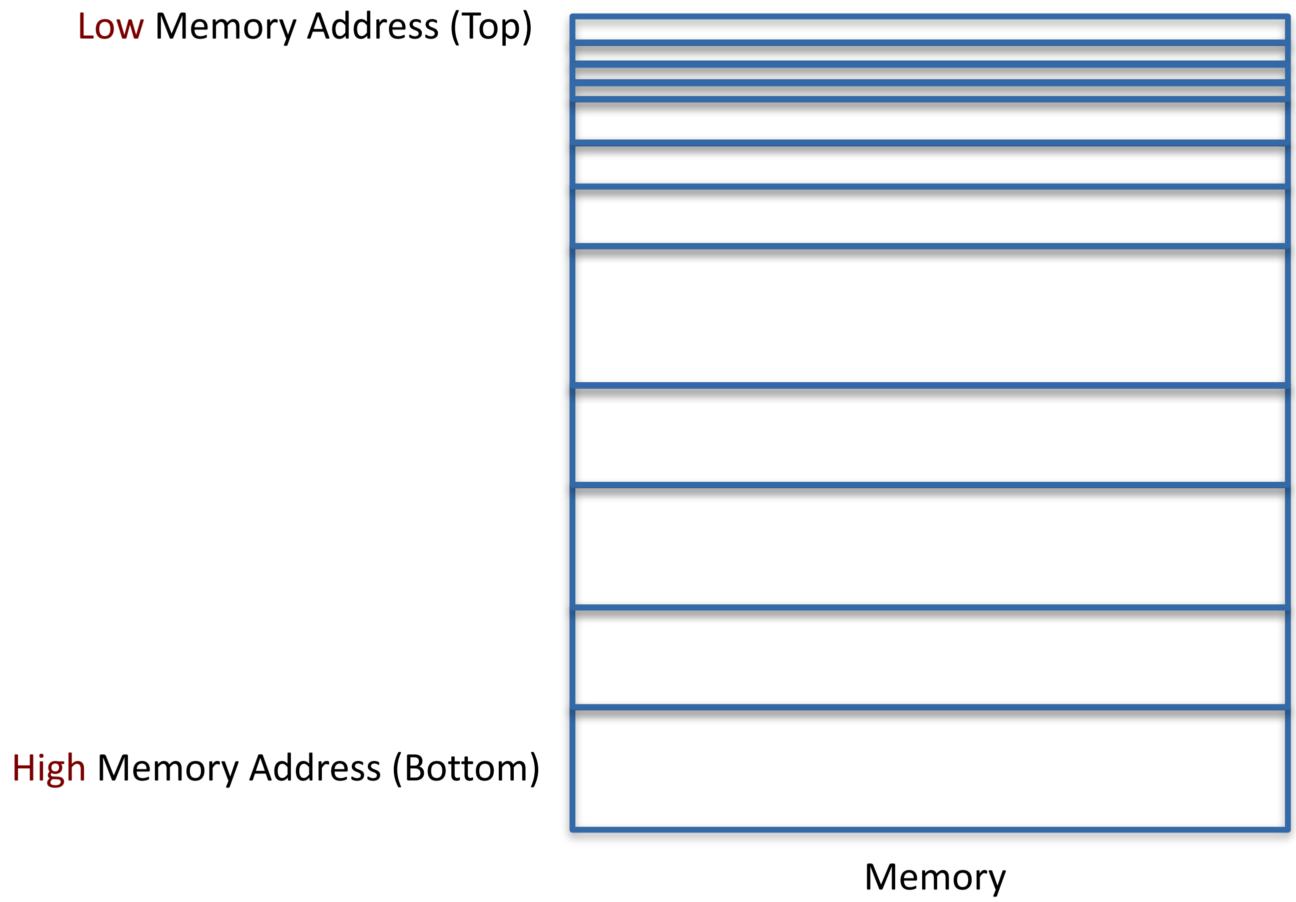
```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
      int localVariable1, 2, ….,N;
      return 0;
}
void bar() {
      foo(1, 5, 10, 20, …., 100);
}
void main() {
      bar();
      int *ptr;
      ptr = malloc(15 * sizeof(*ptr));
}
```



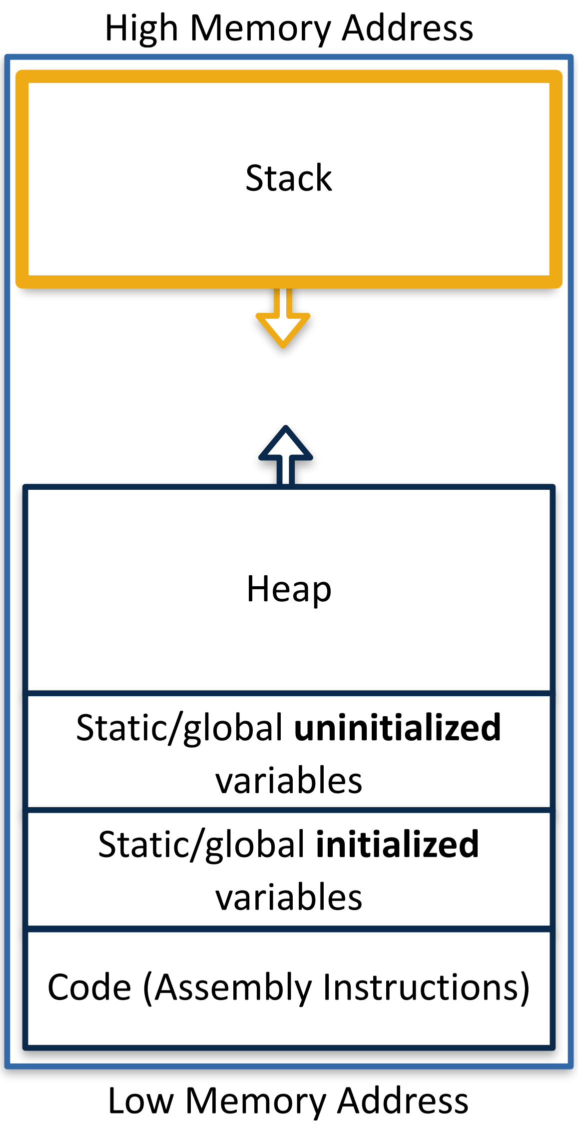
# Stack

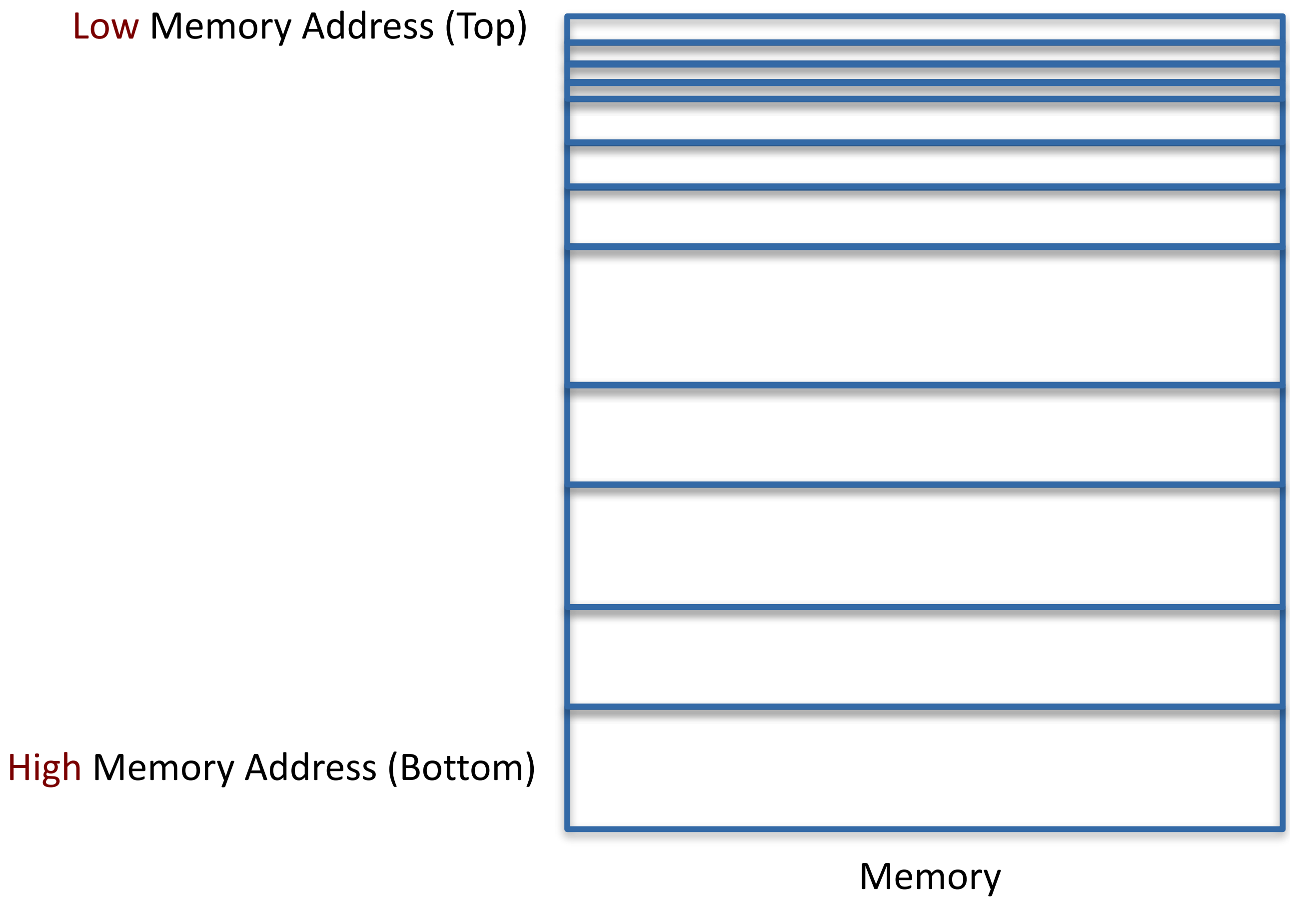


# Stack

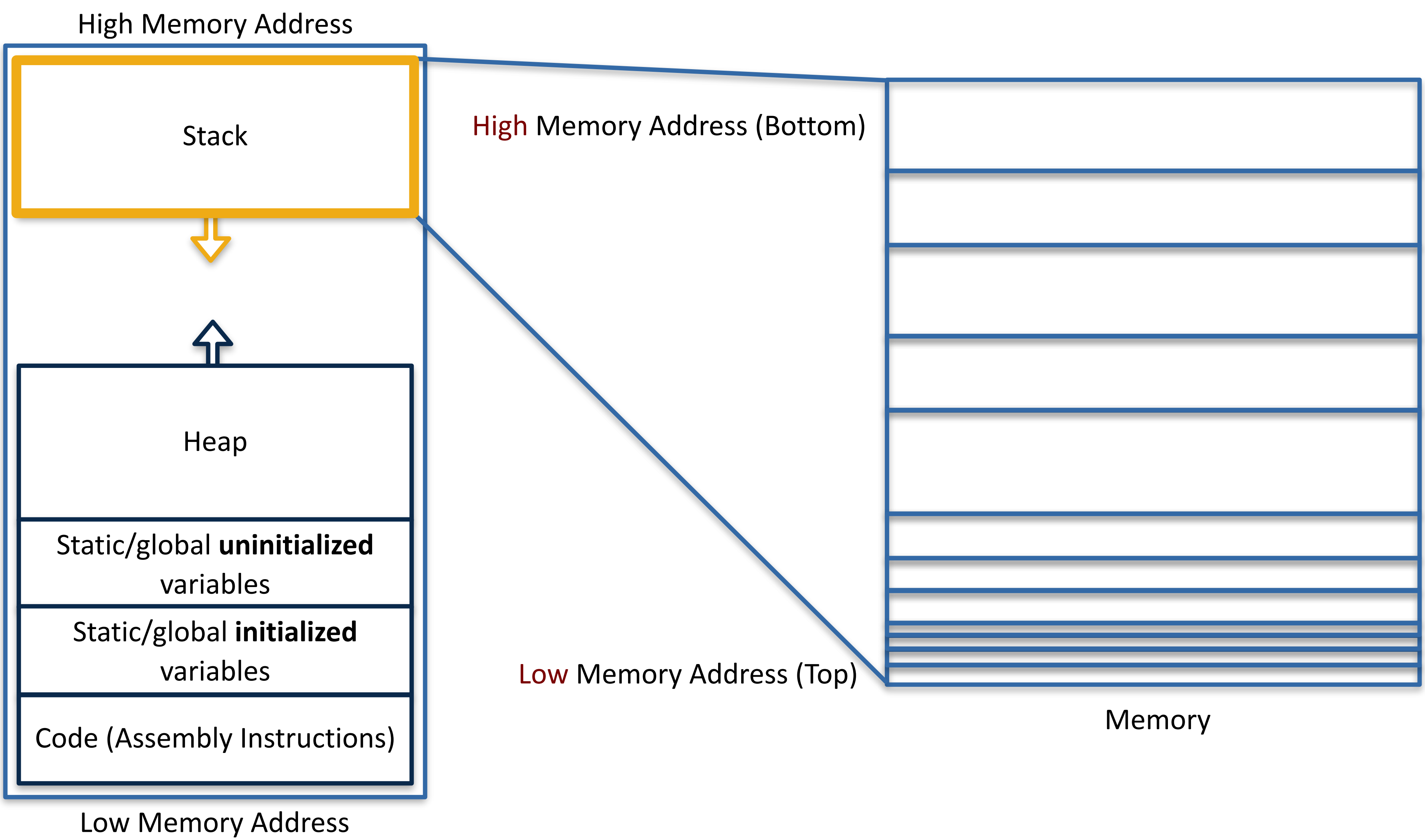


# Stack

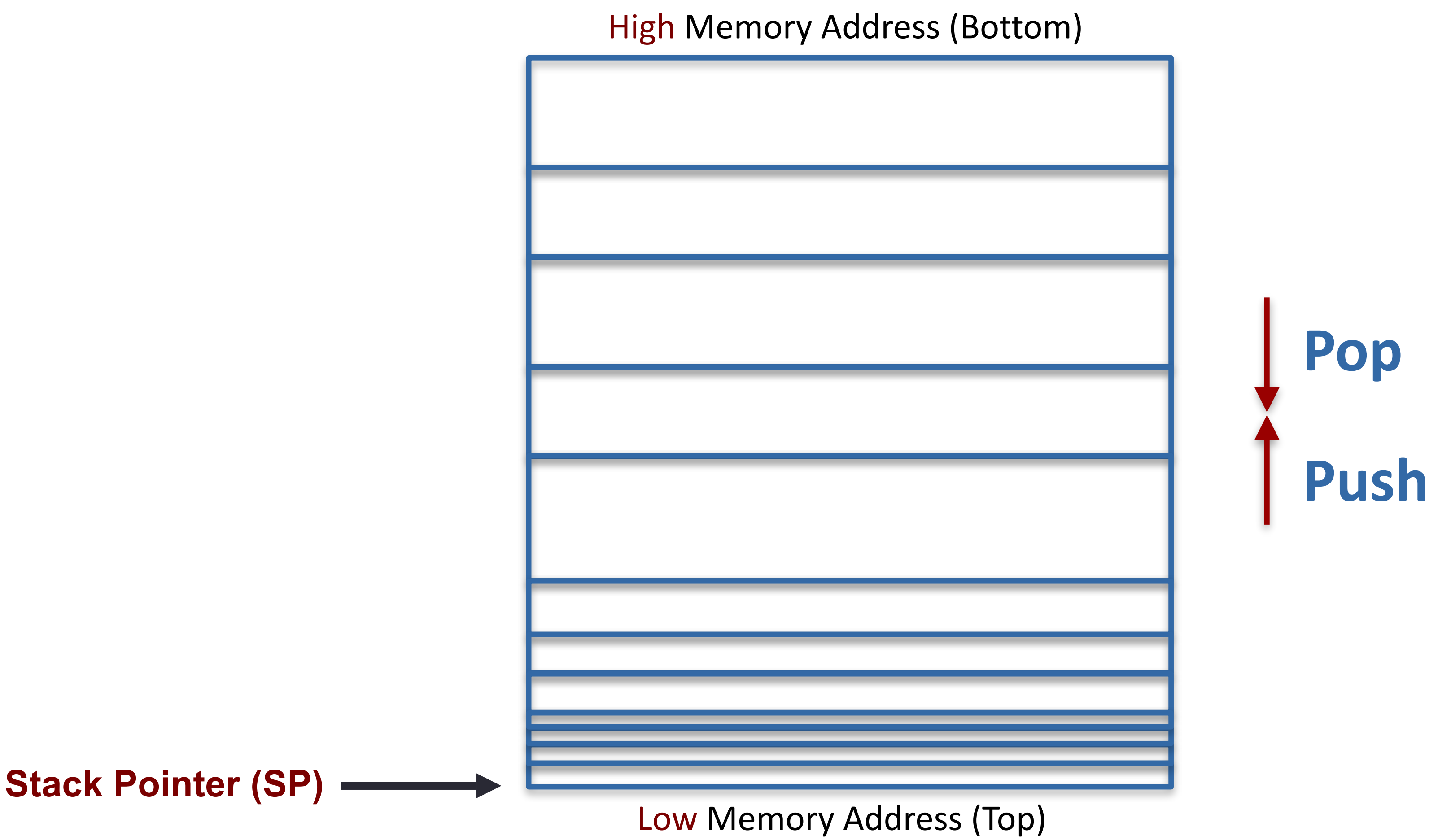

High Memory Address
Stack
Heap
Static/global **uninitialized** variables
Static/global **initialized** variables
Code (Assembly Instructions)
Low Memory Address


Low Memory Address (Top)
High Memory Address (Bottom)
Memory

# Stack


High Memory Address
Stack
Heap
Static/global **uninitialized** variables
Static/global **initialized** variables
Code (Assembly Instructions)
Low Memory Address
High Memory Address (Bottom)
Low Memory Address (Top)
Memory

# Stack


High Memory Address (Bottom)
Pop
Push
Stack Pointer (SP)
Low Memory Address (Top)

# Stack

**1 Byte = 8 bits**

**e.g., 11111100**

**1 Byte = 2 Hex**

**e.g, 0xFC**

**Each Byte is addressed using a 32-bit value**

High Memory Address (Bottom)

| | | | | Address |
|---|---|---|---|---|
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F050 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F04C |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F048 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F044 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F040 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F03C |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F038 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F034 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F030 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F02C |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F028 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F024 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F020 |

**Stack Pointer (SP)** →

Low Memory Address (Top)

# Outline

- Vulnerability and Exploit
- Program memory structure
- Assembly Review
- Activation Records
- Buffer Overflow
- x86-64

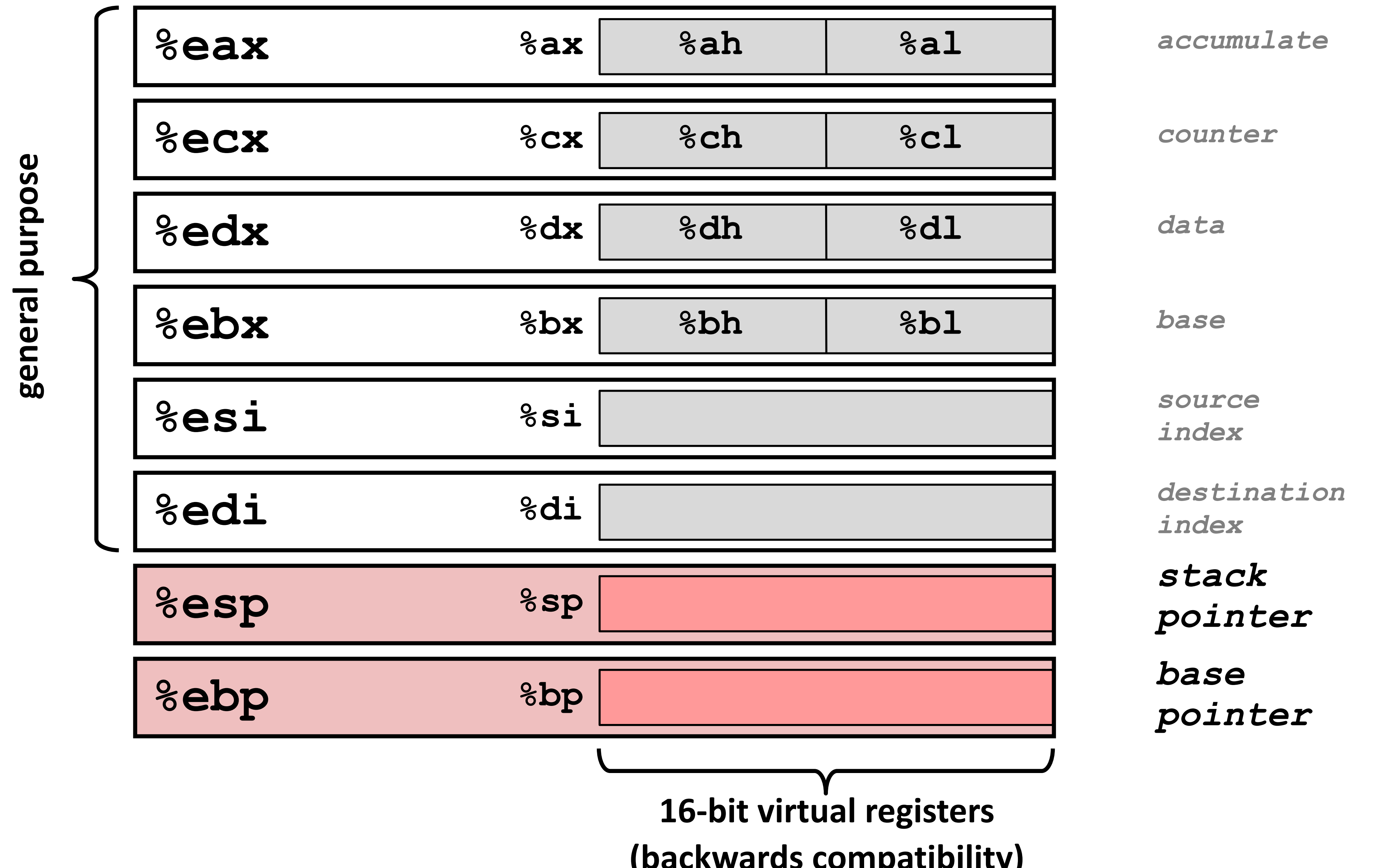
# x86-64 Registers

| | | | |
|---|---|---|---|
| `%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` |

- Backward Compatibility: Can reference ***low-order 4 bytes*** (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

# IA32 Registers

Origin
(mostly obsolete)



# Assembly Characteristics: Data Types

- **"Integer" data of 1 (char), 2 (short), 4 (int), or 8 (long, ptr) bytes**
  - Data values
  - Addresses (untyped pointers)
  - BYTE (1), **WORD** (2), **D**WORD "Double Word" (4), **Q**WORD "Quad Word"(8)
    - The **original 8086 16-bit** arch referred to the 16-bit data type as **word**
- **Floating point data of 4 (float), 8 (double)**
- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

# Assembly x86-64 Characteristics: Data Types

| C Declaration | Intel Data Type | Assembly Code Suffic | Size (bytes) |
|---|---|---|---|
| char | Byte | b | 1 |
| short | Word | w | 2 |
| int | **Double** word | l | 4 |
| long | **Quad** word | q | 8 |
| char * | Quad word | q | 8 |
| float | Single precision | s | 4 |
| double | Double precision | l | 8 |

- **Assembly Code suffix**: e.g., mov**b**, mov**w**, mov**l**, mov**q**
  - No ambiguity between int and double (both use **l** as suffix) since int and floating point have different instructions

# Stack Push/Pop Data Instructions (Revisited)

**Initially**

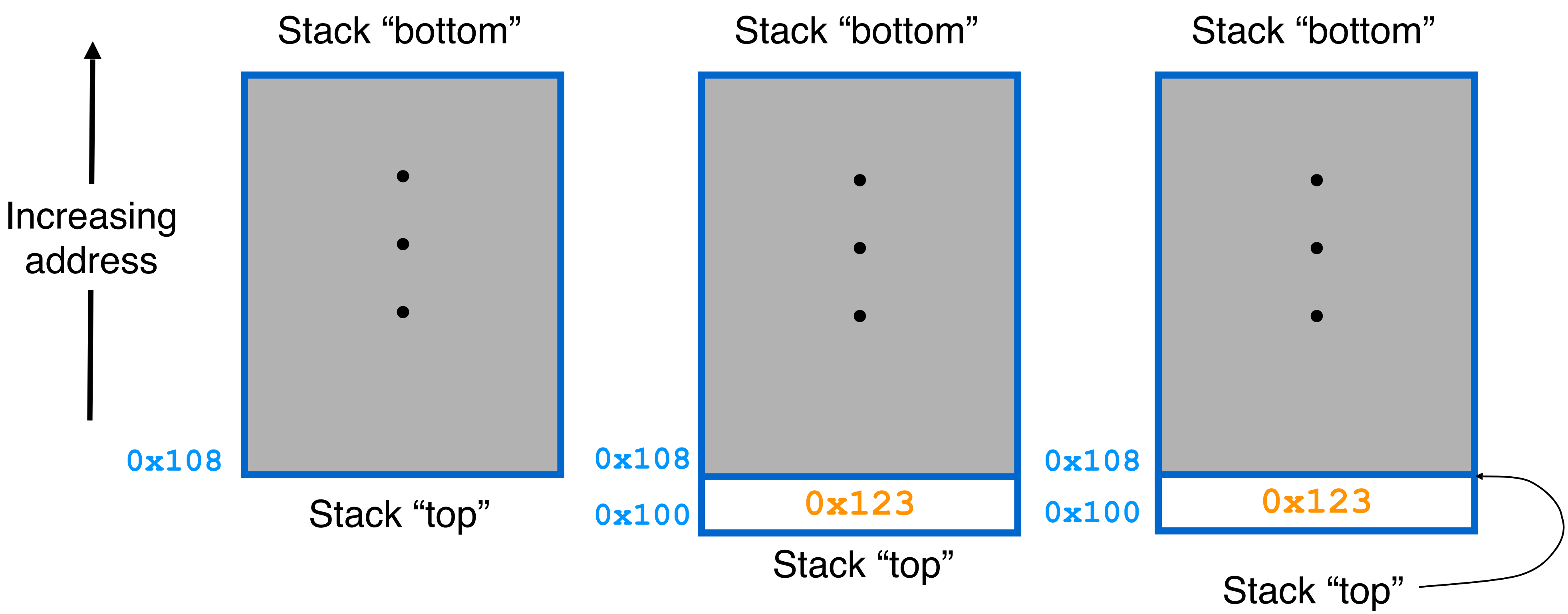
| | |
|---|---|
| `%rax` | `0x123` |
| `%rdx` | `0` |
| **`%rsp`** | `0x108` |

**`pushq %rax`**

| | |
|---|---|
| `%rax` | `0x123` |
| `%rdx` | `0` |
| `%rsp` | `0x100` |

**`popq %rdx`**

| | |
|---|---|
| `%rax` | `0x123` |
| `%rdx` | `0x123` |
| `%rsp` | `0x108` |




Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

# Compiling Into Assembly

**C Code (sum.c)**

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

**Generated x86-64 Assembly**

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

**Obtain with command**

```
gcc –Og –S sum.c
```

**Produces file** `sum.s`

***Warning***: **Will get very different results on different machines (Andrew Linux, Mac OS-X, …) due to different versions of gcc and different compiler settings.**

# What it really looks like

```
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size   sumstore, .-sumstore
```

**Things that look weird and are preceded by a '.' are generally directives (notes to the assembler, not translated to machine code ).**

```
sumstore:
    pushq    %rbx
    movq     %rdx, %rbx
    call     plus
    movq     %rax, (%rbx)
    popq     %rbx
    ret
```

# Object Code

## Code for `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- **Total of 14 bytes**
- **Each instruction 1, 3, or 5 bytes**
- **Starts at address `0x0400595`**

- **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

## Disassembled

```
0000000000400595 <sumstore>:
  400595:  53                 push   %rbx
  400596:  48 89 d3           mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff     callq  400590 <plus>
  40059e:  48 89 03           mov    %rax,(%rbx)
  4005a1:  5b                 pop    %rbx
  4005a2:  c3                 retq
```

- **Disassembler**

  `objdump -d sum`

  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces **approximate** rendition of **assembly** code
  - *Can be run on either `a.out` (complete executable) or `.o` file*

# Alternate Disassembly

## Disassembled from within gdb

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- **Within gdb Debugger**
  - Disassemble procedure

```
gdb sum
disassemble sumstore
```

# Alternate Disassembly

**Object Code**

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

**Disassembled from within gdb**

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- **Within gdb Debugger**
  - Disassemble procedure

```
gdb sum
disassemble sumstore
```

  - Examine the 14 bytes starting at `sumstore`

```
x/14xb sumstore
```

# Alternate Disassembly (Example)

```
linux> gcc -m32 –Og -o main *.c
linux> gdb main
```

## Disassembled from within gdb

```
gdb-peda$ disassemble multstore
Dump of assembler code for function multstore:
   0x000011fb <+0>:     endbr32
   0x000011ff <+4>:     push   DWORD PTR [esp+0x8]
   0x00001203 <+8>:     push   DWORD PTR [esp+0x8]
   0x00001207 <+12>:    call   0x11ed <mult2>
   0x0000120c <+17>:    add    esp,0x8
   0x0000120f <+20>:    mov    edx,DWORD PTR [esp+0xc]
   0x00001213 <+24>:    mov    DWORD PTR [edx],eax
   0x00001215 <+26>:    ret
End of assembler dump.
```

- Examine the **27 bytes** starting at **`multstore`**

**`x/27xb multstore`**

```
gdb-peda$ x/27xb multstore
0x11fb <multstore>:     0xf3  0x0f  0x1e  0xfb  0xff  0x74  0x24  0x08
0x1203 <multstore+8>:   0xff  0x74  0x24  0x08  0xe8  0xe1  0xff  0xff
0x120b <multstore+16>:  0xff  0x83  0xc4  0x08  0x8b  0x54  0x24  0x0c
0x1213 <multstore+24>:  0x89  0x02  0xc3
```

# Alternate Disassembly (Example)

```
linux> gcc -Og -o main *.c
linux> gdb main
```

## Disassembled from within gdb

```
gdb-peda$ disassemble multstore
Dump of assembler code for function multstore:
   0x0000000000001175 <+0>:     endbr64
   0x0000000000001179 <+4>:     push   rbx
   0x000000000000117a <+5>:     mov    rbx,rdx
   0x000000000000117d <+8>:     call   0x1169 <mult2>
   0x0000000000001182 <+13>:    mov    QWORD PTR [rbx],rax
   0x0000000000001185 <+16>:    pop    rbx
   0x0000000000001186 <+17>:    ret
End of assembler dump.
```

- Examine the **18 bytes** starting at **multstore**

  **x/18xb multstore**

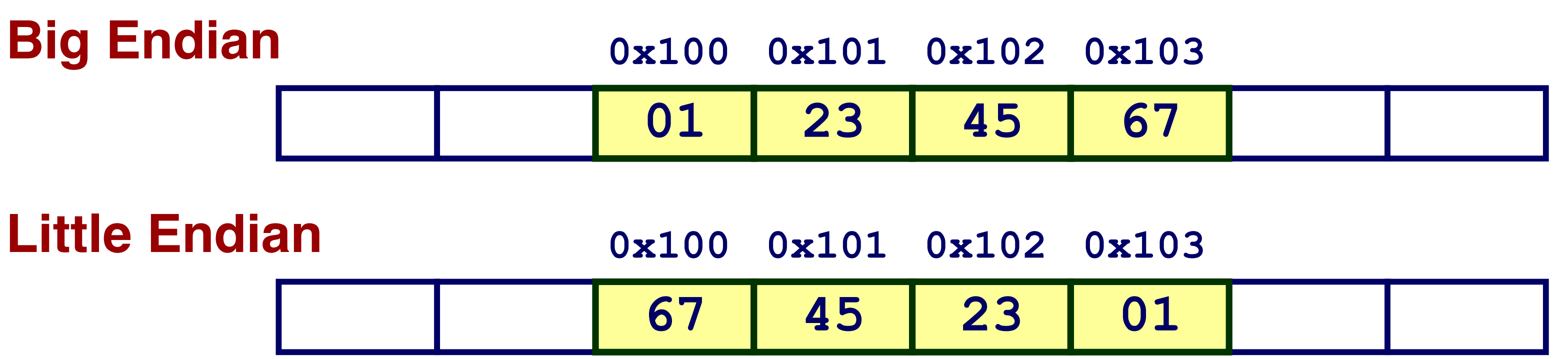
```
gdb-peda$ x/18xb multstore
0x1175 <multstore>:     0xf3  0x0f  0x1e  0xfa  0x53  0x48  0x89  0xd3
0x117d <multstore+8>:   0xe8  0xe7  0xff  0xff  0xff  0x48  0x89  0x03
0x1185 <multstore+16>:  0x5b  0xc3
```

# Byte Ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**
- **Conventions**
  - **Big Endian**: Sun (Oracle SPARC), PPC Mac, ***Internet (e.g., an IP address inside an Internet packet)***
    - Least significant byte has highest address
  - **Little Endian**: ***x86***, ARM processors running Android, iOS, and Linux
    - Least significant byte has lowest address

# Byte Ordering Example

- **Example**
  - Variable x has **4-byte** value of **0x01234567**
  - **Address** given by &x is **0x100**



# Representing Integers

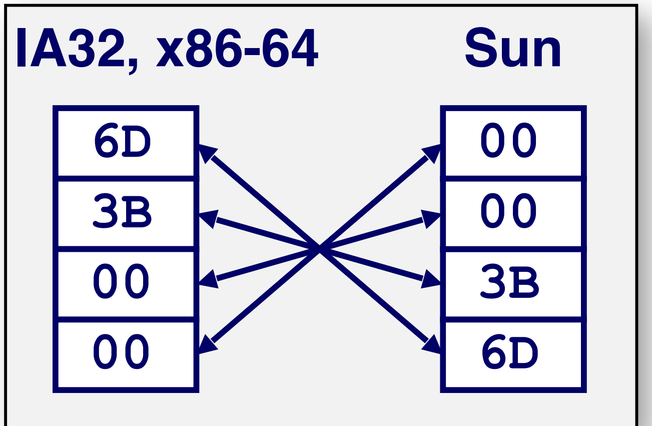
**Decimal:** 15213

**Binary:** 0011 1011 0110 1101

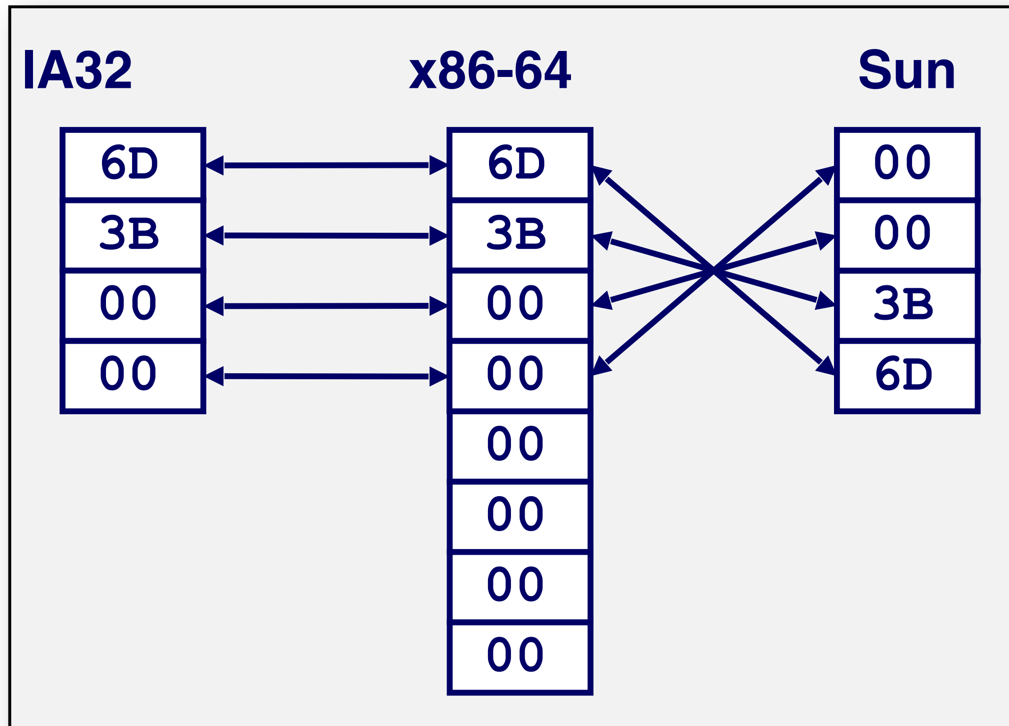
**Hex:** 3 B 6 D

`int A = 15213;`

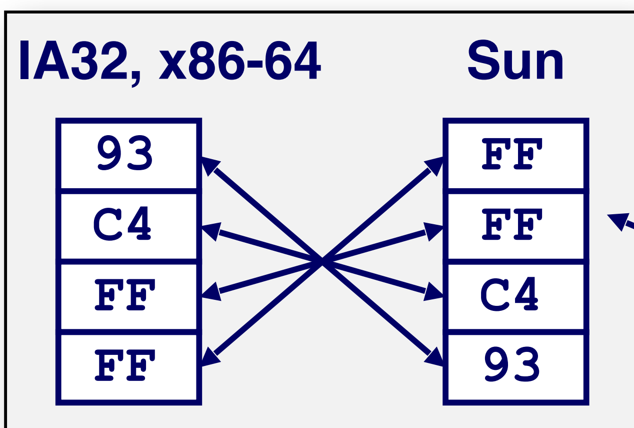
Increasing addresses




`long int C = 15213;`




`int B = -15213;`

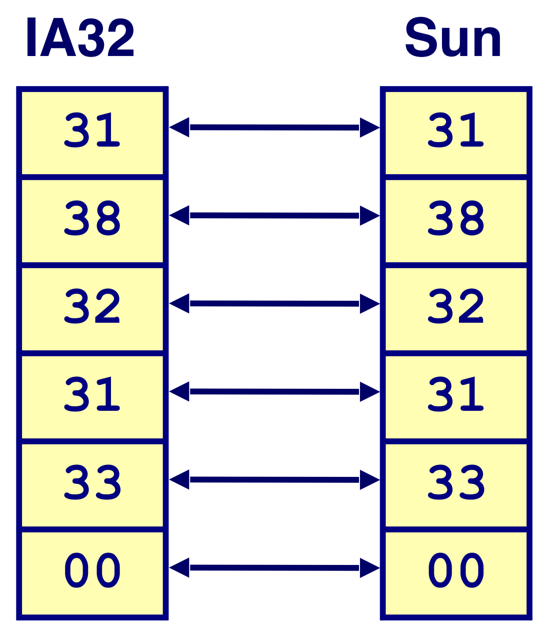



**Two's complement representation**

# Representing Strings

```
char S[6] = "18213";
```

- **Strings in C**
  - Represented by **array of characters**
  - **Each character** encoded in **ASCII format**
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - Digit *i* has code 0x30+*I*
    - ***man ascii** for code table*
  - String should be null-terminated
    - Final character = 0
- **Compatibility**
  - **Byte ordering not an issue**



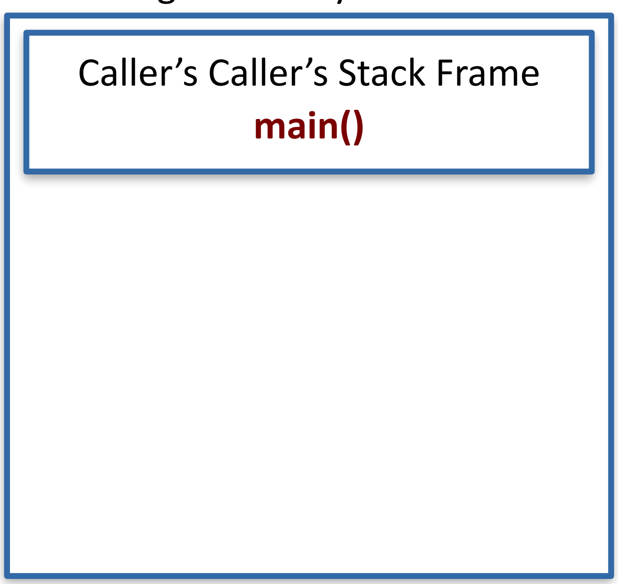
# Outline

- Vulnerability and Exploit
- Program memory structure
- Assembly Review
- Activation Records
- Buffer Overflow
- x86-64

# Stack Frames / Activation Records

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```

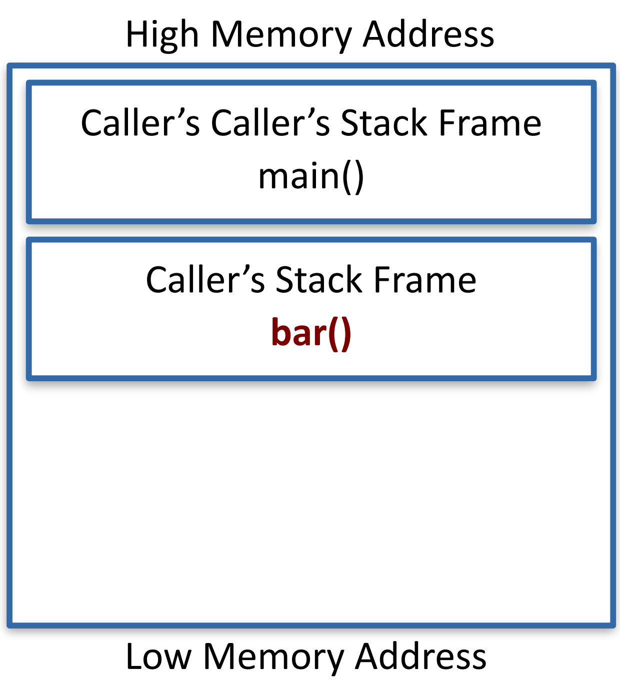
High Memory Address




Low Memory Address

# Stack Frames / Activation Records

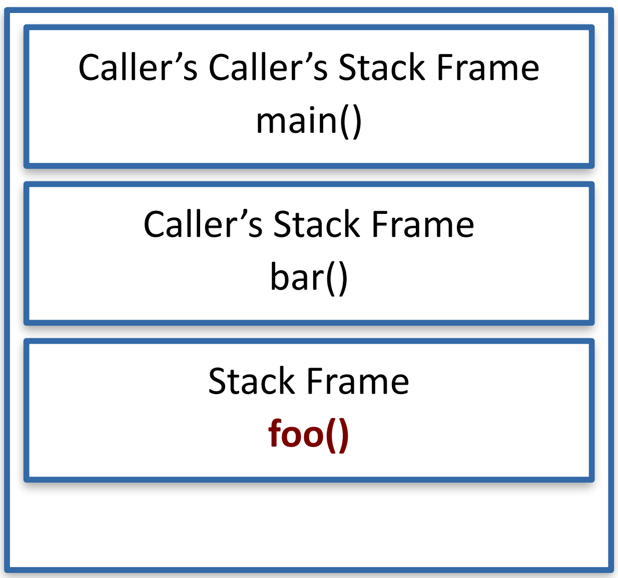
```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



# Stack Frames / Activation Records

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```

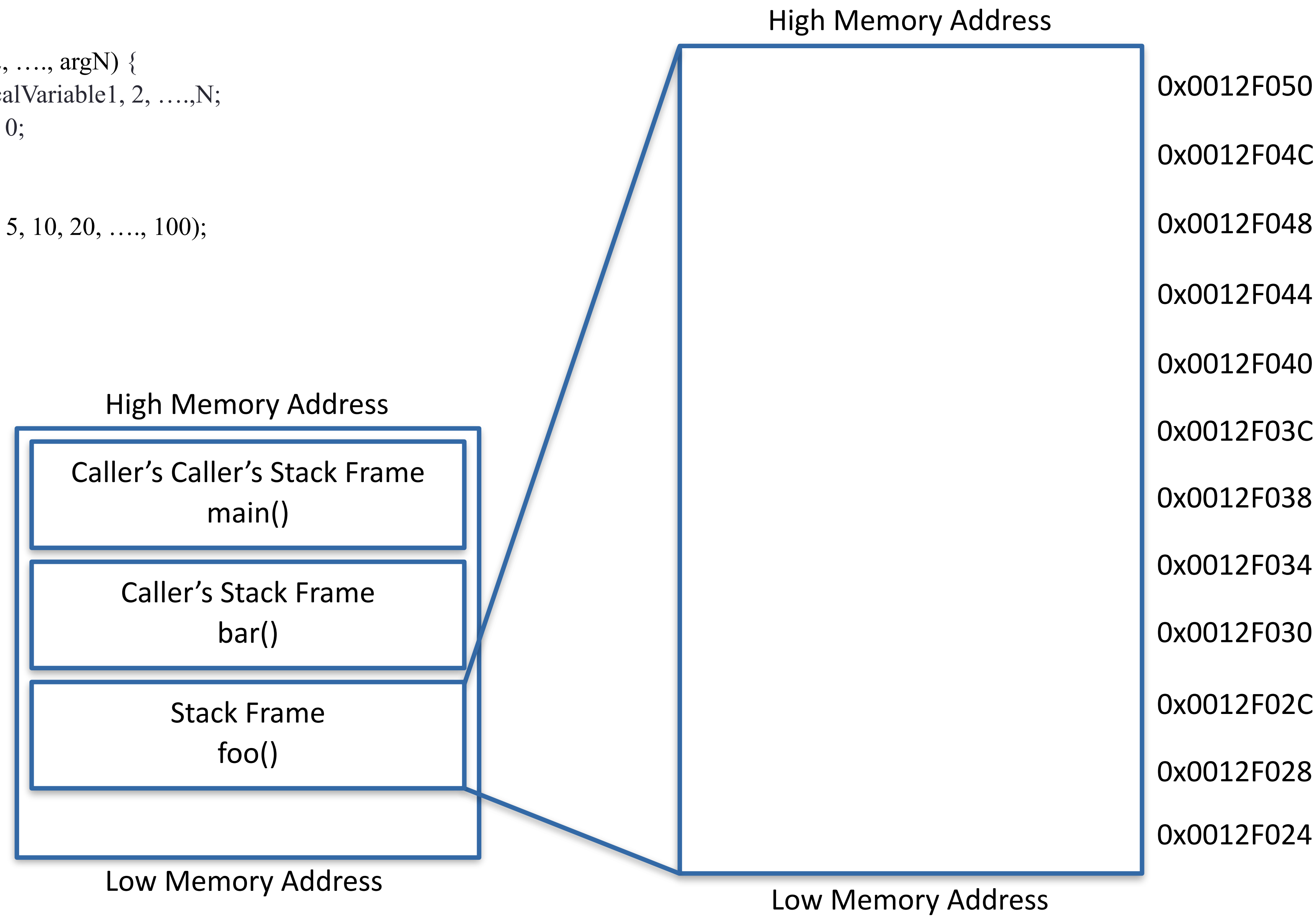
High Memory Address




Low Memory Address

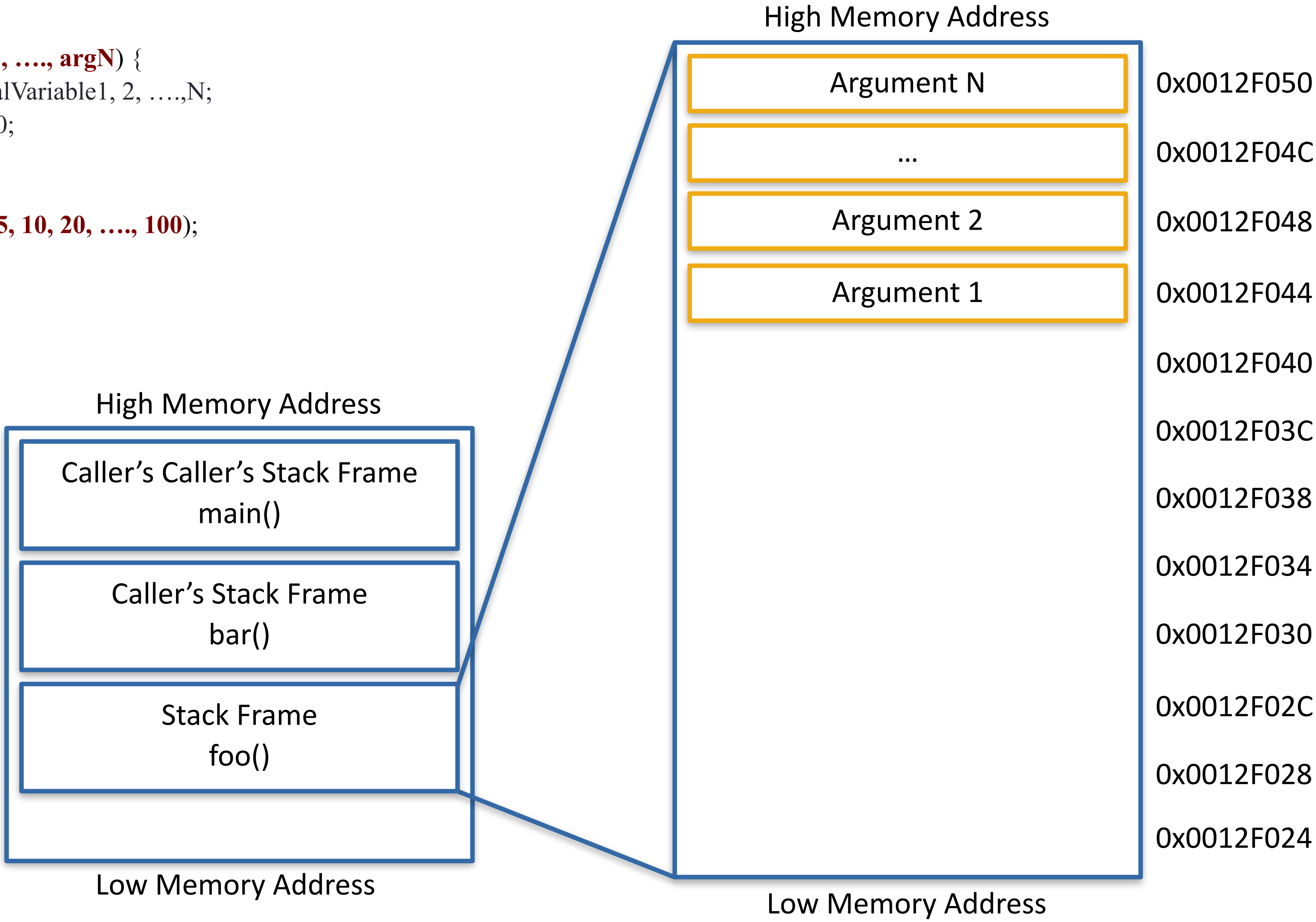
# Stack Frames / Activation Records (Classic IA32)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



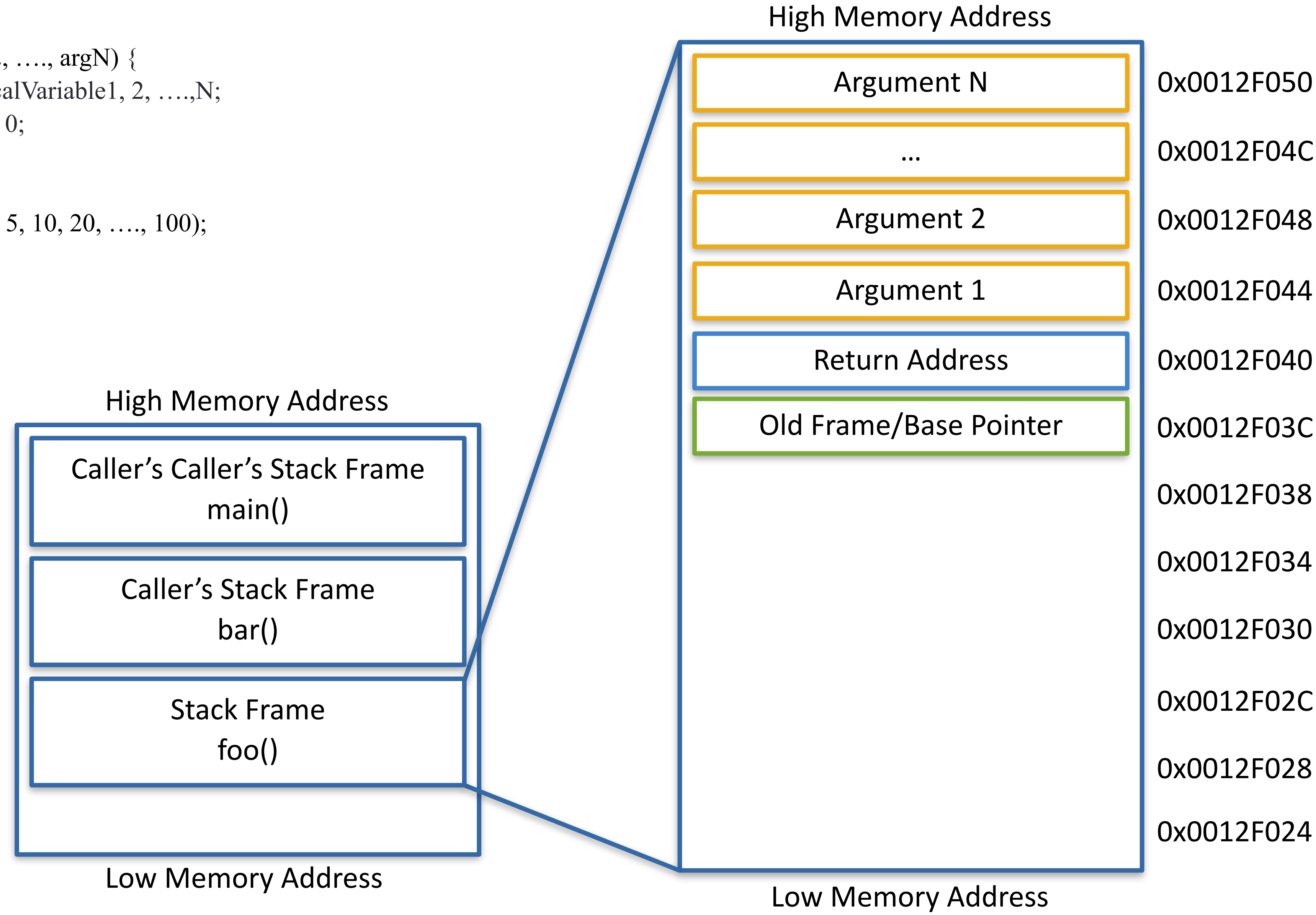
# Stack Frames / Activation Records (Classic IA32)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



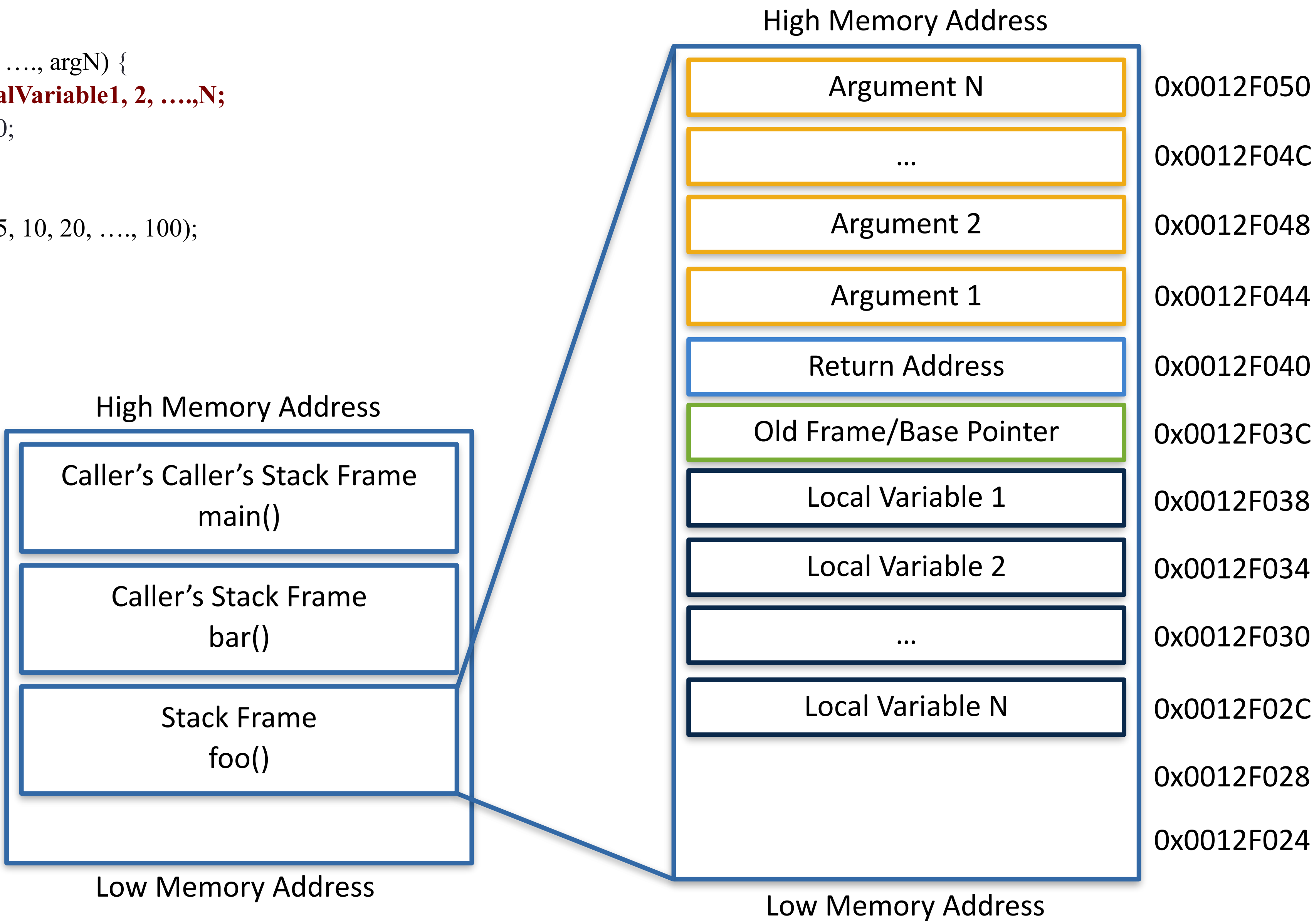
# Stack Frames / Activation Records (Classic IA32)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



# Stack Frames / Activation Records (Classic IA32)

```
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
main() {
        bar();
}
```



# Stack/Base/Instruction Pointers

- **SP** - **S**tack **P**ointer
  - Points to the current **top of the stack**
- **BP** - **B**ase **P**ointer (aka frame pointer)
  - Used to access function parameters and local variables within current stack frame
  - **Required** in **Classic IA32**
  - **Optional** in **Modern IA32/Linux** and **x86-64**: most of the time, the **compiler knows how much needs to be allocated** for local variables and arguments (additional arguments beyond 6 arguments for x86-64) and uses simple arithmetics to add or subtract constant number of bytes from ***SP***
- **IP** – **I**nstruction **P**ointer
  - Tells the CPU what to do **next**
    - Contains the memory address of the next program instruction to be executed

# Activation Record (Example - Classic IA32)

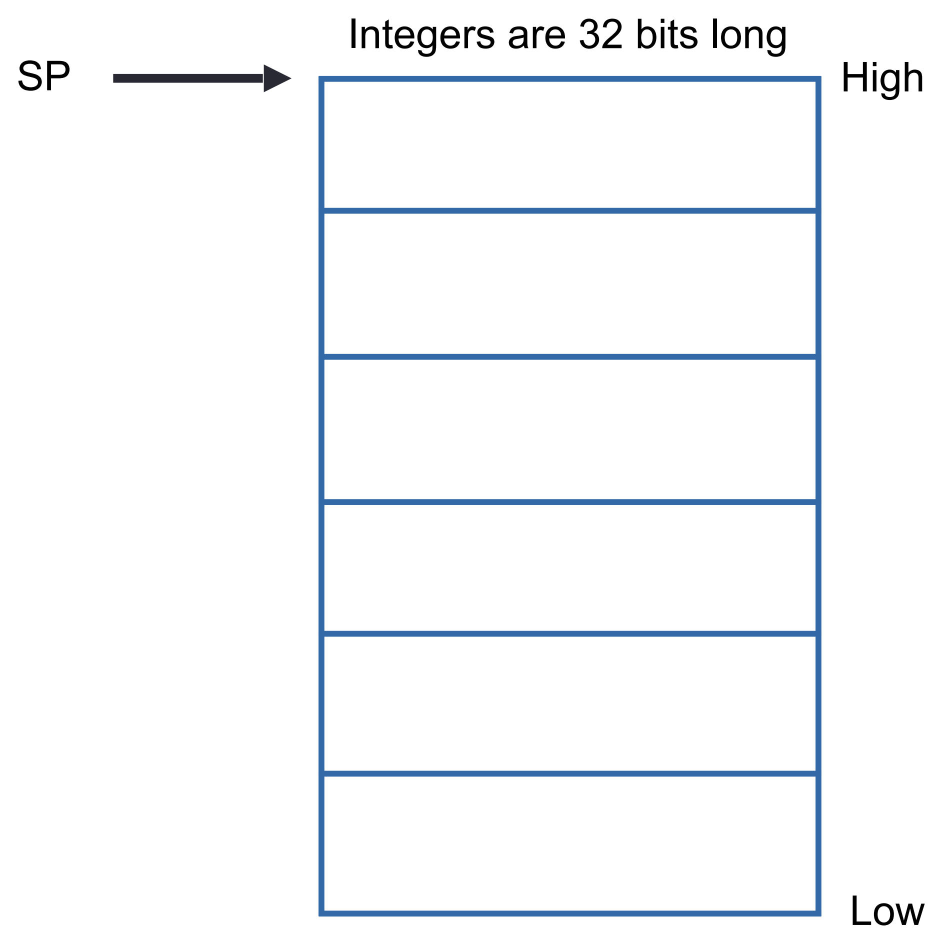
```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}

void main() {
        func(3, 6);
}
```

# Activation Record (Example - Classic IA32)

```
void main() {
        func(3, 6);
}
```

```
30:    6a 06          push   0x6
32:    6a 03          push   0x3
34:    e8 fc ff ff ff call   35
39:    83 c4 08       add    sp,0x8
3c:    90             nop
3d:    c9             leave
3e:    c3             ret
```

**(Intel Format)**

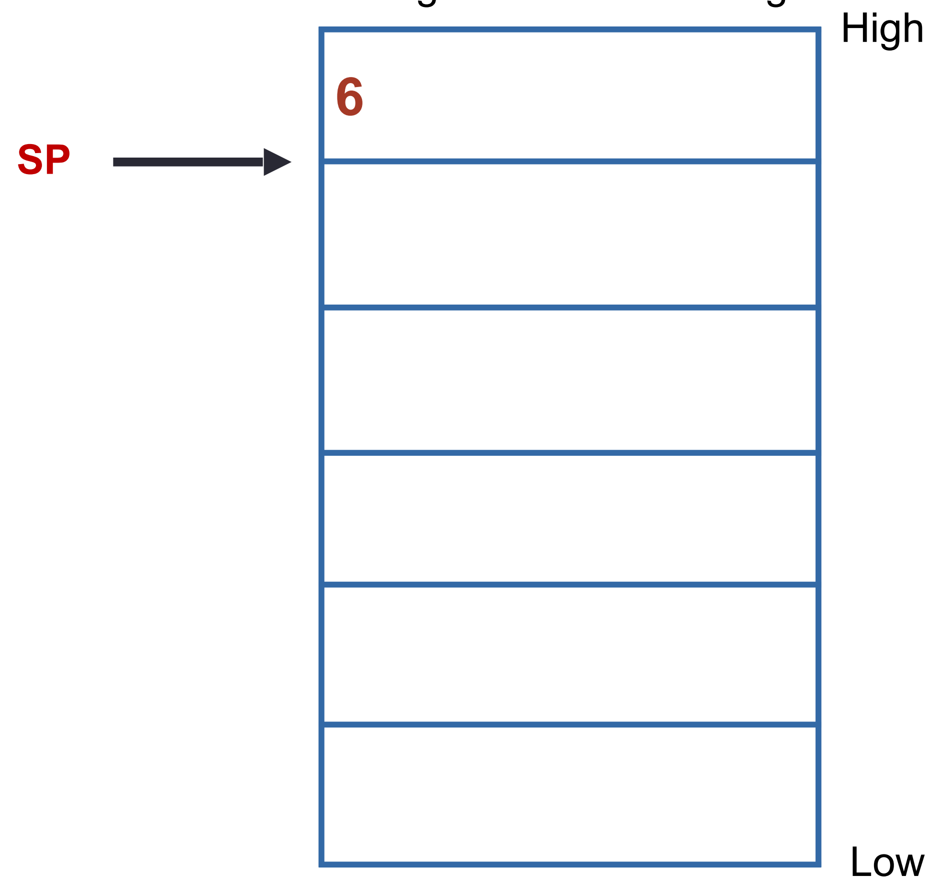


# Activation Record (Example - Classic IA32)

```
void main() {
        func(3, 6);
}
```

```
30:    6a 06            push   0x6
32:    6a 03            push   0x3
34:    e8 fc ff ff ff   call   35
39:    83 c4 08         add    sp,0x8
3c:    90               nop
3d:    c9               leave
3e:    c3               ret
```

Integers are 32 bits long

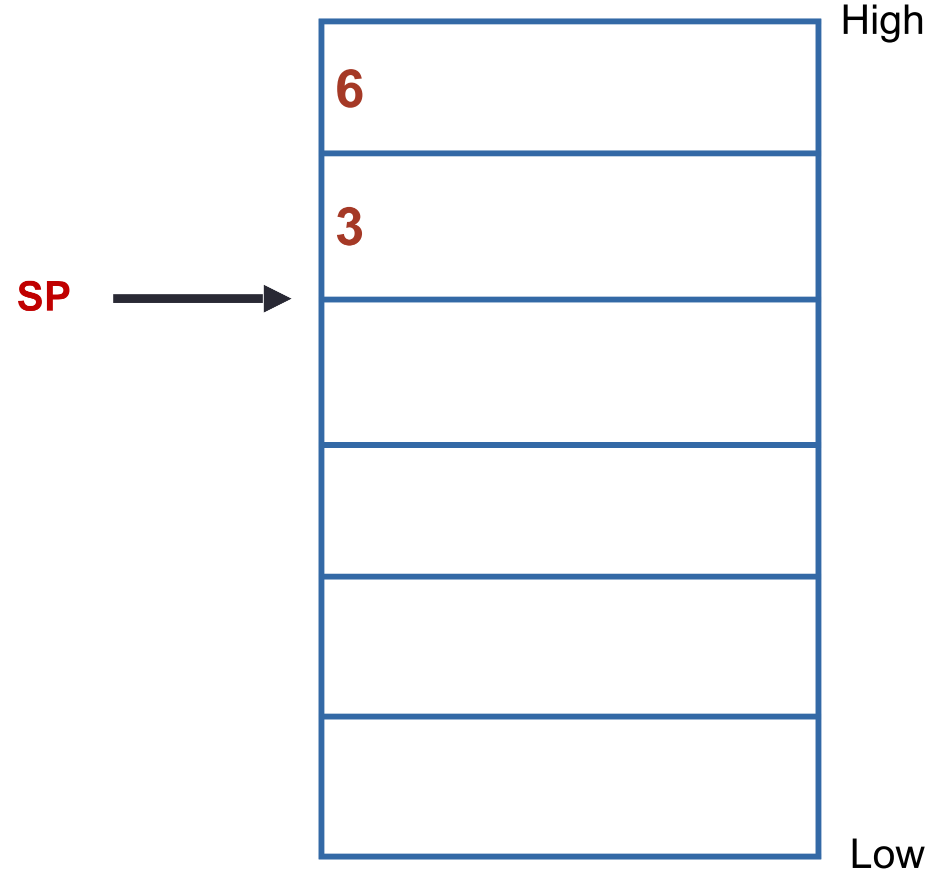


# Activation Record (Example - Classic IA32)

```
void main() {
        func(3, 6);
}
```

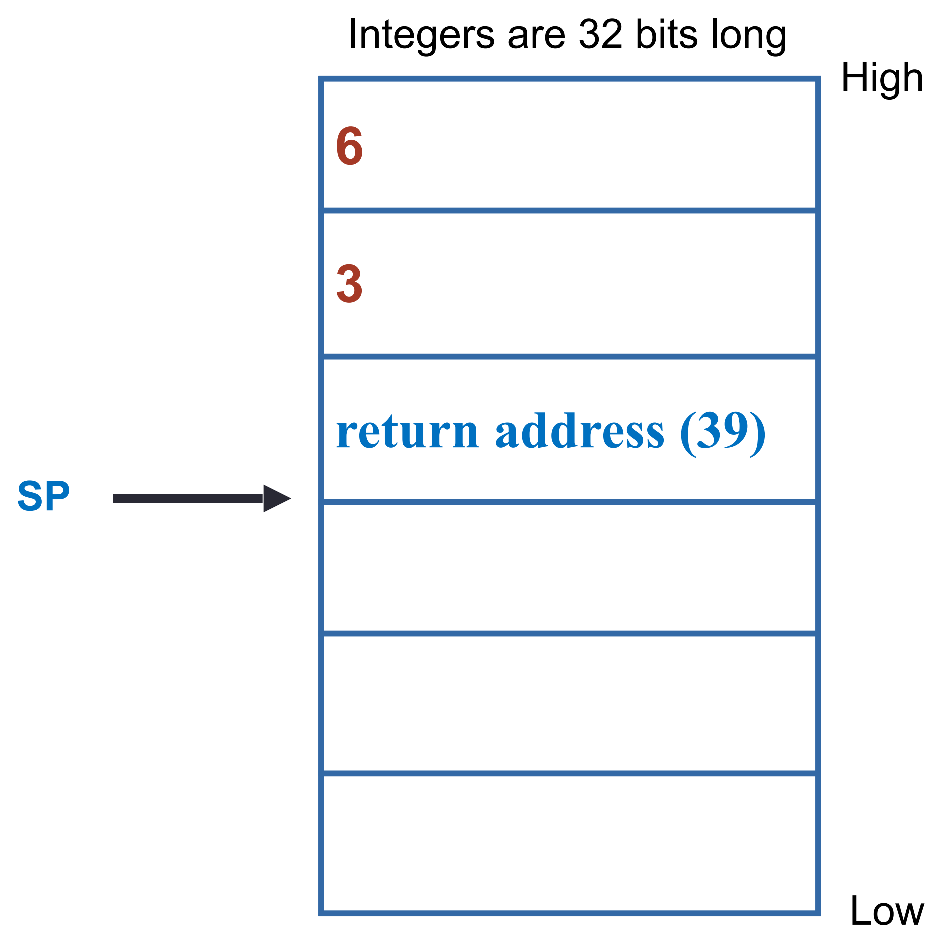
```
30:    6a 06            push   0x6
32:    6a 03            push   0x3
34:    e8 fc ff ff ff   call   35
39:    83 c4 08         add    sp,0x8
3c:    90               nop
3d:    c9               leave
3e:    c3               ret
```

Integers are 32 bits long



# Activation Record (Example - Classic IA32)

```
void main() {
        func(3, 6);
}

 30:    6a 06           push  0x6
 32:    6a 03           push  0x3
 34:    e8 fc ff ff ff  call  35
 39:    83 c4 08        add   sp,0x8
 3c:    90              nop
 3d:    c9              leave
 3e:    c3              ret
```



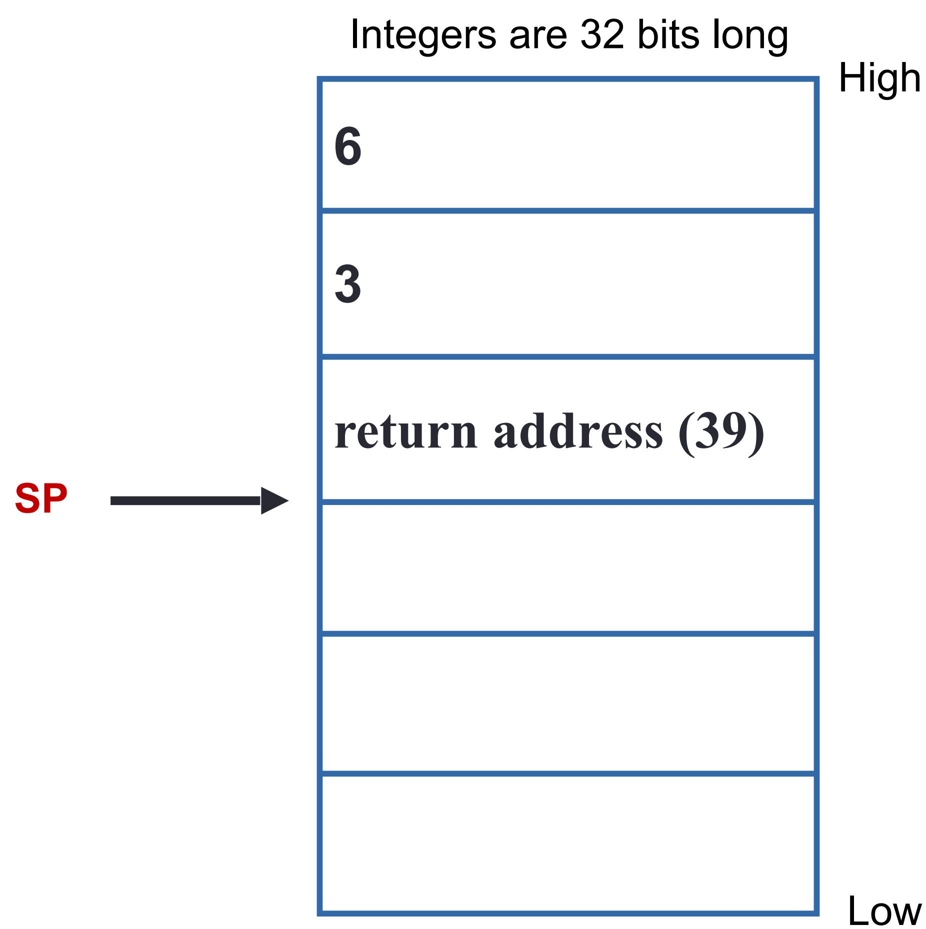

**; IP updates to point to first instruction in func**

# Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Prologue:

```
0:      55      push   bp
```

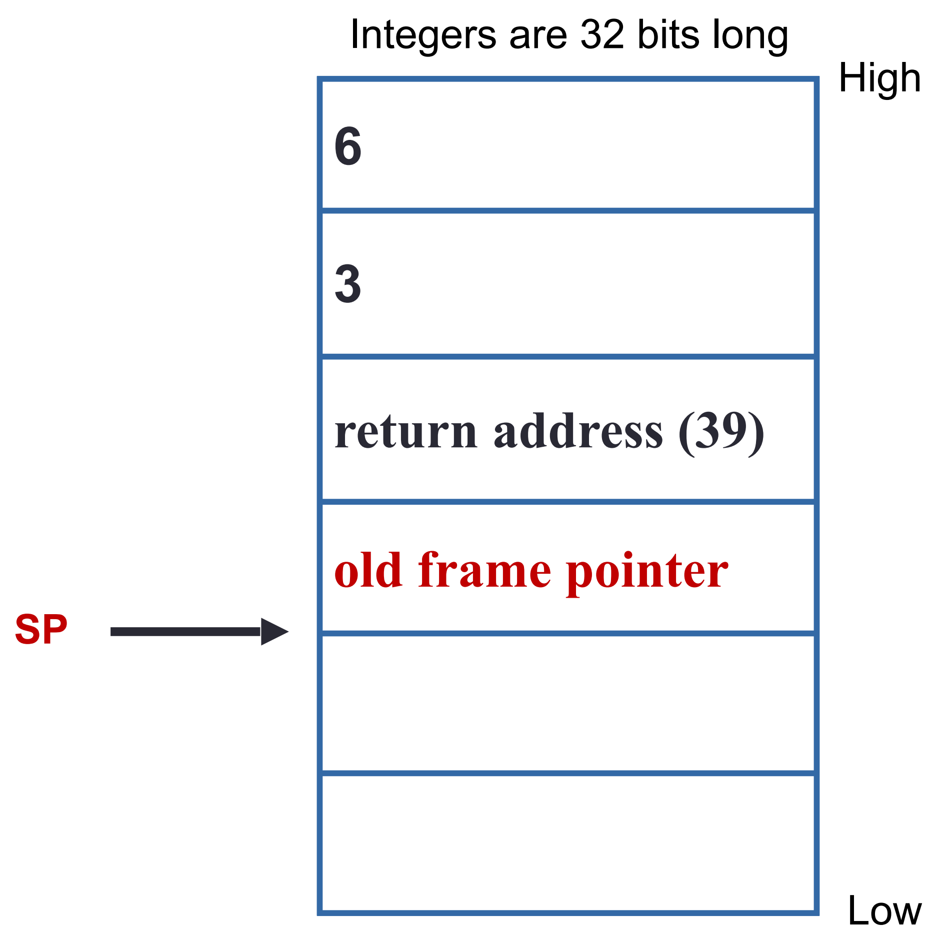


# Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Prologue:

**0:　　55　　push　bp**

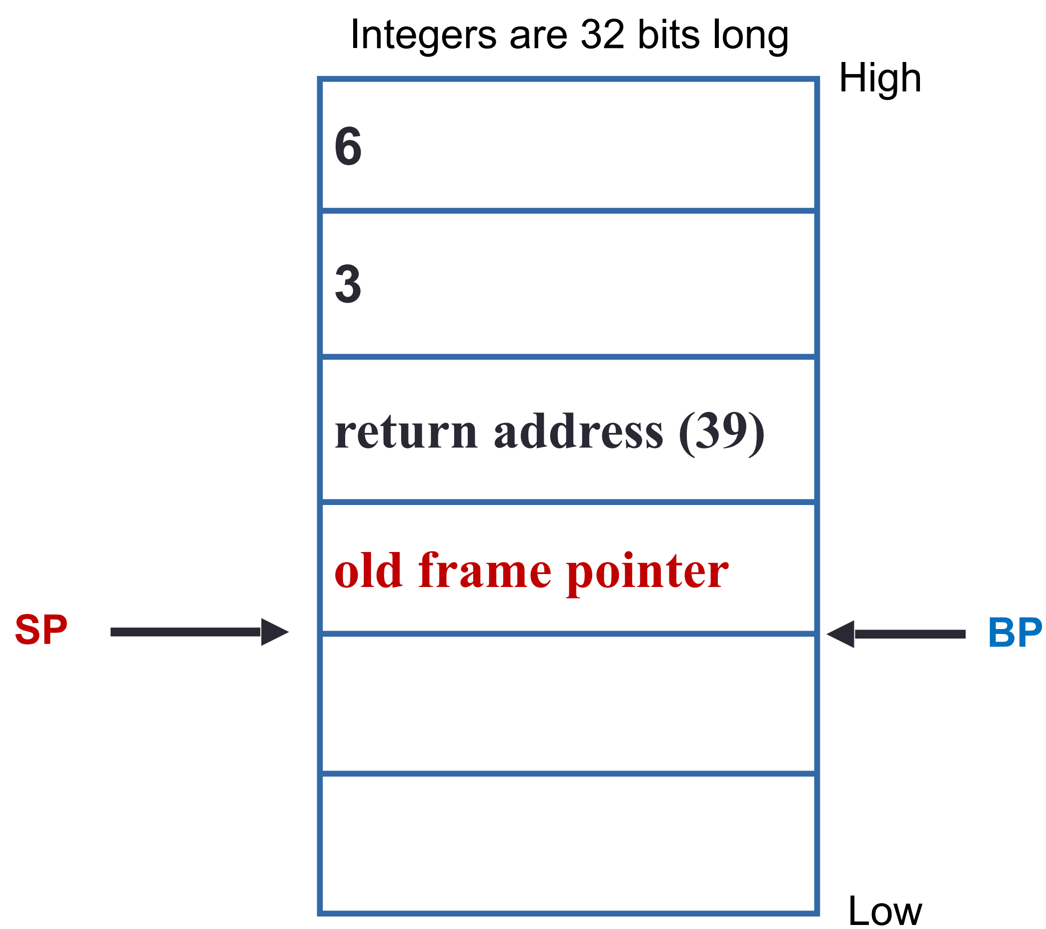


# Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Prologue:

| | | | |
|---|---|---|---|
| **0:** | **55** | **push** | **bp** |
| **1:** | **89 e5** | **mov** | **bp,sp** |

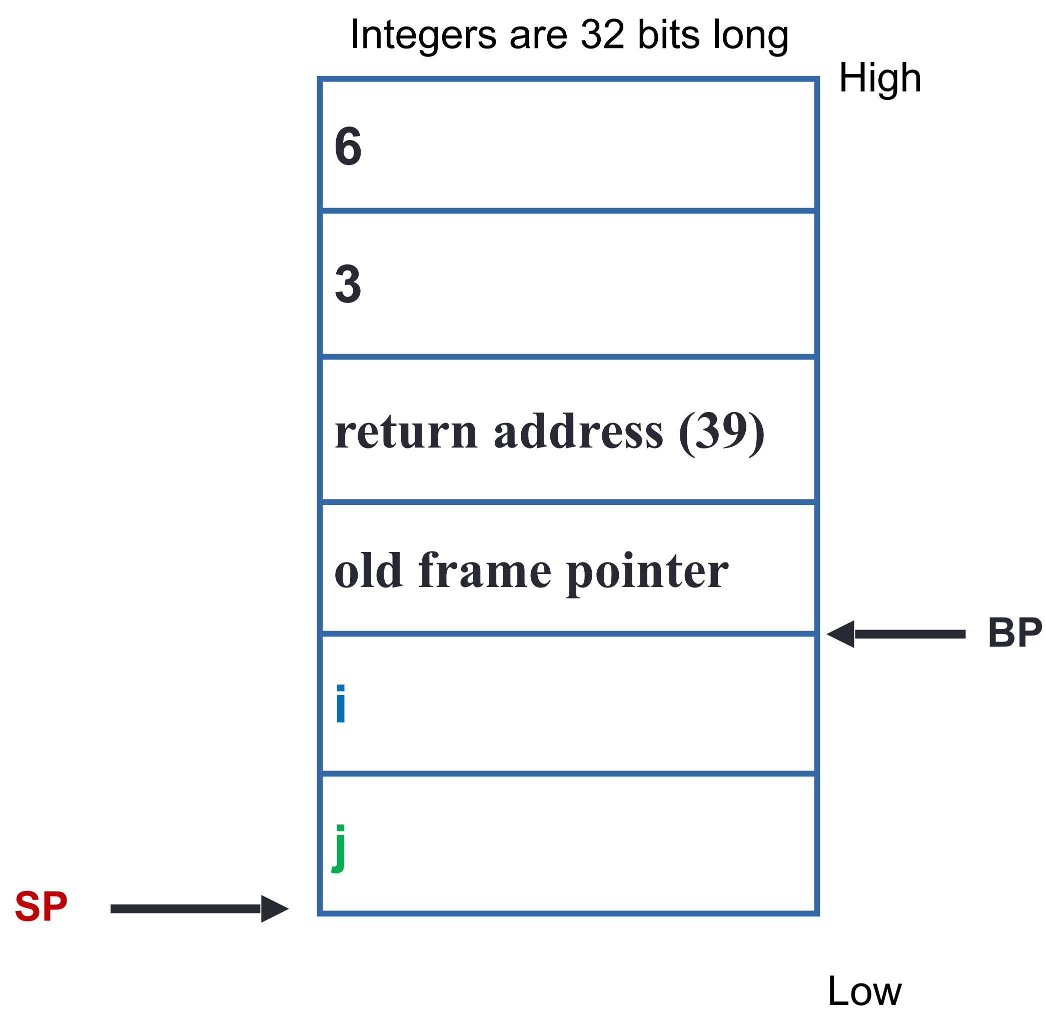


# Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Execution:

```
sub   sp,0x8
mov   eax,DWORD PTR [bp+0x8]
mov   DWORD PTR [bp-0x4],eax
mov   eax,DWORD PTR [bp+0xC]
mov   DWORD PTR [bp-0x8],eax
```

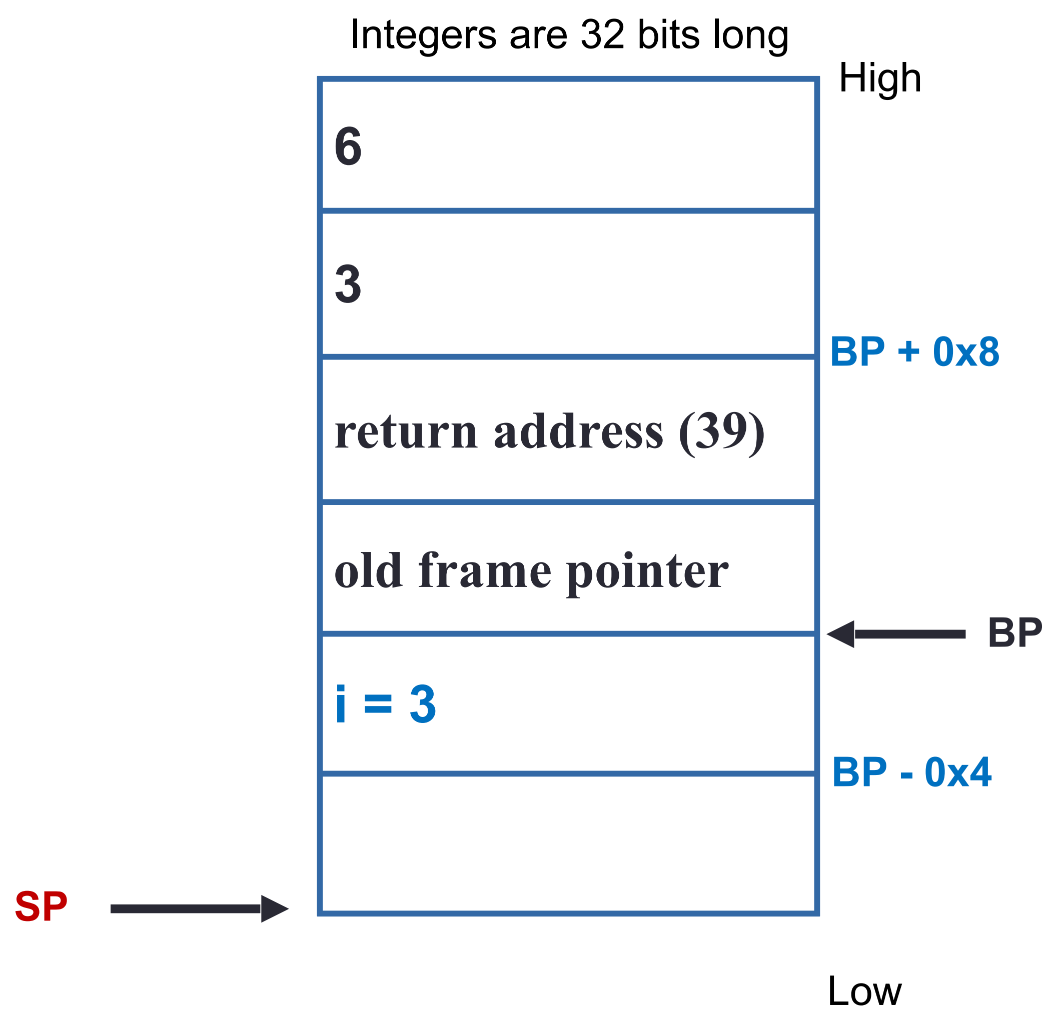


# Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

Function Execution:

```
sub    sp,0x8
mov    eax,DWORD PTR [bp+0x8]
mov    DWORD PTR [bp-0x4],eax
mov    eax,DWORD PTR [bp+0xC]
mov    DWORD PTR [bp-0x8],eax
```

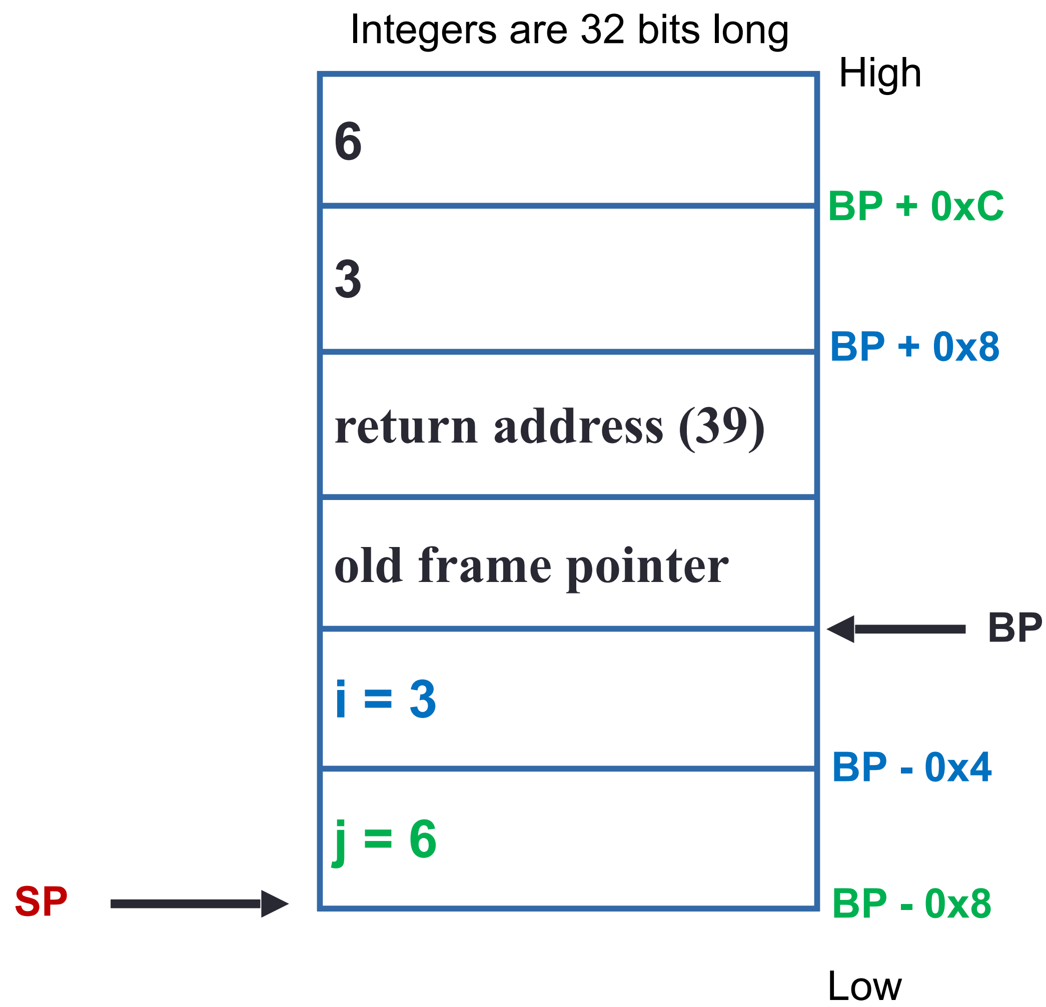


# Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```

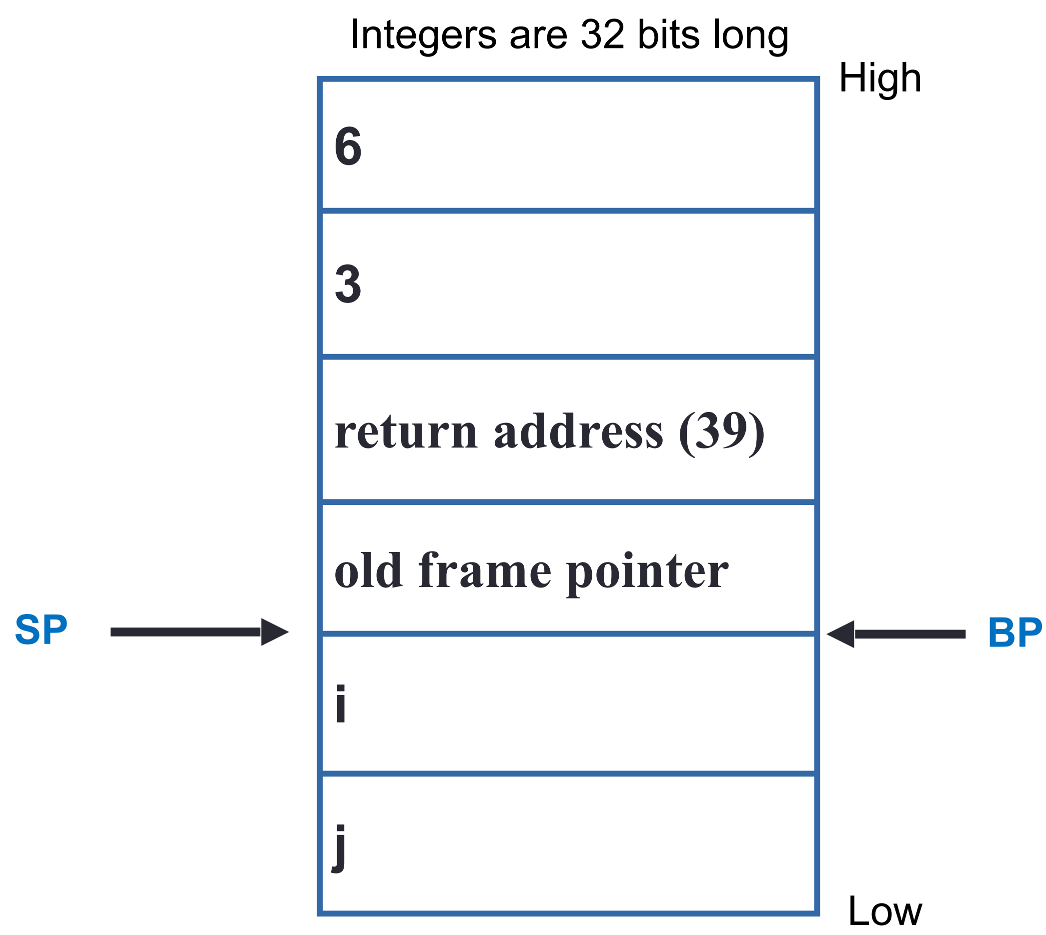
Function Execution:

```
sub    sp,0x8
mov    eax,DWORD PTR [bp+0x8]
mov    DWORD PTR [bp-0x4],eax
mov    eax,DWORD PTR [bp+0xC]
mov    DWORD PTR [bp-0x8],eax
```



# Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```




Function Epilogue:

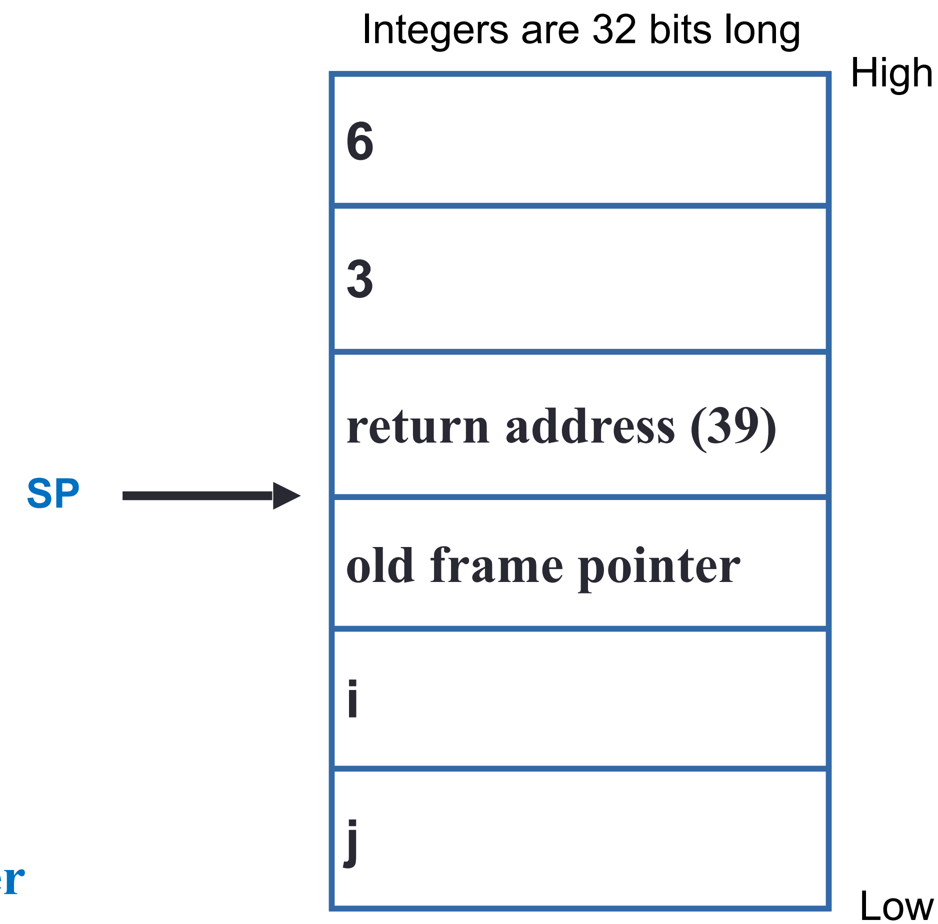
**mov sp, bp**

**pop bp**

**ret**

# Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```




Function Epilogue:

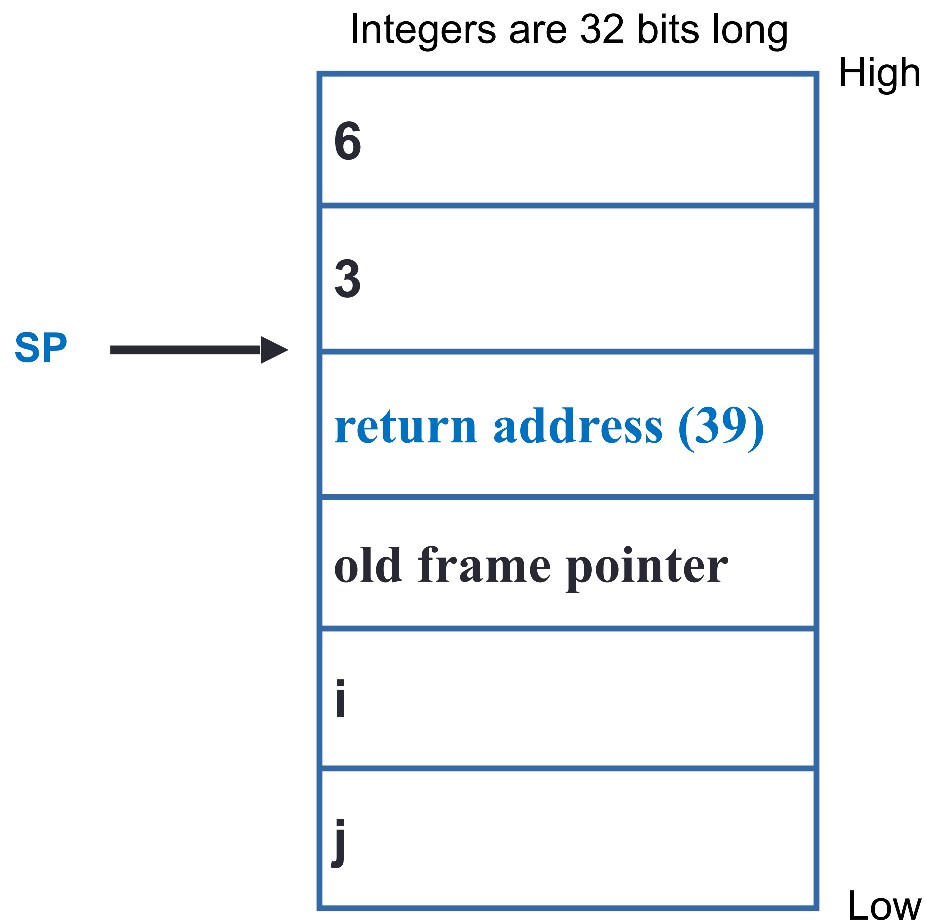
**mov sp, bp**

**pop bp ; now bp has old frame pointer**

**ret**

# Activation Record (Example - Classic IA32)

```
int func(int a, int b) {
        int i, j;
        i = a;
        j = b;
        return 0;
}
```




Function Epilogue:

**mov sp, bp**

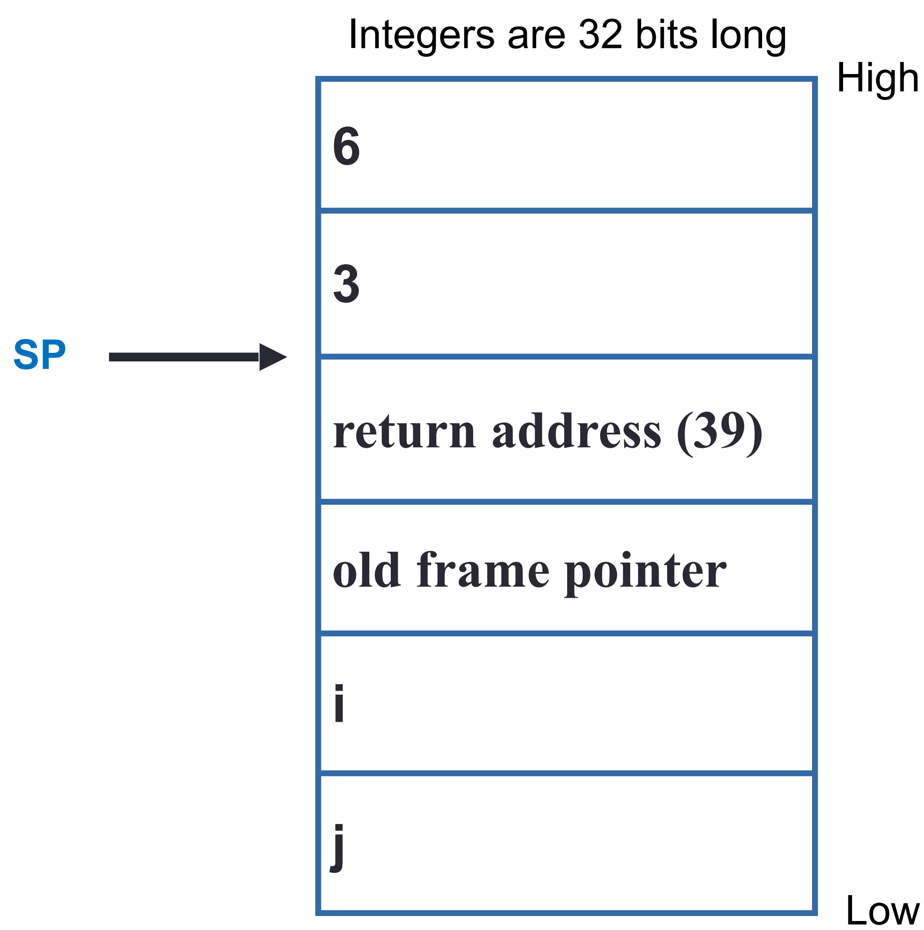
**pop bp**

**ret ; pops return address (39) and places it in IP (i.e., instruction pointer)**

# Activation Record (Example - Classic IA32)

```
void main() {
        func(3, 6);
}
```

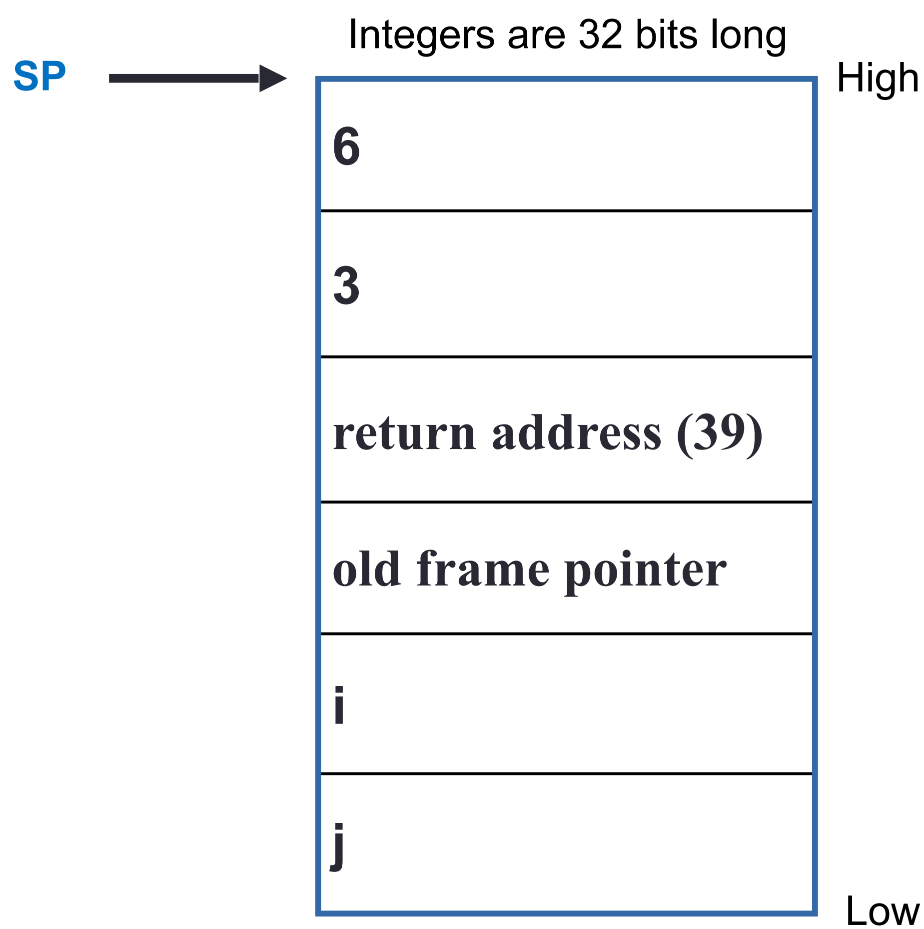
```
30:   6a 06            push  0x6
32:   6a 03            push  0x3
34:   e8 fc ff ff ff   call  35
39:   83 c4 08         add   sp,0x8
3c:   90               nop
3d:   c9               leave
3e:   c3               ret
```



# Activation Record (Example - Classic IA32)

```
void main() {
        func(3, 6);
}
```

```
30:    6a 06            push   0x6
32:    6a 03            push   0x3
34:    e8 fc ff ff ff   call   35
39:    83 c4 08         add    sp,0x8
3c:    90               nop
3d:    c9               leave
3e:    c3               ret
```



# Outline

- Vulnerability and Exploit
- Program memory structure
- Assembly Review
- Activation Records
- Buffer Overflow
- x86-64

# Activation Record (Example 2 - Classic IA32)

```
void func(int i1) {
        char first[8] = "Robert";
        char last[8] = "Jackson";
}

void main() {
        func(5);
}
```

- last[0] = "J"
- first[0] = "R"
- last[8] = ??
  - It is a legal reference only if it is never prevented by any boundary protection mechanism

| | | | | | |
|---|---|---|---|---|---|
| | i1 = 5 | | | | Integers are 32 bits long |
| | return address | | | | |
| | old frame pointer | | | | |
| | r | t | \0 | ? | chars are 8 bits long |
| first | R | o | b | e | |
| | s | o | n | \0 | |
| last | J | a | c | k | |

# Activation Record (Example 2 - Classic IA32)

```
void func(int i1) {
        char first[8] = “Robert”;
        char last[8] = “Jackson”;
}

void main() {
        func(5);
}
```

- Imagine while writing on ***last***, we can **overrun** ***last’s boundary*** and **overwrite *first*** without directly accessing ***first***!

| | | | | | |
|---|---|---|---|---|---|
| | i1 = 5 | | | | Integers are 32 bits long |
| | return address | | | | |
| | old frame pointer | | | | |
| | ~~r~~ \0 | t | \0 | ? | chars are 8 bits long |
| first | ~~R~~H | ~~o~~ a | ~~b~~ c | ~~e~~ k | |
| | s | o | n | \0 | |
| last | J | a | c | k | |

# Buffer Overflow

```
void func(int i1) {
        char first[8] = “Robert”;
        char last[8] = “Jackson”;
}

void main() {
        func(5);
}
```

- A **bug/vulnerability** in the program code which can cause an **overrun** of a buffer’s boundary and allows **overwriting adjacent memory locations** when writing data to a buffer.

| | | | | |
|---|---|---|---|---|
| | i1 = 5 | | | |
| | return address | | | |
| | old frame pointer | | | |
| | ~~r~~ \0 | t | \0 | ? |
| first | ~~R~~H | ~~o~~ a | ~~b~~ c | ~~e~~ k |
| | s | o | n | \0 |
| last | J | a | c | k |

# Outline

- Vulnerability and Exploit
- Program memory structure
- Assembly Review
- Activation Records
- Buffer Overflow
- x86-64

# Note on x86-64 Intel format vs AT&T format

```
% gcc -Og -S -masm=intel sumstore.c

sumstore:
    push    rbx #no '%' as prefix
    mov     rbx, rdx #source: rdx, destination: rbx
    call    plus
    mov     QWORD PTR [rbx], rax
    pop     rbx
    ret
```

```
% gcc -Og -S sumstore.c (if compiler default was AT&T
format)

sumstore:
   pushq   %rbx
   movq    %rdx, %rbx #source: rdx, destination: rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

# Note on x86-64 Intel format vs AT&T format

Intel:

```
$ gcc -Og -S -masm=intel mstore.c
$ cat mstore.s
```

AT&T:

```
$ gcc -Og -S -masm=att mstore.c
$ cat mstore.s
```

# Note on x86-64 Intel format vs AT&T format

- **Switching to Intel format in gdb**

```
gdb-peda$ set disassembly-flavor intel
gdb-peda$ layout asm
```

- **Switching to AT&T format in gdb**

```
gdb-peda$ set disassembly-flavor att
gdb-peda$ layout asm
```

Must step over to next instruction to be able to see the change

Or use the arrows

# Addressing Modes (AT&T Format)

- **Most General Form**

  **D(Rb,Ri,S)** **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  - D: Constant "displacement" 1, 2, or 4 bytes
  - R**b**: **Base** register: Any of 16 integer registers
  - R**i**: **Index** register: Any, except for `%rsp`
  - **S**: **Scale**: 1, 2, 4, or 8
  - E,g, Multidimensional array:
    - **Reg[Ri]** is size of **row**
    - **S** is **row index** / and **D** is **column index**

- **Special Cases**

| | |
|---|---|
| **(Rb,Ri)** | **Mem[Reg[Rb]+Reg[Ri]]** |
| **D(Rb,Ri)** | **Mem[Reg[Rb]+Reg[Ri]+D]** |
| **(Rb,Ri,S)** | **Mem[Reg[Rb]+S*Reg[Ri]]** |

# Addressing Modes (AT&T Format) - Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x0100` |

**D(Rb,Ri,S)** **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- D: Constant "displacement" 1, 2, or 4 bytes
- R**b**: **Base** register: Any of 16 integer registers
- R**i**: **Index** register: Any, except for `%rsp`
- **S**: **Scale**: 1, 2, 4, or 8

| Expression | Address Computation | Address |
|---|---|---|
| `0x8(%rdx)` | | |
| `(%rdx,%rcx)` | | |
| `(%rdx,%rcx,4)` | | |
| `0x80(,%rdx,2)` | | |

# Addressing Modes (AT&T Format) - Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x0100` |

**D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- D: Constant "displacement" 1, 2, or 4 bytes
- R**b**: **Base** register: Any of 16 integer registers
- R**i**: **Index** register: Any, except for `%rsp`
- **S**: **Scale**: 1, 2, 4, or 8

| Expression | Address Computation | Address |
|---|---|---|
| `0x8(%rdx)` | `0xf000 + 0x8` | `0xf008` |
| `(%rdx,%rcx)` | `0xf000 + 0x100` | `0xf100` |
| `(%rdx,%rcx,4)` | `0xf000 + 4*0x100` | `0xf400` |
| `0x80(,%rdx,2)` | `2*0xf000 + 0x80` | `0x1e080` |

# Function Arguments (x86-64)

## Registers

- **First 6 arguments**

| |
|---|
| `%rdi` |
| `%rsi` |
| `%rdx` |
| `%rcx` |
| `%r8` |
| `%r9` |

The registers are used in this specified order.
Register **name depends on the size of the data type being passed.**

E.g.,
Operand size **64bits**: %rdi, %rdx, %r8, %r9
Operand size **32bits**: %edi, %edx, %r8**d**, %r9d
Operand size **16bits**: %di, %dx, %r8**w**, %r9w
Operand size **8bits**: %di**l**, %cl, %r8**b**, %r9b

## Stack

| |
|---|
| • • • |
| Arg *n* |
| • • • |
| Arg 8 |
| Arg 7 |

- **Return value**

| |
|---|
| `%rax` |

- **Only allocate stack space when needed (much less efficient to allocate on stack)**
- **IA32: smaller number of registers, and thus must always use the stack to store all arguments**

# Passing Data Example in x86-64

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

**dest** in **%rdx** is moved in **%rbx** in case the **mult2** call actually needs **%rdx** (may call another function with 3 arguments).

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: mov    %rdx,%rbx       # Save dest
  400544: call   400550 <mult2>  # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)     # Save at dest
  • • •
```

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

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

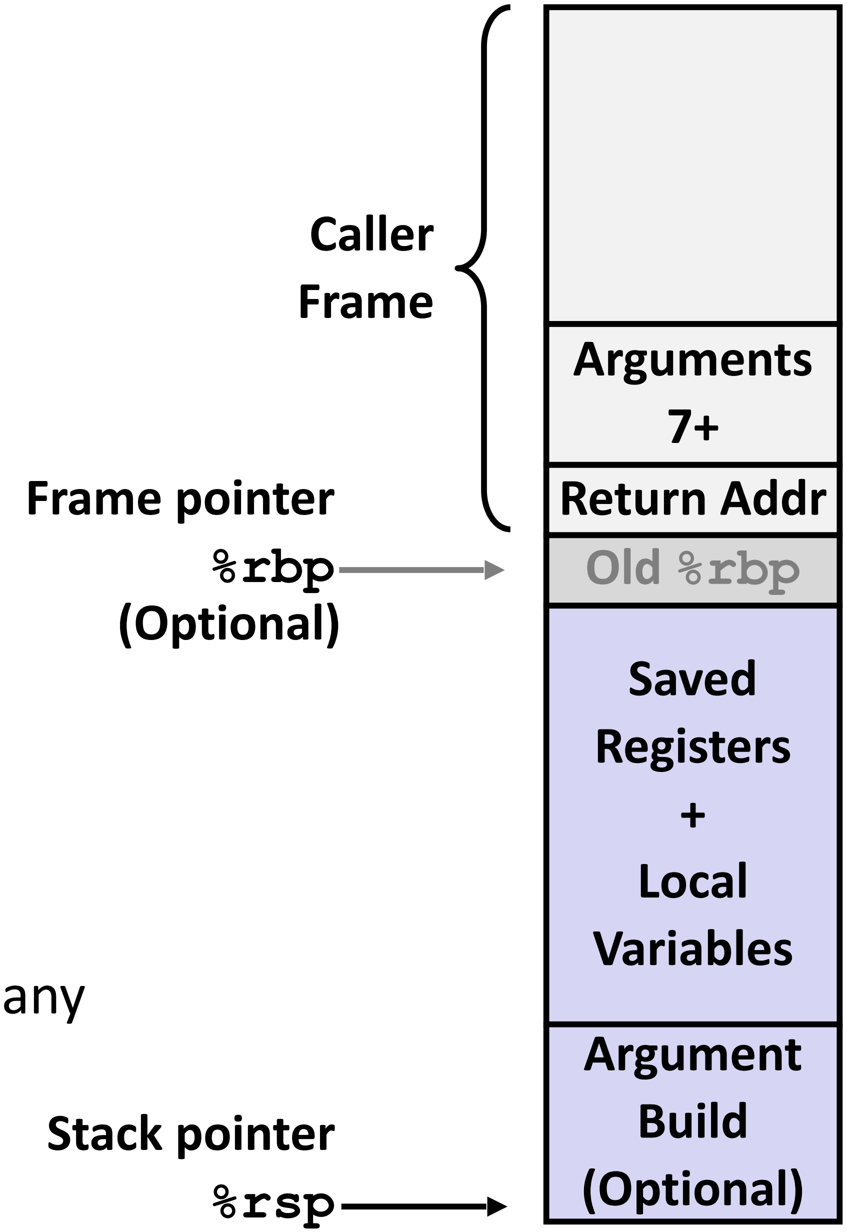
# x86-64/Linux Stack Frame

## Caller Stack Frame

- Arguments for this call (if more than 7)
- Return address
  - Pushed by **`call`** instruction

## Current Stack Frame

- Old frame pointer (**optional**)
- Saved register context
- Local variables
  *if can't keep in registers*
- "Argument Build"
  Parameters for function about to call if any



# x86-64 Example (Local Variables in Registers)

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Although there are **local variables** here, **nothing was allocated onto the stack** since the compiler managed to store all variables in registers (mainly rdi and rsi)
—> much more efficient

```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

| Register | Use(s) |
|---|---|
| `%rdi` | Argument `p` |
| `%rsi` | Argument `val`, `y` |
| `%rax` | `x`, Return value |

# x86-64 Complete Example

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  subq    $24, %rsp ;make room for local var
  movq    $1, 8(%rsp) ;local variable x1
  movl    $2, 4(%rsp) ;local variable x2
  movw    $3, 2(%rsp) ;local variable x3
  movb    $4, 1(%rsp) ;local variable x4
  leaq    4(%rsp), %rcx ;argument &x2
  leaq    8(%rsp), %rsi ;argument &x1
  leaq    1(%rsp), %rax ;argument &x4
  pushq   %rax          ;push arg &x4 on stack
  pushq   $4            ;push arg x4 on stack
  leaq    18(%rsp), %r9 ;argument &x3
  movl    $3, %r8d      ;argument x3
  movl    $2, %edx      ;argument x2
  movl    $1, %edi      ;argument x1
  movl    $0, %eax
  call    proc
  addq    $40, %rsp ;restore stack pointer
  ret
```

# x86-64 Complete Example

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  subq    $24, %rsp ;make room for local var
  movq    $1, 8(%rsp) ;local variable x1
  movl    $2, 4(%rsp) ;local variable x2
  movw    $3, 2(%rsp) ;local variable x3
  movb    $4, 1(%rsp) ;local variable x4
  leaq    4(%rsp), %rcx ;argument &x2
  leaq    8(%rsp), %rsi ;argument &x1
  leaq    1(%rsp), %rax ;argument &x4
  pushq   %rax          ;push arg &x4 on stack
  pushq   $4            ;push arg x4 on stack
  leaq    18(%rsp), %r9 ;argument &x3
  movl    $3, %r8d      ;argument x3
  movl    $2, %edx      ;argument x2
  movl    $1, %edi      ;argument x1
  movl    $0, %eax
  call    proc
  addq    $40, %rsp ;restore stack pointer
  ret
```

**Stack (entry size = 8 bytes)**

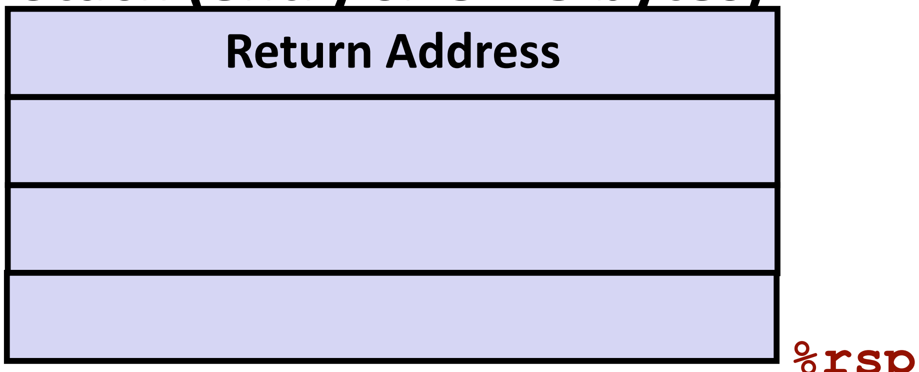
| Return Address | %rsp |
| --- | --- |

# x86-64 Complete Example

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

**Stack (entry size = 8 bytes)**

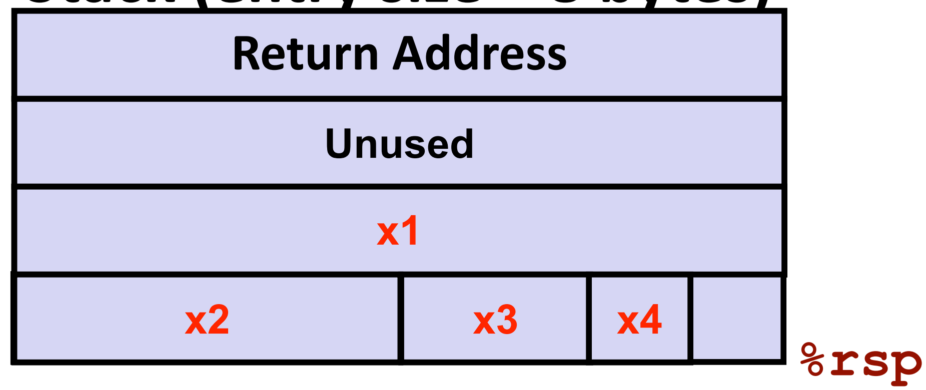


# x86-64 Complete Example

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  subq    $24, %rsp ;make room for local var
  movq    $1, 8(%rsp) ;local variable x1
  movl    $2, 4(%rsp) ;local variable x2
  movw    $3, 2(%rsp) ;local variable x3
  movb    $4, 1(%rsp) ;local variable x4
  leaq    4(%rsp), %rcx ;argument &x2
  leaq    8(%rsp), %rsi ;argument &x1
  leaq    1(%rsp), %rax ;argument &x4
  pushq   %rax          ;push arg &x4 on stack
  pushq   $4            ;push arg x4 on stack
  leaq    18(%rsp), %r9 ;argument &x3
  movl    $3, %r8d      ;argument x3
  movl    $2, %edx      ;argument x2
  movl    $1, %edi      ;argument x1
  movl    $0, %eax
  call    proc
  addq    $40, %rsp ;restore stack pointer
  ret
```

**Stack (entry size = 8 bytes)**

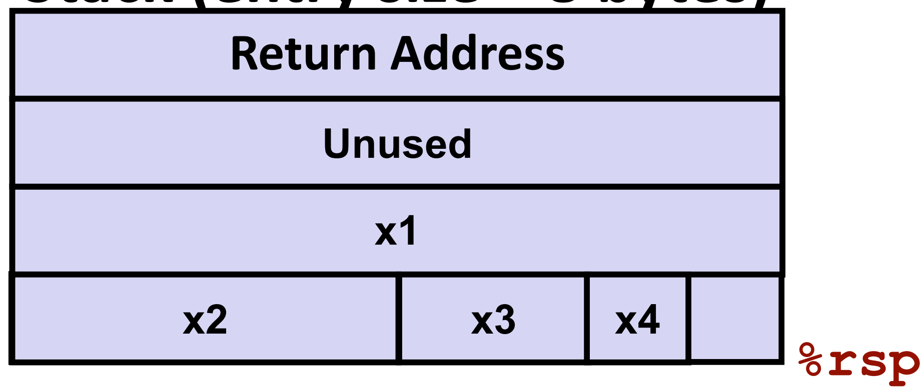


# x86-64 Complete Example

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  subq    $24, %rsp ;make room for local var
  movq    $1, 8(%rsp) ;local variable x1
  movl    $2, 4(%rsp) ;local variable x2
  movw    $3, 2(%rsp) ;local variable x3
  movb    $4, 1(%rsp) ;local variable x4
  leaq    4(%rsp), %rcx ;argument &x2
  leaq    8(%rsp), %rsi ;argument &x1
  leaq    1(%rsp), %rax ;argument &x4
  pushq   %rax          ;push arg &x4 on stack
  pushq   $4            ;push arg x4 on stack
  leaq    18(%rsp), %r9 ;argument &x3
  movl    $3, %r8d      ;argument x3
  movl    $2, %edx      ;argument x2
  movl    $1, %edi      ;argument x1
  movl    $0, %eax
  call    proc
  addq    $40, %rsp ;restore stack pointer
  ret
```

**Stack (entry size = 8 bytes)**



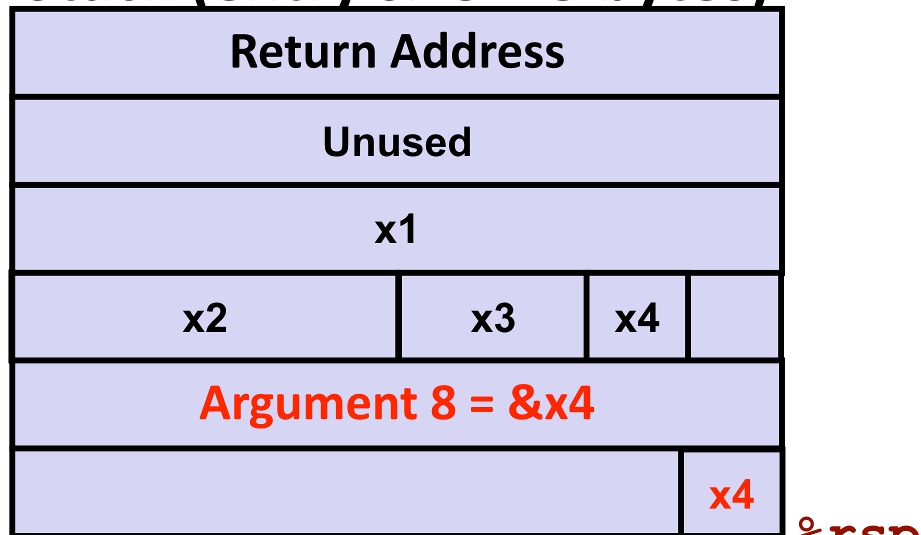

| Register | Use(s) |
|---|---|
| %rdi | |
| %rsi | &x1 |
| %edx | |
| %rcx | &x2 |
| %r8w | |
| %r9 | |

# x86-64 Complete Example

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

## Stack (entry size = 8 bytes)




Remember push instr subtracts 8 from %rsp

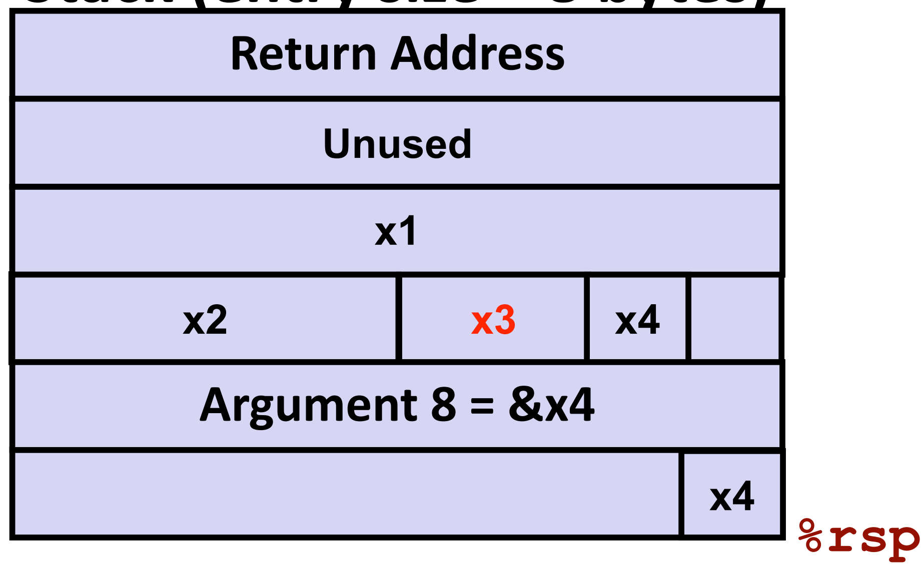
| Register | Use(s) |
|---|---|
| %rdi | |
| %rsi | &x1 |
| %edx | |
| %rcx | &x2 |
| %r8w | |
| %r9 | |

# x86-64 Complete Example

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  subq    $24, %rsp ;make room for local var
  movq    $1, 8(%rsp) ;local variable x1
  movl    $2, 4(%rsp) ;local variable x2
  movw    $3, 2(%rsp) ;local variable x3
  movb    $4, 1(%rsp) ;local variable x4
  leaq    4(%rsp), %rcx ;argument &x2
  leaq    8(%rsp), %rsi ;argument &x1
  leaq    1(%rsp), %rax ;argument &x4
  pushq   %rax          ;push arg &x4 on stack
  pushq   $4            ;push arg x4 on stack
  leaq    18(%rsp), %r9 ;argument &x3
  movl    $3, %r8d      ;argument x3
  movl    $2, %edx      ;argument x2
  movl    $1, %edi      ;argument x1
  movl    $0, %eax
  call    proc
  addq    $40, %rsp ;restore stack pointer
  ret
```

**Stack (entry size = 8 bytes)**



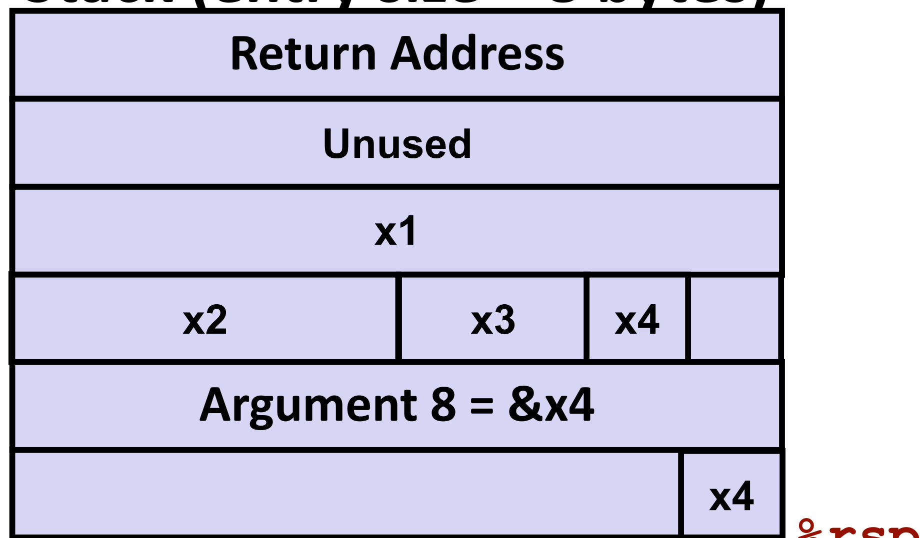

| Register | Use(s) |
|---|---|
| %rdi | |
| %rsi | &x1 |
| %edx | |
| %rcx | &x2 |
| %r8w | |
| %r9 | &x3 |

# x86-64 Complete Example

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

**Stack (entry size = 8 bytes)**



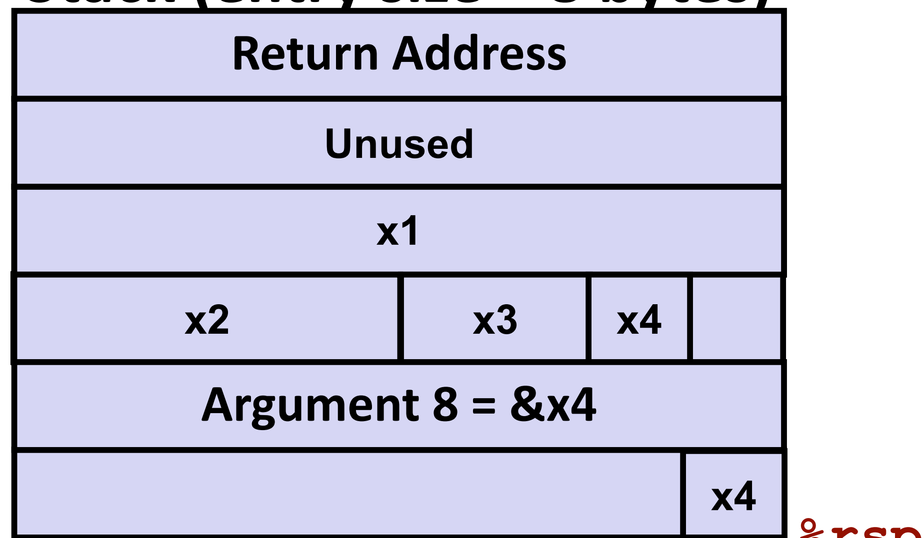

| Register | Use(s) |
|---|---|
| %rdi | x1 = 1 |
| %rsi | &x1 |
| %edx | x2 = 2 |
| %rcx | &x2 |
| %r8w | x3 = 3 |
| %r9 | &x3 |

# x86-64 Complete Example

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

**Stack (entry size = 8 bytes)**



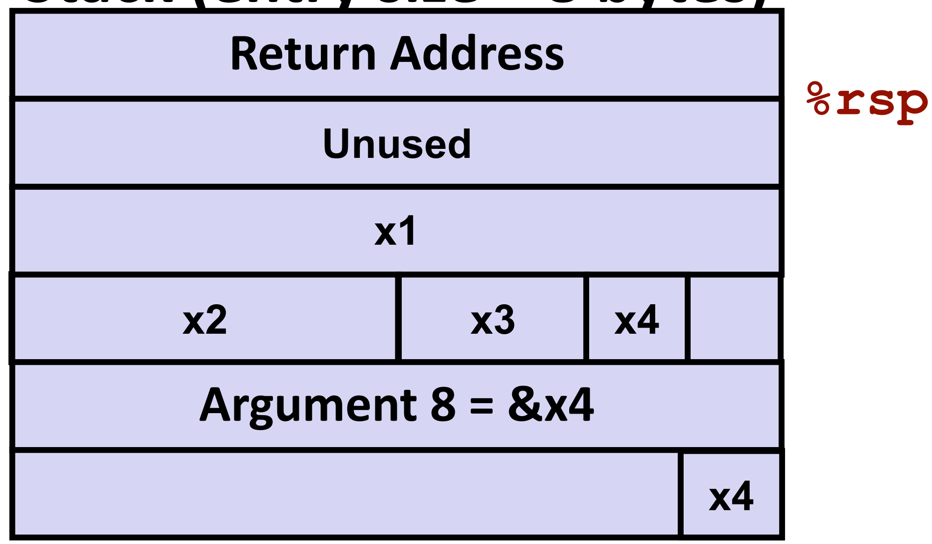

| Register | Use(s) |
|---|---|
| %rdi | x1 |
| %rsi | &x1 |
| %edx | x2 |
| %rcx | &x2 |
| %r8w | x3 |
| %r9 | &x3 |

# x86-64 Complete Example

```
call_proc(){
        long x1 = 1;
        int x2 = 2;
        short x3 = 3;
        char x4 = 4;
        proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
//      return (x1+x2)*(x3-x4);
}
```

```
call_proc:
   subq    $24, %rsp ;make room for local var
   movq    $1, 8(%rsp) ;local variable x1
   movl    $2, 4(%rsp) ;local variable x2
   movw    $3, 2(%rsp) ;local variable x3
   movb    $4, 1(%rsp) ;local variable x4
   leaq    4(%rsp), %rcx ;argument &x2
   leaq    8(%rsp), %rsi ;argument &x1
   leaq    1(%rsp), %rax ;argument &x4
   pushq   %rax          ;push arg &x4 on stack
   pushq   $4            ;push arg x4 on stack
   leaq    18(%rsp), %r9 ;argument &x3
   movl    $3, %r8d      ;argument x3
   movl    $2, %edx      ;argument x2
   movl    $1, %edi      ;argument x1
   movl    $0, %eax
   call    proc
   addq    $40, %rsp ;restore stack pointer
   ret
```

**Stack (entry size = 8 bytes)**




| Register | Use(s) |
|---|---|
| %rdi | x1 |
| %rsi | &x1 |
| %edx | x2 |
| %rcx | &x2 |
| %r8w | x3 |
| %r9 | &x3 |

# Register Saving Conventions

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

```
yoo:
      • • •
    movq $12345, %rdx
    call who
    addq %rdx, %rax
      • • •
    ret
```

```
who:
      • • •
    subq $54321, %rdx
      • • •
    ret
```

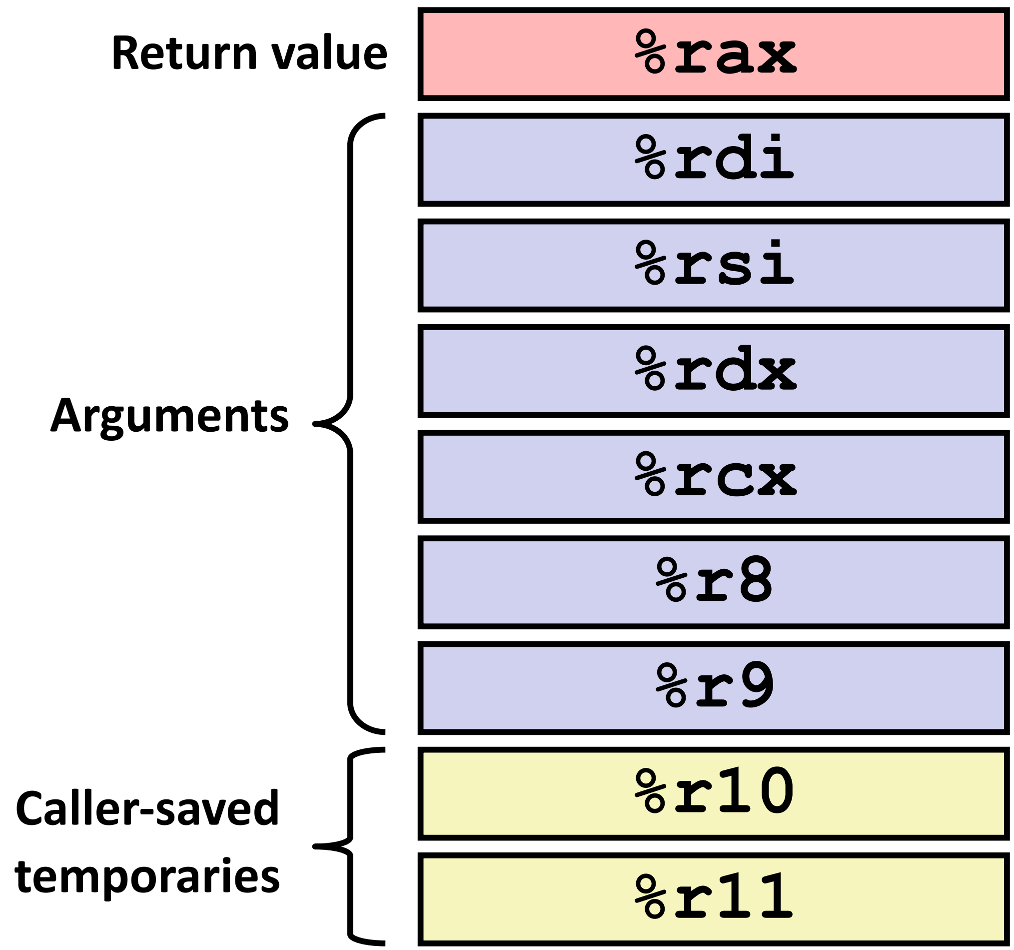
- Contents of register `%rdx` overwritten by `who`
- This could be trouble ➙ something should be done!
  - Need some coordination

# 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 Saved" (aka "Call-Clobbered")***
    - Caller saves temporary values in its frame before the call
  - ***"Callee Saved" (aka "Call-Preserved")***
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller
- **All procedures (including library functions) must follow these conventions**

# x86-64 Linux Register Usage #1

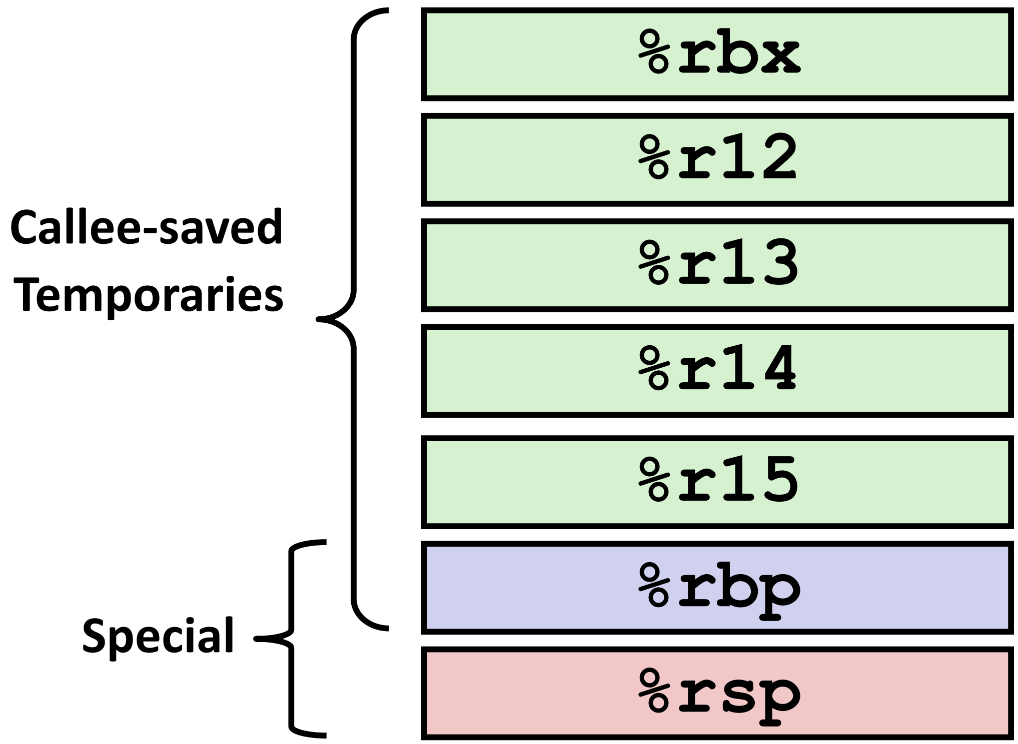
- `%rax`
  - Return value
  - Also **caller-saved**
  - Can be modified by procedure
- `%rdi, ..., %r9`
  - Arguments
  - Also **caller-saved**
  - Can be modified by procedure
- `%r10, %r11`
  - **Caller-saved**
  - Can be modified by procedure




From **callee's** perspective, all these registers can be modified without any issues

# x86-64 Linux Register Usage #2

- **%rbx, %r12, %r13, %r14 , %r15**
  - **Callee-saved**
  - Callee must save & restore
- **%rbp**
  - **Callee-saved**
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- **%rsp**
  - Special form of **callee save**
  - Restored to original value upon exit from procedure




From **caller's** perspective, all these registers are guaranteed to be **unchanged** after the call

# x86-64/Linux Stack Frame (Revisit)

- **Caller Stack Frame**
  - Arguments for this call
  - Return address
    - Pushed by `call` instruction
- **Current Stack Frame**
  - Old frame pointer (**optional**)
  - **Saved register context**
    - **Push old register value**
    - **Make change**
    - **Pop old register value**
  - Local variables
    *if can't keep in registers*
  - "Argument Build"
    Parameters for function about to call