Chapter 3. Assembly (x86-64)

Prof. Jaeseung Choi
Dept. of Computer Science and Engineering
Sogang University

Before We Start

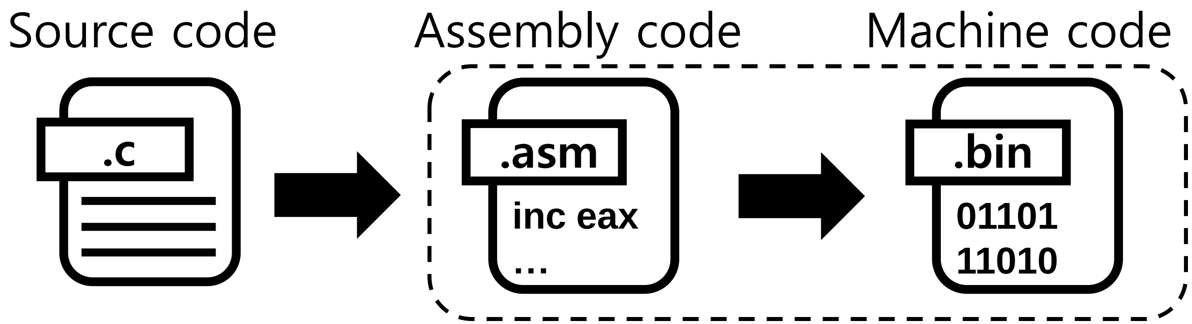
- **This is a summarized version of lecture notes from CSE3030 *(Introduction to Computer Systems)***
- **You don’t have to be an expert in assembly, but certain amount of knowledge is required for this course**
  - Don’t need to memorize all the details in the slide
  - It’s enough if you can use this slide as a reference
  - In the exam, reference sheet (cheating paper) will be given

Topics

- Brief introduction of assembly and Intel x86
- Data representation in CPU and memory
- Basic instructions of x86-64 assembly
- Control instructions of x86-64 assembly
- Function call in x86-64 assembly

Why Learn Assembly?

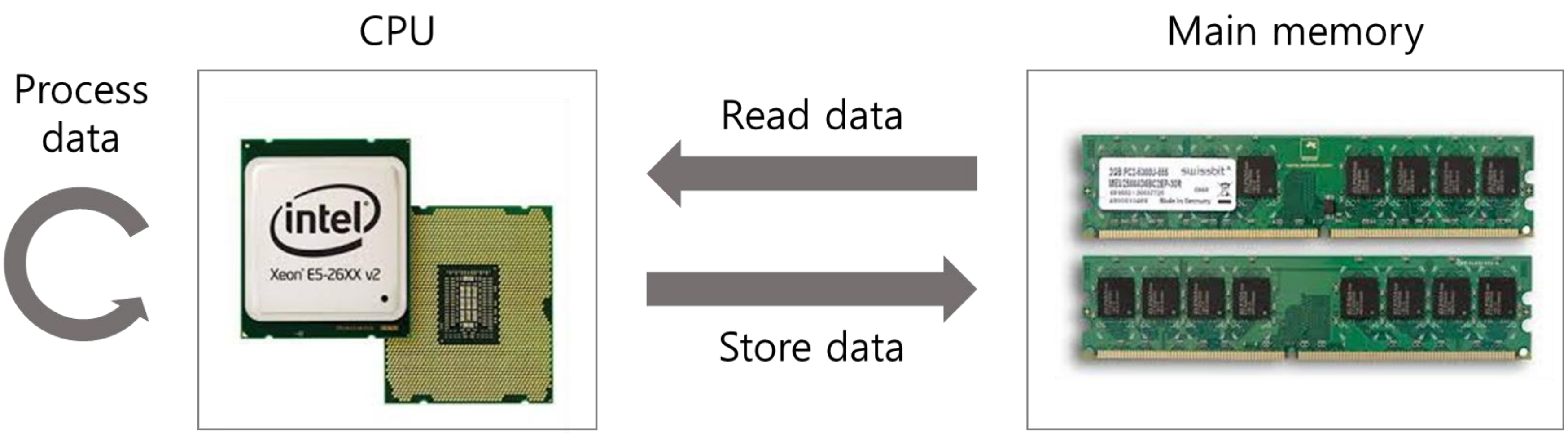
- **When you write and compile a C program, it is translated into assembly code (machine code, to be precise)**
- **This is the form of code that a computer can understand**
- **Therefore, learning assembly is learning how a computer internally operates**



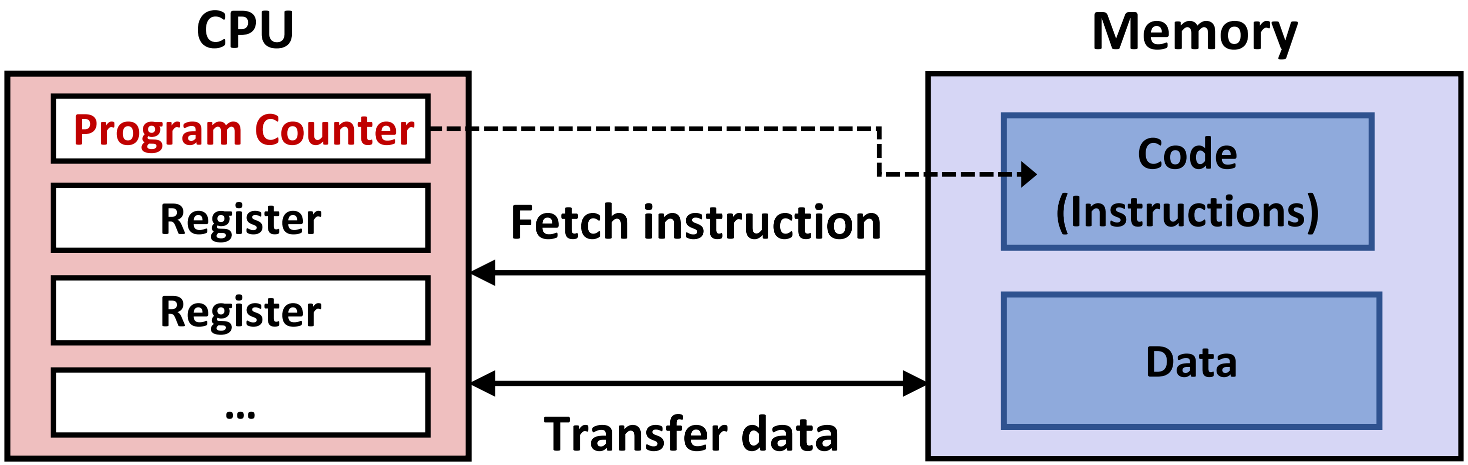
Compile process

Inside Your Computer

- CPU and (Main) Memory: two core components that actually run the program you write
- Assembly code (machine code) controls the operation of these components

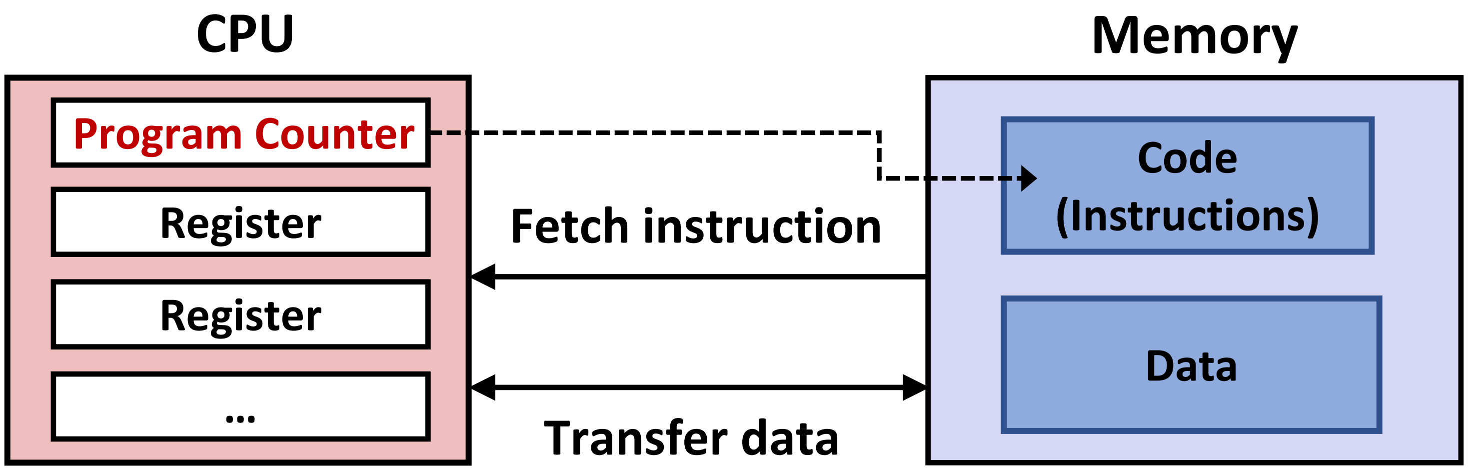


How does CPU work?



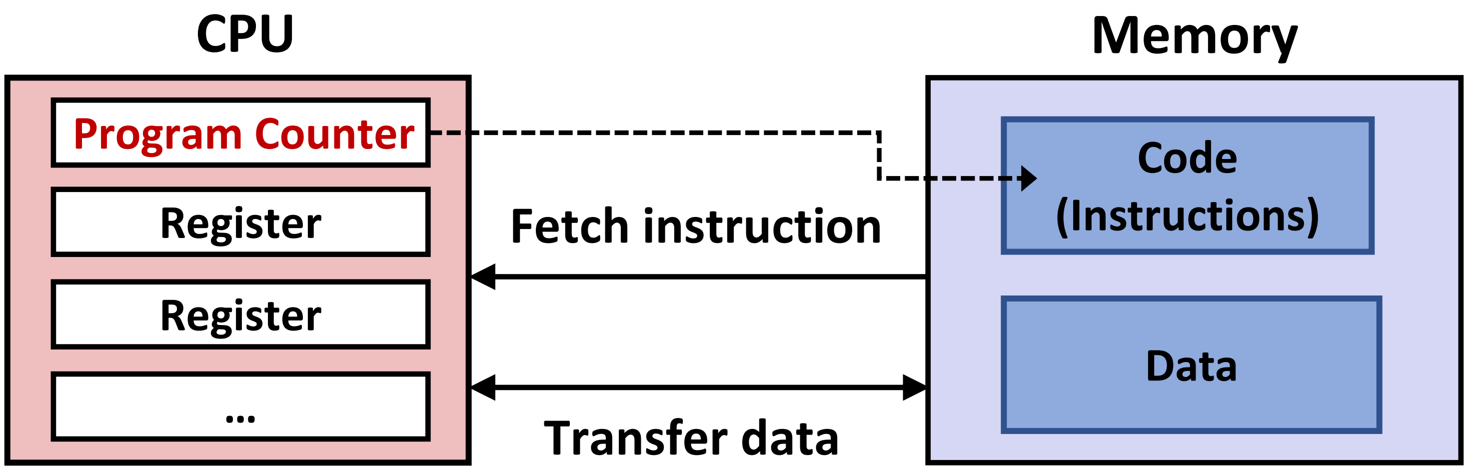
- **Step1. CPU fetches a machine instruction from memory**
  - **Program Counter (PC)** is a special register that contains the address to fetch the instruction from
  - Machine instruction is just a bit sequence with promised meaning

How does CPU work?



- **Step2. The fetched instruction tells CPU what to do**
  - Ex) Add two registers, move data from register to memory, ...
  - **Assembly instruction** is a human-friendly representation of **machine instruction** (there is 1-to-1 mapping between them)

How does CPU work?

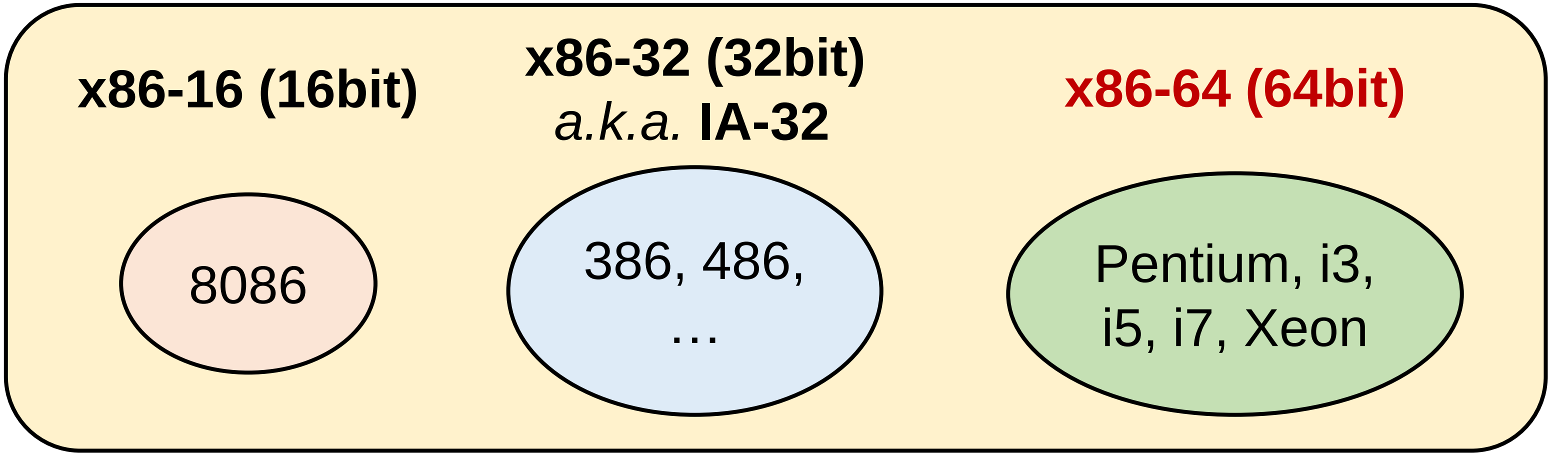


- **Step3. After the execution, PC is automatically updated to point at the next instruction**
  - Certain instruction directly changes the PC
    - Ex) *"Let's jump to address 0x2000"*
  - After the PC is updated, CPU goes back to **Step1** and repeats

What is Intel x86?

- **x86 is a family of CPU architectures developed by Intel**
  - In other words, many architectures belong to this family
  - Series of evolution (new instructions, **increasing word size** ...)
- **This course will focus on x86-64 architecture**
  - Note that x86-64 is the name of assembly language as well

x86 family

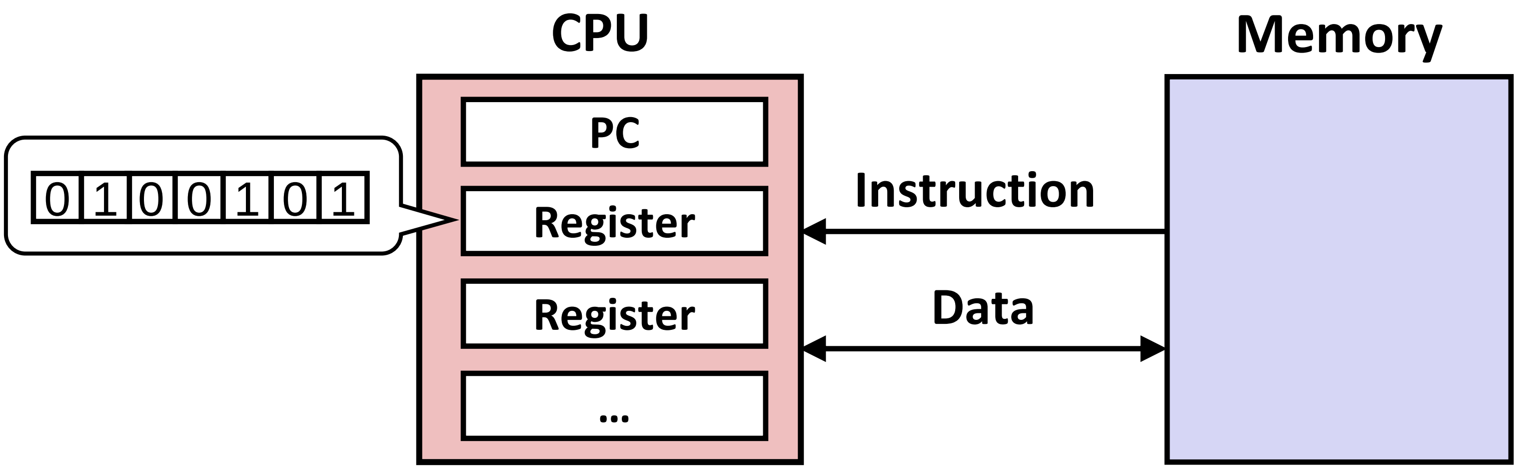


Topics

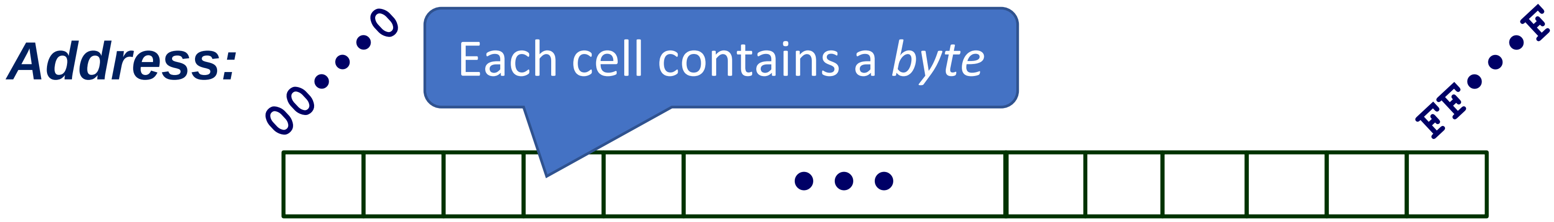
- Brief introduction of assembly and Intel x86
- **Data representation in CPU and memory**
- Basic instructions of x86-64 assembly
- Control instructions of x86-64 assembly
- Function call in x86-64 assembly

Data Representation

- **First of all, everything in computer is stored as bits**
  - Integer, string, code (instructions), etc.
- **Register also contains just a bit sequence (fixed length)**
  - Recall *binary number system, 2's complement system, …*
- **Data representation in memory is similar, but …**
  - We should be careful about **byte ordering (endian issue)**



Basic Structure of Memory



- **Conceptually, memory is a large array of bytes**
  - Each byte space is associated with an address
- **Program accesses memory by using address**
  - Just like using index for an array
  - Program can access multiple bytes at once
    - Ex) Load 4-bytes starting from address `0x200`
  - Not all addresses are used: accessing unused address --> error

Machine Word

- **A computer has a "Word Size"**
  - The data size that your CPU can handle most efficiently
  - First of all, it is the size of a register in CPU
  - Also, it’s maximum data size transferred between CPU & memory
  - At the same time, it’s the *size of a memory address* as well
- **x86-64 has 64-bit word size**
  - Size of an address is 8 bytes: address value can be 0 ~ 2^{64}-1
  - But usually we only use memory space address from 0 to 2^{48}-1

Byte Ordering (Endian)

- **Two conventions when storing multi-byte data (like `int`)**
  - Big endian: Most significant byte stored in the lowest address
  - Little endian: Most significant byte stored in the highest address
  - **x86 family architectures use little endian**
- **Example: Assume C code "`int x = 0x12345678;`"**
  - Here, 0x12 is the most significant byte of **`int x`**
  - Assume that the address returned by "&x" is 0x100

Big Endian

(Address)		0x100	0x101	0x102	0x103		
		12	34	56	78		

Little Endian

(Address)		0x100	0x101	0x102	0x103		
		78	56	34	12		

Checking Byte Order

- **C function to print byte representation of data**
  - This function prints out a sequence of byte
  - By passing a pointer of a variable, we can see its byte pattern

```
void show_bytes(unsigned char* start, size_t len) {
    size_t i;
    for (i = 0; i < len; i++) {
        printf("%p: 0x%.2x\n", start + i, start[i]);
    }
}
```

Checking Byte Order

- **C function to print byte representation of data**
  - This function prints out a sequence of byte
  - By passing a pointer of a variable, we can see its byte pattern

```
int a = 15213;
show_bytes((unsigned char*) &a, sizeof(int));
```

Result (Linux x86-64):

```
0x7ffc99a19b44: 0x6d
0x7ffc99a19b45: 0x3b
0x7ffc99a19b46: 0x00
0x7ffc99a19b47: 0x00
```

15213 = 0x3b6d in hexadecimal

Byte Ordering of Pointer

- **From the viewpoint of CPU, pointer is not so different from integer**
  - It's just a word-size integer that contains a memory address

```
int *p = &a;
show_bytes((unsigned char*) &p, sizeof(int*));
```

Result (Linux x86-64):

```
0x7fff2a742ca0: 0x9c
0x7fff2a742ca1: 0x2c
0x7fff2a742ca2: 0x74
0x7fff2a742ca3: 0x2a
0x7fff2a742ca4: 0xff
0x7fff2a742ca5: 0x7f
0x7fff2a742ca6: 0x00
0x7fff2a742ca7: 0x00
```

The actual address of "a" is 0x7fff2a742c9c

String Representation in Memory

- **String in C**
  - Represented by array of characters
  - Each character is usually encoded in ASCII code
    - Ex) Alphabet 'A' has code 0x41, digit '0' has code 0x30, …
  - String should be null-terminated (null character: ASCII code 0)
- **Same result in both big & little endian system**
  - Byte ordering does not affect string

```
char s[6] = "AB123";
show_bytes((unsigned char*) s, sizeof(s));
```

```
0x7ffcd17a1252: 0x41
0x7ffcd17a1253: 0x42
0x7ffcd17a1254: 0x31
0x7ffcd17a1255: 0x32
0x7ffcd17a1256: 0x33
0x7ffcd17a1257: 0x00
```

Topics

- Brief introduction of assembly and Intel x86
- Data representation in CPU and memory
- **Basic instructions of x86-64 assembly**
- Control instructions of x86-64 assembly
- Function call in x86-64 assembly

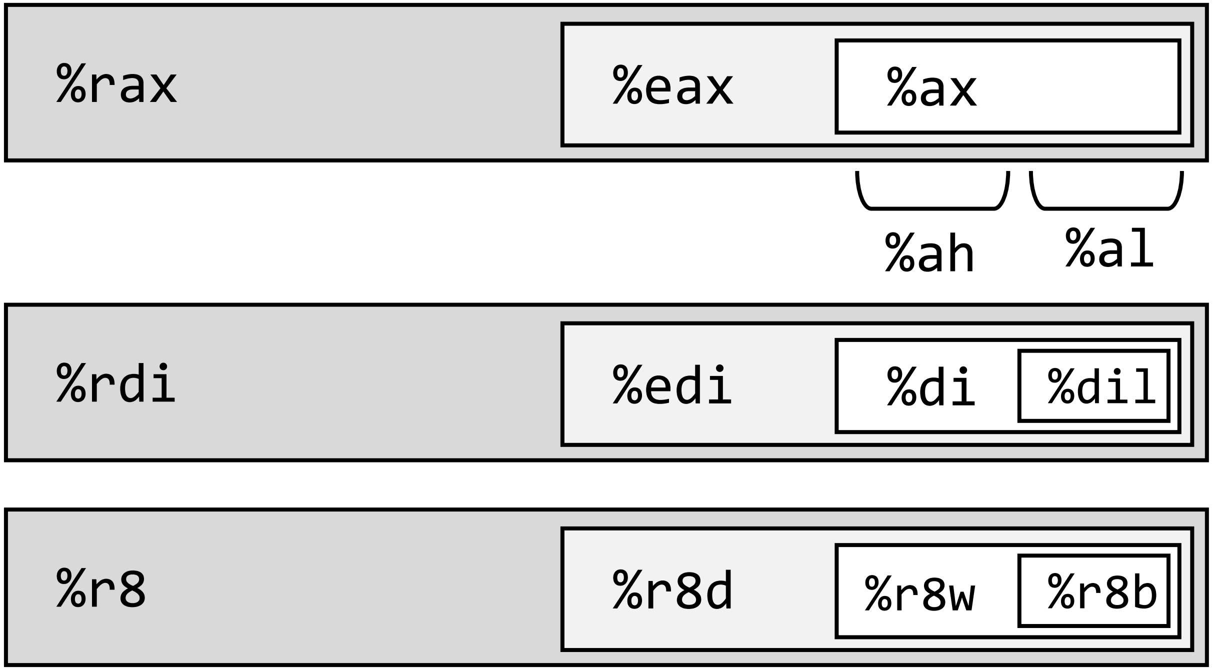
Registers in x86-64

- **Some registers are used for special purpose**
  - **`%rsp`** : stack pointer, **`%rip`** : instruction pointer (program counter)
- **Others are freely usable, but there are some rules**

`%rax`	`%r8`	**`%rip`**
`%rbx`	`%r9`	
`%rcx`	`%r10`	
`%rdx`	`%r11`	
`%rsi`	`%r12`	
`%rdi`	`%r13`	
`%rsp`	`%r14`	
`%rbp`	`%r15`	

Names for Part of Register

- **Each register is 8-byte (e.g., `%rax`), but we can also access its lower 4 bytes (`%eax`), 2 bytes (`%ax`), or 1 byte (`%ah`, `%al`)**
  - Don't feel pressure to memorize these names for now



Similar names for **`%rbx`**, **`%rcx`**, **`%rdx`**

Similar names for **`%rsi`**

Similar names for **`%r9`** - **`%r15`**

What Assembly Looks Like

- **Perform arithmetic operations with registers**
- **Transfer data to and from memory**
- **Variables are mapped to registers or memory slots**
- **Promise (convention)**
  - 1st, 2nd, 3rd … arguments of a function must be passed through **`%rdi`**, **`%rsi`**, **`%rdx`** … registers
  - Return value must be passed through **`%rax`** register

C Code

```
int sum(long x, long y, long *dst)
{
    *dst = x + y;
    return 1;
}
```

x86-64 Assembly Code

```
sum:
   add    %rsi, %rdi
   mov    %rdi, (%rdx)
   mov    $0x1, %eax
   ret
```

Data Move Instruction: `mov`

- **Instruction:** `mov Source, Destination`
  - Ex) **`mov %rax, %rbx`**
  - Sometimes we put a suffix (`movb`, `movw`, `movl`, `movq`)
  - **`b`** = 1 byte, **`w`** = 2 bytes, **`l`** = 4 bytes, **`q`** = 8 bytes
  - We will omit the suffix when it is obvious
- **Operand types (that can come as source or destination)**
  - ***Immediate:*** Constant integer value
    - Example: **`$0x400`**, **`$-533`** ← Note the prefix `'$'`
  - ***Register:*** One of the registers previously discussed
    - Example: **`%rax`**, **`%r13`**
  - ***Memory:*** Consecutive bytes in memory at the specified address
    - Example: **`(%rax)`**, **`0x1000`** ← No prefix `'$'` here

Operand Combinations for `mov`

	Source	Dest	Example	C Analog
`mov`	*Imm*	*Reg*	`mov $0x4,%rax`	`a = 0x4;`
		Mem	`movq $-147,(%rax)`	`*a = -147;`
	Reg	*Reg*	`mov %rax,%rdx`	`d = a;`
		Mem	`mov %rax,(%rdx)`	`*d = a;`
	Mem	*Reg*	`mov (%rax),%rdx`	`d = *a;`
			`mov 0x1000,%rdx`	`d = *(0x1000);`

Cannot perform Mem-to-Mem transfer with a single instruction

Partial Access on Register

- **You can access a register partially**
  - Assume that initial value of `%rax` is `0x1122334455667788`

```
mov $1,  %al     # %rax : 0x1122334455667701
mov $1,  %ax     # %rax : 0x1122334455660001
mov $1,  %eax    # %rax : 0x0000000000000001
```

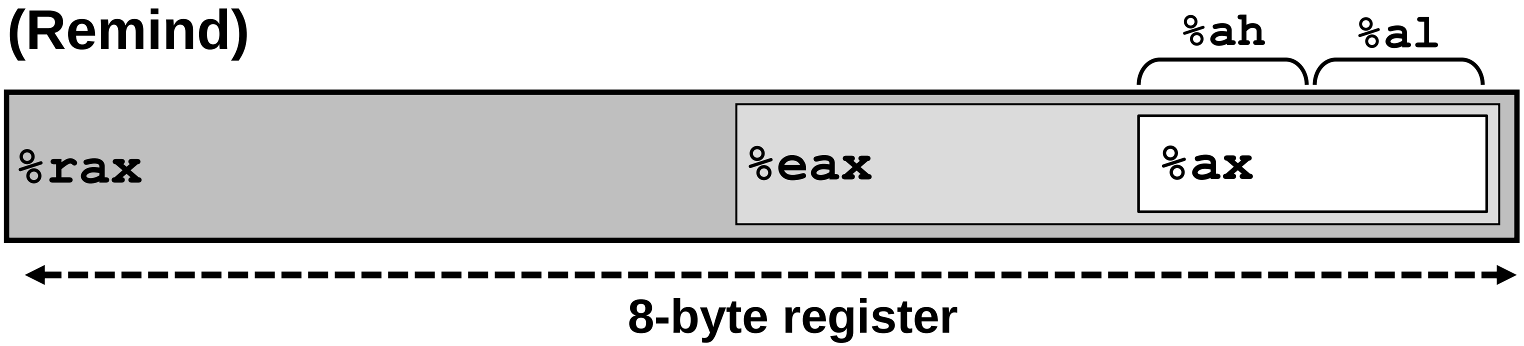
Caution: Set upper bytes to zero

  - Works similarly for other operand combinations

Ex) `mov %bx, %ax` # Set lower 2 bytes of **`%rax`** register

Ex) `mov %ebx, %eax` # Set lower 4 bytes of **`%rax`** & clear upper bytes

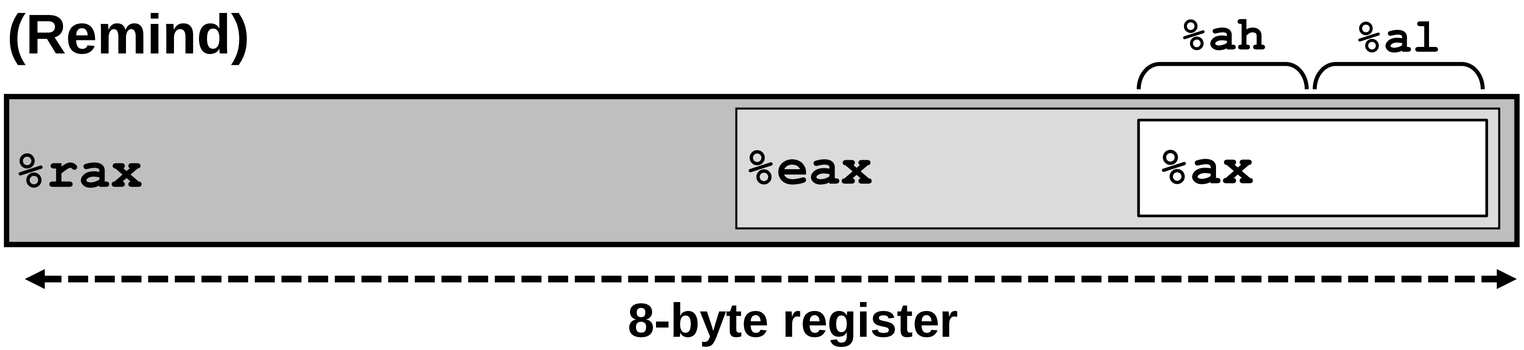
(Remind)



Byte Extension with `movz/movs`

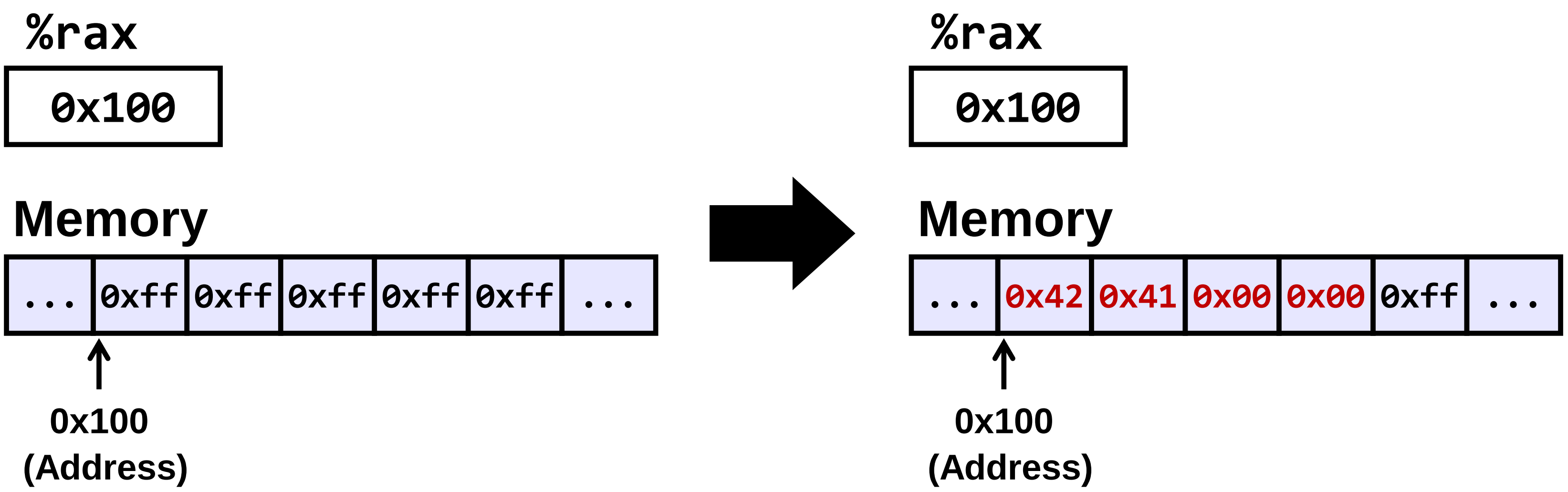
- **Move with z̲ero extension: `movz` *`Source, Reg`***
  - **We can also have suffixes (`b/w/l/q`) here**
  - Ex) `movzbw %bl, %ax` # Zero-extend 1 byte into 2 bytes
- **Move with s̲ign extension: `movs` *`Source, Reg`***
  - Ex) `movslq %ebx, %rax` # Sign-extend 4 bytes into 8 bytes
- **If you don't remember the difference between zero vs. sign extension, check the appendix at the end**

(Remind)



More About Memory Access

- **Here are some more examples of memory access**
- **Ex) `movl $0x4142, (%rax)`**
  - Update the memory address pointed by **`%rax`** with a ***4-byte*** integer **`0x4142`** (suffix '`l`' tells that it's 4-byte data move)
  - In this case, the suffix '`l`' cannot be omitted
  - Note that if you change the suffix, it will behave differently



Complex Memory Access Pattern

- **In x86-64, complex forms of memory operand are allowed**
  - Ex) `mov 0x20(%rbx,%rcx,4), %rax`
  - This reads 8 bytes from address `0x20 + %rbx + %rcx * 4`
  - Scale factor can be one of 1, 2, 4, or 8
  - Useful for *array access* operation

Scale factor

- **Other variants (simplified case of the above form)**
  - Ex) `mov 0x10(%rbx,%rcx), %rax` # Access `0x10 + %rbx + %rcx`
  - Ex) `mov (%rbx,%rcx,4), %rax` # Access `%rbx + %rcx * 4`
  - Ex) `mov (%rbx,%rcx), %rax` # Access `%rbx + %rcx`
  - Ex) `mov 0x1000(%rbx), %rax` # Access `0x1000 + %rbx`
  - Ex) `mov 0x1000(,%rcx,4), %rax` # Access `0x1000 + %rcx * 4`

Arithmetic & Logical Instructions

- **Instructions with two operands:**

Instruction		*Computation*	
`add`	*Src, Dest*	Dest = Dest + Src	# Used for both signed/unsigned
`sub`	*Src, Dest*	Dest = Dest – Src	# Used for both signed/unsigned
`imul`	*Src, Dest*	Dest = Dest * Src	
`shr`	*Src, Dest*	Dest = Dest >> Src	# Logical right shift
`sar`	*Src, Dest*	Dest = Dest >> Src	# Arithmetic right shift
`shl`	*Src, Dest*	Dest = Dest << Src	# Left shift
`xor`	*Src, Dest*	Dest = Dest ^ Src	
`and`	*Src, Dest*	Dest = Dest & Src	
`or`	*Src, Dest*	Dest = Dest | Src	

Ex) **`add %eax, %ebx`** can be thought as **`%ebx += %eax`** in **`C`** syntax.
Also, this clears the upper 4 bytes of **`%ebx`**, like **`mov`** instruction

Arithmetic & Logical Instructions

- **Instructions with one operand:**

Instruction		*Computation*	
`inc`	Dest	Dest = Dest + 1	
`dec`	Dest	Dest = Dest – 1	
`neg`	Dest	Dest = – Dest	
`not`	Dest	Dest = ~Dest	
`shr`	Dest	Dest = Dest >> 1	# Logical right shift by one
`sar`	Dest	Dest = Dest >> 1	# Arithmetic right shift by one
`shl`	Dest	Dest = Dest << 1	# Left shift by one

- **If you don't remember the difference between logical vs. arithmetic shift, check the appendix at the end**

`lea` Instruction

- **`lea` instruction can perform complex computations**
  - Syntax is similar to **`mov`** instruction, but behavior is different
  - Ex) **`lea 0x20(%rbx,%rcx,4), %rax`** : This instruction computes **`0x20` + `%rbx` + `%rcx` * 4** and update **`%rax`** with the result
  - Ex) If **`%rbx`** = **`0x3000`** and **`%rcx`** = **`0x100`**, then the value of **`%rax`** will become **`0x3420`** after executing the above **`lea`** instruction
- **Used for *pointer computation* (originally intended usage) or *integer arithmetic* (abused by compiler developers)**

C code	Assembly
`int* a = &b[c];`	`lea (%rbx,%rcx,4), %rax`
`long a = 3 * b;`	`lea (%rbx,%rbx,2), %rax`

Comparison between `mov` & `lea`

- `mov` **computes an address and accesses the memory with that address;** `lea` **just computes the address**
- **Let's assume that** `%rbx` **=** `0x1000` **and** `%rcx` **=** `0x200`
- `mov 0x8(%rbx,%rcx), %rax` **updates** `%rax` **with the 8-byte value loaded from the memory address** `0x1208`
- `lea 0x8(%rbx,%rcx), %rax` **updates** `%rax` **with** `0x1208`
  - Note that memory access is ***not*** performed here
  - It just performs multiplication and addition

Example

- **Let's review the example code from previous page**
- **(Remind) Promise on register use**
  - 1st, 2nd, 3rd … arguments passed through **`%rdi`**, **`%rsi`**, **`%rdx`** …
  - Return value passed through **`%rax`** register
- **The first instruction (add) computes `x + y`, and the next instruction (`mov`) stores the result into `*dst`**

C Code

```
int sum(long x, long y, long *dst)
{
    *dst = x + y;
    return 1;
}
```

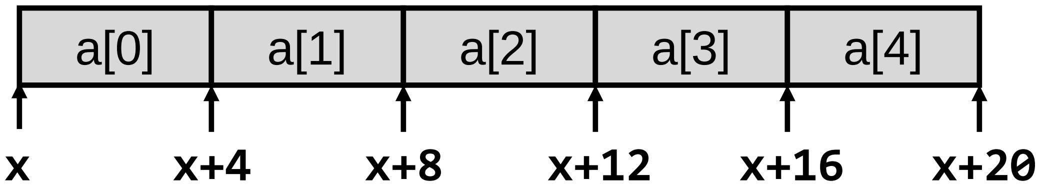
x86-64 Assembly Code

```
sum:
   add    %rsi, %rdi
   mov    %rdi, (%rdx)
   mov    $0x1, %eax
   ret
```

Another Example

- **(Remind) Promise on register use**
  - 1st, 2nd, 3rd … arguments passed through **`%rdi`**, **`%rsi`**, **`%rdx`** …
  - Return value passed through **`%rax`** register
- **Memory layout of a simple 1-dimensional array**

`int a[5];`



C Code

```
int get_elem(int* arr, long idx)
{
    return arr[idx];
}
```

x86-64 Assembly Code

```
get_elem:
  mov (%rdi,%rsi,4), %eax
  ret
```

Topics

- Brief introduction of assembly and Intel x86
- Data representation in CPU and memory
- Basic instructions of x86-64 assembly
- **Control instructions of x86-64 assembly**
- Function call in x86-64 assembly

Control Flow at Assembly Level

- **In assembly, things are translated to conditional jump instructions (similar to using the `goto` syntax in C)**

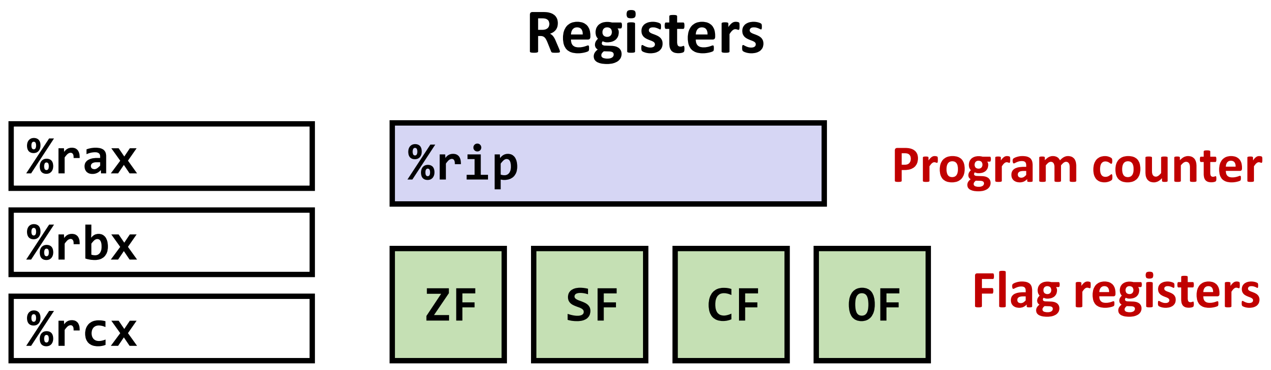
```c
void f(int x) {
    if (x == 1) {
        op1();
    } else {
        op2();
    }
}
```

```
f:  ...
    cmp     $1, %edi
    je      0x100
    call    op2
    jmp     0x105
0x100:
    call    op1
0x105:
    ...
    ret
```

Jump to 0x100 if `%edi == 1`

Flag Registers

- **Flag registers: `%ZF, %SF, %CF, %OF`**
  - Certain instructions automatically update these flag registers
    - Ex) **`add`**, **`sub`**, **`and`**, **`or`**, …
  - Meanwhile, **`mov`** and **`lea`** do **NOT** update flag registers
  - Intuitively, these registers store the result of condition check
- **We will not cover the details of register flag**
  - Ex) Exact rules on how each flag register is updated



Conditional Jump

- **In this course, getting familiar to common pattern is enough**
  - **First**, perform certain operation (**`sub`**, **`cmp`**, **`and`**, **`test`** …)
    - Flag registers are updated based on the result of operation
  - **Then**, run conditional jump instruction (**`je`**, **`jne`**, **`jg`**, **`jl`**, …)
    - Whether to jump or not is decided by the ***status of flag registers***
    - Again, we will not cover the exact rules for making such decision

```
sub %rbx, %rcx
jg  0x100
```

```
cmp %rbx, %rcx
jg  0x100
```

Jump to 0x100 if `%rbx < %rcx`

Think this way: **`sub`** computes **`%rcx - %rbx`**:
`jg` jumps if this result is **g**reater than zero

List of `jxx` Instructions

Instruction	Description
`jmp`	Always jump
`je` (or `jz`)	Jump if equal to zero
`jne` (or `jnz`)	Jump if NOT equal to zero
`js`	Jump if negative (sign check)
`jns`	Jump if zero or positive (sign check)
`jg`	Jump if greater (signed comparison)
`jge`	Jump if greater or equal (signed comparison)
`jl`	Jump if less (signed comparison)
`jle`	Jump if less or equal (signed comparison)
`ja`	Jump if above (unsigned comparison)
`jae`	Jump if above or equal (unsigned comparison)
`jb`	Jump if below (unsigned comparison)
`jbe`	Jump if below or equal (unsigned comparison)

Conditional Jump (Continued)

- **Here are some more common code patterns**
  - For comparison with 0 or sign check, **`and/test`** are often used

```
and %rax, %rax
js  0x100
```

and with itself makes no change

```
test %rax, %rax
js  0x100
```

→ **Jump to 0x100 if `%rax < 0`**

```
cmp $0, %rax
je  0x100
```

```
and %rax, %rax
je  0x100
```

```
test %rax, %rax
je   0x100
```

→ **Jump to 0x100 if `%rax == 0`**

Comparison between `cmp` & `sub`

- **What is the difference between `sub` vs. `cmp`?**
  - And similarly, between `and` vs. `test`?
- **`sub`, `and` update the destination register**
  - Ex) `sub %rbx, %rcx` will change the content of `%rcx`
- **`cmp`, `test` do NOT update the destination register; they only update flag registers**
  - Ex) `cmp %rbx, %rcx` does not change content of `%rcx`; but whether to jump or not is decided by the value of `%rcx - %rbx`
- **In the case of `test %rax, %rax`, there is no essential difference from `and %rax, %rax`**
  - Because the value of `%rax` remains the same anyway

More Conditional Instructions

- **Instructions whose behavior depends on flag registers**
  - Hope you do not meet these instructions, but just in case
- ***`setx Dest`***
  - Set ***`Dest`*** with 1 if condition is satisfied, with 0 otherwise

Instruction	Description
`sete`	Equality check (set if zero/equal)
`...`	...

- ***`cmovx Src, Dest`***
  - Update ***`Dest`*** with ***`Src`*** if condition is satisfied

Instruction	Description
`cmove`	Equality check (move if zero/equal)
`...`	...

Loop in Assembly Code

- **Loop statements (`while`, `for`) can be decomposed into combination of `if` and `goto`**
  - After the decomposition, it can be easily translated to assembly
  - Various translation patterns exist (we will not cover the details)

`while` version

```
while (Test)
  Body
```

`if-goto` version

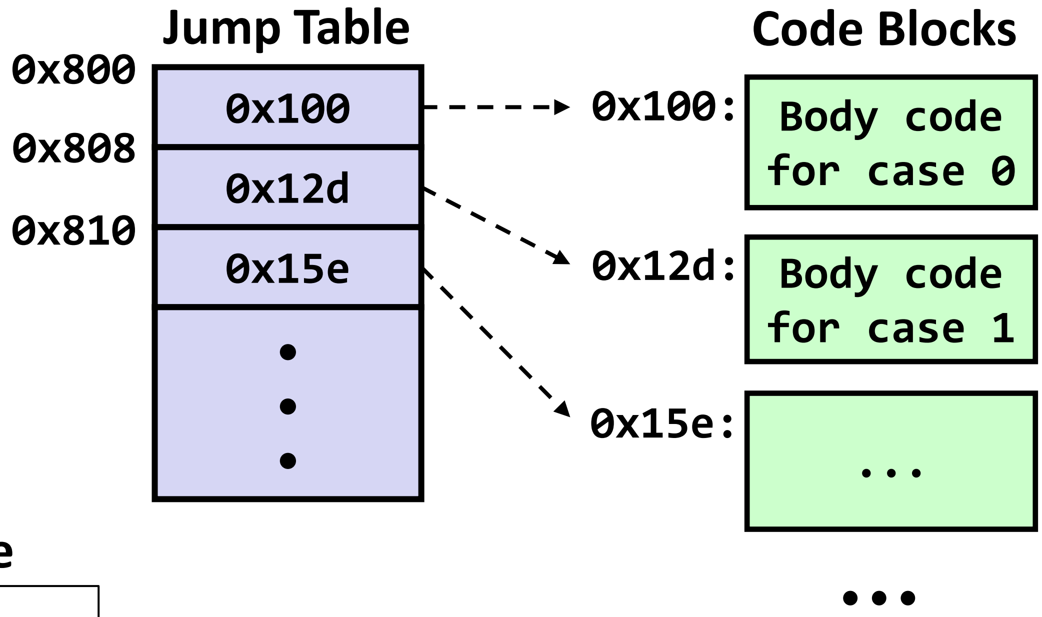
```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
```

Switch in Assembly Code

- **Compiler generates jump table to translate `switch`**
  - But not always (only when it seems to be an efficient solution)

Switch statement

```
switch(x) {
  case 0:
    Body 0
  case 1:
    Body 1
    • • •
}
```



Translated assembly code

```
jmp *0x800(,%rdi,8);
```

Use the content in address **0x800 + %rdi * 8** as jump target

Topics

- Brief introduction of assembly and Intel x86
- Data representation in CPU and memory
- Basic instructions of x86-64 assembly
- Control instructions of x86-64 assembly
- **Function call in x86-64 assembly**

Especially important for learning
buffer overflow

Mechanisms in Function Call

- **Change of control-flow**
  - Call to the entry of a function
  - Return to the call-site
- **Passing data**
  - Function arguments
  - Return value
- **Memory management**
  - Allocate in function entry
  - Deallocate upon return

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

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

Mechanisms in Function Call

- **Change of control-flow**
  - **Call to the entry of a function**
  - **Return to the call-site**
- **Passing data**
  - Function arguments
  - Return value
- **Memory management**
  - Allocate in function entry
  - Deallocate upon return

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

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

Mechanisms in Function Call

- **Change of control-flow**
  - Call to the entry of a function
  - Return to the call-site
- **Passing data**
  - **Function arguments**
  - **Return value**
- **Memory management**
  - Allocate in function entry
  - Deallocate upon return

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

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

Mechanisms in Function Call

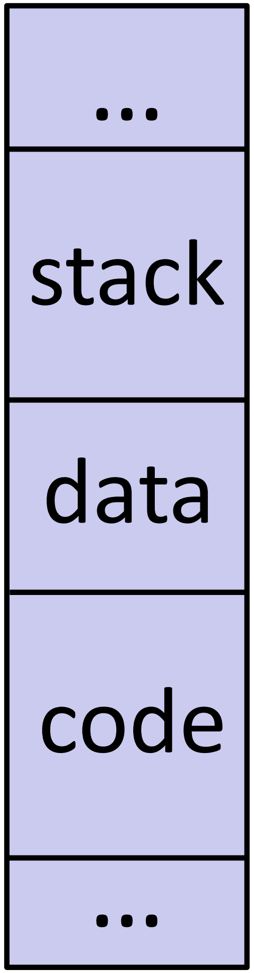
- **Change of control-flow**
  - Call to the entry of a function
  - Return to the call-site
- **Passing data**
  - Function arguments
  - Return value
- **Memory management**
  - **Allocate in function entry**
  - **Deallocate upon return**

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

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

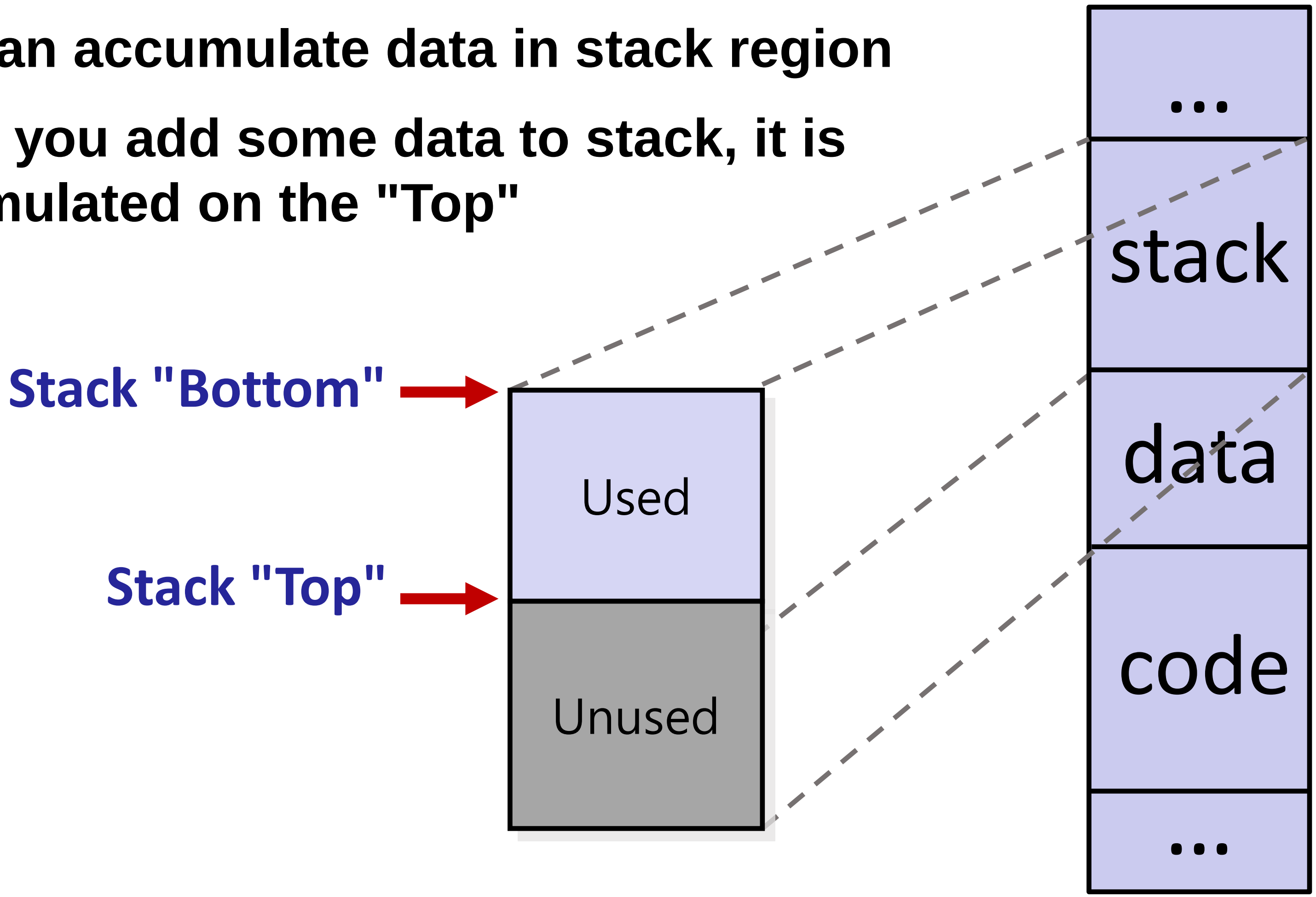
Memory Structure Revisited

- **Memory is viewed as a large array of bytes**
- **Memory can be divided into different regions**
  - Some regions are omitted in this figure
- **Each region is used for different purpose**
  - `Code` region stores your machine instructions
  - `Data` region stores global variables
  - **`Stack`** region is used for executing functions



Stack

- You can accumulate data in stack region
- When you add some data to stack, it is accumulated on the "Top"



Stack Principles

- **Many textbooks draw the memory upside down (just a convention)**
- ***"Bottom"* is assigned high address**
  - When we add an element, stack grows toward lower addresses
- **Stack pointer register `%rsp` points to the element at *"Top"* of the stack**

Registers

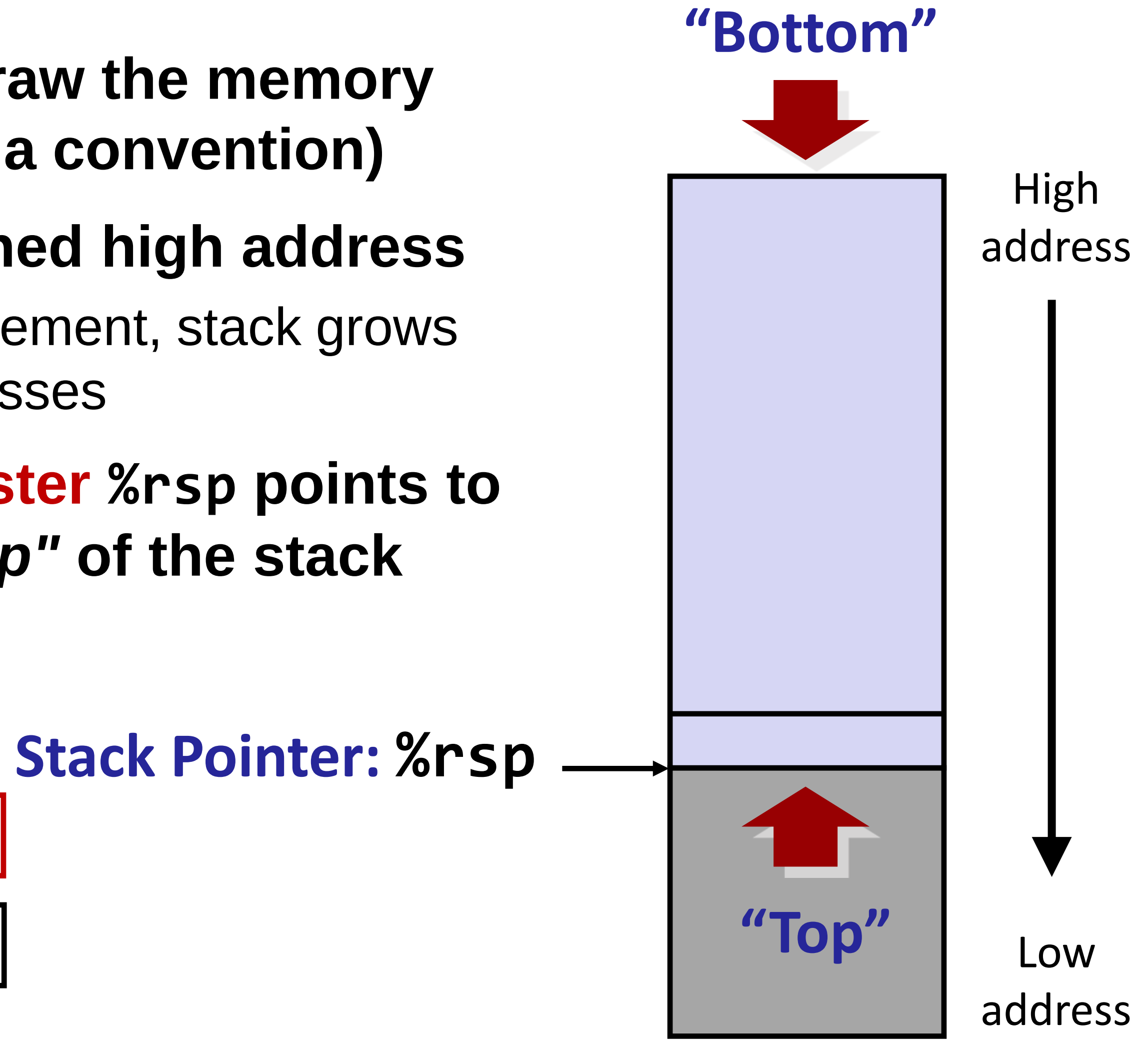
`%rax`

`%rbx`

`%rcx`

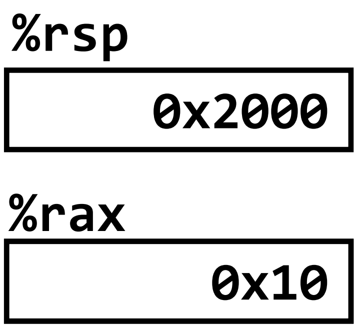
`%rsp`

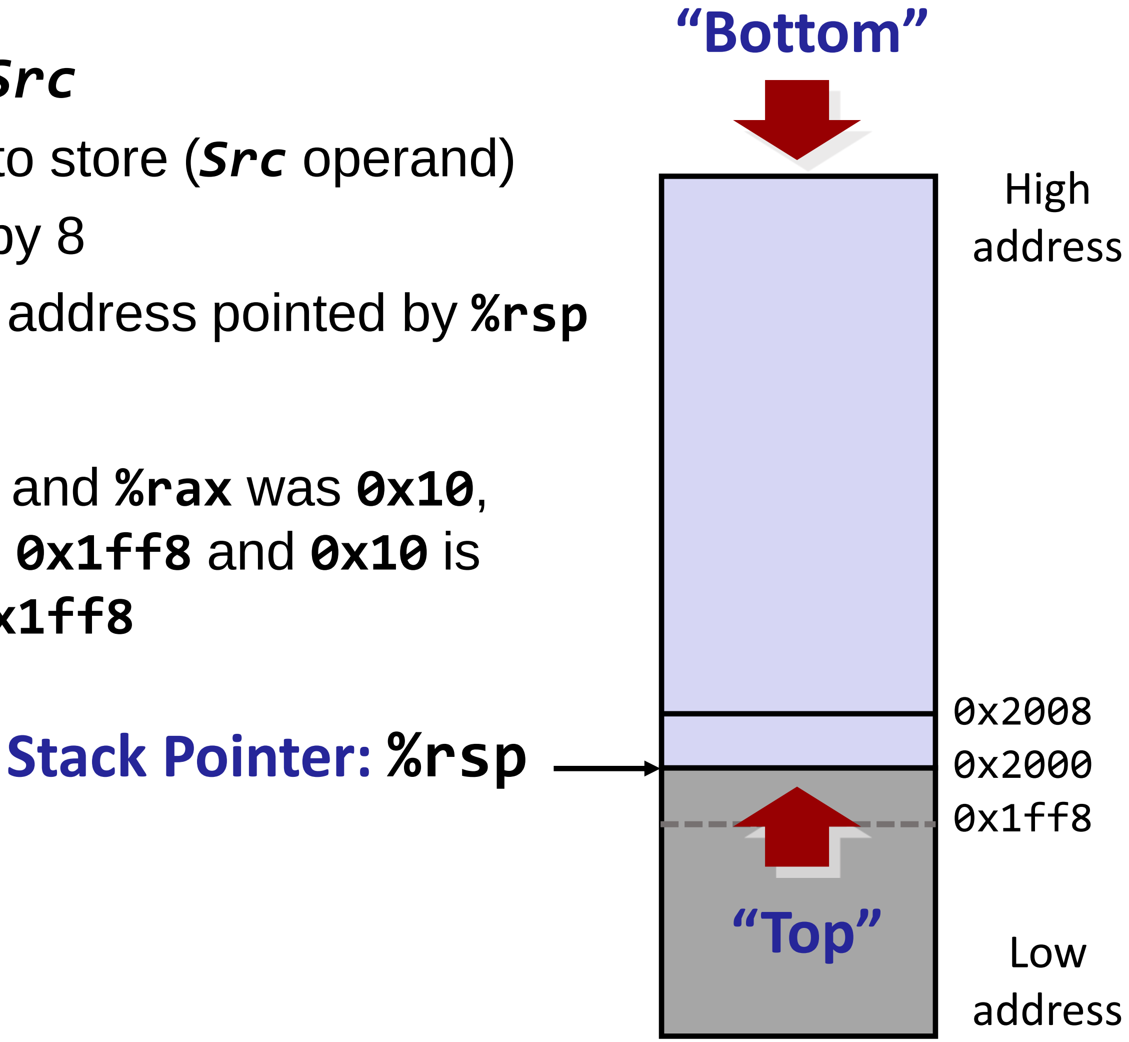
`%rip`



Push Instruction

- **Instruction: `push` *`Src`***
  - (1) Decide the value to store (***`Src`*** operand)
  - (2) Decrement **`%rsp`** by 8
  - (3) Write the value to address pointed by **`%rsp`**
- **Ex) `push %rax`**
  - If **`%rsp`** was **`0x2000`** and **`%rax`** was **`0x10`**, then **`%rsp`** becomes **`0x1ff8`** and **`0x10`** is stored in address **`0x1ff8`**





Push Instruction

- **Instruction: `push` *`Src`***
  - (1) Decide the value to store (***`Src`*** operand)
  - (2) Decrement **`%rsp`** by 8
  - (3) Write the value to address pointed by **`%rsp`**
- **Ex) `push %rax`**
  - If **`%rsp`** was **`0x2000`** and **`%rax`** was **`0x10`**, then **`%rsp`** becomes **`0x1ff8`** and **`0x10`** is stored in address **`0x1ff8`**

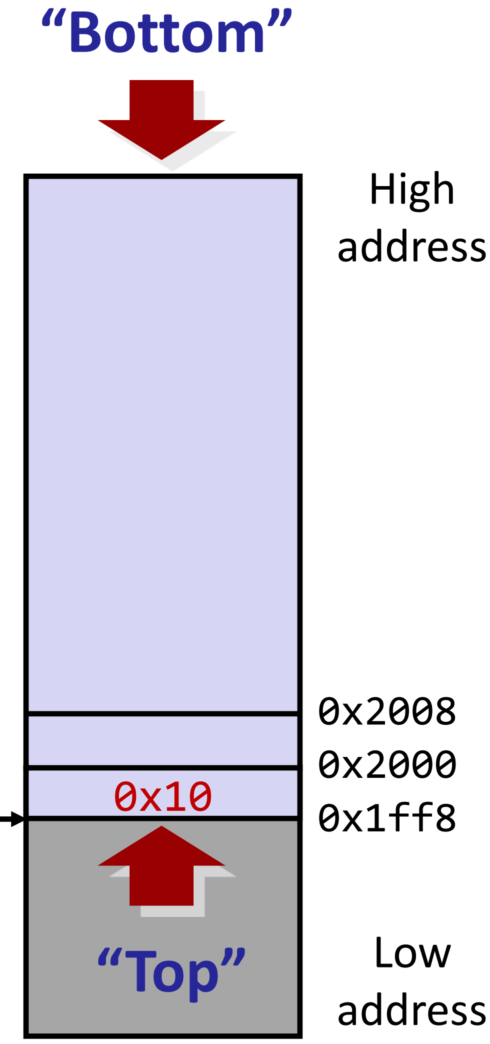
`%rsp`

`0x1ff8`

`%rax`

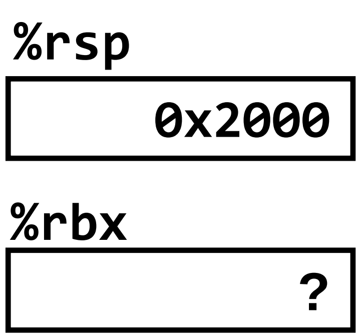
`0x10`

Stack Pointer: `%rsp`

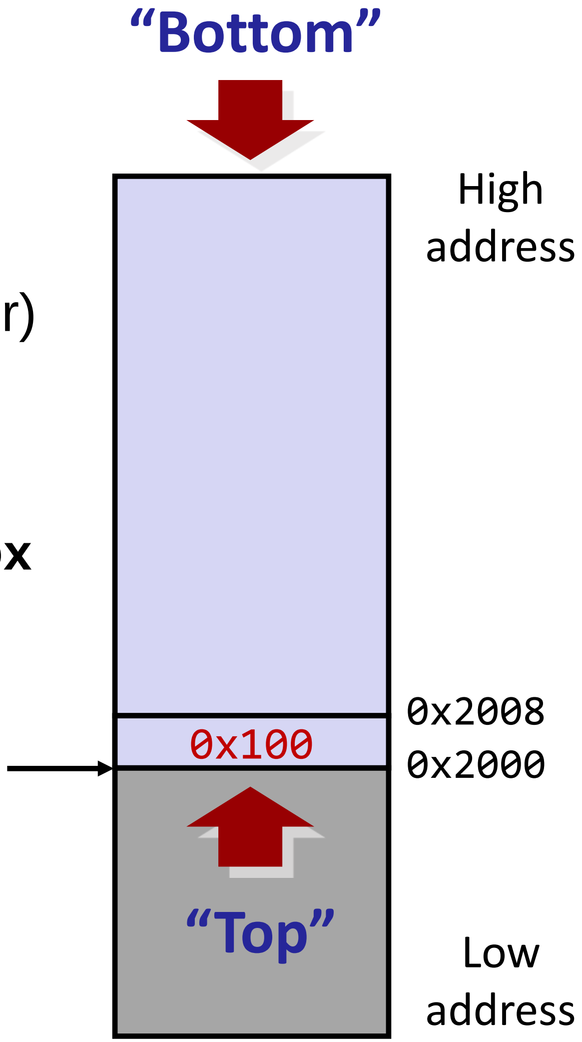


Pop Instruction

- **Instruction: `pop` *`Dest`***
  - (1) Read value at address pointed by **`%rsp`**
  - (2) Increment **`%rsp`** by 8
  - (3) Move the value to *`Dest`* (usually a register)
- **Ex) `pop %rbx`**
  - If **`%rsp`** was **`0x2000`** and **`0x100`** was stored there, then **`%rsp`** becomes **`0x2008`** and **`%rbx`** becomes **`0x100`**



Stack Pointer: `%rsp`



Pop Instruction

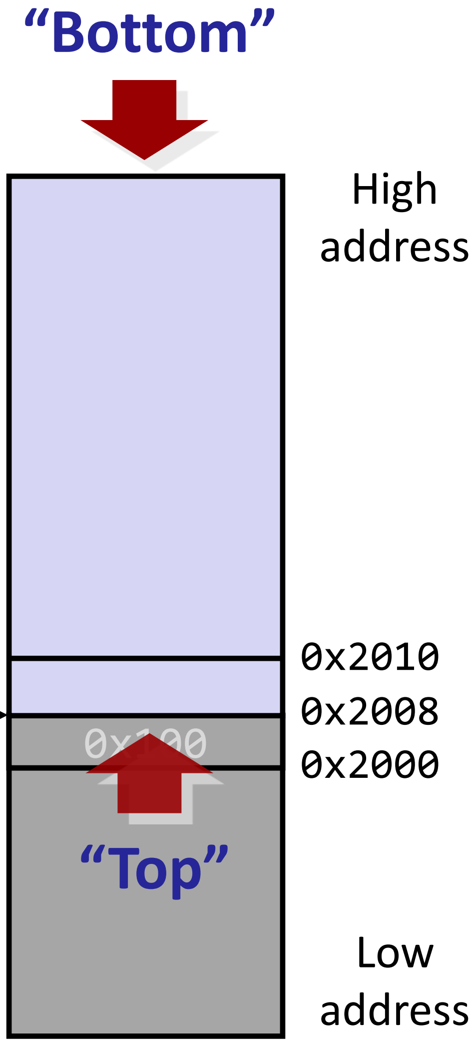
- **Instruction: `pop` *`Dest`***
  - (1) Read value at address pointed by **`%rsp`**
  - (2) Increment **`%rsp`** by 8
  - (3) Move the value to *`Dest`* (usually a register)
- **Ex) `pop %rbx`**
  - If **`%rsp`** was **`0x2000`** and **`0x100`** was stored there, then **`%rsp`** becomes **`0x2008`** and **`%rbx`** becomes **`0x100`**

%rsp
0x2008

%rbx
0x100

Stack Pointer: `%rsp`

`0x100` is still in the memory (but we no longer use it)



Control Flow of Function

- Now let's see how assembly uses stack for function call
- We will use the following example

```
void multstore(long *dest)
{
    long t = mult2(5L, 3L);
    *dest = t;
}
```

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

Control Flow of Function

- Now let’s see how assembly uses stack for function call
- We will use the following example

```
0000000000400536 <multstore>:
  400536:  push    %rbx
  400537:  mov     %rdi,%rbx
  40053a:  mov     $0x3,%esi          # Setup 2nd arg
  40053f:  mov     $0x5,%edi          # Setup 1st arg
  400544:  call    0x400550 <mult2>   # mult2(5,3)
  400549:  mov     %rax,(%rbx)        # Update *dest
  40054c:  pop     %rbx
  40054d:  ret
```

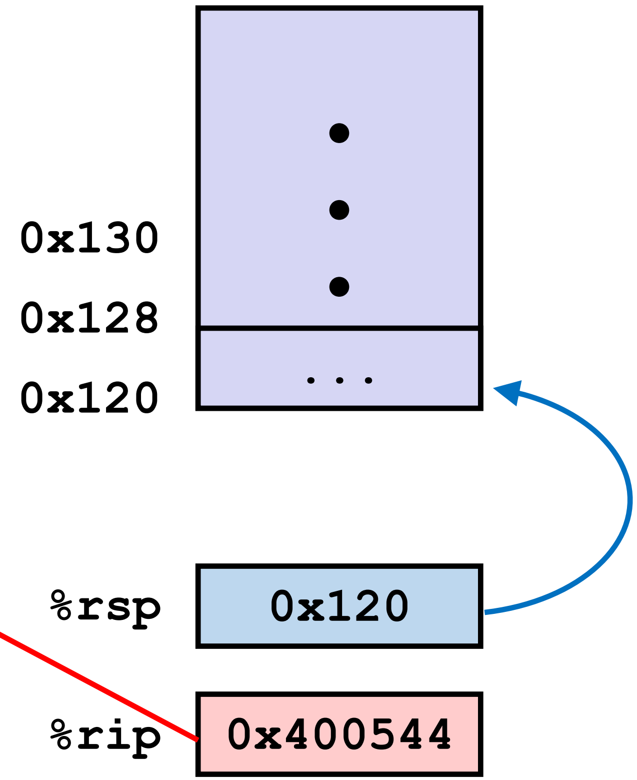
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax        # %rax := a
  400553:  imul   %rsi,%rax        # %rax := a * b
  400557:  ret                     # Return
```

Function Call

- **Instruction: `call` *`Dest`***
  - (1) Push **return address** on stack
    - It means "*where you must return*"
  - (2) Jump to *`Dest`*

```
0000000000400536 <multstore>:
  ...
  400544: call   400550 <mult2>
  400549: mov    %rax,(%rbx)
  ...
```

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

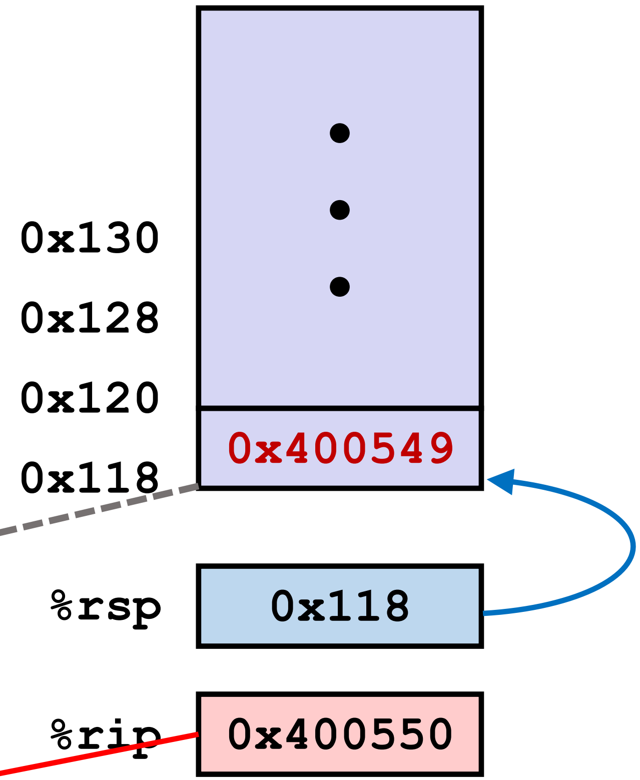


Function Call

- **Instruction: `call` *`Dest`***
  - (1) Push **return address** on stack
    - It means "*where you must return*"
  - (2) Jump to *`Dest`*

```
0000000000400536 <multstore>:
  ...
  400544: call   400550 <mult2>
  400549: mov    %rax,(%rbx)
  ...
```

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

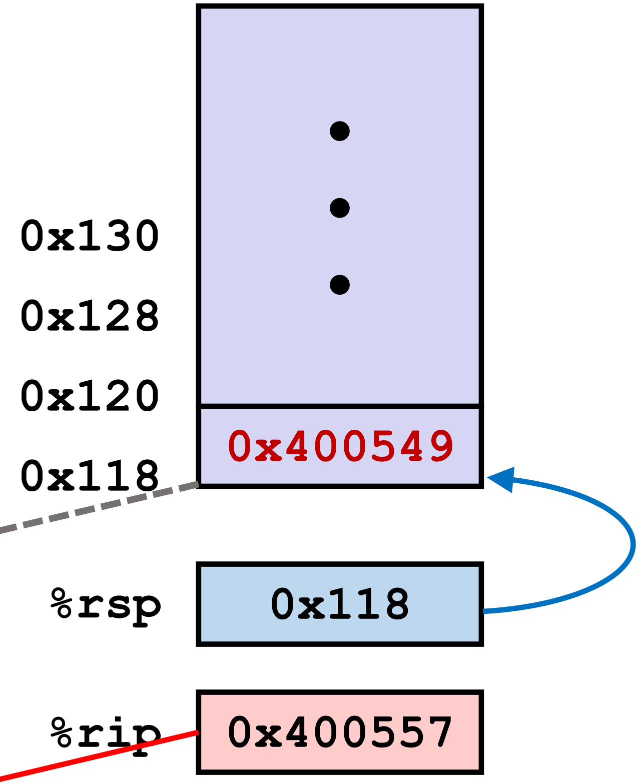


Function Return

- **Instruction: `ret`**
  - (1) Pop a value from stack
  - (2) Jump to the popped value
  - (**`ret`** is equivalent to **`pop %rip`**)

```
0000000000400536 <multstore>:
  ...
  400544: call   400550 <mult2>
  400549: mov    %rax,(%rbx)
  ...
```

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

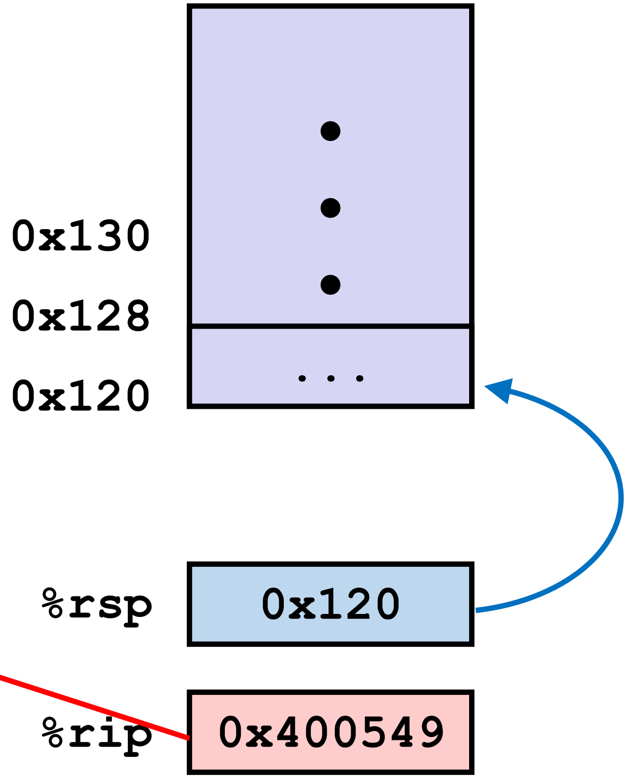


Function Return

- **Instruction: `ret`**
  - (1) Pop a value from stack
  - (2) Jump to the popped value
  - (**`ret`** is equivalent to **`pop %rip`**)

```
0000000000400536 <multstore>:
  ...
  400544: call   400550 <mult2>
  400549: mov    %rax,(%rbx)
  ...
```

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



Calling Convention: Passing Data

- **How can we pass data (arguments and return value) between functions? By making some promise!**
  - First 6 arguments in register (if there are more arguments, pass them through stack)

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

  - Return value in `%rax` register

    `%rax`

Data Passing in `multstore`

```
void multstore(long *dest) {
    long t = mult2(5L, 3L);
    *dest = t;
}
```

```
0000000000400536 <multstore>:
  # At entry, dest is passed through %rdi
  ...
  400537:  mov     %rdi,%rbx          # Backup 'dest' to %rbx
  40053a:  mov     $0x3,%esi          # Setup 2nd arg
  40053f:  mov     $0x5,%edi          # Setup 1st arg
  400544:  call    400550 <mult2>     # %rax = mult2(5,3)
  400549:  mov     %rax,(%rbx)        # Update *dest
  ...
```

`long mult2(long a, long b)`

```
<mult2>:
400550:  mov    %rdi,%rax
400553:  imul   %rsi,%rax
400557:  ret
```

Closer Look on `multstore`

- This function back up `%rbx` register on stack at entry
- Then, the register value is restored before the return
- But why only `%rbx`, and not `%rdi` or `%rsi`?

```
0000000000400536 <multstore>:
  400536:  push    %rbx
  400537:  mov     %rdi,%rbx
  40053a:  mov     $0x3,%esi          # Setup 2nd arg
  40053f:  mov     $0x5,%edi          # Setup 1st arg
  400544:  call    0x400550 <mult2>  # mult2(5,3)
  400549:  mov     %rax,(%rbx)        # Update *dest
  40054c:  pop     %rbx
  40054d:  ret
```

Calling Convention: Who saves?

- **(Note) When `f` calls `g`: `f` is caller, `g` is callee**
- **Caller-saved registers**
  - Callee can freely update these registers
  - If caller doesn't want such changes, ***caller must save*** them before making a call
  - Ex) **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8 ~ %r11 ...`**
- **Callee-saved registers**
  - Callee should guarantee that the values of these registers remain the same at the entry and exit
  - If callee is going to use these registers in its body, ***callee must save*** and restore them before return
  - Ex) **`%rbx`**, **`%r12 ~ %r14 ...`**

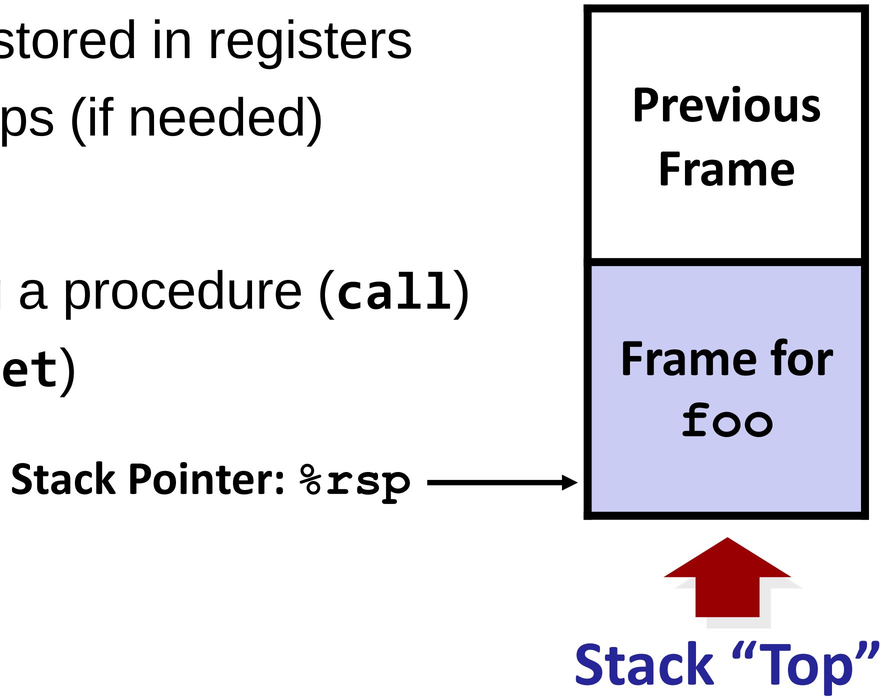
Register Save in `multstore`

- `%rbx` is callee-saved, while others are caller-saved
- `multstore` saves `%rbx` at entry and restores it before `ret`
- Also, `multstore` knows that `%rbx` will remain the same before and after the call of `mult2`

```
0000000000400536 <multstore>:
  400536:  push    %rbx
  400537:  mov     %rdi,%rbx
  40053a:  mov     $0x3,%esi          # Setup 2nd arg
  40053f:  mov     $0x5,%edi          # Setup 1st arg
  400544:  call    0x400550 <mult2> # mult2(5,3)
  400549:  mov     %rax,(%rbx)        # Update *dest
  40054c:  pop     %rbx
  40054d:  ret
```

Stack Frame

- **Stack can be divided into subregions called *frames***
- **Each frame stores the state of executing function**
  - Saved return address
  - Local variables (if needed)
    - So far, all variables were stored in registers
  - Callee-saved register backups (if needed)
- **Management**
  - Allocated right after entering a procedure (`call`)
  - Deallocated before return (`ret`)



Call Chain and Stack Frame

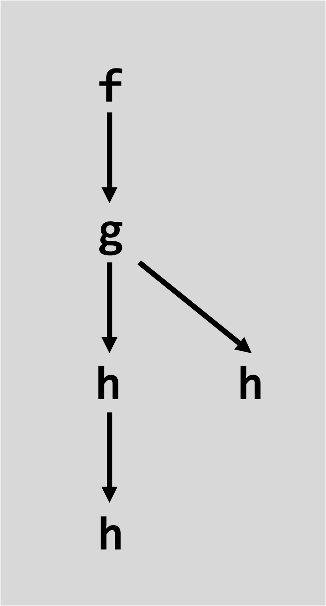
```
f(…)
{
  •
  •
  g();
  •
  •
}
```

```
g(…)
{
  • • •
  h();
  • • •
  h();
  • • •
}
```

h() has recursion

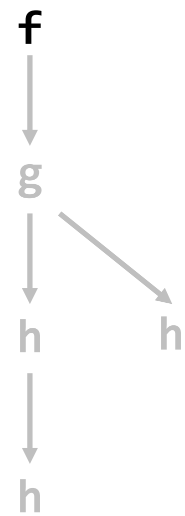
```
h(…)
{
  •
  if(...)
    h();
  •
  •
}
```

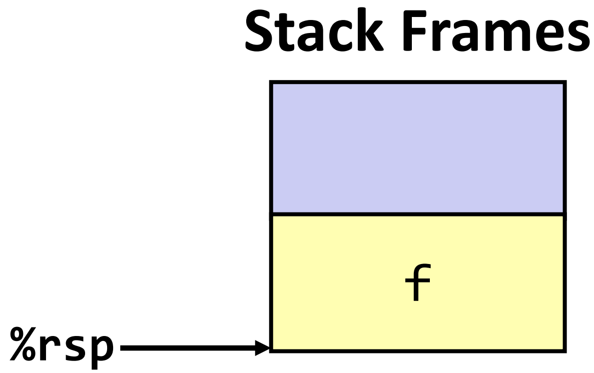
Call Chain (Example)



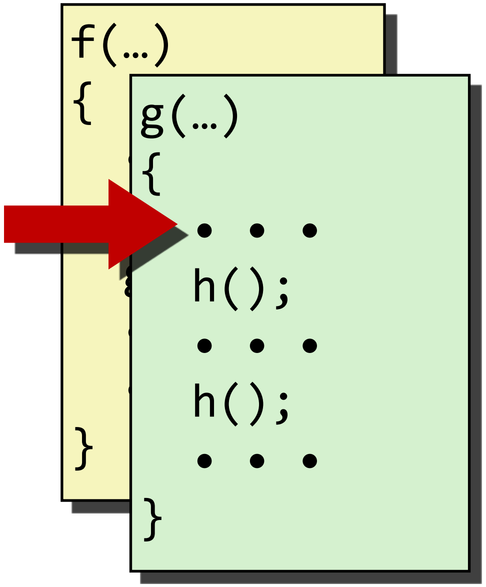
Call Chain and Stack Frame

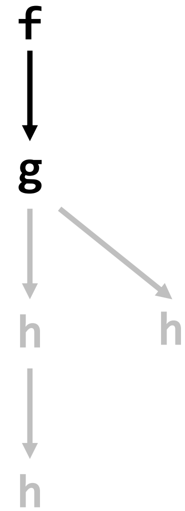
```
f(…)
{
  •
  •
  g();
  •
  •
}
```

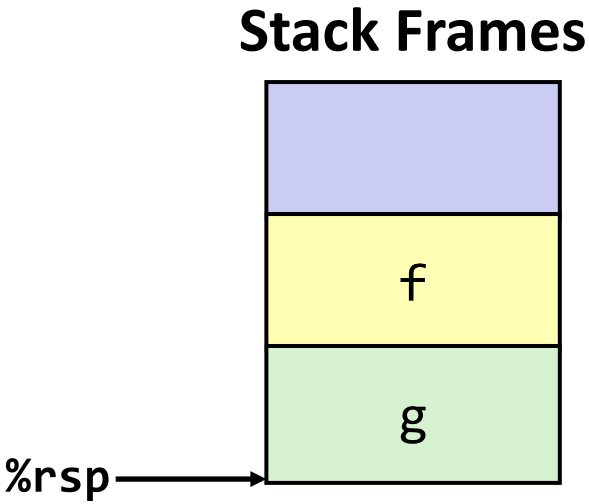




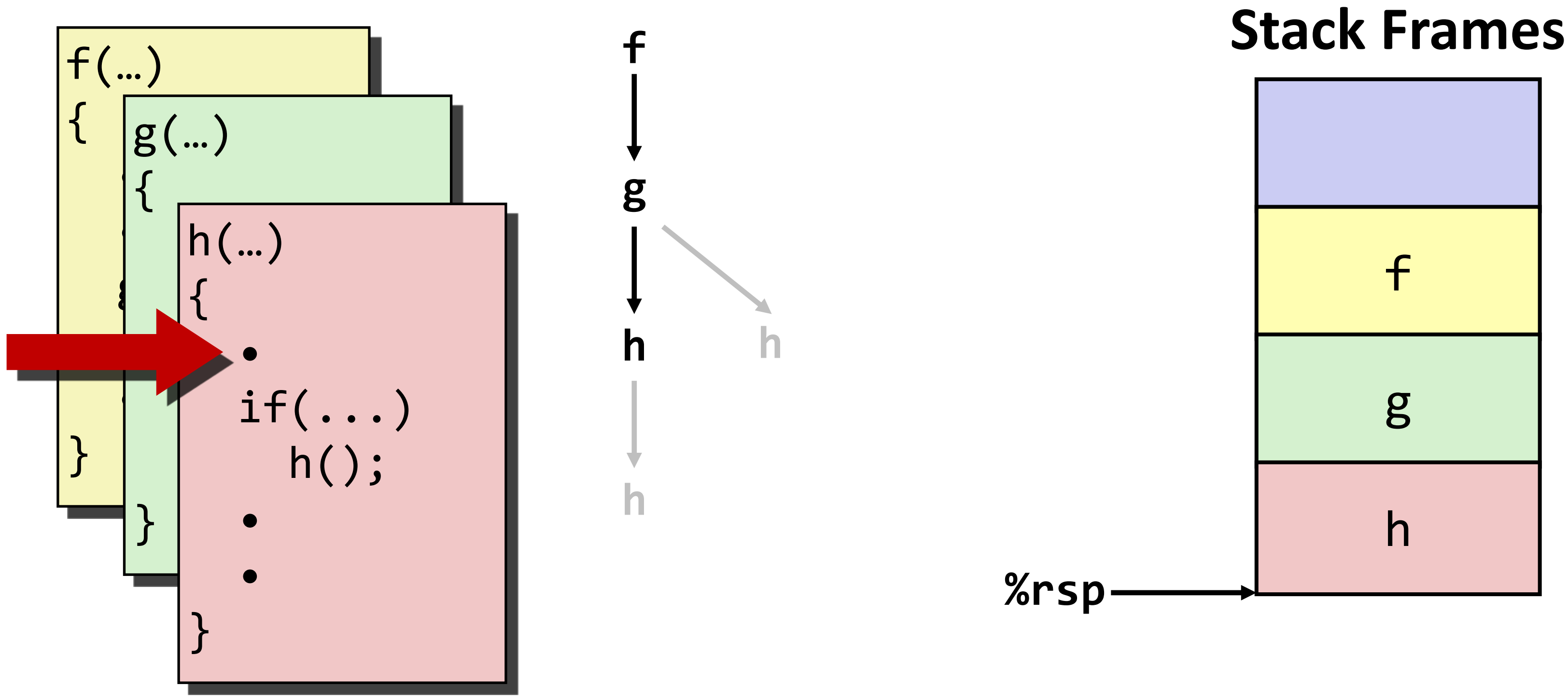
Call Chain and Stack Frame



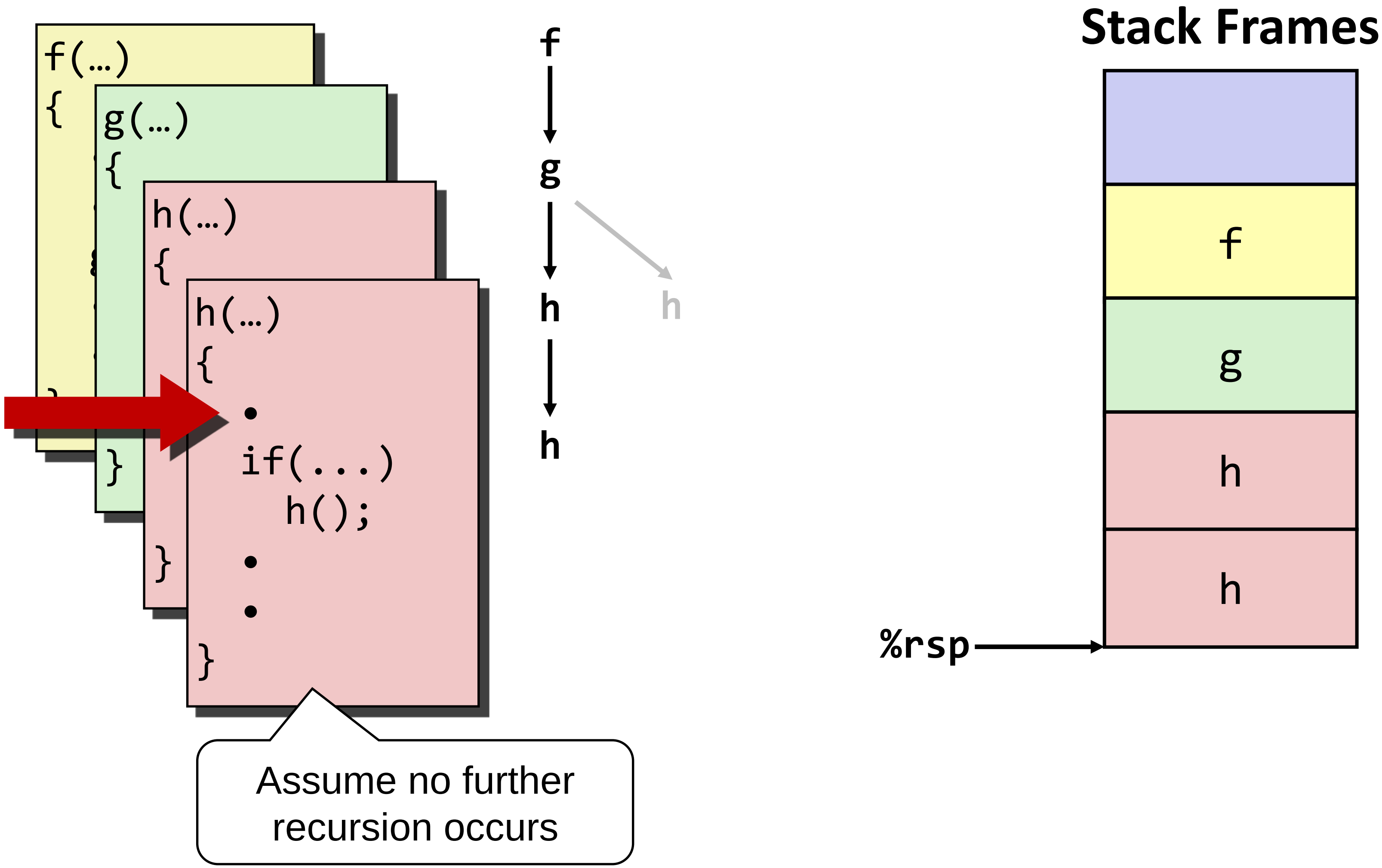




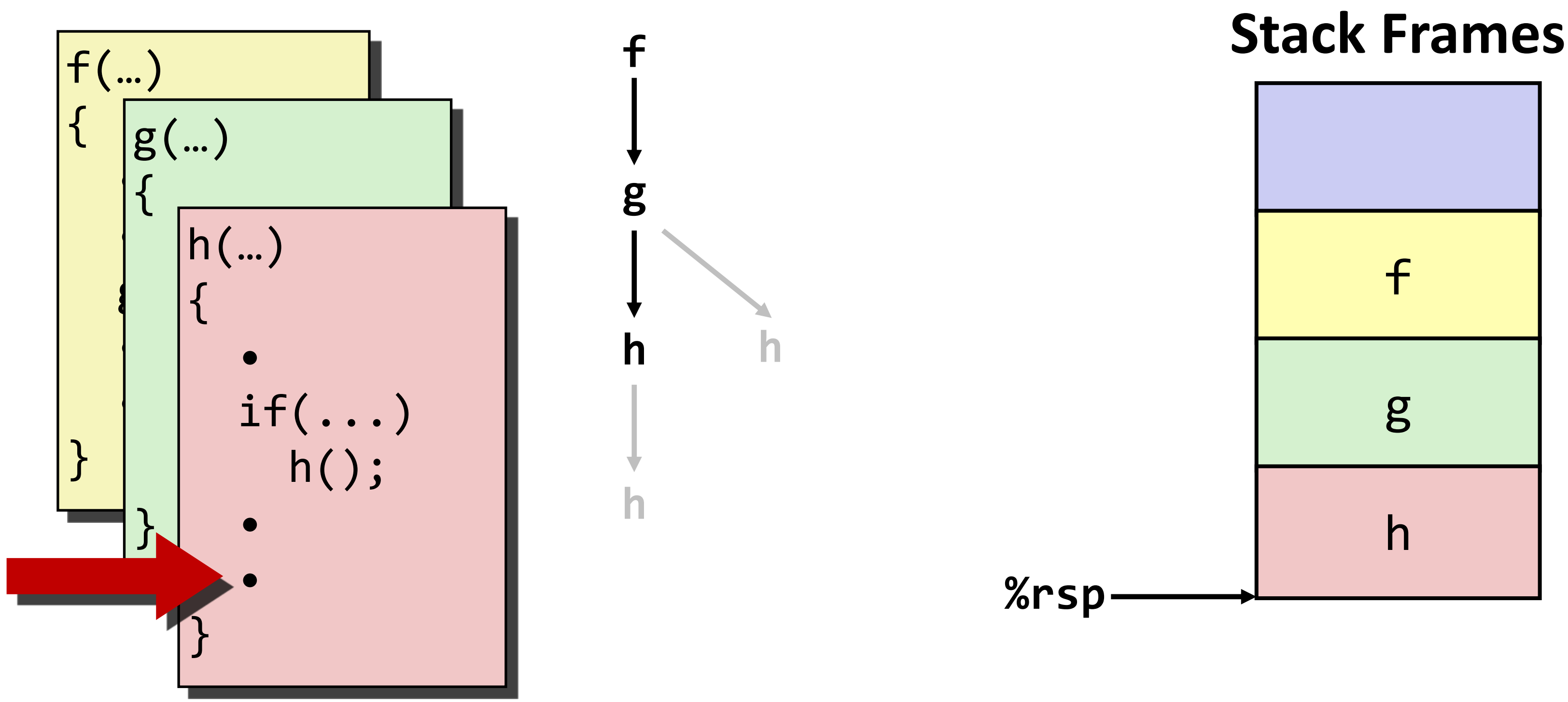
Call Chain and Stack Frame



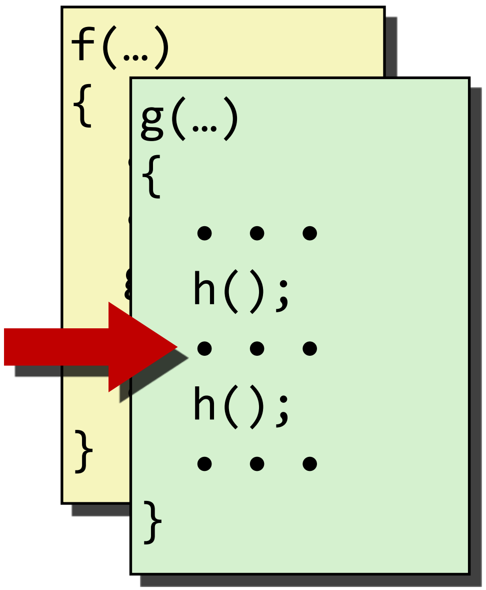
Call Chain and Stack Frame

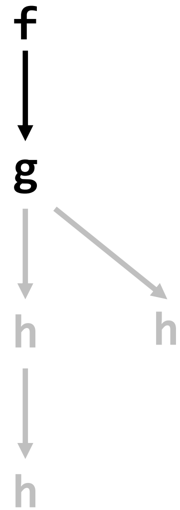


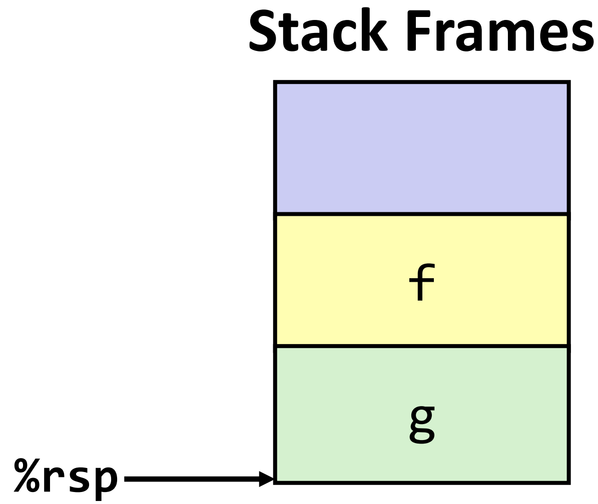
Call Chain and Stack Frame



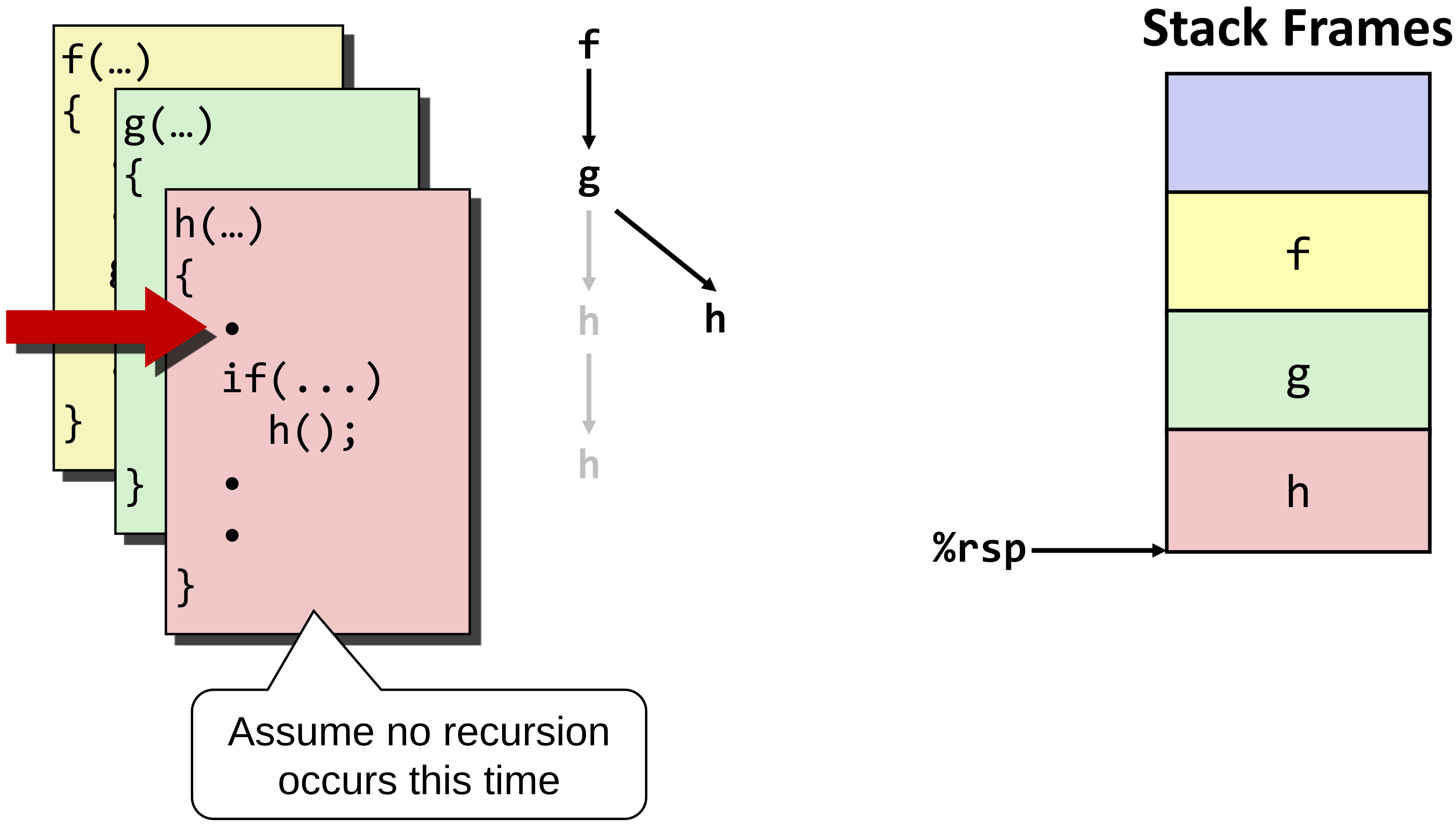
Call Chain and Stack Frame



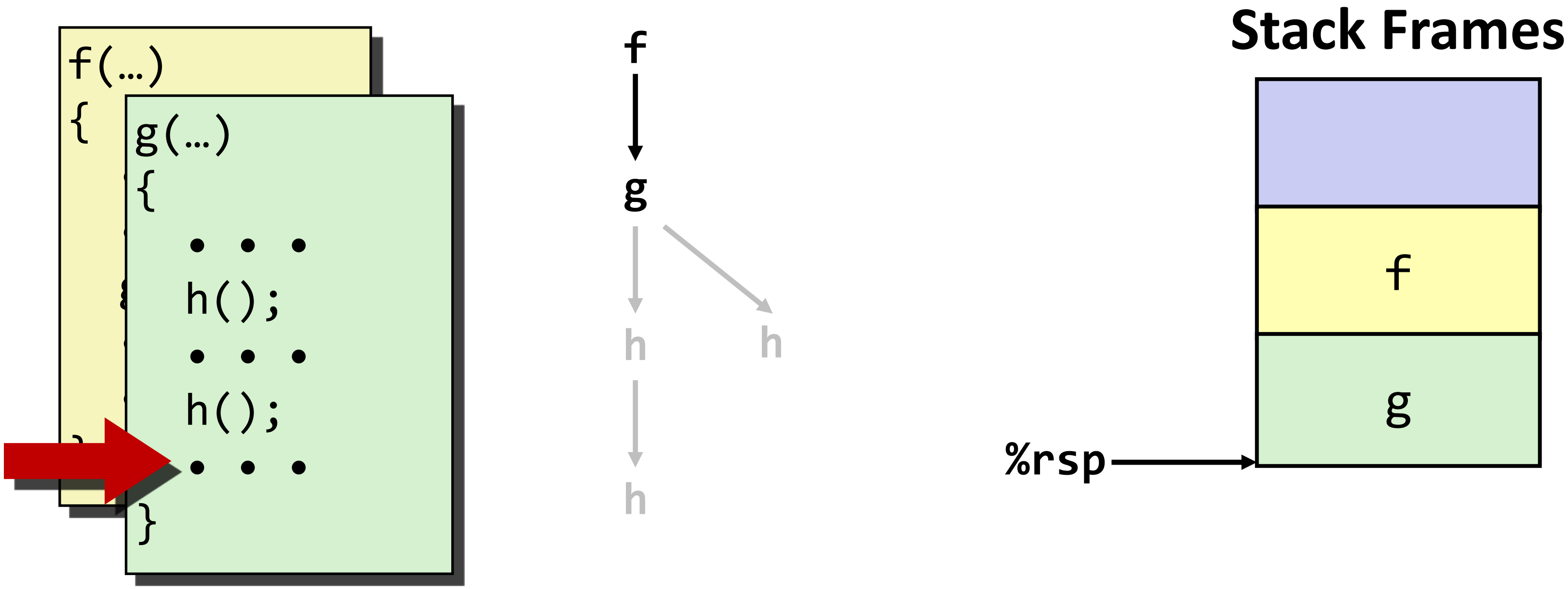




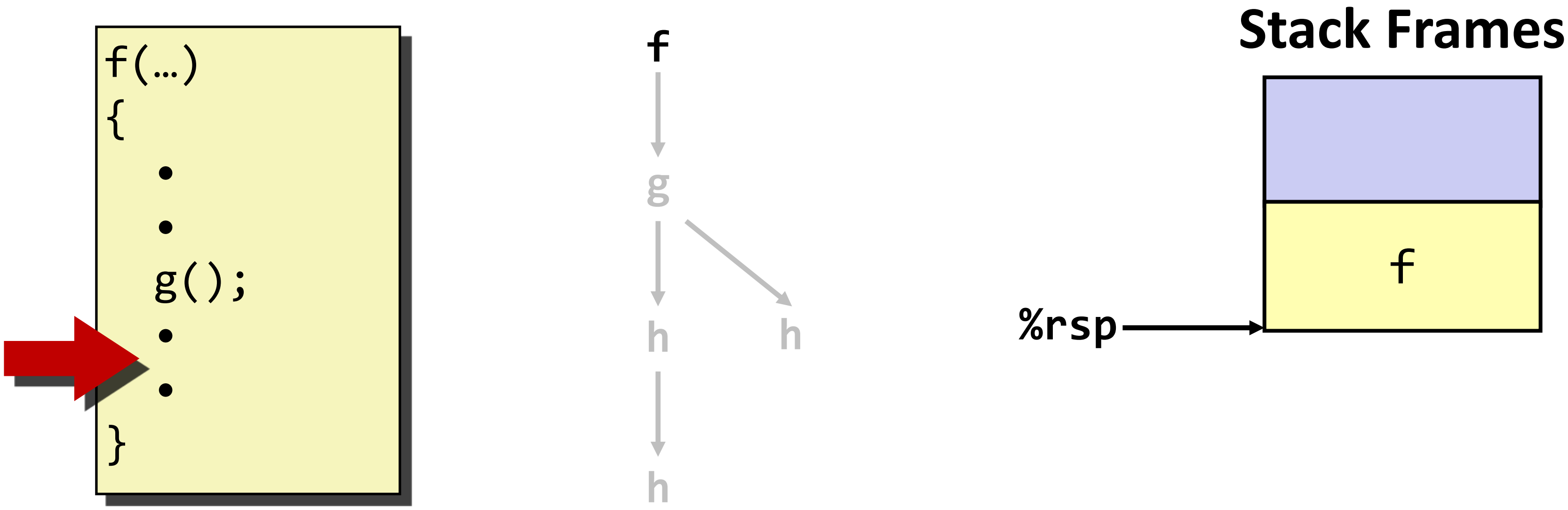
Call Chain and Stack Frame



Call Chain and Stack Frame



Call Chain and Stack Frame



Stack Frame Example: `incr`

```
void incr(long *p, long val)
{
    *p = *p + val;
}
```

```
incr:
  0x401106 <+0>:   add  %rsi,(%rdi)
  0x401109 <+3>:   ret
```

Register	Usage
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**

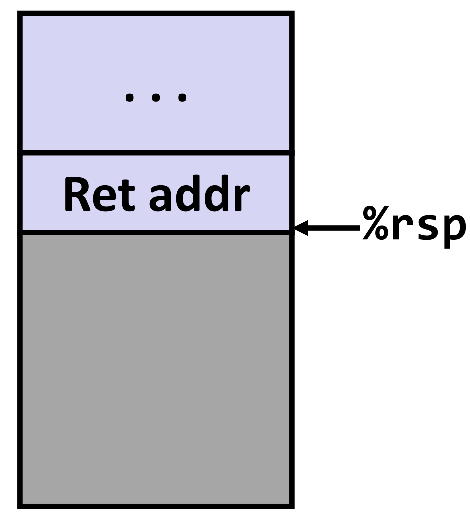
Stack Frame Example: `call_incr`

```
long call_incr() {
    long v1 = 15213;
    incr(&v1, 3000);
    return v1;
}
```

(Status at function entry)

```
call_incr:
 0x40110a <+0>:   sub    $0x10,%rsp
 0x40110e <+4>:   movq   $15213,0x8(%rsp)
 0x401117 <+13>:  mov    $3000,%esi
 0x40111c <+18>:  lea    0x8(%rsp),%rdi
 0x401121 <+23>:  call   0x401106 <incr>
 0x401126 <+28>:  mov    0x8(%rsp),%rax
 0x40112a <+32>:  add    $0x10,%rsp
 0x40112e <+36>:  ret
```

Stack

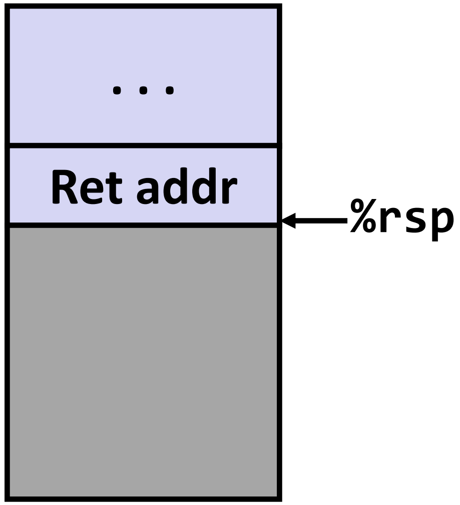


Stack Frame Example: `call_incr`

```c
long call_incr() {
    long v1 = 15213;
    incr(&v1, 3000);
    return v1;
}
```

```
call_incr:
 0x40110a <+0>:   sub    $0x10,%rsp
 0x40110e <+4>:   movq   $15213,0x8(%rsp)
 0x401117 <+13>:  mov    $3000,%esi
 0x40111c <+18>:  lea    0x8(%rsp),%rdi
 0x401121 <+23>:  call   0x401106 <incr>
 0x401126 <+28>:  mov    0x8(%rsp),%rax
 0x40112a <+32>:  add    $0x10,%rsp
 0x40112e <+36>:  ret
```

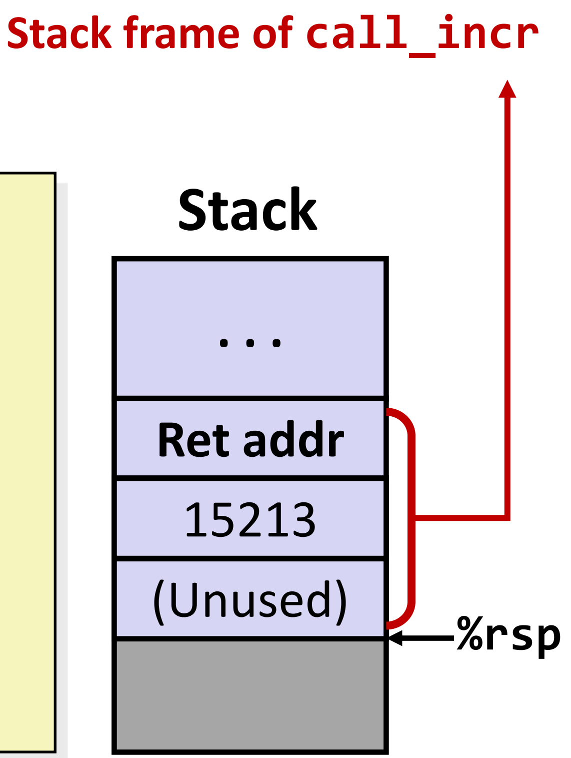
Stack



Stack Frame Example: `call_incr`

```
long call_incr() {
    long v1 = 15213;
    incr(&v1, 3000);
    return v1;
}
```

```
call_incr:
 0x40110a <+0>:   sub    $0x10,%rsp
 0x40110e <+4>:   movq   $15213,0x8(%rsp)
 0x401117 <+13>:  mov    $3000,%esi
 0x40111c <+18>:  lea    0x8(%rsp),%rdi
 0x401121 <+23>:  call   0x401106 <incr>
 0x401126 <+28>:  mov    0x8(%rsp),%rax
 0x40112a <+32>:  add    $0x10,%rsp
 0x40112e <+36>:  ret
```

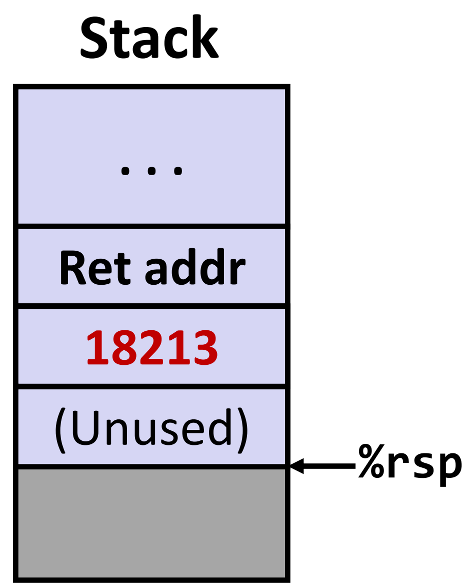


Stack Frame Example: `call_incr`

```
long call_incr() {
    long v1 = 15213;
    incr(&v1, 3000);
    return v1;
}
```

Register	Value at 0x401121
%rdi	&v1
%rsi	3000

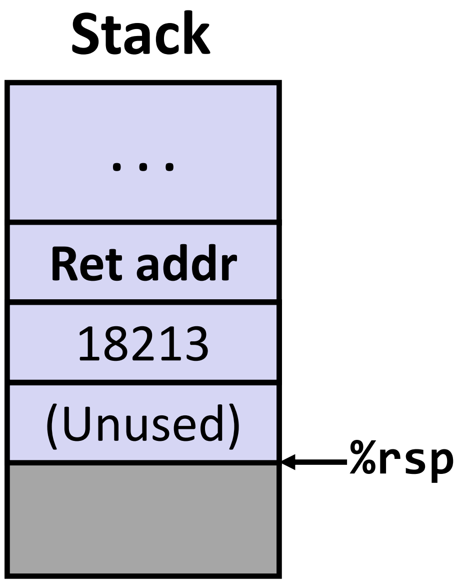
```
call_incr:
 0x40110a <+0>:   sub    $0x10,%rsp
 0x40110e <+4>:   movq   $15213,0x8(%rsp)
 0x401117 <+13>:  mov    $3000,%esi
 0x40111c <+18>:  lea    0x8(%rsp),%rdi
 0x401121 <+23>:  call   0x401106 <incr>
 0x401126 <+28>:  mov    0x8(%rsp),%rax
 0x40112a <+32>:  add    $0x10,%rsp
 0x40112e <+36>:  ret
```



Stack Frame Example: `call_incr`

```
long call_incr() {
    long v1 = 15213;
    incr(&v1, 3000);
    return v1;
}
```

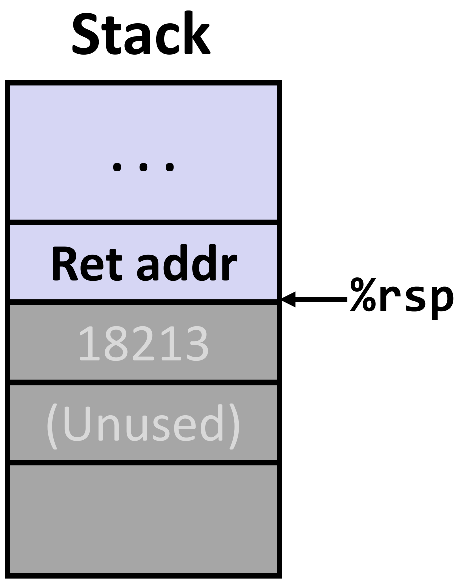
```
call_incr:
 0x40110a <+0>:   sub    $0x10,%rsp
 0x40110e <+4>:   movq   $15213,0x8(%rsp)
 0x401117 <+13>:  mov    $3000,%esi
 0x40111c <+18>:  lea    0x8(%rsp),%rdi
 0x401121 <+23>:  call   0x401106 <incr>
 0x401126 <+28>:  mov    0x8(%rsp),%rax
 0x40112a <+32>:  add    $0x10,%rsp
 0x40112e <+36>:  ret
```



Stack Frame Example: `call_incr`

```
long call_incr() {
    long v1 = 15213;
    incr(&v1, 3000);
    return v1;
}
```

```
call_incr:
 0x40110a <+0>:   sub    $0x10,%rsp
 0x40110e <+4>:   movq   $15213,0x8(%rsp)
 0x401117 <+13>:  mov    $3000,%esi
 0x40111c <+18>:  lea    0x8(%rsp),%rdi
 0x401121 <+23>:  call   0x401106 <incr>
 0x401126 <+28>:  mov    0x8(%rsp),%rax
 0x40112a <+32>:  add    $0x10,%rsp
 0x40112e <+36>:  ret
```

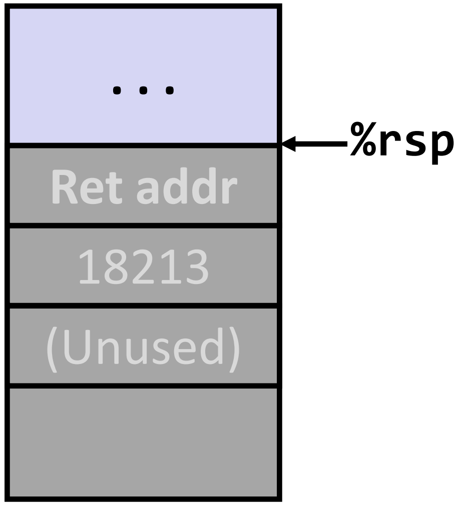


Stack Frame Example: `call_incr`

```
long call_incr() {
    long v1 = 15213;
    incr(&v1, 3000);
    return v1;
}
```

```
call_incr:
 0x40110a <+0>:   sub    $0x10,%rsp
 0x40110e <+4>:   movq   $15213,0x8(%rsp)
 0x401117 <+13>:  mov    $3000,%esi
 0x40111c <+18>:  lea    0x8(%rsp),%rdi
 0x401121 <+23>:  call   0x401106 <incr>
 0x401126 <+28>:  mov    0x8(%rsp),%rax
 0x40112a <+32>:  add    $0x10,%rsp
 0x40112e <+36>:  ret
```

Stack



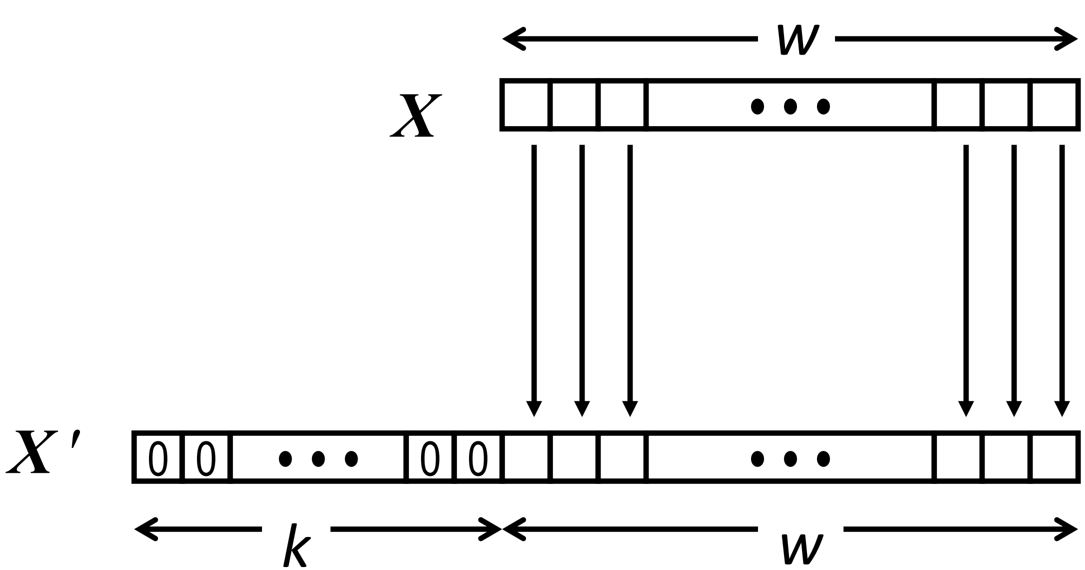
Summary

- **Brief introduction of assembly and Intel x86**
- **Data representation in CPU and memory**
- **Basic instructions of x86-64 assembly**
- **Control instructions of x86-64 assembly**
- **Function call in x86-64 assembly**
  - Stack memory region and `push/pop` instruction
  - Function call and return (`call/ret` instruction)
  - Calling convention
    - Passing arguments and return value
    - Caller saved register vs. callee saved register
  - Stack frame management

Appendix

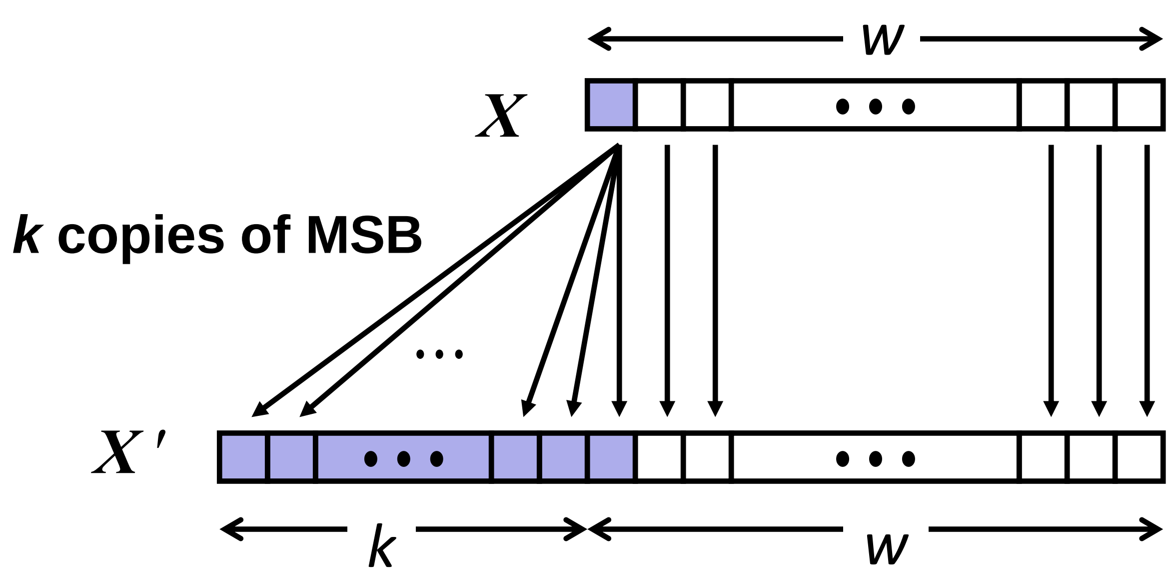
Zero Extension

- **Task**
  - Given *w*-bit ***unsigned integer*** *x*
  - Convert it to (*w*+*k*)-bit integer with same value
- **Rule**
  - Fill the upper *k* bits with 0



Sign Extension

- **Task**
  - Given *w*-bit ***signed integer*** *x*
  - Convert it to (*w*+*k*)-bit integer with same value
- **Rule**
  - Make *k* copies of MSB (most significant bit, or sign bit)



Sign Extension Example

Positive number

	-16	8	4	2	1
10 =	0	1	0	1	0

	-32	16	8	4	2	1
10 =	0	0	1	0	1	0

Negative number

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-32	16	8	4	2	1
-10 =	1	1	0	1	1	0

Shifting

- **Left shift: `x << y`**
  - Shift ***x*** to left direction by ***y*** bits
  - Fill with 0's on right
  - Throw away extra bits on left
  - Equal to $x * 2^y$
- **Right shift: `x >> y`**
  - Shift ***x*** to right direction by ***y*** bits
  - Two kinds of right shifting
    - ***Logical:*** Fill with 0's on left
    - ***Arithmetic:*** Copy MSB on left
  - Throw away extra bits on right
  - Arithmetic right shift is equal to floor($x/2^y$)

Argument x	01100010
<< 3	00010000
Logical >> 2	00011000
Arithmetic >> 2	00011000

Argument x	10100010
<< 3	00010000
Logical >> 2	00101000
Arithmetic >> 2	11101000