Chapter 3.
Assembly Language (x86-64)

Prof. Jaeseung Choi

Dept. of Computer Science and Engineering

Sogang University

Before We Start

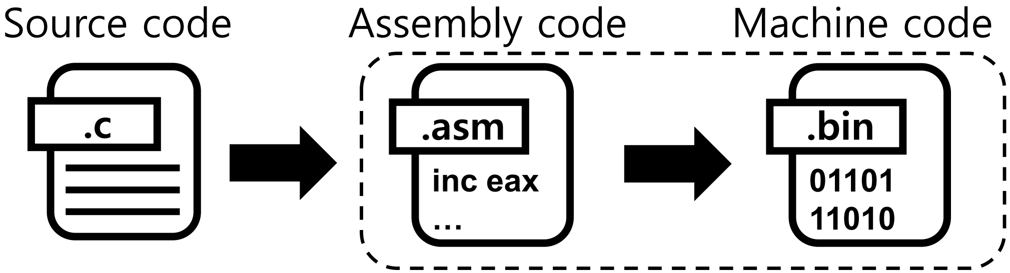
- **This is a summarized version of lecture notes from CSE3030 *(Introduction to Computer Systems)***
- **Full version of slide is also uploaded in *Cyber Campus***
  - Check the full version for more details
- **You don't have to be an expert in assembly, but certain amount of knowledge is required 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

Topics

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

Why Learn Assembly?

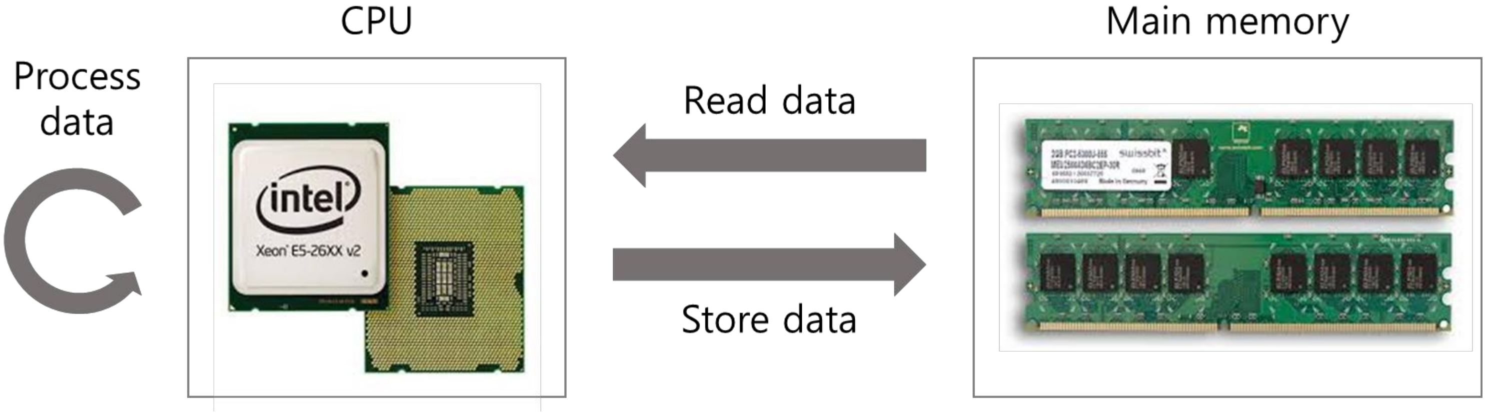
- When you write and compile a C program, it is translated into assembly code (machine code, to be precise)
- This is the form of code that a computer can understand
- Therefore, learning assembly is **learning how a computer internally operates**



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**



CPU

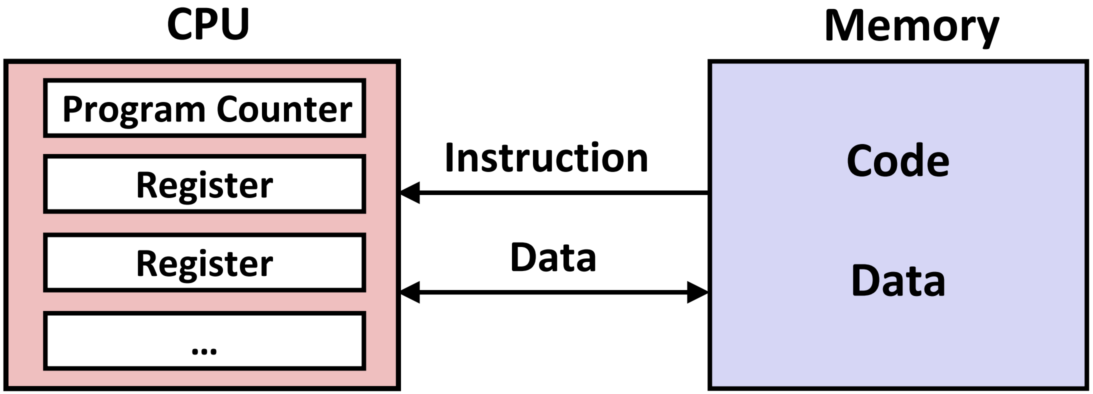
Main memory

Process data

Read data

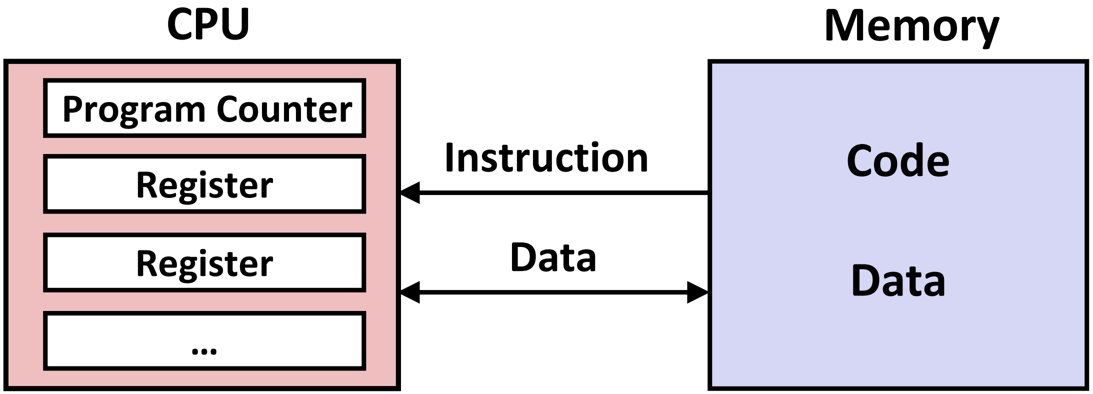
Store data

How does CPU work?



- **CPU fetches a machine instruction from memory**
  - Instruction is just a sequence of bits with promised meanings
  - Fetch from where? **Program Counter (PC)** tells you the address
- **The instruction tells CPU what to do**
  - Ex) Move data from register to memory, add two registers, ...
  - When the task is done, PC changes to point at next instruction

Machine/Assembly Code



CPU

Program Counter

Register

Register

…

Instruction

Data

Memory

Code

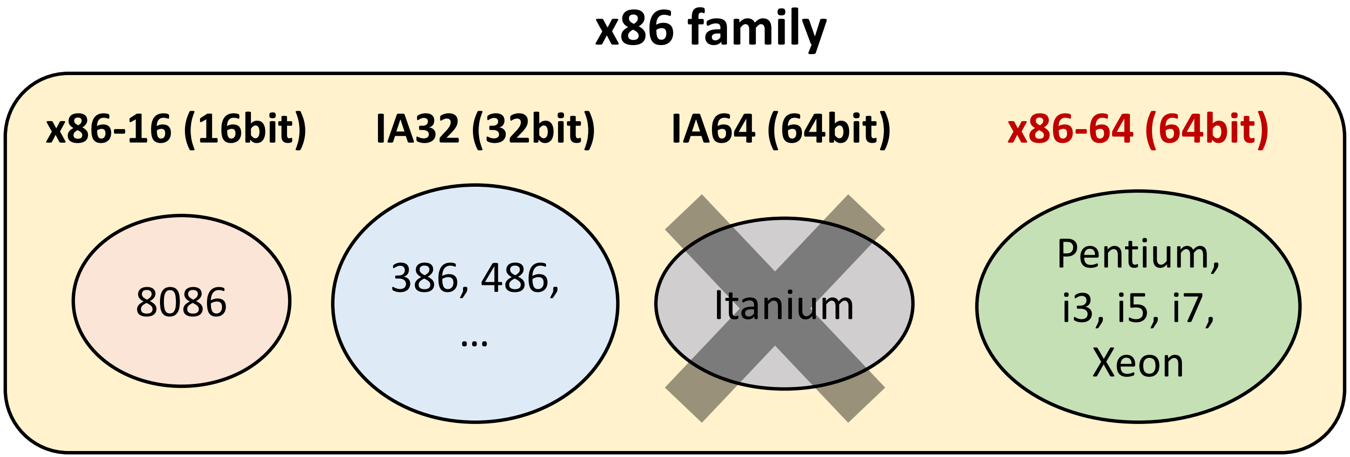
Data

- **Machine instruction has 1-to-1 mapping with assembly instruction: easily translatable to each other**
- **Assembly code (= instructions) is just human-friendly representation of machine code**

Machine instruction		Assembly instruction
0x48 0x01 0xd8	⟷	add %rax, %rbx

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

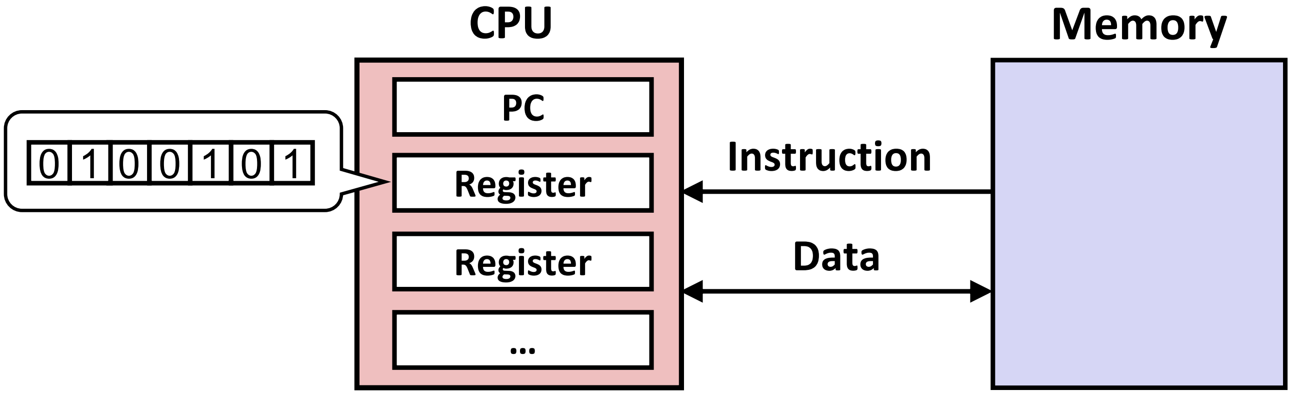


Topics

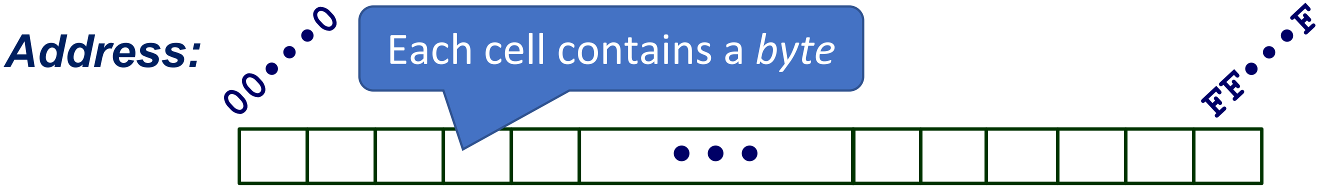
- Brief introduction of assembly and Intel x86
- **Data representation in CPU and memory**
- Basic instructions of x86 assembly
- Control instructions of x86 assembly
- Function call in x86 assembly

Data Representation

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



Basic Structure of Memory



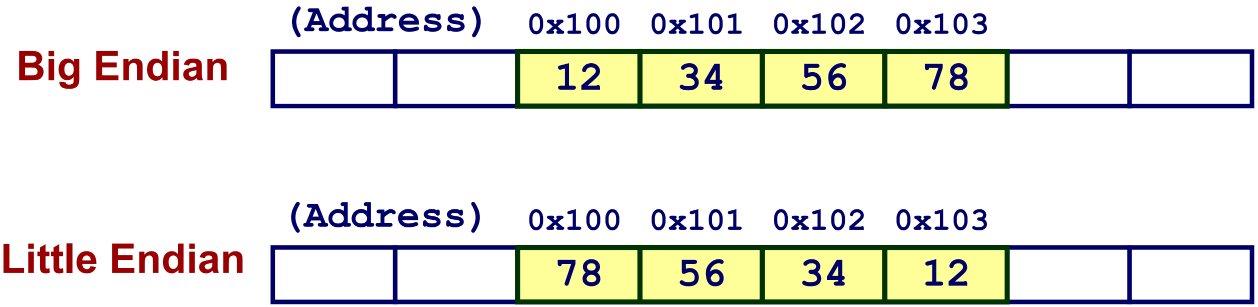
- **Conceptually, memory is a large array of bytes**
  - Each byte space is associated with an address
- **Program accesses memory by using address**
  - Just like using index for an array
  - Program can access multiple bytes at once
    - Ex) Load 4-bytes starting from address 0x200
  - Not all addresses 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 ranges from 0 to 2^{64}
  - But we typically use only memory space address from 0 to 2^{48}

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: C code "`int x = 0x12345678;`"**
  - Note that 0x12 is the most significant byte here
  - 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 assembly**
- Control instructions of x86 assembly
- Function call in x86 assembly

Registers in x86-64

- %rsp : stack pointer

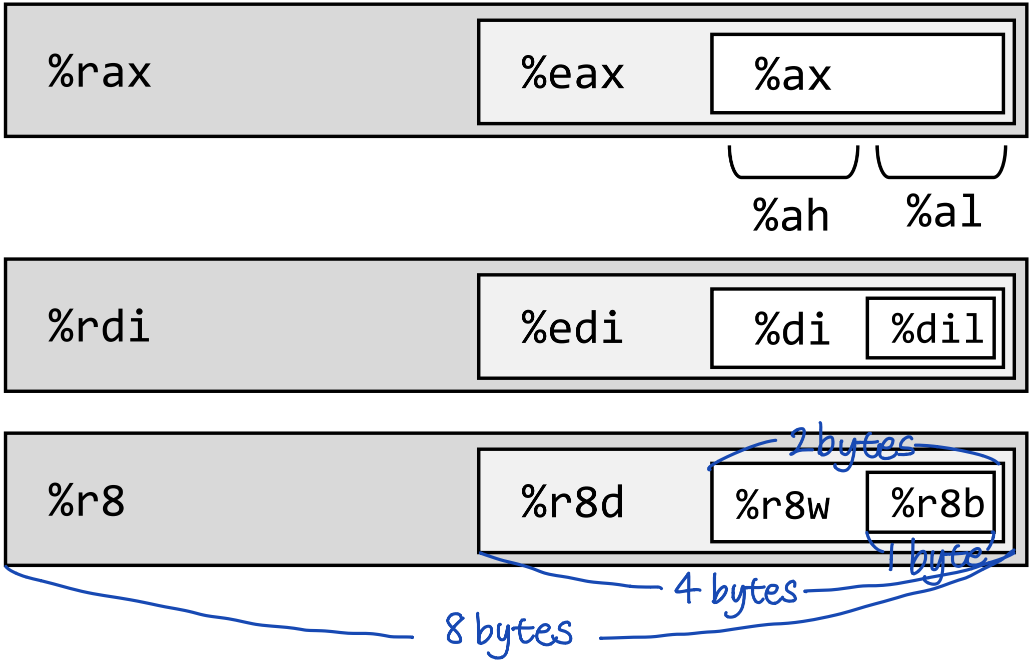
intel architecture 에서
PC를 지칭하는 레지스터 이름

- %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	

Partial Access on Register

- **Each register (e.g., %rax) is 8 byte, but we can also access its lower 4 bytes (%eax), 2 bytes (%ax), or 1 byte (%ah, %al)**
  - You don’t have to memorize these names



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

x86-64 Assembly Code

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

Data Move Instruction: `mov`

- **Instruction: `mov` *`Source, Destination`***
  - Sometimes we put 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**
  - ***Immediate:*** Constant integer value
    - Example: **`$0x400`**, **`$-533`** (prefixed with **`'$'`**)
  - ***Register:*** One of the registers previously discussed
    - Example: **`%rax`**, **`%r13`**
  - ***Memory:*** Consecutive bytes in memory at the specified address
    - Example: **`(%rax)`**, **`0x1000`**
    - Note: existence of **`$`** decides immediate vs. memory access

Operand Combinations for `mov`

	Source	Dest	Example	C Analog
`mov`	*Imm*	*Reg*	`mov $0x4,%rax`	`a = 0x4;`
		Mem	`mov $-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);`
		Mem X		

Cannot do memory-memory 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 : 0x11223344556677FF
mov $1,  %ax     # %rax : 0x1122334455660001
mov $-1, %eax    # %rax : 0x00000000FFFFFFFF
mov $1,  %rax    # %rax : 0x0000000000000001
```

← 2's complement

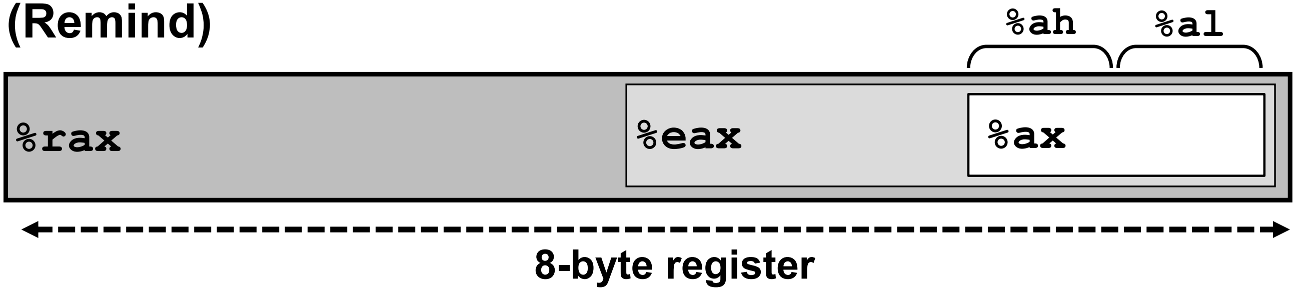
Caution: Sets higher bytes to zero

아래쪽 4bytes 만 update 해도
위쪽 4bytes 를 0으로 채워줌

  - Works similarly for other operand combinations

    Ex) mov %ebx, %eax   # Set lower 4 bytes and clear higher 4 bytes

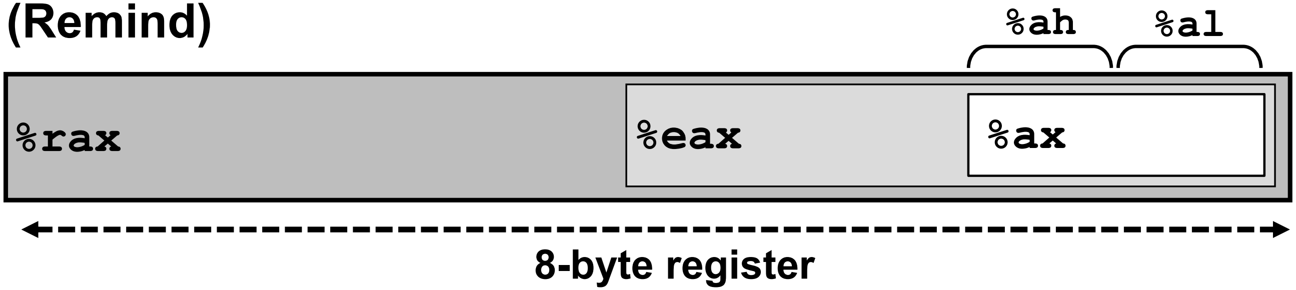
(Remind)



Byte Extension with `movz/movs`

- **Move with zero extension: `movz` *`Reg`*`,` *`Reg`***
  - **We can also have suffixes (`b/w/d/q`) here**
  - Ex) `movzbw %bl, %ax` # Zero-extend 1 byte into 2 bytes
- **Move with sign extension: `movs*` *`Reg`*`,` *`Reg`***
  - Ex) `movslq %ebx, %rax` # Sign-extend 4 bytes into 8 bytes
- **Don’t remember zero extension vs. sign extension?**
  - Check *"Chapter 2. Data Representation"* from the full slides

(Remind)



Complex Memory Access

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

Scale factor

- **Other variants (special case of the above form)**
  - Ex) `mov 0x10(%rbx,%rcx), %rax` **# Scale factor 1 is omitted**
  - Ex) `mov (%rbx,%rcx,4), %rax` **# Offset 0 is omitted**
  - Ex) `mov (%rbx,%rcx), %rax` **# Both are omitted**
  - Ex) `mov 0x1000(%rbx), %rax` **# Only one register used**
  - Ex) `mov 0x1000(,%rcx,4), %rax` **# Similar here**

Arithmetic Instructions

- **Instructions with two operands:**

Instruction		*Computation*	
`add`	*Src, Dest*	Dest = Dest + Src	
`sub`	*Src, Dest*	Dest = Dest – Src	
`imul`	*Src, Dest*	Dest = Dest * Src	
`shr`	*Src, Dest*	Dest = Dest >> Src	# Logical right shift
`sar`	*Src, Dest*	Dest = Dest >> Src	# Arithmetical 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 %rax, %rbx  / sub $0x10, %rcx`

Arithmetic 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	# Arithmetical right shift by one
`shl`	Dest	Dest = Dest << 1	# Left shift by one

- **Don’t remember logical shift vs. arithmetical shift?**
  - Check *"Chapter 2. Data Representation"* from the full slides

Pointer Computation: lea

- **Review: `mov 0x20(%rbx,%rcx,4), %rax` loads memory bytes from address `0x20` + `%rbx` + `%rcx` * 4**
- **This is equal to the following instruction sequence**

```
lea 0x20(%rbx,%rcx,4), %rax   ← 메모리 주소 계산 (pointer computation only)
mov (%rax), %rax              ← 실제 메모리에 접근 (loads memory)
```

  - The first instruction computes `0x20` + `%rbx` + `%rcx` * 4 and store the result into `%rax` **(pointer computation only)**
  - The second instruction loads from the computed address
- **In other words, `lea` is used for pointer computation**

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

Another use of `lea` Instruction

- **`lea` is originally intended for pointer computation, but ...**
- **Compilers often abuse it for arithmetic operation**
  - Can perform complex operations effectively
  - In the example below, using `lea` is more effective than using add and `mul`

C code	Assembly
`long a = b + 2 * c`	`lea (%rbx,%rcx,2), %rax`

Example

- **Let’s review the example code from previous page**
- **(Remind) Convention of using registers**
  - 1st, 2nd, 3rd … arguments passed through `%rdi`, `%rsi`, `%rdx` …
  - Return value passed through `%rax` register

C Code

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

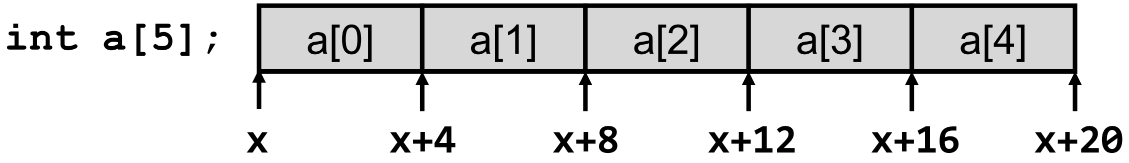
x86-64 Assembly Code

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

하위 4bytes는 $0x1로,
상위 4bytes는 자동 clear됨
(모두 0으로 세팅됨)

Another Example

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



C Code

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

x86-64 Assembly Code

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

Topics

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

Control Flow in Assembly Code

- **Everything is done with *goto-style* instructions**
  - We will talk about function call later

```
void f(int x) {
    if (x == 0) {
        op1();
    } else {
        op2();
    }
}
```

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

Jump to 0x100 if %edi == %edi

세트로 보기!!

true이면 jump

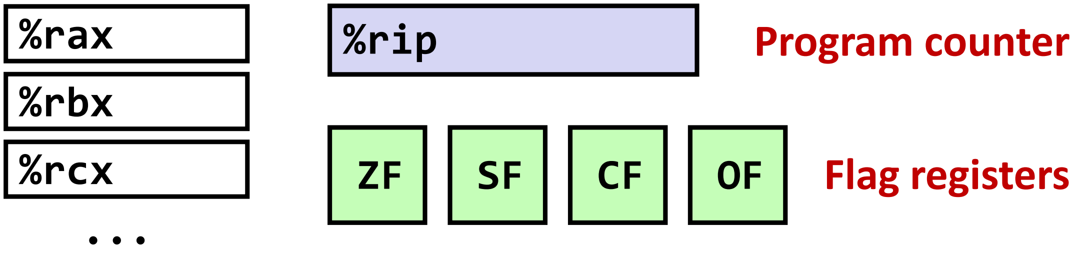
false이면 실행되지 않고 바로 다음 line으로

Instruction이 끝나면 무조건 0x105로 jump

More on x86-64 CPU Registers

- **General-purpose registers:** `%rax, %rbx, ...`
  - Program data (e.g., variable)
- **Program counter *(instruction pointer)*:** `%rip`
  - Address of the instruction to execute
  - Affected by control instructions (like `je` in the previous example)
- **Flag registers:** `%ZF, %SF, %CF, %OF`
  - Store the result of condition checks (like `test` in the example)

Registers



Conditional Jump

- **We will not cover the details of register flags**
  - How each flag register is updated by various instructions
- **Just get familiar to common patterns of control transfer**

(1) First, check for certain condition (`cmp`, `test`, `sub`, `and`, …)

equal greater

(2) Then, run conditional jump instruction (`je`, `jne`, `jg`, `jl`, …)

not equal less

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

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

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

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

[test, cmp : 단순비교
[and, sub : update dest register

Caution on direction

Jump to 0x100 if `%rax == 0`

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

Difference? `test`/`cmp` do not update destination, while `and`/`sub` do

Reference for `jx` Instructions

Instruction	Description
`jmp`	Always jump
`je`	Equality check (jump if zero/equal)
`jne`	Equality check (jump if NOT zero/equal)
`js`	Sign check (jump if negative)
`jns`	Sign check (jump if positive)
`jg`	Signed comparison (jump if greater)
`jge`	Signed comparison (jump if greater or equal)
`jl`	Signed comparison (jump if less)
`jle`	Signed comparison (jump if less or equal)
`ja`	Unsigned comparison (jump if above)
`jae`	Unsigned comparison (jump if above or equal)
`jb`	Unsigned comparison (jump if below)
`jbe`	Unsigned comparison (jump if below or equal)

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

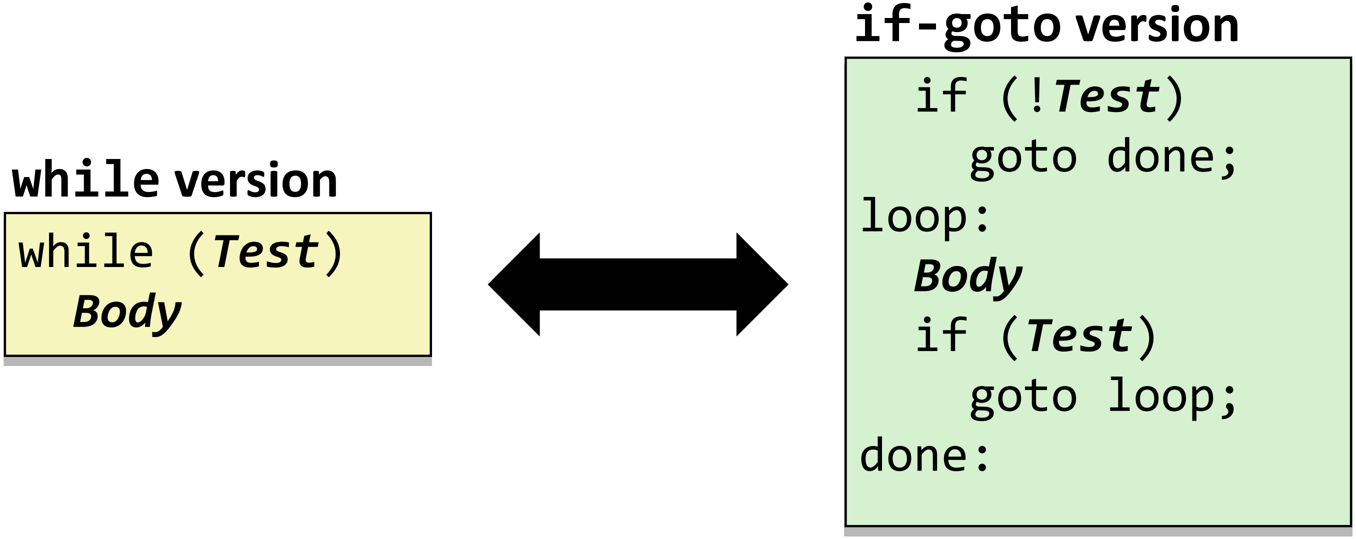
Instruction	Description
sete	Equality check (set if zero/equal)
. . .	...

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

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

Loop in Assembly Code

- **We can also implement loop (`while`, `for`) in assembly by using conditional jump instructions**
  - Various patterns exist (we will not cover the details)



while version

```
while (Test)
  Body
```

if-goto version

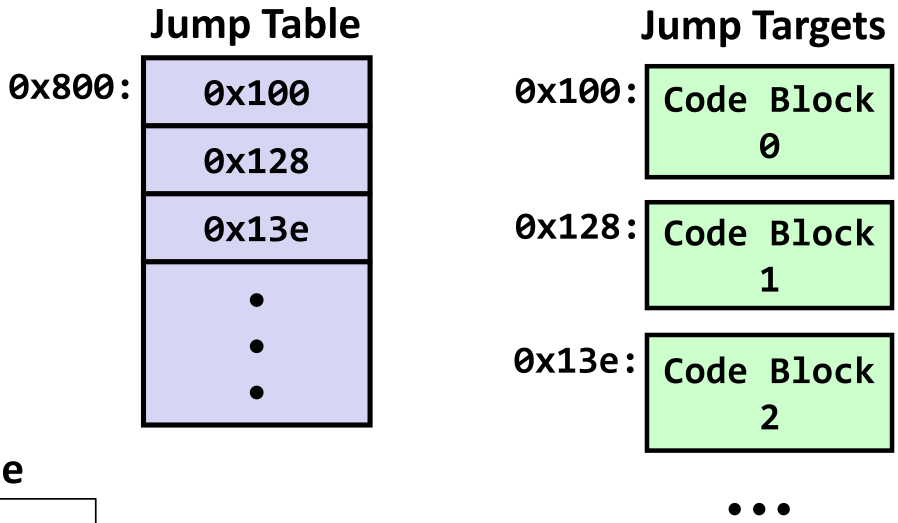
```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

Switch in Assembly Code

- **Compiler generates jump table to implement switch**
  - But not always (sometimes it just uses conditional jumps)

Switch statement

```
switch(x) {
  case val_0:
    Body 0
  case val_1:
    Body 1
    • • •
}
```



Jump Table

0x800:

0x100
0x128
0x13e
• • •

Jump Targets

0x100: Code Block 0

0x128: Code Block 1

0x13e: Code Block 2

• • •

Translated assembly code

```
jmp *0x800(,x,8);
```

Each jump table entry is 8-byte

Using memory content as jump destination

Topics

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

Especially important for learning *buffer overflow*

Mechanisms in Function Call

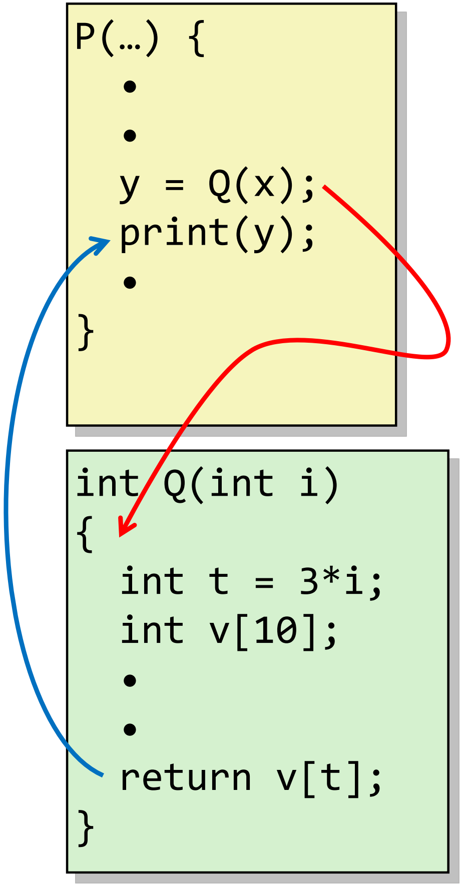
- **Passing control**
  - To the entry of a function
  - Back to return point
- **Passing data**
  - Function arguments
  - Return value
- **Memory management**
  - Allocate in function entry
  - Deallocate upon return

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

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

Mechanisms in Function Call

- **Passing control**
  - **To the entry of a function**
  - **Back to return point**
- **Passing data**
  - Function arguments
  - Return value
- **Memory management**
  - Allocate in function entry
  - Deallocate upon return

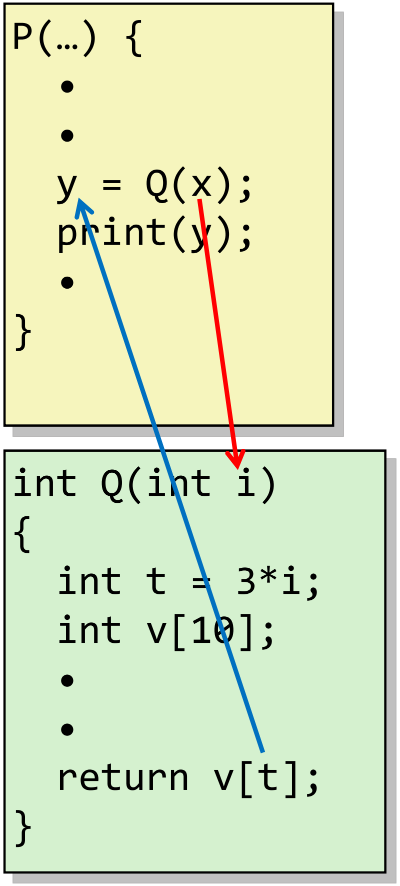


```
P(…) {
  •
  •
  y = Q(x);
  print(y);
  •
}
```

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

Mechanisms in Function Call

- **Passing control**
  - To the entry of a function
  - Back to return point
- **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

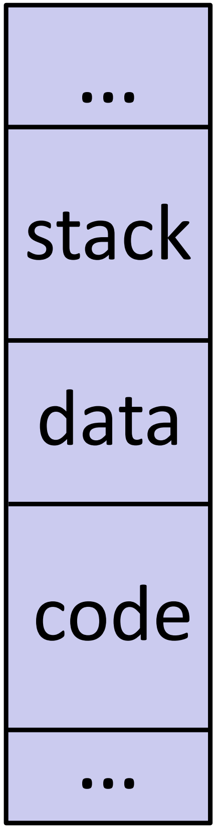
- **Passing control**
  - To the entry of a function
  - Back to return point
- **Passing data**
  - Function arguments
  - Return value
- **Memory management**
  - **Allocate in function entry**
  - **Deallocate upon return**

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

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

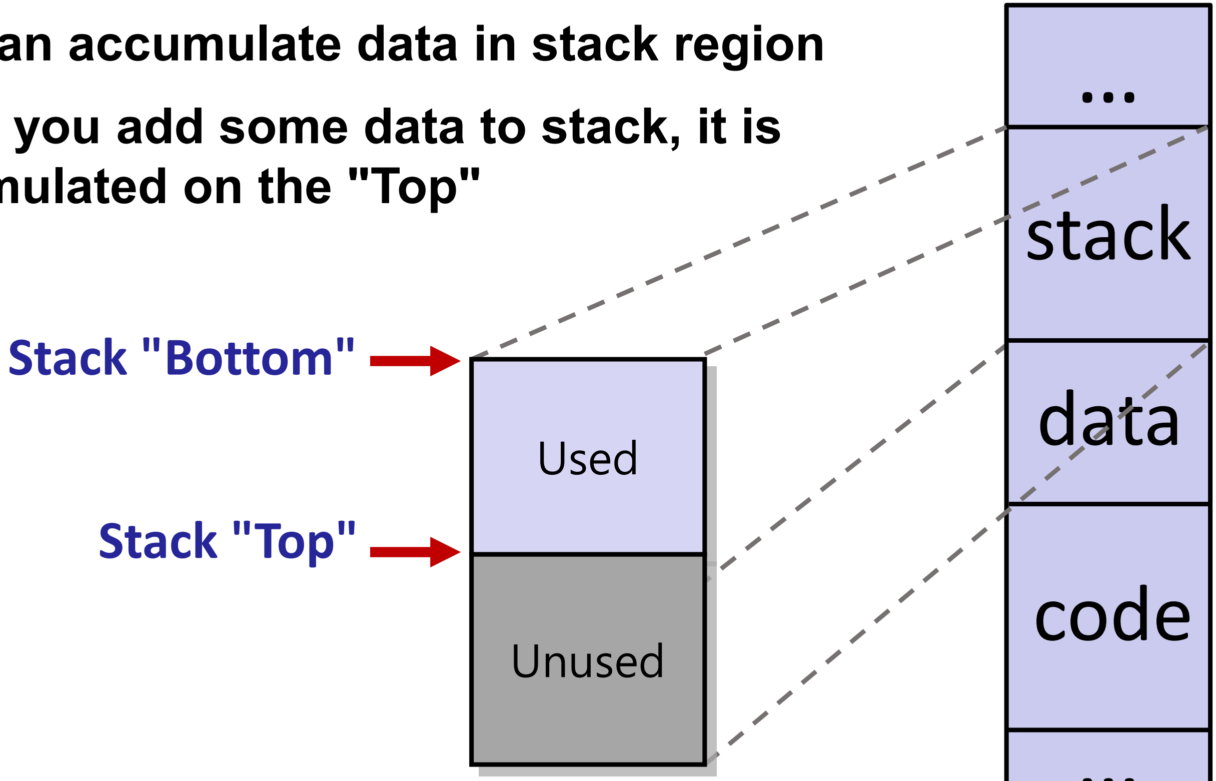
Memory Structure Revisited

- **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



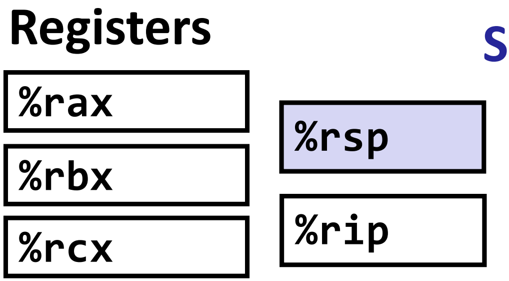
Stack

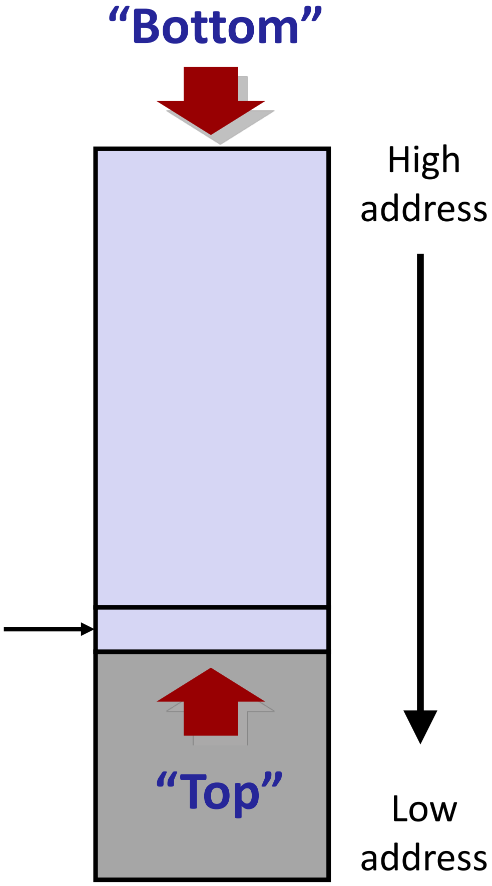
- You can accumulate data in stack region
- When you add some data to stack, it is accumulated on the "Top"



Stack Principles

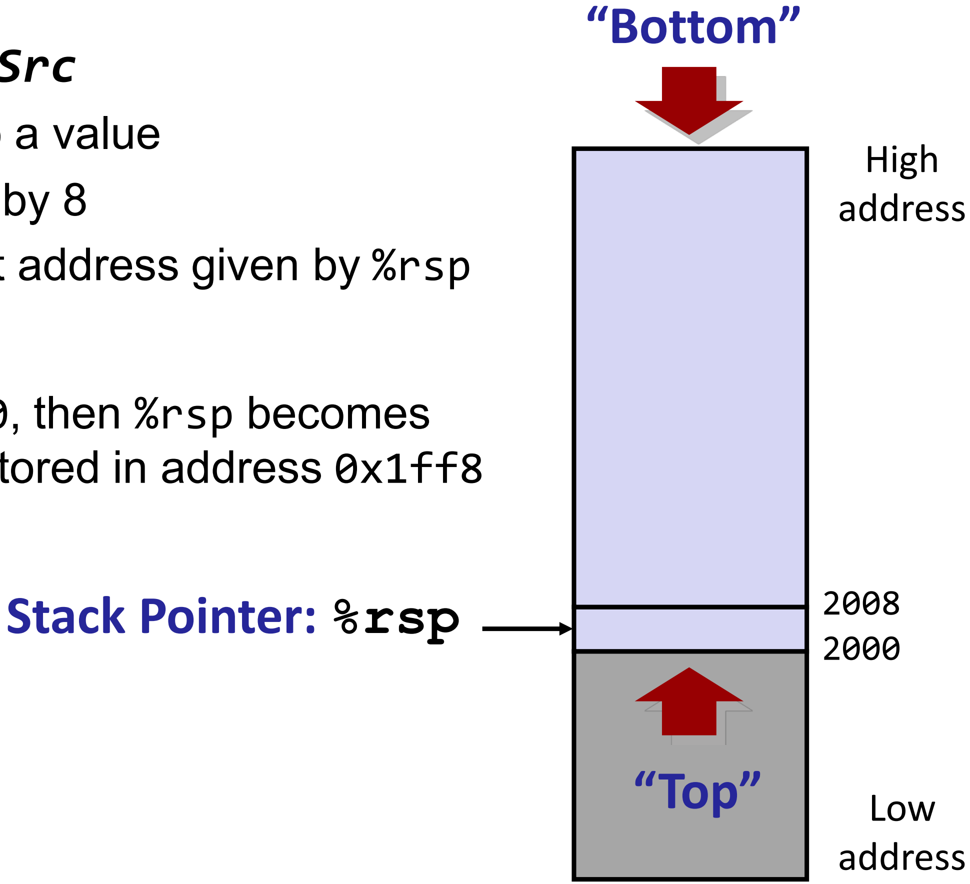
- **Many textbooks draw the memory upside down (just a convention)**
  - "Bottom" is assigned the highest address
- **When we add an element, stack grows toward lower addresses**
- **Stack pointer register `%rsp` points to the element at the "Top"**





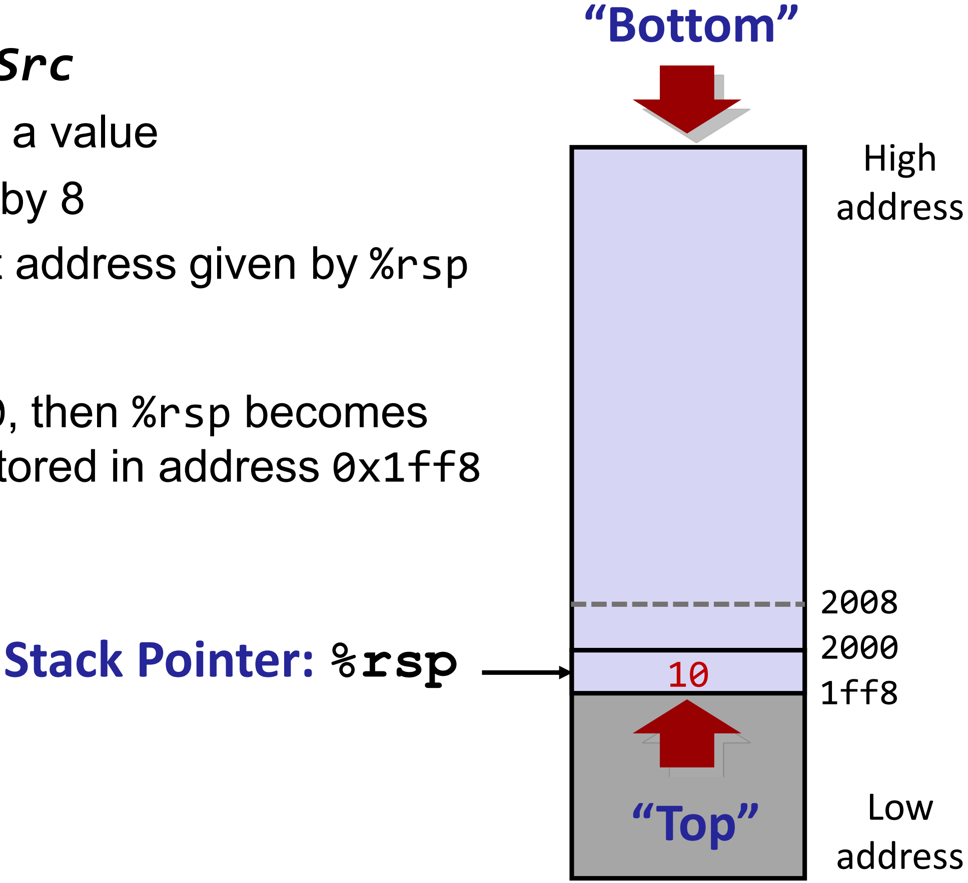
Stack Operation: Push

- **Instruction: `push` *`Src`***
  - (1) Evaluate *`Src`* into a value
  - (2) Decrement %rsp by 8
  - (3) Write the value at address given by `%rsp`
- **Ex) `push $10`**
  - If `%rsp` was `0x2000`, then `%rsp` becomes `0x1ff8` and `10` is stored in address `0x1ff8`



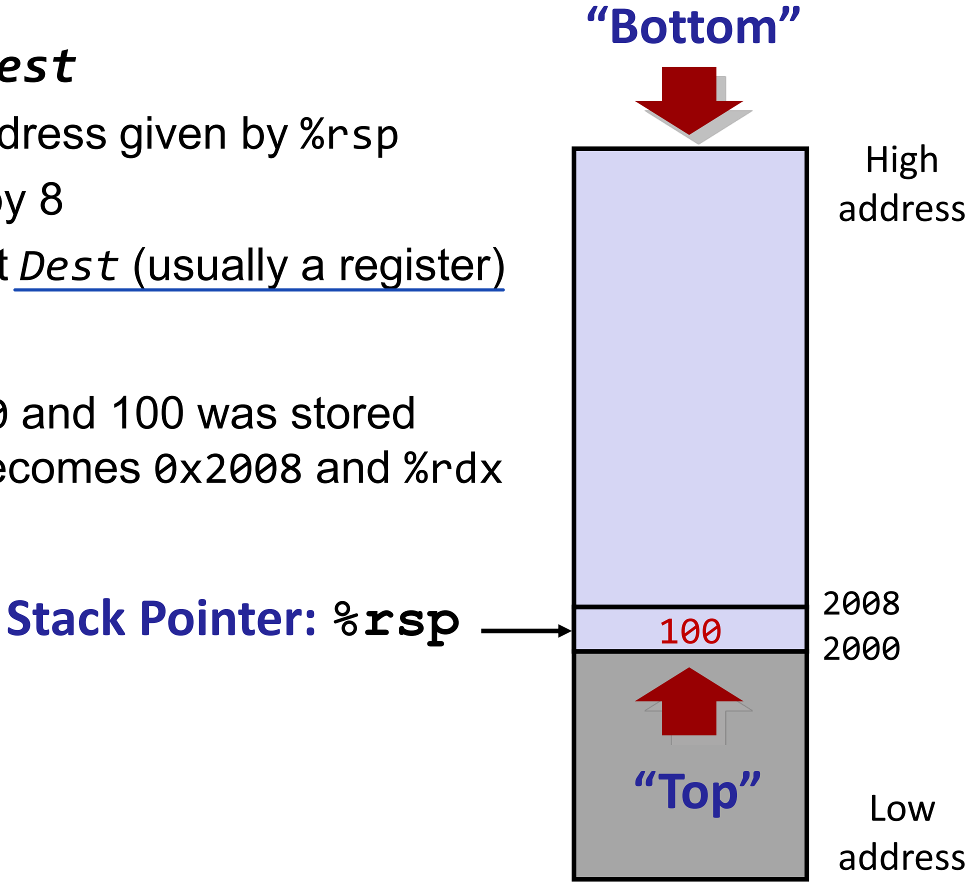
Stack Operation: Push

- **Instruction: `push` *`Src`***
  - (1) Evaluate *`Src`* into a value
  - (2) Decrement %rsp by 8
  - (3) Write the value at address given by `%rsp`
- **Ex) `push $10`**
  - If `%rsp` was `0x2000`, then `%rsp` becomes `0x1ff8` and `10` is stored in address `0x1ff8`



Stack Operation: Pop

- **Instruction: `pop` *`Dest`***
  - (1) Read value at address given by `%rsp`
  - (2) Increment `%rsp` by 8
  - (3) Store the value at *`Dest`* (usually a register)
- **Ex) `pop %rbx`**
  - If `%rsp` was `0x2000` and 100 was stored there, then `%rsp` becomes `0x2008` and `%rdx` becomes `100`



Stack Operation: Pop

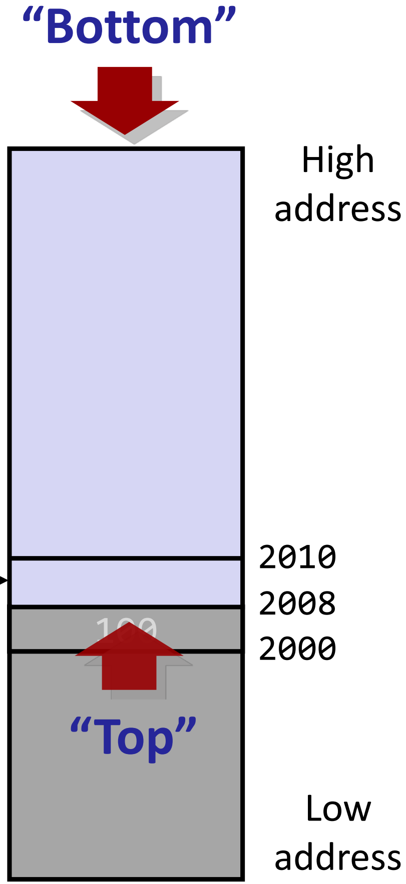
- **Instruction: `pop` *`Dest`***
  - (1) Read value at address given by `%rsp`
  - (2) Increment `%rsp` by 8
  - (3) Store the value at *`Dest`* (usually a register)
- **Ex) `pop %rbx`**
  - If `%rsp` was `0x2000` and 100 was stored there, then `%rsp` becomes `0x2008` and `%rdx` becomes `100`

Stack Pointer: `%rsp`

`%rbx`

100

The value is still there
(we're just not using it
anymore)



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

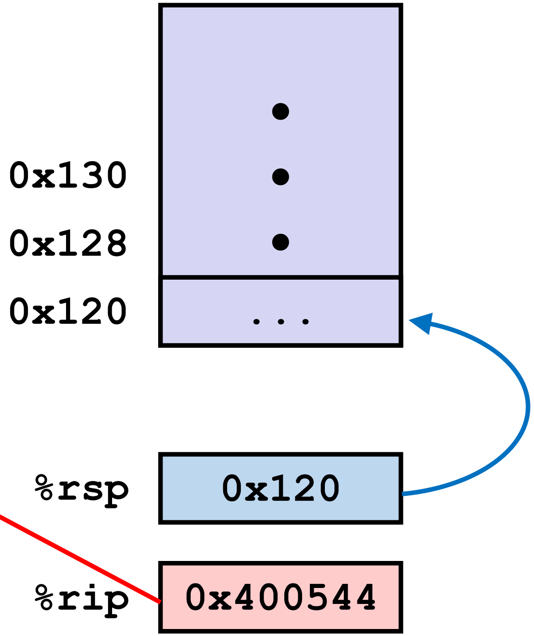
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax      # %rax := a
  400553:  imul   %rsi,%rax      # %rax := a * b
  400557:  ret                   # Return
```

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
```

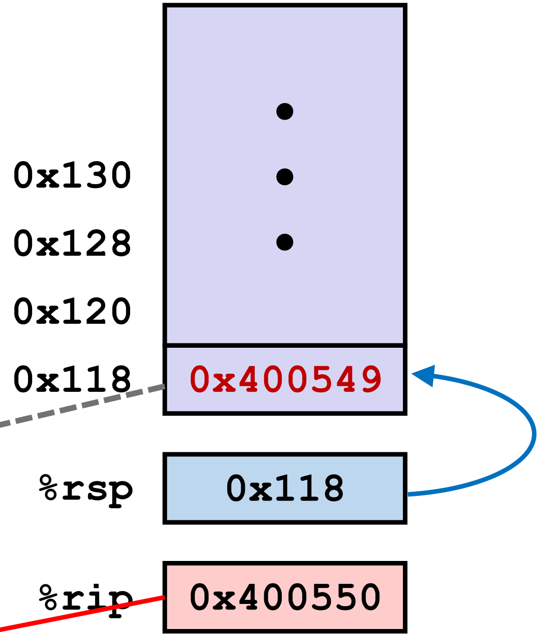


Function Call

- **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
```



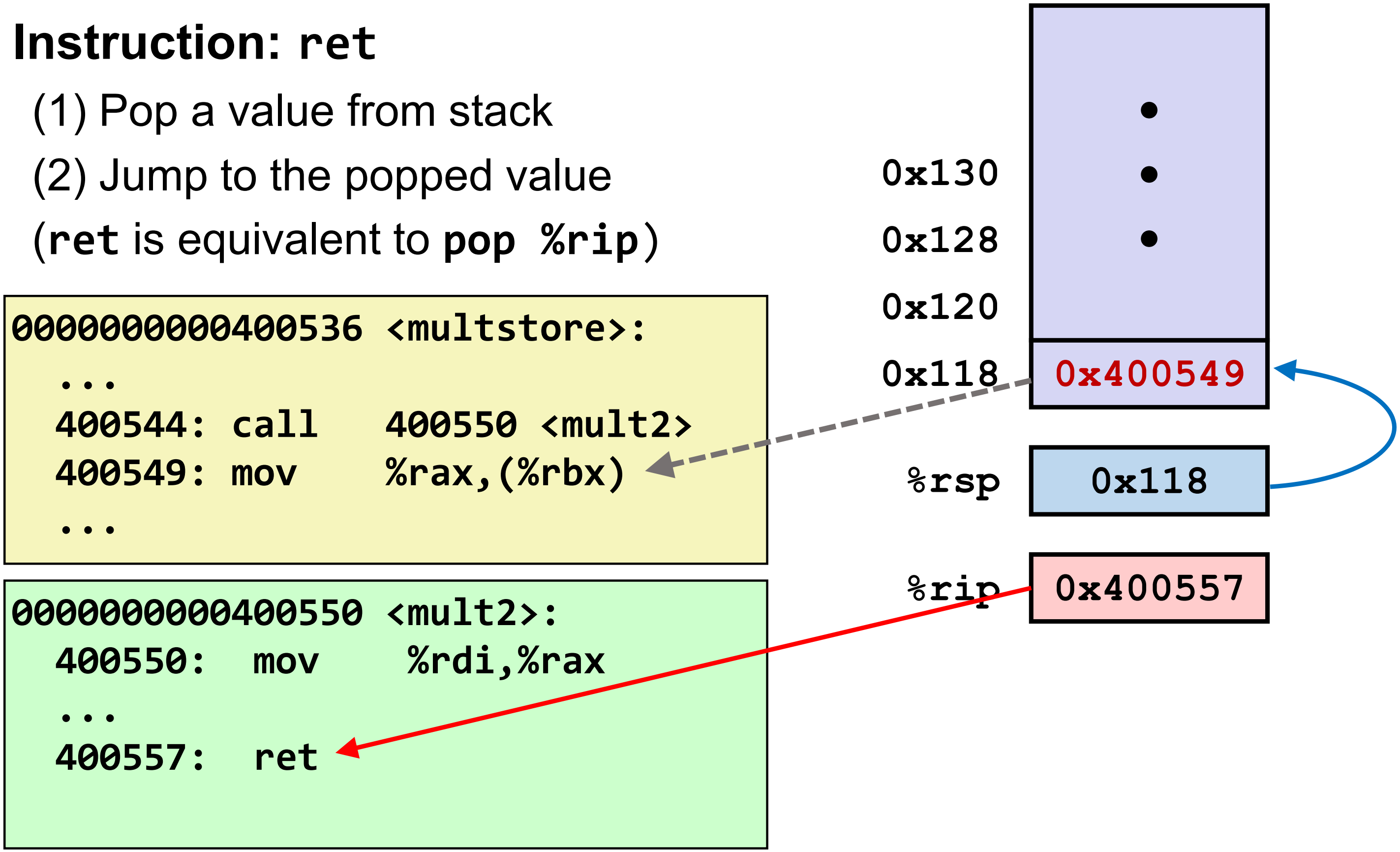
Function Return

- **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)
  ...
```

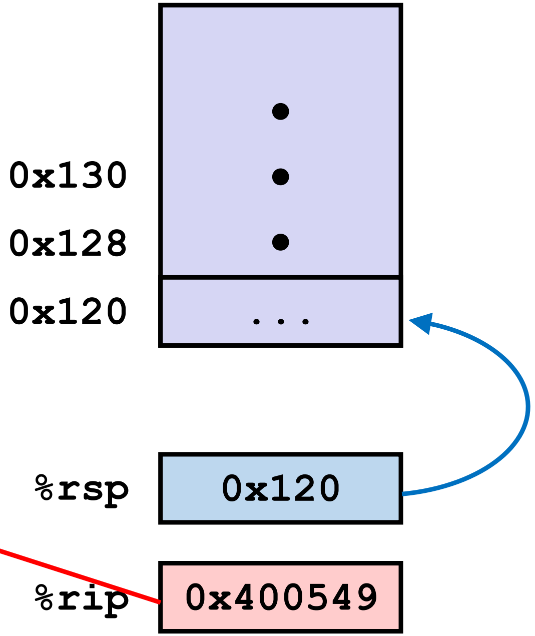
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  ...
  400557:  ret
```

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

    `%rdi`

    `%rsi`

    `%rdx`

    `%rcx`

    `%r8`

    `%r9`

  - If more arguments are needed, pass them through stack
  - Return value in `%rax` register

    `%rax`

Data Passing in multstore

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

```
0000000000400536 <multstore>:
  # At entry, dest is passed through %rdi
  ...
```

%rdi, %rsi의 하위 4bytes를 update (상위 4bytes는 0으로 채워짐)

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

Calling Convention: Saving Regs

- **(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, ...`

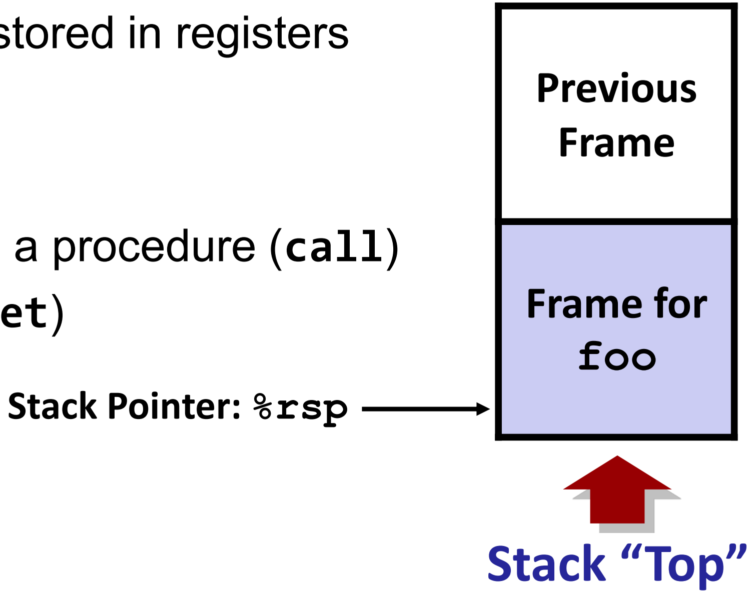
Register Save in multstore

- %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     $0x5,%edi          # Setup 1st arg
  40053f:  mov     $0x3,%esi          # Setup 2nd 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
  - Saved registers (if exists)
- **Management**
  - Allocated right after entering a procedure (**`call`**)
  - Deallocated before return (**`ret`**)



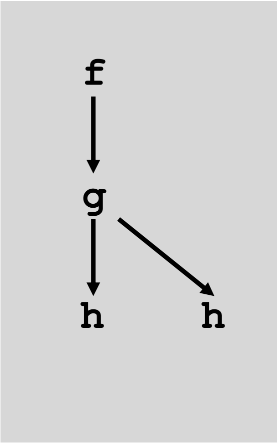
Call Chain and Stack Frame

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

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

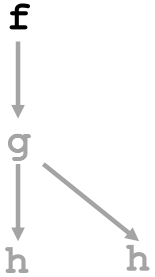
```
h(…)
{
   •
   •
   •
}
```

Call Graph

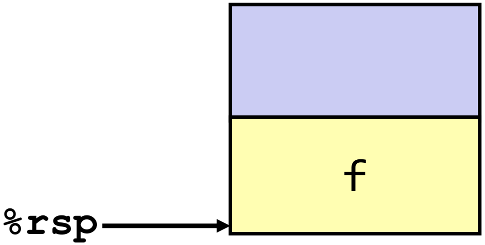


Call Chain and Stack Frame

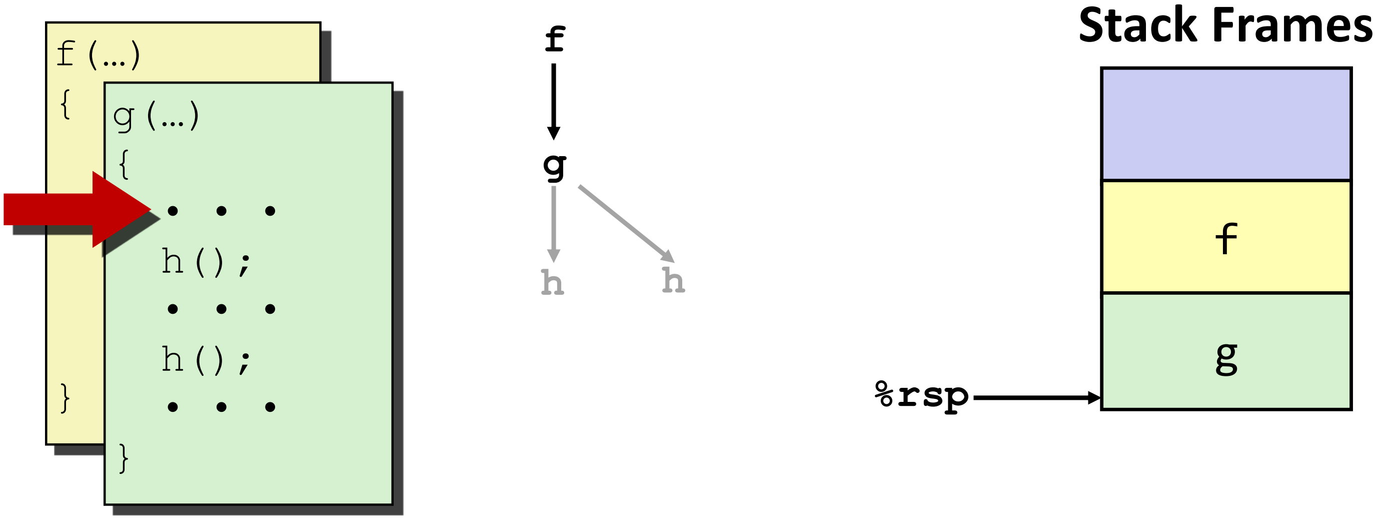
```
f(…)
{
   •
   •
   g();
   •
   •
}
```



Stack Frames



Call Chain and Stack Frame

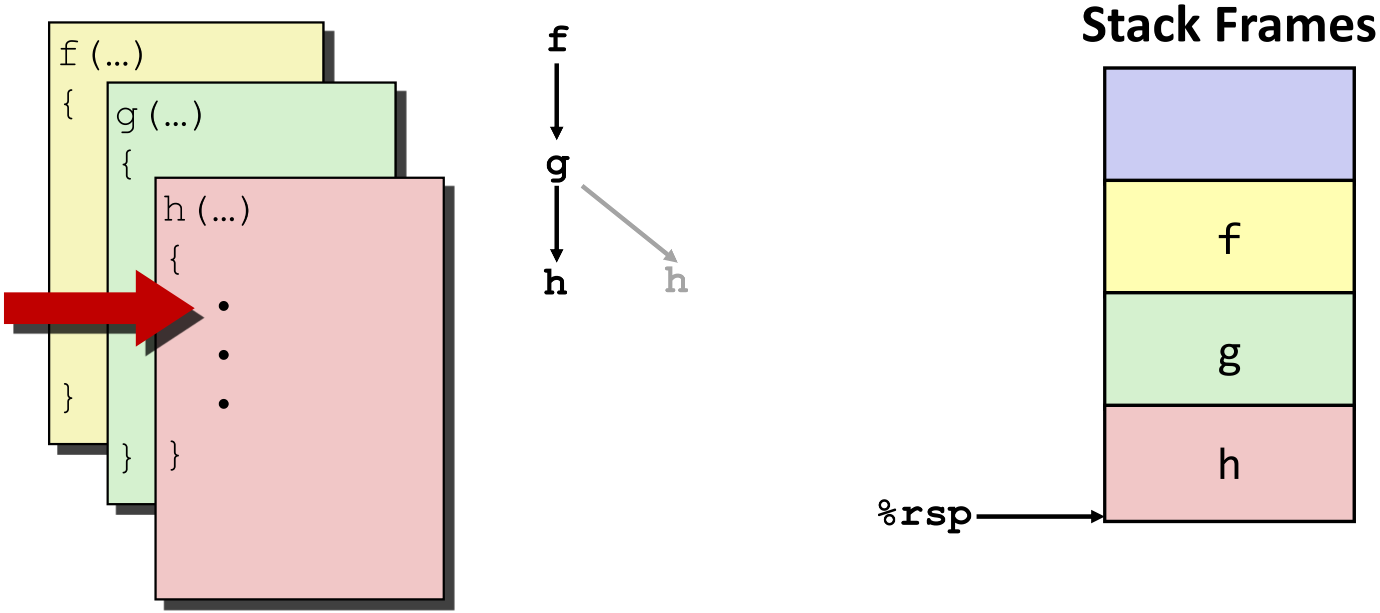


```
f(…)
{
}
```

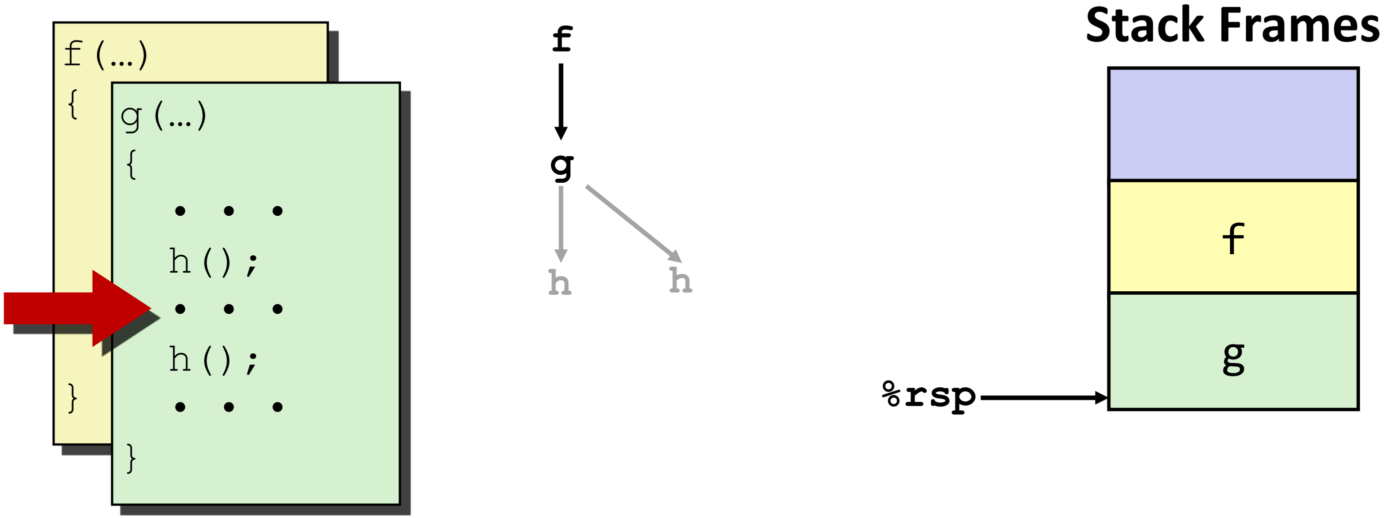
```
g(…)
{
  • • •
  h();
  • • •
  h();
  • • •
}
```

Stack Frames

Call Chain and Stack Frame



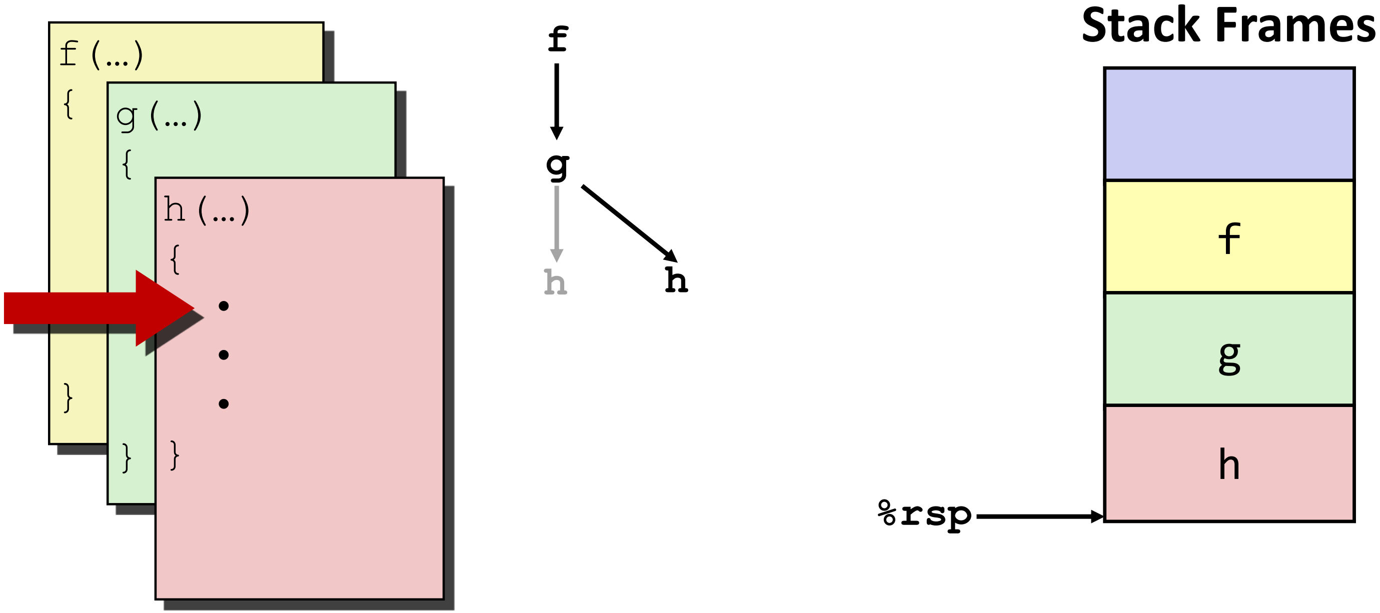
Call Chain and Stack Frame



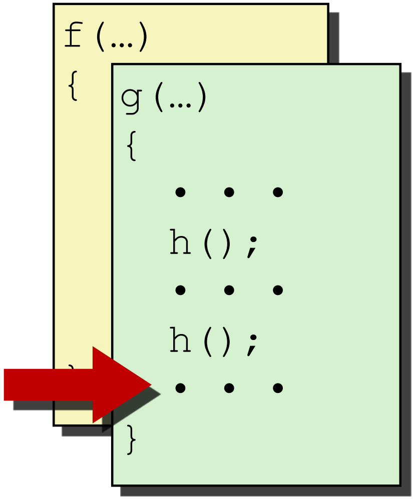
```
f(…)
{

}
```

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

Call Chain and Stack Frame

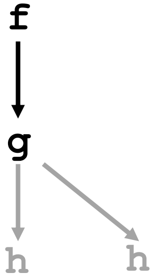


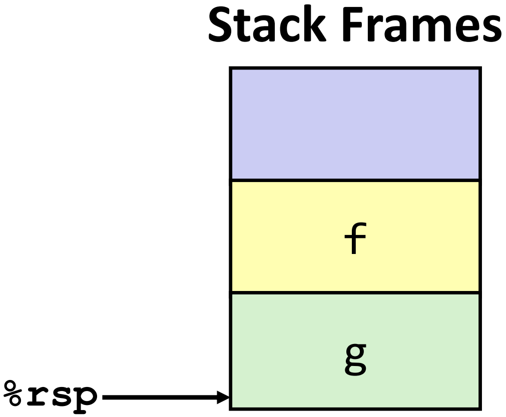
Call Chain and Stack Frame



```
f(…)
{
}
```

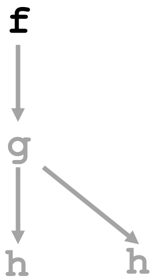
```
g(…)
{
   • • •
   h();
   • • •
   h();
   • • •
}
```



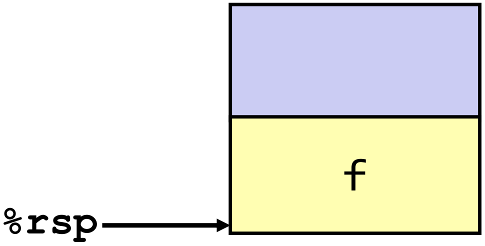


Call Chain and Stack Frame

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



Stack Frames



Stack Frame Example: incr

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

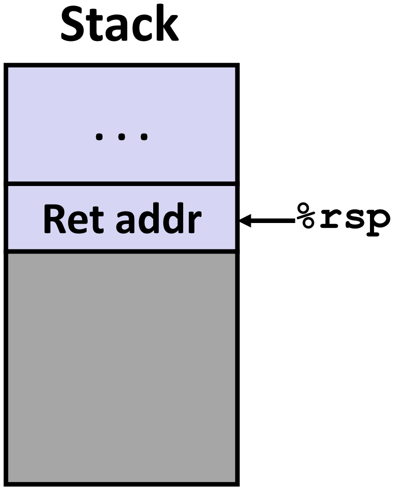
```
incr:
  0x401106 <+0>:   add  %rsi,(%rdi)
  0x401109 <+3>:   ret
```

Register	Use(s)
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**
`%rax`	Return value

Stack Frame Example: call_incr

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

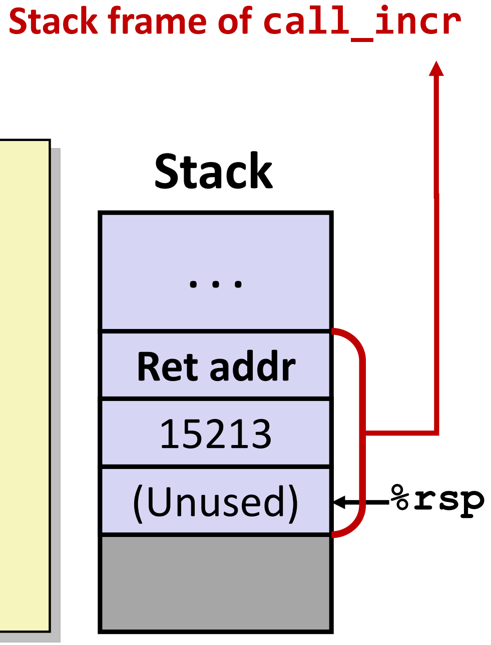
```
call_incr:
 0x40110a <+0>:    sub    $16,%rsp
 0x40110e <+4>:    movq   $15213,0x8(%rsp)
 0x401117 <+13>:   mov    $3000,%esi
 0x40111c <+18>:   lea    0x8(%rsp),%rdi
 0x401121 <+23>:   call   0x401106 <incr>
 0x401126 <+28>:   mov    0x8(%rsp),%rax
 0x40112a <+32>:   add    $16,%rsp
 0x40112e <+36>:   ret
```



Stack Frame Example: call_incr

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

```
call_incr:
 0x40110a <+0>:    sub    $16,%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    $16,%rsp
 0x40112e <+36>:   ret
```



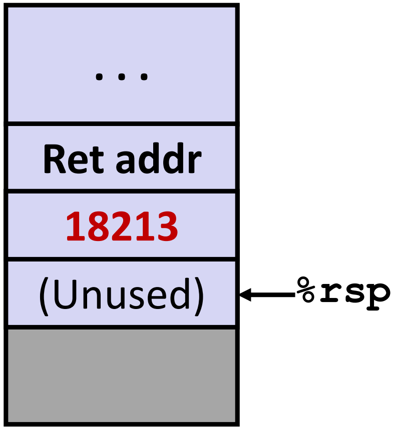
Stack Frame Example: call_incr

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

Register	Use(s)
%rdi	&v1
%rsi	3000

```
call_incr:
 0x40110a <+0>:    sub    $16,%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    $16,%rsp
 0x40112e <+36>:   ret
```

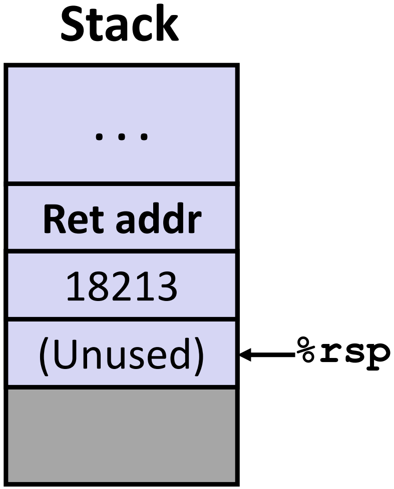
Stack



Stack Frame Example: call_incr

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

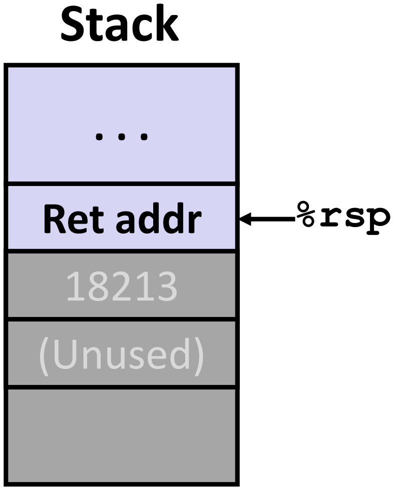
```
call_incr:
 0x40110a <+0>:   sub    $16,%rsp
 0x40110e <+4>:   movq   $15213,0x8(%rsp)
 0x401117 <+13>:  mov    $3000,%esi
 0x40111c <+18>:  lea    0x8(%rsp),%rdi
 0x401121 <+23>:  call   0x401106 <incr>
 0x401126 <+28>:  mov    0x8(%rsp),%rax
 0x40112a <+32>:  add    $16,%rsp
 0x40112e <+36>:  ret
```



Stack Frame Example: call_incr

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

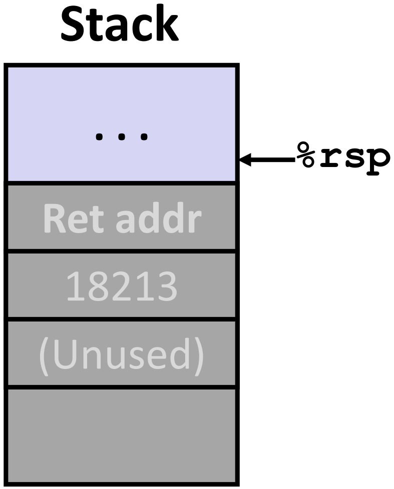
```
call_incr:
 0x40110a <+0>:   sub    $16,%rsp
 0x40110e <+4>:   movq   $15213,0x8(%rsp)
 0x401117 <+13>:  mov    $3000,%esi
 0x40111c <+18>:  lea    0x8(%rsp),%rdi
 0x401121 <+23>:  call   0x401106 <incr>
 0x401126 <+28>:  mov    0x8(%rsp),%rax
 0x40112a <+32>:  add    $16,%rsp
 0x40112e <+36>:  ret
```



Stack Frame Example: call_incr

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

```
call_incr:
 0x40110a <+0>:    sub     $16,%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     $16,%rsp
 0x40112e <+36>:   ret
```



Summary

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