Module III

Machine Language

Dr. Arijit Roy
Computer Science and Engineering Group
Indian Institute of Information Technology Sri City

The concepts are based on Carnegie Mellon University course and the text-book, authored by Patterson and Hennesy

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Data Format

Intel refer to a 16-bit data type as

Similarly, Intel refer to a 32-bit data type as

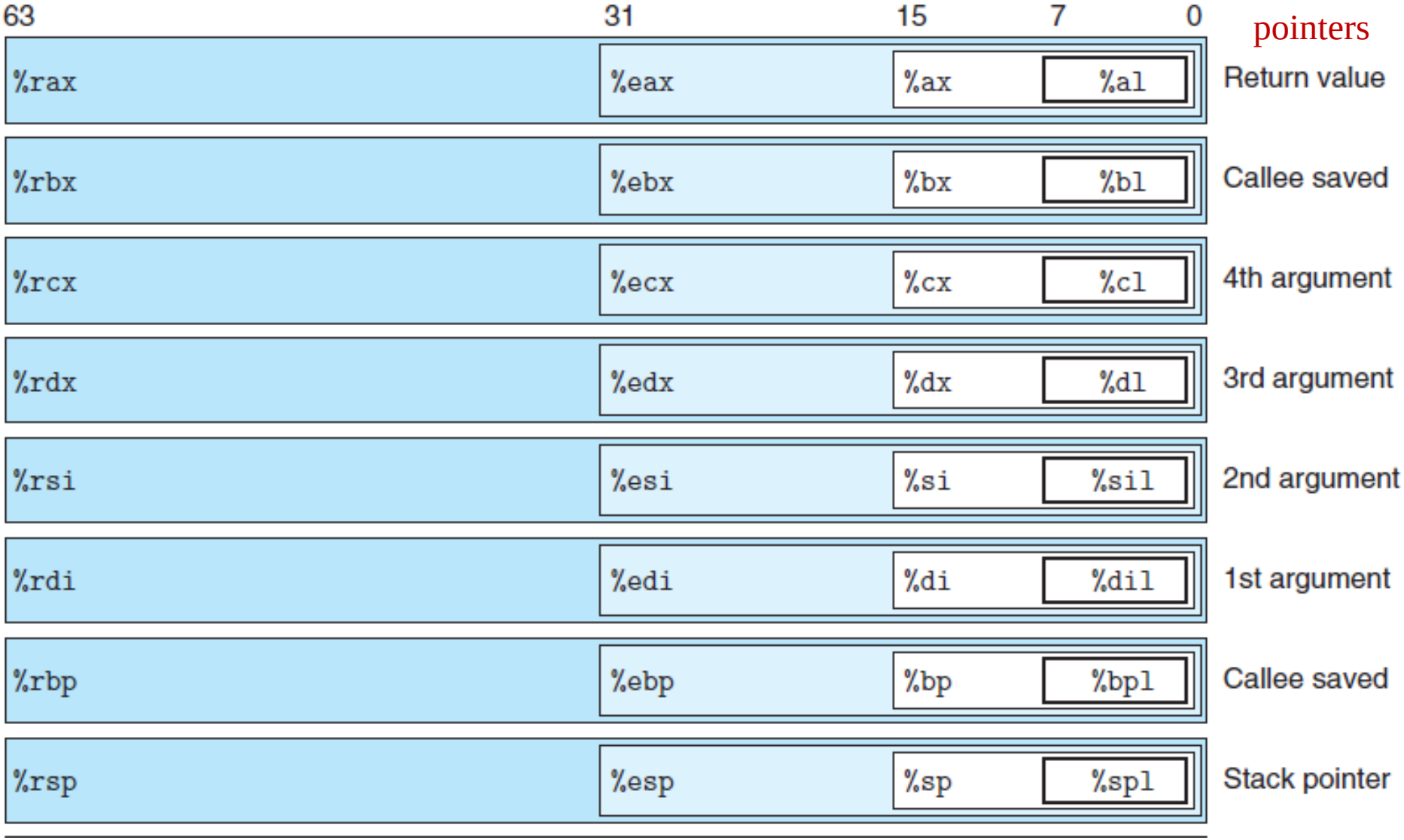
C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8

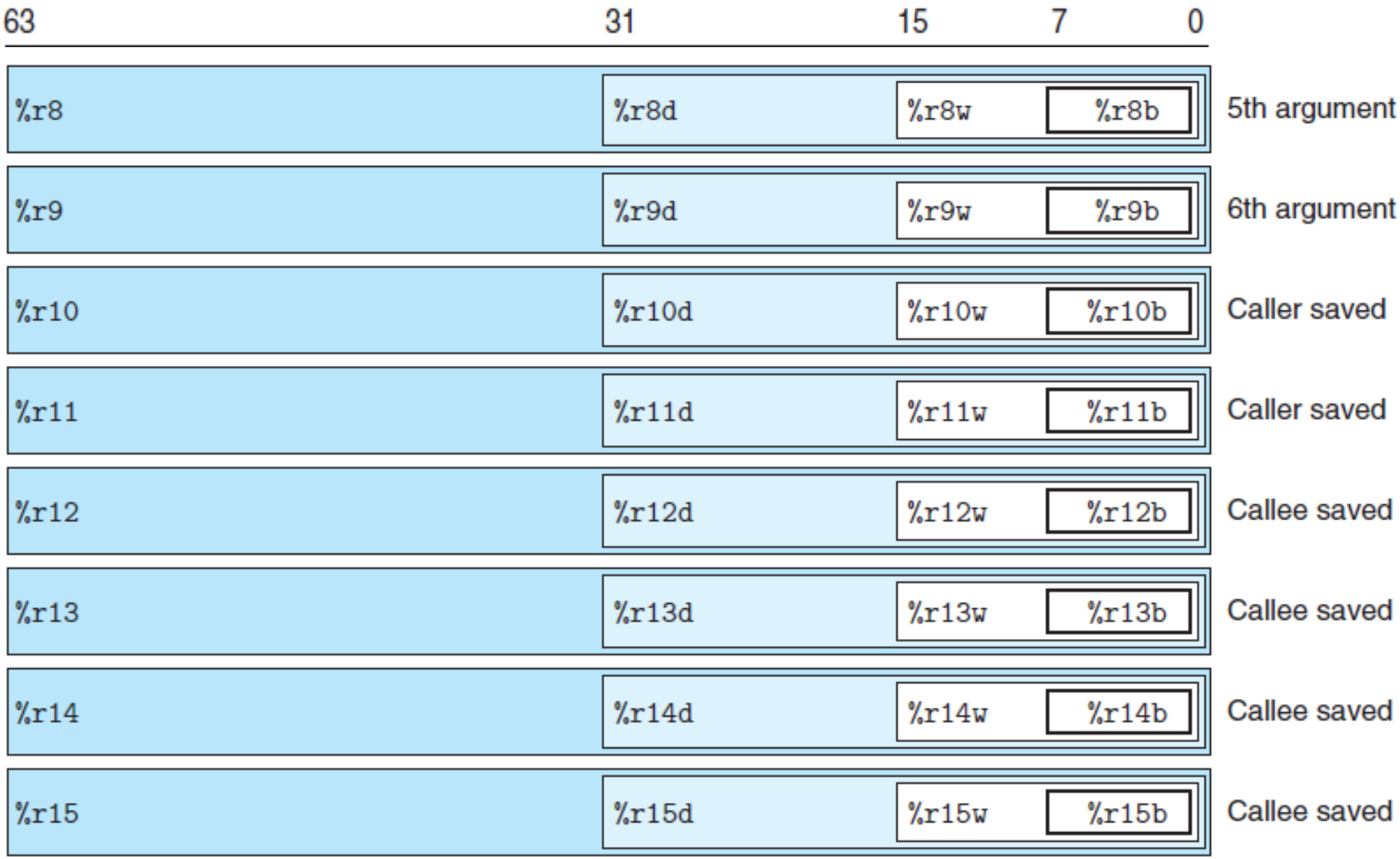
Figure 3.1 Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.

Most assembly code instructions generated by GCC have a single character suffix denoting the size of the operand.

Accessing Information

X86-64 CPU contains a set of 16 general purpose registers, storing 64 bits values, These are used to store integer data and pointers




63
31
15
7
0
%r8
%r8d
%r8w
%r8b
5th argument
%r9
%r9d
%r9w
%r9b
6th argument
%r10
%r10d
%r10w
%r10b
Caller saved
%r11
%r11d
%r11w
%r11b
Caller saved
%r12
%r12d
%r12w
%r12b
Callee saved
%r13
%r13d
%r13w
%r13b
Callee saved
%r14
%r14d
%r14w
%r14b
Callee saved
%r15
%r15d
%r15w
%r15b
Callee saved

Operand Specifiers – Addressing Modes

- Most instructions have one or more *operands* specifying the source values to use
- Operand types: Immediate, Register, Memory

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Source value: To use performing an operation – It can be given as constant or read from the register or memory

Destination value: Location in to which place the result

Immediate is for constant values, typically, $ is used

Register : denotes the content of a register, among one of the sixteen

$R[r_a]$: used as reference – viewing the set of registers as an array

$M_b[Addr]$: Used for memory reference. Memory is large array of bytes. Denotes b-byte values stored in memory starting at address Addr

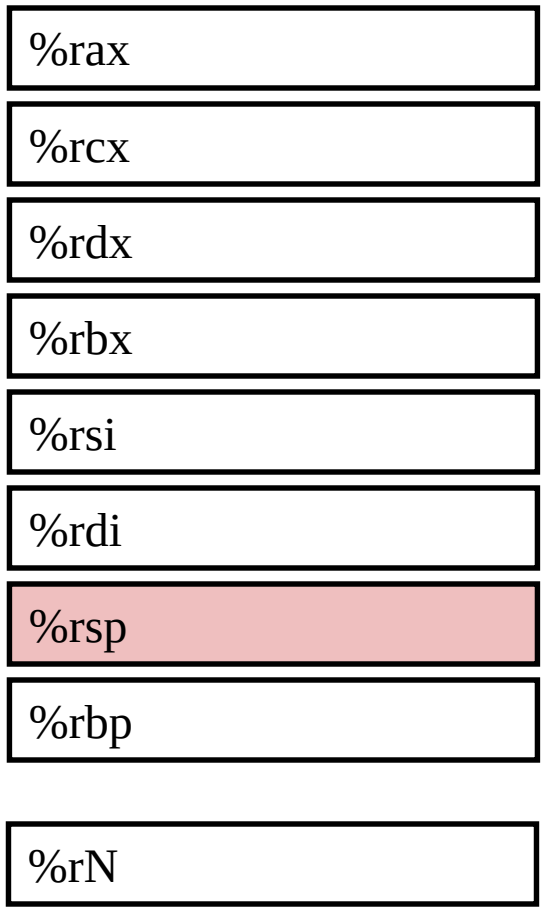
General form: Imm, base register, index register, and scale factor. Base and index must be 64-bit register, s must be 1, 2, 4, or 8.

Moving Data

- Moving Data

  movq *Source*, *Dest*:

- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: **$0x400, $-533**
    - Like C constant, but prefixed with **'$'**
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **%rax, %r13**
    - But **%rsp** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **(%rax)**
    - Various other "address modes"



Example

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Fill in the following table showing the values for the indicated operands:

Operand	Value
%rax	_____
0x104	_____
$0x108	_____
(%rax)	_____
4(%rax)	_____
9(%rax,%rdx)	_____
260(%rcx,%rdx)	_____
0xFC(,%rcx,4)	_____
(%rax,%rdx,4)	_____

Operand	Value	Comment
%rax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%rax)	0xFF	Address 0x100
4(%rax)	0xAB	Address 0x104
9(%rax,%rdx)	0x11	Address 0x10C
260(%rcx,%rdx)	0x13	Address 0x108
0xFC(,%rcx,4)	0xFF	Address 0x100
(%rax,%rdx,4)	0x11	Address 0x10C

movq Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	movq $0x4,%rax	temp = 0x4;
		Mem	movq $-147,(%rax)	*p = -147;
	Reg	*Reg*	movq %rax,%rdx	temp2 = temp1;
		Mem	movq %rax,(%rdx)	*p = temp;
	Mem	*Reg*	movq (%rax),%rdx	temp = *p;

Cannot do memory-memory transfer with a single instruction

Date Movement Instructions

- For most cases, the mov instructions will only update the specific register bytes or memory locations indicated by the destination operand.
- The only exception is that when movl has a register as the destination, it will also set the high-order 4 bytes of the register to 0.
- This exception arises from the convention, adopted in x86-64, that any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0.

Date Movement Instructions

`movabsq`	I, R	$R \leftarrow I$	Move absolute quad word

This instruction is for dealing with 64-bit immediate data.

The regular movq instruction can only have immediate source operands that can be represented as 32-bit two's-complement numbers. This value is then sign extended to produce the 64-bit value for the destination.

The movabsq instruction can have an arbitrary 64-bit immediate value as its source operand and can only have a register as a destination.

Aside Understanding how data movement changes a destination register

As described, there are two different conventions regarding whether and how data movement instructions modify the upper bytes of a destination register. This distinction is illustrated by the following code sequence:

```
1      movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2      movb    $-1, %al                     %rax = 0011223344556FF
3      movw    $-1, %ax                     %rax = 0011223344556FFF
4      movl    $-1, %eax                    %rax = 00000000FFFFFFFF
5      movq    $-1, %rax                    %rax = FFFFFFFFFFFFFFFF
```

In the following discussion, we use hexadecimal notation. In the example, the instruction on line 1 initializes register `%rax` to the pattern `0011223344556677`. The remaining instructions have immediate value −1 as their source values. Recall that the hexadecimal representation of −1 is of the form `FF`$\cdots$`F`, where the number of `F`'s is twice the number of bytes in the representation. The `movb` instruction (line 2) therefore sets the low-order byte of `%rax` to `FF`, while the `movw` instruction (line 3) sets the low-order 2 bytes to `FFFF`, with the remaining bytes unchanged. The `movl` instruction (line 4) sets the low-order 4 bytes to `FFFFFFFF`, but it also sets the high-order 4 bytes to `00000000`. Finally, the `movq` instruction (line 5) sets the complete register to `FFFFFFFFFFFFFFFF`.

Instruction	Effect	Description
MOVZ S, R	$R \leftarrow$ ZeroExtend(S)	Move with zero extension
`movzbw`		Move zero-extended byte to word
`movzbl`		Move zero-extended byte to double word
`movzwl`		Move zero-extended word to double word
`movzbq`		Move zero-extended byte to quad word
`movzwq`		Move zero-extended word to quad word

Figure 3.5 Zero-extending data movement instructions. These instructions have a register or memory location as the source and a register as the destination.

Instruction	Effect	Description
MOVS S, R	$R \leftarrow$ SignExtend(S)	Move with sign extension
`movsbw`		Move sign-extended byte to word
`movsbl`		Move sign-extended byte to double word
`movswl`		Move sign-extended word to double word
`movsbq`		Move sign-extended byte to quad word
`movswq`		Move sign-extended word to quad word
`movslq`		Move sign-extended double word to quad word
`cltq`	%rax $\leftarrow$ SignExtend(%eax)	Sign-extend %eax to %rax

Figure 3.6 Sign-extending data movement instructions. The MOVS instructions have a register or memory location as the source and a register as the destination. The `cltq` instruction is specific to registers %eax and %rax.

Practice Problem

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands. (For example, `mov` can be rewritten as `movb`, `movw`, `movl`, or `movq`.)

```
mov__   %eax, (%rsp)
mov__   (%rax), %dx
mov__   $0xFF, %bl
mov__   (%rsp,%rdx,4), %dl
mov__   (%rdx), %rax
mov__   %dx, (%rax)
```

Practice Problem

Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line.

```
movb $0xF, (%ebx)
movl %rax, (%rsp)
movw (%rax),4(%rsp)
movb %al,%sl
movq %rax,$0x123
movl %eax,%rdx
movb %si, 8(%rbp)
```

Example

(b) Assembly code

```
    long exchange(long *xp, long y)
    xp in %rdi, y in %rsi
1   exchange:
2     movq    (%rdi), %rax     Get x at xp. Set as return value.
3     movq    %rsi, (%rdi)     Store y at xp.
4     ret                      Return.
```

Figure 3.7 C and assembly code for exchange routine. Registers %rdi and %rsi hold parameters xp and y, respectively.

Example

- xp is a pointer to a long integer
- y is a long integer itself
- Indicates that we should read the value stored in the location designated by xp and store it as a local variable named x – *pointer dereferencing*
- It writes the value of parameter y at the location designated by xp

(a) C code

```
long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

(b) Assembly code

```
    long exchange(long *xp, long y)
    xp in %rdi, y in %rsi
1   exchange:
2     movq    (%rdi), %rax    Get x at xp. Set as return value.
3     movq    %rsi, (%rdi)    Store y at xp.
4     ret                     Return.
```

Figure 3.7 C and assembly code for exchange routine. Registers %rdi and %rsi hold parameters xp and y, respectively.

Pushing and Popping Stack Data

Instruction	Effect	Description
pushq S	R[%rsp] ← R[%rsp] − 8; M[R[%rsp]] ← S	Push quad word
popq D	D ← M[R[%rsp]]; R[%rsp] ← R[%rsp] + 8	Pop quad word

Figure 3.8 **Push and pop instructions.**

- Data movement operations: Push and Pop
- Stack plays a vital role in procedure calls
- Procedure call is an important and frequently used programming construct for a compiler
- Stack– a data structure—LIFO
- Program stack is stored in some region of the memory
- Top element has the lowest address
- %rsp holds the address of the top stack element

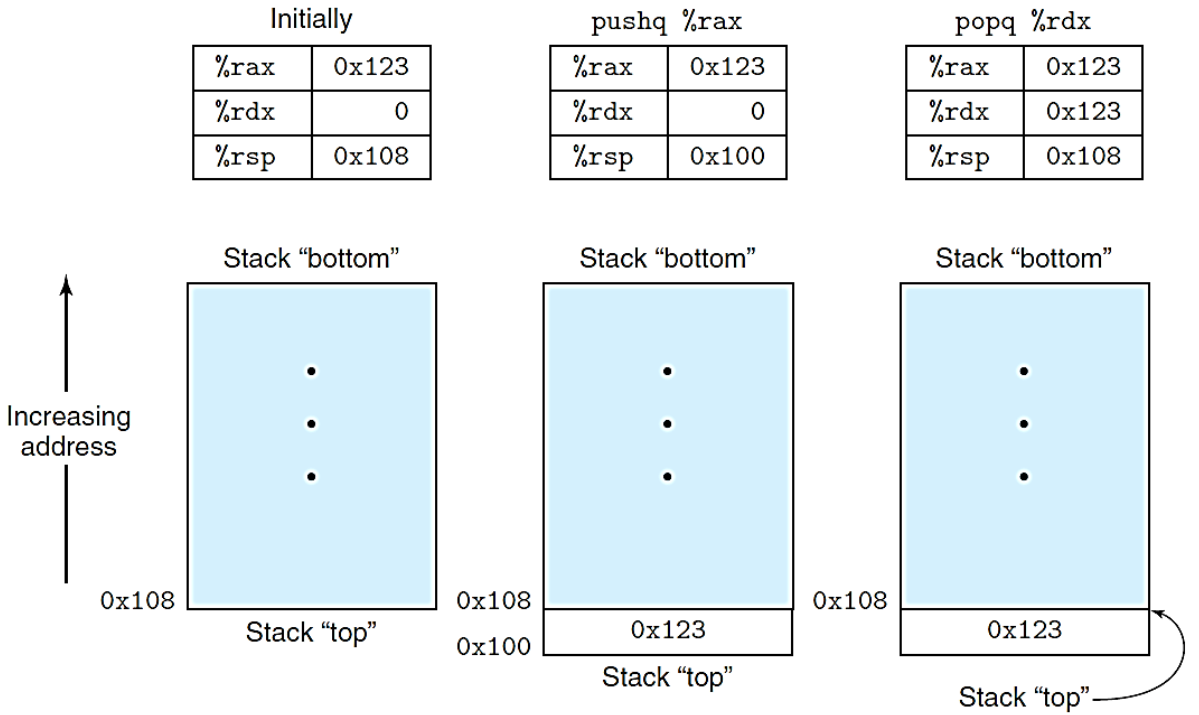


Figure 3.9 **Illustration of stack operation.** By convention, we draw stacks upside down, so that the "top" of the stack is shown at the bottom. With x86-64, stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register %rsp) and storing to memory, while popping involves reading from memory and incrementing the stack pointer.

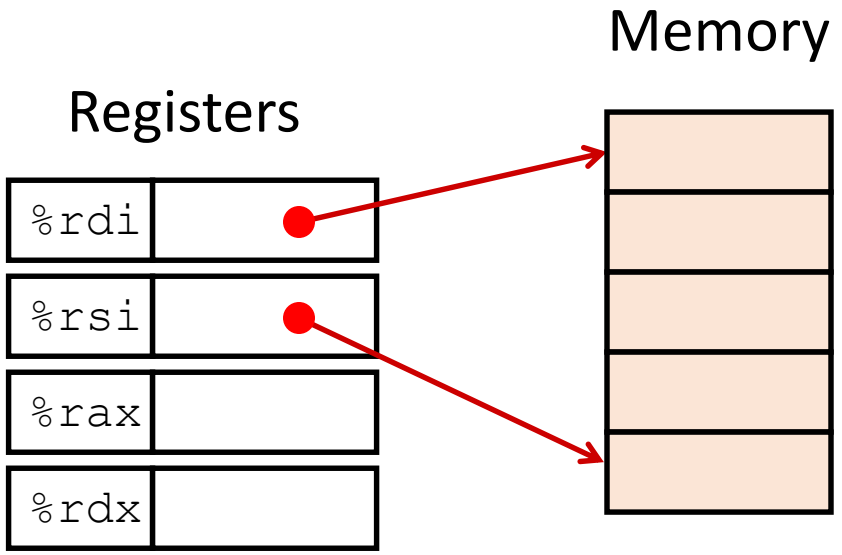
Example of Simple Addressing Modes

```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   ret
```

Understanding Swap()

```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()

Registers

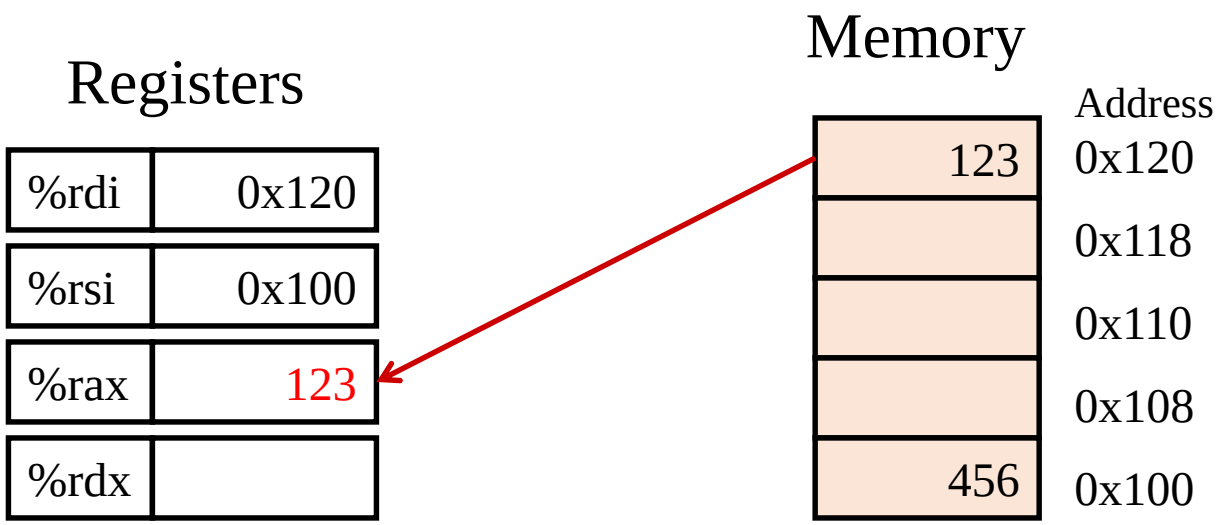
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

	Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

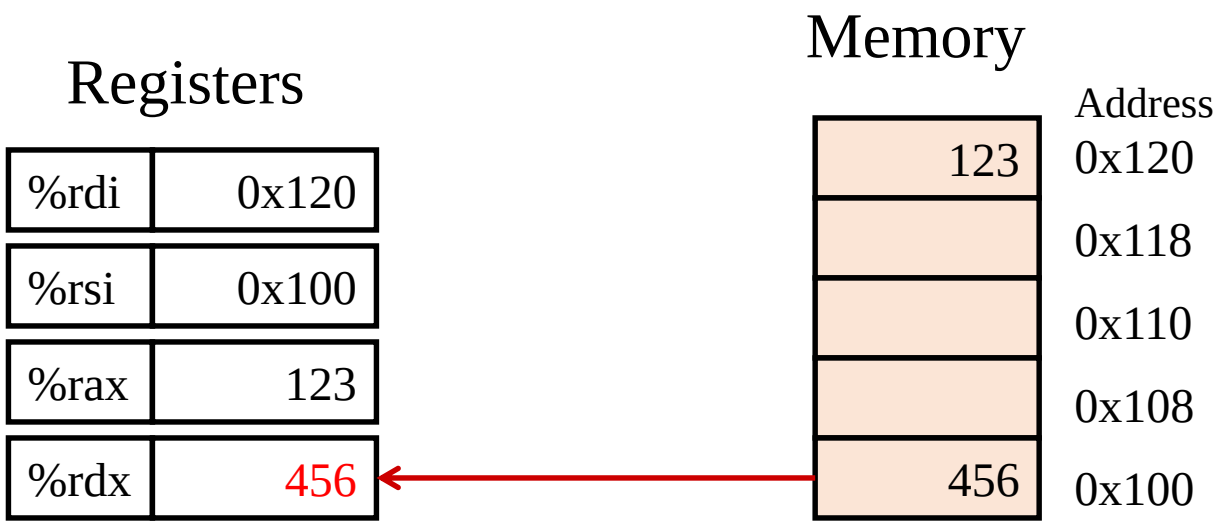
```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding Swap()



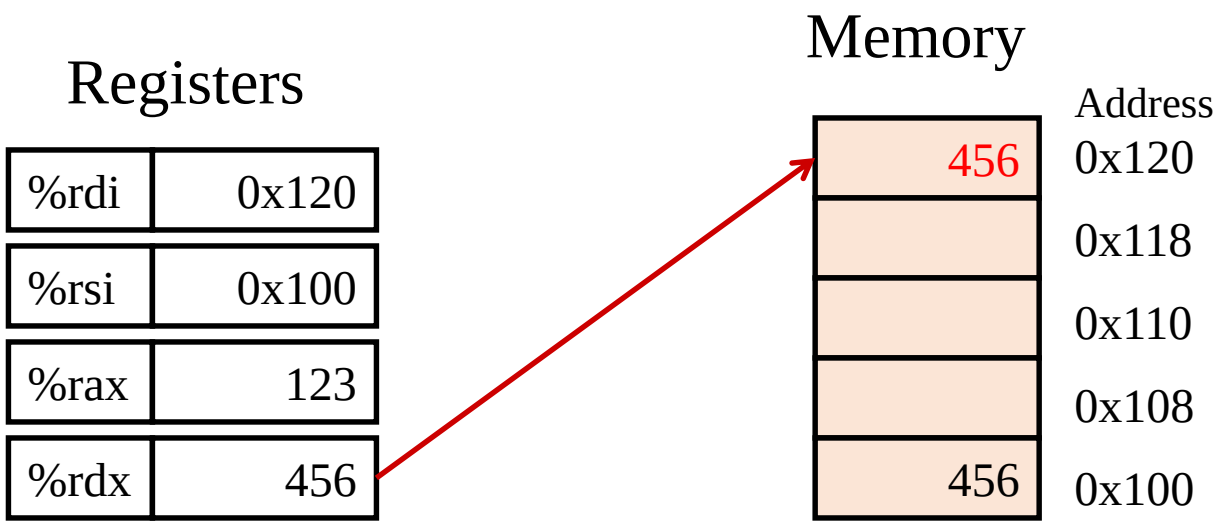

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding Swap()



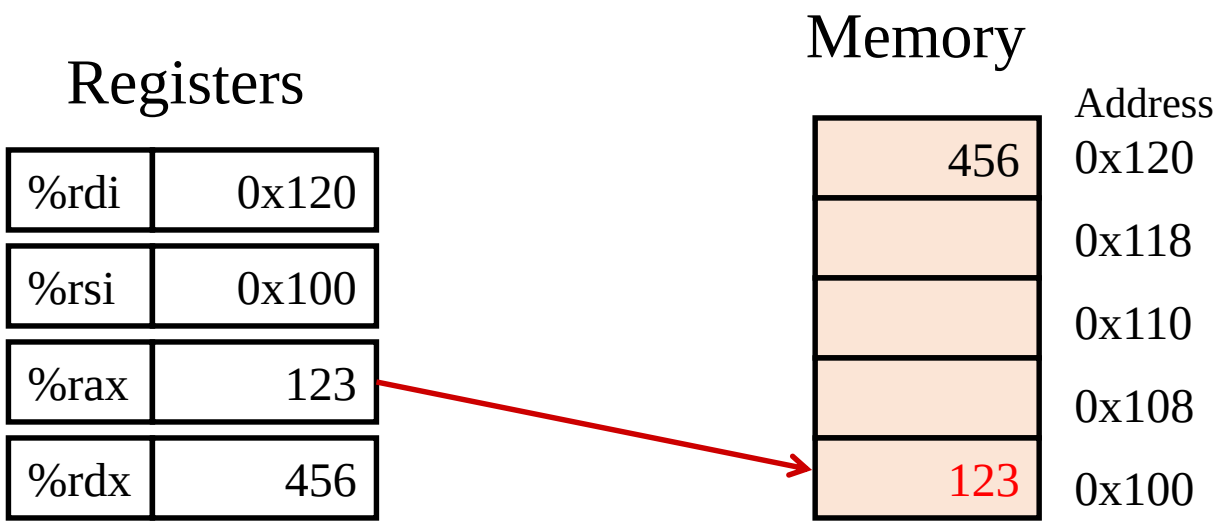

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding Swap()




```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding Swap()




```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```