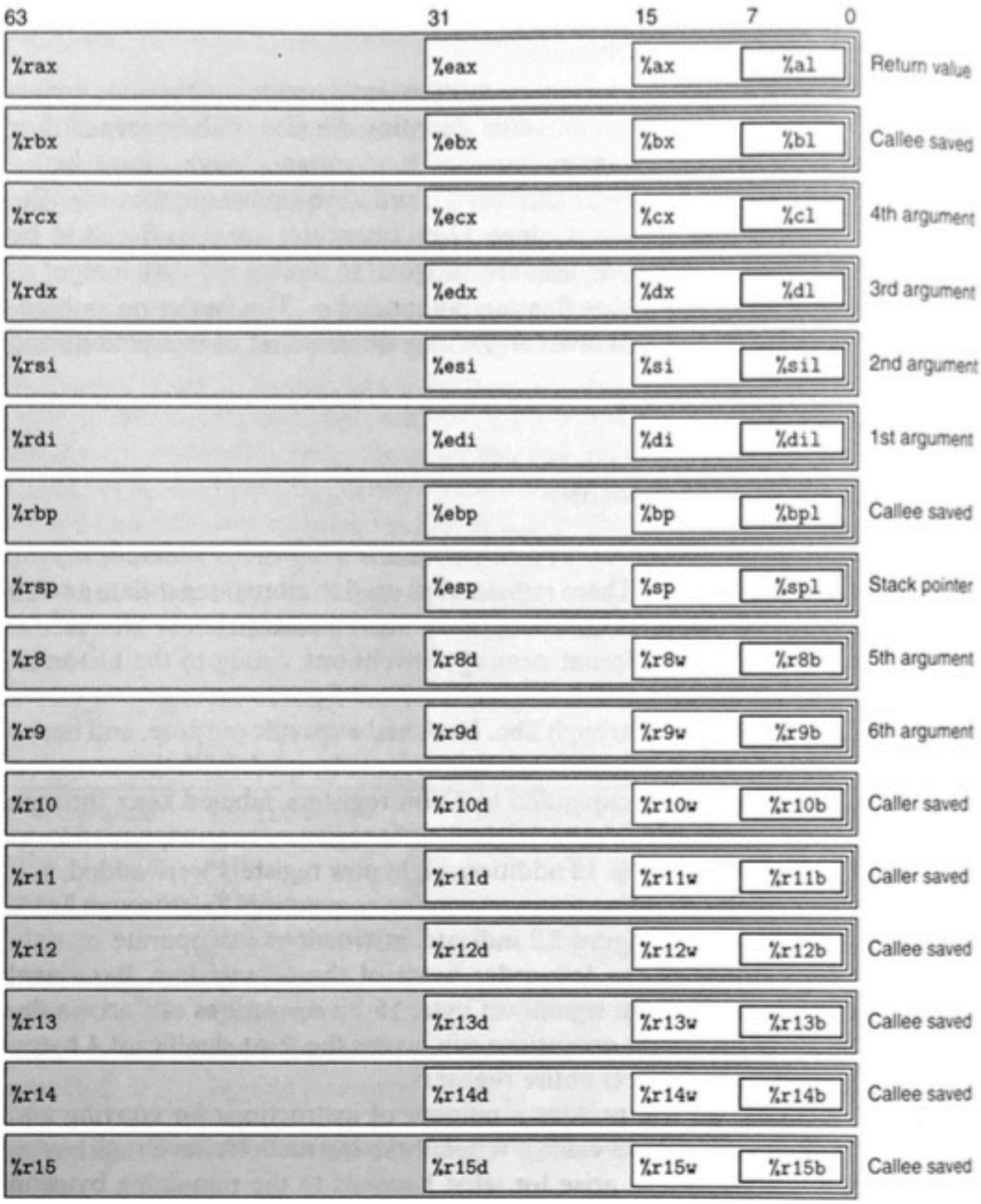
Chapter 3.4: Accessing Information



3.4.1: Operand Specifiers

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

Practice Problem 3.1

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%rax	0x100
%rcx	0x1
%rdx	0x3

Answers

Operand	Value	Comment
%rax	0x100	Register
0x104	0xAB	Mem access
$0x108	0x108	Immediate
(%rax)	0xFF	Mem access
4(%rax)	0xAB	Mem access 0x104
9(%rax, %rdx)	0x11	Mem access 0x10C
260(%rcx, %rdx)	0x13	Mem access 0x108
0xFC(,%rcx,4)	0x100	MA 252 + 4 = 256 = 0x100

(%rax, %rdx, 4)	0x11	MA 0x100 + 4*(0x3) = 0x10C

3.4.2: Data Movement Instructions

MOV

Instruction	Effect	Description
all	S,D move S to D	Move
movb		Move byte
movw		Move word
movl		Move double word
movq		Move quad word
movabsq	I,R move I to R	Move absolute quad word

- x86-64 does not allow both operands for a `mov` instruction to be memory locations
  - copying a value from one memory location to requires two steps: move from mem to reg, from reg to mem
- for most cases, the `MOV` instructions will only update the specific register bytes or memory locations indicated by the destination operand
  - one exception is that if `movl` has a register as the destination, it will also set the high-order 4 bytes of the register to 0
    - `0000 0000 0000 0000 0000 0000 0000 0000`
    - `0101 1111 1011 1111 1000 1001 1010 0010`
  - 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**

```
; This example shows the five possible combinatiosn of source and destination types

    movl $0x4050,%eax       ; Immediate--Register,   4 bytes
    movw %bp,%sp            ; Register--Register,    2 bytes
    movb (%rdi, %rcx),%al   ; Memory--Register,      1 byte
    movb $-17, (%esp)       ; Immediate--Memory,     1 byte
    movq %rax,-12(%rbp)     ; Register--Memory,      8 bytes
```

- the regular `movq` instruction can only have immediate source operands that can be represented as 32-bit two's-complement numbers, which is then sign extended to produce the 64-bit value for the destination
- `movabsq` can have an arbitrary 64-bit immediate value as its source operand and can only have a register as a destination

`MOVZ` and `MOVS`

- instructions in the `MOVZ` class fill out the remaining bytes of the destination with zeros
- instructions in the `MOVS` class fill them out by sign extension, replicating copies of the most significant bit of the source operand
- `movzlq` (move a double word to a quad word) can be implemented using a `movl` instruction with a register as the destination
    - an instruction generating a 4-byte value with a register destination will fill the upper 4 bytes with 0's

MOVZ

Given, S,R, these all move move ZeroExtend(S) to R with zero-extension

Instruction	Description
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

MOVS

Given, S,R, these all move move SignExtend(S) to R with sign-extension

Instruction	Description
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	Short for movslq %eax,%rax

Practice Problem 3.2

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands.
Hint: What is the smallest size among the source and destination?

```
    mov__   %eax, (%rsp)             ; l
    mov__   (%rax), %dx              ; w
    mov__   $0xFF, %b1               ; b
    mov__   (%rsp, %rdx, 4), %dl     ; b
    mov__   (%rdx), %rax             ; q
    mov__   %dx, (%rax)              ; w
```

Practice Problem 3.3

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)    ; Cannot use %ebr as address register
    movl %rax, (%rsp)   ; Mismatch bewteen instruction suffix and register ID
    movw (%rax),4(%rsp) ; Cannot have both source and destination be memory
    movb %al,%sl        ; No register named %sl
    movq %rax,$0x123    ; Cannot have immediate as destination
    movl %eax,%rdx      ; Destination operand incorrect size
    movb %si, 8(%rbp)   ; Mismatch between instruction suffix and register ID
```

3.4.3: Data Movement Example

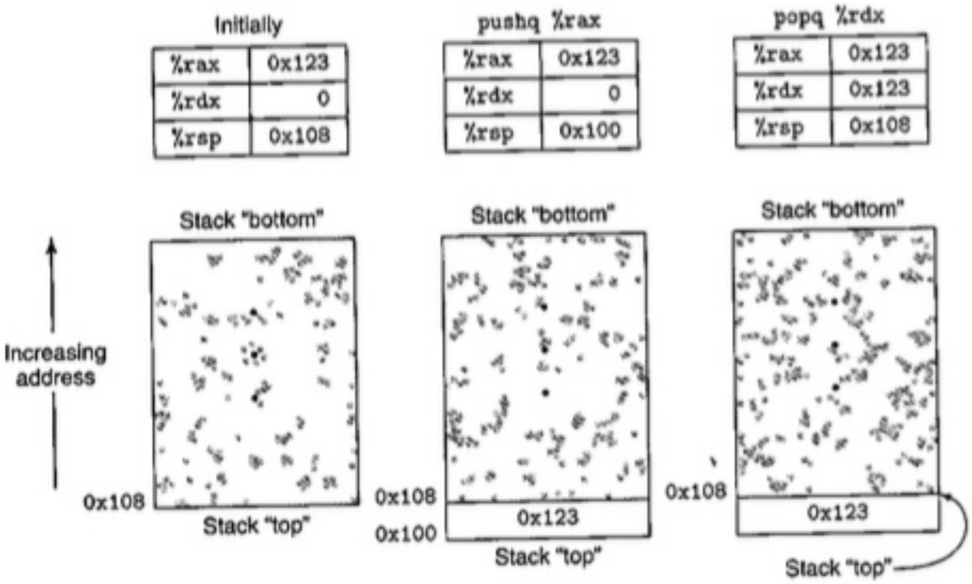
- a function returns a value by storing it in register `%rax`, or in one of the low-order positions of this register
- what we call “pointers” in C are simply addresses
  - dereferencing one involves copying that pointer into a register, and the using this register in a memory reference
- local variables, such as x, are often kept in registers rather than stored in memory locations
  - register access is much faster than memory access

```
long exchange(long *xp, long y) {
    long x = *xp;   // set x to the value stored in ptr xp
    *xp = y;        // change xp to have the value of y
    return x;       // return x
}
```

```
; long exchange(long *xp, long y)
; 1st arg (xp) in %rdi, 2nd arg (y) in %rsi
exchange:
    movq (%rdi), %rax   ; Get x from ML xp. Set as return value.
    movq %rsi, (%rdi)   ; Store y at ML xp.
    ret                 ; Return (x saved in %rax).
```

3.4.4: Pushing and Popping Stack Data

- the stack plays a vital role in the handling of procedure calls
- a data structure where values can be added or deleted, but only on a “last-in first-out” basis
    - add data via a *push* operation and remove via a *pop* operation
- in x86-64, stacks start at a high address and grow toward lower addresses
    - pushing, therefore, involves decrementing the stack pointer (register %rsp) and storing to memory
    - popping involves reading from memory and incrementing the stack pointer



- the stack pointer, `%rsp`, holds the address of the top stack element
- `pushq` instruction provides ability to push data onto the stack
    - takes one operand: data source for pushing
- `popq` instruction pops it
    - takes one operand: data destination for popping

1. pushing a quad word value onto the stack involves first decrementing the stack pointer by 8 bytes and then writing the value at the new top-of-stack address
    a. the behavior of `pushq %rbp` is equivalent to that of, but more efficient than
        i. `subq $8,%rsp       Decrement stack pointer`
        ii. `movq %rbp,(%rsp)   Store %rbp on stack`
2. popping a quad word involves reading from the top-of-stack location and then incrementing the stack pointer by 8 bytes
    a. the behavior of `popq %rax` is equivalent to that of, but more efficient than
        i. `movq (%rsp),%rax   Read %rax from top of stack`
        ii. `addq $8,%rsp       Increment stack pointer`
    b. popping does not change the value that was stored at the previous stack top
        i. it does not overwrite the value until pushed onto again
3. arbitrary positions within the stack can be accessed using standard memory addressing methods
    a. `movq 8(%rsp), %rdx` will copy the **second** quad word from the stack to register `%rdx`