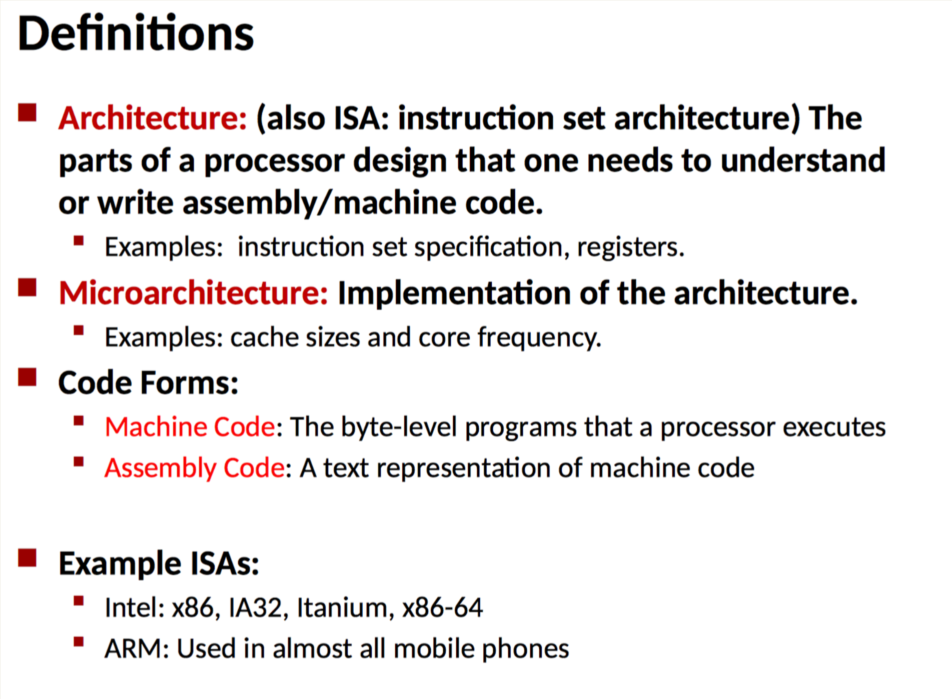
CS 211 RECITATIONS WEEK 8

Wenjie Qiu
Teaching Assistant
Office Hour: Thu. noon – 1pm
wenjie.qiu@rutgers.edu

Content

- Architectures & Microarchitectures
- Assembly Programmer's View
  - *Machine code, assembly code, registers.*
- X86 Instructions – Movement
- X86 Instructions – Arithmetic

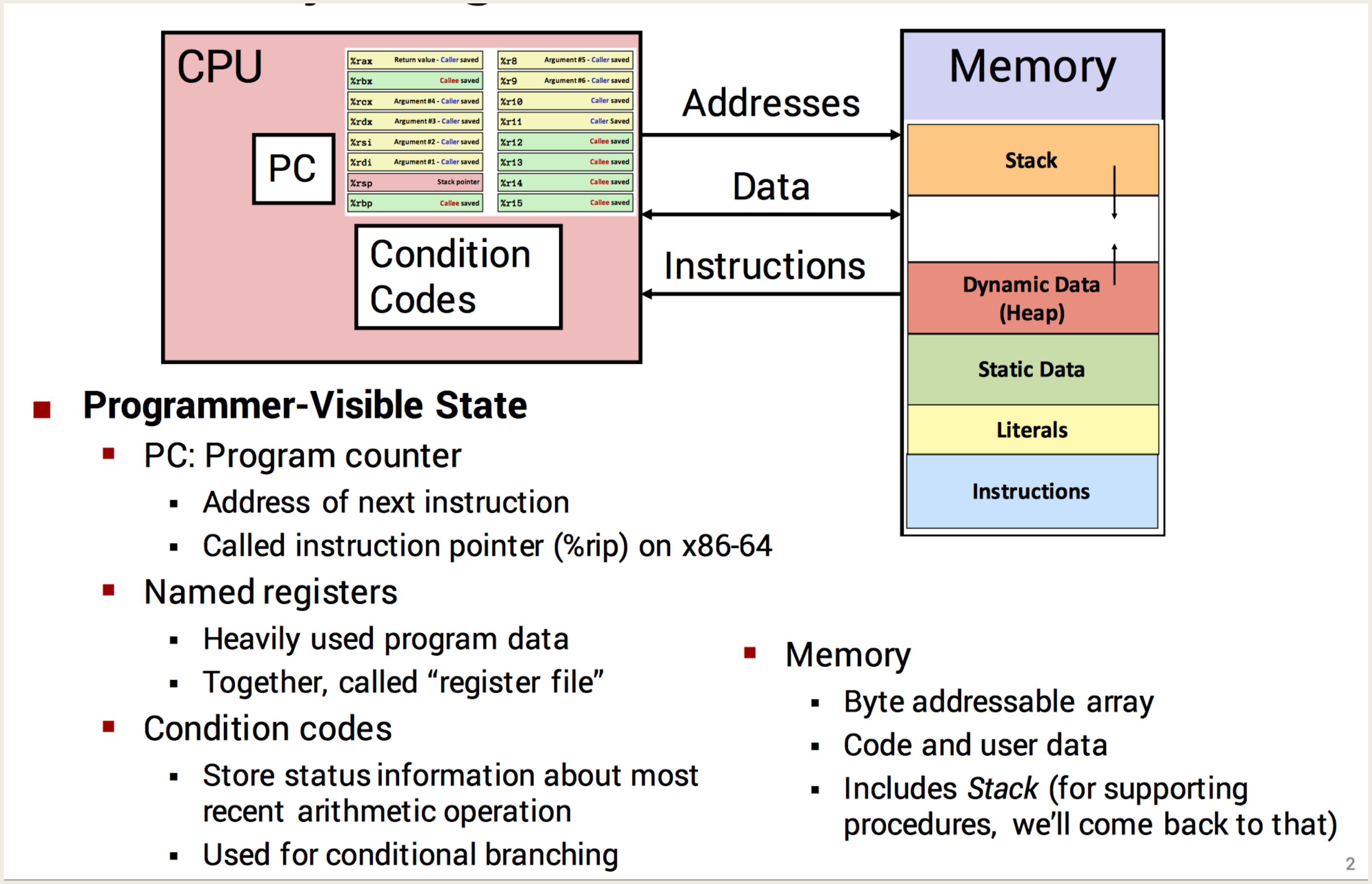
ARCHITECTURES & MICROARCHITECTURES



Definitions

- **Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.**
  - Examples: instruction set specification, registers.
- **Microarchitecture: Implementation of the architecture.**
  - Examples: cache sizes and core frequency.
- **Code Forms:**
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code
- **Example ISAs:**
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones

Assembly Programmer’s View

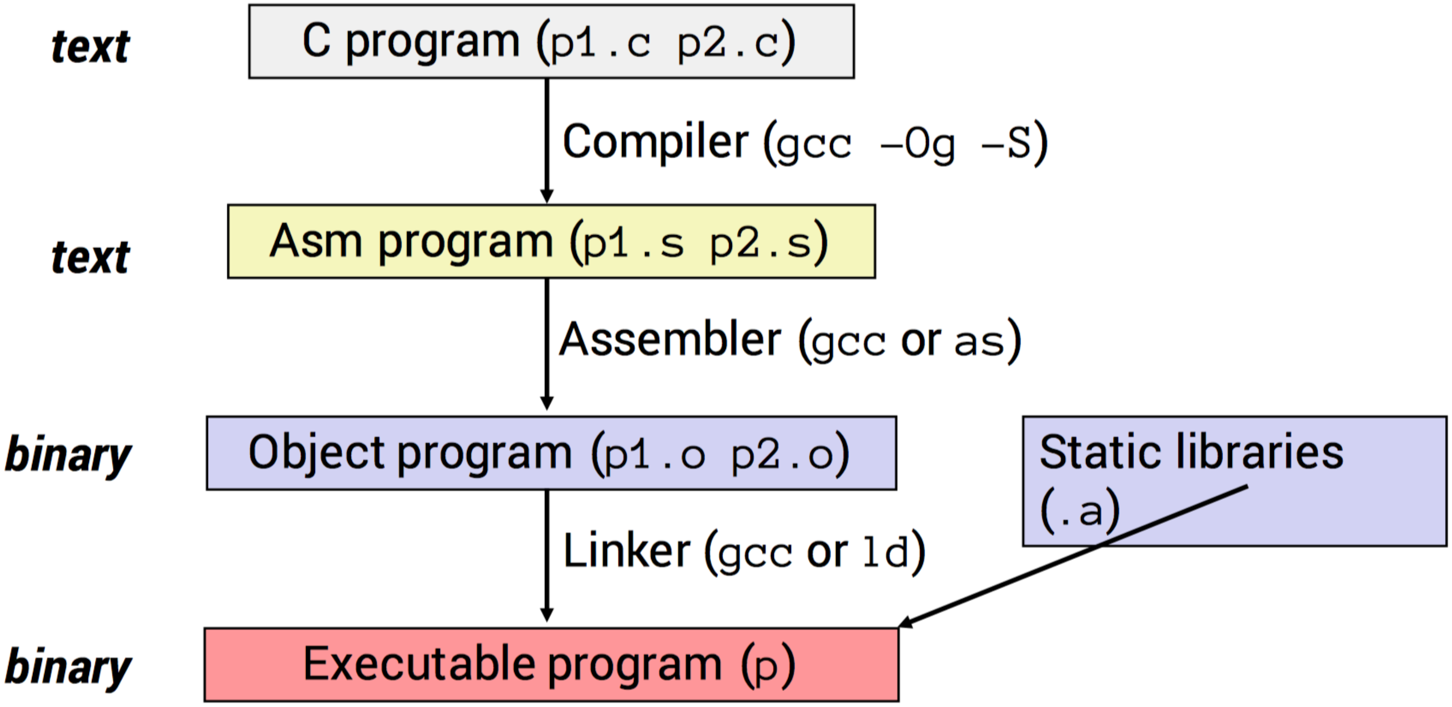


- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called instruction pointer (%rip) on x86-64
  - Named registers
    - Heavily used program data
    - Together, called “register file”
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- Memory
  - Byte addressable array
  - Code and user data
  - Includes *Stack* (for supporting procedures, we’ll come back to that)

Turning C into Object Code

- **Code in files `p1.c p2.c`**
- **Compile with command: `gcc –Og p1.c p2.c –o p`**
  - Use basic optimizations (`–Og`) [New to recent versions of GCC]
  - Put resulting machine code in file `p`



text — C program (`p1.c p2.c`)

Compiler (`gcc –Og –S`)

text — Asm program (`p1.s p2.s`)

Assembler (`gcc` or `as`)

binary — Object program (`p1.o p2.o`)

Static libraries (`.a`)

Linker (`gcc` or `ld`)

binary — Executable program (`p`)

Assembling

Assembly has **labels**.

```
pcount_r:
    movl     $0, %eax
    testq    %rdi, %rdi
    je       .L6
    pushq    %rbx
    movq     %rdi, %rbx
    shrq     %rdi
    call     pcount_r
    andl     $1, %ebx
    addq     %rbx, %rax
    popq     %rbx
.L6:
    rep ret
```

gcc -S pcount.c

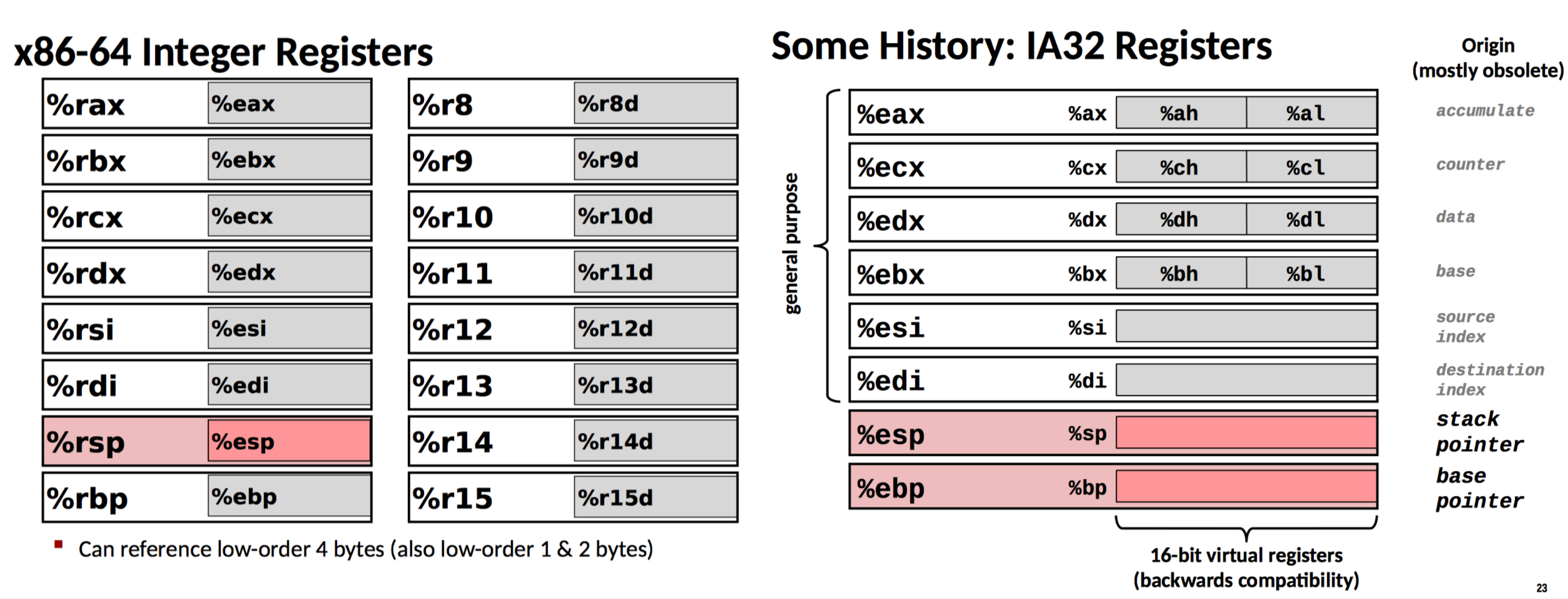
assembler

Executable has **addresses**.

```
00000000004004f6 <pcount_r>:
  4004f6:  b8 00 00 00 00      mov    $0x0,%eax
  4004fb:  48 85 ff            test   %rdi,%rdi
  4004fe:  74 13               je     400513 <pcount_r+0x1d>
  400500:  53                  push   %rbx
  400501:  48 89 fb            mov    %rdi,%rbx
  400504:  48 d1 ef            shr    %rdi
  400507:  e8 ea ff ff ff      callq  4004f6 <pcount_r>
  40050c:  83 e3 01            and    $0x1,%ebx
  40050f:  48 01 d8            add    %rbx,%rax
  400512:  5b                  pop    %rbx
  400513:  f3 c3               repz retq
```

gcc -g pcount.c -o pcount
objdump -d pcount

Assembly Programmer’s View



x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

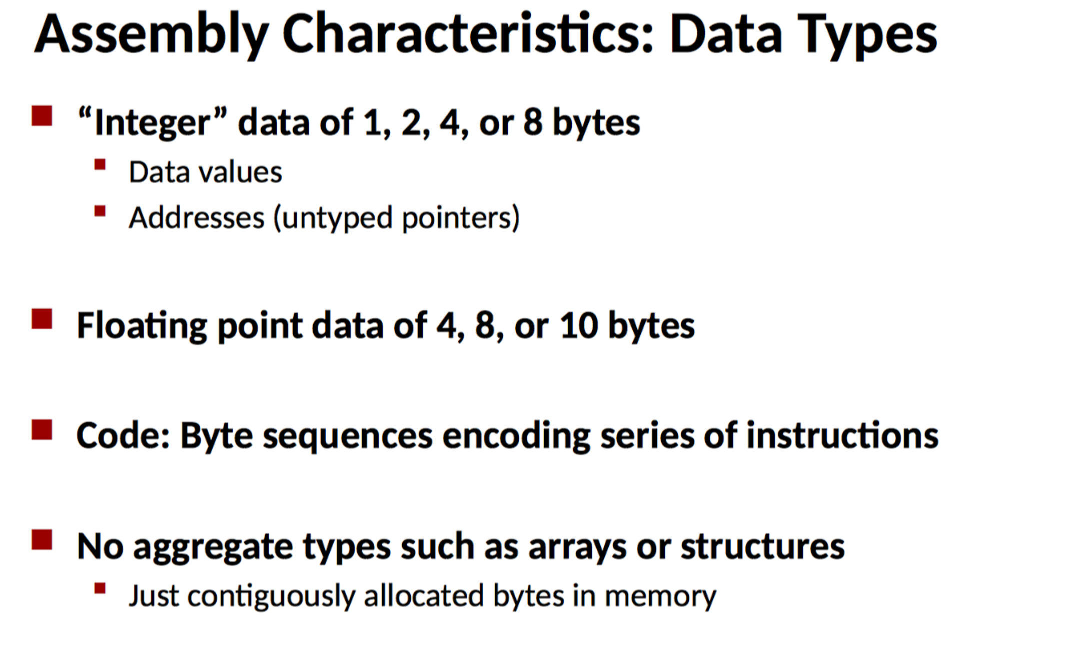
Some History: IA32 Registers

general purpose

				Origin (mostly obsolete)
%eax	%ax	%ah	%al	*accumulate*
%ecx	%cx	%ch	%cl	*counter*
%edx	%dx	%dh	%dl	*data*
%ebx	%bx	%bh	%bl	*base*
%esi	%si			*source index*
%edi	%di			*destination index*
%esp	%sp			***stack pointer***
%ebp	%bp			***base pointer***

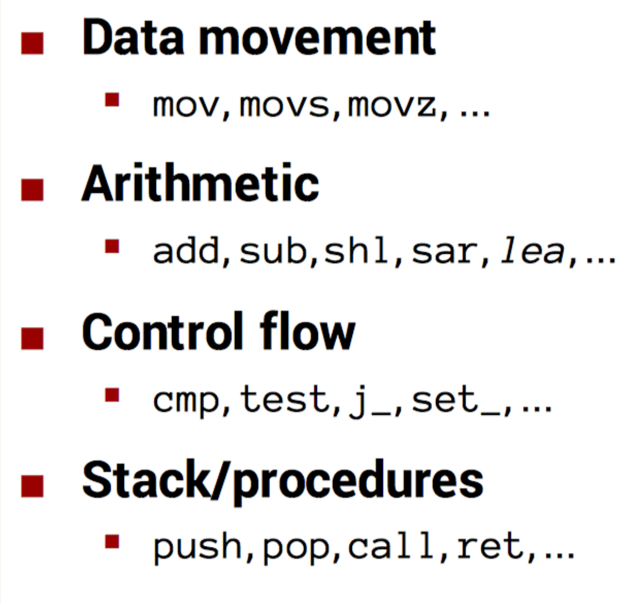
16-bit virtual registers
(backwards compatibility)

Assembly Programmer’s View



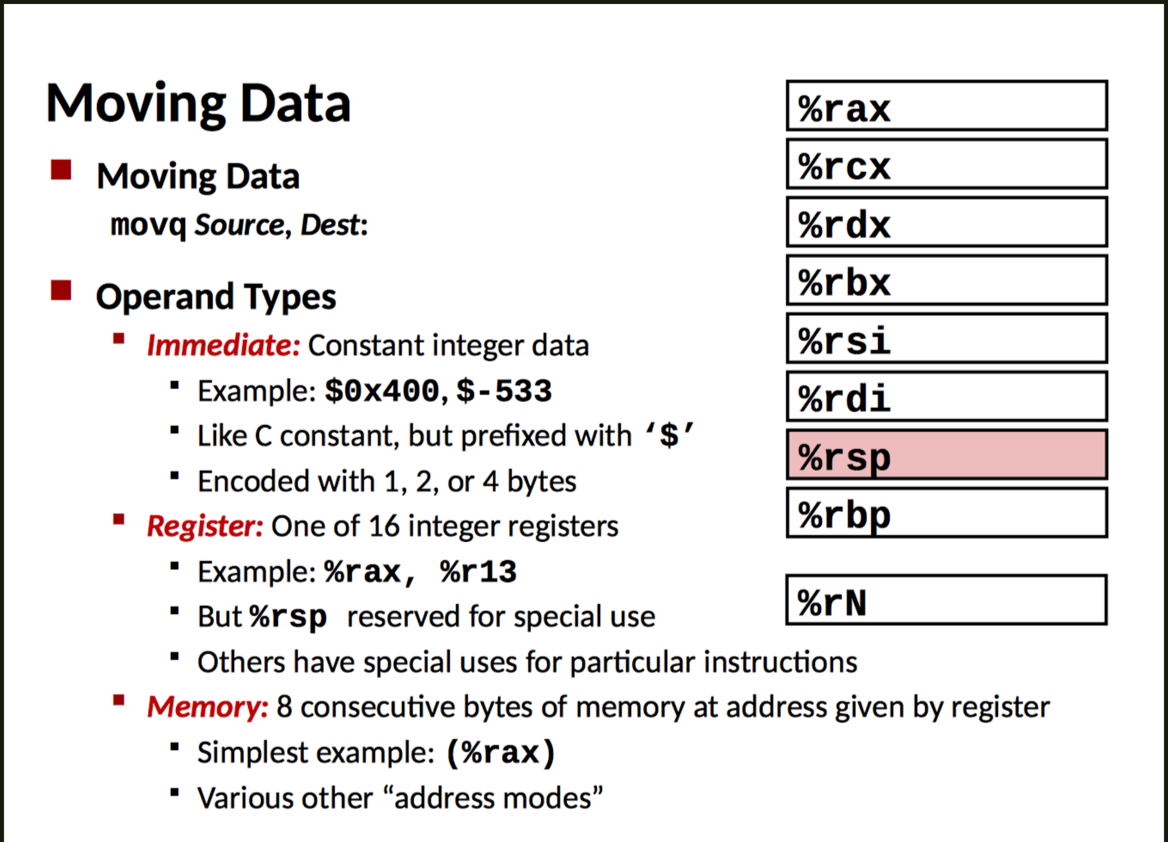
Assembly Characteristics: Data Types

- **“Integer” data of 1, 2, 4, or 8 bytes**
  - Data values
  - Addresses (untyped pointers)
- **Floating point data of 4, 8, or 10 bytes**
- **Code: Byte sequences encoding series of instructions**
- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory



- **Data movement**
  - `mov`, `movs`, `movz`, …
- **Arithmetic**
  - `add`, `sub`, `shl`, `sar`, *`lea`*, …
- **Control flow**
  - `cmp`, `test`, `j_`, `set_`, …
- **Stack/procedures**
  - `push`, `pop`, `call`, `ret`, …

X86 INSTRUCTIONS – MOVEMENT



Moving Data

- **Moving Data**

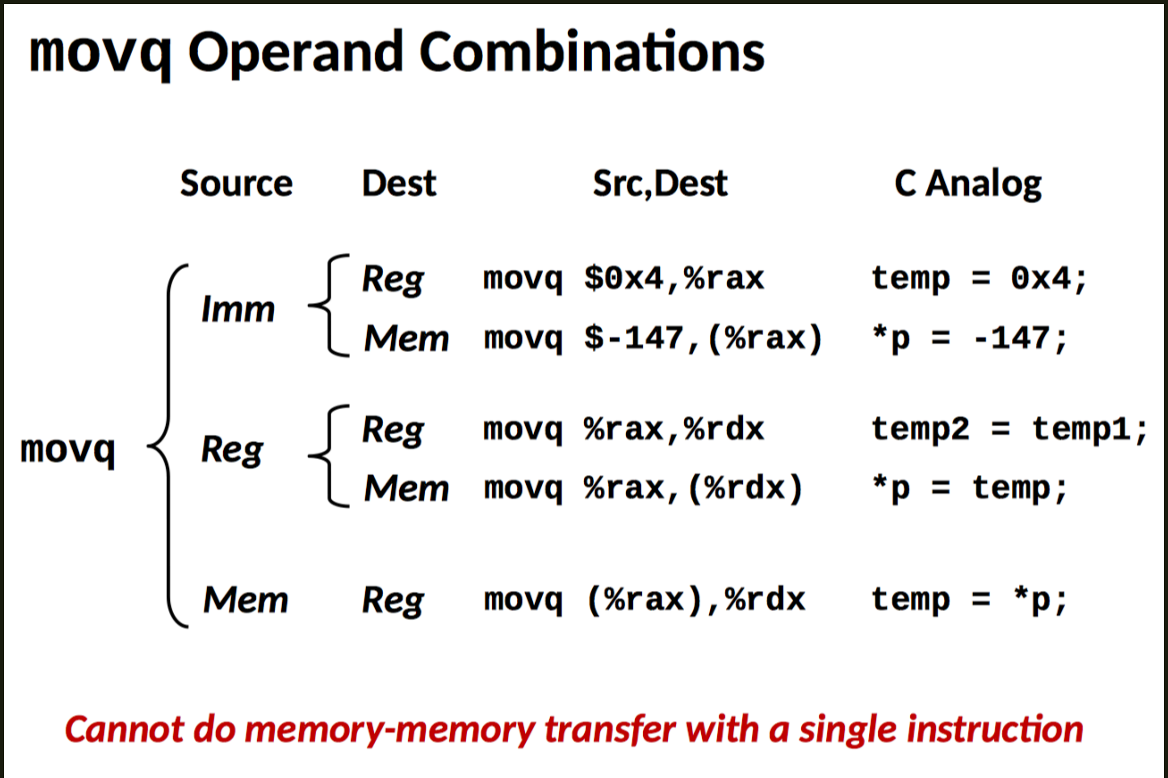
  movq ***Source*, *Dest*:**

- **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"

%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp
%rbp

%rN

X86 INSTRUCTIONS – MOVEMENT

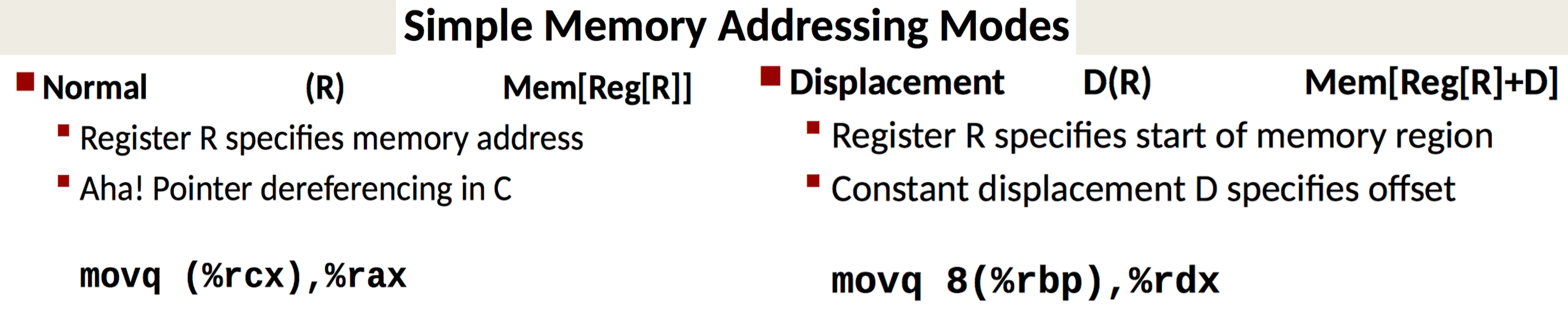


movq Operand Combinations

	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

X86 INSTRUCTIONS – MOVEMENT



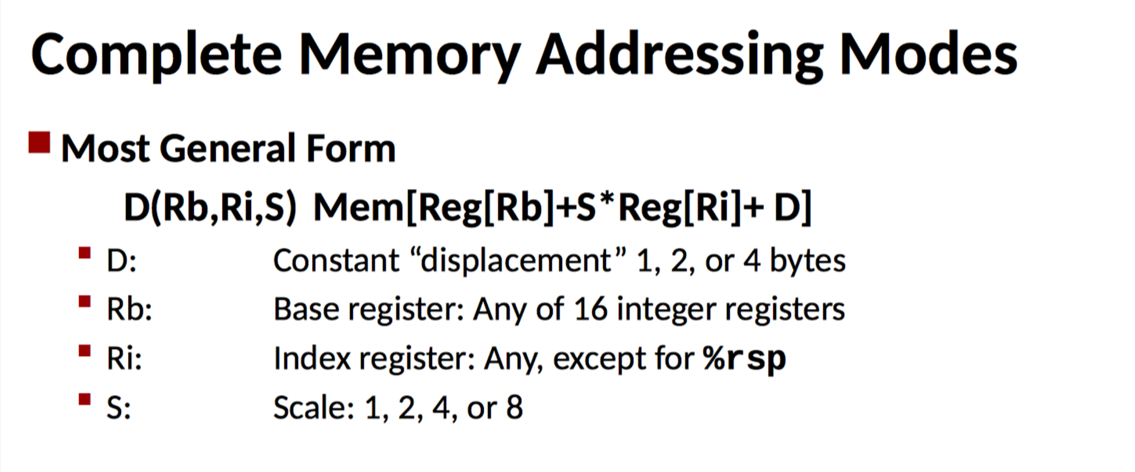
Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```



Complete Memory Addressing Modes

- **Most General Form**

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

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

X86 Instructions – Arithmetic

- **Two Operand Instructions:**

Format		Computation	
`addq`	Src,Dest	Dest = Dest + Src	
`subq`	Src,Dest	Dest = Dest ʌʌ Src	
`imulq`	Src,Dest	Dest = Dest * Src	
`salq`	Src,Dest	Dest = Dest << Src	Also called shlq
`sarq`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrq`	Src,Dest	Dest = Dest >> Src	Logical
`xorq`	Src,Dest	Dest = Dest ^ Src	
`andq`	Src,Dest	Dest = Dest & Src	
`orq`	Src,Dest	Dest = Dest | Src	

- **Watch out for argument order!**

- **One Operand Instructions**

`incq`	Dest	Dest = Dest + 1
`decq`	Dest	Dest = Dest – 1
`negq`	Dest	Dest = – Dest
`notq`	Dest	Dest = ~Dest