Machine-Level Programming: Basics

CSCI 237: Computer Organization
7th Lecture, Feb 24, 2025

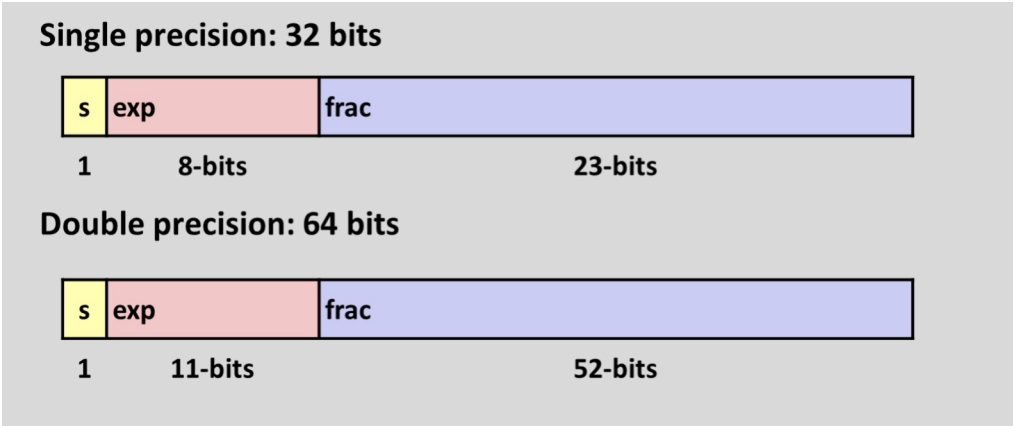
Jeannie Albrecht

Administrative Details

- All puzzles due Tue/Wed at midnight
- Come see me and/or TAs for help!
- Glow HW 2 due Friday
  - Covers signed/unsigned/floating point
- I will be in my office today from 12:45-1:30

Last Time: Floating Point

- Numerical Form: $(-1)^s * M * 2^E$
  - Sign bit s determines whether number is negative or positive
  - Significand (mantissa) M *normally* a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two
  - Normalized, denormalized, and special cases
- Encoding
  - s field encodes sign bit s (0 for positive, 1 for negative value)
  - exp field *encodes* E (but is not equal to E)
  - frac field *encodes* M (but is not equal to M)



Today: Machine Programming: Basics

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

C float Decoding Example

$v = (-1)^s \times M \times 2^E$
$E = exp - bias$

float: `0xC0A00000`

binary:

1 8-bits 23-bits

E =

S =

M =

$v = (-1)^s \times M \times 2^E$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

C float Decoding Example

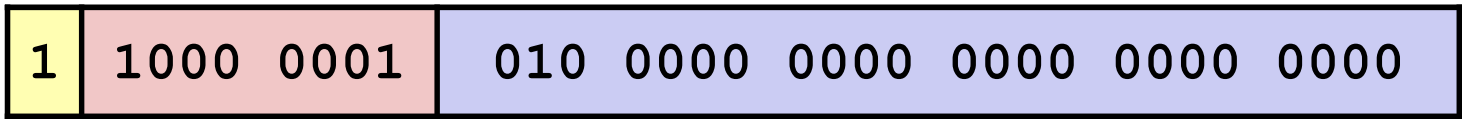
$v = (-1)^s \times M \times 2^E$

$E = exp - bias$

float: `0xC0A00000`

$Bias = 2^{k-1} - 1 = 127$

binary: 1100 0000 1010 0000 0000 0000 0000 0000



1 8-bits 23-bits

E =

S =

M =

$v = (-1)^s \times M \times 2^E$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

C float Decoding Example

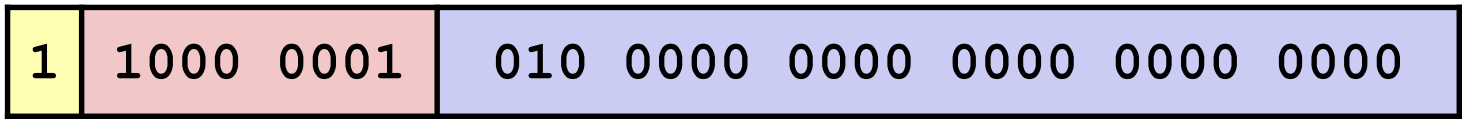
$v = (-1)^s \times M \times 2^E$

$E = exp - bias$

float: `0xC0A00000`

$Bias = 2^{k-1} - 1 = 127$

binary: 1100 0000 1010 0000 0000 0000 0000 0000



$E = exp - bias = 129 - 127 = 2$

$S = 1$ (negative number)

$M = 1.010\ 0000\ 0000\ 0000\ 0000\ 0000$

$= 1 + 1/4 = 1.25$

$v = (-1)^s \times M \times 2^E = (-1)^1 * 1.25 * 2^2 = -5$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Floating Point in C

- C guarantees two levels
  - **`float`** single precision
  - **`double`** double precision
- Conversions/Casting
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation 😩
  - **`double`**/**`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to "rounding mode"

Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

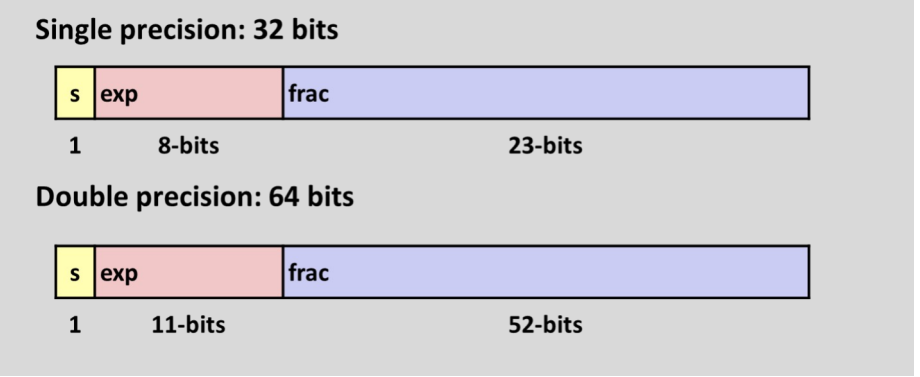
```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NaN

- `x == (int)(float) x` ✗
- `x == (int)(double) x` ✓
- `f == (float)(double) f` ✓
- `d == (double)(float) d` ✗
- `f == -(-f);` ✓
- `2/3 == 2/3.0` ✗
- `d < 0.0 ⇒ ((d*2) < 0.0)` ✓
- `d > f ⇒ -f > -d` ✓
- `d * d >= 0.0` ✓
- `(d + f)- d == f` ✗

FP Summary

- IEEE Floating Point has clear mathematical properties 😊👍
- Represents numbers of form M x 2^E
- One can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not exactly the same as real arithmetic
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers
- Next up: Chapter 3!



Today: Machine Programming: Basics

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

Definitions

- Architecture: The parts of a processor design that one needs to understand for writing assembly/machine code.
  - Examples: instruction set specification, registers
  - Example ISA (instruction set architecture) implementations:
    - Intel: x86, IA32, Itanium, **x86-64**
    - ARM: Used in almost all mobile phones
- 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 (programming language that is assembled into machine code)

Intel x86 Processors

- Dominate laptop/desktop/server market (for now)
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.
- An Aside: Apple's M1/M2 processors use a RISC design
  - Designed using ARM's instruction set (commonly found in smartphones and tablets rather than computers)

Intel x86 Evolution: Milestones

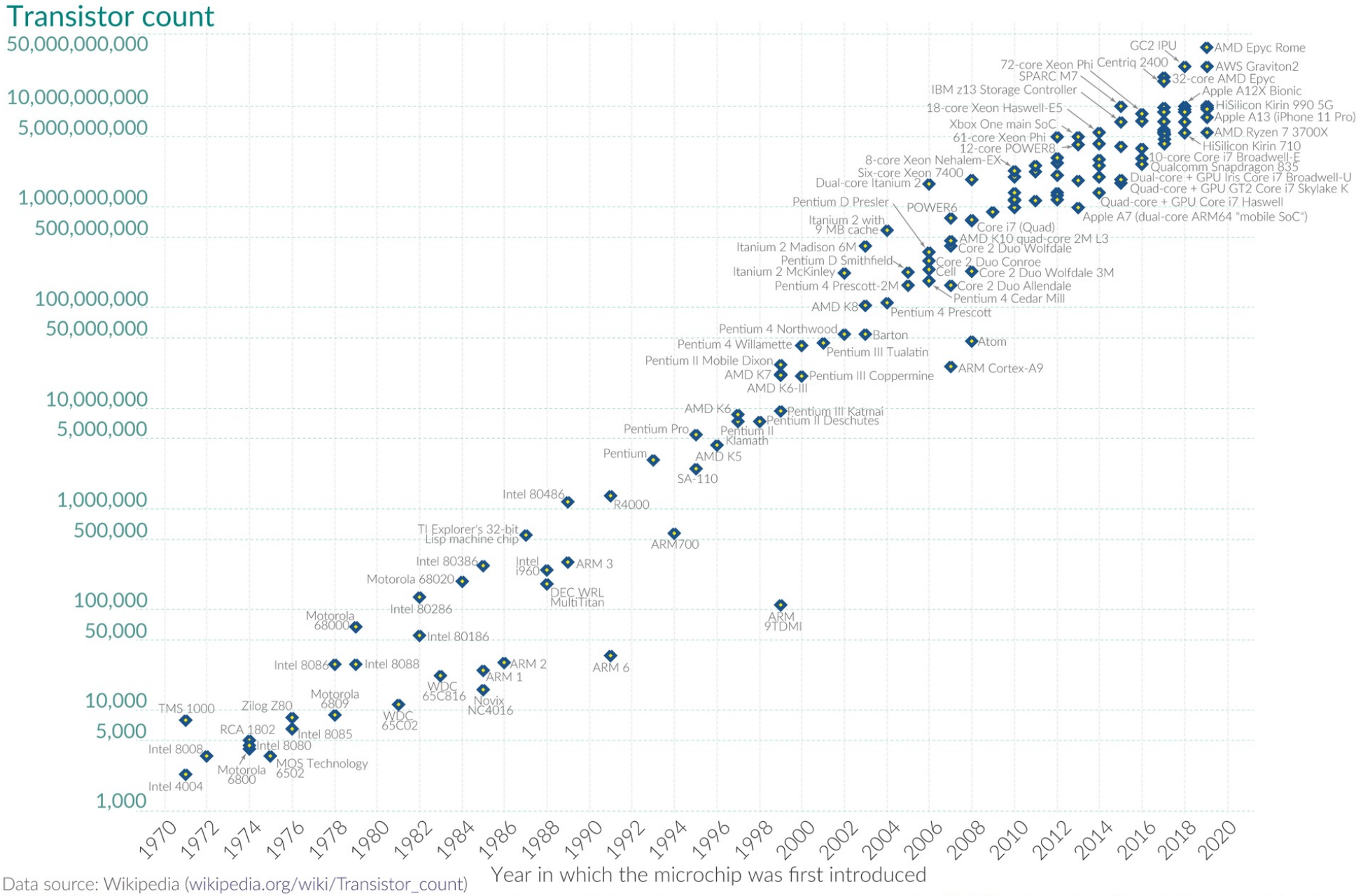
Name	*Date*	*Transistors*	*MHz*
8086	1978	29K	5-10
386	1985	275K	16-33
Pentium 4E	2004	125M	2800-3800
Core 2	2006	291M	1060-3500
Core i7	2008	731M	1700-3900

- First 16-bit Intel processor. Basis for IBM PC & DOS
- 1MB address space
- First 32 bit Intel processor , referred to as IA32
- Added “flat addressing”, capable of running Unix
- First 64-bit Intel x86 processor, referred to as x86-64
- First multi-core Intel processor (core = CPU)
- Four cores

Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

Intel x86 Processors, cont.

- Past Generations

		Process technology
1st Pentium Pro	1995	600 nm
1st Pentium III	1999	250 nm
1st Pentium 4	2000	180 nm
1st Core 2 Duo	2006	65 nm

- Recent Generations

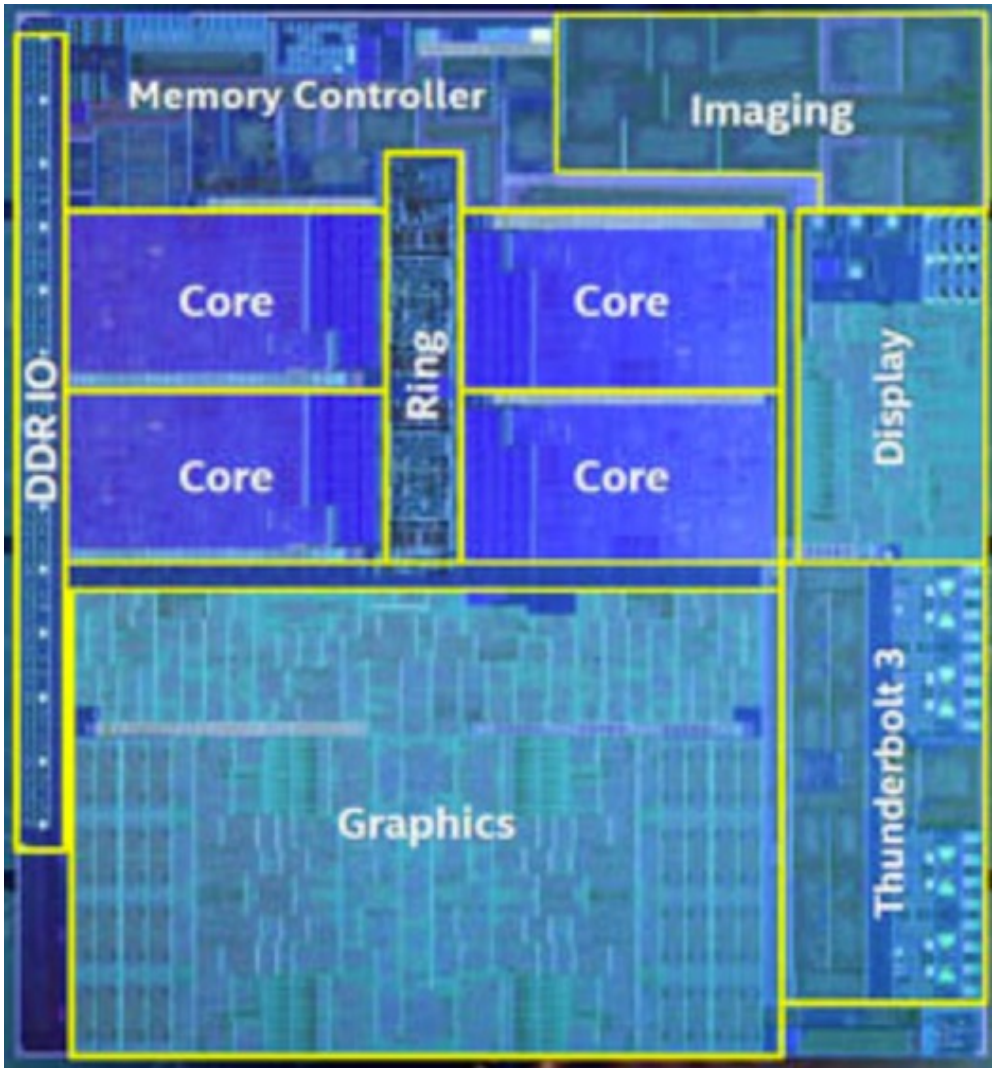
1.	Nehalem	2008	45 nm
2.	Sandy Bridge	2011	32 nm
3.	Ivy Bridge	2012	22 nm
4.	Haswell	2013	22 nm
5.	Broadwell	2014	14 nm
6.	Skylake	2015	14 nm
7.	Kaby Lake	2016/7	14 nm
8.	Coffee Lake	2017	14nm
9.	C.L. Refreshed	2018	14nm
10.	Ice Lake	2019	10nm

Process technology dimension = width of narrowest wires (10 nm ≈ 100 atoms wide)

2019 State of the Art: Ice Lake

- Mobile Device: Core i7
  - 1-2.3 GHz
    - Turbo (3.8-4.1 GHz)
  - 9-28 W
  - Integrated Intel Gen 11 GPU
  - 2-4 CPUs

- 2023 State of the Art is called Meteor Lake and uses a more complex (but similar) design involving tiles



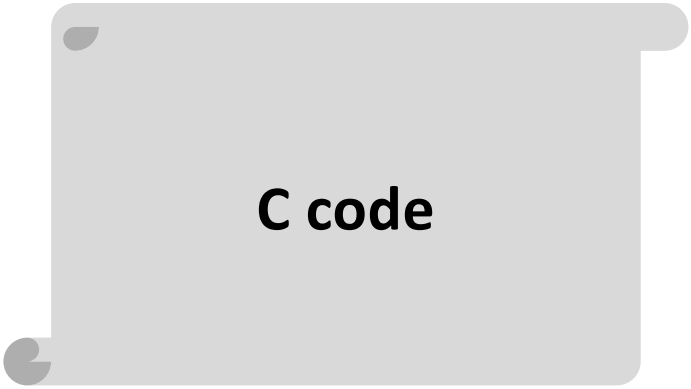
https://www.servethehome.com/intel-ice-lake-era-with-microarchitecture-and-gen11-gpu-improvements/

Our Coverage

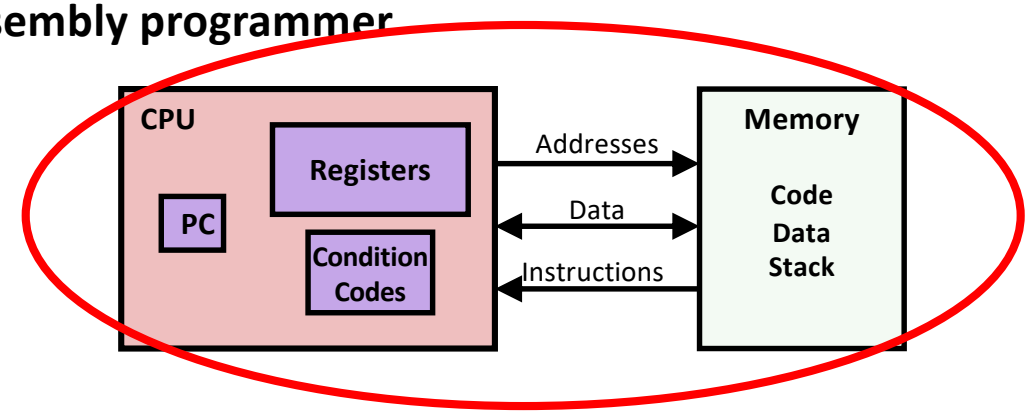
- IA32
  - The traditional x86 (32 bit)
- x86-64 (64 bit)
  - The standard
  - `> gcc hello.c`
  - `> gcc –m64 hello.c`
- Presentation
  - Book covers x86-64
  - “Web aside” on IA32
  - We will only cover x86-64

Levels of Abstraction

C programmer



Assembly programmer

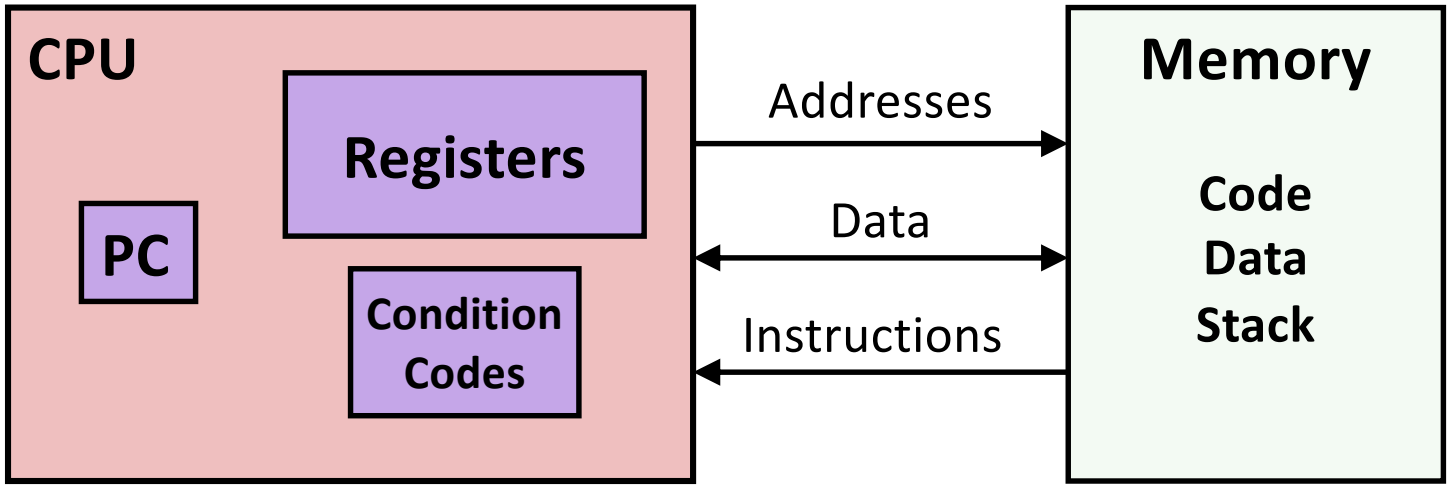


Computer Designer

Caches, clock freq, layout, …

Computer scientists love layers and abstraction…

Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

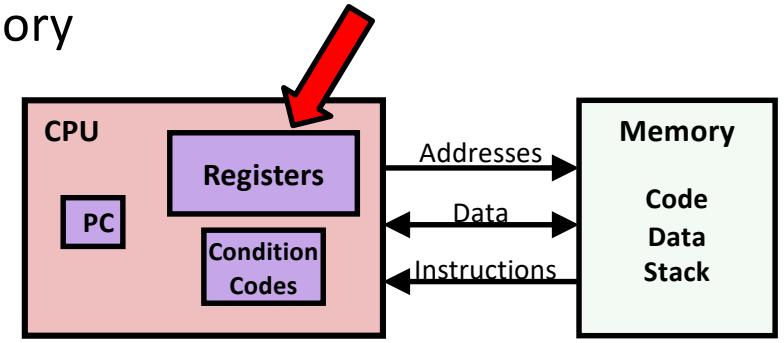
- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

Machine Programming: Basics & Ops

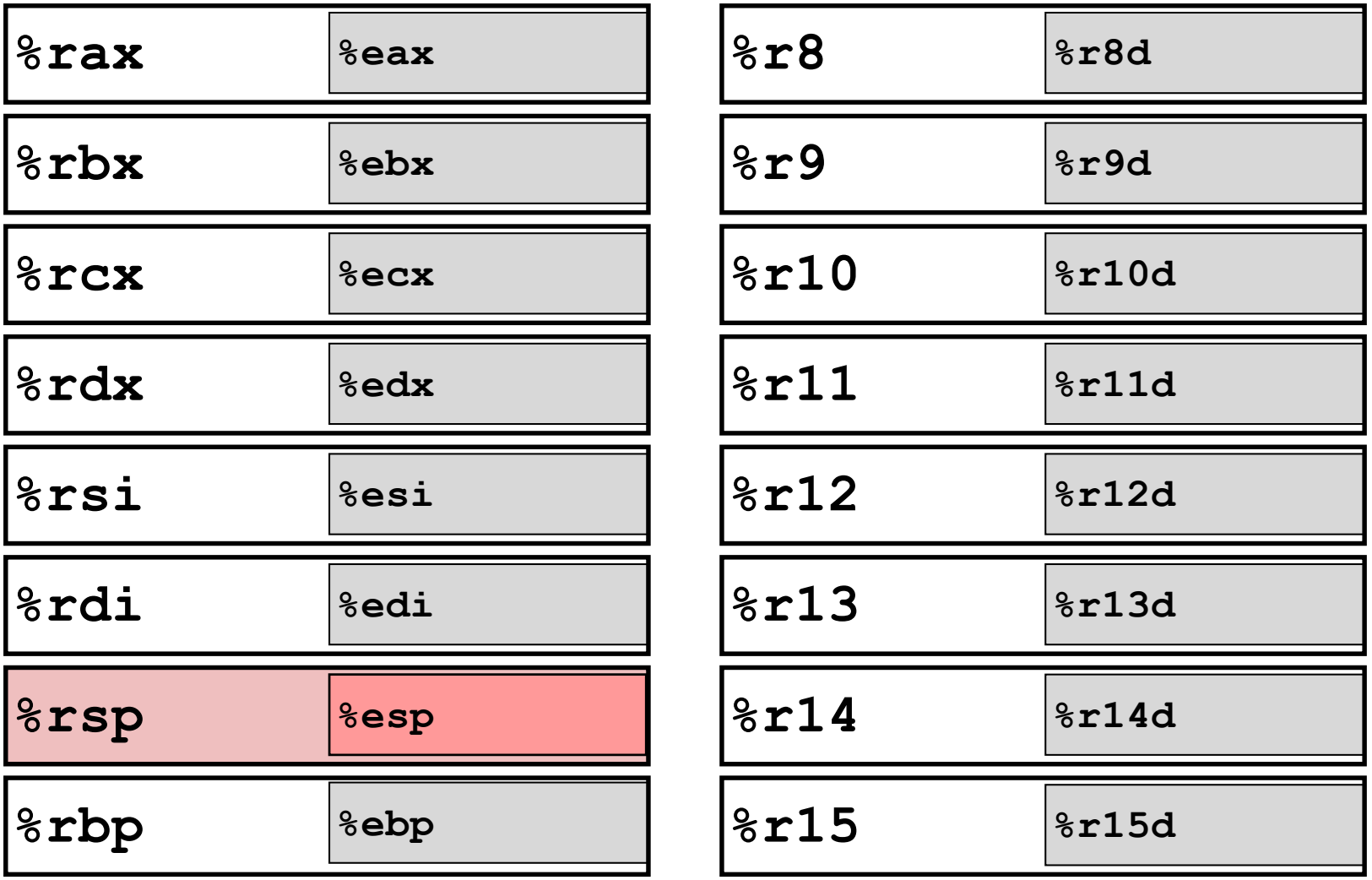
- Floating point summary
- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

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

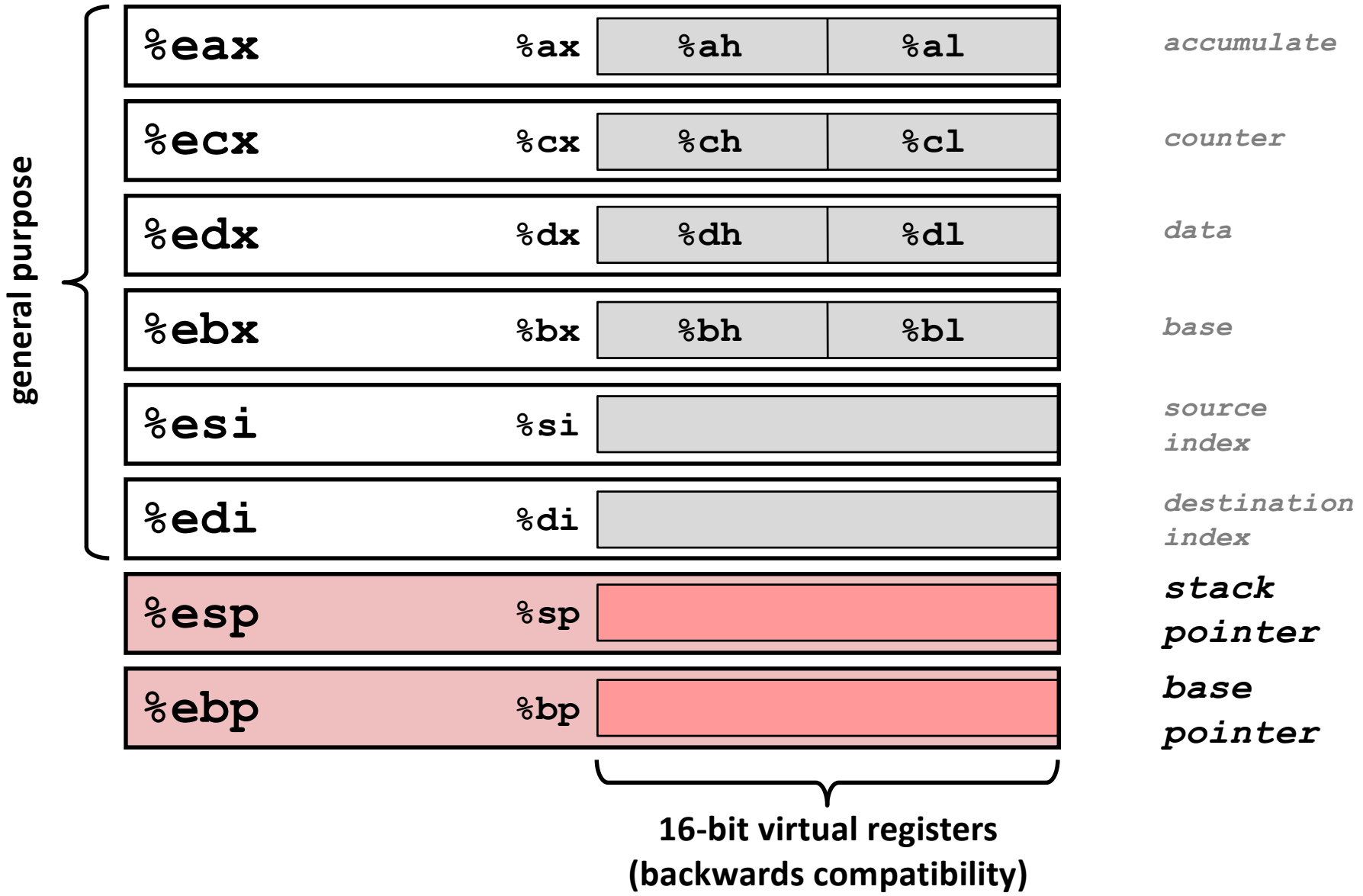


x86-64 Integer Registers



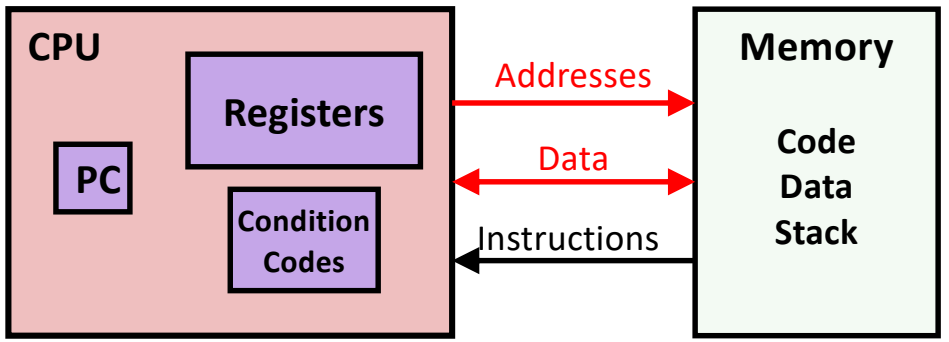
- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache); part of CPU

Some History: IA32 Registers



Assembly Characteristics: Operations

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
  - Data stored in registers is much faster to access than memory
- Perform arithmetic functions on register or memory data
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

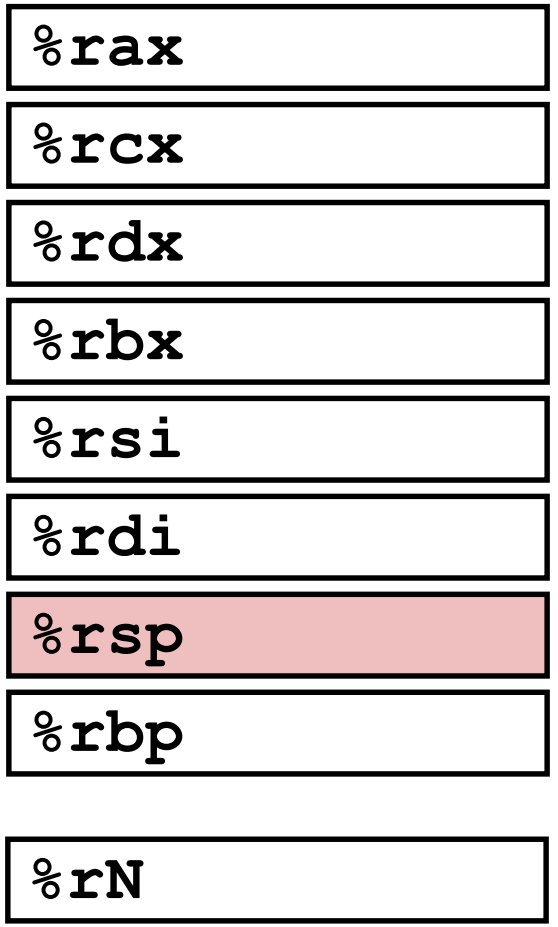


Moving Data (Ch 3.4)

- Moving Data

  `movq` ***Source, Dest***

- Operand Types for source and dest
  - ***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 "addressing modes"
    - Note the parentheses



Warning: Intel docs use mov *Dest, Source* 😩

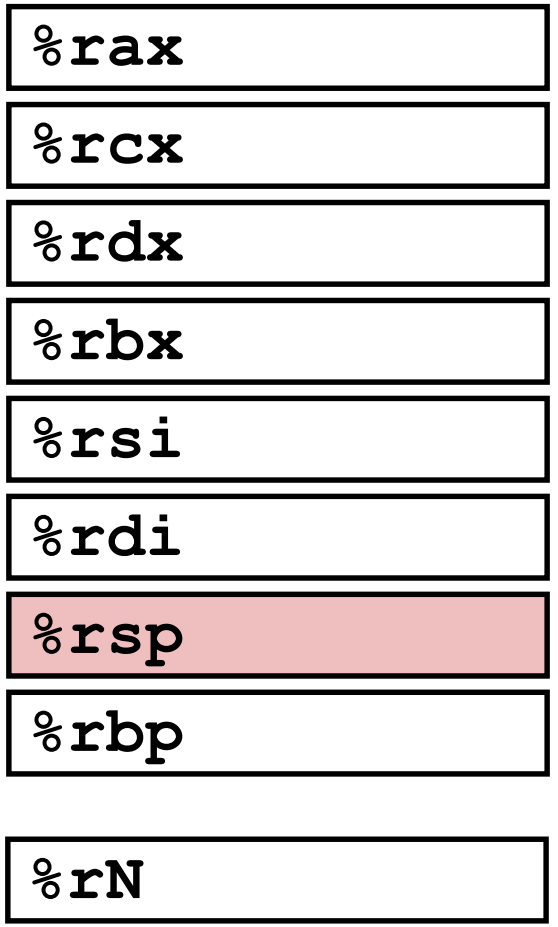
Moving Data (Ch 3.4)

- Moving Data

  `movq` ***Source, Dest***

Q = quad word (8 bytes)
L = double word (4 bytes)
W = word (2 bytes – historical!)
B = byte (1 byte)

- Operand Types for s
  - ***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 "addressing modes"
    - Note the parentheses



Warning: Intel docs use mov *Dest, Source* 😩

Pointer Recap

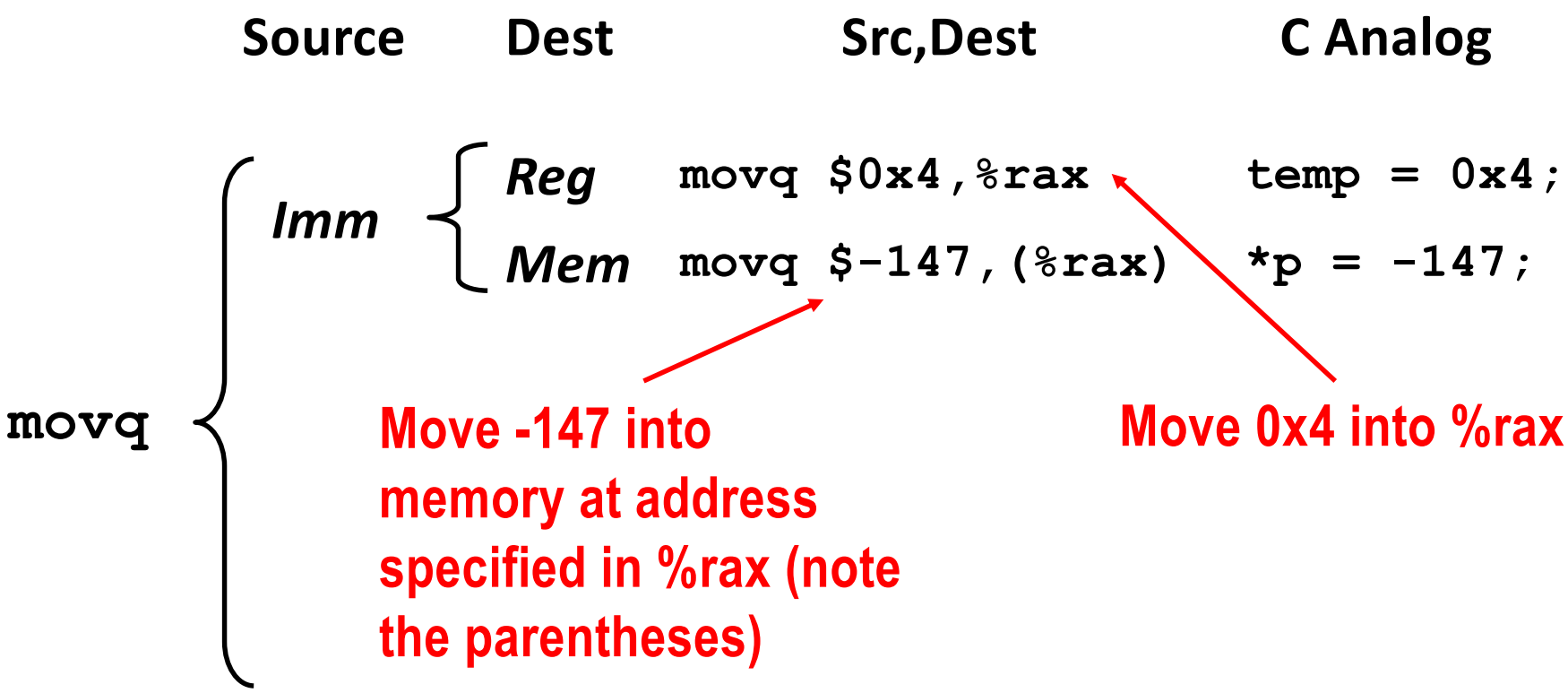
- `int *p;` `//variable p is a pointer to an integer`
- `int i;` `// integer value`

- You dereference a pointer to get value with *:
- `int i2 = *p;` `// integer i2 is assigned the value that`
  `// pointer p is pointing to`

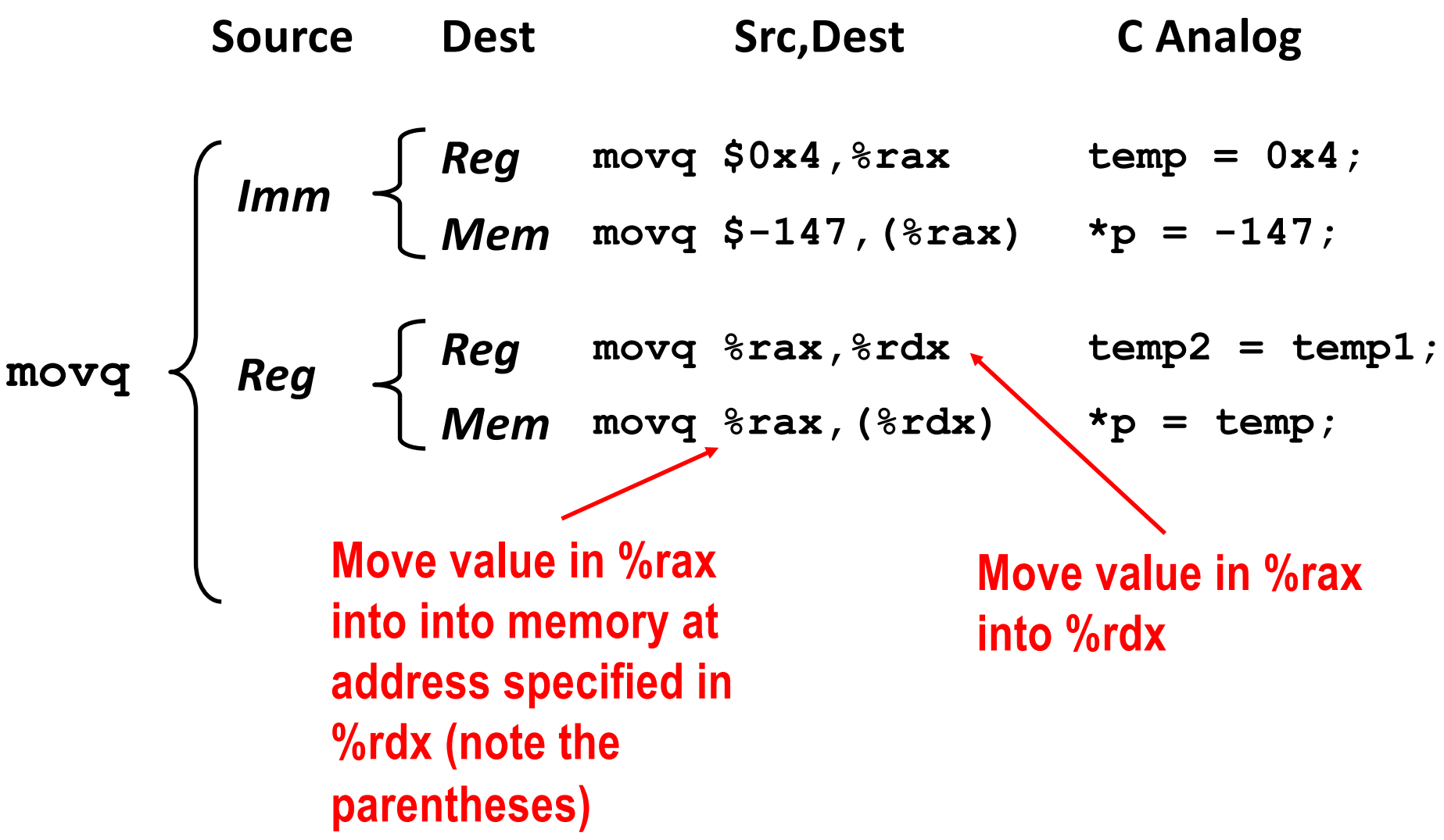
- You find address of value with &:
- `int *p2 = &i;` `// pointer p2 will point to the memory`
  `// address of i`

- Worth spending a few minutes digesting the concept of pointers!
- Check K&R for more info

movq Operand Combinations



movq Operand Combinations



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;`

Move value from memory address specified in %rax into %rdx (note the parentheses)

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;`

NOTE: Cannot do memory-memory transfer with a single instruction!

Simple Memory Addressing Modes

- Normal (Reg[R]) ⟶ Mem[Reg[R]]
  - Register R specifies memory address
  - Like pointer dereferencing in C

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

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

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