CSC 252: Computer Organization
Spring 2025: Lecture 6

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Programming Assignment 1 is due tonight
  - Details: https://www.cs.rochester.edu/courses/252/spring2023/labs/assignment1.html
  - You have 3 slip days

Announcement

- Programming assignment 2 will be released later today.
- You might still have three slip days.

Announcement

- You might still have three slip days.
- Read the instructions before getting started!!!
  - You get 1/4 point off for every wrong answer
  - Maxed out at 10
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.
- Logics and arithmetics problem set: https://cs.rochester.edu/courses/252/spring2025/handouts.html.
  - Not to be turned in.

Intel x86 ISA Evolution (Milestones)

- Evolutionary design: Added more features as time goes on

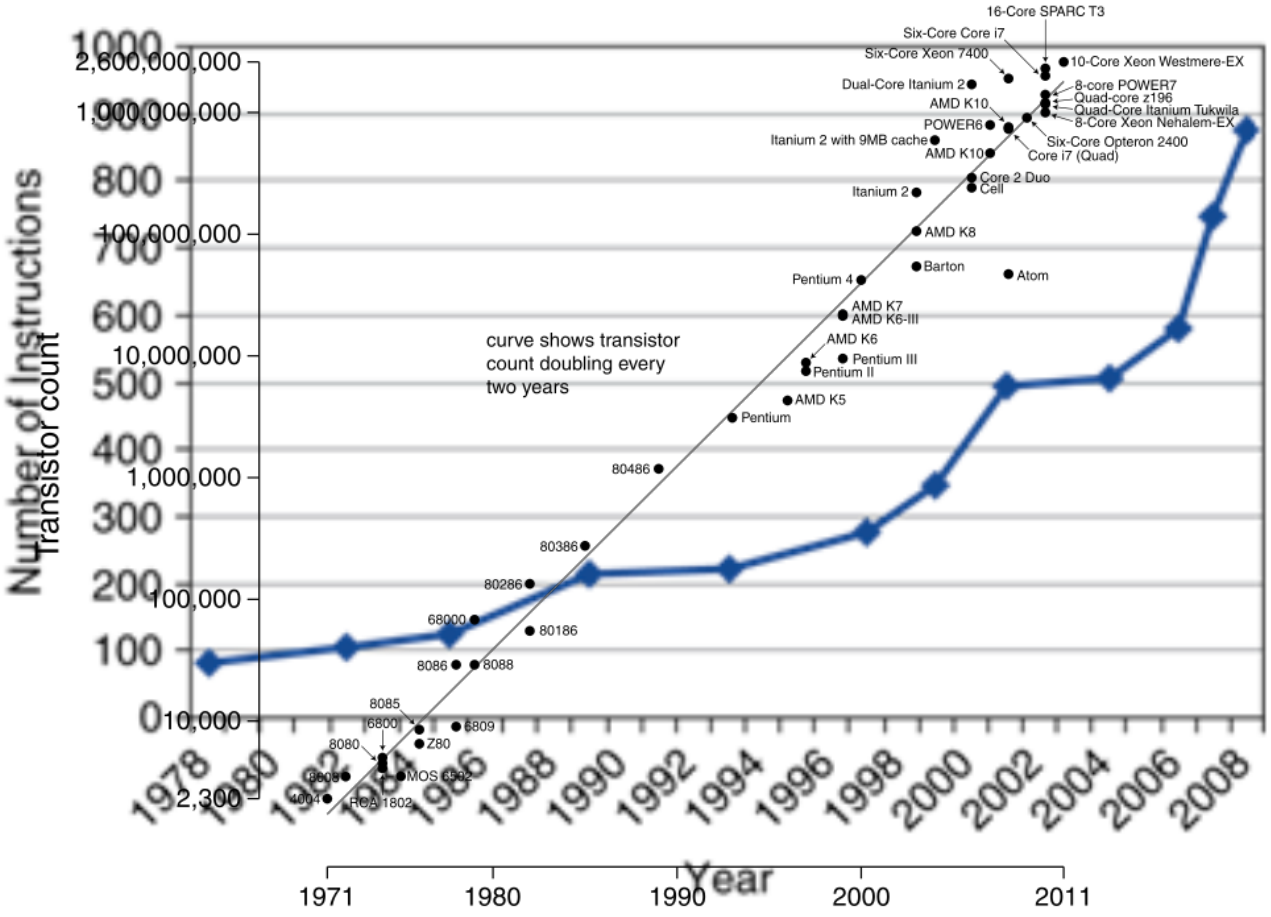
Date	Feature	Notable Implementation
1974	8-bit ISA	8080
1978	16-bit ISA (Basis for IBM PC & DOS)	8086
1980	Add Floating Point instructions	8087
1985	32-bit ISA (Refer to as IA32)	386
1997	Add Multi-Media eXtension (MMX)	Pentium/MMX
1999	Add Streaming SIMD Extension (SSE)	Pentium III
2001	Intel's first attempt at 64-bit ISA (IA64, failed)	Itanium
2004	Implement AMD's 64-bit ISA (x86-64, AMD64)	Pentium 4E
2008	Add Advanced Vector Extension (AVE)	Core i7 Sandy Bridge

Our Coverage

- IA32
  - The traditional x86
  - 2nd edition of the textbook
- x86-64
  - The standard
  - CSUG machine
  - 3rd edition of the textbook
  - Our focus

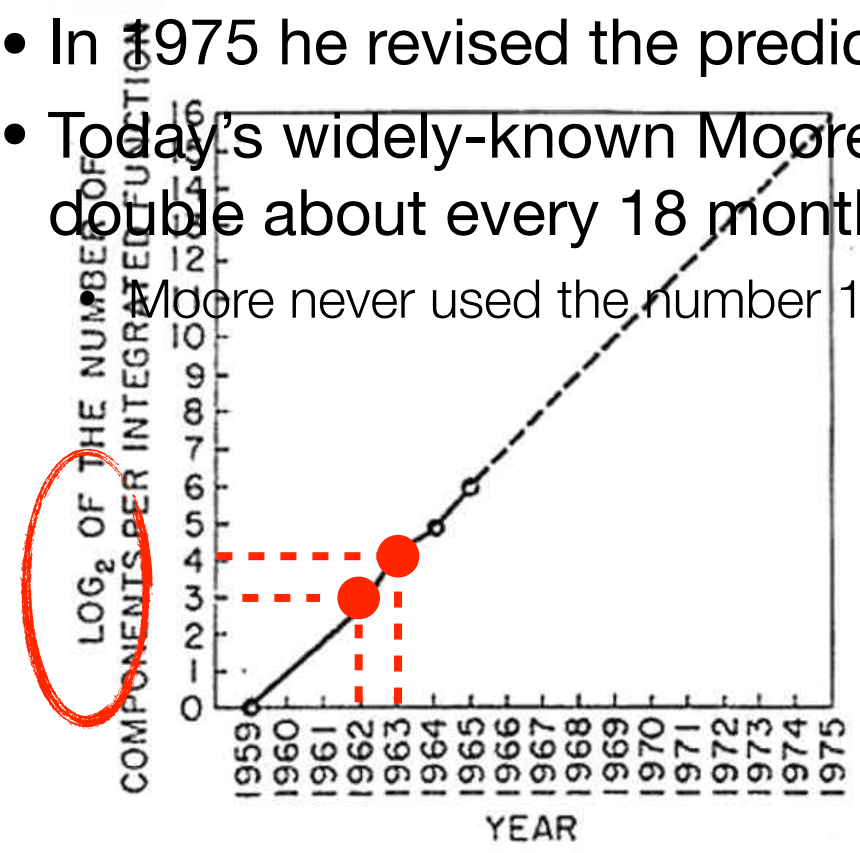
Moore's Law

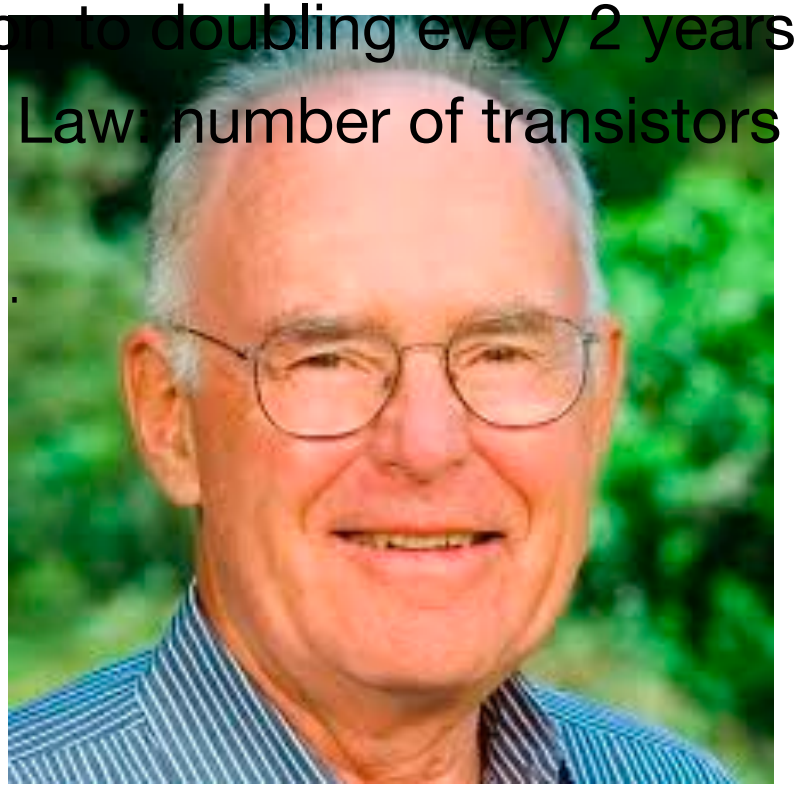
- More instructions typically require more transistors to implement



Moore's Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year
- In 1975 he revised the prediction to doubling every 2 years
- Today's widely-known Moore's Law: number of transistors double about every 18 months
  - Moore never used the number 18…





Moore's Law

ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE

TECH —

Transistors will stop shrinking in 2021, but Moore's law will live on

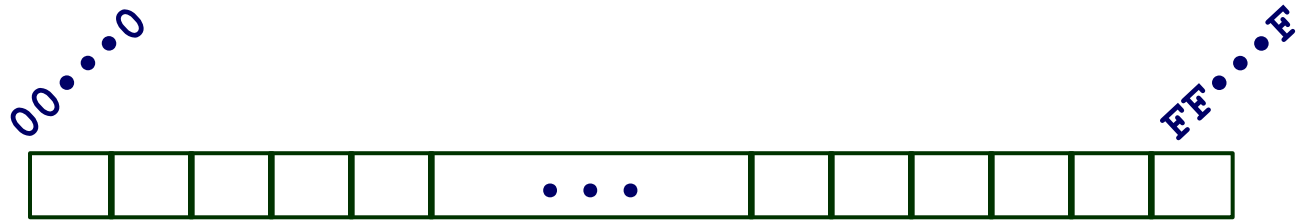
Final semiconductor industry roadmap says the future is 3D packaging and cooling.

The first problem has been known about for a long while. Basically, starting at around the 65nm node in 2006, the economic gains from moving to smaller transistors have been slowly dribbling away. Previously, moving to a smaller node meant you could cram tons more chips onto a single silicon wafer, at a reasonably small price increase. With recent nodes like 22 or 14nm, though, there are so many additional steps required that it costs a lot more to manufacture a completed wafer—not to mention additional costs for things like package-on-package (PoP) and through-silicon vias (TSV) packaging.

Today: Compute and Control Instructions

- Different ISAs and history behind them
- What's in an ISA?
- Move operations (and addressing modes)
- Arithmetic & logical operations
- Control: Conditional branches (`if`… `else`…)
- Control: Loops (`for`, `while`)
- Control: Switch Statements (`case`… `switch`…)

Byte-Oriented Memory Organization

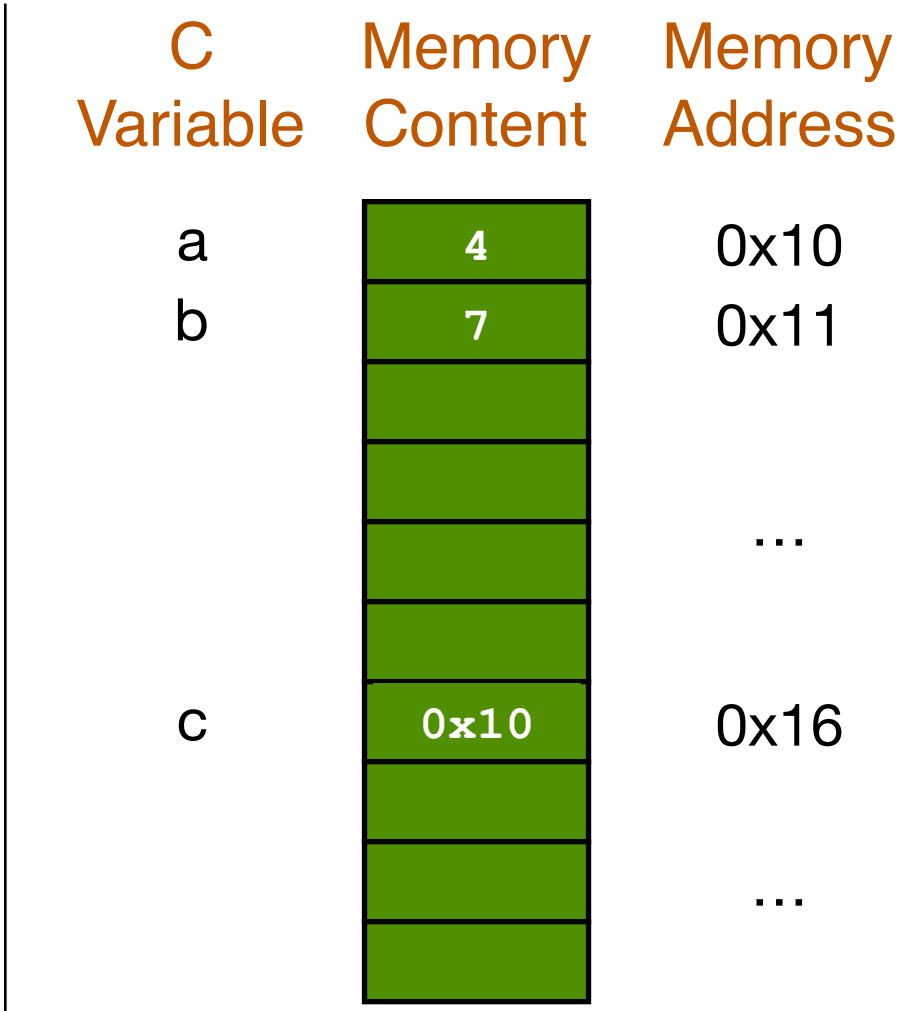


- Data in computers are stored in “memory”
  - Conceptually, envision it as a very large array of bytes: byte-addressable
- Each byte has an address
  - An address is like an index into that array
  - A pointer variable is a variable that stores an address

How Does Pointer Work in C???

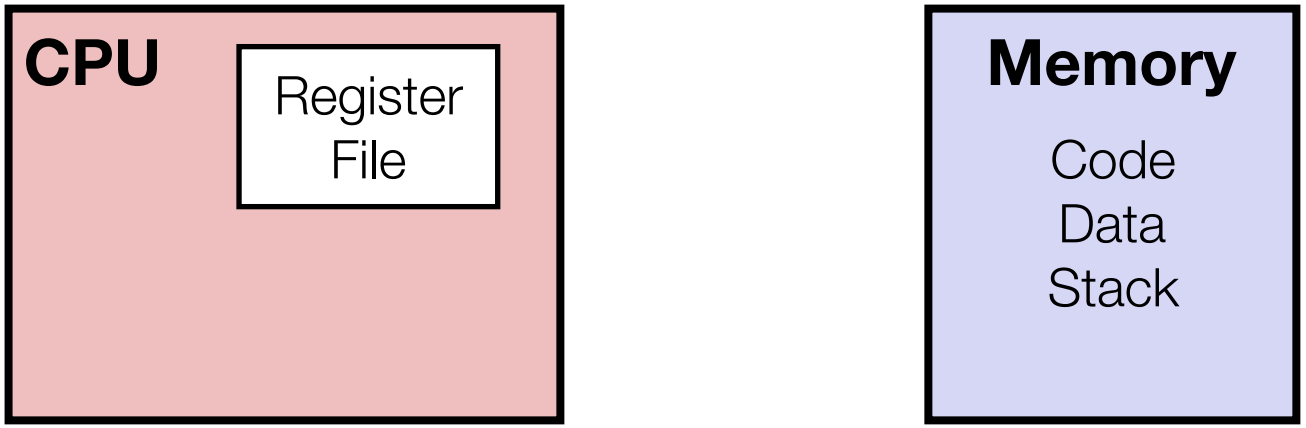
```
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

- The content of a pointer variable is memory address.
- The '**&**' operator (address-of operator) returns the memory address of a variable.
- The '*****' operator returns the content stored at the memory location pointed by the pointer variable (dereferencing)



Assembly Code's View of Computer: ISA

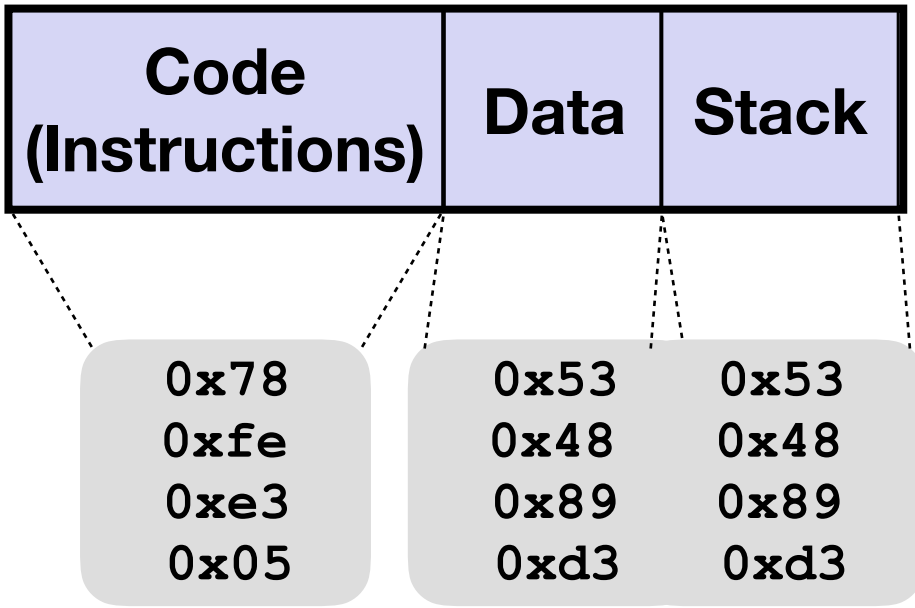
Assembly Programmer's Perspective of a Computer



- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call
- Register file
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data

Instruction is the fundamental unit of work

All instructions are encoded as bits (just like data!)

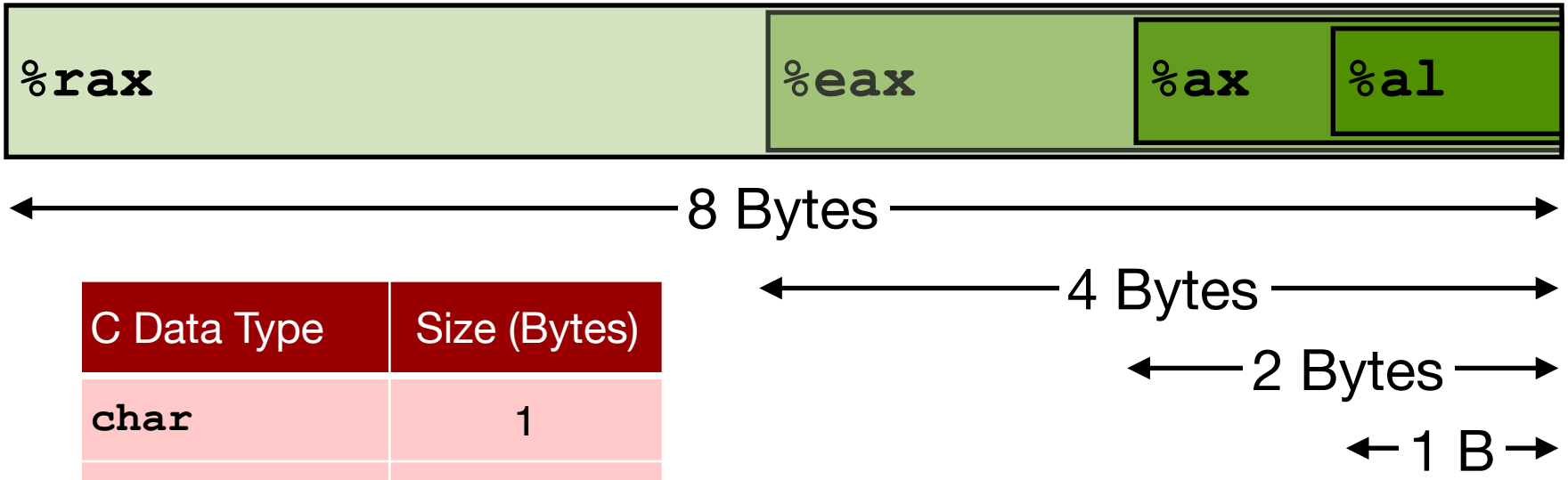


x86-64 Integer Register File

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 1 bytes)

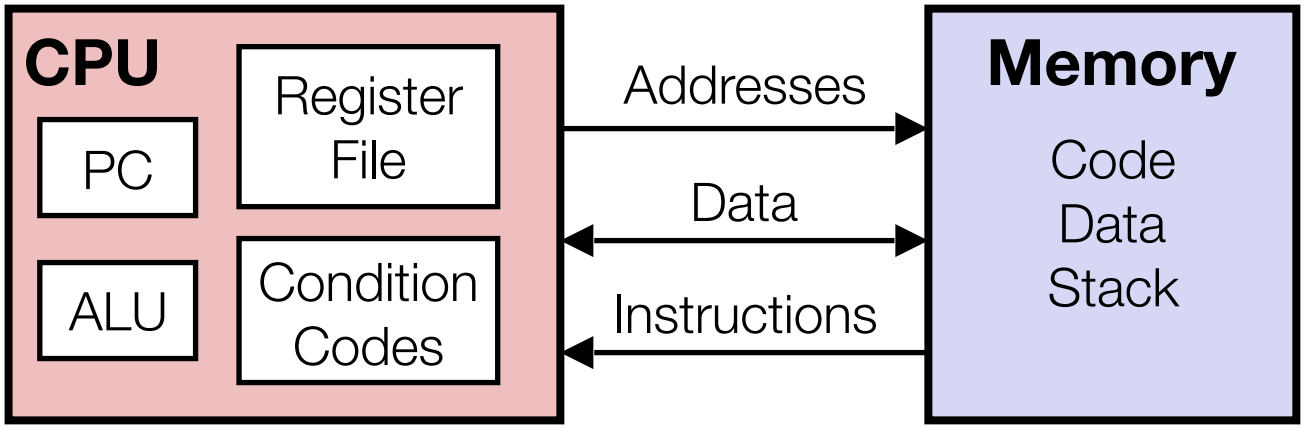


C Data Type	Size (Bytes)
`char`	1
`short`	2
`int`	4
`long`	8
Pointer	8

Floating point data is stored in a separate set of register file

Assembly Code's View of Computer: ISA

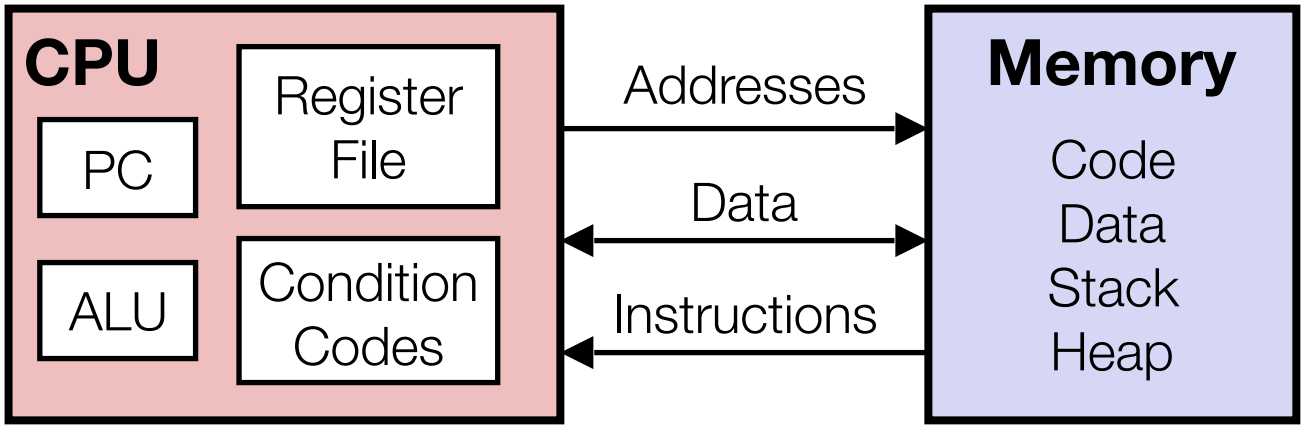
Assembly Programmer's Perspective of a Computer



- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call
- Register file
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data
- PC: Program counter
  - A special register containing address of next instruction
  - Called "RIP" in x86-64
- Arithmetic logic unit (ALU)
  - Where computation happens
- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branch

Assembly Program Instructions

Assembly Programmer's Perspective of a Computer



- *Compute Instruction*: Perform arithmetics on register or memory data
  - **`addq %eax, %ebx`**
  - C constructs: +, -, >>, etc.
- *Data Movement Instruction*: Transfer data between memory and register
  - **`movq %eax, (%ebx)`**
- *Control Instruction*: Alter the sequence of instructions (by changing PC)
  - **`jmp`**, **`call`**
  - C constructs: **`if-else`**, **`do-while`**, function call, etc.

Turning C into Object Code

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Obtain (on CSUG machine) with command

`gcc –Og –S sum.c -o sum.s`

Turning C into Object Code

Generated x86-64 Assembly

```
sumstore:
    pushq   %rbx
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

Binary Code for **`sumstore`**

Address	Memory
0x0400595	0x53
	0x48
	0x89
	0xd3
	0xe8
	0xf2
	0xff
	0xff
	0xff
	0x48
	0x89
	0x03
	0x5b
	0xc3

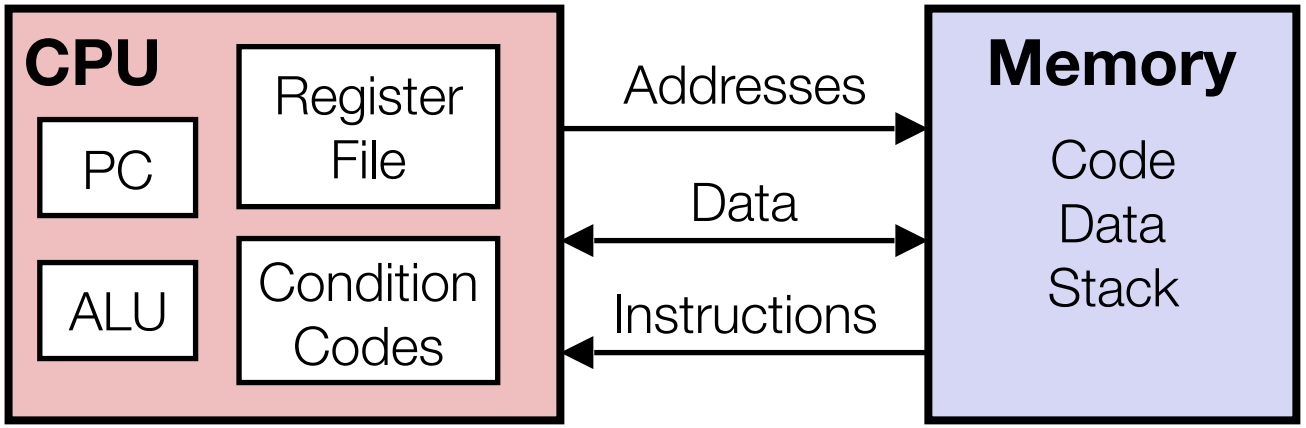
Obtain (on CSUG machine) with command

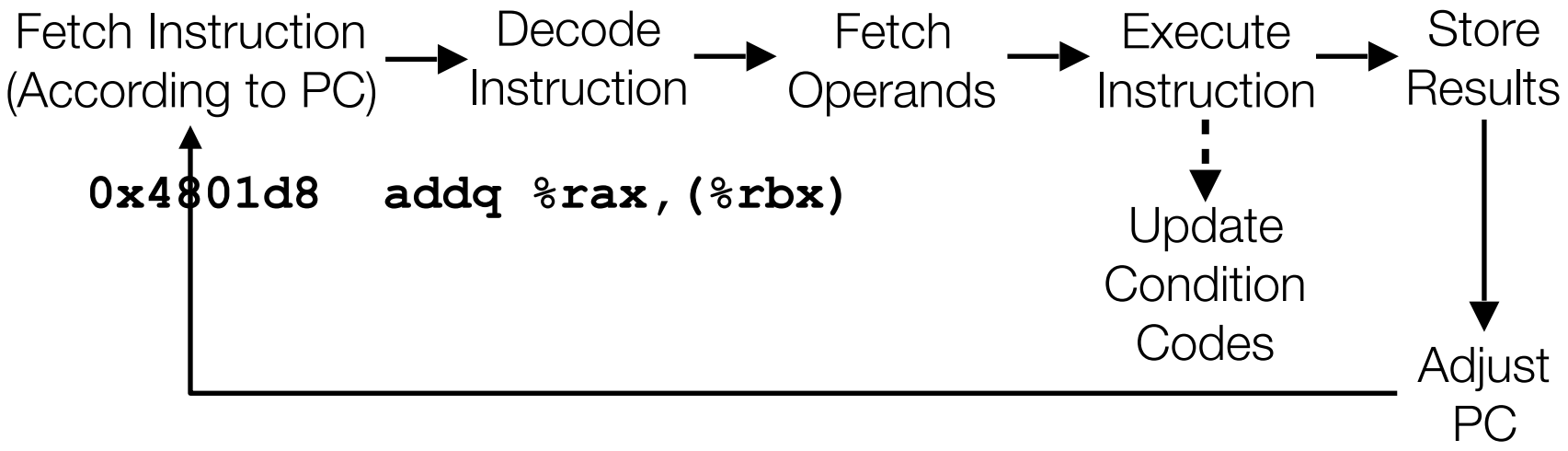
```
gcc -c sum.s -o sum.o
```

- Total of 14 bytes
- Instructions have variable lengths: e.g., 1, 3, or 5 bytes
- Code starts at memory address 0x0400595

Instruction Processing Sequence

Assembly Programmer's Perspective of a Computer



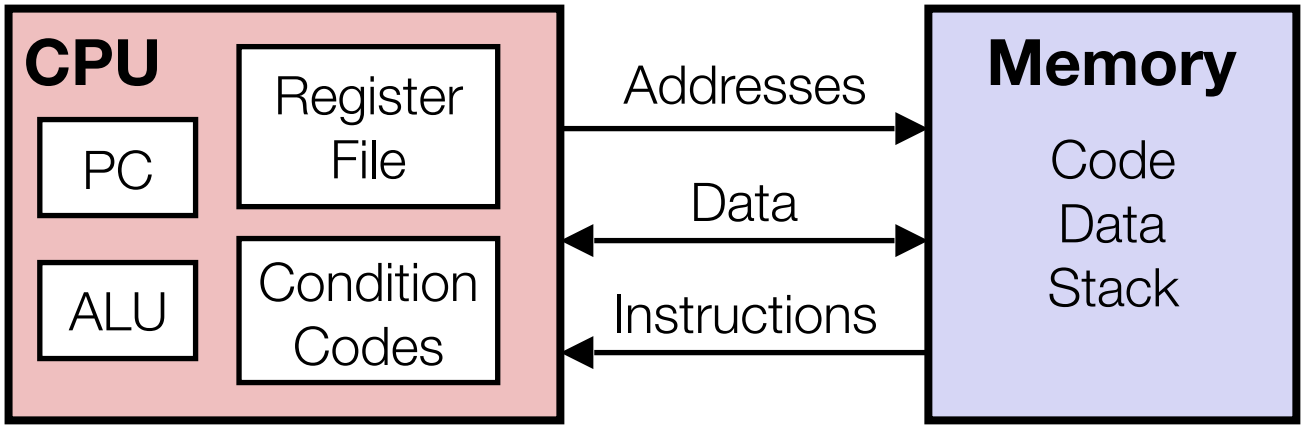


Today: Compute and Control Instructions

- Different ISAs and history behind them
- What's in an ISA?
- Move operations (and addressing modes)
- Arithmetic & logical operations
- Control: Conditional branches (`if`… `else`…)
- Control: Loops (`for`, `while`)
- Control: Switch Statements (`case`… `switch`…)

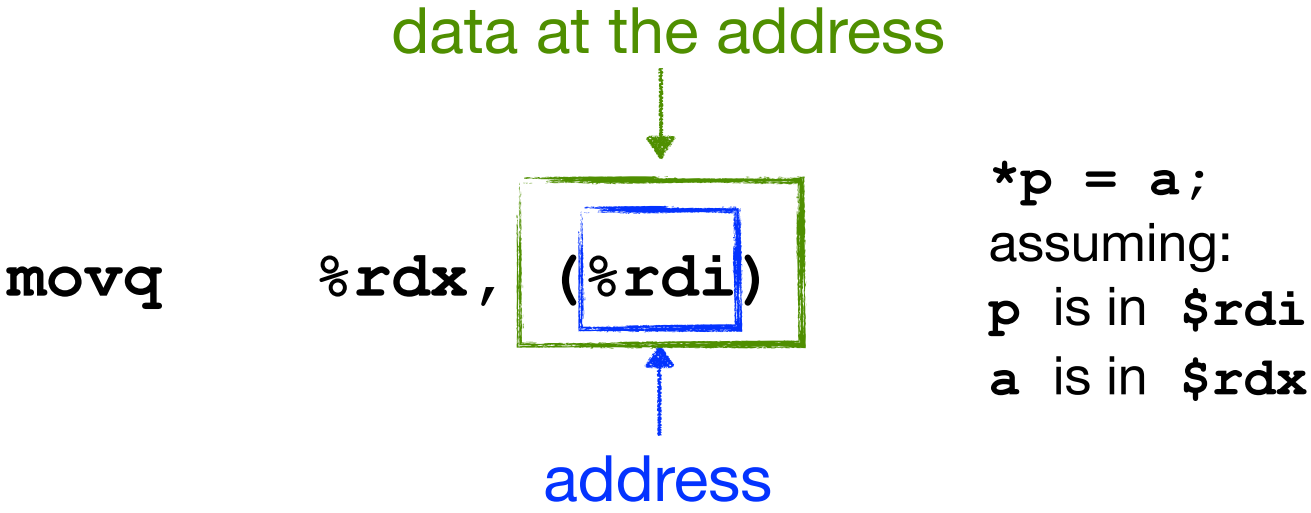
Data Movement in Processors

Assembly Programmer's Perspective of a Computer



- Initially all data is in the memory
- But memory is slow: e.g., 15 ns for each access
- Idea: move the frequently used data to a faster memory
- Register file is faster (but much smaller) memory: e.g., 0.5 ns
- There are other kinds of faster memory that we will talk about later
- Key: register file is programmer visible, i.e., you could use instructions to explicitly move data between memory and register file.

Data Movement Instruction Example



- Semantics:
  - Move (really, **copy**) data in register **`%rdx`** to memory location whose address is the value stored in **`%rdi`**
  - Pointer dereferencing

Memory Addressing Modes

- An addressing mode specifies:
  - how to calculate the effective memory address of an operand
  - by using information held in registers and/or constants
- Normal: (R)
  - Memory address: content of Register R (**Reg[R]**)
  - Pointer dereferencing in C

```
movq (%rcx),%rax; // address = %rcx
```

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

```
movq 8(%rbp),%rdx; // address = %rbp + 8
```

Data Movement Instructions

`movq` *Source*, *Dest*

Operator Operands

- Memory:
  - Simplest example: **`(%rax)`**
  - How to obtain the address is called “addressing mode”
- Register:
  - Example: **`%rax, %r13`**
  - But **`%rsp`** reserved for special use
- Immediate: Constant integer data
  - Example: **`$0x400`**, **`$-533;`** like C constant, but prefixed with ‘**`$`**’
  - Encoded with 1, 2, or 4 bytes; can only be source

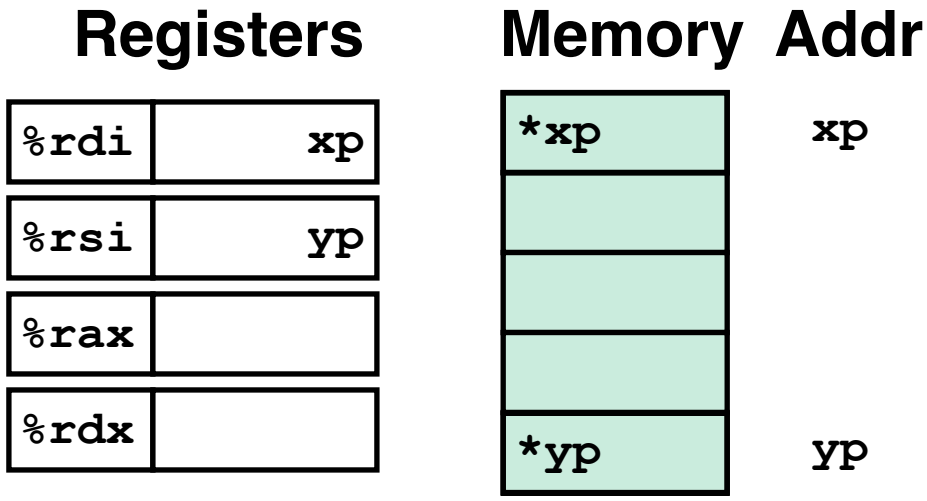
`movq` Operand Combinations

	Source	Dest	Example	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 in x86.

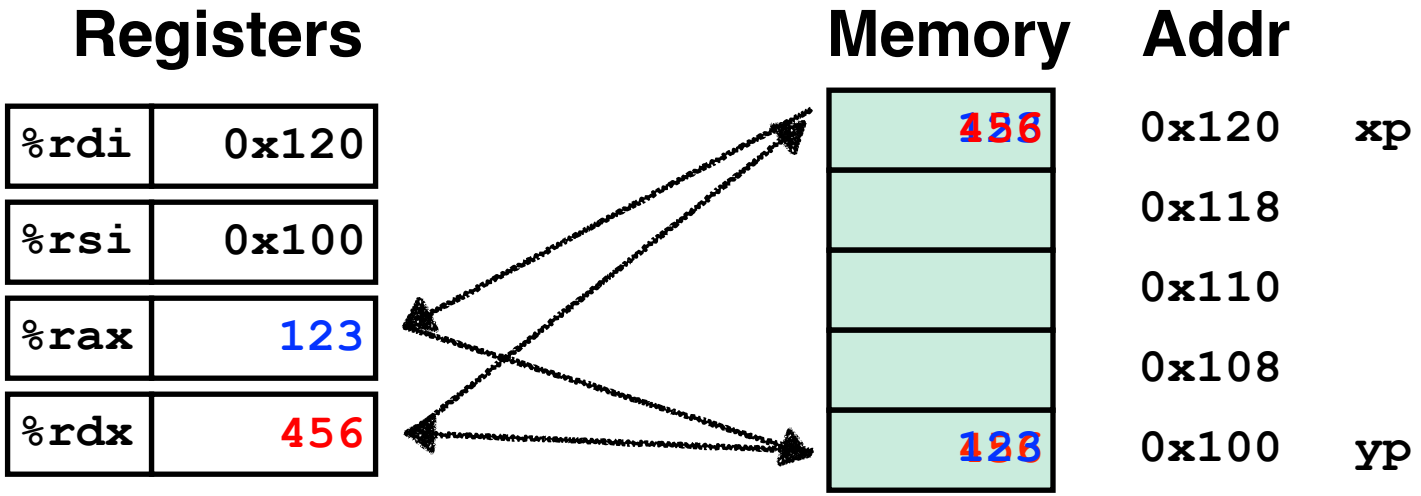
Example of Simple Addressing Modes

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




```
swap:
    movq    (%rdi), %rax  # 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
```

Complete Memory Addressing Modes

- The General Form: D(Rb,Ri,S)
  - Memory address: Reg[Rb] + S * Reg[Ri] + D
  - E.g., **`8(%eax, %ebx, 4)`**; // address = **`%eax`** + 4 * **`%ebx`** + 8
  - D: Constant "displacement"
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for %rsp
  - S: Scale: 1, 2, 4, or 8
- What is **`8(%eax, %ebx, 4)`** used for?
- Special Cases

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

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Address Computation Instruction

```
leaq 4(%rsi,%rdi,2), %rax
```

%rax = %rsi + %rdi * 2 + 4

- **`leaq`** Src, Dst
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
  - No actual memory reference is made
- Uses
  - Computing addresses without a memory reference
    - E.g., translation of p = &x[i];

Data Movement Recap

```
movq    (%rdi), %rdx
```

- Semantics:
  - Move (really, **copy**) data store in memory location whose address is the value stored in **`%rdi`** to register **`%rdx`**

```
movq    %rdx, (%rdi)
movq    8(%rdi), %rdx
addq    8(%rdi), %rdx
```

Accessing memory and doing computation in one instruction. Allowed in x86, but not all ISAs allow that (e.g., MIPS).

inefficient/inelegant.