

CSC 252: Computer Organization

Spring 2023: 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 is out. It's in x86 assembly language. Details at: <https://www.cs.rochester.edu/courses/252/spring2023/labs/assignment2.html>.

22	23	24	25	26	27 Today	28
29	30	31	Feb 1	2	3	4
5	6	7	8	9	10	11
12	13 Due	14	15	16	17	18

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://www.cs.rochester.edu/courses/252/spring2023/handouts.html>.
 - Not to be turned in.

Intel x86 ISA Evolution (Milestones)

- Evolutionary design: Added more features as time goes on

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

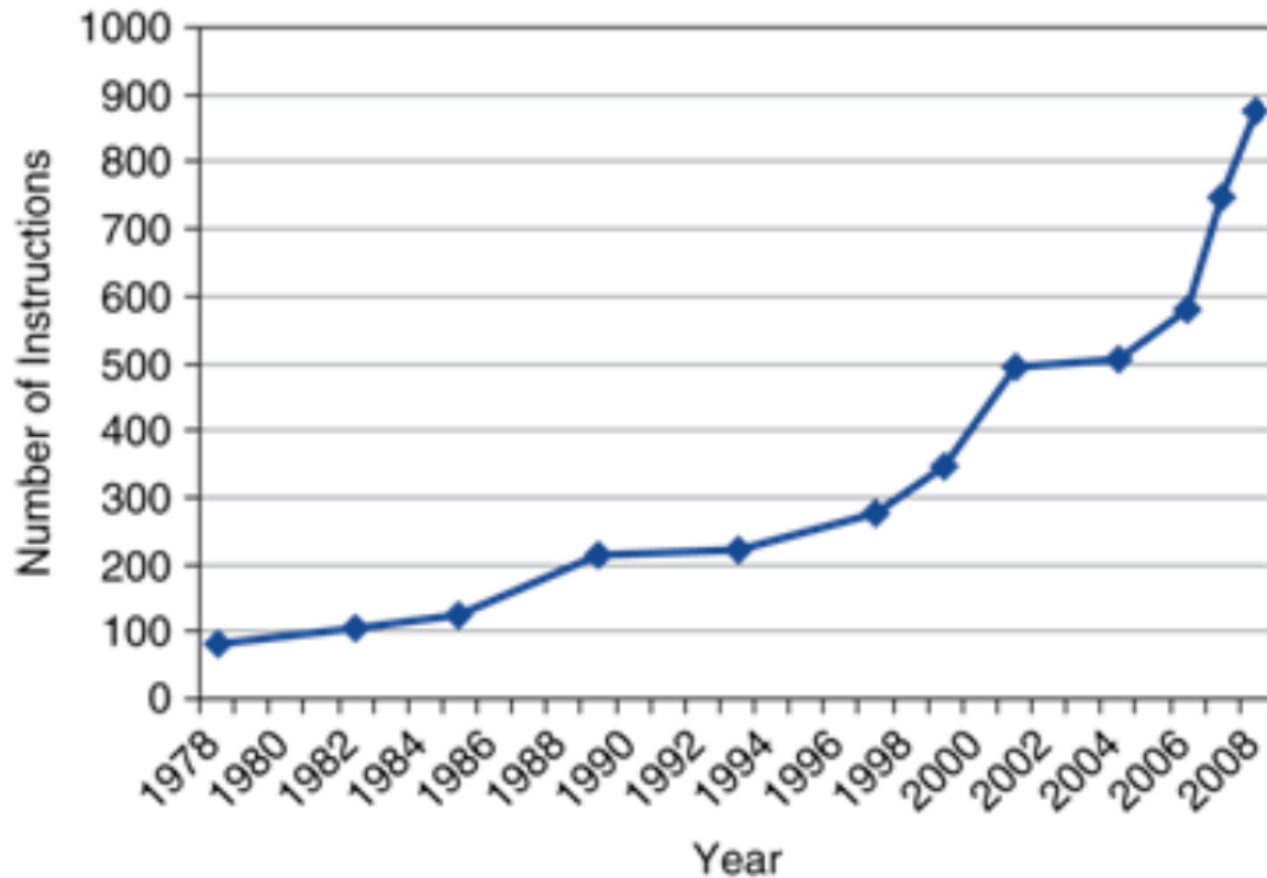
- IA32
 - The traditional x86
 - 2nd edition of the textbook
- x86-64
 - The standard
 - CSUG machine
 - 3rd edition of the textbook
 - Our focus

Moore's Law

- More instructions typically require more transistors to implement

Moore's Law

- More instructions typically require more transistors to implement



Moore's Law

- More instructions require more transistors to implement



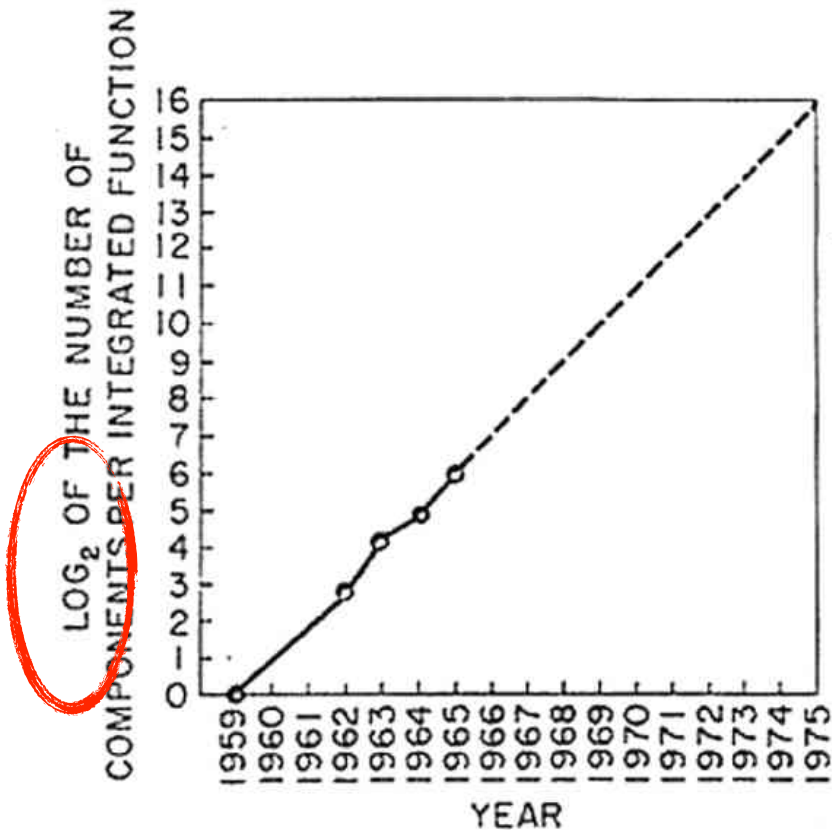
Moore's Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year



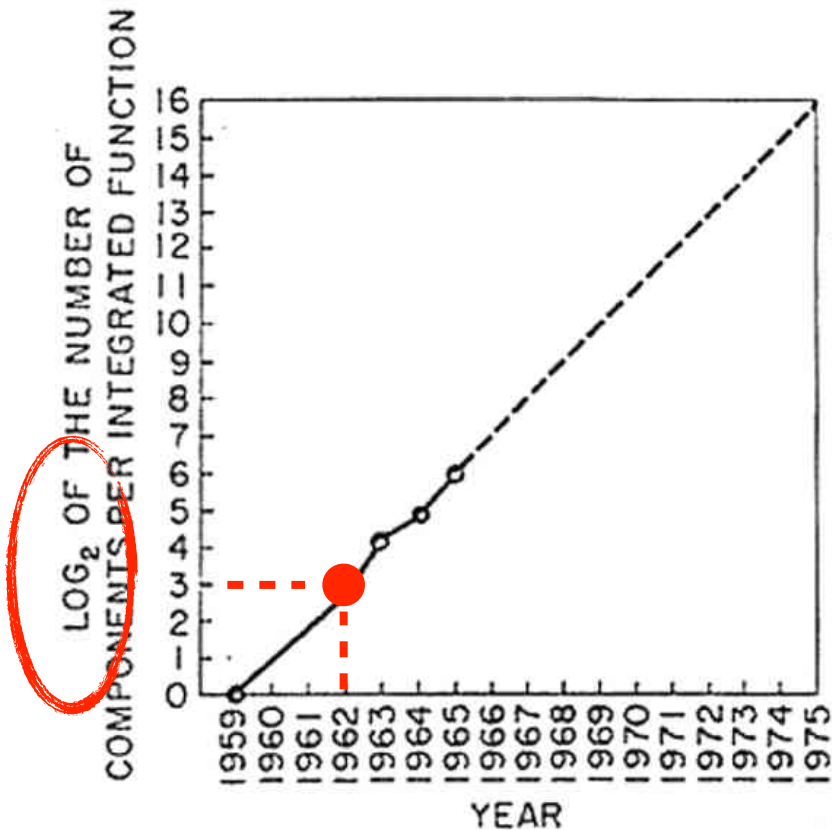
Moore's Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year



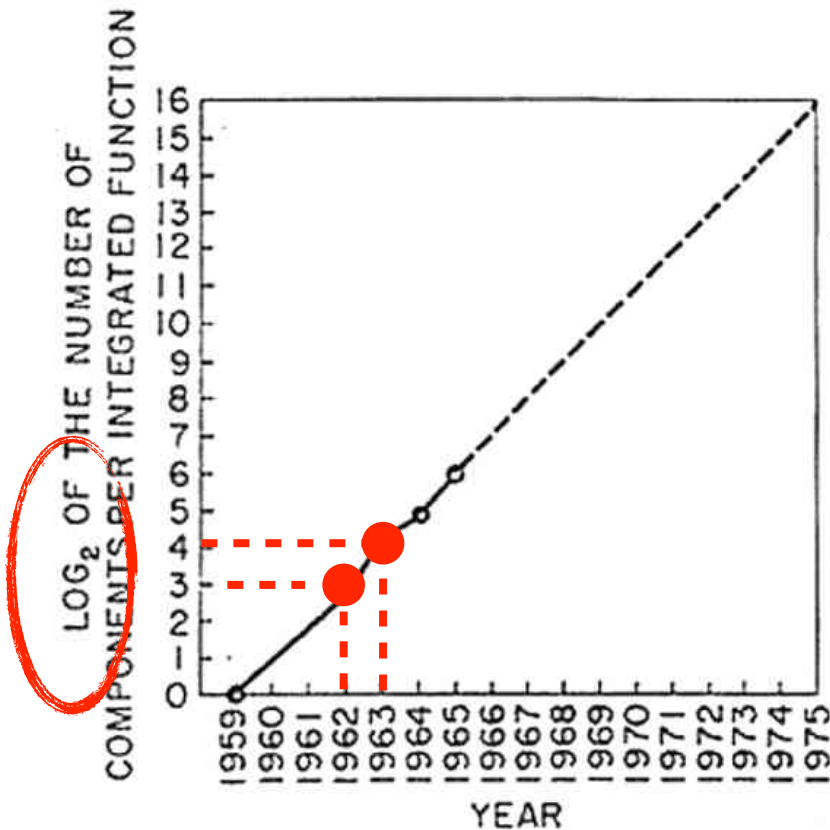
Moore's Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year



Moore's Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year



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

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

- Question: why is transistor count increasing but computers are becoming smaller?

Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller

Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller
 - $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)

Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller
 - $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)
- Moore's Law is:

Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller
 - $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)
- Moore's Law is:
 - A law of physics?

Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller
 - $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)
- Moore's Law is:
 - A law of physics? **No**

Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller
 - $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)
- Moore's Law is:
 - A law of physics? **No**
 - A law of math?

Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller
 - $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)
- Moore's Law is:
 - A law of physics? **No**
 - A law of math? **No**

Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller
 - $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)
- Moore's Law is:
 - A law of physics? **No**
 - A law of math? **No**
 - A law of economy?

Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller
 - $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)
- Moore's Law is:
 - A law of physics? **No**
 - A law of math? **No**
 - A law of economy? **Yes**

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.

Moore's Law

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.

Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller
 - $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)
- Moore's Law is:
 - A law of physics? **No**
 - A law of math? **No**
 - A law of economy? **Yes**
 - A law of psychology?

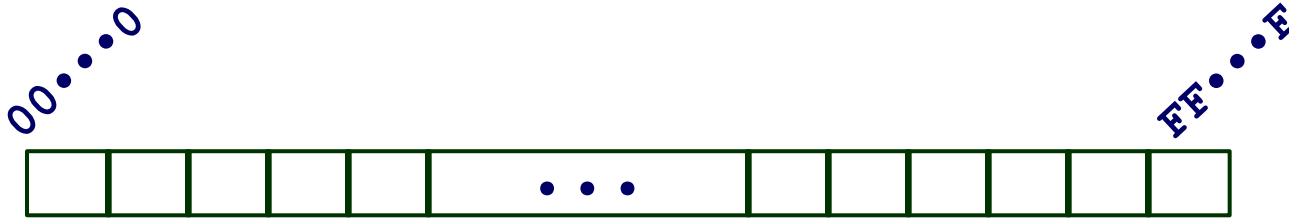
Moore's Law

- Question: why is transistor count increasing but computers are becoming smaller?
 - Because transistors are becoming smaller
 - $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)
- Moore's Law is:
 - A law of physics? **No**
 - A law of math? **No**
 - A law of economy? **Yes**
 - A law of psychology? **Yes**

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;
```

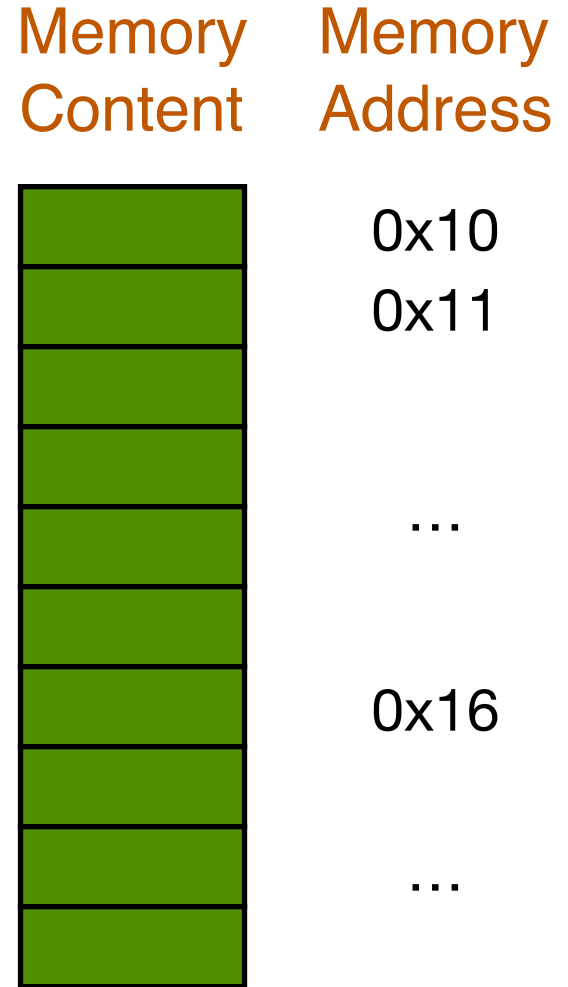
```
char* c;
```

```
c = &a;
```

```
b += (*c);
```

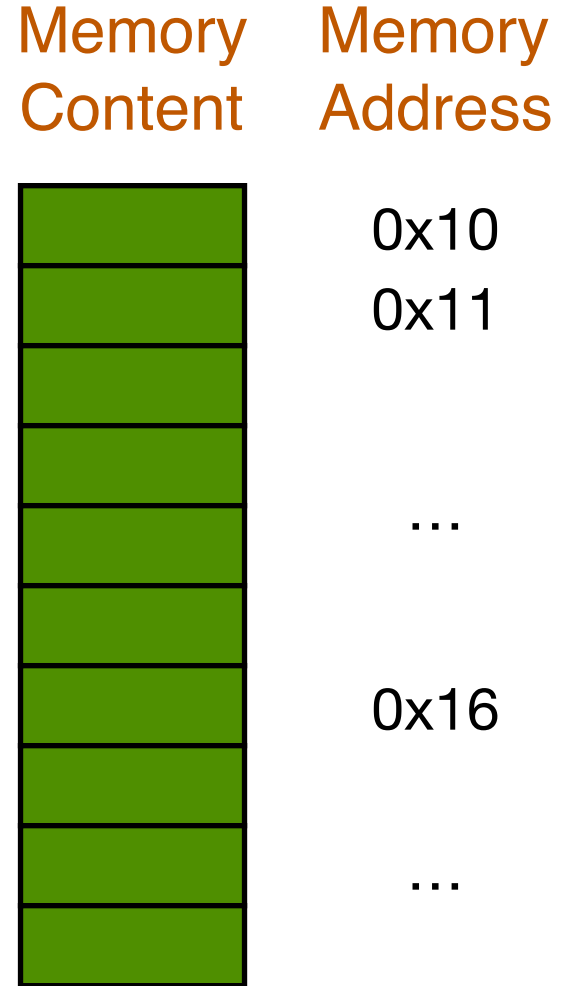
How Does Pointer Work in C???

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



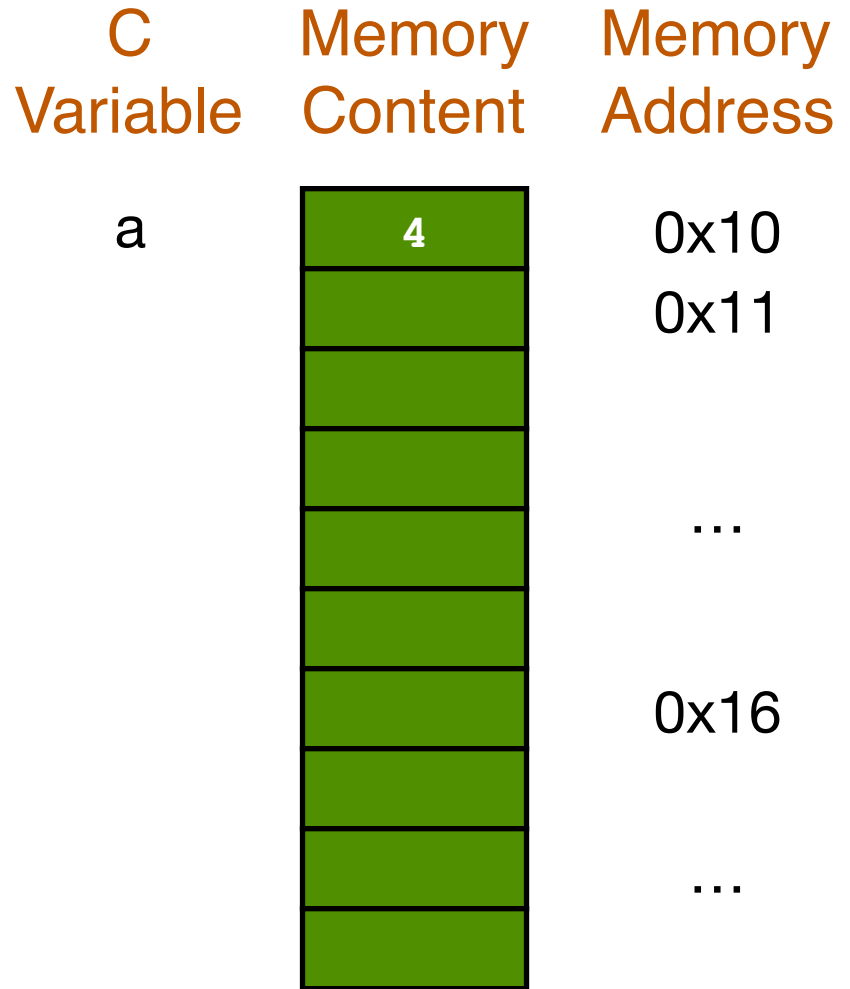
How Does Pointer Work in C???

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



How Does Pointer Work in C???

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



How Does Pointer Work in C???

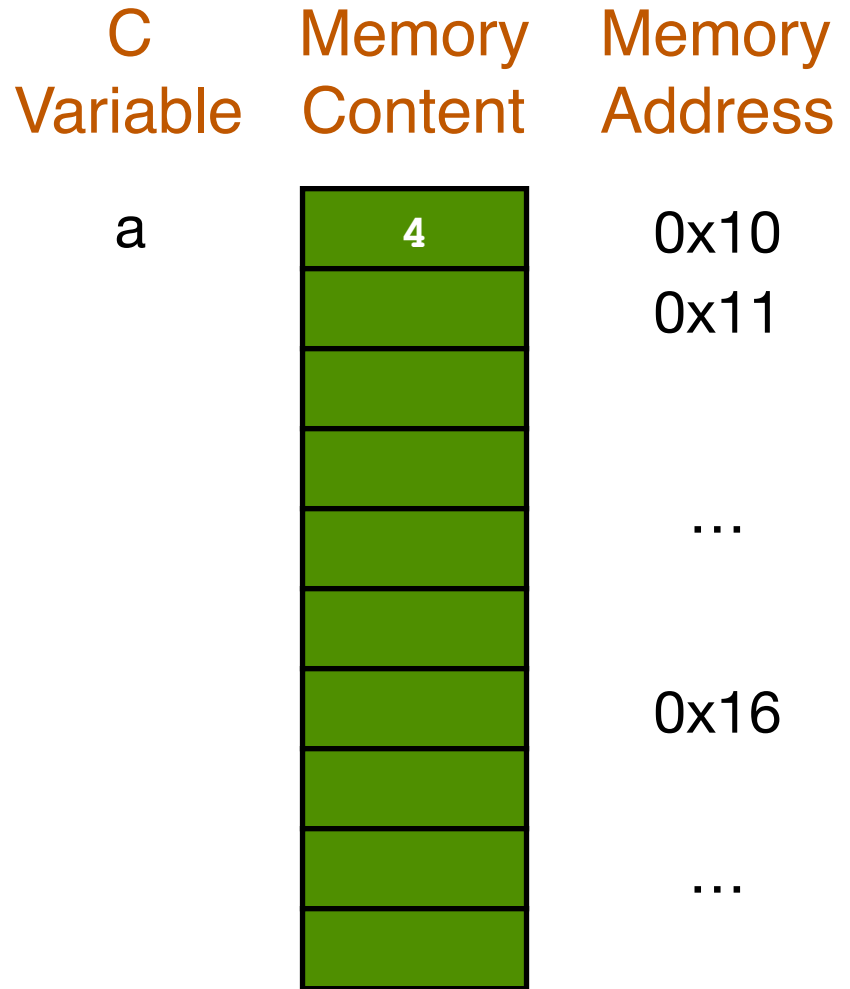
```
char a = 4;
```

```
→ char b = 3;
```

```
char* c;
```

```
c = &a;
```

```
b += (*c);
```



How Does Pointer Work in C???

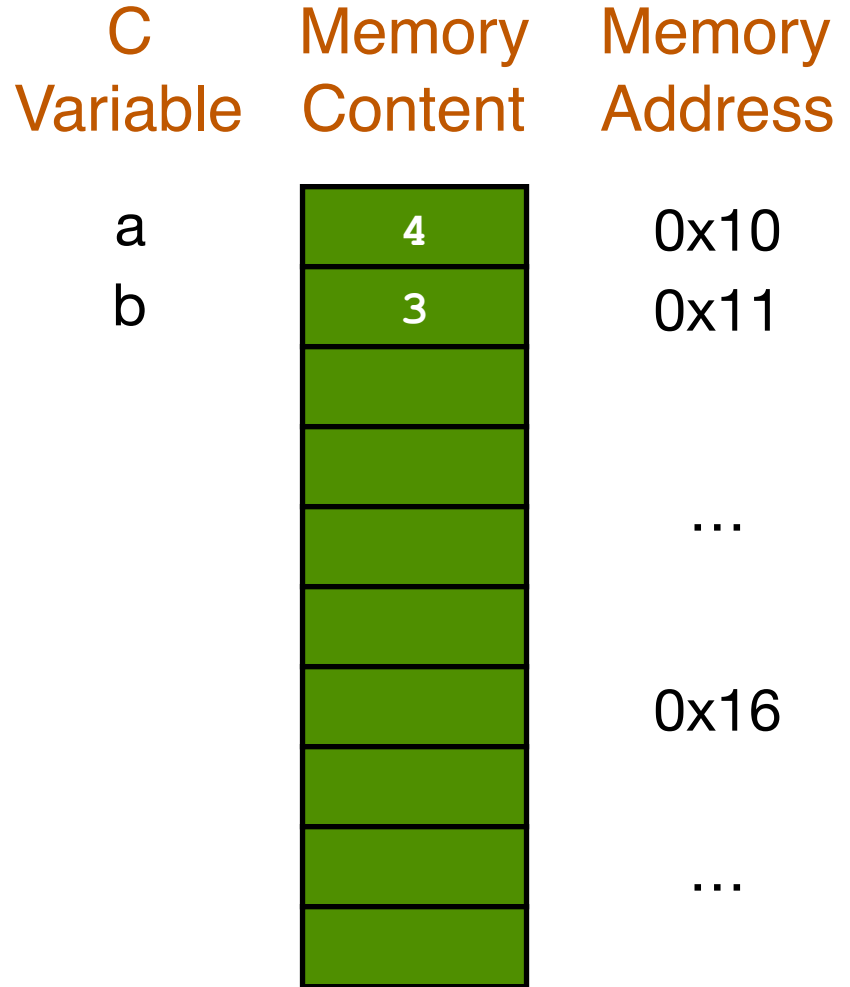
```
char a = 4;
```

```
→ char b = 3;
```

```
char* c;
```

```
c = &a;
```

```
b += (*c);
```



How Does Pointer Work in C???

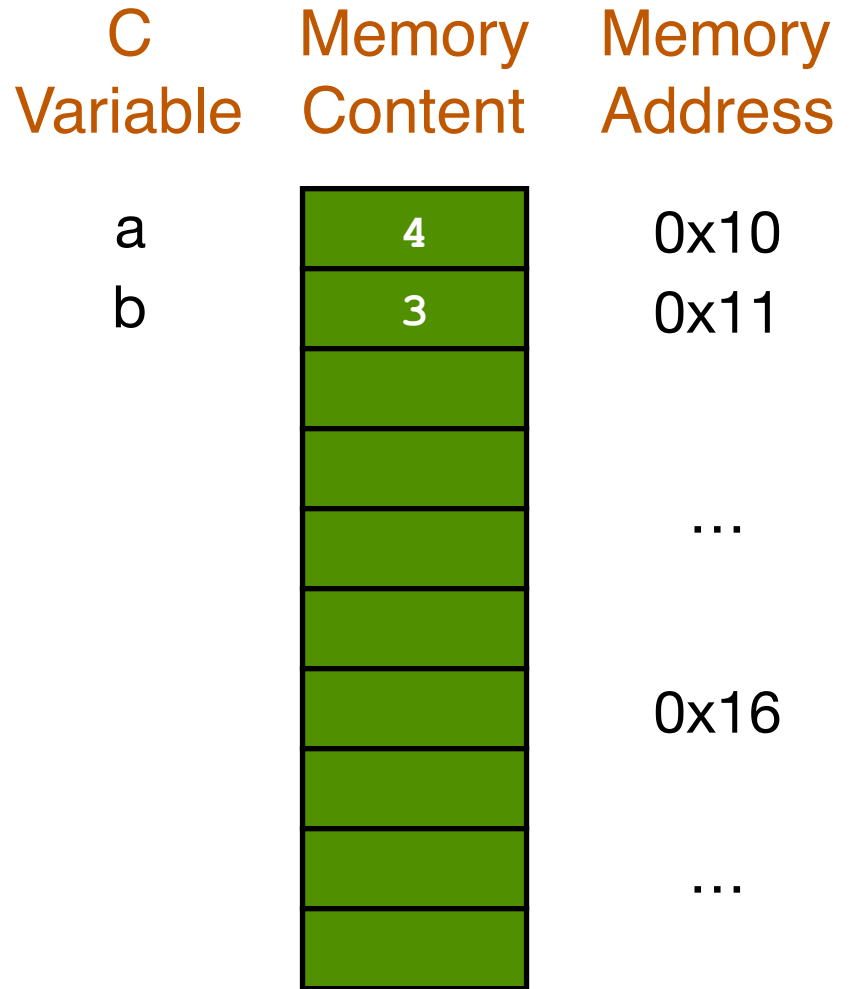
```
char a = 4;
```

```
char b = 3;
```

```
→ char* c;
```

```
c = &a;
```

```
b += (*c);
```



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.

C Variable	Memory Content	Memory Address
a	4	0x10
b	3	0x11
		...
		0x16
		...

How Does Pointer Work in C???

```
char a = 4;
```

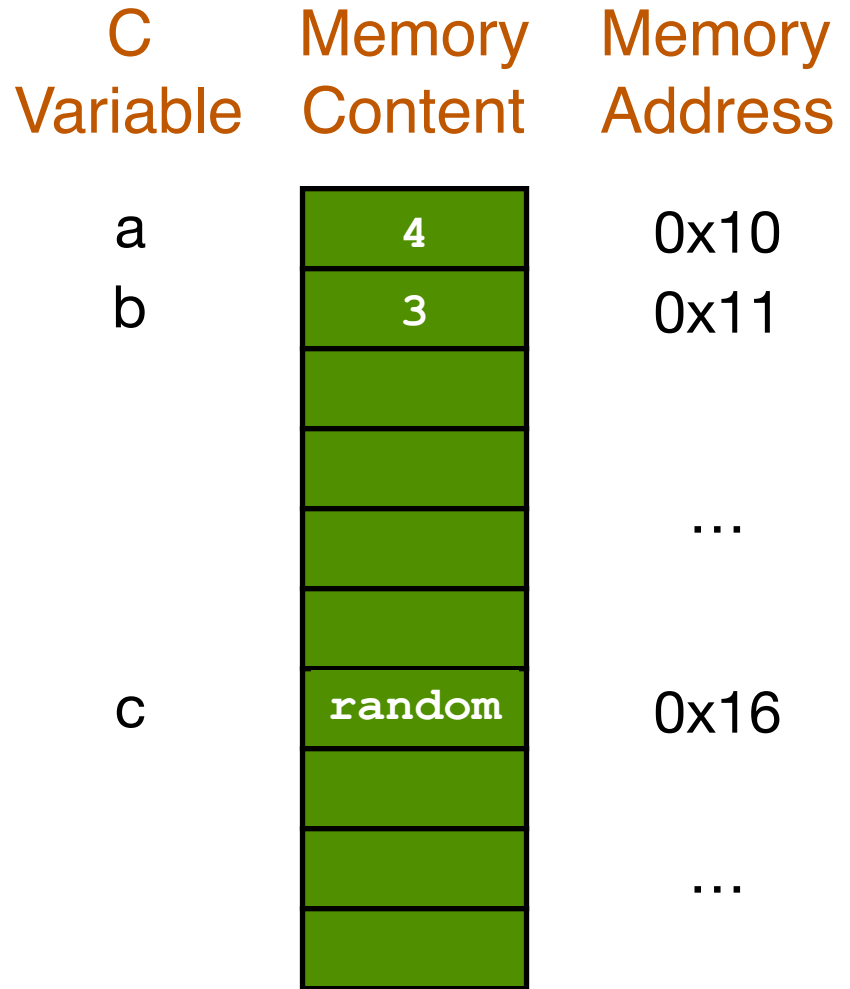
```
char b = 3;
```

```
→ char* c;
```

```
c = &a;
```

```
b += (*c);
```

- The content of a pointer variable is memory address.



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.

C Variable	Memory Content	Memory Address
a	4	0x10
b	3	0x11
		...
c	random	0x16
		...

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.

C Variable	Memory Content	Memory Address
a	4	0x10
b	3	0x11
		...
c	random	0x16
		...

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.

C Variable	Memory Content	Memory Address
a	4	0x10
b	3	0x11
		...
c	0x10	0x16
		...

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.

C Variable	Memory Content	Memory Address
a	4	0x10
b	3	0x11
		...
c	0x10	0x16
		...

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)

C Variable	Memory Content	Memory Address
a	4	0x10
b	3	0x11
		...
c	0x10	0x16
		...

How Does Pointer Work in C???

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

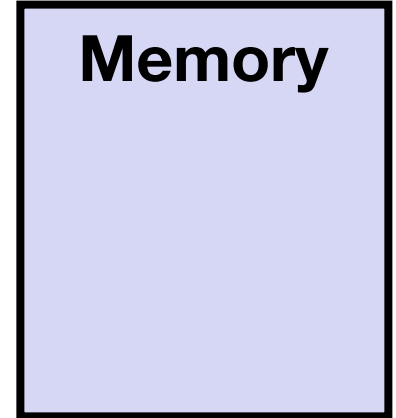
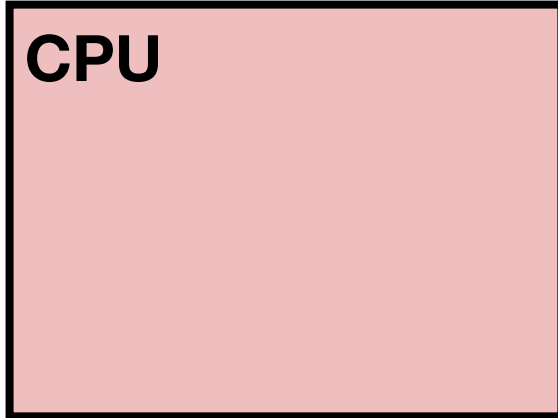
- The content of a pointer variable is memory address.
- The '**&**' operator (address-of operator) returns the memory address of a variable.
- The '*****' operator returns the content stored at the memory location pointed by the pointer variable (dereferencing)

C Variable	Memory Content	Memory Address
a	4	0x10
b	7	0x11
		...
c	0x10	0x16
		...

Assembly Code's View of Computer: ISA

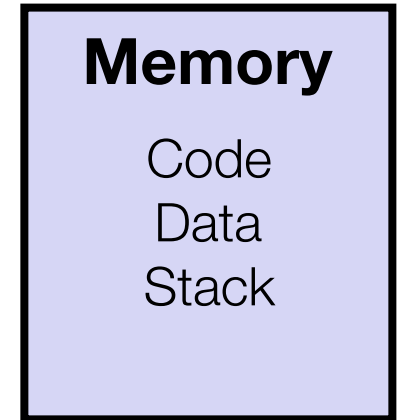
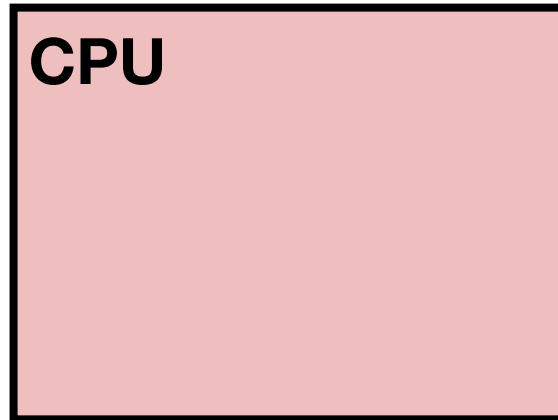
Assembly Code's View of Computer: ISA

Assembly
Programmer's
Perspective
of a Computer



Assembly Code's View of Computer: ISA

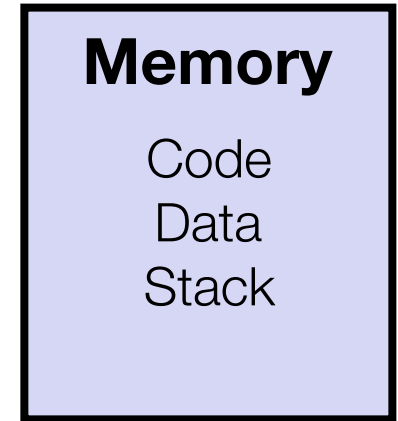
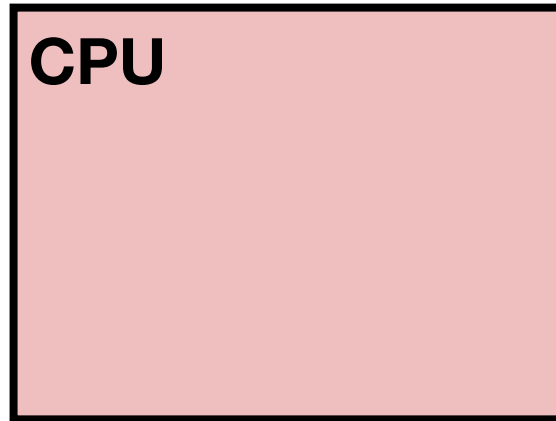
Assembly
Programmer's
Perspective
of a Computer



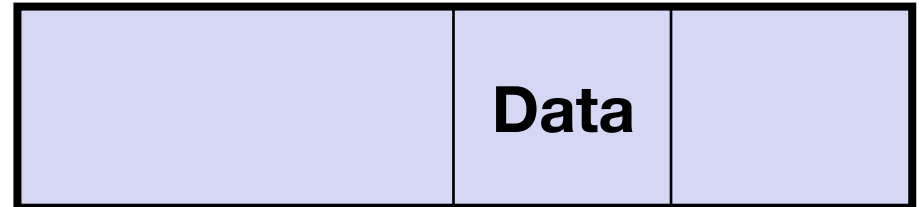
- (Byte Addressable) Memory
 - Code: instructions
 - Data
 - Stack to support function call

Assembly Code's View of Computer: ISA

Assembly
Programmer's
Perspective
of a Computer

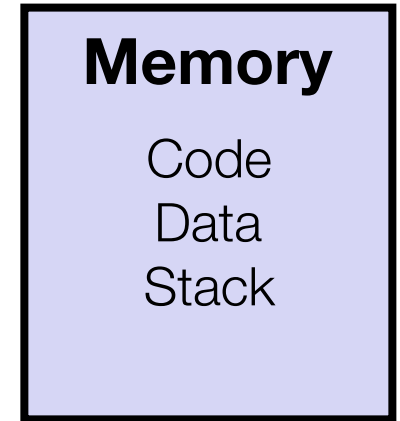
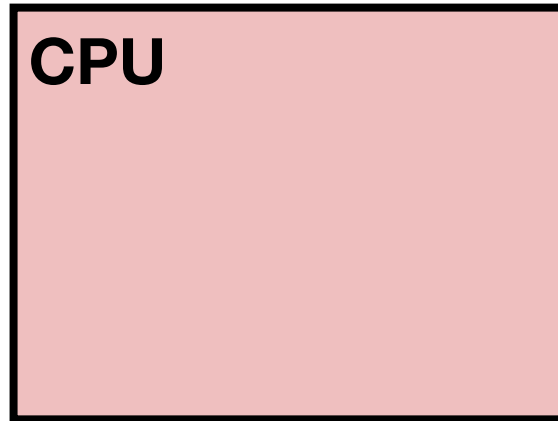


- (Byte Addressable) Memory
 - Code: instructions
 - Data
 - Stack to support function call

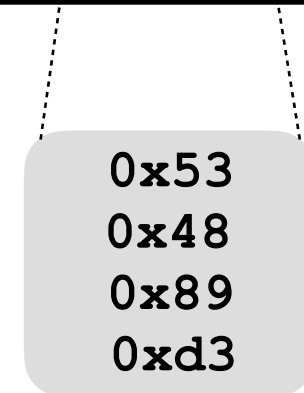
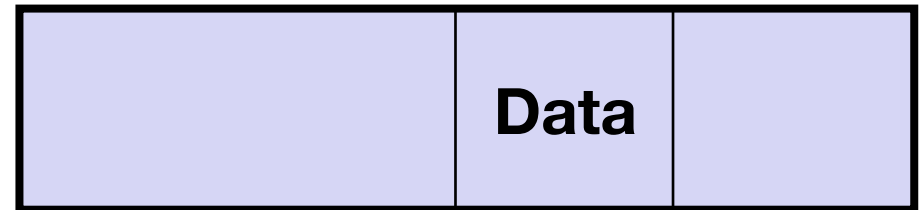


Assembly Code's View of Computer: ISA

Assembly
Programmer's
Perspective
of a Computer

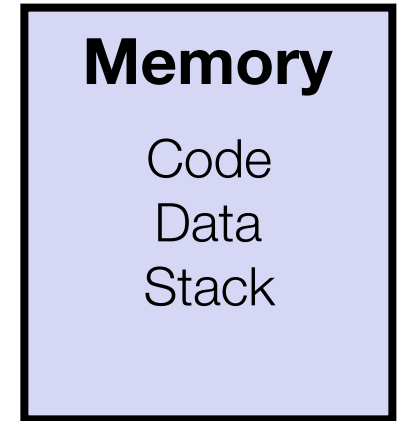
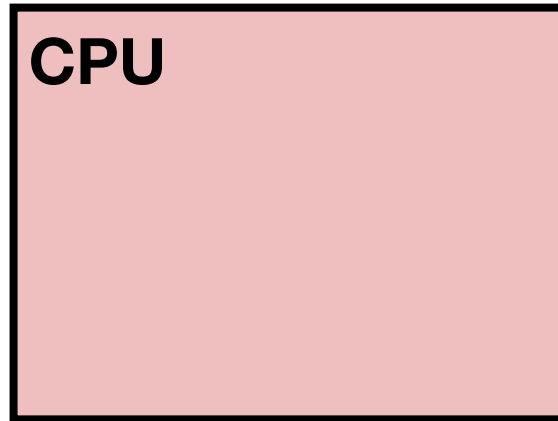


- (Byte Addressable) Memory
 - Code: instructions
 - Data
 - Stack to support function call

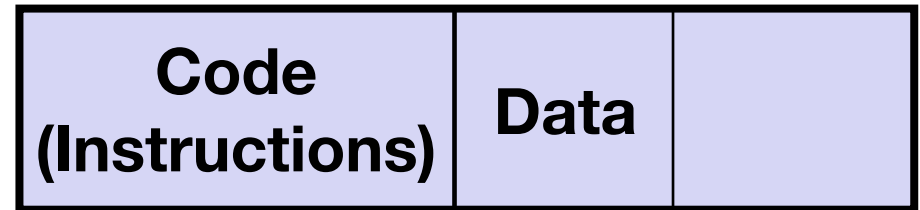


Assembly Code's View of Computer: ISA

Assembly
Programmer's
Perspective
of a Computer



- (Byte Addressable) Memory
 - Code: instructions
 - Data
 - Stack to support function call

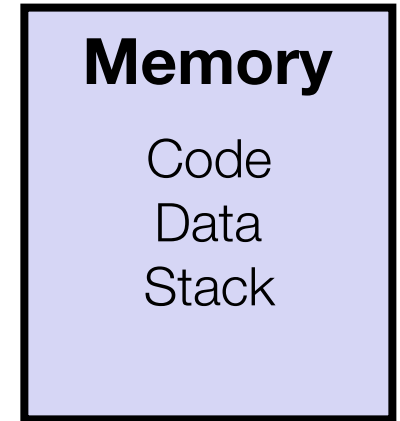
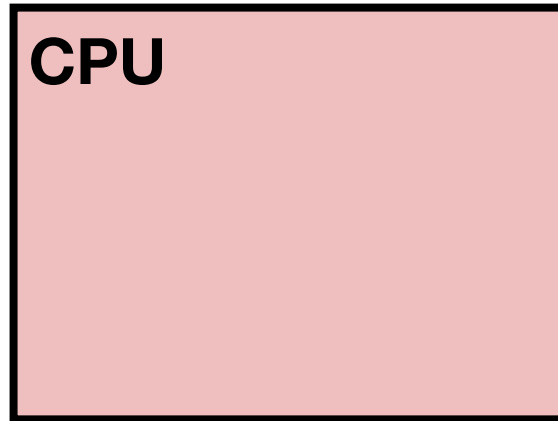


0x78
0xfe
0xe3
0x05

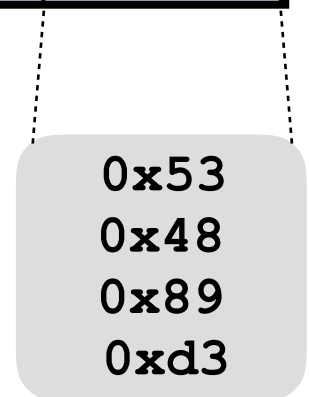
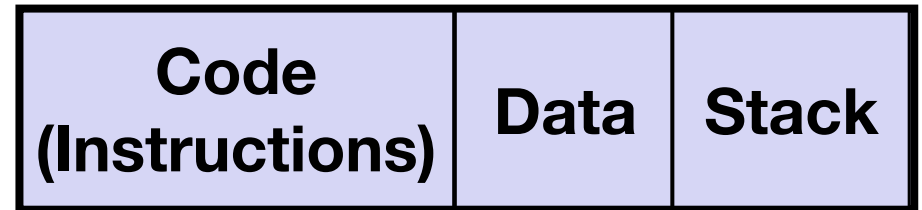
Instruction is the fundamental
unit of work.
All instructions are encoded as
bits (just like data!)

Assembly Code's View of Computer: ISA

Assembly
Programmer's
Perspective
of a Computer

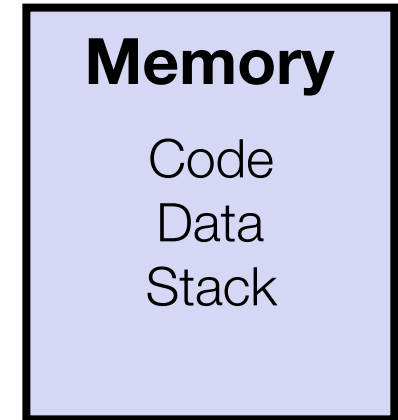
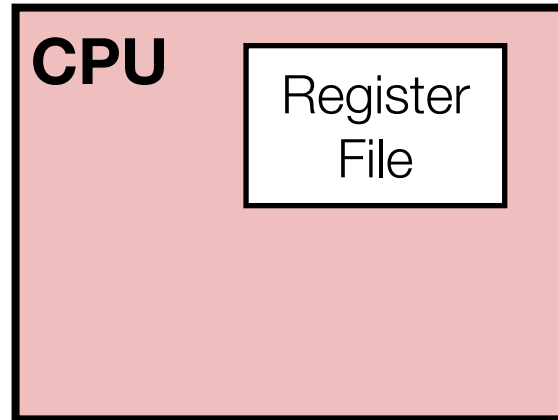


- (Byte Addressable) Memory
 - Code: instructions
 - Data
 - Stack to support function call



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

x86-64 Integer Register File

← 8 Bytes →

<code>%rax</code>
<code>%rbx</code>
<code>%rcx</code>
<code>%rdx</code>
<code>%rsi</code>
<code>%rdi</code>
<code>%rsp</code>
<code>%rbp</code>

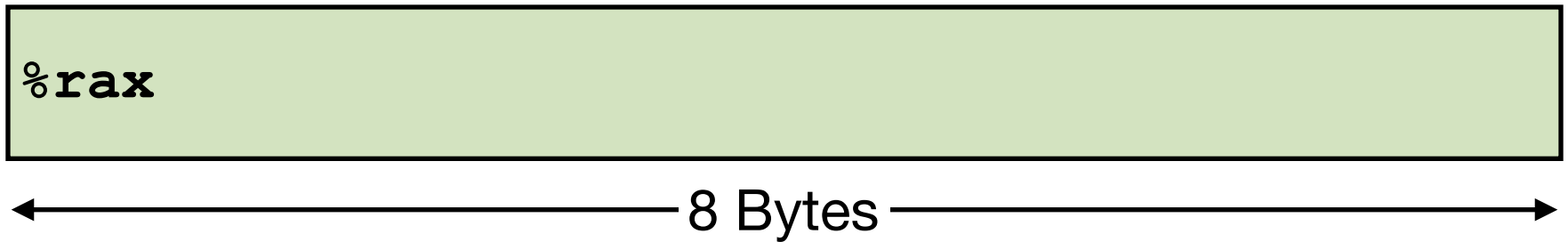
<code>%r8</code>
<code>%r9</code>
<code>%r10</code>
<code>%r11</code>
<code>%r12</code>
<code>%r13</code>
<code>%r14</code>
<code>%r15</code>

x86-64 Integer Register File

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

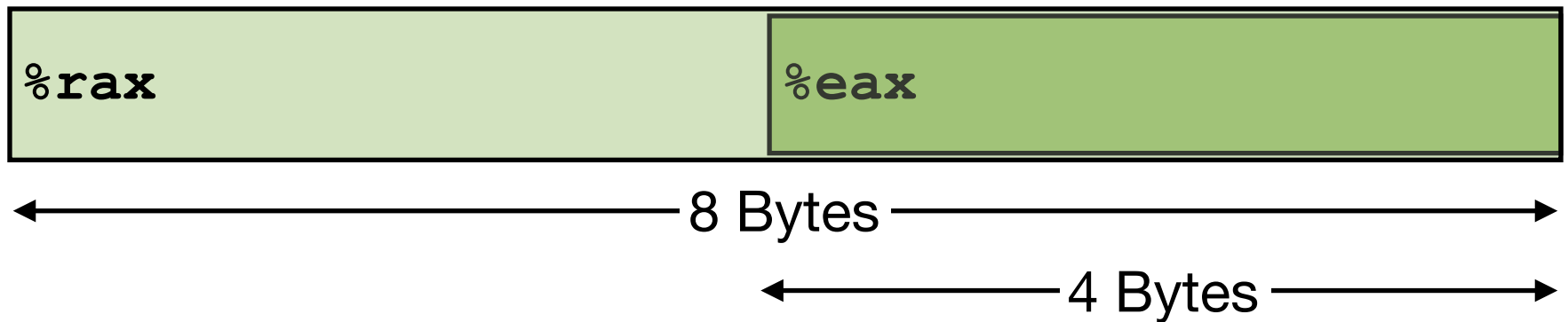
x86-64 Integer Register File

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



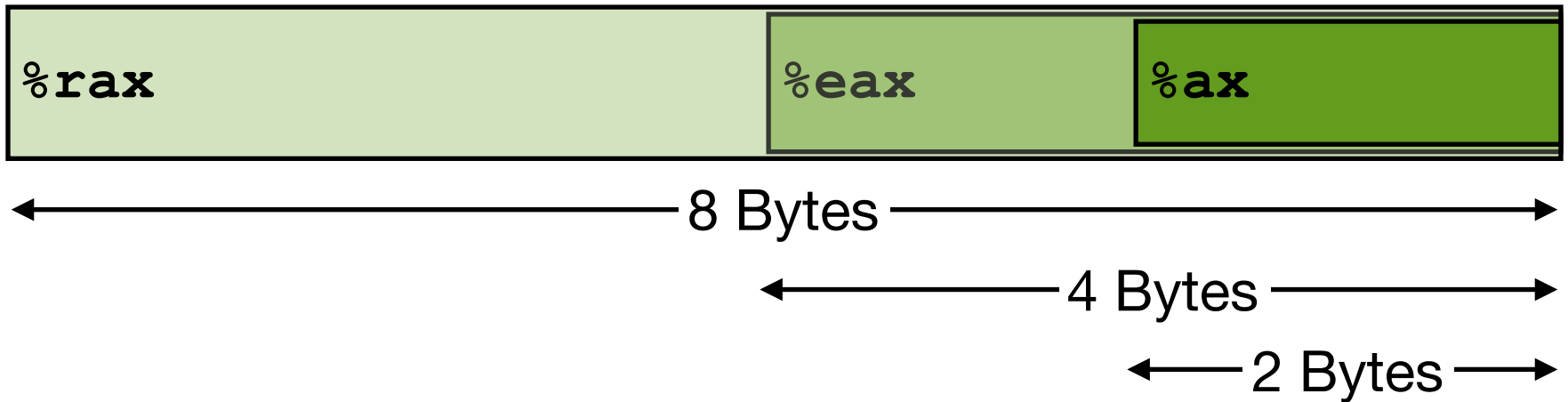
x86-64 Integer Register File

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



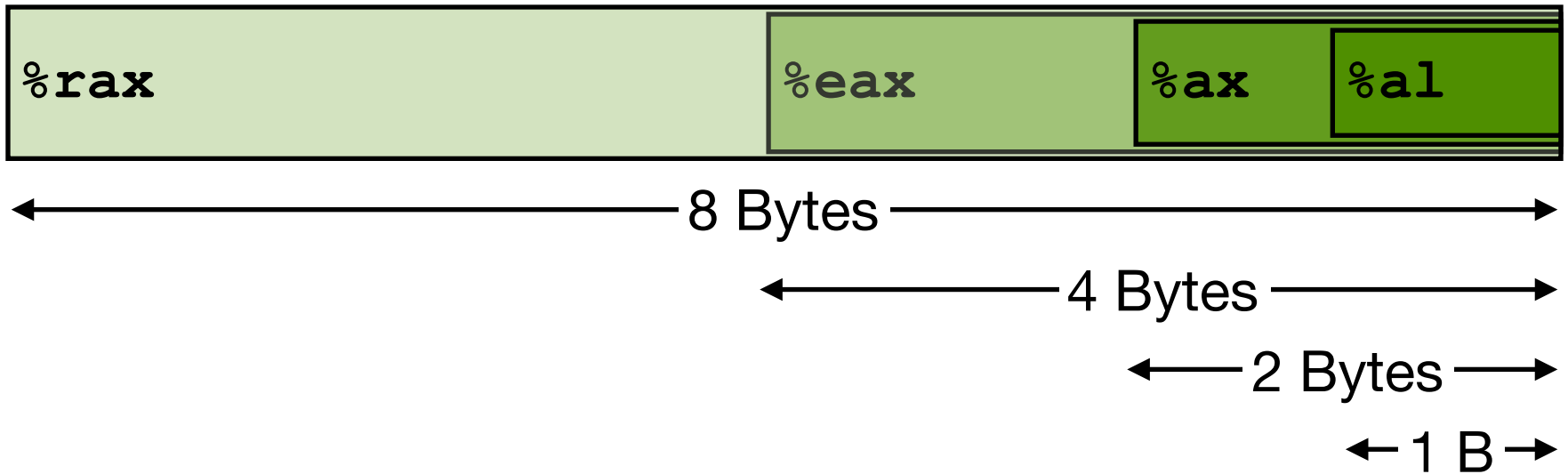
x86-64 Integer Register File

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



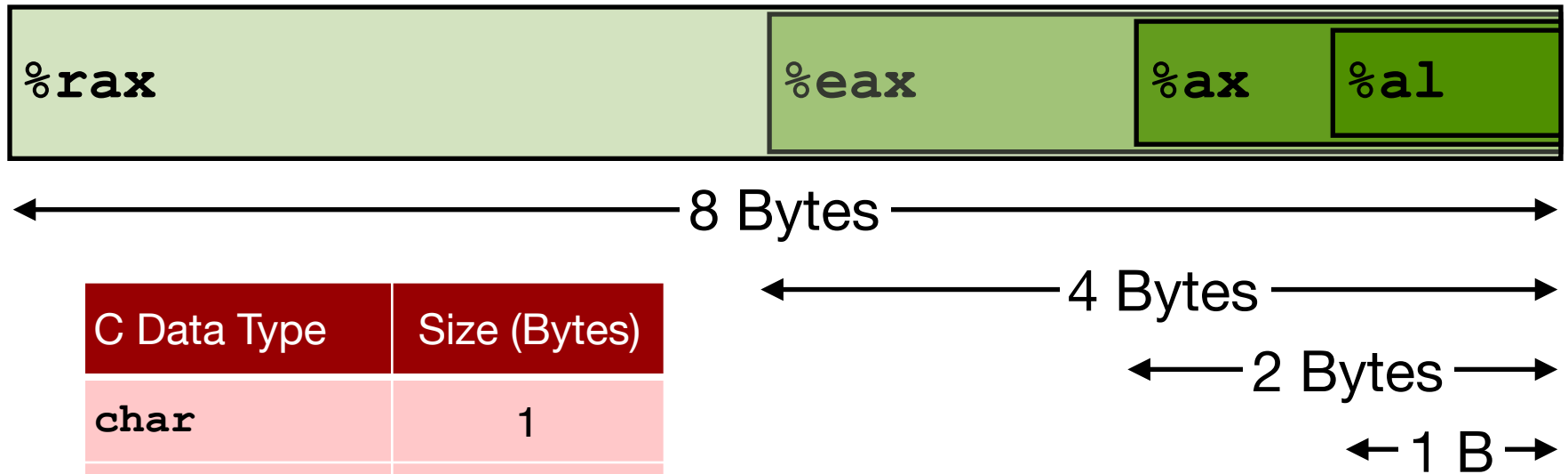
x86-64 Integer Register File

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



x86-64 Integer Register File

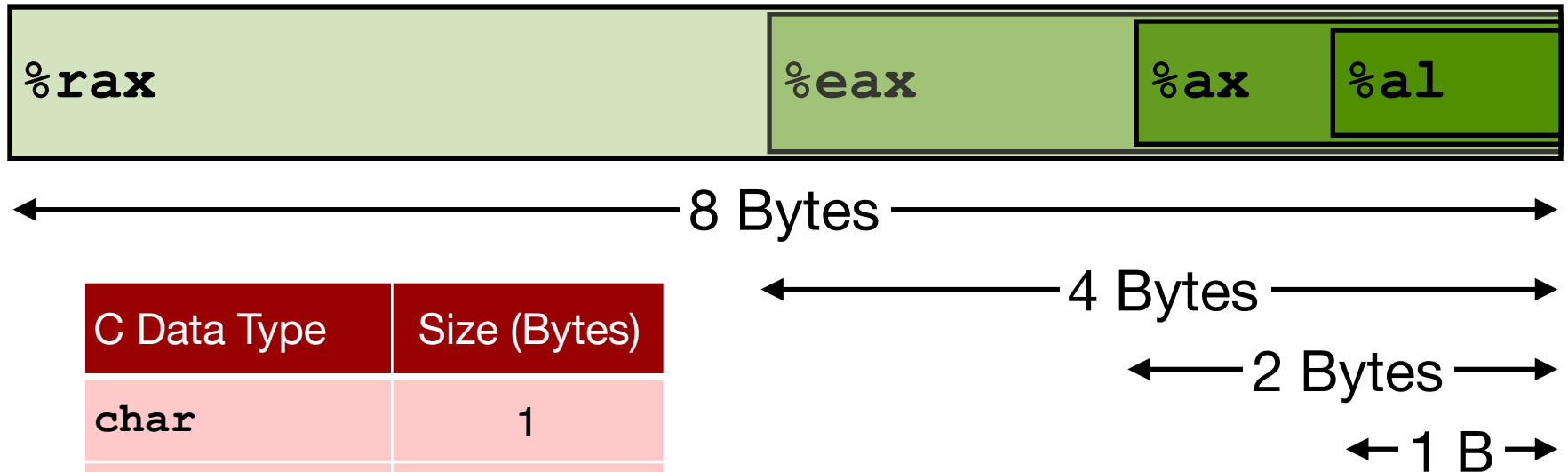
- Lower-half of each register can be independently addressed (until 1 bytes)



C Data Type	Size (Bytes)
<code>char</code>	1
<code>short</code>	2
<code>int</code>	4
<code>long</code>	8
Pointer	8

x86-64 Integer Register File

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

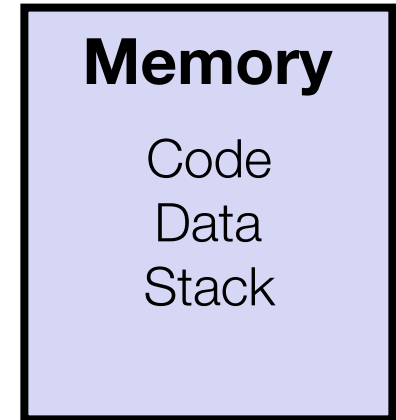
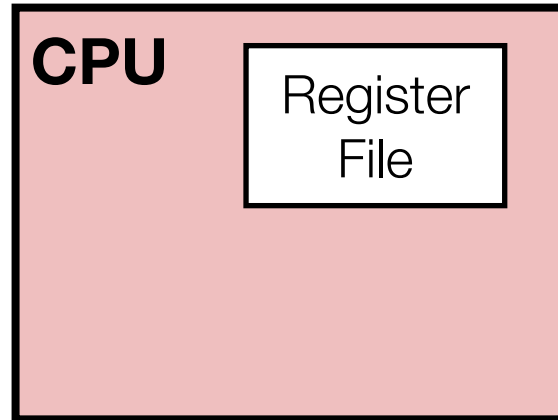


C Data Type	Size (Bytes)
<code>char</code>	1
<code>short</code>	2
<code>int</code>	4
<code>long</code>	8
Pointer	8

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

Assembly Code's View of Computer: ISA

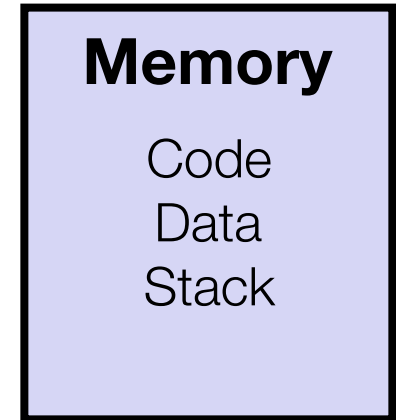
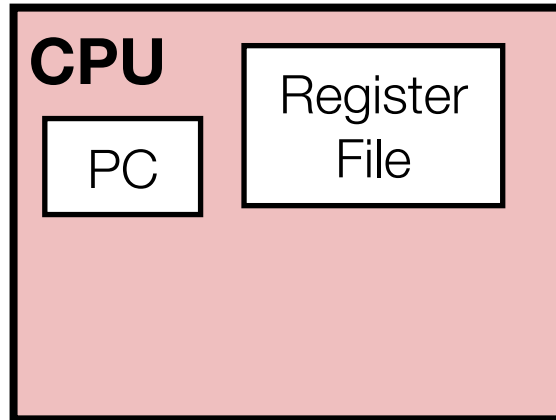
Assembly Programmer's Perspective of a Computer



- (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

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

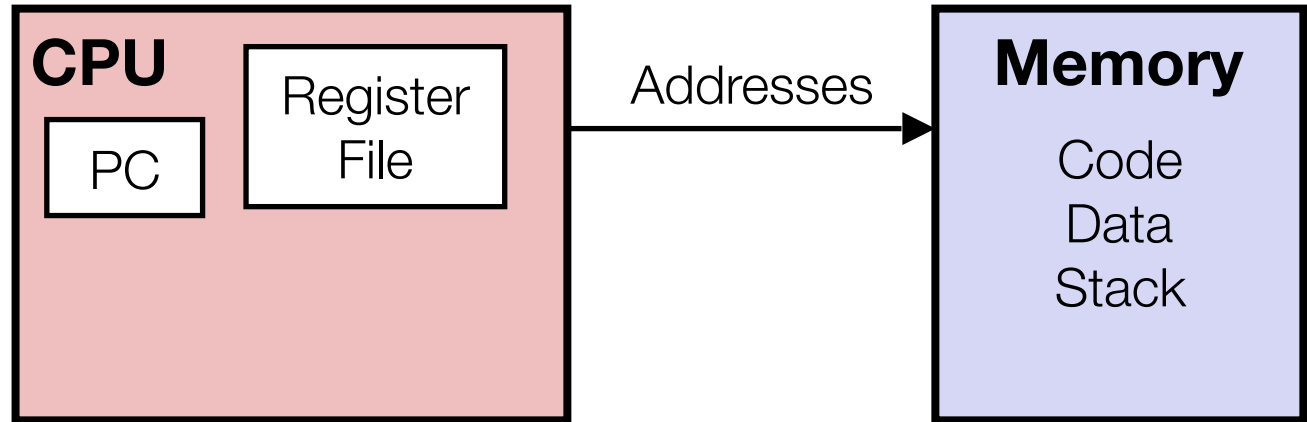
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- PC: Program counter

- A special register containing address of next instruction
- Called "RIP" in x86-64

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

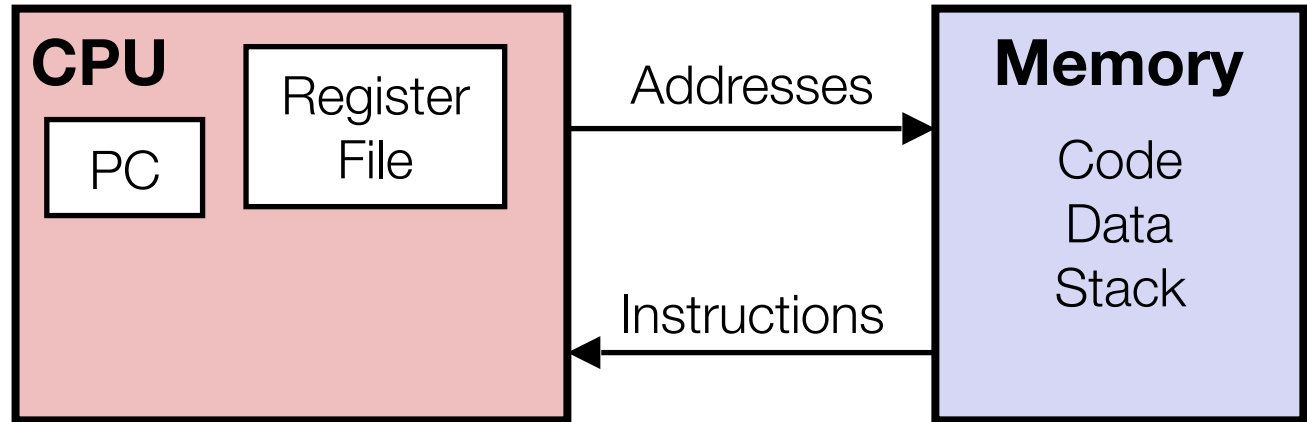
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- PC: Program counter

- A special register containing address of next instruction
- Called "RIP" in x86-64

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

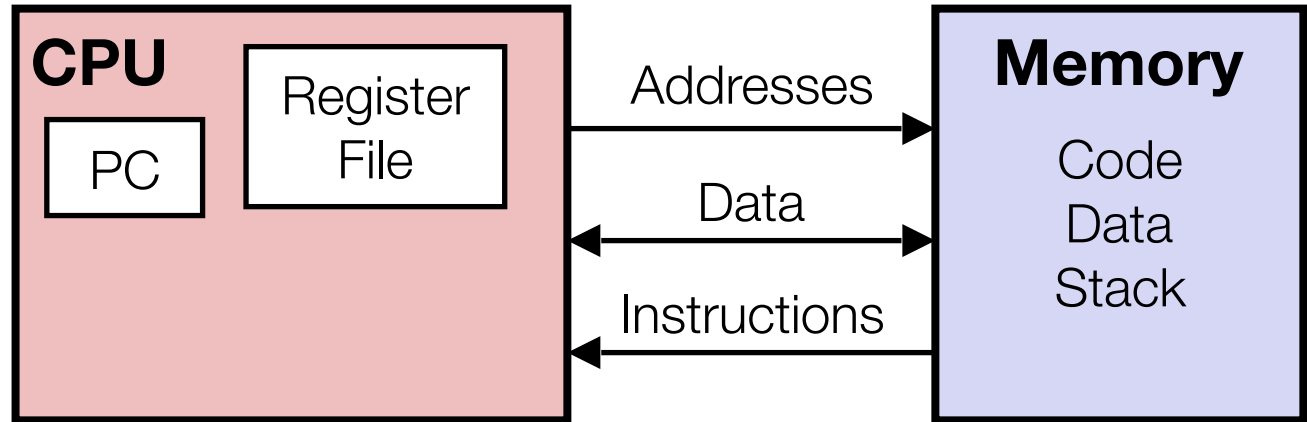
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- PC: Program counter

- A special register containing address of next instruction
- Called "RIP" in x86-64

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

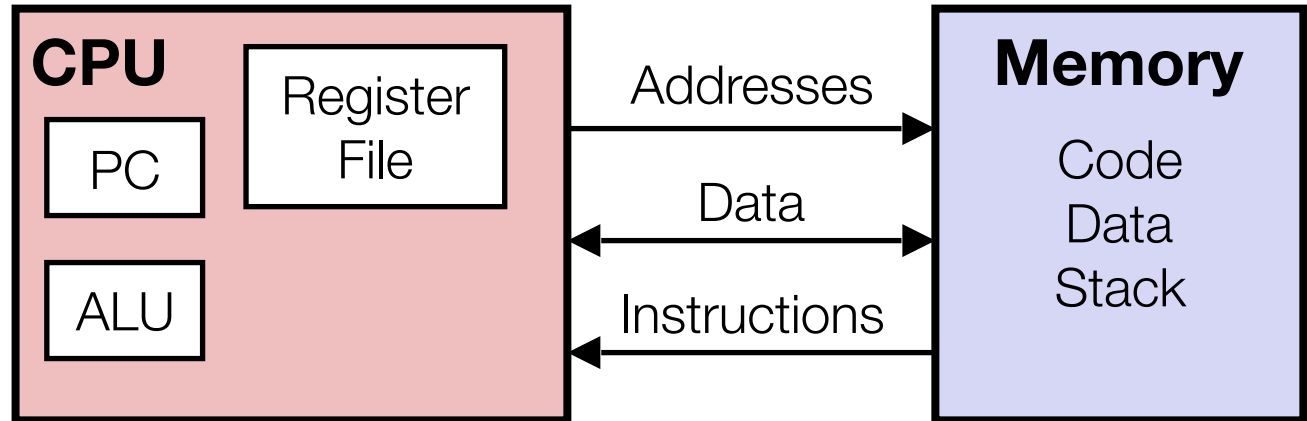
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- PC: Program counter

- A special register containing address of next instruction
- Called "RIP" in x86-64

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

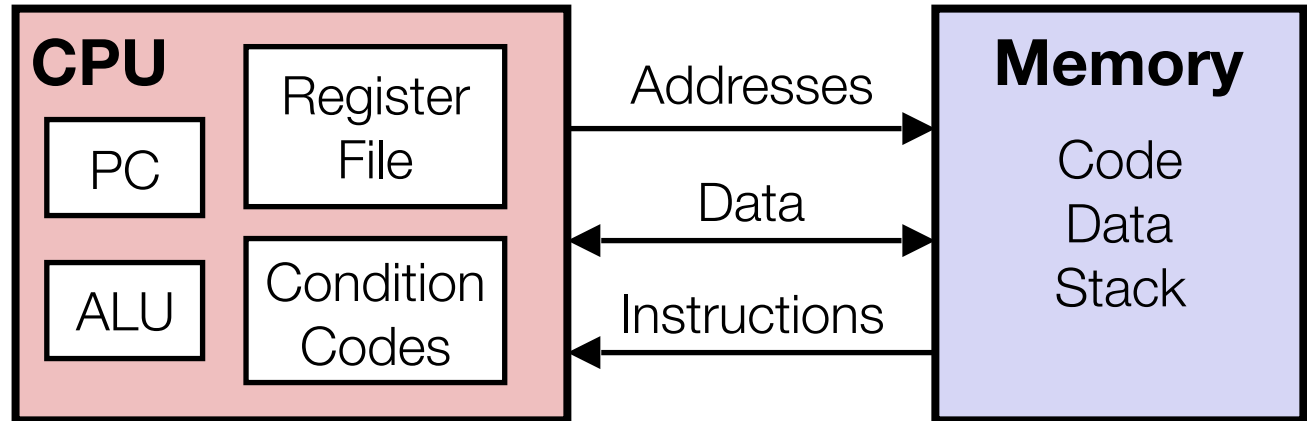
- A special register containing address of next instruction
- Called "RIP" in x86-64

- Arithmetic logic unit (ALU)

- Where computation happens

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)

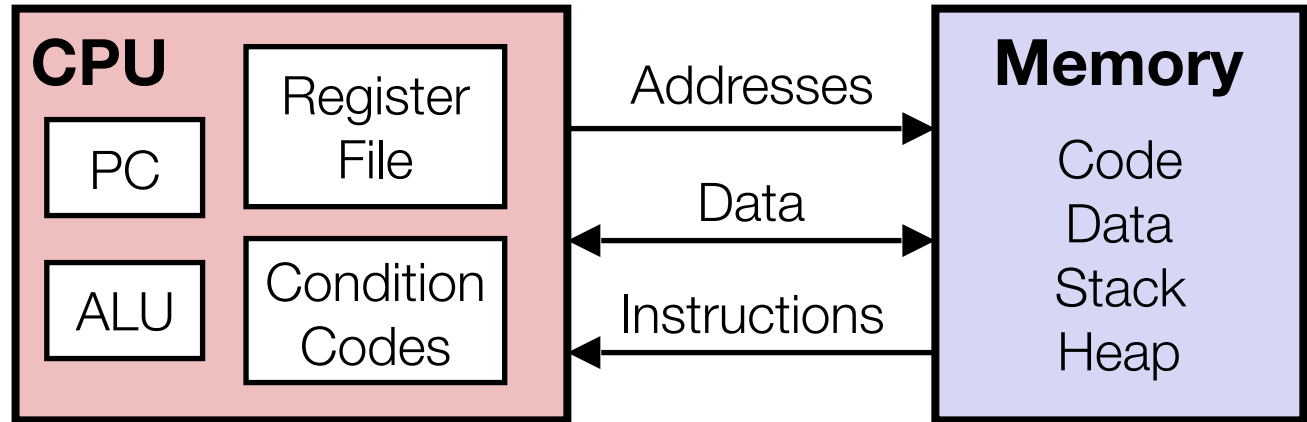
- Where computation happens

- Condition codes

- Store status information about most recent arithmetic or logical operation
- Used for conditional branch

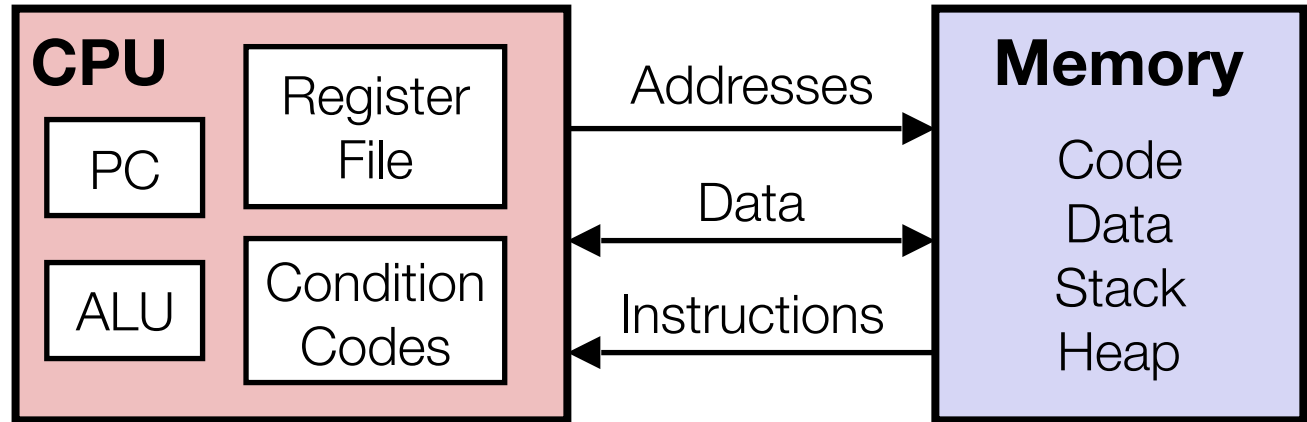
Assembly Program Instructions

Assembly
Programmer's
Perspective
of a Computer



Assembly Program Instructions

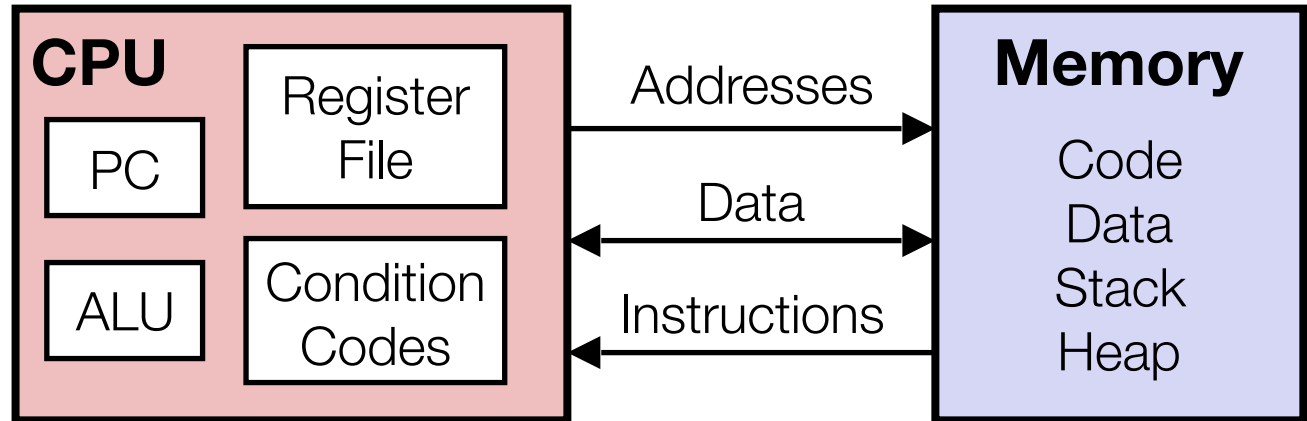
Assembly
Programmer's
Perspective
of a Computer



- *Compute Instruction*: Perform arithmetics on register or memory data
 - `addq %eax, %ebx`
 - C constructs: +, -, >>, etc.

Assembly Program Instructions

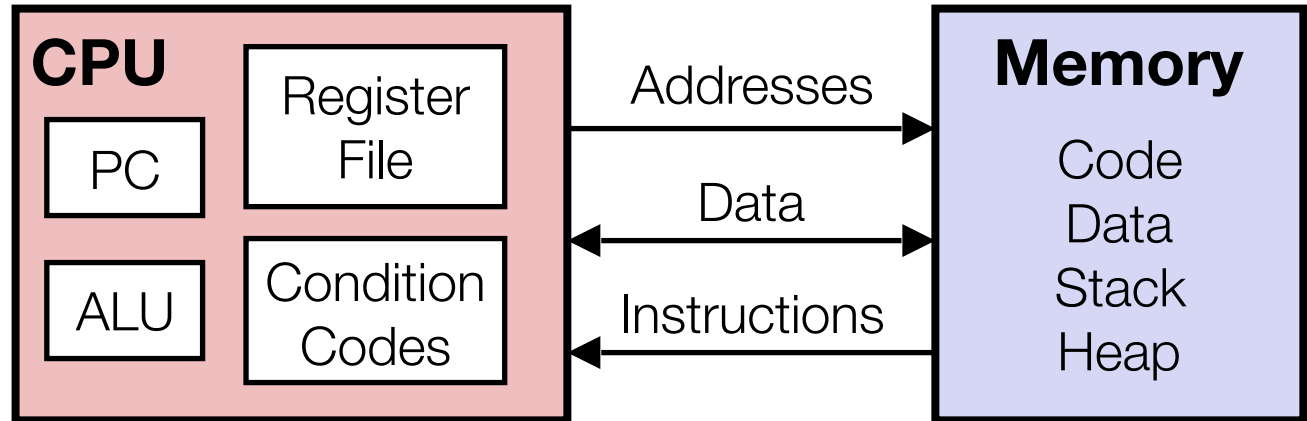
Assembly Programmer's Perspective of a Computer



- *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)`

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;  
}
```

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
```

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
```

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**

Memory

```
0x53  
0x48  
0x89  
0xd3  
0xe8  
0xf2  
0xff  
0xff  
0xff  
0x48  
0x89  
0x03  
0x5b  
0xc3
```


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

Turning C into Object Code

Generated x86-64 Assembly

Binary Code for **sumstore**

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

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
```

Turning C into Object Code

Generated x86-64 Assembly

Binary Code for **sumstore**

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

Address Memory

0x0400595

```
0x53  
0x48  
0x89  
0xd3  
0xe8  
0xf2  
0xff  
0xff  
0xff  
0x48  
0x89  
0x03  
0x5b  
0xc3
```

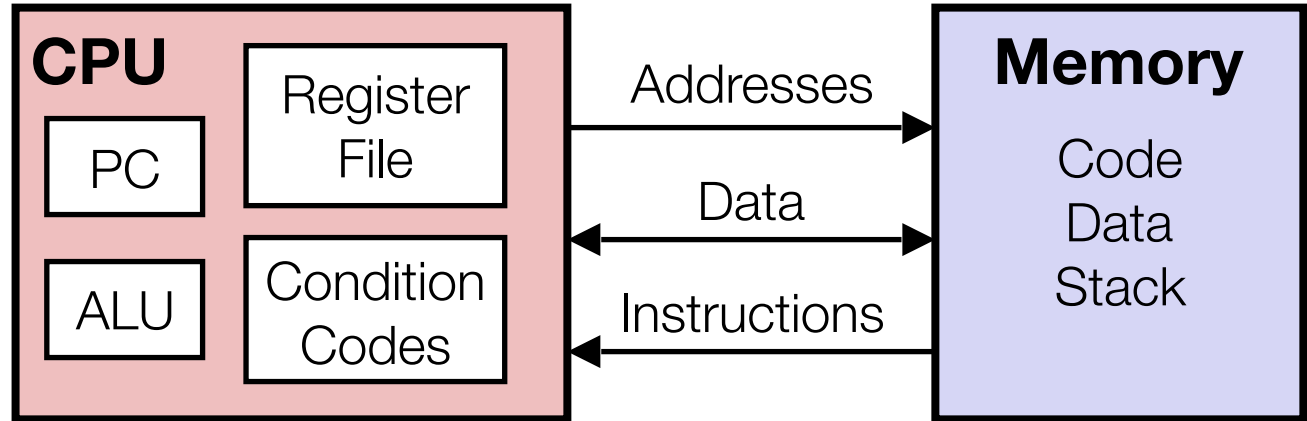
Obtain (on CSUG machine) with command

```
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

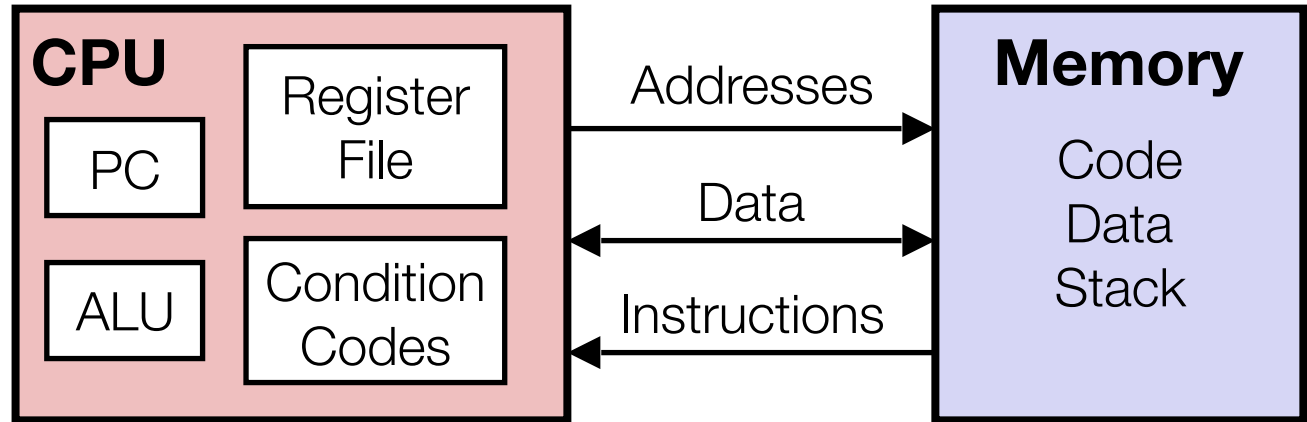
Assembly
Programmer's
Perspective
of a Computer



Fetch Instruction
(According to PC)

Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer

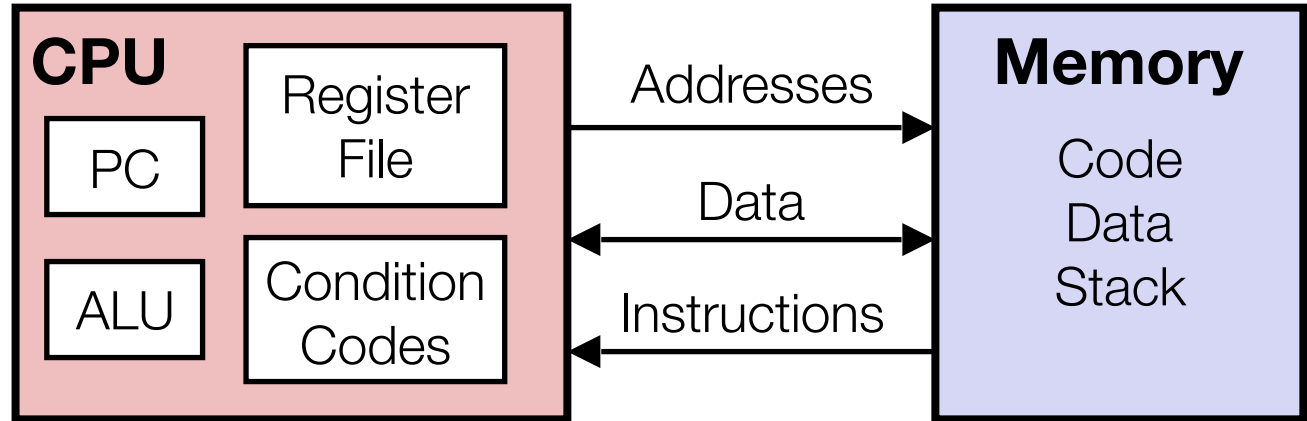


Fetch Instruction
(According to PC)

0x4801d8

Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer

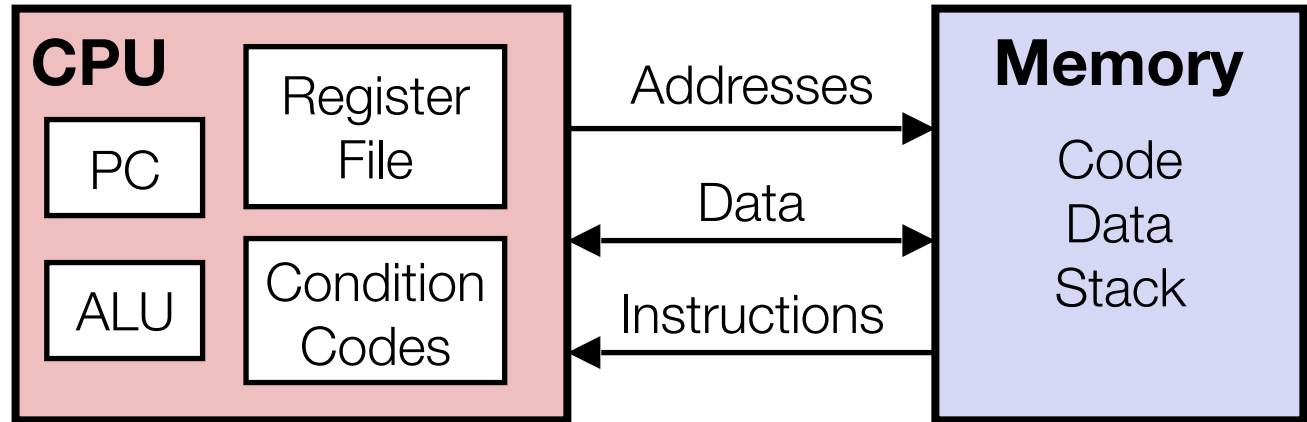


Fetch Instruction (According to PC) → Decode Instruction

```
addq %rax, (%rbx)
```

Instruction Processing Sequence

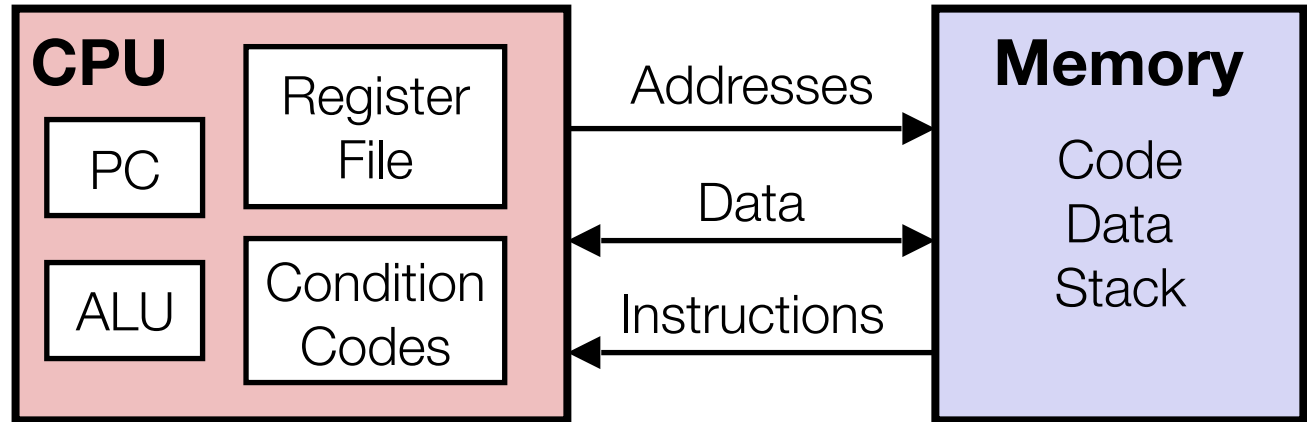
Assembly
Programmer's
Perspective
of a Computer



Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands

Instruction Processing Sequence

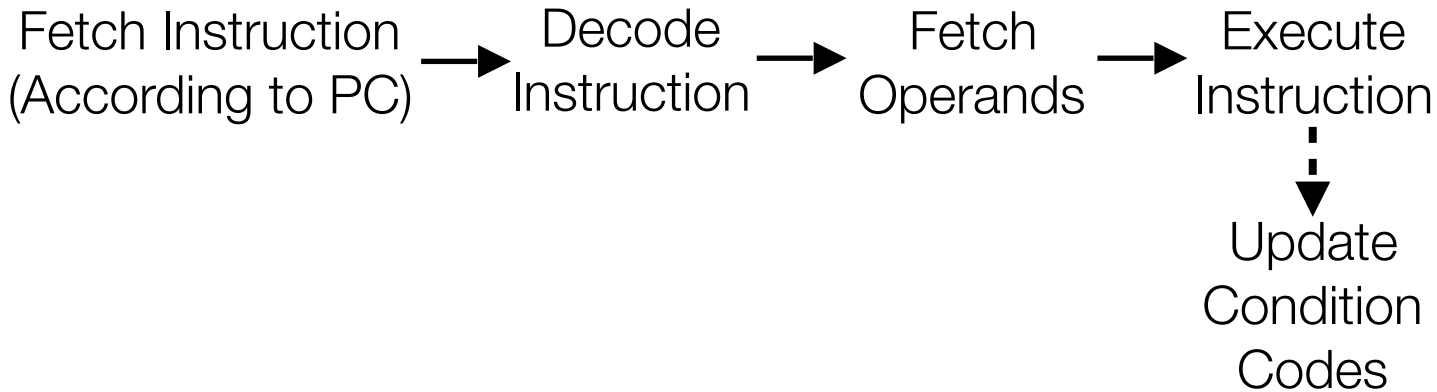
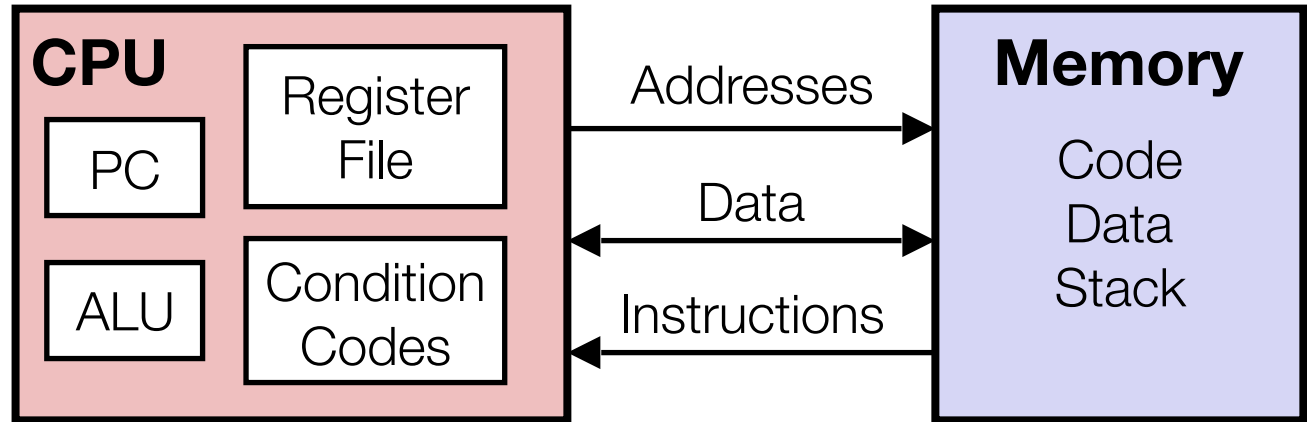
Assembly
Programmer's
Perspective
of a Computer



Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction

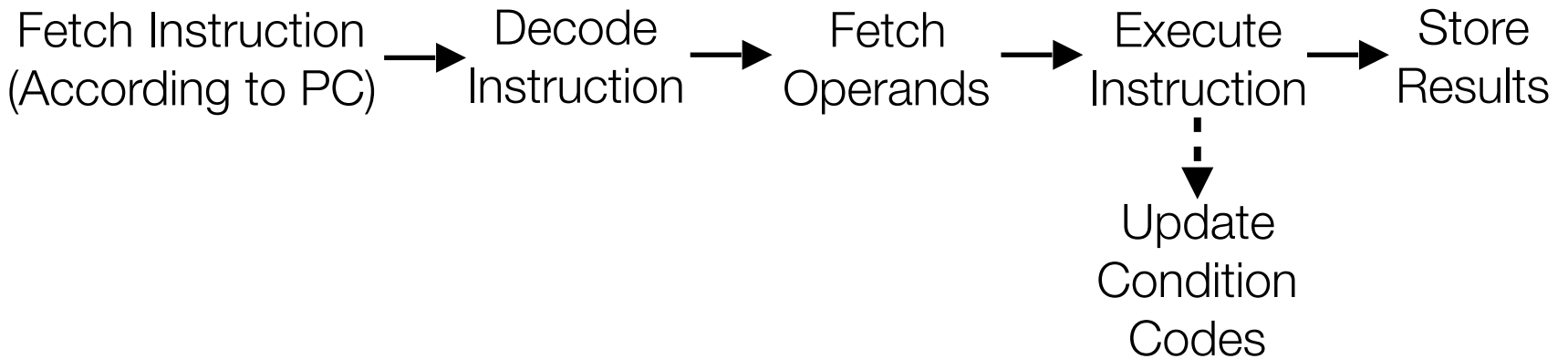
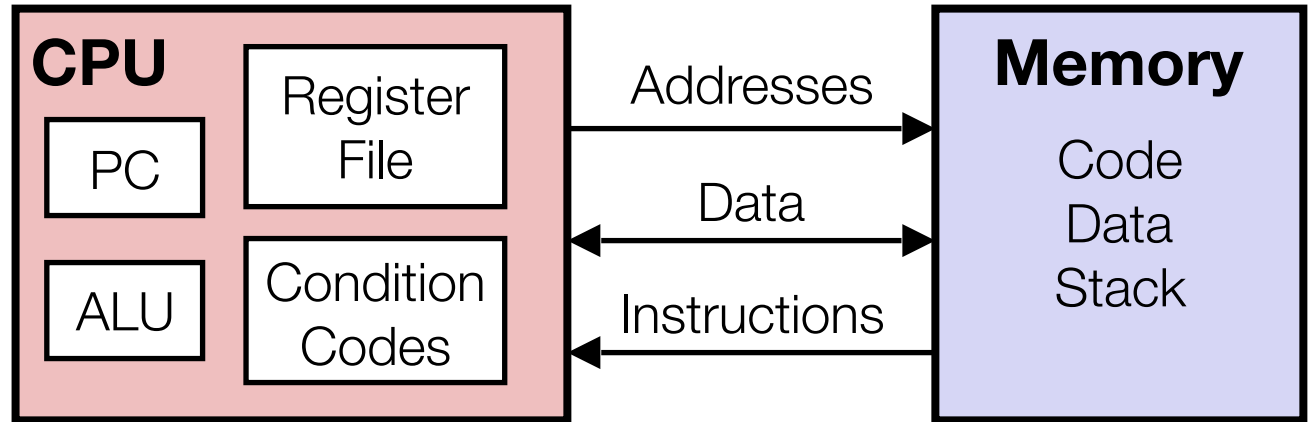
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



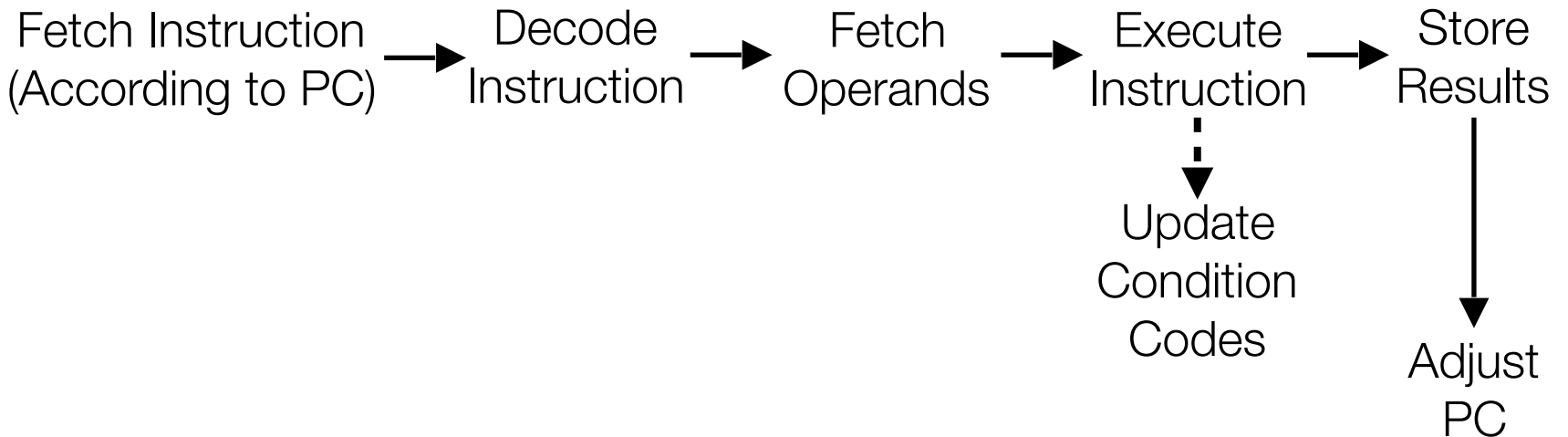
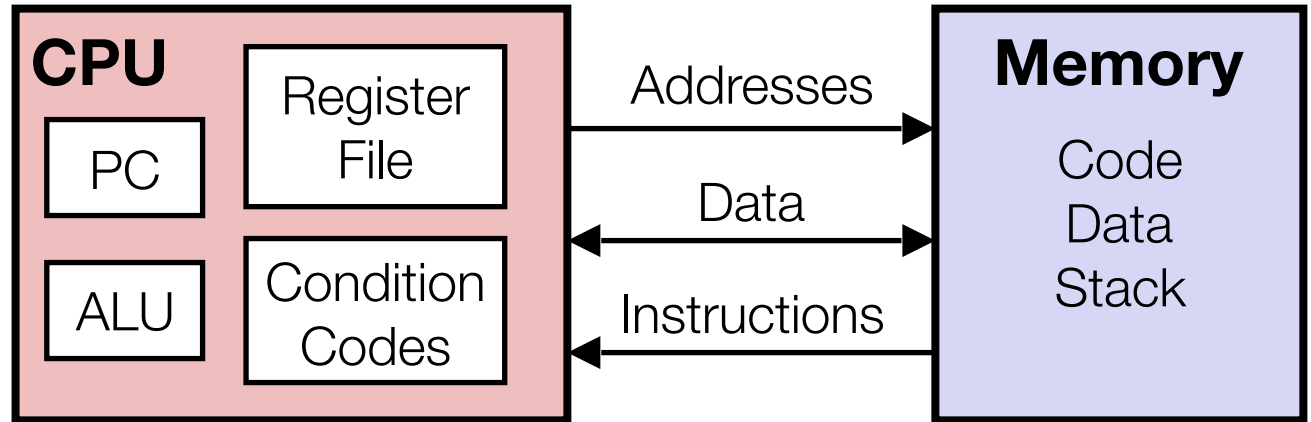
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



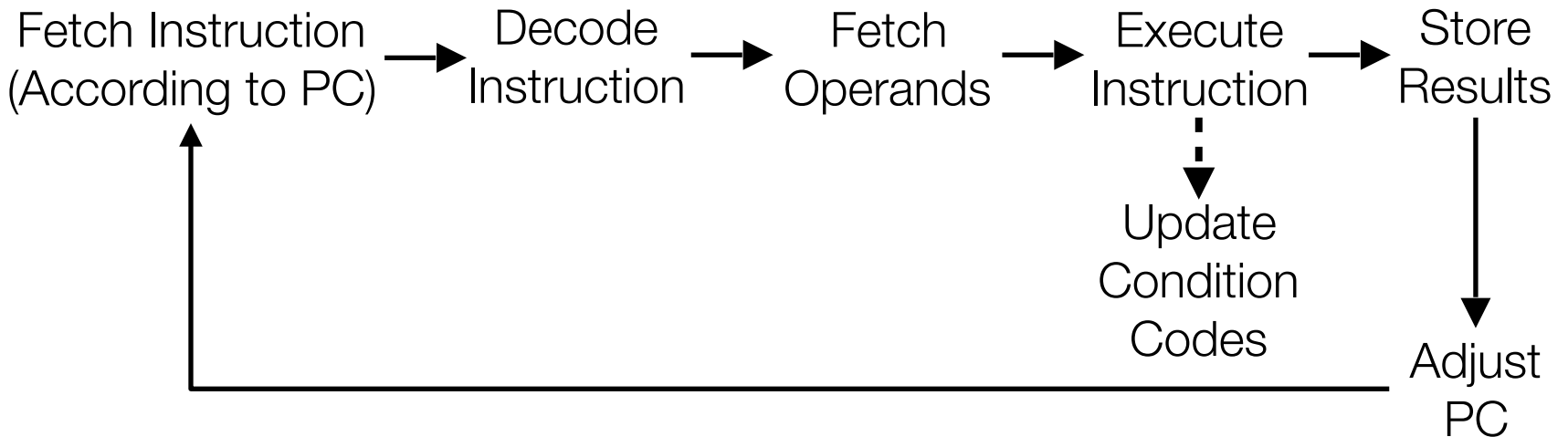
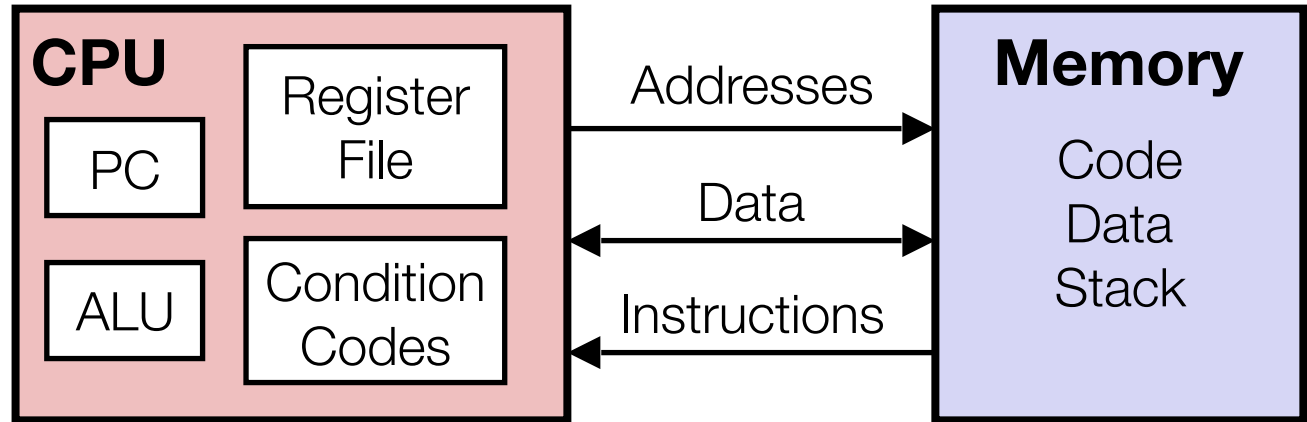
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer

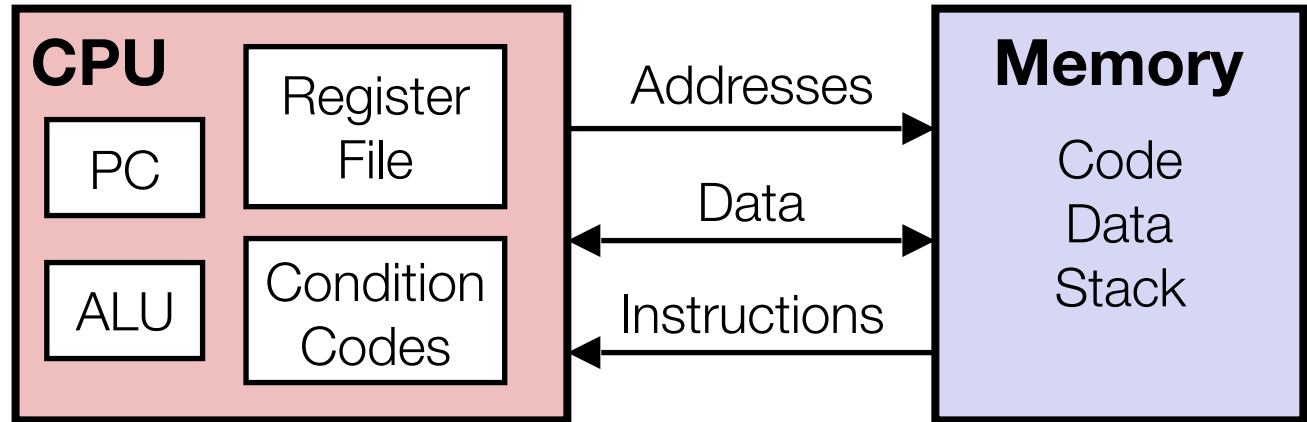


Today: Compute and Control Instructions

- 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

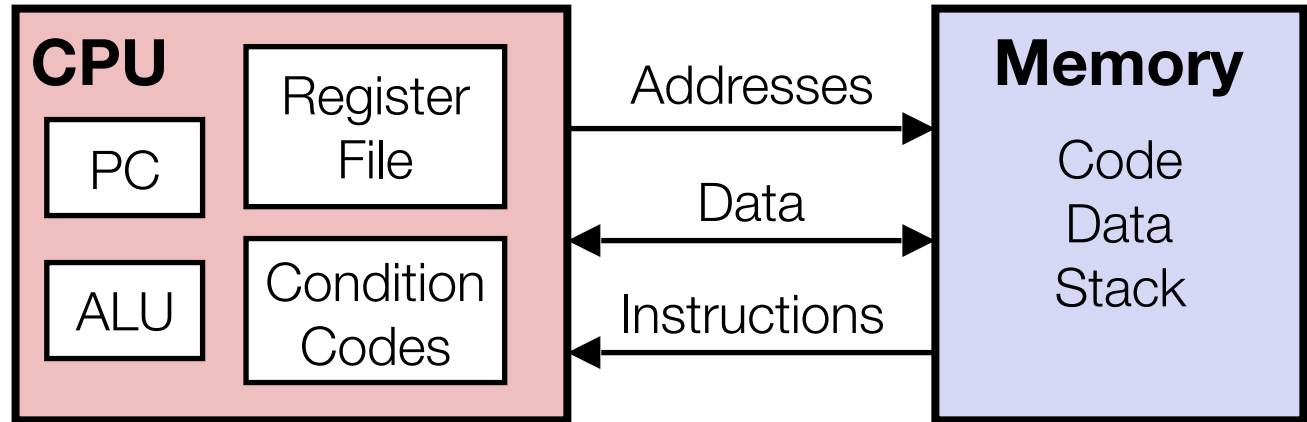
Assembly
Programmer's
Perspective
of a Computer



- Initially all data is in the memory

Data Movement in Processors

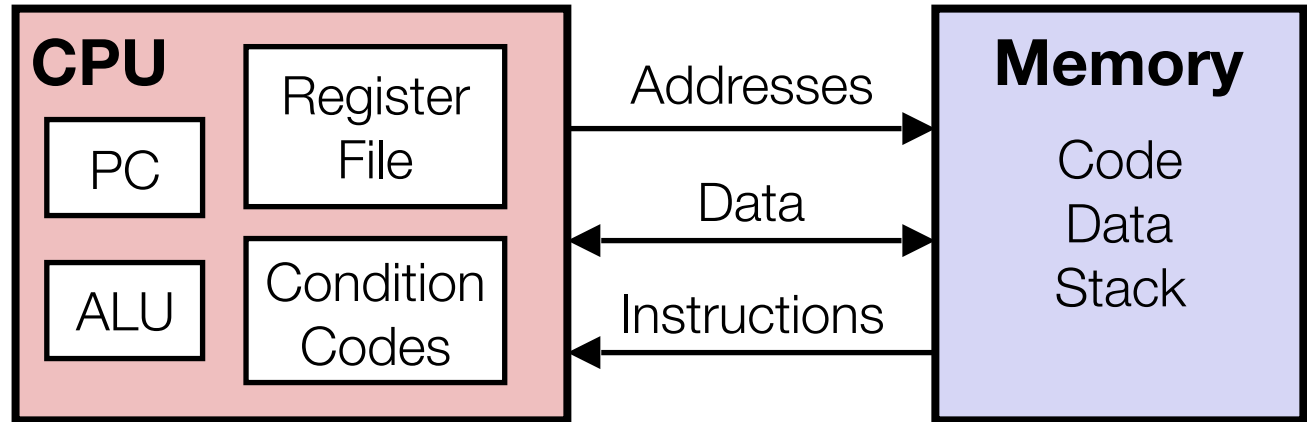
Assembly Programmer's Perspective of a Computer



- Initially all data is in the memory
- But memory is slow: e.g., 15 ns for each access

Data Movement in Processors

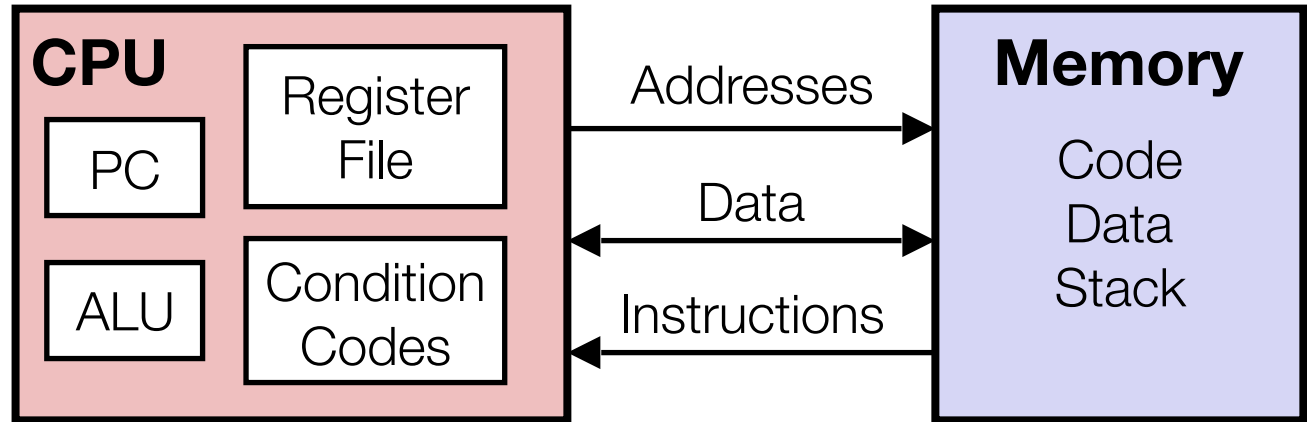
Assembly Programmer's Perspective of a Computer



- 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

Data Movement in Processors

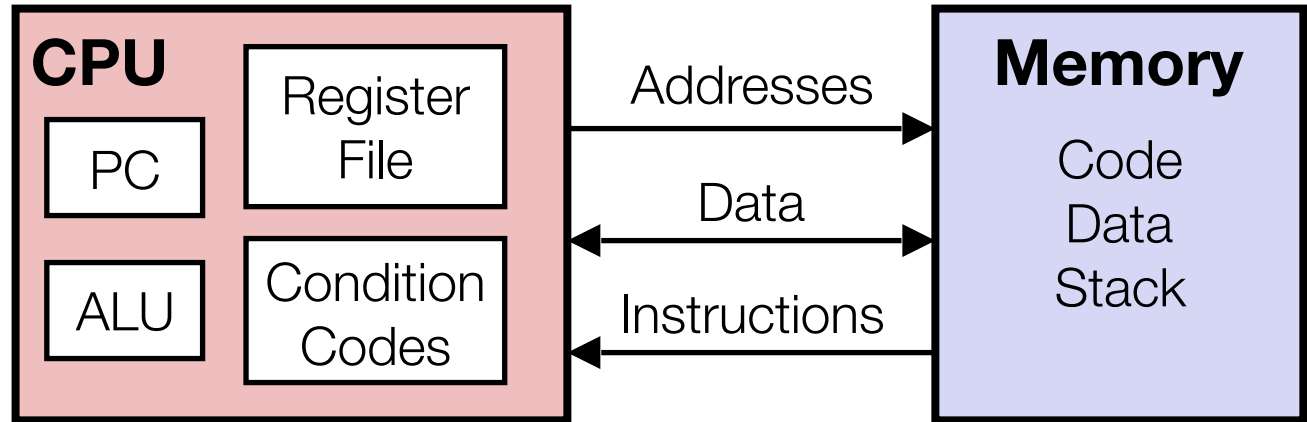
Assembly
Programmer's
Perspective
of a Computer



- 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

Data Movement in Processors

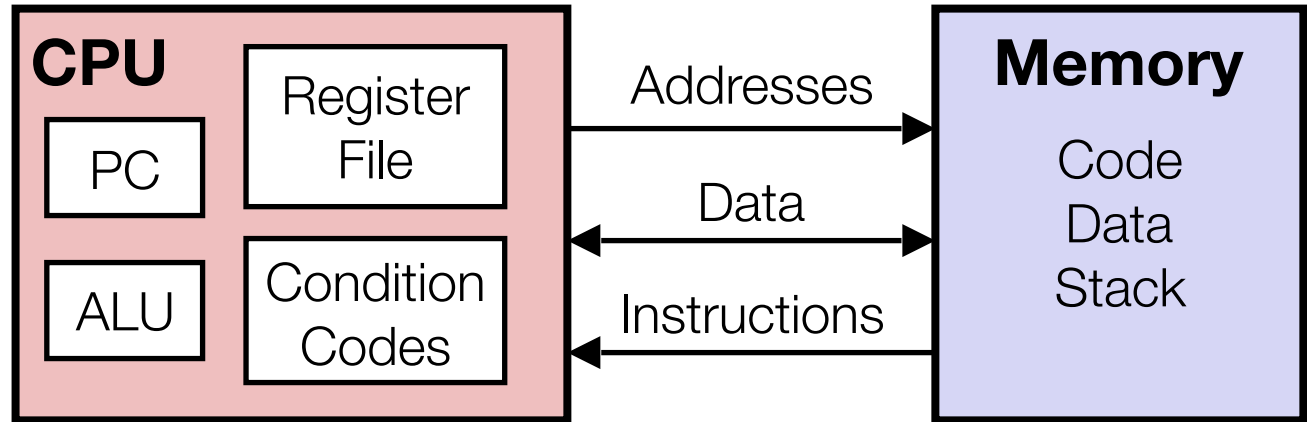
Assembly Programmer's Perspective of a Computer



- 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

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.

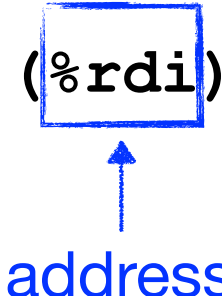
Data Movement Instruction Example

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

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

Data Movement Instruction Example

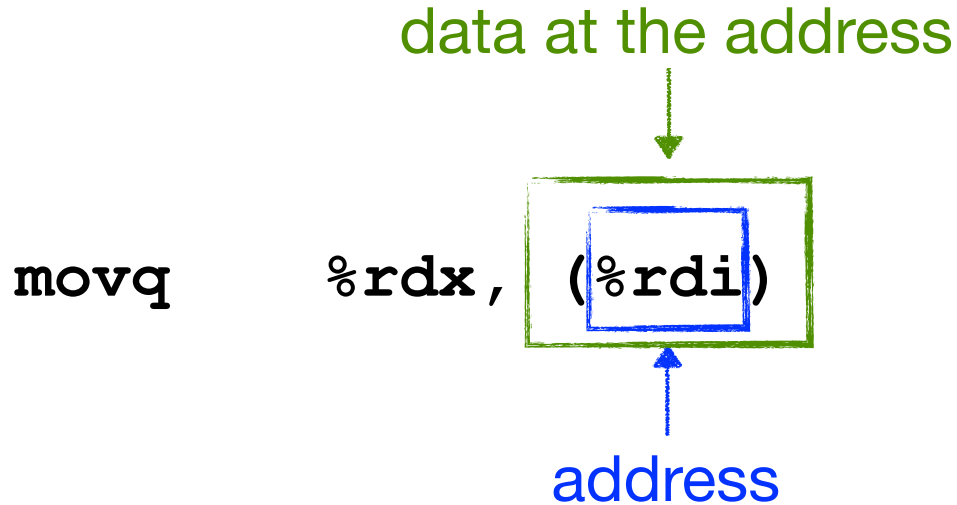
`movq %rdx, (%rdi)`



address

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

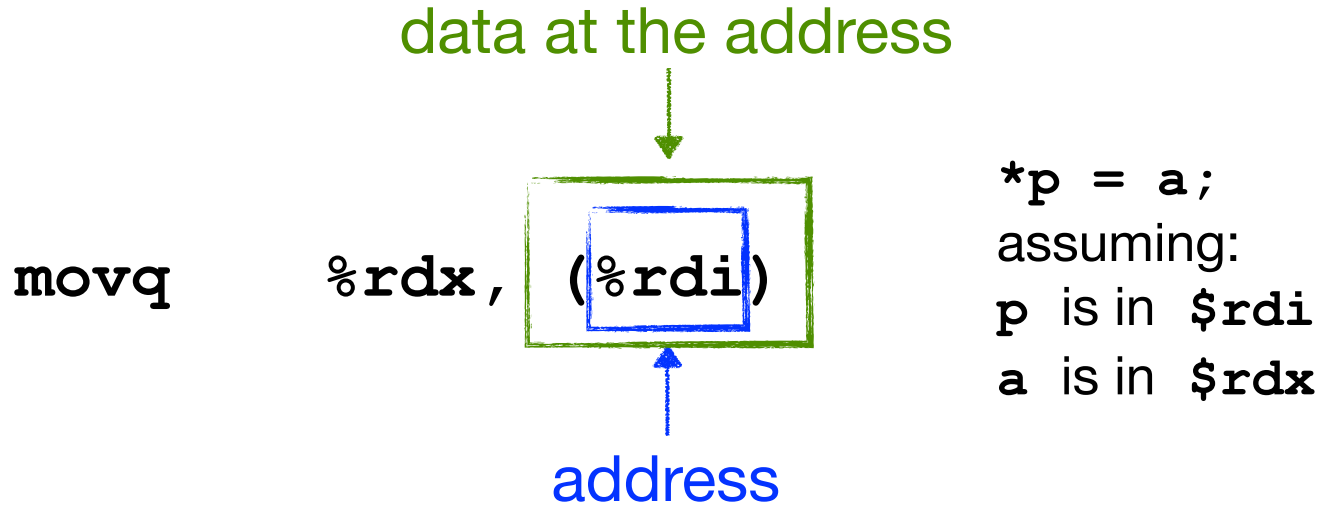
Data Movement Instruction Example



- Semantics:

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

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