

CSC 252/452: Computer Organization

Fall 2024: Lecture 5

Instructor: Yanan Guo

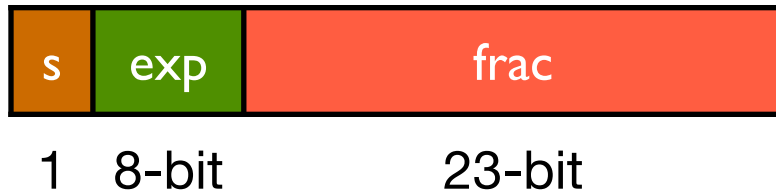
Department of Computer Science
University of Rochester

Announcements

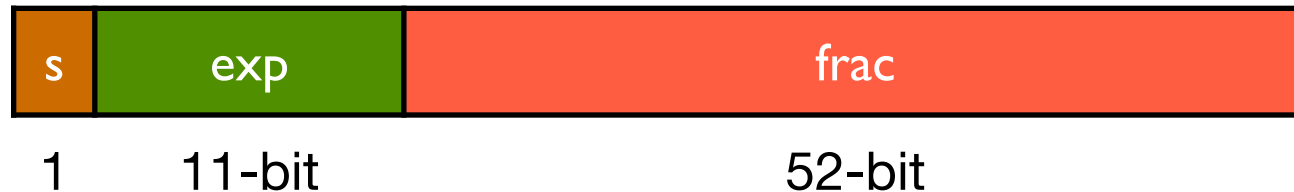
- Programming Assignment 1 is due on Monday!
 - Details:
<https://www.cs.rochester.edu/courses/252/spring2021/labs/assignment1.html>
 - Due on Sep. 16th, 11:59 PM
 - You have 3 slip days

IEEE 754 Floating Point Standard

- Single precision: 32 bits



- Double precision: 64 bits



- In C language

- `float` single precision
- `double` double precision

Floating Point in C

32-bit Machine

Fixed point
(implicit binary point)

SP floating point

DP floating point



C Data Type	Bits	Max Value	Max Value (Decimal)
char	8	$2^7 - 1$	127
short	16	$2^{15} - 1$	32767
int	32	$2^{31} - 1$	2147483647
long	64	$2^{63} - 1$	$\sim 9.2 \times 10^{18}$
float	32	$(2 - 2^{-23}) \times 2^{127}$	$\sim 3.4 \times 10^{38}$
double	64	$(2 - 2^{-52}) \times 2^{1023}$	$\sim 1.8 \times 10^{308}$

Floating Point in C

32-bit Machine

Fixed point
(implicit binary point)

SP floating point

DP floating point



C Data Type	Bits	Max Value	Max Value (Decimal)
char	8	$2^7 - 1$	127
short	16	$2^{15} - 1$	32767
int	32	$2^{31} - 1$	2147483647
long	64	$2^{63} - 1$	$\sim 9.2 \times 10^{18}$
float	32	$(2 - 2^{-23}) \times 2^{127}$	$\sim 3.4 \times 10^{38}$
double	64	$(2 - 2^{-52}) \times 2^{1023}$	$\sim 1.8 \times 10^{308}$

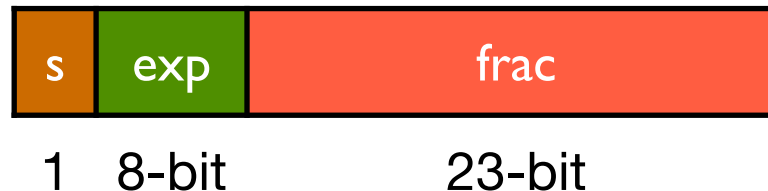
- To represent 2^{31} in fixed-point, you need at least 32 bits
 - Because fixed-point is a *weighted positional* representation
- In floating-point, we directly encode the exponent
 - Floating point is based on scientific notation
 - Encoding 31 only needs 7 bits in the *exp* field

Floating Point Conversions/Casting in C

- **double/float → int**
 - Truncates fractional part
 - Like rounding toward zero
 - Not defined when out of range or NaN

Floating Point Conversions/Casting in C

- **double/float** \rightarrow **int**
 - Truncates fractional part
 - Like rounding toward zero
 - Not defined when out of range or NaN
- **int** \rightarrow **float**



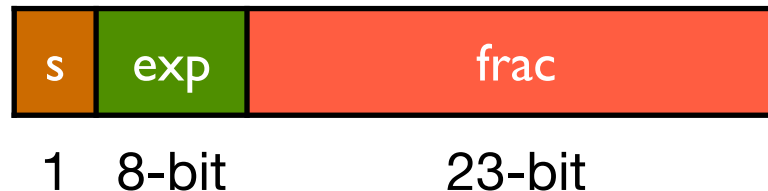
Floating Point Conversions/Casting in C

- **double/float** → **int**

- Truncates fractional part
- Like rounding toward zero
- Not defined when out of range or NaN

- **int** → **float**

- Can't guarantee exact casting. Will round according to rounding mode



Floating Point Conversions/Casting in C

- **double/float** → **int**

- Truncates fractional part
- Like rounding toward zero
- Not defined when out of range or NaN

- **int** → **float**

- Can't guarantee exact casting. Will round according to rounding mode



1 8-bit

23-bit

- **int** → **double**



1 11-bit

52-bit

Floating Point Conversions/Casting in C

- **double/float** → **int**

- Truncates fractional part
- Like rounding toward zero
- Not defined when out of range or NaN

- **int** → **float**

- Can't guarantee exact casting. Will round according to rounding mode



1 8-bit

23-bit

- **int** → **double**

- Exact conversion



1

11-bit

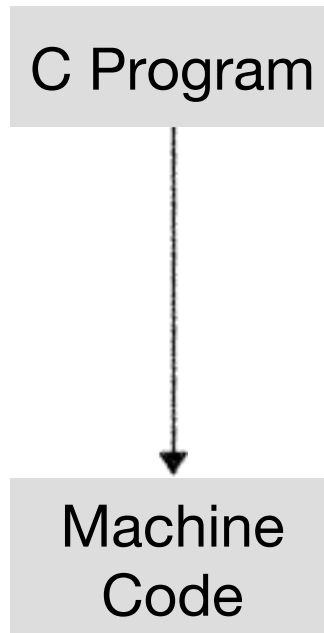
52-bit

So far in 252...

C Program

int, float
if, else
+, -, >>

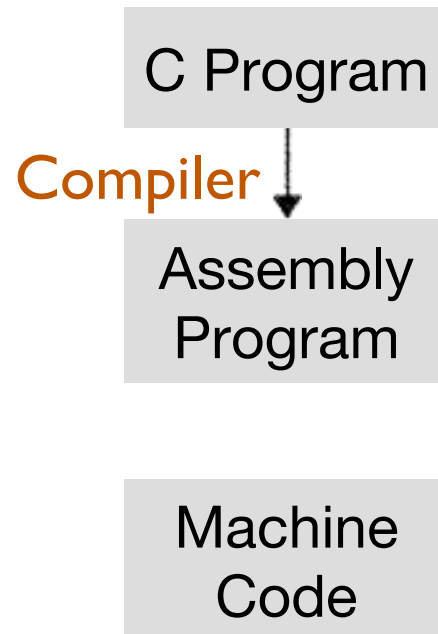
So far in 252...



int, float
if, else
+, -, >>

00001111
01010101
11110000

So far in 252...

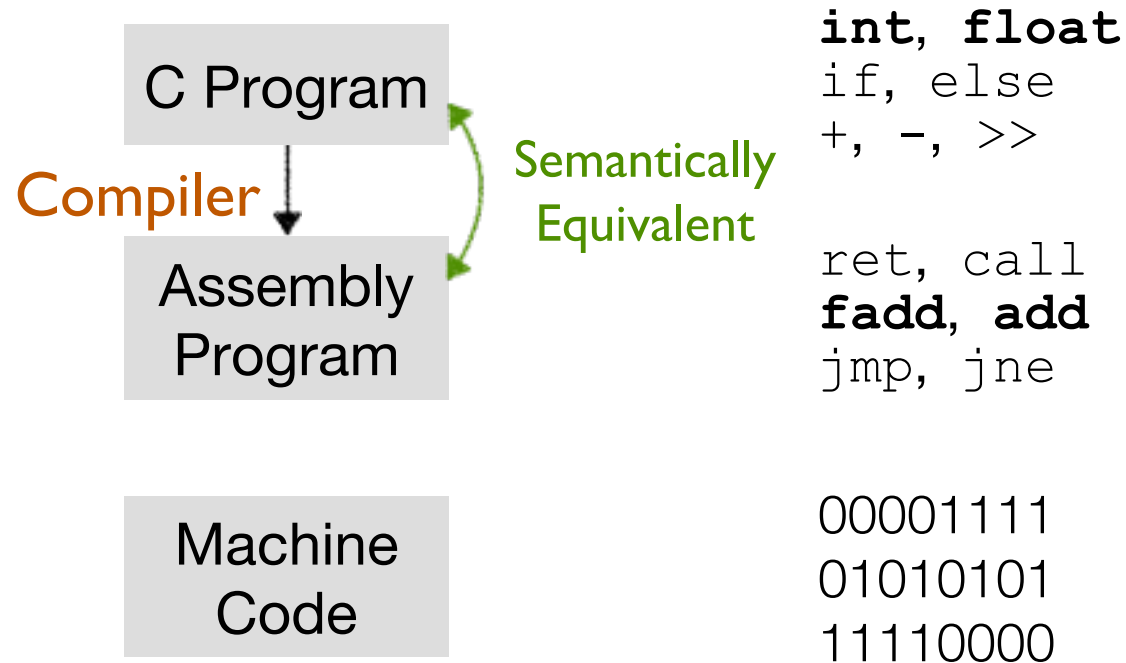


int, float
if, else
+, -, >>

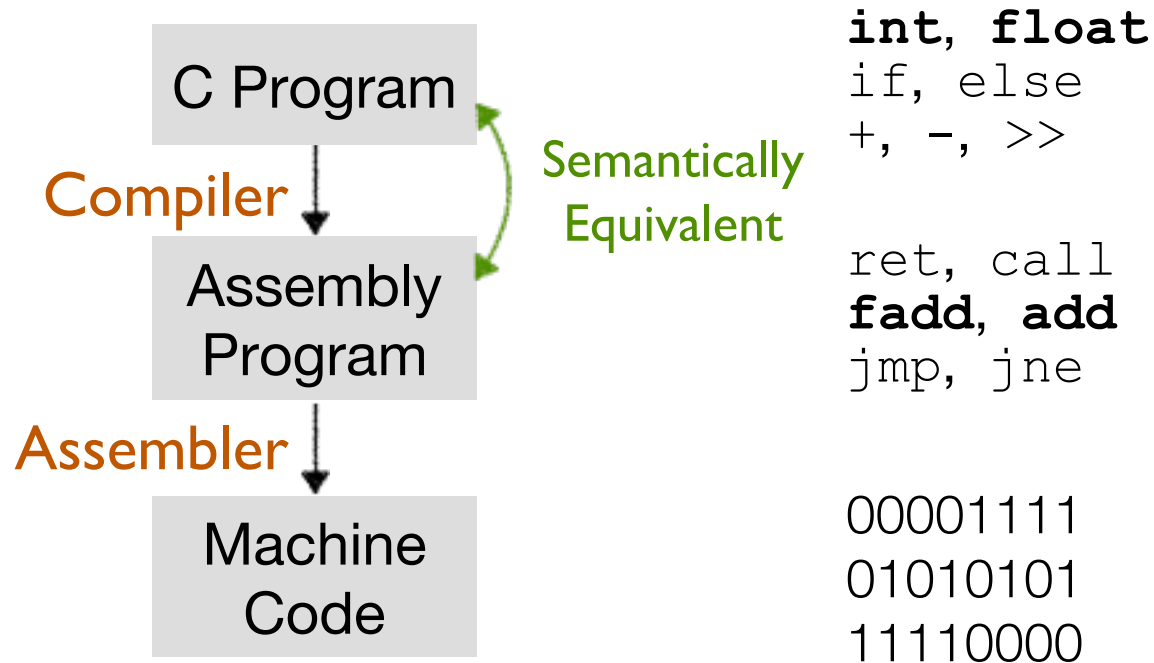
ret, call
fadd, add
jmp, jne

00001111
01010101
11110000

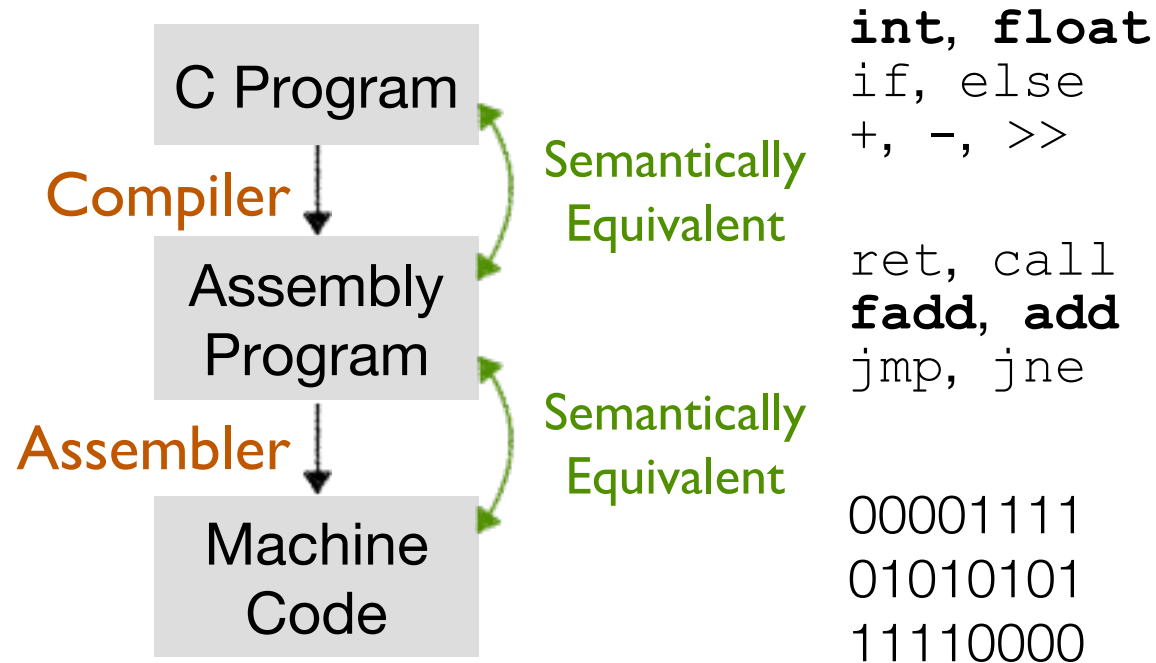
So far in 252...



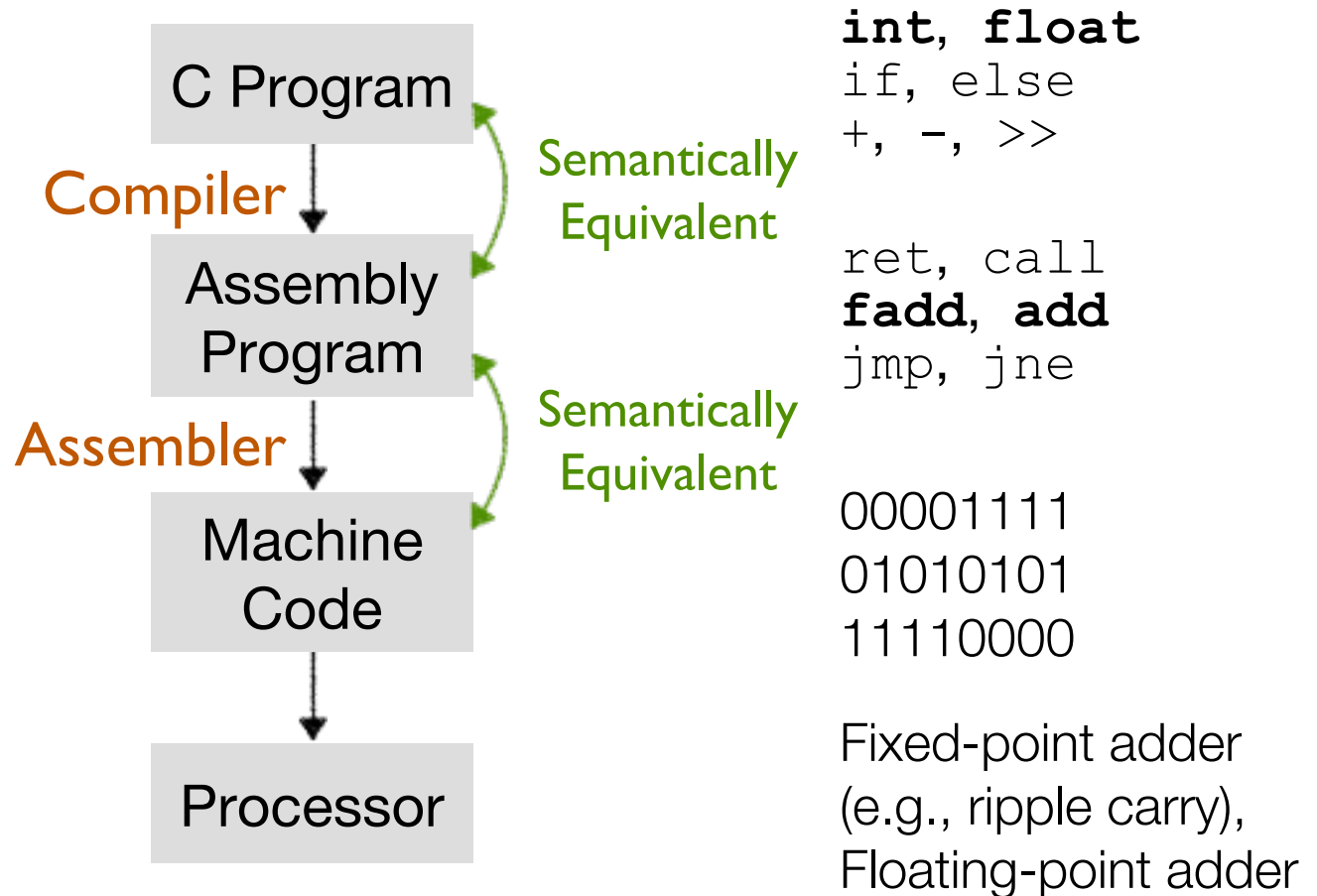
So far in 252...



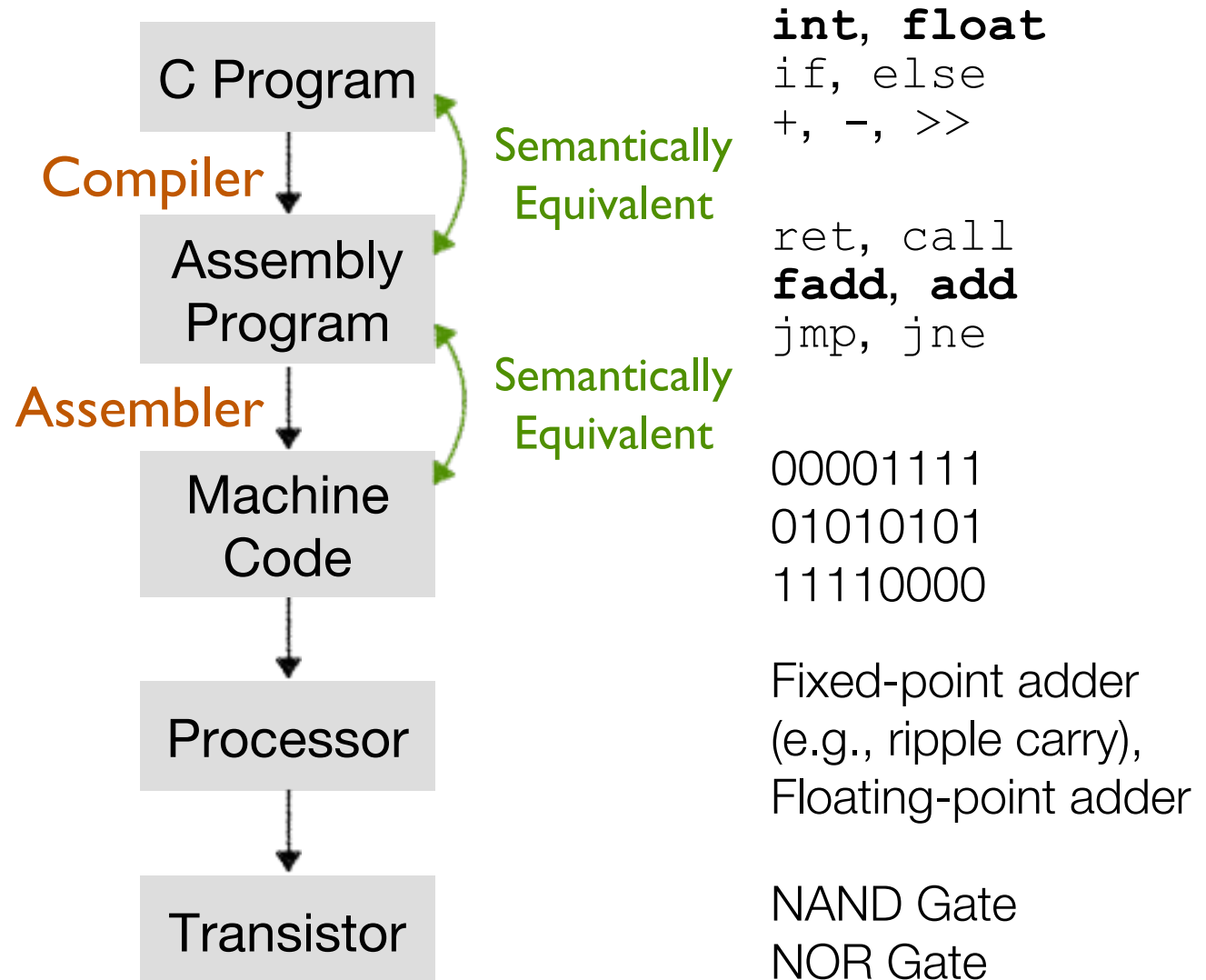
So far in 252...



So far in 252...

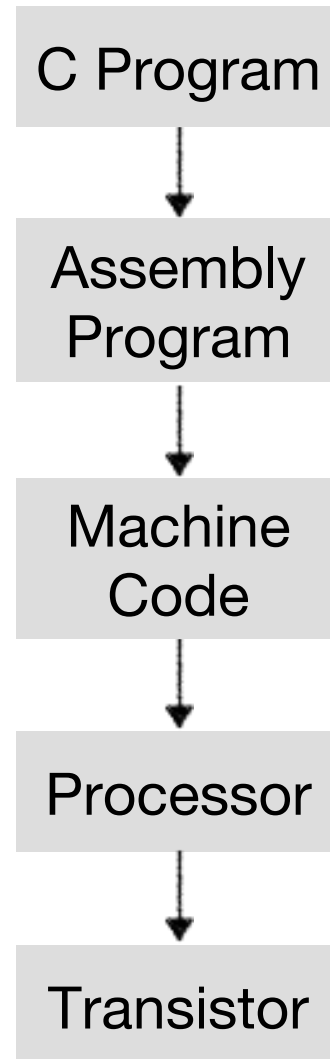


So far in 252...



So far in 252...

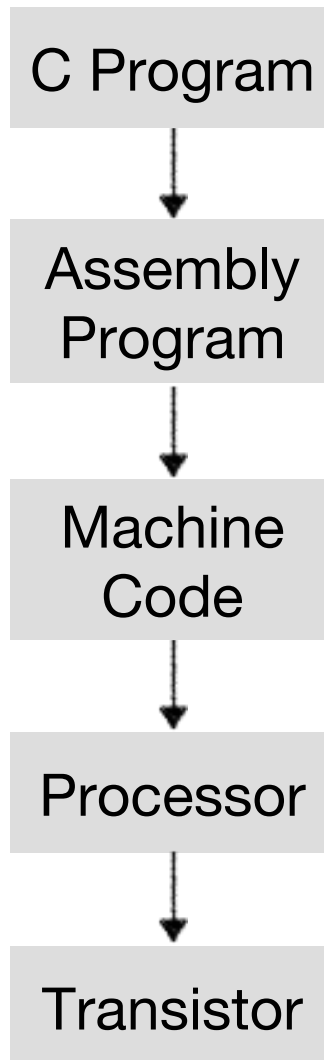
**High-Level
Language**



So far in 252...

High-Level
Language

Instruction Set
Architecture
(ISA)

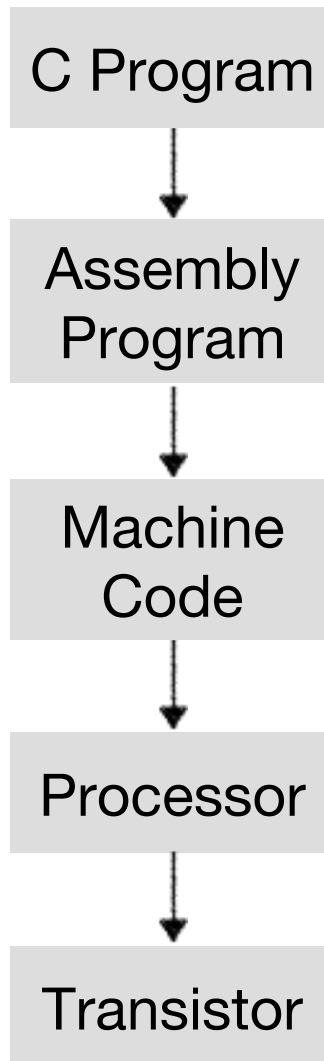


- **ISA:** Software programmers' view of a computer
 - Provide all info for someone wants to write assembly/machine code
 - “Contract” between assembly/machine code and processor

So far in 252...

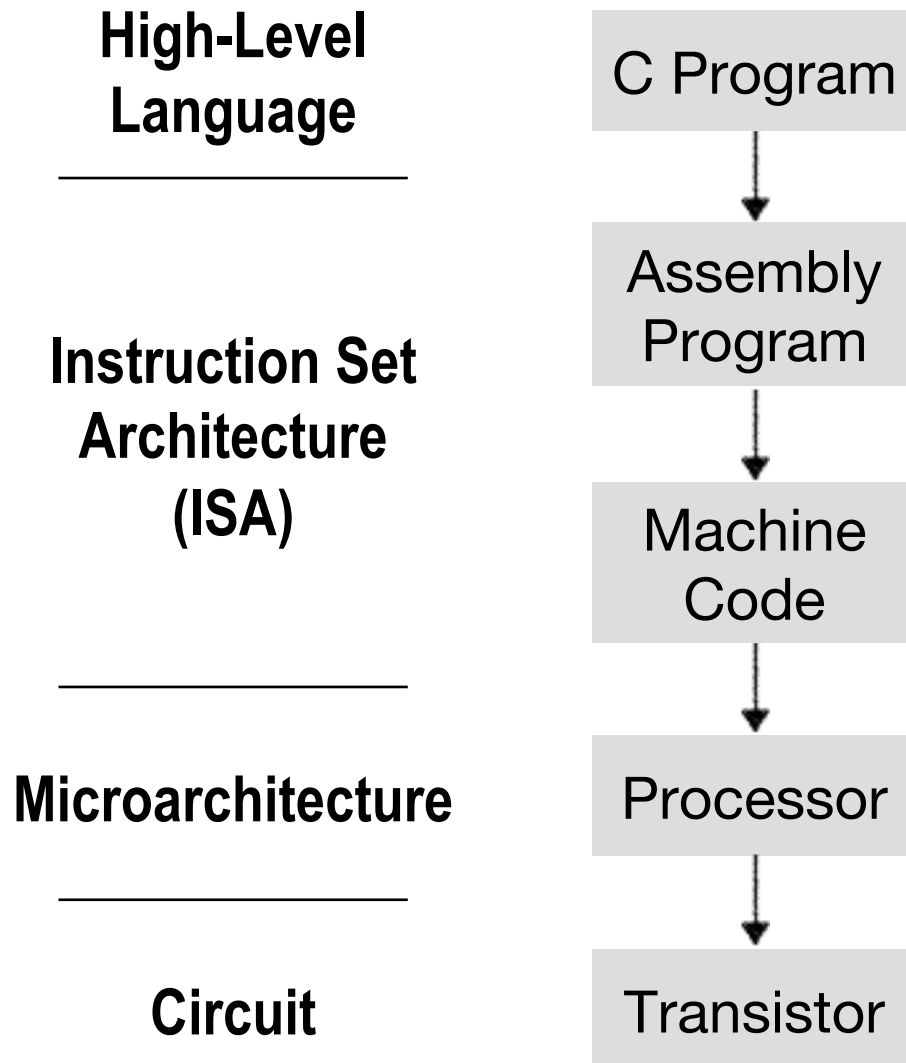
High-Level
Language

Instruction Set
Architecture
(ISA)



- **ISA:** Software programmers' view of a computer
 - Provide all info for someone wants to write assembly/machine code
 - “Contract” between assembly/machine code and processor
- Processors execute machine code (binary). Assembly program is merely a text representation of machine code

So far in 252...



- **ISA:** Software programmers' view of a computer
 - Provide all info for someone wants to write assembly/machine code
 - “Contract” between assembly/machine code and processor
- Processors execute machine code (binary). Assembly program is merely a text representation of machine code
- **Microarchitecture:** Hardware implementation of the ISA (with the help of circuit technologies)

This Module (4 Lectures)

**High-Level
Language**

**Instruction Set
Architecture
(ISA)**

Microarchitecture

Circuit

C Program



**Assembly
Program**



Machine
Code



Processor



Transistor

- **Assembly Programming**
 - Explain how various C constructs are implemented in assembly code
 - Effectively translating from C to assembly program manually
 - Helps us understand how compilers work
 - Helps us understand how assemblers work
- **Microarchitecture is the topic of the next module**

Today: Assembly Programming I: Basics

- Different ISAs and history behind them
- C, assembly, machine code
- Move operations (and addressing modes)

Instruction Set Architecture

- There used to be many ISAs
 - x86, ARM, Power/PowerPC, Sparc, MIPS, IA64, z
 - Very consolidated today: ARM and x86

Instruction Set Architecture

- There used to be many ISAs
 - x86, ARM, Power/PowerPC, Sparc, MIPS, IA64, z
 - Very consolidated today: ARM and x86
- There are even more microarchitectures
 - Apple/Samsung/Qualcomm have their own microarchitecture (implementation) of the ARM ISA
 - Intel and AMD have different microarchitectures for x86

Instruction Set Architecture

- There used to be many ISAs
 - x86, ARM, Power/PowerPC, Sparc, MIPS, IA64, z
 - Very consolidated today: ARM and x86
- There are even more microarchitectures
 - Apple/Samsung/Qualcomm have their own microarchitecture (implementation) of the ARM ISA
 - Intel and AMD have different microarchitectures for x86
- ISA is lucrative business: ARM's Business Model
 - Patent the ISA, and then license the ISA
 - Every implementer pays a royalty to ARM
 - Apple/Samsung pays ARM whenever they sell a smartphone

The ARM Diaries, Part 1: How ARM's Business Model Works:

<https://www.anandtech.com/show/7112/the-arm-diaries-part-1-how-arms-business-model-works>

Intel x86 ISA

- Dominate laptop/desktop/cloud market

Intel x86 ISA

- Dominate laptop/desktop/cloud market



Intel x86 ISA

- Dominate laptop/desktop/cloud market



Intel x86 ISA Evolution (Milestones)

- Evolutionary design: Added more features as time goes on

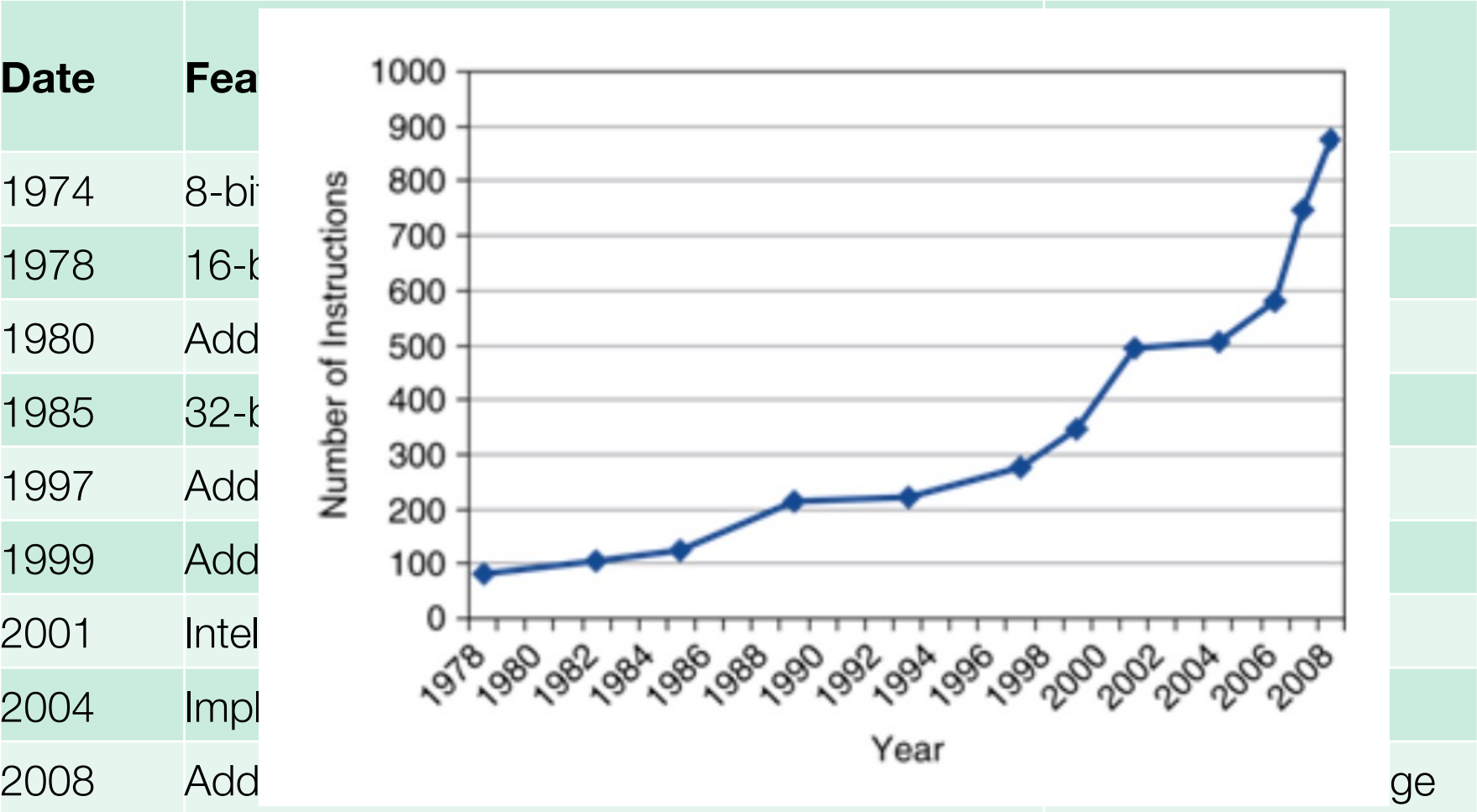
Intel x86 ISA Evolution (Milestones)

- 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

Intel x86 ISA Evolution (Milestones)

- Evolutionary design: Added more features as time goes on



Backward Compatibility

- Binary executable generated for an older processor can execute on a newer processor
- Allows legacy code to be executed on newer machines
 - Buy new machines without changing the software
- **x86 is backward compatible up until 8086 (16-bit ISA)**
 - i.e., an 8086 binary executable can be executed on any of today's x86 machines
- **Great for users, nasty for processor implementers**
 - Every instruction you put into the ISA, you are stuck with it *FOREVER*

x86 Clones: Advanced Micro Devices (AMD)



- **Historically**
 - AMD build processors for x86 ISA
 - A little bit slower, a lot cheaper
- **Then**
 - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
 - Developed x86-64, their own 64-bit x86 extension to IA32
 - Built first 1 GHz CPU
- **Intel felt hard to admit mistake or that AMD was better**
- **2004: Intel Announces EM64T extension to IA32**
 - Almost identical to x86-64!
 - Today's 64-bit x86 ISA is basically AMD's original proposal

x86 Clones: Advanced Micro Devices (AMD)

- Today: Holding up not too badly

Market Summary > Advanced Micro Devices, Inc.

134.35 USD

+ Follow

+103.79 (339.63%) ↑ past 5 years

Closed: Sep 6, 7:59 PM EDT • Disclaimer

After hours 134.02 -0.33 (0.25%)

1D | 5D | 1M | 6M | YTD | 1Y | 5Y | Max



Open	138.70	Mkt cap	217.44B	CDP score	B
High	139.13	P/E ratio	161.68	52-wk high	227.30
Low	132.11	Div yield	-	52-wk low	93.12

More about Advanced Micro Device... →

Feedback

x86 Clones: Advanced Micro Devices (AMD)

- Today: Holding up not too badly

Market Summary > Intel Corp

18.89 USD

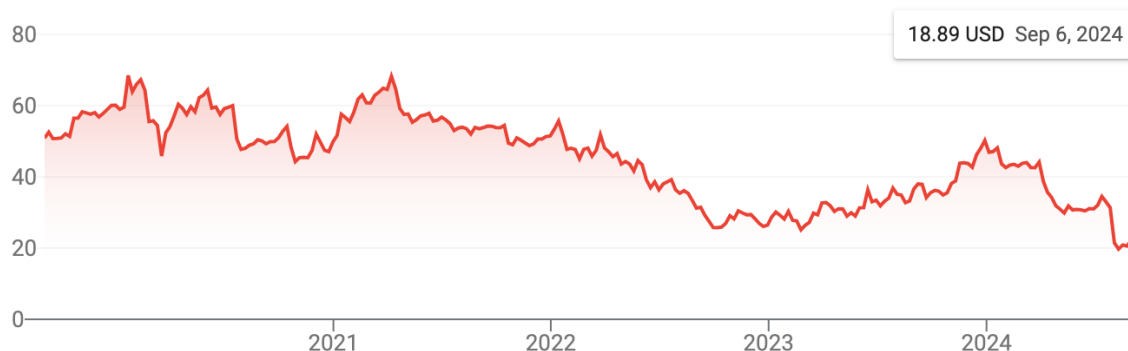
+ Follow

-32.03 (-62.90%) ↓ past 5 years

Closed: Sep 6, 7:59 PM EDT • Disclaimer

After hours 18.83 -0.060 (0.32%)

1D | 5D | 1M | 6M | YTD | 1Y | 5Y | Max



Open	19.44	Mkt cap	80.60B	CDP score	A-
High	19.49	P/E ratio	84.84	52-wk high	51.28
Low	18.64	Div yield	2.65%	52-wk low	18.64

More about Intel Corp →

Feedback

Our Coverage

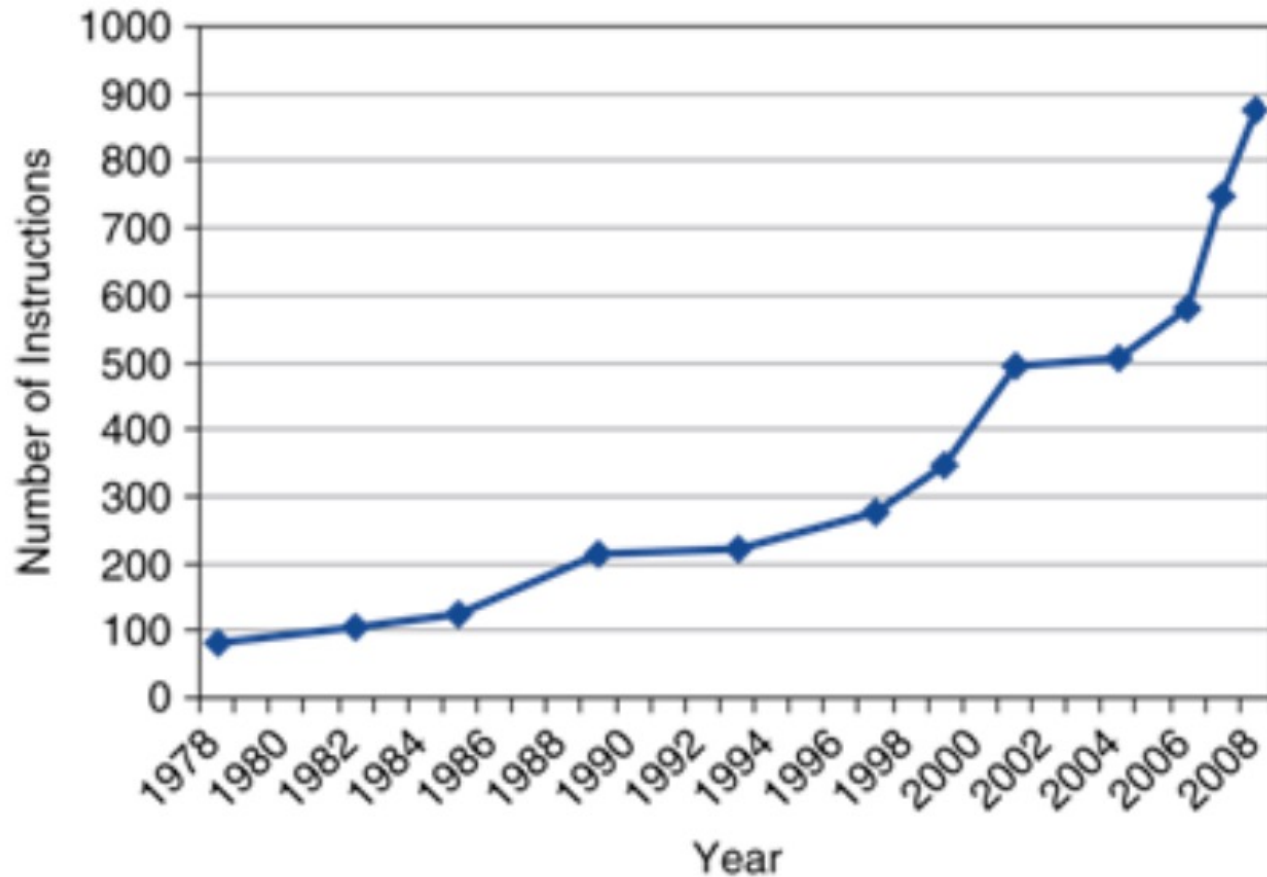
- IA32
 - The traditional x86
 - 2nd edition of the textbook
- x86-64
 - The standard
 - CSUG machine
 - 3rd edition of the textbook
 - Our focus

Moore's Law

- More instructions typically require more transistors to implement

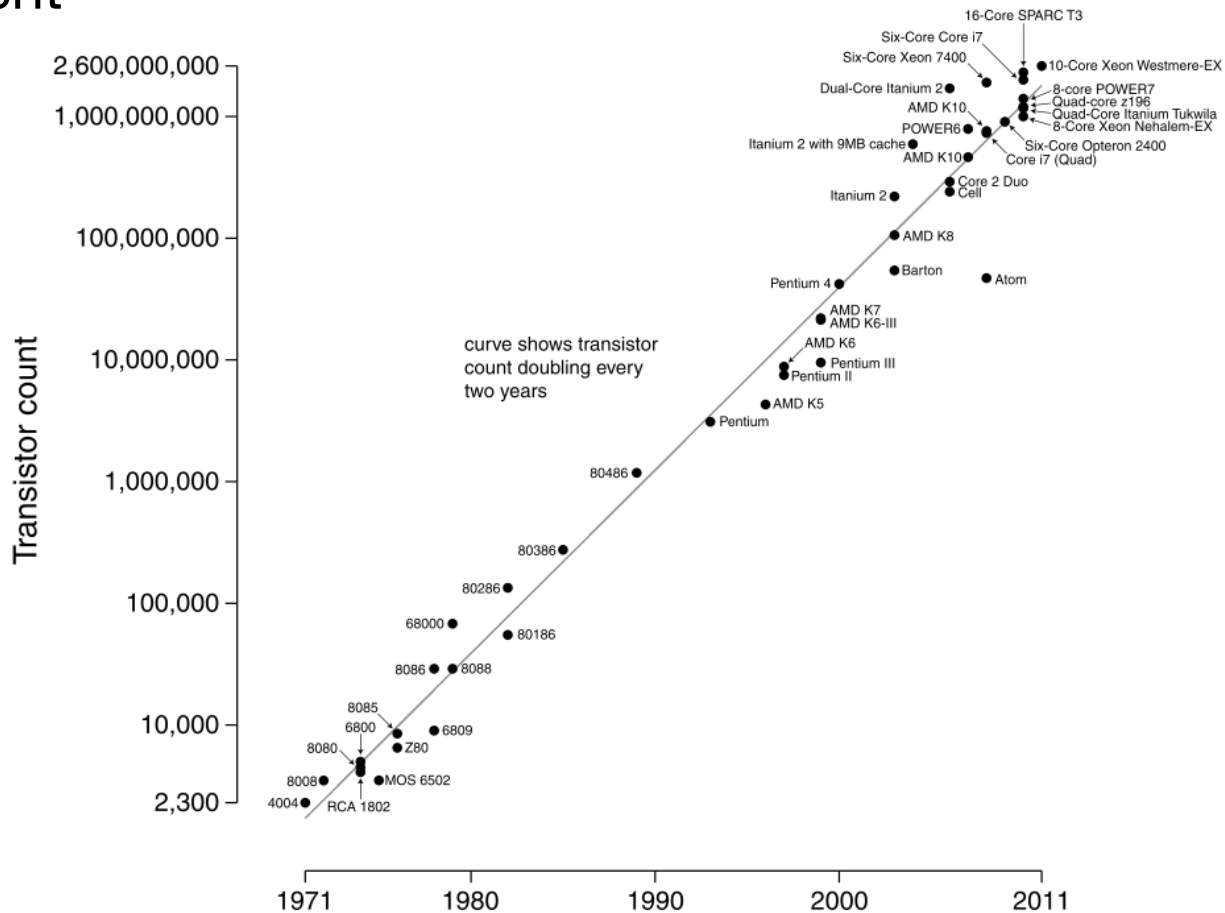
Moore's Law

- More instructions typically require more transistors to implement



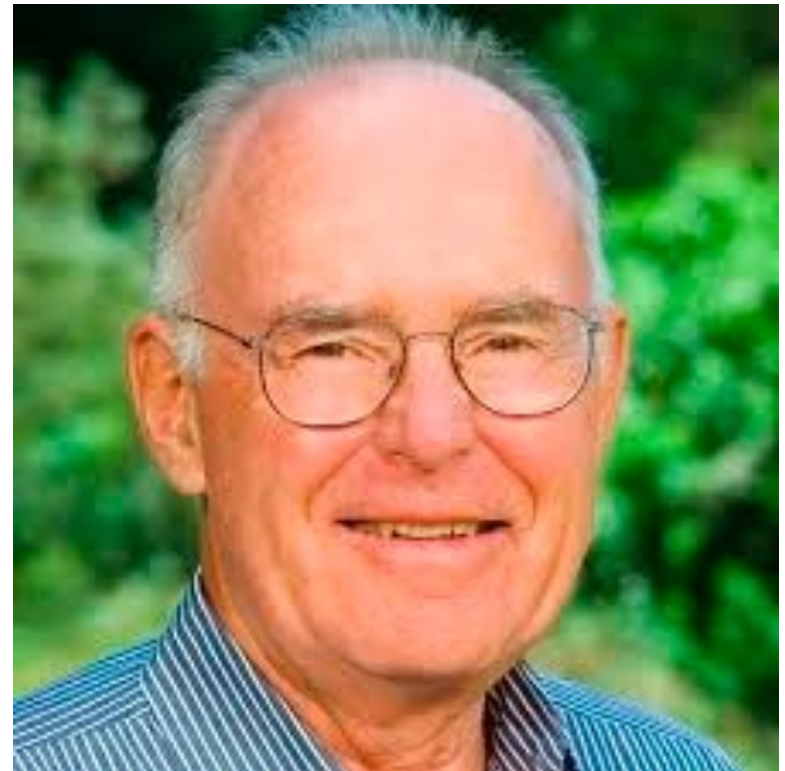
Moore's Law

- More instructions typically require more transistors to implement



Moore's Law

- More instructions require more transistors to implement



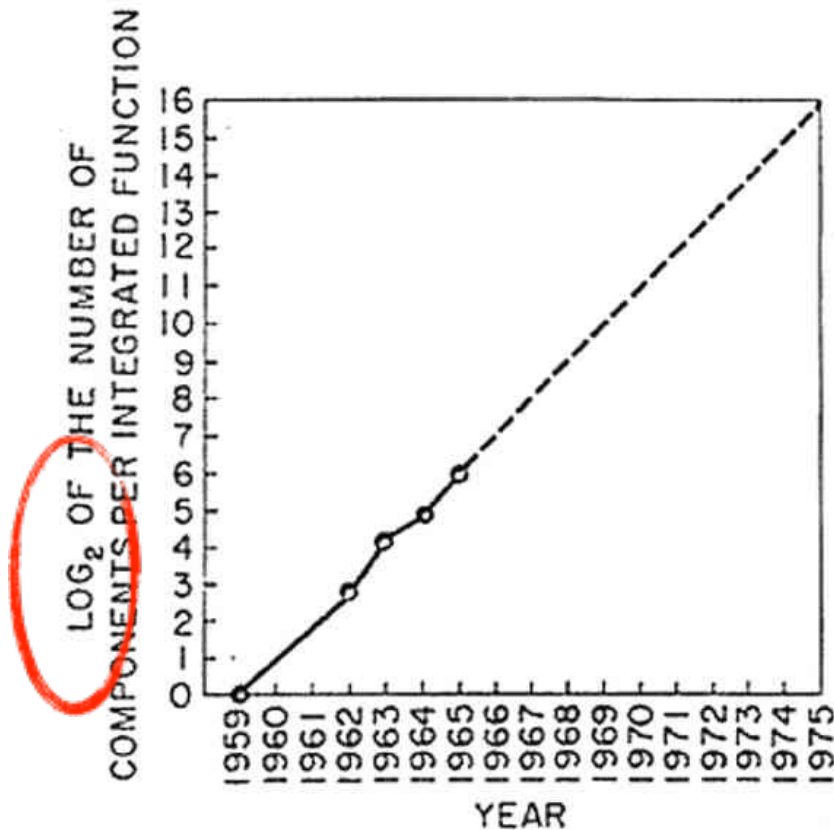
Moore's Law

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



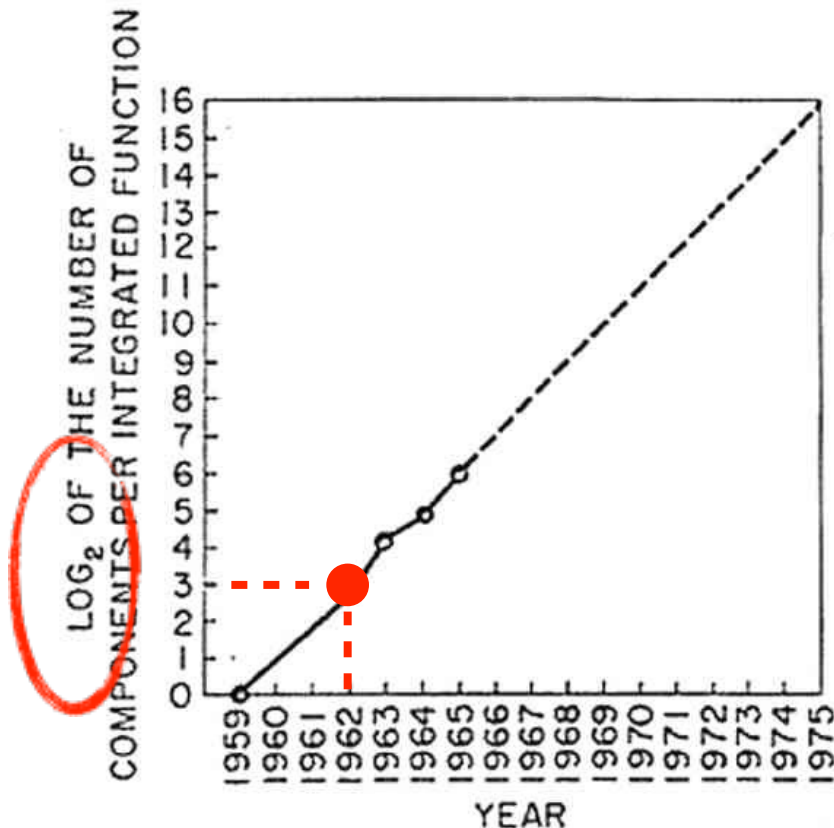
Moore's Law

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



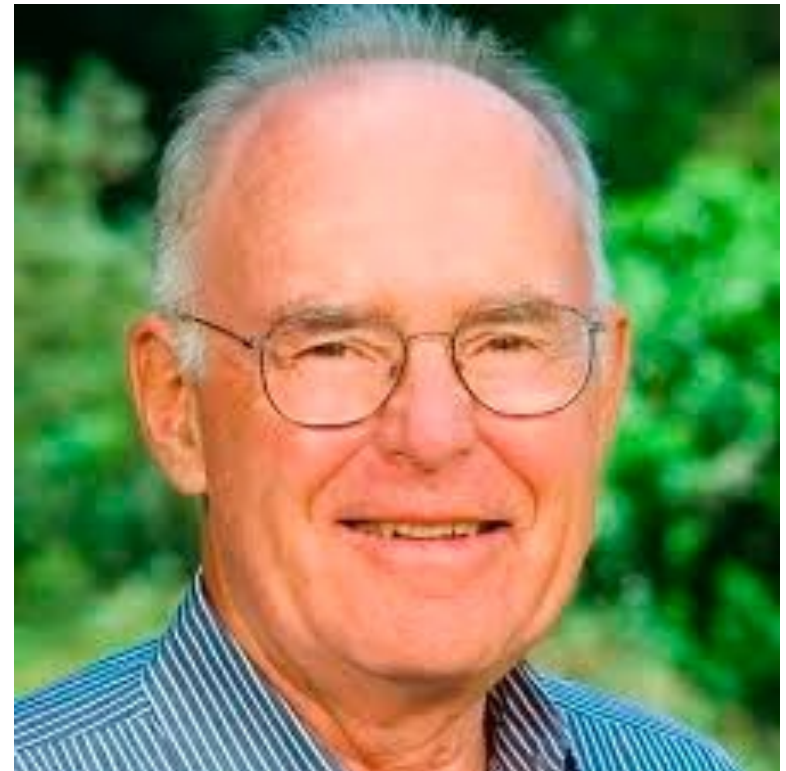
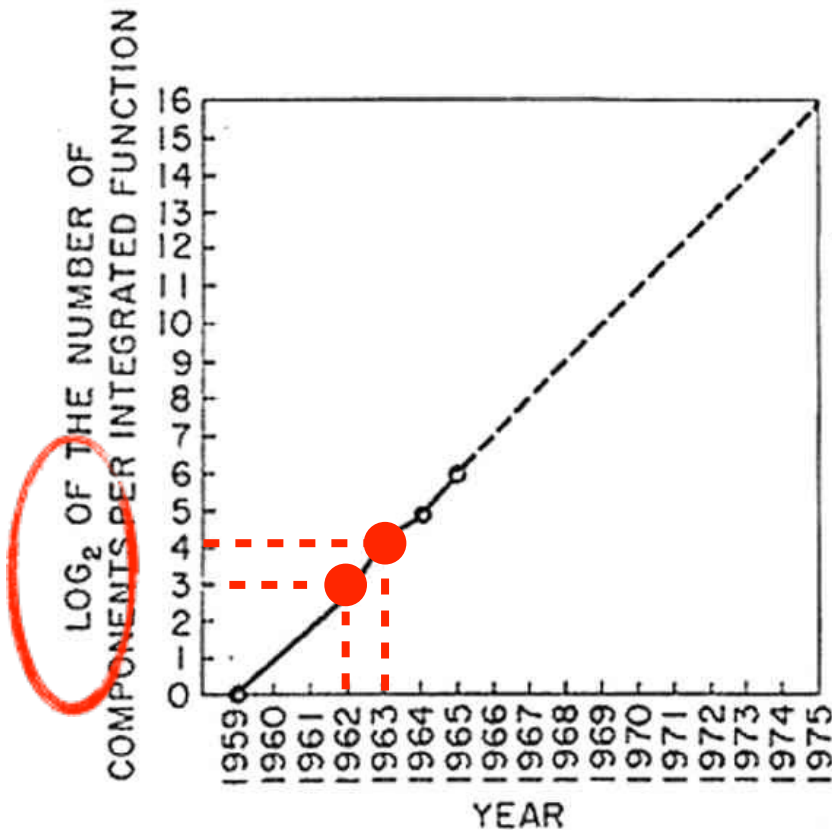
Moore's Law

- 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
 - $\sim 1.4\times$ 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
 - $\sim 1.4\times$ 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
 - $\sim 1.4\times$ 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
 - $\sim 1.4\times$ 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
 - $\sim 1.4\times$ smaller each dimension ($1.4^2 \sim 2$)
- Moore's Law is:
 - A law of physics?
 - A law of circuits?

No

Moore's Law

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

Moore's Law

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

Moore's Law

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

Moore's Law



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



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

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

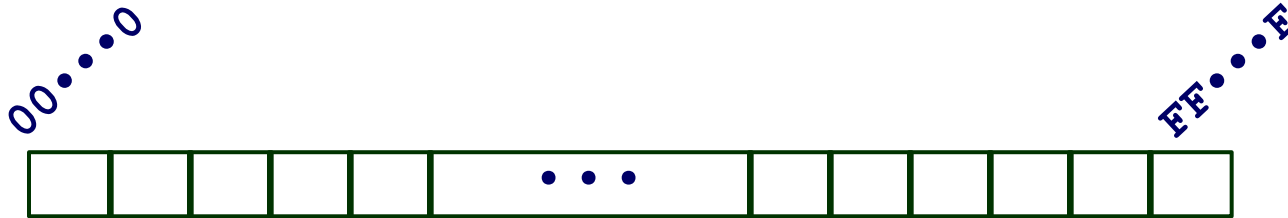
Moore's Law

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

Today: Assembly Programming I: Basics

- Different ISAs and history behind them
- Memory, C, assembly, machine code
- Move operations (and addressing modes)

Byte-Oriented Memory Organization



- Programs refer to data by address
 - Conceptually, envision it as a very large array of bytes: **byte-addressable**
 - An address is like an index into that array
 - and, a pointer variable 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);
```

Memory
Content

Memory
Address



0x10

0x11

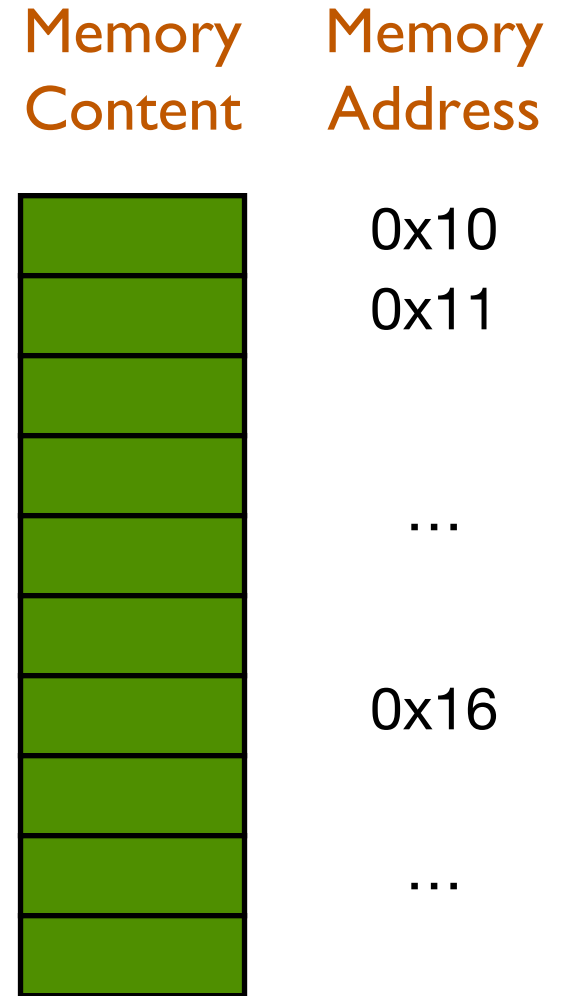
...

0x16

...

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);`

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

How Does Pointer Work in C???

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

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

How Does Pointer Work in C???

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

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);`

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.

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.

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.

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);`

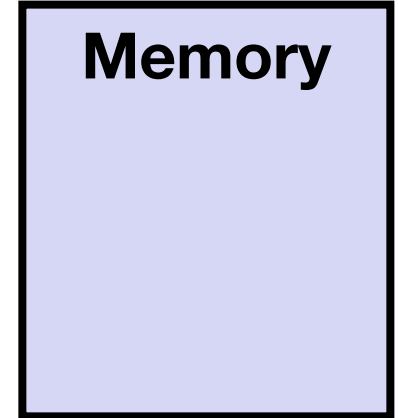
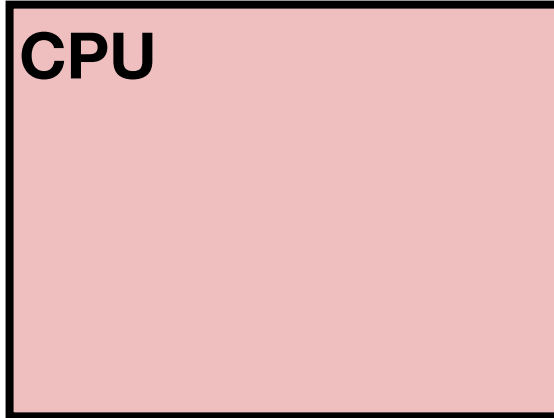
- The content of a pointer variable is memory address.
- The '`&`' operator (address-of operator) returns the memory address of a variable.
- The '`*`' operator returns the content stored at the memory location pointed by the pointer variable (dereferencing)

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

Assembly Code's View of Computer

Assembly Code's View of Computer

Assembly
Programmer's
Perspective
of a Computer



Assembly Code's View of Computer

Assembly
Programmer's
Perspective
of a Computer

CPU



Memory

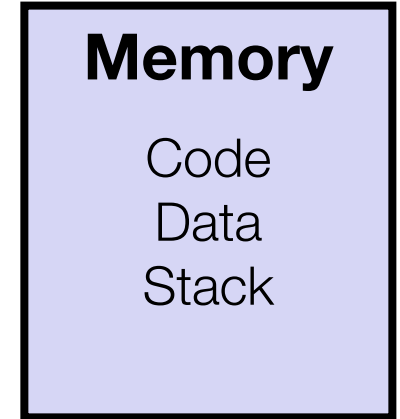
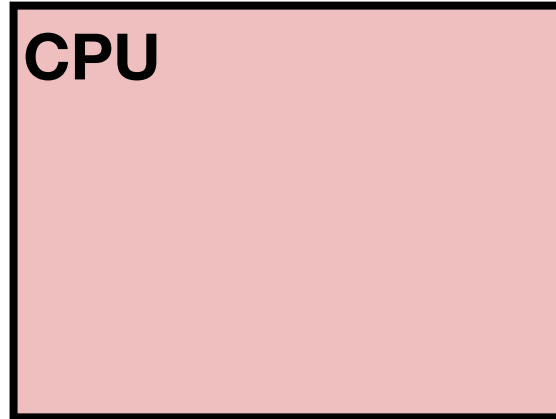
Code
Data
Stack



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

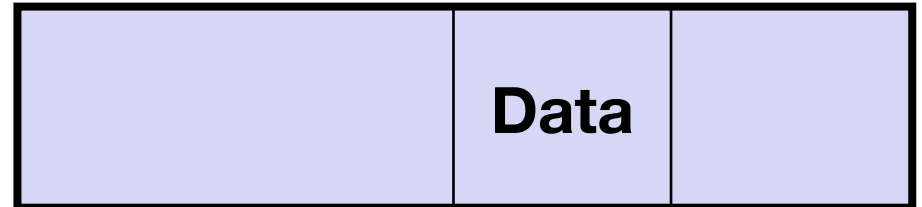
Assembly Code's View of Computer

Assembly
Programmer's
Perspective
of a Computer



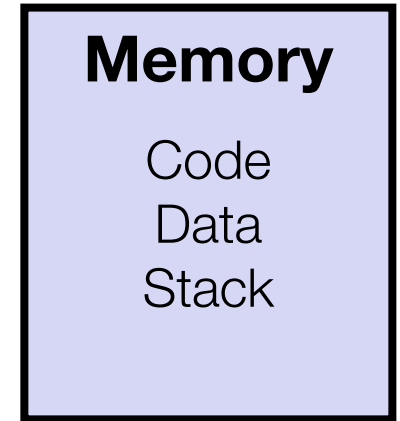
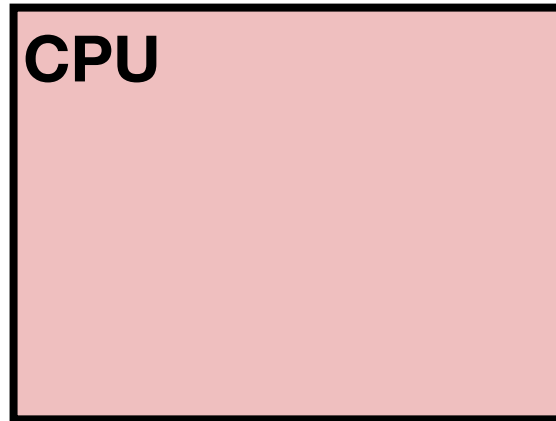
- (Byte Addressable) Memory

- Code: instructions
- Data
- Stack to support function call



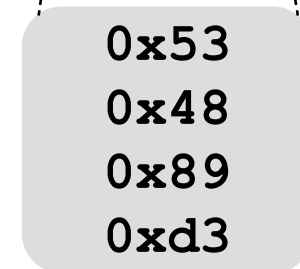
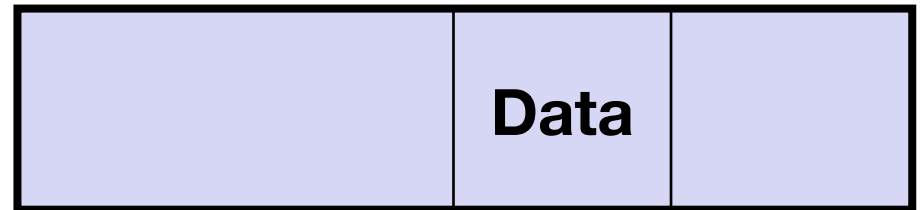
Assembly Code's View of Computer

Assembly
Programmer's
Perspective
of a Computer



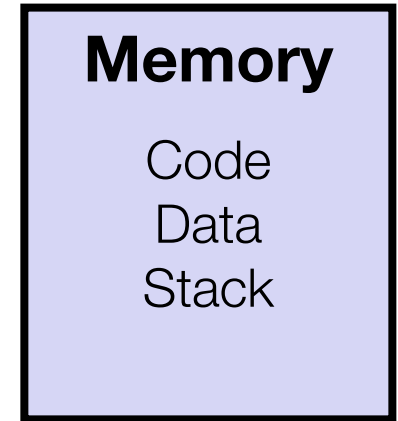
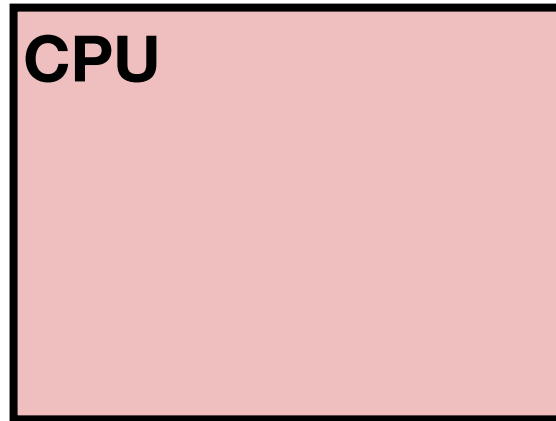
- (Byte Addressable) Memory

- Code: instructions
- Data
- Stack to support function call



Assembly Code's View of Computer

Assembly
Programmer's
Perspective
of a Computer

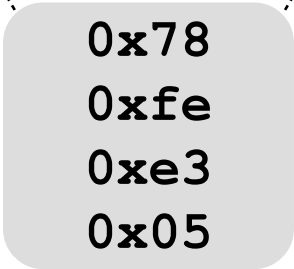
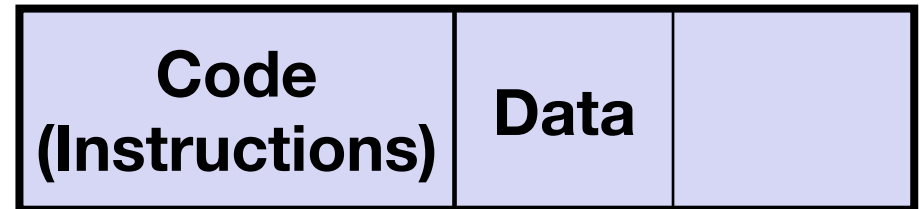


- (Byte Addressable) Memory

- Code: instructions
- Data
- Stack to support function call

Instruction is the fundamental
unit of work.

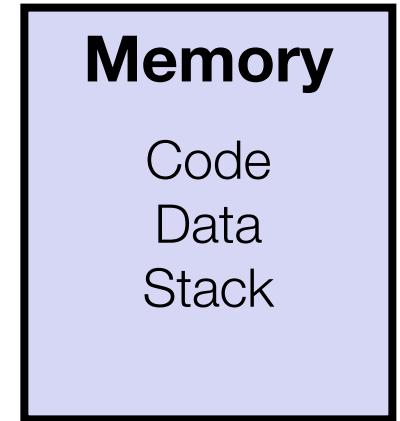
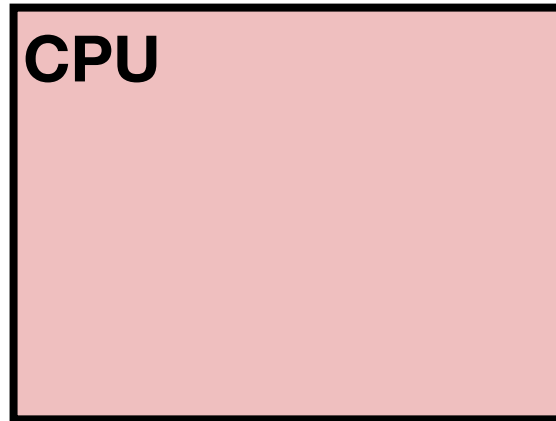
All instructions are encoded as
bits (just like data!)



A list of four hexadecimal values: **0x78**, **0xfe**, **0xe3**, and **0x05**. Dotted lines connect the first and second sections of the memory layout diagram above to the first and second values in this list.

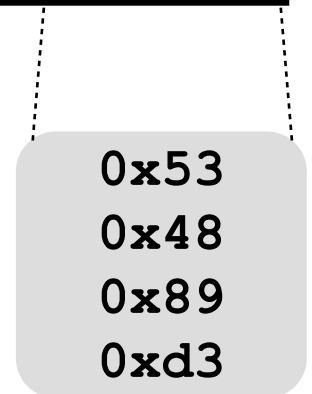
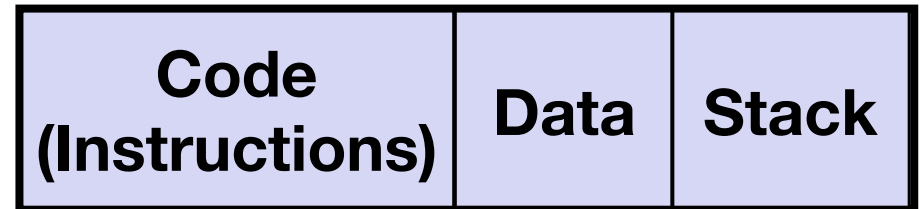
Assembly Code's View of Computer

Assembly
Programmer's
Perspective
of a Computer



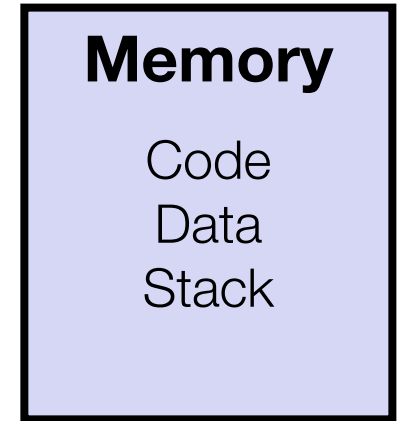
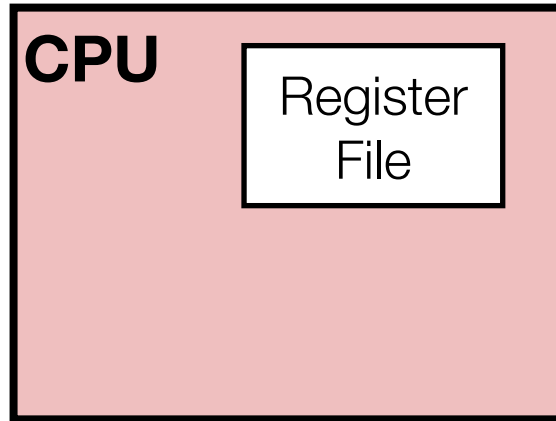
- (Byte Addressable) Memory

- Code: instructions
- Data
- Stack to support function call



Assembly Code's View of Computer

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 →

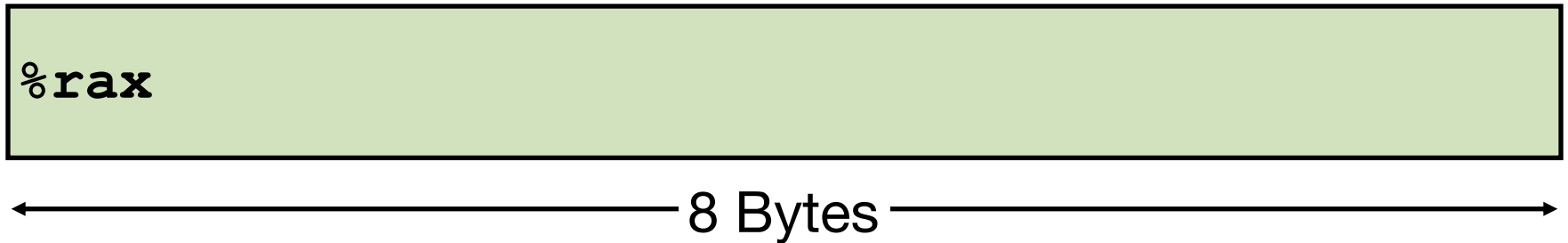
<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

x86-64 Integer Register File

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

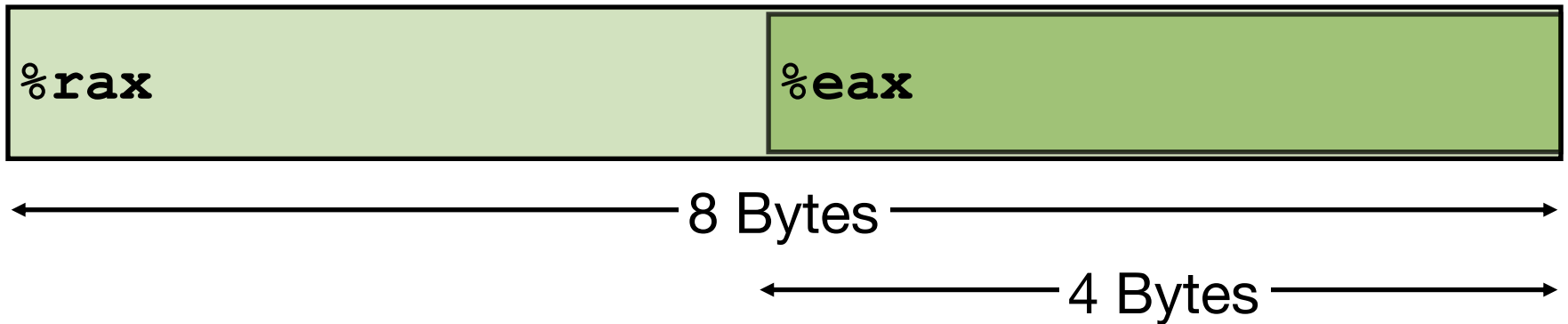
x86-64 Integer Register File

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



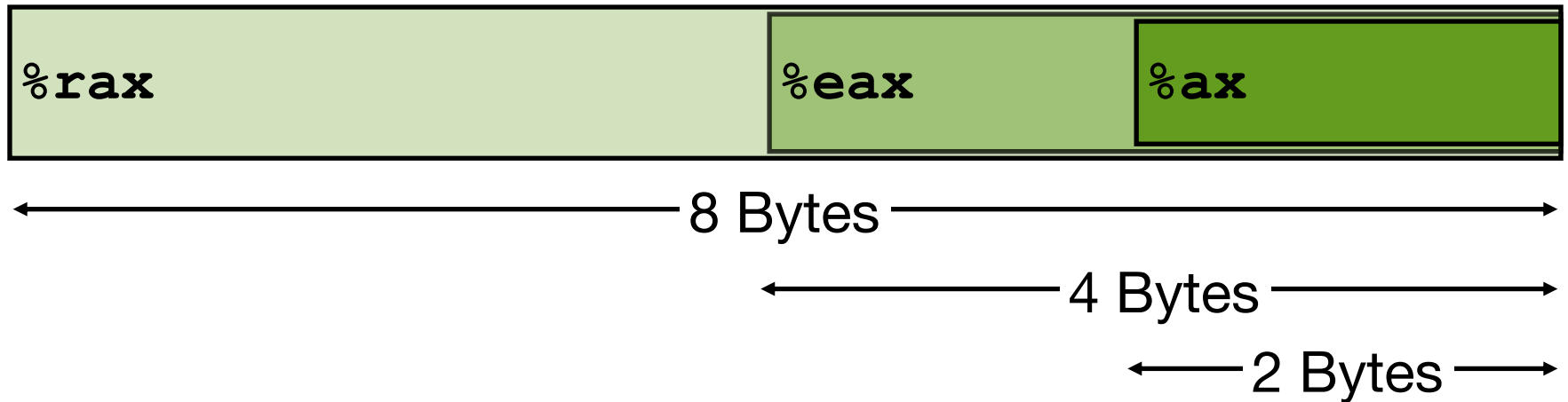
x86-64 Integer Register File

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



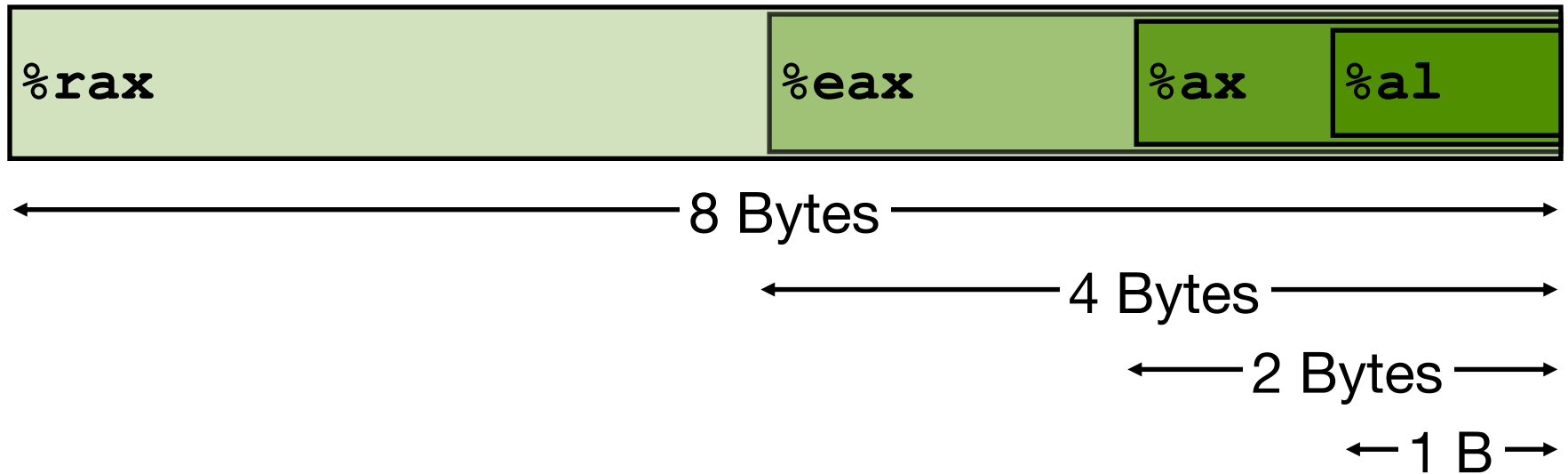
x86-64 Integer Register File

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



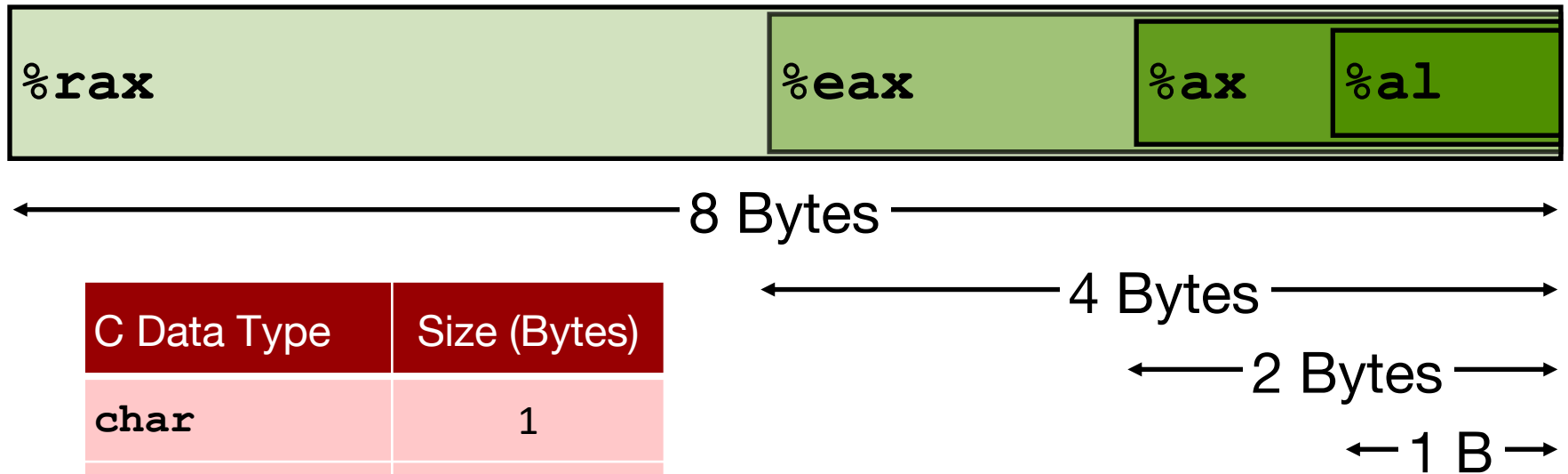
x86-64 Integer Register File

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



x86-64 Integer Register File

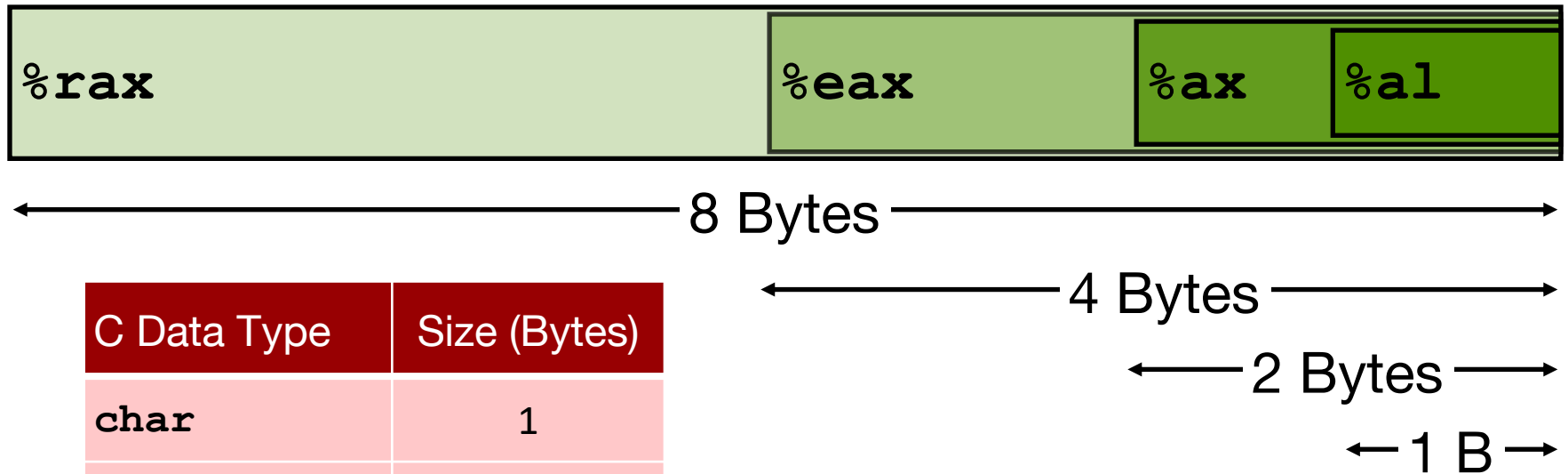
- Lower-half of each register can be independently addressed (until 1 bytes)



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)

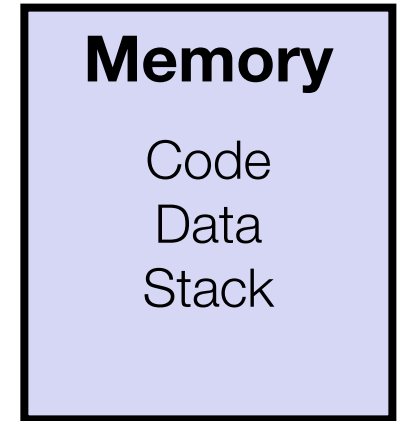
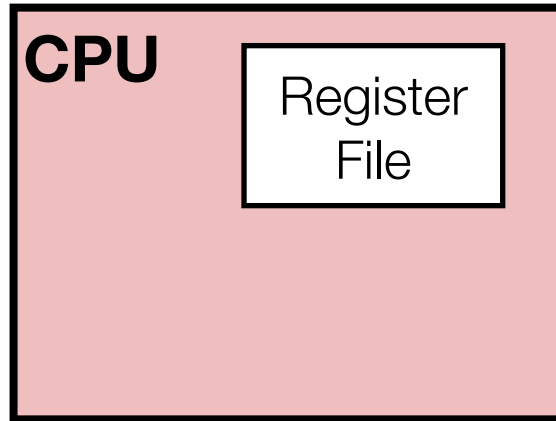


C Data Type	Size (Bytes)
<code>char</code>	1
<code>short</code>	2
<code>int</code>	4
<code>long</code>	8
Pointer	8

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

Assembly Code's View of Computer

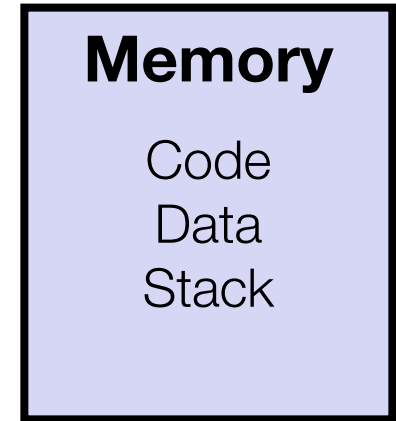
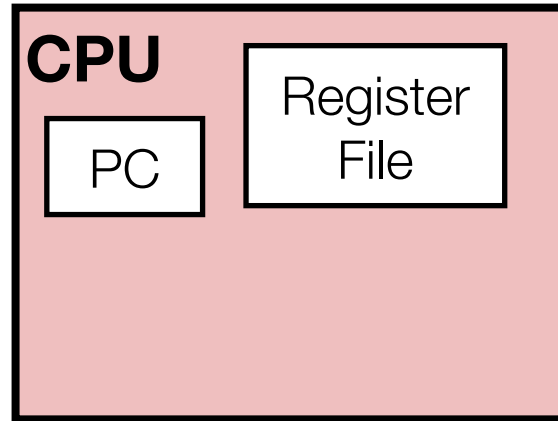
Assembly
Programmer's
Perspective
of a Computer



- (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

Assembly Programmer's Perspective of a Computer



- **(Byte Addressable) Memory**

- Code: instructions
- Data
- Stack to support function call

- **Register file**

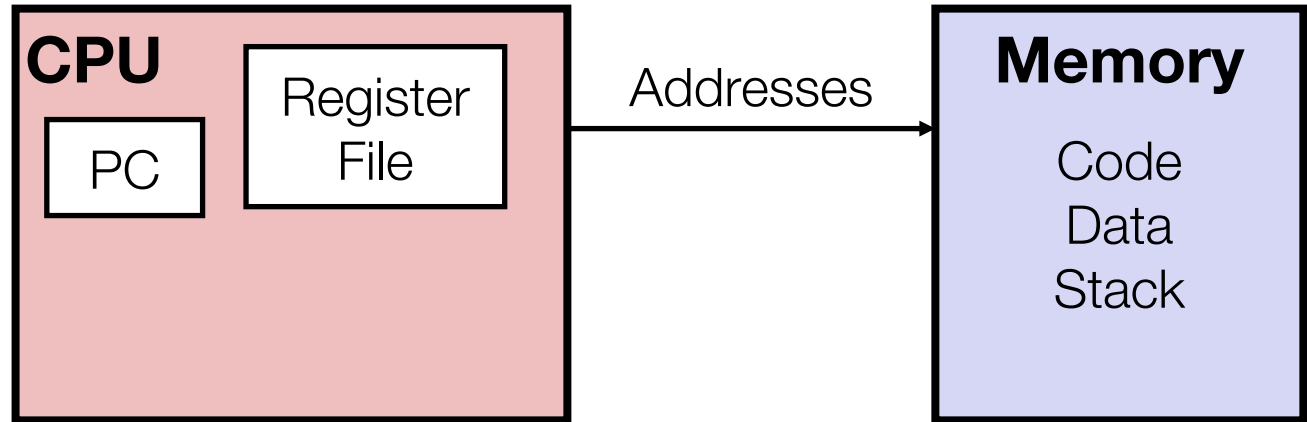
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- **PC: Program counter**

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

Assembly Code's View of Computer

Assembly Programmer's Perspective of a Computer



- (Byte Addressable) Memory

- Code: instructions
- Data
- Stack to support function call

- Register file

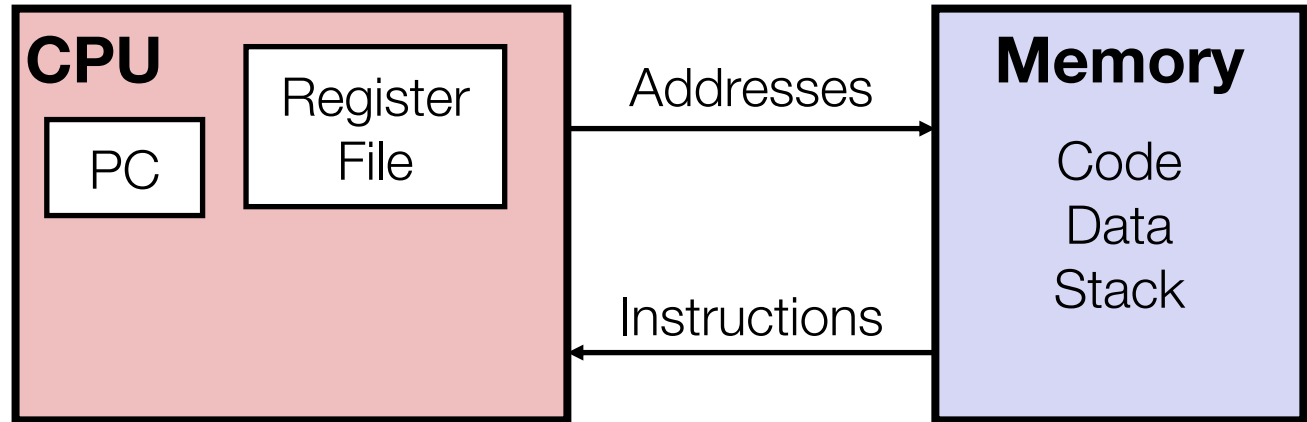
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- PC: Program counter

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

Assembly Code's View of Computer

Assembly Programmer's Perspective of a Computer



- (Byte Addressable) Memory

- Code: instructions
- Data
- Stack to support function call

- Register file

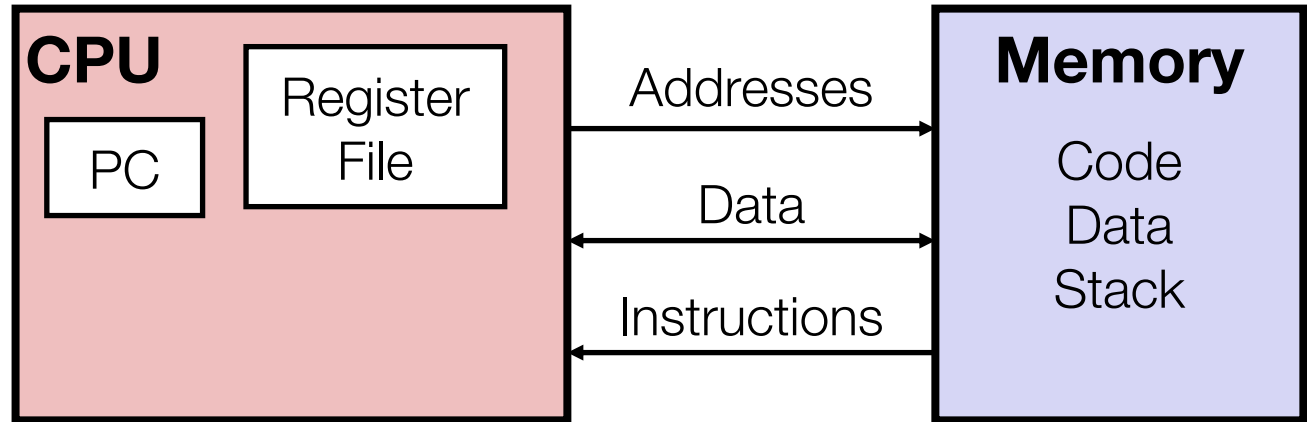
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- PC: Program counter

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

Assembly Code's View of Computer

Assembly Programmer's Perspective of a Computer



- (Byte Addressable) Memory

- Code: instructions
- Data
- Stack to support function call

- Register file

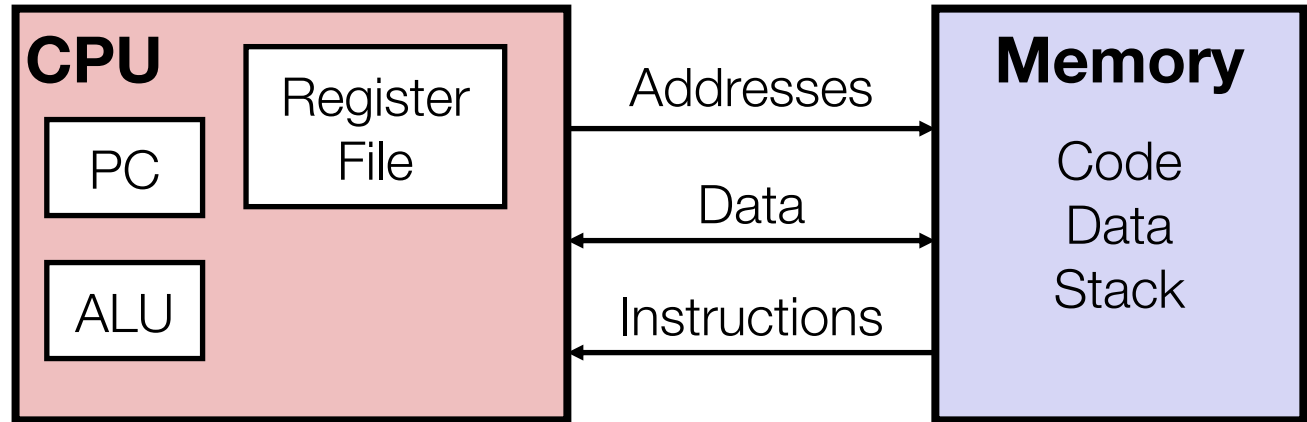
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- PC: Program counter

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

Assembly Code's View of Computer

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

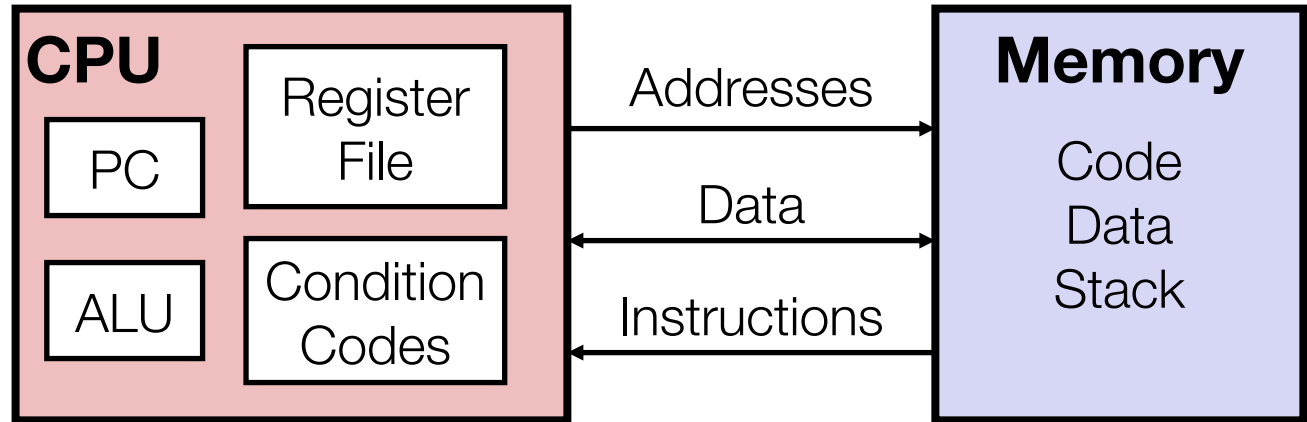
- A special register containing address of next instruction
- Called “RIP” in x86-64

- Arithmetic logic unit (ALU)

- Where computation happens

Assembly Code's View of Computer

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)

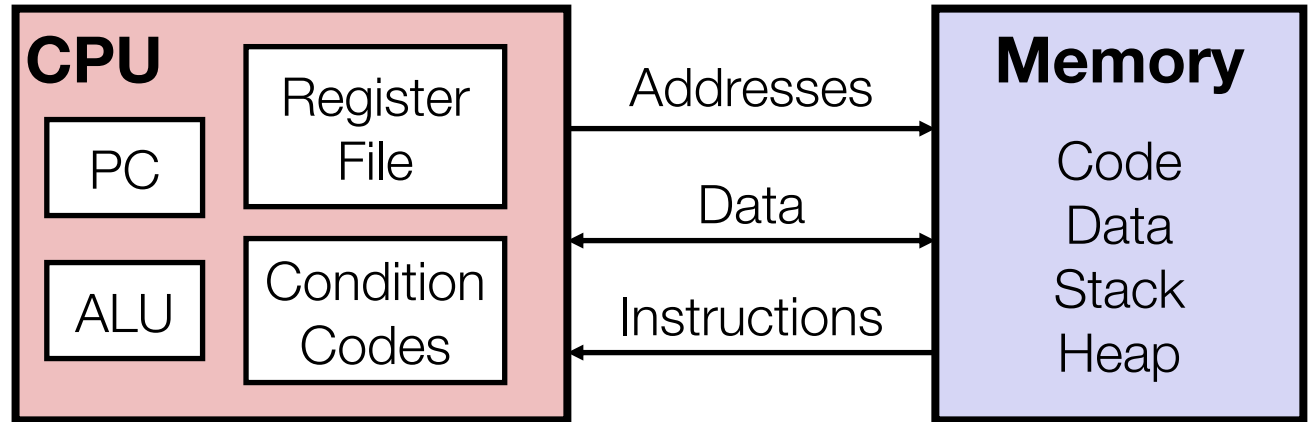
- Where computation happens

- Condition codes

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

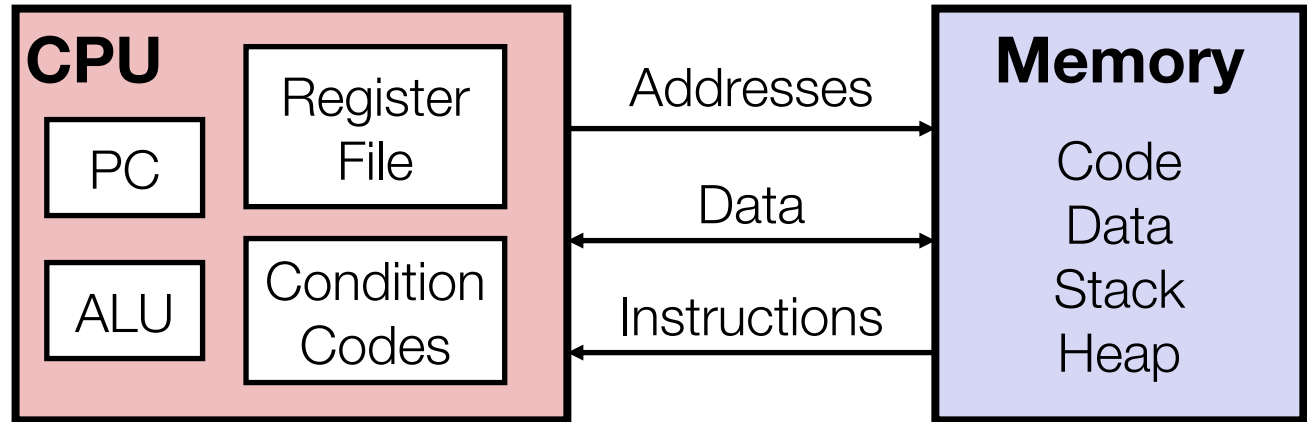
Assembly Program Instructions

Assembly
Programmer's
Perspective
of a Computer



Assembly Program Instructions

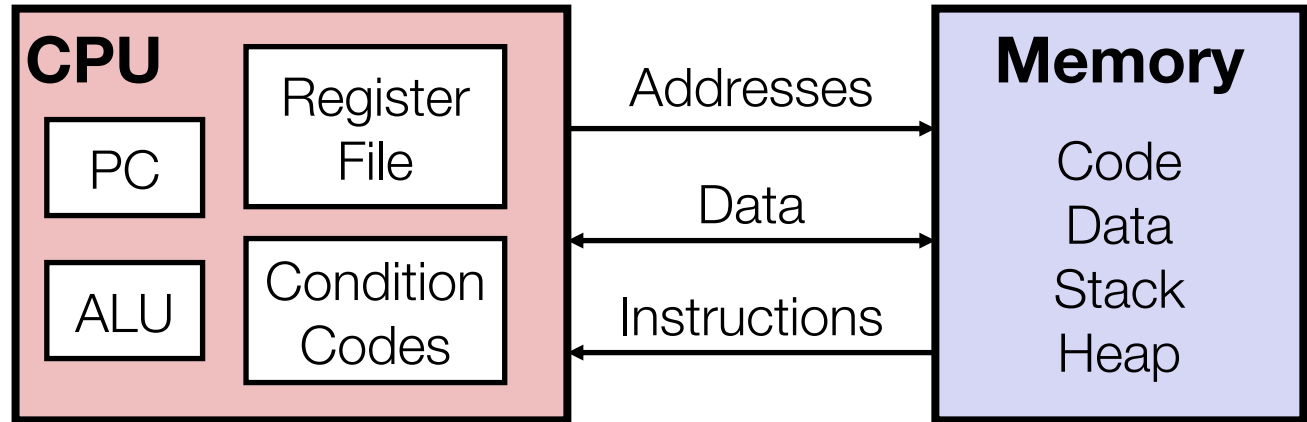
Assembly
Programmer's
Perspective
of a Computer



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

Assembly Program Instructions

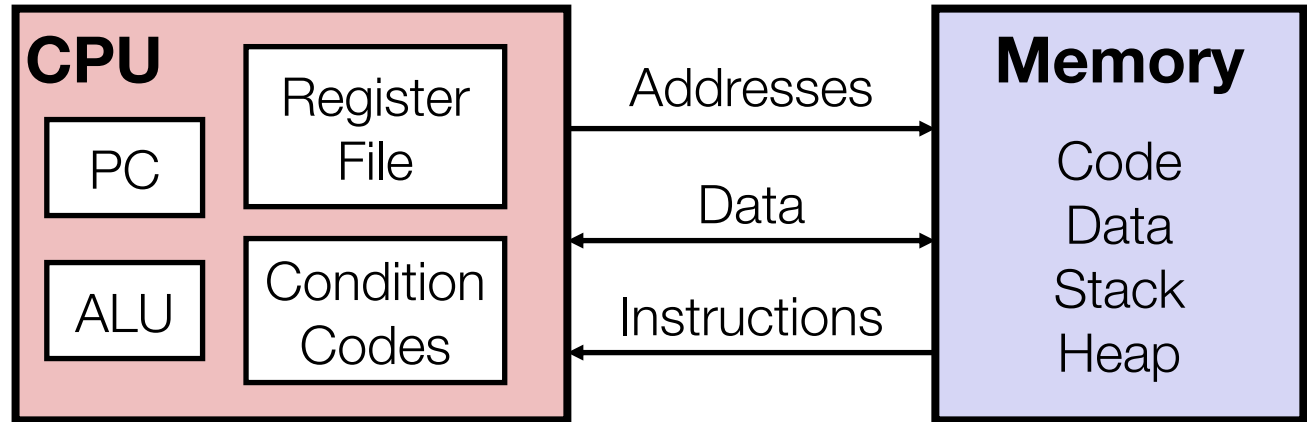
Assembly Programmer's Perspective of a Computer



- *Compute Instruction*: Perform arithmetics on register or memory data
 - `addq %rax, %rbx`
 - C constructs: +, -, >>, etc.
- *Data Movement Instruction*: Transfer data between memory and register
 - `movq %rax, (%rbx)`

Assembly Program Instructions

Assembly Programmer's Perspective of a Computer



- *Compute Instruction*: Perform arithmetics on register or memory data
 - `addq %rax, %rbx`
 - C constructs: `+`, `-`, `>>`, etc.
- *Data Movement Instruction*: Transfer data between memory and register
 - `movq %rax, (%rbx)`
- *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

Binary Code for **sumstore**

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

Memory

```
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

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

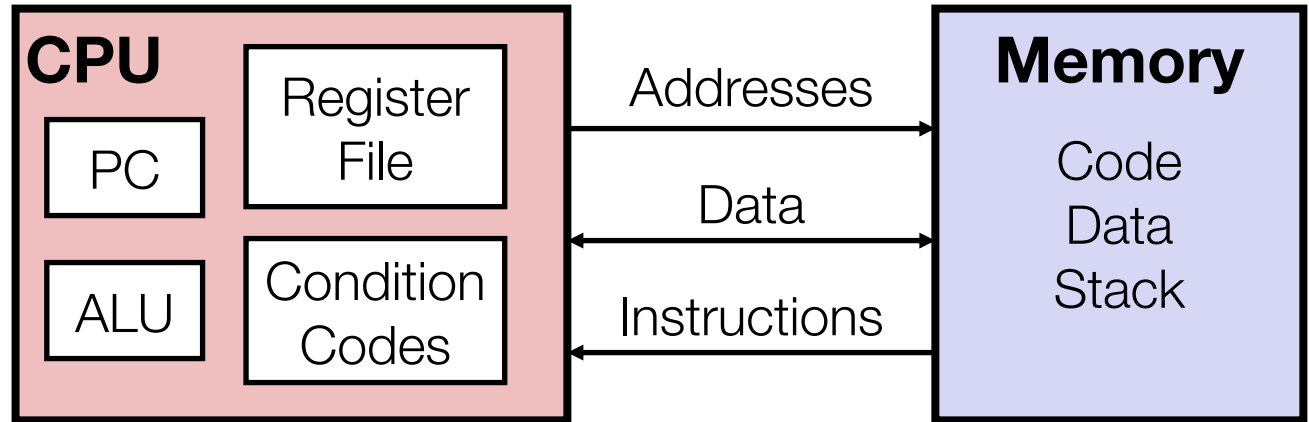
Obtain (on CSUG machine) with command

```
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

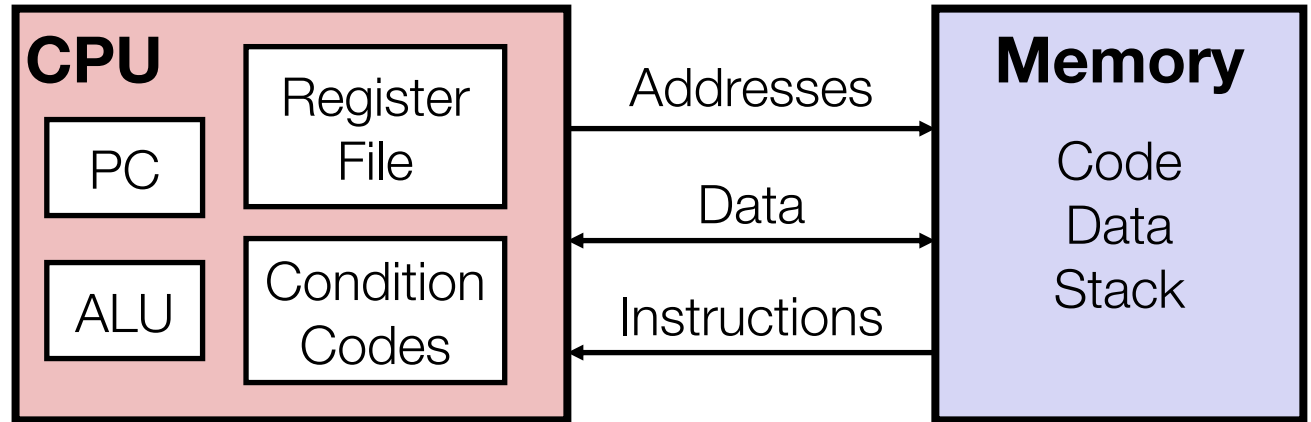
Assembly
Programmer's
Perspective
of a Computer



Fetch Instruction
(According to PC)

Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer

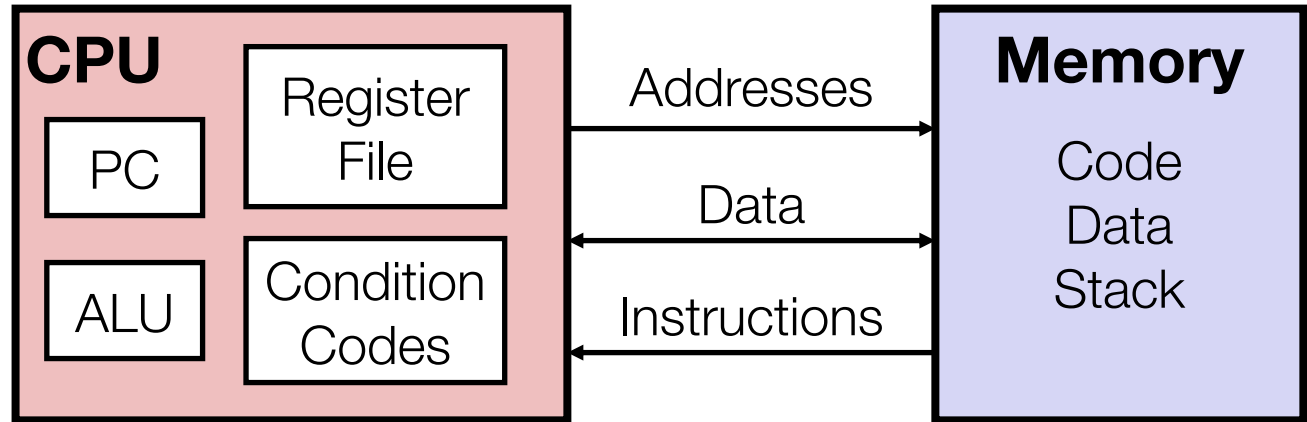


Fetch Instruction
(According to PC)

0x4801d8

Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer

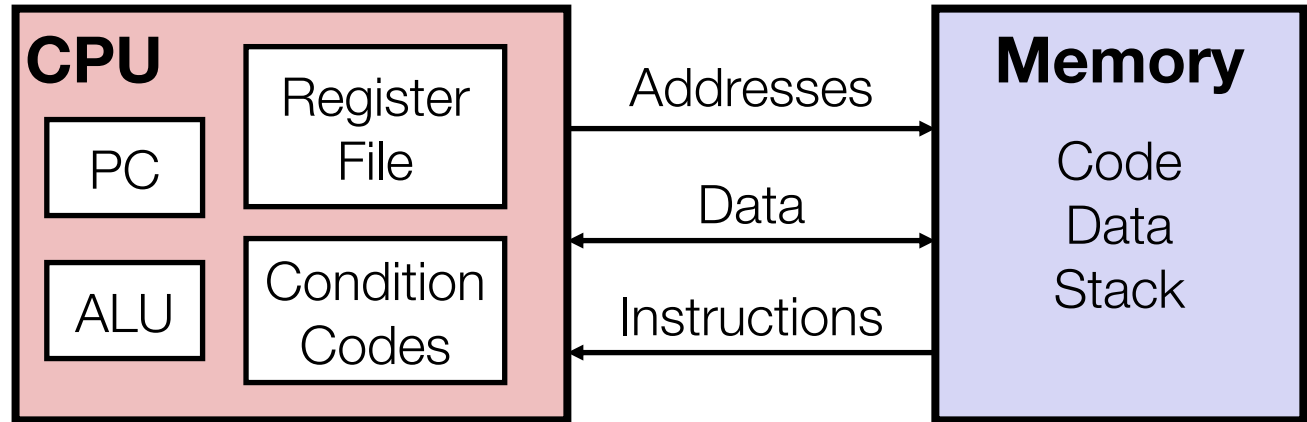


Fetch Instruction (According to PC) → Decode Instruction

`addq %rax, (%rbx)`

Instruction Processing Sequence

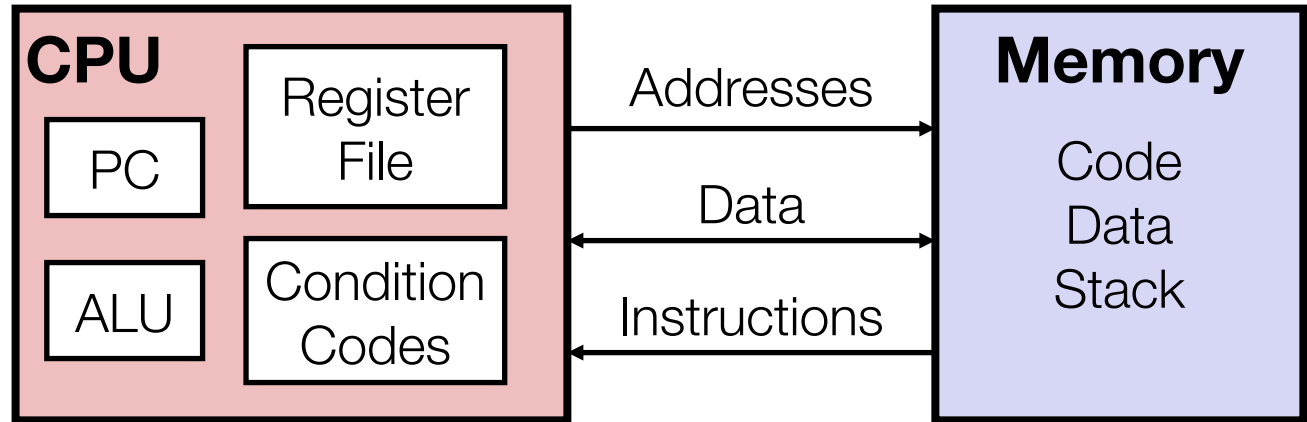
Assembly
Programmer's
Perspective
of a Computer



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

Instruction Processing Sequence

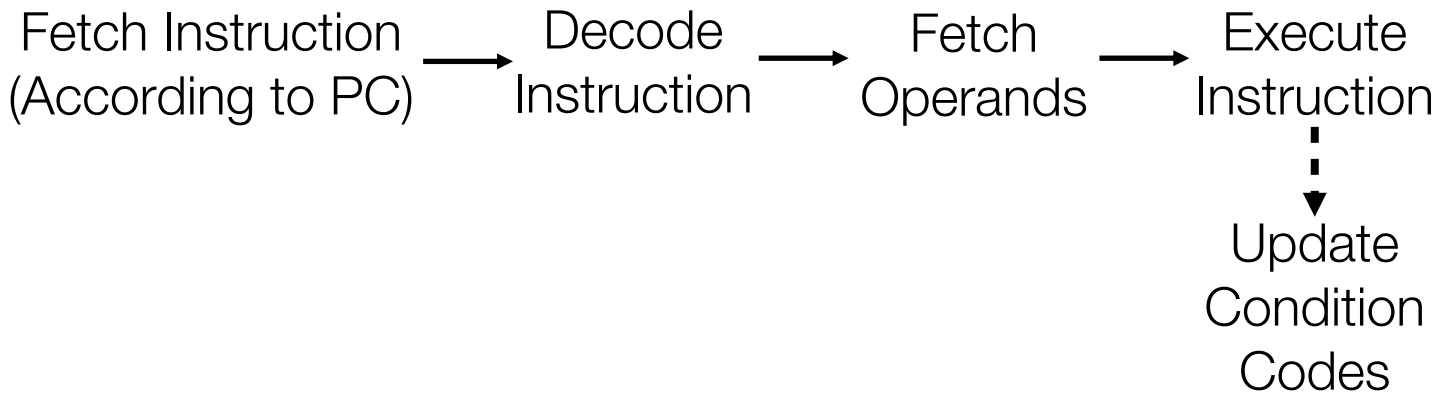
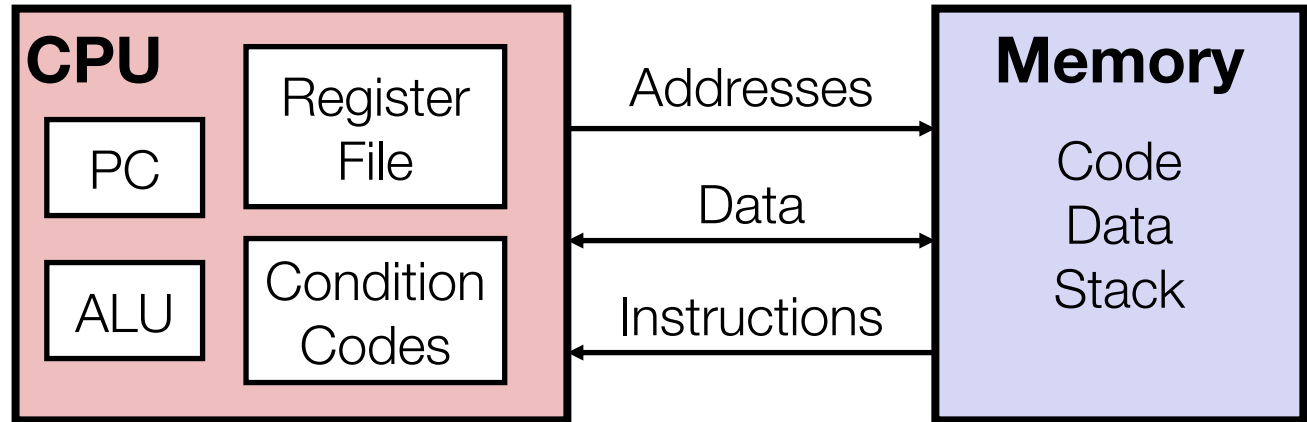
Assembly
Programmer's
Perspective
of a Computer



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

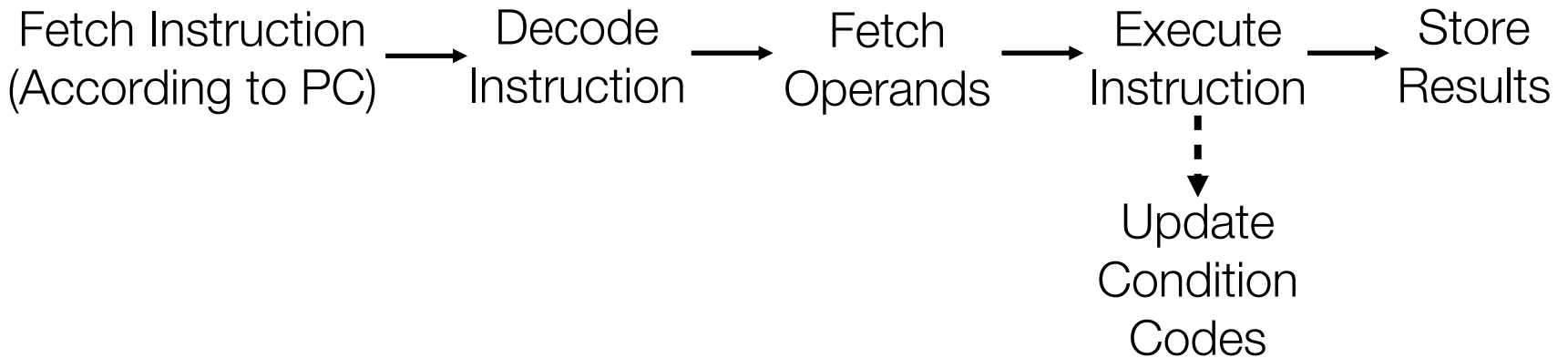
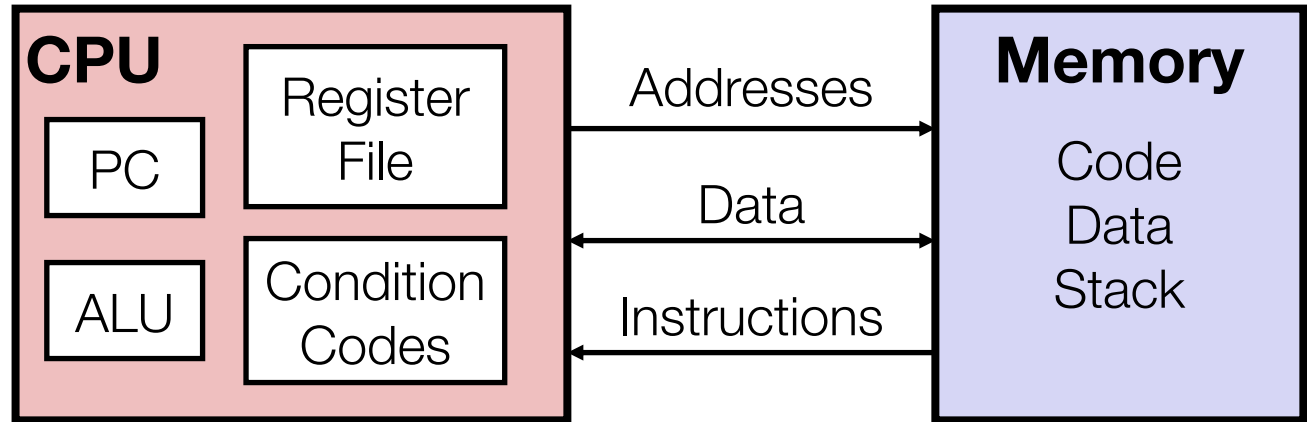
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



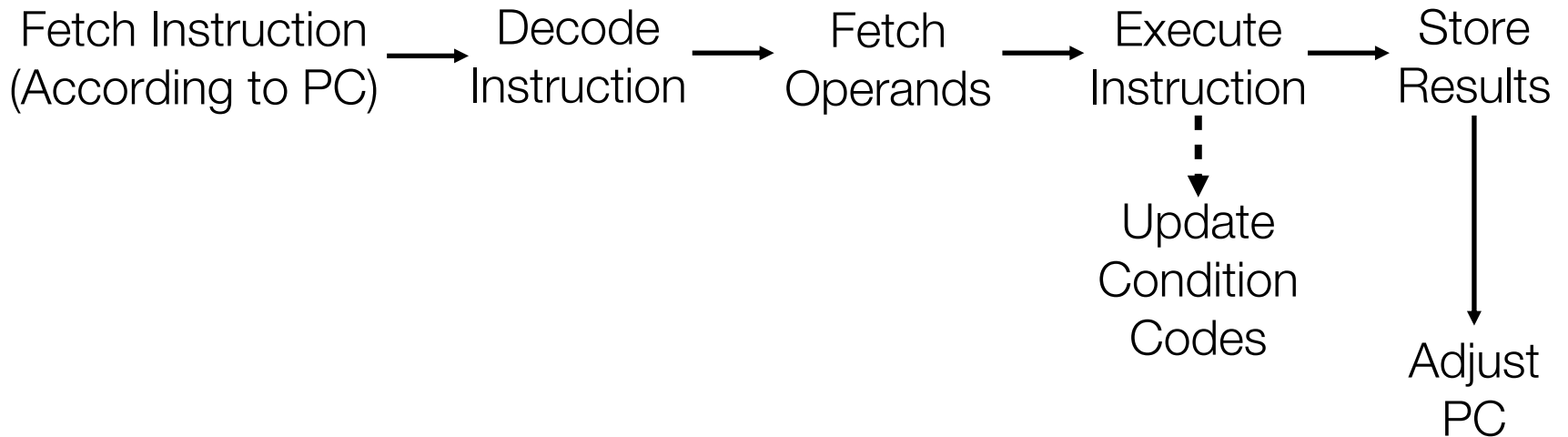
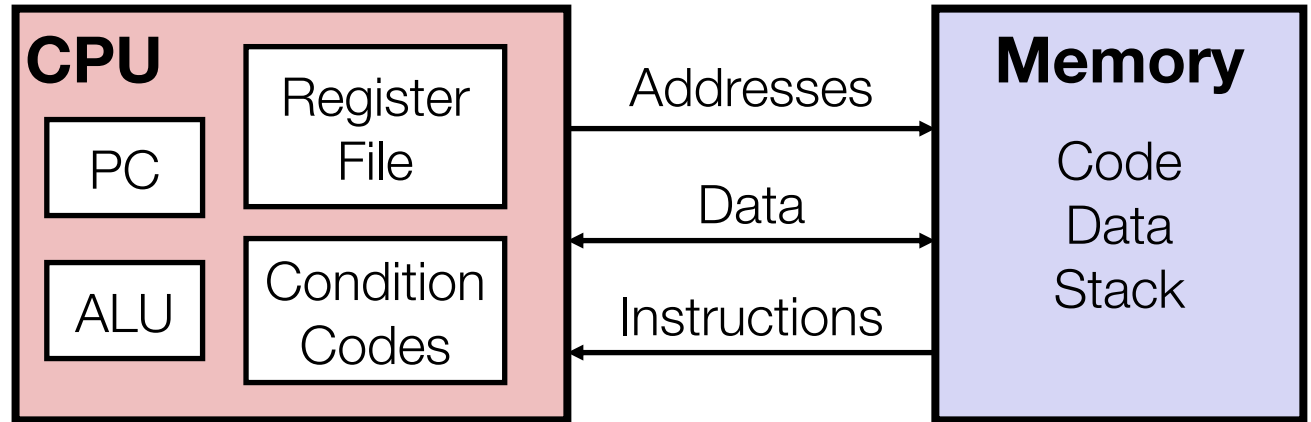
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



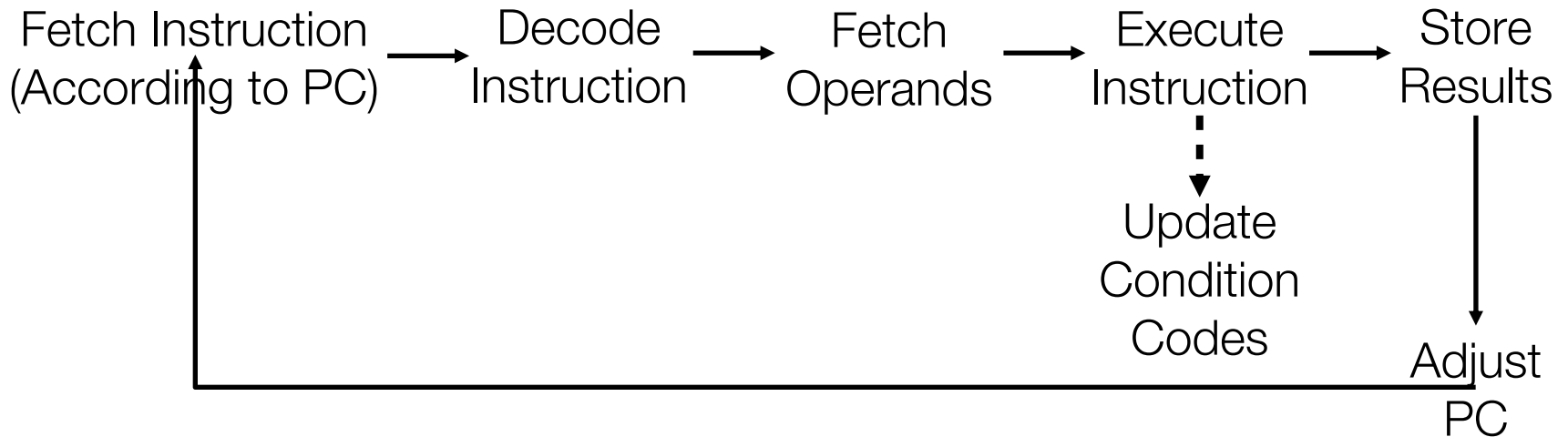
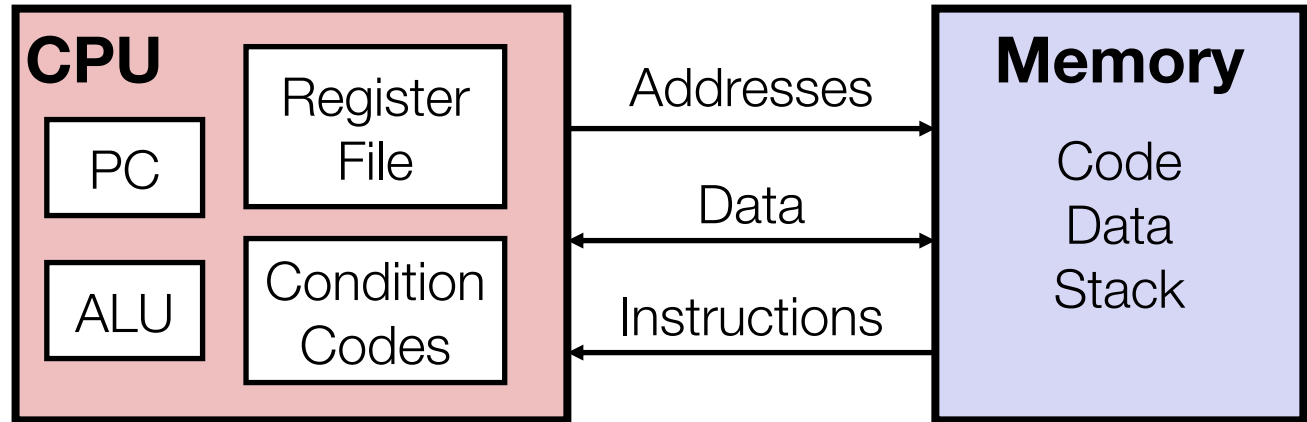
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



Today: Assembly Programming I: Basics

- Different ISAs and history behind them
- Memory, C, assembly, machine code
- Move operations (and addressing modes)

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)`

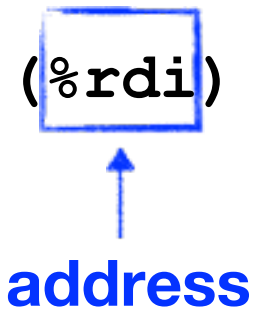
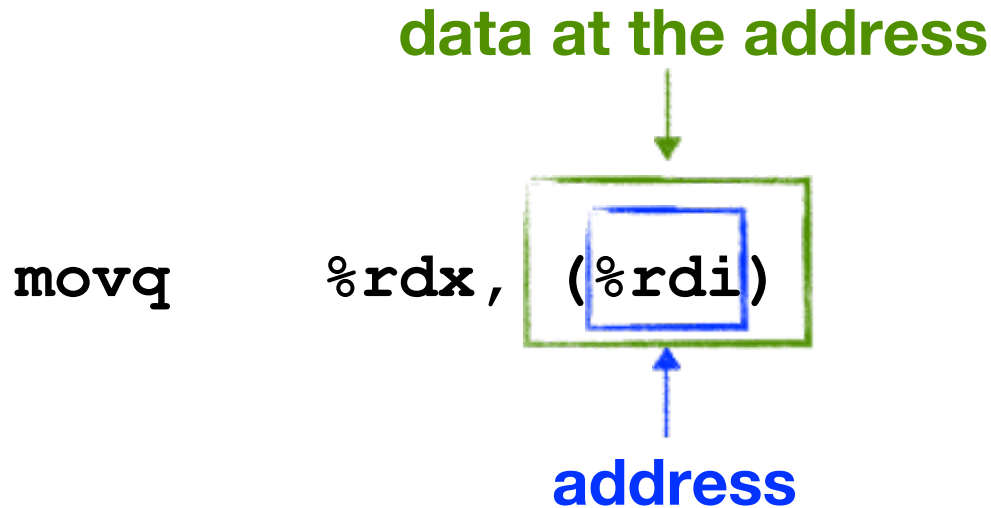


Diagram illustrating the instruction `movq %rdx, (%rdi)`. The register `%rdi` is enclosed in a blue box, and a blue arrow points up to it from the word **address**, indicating that the value in `%rdi` is the memory address.

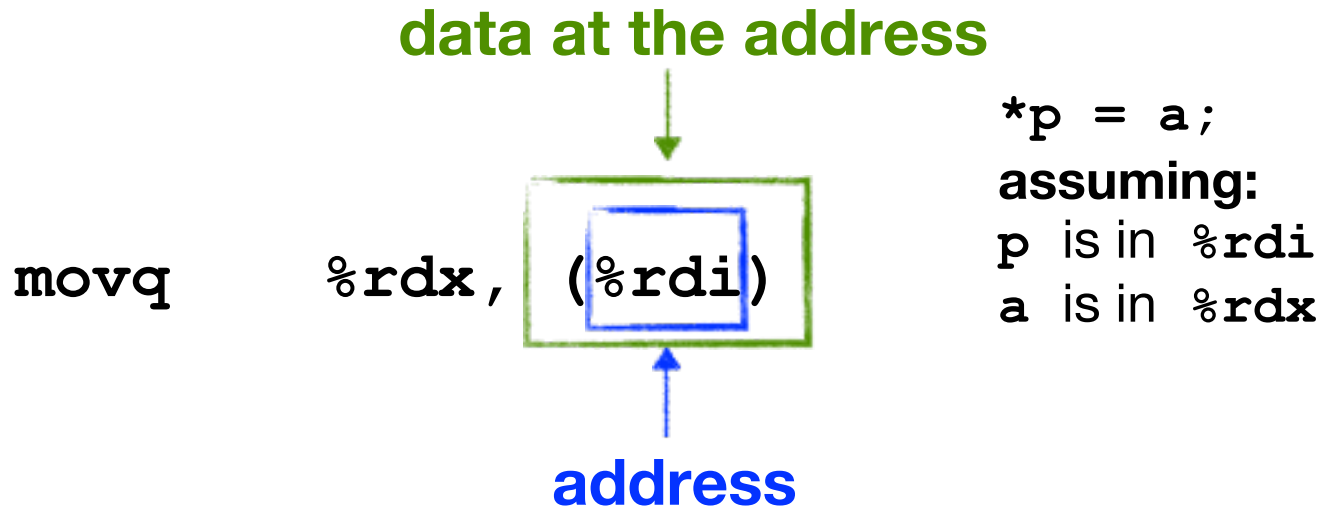
- Semantics:
 - Move (really, **copy**) data in register `%rdx` to memory location whose address is the value stored in `%rdi`
 - Pointer dereferencing

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

Data Movement Instructions

`movq` *Source, Dest*

Data Movement Instructions

`movq Source, Dest`

Operator **Operands**

Data Movement Instructions

`movq Source, Dest`

Operator Operands

- **Memory:**
 - Simplest example: (`%rax`)
 - How to obtain the address is called “addressing mode”

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

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	Mem	Reg		
	Reg	Reg		
		Mem		
	Imm	Reg		
		Mem		

*Cannot do memory-memory transfer
with a single instruction in x86.*

movq Operand Combinations

	Source	Dest	Example	C Analog
movq	Mem	Reg	movq (%rax), %rdx	
	Reg	Reg		
		Mem		
	Imm	Reg		
		Mem		

*Cannot do memory-memory transfer
with a single instruction in x86.*

movq Operand Combinations

	Source	Dest	Example	C Analog
movq	Mem	Reg	movq (%rax), %rdx	temp = *p;
	Reg	<div>Reg Mem</div>		
	Imm	<div>Reg Mem</div>		

*Cannot do memory-memory transfer
with a single instruction in x86.*

movq Operand Combinations

	Source	Dest	Example	C Analog
movq	Mem	Reg	movq (%rax), %rdx	temp = *p;
	Reg	Reg	movq %rax, %rdx	
		Mem		
	Imm	Reg		
		Mem		

*Cannot do memory-memory transfer
with a single instruction in x86.*

movq Operand Combinations

	Source	Dest	Example	C Analog
movq	Mem	Reg	movq (%rax), %rdx	temp = *p;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem		
	Imm	Reg		
		Mem		

*Cannot do memory-memory transfer
with a single instruction in x86.*

movq Operand Combinations

	Source	Dest	Example	C Analog
movq	Mem	Reg	movq (%rax), %rdx	temp = *p;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	
	Imm	Reg		
		Mem		

*Cannot do memory-memory transfer
with a single instruction in x86.*

movq Operand Combinations

	Source	Dest	Example	C Analog
movq	Mem	Reg	movq (%rax), %rdx	temp = *p;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Imm	Reg		
		Mem		

*Cannot do memory-memory transfer
with a single instruction in x86.*

movq Operand Combinations

	Source	Dest	Example	C Analog
movq	Mem	Reg	movq (%rax), %rdx	temp = *p;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Imm	Reg	movq \$0x4, %rax	
		Mem		

*Cannot do memory-memory transfer
with a single instruction in x86.*

movq Operand Combinations

	Source	Dest	Example	C Analog
movq	Mem	Reg	movq (%rax), %rdx	temp = *p;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Imm	Reg	movq \$0x4, %rax	temp = 0x4;
		Mem		

*Cannot do memory-memory transfer
with a single instruction in x86.*

movq Operand Combinations

	Source	Dest	Example	C Analog
movq	Mem	Reg	movq (%rax), %rdx	temp = *p;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Imm	Reg	movq \$0x4, %rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	

*Cannot do memory-memory transfer
with a single instruction in x86.*

movq Operand Combinations

	Source	Dest	Example	C Analog
movq	Mem	Reg	movq (%rax), %rdx	temp = *p;
		Mem		
		Mem		
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Imm	Reg	movq \$0x4, %rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	*p = -147;

*Cannot do memory-memory transfer
with a single instruction in x86.*