

CSC 252: Computer Organization

Spring 2018: Lecture 5

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:

- **Assignment 1 is due tomorrow, midnight**
- **Assignment 2 is out**
- **Trivia 2 is due on the coming Tues, noon**

Announcement

Announcement

- Programming Assignment 2 is out
 - Due on **Feb 16, 11:59 PM**
 - Trivia due Feb 6, noon
 - You have 3 slip days

Sun 28	Mon 29	Tue 30	Wed 31	Thu Feb 1	Fri 2	Sat 3
4	5	6 Trivia	7	8	9	10
11	12	13	14	15	16 Due	17

Announcement

- Programming Assignment 2 is out
 - Due on **Feb 16, 11:59 PM**
 - Trivia due Feb 6, noon
 - You have 3 slip days
- Read the instructions before getting started!!!
 - You get 1/4 point off for every wrong answer
 - Maxed out at 10

Floating Point Review

$$v = (-1)^s \times 1.\textit{frac} \times 2^E$$



Floating Point Review

$$v = (-1)^s \times 1.\text{frac} \times 2^E$$



Denormalized 

s	exp	frac	Value	Value
0	000	00	0.00×2^{-2}	0
0	000	11	0.11×2^{-2}	3/16

- Denormalized
 - $E = (\text{exp} + 1) - \text{bias}$
 - $M = 0.\text{frac}$

Floating Point Review

$$v = (-1)^s \times 1.\text{frac} \times 2^E$$



- Denormalized
 - $E = (\text{exp} + 1) - \text{bias}$
 - $M = 0.\text{frac}$
- Normalized
 - $E = \text{exp} - \text{bias}$
 - $M = 1.\text{frac}$

Denormalized

Normalized

s	exp	frac	Value	Value
0	000	00	0.00×2^{-2}	0
0	000	11	0.11×2^{-2}	3/16
0	001	00	1.00×2^{-2}	1/4
0	001	11	1.11×2^{-2}	7/16
0	010	00	1.00×2^{-1}	1/2
0	010	11	1.11×2^{-1}	7/8
0	100	00	1.00×2^0	1
0	100	11	1.11×2^0	1 3/4
0	101	00	1.00×2^1	2
0	101	11	1.11×2^1	3 1/2
0	110	00	1.00×2^2	4
0	110	11	1.11×2^2	7

Floating Point Review

$$v = (-1)^s \times 1.\text{frac} \times 2^E$$



- Denormalized
 - $E = (\text{exp} + 1) - \text{bias}$
 - $M = 0.\text{frac}$
- Normalized
 - $E = \text{exp} - \text{bias}$
 - $M = 1.\text{frac}$

Denormalized

Normalized

Special Value

s	exp	frac	Value	Value
0	000	00	0.00×2^{-2}	0
0	000	11	0.11×2^{-2}	3/16
0	001	00	1.00×2^{-2}	1/4
0	001	11	1.11×2^{-2}	7/16
0	010	00	1.00×2^{-1}	1/2
0	010	11	1.11×2^{-1}	7/8
0	100	00	1.00×2^0	1
0	100	11	1.11×2^0	1 3/4
0	101	00	1.00×2^1	2
0	101	11	1.11×2^1	3 1/2
0	110	00	1.00×2^2	4
0	110	11	1.11×2^2	7
0	111	00	infinite	infinite
0	111	11	NaN	NaN

Floating Point Review

	s	exp	frac	Value	Value
Denormalized	0	000	00	0.00×2^{-2}	0
	0	000	11	0.11×2^{-2}	3/16
Normalized	0	001	00	1.00×2^{-2}	1/4
	0	001	11	1.11×2^{-2}	7/16
	0	010	00	1.00×2^{-1}	1/2
	0	010	11	1.11×2^{-1}	7/8
	0	100	00	1.00×2^0	1
	0	100	11	1.11×2^0	1 3/4
	0	101	00	1.00×2^1	2
	0	101	11	1.11×2^1	3 1/2
	0	110	00	1.00×2^2	4
	0	110	11	1.11×2^2	7
Special Value	0	111	00	infinite	infinite
	0	111	11	NaN	NaN

Floating Point Review

- If you do an integer increment on a positive FP number, you get the next larger FP number.

Denormalized



Normalized



Special Value



s	exp	frac	Value	Value
0	000	00	0.00×2^{-2}	0
0	000	11	0.11×2^{-2}	3/16
0	001	00	1.00×2^{-2}	1/4
0	001	11	1.11×2^{-2}	7/16
0	010	00	1.00×2^{-1}	1/2
0	010	11	1.11×2^{-1}	7/8
0	100	00	1.00×2^0	1
0	100	11	1.11×2^0	1 3/4
0	101	00	1.00×2^1	2
0	101	11	1.11×2^1	3 1/2
0	110	00	1.00×2^2	4
0	110	11	1.11×2^2	7
0	111	00	infinite	infinite
0	111	11	NaN	NaN

Floating Point Review

- If you do an integer increment on a positive FP number, you get the next larger FP number.
- Bit patterns representing non-negative numbers are ordered the same way as integers, so could use regular integer comparison.

Denormalized



Normalized



Special Value



<i>s</i>	<i>exp</i>	<i>frac</i>	Value	Value
0	000	00	0.00×2^{-2}	0
0	000	11	0.11×2^{-2}	3/16
0	001	00	1.00×2^{-2}	1/4
0	001	11	1.11×2^{-2}	7/16
0	010	00	1.00×2^{-1}	1/2
0	010	11	1.11×2^{-1}	7/8
0	100	00	1.00×2^0	1
0	100	11	1.11×2^0	1 3/4
0	101	00	1.00×2^1	2
0	101	11	1.11×2^1	3 1/2
0	110	00	1.00×2^2	4
0	110	11	1.11×2^2	7
0	111	00	infinite	infinite
0	111	11	NaN	NaN

Floating Point Review

- If you do an integer increment on a positive FP number, you get the next larger FP number.
- Bit patterns representing non-negative numbers are ordered the same way as integers, so could use regular integer comparison.
- You don't get this property if:
 - *exp* is interpreted as signed
 - *exp* and *frac* are swapped

Denormalized



Normalized



Special Value



<i>s</i>	<i>exp</i>	<i>frac</i>	Value	Value
0	000	00	0.00×2^{-2}	0
0	000	11	0.11×2^{-2}	3/16
0	001	00	1.00×2^{-2}	1/4
0	001	11	1.11×2^{-2}	7/16
0	010	00	1.00×2^{-1}	1/2
0	010	11	1.11×2^{-1}	7/8
0	100	00	1.00×2^0	1
0	100	11	1.11×2^0	1 3/4
0	101	00	1.00×2^1	2
0	101	11	1.11×2^1	3 1/2
0	110	00	1.00×2^2	4
0	110	11	1.11×2^2	7
0	111	00	infinite	infinite
0	111	11	NaN	NaN

Floating Point in C

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	$2^{31} - 1$	$\sim 9.2 \times 10^{18}$
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

	C Data Type	Bits	Max Value	Max Value (Decimal)
Fixed point (implicit binary point)	char	8	$2^7 - 1$	127
	short	16	$2^{15} - 1$	32767
	int	32	$2^{31} - 1$	2147483647
	long	64	$2^{31} - 1$	$\sim 9.2 \times 10^{18}$
SP floating point	float	32	$(2 - 2^{-23}) \times 2^{127}$	$\sim 3.4 \times 10^{38}$
DP floating point	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: Generally sets to TMin

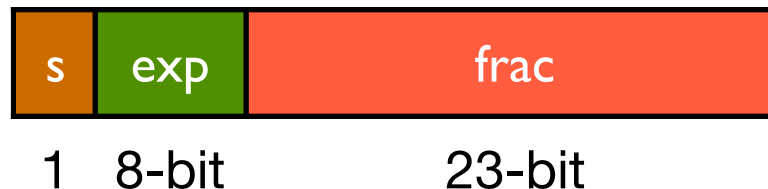
Floating Point Conversions/Casting in C

- **double/float → int**

- Truncates fractional part
- Like rounding toward zero
- Not defined when out of range or NaN: Generally sets to TMin

- **int → float**

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



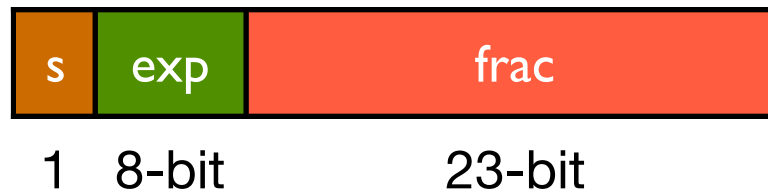
Floating Point Conversions/Casting in C

- **double/float \rightarrow int**

- Truncates fractional part
- Like rounding toward zero
- Not defined when out of range or NaN: Generally sets to TMin

- **int \rightarrow float**

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



- **int \rightarrow double**

- Exact conversion



So far in 252...

C Program

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

So far in 252...

C Program

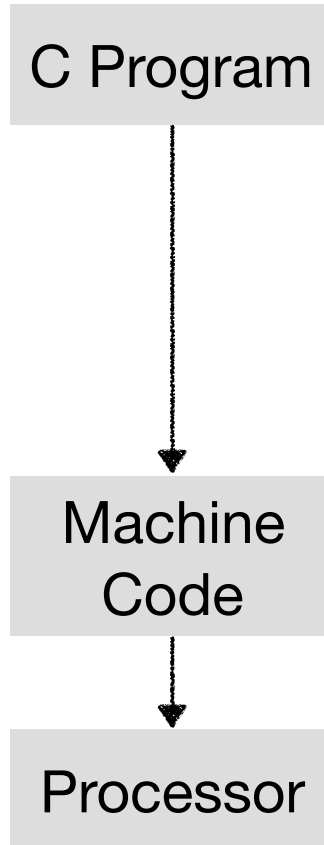


Machine
Code

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

00001111
01010101
11110000

So far in 252...

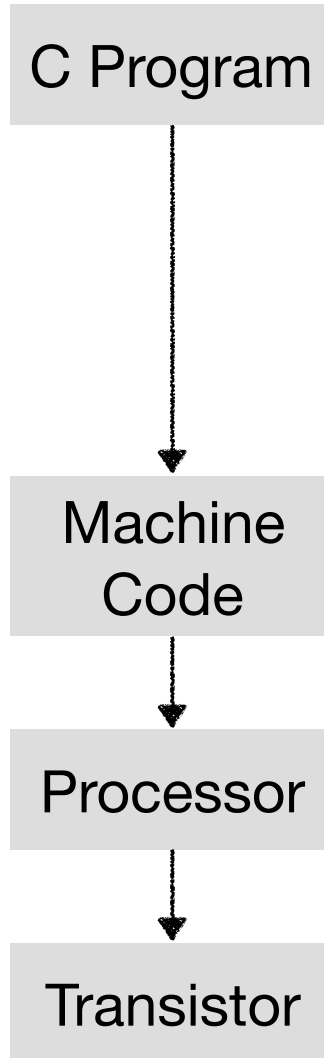


int, float
if, else
+, -, >>

00001111
01010101
11110000

Ripple-carry
Adder

So far in 252...



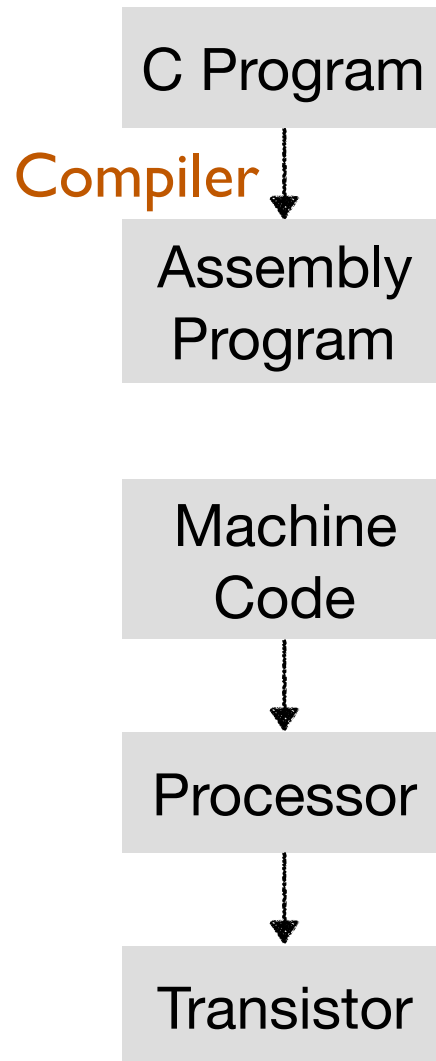
int, float
if, else
+, -, >>

00001111
01010101
11110000

Ripple-carry
Adder

NAND Gate
NOR Gate

So far in 252...



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

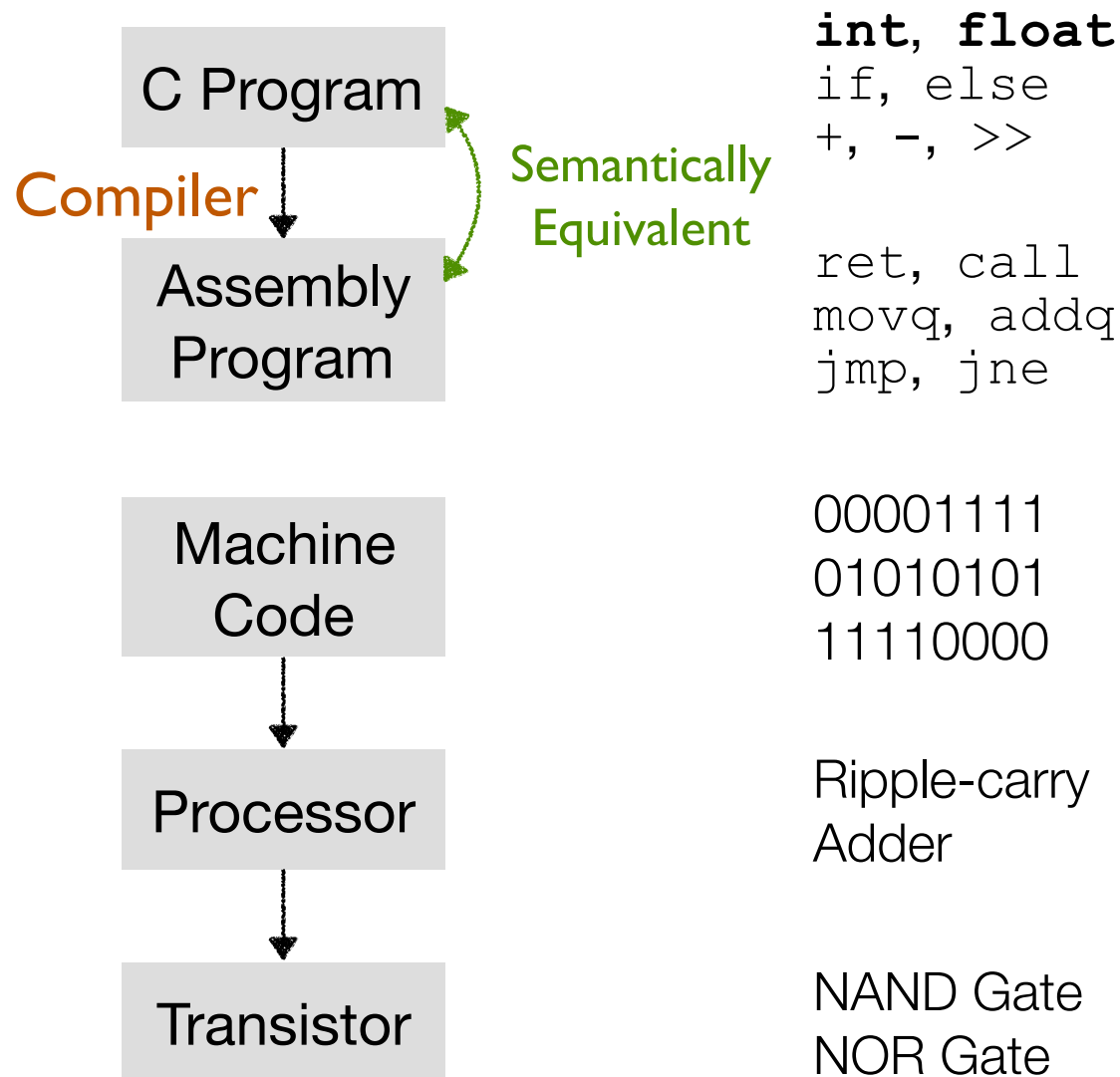
ret, call
movq, addq
jmp, jne

00001111
01010101
11110000

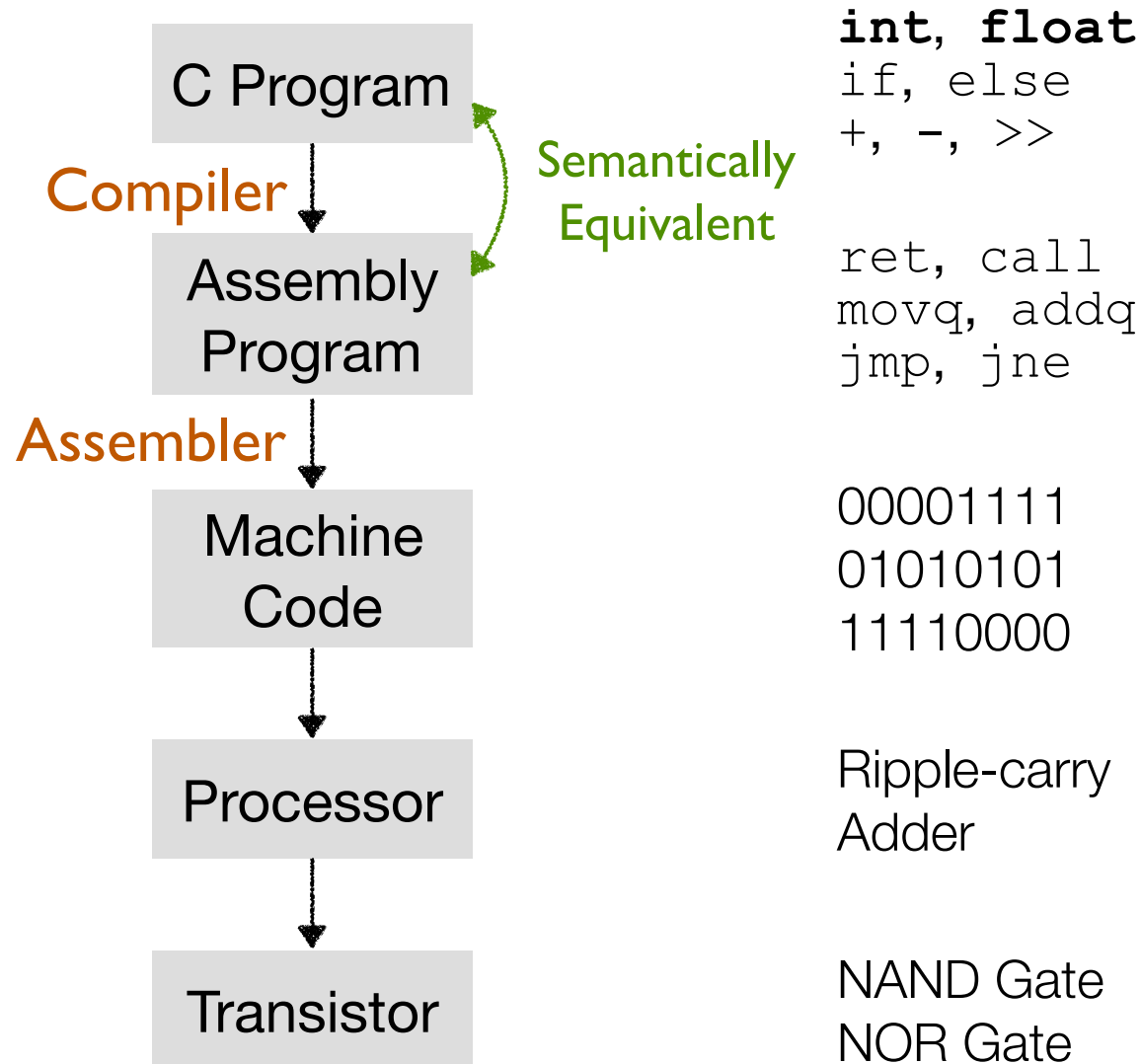
Ripple-carry
Adder

NAND Gate
NOR Gate

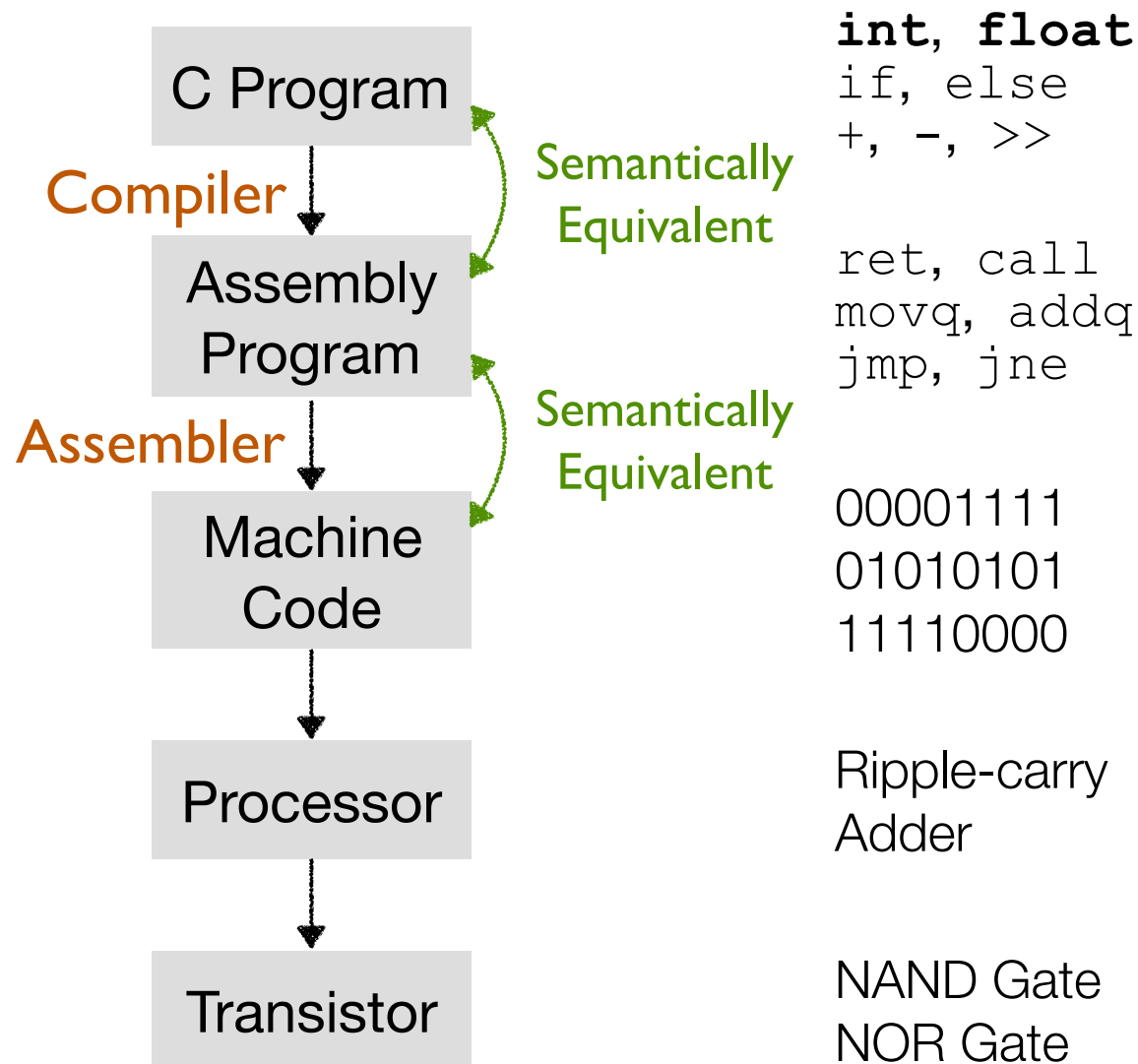
So far in 252...



So far in 252...

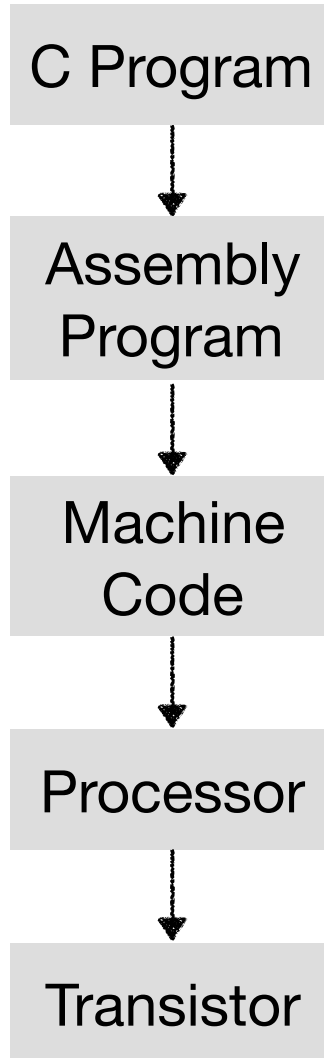


So far in 252...



So far in 252...

**High-Level
Language**



So far in 252...

High-Level
Language

Instruction Set
Architecture
(ISA)

C Program



Assembly
Program



Machine
Code



Processor



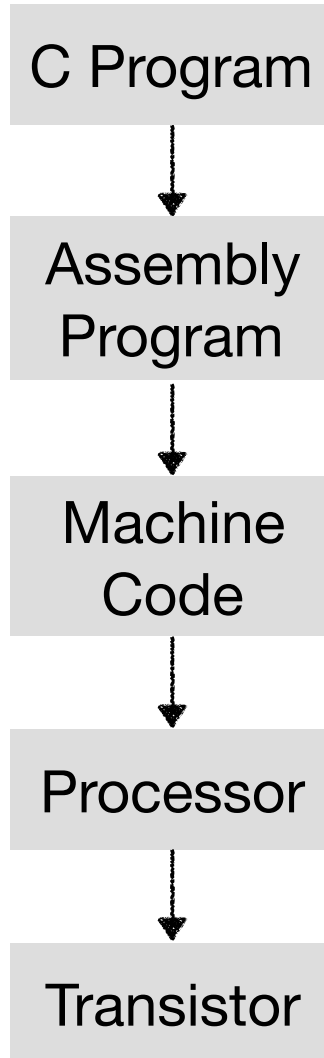
Transistor

- **ISA:** Assembly 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...

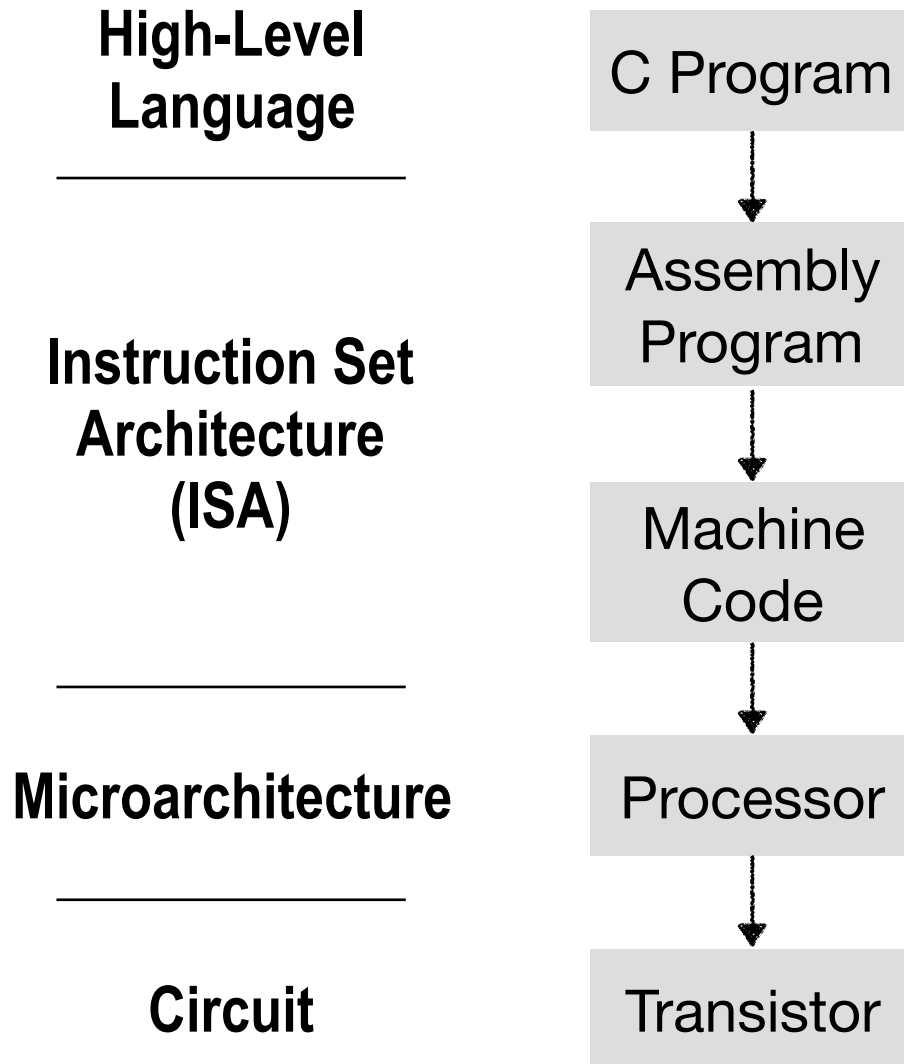
High-Level
Language

Instruction Set
Architecture
(ISA)



- **ISA:** Assembly 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:** Assembly 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

Instruction Set Architecture

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

Instruction Set Architecture

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

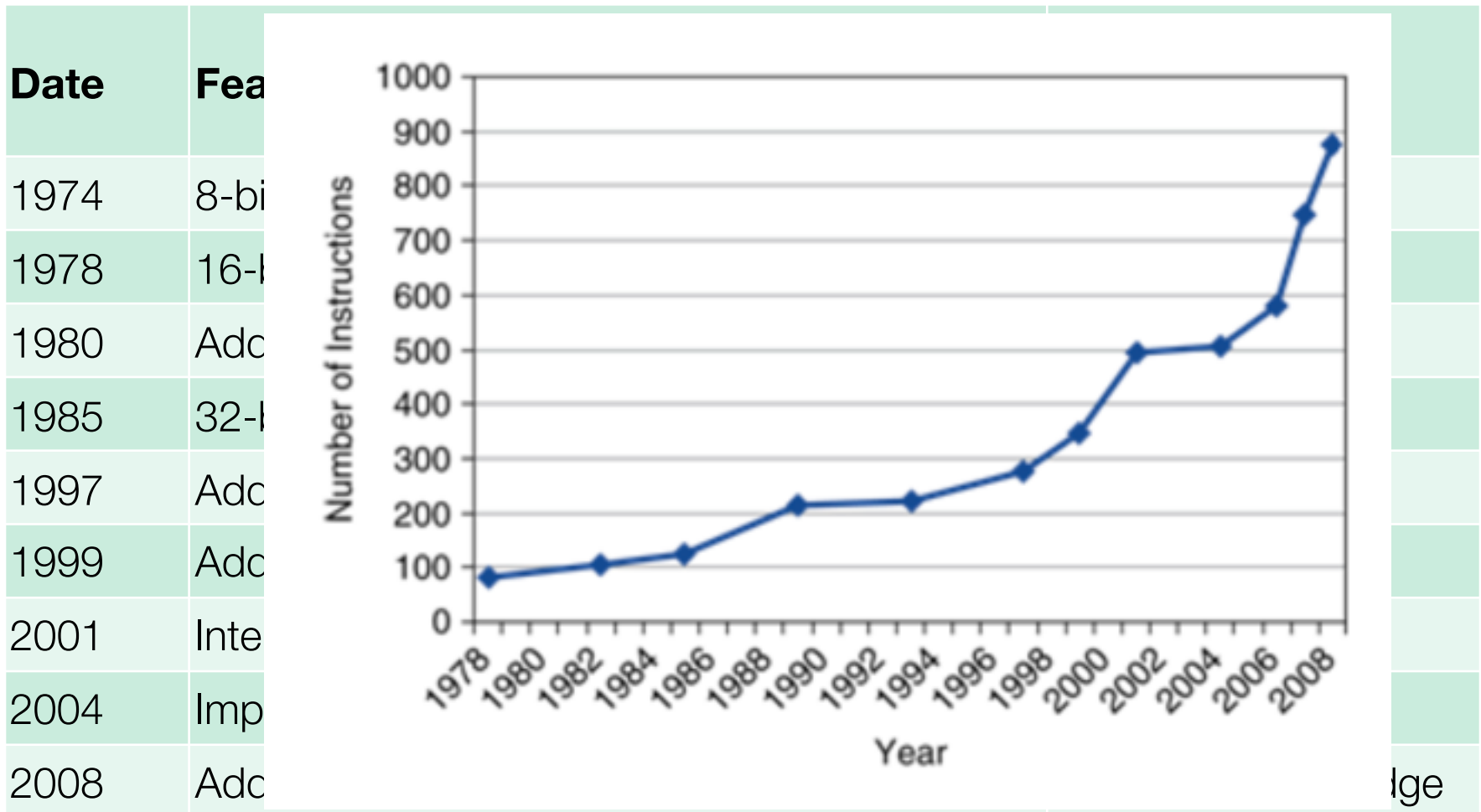
Intel x86 ISA Evolution (Milestones)

- 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

x86 Clones: Advanced Micro Devices (AMD)

- Today: Holding up not too badly



x86 Clones: Advanced Micro Devices (AMD)

- Today: Holding up not too badly



Our Coverage

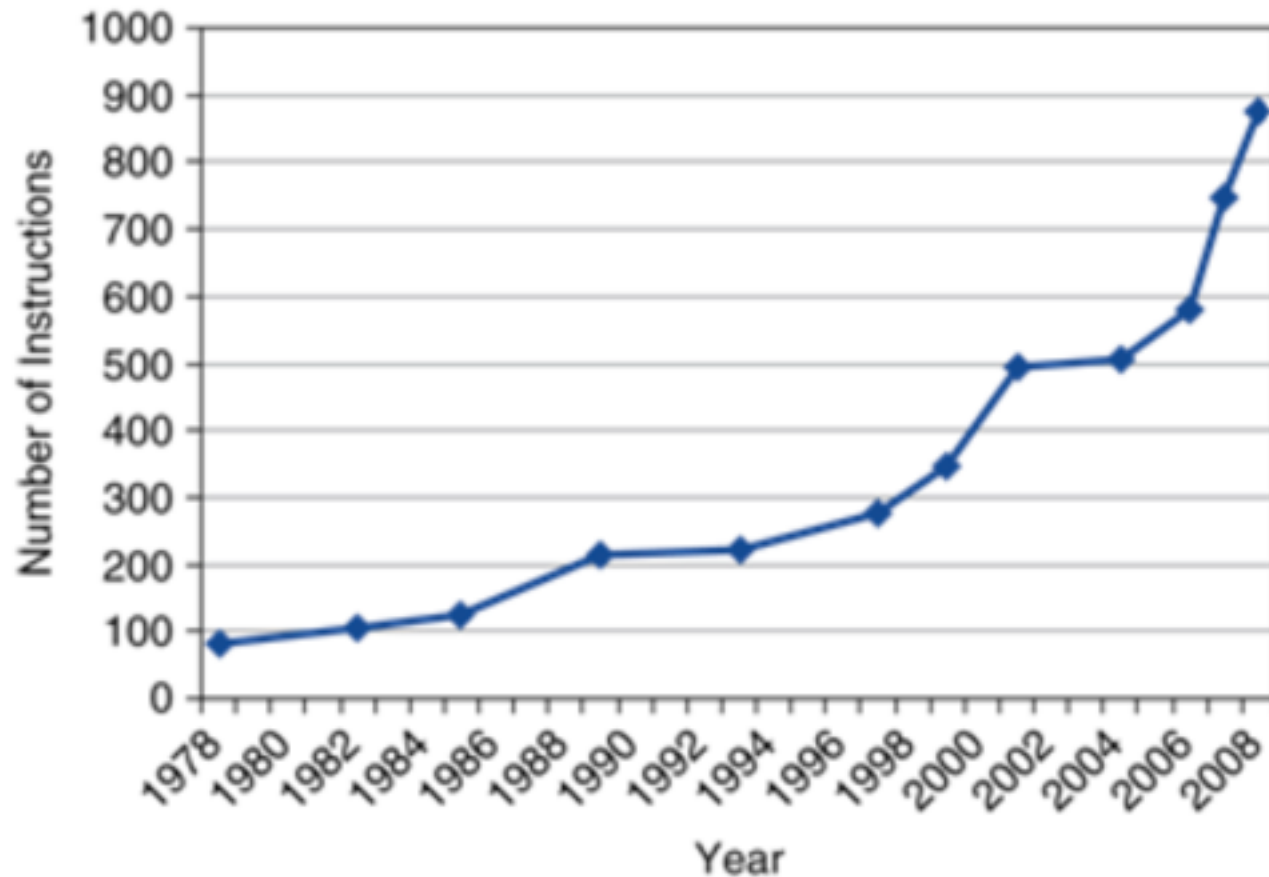
- IA32
 - The traditional x86
 - 2nd edition of the textbook
- x86-64
 - The standard
 - CSUG machine
 - 3rd edition of the textbook
 - Our focus

Aside: Moore's Law

- More instructions require more transistors to implement

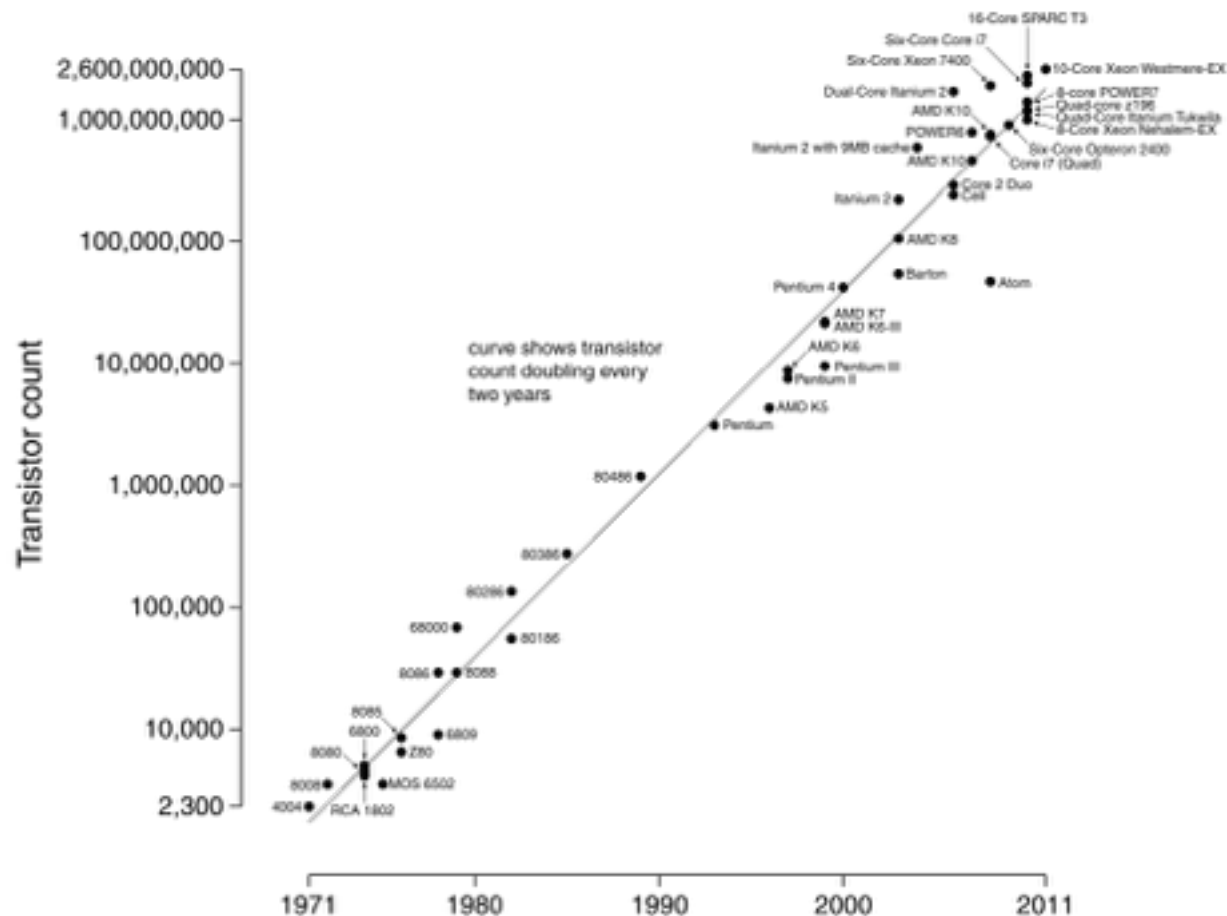
Aside: Moore's Law

- More instructions require more transistors to implement



Aside: Moore's Law

- More instructions require more transistors to implement



Aside: Moore's Law

- More instructions require more transistors to implement



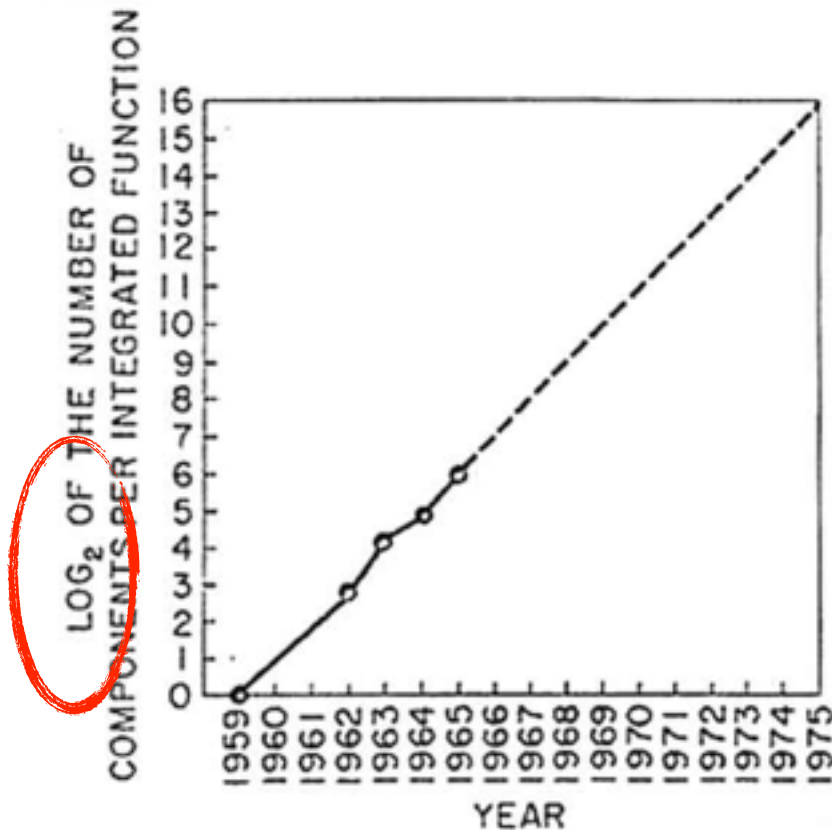
Aside: Moore's Law

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



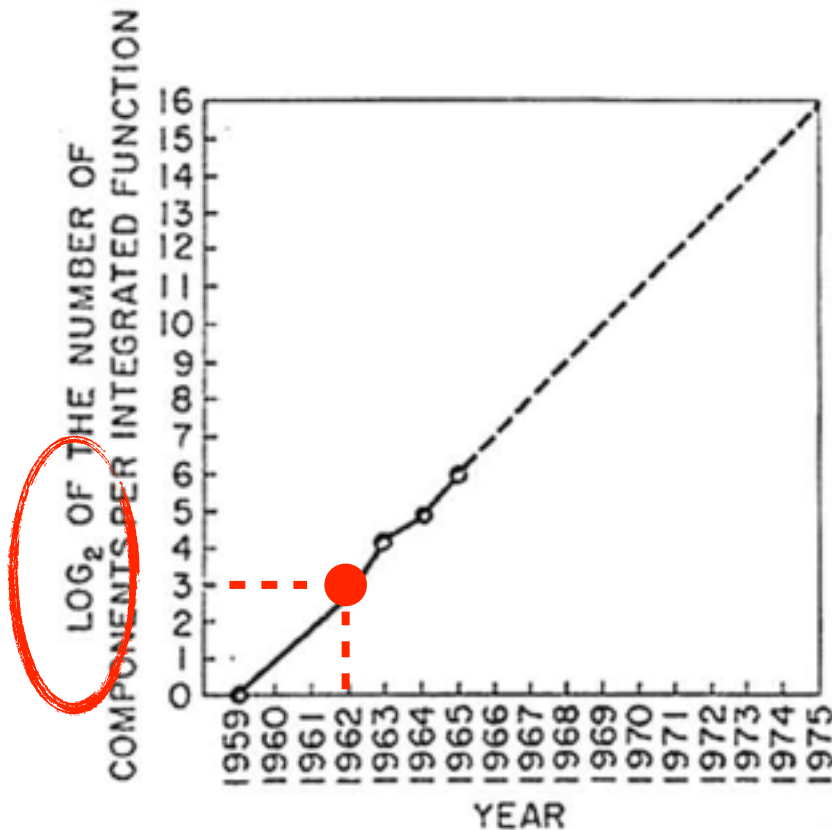
Aside: Moore's Law

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



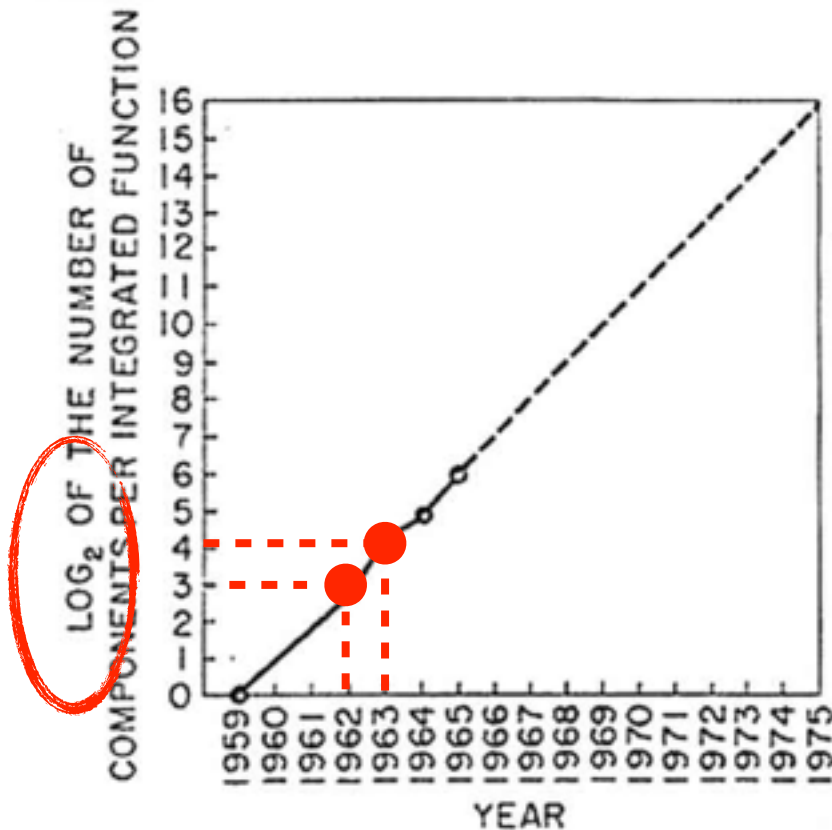
Aside: Moore's Law

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



Aside: Moore's Law

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



Aside: 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

Aside: 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...

Aside: Moore's Law

Aside: Moore's Law

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

Aside: Moore's Law

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

Aside: 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 ($1.4^2 \sim 2$)

Aside: 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:

Aside: 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?

Aside: 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**

Aside: 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?

Aside: 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**

Aside: 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?

Aside: 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**

Aside: Moore's Law

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



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.

Aside: Moore's Law

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



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.

Aside: 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?

Aside: 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**

Aside: 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**

Questions?

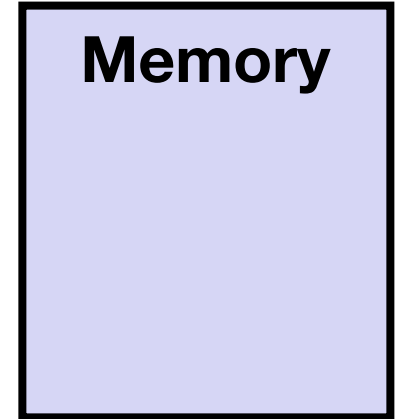
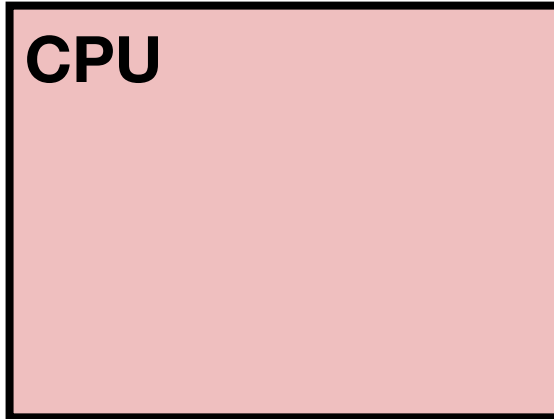
Today: Assembly Programming I: Basics

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

Assembly Code's View of Computer: ISA

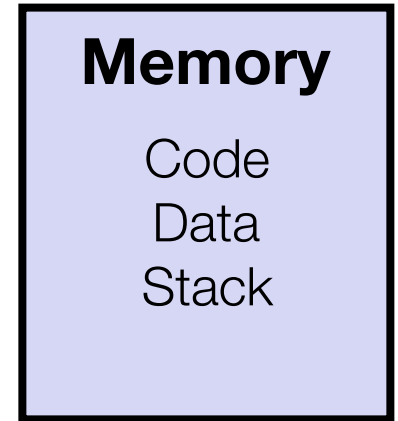
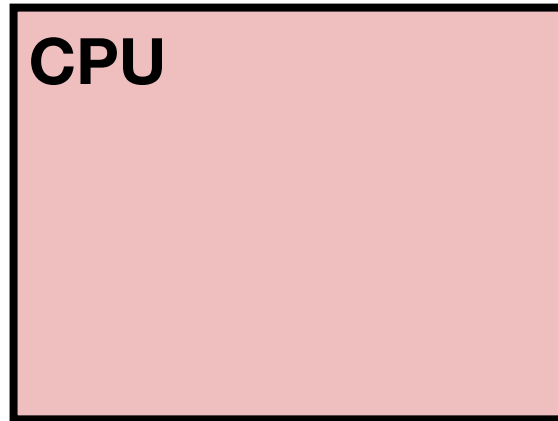
Assembly Code's View of Computer: ISA

Assembly
Programmer's
Perspective
of a Computer



Assembly Code's View of Computer: ISA

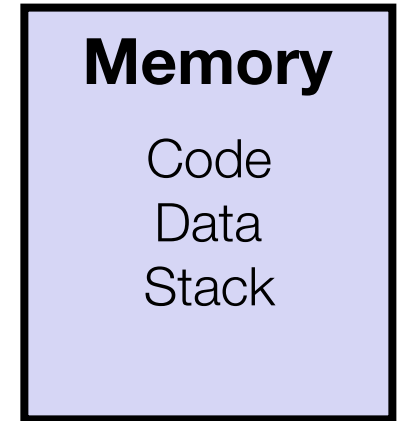
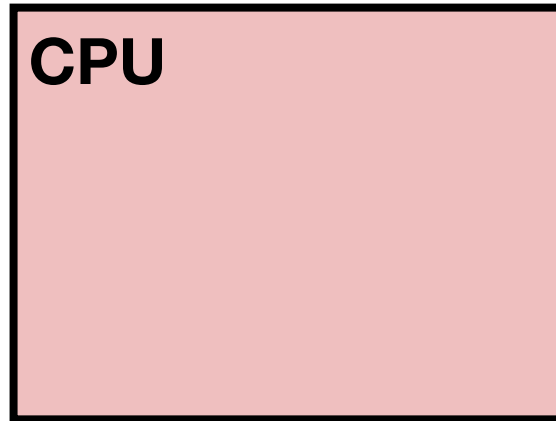
Assembly
Programmer's
Perspective
of a Computer



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

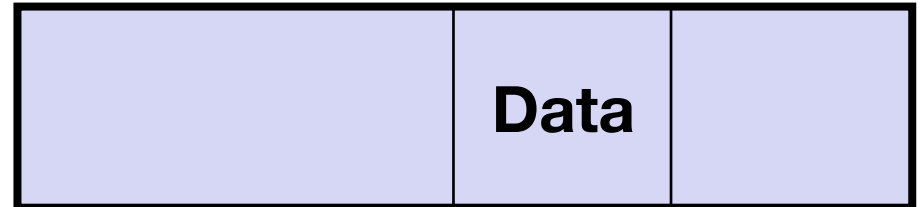
Assembly Code's View of Computer: ISA

Assembly
Programmer's
Perspective
of a Computer



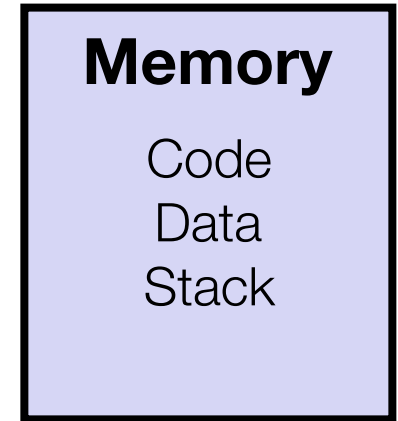
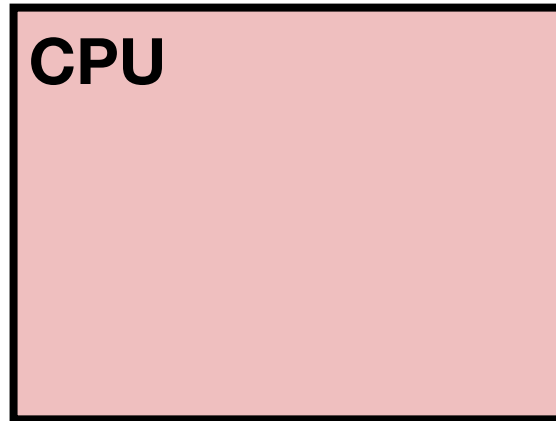
- (Byte Addressable) Memory

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



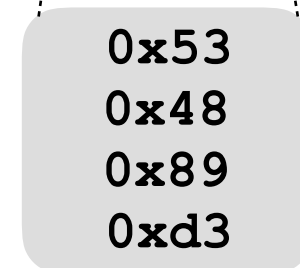
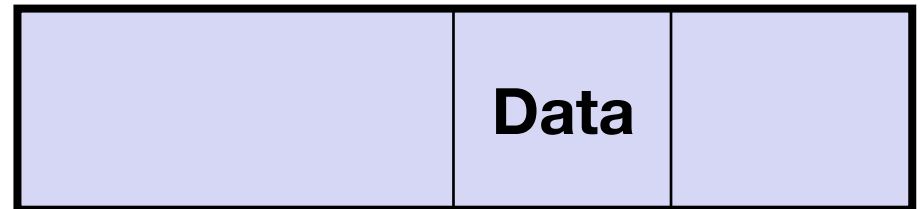
Assembly Code's View of Computer: ISA

Assembly
Programmer's
Perspective
of a Computer



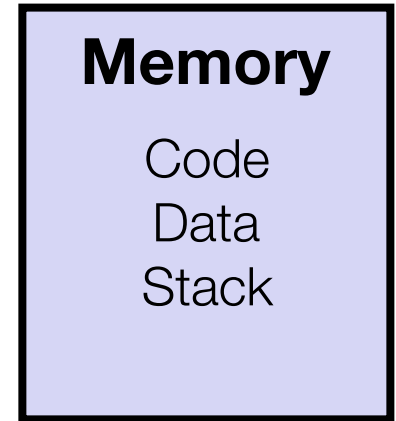
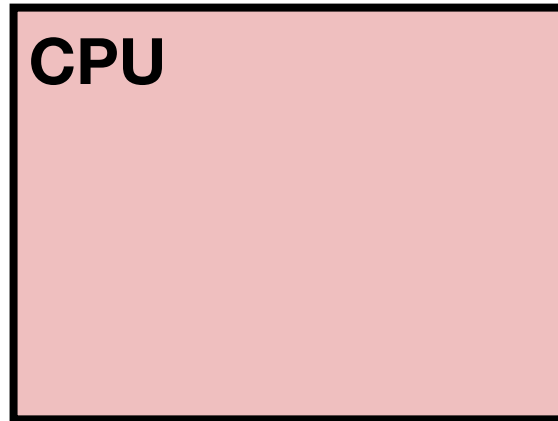
- (Byte Addressable) Memory

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



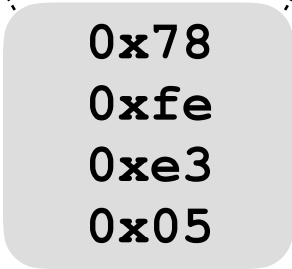
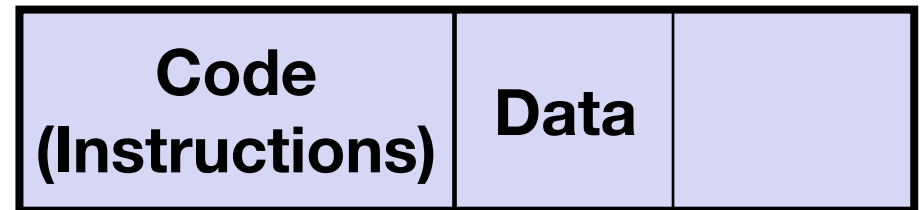
Assembly Code's View of Computer: ISA

Assembly
Programmer's
Perspective
of a Computer



- (Byte Addressable) Memory

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



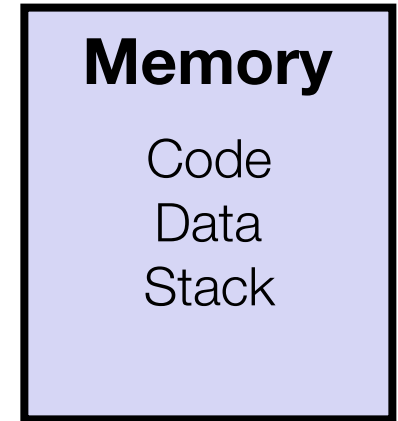
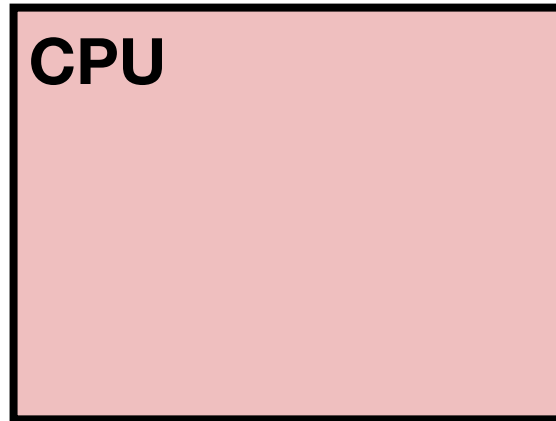
A light gray rounded rectangle containing a list of four hexadecimal values: **0x78**, **0xfe**, **0xe3**, and **0x05**. Dotted lines connect the top-left corner of this box to the bottom-right corner of the 'Code (Instructions)' section, and the top-right corner to the bottom-left corner of the 'Data' section.

Instruction is the fundamental
unit of work.

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

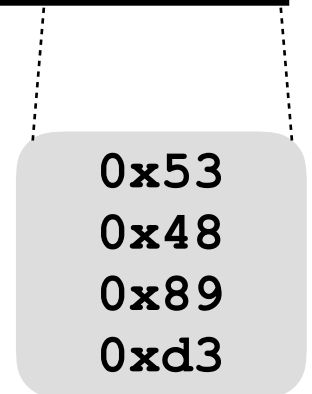
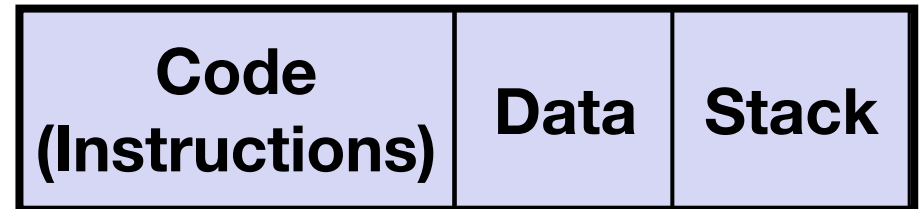
Assembly Code's View of Computer: ISA

Assembly
Programmer's
Perspective
of a Computer



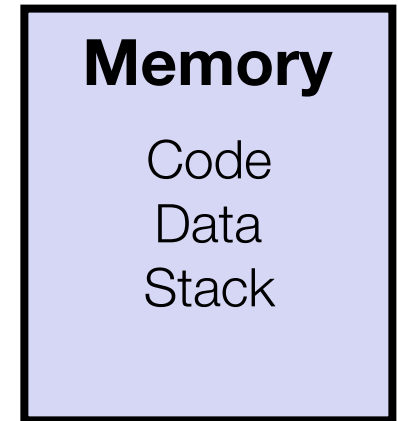
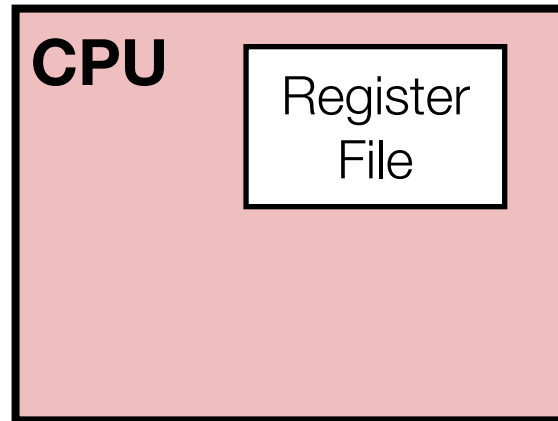
- (Byte Addressable) Memory

- 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 →

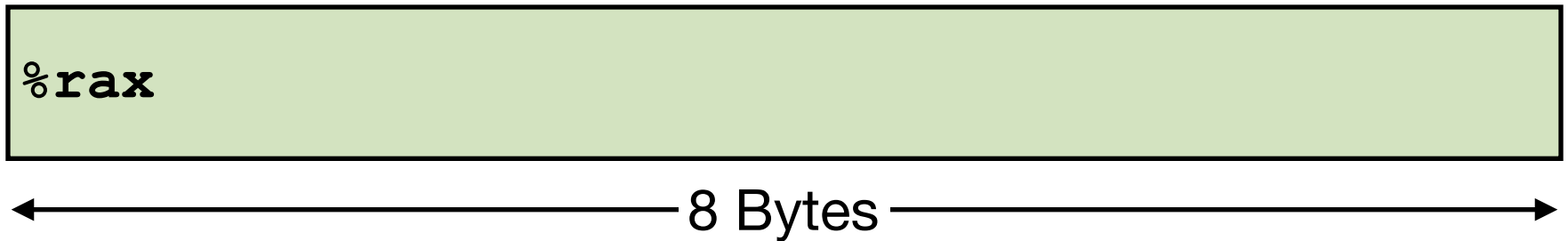
<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 8 bytes)

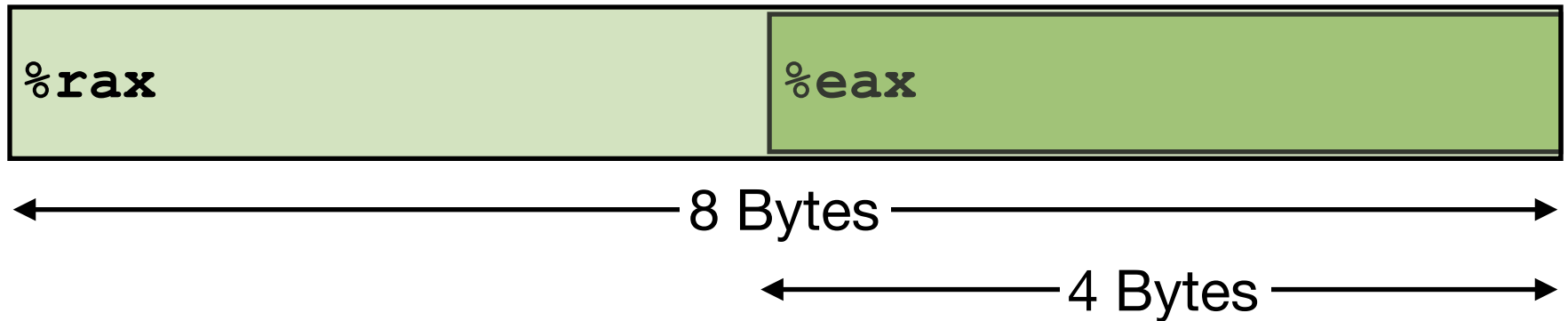
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 8 bytes)



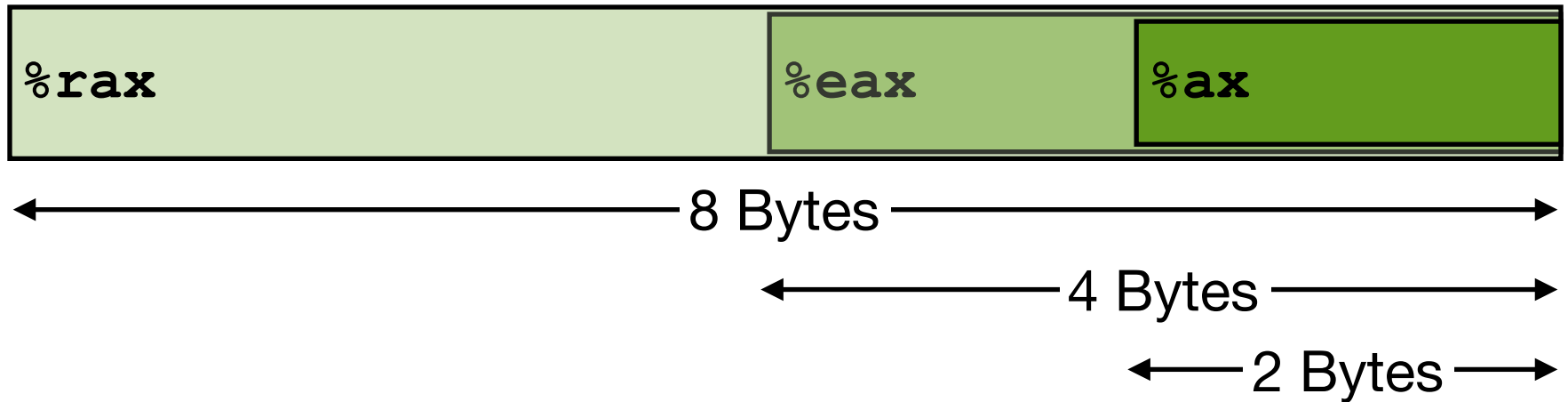
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 8 bytes)



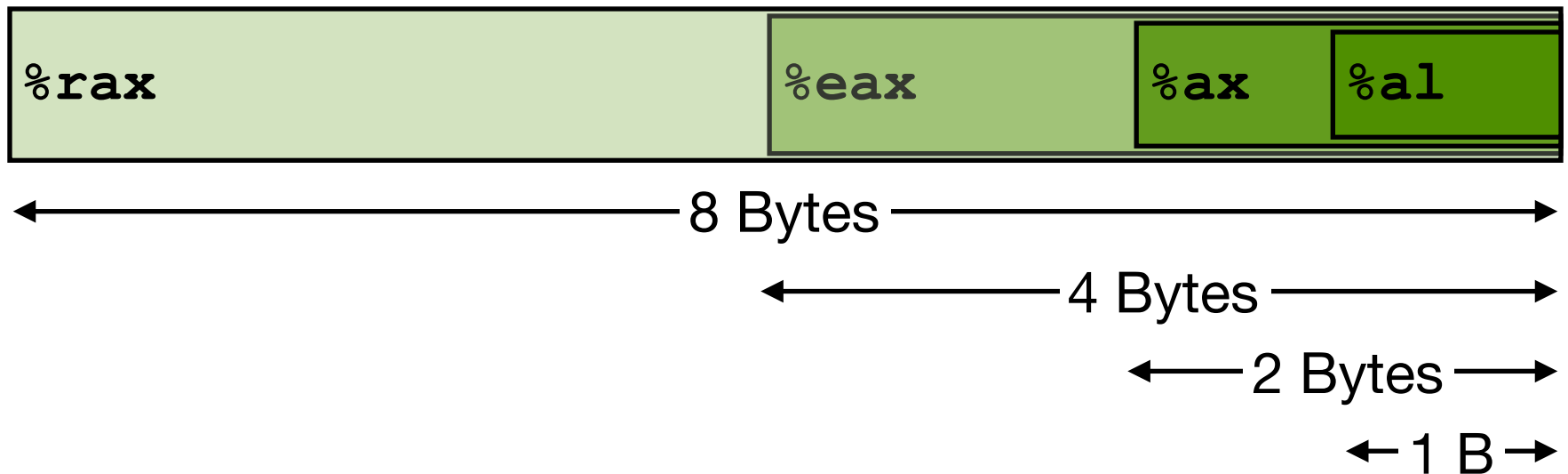
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 8 bytes)



x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 8 bytes)



x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 8 bytes)



← 8 Bytes →

← 4 Bytes →

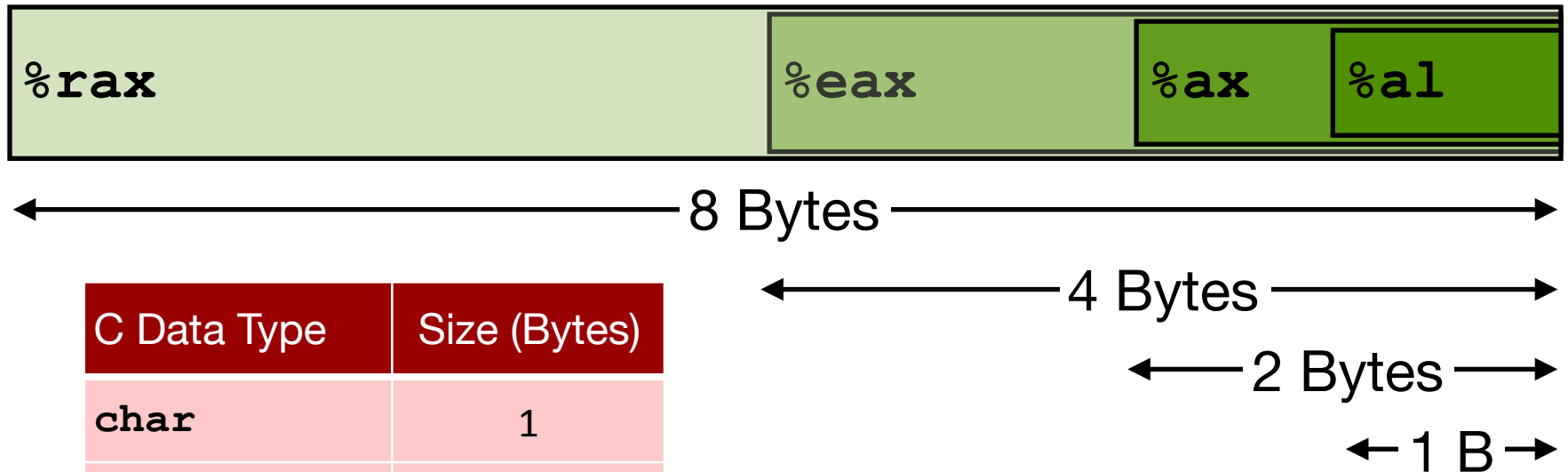
← 2 Bytes →

← 1 B →

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 8 bytes)

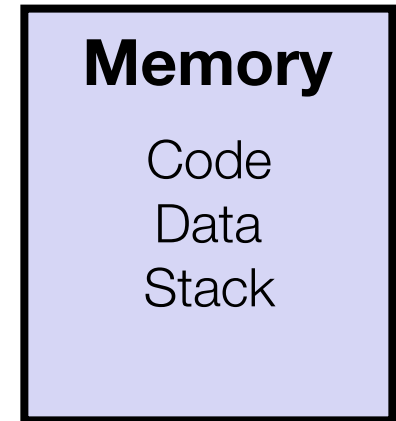
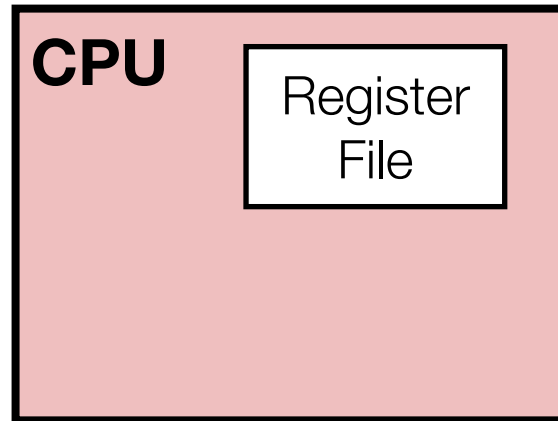


C Data Type	Size (Bytes)
<code>char</code>	1
<code>short</code>	2
<code>int</code>	4
<code>long</code>	8
Pointer	8

Floating point data is stored in a separate set of register file (in 3 lectures...)

Assembly Code's View of Computer: ISA

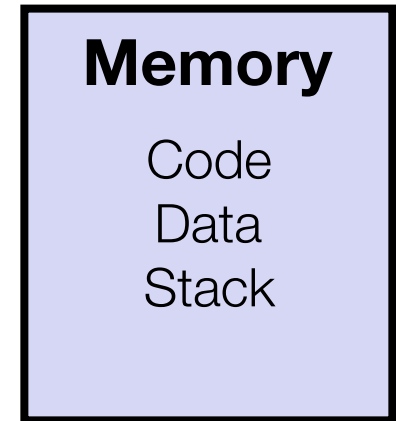
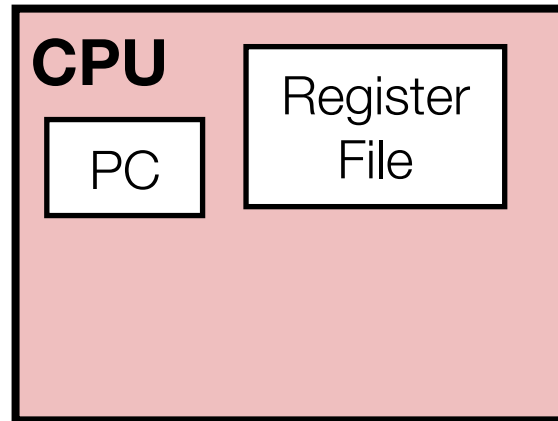
Assembly Programmer's Perspective of a Computer



- (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

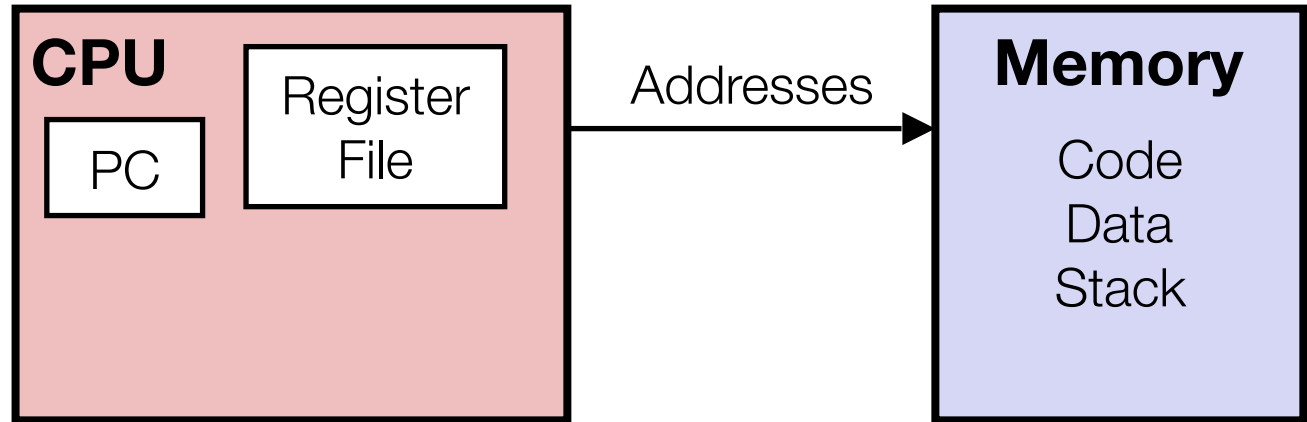
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- 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

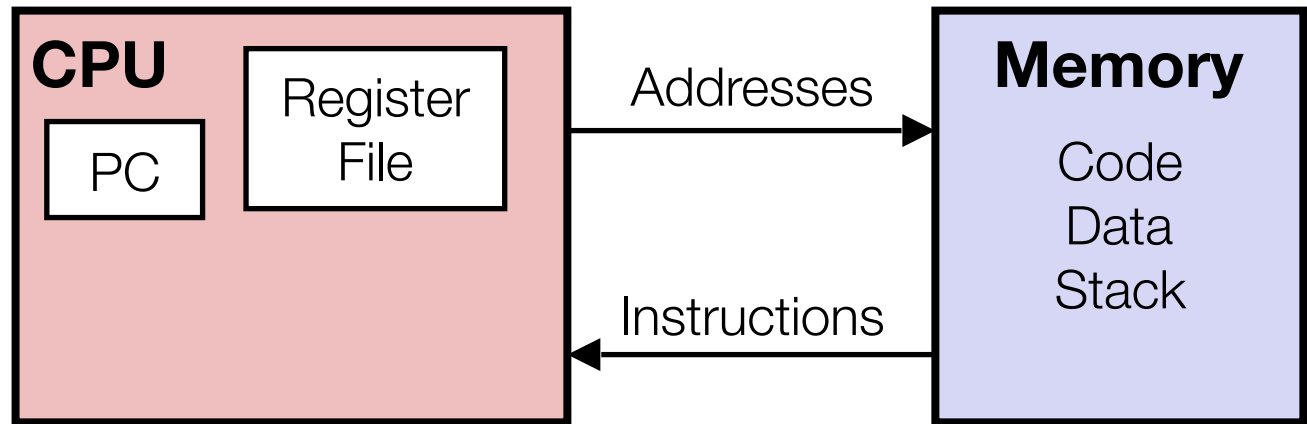
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- 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

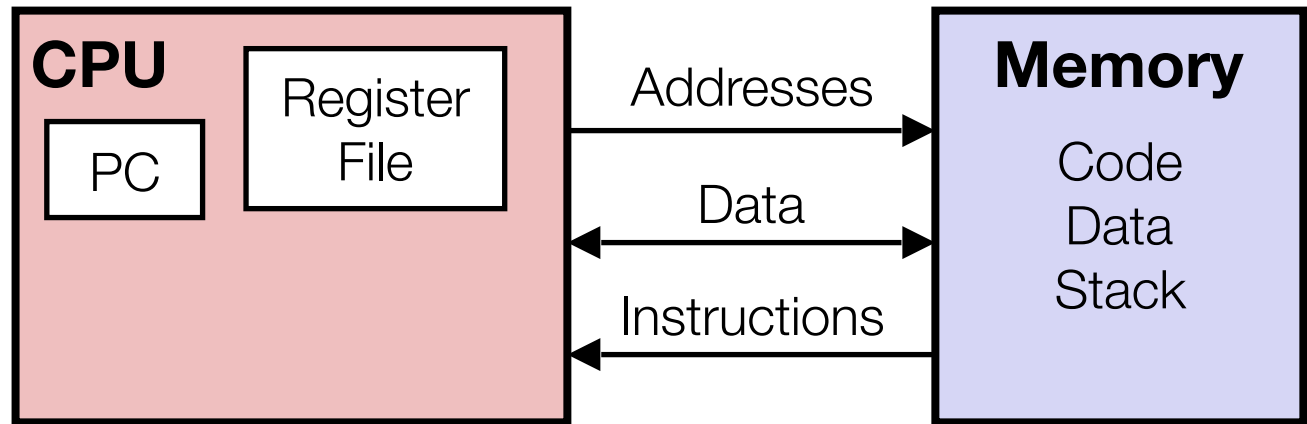
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- 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

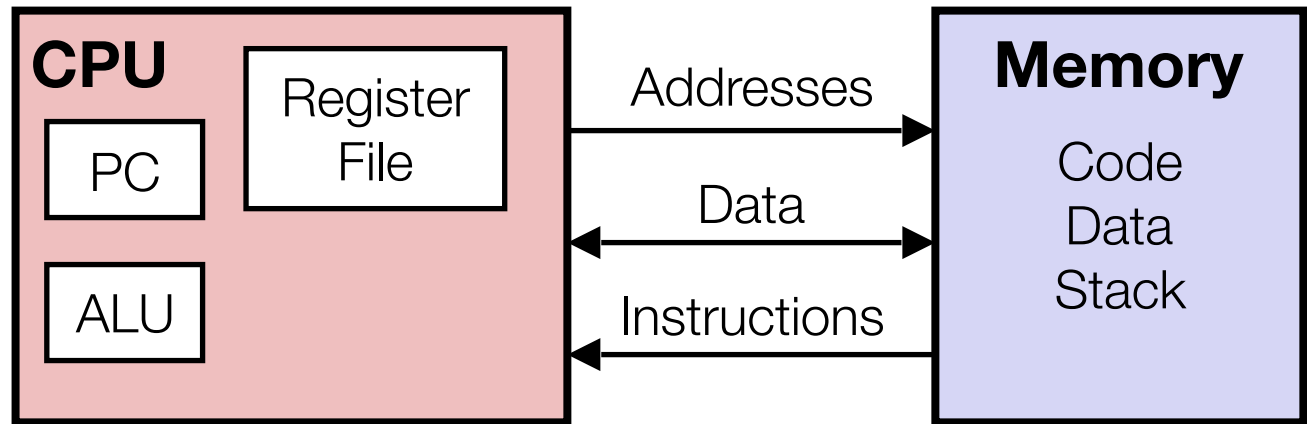
- Faster memory (e.g., 0.5 ns vs. 15 ns)
- Small memory (e.g., 128 B vs. 16 GB)
- Heavily used program data

- 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

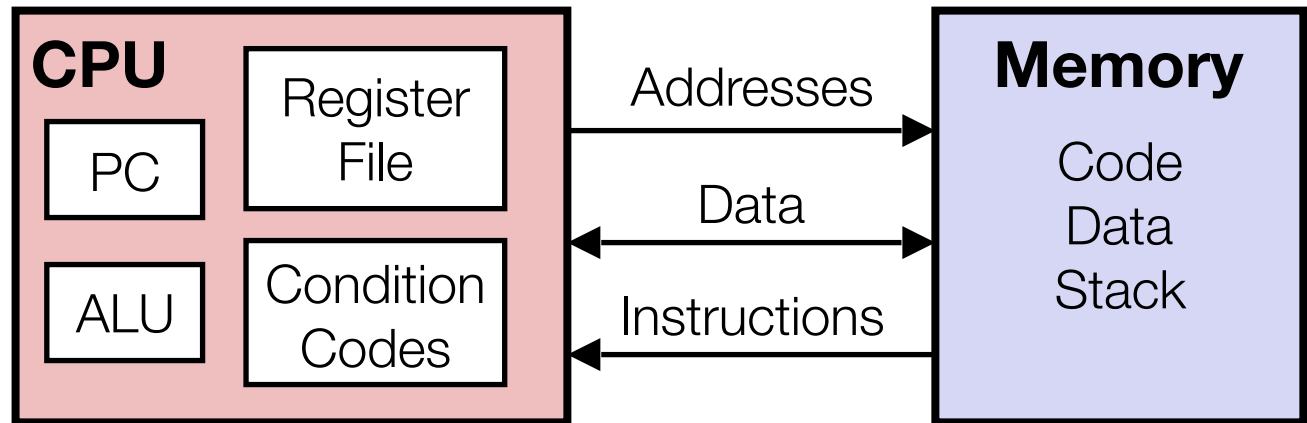
- A special register containing address of next instruction
- Called “RIP” in x86-64

- 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)

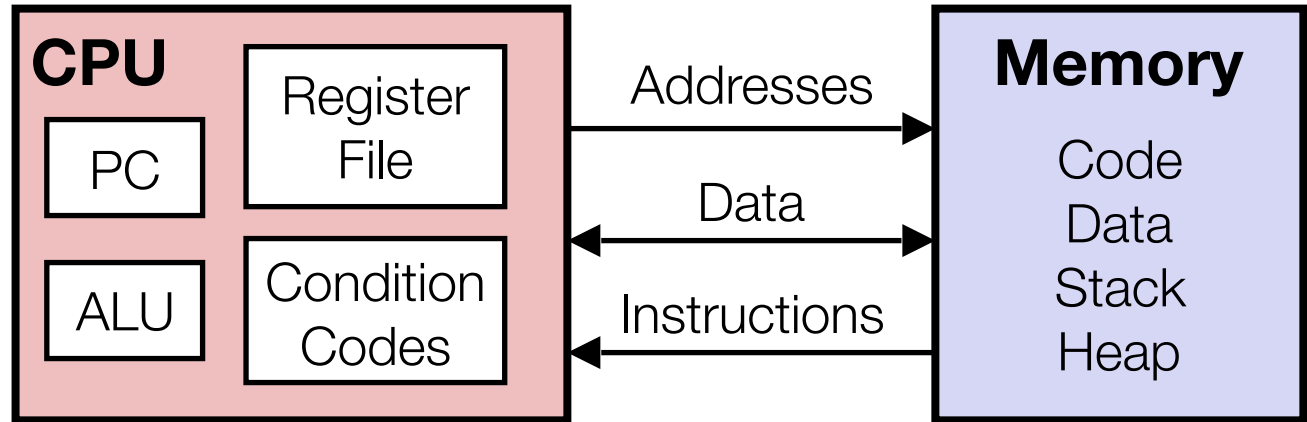
- Where computation happens

- Condition codes

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

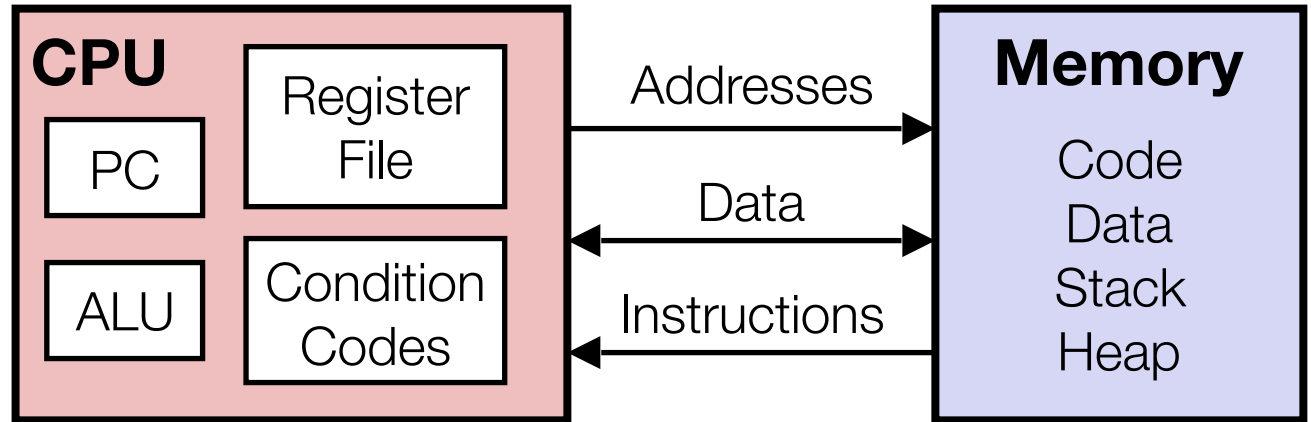
Assembly Program Instructions

Assembly
Programmer's
Perspective
of a Computer



Assembly Program Instructions

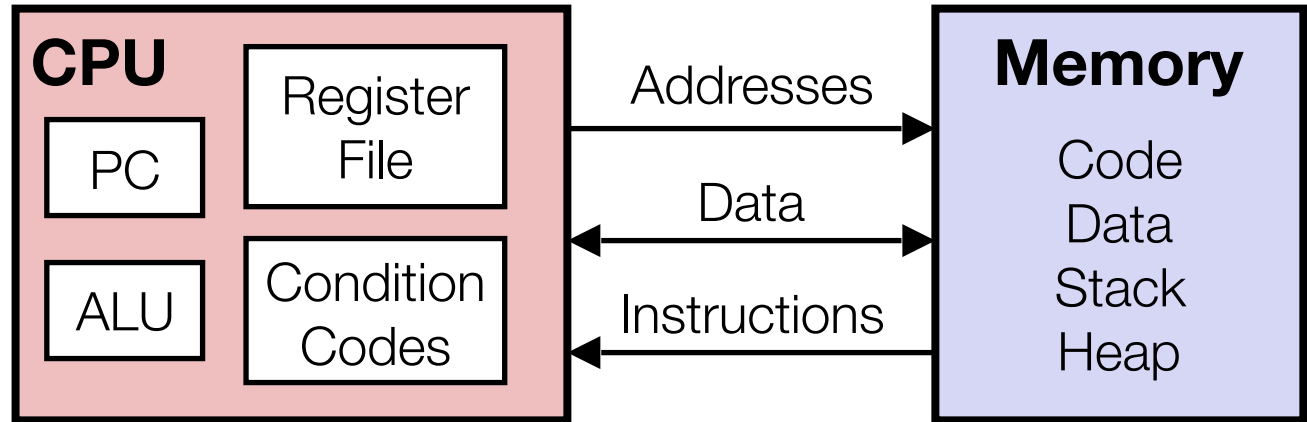
Assembly
Programmer's
Perspective
of a Computer



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

Assembly Program Instructions

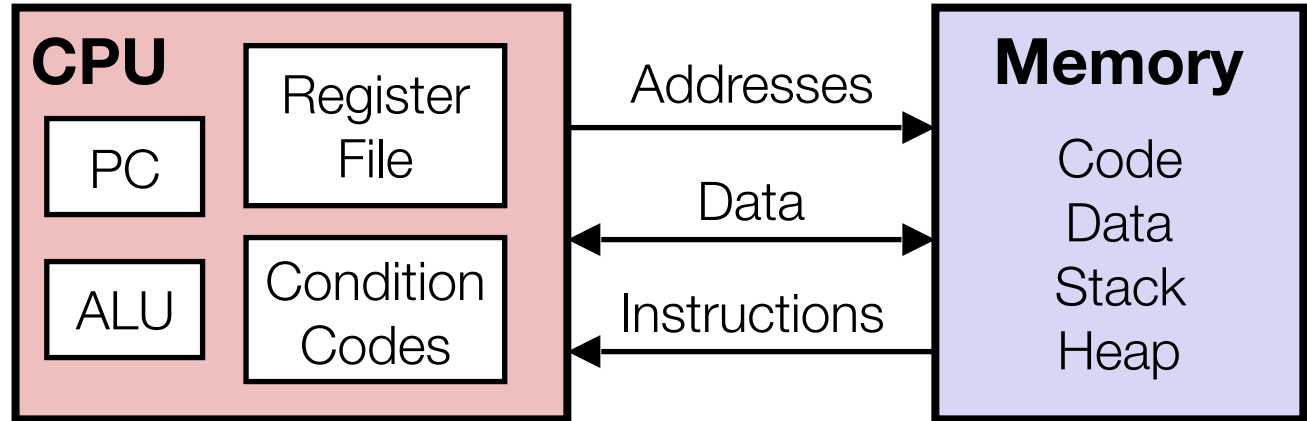
Assembly
Programmer's
Perspective
of a Computer



- *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

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

```
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

Machine Instruction Example

Machine Instruction Example

```
long t;  
long *dest;  
t += *dest;
```

- C Code
 - Add value **t** with value in memory location whose address is **dest** and store the result back to **t**

Machine Instruction Example

```
long t;  
long *dest;  
t += *dest;
```

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

- C Code
 - Add value **t** with value in memory location whose address is **dest** and store the result back to **t**
- Assembly Instruction
 - Operator: Add two 8-byte values
 - Quad words in x86-64 parlance
 - Operands:
 - t: Register %rax
 - dest: Register %rbx
 - *dest: Memory M[%rbx]

Machine Instruction Example

```
long t;  
long *dest;  
t += *dest;
```

Operator

addq %rax, (%rbx)

- C Code

- Add value **t** with value in memory location whose address is **dest** and store the result back to **t**

- Assembly Instruction

- Operator: Add two 8-byte values
 - Quad words in x86-64 parlance
- Operands:
 - t: Register %rax
 - dest: Register %rbx
 - *dest: Memory M[%rbx]

Machine Instruction Example

```
long t;  
long *dest;  
t += *dest;
```

Operand(s)

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

- C Code

- Add value **t** with value in memory location whose address is **dest** and store the result back to **t**

- Assembly Instruction


- Operator: Add two 8-byte values
 - Quad words in x86-64 parlance
- Operands:
 - t: Register %rax
 - dest: Register %rbx
 - *dest: Memory M[%rbx]

Machine Instruction Example

```
long t;  
long *dest;  
t += *dest;
```

- C Code
 - Add value **t** with value in memory location whose address is **dest** and store the result back to **t**
- Assembly Instruction
 - Operator: Add two 8-byte values
 - Quad words in x86-64 parlance
 - Operands:
 - t: Register `%rax`
 - dest: Register `%rbx`
 - *dest: Memory `M[%rbx]`

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



Machine Instruction Example

```
long t;  
long *dest;  
t += *dest;
```

- C Code

- Add value **t** with value in memory location whose address is **dest** and store the result back to **t**

- Assembly Instruction

- Operator: Add two 8-byte values
 - Quad words in x86-64 parlance

- Operands:

t: Register %rax

dest: Register %rbx

*dest: Memory M[%rbx]

addq %rax, (%rbx)

t

dest

Machine Instruction Example

```
long t;  
long *dest;  
t += *dest;
```

- C Code

- Add value **t** with value in memory location whose address is **dest** and store the result back to **t**

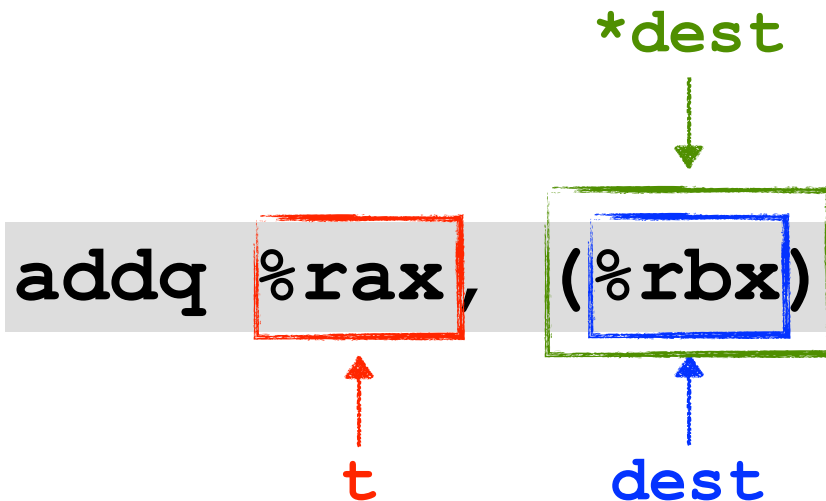
- Assembly Instruction

- Operator: Add two 8-byte values
 - Quad words in x86-64 parlance
- Operands:

t: Register %rax

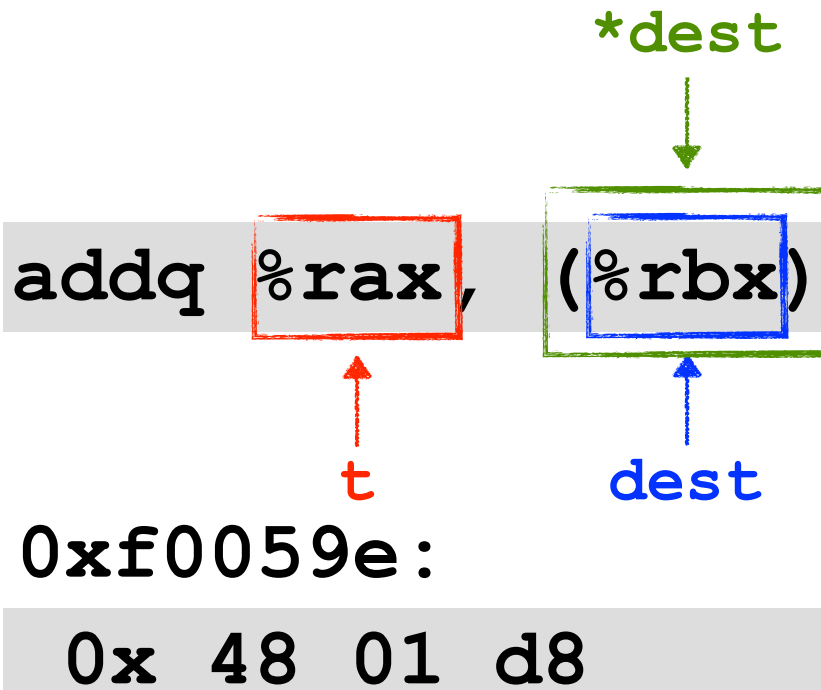
dest: Register %rbx

*dest: Memory M[%rbx]



Machine Instruction Example

```
long t;  
long *dest;  
t += *dest;
```



- C Code

- Add value `t` with value in memory location whose address is `dest` and store the result back to `t`

- Assembly Instruction

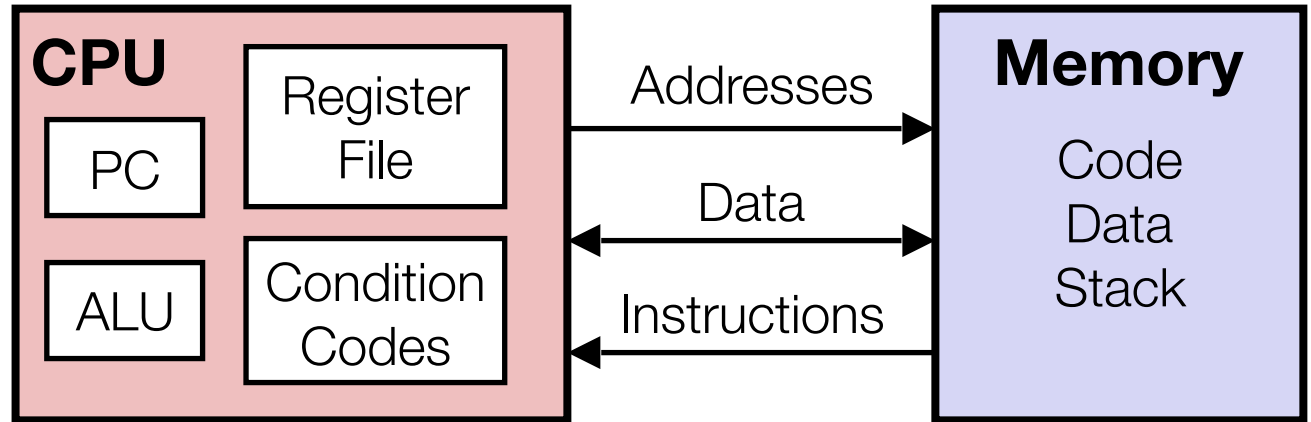
- Operator: Add two 8-byte values
 - Quad words in x86-64 parlance
- Operands:
 - `t`: Register `%rax`
 - `dest`: Register `%rbx`
 - `*dest`: Memory `M[%rbx]`

- Object Code

- 3-byte instruction
- Stored at address `0xf0059e`

Instruction Processing Sequence

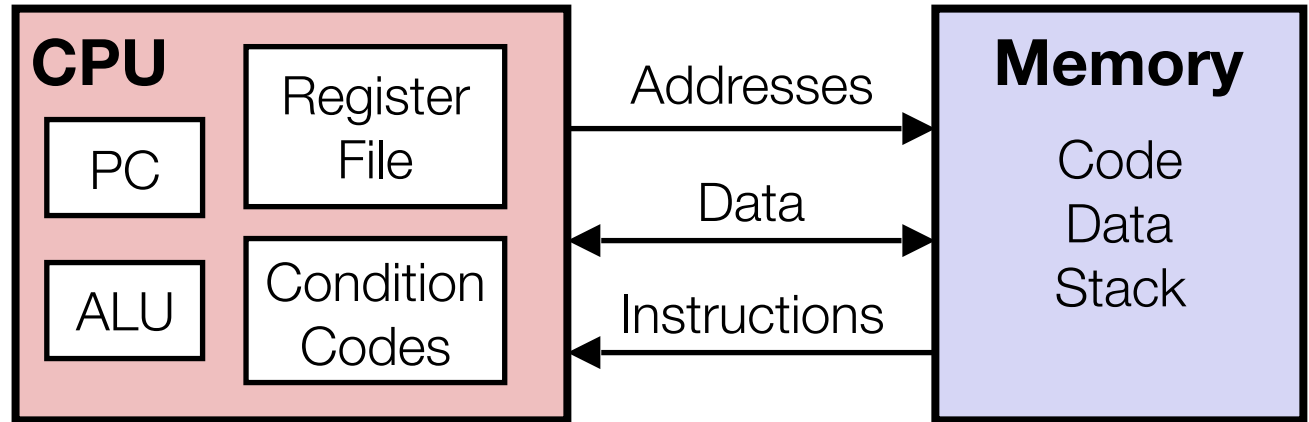
Assembly
Programmer's
Perspective
of a Computer



Fetch Instruction
(According to PC)

Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer

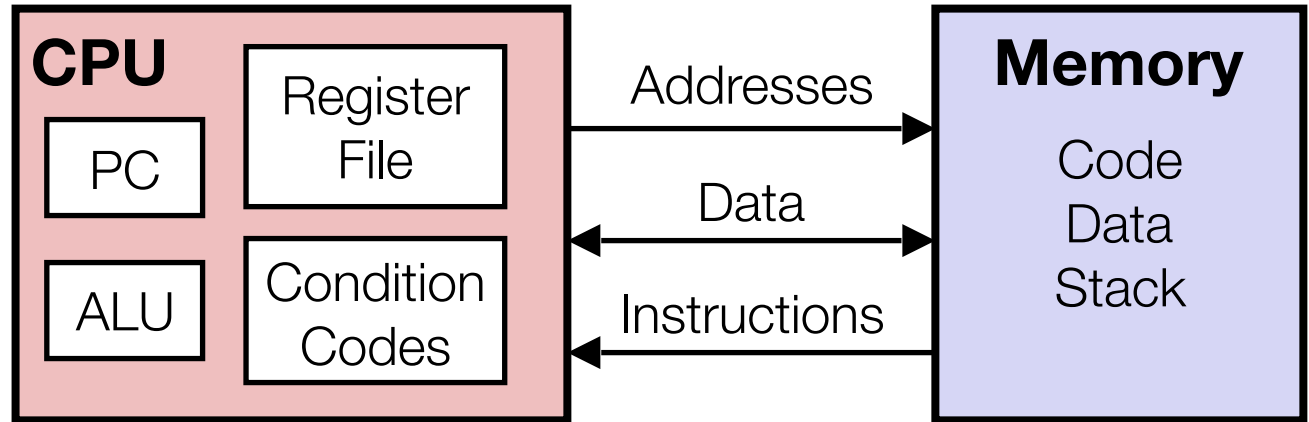


Fetch Instruction
(According to PC)

0x4801d8

Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer

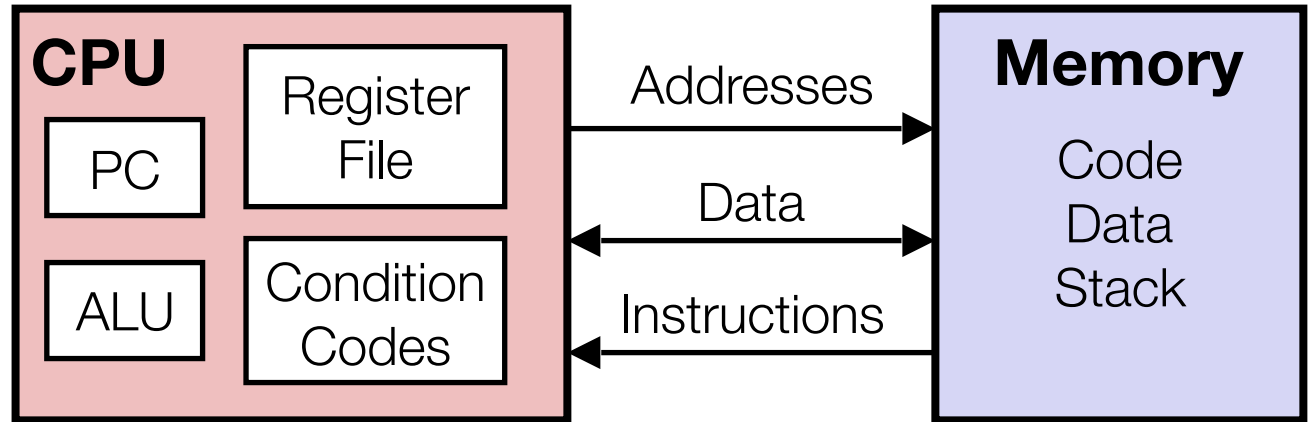


Fetch Instruction
(According to PC) → Decode
Instruction

`addq %rax, (%rbx)`

Instruction Processing Sequence

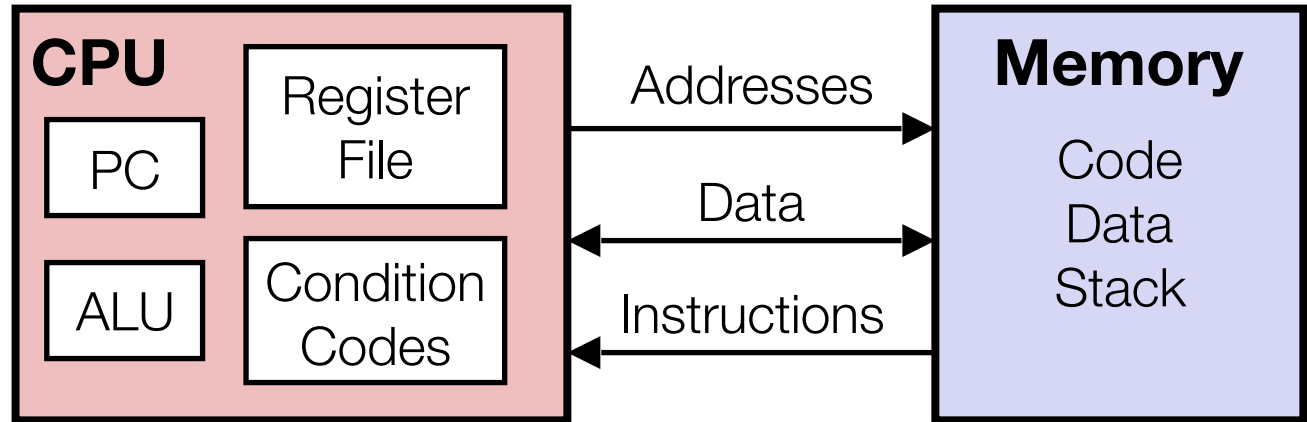
Assembly
Programmer's
Perspective
of a Computer



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

Instruction Processing Sequence

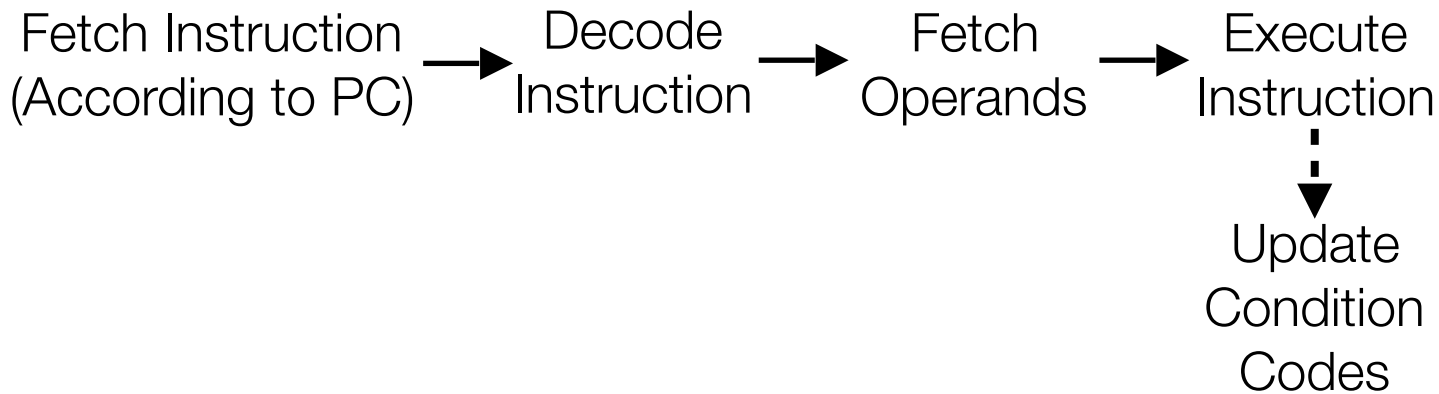
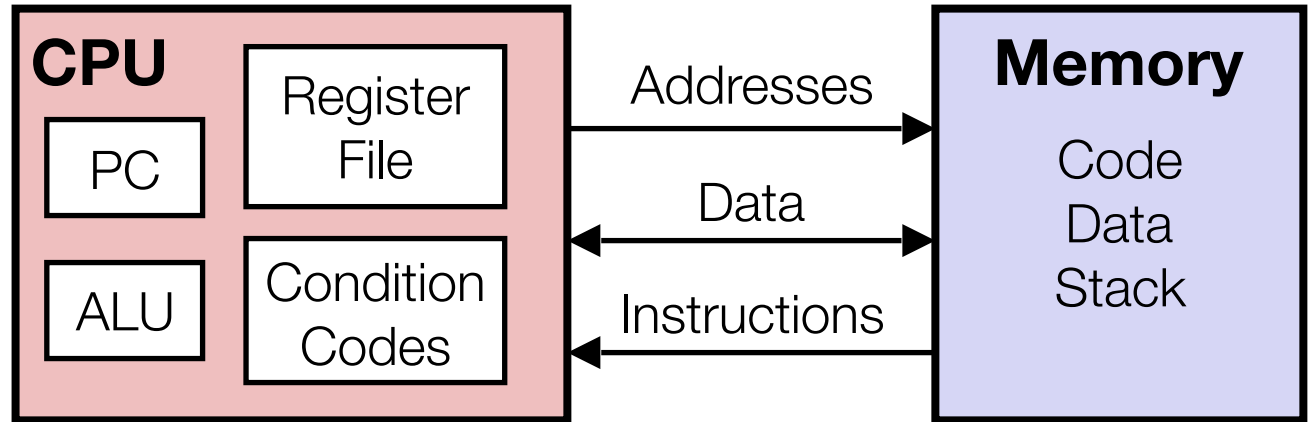
Assembly
Programmer's
Perspective
of a Computer



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

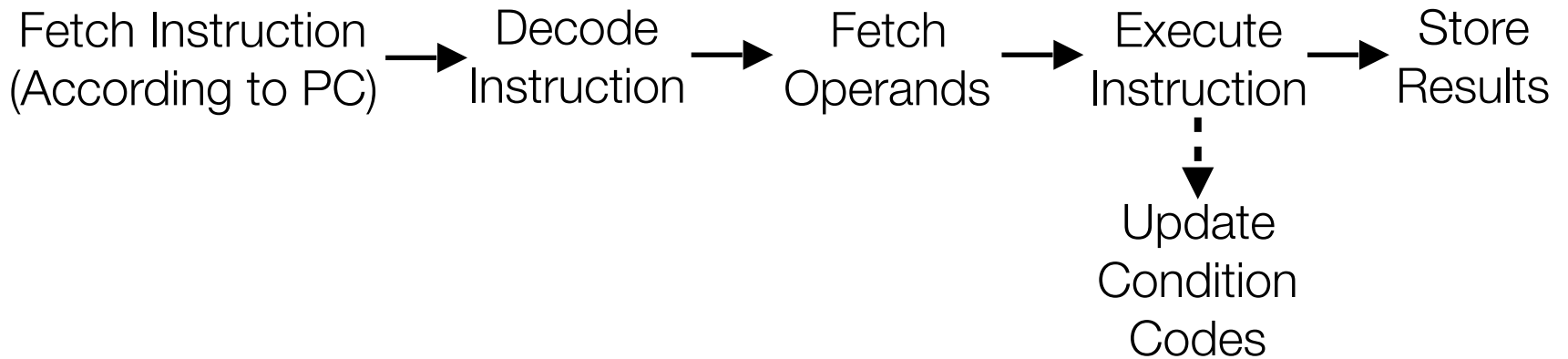
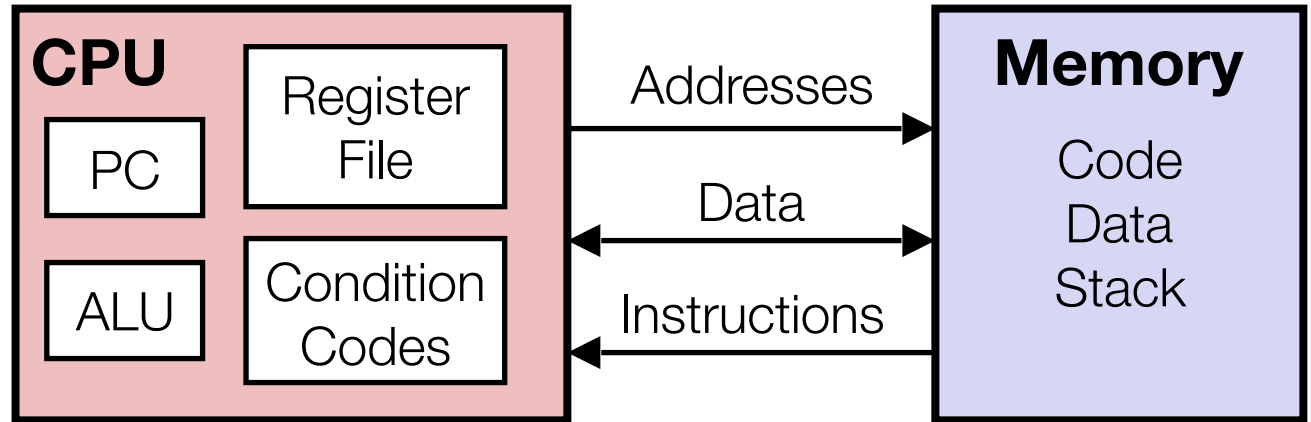
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



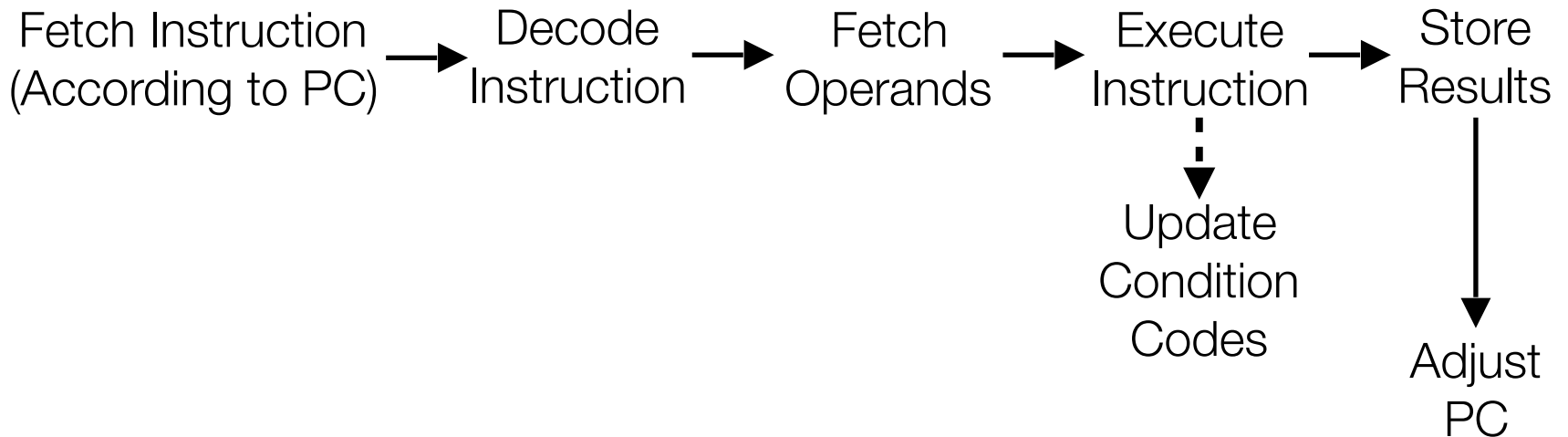
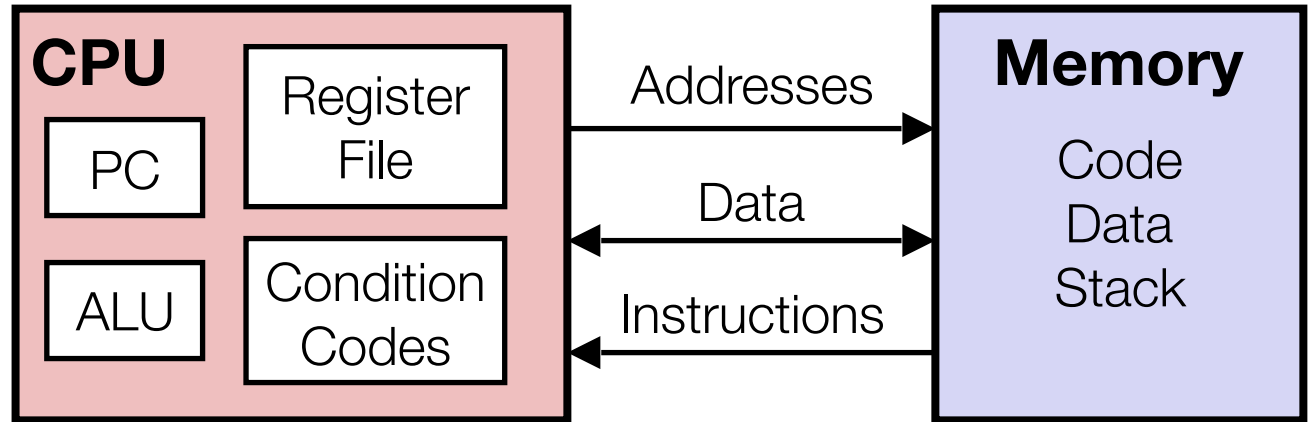
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



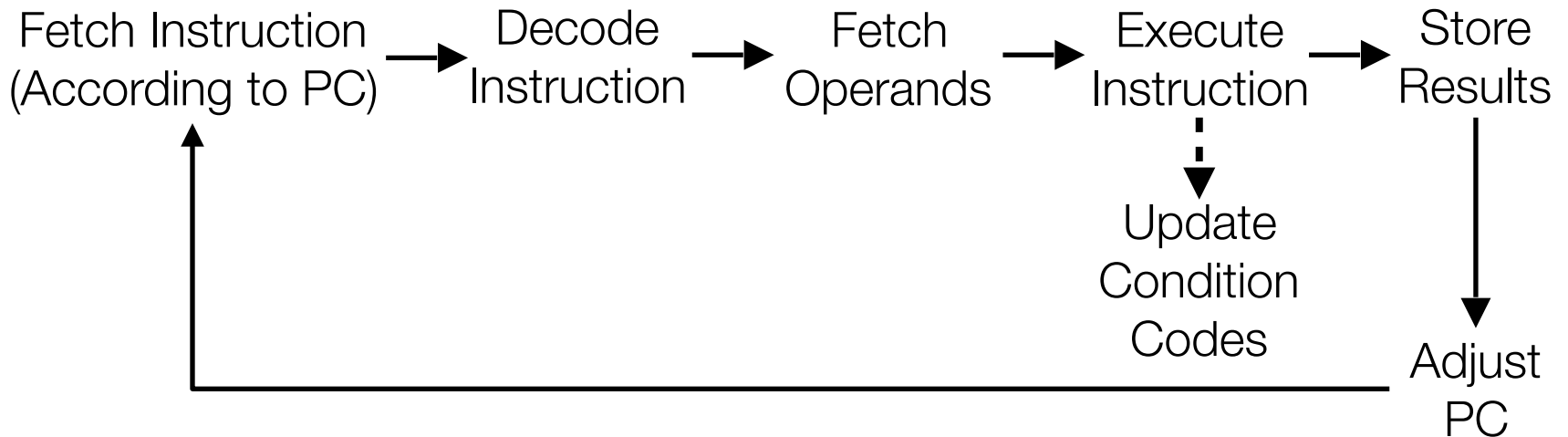
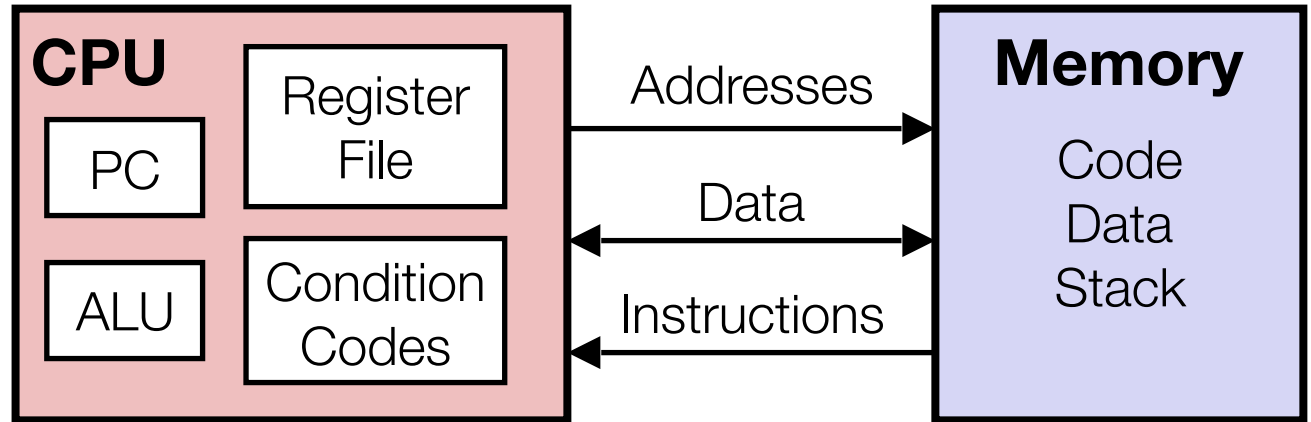
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



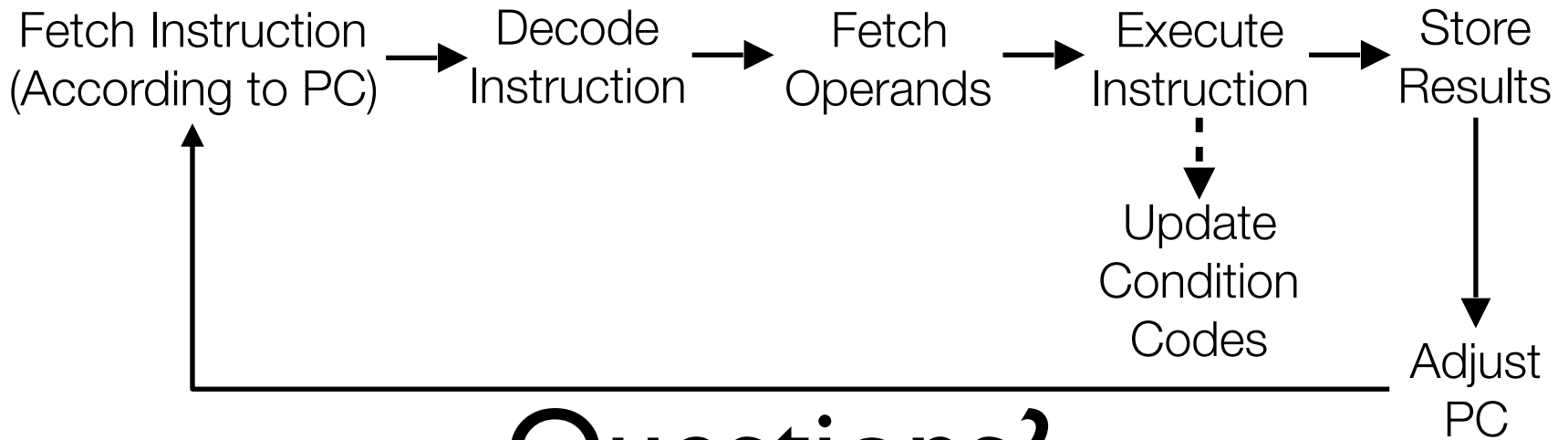
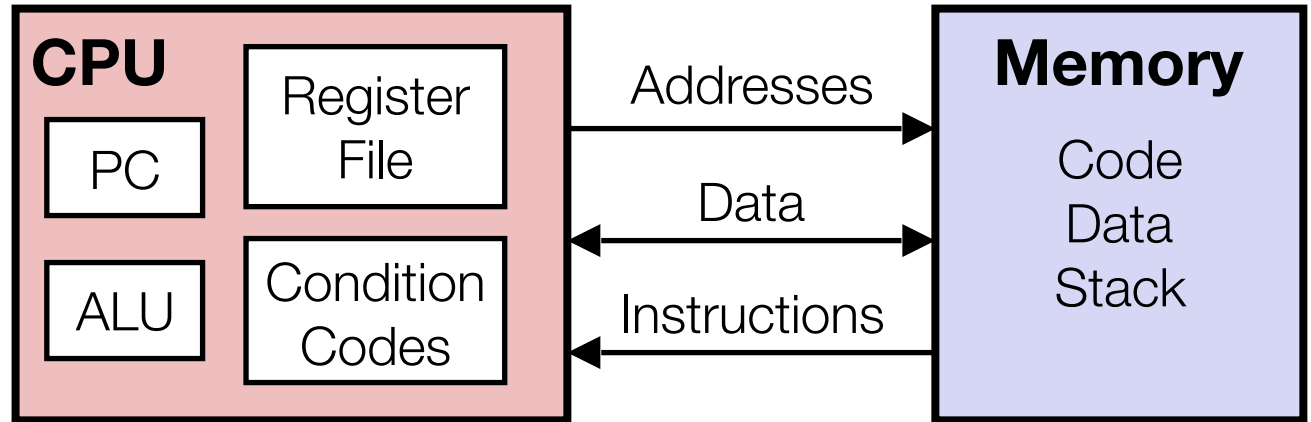
Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



Instruction Processing Sequence

Assembly
Programmer's
Perspective
of a Computer



Questions?

Today: Assembly Programming I: Basics

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

Data Movement Instructions

movq *Source, Dest*

Data Movement Instructions

`movq` *Source, Dest*

Operator Operands

Data Movement Instructions

`movq` *Source, Dest*

Operator Operands

- **Register:** One of 16 integer registers
 - Example: `%rax`, `%r13`
 - But `%rsp` reserved for special use

Data Movement Instructions

`movq` *Source, Dest*

Operator Operands

- **Register:** One of 16 integer registers
 - 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

Data Movement Instructions

movq *Source, Dest*

Operator Operands

- **Register:** One of 16 integer registers
 - 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
- **Memory:** 8 consecutive bytes in memory at given address
 - Simplest example: `(%rax)`
 - How to obtain the address is called “addressing mode” (later...)

movq Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	$\left\{ \begin{array}{l} \text{Reg} \\ \text{Mem} \end{array} \right.$		
	Reg	$\left\{ \begin{array}{l} \text{Reg} \\ \text{Mem} \end{array} \right.$		
	Mem	Reg		

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

movq Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	$\left\{ \begin{array}{l} \text{Reg} \\ \text{Mem} \end{array} \right.$	movq \$0x4, %rax	
	Reg	$\left\{ \begin{array}{l} \text{Reg} \\ \text{Mem} \end{array} \right.$		
	Mem	Reg		

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

movq Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	$\begin{cases} \text{Reg} \\ \text{Mem} \end{cases}$	movq \$0x4, %rax	temp = 0x4;
	Reg	$\begin{cases} \text{Reg} \\ \text{Mem} \end{cases}$		
	Mem	Reg		

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

movq Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4, %rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	
	Reg	Reg		
		Mem		
	Mem	Reg		

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

movq Operand Combinations

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

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

movq Operand Combinations

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

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

movq Operand Combinations

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

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

movq Operand Combinations

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

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

movq Operand Combinations

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

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

movq Operand Combinations

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

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

movq Operand Combinations

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

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

Memory Addressing Modes

- An addressing mode specifies:
 - how to calculate the effective memory address of an operand
 - by using information held in registers and/or constants

Memory Addressing Modes

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

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

Memory Addressing Modes

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

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

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

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

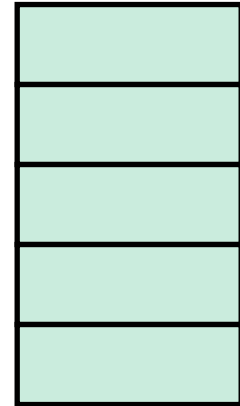
Example of Simple Addressing Modes

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

Example of Simple Addressing Modes

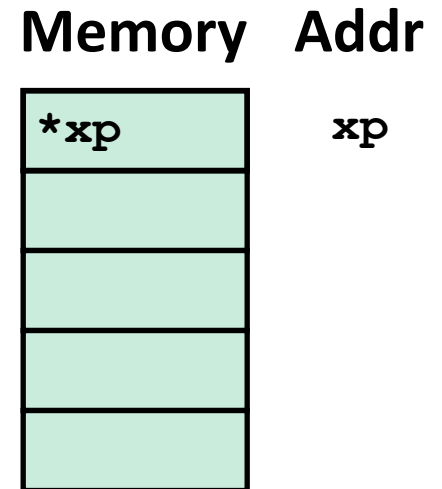
```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Memory



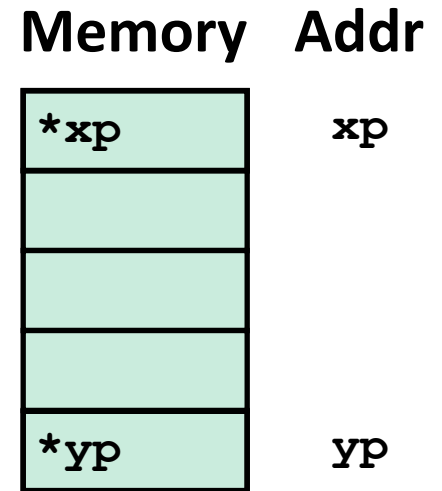
Example of Simple Addressing Modes

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



Example of Simple Addressing Modes

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



Example of Simple Addressing Modes

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

Registers

%rdi	xp
%rsi	yp
%rax	
%rdx	

Memory Addr

*xp	xp
*yp	yp

Example of Simple Addressing Modes

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

Registers

%rdi	xp
%rsi	yp
%rax	
%rdx	

Memory Addr

*xp	xp
*yp	yp

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Example of Simple Addressing Modes

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

Registers

%rdi	xp
%rsi	yp
%rax	
%rdx	

Memory Addr

*xp	xp
*yp	yp

How Does This Work?

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Understanding `Swap()`

Registers

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	
<code>%rdx</code>	

Memory

123
456

Addr

<code>0x120</code>	<code>xp</code>
<code>0x118</code>	
<code>0x110</code>	
<code>0x108</code>	
<code>0x100</code>	<code>yp</code>

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Understanding `Swap()`

Registers

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	
<code>%rdx</code>	

Memory

123
456

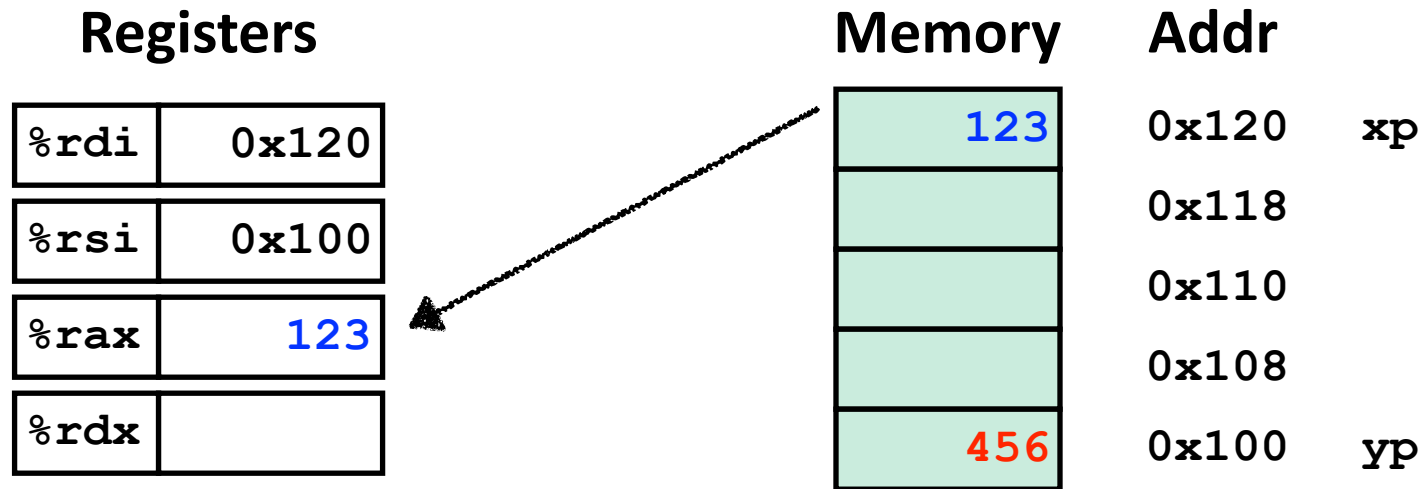
Addr

<code>0x120</code>	<code>xp</code>
<code>0x118</code>	
<code>0x110</code>	
<code>0x108</code>	
<code>0x100</code>	<code>yp</code>

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Understanding `Swap()`



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Understanding `Swap()`

Registers

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	<code>123</code>
<code>%rdx</code>	

Memory

<code>123</code>
<code>456</code>

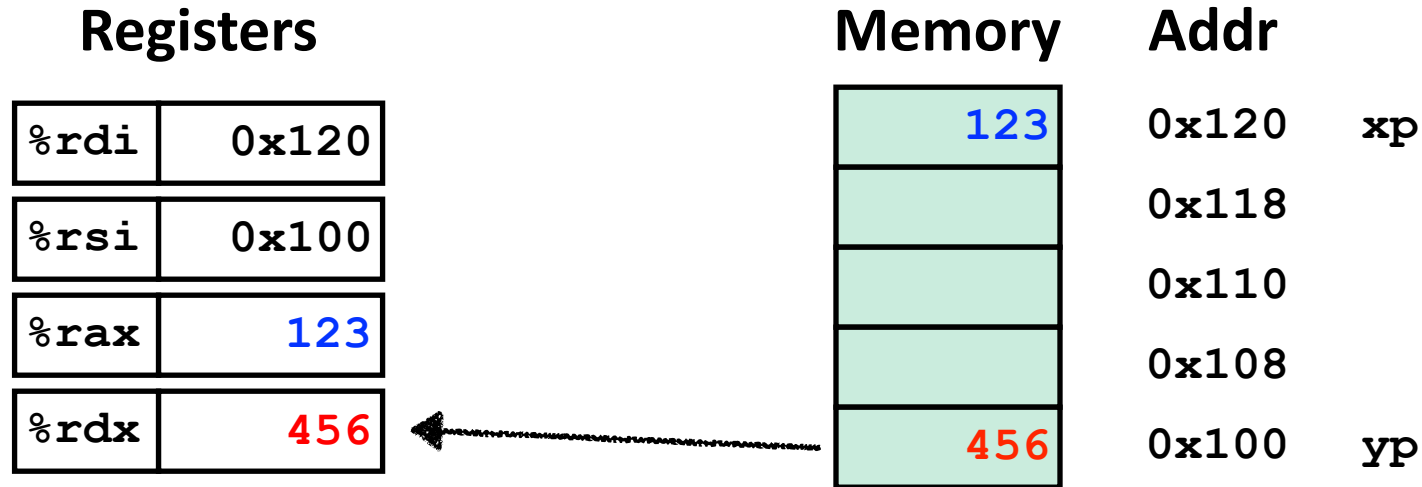
Addr

<code>0x120</code>	<code>xp</code>
<code>0x118</code>	
<code>0x110</code>	
<code>0x108</code>	
<code>0x100</code>	<code>yp</code>

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Understanding `Swap()`



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```


Understanding `Swap()`

Registers

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	<code>123</code>
<code>%rdx</code>	<code>456</code>

Memory

<code>123</code>
<code>456</code>

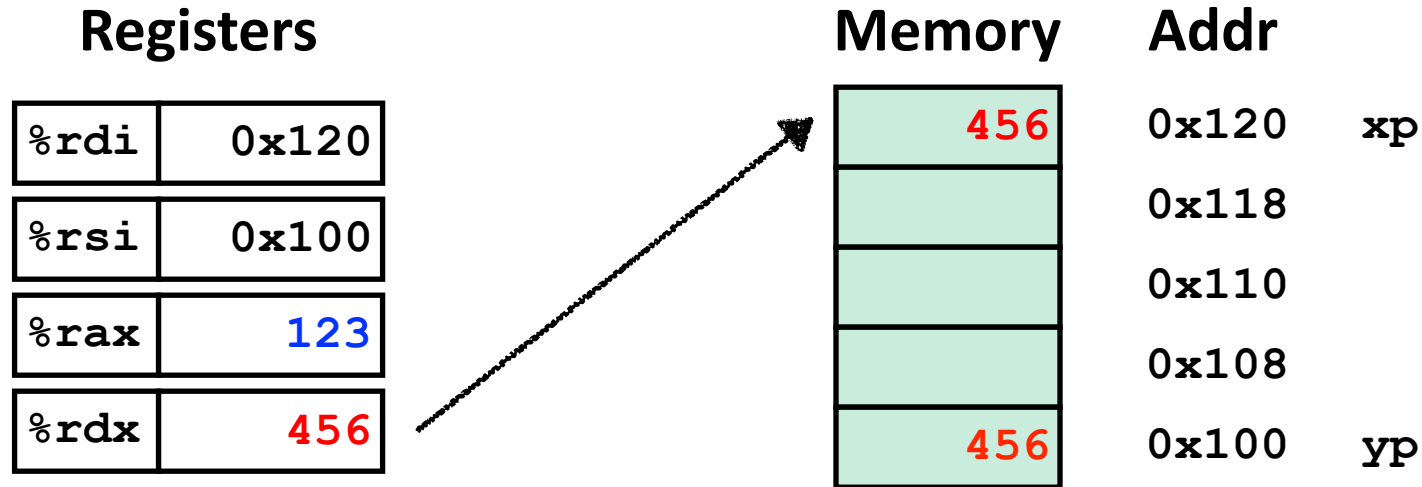
Addr

<code>0x120</code>	<code>xp</code>
<code>0x118</code>	
<code>0x110</code>	
<code>0x108</code>	
<code>0x100</code>	<code>yp</code>

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Understanding `Swap()`



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Understanding `Swap()`

Registers

<code>%rdi</code>	0x120
<code>%rsi</code>	0x100
<code>%rax</code>	123
<code>%rdx</code>	456

Memory

456
456

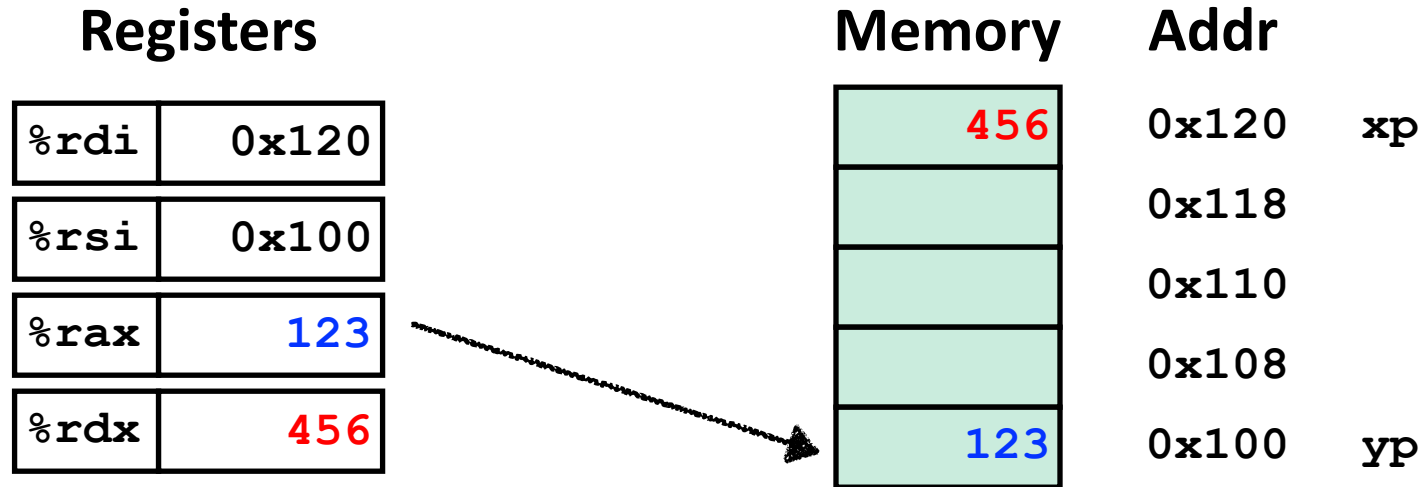
Addr

0x120	<code>xp</code>
0x118	
0x110	
0x108	
0x100	<code>yp</code>

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Understanding `Swap()`



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```