

CSC 252: Computer Organization

Spring 2026: Lecture 5

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Programming Assignment 1 is out
 - Details: <https://cs.rochester.edu/courses/252/spring2026/labs/assignment1.html>
 - Due on Feb. 11, 11:59 PM
 - You have 3 slip days
- TA Office Hours
 - Han Yan, Friday 11:00am-12:00pm at WH 3205
 - Junjie Zhao, Monday 4:00pm-5:00pm at WH 2215
 - Aarav Ahuja, Thursdays 1:00pm-2:00pm at WH 2215
 - Hannah Davisdon, Tuesdays 12:00am-1:00pm at WH 2215

Announcement

- Programming assignment 1 is in C language. Seek help from TAs.
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.
- When emailing TAs, email them all.

Floating Point Multiplication

- $(-1)^{s1} M1 2^{E1} \times (-1)^{s2} M2 2^{E2}$
- Exact Result: $(-1)^s M 2^E$
 - Sign s : $s1 \wedge s2$
 - Significand M : $M1 \times M2$
 - Exponent E : $E1 + E2$
- Fixing
 - If $M \geq 2$, shift M right, increment E
 - If E out of range, overflow
 - Round M to fit frac precision
- Implementation
 - Biggest chore is multiplying significands

Mathematical Properties of FP Mult

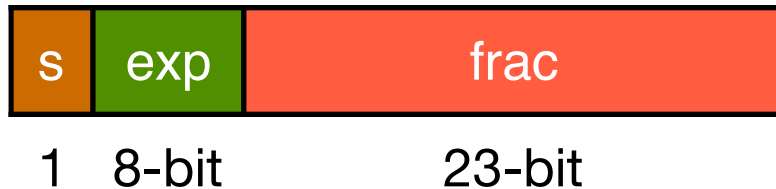
- Multiplication Commutative? **Yes**
- Multiplication is Associative? **No**
 - Possibility of overflow, inexactness of rounding
 - Ex: $(1e20 * 1e20) * 1e-20 = \text{inf}$, $1e20 * (1e20 * 1e-20) = 1e20$
- 1 is multiplicative identity? **Yes**
- Multiplication distributes over addition? **No**
 - Possibility of overflow, inexactness of rounding
 - $1e20 * (1e20 - 1e20) = 0.0$, $1e20 * 1e20 - 1e20 * 1e20 = \text{NaN}$
- Monotonicity: $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$? **Almost**
 - Except for infinities & NaNs

Today: Floating Point

- Background: Fractional binary numbers and fixed-point
- Floating point representation
- IEEE 754 standard
- Rounding, addition, multiplication
- **Floating point in C**
- Summary

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	$(2 - 2^{-52}) \times 2^{1023}$	$\sim 1.8 \times 10^{308}$

- 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 in C

64-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	$2^{31} - 1$	$\sim 9.2 \times 10^{18}$
float	32	$(2 - 2^{-23}) \times 2^{127}$	$\sim 3.4 \times 10^{38}$
double	64	$(2 - 2^{-52}) \times 2^{1023}$	$\sim 1.8 \times 10^{308}$

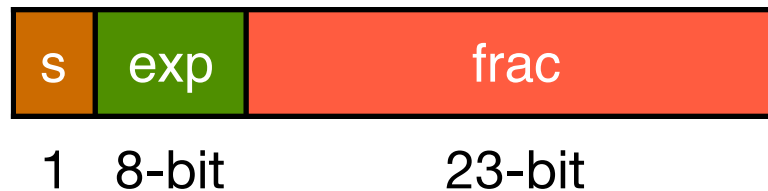
Floating Point Conversions/Casting in C

- **double/float \rightarrow int**

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

- **int \rightarrow float**

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



- **int \rightarrow double**

- Exact conversion



Today: Floating Point

- Background: Fractional binary numbers and fixed-point
- Floating point representation
- IEEE 754 standard
- Rounding, addition, multiplication
- Floating point in C
- **Summary**

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

- 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

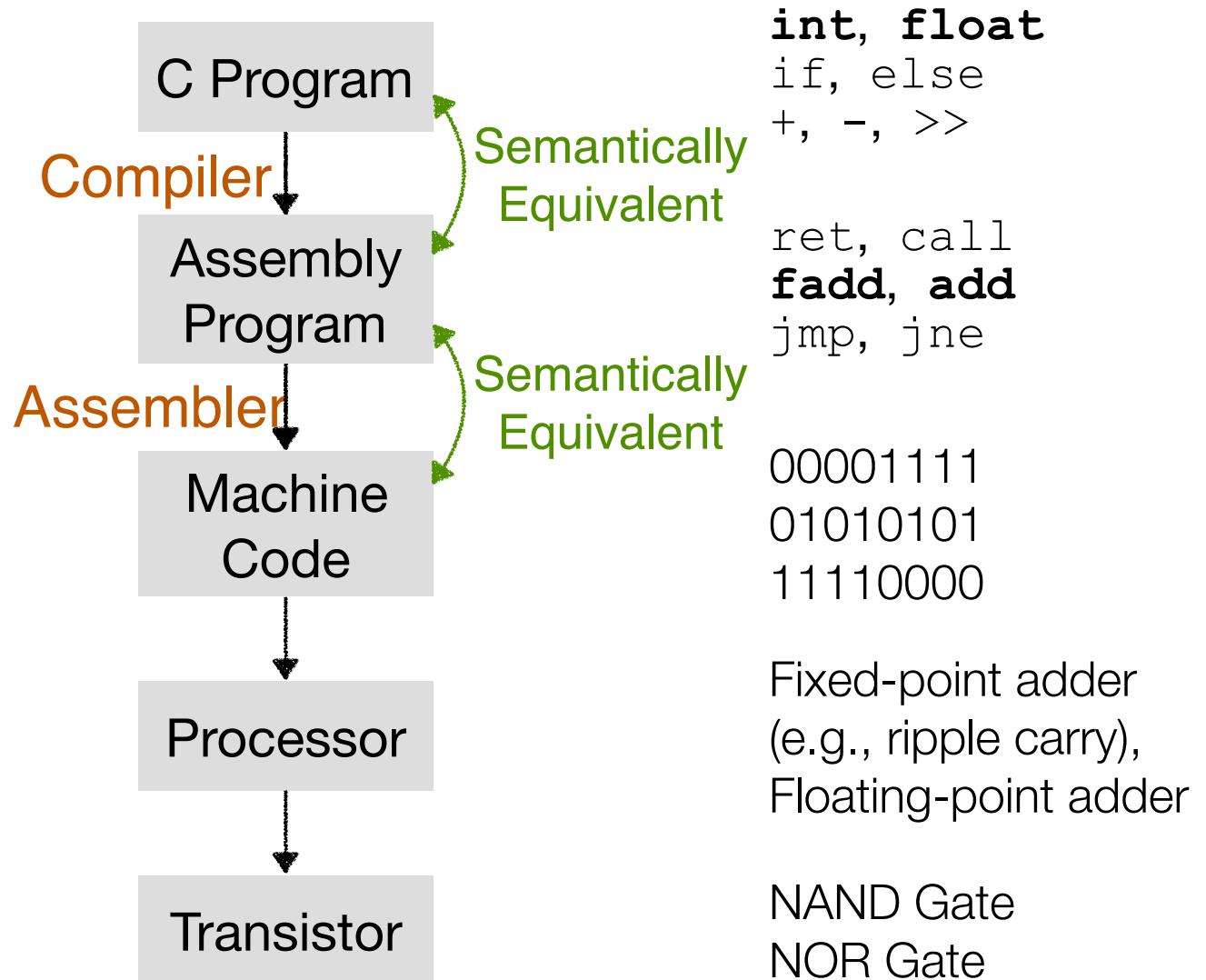


Special Value

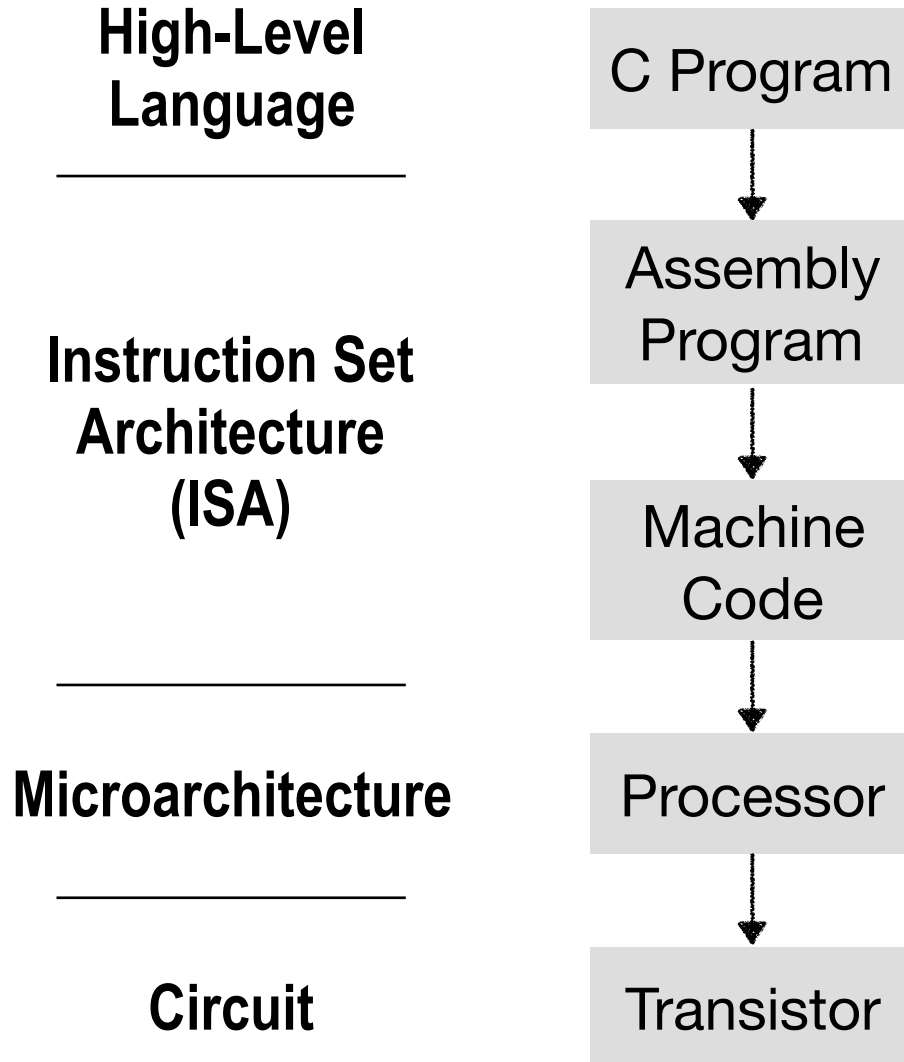


<i>s</i>	<i>exp</i>	<i>frac</i>	Value	Value
0	000	00	0.00×2^{-2}	0
0	000	11	0.11×2^{-2}	3/16
0	001	00	1.00×2^{-2}	1/4
0	001	11	1.11×2^{-2}	7/16
0	010	00	1.00×2^{-1}	1/2
0	010	11	1.11×2^{-1}	7/8
0	100	00	1.00×2^0	1
0	100	11	1.11×2^0	1 3/4
0	101	00	1.00×2^1	2
0	101	11	1.11×2^1	3 1/2
0	110	00	1.00×2^2	4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

So far in 252...



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 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



Aside: Dynamic Binary Translation

macOS Monterey

Version 12.0.1

MacBook Pro (16-inch, 2021)

Chip **Apple M1 Pro**

Memory 16 GB

Serial Number VQ4GVYVN6F

- Apple M1 is based on the Arm ISA. A program compiled to x86 ISA is dynamically translated to Arm ISA by Rosetta.
- Not the first time Apple plays this trick.



Fast performance
Translated at install time
Dynamic translation for JITs
Transparent to user



Rosetta 2

Aside: Dynamic Binary Translation

Circa 2006: PowerPC to x86 translation

Rosetta

Translates PowerPC to Intel



Intel x86 ISA Evolution (Milestones)

- Evolutionary design: Added more features as time goes on

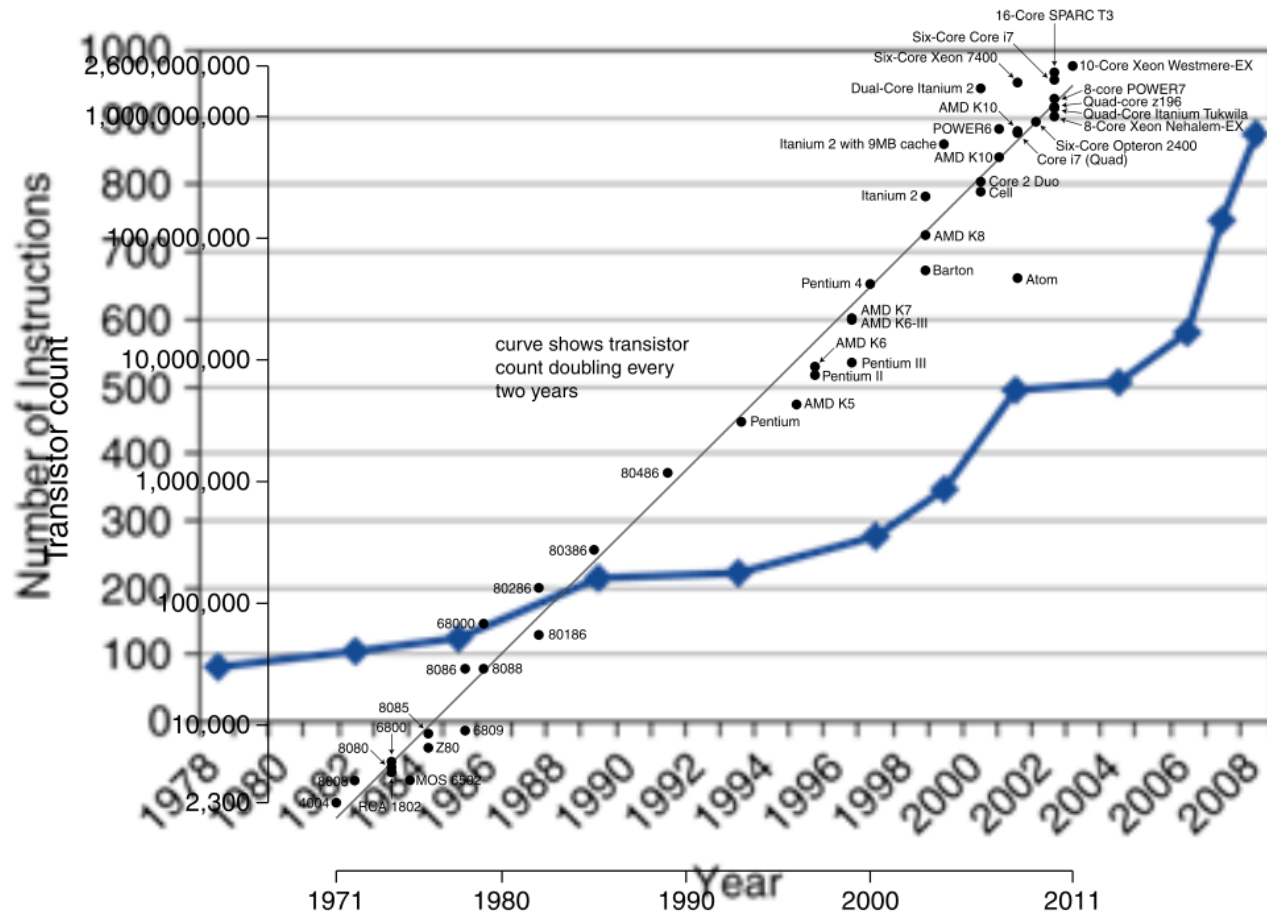
Date	Feature	Notable Implementation
1974	8-bit ISA	8080
1978	16-bit ISA (Basis for IBM PC & DOS)	8086
1980	Add Floating Point instructions	8087
1985	32-bit ISA (Refer to as IA32)	386
1997	Add Multi-Media eXtension (MMX)	Pentium/MMX
1999	Add Streaming SIMD Extension (SSE)	Pentium III
2001	Intel's first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

Our Coverage

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

Moore's Law

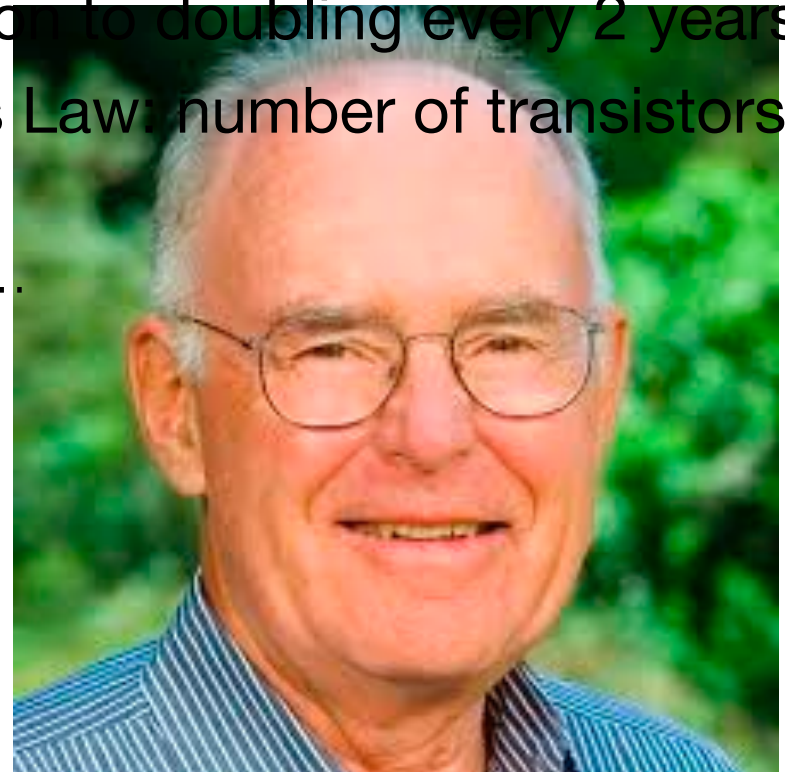
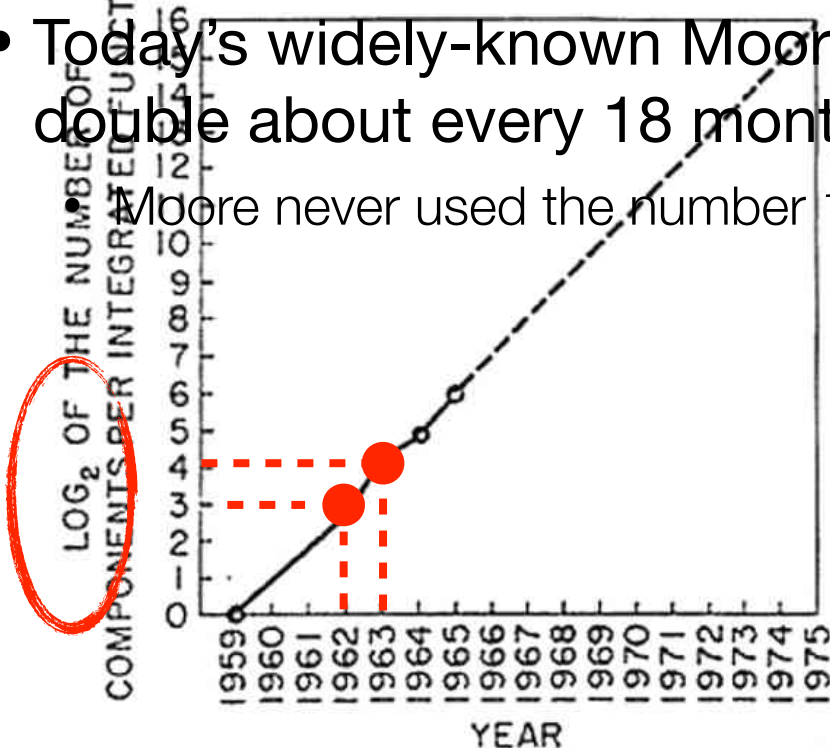
- More instructions typically require more transistors to implement



Moore's Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year
- In 1975 he revised the prediction to doubling every 2 years
- Today's widely-known Moore's Law: number of transistors double about every 18 months

Moore never used the number 18...



Moore's Law



BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE

TECH —

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

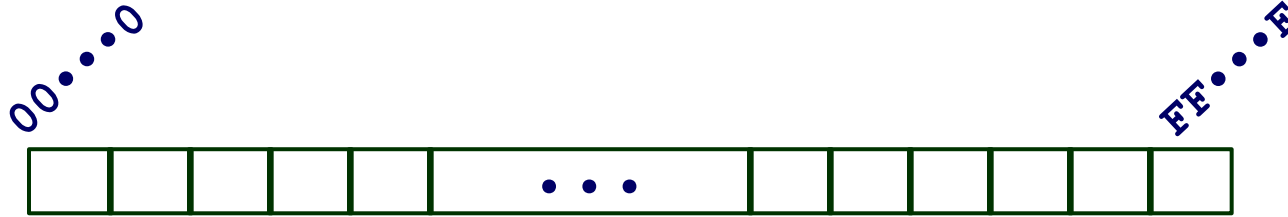
Final semiconductor industry roadmap says the future is 3D packaging and cooling.

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

Today: Compute and Control Instructions

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

Byte-Oriented Memory Organization



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

How Does Pointer Work in C???

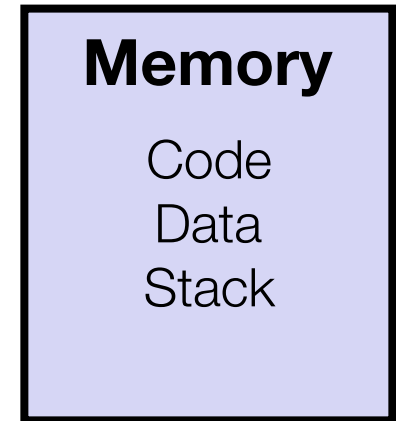
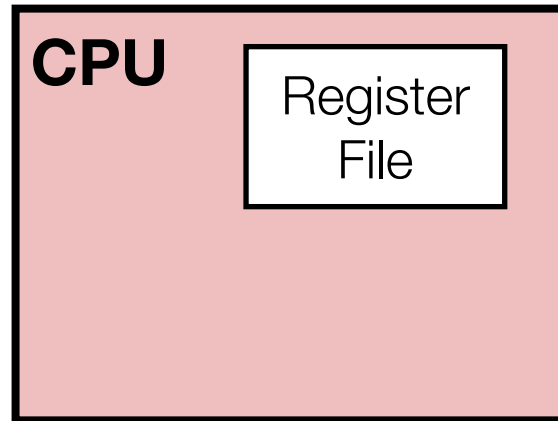
→ `char a = 4;`
`char b = 3;`
`char* c;`
`c = &a;`
`b += (*c);`

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

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

Assembly Code's View of Computer: ISA

Assembly Programmer's Perspective of a Computer



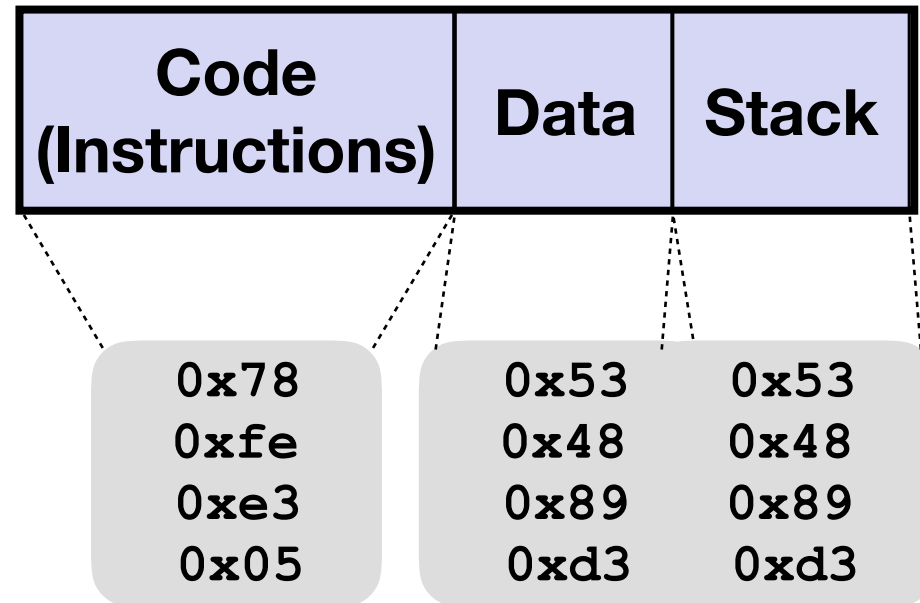
- (Byte Addressable) Memory

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

- Register file

Instruction is the fundamental unit of work.

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



x86-64 Integer Register File

← 8 Bytes →

%rax

%rbx

%rcx

%rdx

%rsi

%rdi

%rsp

%rbp

%r8

%r9

%r10

%r11

%r12

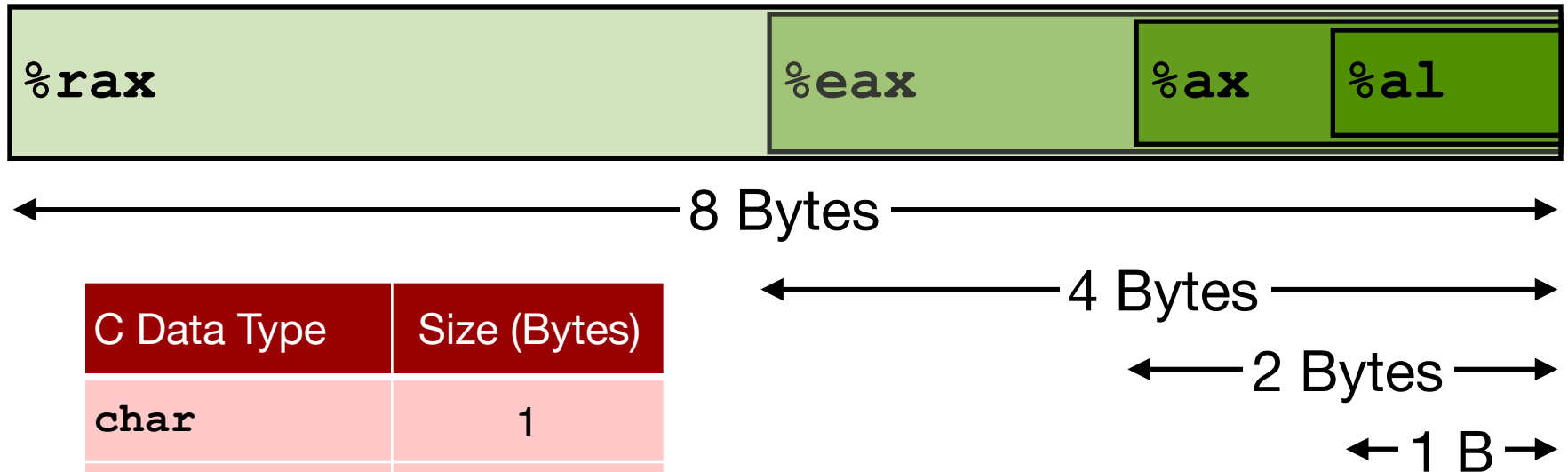
%r13

%r14

%r15

x86-64 Integer Register File

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

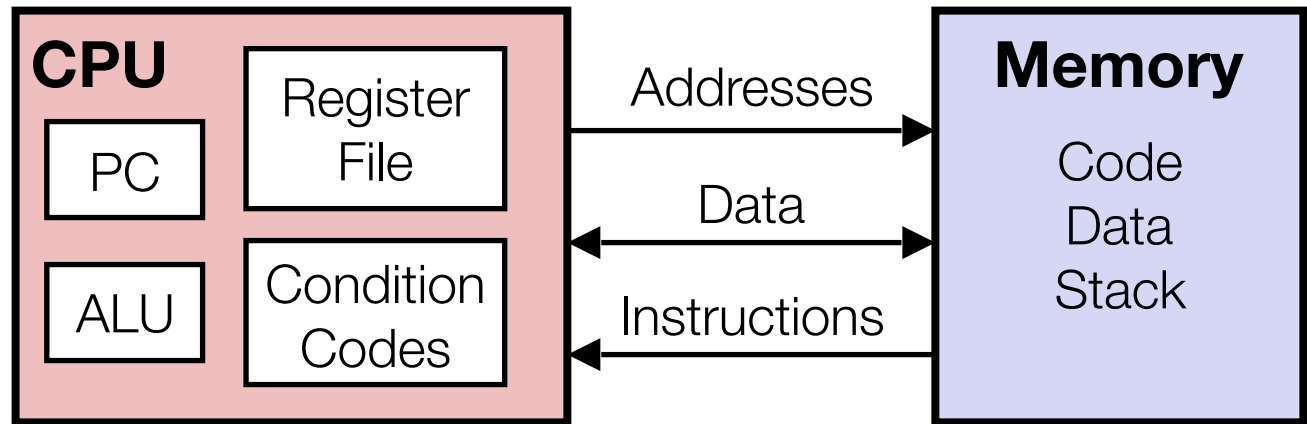


C Data Type	Size (Bytes)
char	1
short	2
int	4
long	8
Pointer	8

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

Assembly Code's View of Computer: ISA

Assembly Programmer's Perspective of a Computer



- (Byte Addressable) Memory

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

- Register file

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

- PC: Program counter

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

- Arithmetic logic unit (ALU)

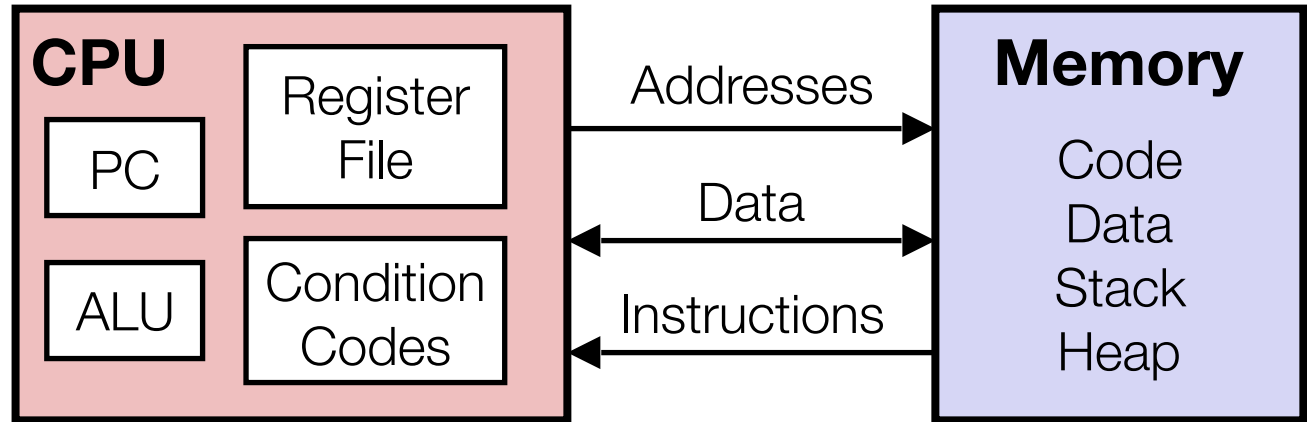
- Where computation happens

- Condition codes

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

Assembly Program Instructions

Assembly Programmer's Perspective of a Computer



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

Turning C into Object Code

C Code (sum.c)

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

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

Generated x86-64 Assembly

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

Obtain (on CSUG machine) with command

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

Turning C into Object Code

Generated x86-64 Assembly

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