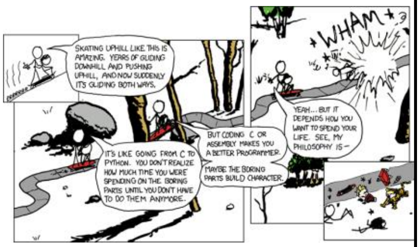
# x86-64 Assembly

CSE 351 Winter 2018

**Instructor:**
Mark Wyse

**Teaching Assistants:**
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan




http://xkcd.com/409/



## Administrivia

- Lab 1 due today!
  - Submit `bits.c` and `pointer.c`
- Homework 2 due next Wednesday (1/24)
  - On Integers, Floating Point, and x86-64





## Floating point topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- **Floating-point in C**

- There are many more details that we won't cover
  - It's a 58-page standard…





## Floating Point in C

!!!

- C offers two (well, 3) levels of precision

| | | |
|---|---|---|
| `float` | `1.0f` | single precision (32-bit) |
| `double` | `1.0` | double precision (64-bit) |
| `long double` | `1.0L` | *("double double" or quadruple)* precision (64-128 bits) |

- `#include <math.h>` to get `INFINITY` and `NAN` constants
- Equality (`==`) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!





## Floating Point Conversions in C

!!!

- Casting between `int`, `float`, and `double` **changes** the bit representation
  - `int` → `float`
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - `int` or `float` → `double`
    - Exact conversion (all 32-bit `int`s representable)
  - `long` → `double`
    - Depends on word size (32-bit is exact, 64-bit may be rounded)
  - `double` or `float` → `int`
    - Truncates fractional part (rounded toward zero)
    - "Not defined" when out of range or NaN: generally sets to `Tmin` (even if the value is a very big positive)





## Number Representation Really Matters

- **1991:** Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- **1996:** Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- **2000:** Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- **2038:** Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to TMin in 2038
- **Other related bugs:**
  - 1982: Vancouver Stock Exchange (truncation instead of rounding)
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown "smart" warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)

# Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

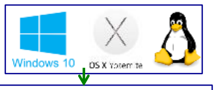
Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

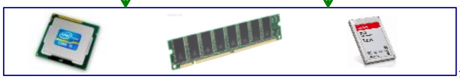
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:




Computer system:



Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C





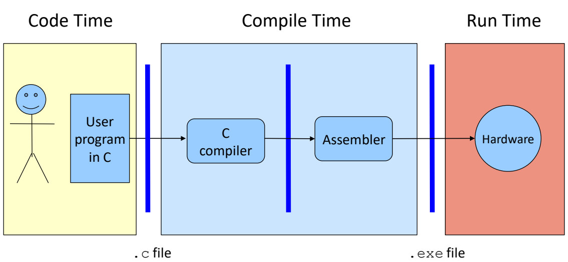
# Basics of Machine Programming & Architecture

- What is an ISA (Instruction Set Architecture)?
- A brief history of Intel processors and architectures
- Intro to Assembly and Registers





# Translation

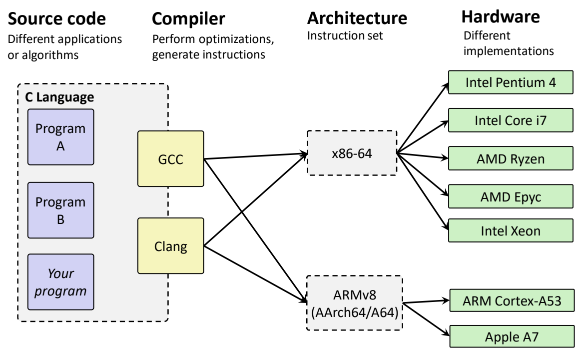



What makes programs run fast(er)?





# HW Interface Affects Performance








# Definitions

- **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - "What is directly visible to software"
- **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469, 470

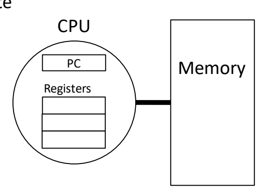
- Are the following part of the architecture?
  - Number of registers?
  - How about CPU frequency?
  - Cache size? Memory size?





# Instruction Set Architectures

- The ISA defines:
  - The system's state (*e.g.* registers, memory, program counter)
  - The instructions the CPU can execute
  - The effect that each of these instructions will have on the system state



## Instruction Set Philosophies

- *Complex Instruction Set Computing* (CISC): Add more and more elaborate and specialized instructions as needed
  - Lots of tools for programmers to use, but hardware must be able to handle all instructions
  - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs
- *Reduced Instruction Set Computing* (RISC): Keep instruction set small and regular
  - Easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones





## General ISA Design Decisions

- Instructions
  - What instructions are available? What do they do?
  - How are they encoded?
- Registers
  - How many registers are there?
  - How wide are they?
- Memory
  - How do you specify a memory location?





## Mainstream ISAs

**x86**

| | |
|---|---|
| Designer | Intel, AMD |
| Bits | 16-bit, 32-bit and 64-bit |
| Introduced | 1978 (16-bit), 1985 (32-bit), 2003 (64-bit) |
| Design | CISC |
| Type | Register-memory |
| Encoding | Variable (1 to 15 bytes) |
| Endianness | Little |

Macbooks & PCs
(Core i3, i5, i7, M)
x86-64 Instruction Set

**ARM architectures**

| | |
|---|---|
| Designer | ARM Holdings |
| Bits | 32-bit, 64-bit |
| Introduced | 1985; 31 years ago |
| Design | RISC |
| Type | Register-Register |
| Encoding | AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1] |
| Endianness | Bi (little as default) |

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

**MIPS**

| | |
|---|---|
| Designer | MIPS Technologies, Inc. |
| Bits | 64-bit (32→64) |
| Introduced | 1981; 35 years ago |
| Design | RISC |
| Type | Register-Register |
| Encoding | Fixed |
| Endianness | Bi |

Digital home & networking equipment
(Blu-ray, PlayStation 2)
MIPS Instruction Set





## Intel/AMD x86 Evolution: Milestones

| Name | Date | Transistors | MHz |
|---|---|---|---|
| 8086 | 1978 | 29K | 5-10 |
| First 16-bit Intel processor. Basis for IBM PC & DOS<br>1 MB address space | | | |
| 386 | 1985 | 275K | 16-33 |
| First 32-bit Intel processor, referred to as IA32<br>Added "flat addressing," capable of running Unix | | | |
| Pentium (P5) | 1993 | 3.2M | 60 |
| First superscalar IA32 | | | |
| Athlon (K7) | 1999 | 22M | 500-2333 |
| First desktop processor with 1 GHz clock (at roughly same time as Pentium III) | | | |
| Athlon 64 (K8) | 2003 | 106M | 1600-3200 |
| First x86-64 processor architecture | | | |
| Pentium 4E | 2004 | 125M | 2800-3800 |
| First 64-bit Intel x86 processor | | | |





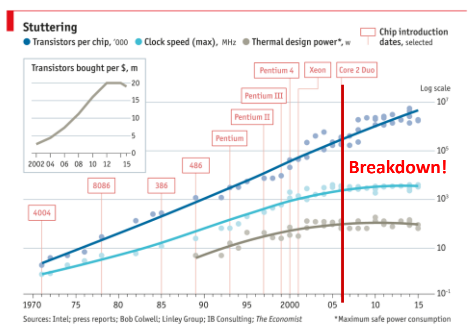
## Intel/AMD x86 Evolution: Milestones

| Name | Date | Transistors | MHz |
|---|---|---|---|
| Core 2 | 2006 | 291M | 1060-3500 |
| First multi-core Intel Processor | | | |
| Core i7 | 2008 | 731M | 1700-3900 |
| Four cores | | | |
| AMD Phenom (K10) | 2008 | 758M | 1800-2600 |
| First "true" quad core, with all cores on same silicon die | | | |
| Core i7 (Coffee Lake) | 2017 | ? | 2800-4700 |
| | | | |
| Ryzen 7 (Zen) | 2017 | 4.8B | 3000-4200 |





## Technology Scaling




http://www.economist.com/technology-quarterly/2016-03-12/after-moores-law

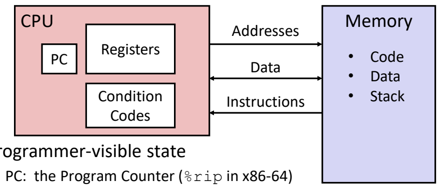
## Transition to 64-bit

- Intel attempted radical shift from IA32 to IA64 (2001)
  - Completely new architecture (Itanium)
  - Execute IA32 code only as legacy
  - Performance disappointing
- AMD solution: "AMD64" (2003)
  - x86-64, evolutionary step from IA32
- Intel pursued IA64
  - Couldn't admit its mistake with Itanium
- Intel announces "EM64T" extension to IA32 (2004)
  - Extended Memory 64-bit Technology
  - Nearly identical to AMD64!





## Assembly Programmer's View




- Programmer-visible state
  - PC: the Program Counter (`%rip` in x86-64)
    - Address of next instruction
  - Named registers
    - Together in "register file"
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
- Memory
  - Byte-addressable array
  - Code and user data
  - Includes *the Stack* (for supporting procedures)





## Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - *Load* data from memory into register
     - `%reg` = Mem[address]
   - *Store* register data into memory
     - Mem[address] = `%reg`

**Remember:** Memory is indexed just like an array of bytes!

2) Perform arithmetic operation on register or memory data
   - `c = a + b;`     `z = x << y;`     `i = h & g;`

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches





## x86-64 Assembly "Data Types"

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (*e.g.* `%xmm1`, `%ymm2`)
  - Come from *extensions to x86* (SSE, AVX, …)

  } Not covered In 351
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - "AT&T": used by our course, slides, textbook, gnu tools, …
  - "Intel": used by Intel documentation, Intel tools, …
  - Must know which you're reading





## What is a Register?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)
- Registers have *names*, not *addresses*
  - In assembly, they start with `%` (*e.g.* `%rsi`)
- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but *especially* x86





## x86-64 Integer Registers – 64 bits wide

| | | | |
|---|---|---|---|
| `%rax` | `%eax` | `%r8` | `%r8d` |
| `%rbx` | `%ebx` | `%r9` | `%r9d` |
| `%rcx` | `%ecx` | `%r10` | `%r10d` |
| `%rdx` | `%edx` | `%r11` | `%r11d` |
| `%rsi` | `%esi` | `%r12` | `%r12d` |
| `%rdi` | `%edi` | `%r13` | `%r13d` |
| `%rsp` | `%esp` | `%r14` | `%r14d` |
| `%rbp` | `%ebp` | `%r15` | `%r15d` |

- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)

## Some History: IA32 Registers – 32 bits wide

general purpose:

| 32-bit | 16-bit | high byte | low byte | Name Origin |
|---|---|---|---|---|
| %eax | %ax | %ah | %al | *accumulate* |
| %ecx | %cx | %ch | %cl | *counter* |
| %edx | %dx | %dh | %dl | *data* |
| %ebx | %bx | %bh | %bl | *base* |
| %esi | %si | | | *source index* |
| %edi | %di | | | *destination index* |
| %esp | %sp | | | *stack pointer* |
| %ebp | %bp | | | *base pointer* |

16-bit virtual registers (backwards compatibility)

Name Origin (mostly obsolete)





## Memory vs. Registers

| Memory | vs. | Registers |
|---|---|---|
| Addresses | vs. | Names |
| `0x7FFFD024C3DC` | | `%rdi` |
| Big | vs. | Small |
| ~ 8 GB | | (16 x 8 B) = 128 B |
| Slow | vs. | Fast |
| ~50-100 ns | | sub-nanosecond timescale |
| Dynamic | vs. | Static |
| Can "grow" as needed while program runs | | fixed number in hardware |





## Operand types

- ***Immediate:*** Constant integer data
  - Examples: **`$0x400`**, **`$-533`**
  - Like C literal, but prefixed with **'$'**
  - Encoded with 1, 2, 4, or 8 bytes *depending on the instruction*
- ***Register:*** 1 of 16 integer registers
  - Examples: **`%rax`**, **`%r13`**
  - But **`%rsp`** reserved for special use
  - Others have special uses for particular instructions
- ***Memory:*** Consecutive bytes of memory at a computed address
  - Simplest example: **`(%rax)`**
  - Various other "address modes"

**%rax**
**%rcx**
**%rdx**
**%rbx**
**%rsi**
**%rdi**
**%rsp**
**%rbp**

**%rN**





## Summary

- x86-64 is a complex instruction set computing (CISC) architecture
- **Registers** are named locations in the CPU for holding and manipulating data
  - x86-64 uses 16 64-bit wide registers
- Assembly operands include immediates, registers, and data at specified memory locations





## Floating Point Summary

- Floats also suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow
  - "Gaps" produced in representable numbers means we can lose precision, unlike `int`s
    - Some "simple fractions" have no exact representation (*e.g.* 0.2)
    - "Every operation gets a slightly wrong result"
- Floating point arithmetic not associative or distributive
  - Mathematically equivalent ways of writing an expression may compute different results
- Never test floating point values for equality!
- Careful when converting between `int`s and `float`s!





## Floating Point Summary

- Converting between integral and floating point data types *does* change the bits
  - Floating point rounding is a HUGE issue!
    - Limited mantissa bits cause inaccurate representations
    - Floating point arithmetic is NOT associative or distributive