CSE 374 Programming concepts and tools

Winter 2024 Instructor: Alex McKinney

Today

Assembly

- The software/hardware interface
- What actually goes on under the hood

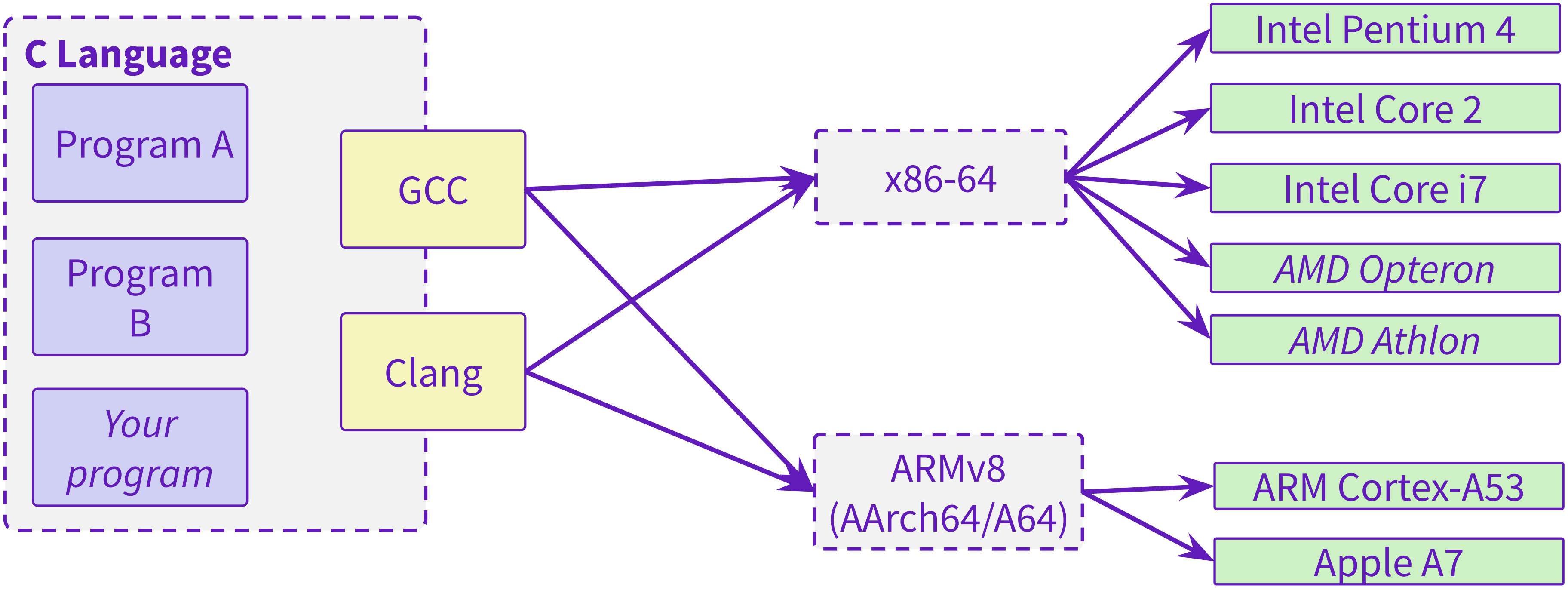
Architecture Sits at the Hardware Interface

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

Architecture
Instruction set

Hardware
Different implementations



Demo: Compiler Explorer (godbolt.org)

Definitions

Instruction Set Architecture (ISA): The parts of a processor design that one needs to understand to write assembly code

- “What is directly visible to software”
- The interface used to connect software to hardware
- Like a header file (**.h**) in C!

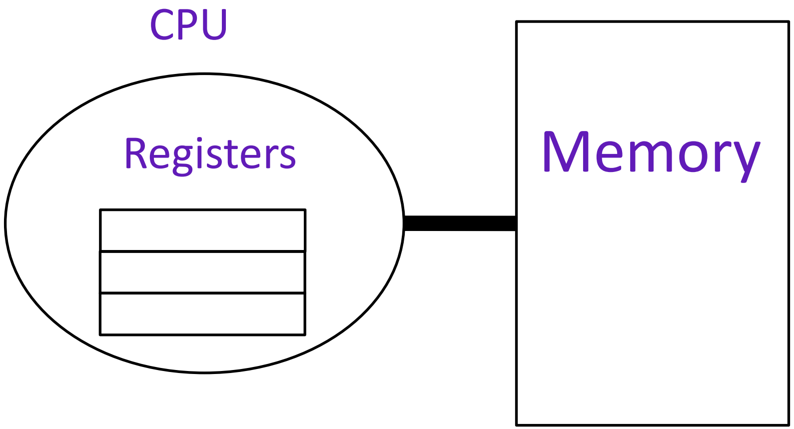
Microarchitecture: Implementation of the ISA

- “Micro” because it deals with the internal, smaller-scale aspects.
- CSE/EE 469

Instruction Set Architectures

The ISA defines:

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



Put simply, how instructions are encoded, executed, and manipulated by the processor.

Instructions

Your CPU executes instructions using machine code

- Machine code is a binary encoding of instructions

The human readable version of machine code is called assembly

What `gcc` does

- C code is *compiled* into assembly
- Assembly is *assembled* into machine code (inside of object files)
    - By the assembler
- Machine code in object files are *linked* into an executable

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 CISC instructions encountered with Linux programs

Reduced Instruction Set Computing (RISC): Keep instruction set small and regular

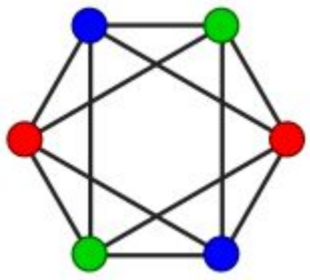
- Easier to build fast hardware
- Let software do the complicated operations by composing simpler ones

RISC-V is an open-source ISA growing in popularity (used by NVIDIA)

RISC vs CISC

RISC puts more onerous on the compiler, but permits more optimizations.

- Finer granularity -> more room for reordering / parallelization.



The study of compiler optimizations is huge.

- Lots of graph algorithms applied in practice (e.g. graph coloring).
- A rich field of research.

CISC simplifies the compiler’s job because the instruction handles so much more.

- Not as much room for optimization.
- Can’t easily decompose the instruction into multiple steps.

RISC vs CISC

RISC	CISC

`LOAD A, 2:3`
`LOAD B, 5:2`
`PROD A, B`
`STORE 2:3, A`
	`MULT 2:3, 5:2`

Four instructions are consolidated into one!

Ref: Stanford

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 (e.g. 8-bit, 16-bit, etc)?
  - Often the same size as the address space (e.g. 32-bit, 64-bit).

Memory

- How do you specify a memory location?

Mainstream ISAs

Different architectures have different assembly syntax

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

Old 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)

New Macbooks & 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

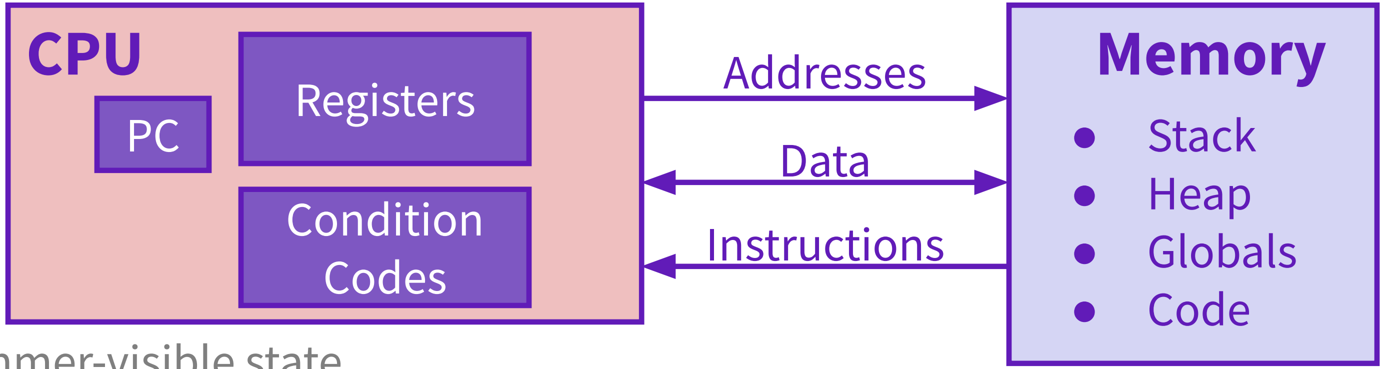
Writing Assembly Code? In 2024?

Chances are, you'll never write a program in assembly, but understanding assembly is the key to the machine-level execution model:

- Behavior of programs in the presence of bugs
  - When high-level language model breaks down
- Tuning program performance
- Implementing systems software
- Fighting malicious software
  - Distributed software is in binary form

Like how C will help you better appreciate the magic that Java does for you. Similarly, understanding assembly will help you understand what the machine is doing for you.

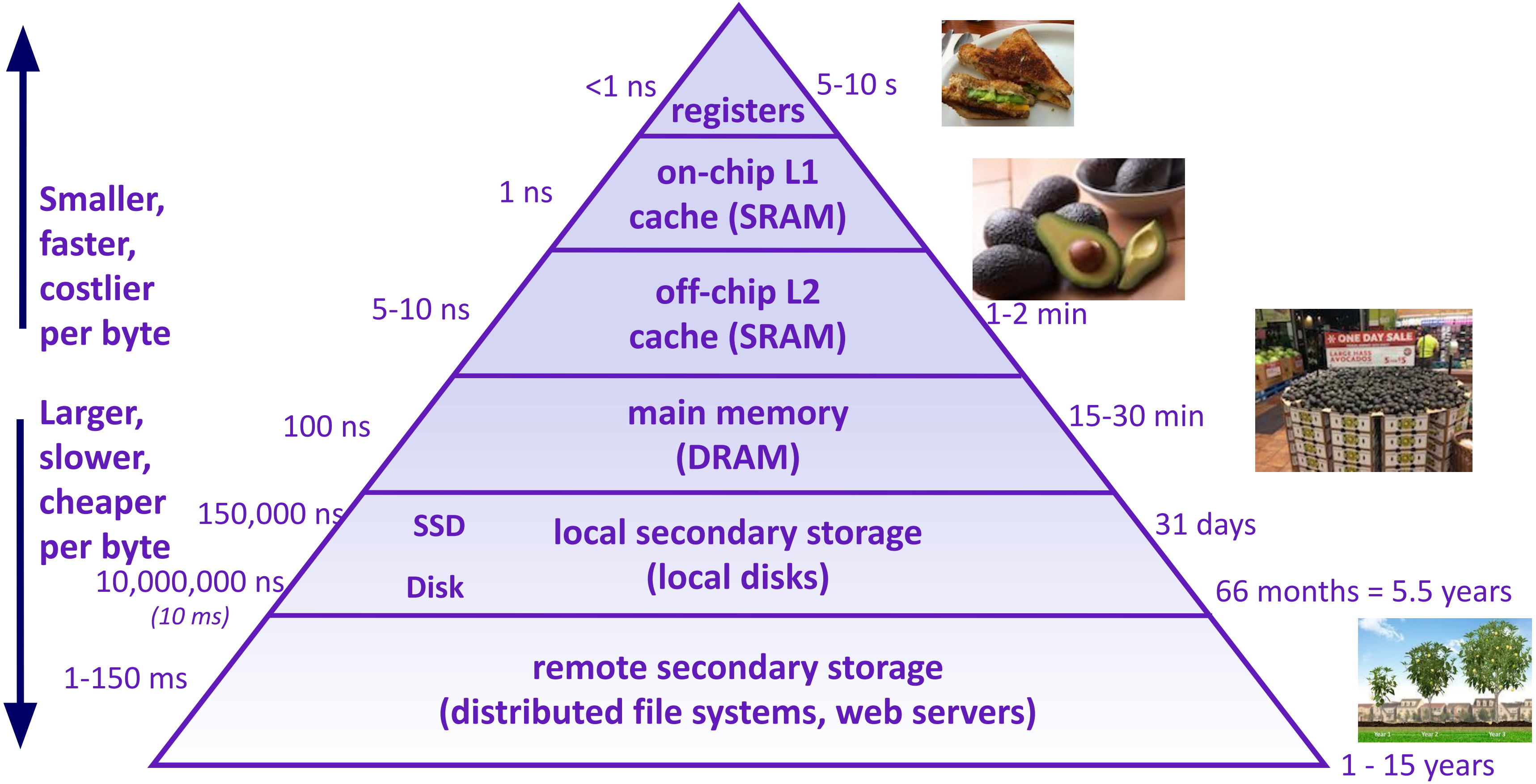
Assembly Programmer's View



Programmer-visible state

- PC: the Program Counter (`%rip` / Instruction Pointer in x86-64)
    - Address of next instruction
- Named registers
    - Heavily used program data
- Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

Review: Memory Hierarchy



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
- Complicated algorithms are used to ensure that registers are used wisely.

x86-64 Integer Registers – 64 bits wide

	r = 8 bytes	e = 4 bytes		
	%rax	%eax	%r8	%r8d
	%rbx	%ebx	%r9	%r9d
	%rcx	%ecx	%r10	%r10d
	%rdx	%edx	%r11	%r11d
	%rsi	%esi	%r12	%r12d
	%rdi	%edi	%r13	%r13d
Stack pointer	%rsp	%esp	%r14	%r14d
	%rbp	%ebp	%r15	%r15d

Memory

- Addresses
  - `0x7FFFD024C3DC`
- Big
  - ~ 8 GiB
- Slow
  - ~50-100 ns
- Dynamic
  - Can "grow" as needed while program runs

Registers

- Names
  - `%rdi`
- Small
  - (16 x 8 B) = 128 B
- Fast
  - sub-nanosecond timescale
- Static
  - fixed number in hardware

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`

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

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

Operand types

Immediate: Constant integer data

- Examples: `$0x400`, `$-533`
- Like C literal, but prefixed with `'$'`
- Encoded with 1, 2, 4, or 8 bytes

Register: 1 of 16 integer registers

- Examples: `%rax`, `%r13`

Memory: Consecutive bytes of memory at a computed address

- Simplest example: `(%rax)`

`%rax`
`%rcx`
`%rdx`
`%rbx`
`%rsi`
`%rdi`
`%rsp`
`%rbp`
`%rN`

Moving Data

General form: **`mov_ source, destination`**

- Missing letter (_) specifies size of operands
- Lots of these in typical code

Example

- `movb src, dst`
  - Move 1-byte "byte"
- `movw src, dst`
  - Move 2-byte "word"
- `movl src, dst`
  - Move 4-byte "long word"
- `movq src, dst`
  - Move 8-byte "quad word"

Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	`movq $0x4, %rax`	rax = 4;
		Mem	`movq $-147, (%rax)`	*rax = -147;
	Reg	Reg	`movq %rax, %rdx`	rdx = rax;
		Mem	`movq %rax, (%rdx)`	*rdx = rax;
	Mem	Reg	`movq (%rax), %rdx`	rdx = *rax;

( ) is like * in C

What about mem to mem?

Cannot do memory-memory transfer with a single instruction!

How would you do it?

1. Mem -> Reg
2. Reg -> Mem

Requiring separate steps is important:

- Actually increases opportunities for more optimizations.
- Keeps the ISA as simple as possible.
- More consistent and predictable.

Some Arithmetic Operations

Binary (two-operand) Instructions:

- Beware argument order!
- How do you implement

"`r3 = r1 + r2`"?

Let say:

- r1 is in `%rdi`
- r2 is in `%rsi`
- r3 is in `%rax`

```
movq %rdi, %rax
addq %rsi, %rax
```

Format	Computation	
addq *src, dst*	*dst = dst + src*	(*dst += src*)
subq *src, dst*	*dst = dst – src*	
imulq *src, dst*	*dst = dst * src*	signed mult
shrq *src, dst*	*dst = dst >> src*	
shlq *src, dst*	*dst = dst << src*	(same as `salq`)
xorq *src, dst*	*dst = dst ^ src*	
andq *src, dst*	*dst = dst & src*	
orq *src, dst*	*dst = dst | src*	

q = operand size specifier
(e.g. b, w, l, q = 1, 2, 4, 8)

Some Arithmetic Operations

Unary (one-operand) Instructions:

Format	Computation	
`incq` *`dst`*	*dst = dst + 1*	increment
`decq` *`dst`*	*dst = dst – 1*	decrement
`negq` *`dst`*	*dst = –dst*	negate
`notq` *`dst`*	*dst = ~dst*	bitwise complement

Arithmetic Example

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

```
long simple_arith(long x, long y)
{
  long t1 = x + y;
  long t2 = t1 * 3;
  return t2;
}
```

```
y += x;
y *= 3;
long r = y;
return r;
```

```
simple_arith:
  addq    %rdi, %rsi
  imulq     $3, %rsi
  movq    %rsi, %rax
  ret
```

Compiler knows

- Arithmetic ops are like +=
- And it doesn't need intermediate values anyway
- So it transforms it to...

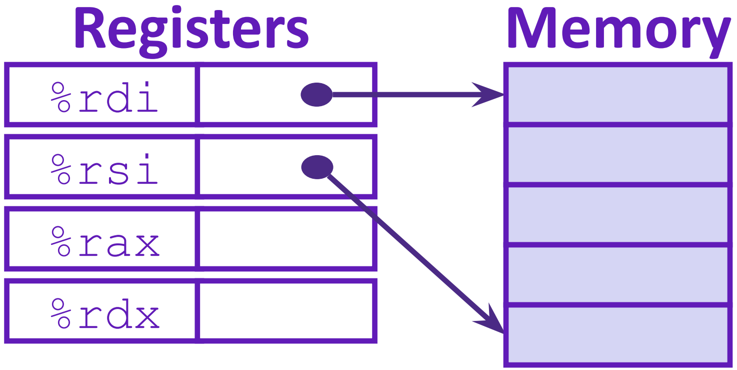
Example of Basic Addressing Modes

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

```
swap:
   movq   (%rdi), %rax
   movq   (%rsi), %rdx
   movq   %rdx, (%rdi)
   movq   %rax, (%rsi)
   ret
```

Understanding `swap()`

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




```
swap:
   movq   (%rdi), %rax
   movq   (%rsi), %rdx
   movq   %rdx, (%rdi)
   movq   %rax, (%rsi)
   ret
```

Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

Understanding `swap()`

Registers

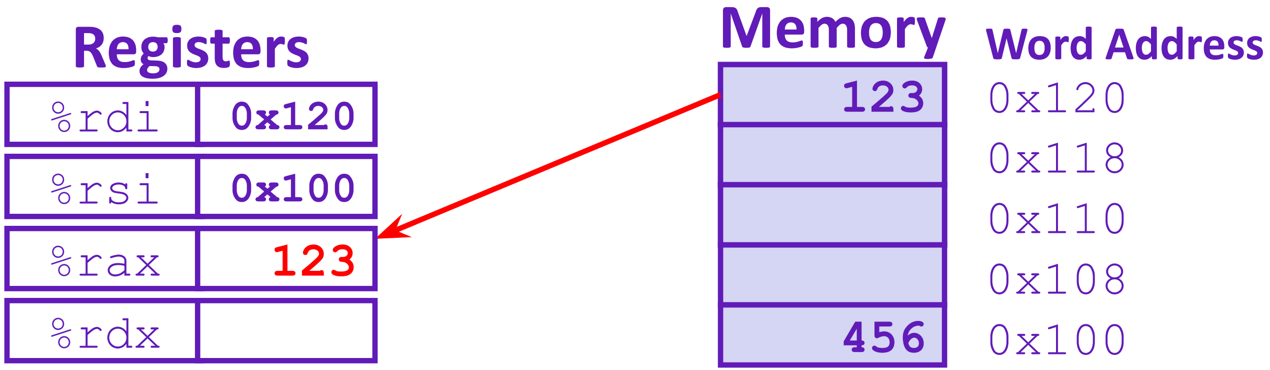
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

Memory	Word Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

```
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

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

Memory	Word Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

```
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

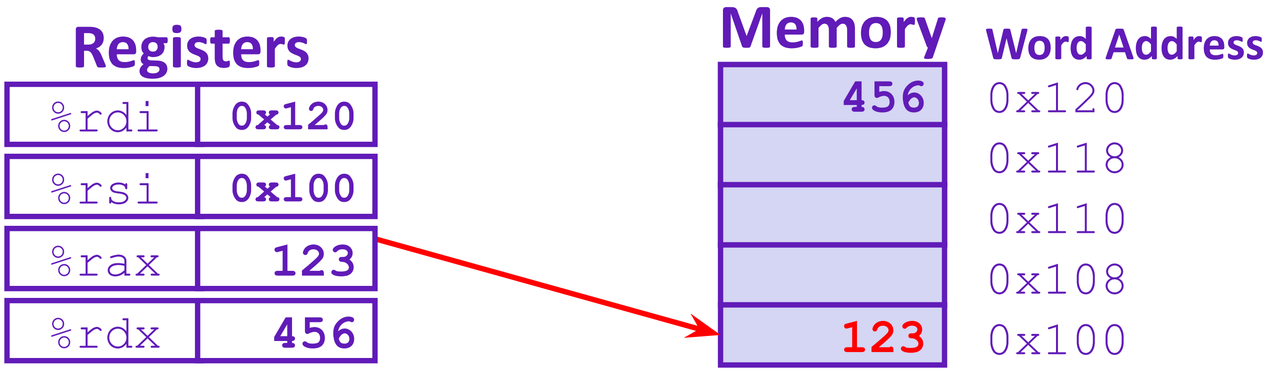
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

	Word Address
456	0x120
	0x118
	0x110
	0x108
456	0x100

```
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
```

What about Java, Bash, etc.?

C is translated directly into assembly (and then into machine code)

Other languages may be translated into another form

- Java is translated into an assembly-like form, which is then run by the Java interpreter/runtime
- The Java runtime is executing assembly instructions!

Some languages are directly interpreted without being translated into another form

- Most Bash implementations will directly interpret the commands without compiling
- Python can do either. It can be used as an interpreter or compile scripts

The Hardware/Software Interface

What's after Assembly?

Every line of assembly gets translated into binary.

- The length of the instruction varies based on the system
- 16-bit, 32-bit, 64-bit.

```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Generally composed of the following:

- Opcode (add, multiply, move, etc).
- Registers (%r1, %r2)
- Immediate values ($3)

Registers and immediate values are operands

Assembly <-> Binary

mov, $r1, $10

|

0001 001 0000000000001010

|

Op: Move | Register: r1 | Immediate: 10

Just like we've with other systems (e.g. networking), the computer knows how to interpret each set of bits.

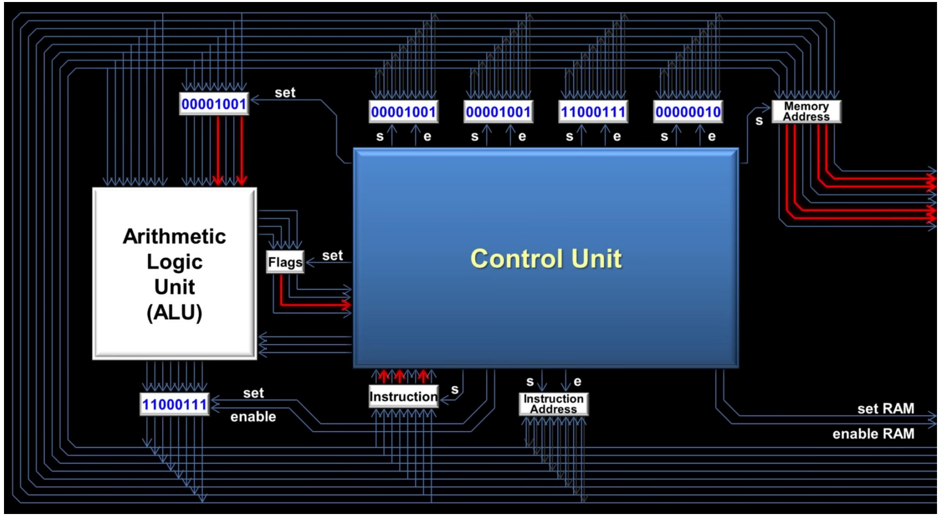
- It's a **protocol** (like floating point)!

CPU

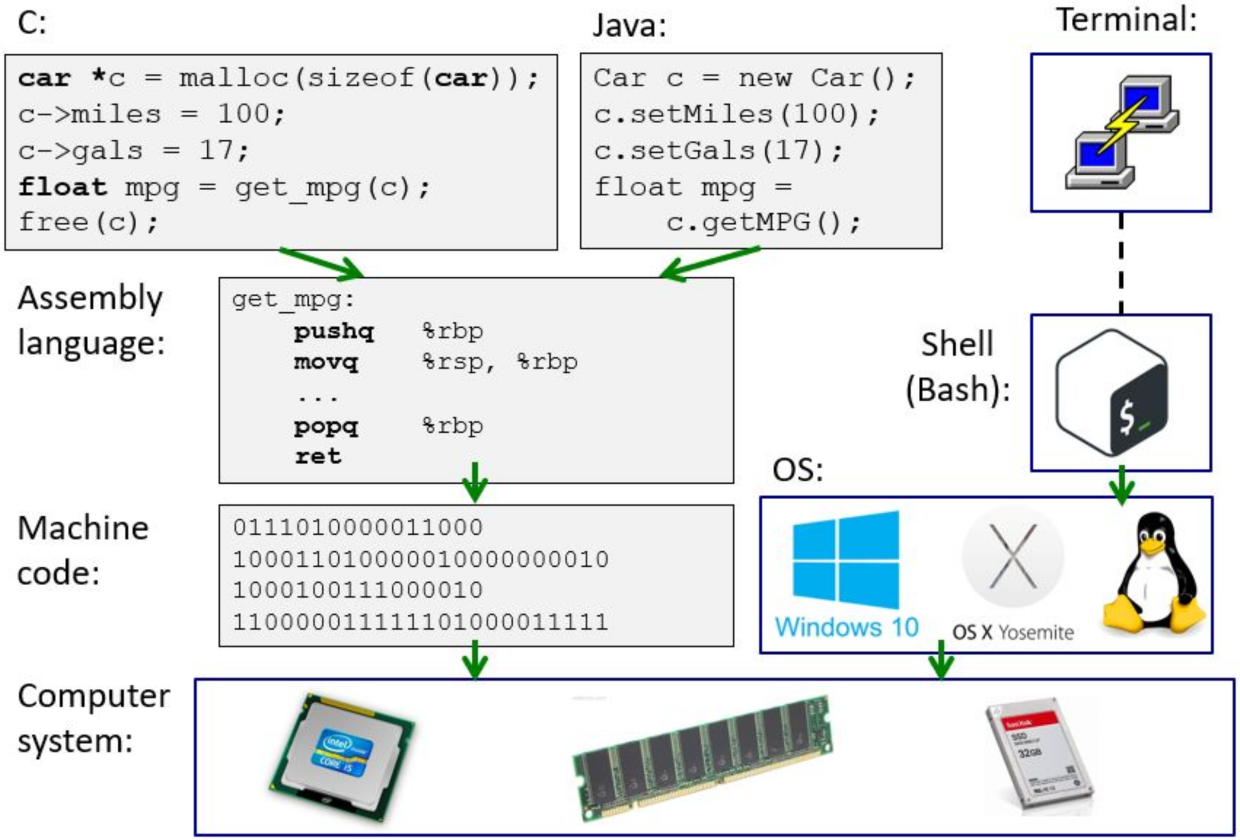
The CPU parses the instruction string.

- Takes action based on what is encoded
- Interacts with other components as needed (e.g. ALU for math)

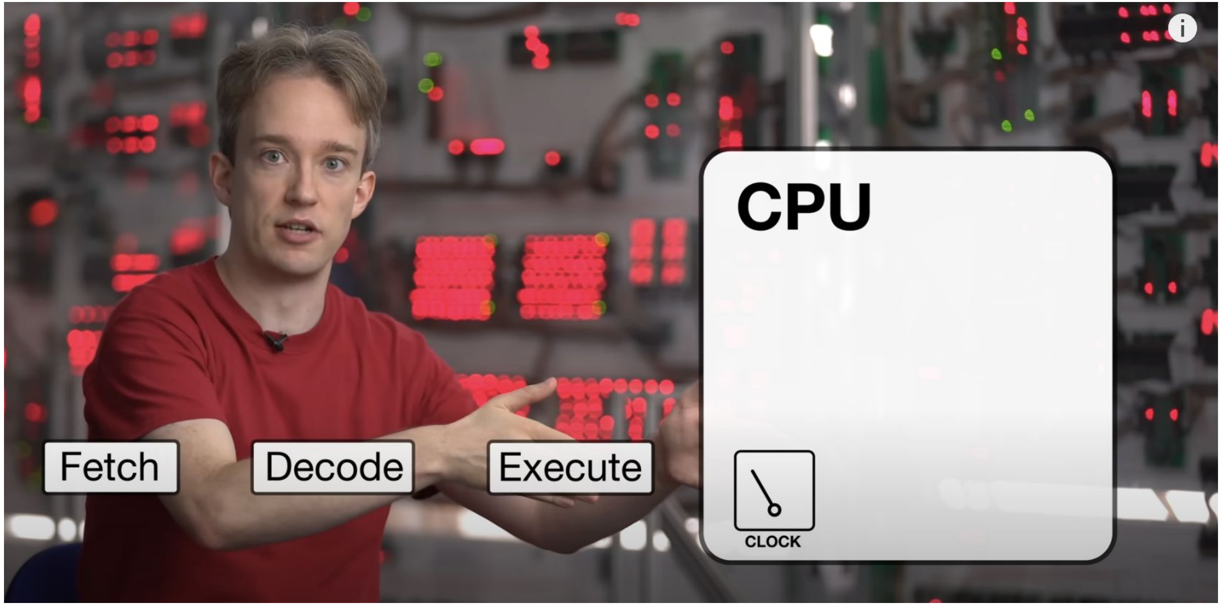
Fetch -> Decode -> Execute



The Big Picture



Computerphile



https://www.youtube.com/watch?v=Z5JC9Ve1sfI&t=30s