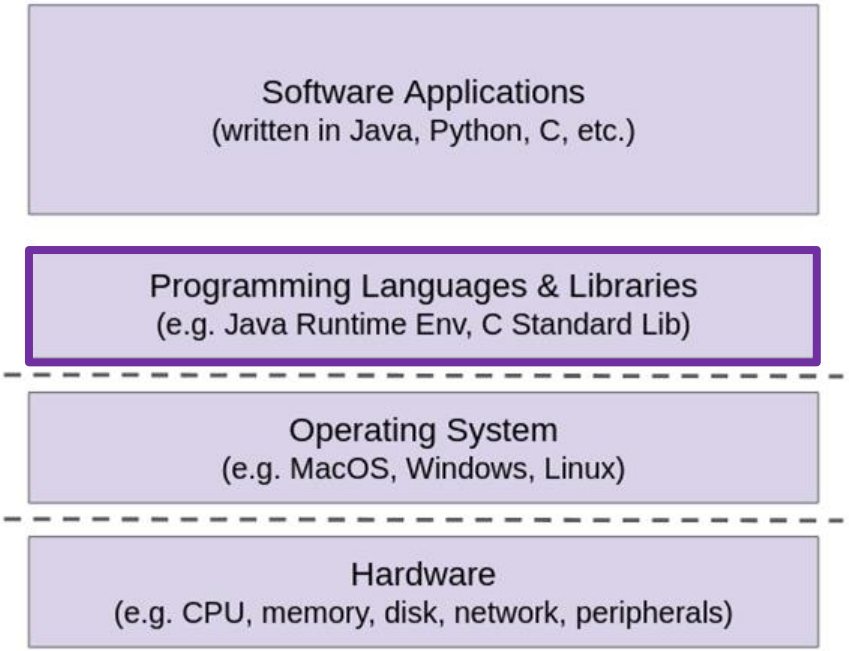
x86-64 Programming I

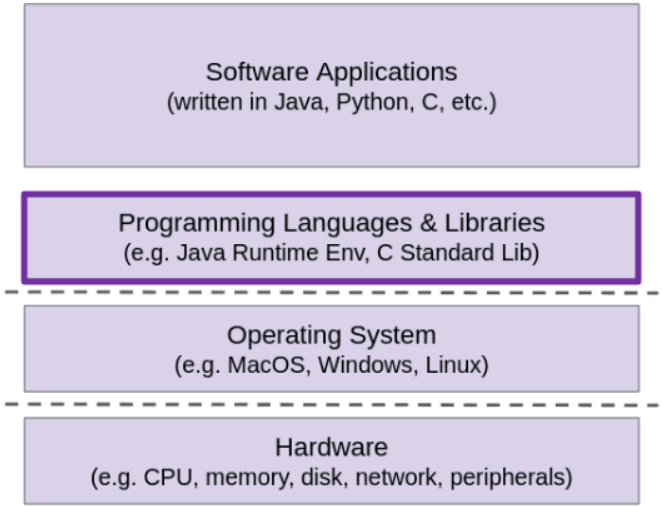
Layers of Computing Revisited

- So far, we've focused on **hardware**
  - How does the CPU store and read data from memory?
- Shifting focus to **languages & libraries**
  - How are programs created and executed on the CPU?



Programming Languages & Libraries:

- Topics:
  - **x86-64 assembly**
  - Procedures
  - Stacks
  - Executables
- How does your source code become something that your computer understands?
- How does the CPU organize and manipulate local data?



Lecture Topics

- **Assembly intro**
  - **Instruction set philosophies**
- X86-64 programming
  - Data types
  - Instructions
  - Registers
  - Memory addressing

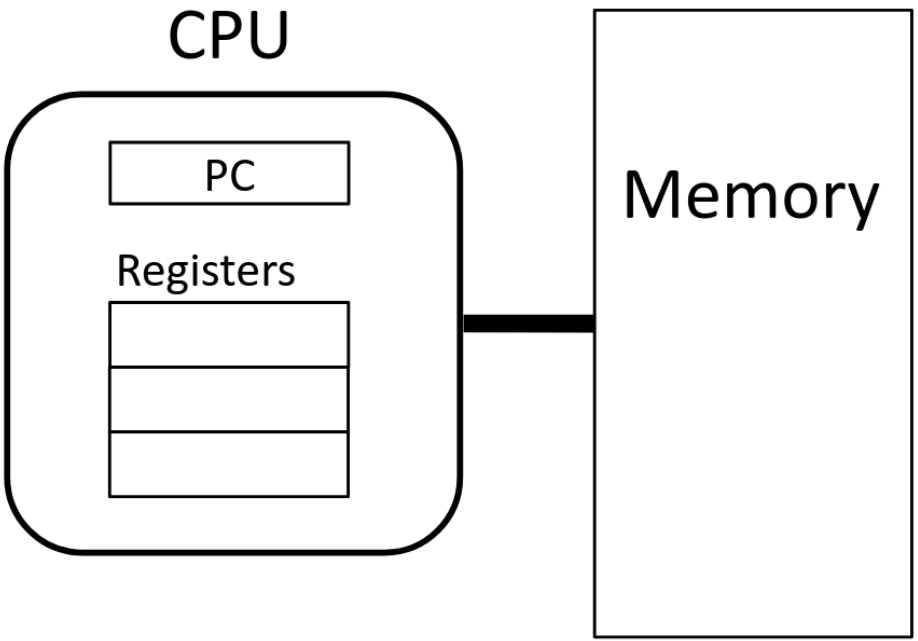
Definitions

- **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 “contract” between hardware and software

- **Microarchitecture**: hardware implementation of the ISA

Instruction Set Architectures

- 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



What is a Register?

- Special locations on the CPU that store a small amount of data
  - Accessed very quickly (once per clock cycle)
- Have *names*, not addresses
  - In x86, start with `%` (e.g., `%rsi`)
- Registers are at the heart of assembly programming
  - Very useful, but scarce, *especially* in x86

Memory vs. Registers

Memory

- Addresses
  - Ex: `0x7FFFD024C3DC`
- Big
  - ~16GB
- Slow
  - ~50-100ns
- Dynamic
  - Can expand as needed

Registers

- Names
  - Ex: `%rdi`
- Small
  - 16 8-byte registers = 128B
- Fast
  - <1ns
- Static
  - Fixed number in hardware

General ISA Design Decisions

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

Instruction Set Philosophies

- **Complex Instruction Set Computing (CISC)**: lots of elaborate instructions
  - 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, less power-hungry hardware
  - Let software do the complicated operations by composing simpler ones
  - ARM, RISC-V

Instruction Set Philosophies

- **Complex Instruction Set Computing (CISC)**: lots of elaborate instructions
  - 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

<u>Example</u>: `ADDSUBPS (operates on 128 bit XMM register)`

- "Adds odd-numbered single-precision floating-point values of the first source operand (second operand) with the corresponding single-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered single-precision floating-point values from the second source operand from the corresponding single-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand."

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)
Branching	Condition code
Endianness	Little

PCs, older Macs
x86-64 instruction set

ARM

Designer	Arm Holdings
Bits	32-bit, 64-bit
Introduced	1985
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]
Branching	Condition code, compare and branch
Endianness	Bi (little as default)

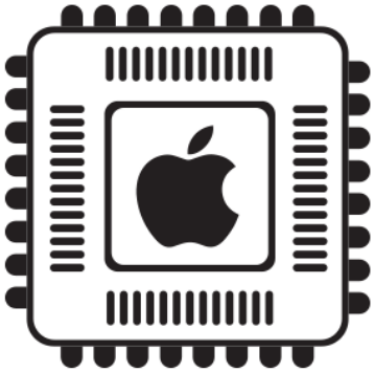
Mobile devices, M1/M2 Macs
ARM instruction set

RISC-V

Designer	University of California, Berkeley
Bits	32 · 64 · 128
Introduced	2010
Design	RISC
Type	Load-store
Encoding	Variable
Endianness	Little[1][3]

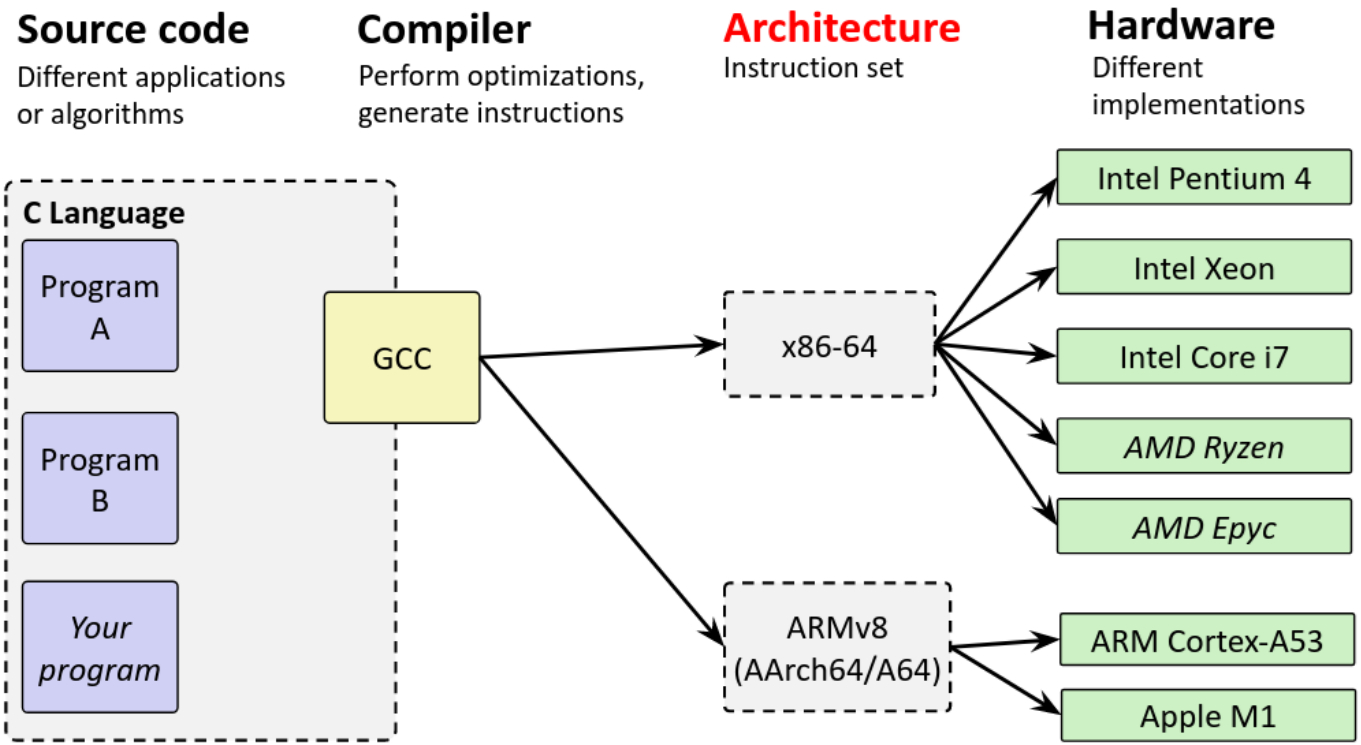
Mostly research
RISC-V instruction set

Current Industry Trends - A RISC-y Shift



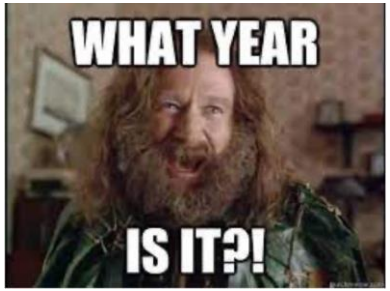
- Historically, there was a lot of debate about RISC vs CISC
  - Intel went the CISC route in the 1980s
    - Would make programming in assembly easier
    - Implementing more things in hardware
- Traditional wisdom says the RISC is better for simple systems, not PCs
- *But* things are shifting!
  - Apple switched to ARM in 2020
- Why?
  - **Efficiency**: RISC uses less power
  - **Performance**: each instruction is faster, easier to parallelize
  - **Scalability**: suitable for devices of all sizes (desktops, laptops, and phones)

Architecture Sits at the Hardware Interface



Writing Assembly Code? In $CURRENT_YEAR???

- Chances are, you'll never write a program in assembly, but understanding it 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
    - Understand optimizations done/not done by the compiler
  - Implementing systems software
    - What are the "states" of processes that the OS must manage
    - Using special units (timers, I/O co-processors, etc.) inside processor!
  - Fighting malicious software
    - Distributed software is in binary form



Lecture Topics

- Assembly intro
  - Instruction set philosophies
- **X86-64 programming**
  - **Data types**
  - **Instructions**
  - **Registers**
  - **Memory addressing**

x86-64 Integer Registers – 64 bits wide

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

x86-64 Assembly “Data Types”

- Integral data of 1, 2, 4, or 8 bytes (`b`, `w`, `l`, `q`)
- Floating point data
  - Different registers for those (e.g., `%xmm1`, `%ymm2`)
  - Come from extensions to x86 (SSE, AVX, …)
- No aggregate types such as arrays or structs
  - Just contiguously allocate bytes in memory
- Two common syntaxes—Must know which you’re reading!
  - **AT&T**: gnu tools (including gcc), …
  - **Intel**: used in Intel documentation, Intel tools, …

Instruction Sizes and Operands

- Size specifiers
  - `b` = 1-byte (“byte”)
  - `w` = 2-byte (“word”)
  - `l` = 4-byte (“long word”)
  - `q` = 8-byte (“quad word”)
  - If using registers, much match width
- Operand types
  - **Immediate**: constant value ($)
  - **Register**: 1 of 16 general-purpose registers (%)
  - **Memory**: consecutive bytes of memory at a computed address (( ))

Why is “word” 2 bytes? Because that was the word size when x86 was new, and it has to be maintained for backwards compatibility.

Instruction Types

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

1. Transfer data between memory and a register
   - Load from memory -> register
     - `%reg = Memory[address]`
   - Store from register -> memory
     - `Memory[address] = %reg`
   - Note: cannot transfer between two memory locations in one instruction!
2. Perform arithmetic operation on register or memory data
   - Ex: c = a + b; z = x << y; i = h & g;
3. Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

Moving Data

- General form: `mov_ <source>, <destination>`
  - More of a “copy” than a “move”
  - Missing letter (_) is for the width specifier

  Ex: `movq %rax, %rbx`

  - Copies the 8-byte value from register `%rax` into register `%rbx`
- Operand Combinations:
  - **Immediate** -> **Register** or **Memory** (copies Immediate value to location)
  - **Register** -> **Register** or **Memory** (copies data in register to location)
  - **Memory** -> **Register** (copies data in memory to register)
    - Can’t go from memory -> memory in a single instruction!

Some Arithmetic Operations

- Binary (two-argument) operations
  - Beware argument order!
    - `src` can be immediate, register, or memory
    - `dst` only register or memory
    - Results always stored in `dst`
  - Maximum of **one** memory operand!
  - No distinction between signed and unsigned
    - Only arithmetic vs logical shifts

Format	Computation	Notes
addq src, dst	dst = dst + src	
subq src, dst	dst = dst - src	
imulq src, dst	dst = dst * src	
sarq src, dst	dst = dst >> src	Arithmetic
shrq src, dst	dst = dst >> src	Logical
shlq src, dst	dst = dst << src	Same as **shl**q
xorq src, dst	dst = dst ^ src	
andq src, dst	dst = dst & src	
orq src, dst	dst = dst | src	

Practice Question

Which of the following are valid implementations of `rcx = rax + rbx`?

- ```
  addq %rax, %rcx
  addq %rbx, %rcx
  ```
- ```
  movq %rax, %rcx
  addq %rbx, %rcx
  ```
- ```
  movq $0, %rcx
  addq %rbx, %rcx
  addq %rax, %rcx
  ```
- ```
  xorq %rax, %rax
  addq %rax, %rcx
  addq %rbx, %rcx
  ```

Arithmetic Example

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

Register	Uses
%rdi	1st arg (x)
%rsi	2nd arg (y)
%rax	return value

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

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

Example of Basic Addressing Modes

```
long add_ptr(long* xp, long* yp)
{
    long t0 = *xp;
    long t1 = *yp;
    return t0 + t1;
}
```

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

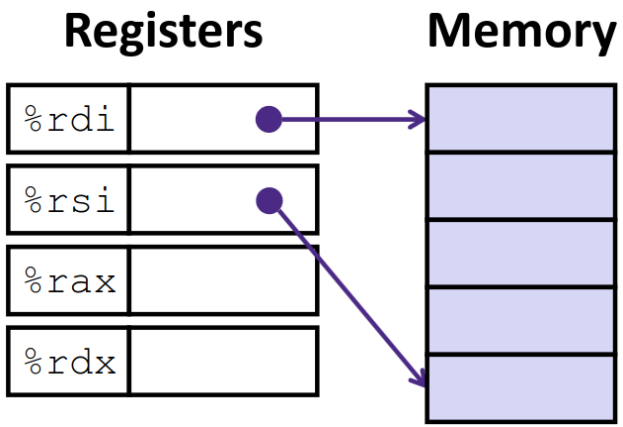
- Parentheses = memory addressing
  - Treat the value in the register as an address

Understanding add_ptr()

```
long add_ptr(long* xp, long* yp)
{
    long t0 = *xp;
    long t1 = *yp;
    return t0 + t1;
}
```

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

Register	Variable
%rdi	xp
%rsi	yp
%rdx	t0
%rax	return



Review Questions

Assume that the register `%rdx` holds the value **`0x 01 02 03 04 05 06 07 08`**

Answer the following questions about the instruction **`subq $1, %rdx`**

1. Operation type:
2. Operand types:
3. Operating width:
4. (extra) Result stored in `%rdx`:

Control Flow

- How do we alter the flow of execution?
  - ex: if/else ladders, loops, etc.

%rdi	x
%rsi	y
%rax	Return value

Example:

```
long max(long x, long y)
{
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

```
max:
        ???
        movq %rdi, %rax
        ???
        ???
        movq %rsi, %rax
        ???
        ret
```

Control Flow (pt 2)

%rdi	x
%rsi	y
%rax	Return value

- How do we alter the flow of execution?
  - ex: if/else ladders, loops, etc.

Example:

```
long max(long x, long y)
{
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

Conditional jump

Unconditional jump

```
max:
    if x<y, jump to else
    movq %rdi, %rax
    jump to done
else:
    movq %rsi, %rax
done:
    ret
```

Conditionals and Control Flow

- Conditional jump
  - Jump to somewhere else <u>if some condition is true</u>, otherwise execute next instruction in order
- Unconditional jump
  - <u>Always</u> jump when you get to this instruction
- Together, they can implement most control flow constructs in high-level languages:
  - `if (condition) {...} else {...}`
  - `while (condition) {...}`
  - `for (initialization; condition; iterative) {...}`
  - `switch {...}`

Using Condition Codes: Jumping

- General format: `j* target`
  - Sets `%rip` to `target` if the condition is met
  - `jmp` is **unconditional** - always jumps
- Used to create if/else statements, loops, etc.
  - More info next lecture

Don't bother memorizing, just use the chart.

Instruction	Condition	Description
jmp target	1	Unconditional
je target	ZF	Equal (to zero)
jne target	~ZF	Not Equal (to zero)
js target	SF	Negative
jns target	~SF	Nonnegative
jg target	~(SF^OF)&~ZF	Greater (signed)
jge target	~(SF^OF)	Greater or Equal (signed)
jl target	(SF^OF)	Less than (signed)
jle target	(SF^OF)|ZF	Less or Equal (signed)
ja target	~CF&~ZF	Above (unsigned ">")
jb target	CF	Below (unsigned "<")

Review Question

What should go in the two blank lines?

A) `cmpq %rsi, %rdi`
   `jle .L4`
B) `cmpq %rsi, %rdi`
   `jg .L4`
C) `testq %rsi, %rdi`
   `jle .L4`
D) `testq %rsi, %rdi`
   `jg .L4`

%rdi	x
%rsi	y
%rax	result

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
absdiff:
    ________________
    ________________
    movq %rsi, %rax        # x>y:
    subq %rdi, %rax
    ret
.L4:                       # x<=y:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
```

Putting it all Together (pt 2)

%rdi	x
%rsi	y
%rax	Return value

```c
long max(long x, long y)
{
    long max;
    if (x > y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

```
max:
        cmpq %rdi, %rsi   # jump if
        jge else          # y >= x
        movq %rdi, %rax
        jmp done
else:
        movq %rsi, %rax
done:
        ret
```

Summary

- **x86-64** is a complex (**CISC**) architecture
  - There are 3 types instructions
    - Data transfer
    - Arithmetic
    - Control flow
  - There are 3 types of operands
    - **Registers** (%)
    - **Immediates** ($)
    - **Memory** (( ))
- **Registers** are small, fast places to store memory
  - Limited number, each with their own name