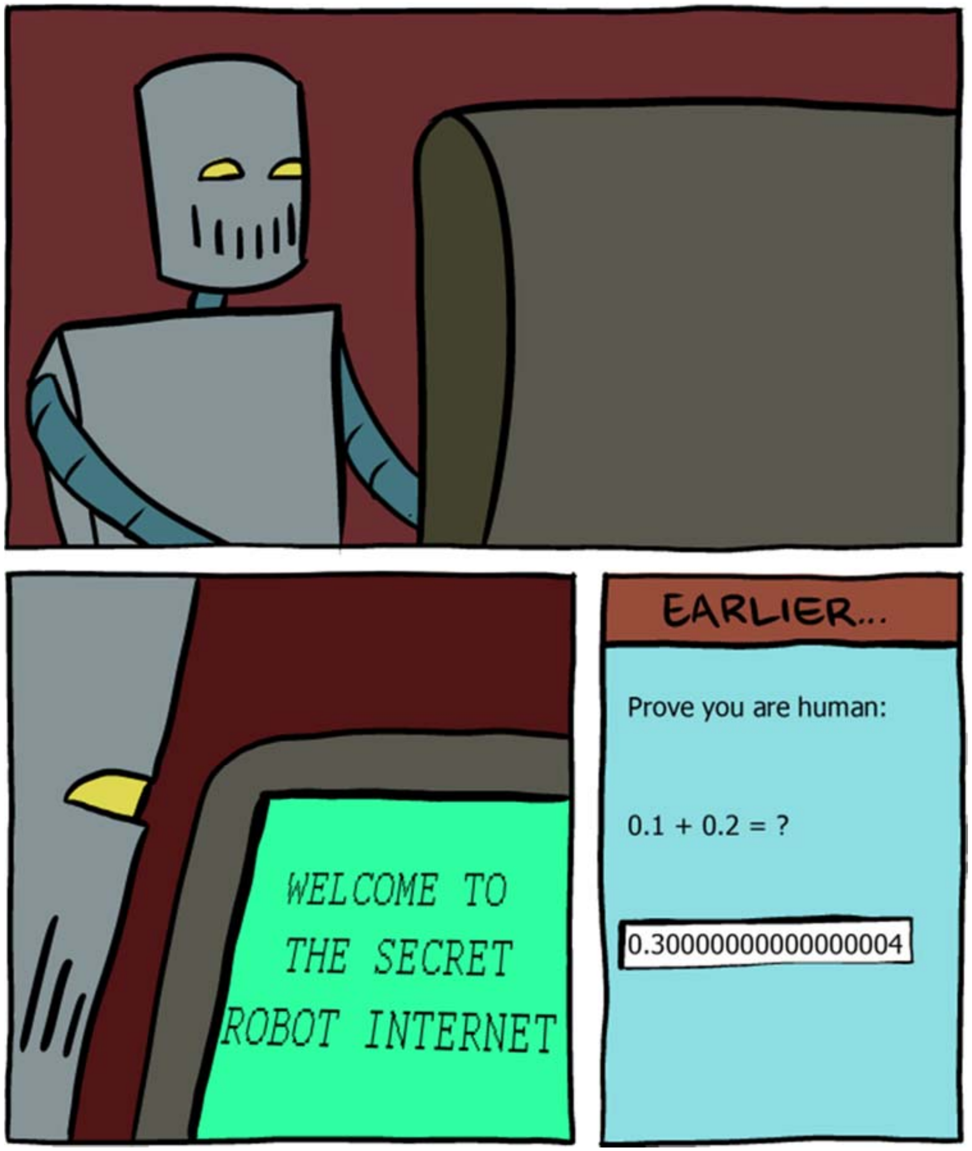
x86-64 Programming I

CSE 351 Summer 2018



http://www.smbc-comics.com/?id=2999

Instructor:

Justin Hsia

Teaching Assistants:

Josie Lee

Natalie Andreeva

Teagan Horkan

Administrivia

- Lab 1b due on Thursday (7/5)
  - Submit `bits.c`, `lab1reflect.txt`
  - Josie has OH on Thursday 1–3 pm
- Homework 2 due next Wednesday (7/11)
  - On Integers, Floating Point, and x86-64

- No lecture on Wednesday!
- Section Thursday on Floating Point

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 `ints`
    - 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 `ints` and `floats`!

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 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (float 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
```

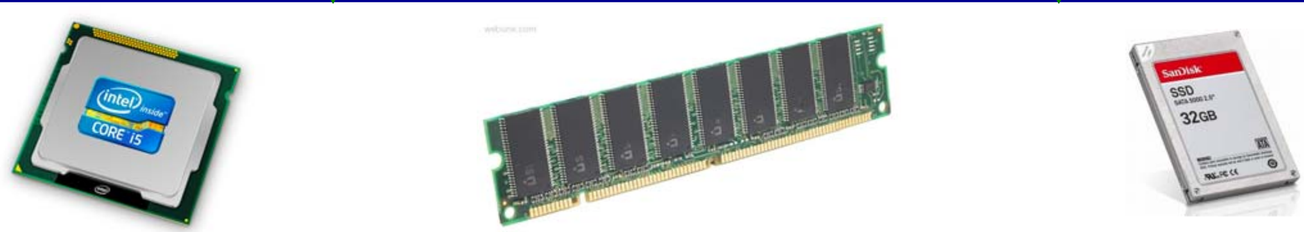
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

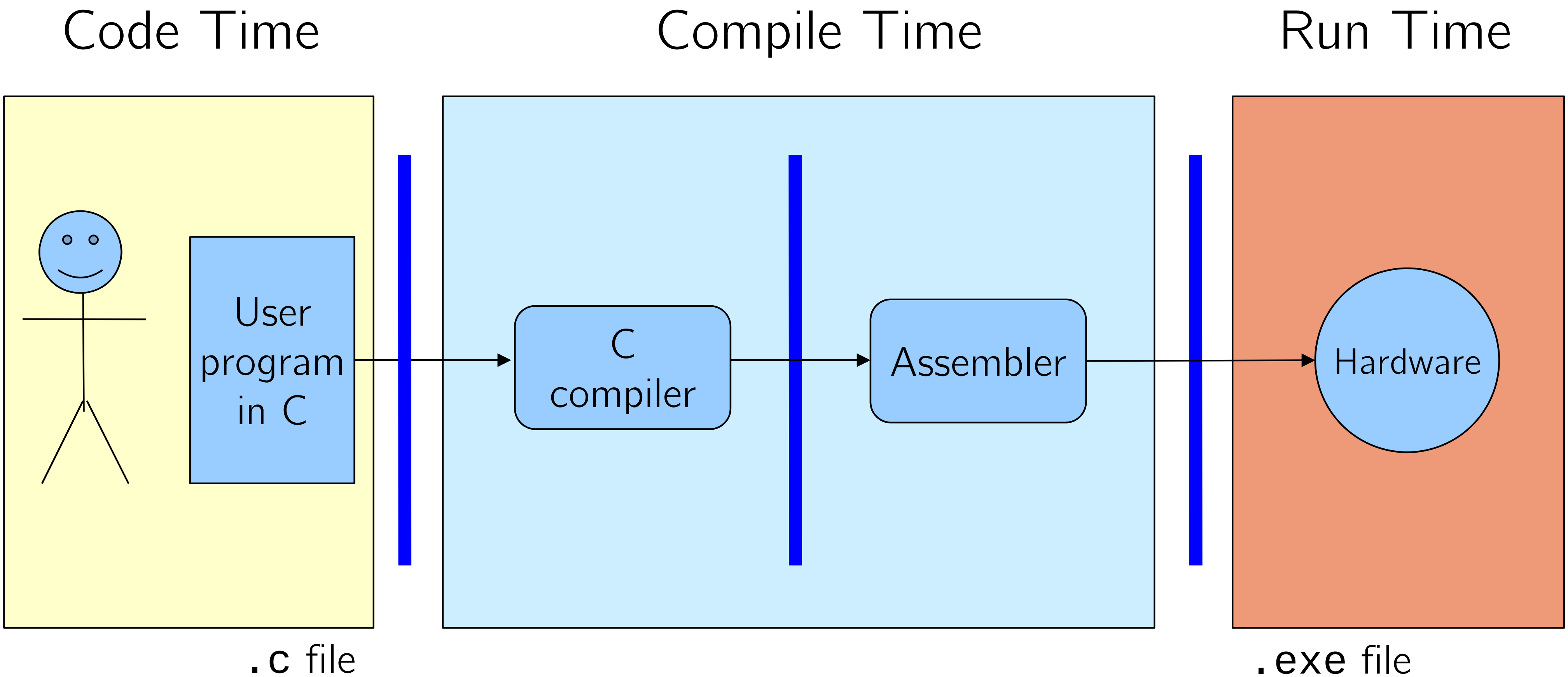
Windows 10
OS X Yosemite

Computer system:



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

Translation



What makes programs run fast(er)?

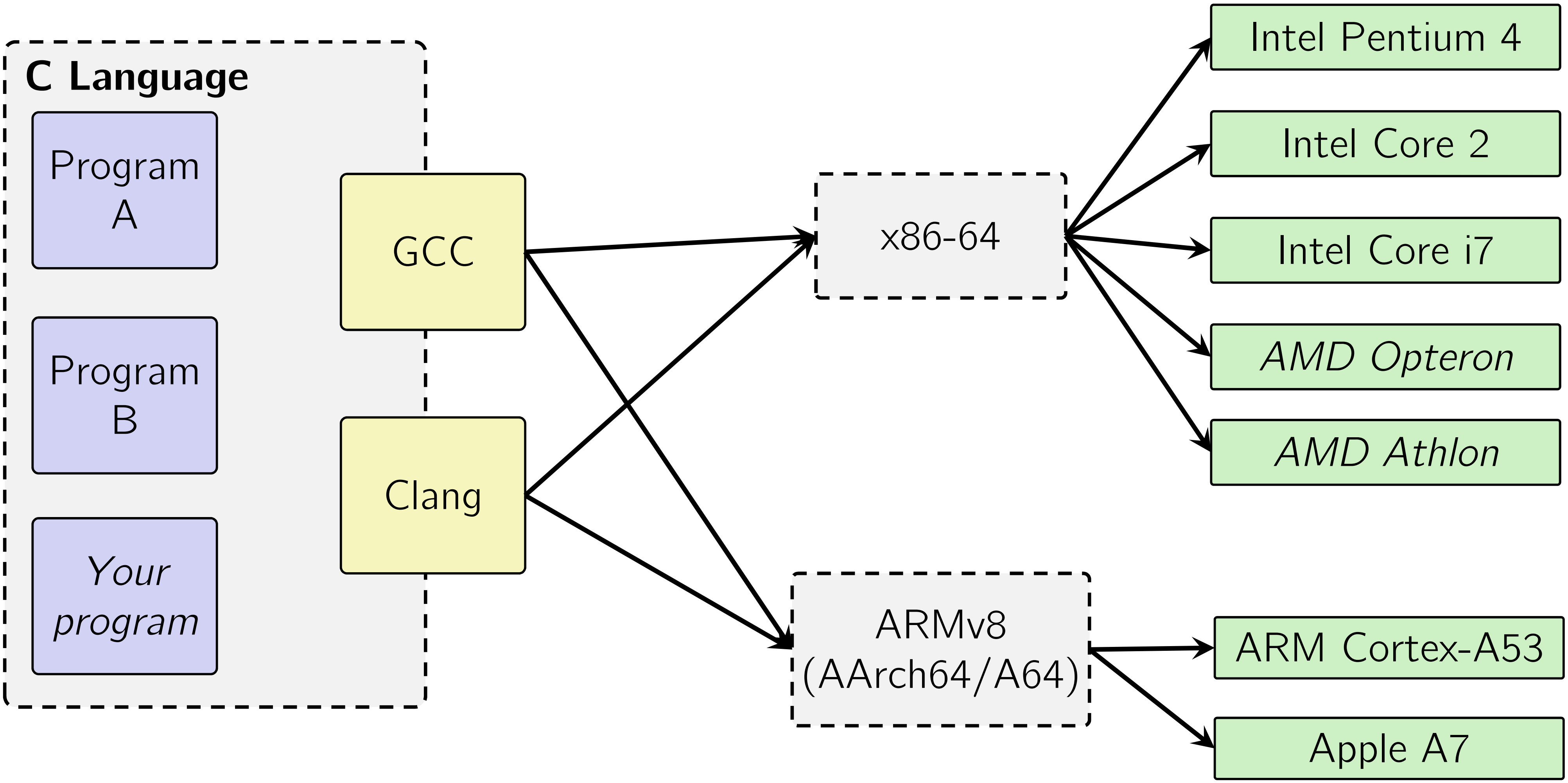
HW Interface Affects Performance

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

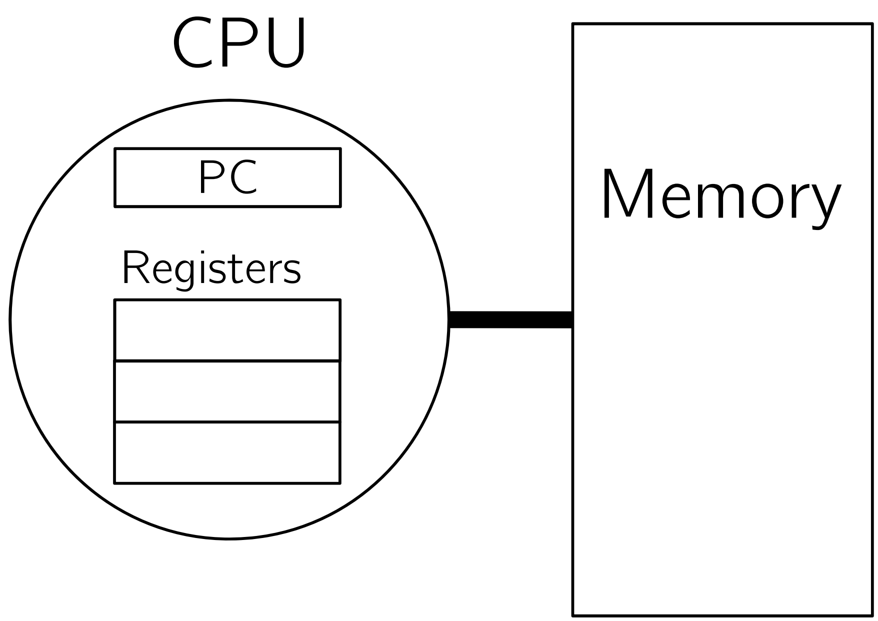
Architecture
Instruction set

Hardware
Different implementations



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

Definitions

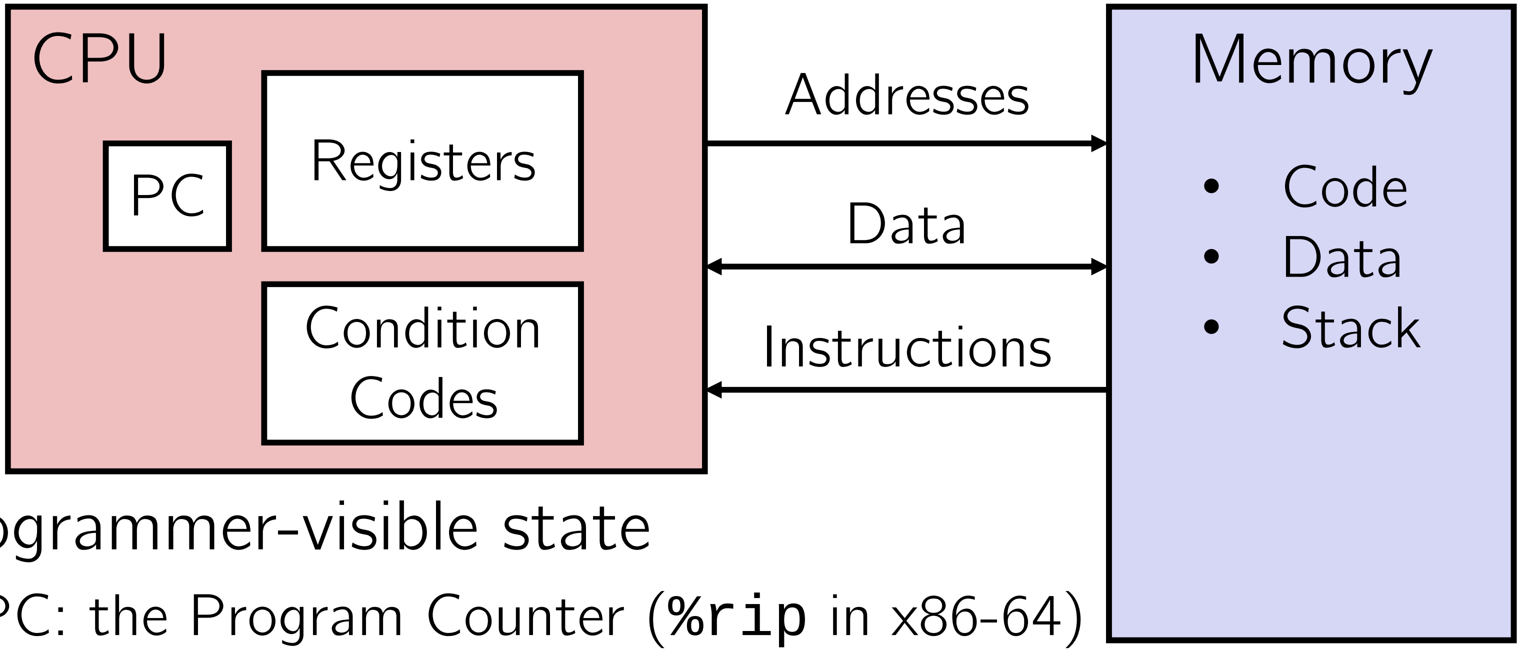
- **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - "What is directly visible to software"
- **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469, 470

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

Writing Assembly Code? In 2018???

- 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
    - Understand optimizations done/not done by the compiler
    - Understanding sources of program inefficiency
  - 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

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)

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, or 2x8, 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`)

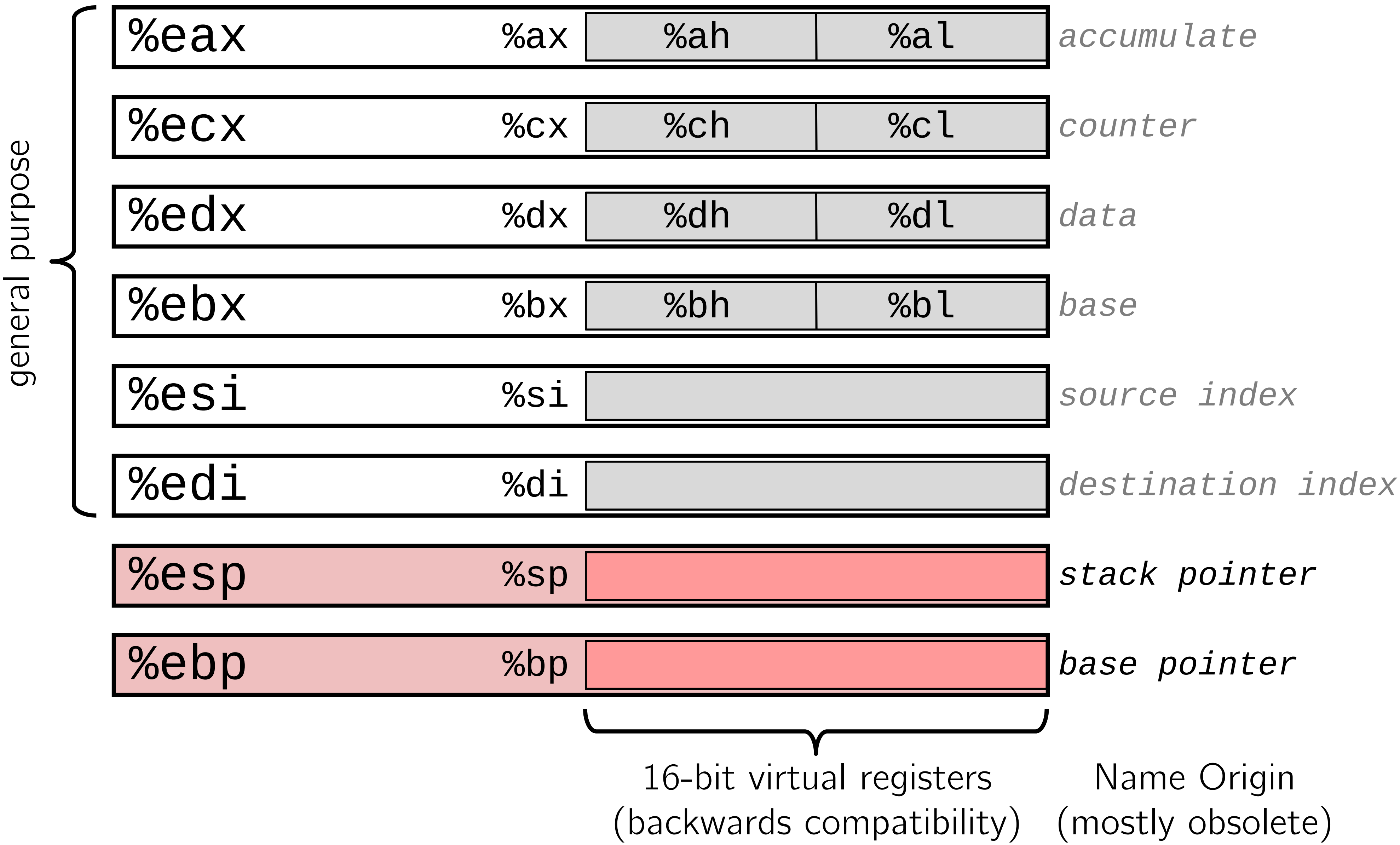
- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but *especially* x86

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)

History: IA32 Registers – 32 bits wide



Memory vs. Registers

Memory		Registers
Addresses	vs.	Names
`0x7FFFD024C3DC`		`%rdi`
Big	vs.	Small
~ 8 GiB		(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

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`

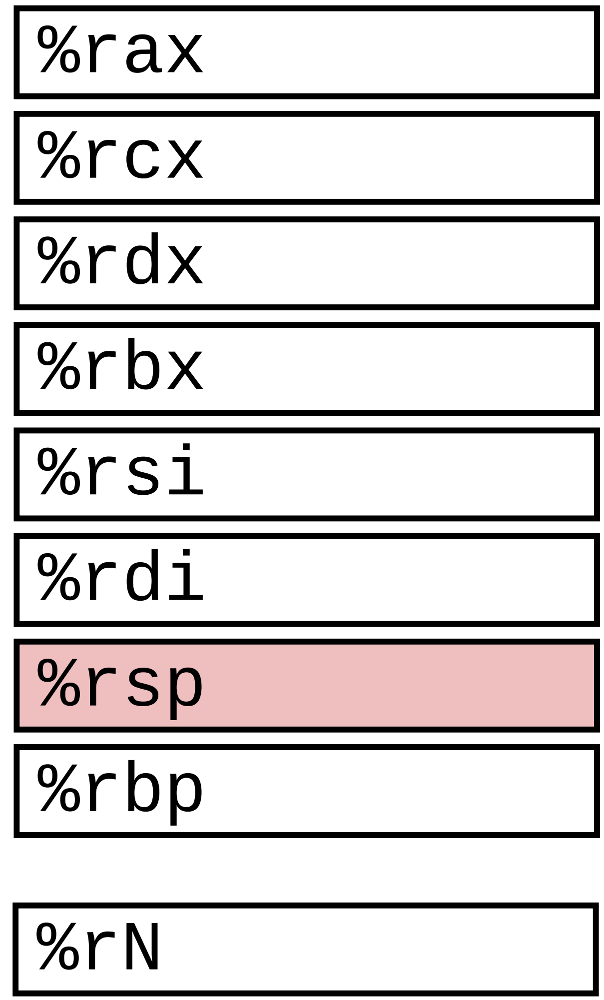
Remember: Memory is indexed just like an array of bytes!

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

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"



Moving Data

- General form: `mov_ source, destination`
  - Missing letter (_) specifies size of operands
  - Note that due to backwards-compatible support for 8086 programs (16-bit machines!), "word" means 16 bits = 2 bytes in x86 instruction names

- `movb src, dst`
  - Move 1-byte "**b**yte"
- `movw src, dst`
  - Move 2-byte "**w**ord"
- `movl src, dst`
  - Move 4-byte "**l**ong word"
- `movq src, dst`
  - Move 8-byte "**q**uad word"

movq Operand Combinations

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

- *Cannot do memory-memory transfer with a single instruction*
  - How would you do it?

x86-64 Introduction

- Arithmetic operations
- Memory addressing modes
  - `swap` example
- Address computation instruction (`lea`)

Some Arithmetic Operations

- Binary (two-operand) instructions:
  - **Maximum of one memory operand**
  - Beware argument order
  - No notion of datatypes
    - Just bits!
    - Only arithmetic vs. logical shifts
  - How do you implement "`r3 = r1 + r2`"?

Format	Computation	
`addq src, dst`	*dst = dst + src*	(*dst += src*)
`subq src, dst`	*dst = dst – src*	
`imulq src, dst`	*dst = dst * src*	signed mult
`sarq src, dst`	*dst = dst >> src*	**A**rithmetic
`shrq src, dst`	*dst = dst >> src*	**L**ogical
`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*	

↑ operand size specifier (the `q` suffix)

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

- See CSPP Section 3.5.5 for more instructions: `mulq`, `cqto`, `idivq`, `divq`

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
```

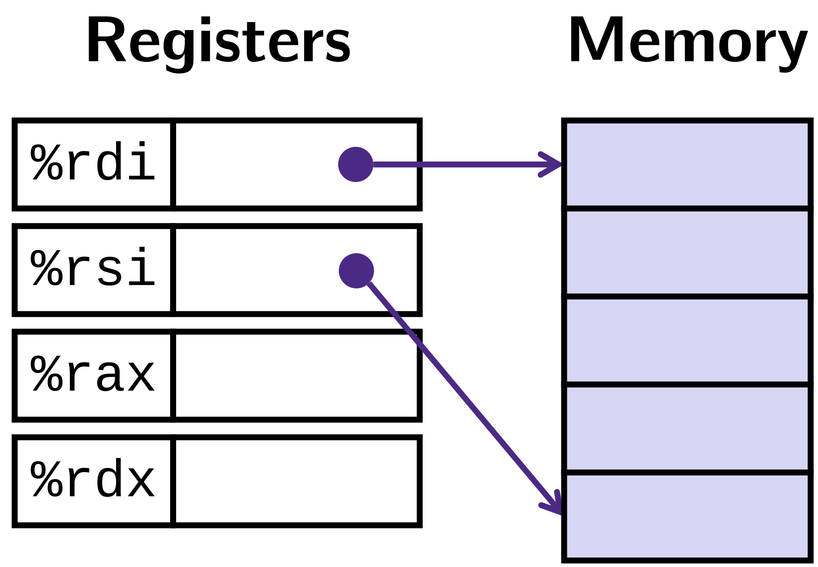
Example of Basic Addressing Modes

```
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()`

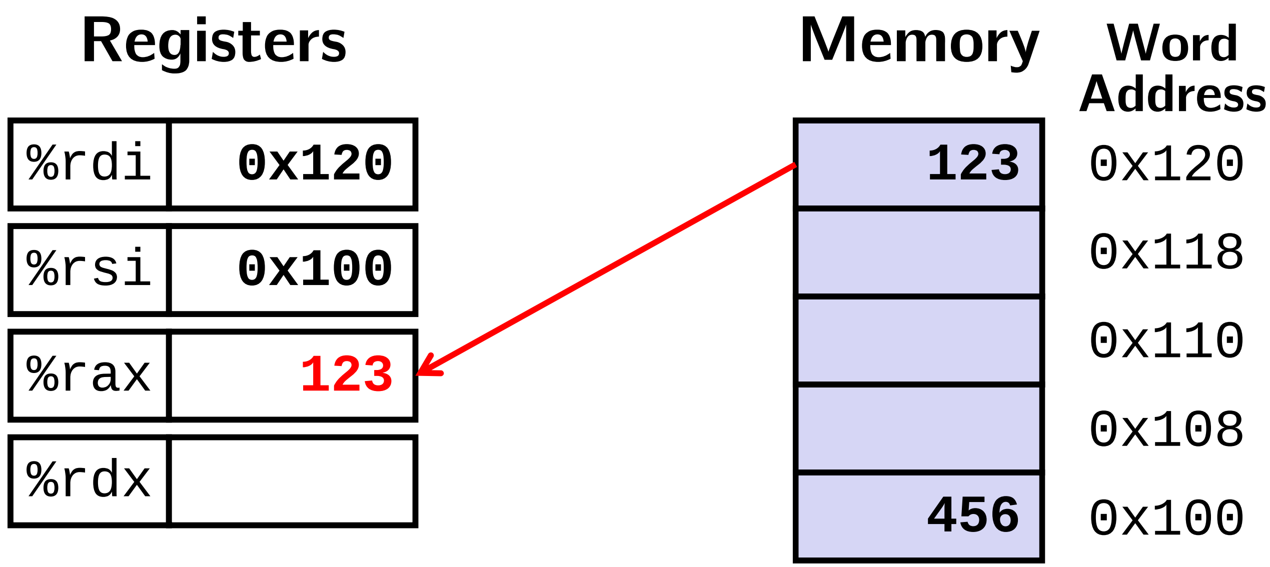
Registers

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

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

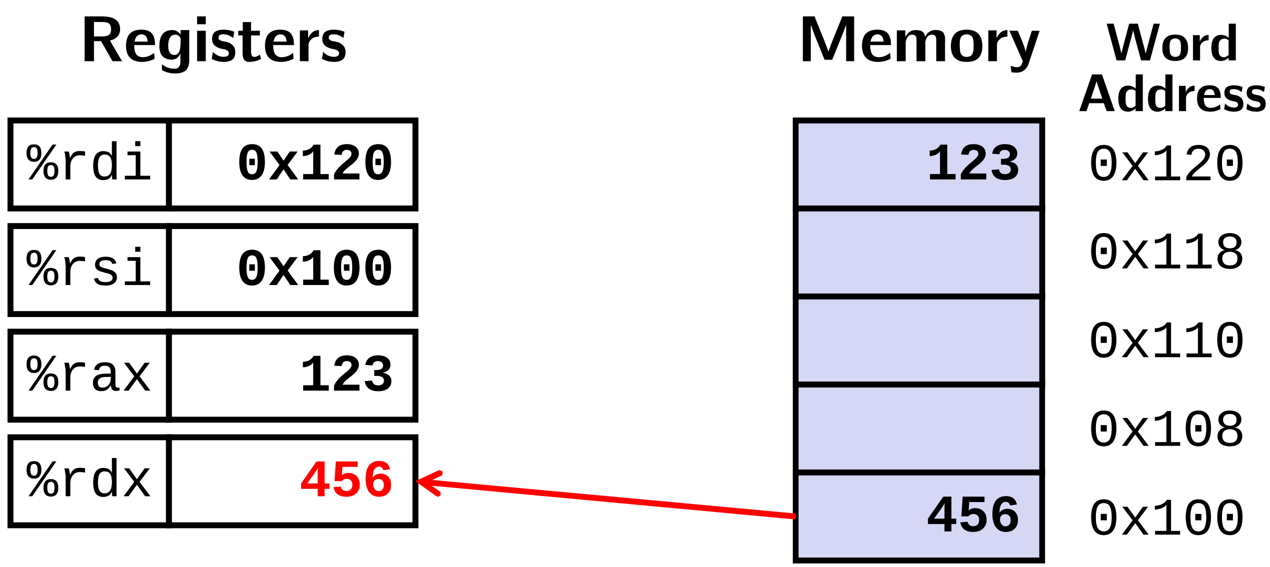
```
swap:
   movq  (%rdi), %rax  #  t0 = *xp
   movq  (%rsi), %rdx  #  t1 = *yp
   movq  %rdx, (%rdi)  # *xp =  t1
   movq  %rax, (%rsi)  # *yp =  t0
   ret
```

Understanding swap()



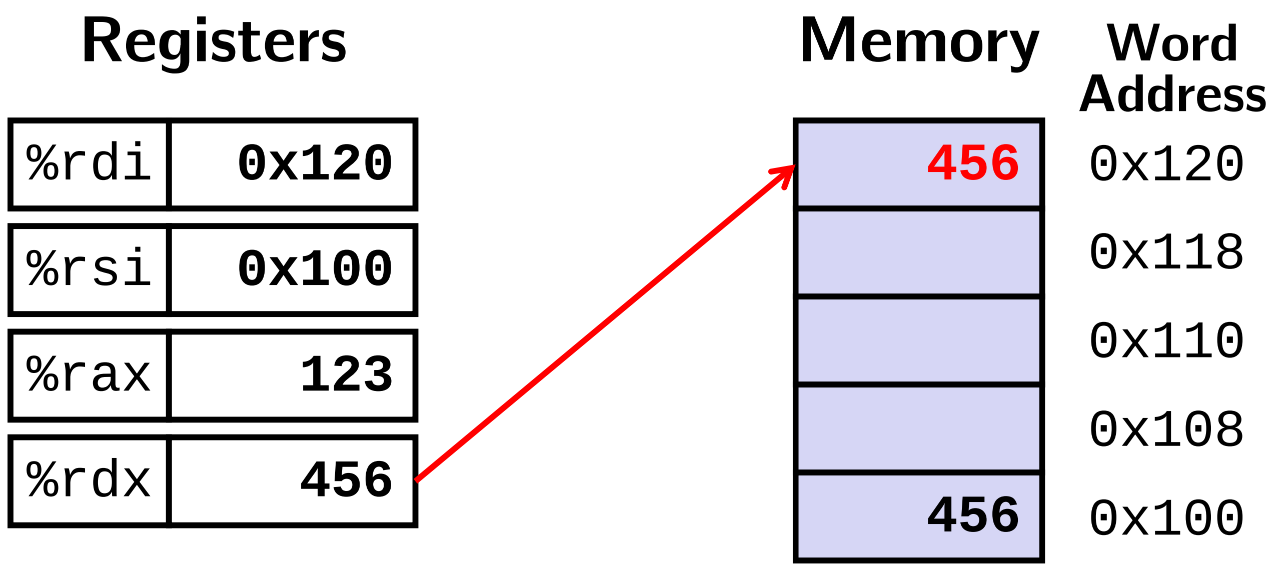

```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

Understanding `swap()`



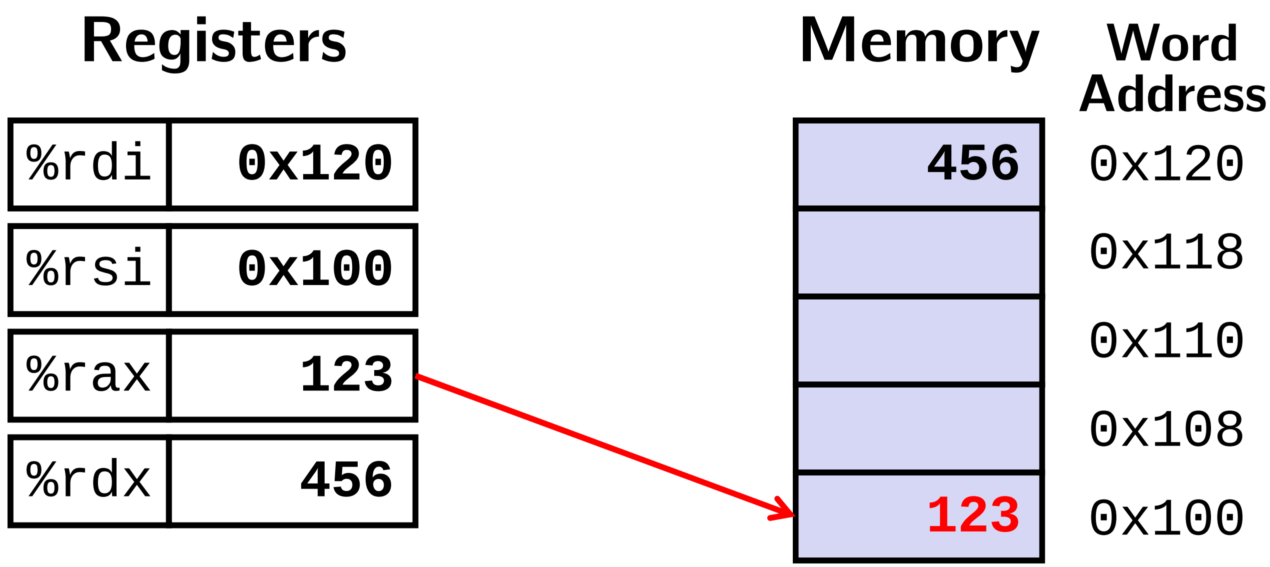

```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

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()




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

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