x86-64 Programming I

CSE 351 Summer 2020

Instructor:

Porter Jones

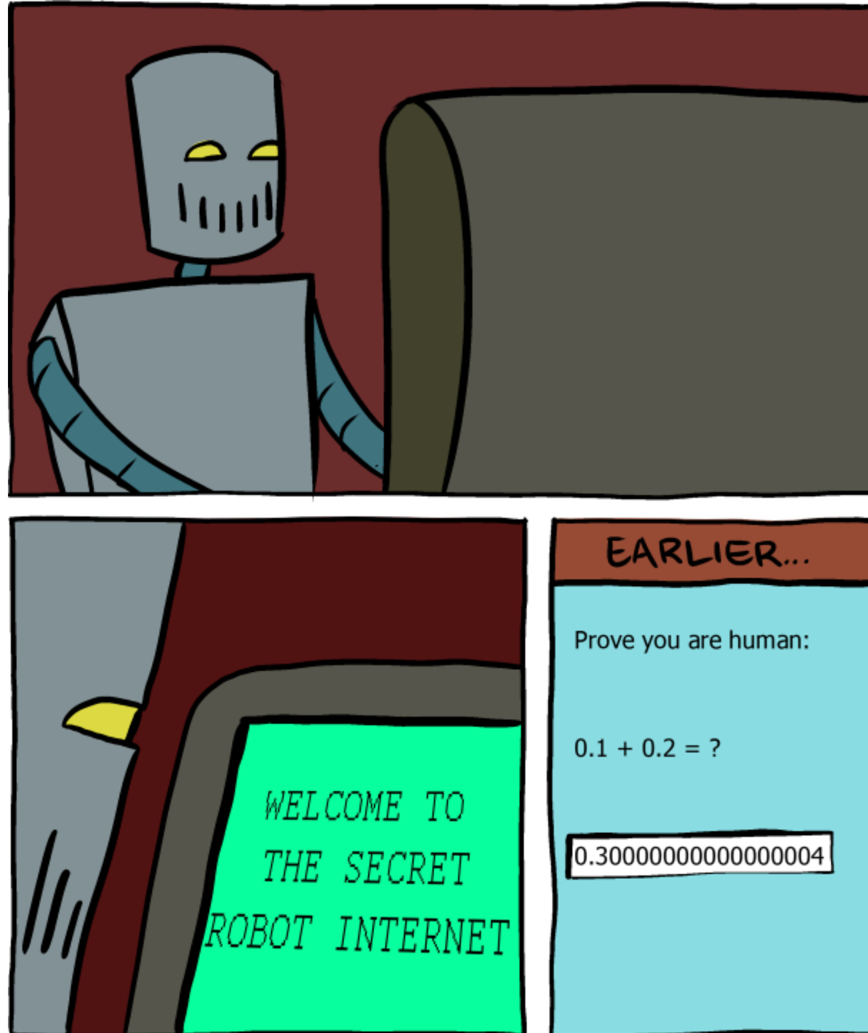
Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk



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

Administrivia

- Accommodations/Extenuating Circumstances
  - These are unfortunate and difficult times for many people for a number of reasons
  - Please contact us if you have unforeseen difficulties that may affect your ability to turn in assignments/stay on track
    - The earlier you contact us the better! (i.e. before an assignment is due, right when a deadline is missed, etc.)
    - See course syllabus for more information
  - It should go without saying there are more important things than CSE 351, we want to work with you to find the balance that works best for you given these difficult times

Administrivia

- Questions doc: https://tinyurl.com/CSE351-7-8

- hw6 & hw7 due Friday (7/10) – 10:30am
- hw8 due Monday (7/13) – 10:30am

- Lab 1b due Friday at 11:59pm (7/10)
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Breflect.txt`

Administrivia

- Unit Summary 1 Due Wednesday 7/15
  - Submitted via Gradescope

- Unit Summaries are meant to encourage review/reflection of material in place of exams
  - See course website for specification and instructions, including small examples

- Grading very lenient and forgiving, mostly based on effort! If you put in a solid effort you will likely get full credit

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

- Two common levels of precision:

`float`	`1.0f`	single precision (32-bit)
`double`	`1.0`	double precision (64-bit)

- `#include <math.h>` to get `INFINITY` and `NAN` constants   <float.h> for additional constants

- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!

you decide for your program

instead use abs(f1 - f2) < 2^{-20}

↑ some arbitrary threshold

Floating Point Conversions in C

- Casting between `int`, `float`, and `double` **changes** the bit representation tries to preserve value
  - `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)

Floating Point and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for (i = 0; i < 10; i++)
    f2 += 1.0/10.0;

  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf("f1 = %10.9f\n", f1);
  printf("f2 = %10.9f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

rounding issue

$10 * \frac{1}{10} = 1$?

don't use == w/ floats

trick to print out bits

$f3 = 1e30 + 1e-30$

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!

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 (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)

Summary

- Floating point approximates real numbers:

  - Handles large numbers, small numbers, special numbers
  - Exponent in biased notation (bias = 2^{w-1}–1)
    - Size of exponent field determines our representable *range*
    - Outside of representable exponents is *overflow* and *underflow*
  - Mantissa approximates fractional portion of binary point
    - Size of mantissa field determines our representable *precision*
    - Implicit leading 1 (normalized) except in special cases
    - Exceeding length causes *rounding*

Summary

E	M	Meaning
0x00	0	± 0
0x00	non-zero	± denorm num
0x01 – 0xFE	anything	± norm num
0xFF	0	± ∞
0xFF	non-zero	NaN

- Floating point encoding has many limitations
  - Overflow, underflow, rounding
  - Rounding is a HUGE issue due to limited mantissa bits and gaps that are scaled by the value of the exponent
  - Floating point arithmetic is NOT associative or distributive
- Converting between integral and floating point data types *does* change the bits

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();
```

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

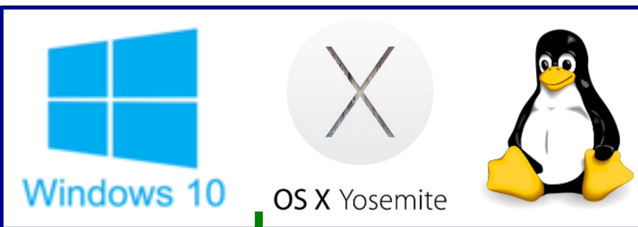
Assembly language:

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

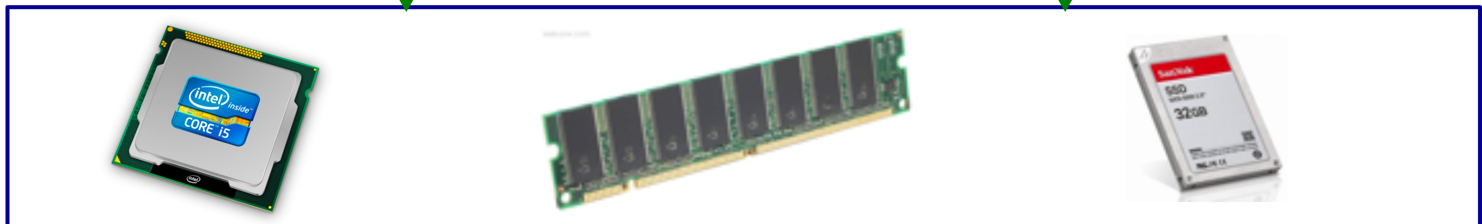
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



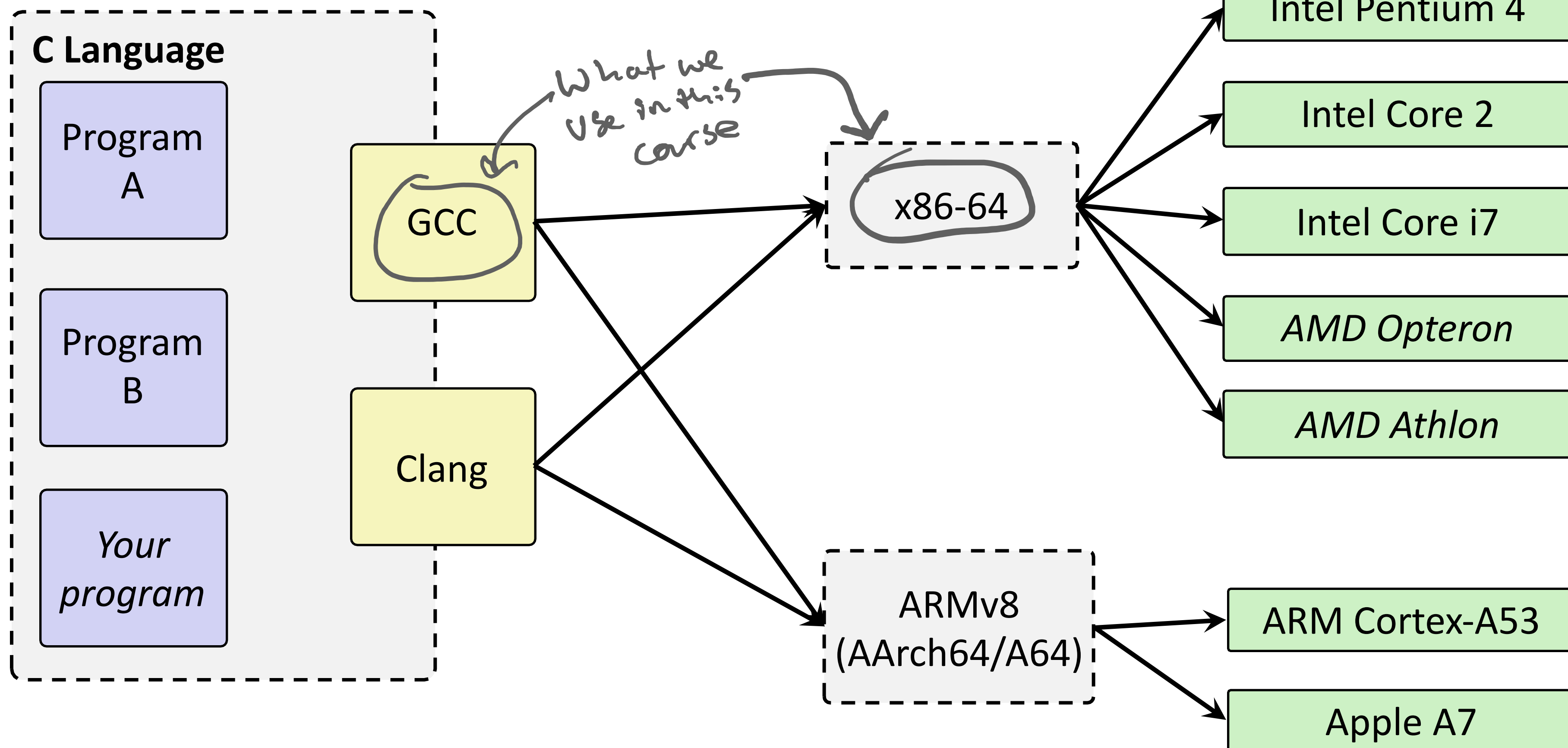
Architecture Sits at the Hardware Interface

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

Architecture
Instruction set
interface for hardware

Hardware
Different implementations

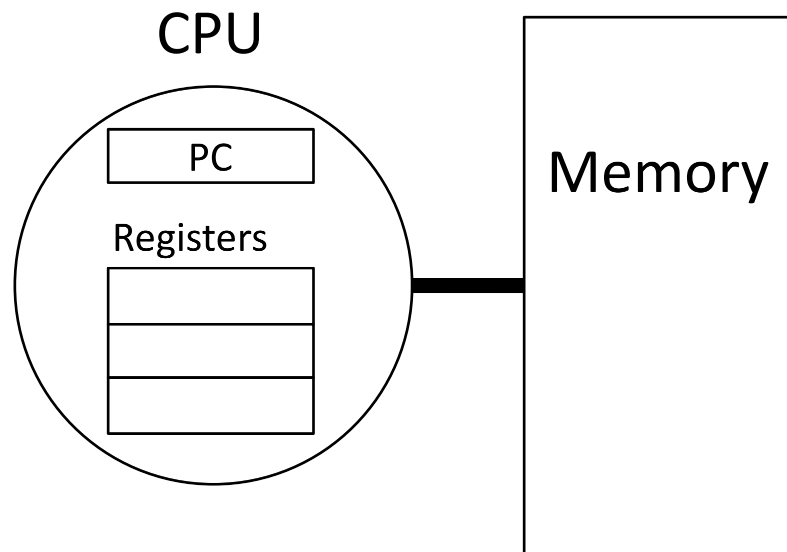


Definitions

- **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

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

Risc-V: ISA geared towards open source hardware

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

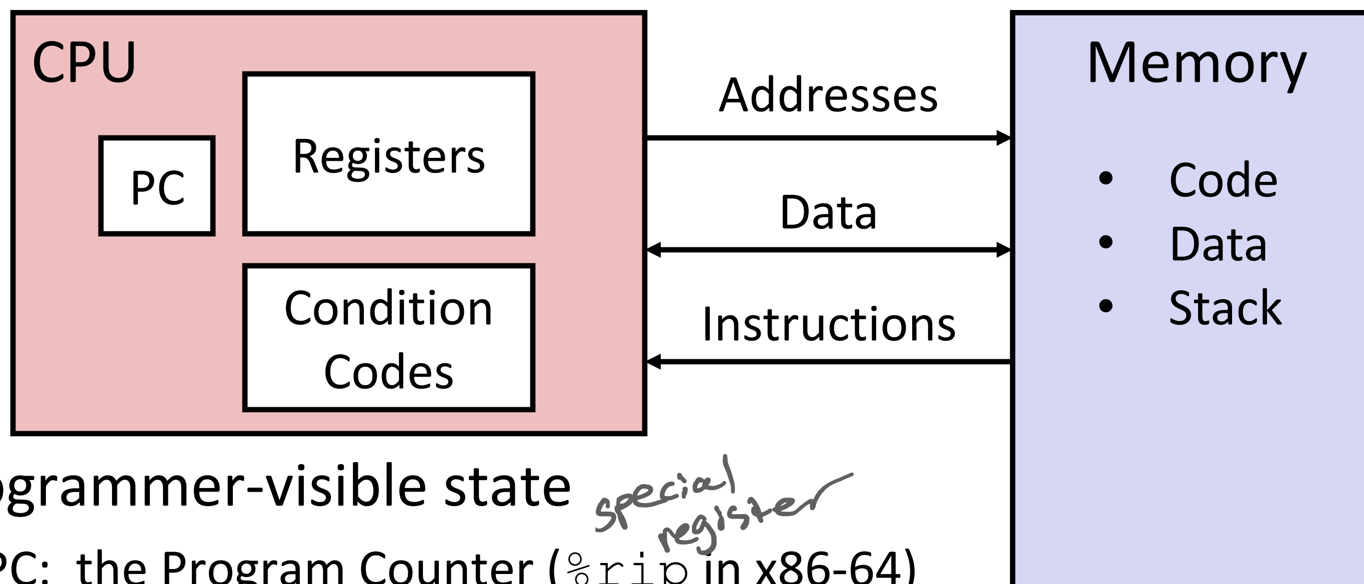
Designer	MIPS Technologies, Inc.
Bits	64-bit (32→64)
Introduced	1981; 35 years ago
Design	RISC
Type	Register-Register
Encoding	Fixed
Endianness	Bi

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

Writing Assembly Code? In 2020???

- 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
- 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, …)

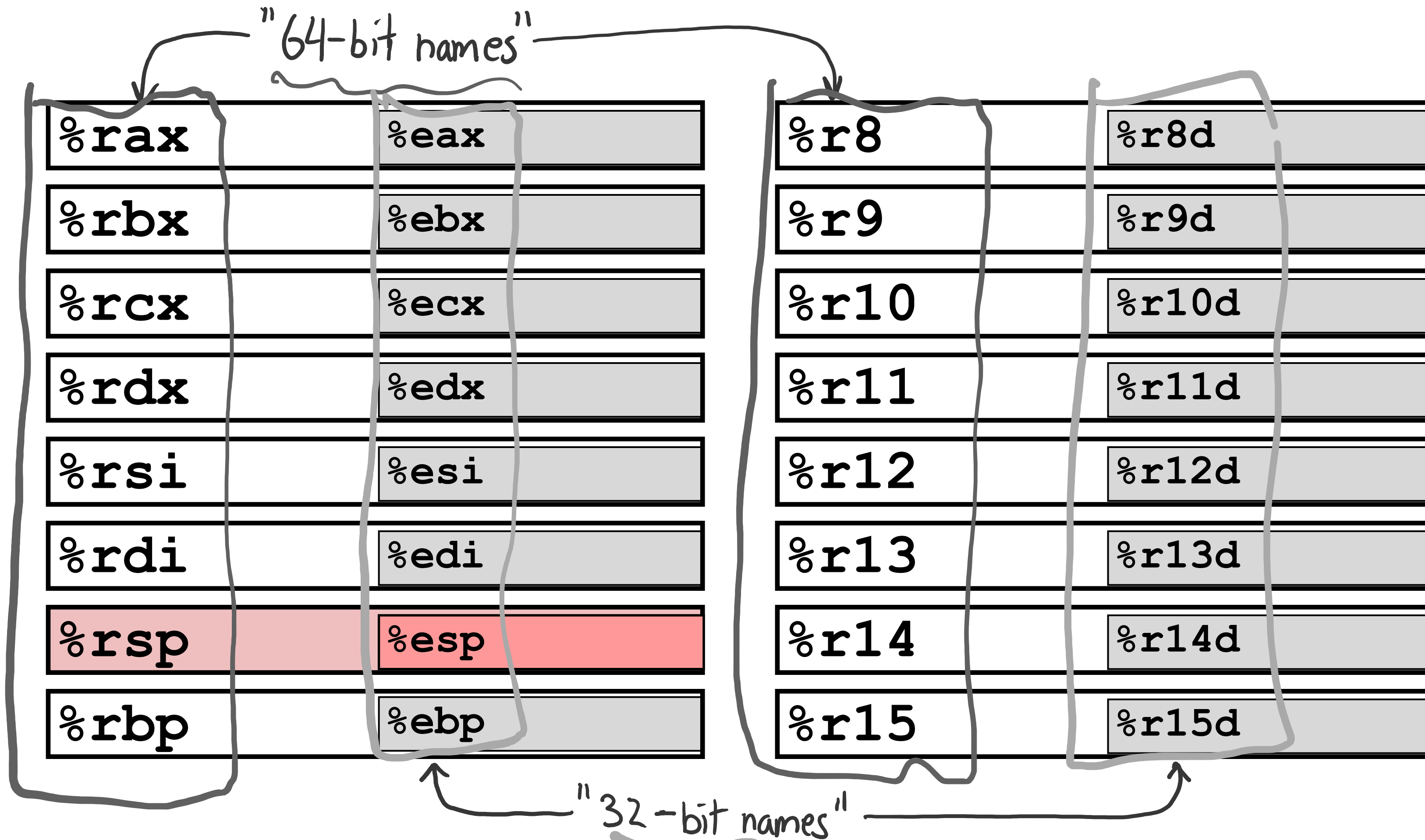
Not covered In 351

- 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?

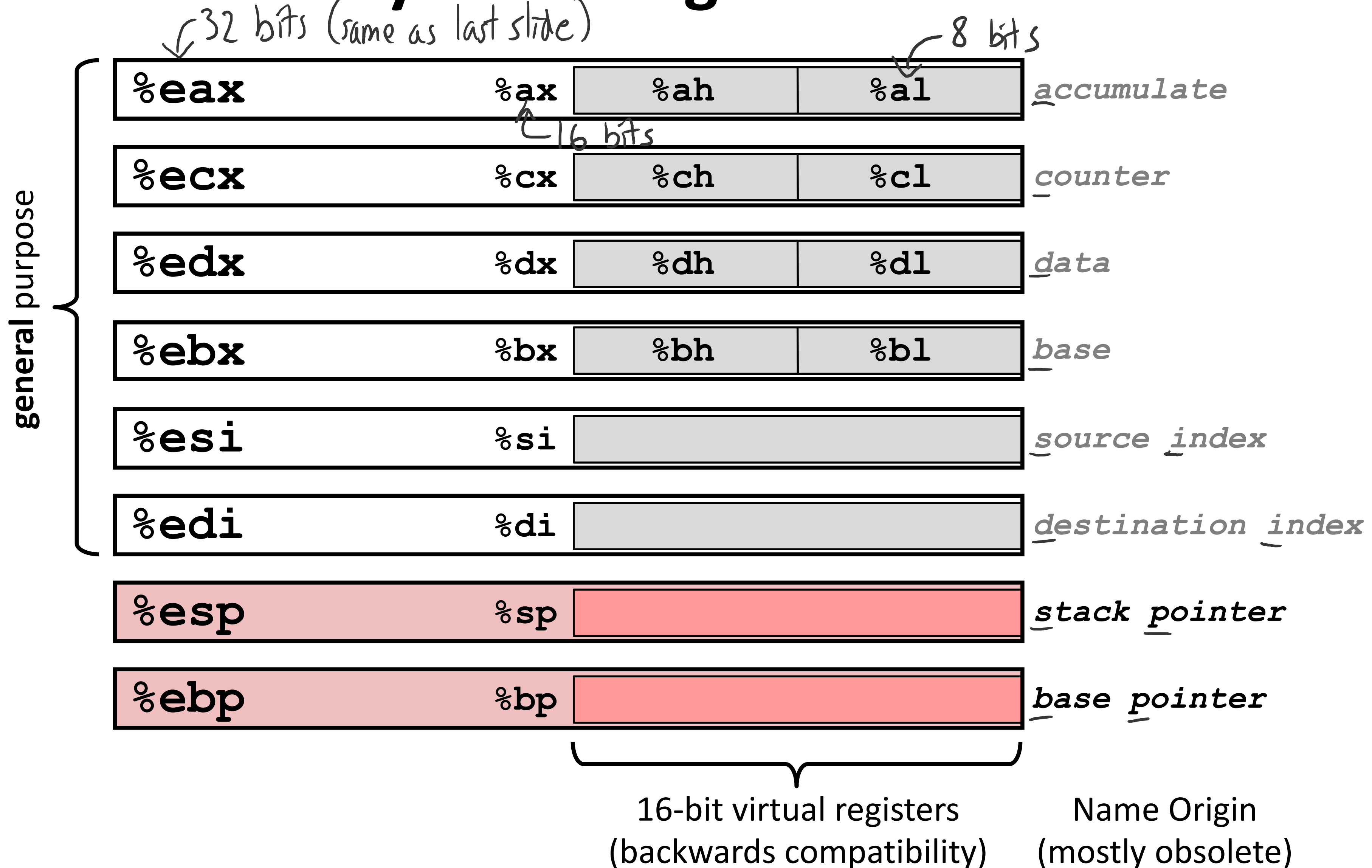
- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)
- Registers have *names*, not *addresses*
  - In assembly, they start with `%` (*e.g.* `%rsi`)
- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but *especially* x86 only 16 integral registers . . . .

x86-64 Integer Registers – 64 bits wide



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

Some 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`

indexed by address

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

instruction op1, op2

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"

like dereference

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

`%rN`

x86-64 Introduction

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

~~Moving~~ Data

Copying

- 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
  - Lots of these in typical code

Handwritten notes: "instruction name" (arrow to `mov`), "width specifier" (arrow to `_`), "copies data" (arrow from source to destination)

Size (bytes)	1	2	4	8
letter	b	w	l	q

- `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"

Operand Combinations

8 byte width

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

① mem → reg movq (%rdi), %rax
② reg → mem movq %rax, (%rsi)

Some Arithmetic Operations

ways to set to 0
movq $0, %rax
subq %rax, %rax
andq $0, %rax
xorq %rax, %rax
imulq $0, %rax

- Binary (two-operand) Instructions:
  - **Maximum of one memory operand**
  - Beware argument order!
  - No distinction between signed and unsigned
    - Only arithmetic vs. logical shifts
  - How do you implement "`r3 = r1 + r2`"? See next slide

Imm, Reg, or Mem

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*	

operation → or; operand size specifier (b, w, l, q)

Polling Question [Asm I – a]

- Assume: r3 is in `%rcx`, r1 is in `%rax`, and r2 is in `%rbx` which of the following would implement:

  r3 = r1 + r2

  - Vote at http://pollev.com/pbjones

A. `addq %rax, %rbx, %rcx`

B. `addq %rcx, %rax, %rbx`

C. `movq %rax, %rcx`
   `addq %rbx, %rcx`

D. `movq (%rbx), %rcx`
   `addq (%rax), %rcx`

E. We're lost…

invalid Syntax

r3 = r1 ✓
r3 += r2

memory dereference which we don't want

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

conventions

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

don't need to preserve x and y

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

move result to %rax

src dst

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

return

Example of Basic Addressing Modes

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

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

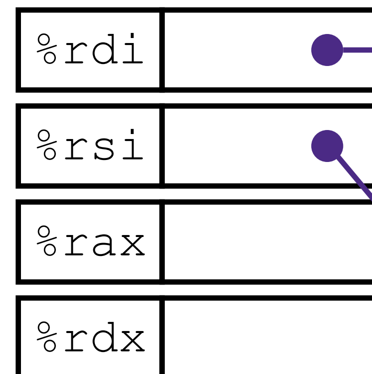
Mem operands

Understanding `swap()`

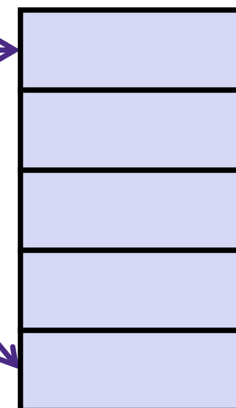
```
void swap(long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```

(handwritten: rdi above xp, rsi above yp)

Registers



Memory



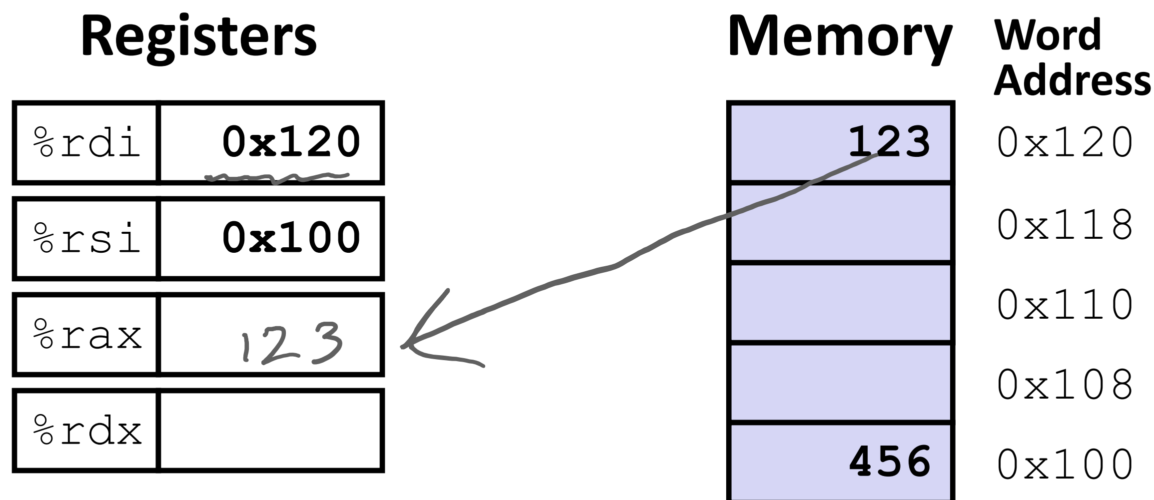
```
swap:
   movq   (%rdi), %rax
   movq   (%rsi), %rdx
   movq   %rdx, (%rdi)
   movq   %rax, (%rsi)
   ret
```

(handwritten: src, dst above operands)

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

(handwritten: temp storage → t0, t1)

Understanding swap()

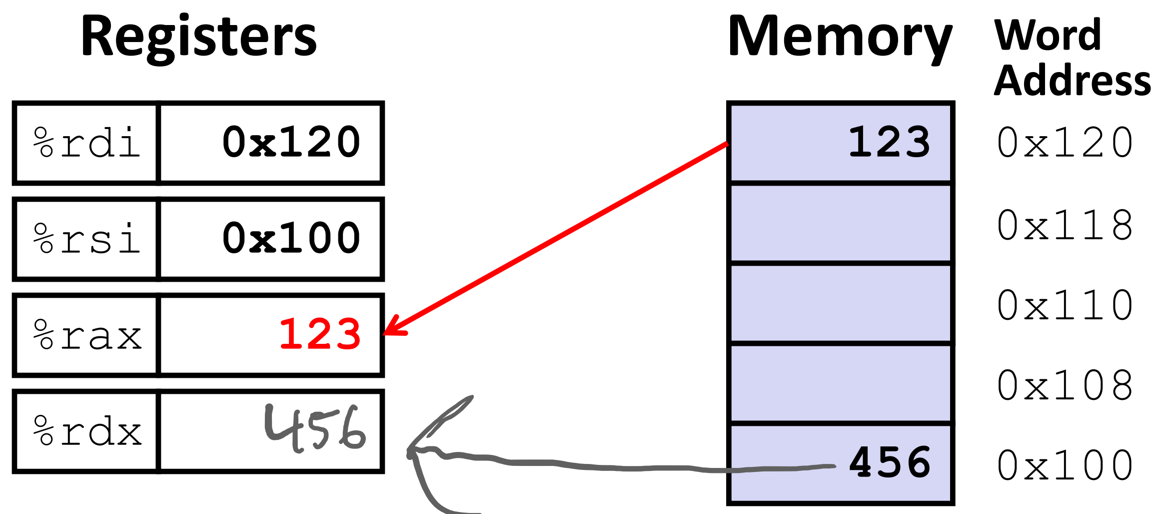


like dereference

src dst

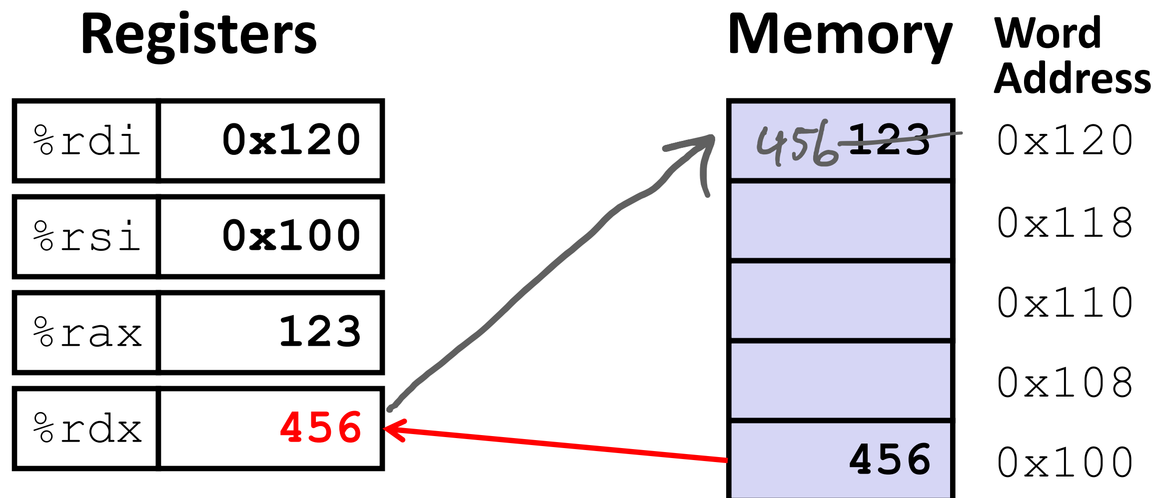
```
swap:
    movq    (%rdi), %rax    #  t0 = *xp
    movq    (%rsi), %rdx    #  t1 = *yp
    movq    %rdx, (%rdi)    # *xp =  t1
    movq    %rax, (%rsi)    # *yp =  t0
    ret
```

Understanding swap()




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

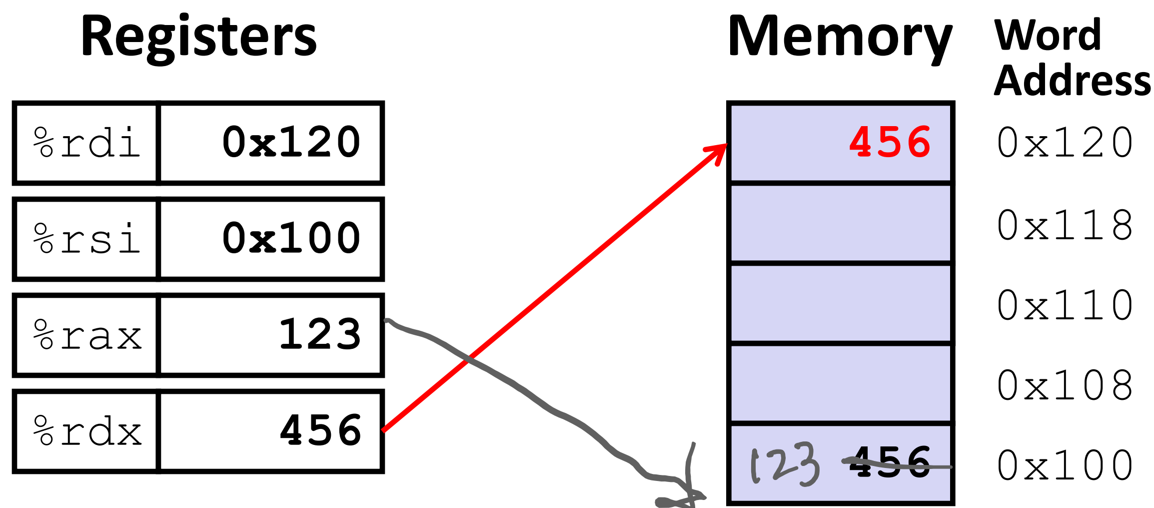
Understanding swap()




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

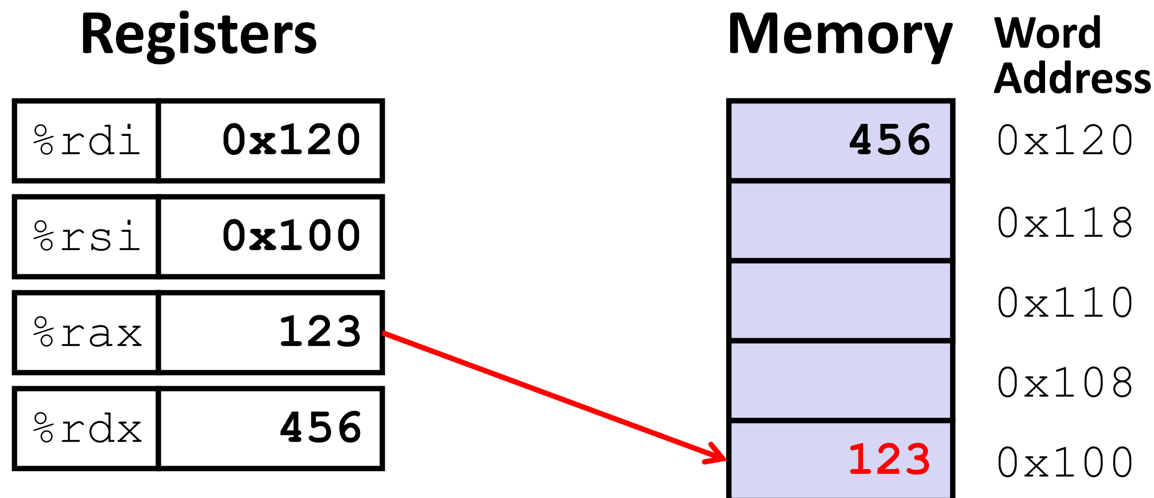
like dereference

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
```

Memory Addressing Modes: Basic

- **Indirect:** (R) Mem[Reg[R]]
  - Data in register R specifies the memory address
  - Like pointer dereference in C
  - Example: **movq** (%rcx), %rax

value/address stored in register

no space

- **Displacement:** D(R) Mem[Reg[R]+D]
  - Data in register R specifies the *start* of some memory region
  - Constant displacement D specifies the offset from that address
  - Example: **movq** 8(%rbp), %rdx

Complete Memory Addressing Modes

ar[i] = *(ar+i) = *(ar + i * sizeof(type))

- **General:**
  - D(Rb,Ri,S) Mem[Reg[Rb]+Reg[Ri]*S+D]
    - Rb: Base register (any register)
    - Ri: Index register (any register except %rsp)
    - S: Scale factor (1, 2, 4, 8) *– why these numbers?*
    - D: Constant displacement value (a.k.a. immediate)

- **Special cases** (see CSPP Figure 3.3 on p.181)
  - D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D] (S=1)
  - (Rb,Ri,S) Mem[Reg[Rb]+Reg[Ri]*S] (D=0)
  - (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]] (S=1,D=0)
  - (,Ri,S) Mem[Reg[Ri]*S] (Rb=0,D=0)

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

`D(Rb,Ri,S)` →
Mem[Reg[`Rb`]+Reg[`Ri`]`*S+D`]

Expression	Address Computation	Address
`0x8(%rdx)`	0xf000 + 0x8	0xf008
`(%rdx,%rcx)`	0xf000 + 0x0100	0xf100
`(%rdx,%rcx,4)`	0xf000+0x0400	0xf400
`0x80(,%rdx,2)`	0xf000*2 + 0x80	0x1e080

shift trick

0xf000 << 1 = 0x1e000

0b 1111 0000 0000 0000   0b1 1110 0000 0000 0000

Summary

- x86-64 is a complex instruction set computing (CISC) architecture
  - There are 3 types of operands in x86-64
    - Immediate, Register, Memory
  - There are 3 types of instructions in x86-64
    - Data transfer, Arithmetic, Control Flow

- **Memory Addressing Modes:** The addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways
  - *Base register*, *index register*, *scale factor*, and *displacement* map well to pointer arithmetic operations

On your index card:

- In general, pace of class is: 1 too **fast**
  2 kind of fast
  3 just right
  4 kind of slow
  5 too **slow**

- Please **Keep** doing this:
- Please **Quit** doing this:
- Please **Start** doing this: