

x86-64 Programming I

CSE 351 Winter 2022

Instructor:

Sam Wolfson

Teaching Assistants:

Angela Xu

Anirudh Kumar

Catherine Guevara

Dara Stotland

Harrison Bay

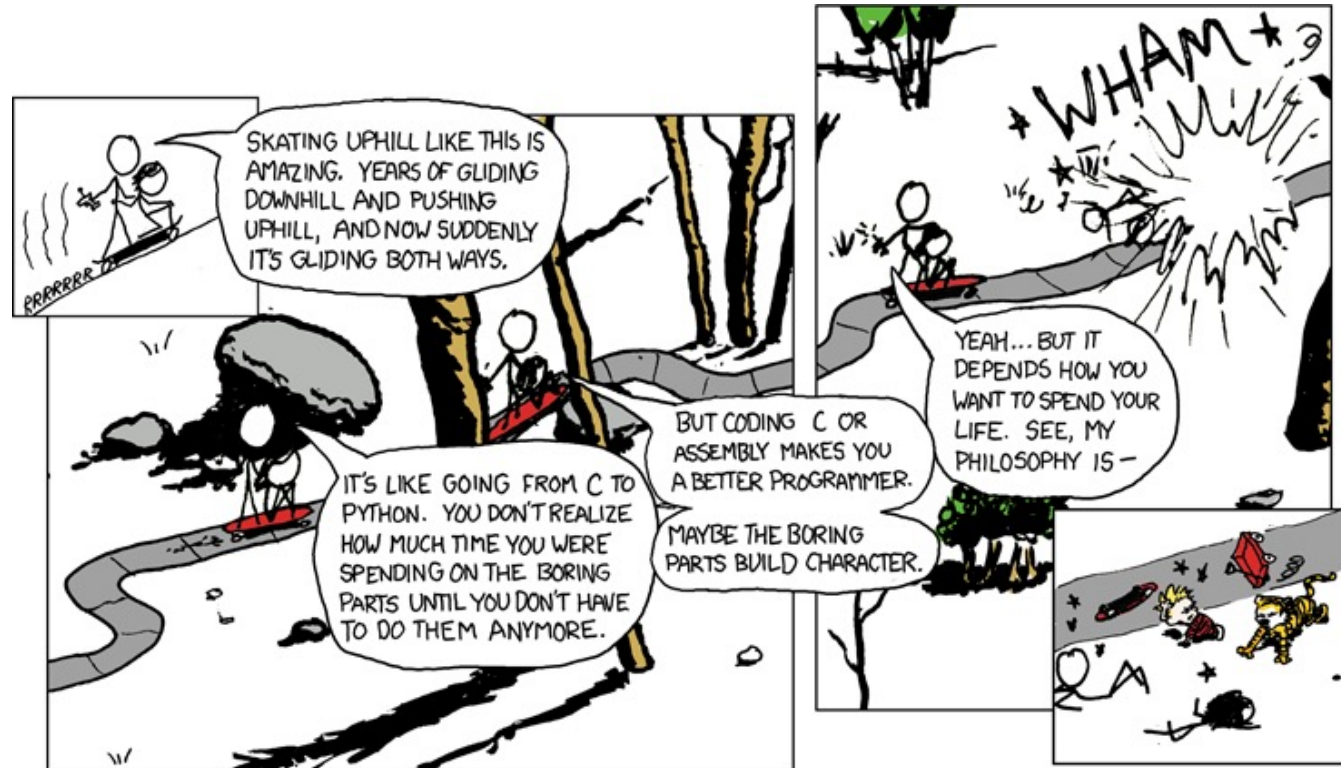
Ian Hsiao

Kevin Wang

Mara Kirdani-Ryan

Nick Durand

Sanjana Sridhar



<http://xkcd.com/409/>

Relevant Course Information

- ❖ hw5 due tonight, hw6 due Friday
- ❖ Lab 1a due tonight!
 - Submit `pointer.c` and `lab1Asynthesis.txt`
 - Make sure there are no lingering `printf` statements in your code!
 - Make sure you submit *something* to Gradescope before the deadline and that the file names are correct
 - Can use late day tokens to submit up until Friday 11:59 pm
- ❖ `lateDays++`
 - Sorry for the inconvenience!
 - This brings you up to **6 late days total**

Relevant Course Information

- ❖ Lab 1b due Friday (10/18) at 11:59 pm
 - No major programming restrictions, but should avoid magic numbers by using C macros (`#define`)
 - For debugging, can use provided utility functions `print_binary_short()` and `print_binary_long()`
 - Pay attention to the output of `aisle_test` and `store_test` – failed tests will show you actual vs. expected



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 ints are representable)
 - `long` → `double` *size of long happens to correspond to word size on x86*
 - 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 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)

Reading Review

❖ Terminology

- Instruction Set Architecture (ISA): CISC vs. RISC
- Instructions: data transfer, arithmetic/logical, control flow
 - Size specifiers: b, w, l, q
- Operands: immediates, registers, memory
 - Memory operand: displacement, base register, index register, scale factor

❖ Questions from the Reading?

Review Questions

- ❖ Assume that the register %rax currently holds the value 0x 01 02 03 04 05 06 07 08

~~07~~ ~~08~~
 F8 F7

← only overwrite 2 lowest bytes
- ❖ Answer the questions about the following instruction (`<instr> <src> <dst>`): "word" (2B)

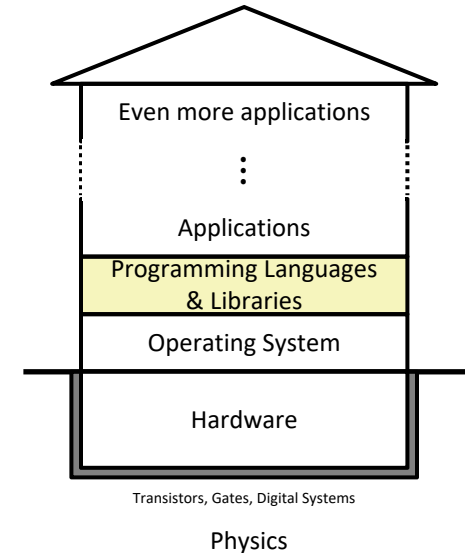
^{xor}
 xorw \$-1, %ax

 - Operation type: logical (xor)
 - Operand types: src: immediate, dst: register
 - Operation width: 2B
 - (extra) Result in %rax:

$$\begin{array}{r}
 0xFFFF \leftarrow -1 \text{ in 2's complement} \\
 \wedge 0x0708 \\
 \hline
 0xF8F7
 \end{array}$$

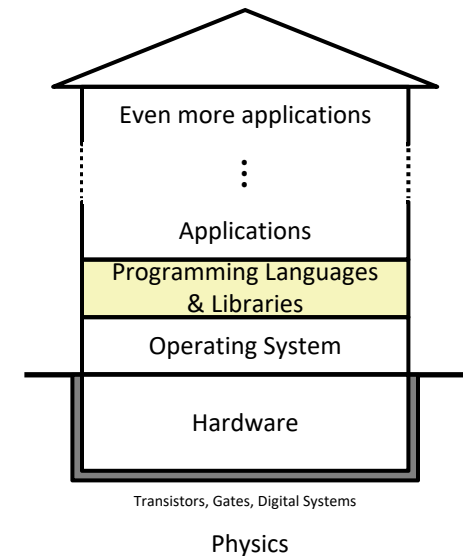
The Hardware/Software Interface

- ❖ Topic Group 1: **Data**
 - Memory, Data, Integers, Floating Point, Arrays, Structs
- ❖ Topic Group 2: **Programs**
 - **x86-64 Assembly**, Procedures, Stacks, Executables
- ❖ Topic Group 3: **Scale & Coherence**
 - Caches, Processes, Virtual Memory, Memory Allocation



The Hardware/Software Interface

- ❖ Topic Group 2: **Programs**
 - **x86-64 Assembly**, Procedures, Stacks, Executables



- ❖ How are programs created and executed on a CPU?
 - How does your source code become something that your computer understands?
 - How does the CPU organize and manipulate local data?

Definitions

- ❖ **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” or “blueprint” between hardware and software

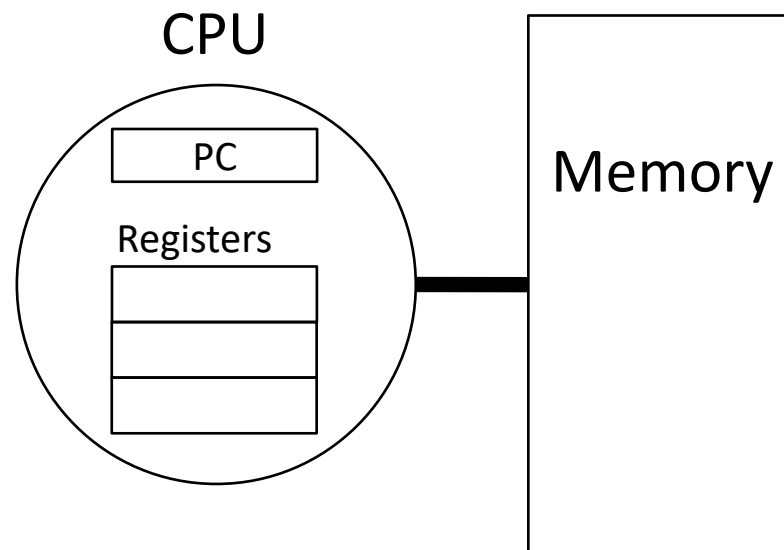
- ❖ **Microarchitecture:** Implementation of the architecture
 - CSE/EE 469

Instruction Set Architectures (Review)

❖ 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

address of next
instruction to be
executed



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?

Instruction Set Philosophies (Review)

❖ *Complex Instruction Set Computing (CISC):*

Add more and more elaborate and specialized instructions as needed

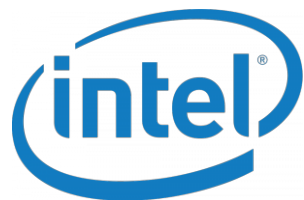
- Lots of tools for programmers to use, but hardware must be able to handle all instructions
- x86-64 is CISC, but only a small subset of instructions encountered with Linux programs

❖ *Reduced Instruction Set Computing (RISC):*

Keep instruction set small and regular

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

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, some Macs
(Core i3, i5, i7, M)
[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)

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
M1 Macs (new!)
[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
(some traction in embedded)
[RISC-V Instruction Set](#)

Architecture Sits at the Hardware Interface

Source code

Different applications
or algorithms

Compiler

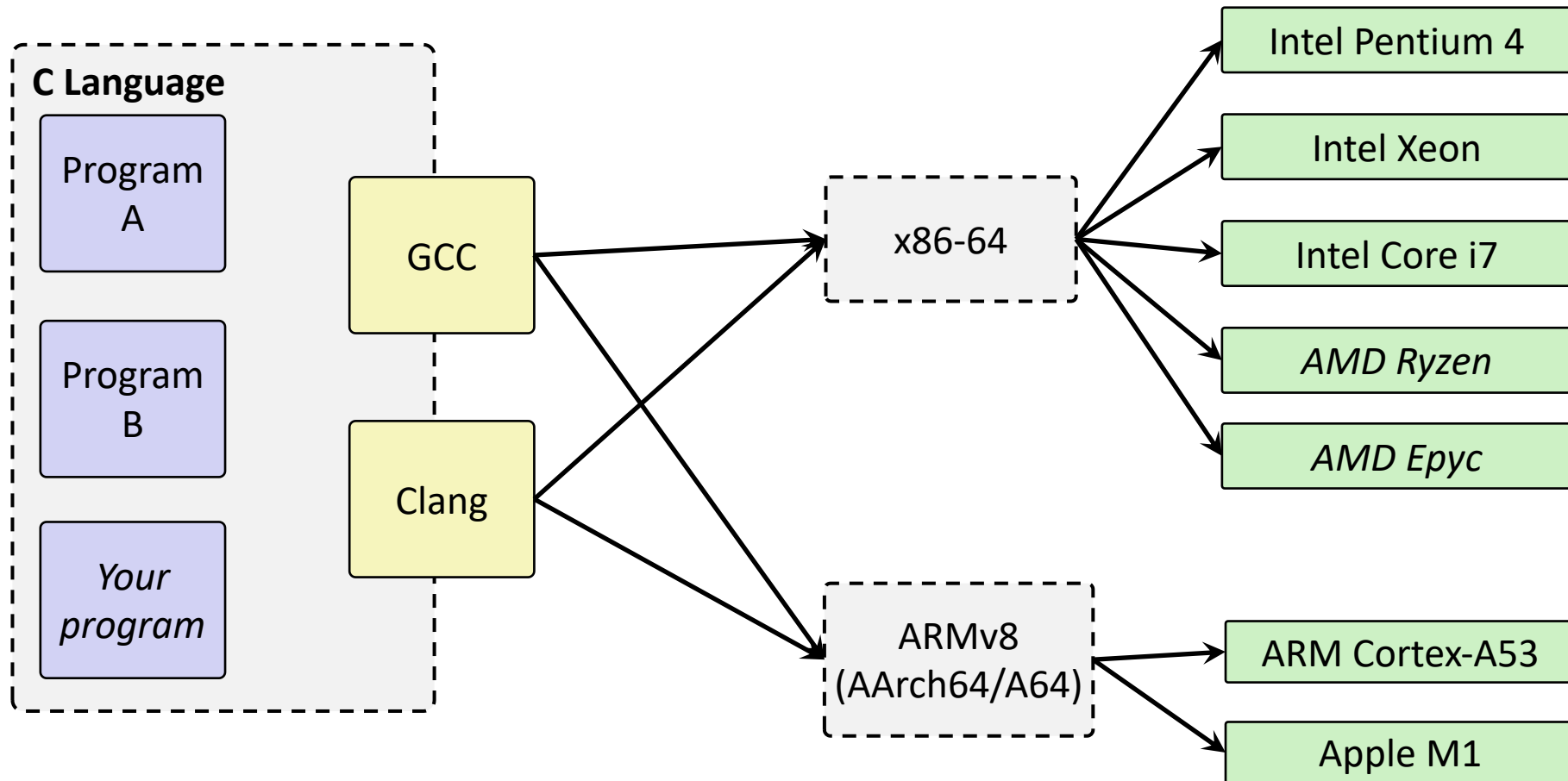
Perform optimizations,
generate instructions

Architecture

Instruction set

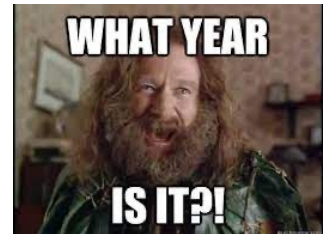
Hardware

Different
implementations

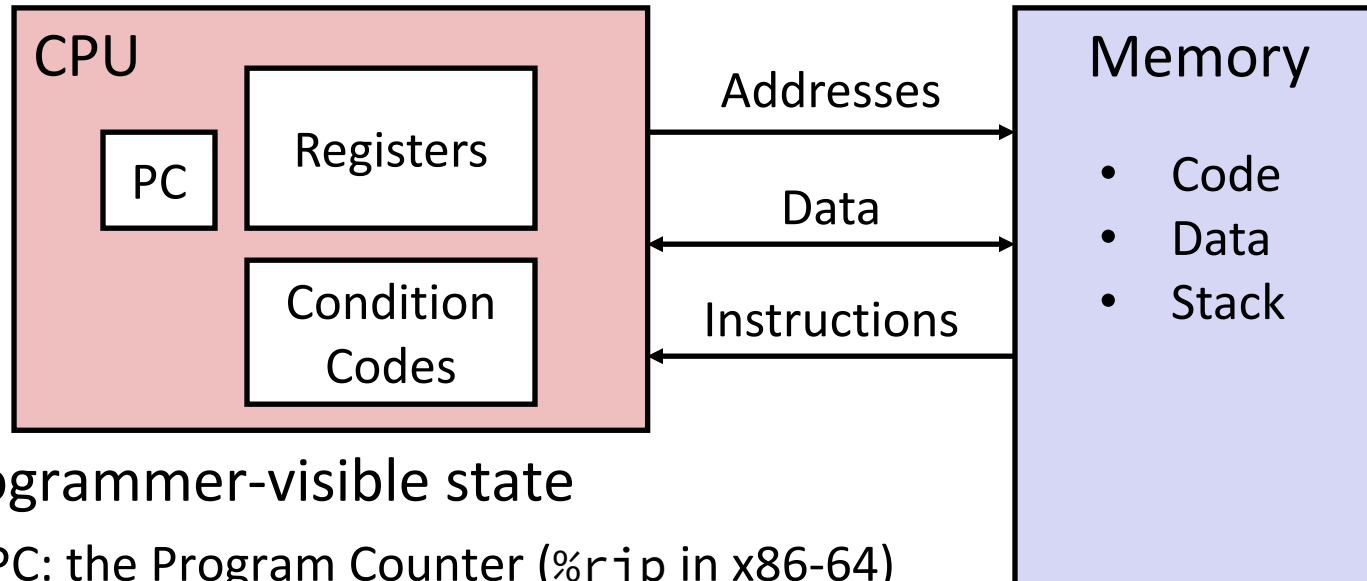


Writing Assembly Code? In \$CURRENT_YEAR???

- ❖ 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, ...)
 - ❖ 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
- } Not covered
In 351

Instruction Types (Review)

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

Instruction Sizes and Operands (Review)

❖ Size specifiers

- b = 1-byte “byte”, w = 2-byte “word”,
l = 4-byte “long word”, q = 8-byte “quad word”
- Note that due to backwards-compatible support for 8086 programs (16-bit machines!), “word” means 16 bits = 2 bytes in x86 instruction names

❖ Operand types

- **Immediate:** Constant integer data (\$)
- **Register:** 1 of 16 general-purpose integer registers (%)
- **Memory:** Consecutive bytes of memory at a computed address (())

What is a Register? (Review)

- ❖ 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* x86-64

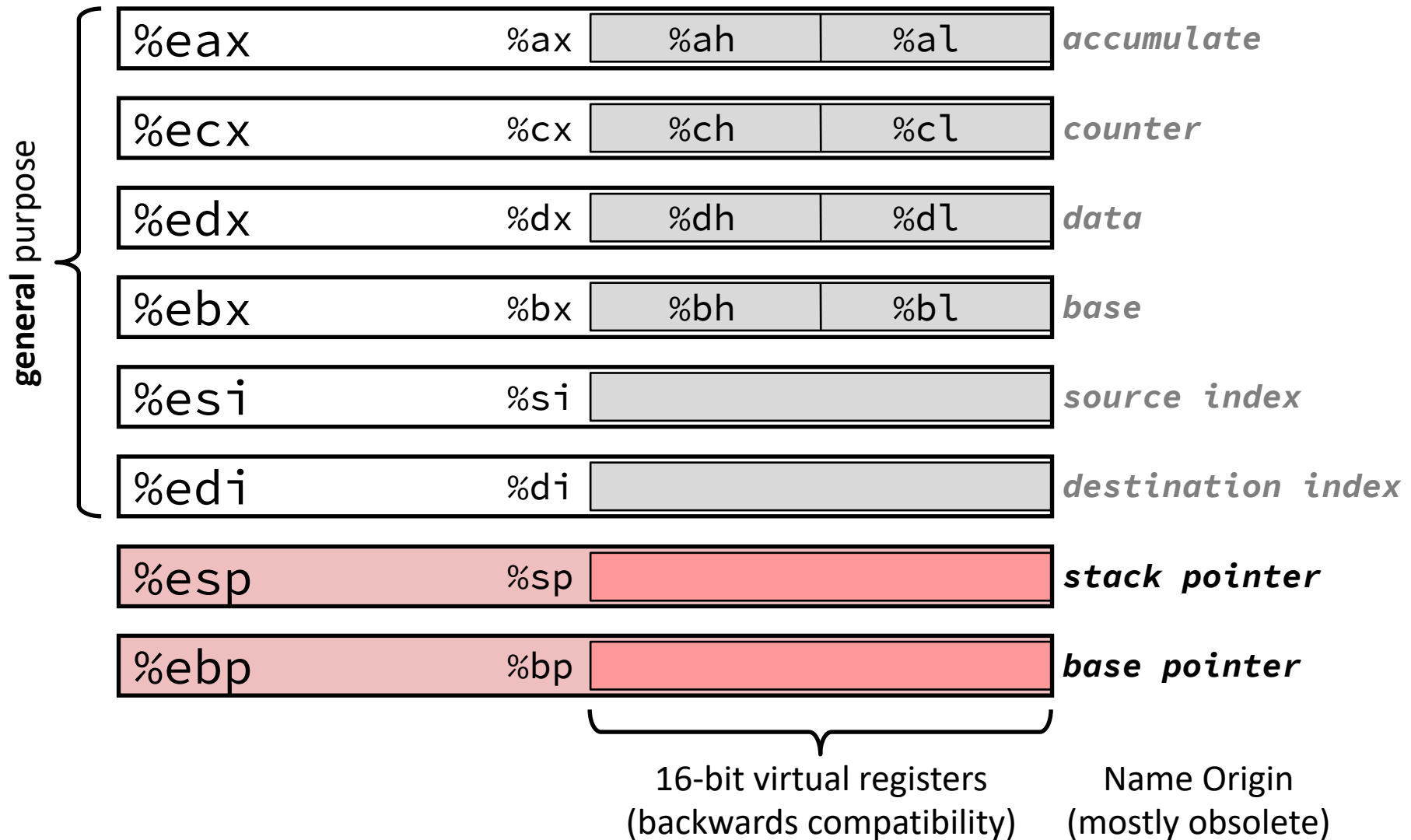
x86-64 Integer Registers – 64 bits wide

%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

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

Some History: IA32 Registers – 32 bits wide



Memory

- ❖ Addresses
 - `0x7FFFD024C3DC`
- ❖ Big
 - $\sim 8 \text{ GiB}$
- ❖ Slow
 - $\sim 50\text{-}100 \text{ ns}$
- ❖ Dynamic
 - Can “grow” as needed while program runs

vs. Registers

- vs. Names
 - `%rdi`
- vs. Small
 - $(16 \times 8 \text{ B}) = 128 \text{ B}$
- vs. Fast
 - sub-nanosecond timescale
- vs. Static
 - fixed number in hardware



Moving Data

- ❖ General form: `mov_ source, destination`
 - Really more of a “copy” than a “move”
 - Like all instructions, missing letter (`_`) is the size specifier
 - Lots of these in typical code

Operand Combinations

	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?

```

movq (%rdi), %rsi    # mem → reg
movq %rsi, (%rdx)    # reg → mem

```

Some Arithmetic Operations

Can't do: $c = a + b$,
need to do: $a += b$ instead

❖ Binary (two-operand) Instructions:


- **Maximum of one memory operand**
- Beware argument order!
- No distinction between signed and unsigned
 - Only arithmetic vs. logical shifts

Format	Computation	
addq <i>src</i> , <i>dst</i>	$dst = dst + src$	($dst += src$)
subq <i>src</i> , <i>dst</i>	$dst = dst - src$	
imulq <i>src</i> , <i>dst</i>	$dst = dst * src$	signed mult
sarq <i>src</i> , <i>dst</i>	$dst = dst \gg src$	Arithmetic
shrq <i>src</i> , <i>dst</i>	$dst = dst \gg src$	Logical
shlq <i>src</i> , <i>dst</i>	$dst = dst \ll src$	(same as <code>salq</code>)
xorq <i>src</i> , <i>dst</i>	$dst = dst \wedge src$	
andq <i>src</i> , <i>dst</i>	$dst = dst \& src$	
orq <i>src</i> , <i>dst</i>	$dst = dst src$	

↑ operand size specifier


Practice Question

- ❖ Which of the following are valid implementations of $rcx = rax + rbx$?

 `addq %rax, %rcx`
`addq %rbx, %rcx`
 $rcx_1 = rax + rcx_0$

$$rcx_2 = rcx_1 + rbx \quad \text{extra term}$$

$$\Rightarrow rcx_2 = rax + rcx_0 + rbx$$


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

$$rcx_0 = 0$$

$$rcx_1 = rbx + rcx_0$$

$$rcx_2 = rax + rcx_1$$


$$\Rightarrow rcx_2 = rax + rbx + 0$$

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

$$rcx_0 = rax$$

$$rcx_1 = rbx + rcx_0$$

$$\Rightarrow rcx_1 = rbx + rax$$

 `xorq %rax, %rax`
`addq %rax, %rcx`
`addq %rbx, %rcx`

$$rax_1 = rax_0 \wedge rax_0 = 0$$

$$rcx_1 = rax_1 + rcx_0$$

$$rcx_2 = rbx + rcx_1$$

$$\Rightarrow rcx_2 = rbx + 0 + rcx_0$$

Arithmetic Example

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

Register	Use(s)
%rdi	1 st argument (x)
%rsi	2 nd argument (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

```
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
```

Compiler Explorer:

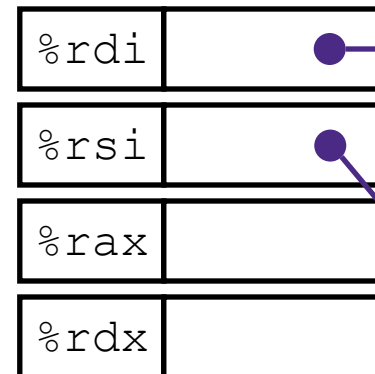
<https://godbolt.org/z/zc4Pcq>

Understanding swap()

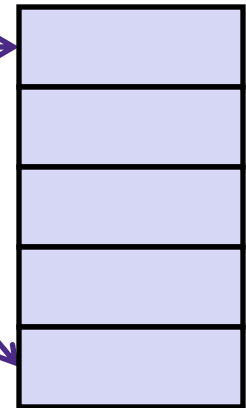
```
void swap(long* xp, long* yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

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

Registers



Memory



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

```


Complete Memory Addressing Modes

❖ General:

- $D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$
 - 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)

- $D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \quad (S=1)$
- $(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S] \quad (D=0)$
- $(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \quad (S=1, D=0)$
- $(, Ri, S) \quad \text{Mem}[\text{Reg}[Ri] * S] \quad (Rb=0, D=0)$

Address Computation Examples

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

$D(Rb, Ri, S) \rightarrow$

$\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$

↑ ignore the memory access for now

Expression	Address Computation	Address (8 bytes wide)
<code>0x8(%rdx)</code>		
<code>(%rdx,%rcx)</code>		
<code>(%rdx,%rcx,4)</code>		
<code>0x80(,%rdx,2)</code>		

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