x86-64 Programming II

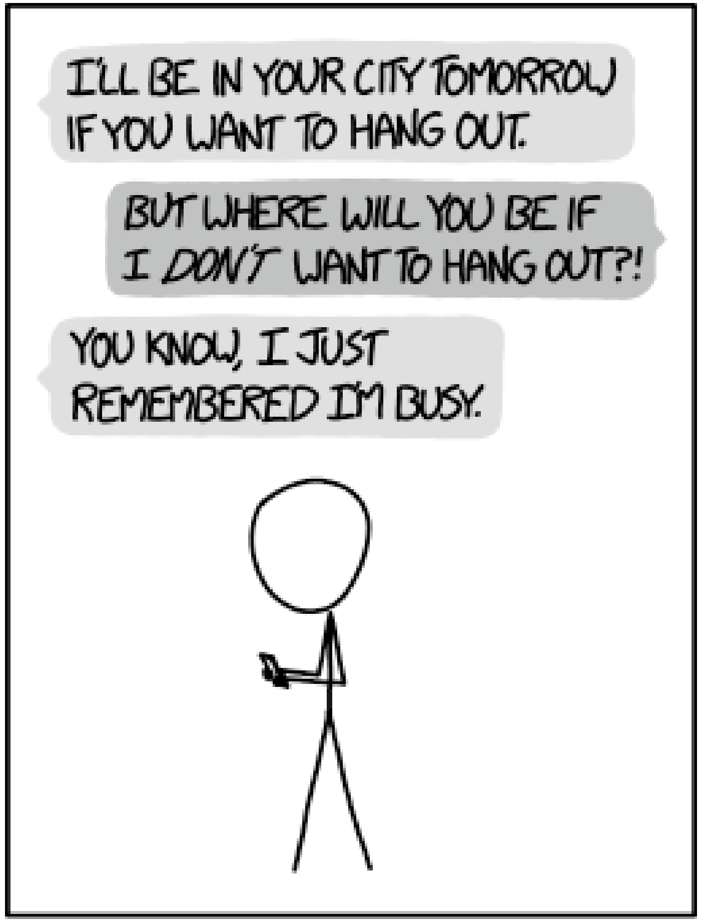
CSE 351 Winter 2024

Instructor:

Justin Hsia

Teaching Assistants:

Adithi Raghavan
Aman Mohammed
Connie Chen
Eyoel Gebre
Jiawei Huang
Malak Zaki
Naama Amiel
Nathan Khuat
Nikolas McNamee
Pedro Amarante
Will Robertson



WHY I TRY NOT TO BE PEDANTIC ABOUT CONDITIONALS.

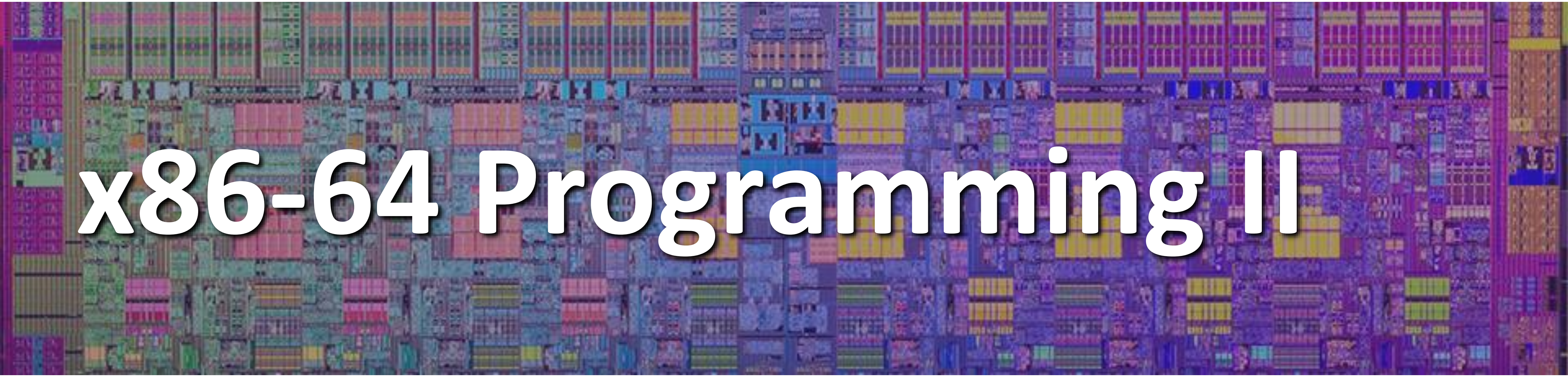
http://xkcd.com/1652/

Relevant Course Information

- Lab submissions that fail the autograder get a **ZERO**
  - No excuses – make full use of tools & Gradescope's interface
  - Leeway on Lab 1a won't be given moving forward

- Lab 2 (x86-64) released Wednesday
  - Learn to trace x86-64 assembly and use GDB

- Midterm is in two weeks (take home, 2/8–10)
  - Open book; make notes and use midterm reference sheet
  - Individual, but discussion allowed via "Gilligan's Island Rule"
  - Mix of "traditional" and design/reflection questions
    - Form study groups and look at past exams!



x86-64 Programming II

Lesson Summary (1/2)

- **Memory Addressing Modes:** Memory operands specify an address in several different forms
  - `D(Rb,Ri,S)` with *base register*, *index register*, *scale factor*, and *displacement* compute the address `Reg[Rb]+Reg[Ri]*S+D` and is usually dereferenced (`Mem[]`) by instructions
    - Defaults when omitted: `Reg[Rb]=0`, `Reg[Ri]=0`, `S=1`, `D=0`
  - These map well to pointer arithmetic operations (S = size of data type)

- **Load effective address** (`lea`) instruction used to compute addresses and perform basic arithmetic
  - *Doesn't* dereference the source memory operand, unlike all other instructions!
  - Useful for computing an address (*e.g.*, `&a[2]`) or basic arithmetic (*e.g.*, `x+4*y+7`)

Lesson Summary (2/2)

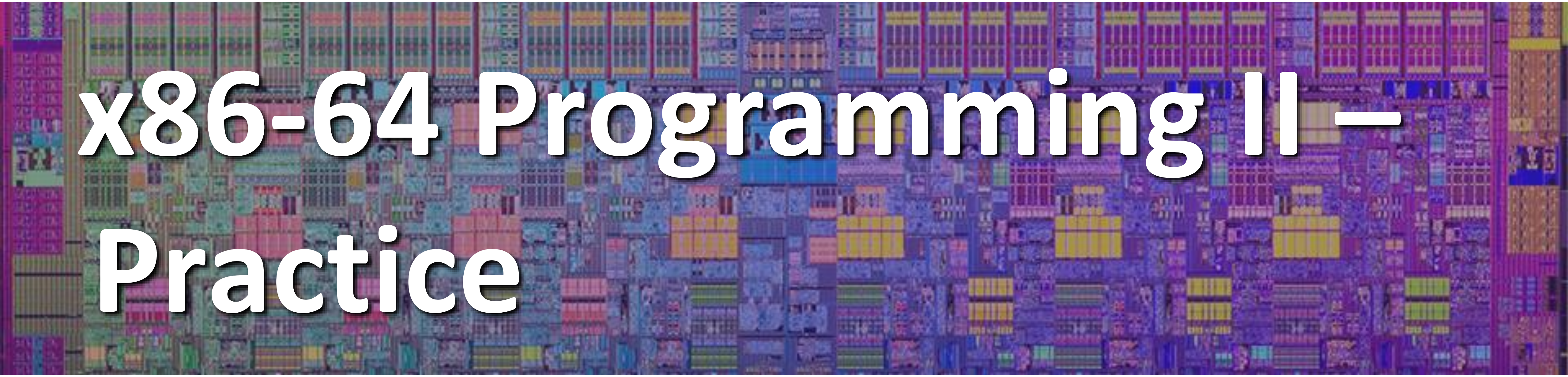
- **Extension instructions** (`movz`, `movs`) allow us to zero and sign extend data into longer widths
  - Require two size suffixes for source (smaller) and destination (larger)

- Control flow in x86 determined by Condition Codes
  - Showed **C**arry, **Z**ero, **S**ign, and **O**verflow, though others exist
  - Set flags with arithmetic & logical instructions (implicit) or Compare and Test (explicit)

Lesson Q&A

- Learning Objectives:
  - Without executing, describe the overall purpose of snippets of x86-64 assembly code containing arithmetic, [if-else statements, and/or loops].
  - Use GDB tools to step through a running program and extract debugging information from a program's disassembly, the state of registers, and values at specific memory locations.

- What lingering questions do you have from the lesson?
  - Chat with your neighbors about the lesson for a few minutes to come up with questions



x86-64 Programming II – Practice

Polling Questions (1/2)

- `D(Rb,Ri,S)` computes address `Reg[Rb]+Reg[Ri]*S+D`
  - Likely will get dereferenced, but that's up to the instruction
  - Default values: `D = 0`, `Reg[Rb] = 0`, `Reg[Ri] = 0`, `S = 1`

- Assuming `%rdx` contains `0xF000` and `%rcx` contains `0x100`, what addresses are computed by the following memory operands?
  - $\overset{D}{\texttt{0x8}}\texttt{(\%}\overset{Rb}{\texttt{rdx}}\texttt{)}$ Reg[Rb]+D = 0xf000 + 0x8   0xf008
  - $\texttt{(\%}\overset{Rb}{\texttt{rdx}}\texttt{,\%}\overset{Ri}{\texttt{rcx}}\texttt{)}$ Reg[Rb]+Reg[Ri]*1   0xf100
  - $\texttt{(\%}\overset{Rb}{\texttt{rdx}}\texttt{,\%}\overset{Ri}{\texttt{rcx}}\texttt{,}\overset{S}{\texttt{4}}\texttt{)}$ *4   0xf400
  - $\overset{D}{\texttt{0x80}}\texttt{(,\%}\overset{Ri}{\texttt{rdx}}\texttt{,}\overset{S}{\texttt{2}}\texttt{)}$ Reg[Ri]*2+0x80   0x1e080

0xf000*2
0xf000 << 1 = 0x1e000

Polling Questions (2/2)

- Which of the following x86-64 instructions correctly calculates `%rax=9*%rdi`?

no memory access, so must be lea

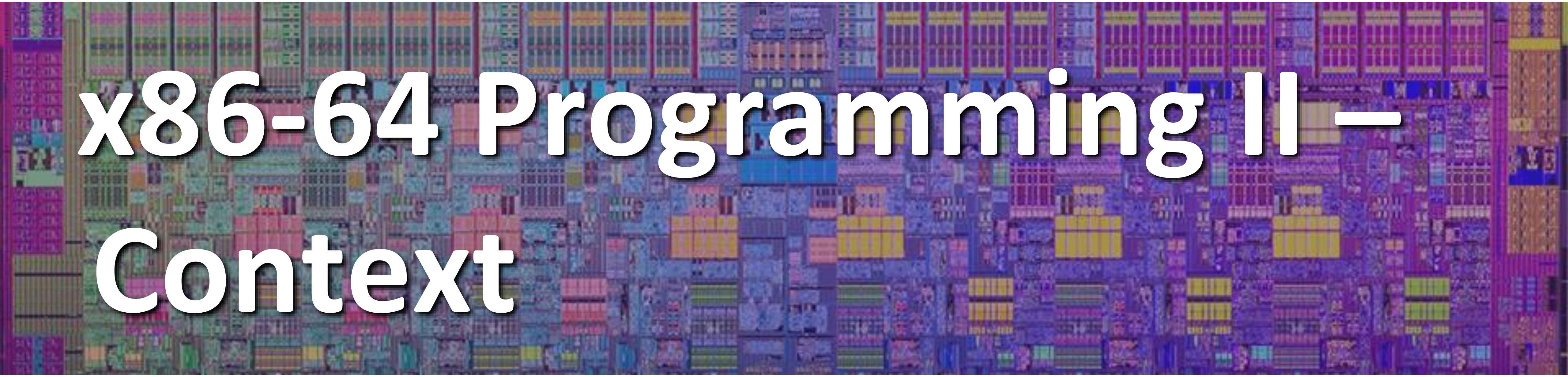
$S \in \{1,2,4,8\}$

A. leaq (,%rdi,~~9~~), %rax — invalid syntax

B. ~~movq~~ (,%rdi,9), %rax — invalid syntax

C. leaq (%rdi,%rdi,8), %rax — %rax = 9*%rdi

D. ~~movq~~ (%rdi,%rdi,8), %rax — %rax = Mem[9*%rdi]



x86-64 Programming II – Context

Extension Instructions (Review)

2 width specifiers: b, w, l, q
1 2 4 8 bytes

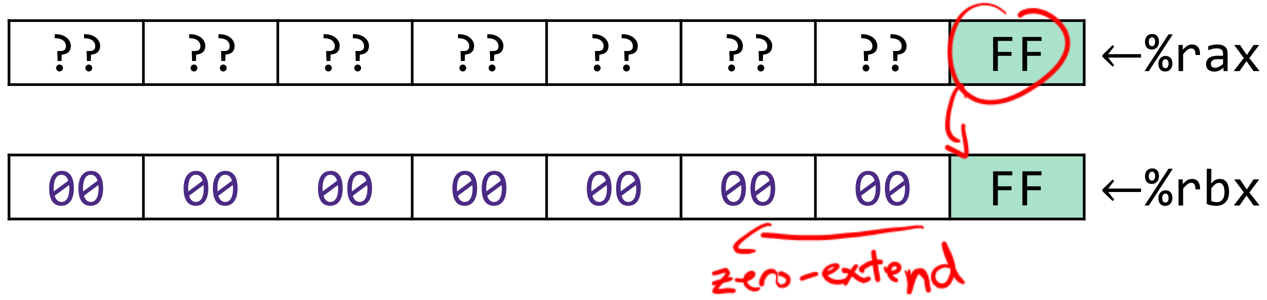
- `movz__ src, dst    # Move with zero extension`
  `movs__ src, dst    # Move with sign extension`
  - Copy from a smaller source value to a larger destination
    - First suffix letter is size of source, second suffix letter is size of destination
    - Recall: zero-extension always fills with `0`, sign-extension fills with copy of the sign bit
  - `src` can be Mem or Reg; `dst` must be Reg

- Example: data shown in hex
  - `movzbq %al, %rbx`

1 byte

zero-extend

8 bytes



??	??	??	??	??	??	??	FF	←%rax
00	00	00	00	00	00	00	FF	←%rbx

zero-extend

Extension Instructions (Review)

- `movz__ src, dst     # Move with zero extension`
  `movs__ src, dst     # Move with sign extension`
  - Copy from a smaller source value to a larger destination
    - First suffix letter is size of source, second suffix letter is size of destination
    - Recall: zero-extension always fills with `0`, sign-extension fills with copy of the sign bit
  - `src` can be Mem or Reg; `dst` must be Reg

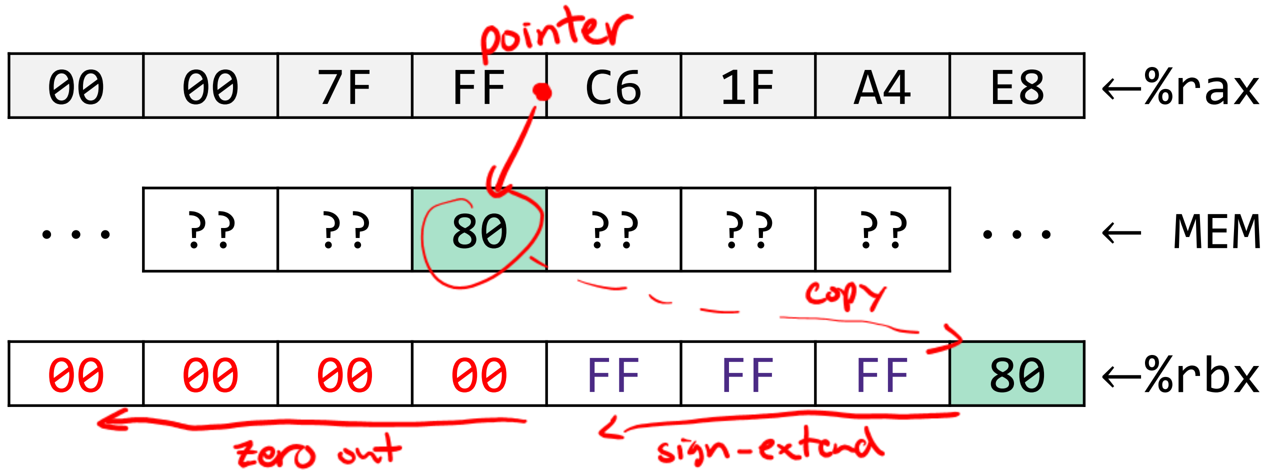
- Example: data shown in hex
  - `movsbl (%rax), %ebx`

4 bytes

sign-extend

1 byte from memory

Recall, *any x86-64 instruction* that stores into a 32-bit (suffix `l`) register zeros out the upper 4 bytes of the register.



00	00	7F	FF	C6	1F	A4	E8		←%rax
. . .	??	??	80	??	??	??	. . .		← MEM
00	00	00	00	FF	FF	FF	80		←%rbx

GDB Demo

- The `movz` and `movs` examples on a real machine!
    - `movzbq %al, %rbx`
    - `movsbl (%rax), %ebx`

- You will need to use GDB to get through Lab 2
    - Useful debugger in this class and beyond!

- Pay attention to:
    - Setting breakpoints (`break`)
    - Stepping through code (`step`/`next` and `stepi`/`nexti`)
    - Printing out expressions (`print` – works with regs & vars)
    - Examining memory (`x`)

Group Work Time

- During this time, you are encouraged to work on the following:
  1) If desired, continue your discussion
  2) Work on the homework problems
  3) Work on the lab (if applicable)

- Resources:
  - You can revisit the lesson material
  - Work together in groups and help each other out
  - Course staff will circle around to provide support