Drew Davidson | University of Kansas

Instruction Set Architectures

Last Time

Lecture Review - Runtimes

Runtimes

- Runtime Environments

  Tradeoff between what's done dynamically vs statically

- Hardware Intuition

  Memory is a big 1D array

You Should Know

- Different runtime environment types
  - Advantages/Disadvantages
- Compiling vs Interpreting

Runtime Environments

This Time

Lecture Outline – ISAs

Instruction-Set Architectures

- What an ISA does
- Our target ISA: x64
- Writing x64

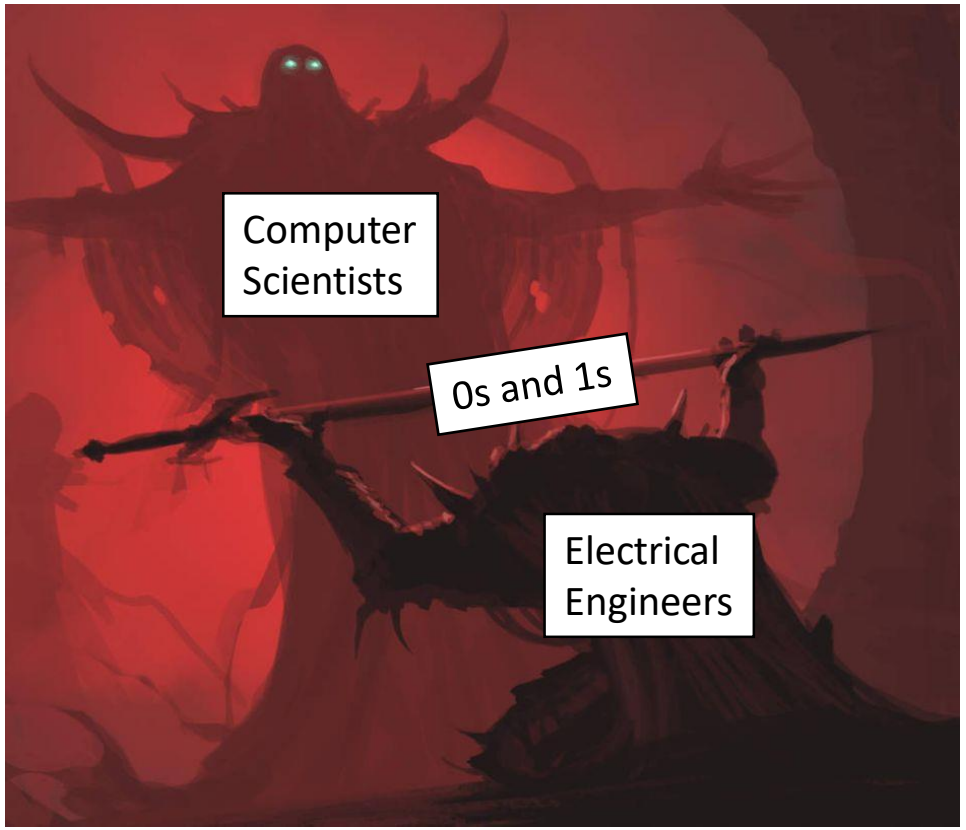
Runtime Environments

Hardware Capabilities

ISAs - Intro

Computers can store binary sequences in memory

- An entire program thus needs to be mapped to binary sequences



Programs as Numeric Sequences

ISAs - Intro

We gotta encode the whole dang program into this sucker!

- Encode data as binary sequences
- Encode instructions as binary sequences

Address 0x0000	Address 0x0001	Address 0x0002	Address 0x0003	Address 0x0004	Address 0x0005	Address 0x0006	Address 0x0007	Address 0x0008	Address 0x0009	Address 0x000A
0x44	0x01	0x02	0x44	0x01	0x03	0x07	0x00	0x00	0x00	0x03

Need to use the same space for many things

The ISA Contract

ISAs - Intro

An ISA specifies

- How data is encoded
- Instructions that can transform data
- Opcodes for how instructions are encoded
- Program state

ISA: A contract of hardware aspects

Instruction Set Architectures

ISAs - Intro

An ISA specifies

- How data is encoded
- Instructions that can transform data
- Opcodes for how instructions are encoded
- Program state

Hypothetical ISA

-2 is encoded as 1110
-1 is encoded as 1111
8 is encoded as 1000
12 is encoded as 1100

The INC_ADDR <X> instruction increments the value at memory address <X>

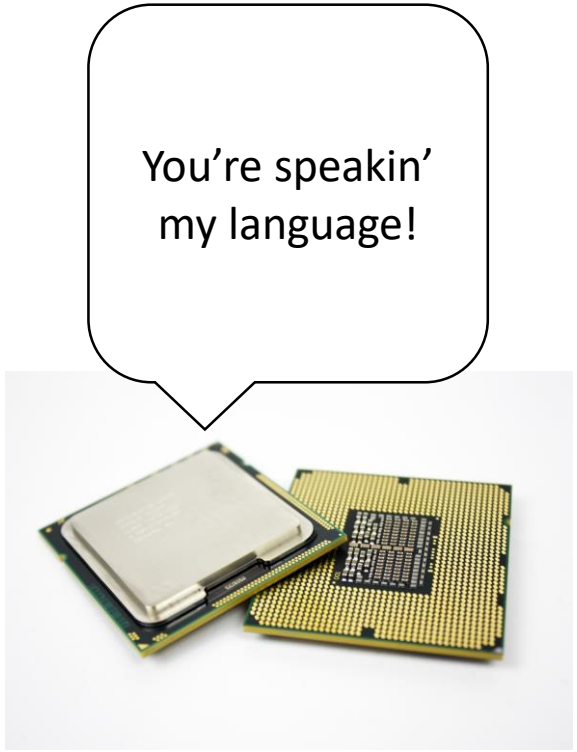
INC_ADDR 8 is encoded as ~~10101010 00010000~~ 1010

Next instruction to execute is stored in register I

Processors Conform to ISAs

ISAs - Intro

- Upon encountering a byte sequence an ISA-conformant “knows” how to interpret the sequence
- Still has some flexibility on how to execute it, specified via the microarchitecture



Completely Hypothetical ISA Example

-2 is encoded as 1110
-1 is encoded as 1111
8 is encoded as 1000
12 is encoded as 1100

The INC_ADDR <X> increments the value at memory address <X>

INC_ADDR 8 encoded as 1010

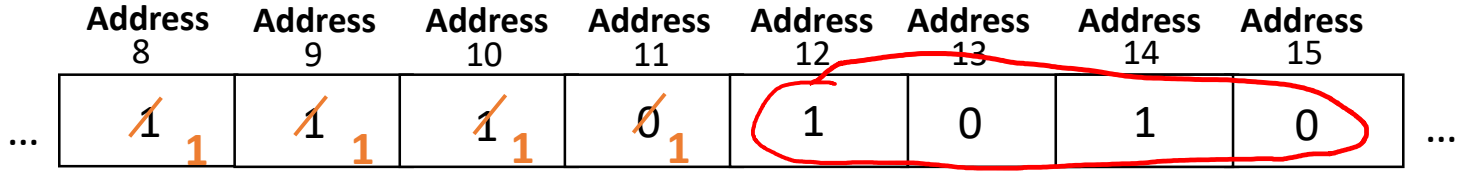
Next instruction to execute stored in register I

Execute the instruction at address 12

The instruction at address 12 is INC_ADDR 8

The value at address 8 is -2

Register I: 1110



More Realistic Encodings

ISAs - Intro

The previous ISA uses unrealistic encodings

- Let's consider some more likely choices

Encoding Data: Granularity of Access

ISAs - Intro

How "big" is a memory cell?

Let's say we're storing the byte 0x61 = 01100001

2^{16} 0xFFFF

Bit-addressable

Address 0x0000	Address 0x0001	Address 0x0002	Address 0x0003	Address 0x0004	Address 0x0005	Address 0x0006	Address 0x0007	Address 0x0008	Address 0x0009	Address 0x000A
0	1	0	0	0	1	0	0	0	0	0

Byte-addressable

$2^{16} \cdot 8$ 0xFFFF

Address 0x0000	Address 0x0001	Address 0x0002	Address 0x0003	Address 0x0004	Address 0x0005	Address 0x0006	Address 0x0007	Address 0x0008	Address 0x0009	Address 0x000A
0x44	0x01	0x02	0x44	0x01	0x03	0x07	0x00	0x00	0x00	0x03

Could even go bigger?

But why (and why not)?

Data Encodings

ISAs - Intro

You should already know the basic idea here

- Type dictates numeric representation
- Devote a certain size (in bits) to representation
- Use binary hardware to store the numeric value

Bit Sequence (binary)

0100 0011 0100 1111 0100 1111 0100 1100

Byte Sequence (Hex)

0x43 0x4F 0x4F 0x4C

ASCII Value: char type (8 bits, i.e. 1 byte)

‘C’ ‘O’ ‘O’ ‘L’

Integer Value: int32 type (32 bits, i.e. 4 bytes)

0x434F4F41

1,129,271,105

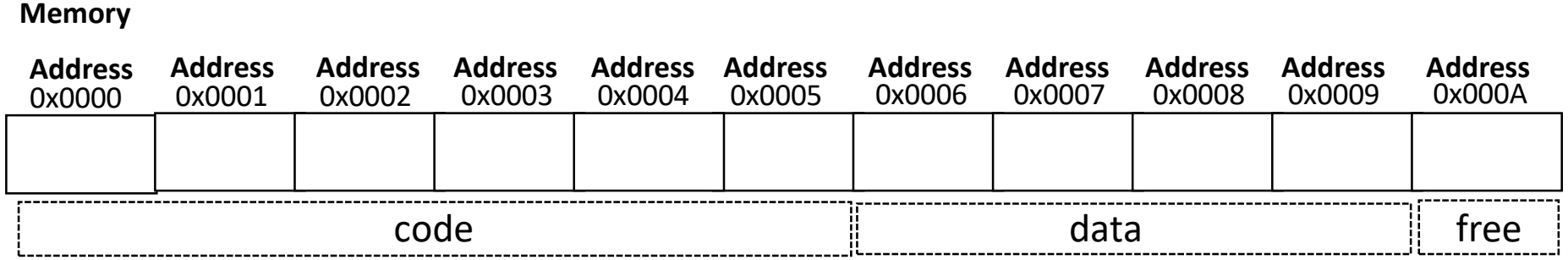
Convention: Memory Regions

ISAs - Intro

Portions of memory "zoned" by purpose

Simplest form:

- Code region
- Data region
- The rest is free space

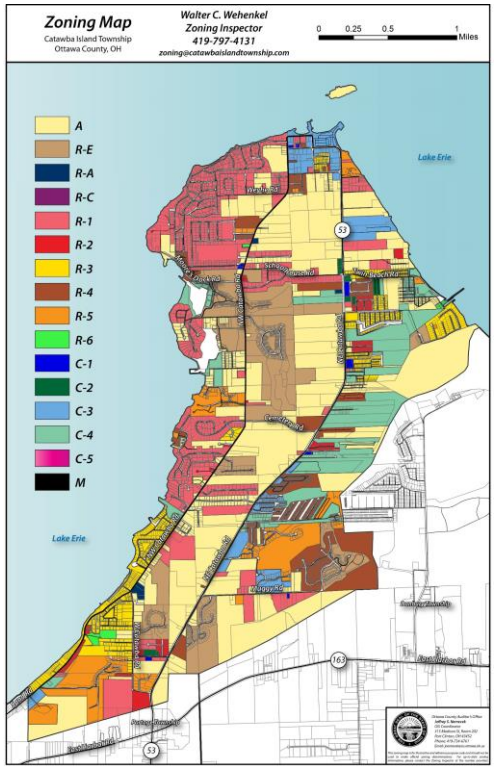


Data Sub-Regions

ISAs - Intro

Further break up data region for different ***kinds*** of data

- Global variables
- Local variables
- Objects



This Time

Lecture Outline – ISAs

Instruction-Set Architectures

- What an ISA does
- Our target ISA: x64
- Writing x64

Runtime Environments

Our ISA: x64

Intro to Assembly: About

- Probably the most popular architecture in modern use
- Almost certainly what your computer is running
- Definitely what the cycle servers are running

x86 and x64: A Reputation for Difficulty

Intro to Assembly: About

Highly complex instruction set

- ~1000 different instructions via the most conservative count*
- Some instructions context-sensitive (i.e. work differently based on preceding instructions)

*that we don't have a canonical instruction count is already a pretty bad sign

Why is it called x64?

Intro to x64: About

Short for x86-64

- The 64-bit extension of the x86 instruction set

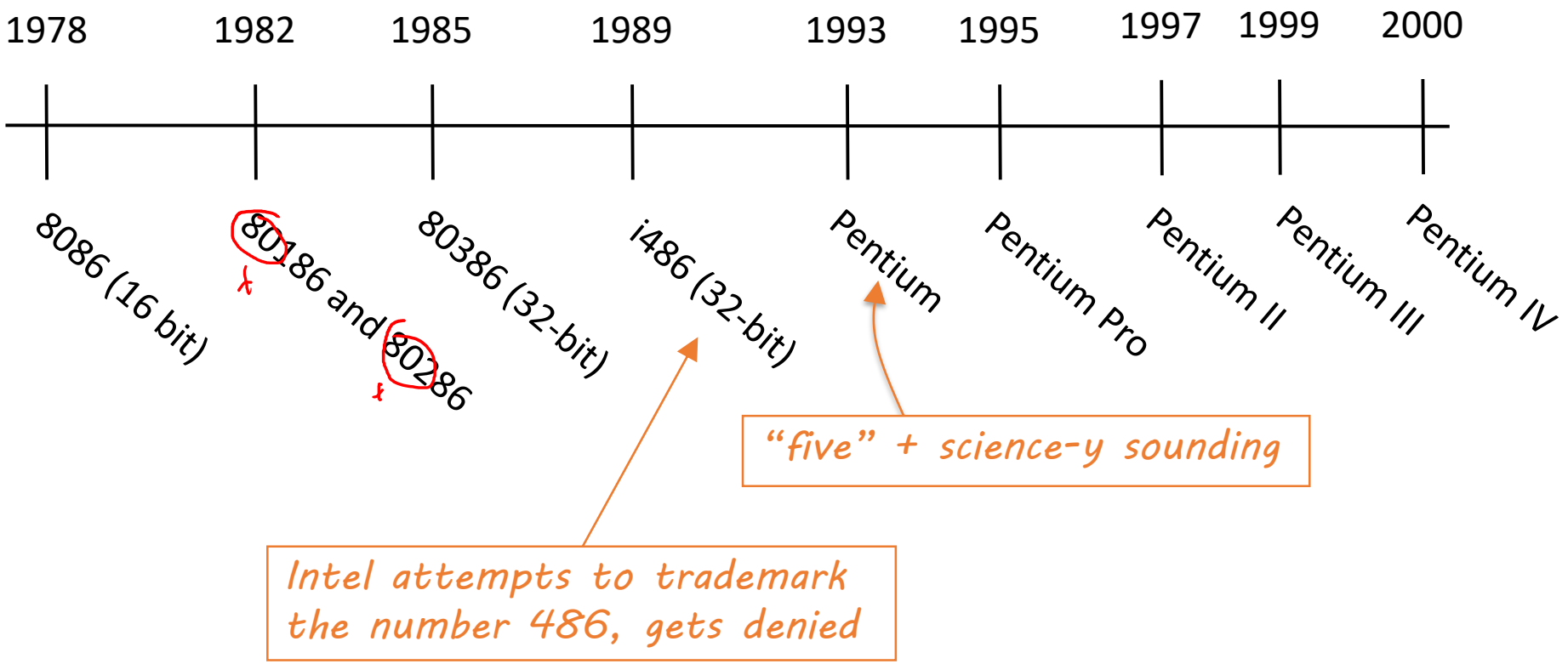
So then what is x86?

- A dang mess!

Some Architecture History: x86

Intro to x64: About

Detour



Itanium: A Fresh Start?

Intro to x64: About

2001: Itanium released by Intel

- non-backward-compatible 64-bit architecture
- A new architecture philosophy heavily reliant on compiler optimization
- Ditches much of the messiness of x86

Highly anticipated

- Development of MIPS and DEC Alpha halted in anticipation

Itanium: How Did it Go?

Intro to x64: About

“one of the greatest fiascos of the last 50 years”

- John C. Dvorak

“a joke in the chip industry”

- Ashlee Vance

“it turned out the wished-for compilers were basically impossible to write”

- Donald Knuth

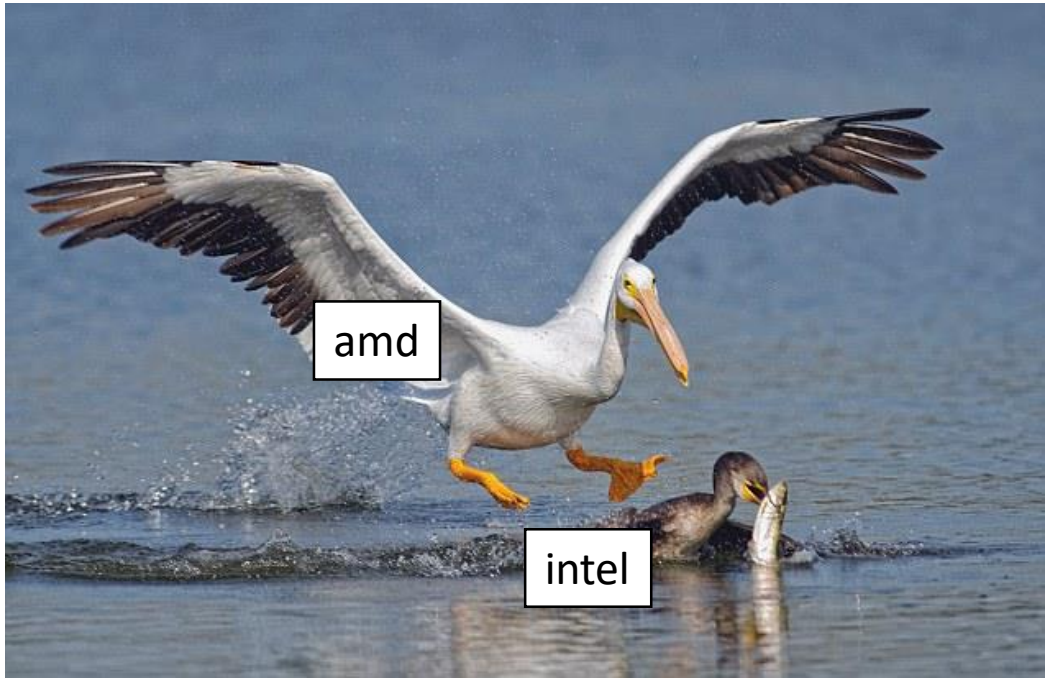
AMD Swoops In

Intro to x64: Background

2003: AMD64

- AMD's backwards-compatible 64-bit x86 extension
- Intel eventually catches up with compatible Intel 64 architecture

X64 used to denote the vendor-neutral intersection of AMD64 and Intel 64



The Upshot: A Patchwork

Intro to x64: Background

- x86 carries many features that lack modern salience
- x64 avoids a lot of these
- Still has some WTF features
  - Many to do with register-poorness

x64 Registers

Intro to Assembly: About

Name	Number	Nominal Purpose
rax	0	Computation Accumulator
rbx	1	Computation Base
rcx	2	Computation counter
rdx	3	Data for I/O
rsi	4	String source address
rdi	5	String destination address
rbp	6	Base pointer (base of the stack)
rsp	7	Stack pointer (edge of the stack)
r08 – r15	8 - 15	True general purpose registers
rip	-	Instruction pointer
rflags	-	Status flags

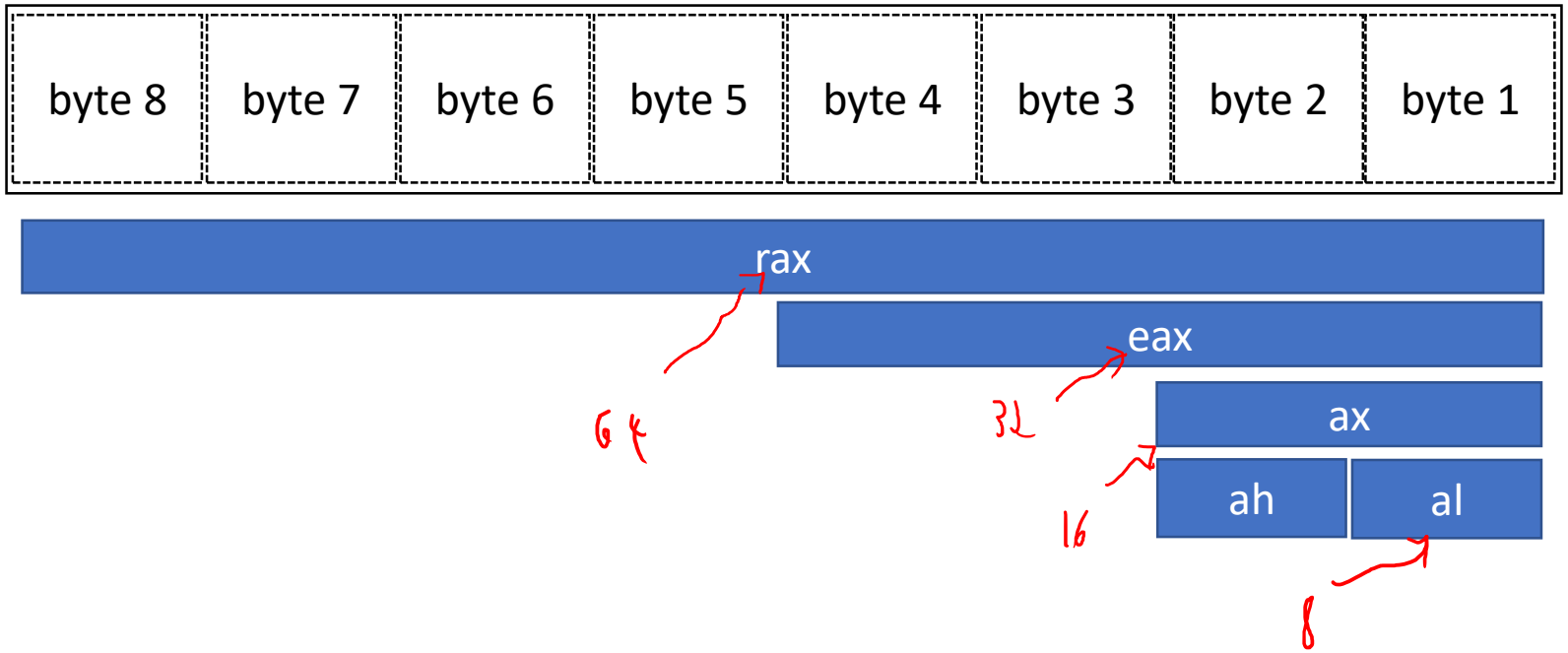
Can be used in Instruction opcodes

Cannot be used in instruction opcodes

x64 Register Compatibility

Intro to Assembly: About

Register #0 – the "A" register



This Time

Lecture Outline – ISAs

Instruction-Set Architectures

- What an ISA does
- Our target ISA: x64
- Writing x64

Runtime Environments

Stepping Back From Binary

Encoding Programs

Dealing with binary directly is tedious and error-prone

- Laying out code / data is super difficult to do manually
- Remembering the binary opcode sequence for every instruction is difficult

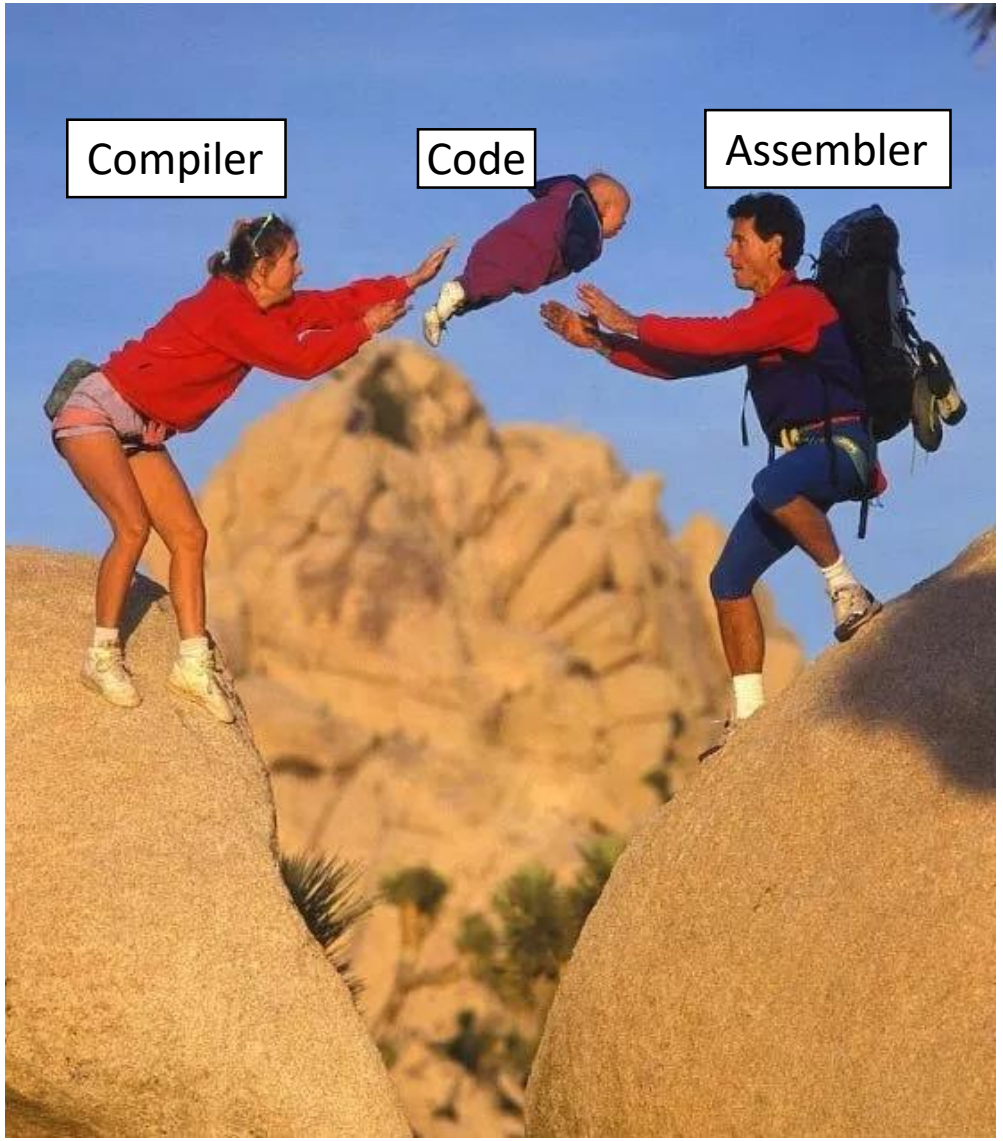
Fortunately, we don't have to do that

The Assembler

Intro to Assembly: About

Write low-level textual *mnemonics* (assembly code)

- Assembly code isn't *directly* executable
- Nearly 1-1 with the binary encoding
- Different assemblers, different syntax



ASM Instruction Syntax

x64 syntax

As with everything x86-related, it's complicated

- We'll use the AT&T Syntax:

  <opcode><sizesuffix> <src operand(s)> <dst operand>

- Immediates (i.e. constant values) prefixed by $
- Registers prefixed by %
- Memory lookup (i.e. dereference) in parens

```
movq $5, (%rax)
```

mov the 64-bit value 5 into the 64-bit memory address specified by register rax

Directives

x64 syntax

- Indicates a command to the assembler
  - Layout, program entrypoint, etc.

Example:

.globl X

Indicates that symbol X is visible outside of the file

Segment Directives

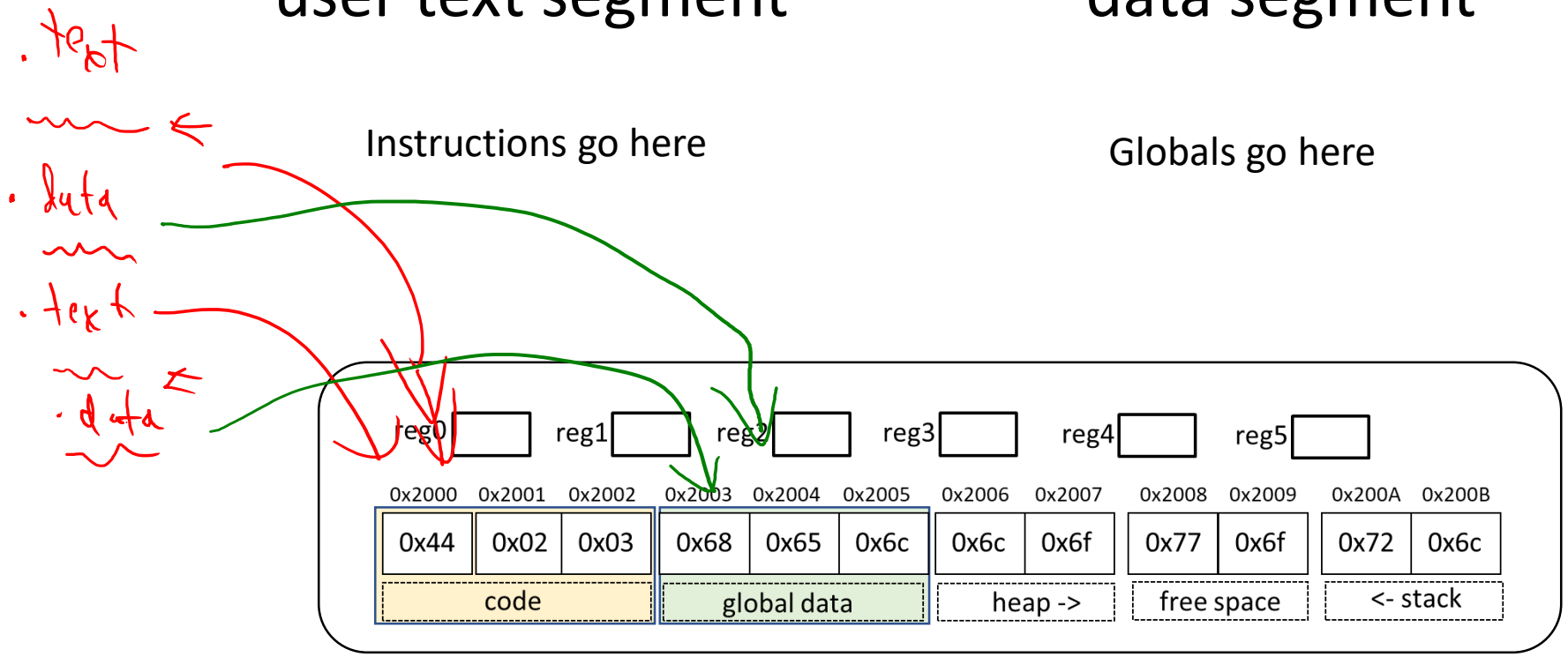
Intro to x64: Writing Assembly

`.text`

Lay out items in the user text segment

`.data`

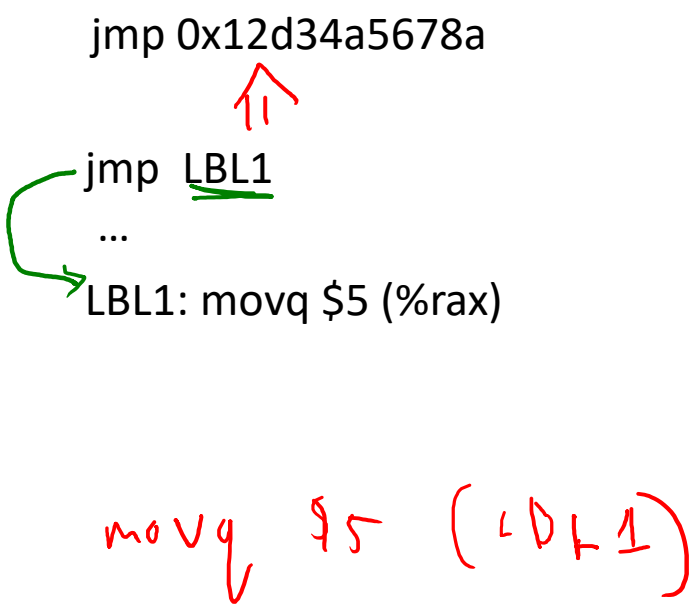
Lay out items in the data segment



Labels

x64 syntax

- The assembler allows us to specify “placeholder” addresses that will be used later
  - Translated to “real” addresses by a utility called the linker
- Valid for both data and code locations



System calls

Intro to x64: A program

To interact outside program memory, need the help of the OS

```
syscall
```

```
%rax      # Which system call (60 is exit)
%rdi      # Set syscall arg – (exit takes the return code)
```

Time to put it all together!

Intro to X64

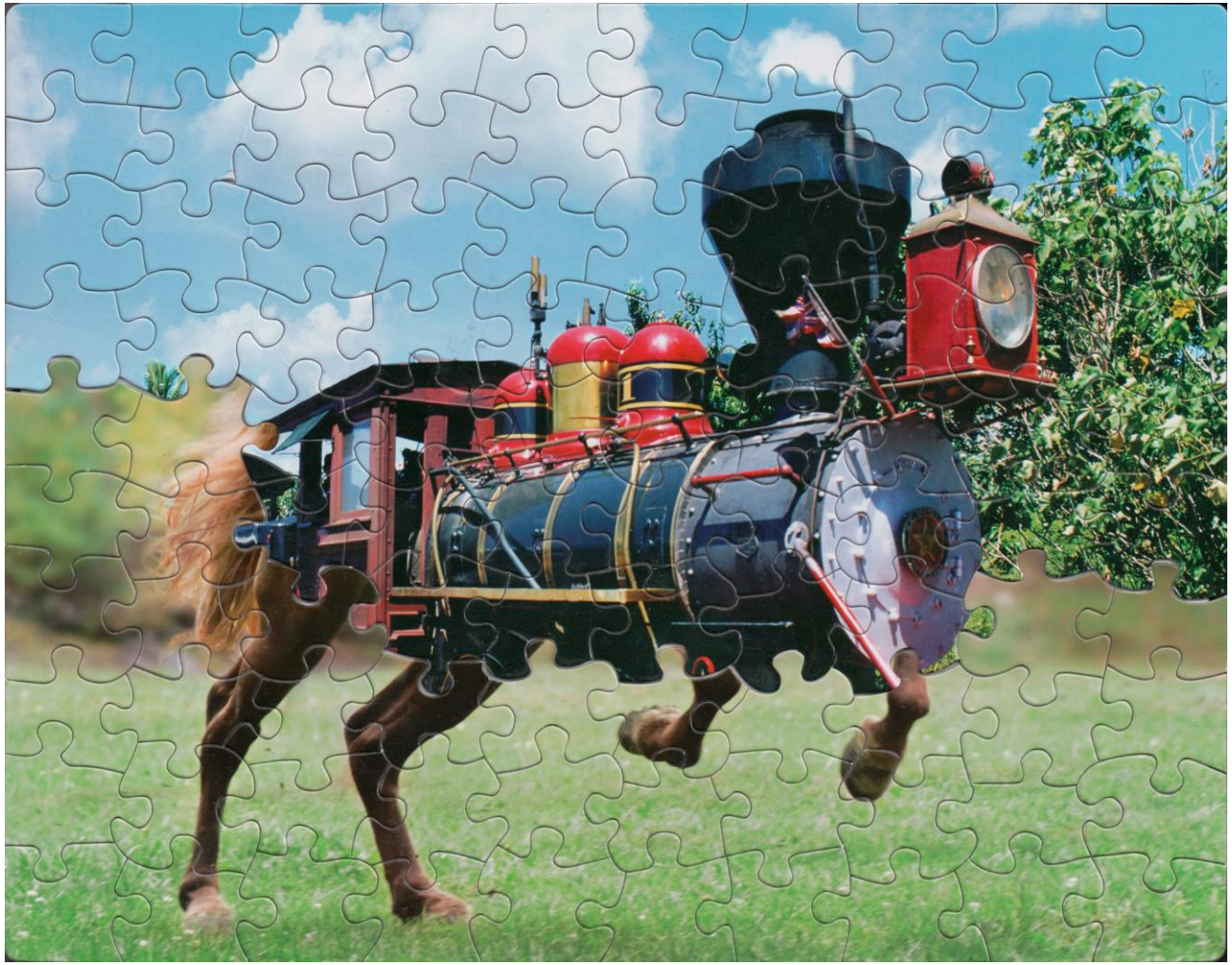


Photo Credit: Tim Klein - https://puzzlemontage.crevado.com

A Complete Program

Intro to x64: A program

```
.text
.globl _start
_start:
    movq $60, %rax      # Choose syscall exit
    movq $4, %rdi       # Set syscall arg - return code
    syscall
```

Actually Running a Program

Intro to x64: A program

Invoking the assembler and linker