This Time

Lecture Outline – ISAs

Write 3AC code for the following program

```
int v(int a, int b){
    if (a > 1){
        if (b < 3){
            return a + b;
        }
    }
}
```

Drew Davidson | University of Kansas

EECS 665 COMPILER CONSTRUCTION

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

Runtime Environments

<u>You Should Know</u>

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

This Time

Lecture Outline – ISAs

Instruction-Set Architectures

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

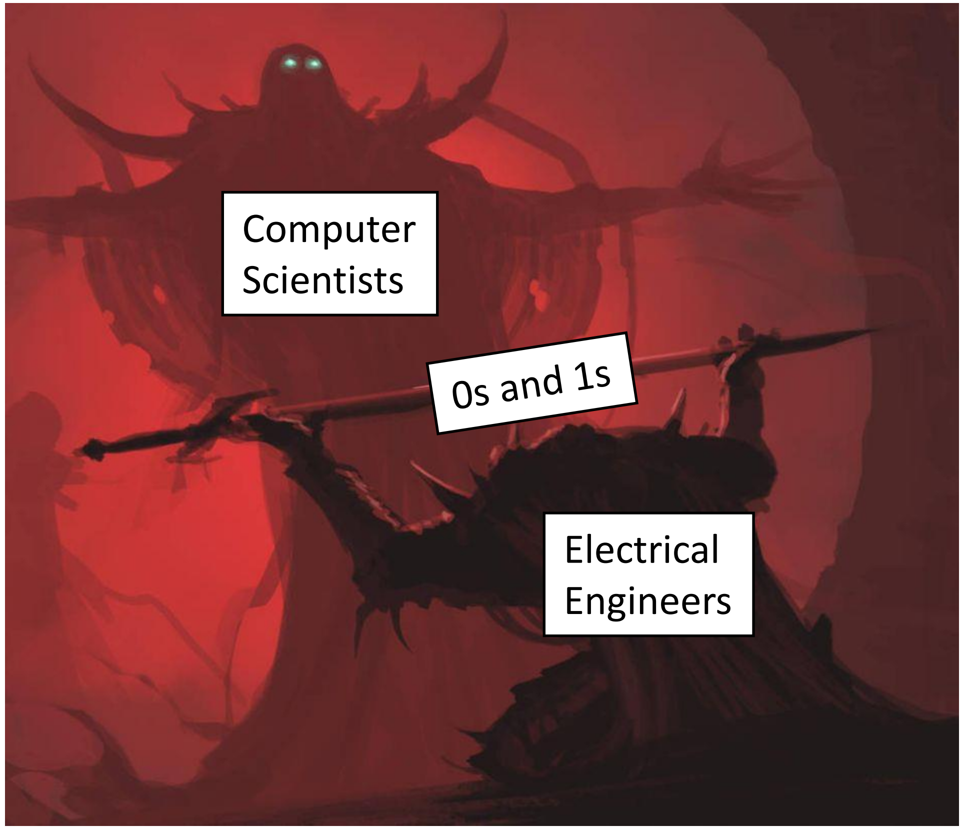
Architecture

Hardware Capabilities

ISAs - Intro

Computers can store binary sequences in memory

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



W.Y.S.I.N.W.Y.X

What You See (in source code) Is Not What You eXecute

- Many of our abstractions lack explicit representation in machine code

WYSINWYX: What You See Is Not What You eXecute

G. Balakrishnan[1], T. Reps[1,2], D. Melski[2], and T. Teitelbaum[2]

[1] Comp. Sci. Dept., University of Wisconsin; {bgogul,reps}@cs.wisc.edu
[2] GrammaTech, Inc.; {melski,tt}@grammatech.com

Abstract. What You See Is Not What You eXecute: computers do not execute source-code programs; they execute machine-code programs that are generated from source code. Not only can the WYSINWYX phenomenon create a mismatch between what a programmer intends and what is actually executed by the processor, it can cause analyses that are performed on source code to fail to detect certain bugs and vulnerabilities. This issue arises regardless of whether one's favorite approach to assuring that programs behave as desired is based on theorem proving, model checking, or abstract interpretation.

1 Introduction

Recent research in programming languages, software engineering, and computer security has led to new kinds of tools for analyzing code for bugs and security vulnerabilities [23, 41, 18, 12, 8, 4, 9, 25, 15]. In these tools, static analysis is used to determine a conservative answer to the question "Can the program reach a bad state?"[3] However, these tools all focus on analyzing *source code* written in a high-level language, which has certain drawbacks. In particular, there can be a mismatch between what a programmer intends and what is actually executed by the processor. Consequently, analyses that are performed on source code can fail to detect certain bugs and vulnerabilities due to the WYSINWYX phenomenon: "What You See Is Not What You eXecute". The following source-code fragment, taken from a login program, illustrates the issue [27]:

```
memset(password, '\0', len);
free(password);
```

The login program temporarily stores the user's password—in clear text—in a dynamically allocated buffer pointed to by the pointer variable `password`. To minimize the lifetime of the password, which is sensitive information, the code fragment shown above zeroes-out the buffer pointed to by `password` before returning it to the heap. Unfortunately, a compiler that performs useless-code elimination may reason that the program never uses the values written by the call on `memset`, and therefore the call on memset can be removed—thereby leaving sensitive information exposed in the heap. This is not just hypothetical; a similar vulnerability was discovered during the Windows security push in 2002 [27]. This vulnerability is invisible in the source code; it can only be detected by examining the low-level code emitted by the optimizing compiler.

The WYSINWYX phenomenon is not restricted to the presence or absence of procedure calls; on the contrary, it is pervasive:

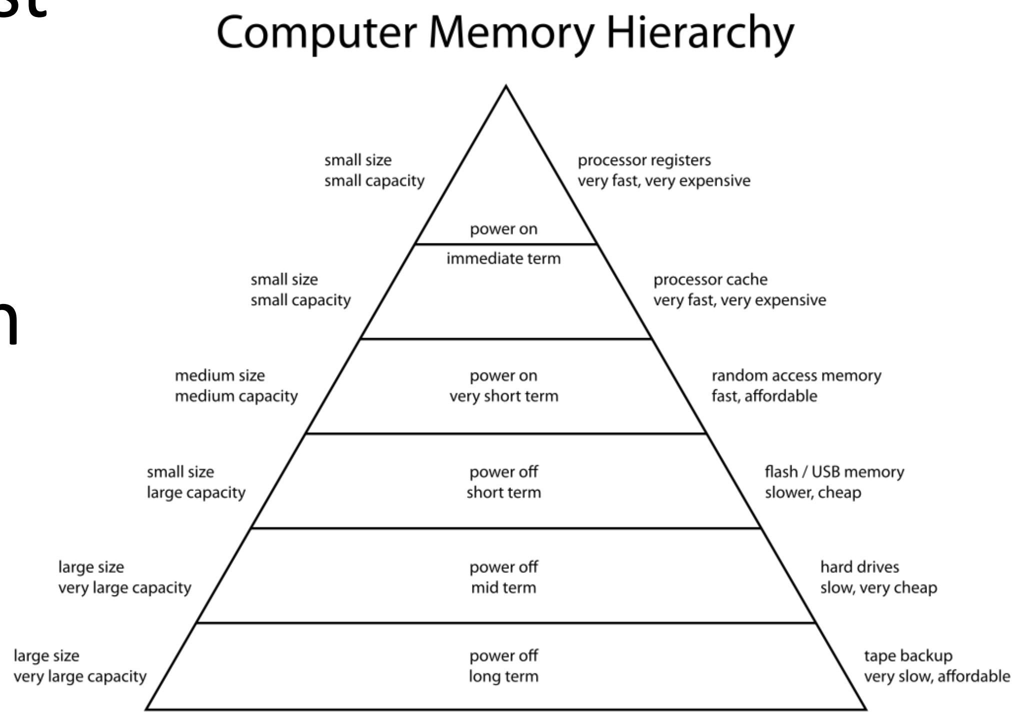
– Bugs and security vulnerabilities can exist because of a myriad of platform-specific details due to features (and idiosyncrasies) of compilers and optimizers, including

[3] Static analysis provides a way to obtain information about the possible states that a program reaches during execution, but without actually running the program on specific inputs. Static-analysis techniques explore the program's behavior for *all* possible inputs and *all* possible states that the program can reach. To make this feasible, the program is "run in the aggregate"—i.e., on descriptors that represent *collections* of memory configurations [13].

Hardware Generally Has…

ISAs - Introduction

- Limited number of very fast registers with which to do computation
- Comparatively large region of memory to hold data
- Some basic instructions from which to build more complex behaviors



Missing Abstractions of Machine Code

- Loops
- Expressions
- Variables
- Scope
- Functions

- Opcodes
- Registers
- Byte-addressable memory

Programs as Numeric Sequences

ISAs - Introduction

We gotta encode the whole dang program into a 1D-array!

- 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

Memory: Intuition

ISAs – Hardware Features

- Cells have a (numeric) address and hold (numeric) value
- We can think of program memory as a big ol' 1D-array

Address 0x0000	Address 0x0001	Address 0x0002	Address 0x0003	Address 0x0004	Address 0x0005	Address 0x0006
0x4	0x4	0x2	0x6	0x2	0x3	0x5
	a				*b*	

- Data access is like indexing into that array

```
[a]  :=  [b]
```

memory[a]   *memory[b]*

assume a takes up address 0x0001
assume b takes up address 0x0005

Address 0x0000	Address 0x0001	Address 0x0002	Address 0x0003	Address 0x0004	Address 0x0005	Address 0x0006
0x4	0x3	0x2	0x6	0x2	0x3	0x5
	a				*b*	

Registers: Intuition

- Specialized, super-fast circuitry
- Computation must be done on registers

Memory: "data at rest"
Registers: "data in flight"

3AC code

```
[a] := [b] + [c]
```

Corresponding Hardware tasks

Get operand 1 into a register

Get operand 2 into another register

Sum the two registers to a destination register

Store destination register back to memory

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

This Time

Lecture Outline – ISA Hardware Features

Instruction-Set Architectures

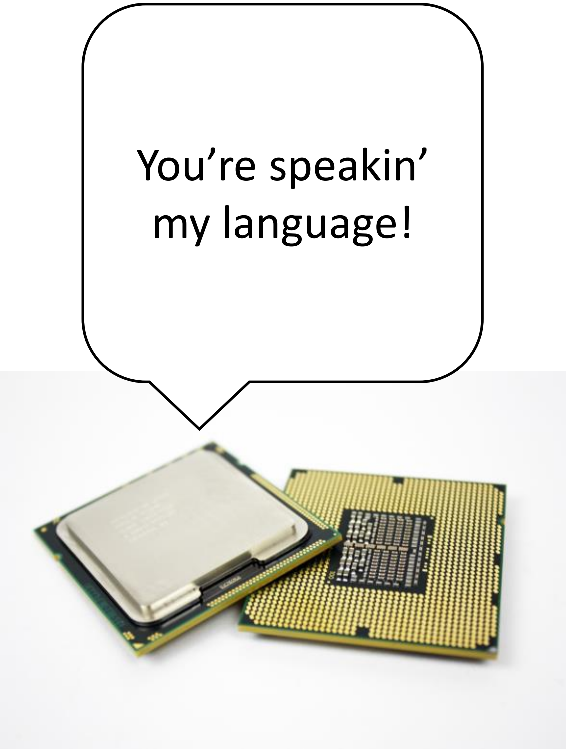
- Introduction
- What an ISA does
- Our target ISA: x64
- Writing x64

Architecture

Processors Conform to ISAs

ISAs – Hardware Features

- 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



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 1010

Next instruction to execute is stored in register I

Completely Hypothetical ISA Example

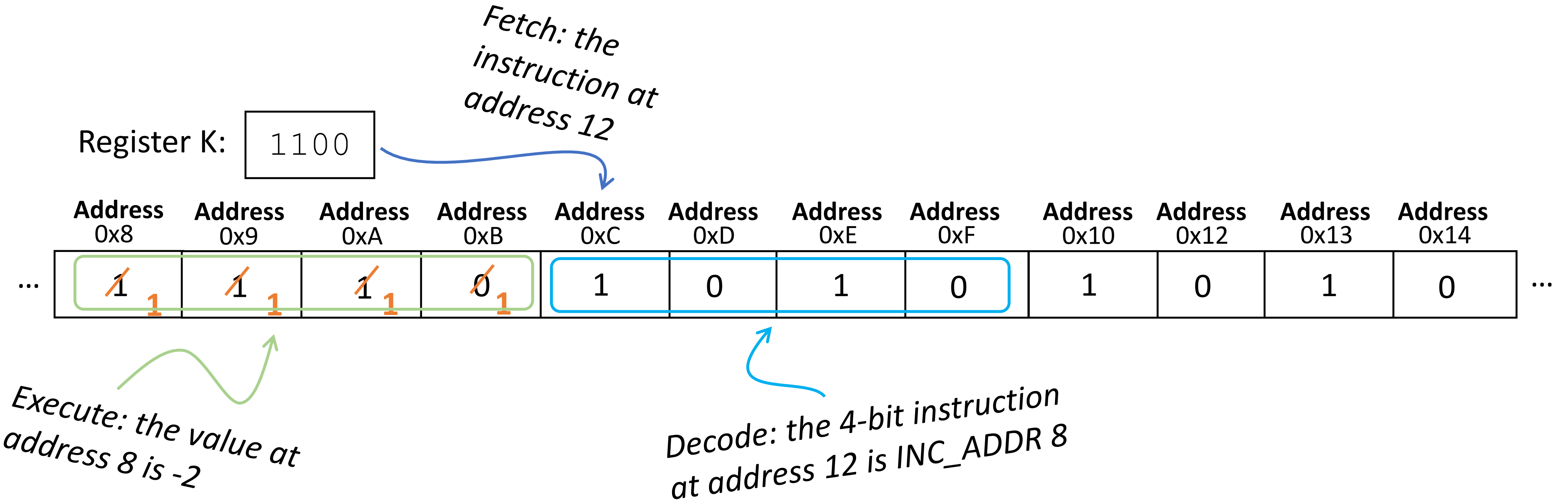
ISAs - Intro

-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 K



More Realistic Encodings
ISAs - Intro

The previous ISA uses unrealistic encodings

- Let's consider some more likely choices

Encoding Data: Granularity of Access

How “big” is a memory cell?

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

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

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

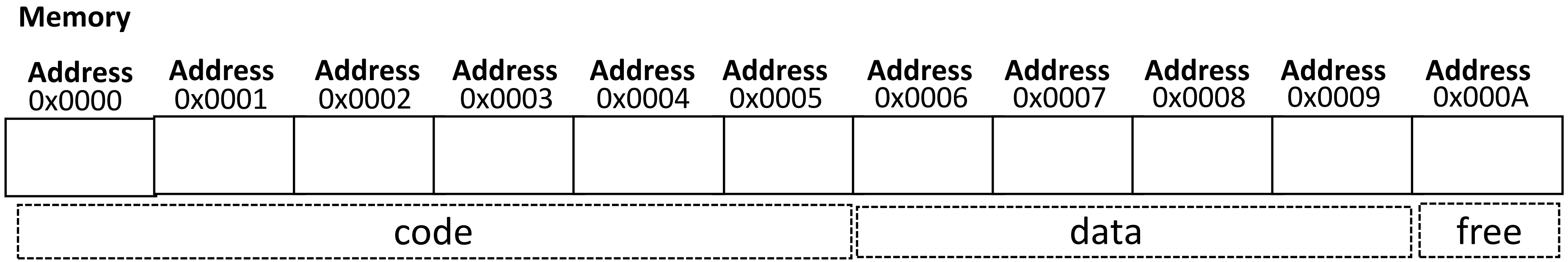
1,129,271,105

Convention: Memory Regions

Portions of memory "zoned" by purpose

Simplest form:

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

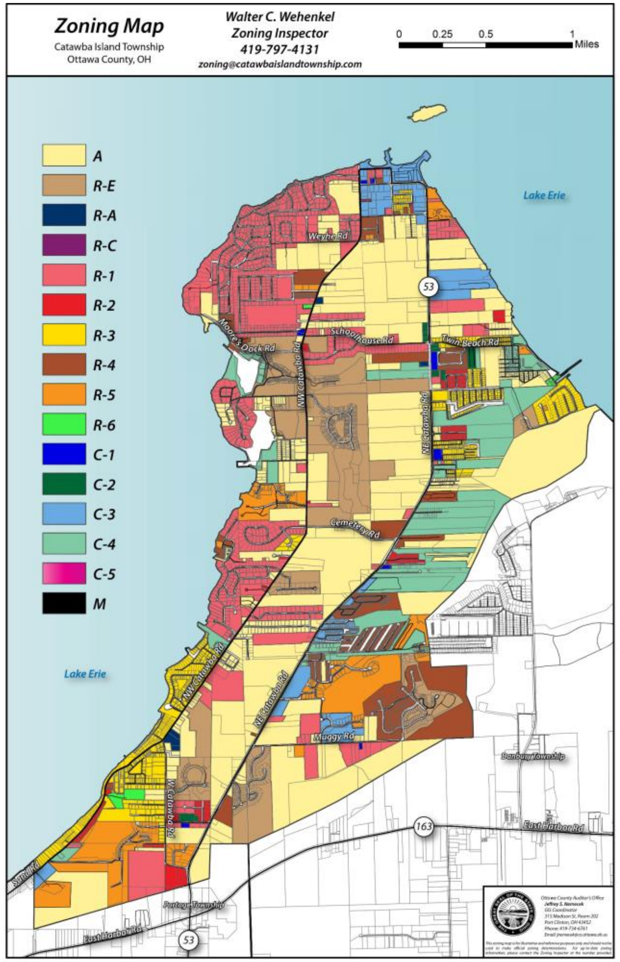


Data Sub-Regions

ISAs - Intro

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

- Global variables
- Local variables
- Objects



This Time

Lecture Outline – About x64

Instruction-Set Architectures

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

Our ISA: x64

- 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

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

x64 Registers

Lecture Outline – About x64

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

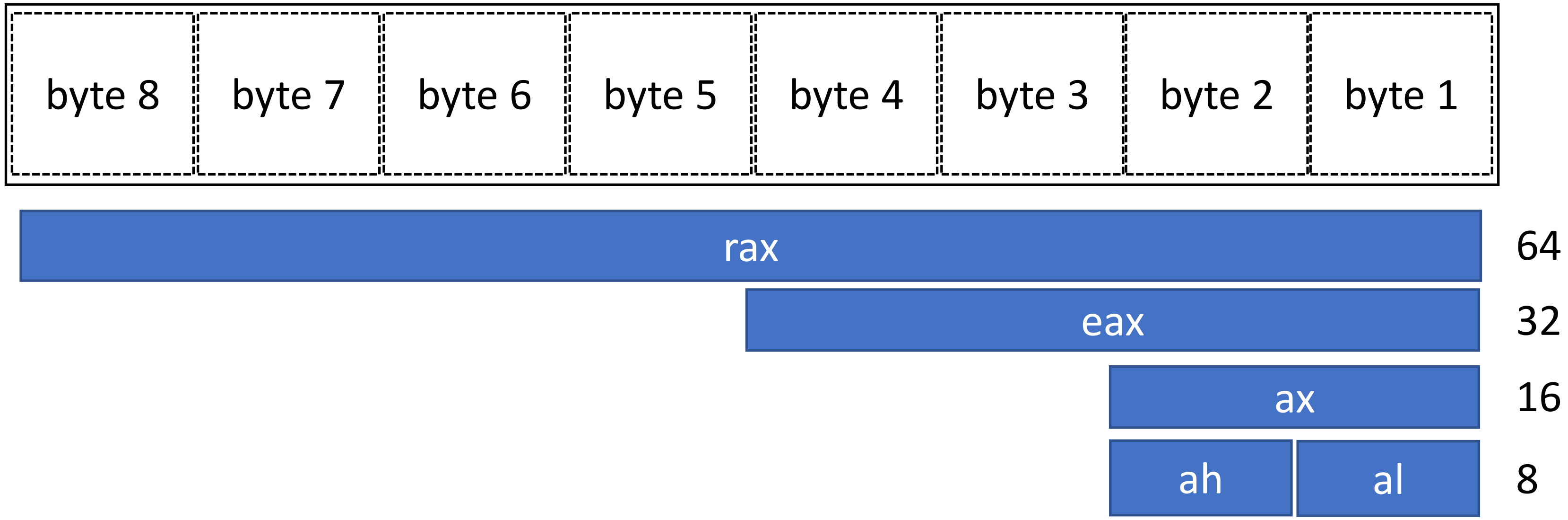
Can be used in Instruction opcodes (rax – r15)

Cannot be used in instruction opcodes (rip, rflags)

x64 Register Compatibility

Lecture Outline – About x64

Register #0 – the "A" register



This Time

Lecture Outline – Writing x64

Instruction-Set Architectures

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

Architecture

Stepping Back From Binary

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

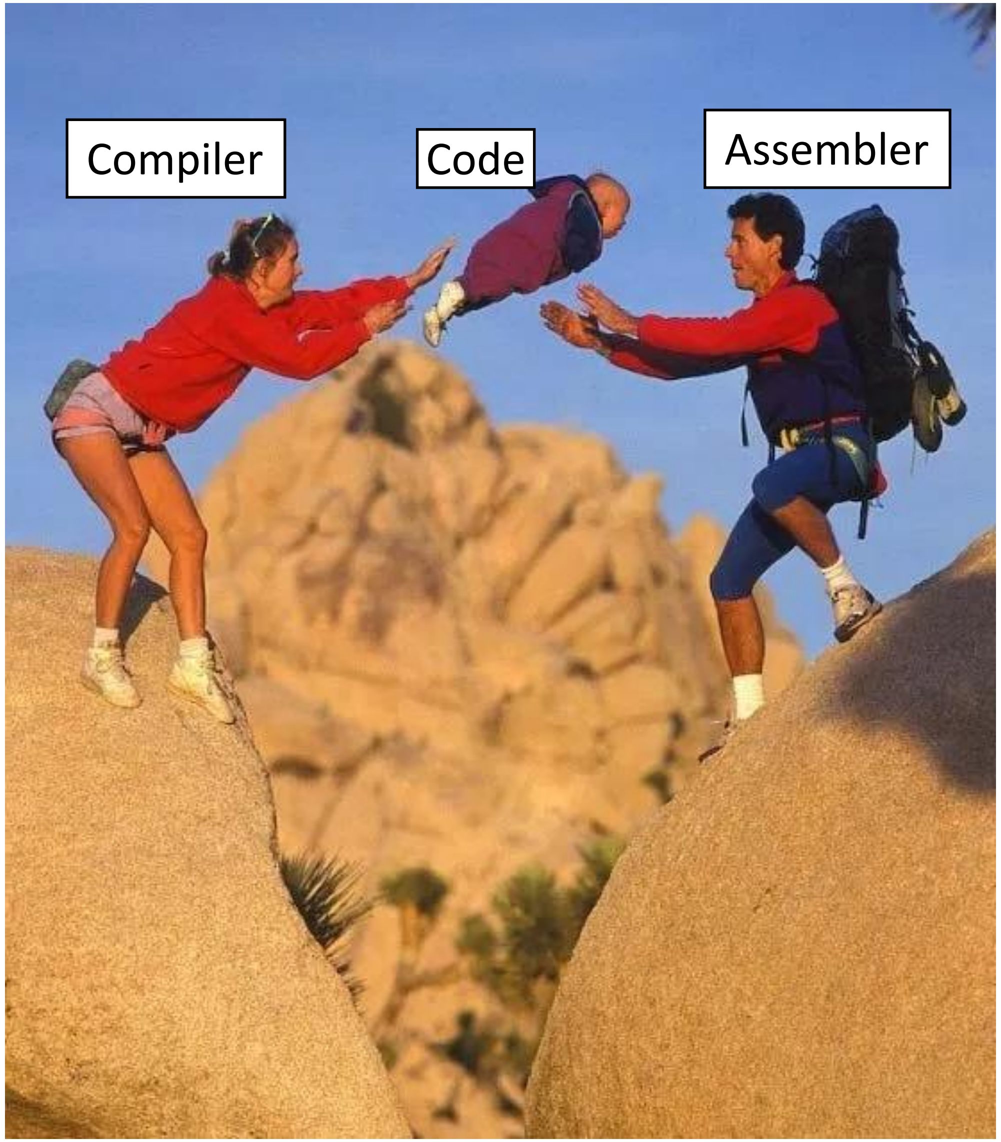
Fortunately, we don't have to do that

The Assembler

Lecture Outline –Writing x64

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

Lecture Outline –Writing x64

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

- 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

`.text`

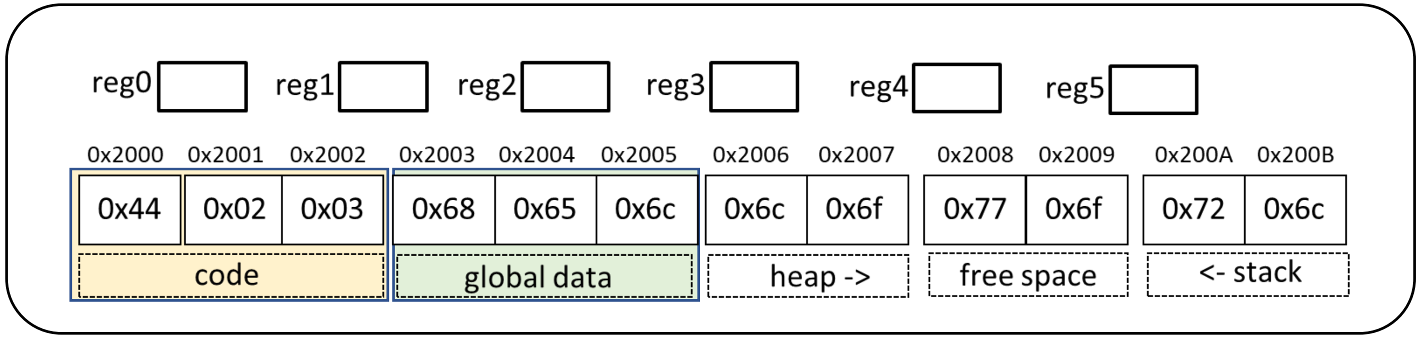
Lay out items in the user text segment

Instructions go here

`.data`

Lay out items in the data segment

Globals go here



Labels

Lecture Outline –Writing x64

- 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

```
jmp  LBL1
 …
LBL1: movq $5 (%rax)
```

```
jmp 0x12d34a5678a
```

System Calls

Lecture Outline –Writing x64

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!

Lecture Outline –Writing x64

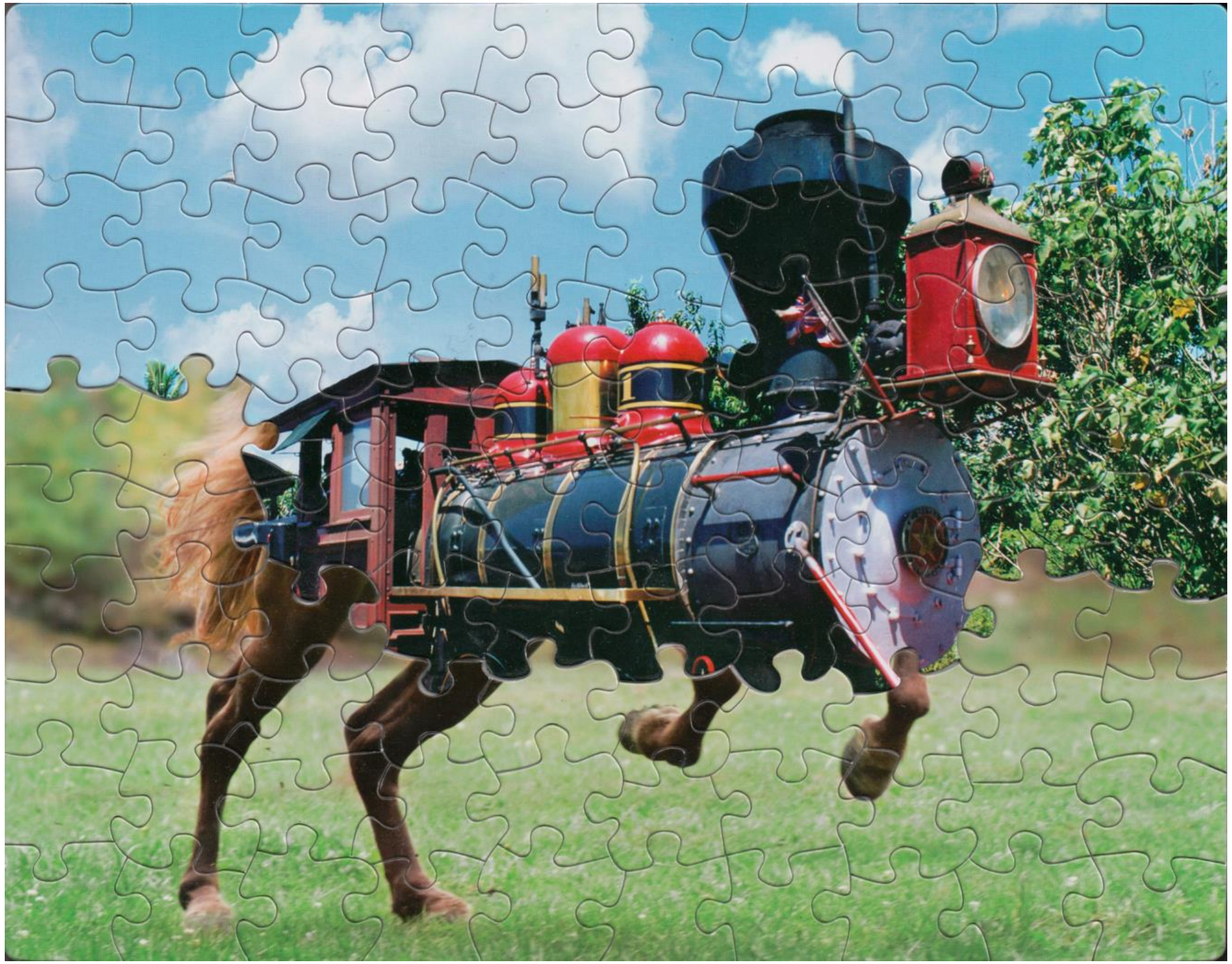


Photo Credit: Tim Klein - https://puzzlemontage.crevado.com

A Complete Program


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

Actually Running a Program

Invoking the assembler and linker

```
as -o start.o start.s
ld start.o -o prog
./prog
echo $?
```

Summary
ISAs

ISAs

- Provide an interface from software to hardware
- We’ll target assembly code, assembler will take it from there

X64

- A popular architecture
- We’ve covered the basic instruction format and a simple program

Next Time
ISAs

We'll dive into more details about X64