CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

July 2

Introduction

The Basics

CPSC 213: Introduction to Computer Systems

- Ask questions!
- What is a computer?
- What is a computer ***system***?
- Why should you care?
- How do you get an A?

A Theoretical Computer

CPSC 213: Introduction to Computer Systems

- A *model of computation* - so we can reason about algorithms
- Turing machine - an infinitely long tape…
- Useful *abstractions* for algorithms and theory

A Practical Computer

CPSC 213: Introduction to Computer Systems

- What are some examples of computers?

A Practical Computer

CPSC 213: Introduction to Computer Systems

- What are some examples of computers?
  - The human brain
  - DNA + Protein
  - Analog water computers
  - Quantum computers
  - **Digital computers**

A Practical Computer

CPSC 213: Introduction to Computer Systems

- Computers are **physical machines** and have **physical limits**
- Abstractions can only hide so much

What is a Computer ***System***?

CPSC 213: Introduction to Computer Systems

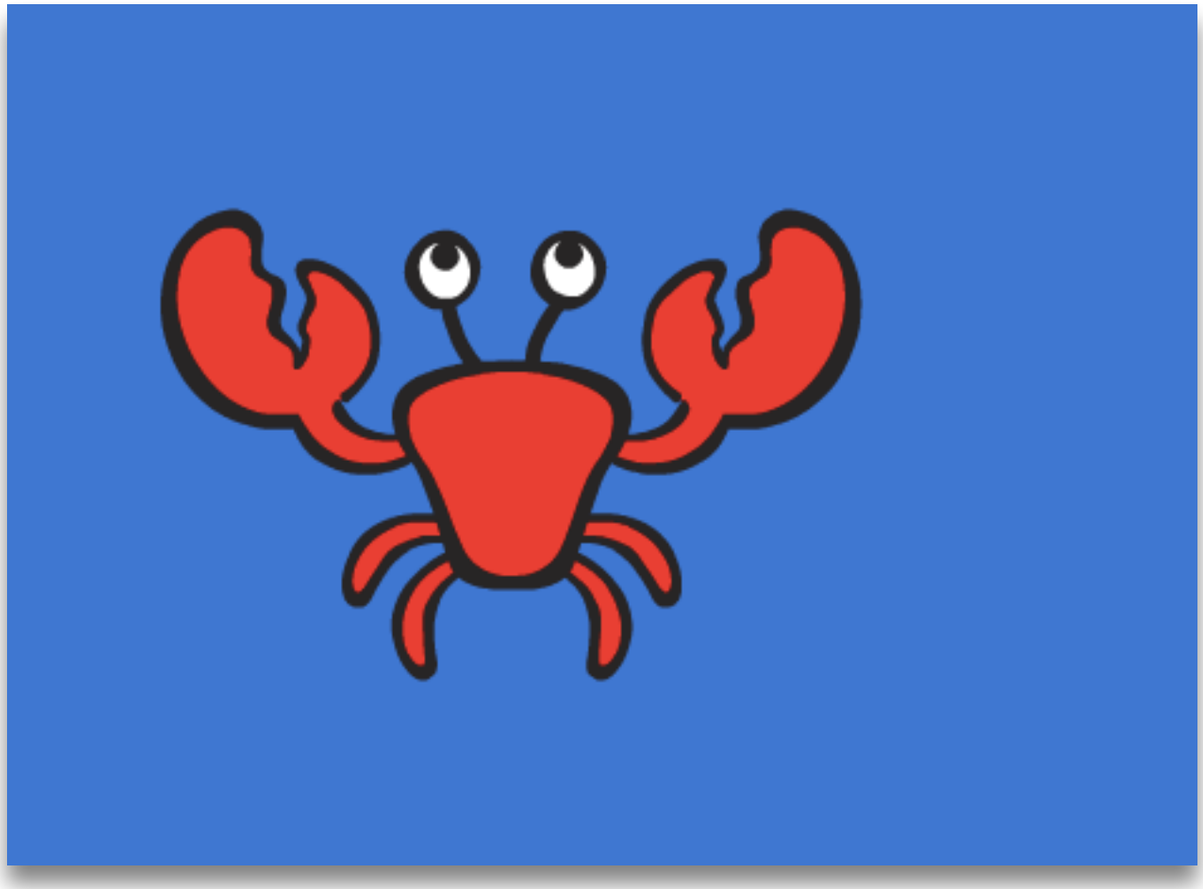
- We'll peel back the *computer* abstraction to reveal *computer systems* abstractions
- Abstractions literally all the way down

What is a Computer ***System***?

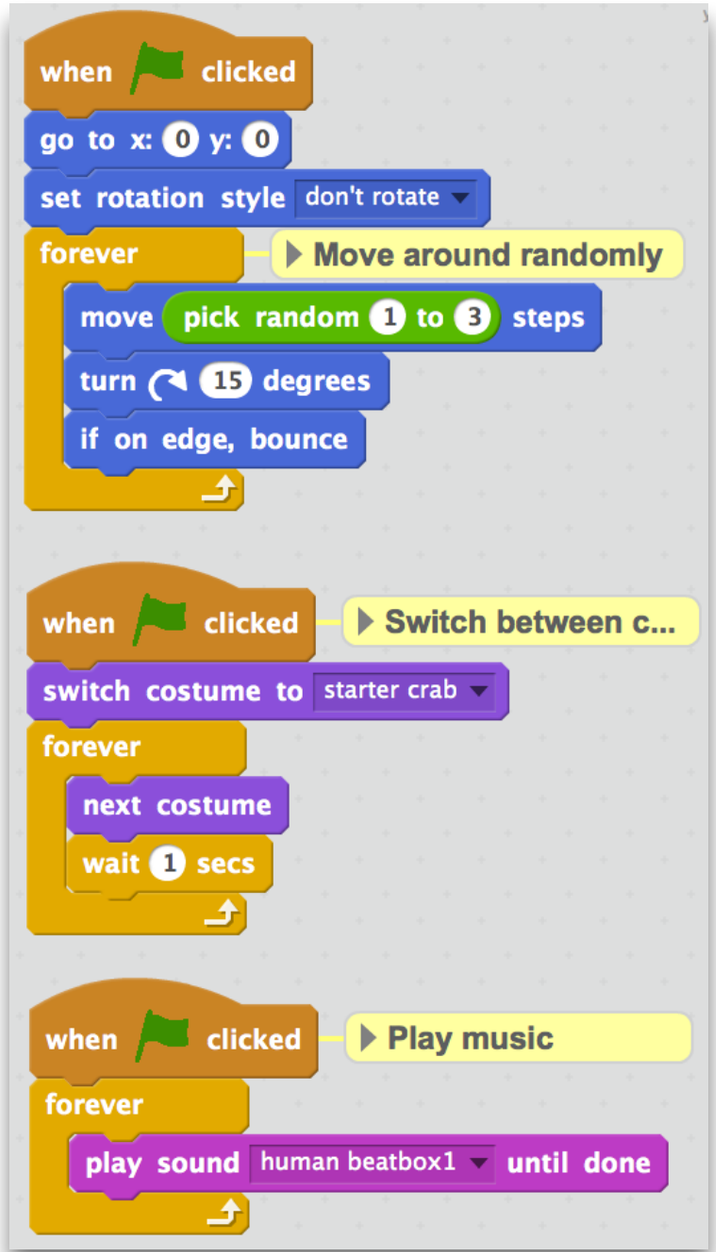
- An abstraction of the computing machine
  - lots of abstractions
  - used in various combinations by programmers
- Things like
  - hardware instruction-set architecture
  - operating system
  - compilers and high-level languages
  - middleware / libraries
- Where do ***you*** fit in?
  - programming is
    - building new abstractions … applications etc
    - using the existing ones … the system

Why Should ***You*** Care?

- You ***can*** use abstractions you don't understand
  - but your understanding is rooted in and restricted to the abstraction
  - e.g., computation in Dr. Racket, Java, C, C++, maybe Scratch …

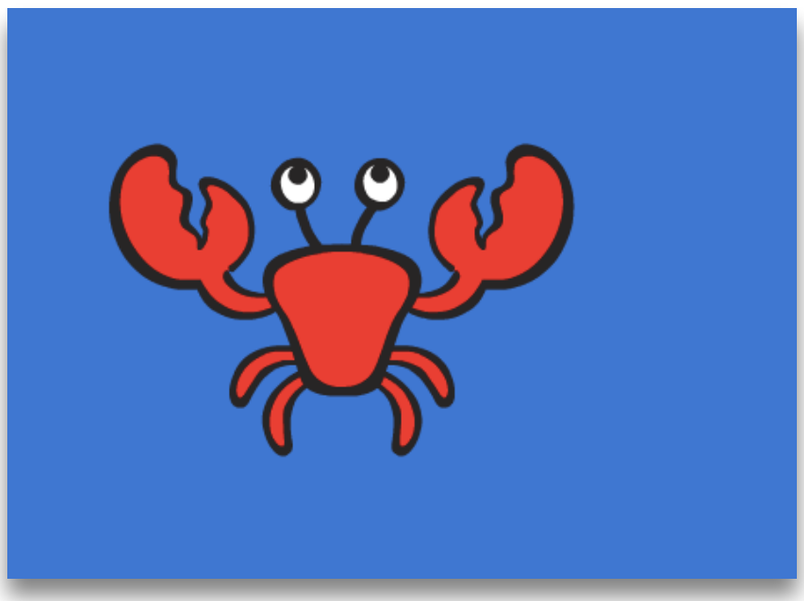


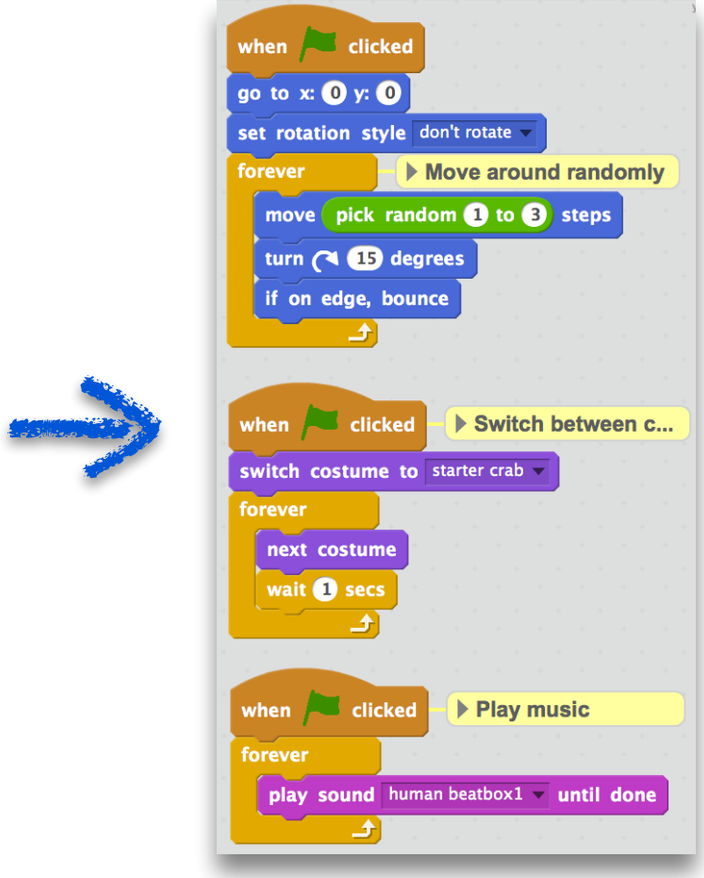
https://scratch.mit.edu/projects/114483977/



A Computer Scientist

- Understands computation deeply
  - mental model of computation rooted in the machine
  - informs understanding of abstraction
  - enables generalization among abstractions (e.g. languages)
  - empowers going deeper, beyond abstraction, when needed

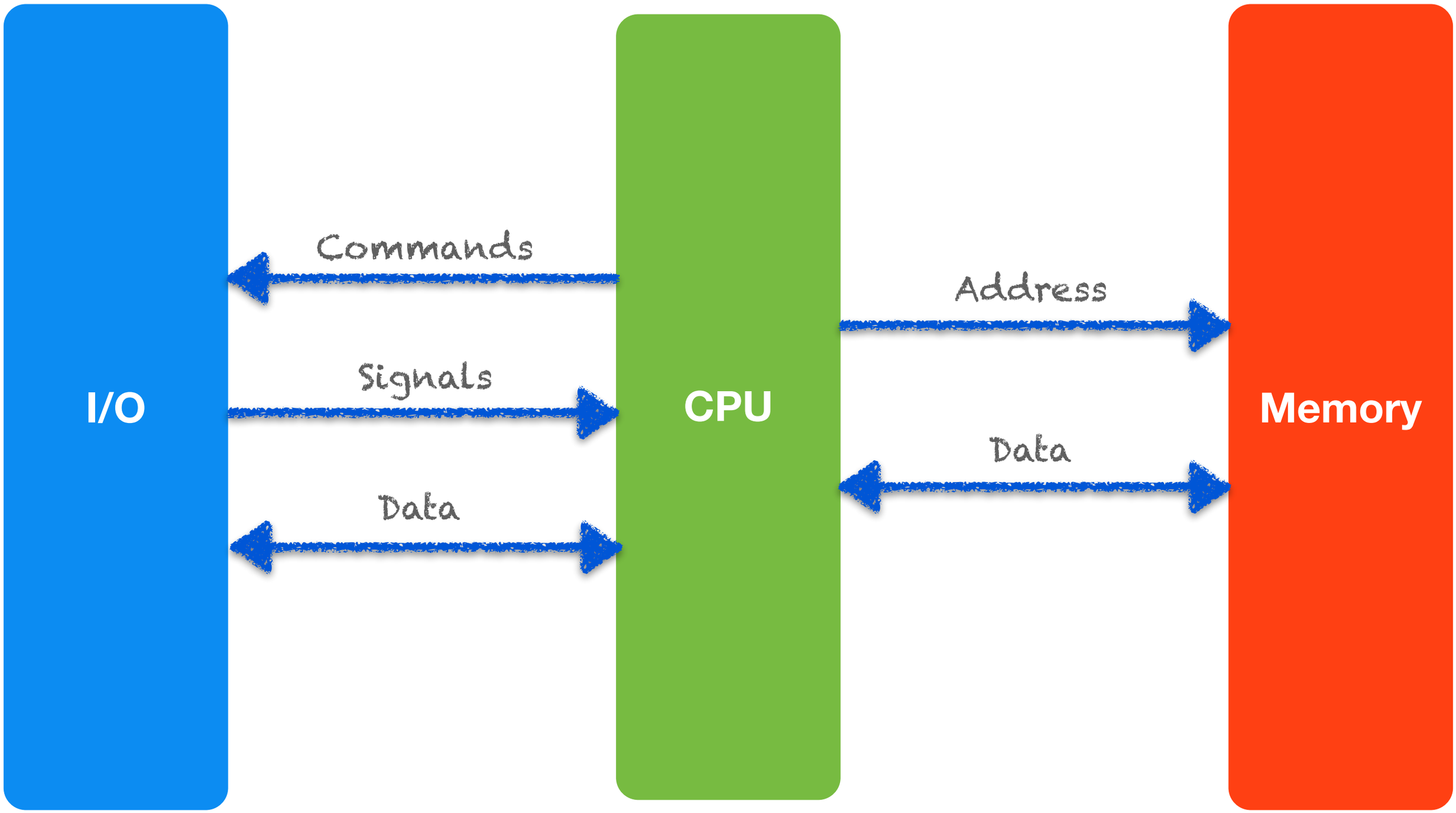




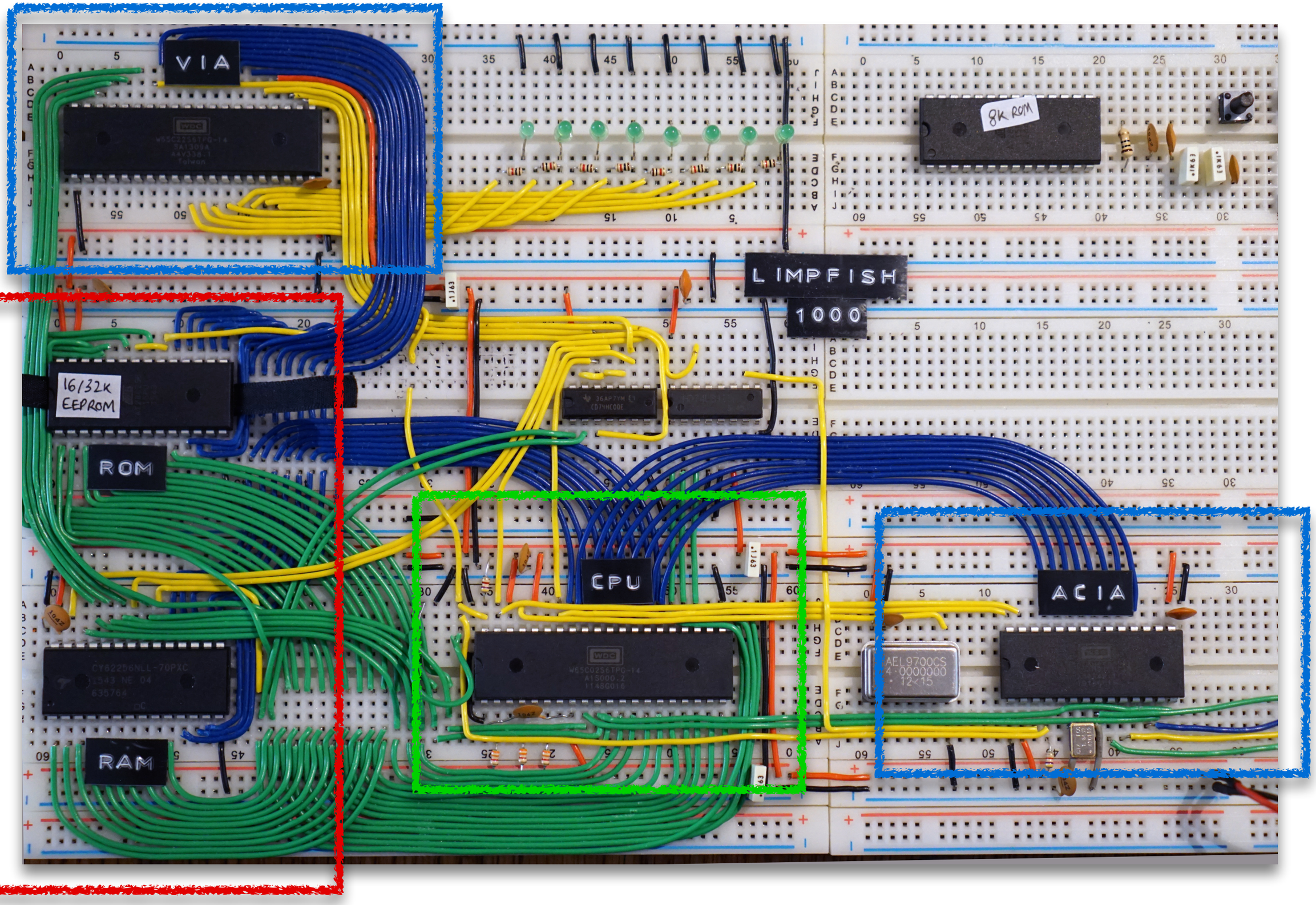
Why Should ***You*** *Care?*

- Let you write *better, faster* code by respecting and understanding limits
- Make you a more *powerful, flexible* programmer able to switch between abstractions
- Give you a *holistic view* of the computational stack and insight into what's actually happening

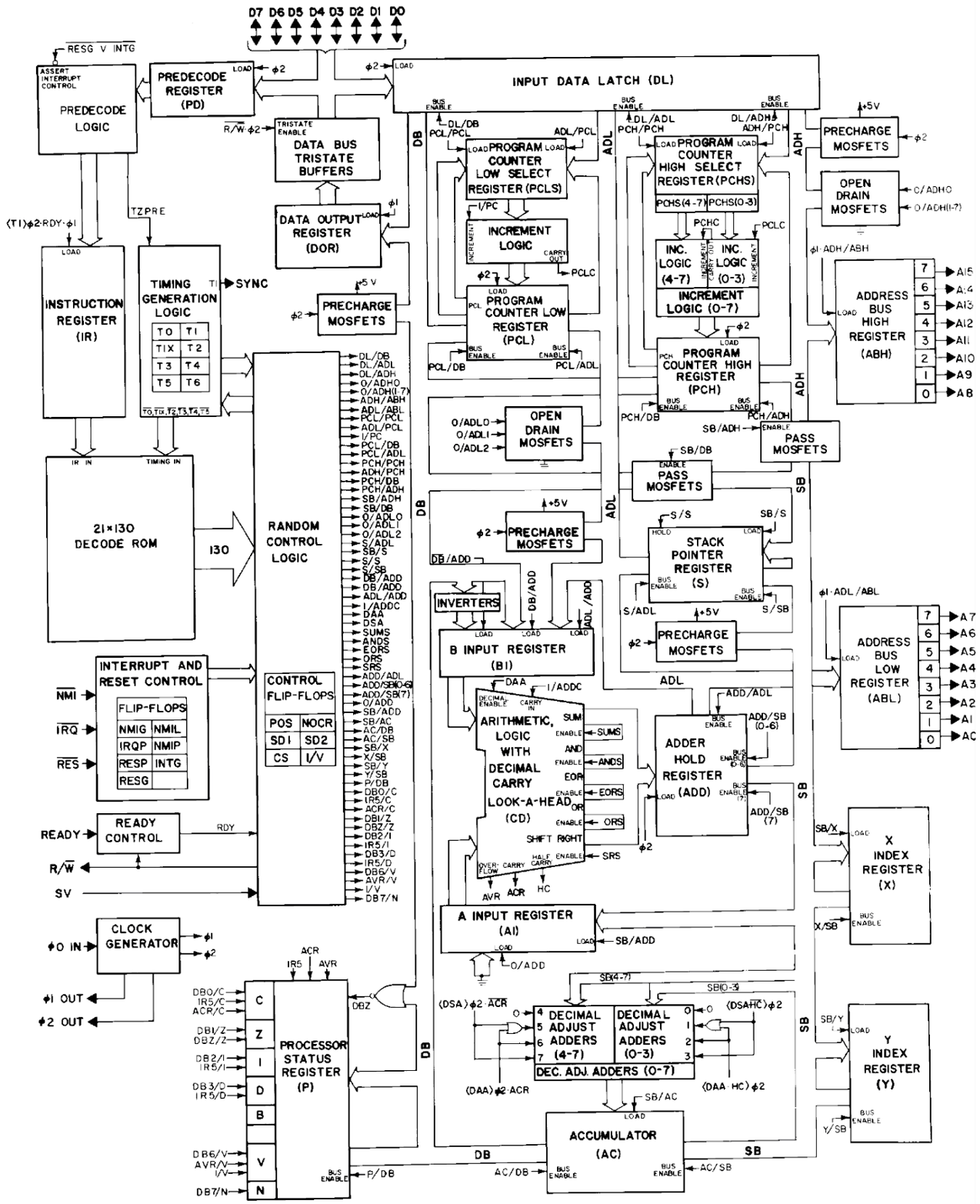
A Computer System Abstraction



A Computer System Abstraction

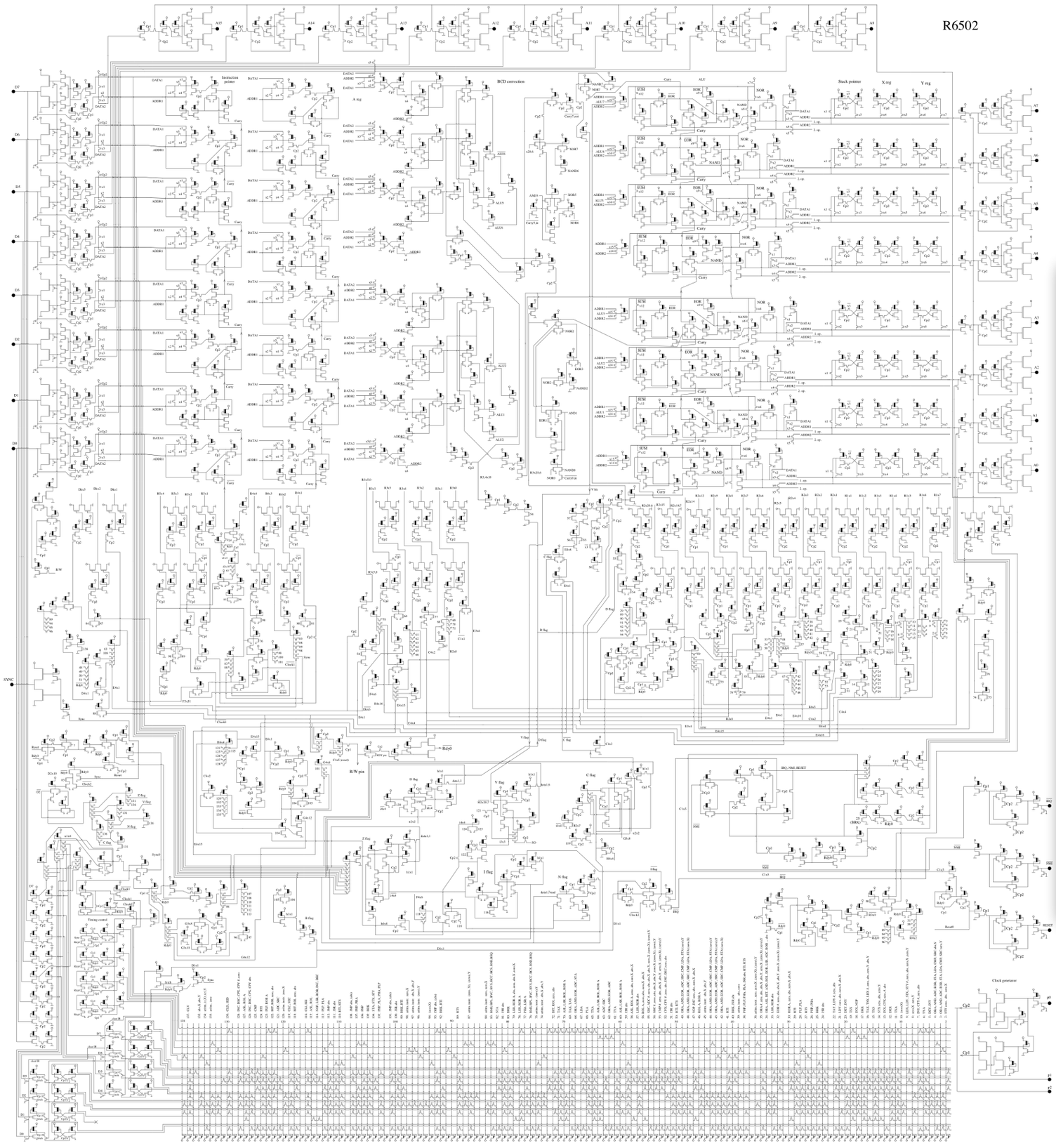


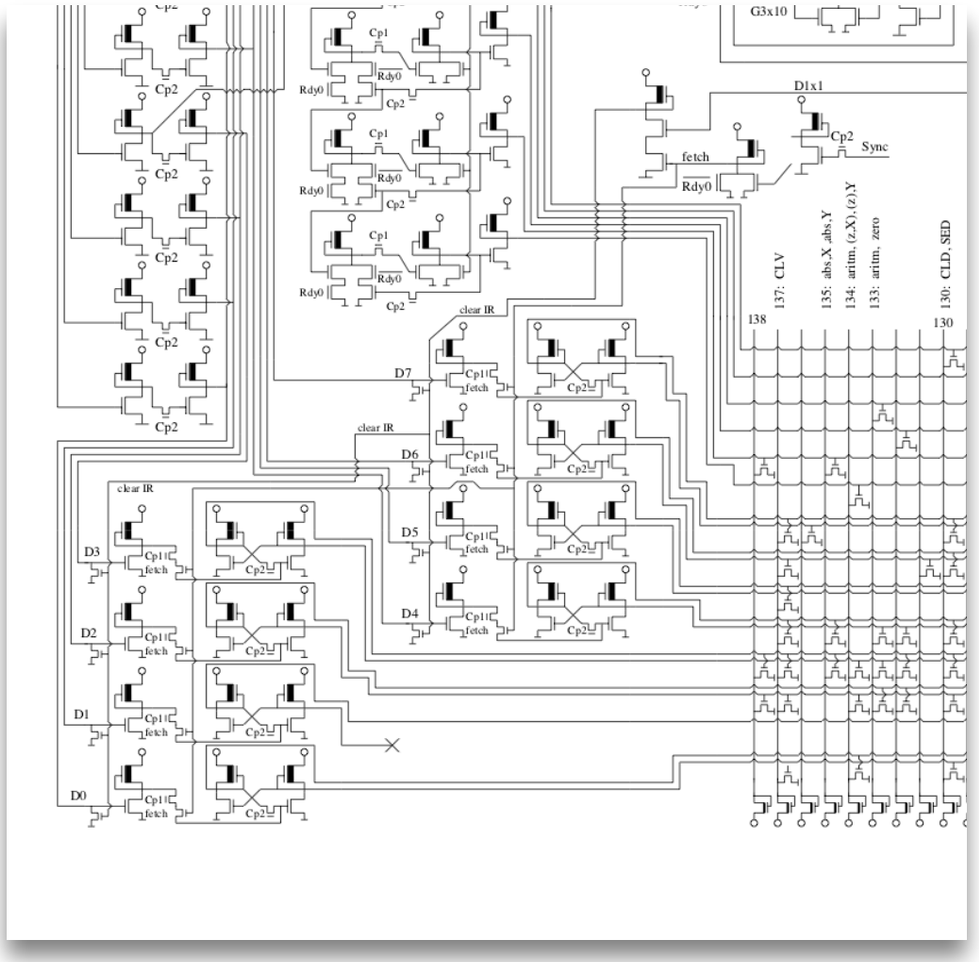
A Computer System Abstraction



Donald F. Hanson, WCAE 1995

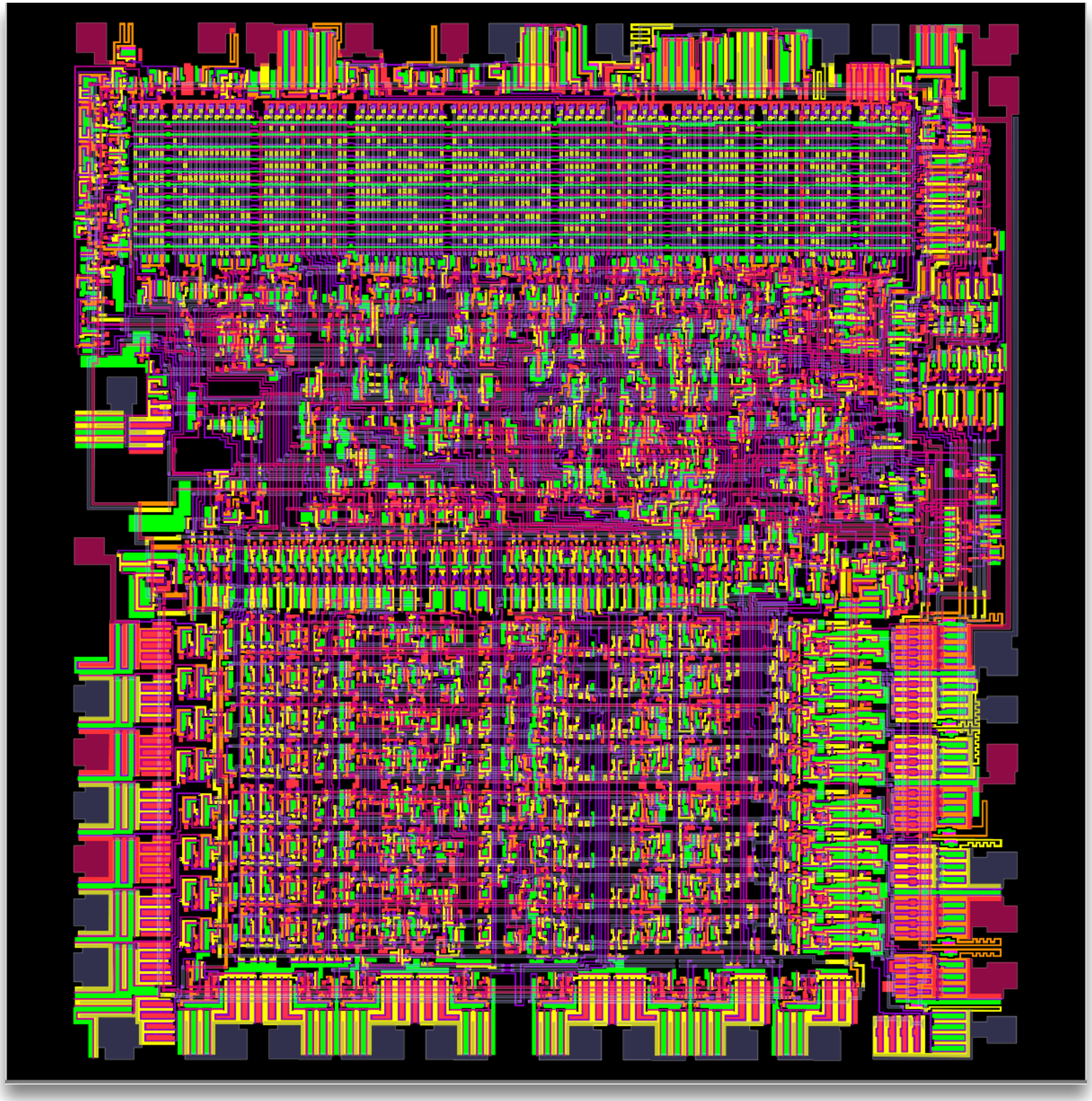
A Computer System Abstraction





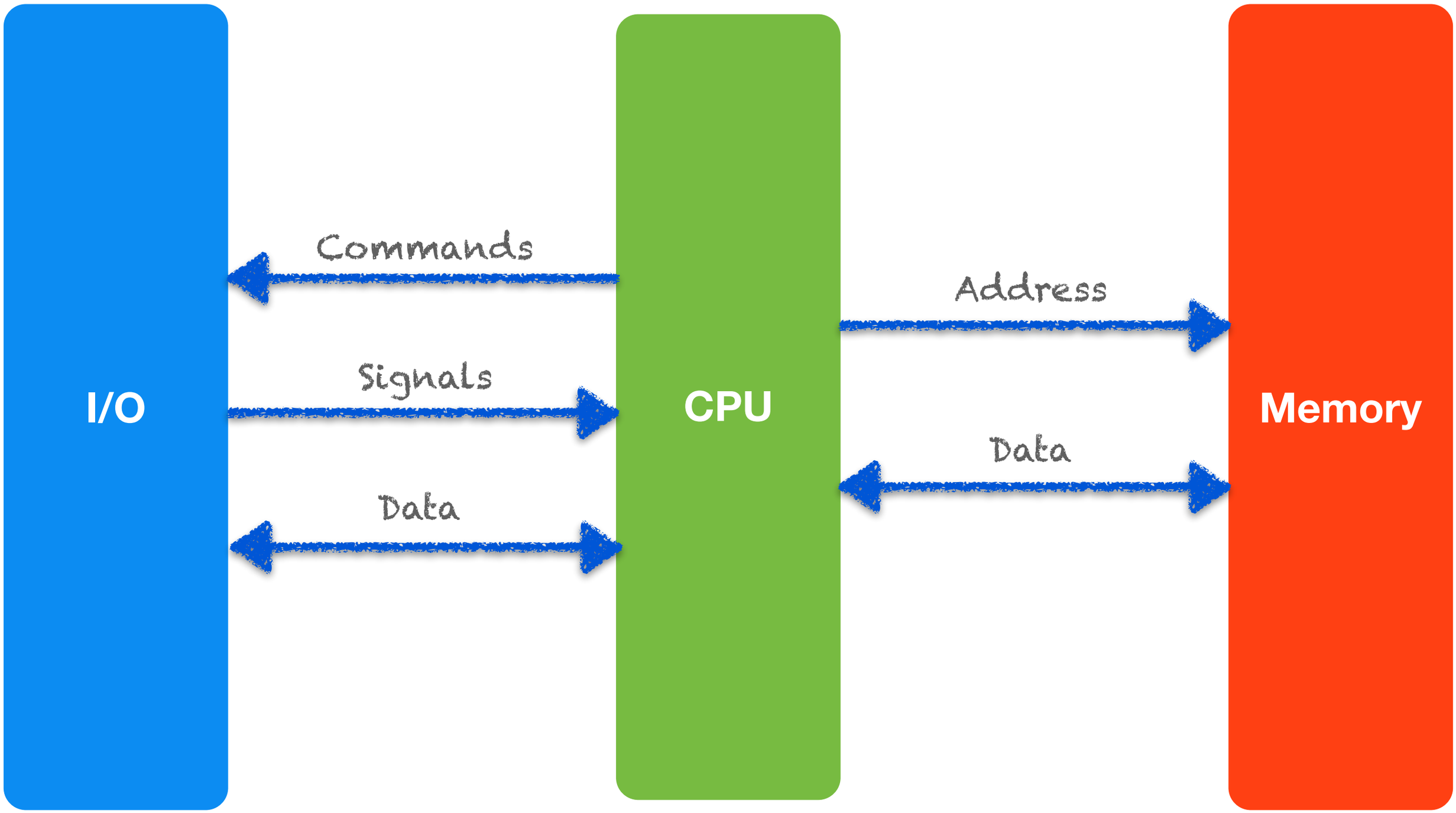
Bereghyei Balazs, visual6502.org

A Computer System Abstraction



visual6502.org

A Computer System Abstraction



Enough already ...
How do I get an A?

Course Learning Goals

Be a better programmer because, you will have a deeper understanding of the features of a programming language in order to be able to

a) understand in detail how your programs are executed,

b) be able to more easily learn new programming languages and

c) be able to evaluate design tradeoffs in considering languages most appropriate for solving a given problem.

Appreciate that system design is a complex set of tradeoffs which, while are important to be able to analyze will not have exactly one optimal answer (while there are often many sub-optimal answers). Tradeoffs exist at a range of levels including the hardware level, programming language level, etc. Experience with these tradeoffs prepares you to deal with tradeoffs in design in real world programming scenarios.

Develop distinctions between static and dynamic components of programs and systems and be able to describe their implications.

Utilize synchronization primitives to control interaction in various situations including among processes, threads, and networked communication.

Understanding how computing systems work.

Approach

- Hardware context of a single executing program
  - CPU and Main Memory
  - Static and dynamic computation
  - CPU architecture for C / Java
- System context of multiple executing programs with IO
  - IO, concurrency and synchronization
  - thread abstraction to hide IO asynchrony and to express parallelism
  - synchronizing to manage concurrency

Topic Learning Goals

- Memory
  - endianness and address alignment
- Programming Languages Features
  - static and dynamic global scalars, arrays and structs/objects
  - pointers in C
  - instance variables
  - dynamic storage
  - ifs and loops
  - procedures
  - dynamic control flow, polymorphism and switch statements
- Instruction Set Architecture
  - read and write assembly language
  - connection between ISA and high-level programming language

- Asynchrony
  - events and asynchronous programming
  - I/O devices: programmed IO, DMA and interrupts
- Concurrency
  - using and implementing threads
  - using and implementing spinlocks, monitors, and condition variables

There is a secret … to getting a A

- Work hard
  - Your brain is a computer
  - Current information **+ WORK** => new information
- Stay engaged
  - Be active in class, in labs and on piazza
- Stay on schedule
- Get help as soon as you need it
  - from me, TAs and each other
- Learn from your mistakes
  - review and correct assignments, quizzes and midterm

Logistics

- Piazza
  - everything will be accessible from there
  - register today: https://piazza.com/ubc.ca/summer2019/cpsc213/
  - in class questions
- Marks
  - 10 assignments (20%) **Labs start today, first assignment out today**
    - drop lowest 1 of first 9 (i.e., you can not drop the last assignment)
  - 10 online quizzes on Canvas (5%)
    - drop lowest 2
  - 1 midterm (25%)
  - final (50%)
- Work together but don't cheat
  - never present anyone else's work as your own
  - everything you hand in in this course should follow this rule
  - but, don't let that stop you in *your* role as teacher … helping each other