Computer Systems from the Ground Up

Winter 2025

https://cs107e.github.io

Have you ever wondered ...

- how a computer represents data?
- what operations a computer understands?
- how a program executes?
- what happens when a user types on keyboard?
- how text and drawing appears on a display?
- how things *really* work inside this wondrous box?

These questions and more to be answered by studying **computer systems!**

Who?

Ben

Haven

Didi

+ you!

Intrepid young padawans

Julie

Chris

Learning goals

1) Understand how computers represent data, execute programs, and control peripherals

2) Master your tools

Understanding is empowering!

RISC-V processor and memory architecture

Peripherals: GPIO, timers, UART, …

Assembly language and machine code

Low-level representation of information / bits

From assembly language to C

Function calls and stack frames

Serial communication and strings

Modules and libraries: Building and linking

Memory management: Memory map & heap

Processor control, interrupts

Master your tools

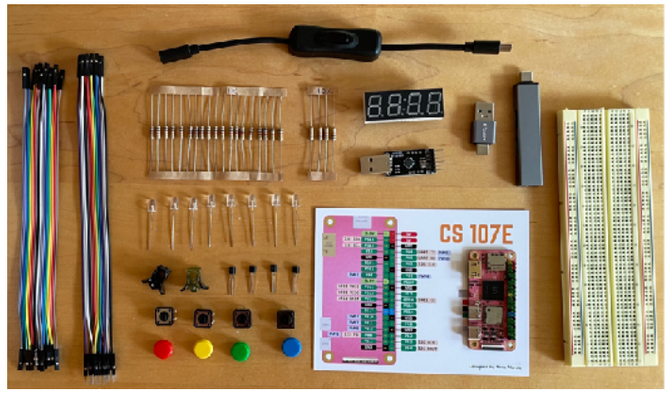
UNIX command line: bash, cd, ls, …
Text editor: vim, emacs, sublime, …
Programming languages: C, assembly
Compiler: gcc
Assembler: as
Linker/loader: ld
binutils: nm, objcopy, objdump, …
make
git and github.com
documentation: markdown

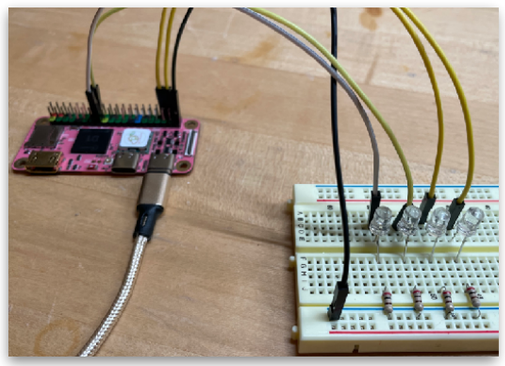

Non-Contact Voltage Detector
EXTECH INSTRUMENTS
REL
NCV
HOLD
MODE
RANGE
OFF
10A
EX330
AutoRanging MultiMeter
COM
600V MAX
Consolidated ELECTRONIC WIRE & CABLE
100 FT RoHS
36920
Consolidated ELECTRONIC WIRE & CABLE
100 FT RoHS
36822
Consolidated ELECTRONIC WIRE & CABLE
100 FT RoHS
36792
Consolidated ELECTRONIC WIRE & CABLE
100 FT RoHS
36856
CHP-170

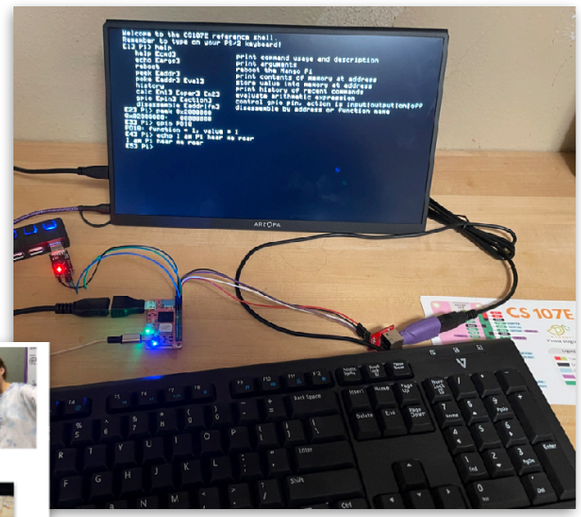
Syllabus

§1 Bare Metal Programming

- RISC-V architecture and assembly language
- C functions and pointers
- Serial communication
- Linking and loading
- Memory allocation







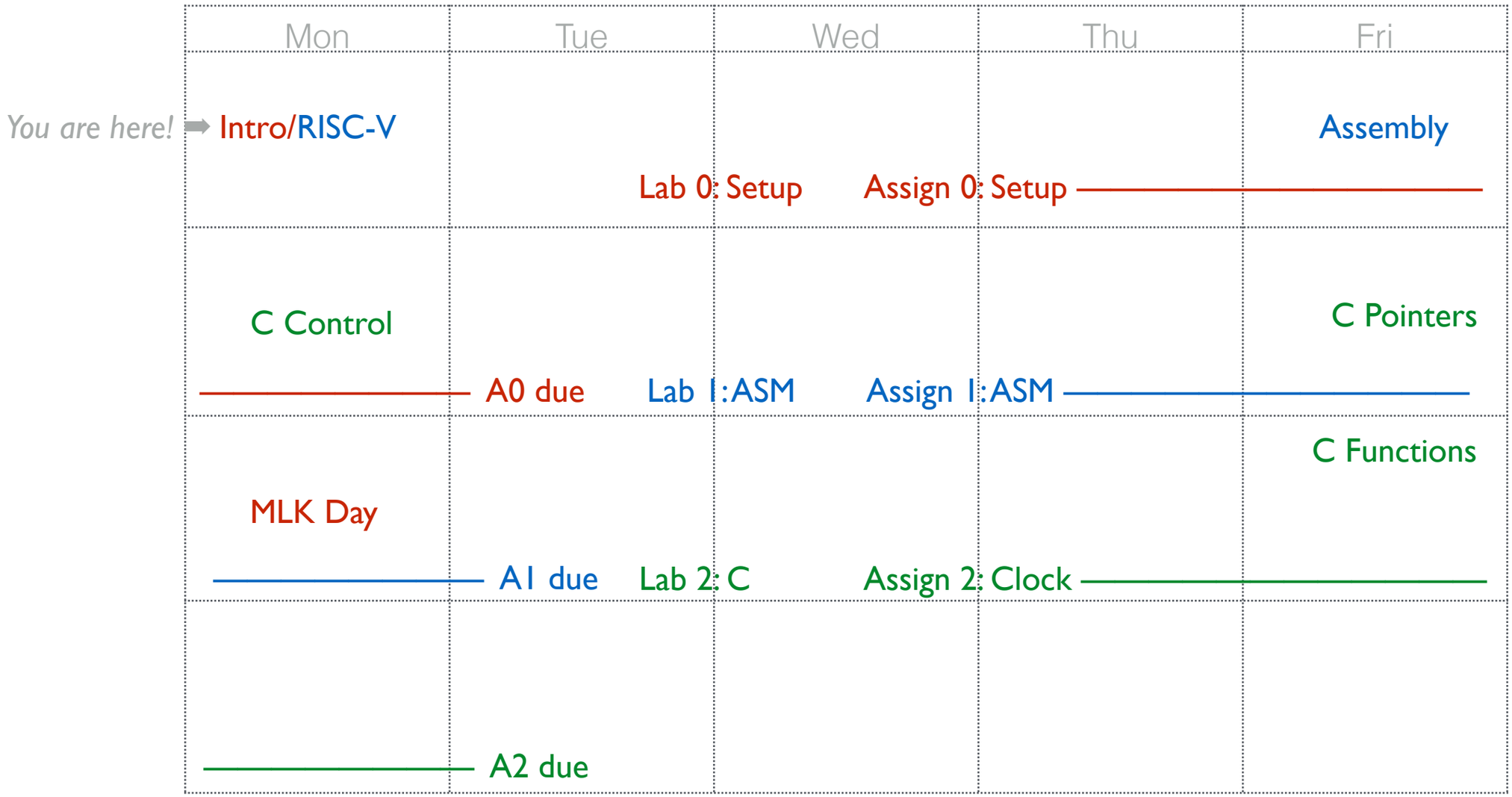
§2 Build a Personal Computer

- Keyboard
- Graphics
- Interrupts

§3 Create Your Own Project

- Sensors
- Performance

Weekly Cadence



https://cs107e.github.io/schedule/

Each week has a focus **topic**

Pair of coordinated **lectures** on Fri and Mon

Lab on Tue/Wed evening

Assignment handed out Wed after lab, due following Tuesday 5pm

Staying on pace leads to best outcomes!

Lectures

Attendance is **necessary**

Content is unique to our course, no textbook
The readings/slides not a standalone resource
Lectures not recorded

In-person attendance allows you to participate, ask questions, keep on schedule, stay connected

Labs

Set of guided exercises, follow up on lecture content

- Work in groups
- Complete exercises and check in with staff
- Leave lab ready to start assignment!

Lab participation is mandatory

Philosophy: lab is hands-on, collaborative, supported, **fun**!

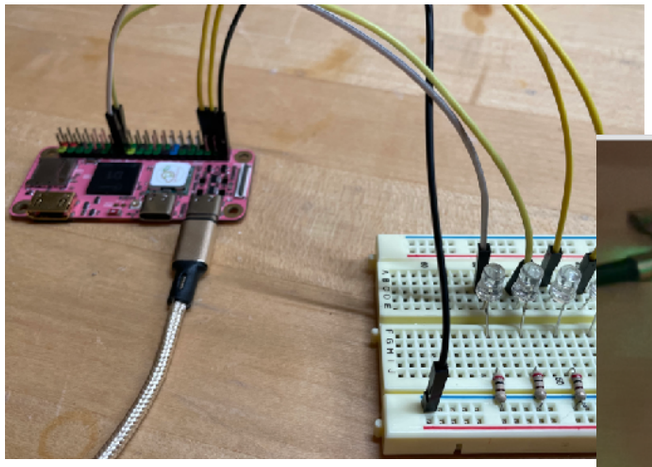
Assignments

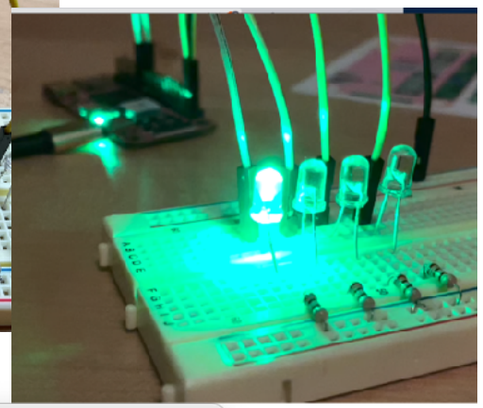
Weekly assignments that build on each other
This is where the learning really happens!

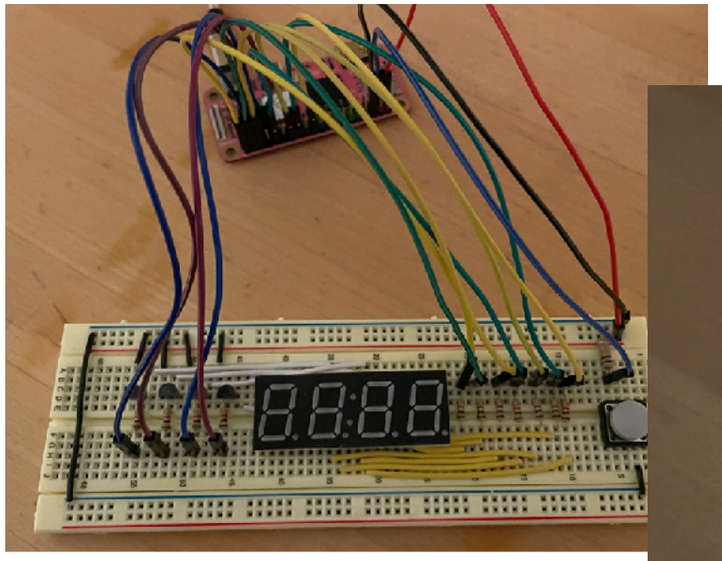
Each assignment has
- **Basic** requirement (tight spec, guided steps)
- Optional **extension** (opportunity for exploration/ creativity)

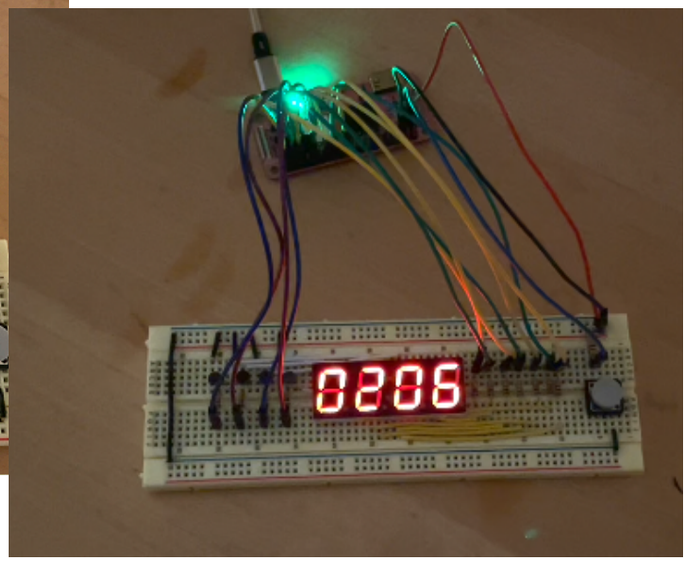
Opportunity to revise/resubmit to correct missed basic requirements

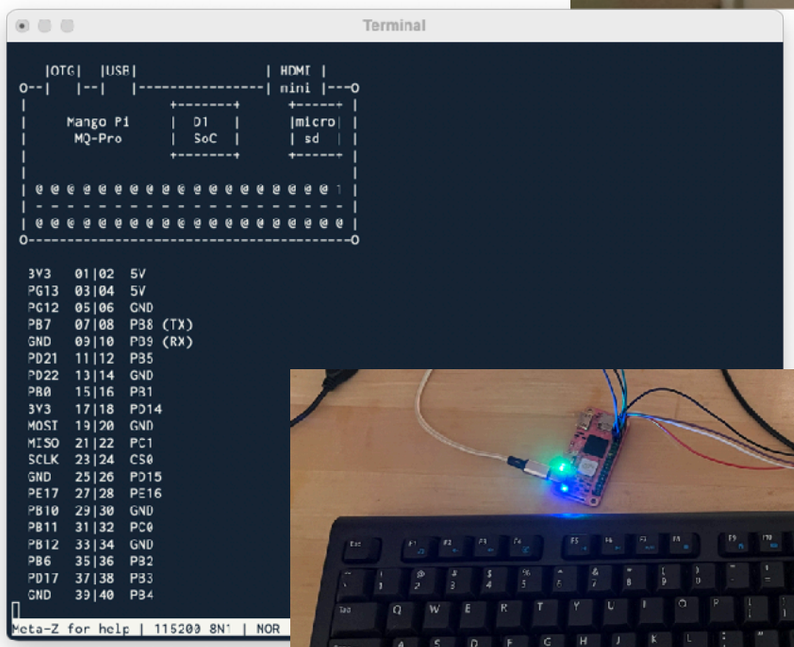
End goal is complete working system of your own

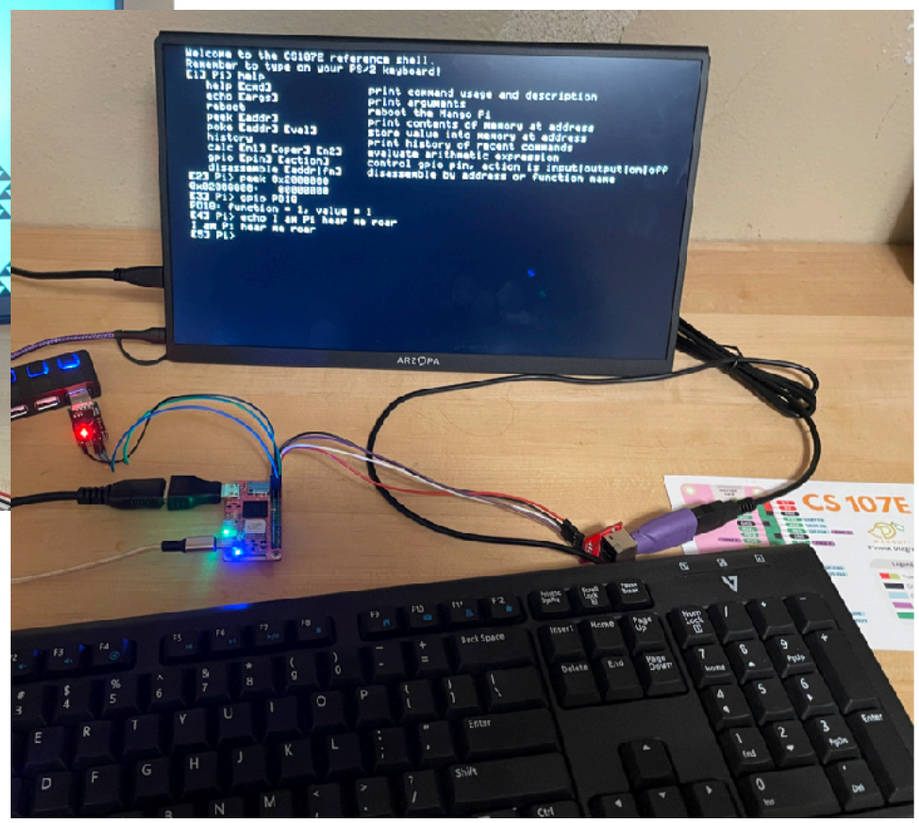






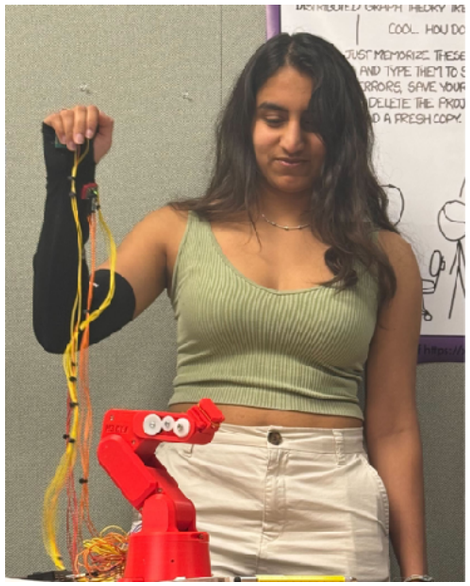


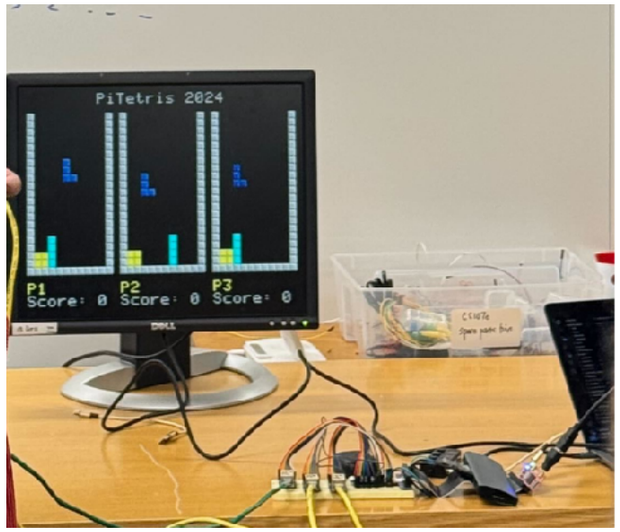


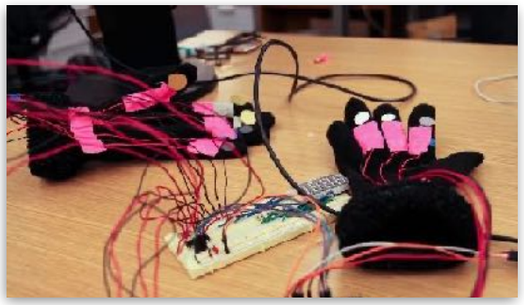


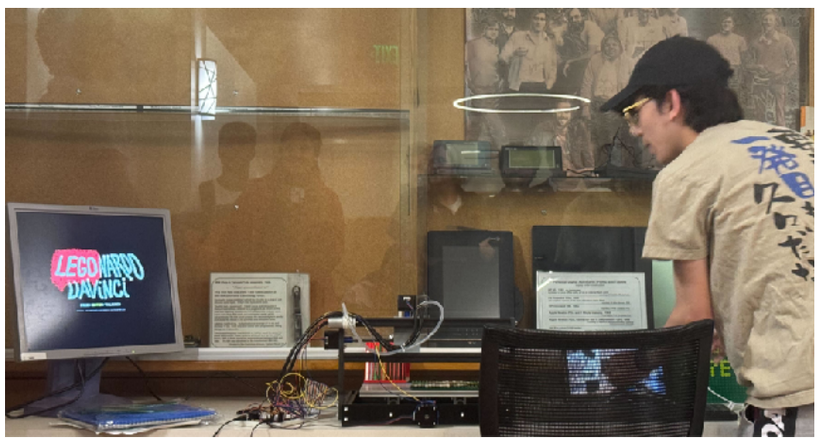
Nearly every instruction is code you wrote yourself!

Projects!




PiTetris 2024
P1
Score: 0
P2
Score: 0
P3
Score: 0




LEGONARDO DAVINCI

Markers for success

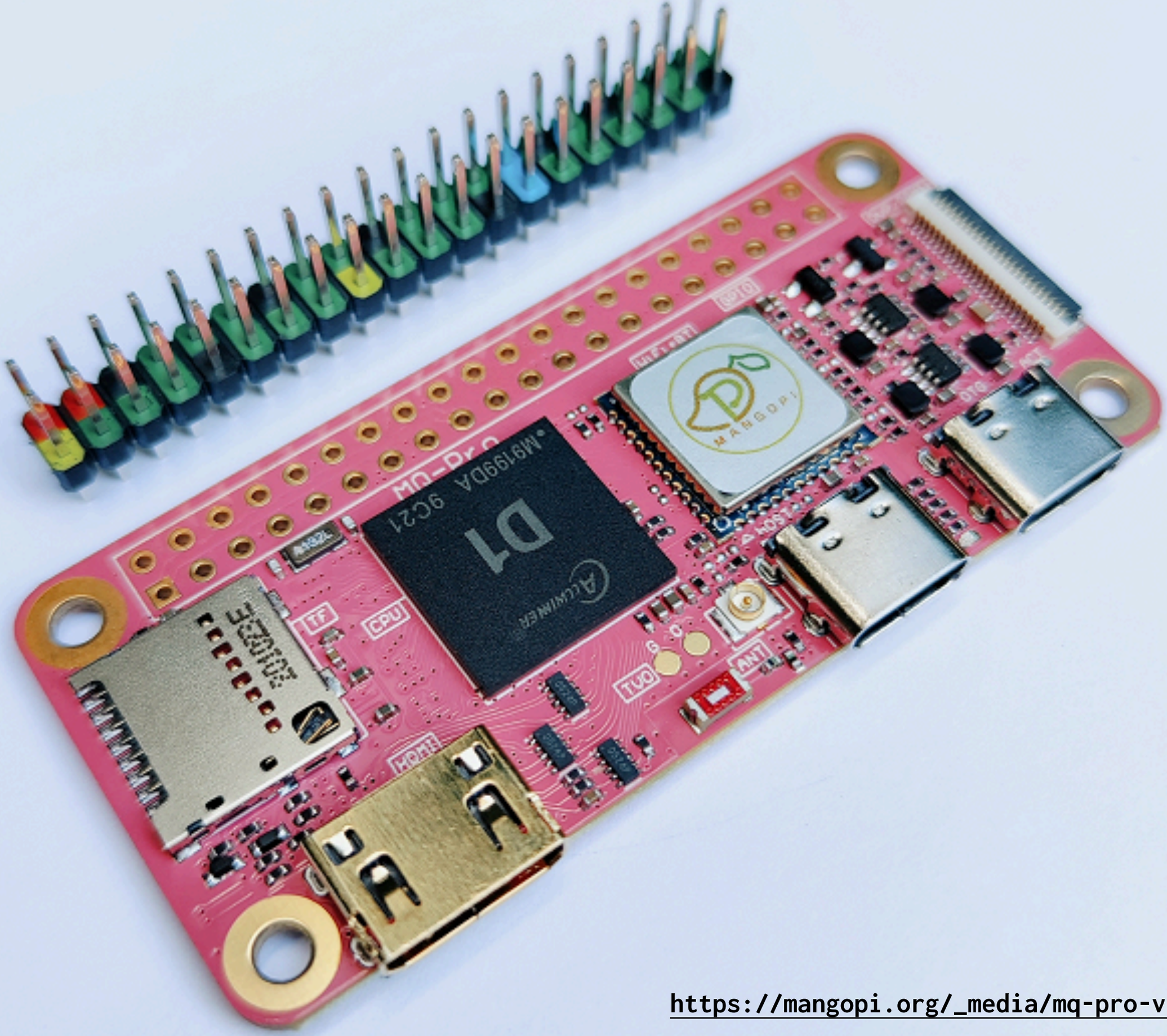
- Necessary prereqs: CS106B, C++, debugging

- Curiosity
- Perseverance
- Motivation

Value and appreciate small learning community!

How to thrive in this course

- Consistency, follow through
- Leverage our resources, support, feedback
- Be **curious**. Learn by **doing**. Ask for and offer **help**.



https://mangopi.org/_media/mq-pro-v12-ibom.html

von Neumann architecture

CPU:
- ALU
- Registers
- Control Unit
  - instruction register, program counter

Memory:
- Stores data and instructions

Mechanisms for input/output

Critical innovation:
- both instructions and data kept in memory
- "stored program" computer

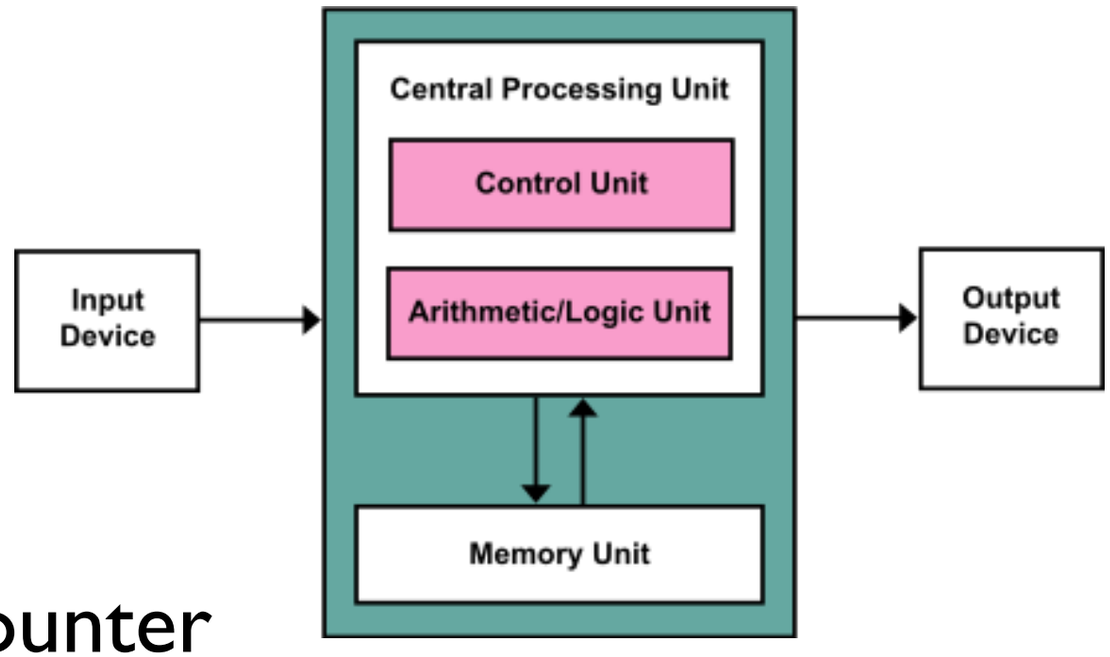


Image credit https://en.wikipedia.org/wiki/Von_Neumann_architecture#/media/File:Von_Neumann_Architecture.svg

Memory Map

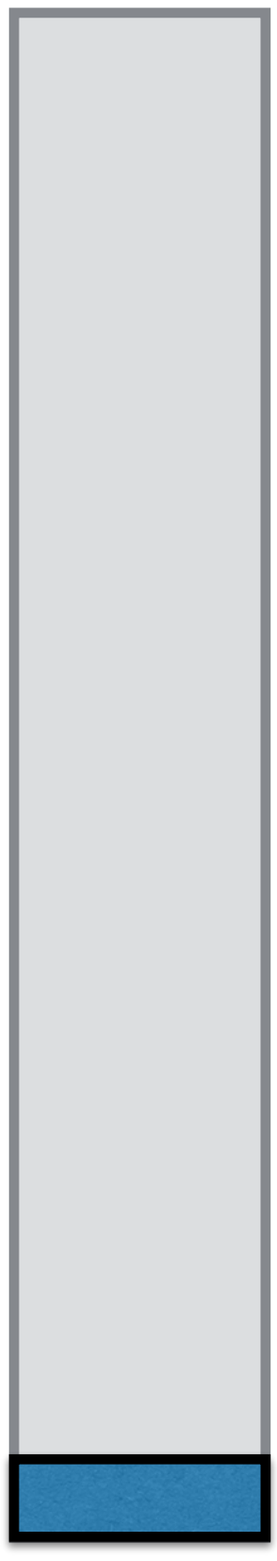
Memory is a large array

Storage locations are accessed using a 64-bit index, called the *address*

Address refers to a *byte* (8-bits)

4 consecutive bytes form a *word* (32-bits)

Maximum addressable memory is 16EB (But... 42-bit address lines)



First week: set up

Before lab:
- Review course guides:
    Unix tools (shell, editor, git)
    Electricity (simple circuits, Ohm's law)
    Number representation (binary, bit operations)
- Install development tools

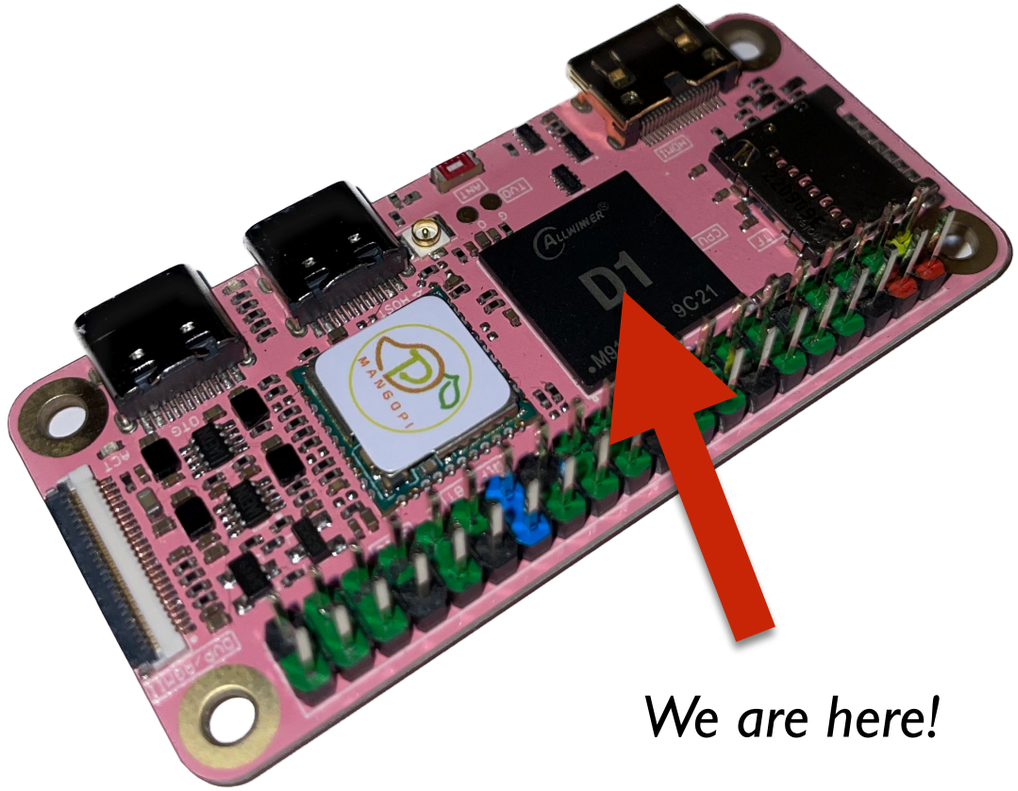
During lab:
- Establish comfort with background topics
- Practice with environment/tools, build productive habits
- Get help resolving any installation snags
- Meet one another!

Today: RISC-V ISA

Historical milestones in computer architecture

Origins of RISC-V

RISC-V instruction set architecture



We are here!

RISC-V®

Whirlwind history tour

Mainframe era (50s,60s)

- IBM had 4 incompatible lines of computers
- `System/360`: one ISA to rule them all (backward compatibility becomes a thing...) now oldest surviving ISA
- DEC `PDP-8 -> PDP-11 -> Vax` (Bell Labs, unix)
- ISA prioritize assembly programmer

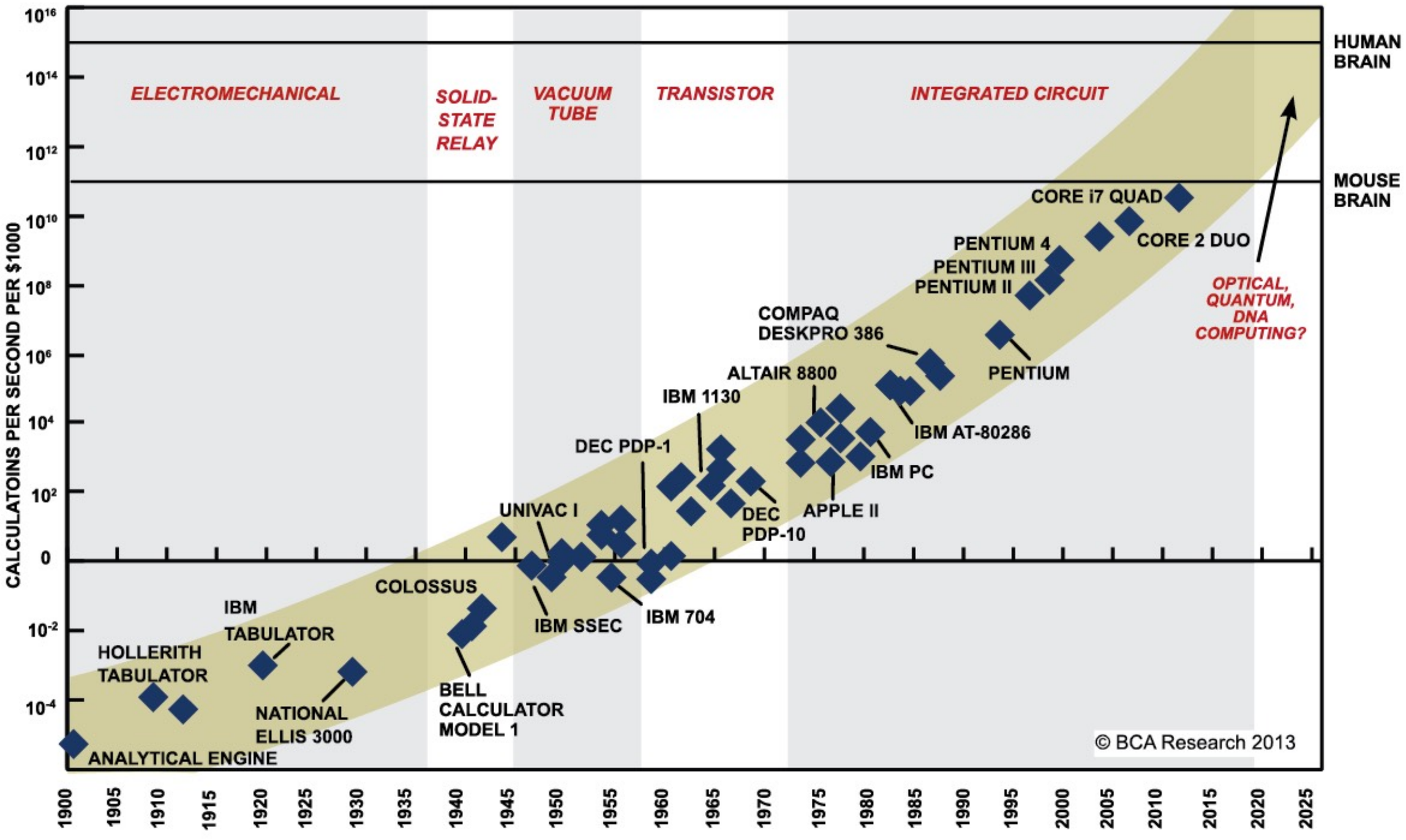
Personal desktop computer

- Intel iAPX 432 6+ year visionary failure (32-bit, OO, Ada)
- 8086 stopgap developed 1 year (16-bit 8Mhz 29K transistors)
- Increasing use of HLL, compiler

Moore's Law

- Count of transistors in IC doubles every 2 years

Progression, Moore's law



SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

The “semantic gap”

Gap between HL languages and machine code

One HL construct = many machine instructions

Improve performance of compiled code by adding fancier machine instructions (procedure calls, array access, etc)?

CISC (complex instruction set computer)

Insns to match HL can be complex to implement in hardware

But... researchers in 70s work out:

Compilers mostly don't emit the complex instructions

Avoid special cases, sequence of simple ops often faster anyway

Real-world programs spend most of their time executing simple ops

Complex ops slow down execution, even when you don't use them

From CISC to RISC

Hennessy & Patterson 1982 **MIPS**
(microprocessor w/o interlocked pipeline stages)
RISC reduce footprint, small number of simple ops

Many advantages to keeping base simple
- Simpler to design/verify
- Lower costs (die area, higher yield, energy consumption)
- Enable pipelined implementation
- Higher throughput (faster clock, fewer cycles per ins)
- Don't pay for what you don't use

Assembly examples

C code

```
void set(int arr[], int i, int val) {
    arr[i] = val;
}
```

x86

```
0: 8914b7       movl  %edx, (%rdi,%rsi,4)
3: c3           retq
```

ARM

```
0:   e7802101    str   r2, [r0, r1, lsl #2]
4:   e12fff1e    bx    lr
```

rv

```
0: 00259593     sll a1,a1,0x2
4: 00b50533     add a0,a0,a1
8: 00c52023     sw  a2,0(a0)
c: 00008067     ret
```

AAA—ASCII Adjust After Addition

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
37	AAA	ZO	Invalid	Valid	ASCII adjust AL after addition.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the addition produces a decimal carry, the AH register increments by 1, and the CF and AF flags are set. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, bits 4 through 7 of the AL register are set to 0.

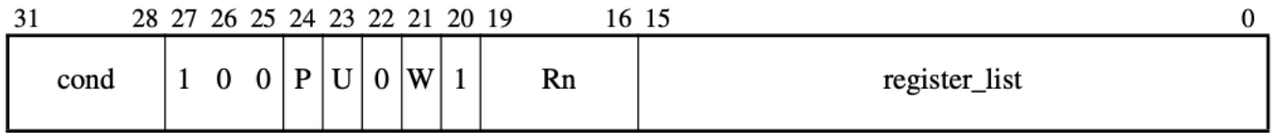
x86 puts the C in CISC

arm is RISC but …

ARM Instructions

```
LDMIAEQ SP!, {R4-R7, PC}
```

A4.1.20 LDM (1)



LDM (1) (Load Multiple) loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations. It is useful for block loads, stack operations and procedure exit sequences.

The general-purpose registers loaded can include the PC. If they do, the word loaded for the PC is treated as an address and a branch occurs to that address. In ARMv5 and above, bit[0] of the loaded value determines whether execution continues after this branch in ARM state or in Thumb state, as though a `BX (loaded_value)` instruction had been executed (but see also *The T and J bits* on page A2-15 for operation on non-T variants of ARMv5). In earlier versions of the architecture, bits[1:0] of the loaded value are ignored and execution continues in ARM state, as though the instruction `MOV PC,(loaded_value)` had been executed.

Syntax

```
LDM{<cond>}<addressing_mode>  <Rn>{!}, <registers>
```

RISC-V origins

2010 Cal EECS, research iterating on HW design

Existing ISAs no good (complex, proprietary, legacy)

Goal define clean-slate ISA

Substrate for future research

Teaching architecture/systems

Andrew Waterman, Yunsup Lee
Dave Patterson, Krste Asanovic


Design of the RISC-V Instruction Set Architecture

Andrew Waterman

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-1
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html

January 3, 2016

https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf

Design goals

"Universal"

Suitable for microcontrollers up to supercomputer

Data-informed

Benchmarks, measurements, 50 years of experience

Set footprint from intersection not union

Modular, extensible

Small standard base ISA

Defined extensions outside core

Opcode space for custom/specialized extension

Stable

Maintain backward compatibility (extensions frozen)

Evolve via additional extensions

Free & open

Allows collaboration, competition, innovation

Open cores -> secure & trustworthy, design your own

Interface/implementation

ISA

Externally visible aspects

registers, instructions, data types, control/exceptions

Huge value of standard, no IP hurdles

Microarchitecture

Implementation of ISA

Logic design, power/performance tradeoffs

Room for innovation, competition

e.g. Intel & AMD both implement x86, but very different microarchitectures

Abstraction for the win!

How to understand an ISA

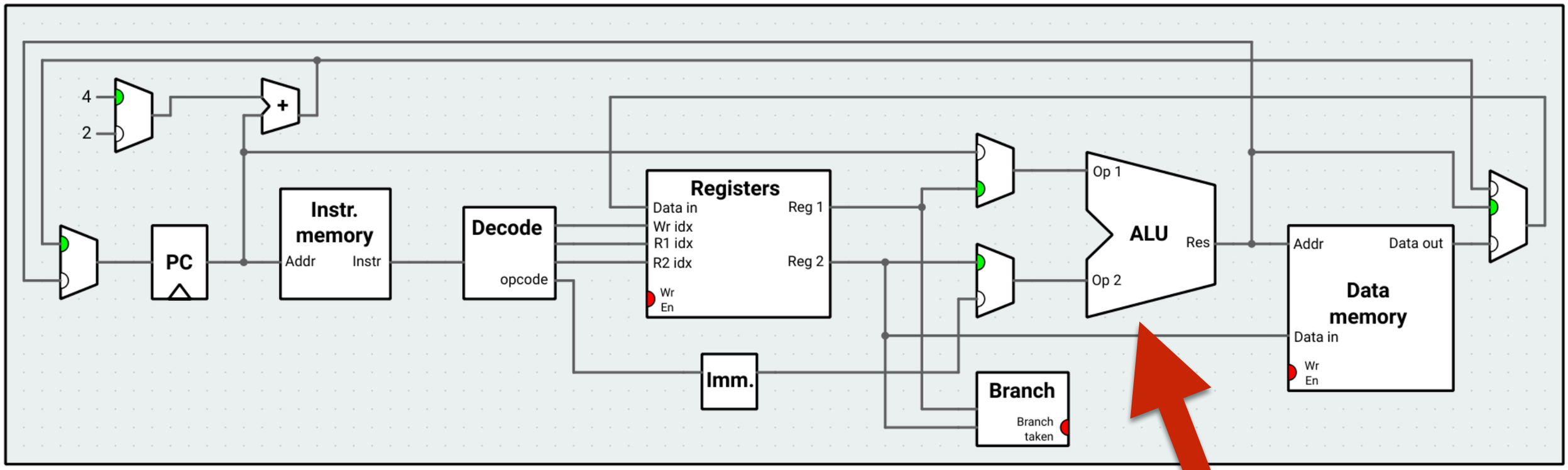
We want to learn how processors represent and execute instructions.

One means of learning an ISA is to follow the data paths in the "floor plan".

Tracing that paths shows where information can flow from/to and how that dictates the operational behavior

Visualizer/simulator can be helpful! `https://ripes.me`

RISC-V Architecture / Floor Plan



https://ripes.me/

The "Arithmetic Logic Unit"
This is where arithmetic (e.g. add)
and logic (e.g., and/or) happen

XLEN-1 … 0
x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

XLEN

32 registers (x0-x31)

XLEN-1 … 0
pc

XLEN

PC = program counter

Example ALU instructions

```
add rd,rs1,rs2
sub rd,rs1,rs2
and rd,rs1,rs2
or  rd,rs1,rs2
sll rd,rs1,rs2
srl rd,rs1,rs2
```

R-type (three registers: dest, source1, source2)

```
addi rd,rs,imm12
andi rd,rs,imm12
ori  rd,rs,imm12
slli rd,rs,imm12
```

I-type (source2 is immediate/constant)

More at https://cs107e.github.io/guides/riscv-onepage/

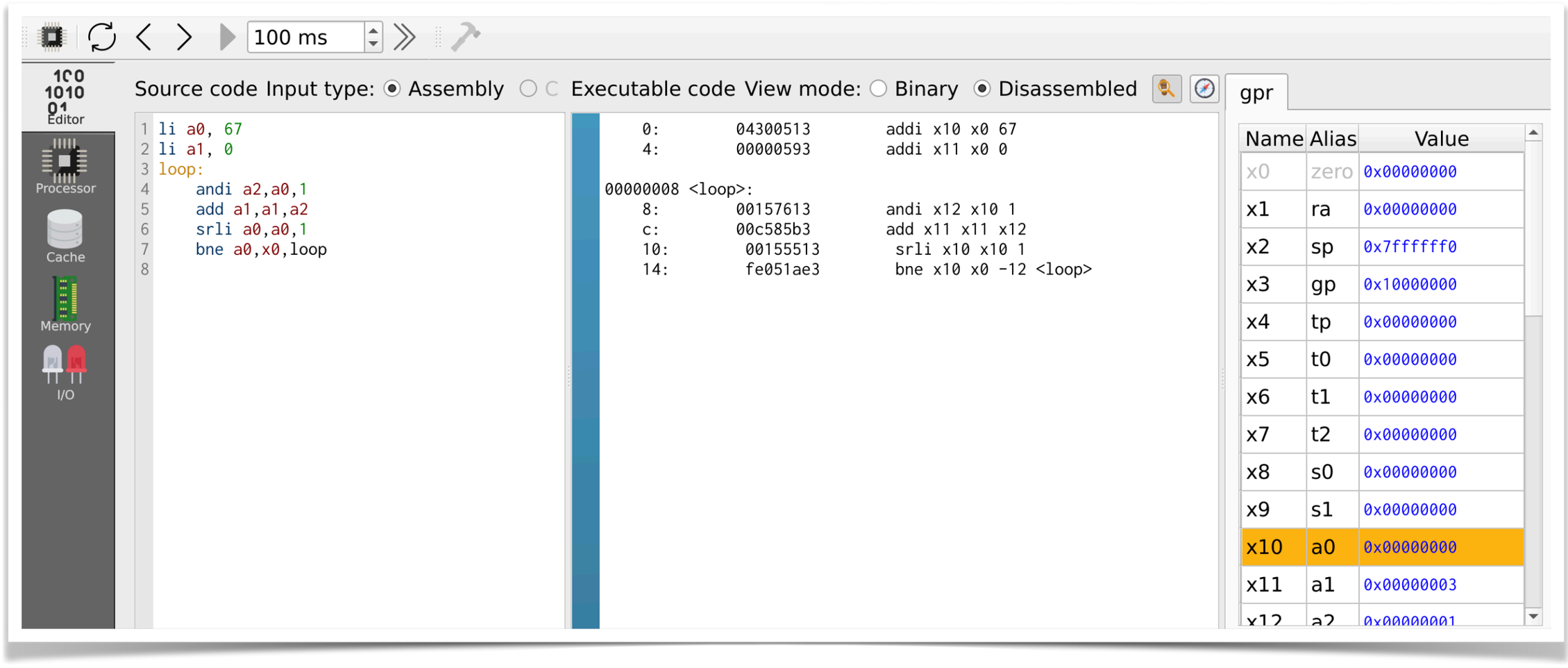
Challenge for you all:

Write an assembly program to count the "on" bits in a given numeric value

```
li a0, some-number
li a1, 0

// a0 initialized to input value
// use a1 to store count of "on" bits in value
```

Ripes visual simulator



Try it yourself! https://ripes.me

Instruction encoding

Another way to understand the design of an ISA is to look at how the bits are used in the instruction encoding.

RISC-V uses 32-bit instructions. Packing all functionality into a 32-bits encoding necessitates trade-offs and careful design.

RISC-V instruction encoding

32-bit RISC-V instruction formats

Format	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- ***opcode* (7 bits):** Partially specifies one of the 6 types of *instruction formats*.
- ***funct7* (7 bits) and *funct3* (3 bits):** These two fields extend the *opcode* field to specify the operation to be performed.
- ***rs1* (5 bits) and *rs2* (5 bits):** Specify, by index, the first and second operand registers respectively (i.e., source registers).
- ***rd* (5 bits):** Specifies, by index, the destination register to which the computation result will be directed.

6 instruction types
Regularity in bit placement to ease decoding
Sparse instruction encoding (room for growth)

ALU encoding

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type

add x3, x1, x2

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

```
0000000 00010 00001 000 00011 0110011
   0     0    2    0   8    1   B   3
```

Tip: in python3: `print(bin(0x002081b3))`

Immediate encoding

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type

imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI

Your turn! `addi a0,zero,21`

```
00000001010100000000010100010011
 0   1   5   0   0   5   1   3
```

Key concepts so far

Bits are bits; bitwise operations

Memory addresses (64-bits) index by byte (8-bits), word is 4 bytes

Memory stores both instructions and data

Computers repeatedly fetch, decode, and execute instructions

RISC-V instructions: ALU, load/store, branch

Resources to keep handy

RISC-V one-page guide

Ripes simulator