Computer Systems from the Ground Up

Winter 2026

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**!

Learning goals

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

2) Master your tools




Who?

Aditri

Daniel

Frances

+ you!

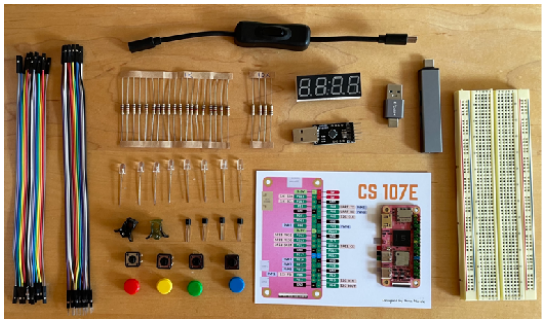
Intrepid young padawans

Elias

Joe

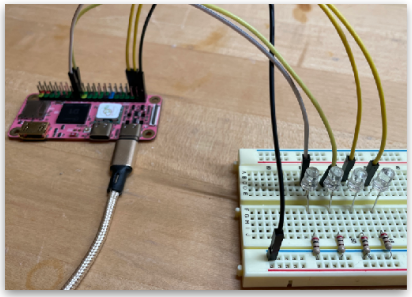
Julie

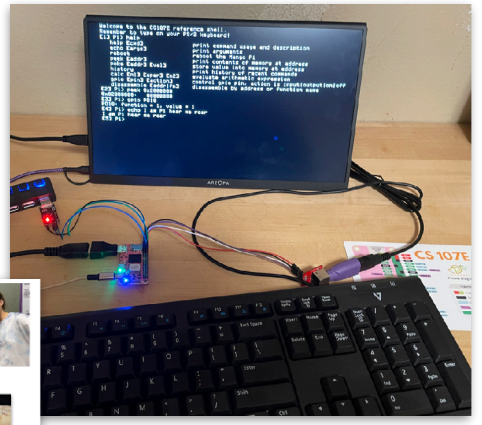
Syllabus



§1 Bare Metal Programming

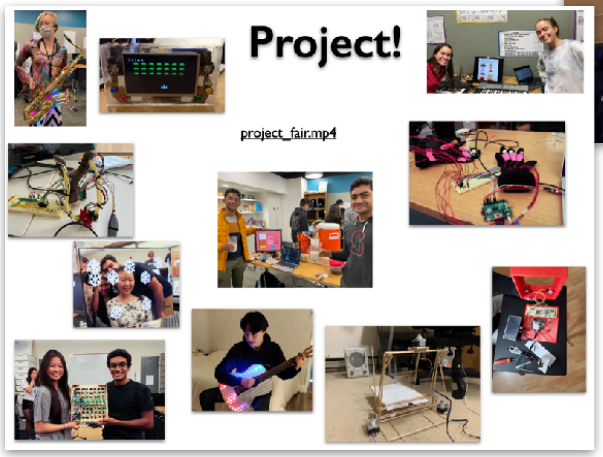
- RISC-V architecture and assembly language
- C functions and pointers
- Serial communication
- Linking and loading
- Memory allocation





§2 Build a Personal Computer

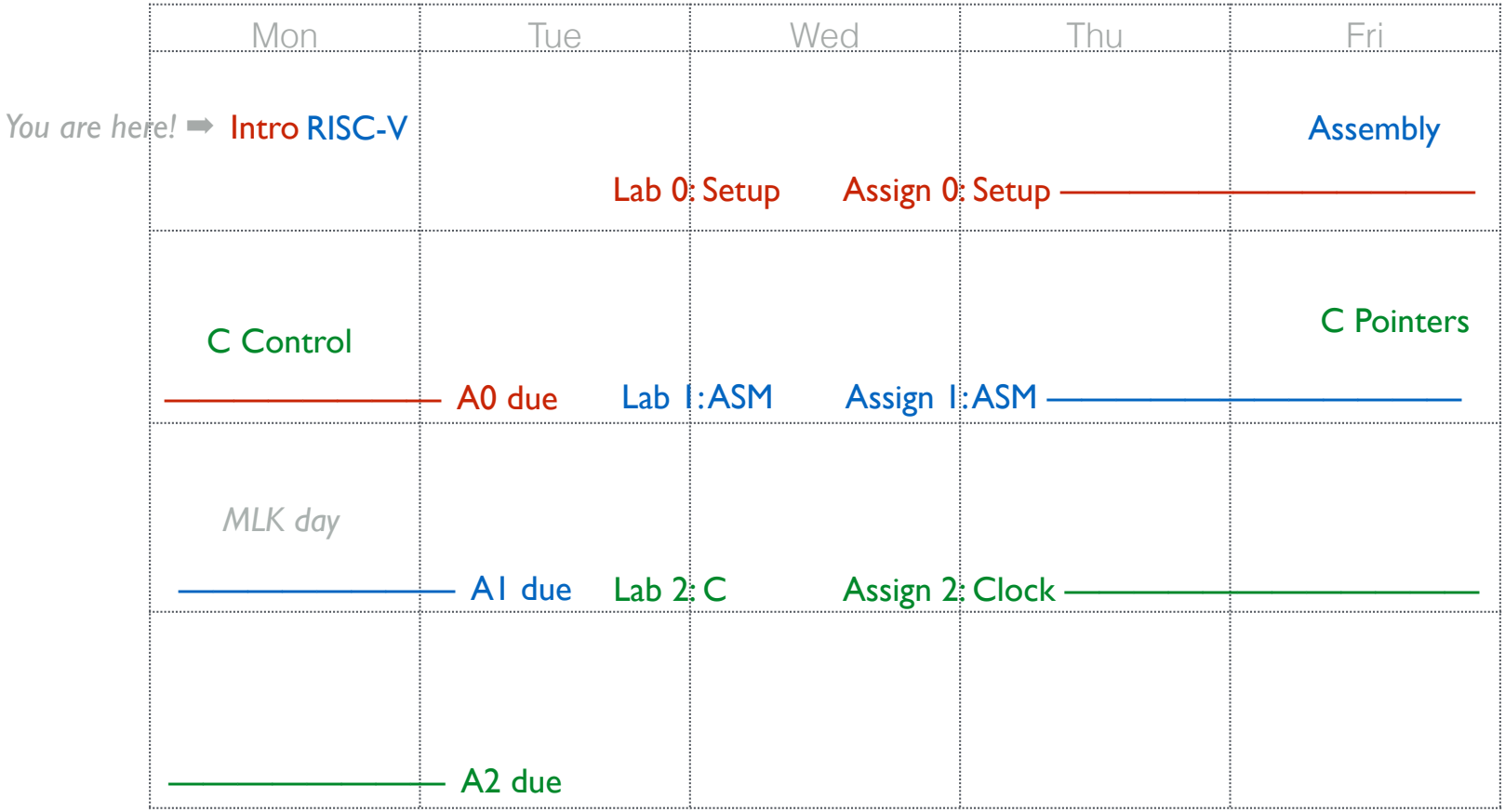
- Keyboard
- Graphics
- Interrupts



§3 Create Your Own Project

- Sensors
- Performance

Weekly Cadence



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

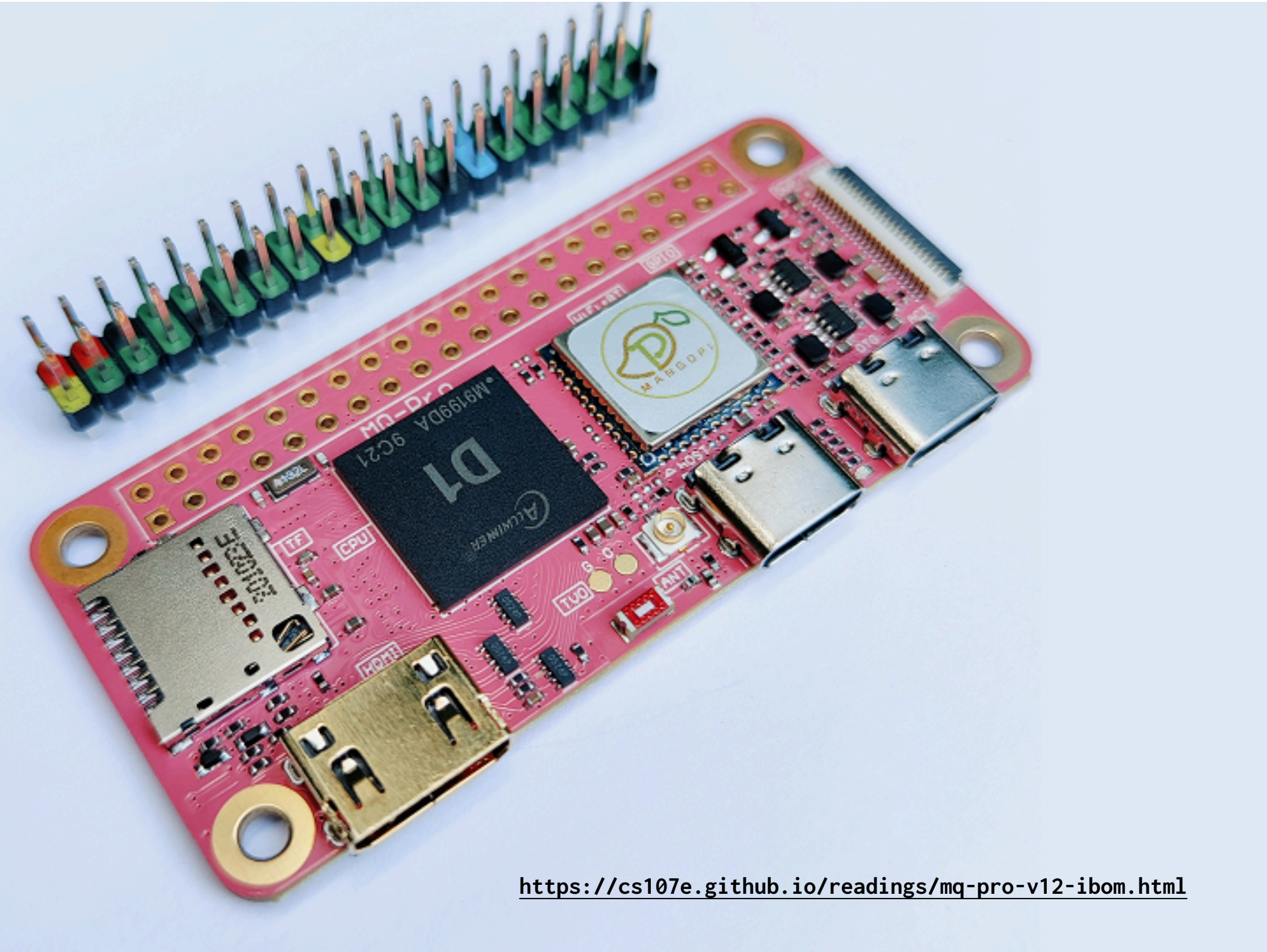
Each week has a focus **topic**

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

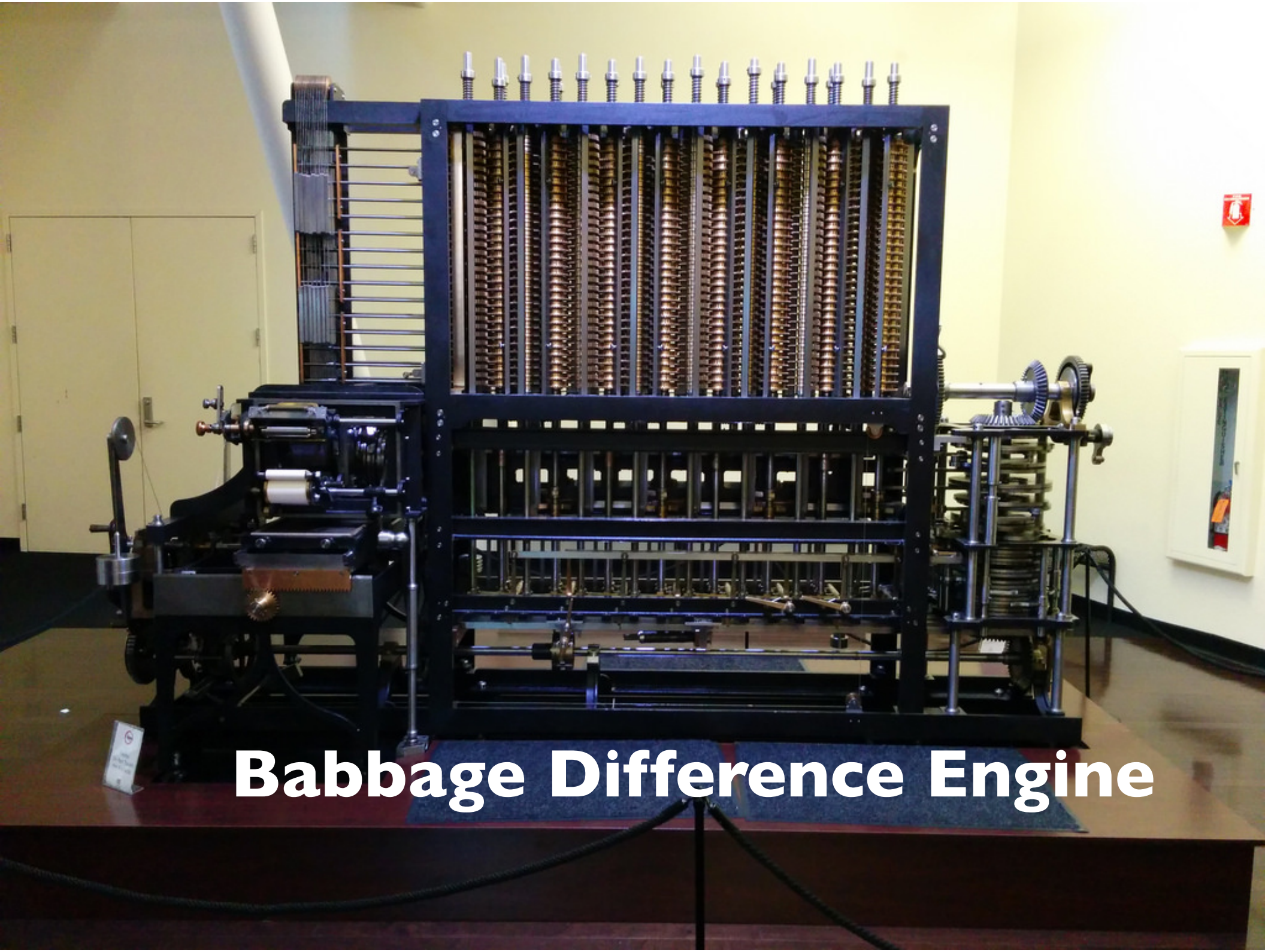
Lab on Tue/Wed evening

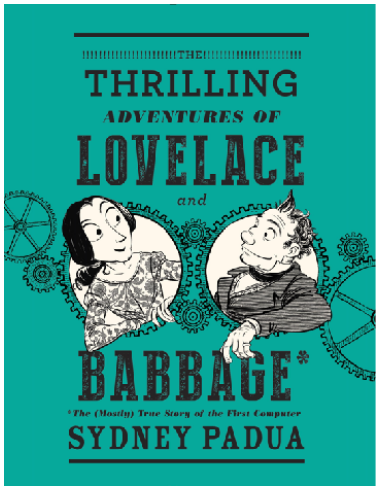
Assignment handed out Wed after lab, due following Tuesday 5pm

Staying on pace leads to best outcomes!



https://cs107e.github.io/readings/mq-pro-v12-ibom.html


Babbage Difference Engine



Analytical Engine

Universal Computer

http://www.theguardian.com/technology/2015/apr/12/thrilling-adventures-ada-lovelace-charles-babbage-sydney-padua

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

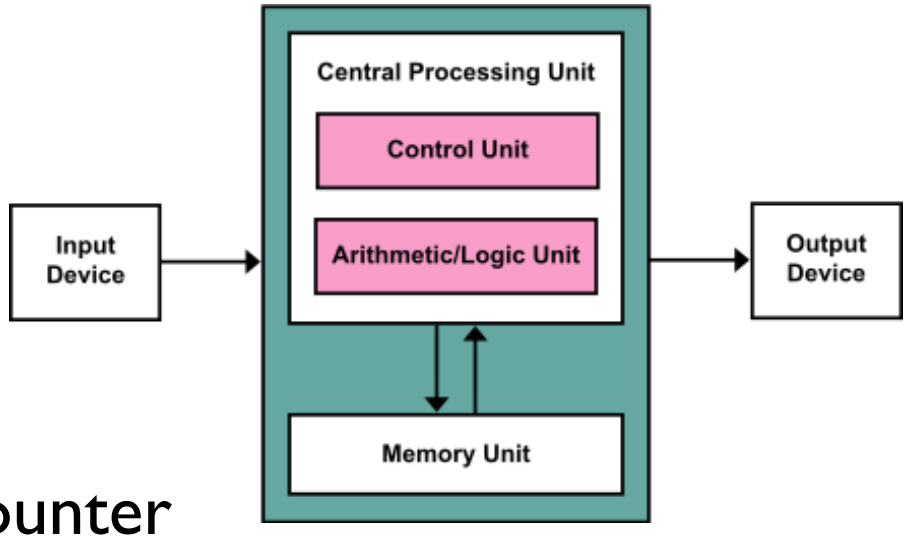


Image credit https://en.wikipedia.org/wiki/Von_Neumann_architecture#/media/File:Von_Neumann_Architecture.svg

Memory Map

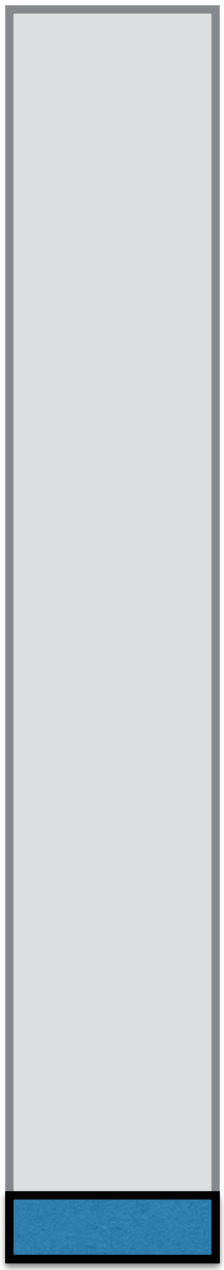
Memory is a large array

Storage location accessed using 64-bit index, called an *address*

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

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

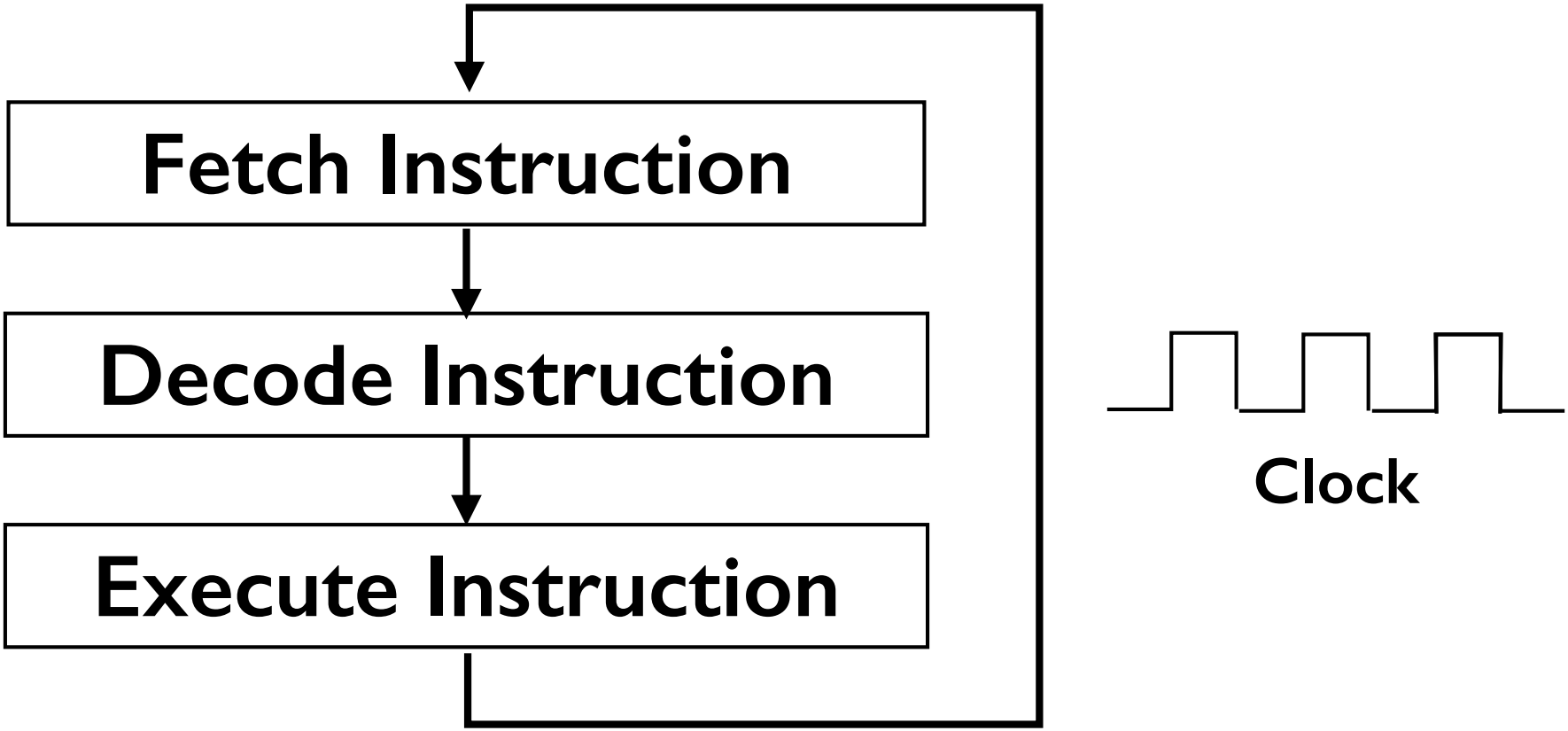
64-bit address space => total of 16EB addressable
(practical limit lower to due to address lines, physical RAM size)



2^{10} = 1024 = 1 KB
2^{20} = 1 MB
2^{30} = 1 GB
2^{32} = 4 GB
2^{64} = 16 EB

512MB physical memory

Running a Program



How to understand an ISA

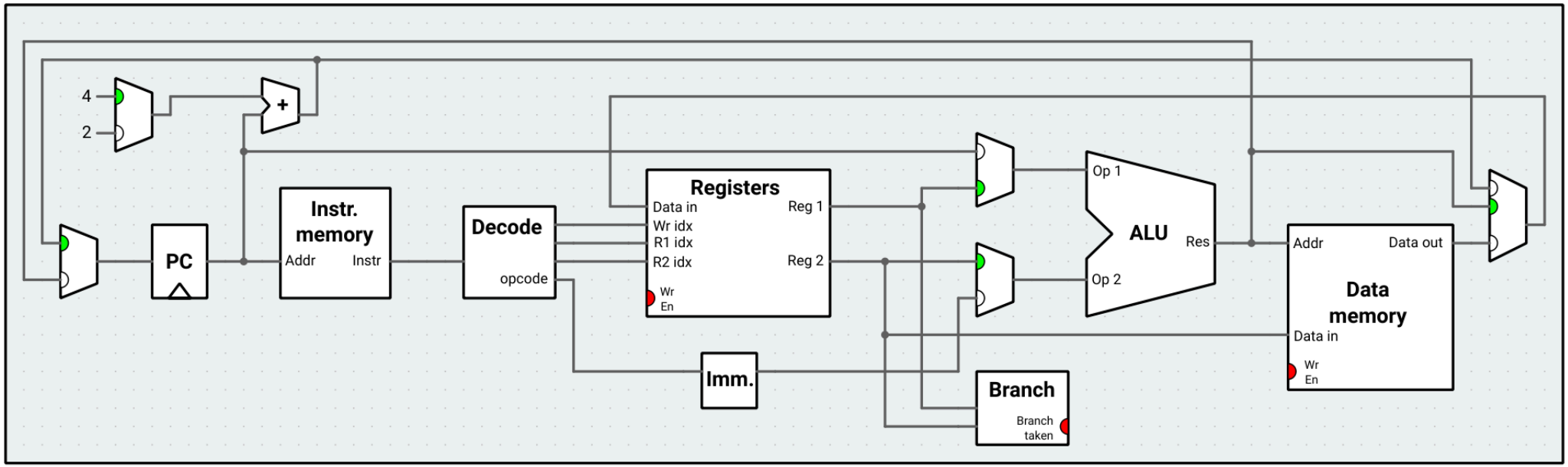
We want to learn how processors represent and execute instructions.

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

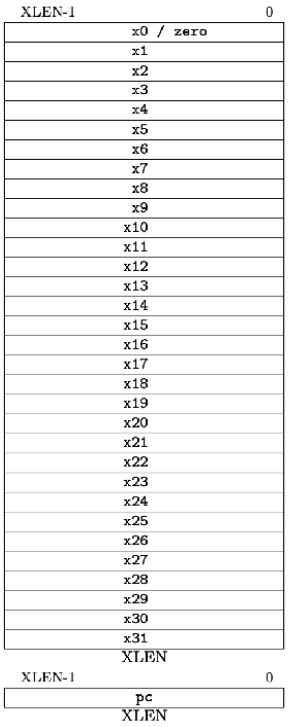
Tracing that paths shows where information can flow from/to and how that dictates the operational behavior

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

RISC-V Architecture / Floor Plan



https://ripes.me/



32 registers (x0-x31)

PC = program counter (address of insn to execute)

RISC-V Instruction-Set

Erik Engheim <erik.engheim@ma.com>

Arithmetic Operation

Mnemonic	Instruction	Type	Description
ADD rd, rs1, rs2	Add	R	rd ← rs1 + rs2
SUB rd, rs1, rs2	Subtract	R	rd ← rs1 - rs2
ADDI rd, rs1, imm12	Add immediate	I	rd ← rs1 + imm12
SLT rd, rs1, rs2	Set less than	R	rd ← rs1 < rs2 ? 1 : 0
SLTI rd, rs1, imm12	Set less than immediate	I	rd ← rs1 < imm12 ? 1 : 0
SLTU rd, rs1, rs2	Set less than unsigned	R	rd ← rs1 < rs2 ? 1 : 0
SLTIU rd, rs1, imm12	Set less than immediate unsigned	I	rd ← rs1 < imm12 ? 1 : 0
LUI rd, imm20	Load upper immediate	U	rd ← imm20 << 12
AUIP rd, imm20	Add upper immediate to PC	U	rd ← PC + imm20 << 12

Logical Operations

Mnemonic	Instruction	Type	Description
AND rd, rs1, rs2	AND	R	rd ← rs1 & rs2
OR rd, rs1, rs2	OR	R	rd ← rs1 | rs2
XOR rd, rs1, rs2	XOR	R	rd ← rs1 ^ rs2
ANDI rd, rs1, imm12	AND immediate	I	rd ← rs1 & imm12
ORI rd, rs1, imm12	OR immediate	I	rd ← rs1 | imm12
XORI rd, rs1, imm12	XOR immediate	I	rd ← rs1 ^ imm12
SLL rd, rs1, rs2	Shift left logical	R	rd ← rs1 << rs2
SRL rd, rs1, rs2	Shift right logical	R	rd ← rs1 >> rs2
SRA rd, rs1, rs2	Shift right arithmetic	R	rd ← rs1 >> rs2
SLLI rd, rs1, shamt	Shift left logical immediate	I	rd ← rs1 << shamt
SRLI rd, rs1, shamt	Shift right logical imm.	I	rd ← rs1 >> shamt
SRAI rd, rs1, shamt	Shift right arithmetic immediate	I	rd ← rs1 >> shamt

Load / Store Operations

Mnemonic	Instruction	Type	Description
LD rd, imm12(rs1)	Load doubleword	I	rd ← mem[rs1 + imm12]
LW rd, imm12(rs1)	Load word	I	rd ← mem[rs1 + imm12]
LH rd, imm12(rs1)	Load halfword	I	rd ← mem[rs1 + imm12]
LB rd, imm12(rs1)	Load byte	I	rd ← mem[rs1 + imm12]
LWU rd, imm12(rs1)	Load word unsigned	I	rd ← mem[rs1 + imm12]
LHU rd, imm12(rs1)	Load halfword unsigned	I	rd ← mem[rs1 + imm12]
LBU rd, imm12(rs1)	Load byte unsigned	I	rd ← mem[rs1 + imm12]
SD rs2, imm12(rs1)	Store doubleword	S	rs2 → mem[rs1 + imm12]
SW rs2, imm12(rs1)	Store word	S	rs2(31:0) → mem[rs1 + imm12]
SH rs2, imm12(rs1)	Store halfword	S	rs2(15:0) → mem[rs1 + imm12]
SB rs2, imm12(rs1)	Store byte	S	rs2(7:0) → mem[rs1 + imm12]

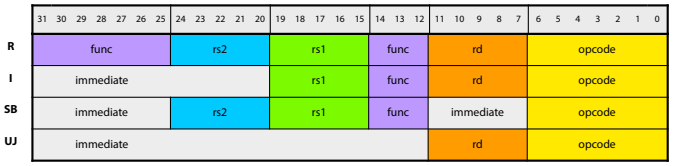
Branching

Mnemonic	Instruction	Type	Description
BEQ rs1, rs2, imm12	Branch equal	SB	if rs1 = rs2 pc ← pc + imm12
BNE rs1, rs2, imm12	Branch not equal	SB	if rs1 ≠ rs2 pc ← pc + imm12
BGE rs1, rs2, imm12	Branch greater than or equal	SB	if rs1 ≥ rs2 pc ← pc + imm12
BGEU rs1, rs2, imm12	Branch greater than or equal unsigned	SB	if rs1 >= rs2 pc ← pc + imm12
BLT rs1, rs2, imm12	Branch less than	SB	if rs1 < rs2 pc ← pc + imm12
BLTU rs1, rs2, imm12	Branch less than unsigned	SB	if rs1 < rs2 pc ← pc + imm12 << 1
JAL rd, imm20	Jump and link	UJ	rd ← pc + 4 pc ← pc + imm20
JALR rd, imm12(rs1)	Jump and link register	I	rd ← pc + 4 pc ← rs1 + imm12

Pseudo Instructions

Mnemonic	Instruction	Base instruction(s)
LI rd, imm12	Load immediate (near)	ADDI rd, zero, imm12
LI rd, imm	Load immediate (far)	LUI rd, imm[31:12] ADDI rd, rd, imm[11:0]
LA rd, sym	Load address (far)	AUIPC rd, sym[31:12] ADDI rd, rd, sym[11:0]
MV rd, rs	Copy register	ADDI rd, rs, 0
NOT rd, rs	One's complement	XORI rd, rs, -1
NEG rd, rs	Two's complement	SUB rd, zero, rs
BGT rs1, rs2, offset	Branch if rs1 > rs2	BLT rs2, rs1, offset
BLE rs1, rs2, offset	Branch if rs1 ≤ rs2	BGE rs2, rs1, offset
BGTU rs1, rs2, offset	Branch if rs1 > rs2 (unsigned)	BLTU rs2, rs1, offset
BLEU rs1, rs2, offset	Branch if rs1 ≤ rs2 (unsigned)	BGEU rs2, rs1, offset
BEQZ rs1, offset	Branch if rs1 = 0	BEQ rs1, zero, offset
BNEZ rs1, offset	Branch if rs1 ≠ 0	BNE rs1, zero, offset
BGEZ rs1, offset	Branch if rs1 ≥ 0	BGE rs1, zero, offset
BLEZ rs1, offset	Branch if rs1 ≤ 0	BGE zero, rs1, offset
BGTZ rs1, offset	Branch if rs1 > 0	BLT zero, rs1, offset
J offset	Unconditional jump	JAL zero, offset
CALL offset12	Call subroutine (near)	JALR ra, ra, offset12
CALL offset	Call subroutine (far)	AUIPC ra, offset[31:12] JALR ra, ra, offset[11:0]
RET	Return from subroutine	JALR zero, 0(ra)
NOP	No operation	ADDI zero, zero, 0

32-bit instruction format



Register File

r0	r1	r2	r3
r4	r5	r6	r7
r8	r9	r10	r11
r12	r13	r14	r15
r16	r17	r18	r19
r20	r21	r22	r23
r24	r25	r26	r27
r28	r29	r30	r31

Register Aliases

zero	ra	sp	gp
tp	t0	t1	t2
s0/fp	s1	a0	a1
a2	a3	a4	a5
a6	a7	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
t3	t4	t5	t6

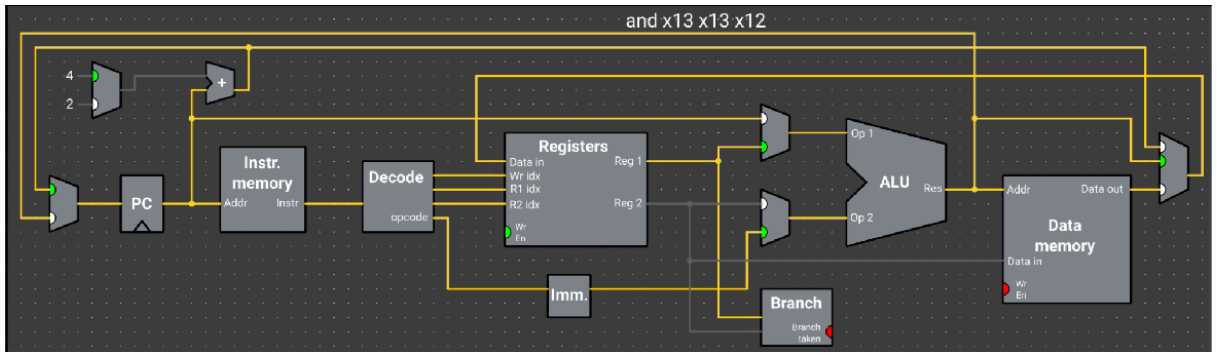
ra - return address
sp - stack pointer
gp - global pointer
tp - thread pointer

t0 - t6 - Temporary registers
s0 - s11 - Saved by callee
a0 - a7 - Function arguments
a0 - a1 - Return value(s)

On course website at https://cs107e.github.io/guides/riscv-onepage/

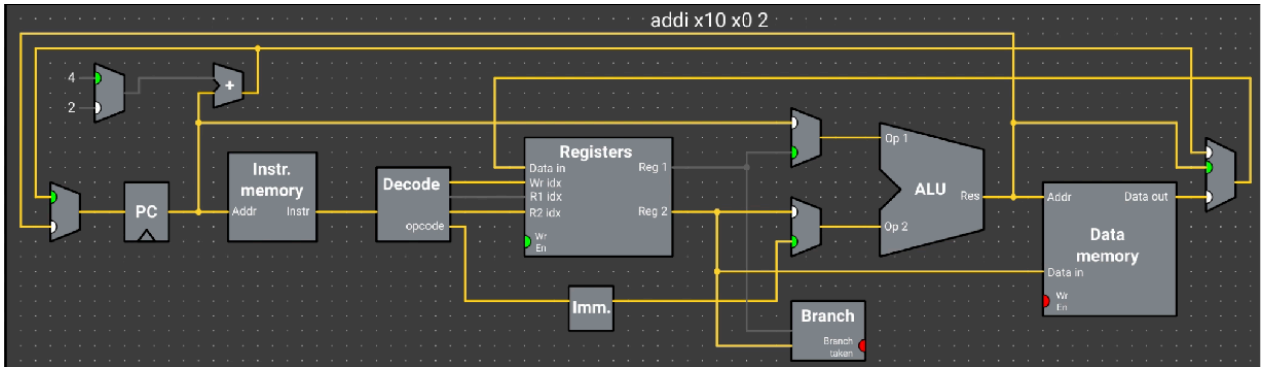
Data paths dictate constraints on operations

```
and a3,a3,a2
```



What are possible inputs to ALU?
Where does result go?

```
addi a2,zero,2
```



Where does an immediate value come from?
Where can it flow to?

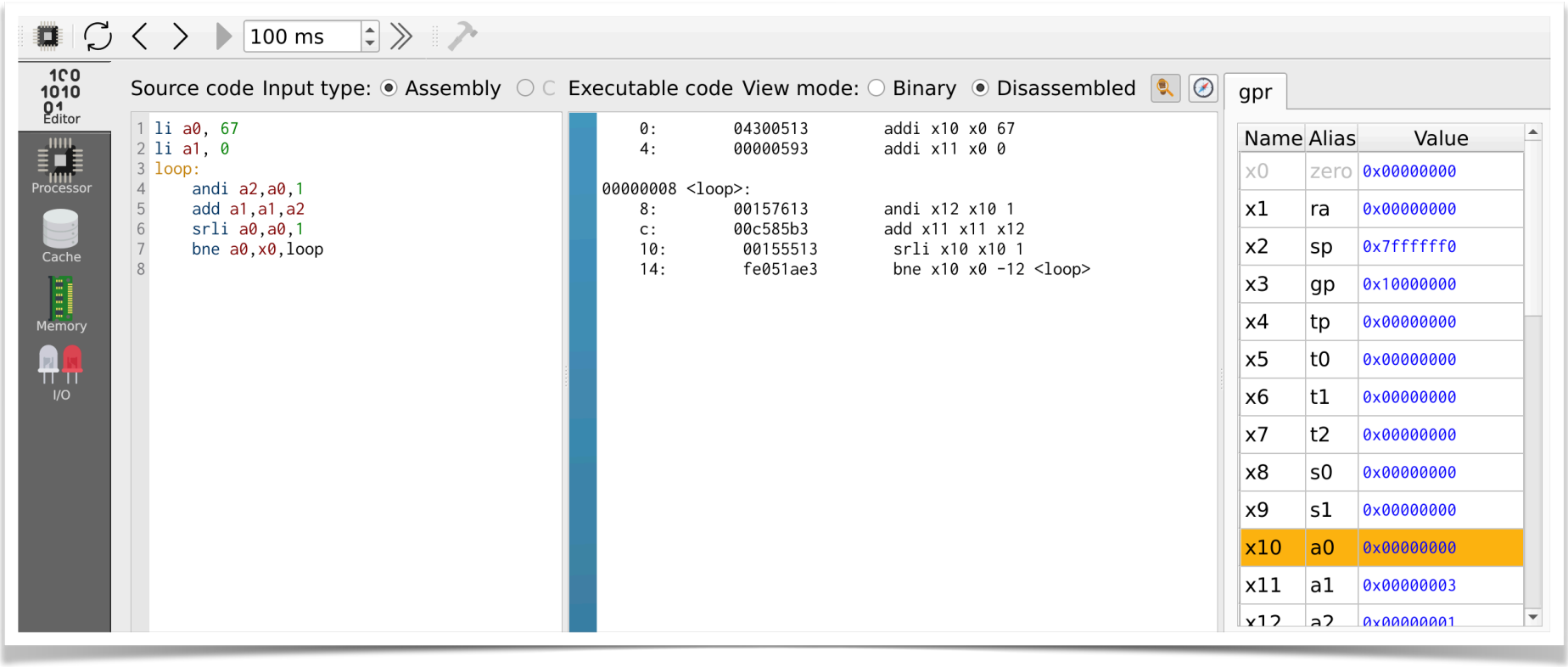
Challenge for you all:

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

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, branch, load/store

Resources to keep handy

RISC-V one-page guide

Ripes simulator

Lab this week

Pre-lab (complete before lab):
- Review course guides:
  - Unix tools (shell, editor, git)
  - Electricity (simple circuits, Ohm's law)
  - Number representation (binary, bit operations)
- Windows users install WSL `https://cs107e.github.io/guides/install/wsl-setup/`

During lab:
- Install development tools
- Practice with environment, start building productive habits
- Establish comfort with background topics
- Meet one another!