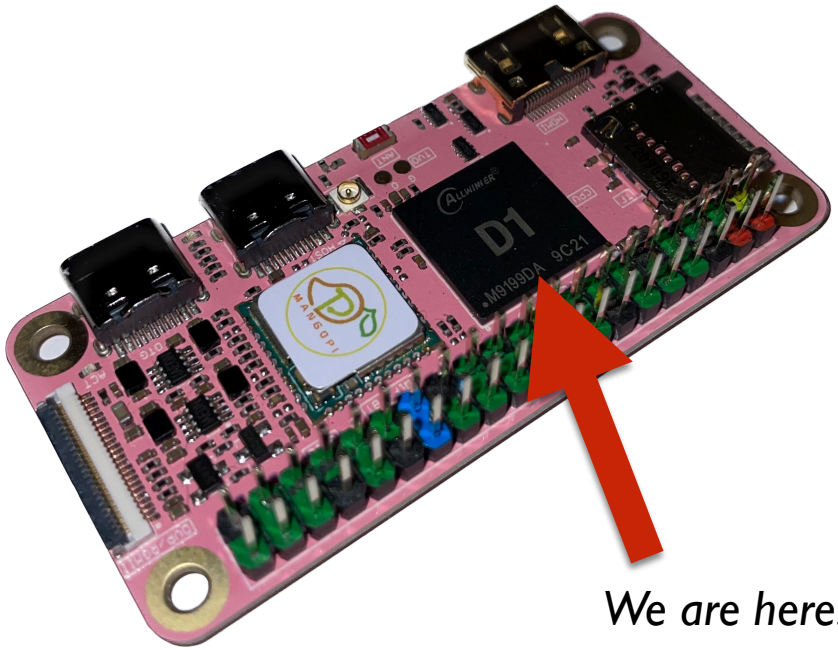
Admin

Lab0 ✅
Assign0
Ed forum
OH coming soon to calendar



Today: RISC-V ISA

Historical milestones in computer architecture

Origins of RISC-V

RISC-V instruction set architecture

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

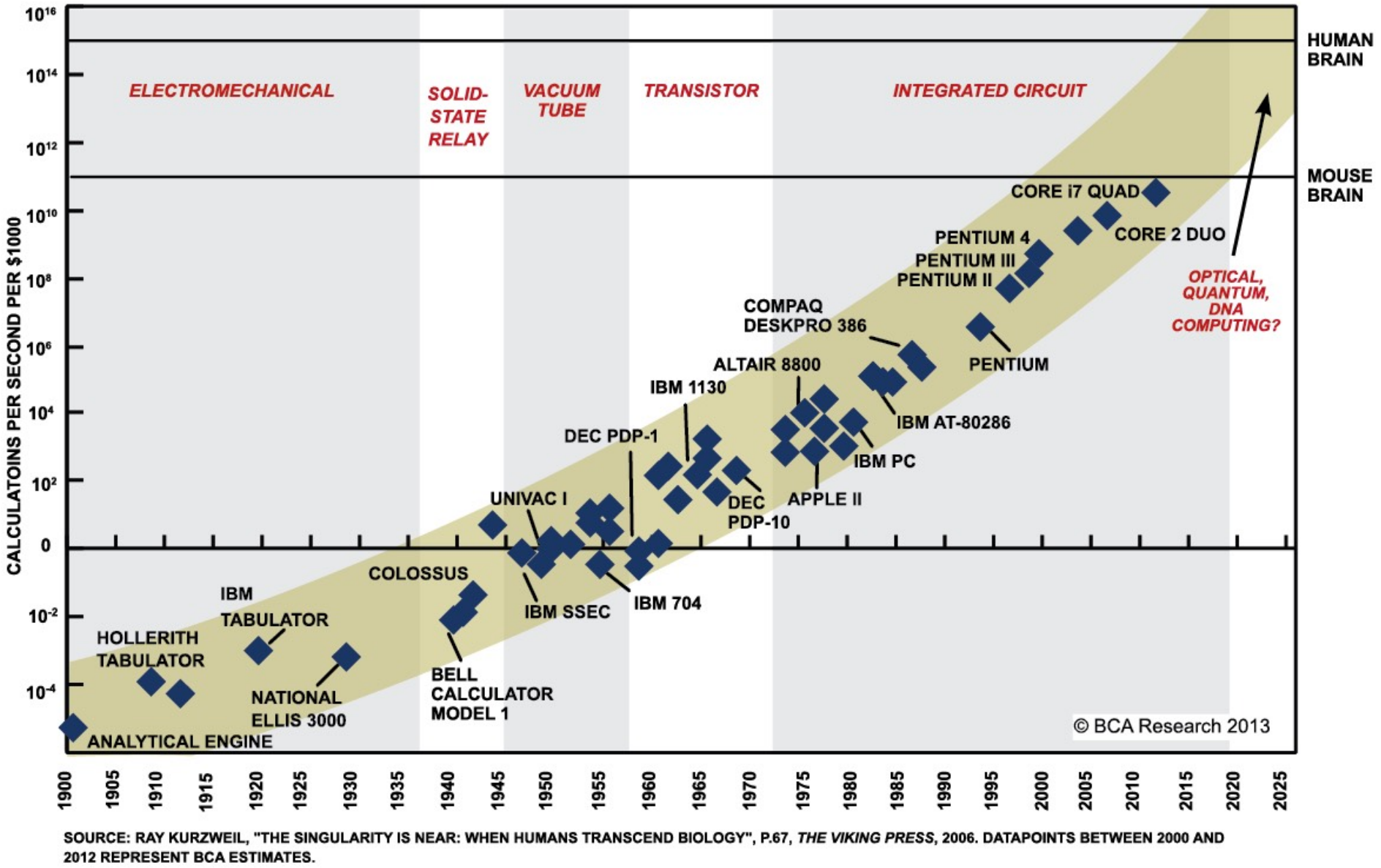
Personal desktop computer

- 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;
}
```

Size of entire ISA

x86

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

1500 instr, 5000 pp

ARM

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

500 instr, 1200 pp

rv

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

40 instr, 220 pp

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.

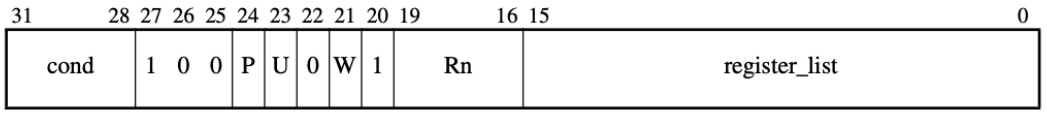
x86 puts the C in CISC

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

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

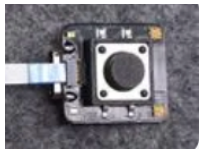
- Externally visible aspects
  - registers, instructions, data types, control/exceptions
- Huge value of standard, no IP hurdles

Microarchitecture

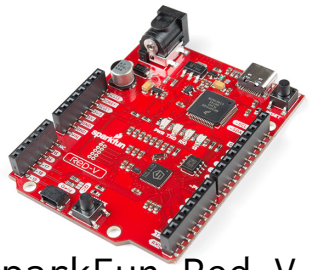
- Implementation of ISA
  - Logic design, power/performance tradeoffs
- Room for innovation, competition
  - e.g. Intel & AMD both implement x86, but very different microarchitectures

Abstraction for the win!

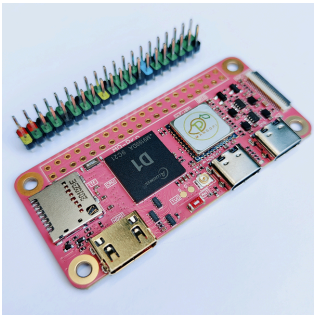
RISC-V today and tomorrow



Picoclick C3T
(IOT button)

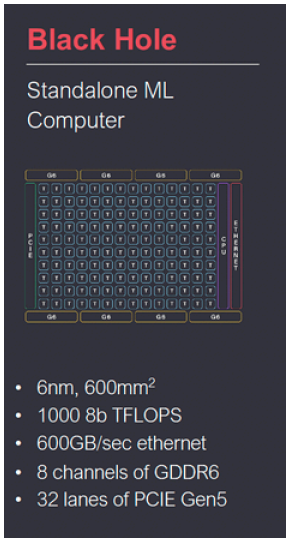


SparkFun Red-V
(Arduino/hobbyist)



MangoPi SBC
(~Rpi)

HiFive Unmatched
(linux desktop)



Tenstorrent
(HPC AI/ML)

Big players paying attention !

MIPS eVocore
Samsung/LG smart TVs
Intel Horse Creek
Qualcomm+Android wearables ...

Open software/standards success stories

IEEE 754, Ethernet, Linux, SQL, FSF, OpenGL, ...

What will be effect of open ISA?

More: collaboration, competition, innovation

How to understand an ISA

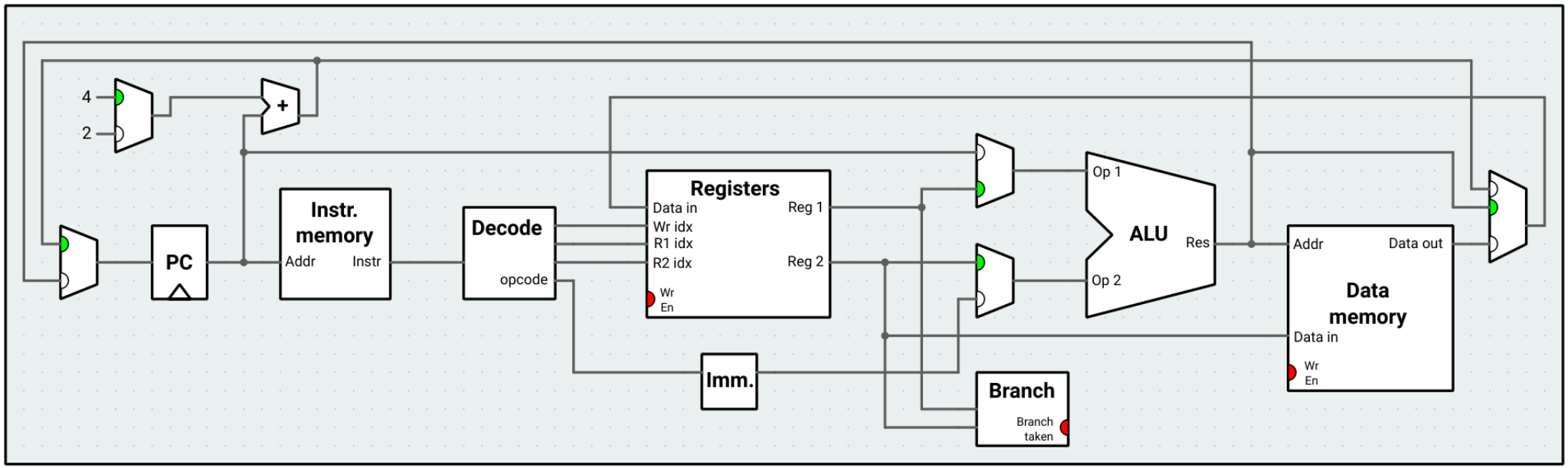
We want to learn how processors represent and execute instructions.

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

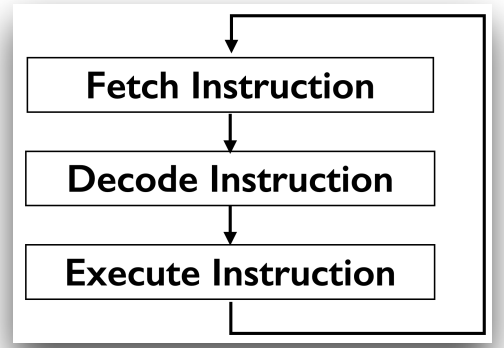
Tracing that paths shows where information can flow from/to and how that dictates the operational behavior

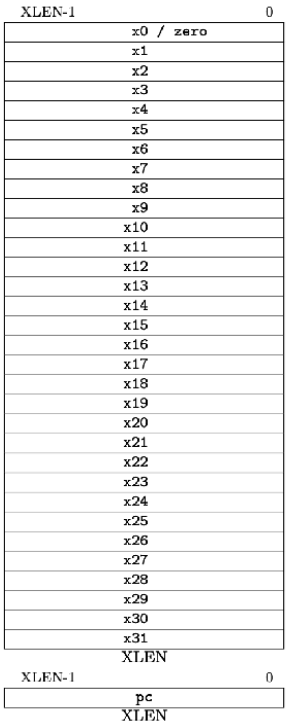
Visualizer/simulator can be helpful! `https://ripes.me`

RISC-V Architecture / Floor Plan



https://ripes.me/



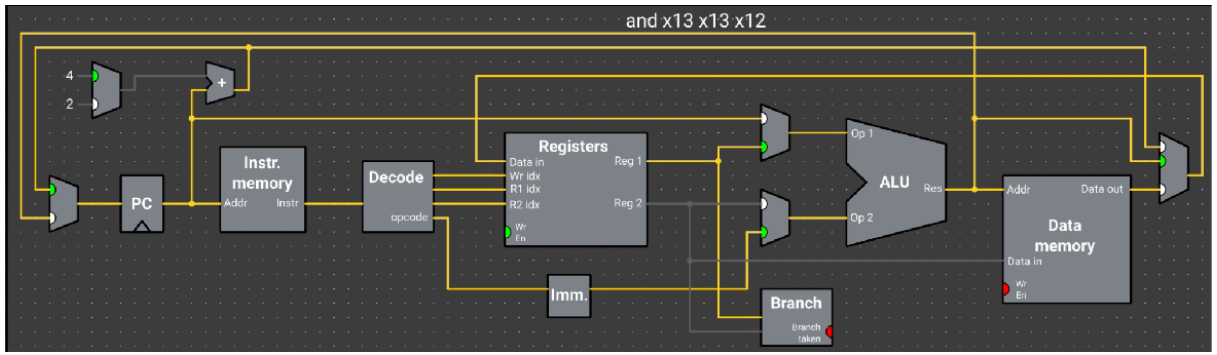


32 registers (x0-x31)

PC = program counter

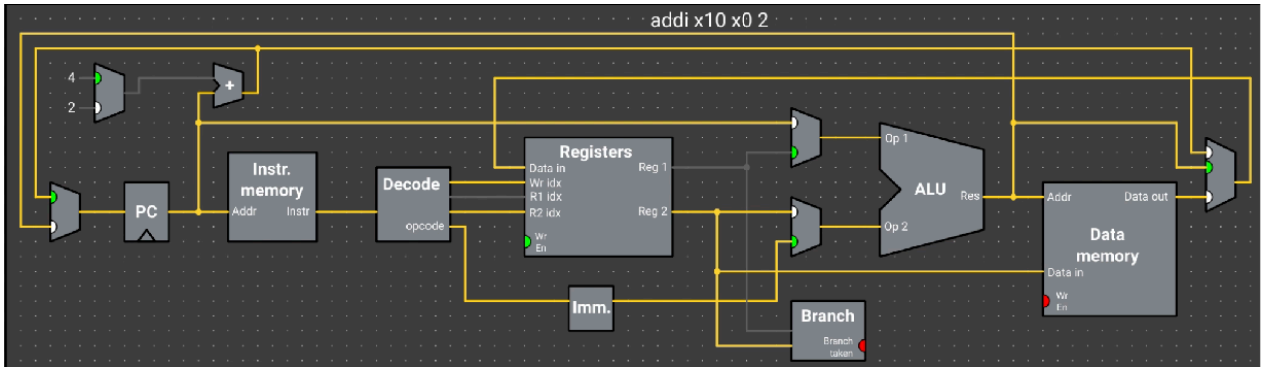
Data paths == operational constraints

```
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?

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/

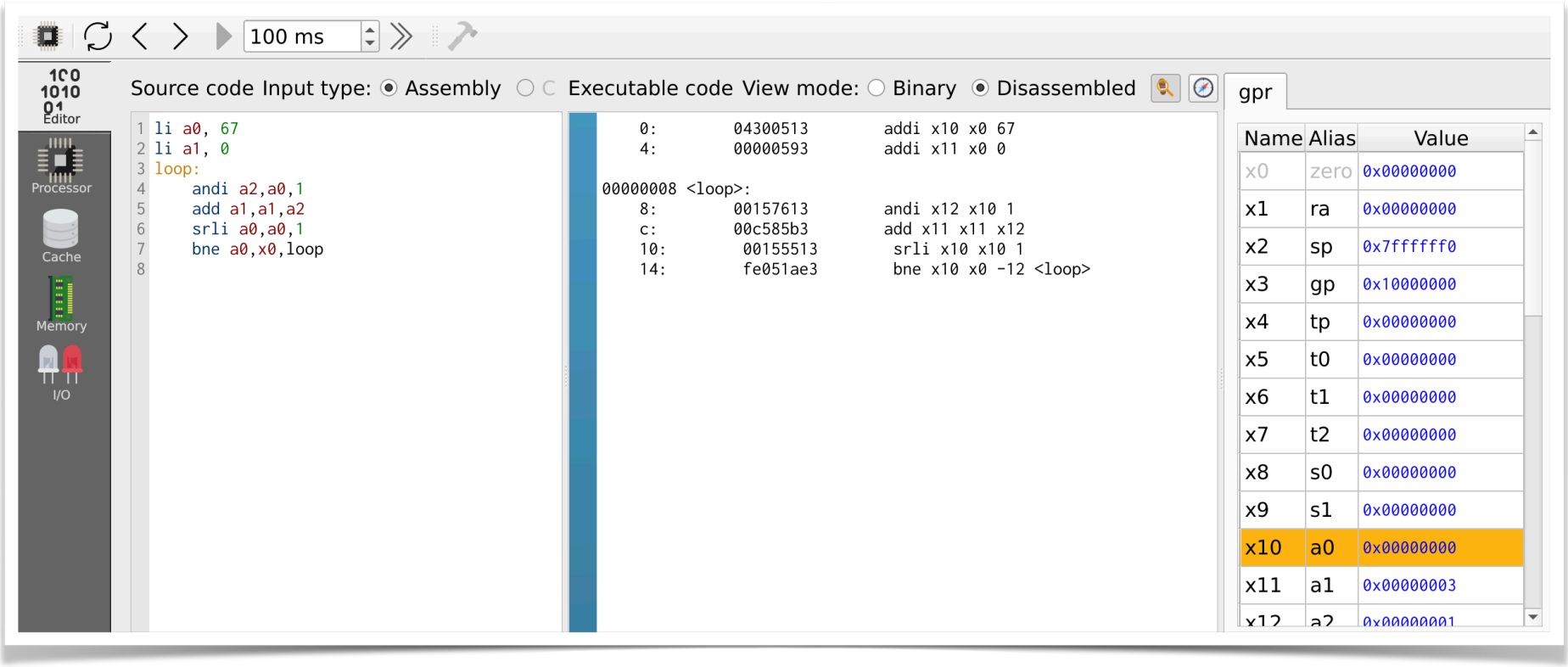
Challenge for you all:

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

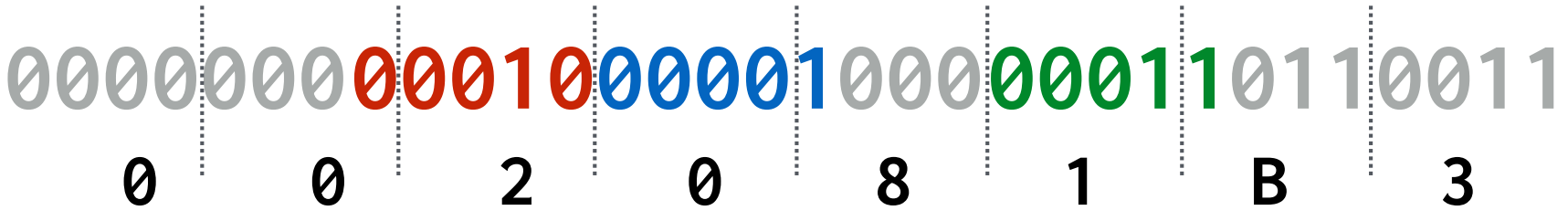
Sparse instruction encoding (room for growth)

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



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