# Toy Instruction Set Architecture

## CS 2130: Computer Systems and Organization 1

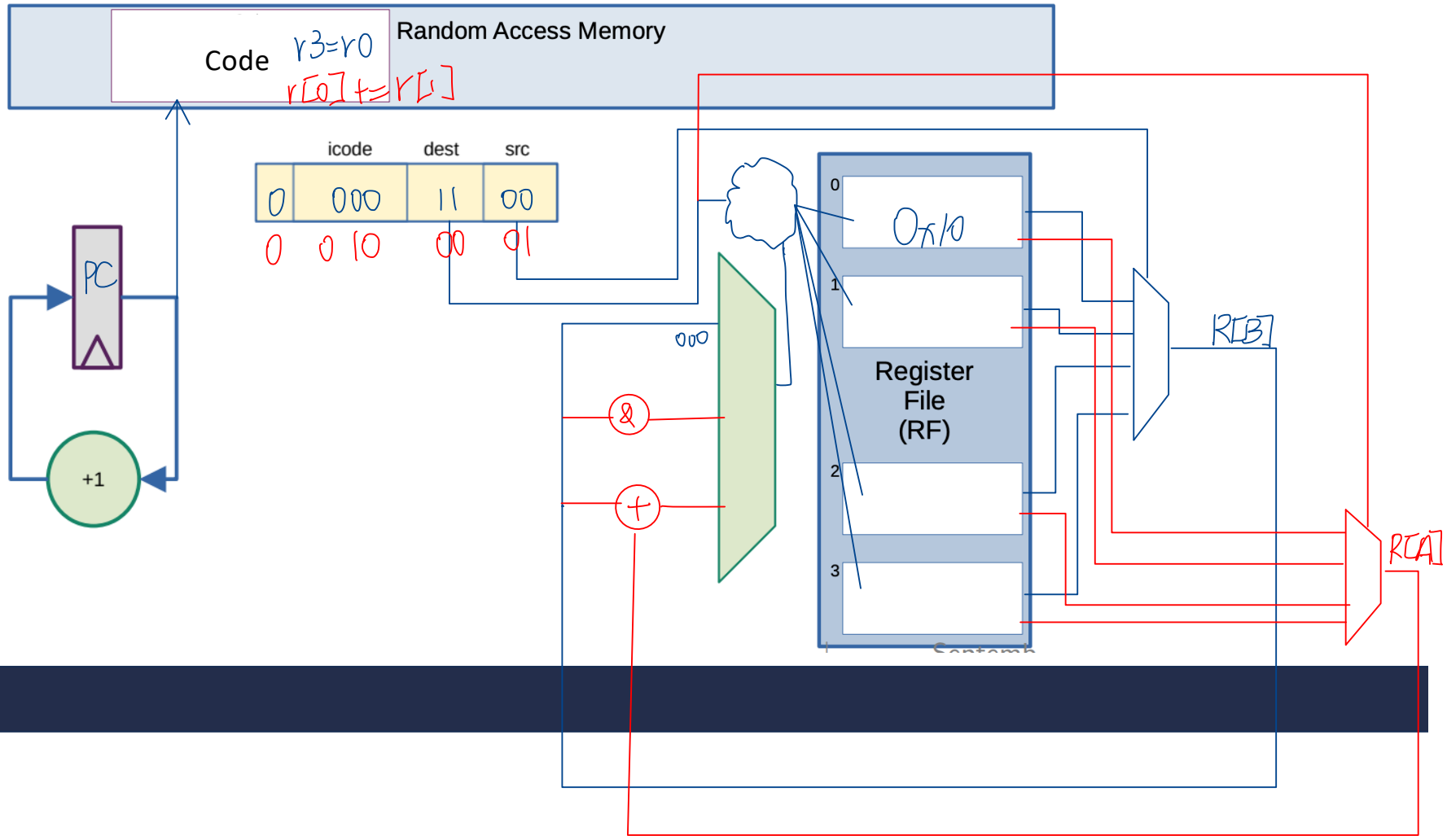
**Xinyao Yi** Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

## Announcements

- Homework 2 due tonight at 11:59pm on Gradescope
- Homework 3 out today, due next Monday at 11:59pm on Gradescope

# Building a Computer



# High-level Instructions

In general, 3 kinds of instructions

- **moves** - move values around without doing "work"
- **math** - broadly doing "work"
- **jumps** - jump to a new place in the code

# Moves

Few forms

- Register to register (icode 0), x = y (primitive variables)
- Register to/from memory (icodes 4-5), x = M[b], M[b] = x (objects or arrays)

or, more details:
R0 = M[R[2]]

Memory

- Address: an index into memory.
  - Addresses are just (large) numbers (maybe 256 bytes?)
  - Usually we will not look at the number and trust it exists and is stored in a register

① go to memory, look at the value of b
② use that value as the index of my big array in memory. (memory is a big array of bytes)

# Moves

| icode | b | action |
|---|---|---|
| 0 | | rA = rB |
| 3 | 3 | rA = pc (next instruction) do things with function calls. |
| 4 | | rA = read from memory at address rB |
| 5 | | write rA to memory at address rB |
| 6 | 0 | rA = read from memory at pc + 1 |
| | 3 | rA = read from memory at the address stored at pc + 1 |

# Math

Broadly doing work

| icode | b | meaning |
|---|---|---|
| 1 | | `rA &= rB` |
| 2 | | `rA += rB` |
| 3 | 0 | `rA = ~rA` flip bits |
| | 1 | `rA = !rA` logical not |
| | 2 | `rA = -rA` take the negation |
| 6 | 1 | `rA &=` read from memory at pc + 1 |
| | 2 | `rA +=` read from memory at pc + 1 |

basic → more complex:

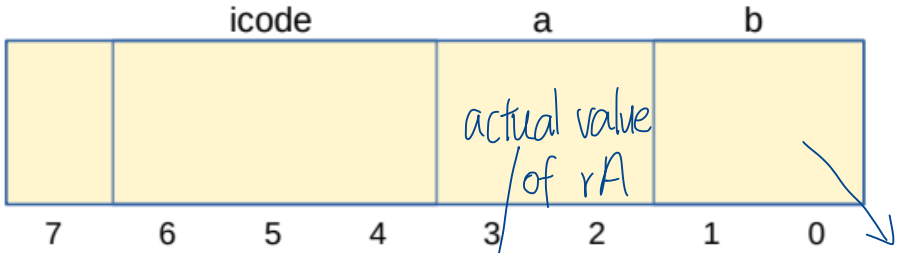
① . substraction: take one value, negating it and adding it to another.

② . multiplication: repeated addtion

*Note: We can implement other operations using these things!*

# icodes 3 and 6

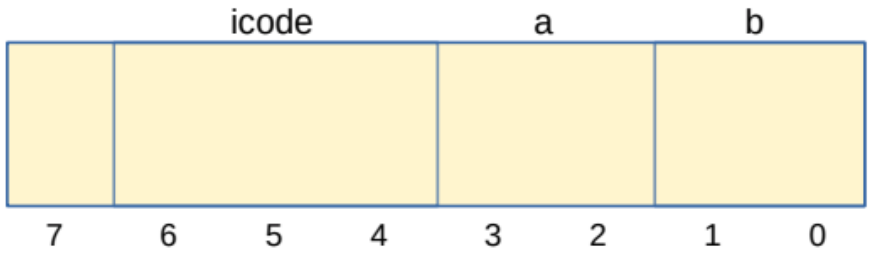
Special property of icodes 3 & 6: only one register used




| icode | b | action |
|---|---|---|
| 3 | 0 | rA = ~rA |
| | 1 | rA = !rA |
| | 2 | rA = -rA |
| | 3 | rA = pc |

# icodes 3 and 6

Special property of icodes 3 & 6: only one register used




- Side effect: all bytes between 0 and 127 are valid instructions!
- As long as high-order bit is 0
- No syntax errors, any instruction given is valid

# Immediate values

icode 6 provides literals, **immediate** values — a literal constant written directly inside the instruction, instead of being fetched from memory later.

| icode | b | action |
|---|---|---|
| 6 | 0 | rA = read from memory at pc + 1 → plus a byte |
| | 1 | rA &= read from memory at pc + 1 |
| | 2 | rA += read from memory at pc + 1 |
| | 3 | rA = read from memory at the address stored at pc + 1 |
| | | For icode 6, increase pc by 2 at end of instruction |

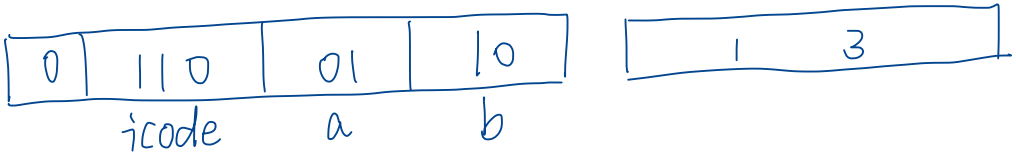
r0 = 05 :


I've used all 8 bits ⟹ just put the value in the next byte.

# Encoding Instructions

Example 1: r1 += 19 → in decimal

19 in hexdecimal: 0x13







hex: 0b13

# Instructions

| icode | b | meaning |
|---|---|---|
| 0 | | `rA = rB` |
| 1 | | `rA &= rB` |
| 2 | | `rA += rB` 2 registers |
| 3 | 0 | `rA = ~rA` |
| | 1 | `rA = !rA` |
| | 2 | `rA = -rA` |
| | 3 | `rA = pc` |
| 4 | | `rA` = read from memory at address `rB` |
| 5 | | write `rA` to memory at address `rB` |
| 6 | 0 | `rA` = read from memory at `pc + 1` |
| | 1 | `rA &=` read from memory at `pc + 1` |
| | 2 | `rA +=` read from memory at `pc + 1` immediate value. |
| | 3 | `rA` = read from memory at the address stored at `pc + 1` |
| | | For icode 6, increase `pc` by 2 at end of instruction |
| 7 | | Compare `rA` as 8-bit 2's-complement to 0 |
| | | if `rA <= 0` set `pc = rB` |
| | | else increment `pc` as normal |

# Encoding Instructions

idea: ①. I have a value in memory at address hex 82

Example 2: M[0x82] += r3 ②. I want to add whatever in R3 to that value.

Read memory at address 0x82, add r3, write back to memory at same address

One point: No instructions allow us to pass an immediate value as the address.
So let's save ourselves some time: just put 82 in a register.
Then we use icode 4 to read it out, icode 5 to write it back.

| | | |
|---|---|---|
| r2 = 0x82 | 0 110 \| 10 00 (6 \| 8) | 82 |
| r1 = M[r2] | 0 100 \| 01 10 (4 \| 6) | |
| r1 += r3 | 0 010 \| 01 11 (2 \| 7) | |
| M[r2] = r1 | 0 101 \| 01 10 (5 \| 6) | |

our machine reads 0 and 1.
But, for us → easier to read → pairs of hex
68 82 46 27 56

One interesting finding: first hex is always our icode!