Toy Instruction Set Architecture

CS 2130: Computer Systems and Organization 1

September 26, 2025

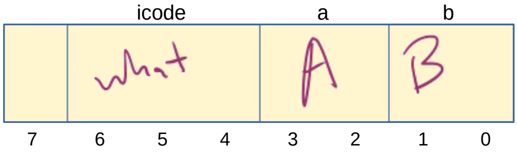
Announcements

- Homework 3 due ~~Monday~~ Wednesday at 11:59pm on Gradescope
- Quiz 4 available today, due Sunday at 11:59pm
- Midterm 1 next Friday (October 3, 2025) in class
  - Written, closed notes
  - If you have SDAC, please schedule ASAP

Encoding Instructions

Encoding of Instructions

- 3-bit icode (which operation to perform)
  - Numeric mapping from icode to operation
- Which registers to use (2 bits each)
- Reserved bit for future expansion



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

Jumps

- Moves and math are large portion of our code
- We also need **control constructs**
  - Change what we are going to do next
  - `if`, `while`, `for`, functions, …
- Jumps provide mechanism to perform these control constructs
- We jump by assigning a new value to the program counter `PC`

Jumps

if (r1 <= 0) jmp r0
A B

icode	meaning
7	Compare `rA` as 8-bit 2's-complement to `0` if `rA <= 0` set `pc = rB` else increment `pc` as normal

7 0100
4

Instruction icode 7 provides a **conditional** jump

- Real code will also provide an **unconditional** jump, but a conditional jump is sufficient

Writing Code

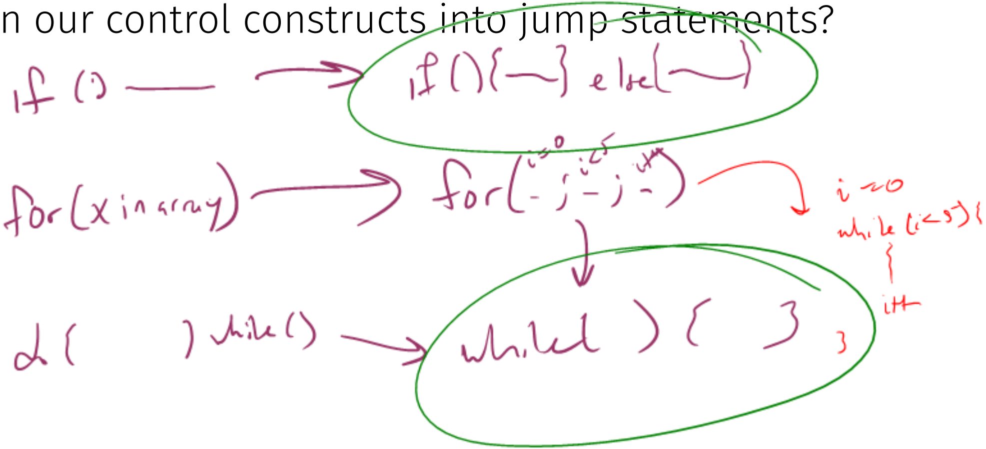
We can now write any* program!

- When you run code, it is being turned into instructions like ours
- Modern computers use a larger pool of instructions than we have (we will get there)

*we do have some limitations, since we can only represent 8-bit values and some operations may be tedious.

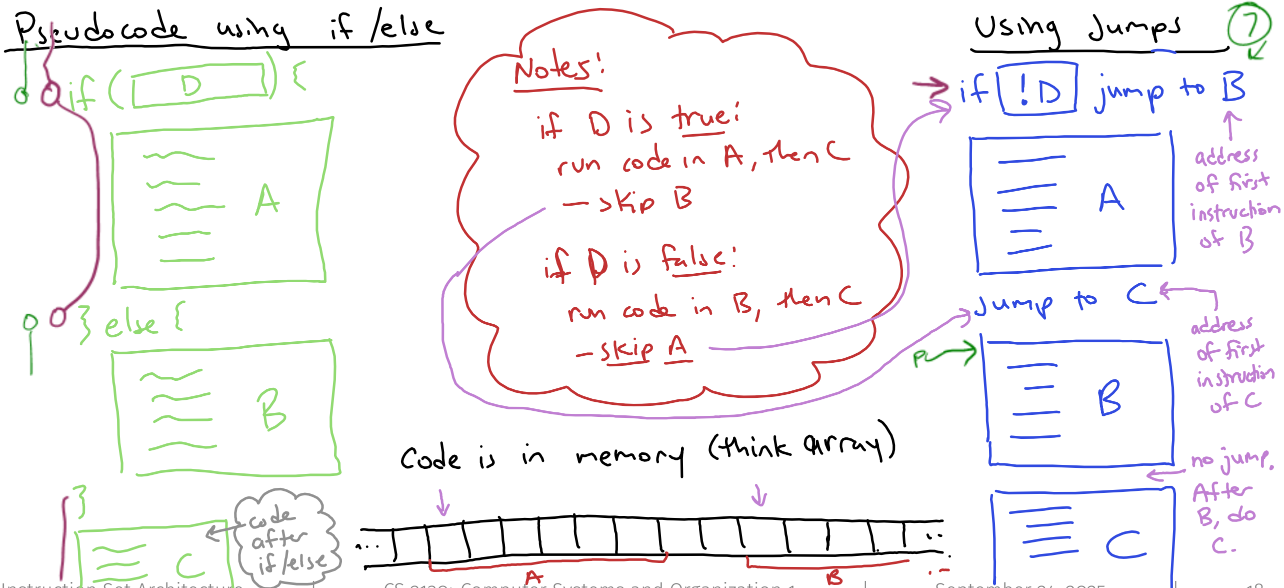
Our code to this machine code

How do we turn our control constructs into jump statements?

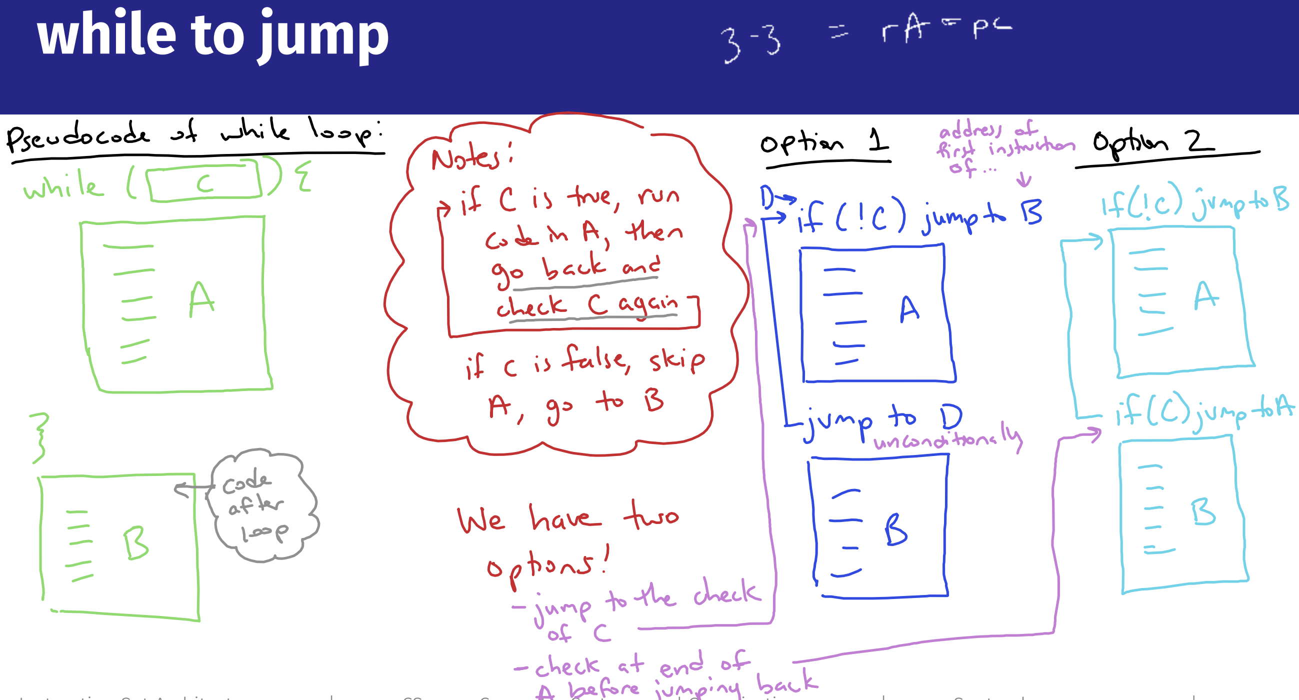


if/else to jump

if (rA <= 0) jmp ___

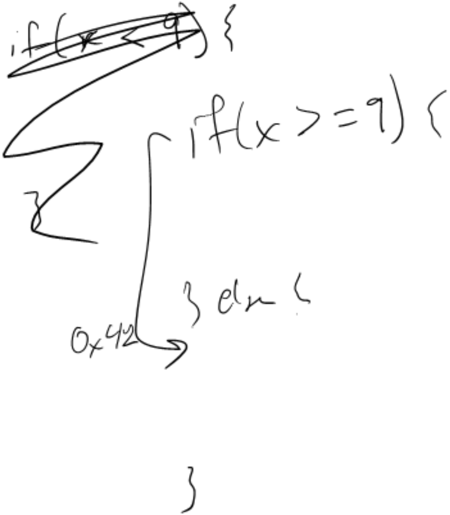


while to jump



Encoding Instructions

Example 3: `if r0 < 9 jump to 0x42`



Instructions

icode	b	meaning
0		`rA = rB`
1		`rA &= rB`
2		`rA += rB`
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`
	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

-128 --- 127

F8 ... FD FE FF 0

if (r0 < 9) jump to 0x42

r0 < 9

r0 <= 8

r0 - 8 <= 0

r0 += -8

r0 += F8

r1 = 0x42 ← 64 42

r0 += F8 ← 62 F8

if (r0 <= 0) pc = r1

71

64 42 62 F8 71

Memory

What kinds of things do we put in memory?

- Code: binary code like instructions in our example ISA
  - Intel/AMD compatible: x86_64
  - Apple Mx and Ax, ARM: ARM
  - And others!
- Variables: we may have more variables that will fit in registers
- Data Structures: organized data, collection of data
  - Arrays, lists, heaps, stacks, queues, ...

Dealing with Variables and Memory

What if we have many variables? Compute: `x += y`

x = 0x80
y = 0x81
z = 0x82
t = 0x83
w = 0x84
q = 0x85

read from mem
r1 = M[0x80] — 67 80
r2 = M[0x81] — 6B 81

execute
r1 += r2 — 26

write to mem
M[0x80] = r1 (r0 = 0x80, M[r0] = r1) — 60 80, 54
M[0x81] = r2 (r0 = 0x81, M[r0] = r2) — 60 81, 58

67 80 6B 81 26 60 80 54 60 81 58