Toy Instruction Set Architecture

CS 2130: Computer Systems and Organization 1

February 17, 2023

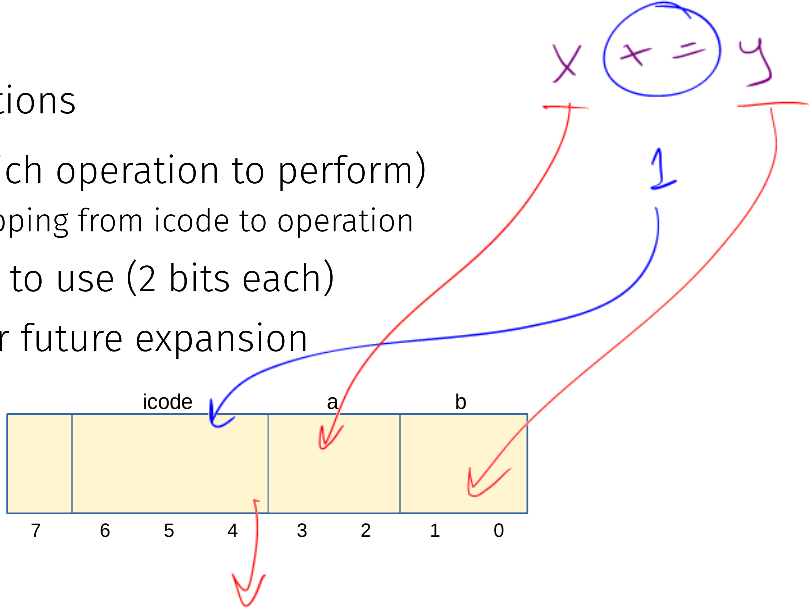
Announcements

- Homework 3 due **Wednesday** at 11pm on Gradescope
- Quiz 4 available today, due Sunday at 11:59pm (submit early)
- Exam 1 next Friday in class, Review on Wednesday

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



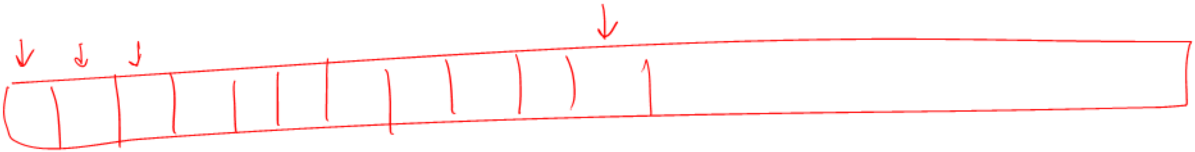
High-level Instructions

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

Toy ISA 3-bit icode

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

Instruction icode 7 provides a **conditional** jump

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

Writing Code

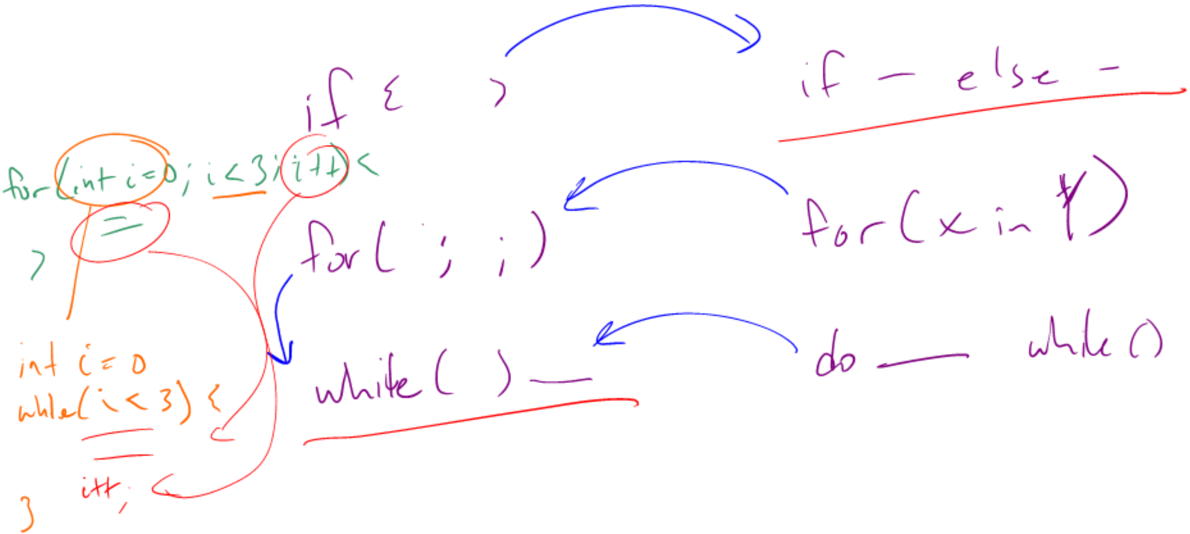
We can now write any* program!

- 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)

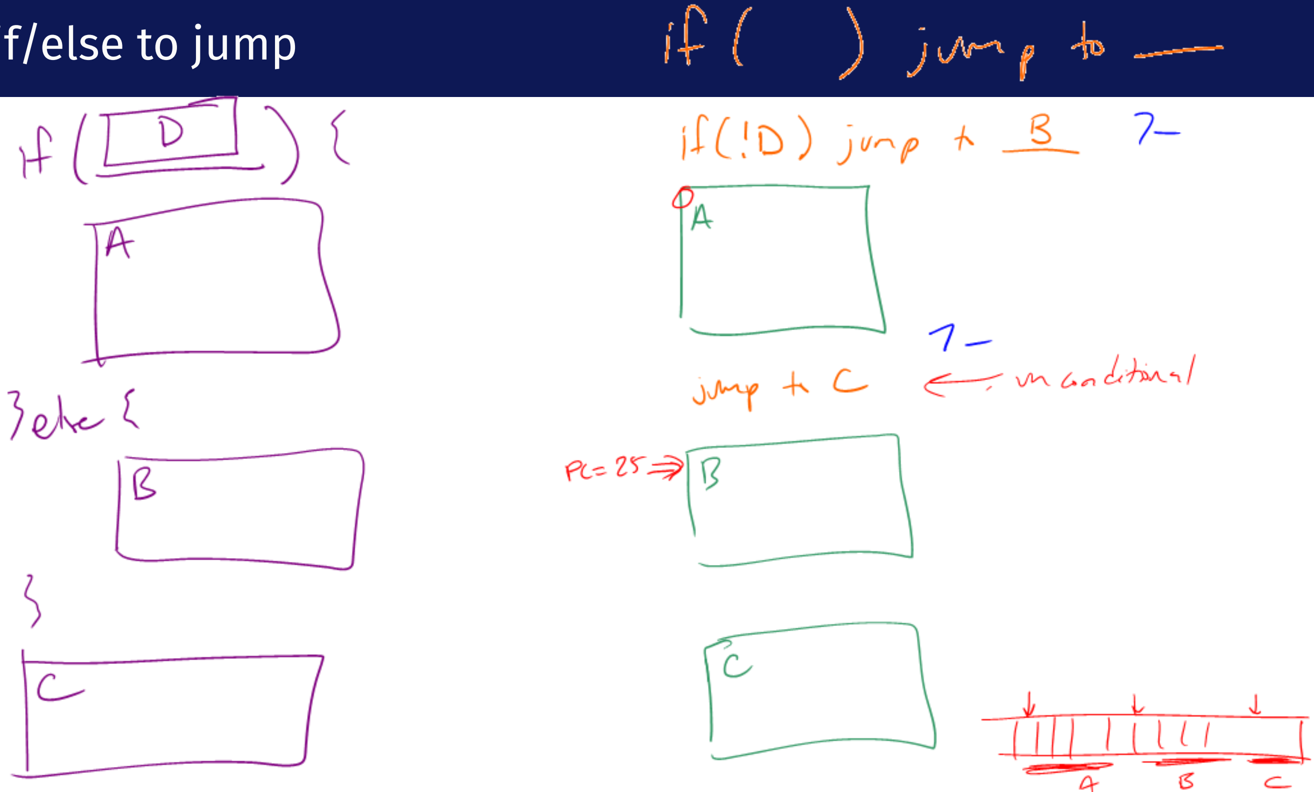
*we do have some limitations, since we can only represent 8-bit values and some operations may be tedious.

Our code to this machine code

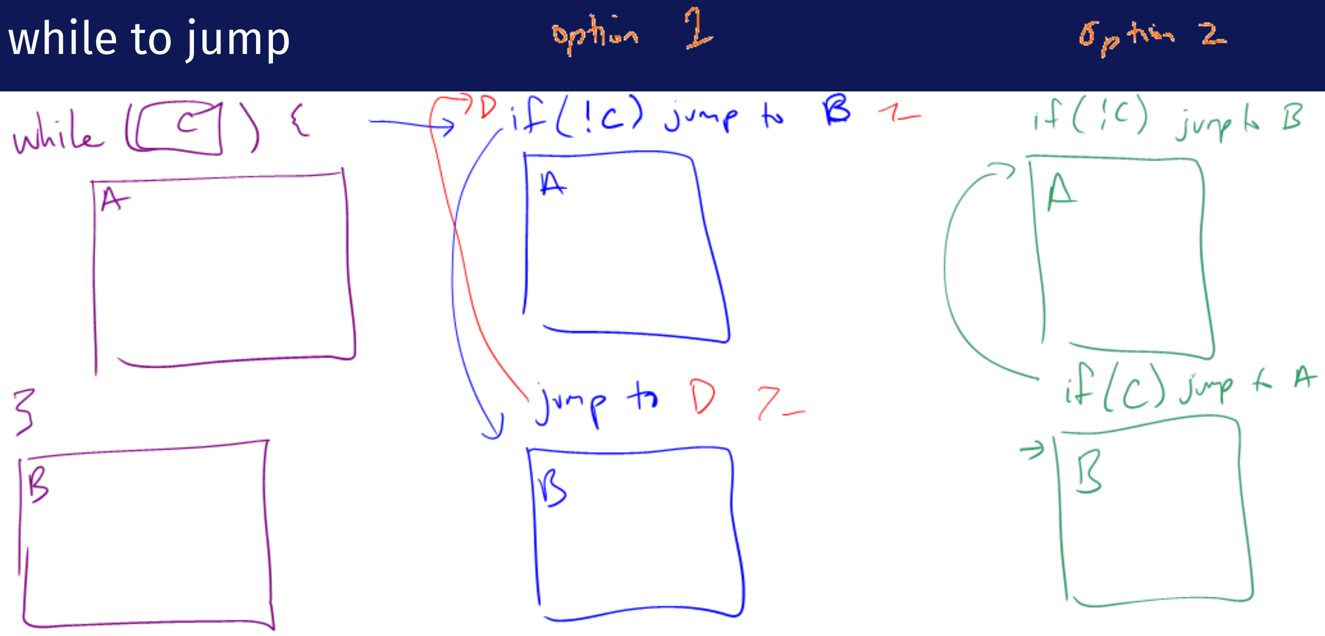
How do we turn our control constructs into jump statements?



if/else to jump



while to jump



Encoding Instructions

icode	b	meaning
0		`rA = rB`
1		`rA += rB`
2		`rA &= rB`
3		`rA` = read from memory at address `rB`
4		write `rA` to memory at address `rB`
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`
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

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

1 byte + 1 byte immediate

1 byte

1 byte

Example

x = 0x17 × 3 → (0x17 + 0x17 + 0x17) / 3

```
x = 0
for (i = 0; i < 3; i++)
    x += 0x17;
```

```
x = 0
i = 0
while (i < 3) {
    x += 0x17;
    i += 1;
}
```

```
r0 = 0
r1 = -2
r3 = pc
r0 += 0x17
r1 += 1
if (r1 < 3) jump to r3
```

r1 <= 2

r1 - 2 <= 0

var	reg
x	r0
i	r1

Example

icode	b	meaning
0		`rA = rB`
1		`rA += rB`
2		`rA &= rB`
3		`rA` = read from memory at address `rB`
4		write `rA` to memory at address `rB`
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`
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

Function Calls

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`

Arrays

Array: a sequence of values (collection of variables)

In Java, arrays have the following properties:

- Fixed number of values
- Not resizable
- All values are the same type

Arrays

Array: a sequence of values (collection of variables)

In Java, arrays have the following properties:

- Fixed number of values
- Not resizable
- All values are the same type

How do we store them in memory?

Arrays

Storing Arrays

In memory, store array sequentially

- Pick address to store array
- Subsequent elements stored at following addresses
- Access elements with math

Example: Store array *arr* at `0x90`

- Access *arr*[3] as `0x90 + 3` assuming 1-byte values

What’s Missing?

What are we missing?

- Nothing says “this is an array” in memory
- Nothing says how long the array is