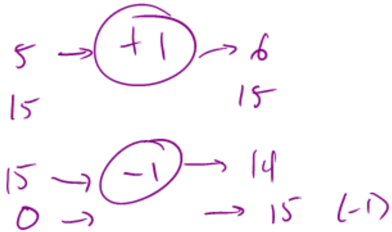
Fetch, Decode, Execute
Instruction Set Architecture

CS 2130: Computer Systems and Organization 1

February 10, 2023

Announcements

- Quiz 3 available today, due Sunday by 11:59pm
- Homework 2 due Monday



Our story so far

- Information modeled by voltage through wires (1 vs 0)
- Transistors
- Gates: & | ~ ^
- Multi-bit values: representing integers
- Floating point
- Multi-bit operations using circuits
- Storing results using registers
- Memory

Code

How do we run code? What do we need?

Example Code

```
...
8:  x = 16
9:  y = x
10: x += y
...
```

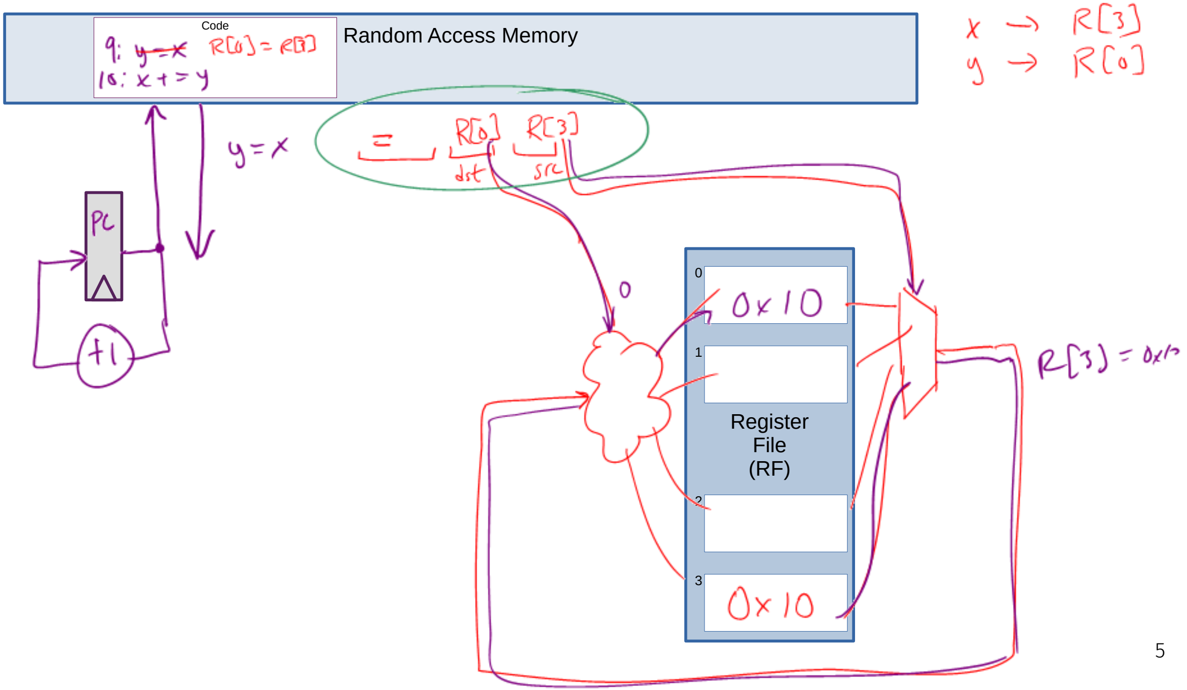
What is the value of *x* after line 10?

Bookkeeping

What do we need to keep track of?

- **Code** - the program we are running
  - RAM (Random Access Memory)
- **State** - things that may change value (i.e., variables)
  - Register file - can read and write values each cycle
- **Program Counter (PC)** - where we are in our code
  - Single register - byte number in memory for next instruction

Building a Computer



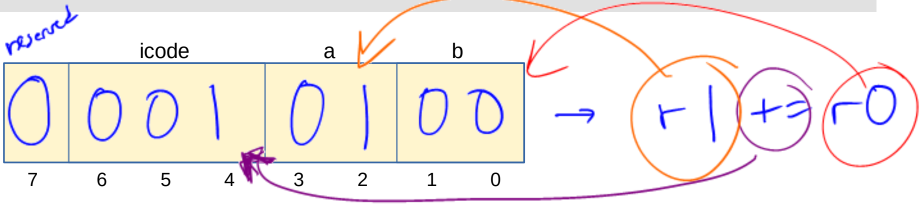
Encoding Instructions

Encoding of Instructions (**icode** or **opcode**)

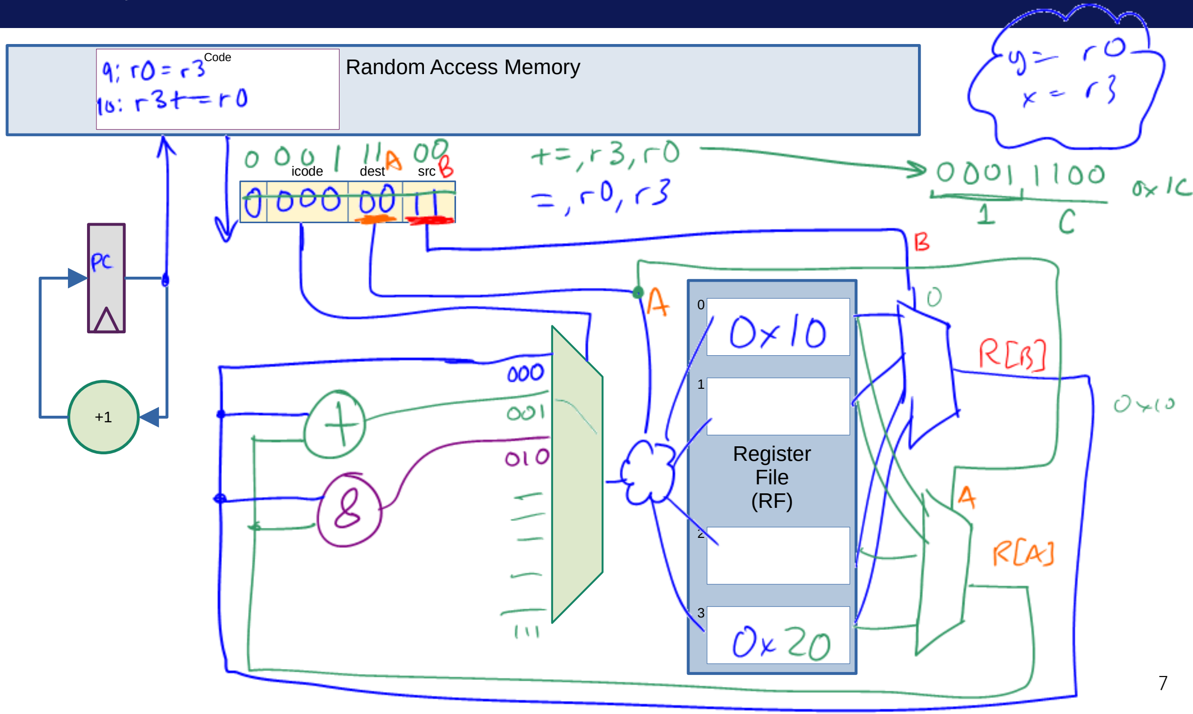
- Numeric mapping from icode to operation

Example 3-bit icode

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



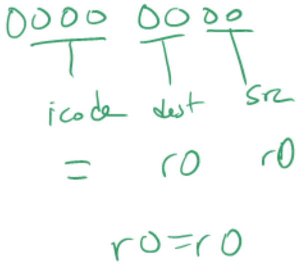
Building a Computer



Question

What happens if we get the 0-byte instruction? `00`

noop



Our Computer's Instructions

Toy ISA 3-bit icode

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

M[r[B]]

Our Computer's Instructions

Toy ISA 3-bit icode

icode	b	action
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

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`
- Register to/from memory (icodes 3-4), `x = M[b]`, `M[b] = x`

Memory

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

Moves

Toy ISA 3-bit icode

icode	b	action
0		`rA = rB`
3		`rA` = read from memory at address `rB`
4		write `rA` to memory at address `rB`
5	3	`rA = pc`
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

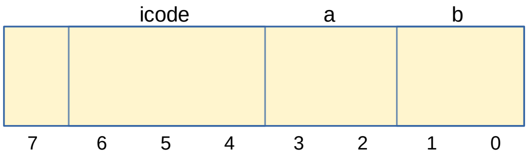
Toy ISA 3-bit icode

icode	b	meaning
1		`rA += rB`
2		`rA &= rB`
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
6	1	`rA` += read from memory at `pc + 1`
	2	`rA` &= read from memory at `pc + 1`

Note: We can implement other operations using these things!

icodes 5 and 6

Special property of icodes 5-6: only one register used

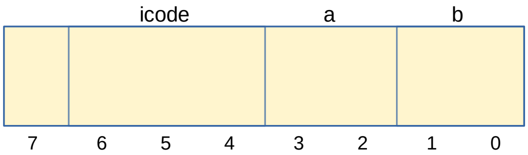


Toy ISA 3-bit icode

icode	b	action
5	0	`rA = ~rA`
	1	`rA = -rA`
	2	`rA = !rA`
	3	`rA = pc`

icodes 5 and 6

Special property of 5-6: only one register used



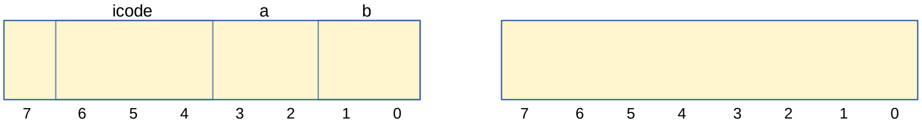
- Side effect: all bytes between 0 and 127 are valid instructions!
- As long as high-order bit is 0
- No syntax errors, any instruction given is valid

Immediate values

icode 6 provides literals, **immediate** values

Example 3-bit icode

icode	b	action
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



Encoding Instructions

Example 1: `r1 += 19`

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

Encoding Instructions

Example 2: `M[0x82] += r3`

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

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

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

For example, consider an `if`

Jumps

Example 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

Function Calls

Encoding Instructions

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

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