UNIVERSITY of VIRGINIA

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 tomorrow, due next Monday at 11:59pm on Gradescope

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 (move a copy of value from one place to another)

Few forms

(primitive variables)

direct register moves like moving the values from r3 to r2 (r2=r3).

- Register to register (icode 0), x = y
- Register to/from memory (icodes 4-5), x = M[b], M[b] = x or, more details: R0=M[R[2]]

(objects or arrays)

Memory

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)

- 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

"dot" operation/square brackets ⇒ read the values from somewhere

just indexes.

UNIVERSITY of VIRGINIA

Moves

icode	b	action
0		`rA = rB`
3	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`
	3	`rA` = read from memory at the address stored at `pc + 1`

next instruction

do things with function calls. (learn it later).

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`

use the basic operations to implement other operations.

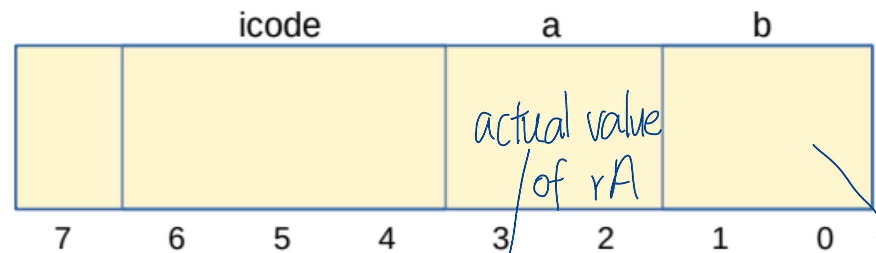
① substract ⇒ taking one value, negating it and adding it to the other.

② multiply → repeated addition

Note: We can implement other operations using these things!

icodes 3 and 6

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

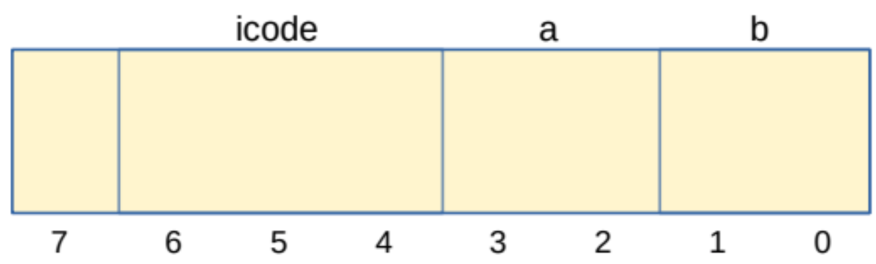


subselect of the instructions.

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



63 2F. . . .

They are valid instructions.
I don't know what they do.
We need to figure out what
they do. But I can write
random things and they are
valid.

- 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 gives us something more powerful.

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 :

	icode				a		b	
bit	7	6	5	4	3	2	1	0
value	0	1	1	0	0	0	0	0

bit	7	6	5	4	3	2	1	0
value	0	0	0	0	0	1	0	1

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

We don't put it in the same instruction ⇒ we would be very limited to the number we can store.

Encoding 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

in decimal

Example 1: r1 += $\boxed{19}$

19 in hexdecimal: 0x13

0	110	01	10
	icode	a	b

1	3

hex: 6b13

Encoding Instructions

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

②. I want to add whatever in R3 to that value.

Example 2: M[0x82] += r3

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 → 6	10 00 → 8	82
r1 = M[r2]	0 100 → 4	01 10 → 6	
r1 += r3	0 010 → 2	01 11 → 7	
M[r2] = r1	0 101 → 5	01 10 → 6	

our machine reads 0 and 1.

But, for us → easier to read → pairs of hex

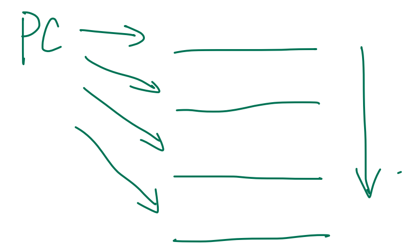
68 82 46 27 56

One interesting finding: first hex is always our icode!

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

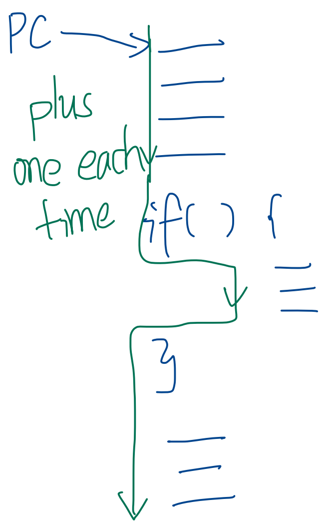
Jumps (control constructs)

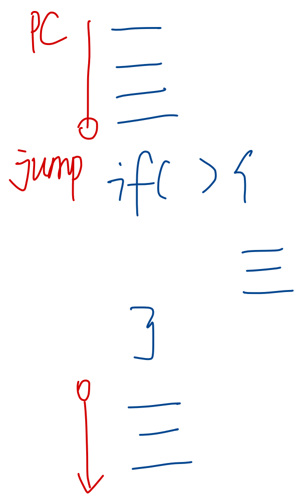


- Moves and math are large portion of our code
- We also need **control constructs**
  - Change what we are going to do next
  - if, while, for, functions, … in terms of machine code, these codes called jumps
- Jumps provide mechanism to perform these control constructs
- We jump by assigning a new value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

Jumps

- For example, consider an if





when we got "if", we have a choice

①. Continue our code, line by line

②. don't want to do the if body, magic teleport. teleports me down to the end of the if statement.

(if the first line after if has index 25, instead of PC+1. I'll say. PC=25)