Toy Instruction Set Architecture

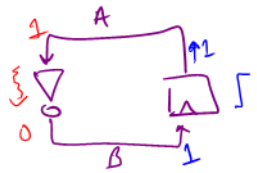
CS 2130: Computer Systems and Organization 1

February 13, 2023

Announcements

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

Quiz Review



0x bebe 9029 → 0x 0e0e 0009

& 0F0F0F0F

>> 4

+ 0x 0b0b 0902

-0b 11 0000

1.011

1.011×2^7

1 1110 011

0111

+ 0111

1110

a = !0 = 0··· 01

b = ~0 = 11··· 11

c = a << (b & 31) = 100··· 0

x = c+1 = 1000··· 001

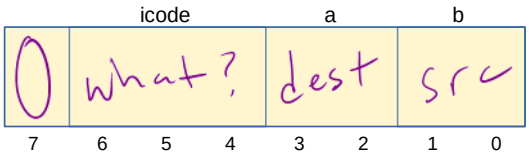
11··· 1 11111

& 0000 0 11111

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

Moves

Few forms

r2 = r3

r0 = M[R[2]]

- Register to register (icode 0), `x = y`
- Register to/from memory (icodes 3-4), `x = M[b]`, `M[b] = x`

M[R[1]] = r2

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

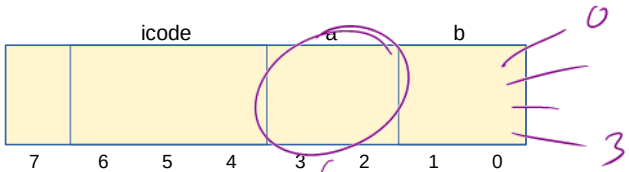
Toy ISA 3-bit icode

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

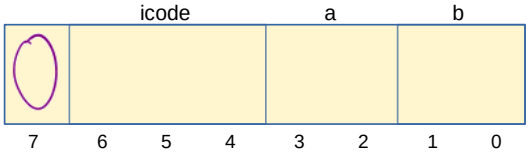


Toy ISA 3-bit icode

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

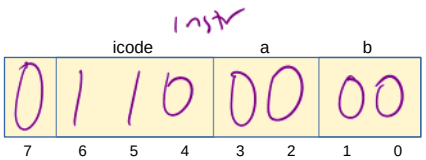
int x = 5;

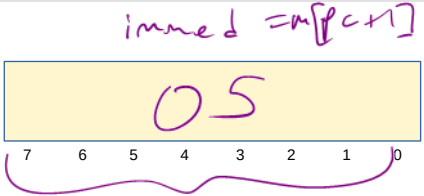
Toy ISA 3-bit icode

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

r0 = 05

rA = M[M[pc+1]]

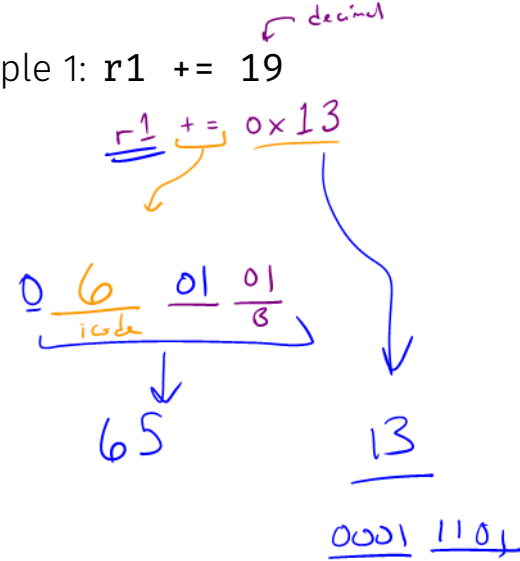




Encoding Instructions

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

Example 1: `r1 += 19`



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

Read memory at address `0x82`, add `r3`,

write back to memory at same address

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

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

Function Calls

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`