Exam Review

CS 2130: Computer Systems and Organization 1

February 22, 2023

Announcements

- Exam 1 Friday (in class)
  - Closed book, closed notes, closed neighbor, closed internet, closed smart-watch
  - Please bring pen or pencil, we will have scratch paper if needed
  - For SDAC accommodations, please schedule a time with their testing center
- Homework 3 due tonight
- Homework 4 available after the exam

Topics

So far, we have electricity on wire to building a computer and programming it!

- Logic: Operations, Gates, Truth tables
- Numbers: Binary, Octal, Decimal, Hexadecimal
- Bitwise Operations: and, or, bitwise not, logical not, xor, …
- Binary Arithmetic: addition, subtraction
- Binary Representations: biased integers, two’s complement, floating point (8-bit)
- Circuits: adder, subtractor, incrementer, registers, clocks
- High-level how these pieces fit together to form a computer
- Instruction Set Architectures (ISAs) and how to write instructions with our ISA

Today’s Agenda

1-bit Logic Gates

- `and`, `or`, `not`
- `nand`, `nor`, `xor`
- Transistors and how to make these gates (high level)

Trinary operator - Mux

- Python: `x = b if a else c`
- Java: `x = a ? b : c`

Numbers

From our oldest cultures, how do we mark numbers?

- Arabic numerals
    - Positional numbering system
    - The `10` is significant:
        - 10 symbols, using 10 as base of exponent
    - The `10` is *arbitrary*
    - *We can use other bases!* $\pi, 2130, 2, \ldots$

Base-8 Example

Try to turn 134_8 into base-10:

Long Numbers in Binary

Making binary more readable

- Typical to group by 3 or 4 bits
- No need for commas *Why?*
- We can use a separate symbol per group
- How many do we need for groups of 3?
- Turn each group into decimal representation
- Converts binary to **octal**

100001010010

Long Numbers in Binary

Making binary more readable

- Groups of 4 more common
- How many symbols do we need for groups of 4?
- Converts binary to **hexadecimal**
- Base-16 is very common in computing

100001010010

Negative Integers

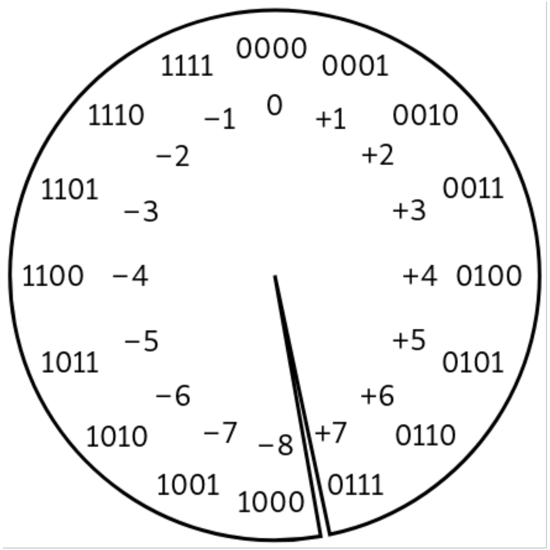
Representing negative integers

- Computers store numbers in fixed number of wires
- Ex: consider 4-digit decimal numbers
- Throw away the last borrow:
  - 0000 - 0001 = 9999
  - 9999 - 0001 = 9998
  - Normal subtraction/addition still works
- This works the same in binary

Two's Complement

This scheme is called **Two's Complement**

- More generically, a *signed* integer
- There is a break as far away from 0 as possible
- First bit acts vaguely like a minus sign
- Works as long as we do not pass number too large to represent



Values of Two's Complement Numbers

Consider the following 8-bit two's complement binary number:

$$11010011$$

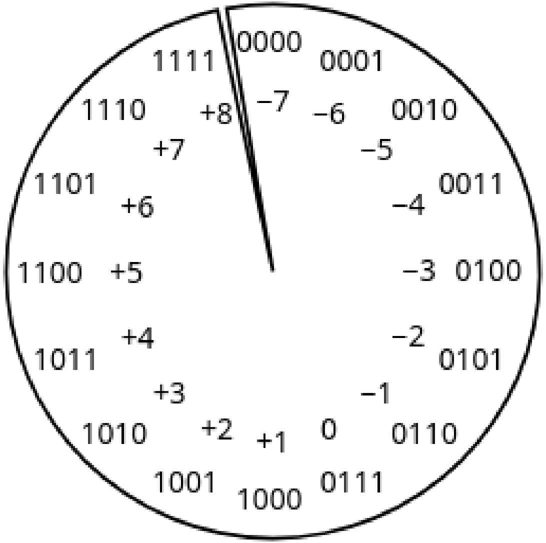
What is its value in decimal?

1. Flip all bits
2. Add 1

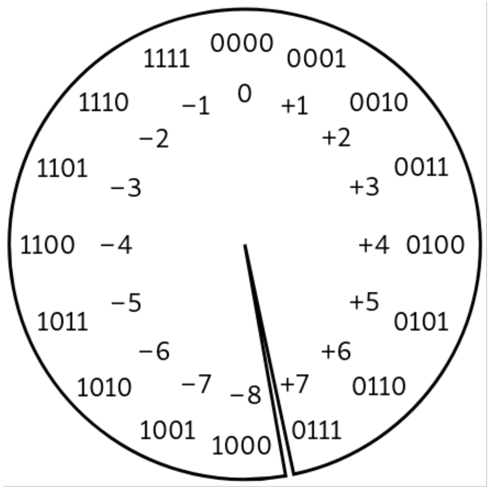
Biased Integers

Similar to Two's Complement, but add **bias**

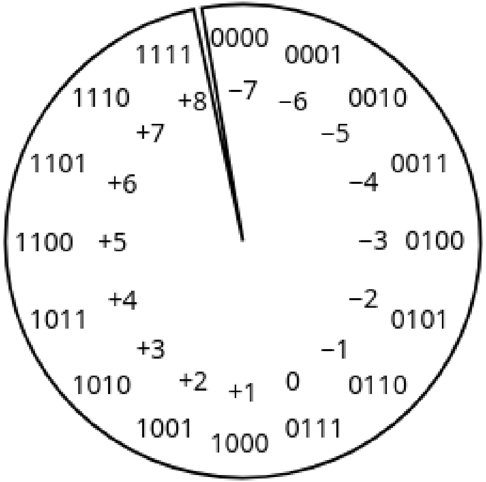
- **Two's Complement**: Define 0 as 00...0
- **Biased**: Define 0 as 0111...1
- Biased wraps from 000...0 to 111...1



Biased Integers



Two’s Complement



Biased

Non-Integer Numbers

Floating point numbers

- Decimal: 3.14159
- Binary: 11.10110
- With integers, the point is always fixed after all digits
- With floating point numbers, the point can move!

Challenge! only 2 symbols in binary

Floating Point in Binary

We must store 3 components

- **sign** (1-bit): 1 if negative, 0 if positive
- **fraction** or **mantissa**: (?-bits): bits after binary point
- **exponent** (?-bits): how far to move binary point

We do not need to store the value before the binary point. Why?

Floating Point in Binary

How do we store them?

- Originally many different systems
- IEEE standardized system (IEEE 754 and IEEE 854)
- Agreed-upon order, format, and number of bits for each

$$1.01101 \times 2^5$$

Exponent

How do we store the exponent?

- Exponents *can* be negative

$$2^{-3} = \frac{1}{2^3} = \frac{1}{8}$$

- Need positive and negative ints (but no minus sign)
- *Don't we always use Two's Complement?* Unfortunately Not
- Biased integers
  - Make comparison operations run more smoothly
  - Hardware more efficient to build
  - Other valid reasons

Floating Point Numbers

Four cases:

- **Normalized**: What we have seen today

$$\mathit{seeeeffff} = \pm 1.\mathit{ffff} \times 2^{\mathit{eeee}-\mathrm{bias}}$$

- **Denormalized**: Exponent bits all 0

$$\mathit{seeeeffff} = \pm 0.\mathit{ffff} \times 2^{1-\mathrm{bias}}$$

- **Infinity**: Exponent bits all 1, fraction bits all 0
- **Not a Number (NaN)**: Exponent bits all 1, fraction bits not all 0

Operations So Far

So far, we have discussed:

- Addition: $x + y$
  - Can get multiplication
- Subtraction: $x - y$
  - Can get division, but more difficult
- Unary minus (negative): $-x$
  - Flip the bits and add 1

Operations (on Integers)

Bit vector: fixed-length sequence of bits (ex: bits in an integer)

- Manipulated by bitwise operations

Bitwise operations: operate over the bits in a bit vector

- Bitwise not: $\sim x$ - flips all bits (unary)
- Bitwise and: $x \& y$ - set bit to 1 if x, y have 1 in same bit
- Bitwise or: $x|y$ - set bit to 1 if either x or y have 1
- Bitwise xor: $x\hat{\ }y$ - set bit to 1 if x, y bit differs

Example: Bitwise AND

```
  11001010
& 01111100
----------
```

Example: Bitwise OR

```
  11001010
| 01111100
```

Example: Bitwise XOR

```
  11001010
^ 01111100
```

Your Turn!

What is: `0x1a ^ 0x72`

Operations (on Integers)

- Logical not: $!x$
  - $!0 = 1$ and $!x = 0, \forall x \neq 0$
  - Useful in C, no booleans
  - Some languages name this one differently
- Left shift: $x << y$ - move bits to the left
  - Effectively multiply by powers of 2
- Right shift: $x >> y$ - move bits to the right
  - Effectively divide by powers of 2
  - Signed (extend sign bit) vs unsigned (extend 0)

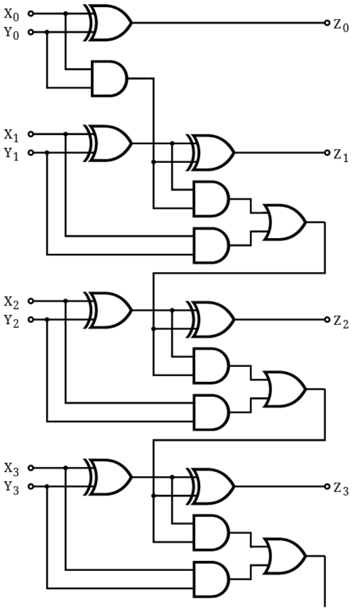
Right Bit-shift Example 2

For **signed** integers, extend the sign bit (1)

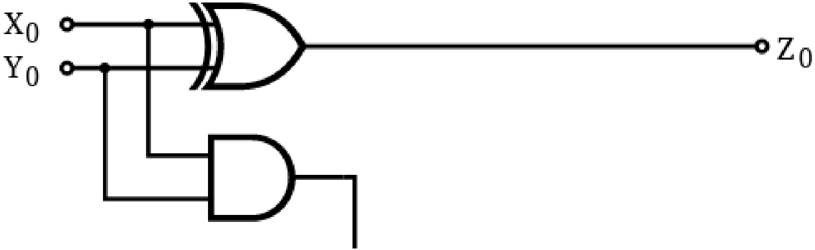
- Keeps negative value (if applicable)
- Approximates divide by powers of 2

```
11001010 >> 1
```

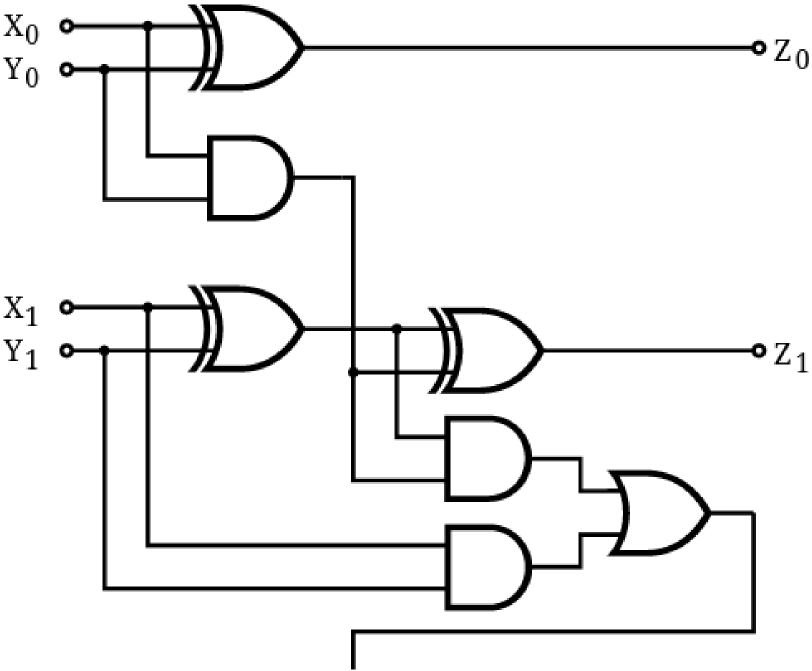
Ripple-Carry Adder



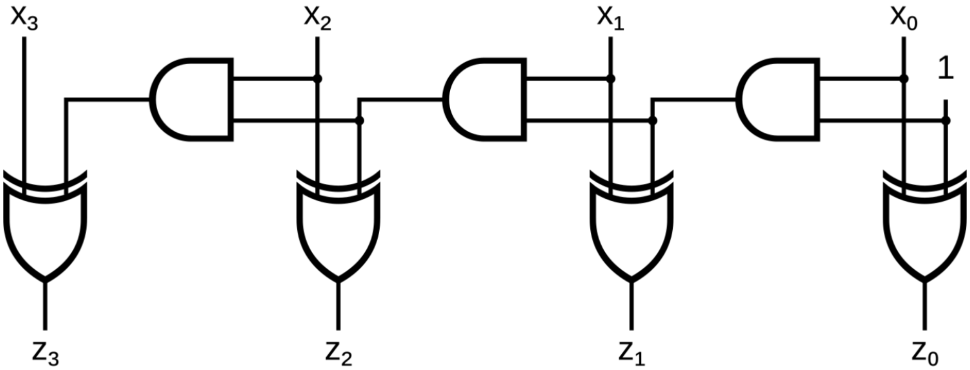
Ripple-Carry Adder



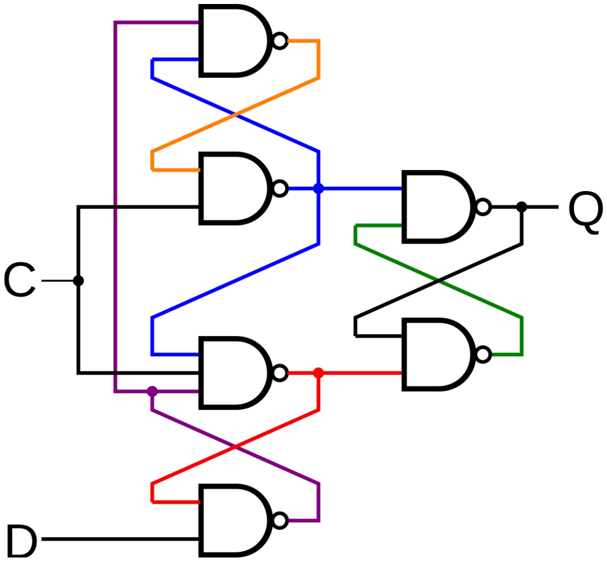
Ripple-Carry Adder



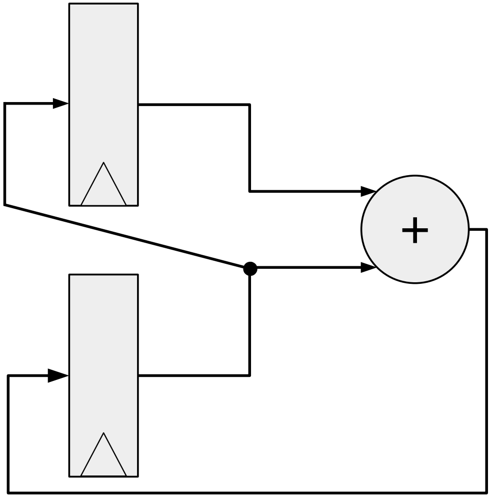
Increment Circuit



1-bit Register Circuit



Another Circuit



Code to Build Circuits from Gates

Write code to build circuits from gates

- Gates we *already* know: `&`, `|`, `^`, `~`
- Operations we can build from gates: `+`, `-`
- Others we can build:
- Ternary operator: `?` `:`

Equals

Equals: `=`

- Attach with a wire (i.e., connect things)
- Ex: `z = x * y`
- What about the following?

  ```
  x = 1
  x = 0
  ```
- **Single assignment**: each variable can only be assigned a value once

Indexing

Indexing with square brackets: `[ ]`

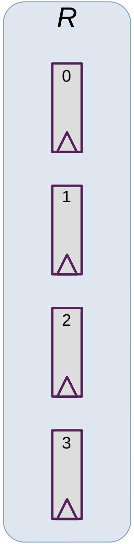
- **Register bank** (or **register file**) - an array of registers
  - Can programmatically pick one based on index
  - I.e., can determine which register while running
- Two important operations:

  `x = R[i]` - Read from a register

  `R[j] = y` - Write to a register

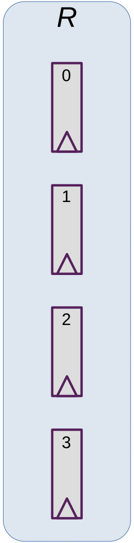
Reading

`x = R[i]` - connect output of registers to *x* based on index *i*



Writing

`R[j] = y` - connect *y* to input of registers based on index *j*



Memory and Storage

Registers $\approx$ KiB

- 6 gates each, ≈ 24 transistors
- Efficient, fast
- Expensive!
- Ex: local variables

Memory $\approx$ GiB

- Two main types: SRAM, DRAM
- DRAM: 1 transistor, 1 capacitor per bit
- DRAM is cheaper, simpler to build
- Ex: data structures, local variables

These do not persist between power cycles

Memory and Storage

Disk $\approx$ GiB-TiB

- Two main types: flash (solid state), magnetic disk
- Magnetic drive
  - Platter with physical arm above and below
  - Cheap to build
  - Very slow! Physically move arm while disk spins
- Ex: files

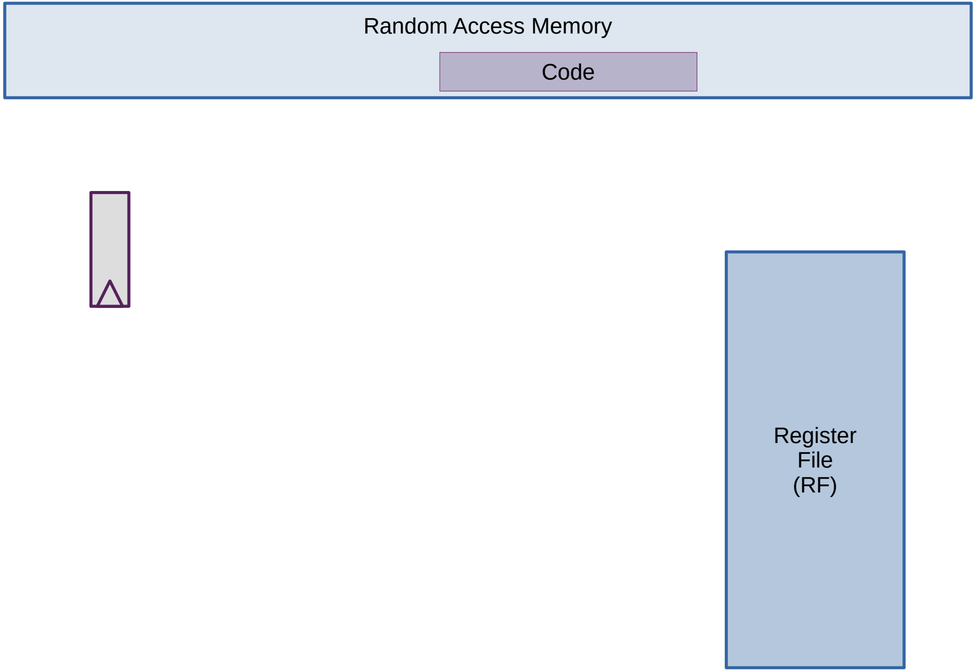
Data on disk does persist between power cycles

Bookkeeping

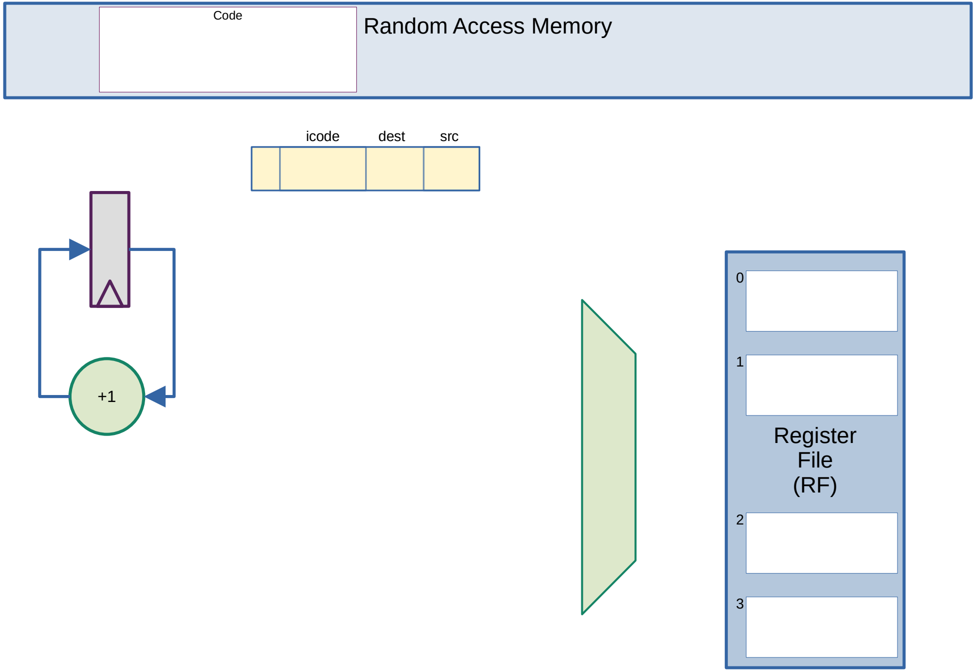
What do we need to keep track of?

- **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)** - were we are in our code
  - Single register - byte number in memory for next instruction

Building a Computer

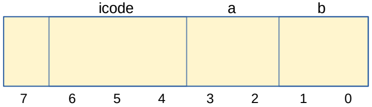


Building a Computer



Our Instruction Set Architecture

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



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

Math

Broadly doing work

Example 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: I can implement other operations using these things!

Jumps

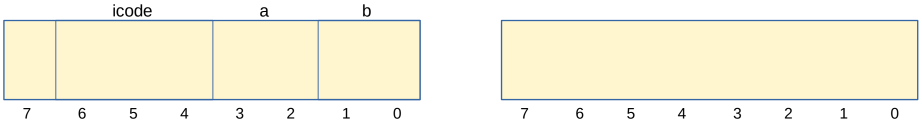
- Moves and math are large portion of our code
- We also need **control constructs**
  - Change what we are going to do next
  - `if`, `while`, `for`, functions, …
- Jumps provide mechanism to perform these control constructs
- We jump by assigning a new value to the program counter PC

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



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.

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?

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