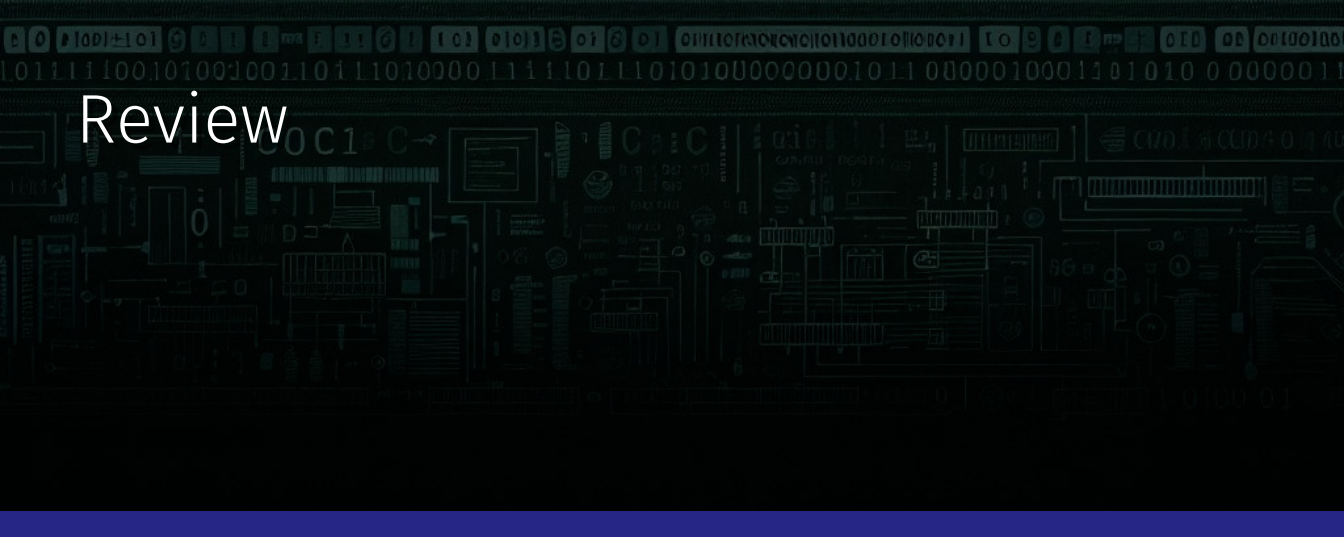


Review

CS 2130: Computer Systems and Organization 1
October 1, 2025

Announcements

- Homework 3 due tonight at 11:59pm on Gradescope
- Midterm 1 Friday (October 3, 2025) in class
  - Written, closed notes
  - If you have SDAC, please schedule ASAP
- No Quiz this Friday!

Putting it together

Overall idea:

- Only need two things (Shannon)
- We can do math with two things (Boole)

Now we need a physical device that deals in two levels

More Vocabulary

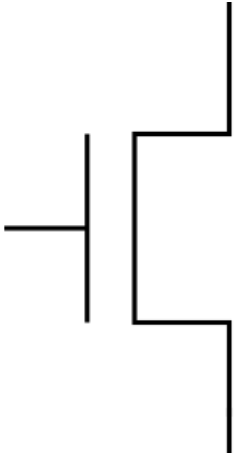
Electricity (conceptually) - involves flow of electrons or other charged carriers through a conductive material

- **current** - rate of flow
- **voltage** - pressure of flow

Examples in water

- High pressure, low flow - squirt gun
- Low pressure, high flow - bucket of water

Transistors

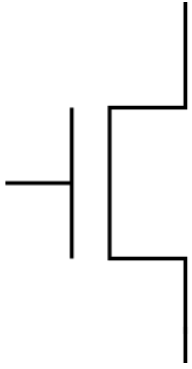


Transistors

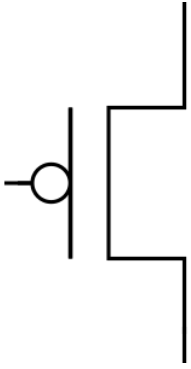
Transistors act like an electrically-triggered switch

- No voltage, no current
- Apply voltage to allow current to flow
- The amount of voltage needed to close the gate is boundary between 0 and 1
- Central technique for how we are going to build binary computers

Transistors

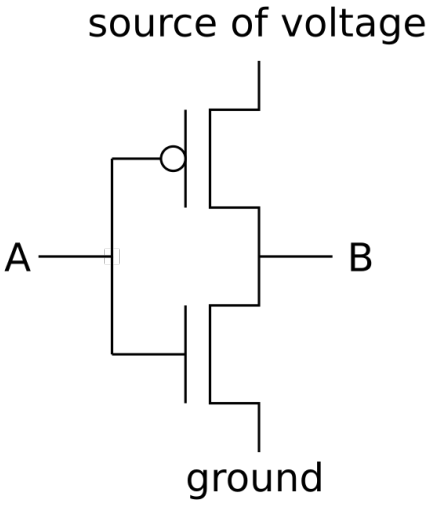


push to close

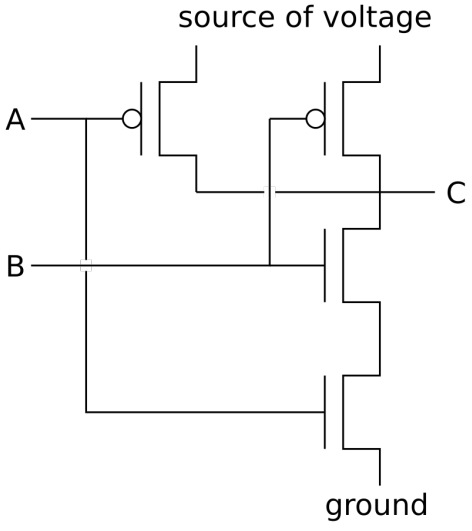


push to open

Circuit Diagram



Circuit Diagram



Multiplexer (mux)

```
x = a ? b : c
```

Numbers

From our oldest cultures, how do we mark numbers?

- **unary** representation: make marks, one per ”thing”
  - Awkward for large numbers, ex: CS 2130?
  - Hard to tell how many marks there are
- Update: group them!
- Romans used new symbols:

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! π, 2130, 2, ...

Bases

We will discuss a few in this class

- Base-10 (decimal) - talking to humans
- Base-8 (octal) - shows up occasionally
- Base-2 (binary) - most important! (we've been discussing 2 things!)
- Base-16 (hexadecimal) - nice grouping of bits

Binary

2 digits: 0, 1

Try to turn 1100101_2 into base-10:

Binary

Any downsides to binary?

Turn 2130_{10} into base-2:
hint: find largest power of 2 and subtract

Long Numbers

How do we deal with numbers too long to read?

- Group them by 3 (right to left)
- In decimal, use commas: ,
- Numbers between commas: 000 - 999
- Effectively base-1000

Long Numbers in Binary - Readability

- 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 - Readability

- 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

Representing Negative Integers

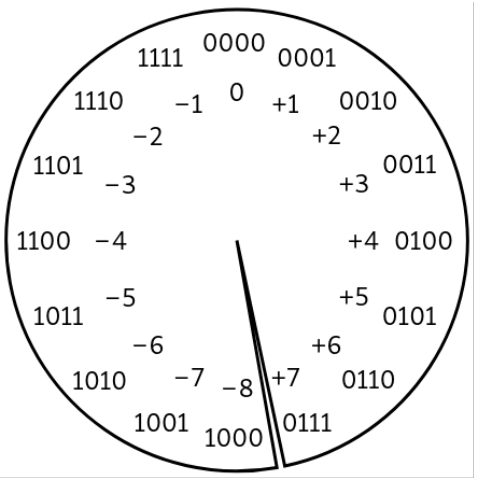
Computers store numbers in fixed number of wires

- Ex: consider 4-digit decimal numbers
- Throw away the last borrow:
  - 0000 - 0001 = 9999 == -1
  - 9999 - 0001 = 9998 == -2
  - Normal subtraction/addition still works
  - Ex: -2 + 3
- 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



Operations

- 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: `~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 ^ y` - set bit to 1 if x, y bit differs

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)

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

Exponent

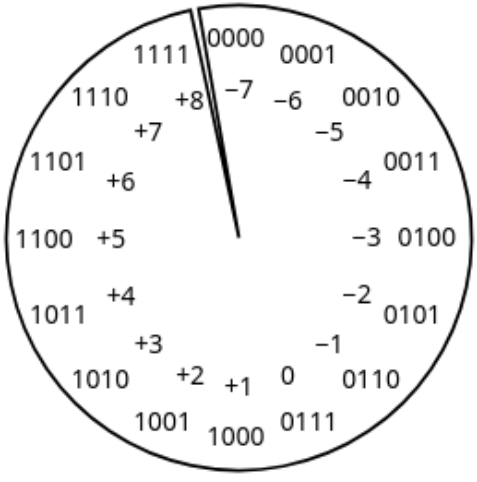
How do we store the exponent?

- Biased integers
  - Make comparison operations run more smoothly
  - Hardware more efficient to build
  - Other valid reasons

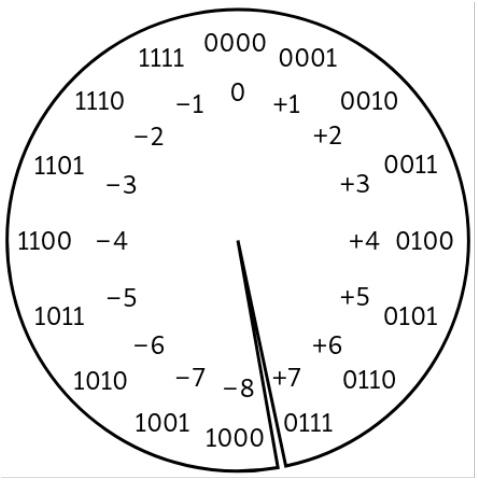
Biased Integers

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

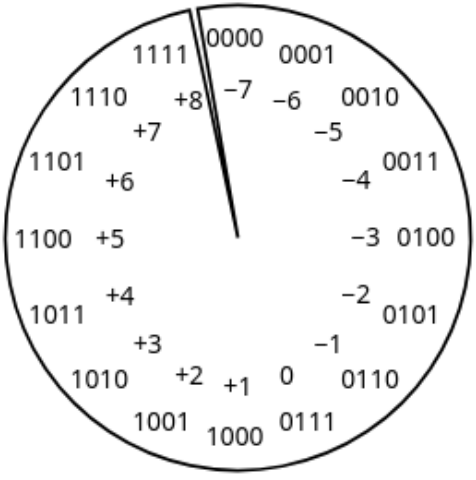
- **Two's Complement**: Define 0 as 00...0
- **Biased**: Define 0 as 0111...1
- Biased wraps from 000...0 to 111...1



Biased Integers



Two's Complement



Biased

Floating Point Numbers

Four cases:

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

$$s\ eeee\ ffff = \pm 1.ffff \times 2^{eeee - \text{bias}}$$

- **Denormalized**: Exponent bits all 0

$$s\ eeee\ ffff = \pm 0.ffff \times 2^{1 - \text{bias}}$$

- **Infinity**: Exponent bits all 1, fraction bits all 0 (i.e., $\pm\infty$)
- **Not a Number (NaN)**: Exponent bits all 1, fraction bits not all 0

Our story so far

- Transistors
- Information modeled by voltage through wires (1 vs 0)
- Gates: `&` `|` `~` `^`

- Multi-bit values: representing integers
  - Signed and unsigned
  - Bitwise operators on bit vectors
- Floating point

Adder

Add 2 1-bit numbers: a, b

Adder

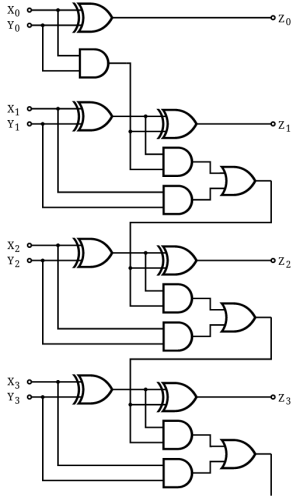
Can we use this in parallel to add multi-bit numbers? What is missing? Consider:

$$\begin{array}{r} 11 \\ +01 \\ \hline \end{array}$$

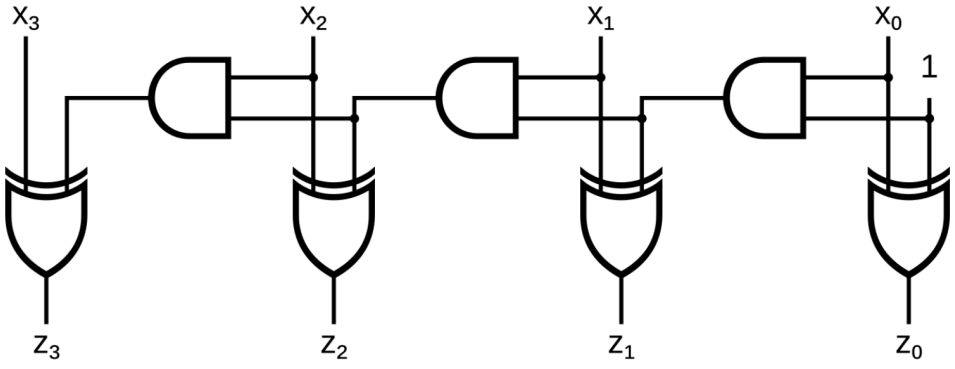
3-input Adder

Add 3 1-bit numbers: a, b, c

ripple-carry adder



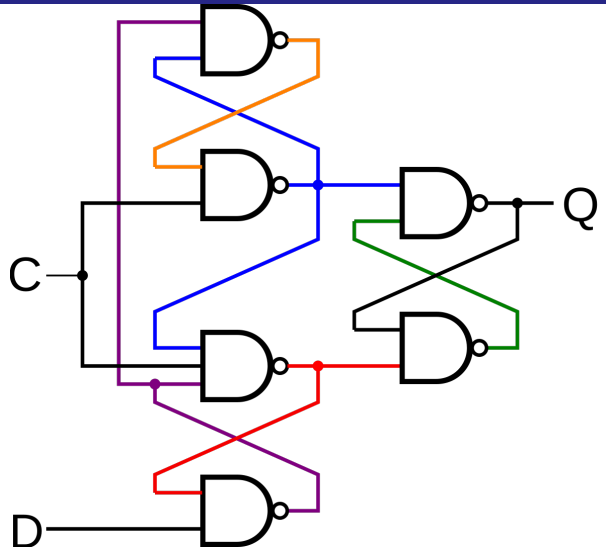
Increment Circuit



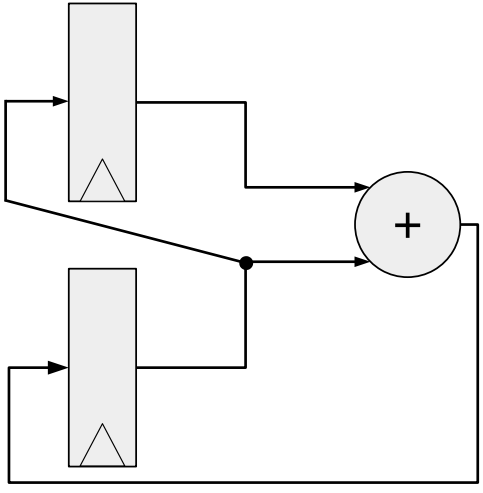
Gate Delay

What happens when I change my input?

1-bit Register Circuit



Another Circuit



Common Model in Computers

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

Subtraction

```
z = x + ~y + 1
```

```
a = ~y
b = a + 1
z = x + y
```

Comparisons

Each of our comparisons in code are straightforward to build:

- == - xor then nor bits of output
- != - same as == without not of output
- < - consider `x < 0`
- >, <=, => are similar

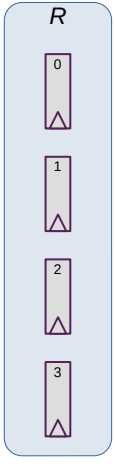
Indexing

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

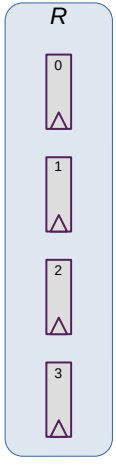
Reading

`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 ≈ KiB

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

These do not persist between power cycles

Memory and Storage

Memory ≈ 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 ≈ 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

Our story so far

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

Code

How do we run code? What do we need?

Consider the following code:

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

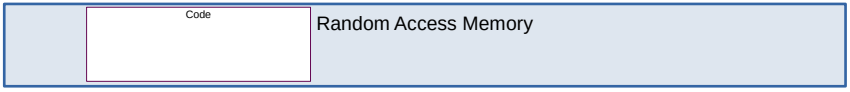
What is the value of x after line 10?

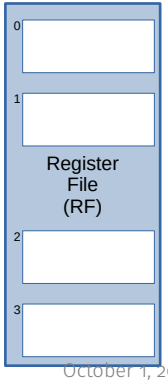
Bookkeeping

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



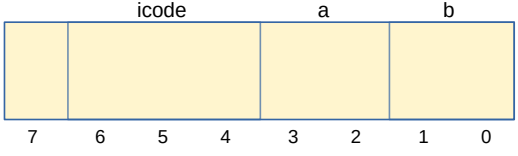


Encoding Instructions

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

- Numeric mapping from icode to operation

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



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

Question

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

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 4-5), `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

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`

Math

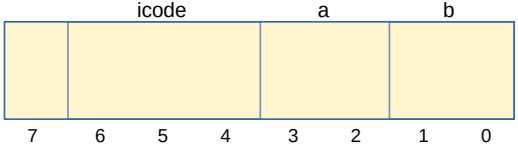
Broadly doing work

icode	b	meaning
1		`rA &= rB`
2		`rA += rB`
3	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 3 and 6

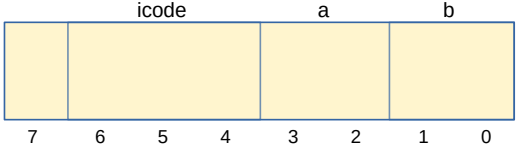
Special property of icodes 3 & 6: only one register used



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

icodes 3 and 6

Special property of 3 & 6: only one register used

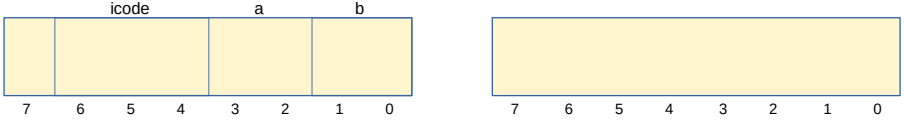


- Side effect: all bytes between 0 and 127 are valid instructions!
- As long as high-order bit is 0
- No syntax errors, any instruction given is valid

Immediate values

icode 6 provides literals, **immediate** values

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

- 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

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.

Writing Code: Homework Hints

1. Write pseudocode that does the desired task

2-3 … deal with control flow

4. Split multi-operation lines into series of single-operation lines
   `x = y-z;` becomes `x = y; x -= z;`
5. Convert operations to those in our instruction set
   `x -= z;` becomes `w = z; w = -w; x += w;`
6. … deal with loops
7. Assign variables to our four registers, ex: `r0=x`, `r1=y`, `r2=z`, `r3=w`
   `r0 = r1; r3 = r2; r3 = -r3; r0 += r3`

10- Write those instructions into triples, then hex

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

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

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

Instruction Set Architecture

Instruction Set Architecture (ISA) is an abstract model of a computer defining how the CPU is controlled by software

- Conceptually, set of instructions that are possible and how they should be encoded
- Results in many *different* machines to implement same ISA
  - Example: How many machines implement our example ISA?
- Common in how we design hardware

Instruction Set Architecture

Instruction Set Architecture (ISA) is an abstract model of a computer defining how the CPU is controlled by software

- Provides an abstraction layer between:
  - Everything computer is really doing (hardware)
  - What programmer using the computer needs to know (software)
- Hardware and Software engineers have freedom of design, if conforming to ISA
- Can change the machine without breaking any programs

Instruction Set Architecture

Instruction Set Architecture (ISA) is an abstract model of a computer defining how the CPU is controlled by software

- Provides an abstraction layer between:
  - Everything computer is really doing (hardware)
  - What programmer using the computer needs to know (software)

CSO: covering many of the times we'll need to think across this barrier

Instruction Set Architecture

Backwards compatibility

- Include flexibility to add additional instructions later
- Original instructions will still work
- Same program can be run on PC from 10+ years ago and new PC today

Most manufacturers choose an ISA and stick with it

- Notable Exception: Apple

Our Instruction Set Architecture

What about our ISA?

- Enough instructions to compute what we need
- As is, lot of things that are painful to do
  - This was on purpose! So we can see limitations of ISAs early
- Add any number of new instructions using the reserved bit (7)

Our Instruction Set Architecture

What about our ISA?

- Enough instructions to compute what we need
- As is, lot of things that are painful to do
  - This was on purpose! So we can see limitations of ISAs early
- Add any number of new instructions using the reserved bit (7)
- Missing something important: *Help to put variables in memory*