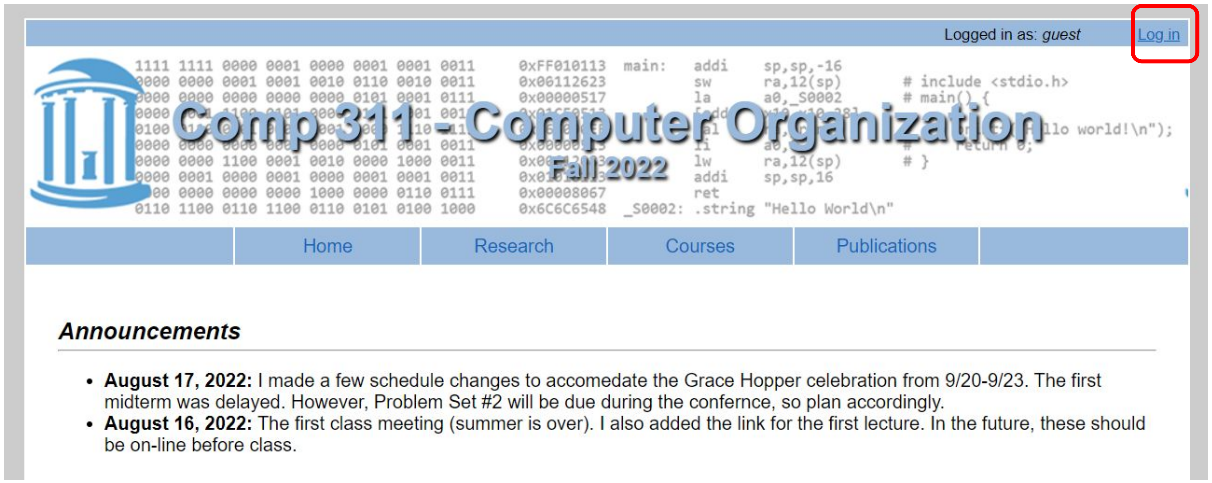
Behind the Curtain

1. Computer organization
2. Computer Instructions
3. Memory concepts
4. Where should code go?
5. Computers as systems

Problem Set #1 will go out on Thursday

Login to Course Website

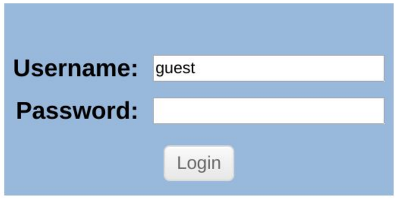
1) Login to your Comp311 account



Announcements

- **August 17, 2022:** I made a few schedule changes to accomedate the Grace Hopper celebration from 9/20-9/23. The first midterm was delayed. However, Problem Set #2 will be due during the confernce, so plan accordingly.
- **August 16, 2022:** The first class meeting (summer is over). I also added the link for the first lecture. In the future, these should be on-line before class.

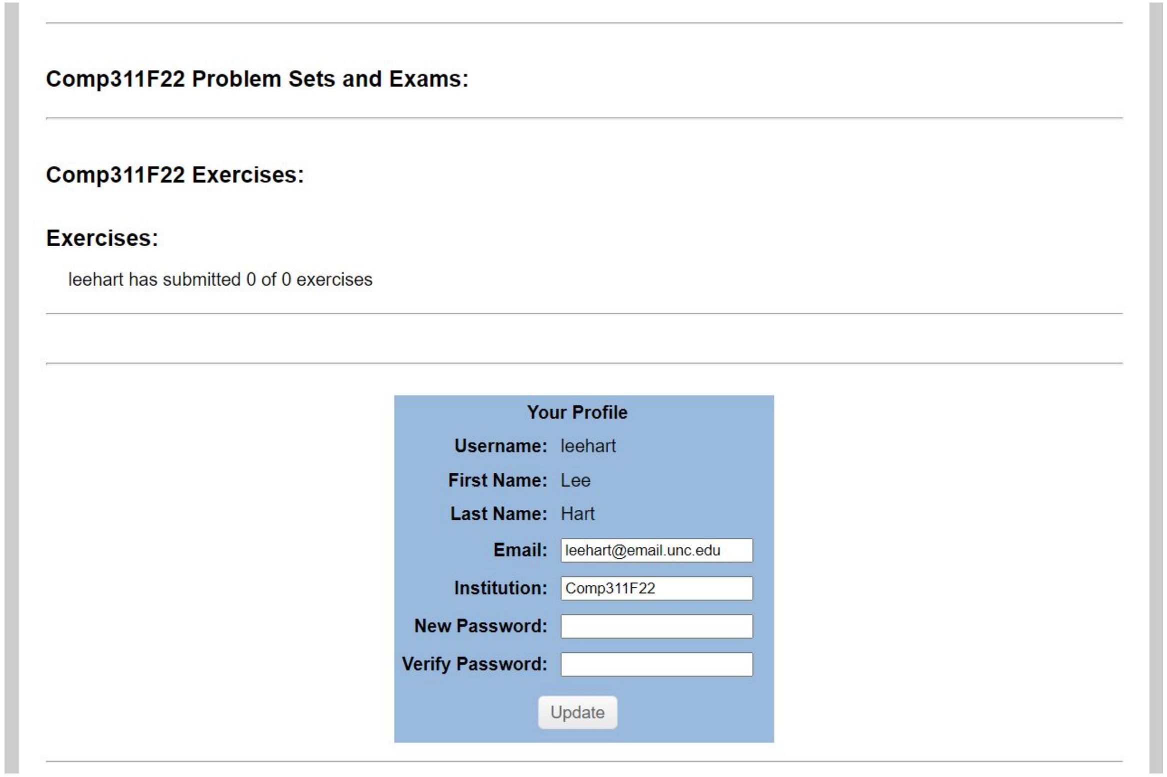
2) Your username is your UNC ONYEN and your password is your PID



Username:	guest
Password:	

Next Steps

3) Once you are logged in, press "Course" and then a "Setup" button should appear. Press "Setup" and you should see something like:



Comp311F22 Problem Sets and Exams:

Comp311F22 Exercises:

Exercises:

leehart has submitted 0 of 0 exercises

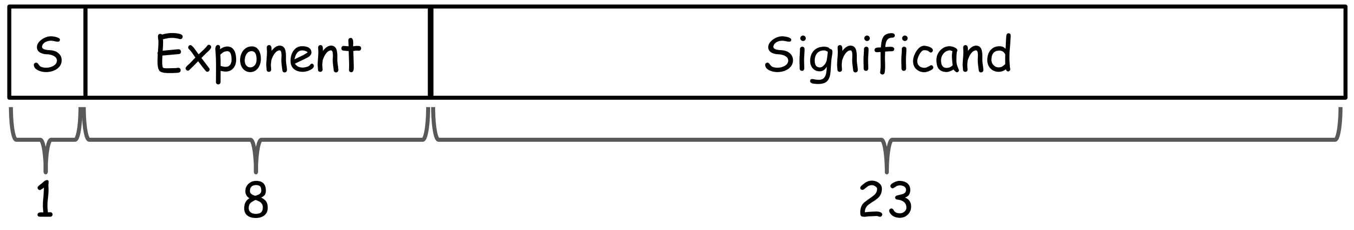
Your Profile

Username:	leehart
First Name:	Lee
Last Name:	Hart
Email:	leehart@email.unc.edu
Institution:	Comp311F22
New Password:	
Verify Password:	

4) (BTW, you can also change your password here if you want).

Catching up from last time...

What decimal value is represented by 0x3f800000, when interpreted as an IEEE 754 single precision floating point number?



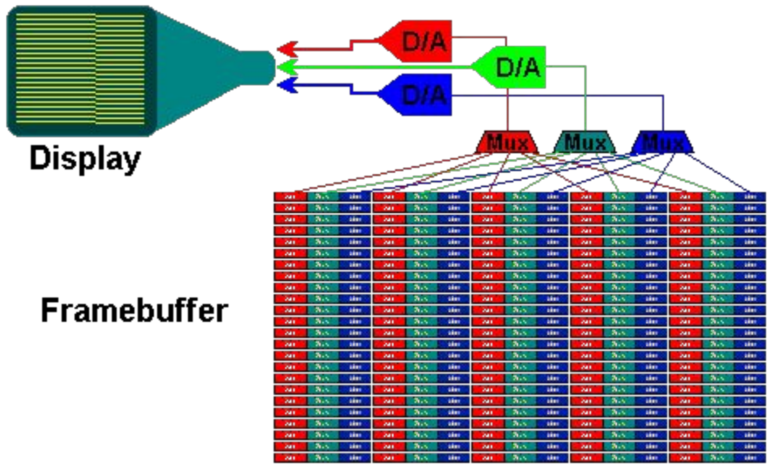
$$v = -1^{s} \times 1.\text{Significand} \times 2^{\text{Exponent}-127}$$

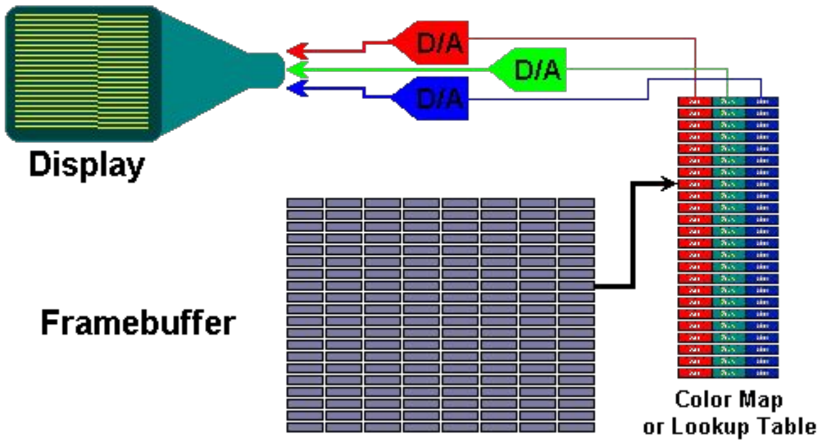
How colors Are Represented

- Each pixel is stored as three primary parts
- Red, green, and blue
- Usually around 8-bits per channel
- Pixels can have individual R,G,B components or they can be stored indirectly via a "look-up table"

8-bits	8-bits	8-bits

3 - 8-bit unsigned binary integers (0,255)
-OR-
3 - fixed point 8-bit values (0-1.0)





Color Specifications

Web colors:

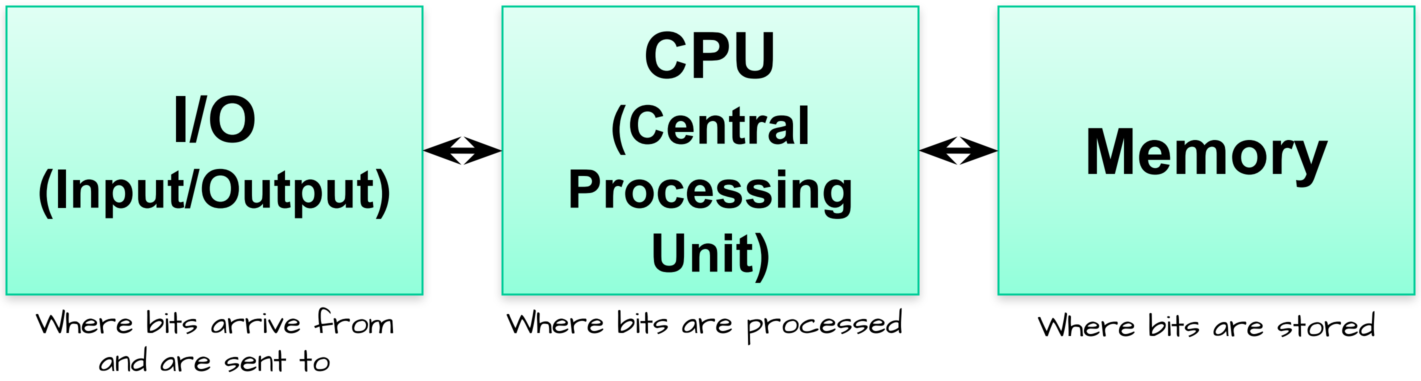
Name	Hex	Decimal Integer	Fractional
Orange	#FFA500	(255, 165, 0)	(1.0, 0.65, 0.0)
Sky Blue	#87CEEB	(135, 206, 235)	(0.52, 0.80, 0.92)
Thistle	#D8BFD8	(216, 191, 216)	(0.84, 0.75, 0.84)

Colors are stored as binary too. You'll commonly see them in Hex, decimal, and fractional formats.

Summary

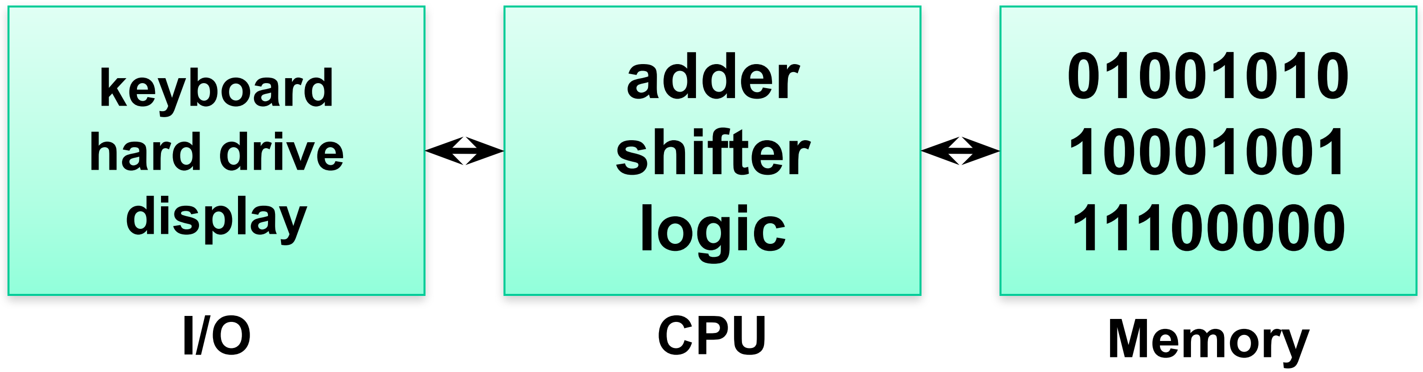
- **ALL** modern computers represent signed integers using a two's-complement representation
- Two's-complement representations eliminate the need for separate addition and subtraction units
- Addition is **identical** using either unsigned and two's-complement numbers
- Finite representations of numbers on computers leads to anomalies
- Floating point numbers have separate fraction and exponent components.

Computer Organization



- Every computer has at least three basic units
  - Input/Output
    - where data arrives from the outside world
    - where data is sent to the outside world
    - where data is archived for the long term (i.e. when the lights go out)
  - Memory
    - where data is stored (numbers, text, lists, arrays, data structures)
  - Central Processing Unit
    - where data is manipulated, analyzed, etc.

Computer Organization (cont)



- Properties of units
  - Input/Output
    - converts symbols to bits and vice versa
    - where the analog "real world" meets the digital "computer world"
    - must somehow synchronize to the CPU's clock
  - Memory
    - stores bits that represent information
    - every unit of memory has an "address" and "contents",
  - Central Processing Unit
    - besides processing, it also coordinates data's movements between units

WHAT SORT OF "PROCESSING"

A CPU performs low-level operations called **INSTRUCTIONS**

Arithmetic

- ADD X to Y then put the result in Z
- SUBTRACT X from Y then put the result back in Y

Logical

- Set Z to 1 if X AND Y are 1, otherwise set Z to 0
  (AND X with Y then put the result in Z)
- Set Z to 1 if X OR Y are 1, otherwise set Z to 0
  (OR X with Y then put the result in Z)

Comparison

- Set Z to 1 if X is EQUAL to Y, otherwise set Z to 0
- Set Z to 1 if X is GREATER THAN OR EQUAL to Y, otherwise set Z to 0

Control

- Skip the next INSTRUCTION if Z is EQUAL to 0

Anatomy of an Instruction

Nearly all instructions can be made to fit a common template

OPCODE DESTINATION, OPERAND$_1$, OPERAND$_2$

What to do:
add
sub
and
or
beq
bne

Where to put the result

Who to apply the operation to… variables, constants, etc..

Issues remaining …

- Which operations to include?
- Where to get variables and constants?
- Where to store the results?

CPU

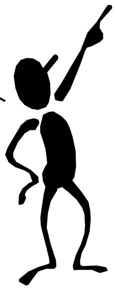
Memory

How is Memory Organized

- By now you know memory is a vast collection of bits
- **Groups of bits** can represent various types of data
  - Integers, Signed integers. Floating-point values, Strings, Pixels
- How do bits get "Grouped"?
- Memory is organized as a vector of bits with indices called "addresses"

0 1 0 0 1 0 1 0 1 0 1 0 [0 0 1 0 1 0 1 0] 0 0 1 0 1 1 1 1 ...

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27



We can address groups of bits like a vector. For example the 8-bits from [12:20], might be the number "42".

Addresses are Key!

The need to "address" bits is one of the most important factors of a computer's design.

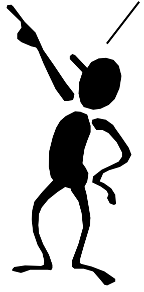
- How many bits will I ever need?
  (remember computer representations are finite)
- The size of scratch variables (registers), is more determined by the need to address bits than the size of the data-types needed..
- Should we squander address space by giving "every" bit a distinct address?
- Perhaps we could address bits in more manageable units

Memory Concepts

- Memory is divided into "addressable" units, each with an address (like an array with indices)
- Addressable units are usually larger than a bit, typically 8 (byte), 16 (halfword), 32 (word), or 64 (long) bits
- Each address has variable "contents"
- Memory contents might be:
  - Integers in 2's complement
  - Floats in IEEE format
  - Strings in ASCII or Unicode
  - Data structure de jour
  - ADDRESSES
  - Nothing distinguishes the difference

Address	Contents
0	42
1	3.141592
2	"Lee "
3	"Hart"
4	"Bud "
5	"Levi"
6	"le "
7	2
8	0x00000293
9	0x00a00313
10	0x006282b3
11	0xfff30313
12	0xfe601ce3
13	0x0000006f
14	0x00004020
15	0x20090001

Here we assume a 32-bit" "Word" address-able machine



One More Thing

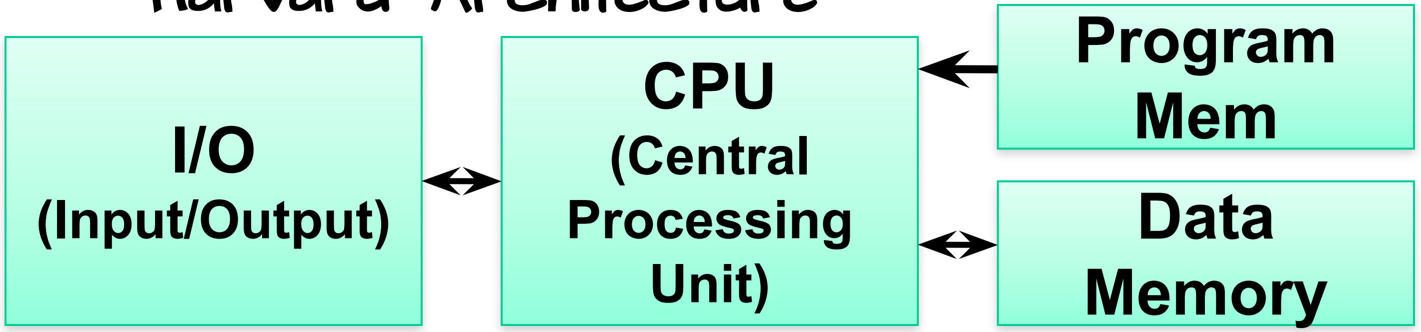
- INSTRUCTIONS for the CPU are stored in memory along with data
- CPU fetches instructions, decodes them and then performs their implied operation
- Mechanism inside the CPU directs which instruction to get next.
- They appear in memory as a string of bits that are typically uniform in size
- Their encoding as "bits" is called "machine language." ex: 0c3c1d7fff
- We assign "mnemonics" to particular bit patterns to indicate meanings.
- These mnemonics are called Assembly language. ex: mv x1, 10

Address	Contents
0	42
1	3.141592
2	"Lee "
3	"Hart"
4	"Bud "
5	"Levi"
6	"le "
7	2
8	li x5,0
9	li x6,10
10	add x5,x5,x6
11	addi x6,x6,-1
12	bne x0,x6,.-2
13	j .
14	0x00004020
15	0x20090001

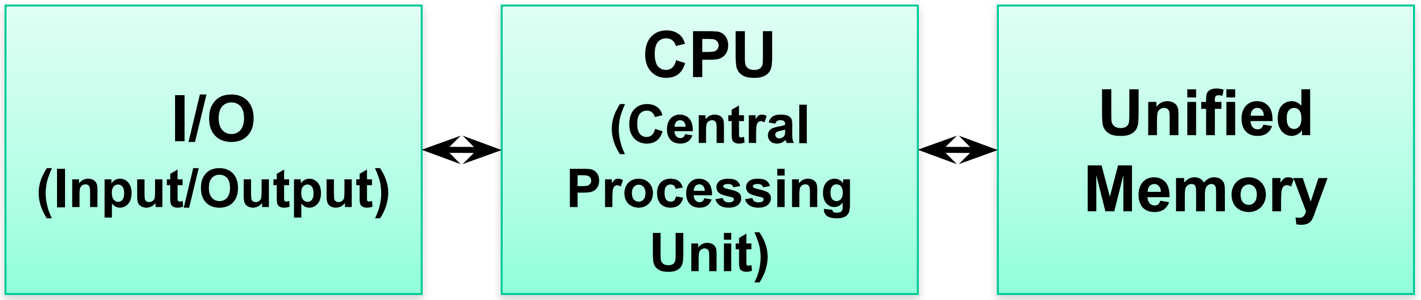
A Bit of History

There is a commonly recurring debate over whether "data" and "instructions" should be mixed. Leads to two common flavors of computer architectures

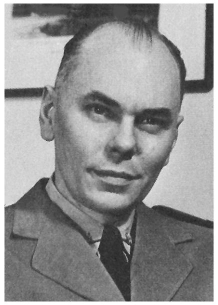
"Harvard" Architecture



"Von Neumann" Architecture



Harvard Architecture


Howard Aiken: Architect of the Harvard Mark 1

Instructions and data do not/should not interact. They can have different "word sizes" and exist in different "address spaces"

- **Advantages**:
    - No self-modifying code (a common hacker trick)
    - Optimize word-lengths of instructions for control and data for applications
    - Higher Throughput (i.e. you can fetch data and instructions from their memories simultaneously)
- **Disadvantages**:
    - The H/W designer decides the trade-off between how big of a program and how large are data
    - Hard to write "Native" programs that generate new programs (i.e. assemblers, compilers, etc.)
    - Hard to write "Operating Systems" which are programs that at various points treat other programs as data (i.e. loading them from disk into memory, swapping out processes that are idle)

Von Neumann Architecture

John Von Neumann: Proponent of unified memory architecture

Instructions are just a type of data that share a common "word size" and "address space" with other types.

- Most common model used today, and what we assume in 411
- **Advantages**:
  - S/W designer decides how to allocate memory between data and programs
  - Can write programs to create new programs (assemblers and compilers)
  - Programs and subroutines can be loaded, relocated, and modified by other programs (dangerous, but powerful)
- **Disadvantages**:
  - Word size must suit both common data types and instructions
  - Slightly lower performance due to memory bottleneck (mediated in modern computers by the use of separate program and data caches)
  - We need to be very careful when treading on memory. Folks have taken advantage of the program-data unification to introduce viruses.

Instructions are Simple

- Computers interpret "programs" by translating them from the high-level language where into "low-level" simple instructions that it understands
- High-Level Languages
  - Compilers (C, C++, Fortran)
  - Interpreters (Basic, Ruby, Lua, Python, Perl, JavaScript)
  - Hybrids (Java)
- Assembly Language

```
x:    .word 0
y:    .word 0
c:    .word 123456
```

```
int x, y;
y = (x-3)*(y+123456);
```

$\longrightarrow$

```
lw   t0,0(gp)        # get x
addi t0,t0,-3
lw   t1,4(gp)        # get y
lw   t2,8(gp)        # get c
add  t1,t1,t2
mul  t0,t0,t1
sw   t0,0(gp)        # save y
```

Instructions are Binary

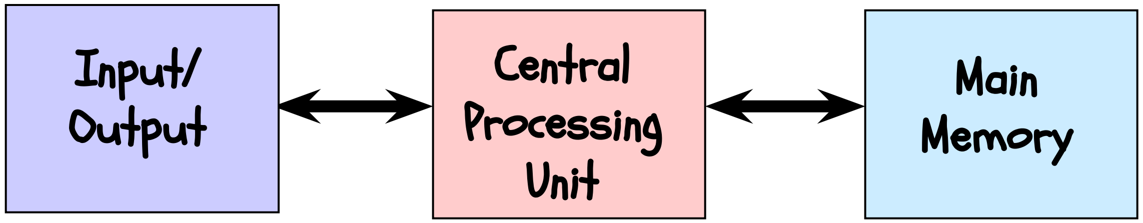
- Computers interpret "assembly programs" by translating them from their mnemonic simple instructions into strings of bits
- Assembly Language
- Machine Language
    - Note the "*mostly*" one-to-one correspondence between lines of assembly code and Lines of machine code

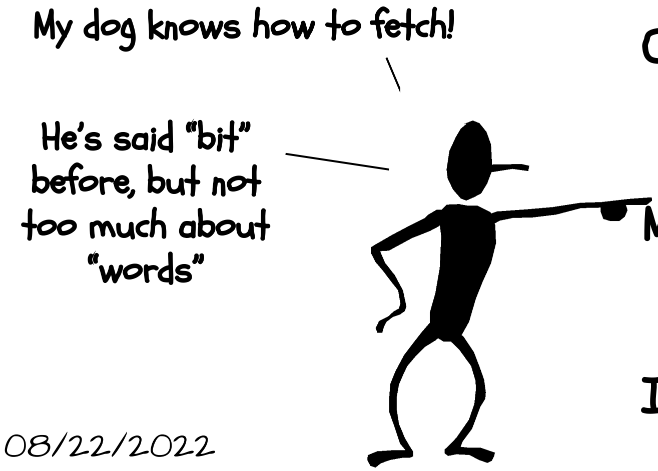
```
                                                         0x00000000
x:   .word 0                                             0x00000000
y:   .word 0                                             0x0001E240
c:   .word 123456
                                                         ...

     lw   t0,0(gp)          # get x          →           0x0001E240
     addi t0,t0,-3                                       0x0001A283
     lw   t1,4(gp)          # get y                      0xFFD28293
     lw   t2,8(gp)          # get c                      0x0081A383
     add  t1,t1,t2                                       0x00730333
     mul  t0,t0,t1                                       0x026282B3
     sw   t0,0(gp)          # save y                     0x0051A023
```

A GENERAL-PURPOSE COMPUTER

THE VON NEUMANN MODEL

Many architectural approaches to the general purpose computer have been explored. The one upon which nearly all modern computers is based was proposed by John von Neumann in the late 1940s. Its major components are:




My dog knows how to fetch!
He's said "bit" before, but not too much about "words"

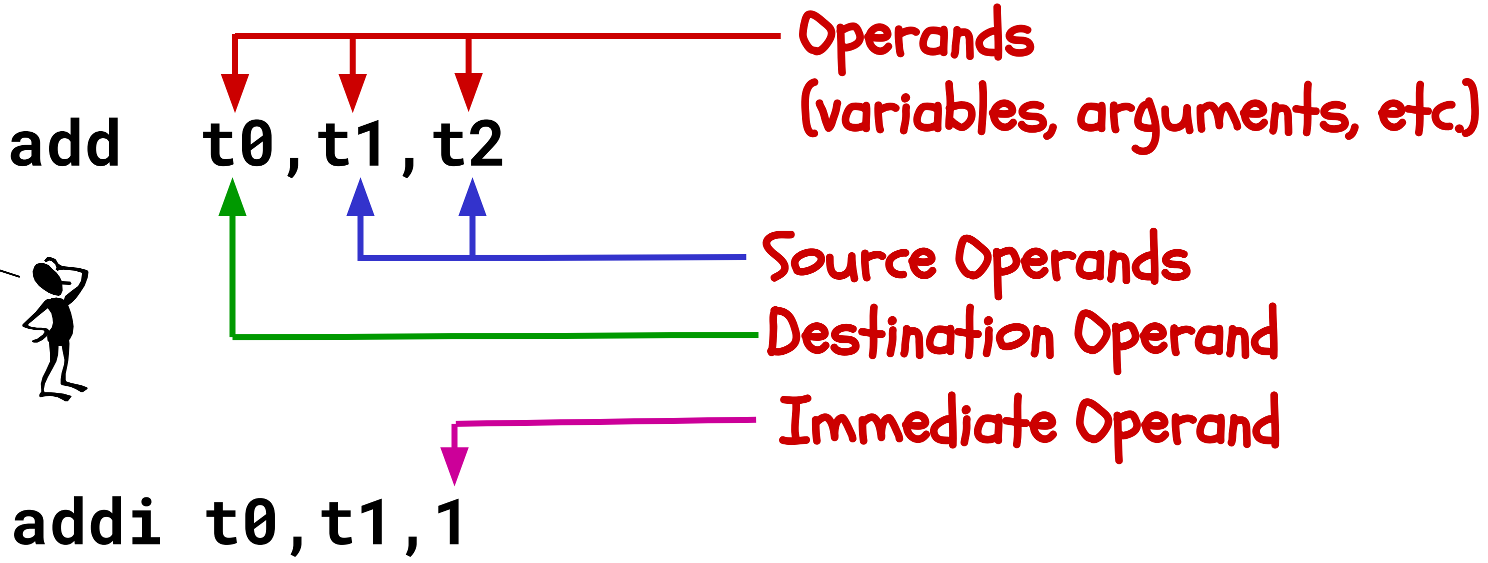
Central Processing Unit (CPU): A device which fetches, interprets, and executes a specified set of bits called Instructions.

Memory: storage of N words of W bits each, where W is a fixed architectural parameter, and N can be expanded to meet needs.

I/O: Devices for communicating with the outside world.

Anatomy of an Instruction

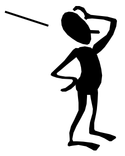
- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (arguments of the operation)
- Types of operands: **destination**, **source**, and **immediate**



```
add  t0,t1,t2

addi t0,t1,1
```

Why do all of the variables start with "R"?

CPU's have a small number (16-32) of registers that are used to hold variables

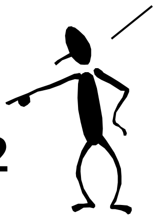


Meaning of an Instruction

- Operations are abbreviated into **opcodes** (1-4 letters)
- Instructions are specified with a very regular syntax
  - Opcodes are followed by arguments
  - Usually the destination is next, then one or more source arguments (This is not strictly the case, but it is generally true)
- Why this order?

  Analogy to high-level language like Java or C

```
add  t0,t1,t2
```



The instruction syntax provides operands in the same order as you would expect in a statement from a high level language.

```
int r0, r1, r2;
r0 = r1 + r2;
```

Instead of:

```
r1 + r2 = r0;
```

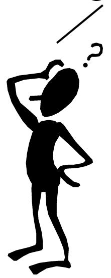
A Series of Instructions

- Generally...
  - Instructions are retrieved sequentially from memory
  - An instruction executes to completion before the next instruction is started
  - But, there are exceptions to these rules

Instructions

`add t0, t1, t1`
`add t0, t0, t0`
`add t0, t0, t0`
`sub t1, t0, t1`

What does this program do?



Variables

t0: ~~0~~ ~~12~~ ~~24~~ 48
t1: ~~6~~ 42
t2:8
t3:10

Program Analysis

- Repeat the process treating the variables as unknowns or "formal variables"
- Knowing what the program does allows us to write down its specification, and give it a meaningful name
- The instruction sequence then becomes a general-purpose tool

Instructions

```
add t0, t1, t1
add t0, t0, t0
add t0, t0, t0
sub t1, t0, t1
```

Variables

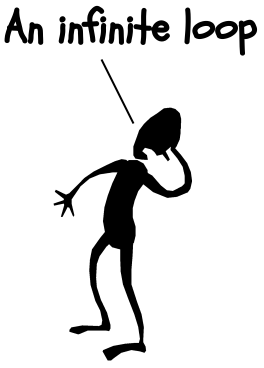
t0: ~~x~~ ~~2x~~ ~~4x~~ 8x
t1: ~~x~~ 7x
t2: y
t3: z

Looping the Flow

- Repeat the process treating the variables as unknowns or "formal variables"
- Knowing what the program does allows us to write down its specification, and give it a meaningful name
- The instruction sequence then becomes a general-purpose tool

Instructions

```
times7: add  t0,t1,t1
        add  t0,t0,t0
        add  t0,t0,t0
        sub  t1,t0,t1
        j    times7
```

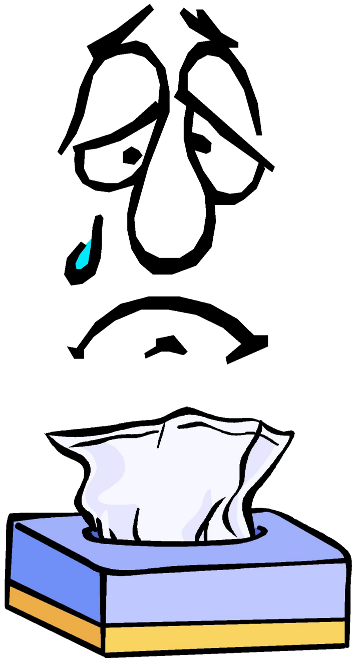


Variables

t0: ~~w~~ ~~8x~~ ~~56x~~ 392x
t1: ~~x~~ ~~7x~~ ~~49x~~ 343x
t2: y
t3: z

Open Issues in our Simple Model

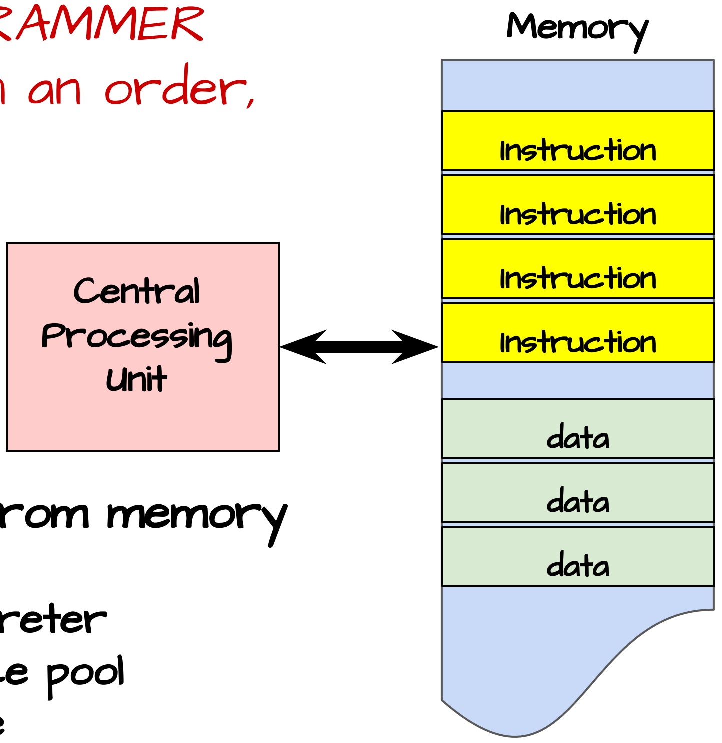
- WHERE in memory are INSTRUCTIONS stored?
- HOW are instructions represented?
- WHERE are VARIABLES stored?
- What are LABELs? How do they relate to where instructions are stored?
- How about more complicated data types?
    - Arrays?
    - Data Structures?
    - Objects?
- Where does a program start executing?
- When does it stop?



The Stored-Program Computer

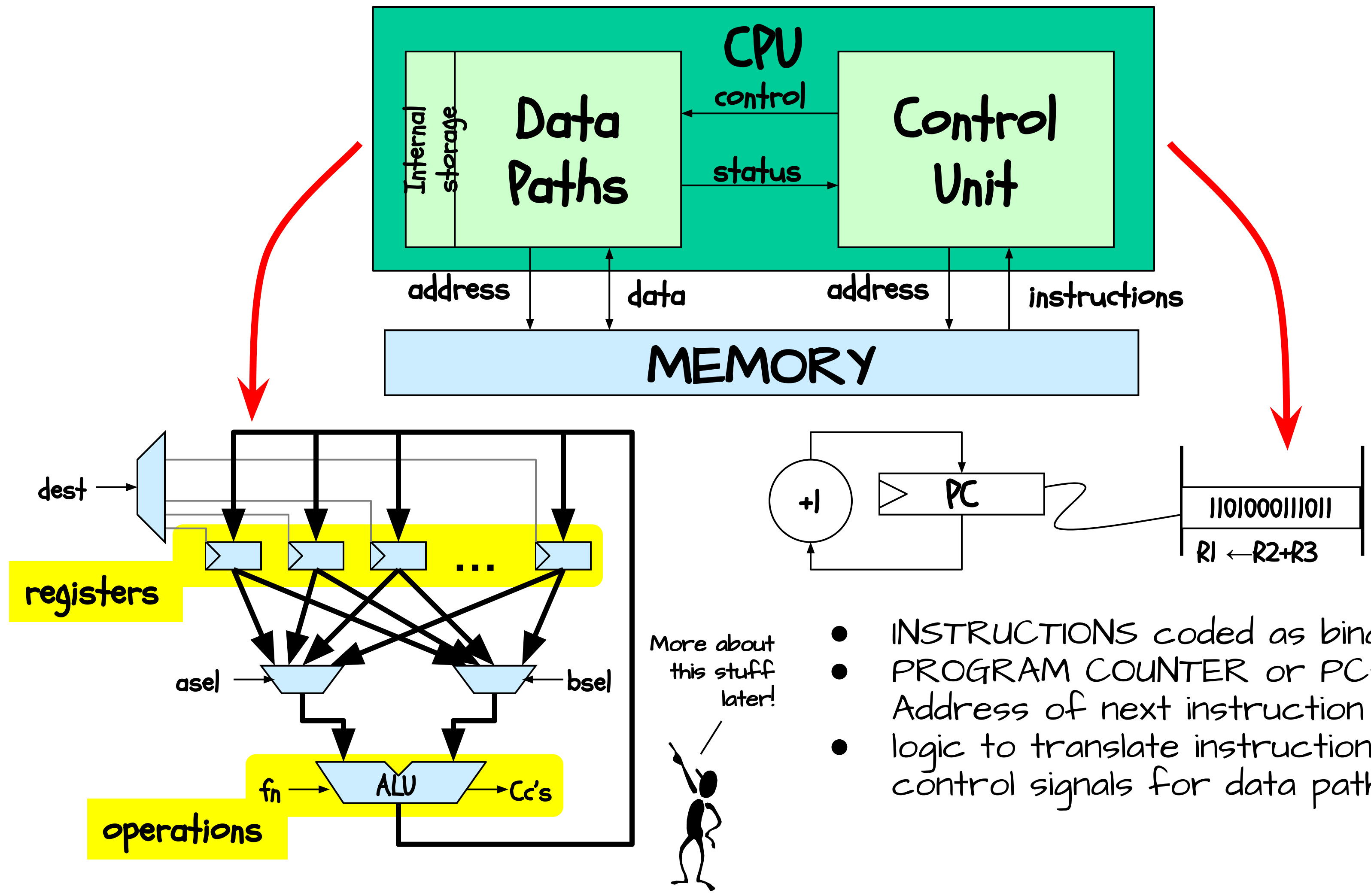
- The von Neumann architecture addresses these issues as follows:
- Instructions and Data are stored in a common memory
- Sequential semantics: To the *PROGRAMMER* all instructions appear to execute in an order, or sequentially

Key idea: Memory holds not only data, but *coded instructions* that make up a *program*.



CPU fetches and executes instructions from memory
- The CPU is a H/W interpreter
- Program IS simply DATA for this interpreter
- Main memory: Single expandable resource pool
  - constrains both data and program size
  - don't need to make separate decisions of how large of a program or data memory to buy

Anatomy of a von Neumann Computer



- INSTRUCTIONS coded as binary data
- PROGRAM COUNTER or PC: Address of next instruction to execute
- logic to translate instructions into control signals for data path

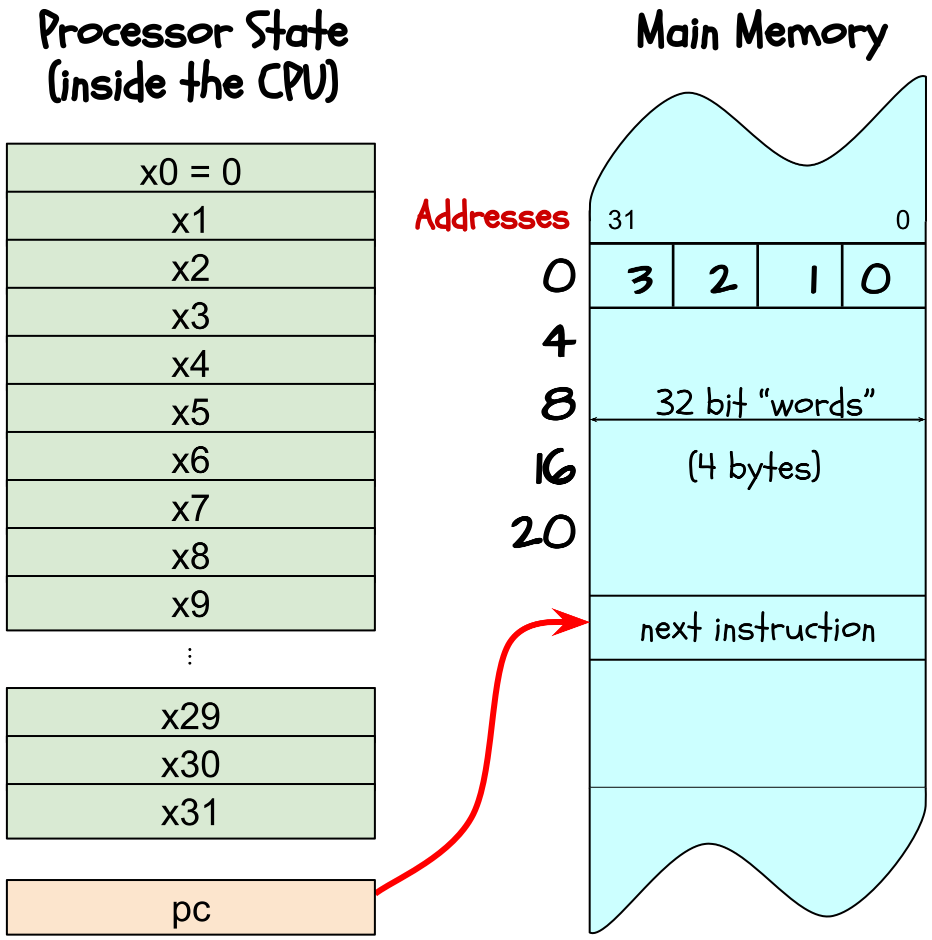
Instruction Set Architecture (ISA)

Encoding of instructions raises some interesting choices...

- Trade Offs: performance, compactness, programmability
- Uniformity. Should different instructions
  - Be the same size (number of bits)?
  - Take the same amount of time to execute?
  - Trend: Uniformity. Affords simplicity, speed, pipelining.
- Complexity. How many different instructions? What level operations?
  - Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc
  - "Reduced Instruction Set Computer" (RISC) philosophy: simple instructions, optimized for speed
- Mix of Engineering & Art...

RISC-V Programming Model

A REPRESENTATIVE RISC MACHINE



In Comp 311 we'll use a subset of the RISC-V core Instruction set as an example ISA (RV32I).

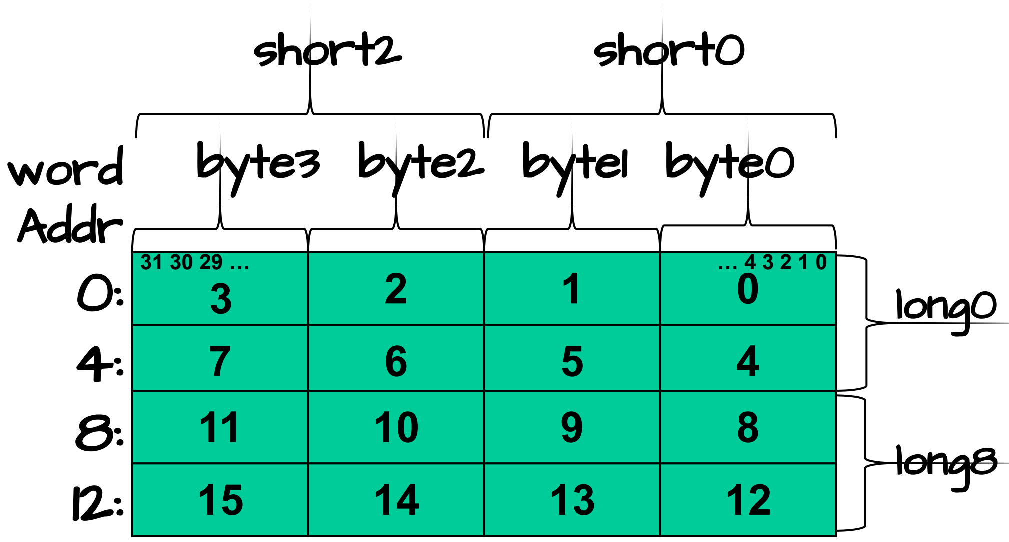
Fetch/Execute loop:

- fetch Mem[PC]
- execute fetched instruction (may change PC!)
- PC = PC + 4
- repeat!

RISC-V uses BYTE memory addresses. However, each instruction is 32-bits wide.. Each word contains four 8-bit bytes. Addresses of consecutive instructions (words) differ by 4.

RISC-V Memory Nits

- Memory locations are addressable in different sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/doubles)
- We also frequently need access to individual bits! (Instructions help with this)
- Every BYTE has a unique address (RISC-V is a byte-addressable machine)
- Most instructions are one word



Next Time

- We'll examine the RISC-V instruction set
  - Assembly language
  - Machine language