COMP311: *COMPUTER ORGANIZATION!*

Lecture 2: High-Level Overview, What is an Instruction, 211 Review!

tinyurl.com/comp311-sp26

What is the course about?

High-level programming languages

```
// High-level (C)
int add3(int a, int b)
{
    return a + b + 3;
}
```

Easy for humans to read/write

Assembly
Low-level languages

```
# Matching RISC-V assembly
# a0 = a, a1 = b;  return in a0

add   a0, a0, a1    # a0 = a0 + a1
addi  a0, a0, 3     # a0 = a0 + 3
ret                 # return
```

Human readable

Machine code

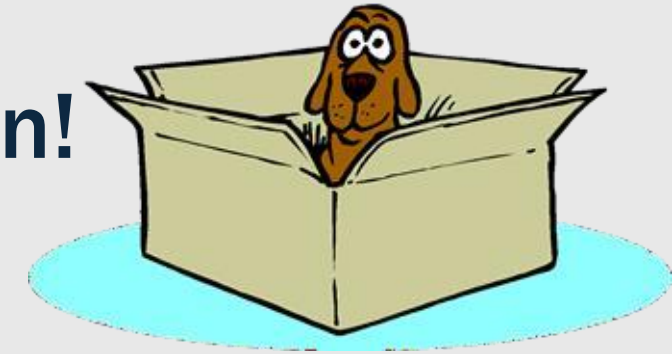
```
0000000 01011 01010
000 01010 0110011

000000000011 01010
000 01010 0010011

000000000000 00001
000 00000 1100111
```

Machine-readable

Recall: Our course themes!

Course Theme #1: Demystify Computers!

Course Theme #2: The Power of Abstraction!



Recall: Goals of this Course

- How does a computer work?
- How is data represented in a computer?
  - Numbers, strings, arrays, photos, music
- How is a program represented in a computer?
- What does a computer do with my program?
- How is data stored? How is data processed?
- Are there any limits to what a computer can do?

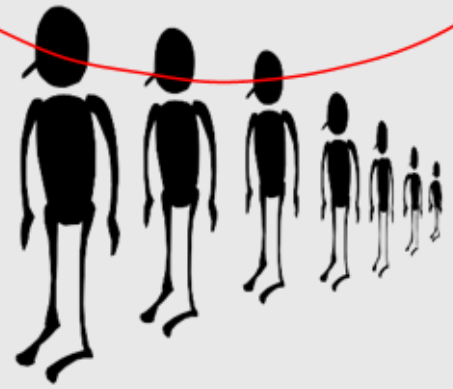
211 REVIEW

Encoding Positive Integers

Encode positive integers as a *sequence of bits.*

Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an n-bit number encoded in this fashion is given by the following formula:

$$v = \sum_{i=0}^{n-1} 2^i b_i$$



2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
0	0	0	0	0	1	1	1	1	1	1	0	1	0	1	0

$2 + 2^3 + 2^5 + 2^6 + \cdots +$

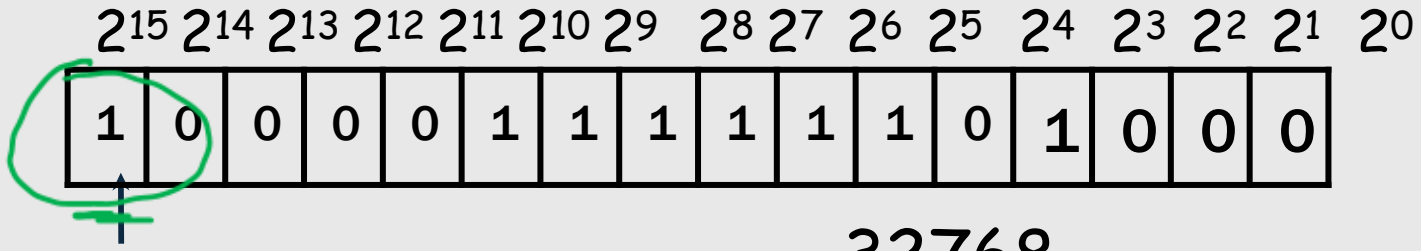
Some Important Bits

- You are going to have to get accustomed to working in binary, but it will be helpful throughout your career as a computer scientist.
- Some good ones to know
  - *The first 10 powers of 2*
  - *The prefixes for powers of 2 that are powers of 10*

Recall: 2's Complement Notation

- The 2's complement representation for signed integers is the most commonly used signed-integer representation.
- It is a simple modification of unsigned integers where the most significant bit is a negative power of 2.

$$v = -2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$



-32768
+2024
-30744

Still a "sign bit"
(It must be "1" for the number to < 0)

Why 2's complement?

- In the two's complement representation for signed integers, the same binary "addition procedure" (mod 2^n) works for adding any combination of positive and negative numbers.
- Don't need a separate "subtraction circuitry" (carries only, no borrows)
  - *The "addition procedure" also handles unsigned numbers!*
  - *In 2's complement adding is "adding" regardless of operand signs.*
  - *You NEVER need to subtract when you use 2's-complement.*

2's complement tricks

- Negation: changing the sign of a number
  - *Invert every bit (i.e. 1 → 0, 0 → 1) and add 1*

Example: $42_{10} = 000000101010_2$

$-42_{10} = 111111010101_2 + 1 =$

111111010110_2

- Sign-Extension - aligning different sized 2's complement integers (for example when adding an 8-bit number to a 32-bit number)
  - *Simply copy the sign bit into higher positions*

Example: 16-bit version of 42: $42_{10} = 0000000000101010_2$

16-bit version of -42: $-42_{10} = 1111111111010110_2$

Two's Complement Overflow

- Overflow: when the result of an operation cannot be represented in the given number of bits

Adding	Overflow occurs when
two positive numbers	the result is negative
two negative numbers	the result is positive
two numbers of opposite signs	will never occur

Two’s Complement Overflow

Take 5 minutes to answer question 5 on your worksheets from last time!

Two's Complement Overflow

5.1

$$\begin{array}{r} 0111_2 \\ +\ 0101_2 \\ \hline \end{array}$$

5.3

$$\begin{array}{r} 1111_2 \\ +\ 0110_2 \\ \hline \end{array}$$

5.2

$$\begin{array}{r} 1000_2 \\ +\ 1001_2 \\ \hline \end{array}$$

5.4

$$\begin{array}{r} 1101_2 \\ +\ 1011_2 \\ \hline \end{array}$$

14

Two's Complement Overflow

5.1

$$\begin{array}{r} {\scriptstyle 111} \\ 0111_2 \\ +\ 0101_2 \\ \hline 1100 \end{array} \longrightarrow -4 \qquad \begin{array}{r} 7 \\ +\ 5 \\ \hline 12 \end{array}$$

Overflow

5.3

$$\begin{array}{r} {\scriptstyle 111} \\ 1111_2 \\ +\ 0110_2 \\ \hline 0101 \end{array} \longrightarrow 5 \qquad \begin{array}{r} -1 \\ +\ 6 \\ \hline 5 \end{array}$$

No Overflow

5.2

$$\begin{array}{r} {\scriptstyle 1} \\ 1000_2 \\ +\ 1001_2 \\ \hline 0001 \end{array} \longrightarrow 1 \qquad \begin{array}{r} -8 \\ +\ -7 \\ \hline -15 \end{array}$$

Overflow

$$\begin{array}{r} {\scriptstyle 1111} \\ 1101_2 \\ +\ 1011_2 \\ \hline 1000 \end{array} \longrightarrow -8 \qquad \begin{array}{r} -3 \\ +\ -5 \\ \hline -8 \end{array}$$

No Overflow

15

Two's Complement Subtraction

$$\begin{array}{rr} & -10 \\ - & 4 \\ \hline & -14 \end{array} \qquad \begin{array}{rr} & -10 \\ + & -4 \\ \hline & -14 \end{array}$$

Two's Complement Subtraction

$$\begin{array}{r} -10 \\ -\quad 4 \\ \hline -14 \end{array} \qquad \begin{array}{r} -10 \\ +\quad -4 \\ \hline -14 \end{array} \qquad \begin{array}{r} 10110_2 \\ -\ 00100_2 \\ \hline \end{array}$$

Two's Complement Subtraction

$$\begin{array}{rr} & -10 \\ - & 4 \\ \hline & -14 \end{array} \qquad \begin{array}{rr} & -10 \\ + & -4 \\ \hline & -14 \end{array} \qquad \begin{array}{rr} & 10110_2 \\ - & 00100_2 \\ \hline & \end{array} \qquad \begin{array}{rr} & 10110_2 \\ + & 11100_2 \\ \hline & 10010_2 \end{array}$$

Two's Complement Subtraction

$$\begin{array}{r} -10 \\ -\ \ 4 \\ \hline -14 \end{array} \qquad \begin{array}{r} -10 \\ +\ -4 \\ \hline -14 \end{array} \qquad \begin{array}{r} 10110_2 \\ -\ 00100_2 \\ \hline \end{array} \qquad \begin{array}{r} 10110_2 \\ +\ 11100_2 \\ \hline 10010_2 \end{array}$$

$$\begin{array}{r} -10 \\ -\ -4 \\ \hline -6 \end{array} \qquad \begin{array}{r} -10 \\ +\ \ 4 \\ \hline -6 \end{array} \qquad \begin{array}{r} 10110_2 \\ -\ 11100_2 \\ \hline \end{array} \qquad \begin{array}{r} 10110_2 \\ +\ 00100_2 \\ \hline 11010_2 \end{array}$$

Two's Complement Subtraction

$$\begin{array}{r} -10 \\ -\ \ 4 \\ \hline -14 \end{array} \qquad \begin{array}{r} -10 \\ +\ -4 \\ \hline -14 \end{array} \qquad \begin{array}{r} 10110_2 \\ -\ 00100_2 \\ \hline \end{array} \qquad \begin{array}{r} 10110_2 \\ +\ 11100_2 \\ \hline 10010_2 \end{array}$$

No extra logic (aka circuitry) needed for subtraction! We can just reuse the logic for addition for subtraction!

$$\begin{array}{r} -10 \\ -\ -4 \\ \hline -6 \end{array} \qquad \begin{array}{r} -10 \\ +\ -4 \\ \hline -6 \end{array} \qquad \begin{array}{r} 10110_2 \\ +\ 00100_2 \\ \hline 11010_2 \end{array}$$

Bitwise Operations

- Apply the operation to each individual bit position

$$\sim 1010_2 = 0101_2$$

$$\begin{array}{r} 1010_2 \\ \& \; 1100_2 \\ \hline 1000_2 \end{array}$$

$$\begin{array}{r} 1010_2 \\ | \; 1100_2 \\ \hline 1110_2 \end{array}$$

$$\begin{array}{r} 1010_2 \\ \wedge \; 1100_2 \\ \hline 0110_2 \end{array}$$

Bitwise Operations: Fill in the blank!

- Fill in the following blanks using the options below. n is an integer. There are multiple correct answers for each question.

  - *A. n ____ _____= n*
  - *B. n ____ _____= 0b 00…00*
  - *C. n ____ _____= 0b 11…11*

First blank	Second blank
AND	n
OR	~n
XOR	0b 00…00
	0b 11…111

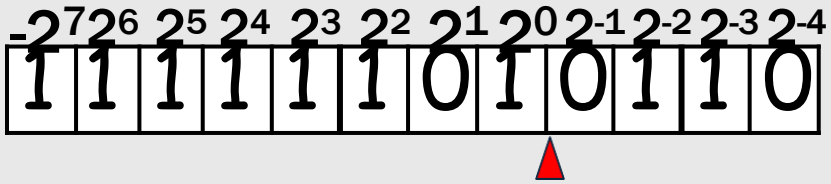
Bitwise Operations: Fill in the blank!

- *A. n ___ _____ = n*
  - OR 0b 00…00
  - AND 0b 11…1
  - XOR 0b 00…00
  - AND n
  - OR n
- *B. n ___ _____ = 0b 00…00*
  - AND 0b 00…00
  - AND ~n
  - XOR n
- *C. n ___ _____ = 0b 11…11*
  - OR 0b11…11
  - OR ~n
  - XOR ~n

First blank	Second blank
AND	n
OR	~n
XOR	0b 00…00
	0b 11…111

Fixed-Point Numbers

- You can always assume that the boundary between 2 bits is a "binary point".
- If you align binary points between addends, there is no effect on how operations are performed.



$$11111101.0110 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3}$$
$$= -128 + 64 + 32 + 16 + 8 + 4 + 1 + 0.25 + 0.125$$
$$= -2.625$$

OR

$$11111101.0110 = -42 \times 2^{-4}$$
$$= -42 / 16$$
$$= -2.625$$

Repeated Binary Fractions

- Not all fractions can be represented exactly using a finite representation.
  - *Ex: 1/3 → 0.333333….*
- In binary, many fractions that you've grown attached to require an infinite number of bits to represent exactly.

Example: $1/10 = 0.1_{10} = 0.00011..._2 = 0.19..._{16}$

$1/5 = 0.2_{10} = 0.0011..._2 = 0.3..._{16}$

$1/3 = 0.3_{10} = 0.01..._2 = 0.5..._{16}$

Finite Representations!

- Everything that a realizable computer does is limited by a finite set of bits.
- You may have grown used to infinite digits in math courses…

```
          ...000000000042.0000000000...
          ...000000000000.0000000000...001000
10000000...000000000000.0
```

- …However, the concept an infinite supply of zero digits is conceptually elegant, but difficult to physically implement

Overflow: Side Effect of being Finite

- **Overflow: when the result of an operation cannot be represented in the given number of bits**

1. $32767_{10} + 1_{10} = -32768_{10}$

$$\begin{array}{r} 0111\ 1111\ 1111\ 1111_2 \\ +\ 0000\ 0000\ 0000\ 0001_2 \\ \hline 1000\ 0000\ 0000\ 0000_2 \end{array}$$

2. $-20000_{10} - 20000_{10} = 25536_{10}$

$$\begin{array}{r} 1011\ 0001\ 1110\ 0000_2 \\ +\ 1011\ 0001\ 1110\ 0000_2 \\ \hline 1\ 0110\ 0011\ 1100\ 0000_2 \end{array}$$

3. $-32768_{10} = -32768_{10}$

A certain number can't be negated

$$\begin{array}{r} 1000\ 0000\ 0000\ 0000_2 \\ 0111\ 1111\ 1111\ 1111_2 \\ +\ 0000\ 0000\ 0000\ 0001_2 \\ \hline 1000\ 0000\ 0000\ 0000_2 \end{array}$$

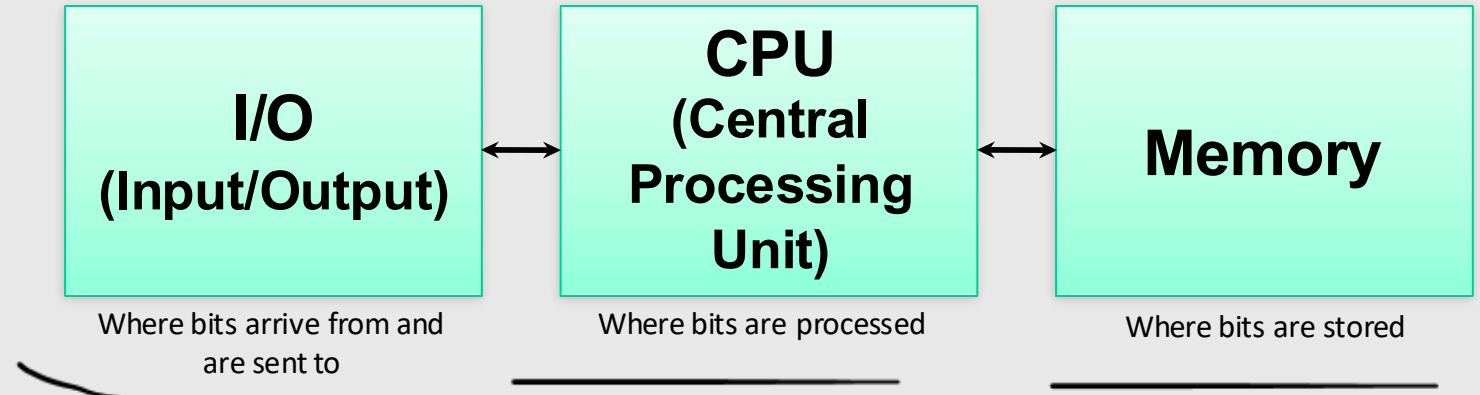
Flip bits first, then add one 1 negate…

Summarizing our review so far…

- 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
  - Overflow!
- Floating-point numbers have separate fraction and exponent components.

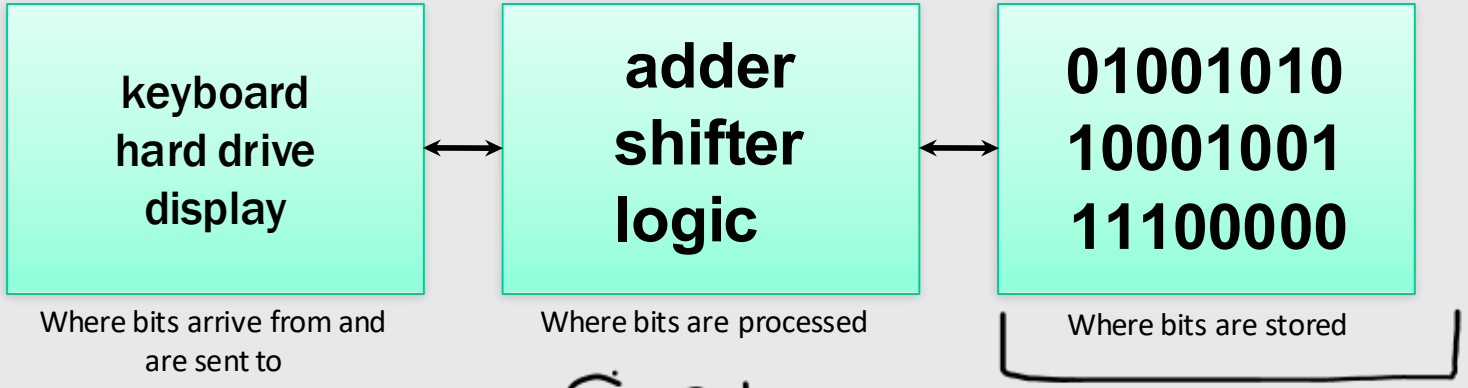
COMPUTER ORGANIZATION!

Let's Dig into *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.

Let's Dig into *Computer Organization*!



- 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 do we mean by “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*

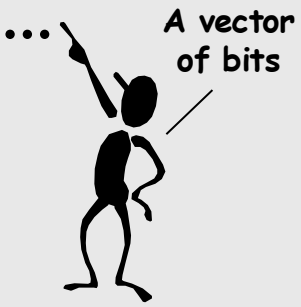
How Memory is Organized

- A group of bits!
- *Groups of bits* can represent various types of data
  - *Integers, Signed integers. Floating-point values, Strings, Pixels*
- 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 ….

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.

Some questions computer architects (you!) may ponder…

- 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

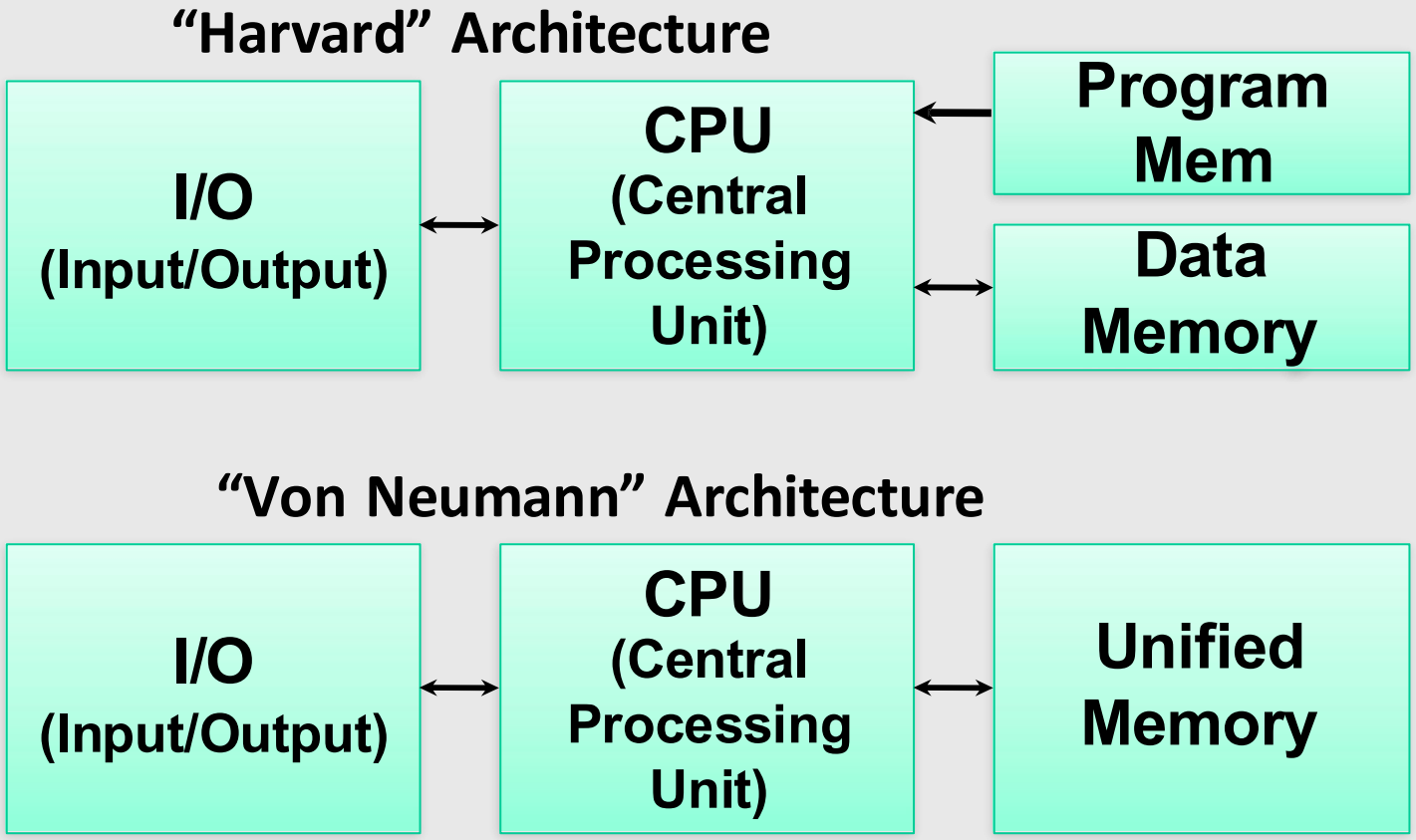
Instructions are also stored in memory!

- 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

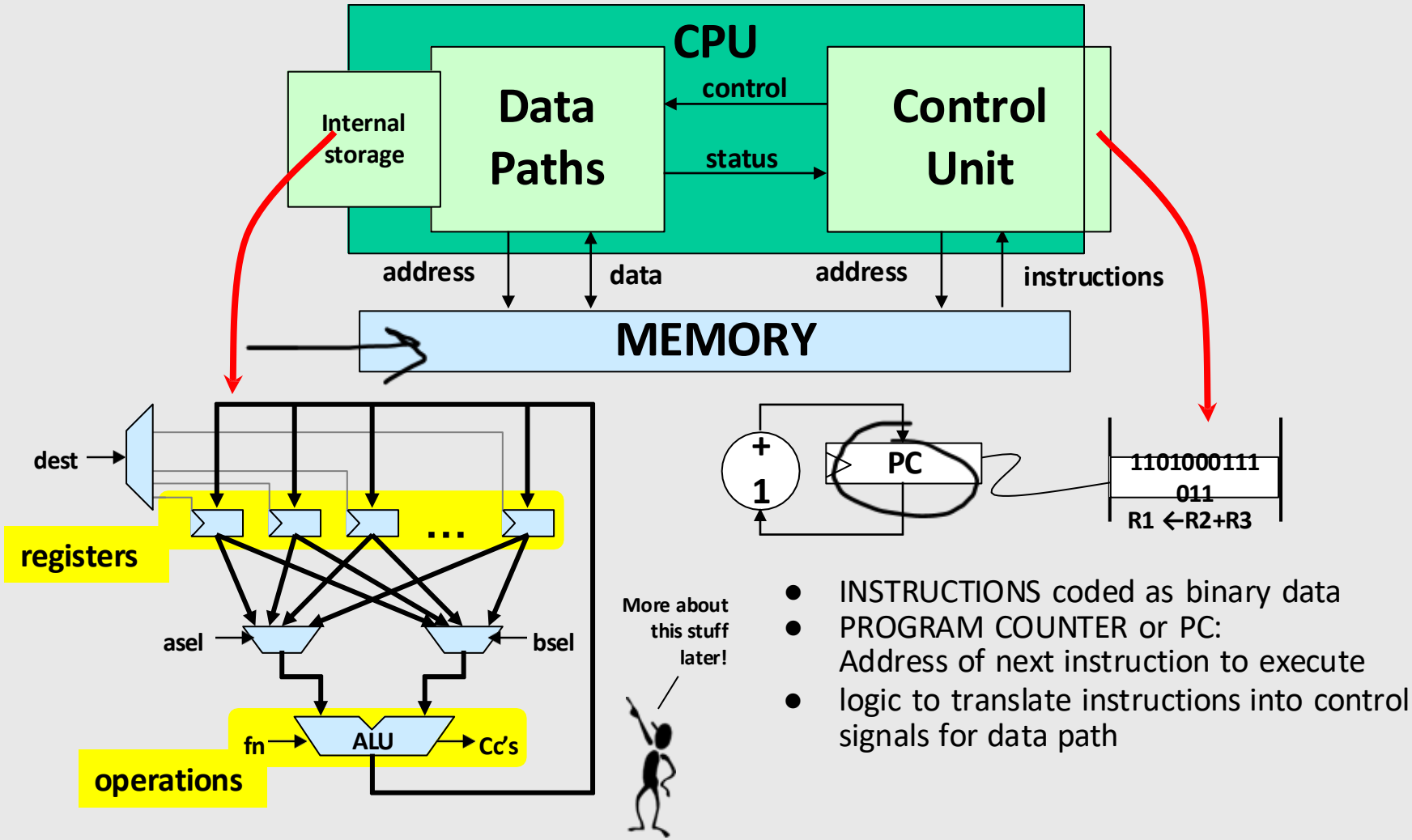
There is a commonly recurring debate over whether “data” and “instructions” should be mixed. Leads to two common flavors of computer architectures



Aside: Undergrad Systems Reading Group!! ☺

- Thursdays at 5pm!
- FB141
- Food provided!
- All are welcome!!
  - *No background in CS research necessary*
- If you are interested in being added to the mailing list, send me an email!

Anatomy of a von Neumann Computer



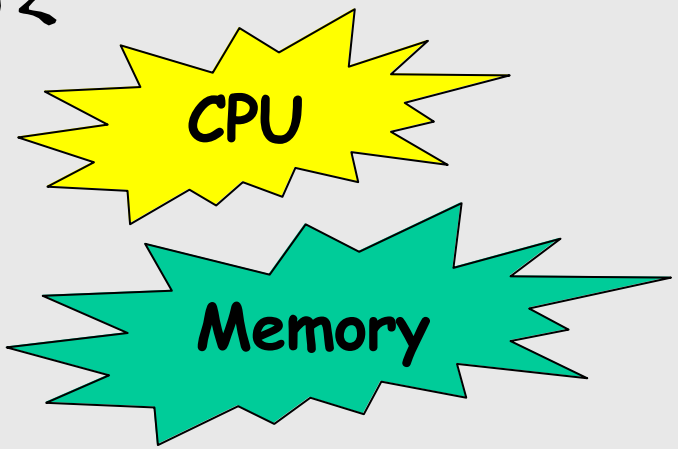
- INSTRUCTIONS coded as binary data
- PROGRAM COUNTER or PC:
  Address of next instruction to execute
- logic to translate instructions into control signals for data path

Anatomy of an Instruction

Nearly all instructions can be made to fit a common template…

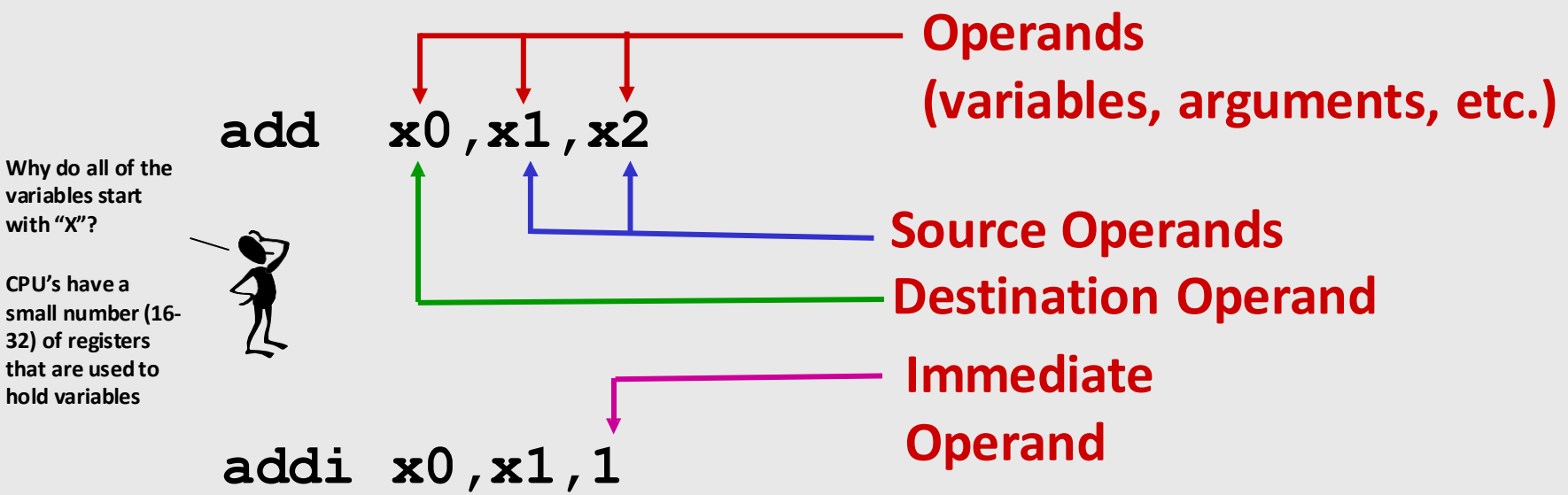
Issues remaining …

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



Anatomy of an Instruction

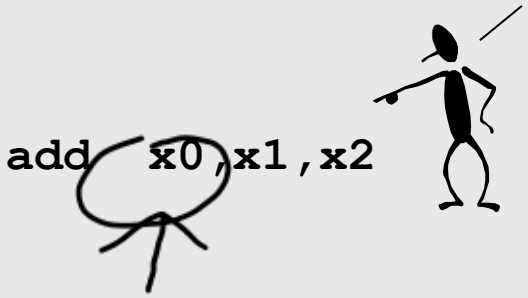
- Computers execute a set of primitive operations called instructions
- Instructions specify an operation and its operands (arguments of the operation)
- Types of operands: destination, source, and immediate



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



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

```
int x0, x1, x2;
x0 = x1 + x2;
```

Instead of:

```
x1 + x2 = x0;
```

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

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

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

      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*

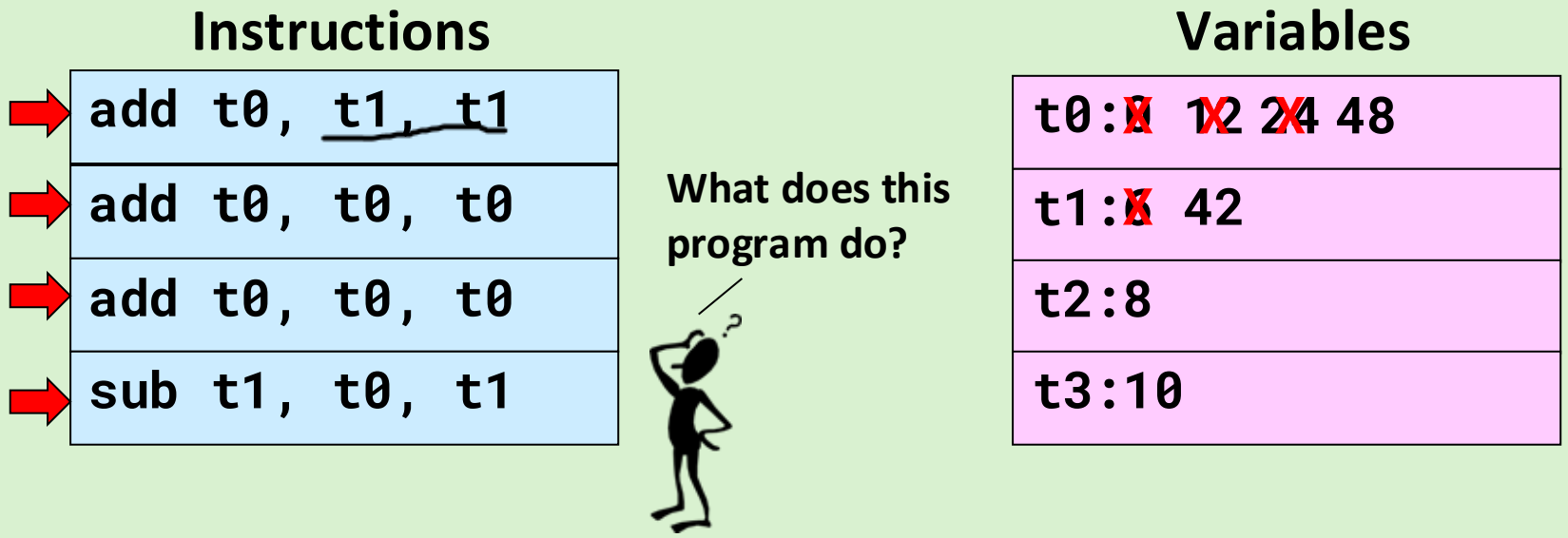
```
x:    .word 0
y:    .word 0
c:    .word 123456


      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
```

```
0x00000000
0x00000000
0x0001E240

...

0x0001E240
0x0001A283
0xFFD28293
0x0081A383
0x00730333
0x026282B3
0x0051A023
```

A Series of Instructions

Generally…

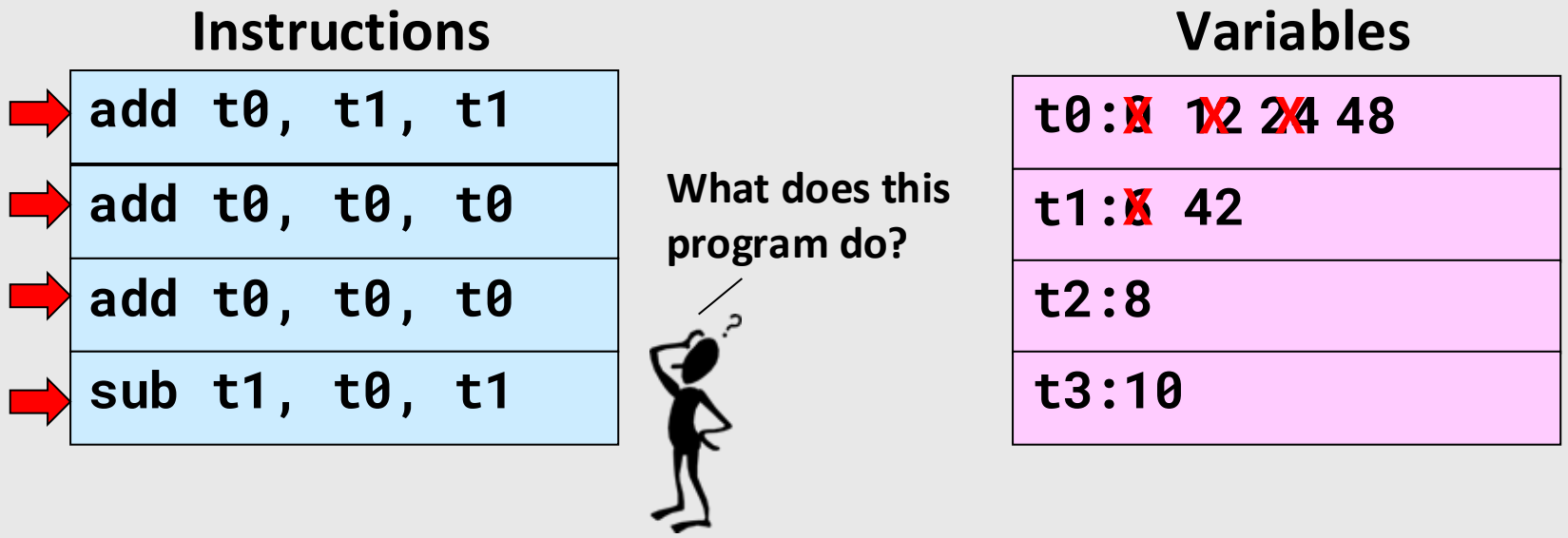
- Instructions are retrieved sequentially from memory
- An instruction executes to completion before the next instruction is started
- But, there are exceptions to these rules



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

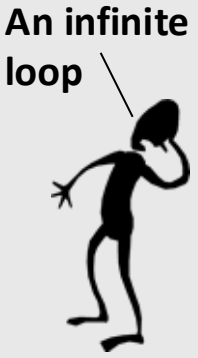


Looping the Flow

- 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
```

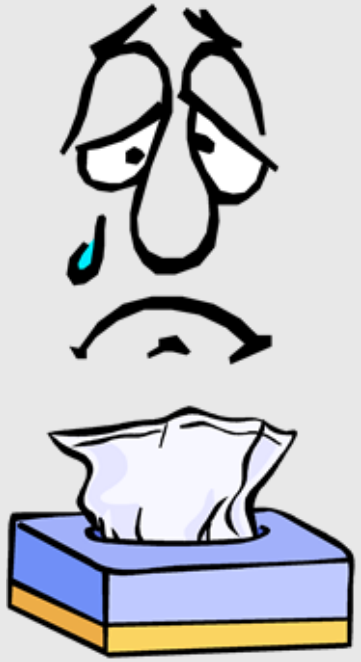


Variables

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

Open Issues in Our Simple Model?

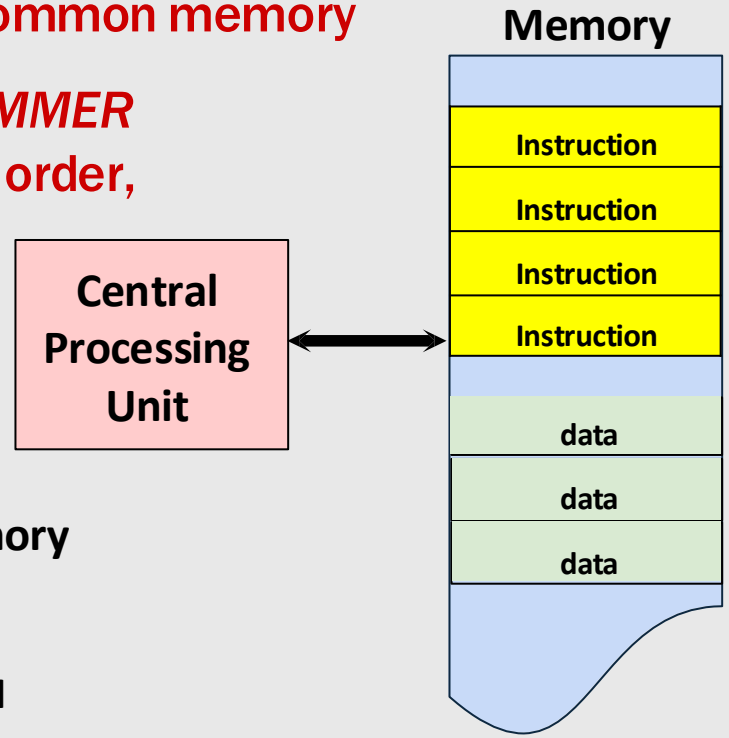
- WHERE in memory are INSTRUCTIONS stored?
- HOW are instructions represented?
- WHERE are VARIABLES stored?
- What are LABELs? How do they relate to where instructions are stored?
- How about more complicated data types?
  - *Arrays?*
  - *Data Structures?*
  - *Objects?*
- Where does a program start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

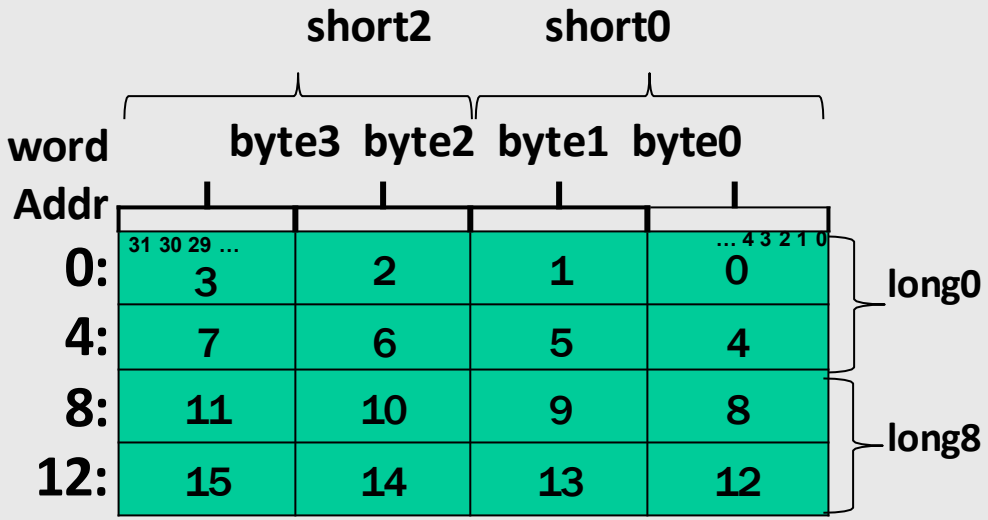


CPU fetches and executes instructions from memory

- 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

RISC-V Memory Details

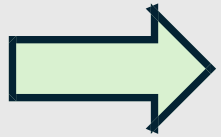
- Memory locations are addressable in different sized chunks
  - *8-bit chunks (bytes)*
  - *16-bit chunks (shorts)*
  - *32-bit chunks (words)*
  - *64-bit chunks (longs/doubles)*
- We also frequently need access to individual bits! (Instructions help with this)
- Every BYTE has a unique address (RISC-V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

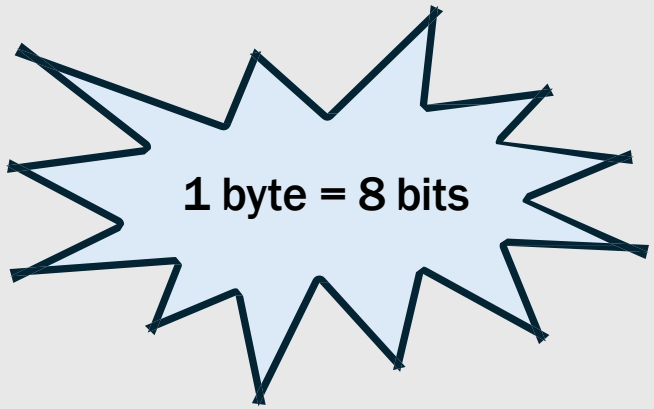


211 REVIEW: MEMORY

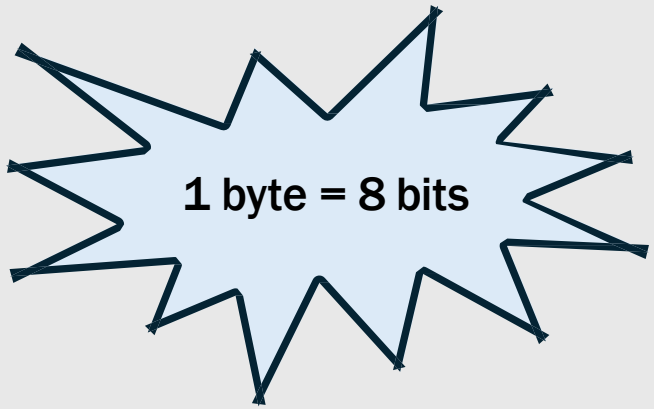
Storing Data in Memory

Each of these rows can store one byte of data

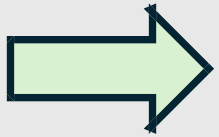




Storing Data in Memory



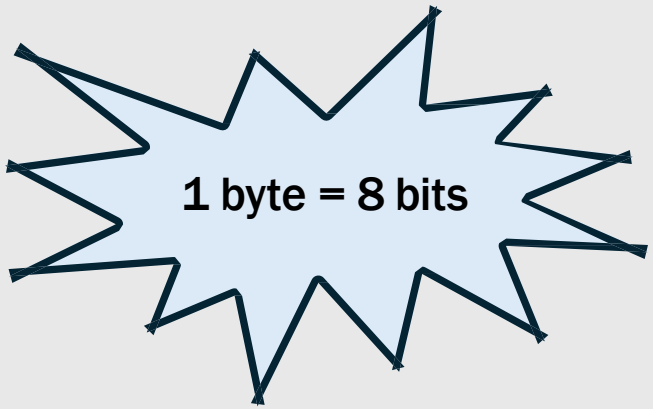
Each of these rows can store one byte of data



0b0000 0101
0b0000 0100
0b1111 1110

Let's say we want to store the number 5 in the top row, 4 in the next row, and -2 in the next row.

Byte Addresses



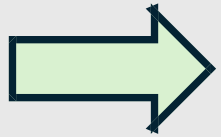
To reference each of these locations, we use something called a byte address.

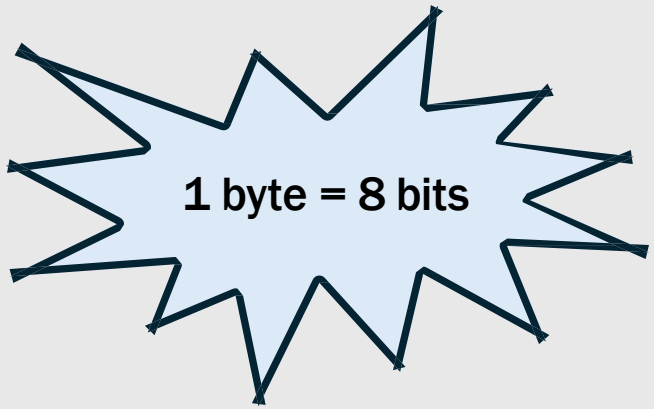
Note that these addresses are not stored anywhere!

Byte Address	
0	0b0000 0101
1	0b0000 0100
2	0b1111 1110
3	
4	
5	
6	
7	
8	
9	
10	
11	

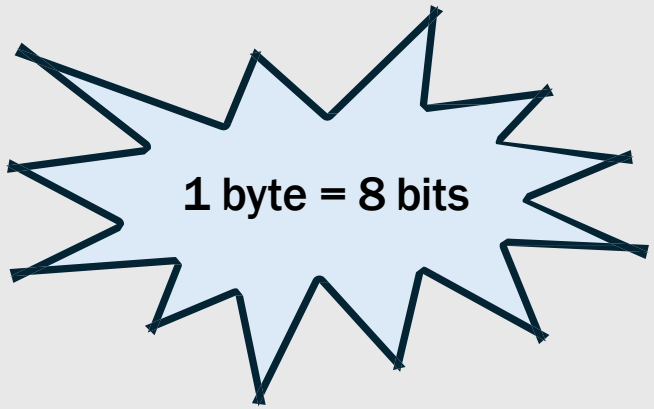
Storing Data in Memory

Each of these rows can store one byte of data

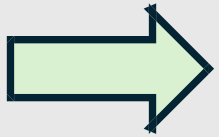




Storing Data in Memory



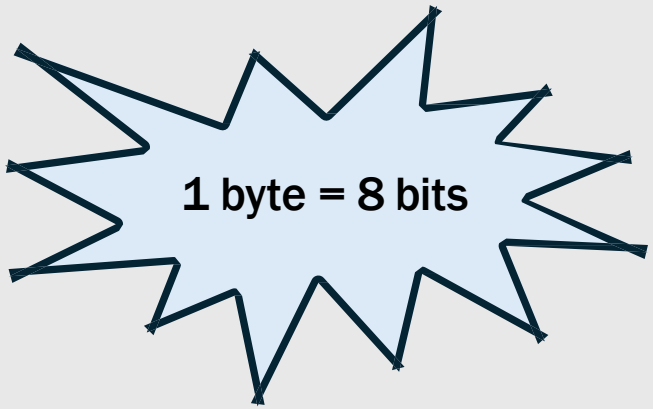
Each of these rows can store one byte of data



0b0000 0101
0b0000 0100
0b1111 1110

Let's say we want to store the number 5 in the top row, 4 in the next row, and -2 in the next row.

Byte Addresses



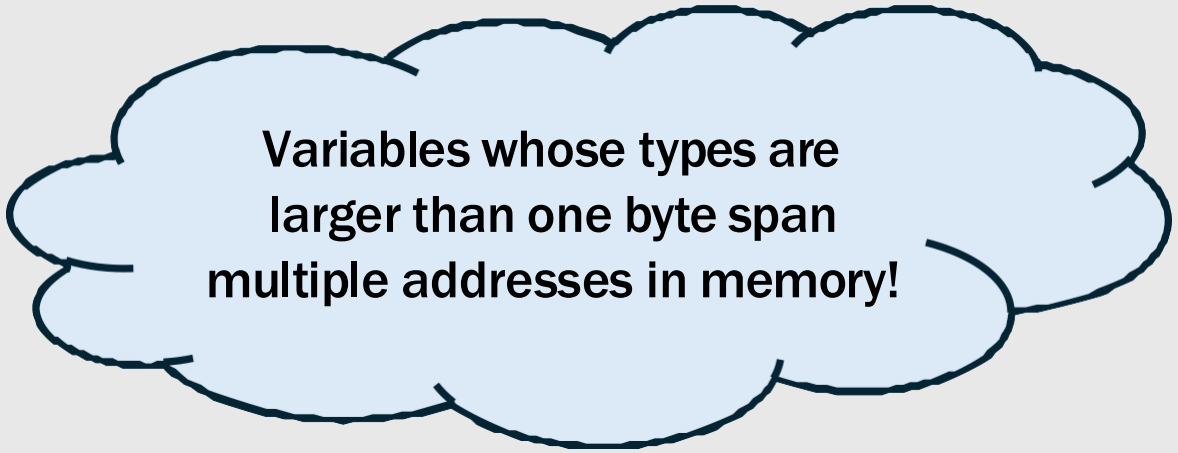
To reference each of these locations, we use something called a byte address.

Note that these addresses are not stored anywhere!

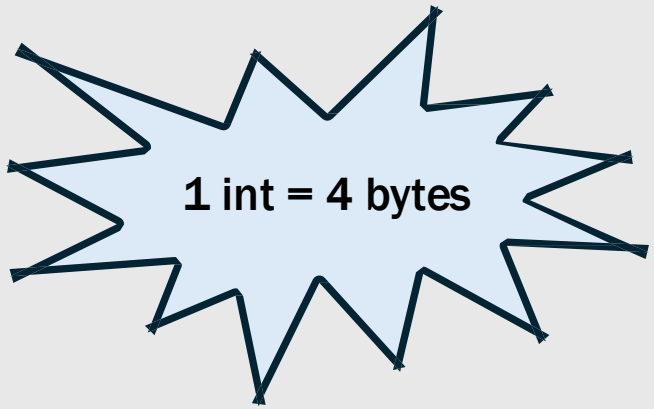
Byte Address	
0	0b0000 0101
1	0b0000 0100
2	0b1111 1110
3	
4	
5	
6	
7	
8	
9	
10	
11	

Storing Larger Data

- What if I want to store an integer?
  - *sizeof(int) = 4 bytes*



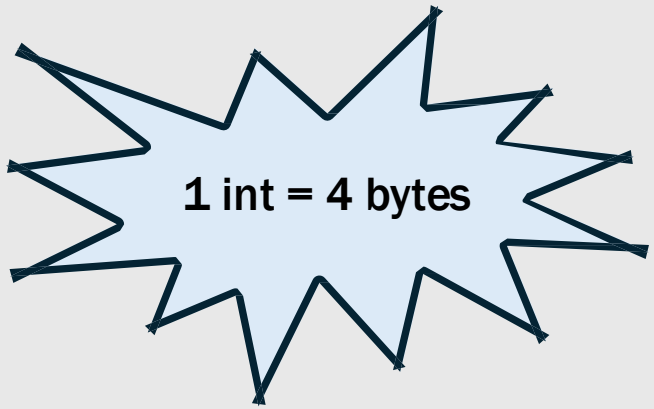
Storing Larger Data



- Let's say I want to store an integer whose value is 329,421

329421 = 0b 0000 0000 0000 0101 0000 0110 1100 1101

Storing Larger Data



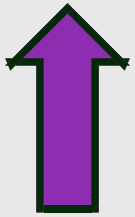
- Let's say I want to store an integer whose value is 329,421

329421 = 0b 0000 0000 0000 0101 0000 0110 1100 1101

Byte Address	Data
0	
1	
2	
3	

MSB and LSB

0b 0000 0000 0000 0101 0000 0110 1100 1101



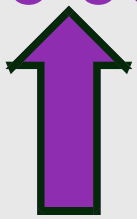
Most Significant Byte
(MSB)

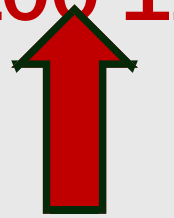
Least Significant Byte
(LSB)

Little Endian Ordering

- The least significant byte of the integer goes in the lowest address

0b 0000 0000 0000 0101 0000 0110 1100 1101





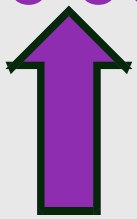
Most Significant Byte (MSB)

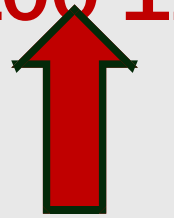
Least Significant Byte (LSB)

Little Endian Ordering

- The least significant byte of the integer goes in the lowest address

0b 0000 0000 0000 0101 0000 0110 1100 1101





Most Significant Byte (MSB)

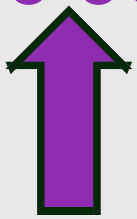
Least Significant Byte (LSB)

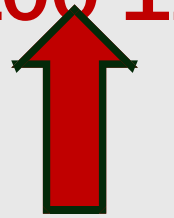
Byte Address	Data
0	1100 1101
1	0000 0110
2	0000 0101
3	0000 0000

Big Endian Ordering

- The least significant byte of the integer goes in the largest address

0b 0000 0000 0000 0101 0000 0110 1100 1101





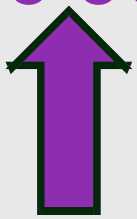
Most Significant Byte (MSB)

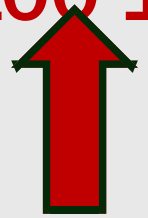
Least Significant Byte (LSB)

Big Endian Ordering

- The least significant byte of the integer goes in the largest address

0b 0000 0000 0000 0101 0000 0110 1100 1101





Most Significant Byte (MSB)

Least Significant Byte (LSB)

Byte Address	Data
0	0000 0000
1	0000 0101
2	0000 0110
3	1100 1101

Little Endian Ordering

- The least significant byte of the integer goes in the lowest address

0b 0000 0000 0000 0101 0000 0110 1100 1101