Behind the Curtain

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

Introduce TA
Friday 1st lab! (1:30-3:30)
(Do Prelab before)
1st Problem set on Thurs

Signed-Numbers

- The obvious method is to encode the sign of the integer using one bit. Conventionally, the most significant bit is used for the sign. This encoding for signed integers is called the SIGNED MAGNITUDE representation.

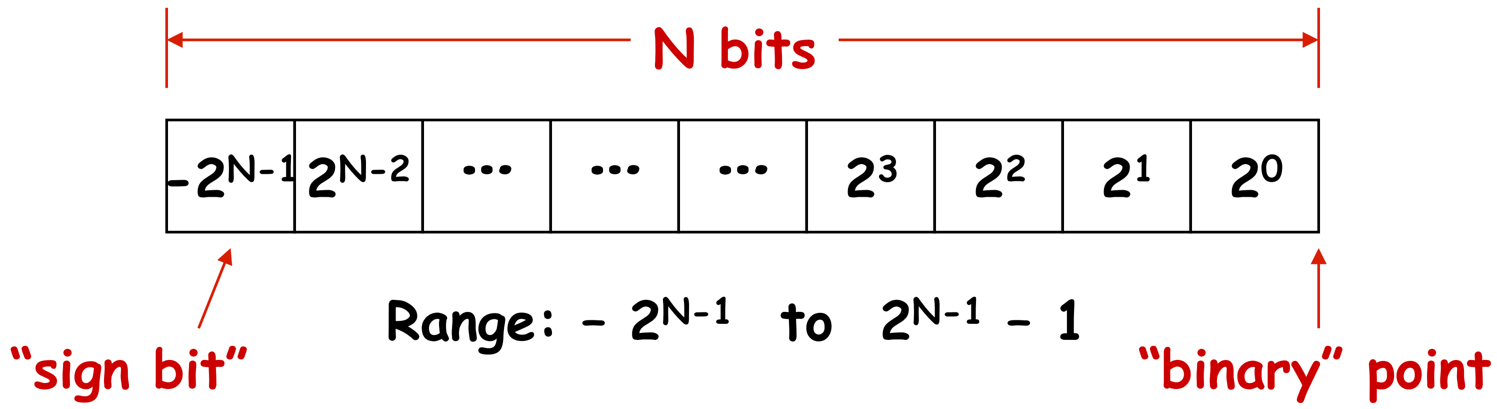
$$v = -1^S \sum_{i=0}^{n-2} 2^i \, b_i$$

S	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	1	1	0	1	0	0	0	0

2000 **-2000**

- The Good:
  - Easy to negate, find absolute value
- The Bad:
  - Add/subtract is complicated; depends on the signs
  - Two different ways of representing a 0
- Not used that frequently in practice
  - with one important exception

2's Complement Integers



Range: -2^{N-1} to $2^{N-1}-1$

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 considered negative.

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

8-bit 2's complement example:

11010110 $= -2^7 + 2^6 + 2^4 + 2^2 + 2^1$

$= -128 + 64 + 16 + 4 + 2 = -42$

Why 2's Complement?

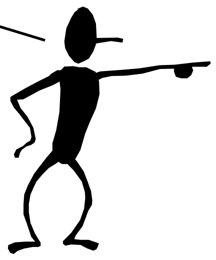
If we use a two's complement representation for signed integers, the same binary addition mod 2^n procedure will work for adding positive and negative numbers (don't need separate subtraction rules). The same procedure will also handle unsigned numbers!

When using signed magnitude representations, adding a negative value really means to subtract a positive value. However, in 2's complement, adding is adding regardless of sign. In fact, you NEVER need to subtract when you use a 2's complement representation.

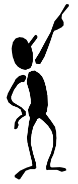
Example:

$$\begin{array}{rrcr} & 55_{10} & = & 00110111_2 \\ + & 10_{10} & = & 00001010_2 \\ \hline & 65_{10} & = & 01000001_2 \end{array}$$

$$\begin{array}{rrcr} & 55_{10} & = & 00110111_2 \\ + & -10_{10} & = & 11110110_2 \\ \hline & 45_{10} & = & 100101101_2 \end{array}$$



ignore this overflow



2's Complement Tricks

- Negation – changing the sign of a number
  - First invert every bit (i.e. 1 → 0, 0 → 1)
  - Add 1

  Example: 20 = 00010100, -20 = 11101011 + 1 = 11101100

- Sign-Extension – aligning different sized 2's complement integers
  - Simply copy the sign bit into higher positions
- 16-bit version of 42 = 0000 0000 0010 1010
- 8-bit version of -2 = 1 1 1 1 1 1 1 1 1 1 1 1 1 110

CLASS EXERCISE

10’s-complement Arithmetic

(You’ll never need to borrow again)

Step 1) Write down two 3-digit numbers that you want to subtract

Step 2) Form the 9’s-complement of each digit in the second number (the subtrahend)

Step 3) Add 1 to it (the subtrahend)

Step 4) Add this number to the first

Step 5) If your result was less than 1000, form the 9’s complement again and add 1 and remember your result is negative else subtract 1000

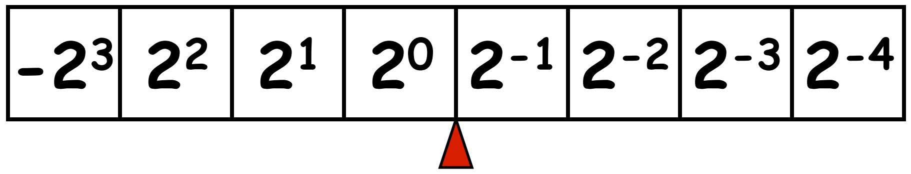
Helpful Table of the 9’s complement for each digit

0 → 9	
1 → 8	
2 → 7	
3 → 6	
4 → 5	
5 → 4	
6 → 3	
7 → 2	
8 → 1	
9 → 0	

What did you get? Why weren’t you taught to subtract this way?

Fixed-Point Numbers

By moving the implicit location of the "binary" point, we can represent signed fractions too. This has no effect on how operations are performed, assuming that the operands are properly aligned.



1101.0110 $= -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3}$

$= -8 + 4 + 1 + 0.25 + 0.125$

$= -2.625$

OR

1101.0110 $= -42 * 2^{-4} = -42/16 = -2.625$

Repeated Binary Fractions

Not all fractions can be represented exactly using a finite representation. You've seen this before in decimal notation where the fraction 1/3 (among others) requires an infinite number of digits to represent (0.3333…). –

In Binary, a great many fractions that you've grown attached to require an infinite number of bits to represent exactly.

EX: $1 / 10 = 0.1_{10} = .00011\overline{0011}\ldots_2$

$1 / 5 = 0.2_{10} = .\overline{0011}\ldots_2 = 0.\overline{3}\ldots_{16}$

Bias Notation

- There is yet one more way to represent signed integers, which is surprisingly simple. It involves subtracting a fixed constant from a given unsigned number. This representation is called "Bias Notation".

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

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	1	1	0

EX: (Bias = 127)

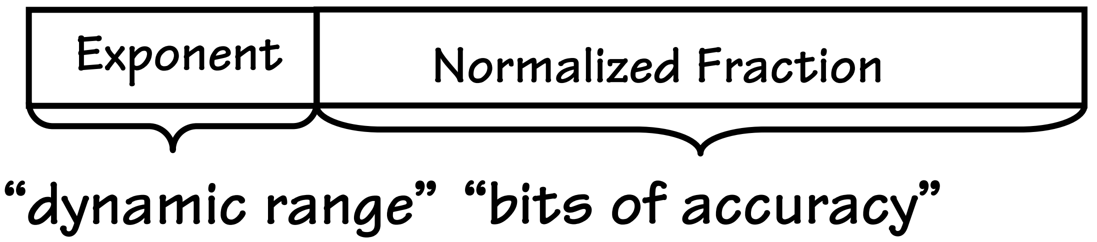
$$\begin{array}{rcr} 6 * 1 & = & 6 \\ 13 * 16 & = & 208 \\ & & -\,127 \\ \hline & & 87 \end{array}$$

Why? Monotonicity

Floating Point Numbers

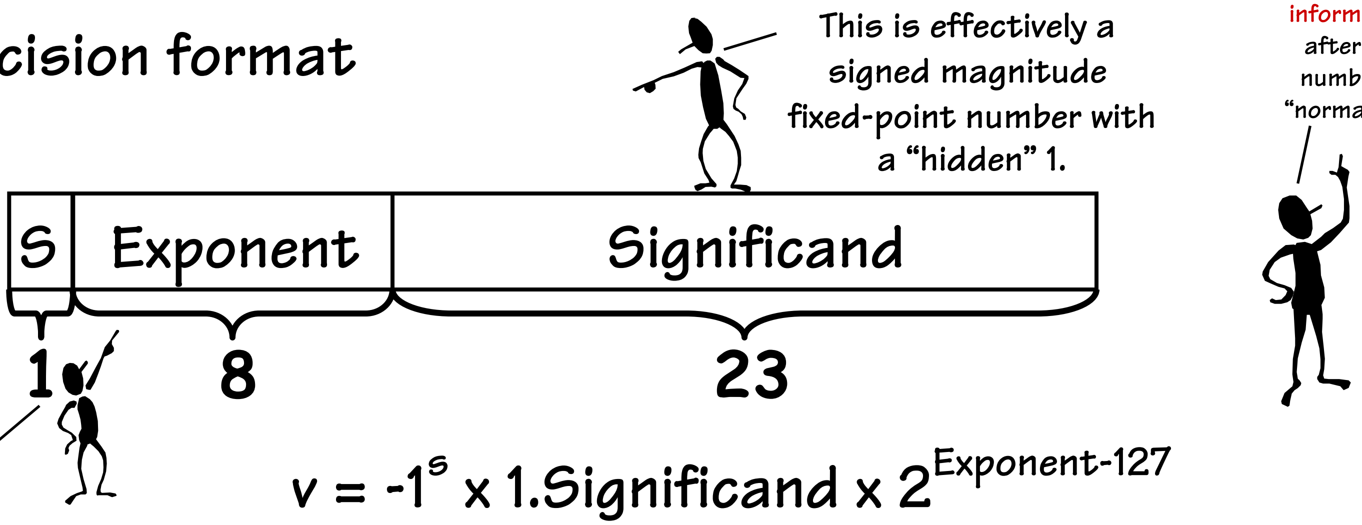
Another way to represent numbers is to use a notation similar to Scientific Notation. This format can be used to represent numbers with fractions (3.90×10^{-4}), very small numbers (1.60×10^{-19}), and large numbers (6.02×10^{23}). This notation uses two fields to represent each number. The first part represents a normalized fraction (called the significand), and the second part represents the exponent (i.e. the position of the "floating" binary point).

$$Normalized \quad Fraction \times 2^{Exponent}$$



IEEE 754 Format

- Single precision format



This is effectively a signed magnitude fixed-point number with a "hidden" 1.

The 1 is hidden because it provides no information after the number is "normalized"

The exponent is represented in bias 127 notation. Why?

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

- Example

$42.75 = 00101010.11000000_2$

Normalize: $001.010101100000_2 \times 2^{5}$

(127+5)

$0\ 10000100\ 01010110000000000000000000_2$

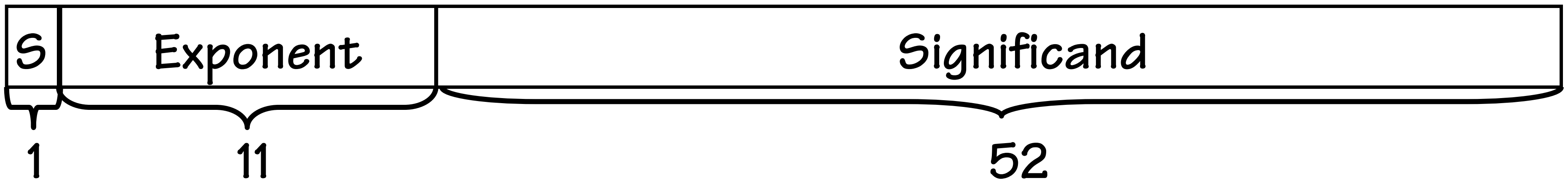
$0100\ 0010\ 0010\ 1011\ 0000\ 0000\ 0000\ 0000_2$

$42.75 = 0x422B0000_{16}$

IEEE 754 Format

- Single precision limitations
  - A little more than 7 decimal digits of precision
  - minimum positive normalized value: ~1.18×10^{-38}
  - maximum positive normalized value: ~3.4×10^{38}
- Inaccuracies become evident after multiple single precision operations

- Double precision format



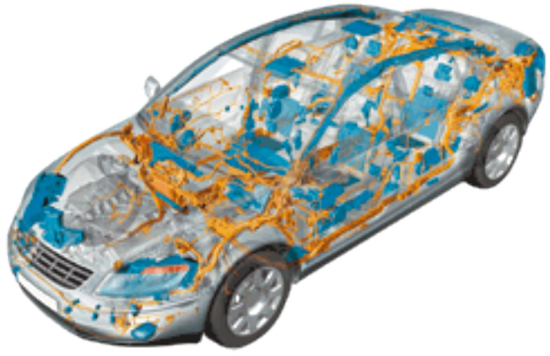
$$v = -1^{s} \times 1.\text{Significand} \times 2^{\text{Exponent}-1023}$$

Summary

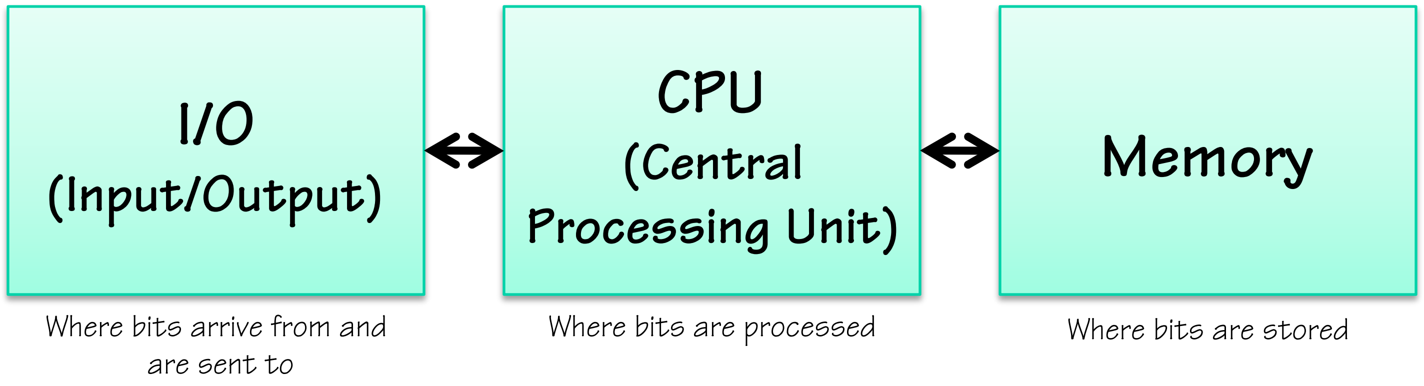
1) Selecting the encoding of information has important implications on how this information can be processed, and how much space it requires.
2) Computer arithmetic is constrained by finite representations, this has advantages (it allows for complement arithmetic) and disadvantages (it allows for overflows, numbers too big or small to be represented).
3) Bit patterns can be interpreted in an endless number of ways, however important standards do exist
   - Two's complement
   - IEEE 754 floating point

Computers Everywhere

- The computers we are used to
  - Desktops
  - Laptops
  - Embedded processors
    - Cars
    - Light bulbs
    - Mobile phones
    - Toasters, irons, wristwatches, happy-meal toys

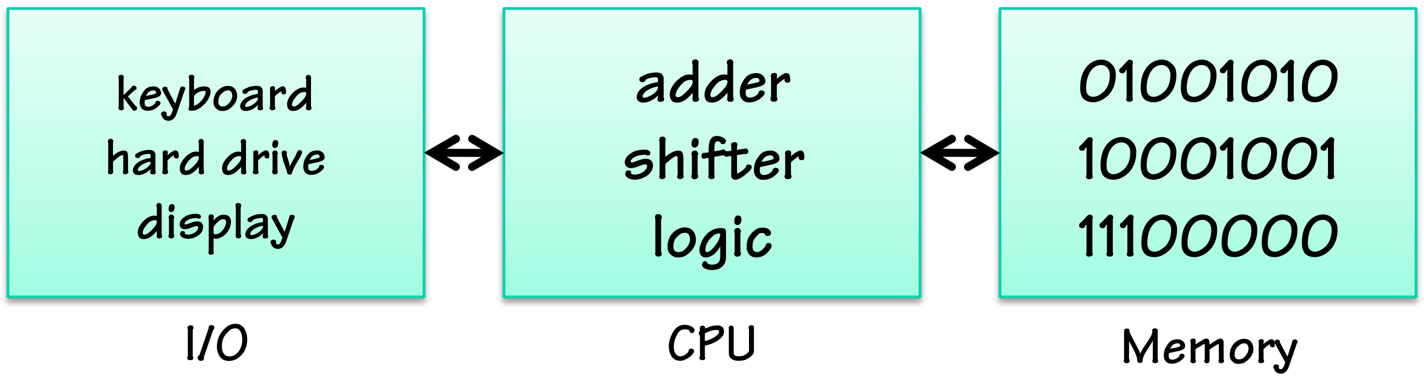


Computer Organization



- 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
    - must convert 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 in "addressable" units, such as bytes or words
    - every memory unit has an "address" and "contents", like a mailbox
  - Central Processing Unit
    - besides processing, it also coordinates data's movements between units

What Sorts of Processing?

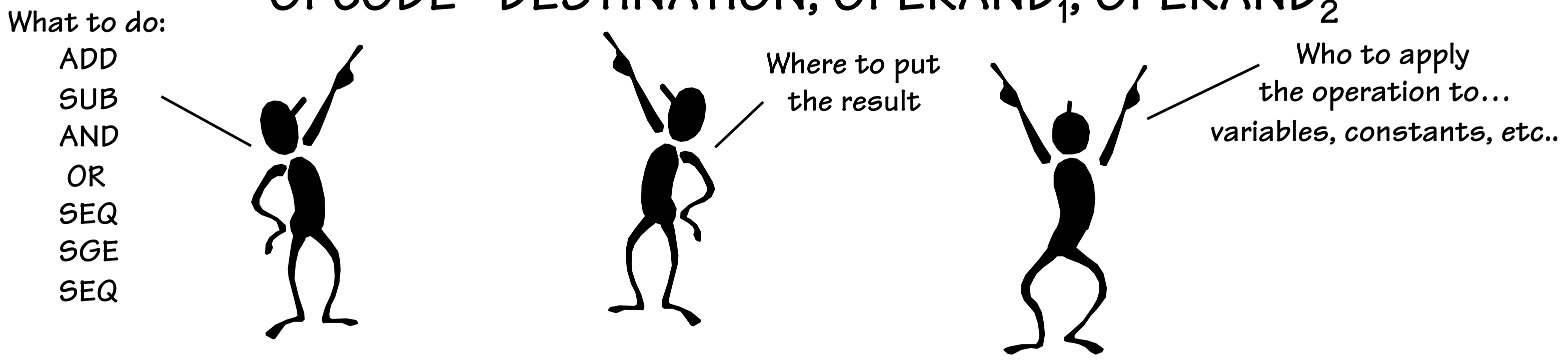
A CPU performs low-level operations called INSTRUCTIONS

- 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

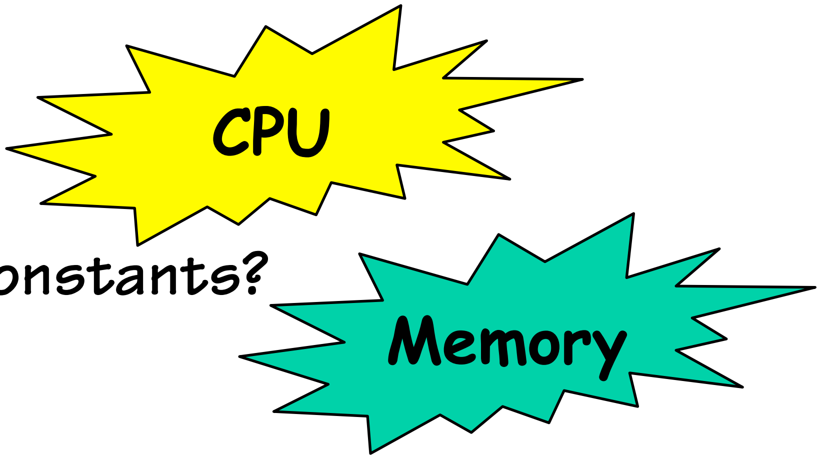
Nearly all instructions can be made to fit a common template

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



Issues remaining ...

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



Memory Concepts

- Memory is divided into "addressable" blocks, each with an address (like an array with indices)
- Addressable blocks are usually larger than a bit, typically 8, 16, 32, or 64 bits
- Each address has variable "contents"
- 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	0c3c1d7fff
9	0x37bdfffc
10	0x24040090
11	0x0c00000e
12	0x1000ffff
13	-100
14	0x00004020
15	0x20090001

One More Thing…

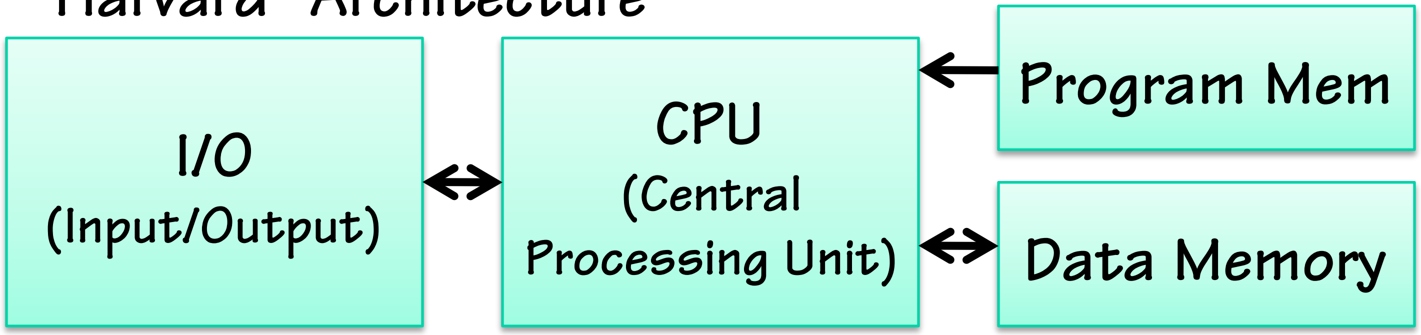
- INSTRUCTIONS for the CPU are stored in memory along with data
- CPU fetches instructions, decodes them and then performs their implied operation
- Mechanism inside the CPU directs which instruction to get next.
- They appear in memory as a string of bits that are typically uniform in size
- Their encoding 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: lui $sp, 0x7fff

Address	Contents
0	42
1	3.141592
2	“Lee “
3	“Hart”
4	“Bud “
5	“Levi”
6	“le “
7	2
8	lui $sp,0x7fff
9	ori $sp,$sp,0x7fff
10	addiu $a0,$0,144
11	jal 0x0000000e
12	beq $0,$0,0x0c
13	-100
14	add $t0,$0,$0
15	addi $t1,$0,1

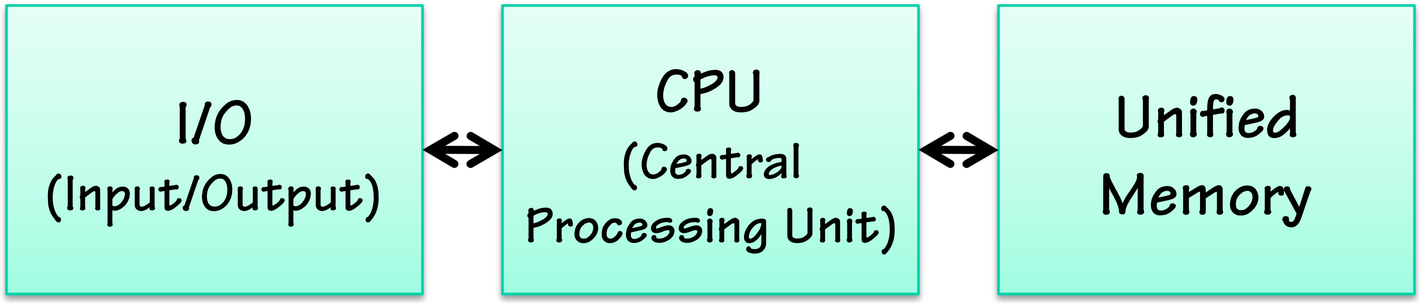
A Bit of History

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

“Harvard” Architecture

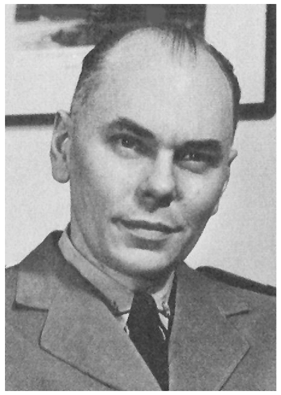


“Von Neumann” Architecture



A Bit of History

- Harvard Architecture
  - Instructions and data do 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, compliers, 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)


Howard Aiken: Architect of the Harvard Mark 1

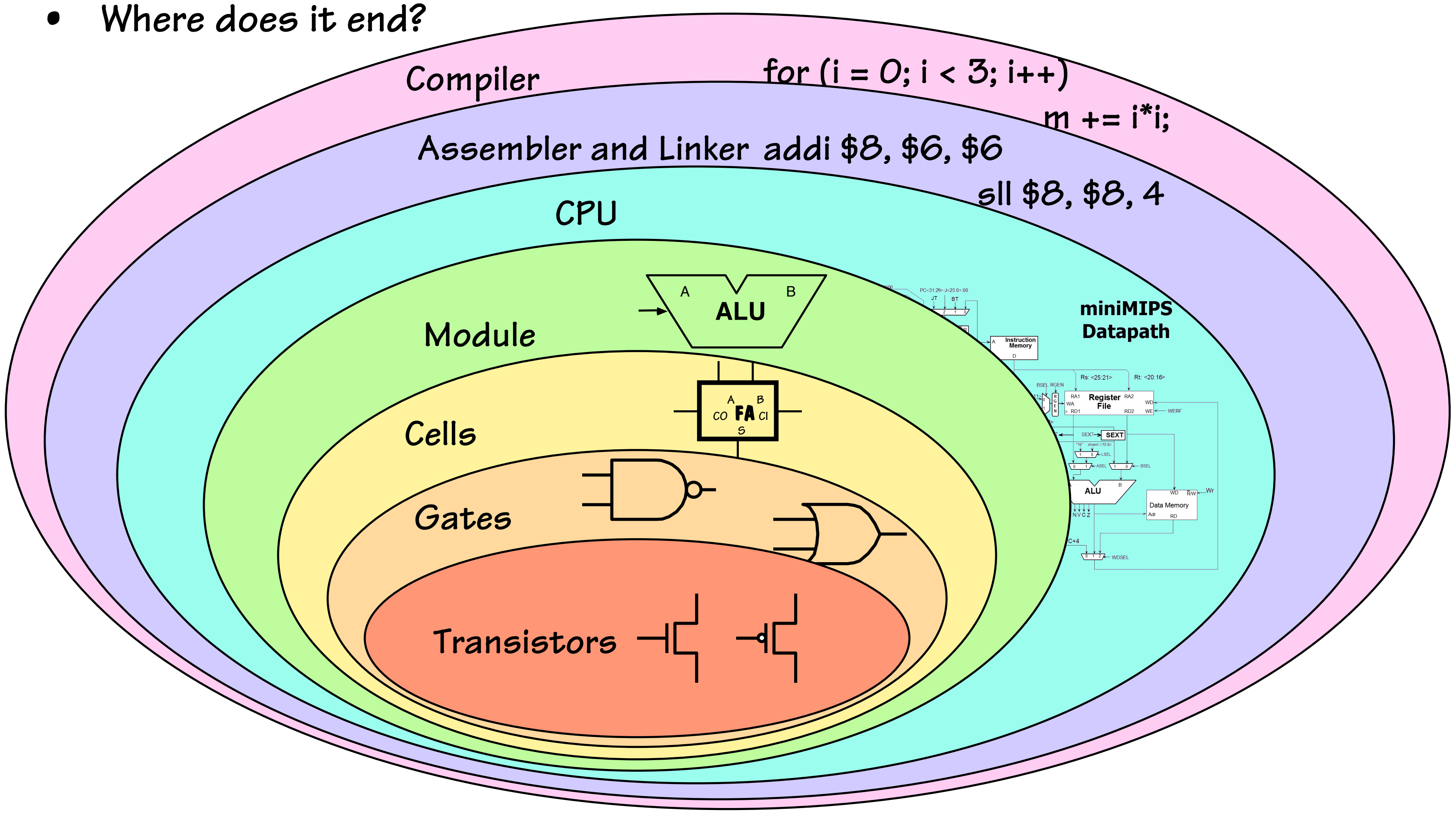
A Bit of History

John Von Neumann: *Proponent of unified memory architecture*

- Von Neumann Architecture
  - Instructions and data are indistinguishable bits in a common memory that share a common "word size" and "address space"
  - 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 "Native" programs to create new programs (assemblers and compliers)
    - 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.

Computer Systems

- What is a computer system?
- Where does it start?
- Where does it end?



Computers as Translators

Much of what computers do is run programs that interpret a "High-level" problem specification and converts it to a "lower-level" problem that is closer the simple instructions that it understands

- High-Level Languages
  - Compilers
  - Interpreters
- Assembly Language

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

$\longrightarrow$

```
x:      .word 0
y:      .word 0
c:      .word 123456
    ...
    lw        $t0, x
    addi      $t0, $t0, -3
    lw        $t1, y
    lw        $t2, c
    add       $t1, $t1, $t2
    mul       $t0, $t0, $t1
    sw        $t0, y
```

Computers as Translators

Much of what computers do is run programs that interpret a "High-level" problem specification and converts it to a "lower-level" problem that is closer the simple instructions that it understands

- Assembly Language
- Machine Language

```
x:      .word 0
y:      .word 0
c:      .word 123456
    ...
    lw          $t0, x
    addi        $t0, $t0, -3
    lw          $t1, y
    lw          $t2, c
    add         $t1, $t1, $t2
    mul         $t0, $t0, $t1
    sw          $t0, y
```

$\longrightarrow$

```
0x04030201
0x08070605
0x00000001
0x00000002
0x00000003
0x00000004
0x706d6f43
```

Why So Many Languages?

- Application Specific
  - Pre-historically: COBOL vs. Fortran
  - Middle ages: C++ vs. Objective C
  - Recent Past: C# vs. Java
  - Today: Python vs. Matlab
- Code Maintainability
  - High-level specifications are easier to understand and modify
- Code Reuse
- Code Portability
- Virtual Machines

Next Time

- A complete Instruction Set
- Assembly Language
- Machine Language