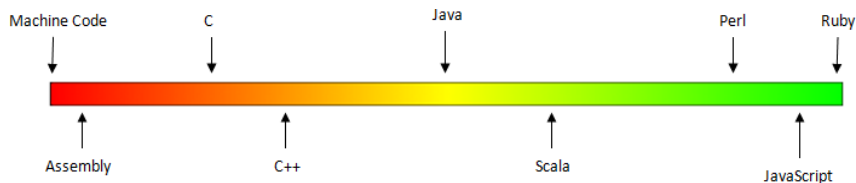


# CSci 127: Introduction to Computer Science



[hunter.cuny.edu/csci](http://hunter.cuny.edu/csci)

# Low-Level vs. High-Level Languages



- Those that directly access machine instructions & memory and have little abstraction are **low-level languages** (e.g. machine language, assembly language).
- Those that have strong abstraction (allow programming paradigms independent of the machine details, such as complex variables, functions and looping that do not translate directly into machine code) are called **high-level languages**.
- Some languages, like C, are in between—allowing both low level access and high level data structures.

# Processing



Dies ist ein Blindtext. An ihm lässt sich vieles über die Schrift ablesen, in der er gesetzt ist. Auf den ersten Blick wird der Grauwert der Schriftfläche sichtbar. Dann kann man prüfen, wie gut die Schrift zu lesen ist und wie sie auf den Leser wirkt. Dies ist ein Blindtext. An ihm lässt sich



```
def totalWithTax(food,tip):
    total = 0
    tax = 0.0875
    total = food + food * tax
    total = total + tip
    return(total)
```

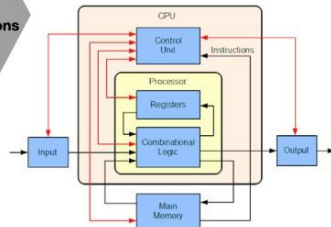
**Data  
&  
Instructions**



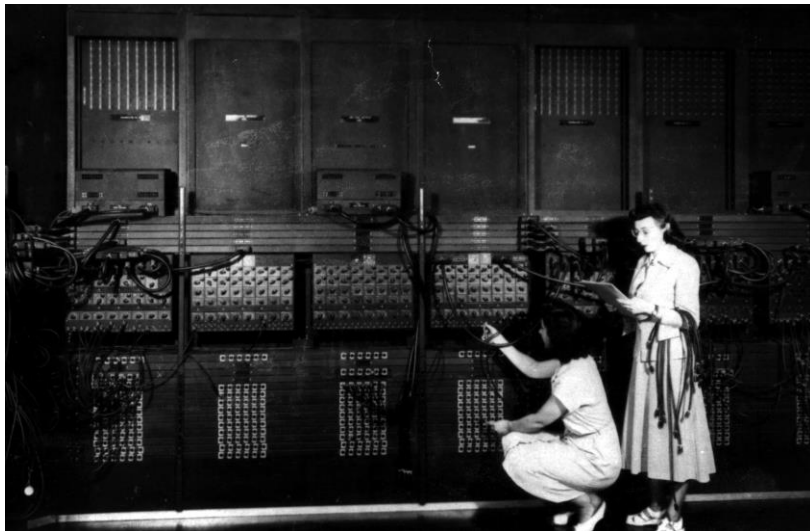
**Data  
&  
Instructions**



**Circuits (switches)**  
**On/Off 1/0 Logic**  
**Billions of switches/bits**



# Machine Language



(Ruth Gordon & Ester Gerston programming the ENIAC, UPenn)

# Machine Language



(wiki)

- Explore MIPS which is a programming language that is very “low level” in that it maps very closely to the actual commands that are used by the computer’s processor
- It is based on a reduced instruction set computer (RISC) design, originally developed by the MIPS Computer Systems.
- Due to its small set of commands, processors can be designed to run those commands very efficiently.

# Algorithmic Machine

Must be able to:

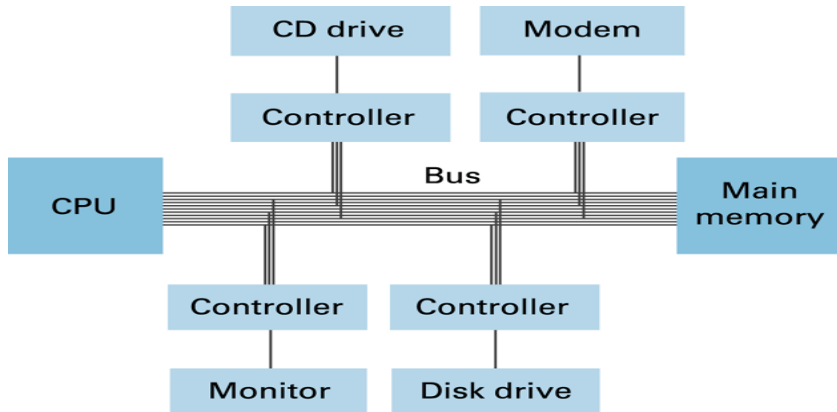
- Manipulate stored data
- Perform operations on data
- Coordinate sequence of operations

Central Processing Unit (CPU) does it (in general)

# Basic Computer Components Overview

- Central Processing Unit (CPU )
- Memory (RAM – Random Access Memory)
- CPU and RAM are installed on the main circuit board (motherboard)
- CPU and RAM communicate on bus
- Disk drive, motherboard, power supply
- There are also devices such as graphics adaptor card, network interface card (NIC), sound card, etc.

# Computer Components Diagram





# Registers

Similar to memory cells, but with a lot faster access time

- Located as part of the CPU
- There is no bus transfer to access the register

## Usually limited number of registers

CPU has a limited capacity for components, therefore cannot pack so many registers

Usually 8, 16, 32 or more (depending on the architecture)

# Registers...

Temporary holding place for data being manipulated by the CPU and ALU operations

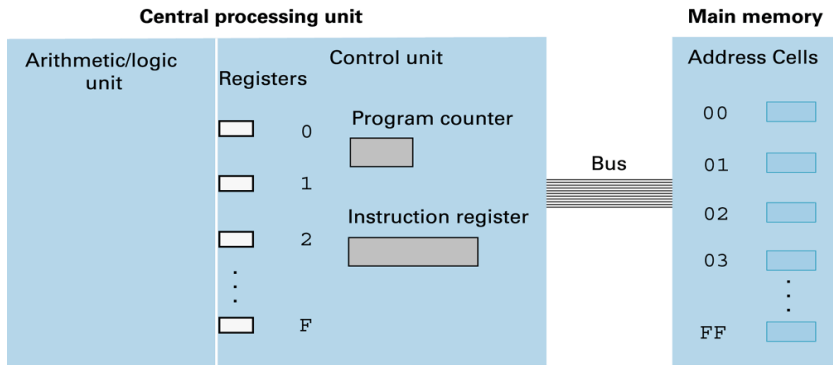
In most architectures, ALU is not allowed to manipulate data directly in memory cells

- Data in memory needs to be moved to a register first


There are two types of registers:

- Special purpose (Program Counter — PC, Instruction Register — IR)
- General purpose (e.g., R0, R1, R2, ...)

# Example: CPU/Registers and Main Memory



# Memory Usage Facts

- 
- Registers hold data immediately applicable to an operation
  - Cache holds recently used memory
  - RAM holds data needed soon
  - External storage (“mass storage”, such as hard disk) holds data and instructions not needed immediately

Price decreases, capacity increases

# Control Unit

1. Transfers data from memory to register
2. Tells ALU when and which register has data
3. Activates logic circuits in ALU
4. Tells ALU which register to put result in
5. Transfers data from register to memory

# Machine Instruction Types

Just a few instructions are sufficient for basic operations

Three main categories:

- Data transfer
- Arithmetic/Logic
- Control

Each instruction type is represented by a bit pattern

# Data Transfer Instructions

Instructions associated with the transfer of data, such as:

load *register, address* (***lw [4bytes]/lb [1byte]***)

Copy bit pattern from memory cell (specified by *address*) to *register*

store *register, address* (***sw [4bytes]/sb [1byte]***)

Copy bit pattern from *register* to memory cell (specified by *address*)

# Arithmetic/Logic Instructions

Instructions associated with the arithmetic and logical manipulation of data in CPU registers, such as:

- add
- sub
- and
- or



# Control Instructions/ Sequencing

- Instead of built-in looping structures like for and while, you create your own loops by “jumping” to the location in the program.
- Can indicate locations by writing labels at the beginning of a line.
- Then give a command to jump to that location.

See next slide..

# Control Instructions/ Sequencing

Instructions associated with program flow control, such as:

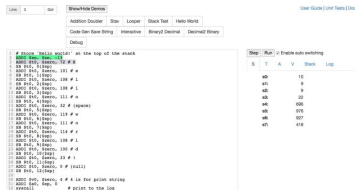
- Jump
- Branch

i.e:

- **j DONE** ;jump to the address with label DONE (unconditional)
- **beq \$R0, \$R1, DoAgain** ; DoAgain will jump to the address with label DoAgain if the registers \$R0 and \$R1 contain the same value
- **bne R1, R2, OUTSIDEL** ; OUTSIDEL is a user defined label

# MIPS

## Commands



The screenshot shows the ShowFPU Demo application. The main window displays MIPS assembly code for a program that prints "Hello World". The code uses registers \$s0 through \$s7. To the right of the code window is a register table with columns for the register name (\$), its type (T), its address (A), its value (V), and a checkbox for the stack. The register table shows the current values of the registers.

\$	T	A	V	Stack	Log
\$s0			10		
\$s1			8		
\$s2			8		
\$s3			10		
\$s4			106		
\$s5			10		
\$s6			107		
\$s7			118		

- **Registers:** locations for storing information that can be quickly accessed. Names start with '\$': \$s0, \$s1, \$t0, \$t1,...
- **R Instructions:** Commands that use data in the registers:  
add \$s1, \$s2, \$s3
- **I Instructions:** instructions that also use intermediate values.  
addi \$s1, \$s2, 100
- **J Instructions:** instructions that jump to another memory location.  
j done

# WeMIPS

View

3

Out

Show/Hide Demos

Addition Doubler

Star

Loop

Stack Test

Hello World

Code Gen Save String

Interactive

Binary2 Decimal

Decimal2 Binary

Debug

1 # Store 'Hello world!' at the top of the stack

2 **ADDI \$t0, \$zero, 32 # \$t0**

3 **SW \$t0, 0(\$t0)**

4 **ADDI \$t0, \$zero, 161 # \$t0**

5 **SW \$t0, 10(\$t0)**

6 **ADDI \$t0, \$zero, 108 # \$t0**

7 **SW \$t0, 20(\$t0)**

8 **ADDI \$t0, \$zero, 108 # \$t0**

9 **SW \$t0, 30(\$t0)**

10 **ADDI \$t0, \$zero, 111 # \$t0**

11 **SW \$t0, 40(\$t0)**

12 **ADDI \$t0, \$zero, 32 # (space)**

13 **SW \$t0, 50(\$t0)**

14 **ADDI \$t0, \$zero, 119 # \$t0**

15 **SW \$t0, 60(\$t0)**

16 **ADDI \$t0, \$zero, 111 # \$t0**

17 **SW \$t0, 70(\$t0)**

18 **ADDI \$t0, \$zero, 114 # \$t0**

19 **SW \$t0, 80(\$t0)**

20 **ADDI \$t0, \$zero, 108 # \$t0**

21 **SW \$t0, 90(\$t0)**

22 **ADDI \$t0, \$zero, 108 # \$t0**

23 **SW \$t0, \$zero, 100 # \$t0**

24 **ADDI \$t0, \$zero, 33 # \$t0**

25 **SW \$t0, \$zero, 0 # (null)**

26 **SW \$t0, 10(\$t0)**

27

28 **ADDI \$v0, \$zero, 4 # \$v0 is for print string**

29 **ADDI \$a0, \$zero, 0**

30 **syscall**

31 **# print to the loop**

Step

Run

Enable auto switching

S	T	A	V	Stack	Log
				\$t0	10
				\$t1	9
				\$t2	9
				\$t3	22
				\$t4	108
				\$t5	976
				\$t6	927
				\$t7	418

(Demo with WeMIPS)

# In Pairs or Triples:

Line: 3Got

Show/Hide Demos

Addition Doubler

Stav

Looper

Stack Test

Hello World

Code Gen Save String

Interactive

Binary2 Decimal

Decimal2 Binary

Debug

```
1 # Store 'Hello world!' at the top of the stack
2 ADDI $sp, $sp, -13
3 SB $t0, 0($sp)
4 SB $t0, 1($sp)
5 ADDI $t0, $zero, 101 # e
6 SB $t0, 2($sp)
7 ADDI $t0, $zero, 108 # l
8 SB $t0, 3($sp)
9 ADDI $t0, $zero, 108 # l
10 SB $t0, 4($sp)
11 ADDI $t0, $zero, 111 # o
12 SB $t0, 5($sp)
13 ADDI $t0, $zero, 32 # (space)
14 SB $t0, 6($sp)
15 ADDI $t0, $zero, 119 # w
16 SB $t0, 7($sp)
17 ADDI $t0, $zero, 111 # o
18 SB $t0, 8($sp)
19 ADDI $t0, $zero, 114 # r
20 SB $t0, 9($sp)
21 ADDI $t0, $zero, 108 # l
22 SB $t0, 10($sp)
23 ADDI $t0, $zero, 100 # d
24 SB $t0, 11($sp)
25 ADDI $t0, $zero, 33 # !
26 SB $t0, 12($sp)
27 ADDI $t0, $zero, 0 # (null)
28 SB $t0, 13($sp)
29
30 ADDI $v0, $zero, 4 # 4 is for print string
31 ADDI $a0, $sp, 0
32 syscall # print to the log
```

StepRun☒ Enable auto switching

S	T	A	V	Stack	Log
s0:				10	
s1:				9	
s2:				9	
s3:				22	
s4:				696	
s5:				976	
s6:				927	
s7:				418	

Write a program that prints out the alphabet: a b c d ... x y z

# WeMIPS

View

3

Out

Show/Hide Demos

Addition Doubler

Star

Loop

Stack Test

Hello World

Code Gen Save String

Interactive

Binary2 Decimal

Decimal2 Binary

Debug

1 # Store 'Hello world!' at the top of the stack

2 **ADDI \$s0, \$zero, 32 # \$0**

3 **SW \$s0, 0(\$sp)**

4 **ADDI \$s0, \$zero, 101 # \$0**

5 **SW \$s0, 10(\$sp)**

6 **ADDI \$s0, \$zero, 108 # 1**

7 **SW \$s0, 20(\$sp)**

8 **ADDI \$s0, \$zero, 108 # 1**

9 **SW \$s0, 30(\$sp)**

10 **ADDI \$s0, \$zero, 111 # 4**

11 **SW \$s0, 40(\$sp)**

12 **ADDI \$s0, \$zero, 32 # (update)**

13 **SW \$s0, 50(\$sp)**

14 **ADDI \$s0, \$zero, 119 # 4**

15 **SW \$s0, 60(\$sp)**

16 **ADDI \$s0, \$zero, 111 # 0**

17 **SW \$s0, 70(\$sp)**

18 **ADDI \$s0, \$zero, 114 # 0**

19 **SW \$s0, 80(\$sp)**

20 **ADDI \$s0, \$zero, 108 # 1**

21 **SW \$s0, 90(\$sp)**

22 **ADDI \$s0, \$zero, 108 # 0**

23 **SW \$s0, 100(\$sp)**

24 **ADDI \$s0, \$zero, 33 # 1**

25 **SW \$s0, 110(\$sp)**

26 **ADDI \$s0, \$zero, 0 # (null)**

27 **SW \$s0, 120(\$sp)**

28

29 **ADDI \$v0, \$zero, 4 # 4 is for print string**

30 **ADDI \$a0, \$s0, 0**

31 **syscall**

32 **# print to the log**

Step

Run

Enable auto stepping

S	T	A	V	Stack	Log
				\$s0	10
				\$t1	9
				\$s0	9
				\$s0	22
				\$s0	108
				\$s0	976
				\$s0	927
				\$t1	418

(Demo with WeMIPS)

# Final Question Review:

- Write a function that takes a weight in kilograms and returns the weight in pounds.
- Write a function that takes a string and returns its length.
- Write a function that, given a DataFrame, returns the minimal value in the first column.
- Write a function that takes a whole number and returns the corresponding binary number as a string.

# Final Question Review:

- Write a function that takes a weight in kilograms and returns the weight in pounds.

```
def kg2lbs(kg):  
    lbs = kg * 2.2  
    return(lbs)
```



# Final Question Review:

- Write a function that takes a string and returns its length.

```
def getLen(val):  
    valLen = len(val)  
    return(valLen)
```

# Final Question Review:

- Write a function that, given a DataFrame, returns the minimal value in the “Manhattan” column.

```
def getMin(df):  
    mM = df['Manhattan'].min()  
    return mM
```

# Final Question Review:

- Write a function that takes a whole number and returns the corresponding binary number as a string.

```
def num2bin(num):  
    binStr = ""  
    while (num > 0):  
        #Divide by 2, and add the remainder to the string  
        r = num % 2  
        binStr = str(r) + binStr  
        num = num / 2  
    return(binStr)
```