

---

# COMP 102: Computers and Computing

## Lecture 7: Machine Language

---

Instructor: Kaleem Siddiqi (siddiqi@cim.mcgill.ca)

Class web page: [www.cim.mcgill.ca/~siddiqi/102.html](http://www.cim.mcgill.ca/~siddiqi/102.html)

---

---

# Quick Recap

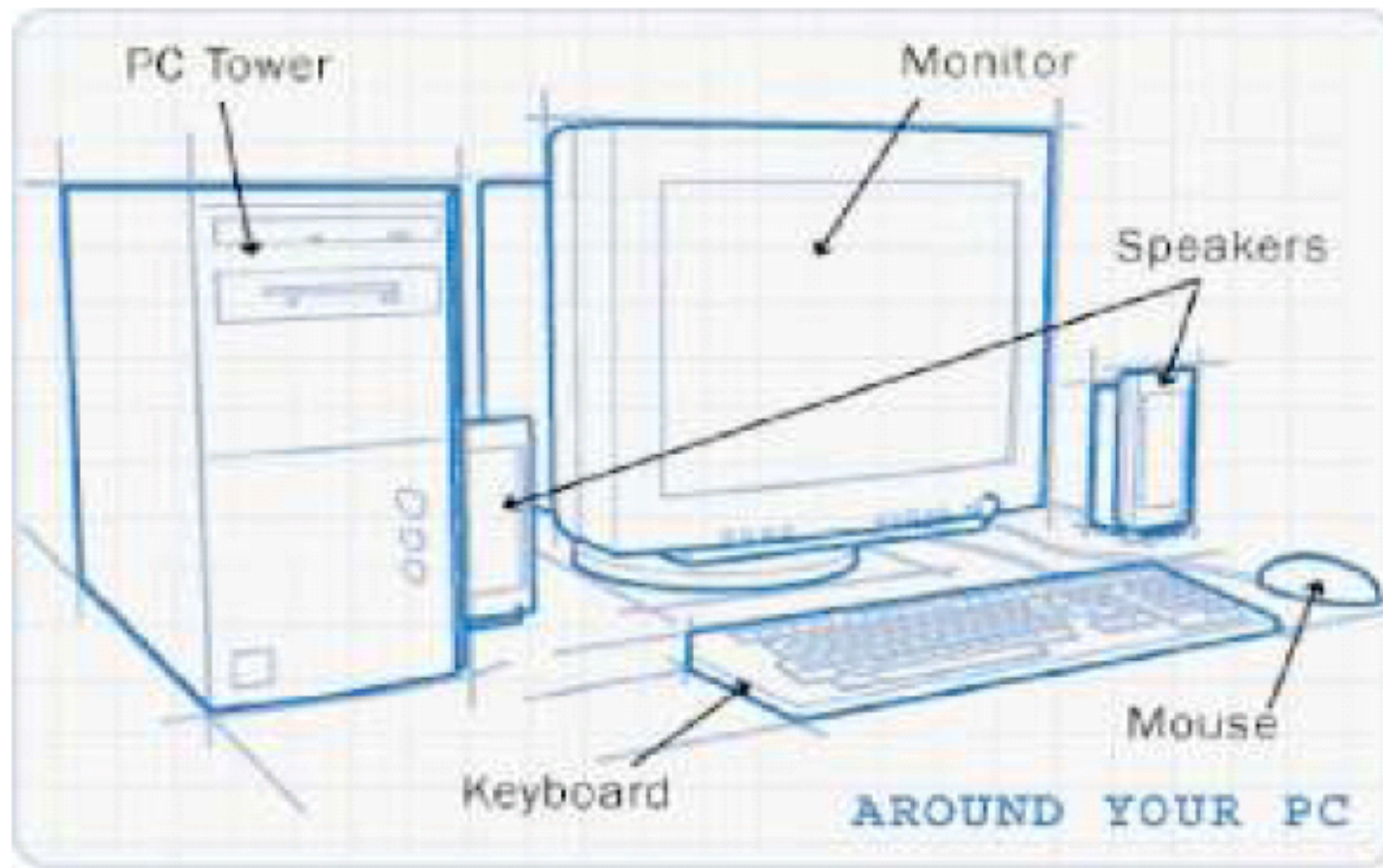
---

- Weeks 1-2: Hardware approach
  - Every problem is expressed with boolean variables and operators.
  - Can implement any function using the right combination of AND, OR, NOT.
  - Hardware solutions are quick (in terms of machine running time.)
  - But this is very inflexible (need a new circuit for each program!)
- Week 3: Software approach
  - Always same hardware, same set of circuits (any standard computer!).
  - Can implement a large variety of programs and be reprogrammed.
  - Need a layer to translate the programming language into something the computer will understand.
- Today's lecture: Machine Language  
(Much more on this in COMP 273!!)

---

# Your PC

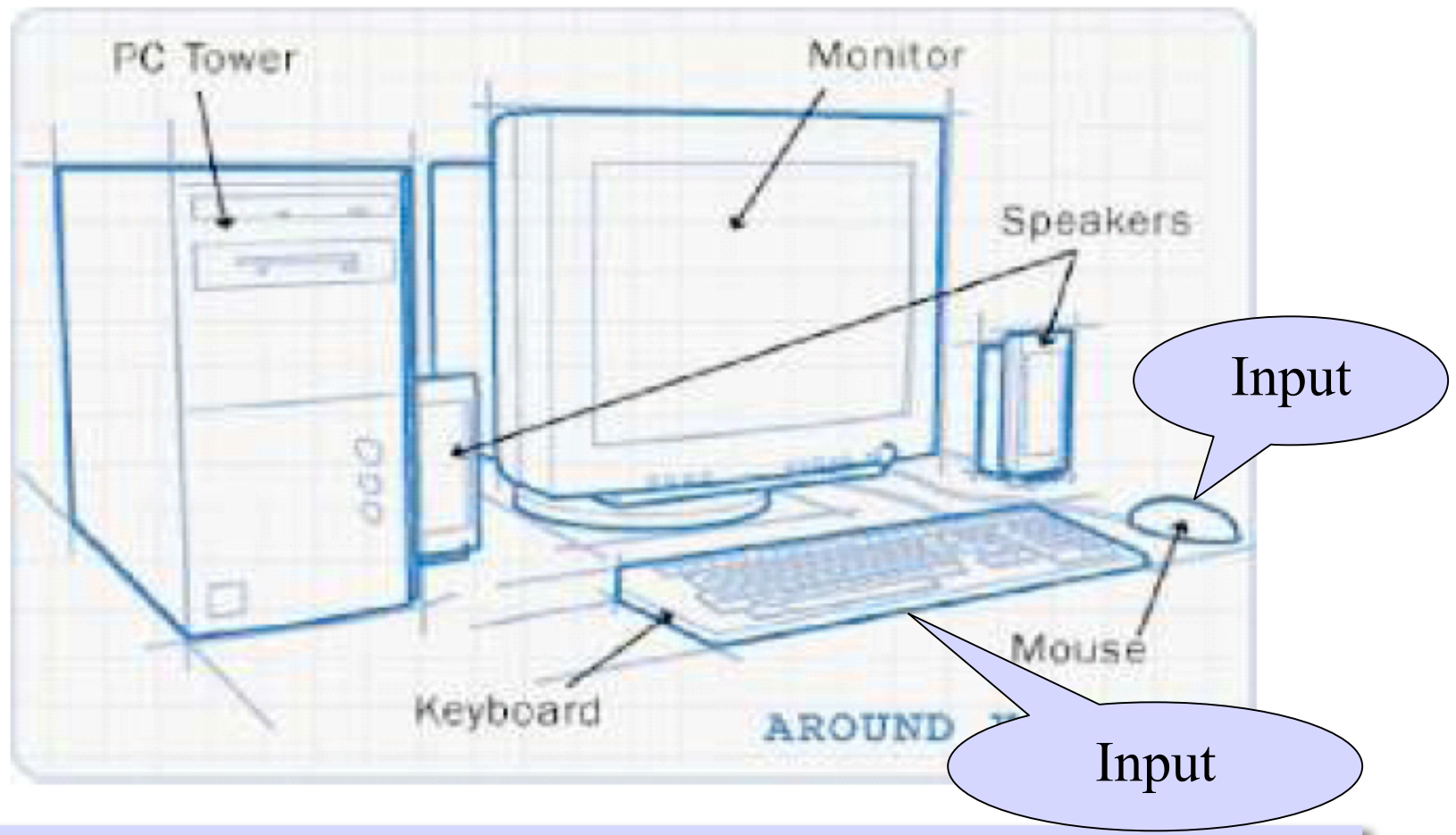
---



---

# Your PC

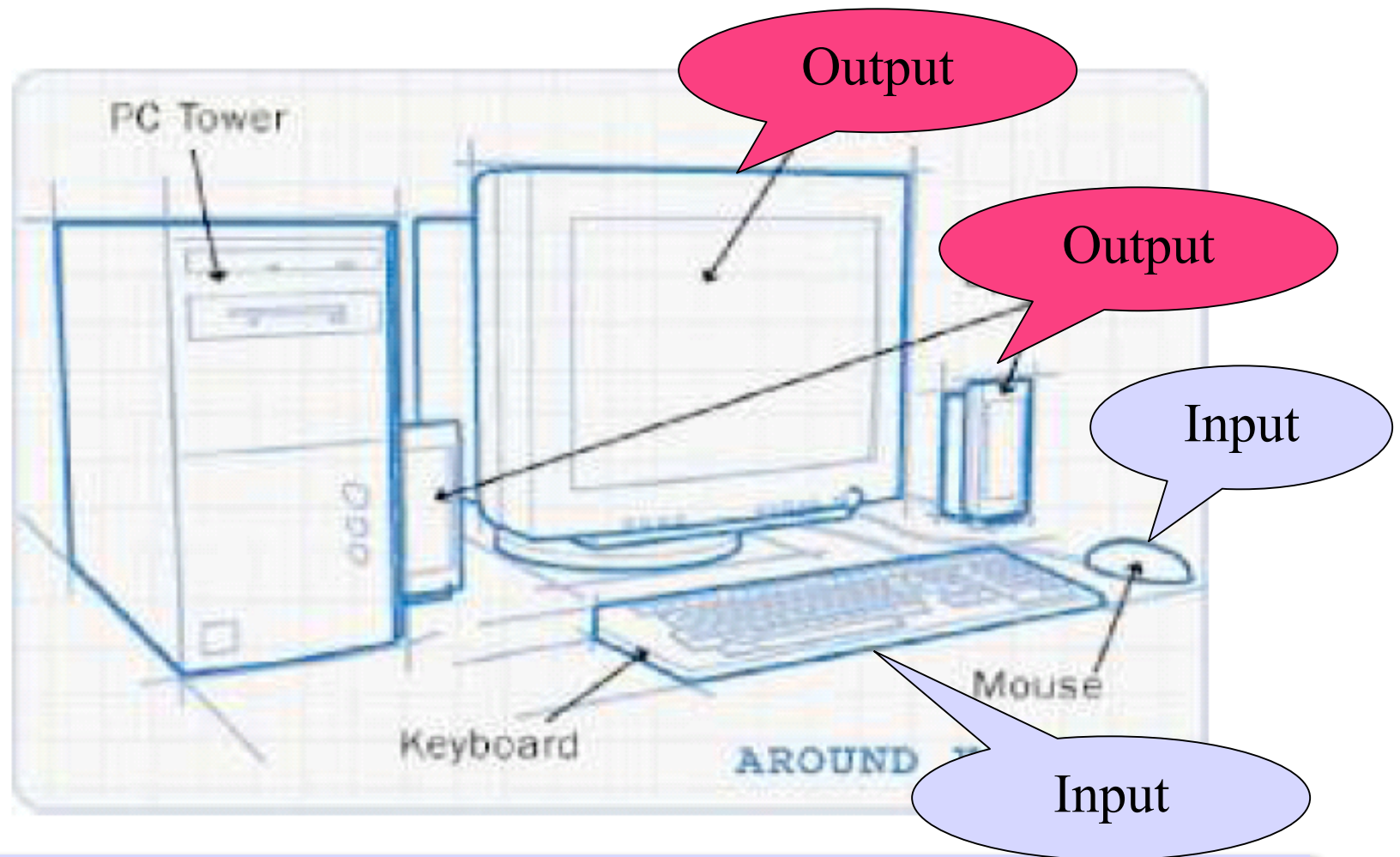
---



---

# Your PC

---

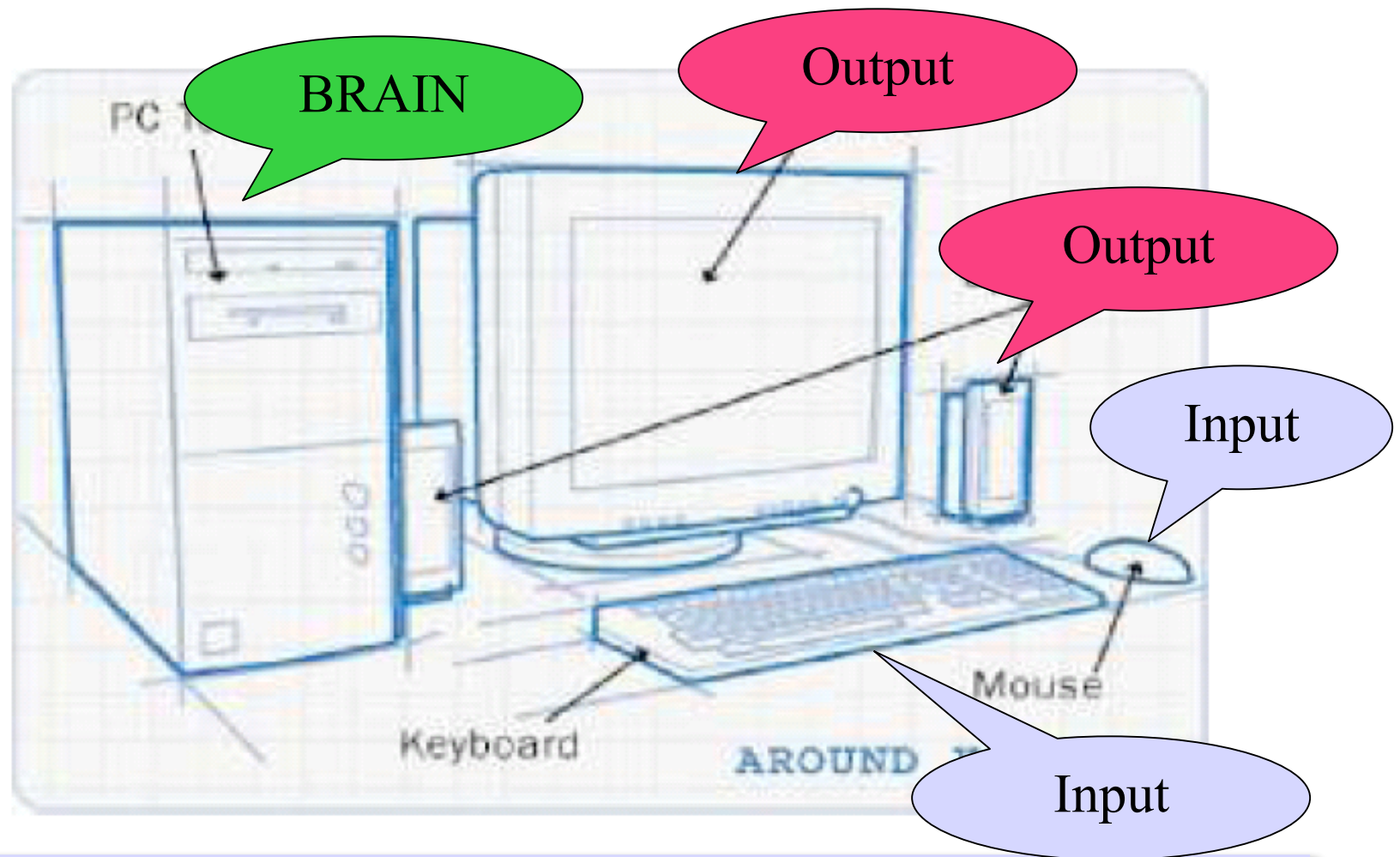




---

# Your PC

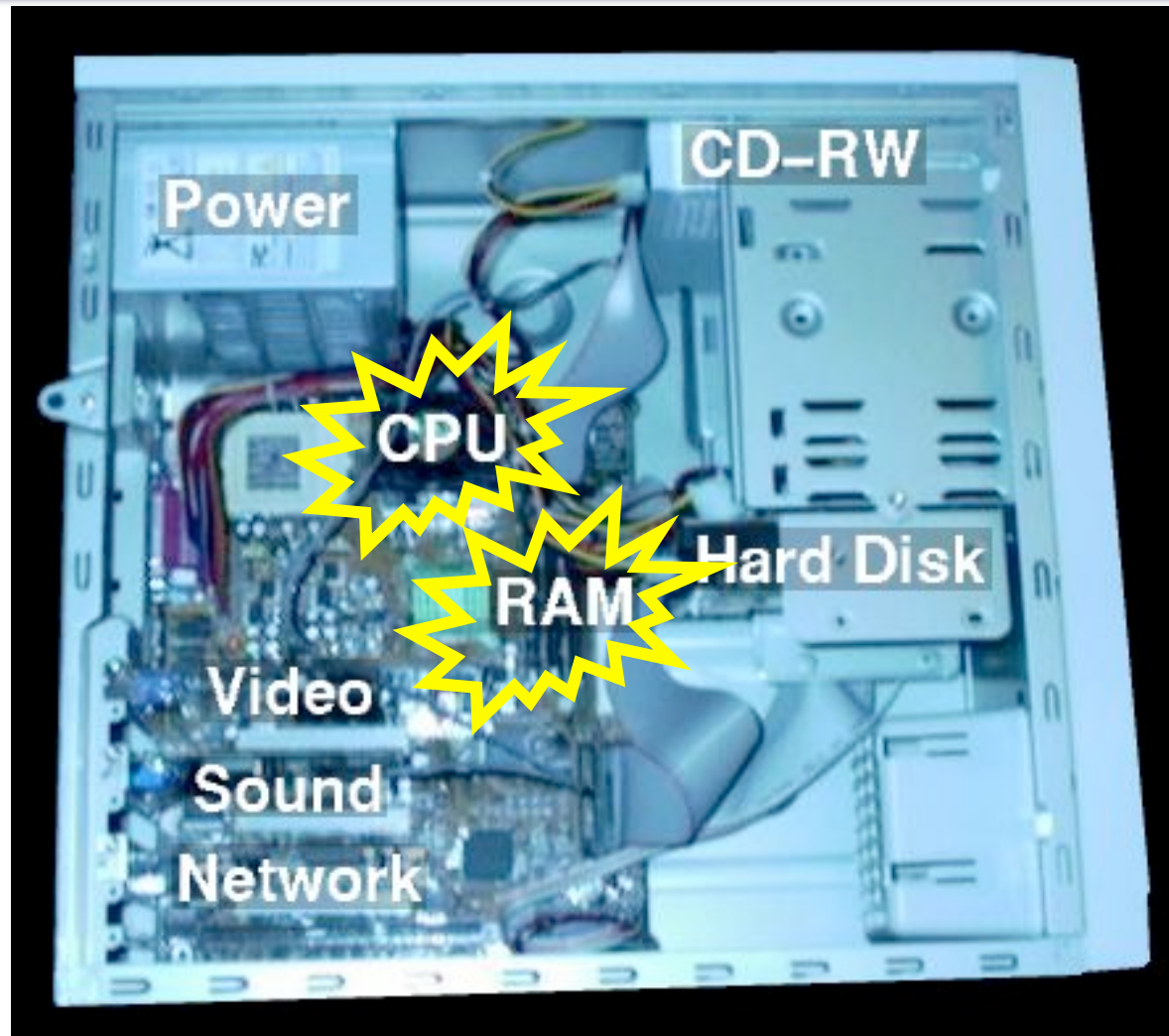
---



---

# Inside the BRAIN

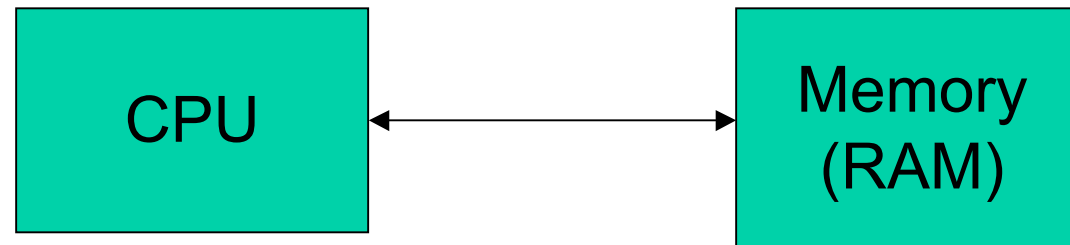
---



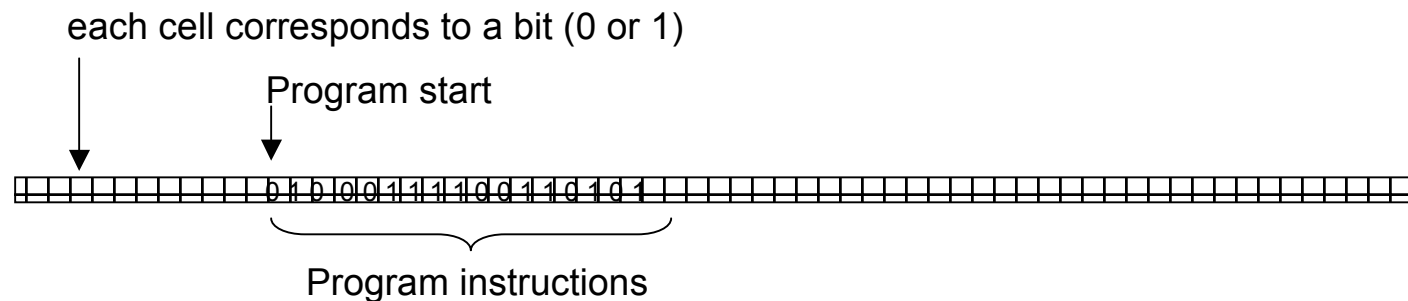
---

## Simplifying the picture

---



- CPU: Performs the operations in the current instruction.
- Memory: Stores the program (sequence of instructions and data).  
RAM = Random Access Memory





---

# The Memory Hierarchy

---

- Level 1: CPU's registers.
- Level 2: cache (L1)
- Level 3: cache (L2)
- Level 4: RAM
- Level 5: Hard disk

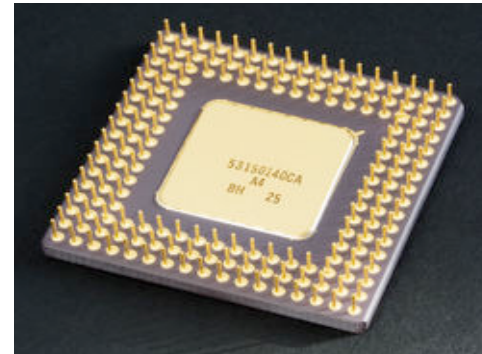
---

# CPU: Central Processing Unit

---

The CPU is composed of 3 major parts:

- ALU (Arithmetic Logic Unit)
  - Arithmetic & Logical operations
- Registers
  - Storage areas for data and machine instructions operated on by the ALU
- Control unit
  - Acts as a coordinator between the ALU and registers



---

# Instructions do very **simple** things

---

- Read bits (i.e. accessing variables).
- Change the bits in a location (i.e. assigning variables).
- Move bits from 1 cell to another.
- Treat some bits as numbers to apply arithmetic operations (add, subtract, multiply, ...)
- Modify which instruction is executed next -> CONTROL FLOW
- Communicate with external devices.

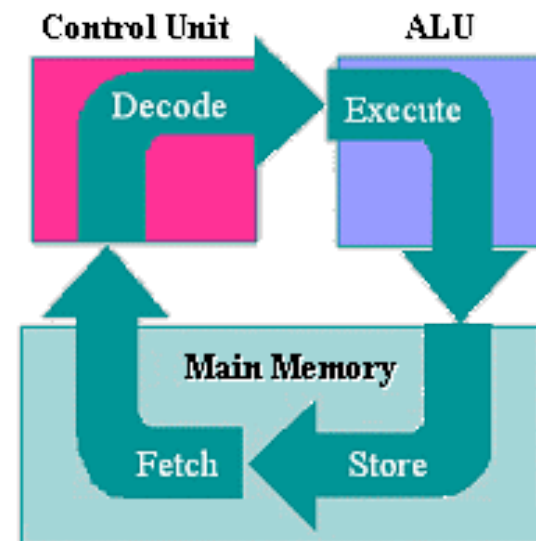
---

# Fetch-Decode-Execute Cycle

---

1. **Fetch Cycle:** The Control Unit *fetches* gets the next instruction from memory.
2. **Decode Cycle:** The Control Unit *decodes* the instruction (figuring out what the bits represent).
3. **Execute Cycle:** The ALU *executes* the required instruction and stores results into memory.

This is the only thing the CPU does!



---

# Fetch-Decode

---

1. Look at the **Program Counter** (PC) to determine the location in memory where the next instruction is stored
2. **Retrieve** this **instruction** from program memory
3. **Decode** this instruction
4. After an instruction is fetched, **increment** the PC by the length of the instruction



---

# Problem!

---

Problem #1: Need to convert the program into a long stream of bits.

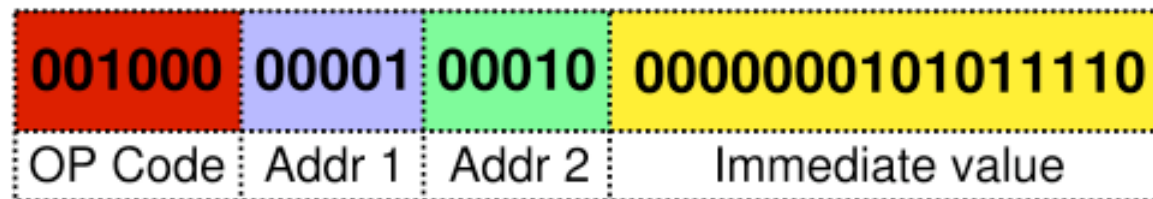
- Some parts are actual memory (e.g. variables).
  - Some parts are the instructions / operations.
- 
- Writing a program as bits directly is tedious!
  - There are human-readable mnemonics for bit patterns.

---

# Machine Instructions

---

## MIPS32 Add Immediate Instruction



Equivalent mnemonic: **addi** \$r1, \$r2, 350

Add the contents of the *register* r2 and the immediate value 350 and store the result in the *register* r1

---

# Machine Instructions

---

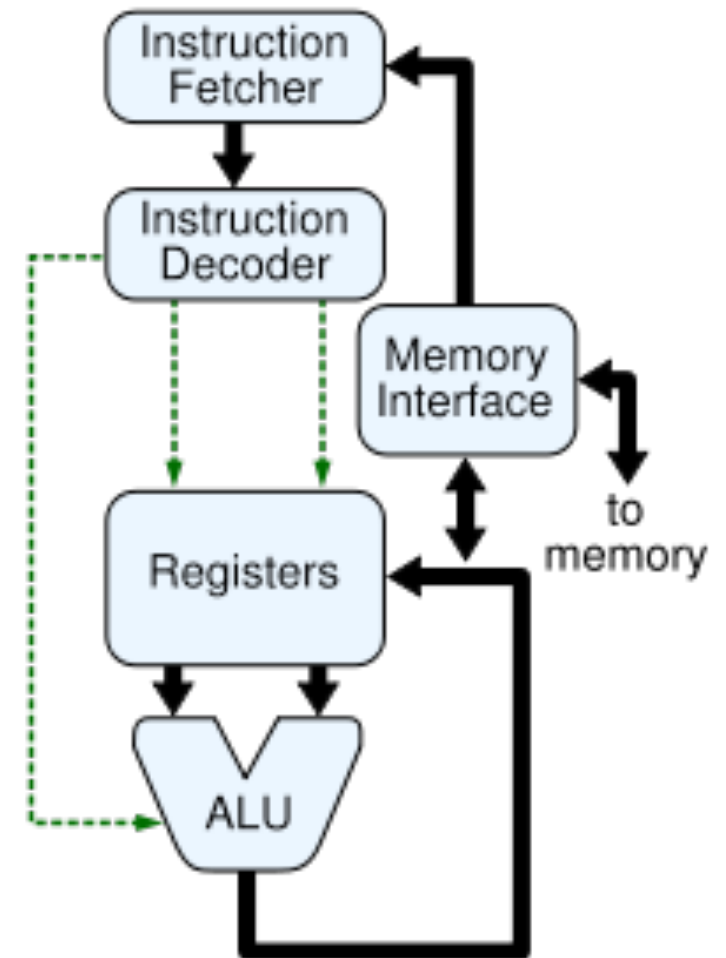
- Other kinds of instructions include:
  - Transferring data between registers or memory locations
  - Arithmetic or logical operations (use the ALU)
  - Control: test contents of a register and jump to a location
- There are binary codes for each of these (and associated mnemonics).

---

# Execute Cycle

---

1. **Execute** the instruction
  - Connects the various components of the computer so that the desired operation may be carried out
2. **Write** back the results (if any) of the execute step to some form of memory.



---

# What does this MIPS program do?

---

```
.data
string: .asciiz "ecaf das a"

.text
.globl main

main:
    la $t0, string      # $t0 and $t1 are pointers to
    la $t1, string      # to the first element of the string

loop1:
    lb $t3, 0($t1)
    addi $t1, $t1, 1
    bne $t3, $0, loop1
    addi $t1, $t1, -1

loop2:
    ble $t1, $t0, exit
    addi $t1, $t1, -1
    lb $t3, 0($t1)
    lb $t4, 0($t0)
    sb $t4, 0($t1)
    sb $t3, 0($t0)
    addi $t0, $t0, 1
    j loop2

exit:
    li $v0, 4
    la $a0, string
    syscall
```



---

## Well, let's run it using spim...

---

This example, which we shall run in class, illustrates many features including the PC, the notion of registers, the fetch-decode-execute cycle, the idea of a machine instruction and the manner in which program code is stored in memory.

You are not responsible for any of the details or for MIPS syntax, but seeing such an example in action is quite useful...

---

# Computer Speed

---

- The CPU experiences high and low voltage changes, driven by the **clock** (vibrating quartz crystal).
  - The clock operates with a predetermined **frequency** (such as 500MHz).
- Each time the clock changes, the computer's processor processes a machine instruction.
- A more accurate measurement would compare the **number of instructions per second** (MIPS: million instructions per second) as some computers use the clock ticks more efficiently than others.

---

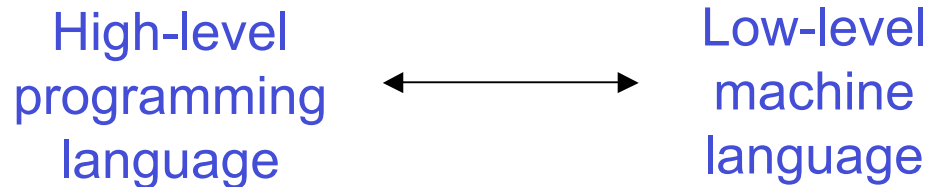
# Back to programming languages

---

- We write programs in a user-friendly programming language:

```
Integer x
Integer i
x = 0
For i = 0 to 100
    x = x + i
End loop
Print x
```

- How can we convert:

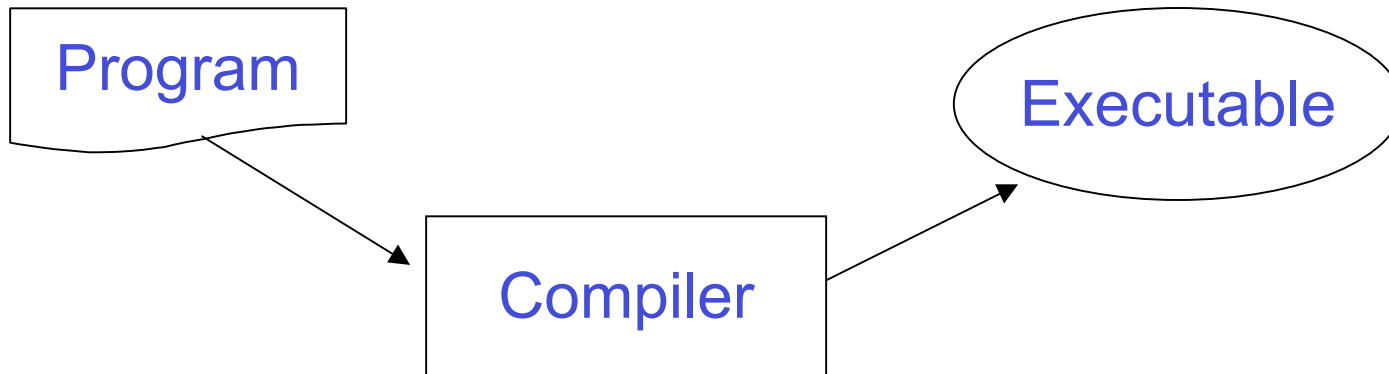


- This is a job for a **compiler**.

---

# Compiler

---



- The compiler translates high-level programming language into low-level machine language: **Program** -> **Executable**
  - Each programming language needs its own compiler.
- Note: The compiler is a program! (So need a compiler for the compiler...)
  - Here's an idea: Our compiler is actually compiled by itself!

---

# Example of an Executable

---

```
00100111101111011111111111100000
101011111011111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
1010111110100000000000000011000
1010111110100000000000000011100
1000111110101110000000000011100
1000111110111000000000000011000
0000001110011100000000000011001
0010010111001000000000000000001
0010100100000001000000001100101
1010111110101000000000000011100
000000000000000011110000010010
0000001100001111100100000100001
0001010000100000111111111110111
1010111110111001000000000011000
0011110000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
000000111100000000000000001000
0000000000000000000100000100001
```

---

**FIGURE A.1.2** MIPS machine language code for a routine to compute and print the sum of the squares of integers between 0 and 100.

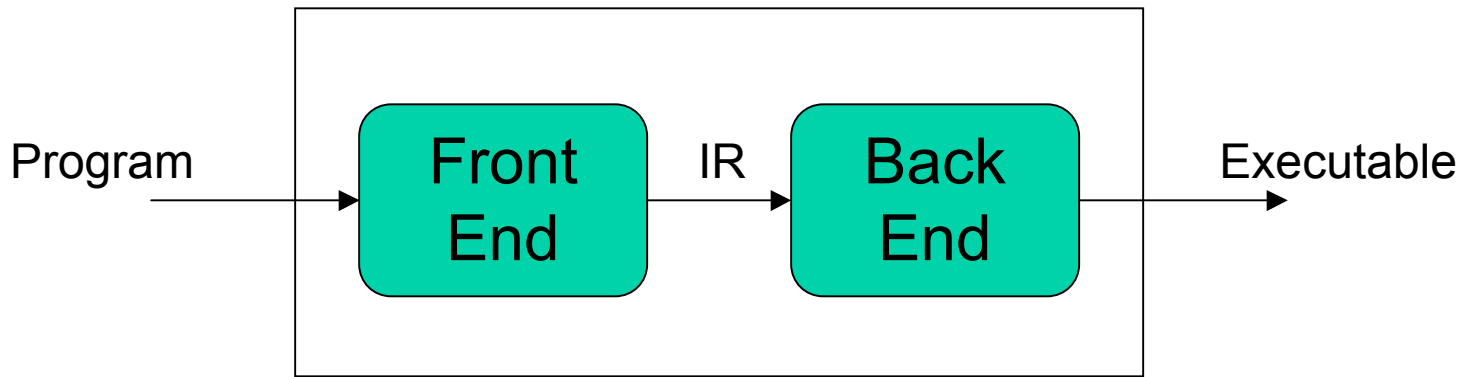
Figure reproduced from: [http://pages.cs.wisc.edu/~larus/HP\\_AppA.pdf](http://pages.cs.wisc.edu/~larus/HP_AppA.pdf)



---

# Compiler

---

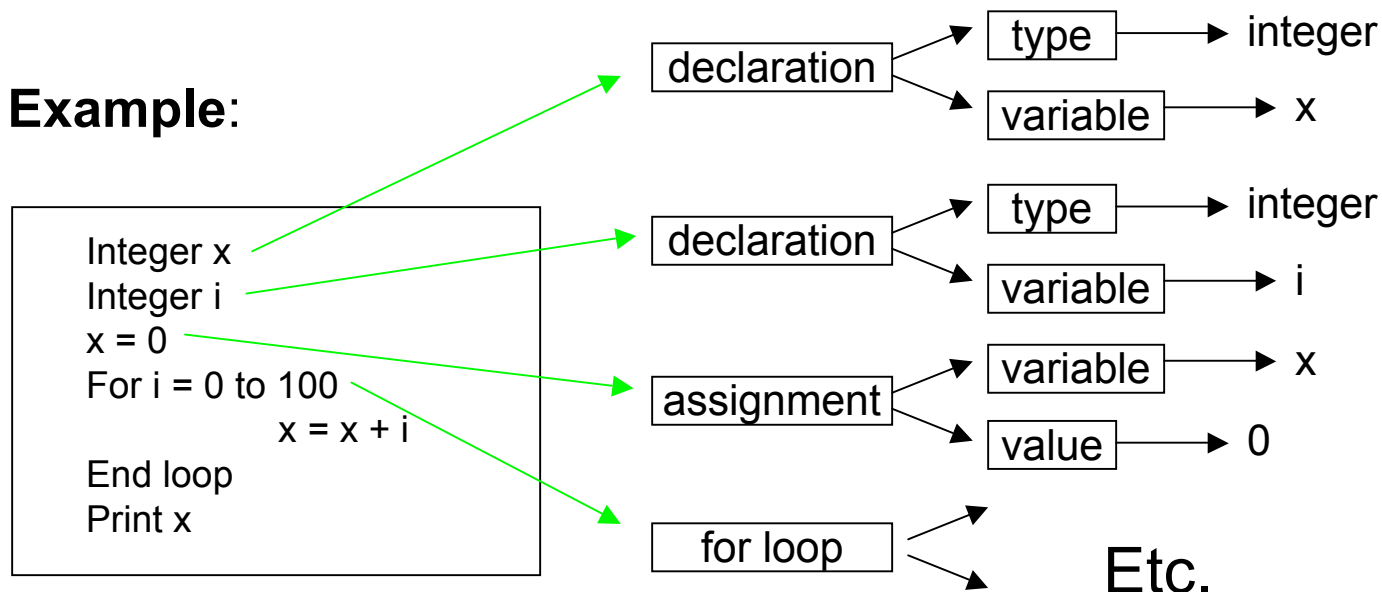


- Front End: Needs to know all about the input language.
- Back End: Knows all about the machine itself (CPU).
- Intermediate representation (IR):
  - Generated by the Front End, understood by the Back End.
  - Generic, medium-level “universal” language

# Front End Parser

- Front End will parse the input program.
  - All computer languages come with parsing rules.
  - These parsing rules are the grammar (syntax) for the language.
  - This produces a parse tree.

- **Example:**



---

# Back End

---

- Once the program is in an Intermediate Representation:
  - Step through the IR, considering each piece of the parse tree.
  - Figure out which machine instruction template matches each piece.
  - Use a look-up table to find the corresponding instruction in binary.

E.g.	$x = 0$	<code>mov, 0, [ x ]</code>
	$x = y + z$	<code>mov, r1, [ y ]</code> <code>mov, r2, [ z ]</code> <code>add, r3, r1, r2</code> <code>store, r3, [ x ]</code>

- Often we need to optimize the code (to *make it faster*):
  - Lots of clever techniques to improve the code.

Now we have a program as a sequence of bits, which can be executed by the CPU!

---

# Linking different programs

---

- Often different pieces of the program are built separately.
- Each piece can go through the compiler individually, to get separate executables. Need to put all this together somehow!
- Linker: Takes pieces of the program and puts them together into an executable.
  - Sometimes this is done as part of the compiler, sometimes separately.

---

# Take-home message

---

- Understand the main components of the computer.
- Be familiar with the principles of Fetch-Decode-Execute.
- Understand the role of machine language.
- Know the main components of the compiler (Front End, Back End, Intermediate Representation) and what purpose they serve.



---

# Final comments

---

- Some material from these slides was taken from:
  - <http://www.cim.mcgill.ca/~sveta/COMP102/>