## The Central Processing Unit
better known as

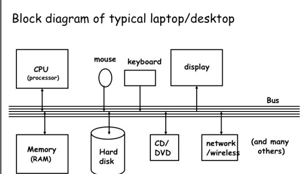
## Today: Inside the CPU

- **how does the CPU work?**
  - what operations can it perform?
  - how does it perform them? on what kind of data?
  - where are instructions and data stored?
- **some short, boring programs**
  - illustrate the basics
- **a simplified machine: the "toy machine"**
  - to try the short, boring programs
- **a program that simulates the toy machine**
  - so we can run programs written for the toy machine

## Block diagram of computer

- ***CPU can perform a small set of basic operations***
  - arithmetic: add, subtract, multiply, divide, …
  - memory access: fetch data from memory, store results back in memory
  - decision making: compare numbers, letters, …, and decide what to do next according to result
  - control the rest of the machine
- **operates by performing sequences of very simple operations *very* fast**

Block diagram of typical laptop/desktop



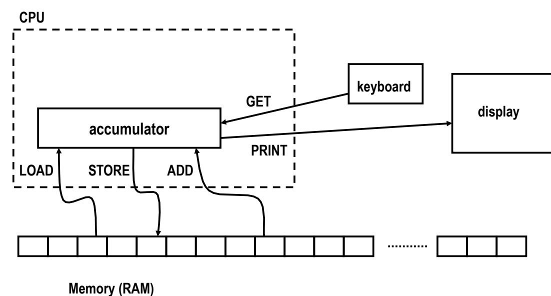

## A simple "toy" computer (a "paper" design)

- **repertoire** ("instruction set"): a handful of instructions
- **each RAM location holds one number or one instruction**
- CPU has one "**accumulator**" for arithmetic and input & output
  - a place to store one value temporarily
- **execution: *CPU operates by a simple cycle***
  - FETCH: get the next instruction from RAM
  - DECODE: figure out what it does
  - EXECUTE: do the operation
  - go back to FETCH
- **programming:** writing instructions to put into RAM and execute

## Repertoire of simple "toy" computer includes

- GET a number from keyboard and put it into the accumulator
- PRINT number that's in the accumulator
  - accumulator contents don't change!
- STORE the number that's in the accumulator into a specific RAM location
  - accumulator doesn't change!
- LOAD the number from a particular RAM location into the accumulator
  - original RAM contents don't change!
- ADD the number from a particular RAM location to the accumulator value, put the result back in the accumulator
  - original RAM contents don't change!

## Toy computer block diagram (non-artist's conception)



## A program to print a number

```
GET      get a number from keyboard into accumulator
PRINT    print the number that's in the accumulator
STOP
```

- **convert these instructions into numbers**
- **put them into RAM starting at first location**
- **tell CPU to start processing instructions at first location**

- CPU fetches GET, decodes it, executes it
- CPU fetches PRINT, decodes it, executes it
- CPU fetches STOP, decodes it, executes it

## A program to add any two numbers

```
      GET          get first number from keyboard into accumulator
      STORE  NUM   save value in RAM location labeled "NUM"
      GET          get second number from keyboard into accumulator
      ADD  NUM     add value from NUM (1st number) to accumulator
      PRINT        print the result (from accumulator)
      STOP
NUM   ---          a place to save the first number
```

- **questions:**
  - how would you extend this to adding three numbers?
  - how would you extend this to adding 1000 numbers?
  - how would you extend this to adding as many numbers as there were?

## Looping and testing and branching

- we need a way to re-use instructions
- **add a new instruction to CPU's repertoire:**
  - GOTO take next instruction from a specified RAM location instead of just using next location
- this lets us repeat a sequence of instructions indefinitely

- how do we stop the repetition?
- **add another new instruction to CPU's repertoire:**
  - IFZERO if accumulator value is zero, go to specified location instead of using next location

- **these two instructions let us write programs that repeat instructions until a specified condition becomes true**
- **the CPU can change the course of a computation according to the results of previous computations**

## Add up a lot of numbers and print the sum

```
Start   GET            get a number from keyboard
        IFZERO Show    if number was zero, go to "Show"
        ADD Sum        add Sum so far to new number
        STORE Sum      store it back in Sum so far
        GOTO Start     go back to "Start" to get the next number
Show    LOAD Sum       load sum into accumulator
        PRINT          print result
        STOP
Sum     0              initial value set to 0 before program runs
                                                   (by assembler)
```

## Assembly languages and assemblers

- **assembly language: instructions specific to a particular machine**
  - X86 (PC) family; PowerPC (older Macs); ARM (cellphones); ...
  - Shorthand for instructions humans can remember: LOAD, ADD, ...

- **assembler: a program that converts a program into numbers for loading into RAM**
- **handles clerical tasks**
  - replaces instruction names (ADD) with corresponding numeric value
  - replaces labels (names for memory locations) with corresponding numeric values: location "Start" becomes 0 or whatever
  - loads initial values into specified locations

- **terminology is archaic but still used**

- **each CPU architecture has its own instruction format and one (or more) assemblers**

## A simulator for the toy computer

- simulator is a program (software)
- simulator reads a program written for the toy computer
  - toy computer would be hardware if we built it
- simulator simulates what the toy computer would do

Is program written for toy computer program or data for simulator?

## Toy machine's complete instruction repertoire:

| | |
|---|---|
| **get** | read a number from the keyboard into accumulator |
| **print** | print contents of accumulator |
| **load Val** | load accumulator with Val (which is unchanged) |
| **store Lab** | store contents of accumulator into location labeled Lab |
| **add Val** | add Val to accumulator |
| **sub Val** | subtract Val from accumulator |
| **goto Lab** | go to instruction labeled Lab |
| **ifpos Lab** | go to instruction labeled Lab if accumulator ≥ 0 (non-negative) |
| **ifzero Lab** | go to instruction labeled Lab if accumulator is zero |
| **stop** | stop execution |
| **Num** | initialize this memory location to numeric value Num (once, before simulation starts) |

if **Val** is a name like **Sum**, it refers to a memory location with that label;
if **Val** is a number like 17, that value is used literally

## Summary

- **each memory location holds an instruction or a data value** **(or part)**
- **instructions are encoded numerically**
  (so they look the same as data)
  - e.g., GET = 1, PRINT = 2, LOAD = 3, STORE = 4, ...
- **can't tell whether a specific memory location holds an instruction or a data value** (except by context)
  - everything looks like numbers
- ***CPU* operates by a simple cycle**
  - FETCH: get the next instruction from memory
  - DECODE: figure out what it does
  - EXECUTE: do the operation
    move operands between memory and accumulator, do arithmetic, etc.
  - go back to FETCH