

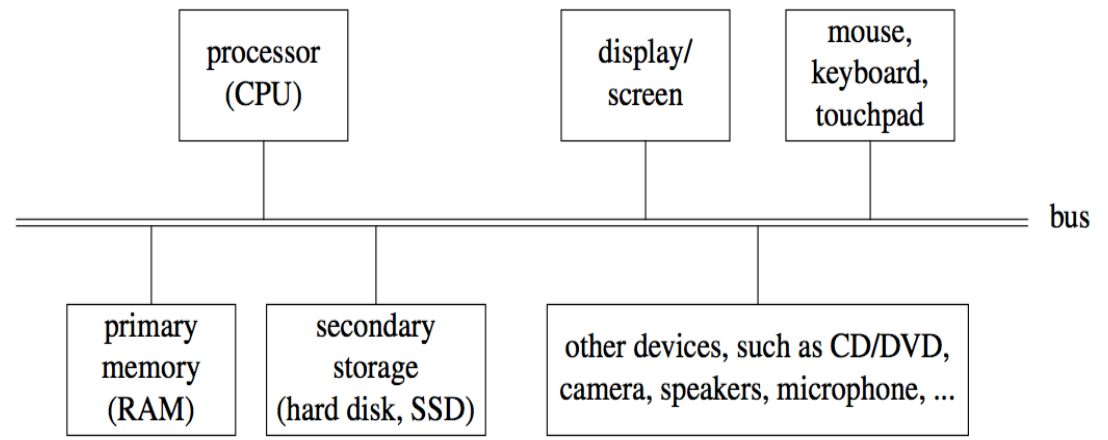
Lecture 5: Inside the processor

- **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 to illustrate the basics**
- **a toy machine to try the programs**
 - a program that simulates the toy machine
 - so we can run programs written for the toy machine
- **computer architecture: real machines**
- **caching: making things seem faster than they are**
- **how chips are made**
- **Moore's Law**
- **von Neumann architecture**
- **Turing machines**



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*

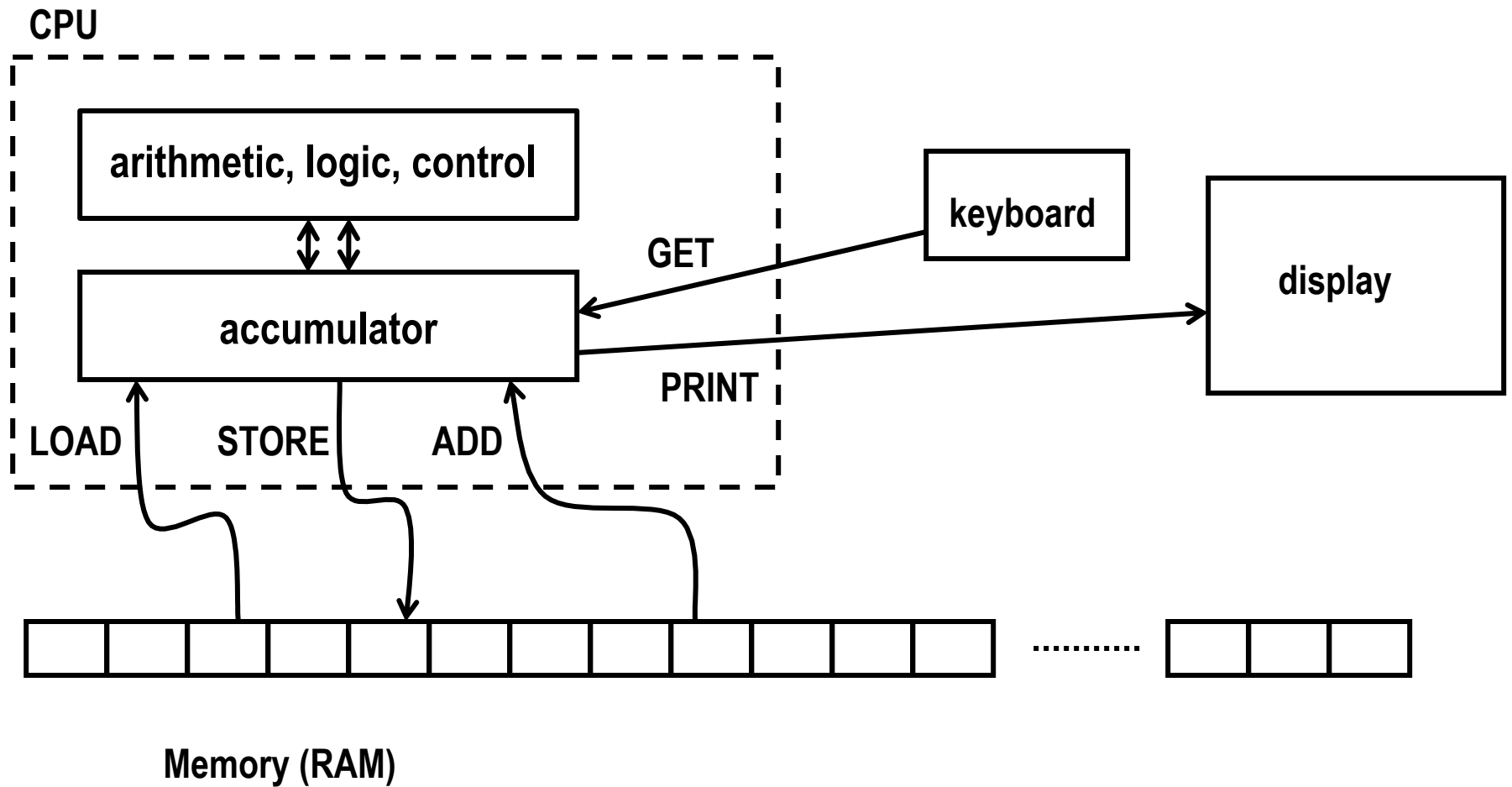


John von Neumann
1903-1957

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

- **repertoire ("instruction set"): a handful of instructions, including**
 - 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)
 - STOP** running: don't execute any more 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**

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

Assembly languages and assemblers

- **assembly language: instructions specific to a particular machine**
 - X86 (PC) family; ARM (cellphones); Toys (COS 109, COS 126), ...
- **assembler: a program that converts a program written in assembly language into numbers for loading into RAM**
- **handles clerical tasks**
 - replaces instruction names (e.g., ADD) with corresponding numeric values
 - replaces labels (names for memory locations) with corresponding numeric values: location "Start" becomes 1, "Show" becomes 6, etc.
 - loads initial values into specified locations ("Sum" set to 0)
- **each CPU architecture has its own instruction format and one (or more) assemblers**

A simulator for the toy computer (toysim.html)

- simulator (a program) reads a program written for the toy computer
- simulates what the toy computer would do
- toy machine's instruction repertoire:

<code>get</code>	read a number from the keyboard into accumulator
<code>print</code>	print contents of accumulator
<code>load Val</code>	load accumulator with Val (which is unchanged)
<code>store Lab</code>	store contents of accumulator into location labeled Lab
<code>add Val</code>	add Val to accumulator
<code>sub Val</code>	subtract Val from accumulator
<code>goto Lab</code>	go to instruction labeled Lab
<code>ifpos Lab</code>	go to instruction labeled Lab if accumulator positive (≥ 0)
<code>ifzero Lab</code>	go to instruction labeled Lab if accumulator is zero
<code>stop</code>	stop execution

`M Num` before program runs, set this memory location to Num

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 of how CPU operates

- 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

Real processors

- multiple accumulators (called "registers")
- many more instructions, though basically the same kinds
 - **arithmetic** of various kinds and sizes (e.g., 8, 16, 32, 64-bit integers):
add, subtract, etc., usually operating on registers
 - **move data** of various kinds and sizes
load a register from value stored in memory
store register value into memory
 - **comparison, branching**: select next instruction based on results of computation
changes the normal sequential flow of instructions
normally CPU just steps through instructions in successive memory locations
 - **control** rest of computer
- typical CPU repertoire: dozens to a few hundreds of instructions
- instructions and data usually occupy multiple memory locations
 - typically 2 - 8 bytes
- modern processors have multiple "cores" that are all CPUs on the same chip