

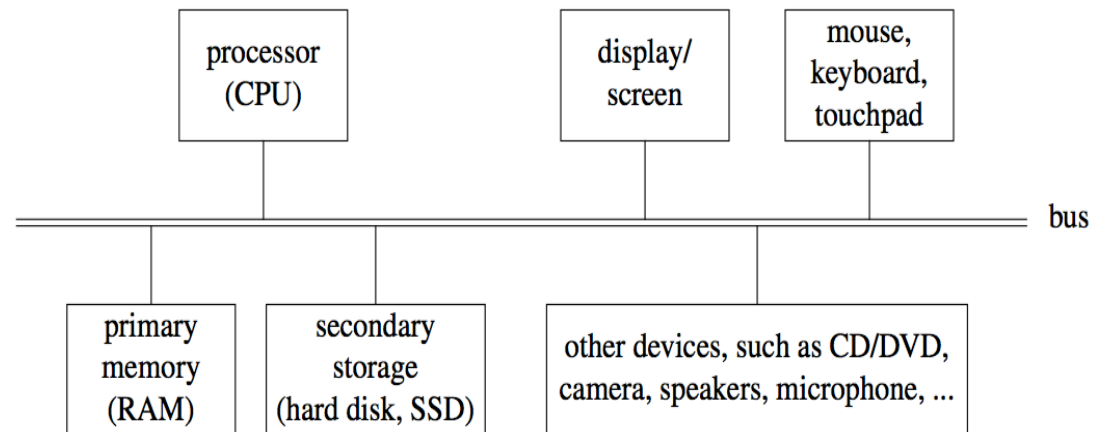
Lecture 5 (?): Inside the processor

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*

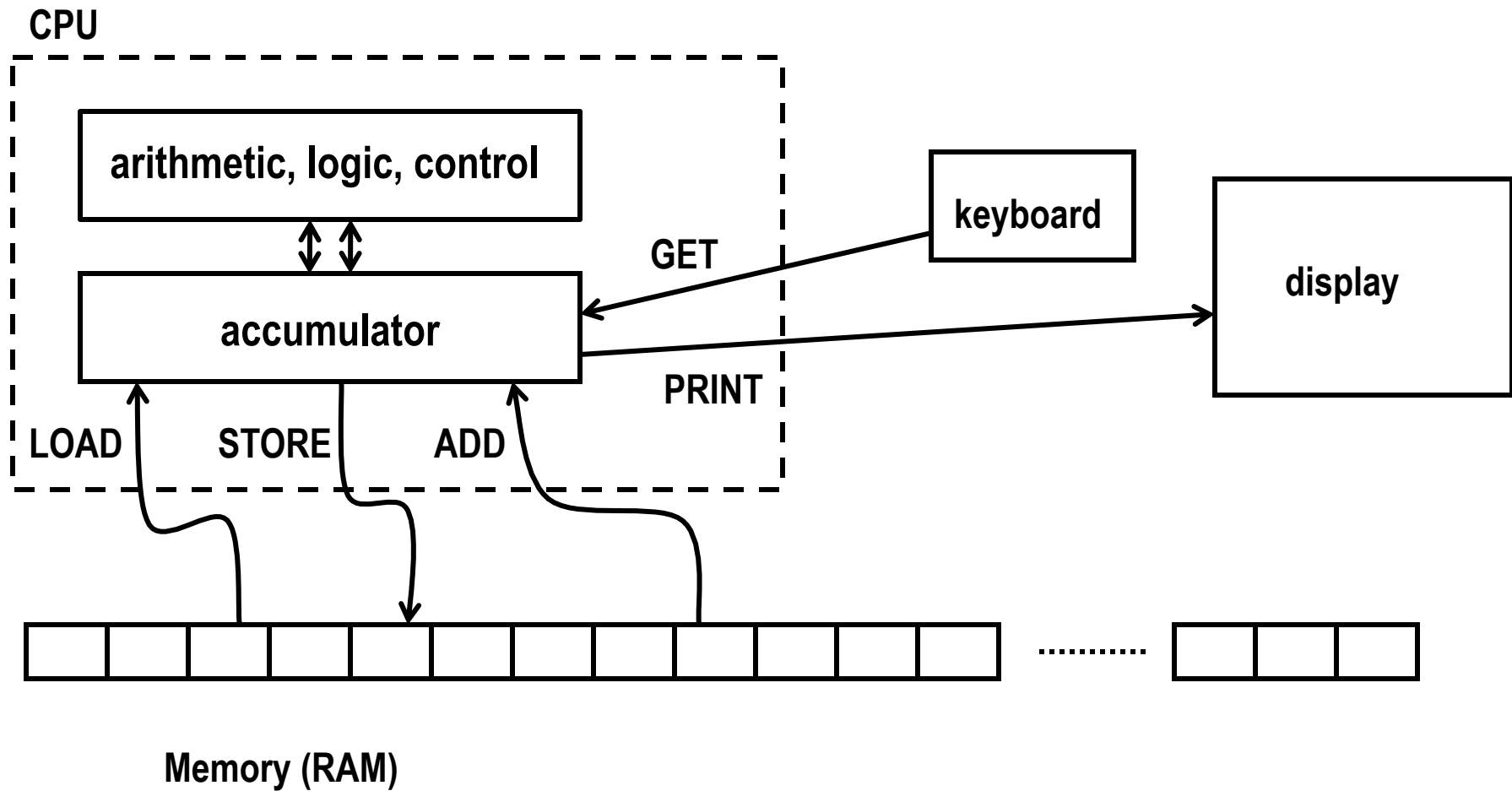


John von Neumann
1903-1957

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

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

- 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

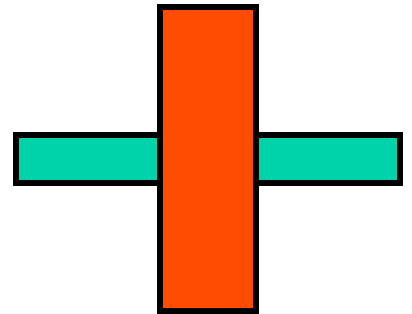
- each memory location holds an instruction or a data value (or part)
- instructions are encoded numerically (so they look the same as data)
 - e.g., GET = 1, PRINT = 2, LOAD = 3, STORE = 4, ...
- can't tell whether a specific memory location holds an instruction or a data value (except by context)
 - 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

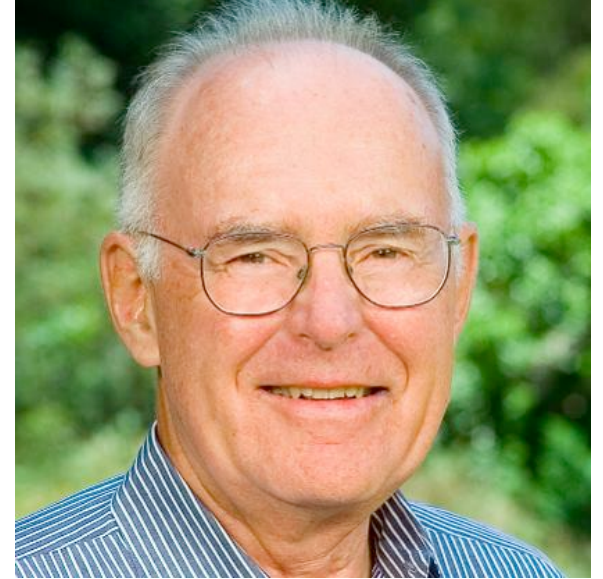
Fabrication: making chips

- grow layers of conducting and insulating materials on a thin wafer of very pure silicon
- each layer has intricate pattern of connections
 - created by complex sequence of chemical and photographic processes
- dice wafer into individual chips, put into packages
 - yield is less than 100%, especially in early stages
- how does this make a computer?
 - when conductor on one layer crosses one on lower layer, voltage on upper layer controls current on lower layer
 - this creates a transistor that acts as off-on switch that can control what happens at another transistor
- wire widths keep getting smaller: more components in given area
 - today ~ 0.01 micron = 10 nanometers
 - 1 micron == 1/1000 of a millimeter (human hair is about 100 microns)
 - eventually this will stop

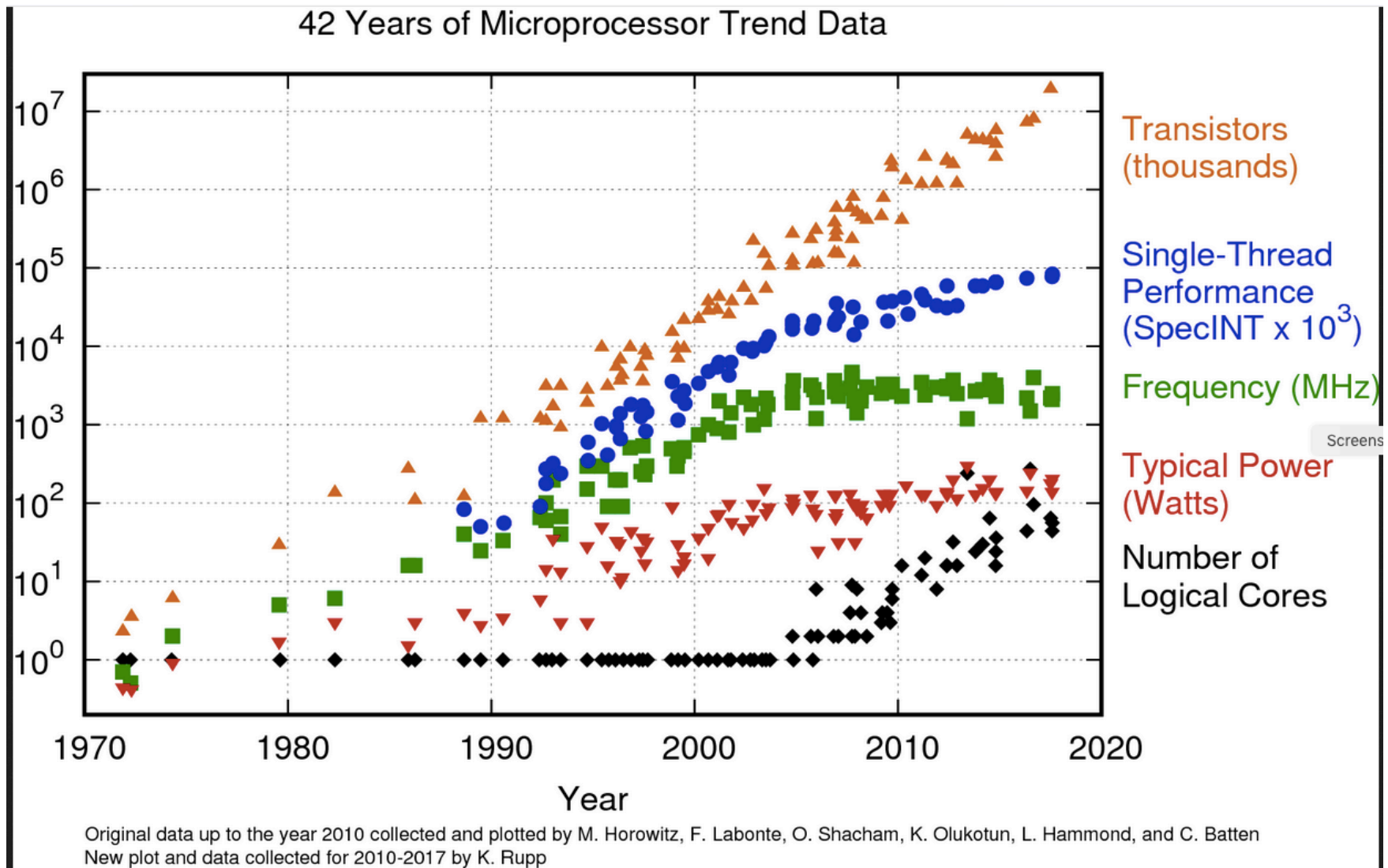


Moore's Law (1965, Gordon Moore, founder & former CEO of Intel)

- **number of transistors on a chip doubles about every 18 months**
 - and has done so since ~1961
- **consequences**
 - cheaper, faster, smaller, less power use per unit
 - ubiquitous computers and computing
- **limits to growth**
 - fabrication plants now cost \$2-4B; most are outside US
 - line widths are nearing fundamental limits
 - complexity is increasing
 - processors don't run faster
 - speed of light limitations across chip area
- **maybe some other technology will come along**
 - atomic level; quantum computing
 - optical
 - biological: DNA computing



Transistor counts and Moore's Law



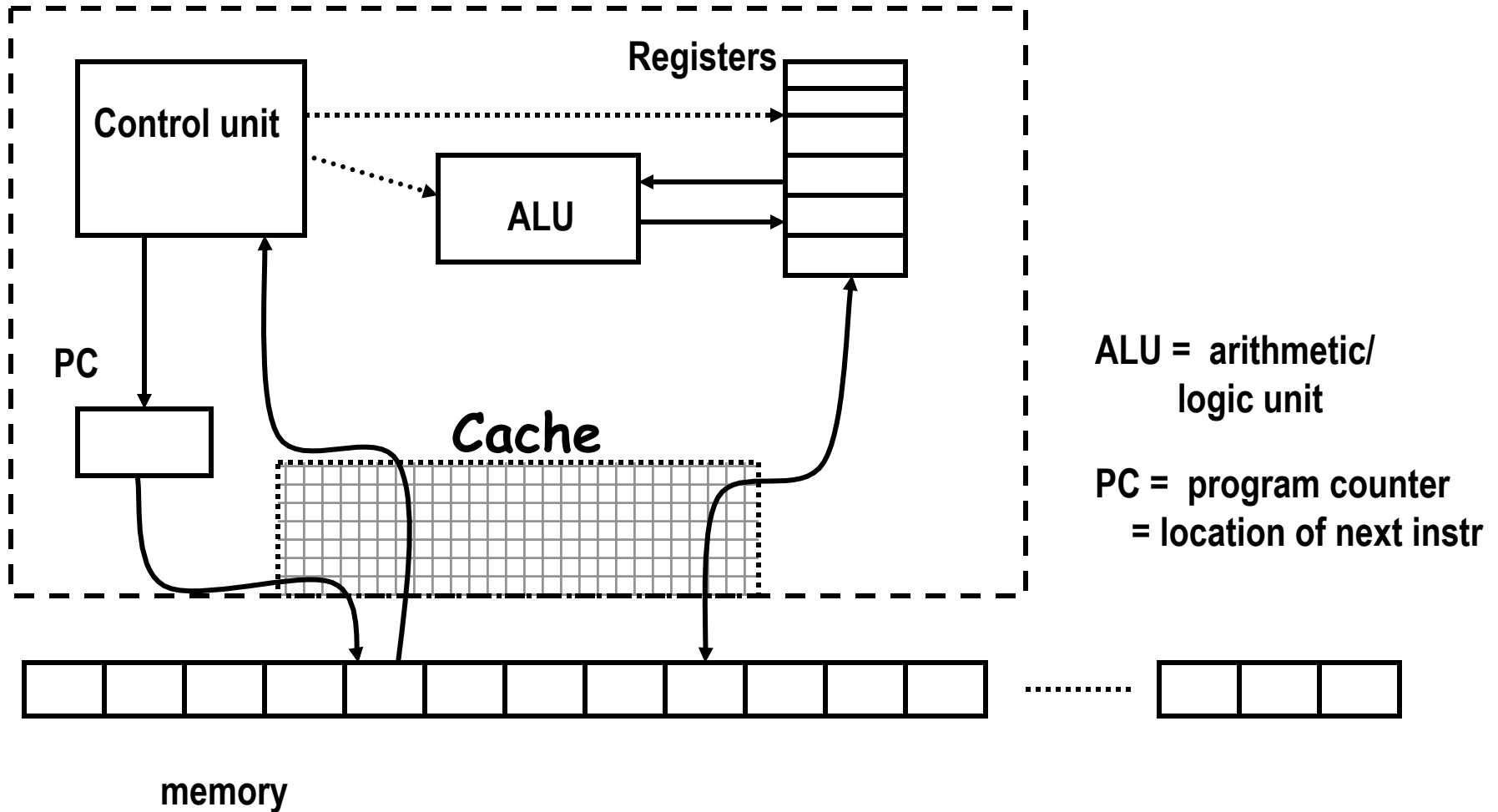
Computer architecture

- **what instructions does the CPU provide?**
 - CPU design involves complicated tradeoffs among functionality, speed, complexity, programmability, power consumption, ...
 - Intel and ARM are unrelated, totally incompatible
 - Intel: lot more instructions, many of which do complex operations
e.g., add two memory locations and store result in a third
 - ARM: fewer instructions that do simpler things, but faster
e.g., load, add, store to achieve same result
- **how is the CPU connected to the RAM and rest of machine?**
 - memory is the real bottleneck; RAM is slow (25-50 nsec to fetch)
modern computers use a hierarchy of memories (caches) so that frequently used information is accessible to CPU without going to memory
- **what tricks do designers play to make it go faster?**
 - overlap fetch, decode, and execute so several instructions are in various stages of completion (pipeline)
 - do several instructions in parallel
 - do instructions out of order to avoid waiting
 - multiple "cores" (CPUs) in one package to compute in parallel
- **speed comparisons are hard, not very meaningful**

Caching: making things seem faster than they are

- **cache: a small very fast memory for recently-used information**
 - loads a block of info around the requested info
- **CPU looks in the cache first, before looking in main memory**
 - separate caches for instructions and data
- **CPU chip usually includes multiple levels of cache**
 - faster caches are smaller
- **caching works because recently-used info is likely to be used again soon**
 - therefore more likely to be in the cache already
- **cache usually loads nearby information at the same time**
 - nearby information is more likely to be used soon
 - therefore more likely to be in the cache when needed
- **this kind of caching is invisible to users**
 - except that machine runs faster than it would without caching

CPU block diagram (non-artist's conception)



Caching is a much more general idea

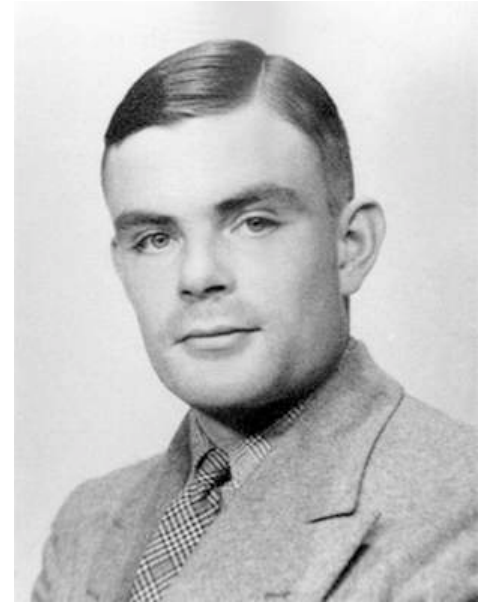
- things work more efficiently if what we need is close
- if we use something now
 - we will likely use it again soon (time locality)
 - or we will likely use something nearby soon (space locality)
- other caches in computers:
 - CPU registers
 - cache(s) in CPU
 - RAM as a cache for disk or network or ...
 - disk as a cache for network
 - network caches as a cache for faraway networks
 - caches at servers
- some are automatic (in hardware), some are controlled by software, some you have some control over

Other kinds of computers

- not all computers are Macs or PCs
- **"supercomputers"**
 - usually large number of fairly standard processors
 - extra instructions for well-structured data
- **"distributed" computing**
 - sharing computers and computation by network
 - e.g., web servers
- **embedded computers**
 - phones, games, music players, ...
 - cars, planes, weapons, ...
- **GPU (graphics processing unit)**
 - specialized processor for 3-d graphics, other streaming computations
- each represents some set of tradeoffs among cost, computing power, size, speed, reliability, ...

Turing machines

- in 1936, Turing showed that a simple model of a computer is universal
 - now called a Turing machine
- **all computers have the same computational power**
 - i.e., they can compute the same things
 - though they may vary enormously in speed, memory, etc.
- **equivalence proven / demonstrated by simulation**
 - any machine can simulate any other
 - a "universal Turing machine" can simulate any other Turing machine
- **see also**
 - Turing Test
 - Turing Award
 - Enigma



Alan Turing *38
1912-1954

Fundamental ideas

- **programmable, general-purpose computers**
 - simple instructions for arithmetic, moving data, comparison of values
 - select next instruction based on results
 - controls its own operation according to computed results
- **von Neumann architecture**
 - change what it does by putting new instructions in memory
 - instructions & data stored in same memory, indistinguishable except by context
attributed to von Neumann, 1946 (and Charles Babbage, Analytical Engine, 1830's)
 - logical structure largely unchanged for 60+ years, evolving now
 - physical structures changing very rapidly
- **Turing machines**
 - all computers have exactly the same logical power:
they can compute exactly the same things; differ only in performance
 - one computer can simulate another computer
a program can simulate a computer
- **everything is ultimately represented in bits** (binary numbers)
 - groups of bits represent larger entities: numbers of various sizes, letters in various character sets, instructions, memory addresses
 - interpretation of bits depends on context
one person's instructions are another person's data