Computer Architecture – Introduction to Computer (計算機概論)

Winston H. Hsu (徐宏民)
National Taiwan University, Taipei

November 14, 2022

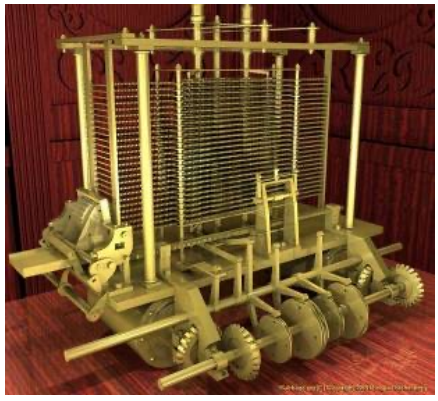
Office: R512, CSIE Building
Communication and Multimedia Lab (通訊與多媒體實驗室)
http://winstonhsu.info

The majority of the slides are from the chapter 5 in [Nisan and Schocken]

Babbage's Analytical Engine (1835)

"We may say most aptly that the Analytical Engine weaves algebraic patterns just as the Jacquard-loom weaves flowers and leaves" **提花織布機**
(Ada Lovelace)

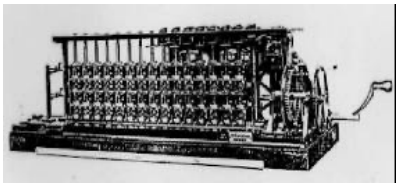
Charles Babbage (1791-1871)



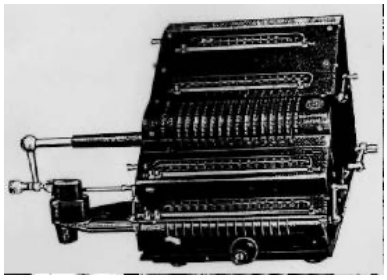
https://en.wikipedia.org/wiki/Analytical_Engine

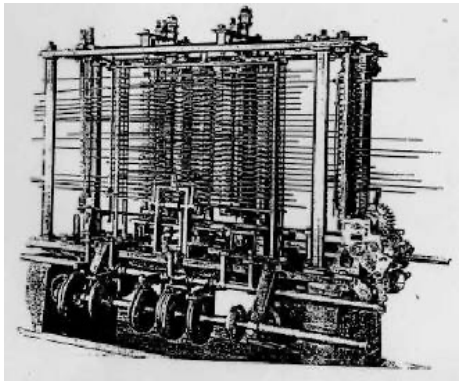
Some Early Computers and Computer Scientists

Blaise Pascal
1623-1662

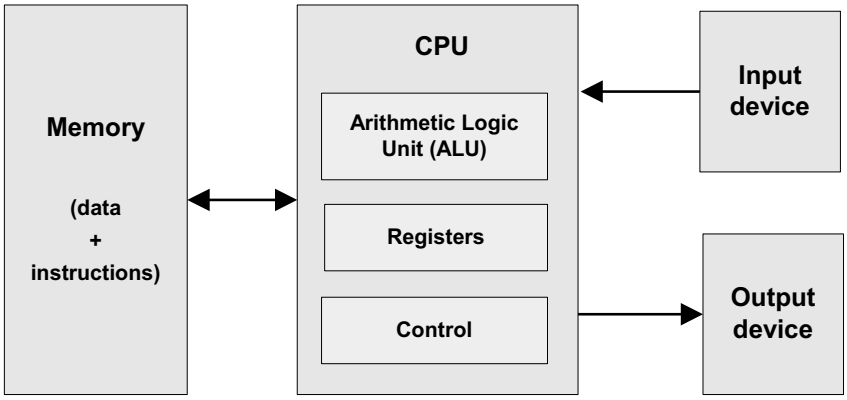


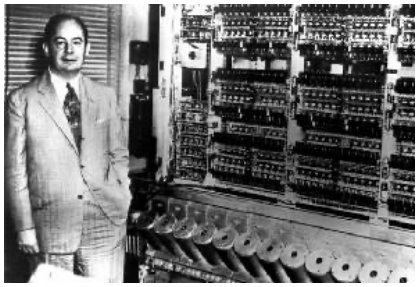
Gottfried Leibniz
1646-1716





Von Neumann Machine (around 1940)





John Von Neumann (and others) ... made it possible

Stored program concept!

Andy Grove (and others) ... made it small and fast.

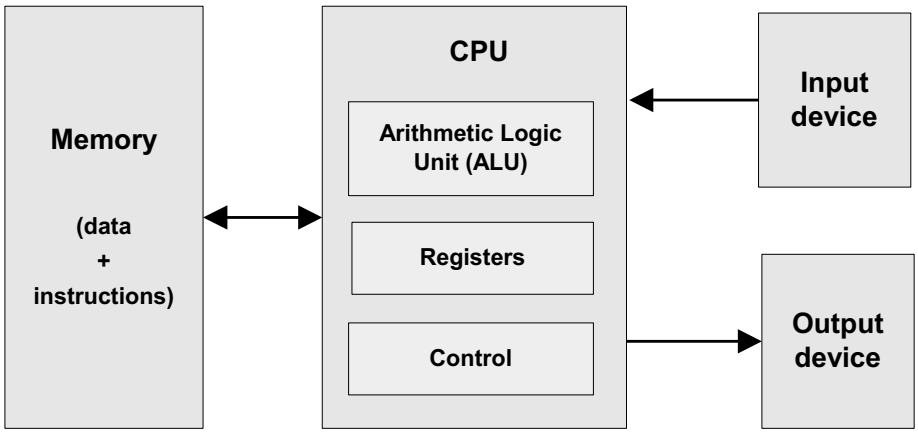
Stored Program Concept

- Formulated independently by several mathematicians in the 1930s, the stored program concept is still considered the most profound invention in, if not the very foundation of, modern computer science.
- A scientific breakthrough
- The program's code is stored and manipulated in the computer memory, just like data, becoming what is known as "software."

Turing Machine vs. von Neumann Machine

- The Turing machine – an abstract artifact describing a deceptively simple computer – is used mainly to analyze the logical foundations
- In contrast, the von Neumann machine is a practical architecture and the conceptual blueprint of almost all computer platforms today.

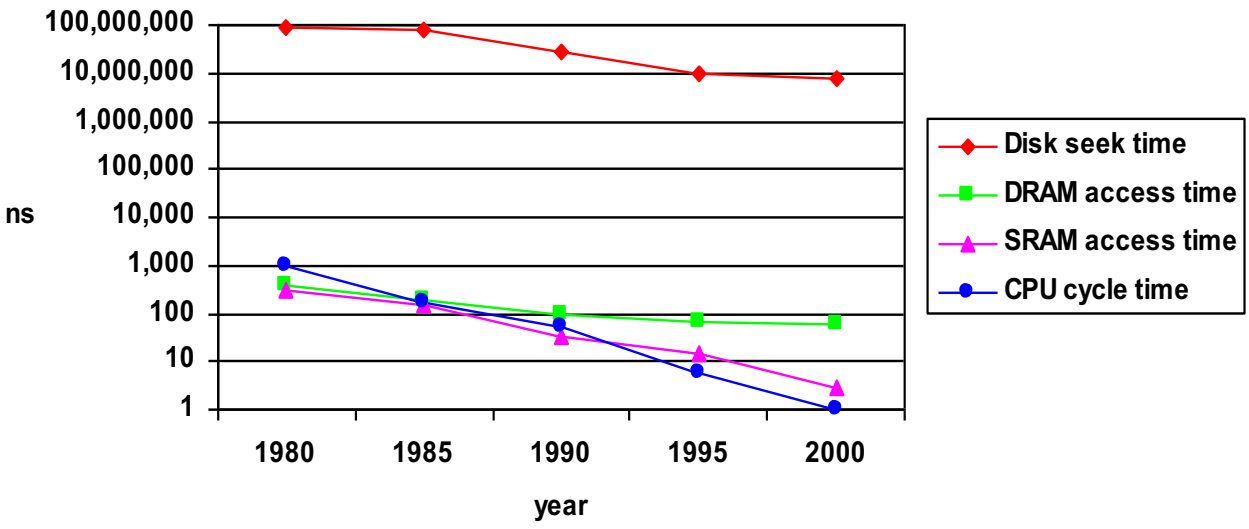
Processing Logic: Fetch-Execute Cycle



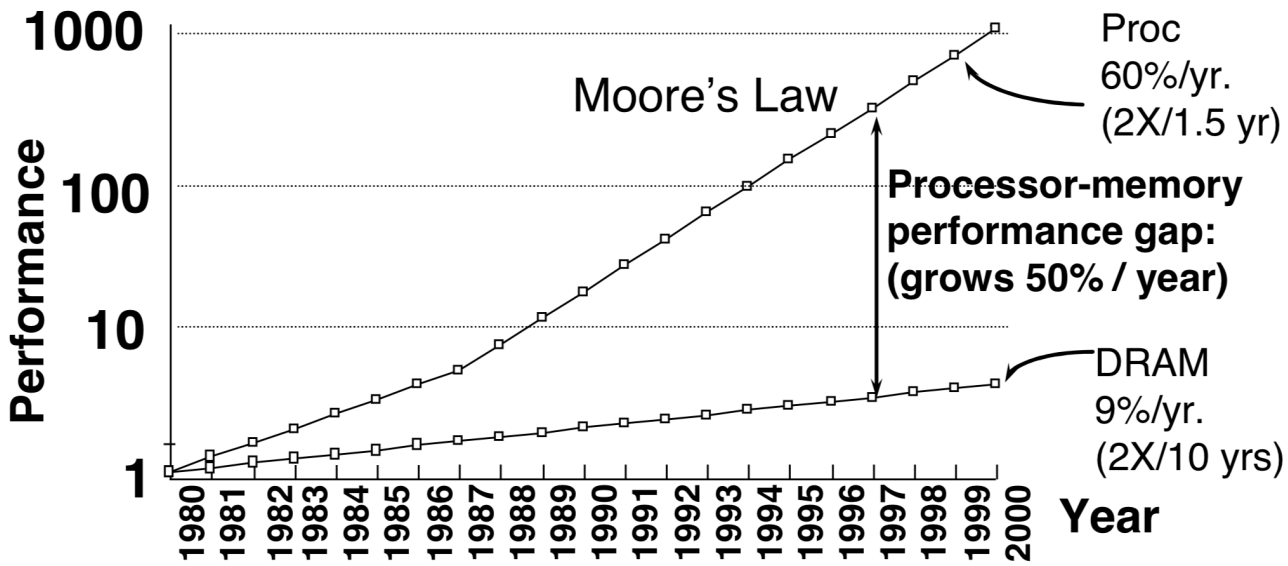
- Executing the current instruction involves one or more of the following micro-tasks:
  - Have the ALU compute some function out = f (register values)
  - Write the ALU output to selected registers
  - As a side-effect of this computation, figure out which instruction to fetch and execute next.

The CPU-Memory Gap (1/2)

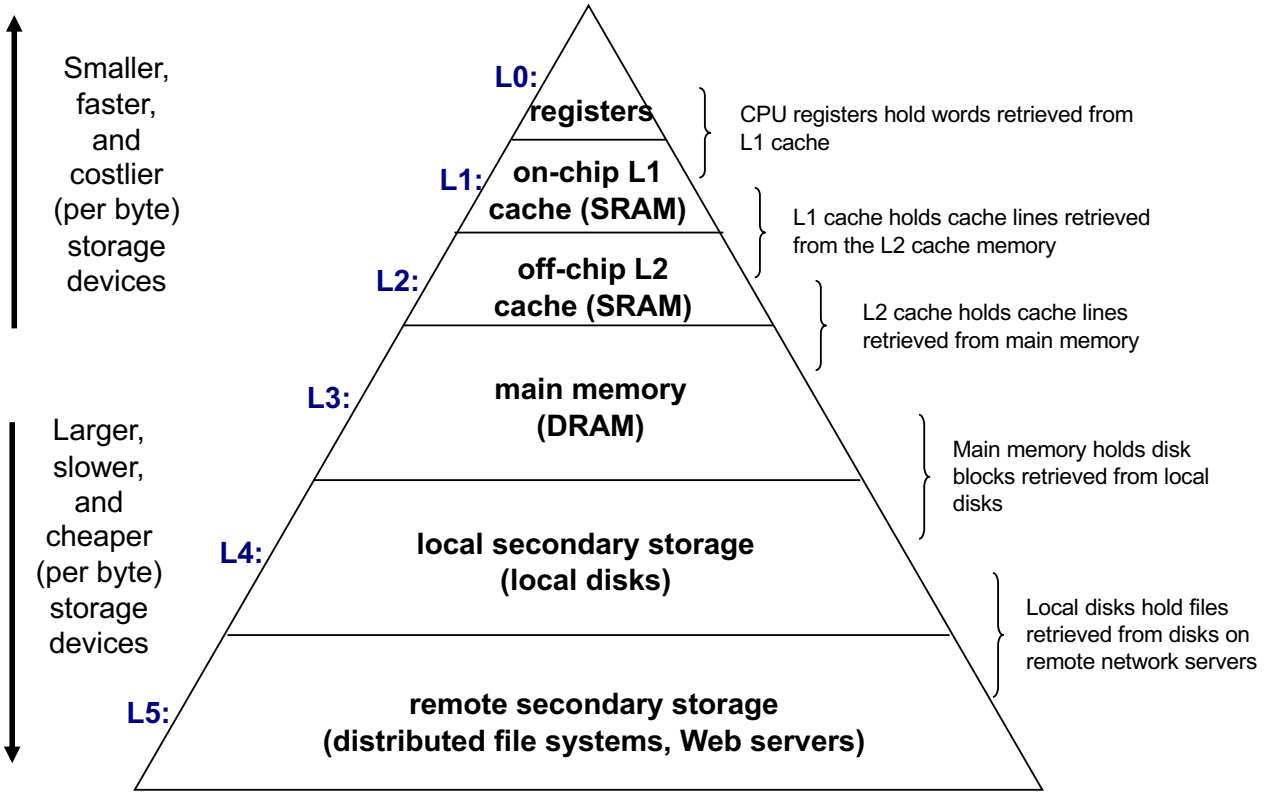
- The increasing gap between DRAM, disk, and CPU speeds.



The CPU-Memory Gap (2/2)

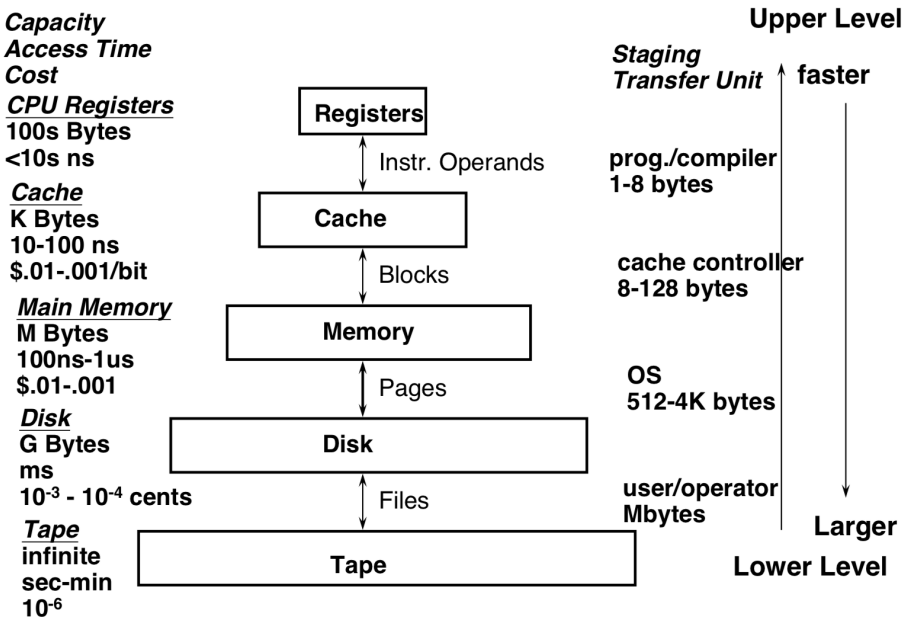


An Example Memory Hierarchy (1/2)



An Example Memory Hierarchy (2/2)

- Take the numbers as relative measures; outdated.



The "Hack" Chip-Set and Hardware Platform

Elementary logic gates

- Nand
- Not
- And
- Or
- Xor
- Mux
- Dmux
- Not16
- And16
- Or16
- Mux16
- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

Combinational chips

- HalfAdder
- FullAdder
- Add16
- Inc16
- ALU

Sequential chips

- DFF
- Bit
- Register
- RAM8
- RAM64
- RAM512
- RAM4K
- RAM16K
- PC

Computer Architecture

- Memory
- CPU
- Computer

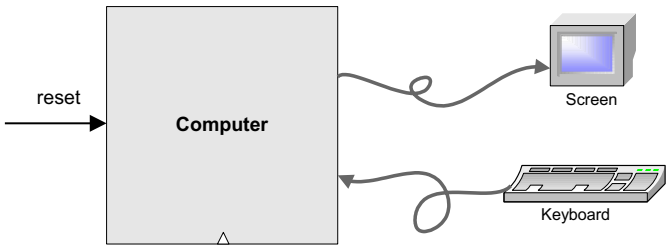
this lecture

The "Hack" Computer (Example)

- A 16-bit Von Neumann platform
- The *instruction memory* and the *data memory* are physically separate
- Screen: 256 rows by 512 columns, black and white
- Keyboard: standard
- Designed to execute programs written in the Hack machine language
- Can be easily built from the chip-set that we built so far in the course

Main parts of the Hack computer:

- Instruction memory (ROM)
- Memory (RAM):
  - Data memory
  - Screen (memory map)
  - Keyboard (memory map)
- CPU
- Computer (the logic that holds everything together).

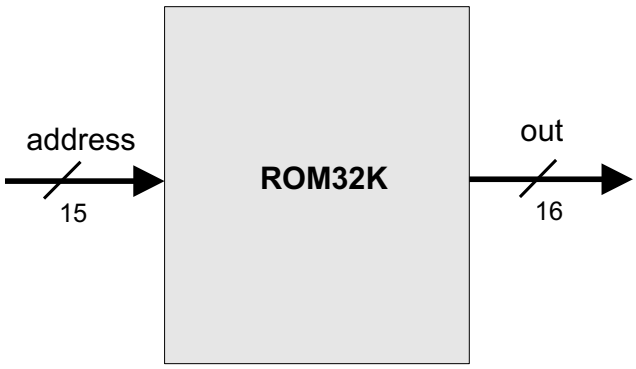


Why OS?

Lecture / construction plan

- Instruction memory
- Memory:
  - Data memory
  - Screen
  - Keyboard
- CPU
- Computer

Instruction Memory



Function:

- The `ROM` is pre-loaded with a program written in the Hack machine language
- The `ROM` chip always emits a 16-bit number:

```
out = ROM32K[address]
```

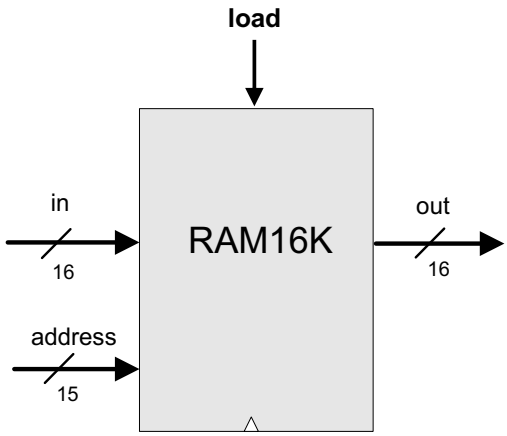
- This number is interpreted as the *current instruction*.

Data Memory

Low-level (hardware) read/write logic:

To read `RAM[k]`: set `address` to k,
probe `out`

To write `RAM[k]=x`: set `address` to k,
set `in` to x,
set `load` to 1,
run the clock



High-level (OS) read/write logic:

To read `RAM[k]`: use the OS command `out = peek(k)`

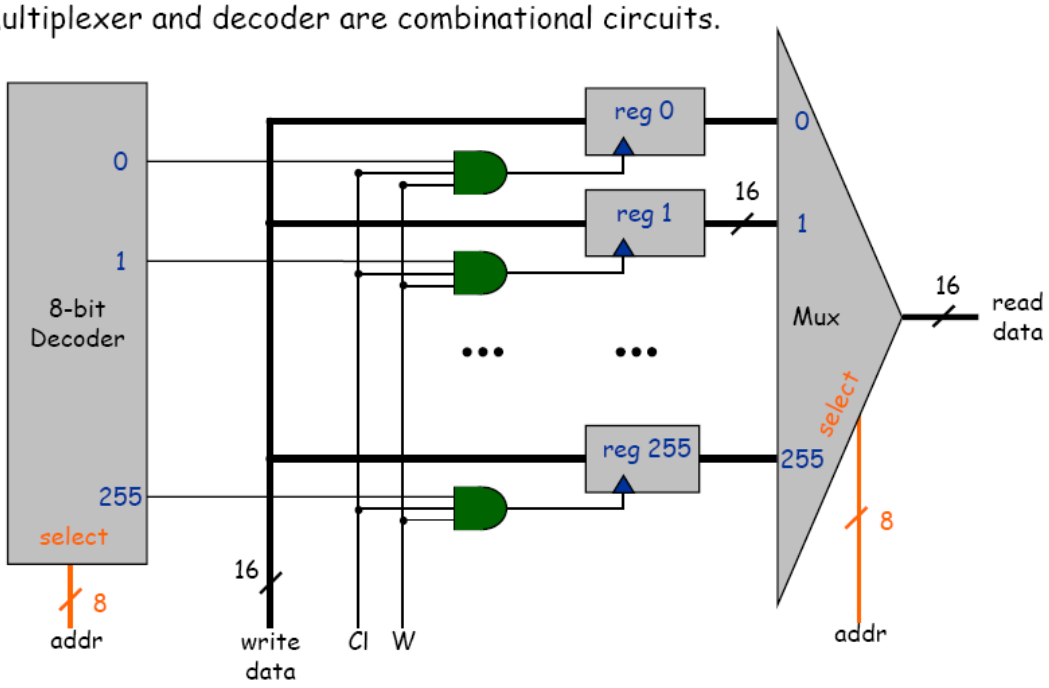
To write `RAM[k]=x`: use the OS command `poke(k,x)`

`peek` and `poke` are OS commands whose implementation should effect the same behavior as the low-level commands (in the sample computer)

Register file implementation

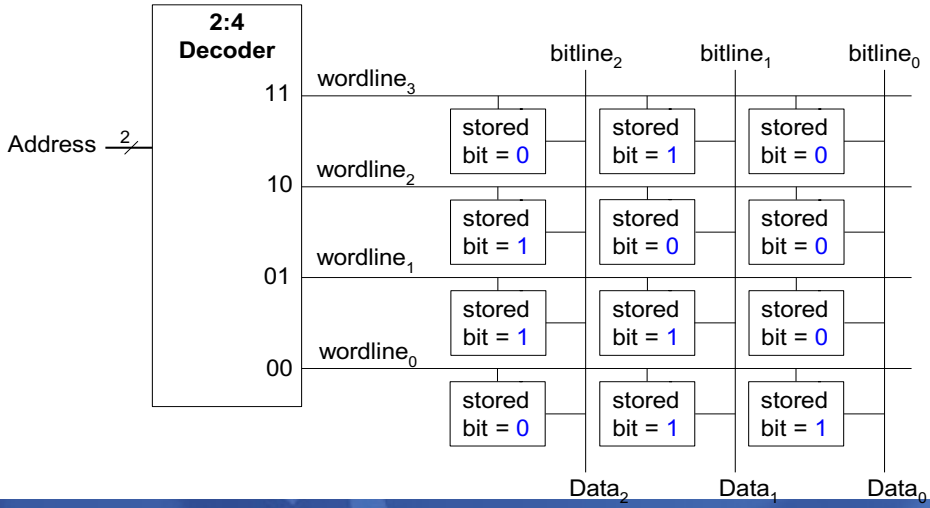
Implementation example: TOY main memory.

- Use 256 16-bit registers.
- Multiplexer and decoder are combinational circuits.



Memory Array

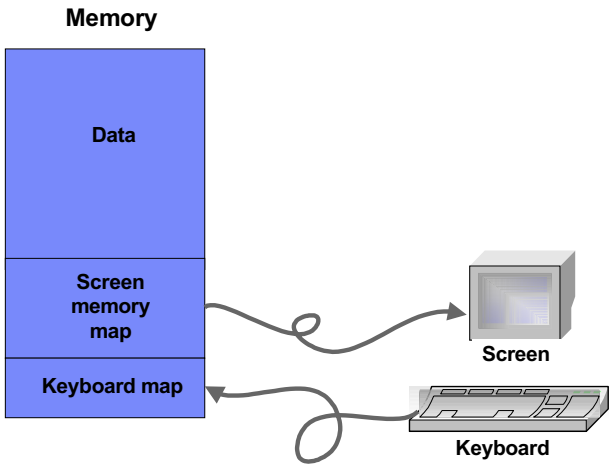
- Wordline:
  - like an enable
  - single row in memory array read/written
  - corresponds to unique address
  - only one wordline HIGH at once



Lecture / construction plan

- Instruction memory
- Memory:
  - Data memory
  - Screen
  - Keyboard
- CPU
- Computer

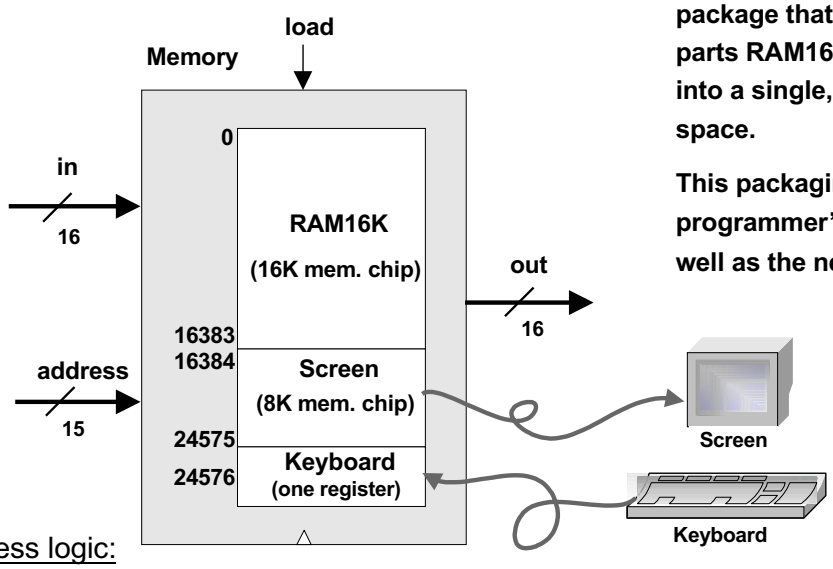
Memory: conceptual / programmer's view



Using the memory:

- To record or recall values (e.g. variables, objects, arrays), use the first 16K words of the memory
- To write to the screen (or read the screen), use the next 8K words of the memory
- To read which key is currently pressed, use the next word of the memory.

Memory: physical implementation



The Memory chip is essentially a package that integrates the three chip-parts RAM16K, Screen, and Keyboard into a single, contiguous address space.

This packaging effects the programmer's view of the memory, as well as the necessary I/O side-effects.

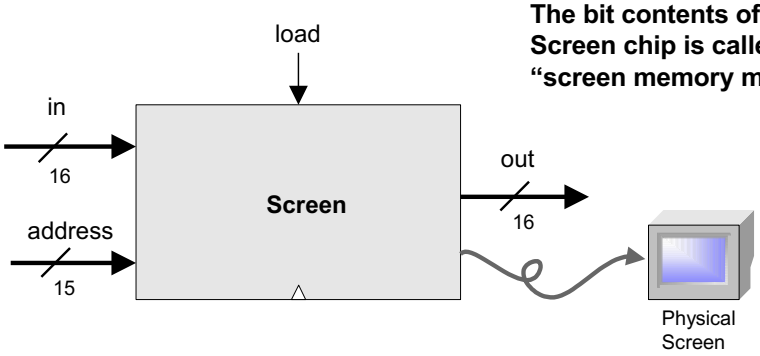
Access logic:

- Access to any address from 0 to 16,383 results in accessing the `RAM16K` chip-part
- Access to any address from 16,384 to 24,575 results in accessing the `Screen` chip-part
- Access to address 24,576 results in accessing the `keyboard` chip-part
- Access to any other address is invalid.

Lecture / construction plan

- Instruction memory
- Memory:
  - Data memory
  - Screen
  - Keyboard
- CPU
- Computer

Screen



The bit contents of the Screen chip is called the "screen memory map"

The Screen chip has a basic RAM chip functionality:

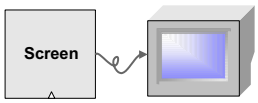
- read logic: `out = Screen[address]`
- write logic: `if load then Screen[address] = in`

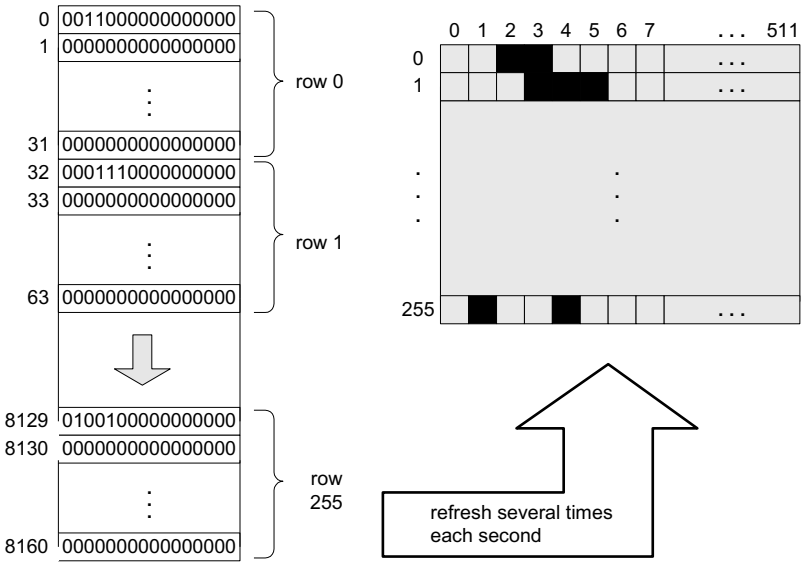
Side effect:

Continuously refreshes a 256 by 512 black-and-white screen device

Screen Memory Map

In the Hack platform, the screen is implemented as an 8K 16-bit RAM chip.

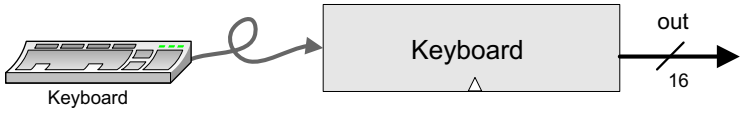




How to set the `(row,col)` pixel of the screen to black or to white:

- Low-level (machine language): Set the `col%16` bit of the word found at `Screen[row*32+col/16]` to `1` or to `0`
  (`col/16` is integer division)
- High-level: Use the OS command `drawPixel(row,col)`
  (effects the same operation, discussed later in the course, when we'll write the OS).

Keyboard



Keyboard chip: a single 16-bit register

Input: scan-code (16-bit value) of the currently pressed key, or 0 if no key is pressed

Output: same

Special keys:

Key pressed	Keyboard output	Key pressed	Keyboard output
newline	128	end	135
backspace	129	page up	136
left arrow	130	page down	137
up arrow	131	insert	138
right arrow	132	delete	139
down arrow	133	esc	140
home	134	f1-f12	141-152

- Synchronous vs. Asynchronous
- Signal

How to read the keyboard:

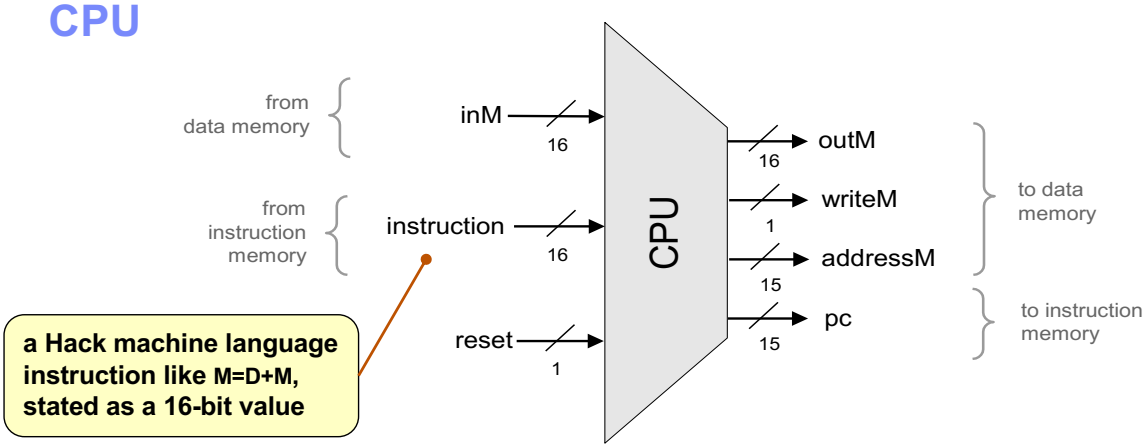
- **Low-level (hardware): probe the contents of the Keyboard chip**
- **High-level: use the OS command keyPressed()**

(effects the same operation, discussed later in the course, when we'll write the OS).

Lecture / construction plan

- Instruction memory
- Memory:
  - Data memory
  - Screen
  - Keyboard
- CPU
- Computer

CPU



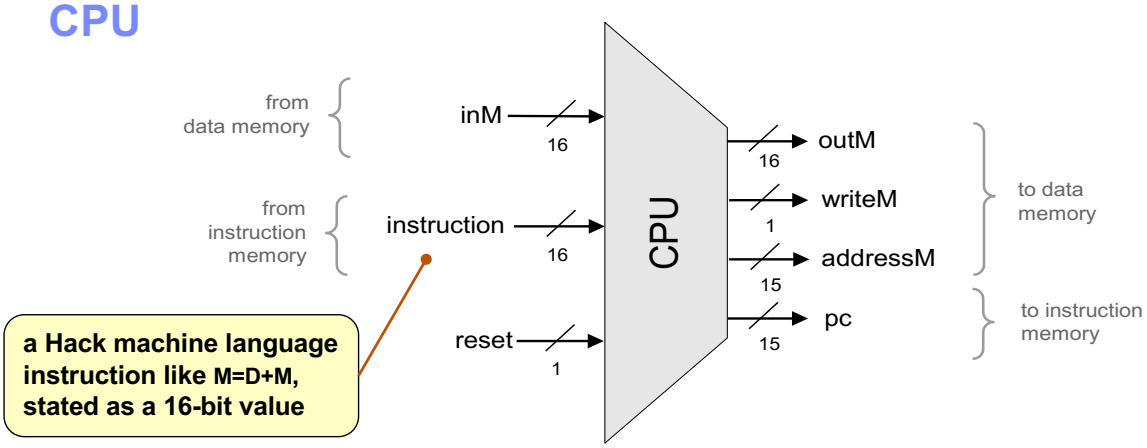
CPU internal components (invisible in this chip diagram): ALU and **3 registers: A, D, PC**

CPU execute logic:

The CPU executes the `instruction` according to the Hack language specification:

- The `D` and `A` values, if they appear in the instruction, are read from (or written to) the respective CPU-resident registers
- The `M` value, if there is one in the instruction's RHS, is read from `inM`
- If the instruction's LHS includes `M`, then the ALU output is placed in `outM`, the value of the CPU-resident `A` register is placed in `addressM`, and `writeM` is asserted.

CPU



CPU internal components (invisible in this chip diagram): ALU and 3 registers: A, D, PC

CPU fetch logic:

Recall that:

1. the `instruction` may include a jump directive (expressed as non-zero jump bits)
2. the `ALU` emits two control bits, indicating if the `ALU` output is zero or less than zero

If `reset==0:` the CPU uses this information (the jump bits and the `ALU` control bits) as follows:

If there should be a jump, the `PC` is set to the value of `A`; else, `PC` is set to `PC+1`

If `reset==1:` the `PC` is set to `0`. (restarting the computer)

The C-instruction

dest = comp; jump				*comp*						*dest*		*jump*		
binary:	1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

(when a=0) *comp*	c1	c2	c3	c4	c5	c6	(when a=1) *comp*
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

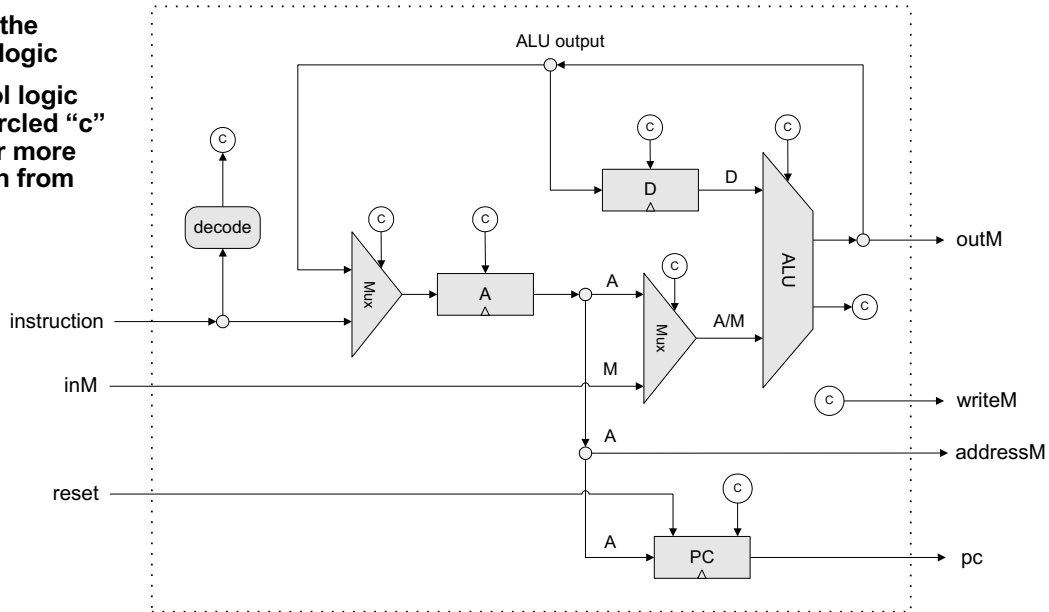
j1 (*out* < 0)	j2 (*out* = 0)	j3 (*out* > 0)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If *out* > 0 jump
0	1	0	JEQ	If *out* = 0 jump
0	1	1	JGE	If *out* ≥ 0 jump
1	0	0	JLT	If *out* < 0 jump
1	0	1	JNE	If *out* ≠ 0 jump
1	1	0	JLE	If *out* ≤ 0 jump
1	1	1	JMP	Jump

CPU implementation

dest = comp; jump | *comp* | *dest* | *jump*
binary: 1 1 1 a | c1 c2 c3 c4 | c5 c6 d1 d2 | d3 j1 j2 j3

Chip diagram:

- Includes most of the CPU's execution logic
- The CPU's control logic is hinted: each circled "c" represents one or more control bits, taken from the instruction
- The "decode" bar does not represent a chip, but rather indicates that the instruction bits are decoded somehow.



Cycle:

- Execute
- Fetch

Execute logic:

- Decode
- Execute

Fetch logic:

If there should be a jump,
  set PC to A
else set PC to PC+1

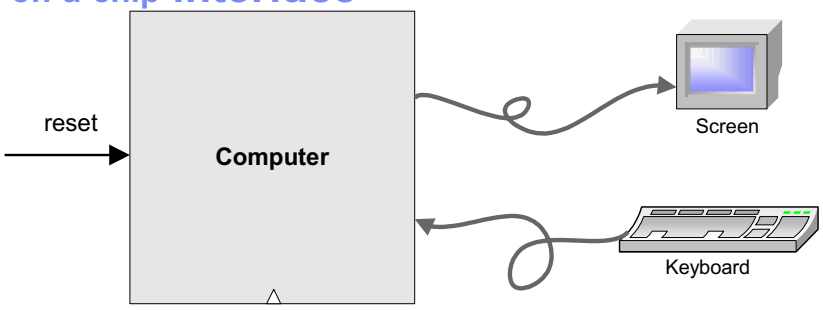
Resetting the computer:

Set reset to 1,
then set it to 0.

Lecture / Construction plan

- Instruction memory
- Memory:
  - Data memory
  - Screen
  - Keyboard
- CPU
- Computer

Computer-on-a-chip interface




```
Chip Name:  Computer  // Topmost chip in the Hack platform
Input:      reset
Function:   When reset is 0, the program stored in the
            computer's ROM executes. When reset is 1, the
            execution of the program restarts. Thus, to start a
            program's execution, reset must be pushed "up" (1)
            and "down" (0).

            From this point onward the user is at the mercy of
            the software. In particular, depending on the
            program's code, the screen may show some output and
            the user may be able to interact with the computer
            via the keyboard.
```

Perspective: from Here to a "Real" Computer

- Caching
- More I/O units
- Special-purpose processors (I/O, graphics, GPU, communications, …)
- Multi-core / parallelism
- Efficiency
- Energy consumption considerations
- And more ...

Perspective: Some Issues we haven't Discussed (among many)

- CISC / RISC (hardware / software trade-off)
  - complex instruction set computer vs. reduced instruction set computer
- Hardware diversity: desktop, laptop, hand-held, game machines, …
- General-purpose vs. embedded computers
- Silicon compilers
- And more ...

CISC vs. RISC

CISC

- Emphasis on hardware
- Multiple instruction sizes and formats
- Less registers
- More addressing modes
- Extensive use of microprogramming
- Instructions take a varying amount of cycle time
- Pipeline is difficult

RISC

- Emphasis on software
- Instructions of same set with few formats
- Use more registers
- Few addressing modes
- Complexity in computer
- Instructions take one cycle time
- Pipeline is easy