3a. Computer architecture and hardware (continued)

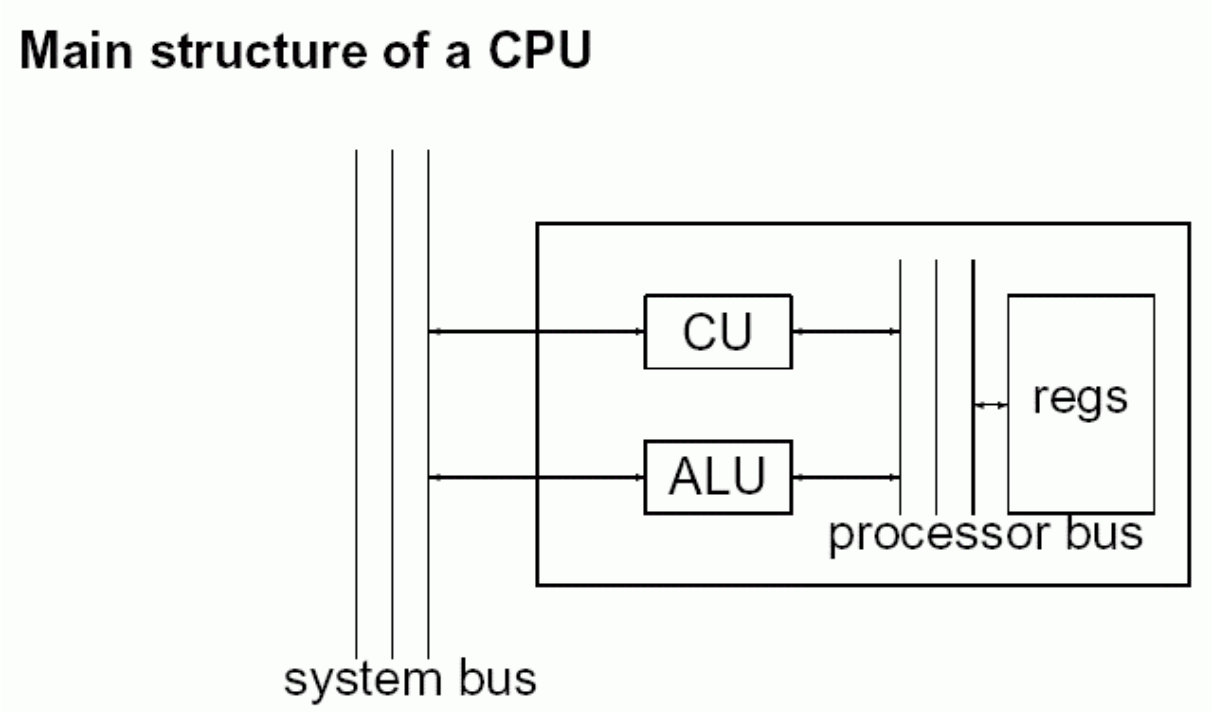
CPU components:

arithmetical-logical unit (ALU)

data registers:
data (inputs from system bus and output),
register address (input from control unit)

control unit (CU): instruction register, sequencer,
programme counter, status register *(see below)*

internal CPU bus for interconnection of the components



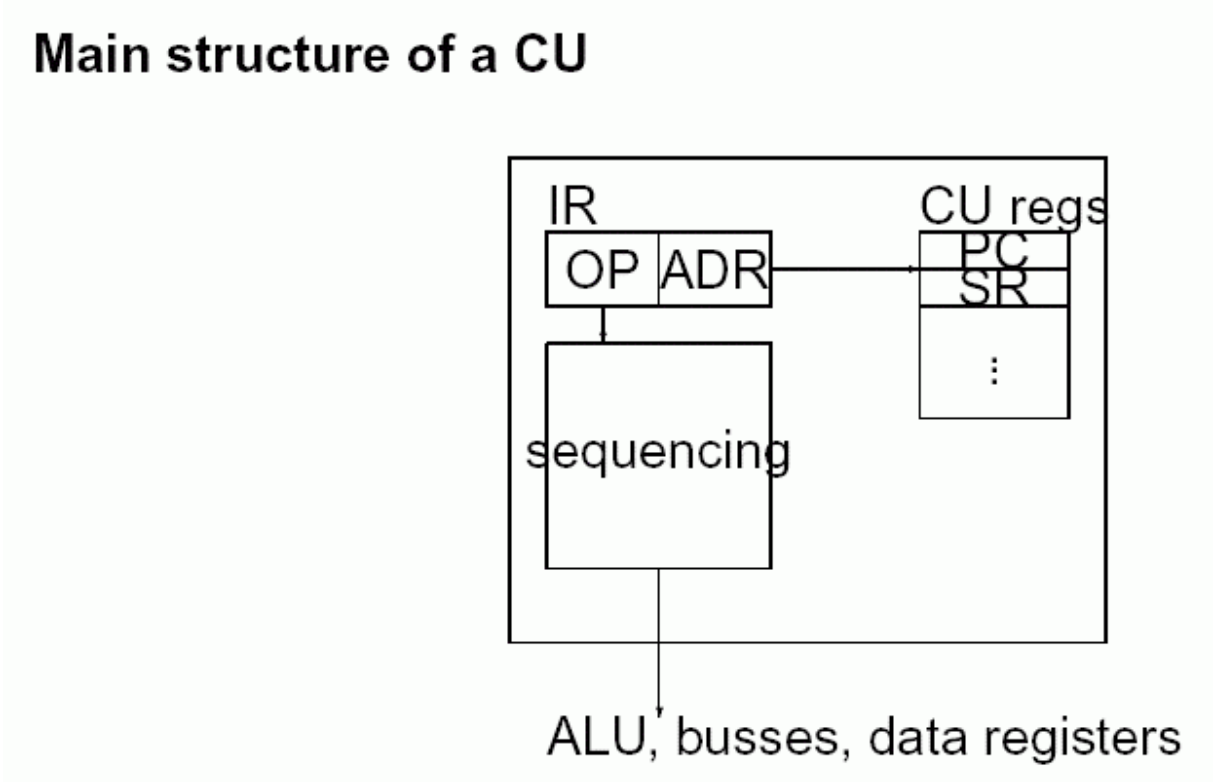
Components of the control unit (CU):

Instruction register (IR):
data (input from system bus), opcode (output to ALU), modifier (output to ALU), data address (output to system bus, to data registers)

Sequencer: controls sequence of operations for content of the instruction register

Programme counter (PC):
data (input from ALU, output to system bus), add2/add4 (inputs from instruction register)

Status register (SR):
flagsIn (input from ALU), flagsOut (output to ALU), load (input from instruction register)



Registers of the CPU

PC (program counter): A register containing the **address of the next instruction to be executed**. Normally just incremented by the length of the current command in cells after each load. Jump instructions load this register with some other value.

IR (instruction register): Contains the binary code of the **currently executed instruction**.

SR (status register): Contains flags signaling **properties of last computation performed**; they are used to control if conditional jumps are performed. Example: In some bits, it signals if the last result was zero, negative, there was an overflow etc.

Data registers (multipurpose registers): Contains typically **data items** to be worked on quickly, as sources and targets for movements and ALU-operations.

A typical 6-step ALU machine cycle:

Different types of cycles: ALU, load, store; "jump" as a special load instruction

ALU cycle: controlled by sequencer in CU

1. **Fetch Instruction**: The content of PC is used as address of the next instruction to be executed. The instruction at this address is fetched from MEM and put into IR.

2. **Decode**: Decide which sequence of steps are to be executed for the current content of IR.

3. **Fetch Operands**: As described by sequence of steps to be executed, fetch data from data registers into internal ALU registers. Register addresses are given in IR.

4. **Execute**: With the operands fetched, execute the arithmetical or logical operation described by the IR content. Store result in an intermediate result register.

5. **Store Result**: Write out the result of the instruction into register, to a register address given in IR. Set status register according to result.

6. **Increment PC**: Compute address of next instruction to execute in PC. This is often done in parallel to the sequence from step 2 to step 5.

Assembler Code:

Low-level computer language

machine code: programs as number sequences

assembler: symbolical instructions; memory layout

each instruction: (a) one operation OP, (b) one or more explicit operands, (c) (possibly) implicit operands

Example: `ld addr_j, r1` to LOAD content of MEM at address at `addr_j` to register with number 1 (two explicit operands)

Implicit operands: sometimes encoded in OP, if only few registers used (older or smaller microprocessors)

Addressing operands and target

Direct operand: One value to work with is **contained in instruction word**

Example: `add #13,r1`

Register operand: Both numbers to be worked with are in registers

Example: `add r1,r2`

Memory operand: One value to be worked with is in the memory

Example: `ld addr_i,r1`

Machine code structures

- **Load/store code:** Arithmetics only on registers; memory content must be transferred into a register before it can be worked upon.

- **One-address code:** One operand of an arithmetical operation comes from an explicitly given source (direct, memory or register), the other is an implicit default register.

- **Two-address code:** two data sources are given explicitly; the result of the operation is put into one of them.

- **Three-address code:** two data sources and a result target are given explicitly.

Example of two-address assembler code

```
ld #<value>,<register>      ld <address>,<register>
st <register>,<address>

add <register>,<register>   add #<value>,<register>
cmp <register>,<register>   cmp #<value>,<register>
sub <register>,<register>   sub #<value>,<register>

jmp #<address>   jeq #<address>   jne #<address>
jle #<address>   jlt #<address>   jge #<address>
jgt #<address>

<label>:

byte <num. of allocated bytes>
```

Different variants of the same programme:
(left column: programme in a high-level language – Java or C)

```
int i=0;
int j=0;
for(j=0;
    j<10;
    ++j)
{
  i += j;
}
```

Non-optimized

```
00: addr_i: byte 4
04: addr_j: byte 4
08: program_start:
08:   ld #0,r1
12:   st r1,addr_i
16:   ld #0,r1
20:   st r1,addr_j
24: loop:
24:   ld addr_j,r1
28:   cmp #10,r1
32:   jle #loop_end
36:   ld addr_i,r1
40:   ld addr_j,r2
44:   add r2,r1
48:   st r1,addr_i
52:   ld addr_j,r1
56:   add #1,r1
60:   st r1,addr_j
64:   jmp #loop
68: loop_end:
```

Optimized

```
00: addr_i: byte 4
04: addr_j: byte 4
08: program_start:
08:   load #0,r1
12:   st r1,addr_i
16:   st r1,addr_j
20: loop:
20:   ld addr_j,r1
24:   cmp #10,r1
28:   jle #loop_end
32:   ld addr_i,r1
36:   ld addr_j,r2
40:   add r2,r1
44:   st r1,addr_i
48:   add #1,r2
52:   st r2,addr_j
56:   jmp #loop
60: loop_end:
```

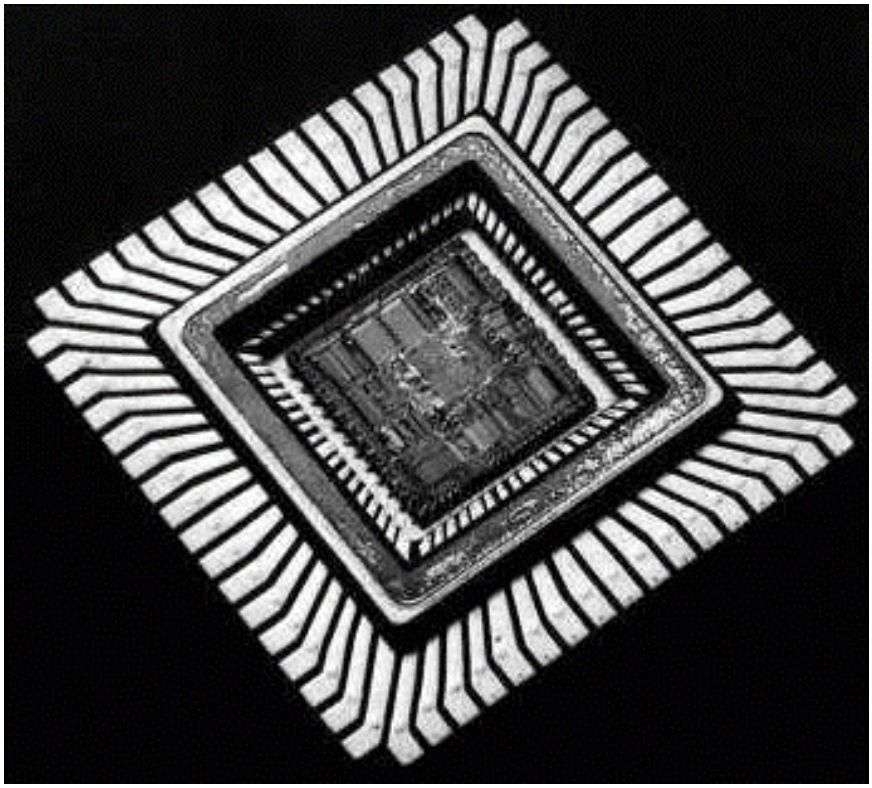
'j' in register 1

```
00: addr_i: byte 4
08: program_start:
08:   ld #0,r1
12:   st r1,addr_i
16: loop:
16:   cmp #10,r1
20:   jle #loop_end
24:   ld addr_i,r2
28:   add r1,r2
32:   st r2,addr_i
36:   add #1,r1
40:   jmp #loop
44: loop_end:
```

Physical realisation of the CPU and memory (hardware):

Semiconductor (silicon) chips
highly integrated arrangements of electronic switches (transistors) and electronic storage elements on (mostly) 2-dimensional plates
produced with optical-chemical imprinting methods

A microprocessor:



Alternative techniques:
- optical computing (using light instead of electricity)
- quantum computing (using quantum effects, especially for parallelisation)

both are currently more of theoretical interest,
no mature technologies

other hardware components:

The periphery

(= all what is connected with a computer but the task of which is not primarily to compute)

Keyboard
normally resembling traditional typewriter keyboard, with additional number panel and function keys
– electrical switches below the buttons

Caution: placement and lettering of keys depends on country
Operating system of the computer must "know" the key scheme

Mouse
most important graphical input device
– mechanical mouse: ball with motion sensors
– optical mouse: tracking of motions using light emission and sensing
Motion signals are interpreted by operating system and/or application programmes

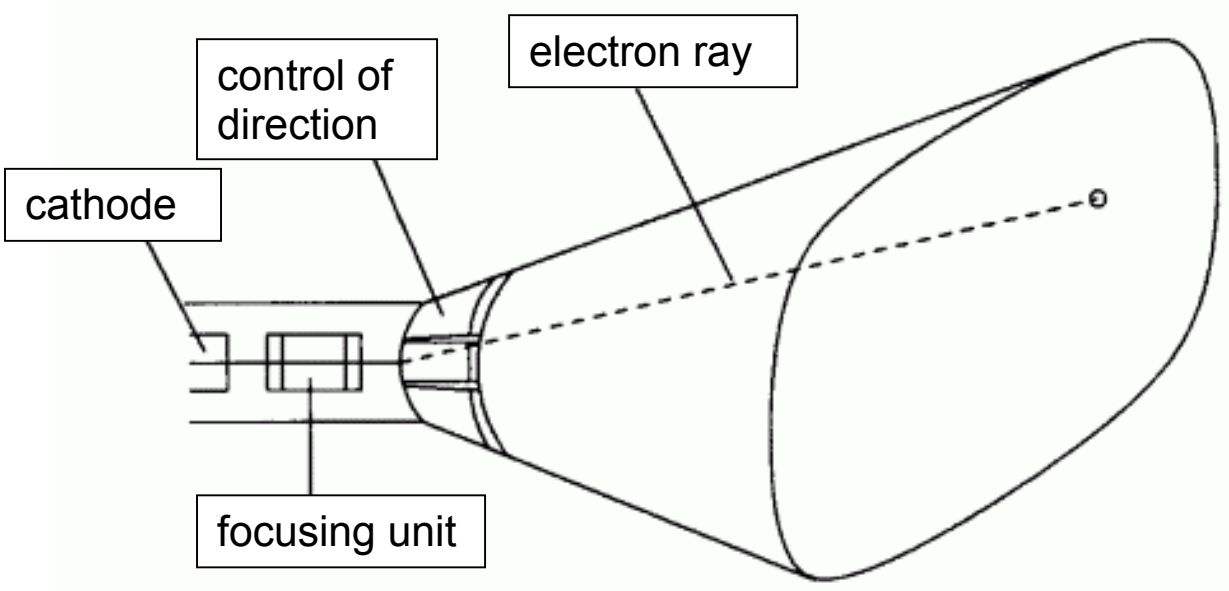
Display

transforms electrical signals in non-permanent optical signals
traditionally solutions technically different from television (but currently trend to converge: digital TV, multimedia programmes...)

2 main techniques:
- cathode ray tube (CRT)
- liquid crystal disply (LCD)

Cathode ray tube:

- Electron ray is emitted by a cathode
- direction controlled by magnetic fields
- hits substance on the inner side of the screen glass which emits light when stimulated by electrons



electron ray hits pixel only for short time
but: *persistence* of light emission for some time after stimulation (phosphorescence)

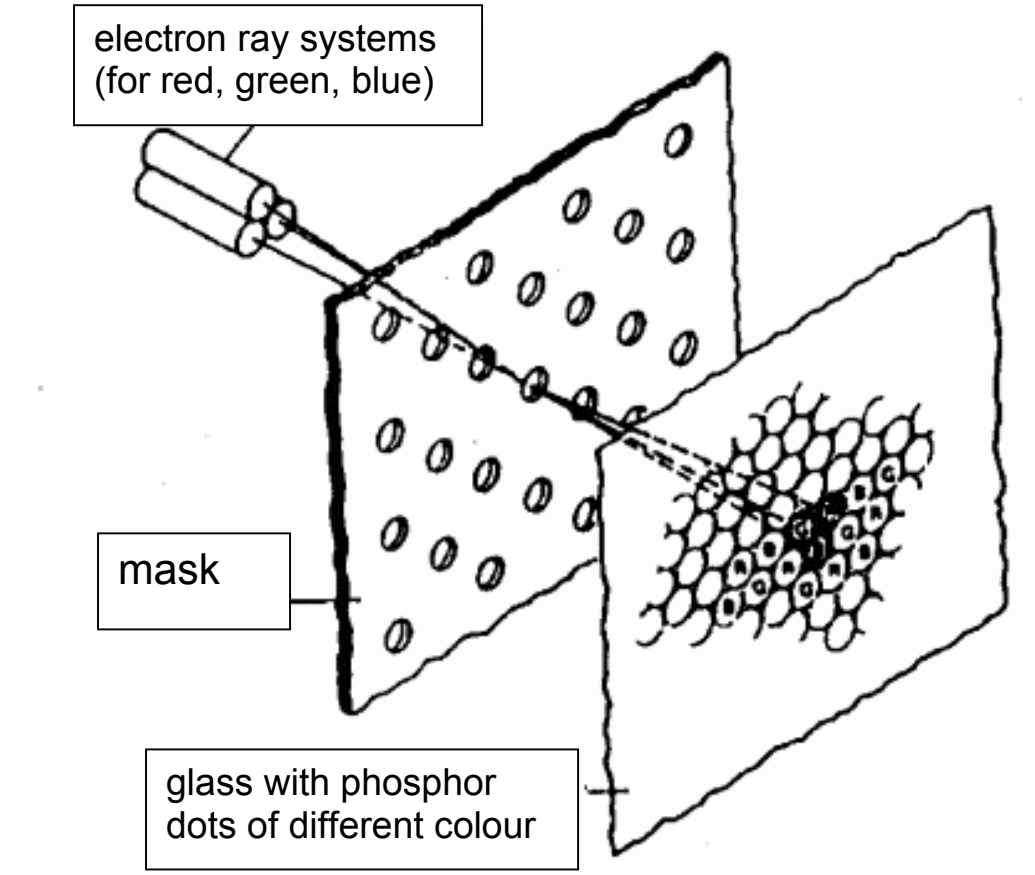
$\Rightarrow$ persistence determines the necessary *refresh rate* to ensure the impression of a flicker-free, standing image

picture repetition frequency (for refresh):
usually between 30 and 80 pictures per second

(if it is too small in relation to the persistence: Blurring when motion occurs, "ghost images")

Colour display:
Shadow mask colour CRT

3 separate cathodes
screen covered with triples of red, green and blue phosphor dots
mask with small holes in front of the phosphor dots ensures that each of the 3 rays hits only its appropriate colour dots
colour perception by additive composition of red, green and blue light



(other arrangements of the dots exist, e.g. in lines instead of triangular – "inline displays")

Evaluation of the CRT technique:

(+) high resolution
(+) good colours, high luminance
(+) relatively cheap
(+) mature technique, low defect rate
(–) vacuum tubes are heavy and clumsy
(–) high power input (~ 80 W)
(–) flickering
(–) geometrical distortions
(–) X-ray emission

Liquid Crystal Display (LCD):

liquid crystals (discovered 1888 by Reinitzer):

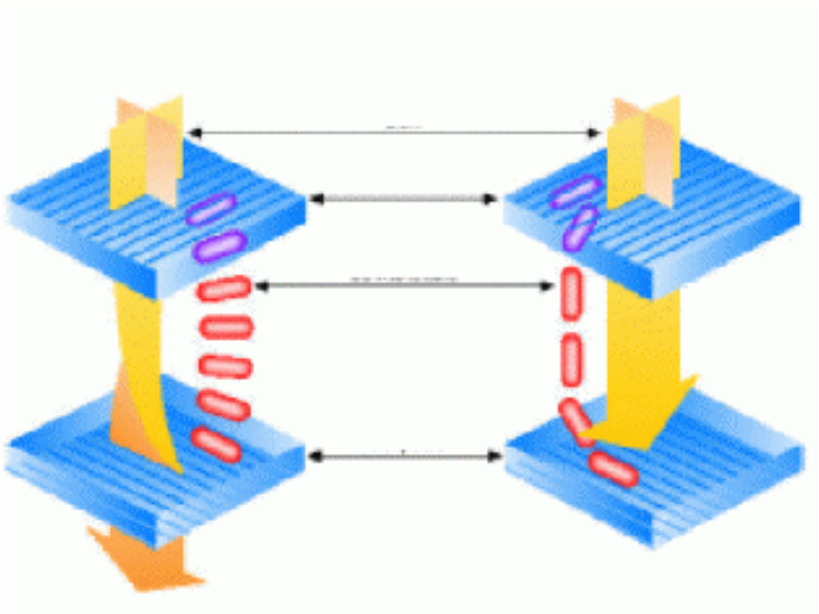
- organic molecules, oval or disk-like form
- axes tend to be orientated in parallel

Twisted nematic cells:
liquid crystals enclosed between two glass plates with polarising characteristics, caused by tiny ridges rotated by 90° against each other

glass plates transmit only light oscillating in one specific direction (polarised light)

- Without tension, the LC molecules arrange in a way that causes a light beam to twist and to pass both glasses

- With tension (5 V), the molecules reorientate and the light passing the first glass is not twisted $\Rightarrow$ it cannot pass the second glass

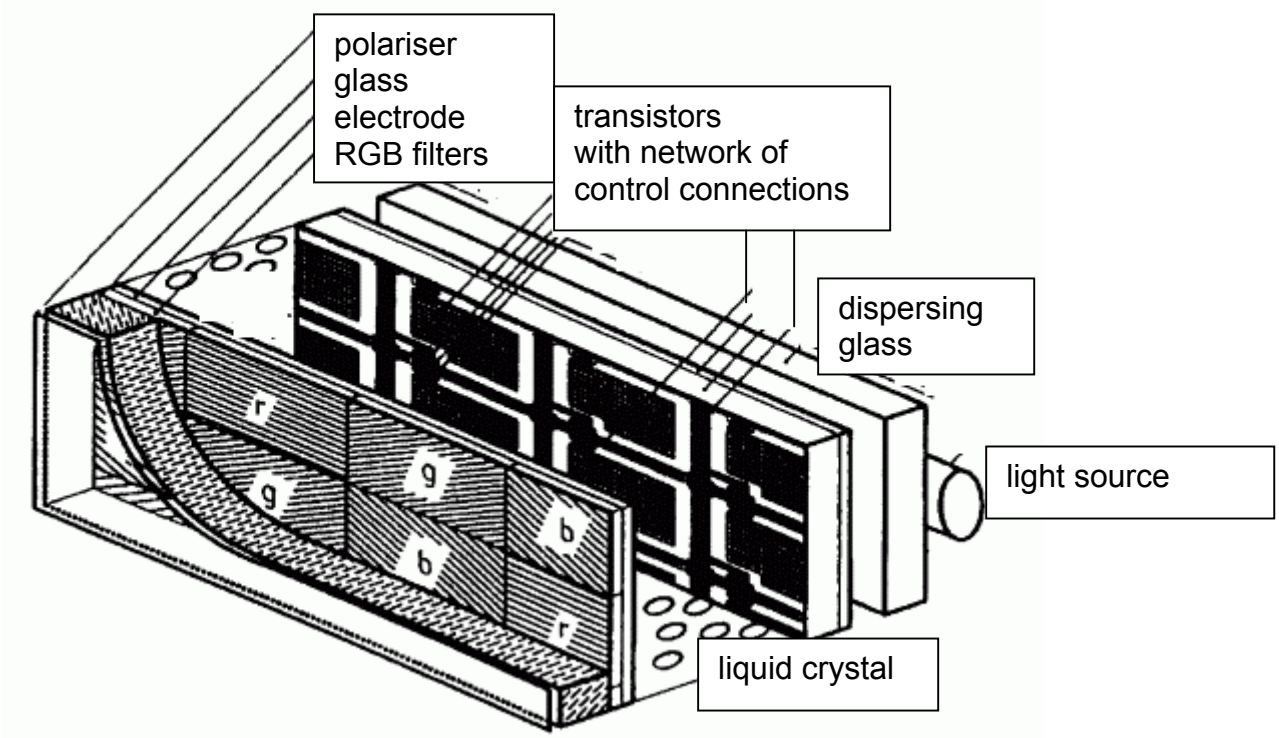


Each cell of the screen must be controlled separately

⇒

Thin Film Transistor (TFT) technology:
each cell has its own electronic switch (transistor)
positioned in one corner of the pixel
production with photochemical techniques

Structure of an LCD:

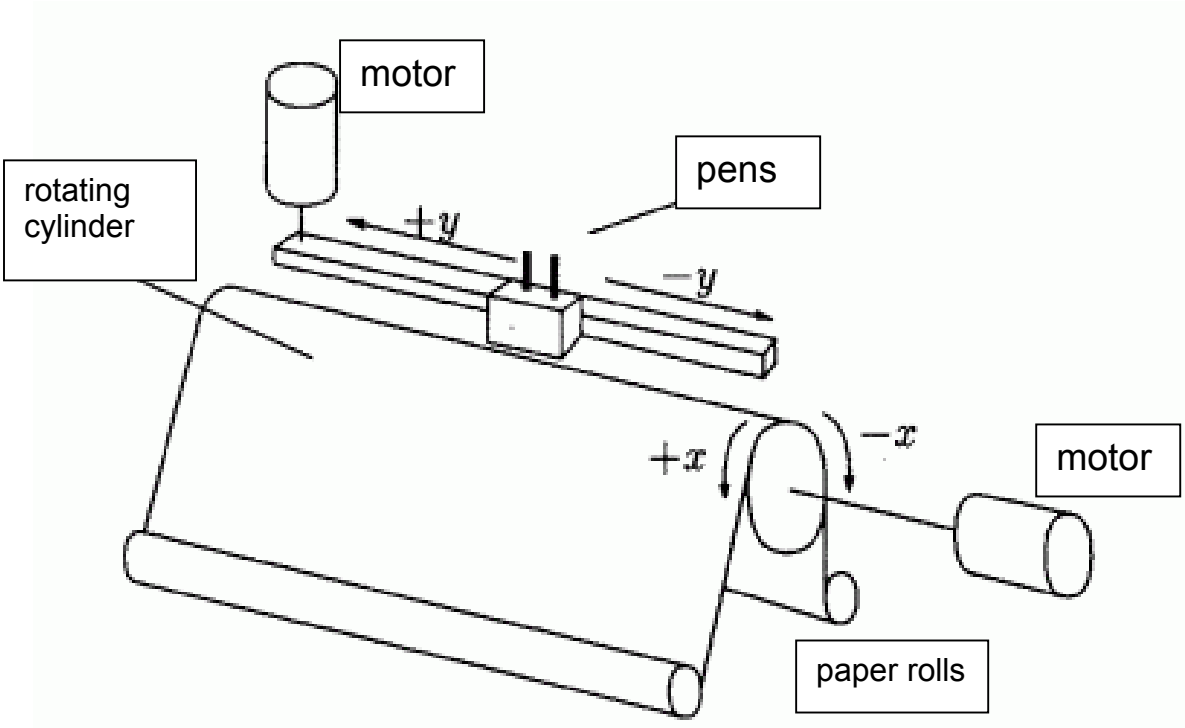


Evaluation of the LCD technique:

(+) low power input (25 W)
(+) low voltage
(+) no flickering
(+) good contrasts
(+) digital
(+) low weight, small
(+) appropriate for mobile machines
(–) cells work passively: extra light source necessary
(–) narrow view sector
(–) production relatively complicated

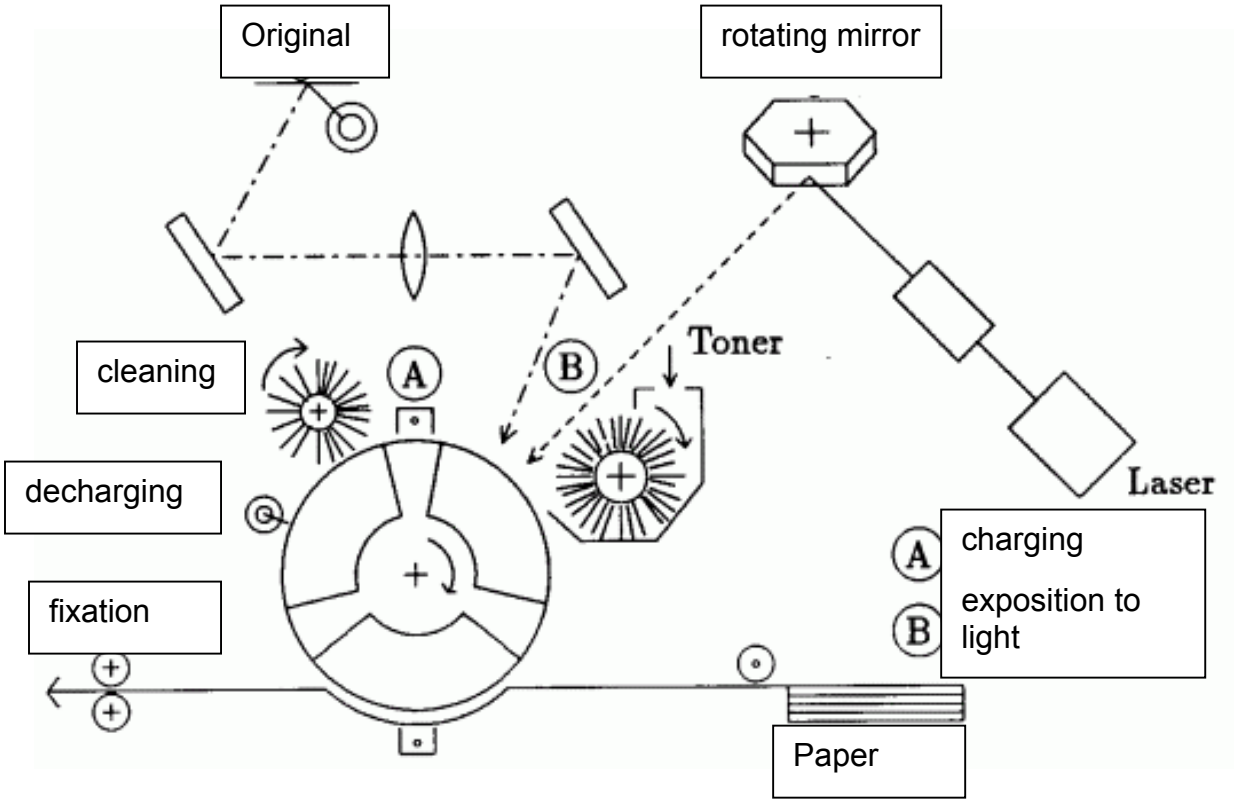
Further graphical output devices:

Plotters



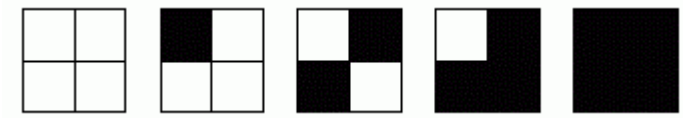
Printers:

- matrix printer (needles, mechanically printing, typewriter principle)
- inkjet printer (jets pressing ink on the paper)
- *xerographic printers* ("laser printers")
  (same principle as in copy machines):



color printing realised by serial combination of several stations with differently coloured toners.

Realisation of gray values and colours with printers which can only print black and white (or only few colours):
Halftone technique



halftone matrices for 5 gray values using only black and white

4. Basic facts about operating systems and programming languages

Operating system (OS):
Basic software which manages all technical procedures necessary when using a computer, like
- starting the computer when it is switched on ("booting")
- allocating memory space
- reading and writing files
- organising files (e.g., in "directories")
- communicating with the periphery
- organizing input and output
- control of programme execution

Different philosophies of giving commands to the OS:

- typing the command in a command line
- clicking with the mouse on (virtual) buttons, menues, sliders, icons...: The command line is replaced by a GUI (Graphical User Interface), often with "desktop metaphor" (attempt to mimick a real office desktop – but it isn't really so!)

Important contemporary operating systems:

- Microsoft (MS) Windows (variants XP, 2000, NT, 98...)
- Unix (several variants exist, e.g. for Sun, IBM, SGI...)
- Linux (similar to Unix, free software)
- Mac-OS (Apple)
- MS-DOS (old predecessor of MS-Windows): pure command line system

Files:
sequential collections of data, stored on data carriers (harddisks, floppies, CD-ROM, DVD, tape...),
can be identified by *file names*.

Caution:
The sequential order of data in a file does generally not directly correspond to a *physical order* of the data on the carrier medium!
Data are often "physically fragmented"
(you can use defragmentation programmes to speed up access to files on your computer)

The data carrier must be formatted and initialised:
this induces a subdivision in so-called "cylinders" and "segments" (abstract notions!)
On each data carrier, there is an FAT = File Access Table – to find the files on the carrier

In nearly all operating systems, there is the possibility to organise files in hierarchical *directories* (or "folders").

Navigation in the tree of directories:
- in MS-DOS and Windows:
  X:\directory\directory\... \filename = "path"
  (X: is the name of the drive,
  normally: A: floppy disk drive,
  B: floppy disk drive,
  C: hard disk drive,
  D: hard disk drive,
  E: CD-ROM drive or DVD drive)
- in Unix and Linux:
  /directory/directory/... /directory
  - there is only 1 root directory
  - the separating character is "/" instead of "\"

Naming of files:

- in MS-DOS and Windows:
  often the name structure name.extension
  is used,
  where "extension" is short (preferentially 3 letters)
  – historical reasons (MS-DOS).
  Attention: the extension is often not displayed
  (this feature can be switched on or off).

  Typical extensions:

  .EXE executable programme
  .COM (small) executable programme
  .TXT text file
  .DOC Word document (formatted text)
  .GIF image file
  .JPG image file
  .PS PostScript file (printable)
  .PDF document file for Acrobat Reader software
  .HTM or .HTML web document
  .JAVA Java code (programme source code)
  .BAT batch file: command sequence for the OS

  Special files: AUTOEXEC.BAT, CONFIG.SYS,
  WIN.INI
  - control booting behaviour and configuration
  of the computer (known periphery, keyboard
  language, mouse...)

- in Unix and Linux:
  less restrictions to file names than in DOS
  names beginning with a dot (.) designate "hidden" files
  Examples: .login , .profile (control files like AUTO-
  EXEC.BAT)

Files can have *attributes*:

- hidden
- write-protected
- Archive
- system file ...

Systematic *access control flags* under Unix and Linux:

r = read (you are allowed to read)
w = write
x = execute

each can be given for:

u = user
g = group
o = others

The most important commands to manipulate files

	in MS-DOS *(under Windows: in the* *MS-DOS input facility)*	*in Unix / Linux*
Copying	`copy` source target(path) specification of the target is optional in DOS, but always necessary in Unix/Linux!	`cp` source target(path)
Listing files	`dir` pathname specification of the path name is optional	`ls` pathname `ls -al` pathname *(more informative version)* specification of the path name is optional
Deletion of file (delete, remove)	`del` filename	`rm` filename
Creation of a directory (make directory)	`mkdir` directoryname *or* `md` directoryname	`mkdir` directoryname
Deletion of a directory	`rmdir` directoryname *or* `rd` directoryname	`rmdir` directoryname
Showing the content of a file (for lisible files only)	`type` filename `|more`	`cat` filename `|more`
Changing the current directory	`cd` pathname	`cd` pathname

currently valid path is encoded by: `.` next-higher directory by: `..`

Up to now: 3 sorts of programming languages were introduced

- machine code
  (machine-specific, directly executable,
  sequences of bits, resp. half-bytes – hexadecimal code:
  hard to read for humans)
- assembler languages
  (translation of machine code in mnemonic abbreviations like
  "add", "jmp"...; but still machine-specific and
  of a technical nature)
- operating system commands
  and several of them combined: "batch files"
  → OS translates them into machine code instructions
  machine-independent, but specific for the OS
  – only of limited power and flexibility

"*High-level programming languages*":

artificial languages
- designed for humans: lisible, easy to learn, intuitive
- also designed for computers: efficient to implement
- providing full power and flexibility

Programmes written in a high-level language must be translated into machine instructions

Two ways of implementation:

- *Interpreter*

simulates a computer which understands the high-level language (i.e., the *source code* written in it)
→ enables interactive programming

source code → interpreter ↔ computer

Examples of languages: Smalltalk; simple versions of Basic

- *Compiler*

translates the whole source code into machine code
→ this can enable an improvement of efficiency (optimisation) already during translation (*compilation*)

source code → compiler → target code → computer

Examples of languages: C, C++, Pascal, Fortran

In the case of larger programmes, often two steps are done:
- Compiler translates single modules (parts of the software)
- *Linker* merges these into an executable programme

Development tools for specific languages often contain editor (for source code text) + compiler + linker + debugger + further tools.

Combination of both approaches (aim: platform independence, "write once – run everywhere"):

source code → compiler → intermediate code
(byte code)
indep. of processor and OS

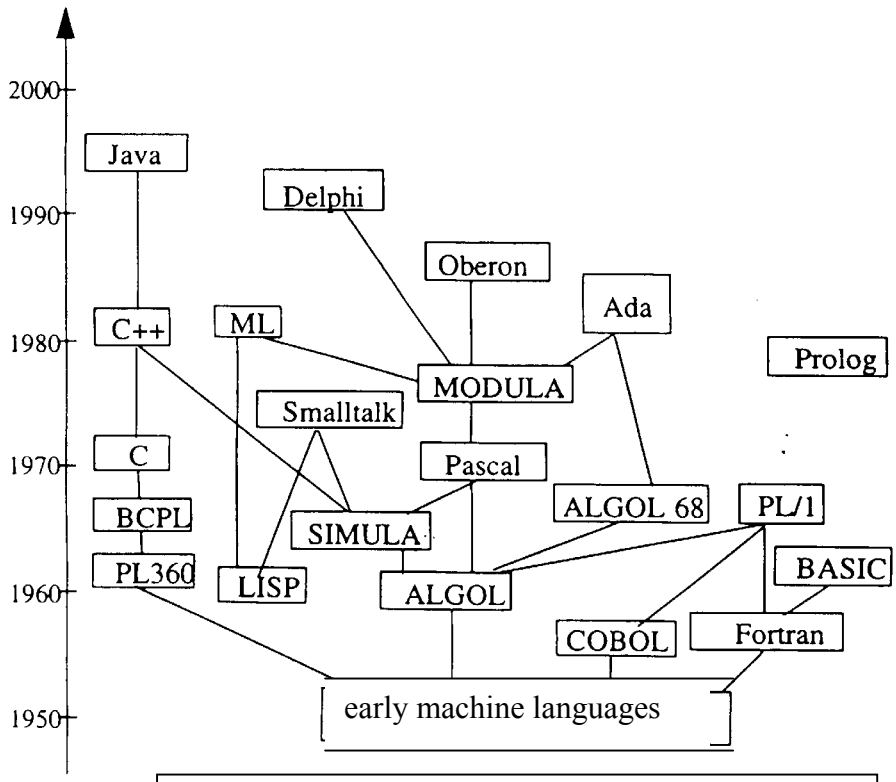
intermediate code → interpreter ↔ computer
(e.g.
JVM = Java virtual machine)

Example language: Java.

To be distinguished:

- source code files (name extensions .java, .c, ...)
- objectcode files (compiled, but not yet linked modules)
- intermediate code files (.class)
- executable programmes (applications; name extensions .exe, .com or without extension)
- applets: executable programmes which cannot run alone but only from other applications, usually from web browsers (are not allowed to modify files on harddisk)
- control files for compiler or development tools (.prj, makefile)

"Evolution tree" of high-level programming languages:



Development of programming languages

Very widespread programming languages:

FORTRAN (1955) ("*Formula Translator*")
particularly for numerical computations, programme libraries

COBOL (1960) ("Common Business Oriented Language")
classical commercial applications

C, C++ (1971, 1992)

PASCAL (1971)

JAVA (1995)
internet applets

To be distinguished from "proper" programming languages are document description languages like, e.g.,
HTML, XML, PDF, RTF, TEX

Intermediate form: PostScript
(Printer language, but incorporates also the possibilities of a universal programming language)

How to order the many programming languages in a meaningful way?

"**Paradigms of programming**":
basic ideas, philosophies behind the language concepts, patterns of comprehension of "programming"

1. Control flow paradigm (von Neumann paradigm)

is the basis of classical procedural programming

- also realised in assembler code

High-level languages: Fortran, Basic, Pascal, C.

Computer = machine for changing values of variables.

Programme = plan for the process of calculation with specification of commands and flow of control (e.g., loops).

Finding a programme: To find elementary single steps and to bring them into a flexible order.

Example (in C):

```
int i, n, z;
n = 10;
i = 1;
while (i < n)
      {
      z = i*i;
      printf("%d\n", z);
      i = i+1;
      }
printf("Finished.\n");
```

2. Object-oriented paradigm

Typical programming languages: Simula, C++, Smalltalk, Delphi, Java

Computer = Environment for virtual objects

Programme = Listing of (object) *classes*, i.e., of general specifications of objects which can be (multiply) created and destroyed at runtime of the programme and which can communicate with each other.

Finding a programme: Specification of the classes (data and methods) which determine object structure and behaviour.

Example (in C++):

```
class matrix
   {
   float field[3][3];
   public:
   void initmatrix(float value);
   void identitymatrix();
   void showmatrix();
   friend matrix sum(matrix a, matrix b);
   };
```

3. Functional paradigm

(applicative programming, McCarthy paradigm)

Programming languages: Lisp, Lambda calculus, APL, FP systems

Computer = machine which forms generalisations of operations, i.e., which can define functionals (like construction of new notions in mathematics)

Programme = nested expression of applications of functions

Finding a programme: specification of functions which can solve the problem

Spezifikation von Funktionen, die das Problem lösen

Example (FP system according to Backus):

```
def innerprod = (/+) ° (a*) ° trans
```

defines the inner product of two vectors of arbitrary dimension

Note: no sequentialisation, no declarations of variables, very compact programmes are possible.

4. Rule-based paradigm (van Wijngaarden paradigm)

Programming languages:
PROLOG, rule-based AI languages, L-systems, Intran

Computer = transformation machine for states or structures

There is a *current state* (a current structure) which is transformed as long as this is possible.

Working process: process of search and application (of rules) – "matching" and "rewriting".

Programme = set of transformation rules.

Finding a programme: Specification of the rules.

Example (in Intran):

```
GCD OF M, N where
    M=N -> M;
    M>N -> GCD OF M-N, N;
    M<N -> GCD OF M, N-M.
```

Real programmes often contain elements from different paradigms.

Examples:

- procedural (imperative) parts in Simula, C++ and Java

- declaration of variables in L-systems, etc.

Using a certain paradigm of programming does not yet determine the choice of the programming language!

Example: Object-oriented programming (OOP) is also possible in Fortran or C (but with high effort).

But: Using the "appropriate" programming language is helpful because it already offers the constructions needed for realising the paradigm – thus it is not necessary to define them again (e.g. classes in OOP).