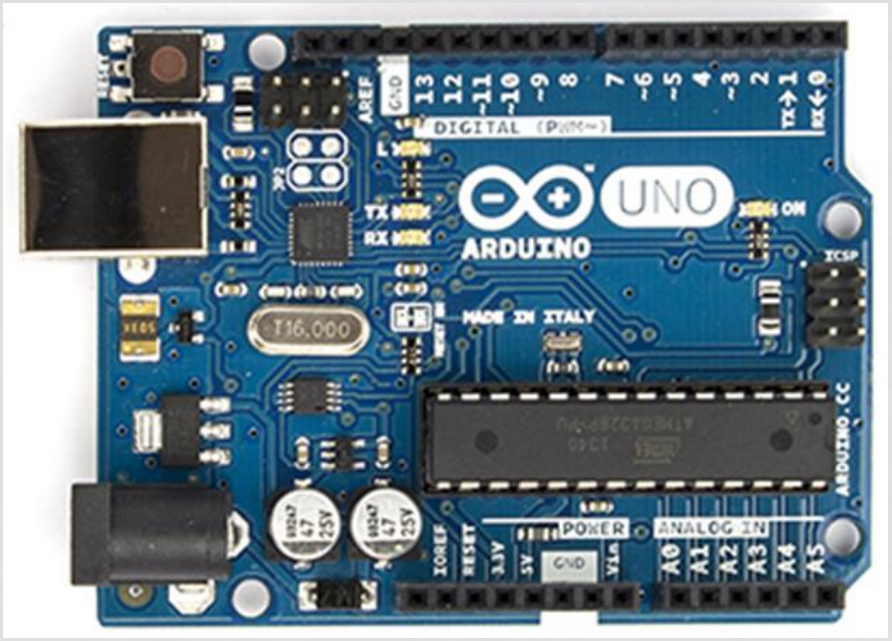

DIGITAL (PWM~)
UNO
ARDUINO
MADE IN ITALY
POWER
ANALOG IN
ARDUINO.CC

CS

WEEK 3-1

AGENDA

week	onderwerp	P&H	AT	Dijkstra
1	coderingen en talstelsels representatie van getallen optellen en aftrekken vermenigvuldigen en delen logische poorten schakelingen met poorten geheugen-elementen systeemklok & timers	App. B2, B3, B7, B8, B9 2.4 3.2 t/m 3.5 app B	App. A, B 3.1, 3.2, 3.3	H1 H2
2	typen computers 8 great ideas organisatie van de computer CPU intern, instructies uitvoeren geheugen systeem adres- en databus byte ordering pipelining de AVR MCU	1.1 t/m 1.4 2.12 4.1 t/m 4.5	1.3 2.1, 2.2 3.7	H3 6.1 en 6.2 7.1 en 7.2
3	typen geheugen caching opslag (ssd, harddisk) translating and starting a program parallelle architecturen - h/w multi-threading - multicore - GPU	5.2, 5.3 6.4 t/m 6.6	2.2, 2.3 7.3, 7.4 H8	4.1 7.3

AGENDA

- **typen geheugen**
- caching
- harddisk
- compiler, interpreter en VM

TYPEN GEHEUGEN

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

TYPEN GEHEUGEN

- RAM :
  - verlies data bij power down (= 'volatile' of 'vluchtig')
  - SRAM statisch (“flipflop”)
  - DRAM dynamisch (“transistor + condensator”)
    - DRAM controller nodig voor periodieke refresh
    - SDRAM is DRAM met synchroon interface, d.w.z. system clock vervangt control line
  - SRAM sneller en neemt minder energie dan DRAM, maar duurder
- ROM :
  - Mask ROM (permanent)
  - PROM (1x programmeren; burning fuses)
  - EPROM (alleen wissen met UV licht)

TYPEN GEHEUGEN

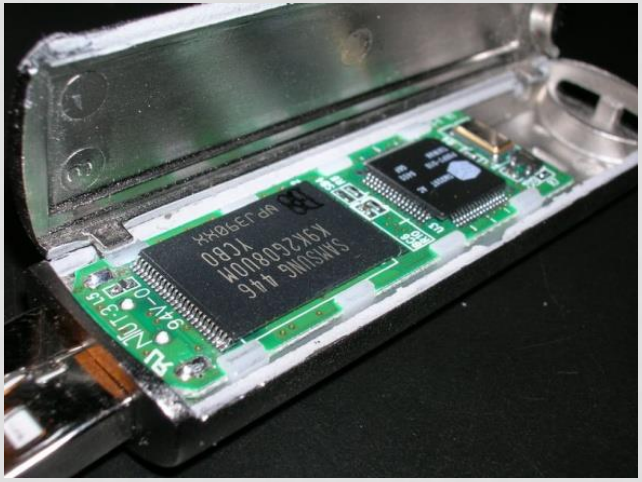
- Hybrid / NVRAM (Non Volatile RAM) : EEPROM en flash
- EEPROM
  - erase per byte
  - wissen kan elektrisch
  - lezen en schrijven langzaam
  - geschikt voor configuratie data
- flash
  - later type EEPROM ("MOSFET met extra floating gate")
  - erase *per block* (256 .. 4K bytes)
  - schrijven en wissen gaat langzaam
  - er zijn flash filesystems (linux : JFFS2)
  - probleem : memory wear

TYPEN GEHEUGEN

- behoefte aan zowel vluchtig (data) als niet-vluchtig geheugen (code)
- flash gebruikt als vervanging van hard disk :
  - SSD (Solid State Drive)
  - minder warmte, minder voeding, kleiner en betrouwbaarder
  - OS en applicatie opslaan in flash
  - booten van flash
- flash is erg langzaam i.v.m. RAM

FLASH STORAGE

- non-volatile semiconductor storage
  - 100× – 1000× faster than disk
  - smaller, lower power, more robust
  - but more $/GB (between disk and DRAM)



TYPEN GEHEUGEN

Type	Category	Erasure	Byte alterable	Volatile	Typical use
SRAM	Read/write	Electrical	Yes	Yes	Level 2 cache *data*
DRAM	Read/write	Electrical	Yes	Yes	Main memory (old)
SDRAM	Read/write	Electrical	Yes	Yes	Main memory (new)
ROM	Read-only	Not possible	No	No	Large volume appliances
PROM	Read-only	Not possible	No	No	Small volume equipment
EPROM	Read-mostly	UV light	No	No	Device prototyping
EEPROM	Read-mostly	Electrical	Yes	No	Device prototyping
Flash	Read/write	Electrical	No	No	Film for digital camera *code*

MEMORY HIERARCHY

- store everything on disk
- copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - main memory
- copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - cache memory attached to CPU

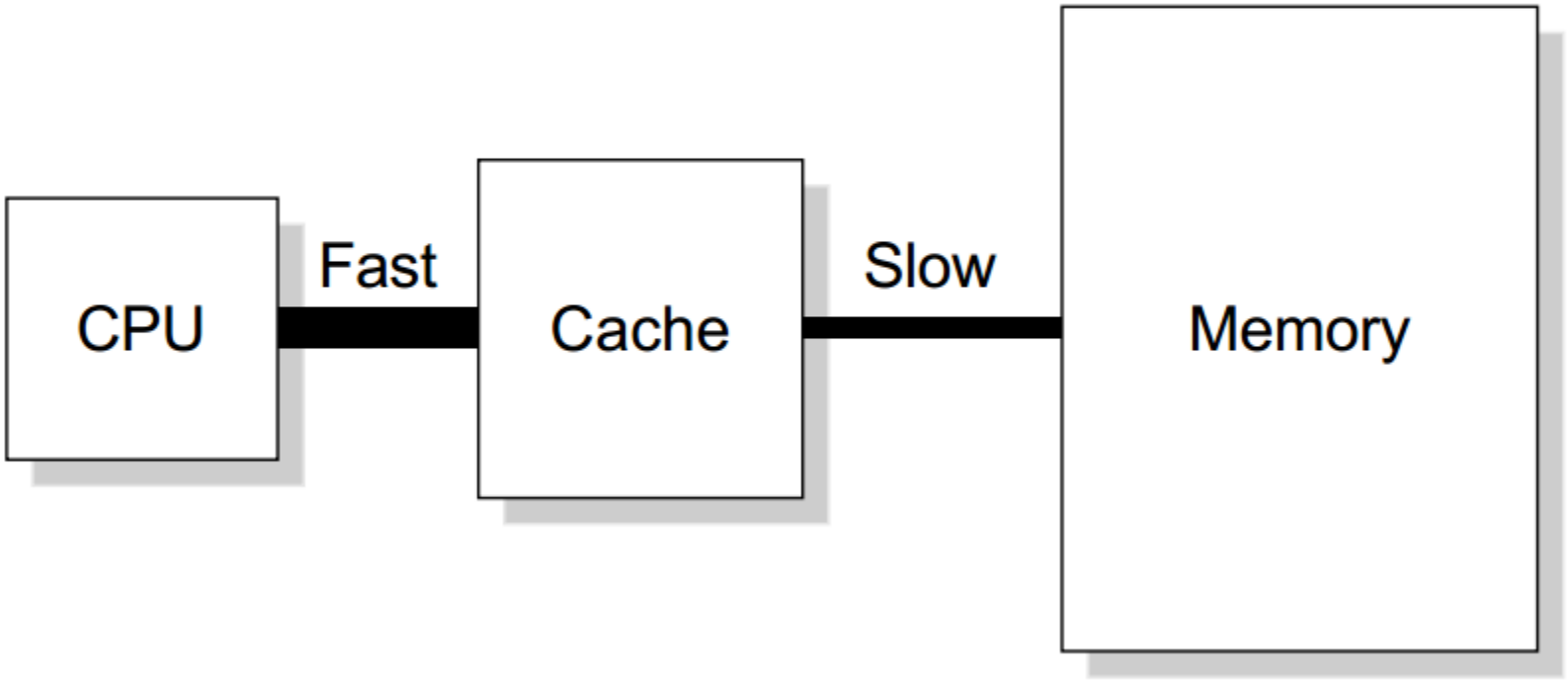
AGENDA

- typen geheugen
- **caching**
- harddisk
- compiler, interpreter en VM

CPU CACHE

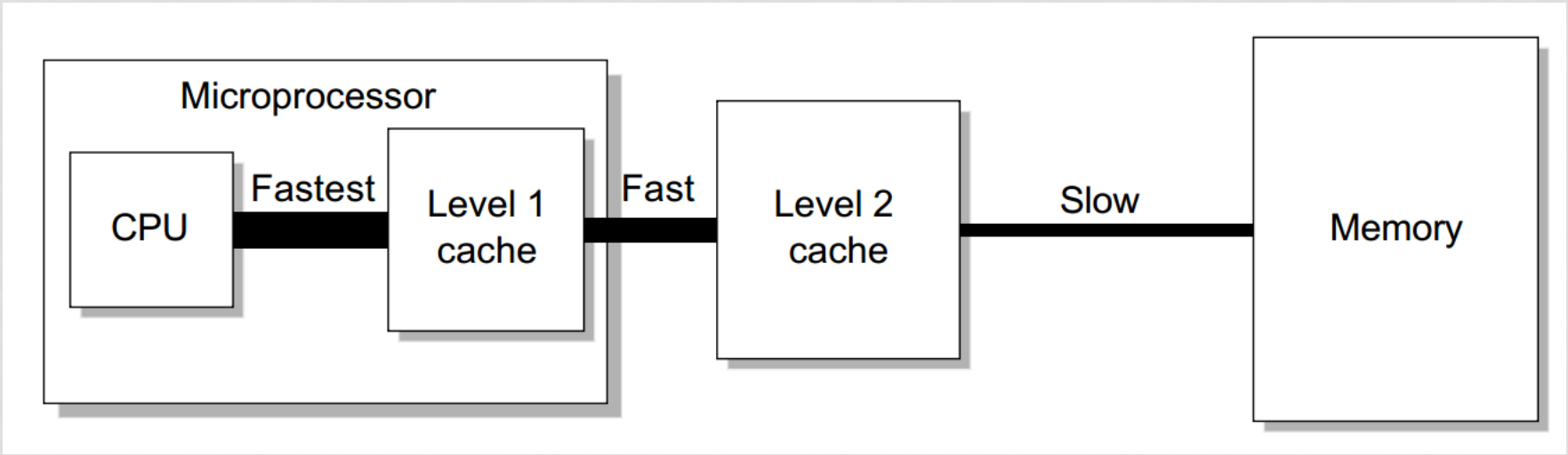
- CPU veel sneller dan RAM
- om te voorkomen dat applicatie moet wachten : cache
- data en instructie cache
  - load : data RAM => cache => CPU
  - store : CPU => cache => RAM
- cache hit ratio : kans dat wat CPU zoekt (ook) in de cache staat

CPU CACHE



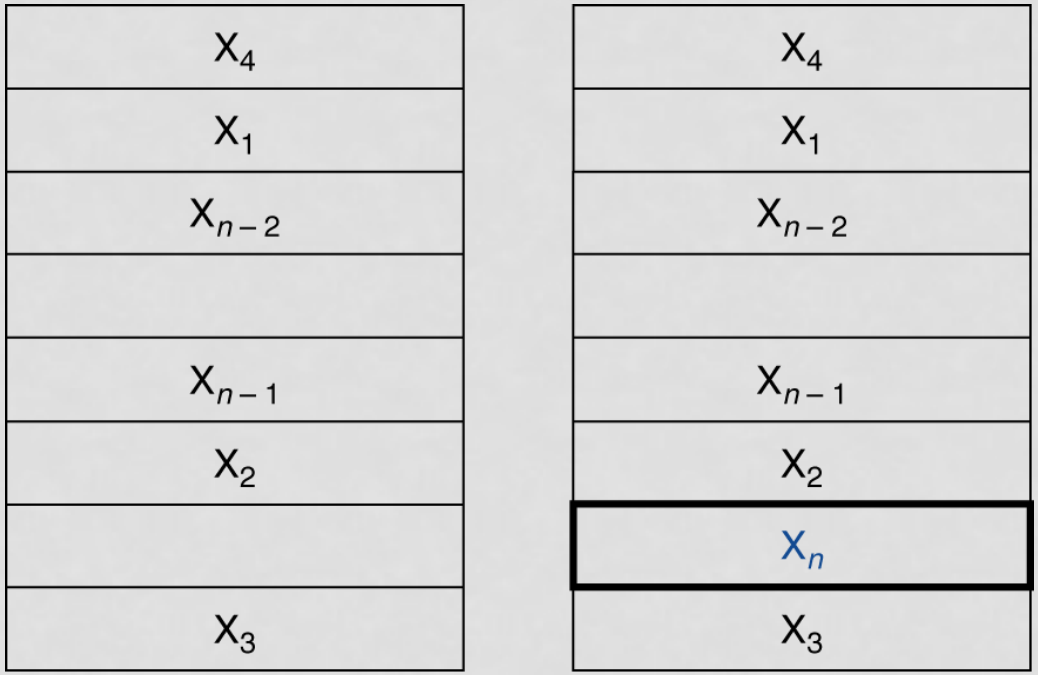
CACHE LEVEL

- on- en off-chip cache
- L2 verder van CPU dan L1



CACHE MEMORY

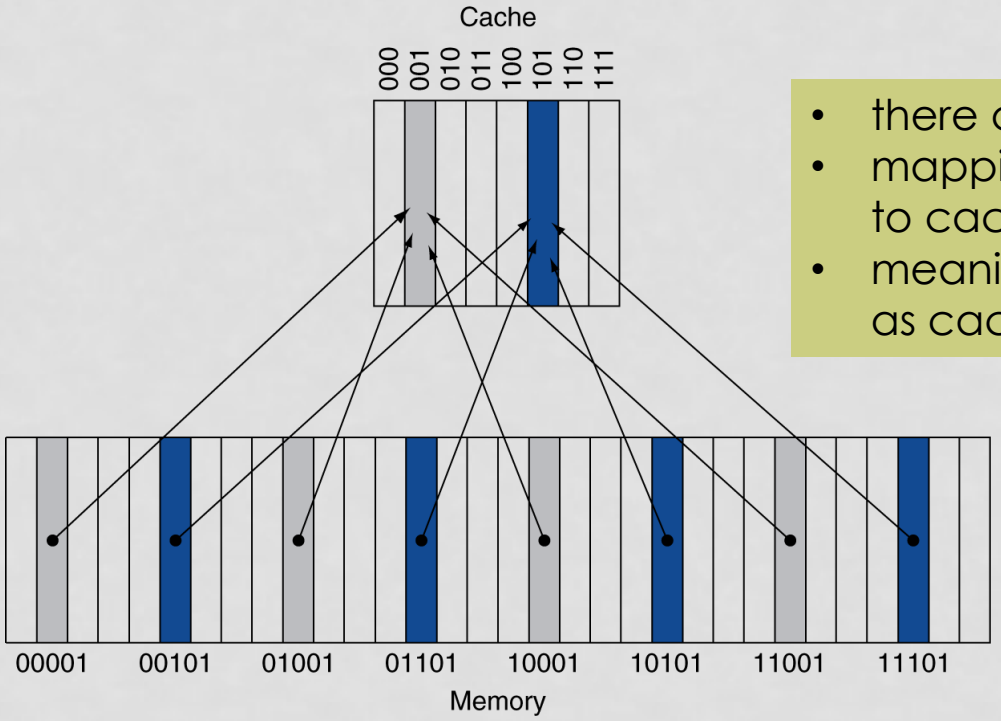
- how do we know if the data is present?
- where do we look?



a. Before the reference to X_n b. After the reference to X_n

DIRECT MAPPED CACHE

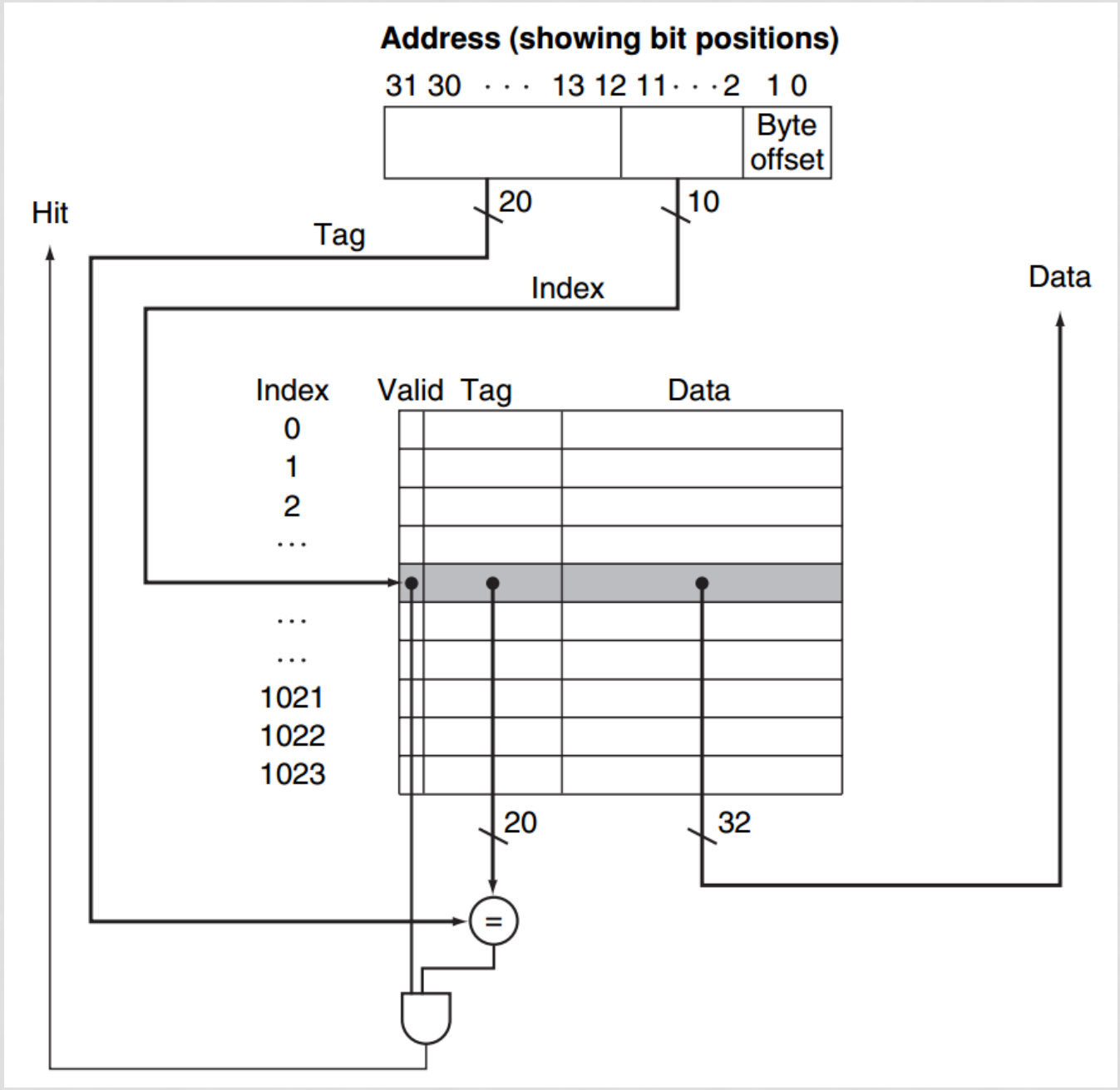
- memory location determines cache index
  - (memory address) modulo (cache size)
- direct mapped: only one choice



- there are 2^3 words in the cache
- mapping : memory address x maps to cache index (x % 8)
- meaning : the lower 3 bits are used as cache index

TAGS, VALID BITS AND OFFSET

- how do we know which particular block is stored in a cache location?
  - store block address as well as the data
  - actually, only need the high-order bits
  - the are called 'the tag'
- how do we know if data in cache is valid ?
  - valid bit: 1 = present, 0 = not present
  - initially 0
- index can be split into index and offset
  - then index refers to a block, and offset refers to a entry in that block


Address (showing bit positions)
31 30 · · · 13 12 11 · · · 2 1 0
Byte offset
20
10
Hit
Tag
Index
Data
Index
Valid
Tag
Data
0
1
2
· · ·
· · ·
· · ·
1021
1022
1023
20
32
=

AVERAGE ACCESS TIME

- hit time is also important for performance
- Average Memory Access Time (AMAT)
  - AMAT = hit rate * hit time + miss rate × miss penalty
- example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 0,95 * 1 + 0,05 × 20 = 1,95 ns

AGENDA

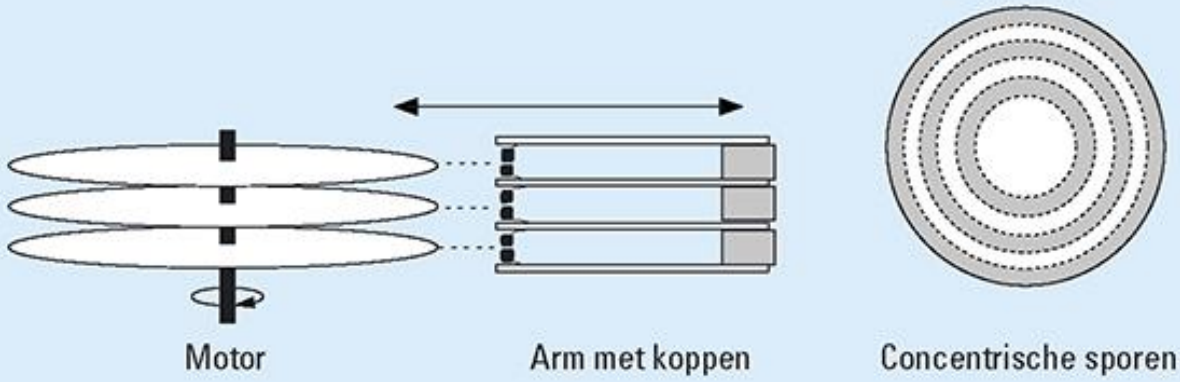
- typen geheugen
- caching
- **harddisk**
- compiler, interpreter en VM

HARDDISK

- 1-8 schijven
- elke schijf heeft 2 koppen
- koppen gemonteerd op arm
- luchtkussen tussen kop en schijf < 1 µm
- tracks zijn concentrisch
- blokken heten sectoren gescheiden door IBG's
  - sector = 512 bytes
- access tijd :
  - seek : arm in juiste positie (2..10ms)
  - search : juiste sector (2..5ms)

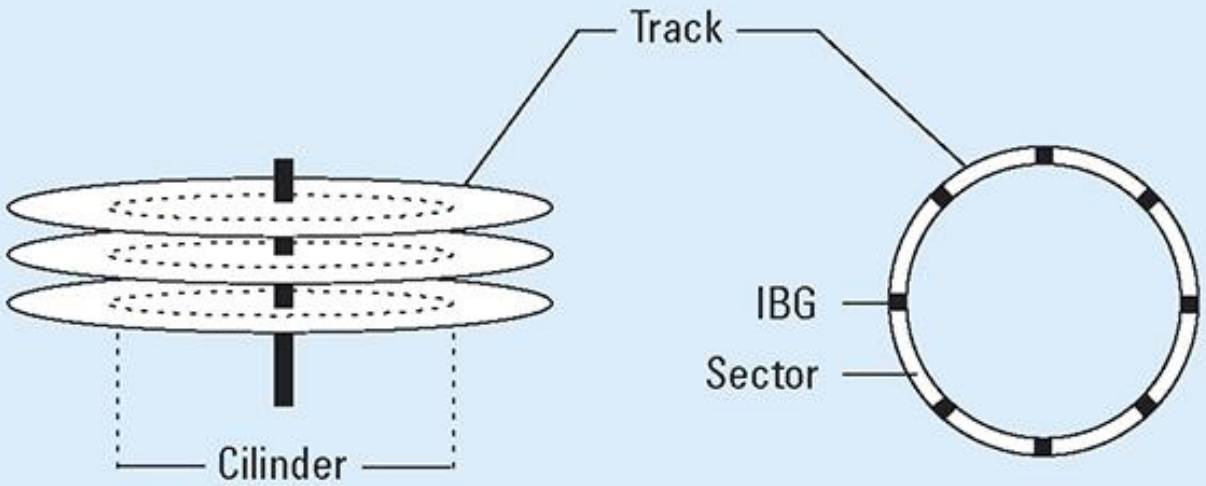
HARDDISK

Figuur 4.7 **Diskdrive**

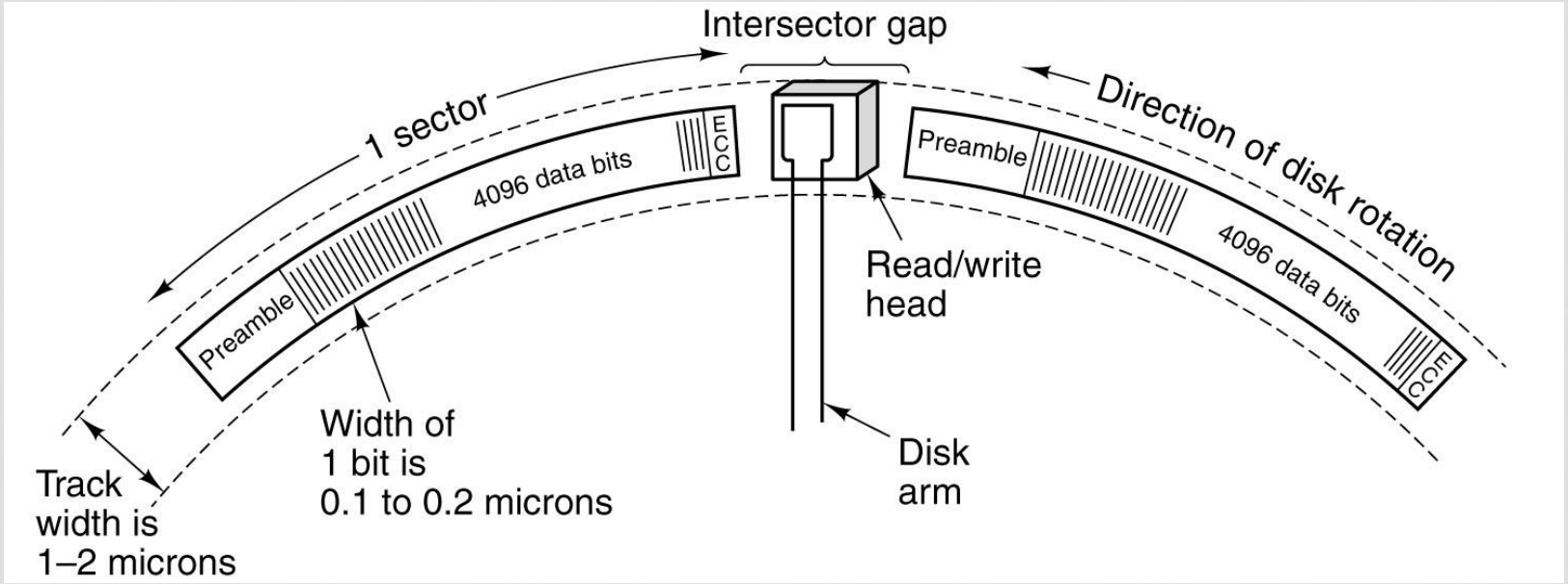


HARDDISK

Figuur 4.8 **Cilinder, track en sector**



HARDDISK



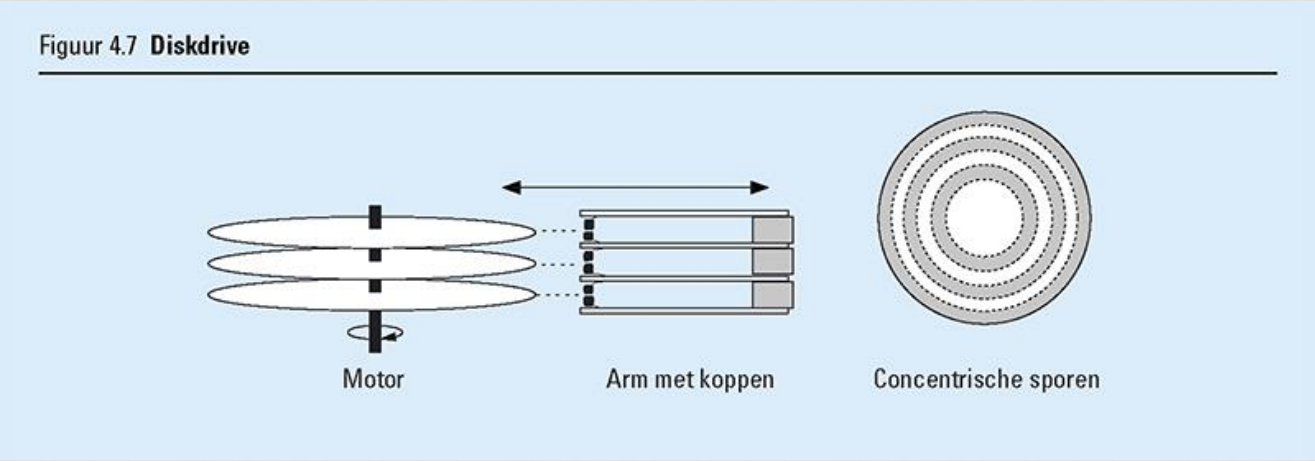
A portion of a disk track. Two sectors are illustrated.

SEEK, READ & WRITE

- time required to read a block into memory :
  - seek time
  - rotational latency
  - transfer time
- seek time
  - usually lower than 10ms
  - so in theory about 100 seeks a second
- transfer time
  - a disk can delivers at least 10–20MB/s throughput
  - you can read in parallel from multiple disks

OPGAVE

- gegeven :
  - 7200 rpm
  - 3,5 " disk
- hoe hoog is de snelheid van de rand (in km/u) ?



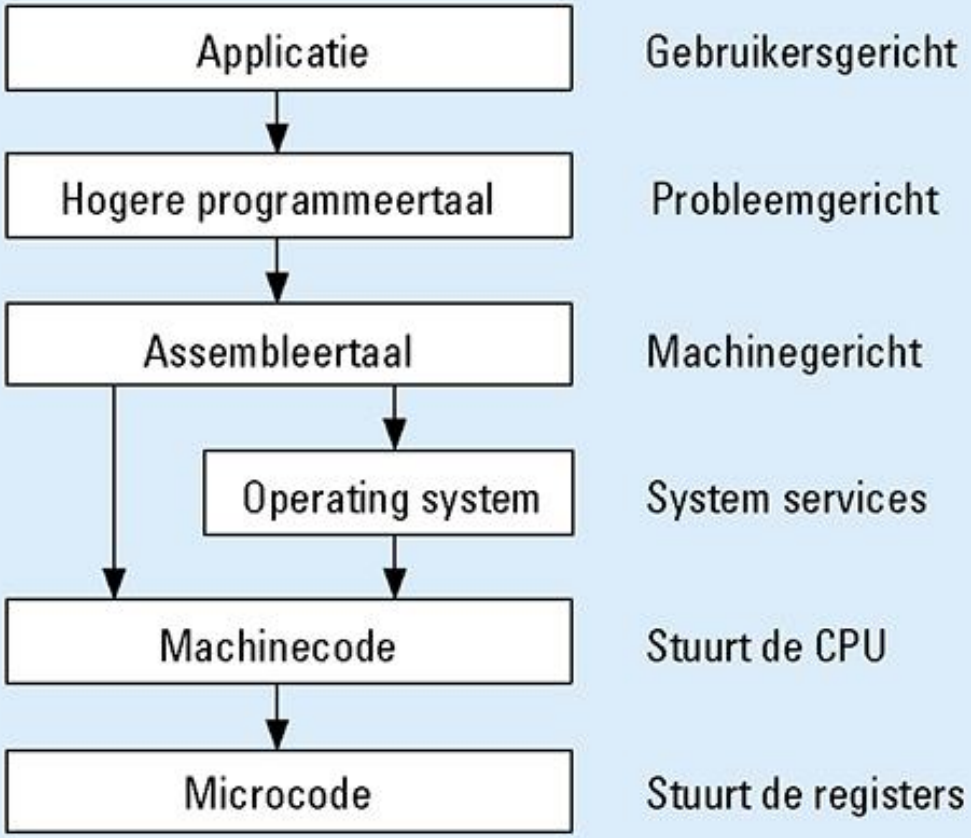
ANTWOORD

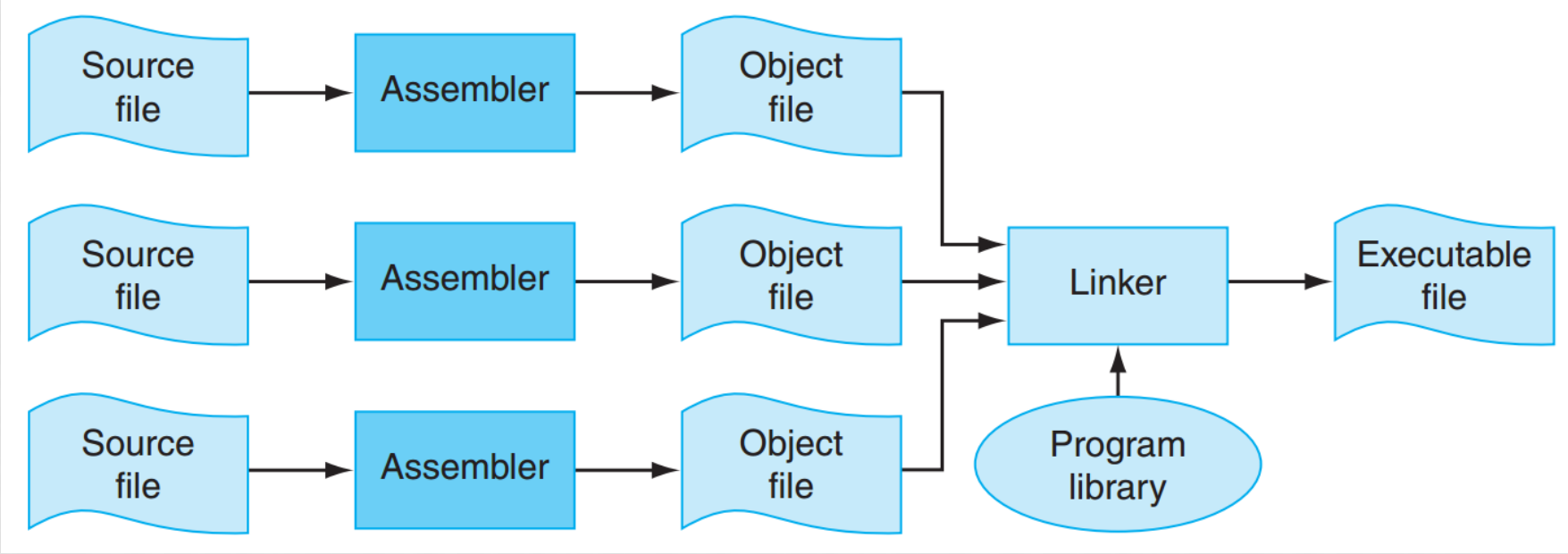
- o = pi * 3,5 * 2,54 [cm]
- v1 = 7200 * o [cm/min]
- v2 = 60 * v1 * 10E-5 [km/u] = 121 km/u

AGENDA

- typen geheugen
- caching
- harddisk
- **compiler, interpreter en VM**

Figuur 6.1 **Taalniveaus**




Source file
Assembler
Object file
Source file
Assembler
Object file
Linker
Executable file
Source file
Assembler
Object file
Program library

LOADING PROGRAMS

- program must be brought (from disk) into memory and 'placed within a process' for it to be run
- compiler is (usually) not aware physical addresses
- linker or loader must relocate object modules by translating symbolic addresses to physical addresses

A SIDE-BY-SIDE COMPARISON OF COMPILED LANGUAGES AND INTERPRETED LANGUAGES

Upwork™

A look at how compilers and interpreters work, and how their differences affect memory, runtime speed, and computer workload.

	A COMPILER	AN INTERPRETER
Input	... takes an entire program as its input.	... takes a single line of code, or instruction, as its input.
Output	... generates intermediate object code.	... does not generate any intermediate object code.
Speed	... executes faster.	... executes slower.
Memory	... requires more memory in order to create object code.	... requires less memory (doesn't create object code).
Workload	... doesn't need to compile every single time, just once.	... has to convert high-level languages to low-level programs at execution.
Errors	... displays errors once the entire program is checked.	... displays errors when each instruction is run.

COMPILER

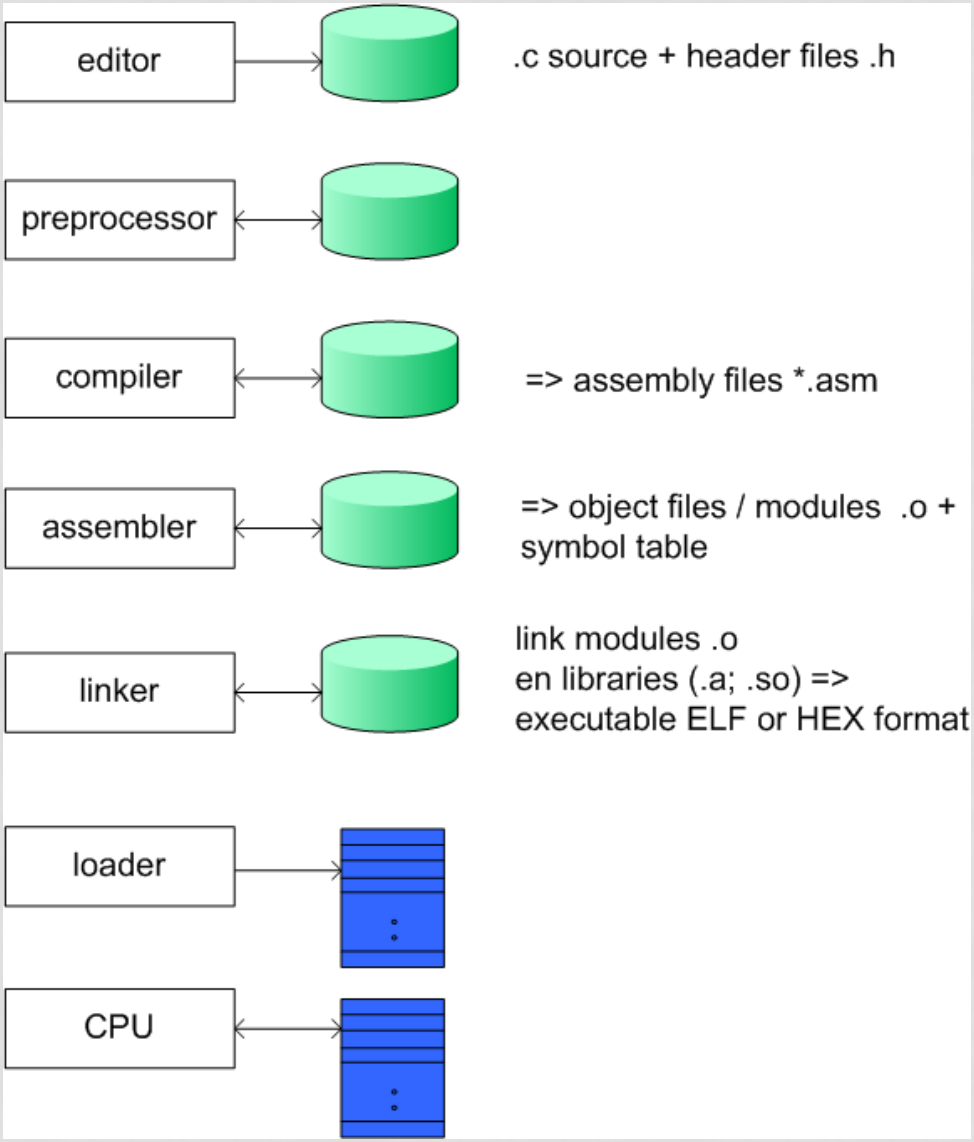
- examples : assembly, C, C++
- translates source code to machine code
- does not execute the source
- translation at compile time takes a lot of time, but execution at run time is fast
- does some memory management e.g. register allocation and code optimization

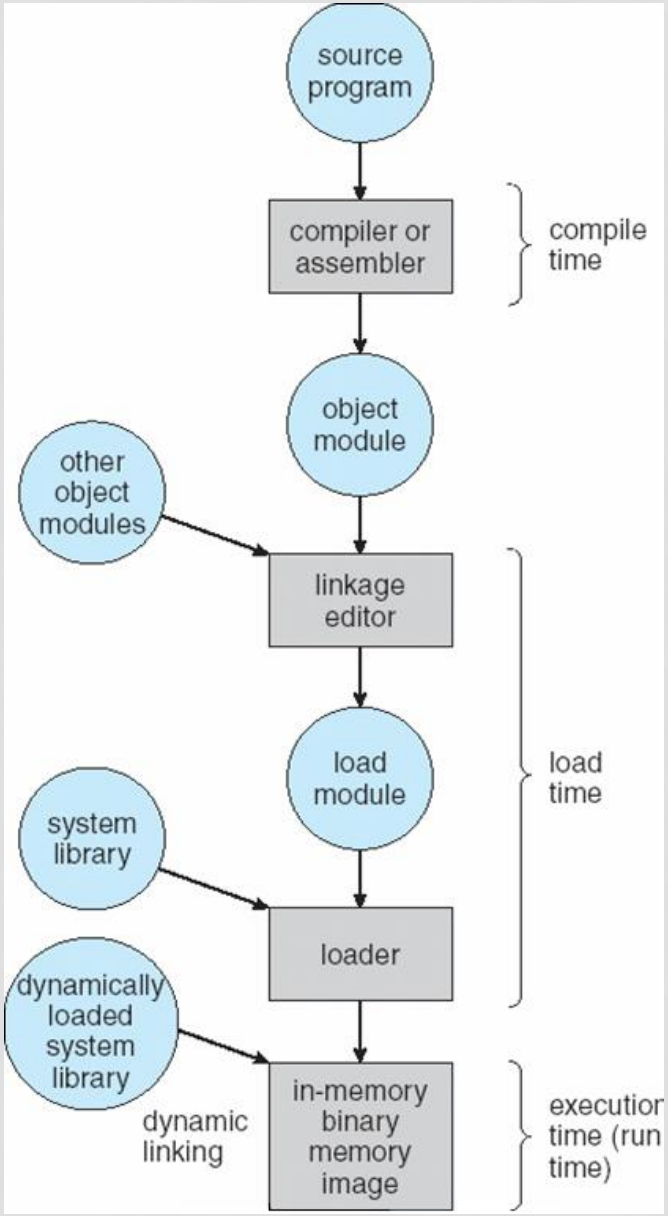
INTERPRETER

- examples : PHP, Perl, Python, Ruby, shell or other scripting languages
- interprets (=executes) one line at a time from the source file
- translate source code into some efficient intermediate representation and immediately execute this
- interpreting a program is much slower than executing native machine code
  - interpreting a high-level language is ~100 times slower
  - interpreter must analyze each statement in the program each time it is executed and then perform the desired action - slow!
- development can be quicker
  - edit-interpret-debug vs. edit-compile-run-debug

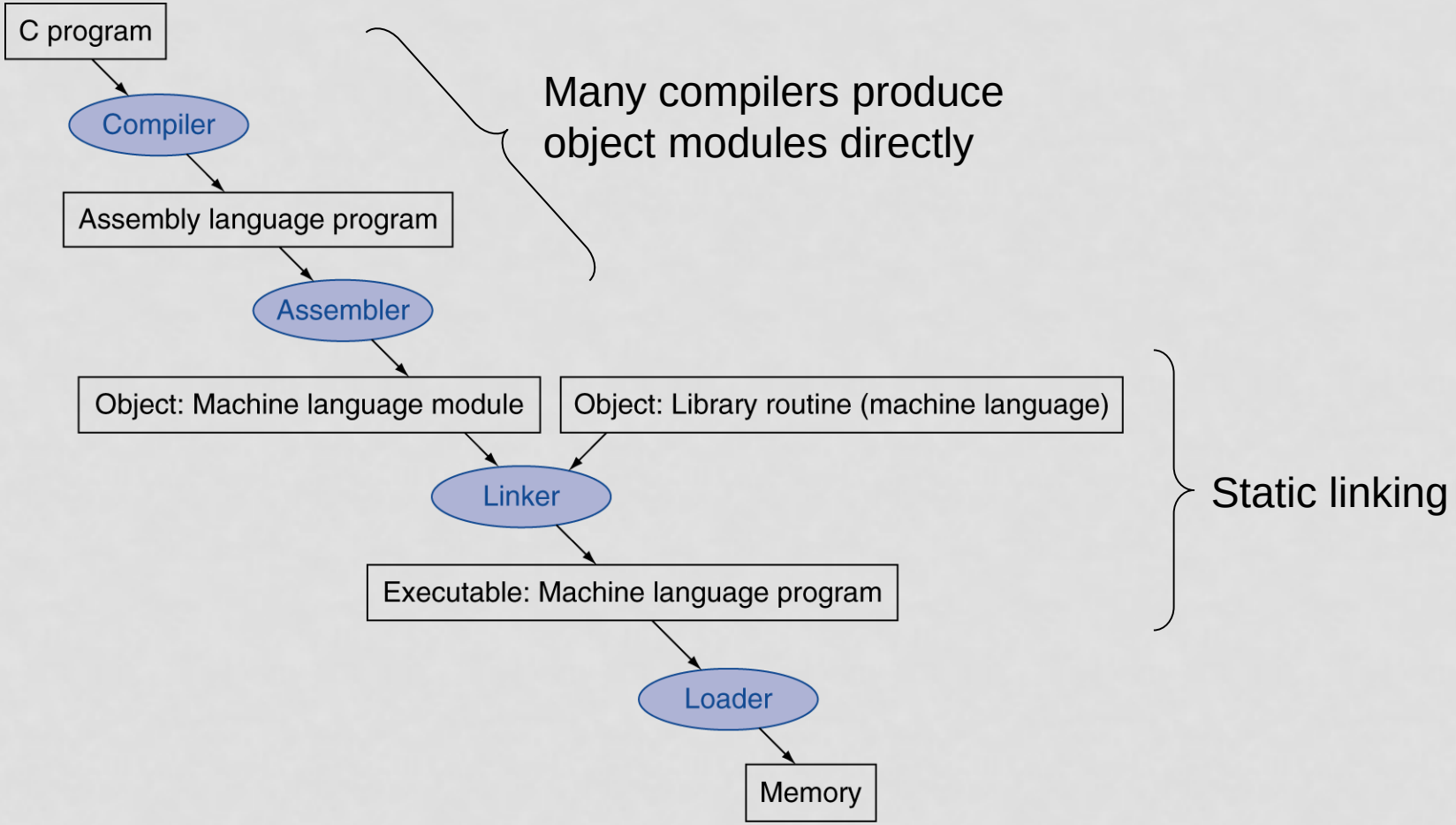
VM'S

- system virtual machine : provides a platform which supports execution of an OS
  - emulates existing architecture
  - multiple instances of VMs to provide more efficient use of computing resources
- language virtual machine : to run a single program (a singe process)
  - a software CPIU with an instruction set (e.g. Scala runs on JVM)
  - to run platform independent intermediate languages (JVM, CLR, byte code)
  - may include memory management, object creation/cleanup, threading, security etc.
  - this byte code is interpreted or compiled to native machine code
    - JIT : parts of code that are will be run multiple times


editor
.c source + header files .h
preprocessor
compiler
=> assembly files *.asm
assembler
=> object files / modules .o +
symbol table
linker
link modules .o
en libraries (.a; .so) =>
executable ELF or HEX format
loader
CPU


source program
compiler or assembler
compile time
object module
other object modules
linkage editor
load module
load time
system library
loader
dynamically loaded system library
dynamic linking
in-memory binary memory image
execution time (run time)

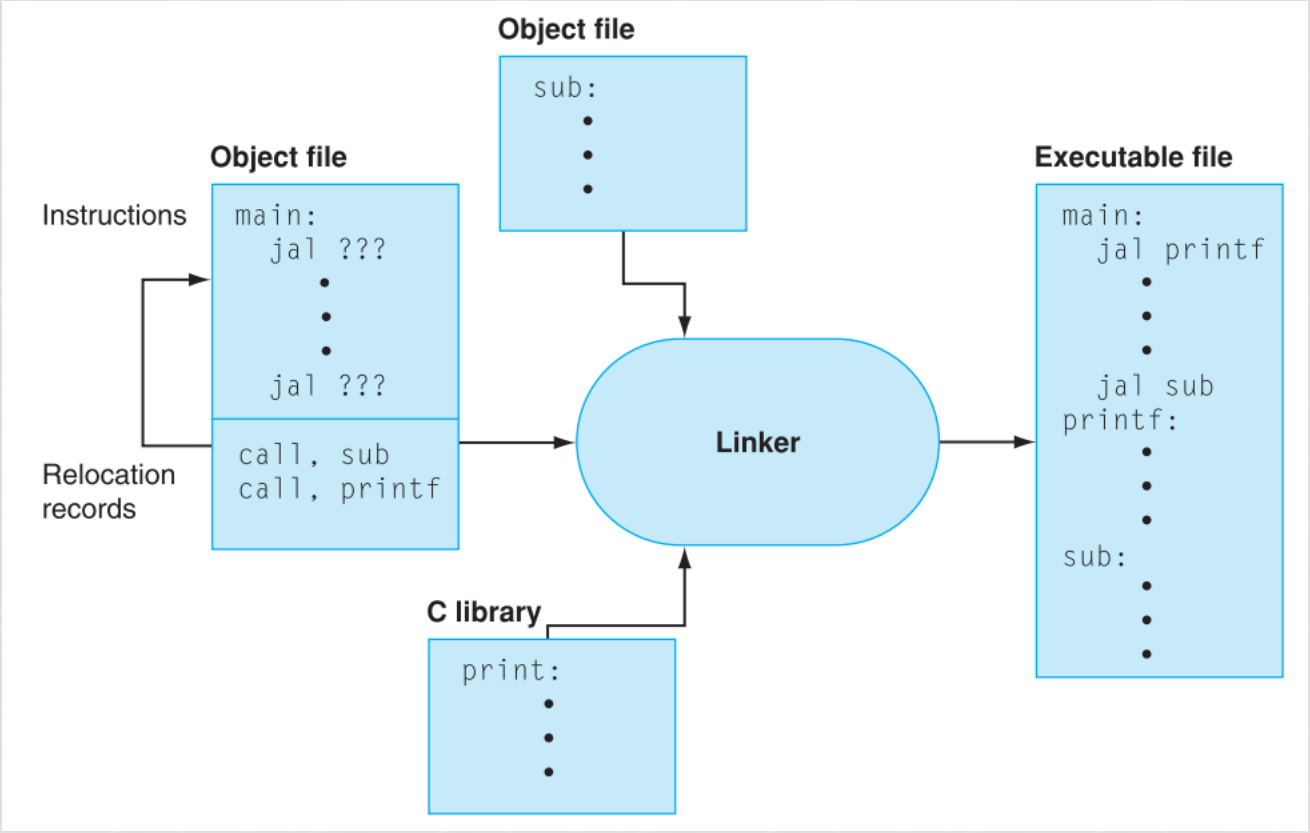
TRANSLATION AND STARTUP



PRODUCING AN OBJECT MODULE

- assembler (or compiler) translates program into machine instructions
- provides information for building a complete program from the pieces
  - header: described contents of object module
  - text segment: translated instructions
  - static data segment: data allocated for the life of the program
  - relocation info: for contents that depend on absolute location of loaded program
  - symbol table: global definitions and external refs
  - debug info: for associating with source code

LINKER


Object file
sub:
Object file
Instructions
main:
jal ???
jal ???
Relocation records
call, sub
call, printf
Linker
Executable file
main:
jal printf
jal sub
printf:
sub:
C library
print:

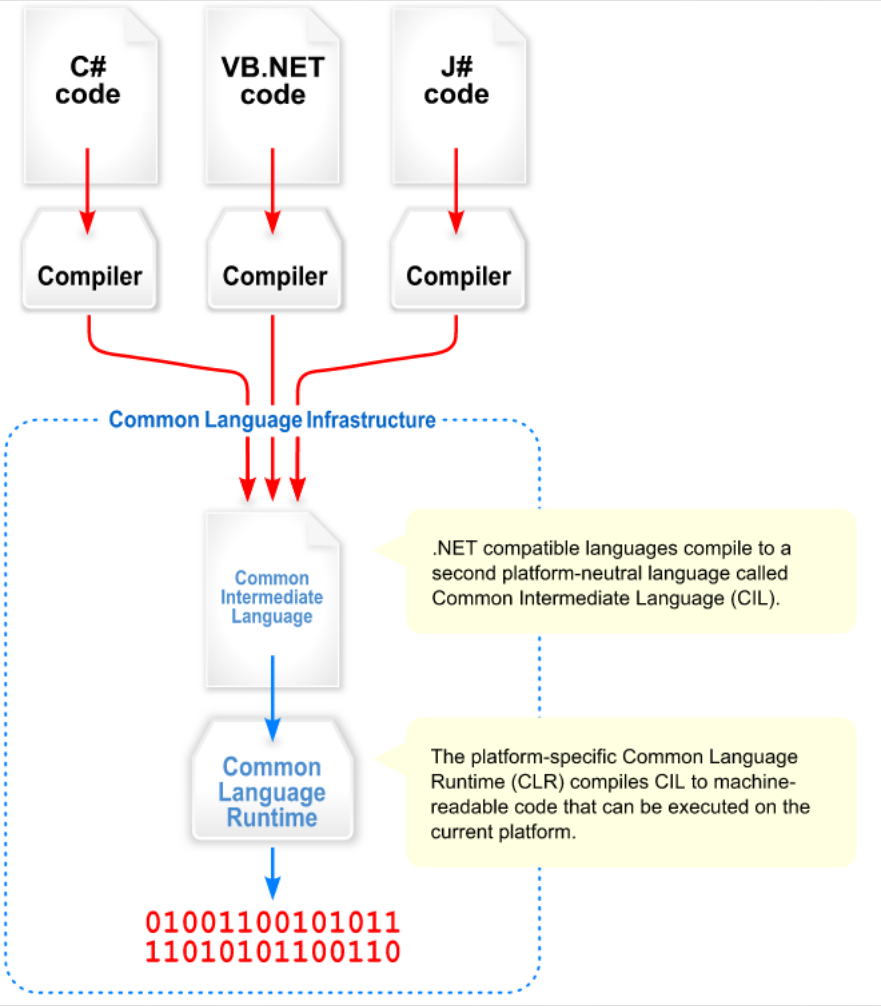
LINKING OBJECT MODULES

- produces an executable image
  1. merges segments
  2. resolve labels (determine their addresses)
  3. patch location-dependent and external refs
- could leave location dependencies for fixing by a relocating loader
  - but with virtual memory, no need to do this
  - program can be loaded into absolute location in virtual memory space

STATIC AND DYNAMIC LIBRARIES

- dynamisch
  - shared object .so (default)
  - geen duplicatie van code in executable, benodigde code word tijdens runtime geladen in geheugen
  - kleine executable, minder geheugen en schijfruimte nodig
  - upgraden libraries zonder re-compilen (mits API niet verandert)
- statisch
  - archives .a
  - duplicatie van code in executable tijdens linken
  - geen afhankelijkheden met libraries

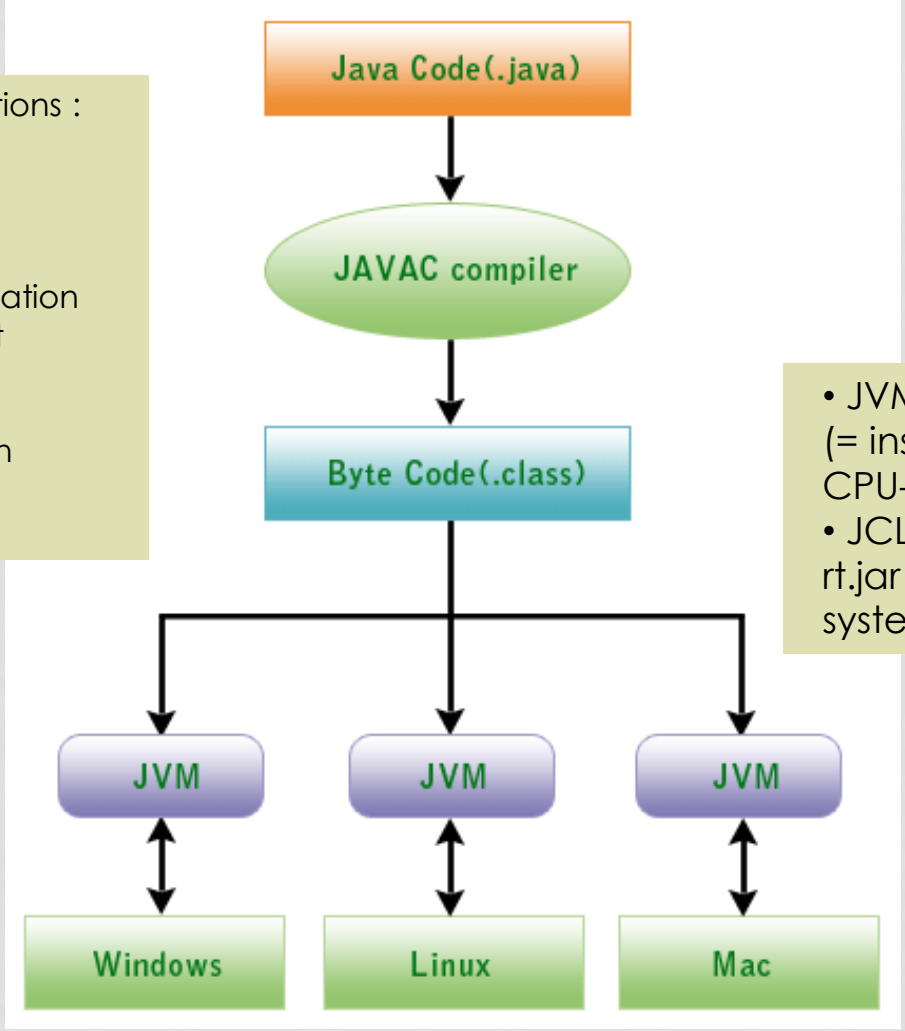
EXAMPLE .NET CIL AND CLR



CPU EMULATIE : JVM

The JVM has following instructions :

Load and store
Arithmetic
Type conversion
Object creation and manipulation
Operand stack management (push/pop)
Control transfer (branching)
Method invocation and return
Throwing exceptions
Monitor-based concurrency



- JVM vertaalt byte codes (= instructies) naar (x86) CPU- instructies
- JCL (Java Class Library) rt.jar afbeelden op OS system calls

STARTING JAVA APPLICATIONS