# 64-040 Modul IP7: Rechnerstrukturen

http://tams.informatik.uni-hamburg.de/lectures/2011ws/vorlesung/rs

## Kapitel 21

Andreas Mäder

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

**Technische Aspekte Multimodaler Systeme**

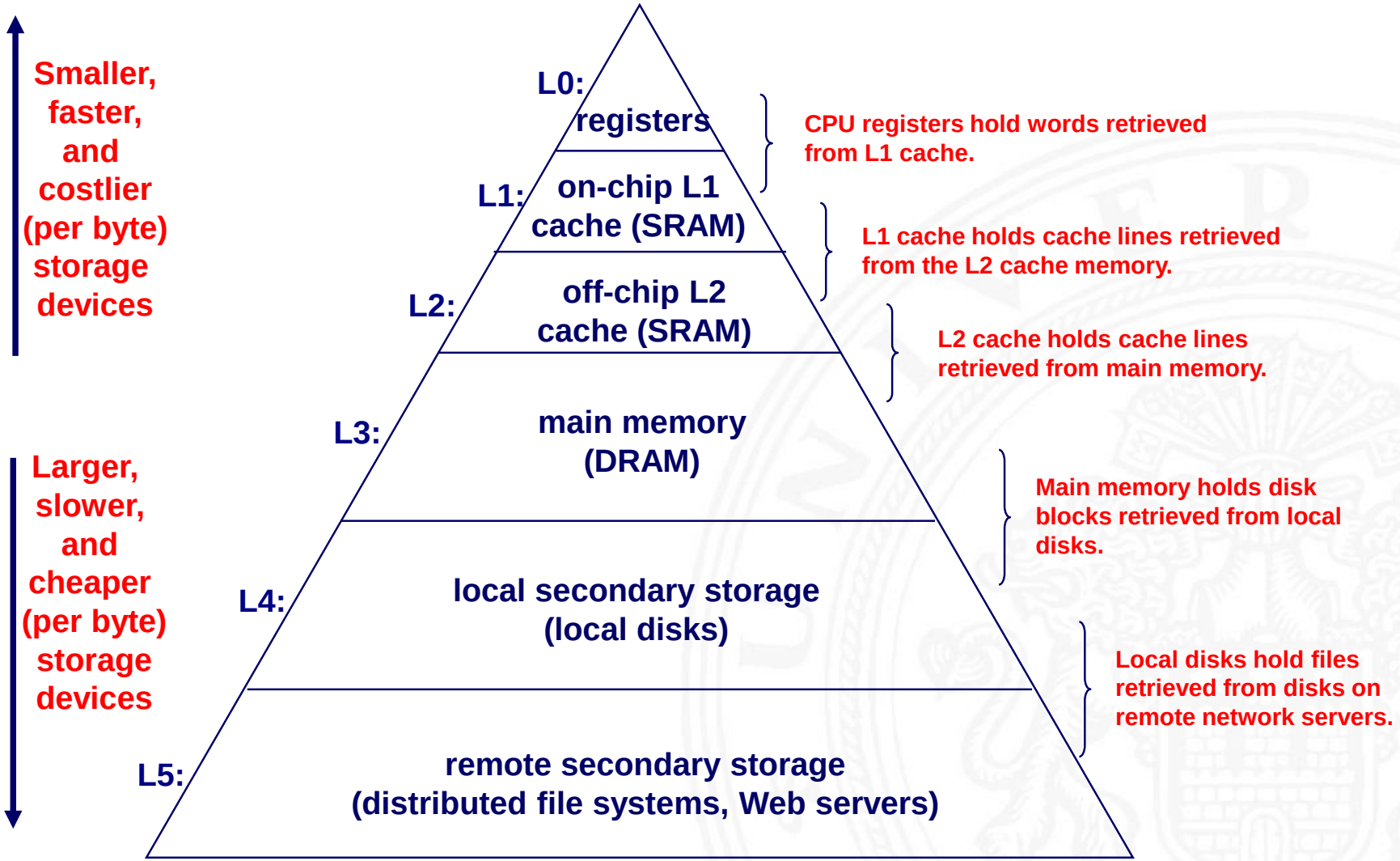
Wintersemester 2011/2012

# Kapitel 21

## Speicherhierarchie

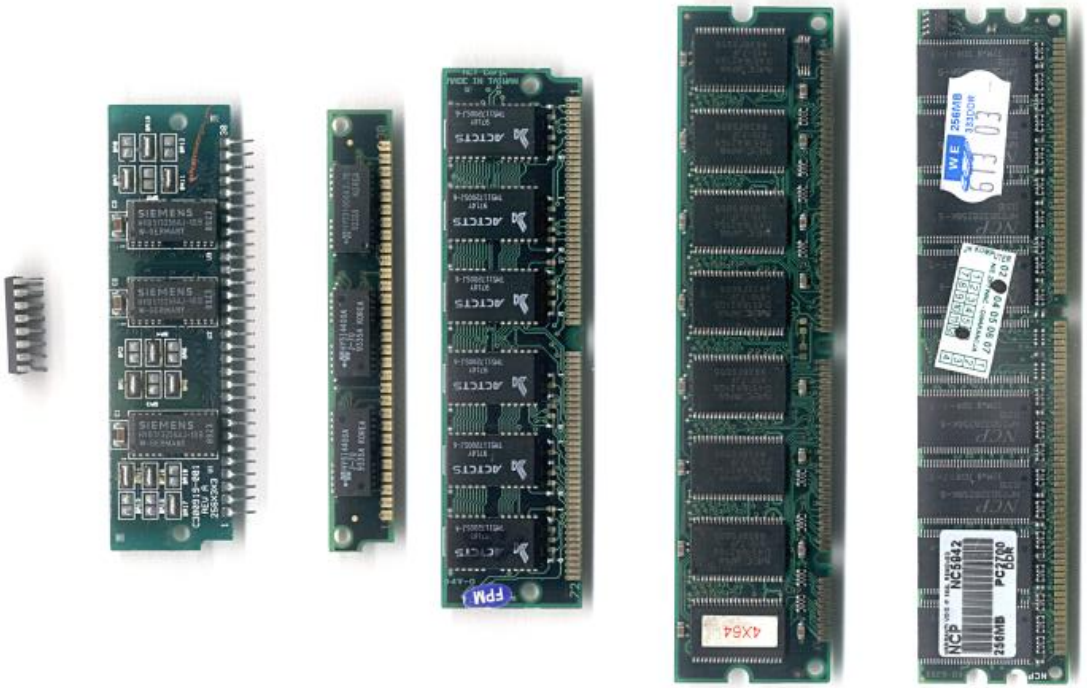
- Speichertypen
  - Halbleiterspeicher
  - Festplatten
  - spezifische Eigenschaften
- Motivation
- Cache Speicher
- Virtueller Speicher
  - Beispiel: Pentium und Linux

# Speicherhierarchie: Konzept



# Random-Access Memory / RAM

- RAM ist als Chip gepackt
- Grundspeichereinheit ist eine Zelle (ein Bit pro Zelle)
- Viele RAM Chips bilden einen Speicher



# Random-Access Memory / RAM (cont.)

Statischer RAM (SRAM)

- jede Zelle speichert Bits mit einer 6-Transistor Schaltung
- speichert Wert solange er mit Energie versorgt wird
- relativ unanfällig gegen Störungen wie elektrische Brummspannungen
- schneller und teurer als DRAM

Dynamischer RAM (DRAM)

- jede Zelle speichert Bits mit einem Kondensator und einem Transistor
- der Wert muss alle 10-100 ms aufgefrischt werden
- anfällig für Störungen
- langsamer und billiger als SRAM

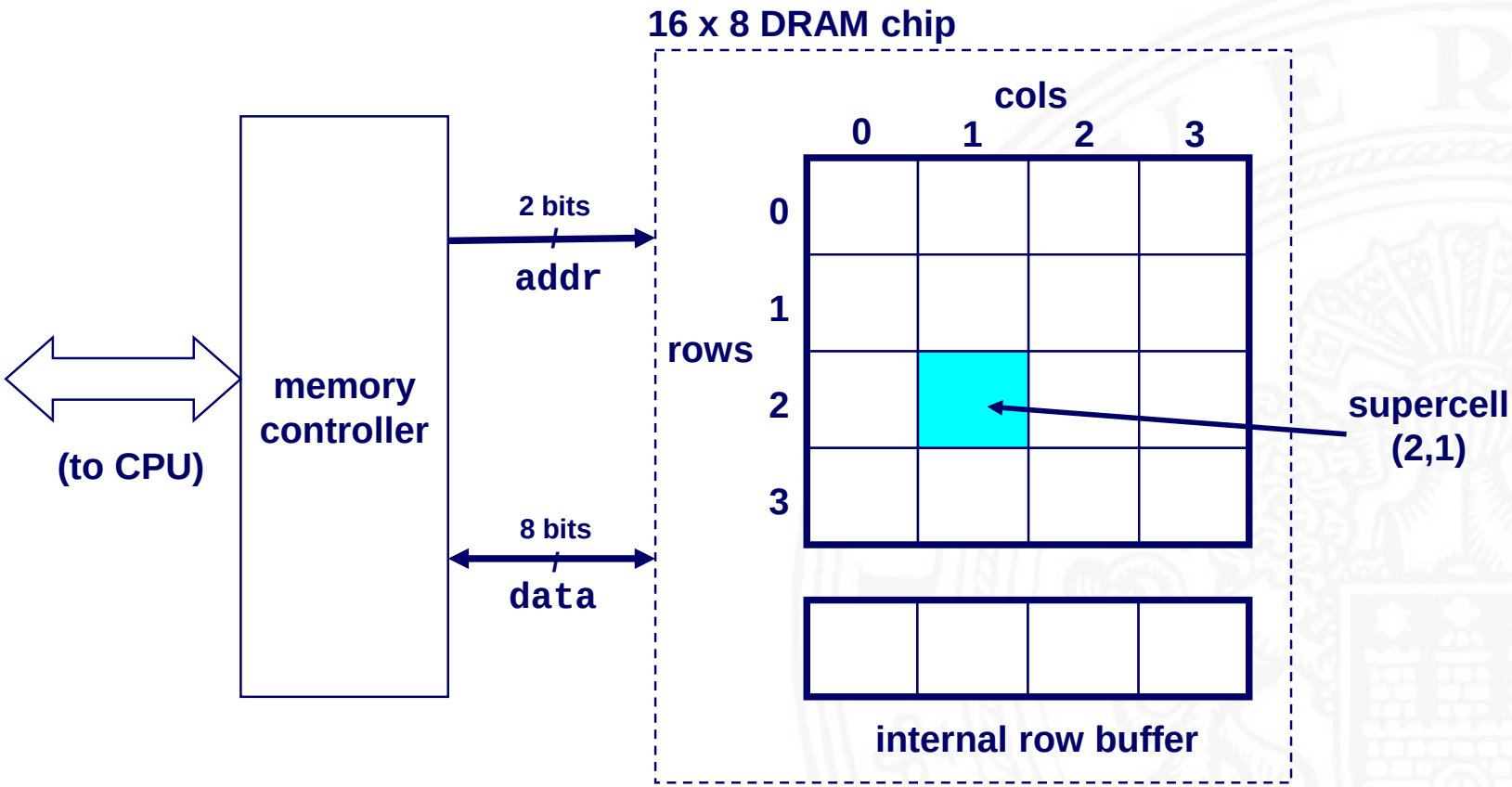
# SRAM vs. DRAM

| | SRAM | DRAM |
|---|---|---|
| Zugriffszeit | 5...50 ns | 60...100 ns $t_{rac}$ |
| | | 20...300 ns $t_{cac}$ |
| | | 110...180 ns $t_{cyc}$ |
| Leistungsaufnahme | 200...1300 mW | 300...600 mW |
| Speicherkapazität | $<$ 72 Mbit | $<$ 4 Gbit |
| Preis | 10 €/Mbit | 0,2 Ct./Mbit |

| | Tran. per bit | Access time | Persist? | Sensitive? | Cost | Applications |
|---|---|---|---|---|---|---|
| SRAM | 6 | 1X | Yes | No | 100x | cache memories |
| DRAM | 1 | 10X | No | Yes | 1X | Main memories, frame buffers |

# DRAM Organisation

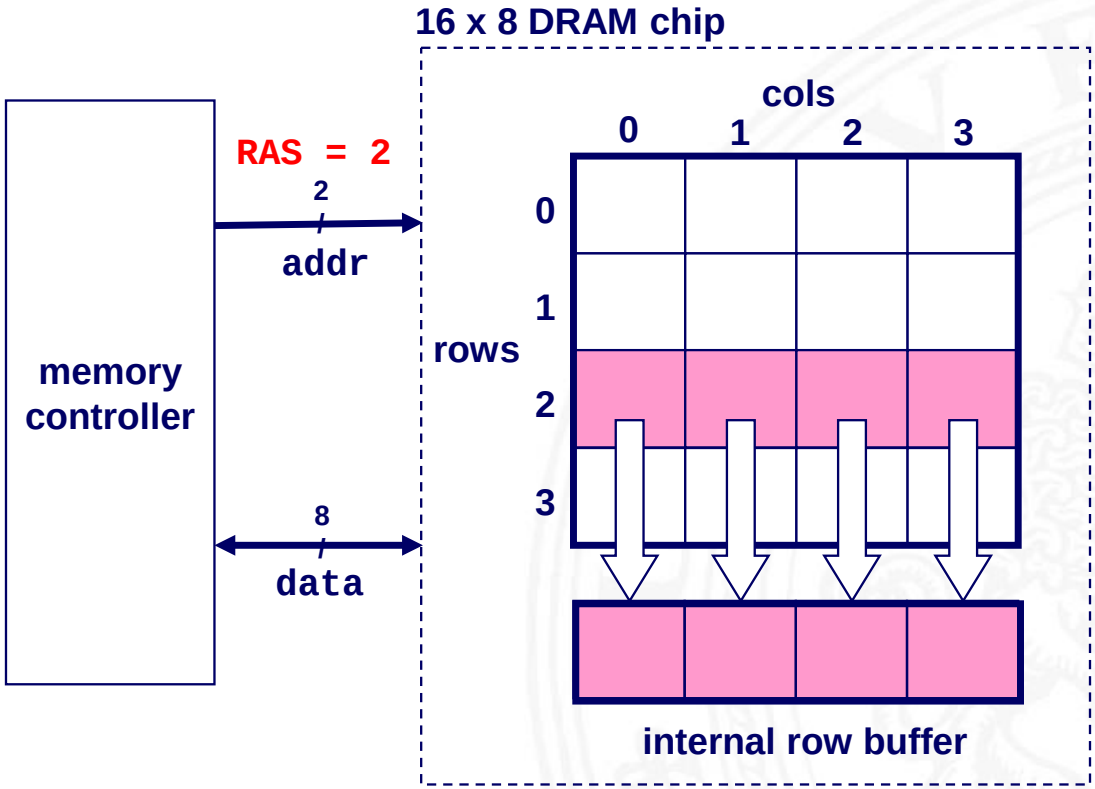
- (d × w) DRAM: organisiert als d-Superzellen mit w-bits



# Lesen der DRAM Zelle (2,1)

1.a „Row Access Strobe“ (RAS) wählt Zeile 2

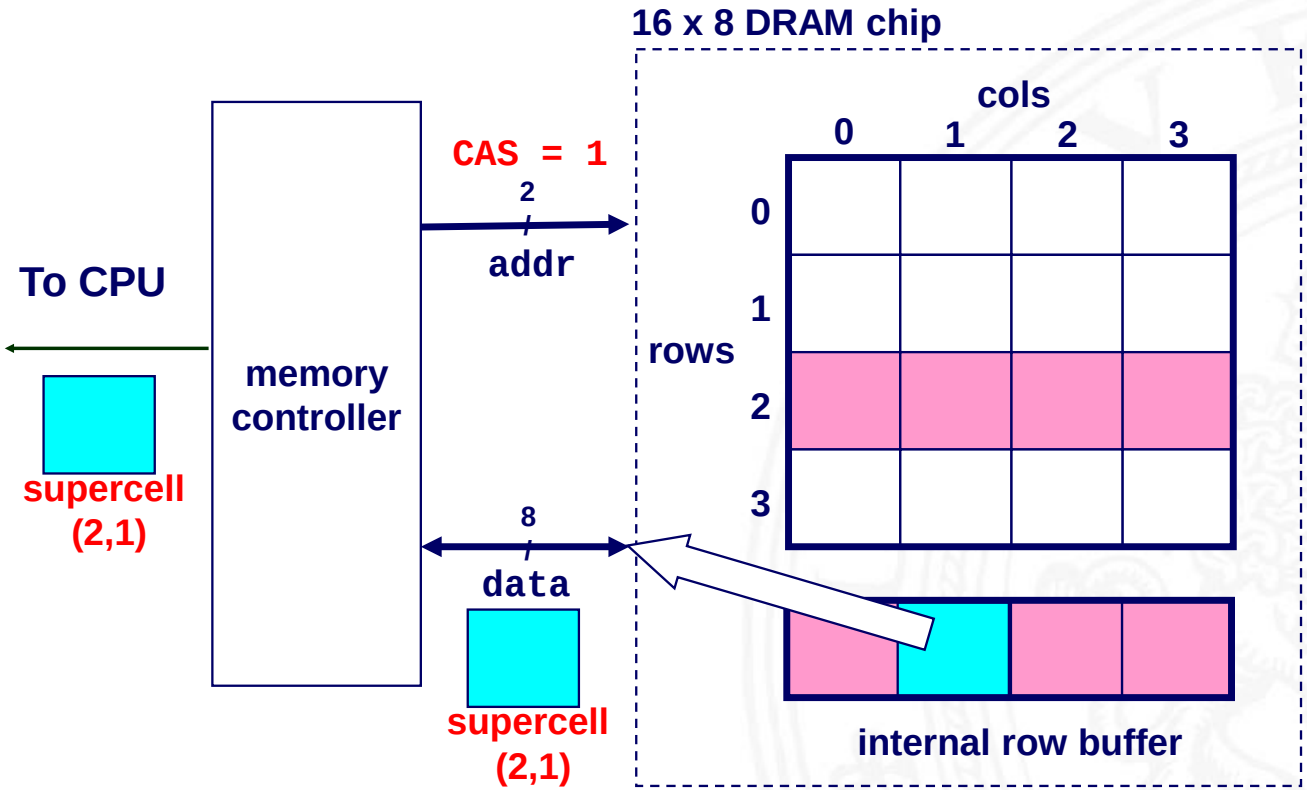
1.b Zeile aus DRAM Array in Zeilenpuffer („Row Buffer“) kopieren



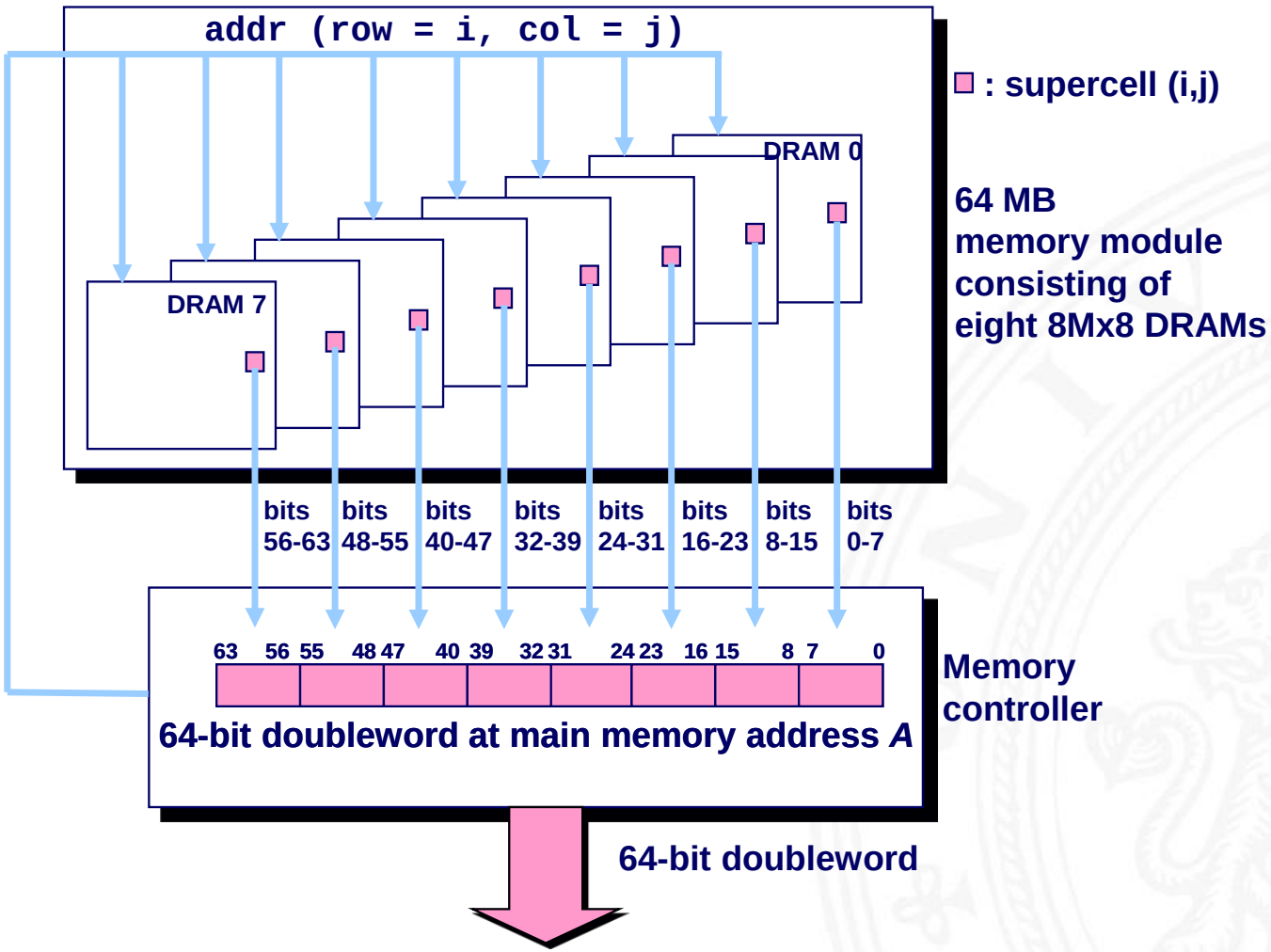
# Lesen der DRAM Zelle (2,1) (cont.)

2.a „Column Access Strobe“ (CAS) wählt Spalte 1

2.b Superzelle (2,1) aus Buffer lesen und auf Datenleitungen legen



# Speichermodule



# Nichtflüchtige Speicher

- DRAM und SRAM sind flüchtige Speicher
  - Informationen gehen beim Abschalten verloren
- nichtflüchtige Speicher speichern Werte selbst wenn sie spannungslos sind
  - allgemeiner Name ist „Read-Only-Memory“ (ROM)
  - irreführend, da einige ROMs auch verändert werden können
- Arten von ROMs
  - PROM: programmierbarer ROM
  - EPROM: „Eraseable Programmable ROM“ löschbar (UV Licht), programmierbar
  - EEPROM: „Electrically Eraseable PROM“ elektrisch löschbarer PROM
  - Flash Speicher

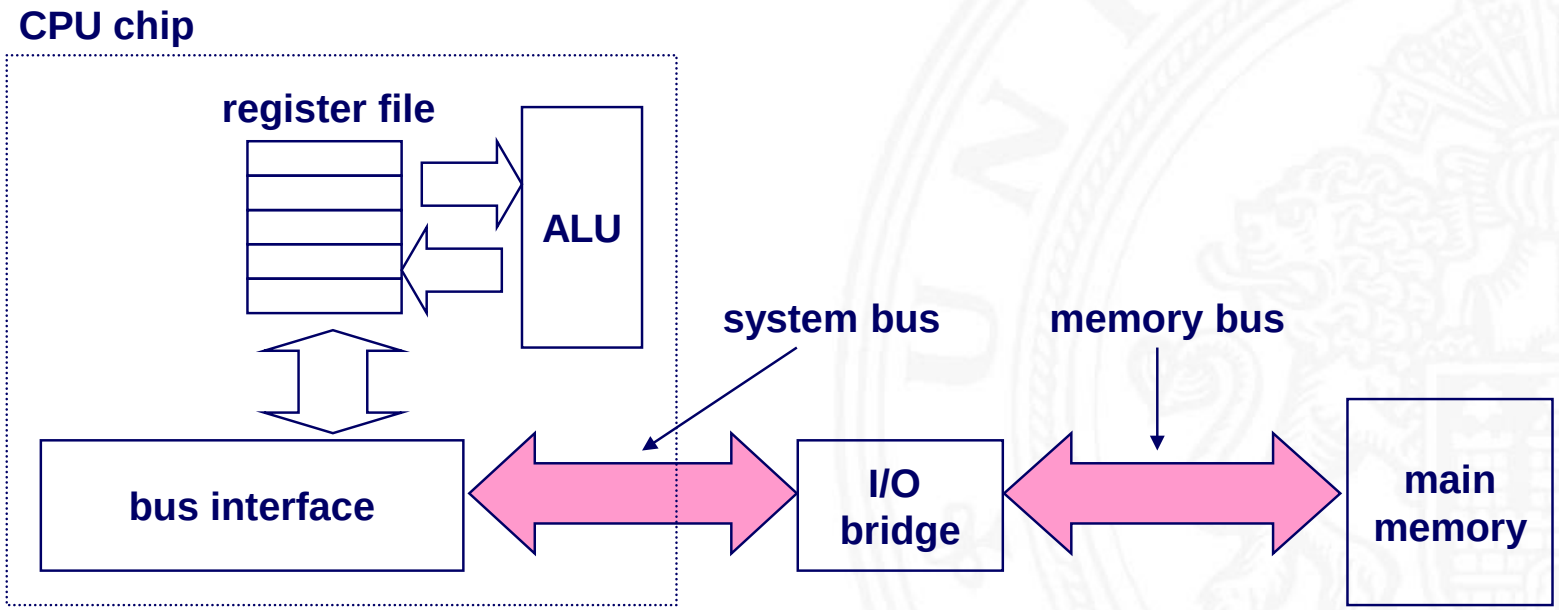
# Nichtflüchtige Speicher (cont.)

Anwendungsbeispiel: nichtflüchtige Speicher

- Firmware
- Programm wird in einem ROM gespeichert
  - Boot Code, BIOS („Basic Input/Output System“)
  - Grafikkarten, Festplattencontroller

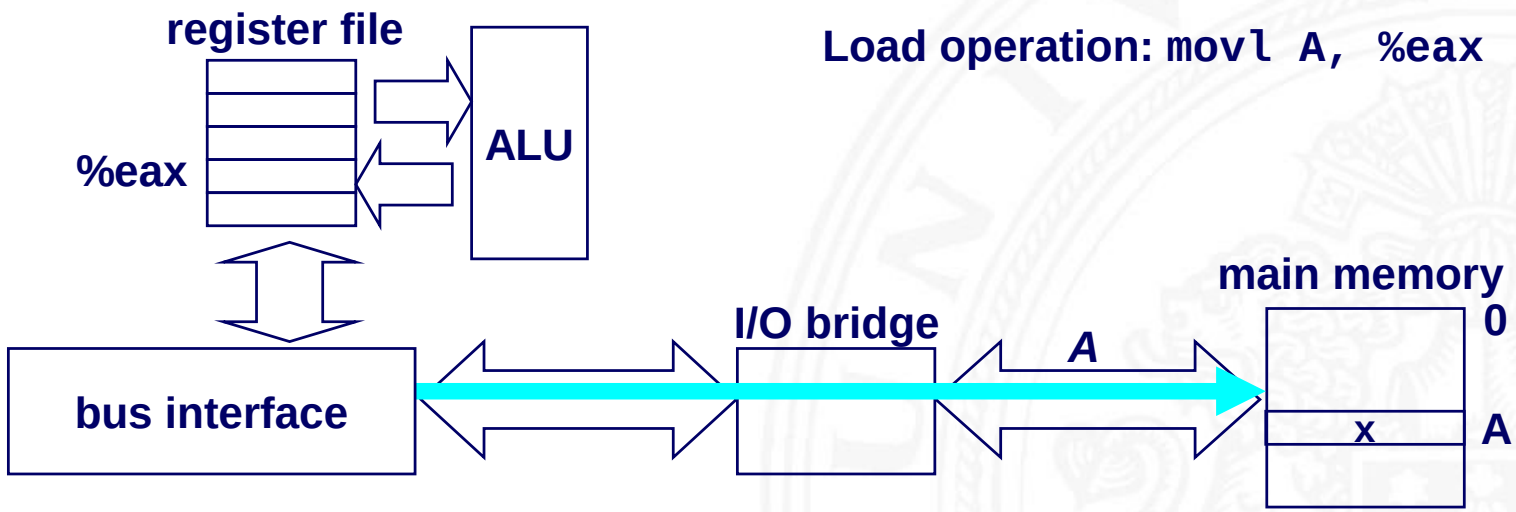
# Bussysteme verbinden CPU und Speicher

- Busse
  - Bündel paralleler Leitungen
  - es gibt mehr als einen Treiber ⇒ Tristate-Treiber
- Busse im Rechner
  - zur Übertragung von Adressen, Daten und Kontrollsignalen
  - werden üblicherweise von mehreren Geräten genutzt



# lesender Speicherzugriff

1. CPU legt Adresse A auf den Speicherbus

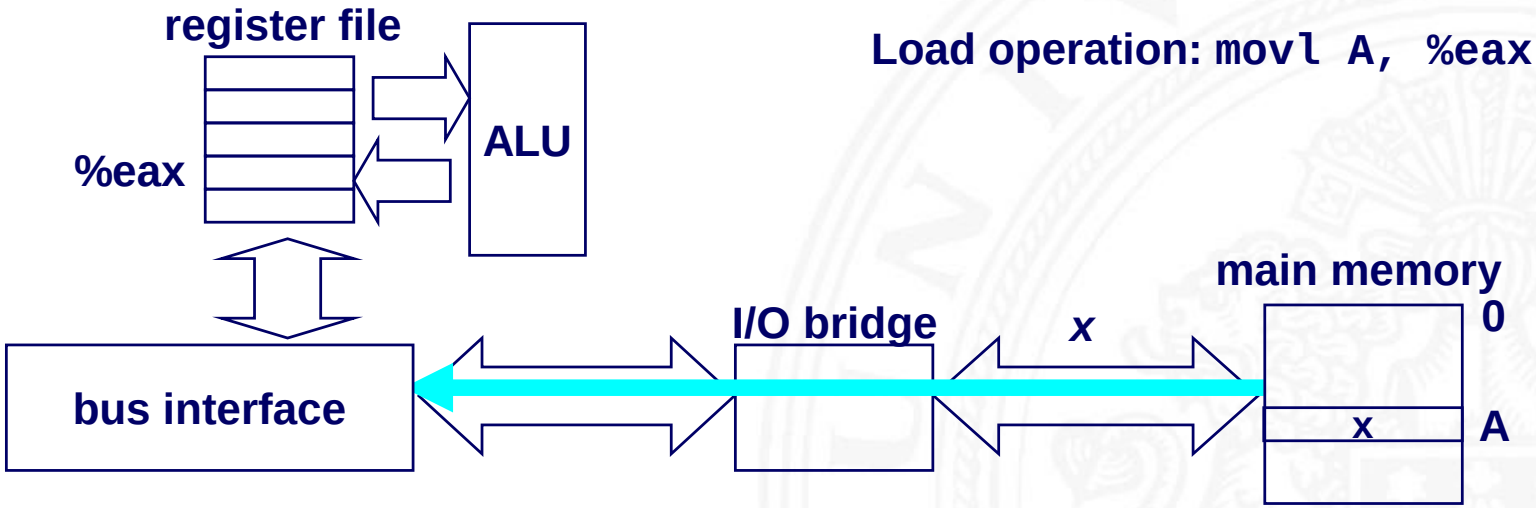


# lesender Speicherzugriff (cont.)

2.a Hauptspeicher liest Adresse A vom Speicherbus

2.b –"– ruft das Wort x unter der Adresse A ab

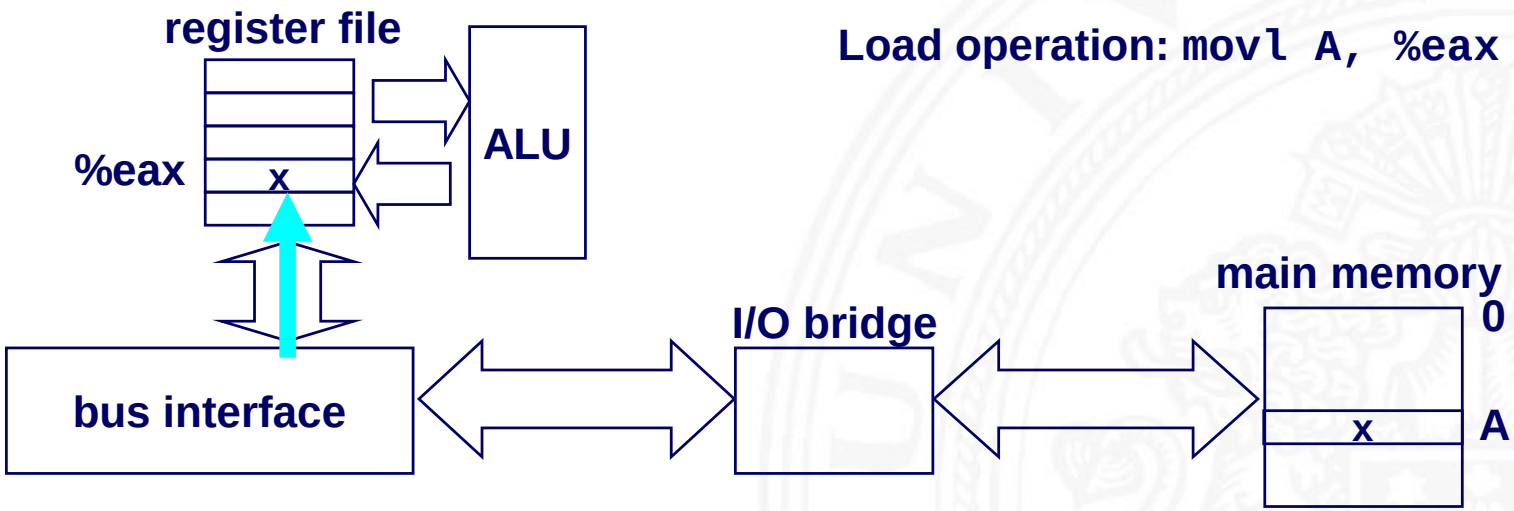
2.c –"– legt das Wort x auf den Bus



# lesender Speicherzugriff (cont.)

3.a CPU liest Wort x vom Bus

3.b –"– kopiert Wert x in Register `%eax`

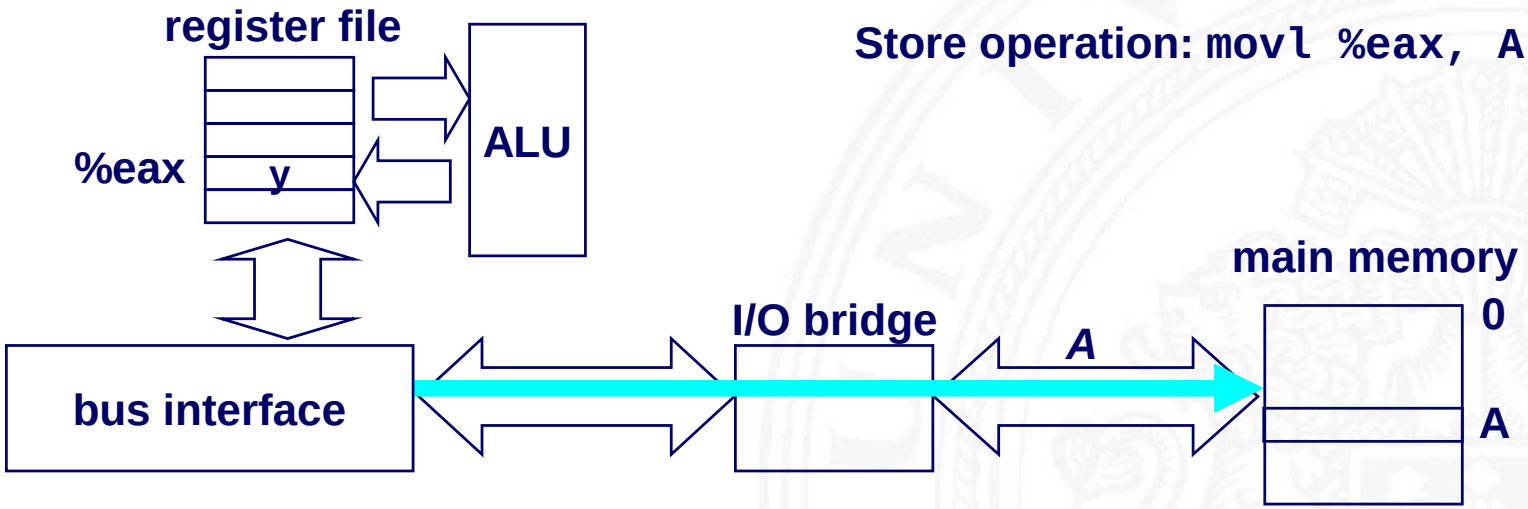


# schreibender Speicherzugriff

1 CPU legt die Adresse A auf den Bus

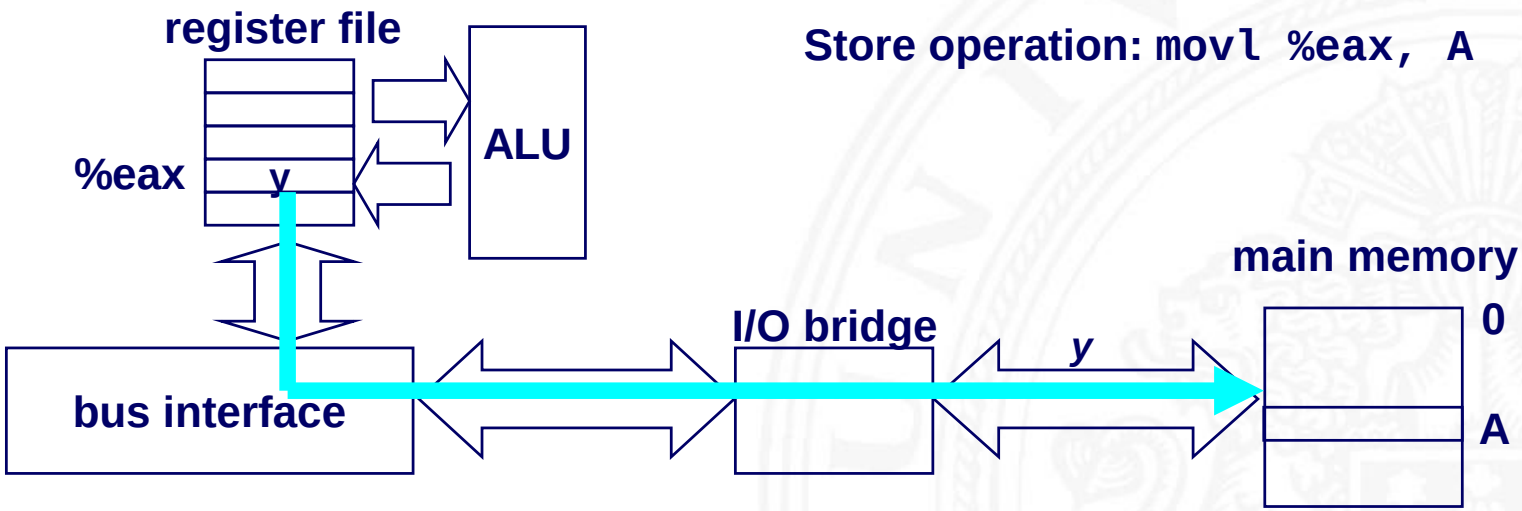
2.b Hauptspeicher liest Adresse

2.c –"– wartet auf Ankunft des Datenworts



# schreibender Speicherzugriff (cont.)

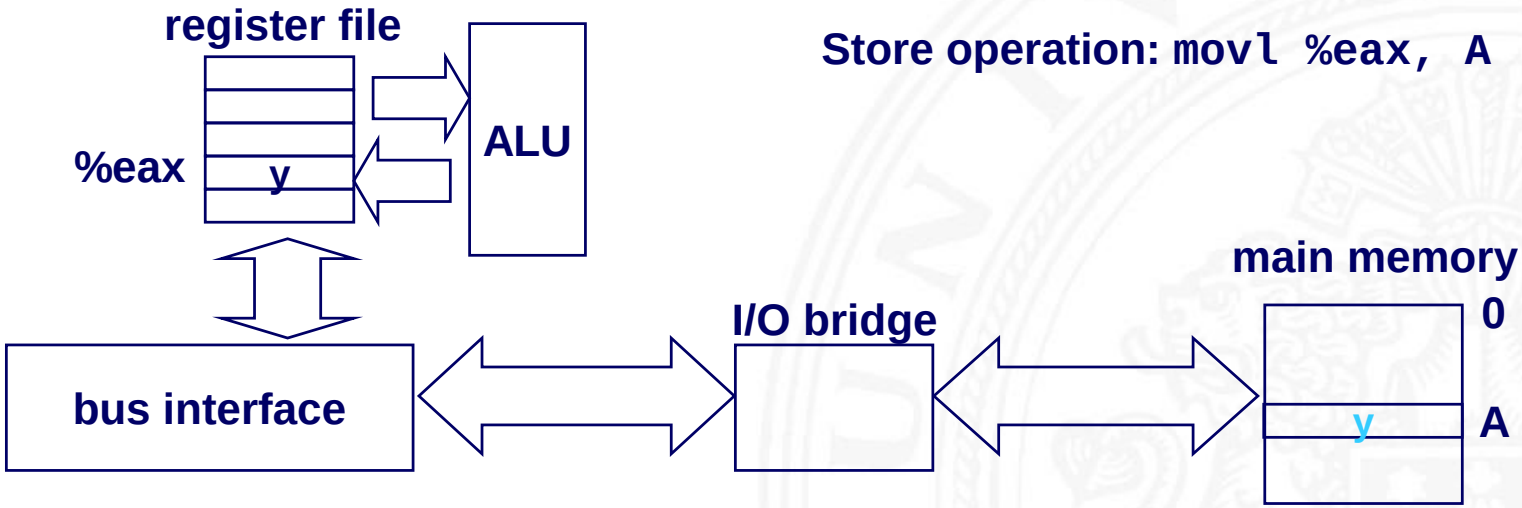
3. CPU legt Datenwort y auf den Bus



# schreibender Speicherzugriff (cont.)

4.a Hauptspeicher liest Datenwort y vom Bus

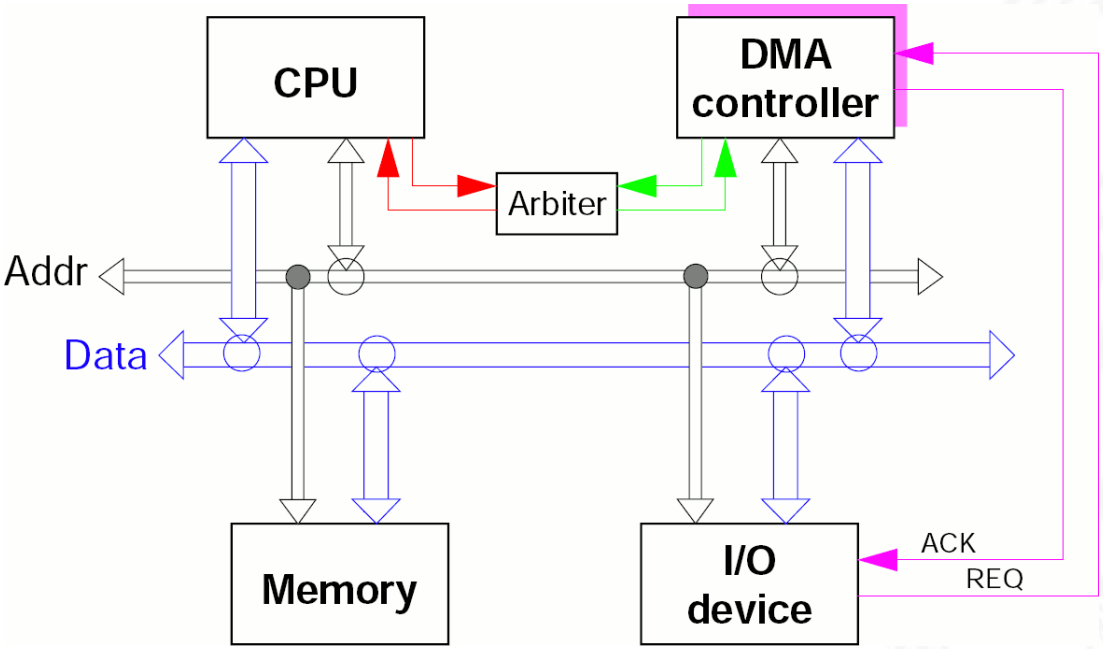
4.b –"– speichert Datenwort y unter Adresse A



# Speicheranbindung – DMA

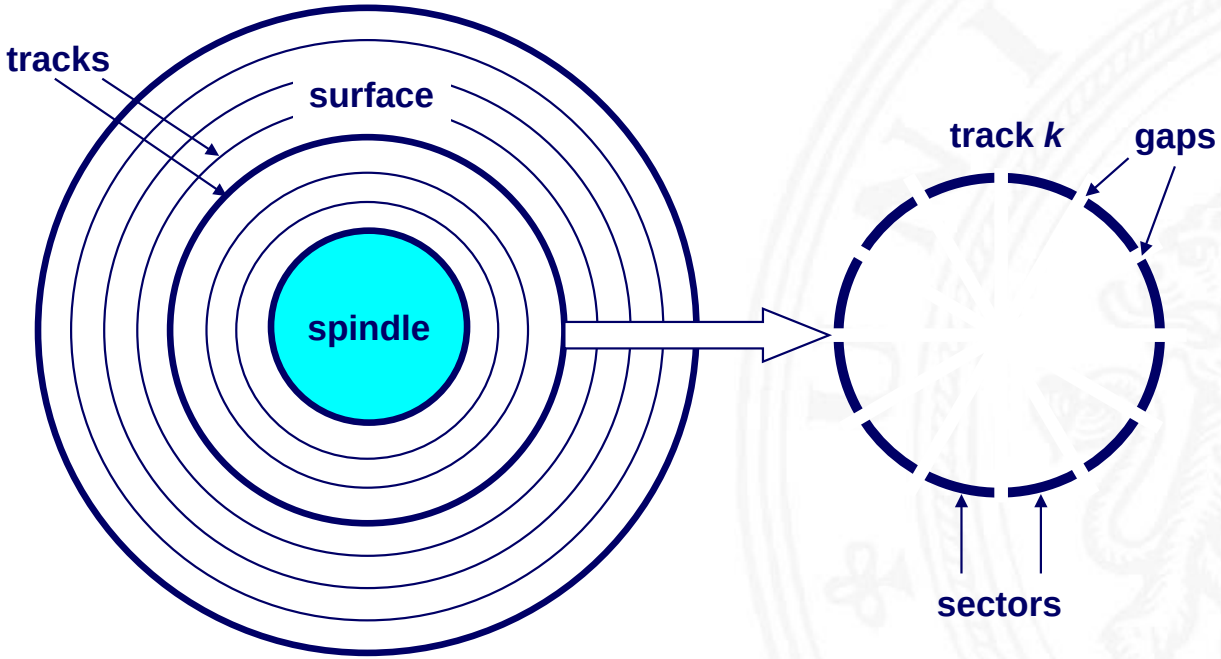
DMA – **D**irect **M**emory **A**ccess

- eigener Controller zum Datentransfer
- \+ Speicherzugriffe unabhängig von der CPU
- \+ CPU kann lokal (Register und Cache) weiterrechnen



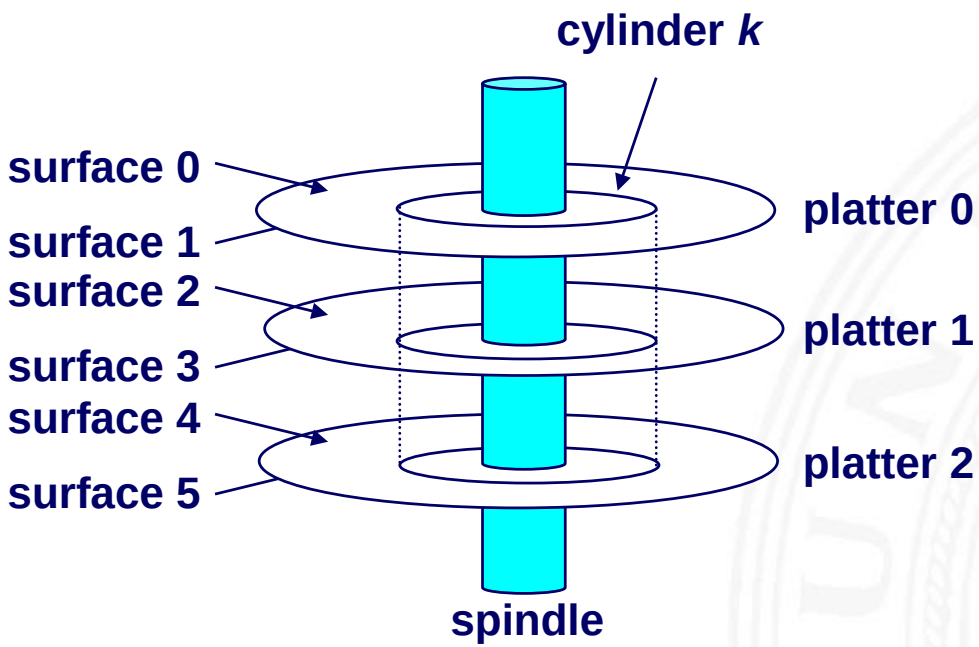
# Festplattengeometrie

- Platten mit jeweils zwei Oberflächen („surfaces“)
- konzentrische Ringe der Oberfläche bilden Spuren („tracks“)
- Spur unterteilt in Sektoren („sectors“),
  durch Lücken (“gaps”) getrennt



# Festplattengeometrie (cont.)

- untereinander liegende Spuren (mehrerer Platten) bilden einen Zylinder



# Festplattenkapazität

- Kapazität: Höchstzahl speicherbarer Bits
- bestimmende technologische Faktoren
  - Aufnahmedichte [bits/in]: # Bits / 1-Inch Segment einer Spur
  - Spurdichte [tracks/in]: # Spuren / 1-Inch (radial)
  - Flächendichte [bits/in$^2$]: Aufnahme- $\times$ Spurdichte
- Spuren unterteilt in getrennte Zonen („Recording Zones“)
  - jede Spur einer Zone hat gleichviel Sektoren (festgelegt durch die Ausdehnung der innersten Spur)
  - jede Zone hat unterschiedlich viele Sektoren/Spuren

# Festplattenkapazität (cont.)

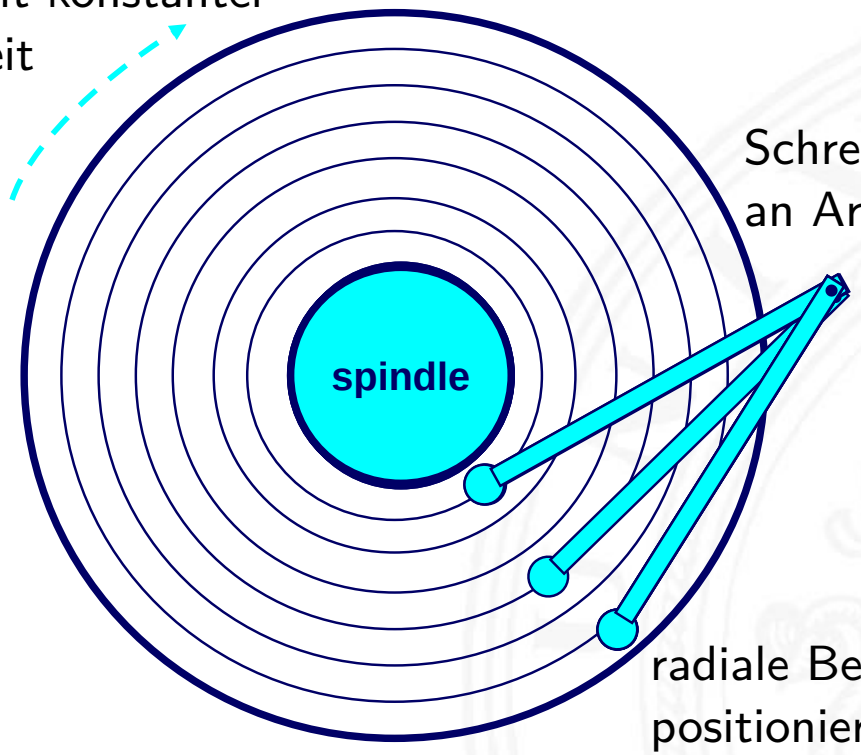
- Kapazität = Bytes/Sektor $\times$ $\varnothing$ Sektoren/Spur $\times$ Spuren/Oberfläche $\times$ Oberflächen/Platten $\times$ Platten/Festplatte
- Beispiel
  - 512 Bytes/Sektor
  - 300 Sektoren/Spuren (im Durchschnitt)
  - 20 000 Spuren/Oberfläche
  - 2 Oberflächen/Platten
  - 5 Platten/Festplatte

  $\Rightarrow$ Kapazität $= 512 \times 300 \times 20\,000 \times 2 \times 5$
  $= 30\,720\,000\,000 = 30{,}72\,\text{GB}$

# Festplatten-Operation

- Ansicht einer Platte

  Umdrehung mit konstanter Geschwindigkeit



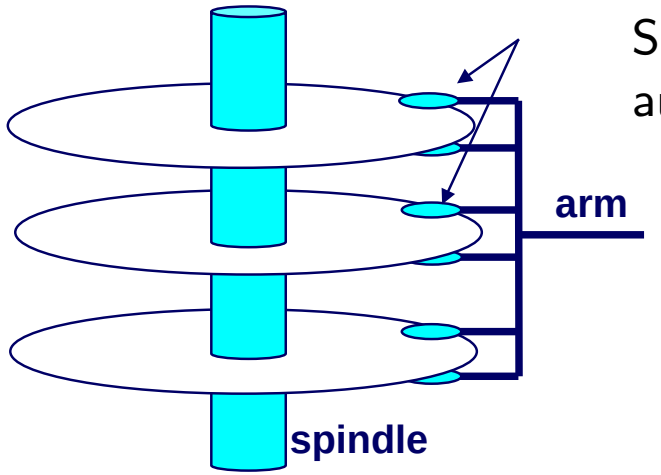

Schreib/Lese Kopf ist an Arm befestigt

„fliegt“ auf Luftpolster über Plattenoberfläche

radiale Bewegung des Arms positioniert Kopf auf Spuren

# Festplatten-Operation (cont.)

- Ansicht mehrerer Platten




Schreib/Lese Köpfe werden gemeinsam auf Zylindern positioniert

# Festplatten-Zugriffszeit

Durchschnittliche (avg) Zugriffszeit auf einen Zielsektor wird angenähert durch

- $T_{Zugriff} = T_{avgSuche} + T_{avgRotationslatenz} + T_{avgTransfer}$

Suchzeit ($T_{avgSuche}$)

- Zeit in der Schreib-Lese Köpfe („heads“) über den Zylinder mit dem Targetsektor positioniert werden
- üblicherweise $T_{avgSuche} = 8\,\text{ms}$

# Festplatten-Zugriffszeit (cont.)

Rotationslatenzzeit ($T_{avgRotationslatenz}$)

- Wartezeit, bis das erste Bit des Targetsektors unter dem Schreib-Lese-Kopf durchrotiert
- $T_{avgRotationslatenz} = 1/2 \times 1/\text{RPMs} \times 60\,\text{Sek}/1\,\text{Min}$

Transferzeit ($T_{avgTransfer}$)

- Zeit, in der die Bits des Targetsektors gelesen werden
- $T_{avgTransfer} = 1/\text{RPM} \times 1/(\text{Durchschn. } \#\,\text{Sektoren/Spur}) \times 60\,\text{Sek}/1\,\text{Min}$

# Festplatten-Zugriffszeit (cont.)

Beispiel für Festplatten-Zugriffszeit

- Umdrehungszahl = 7 200 RPM („Rotations per Minute“)
- Durchschnittliche Suchzeit = 8 ms
- Avg # Sektoren/Spur = 400

$\Rightarrow$ $T_{avgRotationslatenz}$
$= 1/2 \times (60\,\text{Sek}/7\,200\,\text{RPM}) \times 1\,000\,\text{ms/Sek} = 4\,\text{ms}$

$\Rightarrow$ $T_{avgTransfer}$
$= 60/7\,200\,\text{RPM} \times 1/400\,\text{Sek/Spur} \times 1\,000\,\text{ms/Sek} = 0{,}02\,\text{ms}$

$\Rightarrow$ $T_{avgZugriff}$
$= 8\,\text{ms} + 4\,\text{ms} + 0{,}02\,\text{ms}$

# Festplatten-Zugriffszeit (cont.)

Fazit

- Zugriffszeit wird von Suchzeit und Rotationslatenzzeit dominiert
- erstes Bit eines Sektors ist das „teuerste“, der Rest ist quasi umsonst
- SRAM Zugriffszeit ist ca. 4 ns  DRAM ca. 60 ns
- Kombination aus Zugriffszeit und Datentransfer
  - Festplatte ist ca. 40 000 mal langsamer als SRAM
  - 2 500 mal langsamer als DRAM

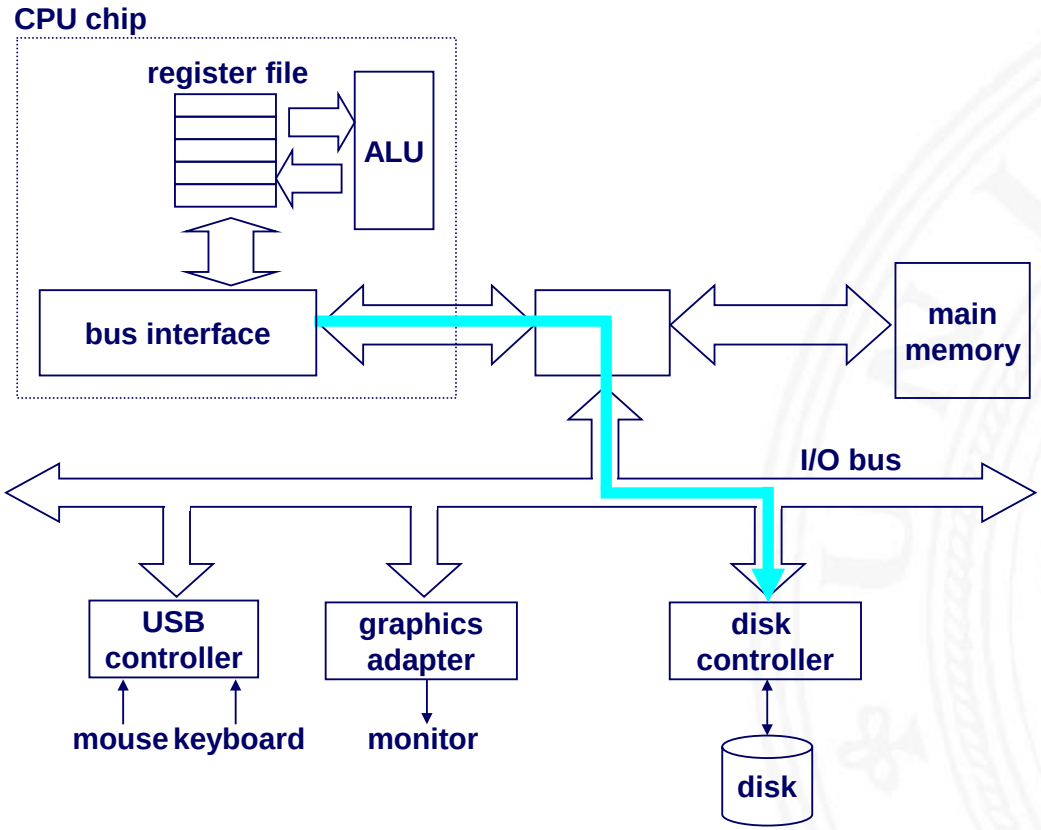
# Logische Festplattenblöcke

- simple, abstrakte Ansicht der komplexen Sektorengeometrie
  - verfügbare Sektoren werden als Sequenz logischer Blöcke der Größe $b$ modelliert (0,1,2,..,n)
- Abbildung der logischen Blöcke auf die tatsächlichen (physikalischen) Sektoren
  - durch Hard-/Firmware Einheit (Festplattencontroller)
  - konvertiert logische Blöcke zu Tripeln (Oberfläche, Spur, Sektor)
- Controller kann für jede Zone Ersatzzylinder bereitstellen

  $\Rightarrow$ Unterschied: „formatierte Kapazität“ und „maximale Kapazität“

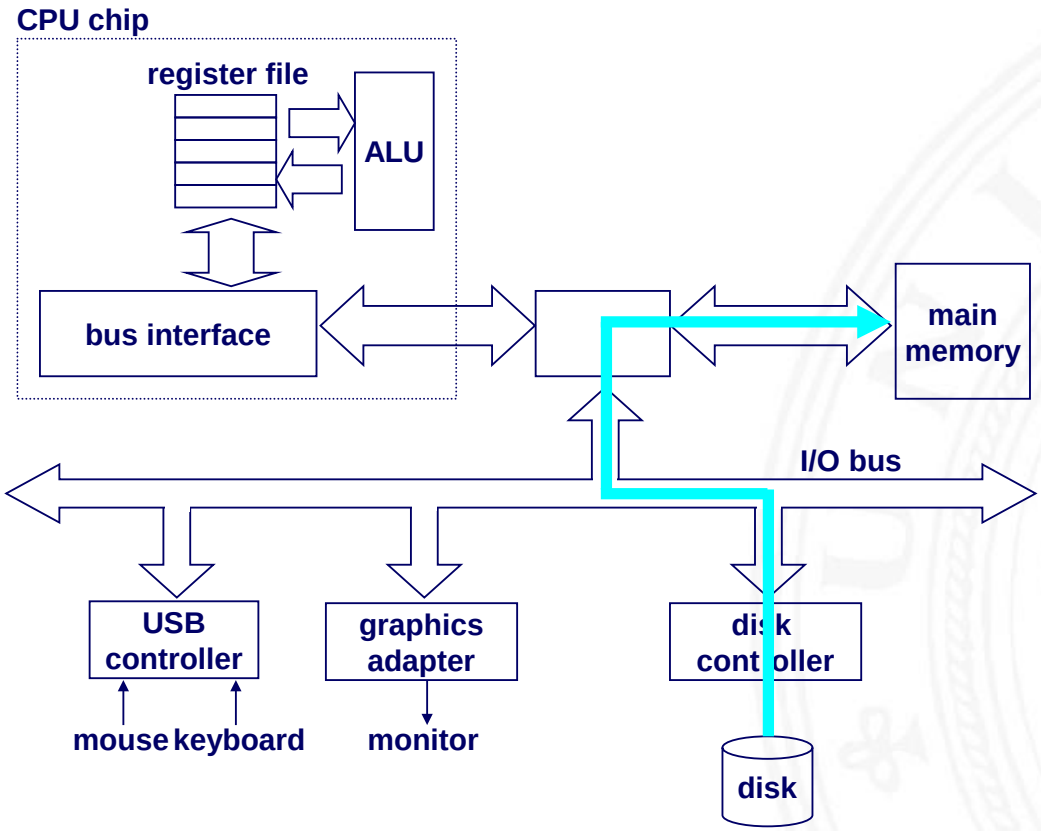
# Lesen eines Festplattensektors

1. CPU initiiert Lesevorgang von Festplatte
   auf Port (Adresse) des Festplattencontrollers wird geschrieben
   - Befehl, logische Blocknummer, Zielspeicheradresse



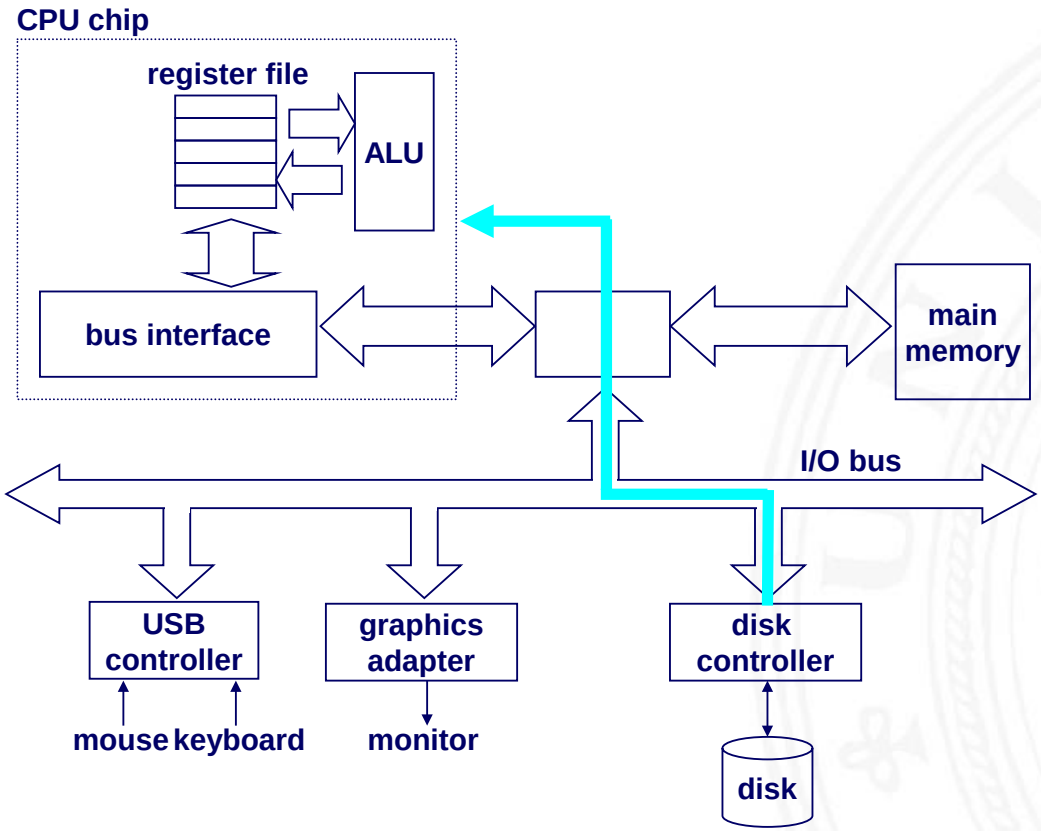
# Lesen eines Festplattensektors (cont.)

2. Festplattencontroller liest den Sektor aus
3. –"– führt DMA-Zugriff auf Hauptspeicher aus



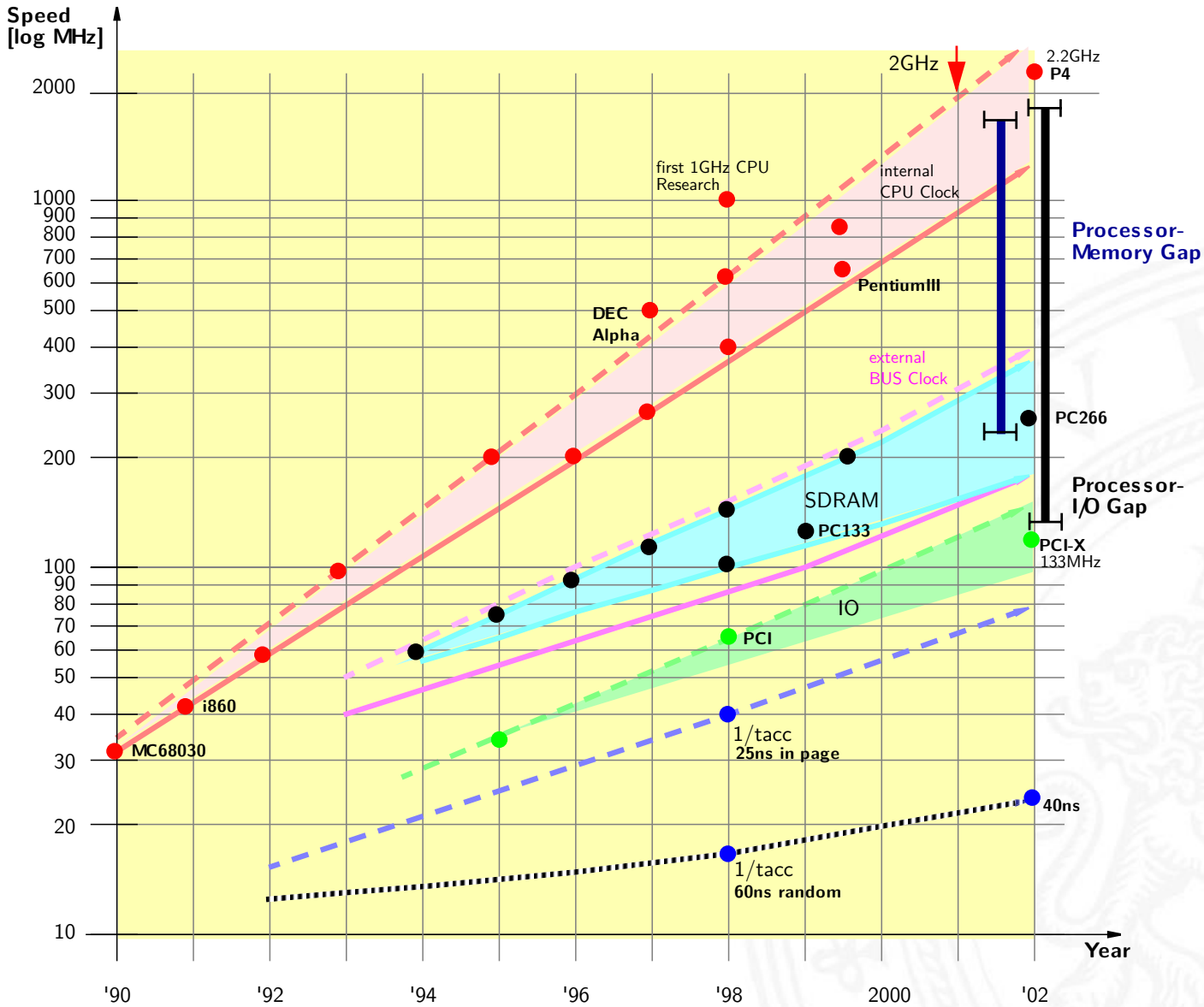
# Lesen eines Festplattensektors (cont.)

4. Festplattencontroller löst Interrupt aus



# Eigenschaften der Speichertypen

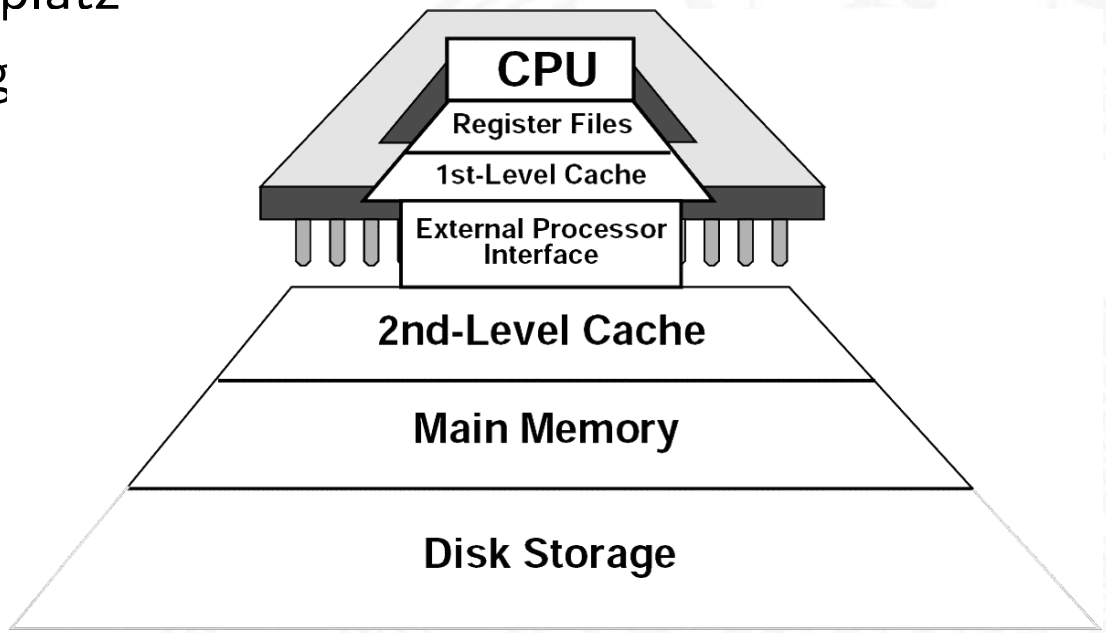
| Speicher | Vorteile | Nachteile |
|---|---|---|
| Register | sehr schnell | sehr teuer |
| SRAM | schnell | teuer, große Chips |
| DRAM | hohe Integration | Refresh nötig, langsam |
| Platten | billig, Kapazität | sehr langsam, mechanisch |
| Beispiel | Hauptspeicher | Festplatte |
| Latenz | 8 ns | 4 ms |
| Bandbreite | $\approx$ 38,4 GByte/sec (triple Channel) | $\approx$ 750 MByte/sec (typisch $<$ 300) |
| Kosten | 1 GByte, 5 € | 1 TByte, 40 € (4 Ct./GByte) |


Speed
[log MHz]
2000
1000
900
800
700
600
500
400
300
200
100
90
80
70
60
50
40
30
20
10
2GHz
2.2GHz
P4
first 1GHz CPU
Research
internal
CPU Clock
PentiumIII
DEC
Alpha
external
BUS Clock
PC266
Processor-
Memory Gap
SDRAM
PC133
Processor-
I/O Gap
PCI-X
133MHz
IO
PCI
i860
MC68030
1/tacc
25ns in page
40ns
1/tacc
60ns random
Year
'90
'92
'94
'96
'98
2000
'02

# Speicherhierarchie

Motivation

- Geschwindigkeit der Prozessoren
- Kosten für den Speicherplatz
- permanente Speicherung
  - magnetisch
  - optisch
  - mechanisch



# Speicherhierarchie (cont.)

Fundamentale Eigenschaften von Hard- und Software

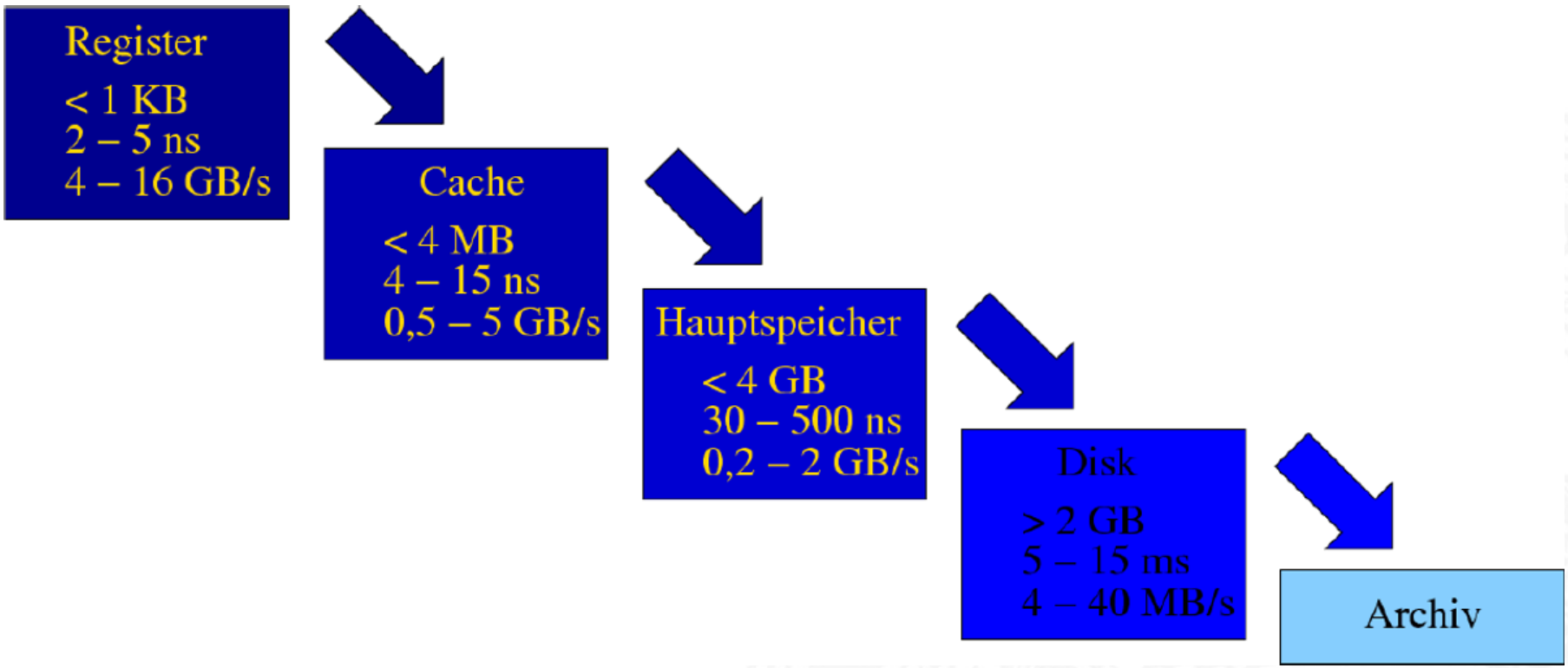
- schnelle vs. langsame Speichertechnologie
  schnell : hohe Kosten/Byte geringe Kapazität
  langsam : geringe –"– hohe –"–
- Abstand zwischen CPU und Hauptspeichergeschwindigkeit vergrößert sich
- Lokalität der Programme wichtig
  - kleiner Adressraum im Programmkontext
  - gut geschriebene Programme haben meist eine gute Lokalität

$\Rightarrow$ Motivation für spezielle Organisation von Speichersystemen
**Speicherhierarchie**

# Speicherhierarchie (cont.)

| Cache Type | What Cached | Where Cached | Latency (cycles) | Managed By |
|---|---|---|---|---|
| Registers | 4-byte word | CPU registers | 0 | Compiler |
| TLB | Address translations | On-Chip TLB | 0 | Hardware |
| L1 cache | 32-byte block | On-Chip L1 | 1 | Hardware |
| L2 cache | 32-byte block | Off-Chip L2 | 10 | Hardware |
| Virtual Memory | 4-KB page | Main memory | 100 | Hardware+ OS |
| Buffer cache | Parts of files | Main memory | 100 | OS |
| Network buffer cache | Parts of files | Local disk | 10,000,000 | AFS/NFS client |
| Browser cache | Web pages | Local disk | 10,000,000 | Web browser |
| Web cache | Web pages | Remote server disks | 1,000,000,000 | Web proxy server |

# Verwaltung der Speicherhierarchie



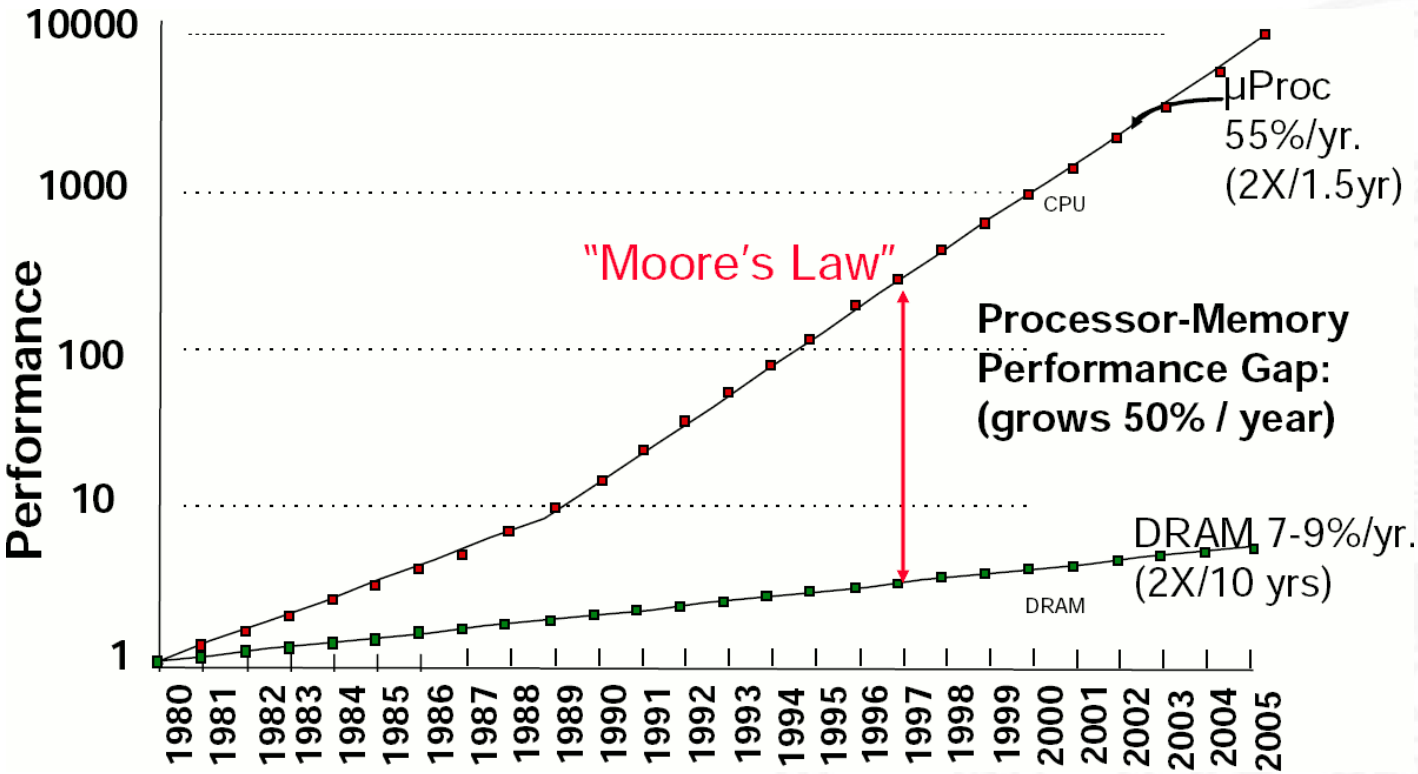
# Verwaltung der Speicherhierarchie (cont.)

Verwaltung der Speicherhierarchie

- Register $\leftrightarrow$ Memory
  - Compiler
  - Assembler-Programmierer
- Cache $\leftrightarrow$ Memory
  - Hardware
- Memory $\leftrightarrow$ Disk
  - Hardware und Betriebssystem (Paging)
  - Programmierer (Files)

# Cache

- „*Memory Wall*“: DRAM zu langsam für CPU

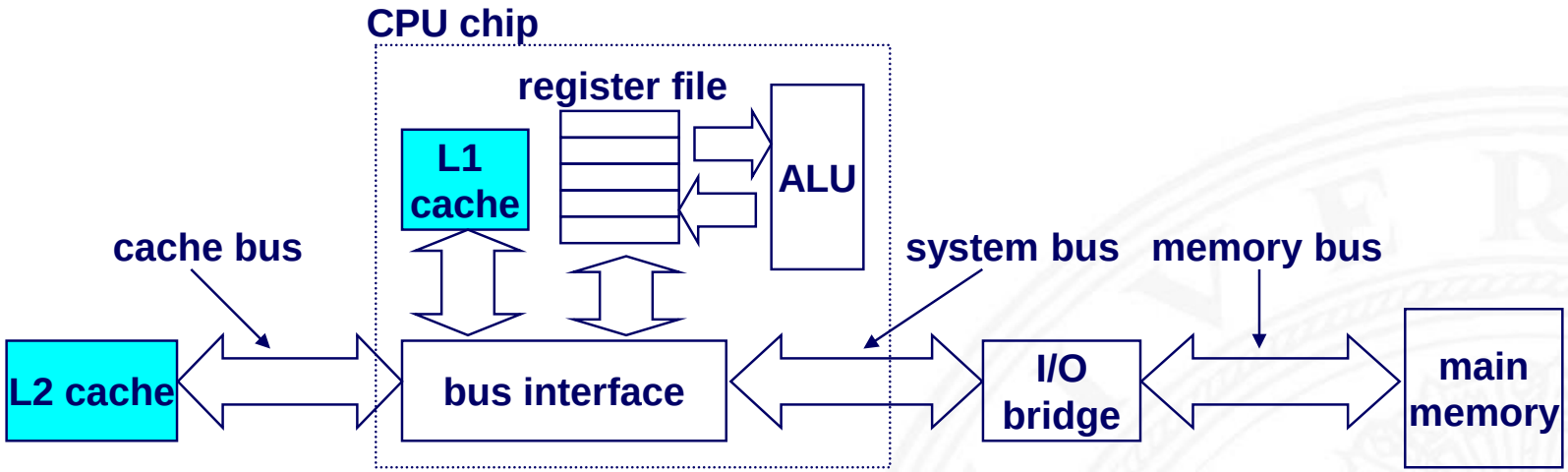


# Cache (cont.)

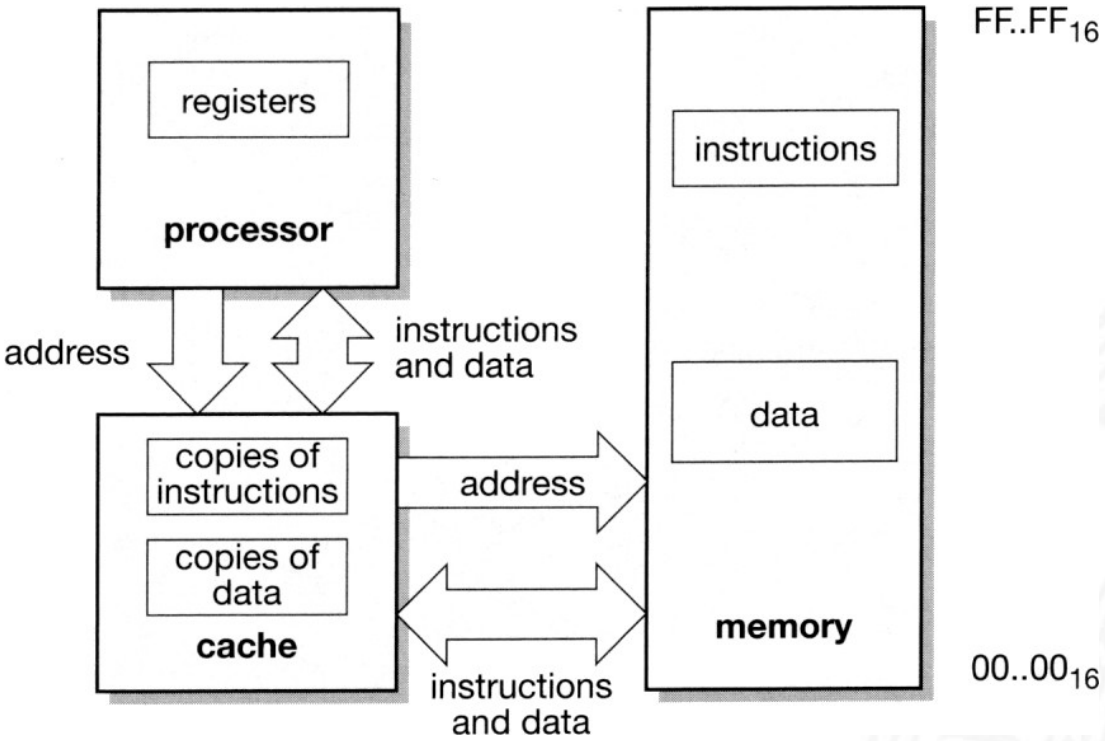
⇒ Cache als schneller Zwischenspeicher zum Hauptspeicher

- technische Realisierung: SRAM
- transparenter Speicher
  - Cache ist für den Programmierer nicht sichtbar!
  - wird durch Hardware verwaltet
- `http://de.wikipedia.org/wiki/Cache`
  `http://en.wikipedia.org/wiki/Cache`
- enthält Hauptspeicherblöcke mit erhöhter Zugriffswahrscheinlichkeit
- CPU referenziert Adresse
  - parallele Suche in L1 (level 1), L2... und Hauptspeicher
  - erfolgreiche Suche liefert Datum, Abbruch laufender Suchen

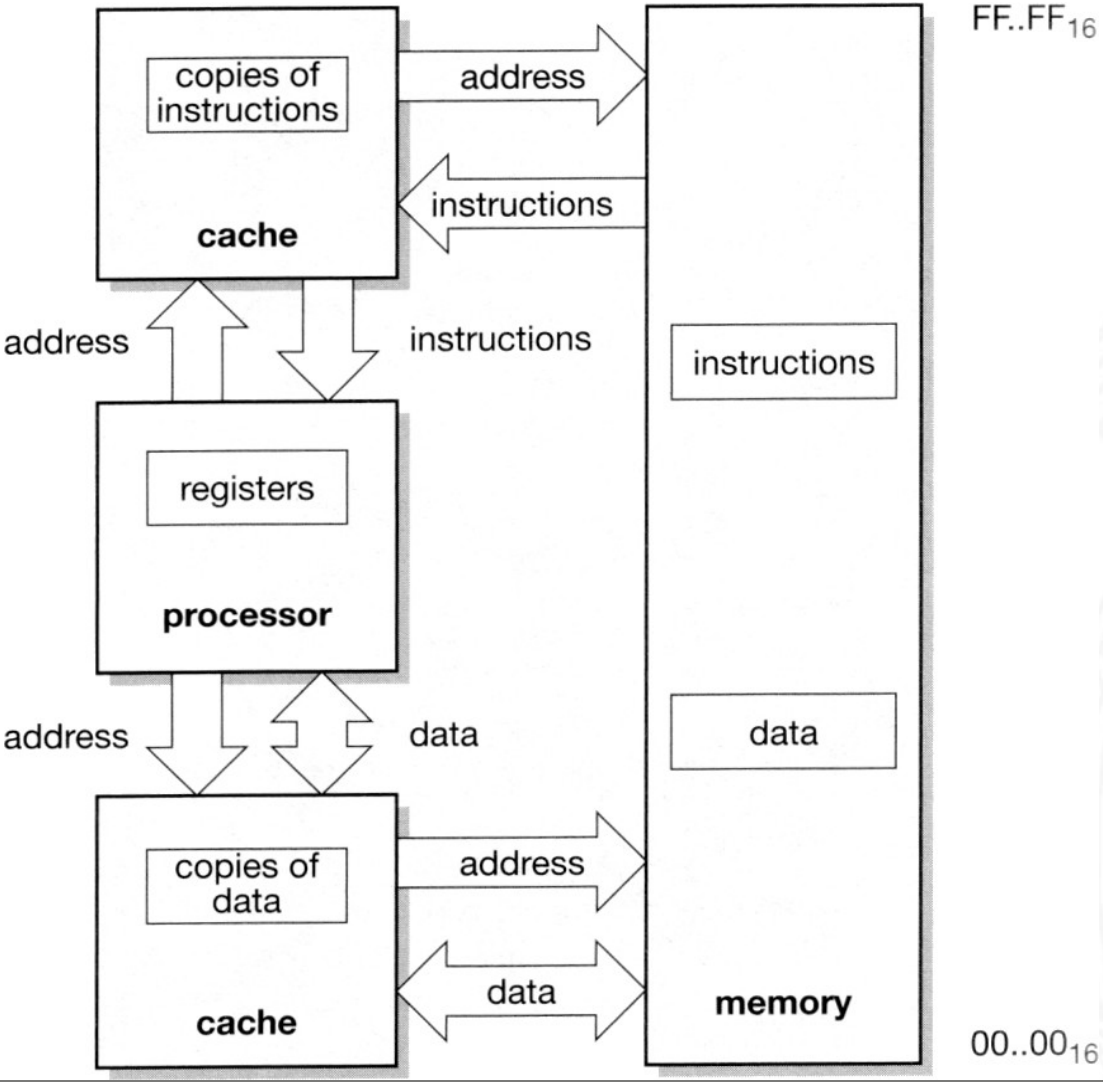
# Cache (cont.)


CPU chip
register file
L1 cache
ALU
cache bus
system bus
memory bus
L2 cache
bus interface
I/O bridge
main memory

# gemeinsamer Cache / „unified Cache“

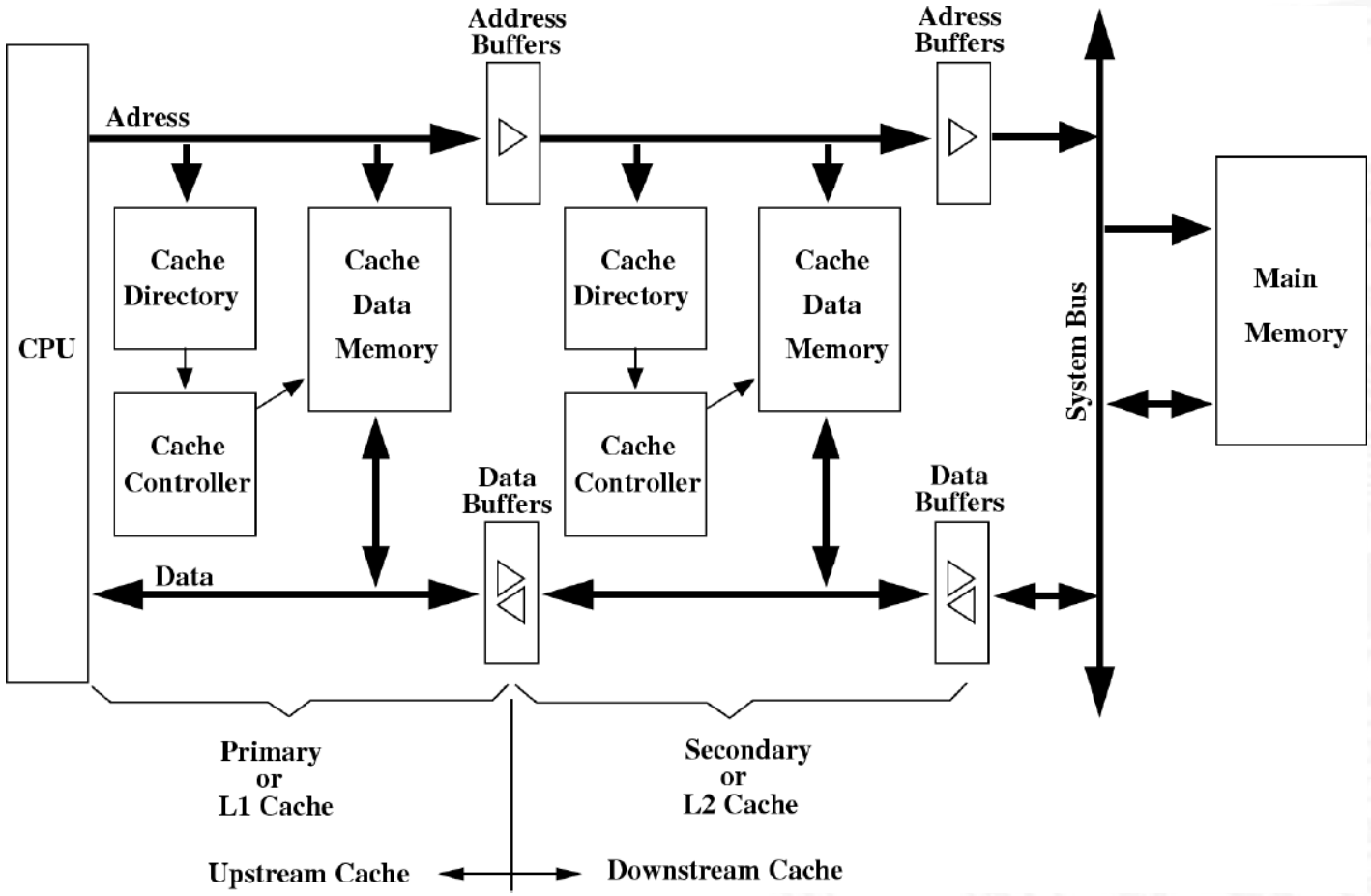


# separate Instruction-/Data Caches



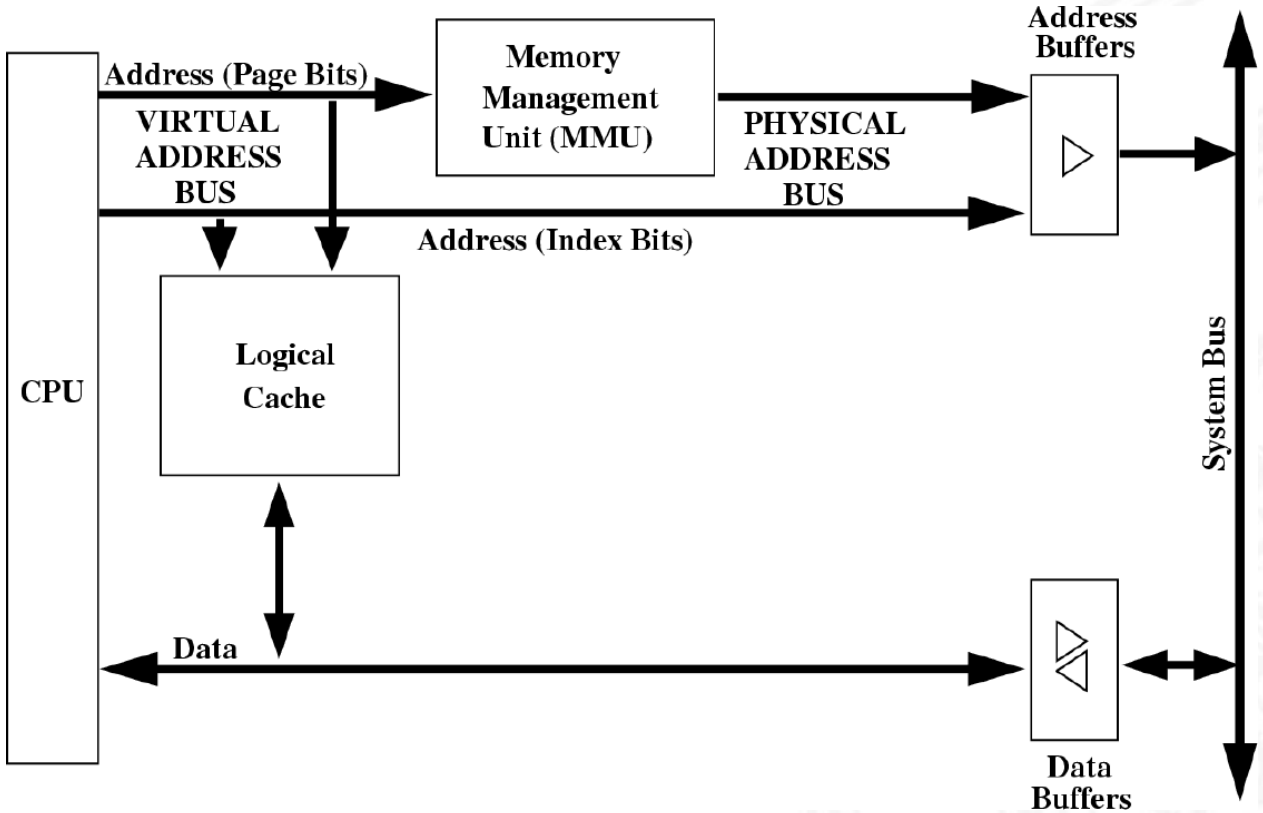
# Cache – Position

- First- und Second-Level Cache



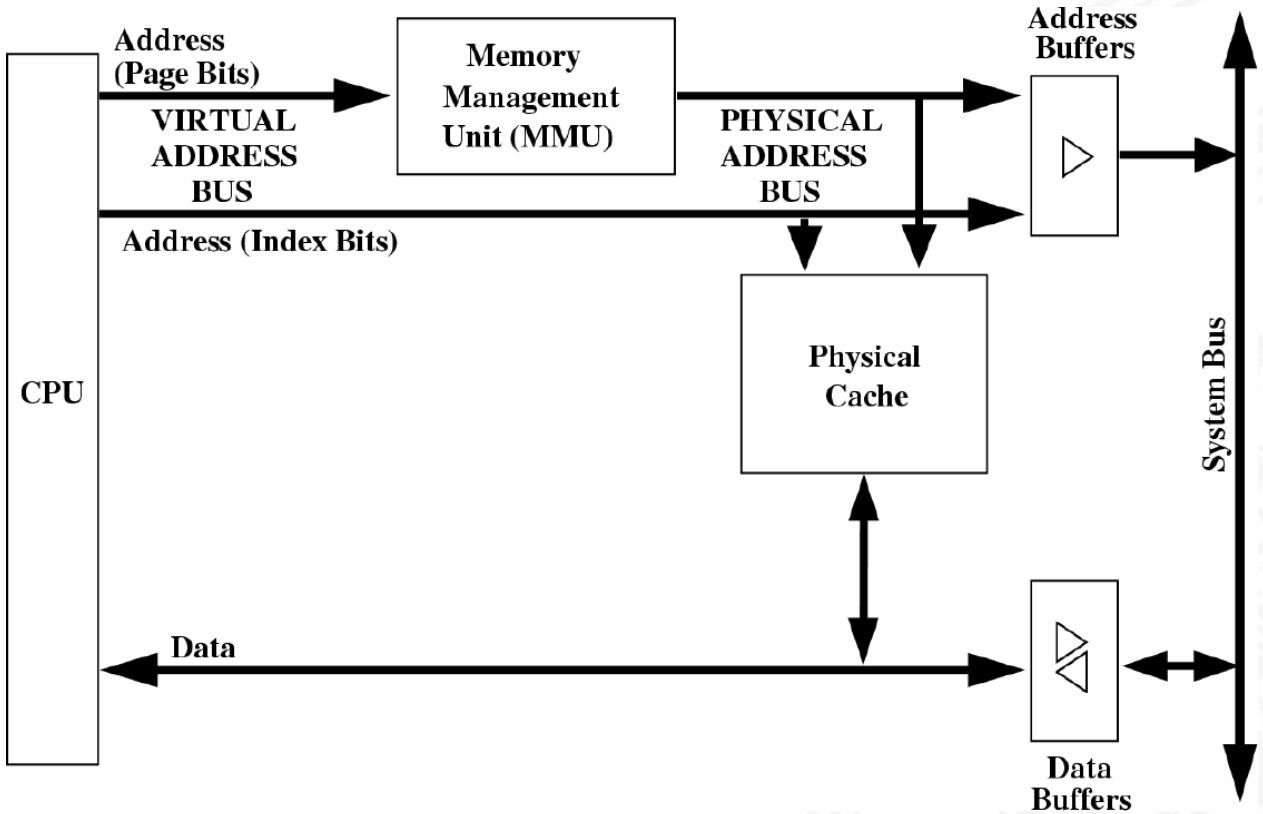
# Cache – Position (cont.)

- Virtueller Cache
  - \+ Adressumrechnung durch MMU oft nicht nötig
  - – Cache leeren bei Kontextwechseln



# Cache – Position (cont.)

- Physikalischer Cache
  - \+ Cache muss nie geleert werden
  - – Adressumrechnung durch MMU immer nötig



# Cache – Position (cont.)

- typische Cache Organisation
  - First-Level Cache: getrennte Instruktions- und Daten-Caches
  - Second-Level Cache: gemeinsamer Cache je Prozessorkern
  - Third-Level Cache: gemeinsamer Cache für alle Prozessorkerne
- bei mehreren Prozessoren / Prozessorkernen $\Rightarrow$ Cache-Kohärenz wichtig
  - gemeinsam genutzte Daten konsistent halten (s.u.)

# Cache – Strategie

Cachestrategie: *Welche Daten sollen in den Cache?*
Diejenigen, die bald wieder benötigt werden!

- *temporale Lokalität*:
  die Daten, die zuletzt häufig gebraucht wurden
- *räumliche Lokalität*:
  die Daten, die nahe den zuletzt gebrauchten liegen
- verschiedene Platzierungs-, Ersetzungs- und Rückschreibestategien für den Cache

# Cache – Performanz

## Cacheperformanz

- Begriffe

| | | |
|---|---|---|
| Treffer (Hit) | | Zugriff auf Datum, ist bereits im Cache |
| Fehler (Miss) | | –"– ist nicht –"– |
| Treffer-Rate | $R_{Hit}$ | Wahrscheinlichkeit, Datum ist im Cache |
| Fehler-Rate | $R_{Miss}$ | $1 - R_{Hit}$ |
| Hit-Time | $T_{Hit}$ | Zeit, bis Datum bei Treffer geliefert wird |
| Miss-Penalty | $T_{Miss}$ | zusätzlich benötigte Zeit bei Fehler |

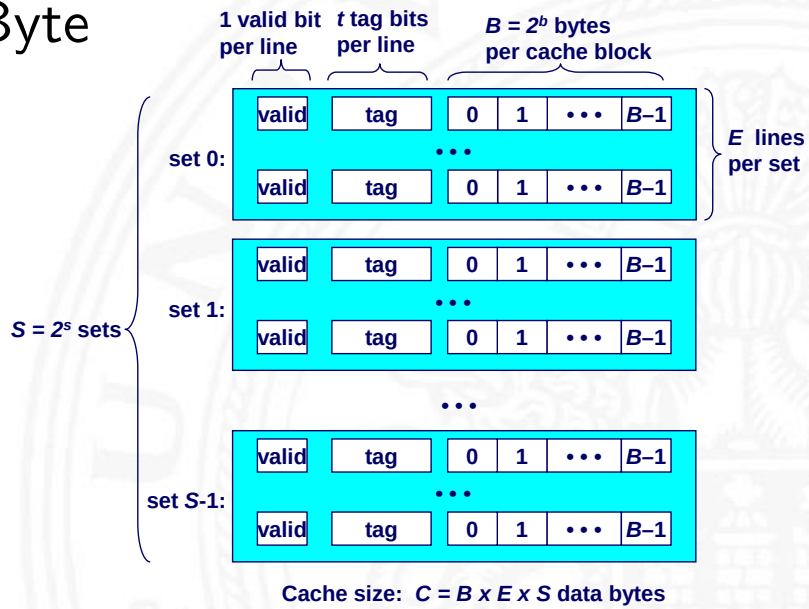
- Mittlere Speicherzugriffszeit $= T_{Hit} + R_{Miss} \cdot T_{Miss}$
- Beispiel

  $T_{Hit} = 1\,Takt,\ T_{Miss} = 20\,Takte,\ R_{Miss} = 5\,\%$

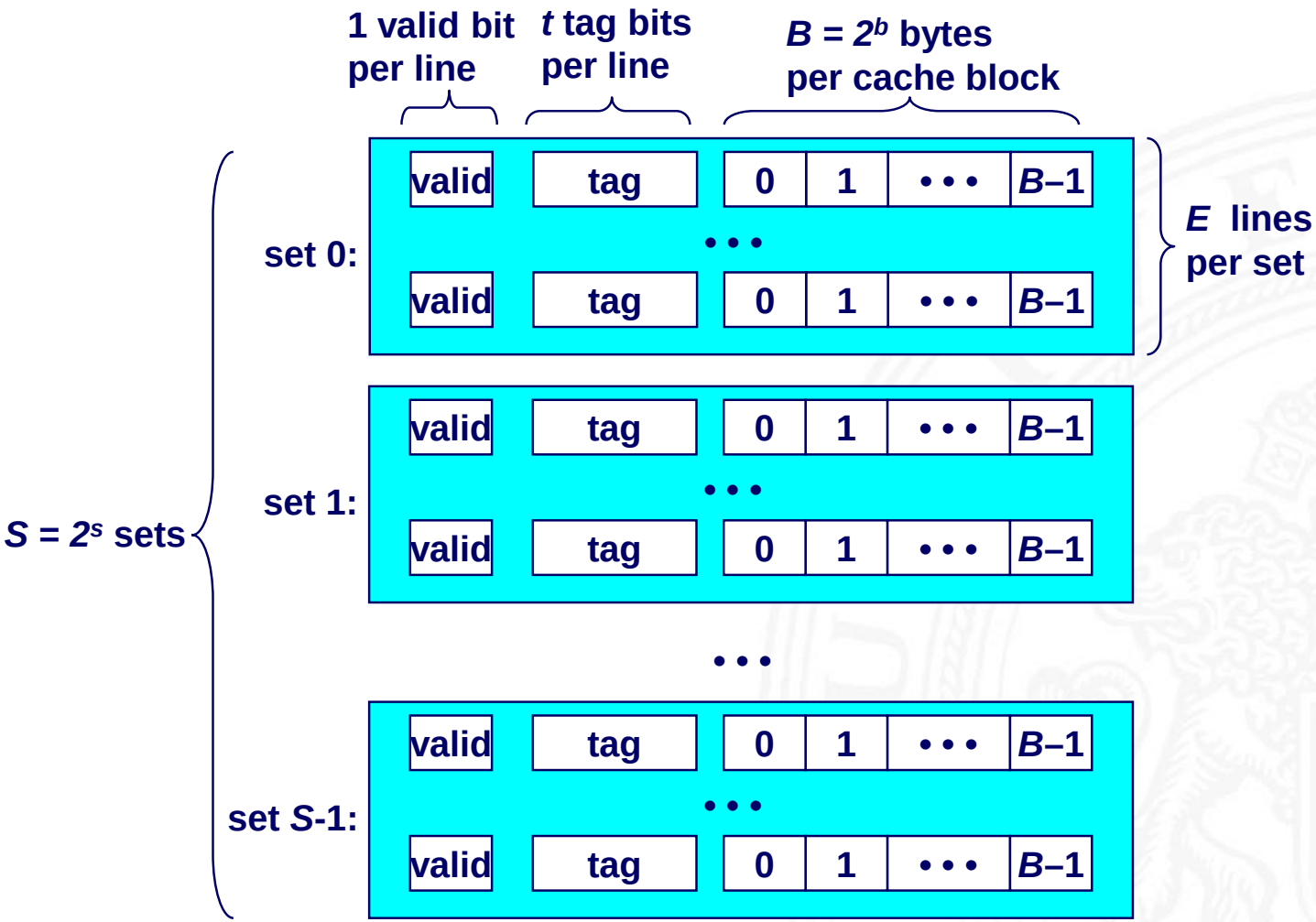
  Mittlere Speicherzugriffszeit $= 2\,Takte$

# Cache Organisation

- Cache ist ein Array von Speicher-Bereichen („sets“)
- jeder Bereich enthält eine oder mehrere Zeilen
- jede  Zeile enthält einen Datenblock
- jeder Block enthält mehrere Byte

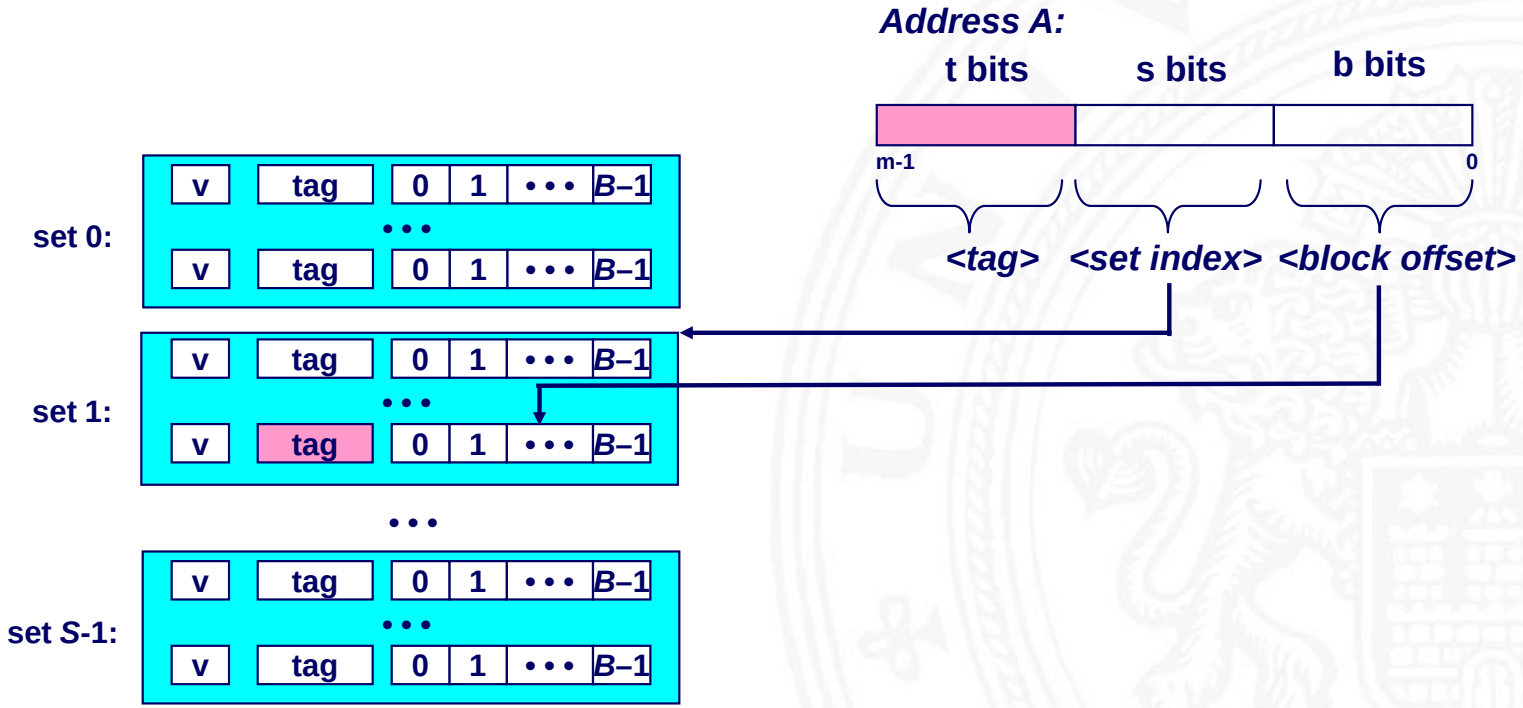


# Cache Organisation (cont.)



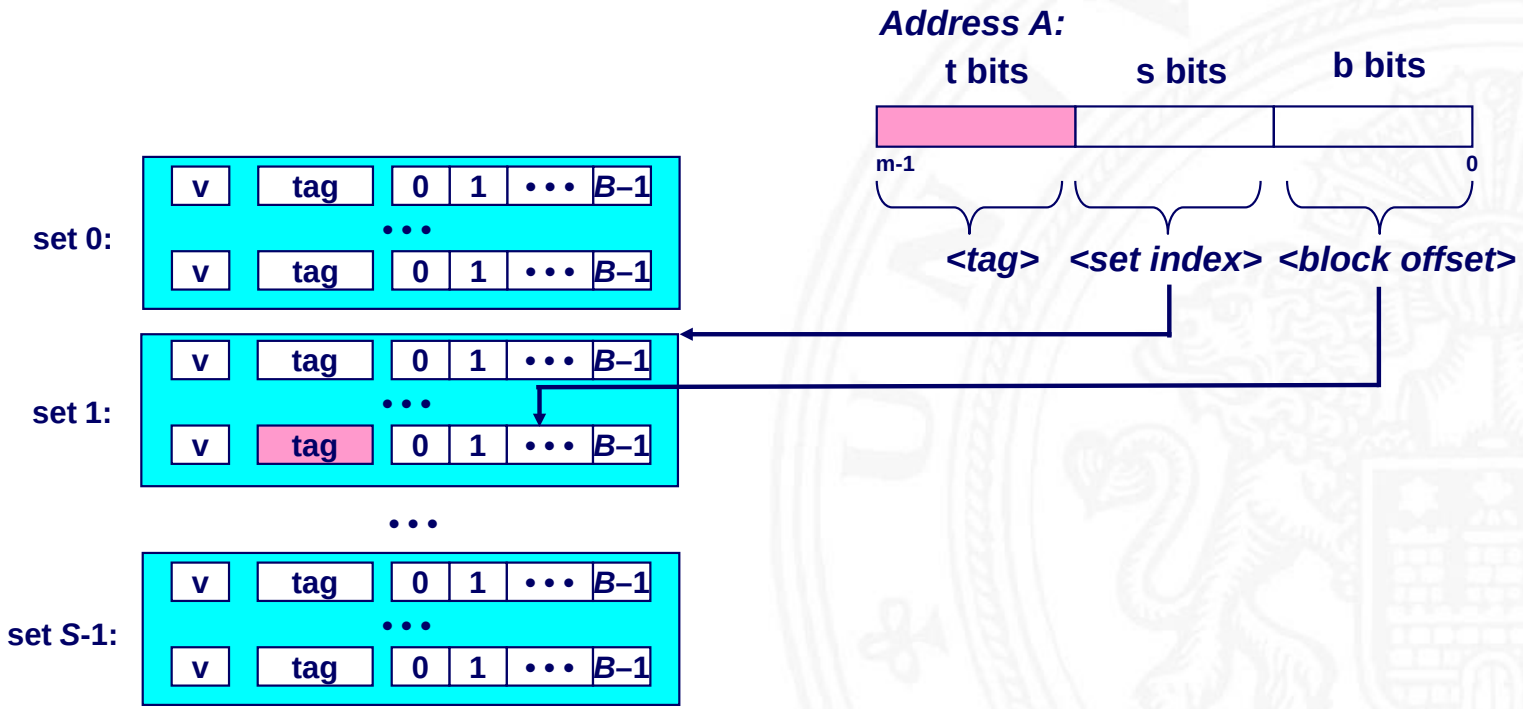
# Adressierung von Caches

- Adressteil $\langle set\ index \rangle$ von $A$ bestimmt Bereich („set“)
- Adresse $A$ ist im Cache, wenn
  1. Adressteil $\langle tag \rangle$ von $A$ = „tag“ Bits des Bereichs
  2. Cache-Zeile ist als gültig markiert („valid“)



# Adressierung von Caches (cont.)

- Cache-Zeile ("cache line") enthält Datenbereich von $2^b$ Byte
- gesuchtes Wort mit Offset $\langle block\ offset \rangle$



# Cache – Organisation

- *Welchen Platz im Cache belegt ein Datum des Hauptspeichers?*
- drei Verfahren

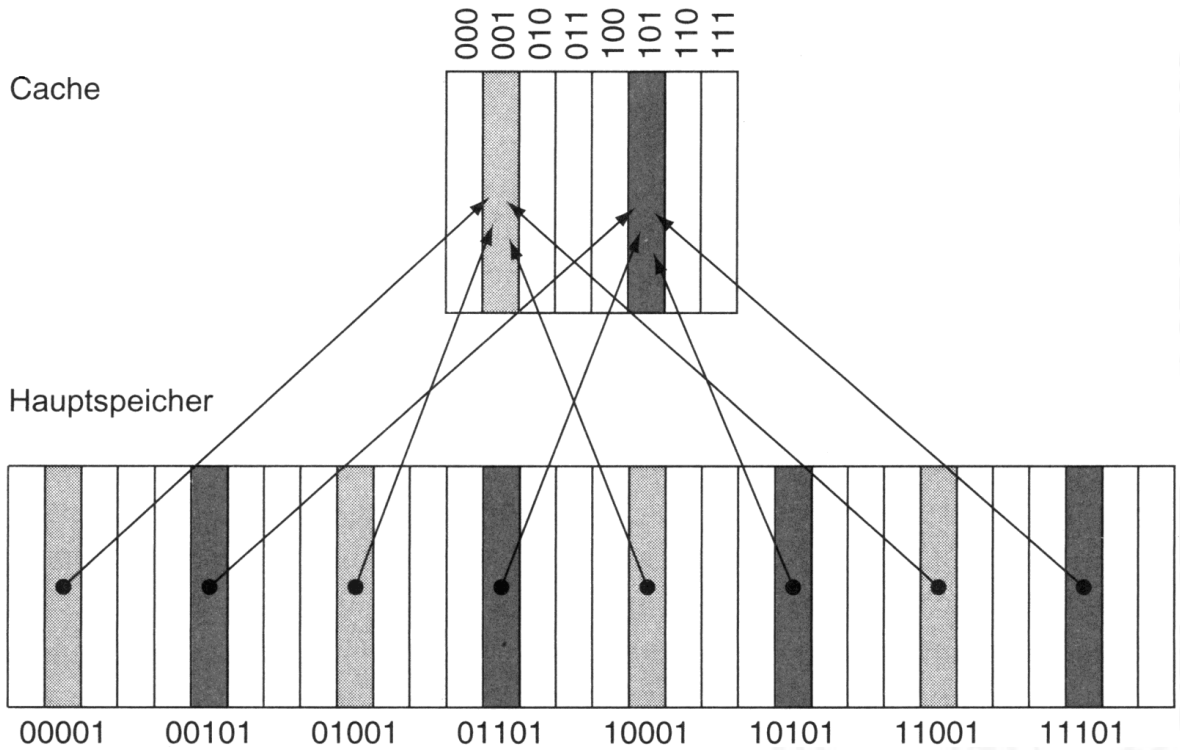
  direkt abgebildet / direct mapped jeder Speicheradresse ist genau eine Cache-Speicherzelle zugeordnet

  n-fach bereichsassoziativ / set associative jeder Speicheradresse ist eine von $E$ möglichen Cache-Speicherzellen zugeordnet

  voll-assoziativ jeder Speicheradresse kann jede beliebige Cache-Speicherzelle zugeordnet werden

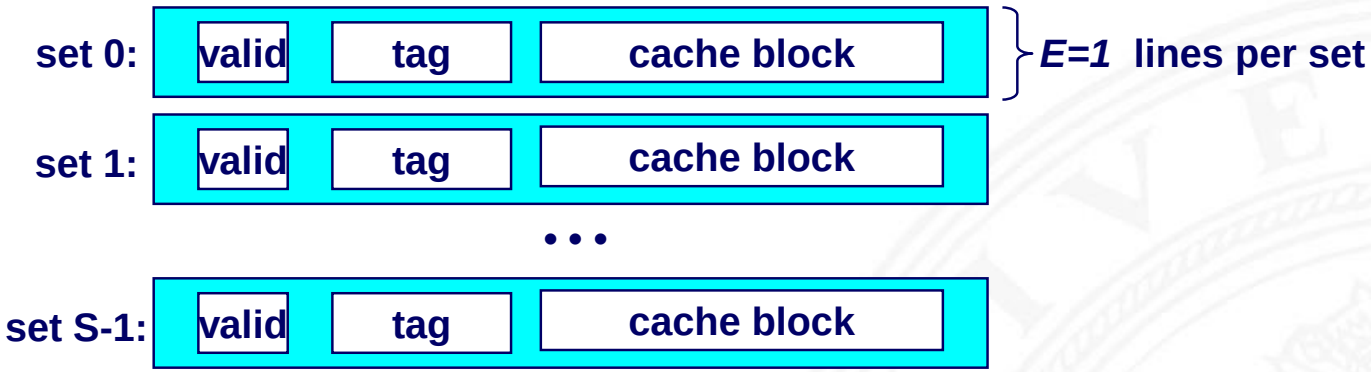
# Cache: direkt abgebildet / „direct mapped“

- jeder Adresse ist genau eine Speicherzelle im Cache zugeordnet



# Cache: direkt abgebildet / „direct mapped“ (cont.)

- verfügt über genau 1 Zeile pro Bereich  $S$ Bereiche (**S**ets)




\+ einfachste Cache-Art

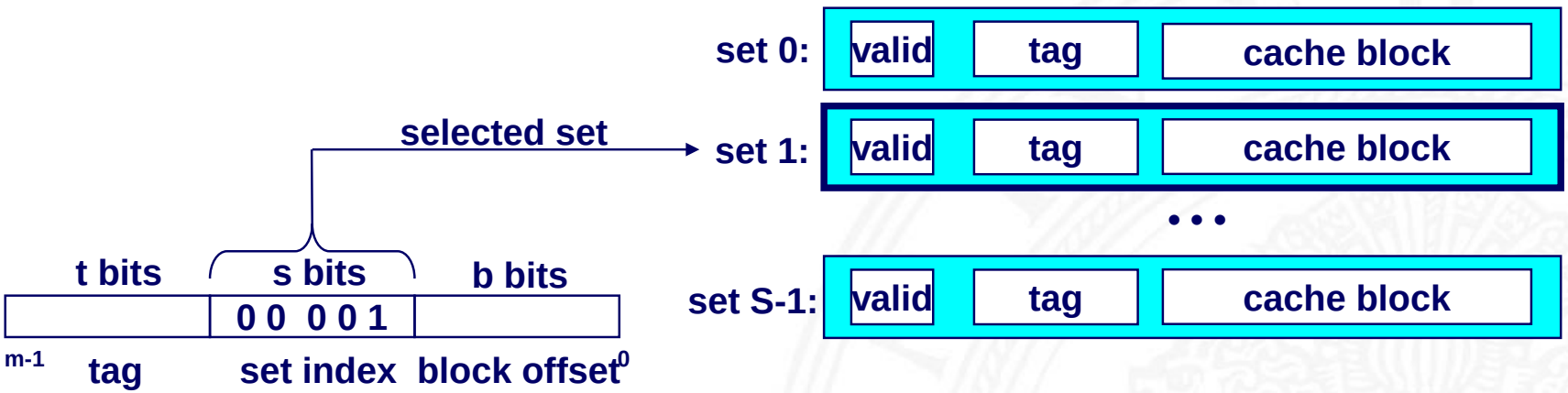
\+ große Caches möglich

– Effizienz, z.B. Zugriffe auf $A$, $A + n \cdot S$...

# Cache: direkt abgebildet / „direct mapped“ (cont.)

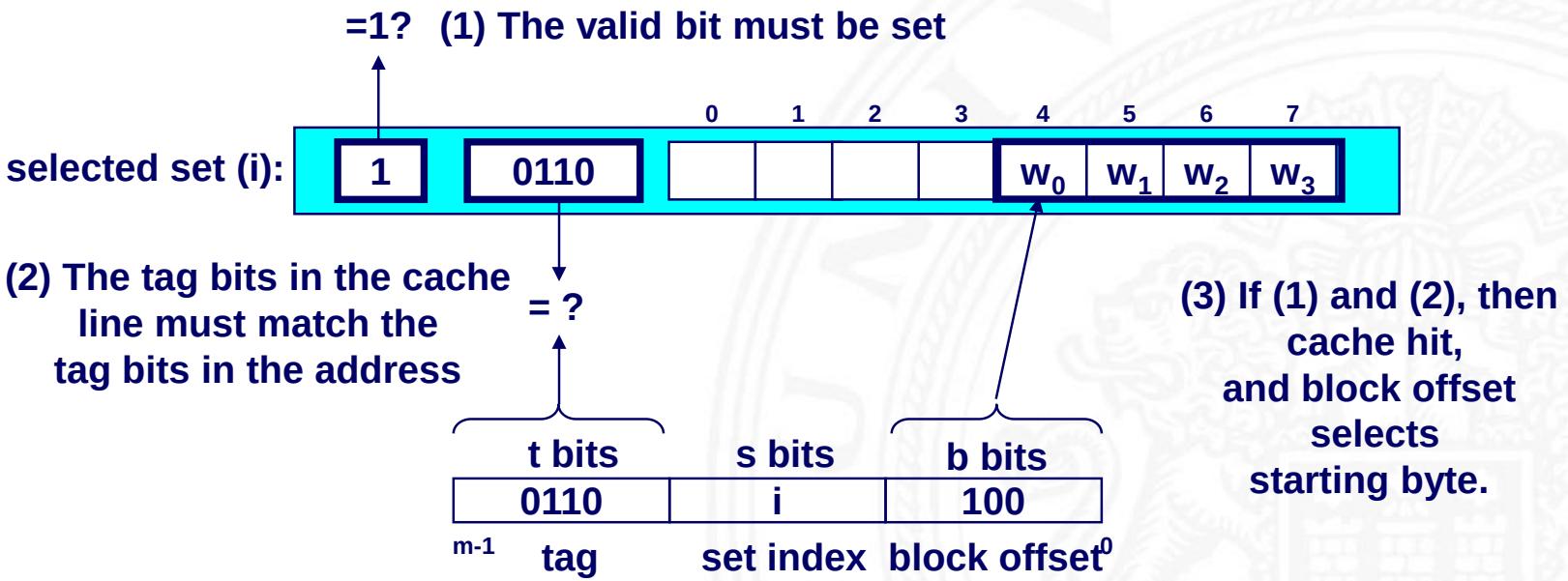
Zugriff auf direkt abgebildete Caches

1. Bereichsauswahl durch Bits ⟨*set index*⟩



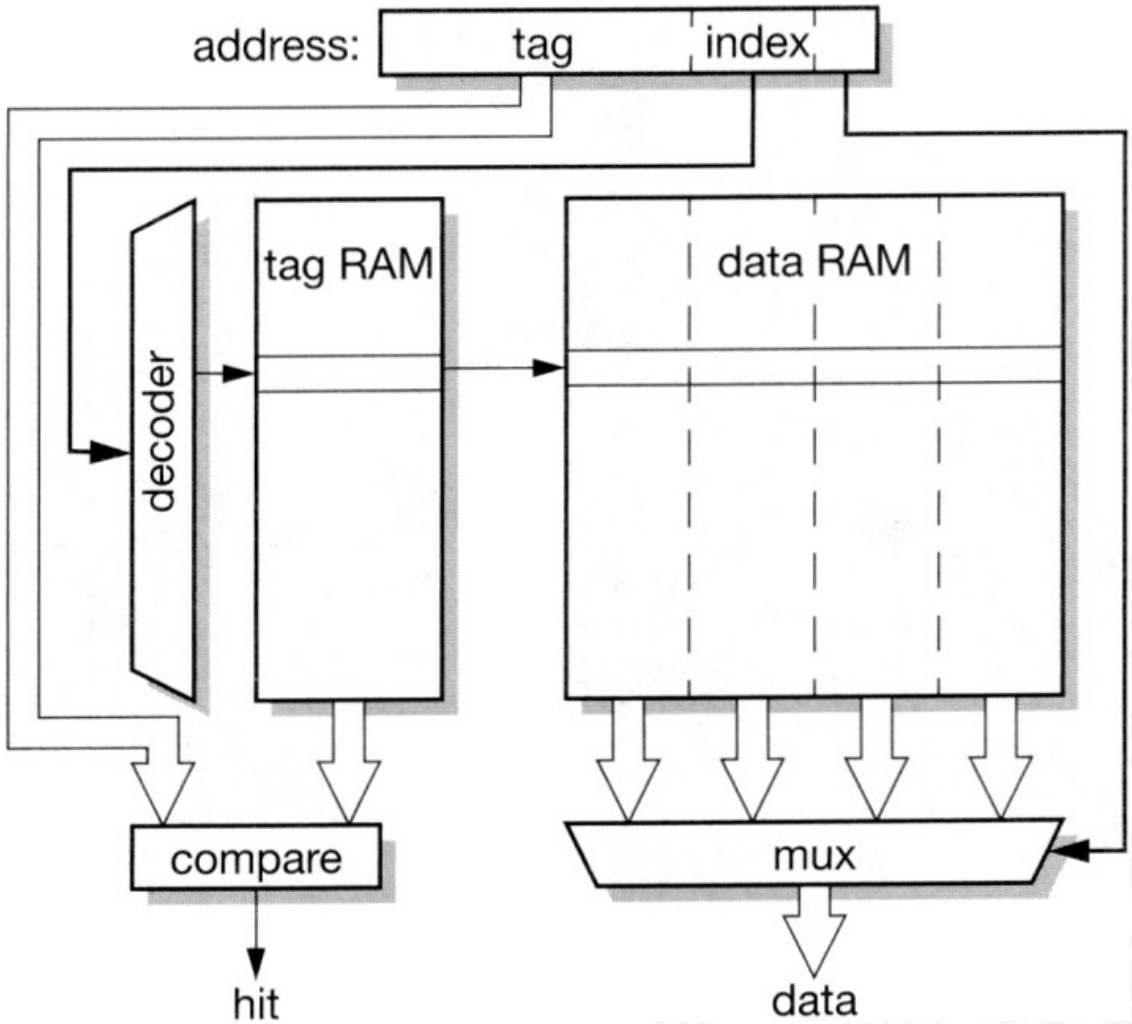
# Cache: direkt abgebildet / „direct mapped“ (cont.)

2. ⟨*valid*⟩: sind die Daten gültig?
3. „Line matching“: stimmt ⟨*tag*⟩ überein?
4. Wortselektion extrahiert Wort unter Offset ⟨*block offset*⟩



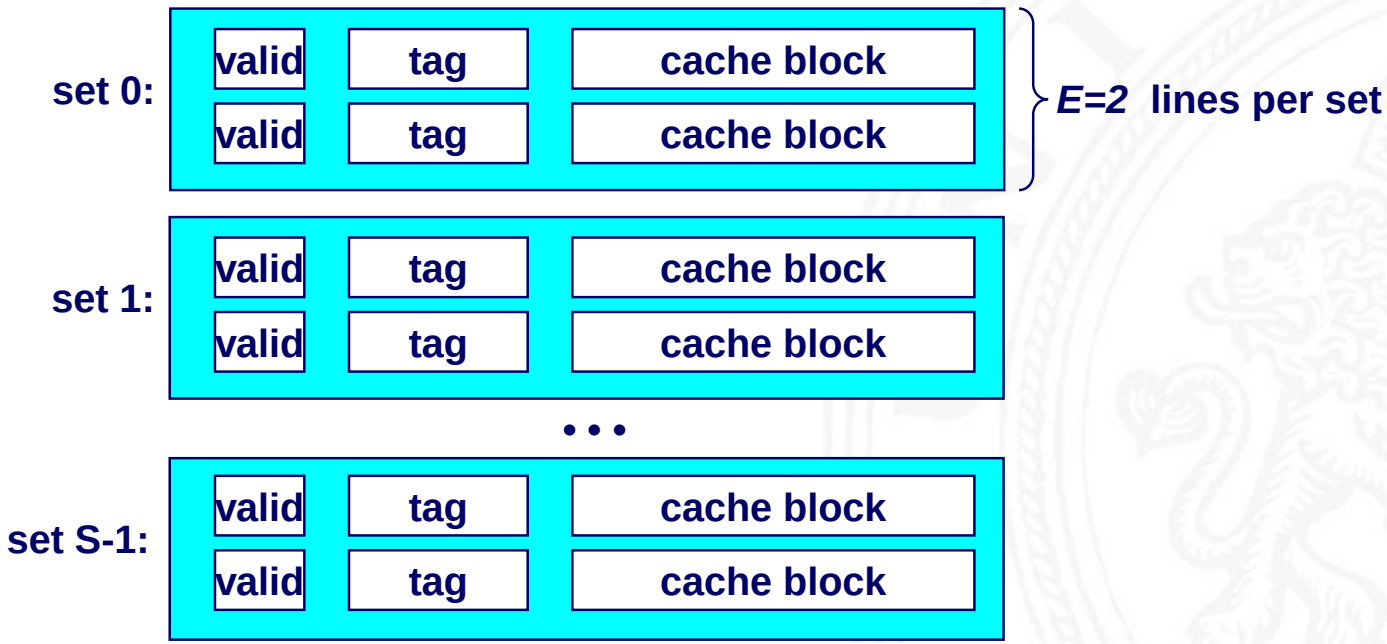
# Cache: direkt abgebildet / „direct mapped“ (cont.)

Prinzip



# Cache: bereichsassoziativ / „set associative“

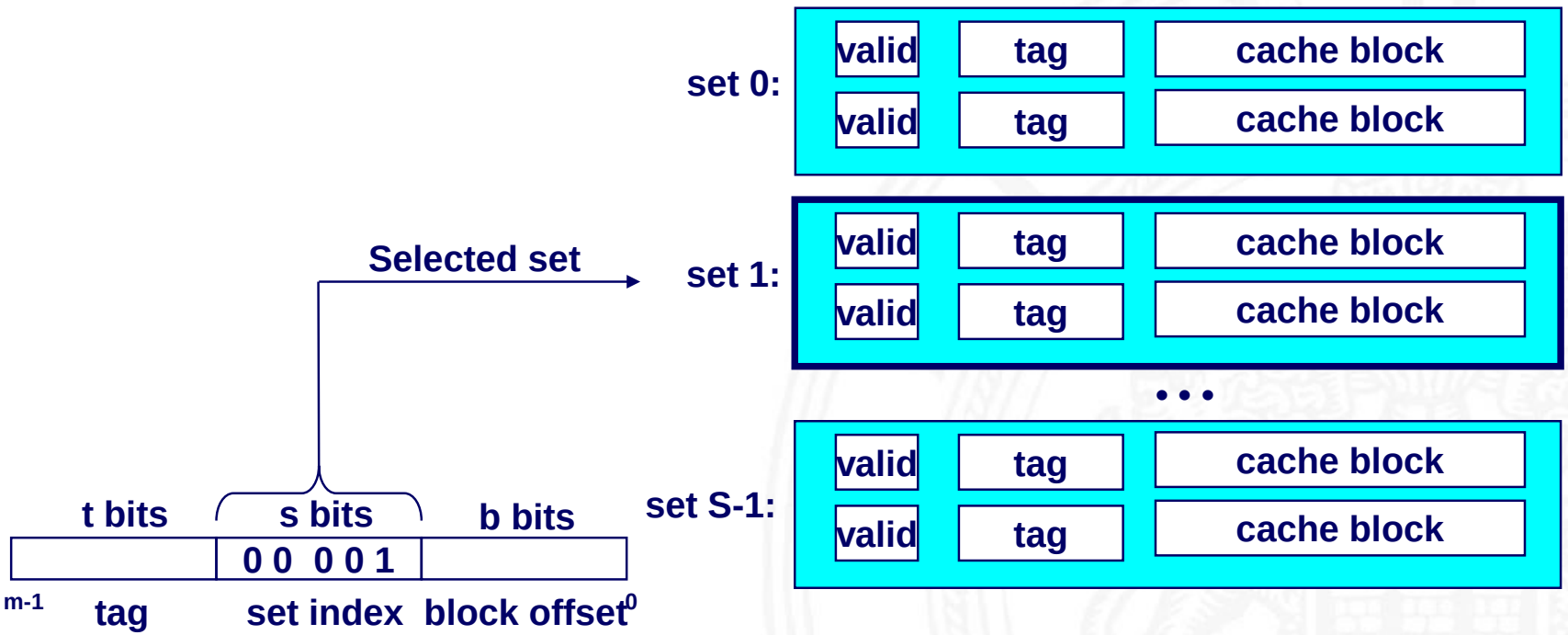
- jeder Speicheradresse ist ein Bereich $S$ mit mehreren ($E$) Cachezeilen zugeordnet
- n-fach assoziative Caches: E=2, 4. . .
  „2-way set associative cache“, „4-way. . . “



# Cache: bereichsassoziativ / „set associative“ (cont.)

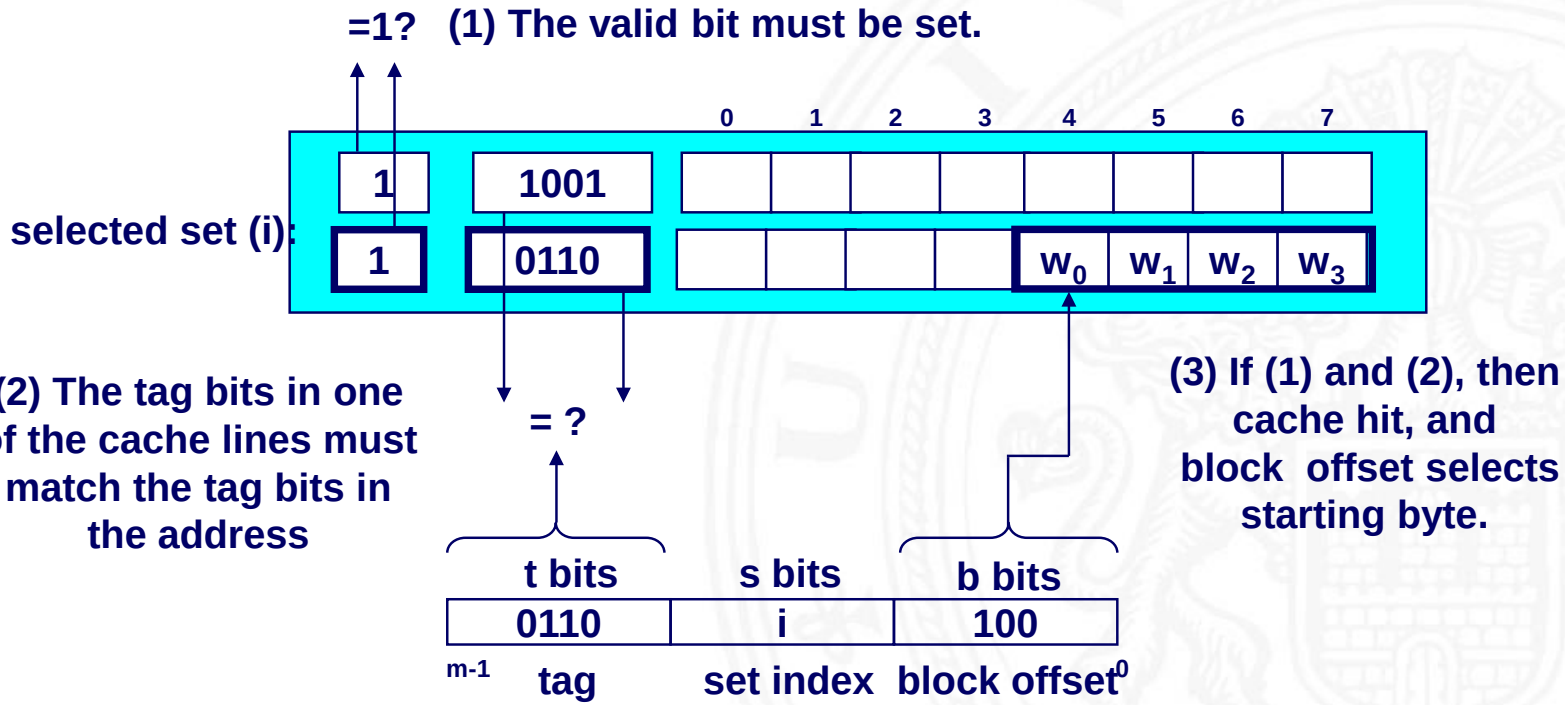
Zugriff auf n-fach assoziative Caches

1. Bereichsauswahl durch Bits ⟨*set index*⟩



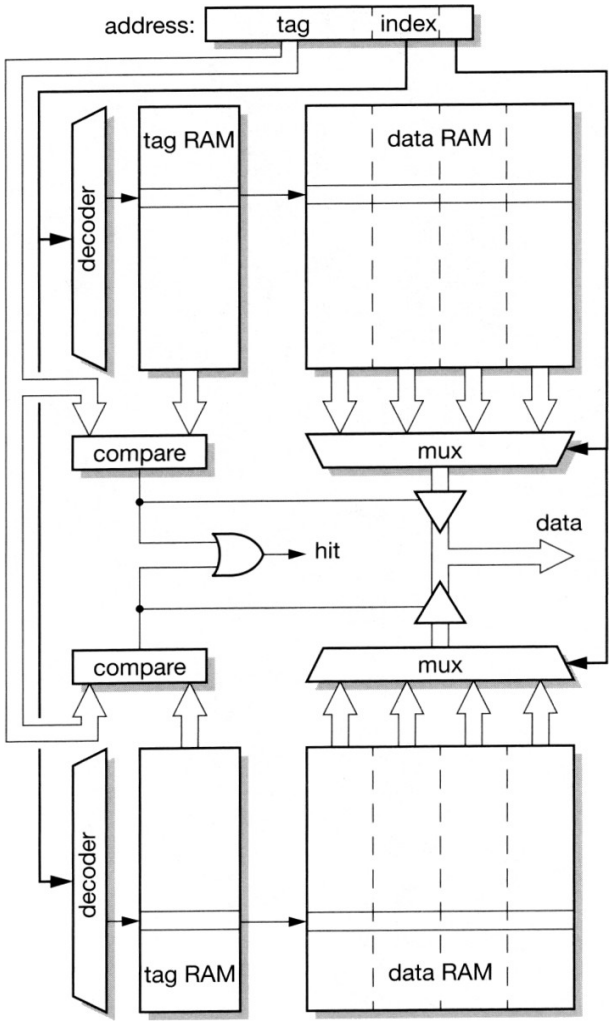
# Cache: bereichsassoziativ / „set associative“ (cont.)

2. „Line matching“: Cache-Zeile mit passendem ⟨*tag*⟩ finden? dazu Vergleich aller „tags“ des Bereichs ⟨*set index*⟩
3. ⟨*valid*⟩: sind die Daten gültig?
4. Wortselektion extrahiert Wort unter Offset ⟨*block offset*⟩



# Cache: bereichsassoziativ / „set associative“ (cont.)

Prinzip

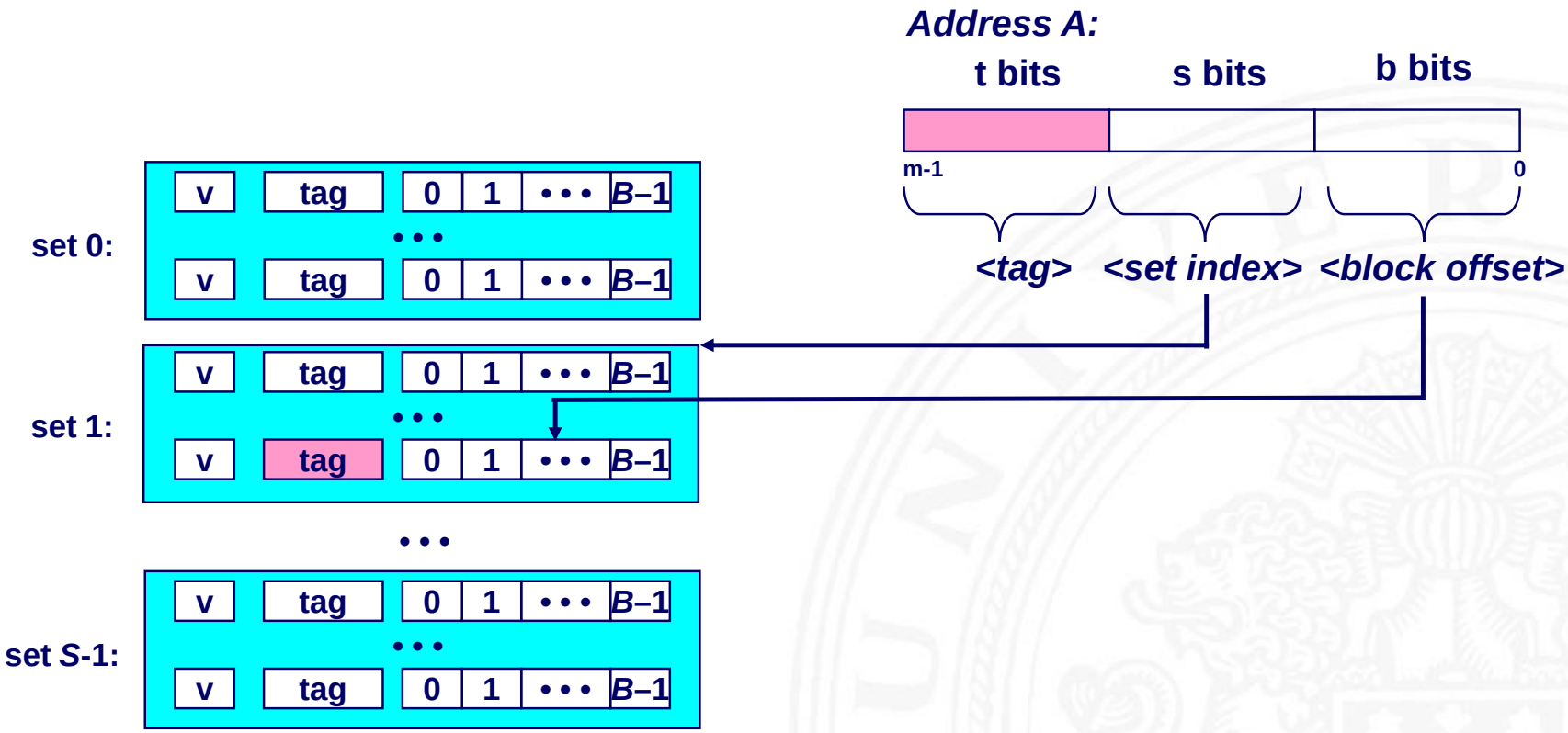


# Cache: voll-assoziativ

- jeder Adresse des Speichers kann jede beliebige Cachezeile zugeordnet werden
- Spezialfall: nur ein Cachebereich $S$

– benötigt $E$-Vergleicher
– nur für sehr kleine Caches realisierbar

# Cache – Dimensionierung




- Parameter: $S$, $B$, $E$

# Cache – Dimensionierung (cont.)

- Cache speichert immer größere Blöcke / „*Cache-Line*“
- Wortauswahl durch ⟨*block offset*⟩ in Adresse

\+ nutzt räumliche Lokalität aus

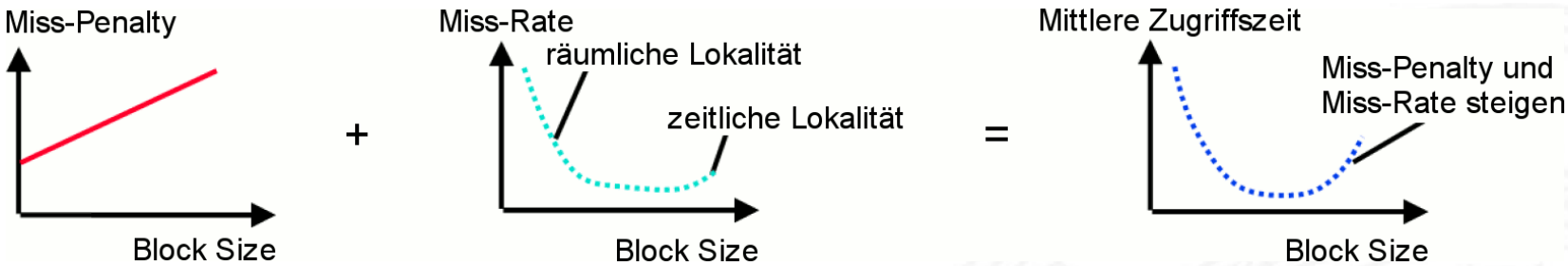
\+ Breite externe Datenbusse

\+ nutzt Burst-Adressierung des Speichers: Adresse nur für erstes Wort vorgeben, dann automatisches Inkrement

\+ kürzere interne Cache-Adressen

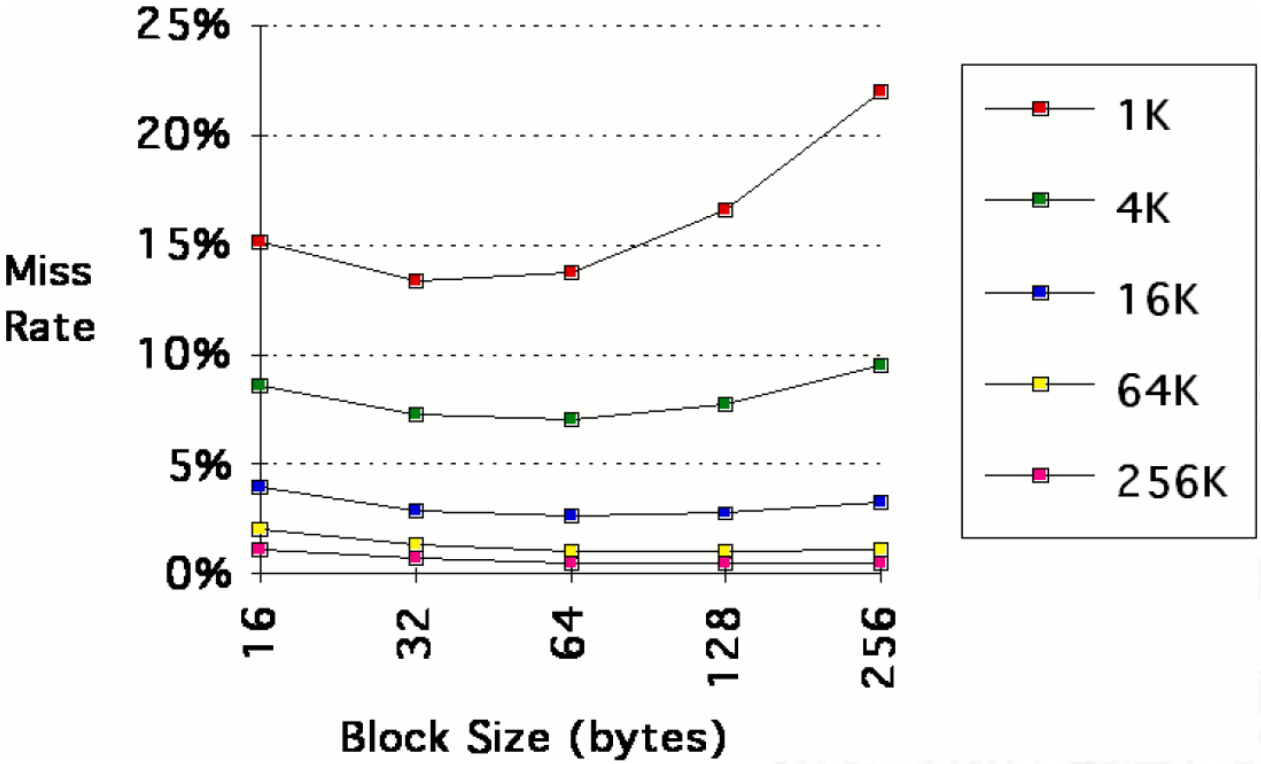
# Cache – Dimensionierung (cont.)

## Cache- und Block-Dimensionierung




- Blockgröße klein, viele Blöcke
  - \+ kleinere Miss-Penalty
  - \+ temporale Lokalität
  - – räumliche Lokalität
- Blockgröße groß, wenig Blöcke
  - – größere Miss-Penalty
  - – temporale Lokalität
  - \+ räumliche Lokalität

# Cache – Dimensionierung (cont.)




- Block-Size: 32...128 Byte
  L1-Cache: 4...256 KByte
  L2-Cache: 256...4096 KByte

# Cache – Dimensionierung (cont.)

- zusätzliche Software-Optimierungen
  - Ziel: Cache Misses reduzieren
  - Schleifen umsortieren, verschmelzen...
  - Datenstrukturen zusammenfassen...

# Cache Ersetzungsstrategie

*Wenn der Cache gefüllt ist, welches Datum wird entfernt?*

- zufällige Auswahl
- LRU (**L**east **R**ecently **U**sed):
  der „älteste“ nicht benutzte Cache Eintrag
  - echtes LRU als Warteschlange realisiert
  - Pseudo LRU mit baumartiger Verwaltungsstruktur:
    Zugriff wird paarweise mit einem Bit markiert,
    die Paare wieder zusammengefasst usw.
- LFU (**L**east **F**requently **U**sed):
  der am seltensten benutzte Cache Eintrag
  - durch Zugriffszähler implementiert

# Cache Schreibstrategie

*Wann werden modifizierte Daten des Cache zurückgeschrieben?*

- Write-Through: beim Schreiben werden Daten sofort im Cache und im Hauptspeicher modifiziert
  - \+ andere Bus-Master sehen immer den „richtigen“ Speicherinhalt: *Cache-Kohärenz*
  - – Werte werden unnötig oft in Speicher zurückgeschrieben
- Write-Back: erst in den Speicher schreiben, wenn Datum des Cache ersetzt werden würde
  - \+ häufig genutzte Werte (z.B. lokale Variablen) werden nur im Cache modifiziert
  - – Cache-Kohärenz ist nicht gegeben
  - ⇒ spezielle Befehle für „Cache-Flush“
  - ⇒ „non-cacheable“ Speicherbereiche

# Cache Kohärenz

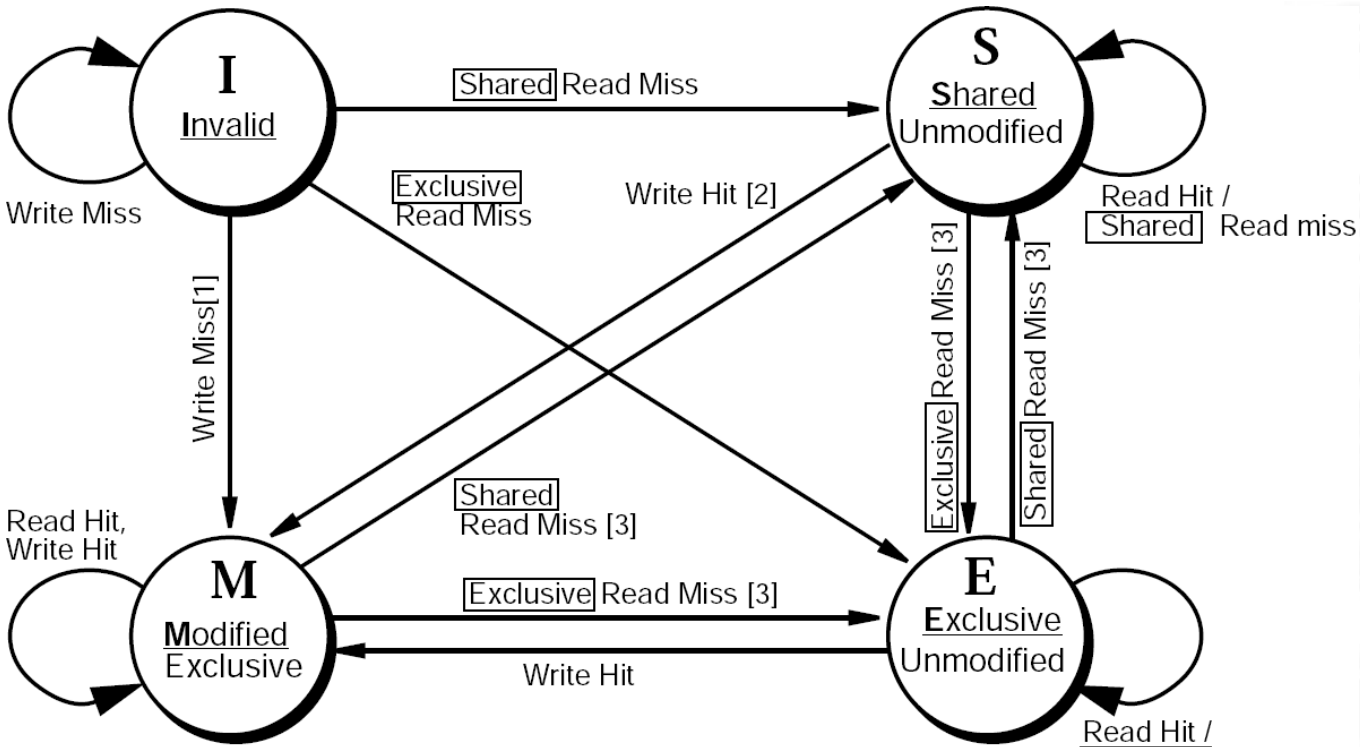
- Daten zwischen Cache und Speicher konsistent halten
- notwendig wenn auch andere Einheiten (Bus-Master) auf Speicher zugreifen können (oder Mehrprozessorsysteme!)
- Harvard-Architektur hat getrennte Daten- und Instruktions-Speicher: einfacherer Instruktions-Cache
  - Instruktionen sind read-only
  - kein Cache-Kohärenz Problem
- „Snooping“
  - Cache „lauscht“ am Speicherbus
  - externer Schreibzugriff $\Rightarrow$ Cache aktualisieren / ungültig machen
  - externer Lesezugriff $\Rightarrow$ Cache liefert Daten

# Cache Kohärenz (cont.)

- MESI Protokoll
  - besitzt zwei Statusbits für die Protokollzustände
  - **M**odified: Inhalte der Cache-Line wurden modifiziert
  - **E**xclusive unmodified: Cache-Line „gehört“ dieser CPU, nicht modifiz.
  - **S**hared unmodified: Inhalte sind in mehreren Caches vorhanden
  - **I**nvalid: ungültiger Inhalt, Initialzustand

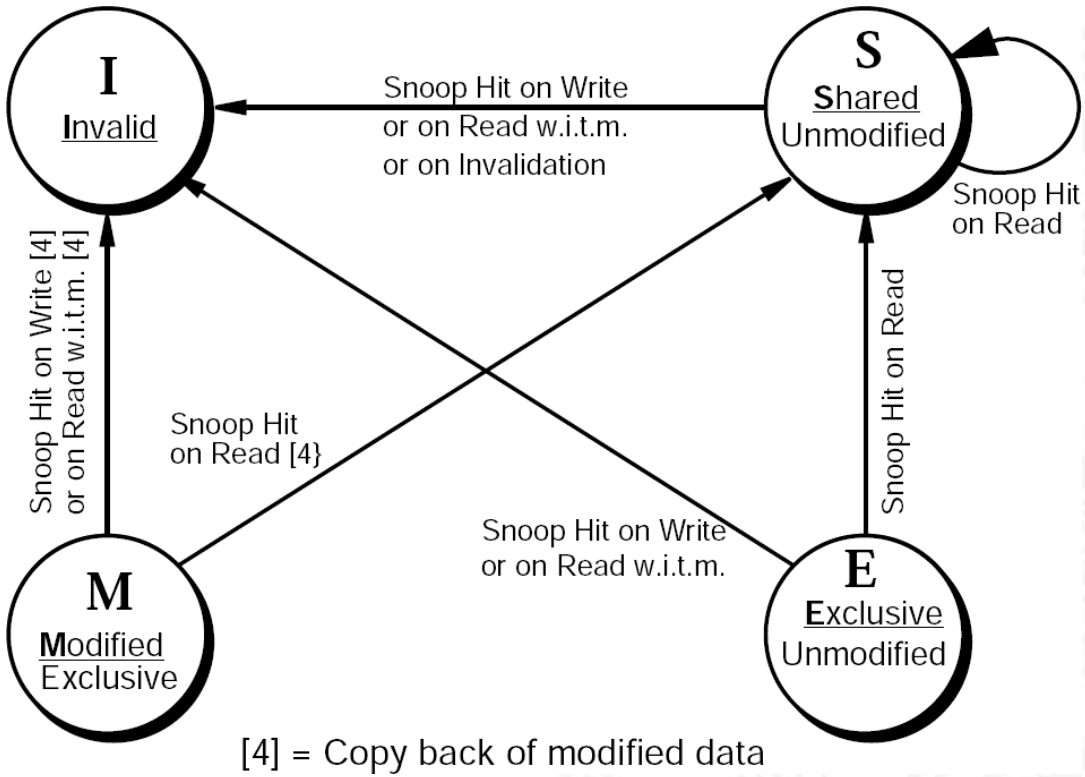
# Cache Kohärenz (cont.)

- Zustandsübergänge für „Bus Master“ CPU



# Cache Kohärenz (cont.)

- Zustandsübergänge für "Snooping" CPU



# Optimierung der Cachezugriffe

- Mittlere Speicherzugriffszeit $= T_{Hit} + R_{Miss} \cdot T_{Miss}$

$\Rightarrow$ Verbesserung der Cache Performanz durch kleinere $T_{Miss}$ am einfachsten zu realisieren

- mehrere Cache Ebenen
- Critical Word First: bei großen Cache Blöcken (mehrere Worte) gefordertes Wort zuerst holen und gleich weiterleiten
- Read-Miss hat Priorität gegenüber Write-Miss
  $\Rightarrow$ Zwischenspeicher für Schreiboperationen (Write Buffer)
- Merging Write Buffer: aufeinanderfolgende Schreiboperationen zwischenspeichern und zusammenfassen
- Victim Cache: kleiner voll-assoziativer Cache zwischen direct-mapped Cache und nächster Ebene „sammelt“ verdrängte Cache Einträge

# Optimierung der Cachezugriffe (cont.)

⇒ Verbesserung der Cache Performanz durch kleinere $R_{Miss}$
- größere Caches (– mehr Hardware)
- höhere Assoziativität (– langsamer)

⇒ Optimierungstechniken
- Software Optimierungen
- Prefetch: Hardware (Stream Buffer)
  Software (Prefetch Operationen)
- Cache Zugriffe in Pipeline verarbeiten
- Trace Cache: im Instruktions-Cache werden keine Speicherinhalte, sondern ausgeführte Sequenzen (*trace*) einschließlich ausgeführter Sprünge gespeichert

  Beispiel: NetBurst Architektur (Pentium 4)

# Chiplayout

## ARM7 / ARM10

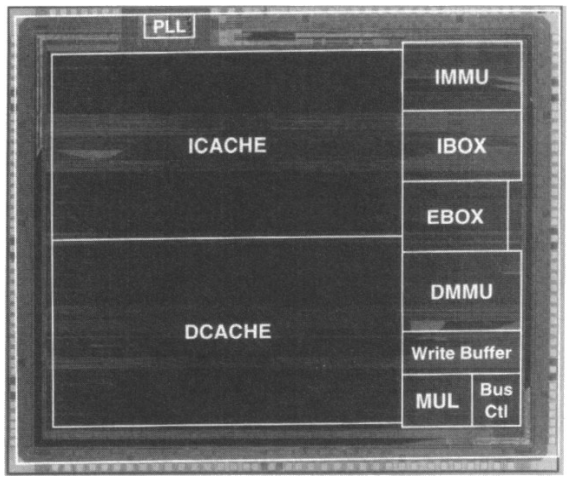
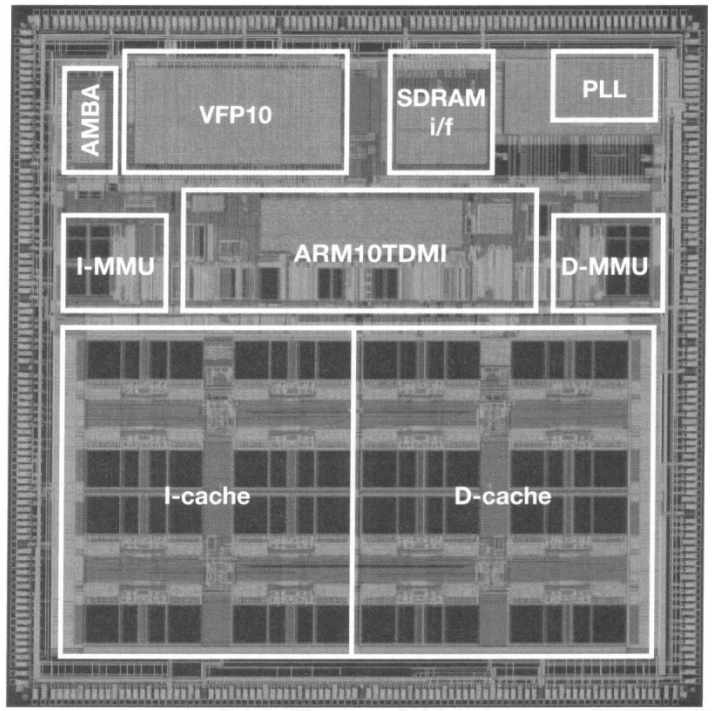


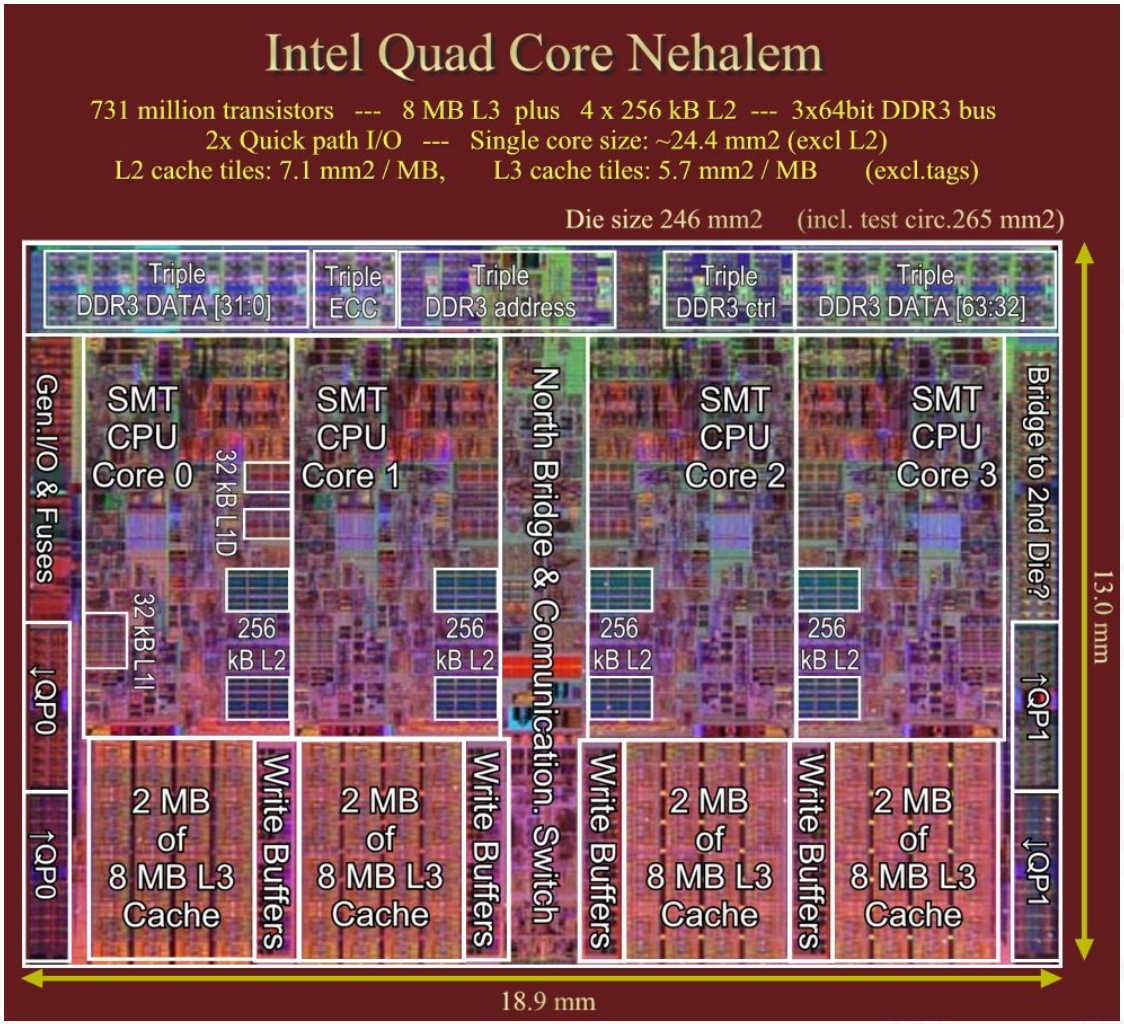

Photo courtesy of Intel Corp.



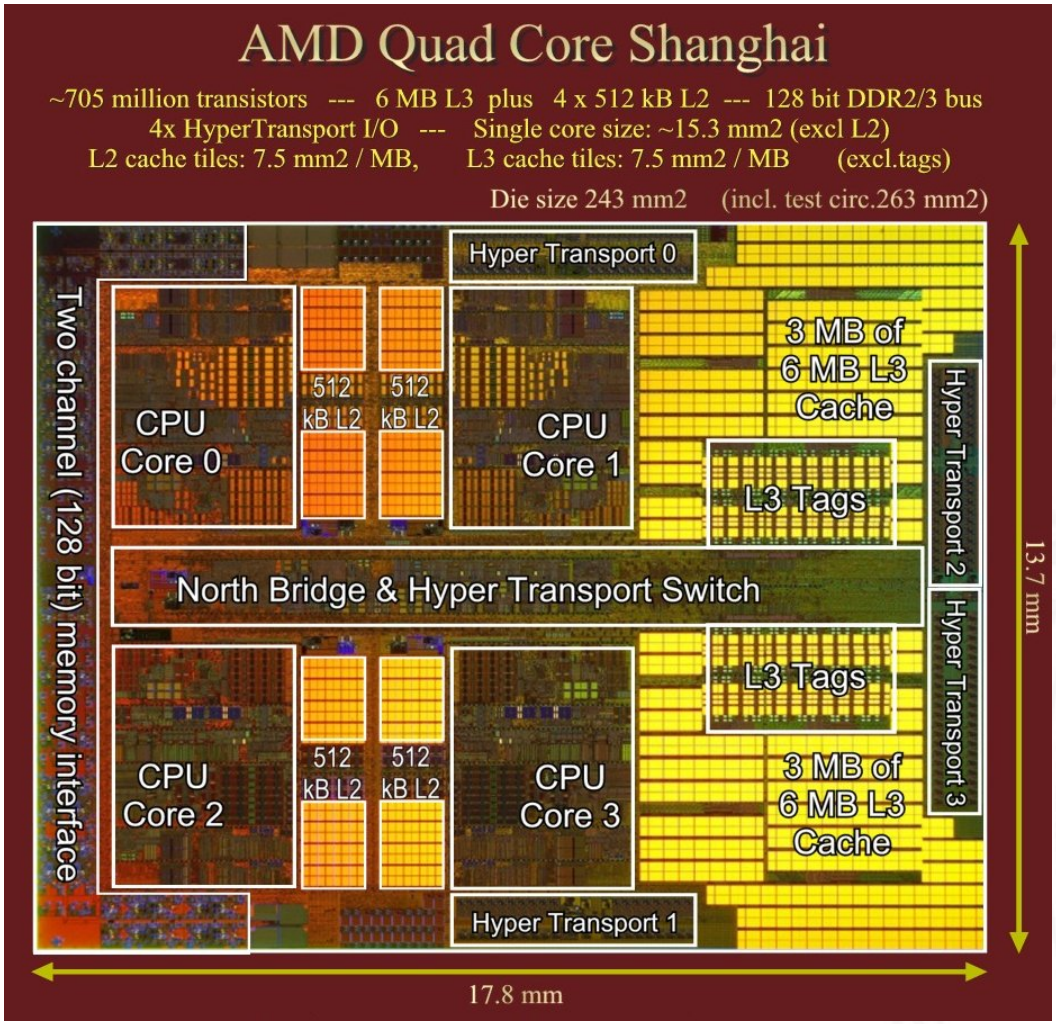

© ARM Limited

- IBOX: Steuerwerk (instruction fetch und decode)
  EBOX: Operationswerk, ALU, Register (execute)
  IMMU/DMMU: Virtueller Speicher (instruction/data TLBs)
  ICACHE: Instruction Cache
  DCACHE: Data Cache

# Chiplayout (cont.)



# Chiplayout (cont.)



# Cache vs. Programmcode

Programmierer kann für maximale Cacheleistung optimieren

- ▷ Datenstrukturen werden fortlaufend alloziert

1. durch entsprechende Organisation der Datenstrukturen
2. durch Steuerung des Zugriffs auf die Daten
   - Geschachtelte Schleifenstruktur
   - Blockbildung ist eine übliche Technik

Systeme bevorzugen einen *Cache-freundlichen* Code

- Erreichen der optimalen Leistung ist plattformspezifisch
  - Cachegrößen, Zeilengrößen, Assoziativität etc.
- generelle Empfehlungen
  - „working set“ klein ⇒ zeitliche Lokalität
  - kleine Adressfortschaltungen („strides“) ⇒ räumliche Lokalität

# Virtueller Speicher – Motivation

Speicher-Paradigmen

- Programmierer
  - ein großer Adressraum
  - linear adressierbar
- Betriebssystem
  - eine Menge laufender Tasks / Prozesse
  - read-only Instruktionen
  - read-write Daten

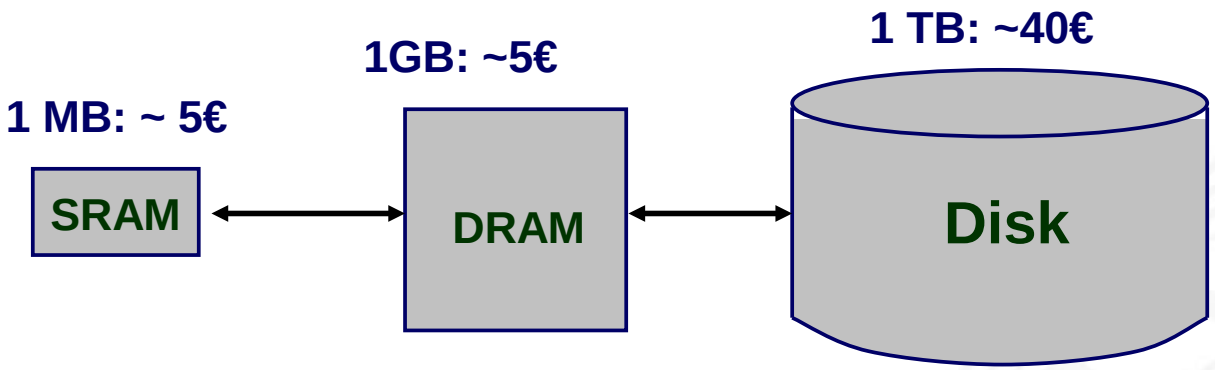
Konsequenz

**virtueller Speicher** umfasst drei Aspekte der (teilweise) widersprüchlichen Anforderungen an „Speicher“

# Virtueller Speicher – Motivation (cont.)

1. Benutzung der Festplatte als *zusätzlichen* Hauptspeicher
   - Prozessadressraum kann physikalische Speichergröße übersteigen
   - Summe der Adressräume mehrerer Prozesse kann physikalischen Speicher übersteigen
2. Vereinfachung der Speicherverwaltung
   - viele Prozesse liegen im Hauptspeicher
     - jeder Prozess mit seinem eigenen Adressraum (0...n)
   - nur *aktiver* Code und Daten sind tatsächlich im Speicher
     - bedarfsabhängige, dynamische Speicherzuteilung
3. Bereitstellung von Schutzmechanismen
   - ein Prozess kann einem anderen nicht beeinflussen
     - sie operieren in verschiedenen Adressräumen
   - Benutzerprozess hat keinen Zugriff auf privilegierte Informationen
     - jeder virtuelle Adressraum hat eigene Zugriffsrechte

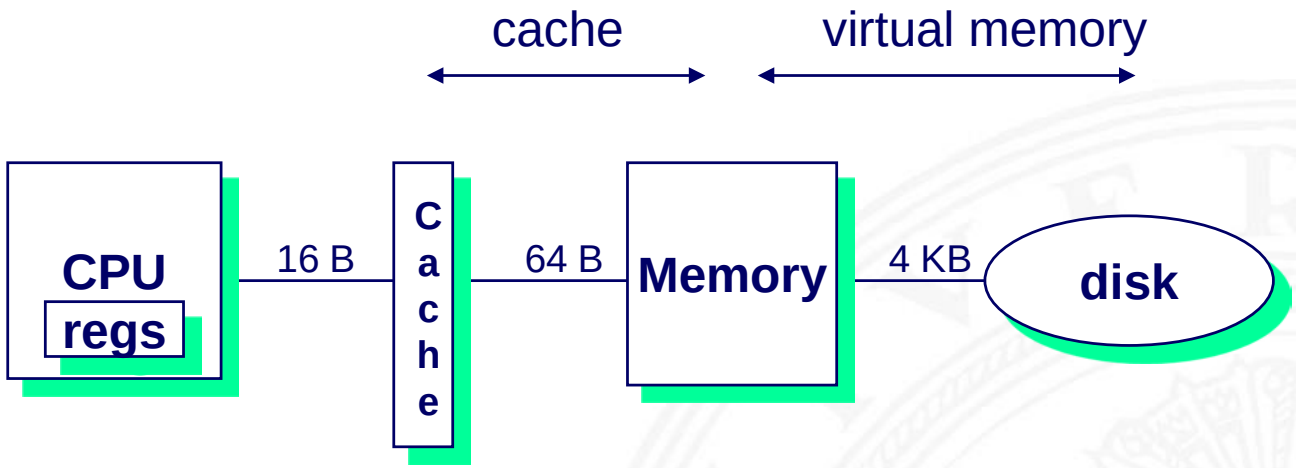
# Festplatte „erweitert“ Hauptspeicher




- Vollständiger Adressraum zu groß ⇒ DRAM ist *Cache*
  - 32-bit Adressen: $\approx 4 \times 10^9$ Byte 4 Milliarden
  - 64-bit Adressen: $\approx 16 \times 10^{16}$ Byte 16 Quintillionen
- Speichern auf Festplatte ist $\approx 125\times$ billiger als im DRAM
  - 1 TiB DRAM: $\approx 5\,000\,€$
  - 1 TiB Festplatte: $\approx 40\,€$

⇒ kostengünstiger Zugriff auf große Datenmengen

# Ebenen in der Speicherhierarchie

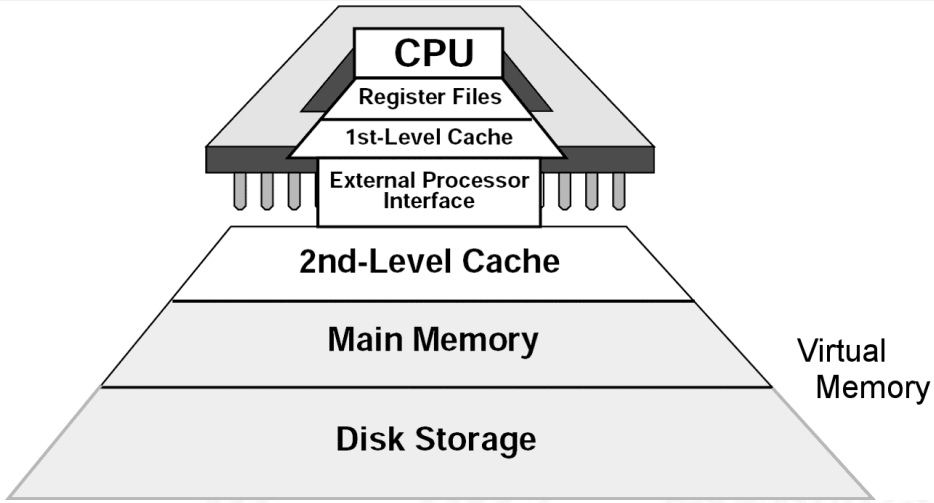



| | Register | Cache | Memory | Disk Memory |
|---|---|---|---|---|
| size: | 64 B | 32 KB-12MB | 8 GB | 2 TB |
| speed: | 300 ps | 1 ns | 8 ns | 4 ms |
| $/Mbyte: | | 5€/MB | 5€/GB | 4 Ct./GB |
| line size: | 16 B | 64 B | 4 KB | |

larger, slower, cheaper

# Ebenen in der Speicherhierarchie (cont.)

- Hauptspeicher als Cache für den Plattenspeicher

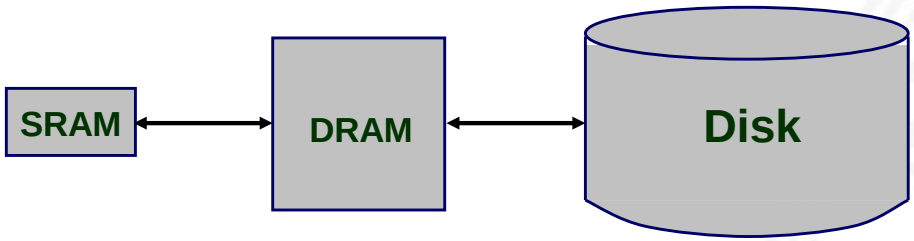



- Parameter der Speicherhierarchie

| | 1st-Level Cache | virtueller Speicher |
|---|---|---|
| Blockgröße | 16-128 Byte | 4-64 kByte |
| Hit-Dauer | 1-2 Zyklen | 40-100 Zyklen |
| Miss Penalty | 8-100 Zyklen | 70.000-6.000.000 Zyklen |
| Miss Rate | 0,5-10 % | 0,00001-0,001 % |
| Speichergröße | 8-64 kByte | 16-8192 MByte |

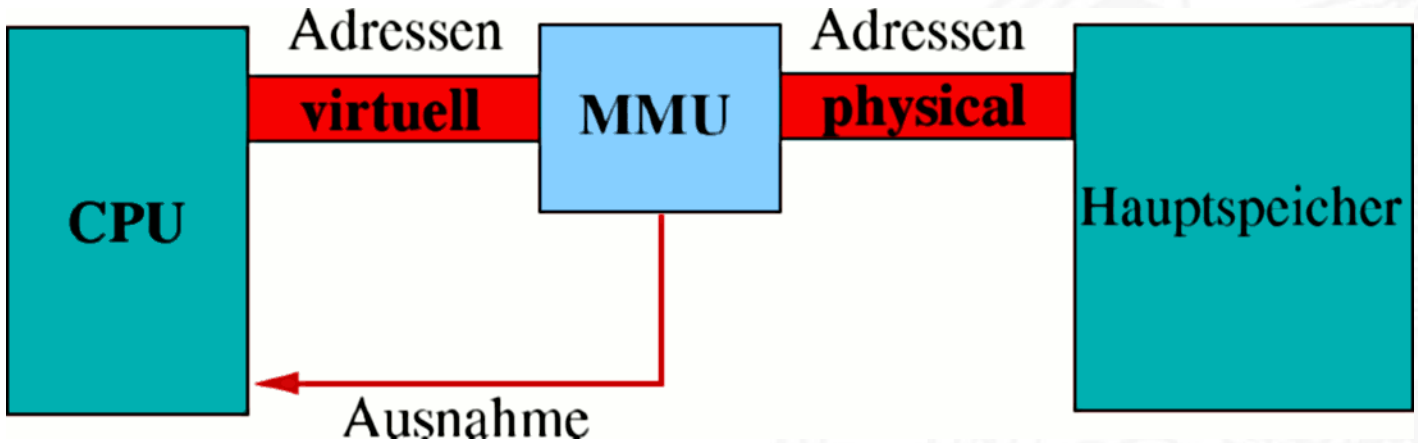
# Ebenen in der Speicherhierarchie (cont.)

- DRAM vs. Festplatte ist extremer als SRAM vs. DRAM
  - Zugriffswartezeiten
    - DRAM $\approx 10\times$ langsamer als SRAM
    - Festplatte $\approx 500\,000\times$ langsamer als DRAM
  - $\Rightarrow$ Nutzung der räumlichen Lokalität wichtig
    - erstes Byte $\approx 500\,000\times$ langsamer als nachfolgende Bytes



# Prinzip des virtuellen Speichers

- jeder Prozess besitzt seinen eigenen virtuellen Adressraum
- Kombination aus Betriebssystem und Hardwareeinheiten
- MMU – **M**emory **M**anagement **U**nit




  - Umsetzung von virtuellen zu physikalischen Adressen, Programm-Relokation

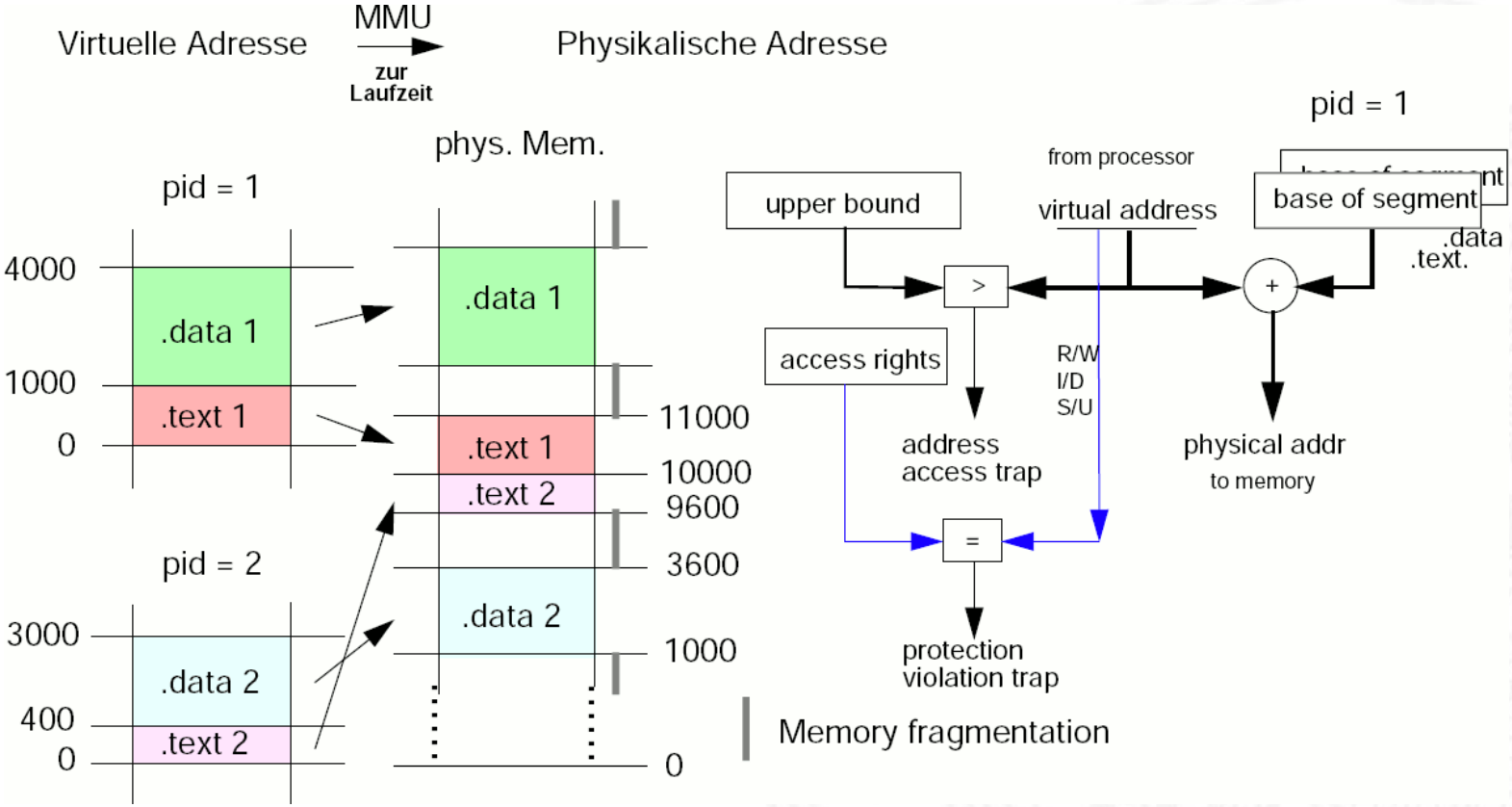
# Prinzip des virtuellen Speichers (cont.)

- Umsetzungstabellen werden vom Betriebssystem verwaltet
- wegen des Speicherbedarfs der Tabellen beziehen sich diese auf größere Speicherblöcke (*Segmente* oder *Seiten*)

- Umgesetzt wird nur die Anfangsadresse, der Offset innerhalb des Blocks bleibt unverändert
- Blöcke dieses virtuellen Adressraums können durch Betriebssystem auf Festplatte ausgelagert werden
  - Windows: Auslagerungsdatei
  - Unix/Linux: swap Partition und -Datei(en)
- Konzepte zur Implementation virtuellen Speichers
  - *Segmentierung*
  - Speicherzuordnung durch *Seiten* („*Paging*“)
  - gemischte Ansätze (Standard bei: Desktops, Workstations. . . )

# Virtueller Speicher: Segmentierung

- Unterteilung des Adressraums in kontinuierliche Bereiche *variabler* Größe



# Virtueller Speicher: Segmentierung (cont.)

- Idee: Trennung von Instruktionen, Daten und Stack

$\Rightarrow$ Abbildung von *Programmen* in den *Hauptspeicher*

$+$ Inhalt der Segmente: logisch zusammengehörige Daten

$+$ getrennte Zugriffsrechte, Speicherschutz

$+$ exakte Prüfung der Segmentgrenzen

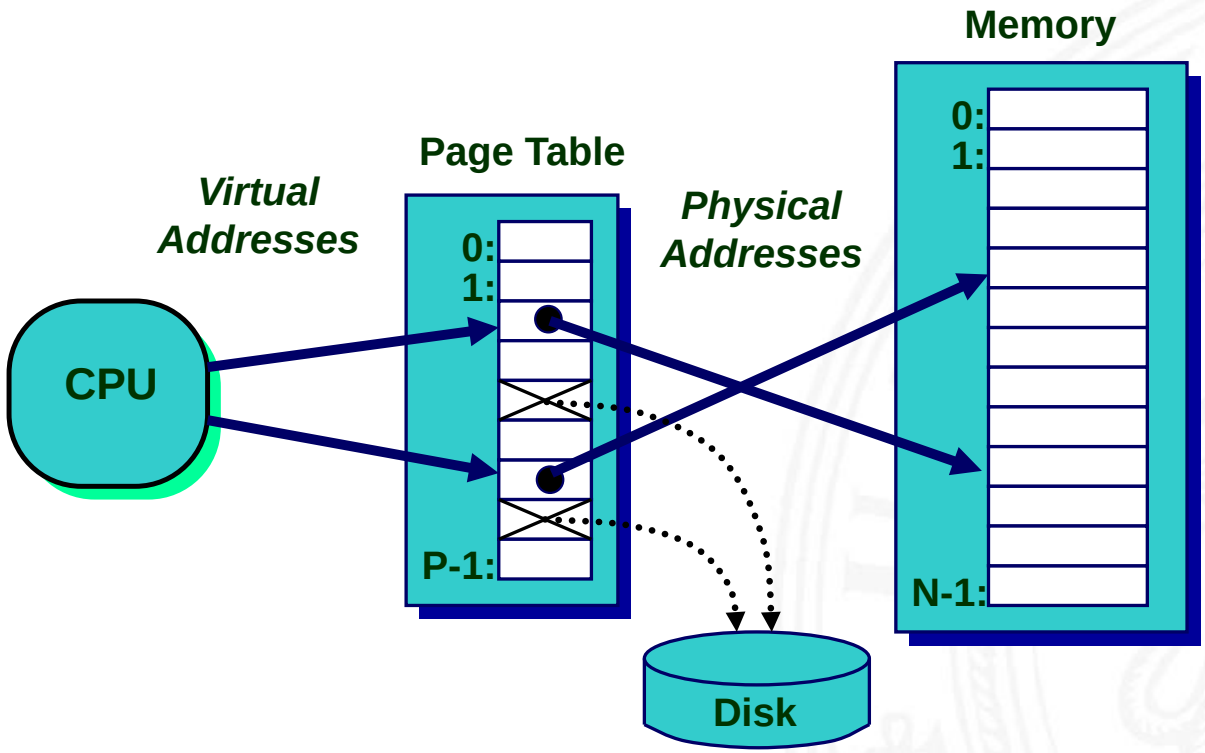
$-$ Segmente könne sehr groß werden

$-$ Ein- und Auslagern von Segmenten kann sehr lange dauern

$-$ „Verschnitt“, **Memory Fragmentation**

# Virtueller Speicher: Paging / Seitenadressierung

- Unterteilung des Adressraums in Blöcke *fester* Größe = Seiten
  Abbildung auf Hauptspeicherblöcke = Kacheln

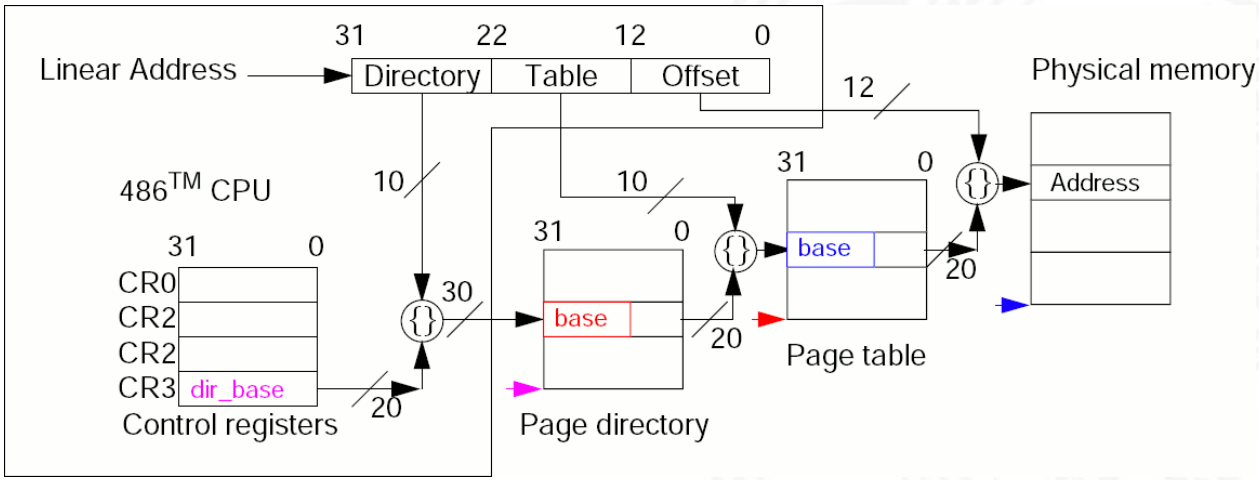


# Virtueller Speicher: Paging / Seitenadressierung (cont.)

- ⇒ Abbildung von *Adressen* in den *virtuellen Speicher*
- \+ Programme können größer als der Hauptspeicher sein
- \+ Programme können an beliebige physikalischen Adressen geladen werden, unabhängig von der Aufteilung des physikalischen Speichers
- \+ feste Seitengröße: einfache Verwaltung in Hardware
- \+ Zugriffsrechte für jede Seite (read/write, User/Supervisor)
- \+ gemeinsam genutzte Programmteile/-Bibliotheken können sehr einfach in das Konzept integriert werden
  - Windows: `.dll`-Dateien
  - Unix/Linux: `.so`-Dateien

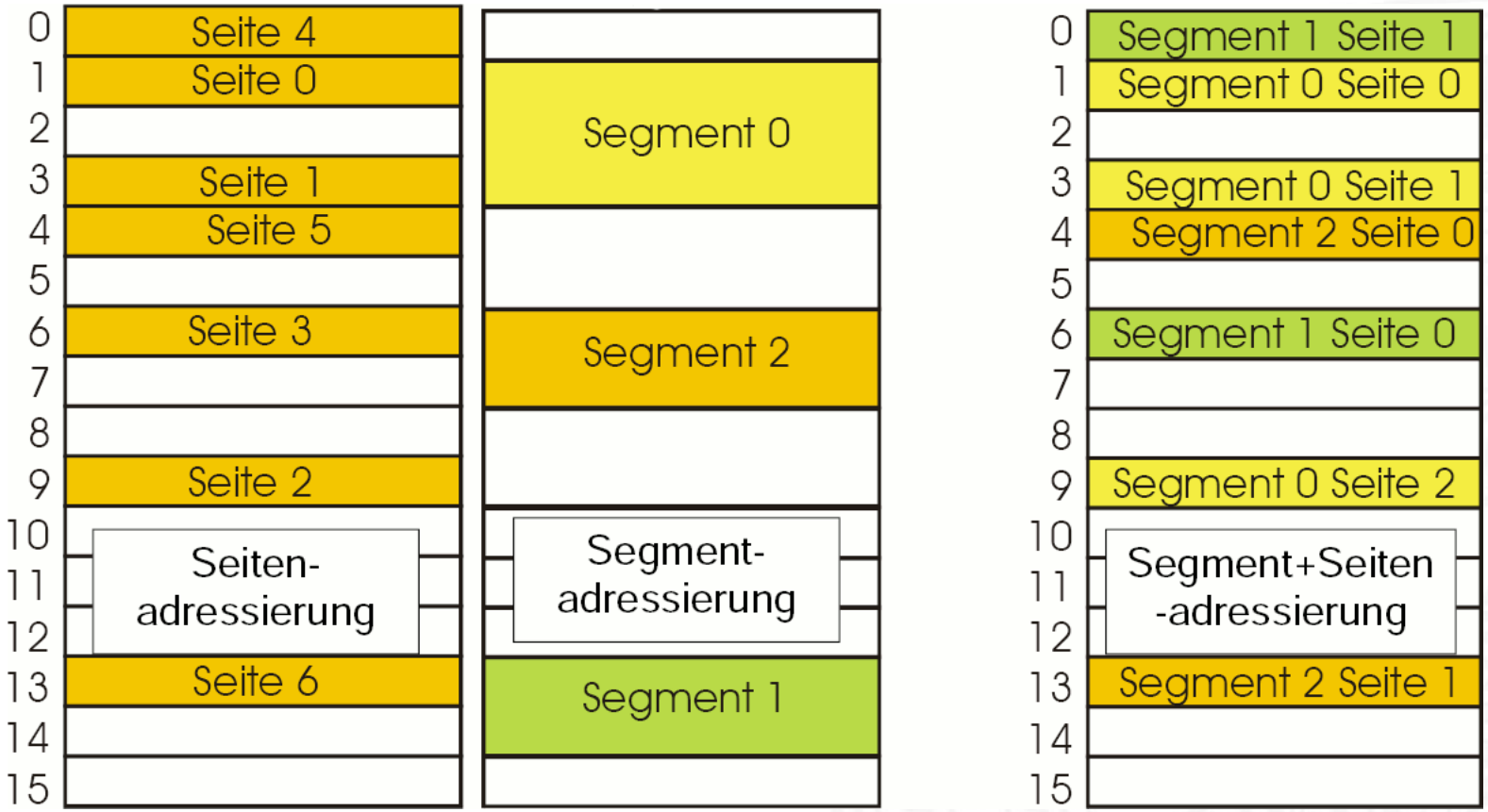
# Virtueller Speicher: Paging / Seitenadressierung (cont.)

- große Miss-Penalty (Nachladen von der Platte)
  - ⇒ Seiten sollten relativ groß sein: 4 oder 8 kByte
- Speicherplatzbedarf der Seitentabelle
  viel virtueller Speicher, 4 kByte Seitengröße
  - = sehr große Pagetable
  - ⇒ Hash-Verfahren (*inverted page tables*)
  - ⇒ mehrstufige Verfahren



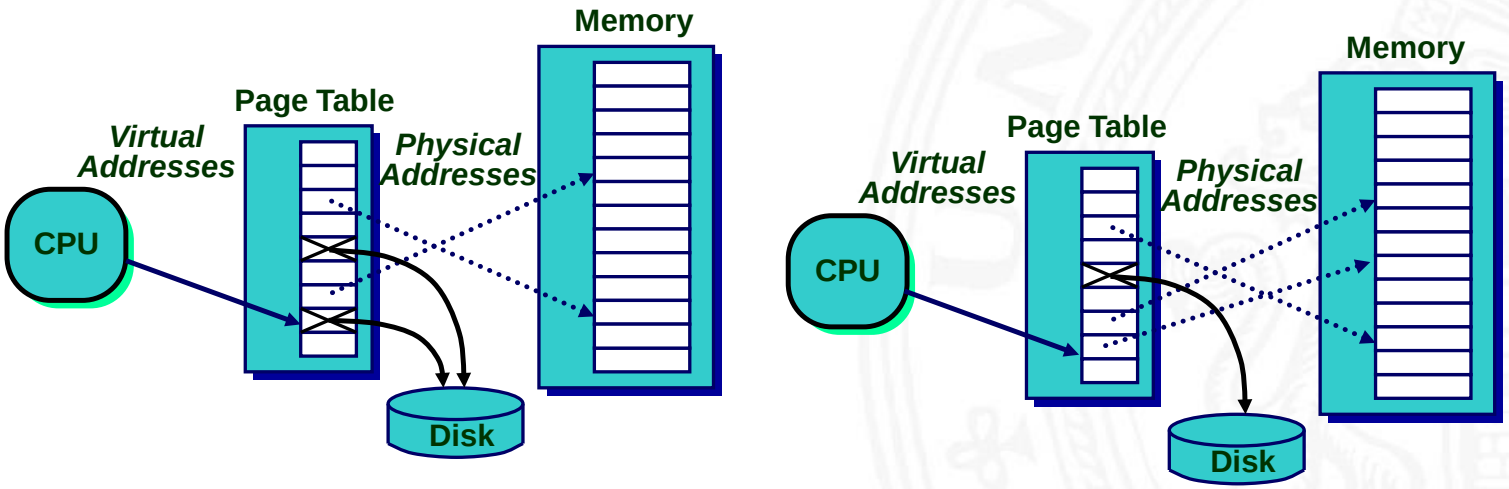
# Virtueller Speicher: Segmentierung + Paging

aktuell = Mischung: Segmentierung und Paging (seit I386)



# Seitenfehler

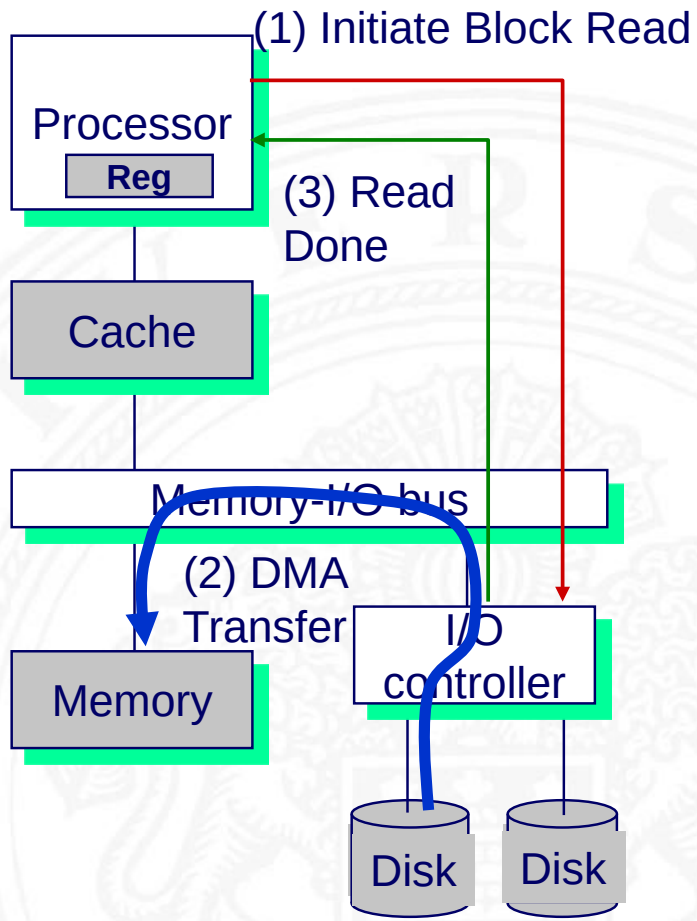
- Seiten-Tabelleneintrag: Startadresse der virt. Seite auf Platte
- Daten von Festplatte in Speicher laden:
  Aufruf des „Exception handler“ des Betriebssystems
  - laufender Prozess wird unterbrochen, andere können weiterlaufen
  - Betriebssystem kontrolliert die Platzierung der neuen Seite im Hauptspeicher (Ersetzungsstrategien) etc.



# Seitenfehler (cont.)

Behandlung des Seitenfehlers

1. Prozessor signalisiert DMA-Controller
   - lies Block der Länge $P$ ab Festplattenadresse $X$
   - speichere Daten ab Adresse $Y$ in Hauptspeicher
2. Lesezugriff erfolgt als
   - Direct Memory Access (DMA)
   - Kontrolle durch I/O Controller
3. I/O Controller meldet Abschluss
   - Gibt Interrupt an den Prozessor
   - Betriebssystem lässt unterbrochenen Prozess weiterlaufen

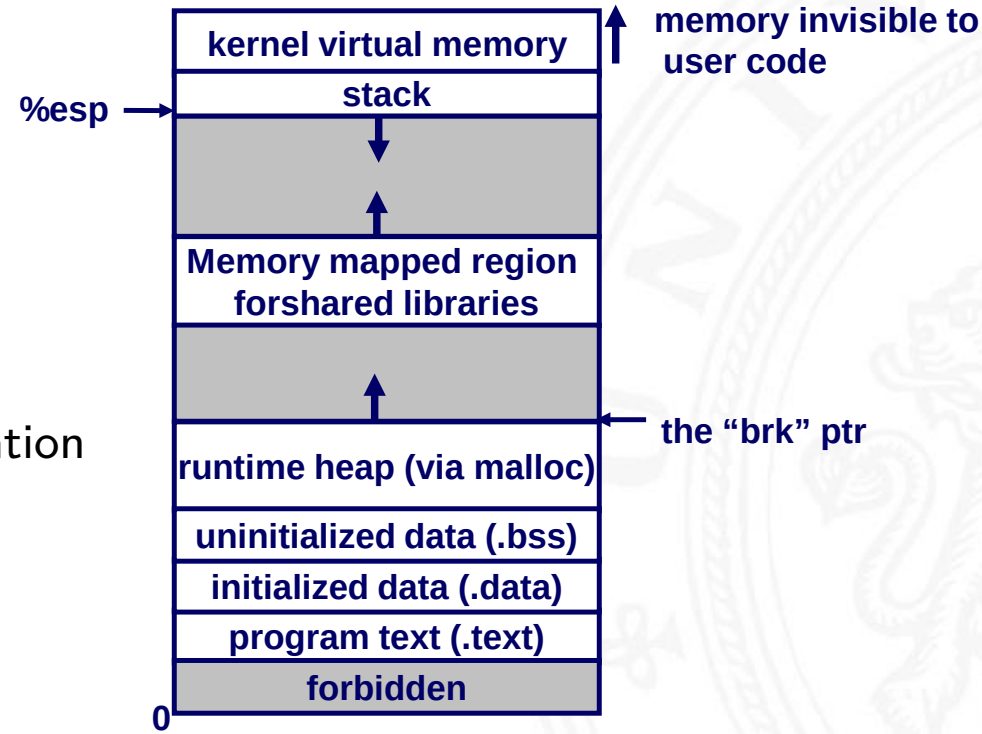


# Separate virtuelle Adressräume

Mehrere Prozesse können im physikalischen Speicher liegen

- Wie werden Adresskonflikte gelöst?
- Was passiert, wenn Prozesse auf dieselbe Adresse zugreifen?

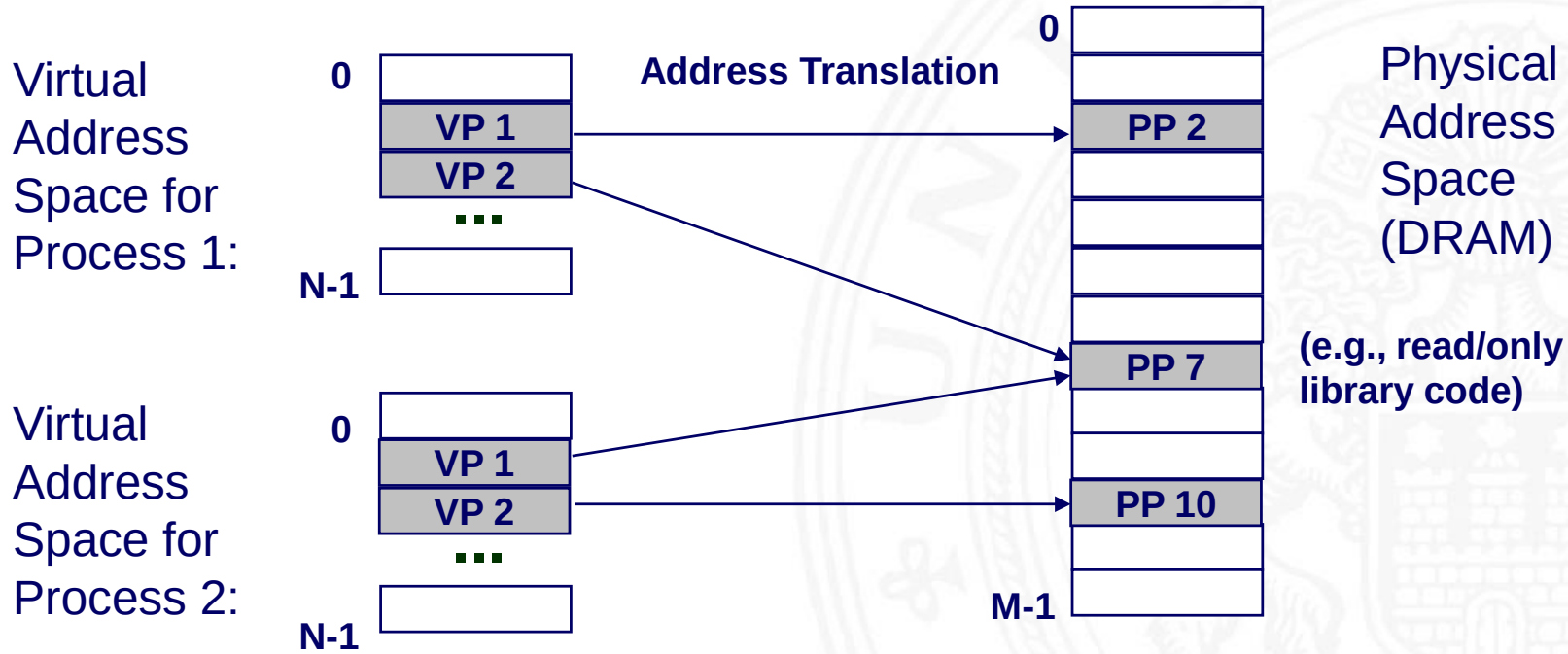
Linux x86
Speicherorganisation



# Separate virtuelle Adressräume (cont.)

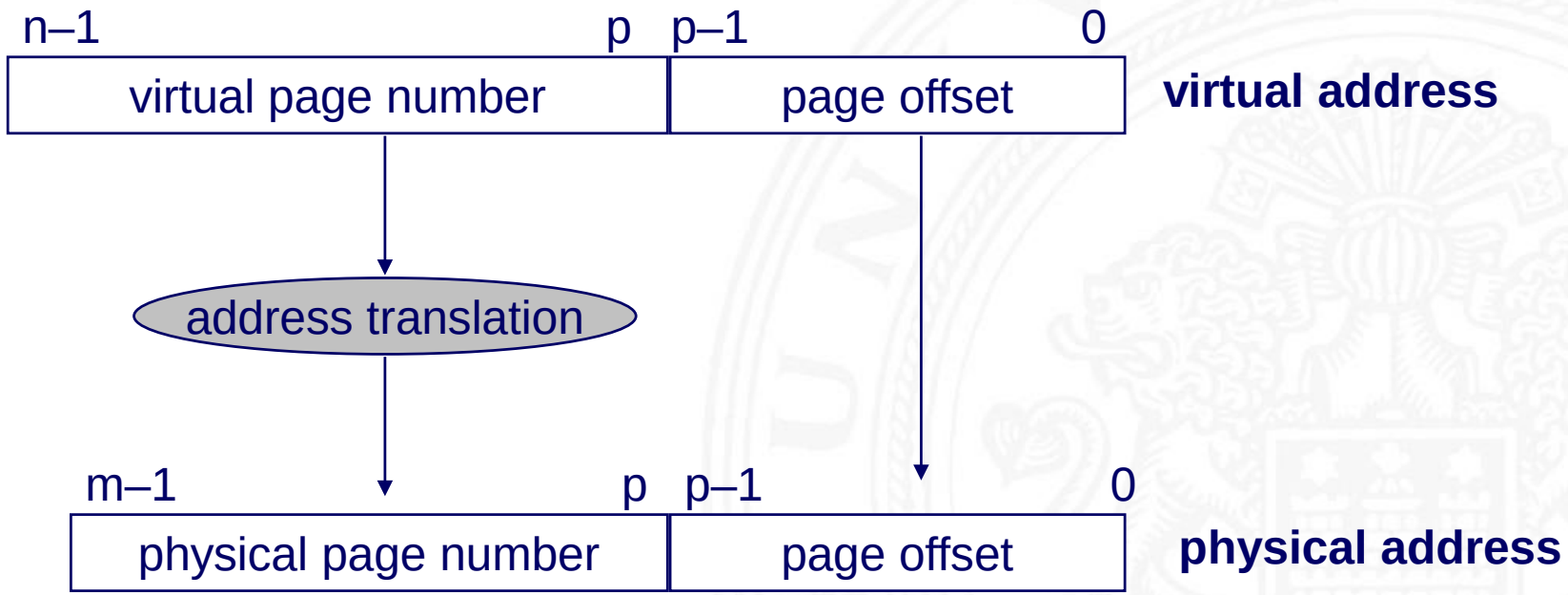
Auflösung der Adresskonflikte

- jeder Prozess hat seinen eigenen virtuellen Adressraum
- Betriebssystem kontrolliert wie virtuelle Seiten auf den physikalischen Speicher abgebildet werden



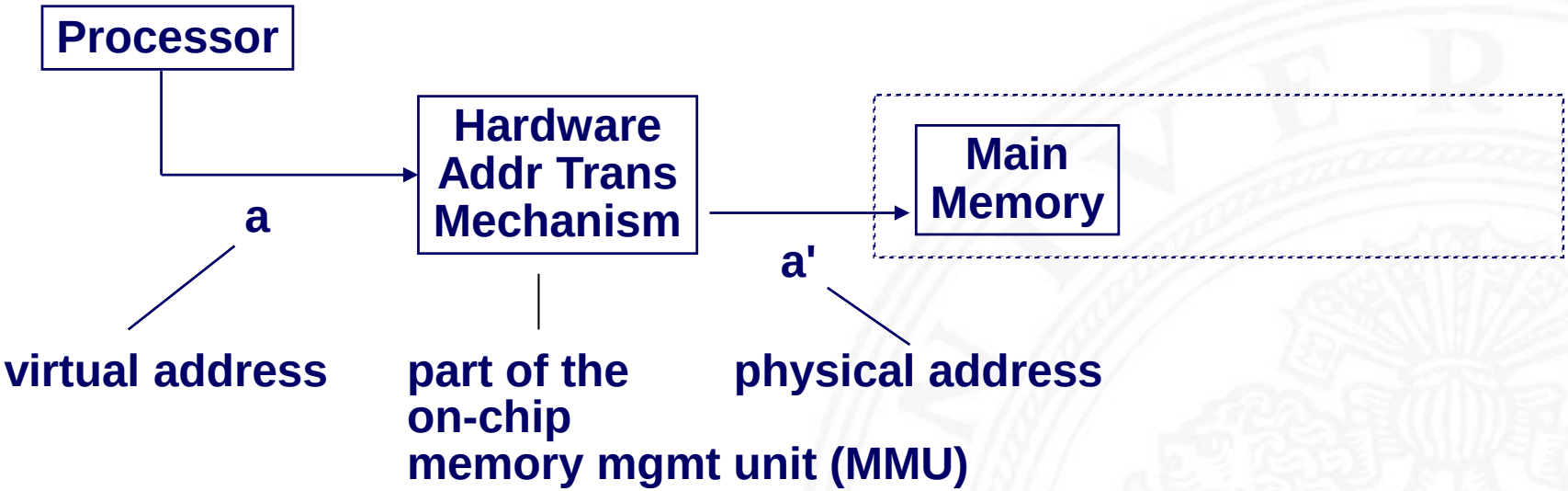
# Virtueller Speicher – Adressumsetzung

- Parameter
  - P $= 2^p$ = Seitengröße (Bytes)
  - N $= 2^n$ = Limit der virtuellen Adresse
  - M $= 2^m$ = Limit der physikalischen Adresse



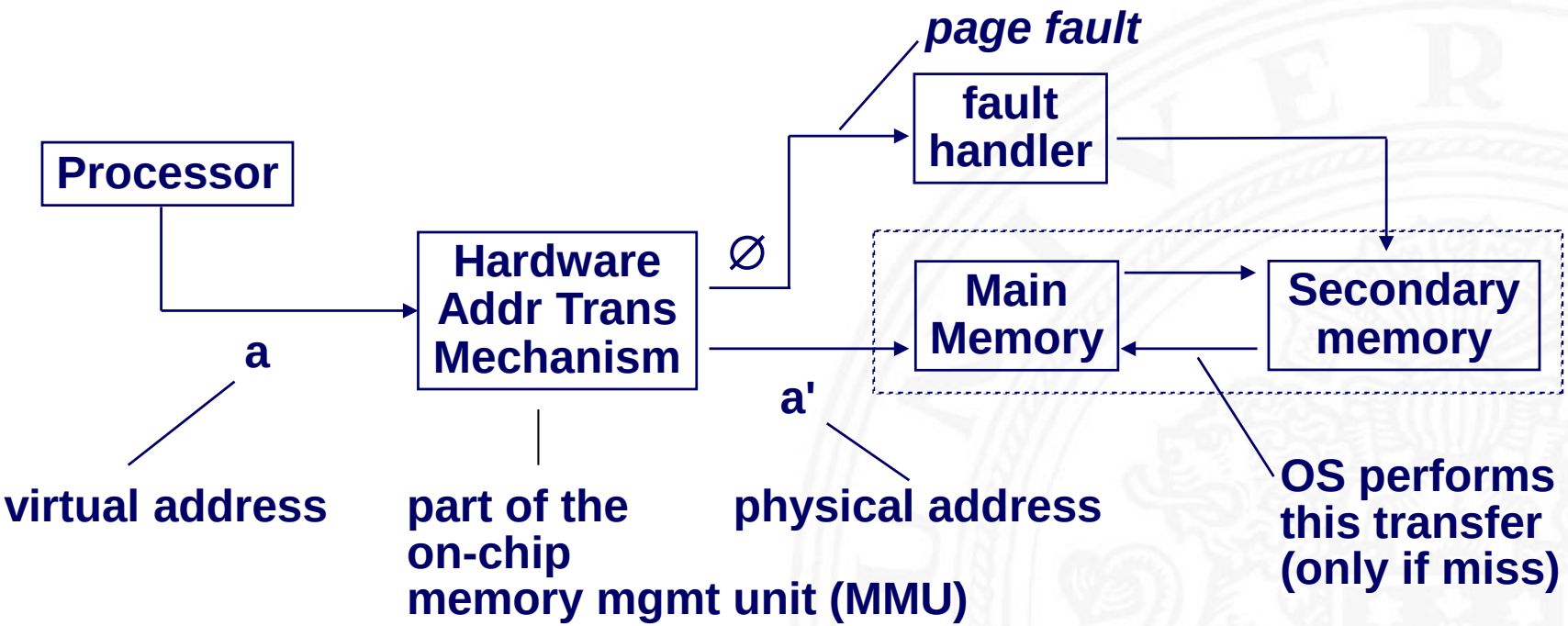
# Virtueller Speicher – Adressumsetzung (cont.)

- virtuelle Adresse: Hit

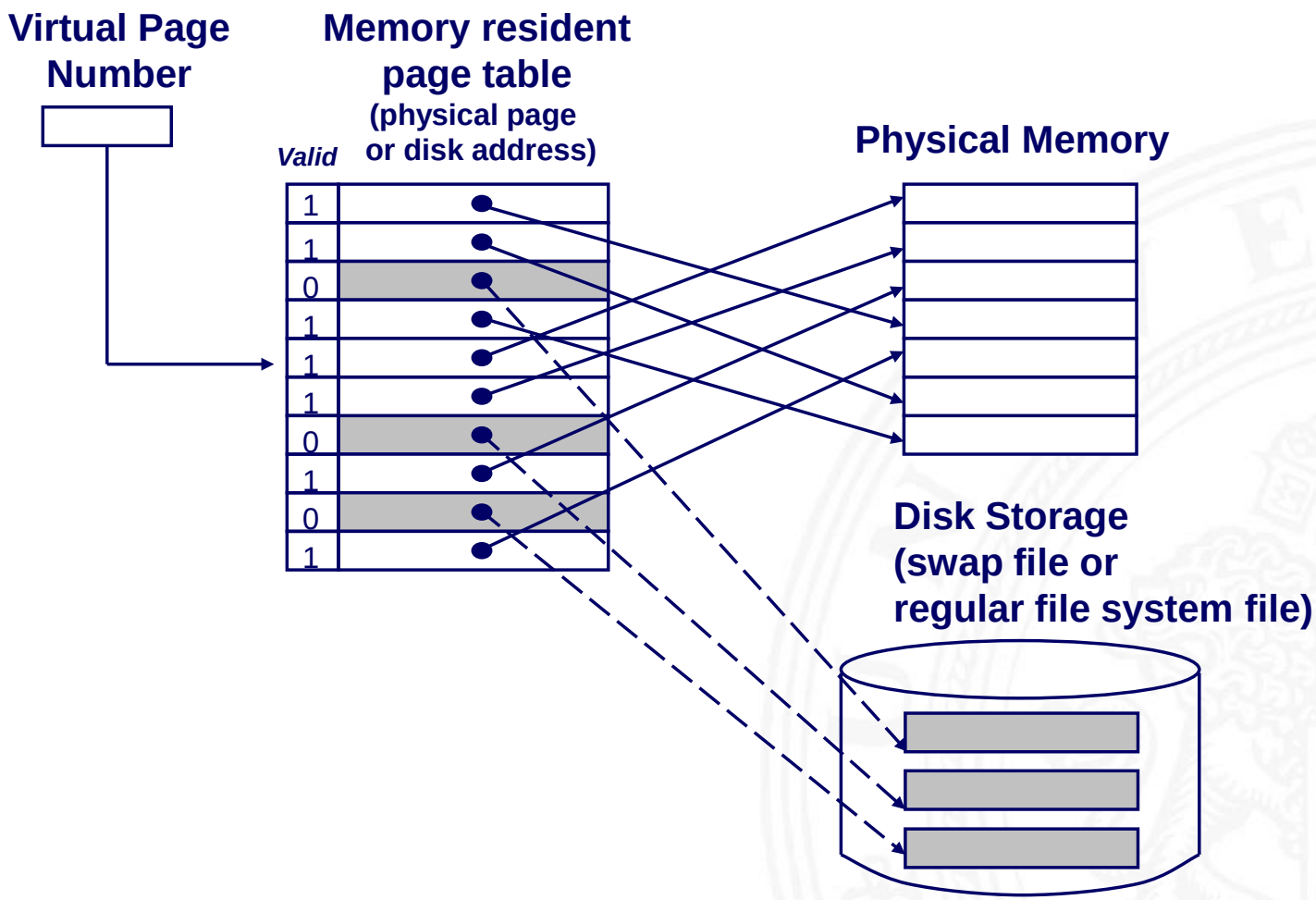


# Virtueller Speicher – Adressumsetzung (cont.)

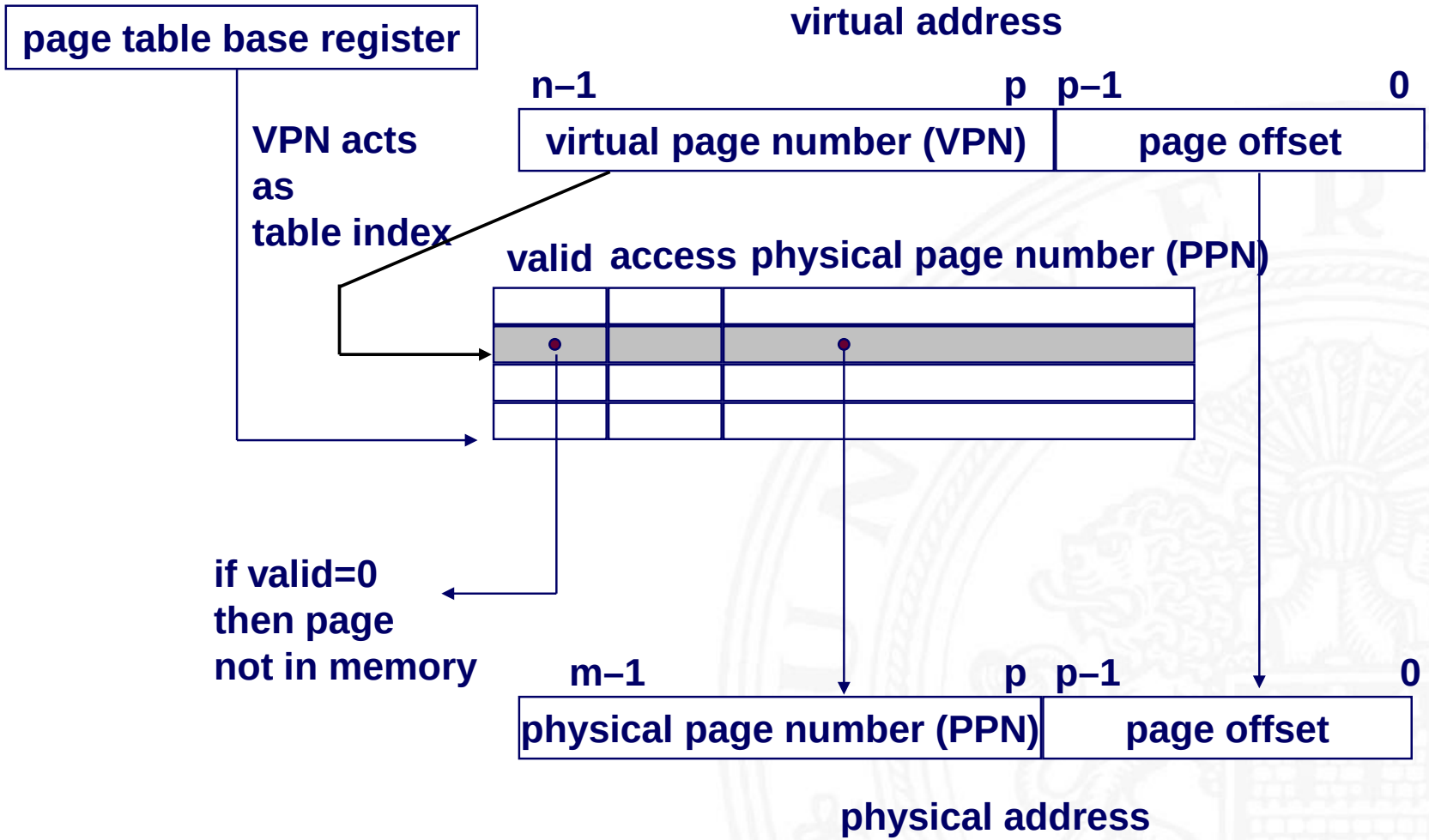
- virtuelle Adresse: Miss



# Seiten-Tabelle

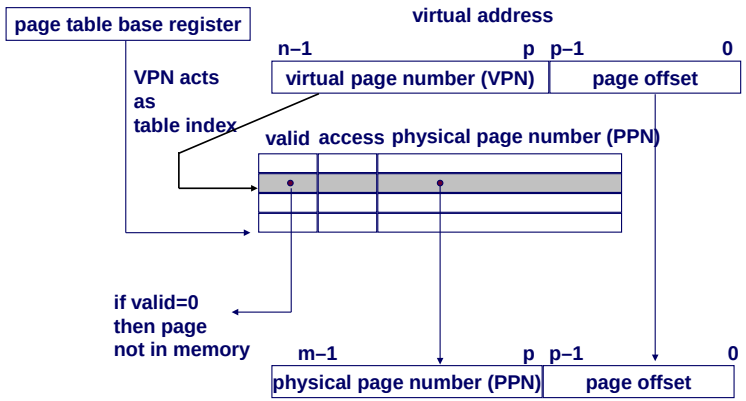


# Seiten-Tabelle (cont.)



# Seiten-Tabelle (cont.)

- separate Seiten-Tabelle für jeden Prozess
- VPN („Virtual Page Number“) bildet den Index der Seiten-Tabelle ⇒ zeigt auf Seiten-Tabelleneintrag
- Seiten-Tabelleneintrag liefert Informationen über die Seite
- Daten im Hauptspeicher: valid-Bit
  - valid-Bit = 1: die Seite ist im Speicher ⇒ benutze physikalische Seitennummer („Physical Page Number“) zur Adressberechnung
  - valid-Bit = 0: die Seite ist auf der Festplatte ⇒ Seitenfehler

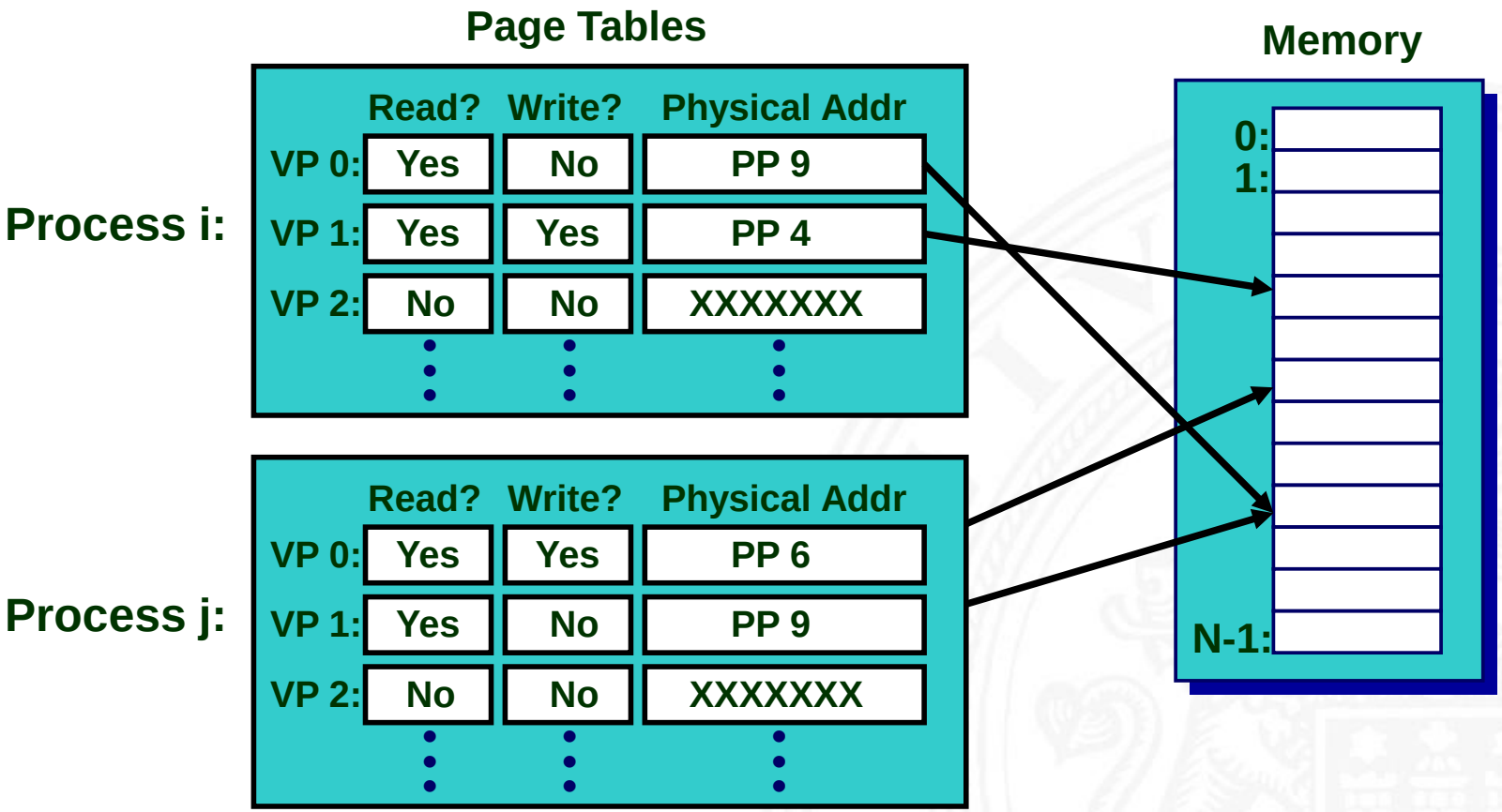


# Zugriffsrechte

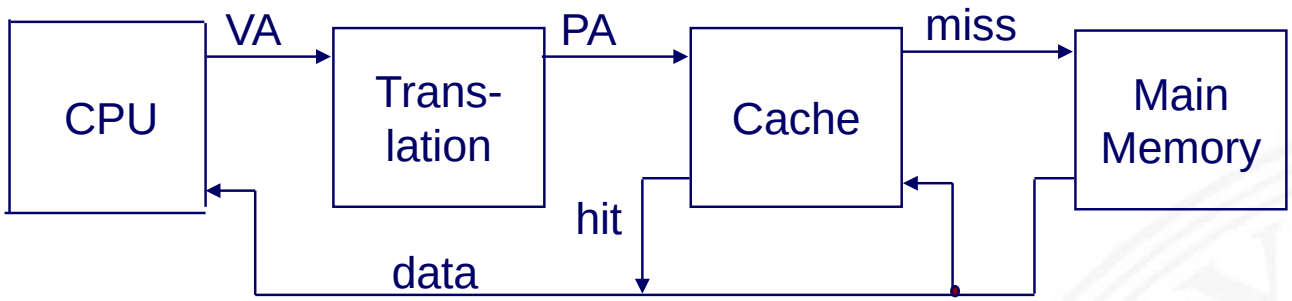
Schutzüberprüfung

- Zugriffsrechtefeld gibt Zugriffserlaubnis an
  - z.B. read-only, read-write, execute-only
  - typischerweise werden zahlreiche Schutzmodi unterstützt
- Schutzrechteverletzung wenn Prozess/Benutzer nicht die nötigen Rechte hat
- bei Verstoß erzwingt die Hardware den Schutz durch das Betriebssystem („Trap“ / „Exception“)

# Zugriffsrechte (cont.)



# Integration von virtuellem Speicher und Cache




Die meisten Caches werden *physikalisch adressiert*

- Zugriff über physikalische Adressen
- mehrere Prozesse können, gleichzeitig Blöcke im Cache haben
- –"– sich Seiten teilen
- Cache muss sich nicht mit Schutzproblemen befassen
  - Zugriffsrechte werden als Teil der Adressumsetzung überprüft

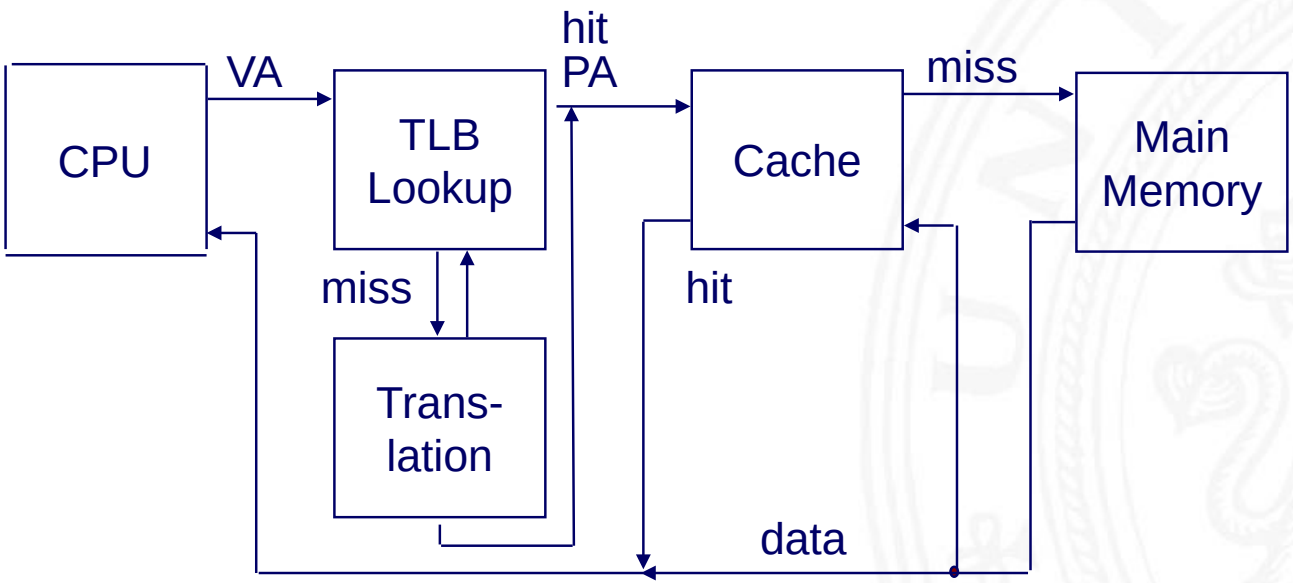
Die Adressumsetzung wird vor dem Cache „Lookup“ durchgeführt

- kann selbst Speicherzugriff (auf den PTE) beinhalten
- Seiten-Tabelleneinträge können auch gecacht werden

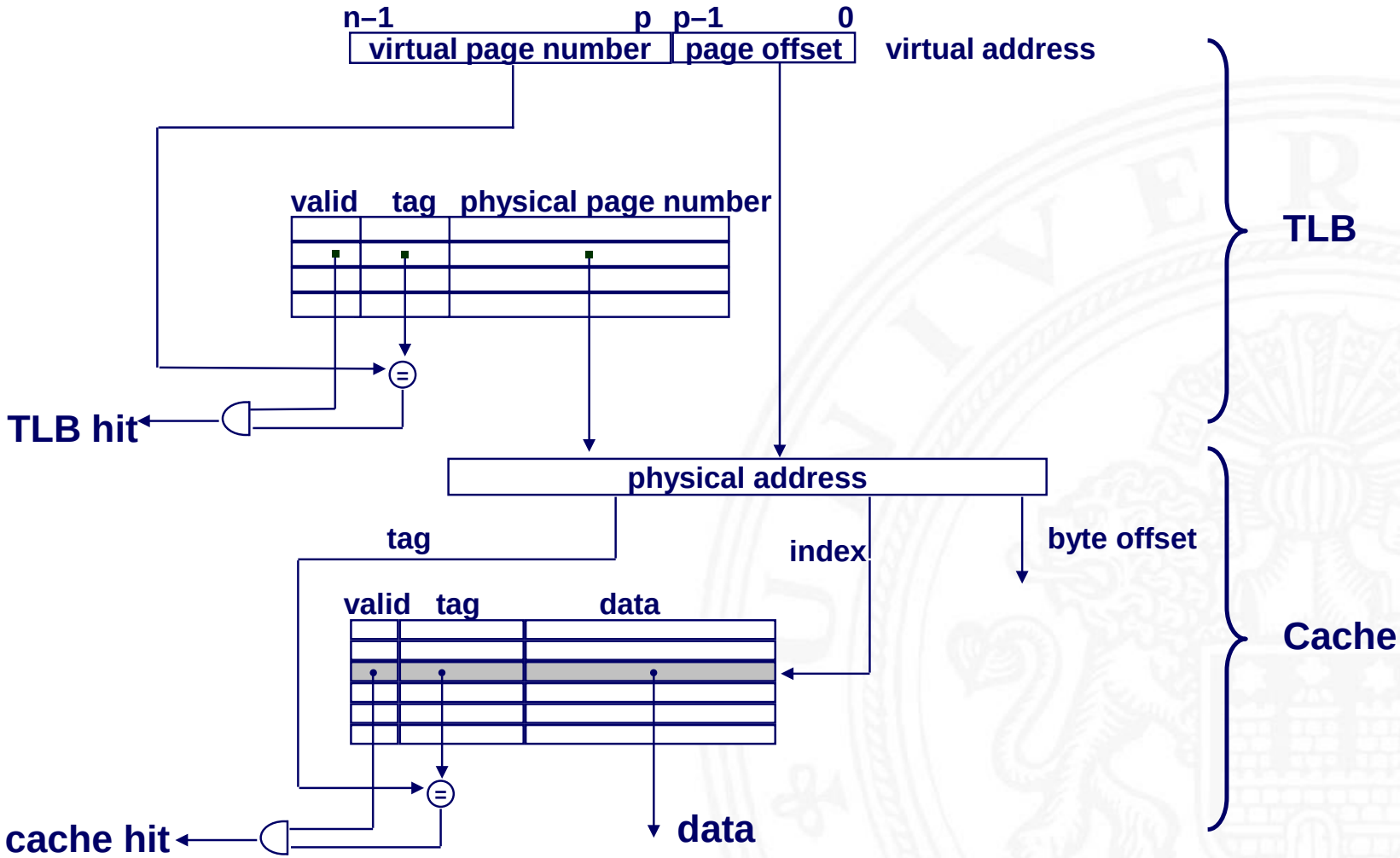
# TLB / „Translation Lookaside Buffer“

Beschleunigung der Adressumsetzung für virtuellen Speicher

- kleiner Hardware Cache in MMU (Memory Management Unit)
- bildet virtuelle Seitenzahlen auf physikalische ab
- enthält komplette Seiten-Tabelleneinträge für wenige Seiten



# TLB / „Translation Lookaside Buffer“ (cont.)

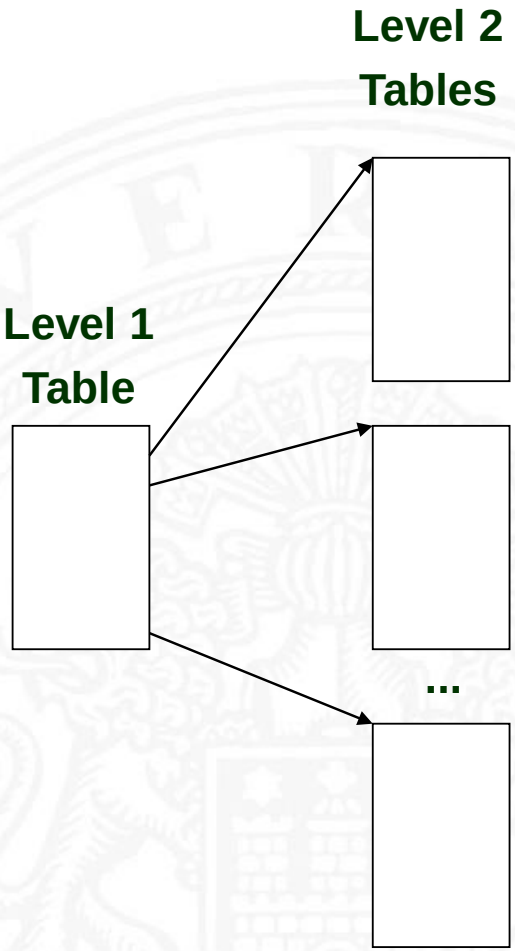


# mehrstufige Seiten-Tabellen

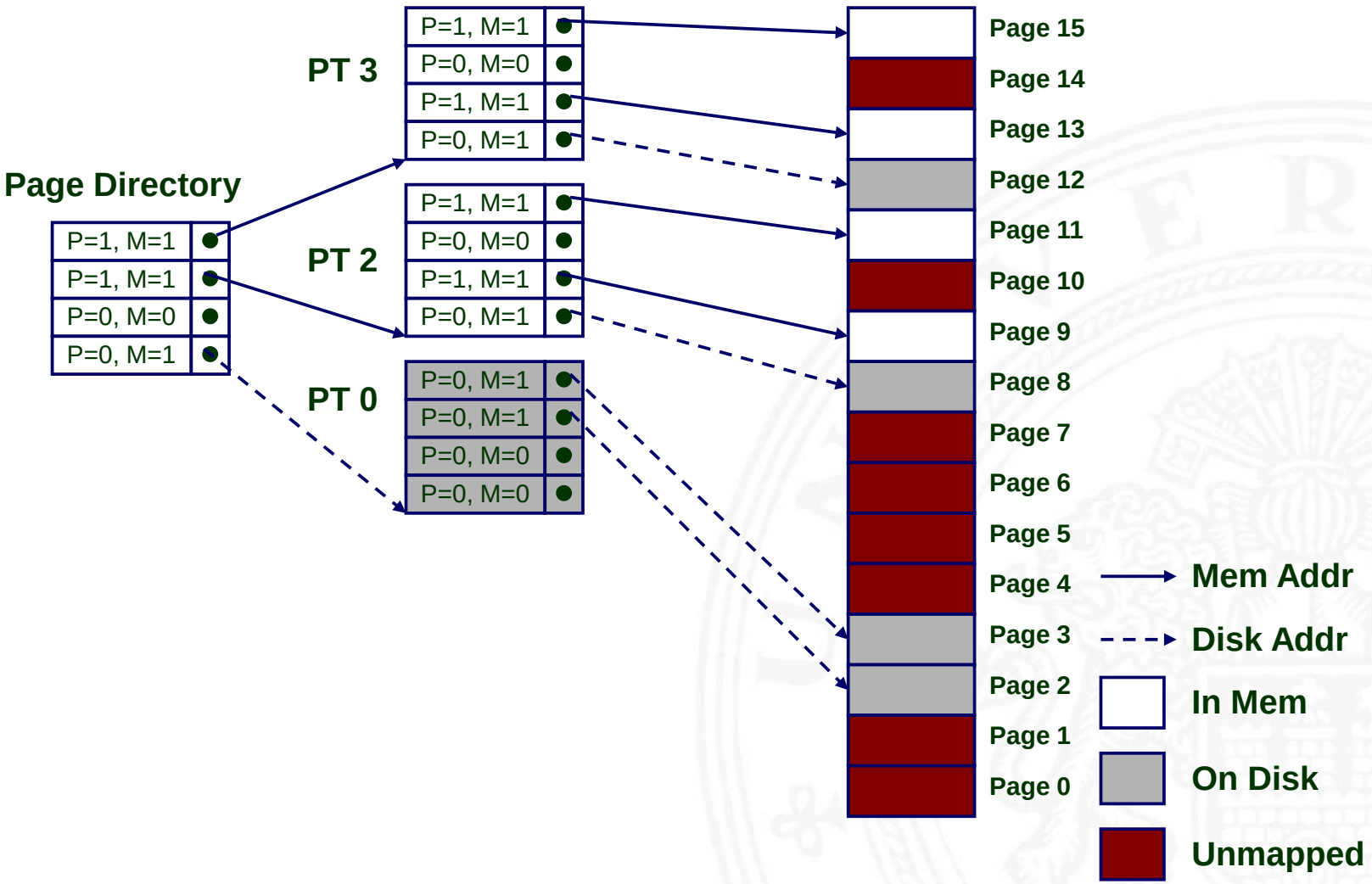
- Gegeben
  - 4 KiB ($2^{12}$) Seitengröße
  - 32-Bit Adressraum
  - 4-Byte PTE („Page Table Entry“) Seitentabelleneintrag
- Problem
  - erfordert 4 MiB Seiten-Tabelle
  - $2^{20}$ Bytes

$\Rightarrow$ übliche Lösung

- mehrstufige Seiten-Tabellen („multi-level“)
- z.B. zweistufige Tabelle (Pentium P6)
  - Ebene-1: 1024 Einträge $\rightarrow$ Ebene-2 Tabelle
  - Ebene-2: 1024 Einträge $\rightarrow$ Seiten

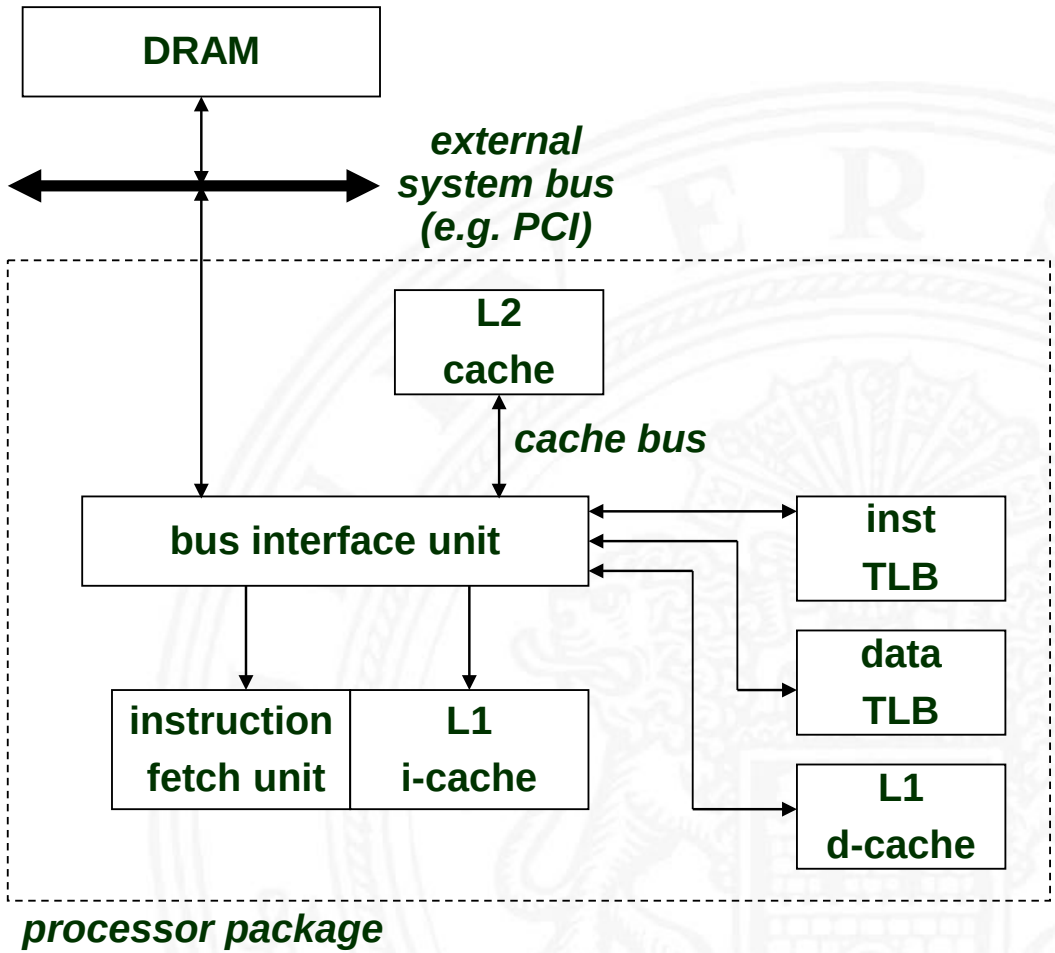


# mehrstufige Seiten-Tabellen (cont.)

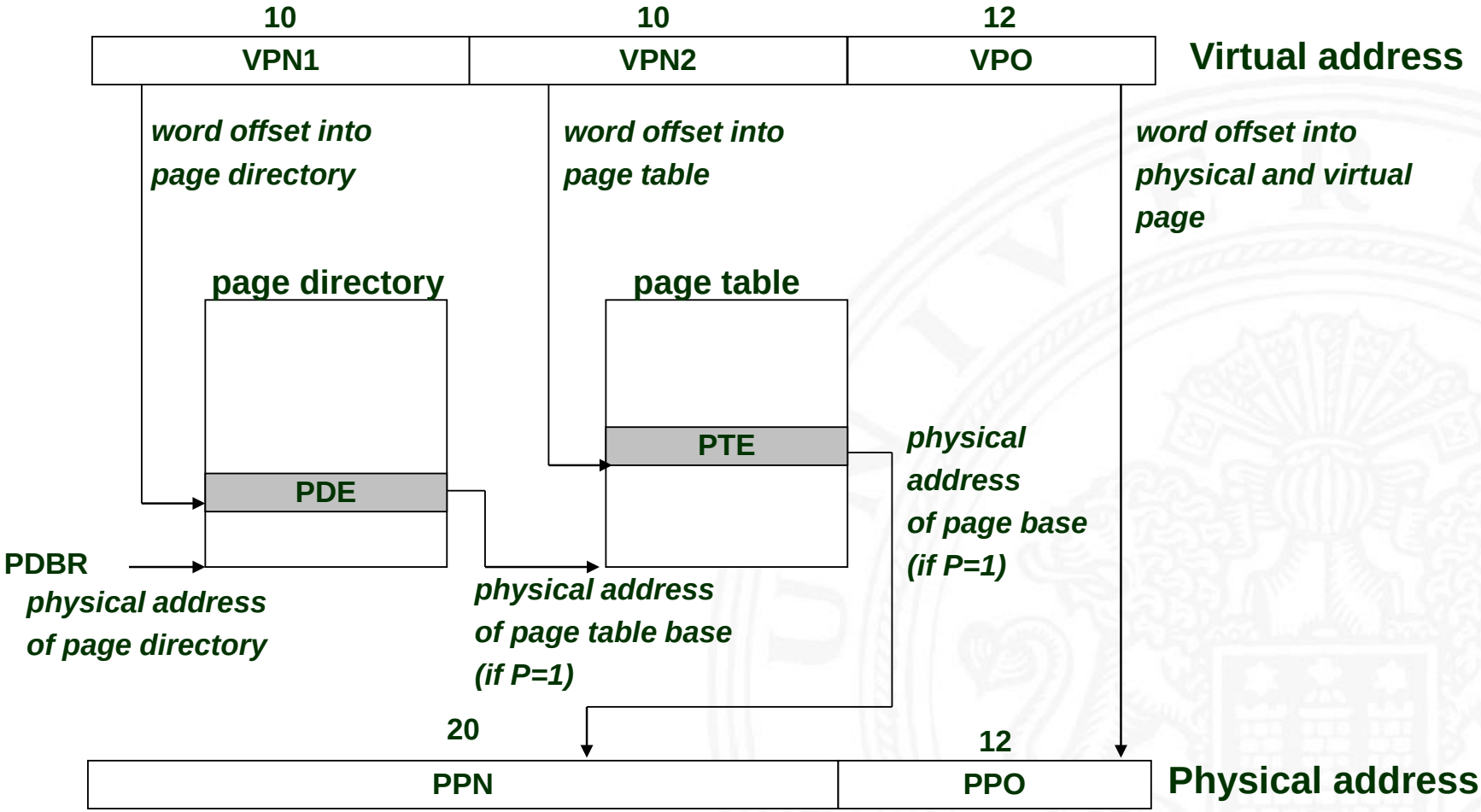


# Beispiel: Pentium und Linux

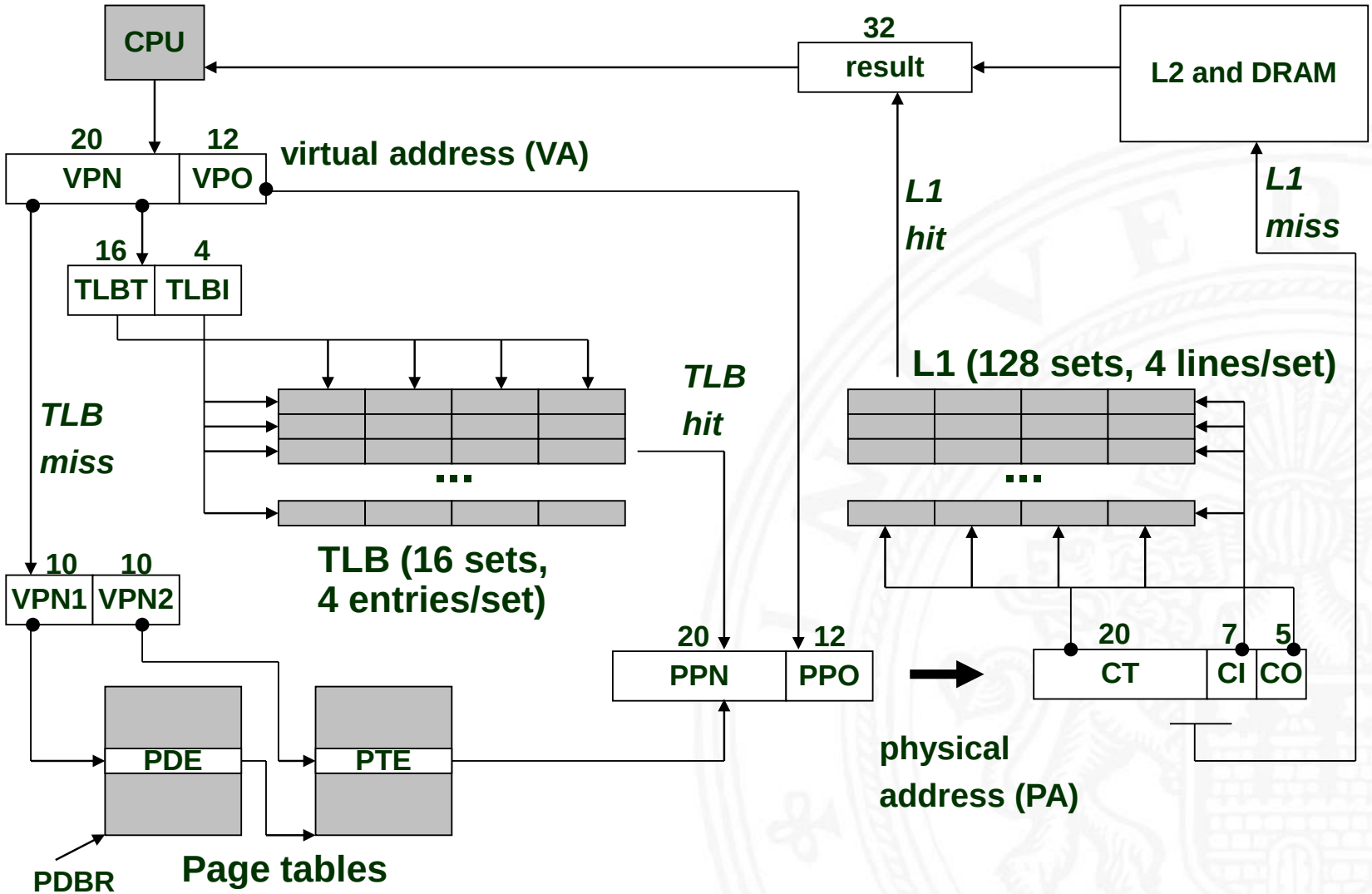
- 32-bit Adressraum
- 4 KB Seitengröße
- L1, L2 TLBs
  4fach assoziativ
- Instruktions TLB
  32 Einträge
  8 Sets
- Daten TLB
  64 Einträge
  16 Sets
- L1 I-Cache, D-Cache
  16 KB
  32 B Cacheline
  128 Sets
- L2 Cache
  Instr.+Daten zusammen
  128 KB . . . 2 MB



# Beispiel: Pentium und Linux (cont.)



# Beispiel: Pentium und Linux (cont.)



# Zusammenfassung

Cache Speicher

- Dient nur zur Beschleunigung
- Verhalten unsichtbar für Anwendungsprogrammierer und OS
- Komplett in Hardware implementiert

# Zusammenfassung (cont.)

Virtueller Speicher

- Ermöglicht viele Funktionen des Betriebssystems
  - Prozesse erzeugen („exec“ / „fork“)
  - Taskwechsel
  - Schutzmechanismen

- Implementierung mit Hardware und Software
  - Software verwaltet die Tabellen und Zuteilungen
  - Hardwarezugriff auf die Tabellen
  - Hardware-Caching der Einträge (TLB)

# Zusammenfassung (cont.)

- Sicht des Programmierers
  - großer „flacher“ Adressraum
    - kann große Blöcke benachbarter Adressen zuteilen
  - Prozessor „besitzt“ die gesamte Maschine
    - hat privaten Adressraum
    - bleibt unberührt vom Verhalten anderer Prozesse
- Sicht des Systems
  - Virtueller Adressraum von Prozessen durch Abbildung auf Seiten
    - muss nicht fortlaufend sein
    - wird dynamisch zugeteilt
    - erzwingt Schutz bei Adressumsetzung
  - Betriebssystem verwaltet viele Prozesse gleichzeitig
    - ständiger Wechsel zwischen Prozessen
    - vor allem wenn auf Ressourcen gewartet werden muss