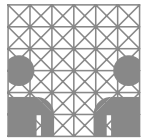
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MIN-Fakultät
Fachbereich Informatik



64-040 Modul InfB-RS: Rechnerstrukturen

https://tams.informatik.uni-hamburg.de/lectures/2016ws/vorlesung/rs

– Kapitel 18 –

Andreas Mäder

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

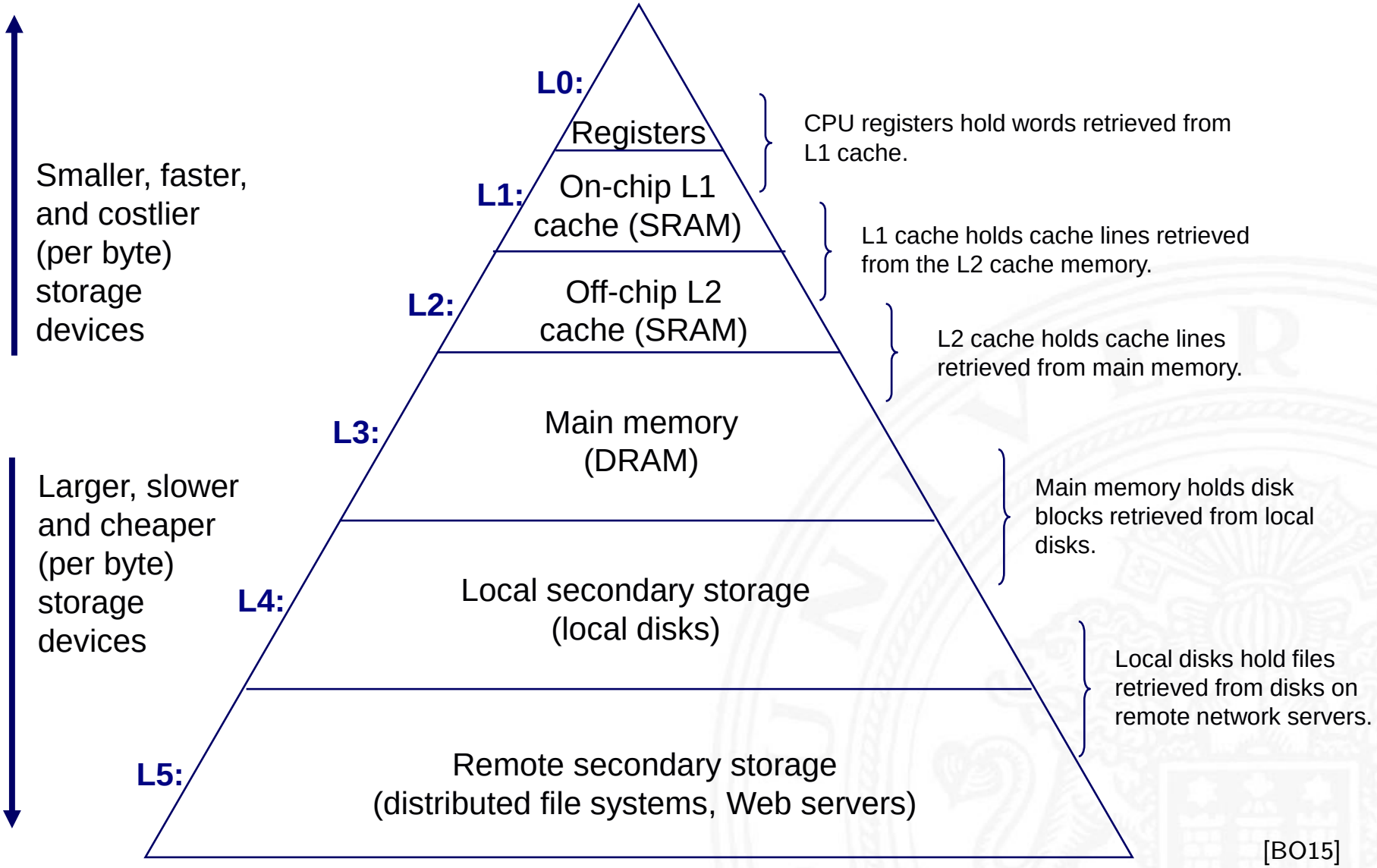
Technische Aspekte Multimodaler Systeme

Wintersemester 2016/2017

Kapitel 18

Speicherhierarchie

Speicherhierarchie: Konzept



[BO15]

Speicherhierarchie: Konzept (cont.)

Gesamtsystem kombiniert verschiedene Speicher

- wenige KByte Register (-bank) im Prozessor
- einige MByte SRAM als schneller Zwischenspeicher
- einige GByte DRAM als Hauptspeicher
- einige TByte Festplatte als nichtflüchtiger Speicher
- Hintergrundspeicher (CD/DVD/BR, Magnetbänder)
- das WWW und Cloud-Services

Kompromiss aus Kosten, Kapazität, Zugriffszeit

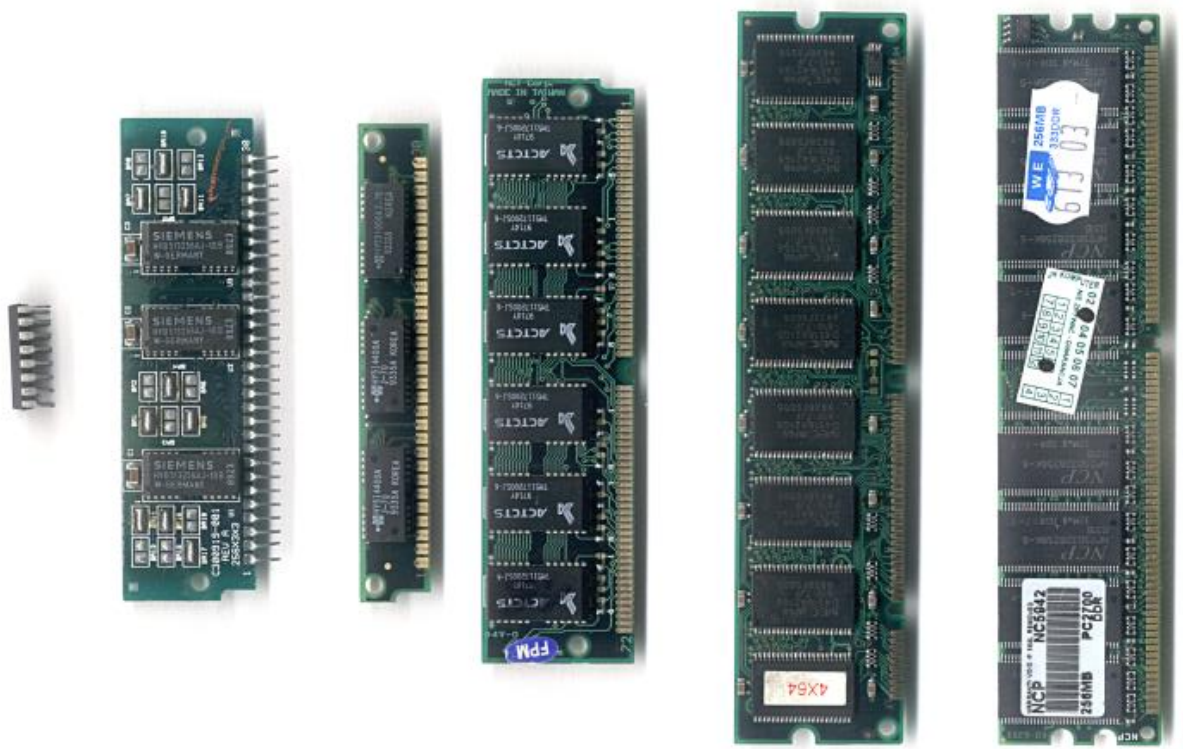
- Illusion aus großem schnellem Speicher
- funktioniert nur wegen räumlicher/zeitlicher Lokalität

L0: Register

- Register im Prozessor integriert
  - Program-Counter und Datenregister für Programmierer sichtbar
  - ggf. weitere Register für Systemprogrammierung
  - zusätzliche unsichtbare Register im Steuerwerk
- Flipflops oder Registerbank mit 6 Trans.-Speicherzellen
  - Lesen und Schreiben in jedem Takt möglich
  - ggf. mehrere parallele Lesezugriffe in jedem Takt
  - Zugriffszeiten ca. 100 ps
- typ. Größe einige KByte, z.B. 16 Register á 64-bit *x86-64*

L1-L3: Halbleiterspeicher RAM

- „*Random-Access Memory*“ (RAM) aufgebaut aus Mikrochips
- Grundspeichereinheit ist eine Zelle (ein Bit pro Zelle)
- SRAM (6T-Zelle) oder DRAM (1T-Zelle) Technologie
- mehrere RAM Chips bilden einen Speicher



- dominierende Technologie für nichtflüchtigen Speicher
- hohe Speicherkapazität, derzeit einige TB
  - Daten bleiben beim Abschalten erhalten
  - aber langsamer Zugriff
  - besondere Algorithmen, um langsamen Zugriff zu verbergen

  - Einsatz als Speicher für dauerhafte Daten
  - Einsatz als erweiterter Hauptspeicher („*virtual memory*“)

- FLASH/SSD zunehmend als Ersatz für Festplatten
  - Halbleiterspeicher mit sehr effizienten multibit-Zellen
  - Verwaltung (derzeit) wie Festplatten
  - signifikant schnellere Zugriffszeiten

L5: Hintergrundspeicher

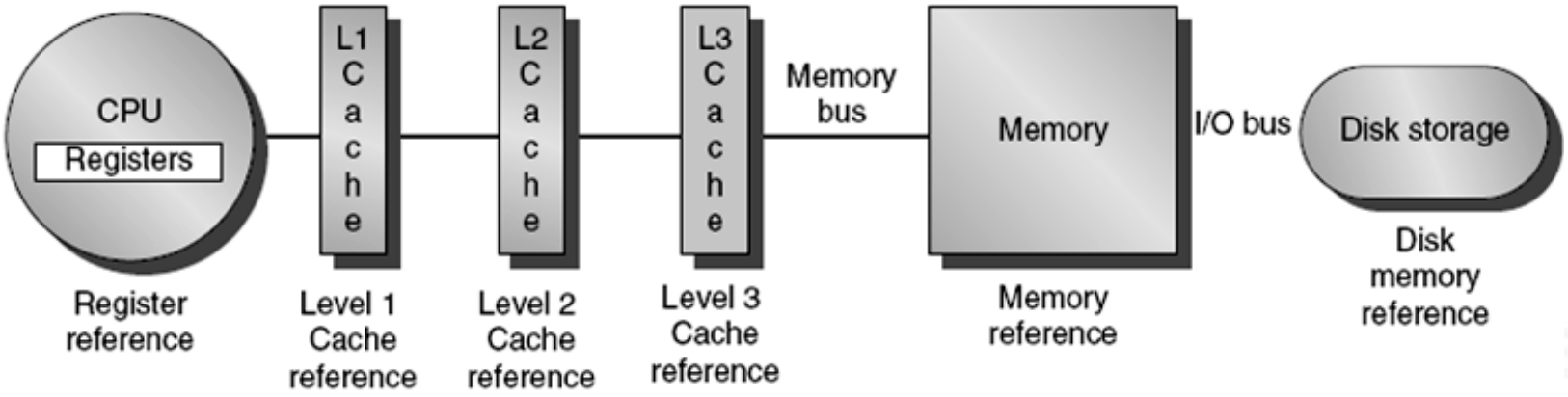
- enorme Speicherkapazität
- langsame Zugriffszeiten

- Archivspeicher und Backup für (viele) Festplatten
  - Magnetbänder
  - RAID-Verbund aus mehreren Festplatten
  - optische Datenspeicher: CD-ROM, DVD-ROM, BlueRay

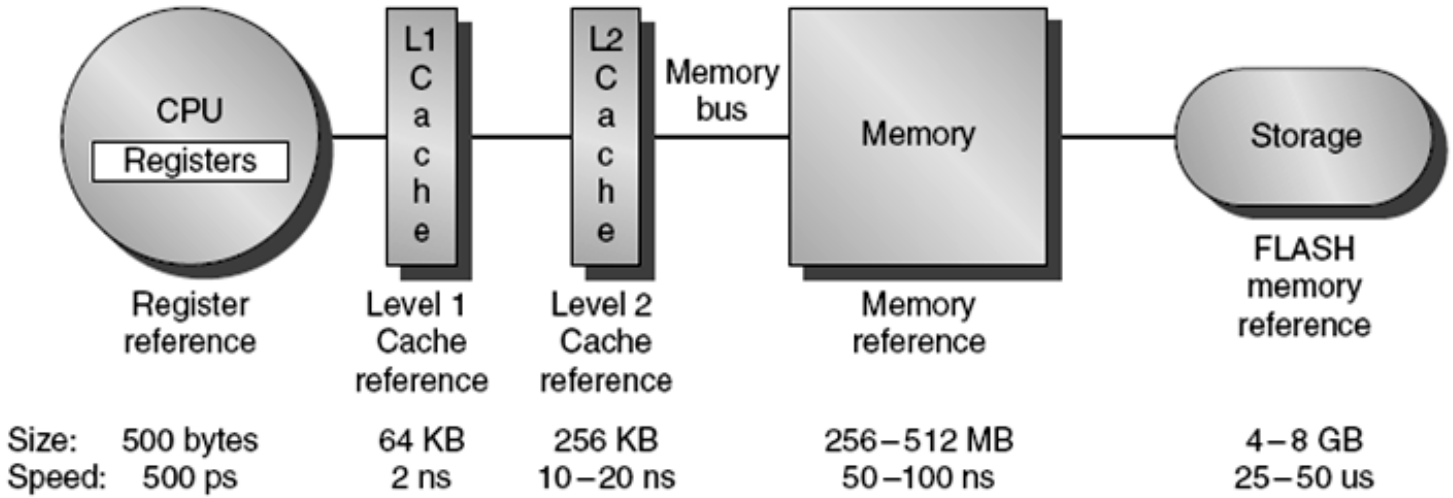
- WWW und Internet-Services, Cloud-Services
  - Cloud-Farms ggf. ähnlich schnell wie L4 Festplatten, da Netzwerk schneller als der Zugriff auf eine lokale Festplatte

- in dieser Vorlesung nicht behandelt

Speicherhierarchie: zwei Beispiele



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

[HP12]

Random-Access Memory / RAM

SRAM „statisches RAM“

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

DRAM „dynamisches RAM“

- jede Zelle speichert Bit mit 1 Kondensator und 1 Transistor
- der Wert muss alle 10-100 ms aufgefrischt werden
- anfällig für Störungen
- langsamer und billiger als SRAM

SRAM vs. DRAM

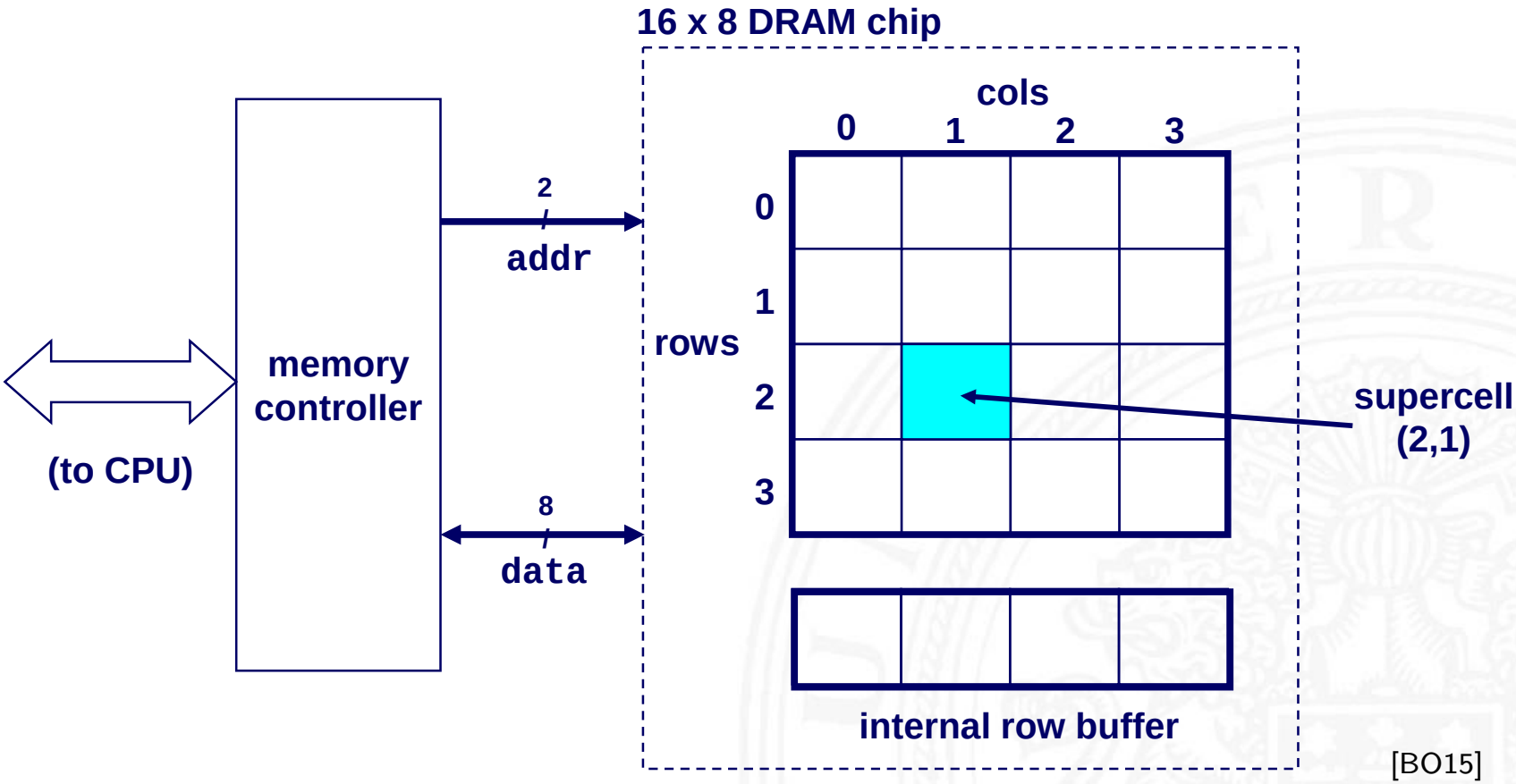
	SRAM	DRAM
Zugriffszeit	5...50 ns	60...100 ns t_{rac}
		20...300 ns t_{cac}
		110...180 ns t_{cyc}
Leistungsaufnahme	200...1300 mW	300...600 mW
Speicherkapazität	$<$ 72 Mbit	$<$ 4 Gbit
Preis	10 €/Mbit	0,1 Ct./Mbit

	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

[BO15]

DRAM Organisation

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

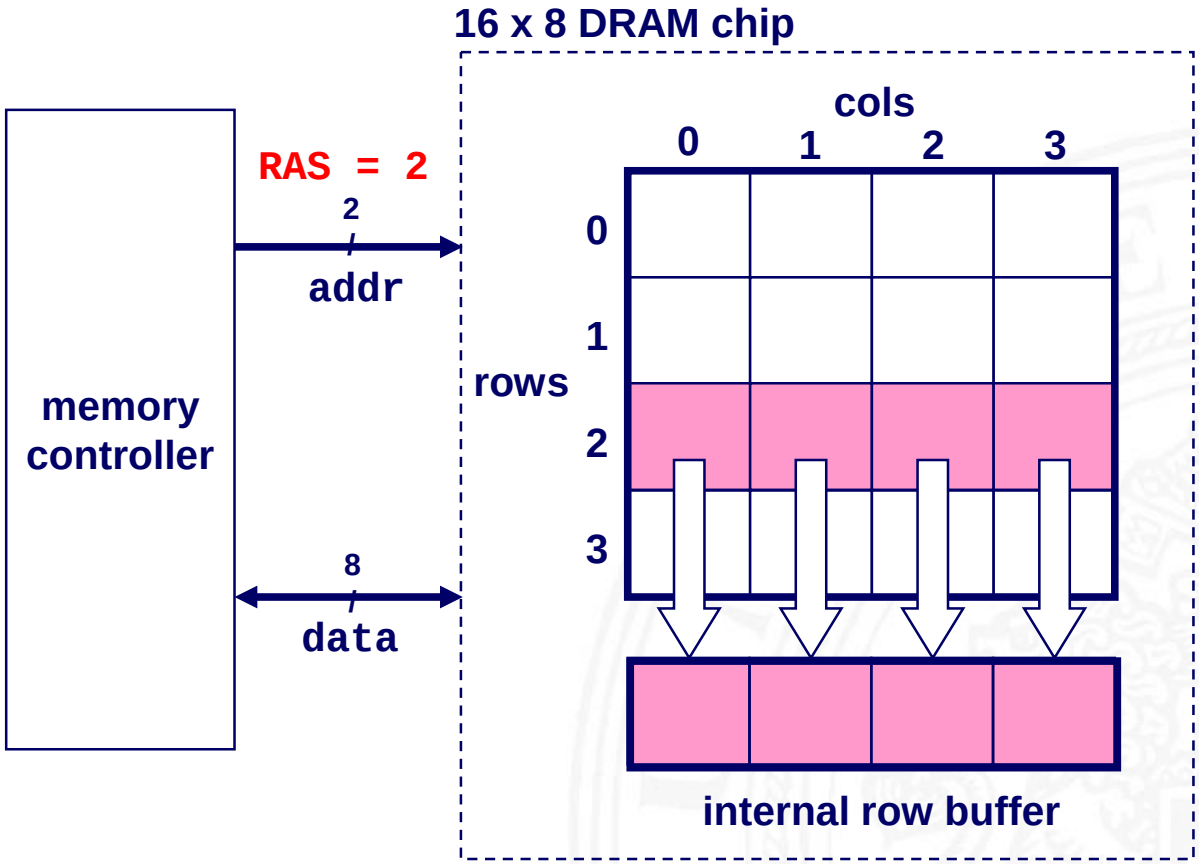


[BO15]

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

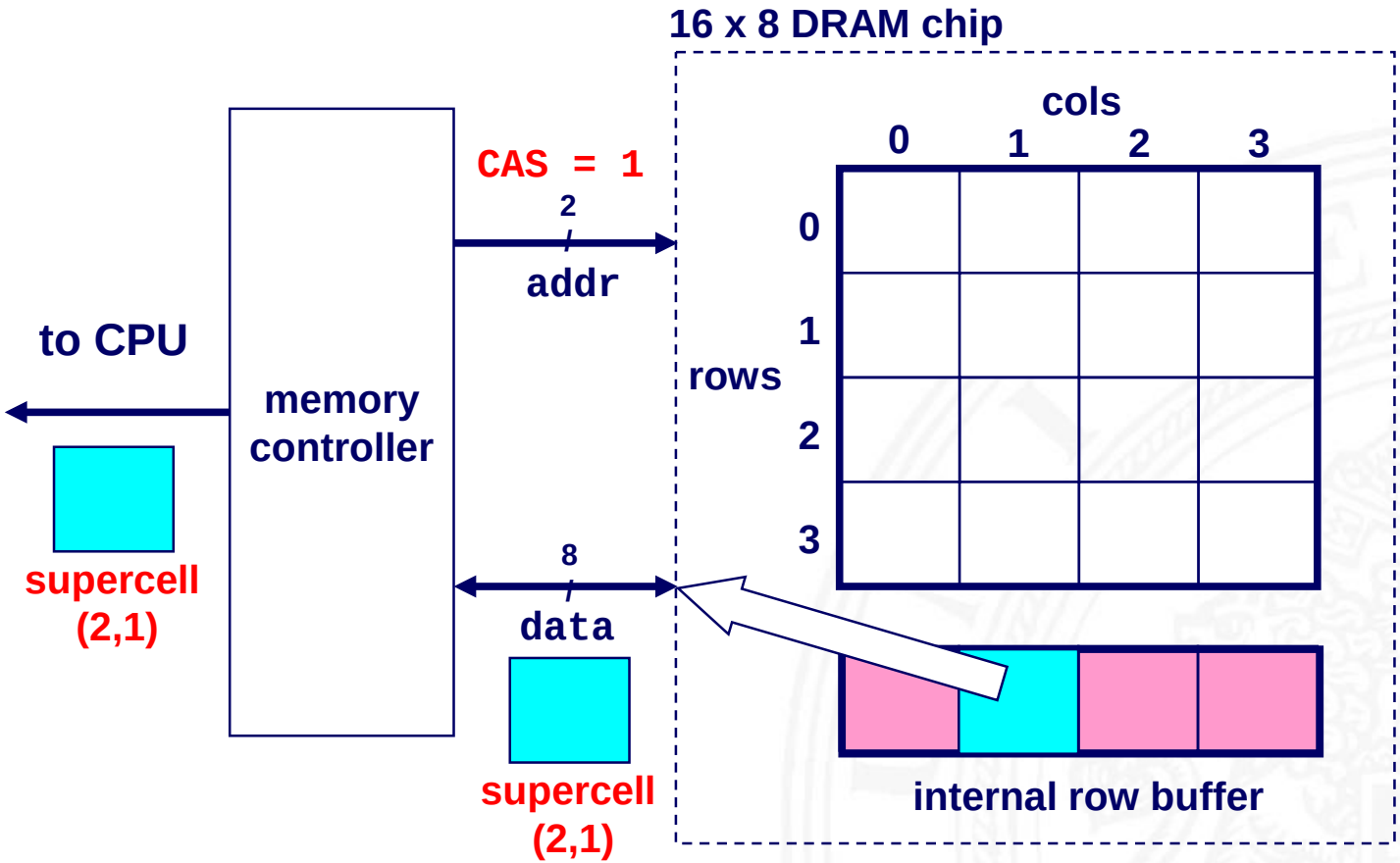


[BO15]

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

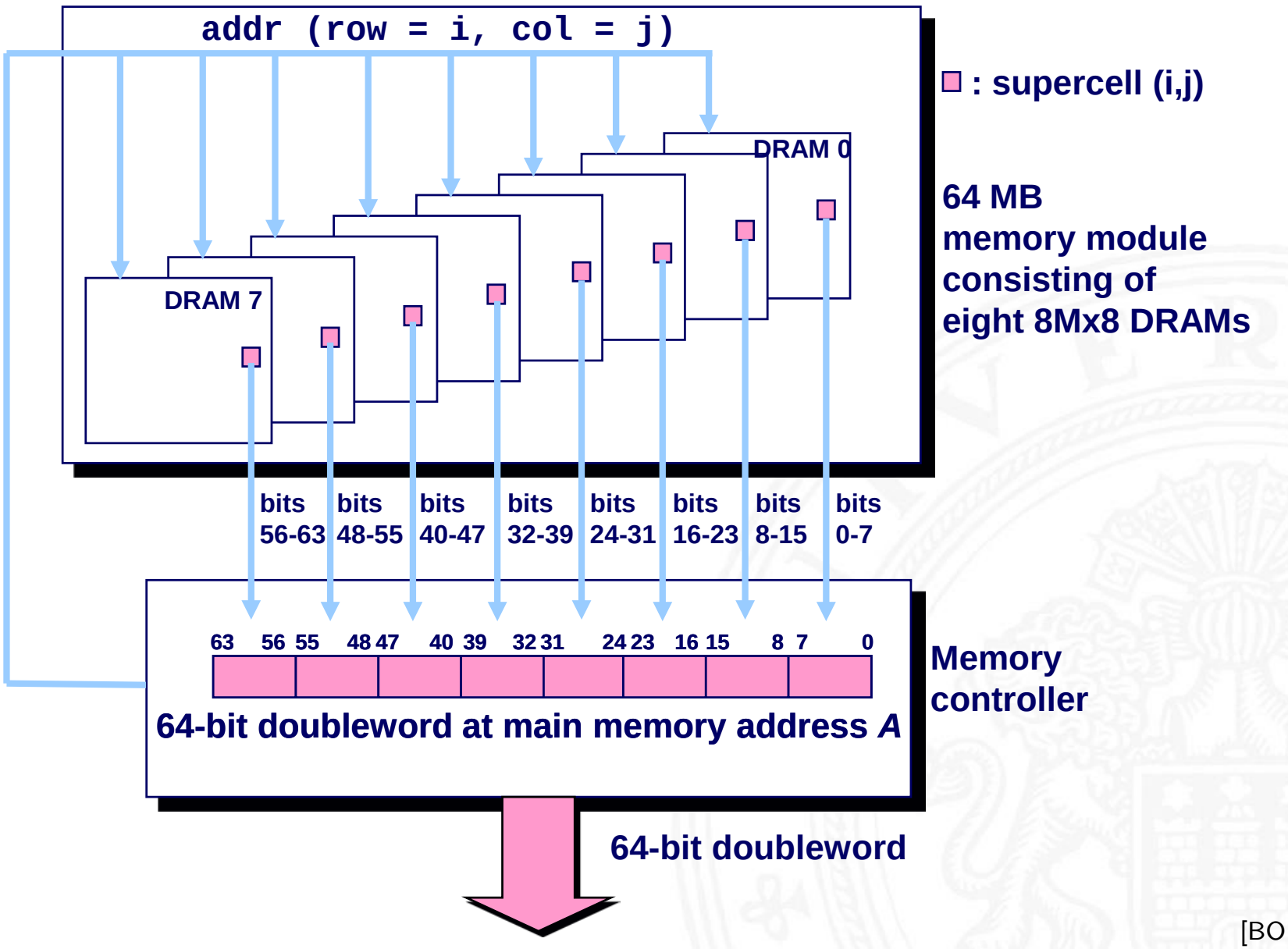
2.a „Column Access Strobe“ (CAS) wählt Spalte 1

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



[BO15]

Speichermodule



[BO15]

Nichtflüchtige Speicher

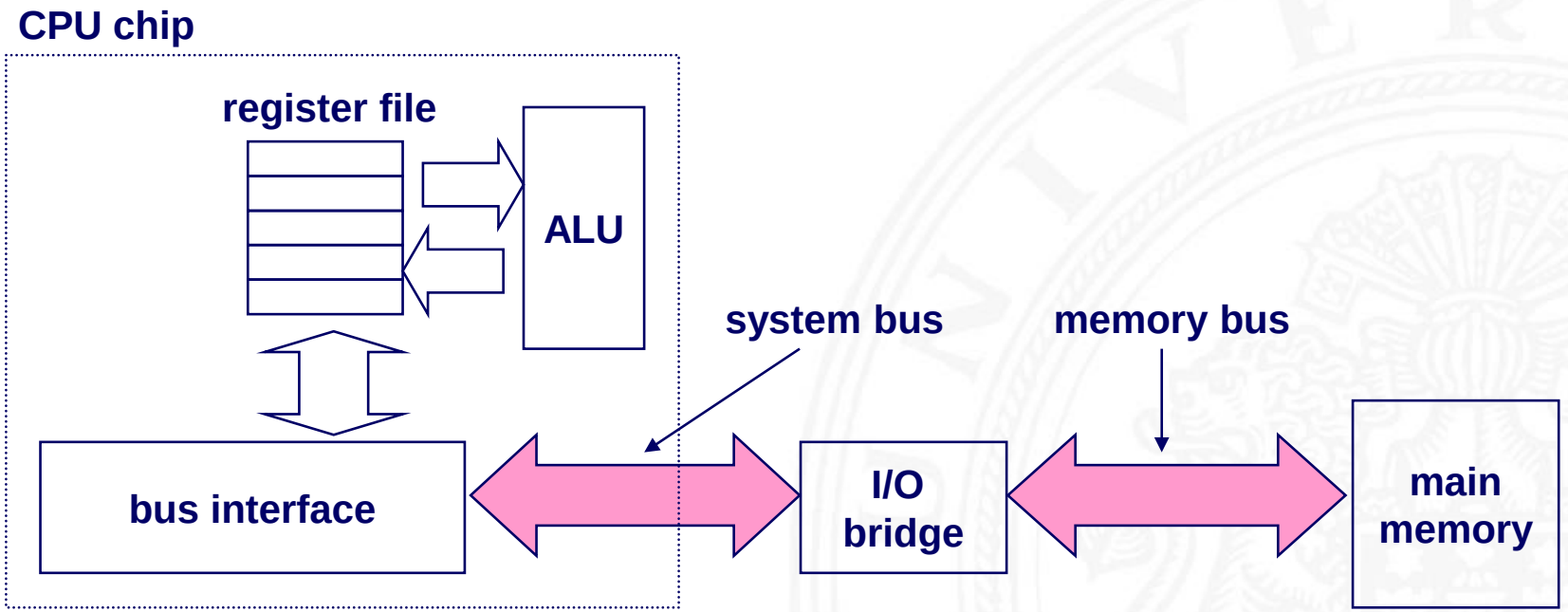
- DRAM und SRAM sind flüchtige Speicher
  - Informationen gehen beim Abschalten verloren
- nichtflüchtige Speicher speichern Werte selbst wenn sie spannungslos sind
  - allgemeiner Name ist „Read-Only-Memory“ (ROM)
  - irreführend, da einige ROMs auch verändert werden können
- Arten programmierbarer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

Anwendungen für nichtflüchtigen Speicher

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

Bussysteme verbinden CPU und Speicher

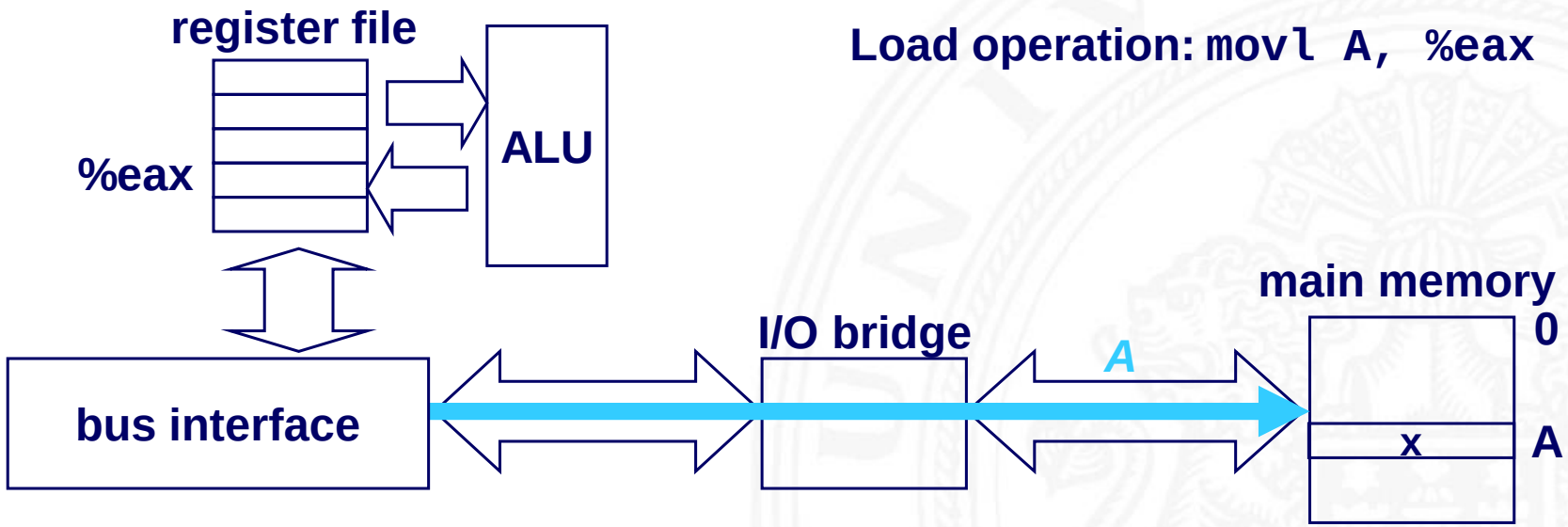
- Bus: Bündel paralleler Leitungen
- es gibt mehr als eine Quelle $\Rightarrow$ Tristate-Treiber
- Busse im Rechner
  - zur Übertragung von Adressen, Daten und Kontrollsignalen
  - werden üblicherweise von mehreren Geräten genutzt



[BO15]

lesender Speicherzugriff

1. CPU legt Adresse A auf den Speicherbus



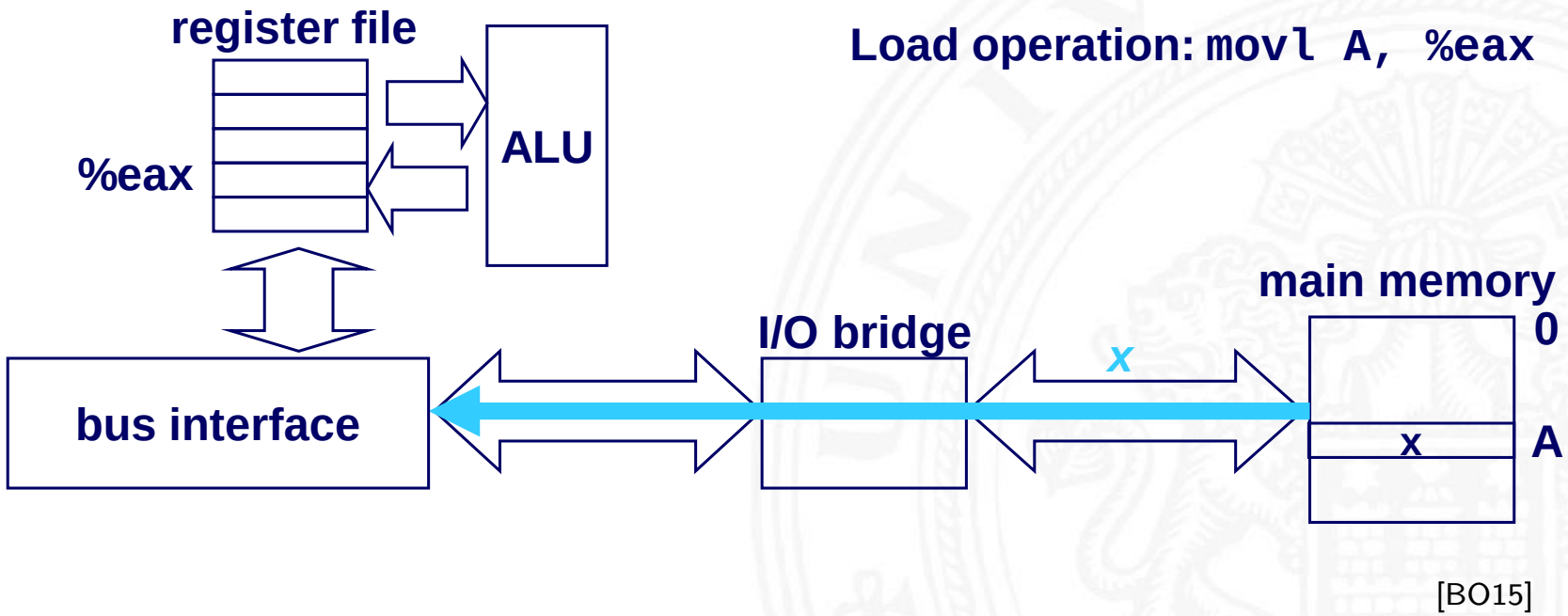
[BO15]

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

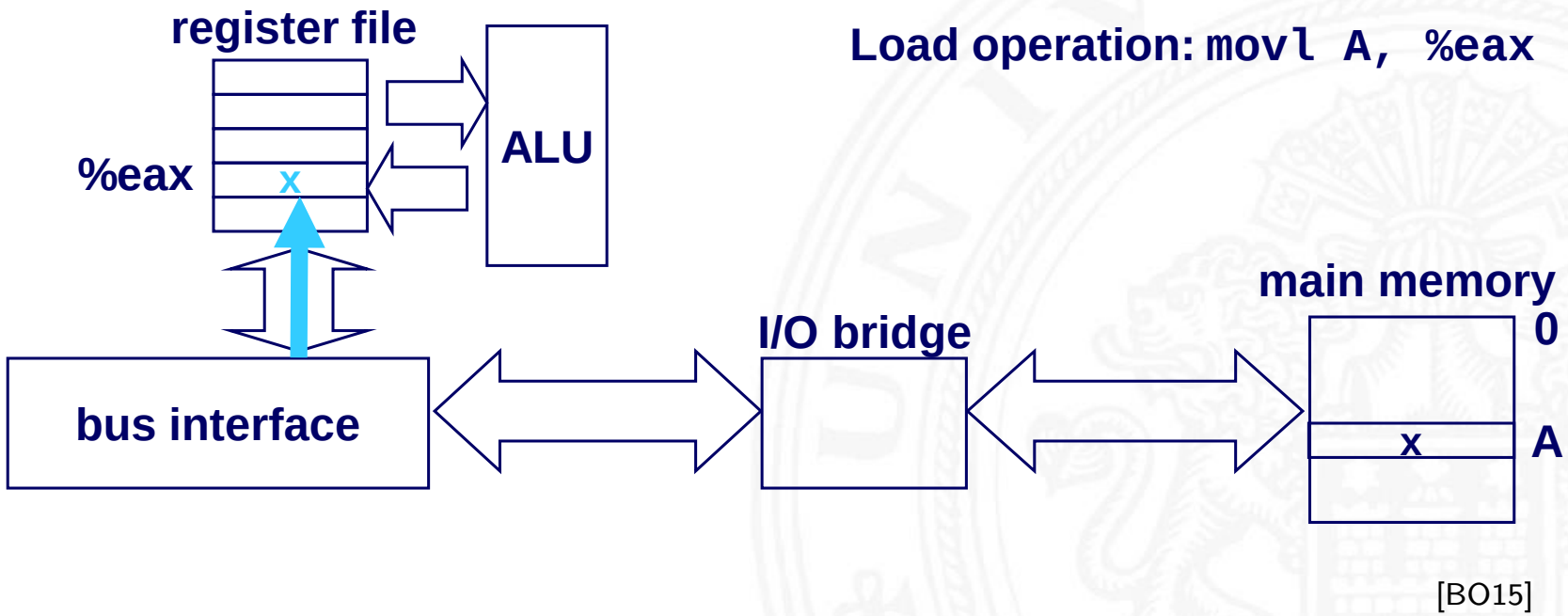


[BO15]

lesender Speicherzugriff (cont.)

3.a CPU liest Wort x vom Bus

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



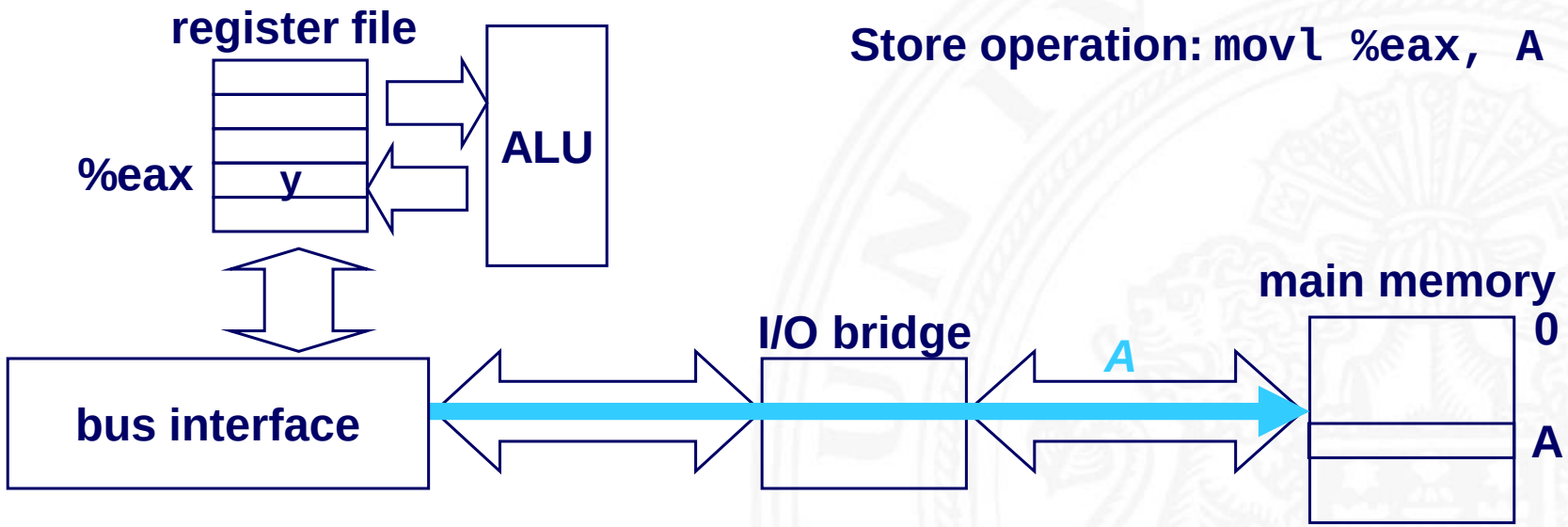
[BO15]

schreibender Speicherzugriff

1 CPU legt die Adresse A auf den Bus

2.b Hauptspeicher liest Adresse

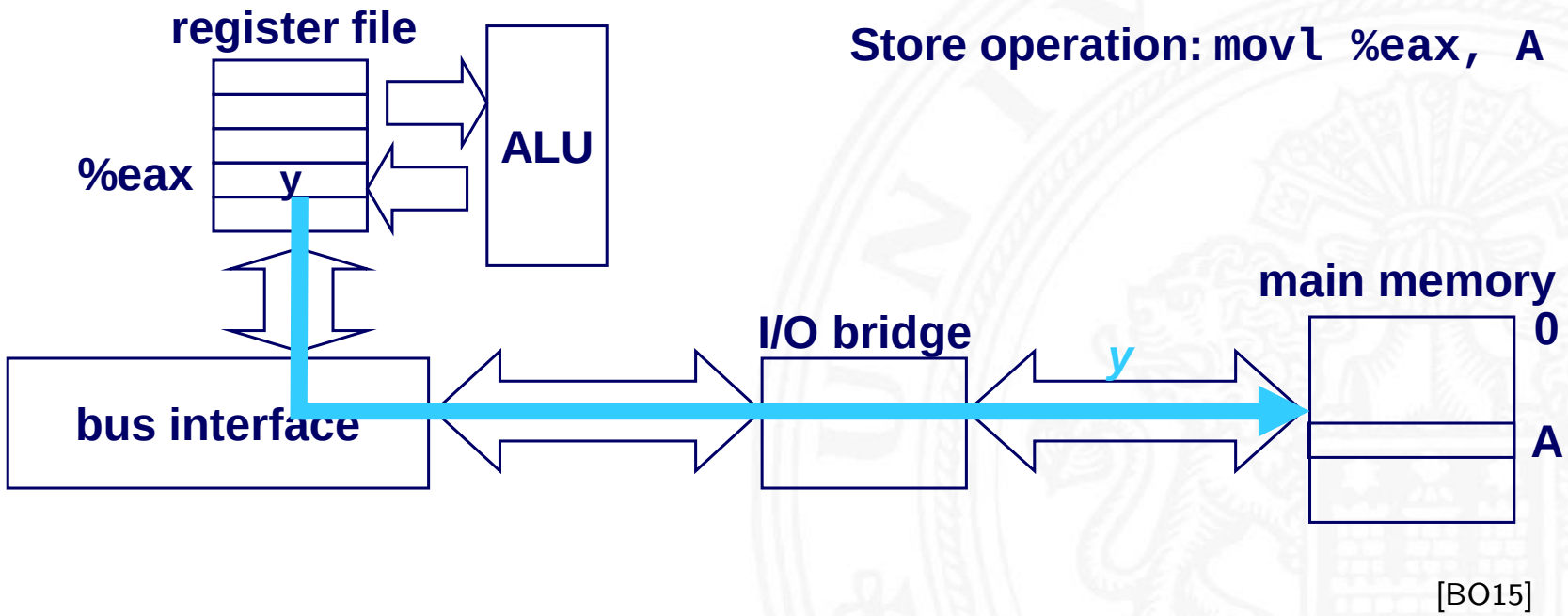
2.c –"– wartet auf Ankunft des Datenworts



[BO15]

schreibender Speicherzugriff (cont.)

3. CPU legt Datenwort y auf den Bus

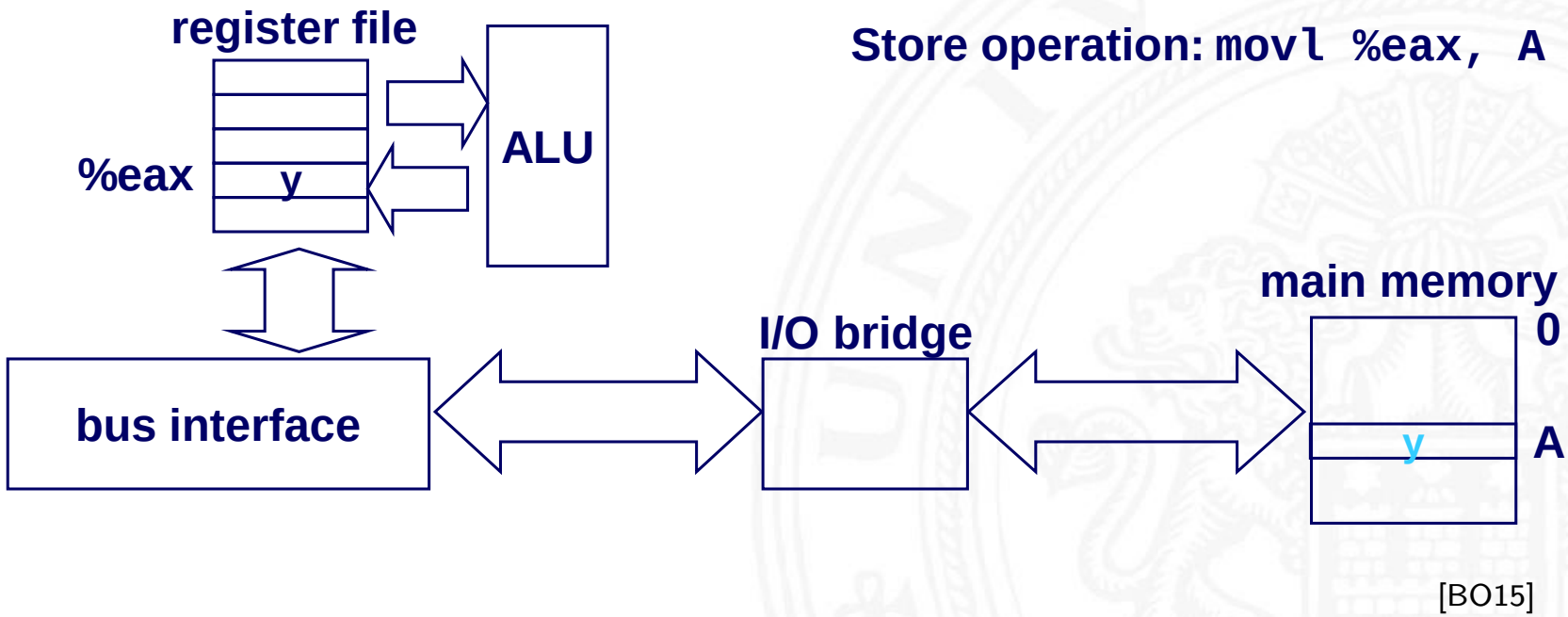


[BO15]

schreibender Speicherzugriff (cont.)

4.a Hauptspeicher liest Datenwort y vom Bus

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

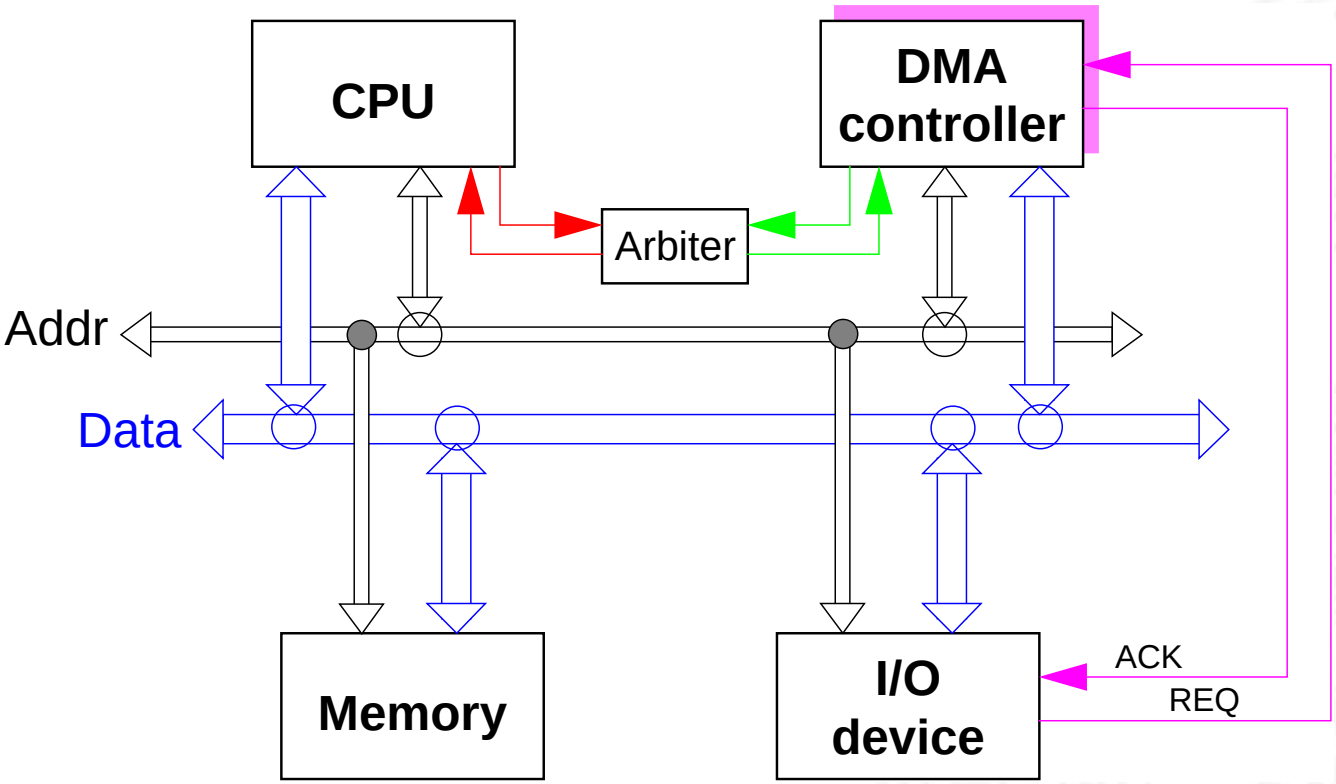


[BO15]

Speicheranbindung – DMA

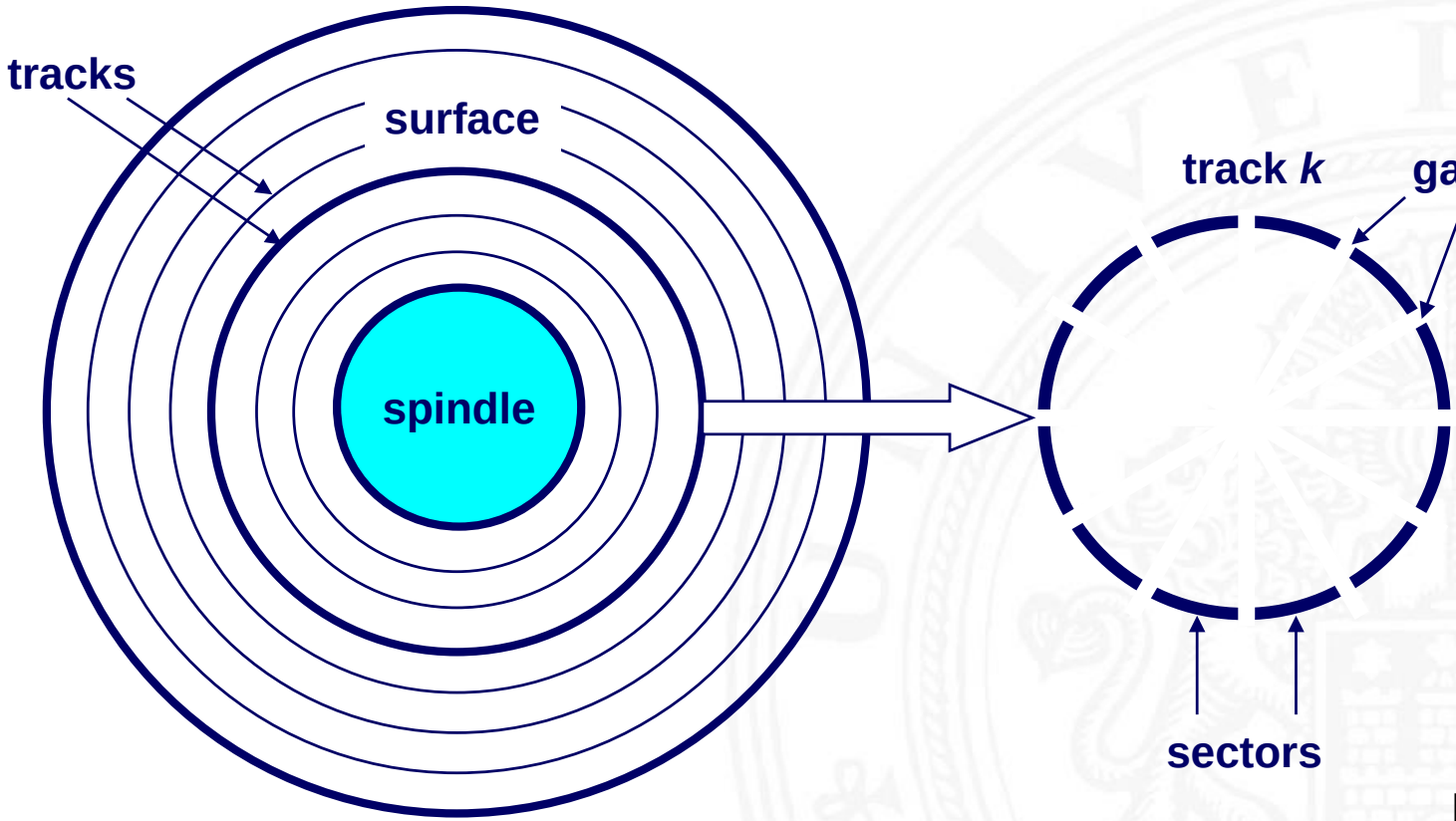
DMA – **D**irect **M**emory **A**ccess

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



Festplattengeometrie

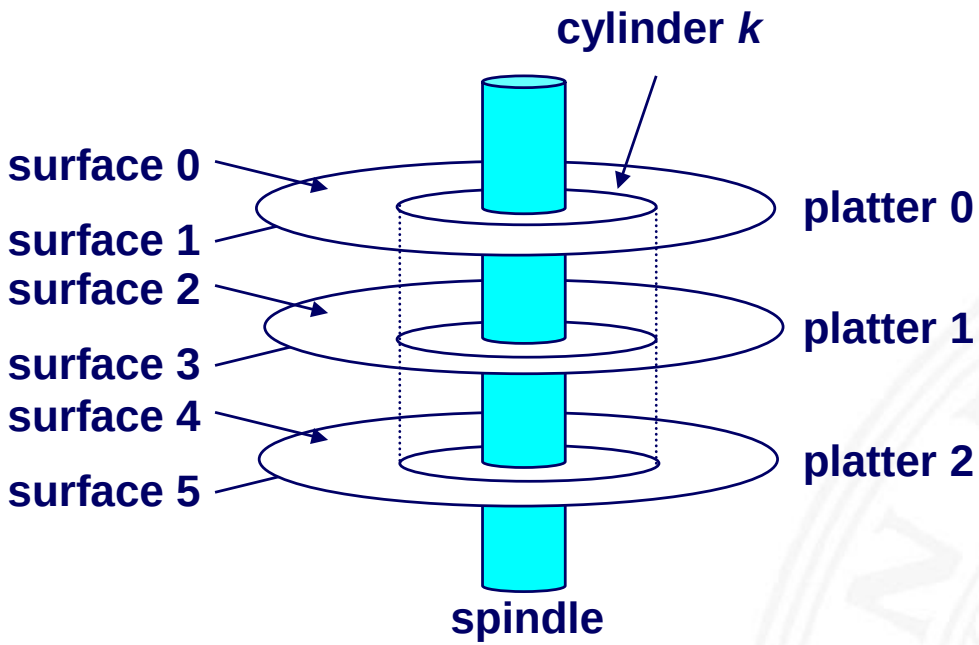
- Platten mit jeweils zwei Oberflächen („surfaces“)
- Spuren als konzentrische Ringe auf den Oberflächen („tracks“),
- jede Spur unterteil in Sektoren („sectors“), kurze Lücken („gaps“) dienen zur Synchronisierung



[BO15]

Festplattengeometrie (cont.)

- untereinander liegende Spuren (mehrerer Platten) bilden einen Zylinder



[BO15]

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

  - limitiert durch minimal noch detektierbare Magnetisierung
  - sowie durch Positionierungsgenauigkeit der Köpfe

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

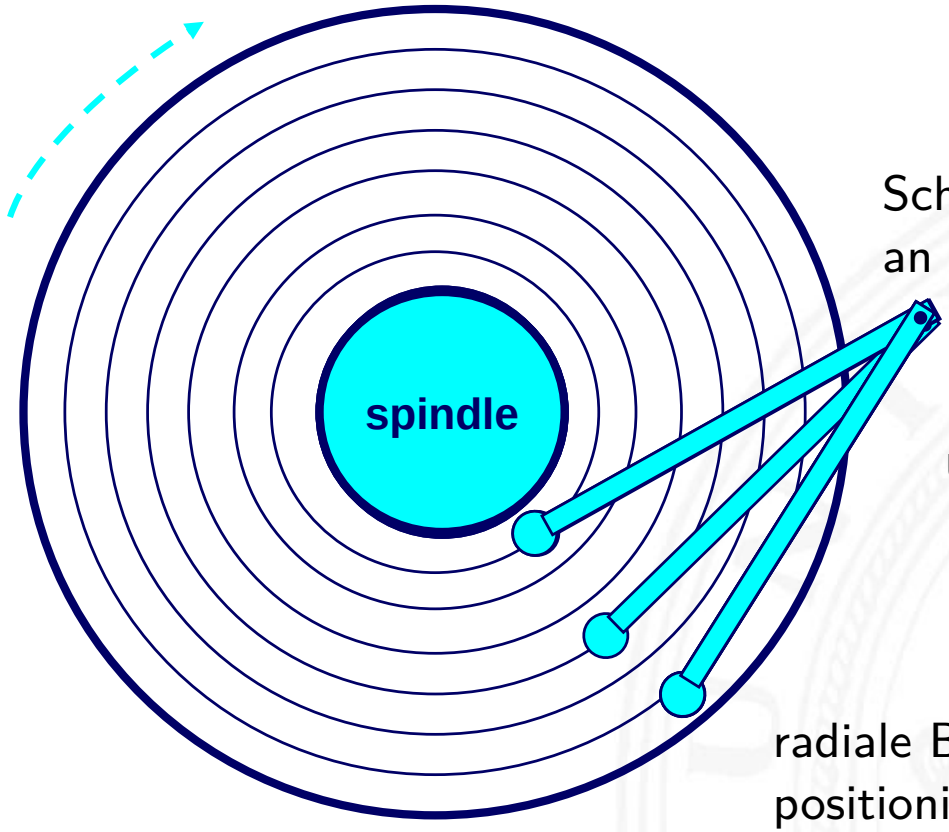
  $\Rightarrow$ Kapazität $= 512 \times 300 \times 20\,000 \times 2 \times 5$
  $= 30\,720\,000\,000 = 30{,}72\,\text{GB}$

  $\Rightarrow$ uraltes Modell

Festplatten-Operation

- Ansicht einer Platte

  Umdrehung mit konstanter Geschwindigkeit



Schreib/Lese Kopf ist an Arm befestigt

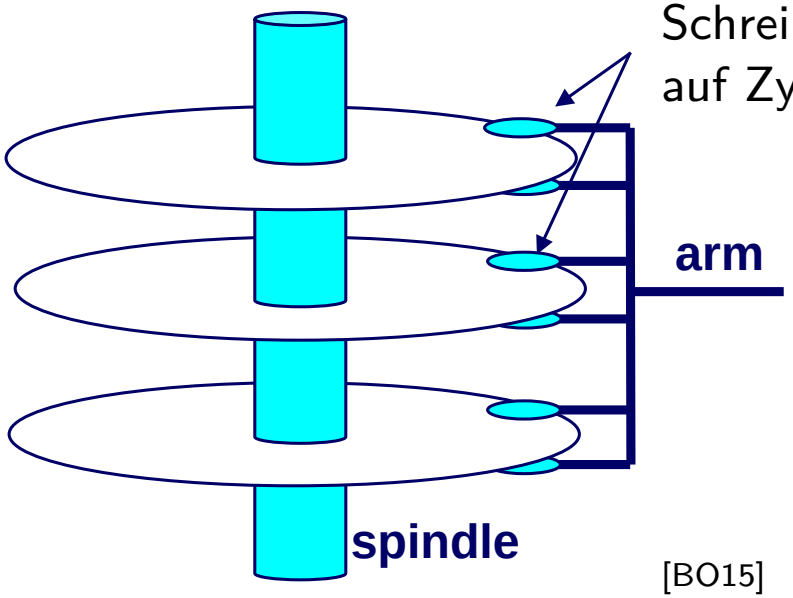
„fliegt“ auf Luftpolster über Plattenoberfläche

radiale Bewegung des Arms positioniert Kopf auf Spuren

[BO15]

Festplatten-Operation (cont.)

- Ansicht mehrerer Platten



Schreib/Lese Köpfe werden gemeinsam auf Zylindern positioniert

[BO15]

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
- $T_{avgRotationslatenz} = 1/2 \times 1/\text{RPMs} \times 60\,\text{Sek}$
- RPM „rotations per minute“
  - typische Drehzahlen: 5400 .. 7200 .. 10000 .. 15000 RPM
  - Desktop .. Server
  - Tradeoff: Geschwindigkeit vs. Leistungsaufnahme, Geräusch etc.

$\Rightarrow$ $T_{avgRotation} \approx 5.5\,\text{ms} \,..\, 2.0\,\text{ms}$

Transferzeit ($T_{avgTransfer}$)

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

Festplatten-Zugriffszeit (cont.)

Beispiel für Festplatten-Zugriffszeit

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

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

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

$\Rightarrow$ $T_{avgZugriff}$
$= 8\,\text{ms} + 4\,\text{ms} + 0{,}02\,\text{ms} \approx 12\,\text{ms}$

Festplatten-Zugriffszeit (cont.)

Fazit

- Zugriffszeit wird von Such- und Rotationslatenzzeit dominiert
- erstes Bit eines Sektors ist das „teuerste“, der Rest ist quasi umsonst
- typische Dauertransferraten aktueller Festplatten sind im Bereich 50 . . . 200 MB/s

- SRAM Zugriffszeit ist ca. 4 ns/64 bit, DRAM ca. 60 ns
- Kombination aus Zugriffszeit und Datentransfer
  - Festplatte ist ca. 40 000 mal langsamer als SRAM
  - 2 500 mal langsamer als DRAM

$\Rightarrow$ hoher Aufwand in Hardware und Betriebssystem, um dieses Problem (weitgehend) zu vermeiden

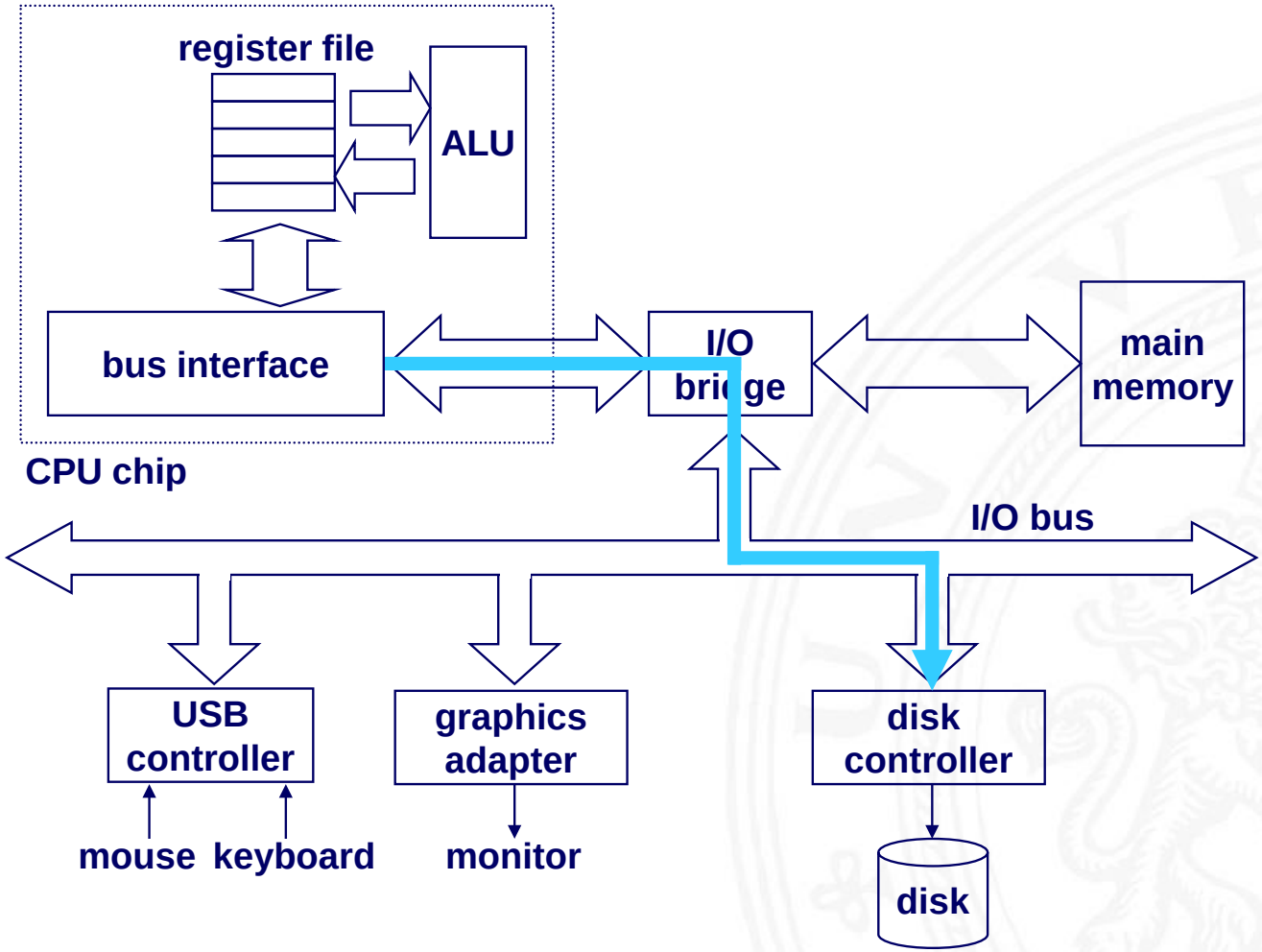
Logische Festplattenblöcke

- abstrakte Benutzersicht der komplexen Sektorengeometrie
  - verfügbare Sektoren werden als Sequenz logischer Blöcke der Größe b modelliert (0,1,2,..,n)
  - typische Blockgröße war jahrzehntelang $b = 512$ Bytes
  - neuere Festplatten zunehmend mit $b = 4096$ Bytes
- 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 zwischen „formatierter-“ und „maximaler Kapazität“

Lesen eines Festplattensektors

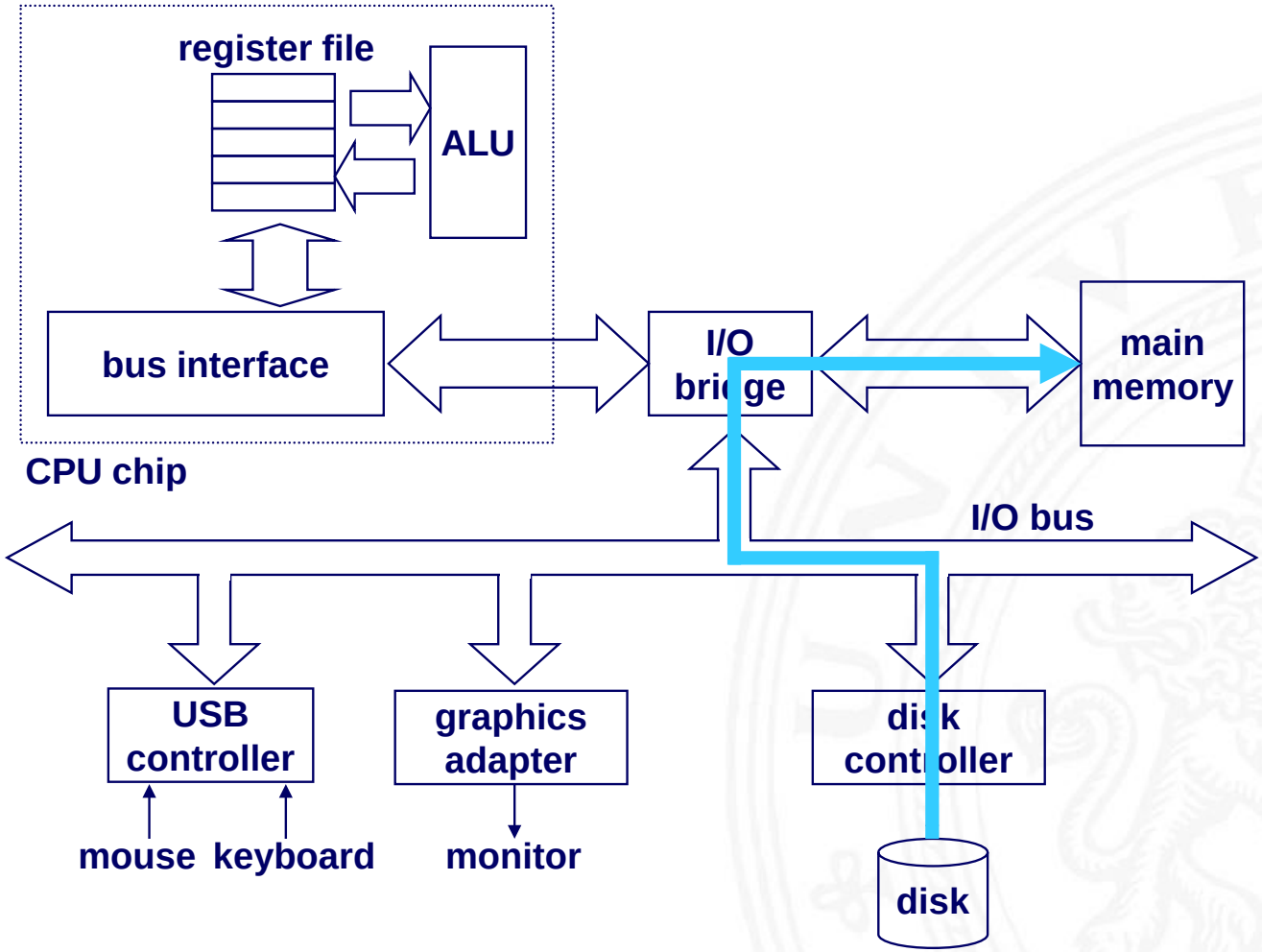
1. CPU initiiert Lesevorgang von Festplatte
   - schreibt auf Port (Adresse) des Festplattencontrollers: Befehl, logische Blocknummer, Zielspeicheradresse



[BO15]

Lesen eines Festplattensektors (cont.)

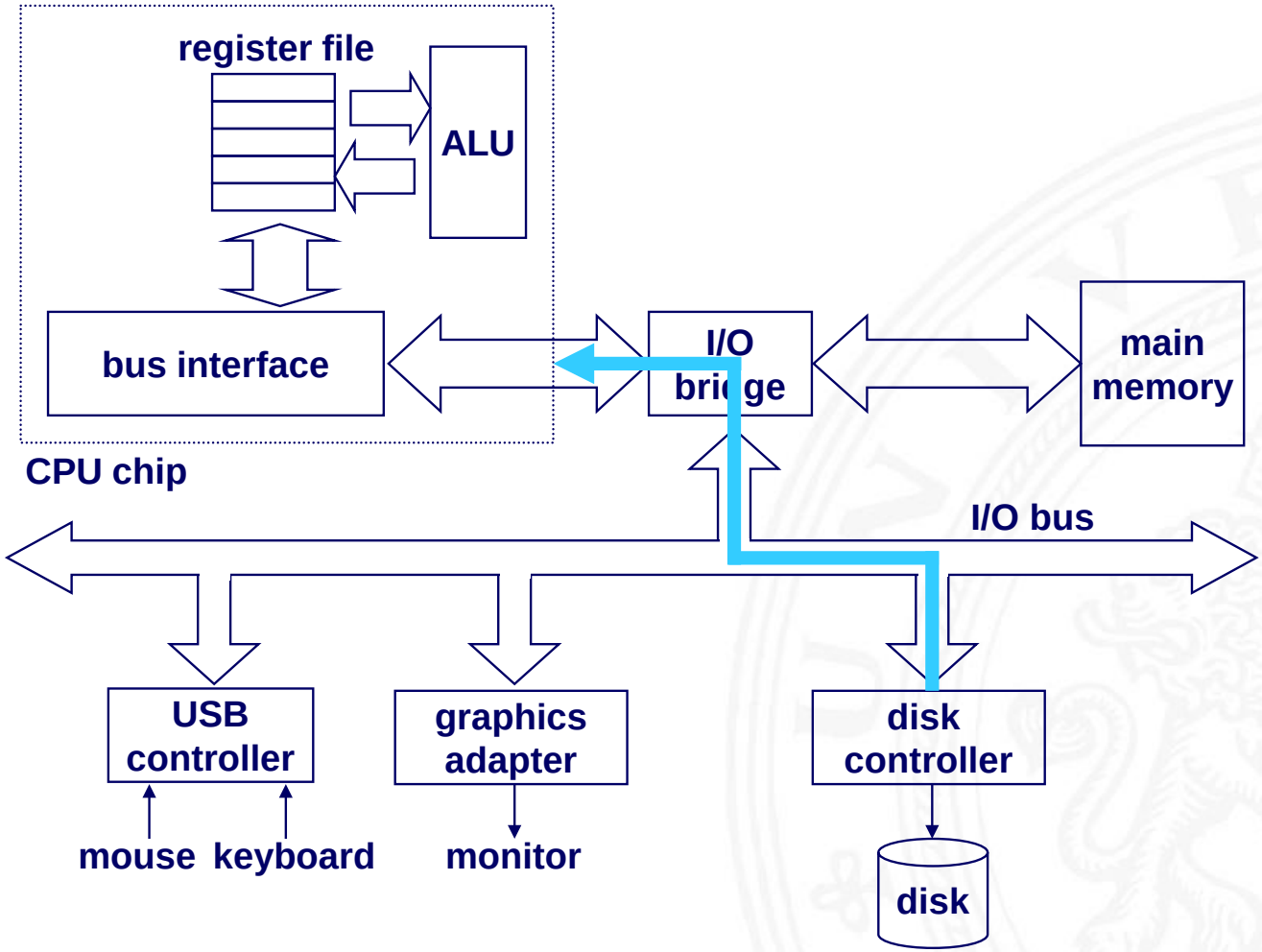
2. Festplattencontroller liest den Sektor aus
3. –"– führt DMA-Zugriff auf Hauptspeicher aus



[BO15]

Lesen eines Festplattensektors (cont.)

4. Festplattencontroller löst Interrupt aus



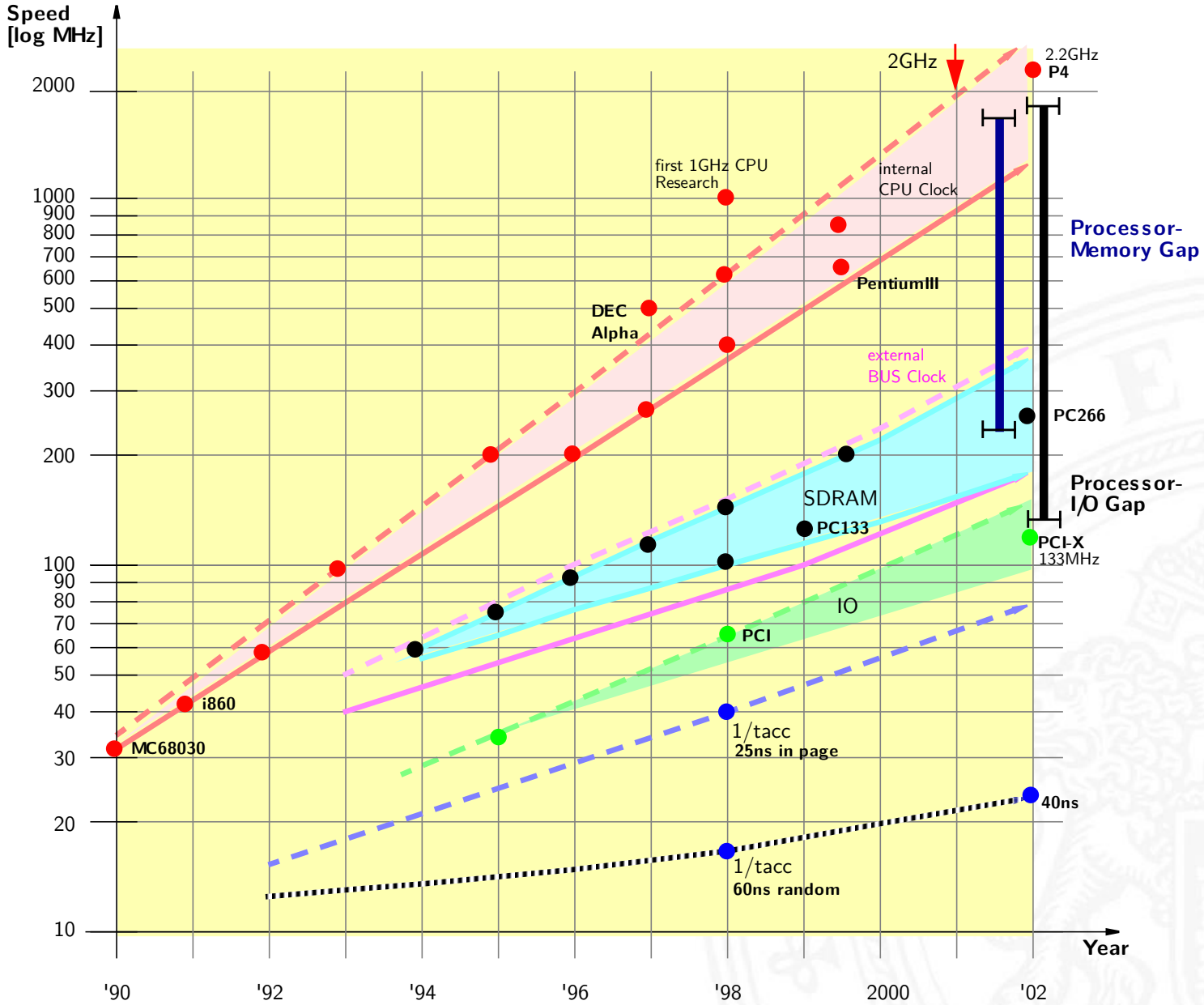
[BO15]

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	SSD
Latenz	8 ns	4 ms	0,2/0,4 ms
Bandbreite	$\approx$ 25,6 GB/sec (pro Kanal, bis 4)	$\approx$ 750 MB/sec typ.: $<$ 300	500
Kosten/GB	5 €	4 ct. 1 TB, 40 €	70 ct.

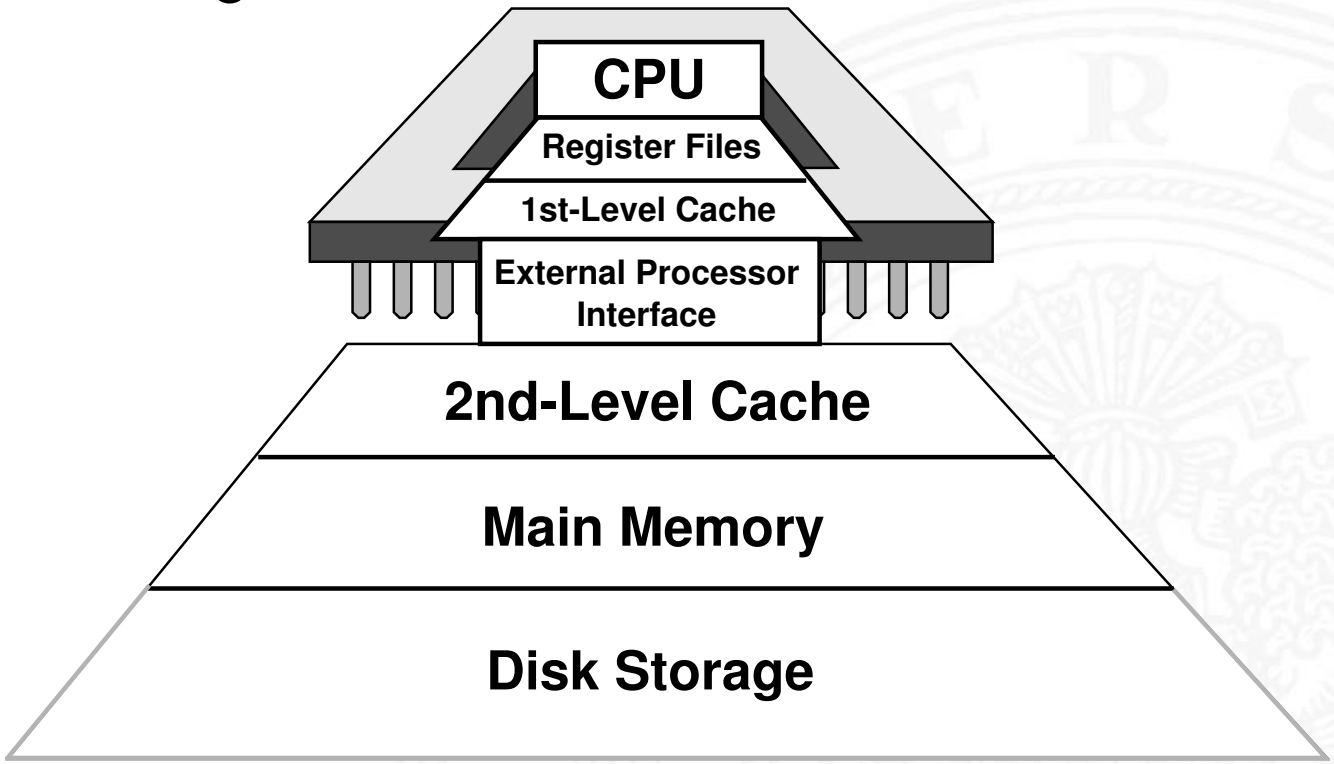
Prozessor-Memory Gap



Speicherhierarchie

Motivation

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



Speicherhierarchie (cont.)

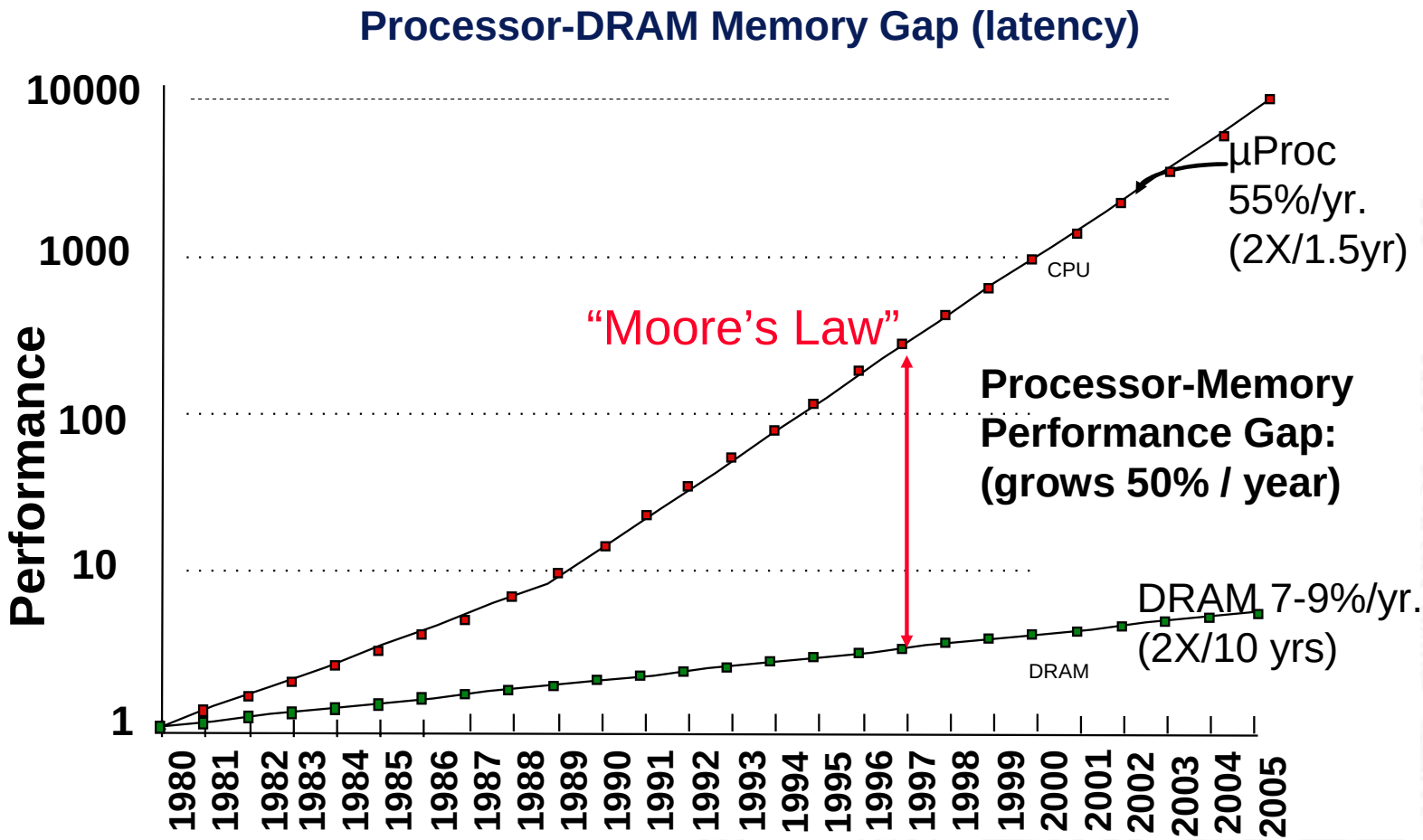
- schnelle vs. langsame Speichertechnologie
  schnell : hohe Kosten/Byte geringe Kapazität
  langsam : geringe –"– hohe –"–
- wachsender Abstand zwischen CPU und Speichergeschwindigkeit
  - Prozessor läuft mit einigen GHz Takt
  - Register können mithalten, aber nur einige KByte Kapazität
  - DRAM braucht 60...100 ns für Zugriff: 100 × langsamer
  - Festplatte braucht 10 ms für Zugriff: 1 000 000 × langsamer
- Lokalität der Programme wichtig
  - aufeinanderfolgende Speicherzugriffe sind meistens „lokal“
  - gut geschriebene Programme haben meist eine gute Lokalität

⇒ Motivation für spezielle Organisation von Speichersystemen
Speicherhierarchie

Verwaltung der Speicherhierarchie

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

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



[PH16b]

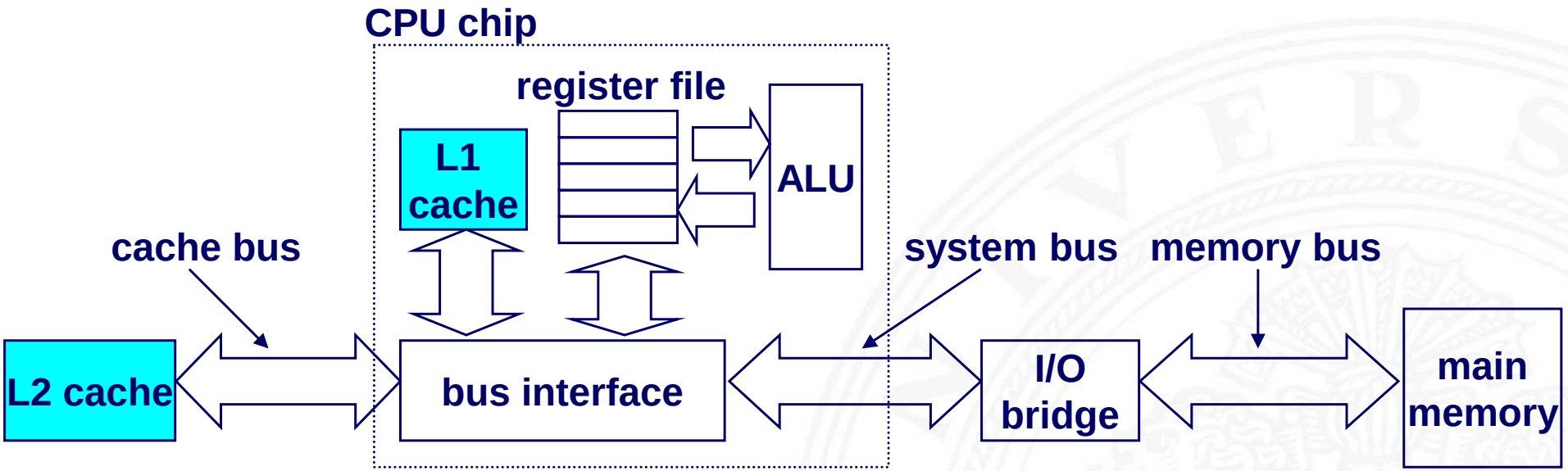
⇒ Cache als schneller Zwischenspeicher zum Hauptspeicher

Cache (cont.)

- technische Realisierung: SRAM
- transparenter Speicher
  - Cache ist für den Programmierer nicht sichtbar!
  - wird durch Hardware verwaltet
- ggf. getrennte Caches für Befehle und Daten
- enthält Hauptspeicherblöcke mit erhöhter Zugriffswahrscheinlichkeit
- basiert auf Prinzip der Lokalität von Speicherzugriffen durch ein laufendes Programm
  - ca. 80% der Zugriffe greifen auf 20% der Adressen zu
  - manchmal auch 90% / 10% oder noch besser
- `https://de.wikipedia.org/wiki/Cache`
  `https://en.wikipedia.org/wiki/Cache`

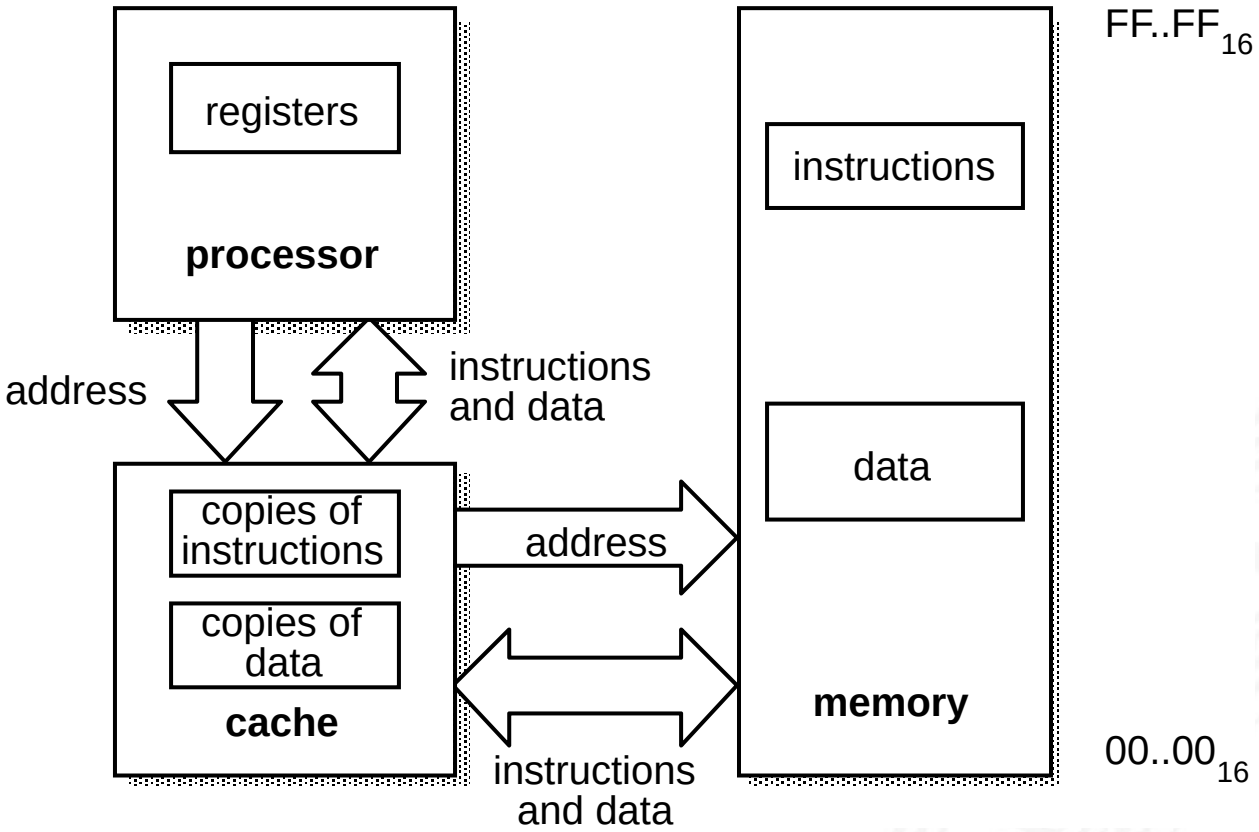
Cache (cont.)

- CPU referenziert Adresse
  - parallele Suche in L1 (level 1), L2... und Hauptspeicher
  - erfolgreiche Suche liefert Datum, Abbruch laufender Suchen



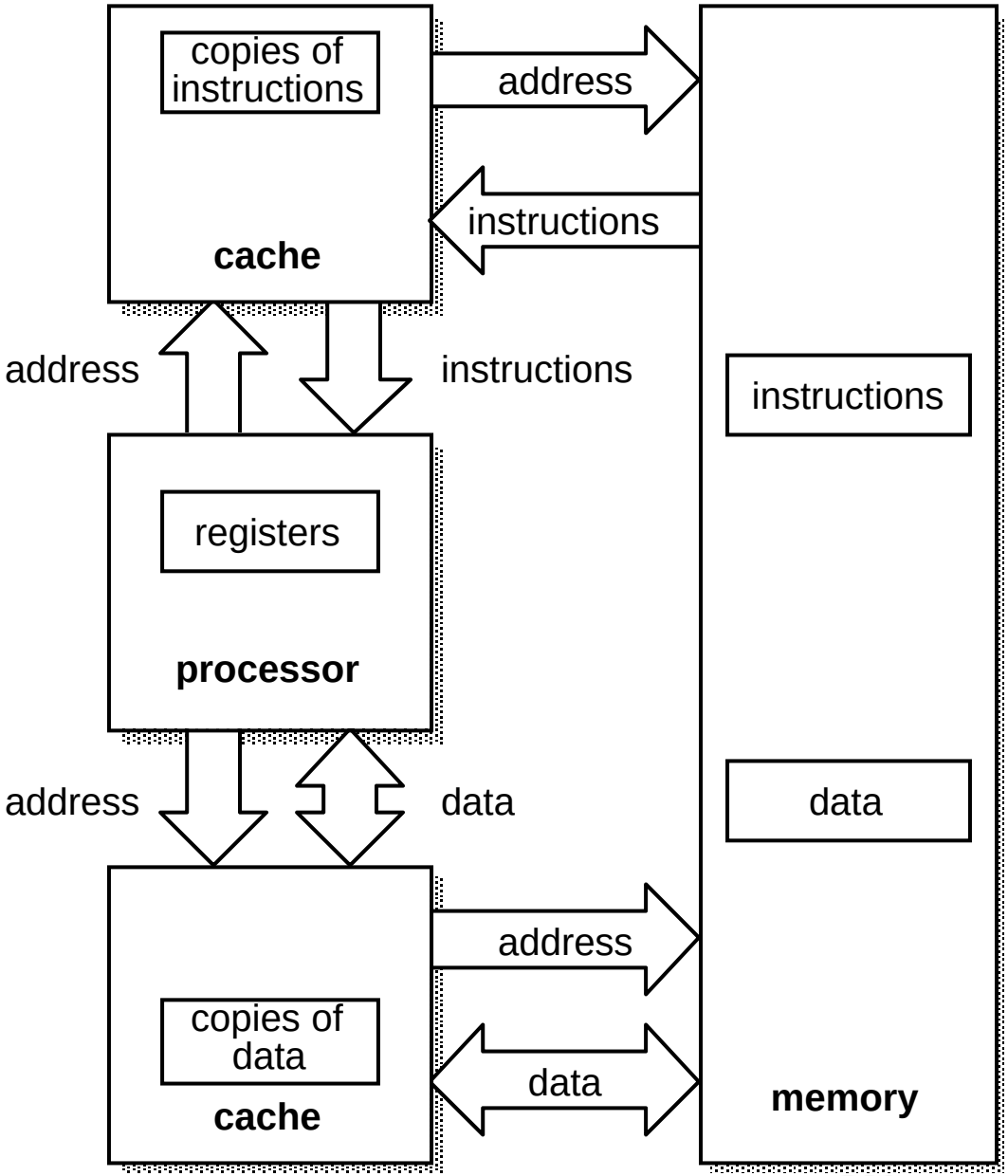
[BO15]

gemeinsamer Cache / „unified Cache“



[Fur00]

separate Instruction-/Data Caches



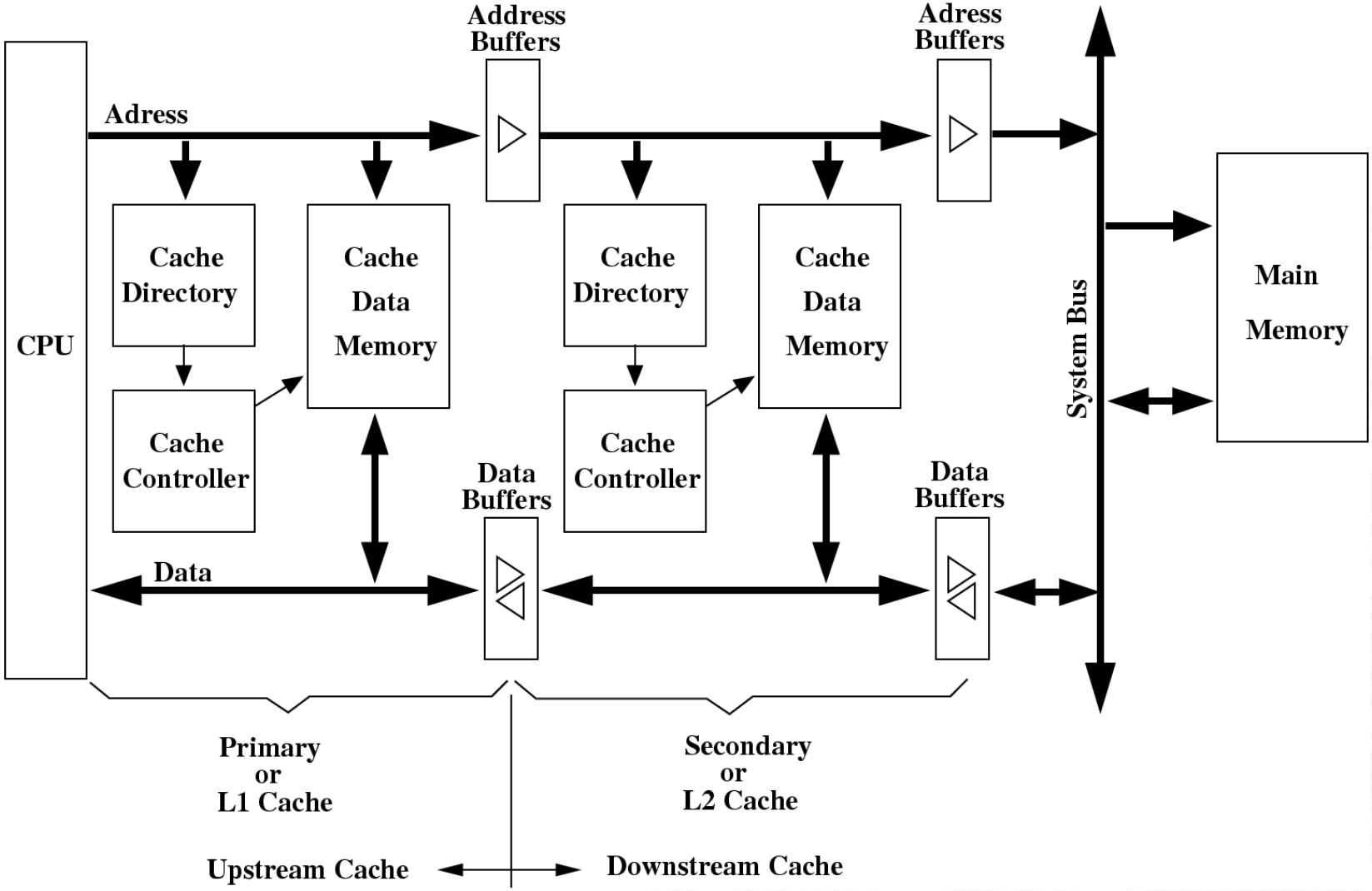
$FF..FF_{16}$

$00..00_{16}$

[Fur00]

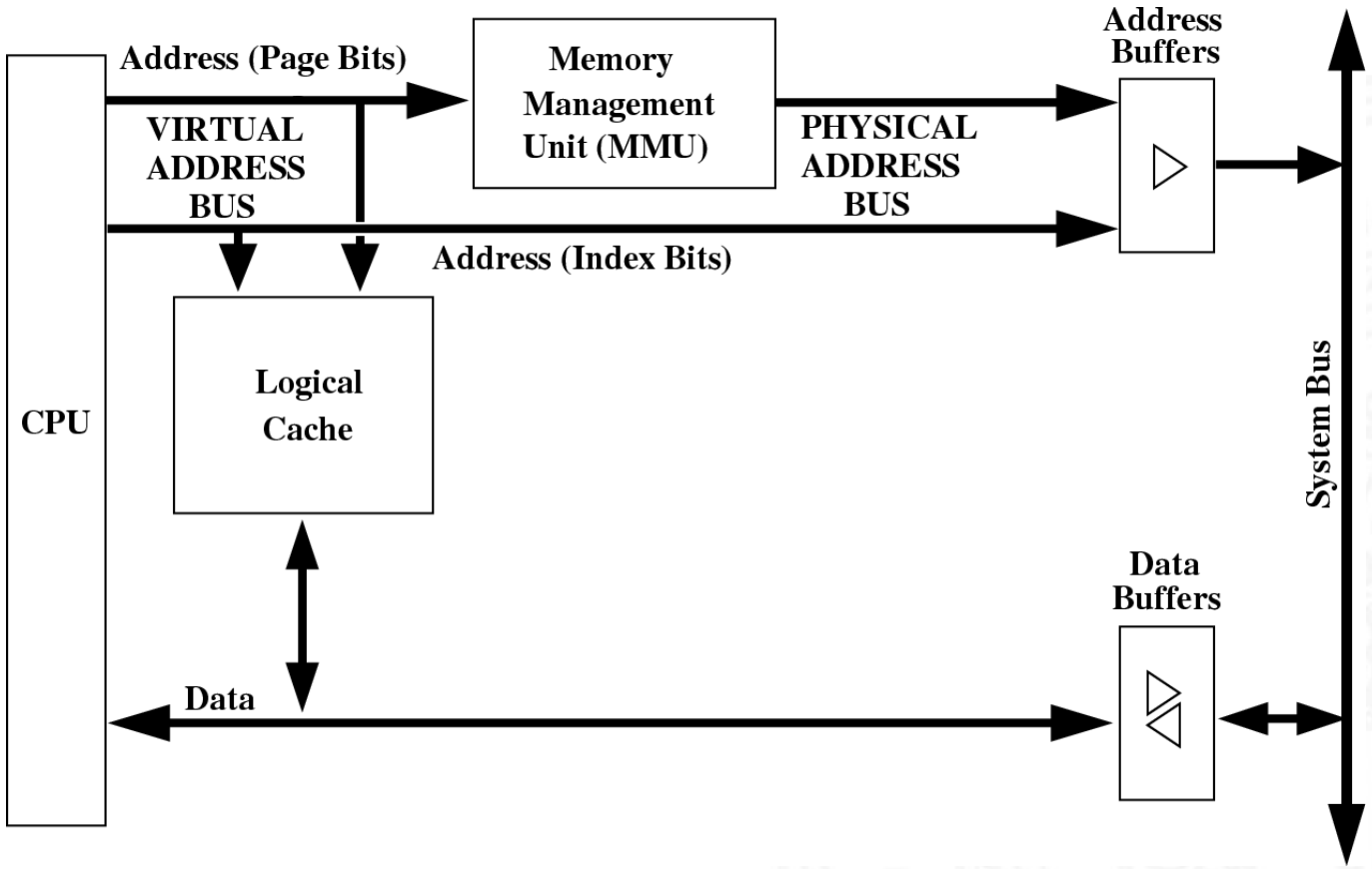
Cache – Position

- First- und Second-Level Cache



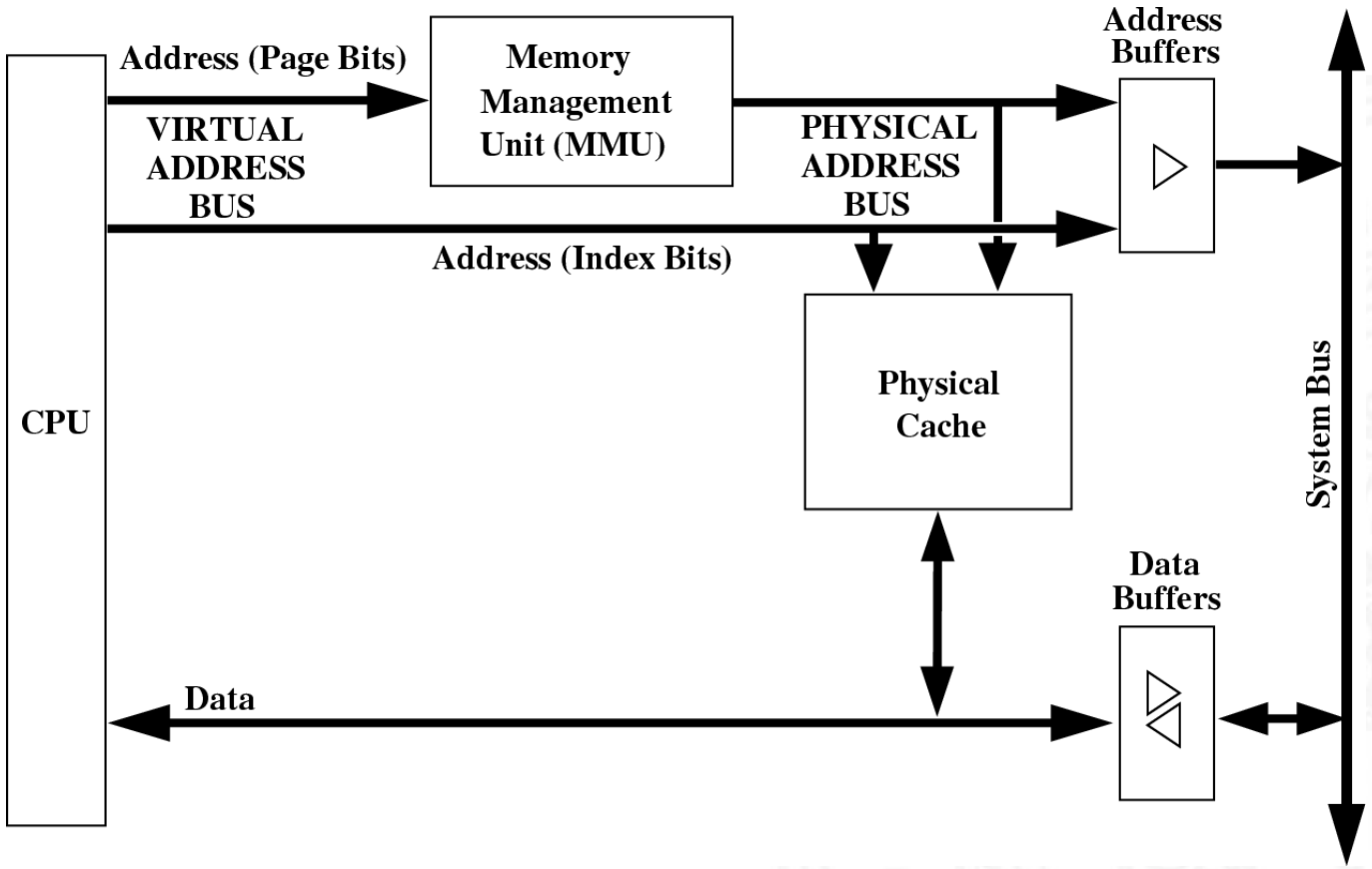
Cache – Position (cont.)

- 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

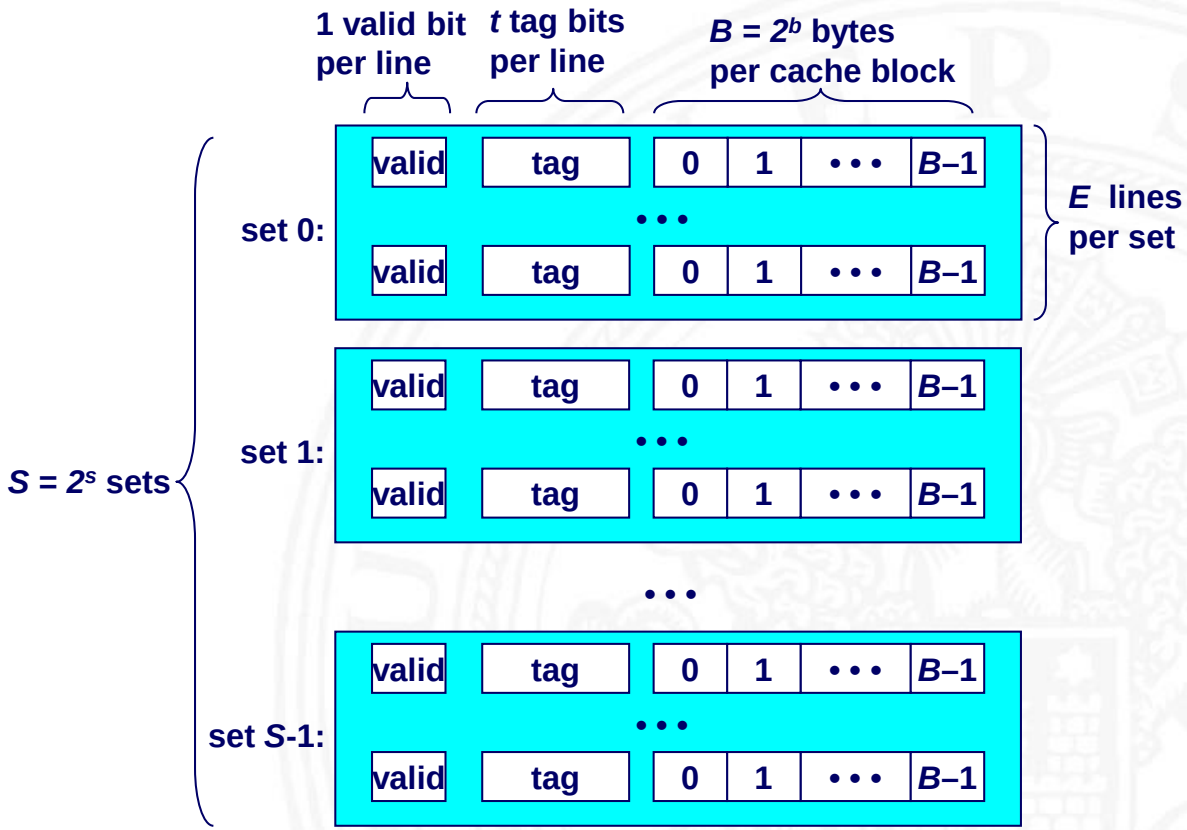
Treffer (Hit)		Zugriff auf Datum, ist bereits im Cache
Fehler (Miss)		–"– ist nicht –"–
Treffer-Rate	R_{Hit}	Wahrscheinlichkeit, Datum ist im Cache
Fehler-Rate	R_{Miss}	$1 - R_{Hit}$
Hit-Time	T_{Hit}	Zeit, bis Datum bei Treffer geliefert wird
Miss-Penalty	T_{Miss}	zusätzlich benötigte Zeit bei Fehler

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

  $T_{Hit} = 1$ Takt, $T_{Miss} = 20$ Takte, $R_{Miss} = 5\,\%$
  $\Rightarrow$ Mittlere Speicherzugriffszeit $= 2$ Takte

Cache Organisation

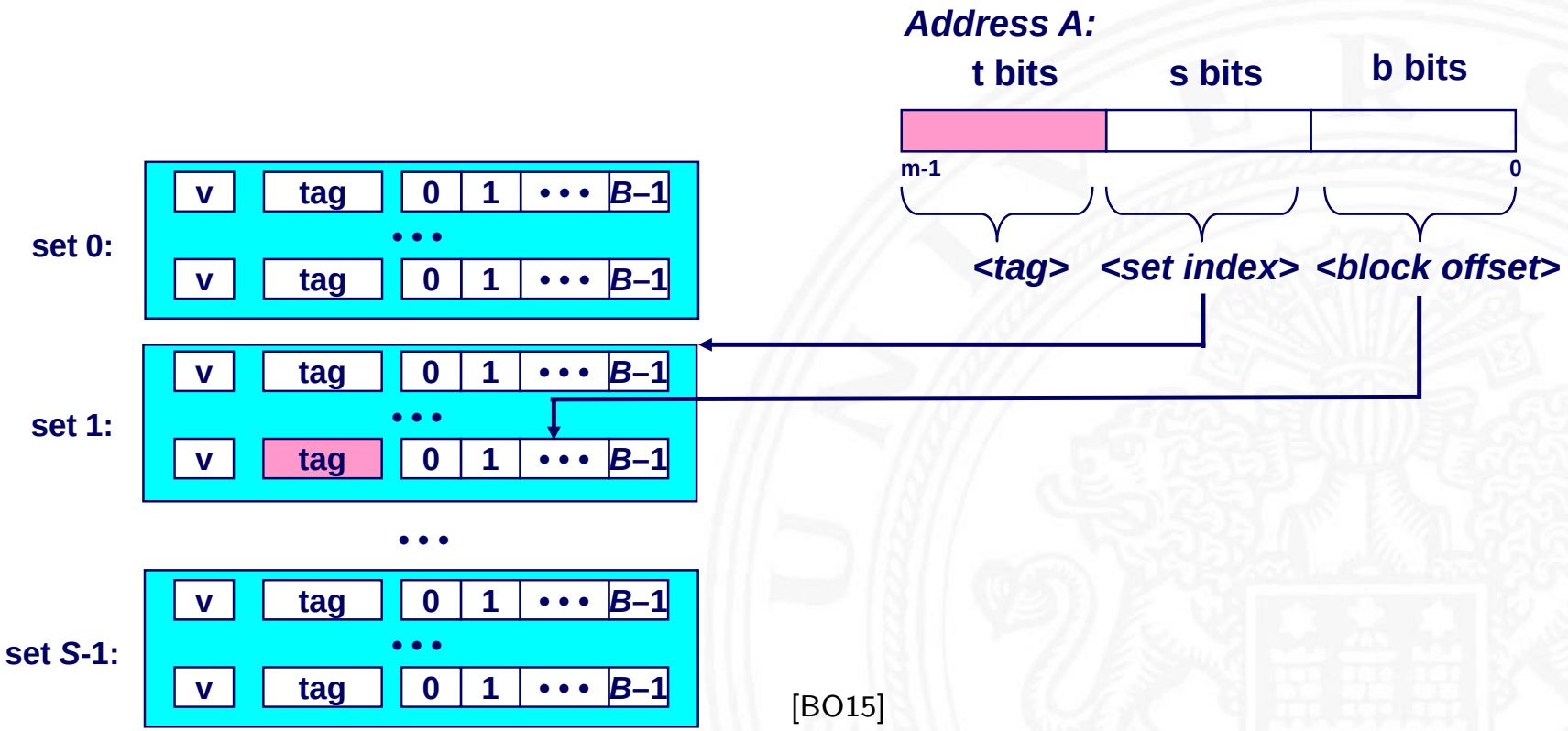
- Cache ist ein Array von Speicher-Bereichen („sets“)
- jeder Bereich enthält eine oder mehrere Zeilen
- jede Zeile enthält einen Datenblock
- jeder Block enthält mehrere Byte



[BO15]

Adressierung von Caches

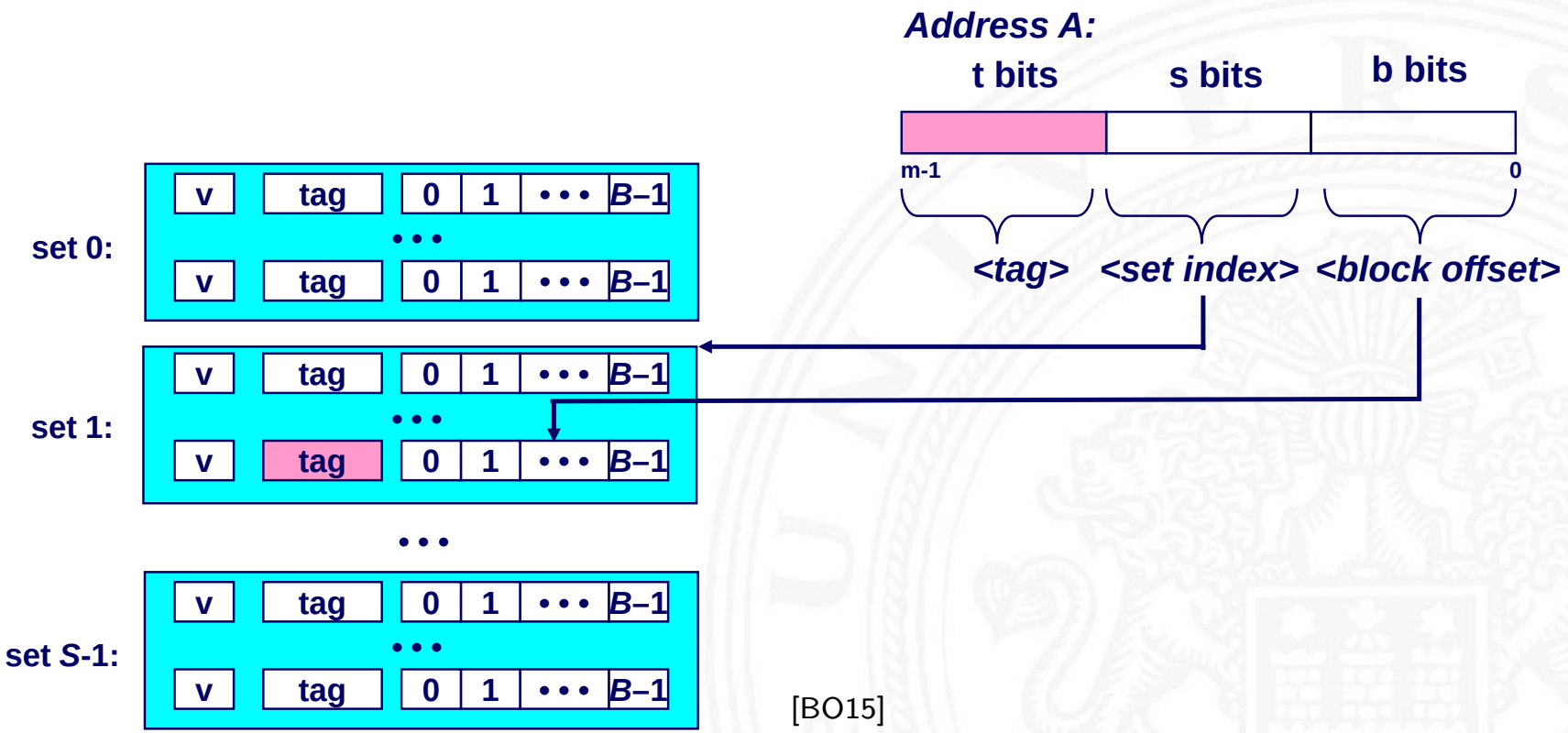
- Adressteil $\langle set\ index \rangle$ von A bestimmt Bereich („set“)
- Adresse A ist im Cache, wenn
  1. Cache-Zeile ist als gültig markiert („valid“)
  2. Adressteil $\langle tag \rangle$ von A = „tag“ Bits des Bereichs



[BO15]

Adressierung von Caches (cont.)

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



[BO15]

Cache – Organisation

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

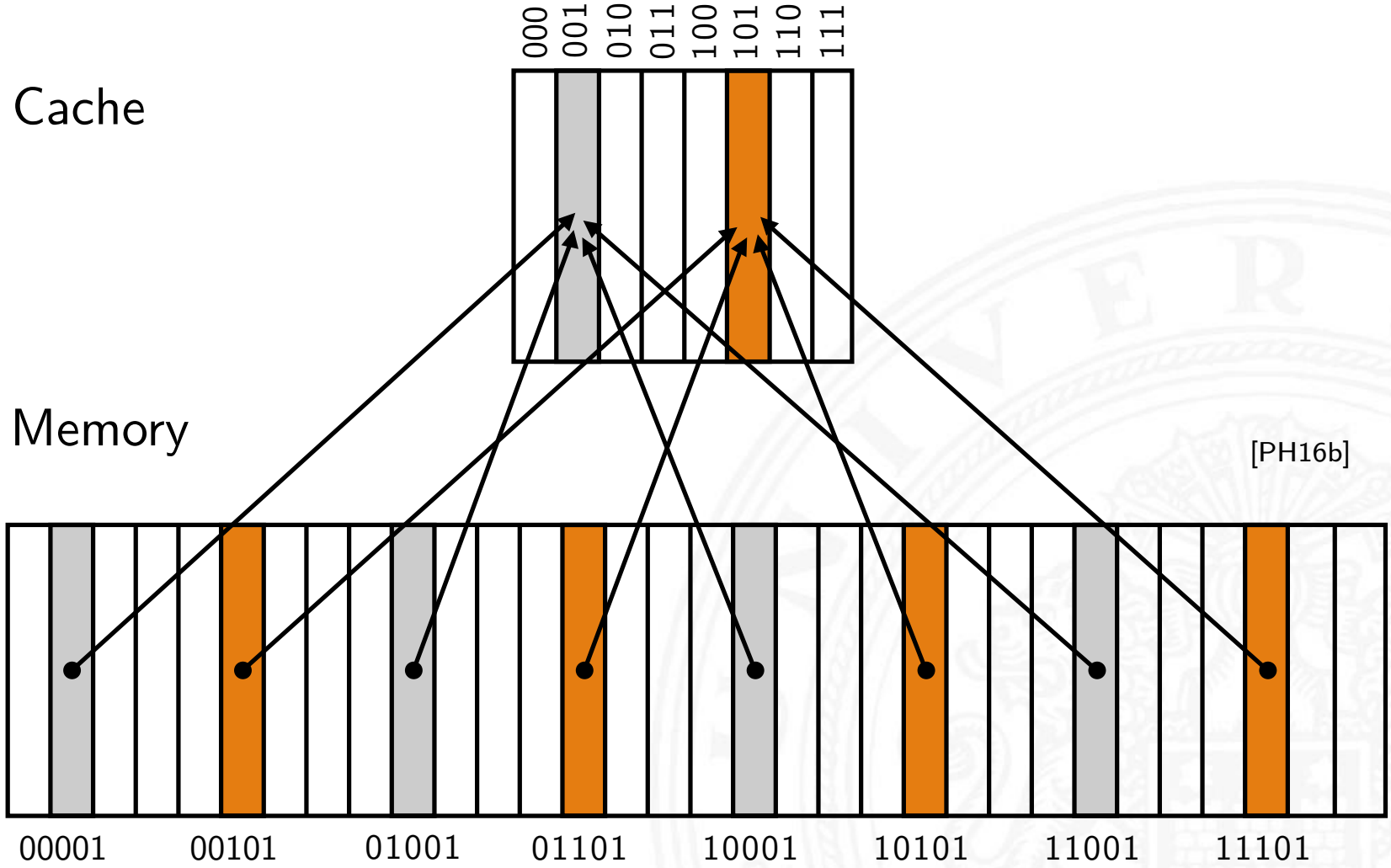
direkt abgebildet / direct mapped jeder Speicheradresse ist genau eine Cache-Speicherzelle zugeordnet

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

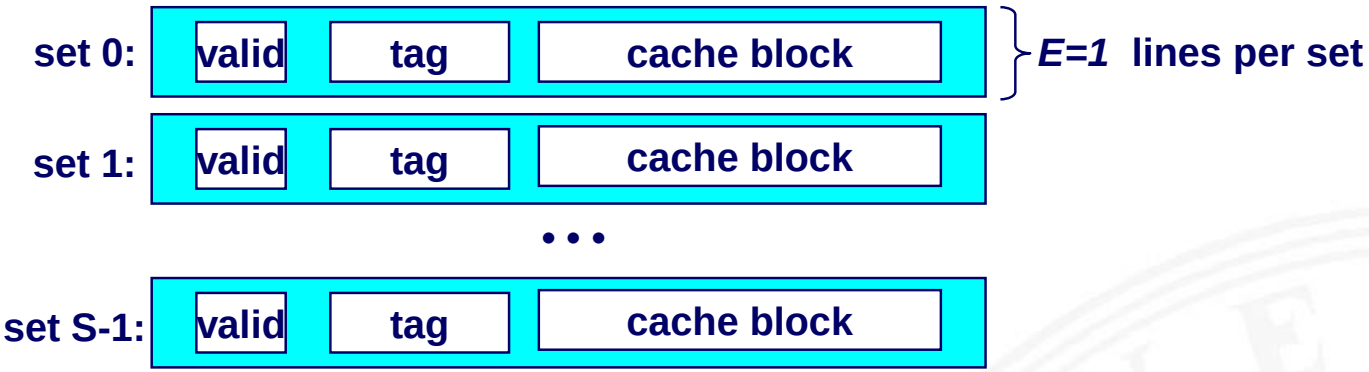
Cache: direkt abgebildet / „direct mapped“

- 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)



[BO15]

\+ einfachste Cache-Art

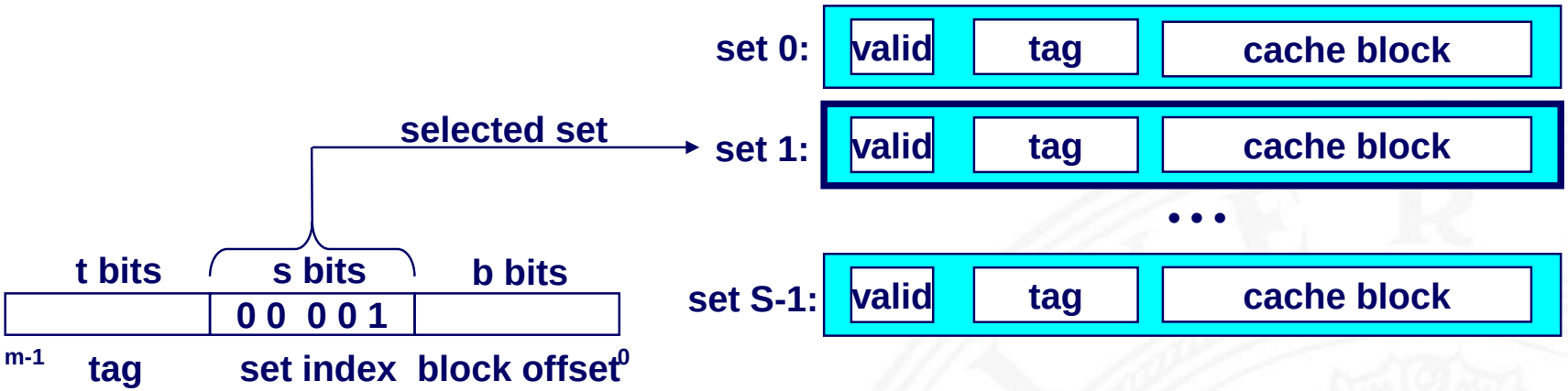
\+ große Caches möglich

– Effizienz, z.B. Zugriffe auf A, $A + n \cdot S \ldots$
$\Rightarrow$ „*Cache Thrashing*“

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

Zugriff auf direkt abgebildete Caches

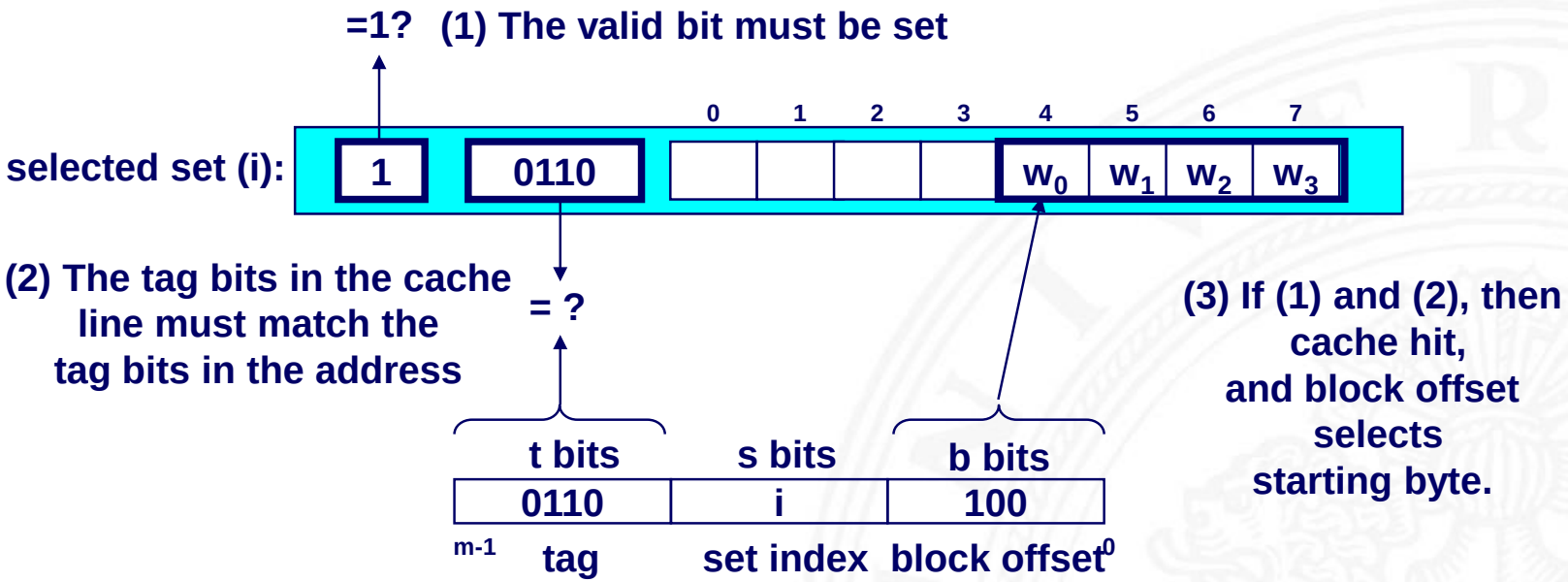
1. Bereichsauswahl durch Bits ⟨*set index*⟩



[BO15]

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

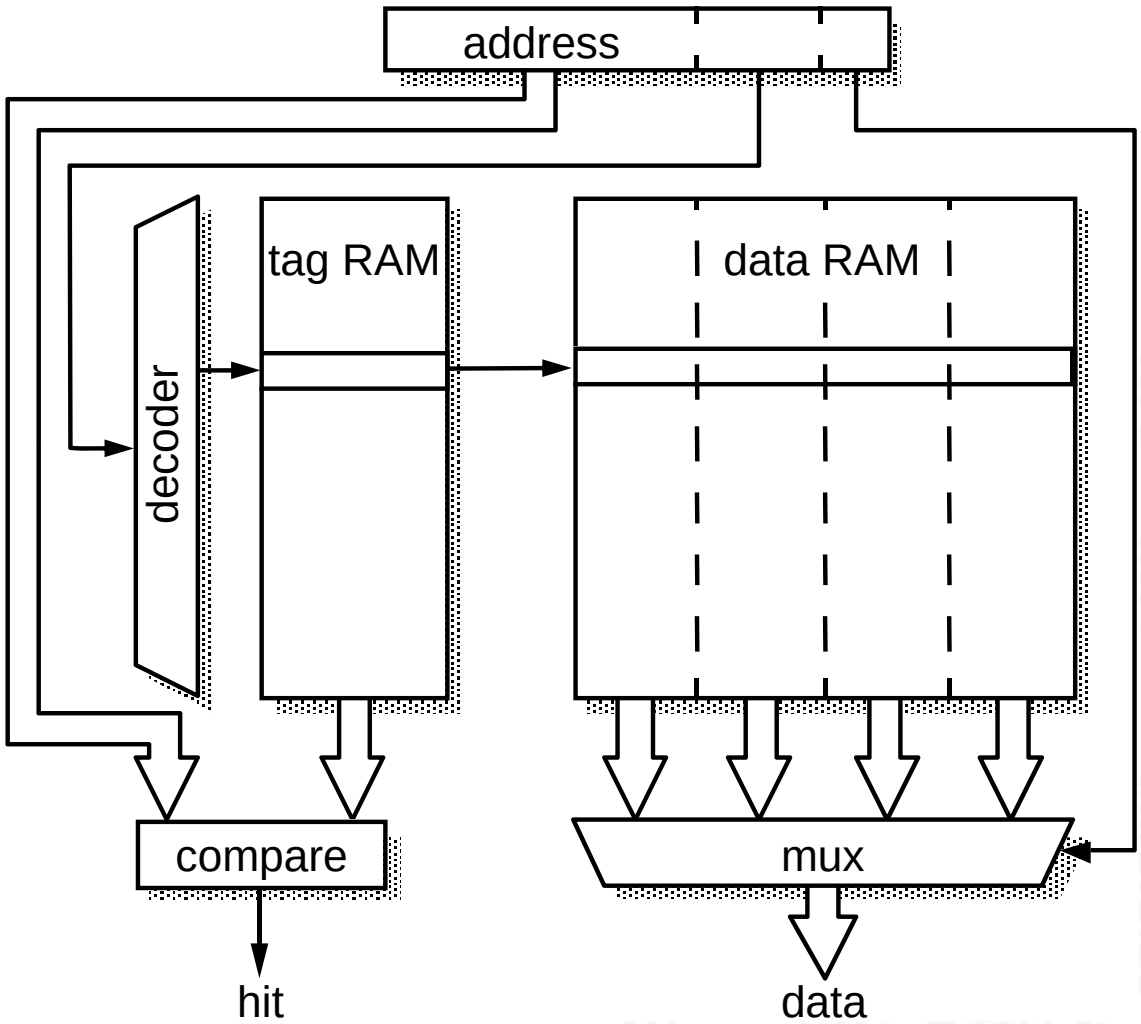
2. ⟨*valid*⟩: sind die Daten gültig?
3. „Line matching“: stimmt ⟨*tag*⟩ überein?
4. Wortselektion extrahiert Wort unter Offset ⟨*block offset*⟩



[BO15]

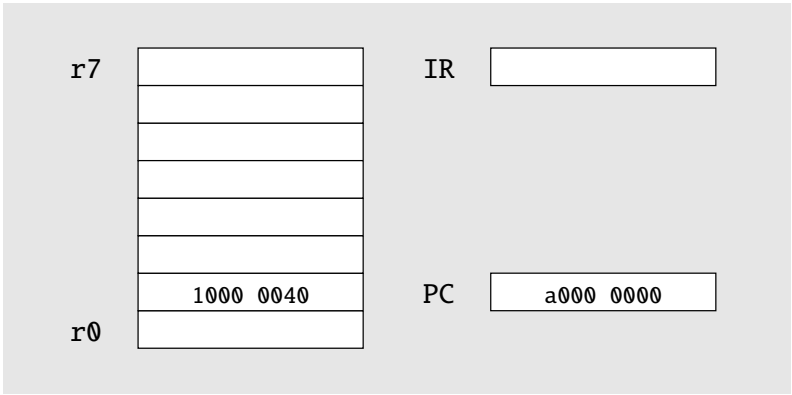
Cache: direkt abgebildet / „direct mapped“ (cont.)

Prinzip



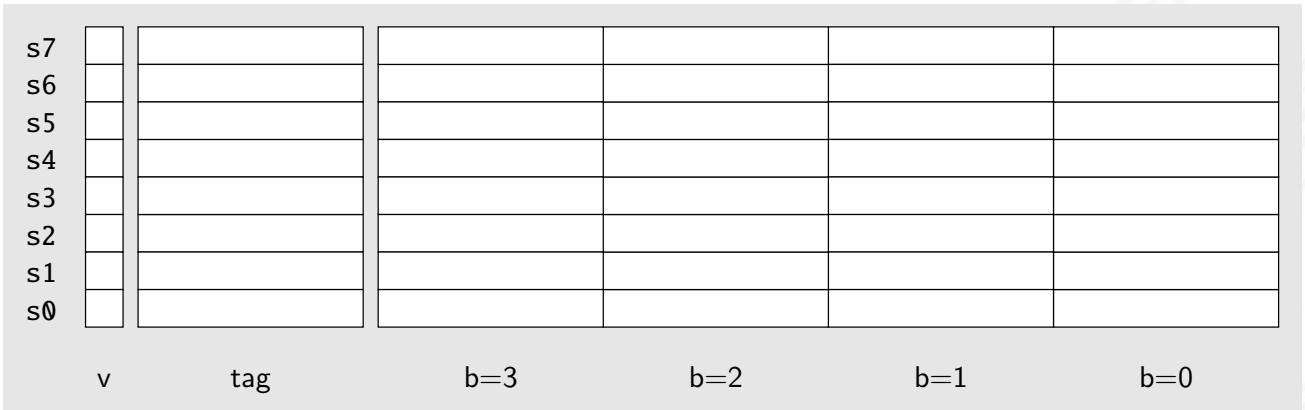
[Fur00]

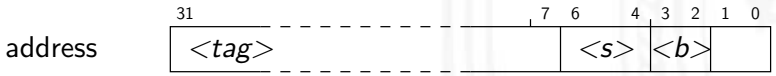
Direct mapped cache: Beispiel – leer



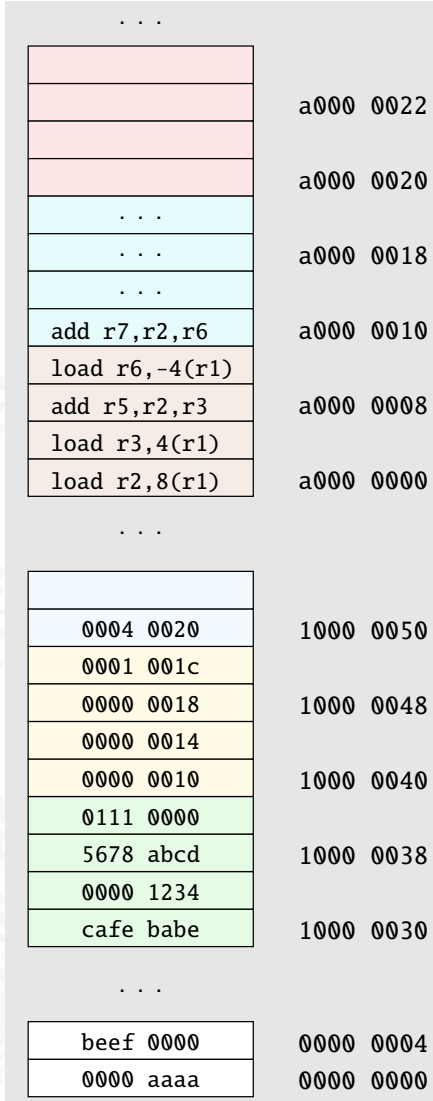
CPU

Cache

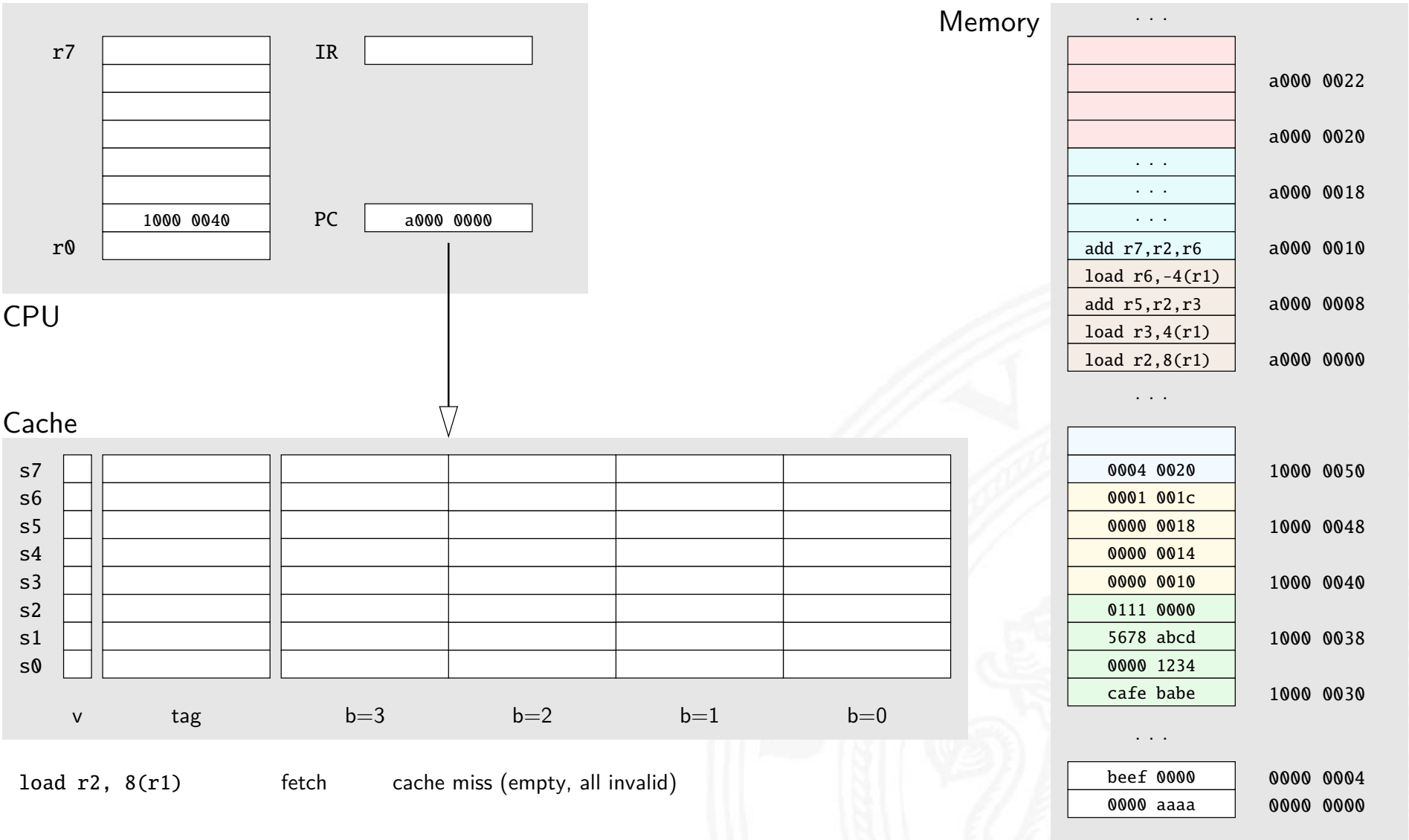




Memory

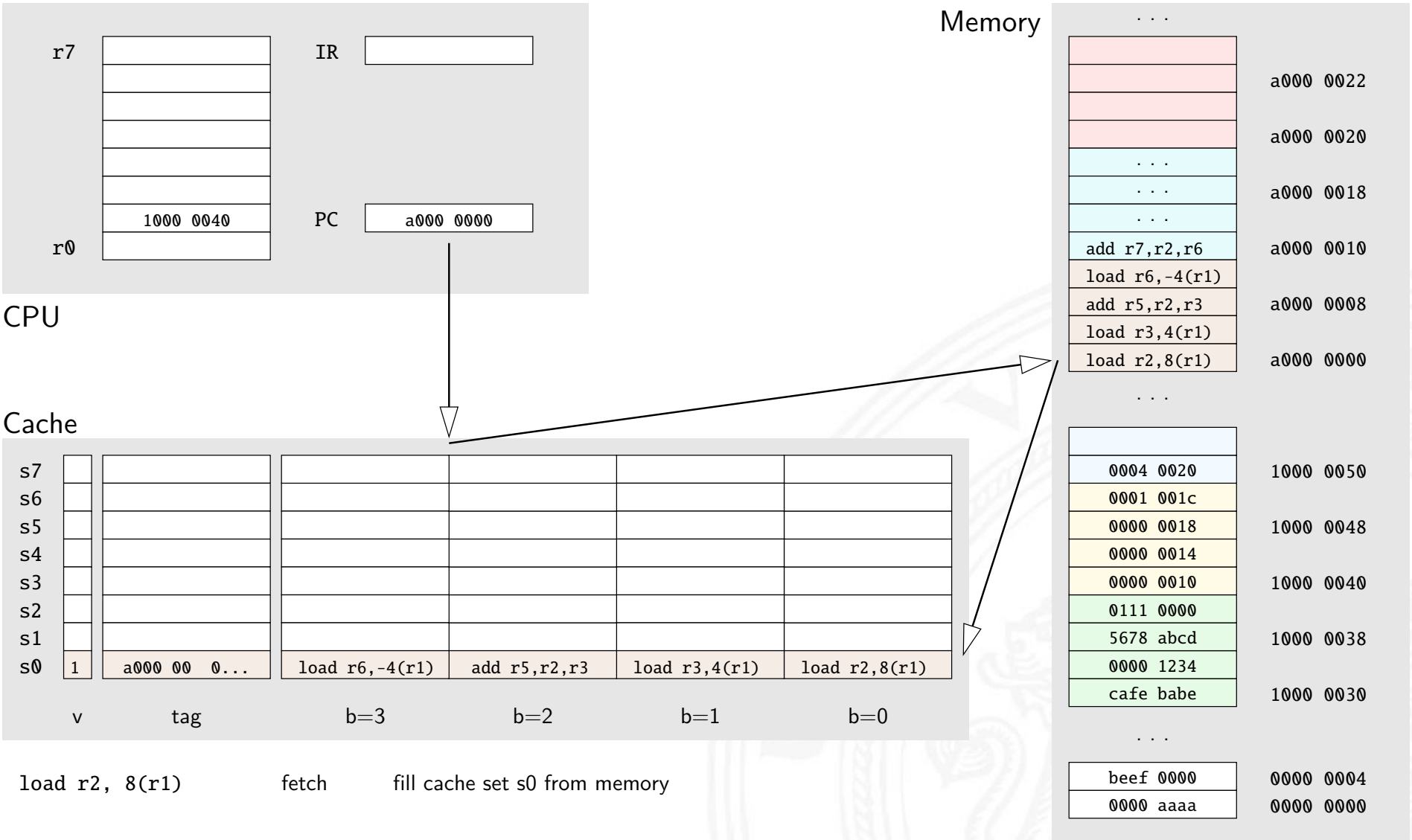


Direct mapped cache: Beispiel – fetch miss



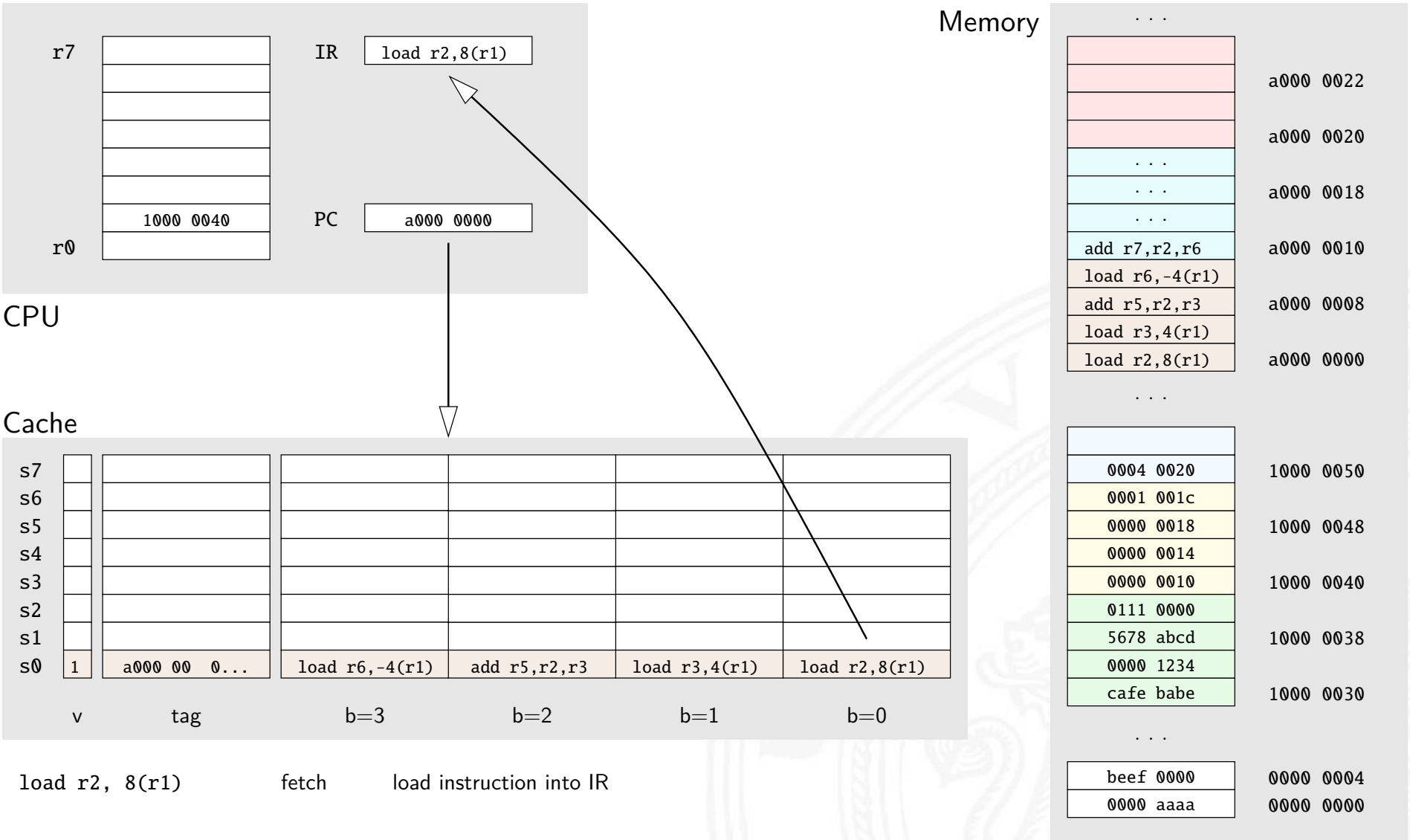
`load r2, 8(r1)` fetch cache miss (empty, all invalid)

Direct mapped cache: Beispiel – fetch fill



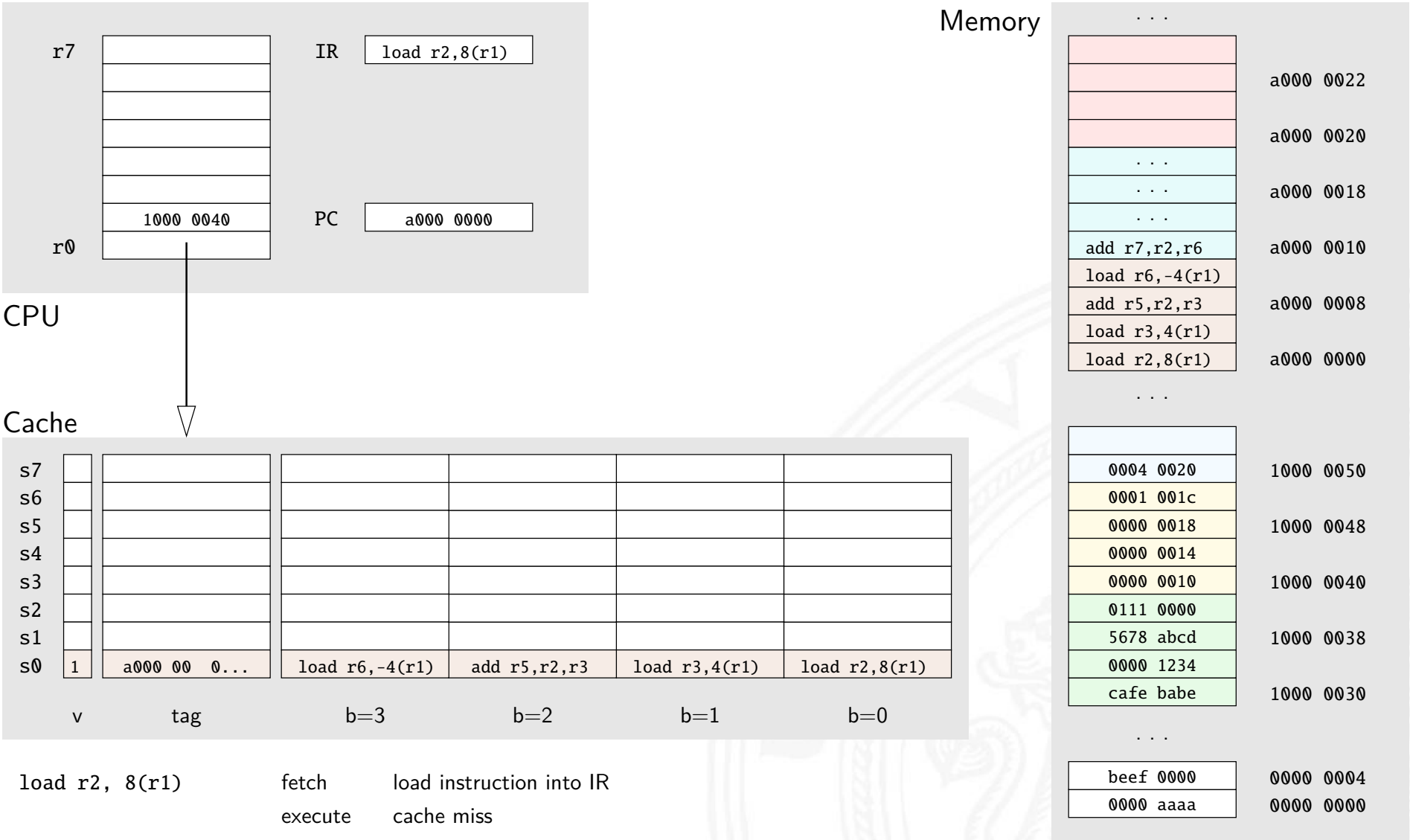
`load r2, 8(r1)` fetch fill cache set s0 from memory

Direct mapped cache: Beispiel – fetch



`load r2, 8(r1)` fetch load instruction into IR

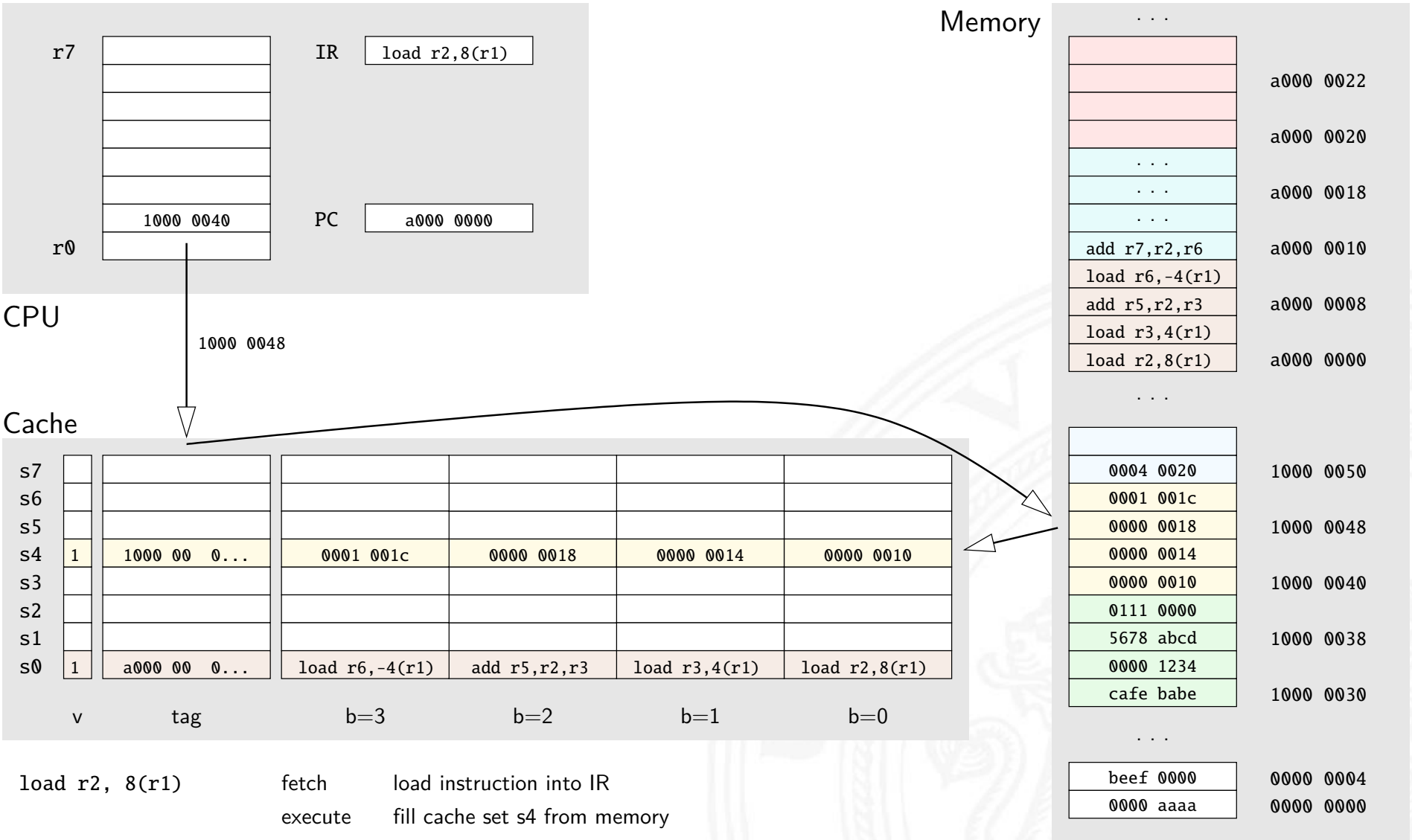
Direct mapped cache: Beispiel – execute miss



`load r2, 8(r1)` fetch load instruction into IR

execute cache miss

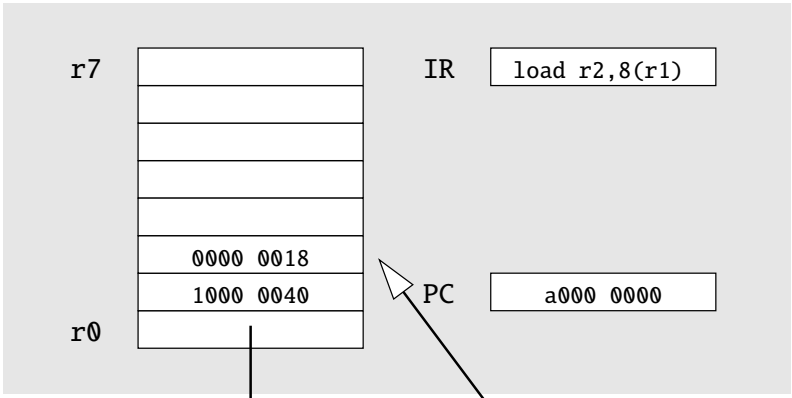
Direct mapped cache: Beispiel – execute fill



`load r2, 8(r1)` fetch load instruction into IR

execute fill cache set s4 from memory

Direct mapped cache: Beispiel – execute



Cache

	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3						
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

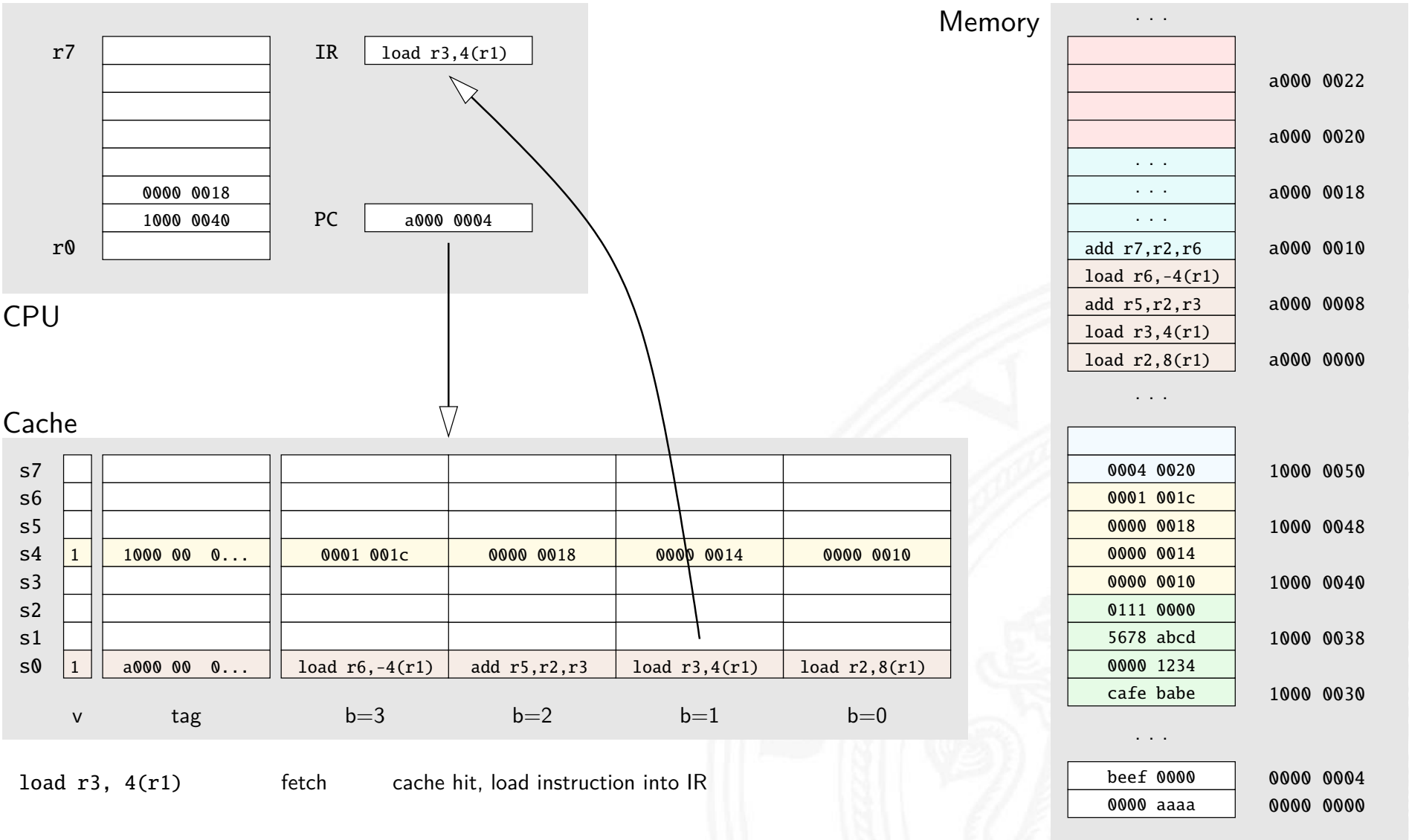
Memory

	a000 0022
	a000 0020
	a000 0018
add r7,r2,r6	a000 0010
load r6,-4(r1)	
add r5,r2,r3	a000 0008
load r3,4(r1)	
load r2,8(r1)	a000 0000
	1000 0050
0004 0020	
0001 001c	
0000 0018	1000 0048
0000 0014	
0000 0010	1000 0040
0111 0000	
5678 abcd	1000 0038
0000 1234	
cafe babe	1000 0030
beef 0000	0000 0004
0000 aaaa	0000 0000

`load r2, 8(r1)` fetch load instruction into IR

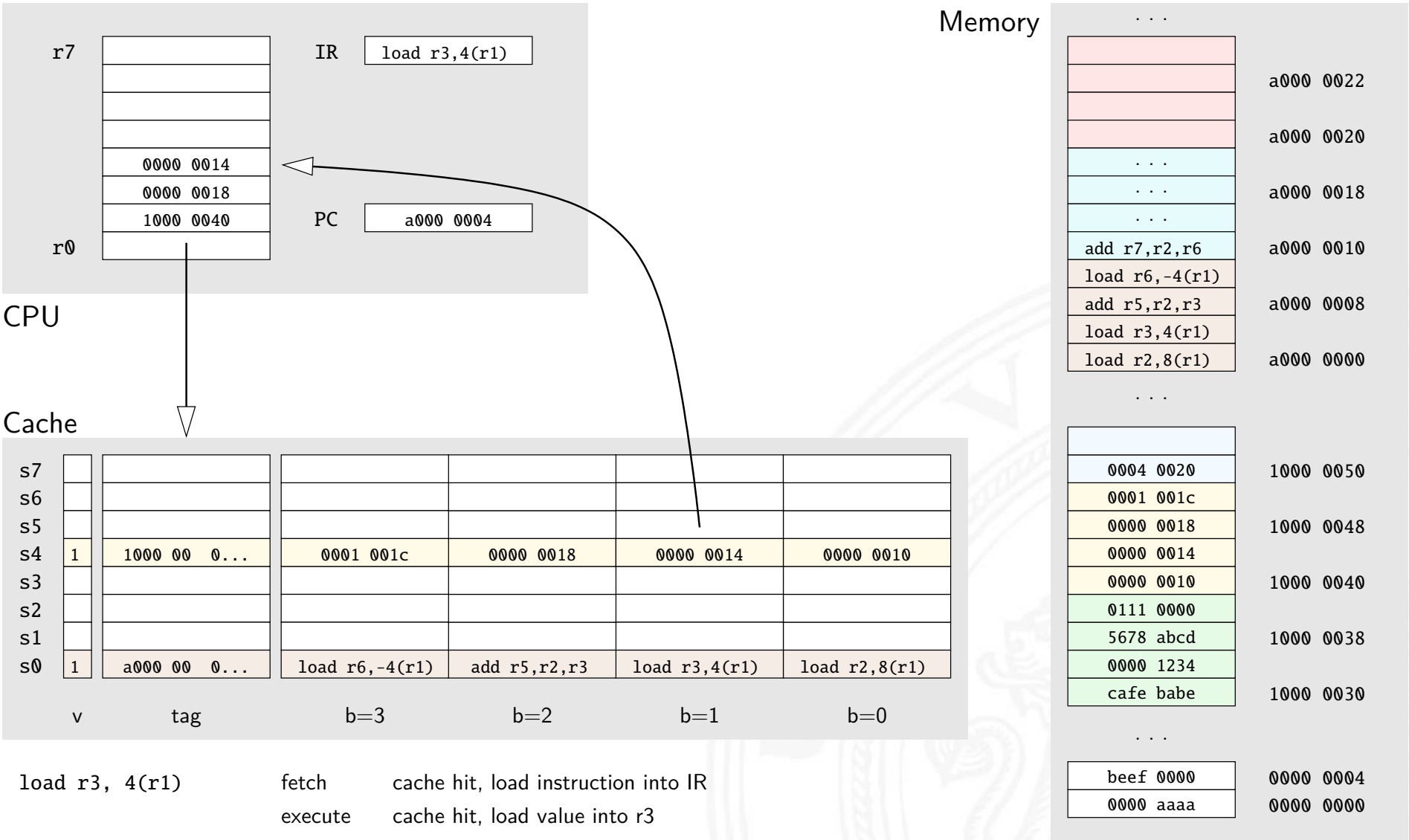
execute load value into r2

Direct mapped cache: Beispiel – fetch hit



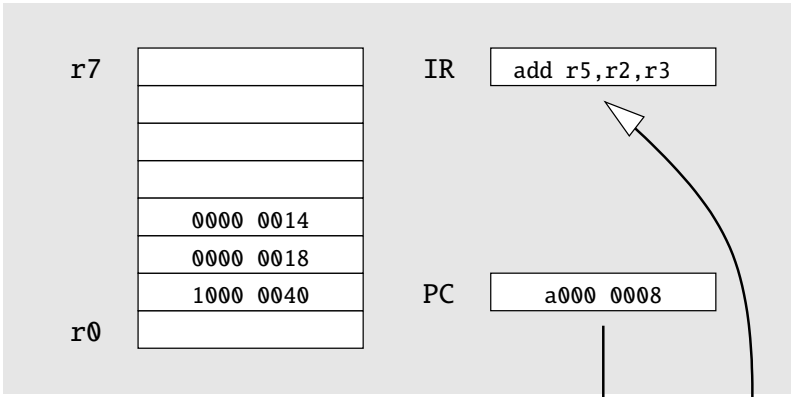
`load r3, 4(r1)` fetch cache hit, load instruction into IR

Direct mapped cache: Beispiel – execute hit



`load r3, 4(r1)`	fetch	cache hit, load instruction into IR
	execute	cache hit, load value into r3

Direct mapped cache: Beispiel – fetch hit



CPU

Cache

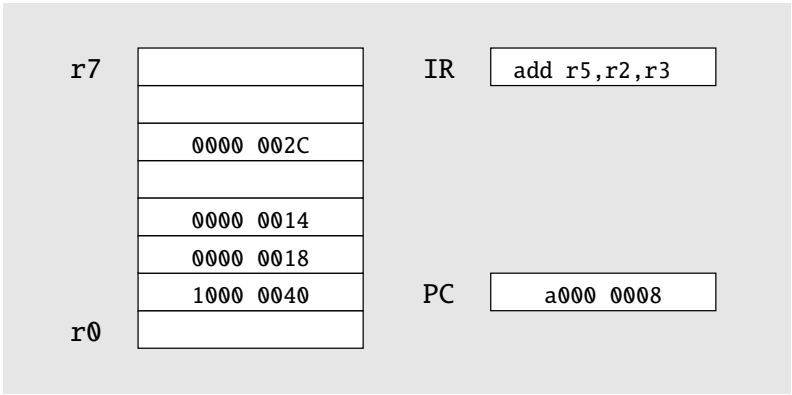
	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3						
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

Memory

	a000 0022
	a000 0020
	a000 0018
add r7,r2,r6	a000 0010
load r6,-4(r1)	
add r5,r2,r3	a000 0008
load r3,4(r1)	
load r2,8(r1)	a000 0000
0004 0020	1000 0050
0001 001c	
0000 0018	1000 0048
0000 0014	
0000 0010	1000 0040
0111 0000	
5678 abcd	1000 0038
0000 1234	
cafe babe	1000 0030
beef 0000	0000 0004
0000 aaaa	0000 0000

`add r5,r2,r3` fetch cache hit, load instruction into IR

Direct mapped cache: Beispiel – execute hit



CPU

Cache

	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3						
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

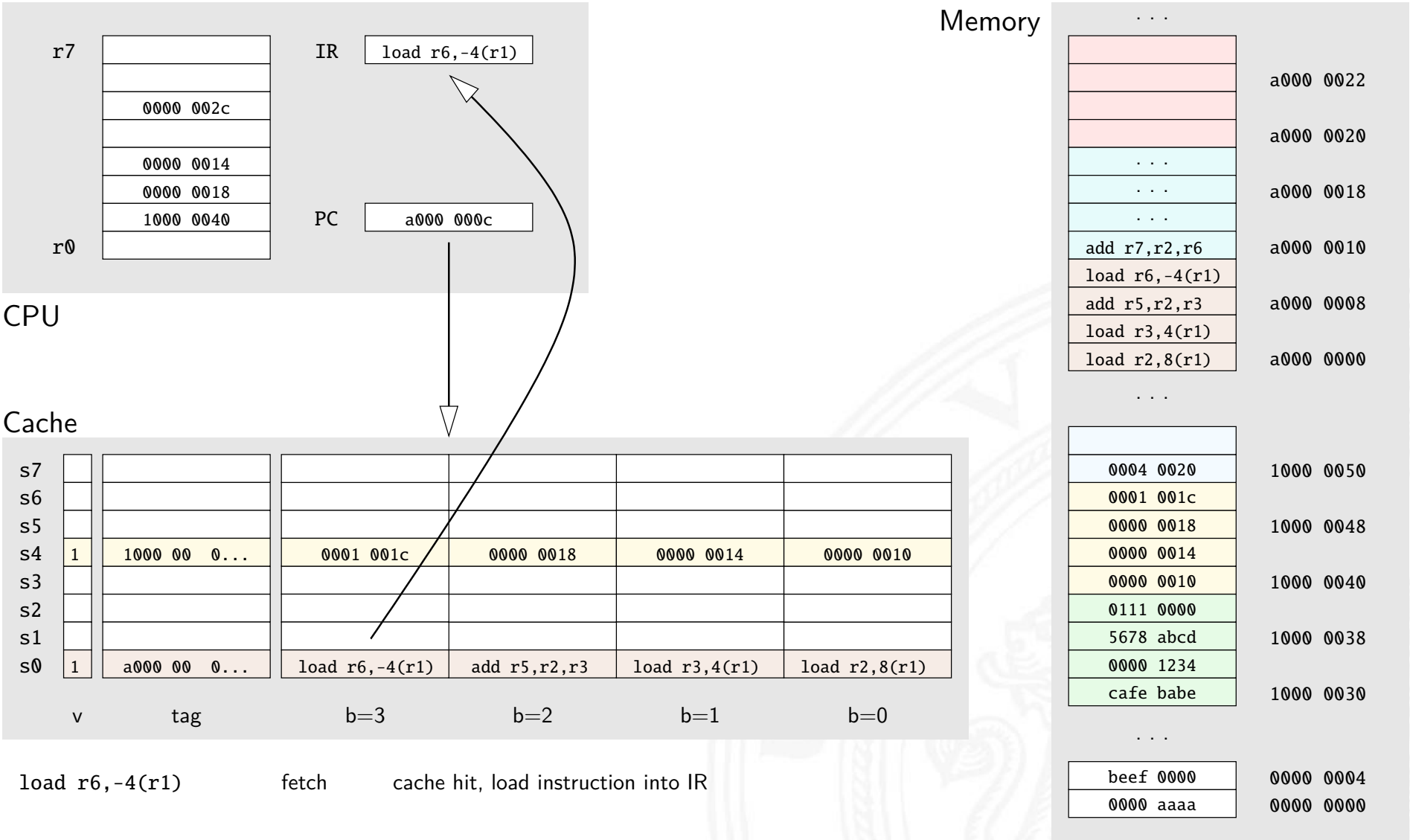
Memory

	a000 0022
	a000 0020
...	
...	a000 0018
...	
add r7,r2,r6	a000 0010
load r6,-4(r1)	
add r5,r2,r3	a000 0008
load r3,4(r1)	
load r2,8(r1)	a000 0000
...	
	1000 0050
0004 0020	
0001 001c	1000 0048
0000 0018	
0000 0014	1000 0040
0000 0010	
0111 0000	1000 0038
5678 abcd	
0000 1234	1000 0030
cafe babe	
...	
beef 0000	0000 0004
0000 aaaa	0000 0000

`add r5,r2,r3` fetch cache hit, load instruction into IR

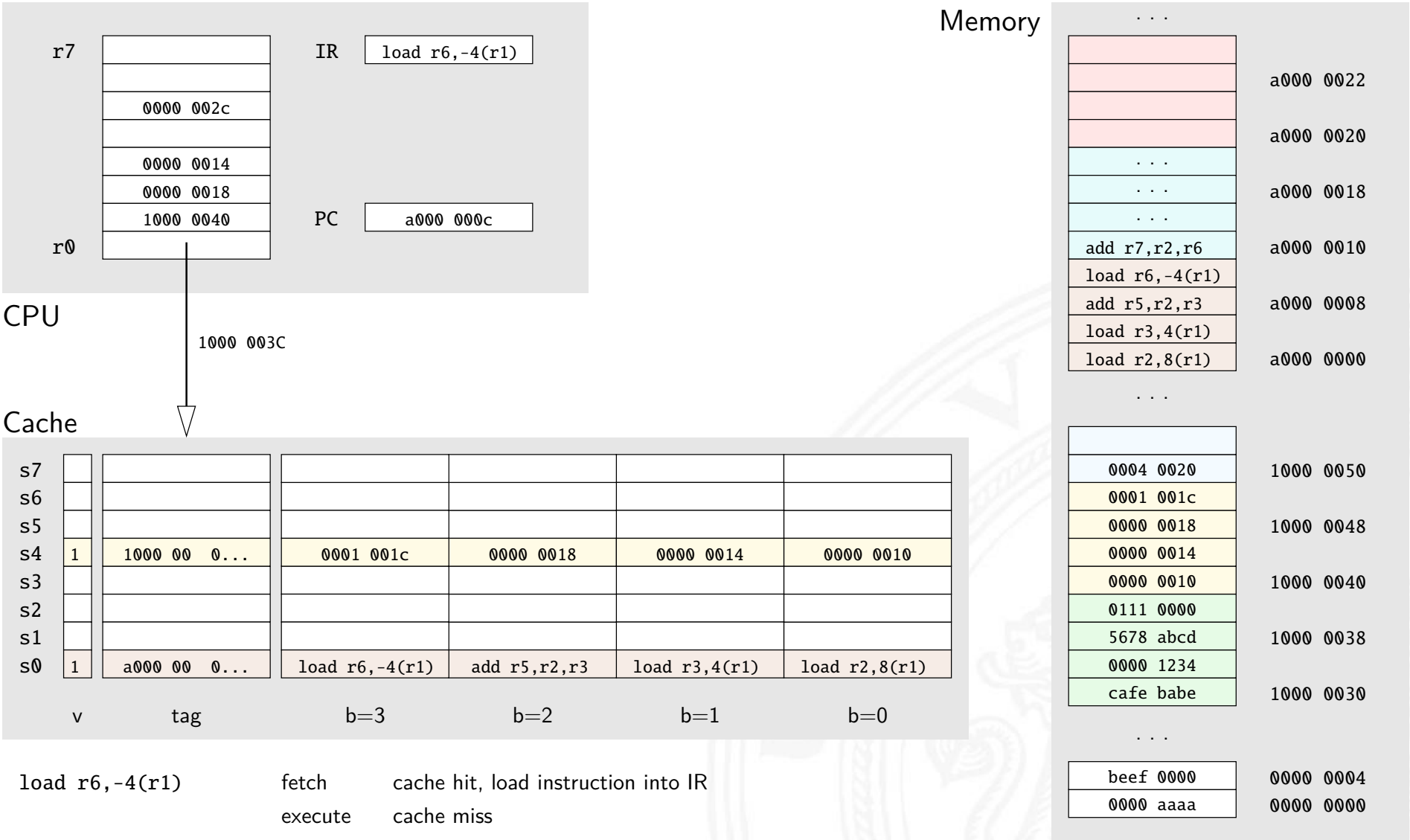
execute no memory access

Direct mapped cache: Beispiel – fetch hit



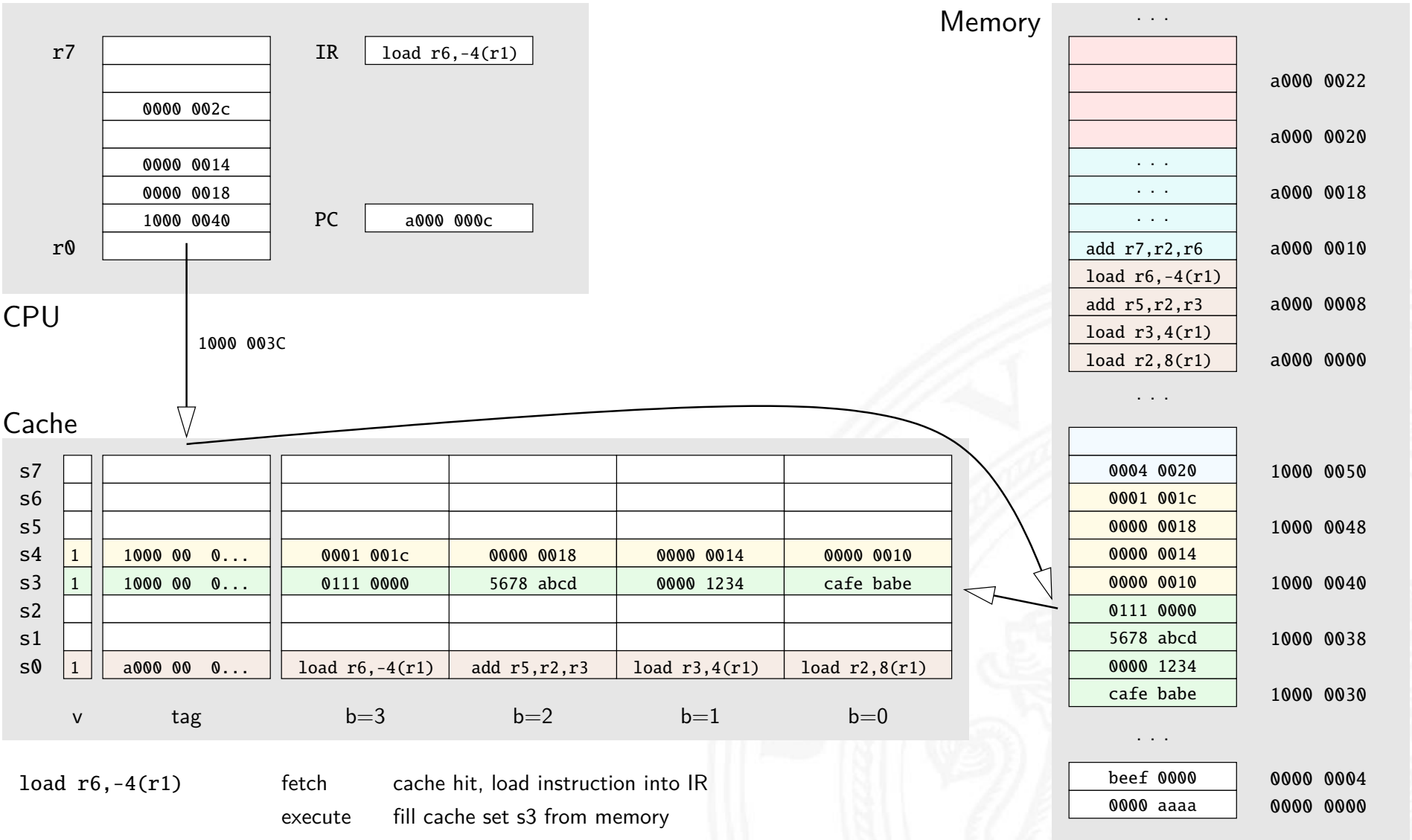
`load r6,-4(r1)` fetch cache hit, load instruction into IR

Direct mapped cache: Beispiel – execute miss



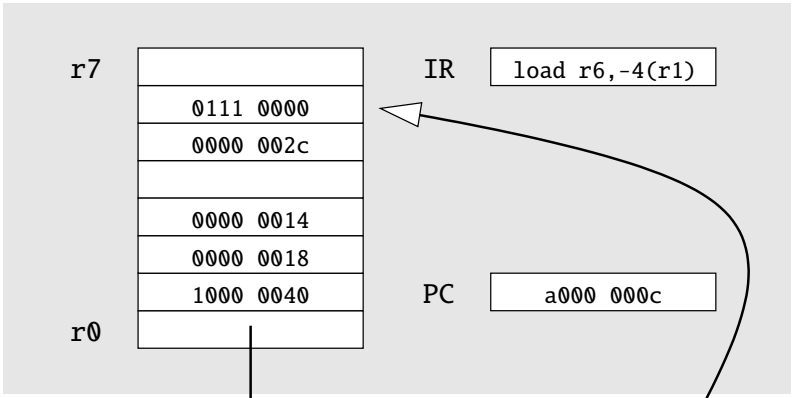
`load r6,-4(r1)`	fetch	cache hit, load instruction into IR
	execute	cache miss

Direct mapped cache: Beispiel – execute fill



`load r6,-4(r1)`	fetch	cache hit, load instruction into IR
	execute	fill cache set s3 from memory

Direct mapped cache: Beispiel – execute



Cache

	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3	1	1000 00 0...	0111 0000	5678 abcd	0000 1234	cafe babe
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

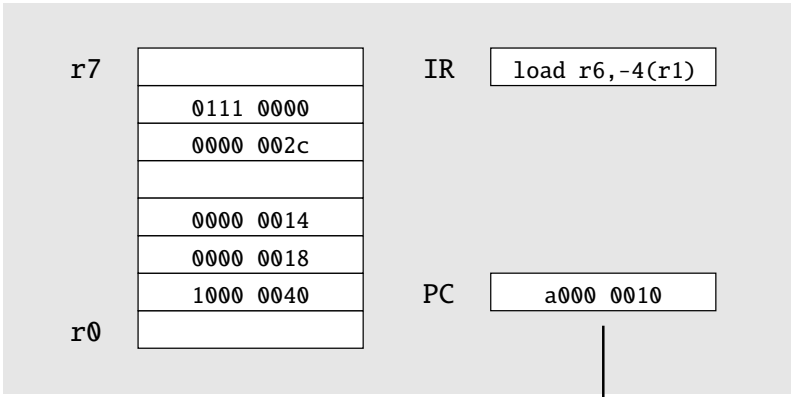
Memory

	a000 0022
	a000 0020
	a000 0018
add r7,r2,r6	a000 0010
load r6,-4(r1)	
add r5,r2,r3	a000 0008
load r3,4(r1)	
load r2,8(r1)	a000 0000
0004 0020	1000 0050
0001 001c	
0000 0018	1000 0048
0000 0014	
0000 0010	1000 0040
0111 0000	
5678 abcd	1000 0038
0000 1234	
cafe babe	1000 0030
beef 0000	0000 0004
0000 aaaa	0000 0000

`load r6,-4(r1)` fetch cache hit, load instruction into IR

execute load value into r6

Direct mapped cache: Beispiel – fetch miss



CPU

Cache

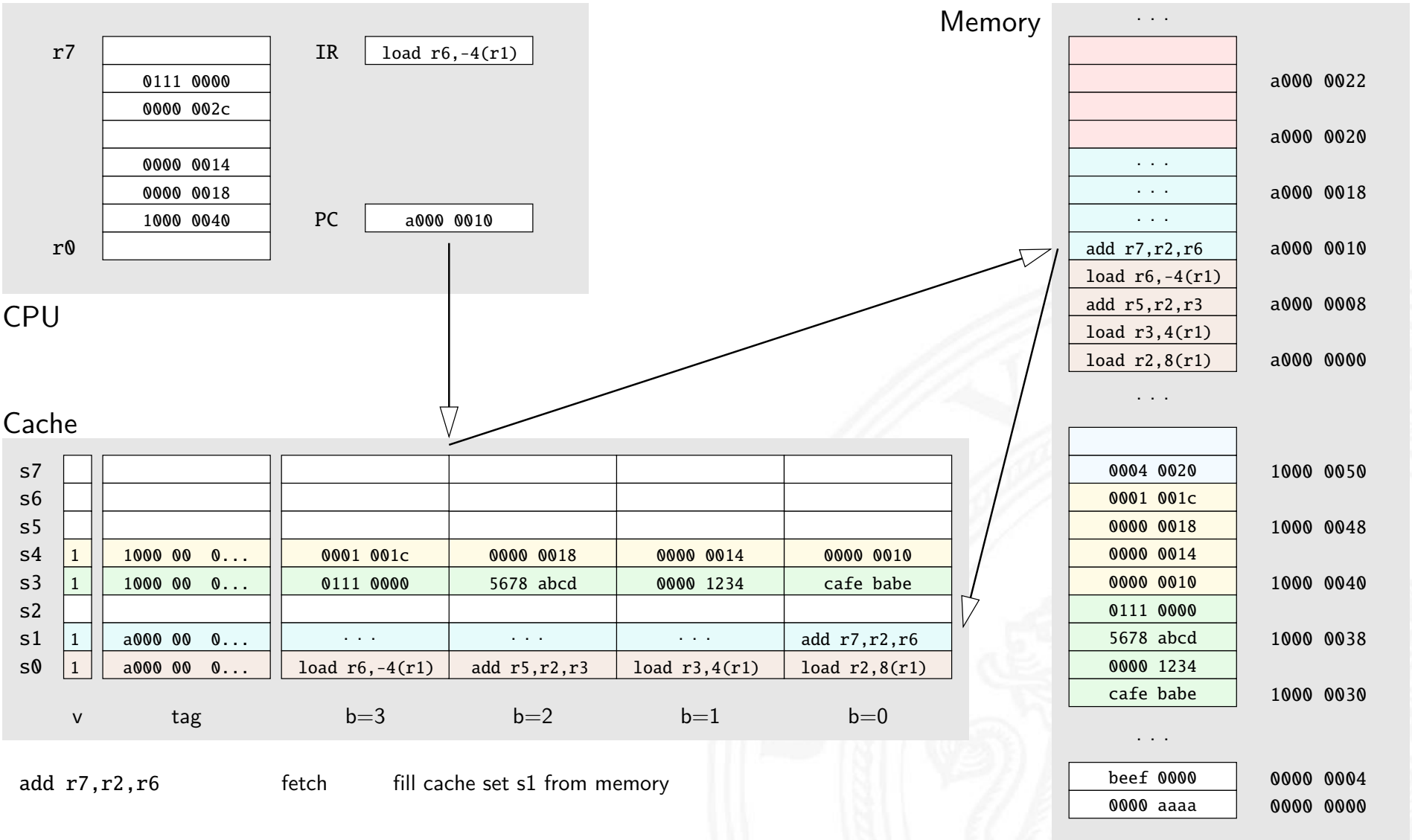
		v	tag	b=3	b=2	b=1	b=0
s7							
s6							
s5							
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010	
s3	1	1000 00 0...	0111 0000	5678 abcd	0000 1234	cafe babe	
s2							
s1							
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)	

Memory

	a000 0022
	a000 0020
...	
...	a000 0018
...	
add r7,r2,r6	a000 0010
load r6,-4(r1)	
add r5,r2,r3	a000 0008
load r3,4(r1)	
load r2,8(r1)	a000 0000
...	
0004 0020	1000 0050
0001 001c	
0000 0018	1000 0048
0000 0014	
0000 0010	1000 0040
0111 0000	
5678 abcd	1000 0038
0000 1234	
cafe babe	1000 0030
...	
beef 0000	0000 0004
0000 aaaa	0000 0000

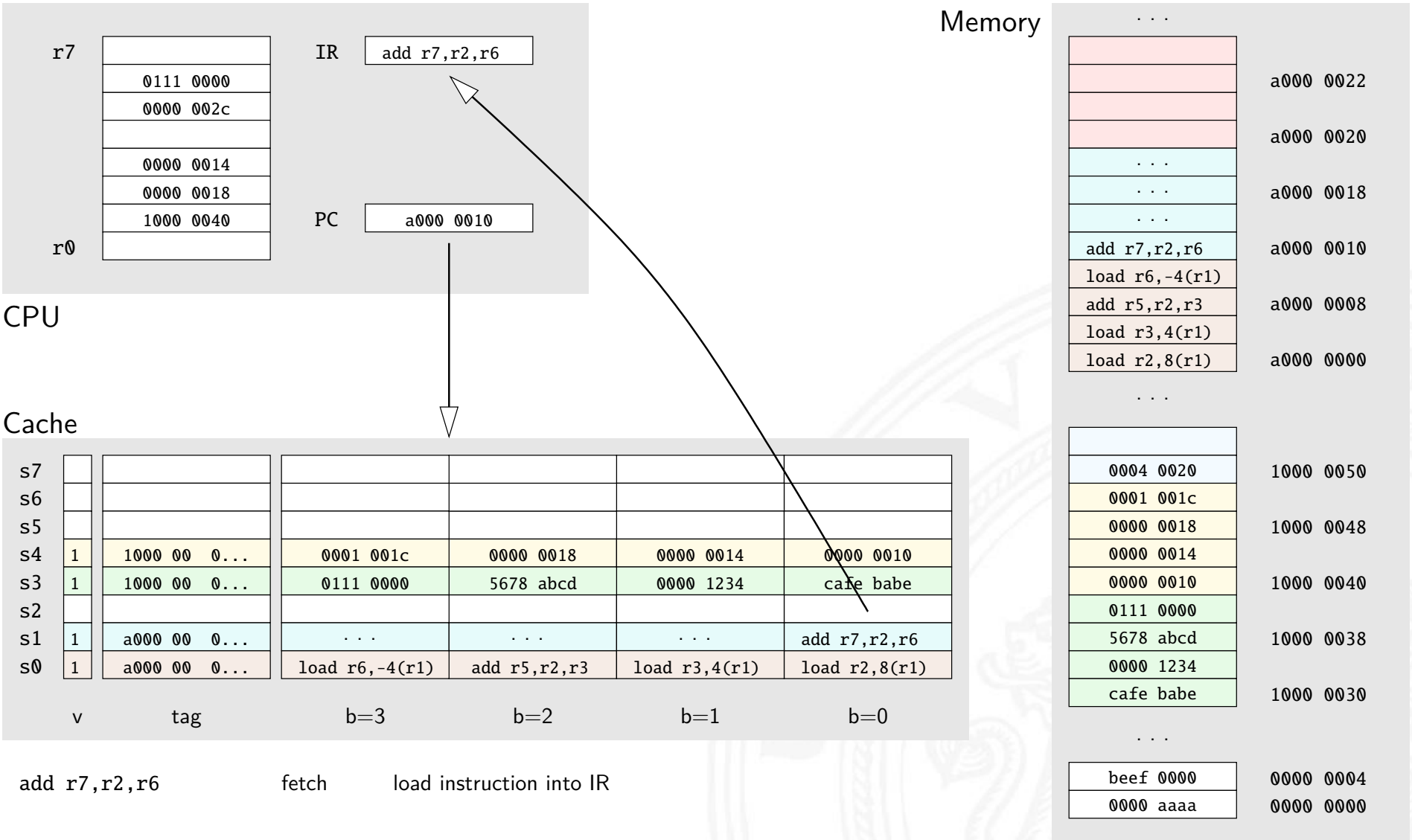
`add r7,r2,r6` fetch cache miss

Direct mapped cache: Beispiel – fetch fill



`add r7,r2,r6` fetch fill cache set s1 from memory

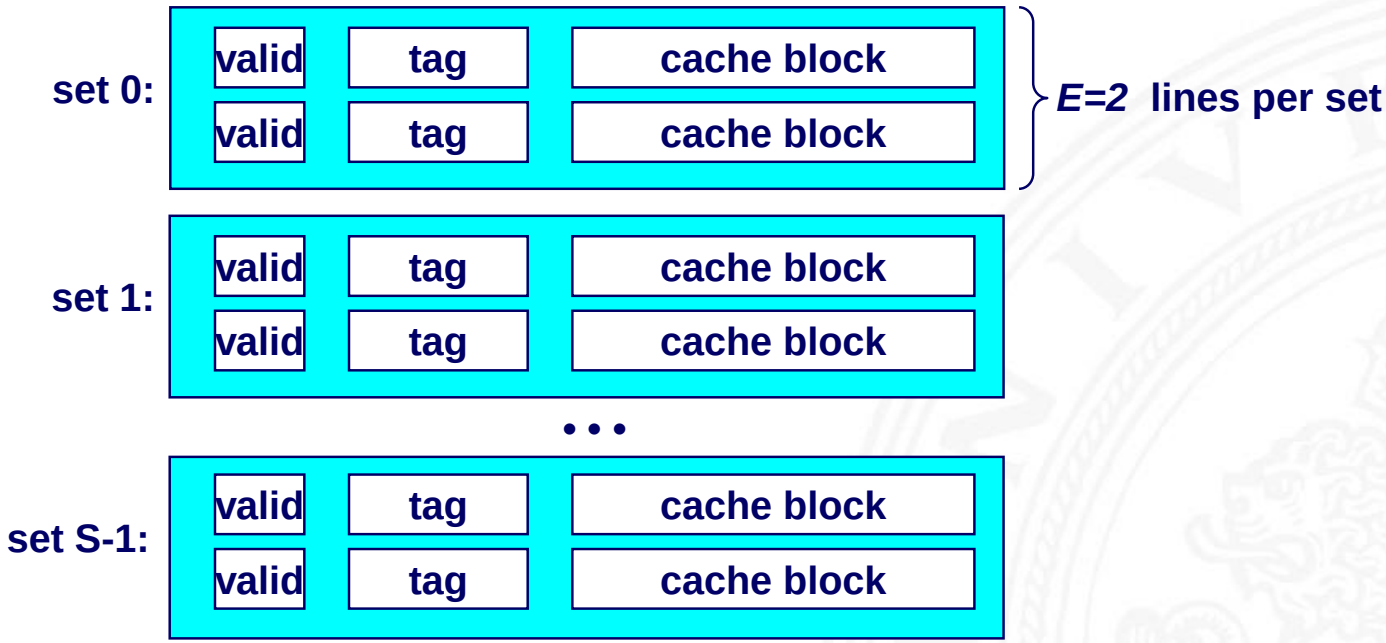
Direct mapped cache: Beispiel – fetch



`add r7,r2,r6` fetch load instruction into IR

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...“

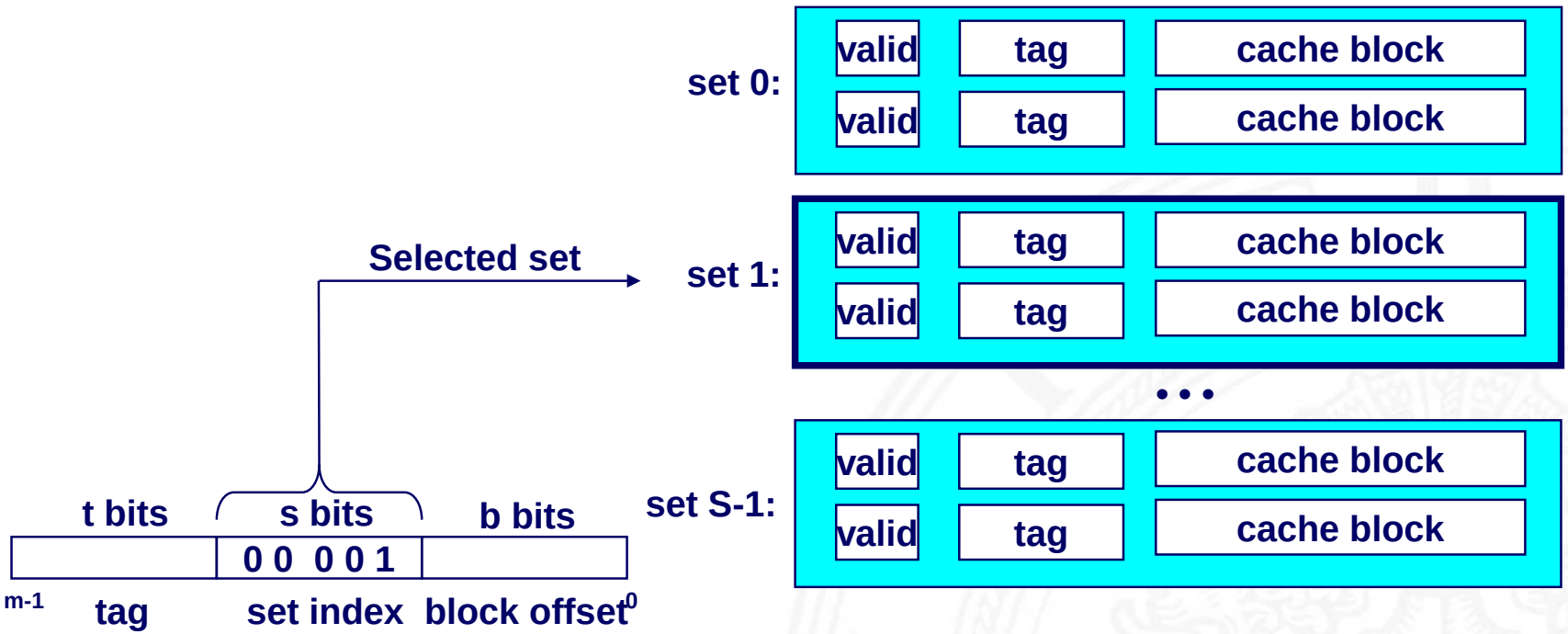


[BO15]

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

Zugriff auf n-fach assoziative Caches

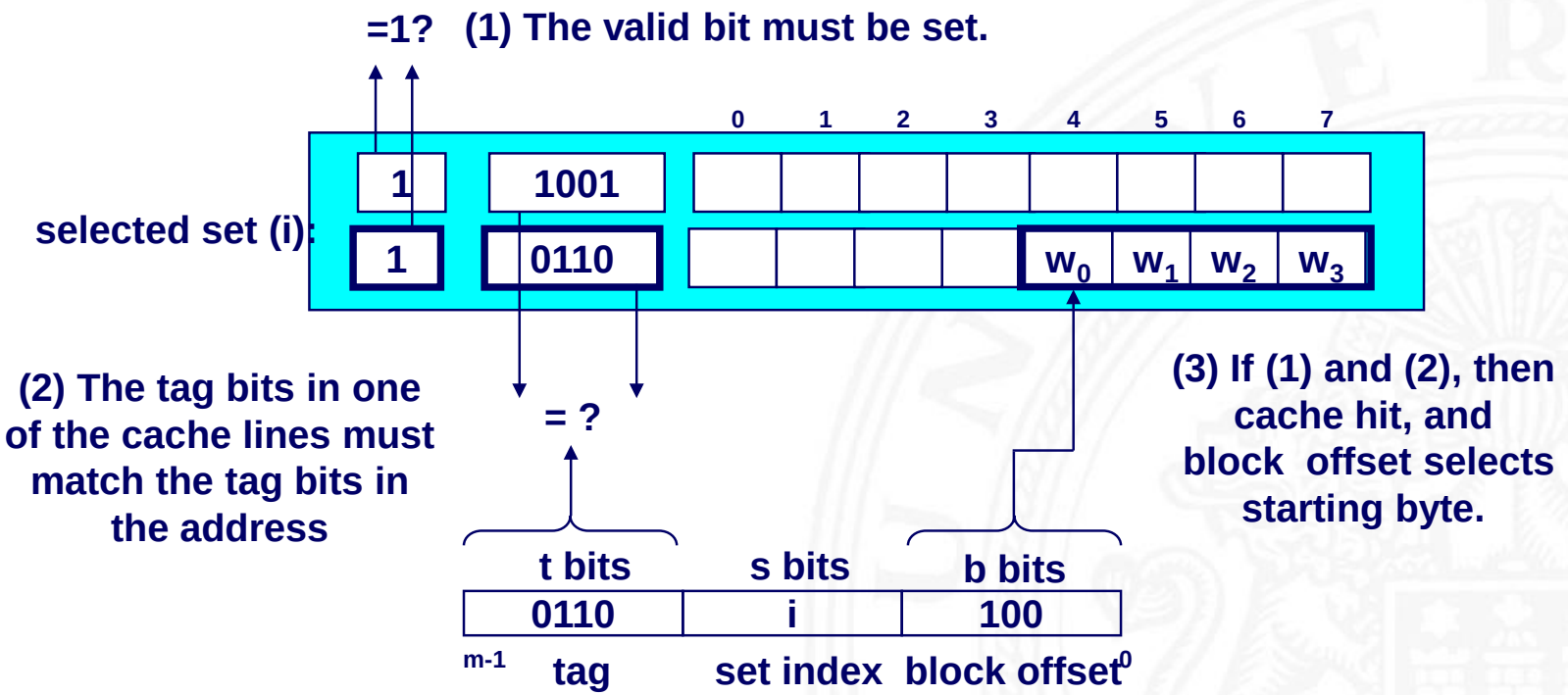
1. Bereichsauswahl durch Bits ⟨*set index*⟩



[BO15]

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

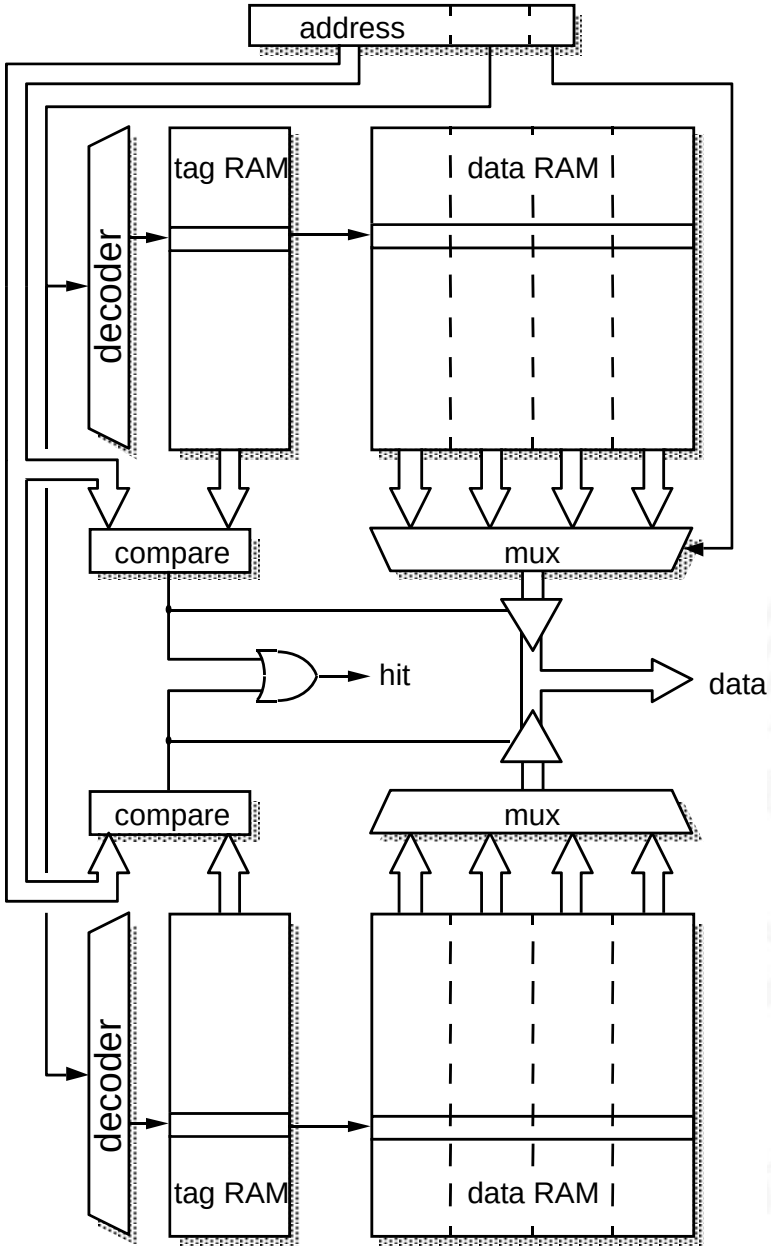
2. ⟨*valid*⟩: sind die Daten gültig?
3. „Line matching“: Cache-Zeile mit passendem ⟨*tag*⟩ finden?
   dazu Vergleich aller „tags“ des Bereichs ⟨*set index*⟩
4. Wortselektion extrahiert Wort unter Offset ⟨*block offset*⟩



[BO15]

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

Prinzip



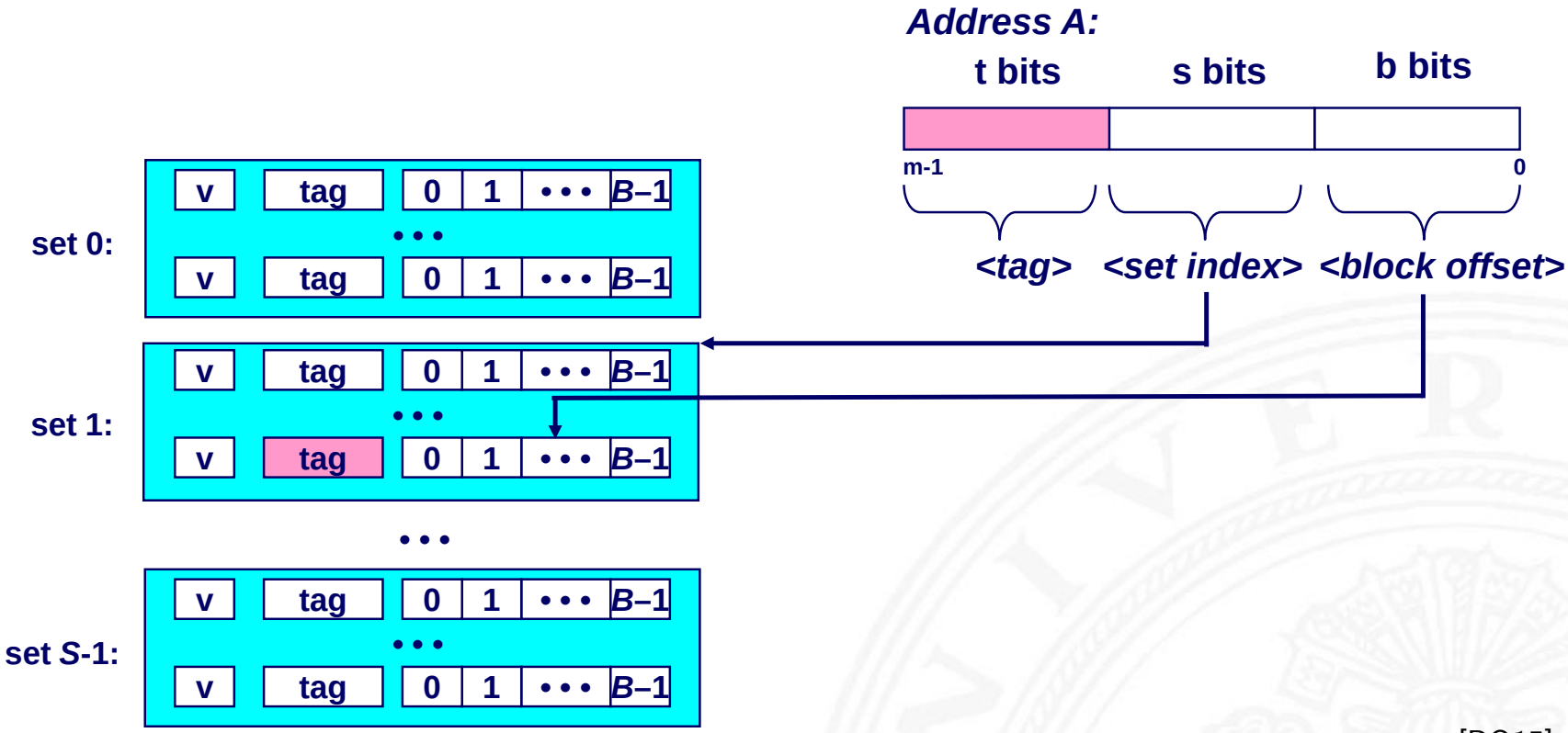
[Fur00]

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



[BO15]

- Parameter: S, B, E
- Cache speichert immer größere Blöcke / „*Cache-Line*“
- Wortauswahl durch $\langle$*block offset*$\rangle$ in Adresse

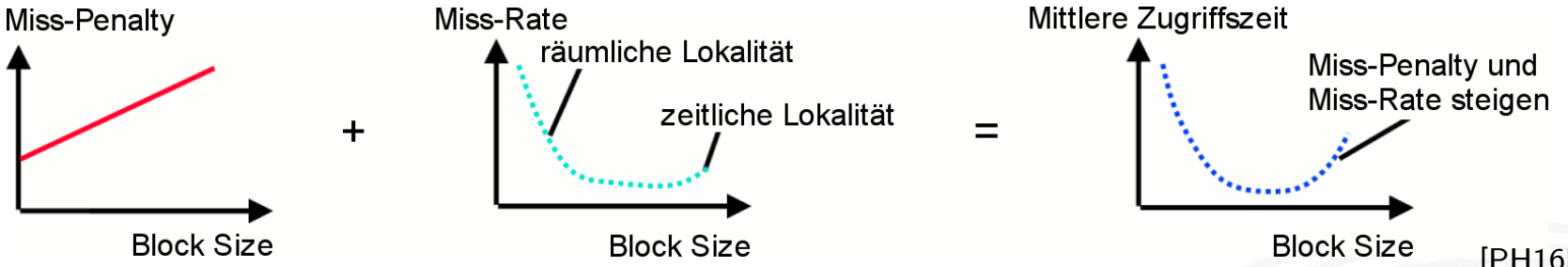
Cache – Dimensionierung (cont.)

Vor- und Nachteile des Cache

+ nutzt räumliche Lokalität aus
  Speicherzugriffe von Programmen (Daten und Instruktionen) liegen in ähnlichen/aufeinanderfolgenden Adressbereichen
+ breite externe Datenbusse, es werden ganze Bereiche übertragen
+ nutzt Burst-Adressierung des Speichers: Adresse nur für erstes Wort vorgeben, dann automatisches Inkrement
+ kürzere interne Cache-Adressen
– Hardwareaufwand und Kosten

Cache – Dimensionierung (cont.)

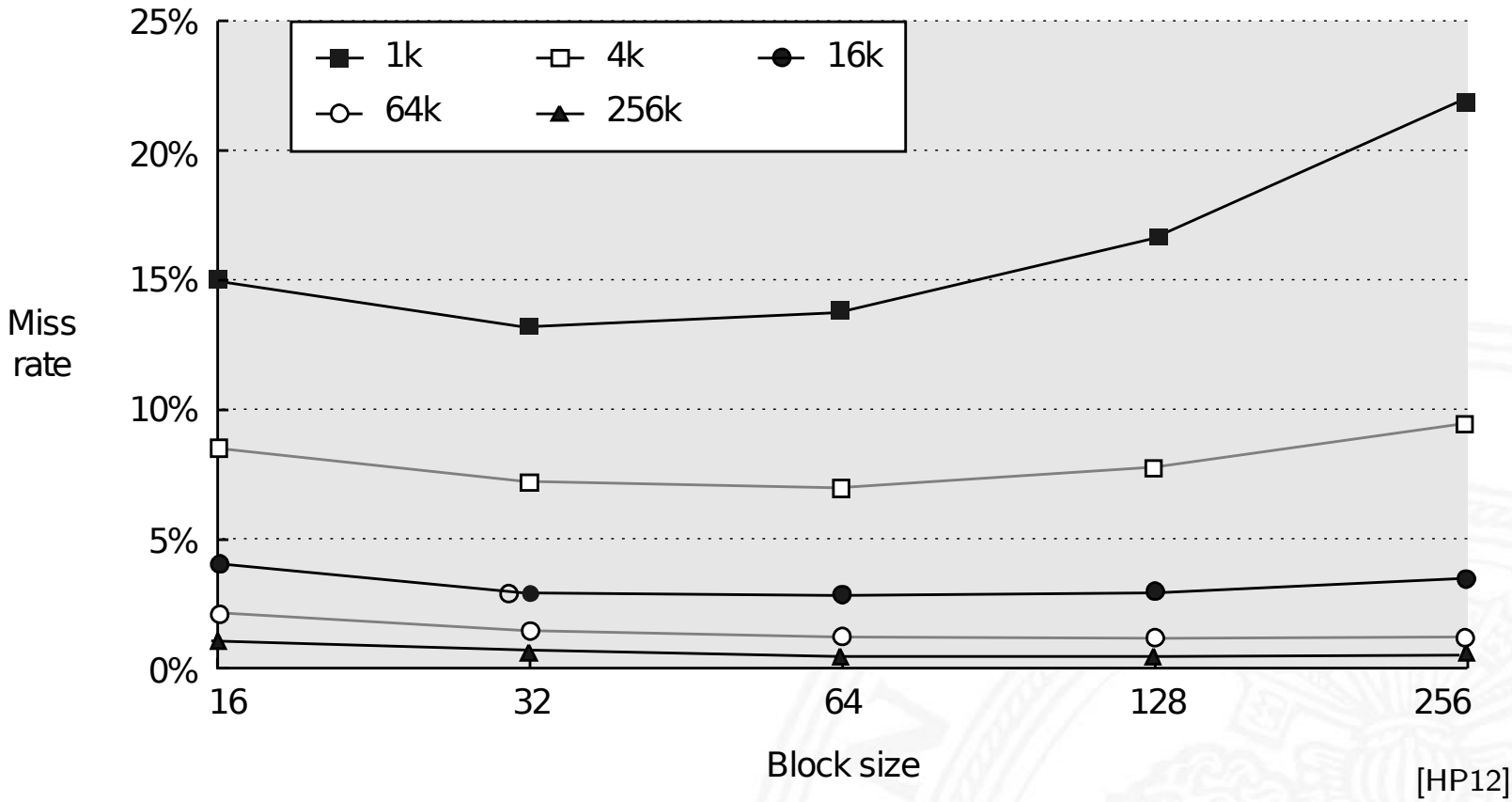
Cache- und Block-Dimensionierung



[PH16b]

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

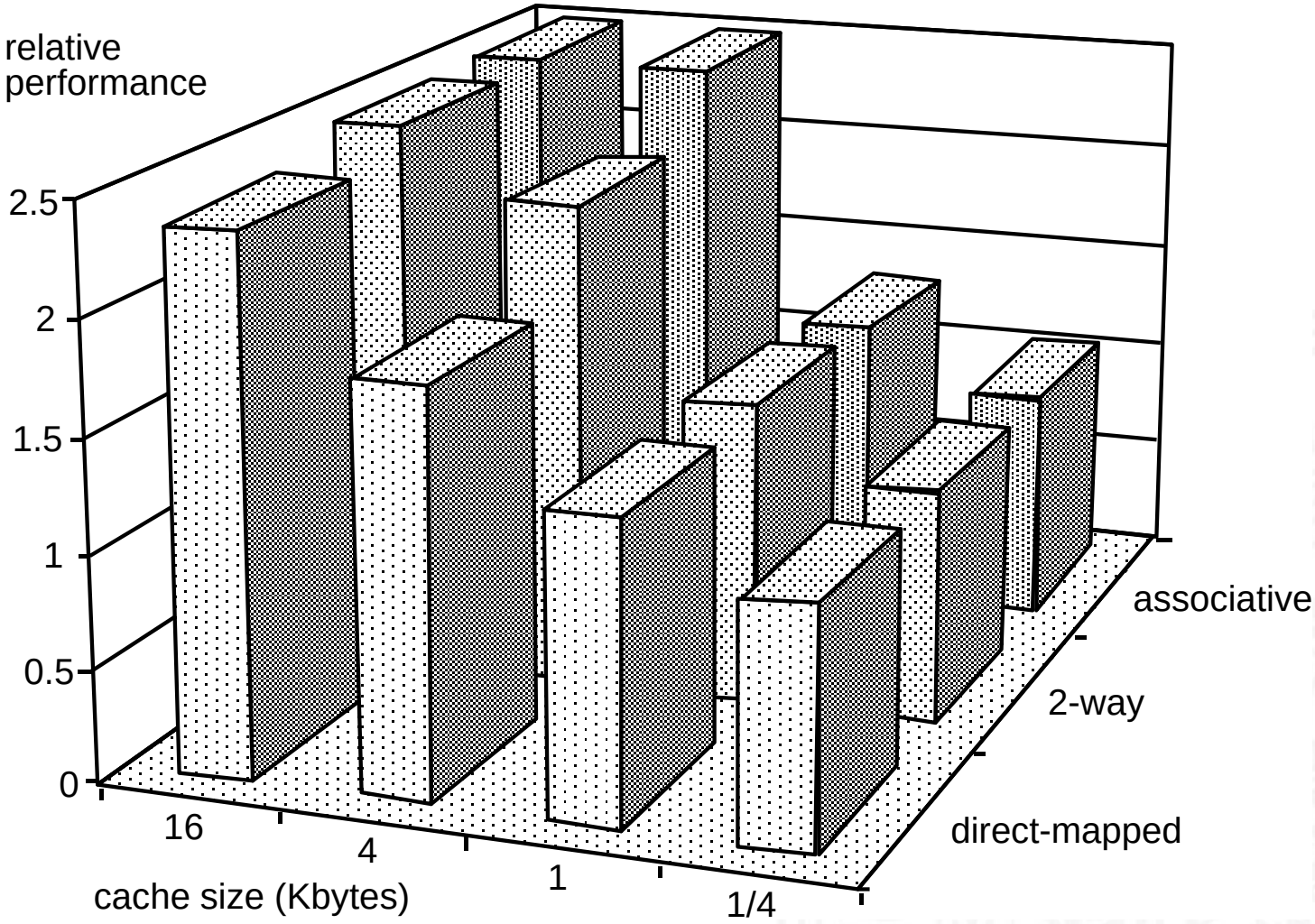
Cache – Dimensionierung (cont.)



[HP12]

- Block-Size: 32. . . 128 Byte
  L1-Cache: 4. . . 256 KiByte
  L2-Cache: 256. . . 4 096 KiByte

Cache – Dimensionierung: relative Performanz



[Fur00]

Drei Typen von Cache-Misses

- cold miss
  - Cache ist (noch) leer
- conflict miss
  - wenn die Kapazität des Cache eigentlich ausreicht, aber unterschiedliche Daten in den selben Block abgebildet werden
  - Beispiel für „Trashing“ beim direct-mapped Cache mit S=8: abwechselnder Zugriff auf Blöcke 0, 8, 0, 8, 0, 8, ... ist jedesmal ein Miss
- capacity miss
  - wenn die Menge der aktiven Blöcke („working set“) größer ist als die Kapazität des Cache

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
  - $\Rightarrow$ spezielle Befehle für „Cache-Flush“
  - $\Rightarrow$ „non-cacheable“ Speicherbereiche

Cache-Kohärenz

- Daten zwischen Cache und Speicher konsistent halten
- notwendig wenn mehrere Einheiten (Bus-Master: Prozessor, DMA-Controller) auf Speicher zugreifen können:
  wichtig für „*Symmetric Multiprocessing*“
- Harvard-Architektur hat getrennte Daten- und Instruktions-Speicher
  - Instruktionen sind read-only

  $\Rightarrow$ einfacherer Instruktions-Cache

  $\Rightarrow$ kein Cache-Kohärenz Problem
- Cache-Kohärenz Protokolle und „*Snooping*“
  - alle Prozessoren(P_1, P_2...) überwachen alle Bus-Transaktionen
    Cache „schnüffelt“ am Speicherbus
  - Prozessor P_2 greift auf Daten zu, die im Cache von P_1 liegen
    P_2 Schreibzugriff $\Rightarrow$ P_1 Cache aktualisieren / ungültig machen
    P_2 Lesezugriff $\Rightarrow$ P_1 Cache liefert Daten
  - *Was ist mit gleichzeitige Zugriffen von P_1, P_2?*

Cache-Kohärenz (cont.)

- viele verschiedene Protokolle: Hersteller- / Prozessor-spezifisch
  - SI („*Write Through*“)
  - MSI, MOSI,
  - MESI: *Modified*, *Exclusive*, *Shared*, *Invalid*
  - MOESI: *Modified* (exclusive), *Owned* (Modified shared), *Exclusive*, *Shared*, *Invalid*
  - ...

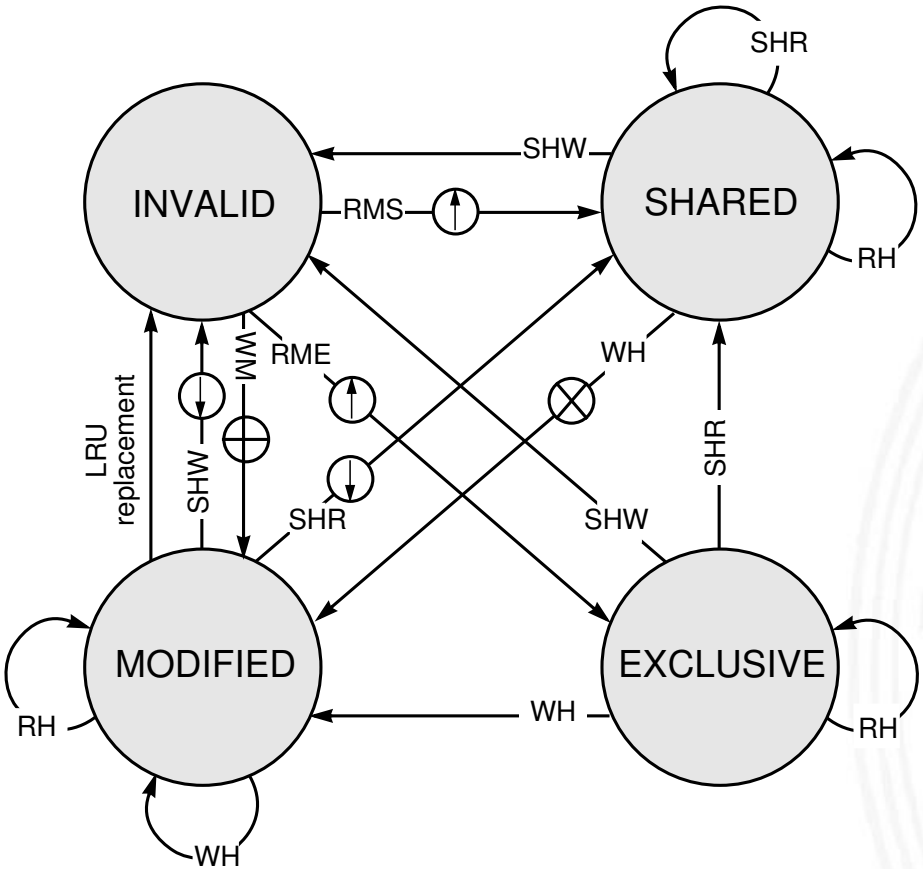
  siehe z.B.: `en.wikipedia.org/wiki/Cache_memory`

MESI Protokoll

- Caches enthalten Wert, Tag und zwei Statusbits für die vier Protokollzustände
  - **M**odified: gültiger Wert, nur in diesem Cache, gegenüber Hauptspeicher-Wert verändert
  - **E**xclusive: gültiger Wert, nur in diesem Cache nicht verändert (unmodified)
  - **S**hared: gültiger Wert, in mehreren Caches vorhanden nicht verändert (unmodified)
  - **I**nvalid: ungültiger Inhalt, Initialzustand
- alle Prozessoren überwachen alle Bus-Transaktionen
- bei Speicherzugriffen Aktualisierung des Status'
- Zugriffe auf „modified"-Werte werden erkannt:
  1. fremde Bus-Transaktion unterbrechen
  2. eigenen (=modified) Wert zurückschreiben
  3. Status auf shared ändern
  4. unterbrochene Bus-Transaktion neu starten

MESI Protokoll (cont.)

- erfordert spezielle Snoop-Logik im Prozessor
- garantiert Cache-Kohärenz
- gute Performance, aber schlechte Skalierbarkeit
- Zustandsübergänge: MESI Protokoll



PowerPC 604 RISC Microprocessor User's Manual [Motorola / IBM]

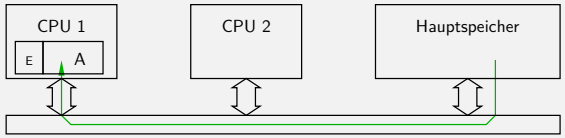
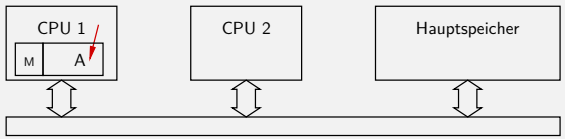
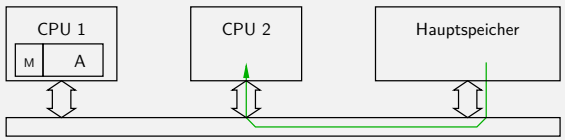
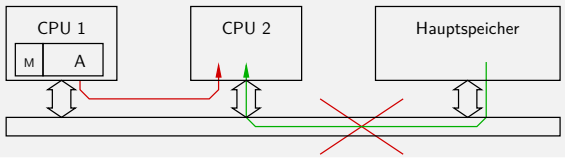
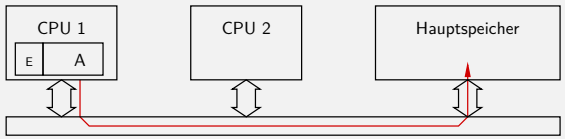
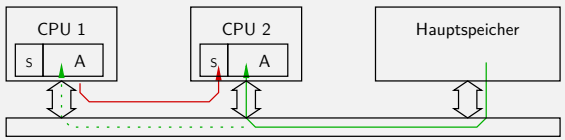
RH = Read hit
RMS = Read miss, shared
RME = Read miss, exclusive
WH = Write hit
WM = Write miss
SHR = Snoop hit on a read
SHW = Snoop hit on a write or read-with-intent-to-modify

Bus Transactions

= Snoop push
= Invalidate transaction
= Read-with-intent-to-modify
= Read

MESI Protokoll (cont.)

- „*Snooping*“ Beispiel

	MESI-Status Wert A:	CPU1	CPU2
	CPU 1 lädt Wert A	E	I
	CPU 1 modifiziert A	M	I
	CPU 2 lädt A (aber Wert modified)	- -	- -
	CPU 1 SNOOP! (CPU2 read gestoppt)	- -	- -
	CPU 1 schreibt A (CPU2 read gestoppt)	E	I
	CPU 2 lädt A (CPU2 read restart, A shared)	S	S

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

$\Rightarrow$ Verbesserung der Cache Performanz durch kleinere R_{Miss}

- größere Caches (– mehr Hardware)
- höhere Assoziativität (– langsamer)

Optimierung der Cachezugriffe (cont.)

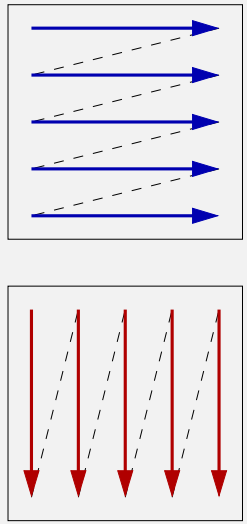
$\Rightarrow$ Optimierungstechniken

- 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)

Cache Effekte bei Matrixzugriffen


```
public static double sumRowCol( double[][] matrix ) {
  int rows = matrix.length;
  int cols = matrix[0].length;
  double sum = 0.0;
  for( int r = 0; r < rows; r++ ) {
    for( int c = 0; c < cols; c++ ) {
      sum += matrix[r][c];
    }
  }
  return sum;
}
```

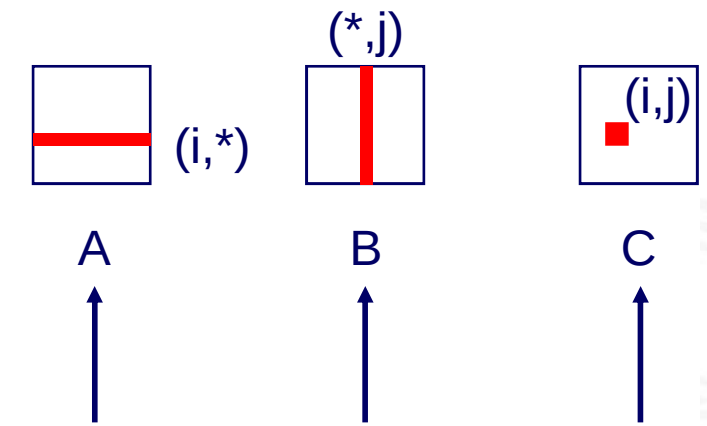


Matrix creation (5000×5000)	2105 msec.	
Matrix row-col summation	75 msec.	
Matrix col-row summation	383 msec.	⇒ 5 × langsamer
Sum = 600,8473695346258 / 600,8473695342268		⇒ andere Werte

Cache Effekte bei Matrixzugriffen (cont.)


```
/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:



Misses per Inner Loop Iteration:

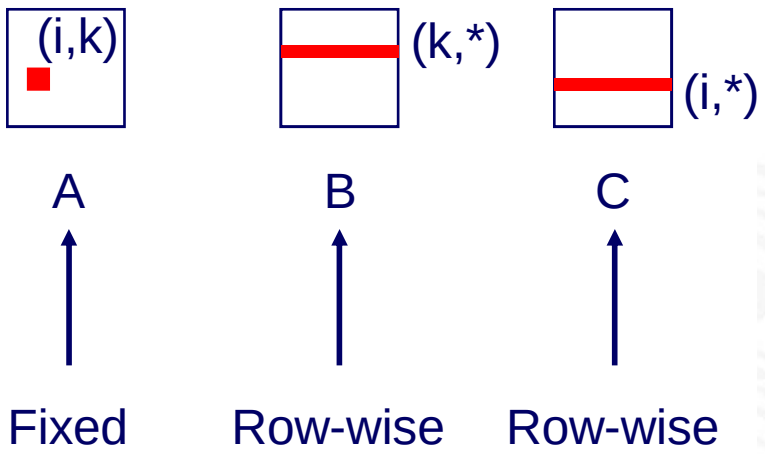
A	B	C
0.25	1.0	0.0

[BO15]

Cache Effekte bei Matrixzugriffen (cont.)


```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Inner loop:



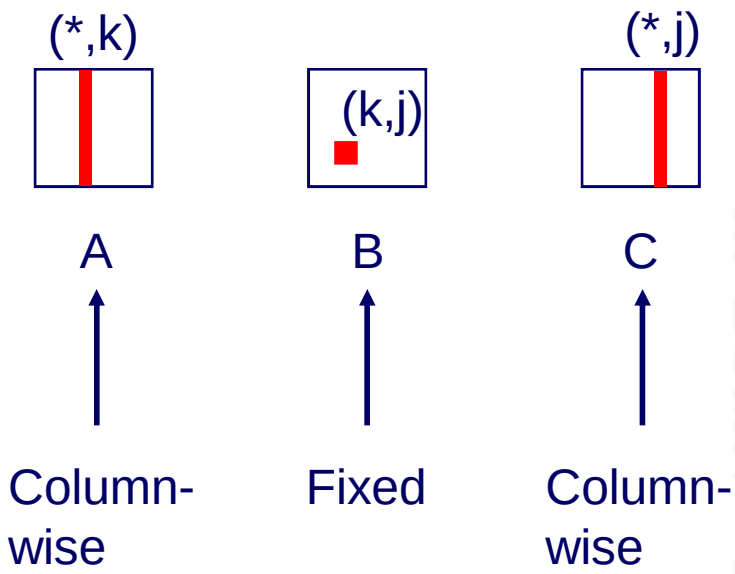
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

[BO15]

Cache Effekte bei Matrixzugriffen (cont.)


```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

[BO15]

Cache Effekte bei Matrixzugriffen (cont.)

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (i=0; i<n; i++)  {
   for (j=0; j<n; j++) {
      sum = 0.0;
      for (k=0; k<n; k++)
         sum += a[i][k] * b[k][j];
      c[i][j] = sum;
   }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

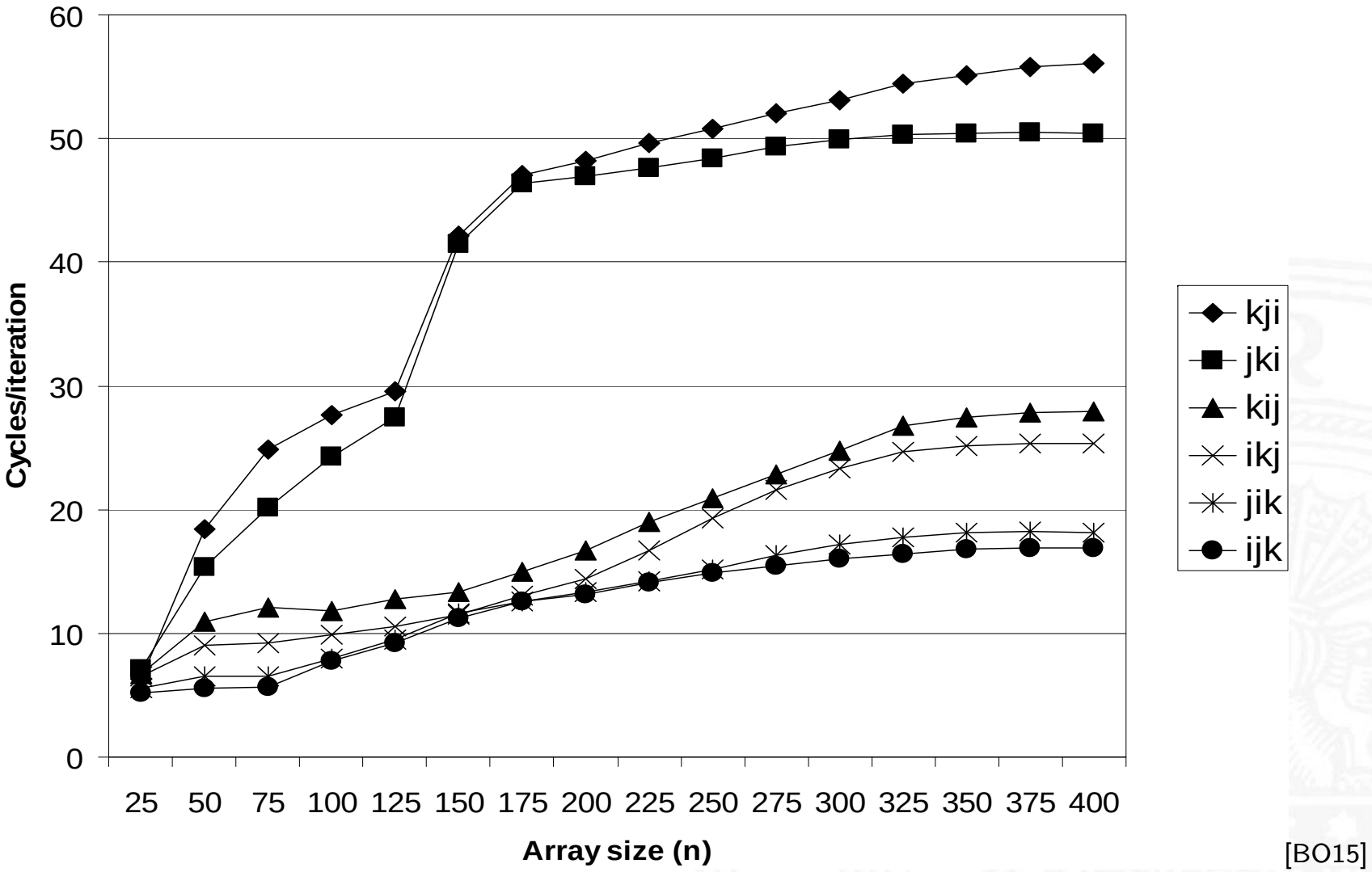
```
for (k=0; k<n; k++) {
   for (i=0; i<n; i++) {
      r = a[i][k];
      for (j=0; j<n; j++)
         c[i][j] += r * b[k][j];
   }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
for (j=0; j<n; j++) {
   for (k=0; k<n; k++) {
      r = b[k][j];
      for (i=0; i<n; i++)
         c[i][j] += a[i][k] * r;
   }
}
```

[BO15]

Cache Effekte bei Matrixzugriffen (cont.)



[BO15]

ARM7 / ARM10

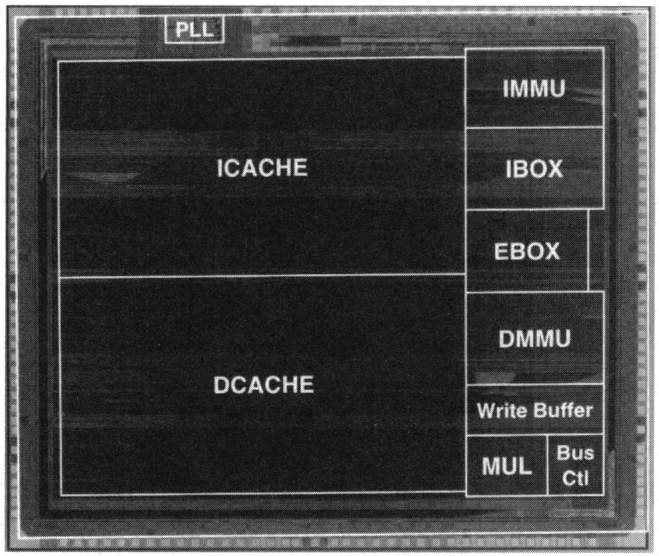
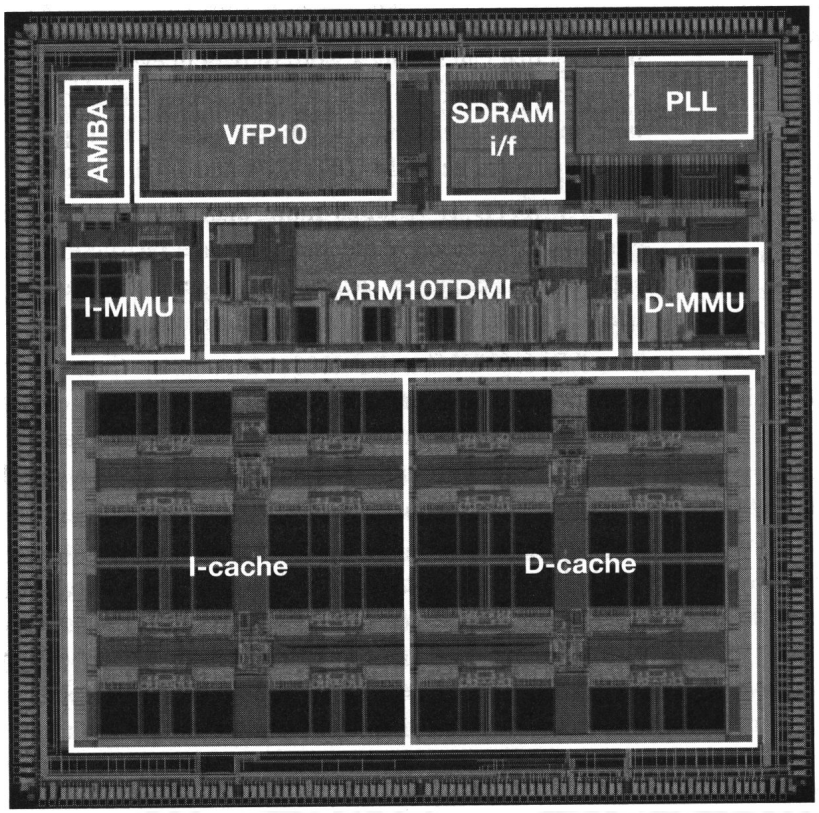


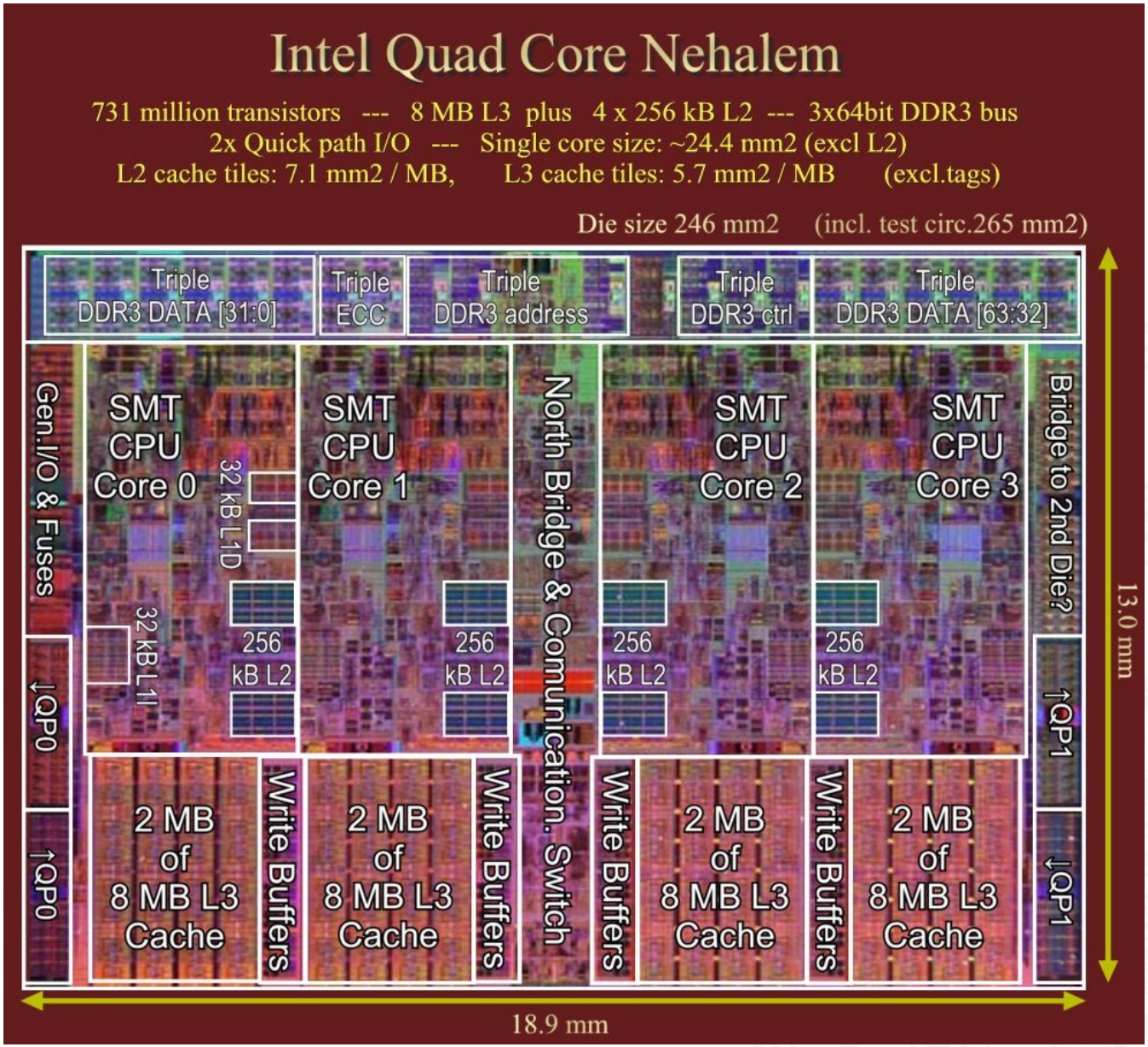
Photo courtesy of Intel Corp.



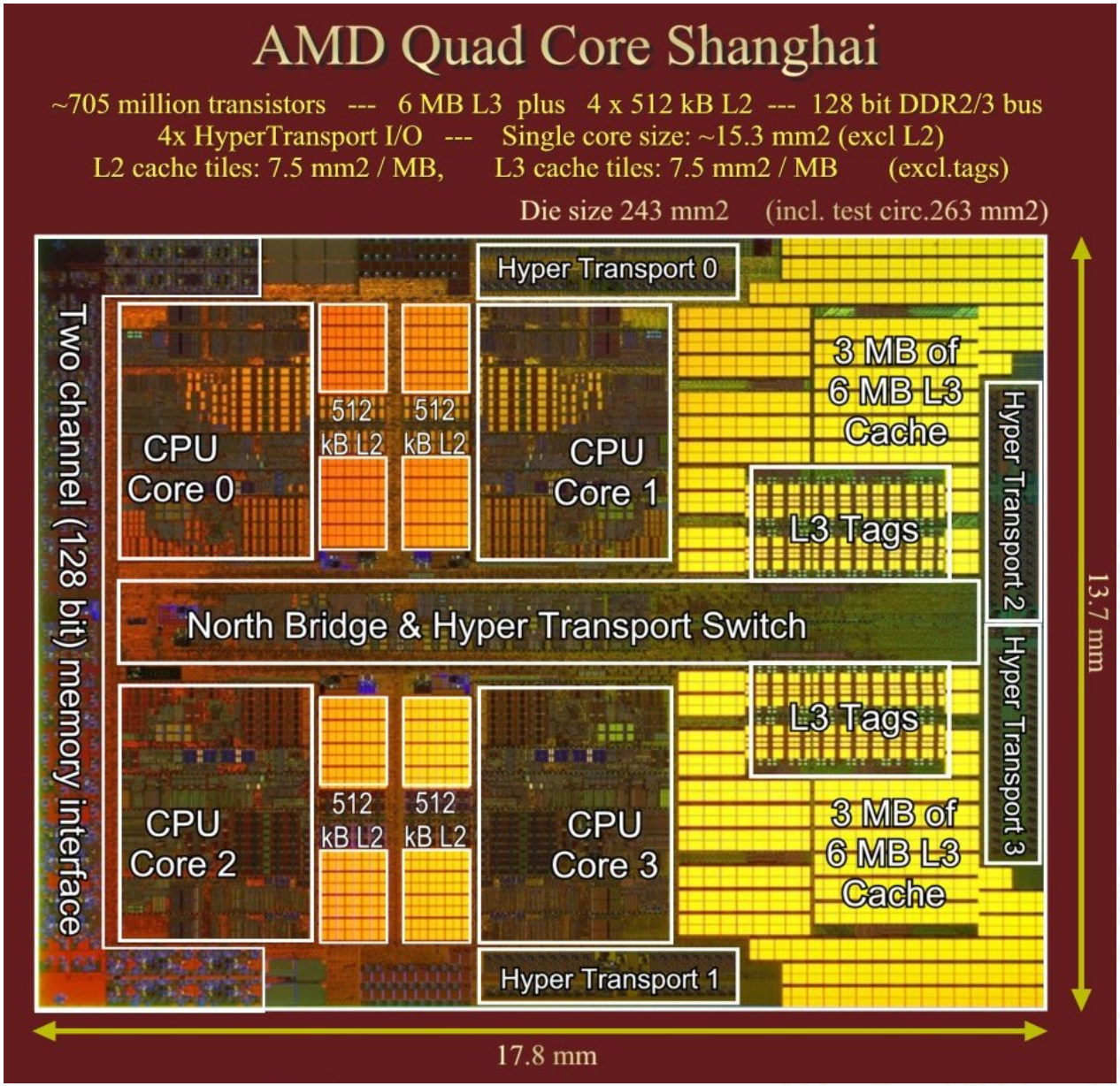
© ARM Limited

- 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

- Wunsch des Programmierers
  - möglichst großer Adressraum, ideal 2^{32} Byte oder größer
  - linear adressierbar

- Sicht des Betriebssystems
  - verwaltet eine Menge laufender Tasks / Prozesse
  - jedem Prozess steht nur begrenzter Speicher zur Verfügung
  - strikte Trennung paralleler Prozesse
  - Sicherheitsmechanismen und Zugriffsrechte
  - read-only Bereiche für Code
  - read-write Bereiche für Daten

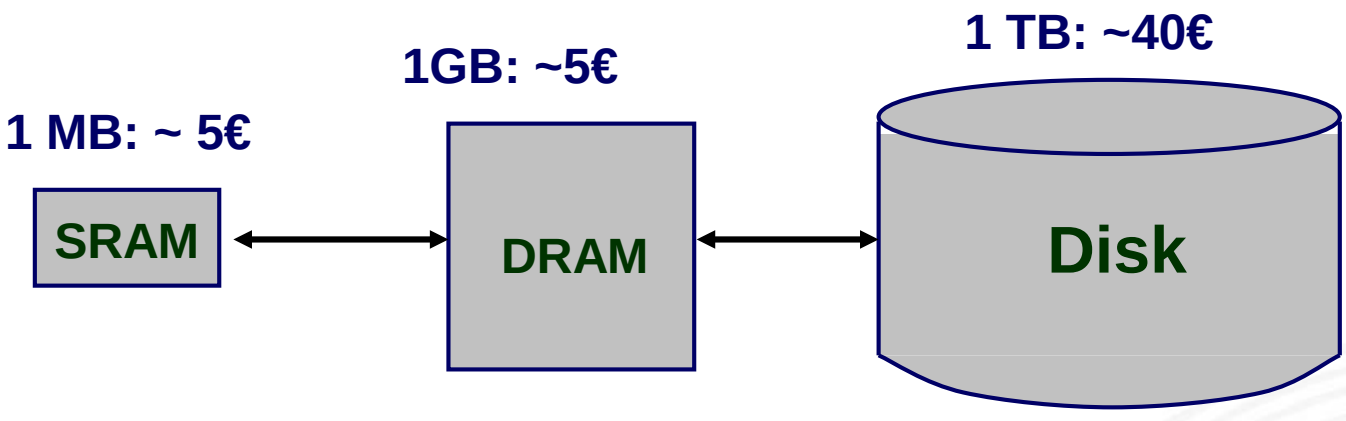
$\Rightarrow$ widersprüchliche Anforderungen

$\Rightarrow$ Lösung mit virtuellem Speicher und Paging

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

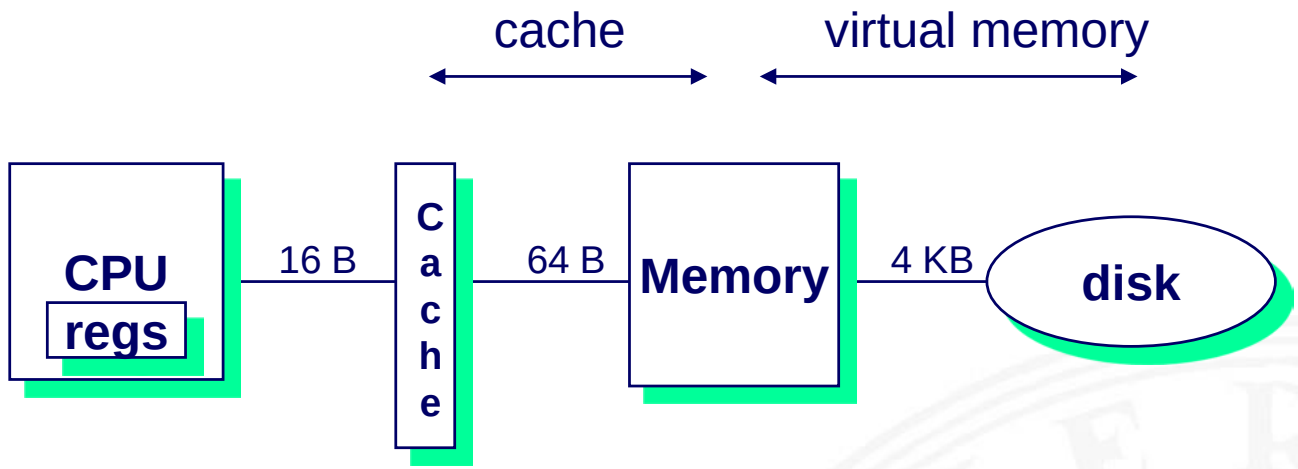


[BO15]

- Vollständiger Adressraum zu groß ⇒ DRAM ist *Cache*
  - 32-bit Adressen: $\approx 4 \cdot 10^9$ Byte 4 Milliarden
  - 64-bit Adressen: $\approx 16 \cdot 10^{16}$ 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



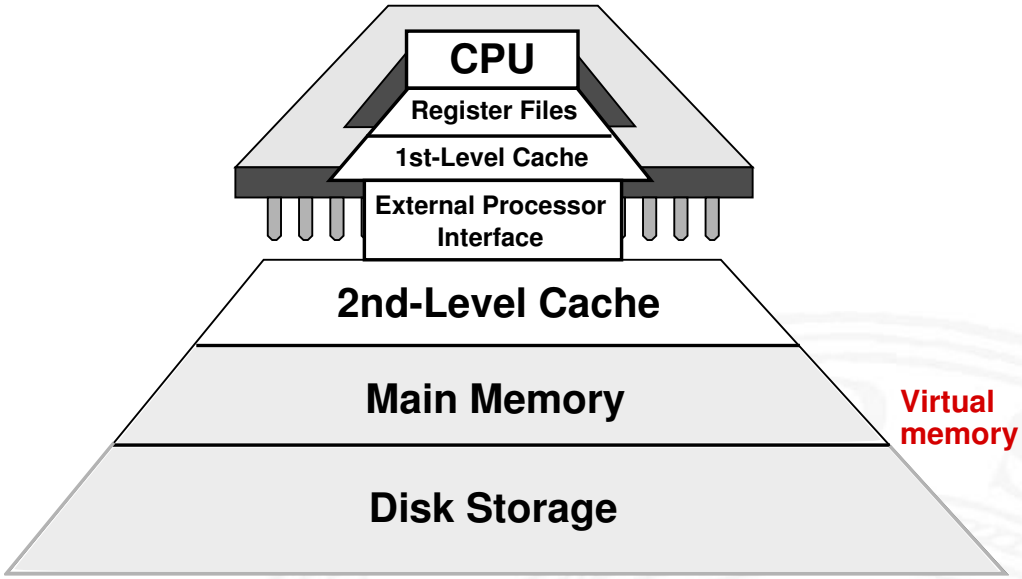
	Register	Cache	Memory	Disk Memory
size:	64 B	32 KB-12MB	8 GB	2 TB
speed:	300 ps	1 ns	8 ns	4 ms
$/Mbyte:		5€/MB	5€/GB	4 Ct./GB
line size:	16 B	64 B	4 KB	

larger, slower, cheaper →

[BO15]

Ebenen in der Speicherhierarchie (cont.)

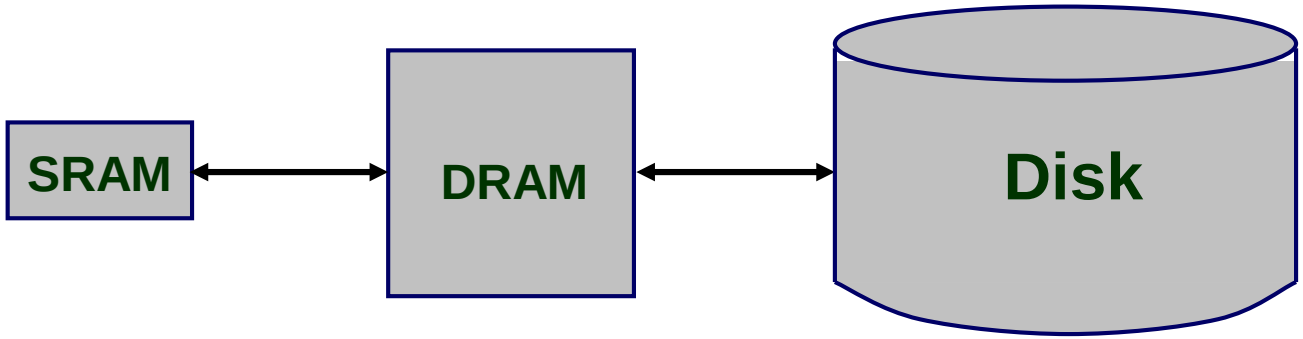
- Hauptspeicher als Cache für den Plattenspeicher



- Parameter der Speicherhierarchie

	1st-Level Cache	virtueller Speicher
Blockgröße	16-128 Byte	4-64 KiByte
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 %
Adressraum	14-20 bit	25-45 bit

Ebenen in der Speicherhierarchie (cont.)

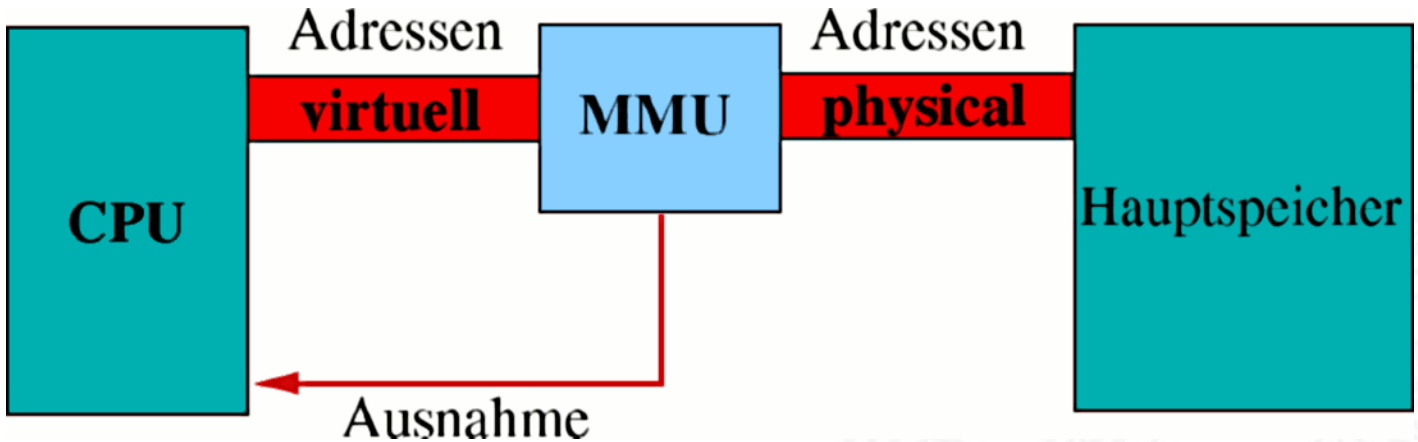


[BO15]

- 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



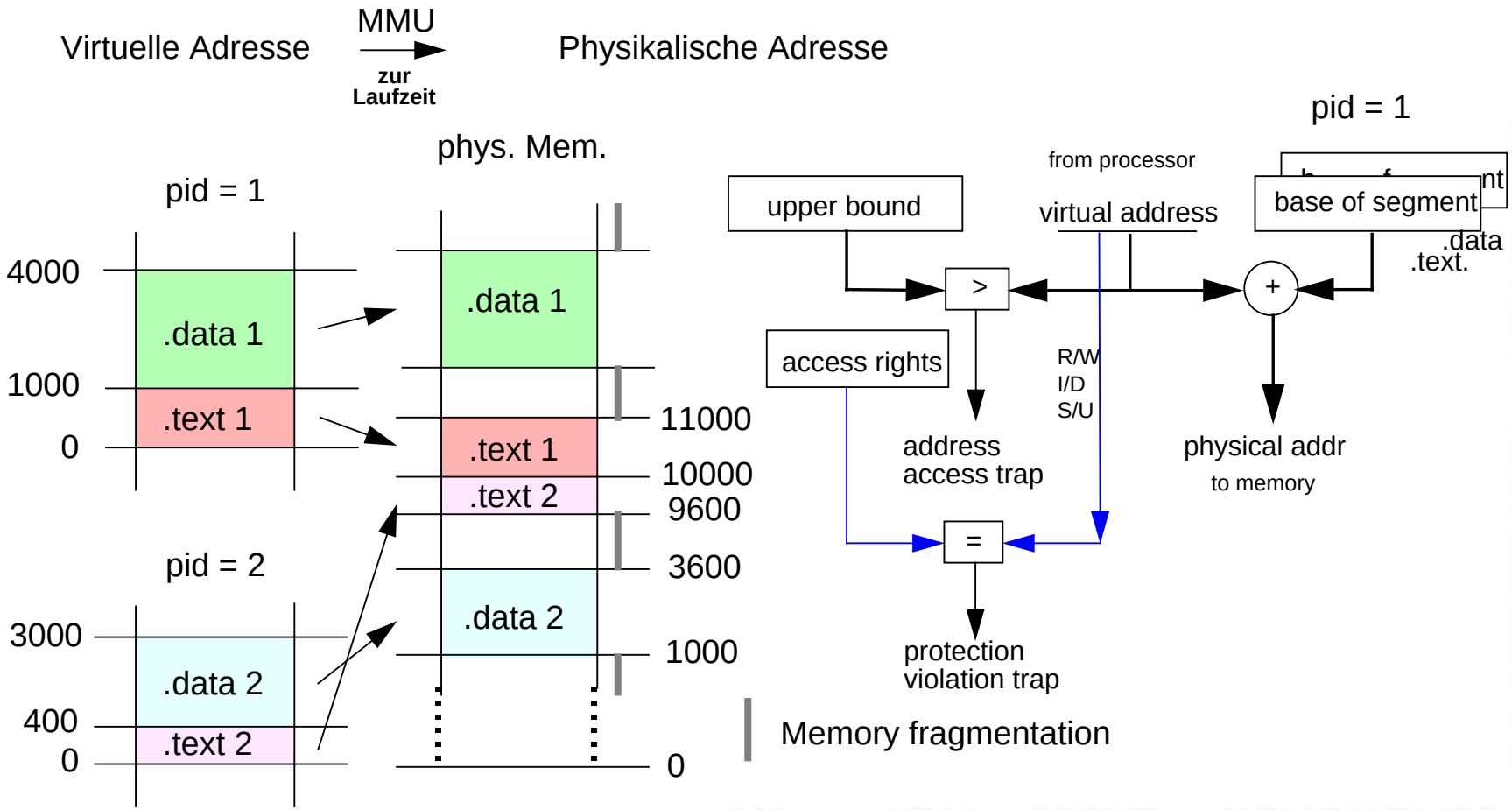
  - Umsetzung von virtuellen zu physikalischen Adressen, Programm-Relokation
  - Umsetzungstabellen werden vom Betriebssystem verwaltet
  - wegen des Speicherbedarfs der Tabellen beziehen sich diese auf größere Speicherblöcke (*Segmente* oder *Seiten*)

Prinzip des virtuellen Speichers (cont.)

- 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.)

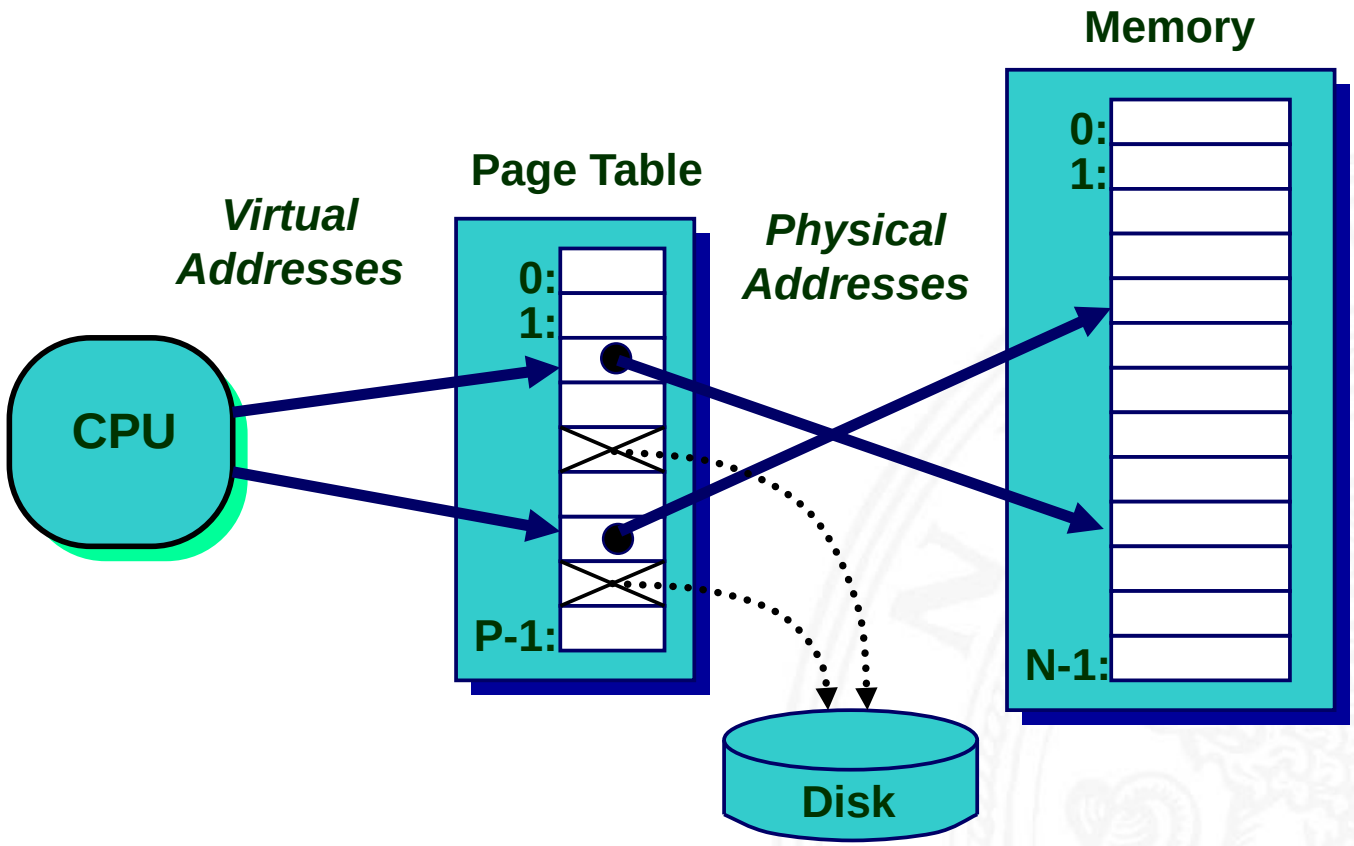
- Idee: Trennung von Instruktionen, Daten und Stack

$\Rightarrow$ 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



[BO15]

Virtueller Speicher: Paging / Seitenadressierung (cont.)

⇒ Abbildung von *Adressen* in den *virtuellen Speicher*: Hauptspeicher + Festplatte

\+ 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

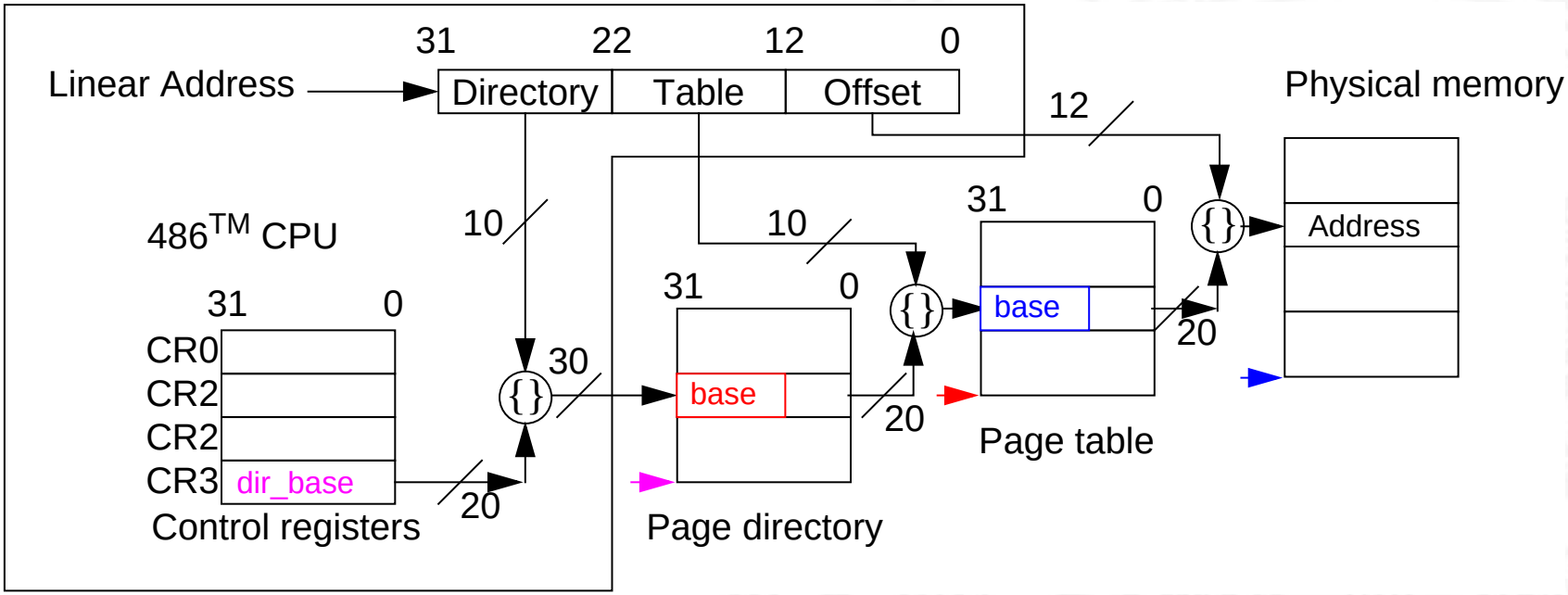
\+ Zugriffsrechte für jede Seite (read/write, User/Supervisor)

\+ gemeinsam genutzte Programmteile/-Bibliotheken können sehr einfach in das Konzept integriert werden

- Windows: `.dll`-Dateien
- Unix/Linux: `.so`-Dateien

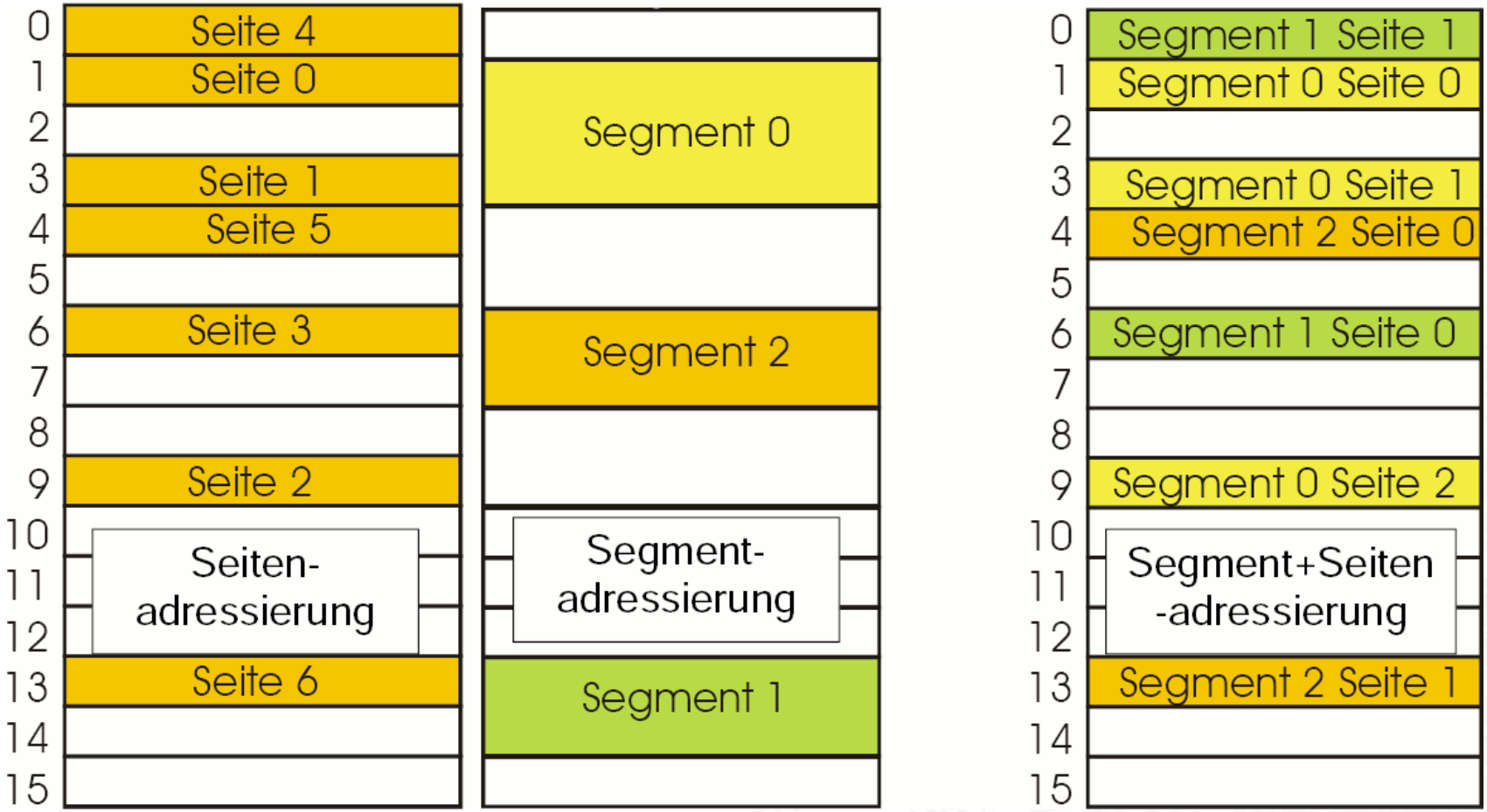
Virtueller Speicher: Paging / Seitenadressierung (cont.)

- große Miss-Penalty (Nachladen von der Platte)
  - $\Rightarrow$ Seiten sollten relativ groß sein: 4...64 KiByte
- Speicherplatzbedarf der Seitentabelle
  viel virtueller Speicher, 4 KiByte Seitengröße
  - $=$ sehr große Pagetable
  - $\Rightarrow$ Hash-Verfahren (*inverted page tables*)
  - $\Rightarrow$ mehrstufige Verfahren



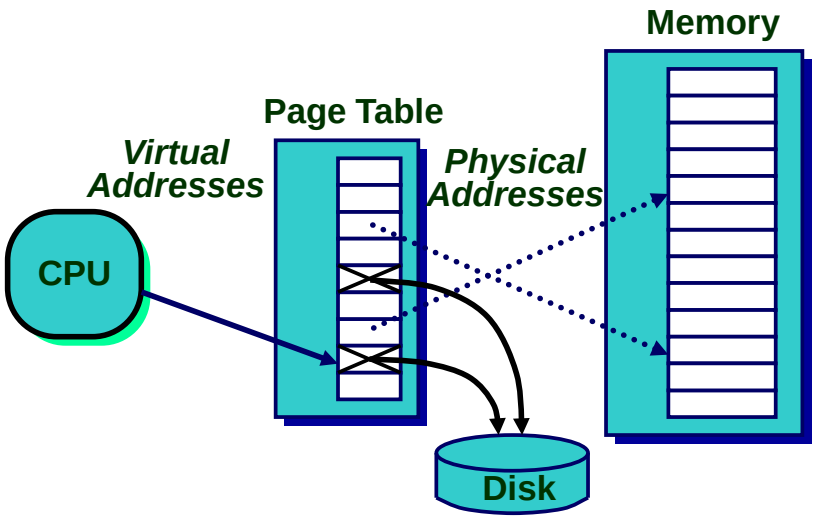
Virtueller Speicher: Segmentierung + Paging

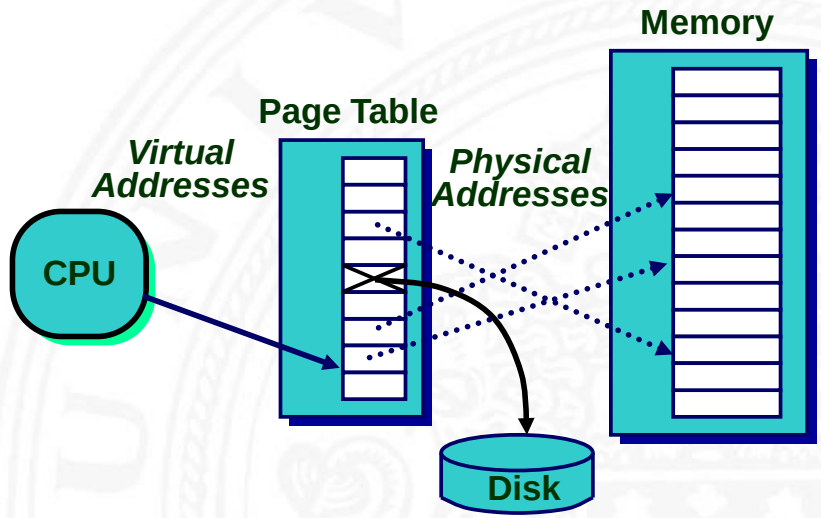
aktuell = Mischung: Segmentierung und Paging (seit I386)



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.



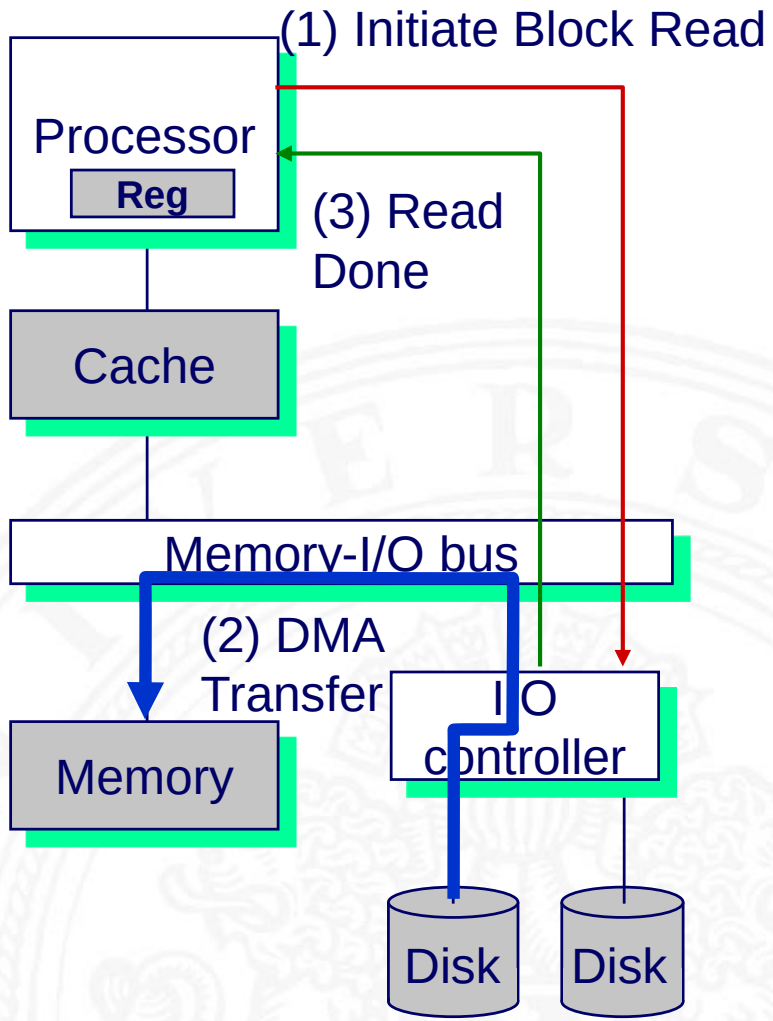


[BO15]

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



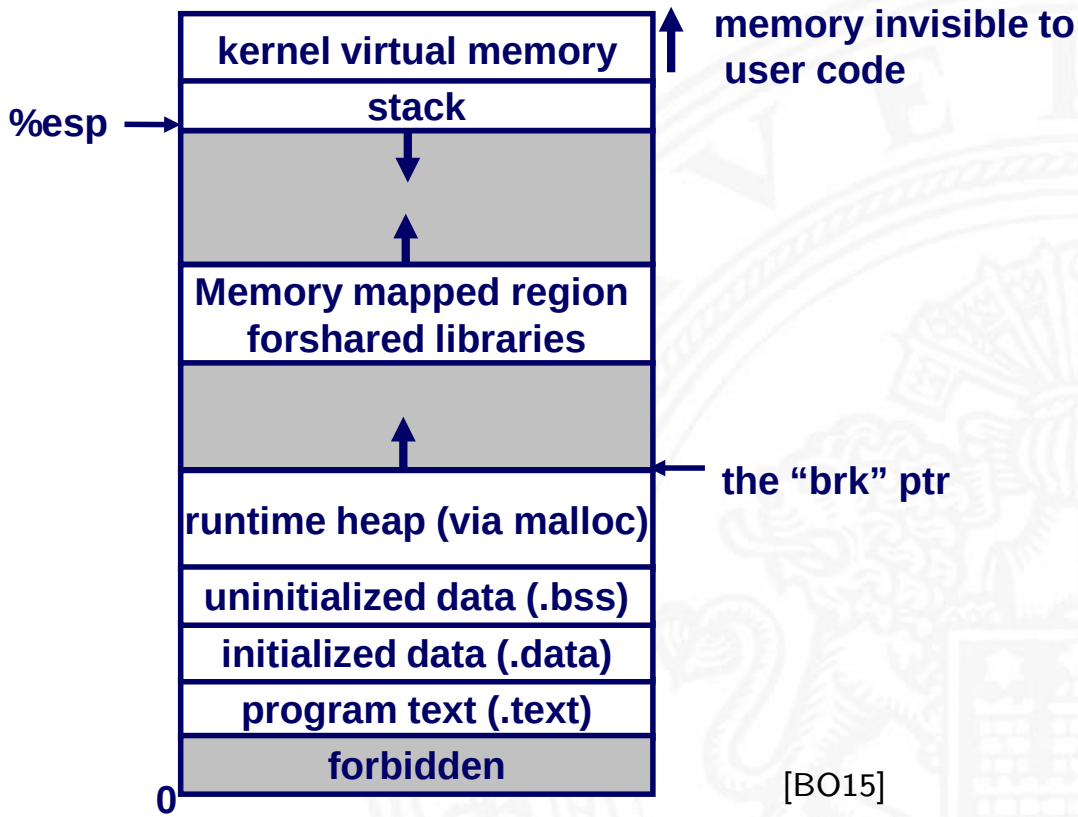
[BO15]

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

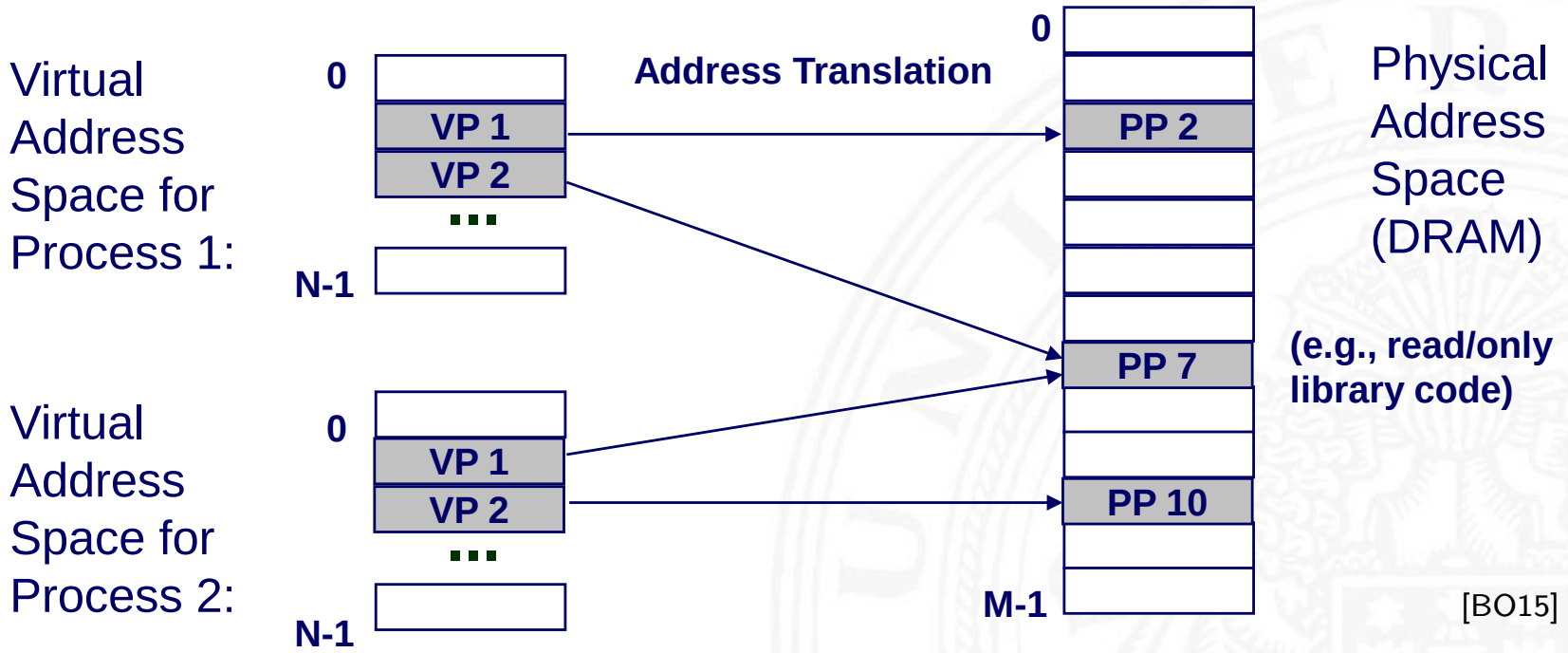


[BO15]

Separate virtuelle Adressräume (cont.)

Auflösung der Adresskonflikte

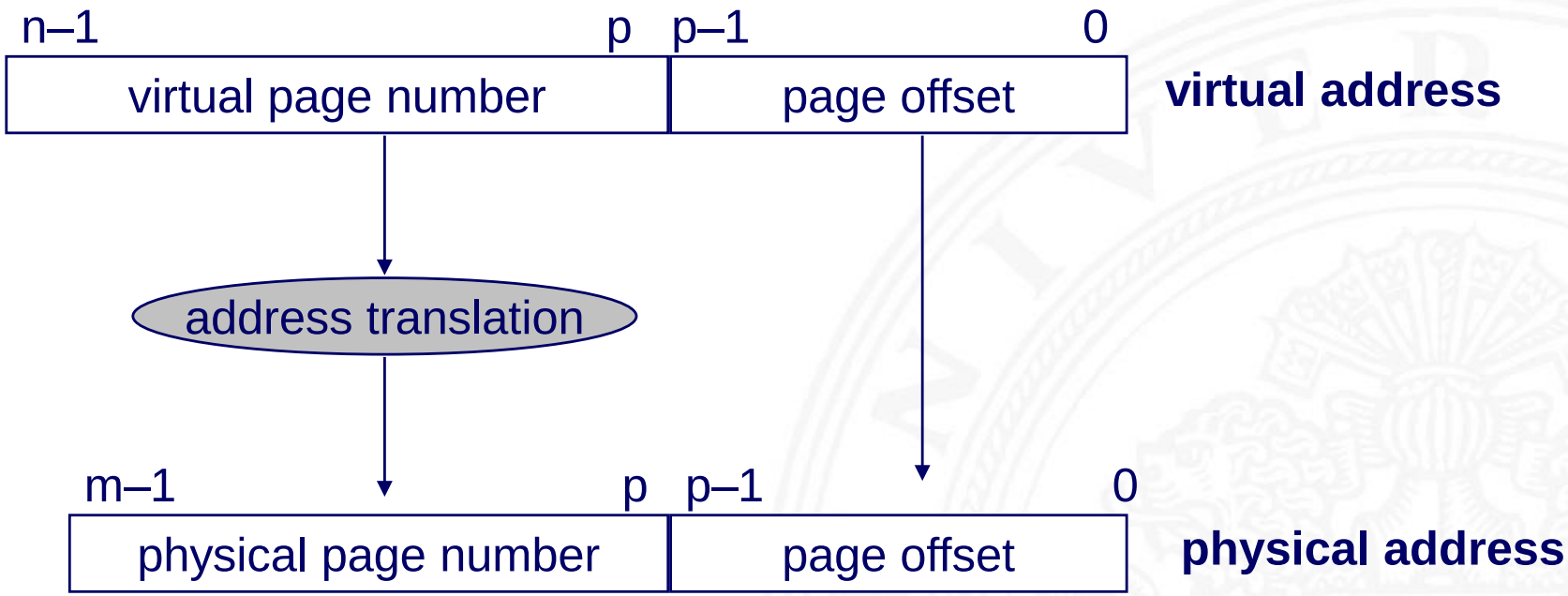
- jeder Prozess hat seinen eigenen virtuellen Adressraum
- Betriebssystem kontrolliert wie virtuelle Seiten auf den physikalischen Speicher abgebildet werden



[BO15]

Virtueller Speicher – Adressumsetzung

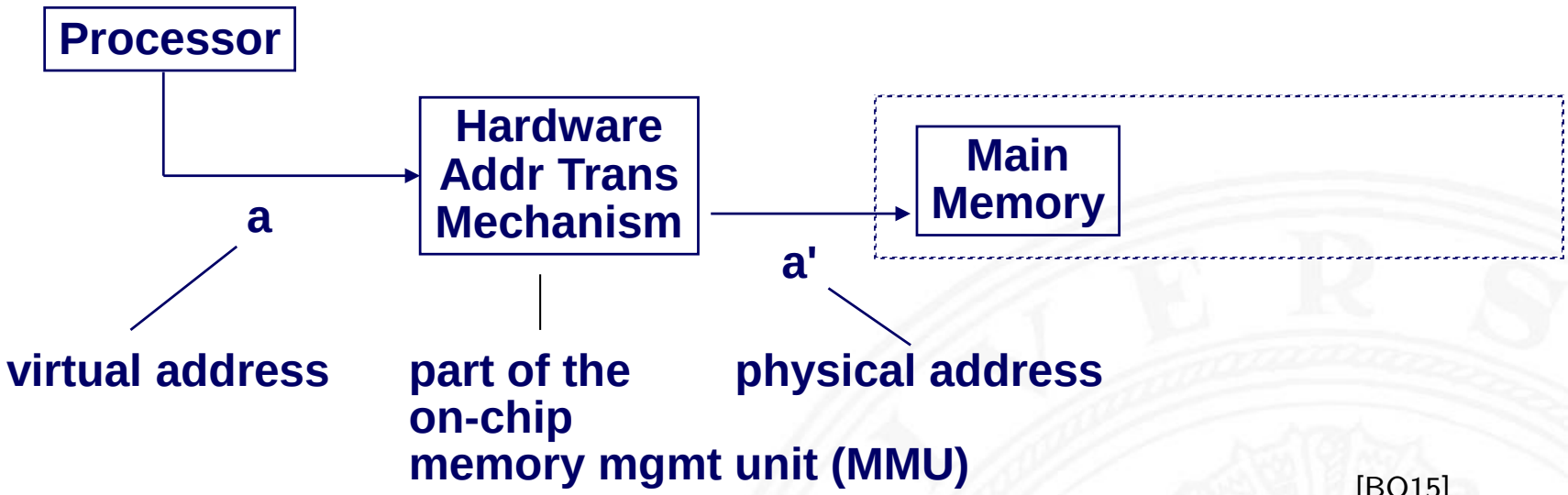
- Parameter
  - P $= 2^p$ = Seitengröße (Bytes)
  - N $= 2^n$ = Limit der virtuellen Adresse
  - M $= 2^m$ = Limit der physikalischen Adresse



[BO15]

Virtueller Speicher – Adressumsetzung (cont.)

- virtuelle Adresse: Hit

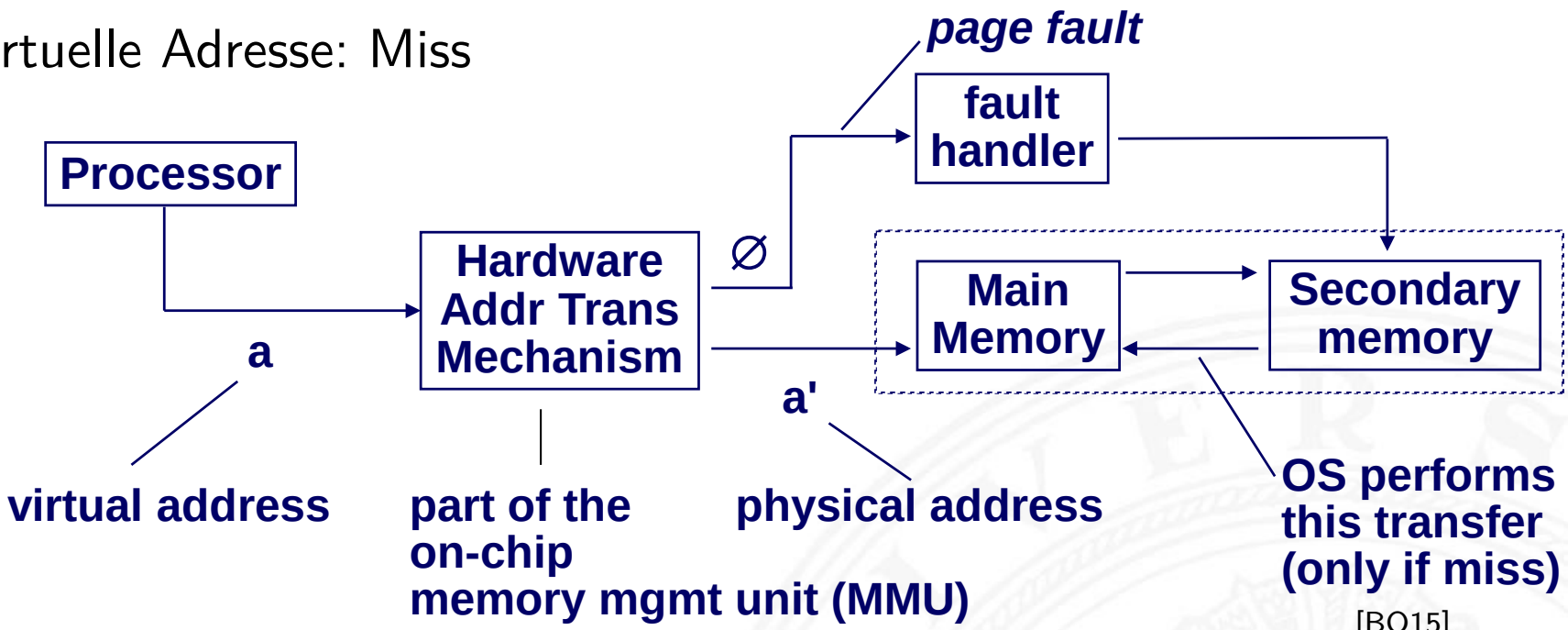


[BO15]

- Programm greift auf virtuelle Adresse a zu
- MMU überprüft den Zugriff, liefert physikalische Adresse a'
- Speicher liefert die zugehörigen Daten $d[a']$

Virtueller Speicher – Adressumsetzung (cont.)

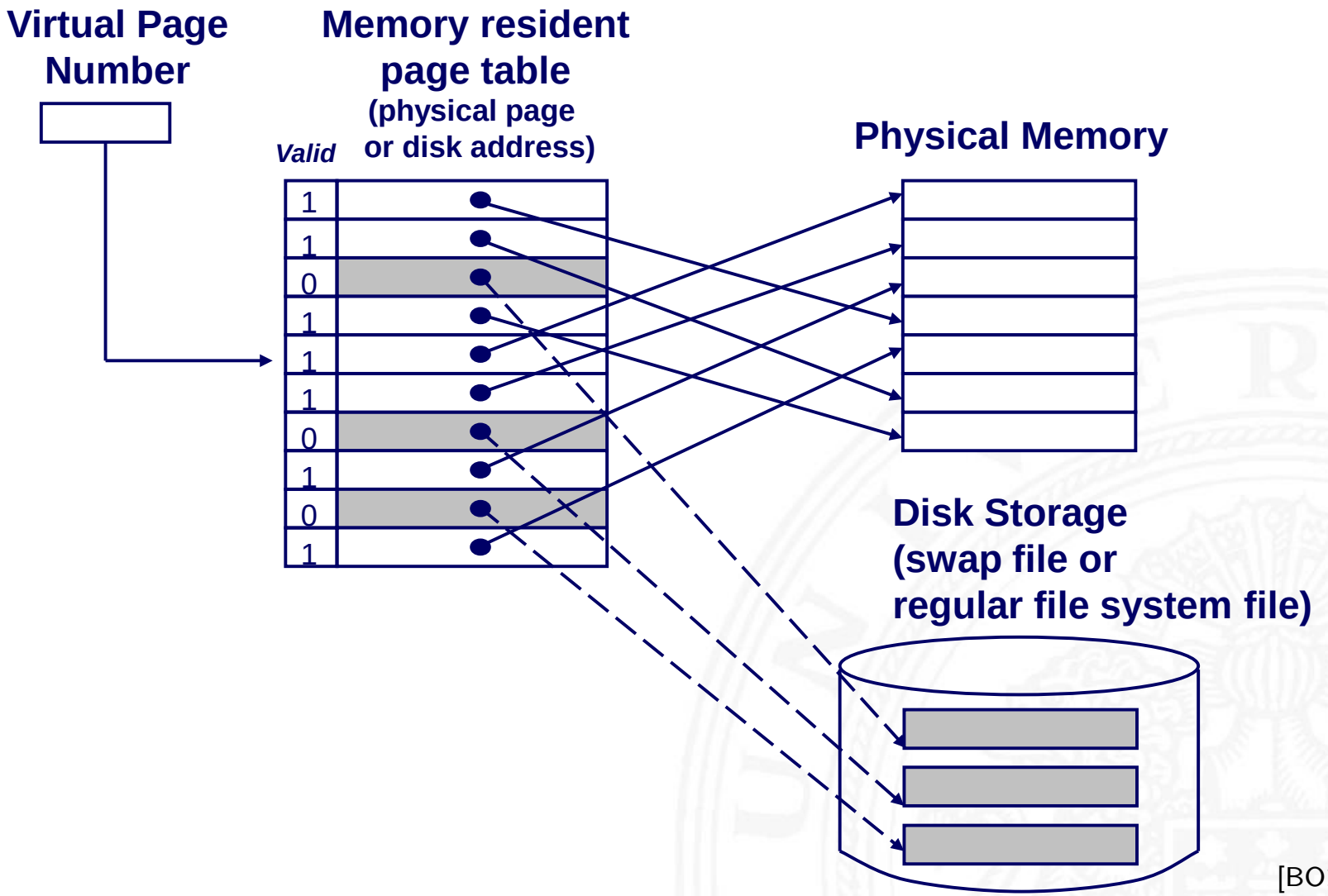
- virtuelle Adresse: Miss



[BO15]

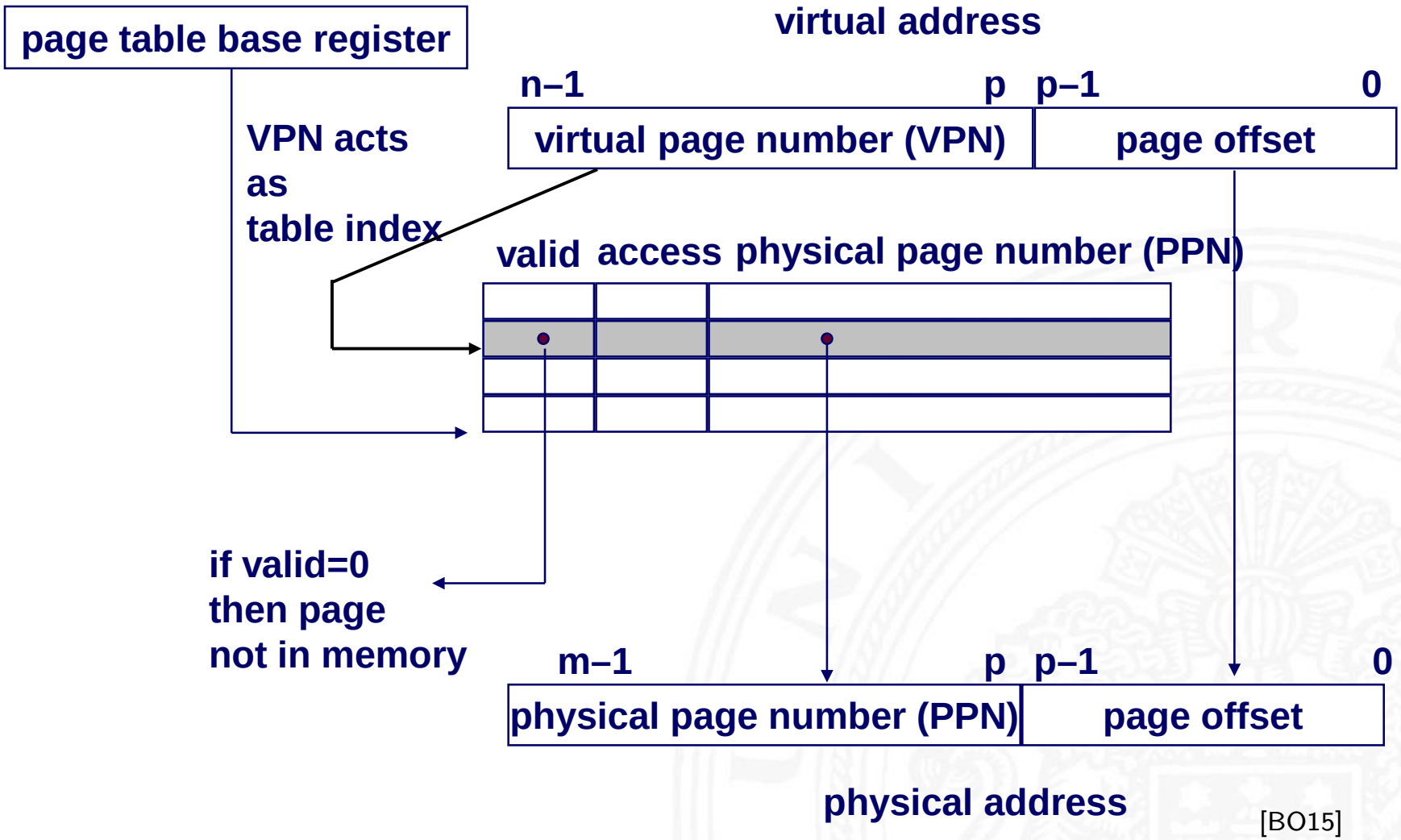
- Programm greift auf virtuelle Adresse *a* zu
- MMU überprüft den Zugriff, Adresse nicht in Hauptspeicher
- „page-fault“ ausgelöst, Betriebssystem übernimmt

Seiten-Tabelle



[BO15]

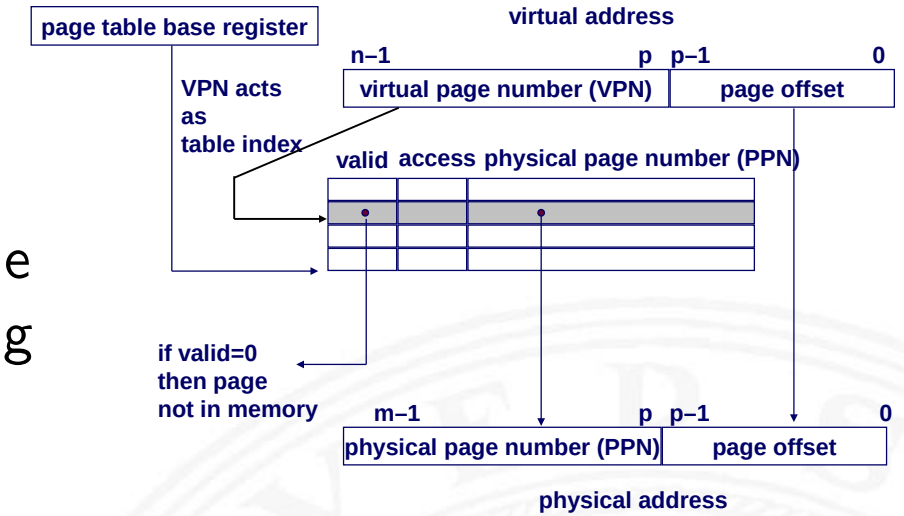
Seiten-Tabelle (cont.)



[BO15]

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

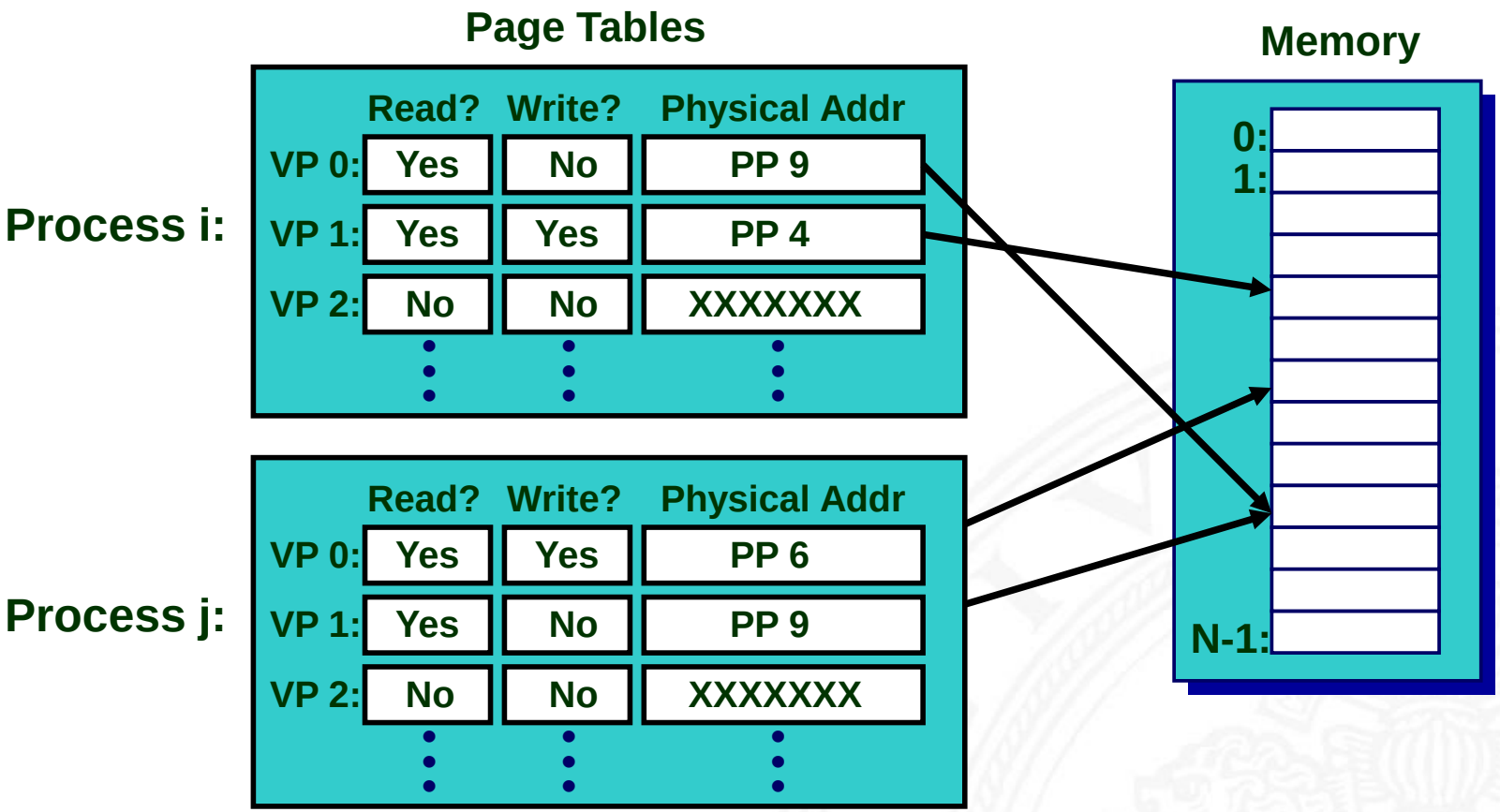


Zugriffsrechte

Schutzüberprüfung

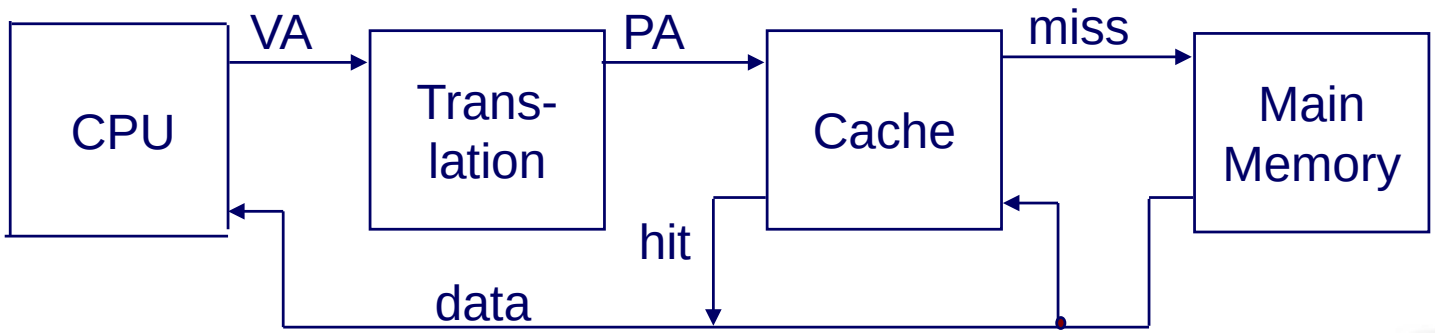
- Zugriffsrechtefeld gibt Zugriffserlaubnis an
  - typischerweise werden zahlreiche Schutzmodi unterstützt
  - Unterscheidung zwischen Kernel- und User-Mode
  - z.B. read-only, read-write, execute-only, no-execute
  - no-execution Bits gesetzt für Stack-Pages: Erschwerung von Buffer-Overflow-Exploits
- Schutzrechteverletzung wenn Prozess/Benutzer nicht die nötigen Rechte hat
- bei Verstoß erzwingt die Hardware den Schutz durch das Betriebssystem („Trap“ / „Exception“)

Zugriffsrechte (cont.)



[BO15]

Integration von virtuellem Speicher und Cache



[BO15]

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

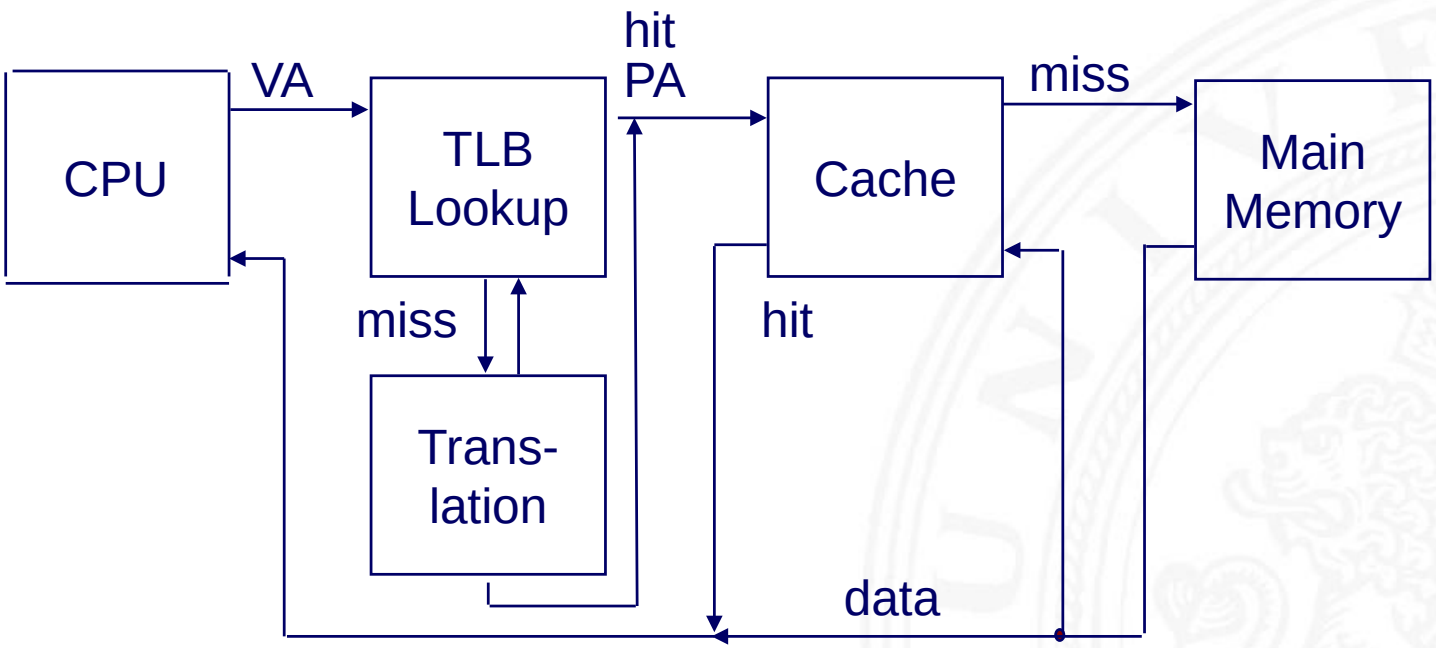
Die Adressumsetzung wird vor dem Cache „Lookup“ durchgeführt

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

TLB / „Translation Lookaside Buffer“

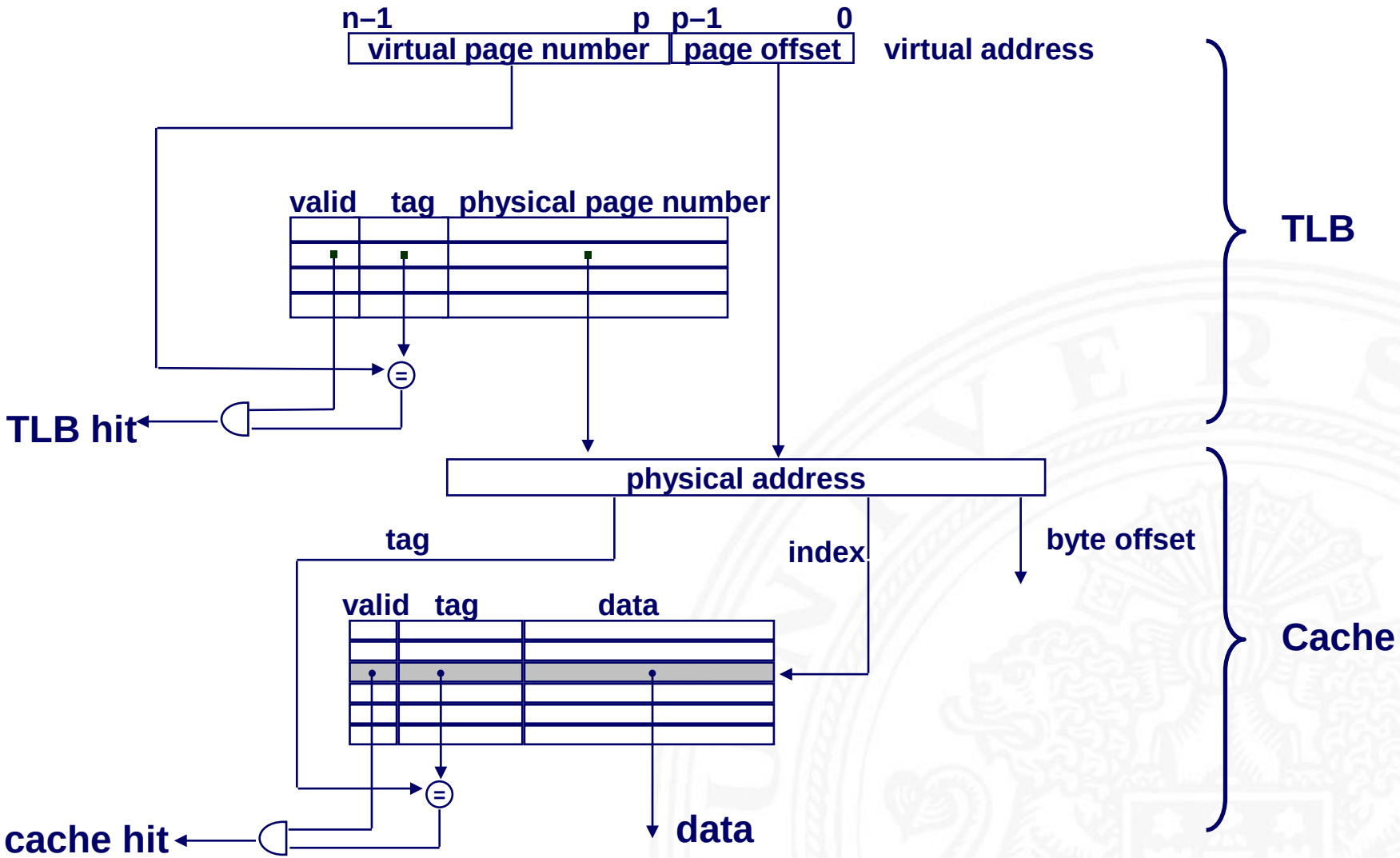
Beschleunigung der Adressumsetzung für virtuellen Speicher

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



[BO15]

TLB / „Translation Lookaside Buffer“ (cont.)



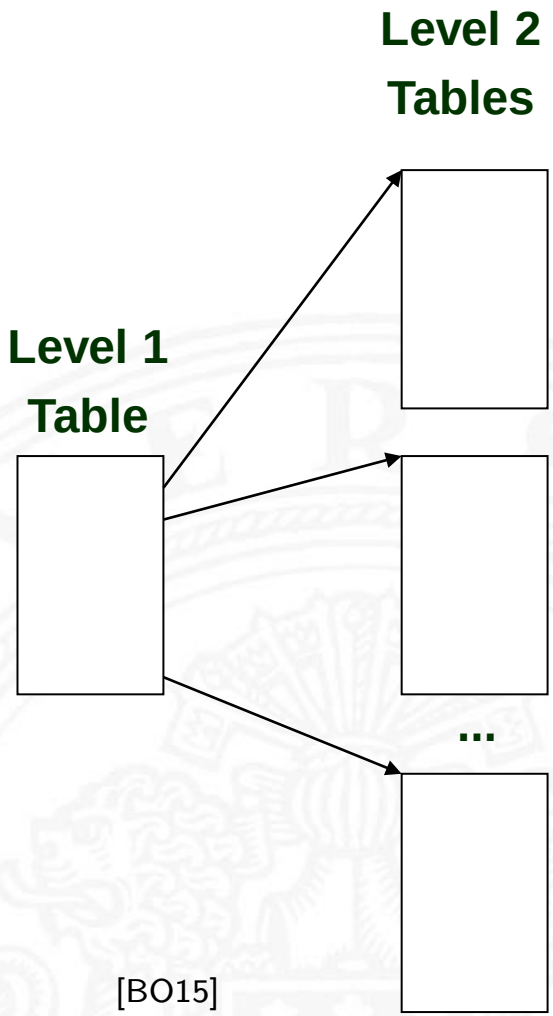
[BO15]

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

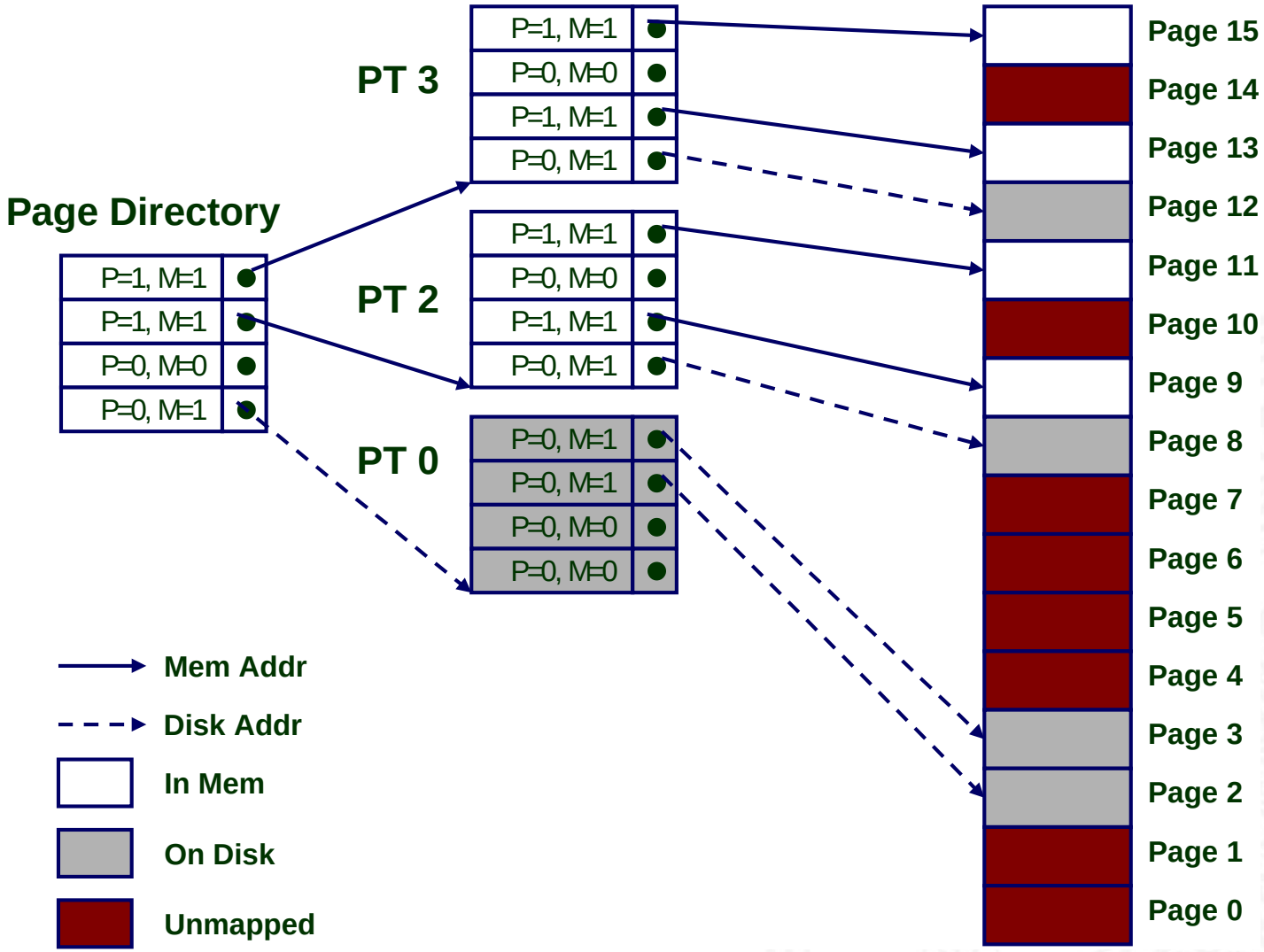
$\Rightarrow$ übliche Lösung

- 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



[BO15]

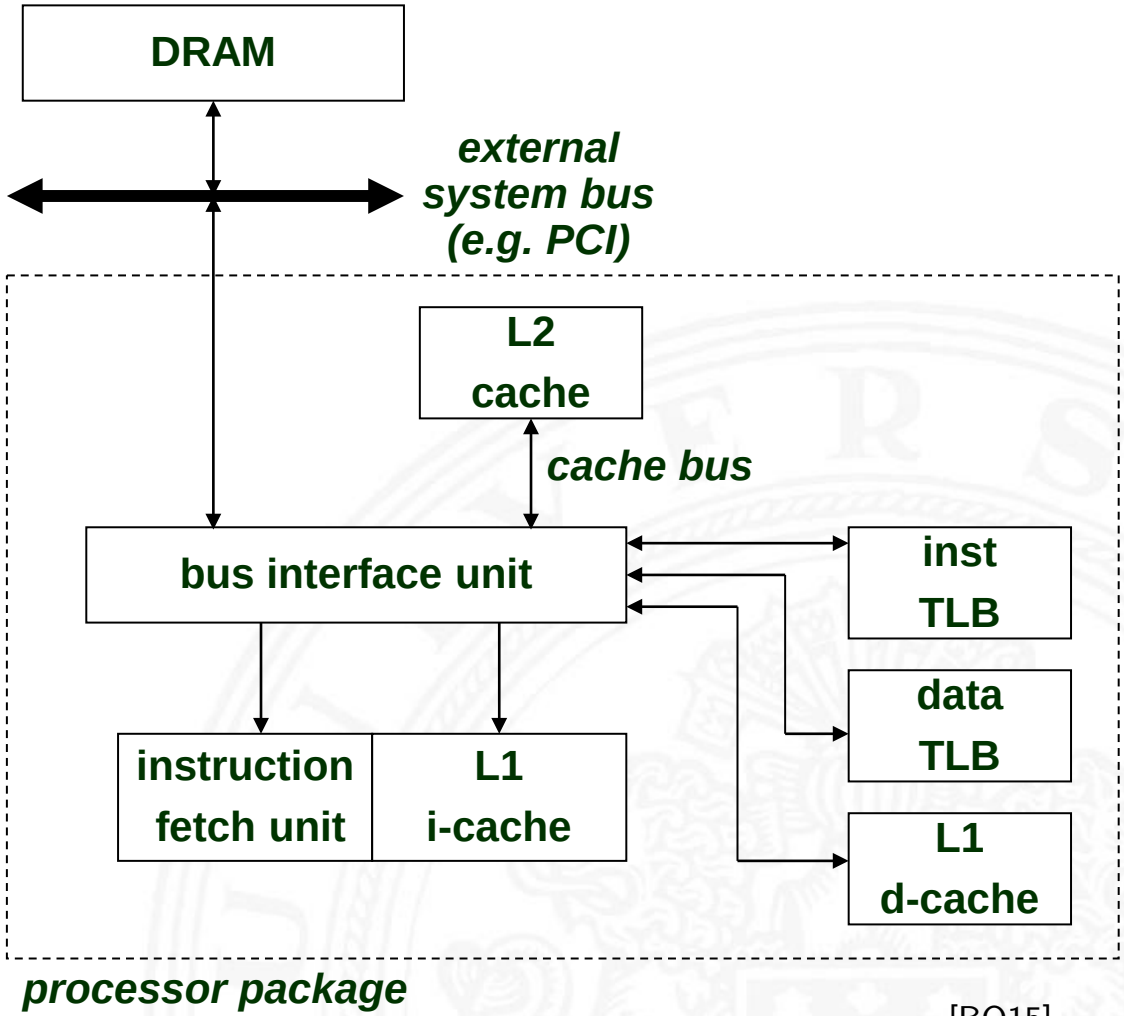
mehrstufige Seiten-Tabellen (cont.)



[BO15]

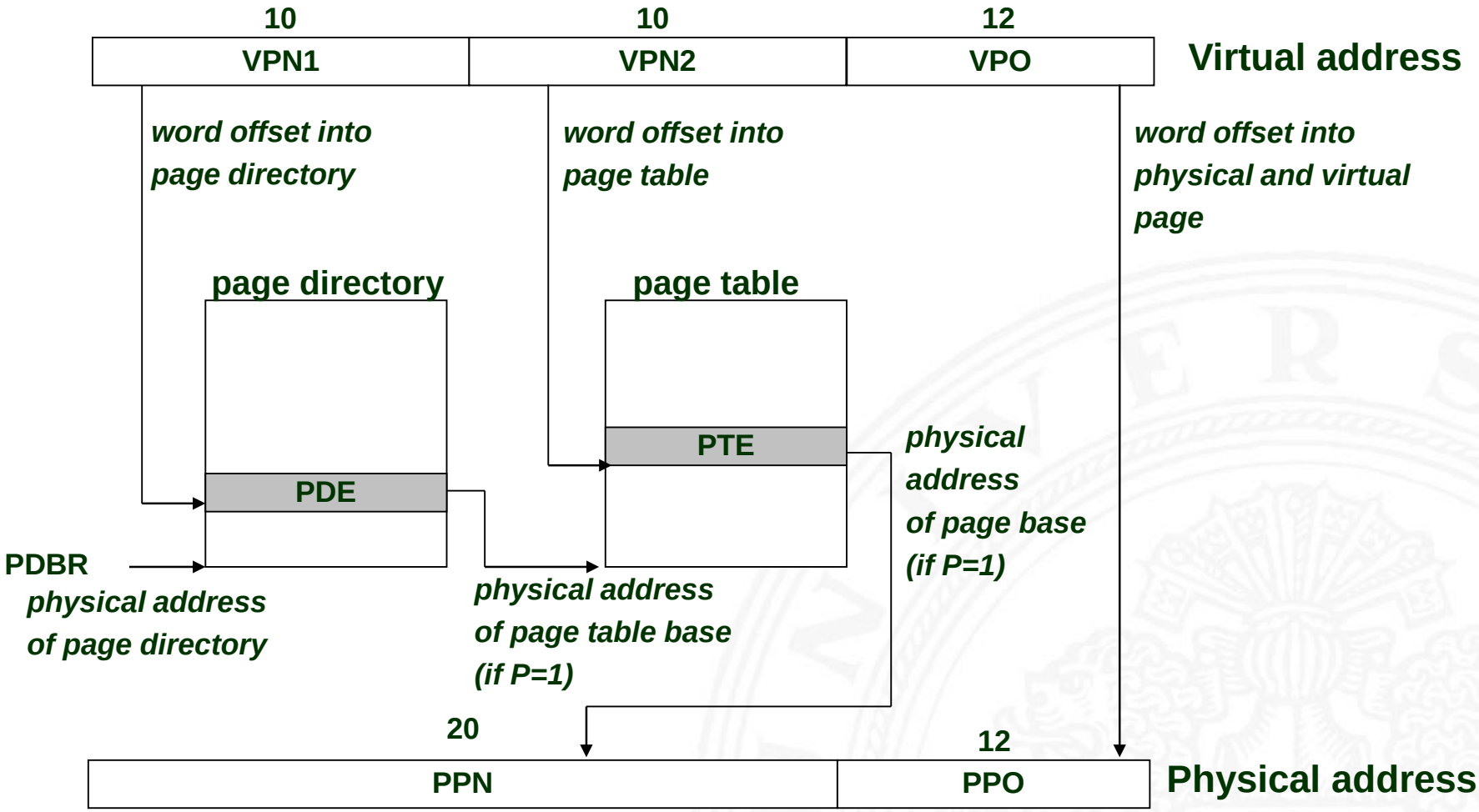
Beispiel: Pentium und Linux

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



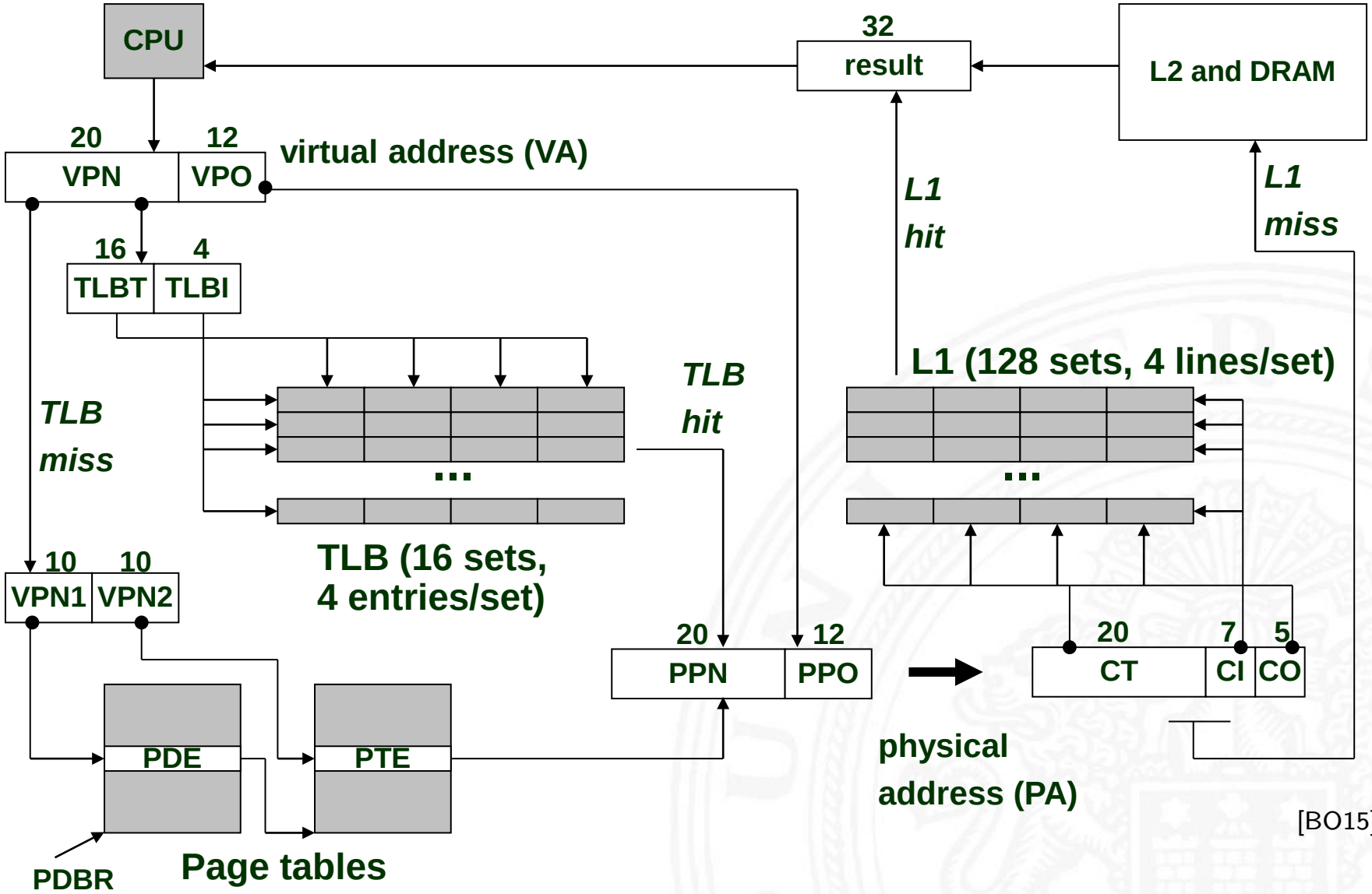
[BO15]

Beispiel: Pentium und Linux (cont.)



[BO15]

Beispiel: Pentium und Linux (cont.)



[BO15]

Zusammenfassung

Cache Speicher

- dient nur zur Beschleunigung
- unsichtbar für Anwendungsprogrammierer und OS
- komplett in Hardware implementiert

Virtueller Speicher

- ermöglicht viele Funktionen des Betriebssystems
  - größerer virtueller Speicher als reales DRAM
  - Auslagerung von Daten auf die Festplatte
  - 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
- Programm „besitzt“ die gesamte Maschine
  - hat privaten Adressraum
  - bleibt unberührt vom Verhalten anderer Prozesse

Sicht des Systems

- Adressraum von Prozessen auf Seiten abgebildet
  - muss nicht fortlaufend sein
  - wird dynamisch zugeteilt
  - erzwingt Schutz bei Adressumsetzung
- Betriebssystem verwaltet viele Prozesse gleichzeitig
  - jederzeit schneller Wechsel zwischen Prozessen
  - u.a. beim Warten auf Ressourcen (Seitenfehler)

Literatur

[BO15] R.E. Bryant, D.R. O'Hallaron:
Computer systems – A programmers perspective.
3rd global ed., Pearson Education Ltd., 2015.
ISBN 978–1–292–10176–7. `csapp.cs.cmu.edu`

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–86894–238–5

[Fur00] S. Furber: *ARM System-on-Chip Architecture*.
2nd edition, Pearson Education Limited, 2000.
ISBN 978–0–201–67519–1

Literatur (cont.)

[HP12] J.L. Hennessy, D.A. Patterson:
Computer architecture – A quantitative approach.
5th edition, Morgan Kaufmann Publishers Inc., 2012.
ISBN 978–0–12–383872–8

[PH16a] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface: ARM Edition*.
Morgan Kaufmann Publishers Inc., 2016.
ISBN 978–0–12–801733–3

[PH16b] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle*.
5. Auflage, Oldenbourg, 2016. ISBN 978–3–11–044605–0