64-040 Modul IP7: Rechnerstrukturen

14 Speicherhierarchie

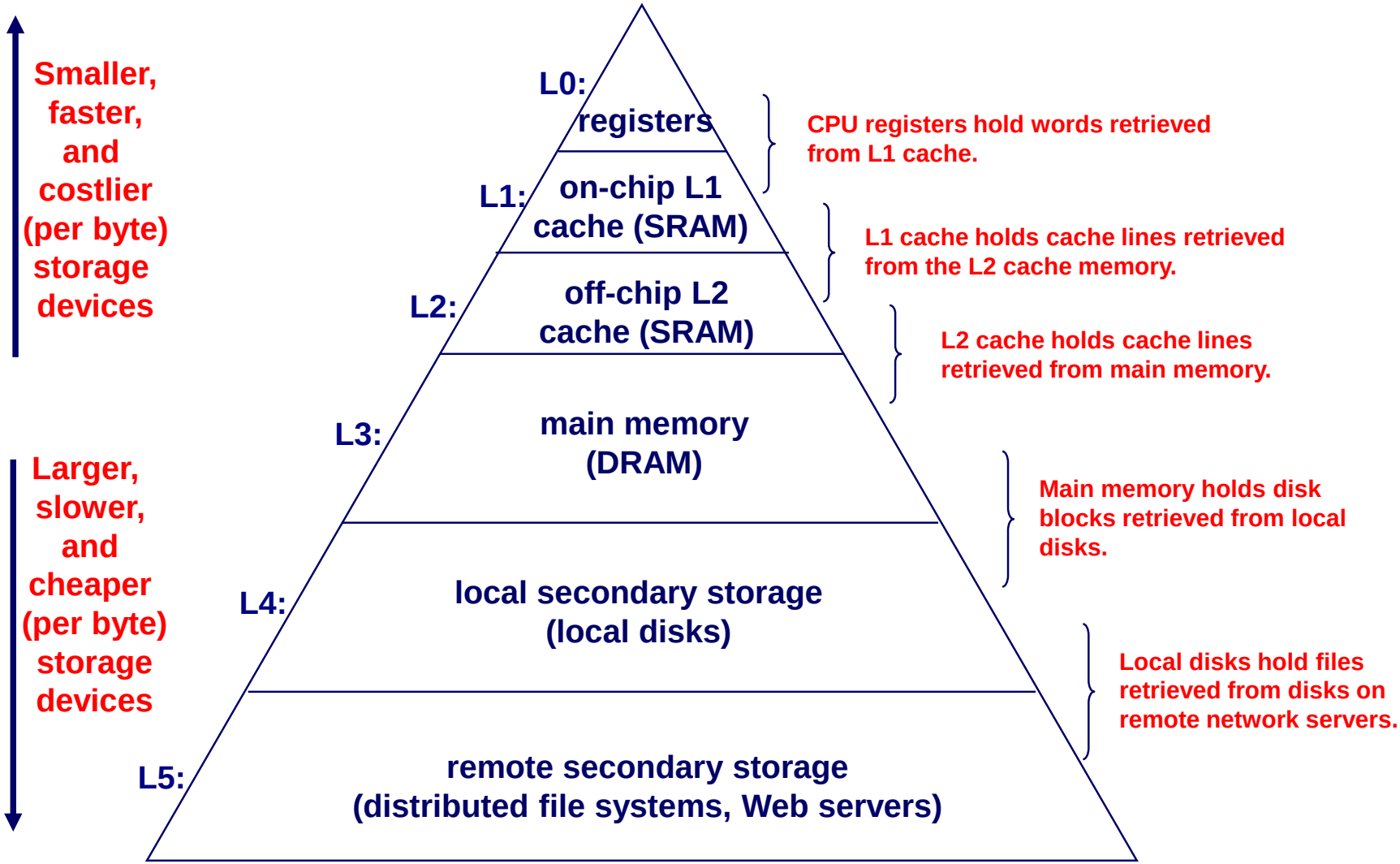
Norman Hendrich

Universität Hamburg
MIN Fakultät, Department Informatik
Vogt-Kölln-Str. 30, D-22527 Hamburg
hendrich@informatik.uni-hamburg.de

WS 2013/2014

Inhalt

Speicherhierarchie: Konzept



Speicherhierarchie: Konzept

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

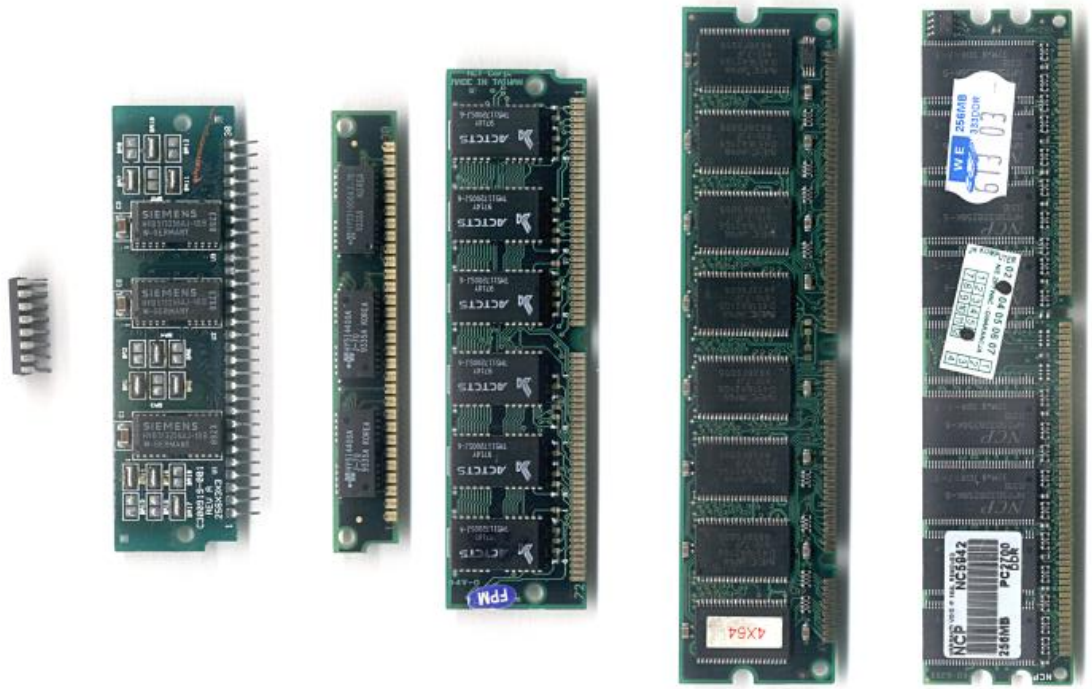
- Illusion aus großem schnellem Speicher
- funktioniert nur wegen räumlicher/zeitlicher Lokalität

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 6T-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. 32 Register á 32-bit

L1-L3: Halbleiterspeicher: SRAM und DRAM

- „Random-Access Memory“ aufgebaut aus Mikrochips
- SRAM (6T-Zelle) oder DRAM (1T-Zelle) Technologie
- mehrere RAM Chips bilden einen Speicher



L4: Festplatten

- dominierende Technologie für nichtflüchtigen Speicher
- hohe Speicherkapazität, derzeit ca. 1 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

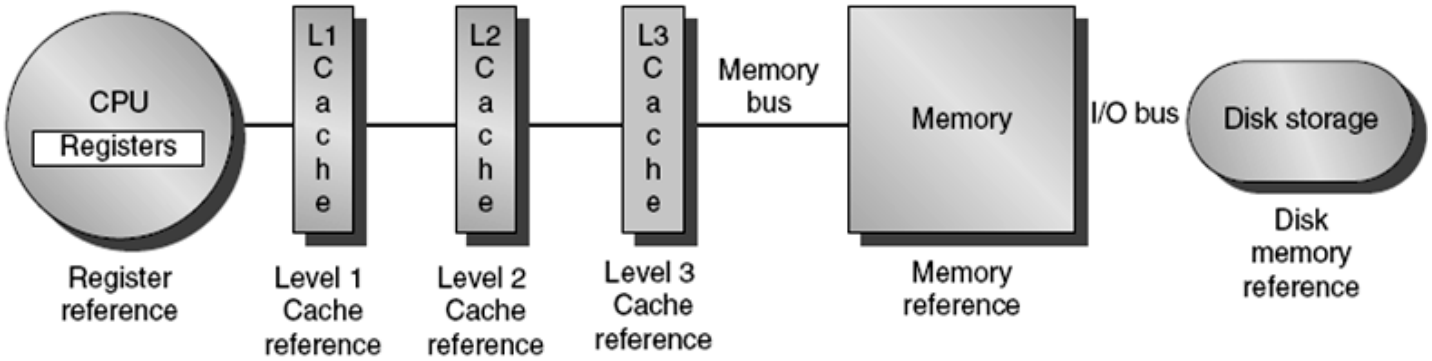
- Archivspeicher und Backup für (viele) Festplatten
  - Magnetbänder
  - RAID-Verbund aus mehreren Festplatten
  - optische Datenspeicher: CD-ROM, DVD-ROM, BlueRay
- WWW und Internet-Services, Cloud-Services

- enorme Speicherkapazität
- langsame Zugriffszeiten

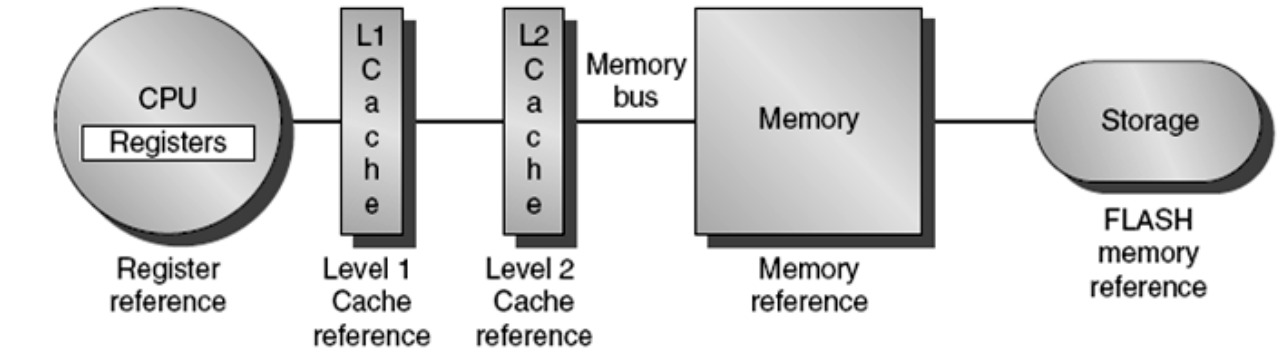
  Cloud-Farms ggf. ähnlich schnell wie L4 Festplatten, da Netzwerk schneller als der Zugriff auf eine lokale Festplatte

- in dieser Vorlesung nicht behandelt

Speicherhierarchie: zwei Beispiele



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

Wiederholung: Halbleiterspeicher

SRAM „statisches RAM“

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

DRAM „dynamisches RAM“

- ein Kondensator und ein Transistor pro Zelle
- der Wert muss alle 10-100 ms aufgefrischt werden
- anfällig für Störungen
- langsamer und billiger als SRAM

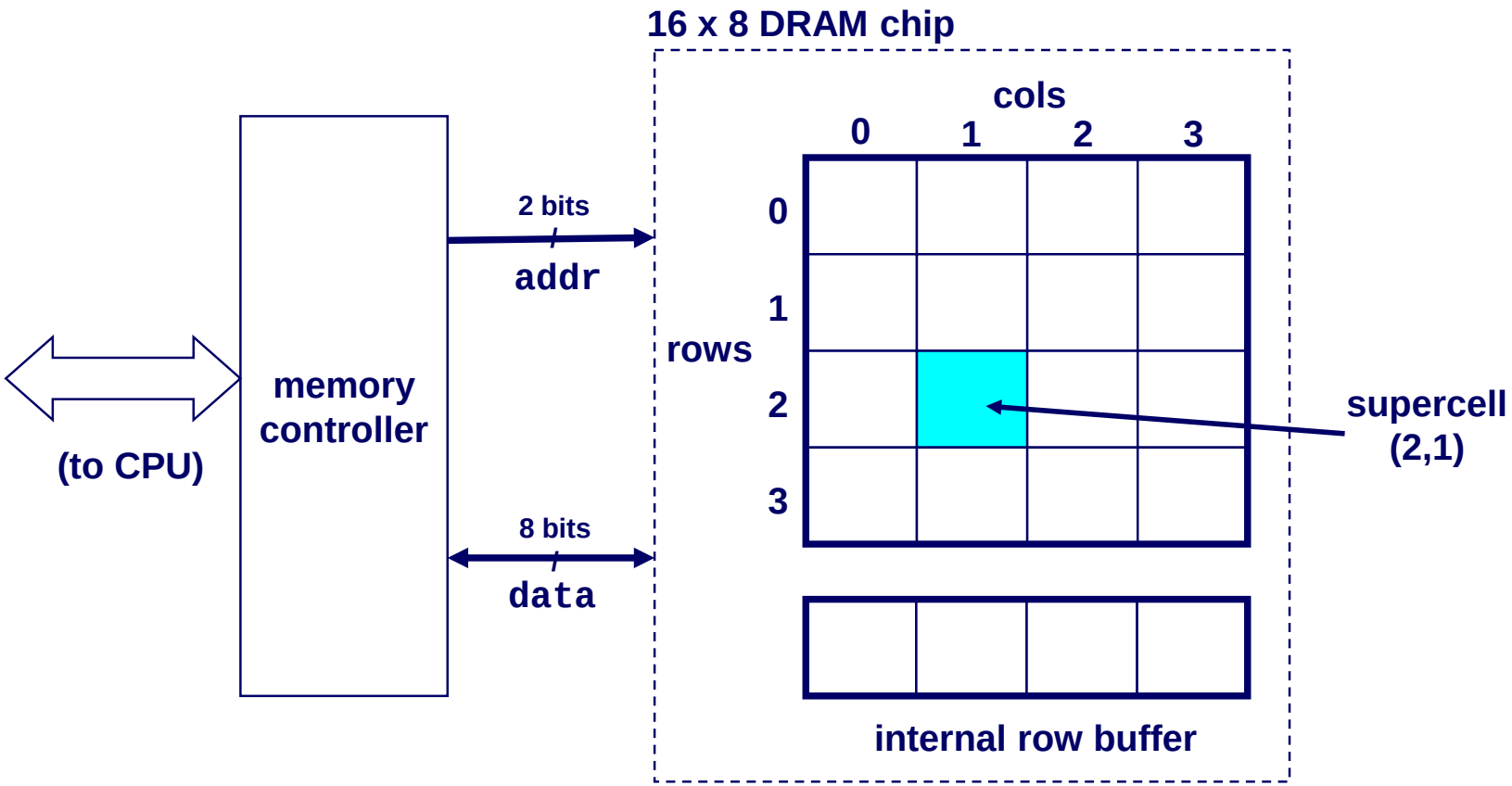
SRAM vs. DRAM

	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

DRAM Organisation

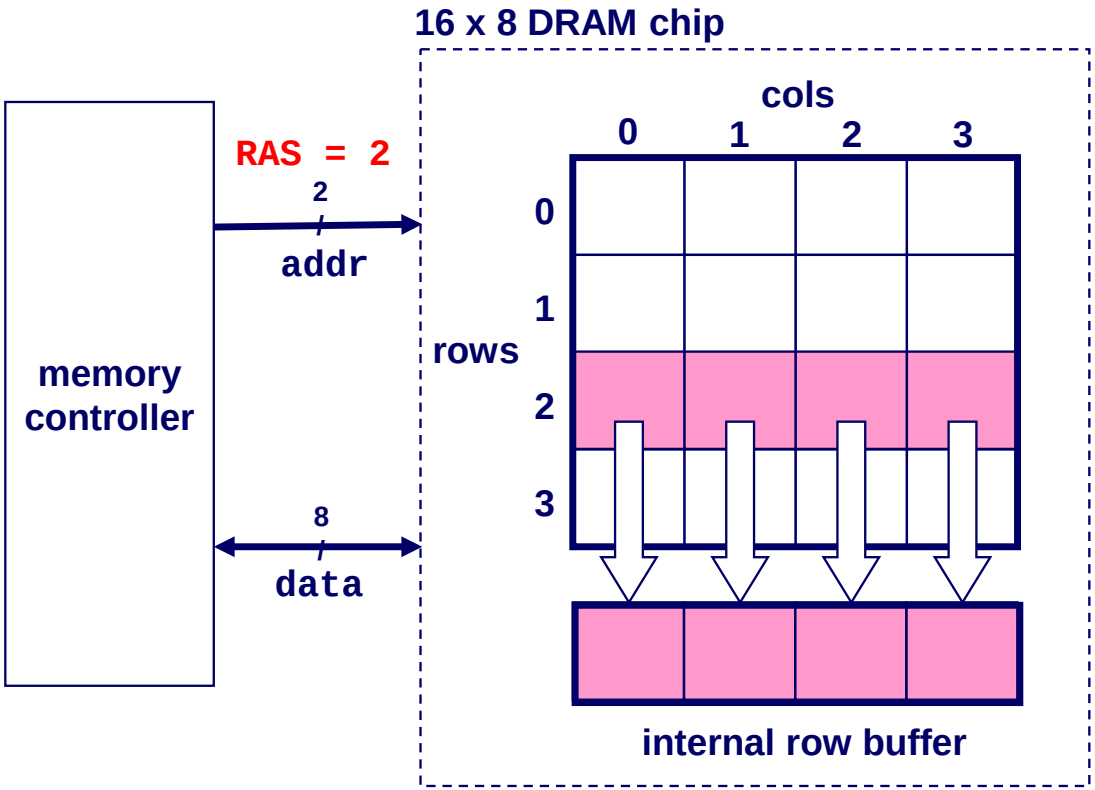
- (d × w) DRAM: organisiert als d-Superzellen mit w-bits



Beispiel: Lesen der DRAM Zelle (2,1)

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

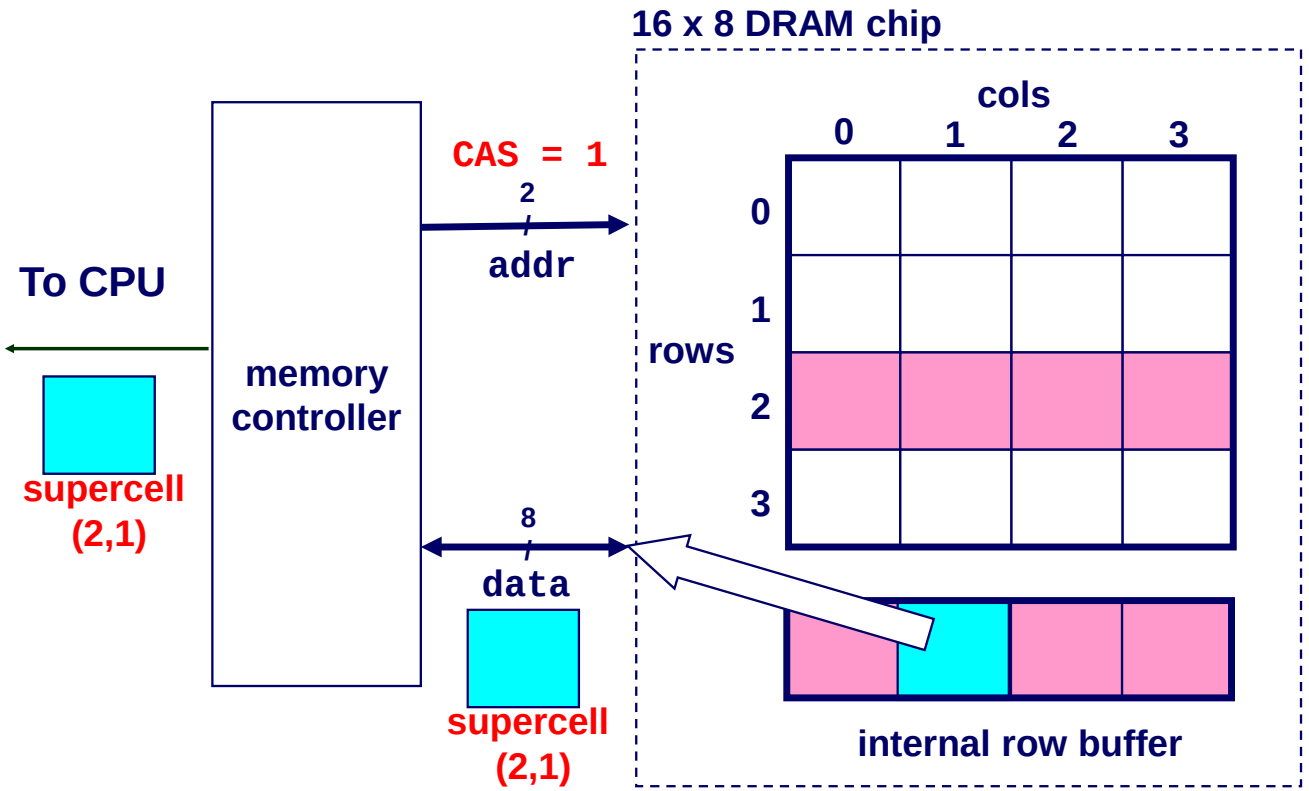
1.b Zeile aus DRAM Array in Zeilenpuffer („Row Buffer“) kopieren



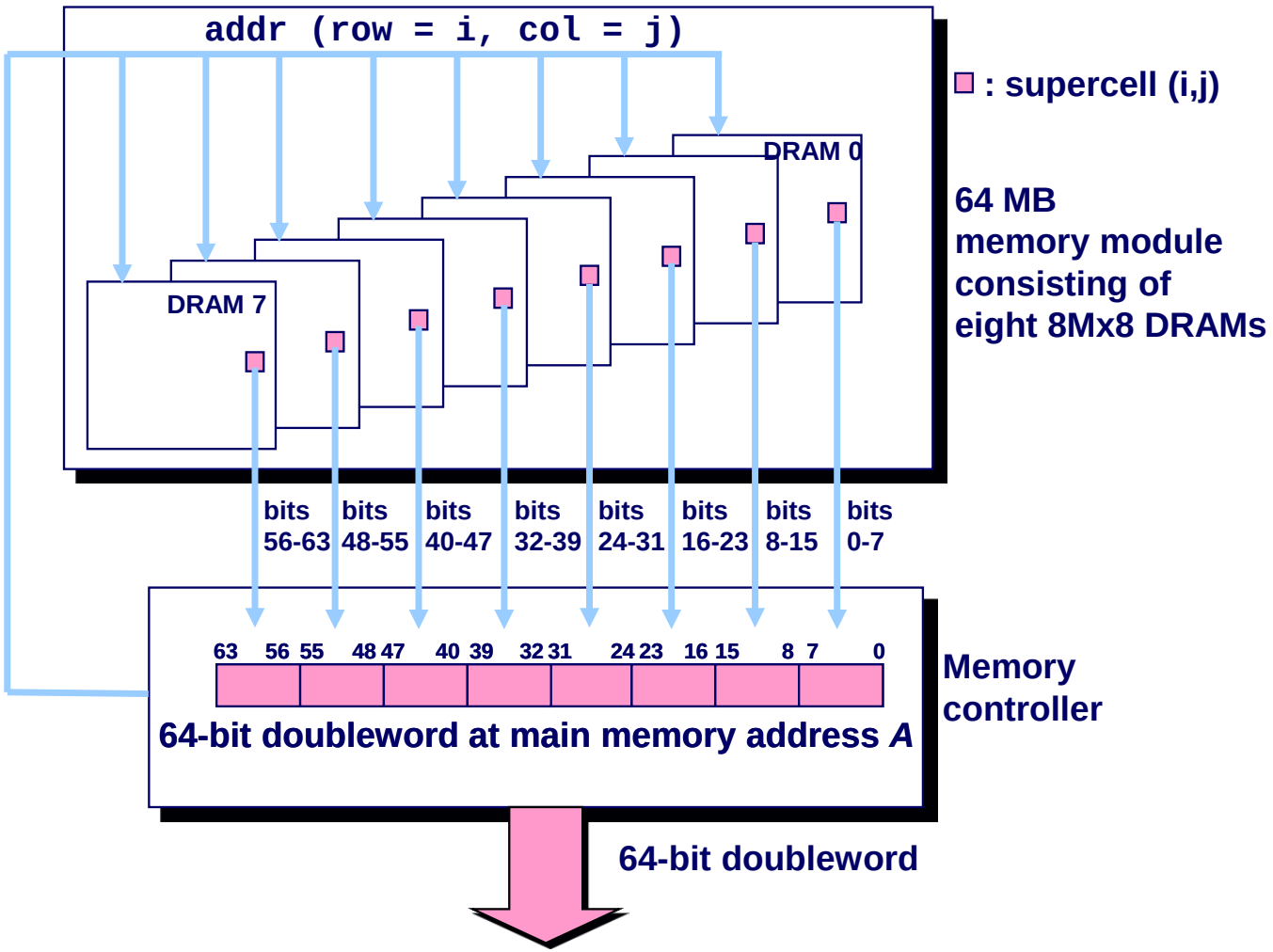
Beispiel: Lesen der DRAM Zelle (2,1)

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

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

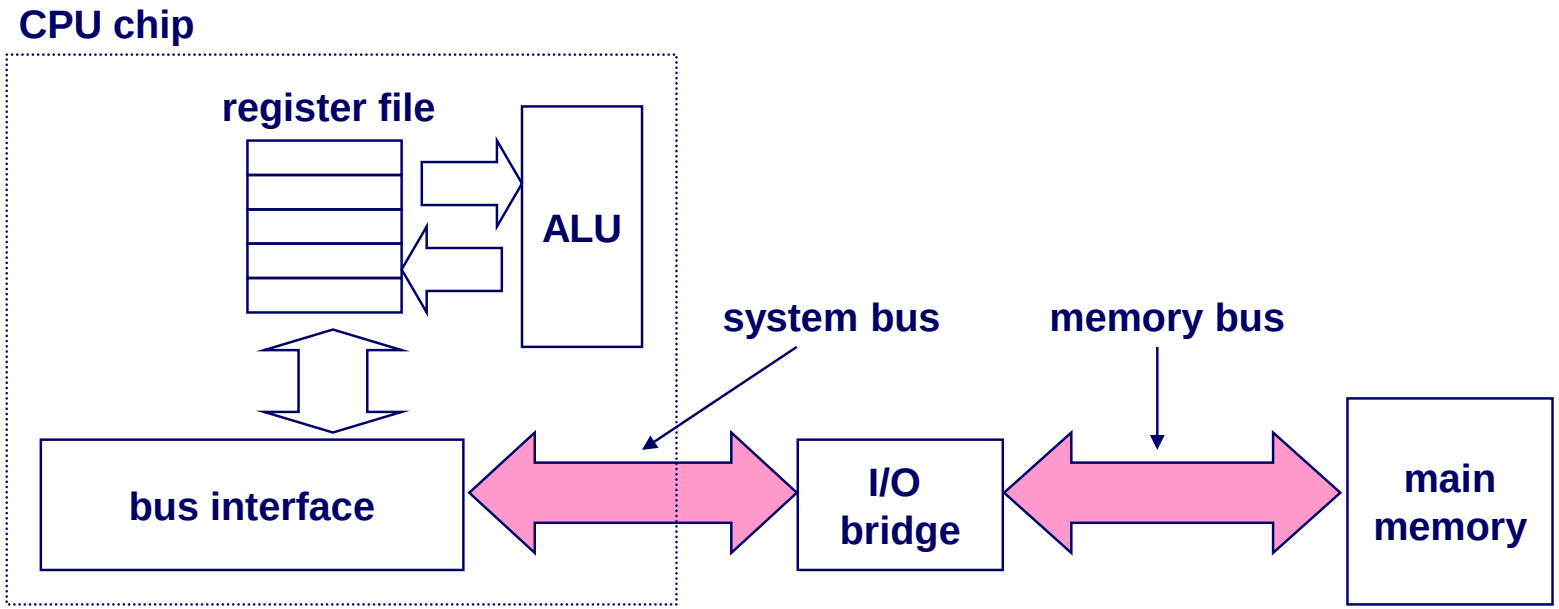


Speichermodule



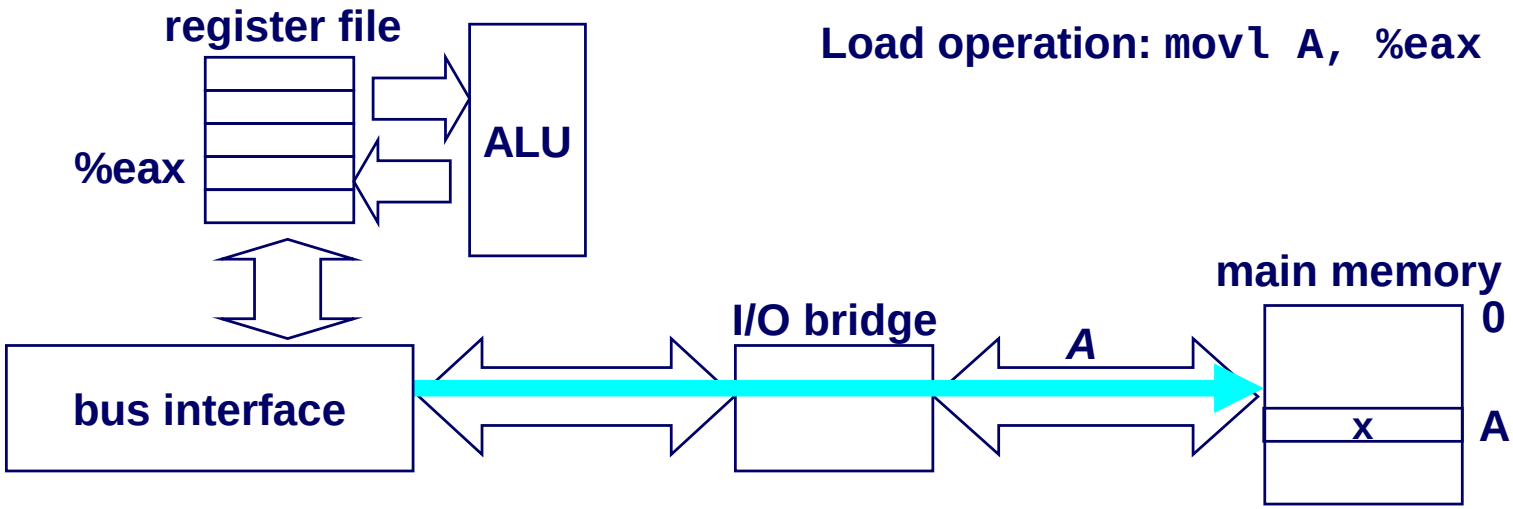
Bussysteme verbinden CPU und Speicher

- Systembus im Rechner
  - zur Übertragung von Adressen, Daten und Kontrollsignalen
  - zum Anschluß von mehreren Geräten genutzt
  - ggf. getrennte Busse für Hauptspeicher und Peripherie



Lesender Speicherzugriff (1)

1. CPU legt Adresse A auf den Speicherbus

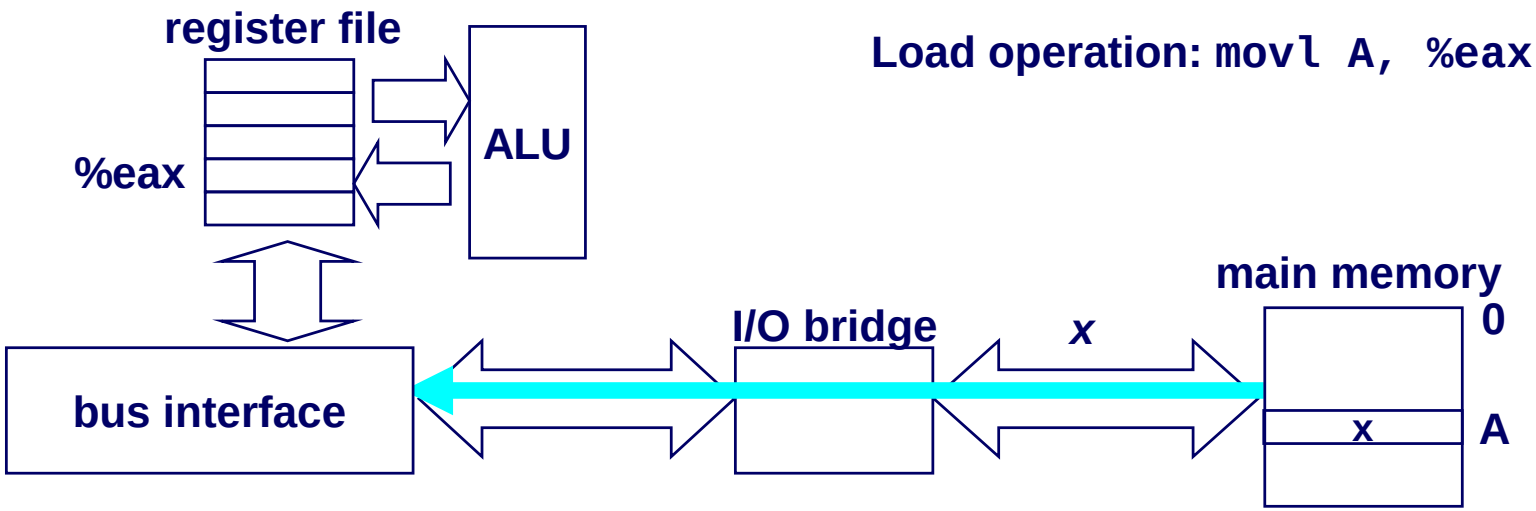


Lesender Speicherzugriff (2)

2.a Hauptspeicher liest Adresse A vom Speicherbus

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

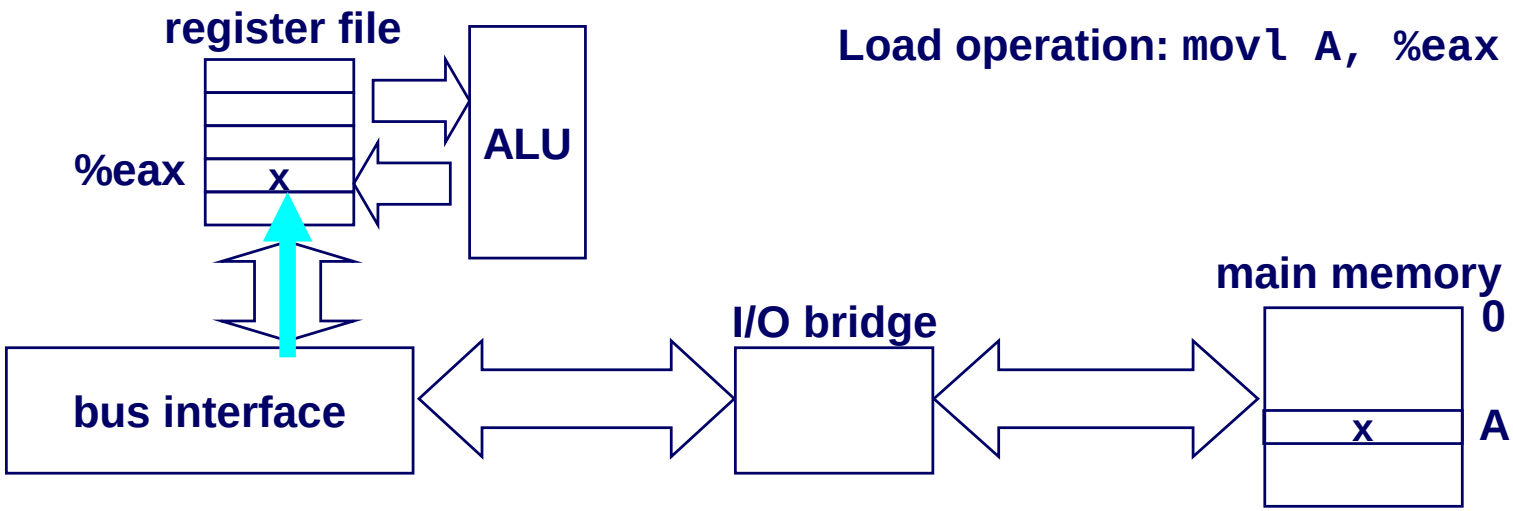
2.c Hauptspeicher legt das Wort x auf den Bus



Lesender Speicherzugriff (3)

3.a CPU liest Wort x vom Bus

3.b CPU kopiert Wert x in Register %eax

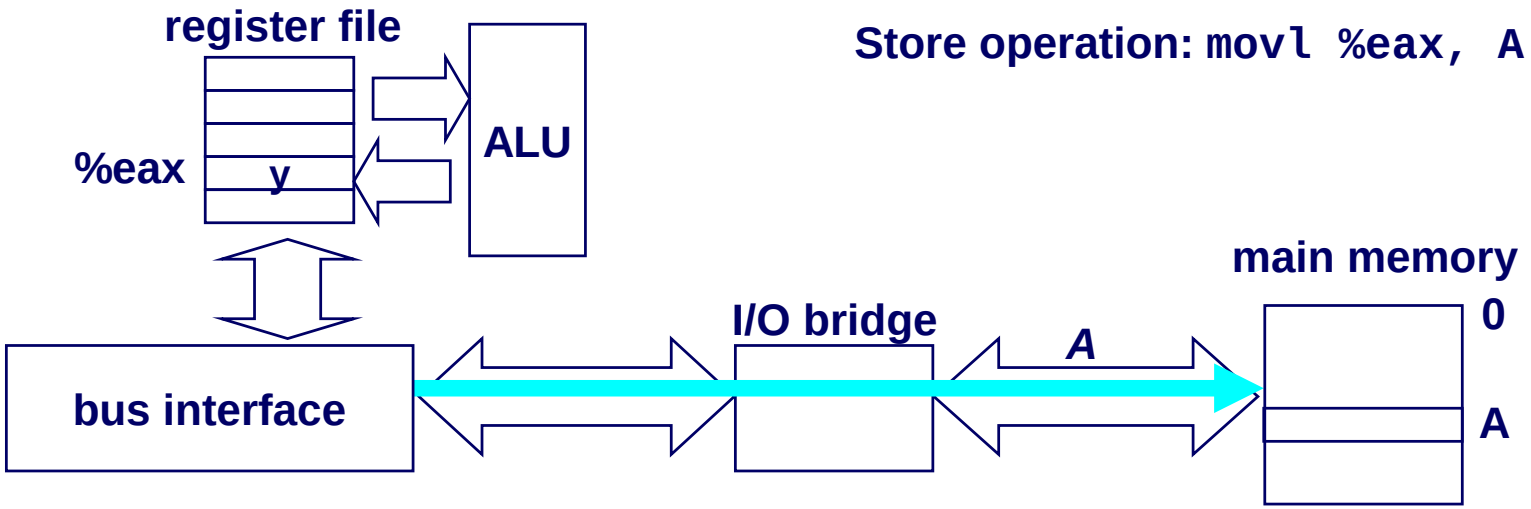


Schreibender Speicherzugriff (1+2)

1 CPU legt die Adresse A auf den Bus

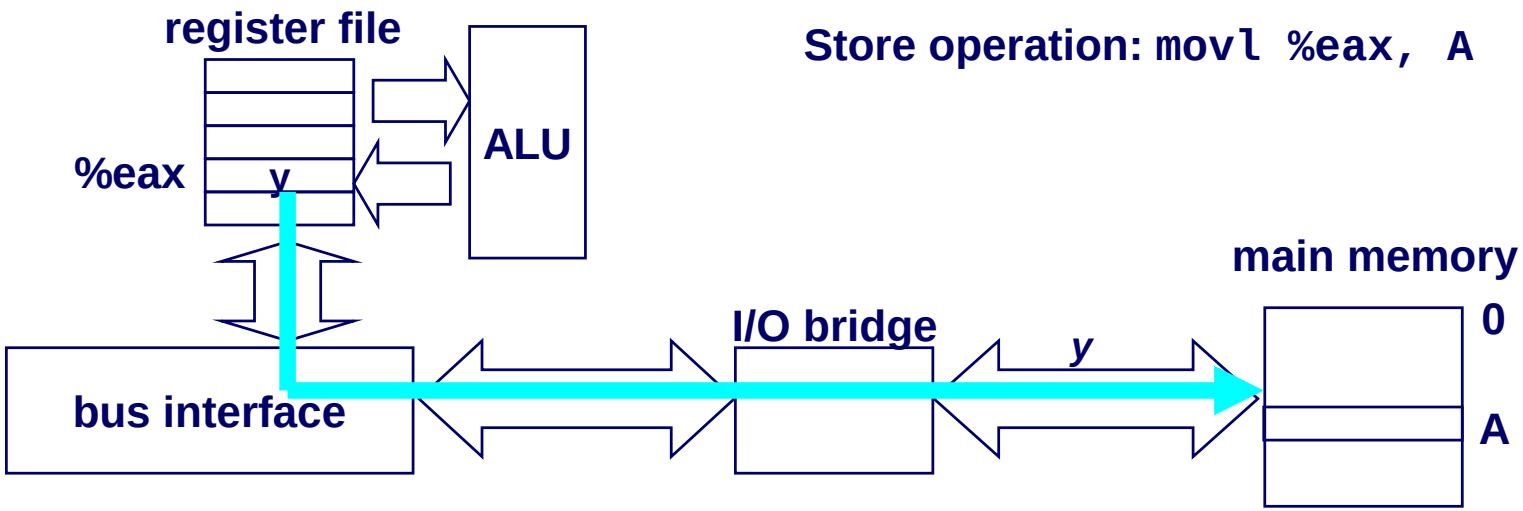
2.b Hauptspeicher liest Adresse

2.c Hauptspeicher wartet auf Ankunft des Datenworts



Schreibender Speicherzugriff (3)

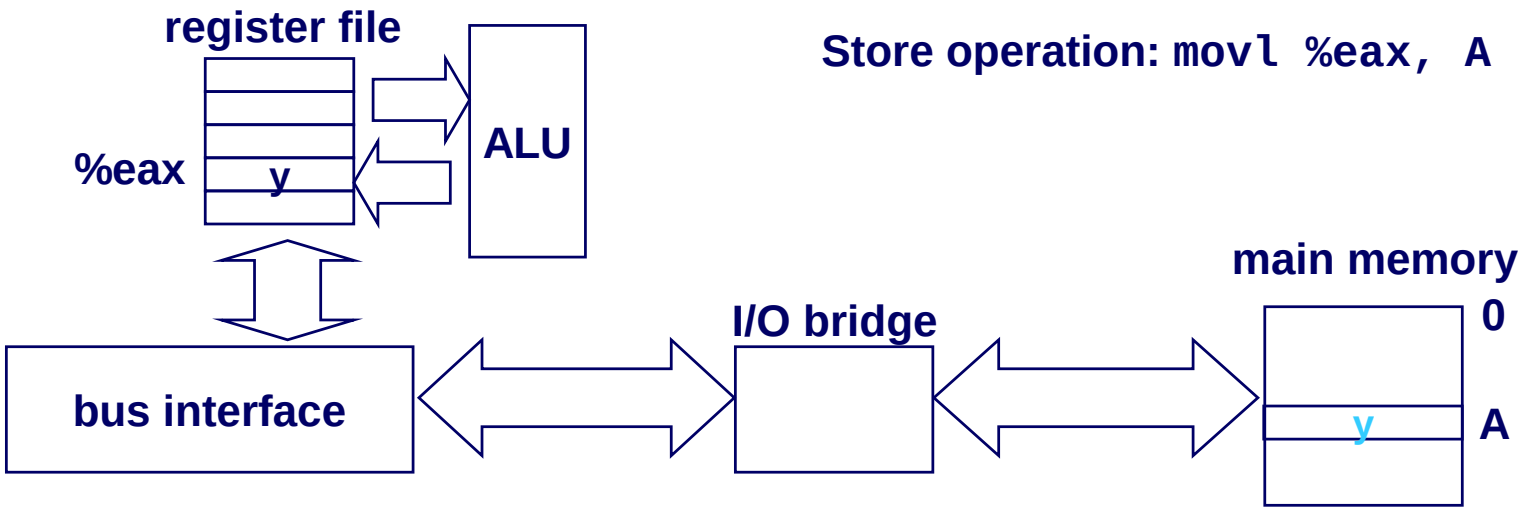
3. CPU legt Datenwort y auf den BusA



Schreibender Speicherzugriff (4)

4.a Hauptspeicher liest Datenwort y vom Bus

4.b Hauptspeicher speichert Datenwort y unter Adresse A



DMA: Direct Memory Access

beim Transfer größerer Datenmengen:

- separaten DMA-Controller initialisieren und verwenden
- autonomer Transfer ohne Beteiligung der CPU
- CPU via Interrupt benachrichtigt, sobald Transfer komplett

- Prozess läuft ohne Beteiligung des Prozessors
- Rechner kann währenddessen andere Programme bearbeiten

Cache

- Prozessor läuft mit ca. 1 GHz Takt
  - Register können mithalten, aber nur einige kByte Kapazität
  - DRAM braucht 60..100 ns für einen Zugriff: 100× langsamer
  - Festplatte braucht 10 ms für einen Zugriff: 1 000 000× langsamer
  - aufeinanderfolgende Speicherzugriffe sind meistens „lokal“

Themen

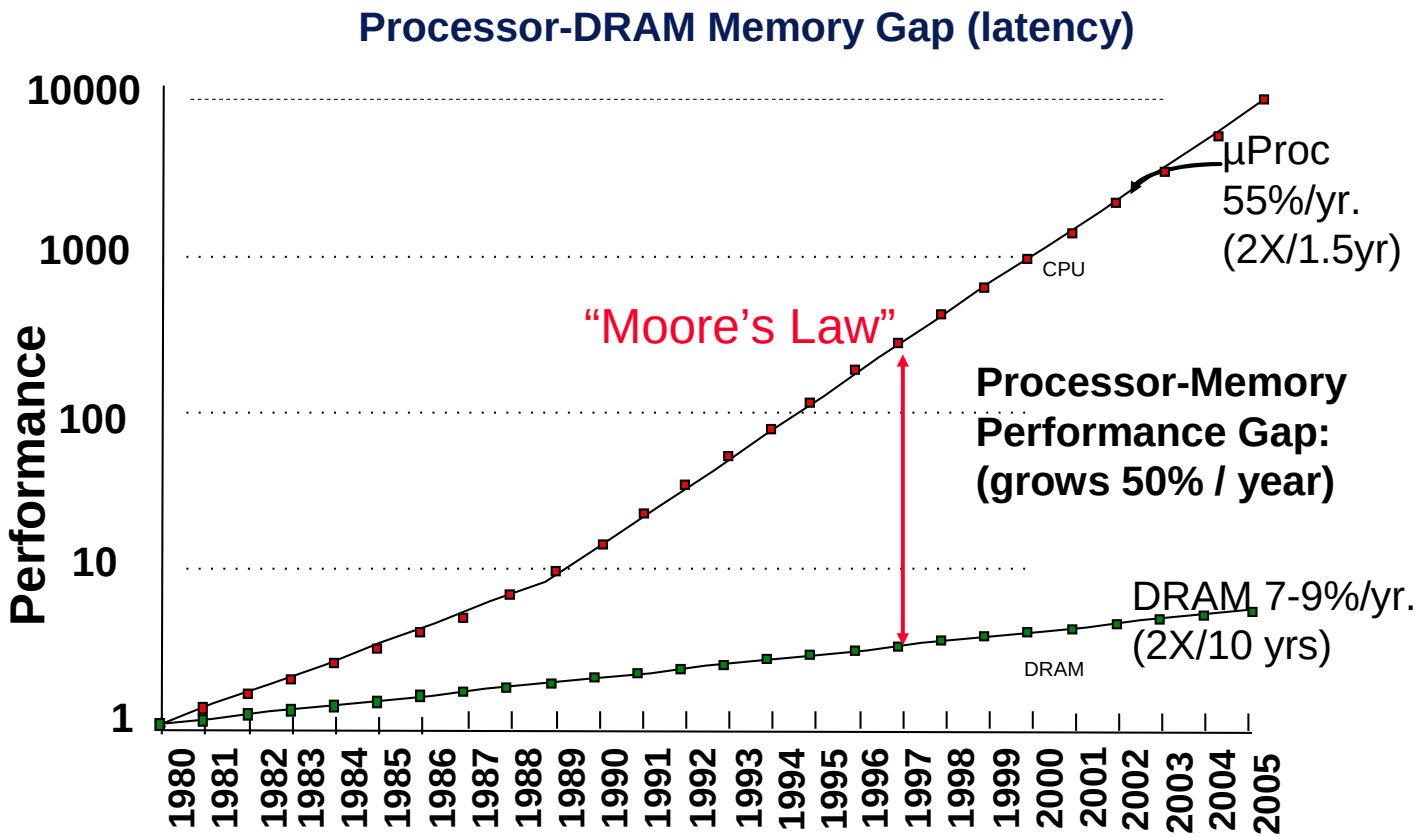
- Prinzip und Organisation von Cache-Speicher
- direkt abgebildete Caches („direct mapped“)
- bereichsassoziative Caches („set associative“)
- Einfluss der Caches auf die Leistung

Verwaltung der Speicherhierarchie

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

Speicherhierarchie: Memory Wall

- „Memory Wall“: DRAM zu langsam für CPU



Cache: schneller Zwischenspeicher zum Hauptspeicher

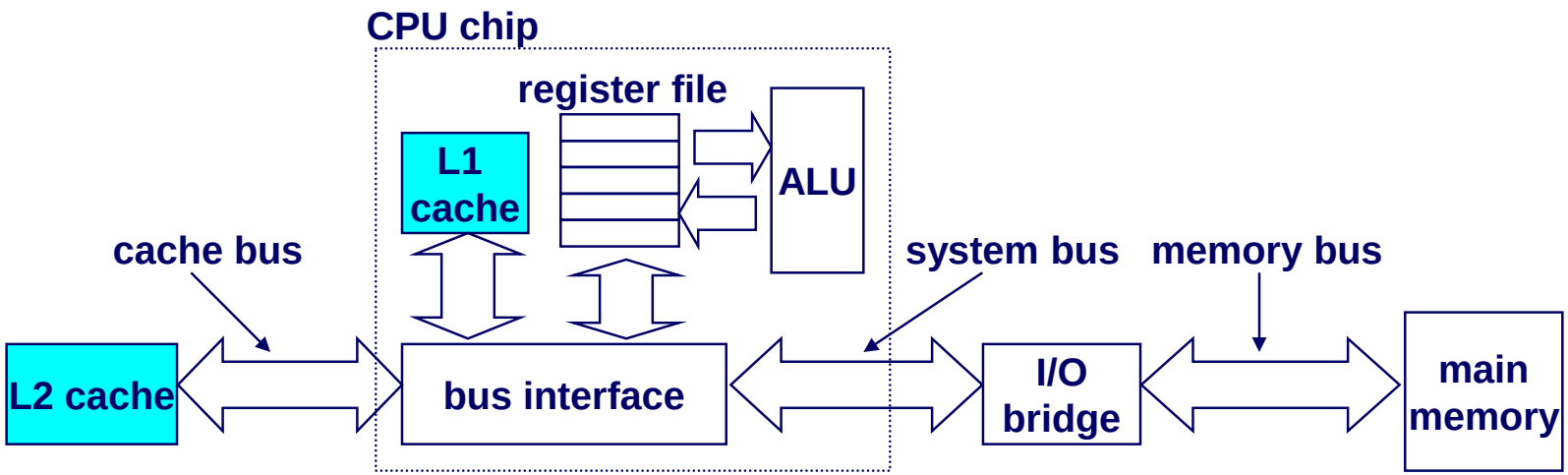
- technische Realisierung: SRAM
- transparenter Speicher
  - Cache ist für den Programmierer nicht sichtbar!
  - wird durch Hardware verwaltet
- `http://de.wikipedia.org/wiki/Cache`

- enthält Hauptspeicherblöcke mit erhöhter Zugriffswahrscheinlichkeit
- ggf. getrennte Caches für Befehle und Daten
- basiert auf Prinzip der Lokalität
  - 80% der Zugriffe greifen auf 20% der Adressen zu
  - manchmal auch 90% / 10% oder noch besser

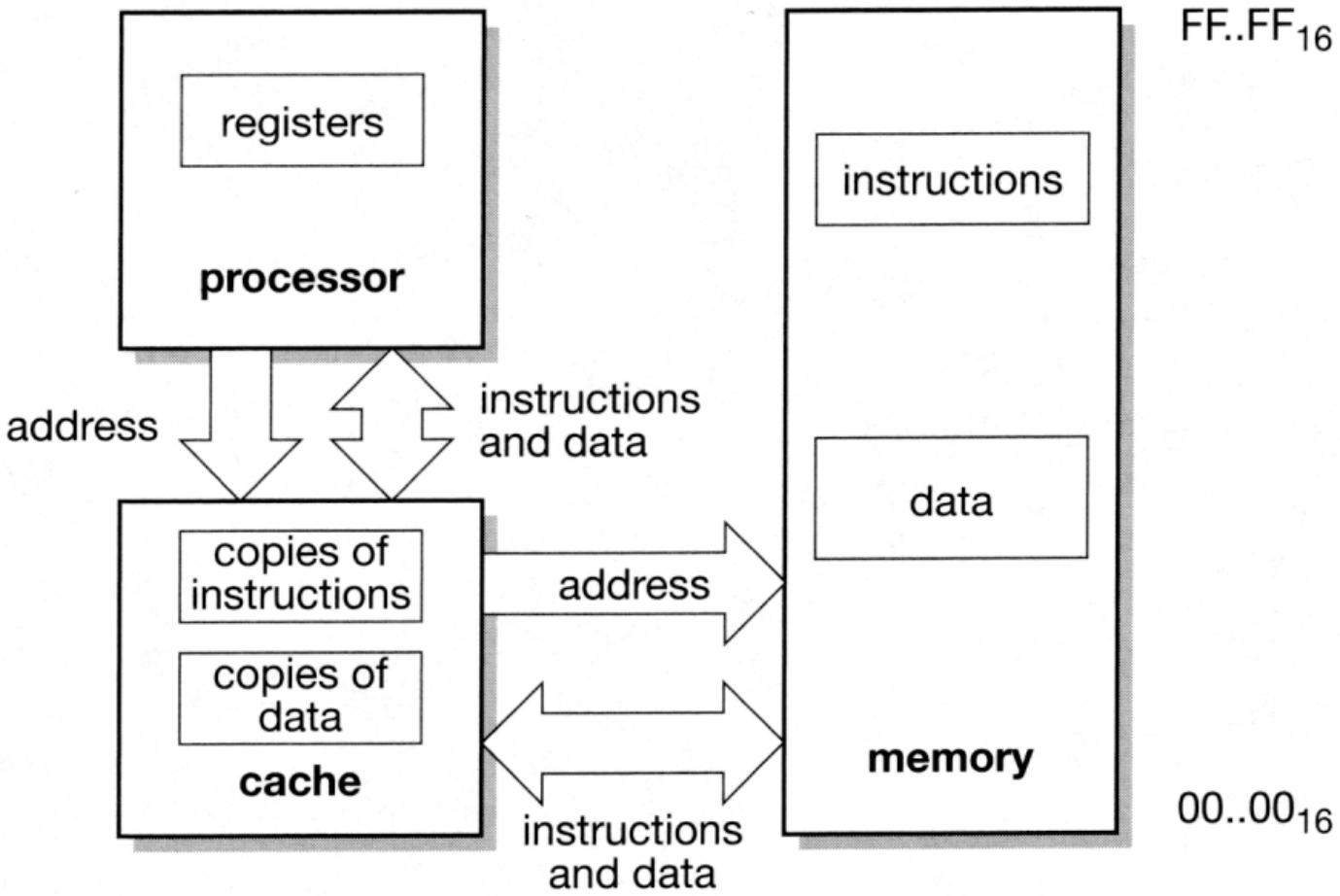
Cache: Funktionsprinzip

CPU referenziert Adresse

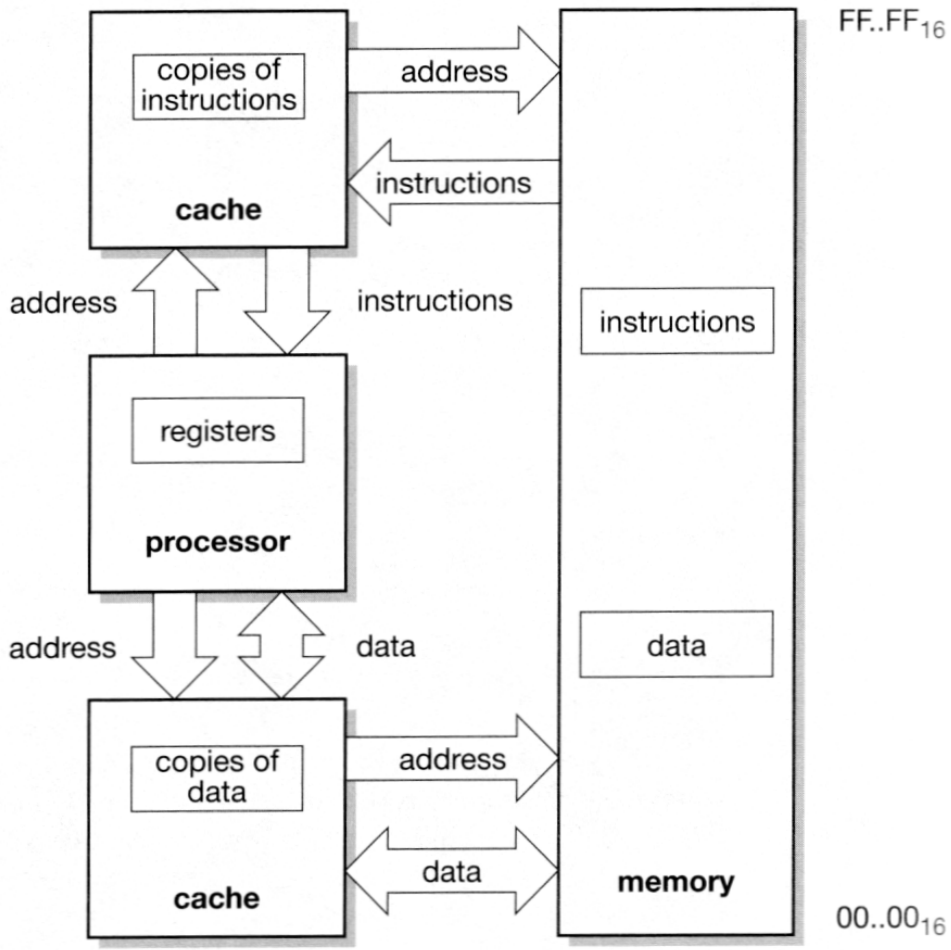
- parallele Suche in L1 (level 1), L2... und Hauptspeicher
- erfolgreiche Suche liefert Datum, Abbruch laufender Suchen



Unified-Cache: gemeinsam für Befehle und Daten



Separate Instruction-/Data-Caches



Cache: Strategie

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: Performance

- Begriffe

Treffer (Hit)		Zugriff auf Datum, ist bereits im Cache
Fehler (Miss)		Zugriff auf Datum, 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

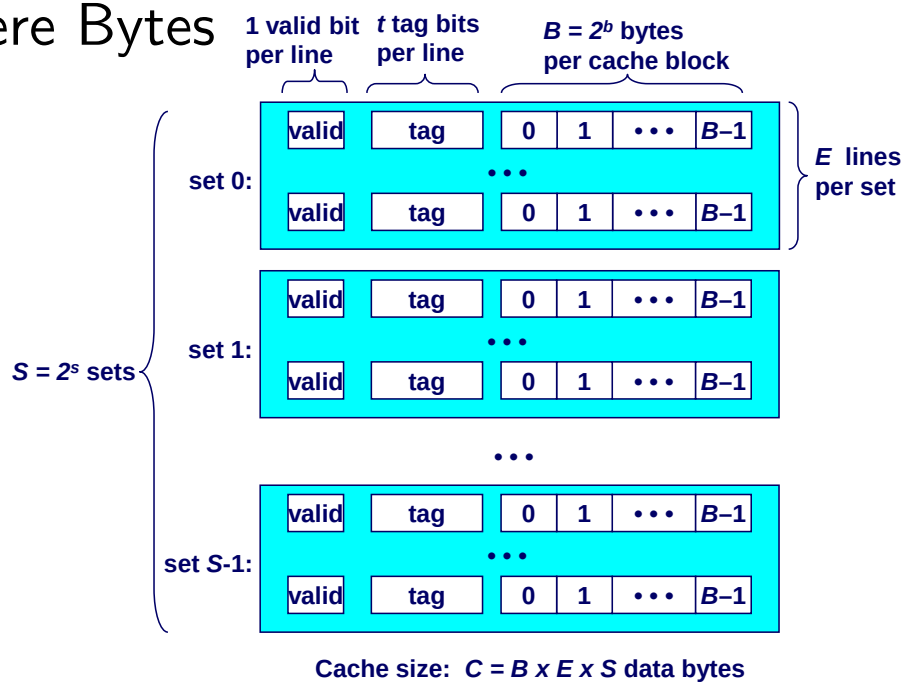
- Mittlere Speicherzugriffszeit $= T_{Hit} + R_{Miss} \cdot T_{Miss}$
- Beispiel

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

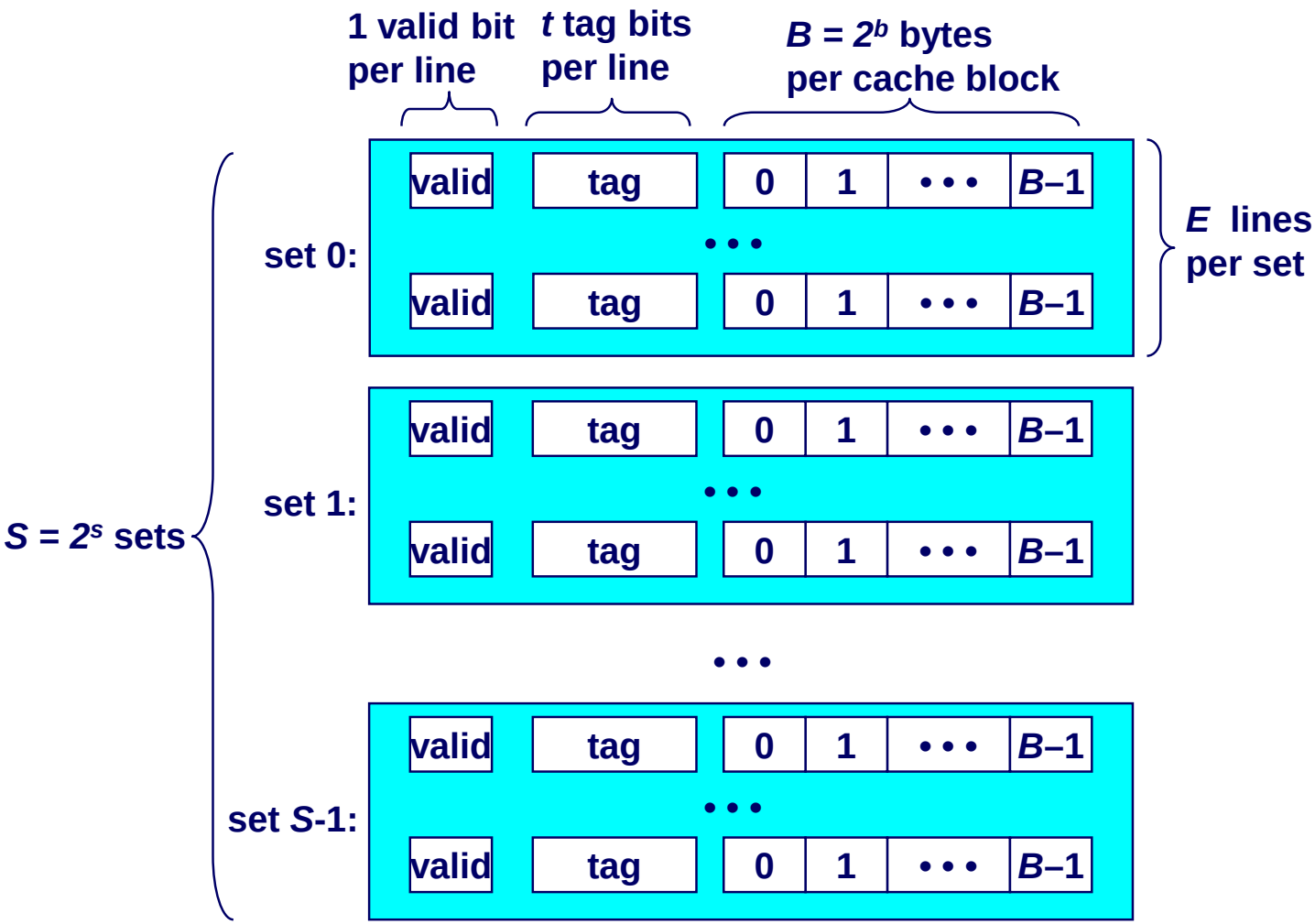
  Mittlere Speicherzugriffszeit $= 2\ Takte$

Cache: Organisation

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

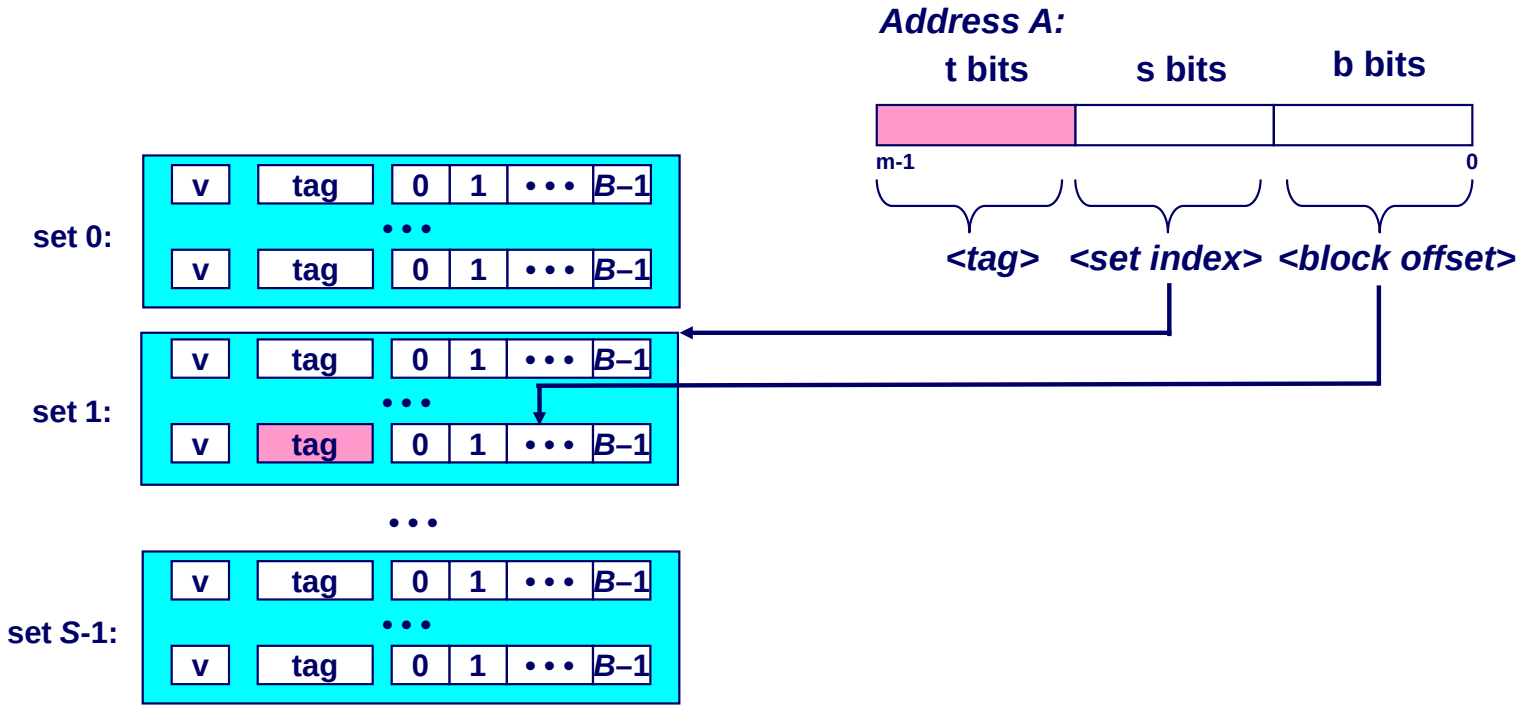


Cache: Tags und Daten



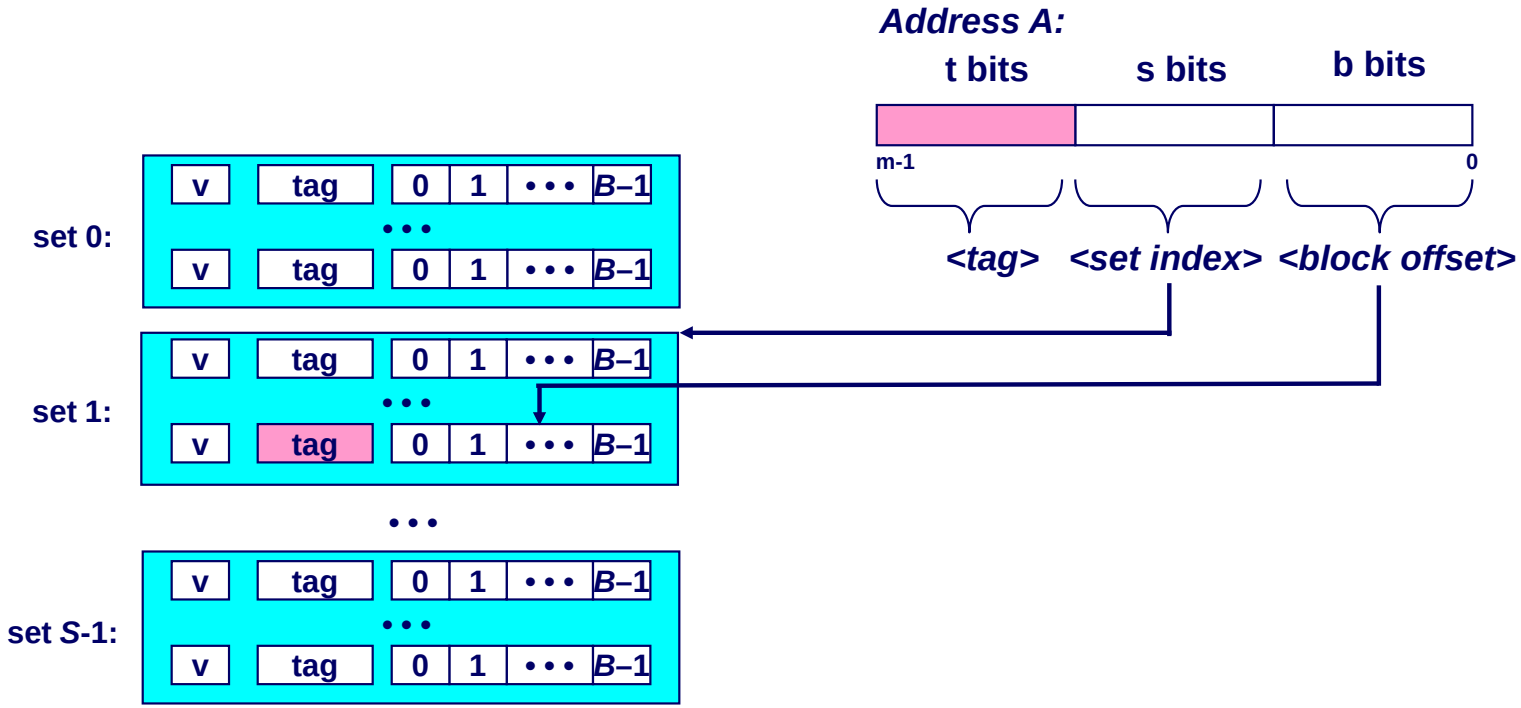
Cache: Adressierung und Zugriff

- 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



Cache: Zugriff

- 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 Möglichkeiten:

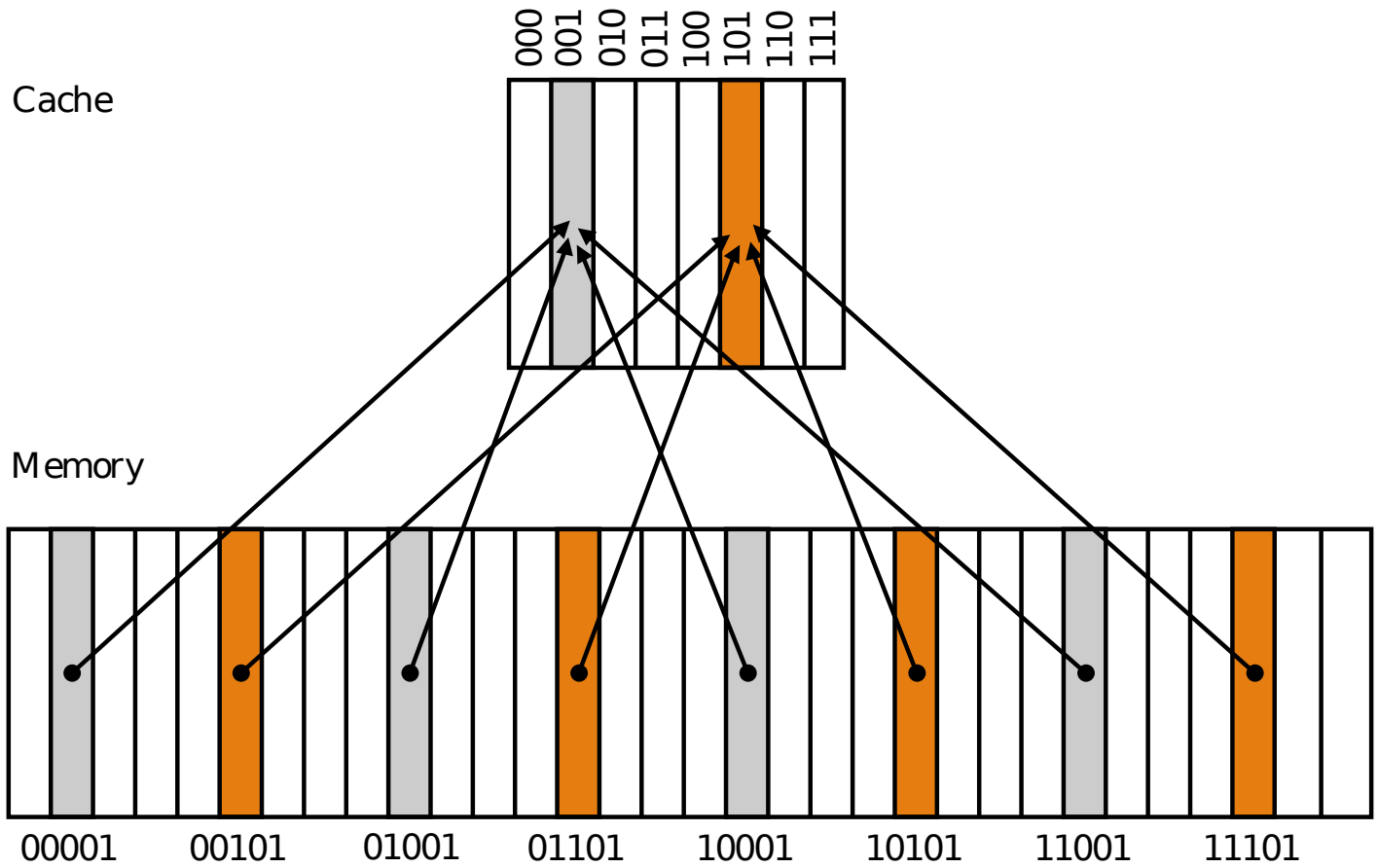
direkt abgebildet / direct mapped jeder Speicheradresse ist genau eine Cache-Speicherzelle zugeordnet

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

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

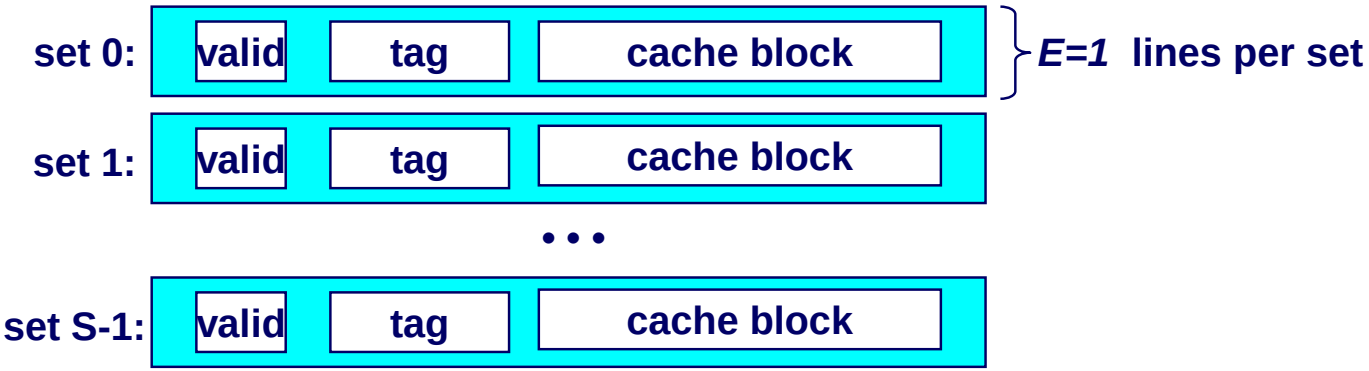
Cache: direkt abgebildet / „direct mapped“

- jeder Adresse ist genau eine Speicherzelle im Cache zugeordnet



Cache: direkt abgebildet / „direct mapped“

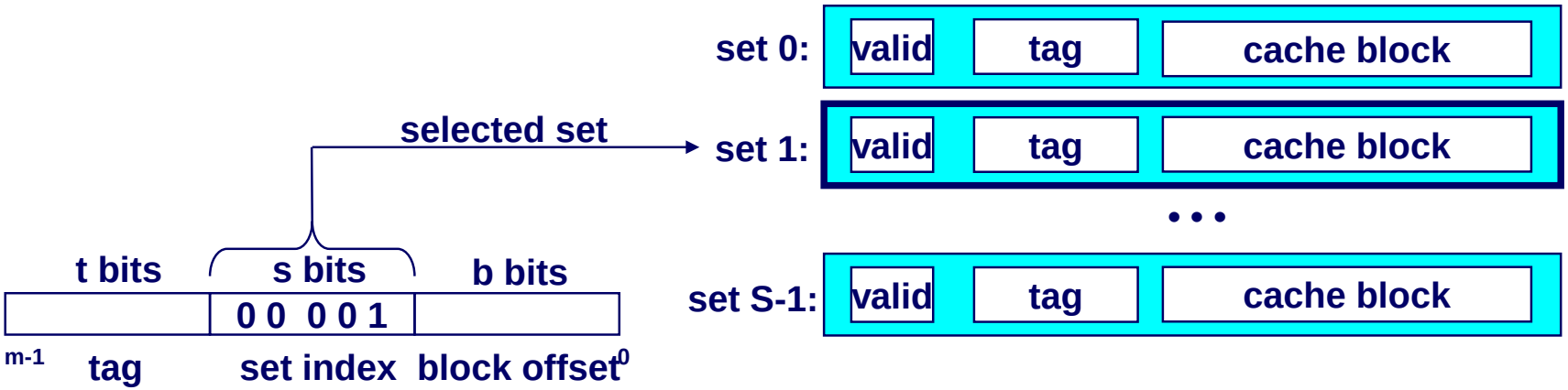
- eindeutige Zuordnung von Zeilen zu Adressen
- Wert im „tag“ entspricht genau einer Adresse



- die einfachste Cache-Art
- große Caches realisierbar
- „Thrashing“ möglich, z.B. Zugriffe auf A, $A + n \cdot S \ldots$

Direct mapped cache: Zugriff

1 Bereichsauswahl durch Bits ⟨*set index*⟩

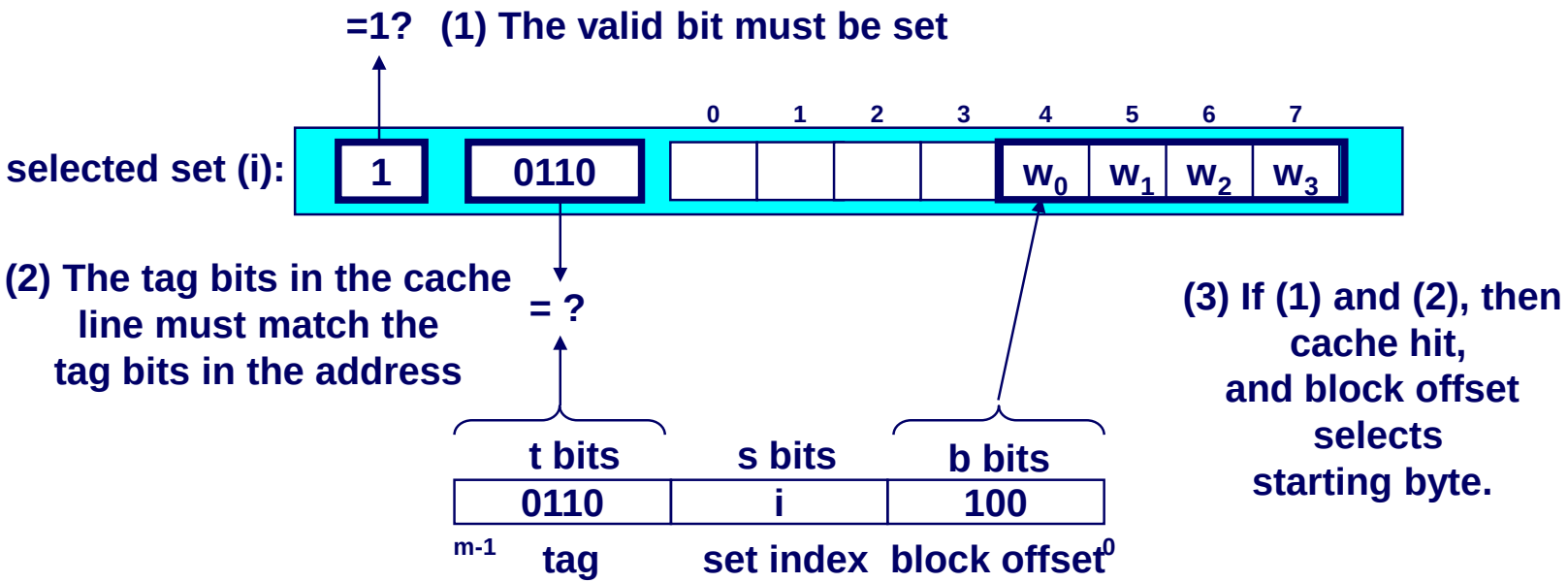


Direct mapped cache: Zugriff

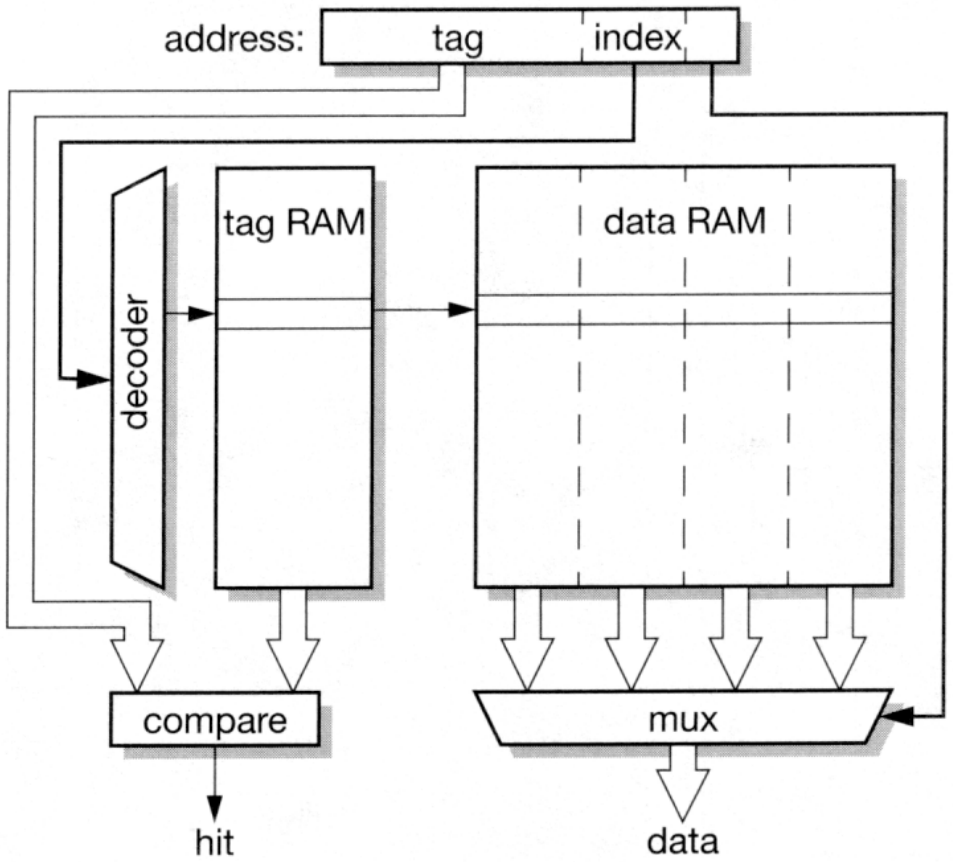
2 Test auf valid: sind die Daten gültig?

3 „Line matching“: stimmt $\langle tag \rangle$ überein?

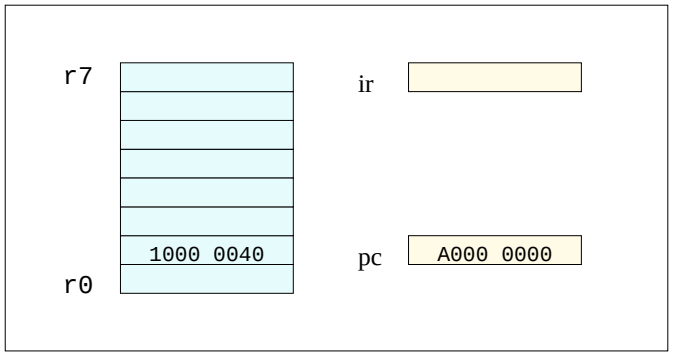
4 Wortselektion extrahiert Wort unter Offset $\langle block\ offset \rangle$

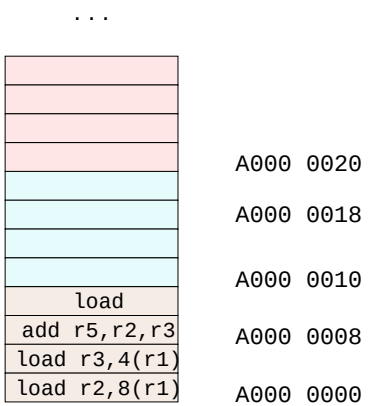


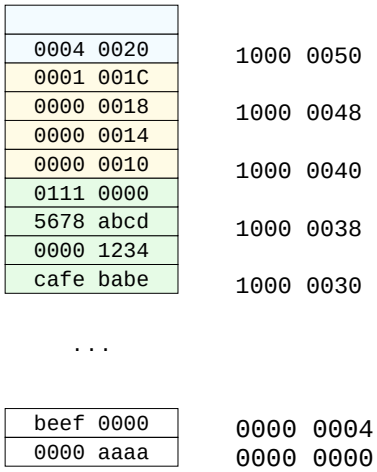
Direct mapped cache: Prinzip



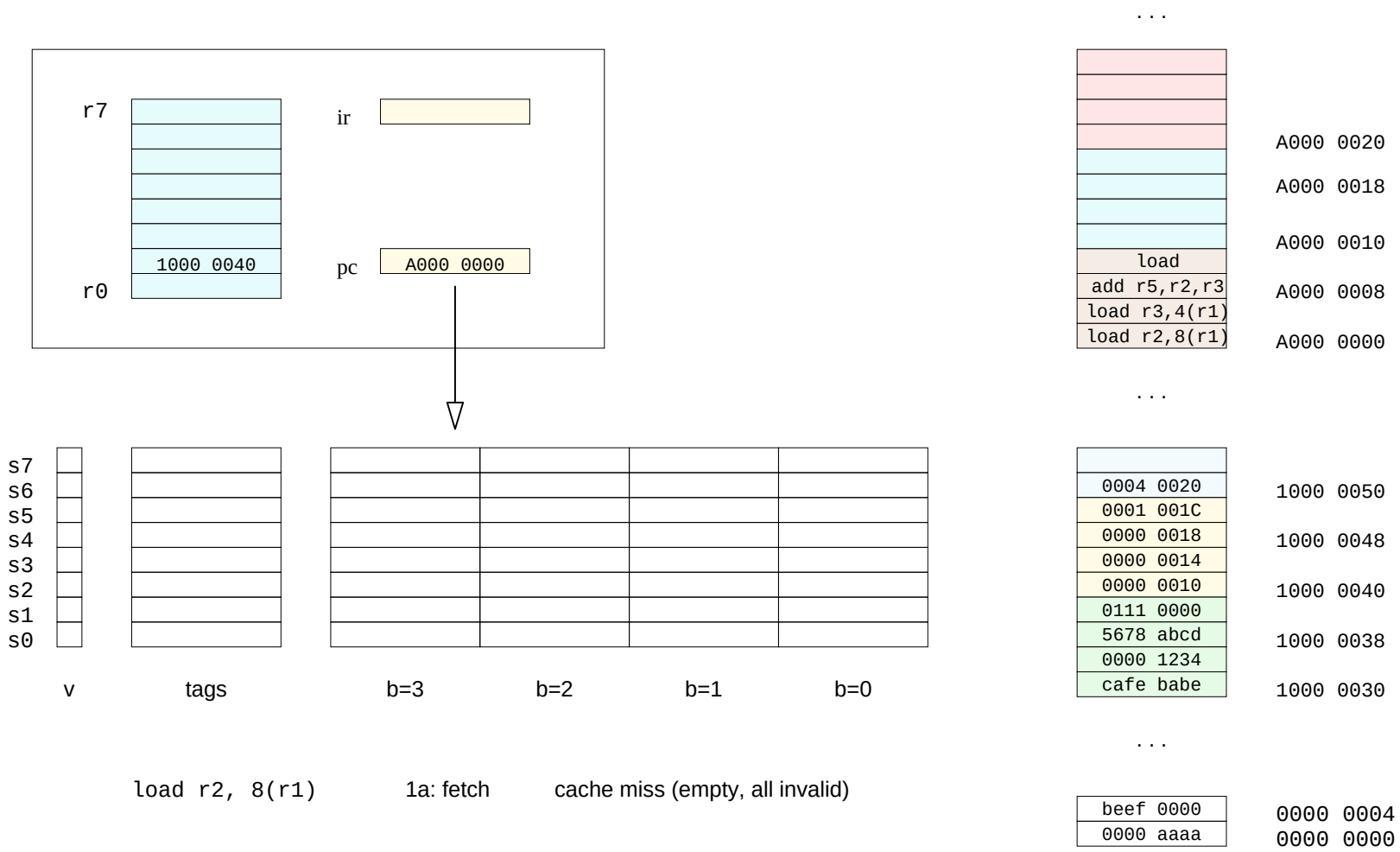
Direct mapped cache: Beispiel



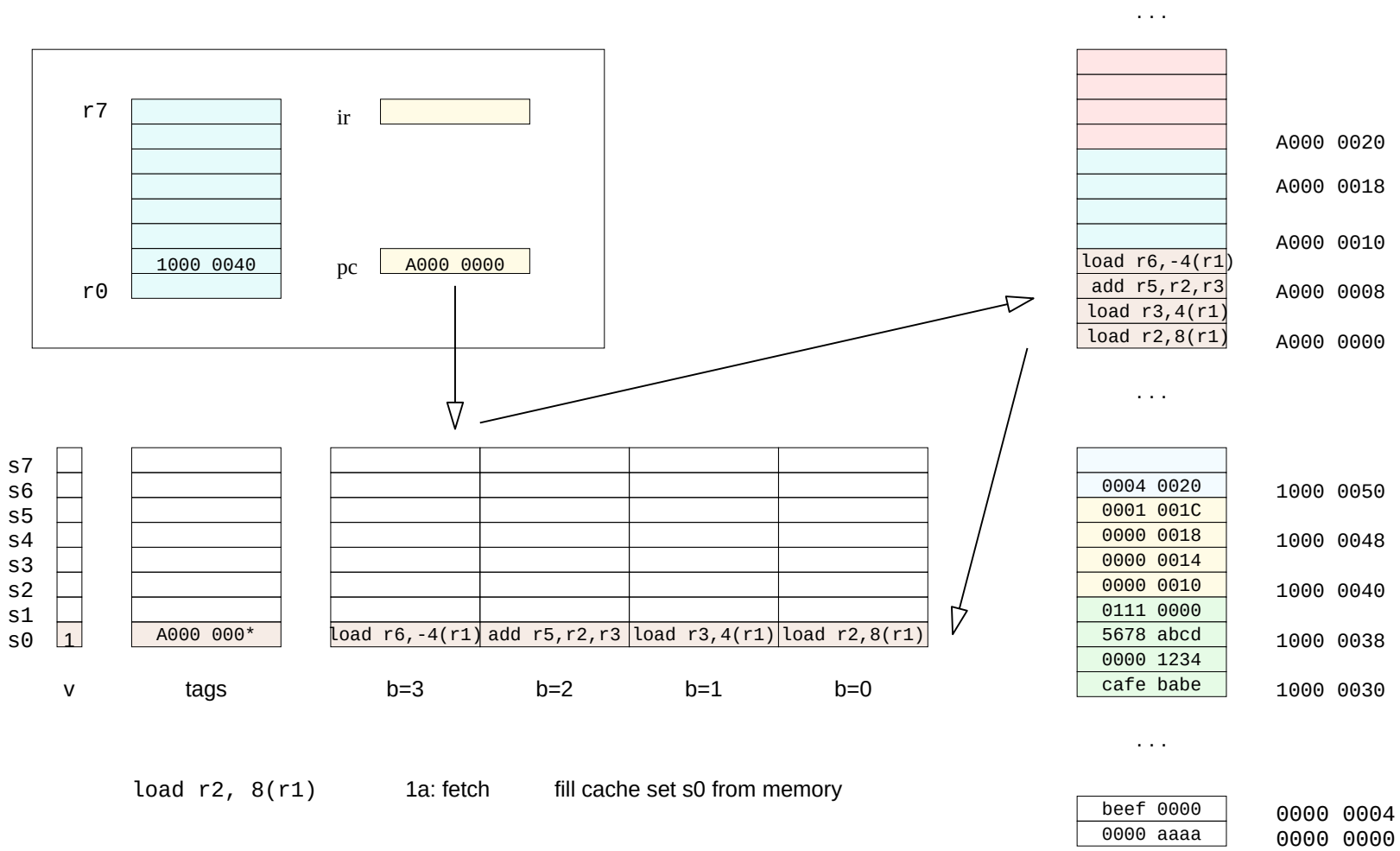




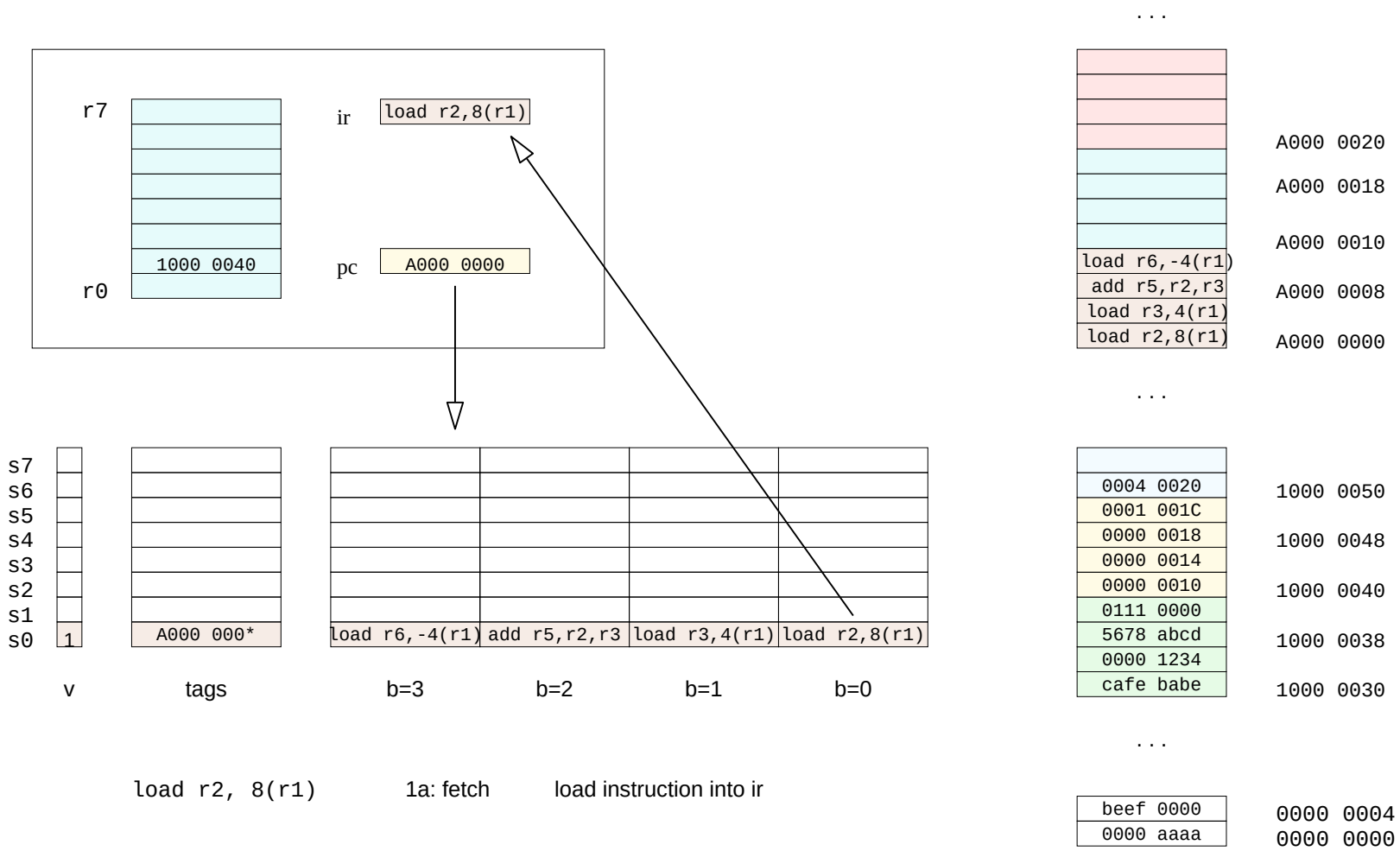
Direct mapped cache: Beispiel



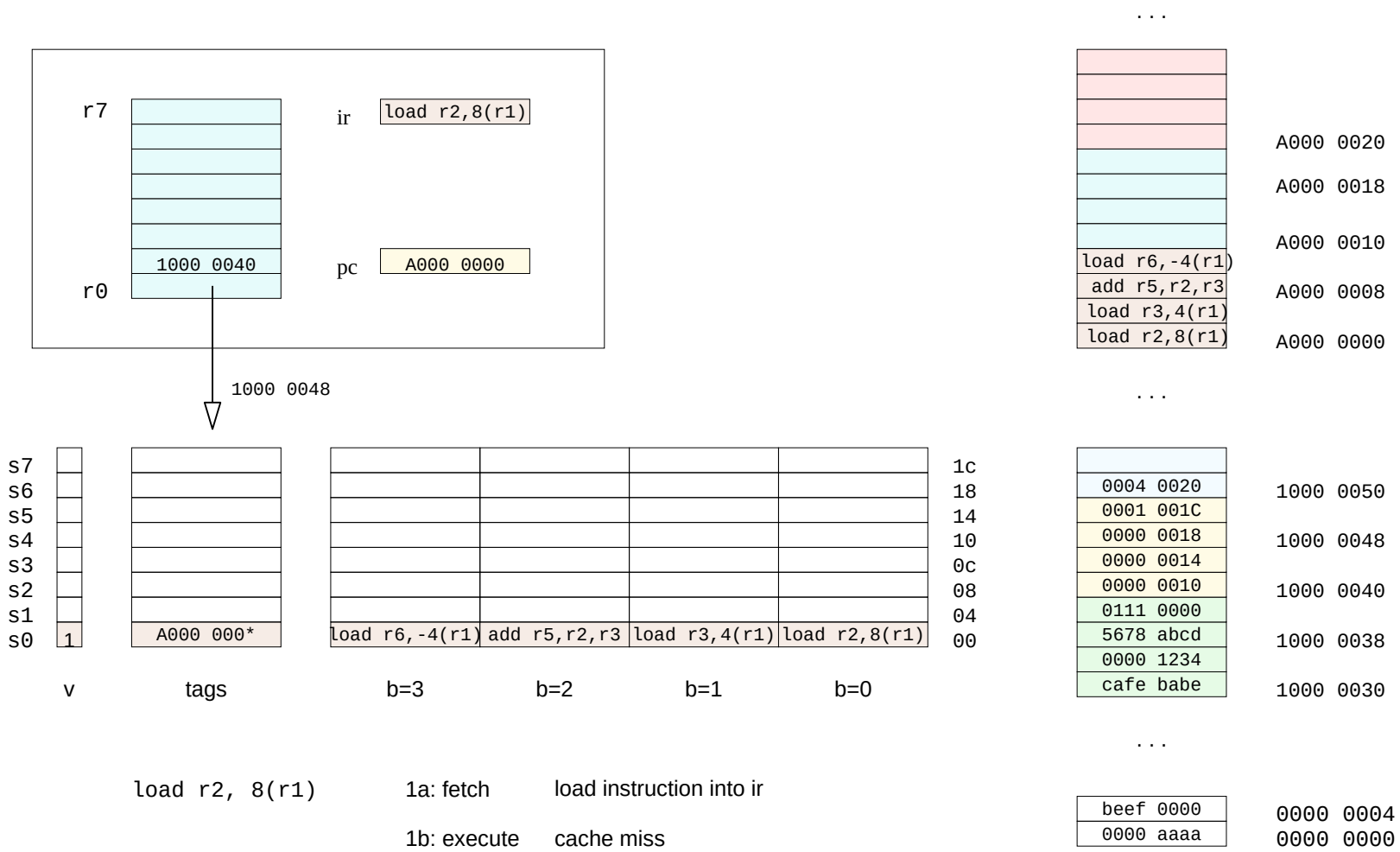
Direct mapped cache: Beispiel



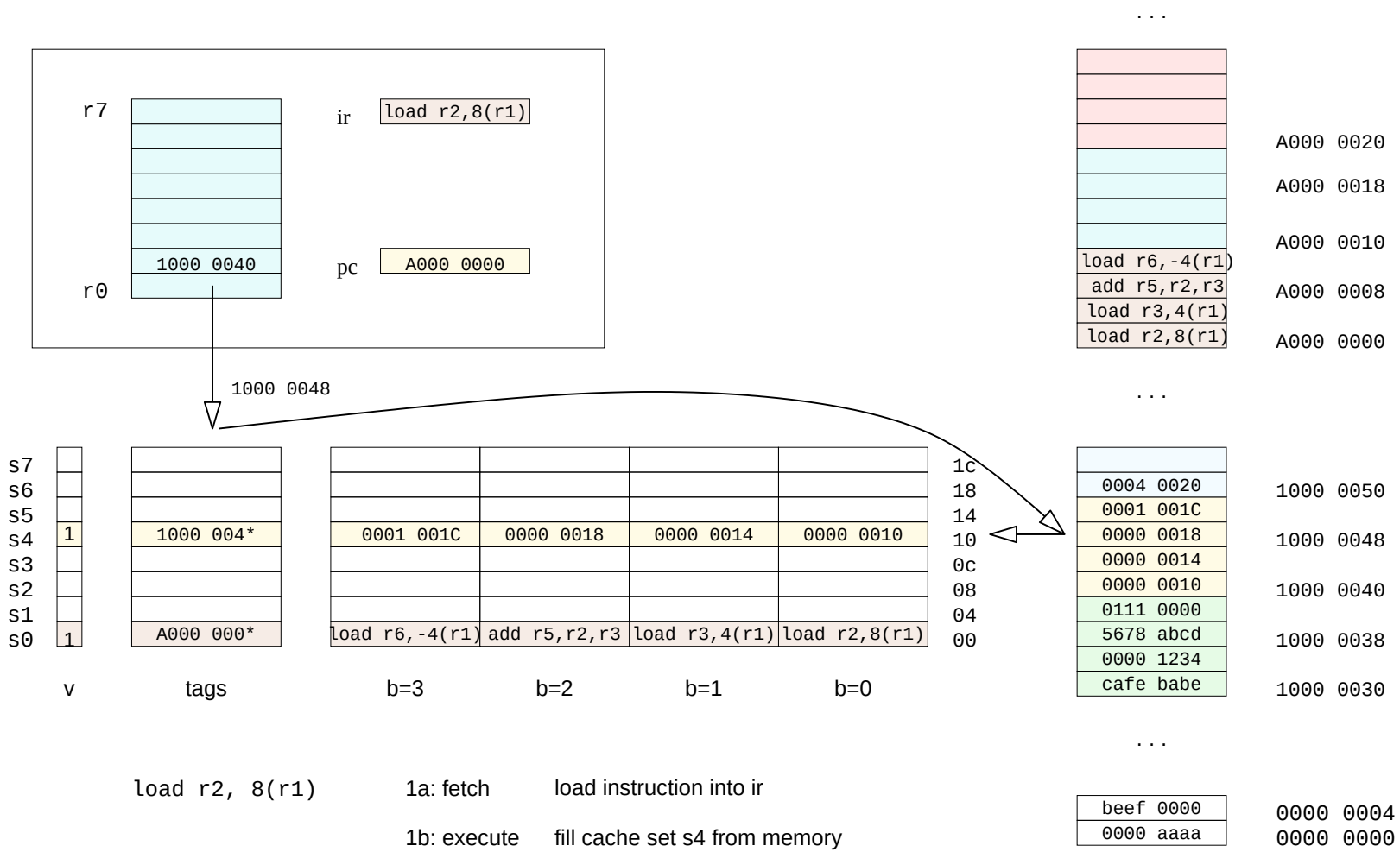
Direct mapped cache: Beispiel



Direct mapped cache: Beispiel



Direct mapped cache: Beispiel

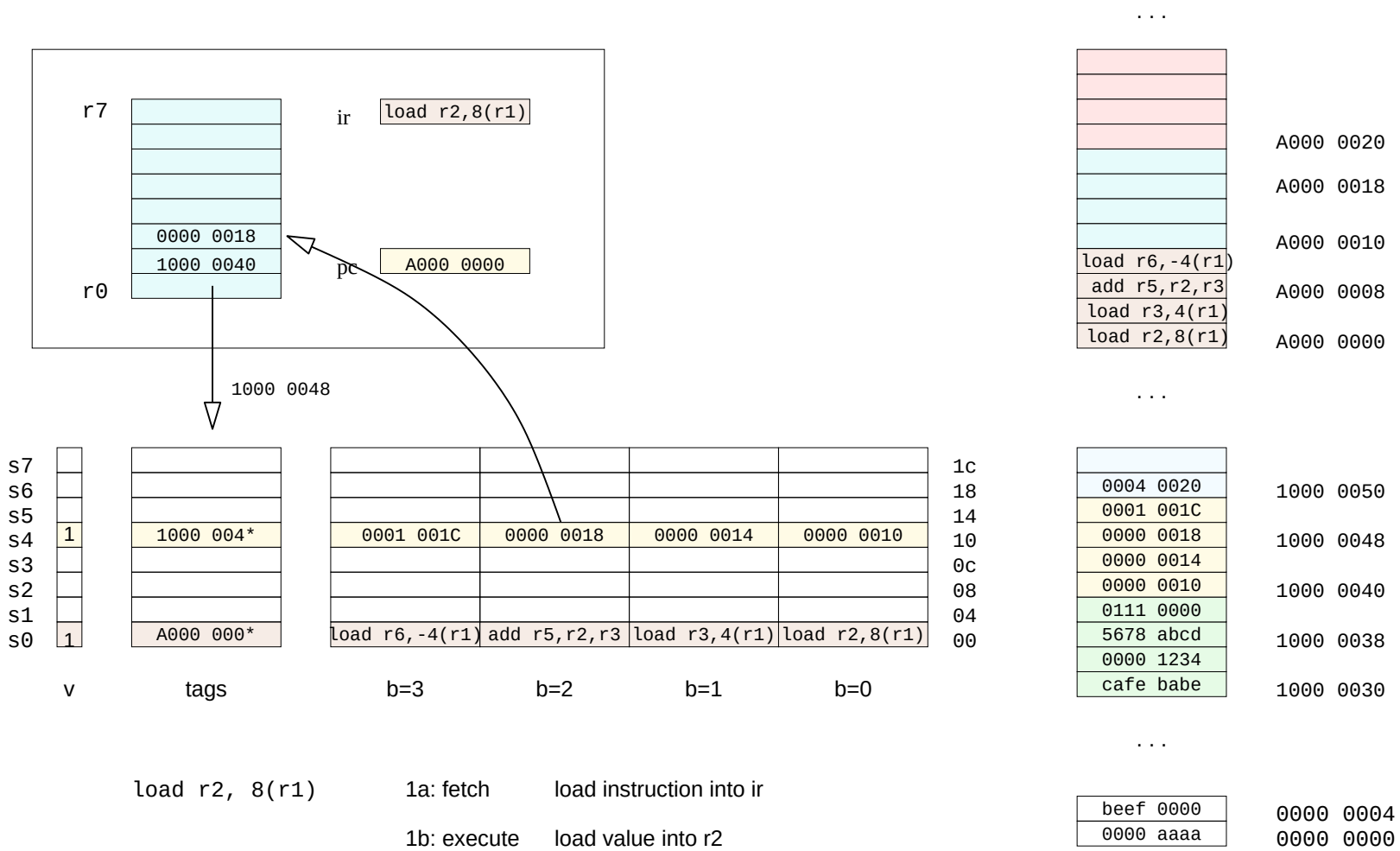


`load r2, 8(r1)`

1a: fetch — load instruction into ir

1b: execute — fill cache set s4 from memory

Direct mapped cache: Beispiel

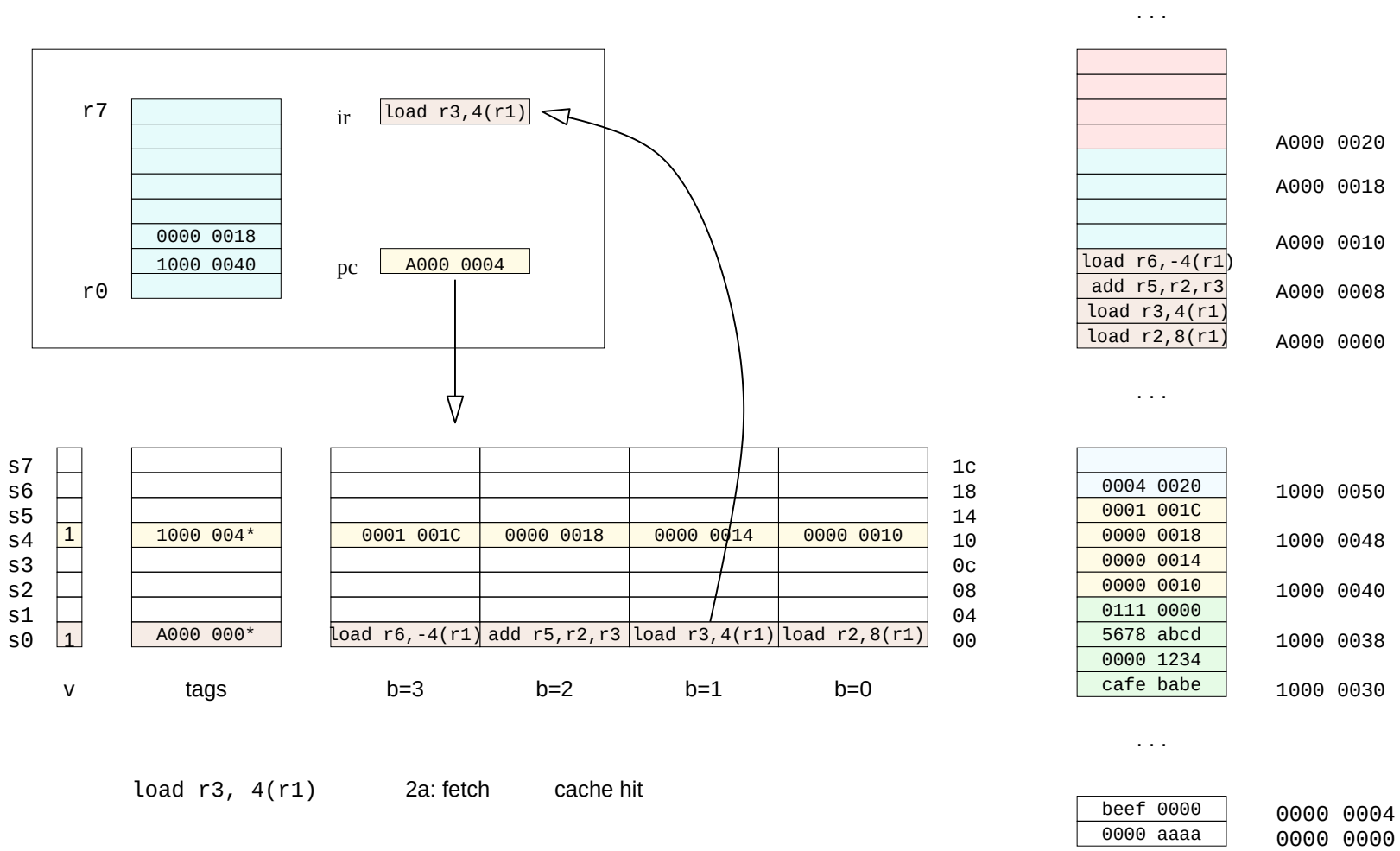


`load r2, 8(r1)`

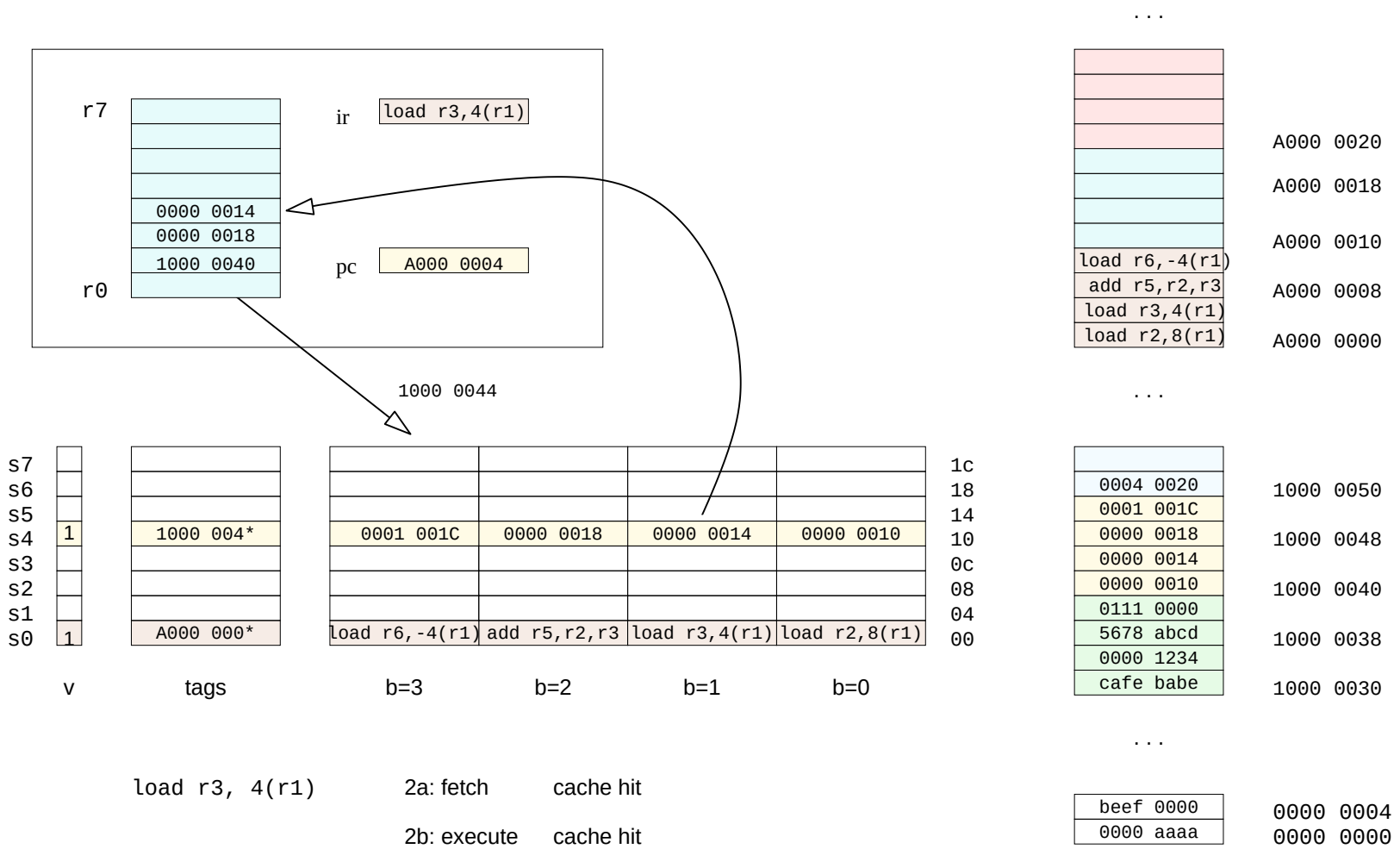
1a: fetch — load instruction into ir

1b: execute — load value into r2

Direct mapped cache: Beispiel



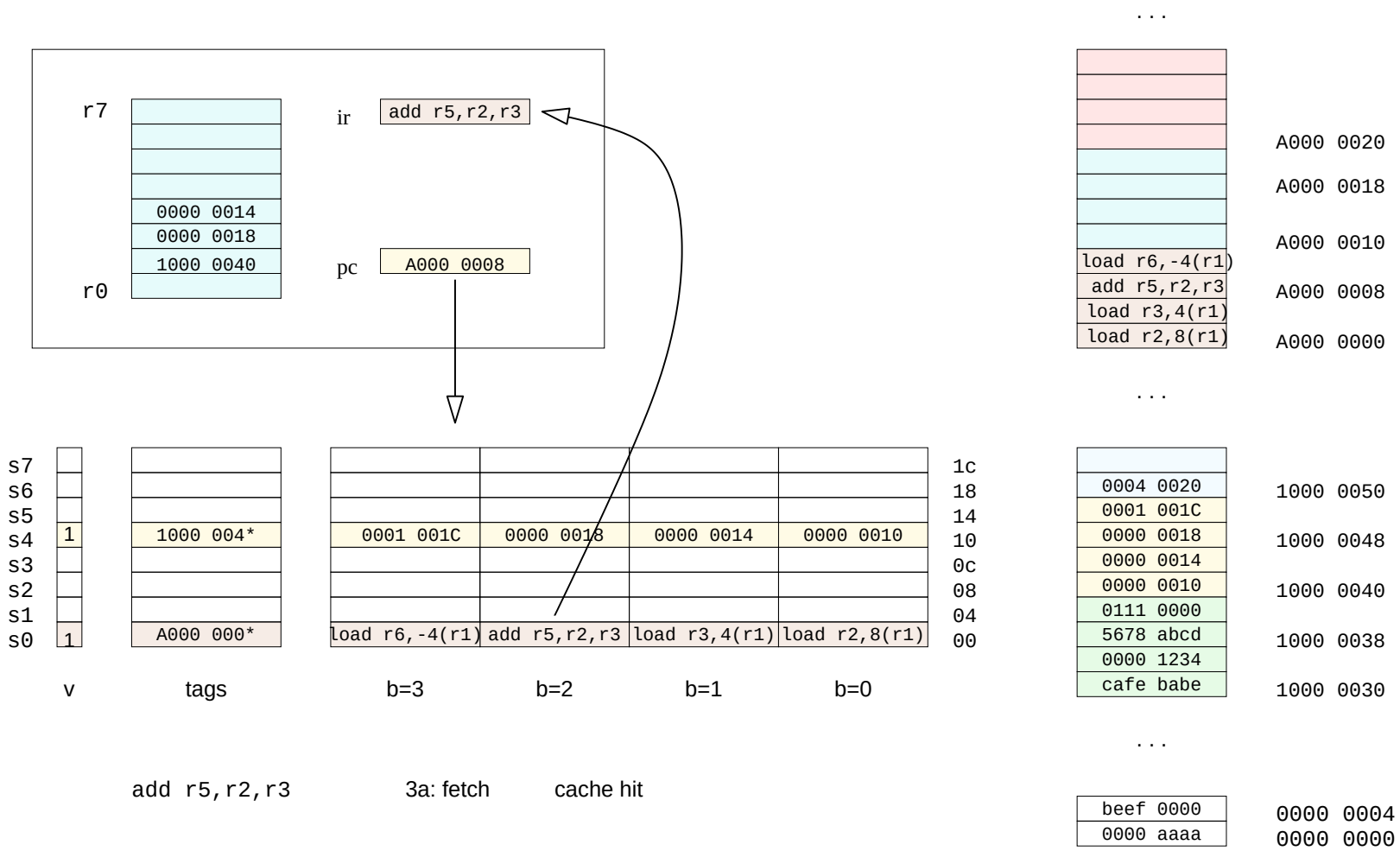
Direct mapped cache: Beispiel



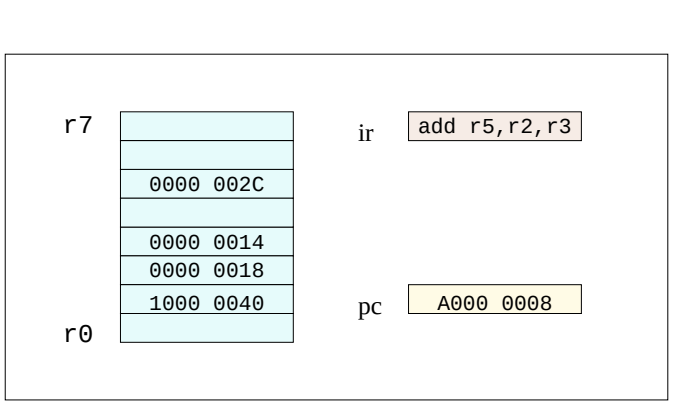
load r3, 4(r1) 2a: fetch cache hit

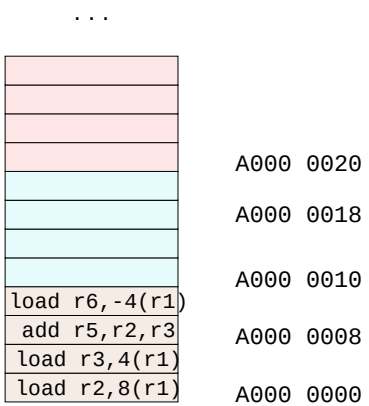
2b: execute cache hit

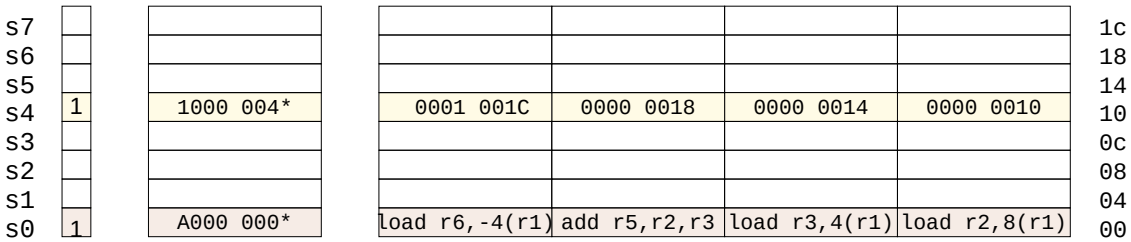
Direct mapped cache: Beispiel

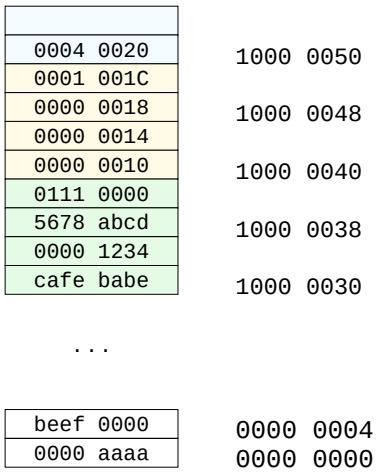


Direct mapped cache: Beispiel







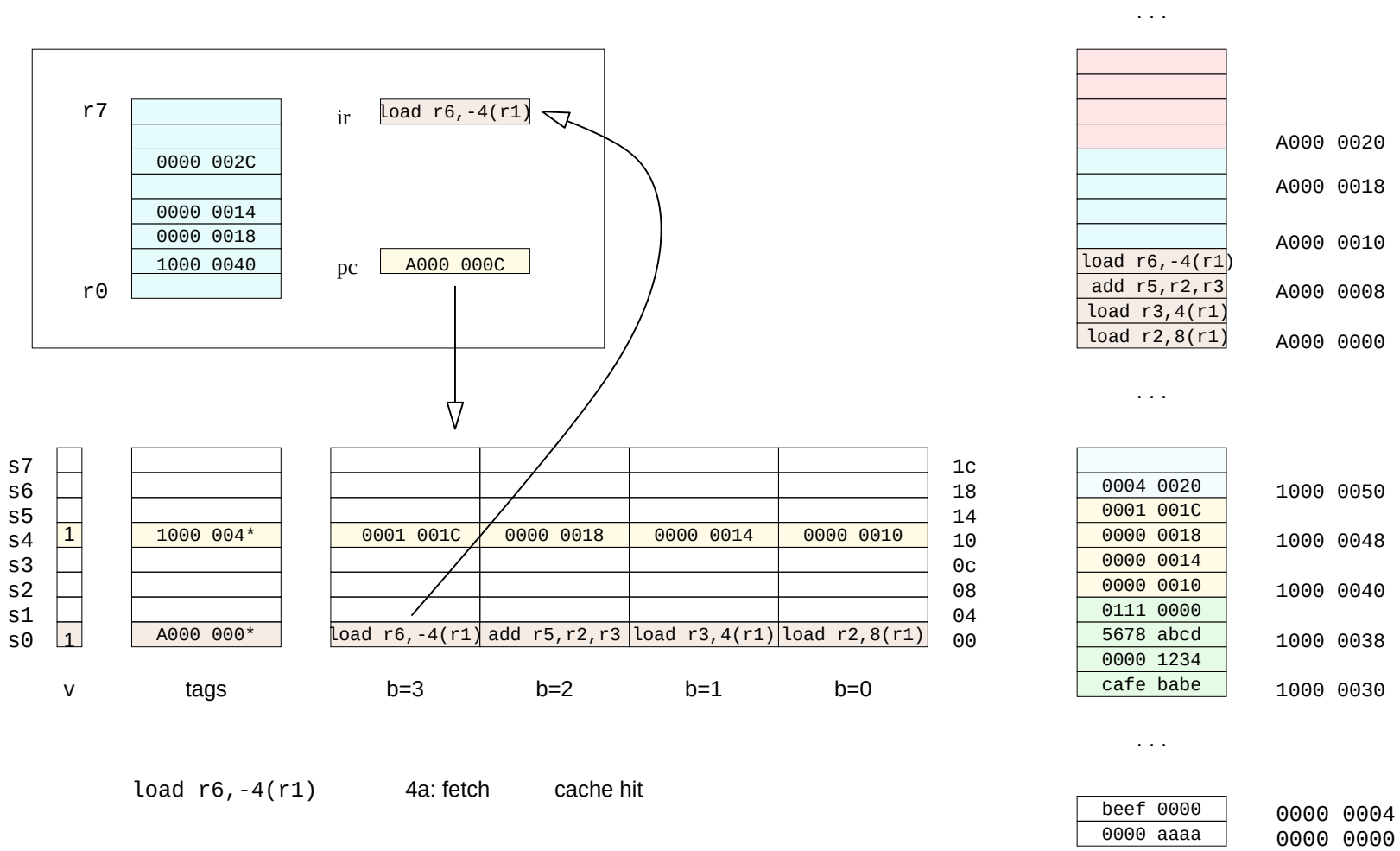


`add r5,r2,r3`

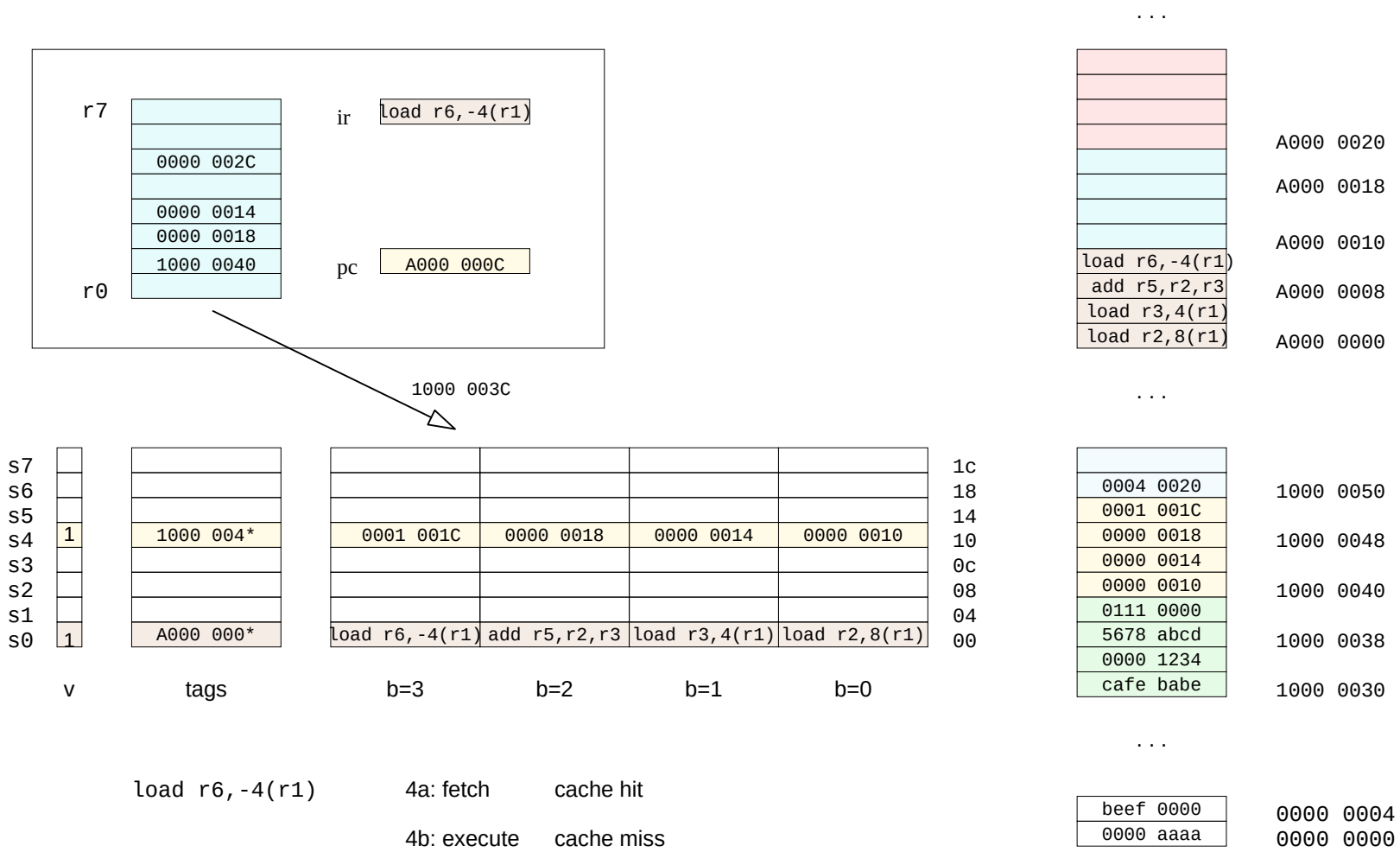
3a: fetch — cache hit

3b: execute — no memory access

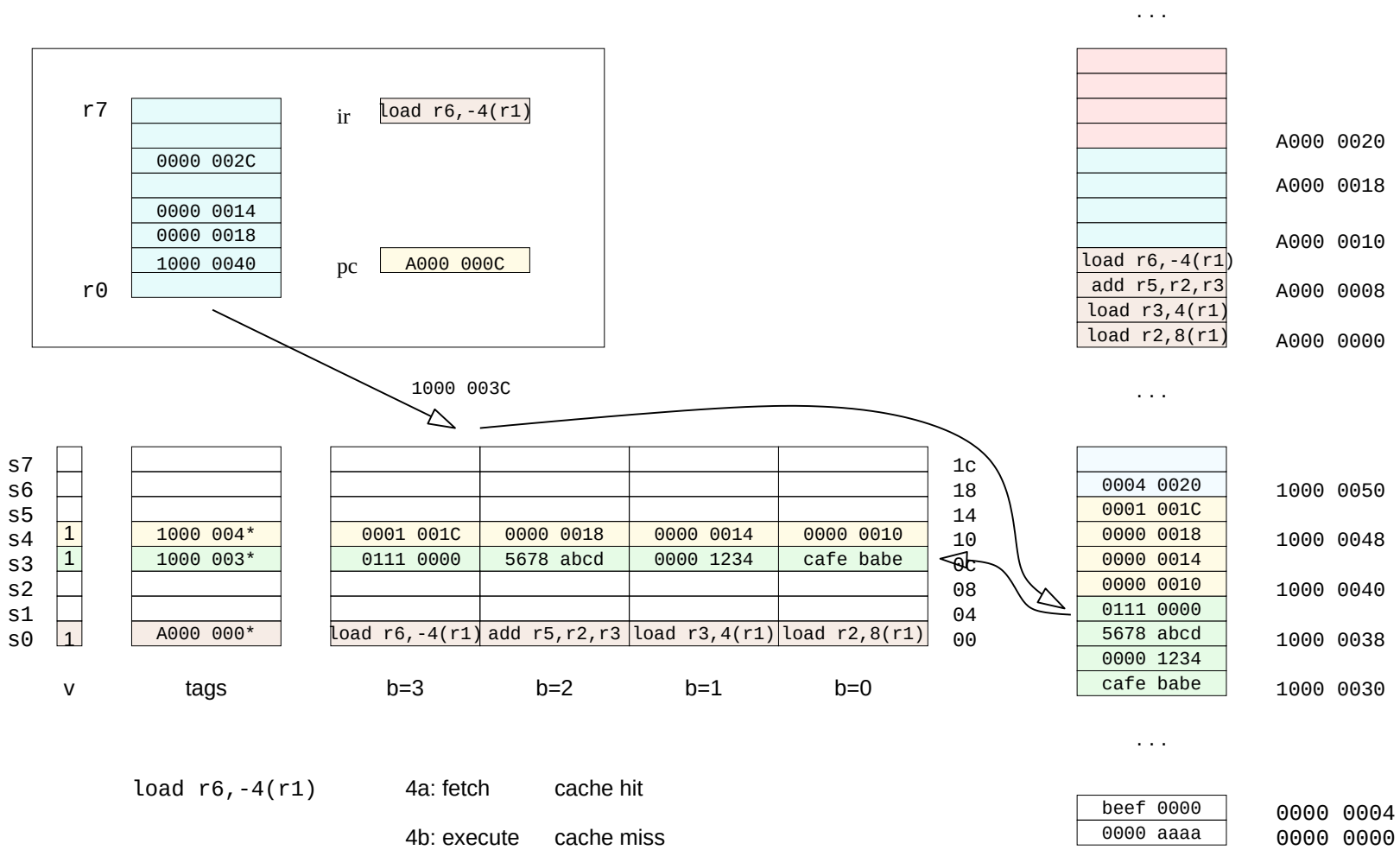
Direct mapped cache: Beispiel



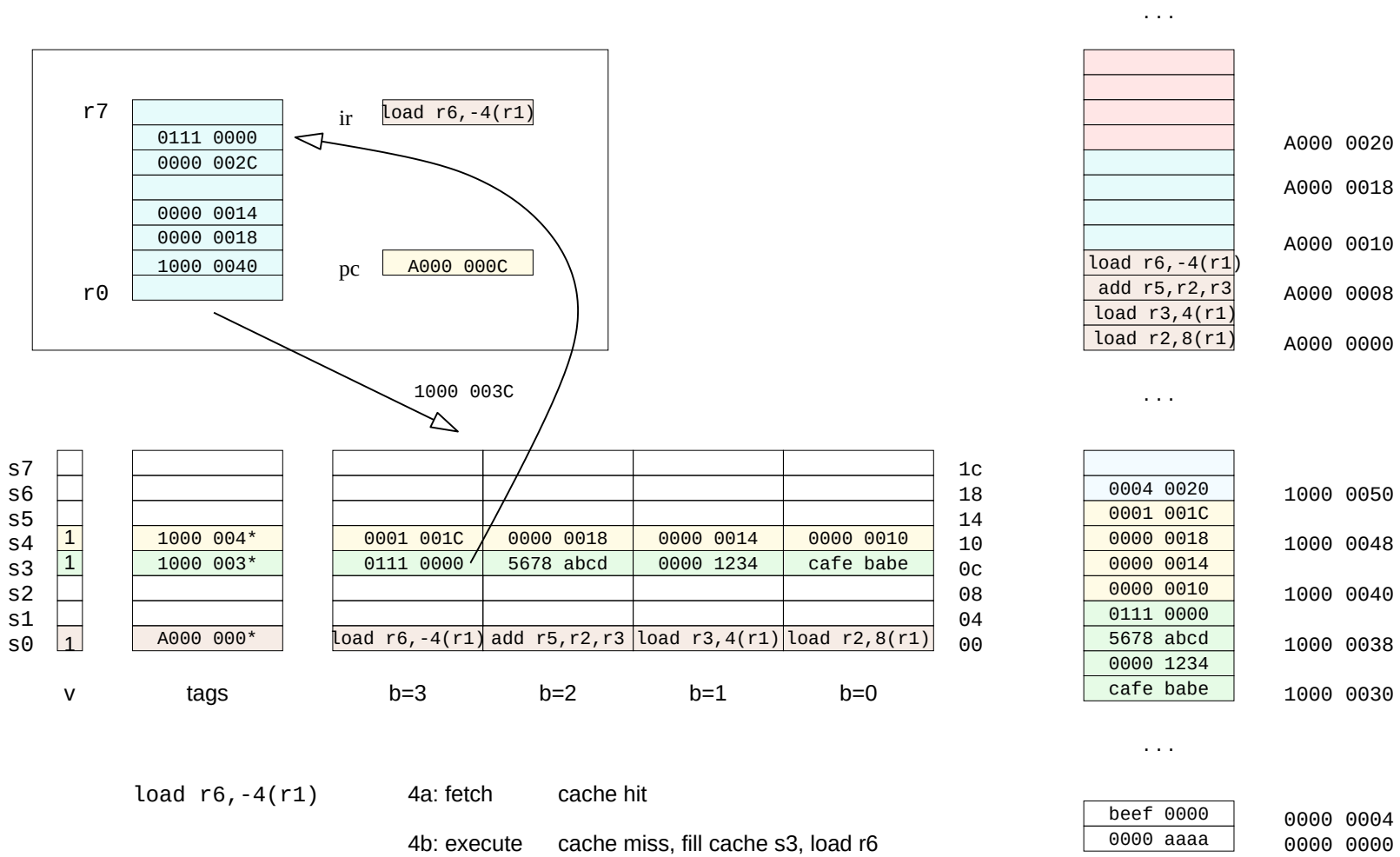
Direct mapped cache: Beispiel



Direct mapped cache: Beispiel



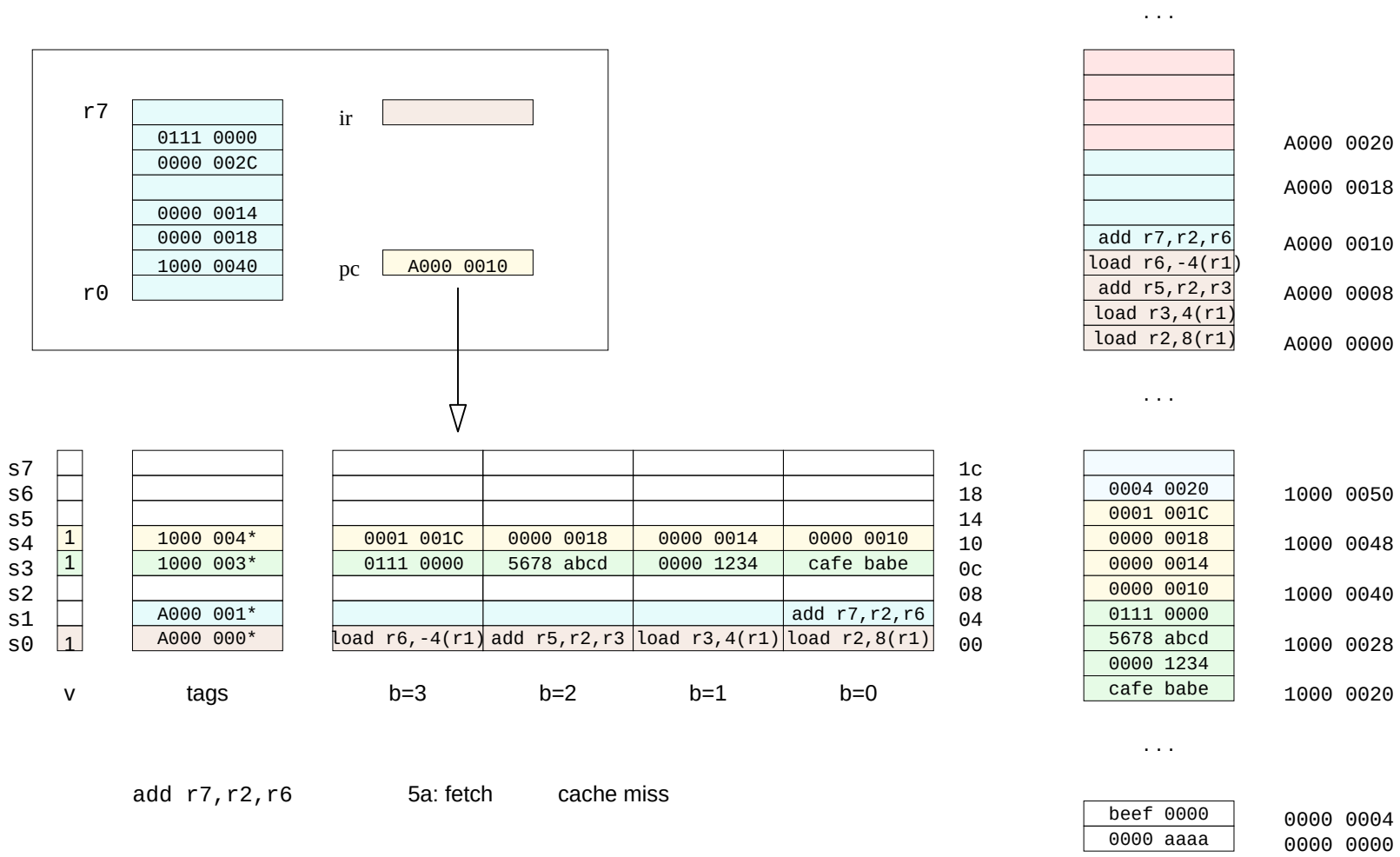
Direct mapped cache: Beispiel



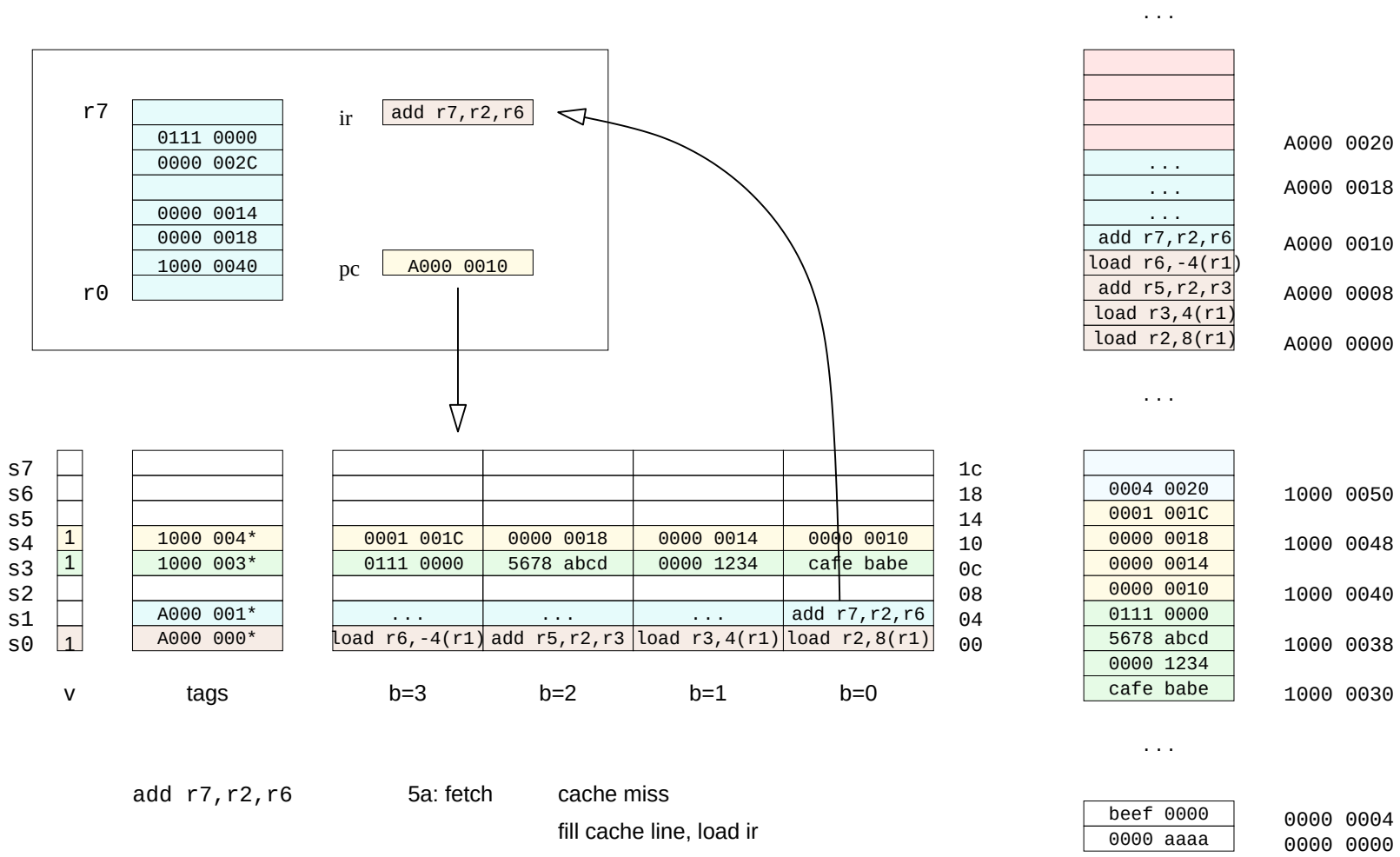
`load r6,-4(r1)` 4a: fetch cache hit

4b: execute cache miss, fill cache s3, load r6

Direct mapped cache: Beispiel



Direct mapped cache: Beispiel

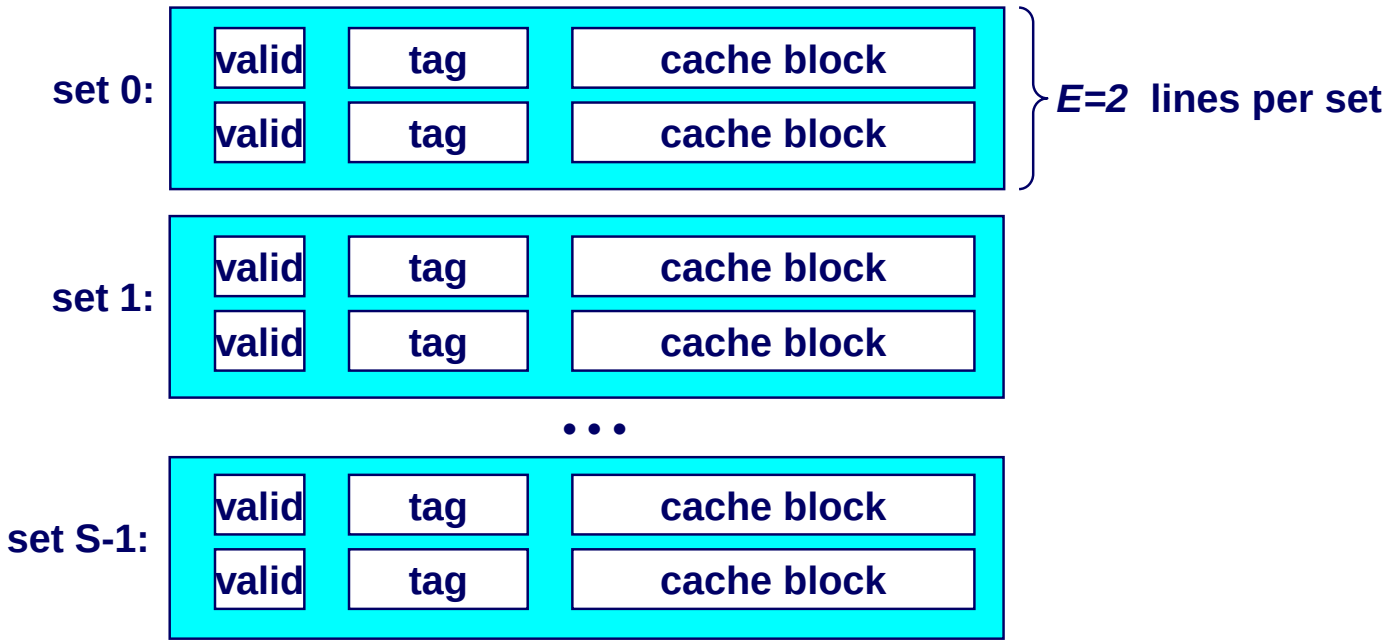


`add r7,r2,r6` 5a: fetch cache miss

fill cache line, load ir

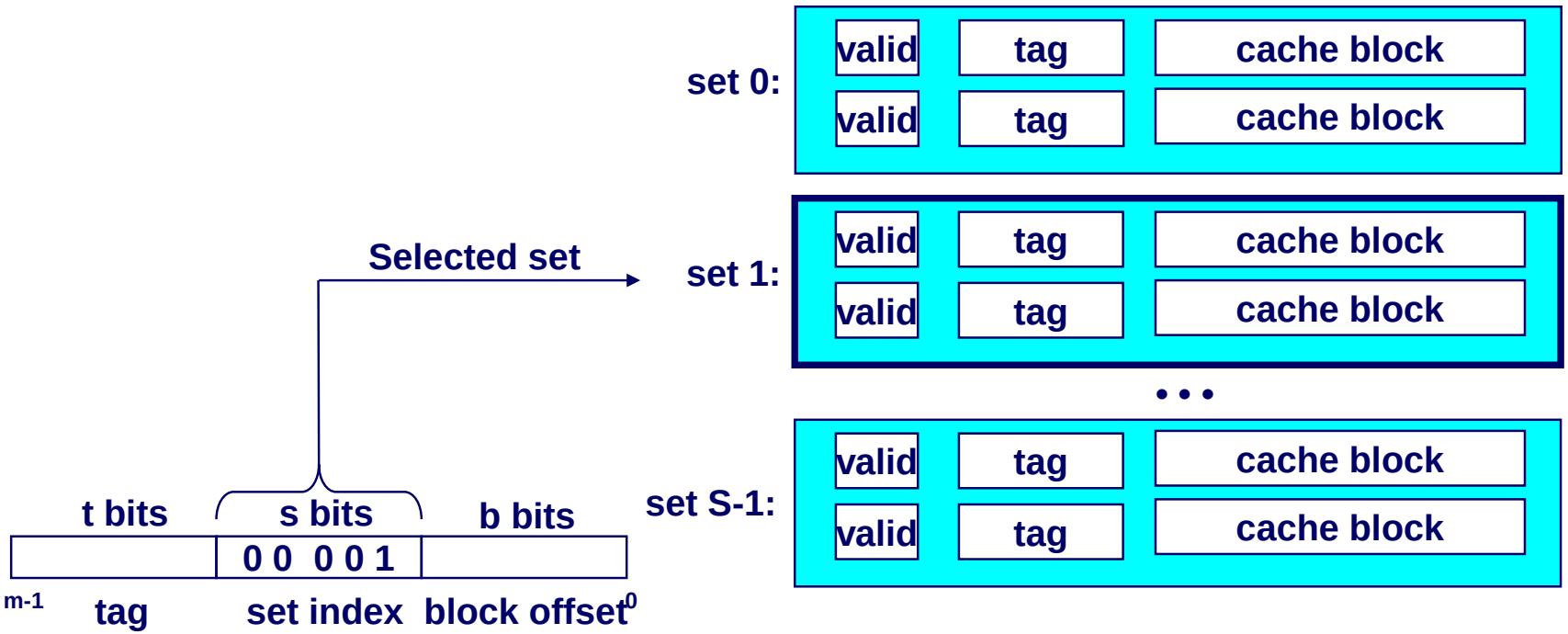
Cache: bereichsassoziativ / „set associative“

- 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. . . “



Zugriff auf n-fach assoziative Caches

1 Bereichsauswahl durch Bits ⟨*set index*⟩

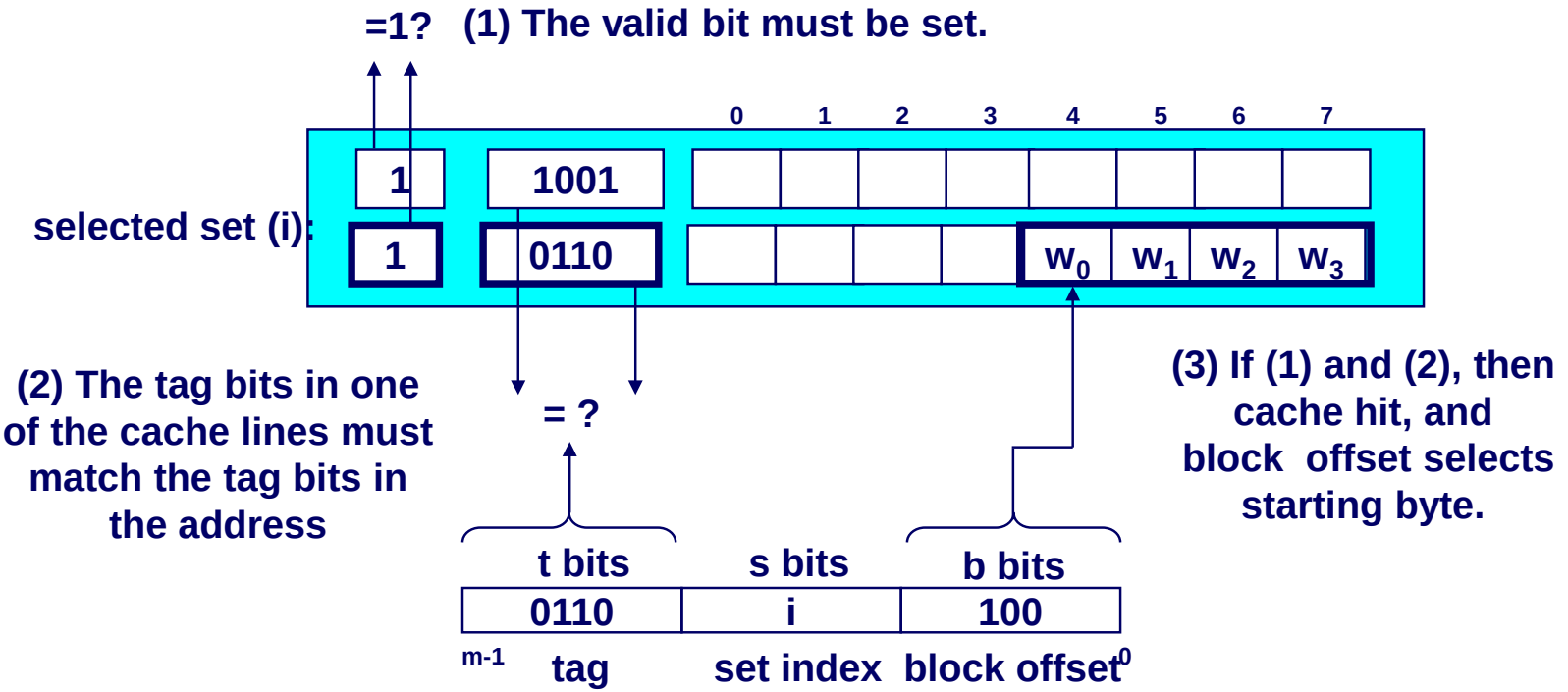


Zugriff auf n-fach assoziative Caches

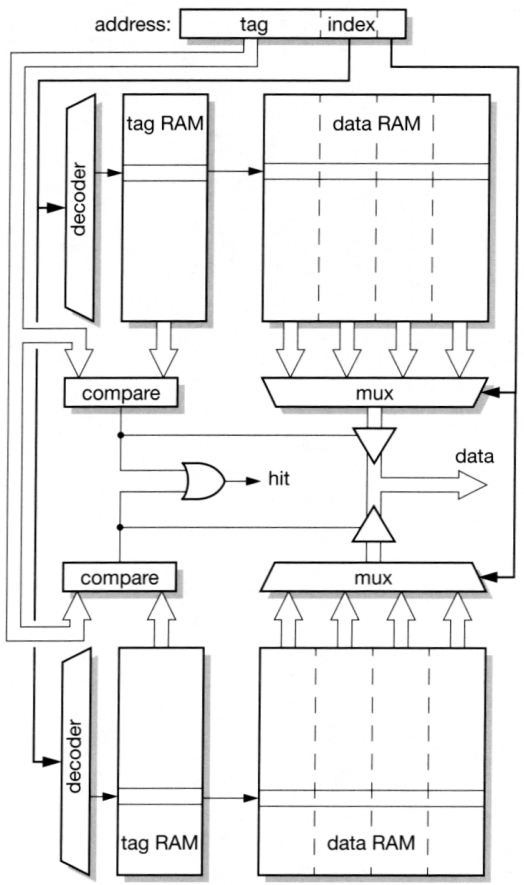
2 ⟨*valid*⟩: sind die Daten gültig?

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*⟩



2-fach set-associative Cache: Prinzip



Cache: voll-assoziativ

- jeder Adresse des Speichers kann jede beliebige Cachezeile zugeordnet werden, theoretisch beste Performance
- also Spezialfall: nur ein Cachebereich $S = 1$, aber $E =$ Anzahl der Lines

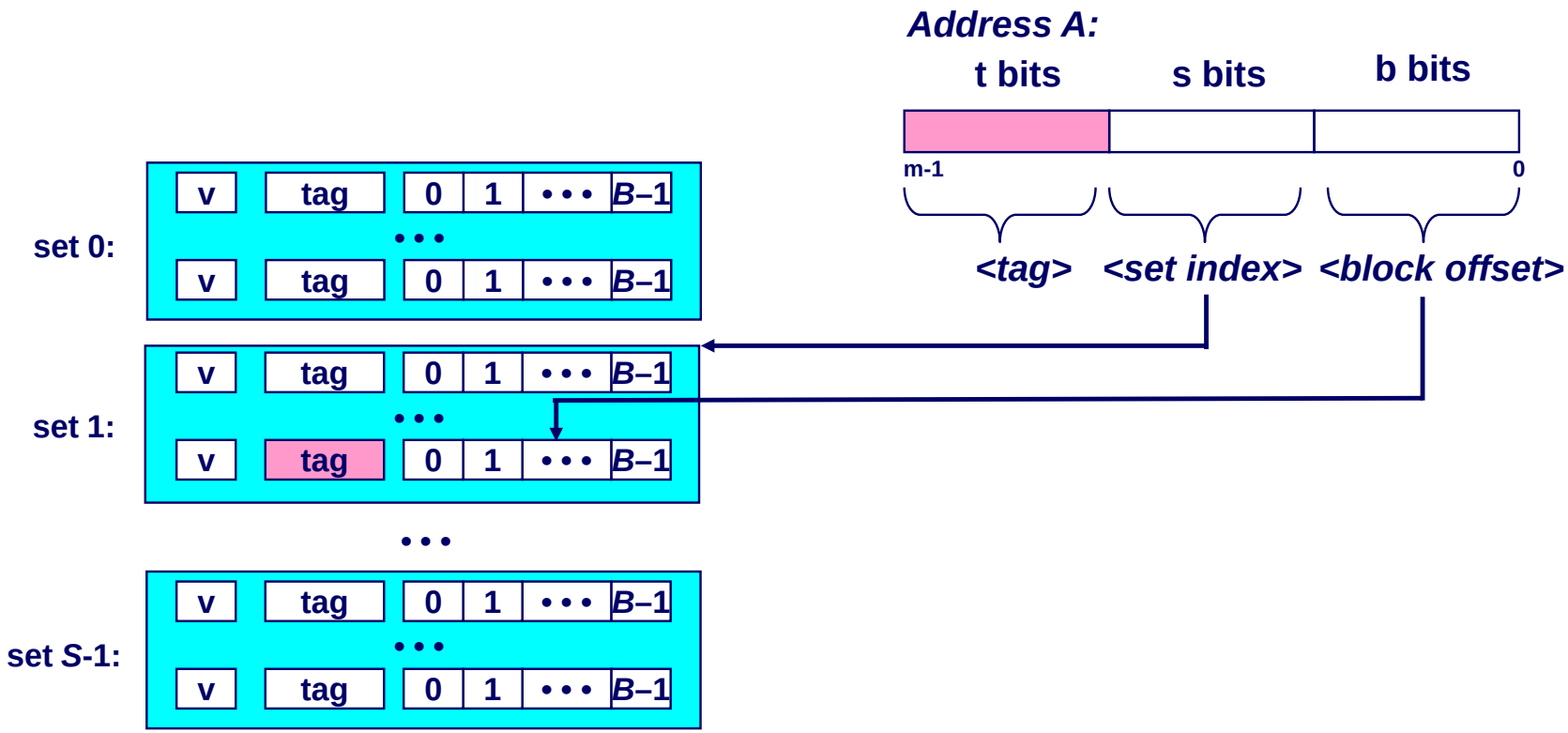
\+ besonders flexibel: Speicherung an allen Adressen im Cache

– benötigt E-Vergleicher

– derzeit nur für sehr kleine Caches realisierbar

– in der Praxis selten verwendet

Cache: Dimensionierung



- Parameter: S, B, E

Cache: Dimensionierung

- Blöcke mit mehreren Datenworten / „*Cache-Line*“
- Wortauswahl durch $\langle$*block offset*$\rangle$ in Adresse

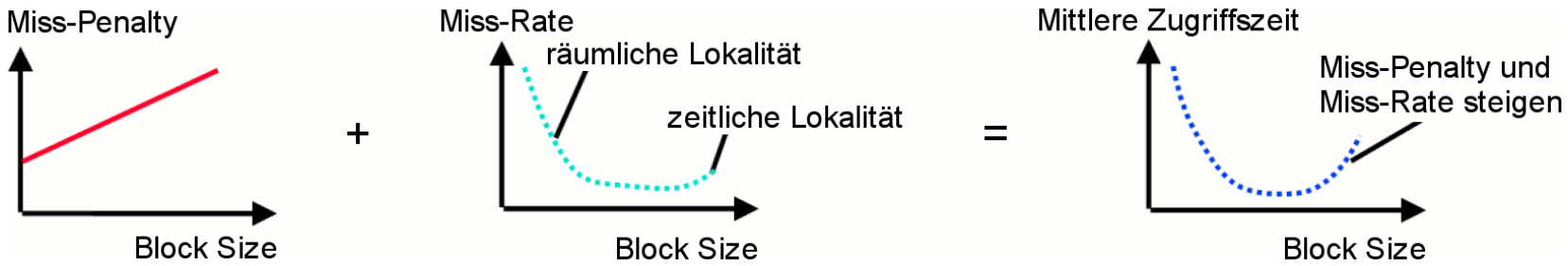
\+ nutzt räumliche Lokalität aus

\+ Breite externe Datenbusse

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

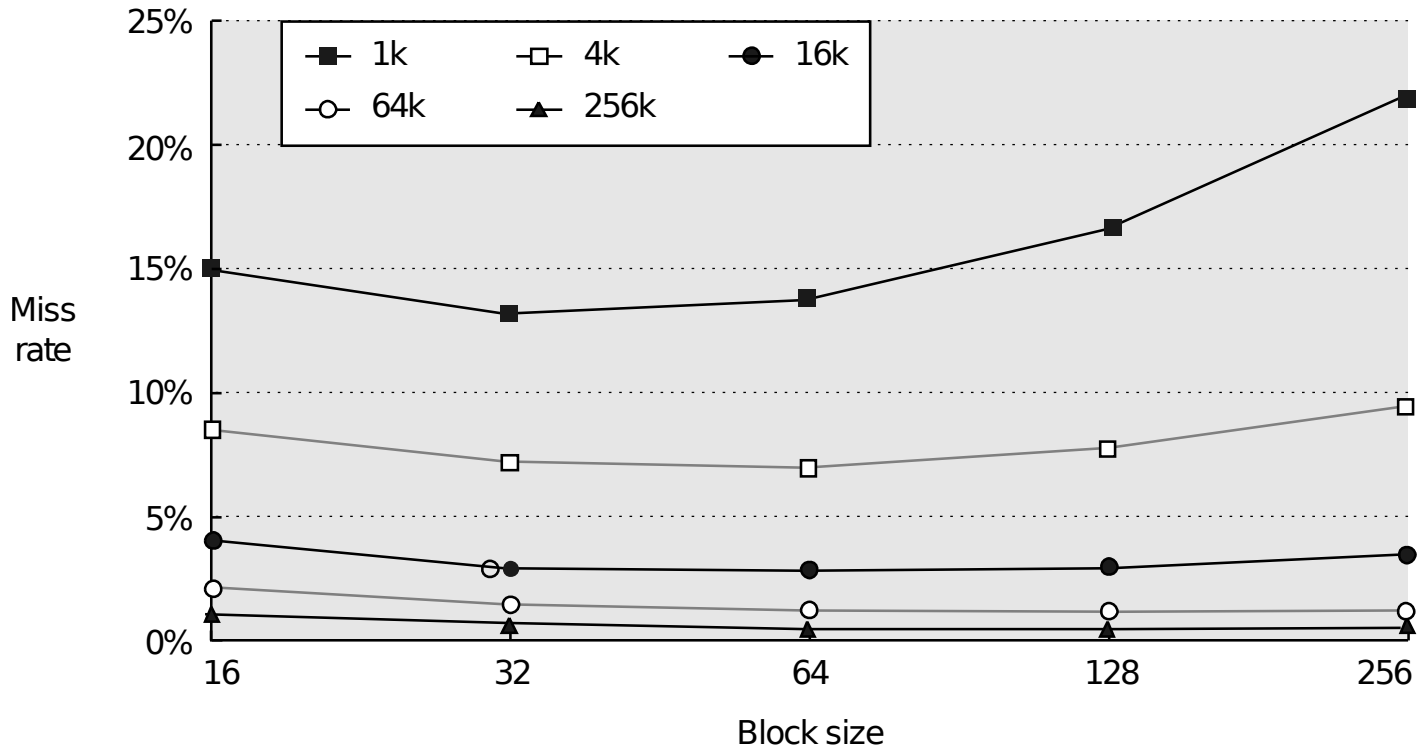
\+ kürzere interne Cache-Adressen

Cache- und Block-Dimensionierung



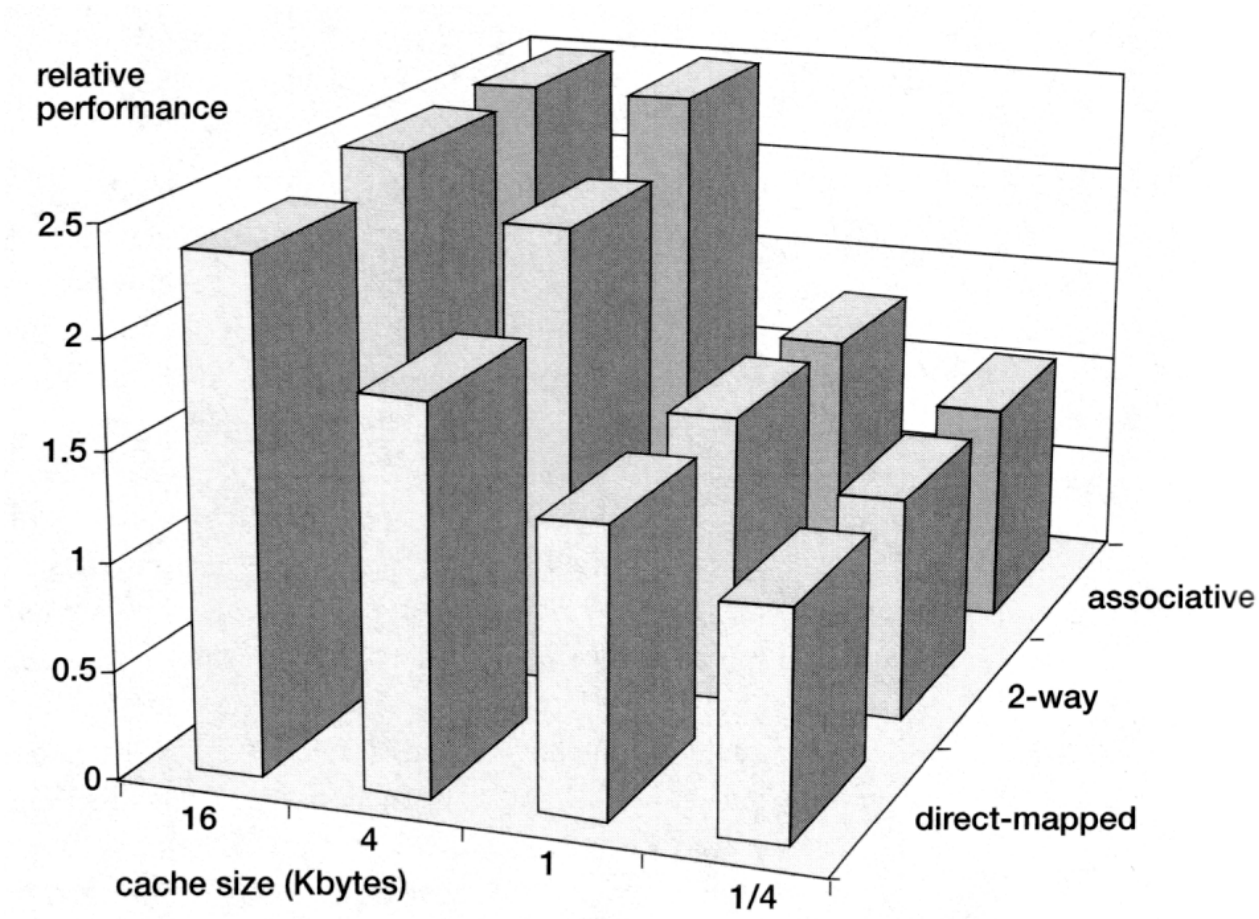
- Blockgröße klein, viele Blöcke
  - \+ kleinere Miss-Penalty
  - \+ temporale Lokalität
  - – räumliche Lokalität
- Blockgröße groß, wenig Blöcke
  - – größere Miss-Penalty
  - – temporale Lokalität
  - \+ räumliche Lokalität

Cache- und Block-Dimensionierung



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

Cache: Dimensionierung, Relative Performance



Furber, Arm SoC Architecture

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 „least recently used“:
  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 „least frequently used“:
  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 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
- MESI-Protokoll: nächste Woche

Cache: Performance

- 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

Cache: Performance

⇒ Verbesserung der Cache Performanz durch kleinere R_{Miss}

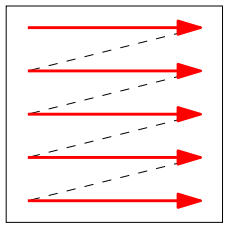
- größere Caches (– mehr Hardware)
- höhere Assoziativität (– langsamer)

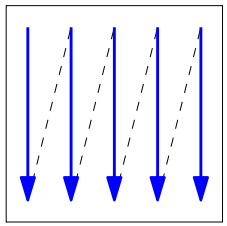
⇒ Optimierungstechniken

- 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

Cache: Matrix-Beispiel aus Teil 1

```
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;
}
```

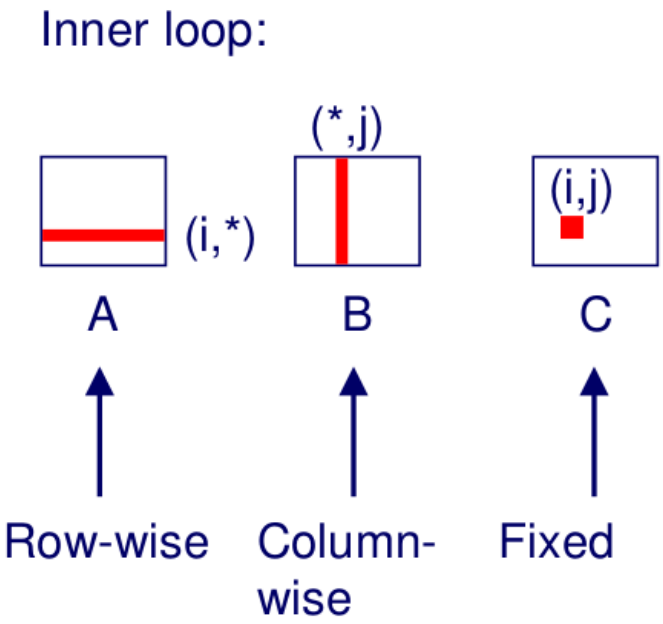




```
Matrix (5000,5000) creation took 2105 msec.
Matrix row-col summation took 75 msec.
Matrix col-row summation took 383 msec.          (5x langsamer)
Sum is 600.8473695346258 600.8473695342268       (andere Werte)
```

Cache: Beispiel Matrix-Multiplikation

```
/* 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;
  }
}
```

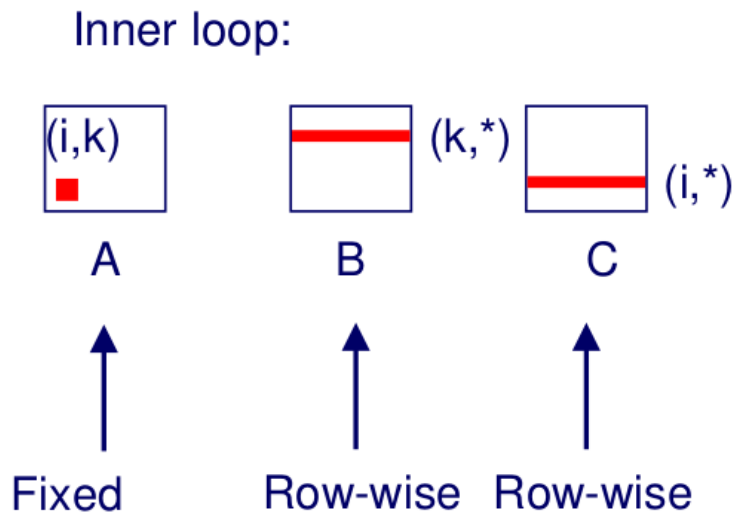


Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Cache: Beispiel Matrix-Multiplikation

```
/* 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];
  }
}
```



Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Cache: Beispiel Matrix-Multiplikation

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;
  }
}
```

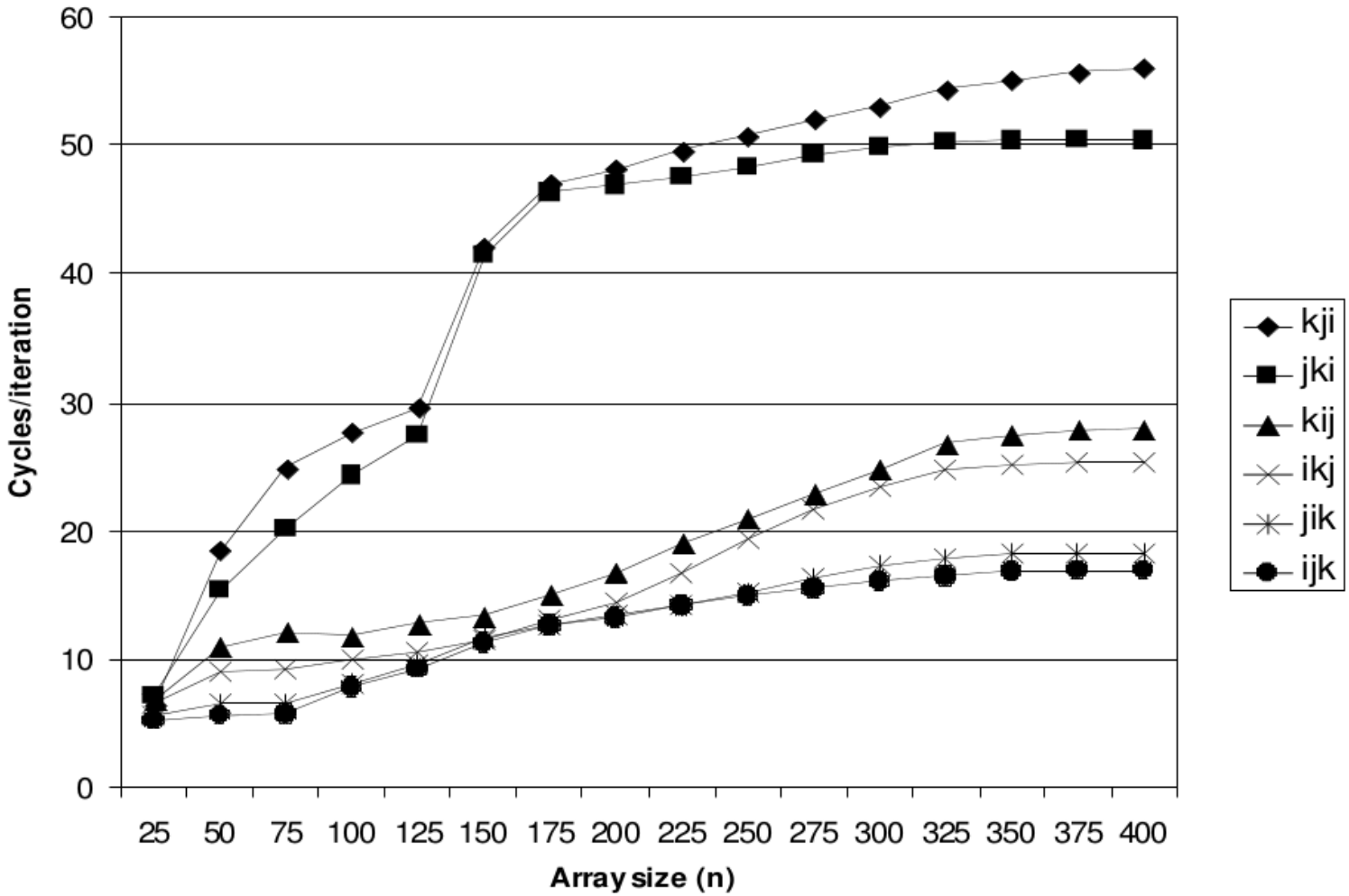
kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

```
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];
  }
}
```

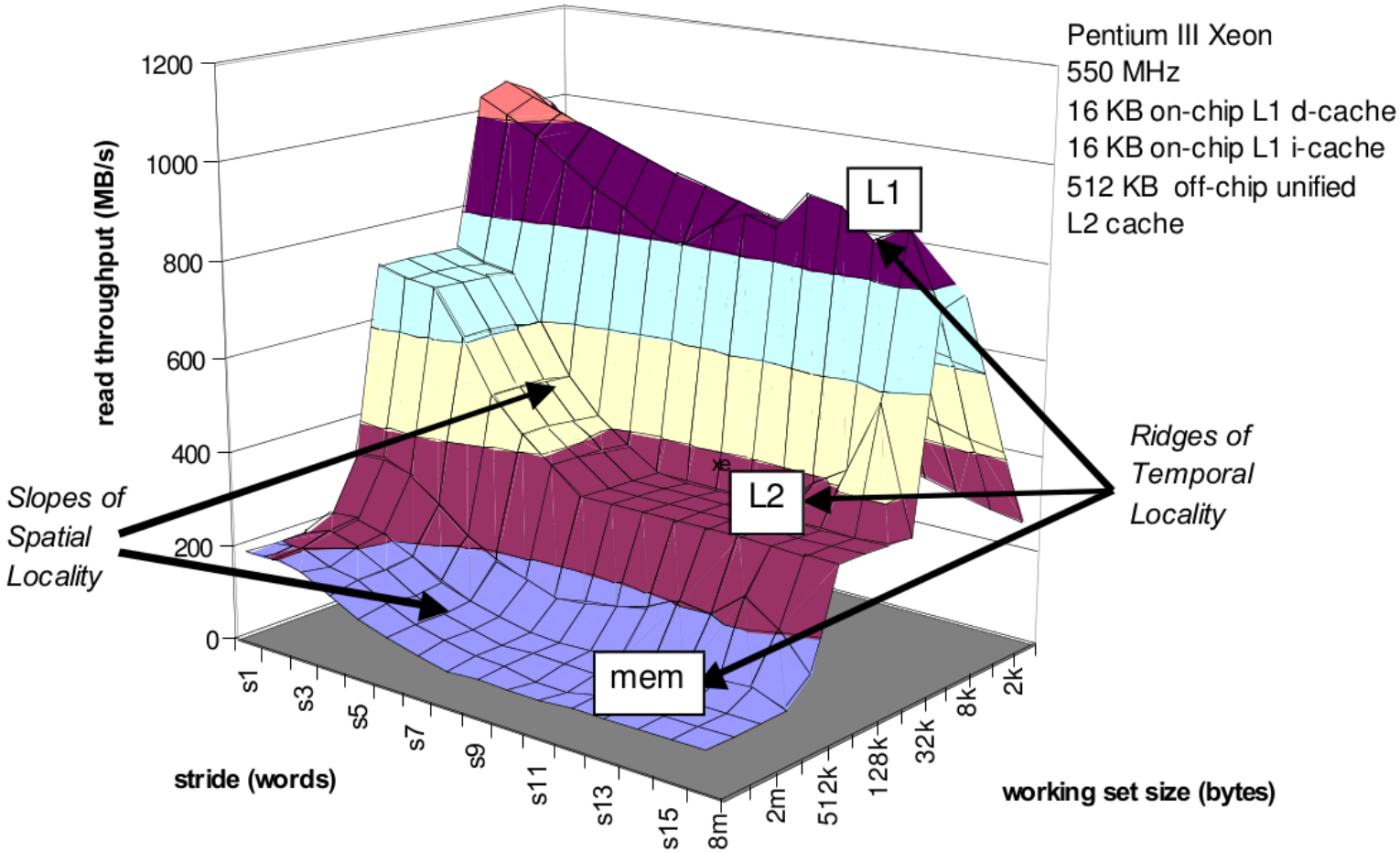
jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
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;
  }
}
```

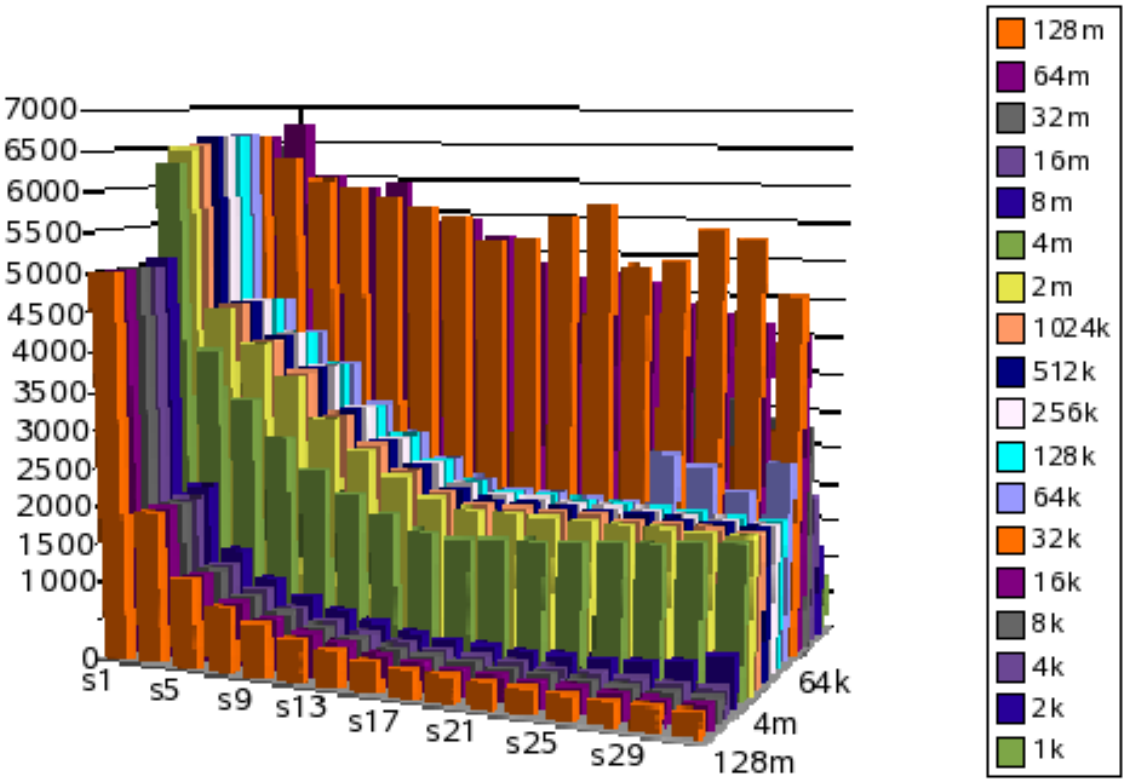
Cache: Beispiel Matrix-Multiplikation



Cache: Memory Mountain

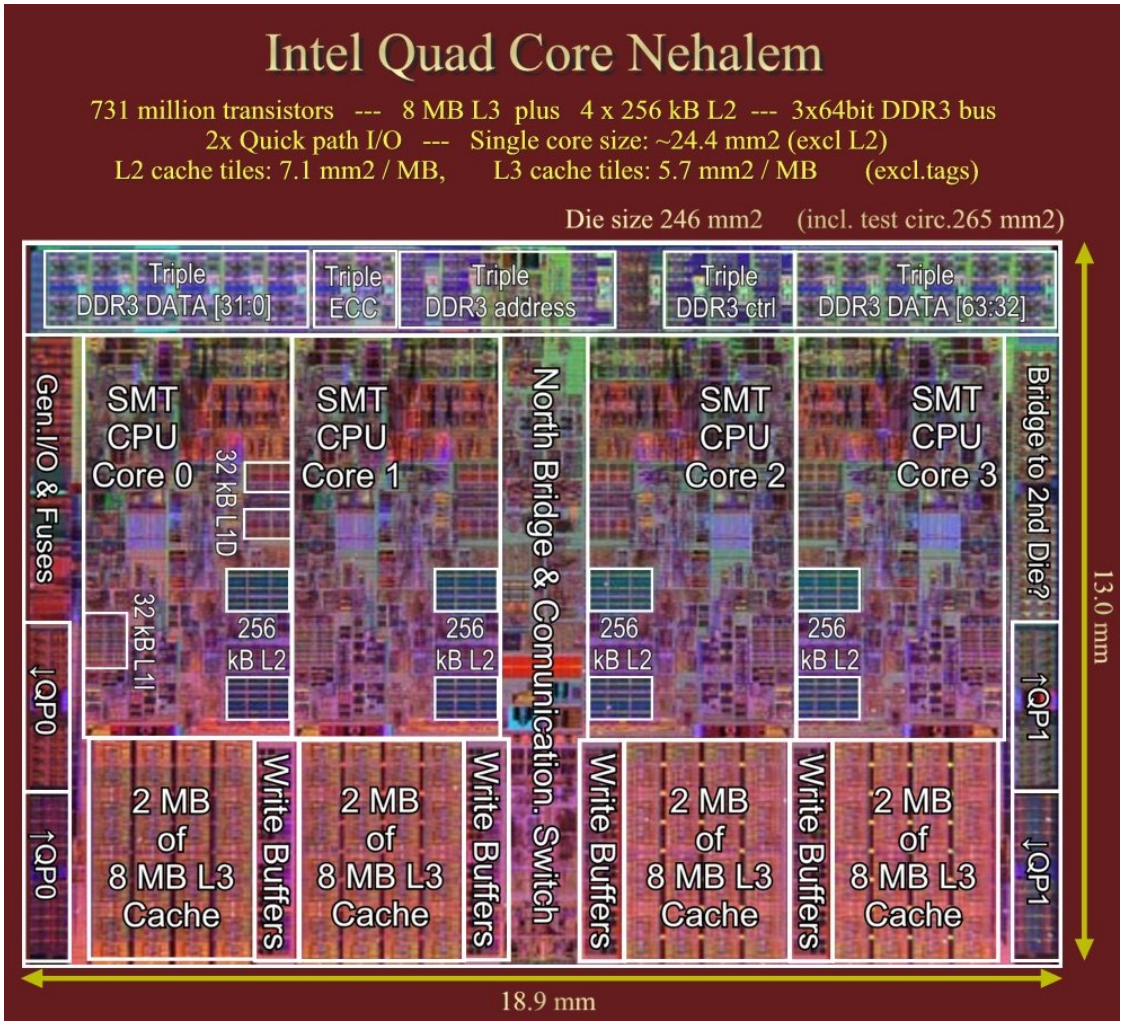


Cache: Memory Mountain



- Transferrate in MB/s vs. Blockgröße und Stride
- Beispiel-PC: 7 GB/s max., 200 MB/s sustained: Faktor 35

Chiplayout: Intel Core-i Prozessoren „Nehalem“



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

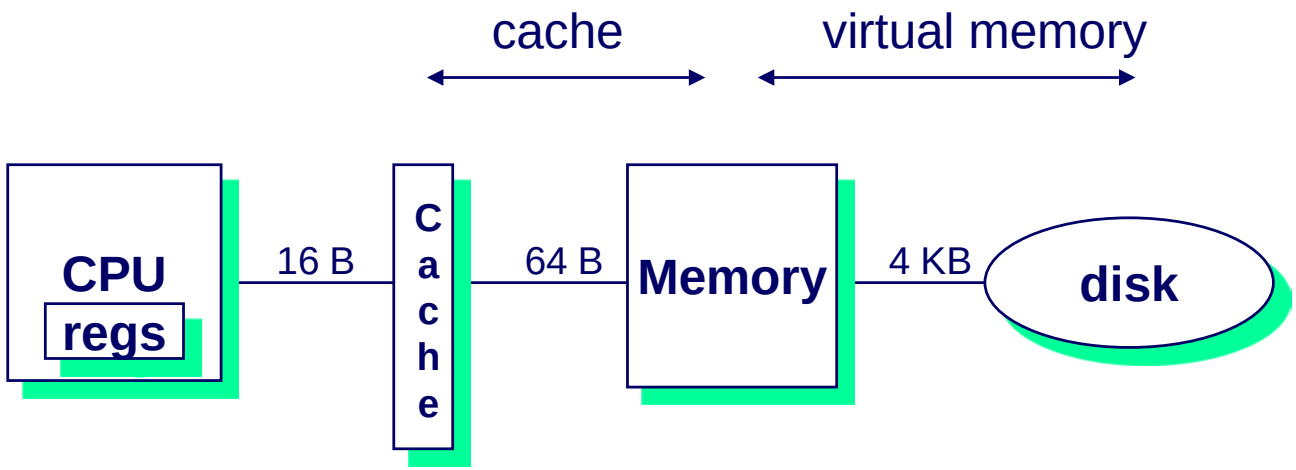
$\Rightarrow$ widersprüchliche Anforderungen

$\Rightarrow$ Lösung mit virtuellem Speicher und Paging

Virtueller Speicher: Idee

1. Benutzung der Festplatte als *erweiterten Hauptspeicher*
   - Prozessadressraum kann physikalische Speichergröße übersteigen
   - Summe der Adressräume mehrerer Prozesse kann physikalischen Speicher übersteigen
2. Unterstützung mehrerer Prozesse
   - mehrere Prozesse gleichzeitig im Hauptspeicher
   - jeder Prozess hat seinen eigenen Adressraum (0...n)
   - nur *aktiver* Code und Daten sind tatsächlich im Speicher
   - bedarfsabhängige, dynamische Speicherzuteilung
3. Bereitstellung von Schutzmechanismen
   - ein Prozess kann einen anderen nicht beeinflussen
   - Benutzerprozess hat keinen Zugriff auf privilegierte Informationen
   - jeder virtuelle Adressraum hat eigene Zugriffsrechte

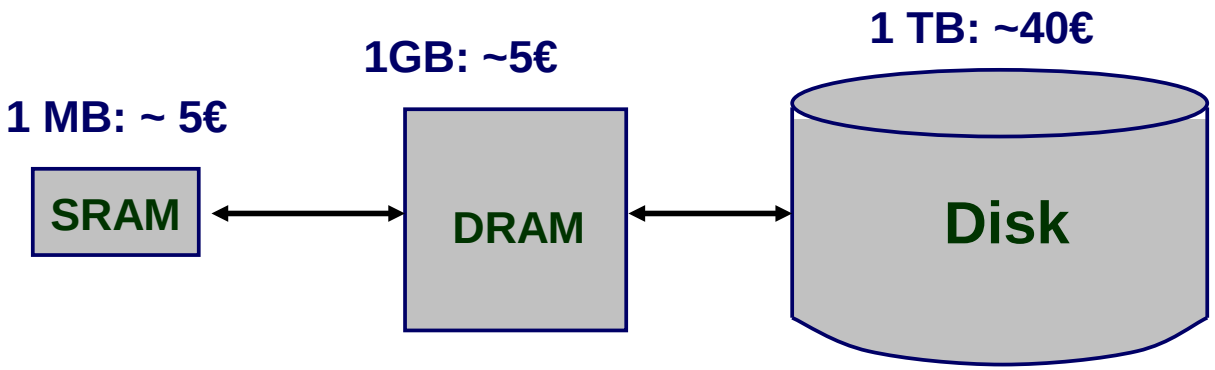
Ebenen in der Speicherhierarchie



	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

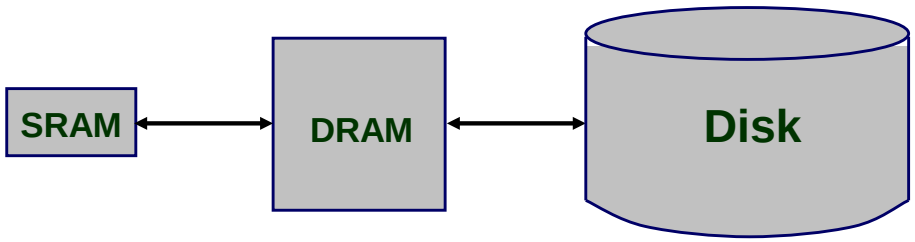
Festplatte „erweitert“ Hauptspeicher



- Adressraum sehr groß $\Rightarrow$ nur teilweise als DRAM realisiert
  - 32-bit Adressen: $\approx 4 \times 10^9$ Byte
  - 64-bit Adressen: $\approx 16 \times 10^{16}$ Byte
- Speichern auf Festplatte ist $\approx 125\times$ billiger als im DRAM
  - 1 TB DRAM: $\approx 5\,000\,€$
  - 1 TB Festplatte: $\approx 40\,€$

$\Rightarrow$ kostengünstiger Zugriff auf große Datenmengen

Festplatte als Hauptspeicher: Zugriffszeiten

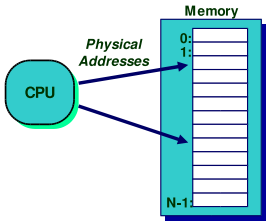


- Zugriff auf Daten auf der Festplatte ist langsam
- Unterschied DRAM vs. Festplatte ist extremer als SRAM vs. DRAM
  - DRAM $\approx 10\times$ langsamer als SRAM
  - Festplatte $\approx 200\,000\times$ langsamer als DRAM

$\Rightarrow$ Nutzung der räumlichen Lokalität wichtig

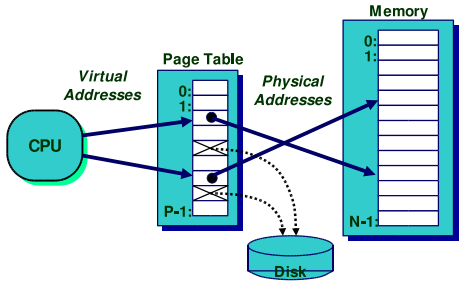
  - erstes Byte $\approx 200\,000\times$ langsamer als nachfolgende Bytes

System ohne/mit virtuellem Speicher



System mit rein physikalischem Speicher (SRAM oder DRAM)

- Programme addressieren den Speicher direkt
- fast alle Microcontroller
- frühe PCs (DOS)
- Cray Supercomputer



System mit MMU und virtuellem Speicher (DRAM plus Festplatte)

- Memory-Management-Unit
- Programme addressieren den Speicher virtuell
- virtuelle Adressen werden von der MMU über eine „Seiten-Tabelle“ umgesetzt
- Workstations, moderne PCs

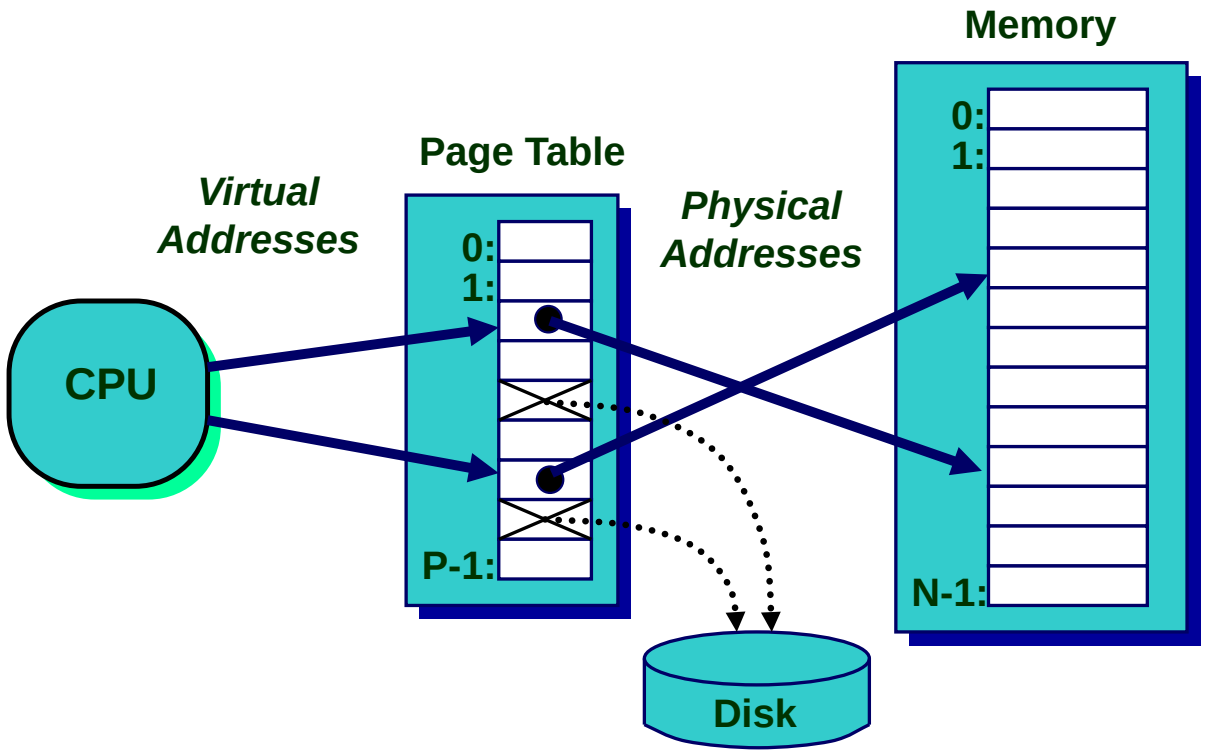
Prinzip des virtuellen Speichers

- jeder Prozess bekommt seinen eigenen virtuellen Adressraum
- Kombination aus Betriebssystem und Hardwareeinheiten
  - Umsetzung von virtuellen zu physikalischen Adressen
  - Umsetzungstabellen werden vom Betriebssystem verwaltet
  - wegen des Speicherbedarfs der Tabellen beziehen sich diese auf größere Speicherblöcke: „Seiten“
  - Umgesetzt wird nur die Anfangsadresse der Seite, der Offset innerhalb des Blocks bleibt unverändert

- Blöcke des virtuellen Adressraums können durch das Betriebssystem auf Festplatte ausgelagert werden
  - Windows: Auslagerungsdatei
  - Unix/Linux: `swap` Partition und -Datei(en)

Virtueller Speicher: Paging / Seitenadressierung

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



Paging / Seitenadressierung: Vorteile

Abbildung der Adressen in den virtuellen Speicher

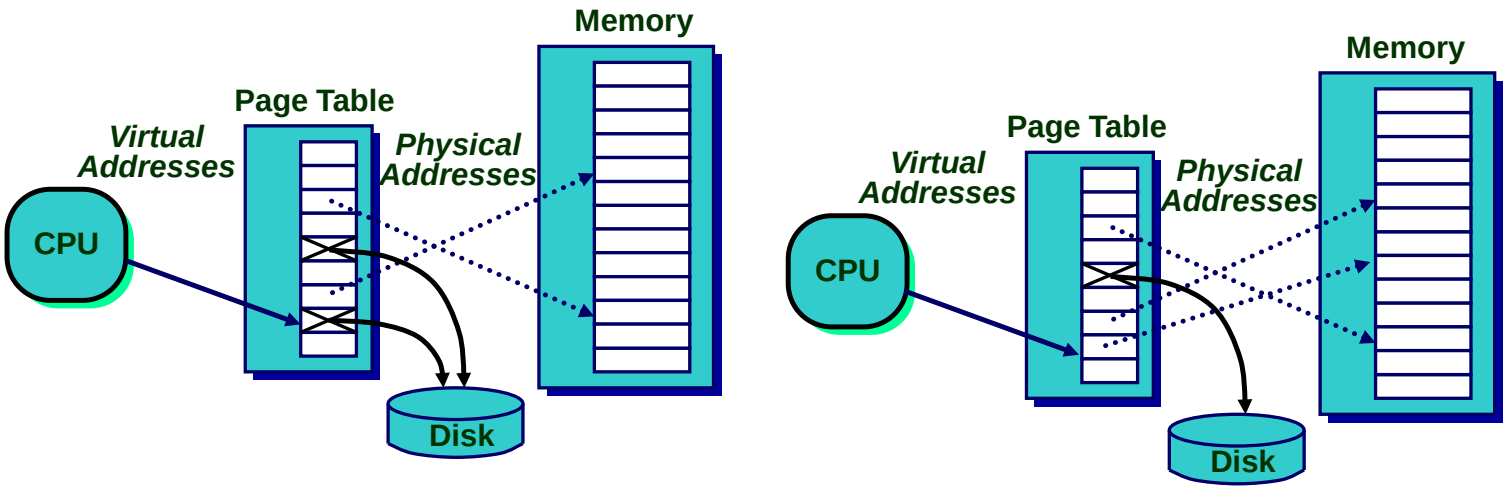
- Programme/Daten können größer als der Hauptspeicher sein
- Programme können an beliebige physikalischen Adressen geladen werden, unabhängig von der Aufteilung des physikalischen Speichers
- feste Seitengröße: effiziente Umsetzung in Hardware
- Zugriffsrechte für jede Seite (read/write, User/Supervisor)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

Paging / Seitenadressierung: Nachteile

- große Miss-Penalty beim Zugriff auf ausgelagerte Seiten
  - Zeitbedarf für das Nachladen von der Platte
  - Seiten sollten relativ groß sein: 4...64 kB
- Speicherplatzbedarf der Seitentabelle
  - abhängig vom Adressraum und Seitengröße
  - ab 32-bit Adressraum sehr große Tabellen
    - mehrstufige Tabellen
    - Hash-Verfahren (*inverted page tables*)
    - Kombinationen davon

Seitenfehler

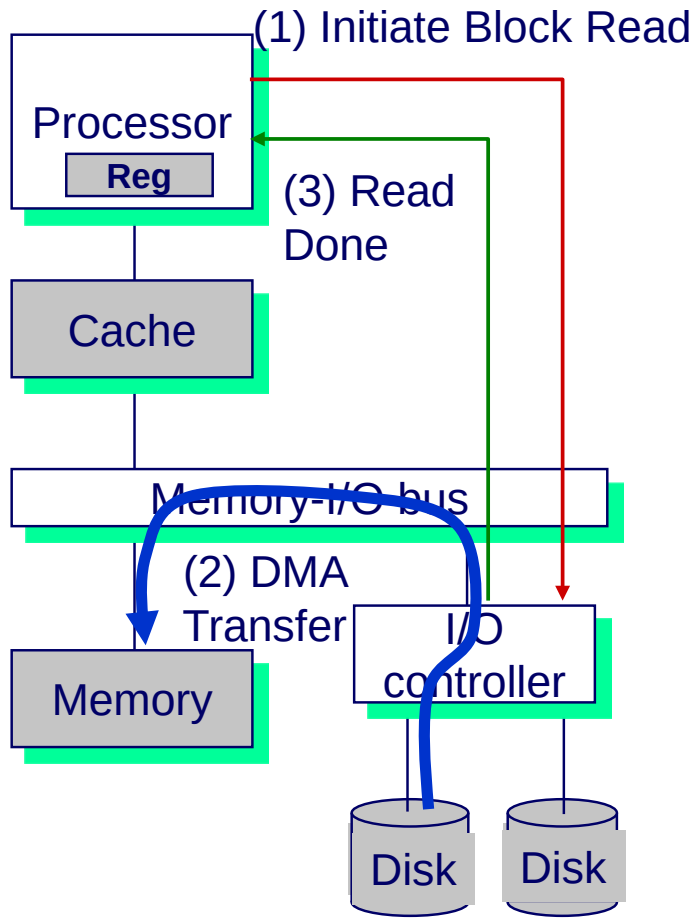
- Seiten-Tabelleneintrag: Startadresse der virt. Seite auf Platte
- Daten von Festplatte in Speicher laden:
  Aufruf des „Exception handler“ des Betriebssystems
  - laufender Prozess wird unterbrochen, andere können weiterlaufen
  - Betriebssystem kontrolliert die Platzierung der neuen Seite im Hauptspeicher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

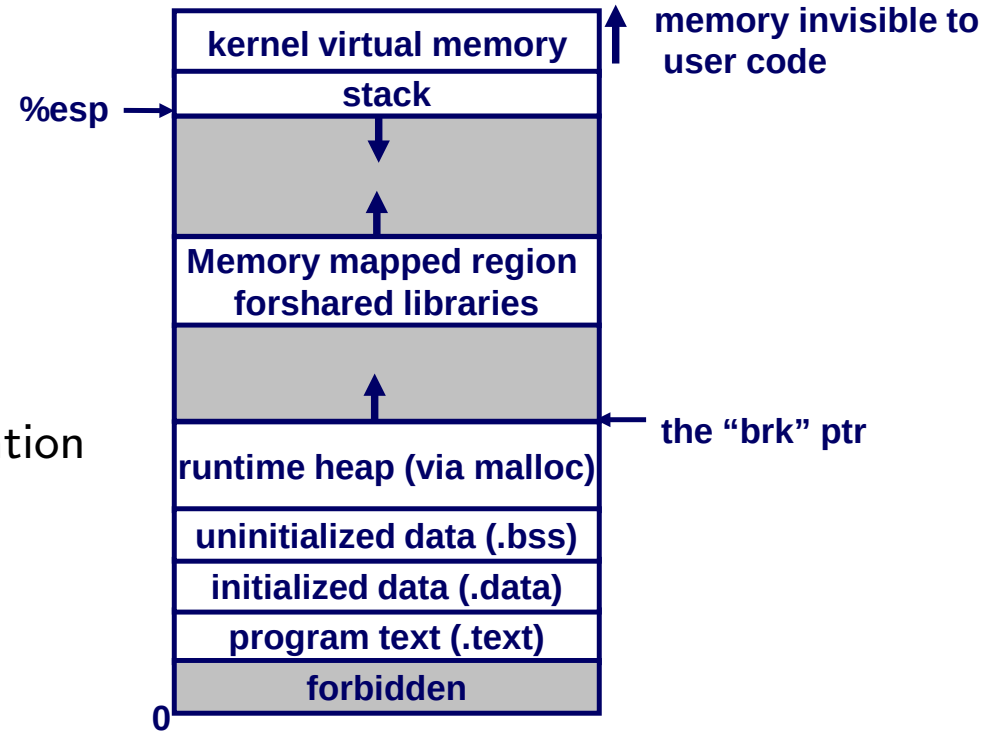


Separate virtuelle Adressräume

Mehrere Prozesse können im physikalischen Speicher liegen

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

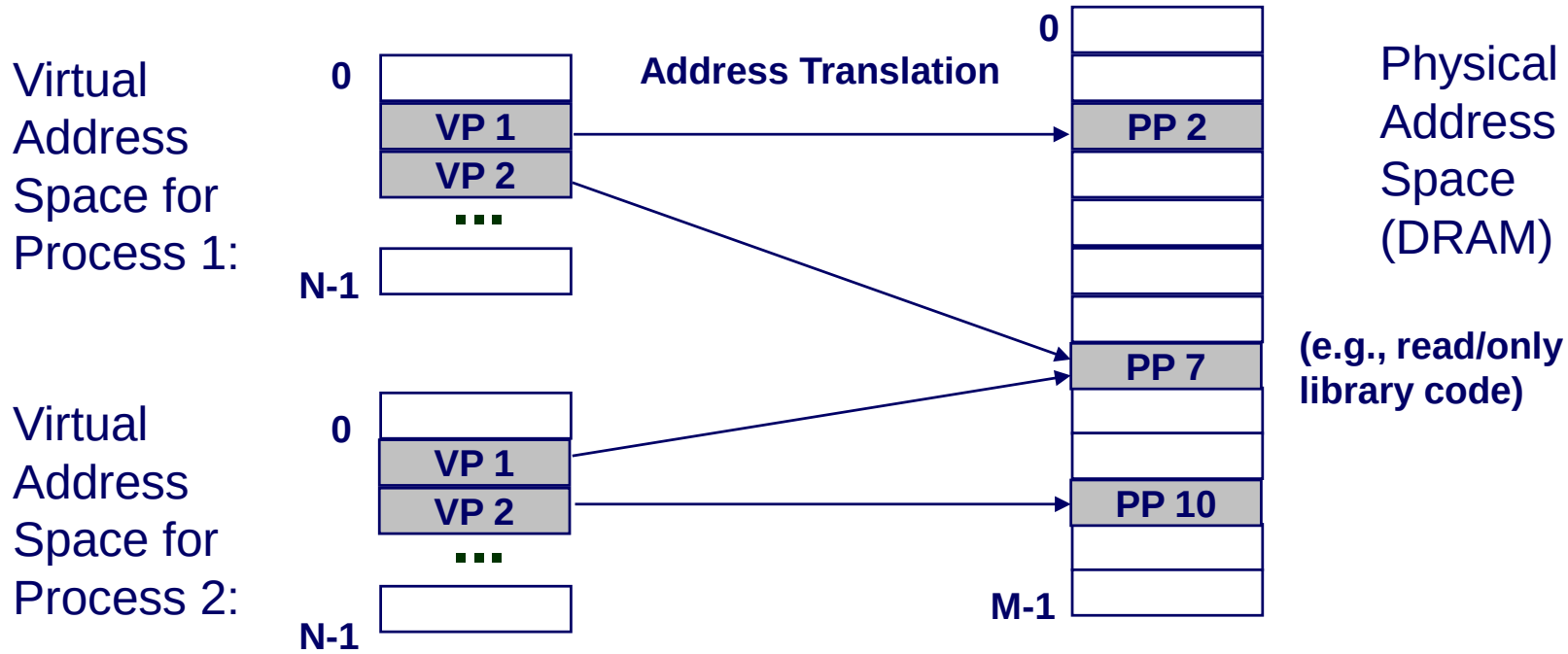
Linux x86
Speicherorganisation



Separate virtuelle Adressräume (cont.)

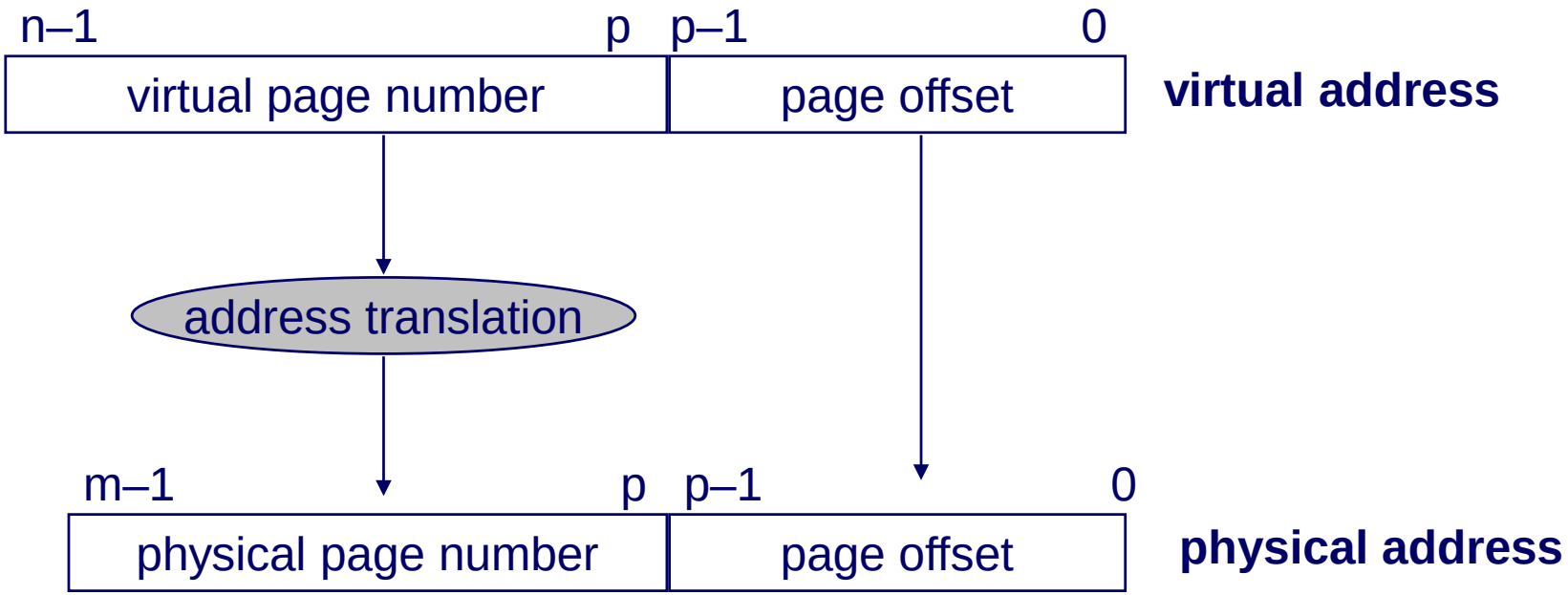
Auflösung der Adresskonflikte

- 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

Virtueller Adressraum

- V = {0, 1, ..., N-1}

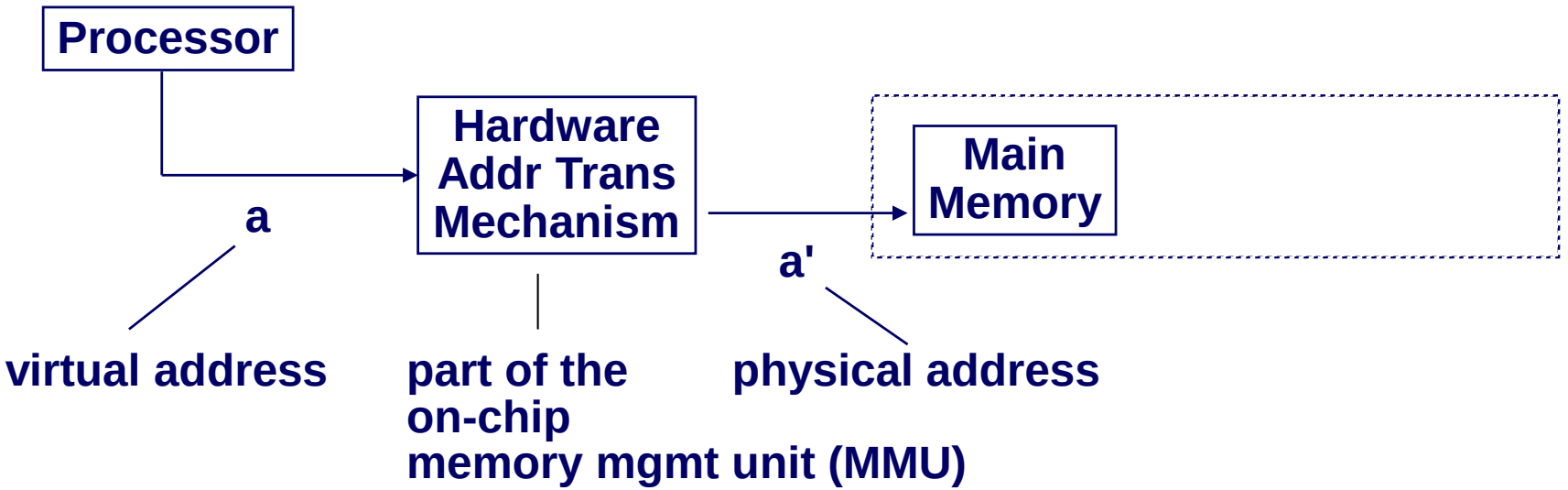
Physikalischer Adresssraum

- P = {0, 1, ..., M-1}
- meistens M<N

Adressumsetzung

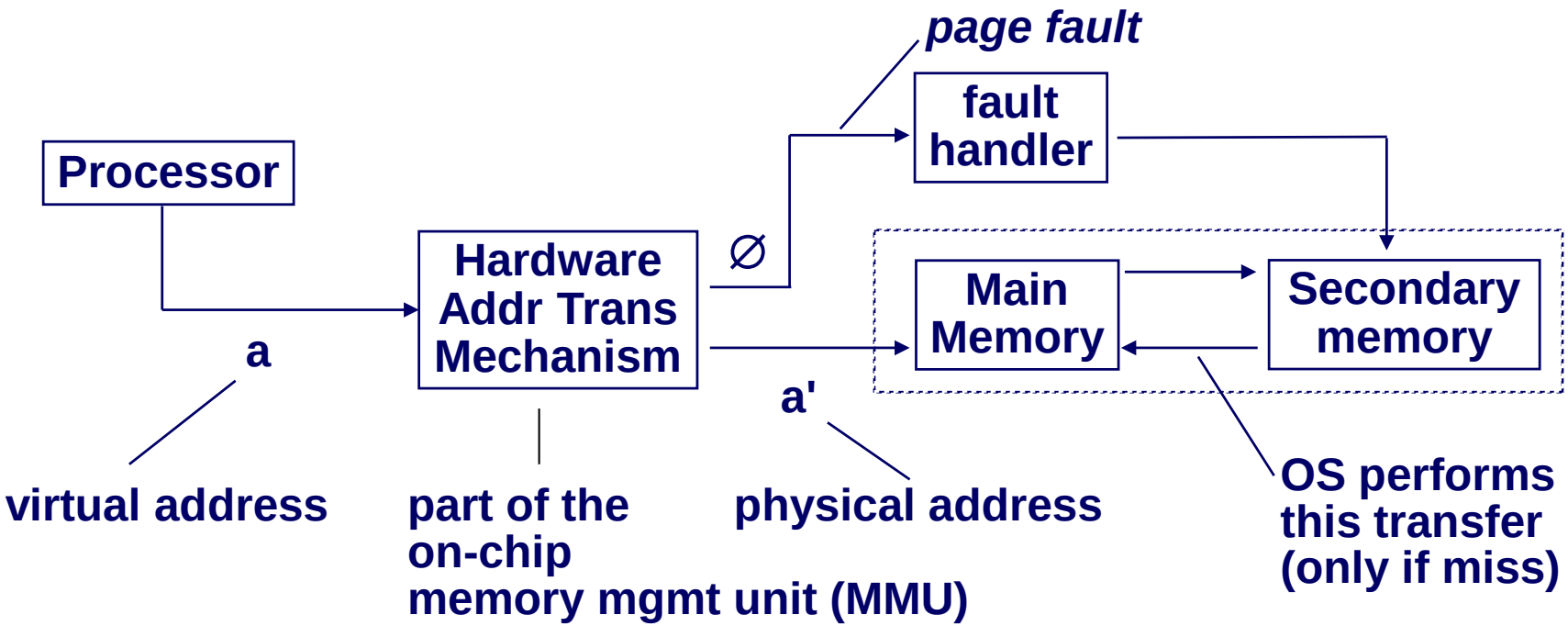
- MAP: $V \rightarrow P \cup \{\varnothing\}$
- Für eine virtuelle Adresse a:
  - MAP(a) = a', wenn Daten bei virtueller Adresse a und physikalischer Adresse a' in P sind
  - MAP(a) = $\varnothing$, wenn Daten bei virtueller Adresse a nicht im physikalischen Speicher sind, d. h. entweder ungültig oder nur auf Festplatte gespeichert sind.
- Realisierung von MAP mit einer „Seiten-Tabelle“

Virtueller Speicher: Hit



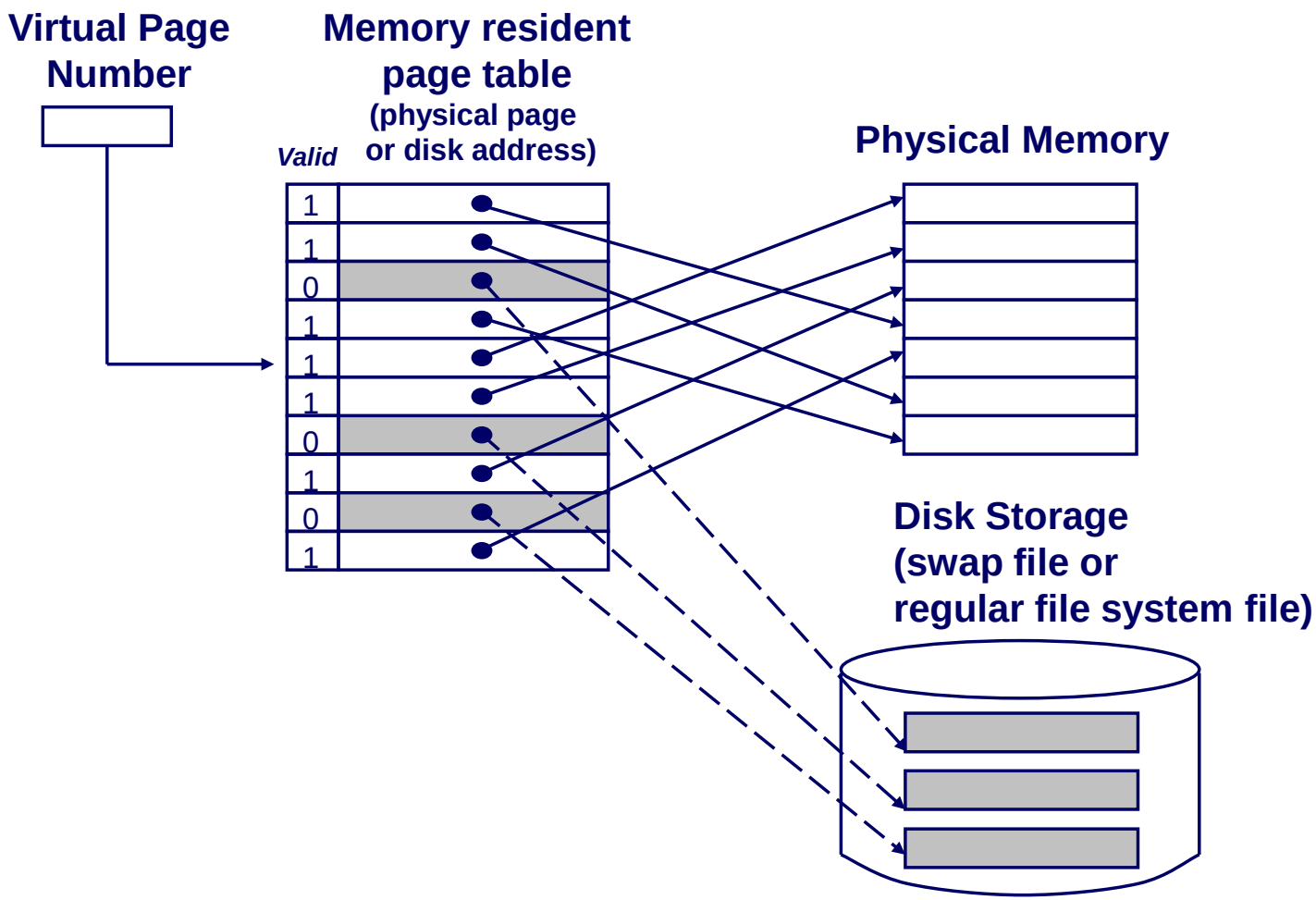
- Programm greift auf virtuelle Adresse a zu
- MMU überprüft den Zugriff, liefert physikalische Adresse a'
- Speicher liefert die zugehörigen Daten $d[a']$

Virtueller Speicher: Miss

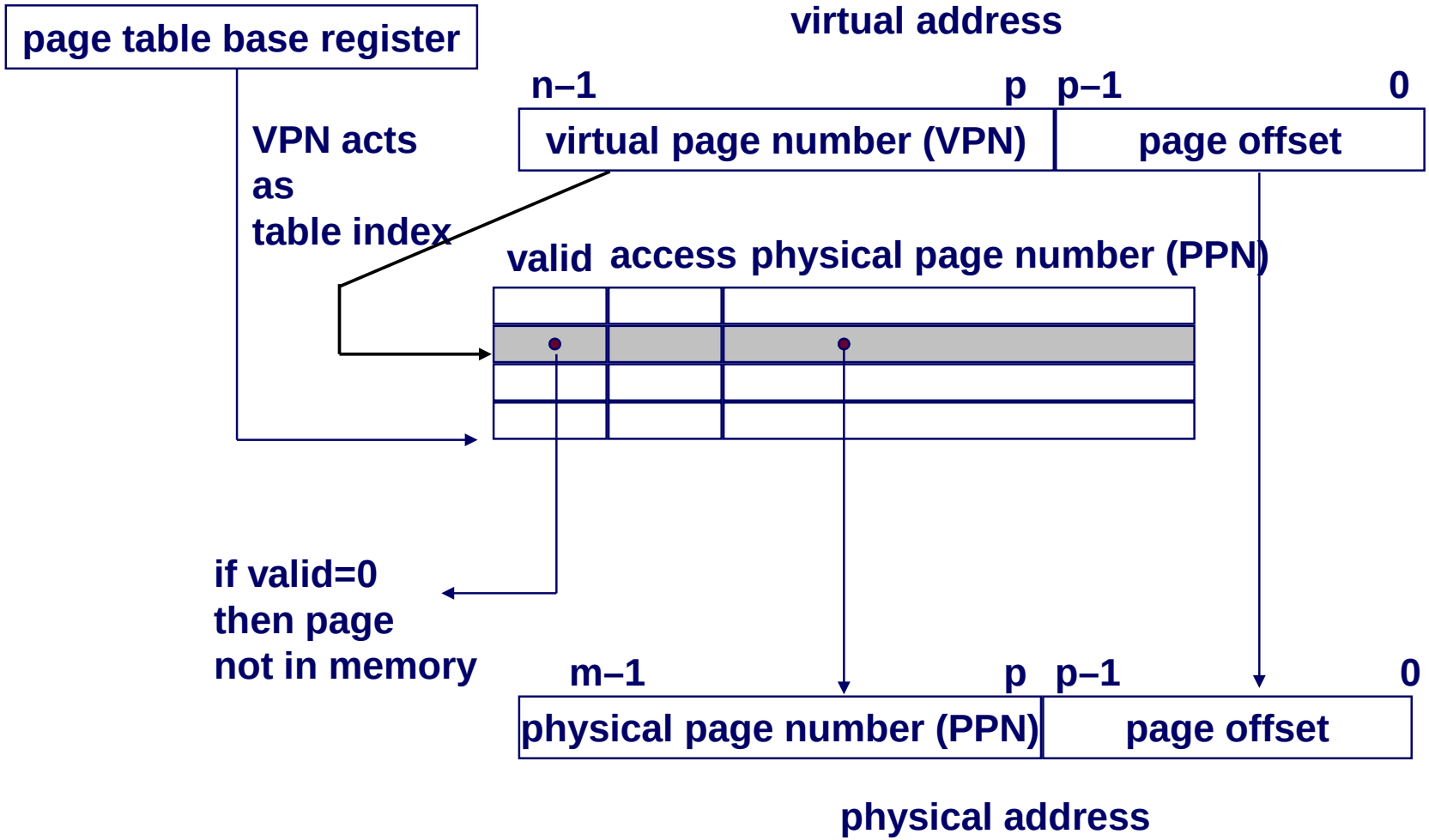


- Programm greift auf virtuelle Adresse *a* zu
- MMU überprüft den Zugriff, Adresse nicht in MAP
- „page-fault“ ausgelöst, Betriebssystem übernimmt

Seiten-Tabelle: Konzept

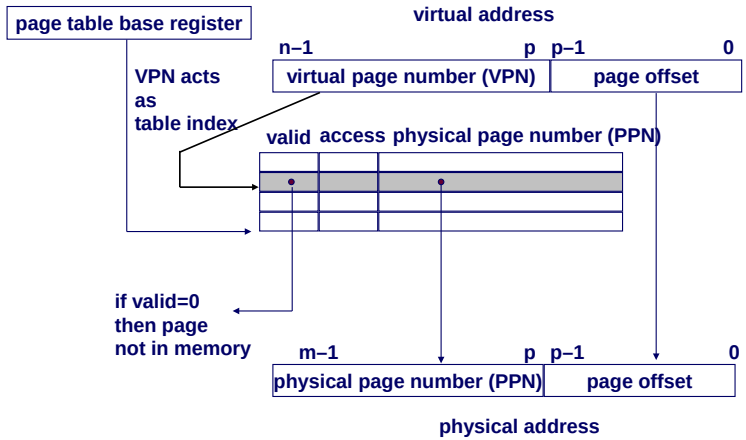


Seiten-Tabelle: Details



Adressumsetzung mit Seiten-Tabelle

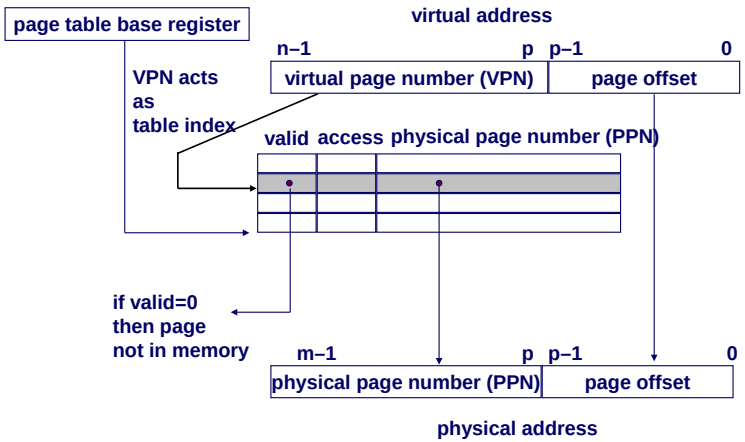
- „virtual page number“ VPN zeigt auf Seiten-Tabelleneintrag
- Seiten-Tabelleneintrag liefert Informationen über die Seite
  - valid-Bit = 1: die Seite ist im Speicher ⇒ benutze physikalische Seitennummer („Physical Page Number“) zur Adressberechnung
  - valid-Bit = 0: die Seite ist auf der Festplatte ⇒ Seitenfehler
- je eine Seitentabelle für jeden Prozess



Separate Seiten-Tabelle für jeden Prozess

je eine Seitentabelle für jeden Prozess

- spezielles Register („page table base register“) in der MMU
- Adresse der Seitentabelle des aktiven Prozesses
- wird vom Betriebssystem beim Prozesswechsel gesetzt



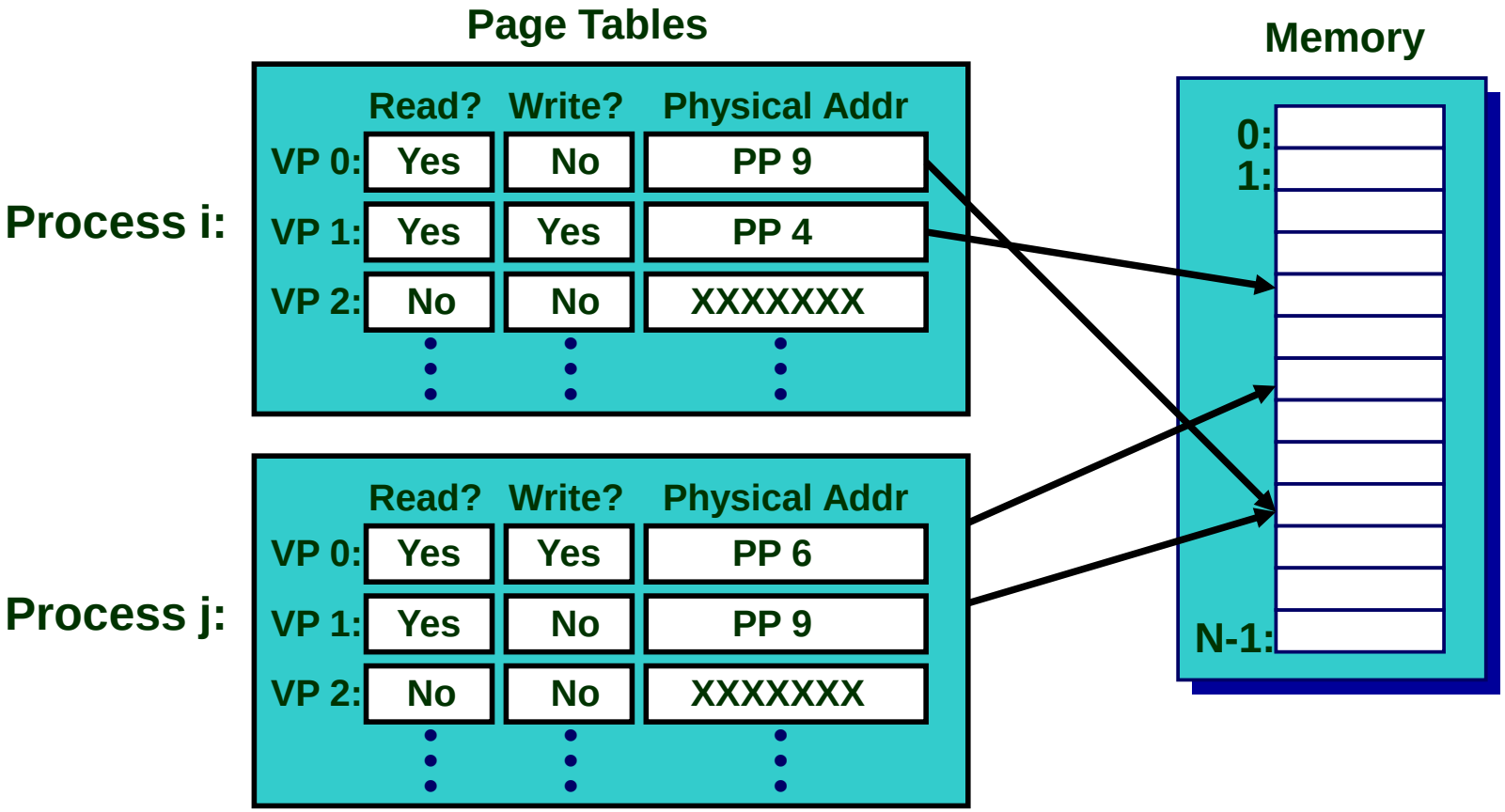
Verwaltung von Zugriffsrechten

Schutzüberprüfung

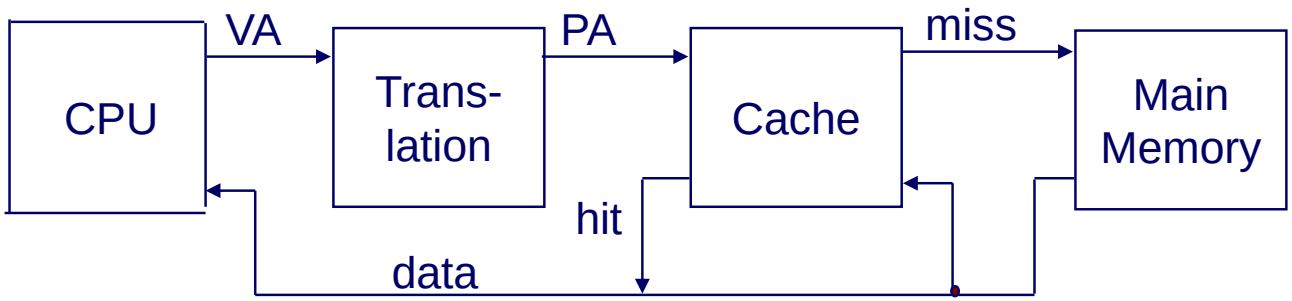
- zusätzliche Einträge (bits) in der Seitentabelle
  - typischerweise werden zahlreiche Schutzmodi unterstützt
  - Unterscheidung zwischen Kernel- und User-Mode
  - z.B. read-only, read-write, execute-only, no-execute
  - no-execution Bits gesetzt für Stack-Pages: Erschwerung von Buffer-Overflow-Exploits

- Schutzrechteverletzung wenn Prozess/Benutzer nicht die nötigen Rechte hat
- bei Verstoß erzwingt die Hardware den Schutz durch das Betriebssystem („Trap“ / „Exception“)

Zugriffsrechte: Read/Write-Bits in der Seiten-Tabelle



Integration von virtuellem Speicher und Cache



Die meisten Caches werden *physikalisch adressiert*

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

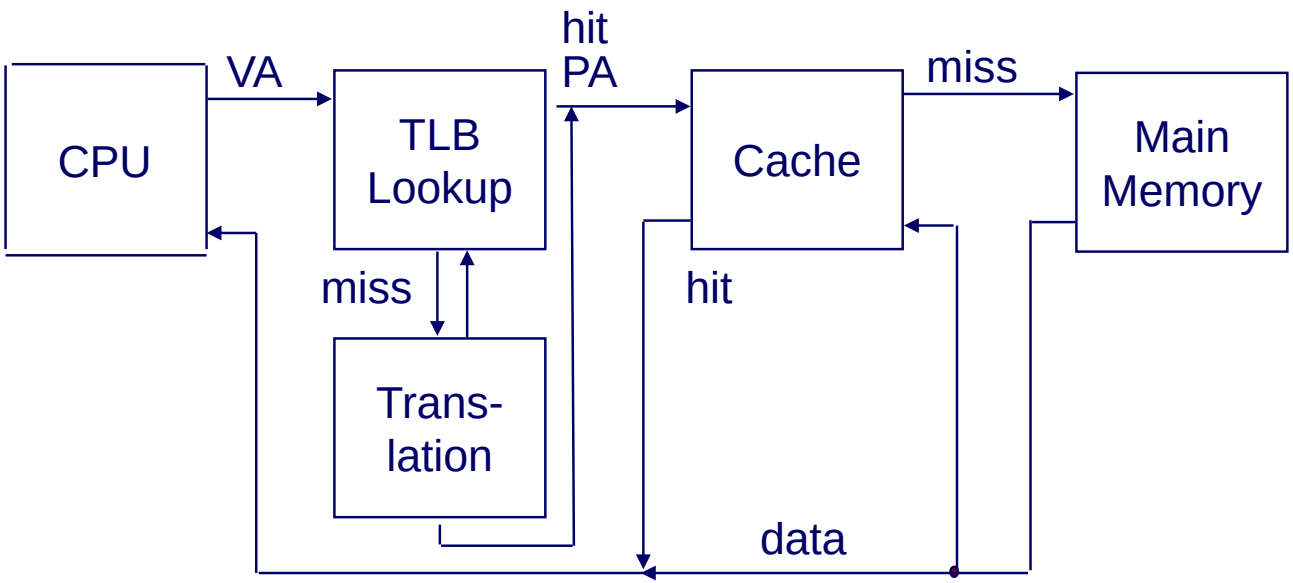
Die Adressumsetzung wird vor dem Cache „Lookup“ durchgeführt

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

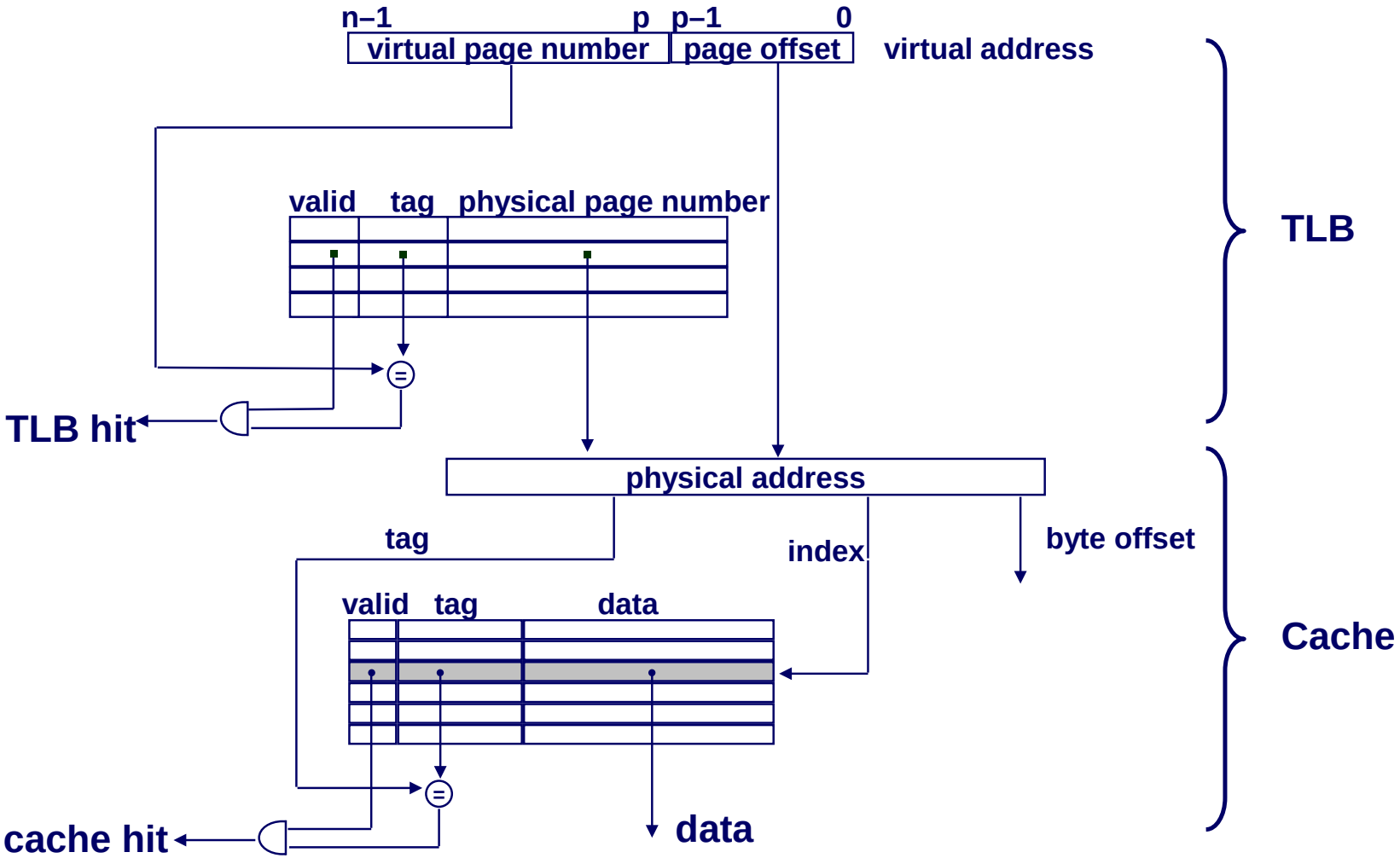
TLB / „Translation Lookaside Buffer“

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



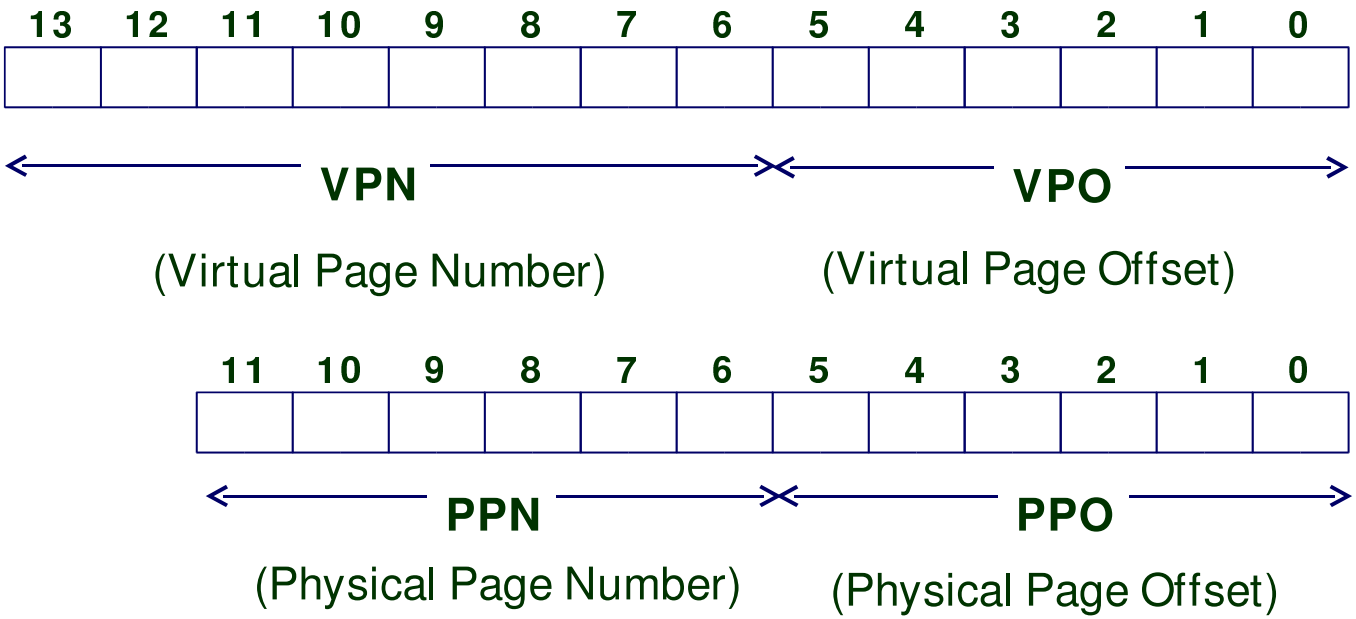
TLB / „Translation Lookaside Buffer“ (cont.)



Beispiel für einfaches Speichersystem

Adressierung

- 14-Bit virtuelle Adresse
- 12-Bit physikalische Adresse
- Seitengröße = 64 Bytes



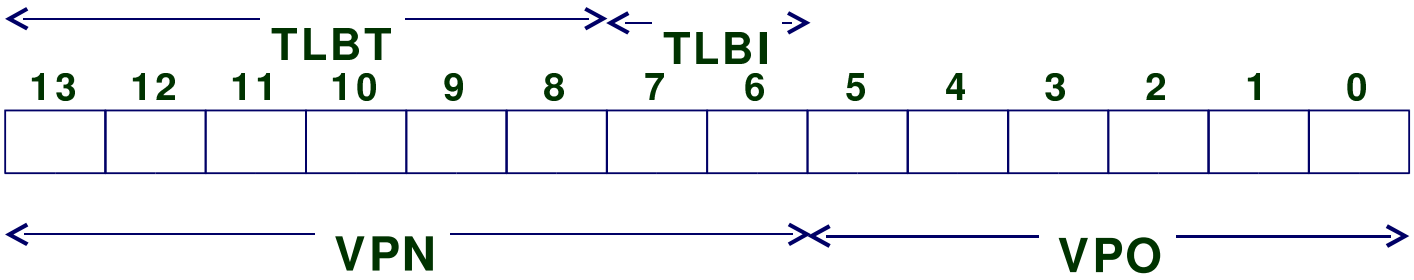
Einfaches Speichersystem: Seiten-Tabelle

- Nur die ersten 16 Einträge werden gezeigt

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Einfaches Speichersystem: TLB

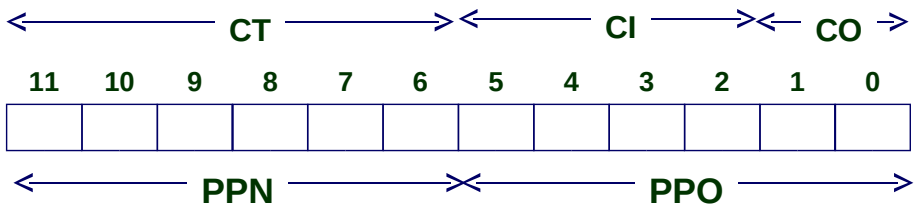
- 16 Einträge
- 4-fach assoziativ



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Einfaches Speichersystem: Cache

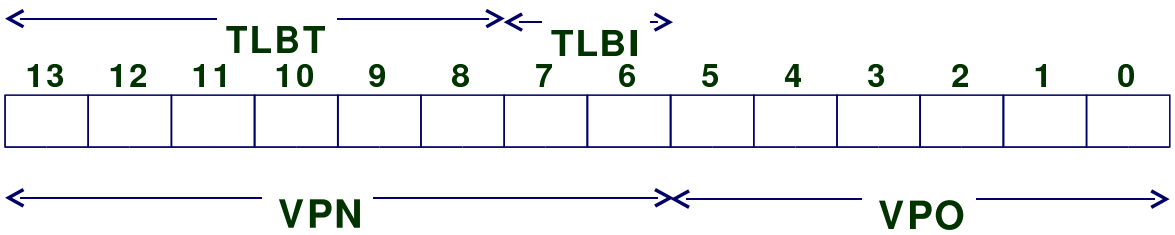
- 16 Zeilen
- 4-byte pro Zeile (line size)
- Direkt abgebildet



Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

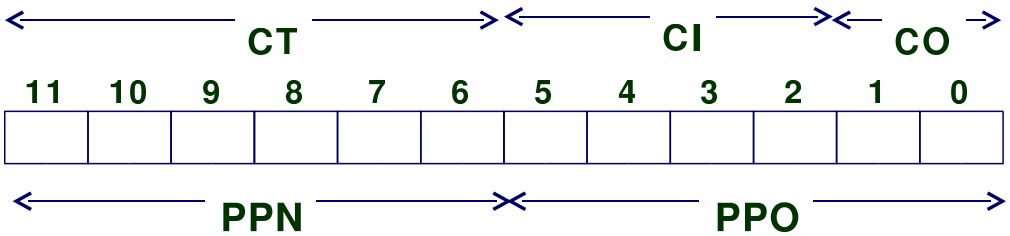
Adressumsetzungsbeispiel Nr. 1

Virtual Address `0x03D4`



VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

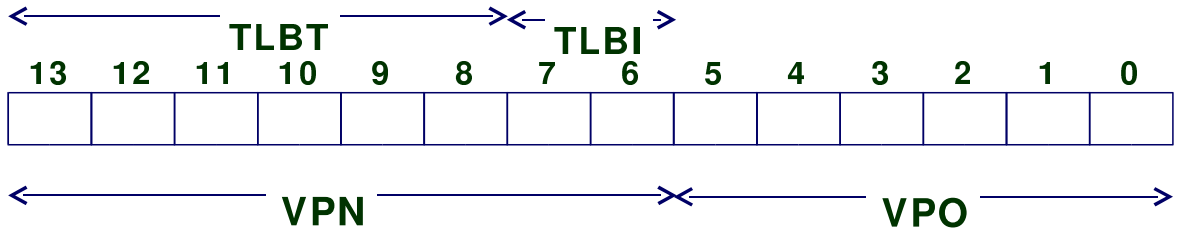
Physical Address



Offset ___ CI___ CT ____ Hit? __ Byte: ____

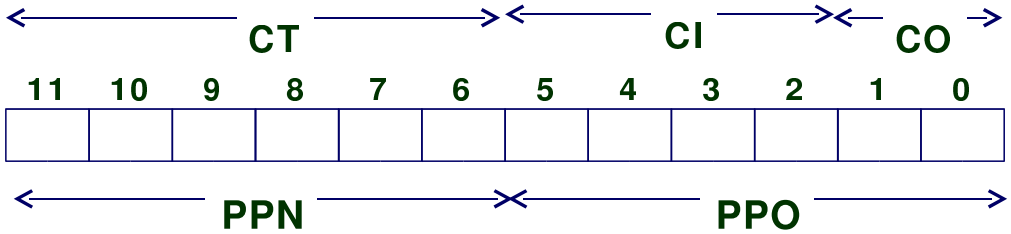
Adressumsetzungsbeispiel Nr. 2

Virtual Address `0x0B8F`



VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

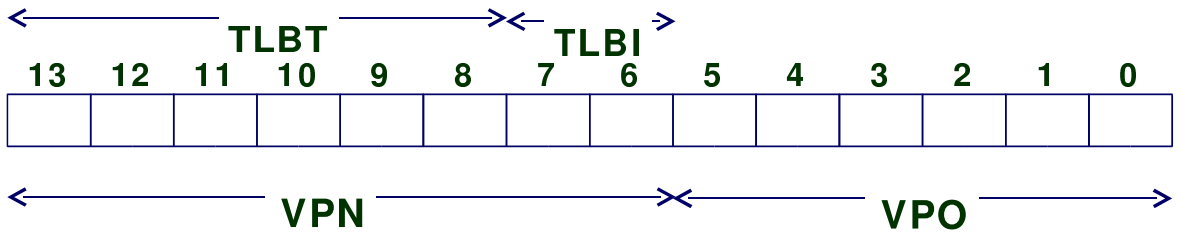
Physical Address



Offset ___ CI___ CT ____ Hit? __ Byte: ____

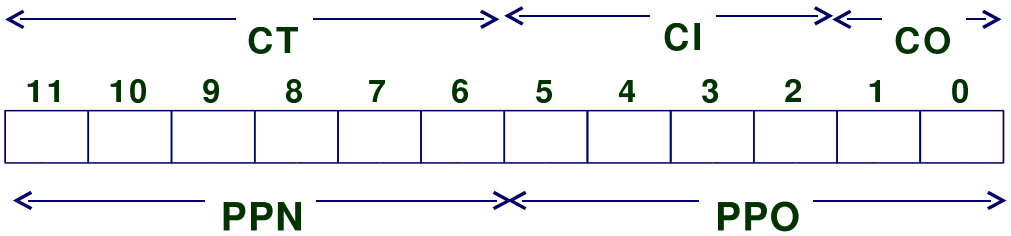
Adressumsetzungsbeispiel Nr. 3

Virtual Address 0x0040



VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

Physical Address



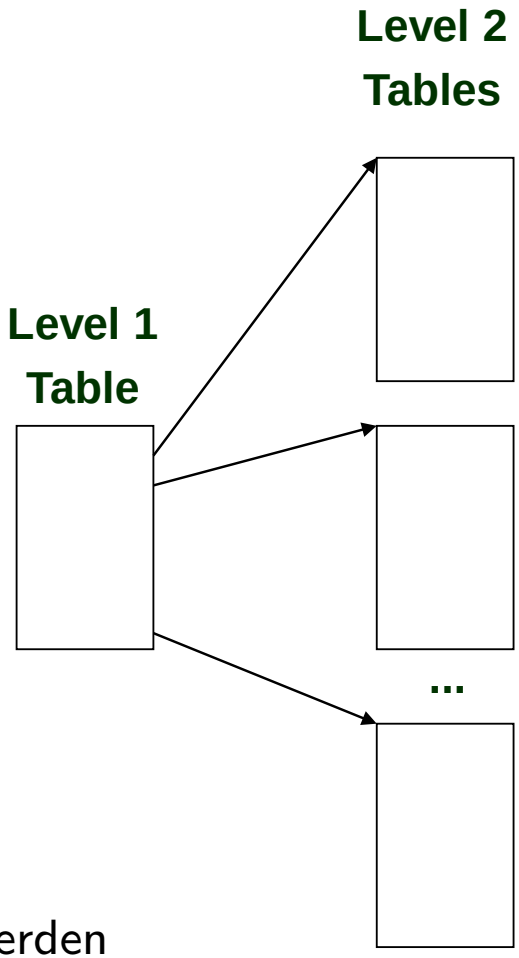
Offset ___ CI___ CT ____ Hit? __ Byte: ____

mehrstufige Seiten-Tabellen

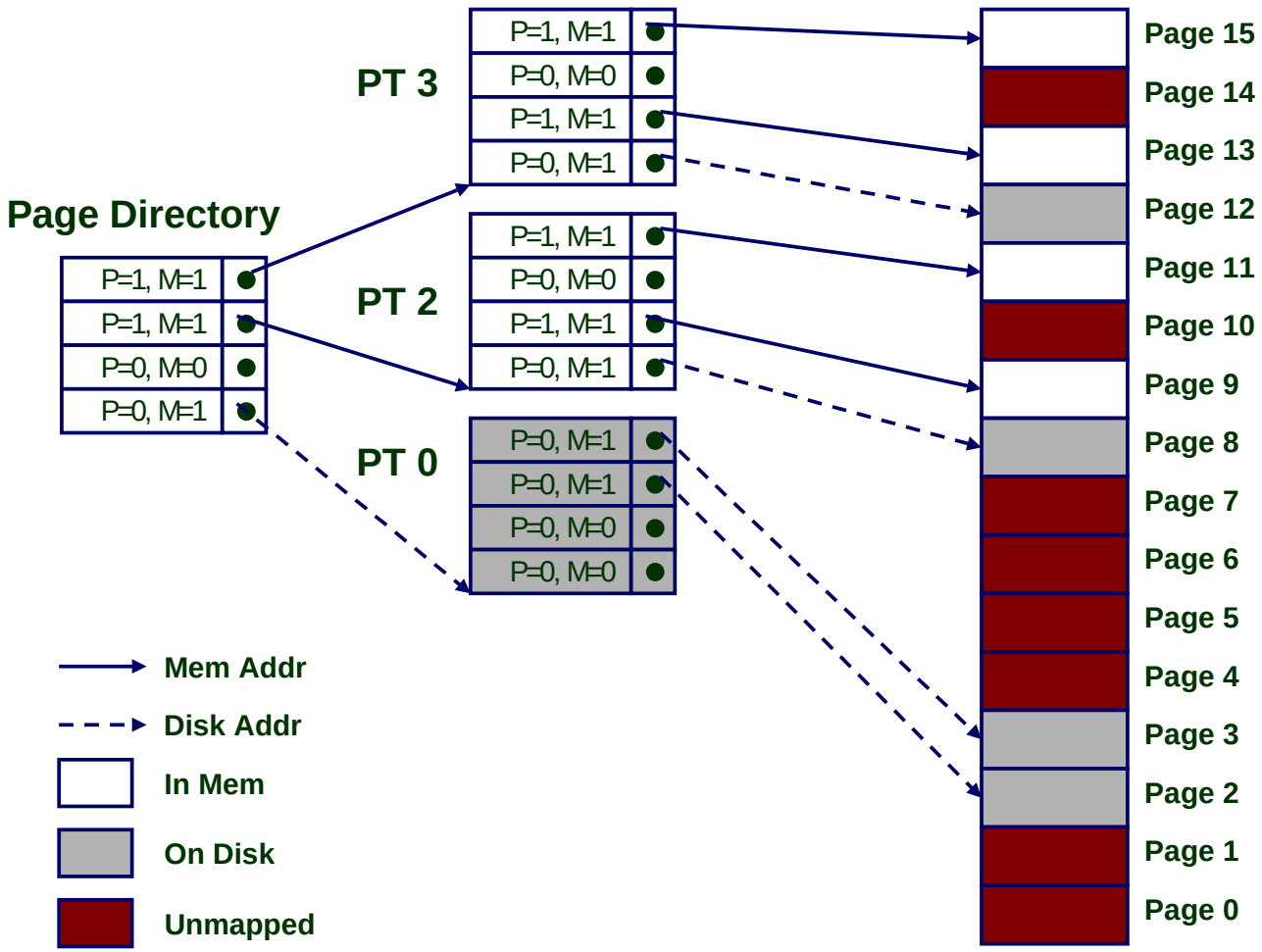
- typische Werte für 32-bit System
  - 4 kB (2^{12}) Seitengröße
  - 32-Bit Adressraum
  - 4-Byte PTE („Page Table Entry“) Seitentabelleneintrag
- Problem
  - würde 4 MB Seiten-Tabelle erfordern
  - 2^{20} Bytes (pro Prozess)

$\Rightarrow$ 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
  - Einträge können auf Festplatte ausgelagert werden

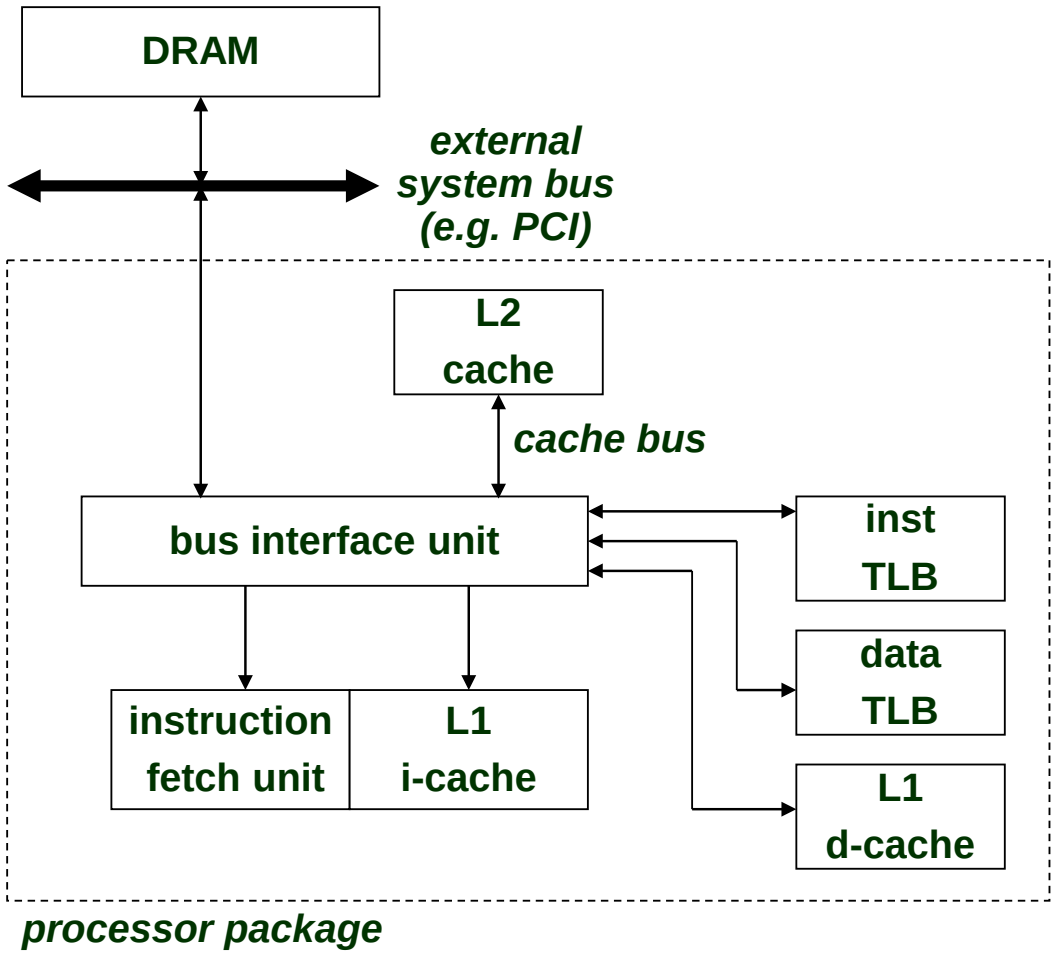


mehrstufige Seiten-Tabelle

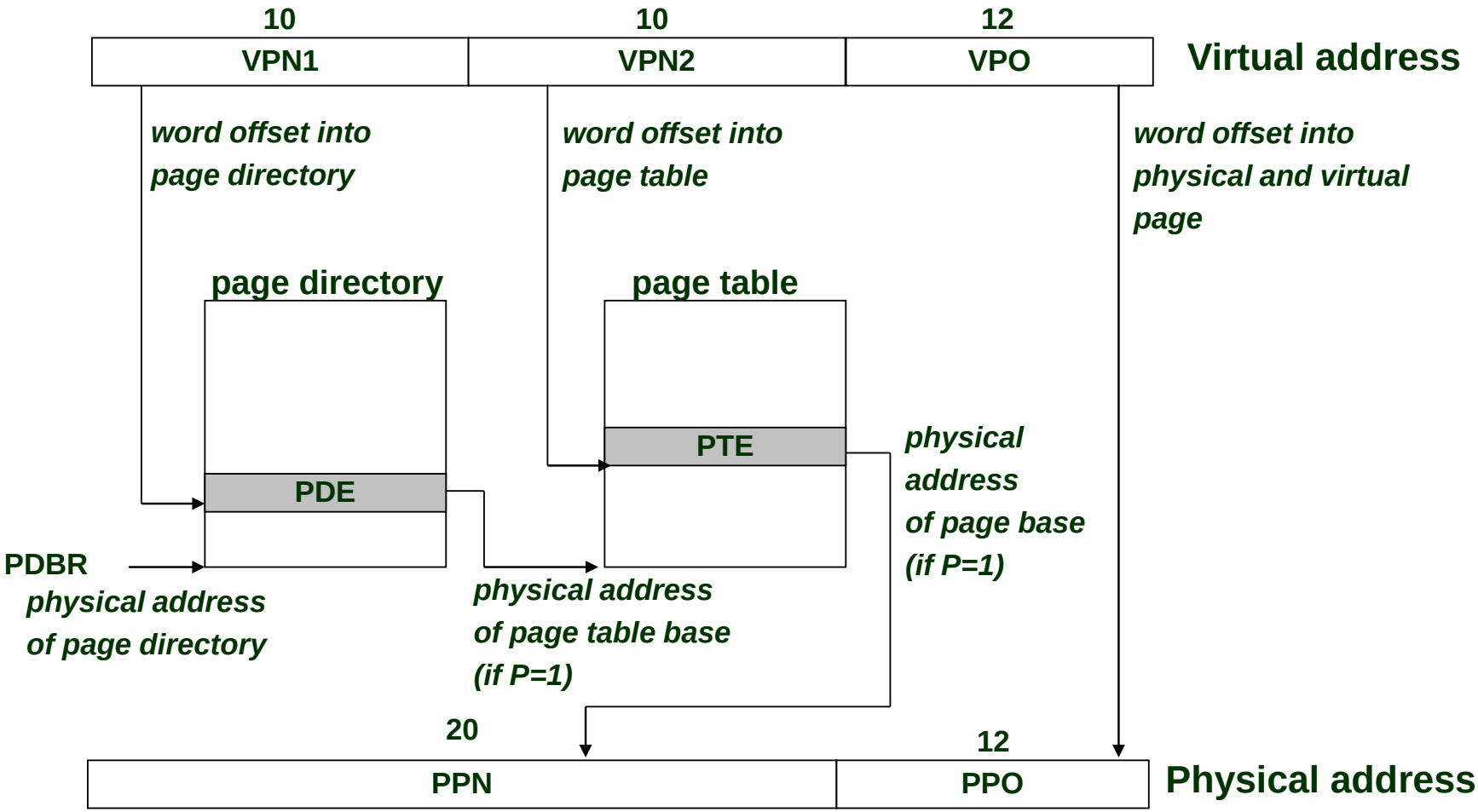


Beispiel: Pentium und Linux

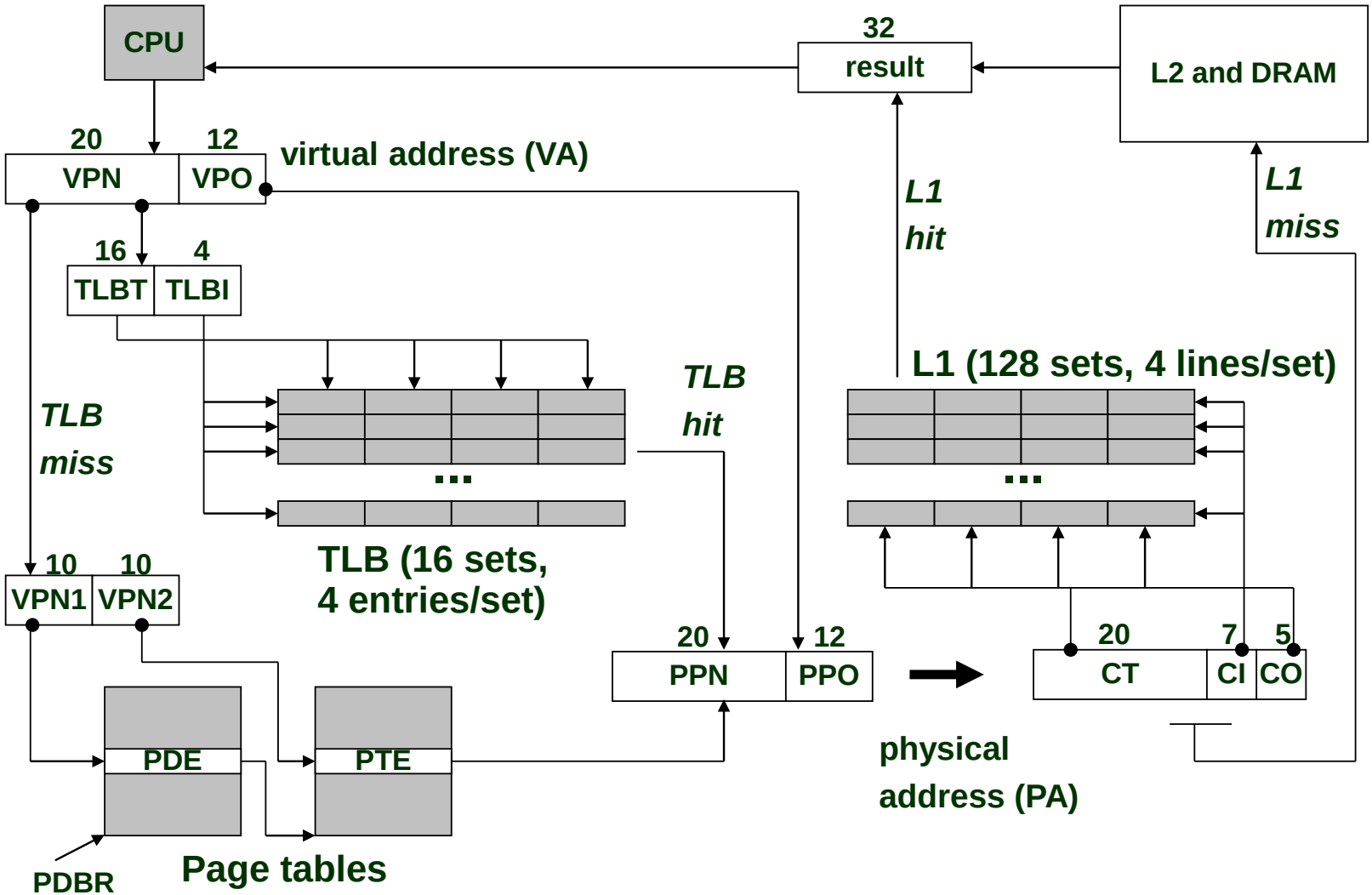
- 32-bit Adressraum
- 4 kB Seitengröße
- L1, L2 TLBs
  4fach assoziativ
- Instruktions TLB
  32 Einträge
  8 Sets
- Daten TLB
  64 Einträge
  16 Sets
- L1 I-Cache, D-Cache
  16 kB
  32 B Cacheline
  128 Sets
- L2 Cache
  Instr.+Daten zusammen
  128 kB . . . 2 MiB



Pentium und Linux: Seiten-Tabelle



Pentium und Linux: TLB und Cache



Zusammenfassung: Speicherhierarchie

Cache Speicher

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

Virtueller Speicher

- Nutzung der Festplatte als erweiterter Hauptspeicher
- ermöglicht viele Funktionen des Betriebssystems
- mehrere Prozesse, Schutzmechanismen
- Implementierung mit Hardware und Software

Zusammenfassung: 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: Virtueller Speicher

Sicht des Programmierers

- großer „flacher“ Adressraum
- Programm „besitzt“ die gesamte Maschine
  - hat privaten Adressraum
  - bleibt unberührt vom Verhalten anderer Prozesse

Sicht des Systems

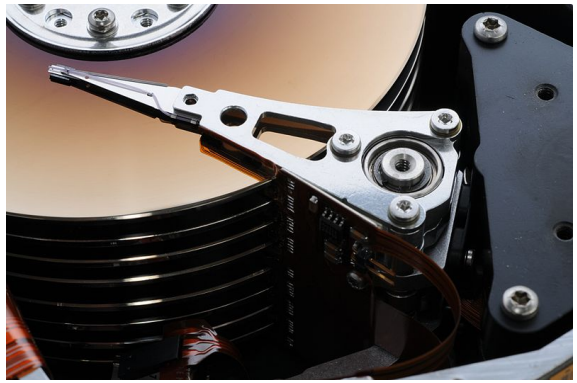
- Adressraum von Prozessen auf Seiten abgebildet
  - muss nicht fortlaufend sein
  - wird dynamisch zugeteilt
  - erzwingt Schutz bei Adressumsetzung
- Betriebssystem verwaltet viele Prozesse gleichzeitig
  - jederzeit schneller Wechsel zwischen Prozessen
  - u.a. beim Warten auf Ressourcen (Seitenfehler)

Festplatten

- magnetischer Massenspeicher

  IBM „RAMAC 150“, 1956, 5 MB
  - nichtflüchtige Datenspeicherung
  - hohe Speicherkapazität
  - gutes Preis-Leistungsverhältnis
  - dominierende Technologie
  - langsamer, serieller Zugriff
  - Dateisystemcaching als Abhilfe

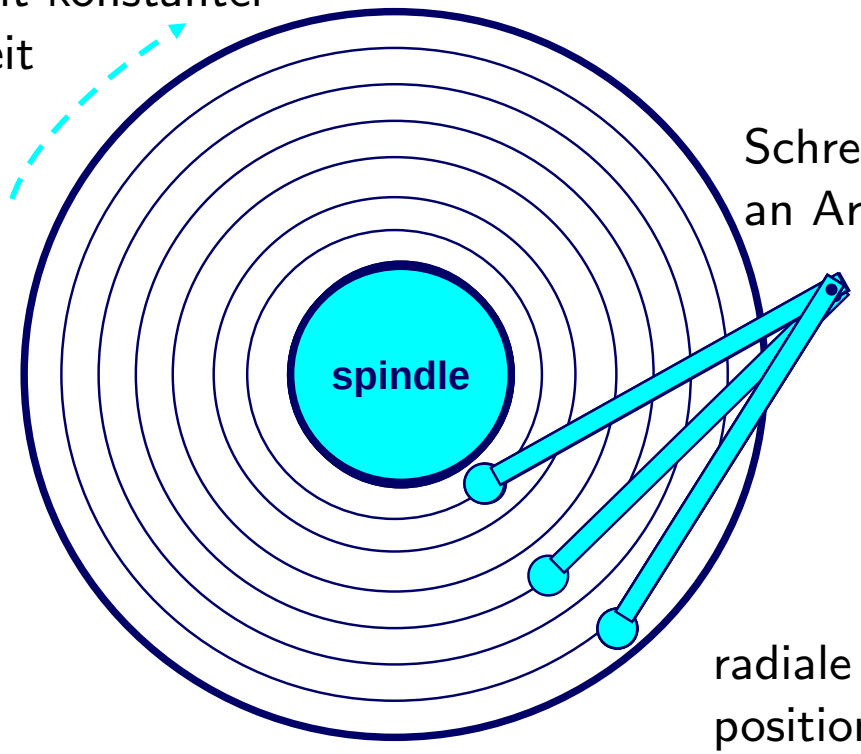
- Kenngrößen
  - Speicherkapazität, z.B. 2 TB
  - Dauertransferrate, z.B. 100 MB/s
  - mittlere Zugriffszeit, z.B. 18 ms
  - MTBF, z.B. 30.000 h



Festplatte: Prinzip

- Ansicht einer Platte

  Umdrehung mit konstanter Geschwindigkeit

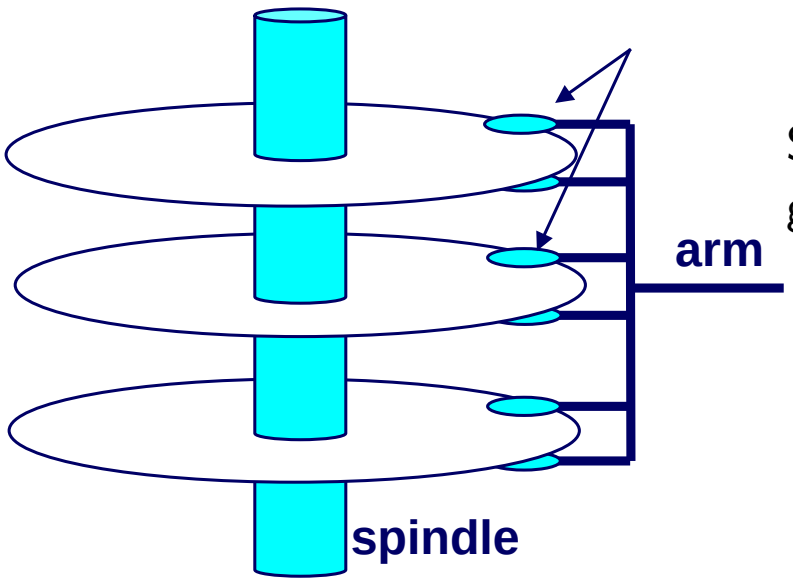


Schreib/Lese Kopf ist an Arm befestigt

„fliegt“ auf Luftpolster über Plattenoberfläche

radiale Bewegung des Arms positioniert Kopf auf Spuren

Festplatte: mehrere Schreib/Lese-Köpfe

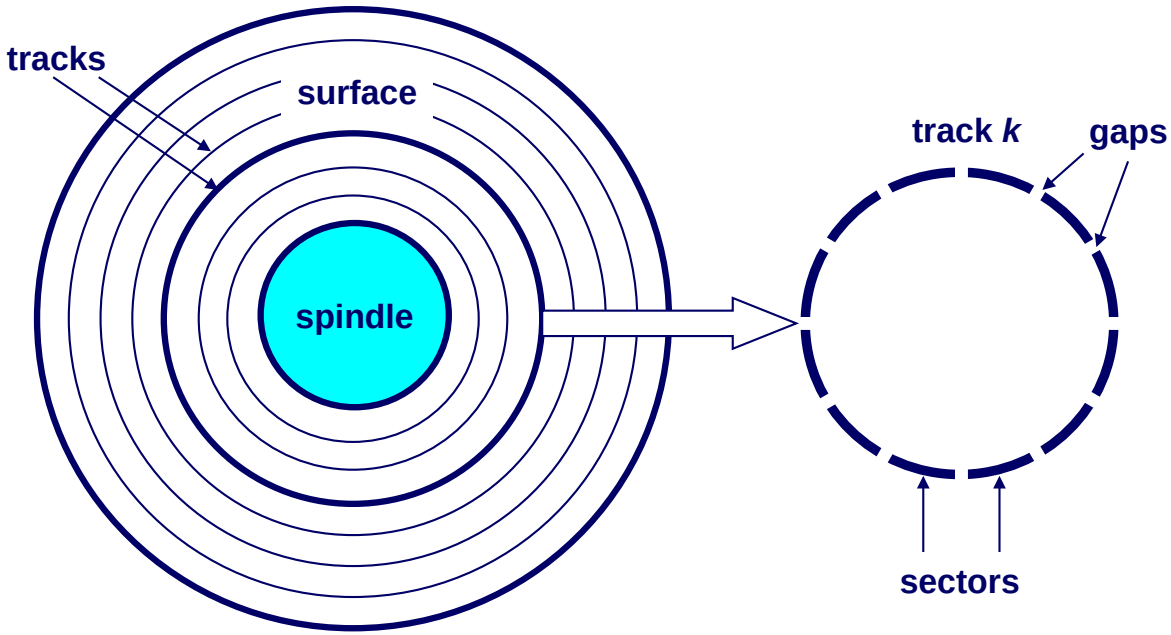


Schreib/Lese Köpfe werden gemeinsam auf „Zylindern“ positioniert

- Kapazität und Kosten proportional zur Anzahl der Platten
- günstige Festplatten: eine Platte, zwei Köpfe (oben/unten)

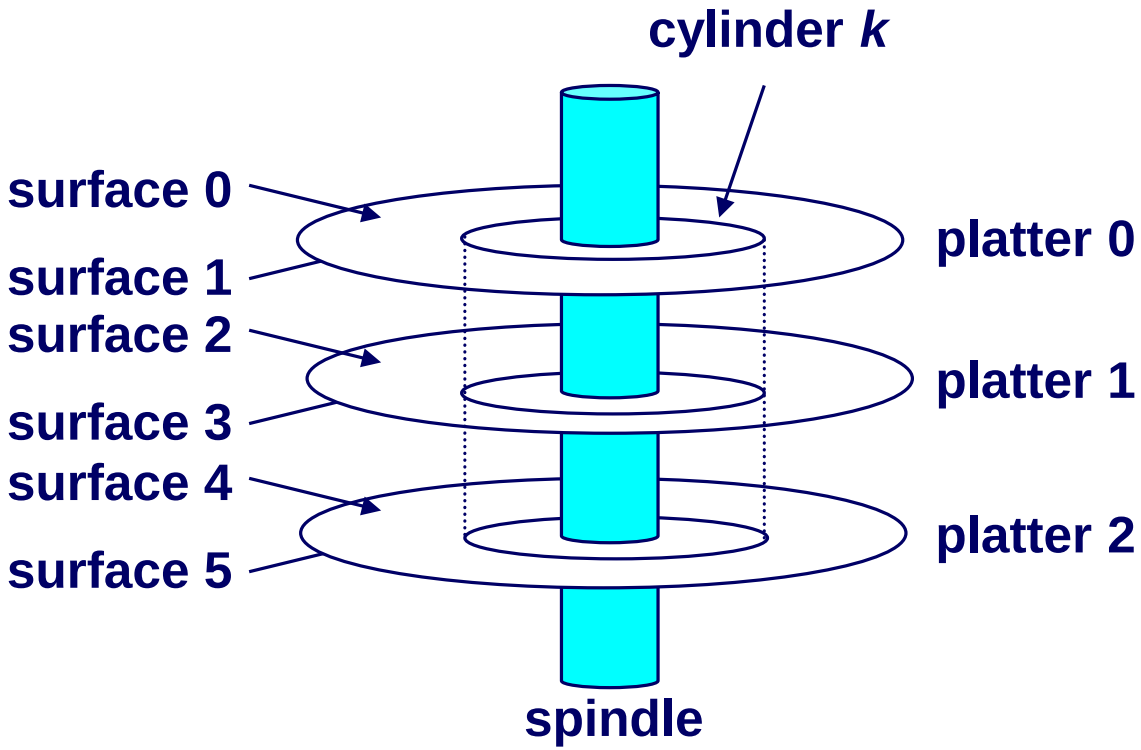
Festplattengeometrie

- 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



Festplattengeometrie (Ansicht mehrerer Platten)

- untereinander liegende Spuren (mehrerer Platten) bilden logisch einen „Zylinder“: gleiche Zugriffsmuster



Festplattenkapazität

- Kapazität: Höchstzahl speicherbarer Bits
  - typische Kapazität ca. 1 TB (hier: $1\,\text{TB} = 10^{12}\,Byte$) (2013)
- technologischen Faktoren
  - Aufnahmedichte [bits/in]: bits pro 1-inch Segment einer Spur
  - Spurdichte [tracks/in]: Anzahl der Spuren pro 1-inch Radius
  - Flächendichte [bits/in^2]: Produkt aus Aufnahme- und Spurdichte
  - limitiert durch minimal noch detektierbare Magnetisierung
  - sowie durch Positionierungsgenauigkeit der Köpfe
- Spuren in getrennte Aufnahmezonen („recording zones“) unterteilt
  - jede Spur einer Zone hat gleichviele Sektoren (festgelegt durch die Ausdehnung der innersten Spur)
  - Zone haben unterschiedlich viele Sektoren/Spuren

Berechnen der Festplattenkapazität

- Kapazität = Bytes/Sektor × mittlere Anzahl Sektoren/Spur × Spuren/Oberfläche × Oberflächen/Platten × Platten/Festplatte
- Beispiel
  - 512 Bytes/Sektor
  - 300 Sektoren/Spuren (im Durchschnitt)
  - 20 000 Spuren/Oberfläche
  - 2 Oberflächen/Platten
  - 5 Platten/Festplatte

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

$\Rightarrow$ uraltes Modell

Festplatten-Zugriffszeit

Durchschnittliche (avg) Zugriffszeit auf einen Zielsektor wird angenähert durch:

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

Suchzeit ($T_{avgSuche}$)

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

Festplatten-Zugriffszeit (cont.)

Rotationslatenzzeit ($T_{avgRotation}$)

- Wartezeit, bis das erste Bit des Zielsektors unter dem r/w Schreib-Lese-Kopf durchrotiert.
- $T_{avgRotation} = 1/2 \times 1/\text{RPMs} \times 60\ \text{Sek}/1\ \text{Min}$
- typische Drehzahlen, „rotations per minute“, sind 5400 .. 7200 .. 10000 .. 15000 RPM (desktop .. server)

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

Transferzeit ($T_{avgTransfer}$)

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

Beispiel für Festplatten-Zugriffszeit

Beispiel:

- Umdrehungszahl = 7.200 RPM
- Durchschnittliche Suchzeit = 9 ms.
- mittlere Anzahl der Sektoren pro Spur = 400

eingesetzt

- $T_{avgRotation} = 1/2 \times (60\,s/7200\,RPM) \times 1000\,ms/s = 4\ ms$
- $T_{avgTransfer} = 60/7200\,RPM \times 1/400\,s/sektor \times 1000\ ms/s = 0.02\ ms$
- $T_{avgZugriff} = 9\ ms + 4\ ms + 0.02\ ms \approx 13\,ms$

Festplatten-Zugriffzeit: wichtige Punkte

- Zugriffszeit von Suchzeit und Rotationslatenzzeit dominiert
- erstes Bit eines Sektors ist das „teuerste“, der Rest ist danach 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
  - Zugriff auf Festplatte ist um Faktor 13 ms / 4 ns bzw. $\approx 3\,000\,000\times$ langsamer als SRAM
  - ensprechend $200\,000\times$ langsamer als DRAM

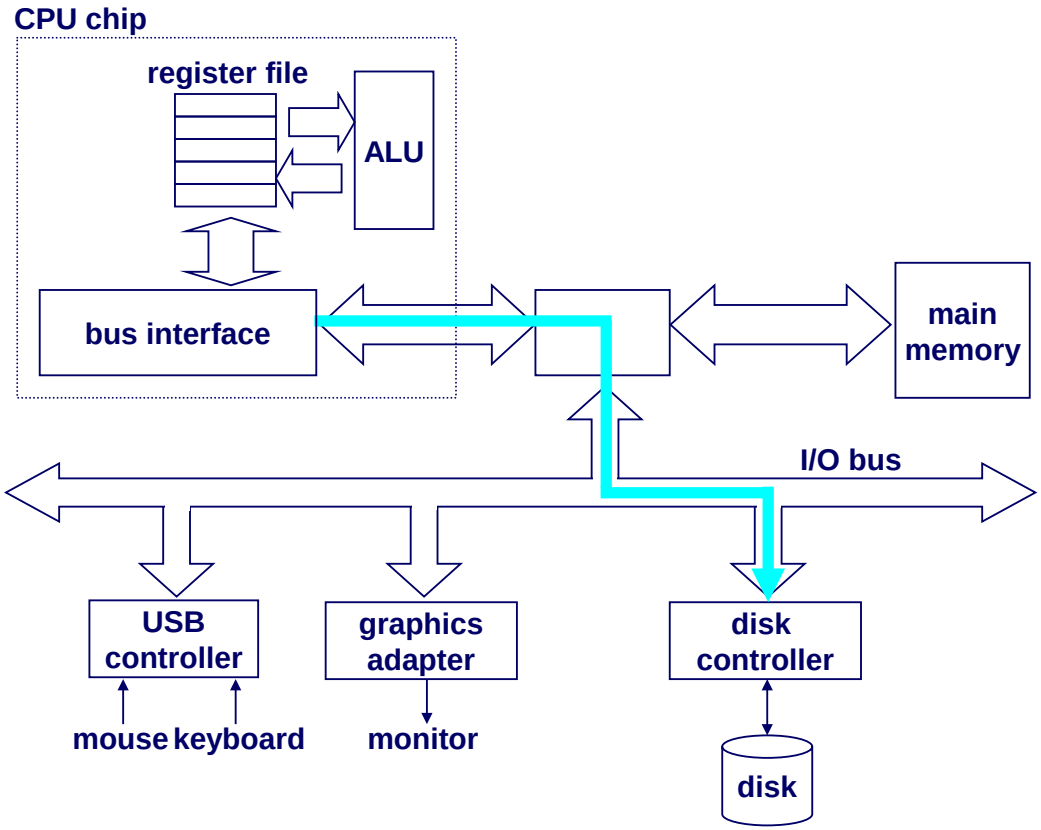
$\Rightarrow$ hoher Aufwand in Hardware und Betriebssystem, um dieses Problem (weitgehend) zu vermeiden

Logische Festplattenblöcke

- Benutzersicht der komplexen Sektorengeometrie:
  - Menge verfügbarer Sektoren wird als eine 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 zwischen logischen Blöcken und tatsächlichen physikalischen Sektoren
  - ausgeführt durch den Festplattencontroller
  - konvertiert logische Blöcke zu Tripeln (Oberfläche, Spur, Sektor)
- Controller reserviert ggf. unbenutze Bereiche als Ersatzzylinder
  - Unterschied zwischen „formatierter-“ und „maximaler Kapazität“

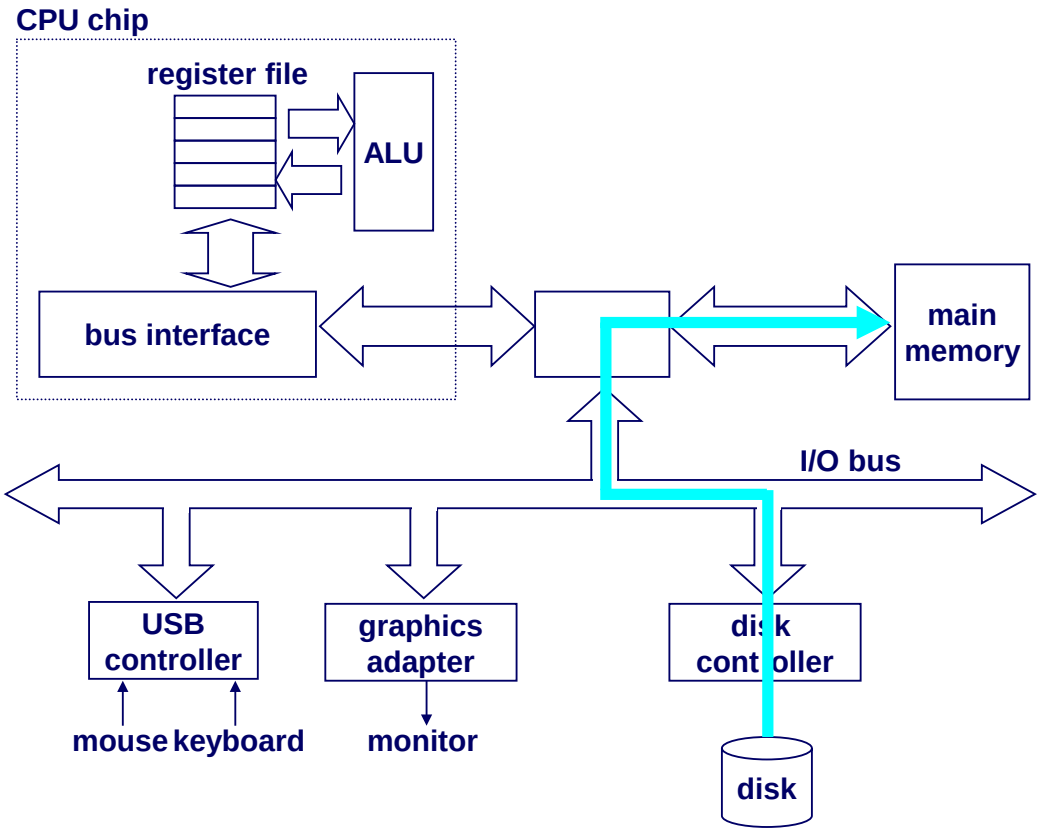
Lesen eines Festplattensektors

- CPU übergibt Parameter an den Festplattencontroller
  - Befehl, logische Blocknummer, Zielspeicheradresse
  - Zugriff auf den Port (I/O-Adresse) des Controllers



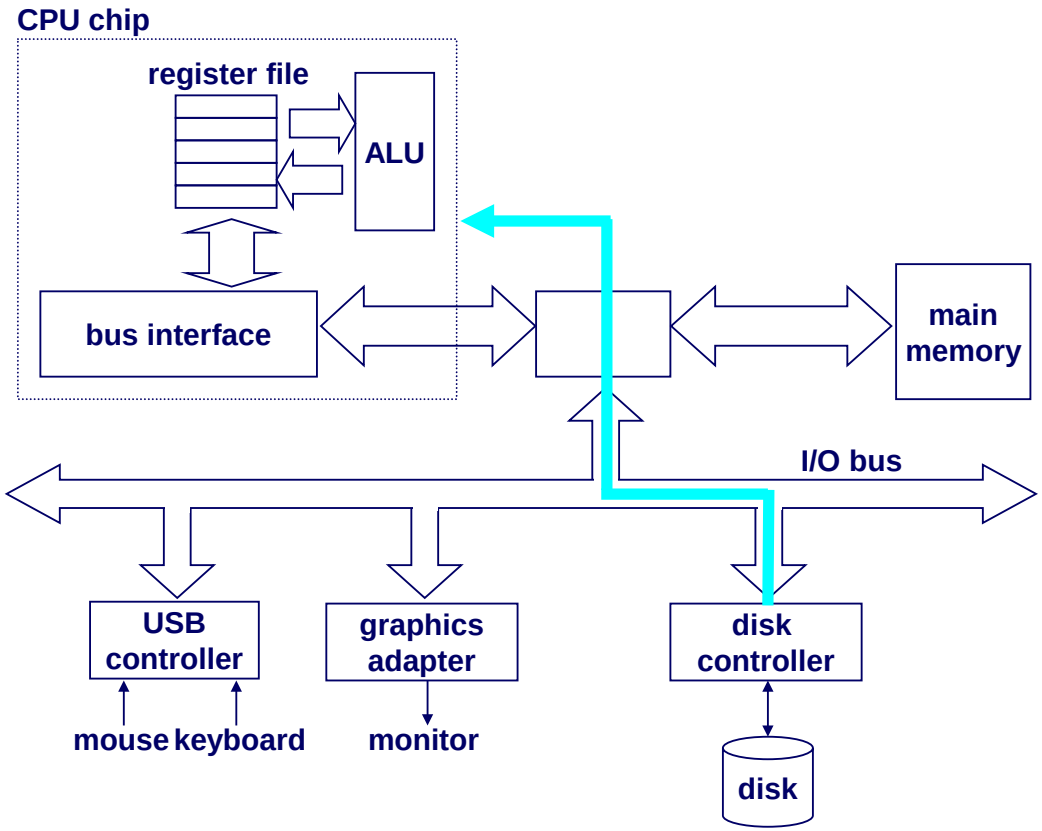
Lesen eines Festplattensektors (2)

- Festplattencontroller liest den gewünschten Sektor aus
- Festplattencontroller führt DMA-Zugriff auf Hauptspeicher aus



Lesen eines Festplattensektors (3)

- Festplattencontroller löst Interrupt aus



RAID: Redundant Array of Independent Disks

Aufteilung von Daten auf mehrere Festplatten?!

- bahnbrechende Untersuchung von Festplatten-Performance
- motiviert durch Analyse von Großrechner- und PC-Festplatten
- schnelles Speichersystem aus mehreren kleinen Festplatten (anstelle weniger großer teurer Platten)
- Zuverlässigkeit des Gesamtsystems?

- sieben RAID-Varianten („level“)
  - unterschiedliche Anzahl von Platten
  - Strategien zur Verwendung von Nutz- und Reserveplatten
  - Ausfallsicherheit, Hot-Plugging
  - Optimierung auf Schreib- und/oder Leseperformance
  - für Server vielfach eingesetzt

Originalpaper: Patterson, Gibson, Katz: UCB report CS-87-391)

RAID: Ausgangsbasis (1998)

Characteristics	*IBM 3380*	*Fujitsu M2361A*	*Conners CP3100*	*3380 v. CP3100*	*2361 v. CP3100*
				(>1 means 3100 better)	
Disk diameter (inches)	14	10.5	3.5	4	3
Formatted Data Capacity (MB)	7500	600	100	.01	.2
Price/MB(controller incl.)	$18-$10	$20-$17	$10-$7	1-2.5	1.7-3
MTTF Rated (hours)	30,000	20,000	30,000	1	1.5
MTTF in practice (hours)	100,000	?	?	?	?
No. Actuators	4	1	1	.2	1
Maximum I/O's/second/Actuator	50	40	30	.6	.8
Typical I/O's/second/Actuator	30	24	20	.7	.8
Maximum I/O's/second/box	200	40	30	.2	.8
Typical I/O's/second/box	120	24	20	.2	.8
Transfer Rate (MB/sec)	3	2.5	1	.3	.4
Power/box (W)	6,600	640	10†	660	64
Volume (cu. ft.)	24	3.4	.03	800	11

Table I. *Comparison of IBM 3380 disk model AK4 for mainframe computers, the Fujitsu M2361A "Super Eagle" disk for minicomputers, and the Conners Peripherals CP 3100 disk for personal computers. By "Maximum I/O's/second" we mean the maximum number of average seeks and average rotates for a single sector access. Cost and reliability information on the 3380 comes from widespread experience [IBM 87] [Gawlick87] and the information on the Fujitsu from the manual [Fujitsu 87], while some numbers on the new CP3100 are based on speculation. The price per megabyte is given as a range to allow for different prices for volume discount and different mark-up practices of the vendors.*

†The 8 watt maximum power of the CP3100 was increased to 10 watts to allow for the inefficiency of ar

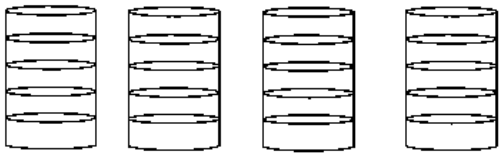
Zuverlässigkeit: Headcrash

Zerstörung der Magnetschicht durch Kontakt mit dem Lesekopf

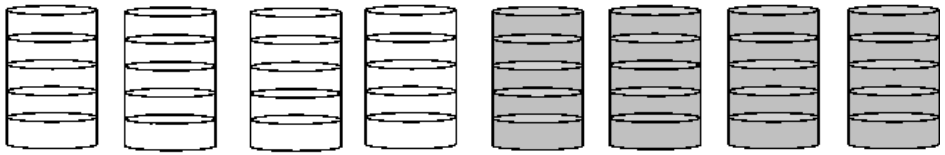
(wikimedia commons, Alchemist-hp, www.pse-mendelejew.de)

RAID: mehrere Varianten, „level 0..6“

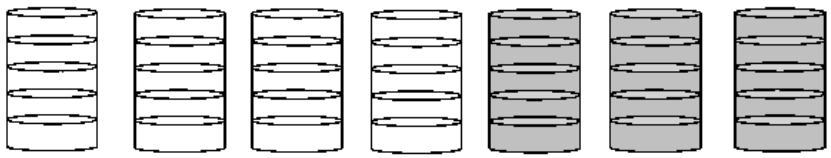
RAID 0: Keine Redundanz



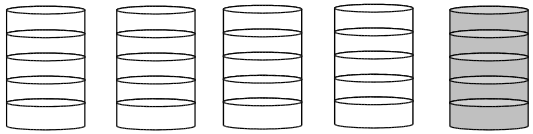
RAID 1: Spiegelung von Einzelplatten



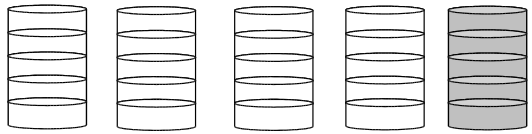
RAID 2: Einsatz fehlerkorrigierender Bitcodes



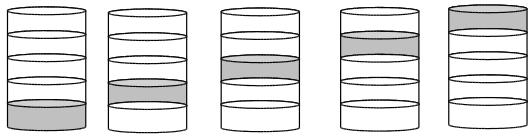
RAID 3: Bit-Parität



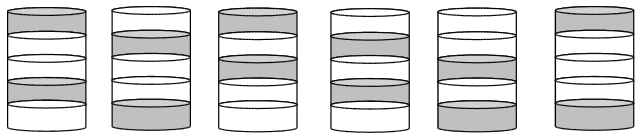
RAID 4: Block-Parität



RAID 5: Rotierende Block-Parität



RAID 6: Doppelte Redundanz



RAID 0: „Striping“

die Datenblöcke werden gleichmäßig auf zwei oder mehr unabhängige Platten verteilt

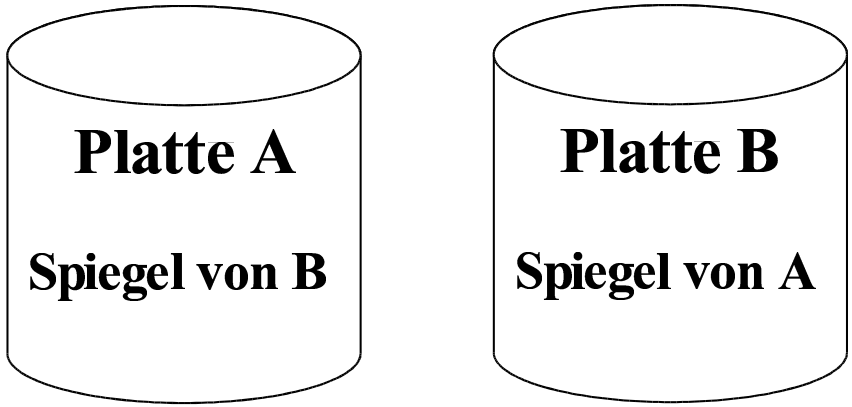
- gleichzeitiges Lesen Blöcke von mehreren Platten
- gleichzeitiges Schreiben auf mehrere Platten
- Performance wächst (linear) mit der Anzahl der Platten
- bei Ausfall einer Platte sind sämtliche Daten verloren
- Einsatz z.B. zwei Platten für schnellen Videoschnitt

Block 0	**Block 1**	**Block 2**	**Block 3**
Block 4	**Block 5**	**Block 6**	**Block 7**
Block 8	**Block 9**	**Block 10**	**Block 11**
Block 12	**Block 13**	**Block 14**	**Block 15**

RAID 1: „Mirroring“

jeder Platteninhalt ist zweimal vorhanden

- Lesezugriffe können auf beiden Platten verteilt werden
- Schreibzugriffe immer doppelt auf beide Platten
- bei Ausfall einer Platte nutzt man das Duplikat und erstellt eine neue Reserveplatte



RAID 2: fehlerkorrigierende Codes

- Kodieren aller Daten mit fehlerkorrigierendem Bitcode
- Originalbits und Prüfbits werden auf verschiedene Platten verteilt
  - Beispiel: (7,4)-Hamming-Code, mit 4+3 Platten
  - jeder Datenzugriff erfordert viele synchronisierte Plattenoperationen
  - vor jedem Schreibzugriff Berechnung der bitweisen Prüfbits
- insgesamt erscheint der Aufwand nicht gerechtfertigt

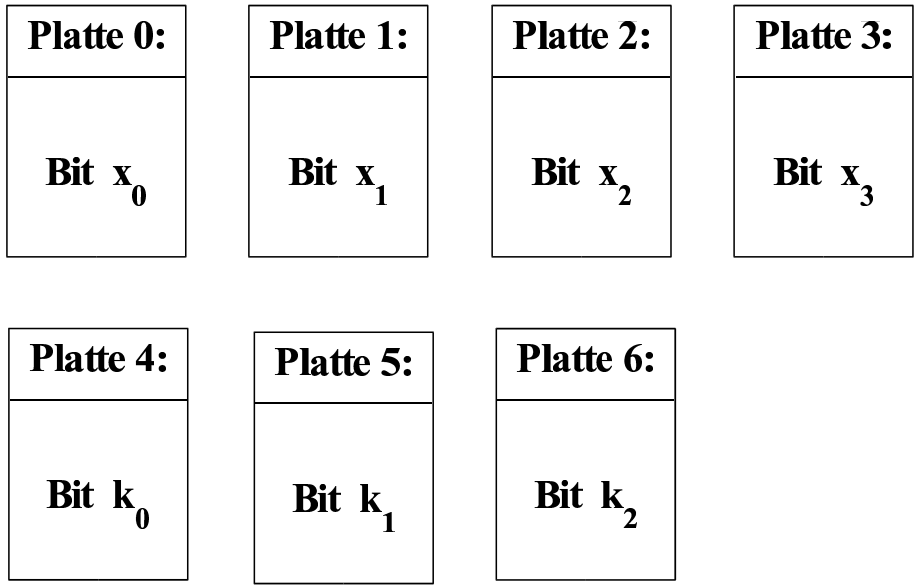
RAID 2: mit (7,4) Hamming-Code

Gleichungssystem:

$k2 = x2 \oplus x1 \oplus x0$

$k1 = x3 \oplus x1 \oplus x0$

$k0 = x3 \oplus x2 \oplus x0$



RAID 3: Bit-Parität

- mehrere Platten plus eine Paritätsplatte
  - dadurch Schutz gegen Totalausfall einer einzelnen Platte
  - Daten müssen bitweise berechnet werden

- Beispiel: Originaldaten auf vier Platten aufgeteilt, zusätzlich eine Paritätsplatte
  - Bildung der Modulo-2-Summe über die entsprechenden Bits, $p = x_0 \oplus x_1 \oplus x_2 \oplus x_3$

  - bei Ausfall einer der Platten 0 bis 4 (hier: Platte 1) Rekonstruktion des Inhalt mittels Modulo-2 Summe: $x_1 = x_0 \oplus x_2 \oplus x_3 \oplus x_4 \oplus p$

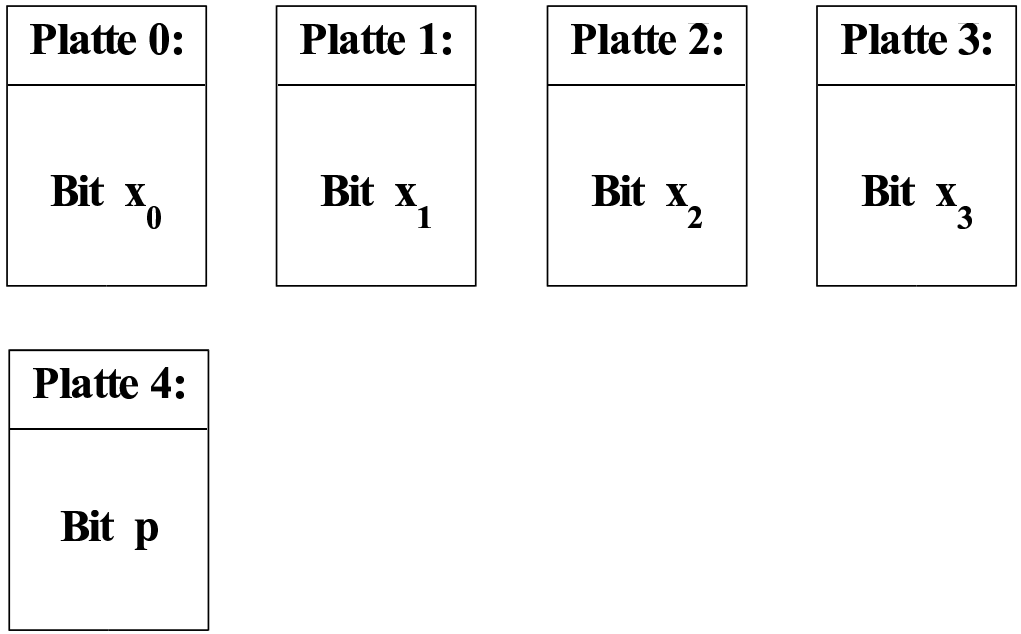
RAID 3: Beispiel mit 4+1 Platten

$$X_0 \oplus X_1 \oplus X_2 \oplus X_3 = p$$

X_0	X_1	X_2	X_3	p
1	1	0	1	1
0	1	1	0	0
0	0	0	0	0
1	1	0	0	0

$$X_1 = X_0 \oplus X_2 \oplus X_3 \oplus p$$

X_1	X_0	X_2	X_3	p
1	1	0	1	1
1	0	1	0	0
0	0	0	0	0
1	1	0	0	0



RAID 4, 5, 6

RAID 4:
Dies ist ähnlich zu RAID 3. Die Paritätsinformation nutzt man nur bei Totalausfall einer Platte. Daher führen nur Schreibzugriffe zu einer erhöhten Last. Jeder Schreibzugriff erfordert zwei Lese- und zwei Schreibzugriffe.

RAID 5:
Um den Flaschenhals der Paritätsplatte zu mildern, verteilt man die Redundanzinformation zyklisch über alle Platten.

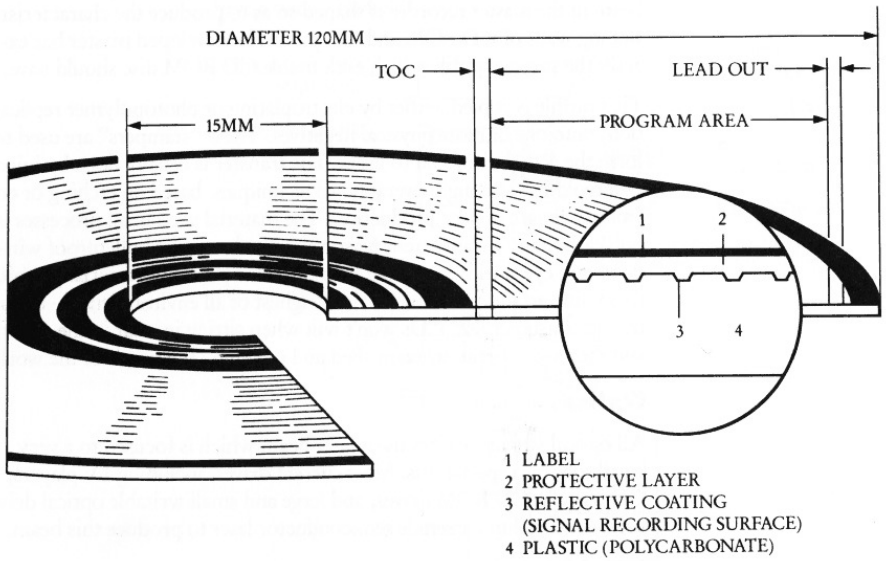
RAID 6:
Hier erhöht man die Datensicherheit gegenüber RAID 5 durch Bildung mehrerer unabhängiger Schutzinformationen.

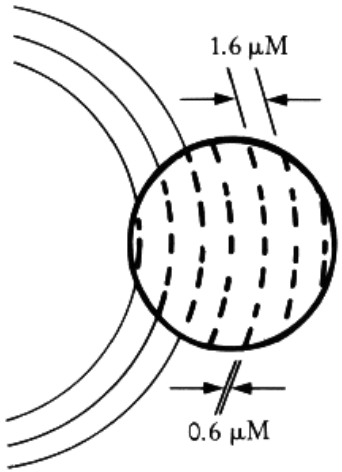
Optische Speichermedien: Beispiel CD-ROM

CD-ROM = „Compact Disc Read-Only Memory“

- Weiterentwicklung der Audio-CD (1982)
- Aufzeichnungspirale der Daten ca. 5 km Länge
- Speicherkapazität:
  - 74-Minuten-CD-DA: ca. 783 MB Audiodaten
  - CD-ROM: zusätzliche Fehlerkorrektur, netto ca. 650 MB Nutzdaten
- Standards:
  - Audio-CD und CD-ROM sind IEC 908 („red book“)
  - ISO/IEC 10149 („yellow book“)

CD/DVD: Prinzip

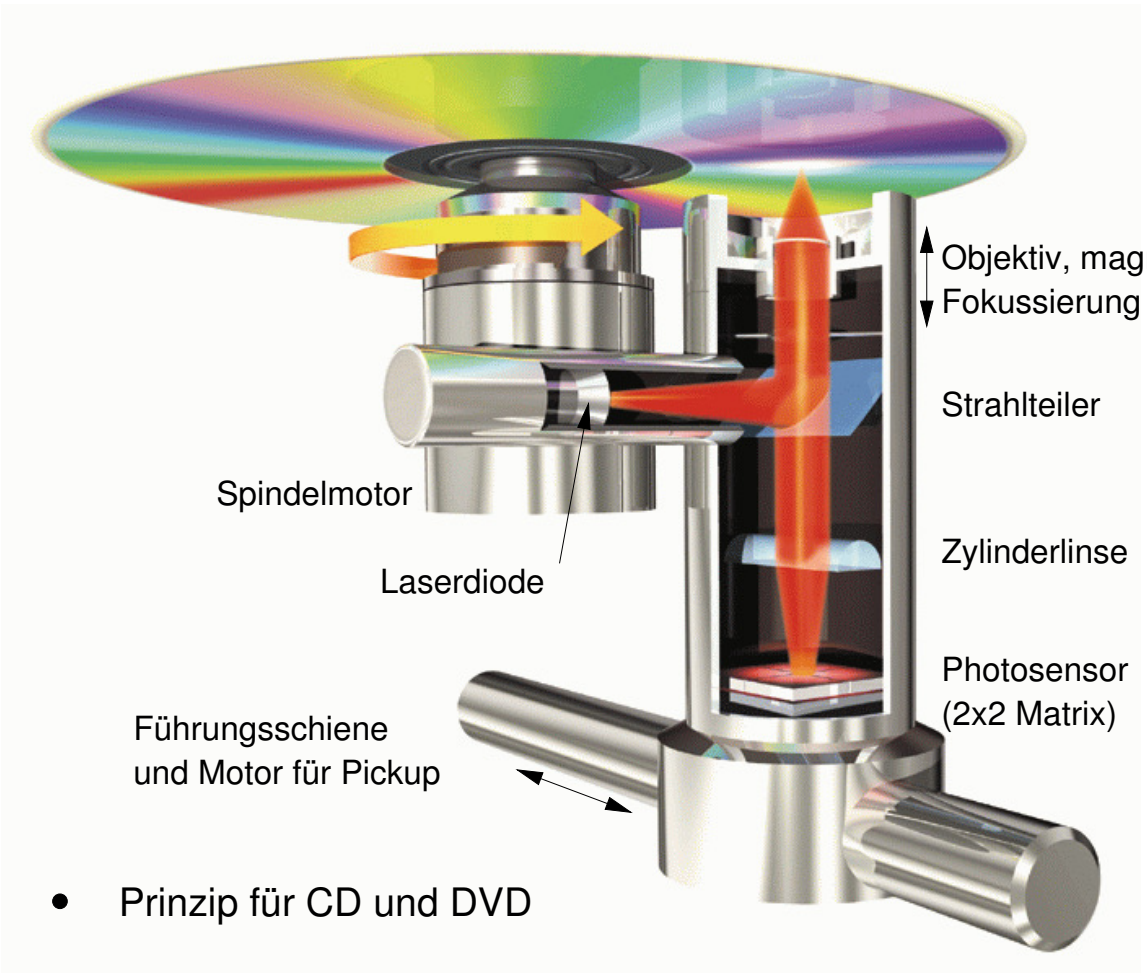




- Polycarbonatträger, 12cm Durchmesser
- eingeprägte Vertiefungen ("pits") bilden die Daten
- spiralförmige Datenspur, 1.6µm Abstand, ca. 16000 Windungen
- Fertigungsmängel fest eingeplant => leistungsfähige Fehlerkorrektur

(CD–ROM – the new papyrus)

CD/DVD: Aufbau eines Players



- Prinzip für CD und DVD

Aufzeichnungsverfahren für die Rohdaten

- zu je 24 8-Bit-Datenbyte werden hinzugefügt:
  - 4 Q-Paritäts-Byte
  - 4 P-Paritäts-Byte
  - ein Subcode-Byte
- jedes Byte wird durch 17 Kanalbit codiert
- geschickte Verschachtelung der Datenbytes
- Wahrscheinlichkeit von 10^{-8} für nicht korrigierbare Lesefehler auf der Audio-Ebene

- für Daten ist dies zu gering
  - auf 2048 Datenbytes zusätzliche 276 Bytes Fehlerkorrekturcode; Wahrscheinlichkeit von 10^{-12} für nicht korrigierbare Lesefehler auf Daten-Ebene

Ein Audio-Rahmen:

Synchronisation	24	Bit
Subcode	14	Bit
6 * 2 * 2 * 14 Datenbit	336	Bit
8 * 14 Paritätsbit	112	Bit
34 * 3 Pufferbit	102	Bit
Summe:	588	Bit

Ein Audio-Rahmen: Bemerkungen

- 192 Datenbit werden in 588 Kanalbit codiert.
- das Subcode-Byte enthält 8 Bit, die den sog. Subkanälen P, Q, R, S, T, U, V, W zugeordnet sind. Nur die Subkanäle P und Q werden von der Audio-disc genutzt.
- Fehlerbehandlung erfolgt in drei Schritten:
  - zunächst wird versucht, die fehlerhaften Bytes zu korrigieren.
  - unkorrigierbare Bytewerte versucht man durch Interpolation zu gewinnen.
  - läßt sich auch die Interpolation nicht durchführen, dann wird der entsprechende Musikteil durch Stille ersetzt.

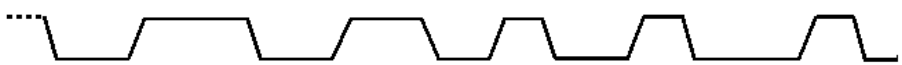
Beispiel zur Codeverschränkung:

- Länge der Nachricht: 103 Zeichen
- ohne Codeverschränkung gespeichert:
  - Auslesen der teilweise zerstörten Nachricht:
    Ein Buch mit dem Titel „Das Geheimnis meiner XXXXXXXXX“
    kann nichts anderes als leere Seiten enthalten.
- Nachricht gespeichert mit Verschränkungszahl 11:
  - Gespeicherte Zeichenfolge:
    E Itnsmi aeialet etSnhnlilhai ene so ann di iBXXXXXXXXXmul
    eDtermeincenaer ncihe„ sneMT hs r .siiet mekG
  - Auslesen der teilweise zerstörten Nachricht:
    Ein Buch mit dXm Titel XDas Geheimnis meinXr
    MillioXen„ kann Xichts anXeres alsX leere SeiXen enthaXten.
- ursprüngliche Nachricht jetzt rekonstruierbar.

Bemerkung: Mittels Codeverschränkung werden Bündelfehler in Einzelfehler transformiert.

Konversion von 8-Bit-Byte zu „pits and lands“

- Daten:
  11101000 11100010 10111010 11101011
- Umsetzung in 14-Bit-Darstellung (Eight-to-Fourteen-Modulation):
  10000100000010 10000100010010 10010000001001 00001001000010
- Einfügen von Pufferbits:
  10000100000010 000 10000100010010 000 10010000001001 001
  00001001000010
- Bitfolge:
  10000100000010000100001000100100001001000000100100100001001000010
- „Pits and Lands“:



  Jede 1 codiert einen Wechsel zwischen „pit“ und „land“

CD-ROM/DVD als Datenspeicher

- (ehemals) hohe Speicherkapazität: 650 MB, 4.7 GB, 9 GB
- günstige Massenfabrikation
- CD-R und DVD-R als beschreibbare Variante

- Datentransferrate 150 kB/s (x1) bis ca. 7 MB/s (x48)
  - Drehzahl des Laufwerks ändert sich beim Lesen
  - innere Spuren höhere Drehzahl als äußere
- sehr langsamer Zugriff auf einzelne Sektoren (ca. 100 ms)
  - Laufwerk muß passende Position „raten“
  - Lesekopf dann auf Datenspur justieren
  - Drehzahl anpassen, Daten bitweise lesen
  - ggf. viele Daten-Frames lesen bis gesuchter Block erreicht ist