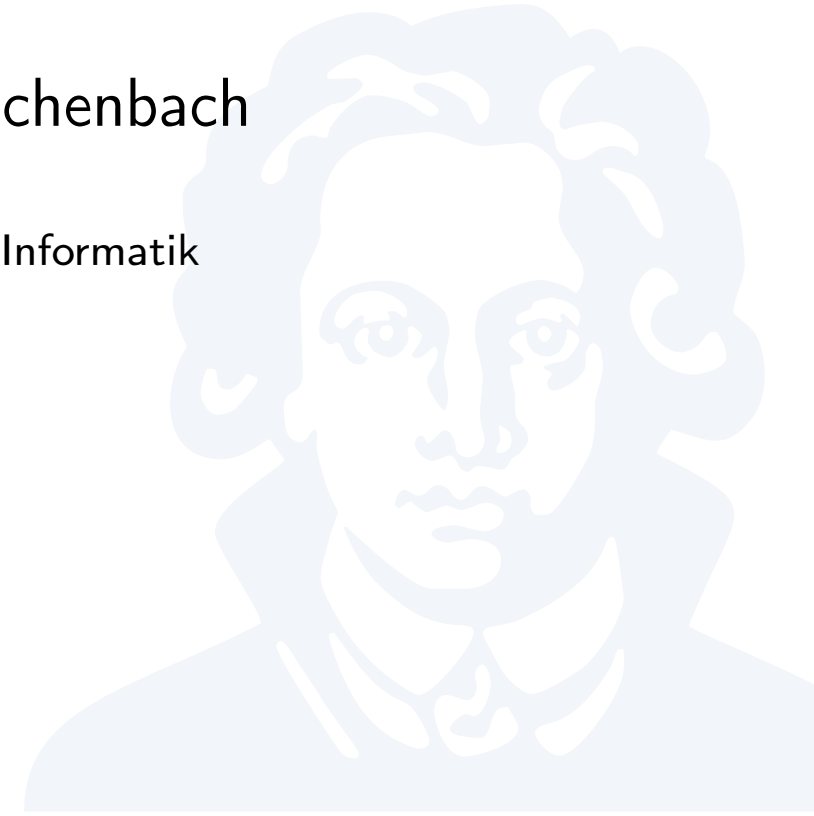
Einführung in die Systemprogrammierung 03

Prof. Dr. Christoph Reichenbach

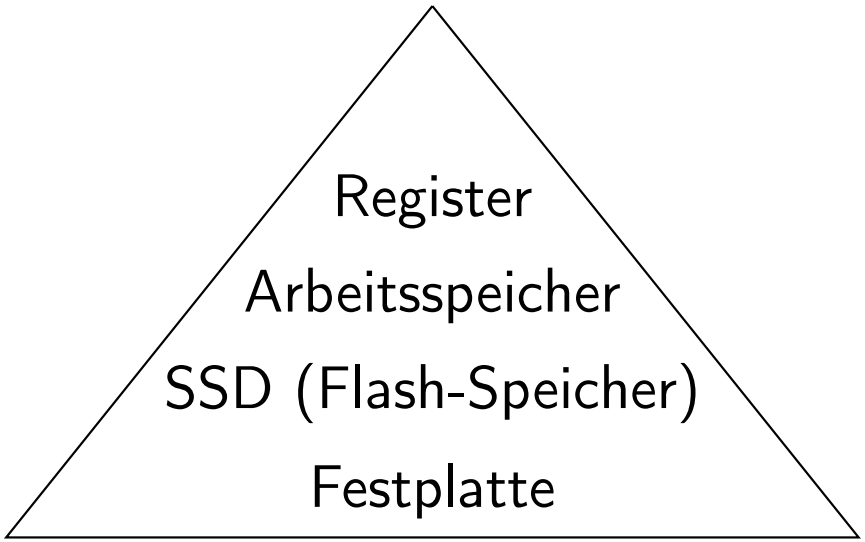
Fachbereich 12 / Institut für Informatik

6. Mai 2014



Speicherformen und deren Eigenschaften

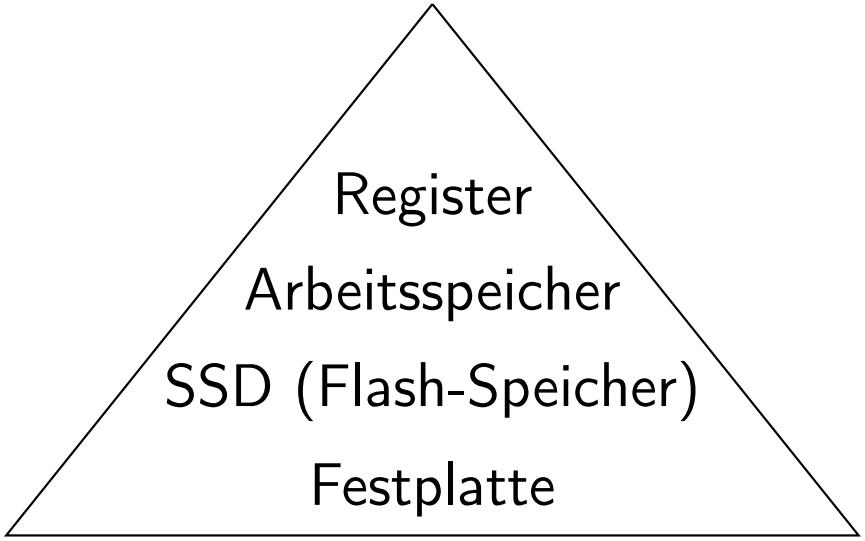
2014/2015



	Speicherplatz	**Zugriffszeit**
Register	$\approx$ 16 kiB	sofort
Arbeitsspeicher	$\approx$ 512 GiB	$\approx$ 200 Zyklen
SSD (Flash-Speicher)	$\approx$ 2 TiB	$>$ 100000 Zyklen
Festplatte	$\approx$ 6 TiB	$>$ 30000000 Zyklen

Speicherformen und deren Eigenschaften

2014/2015

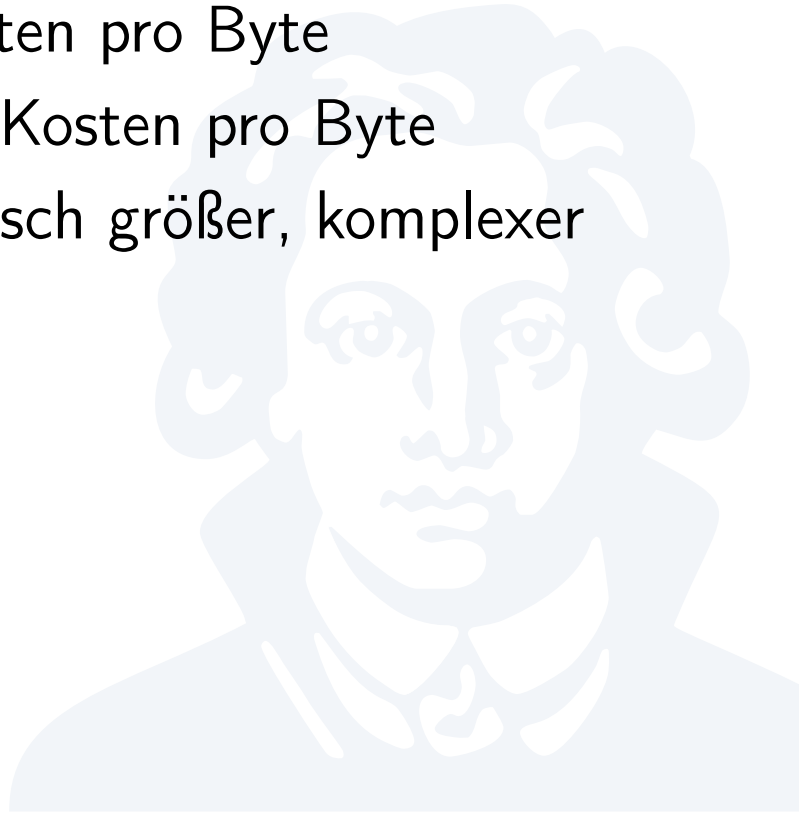


	Speicherplatz	**Zugriffszeit**
Register	$\approx$ 16 kiB	sofort
Arbeitsspeicher	$\approx$ 512 GiB	$\approx$ 200 Zyklen
SSD (Flash-Speicher)	$\approx$ 2 TiB	$>$ 100000 Zyklen
Festplatte	$\approx$ 6 TiB	$>$ 30000000 Zyklen

Hierarchie von Speicherformen

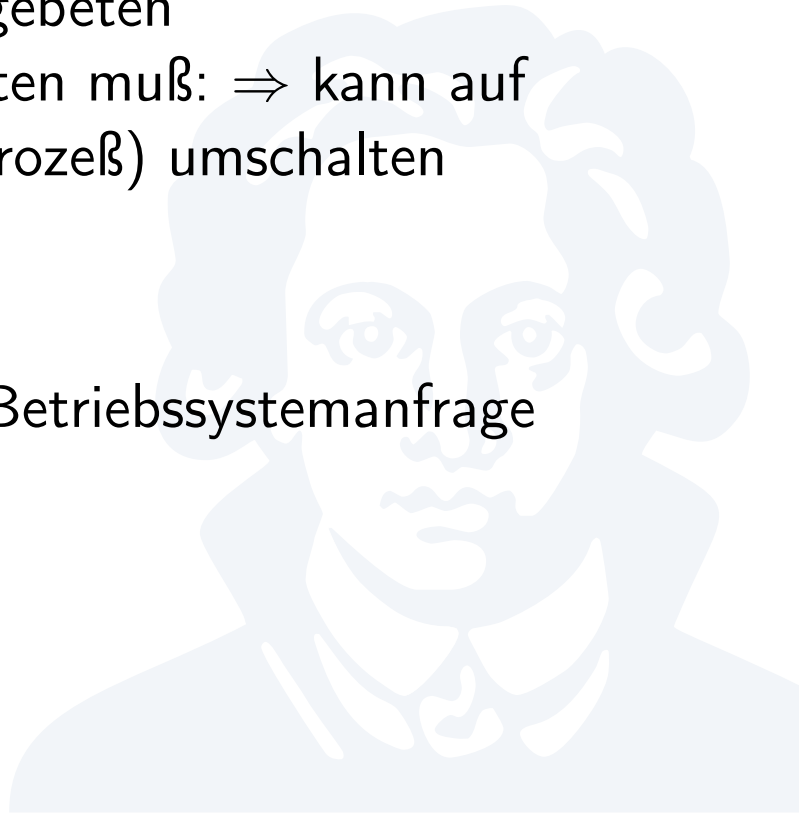
Die Speicherhierarchie

- Schnelle Speicher: klein, hohe Kosten pro Byte
- Langsame Speicher: groß, geringe Kosten pro Byte
- Gründe: größere Speicher physikalisch größer, komplexer organisiert



Wartezyklen

- Zugriff auf langsamere Speicher kann Millionen von *Wartezyklen* kosten
- Bei Festspeicherzugriff:
  - Betriebssystem wird um Daten gebeten
  - Betriebssystem weiß, das es warten muß: $\Rightarrow$ kann auf anderes laufendes Programm (Prozeß) umschalten
  - Prozeßwechsel: $>$ 20.000 Zyklen
- Bei Arbeitsspeicher-Zugriff:
  - Zu schnell für Prozeßwechsel / Betriebssystemanfrage
  - Hunderte von Wartezyklen



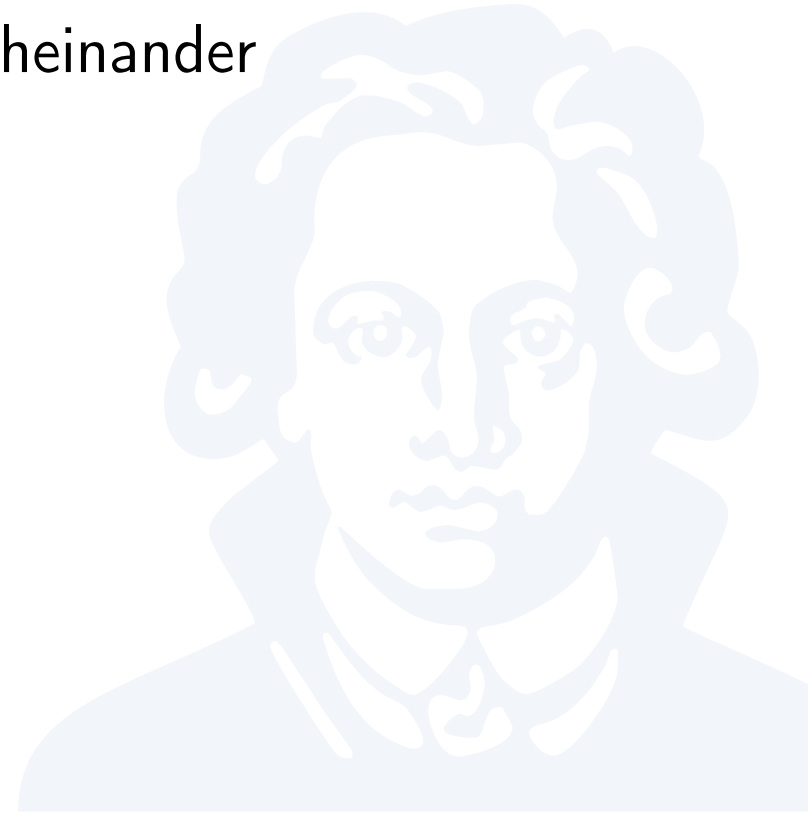
Wartezyklen

- Zugriff auf langsamere Speicher kann Millionen von *Wartezyklen* kosten
- Bei Festspeicherzugriff:
  - Betriebssystem wird um Daten gebeten
  - Betriebssystem weiß, das es warten muß: $\Rightarrow$ kann auf anderes laufendes Programm (Prozeß) umschalten
  - Prozeßwechsel: $>$ 20.000 Zyklen
- Bei Arbeitsspeicher-Zugriff:
  - Zu schnell für Prozeßwechsel / Betriebssystemanfrage
  - Hunderte von Wartezyklen

Können wir das Konzept der Speicherhierarchie effizienter nutzen?

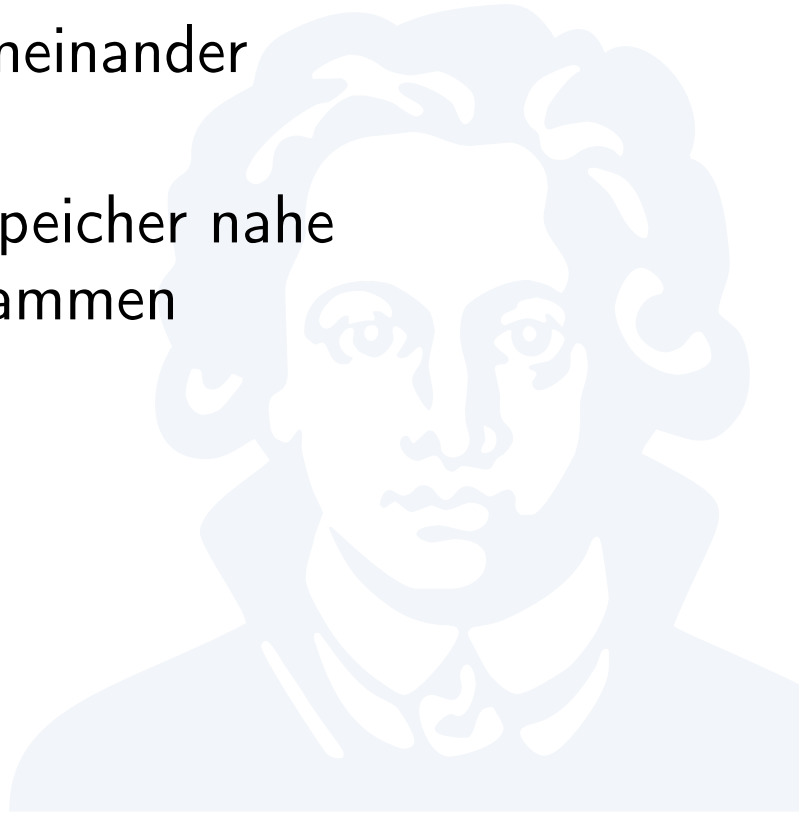
Lokalität

- Beobachtung #1: Daten an der gleichen Stelle im Speicher werden oft mehrfach nacheinander gelesen/geschrieben



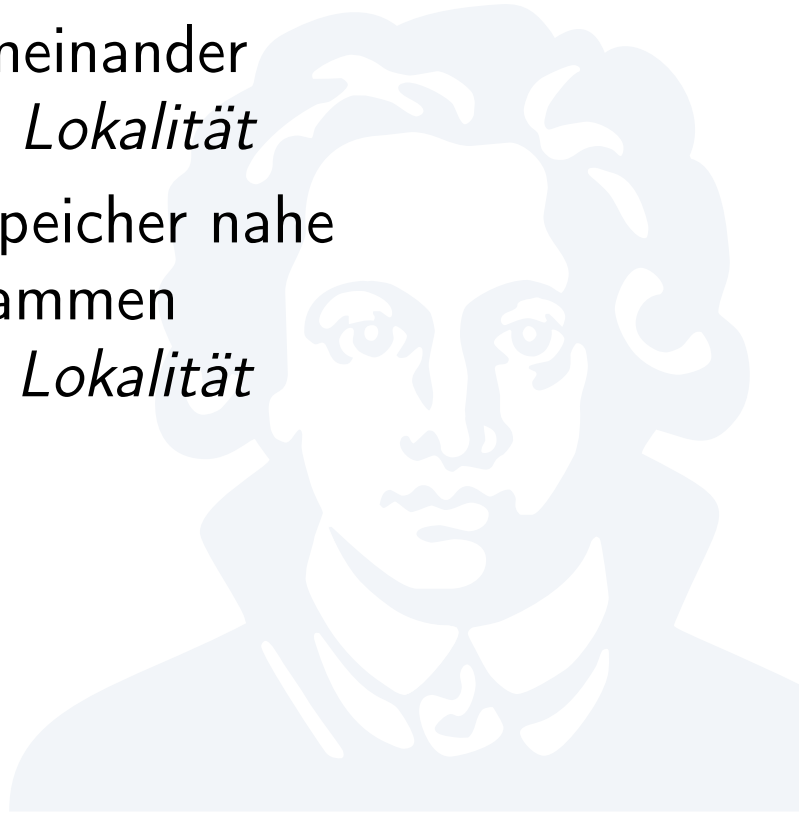
Lokalität

- Beobachtung #1: Daten an der gleichen Stelle im Speicher werden oft mehrfach nacheinander gelesen/geschrieben
- Beobachtung #2: Daten, die im Speicher nahe beieinander liegen, werden oft zusammen gelesen/geschrieben



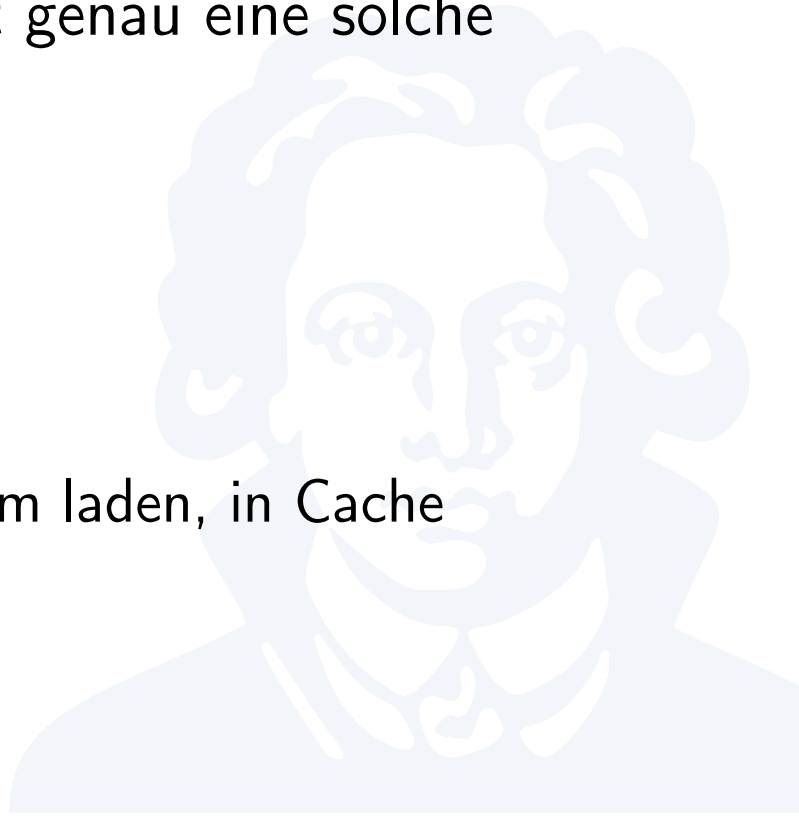
Lokalität

- Beobachtung #1: Daten an der gleichen Stelle im Speicher werden oft mehrfach nacheinander gelesen/geschrieben $\Rightarrow$ *Temporale Lokalität*
- Beobachtung #2: Daten, die im Speicher nahe beieinander liegen, werden oft zusammen gelesen/geschrieben $\Rightarrow$ *Räumliche Lokalität*



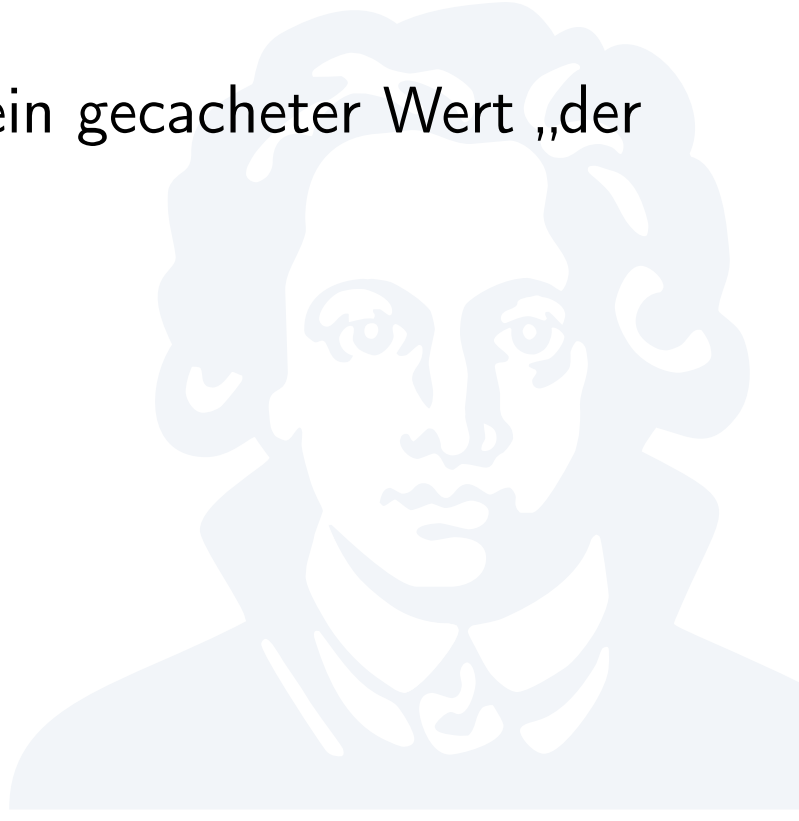
Caches

- Idee: Wir legen *Kopien* von wahrscheinlich bald benötigten Daten in schnellerem Speicher ab
- Ein *Cache* zu einem Speicher S ist genau eine solche Kopie:
  - Schneller als S
  - Kleiner als S
- Lesezugriff:
  - Im Cache? Schneller laden
  - Nicht im Cache? Normal/langsam laden, in Cache kopieren, dann wie zuvor



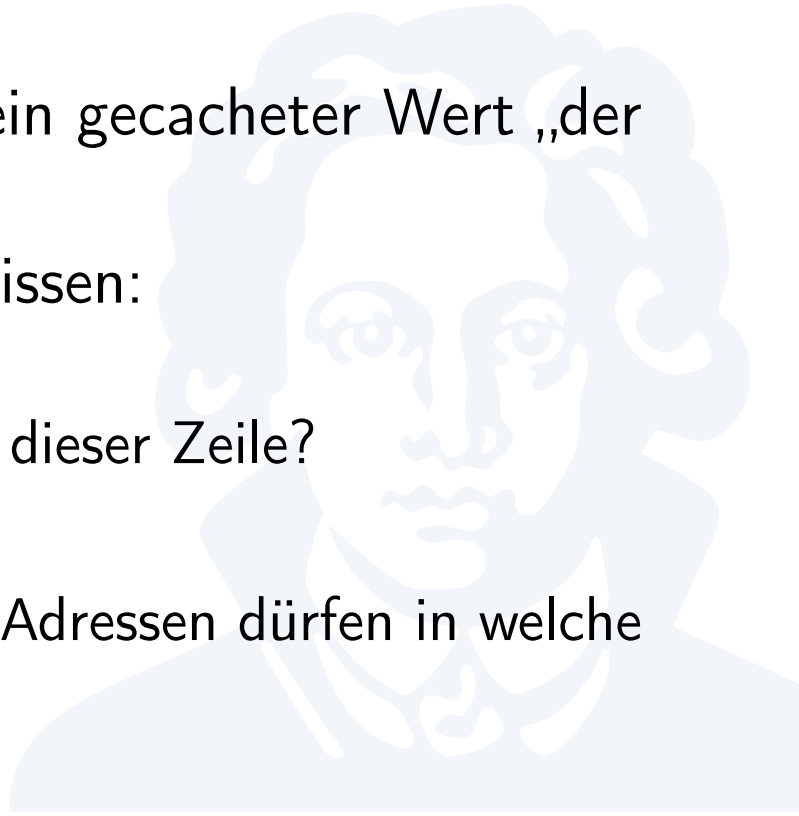
Struktur eines Caches

- Cache zu S besteht aus mehreren Einträgen (Blöcke (*cache block*), auch Zeilen (*cache line*) genannt)
- Weniger Platz als S: Mehrere Adressen von S 'drängeln' sich in den Cache
- *Problem*: Wie stellen wir fest, ob ein gecacheter Wert „der richtige“ ist?



Struktur eines Caches

- Cache zu S besteht aus mehreren Einträgen (Blöcke (*cache block*), auch Zeilen (*cache line*) genannt)
- Weniger Platz als S: Mehrere Adressen von S 'drängeln' sich in den Cache
- *Problem*: Wie stellen wir fest, ob ein gecacheter Wert „der richtige“ ist?
- Jeder Zeile in einem Cache muß wissen:
  - Ist diese Zeile gültig?
  - Welcher Adresse in S entspricht dieser Zeile?
- Verschiedene Cache-Typen
  - Hauptunterscheidung: welche S-Adressen dürfen in welche Cache-Stellen?

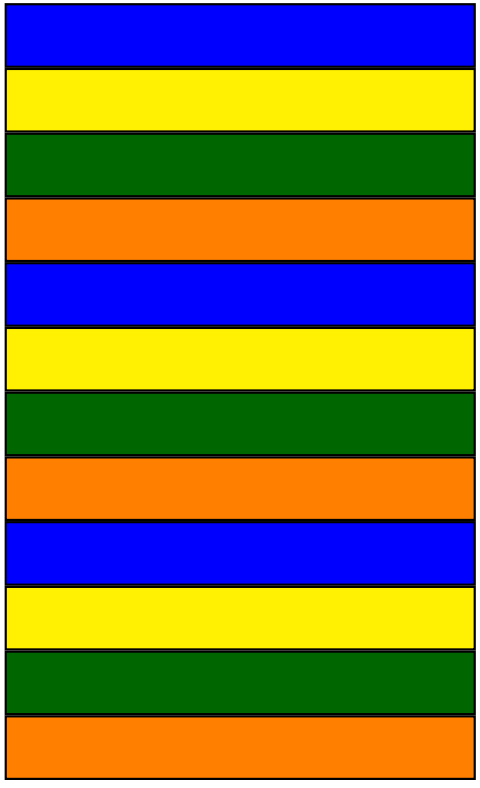


Direkt abgebildete Caches

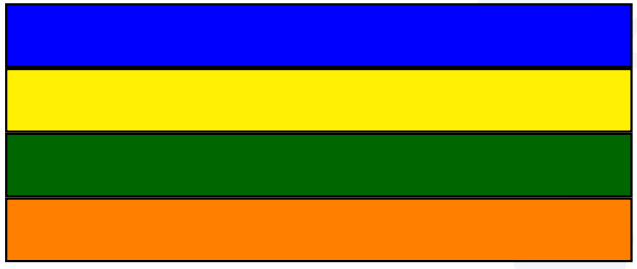
Speicherzuordnung bei *direkt abgebildeten Caches*

- Jede Speicheradresse hat genau eine Cachezeile zugeordnet

Speicher S

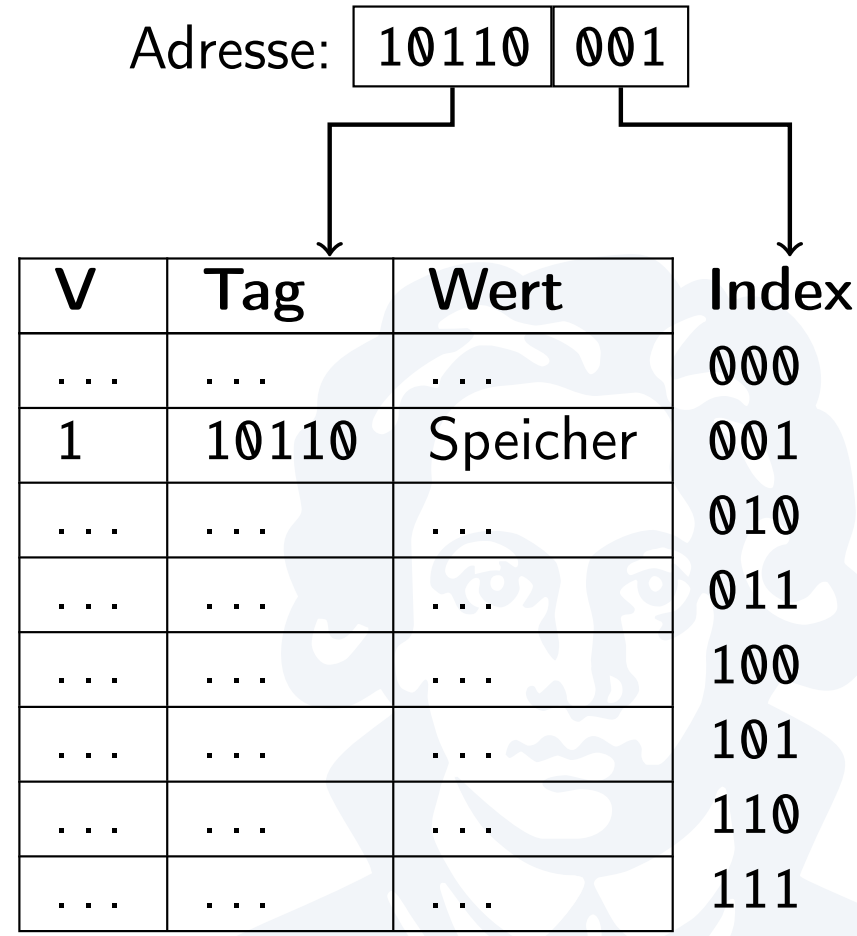


Cache



Direkt abgebildete Caches

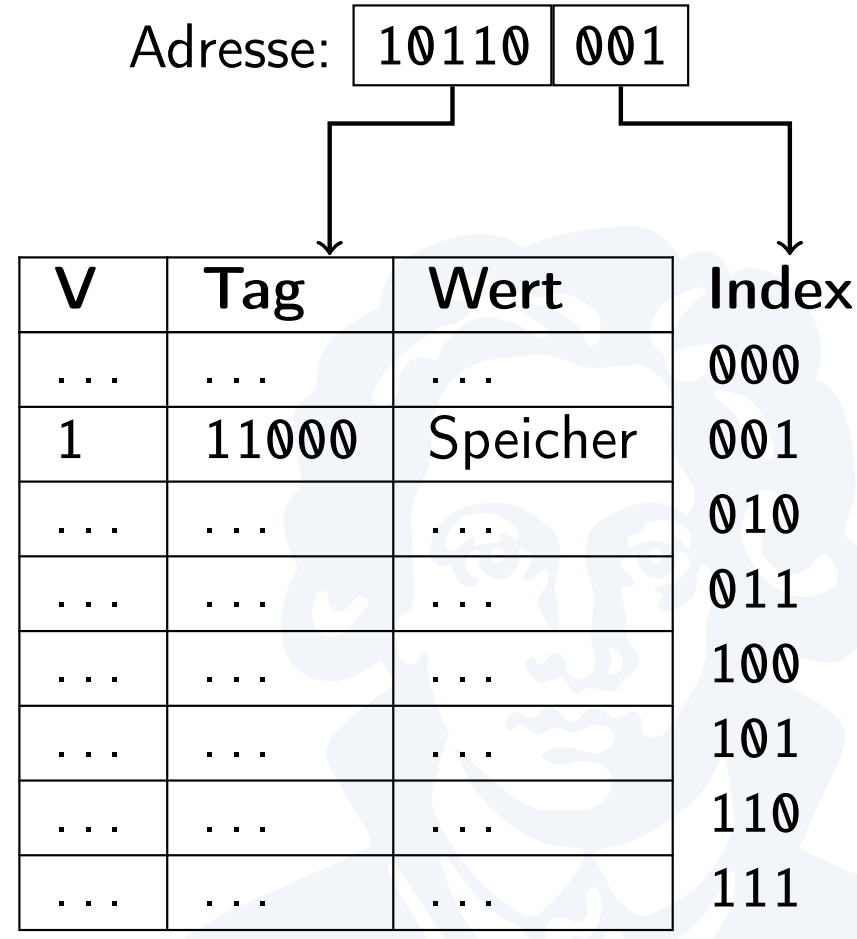
- Beispiel: mit 8 Einträgen
- Gültigkeitsbit **V** merkt sich, ob Eintrag valide ist
- Adressbits in *Tag*, *Index* geteilt
- Bei 2^n Einträgen:
  - Hintere n Bits der Adresse sind Index
  - Rest ist Tag
- *S*-Adresse ergibt sich aus *Tag* und *Cache-Index*



Direkt abgebildete Caches: Lesen

Prozedur: Laden einer Speicherzelle

- Hintere Bits: *Index*
- Aktueller Cache-Eintrag am Index gültig?
  - *nein*: *Fehlzugriff*
  - *ja*: (Tag, Index) stimmt mit Adresse überein?
    - *nein*: *Fehlzugriff*
    - *ja*: Speicher aus Cache laden
- Bei Fehlzugriff:
  - Wert aus Speicher S laden (langsam)
  - In Cache auf Eintrag *Index* schreiben
  - Aus Cache an Prozessor weiterreichen



Direkt abgebildete Caches: Beispiel

```
⇒   lbu $t0, 0b10110001($0)
    lbu $t0, 0b10001110($0)
    lbu $t0, 0b10110001($0)
    lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Idx
0	...	...	000
0	...	...	001
0	...	...	010
0	...	...	011
0	...	...	100
0	...	...	101
0	...	...	110
0	...	...	111

Direkt abgebildete Caches: Beispiel

```
⇒   lbu $t0, 0b10110001($0)
    lbu $t0, 0b10001110($0)
    lbu $t0, 0b10110001($0)
    lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Idx
0	...	...	000
0	...	...	**001**
0	...	...	010
0	...	...	011
0	...	...	100
0	...	...	101
0	...	...	110
0	...	...	111

Direkt abgebildete Caches: Beispiel

```
    lbu $t0, 0b10110001($0)    Fehlz.
⇒   lbu $t0, 0b10001110($0)
    lbu $t0, 0b10110001($0)
    lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Idx
0	...	...	000
1	10110	Speicher	**001**
0	...	...	010
0	...	...	011
0	...	...	100
0	...	...	101
0	...	...	110
0	...	...	111

Direkt abgebildete Caches: Beispiel

```
    lbu $t0, 0b10110001($0)    Fehlz.
    lbu $t0, 0b10001110($0)    Fehlz.
⇒   lbu $t0, 0b10110001($0)
    lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Idx
0	...	...	000
1	10110	Speicher	001
0	...	...	010
0	...	...	011
0	...	...	100
0	...	...	101
1	10001	Speicher	110
0	...	...	111

Direkt abgebildete Caches: Beispiel

```
    lbu $t0, 0b10110001($0)    Fehlz.
    lbu $t0, 0b10001110($0)    Fehlz.
    lbu $t0, 0b10110001($0)    Treffer
⇒   lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Idx
0	...	...	000
1	10110	Speicher	**001**
0	...	...	010
0	...	...	011
0	...	...	100
0	...	...	101
1	10001	Speicher	110
0	...	...	111

Direkt abgebildete Caches: Beispiel

```
    lbu $t0, 0b10110001($0)    Fehlz.
    lbu $t0, 0b10001110($0)    Fehlz.
    lbu $t0, 0b10110001($0)    Treffer
    lbu $t0, 0b00000001($0)    Fehlz.
⇒   lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Idx
0	...	...	000
1	~~10110~~	Speicher	**001**
0	...	...	010
0	...	...	011
0	...	...	100
0	...	...	101
1	10001	Speicher	110
0	...	...	111

Direkt abgebildete Caches: Beispiel

```
    lbu $t0, 0b10110001($0)    Fehlz.
    lbu $t0, 0b10001110($0)    Fehlz.
    lbu $t0, 0b10110001($0)    Treffer
    lbu $t0, 0b00000001($0)    Fehlz.
⇒   lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Idx
0	...	...	000
1	00000	Speicher	**001**
0	...	...	010
0	...	...	011
0	...	...	100
0	...	...	101
1	10001	Speicher	110
0	...	...	111

Direkt abgebildete Caches: Beispiel

```
lbu $t0, 0b10110001($0)    Fehlz.
lbu $t0, 0b10001110($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer
lbu $t0, 0b00000001($0)    Fehlz.
lbu $t0, 0b10110001($0)    Fehlz.
```

V	Tag	Wert	Idx
0	...	...	000
1	~~00000~~	Speicher	**001**
0	...	...	010
0	...	...	011
0	...	...	100
0	...	...	101
1	10001	Speicher	110
0	...	...	111

Direkt abgebildete Caches: Beispiel

```
lbu $t0, 0b10110001($0)    Fehlz.
lbu $t0, 0b10001110($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer
lbu $t0, 0b00000001($0)    Fehlz.
lbu $t0, 0b10110001($0)    Fehlz.
```

- **1** Treffer
- **4** Fehlzugriffe

V	Tag	Wert	Idx
0	...	...	000
1	10110	Speicher	001
0	...	...	010
0	...	...	011
0	...	...	100
0	...	...	101
1	10001	Speicher	110
0	...	...	111

Direkt abgebildete Caches: Beispiel

```
lbu $t0, 0b10110001($0)    Fehlz.
lbu $t0, 0b10001110($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer
lbu $t0, 0b00000001($0)    Fehlz.
lbu $t0, 0b10110001($0)    Fehlz.
```

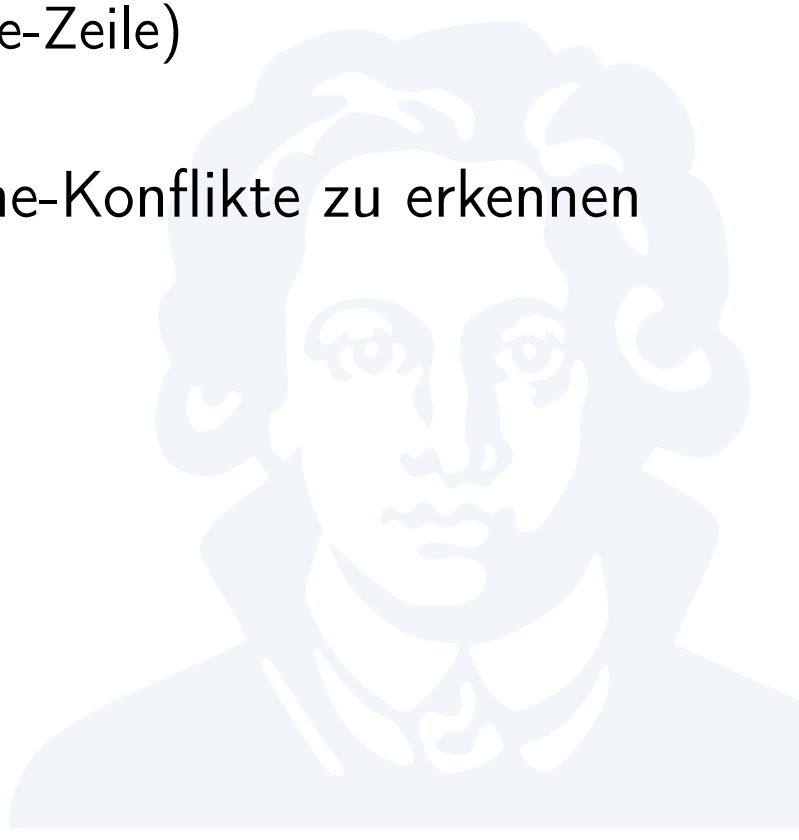
- **1** Treffer
- **4** Fehlzugriffe

V	Tag	Wert	Idx
0	...	...	000
1	10110	Speicher	001
0	...	...	010
0	...	...	011
0	...	...	100
0	...	...	101
1	10001	Speicher	110
0	...	...	111

In der Praxis Speedup von 1,25 plausibel

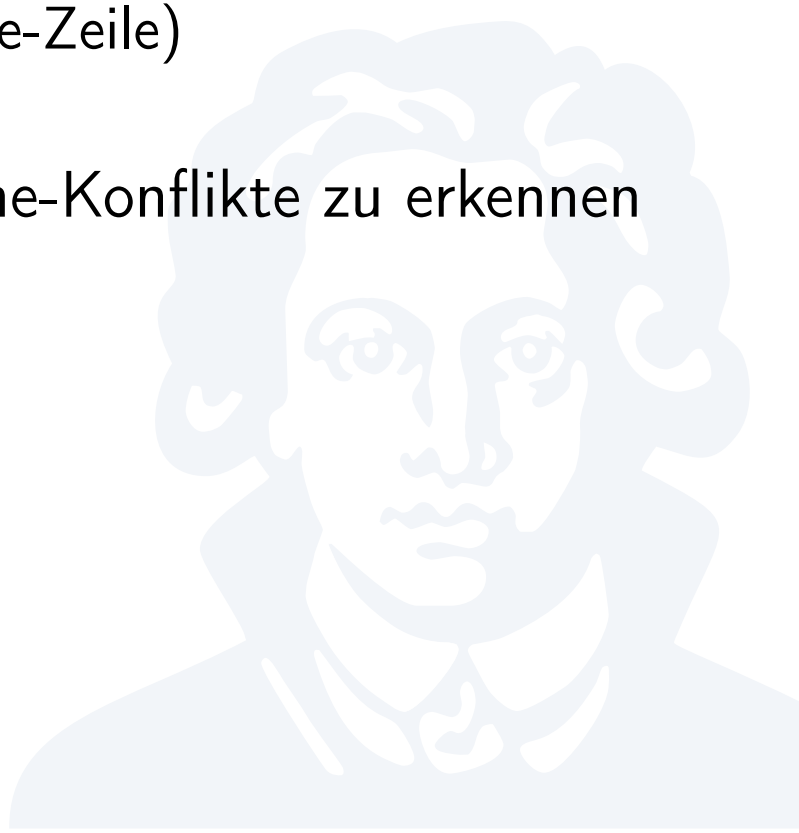
Direkt abgebildete Caches: Zusammenfassung

- Jeder Adresse des gecacheten Speichers S wird genau eine Cache-Zeile zugeordnet
- Adresse teilt sich in:
  - Cache-Index (Nummer der Cache-Zeile)
  - Tag (Rest der Adresse)
- Cache speichert das Tag, um Cache-Konflikte zu erkennen
- Bei Zugriff auf Adresse:
  Cache-Zeile belegt?
  - *nein*
  - *ja*



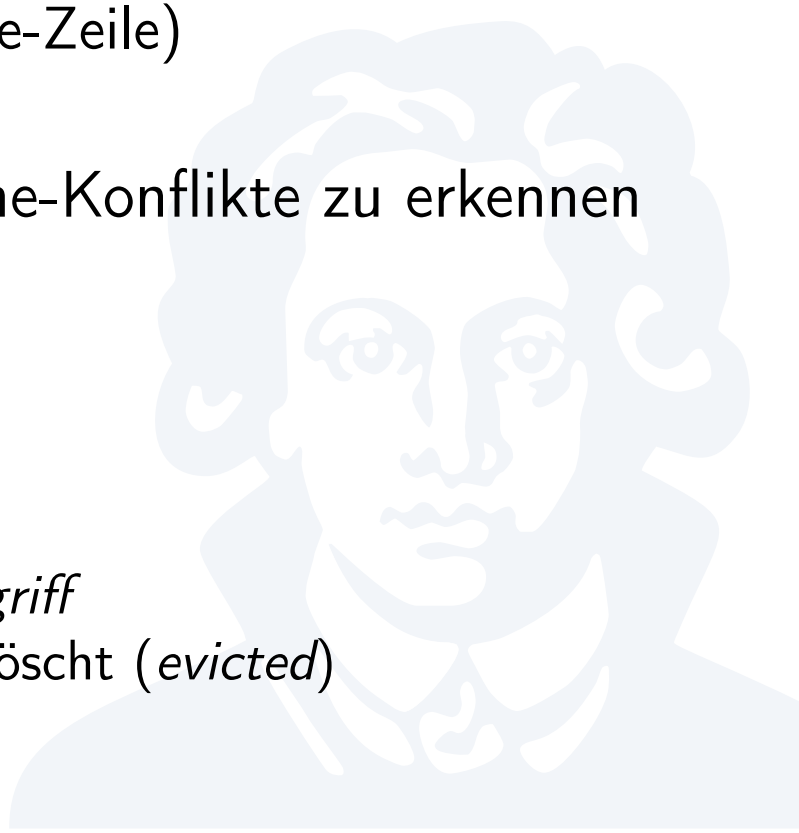
Direkt abgebildete Caches: Zusammenfassung

- Jeder Adresse des gecacheten Speichers S wird genau eine Cache-Zeile zugeordnet
- Adresse teilt sich in:
  - Cache-Index (Nummer der Cache-Zeile)
  - Tag (Rest der Adresse)
- Cache speichert das Tag, um Cache-Konflikte zu erkennen
- Bei Zugriff auf Adresse:
  Cache-Zeile belegt?
  - *nein* $\Rightarrow$ *Fehlzugriff*
  - *ja*



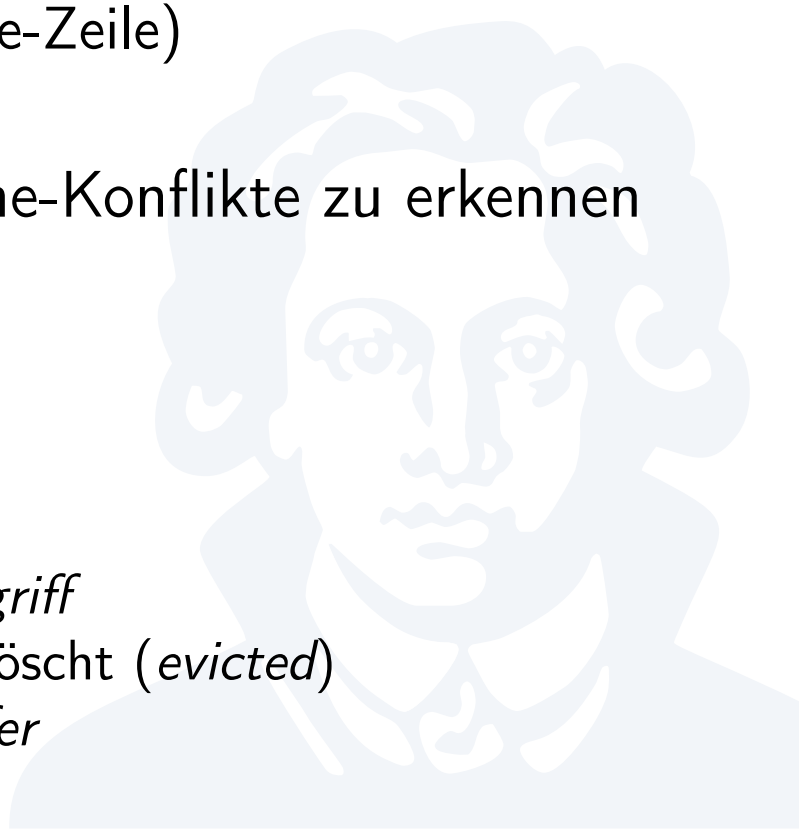
Direkt abgebildete Caches: Zusammenfassung

- Jeder Adresse des gecacheten Speichers S wird genau eine Cache-Zeile zugeordnet
- Adresse teilt sich in:
  - Cache-Index (Nummer der Cache-Zeile)
  - Tag (Rest der Adresse)
- Cache speichert das Tag, um Cache-Konflikte zu erkennen
- Bei Zugriff auf Adresse:
  Cache-Zeile belegt?
  - *nein* $\Rightarrow$ *Fehlzugriff*
  - *ja*:
    - Tag stimmt nicht? $\Rightarrow$ *Fehlzugriff*
      Alter Inhalt der Zeile wird gelöscht (*evicted*)
    - Tag stimmt überein?



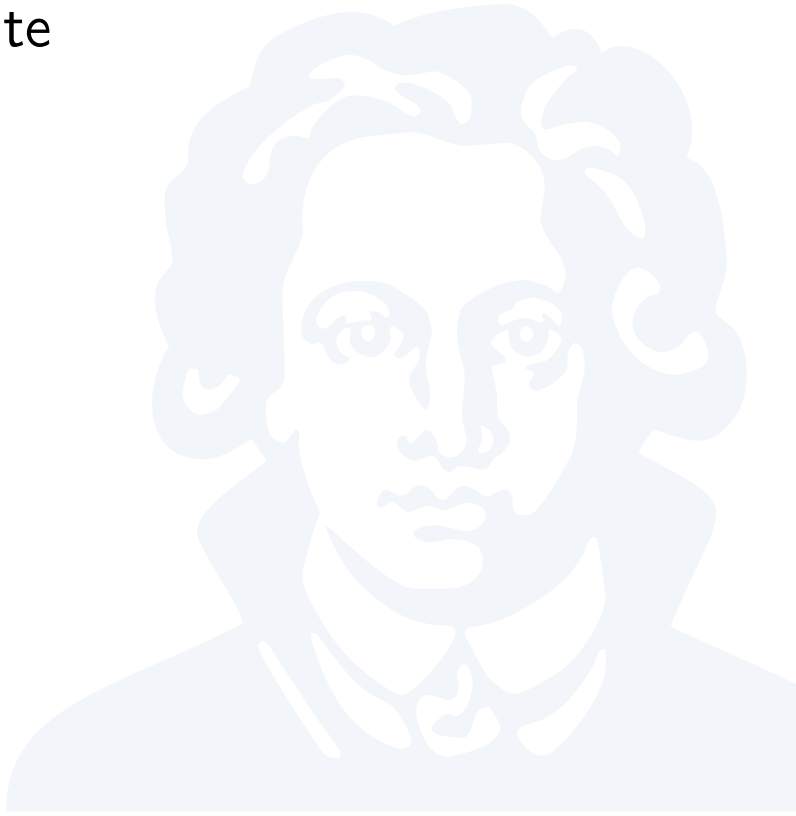
Direkt abgebildete Caches: Zusammenfassung

- Jeder Adresse des gecacheten Speichers S wird genau eine Cache-Zeile zugeordnet
- Adresse teilt sich in:
  - Cache-Index (Nummer der Cache-Zeile)
  - Tag (Rest der Adresse)
- Cache speichert das Tag, um Cache-Konflikte zu erkennen
- Bei Zugriff auf Adresse:
  Cache-Zeile belegt?
  - *nein* $\Rightarrow$ *Fehlzugriff*
  - *ja*:
    - Tag stimmt nicht? $\Rightarrow$ *Fehlzugriff*
      Alter Inhalt der Zeile wird gelöscht (*evicted*)
    - Tag stimmt überein? $\Rightarrow$ *Treffer*



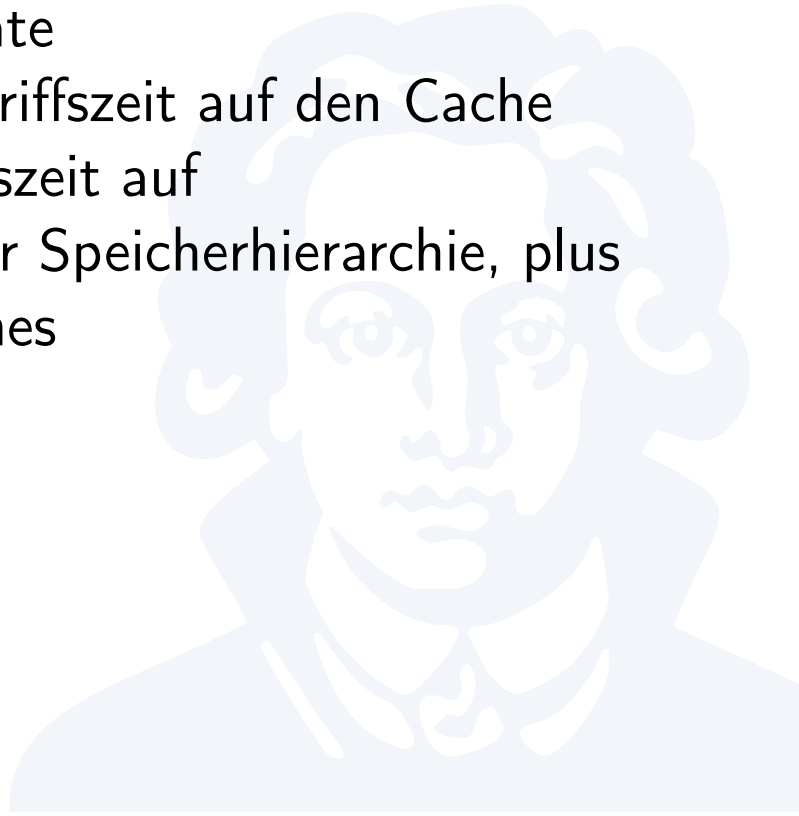
Performanz von Caches

- Performanz von Caches: Messung durch Speedup
- Detaillierte Analyse:
  - **Trefferrate** $= \frac{\text{Treffer}}{\text{Zugriffe}}$
  - **Fehlzugriffsrate** $= 1$ - Trefferrate



Performanz von Caches

- Performanz von Caches: Messung durch Speedup
- Detaillierte Analyse:
  - **Trefferrate** = $\frac{\text{Treffer}}{\text{Zugriffe}}$
  - **Fehlzugriffsrate** = 1 - Trefferrate
  - **Zugriffszeit bei Treffer** = Zugriffszeit auf den Cache
  - **Fehlzugriffsaufwand** = Zugriffszeit auf nächstniedrigeren Speicher in der Speicherhierarchie, plus Zeit zur Aktualisierung des Caches



Performanz von Caches

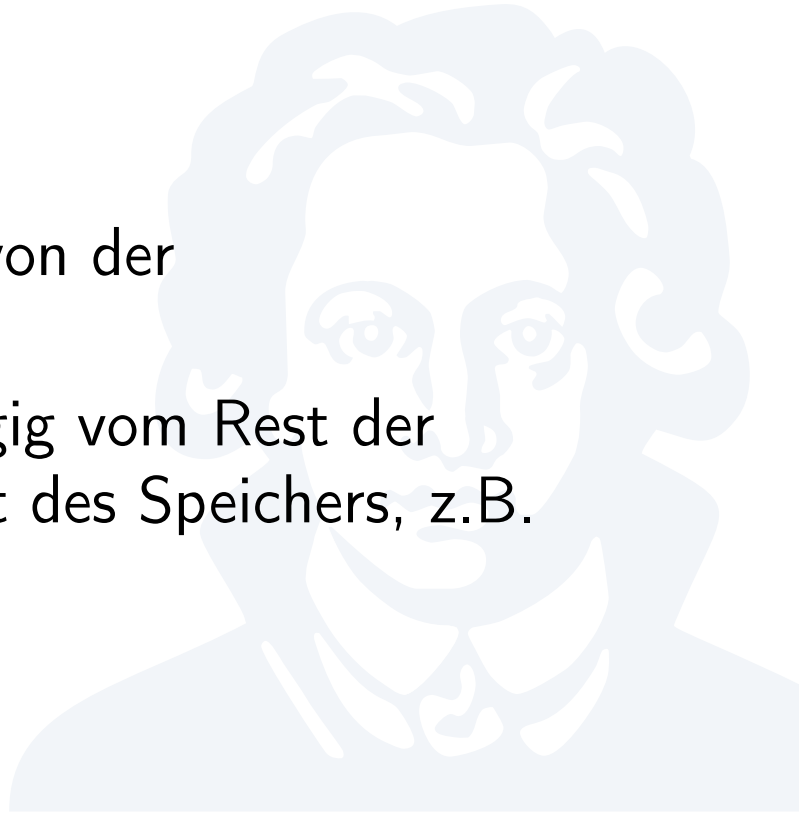
- Performanz von Caches: Messung durch Speedup
- Detaillierte Analyse:
  - **Trefferrate** = $\frac{\text{Treffer}}{\text{Zugriffe}}$
  - **Fehlzugriffsrate** = 1 - Trefferrate
  - **Zugriffszeit bei Treffer** = Zugriffszeit auf den Cache
  - **Fehlzugriffsaufwand** = Zugriffszeit auf nächstniedrigeren Speicher in der Speicherhierarchie, plus Zeit zur Aktualisierung des Caches
- Durchschnittliche Speicherzugriffszeit =

  Zugriffszeit bei Treffer+Fehlzugriffsrate×Fehlzugriffsaufwand

Beispiel: Performanz von Caches

Aus unserem Beispielprogramm:

- **1** Treffer
- **4** Fehlzugriffe
- Trefferrate $= \frac{\text{Treffer}}{\text{Zugriffe}} = \frac{1}{5}$
- Fehlzugriffsrate $= \frac{4}{5}$
- Zugriffszeit bei Treffer: abhängig von der Cache-Architektur, z.B. 4 Zyklen
- Zugriffszeit bei Fehlzugriff: abhängig vom Rest der Speicherhierarchie und von der Art des Speichers, z.B. 200 Zyklen



Beispiel: Performanz von Caches

Aus unserem Beispielprogramm:

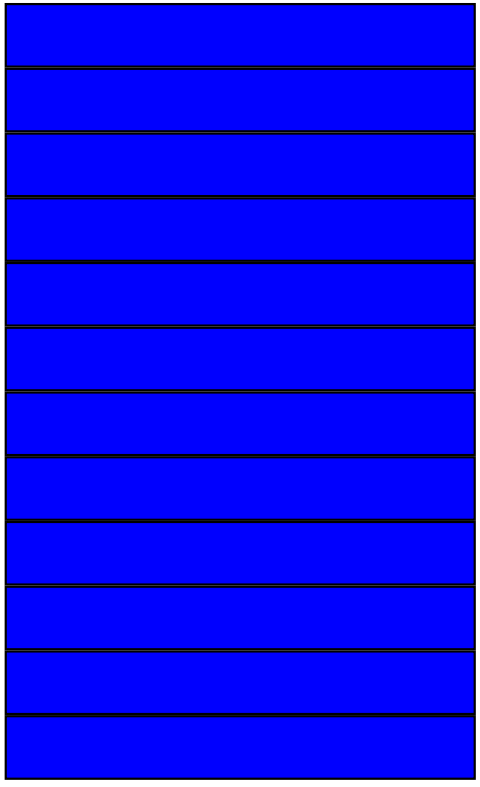
- **1** Treffer
- **4** Fehlzugriffe
- Trefferrate $= \frac{\text{Treffer}}{\text{Zugriffe}} = \frac{1}{5}$
- Fehlzugriffsrate $= \frac{4}{5}$
- Zugriffszeit bei Treffer: abhängig von der Cache-Architektur, z.B. 4 Zyklen
- Zugriffszeit bei Fehlzugriff: abhängig vom Rest der Speicherhierarchie und von der Art des Speichers, z.B. 200 Zyklen
- Durchschnittliche Zugriffszeit $= 4 + \frac{4}{5} \times 200 = 164$

Vollassoziative Caches

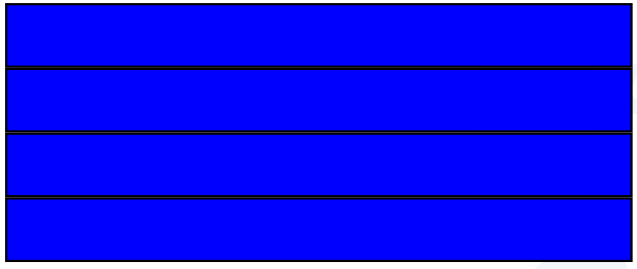
Speicherzuordnung bei *vollassoziativen Caches*

- Jede Speicheradresse darf in jeder Cachezeile sitzen

Speicher S

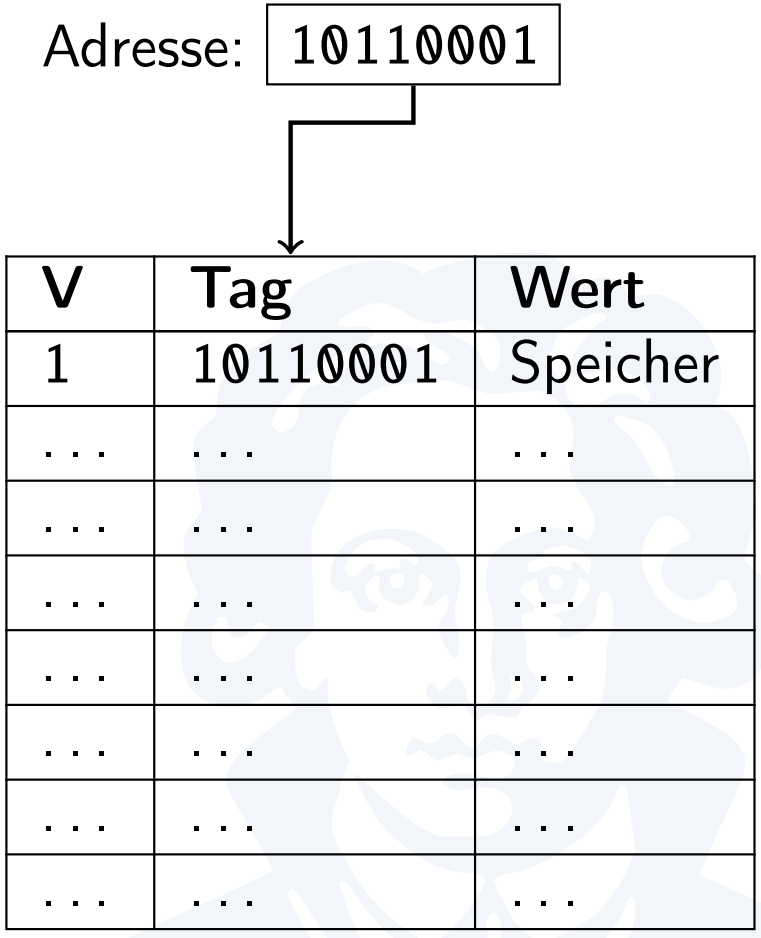


Cache



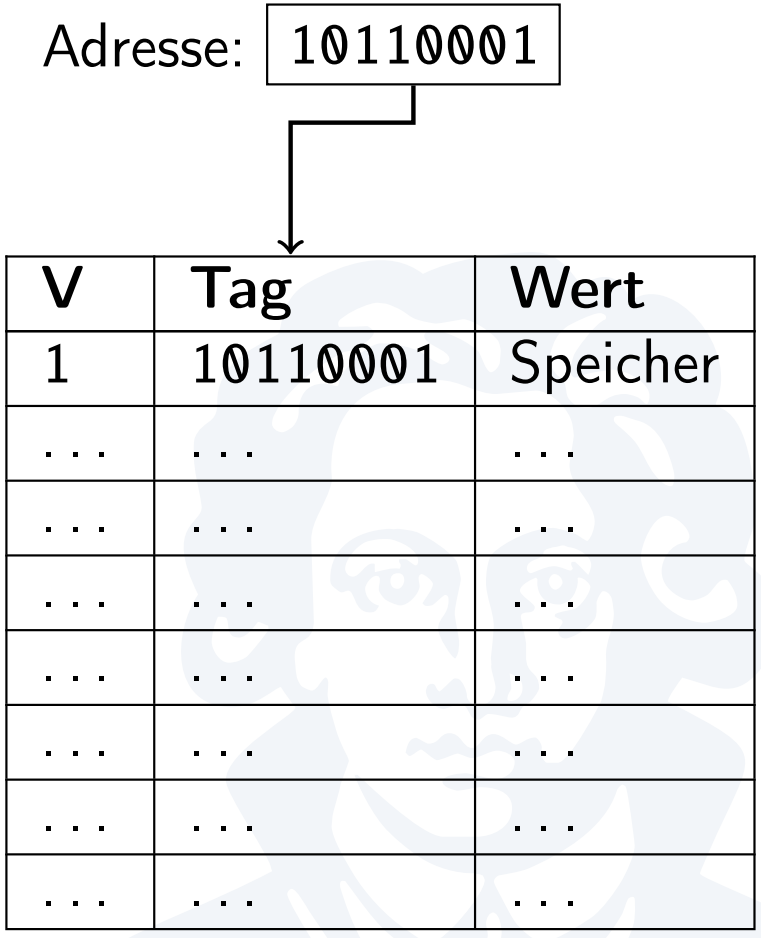
Vollassoziative Caches

- Cache für Speicher S
- Jede Speicherzelle aus S kann in jeder Zeile des Caches gespiegelt werden



Vollassoziative Caches

- Cache für Speicher S
- Jede Speicherzelle aus S kann in jeder Zeile des Caches gespiegelt werden
- Tag = komplette Adresse
- kein Index nötig



Vollassoziative Caches: Beispiel

```
⇒  lbu $t0, 0b10110001($0)
   lbu $t0, 0b10001110($0)
   lbu $t0, 0b10110001($0)
   lbu $t0, 0b00000001($0)
   lbu $t0, 0b10110001($0)
```

V	Tag	Wert
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...

Vollassoziative Caches: Beispiel

```
    lbu $t0, 0b10110001($0)   Fehlz.
⇒   lbu $t0, 0b10001110($0)
    lbu $t0, 0b10110001($0)
    lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert
1	10110001	Speicher
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...

Vollassoziative Caches: Beispiel

```
    lbu $t0, 0b10110001($0)    Fehlz.
    lbu $t0, 0b10001110($0)    Fehlz.
⇒   lbu $t0, 0b10110001($0)
    lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert
1	10110001	Speicher
1	10001110	Speicher
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...

Vollassoziative Caches: Beispiel

```
    lbu $t0, 0b10110001($0)    Fehlz.
    lbu $t0, 0b10001110($0)    Fehlz.
    lbu $t0, 0b10110001($0)    Treffer
⇒   lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert
1	10110001	Speicher
1	10001110	Speicher
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...

Vollassoziative Caches: Beispiel

```
    lbu $t0, 0b10110001($0)     Fehlz.
    lbu $t0, 0b10001110($0)     Fehlz.
    lbu $t0, 0b10110001($0)     Treffer
    lbu $t0, 0b00000001($0)     Fehlz.
⇒   lbu $t0, 0b10110001($0)
```

V	Tag	Wert
1	10110001	Speicher
1	10001110	Speicher
1	00000001	Speicher
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...

Vollassoziative Caches: Beispiel

```
lbu $t0, 0b10110001($0)    Fehlz.
lbu $t0, 0b10001110($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer
lbu $t0, 0b00000001($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer.
```

- 2 Treffer
- 3 Fehlzugriffe

V	Tag	Wert
1	10110001	Speicher
1	10001110	Speicher
1	00000001	Speicher
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...

Vollassoziative Caches: Beispiel

```
lbu $t0, 0b10110001($0)    Fehlz.
lbu $t0, 0b10001110($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer
lbu $t0, 0b00000001($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer.
```

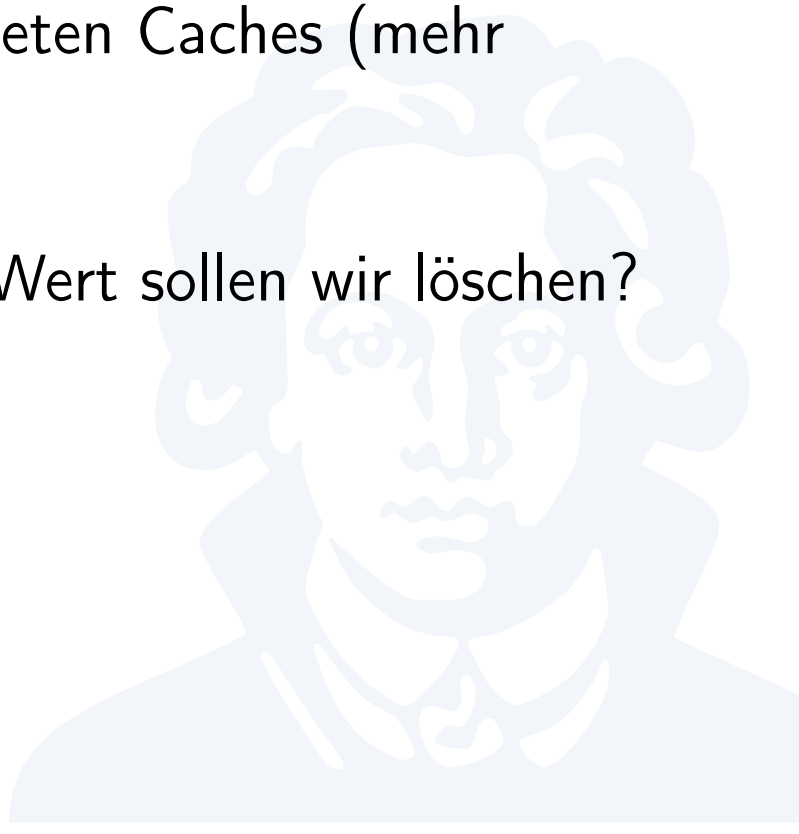
- 2 Treffer
- 3 Fehlzugriffe

V	Tag	Wert
1	10110001	Speicher
1	10001110	Speicher
1	00000001	Speicher
0	...	...
0	...	...
0	...	...
0	...	...
0	...	...

Durchschnittliche Zugriffszeit $= 4 + \frac{3}{5} \times 200 = 124$

Vollassoziative Caches: Herausforderungen

- Sehr effektive Caches
- Teuer (viele Transistoren)
- Tags größer als bei direkt abgebildeten Caches (mehr Platz nötig)
- Herausforderung:
  Wenn der Cache voll ist: welchen Wert sollen wir löschen?



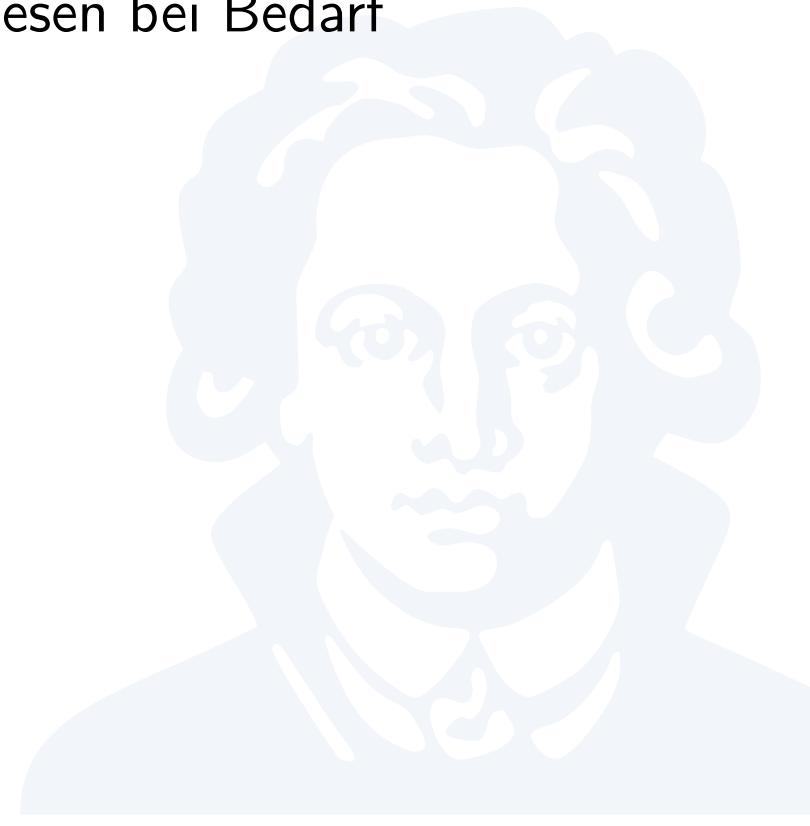
Vollassoziative Caches: Herausforderungen

- Sehr effektive Caches
- Teuer (viele Transistoren)
- Tags größer als bei direkt abgebildeten Caches (mehr Platz nötig)
- Herausforderung:
  Wenn der Cache voll ist: welchen Wert sollen wir löschen?

Ersetzungsstrategien entscheiden, welcher Wert gelöscht wird

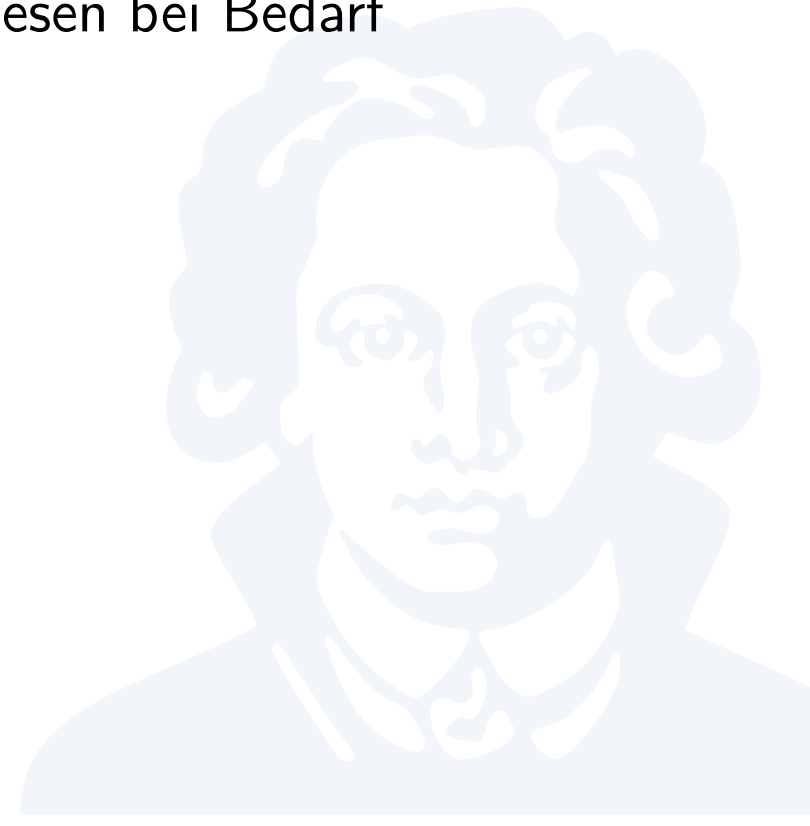
Ersetzungsstrategien

- LRU (*least recently used*):
  - Wir merken uns, welcher Eintrag am längsten nicht verwendet wurde und löschen diesen bei Bedarf



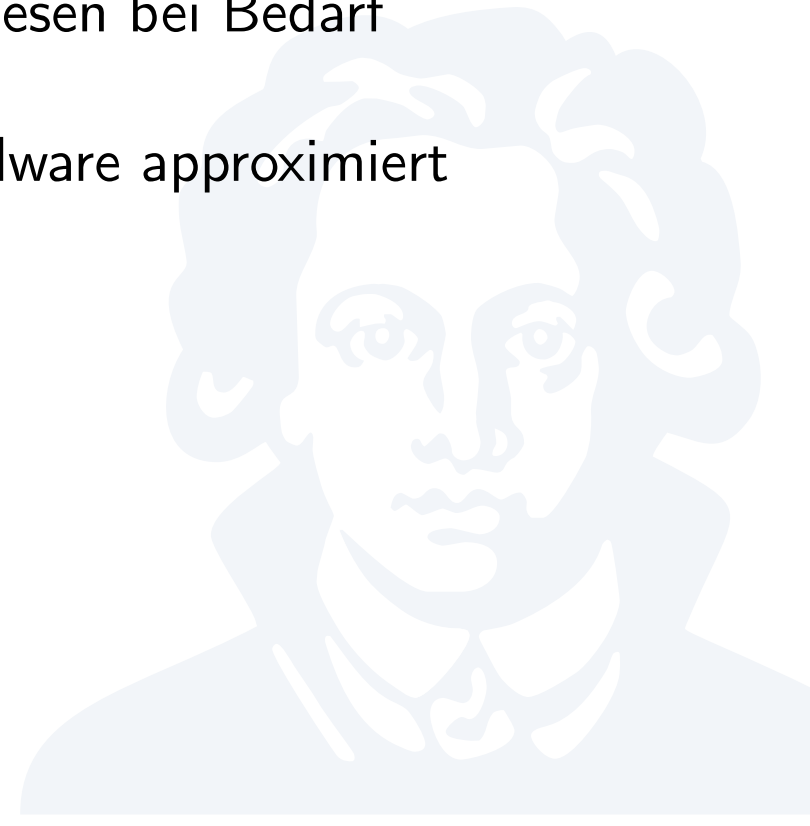
Ersetzungsstrategien

- LRU (*least recently used*):
  - Wir merken uns, welcher Eintrag am längsten nicht verwendet wurde und löschen diesen bei Bedarf
  - Einfach bei 2 Einträgen



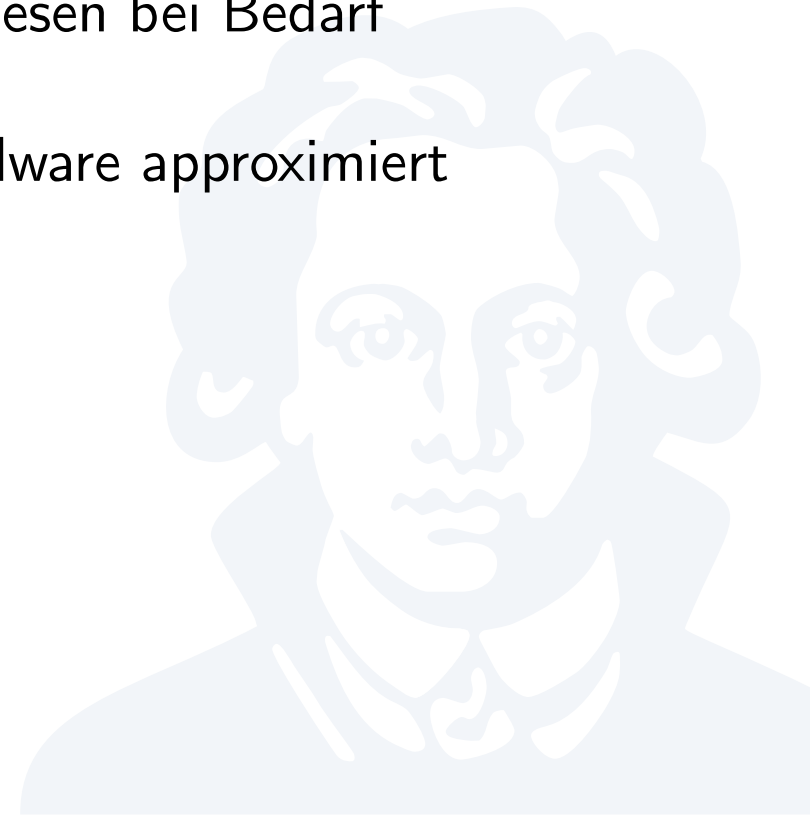
Ersetzungsstrategien

- LRU (*least recently used*):
  - Wir merken uns, welcher Eintrag am längsten nicht verwendet wurde und löschen diesen bei Bedarf
  - Einfach bei 2 Einträgen
  - Bei mehr Einträgen ggf. in Hardware approximiert



Ersetzungsstrategien

- LRU (*least recently used*):
  - Wir merken uns, welcher Eintrag am längsten nicht verwendet wurde und löschen diesen bei Bedarf
  - Einfach bei 2 Einträgen
  - Bei mehr Einträgen ggf. in Hardware approximiert
  - Alternative: Zufällige Ersetzung



Ersetzungsstrategien

- LRU (*least recently used*):
  - Wir merken uns, welcher Eintrag am längsten nicht verwendet wurde und löschen diesen bei Bedarf
  - Einfach bei 2 Einträgen
  - Bei mehr Einträgen ggf. in Hardware approximiert
  - Alternative: Zufällige Ersetzung

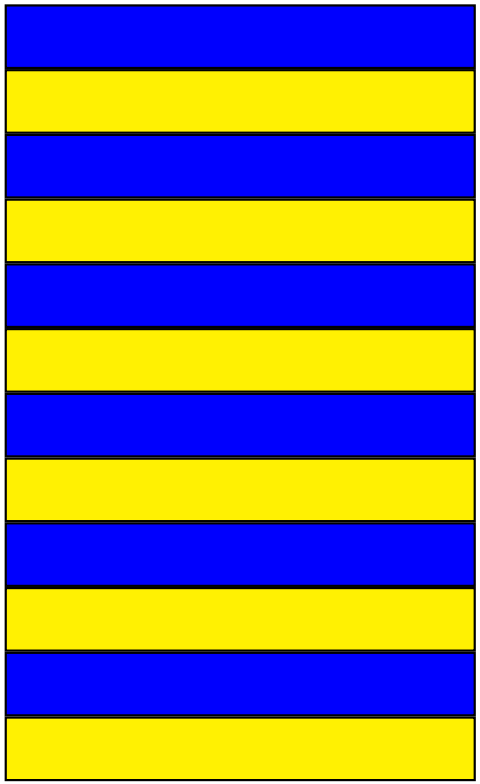
Bei großen vollassoziativen Caches ist zufällige Ersetzung meist fast so effektiv wie LRU

k-fach satzassoziative Caches

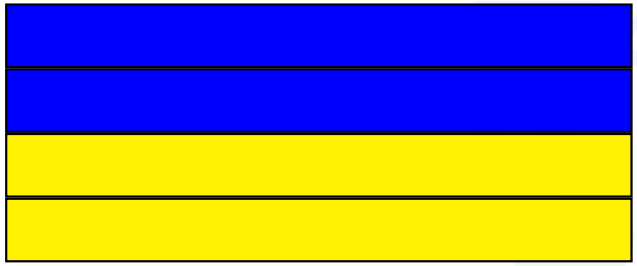
Speicherzuordnung bei *k-fach satzassoziativen Caches*

- Jede Speicheradresse darf in einer von k Cachezeile sitzen (hier $k = 2$)

Speicher S

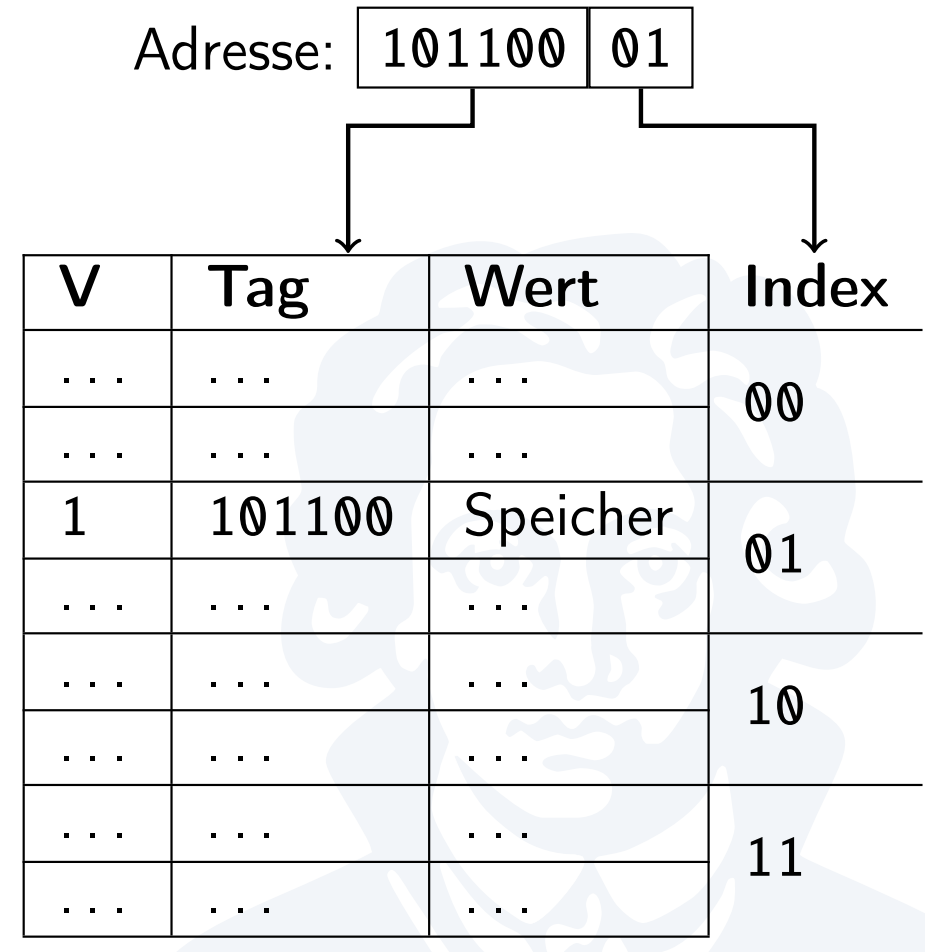


Cache



k-fach satzassoziative Caches

- Kompromiß zwischen vollassoziativ und direkt abgebildet
- Beispiel: 2fach satzassoziativ, 8 Einträge
- Adressbits in *Tag*, *Index* geteilt
- 2^n Einträge, 2^m-fach satzassoziativ:
  - Hintere $n - m$ Bits der Adresse sind Index
  - Rest ist Tag
- Index verweist auf einen *Satz* von $k = 2^m$ Einträgen
- Innerhalb von Sätzen wird wie bei vollassoziativem Cache gearbeitet



Beispiel: 2-fach satzassoziativer Cache

```
⇒   lbu $t0, 0b10110001($0)
    lbu $t0, 0b10001110($0)
    lbu $t0, 0b10110001($0)
    lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Index
0	...	...	00
0	...	...	
0	...	...	01
0	...	...	
0	...	...	10
0	...	...	
0	...	...	11
0	...	...	

Beispiel: 2-fach satzassoziativer Cache

```
    lbu $t0, 0b10110001($0)    Fehlz.
⇒   lbu $t0, 0b10001110($0)
    lbu $t0, 0b10110001($0)
    lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Index
0	...	...	00
0	...	...	
1	101100	Sp.	**01**
0	...	...	
0	...	...	10
0	...	...	
0	...	...	11
0	...	...	

Beispiel: 2-fach satzassoziativer Cache

```
    lbu $t0, 0b10110001($0)    Fehlz.
    lbu $t0, 0b10001110($0)    Fehlz.
⇒   lbu $t0, 0b10110001($0)
    lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Index
0	...	...	00
0	...	...	
1	101100	Sp.	01
0	...	...	
1	100011	Sp.	10
0	...	...	
0	...	...	11
0	...	...	

Beispiel: 2-fach satzassoziativer Cache

```
    lbu $t0, 0b10110001($0)    Fehlz.
    lbu $t0, 0b10001110($0)    Fehlz.
    lbu $t0, 0b10110001($0)    Treffer
⇒   lbu $t0, 0b00000001($0)
    lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Index
0	...	...	00
0	...	...	
1	101100	Sp.	01
0	...	...	
1	100011	Sp.	10
0	...	...	
0	...	...	11
0	...	...	

Beispiel: 2-fach satzassoziativer Cache

```
    lbu $t0, 0b10110001($0)    Fehlz.
    lbu $t0, 0b10001110($0)    Fehlz.
    lbu $t0, 0b10110001($0)    Treffer
    lbu $t0, 0b00000001($0)    Fehlz.
⇒   lbu $t0, 0b10110001($0)
```

V	Tag	Wert	Index
0	...	...	00
0	...	...	
1	101100	Sp.	01
1	000000	Sp.	
1	100011	Sp.	10
0	...	...	
0	...	...	11
0	...	...	

Beispiel: 2-fach satzassoziativer Cache

```
lbu $t0, 0b10110001($0)    Fehlz.
lbu $t0, 0b10001110($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer
lbu $t0, 0b00000001($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer.
```

- 2 Treffer
- 3 Fehlzugriffe

V	Tag	Wert	Index
0	...	...	00
0	...	...	
1	101100	Sp.	01
1	000000	Sp.	
1	100011	Sp.	10
0	...	...	
0	...	...	11
0	...	...	

Beispiel: 2-fach satzassoziativer Cache

```
lbu $t0, 0b10110001($0)    Fehlz.
lbu $t0, 0b10001110($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer
lbu $t0, 0b00000001($0)    Fehlz.
lbu $t0, 0b10110001($0)    Treffer.
```

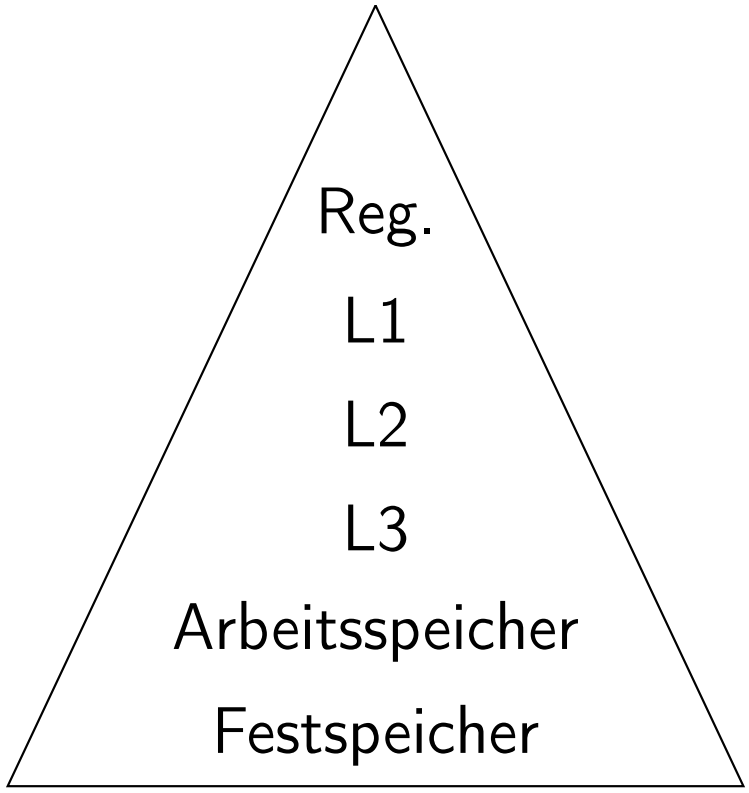
- 2 Treffer
- 3 Fehlzugriffe

V	Tag	Wert	Index
0	...	...	00
0	...	...	
1	101100	Sp.	01
1	000000	Sp.	
1	100011	Sp.	10
0	...	...	
0	...	...	11
0	...	...	

Durchschnittliche Zugriffszeit $= 4 + \frac{3}{5} \times 200 = 124$

Cache-Architektur

- Mehrere Cache-Stufen, z.B.:

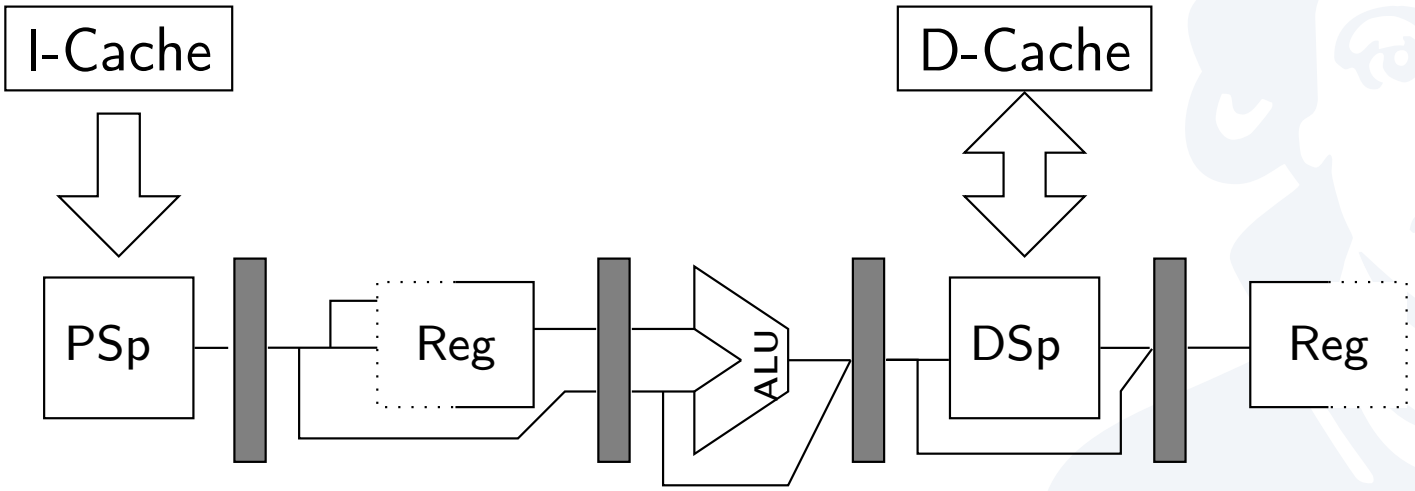


Speicherplatz	**Zugriffszeit**
$\approx$ 1 kiB	sofort
$\approx$ 64 kiB	$\approx$ 4 Zyklen
$\approx$ 256 kiB	$\approx$ 10 Zyklen
$\approx$ 8 MiB	$\approx$ 40–75 Zyklen
$\approx$ 512 GiB	$\approx$ 200 Zyklen
groß	$>$ 100000 Zykler

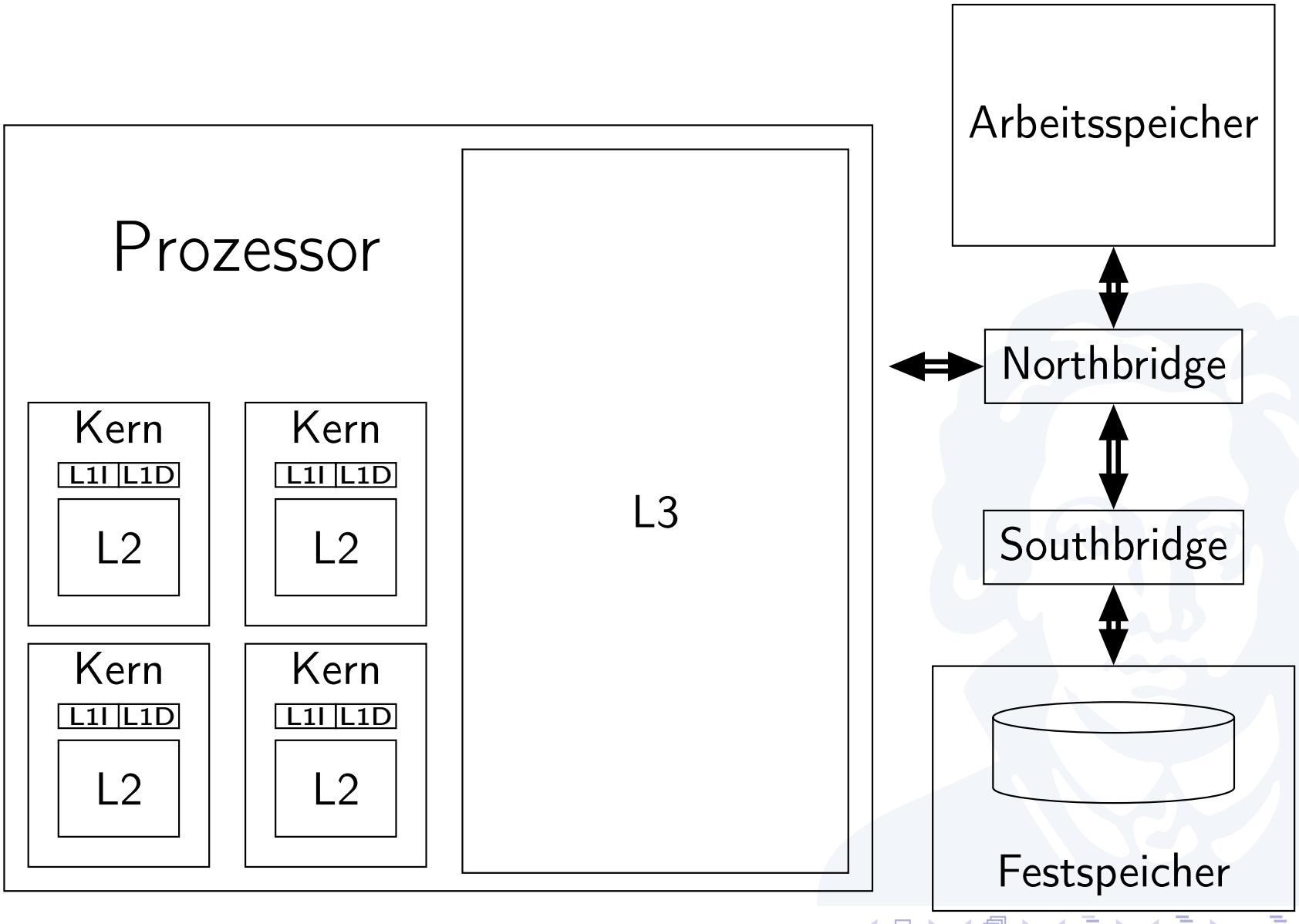
- L1-Cache wird meist getrennt in:
  - Instruktionscache (I-Cache)
  - Datencache (D-Cache)

Modifizierte Harvard-Architektur

- Trennung in Instruktions- und Datencaches ergibt Struktur wie in Harvard-Architektur
- Lesen, Schreiben, Ausführen prinzipiell mit allen Speicheradressen weiter möglich
- Ausführung fließt durch I-Cache
- Lesen/Schreiben fließt durch D-Cache

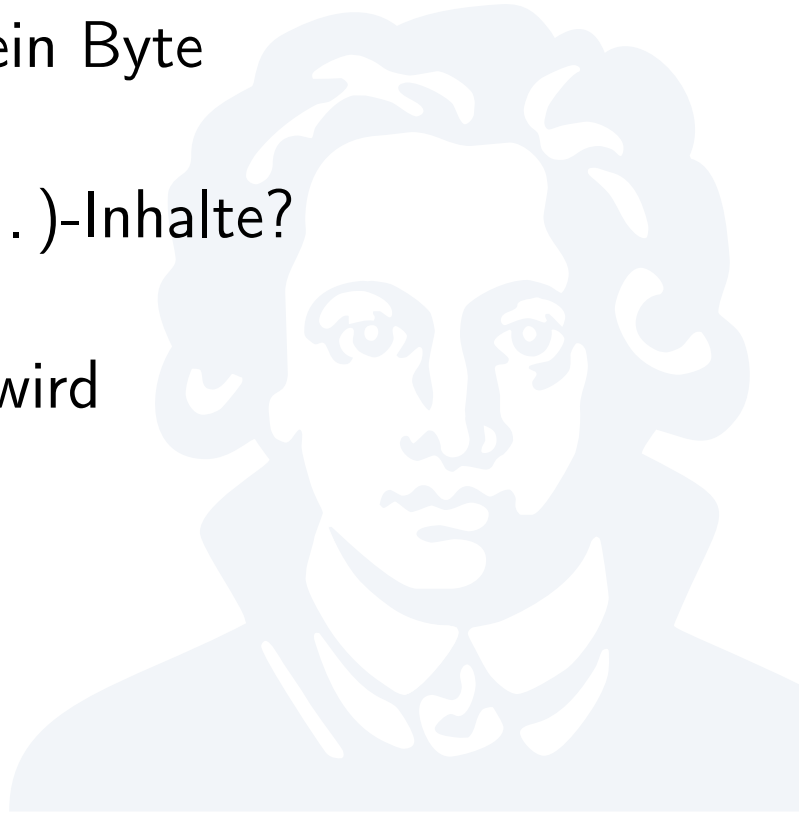


Speicherlayout mit Caches (Beispiel)



Weitere Cache-Details

- *Zeilengröße*:
  Cachezeilen sind meist größer als ein Byte
- *Datenteilung zwischen Caches*:
  Dupliziert L1 die L2 (L2 die L3, . . .)-Inhalte?
- *Prefetching*:
  Speicher laden, bevor er benötigt wird

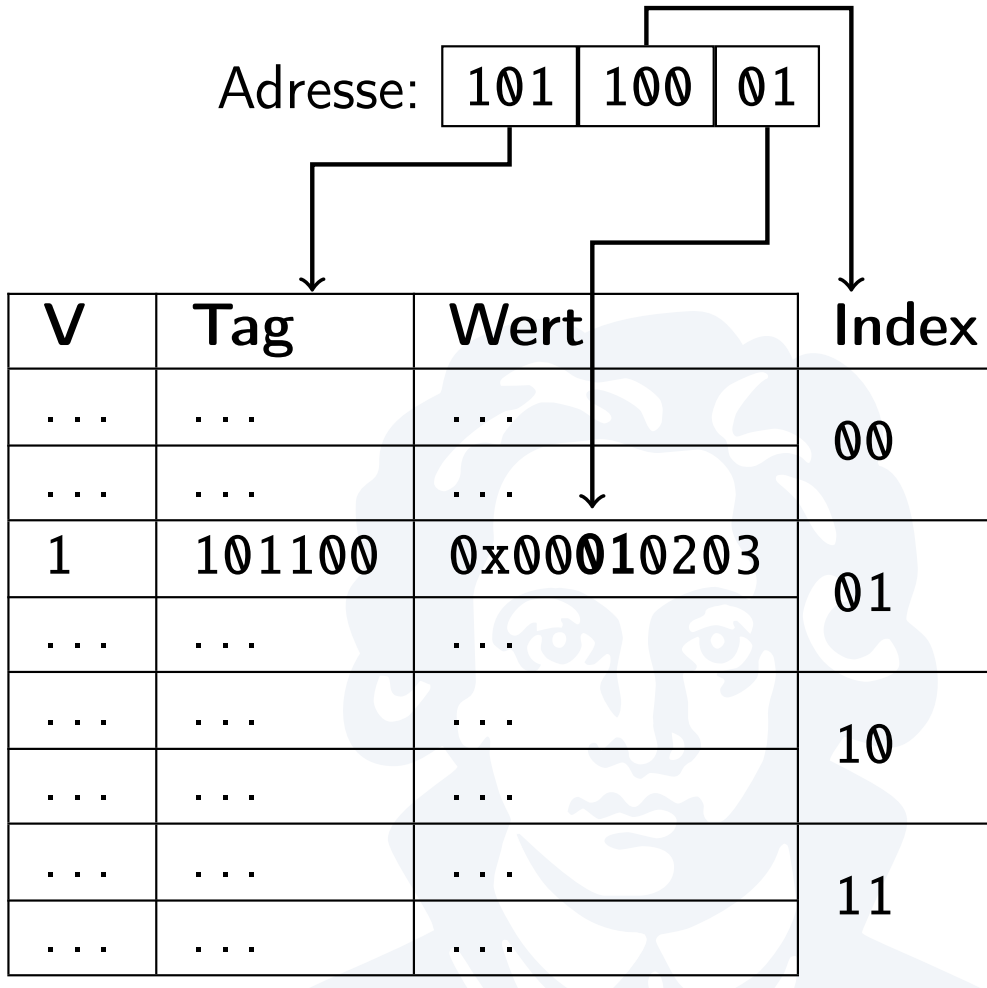


Zeilengröße

- Caches wie bisher gezeigt nutzen *temporale Lokalität*
- Nutzung von *räumlicher Lokalität*:
  - Ein Rechner mit 64-Bit-Registern schreibt/liest meist 32 oder 64 Bit (n^k) gleichzeitig
  - Wir können mehrere Bytes pro Cache-Zeile speichern

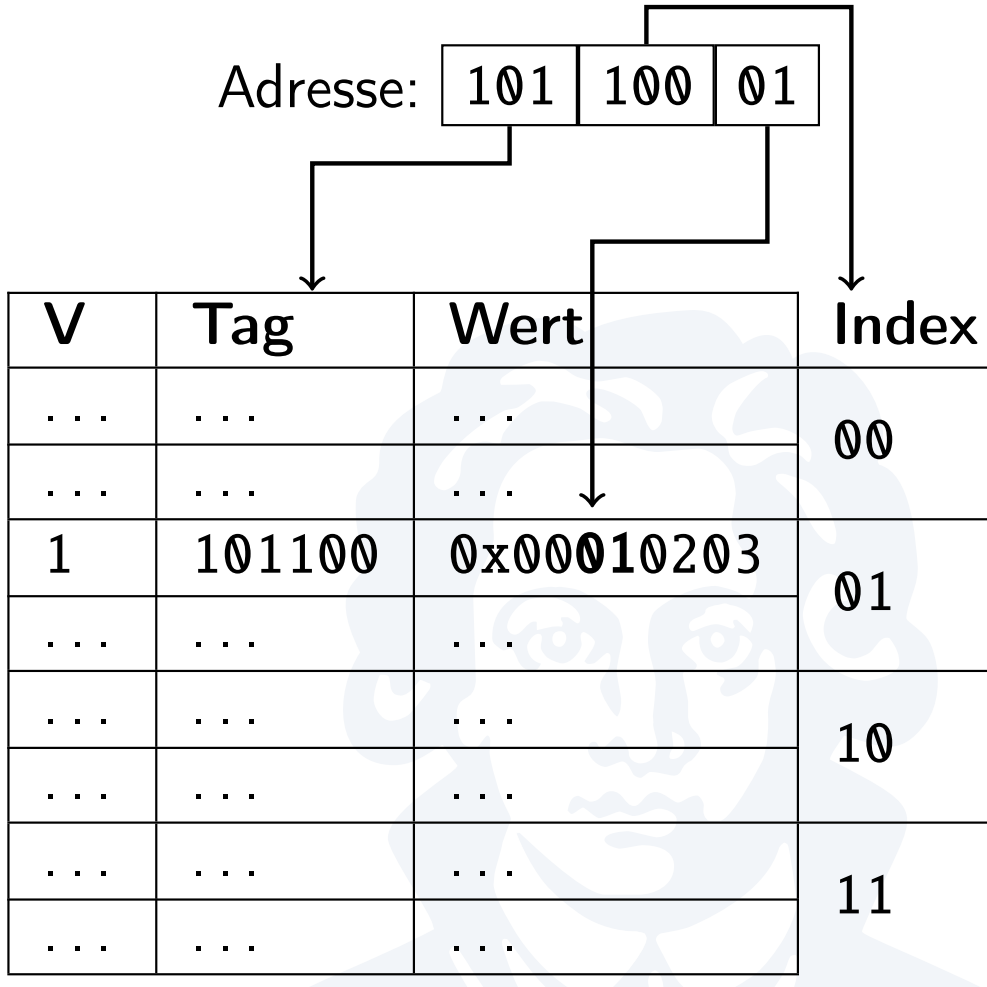
Zeilengröße

- Caches wie bisher gezeigt nutzen *temporale Lokalität*
- Nutzung von *räumlicher Lokalität*:
  - Ein Rechner mit 64-Bit-Registern schreibt/liest meist 32 oder 64 Bit (n^k) gleichzeitig
  - Wir können mehrere Bytes pro Cache-Zeile speichern



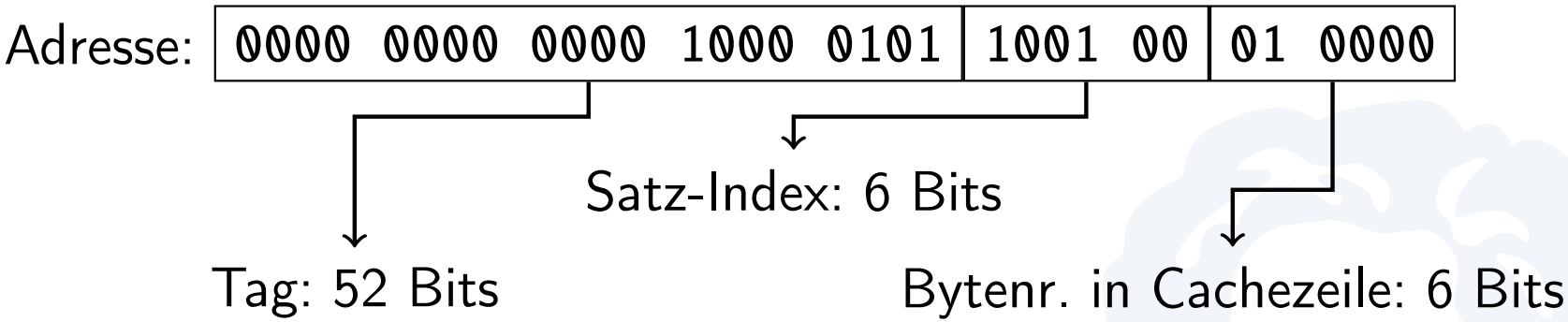
Zeilengröße

- Caches wie bisher gezeigt nutzen *temporale Lokalität*
- Nutzung von *räumlicher Lokalität*:
  - Ein Rechner mit 64-Bit-Registern schreibt/liest meist 32 oder 64 Bit (n^k) gleichzeitig
  - Wir können mehrere Bytes pro Cache-Zeile speichern
- Hintere k bit werden als Indizes in Werteintrag in Cachezeile verwendet



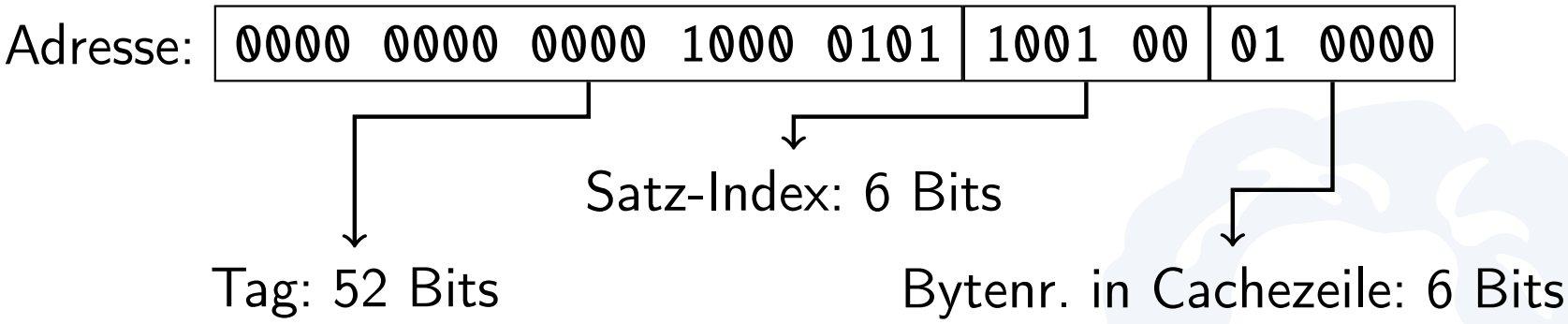
Realistischer satzassoziativer L1-Cache

Haswell, L1 Datencache:



Realistischer satzassoziativer L1-Cache

Haswell, L1 Datencache:



- 2^{15} Bytes (= 32 kiB) insgesamt im Cache
- 64 Bytes Cachezeilengröße
- 64 Cachesätze
- 8−fach satzassoziativ

$\Rightarrow$ 512 Cache-Zeilen

Datenteilung zwischen Caches

Inklusive Cacheteilung

- Daten in L1 müssen auch in L2 liegen
- Daten in L2 müssen auch in L3 liegen
- Vereinfacht Kommunikation zwischen Prozessorkernen
- (z.B. Intel seit Core)

L1	L2
0100	0100
0E20	0E20
	0730
	0050

Exklusive Cacheteilung

- Daten dürfen nicht gleichzeitig in mehreren Caches liegen
- Nutzt Cache-Speicher besser aus
- (z.B. AMD Athlon)

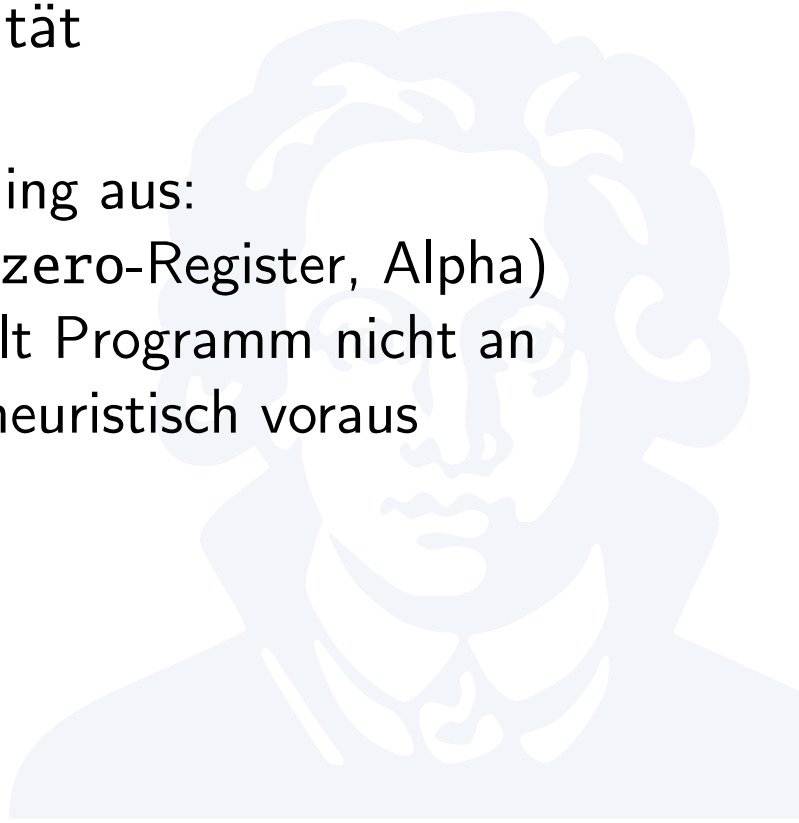
L1	L2
0100	AF90
0E20	AF80
	0730
	0050

Prefetching

- Wenn wir vorhersagen könnten, welche Daten gelesen werden müssen, könnten wir Caches weiter beschleunigen
- *Prefetching*: Laden von Werten in den Cache, bevor sie benötigt werden
- Bessere Nutzung räumlicher Lokalität

Prefetching

- Wenn wir vorhersagen könnten, welche Daten gelesen werden müssen, könnten wir Caches weiter beschleunigen
- *Prefetching*: Laden von Werten in den Cache, bevor sie benötigt werden
- Bessere Nutzung räumlicher Lokalität
- Varianten:
  - Explizit: Programm löst prefetching aus:
    `uldl zero, 0(t2)` (Lesen in `$zero`-Register, Alpha)
    Prozessor läd Wert in Cache, hält Programm nicht an
  - Implizit: Prozessor sagt Zugriff heuristisch voraus



Prefetching

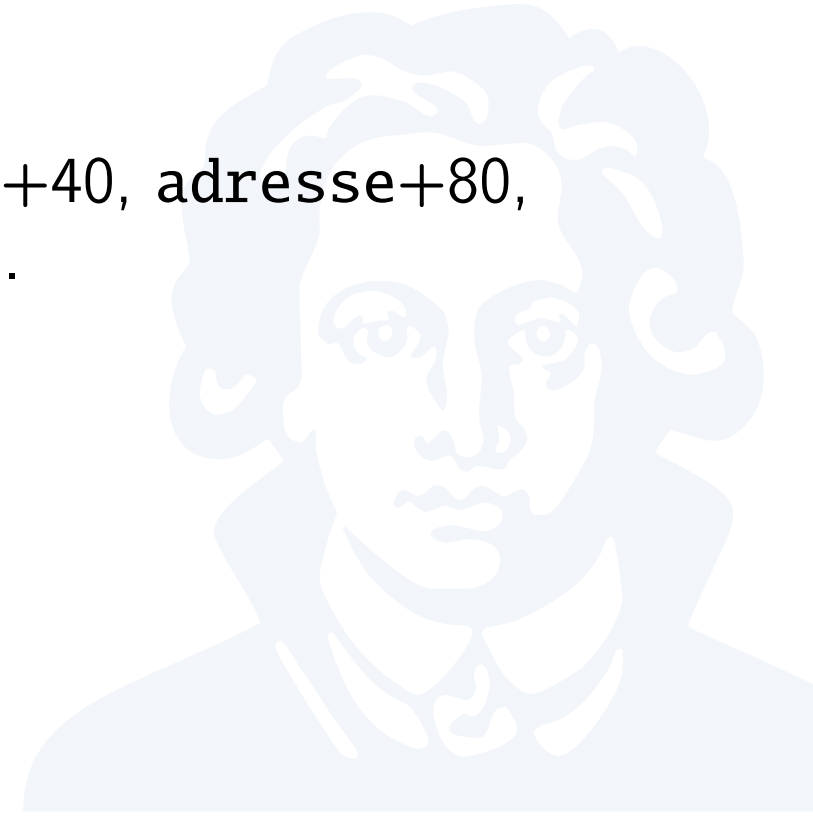
- Wenn wir vorhersagen könnten, welche Daten gelesen werden müssen, könnten wir Caches weiter beschleunigen
- *Prefetching*: Laden von Werten in den Cache, bevor sie benötigt werden
- Bessere Nutzung räumlicher Lokalität
- Varianten:
  - Explizit: Programm löst prefetching aus:
    `uldl zero, 0(t2)` (Lesen in `$zero`-Register, Alpha)
    Prozessor läd Wert in Cache, hält Programm nicht an
  - Implizit: Prozessor sagt Zugriff heuristisch voraus
- Herausforderungen:
  - Falsche Adresse geladen: Speicherbandbreite/Cachezeile verschwendet
  - Zu früh geladen: Wert evtl. gelöscht vor Nutzung
  - Zu spät geladen: Prefetch hilft nur teilweise

k-Schritt prefetching

- Häufiges Speicherzugriffsmuster:

```
        la $t0, adresse
loop:   lw 0($t0)
        addi $t0, $t0, 40
        ...
        j loop
```

- Wir lesen von `adresse`, `adresse`+40, `adresse`+80, `adresse`+120, `adresse`+160, ...

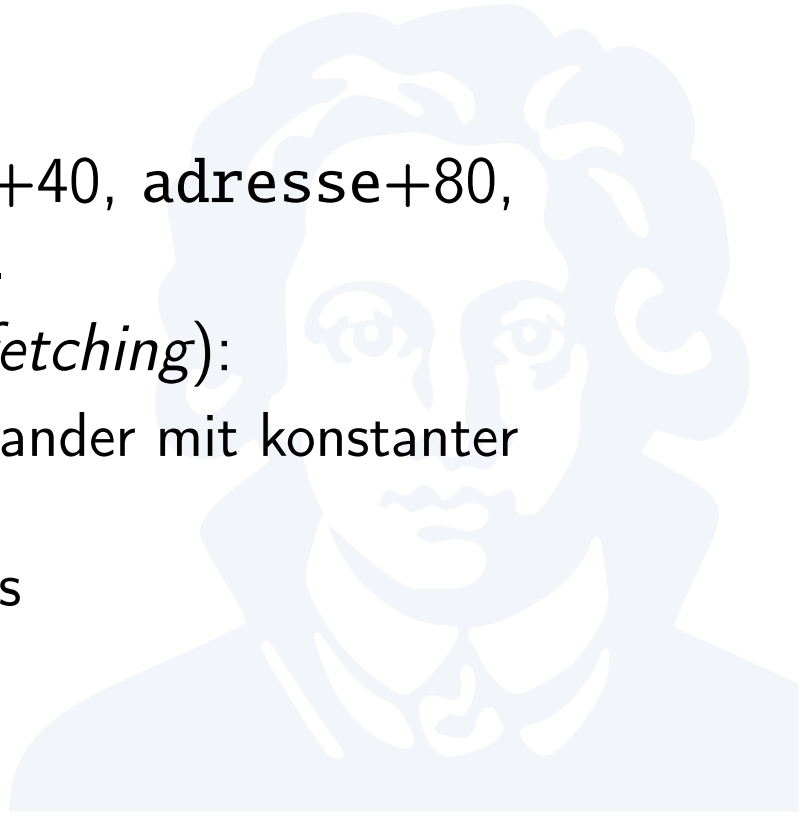


k-Schritt prefetching

- Häufiges Speicherzugriffsmuster:

```
        la $t0, adresse
loop:   lw 0($t0)
        addi $t0, $t0, 40
        ...
        j loop
```

- Wir lesen von `adresse`, `adresse`+40, `adresse`+80, `adresse`+120, `adresse`+160, ...
- k-Schritt prefetching (*k-stride prefetching*):
  - Erkennt Lesezugriffe nahe beieinander mit konstanter Schrittgröße
  - Löst automatische prefetches aus



k-Schritt prefetching

- Häufiges Speicherzugriffsmuster:

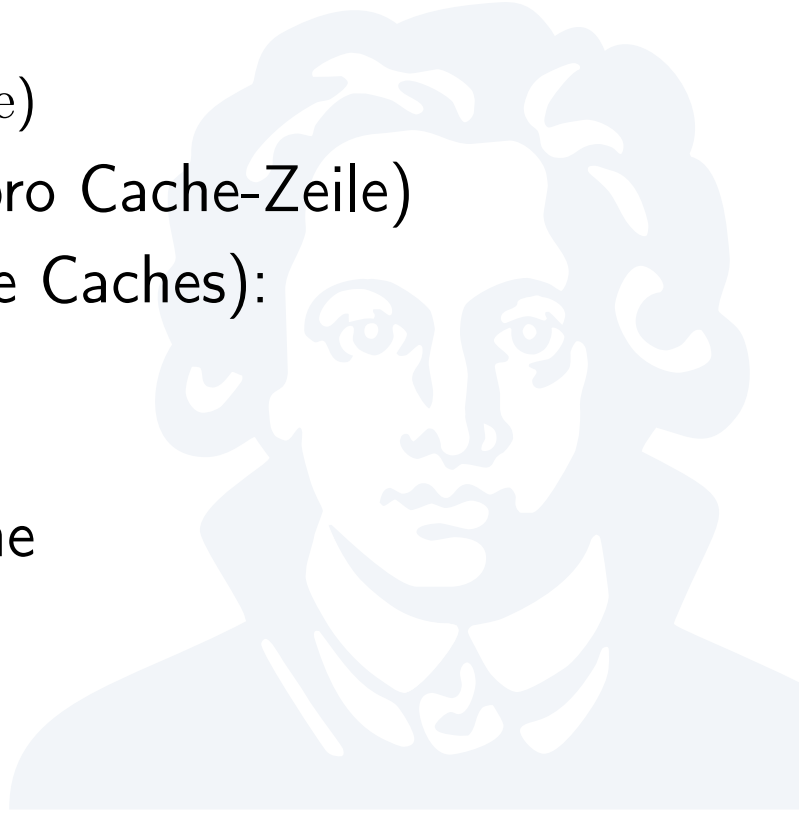
```
        la $t0, adresse
loop:   lw 0($t0)
        addi $t0, $t0, 40
        ...
        j loop
```

- Wir lesen von `adresse`, `adresse`+40, `adresse`+80, `adresse`+120, `adresse`+160, ...
- k-Schritt prefetching (*k-stride prefetching*):
  - Erkennt Lesezugriffe nahe beieinander mit konstanter Schrittgröße
  - Löst automatische prefetches aus

Von vielen modernen Prozessoren unterstützt

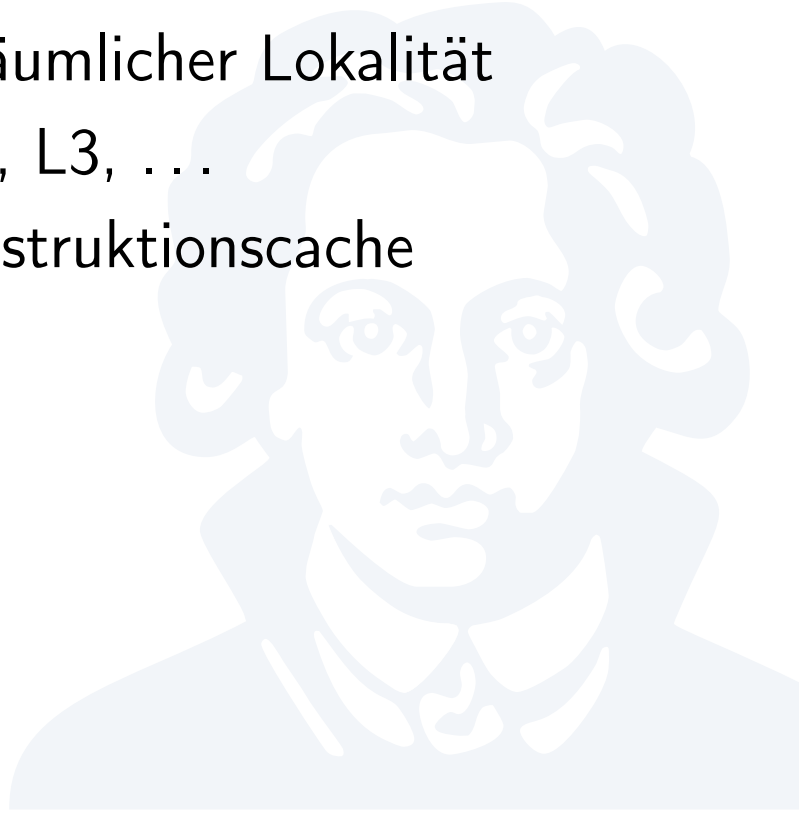
Zusammenfassung: Cache-Eigenschaften

- Cache-Typ
  - k-fach satzassoziativ
    Sonderfälle:
    - Direkt abgebildet ($k = 1$)
    - Vollassoziativ ($k = \#\text{Einträge}$)
- Zeilengröße / Blockgröße (Bytes pro Cache-Zeile)
- Ersetzungsstrategie (für assoziative Caches):
  - LRU
  - Zufällig
- Inklusiver Cache / Exklusiver Cache
- Prefetch ermöglicht frühes Laden

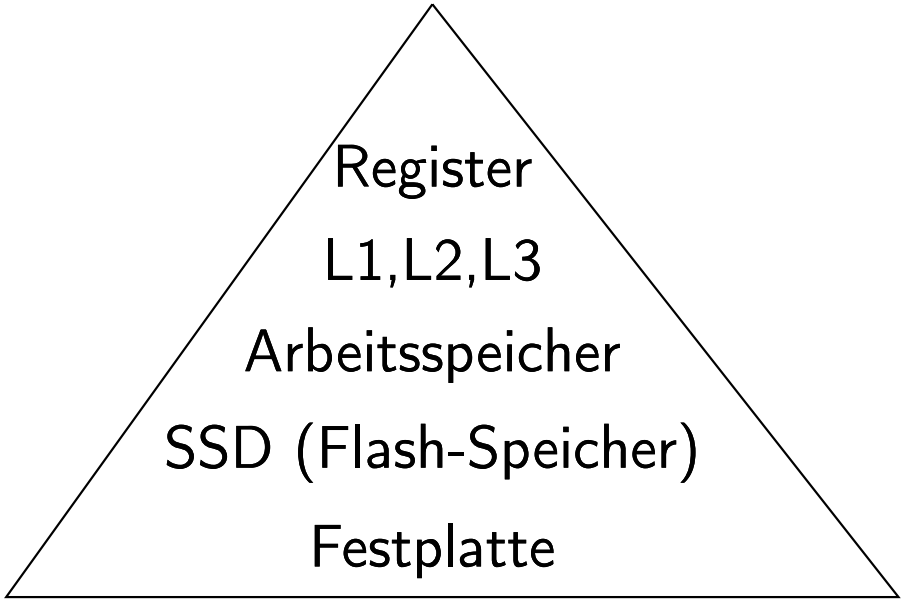


Zusammenfassung: Caches

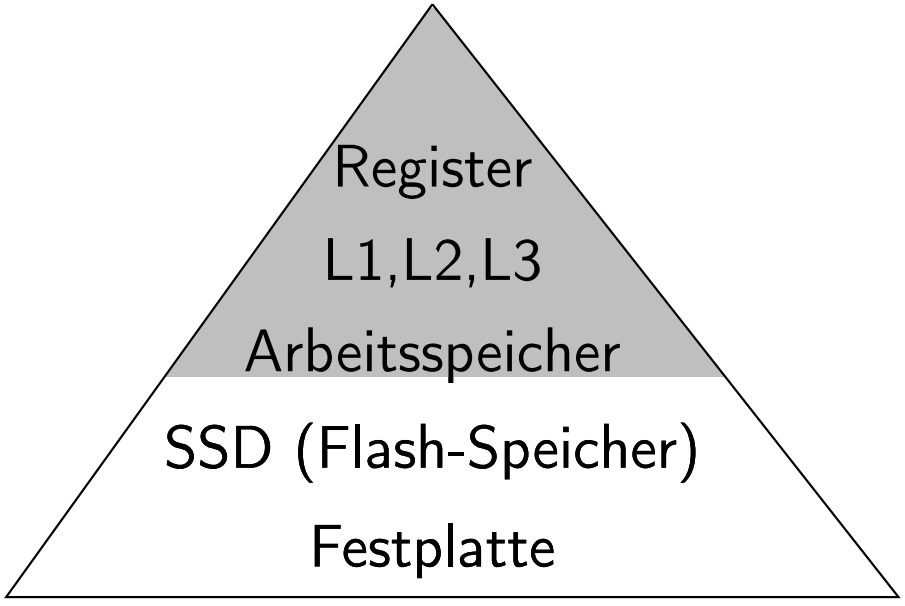
- Ausnutzung von temporaler und räumlicher Lokalität
- Teil der Speicherhierarchie: L1, L2, L3, ...
- Trennung von L1 in Daten- und Instruktionscache



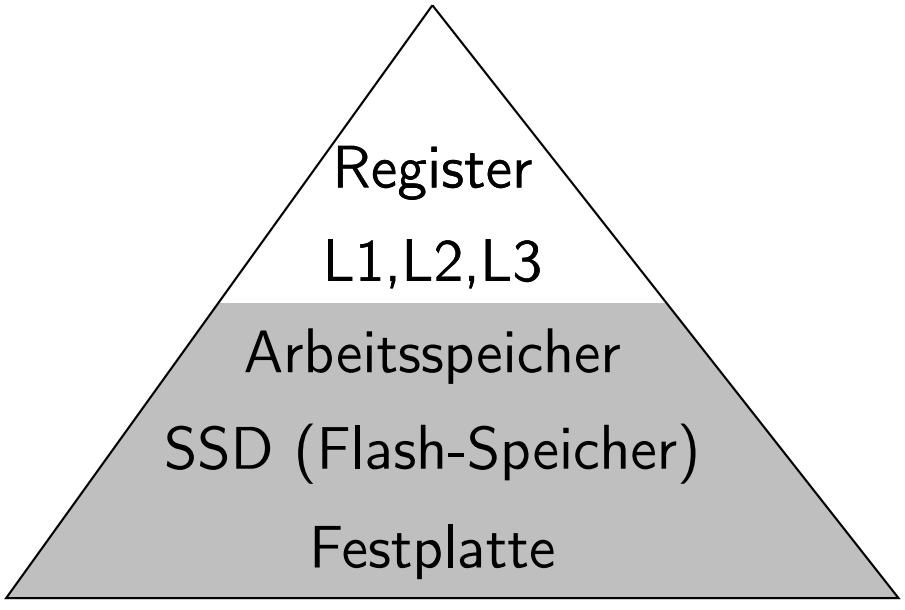
Die Basis der Speicherhierarchie



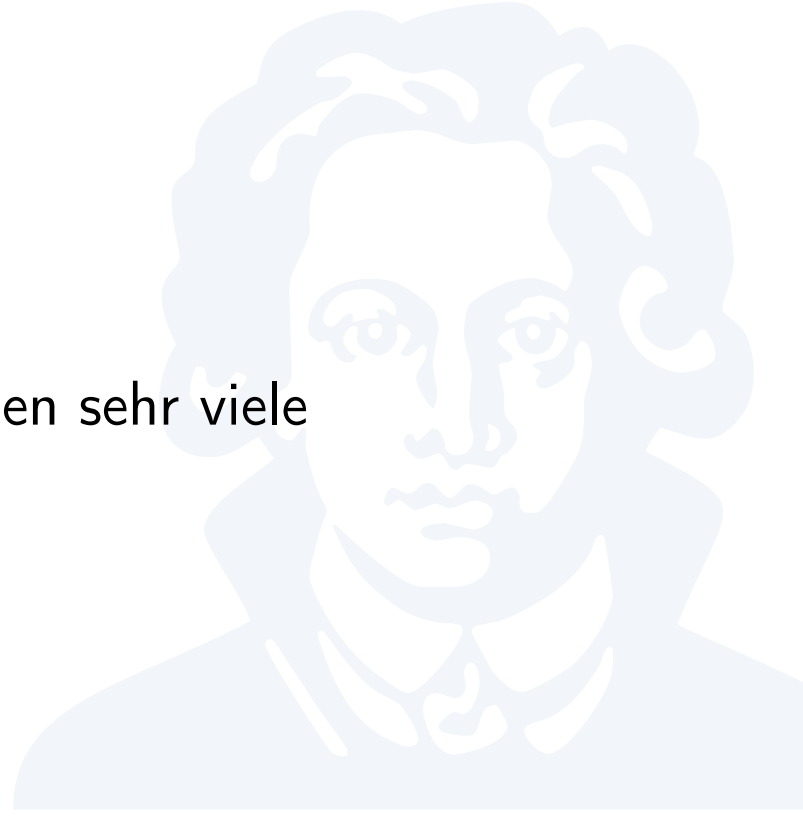
Die Basis der Speicherhierarchie



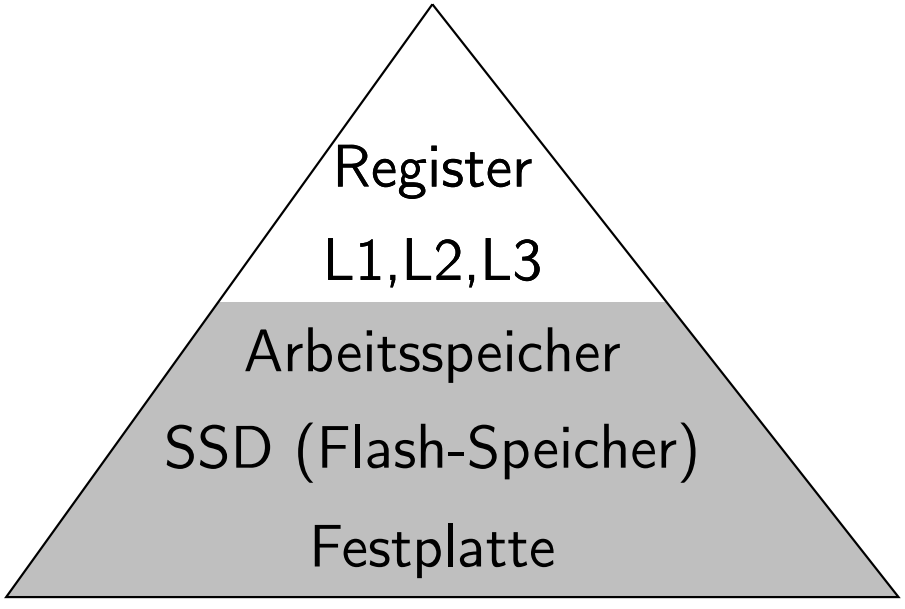
Die Basis der Speicherhierarchie



Basis der Speicherhierarchie: Zugriffe kosten sehr viele (hunderttausende) Zyklen



Die Basis der Speicherhierarchie



Basis der Speicherhierarchie: Zugriffe kosten sehr viele (hunderttausende) Zyklen

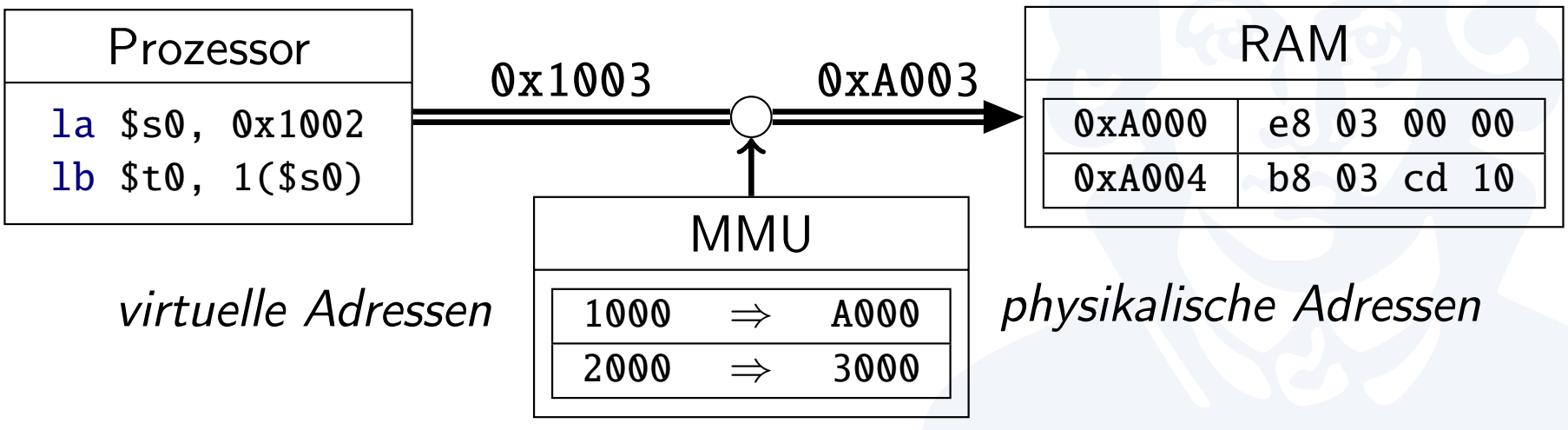
Gutes Caching besonders wichtig. Vorteil: viel Zeit, genug für komplexe Fehlzugriffbehandlung

Caching in der Basis der Speicherhierarchie

- Realisiert durch „Virtuellen Speicher“
- Fehlzugriffbehandlung im Betriebssystem
- Behandlung von Treffern?

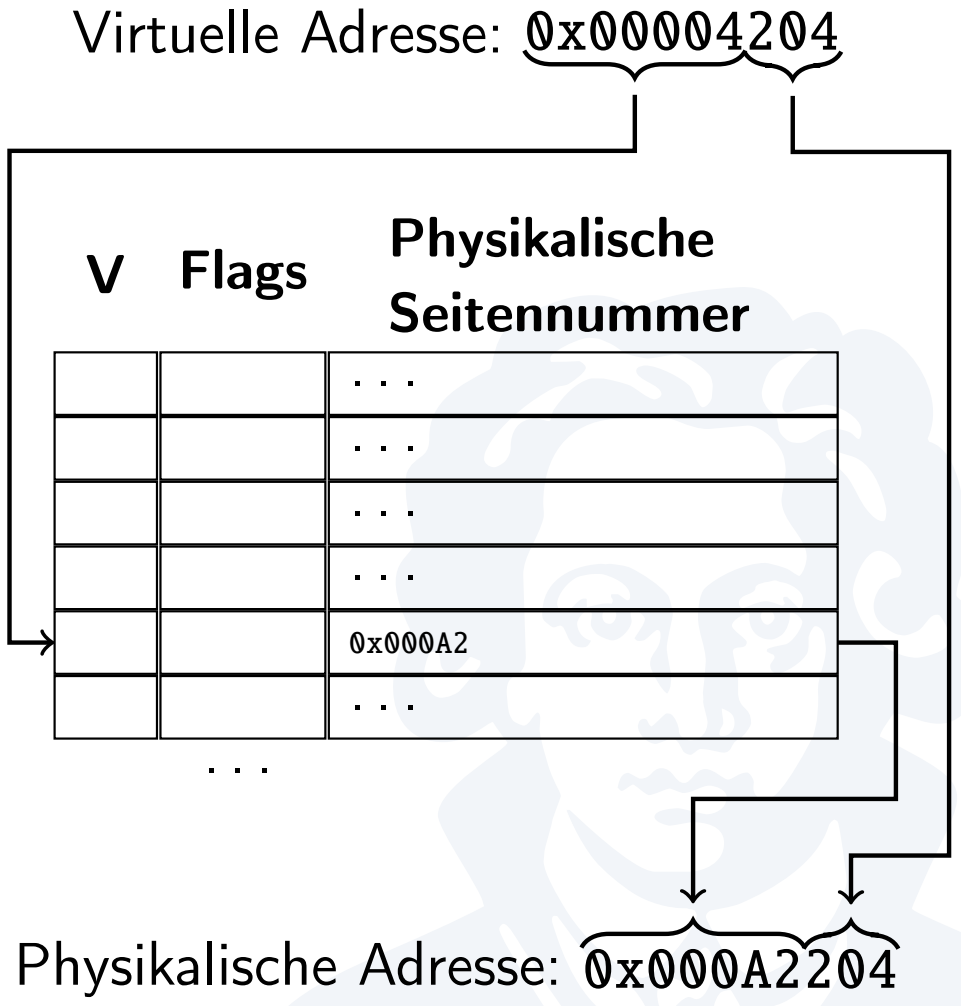
Caching in der Basis der Speicherhierarchie

- Realisiert durch „Virtuellen Speicher“
- Fehlzugriffbehandlung im Betriebssystem
- Behandlung von Treffern
- Speicherverwaltungseinheit (MMU, *memory mgmt. unit*):
  - Übersetzt *virtuelle Adressen* in *physikalische Adressen*
    - *Virtuelle Adresse*: Sicht des Prozessors
    - *Physikalische Adresse*: Sicht des Speicherbusses
  - MIPS: MMU ist Hauptkomponente von Koprozessor #0

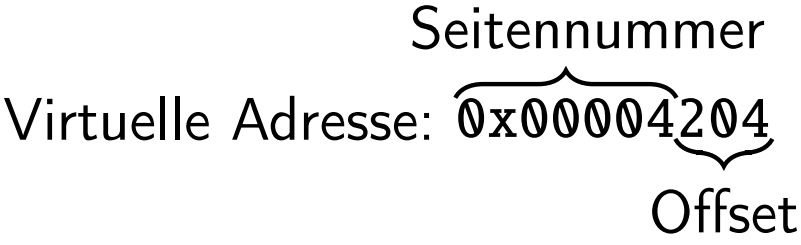


Adreßübersetzung über die Seitentabelle

- Adreßübersetzung durch MMU über *Seitentabelle*
- Seitentabelle liegt im Arbeitsspeicher
- Beginn der Seitentabelle als physikalische Adresse in *Seitentabellenregister*
- Eintrag in Seitentabelle besteht aus:
  - Gültigkeitsbit (**V**)
  - Flags
  - Physikalische Adresse

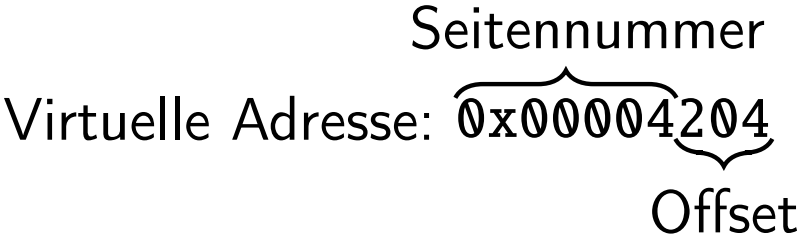


Die Seitentabelle im Kontext



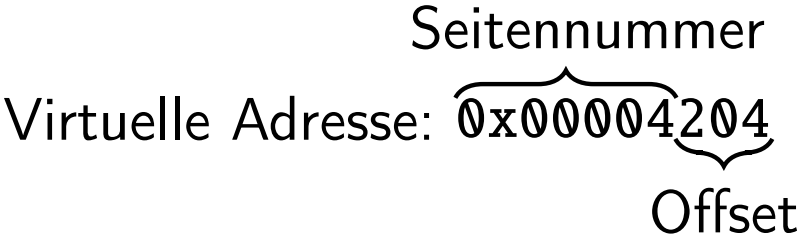
- *Seitengröße* gibt an, wieviele Adreßbits übersetzt werden
  - *Beispiel*: Seitengröße 4kiB = 2^{12}B
  - 12 Bits für Offset, Rest für Seitennummer
- **Flags** speichern Informationen über die Seite:

Die Seitentabelle im Kontext



- *Seitengröße* gibt an, wieviele Adreßbits übersetzt werden
  - *Beispiel*: Seitengröße 4kiB = 2^{12}B
  - 12 Bits für Offset, Rest für Seitennummer
- **Flags** speichern Informationen über die Seite:
  - Darf die Seite *geschrieben*, *gelesen*, *ausgeführt* werden? (MMU löst Ausnahme bei nicht erlaubtem Zugriff aus)

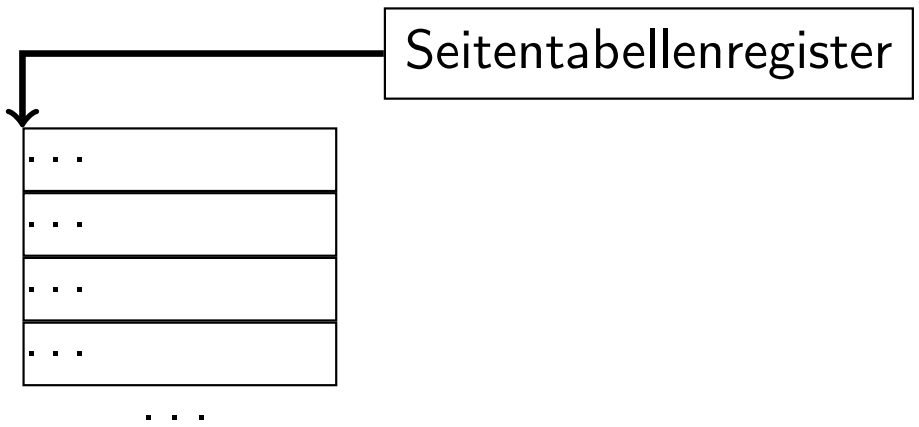
Die Seitentabelle im Kontext



- *Seitengröße* gibt an, wieviele Adreßbits übersetzt werden
  - *Beispiel*: Seitengröße 4kiB = 2^{12}B
  - 12 Bits für Offset, Rest für Seitennummer
- **Flags** speichern Informationen über die Seite:
  - Darf die Seite *geschrieben*, *gelesen*, *ausgeführt* werden? (MMU löst Ausnahme bei nicht erlaubtem Zugriff aus)
  - Wurde die Seite *verwendet* oder sogar *beschrieben*? (MMU aktualisiert diese Statusbits automatisch)

Beispiel: Seitentabelle der DEC Alpha

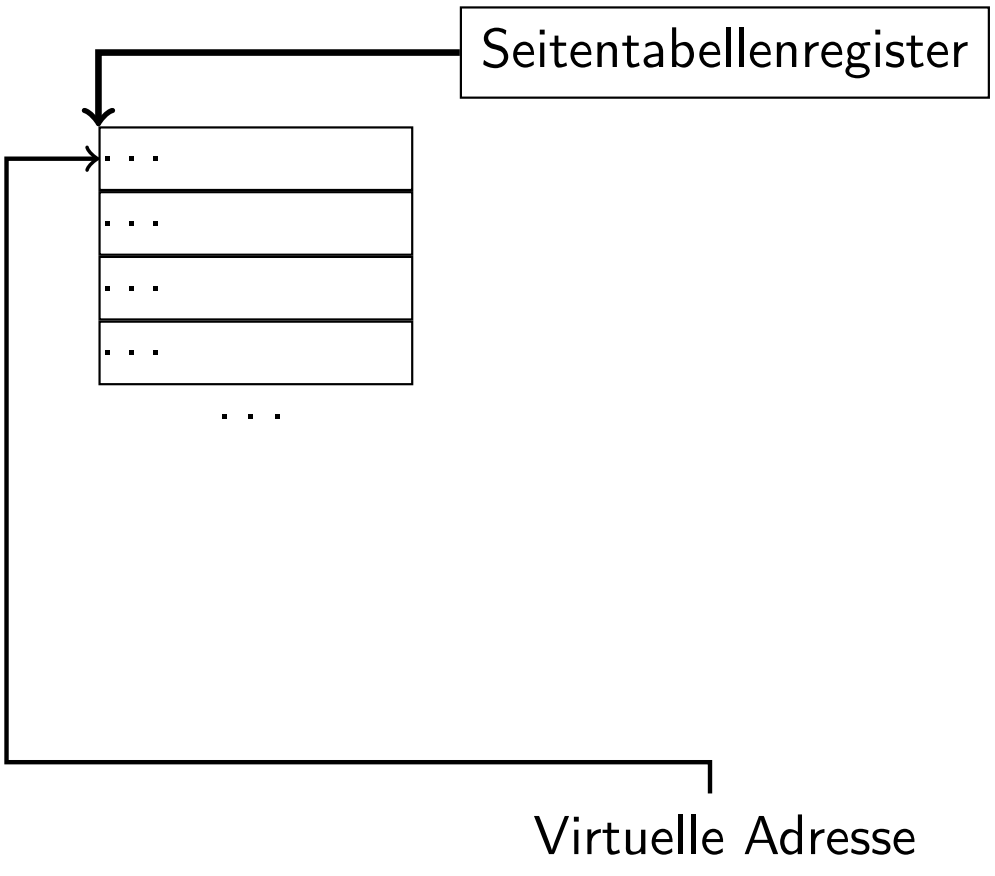
Moderne Prozessoren verwenden Hierarchie von Seitentabellen:



Virtuelle Adresse

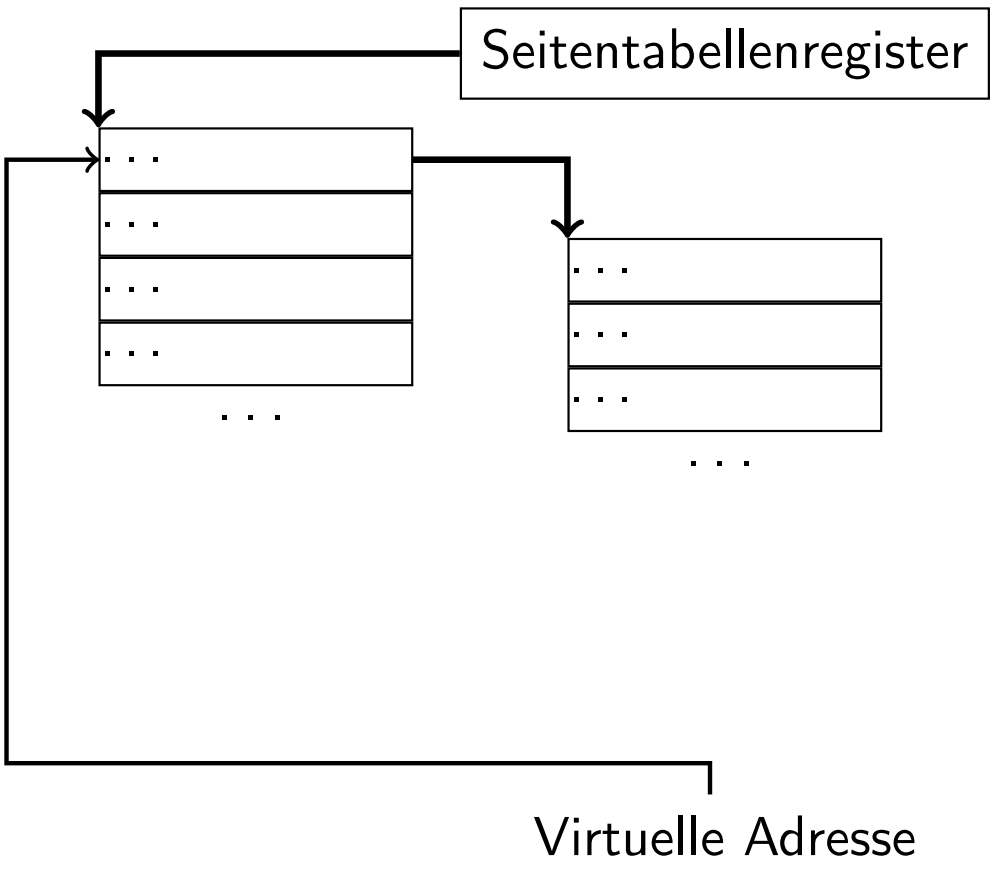
Beispiel: Seitentabelle der DEC Alpha

Moderne Prozessoren verwenden Hierarchie von Seitentabellen:



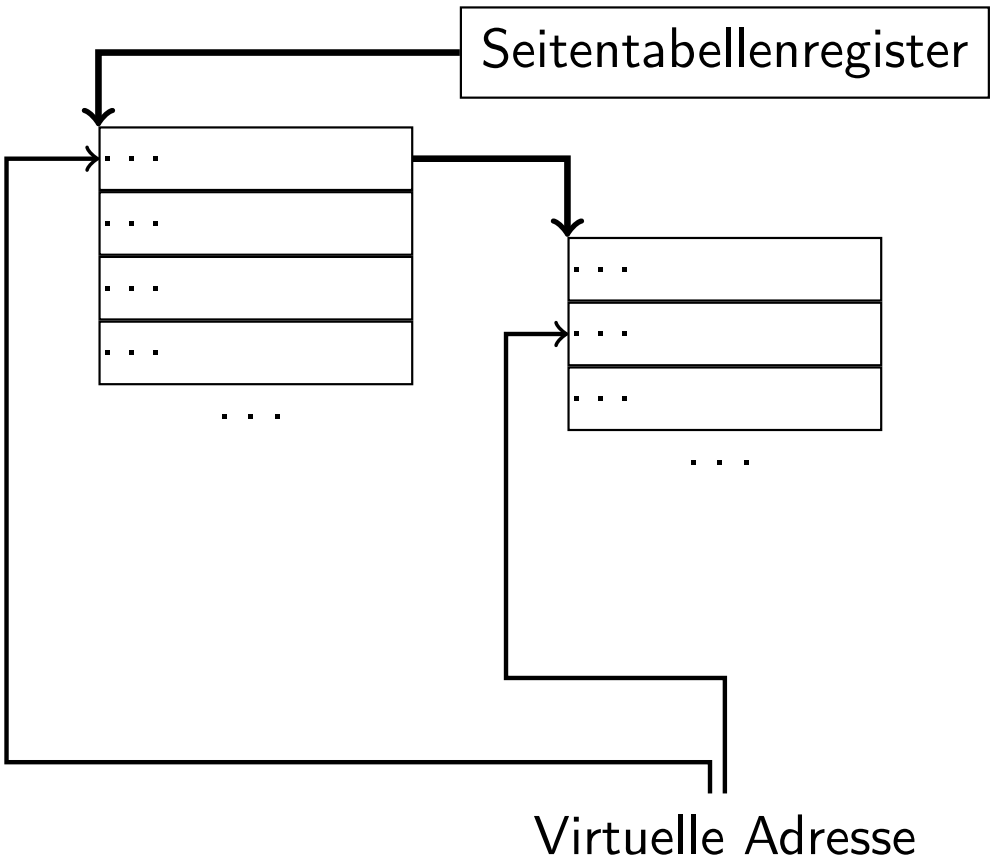
Beispiel: Seitentabelle der DEC Alpha

Moderne Prozessoren verwenden Hierarchie von Seitentabellen:



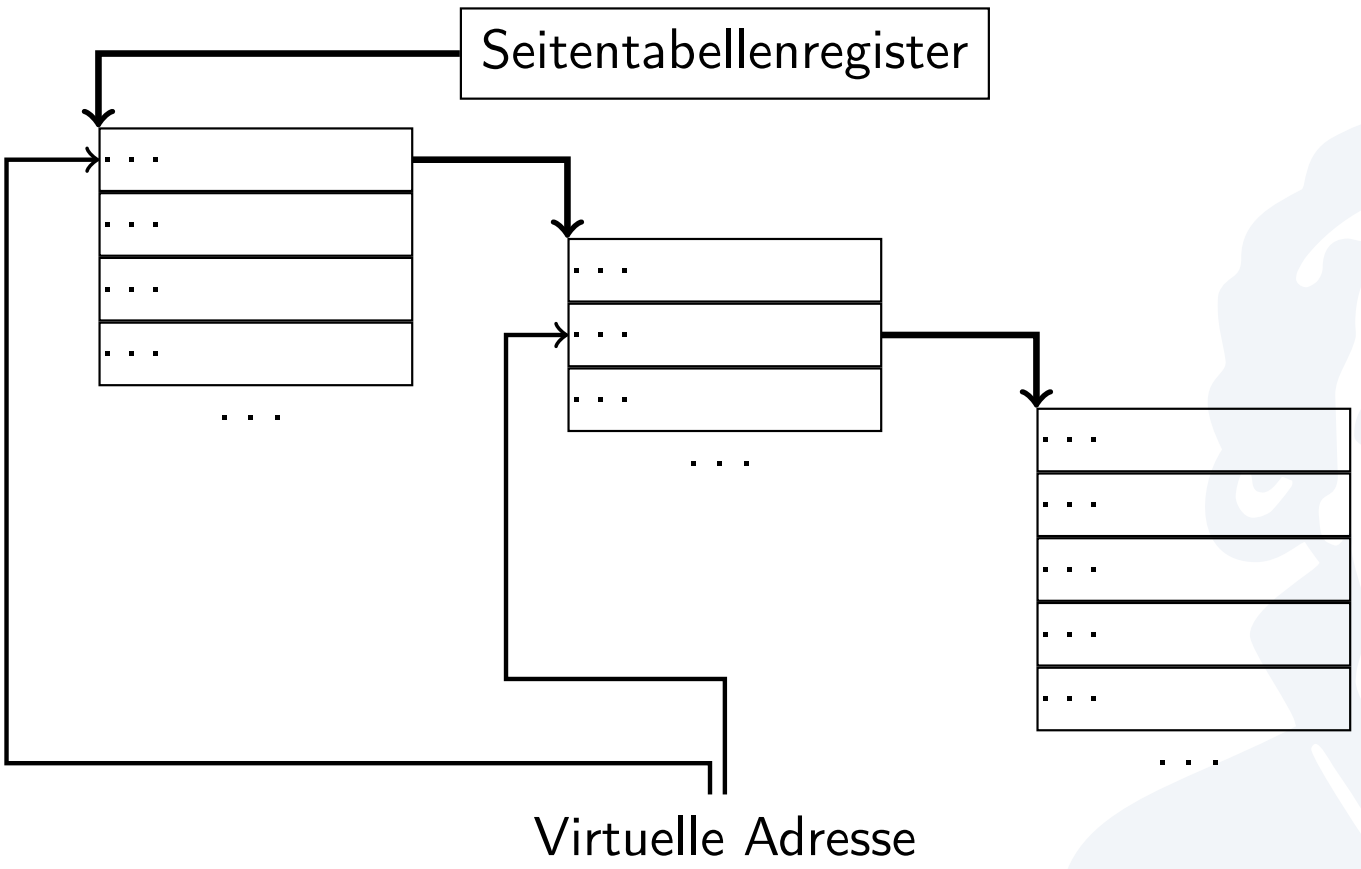
Beispiel: Seitentabelle der DEC Alpha

Moderne Prozessoren verwenden Hierarchie von Seitentabellen:



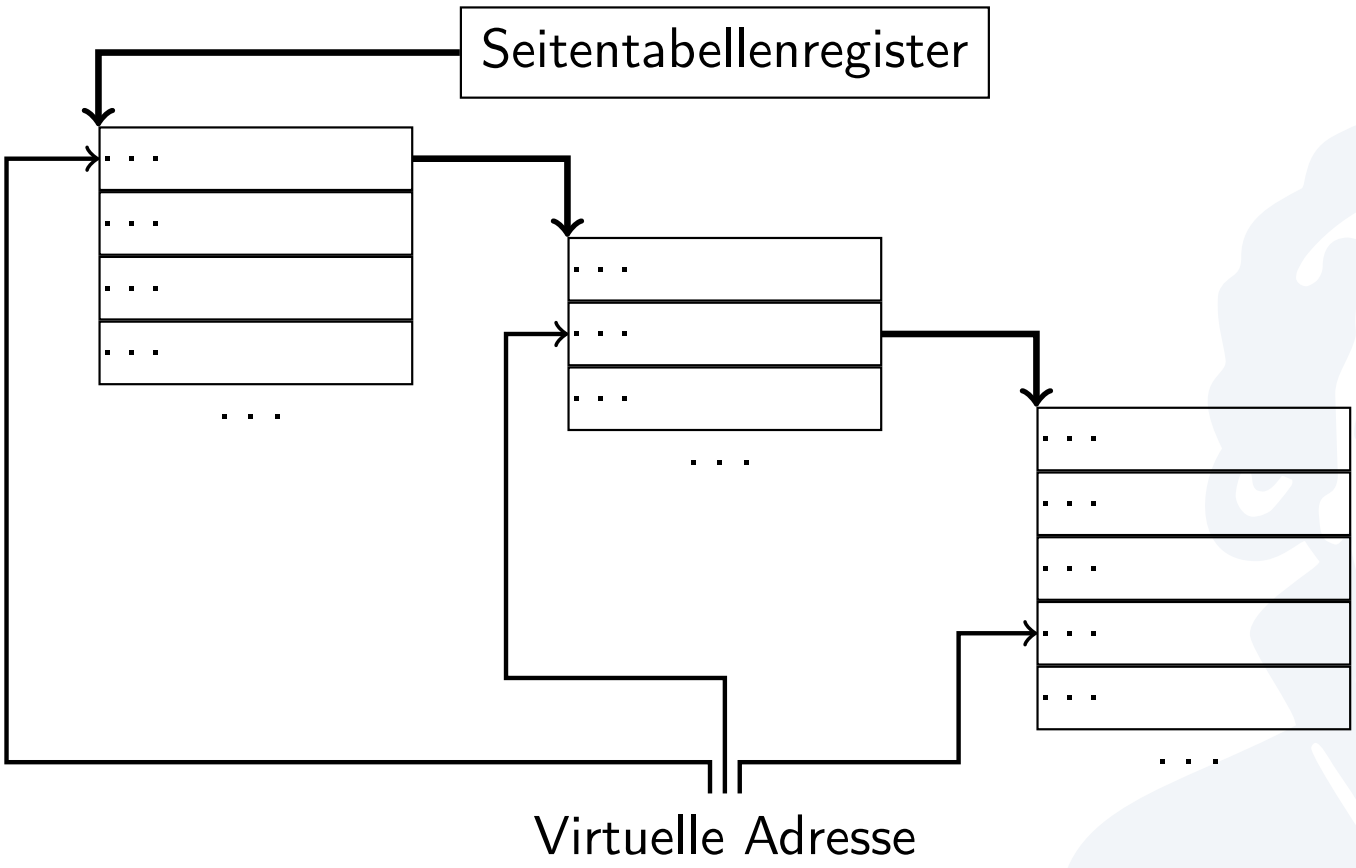
Beispiel: Seitentabelle der DEC Alpha

Moderne Prozessoren verwenden Hierarchie von Seitentabellen:



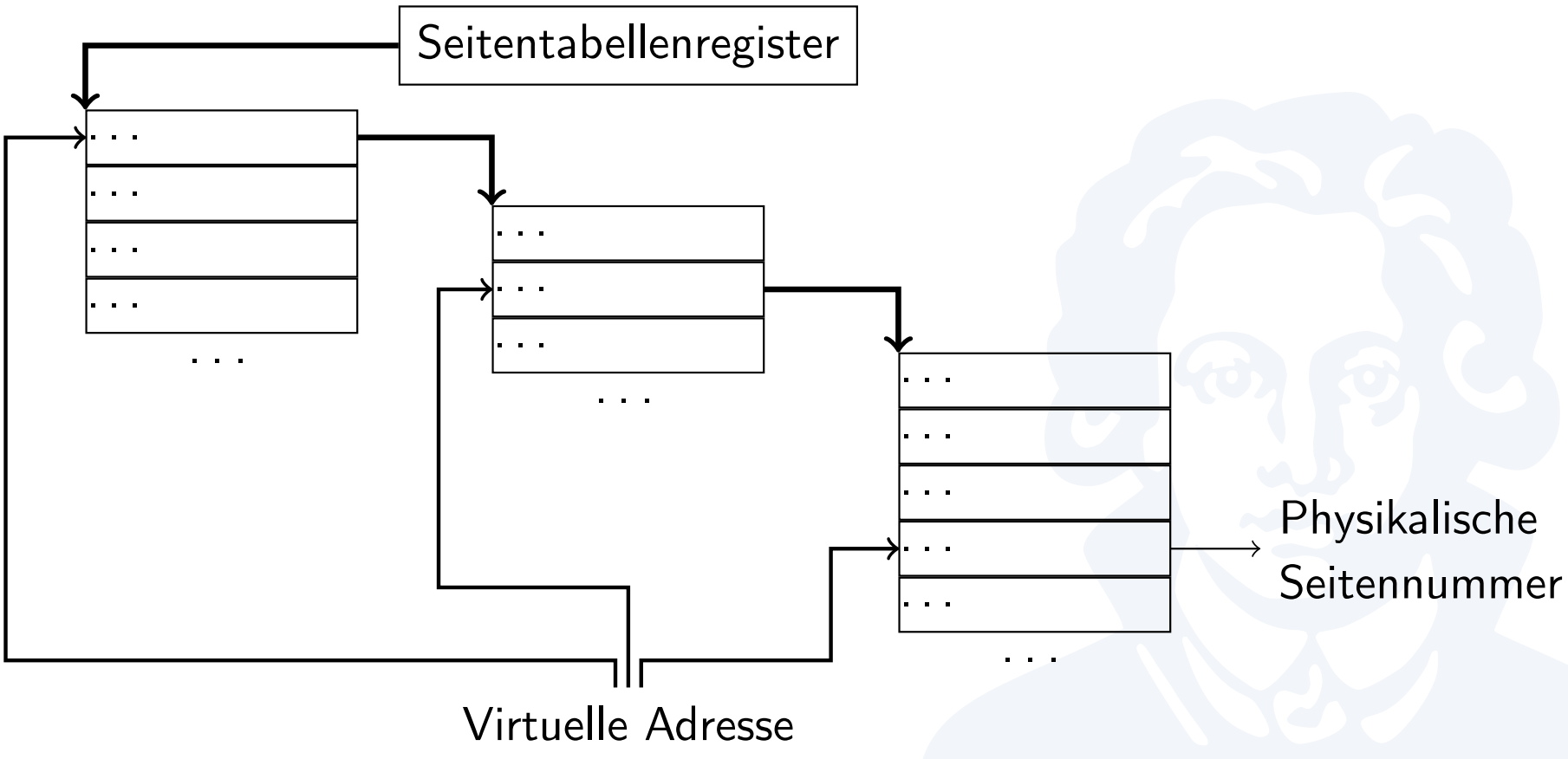
Beispiel: Seitentabelle der DEC Alpha

Moderne Prozessoren verwenden Hierarchie von Seitentabellen:



Beispiel: Seitentabelle der DEC Alpha

Moderne Prozessoren verwenden Hierarchie von Seitentabellen:

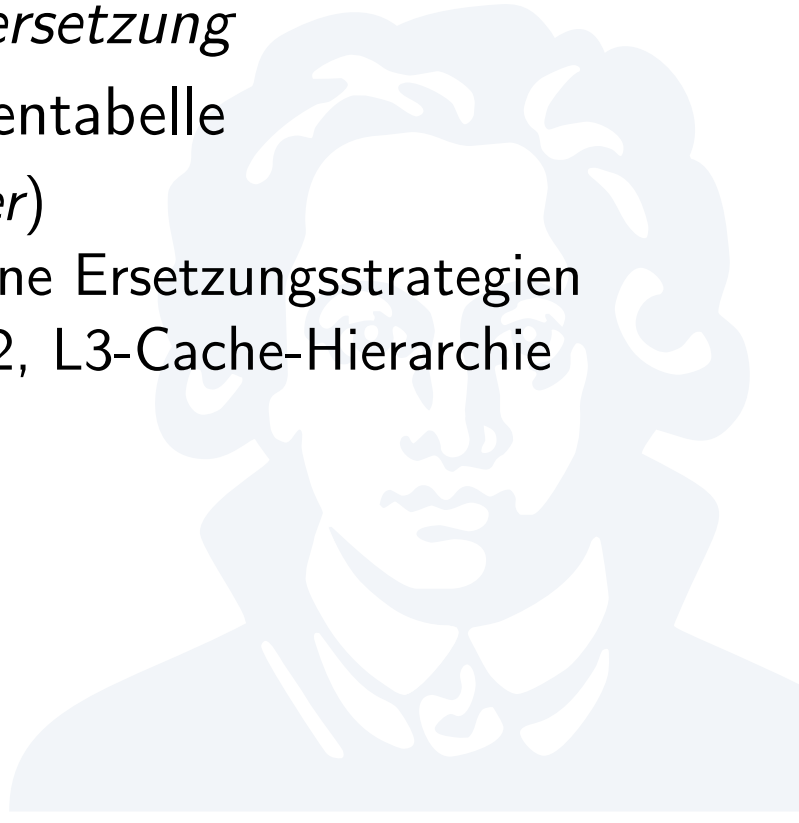


Der TLB

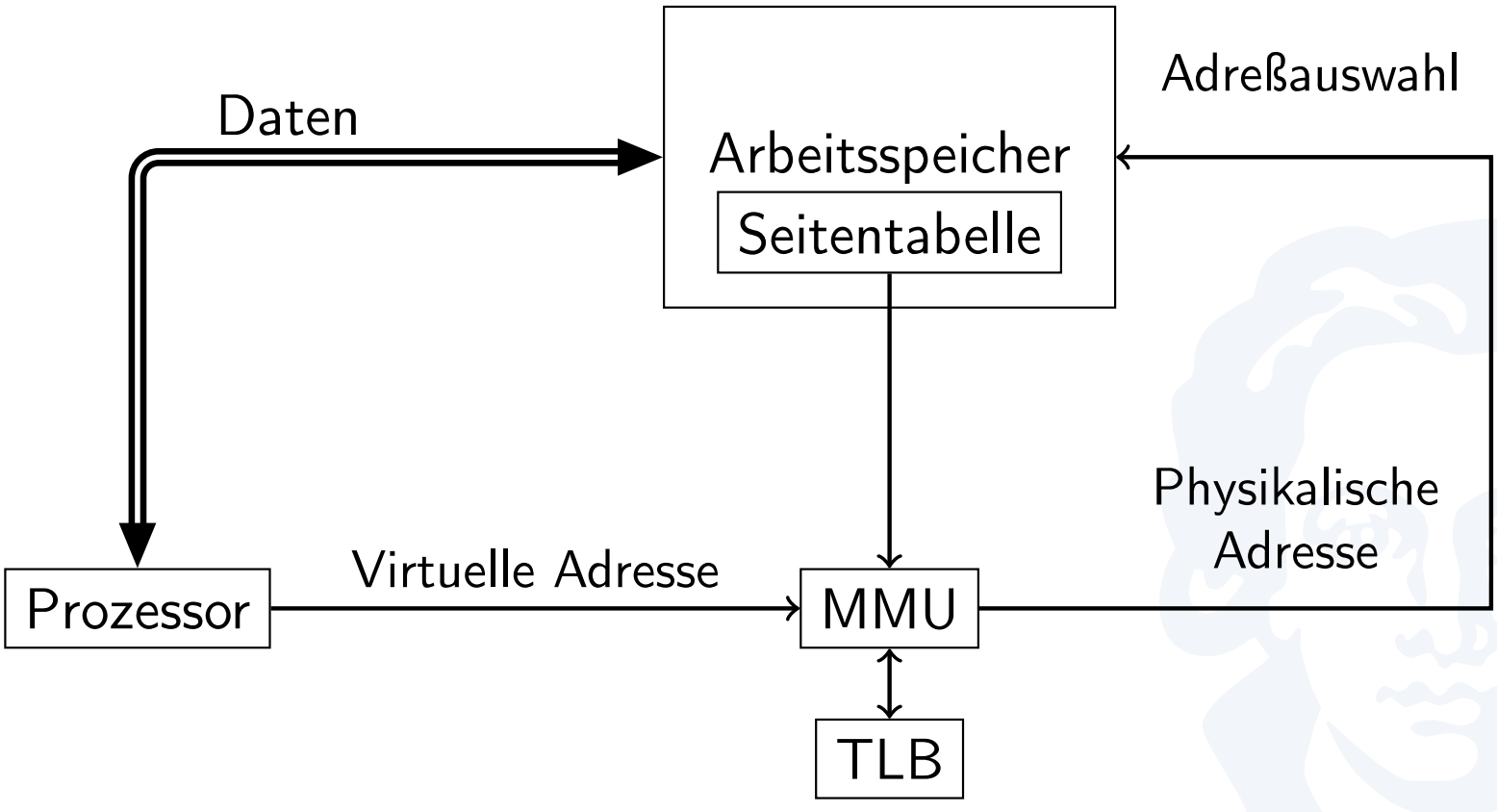
- *Problem:* Seitentabelle im Hauptspeicher
  Hunderte von Zyklen für Adreßübersetzung

Der TLB

- *Problem:* Seitentabelle im Hauptspeicher
  Hunderte von Zyklen für Adreßübersetzung
- Lösung: Effizienter Cache der Seitentabelle
  - TLB (*translation lookaside buffer*)
  - Meist hochassoziativ, verschiedene Ersetzungsstrategien
  - TLB-Hierarchie analog zu L1, L2, L3-Cache-Hierarchie möglich



MMU, TLB und Seitentabelle



Zusammenfassung: Grundlagen der Seitentabelle

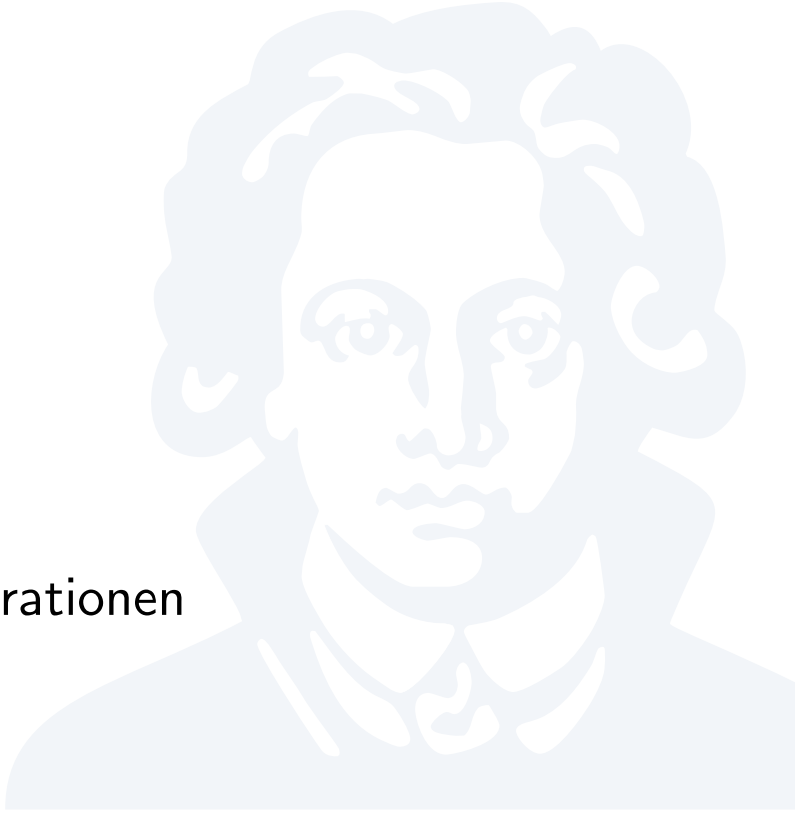
- Speicher-Hardware, Geräte haben *physikalische* Speicheradressen
- Maschinenprogramme sehen *virtuelle* Speicheradressen
  - Ausnahmen möglich: Bootvorgang, Betriebssystemkern
- Seitentabelle: virtuelle Adressen $\rightarrow$ physikalische Adressen
  - Wird vom Betriebssystemkern gewartet
  - Im 'normalen' RAM gespeichert
  - Speicherverwaltungseinheit (MMU) schlägt automatisch in Tabelle nach
- TLB: Cache für Seitentabelle
- Speicherzugriff:
  - Adreßsuche im TLB
  - TLB-Fehlzugriff: MMU schlägt in Seitentabelle nach
  - Nicht geunden: *Seitenfehler*

  $\Longrightarrow$ *Ausnahme*: Spezialbehandlung durch Betriebssystem

Das Betriebssystem

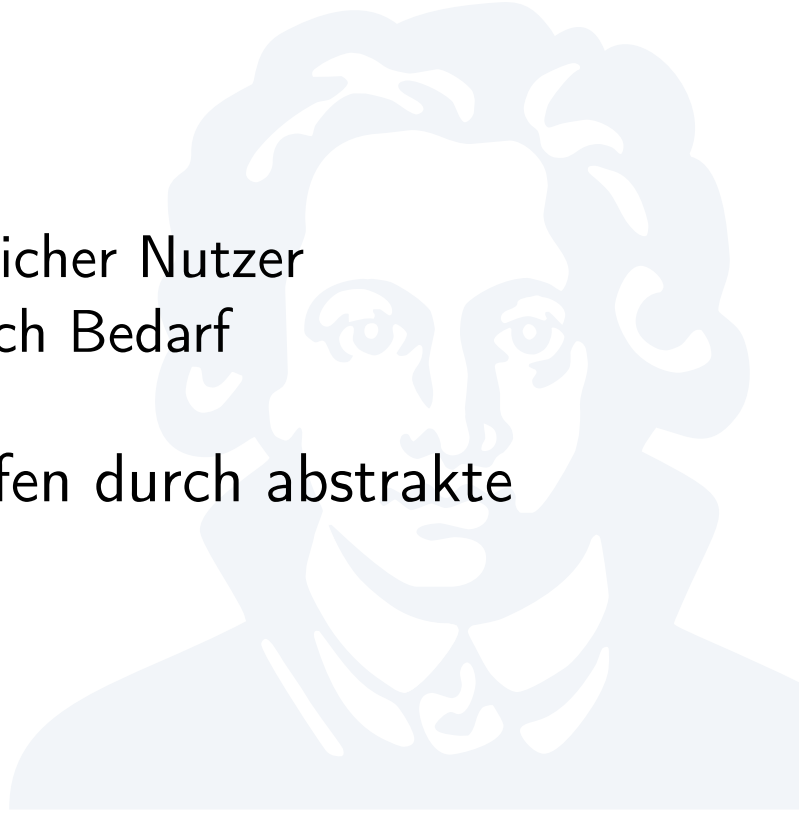
besteht aus:

- *Systemwerkzeuge*
  - Binder
  - Lader
  - Übersetzer
  - Benutzerschnittstellen
  - ...
- *Systemkern* (kernel)
  - Zentrale Ressourcenverwaltung:
    - Kontrolle von Zugriffsrechten
    - Vereinfachung der Zugriffsoperationen



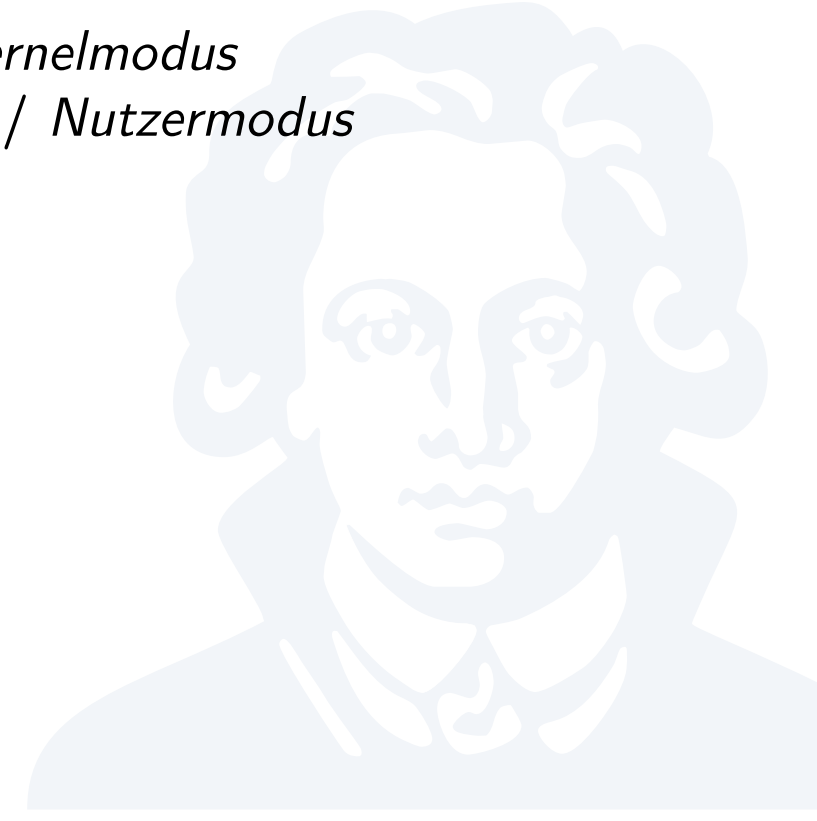
Aufgaben des Betriebssystemkerns

- Ressourcenverteilung:
  - Speicher:
    - RAM
    - Festplatte
  - Ausgabegeräte
  - Eingabegeräte
  - **Prozessorkerne/Zeit**
- Privilegienverwaltung
  - Trennen der Rechte unterschiedlicher Nutzer
  - Zugriffsprivilegien auf Geräte nach Bedarf
  - Isolierung von Programmen
- Vereinfachung von Hardwarezugriffen durch abstrakte Schnittstellen:
  - *Dateien*
  - *Prozesse*
  - *Signale*



Kernel und Nutzerprogramme

- Prozessorarchitektur trennt Betriebssystem und Anwendungsprogramme
  - Bit(s) in Spezialregister: Darf Prozessor auf Betriebssystemdaten zugreifen?
    - Ja: *privilegierter Modus / Kernelmodus*
    - Nein: *unprivilegierter Modus / Nutzermodus*



Kernel und Nutzerprogramme

- Prozessorarchitektur trennt Betriebssystem und Anwendungsprogramme
  - Bit(s) in Spezialregister: Darf Prozessor auf Betriebssystemdaten zugreifen?
    - Ja: *privilegierter Modus / Kernelmodus*
    - Nein: *unprivilegierter Modus / Nutzermodus*

Kernelmodus

- Direktzugriff auf Hardware
- Beliebiger Speicherzugriff
- Zugriff auf MMU

Nutzermodus

- Hardwarezugriff nur indirekt per Kernel
- Speicherzugriff erst nach MMU-Zuweisung durch Kernel

Kernel und Nutzerprogramme

- Prozessorarchitektur trennt Betriebssystem und Anwendungsprogramme
  - Bit(s) in Spezialregister: Darf Prozessor auf Betriebssystemdaten zugreifen?
    - Ja: *privilegierter Modus / Kernelmodus*
    - Nein: *unprivilegierter Modus / Nutzermodus*

Kernelmodus

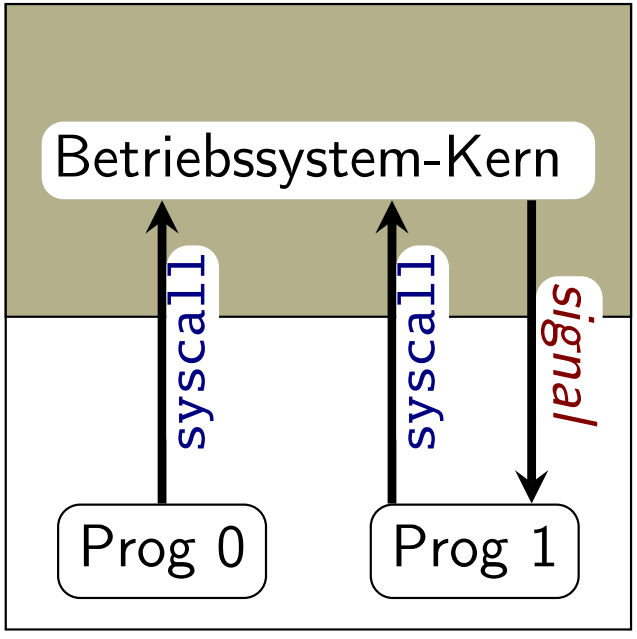
- Direktzugriff auf Hardware
- Beliebiger Speicherzugriff
- Zugriff auf MMU
- Programmierfehler beeinflussen *alle* Programme

Nutzermodus

- Hardwarezugriff nur indirekt per Kernel
- Speicherzugriff erst nach MMU-Zuweisung durch Kernel
- Programmierfehler betreffen nur *wenige* Programme

Kommunikation Kernelmodus ⇔ Nutzermodus

Kernel-Adreßraum
(*kernel space*)

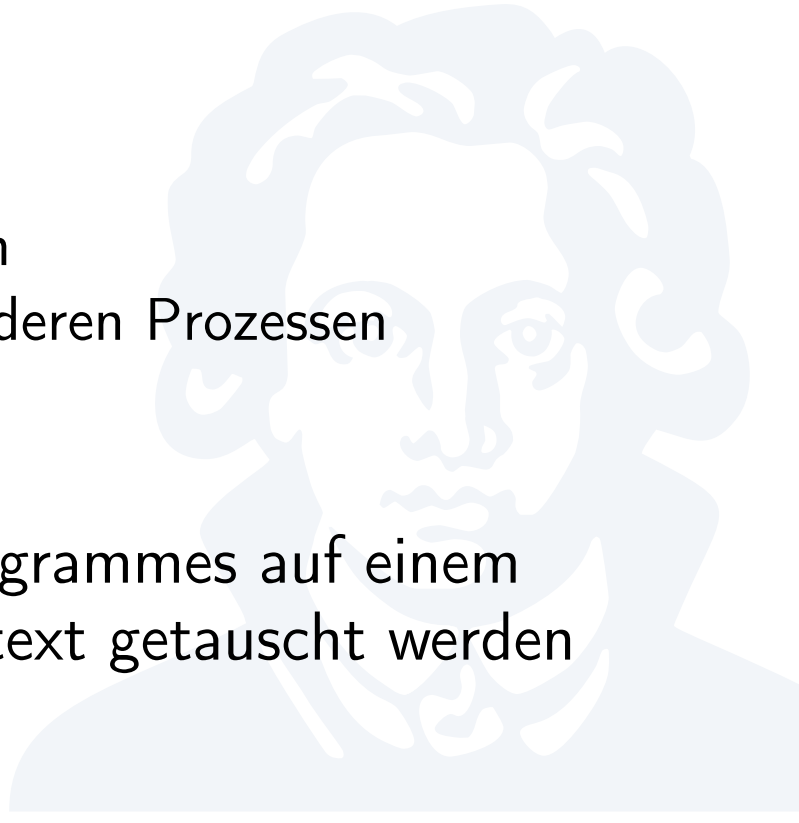


Nutzer-Adreßraum
(*user space*)

- Kommunikation Anwendung → Kernel:
  - `syscall`: Aktiviert Kernelmodus
  - Springt auf Syscall-Behandlungsroutine
  - Kernel kann Antwort senden (meist in Registern)
- Kommunikation Kernel → Anwendung:
  - Synchron: Kernel wartet auf `syscall`
  - Asynchron: Kernel sendet *Signal* (UNIX-spezifisch)

Der Prozeß

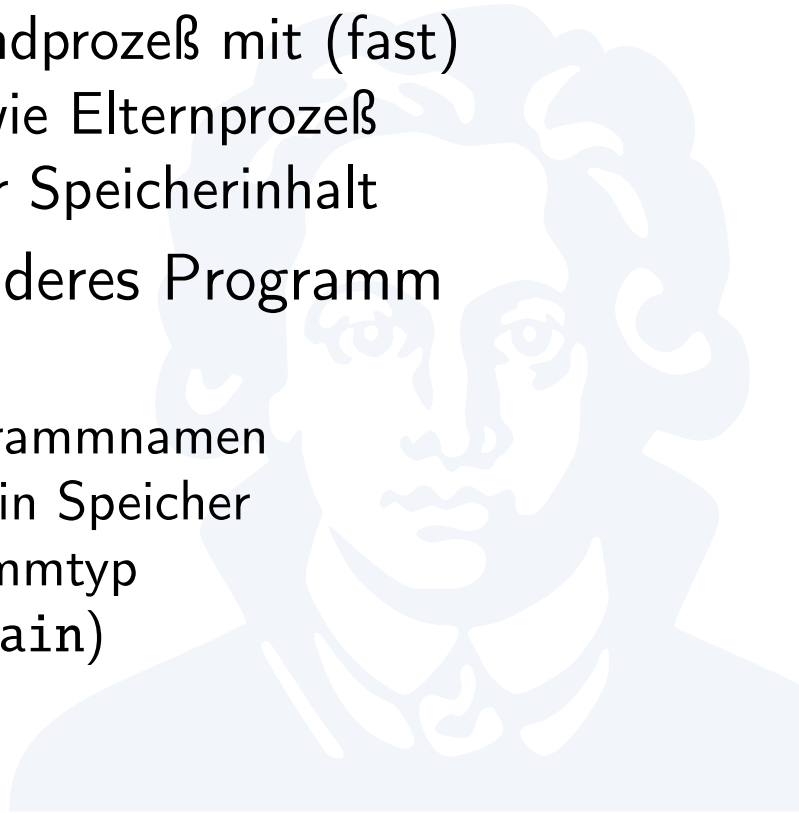
- Jedes laufende Programm besteht aus einem oder mehreren *Prozessen*
- Kann selbst Kindprozesse starten
- Jeder Prozeß hat *Prozeßkontext*:
  - Seitentabelle
  - Geöffnete Dateien
  - Geöffnete Netzwerkverbindungen
  - Kommunikationsleitungen zu anderen Prozessen
  - Kindprozesse
  - ...
- Bei Umschalten des laufenden Programmes auf einem Prozessorkern muß der Prozeßkontext getauscht werden (Kontextwechsel)



Starten eines neuen Programmes in UNIX

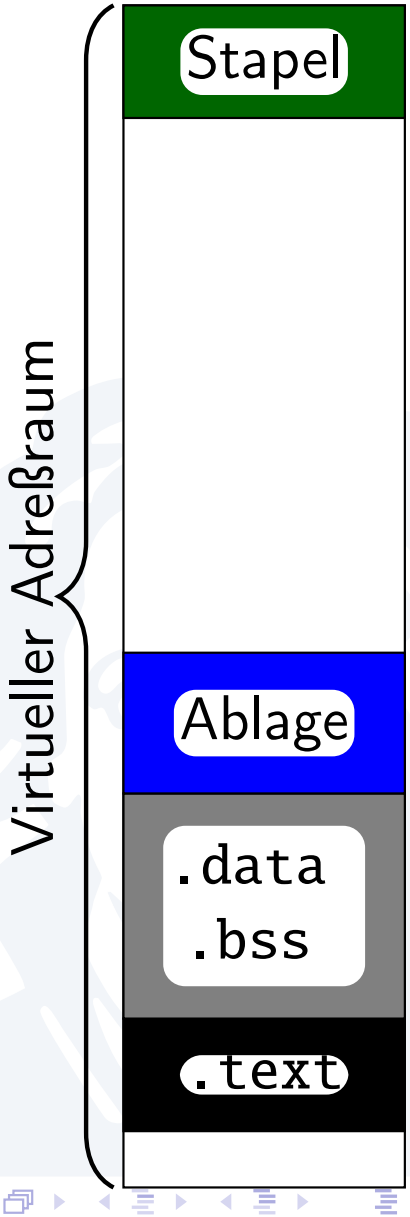
Start aus aufrufendem Programm (Kommando-shell, graphische Benutzerumgebung):

- Erzeugung eines Kindprozesses:
  - Systemaufruf `fork`: Erzeuge Kindprozeß mit (fast) inhaltsgleichem Prozeßkontext wie Elternprozeß
  - Gleiche Seitentabelle $\Rightarrow$ Gleicher Speicherinhalt
- Kindprozeß: ersetze mich durch anderes Programm
  - Systemaufruf `execve`:
    - Nimm Zeichenkette mit Programmnamen
    - Lade Anfang des Programms in Speicher
    - Identifiziere Lader für Programmtyp
    - Starte Lader ($\rightarrow$ Binder, $\rightarrow$ `main`)



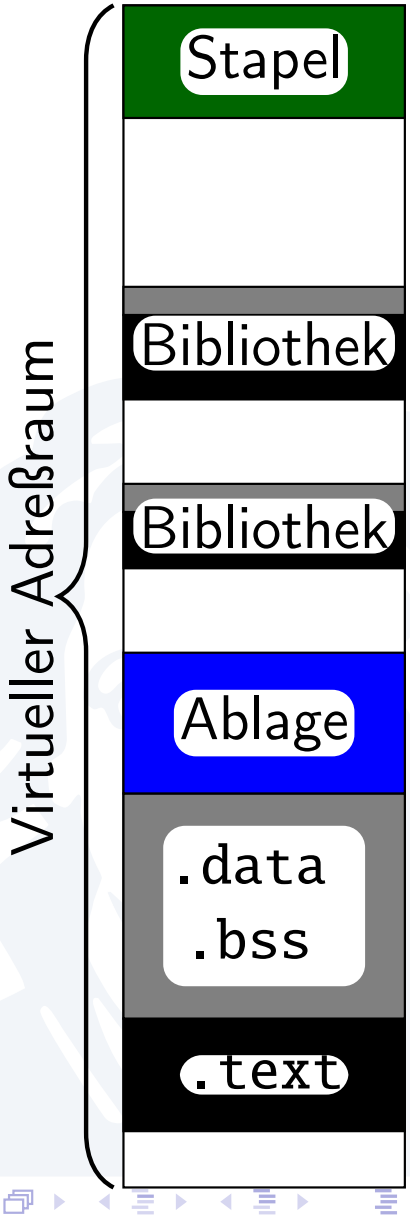
Adreßraum eines Prozesses

- Ein geladenes Programm hat:
  - `.text`
  - `.data`, `.bss`
  - Stapelspeicher
  - Dynamische Bibliotheken
  - Auf Speicher abgebildete Dateien
  - Mit anderen Programmen geteilter Speicher
  - Prozeßkontext (im Betriebssystemkern)



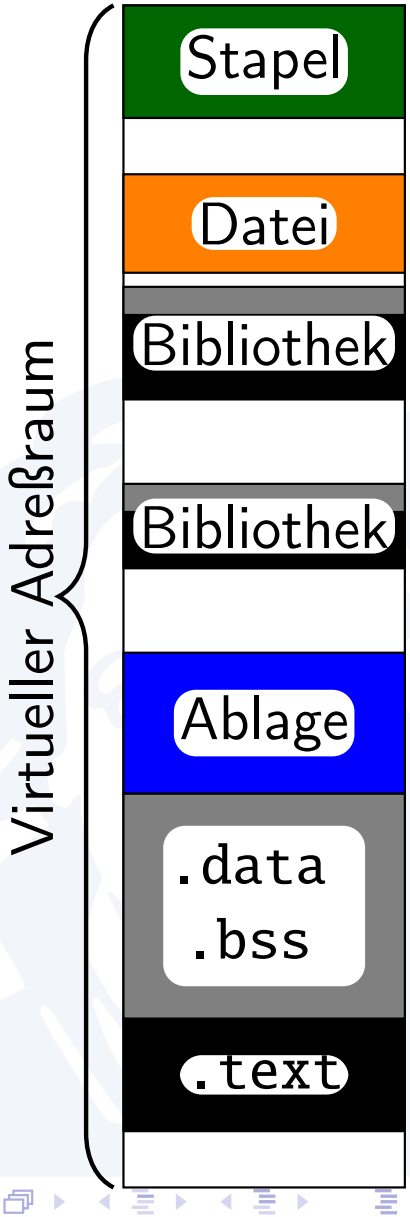
Adreßraum eines Prozesses

- Ein geladenes Programm hat:
  - `.text`
  - `.data`, `.bss`
  - Stapelspeicher
  - Dynamische Bibliotheken
  - Auf Speicher abgebildete Dateien
  - Mit anderen Programmen geteilter Speicher
  - Prozeßkontext (im Betriebssystemkern)



Adreßraum eines Prozesses

- Ein geladenes Programm hat:
  - `.text`
  - `.data`, `.bss`
  - Stapelspeicher
  - Dynamische Bibliotheken
  - Auf Speicher abgebildete Dateien
  - Mit anderen Programmen geteilter Speicher
  - Prozeßkontext (im Betriebssystemkern)

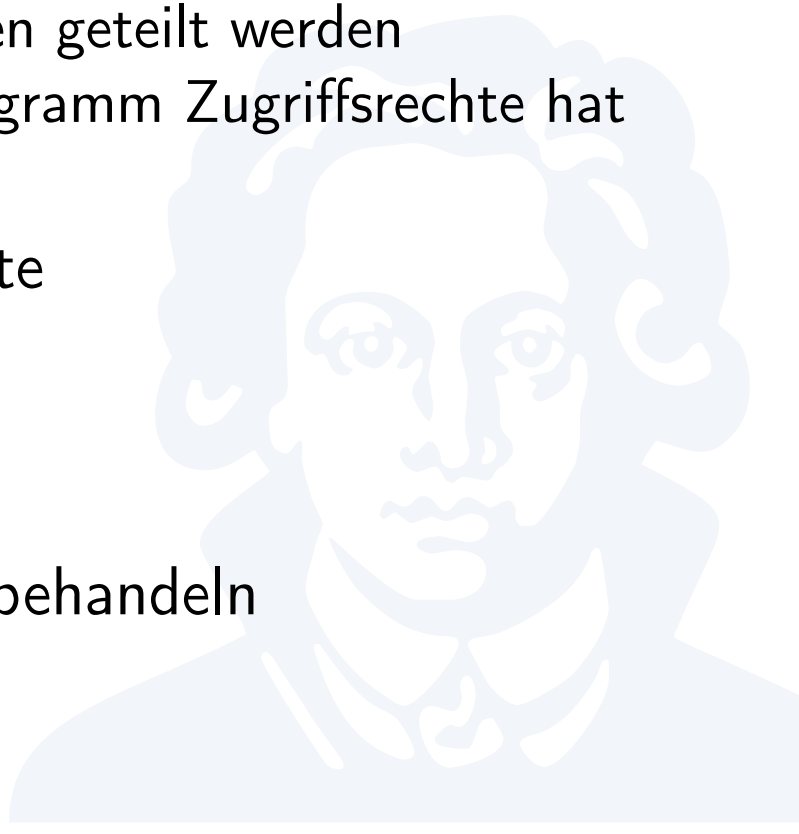


Speicherverwaltung zur Laufzeit

- Statischer Speicher (`.data`, `.bss`) hat feste Größe
- Stapelspeicher (stack):
  - Anfangs feste Größe, wächst nach unten
  - Unteres Ende:
    MMU-Fehlzugriff $\Rightarrow$ zusätzlicher Stapelspeicher wird von Betriebssystemkern angehängt
  - Per-Prozeß-Limit
- Ablagespeicher (heap):
  - Zwischen statischem Speicher und *Programmpause*
  - Wächst mit explizitem Systemaufruf
  - Meist automatisch durch Systembibliotheken
- Kommunikation mit Restsystem:
  z.B. Explizite MMU-Anfragen via `mmap`, `mprotect` (UNIX)

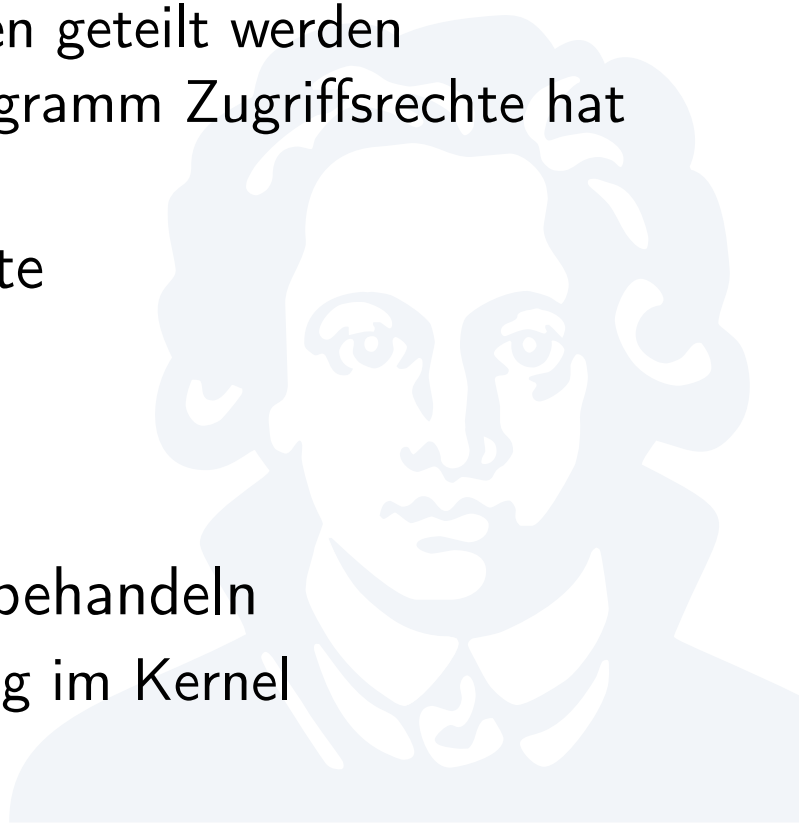
Seitentabellen und Nutzerprogramme

- Prozesse können Kernel bitten, Daten im Speicher abzubilden:
  - Dateien
  - Seiten, die mit anderen Prozessen geteilt werden
  - Gerätespeicher, auf den das Programm Zugriffsrechte hat
  - Dynamische Bibliotheken
- Individuelle Rechtevergabe pro Seite
  - Lesen
  - Schreiben
  - Ausführen
- Prozeß kann eigene Zugriffsfehler behandeln



Seitentabellen und Nutzerprogramme

- Prozesse können Kernel bitten, Daten im Speicher abzubilden:
  - Dateien
  - Seiten, die mit anderen Prozessen geteilt werden
  - Gerätespeicher, auf den das Programm Zugriffsrechte hat
  - Dynamische Bibliotheken
- Individuelle Rechtevergabe pro Seite
  - Lesen
  - Schreiben
  - Ausführen
- Prozeß kann eigene Zugriffsfehler behandeln
  - Ansonsten: Fehlzugriffbehandlung im Kernel



Fehlzugriffbehandlung

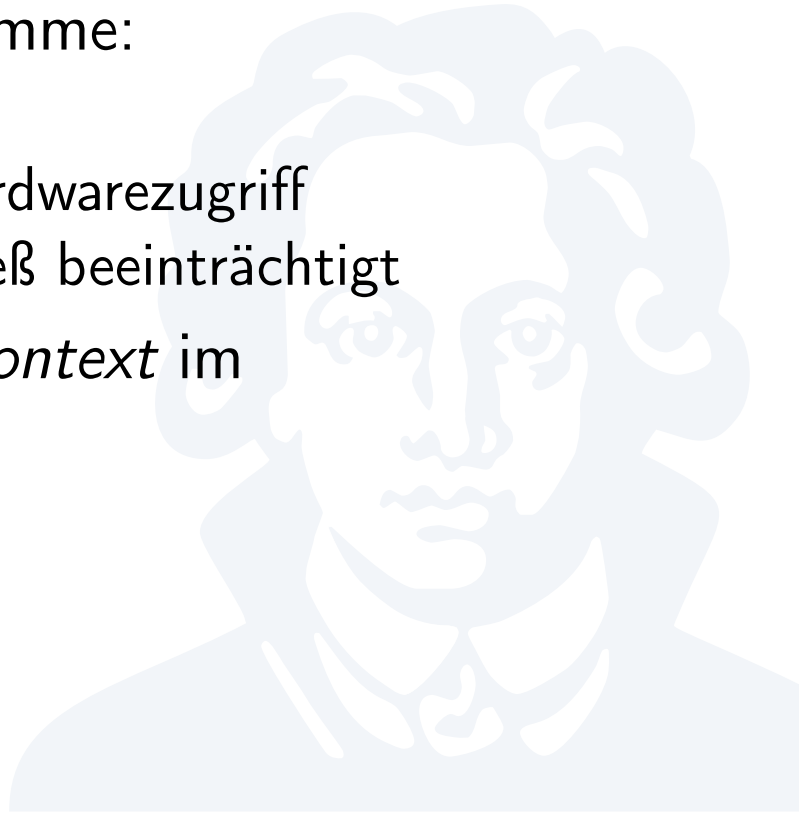
- MMU löst Unterbrechung bei Fehlzugriff aus (*page fault*)
- Mögliche Gründe:
  - Unberechtigter Zugriff (z.B. Schreiben auf nur-Lesen-Seite)
  - Seite fehlt

Fehlzugriffbehandlung

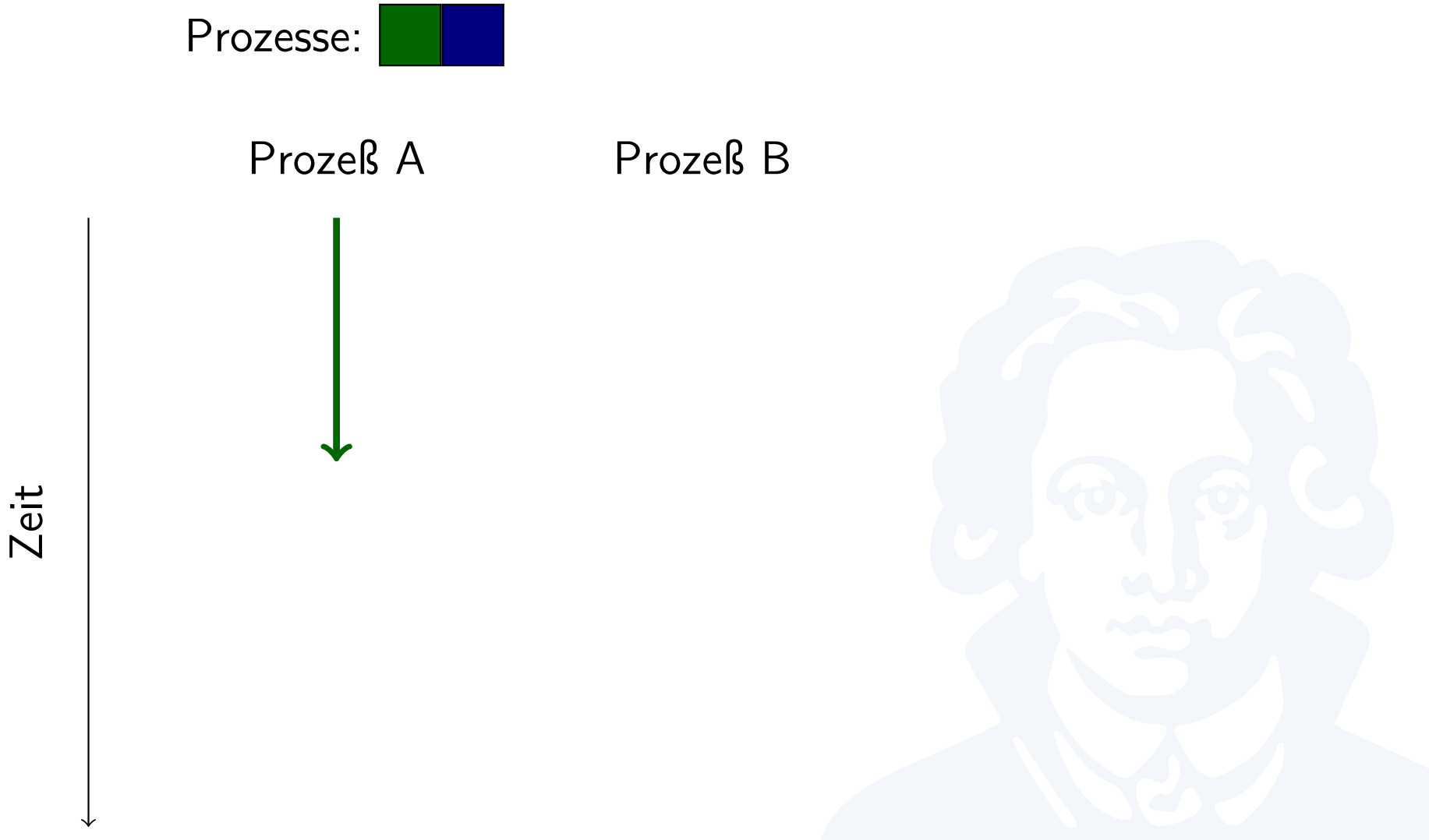
- MMU löst Unterbrechung bei Fehlzugriff aus (*page fault*)
- Mögliche Gründe:
  - Unberechtigter Zugriff (z.B. Schreiben auf nur-Lesen-Seite)
  - Seite fehlt
- Fehlende Seiten von Betriebssystemkern behandelt:
  - Seite in Bereitschaft, aber noch nicht zugewiesen: Seitentabelle aktualisiert, zurück zum Prozeß
  - Seite ungültig: Programmfehler (*segmentation fault*)
  - Seite ausgelagert: Block im Festspeicher
    - Seite repräsentiert Dateiinhalt auf Festplatte, aber nicht im Hauptspeicher gecached:
    - Seite wird von Festspeicher geladen, dann Rückkehr zum Programm

Prozesse: Zusammenfassung

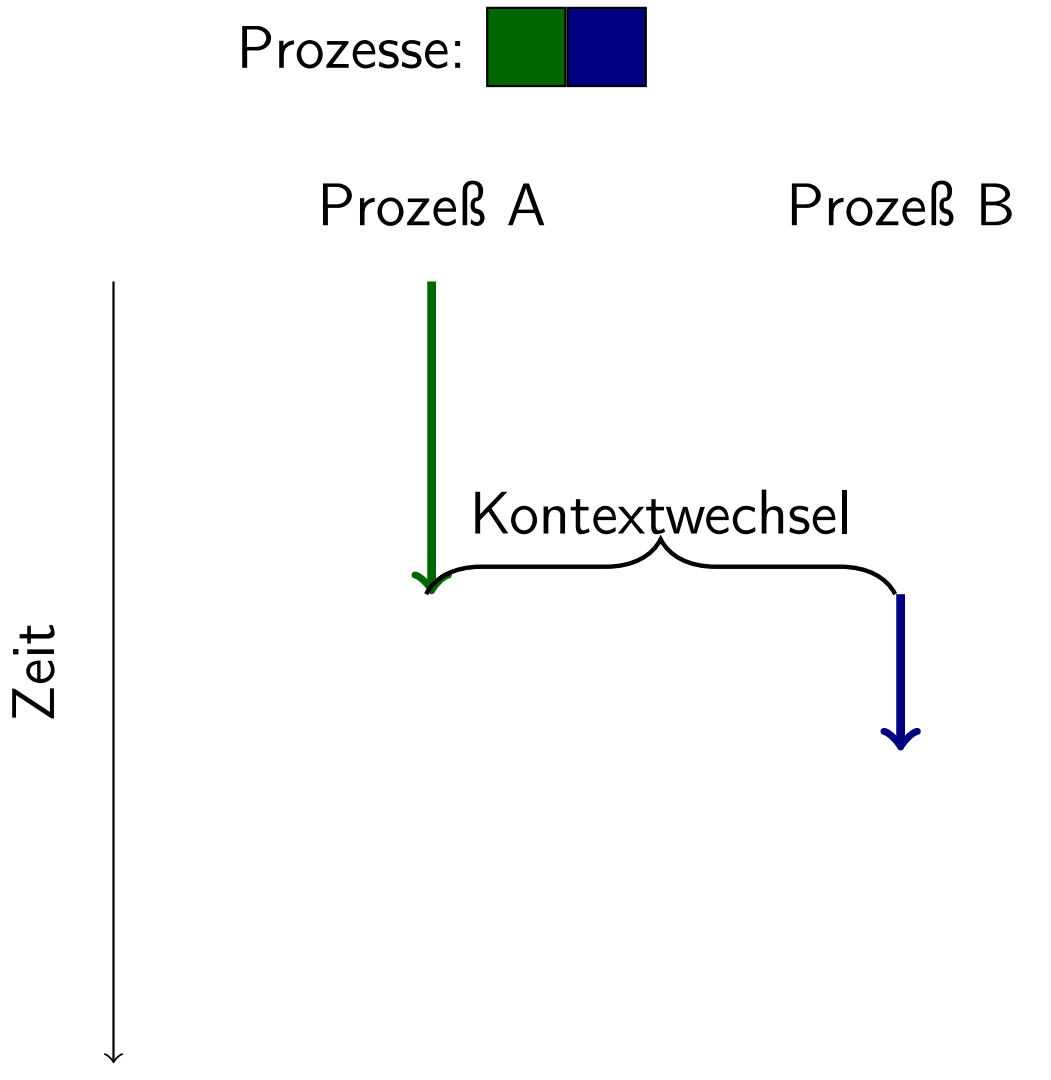
- Prozesse werden von anderen Prozessen gestartet
- Betriebssystemkern isoliert Programme:
  - Eigene Speicherbereiche
  - Abstrakte Systemaufrufe für Hardwarezugriff
  - Programmabsturz: nur ein Prozeß beeinträchtigt
- Jeder Prozeß hat eigenen *Prozeßkontext* im Betrisbssystemkern



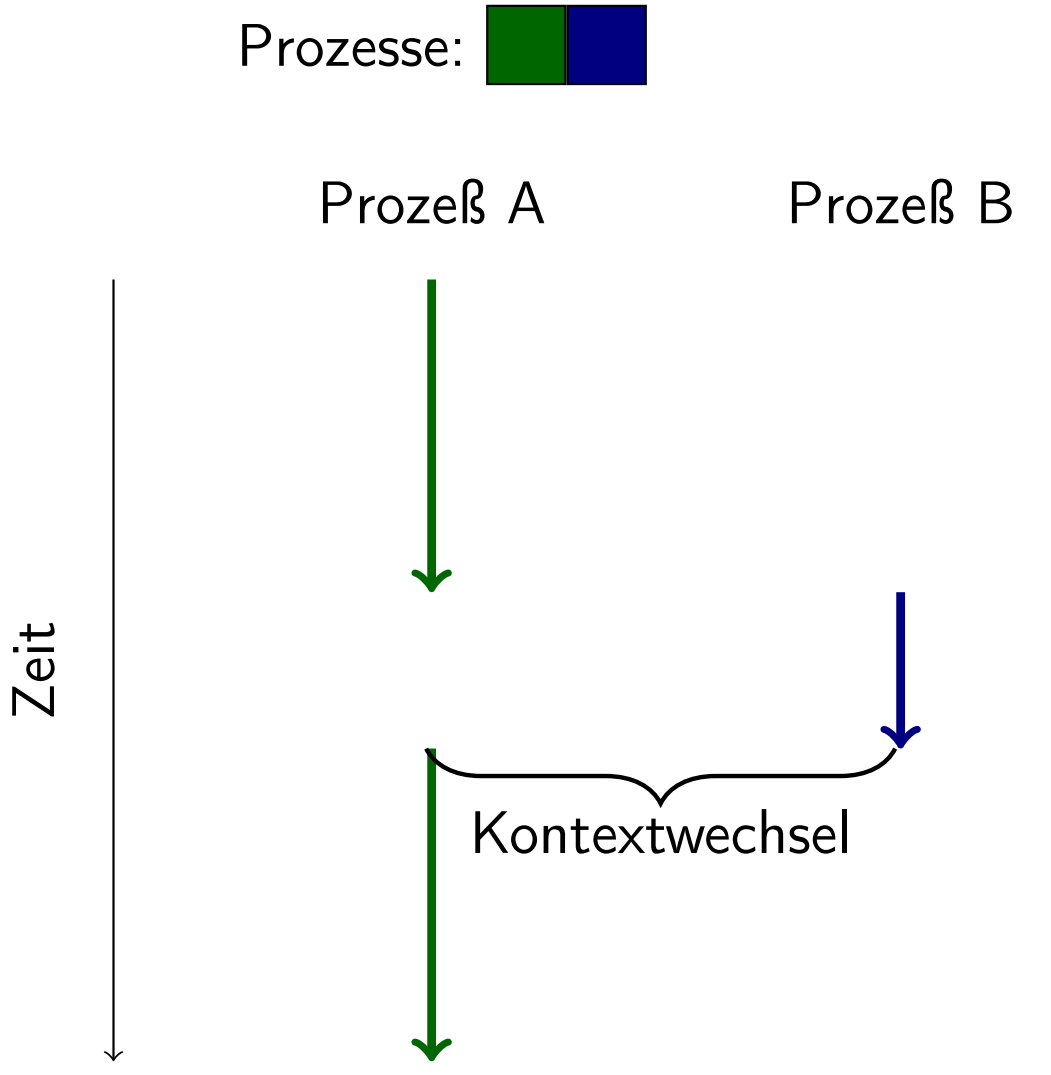
Mehrere Prozesse auf einem Prozessorkern



Mehrere Prozesse auf einem Prozessorkern



Mehrere Prozesse auf einem Prozessorkern



Die Round-Robin-Strategie der Prozeßveraltung

Round-Robin: „Rundum-Modell“

Laufzeit: →

Prozeß A *bereit*

Prozeß B *bereit*

Prozeß C *bereit*

Prozeß D *bereit*

Die Round-Robin-Strategie der Prozeßveraltung

Round-Robin: „Rundum-Modell“

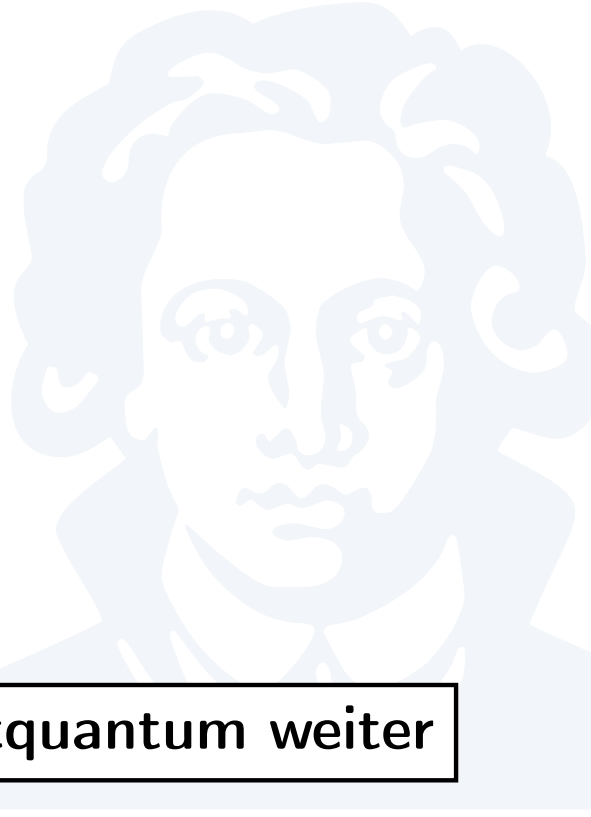
Laufzeit: ⟶

Prozeß A **läuft** ⟶

Prozeß B *bereit*

Prozeß C *bereit*

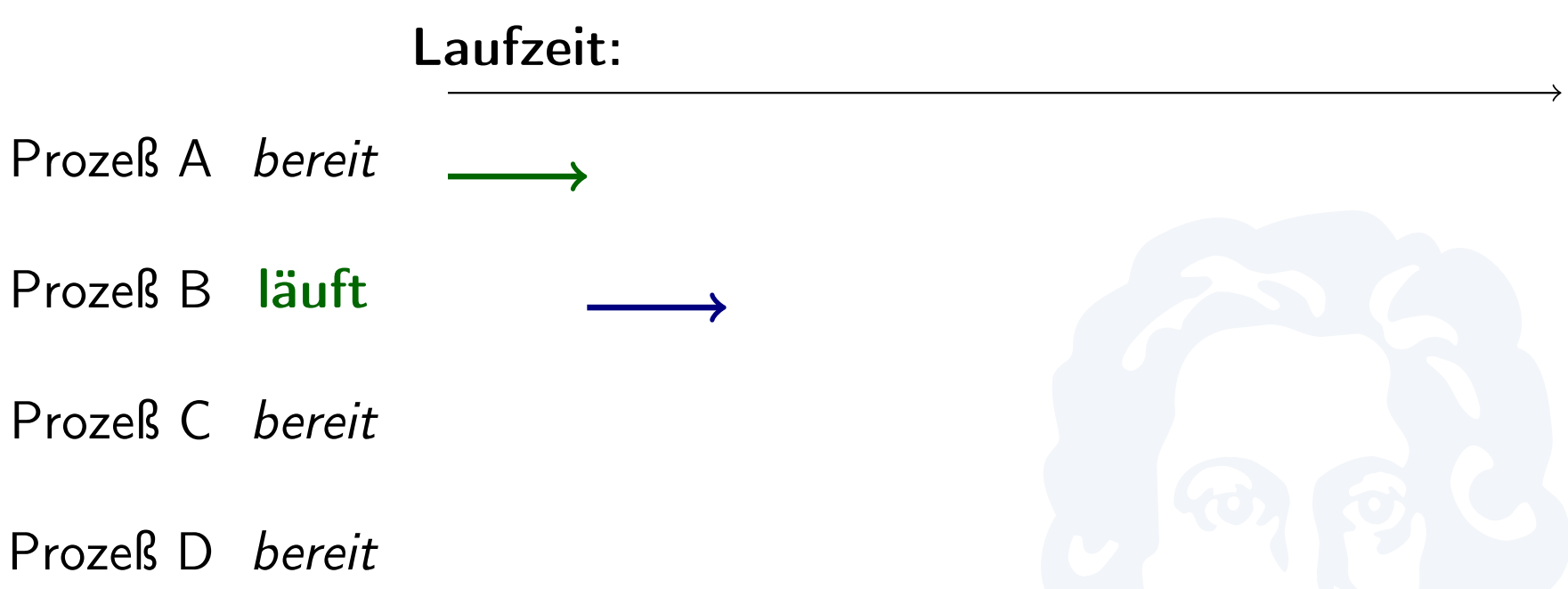
Prozeß D *bereit*



Betriebssystem schaltet nach festem Zeitquantum weiter

Die Round-Robin-Strategie der Prozeßveraltung

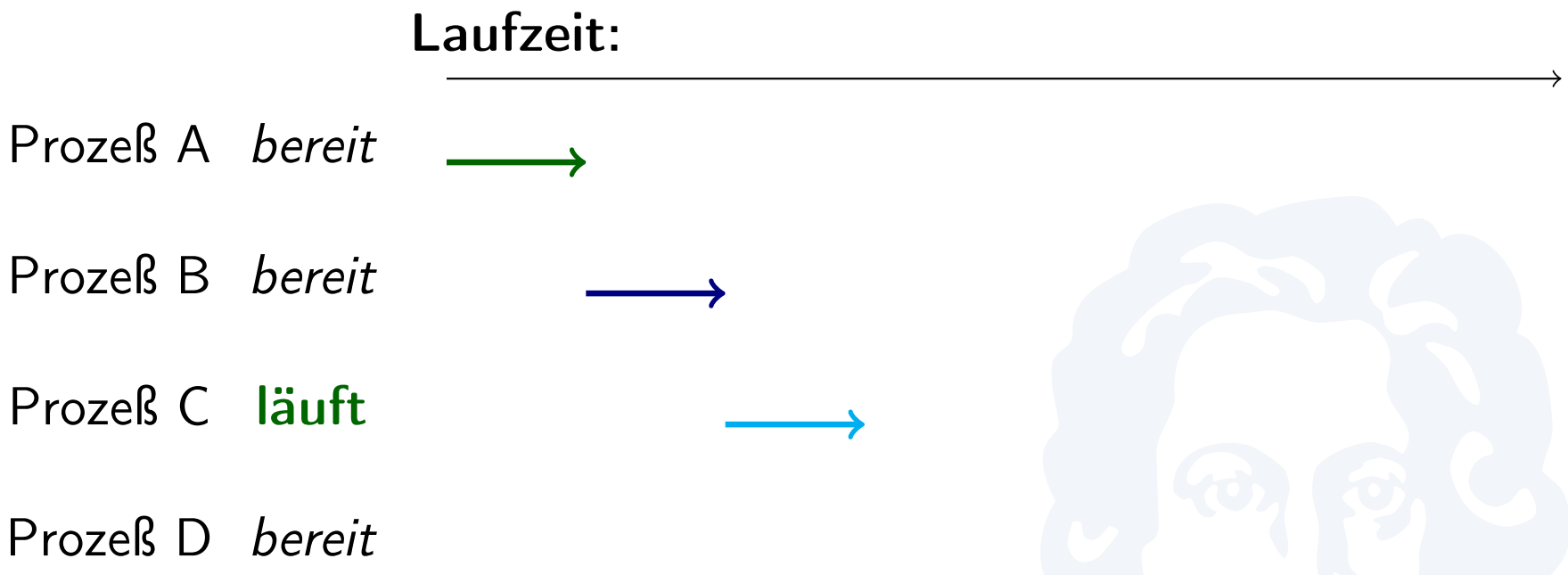
Round-Robin: „Rundum-Modell“



Betriebssystem schaltet nach festem Zeitquantum weiter

Die Round-Robin-Strategie der Prozeßveraltung

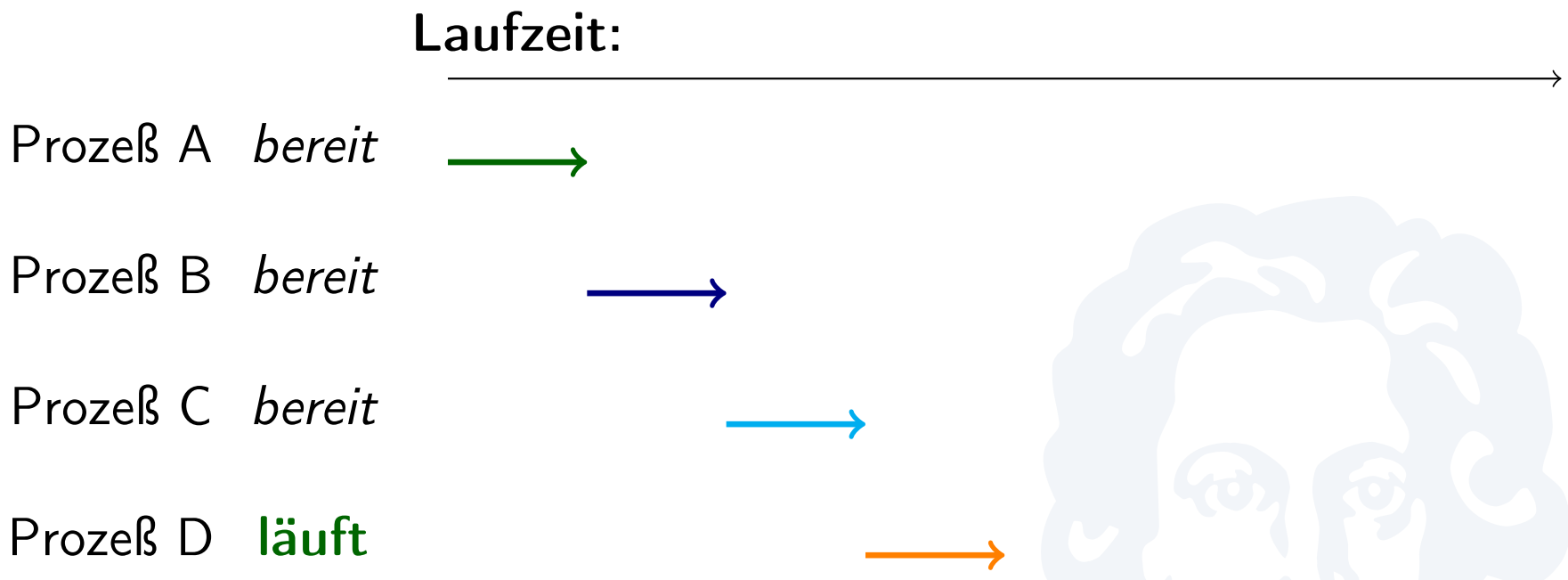
Round-Robin: „Rundum-Modell“



Betriebssystem schaltet nach festem Zeitquantum weiter

Die Round-Robin-Strategie der Prozeßveraltung

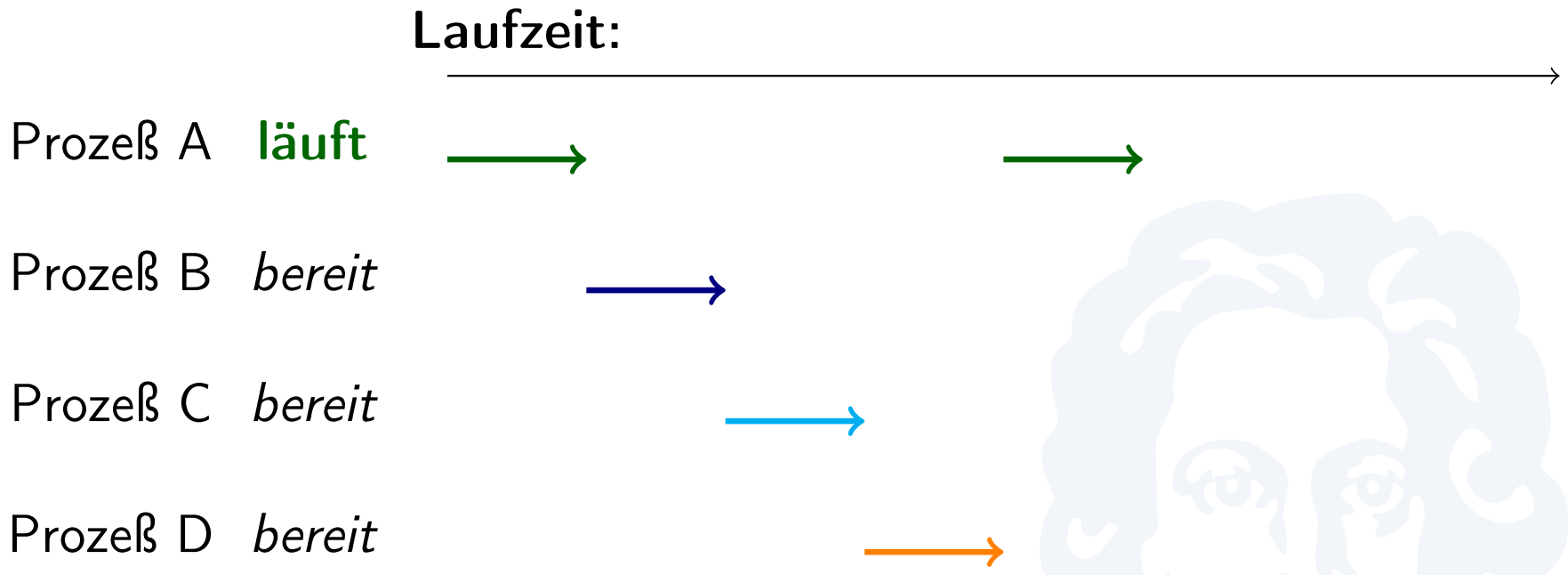
Round-Robin: „Rundum-Modell“



Betriebssystem schaltet nach festem Zeitquantum weiter

Die Round-Robin-Strategie der Prozeßveraltung

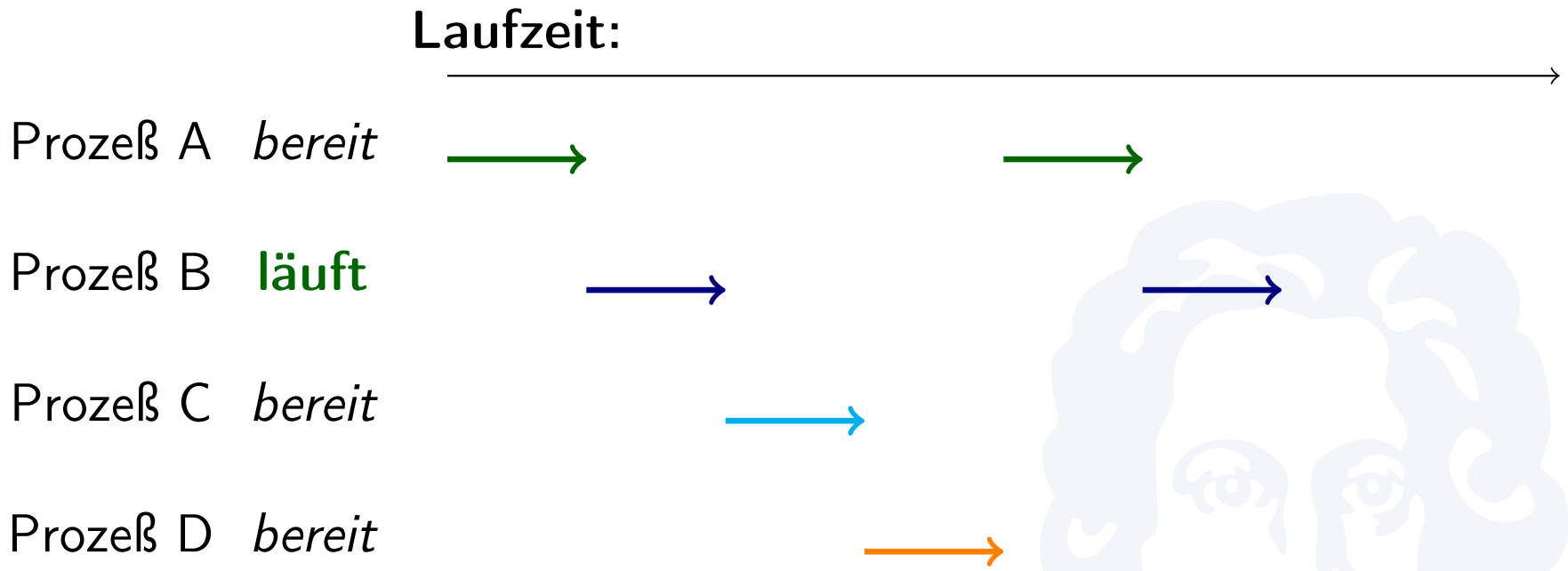
Round-Robin: „Rundum-Modell“



Betriebssystem schaltet nach festem Zeitquantum weiter

Die Round-Robin-Strategie der Prozeßveraltung

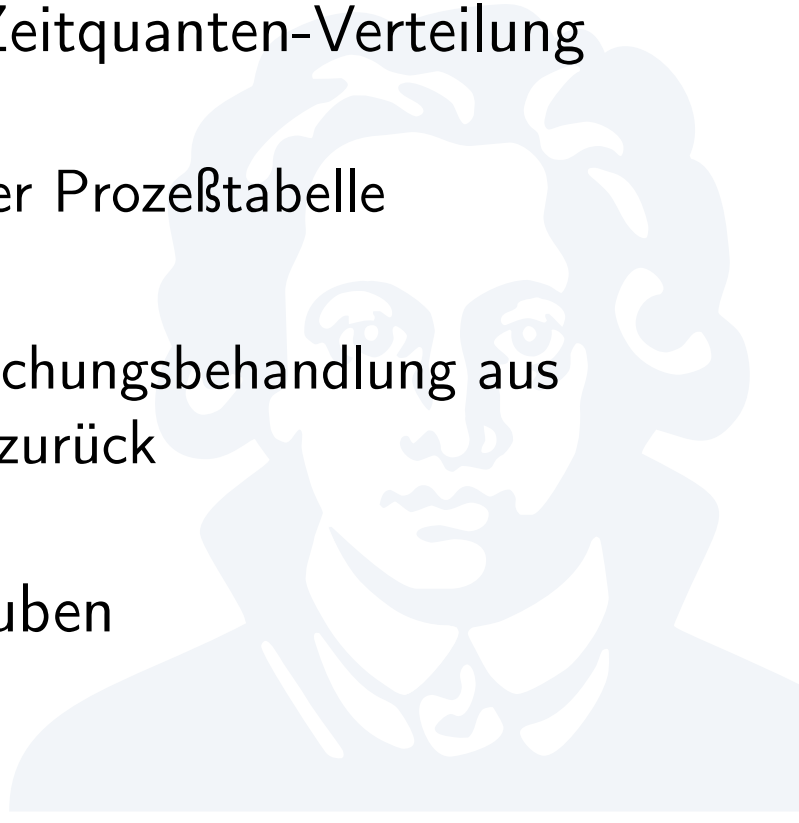
Round-Robin: „Rundum-Modell“



Betriebssystem schaltet nach festem Zeitquantum weiter

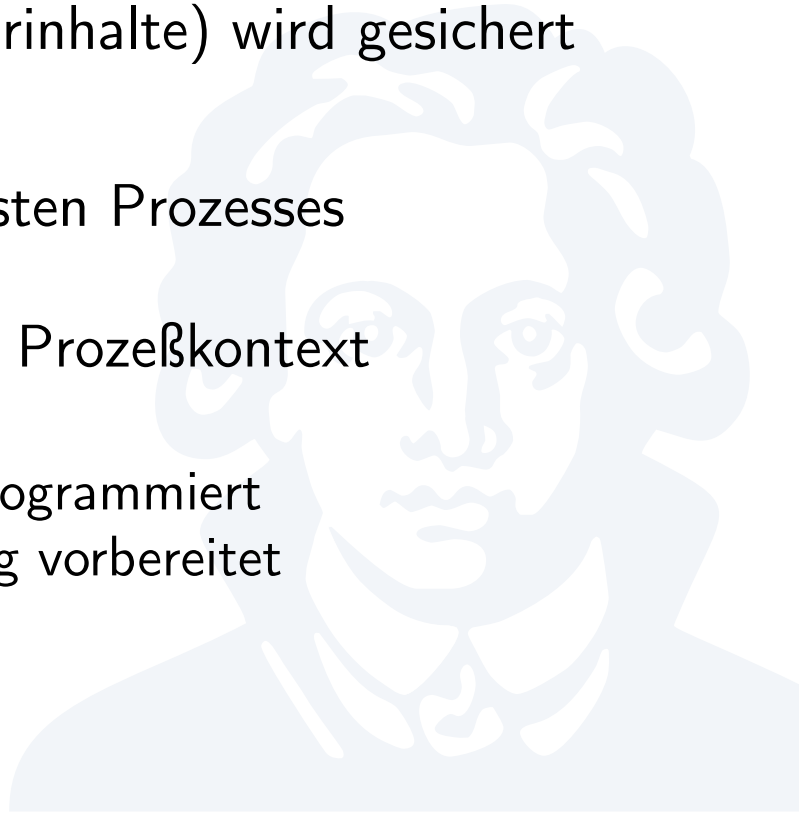
Der Scheduler

- *Scheduler* ist verantwortlich für Wahl des nächsten zu laufenden Prozesses
- Prozessen werden *Zeitquanten* zur Ausführung zugewiesen
- *Round-Robin-Strategie* versucht, Zeitquanten-Verteilung fair zu machen:
  - Iteration über alle Prozesse in der Prozeßtabelle
- Am Ende des Zeitquantums:
  - Zeitmesser (*timer*) löst Unterbrechungsbehandlung aus
  - Betriebssystem erhält Kontrolle zurück
  - *Kontextwechsel*
- Andere Scheduling-Strategien erlauben Prozeß-Prioritisierung



Kontextwechsel

- Alter Prozeß:
  - Scheduler setzt Status des aktuellen Prozesses von *läuft* auf *bereit*
  - Prozeßkontext (inklusive Registerinhalte) wird gesichert
- Neuer Prozeß:
  - Scheduler setzt Status des nächsten Prozesses von *bereit* auf *läuft*
  - Prozeß wird aktiviert, indem der Prozeßkontext wiederhergestellt wird:
    - Seitentabelle geladen, TLB programmiert
    - Registerinhalte für Rücksprung vorbereitet
- Sprung zum neu-laufenden Prozeß



Erzwungene Kontextwechsel

```
li   $v0, 5
syscall
```

Erzwungene Kontextwechsel

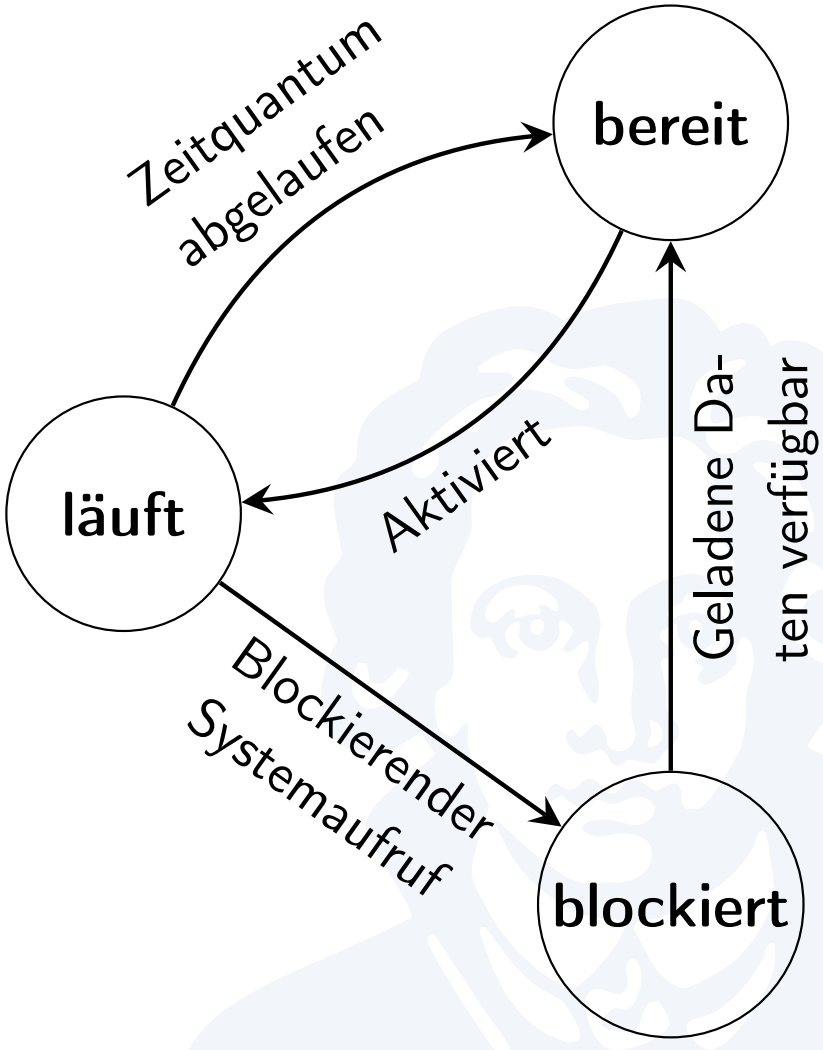
```
li   $v0, 5
syscall
```

- Führt zu Systemaufruf (in Standardbibliothek), der auf Eingabe wartet
- Betriebssystem *blockiert* Programm, bis Daten verfügbar sind

Erzwungene Kontextwechsel

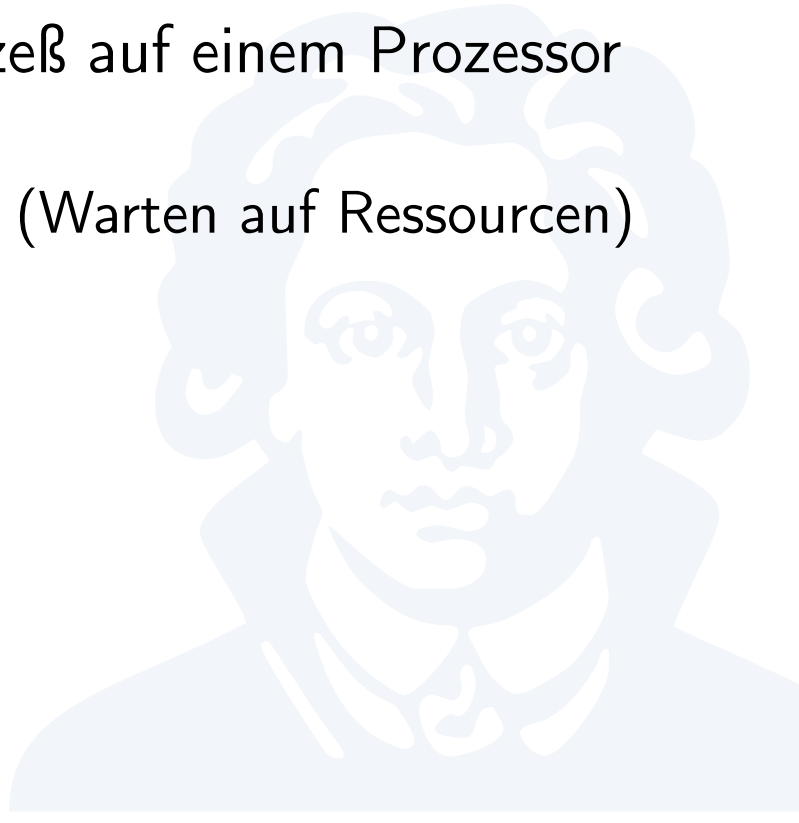
```
li   $v0, 5
syscall
```

- Führt zu Systemaufruf (in Standardbibliothek), der auf Eingabe wartet
- Betriebssystem *blockiert* Programm, bis Daten verfügbar sind
- Analog bei MMU-Fehlzugriff



Zusammenfassung: Scheduling-Grundlagen

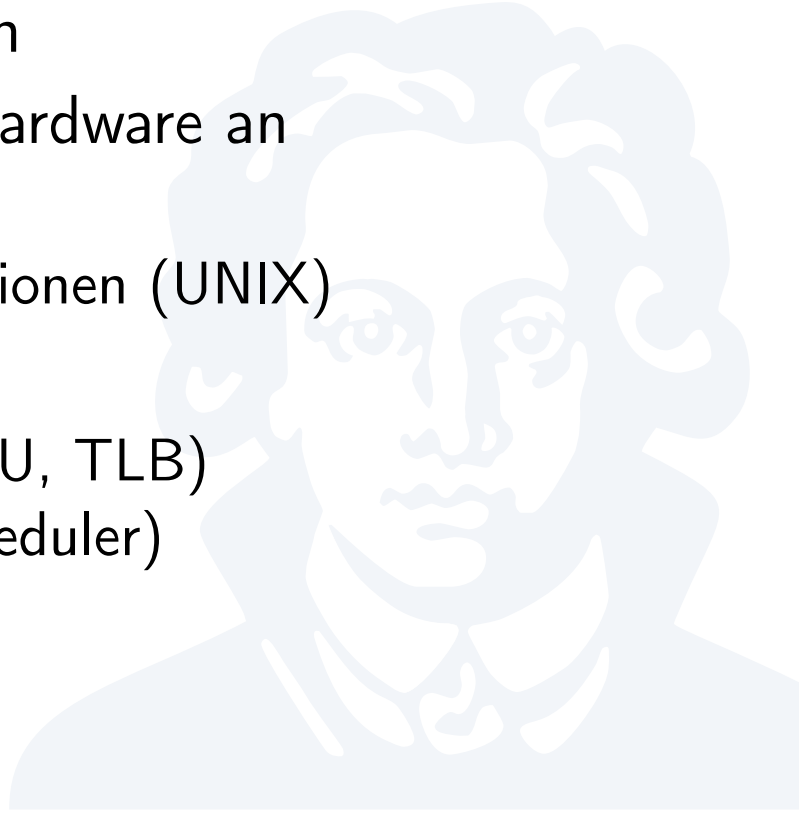
- Mehrere Prozesse auf einem System
- Scheduler wählt laufende Prozesse
- Kontextwechsel: wechselt den Prozeß auf einem Prozessor
  - Nach Ablauf des Zeitquantums
  - Bei blockierendem Systemaufruf (Warten auf Ressourcen)
- Mögliche Prozeßzustände:
  - **läuft** (*running*)
  - **bereit** (*ready*)
  - **blockiert** (*blocked*)



Betriebssystemkern: Zusammenfassung

Betriebssystemkern/Kernel:

- Arbitriert Hardwarezugriff
- Implementiert Sicherheitsprüfungen
- Bietet einfache Schnittstelle auf Hardware an
  - `syscall` für synchronen Zugriff
  - *Signale* für asynchrone Notifikationen (UNIX)
- Verwaltet Ressourcen:
  - Speicher: *Seitentabelle* (via MMU, TLB)
  - Prozessorzeit: *Prozesse* (via Scheduler)



Performanz: Übersicht

Für moderne Prozessoren:

Operation	**Zeit**	$\times 2E9$
L1-Cachezugriff	0,5 ns	1 s
Sprungfehlvorhersage	5 ns	10 s
L2-Cachezugriff	7 ns	12 s
Hauptspeicherzugriff	100 ns	3 min, 20 s
2 kiB über 1 Gbps-Netz senden	20.000 ns	$>$ 11 h
SSD-Zugriff (beliebige Adresse)	150.000 ns	3,5 Tage
1 MiB sequentiell aus RAM lesen	250.000 ns	5,75 Tage
1 MiB sequentiell von SSD lesen	1.000.000 ns	$>$ 3 Wochen
1 MiB sequentiell von Festplatte	20.000.000 ns	$>$ 7,5 Monate
Netzwerk: Kalifornien $\leftrightarrow$ NL	150.000.000 ns	$>$ 4,75 Jahre

(Daten von Peter Norvig, Jeff Dean, ca. 2010)

Nächste Woche:

Die Programmiersprache C