# Kapitel 4: Speicherverwaltung

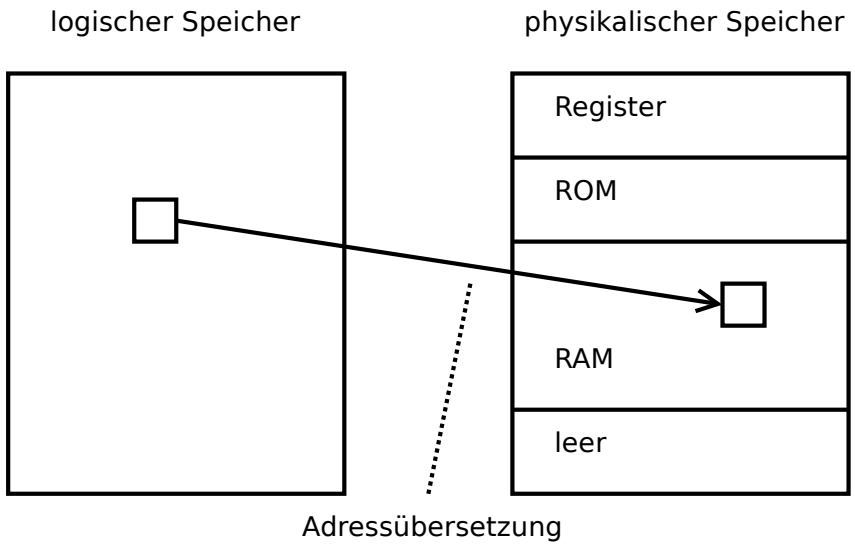
# Motivation

- Speicherbedarf einer Anwendung und HW-Speicherangebot eines Systems sind ggf. um Größenordnungen unterschiedlich
  Bsp.: Prg. allokiert 2 GB Speicher auf System mit 1 GB RAM
- Programm soll auf unterschiedlicher HW mit unterschiedlicher Speicherstruktur ausgeführt werden
  - könnte jedesmal für Zielsystem angepasst werden
  - aber was, wenn das Programm 3x parallel auf demselben System ausgeführt wird? $\Rightarrow$ Globale Variablen an derselben Adresse?
- Schutz der Anwendungsteile vor gegenseitigem Überschreiben der Daten (durch SW-Fehler oder Angriffe)

$\Rightarrow$ **Skalierung, Portabilität, Schutz/Stabilität**

# Physikalische und logische Adressräume

Eine "Adresse" ist die Grundlage für den Zugriff auf ein Speicherwort (z.B. 32-Bit)




Logischer Speicher:

- Programm-Sicht
- homogen (keine Speicherarten, Ausnahme: Harvard)
- linearer Adressraum
- Wortbreite einheitlich (= Systemarchitektur, z.B. 32-Bit)

⇒ Speicher*abstraktion*

in C: uno 1 Art v. Pointer

Physikalischer Speicher:

- Technologie-abhängige Eigenschaften
- Speicherarten
- Zugriffszeiten
- r/w (beschreibbar?)
- Wortbreite

⇒ physikalische *Realität*

# Zugriffszeiten (Zeit-Kosten)

- Eine Motivation für Speichermanagement: mehr RAM "vorgaukeln"
- Beobachtung: Speichergrößen und -zugriffsgeschwindigkeiten skalieren gemeinsam ( Modulgröße $\sim$ Zugriffszeit, $N \sim t_{acc}$)

| | N | $t_{acc}$ |
|---|---|---|
| Register | 8-128 | $10^{-9}$s |
| Cache 1. Lev. | n KB | $< 10^{-8}$s |
| Cache 2. Lev. | n MB | $< 10^{-8}$s |
| RAM | $2^{30}$ Byte | $10^{-8}$s |
| HDD (sekundär, SSD,) | $2^{40}$ Byte | $10^{-5}$s |
| HDD (sekundär, mag) | $2^{40}$ Byte | $10^{-2}$s |
| FDD, Tape (tertiär) | $> 2^{50}$ Byte | groß |

- ideal: schneller, großer Speicher
- real: kleine schnelle Speicher maximal ausnutzen und Daten aus großen langsamen Speichern rechtzeitig nachliefern

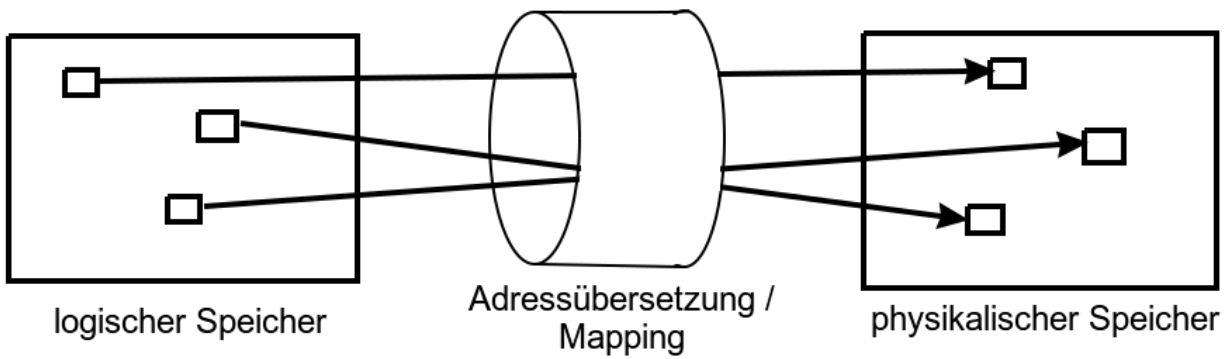
# Adressübersetzung (AÜ): logisch

Ziel: finde für eine logische Adresse (Sicht des Programms) eine physikalische Adresse (im realen Speicher)

"*Wo* ist mein Datenwort abgespeichert?"

"logisch" meint hier: *was* soll die Übersetzung leisten?

- Prozesse (= Programme in Ausführung) fordern Speicher an
  - realer Speicher wird dafür reserviert und
  - es wird eine Abbildung der logischen auf die physikalische Adresse etabliert

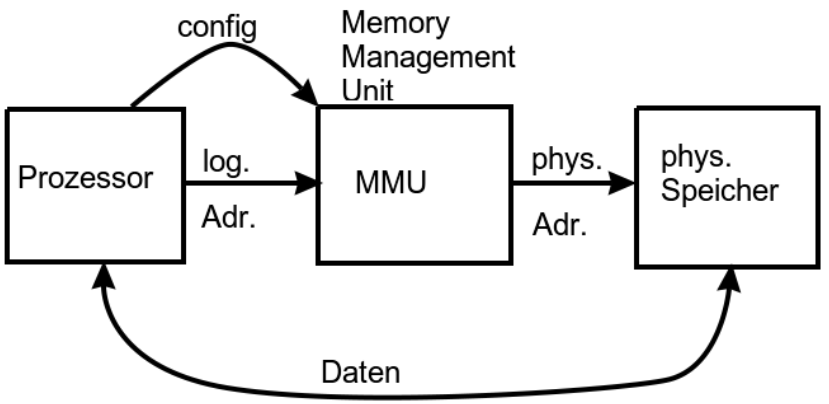


# Adressübersetzung: physikalisch

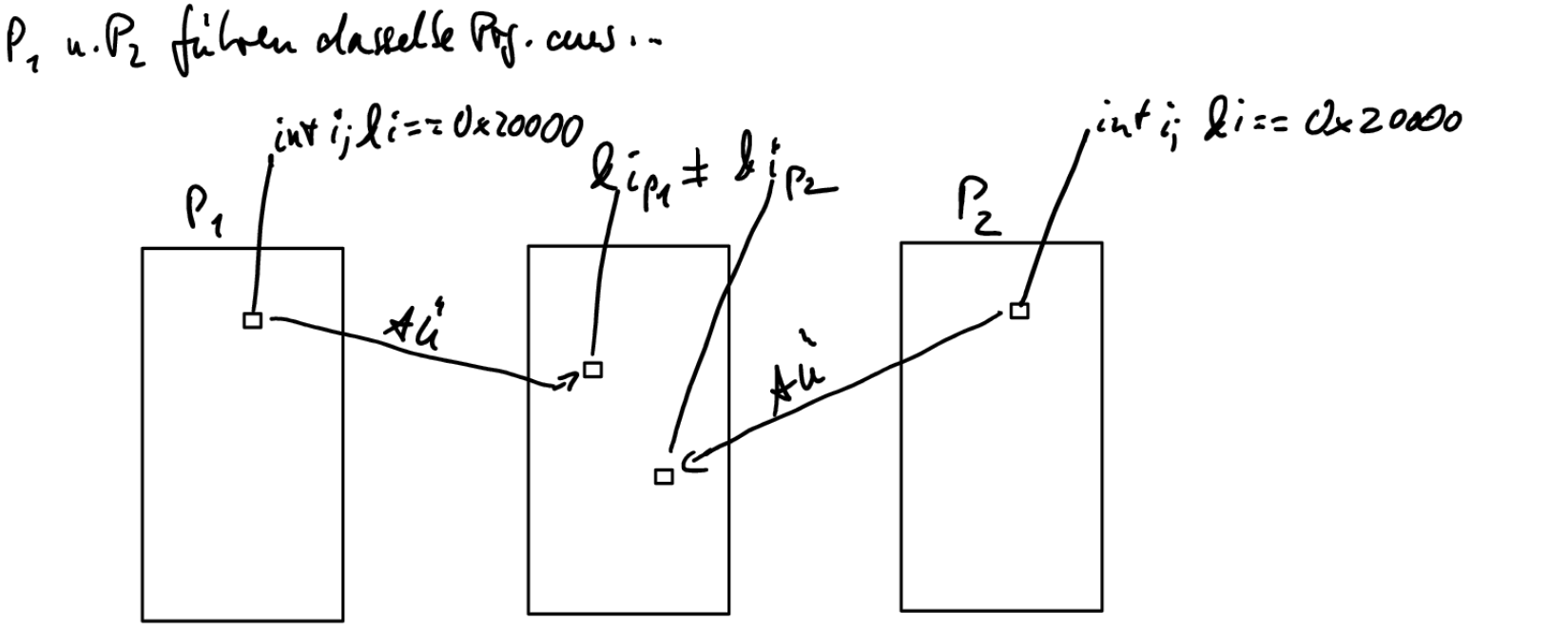
"physikalisch" meint hier: *wie* wird die Übersetzung realisiert?

- Aufteilung des Adressraums in
  - Seiten: einheitliche Größe
  - oder Segmente: veränderliche Größe, z.B. Größe des Speicherraums eines Prozesses
- Abbildung wird durch die HW-Komponente "Memory Management Unit" (MMU) realisiert

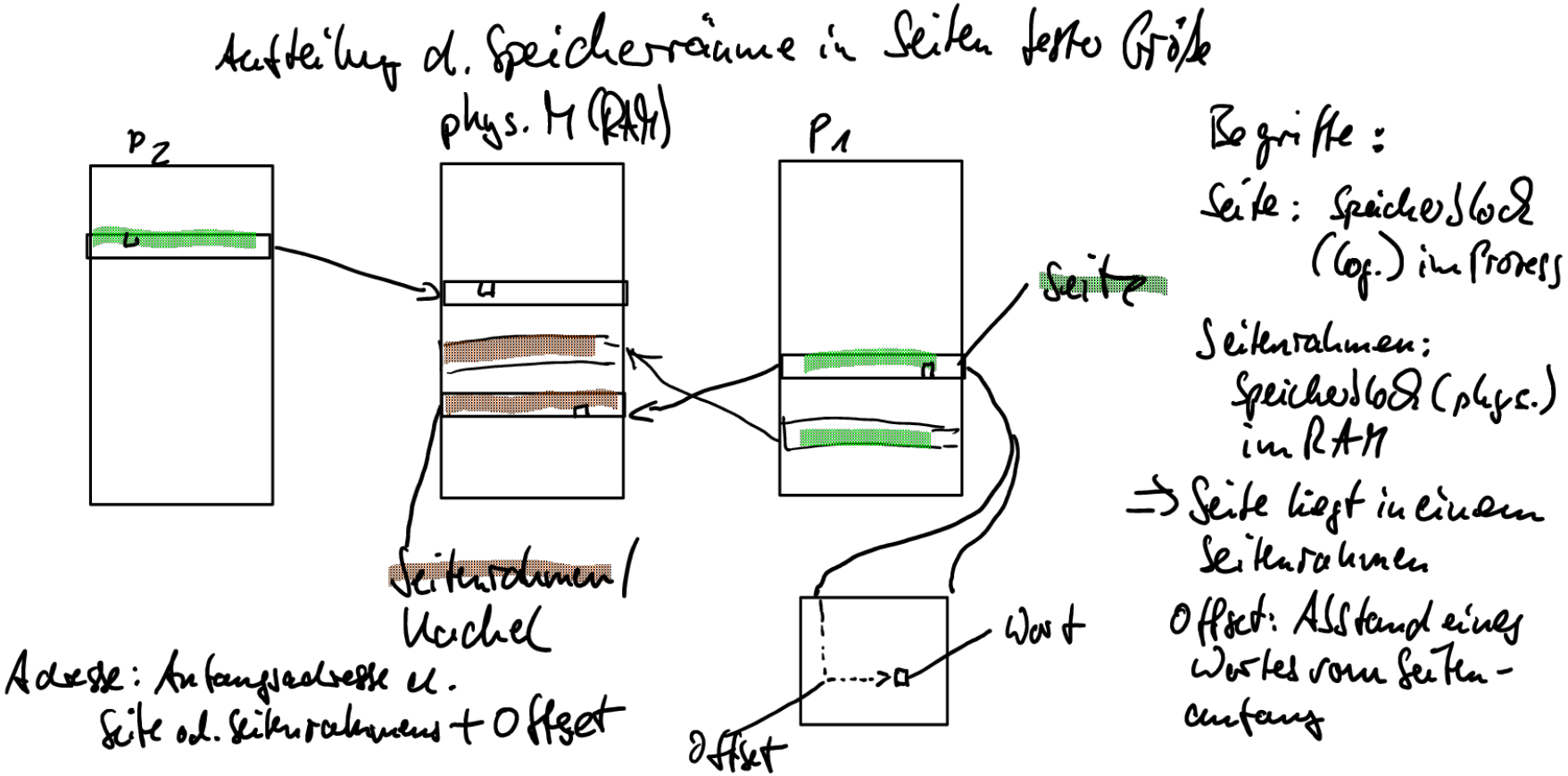
Technische Abbildung der logischen auf physikalische Adressen mittels MMU



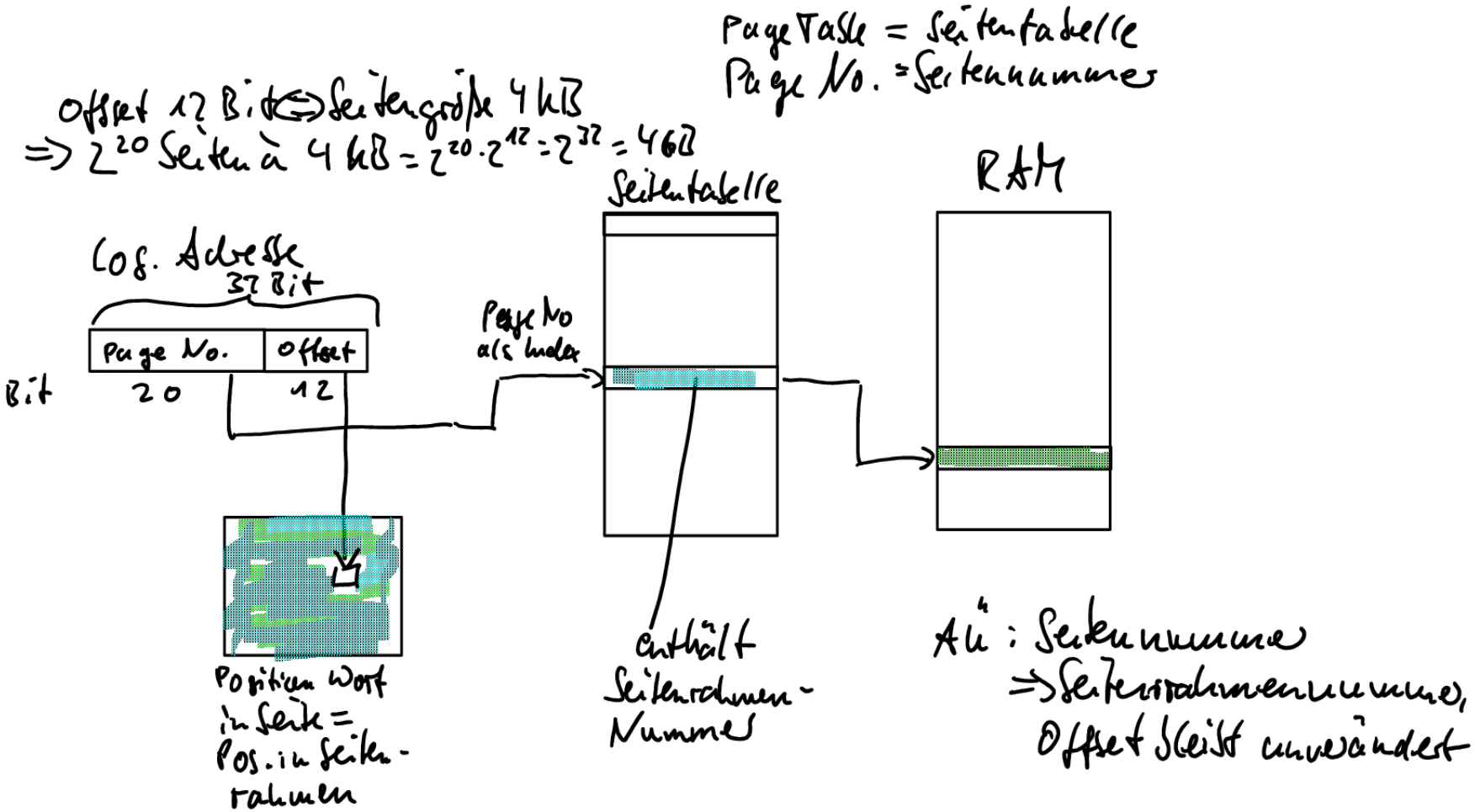
# Speicherabbildung mehrerer Prozesse



# Seitenbasiertes Speichermanagement

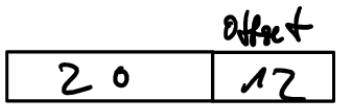


# Abbildung virtueller Adressen



# Struktur von Adressen und Skalierung

32 Bit



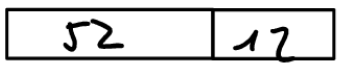

$\Rightarrow 2^{20}$ Seiten

Annahme: 4 Byte/Tabelleneintrag

Größe 1 Seitentabelle:

$2^{20} \cdot 4\,\text{Byte} = 2^{22}\,\text{Byte} = 4\,\text{MB}$

64 Bit




größer als RAM 1980er?!

Seitentabelle:

$2^{52} \cdot 8\,\text{Byte} = 2^{55}\,\text{Byte} = 32\,\text{PB}$

↑ für 52-Bit Rahmennummer

(Beispiele)

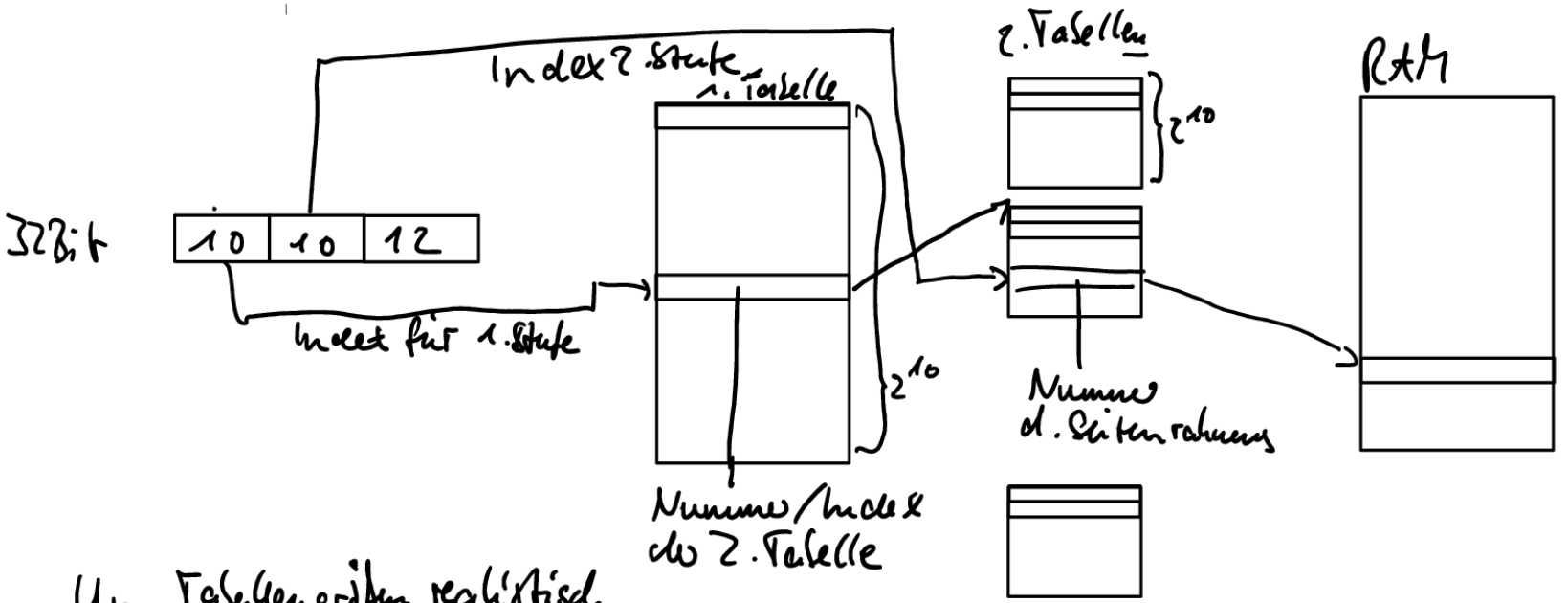
Beobachtung:

Prozess nutzt log. Adressraum i.d.R nie vollständig aus

⇒ nur so möglich, n Prozesse mit realem RAM zu bedienen

Idee: pro Prozess eine Seitentabelle

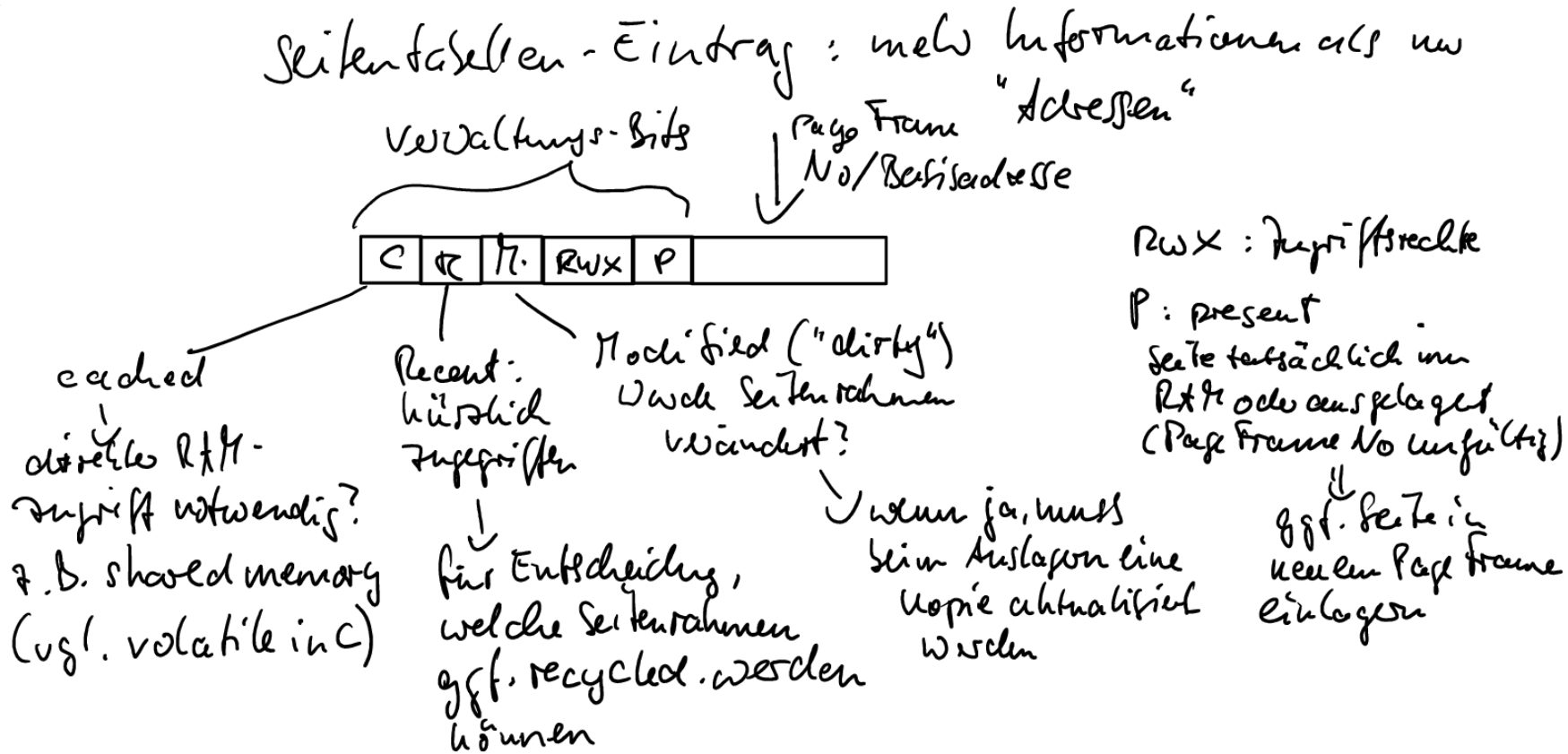
# Mehrstufige Seitentabellen




Um Tabellengrößen realistisch zu halten, wird mehrstufig aufgebildet.

Vorteil: 1. Tabelle (muss ∃) ist wesentlich kleiner:
z.B. $2^{10} \cdot 4$ Byte = 4 kB (Seitengröße!)
Und es werden nur genutzte Tabellen der 2. Stufe angelegt (inkrementell), also initial auch 4 kB.

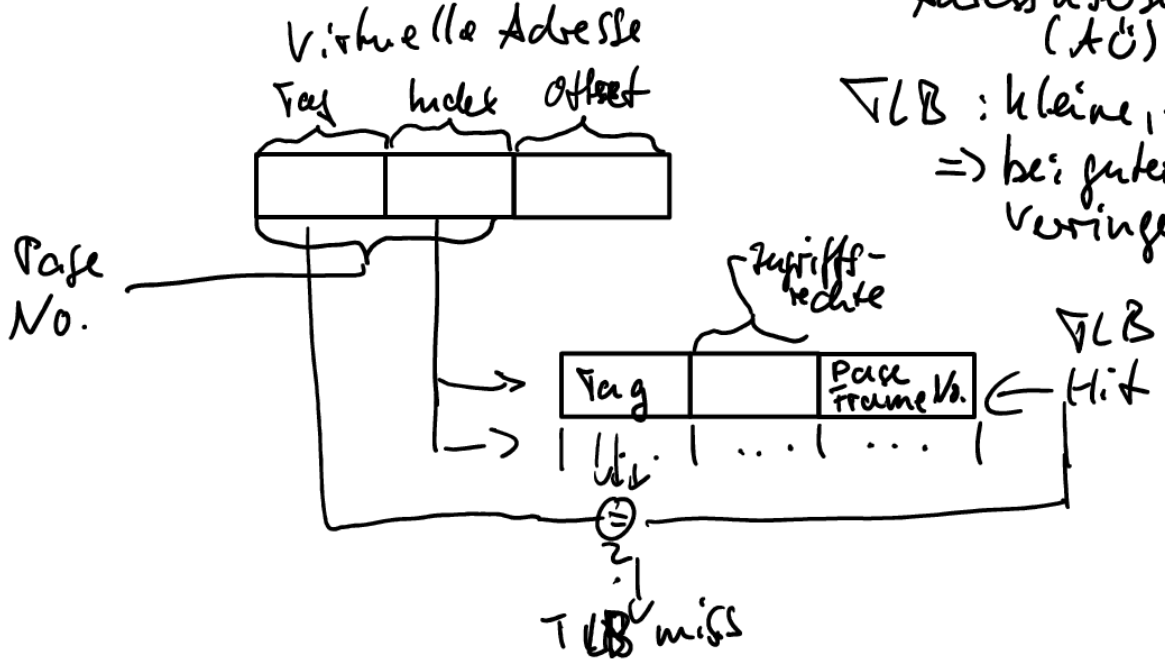
# Management-Informationen



# Translation Lookaside Buffer (TLB)

TLB-Aufgabe: Abkürzung für Adressübersetzung zu finden (AÜ)

TLB: kleine, schnelle Speicher
⇒ bei guter TLB-Trefferquote eff. Verringerung eines AÜ-Prozesses




Prinzip: häufig/kürzlich verwendete Seiten werden in eine Extra-Tabelle notiert, wenn die AÜ erfolgt ist.

# TLB-Abbildungsklassen (TLB Mapping)

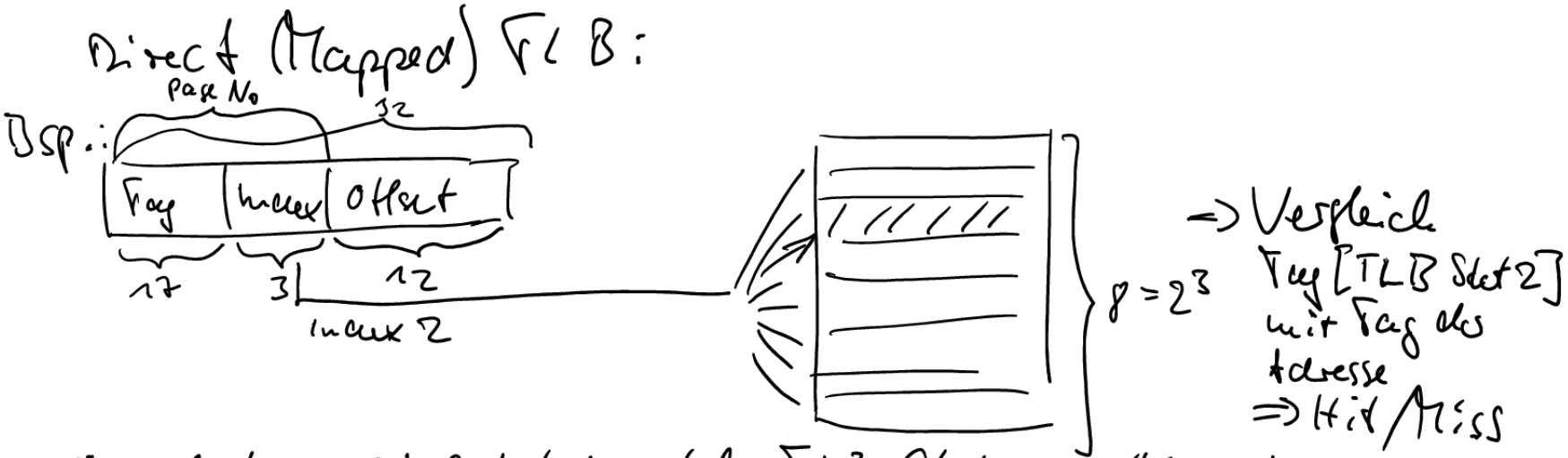
Aufteilung der Seitennummer in Tag und Index bestimmt die TLB-Architektur

Extremfälle: Index 0 Bit : Fully Associative

Index = $\log_2$ (Anz. TLB-Einträge): direct mapped

Mix der Strategien: Set Associative

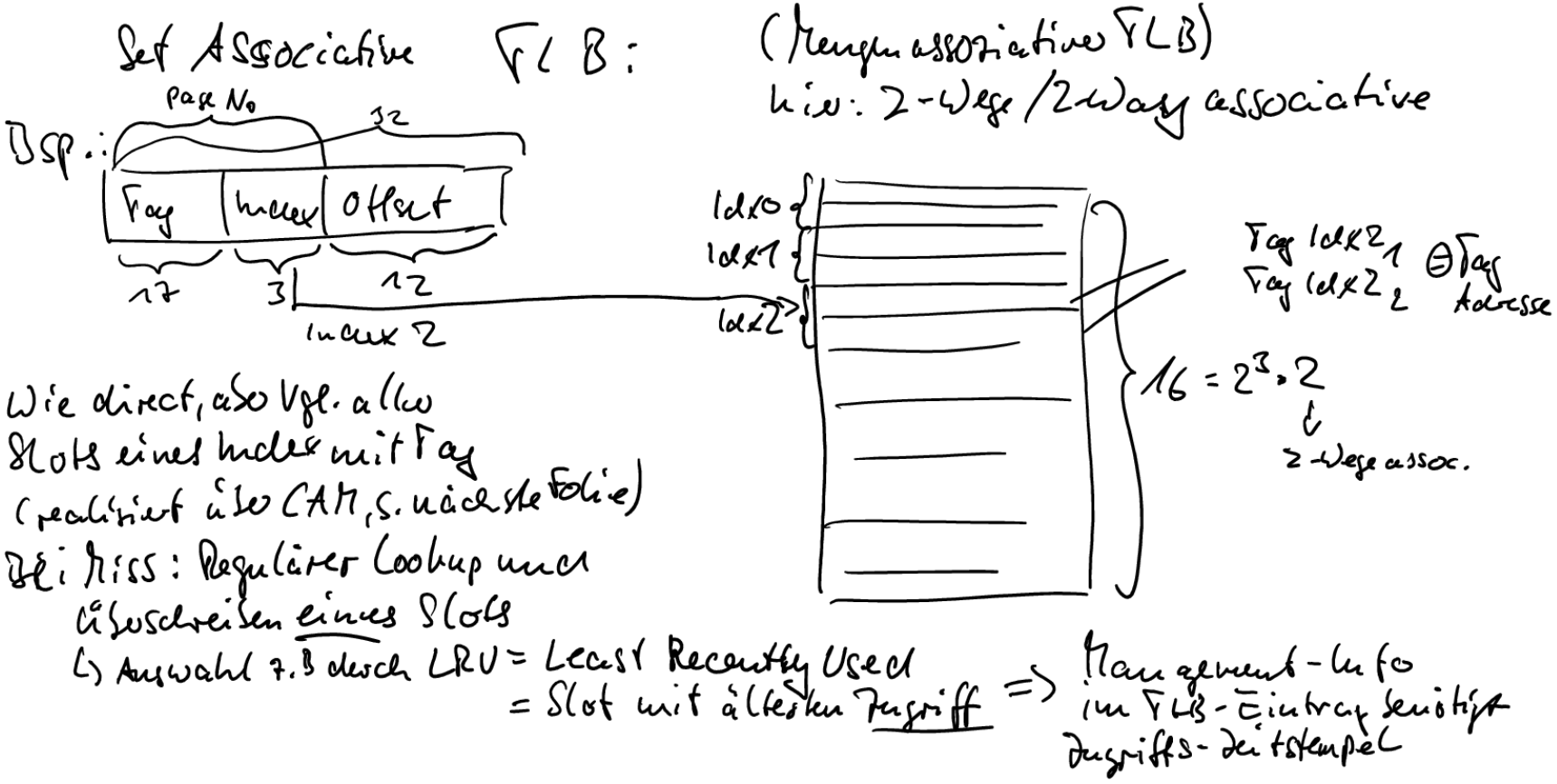
# Direct Mapping




Für jede Adresse ist festgelegt, welcher TLB-Slot sie abbilden kann.

- Index nimmt untere Bits der Page No. ein ⇒ viele Seiten (Adressen mod $2^3$ · Seitengröße) würden denselben TLB-Slot nutzen, also bei linearem Prg.-ablauf erfolgt lineare Belegung (jeweils der nächste Slot)
- Wenn TLB miss (anderes Tag im Slot), wird regulärer Page Table lookup durchgeführt und TLB Slot überschrieben

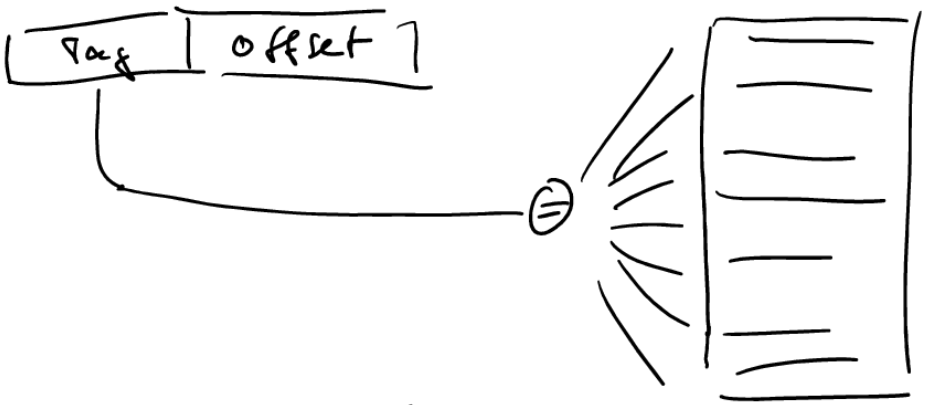
# Set Associative Mapping




- Wie direct, also Vgl. aller Slots eines Index mit Tag (realisiert über CAM, s. nächste Folie)
- Bei Miss: Regulärer Lookup durch Überschreiben eines Slots
  - Auswahl z.B. durch LRU = Least Recently Used = Slot mit ältestem Zugriff ⇒ Management-Info im TLB-Eintrag speichert Zugriffs-Zeitstempel

# Fully Associative Mapping

Vollassoziative TLB: Index-Bits = 0 ⇒ Adresse kann in jedem Slot liegen




Vergleich aller Tags in allen Slots wäre hinaus aufwendig, daher Special-HW: Content Addressable Memory (CAM)

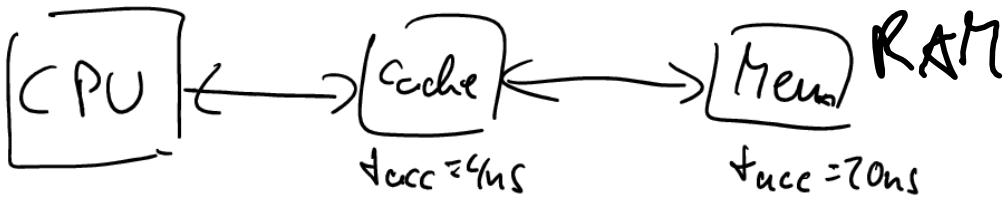
CAM: Input: gesuchter Inhalt, Output: gefundener Slot (oder Miss)

Parallel-Vergleich aller Tag-Bits der Adresse mit Bits aller Tags in allen Slots realisiert in HW (viele parallele Gatter)

Hoher HW-Aufwand, aber konstanter Aufwand ($t_{pd}$) und optimale LRU (o.ä.)-Effizienz

# Caching

Methode: Durchschnittliche Speicher-Zugriffszeit durch kleine, schnelle Zwischenspeicher erhöhen (Cache ≈ cacher, verstecken)




Wenn Daten häufig im Cache gefunden werden, sinkt $t_{acc}$ insgesamt.

oft separate Instruktions- und Datencaches (auch ohne Harvard-Arch)

→ unterschiedliche Adressbereiche hinsichtlich Lokalität

# Cache Lines und Blocks

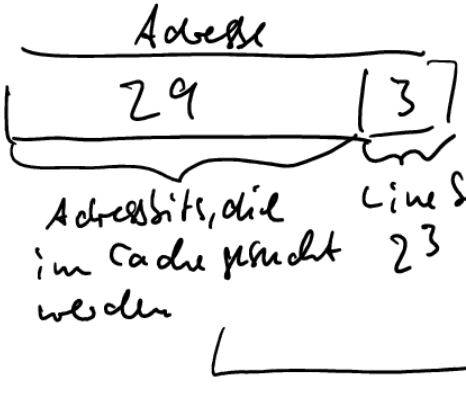
Arbeitsprinzip für Zugriff und Mapping ähnl. TLB, auch Aktualisierung nach Miss, usw:

Organisationsform basiert auf

Lines: Adressiert werden nicht einzelne Wörter, sondern n Wörter auf einmal

und

Blocks in Lines, die bei einem Miss nachgeladen werden



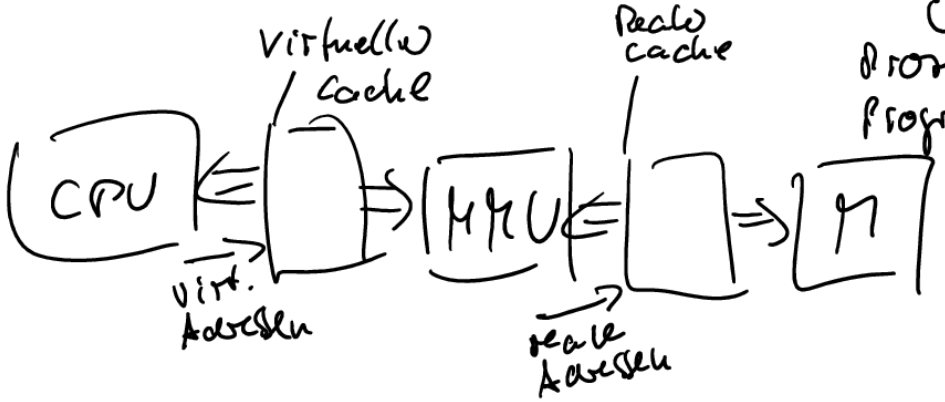


Direct
Set Assoc. → Tags/Index
Fully Assoc.

Offsets irrelevant bis hier

# Virtuelle und reale Caches

Problem: Speicherzugriff benutzt virtuelle und reale Adressen
(Prozess / Programmsicht) (phys. (RAM+ROM+I/O-Regs))

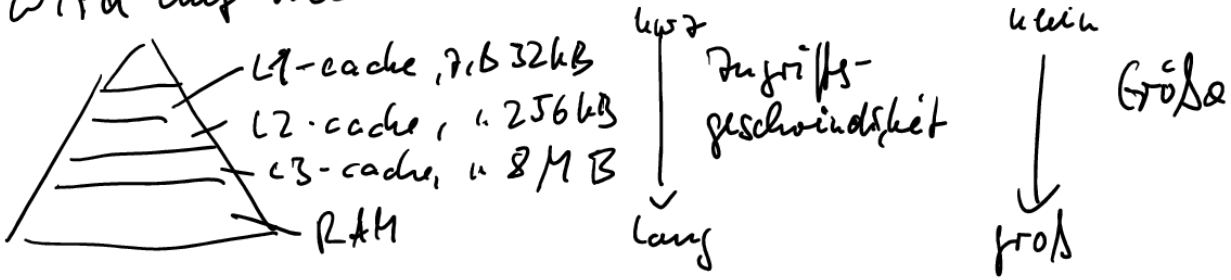



Vorteil virtueller Cache: schneller, da keine Adressübersetzung vorher nötig
(Datenverbindung Cache → CPU wäre trotzdem direkt)

Vorteil realer Cache: Daten sind unabhängig von der Prozesssicht,
Daten des virtuellen Caches, die von einem Prozess angefordert wurden, sind für den nächsten Prozess trotz gleicher virtueller Adresse völlig andere → werden ungültig

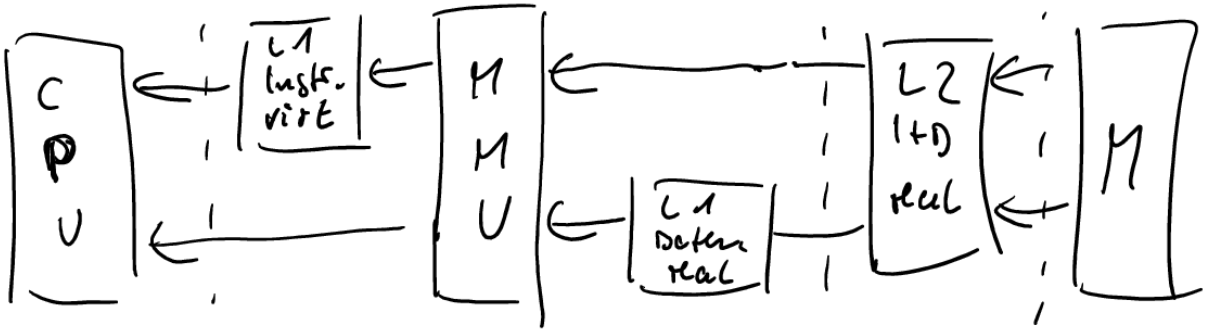
# Mehrstufige Caches

Prinzip kleiner, schneller Cache – großer, langsamerer Hauptspeicher wird auf mehrere Cache-Ebenen erweitert:




⇒ Bei ausgewogenem Design sind die effektive mittlere Zugriffsgeschwindigkeit

Oft Mix aus virtuellen/realen Caches, oft getrennt für Instruktionen und Daten



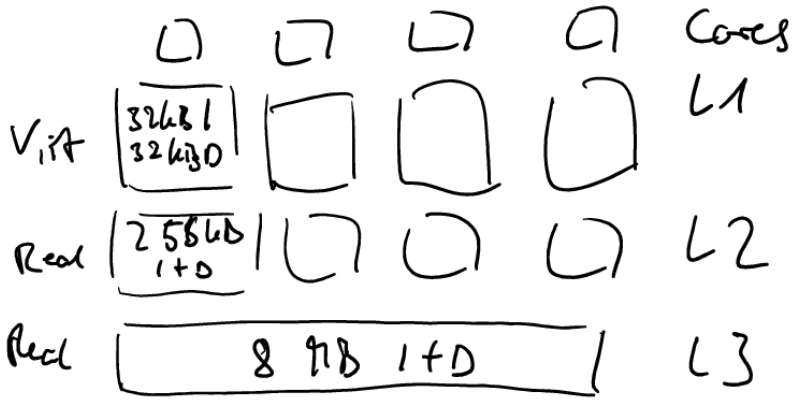
# Caches, Prozesse und shared Memory

- Shared Memory
  - virtueller Cache: Daten für neuen Prozess noch gültig, aber virtuelle Adresse unterschiedlich
  - realer Cache: kein Problem, identische reale Adresse

- Invalidierung des virtuellen Caches bei Prozesswechsel
  - ↳ alternativ: zusätzliches Tag im Cache mit Prozess-ID

---

Multiprocessor/-core Cache, Bsp. Core i7




gemeinsamer Cache über mehrere Cores, da ein Prozess Thread auf unterschiedliche Cores wechseln könnte

# Cache-Kohärenz

Problem bei
- mehreren parallelen Caches
- Cache und Speicherzugriffe von I/O-Geräten
- Multiprozessor-Architekturen

Haupt-
- Speicher wurde durch A gecached und von B verändert
- In Cache von A wurde neuer Wert geschrieben, aber noch nicht in Hauptspeicher zurückgeschrieben

⇒ Verletzung der Cache-Kohärent: Übereinstimmung zw. HS und Cache(s)

Lösung durch Cache-Kohärent-Protokolle (Coherence): Regeln für das Laden/Speichern/Aktualisieren zwischen HS und Cache

Unterstützung durch HW: Bus Snooping: Cache hört aktiv den Adressbus auf Zugriffe ab, die von ihm gecachte Adressen betreffen

# Aktualisierung

Zwei Grundstrategien:

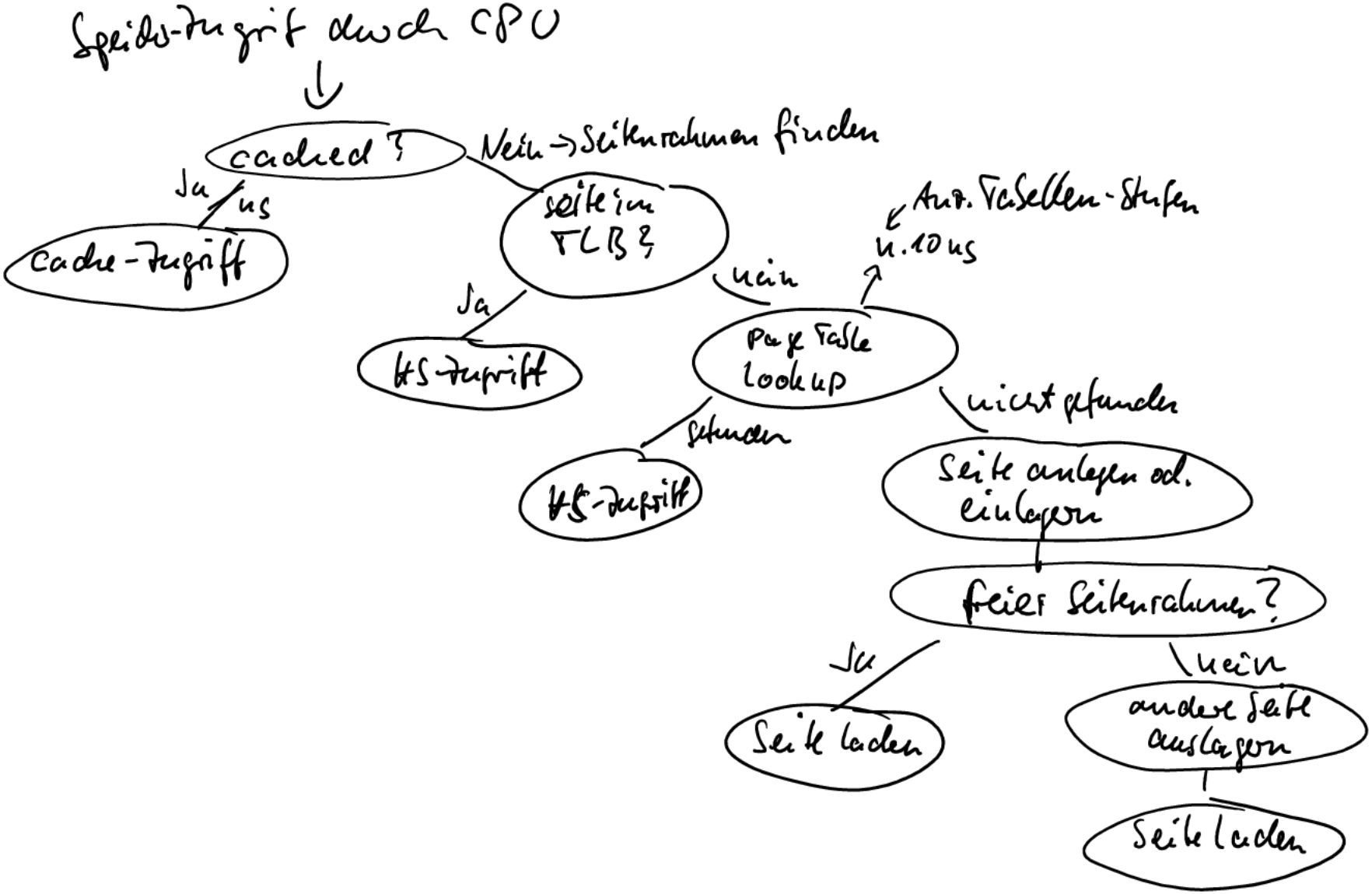
- Write Back: CPU schreibt neuen Wert in Cache, Aktualisierung des HS erfolgt später

  Vorteil: Schreibzugriffe schnelle
  Nachteil: HS zeitweise ungültig
  ggf Lösung: Cache teilt Änderung mit, so dass andere Caches diese Adresse als "ungültig" markieren

- Write Through: Wert wird sofort in Cache und HS geschrieben

  ⇒ Langsamere Schreibzugriffe, aber keine Inkonsistenz

# Speicherzugriff im Überblick



# Memory Protection Unit (MPU)

- Reduzierte Variante der MMU
- Speicherschutz implementiert, aber Adressübersetzung fehlt
  - Oft kein Sekundärer Speicher (HDD) in Embedded-Systemen

  $\Rightarrow$ Möglichkeit, Seiten auszulagern fehlt
  - Auch hinsichtlich deterministischem Zeitverhalten ist "schlankes" Speichermanagement vorteilhaft

  $\Rightarrow$ Nur Hauptspeicher nutzen, der auch physikalisch vorhanden ist

  $\Rightarrow$ Anwendungen müssen speichereffizient umgesetzt sein
- Schutzmechanismen unterstützen Embedded-OS
  - Anlegen seprater Speicherräume für BS und einzelne Tasks
  - Durchsetzung der Privilegstufen auch bei Speicherzugriffen

# Beispiel: MPUs in der ARM-Familie

- MPUs bereits bei einfacheren, günstige Varianten erhältlich (schon ab Cortex M0+ !)
- MMUs nur bei Hochleistungsvarianten (= auch mehr Speicher)
- Optionale MPU auf Cortex-M3:
  - Acht "Regionen" (≈ Segmente) unterschiedlicher Größe mit separaten Attributen definierbar
    - Können überlappen
    - Jeweils acht gleich große Subregionen ein- und ausschaltbar
    - Rest liegt in der 9. Region
  - Ausführungsreihenfolge einer Region kann Pipelinig übergehen ("Normal", "Device", "Strongly Ordered")
  - Shared Memory und Codeausführungsverbot pro Region möglich
  - Zugriffsverletzungen ⇒ Memory Management Fault