# Kapitel 6 – Weitergehende Architekturkonzepte

## 1. Pipelining und Parallelverarbeitung

## 2. **Speicherorganisation**

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik

WS 2015/16

UNI FREIBURG

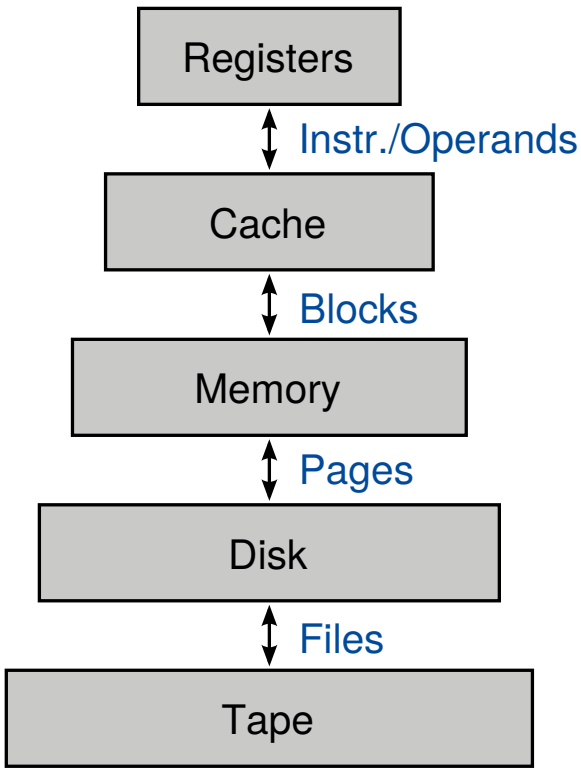
# Speicherorganisation im Überblick

- **Fragen**:
  - Wie kann verhindert werden, dass ein Rechner durch Hauptspeicherzugriffe zu sehr verlangsamt wird?
  - Was passiert, wenn nicht alle Daten in den Hauptspeicher passen?
- **Begriffe**:
  - Speicherhierarchie
  - Cache
  - Virtueller Speicher, Verdrängungsstrategien
  - Festplatte, ...

# Gründe für komplexe Speicherorganisation

- Ein Zugriff auf eine Hauptspeicherzelle ist langsamer, als ein Zugriff auf ein Register.
  - Hauptspeicherzellen sind DRAM-Zellen (dynamische Speicherzellen), während Register in der Regel SRAM-Zellen (statische Speicherzellen) sind!
  - Bei einem Registerzugriff kommt man ohne Bus-Operation aus!
- **Idee**: Man stellt dem Prozessor einfach einige Mbyte Register zur Verfügung.
  - Aber: SRAM-Zellen sind wesentlich größer als DRAM-Zellen (Faktor 3-4).
  - So abwegig ist die Idee nicht!
  - Mit der weiteren Technologieentwicklung (noch kleinere Strukturen) wird die verfügbare Chip-Fläche vorwiegend dazu benutzt werden, um schnellen Speicher zu integrieren.

# Speicherorganisation heute




Programm
1-8 bytes

Cache
8-128 bytes

OS
512-4K bytes

Benutzer
MByte, GByte, TByte

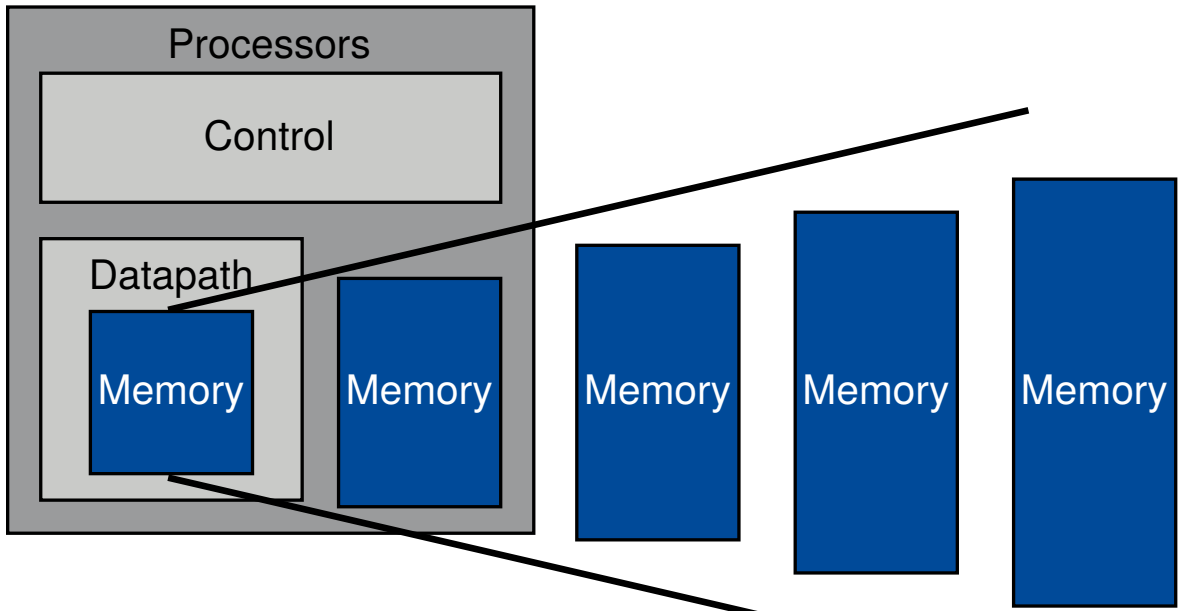
Upper Level

Klein, schnell, teuer

Groß, langsam, billig

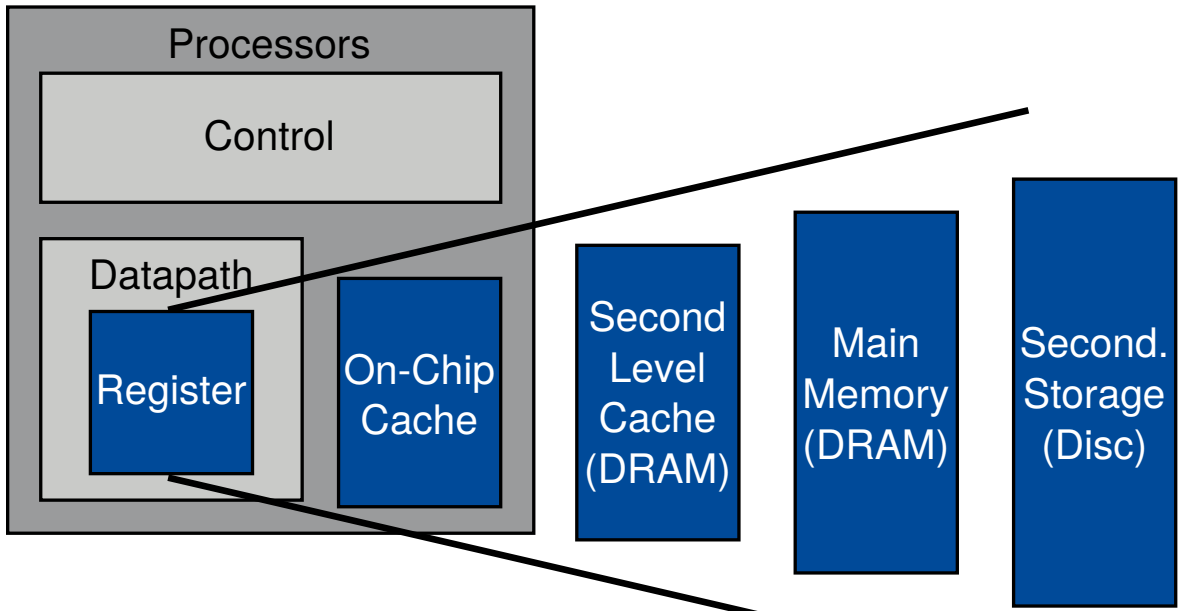
Lower Level

# Speicherhierarchie (1/3)




| | | |
|---|---|---|
| **Geschwindigkeit:** | Am schnellsten | Am langsamsten |
| **Größe:** | Am kleinsten | Am größten |
| **Kosten:** | Am teuersten | Am billigsten |

# Speicherhierarchie (2/3)



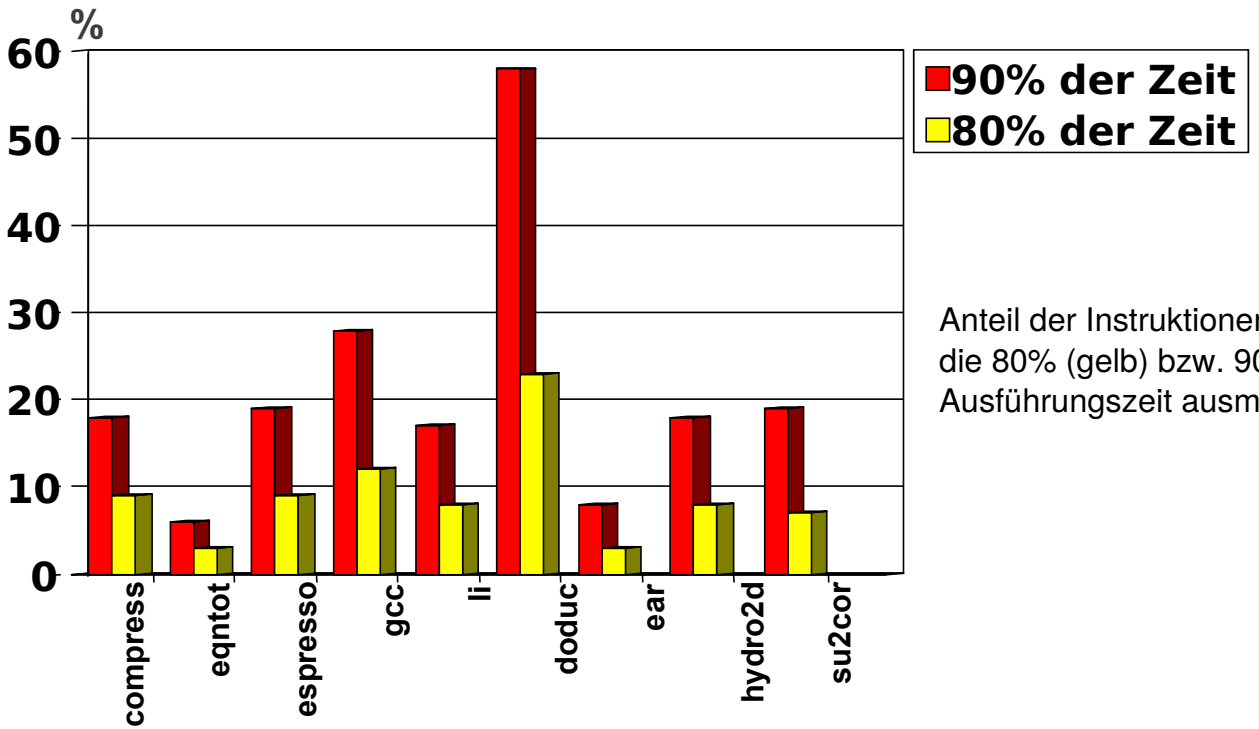

| | | |
|---|---|---|
| **Geschwindigkeit:** | Am schnellsten | Am langsamsten |
| **Größe:** | Am kleinsten | Am größten |
| **Kosten:** | Am teuersten | Am billigsten |

# Speicherhierarchie (3/3)

- Grundprinzip:
  - Wenig schneller und teurer Speicher
  - Viel langsamer und billiger Speicher
  - Speicherhierarchie
- Wieso funktioniert das?
  - Wegen Prinzip der Lokalität!

# Prinzip der Lokalität (1/2)

Typische Programme verbringen 80% der Ausführungszeit in etwa 10% des Gesamtprogramms.




Anteil der Instruktionen des Gesamtcodes, die 80% (gelb) bzw. 90% (rot) der Ausführungszeit ausmachen.

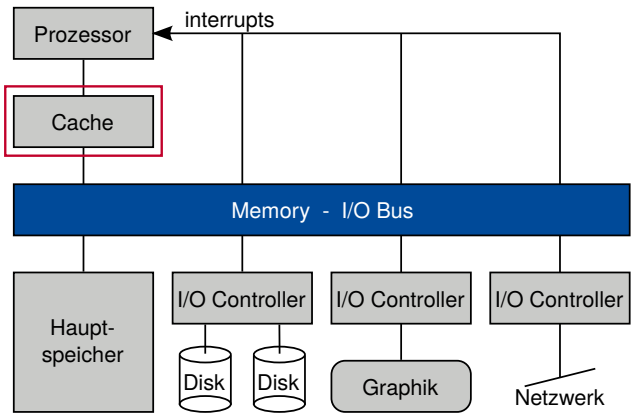
Patterson, Hennessey Computer Architecture a Quantitative Approach

# Prinzip der Lokalität (2/2)

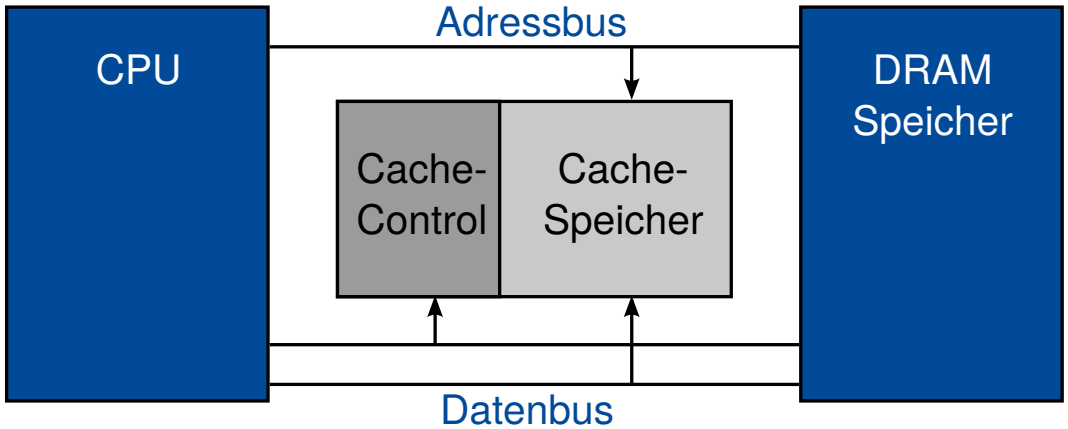
- Es gibt zwei Arten von Lokalität:
  - Zeitliche Lokalität (Locality in Time): Wenn ein Datum verwendet wird, wird es auch bald wieder verwendet werden.
  - Räumliche Lokalität (Locality in Space): Wenn ein Datum verwendet wird, werden auch die Adressen in dessen Nähe bald verwendet werden.
  - Siehe z.B. Schleifen ...
- Wegen Lokalität kommt man auf höheren Ebenen der Speicherhierarchie mit weniger Speicher aus.

# Caches (1/2)

- Zwischenspeicher für Daten
- Stellen Daten aus einer niedrigeren Ebene der höheren Ebene bereit (Speicherhierarchie)
- Häufig mehrere Cache-Level
- Kenngrößen:
  - Größe (Anzahl der Cachezellen).
  - Zugriffszeit (lesend).
- Hierarchischer vs. Nicht-hierarchischer Cache
  - Hierarchisch: Cache trennt CPU vom Systembus
  - Nicht-hierarchisch: Es besteht direkte Verbindung zwischen CPU und Systembus



# Caches (2/2)




Schematischer Aufbau Nicht-hierarchischer Cache

- In der Regel besitzt ein Rechner einen getrennten Cache für Instruktionen (Instruktionscache) und für Daten (Datencache).
- Er kann durch ein Anwendungsprogramm nicht explizit adressiert werden.
- Er ist software-transparent, d.h. der Benutzer braucht nichts von seiner Existenz zu wissen.

# Ziel des Cache-Einsatzes

- Versuche stets die Daten im Cache zu halten, die als nächstes gebraucht werden.

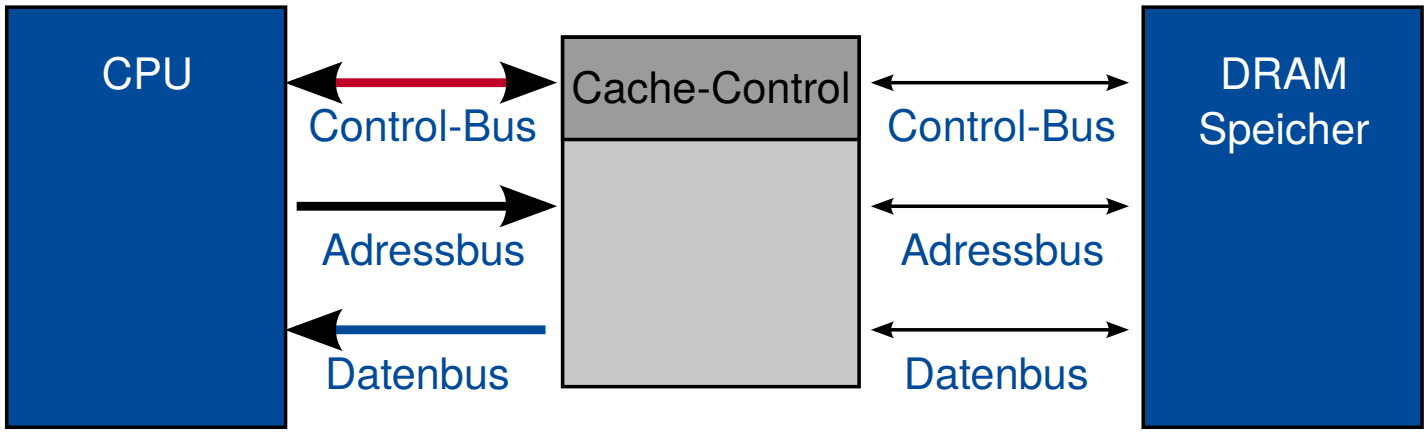
  → Der Prozessor kann die Mehrzahl der Zugriffe auf dem Cache und nicht auf dem langsamen DRAM-Speicher ausführen.

- Voraussetzung, um dieses Ziel erreichen zu können:
  - Lokalitätsprinzip

# Prinzipielle Funktionsweise: Lesezugriff

- Lese Datum aus dem Arbeitsspeicher unter Adresse *a*.
- CPU überprüft, ob eine Kopie der Hauptspeicherzelle *a* im Cache abgelegt ist.
  - Falls ja (Cache Hit),
    - so entnimmt die CPU das Datum aus dem Cache. Die Überprüfung und das eigentliche Lesen aus dem Cache erfolgt in einem Zyklus, ohne einen Wartezyklus einfügen zu müssen.
  - Falls nein (Cache Miss),
    - so greift die CPU auf den Arbeitsspeicher zu,
    - lädt das Datum in den Cache und
    - lädt das Datum gleichzeitig in die CPU.
    - **Anmerkung**: Mit jedem Datum wird auch dessen umgebender Block von Daten („cache line") geladen in der Erwartung, dass folgende Zugriffe auf diese Daten erfolgen (siehe auch räumliche Lokalität, hier aber nicht weiter betrachtet).

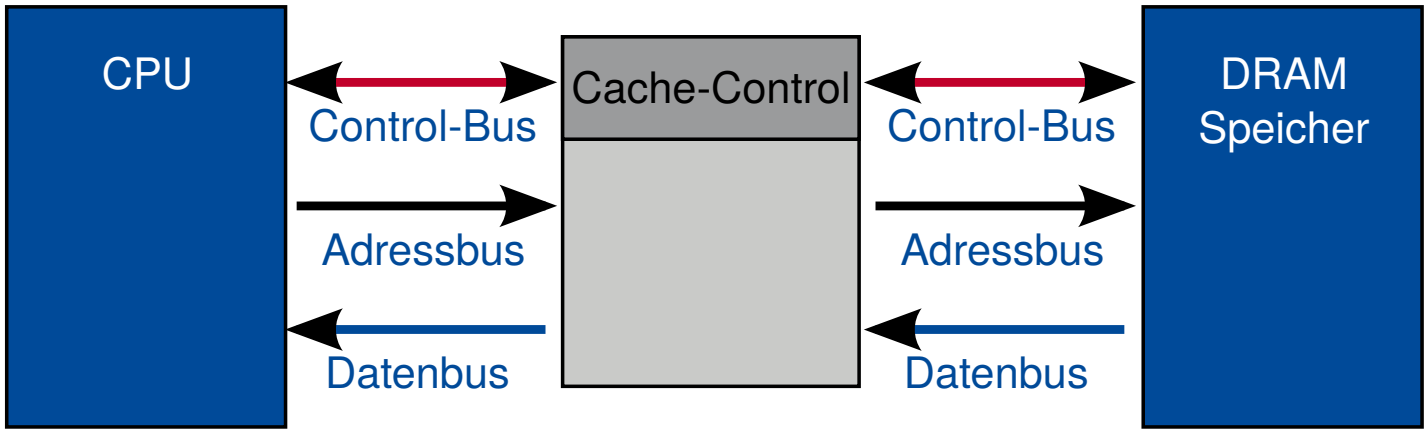
# Illustration des Lesezugriffs: Cache Hit




Schematischer Aufbau Hierarchischer Cache

- Angefragtes Datum befindet sich im Cache und wird direkt aus Cache an CPU weitergegeben
- Systembus wird nicht belastet

# Illustration des Lesezugriffs: Cache Miss




... einige Wartezyklen

Schematischer Aufbau Hierarchischer Cache

- Angefragtes Datum befindet sich nicht im Cache
- Hauptspeicher wird über Systembus nach dem Datum angefragt
- Datum wird im Cache abgelegt und an CPU weitergegeben

# Mittlere Zugriffszeit beim Lesen

- $c$: Zugriffszeit des Caches
- $m$: Zugriffszeit beim Hauptspeicher
- $h$: Trefferrate

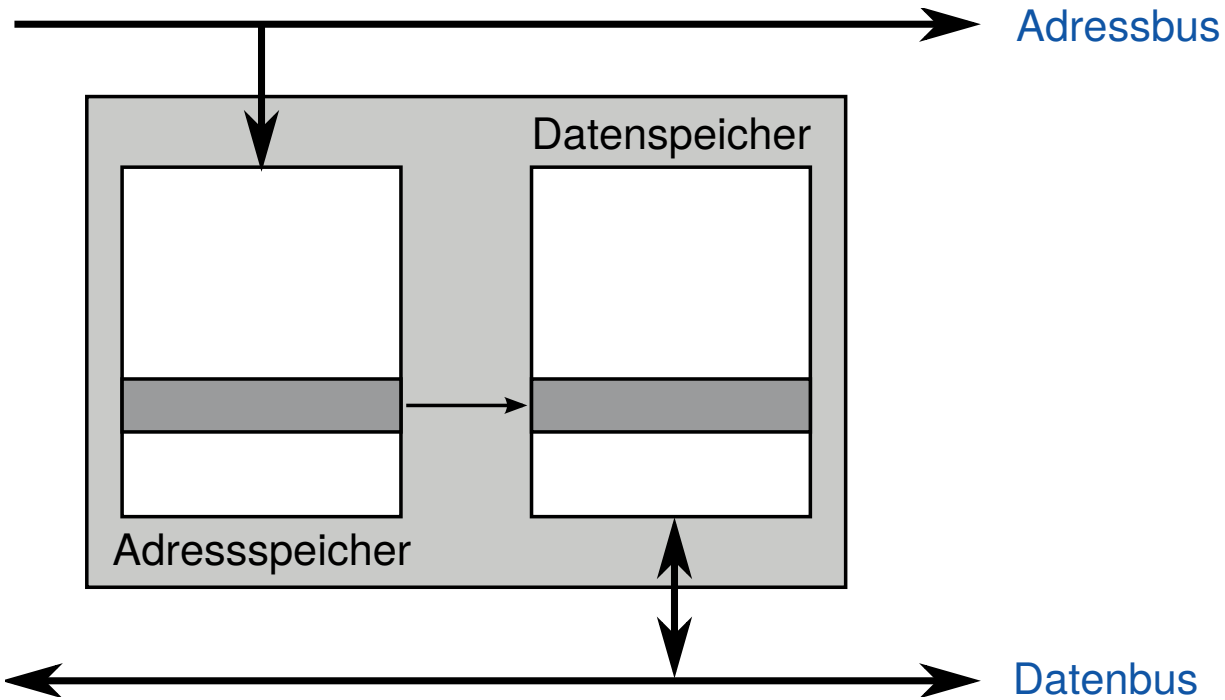
→ Durchschnittliche Zugriffszeit: $c + (1 - h) \cdot m$

- Rechenbeispiel für $c = 50\ ns$ und $m = 200\ ns$:

| Trefferrate $h$ | ∅-Zugriffszeit |
|---|---|
| 50% | 150 *ns* |
| 60% | 130 *ns* |
| 70% | 110 *ns* |
| 80% | 90 *ns* |
| 90% | 70 *ns* |
| 95% | 60 *ns* |

# Aufbau eines Caches

- Ein Cache besteht aus zwei Speicher-Einheiten, die wortweise einander fest zugeordnet sind.



# Cache als assoziativer Speicher

- **Idealfall**: Assoziativer (inhaltsorientierter) Speicher.
- Die von der CPU angelegte Adresse wird parallel mit allen im Adressspeicher des Caches vorhandenen Adressen verglichen.
- Außerdem kann ein neues Datum an jeder beliebigen freien Stelle im Cache abgelegt werden.

→ Aufwendige Logik für den parallelen Vergleich!

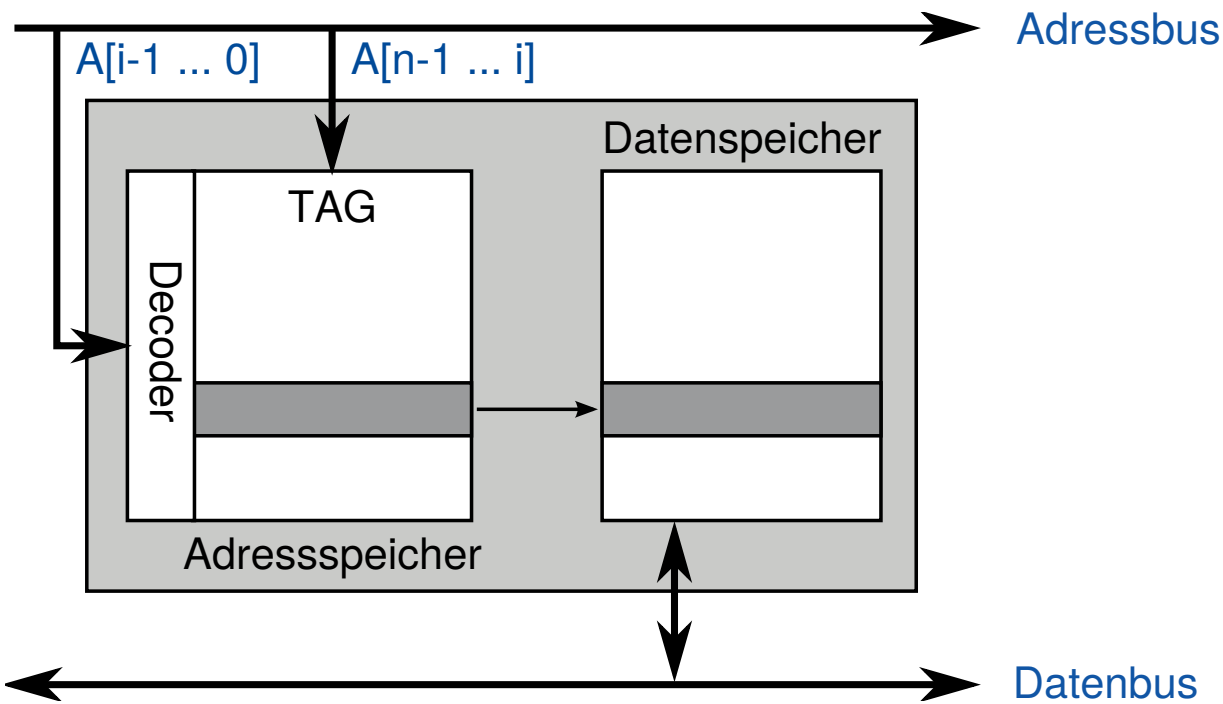
→ Assoziative Speicher nur für kleine Cache-Größen.

# Verdrängen alter Daten aus dem Cache

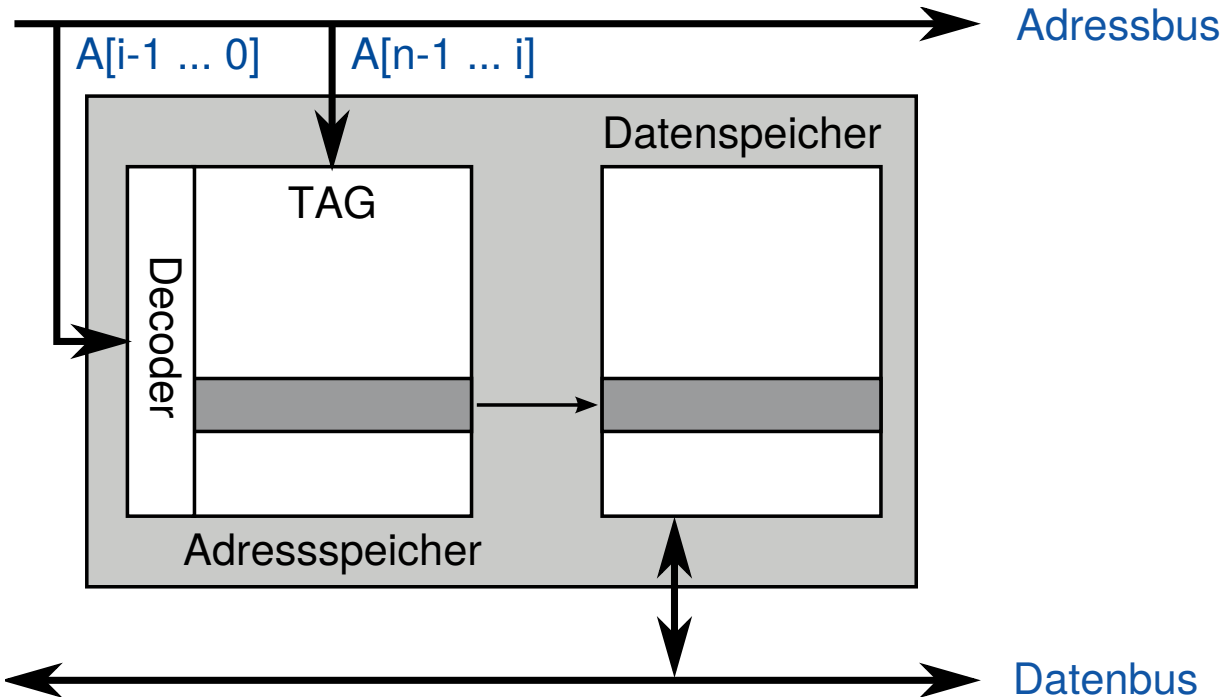
- **Szenario**:
  - Cache Miss.
  - Alle Speicherbereiche des Caches belegt.
- **Ausweg**:
  - Verdränge Datum (Block) aus dem Cache,
  - lade an seine Stelle das gerade benötigte Datum (Block).
- **Denkbare Verdrängungsstrategien**
  - Least Recently Used (LRU)
    Verdränge das Datum (Block), das am längsten nicht benutzt wurde.
  - Least Frequently Used (LFU)
    Verdränge das Datum (Block), auf das am wenigsten zugegriffen wurde.
  - First in, First out (FIFO)
    Verdränge das Datum (Block), das am längsten im Cache ist.

# Direct Mapped Cache (1/2)

- Feste Abbildung der Hauptspeicher-Adressen auf die Cache-Adressen.
  - → Kein assoziativer Speicher nötig.
  - → Keine Verdrängungsstrategie nötig.

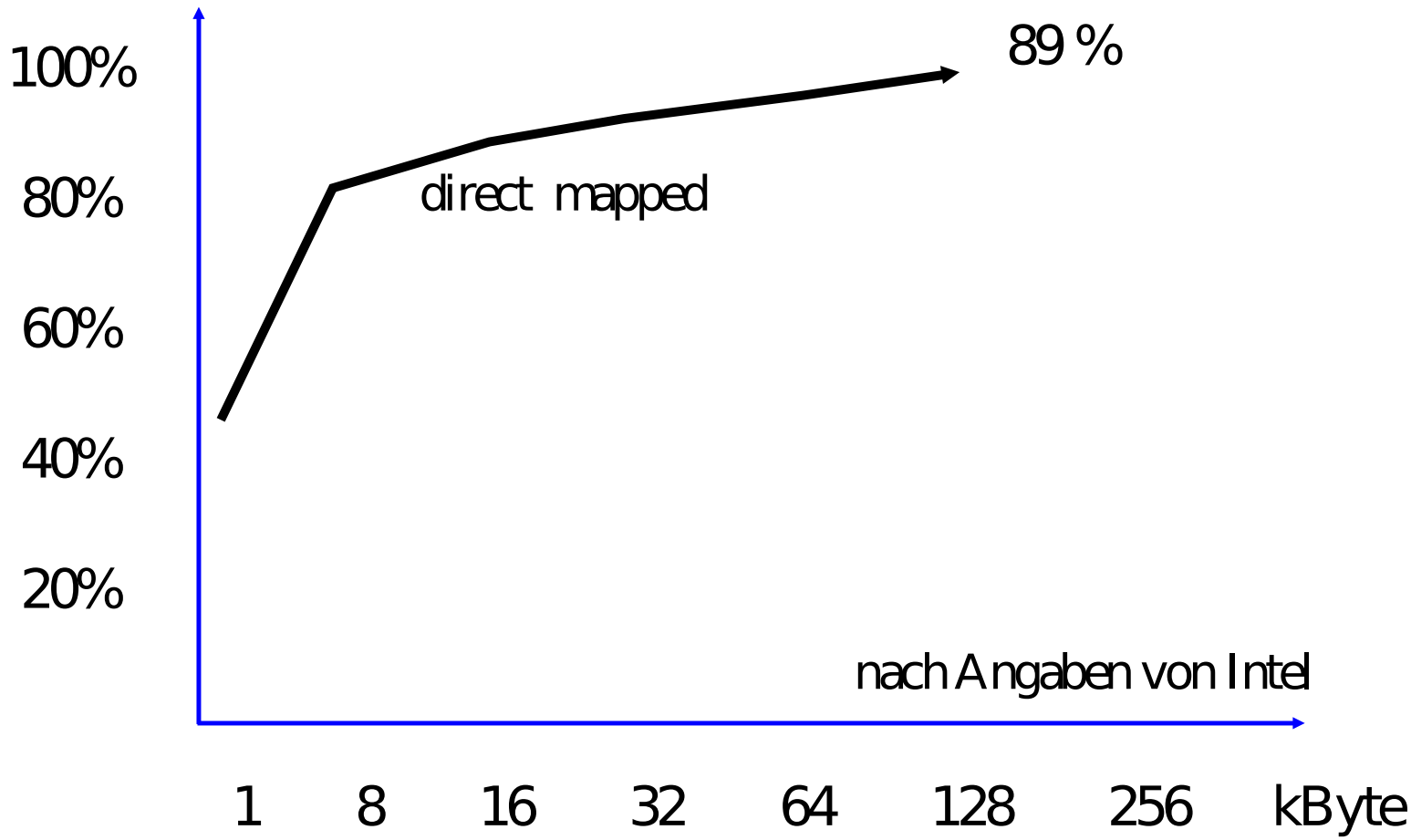


# Direct Mapped Cache (2/2)




- Die von der CPU angelegte Adresse wird in 2 Teile gespalten.
  - Die $i$ niederwertigen Bits adressieren einen Eintrag im Adressspeicher.
  - Dieser Eintrag wird mit den $n - i$ höherwertigen Bits der Adresse verglichen.
- Speicherzellen des Hauptspeichers, deren $i$ niederwertige Stellen gleich sind, werden auf die gleiche Position im Cache abgebildet.

# Direct Mapped Cache: Trefferquote



# Prinzipielle Funktionsweise: Schreibzugriff, Cache Hit

- Schreibe Datum in den Arbeitsspeicher unter Adresse *a*:
- Überprüfung durch CPU ergibt, dass Kopie der Hauptspeicherzelle *a* im Cache abgelegt ist (Cache Hit).
  - Write-through Verfahren:
    CPU schreibt das Datum in den Cache und gleichzeitig in die Hauptspeicherzelle mit Adresse *a*.
  - Write-back Verfahren:
    CPU schreibt das Datum nur in den Cache. Durch Setzen eines „dirty bit" wird markiert, dass die entsprechende Hauptspeicherzelle noch nicht beschrieben wurde. Die Hauptspeicherzelle selbst wird erst später aktualisiert, wenn die Kopie aus dem Cache verdrängt wird.

# Write-through versus Write-back

- **Vorteile von Write-back**:
  - Schreibzugriffe auf Cache ohne Wartezyklus möglich.
  - Belastung des Systembusses kleiner, wenn das Rückschreiben in den Hauptspeicher erst nach mehreren Schreibvorgängen erfolgen muss.
- **Nachteile von Write-back**:
  - Schwierigkeit bei der Datenkonsistenz, wenn andere Komponenten auf den Hauptspeicher zugreifen können, z.B. ein DMA-Controller oder ein zweiter Prozessor in einer Shared Memory Machine. Zu vermeiden ist, dass die anderen Komponenten veraltete Werte vorfinden und mit ihnen rechnen.

# Prinzipielle Funktionsweise: Schreibzugriff, Cache Miss

- Schreibe Datum in den Arbeitsspeicher unter Adresse *a*:
- Überprüfung durch CPU ergibt, dass Kopie der Hauptspeicherzelle *a* nicht im Cache abgelegt ist (Cache Miss).
  - Write-non-allocate Verfahren:
    CPU schreibt das Datum in die Hauptspeicherzelle mit Adresse *a*, Cache bleibt unverändert.
  - Write-allocate Verfahren:
    CPU schreibt das Datum in den Cache (evtl. unter Verdrängung eines anderen Cache–Eintrags).
    - (Bei Verwaltung ganzer „Cache Lines” muss restliche Cache Line aus Hauptspeicher in den Cache nachgeladen werden.)
    - In Kombination mit „write back”: Markiere Cache–Eintrag als „dirty”
    - In Kombination mit „write through”: Schreibe Cache–Eintrag in den Hauptspeicher

# In der Realität . . .

- Mischung aus assoziativem und Direct Mapped Cache.

→ Satzassoziativer Cache

  - Endstück der Adresse entspricht der Nummer eines „Satzes” von mehreren Cache–Einträgen (vgl. Direct Mapped Cache) (genauer: bei größeren Cache–Lines eigentlich Mittelstück ...).
  - Sätze werden assoziativ verwaltet (paralleler Adressvergleich innerhalb eines Satzes).
  - Weniger Hardwareaufwand für parallelen Vergleich als bei (voll-)assoziativem Cache, weniger Kollisionen als bei Direct Mapped Cache.

- Verschiedene Cache-Level bei Prozessoren (Cache-Hierarchie)
  - L1-Cache (4-256 kByte - nah am Prozessor)
  - L2-Cache (64-512 kByte)
  - L3-Cache (2-32 MByte - bei mehreren Kernen: Für alle Kerne gemeinsam)

# Zusammenfassung

- Einige weiterführende Architekturkonzepte tragen zur Beschleunigung heutiger Rechner bei.
- Die wesentlichen sind:
  - Pipelining
  - Parallelverarbeitung in Multiprozessoren / Multicores
  - Leistungsfähige Speicherhierarchie (Caches, ...)
- Grundlage für die Anwendung dieser Architekturkonzepte sind technologische Weiterentwicklungen, die die Integration von Milliarden an Transistoren auf einem Chip ermöglichen.