Kapitel 4: Speicher

Speicherorganisation

Caches

Hintergrundspeicher

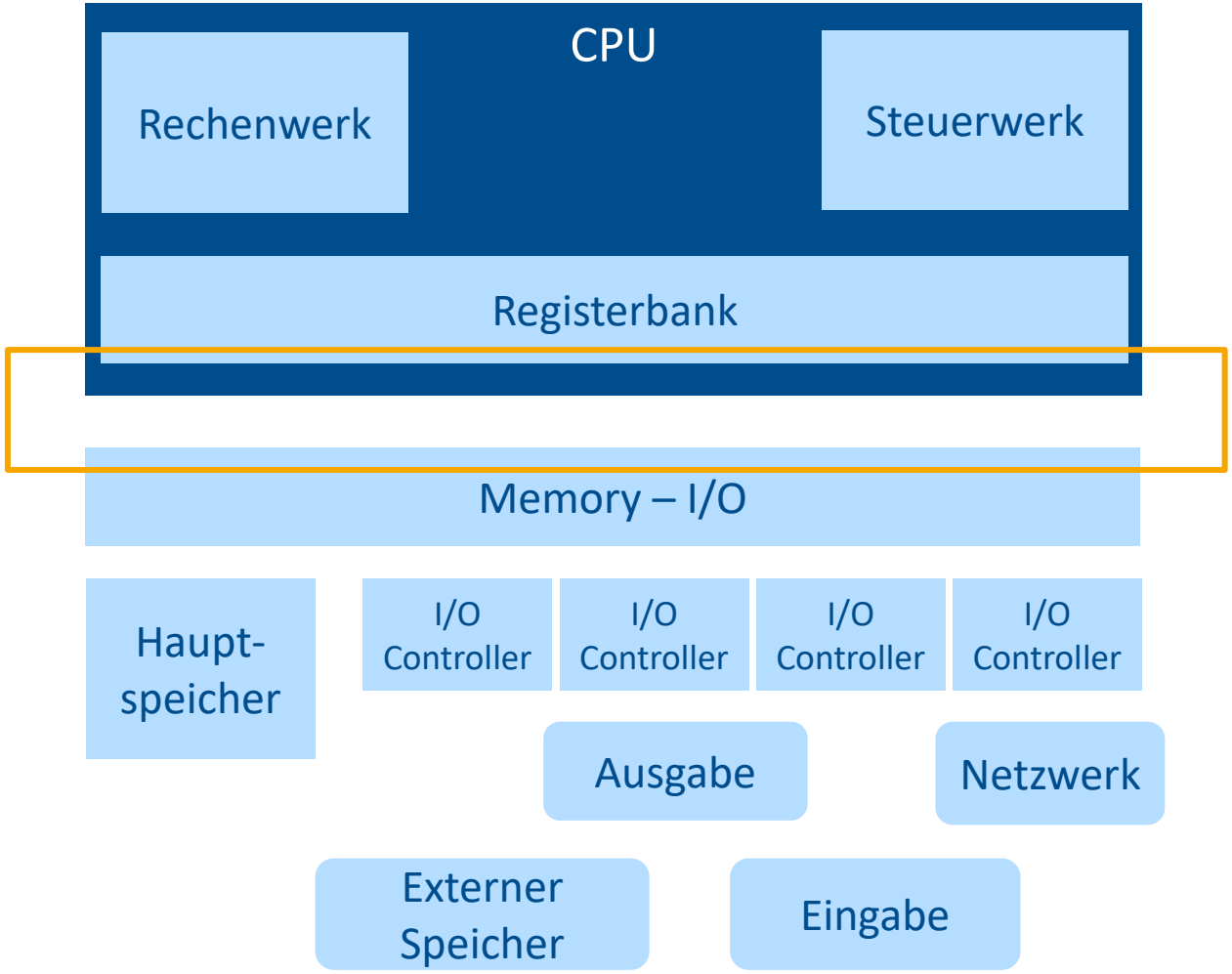
Lernziele

- Den Nutzen eines Caches verstehen und erklären können
- Den Lesezugriff eines Caches verstehen und erklären können
- Die Begriffe voll-assoziativer Cache und direct-mapped Cache unterscheiden und erklären können
- Die Verdrängungsstrategien FIFO, LFU, LRU kennen und anwenden können
- Die unterschiedlichen Formen der Schreibzugriffe kennen und ihre Vor- und Nachteile darlegen können

Caches

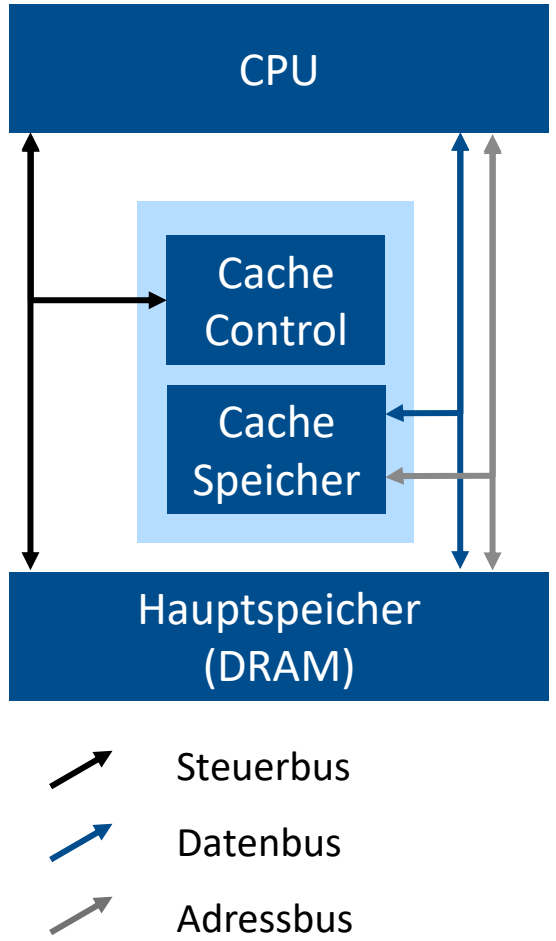
- Cache kommt aus dem Französischen: **cacher** (verstecken)
- Er kann durch ein Anwendungsprogramm nicht explizit adressiert werden
- Er ist **software-transparent**, d.h. der/die Benutzer*in braucht nichts von seiner Existenz zu wissen.

Lage des Caches



Lage und Aufbau des Caches

- „Zwischen“ CPU und Hauptspeicher
- Besteht aus
  - Cache Control
  - Cache Speicher (Adressspeicher/Datenspeicher)
- Bussysteme:
  - Steuerbus
  - Datenbus
  - Adressbus
- Getrennte Caches für
  - Instruktionen (Instruktionscache)
  - Daten (Datencache)



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

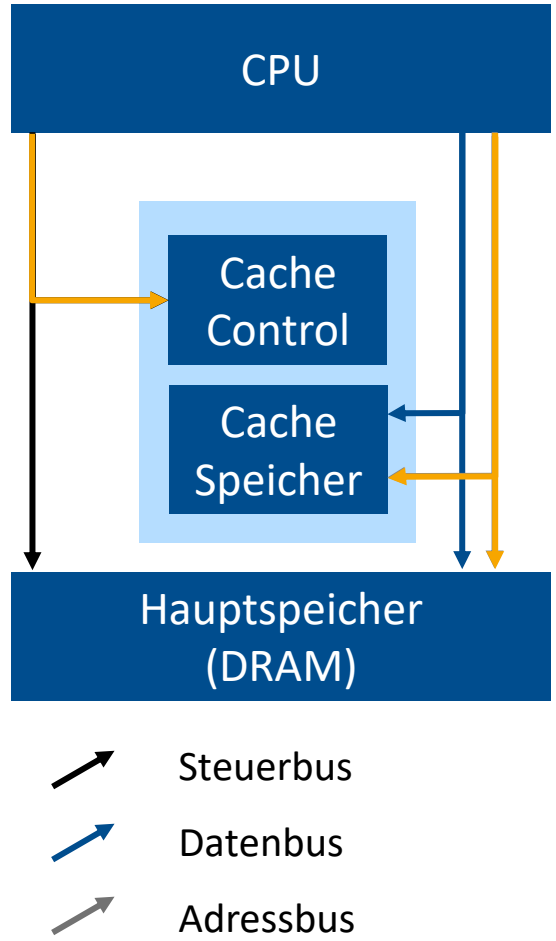
- Voraussetzung, um dieses Ziel erreichen zu können:

Lokalitätsprinzip

Prinzipielle Funktionsweise: Lesezugriff

Annahme: Lese Datum aus dem Hauptspeicher unter Adresse a

CPU überprüft, ob eine Kopie der Hauptspeicherzelle mit Adresse a im Cache abgelegt ist:

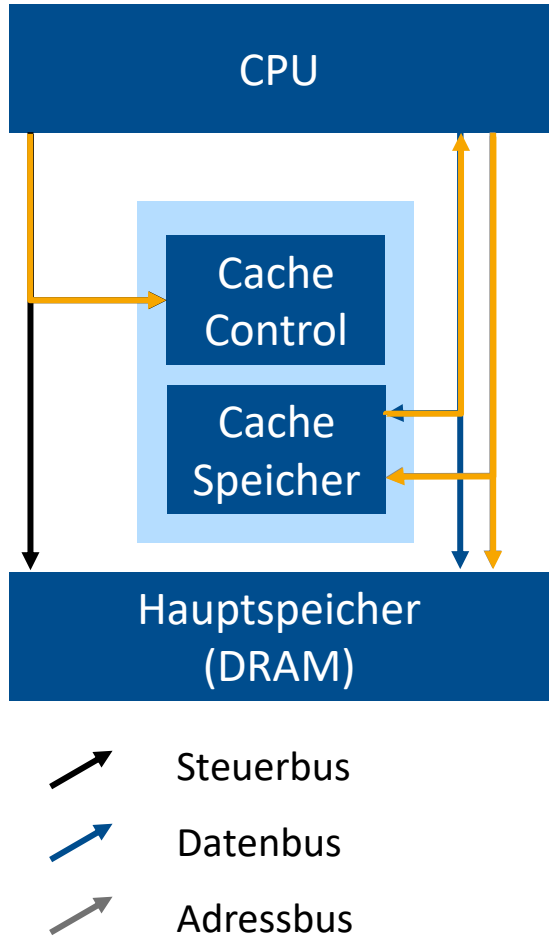


Prinzipielle Funktionsweise: Lesezugriff

Annahme: Lese Datum aus dem Hauptspeicher unter Adresse a

CPU überprüft, ob eine Kopie der Hauptspeicherzelle mit Adresse 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.



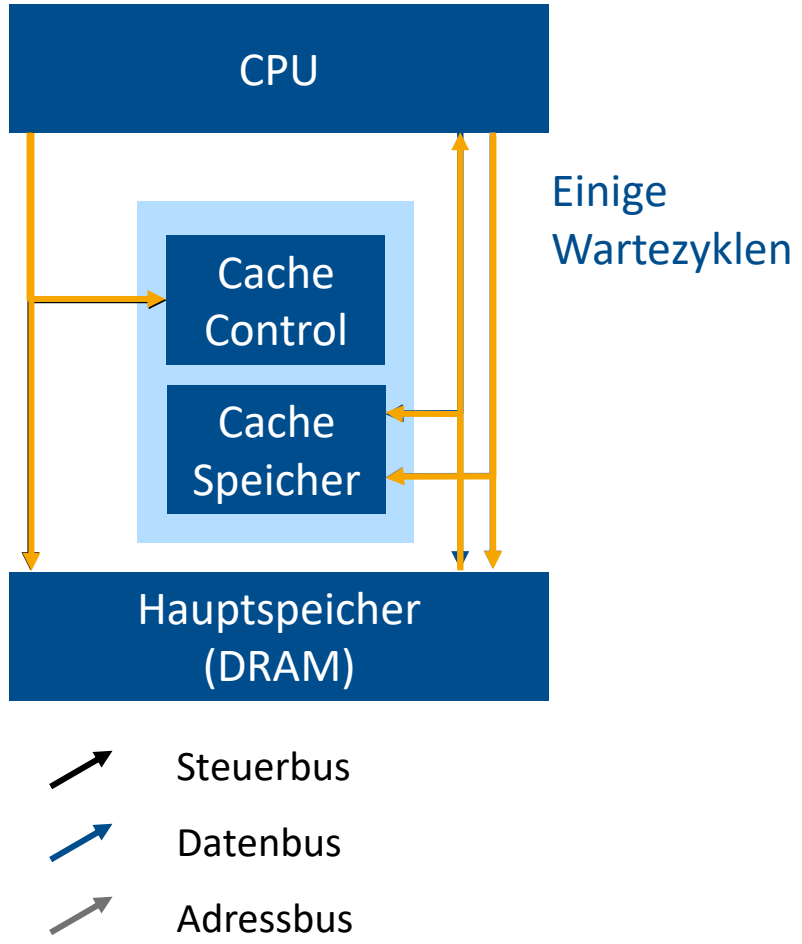
Prinzipielle Funktionsweise: Lesezugriff

Annahme: Lese Datum aus dem Hauptspeicher unter Adresse *a*

CPU überprüft, ob eine Kopie der Hauptspeicherzelle mit Adresse 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: Insbesondere bei Großrechnern wird mit jedem Datum auch dessen umgebender Block von Daten geladen in der Erwartung, dass folgende Zugriffe auf diese Daten erfolgen (wird hier aber nicht betrachtet).



Mittlere Zugriffszeit beim Lesen

- Einflussfaktoren zu beachten:
  - Zugriffszeit des Caches: c
  - Zugriffszeit beim Hauptspeicher: m
  - Trefferrate (cache hit): h
- Zugriffszeit bei Cache-hit: $t_{hit} = c$
- Zugriffszeit bei Cache-miss: $t_{miss} = c + m$
- Durchschnittliche Zugriffszeit: $t_{avg} = c + (1 - h) \cdot m$

Annahme: $c = 50\ ns$ und $m = 200\ ns$

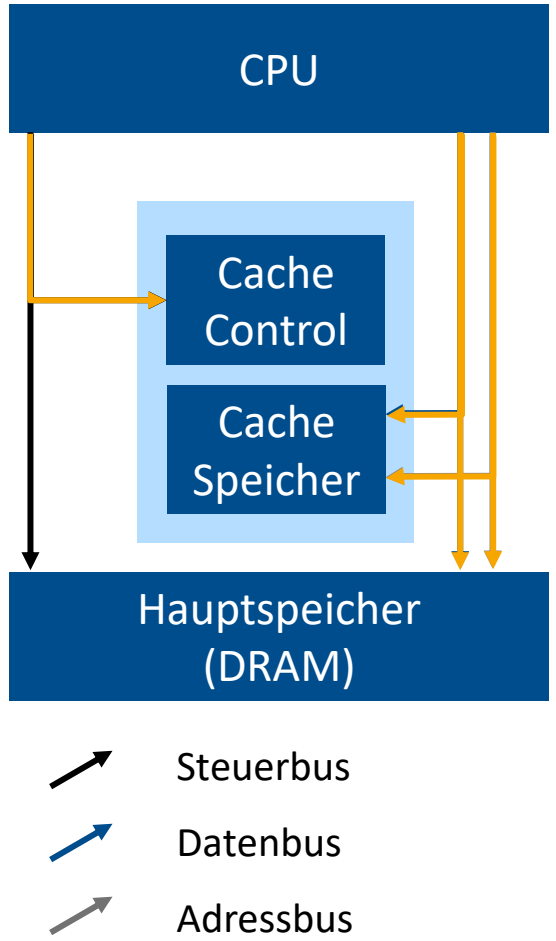
Trefferrate h	**Zugriffszeit** t_{avg}
50%	$150\ ns$
60%	$130\ ns$
70%	$110\ ns$
80%	$90\ ns$
90%	$70\ ns$
95%	$60\ ns$

Prinzipielle Funktionsweise: Schreibzugriff

Annahme: Schreibe Datum aus dem Hauptspeicher unter Adresse *a*

CPU überprüft, ob eine Kopie der Hauptspeicherzelle mit Adresse a im Cache abgelegt ist:

- write-through Verfahren
  - cache miss:
    - CPU schreibt Datum in Hauptspeicherzelle mit Adresse a
  - cache hit:
    - Hauptspeicherzelle wird aktualisiert
    - Kopie im Cache wird aktualisiert

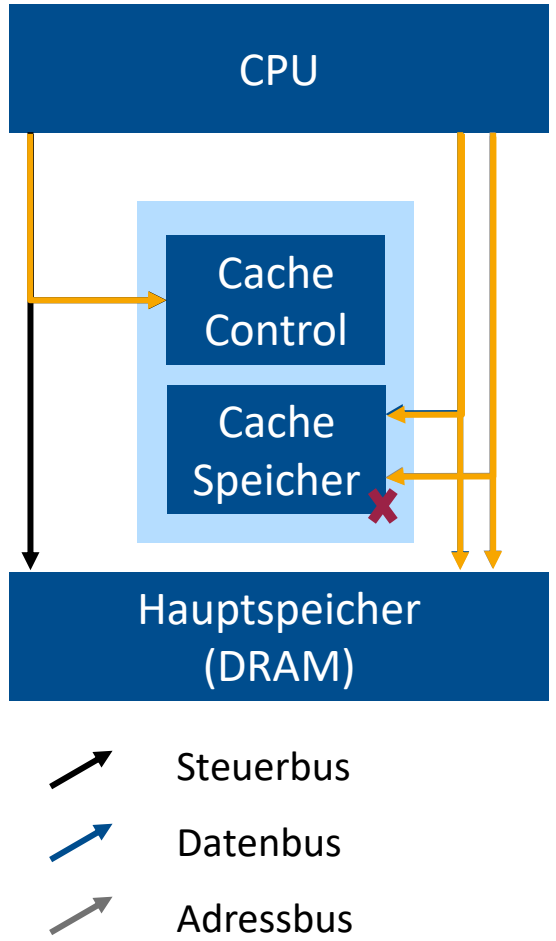


Prinzipielle Funktionsweise: Schreibzugriff

Annahme: Schreibe Datum aus dem Hauptspeicher unter Adresse *a*

CPU überprüft, ob eine Kopie der Hauptspeicherzelle mit Adresse a im Cache abgelegt ist:

- write-back Verfahren
  - cache miss:
    - CPU schreibt Datum in Hauptspeicherzelle mit Adresse a
  - cache hit:
    - Kopie im Cache wird aktualisiert und mit dirty bit markiert
    - Hauptspeicherzelle wird erst aktualisiert, wenn Kopie aus dem Cache verdrängt wird

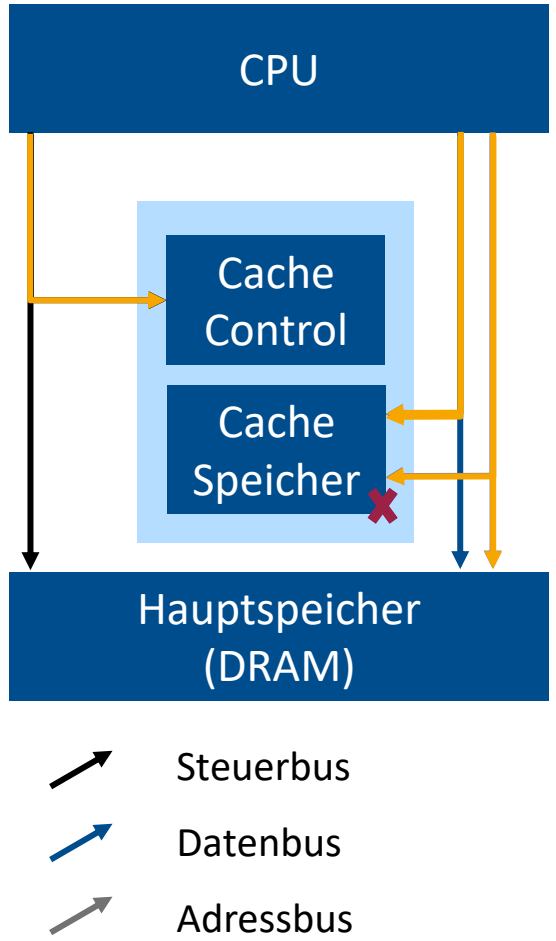


Prinzipielle Funktionsweise: Schreibzugriff

Annahme: Schreibe Datum aus dem Hauptspeicher unter Adresse *a*

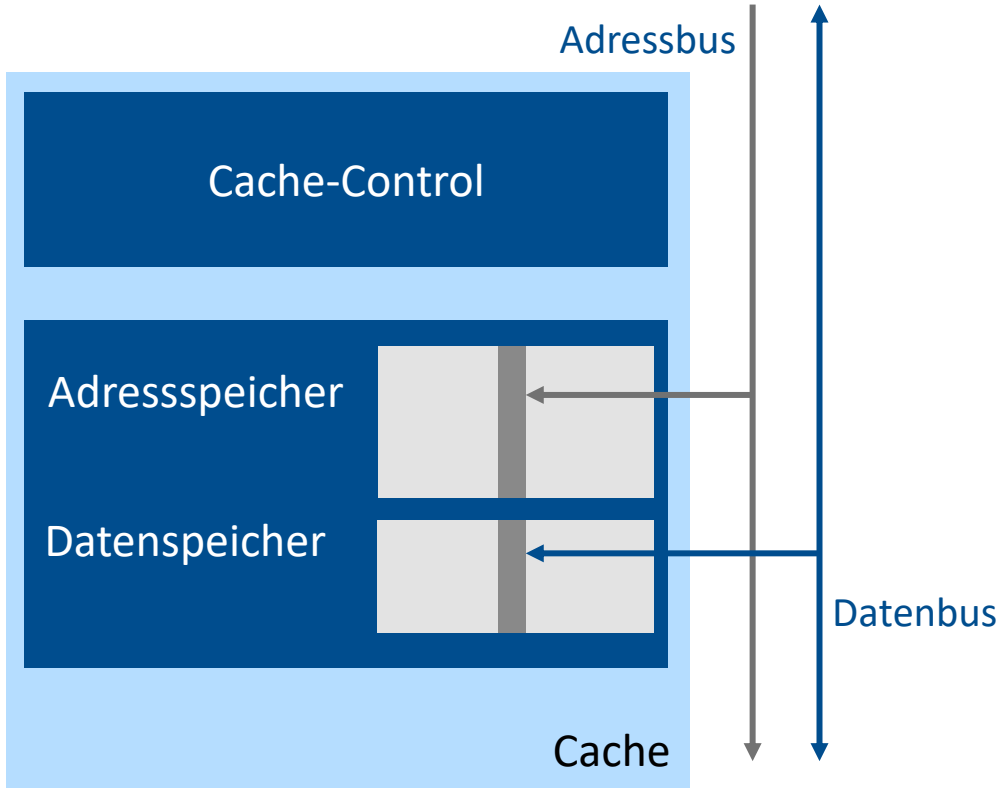
CPU überprüft, ob eine Kopie der Hauptspeicherzelle mit Adresse a im Cache abgelegt ist:

- write-allocation Verfahren
  - cache miss:
    - CPU schreibt Datum in Cache mit Adresse a markiert mit dirty bit
    - Hauptspeicher wird erst aktualisiert, wenn Kopie aus Cache verdrängt wird
  - cache hit:
    - Kopie im Cache wird aktualisiert und mit dirty bit markiert
    - Hauptspeicherzelle wird erst aktualisiert, wenn Kopie aus dem Cache verdrängt wird



Aufbau eines Caches

- Der Cache-Speicher besteht aus zwei Speicher-Einheiten, die wortweise einander fest zugeordnet sind
  - Adressspeicher
  - Datenspeicher
- Zwei Arten der Assoziation:
  - Voll-assoziativer Cache (VA)
  - Direct Mapped Cache (DMC)



Voll-assoziativer Cache

- Idealfall: assoziativer (inhaltsorientierter) Speicher
  - Paralleler Vergleich der von der CPU angelegten Adresse mit allen im Adressspeicher des Caches vorhandenen Adressen
  - Ablegen eines neuen Datums an jeder beliebigen freien Stelle im Cache
- Aber: Paralleler Vergleich erfordert aufwendige Logik

➡ Assoziative Speicher nur für kleine Cache-Größen

Voll-assoziativer Cache - Verdrängungsstrategien

Szenario

- cache miss
- alle Speicherbereiche des Caches belegt

Ausweg

- verdränge Datum (Block) aus dem Cache
- lade an seine Stelle das gerade benötigte Datum (Block)

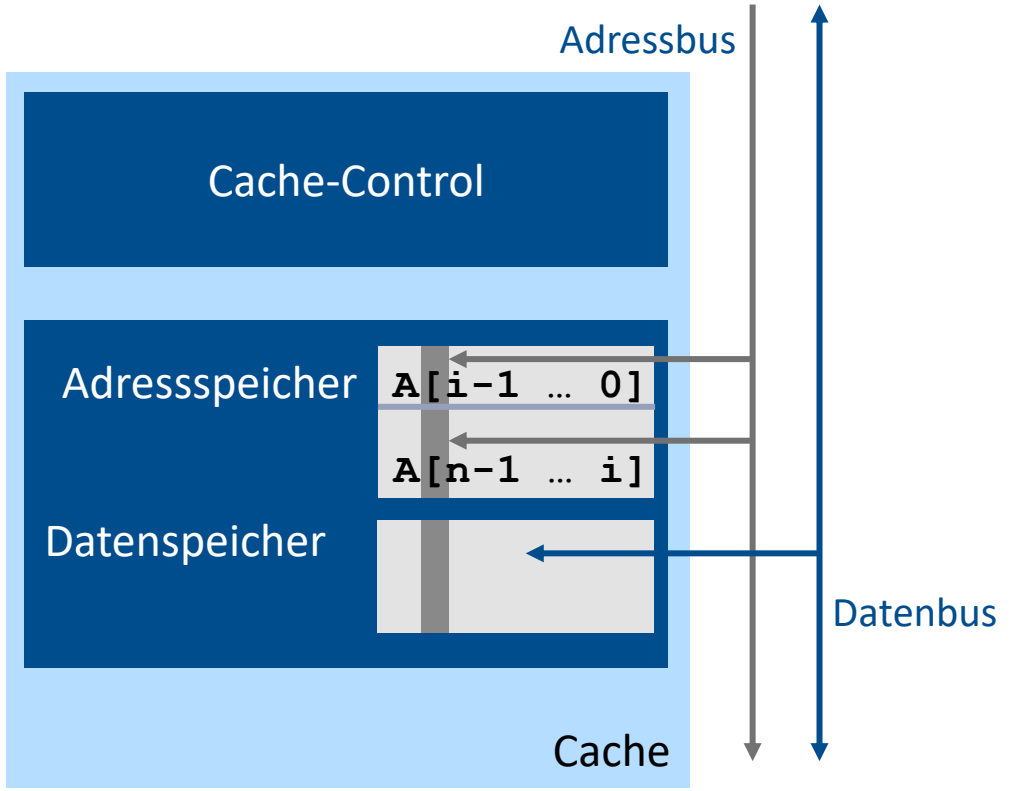
Gebräuchliche Verdrängungsstrategien

- 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

verlangt einen mehr oder weniger großen Aufwand

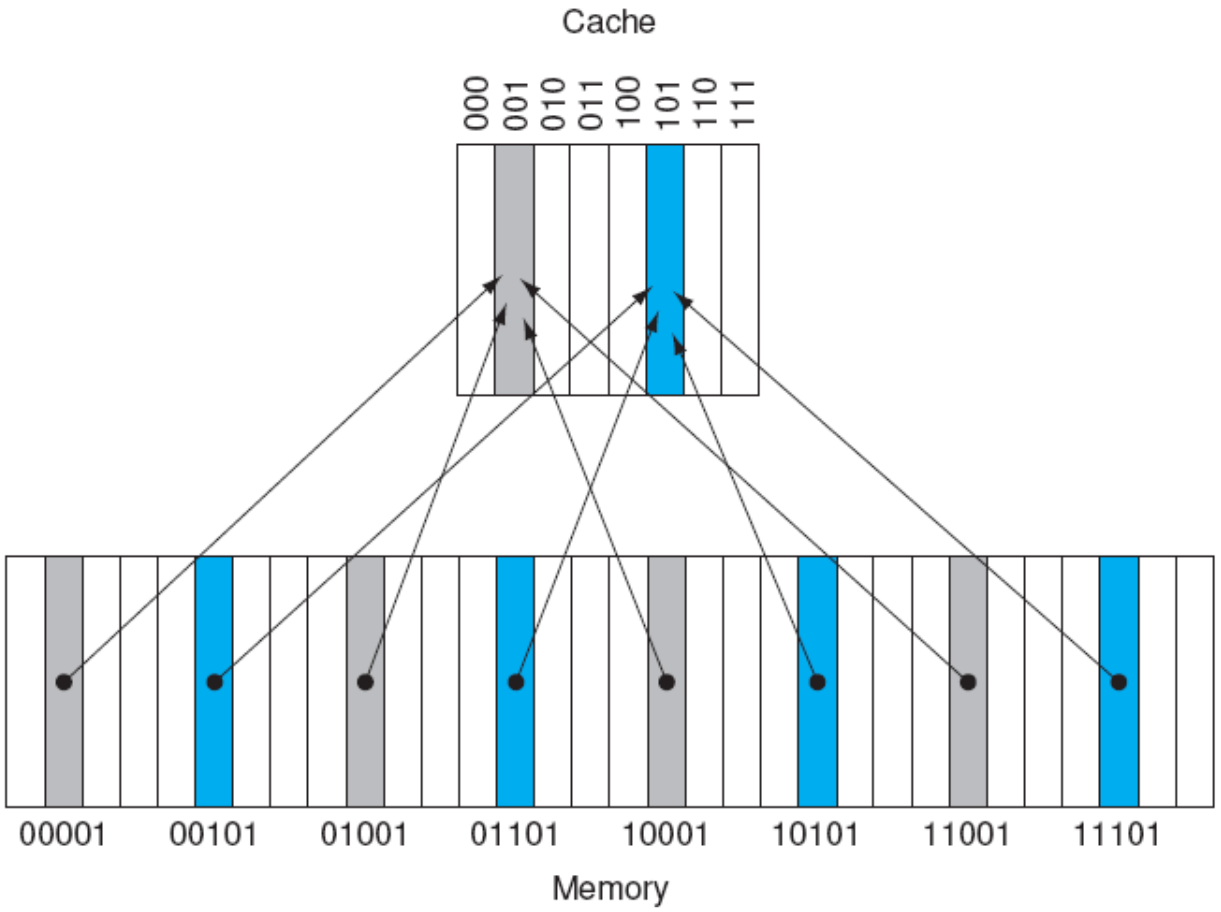
Direct Mapped Cache - Funktionsprinzip

- Feste Abbildung der Hauptspeicher-Adressen auf die Cache-Adressen
  - Kein assoziativer Speicher nötig
  - Keine Verdrängungsstrategien nötig
- Von der CPU angelegte Adresse `A[n-1 … 0]` wird in zwei Teile gespalten
  - `A[i-1 … 0]`: i niederwertige Bits dekodieren zu Eintrag im Adressspeicher
  - `A[n-1 … i]`: $n - i$ höherwertige Bits werden zum Vergleich mit Eintrag im Adressspeicher genutzt



Speicherzellen des Hauptspeichers, deren i niederwertige Bits gleich sind, werden auf dieselbe Position im Cache abgebildet.

Direct Mapped Cache - Illustration



Direct Mapped Cache

- Cache Tag
  - Keine eindeutige Zuordnung des Index (Cacheblockadresse) zu einer Speicherblockadresse
  - Inhalt des Caches an dieser Position somit nicht eindeutig
  - Verwendung der restlichen Bits der Speicherblockadresse als Tag (d.h. zu jedem Block im Cache wird auch der Tag gespeichert)
  - Auffinden von Daten durch Abgleich der Tags des gesuchten Blocks und des Cacheeintrages

- Valid Bit
  - Gibt an, ob ein Cacheeintrag gültig ist oder nicht
    - Am Anfang – und nach jedem Leeren des Caches – enthält der Cache keine gültigen Daten (Valid Bit wird auf 0 gesetzt)
    - Wenn der Cacheeintrag gültig ist, wird das Valid Bit auf 1 gesetzt

Direct Mapped Cache – Ein Beispiel

- Annahmen:
  - 5-Bit Adressen,
  - Cache mit direkter Abbildung,
  - 8 Cacheblöcke,
  - ein Block besteht aus einem Wort

  Also: 32 Speicherblöcke

Zeit	Speicheradresse	Hit / Miss	Cacheblock
(1)	$22_{10} = 10110$		
(2)	$26_{10} = 11010$		
(3)	$22_{10} = 10110$		
(4)	$26_{10} = 11010$		
(5)	$16_{10} = 10000$		
(6)	$3_{10} = 00011$		
(7)	$16_{10} = 10000$		
(8)	$18_{10} = 10010$		

vor (1)

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

nach (1)

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

nach (2)

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

nach (3),(4)

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

nach (5)

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

nach (6)

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

nach (7)

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

nach (8)

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	10	Mem[10010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct Mapped Cache – Ein Beispiel

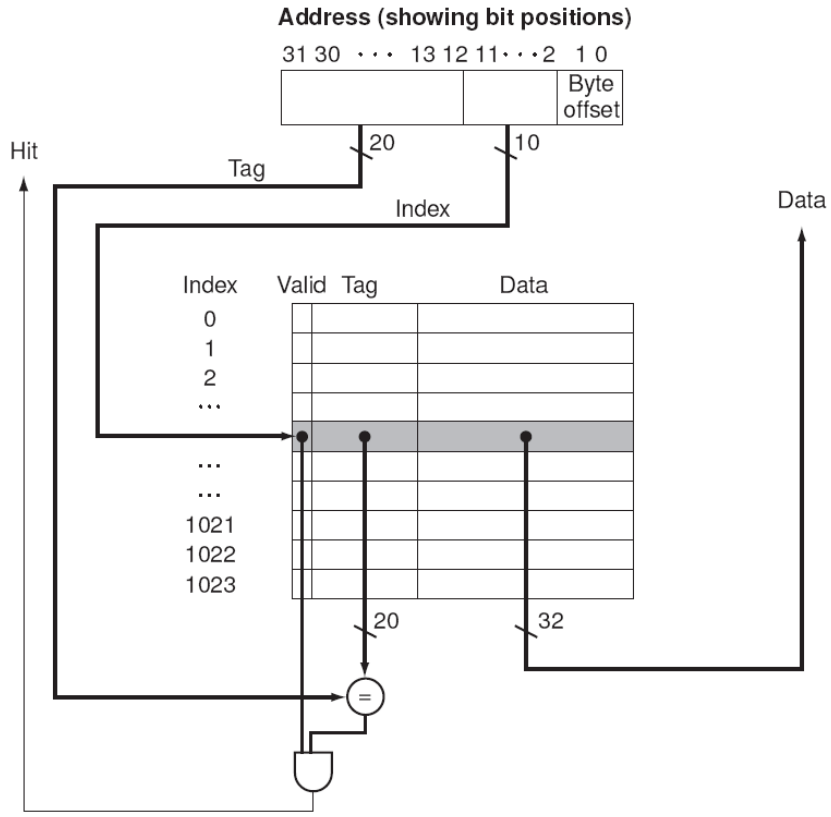
- Annahmen:
  - 5-Bit Adressen,
  - Cache mit direkter Abbildung,
  - 8 Cacheblöcke,
  - ein Block besteht aus einem Wort

  Also: 32 Speicherblöcke

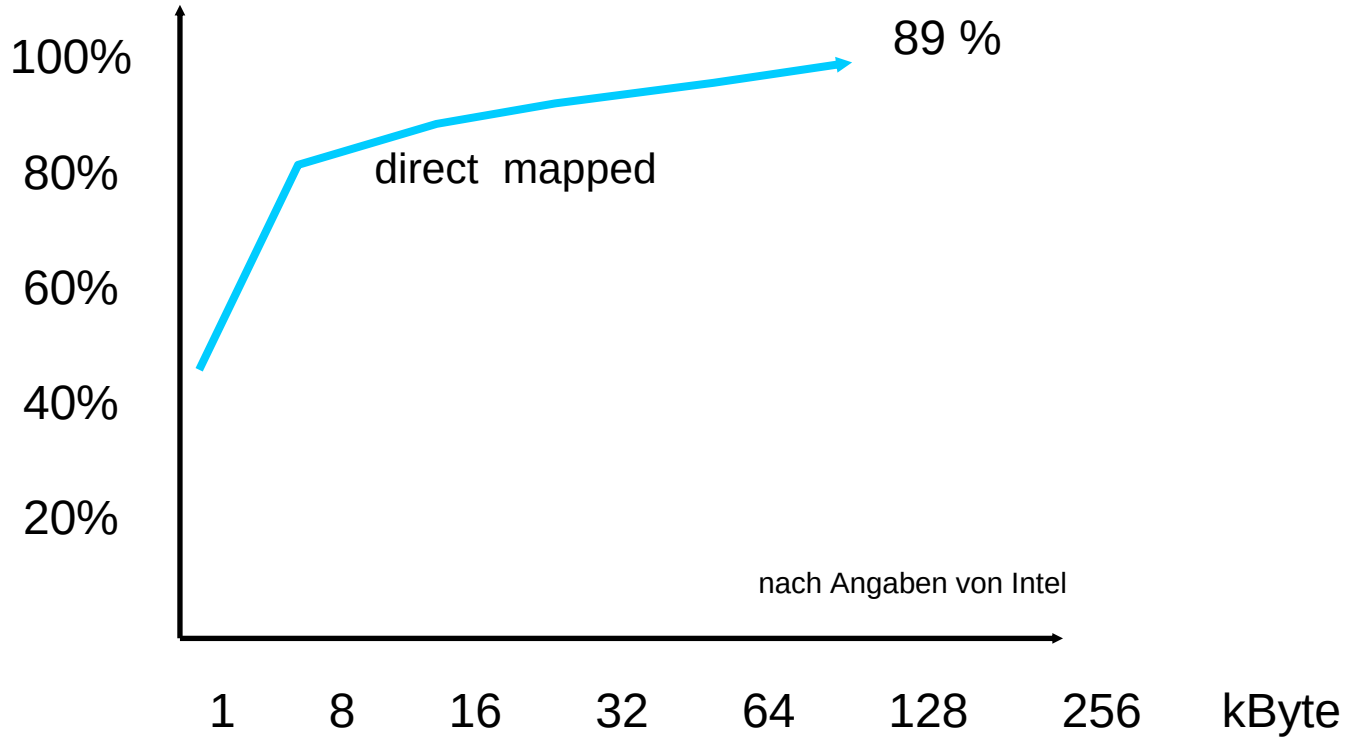
Zeit	Speicheradresse	Hit / Miss	Cacheblock
(1)	$22_{10} = 10110$	Miss	$10110\ mod\ 8 = 110$
(2)	$26_{10} = 11010$	Miss	$11010\ mod\ 8 = 010$
(3)	$22_{10} = 10110$	Hit	$10110\ mod\ 8 = 110$
(4)	$26_{10} = 11010$	Hit	$11010\ mod\ 8 = 010$
(5)	$16_{10} = 10000$	Miss	$10000\ mod\ 8 = 000$
(6)	$3_{10} = 00011$	Miss	$00011\ mod\ 8 = 011$
(7)	$16_{10} = 10000$	Hit	$10000\ mod\ 8 = 000$
(8)	$18_{10} = 10010$	miss	$10010\ mod\ 8 = 010$

Direct Mapped Cache - Prüfungsablauf

- Index: Bestimmt zu überprüfende Cachezelle
- Tag: Wird verglichen mit eingetragenem Tag im Cache
- Valid: Nur wenn das Valid Bit gesetzt ist, wird ein Hit zurückgegeben

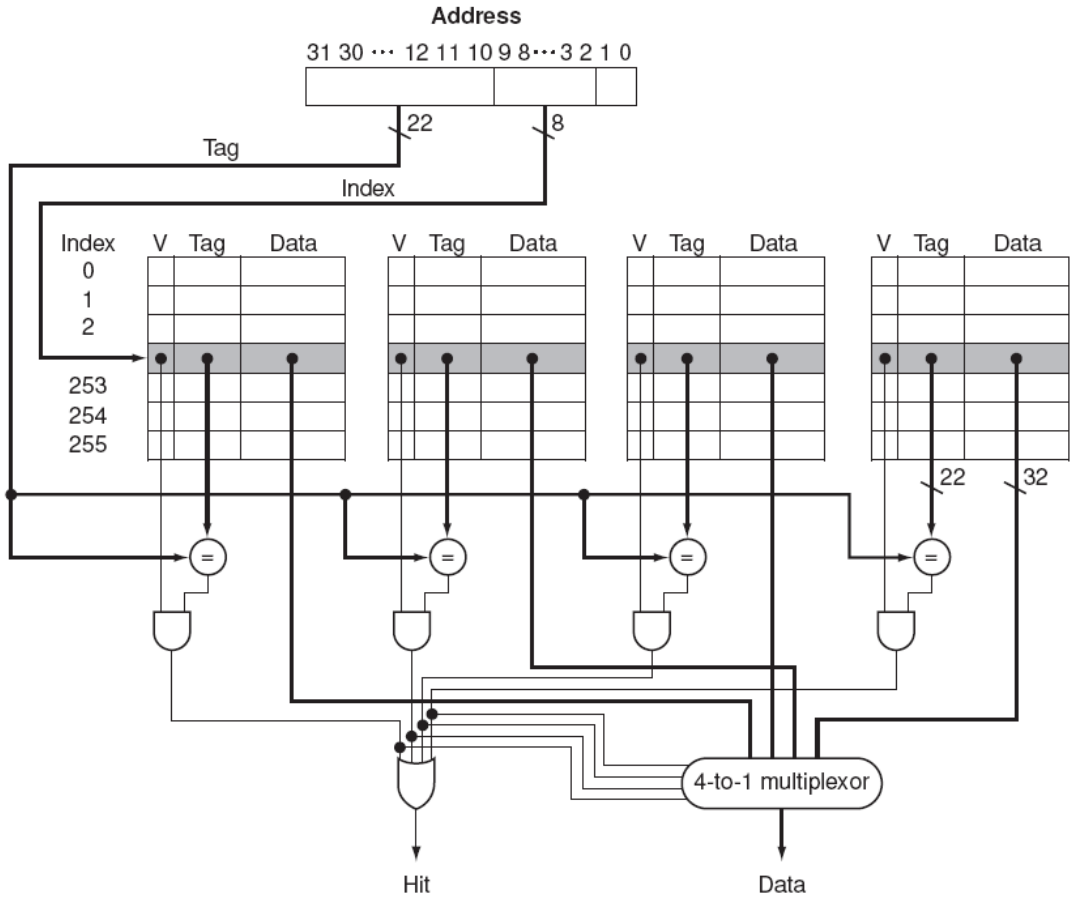


Direct Mapped Cache: Trefferquote



Direct Mapped Cache UND Assoziativer Cache

- Kombination mehrerer DMCs mit parallelem Vergleich
  - Index wird in allen DMCs gewählt
  - Tag wird mit allen Kandidaten parallel verglichen (wie im VA-Cache)
- Vorteile von DMC und VA genutzt



Write-through versus Write-back/Write-alloaction

Vorteile von Write-back/Write-allocation

- Schreibzugriffe auf Cache bei cache hit ohne Wartezyklus möglich (bei write-allocation sogar bei cache miss)
- Belastung des Systembusses kleiner, wenn das Rückschreiben in den Hauptspeicher erst nach mehreren Schreibvorgängen erfolgen muss

Nachteile von Write-back/Write-allocation

- Datenkonsistenz nicht gesichert bei Zugriff anderer Komponenten auf Hauptspeicher, z.B.
  - DMA – Controller
  - Zweiter Prozessor
- Vermeide das Vorfinden alter Werte

Speicherorganisation