Universität Bamberg

GRABS: Einführung in Rechnersysteme und Betriebssysteme

Vorlesung 5: Speicher

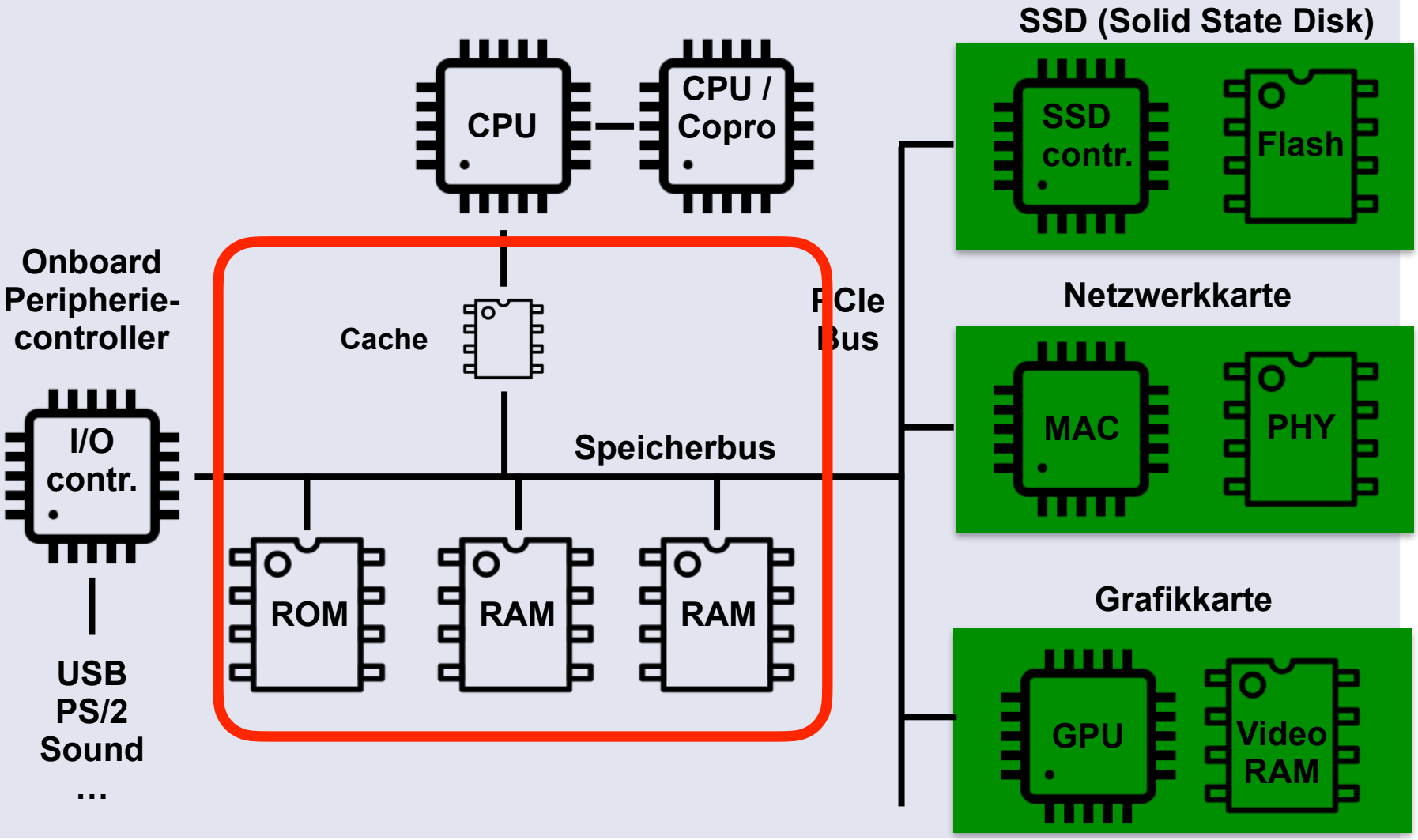
Literatur:
Slomka [1]
Kap. 4.3
H/P [2]
Kap. 2

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Aufbau eines Computersystems

Universität Bamberg

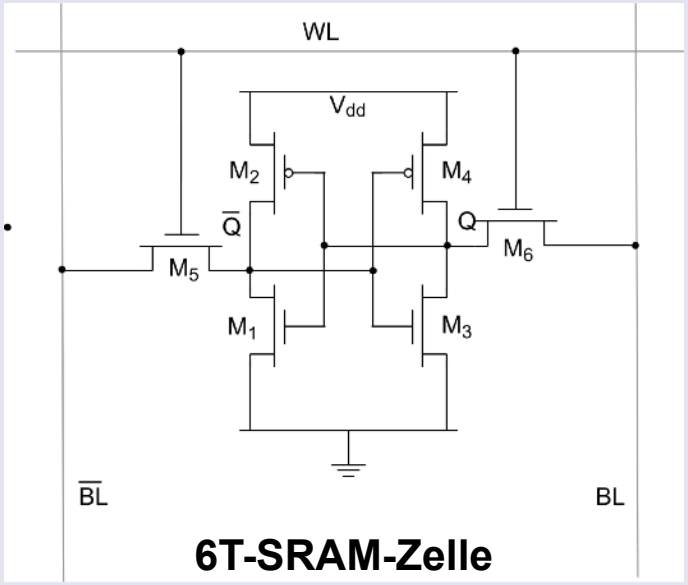
(aus Vorlesung 2)



Halbleiterspeicher: SRAM

Wir kennen **Flipflops** und **Register**

Flipflops und daraus konstruierte Register **behalten ihren Wert, solange die Versorgungsspannung anliegt**:



6T-SRAM-Zelle
(CC BY-SA 3.0 Abelsson)

Statischer Schreib-/Lesespeicher (static random access memory): **SRAM**

- Schnell, energiesparend, aber teuer: 6 Transistoren pro Bit
- Verwendet für Prozessorregister, Caches, Mikrocontroller

Alternative: Dynamisches RAM: **DRAM**

Halbleiterspeicher: DRAM

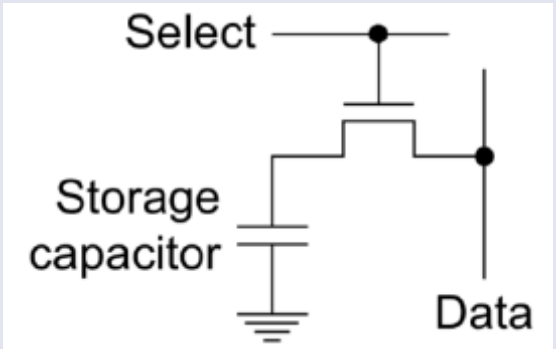
Dynamische Schreib-/Lesespeicher
(dynamic random access memory): **DRAM**

- Speichert Information mit einem **Kondensator** pro Bit

- **Problem:**
  Kondensatoren **entladen** sich mit der Zeit ⇒ **Information geht verloren**!
  - ***Periodischer Refresh notwendig!***

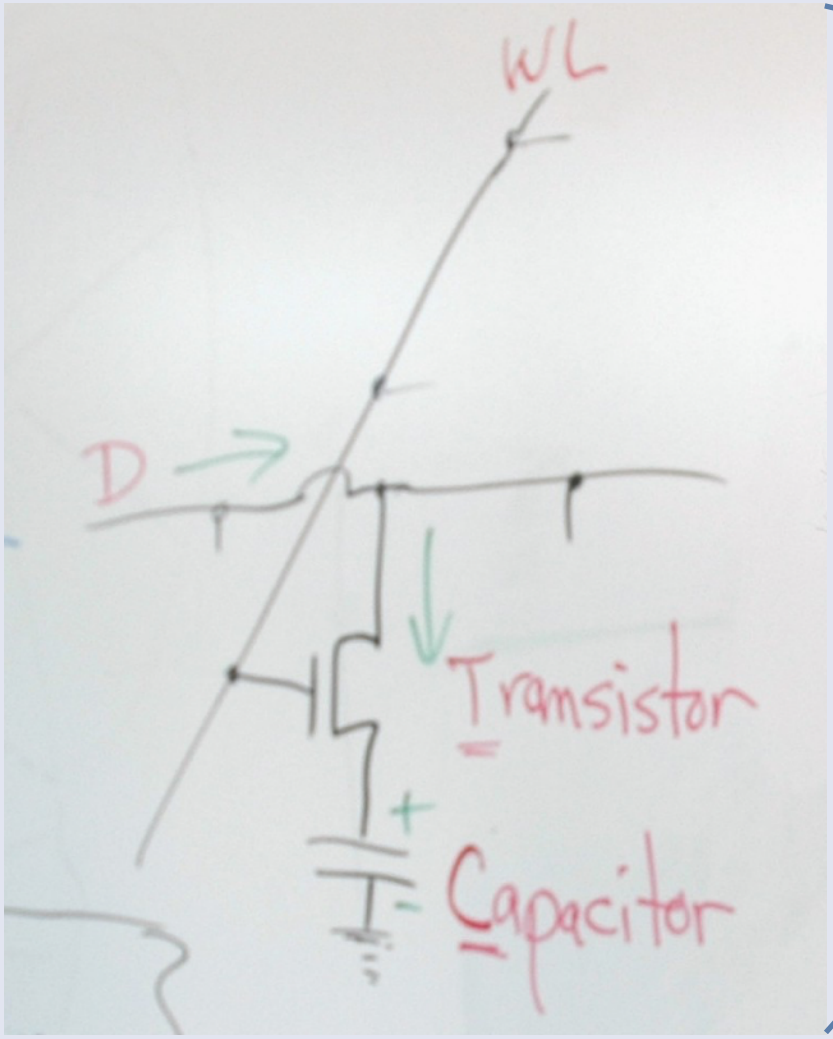
- Langsam, energieaufwendig, aber kostengünstig

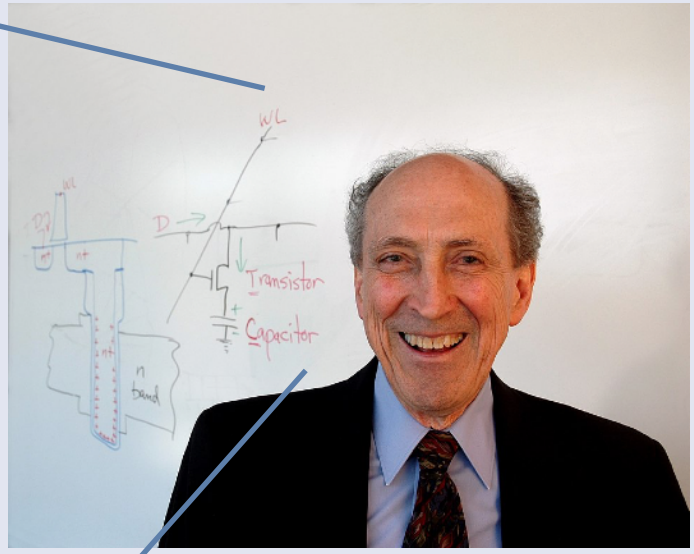
- Verwendet für Hauptspeicher, Video-RAM



DRAM-Bitzelle

DRAM





Robert H. Dennard (1932-2024)

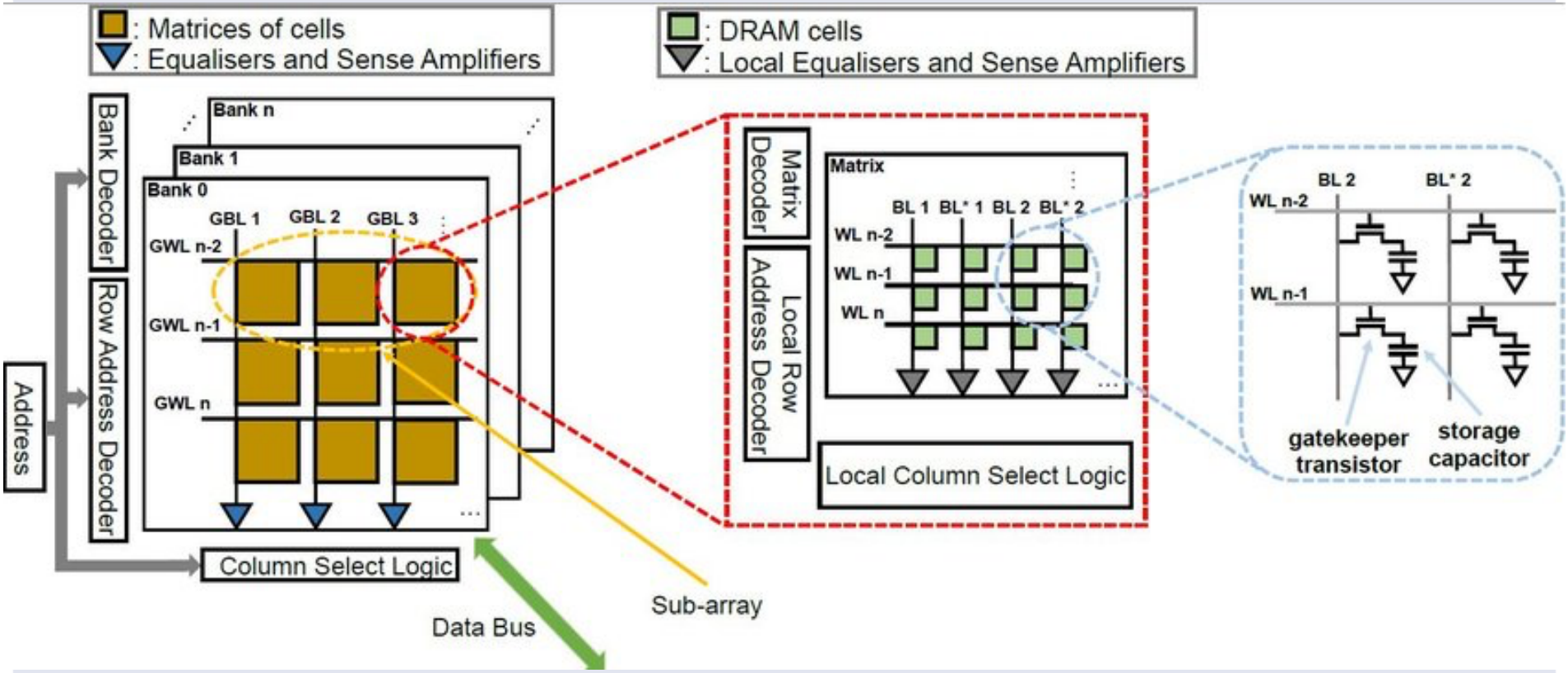
Erfinder des DRAM und
Entdecker von Dennard's Law

Bild: CC BY-SA 3.0 Fred Holland

Halbleiterspeicher: Adressierung

Universität Bamberg

DRAMs erhalten Zeilen- und Spaltenadressen separat **auf den selben Leitungen**: Row/Column-Select zusätzlich



Aus [3] von Fatemeh Tehranipoor

Entwurfsziele für Speicher

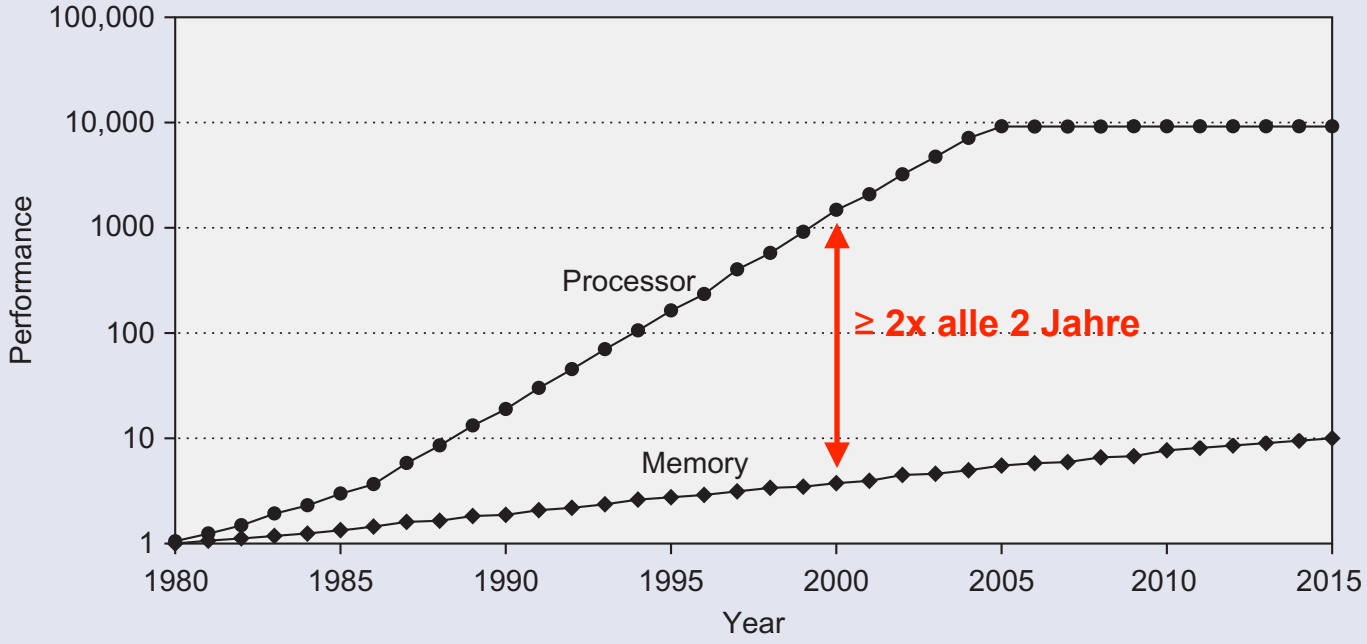
- Möglichst großer Speicher
- Möglichst große Geschwindigkeit
  - Geringe Zeit bis zur Verfügbarkeit eines Speicherinhalts (kleine ***Latenz***)
  - Hoher ***Durchsatz***
- Persistente (nicht-flüchtige) Speicherung
- Geringe Energieaufnahme
- Hohe Datensicherheit
- Geringer Preis
- Klein

Alle Anforderungen gleichzeitig zu erfüllen ist unmöglich!

Problem: Performance-Lücke

Seit ca. 1980 entwickeln sich die Prozessorgeschwindigkeiten und die Speicherzugriffsgeschwindigkeiten (für einzelne Cores) immer weiter auseinander

- Diese Performance-Lücke erforderte ein Umdenken in der Anbindung von großen Speichern



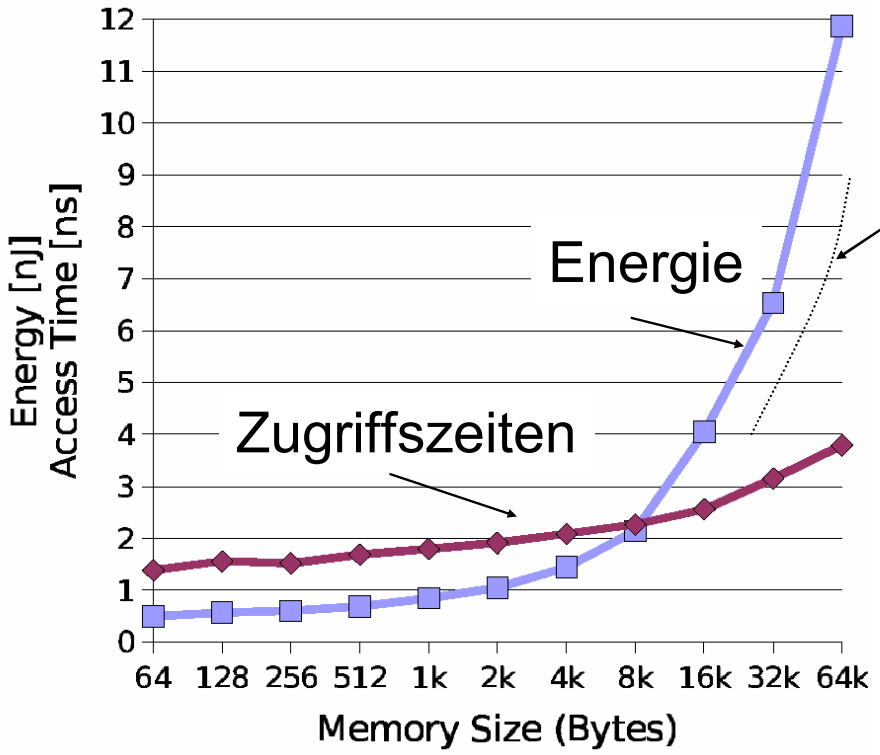
Prozessoren: ~1,5–2x schneller pro Jahr **bis ~2005** (siehe auch das Paper vom 2. Mai zu Moore's Law)

DRAM: ~1,07x schneller pro Jahr

Abbildung aus [2]

Zugriffsgeschwindigkeit vs. Speichergröße

Mit zunehmender Größe von Speichern steigen Zugriffszeiten und Energieaufnahme



Anwendungen werden größer und größer …

Quelle: CACTI

…schon bei den ersten Computern

*Ideally one would desire an **indefinitely large memory** capacity such that **any** particular... **word would be immediately available**... We are... forced to recognize the possibility of **constructing a hierarchy of memories** each of which has greater capacity than the preceding but which is less quickly accessible.*

A. W. Burks, H. H. Goldstine, and J. von Neumann, Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (**1946**).

Speicherhierarchie

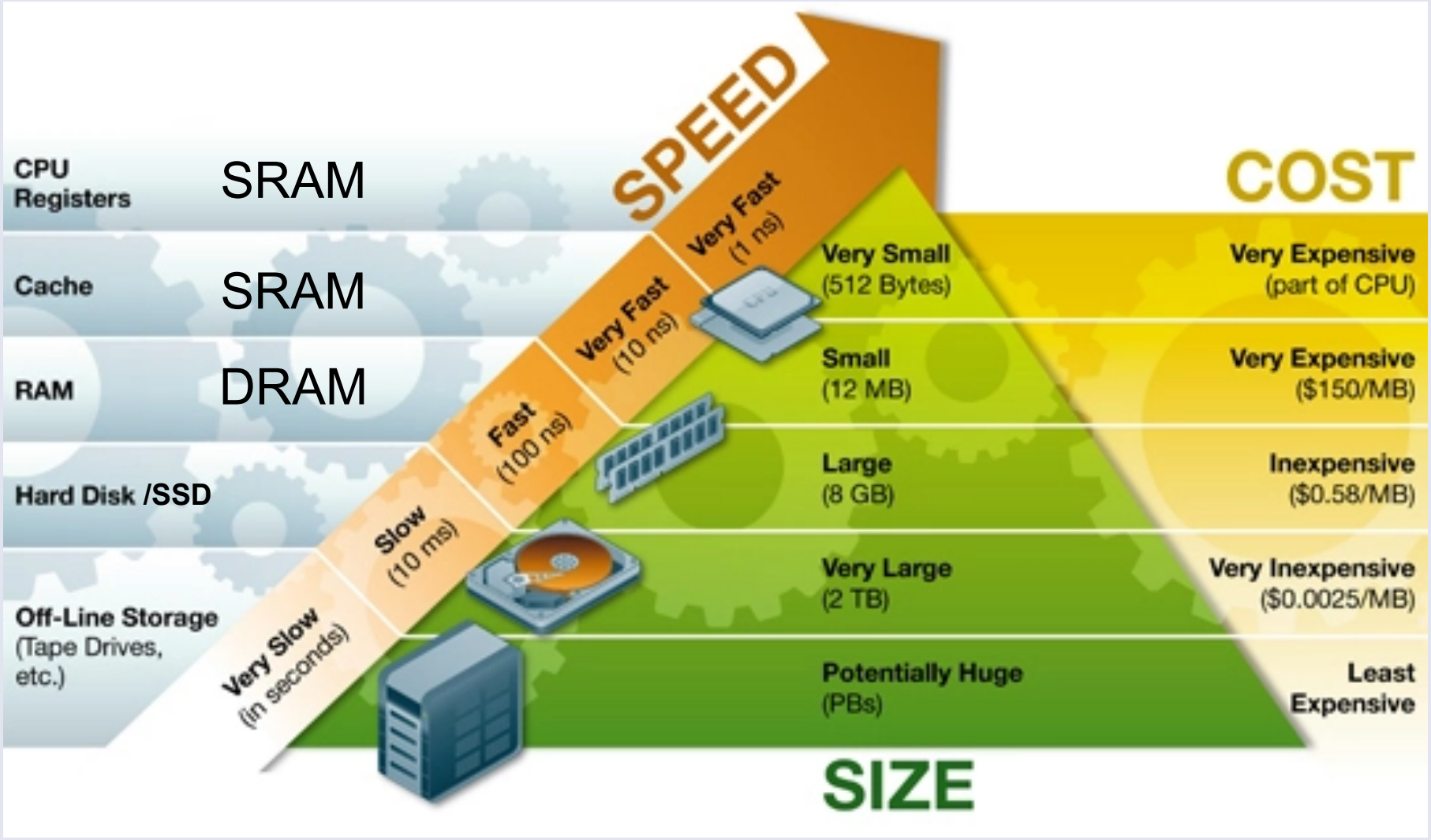
- Große Speicher sind langsam
- Anwendungen verhalten sich üblicherweise lokal

  ⇒ Wir versuchen, häufig benötigte Speicherinhalte in kleinen Speichern, seltener benötigte Inhalte in großen Speichern abzulegen

  ⇒ Einführung einer „**Speicherhierarchie**“

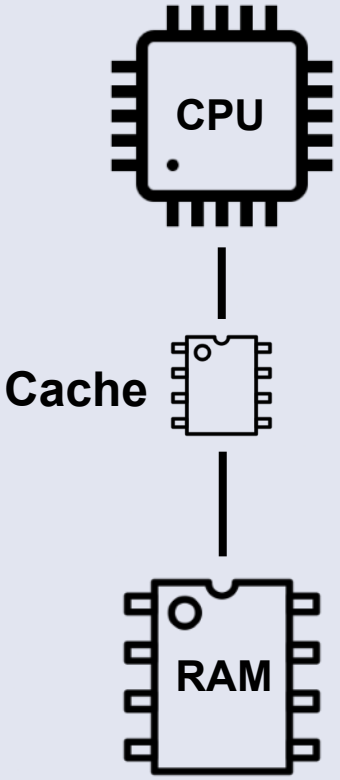
- Man möchte durch diese Kombination die Illusion eines großen Speichers mit kleinen Zugriffszeiten erzielen, zumindest im Mittel
- Eine enge Grenze für die Zugriffszeit ist vielfach kaum zu garantieren
- Bei Realzeitsystemen ergeben sich spezielle Überlegungen

Speicherhierarchie



Caches

- ***Cache*** = Speicher, der vor einen größeren, langsamen Speicher geschaltet wird
  - von frz. "cacher" = "verstecken"
- Im weiteren Sinn:
  Puffer zur Aufnahme häufig benötigter Daten
- Im engeren Sinn:
  Puffer zwischen Hauptspeicher und Prozessor
  - Inhalte von Caches transparent in Hardware verwaltet ("unsichtbar" für Software)*
  - Caches für Code und für Daten (gemeinsam oder getrennt)



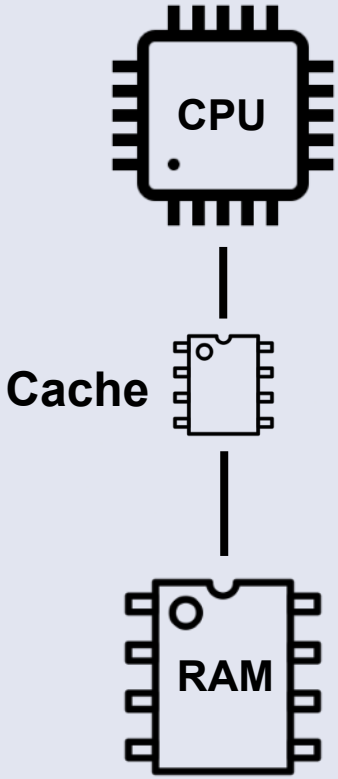
- Größe von Caches (i.A.) << Größe des Hauptspeichers

** soweit die Theorie, später mehr dazu…*

Warum funktionieren Caches?

- Sowohl die Ausführung von Programmcode wie auch Daten weisen ***Lokalität*** auf

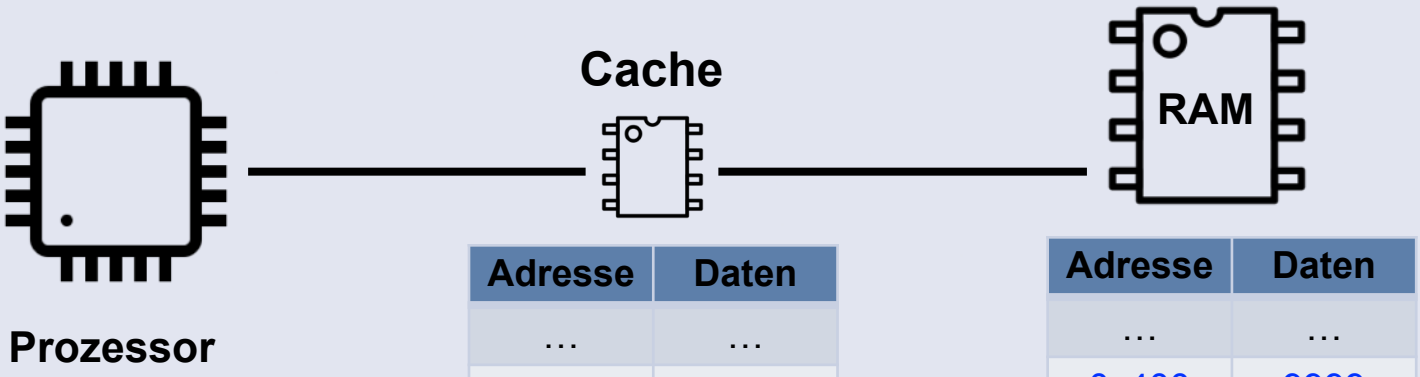
- **Zeitliche Lokalität**
  - Adressbereiche, auf die zugegriffen wird, werden auch in naher Zukunft mit hoher Wahrscheinlichkeit wieder benutzt werden
  - Im zeitlichen Verlauf erfolgt also relativ häufig der Zugriff auf eine gleiche Speicheradresse
  - Caches speichern solche bereits verwendeten Werte
  - Beispiel: Programmschleifen
- **Räumliche Lokalität**
  - Nach einem Zugriff auf einen Adressbereich folgt mit hoher Wahrscheinlichkeit der nächste Zugriff auf eine Adresse in unmittelbarer Nachbarschaft
  - Caches laden nicht nur einen angefragten Wert, sondern auch weitere Werte aus der "Nachbarschaft"
  - Beispiel: sequentielle Programmausführung, Arrays



Funktionsweise von Caches

Organisation von Caches (im engeren Sinn):

- Prüfung anhand der Speicheradresse, ob benötigte Daten im Cache vorhanden sind („Treffer“; ***cache hit***)
- Falls nicht (***cache miss***):
  - Zugriff auf den Hauptspeicher zum Lesen der Daten
  - Speichern der Daten in einem Cacheeintrag



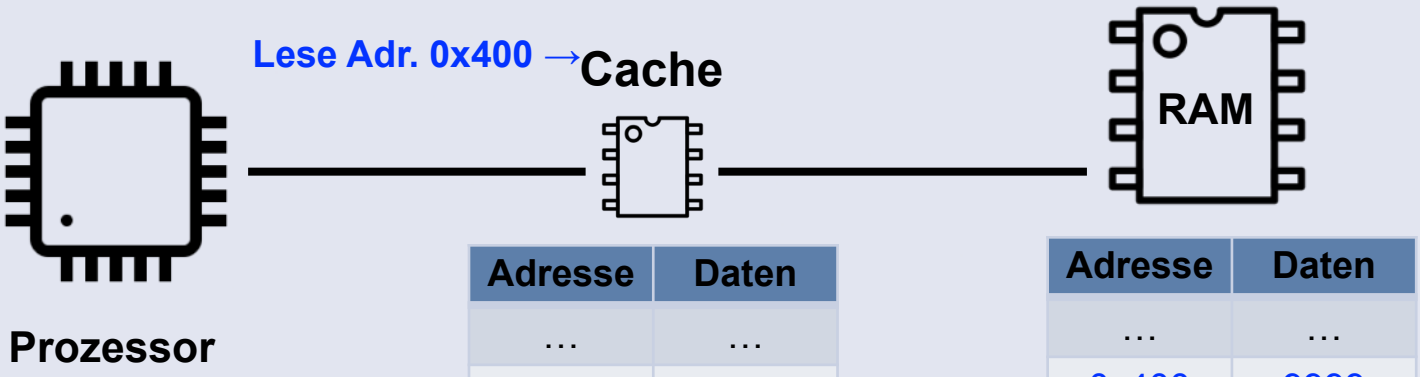
Adresse	Daten
…	…
0x400	9999
…	…
…	…

Adresse	Daten
…	…
0x400	9999
…	…
0x600	5555

Funktionsweise von Caches

Organisation von Caches (im engeren Sinn):

- Prüfung anhand der Speicheradresse, ob benötigte Daten im Cache vorhanden sind („Treffer“; ***cache hit***)
- Falls nicht (***cache miss***):
  - Zugriff auf den Hauptspeicher zum Lesen der Daten
  - Speichern der Daten in einem Cacheeintrag



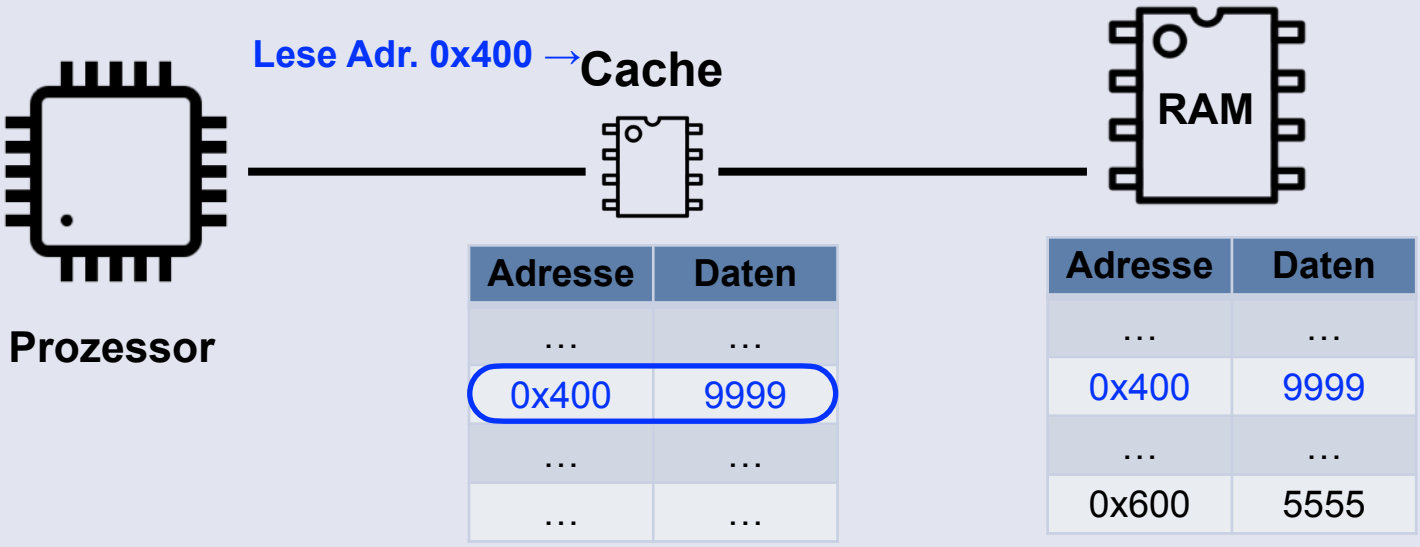
Adresse	Daten
…	…
0x400	9999
…	…
…	…

Adresse	Daten
…	…
0x400	9999
…	…
0x600	5555

Funktionsweise von Caches

Organisation von Caches (im engeren Sinn):

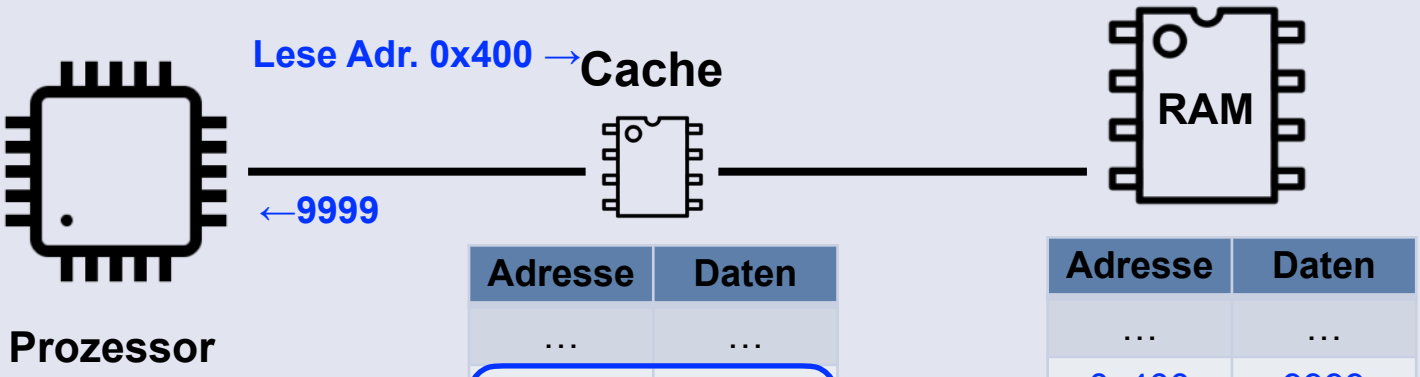
- Prüfung anhand der Speicheradresse, ob benötigte Daten im Cache vorhanden sind („Treffer“; ***cache hit***)
- Falls nicht (***cache miss***):
  - Zugriff auf den Hauptspeicher zum Lesen der Daten
  - Speichern der Daten in einem Cacheeintrag



Funktionsweise von Caches

Organisation von Caches (im engeren Sinn):

- Prüfung anhand der Speicheradresse, ob benötigte Daten im Cache vorhanden sind („Treffer“; ***cache hit***)
- Falls nicht (***cache miss***):
  - Zugriff auf den Hauptspeicher zum Lesen der Daten
  - Speichern der Daten in einem Cacheeintrag



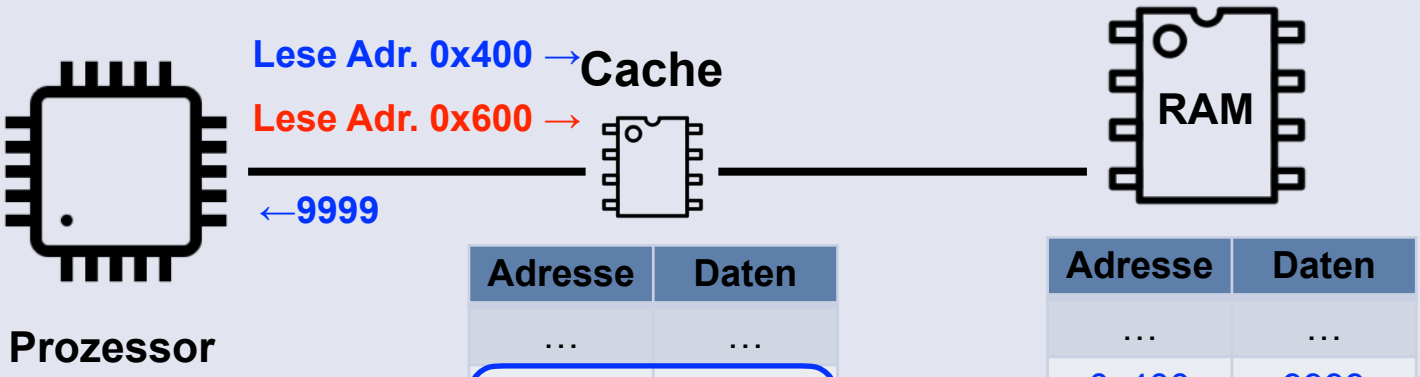
Adresse	Daten
…	…
0x400	9999
…	…
…	…

Adresse	Daten
…	…
0x400	9999
…	…
0x600	5555

Funktionsweise von Caches

Organisation von Caches (im engeren Sinn):

- Prüfung anhand der Speicheradresse, ob benötigte Daten im Cache vorhanden sind („Treffer“; ***cache hit***)
- Falls nicht (***cache miss***):
  - Zugriff auf den Hauptspeicher zum Lesen der Daten
  - Speichern der Daten in einem Cacheeintrag



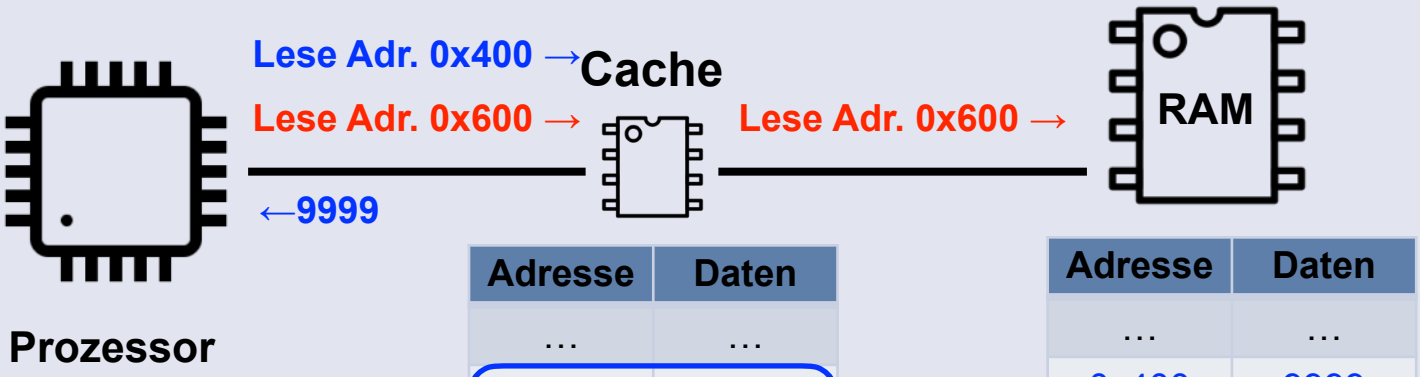
Adresse	Daten
…	…
0x400	9999
…	…
…	…

Adresse	Daten
…	…
0x400	9999
…	…
0x600	5555

Funktionsweise von Caches

Organisation von Caches (im engeren Sinn):

- Prüfung anhand der Speicheradresse, ob benötigte Daten im Cache vorhanden sind („Treffer“; ***cache hit***)
- Falls nicht (***cache miss***):
  - Zugriff auf den Hauptspeicher zum Lesen der Daten
  - Speichern der Daten in einem Cacheeintrag



Adresse	Daten
…	…
0x400	9999
…	…
…	…

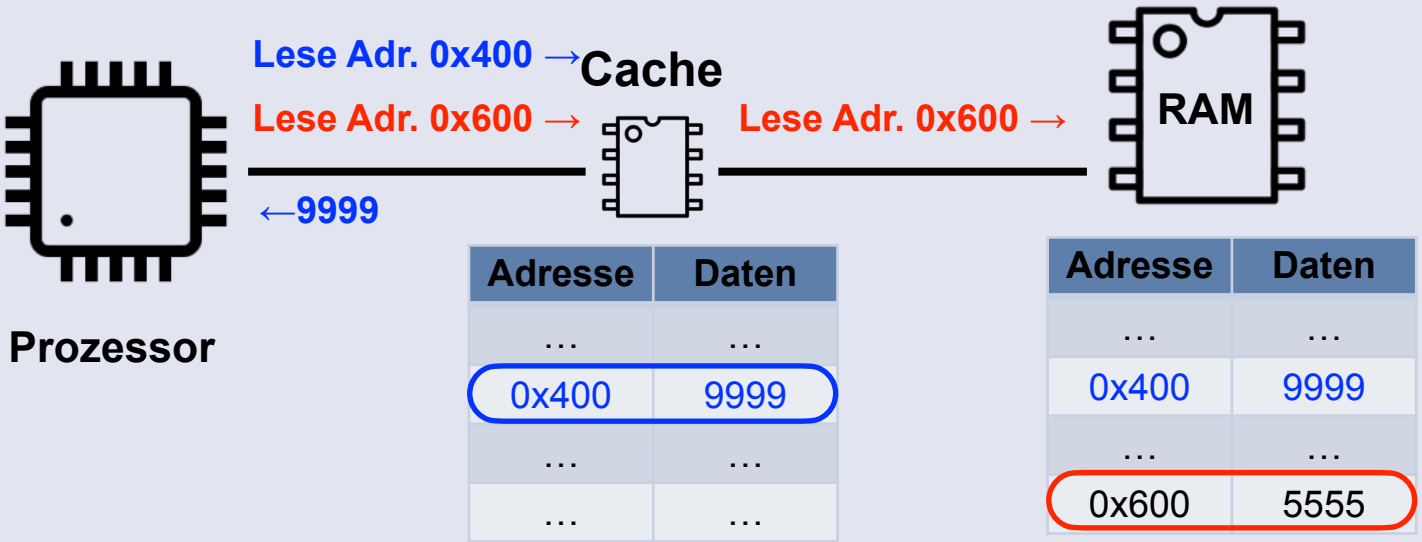
Adresse	Daten
…	…
0x400	9999
…	…
0x600	5555

Funktionsweise von Caches

Universität Bamberg

Organisation von Caches (im engeren Sinn):

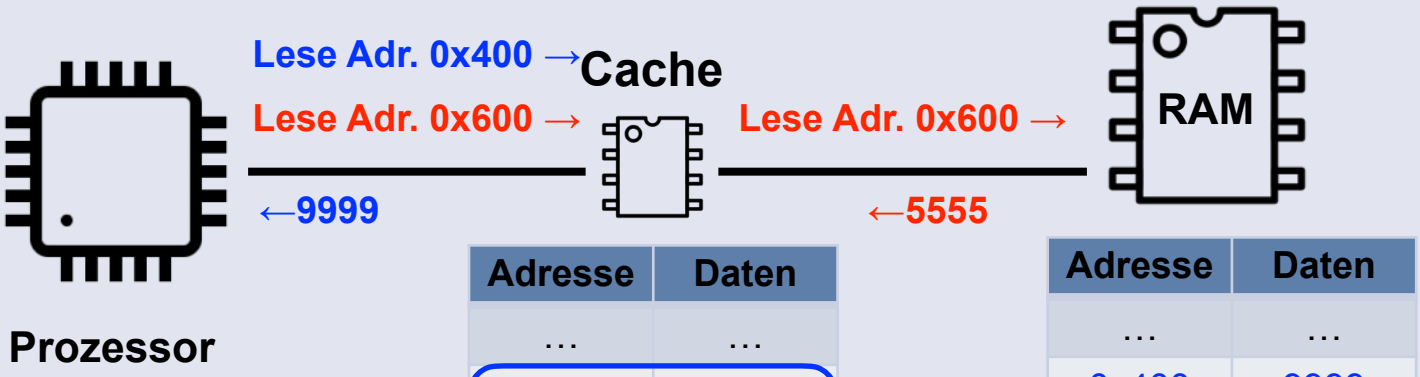
- Prüfung anhand der Speicheradresse, ob benötigte Daten im Cache vorhanden sind („Treffer“; ***cache hit***)
- Falls nicht (***cache miss***):
  - Zugriff auf den Hauptspeicher zum Lesen der Daten
  - Speichern der Daten in einem Cacheeintrag



Funktionsweise von Caches

Organisation von Caches (im engeren Sinn):

- Prüfung anhand der Speicheradresse, ob benötigte Daten im Cache vorhanden sind („Treffer“; ***cache hit***)
- Falls nicht (***cache miss***):
  - Zugriff auf den Hauptspeicher zum Lesen der Daten
  - Speichern der Daten in einem Cacheeintrag



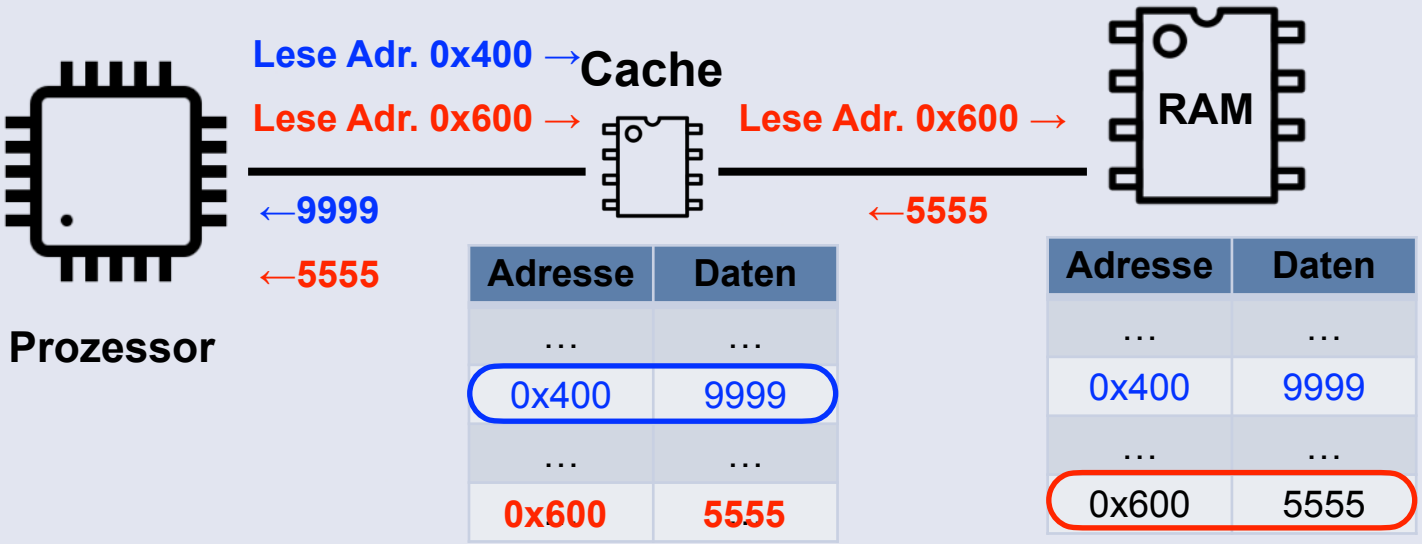
Adresse	Daten
…	…
0x400	9999
…	…
…	…

Adresse	Daten
…	…
0x400	9999
…	…
0x600	5555

Funktionsweise von Caches

Organisation von Caches (im engeren Sinn):

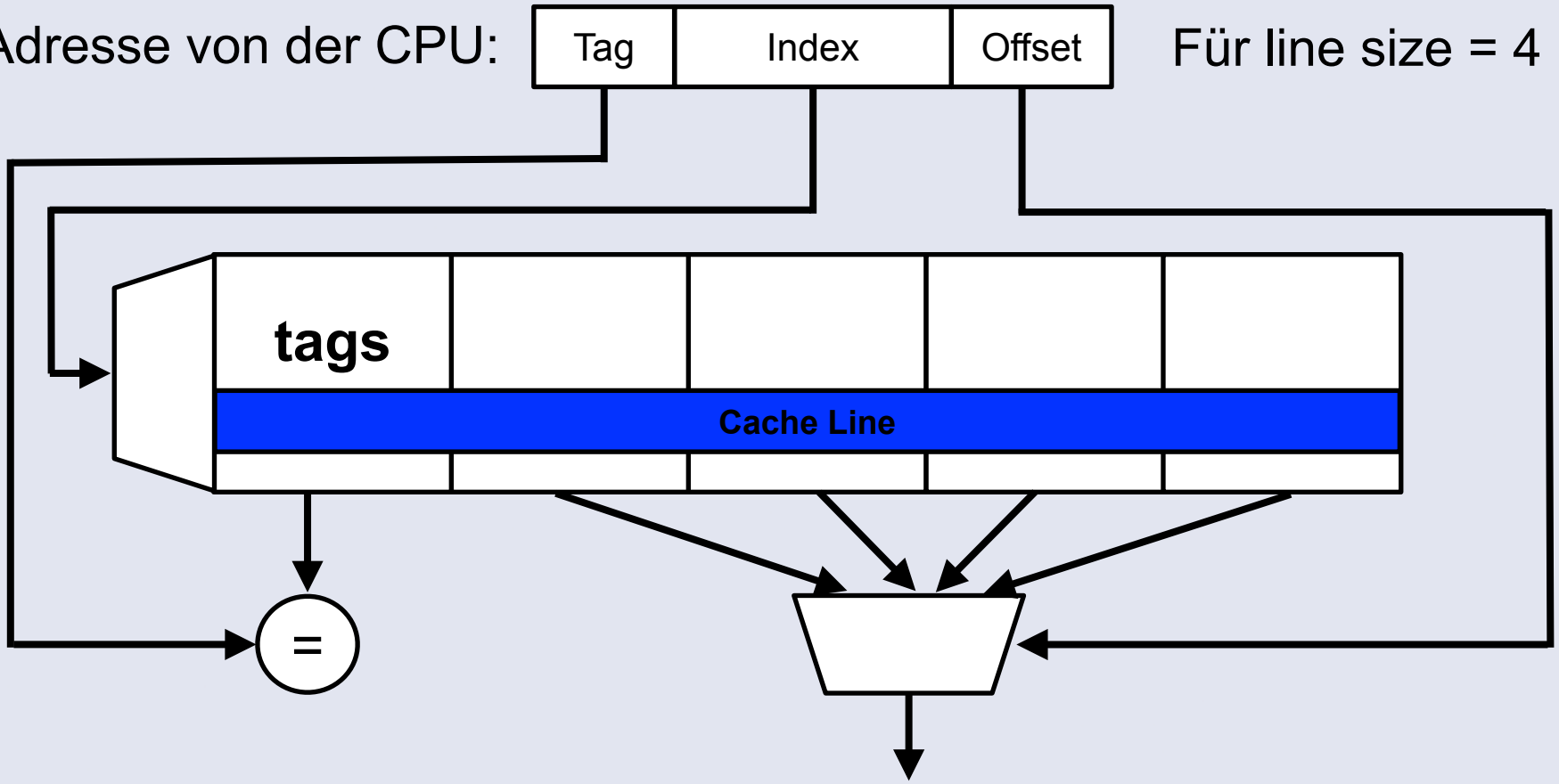
- Prüfung anhand der Speicheradresse, ob benötigte Daten im Cache vorhanden sind („Treffer“; ***cache hit***)
- Falls nicht (***cache miss***):
  - Zugriff auf den Hauptspeicher zum Lesen der Daten
  - Speichern der Daten in einem Cacheeintrag



Prüfung auf *cache hit*: Cache-Zeilen

Such-Einheit im Cache: Cache-Zeile (cache line)

Adresse von der CPU:

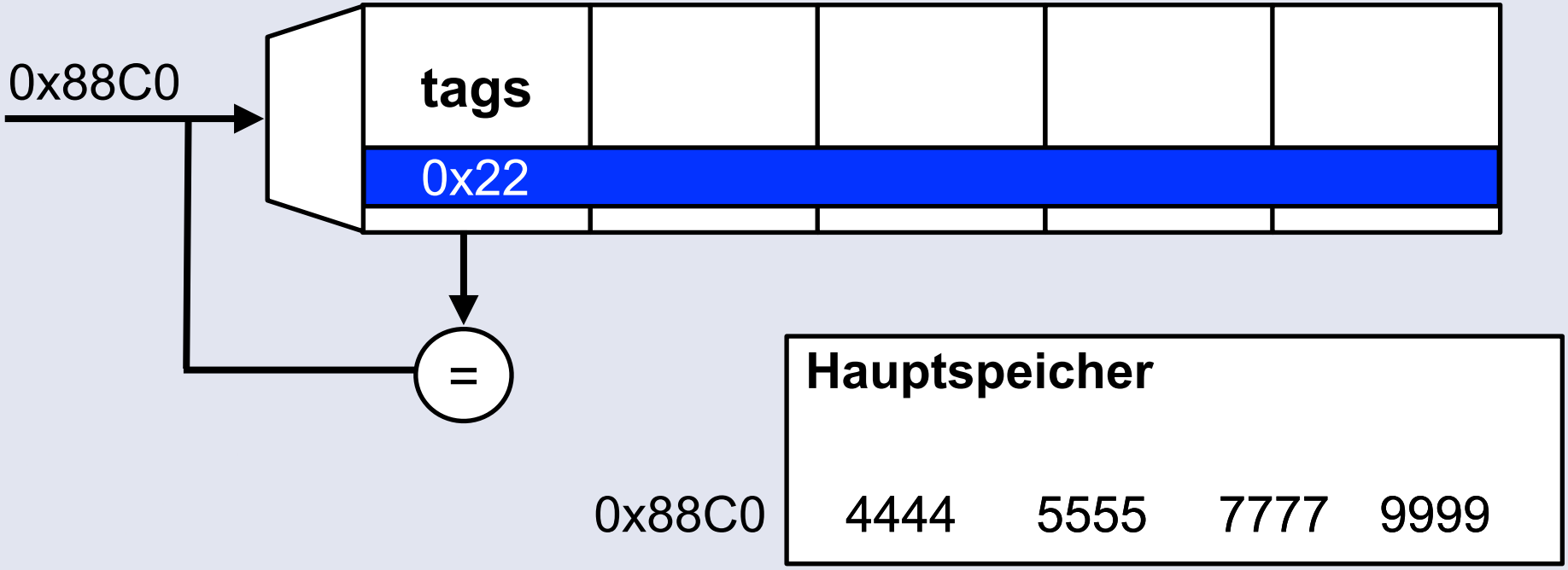


Für line size = 4

Benötigt weniger *tag bits*, als wenn man jedem Wort *tag bits* zuordnen würde

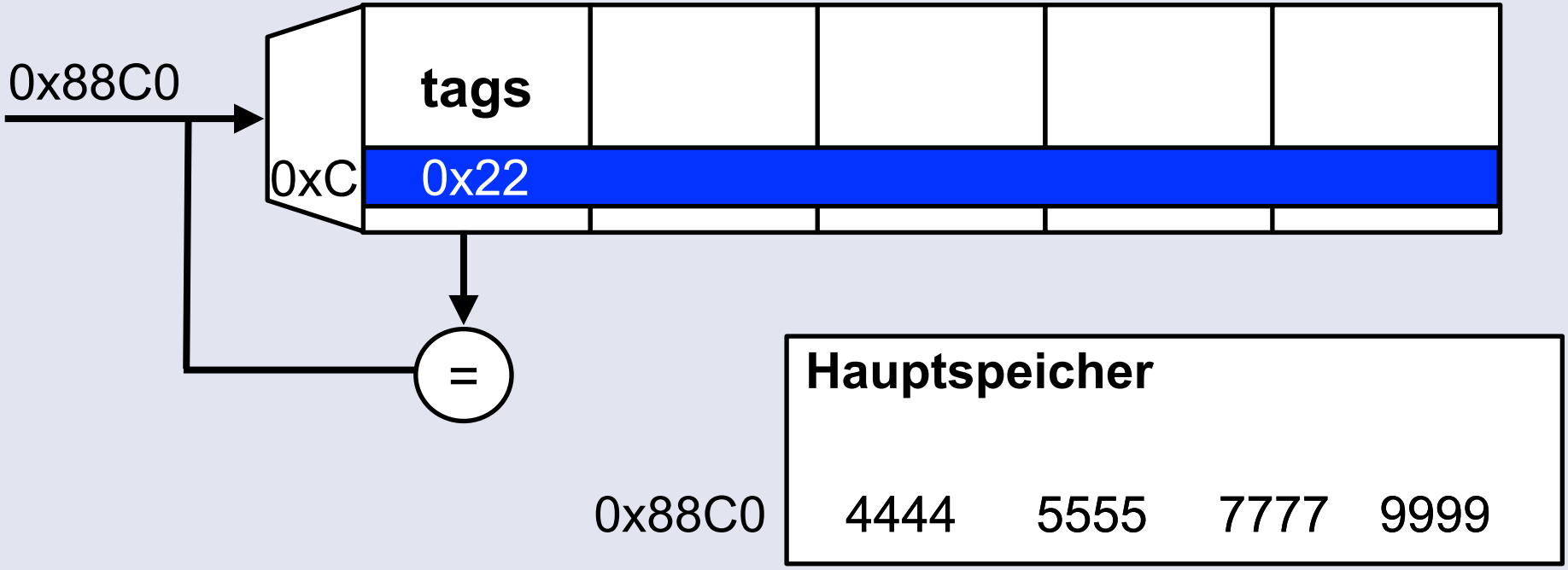
Cache-Blöcke

- Die **Blockgröße** ist die Anzahl der Worte, die im Fall eines *cache miss* aus dem Speicher nachgeladen werden
  - Beispiel: (Blockgröße = *line size*):



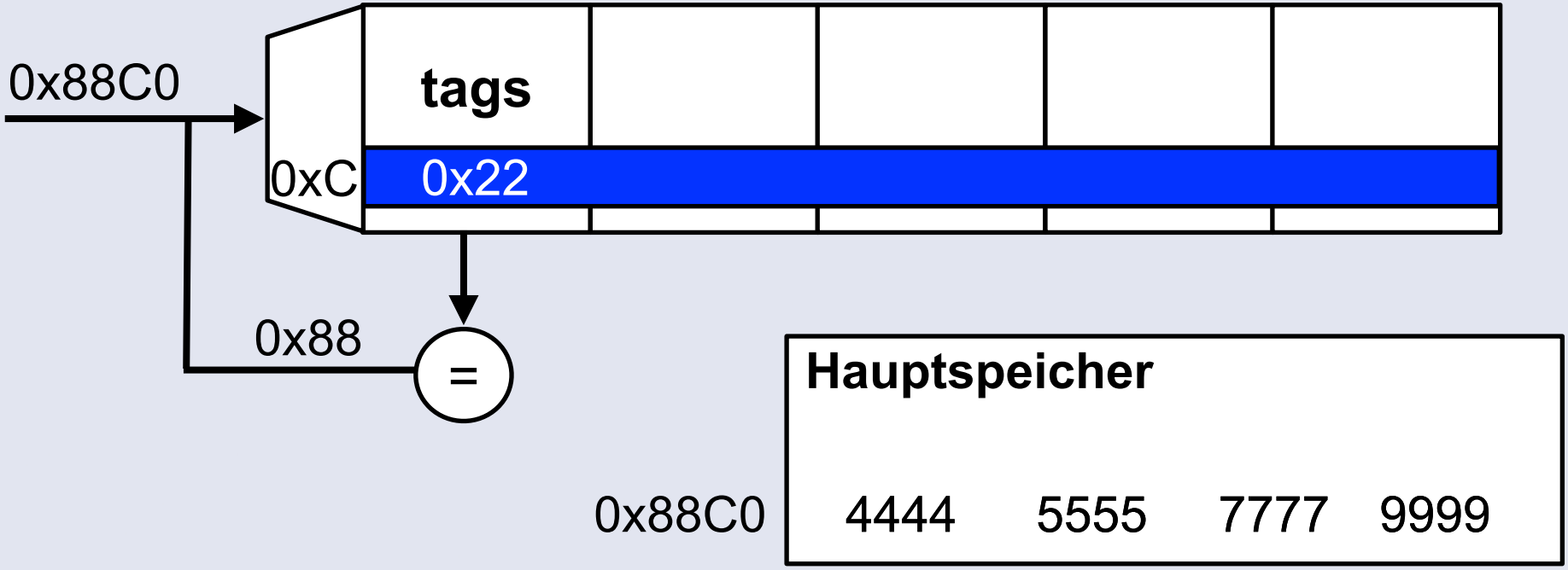
Cache-Blöcke

- Die **Blockgröße** ist die Anzahl der Worte, die im Fall eines *cache miss* aus dem Speicher nachgeladen werden
  - Beispiel: (Blockgröße = *line size*):



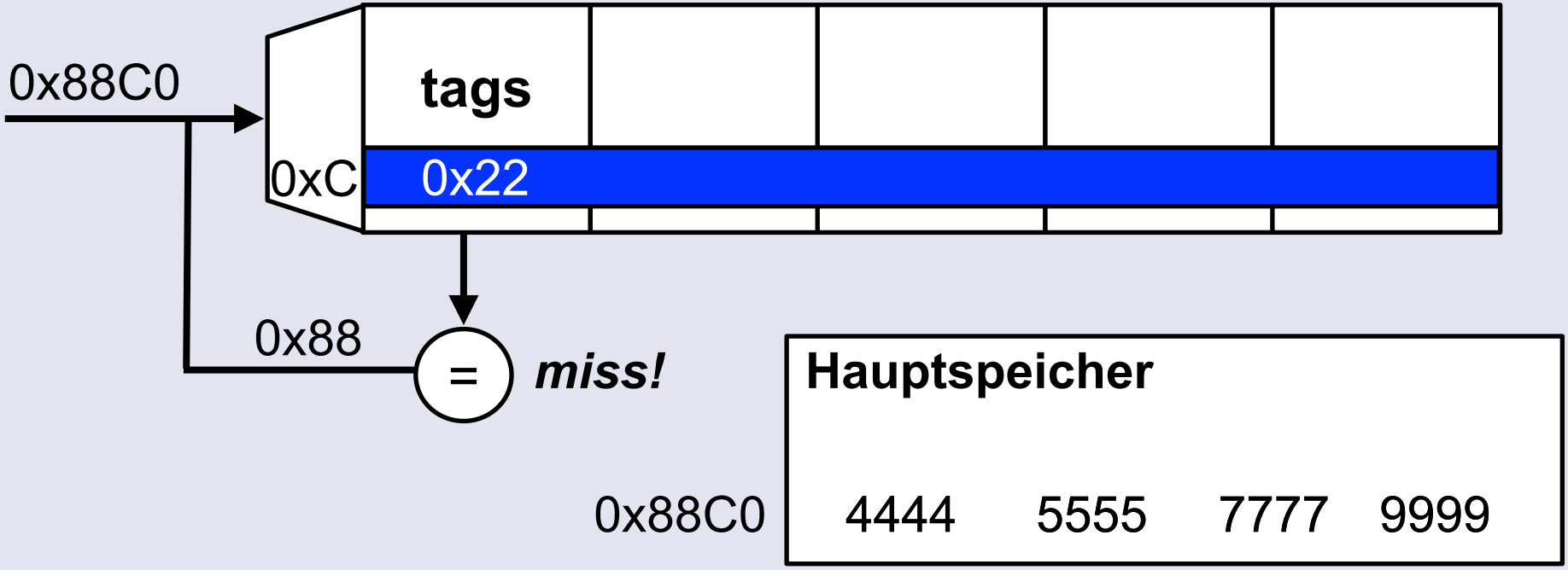
Cache-Blöcke

- Die **Blockgröße** ist die Anzahl der Worte, die im Fall eines *cache miss* aus dem Speicher nachgeladen werden
  - Beispiel: (Blockgröße = *line size*):



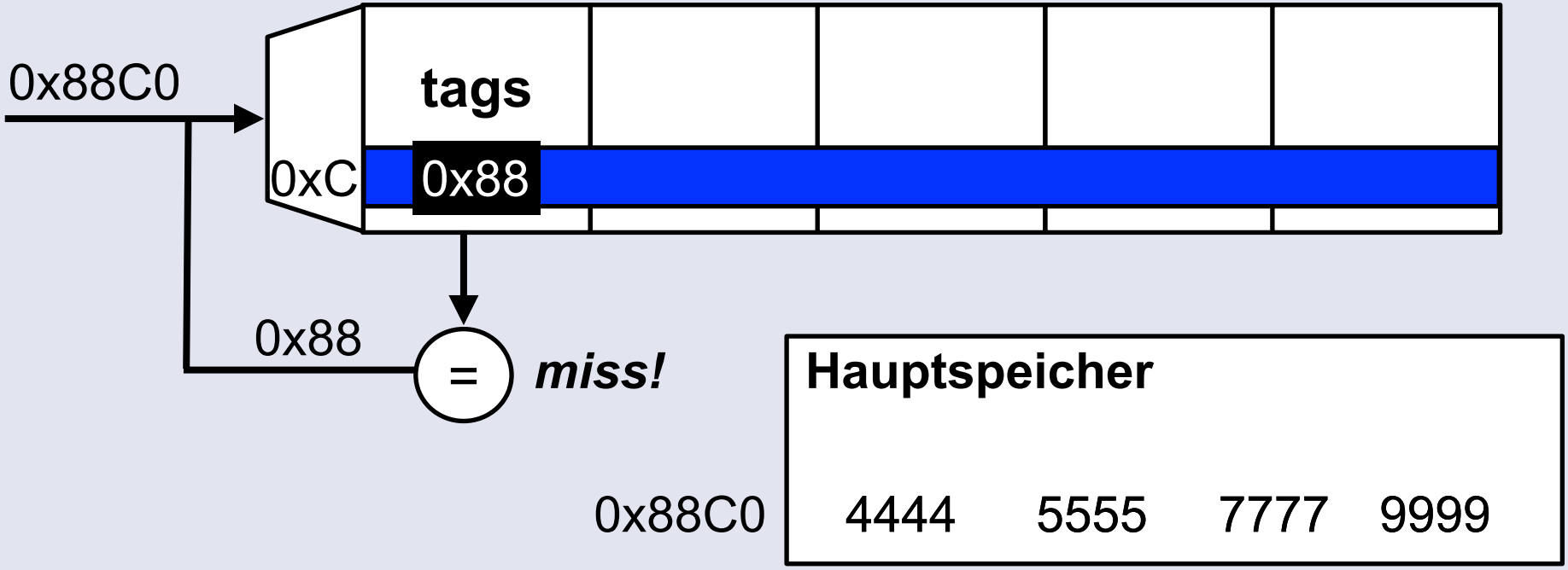
Cache-Blöcke

- Die **Blockgröße** ist die Anzahl der Worte, die im Fall eines *cache miss* aus dem Speicher nachgeladen werden
  - Beispiel: (Blockgröße = *line size*):



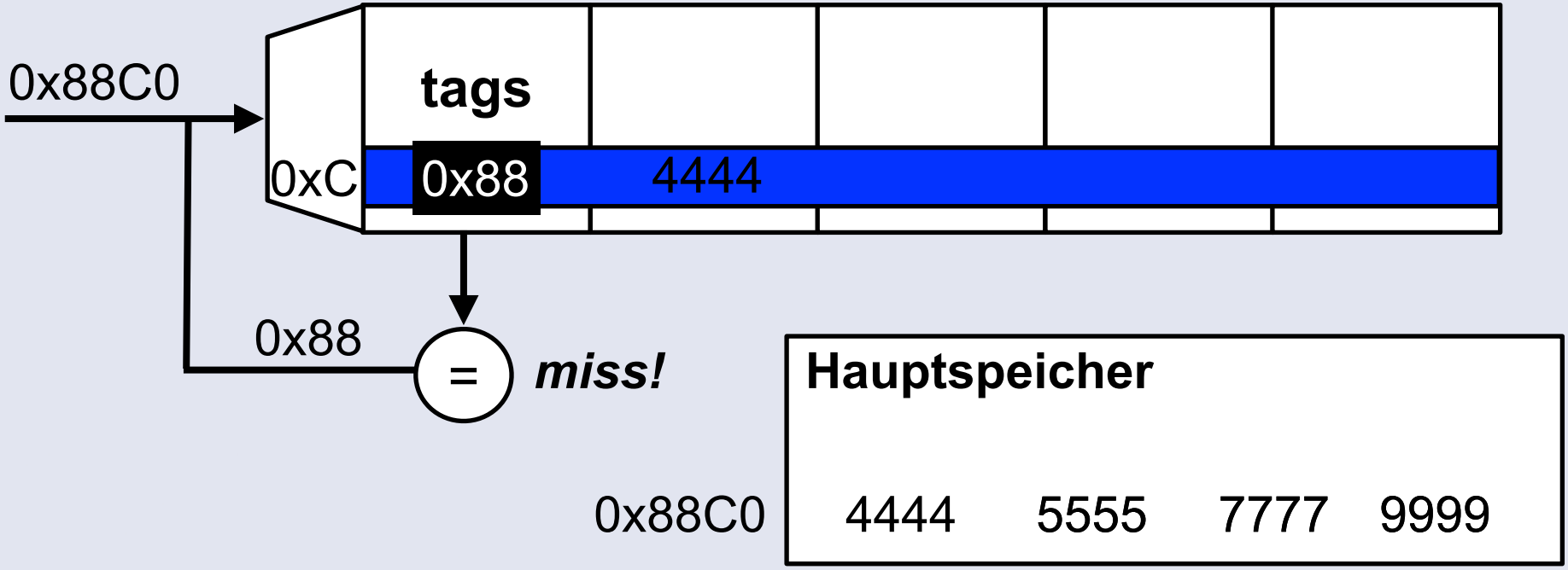
Cache-Blöcke

- Die **Blockgröße** ist die Anzahl der Worte, die im Fall eines *cache miss* aus dem Speicher nachgeladen werden
  - Beispiel: (Blockgröße = *line size*):



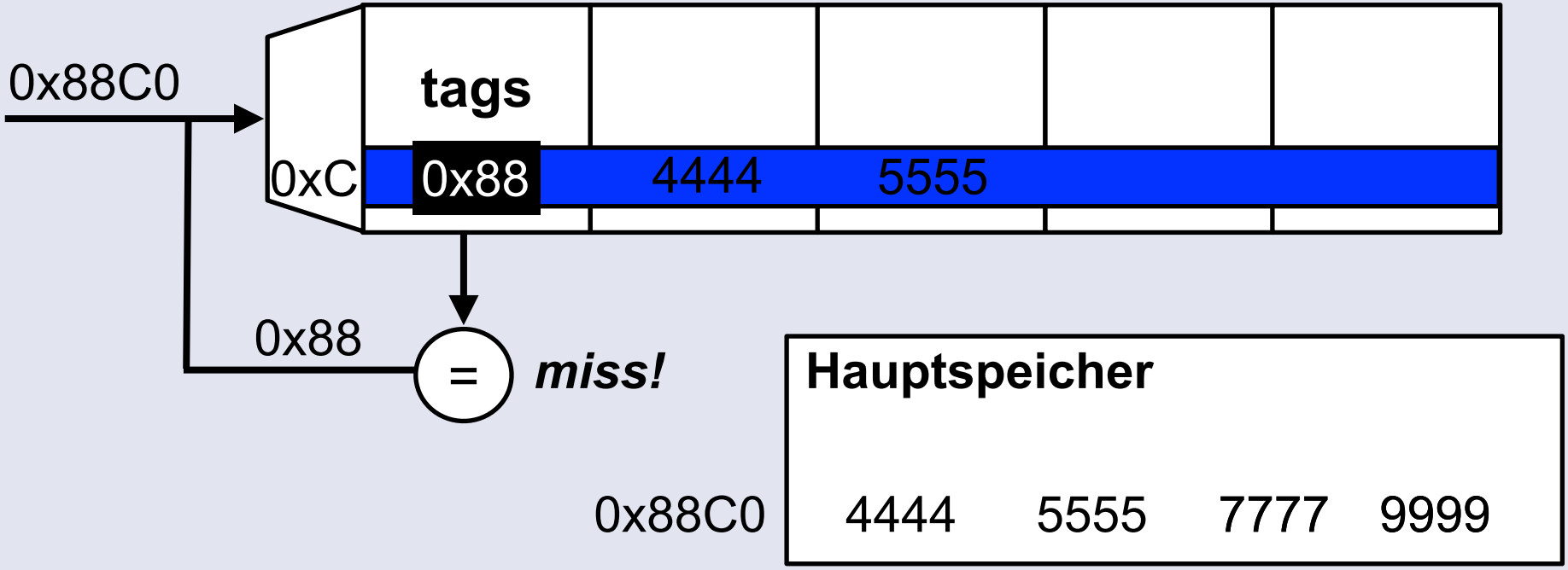
Cache-Blöcke

- Die **Blockgröße** ist die Anzahl der Worte, die im Fall eines *cache miss* aus dem Speicher nachgeladen werden
  - Beispiel: (Blockgröße = *line size*):



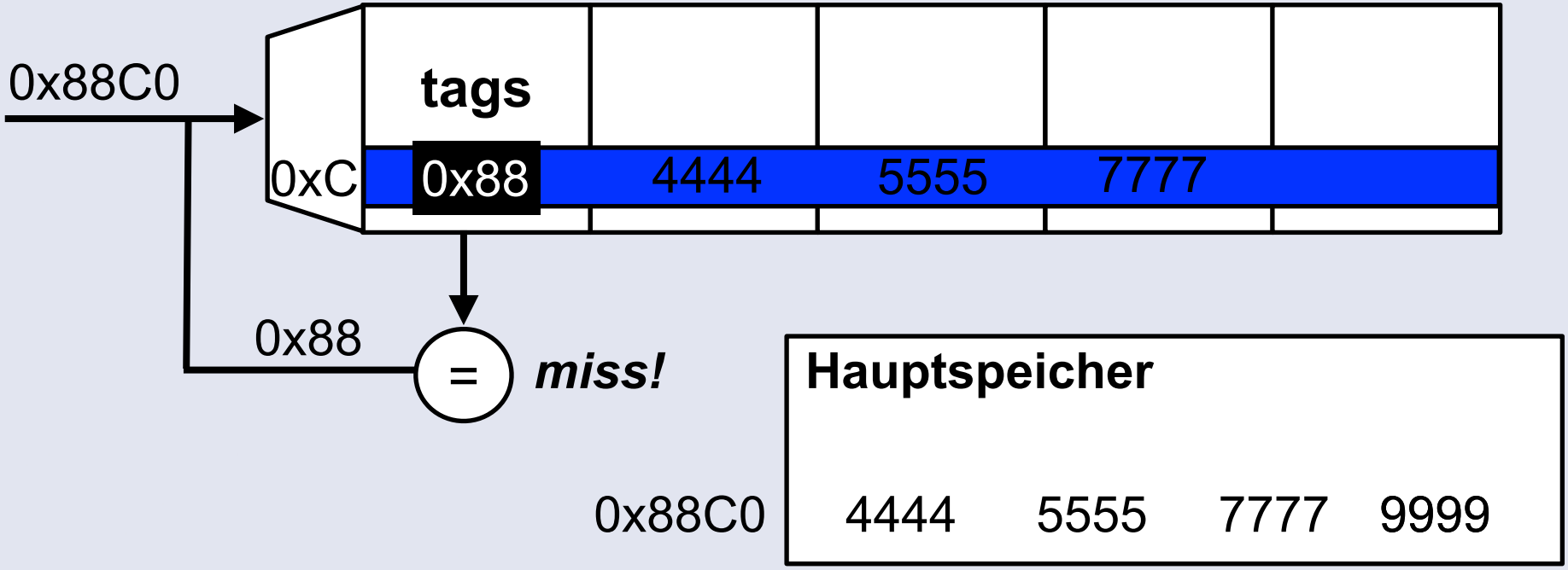
Cache-Blöcke

- Die **Blockgröße** ist die Anzahl der Worte, die im Fall eines *cache miss* aus dem Speicher nachgeladen werden
  - Beispiel: (Blockgröße = *line size*):



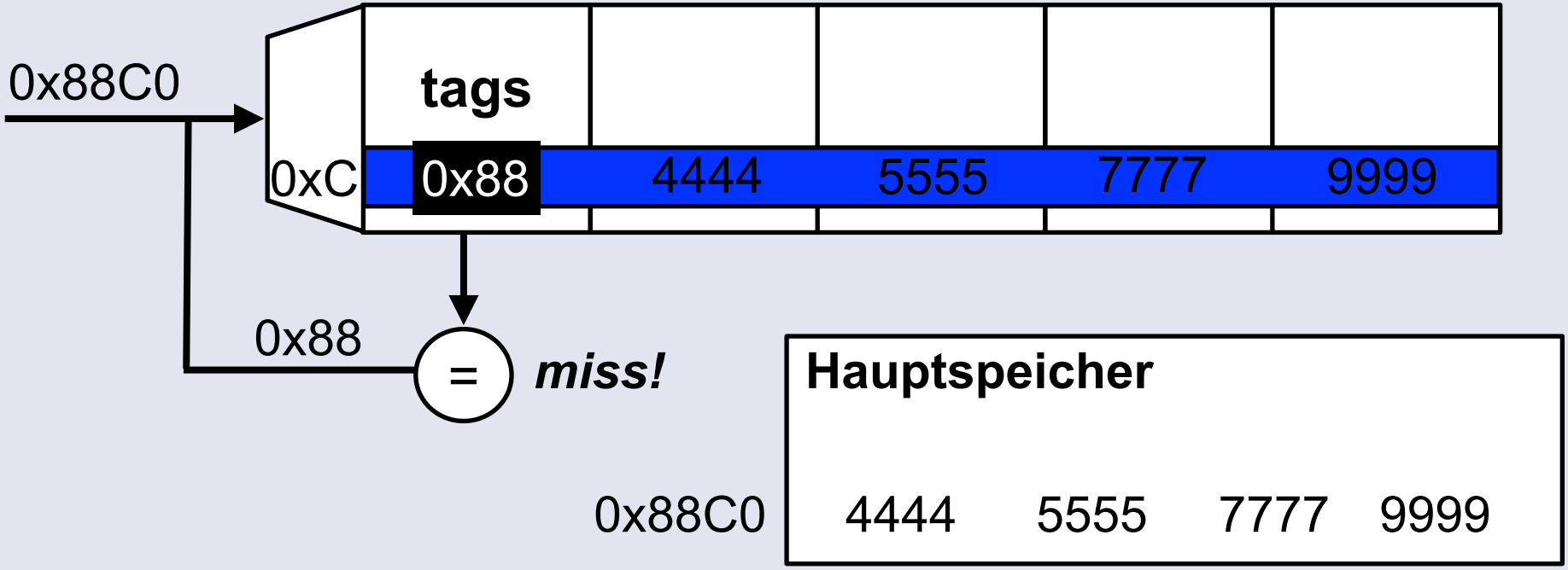
Cache-Blöcke

- Die **Blockgröße** ist die Anzahl der Worte, die im Fall eines *cache miss* aus dem Speicher nachgeladen werden
  - Beispiel: (Blockgröße = *line size*):



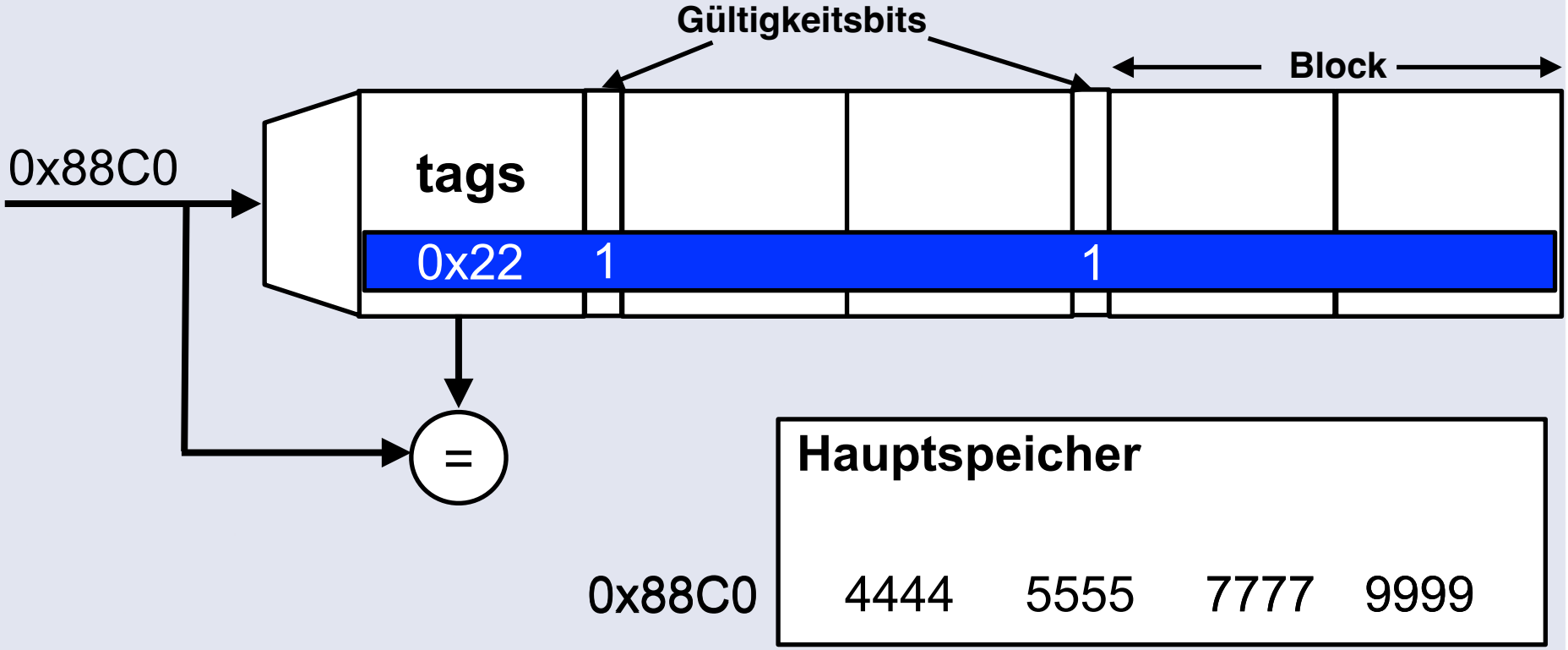
Cache-Blöcke

- Die **Blockgröße** ist die Anzahl der Worte, die im Fall eines *cache miss* aus dem Speicher nachgeladen werden
  - Beispiel: (Blockgröße = *line size*):



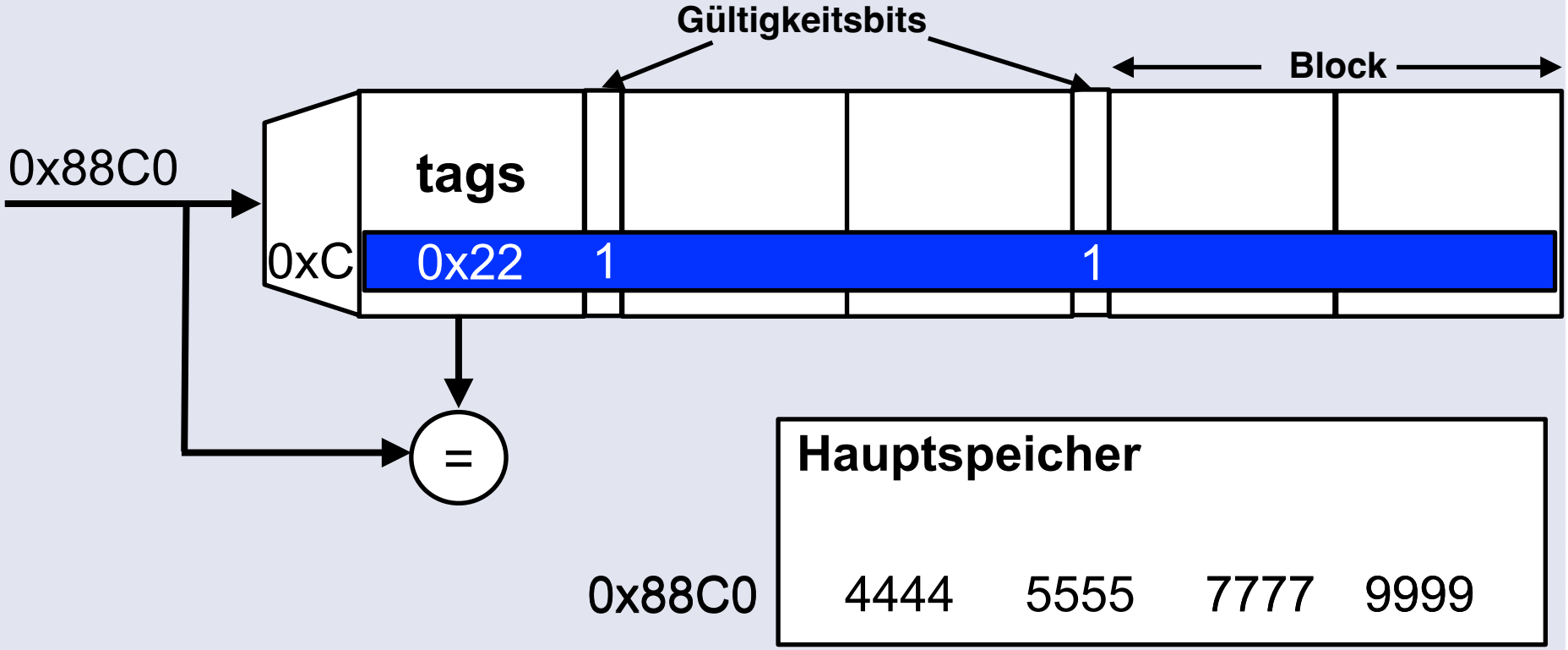
Cache-Blöcke (2)

- Wenn *block size* < *line size*, dann sind zusätzliche Gültigkeitsbits erforderlich
  - Beispiel: Blockgröße = *line size* / 2



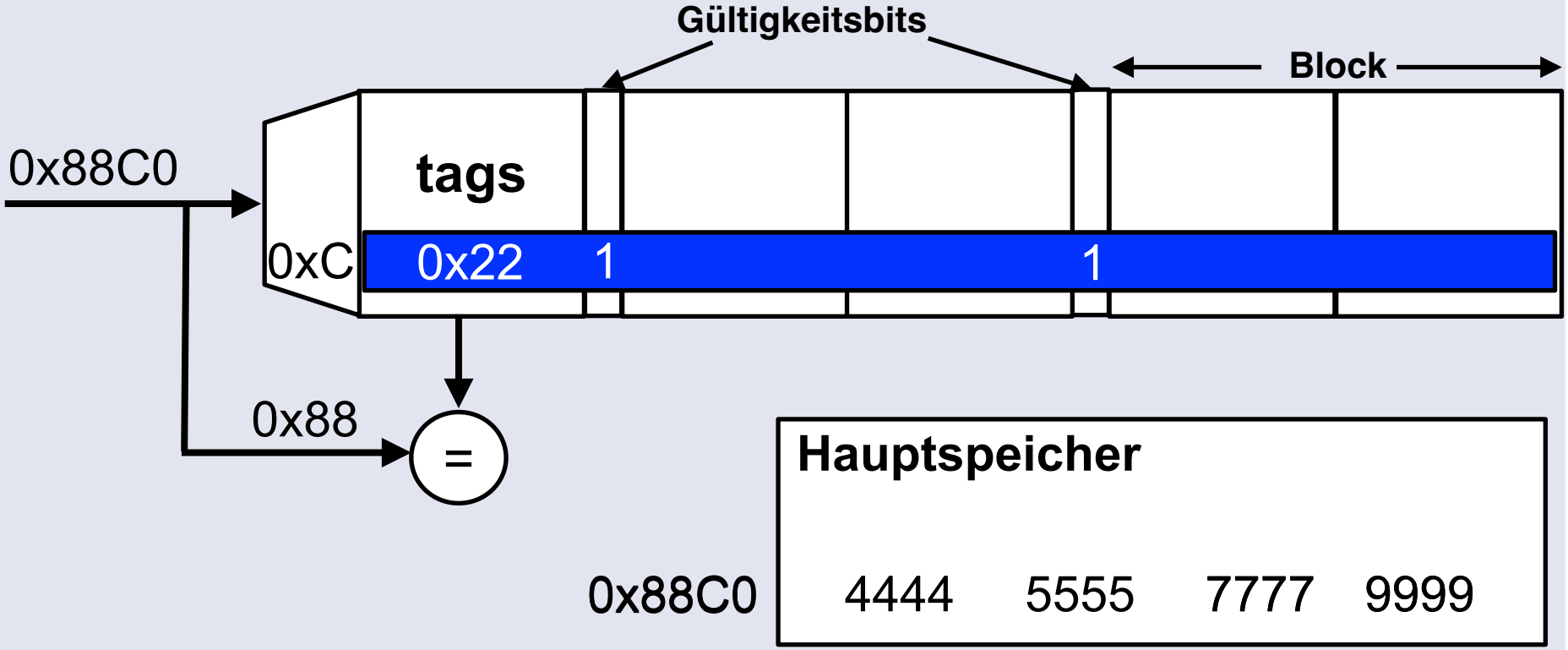
Cache-Blöcke (2)

- Wenn *block size* < *line size*, dann sind zusätzliche Gültigkeitsbits erforderlich
  - Beispiel: Blockgröße = *line size* / 2



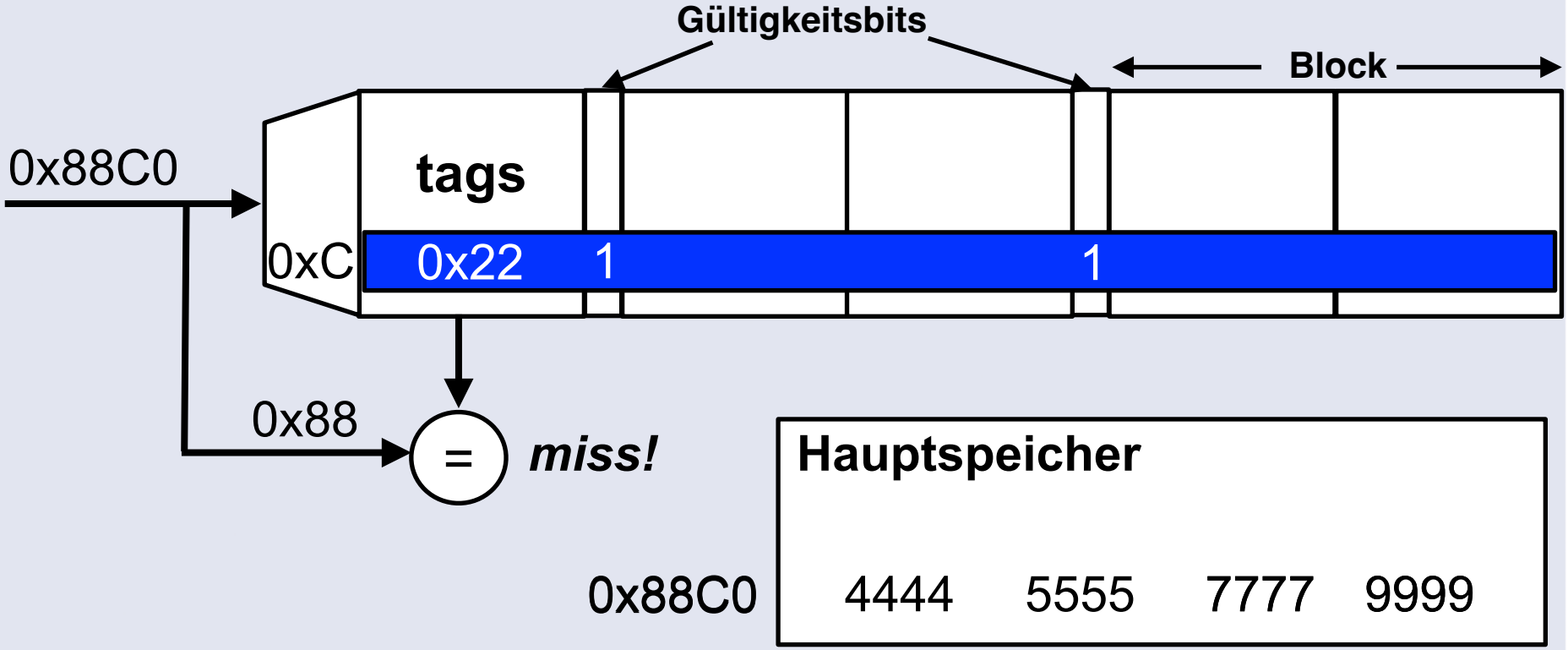
Cache-Blöcke (2)

- Wenn *block size* < *line size*, dann sind zusätzliche Gültigkeitsbits erforderlich
  - Beispiel: Blockgröße = *line size* / 2



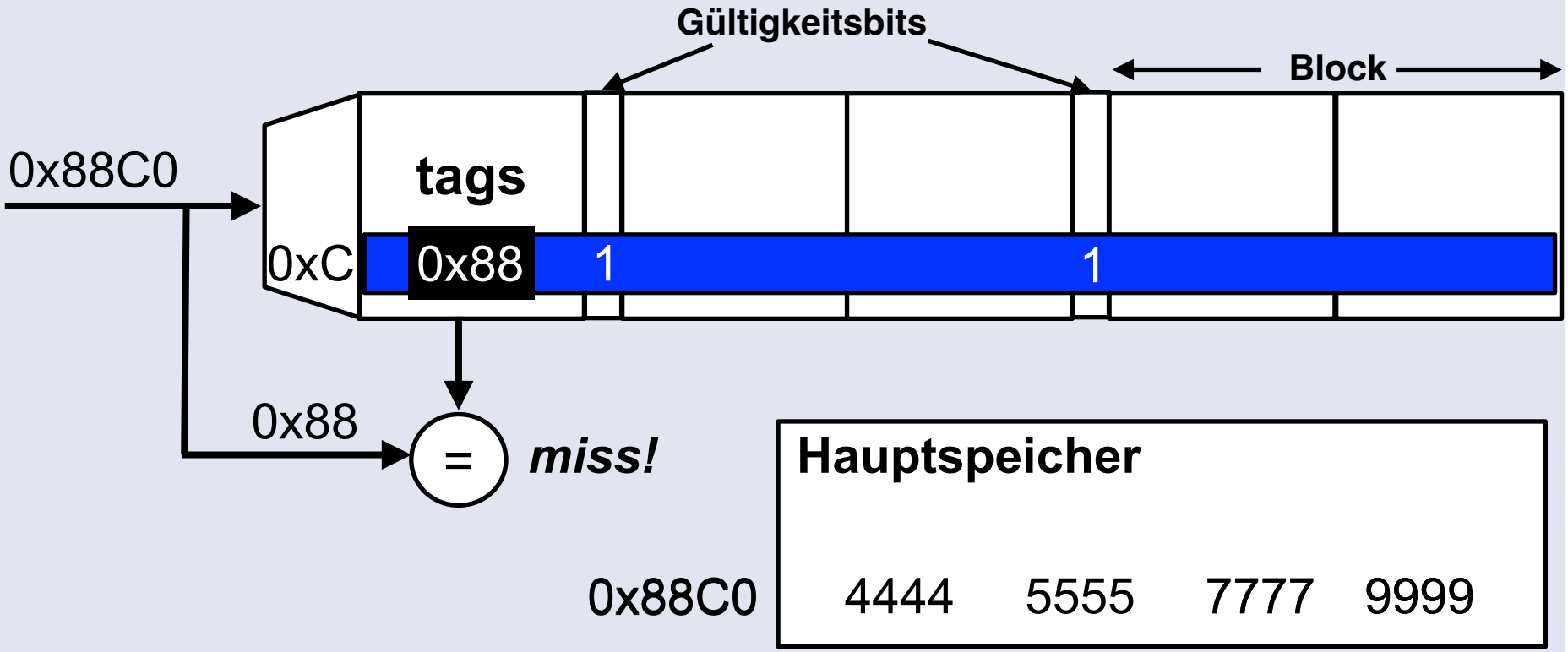
Cache-Blöcke (2)

- Wenn *block size* < *line size*, dann sind zusätzliche Gültigkeitsbits erforderlich
  - Beispiel: Blockgröße = *line size* / 2



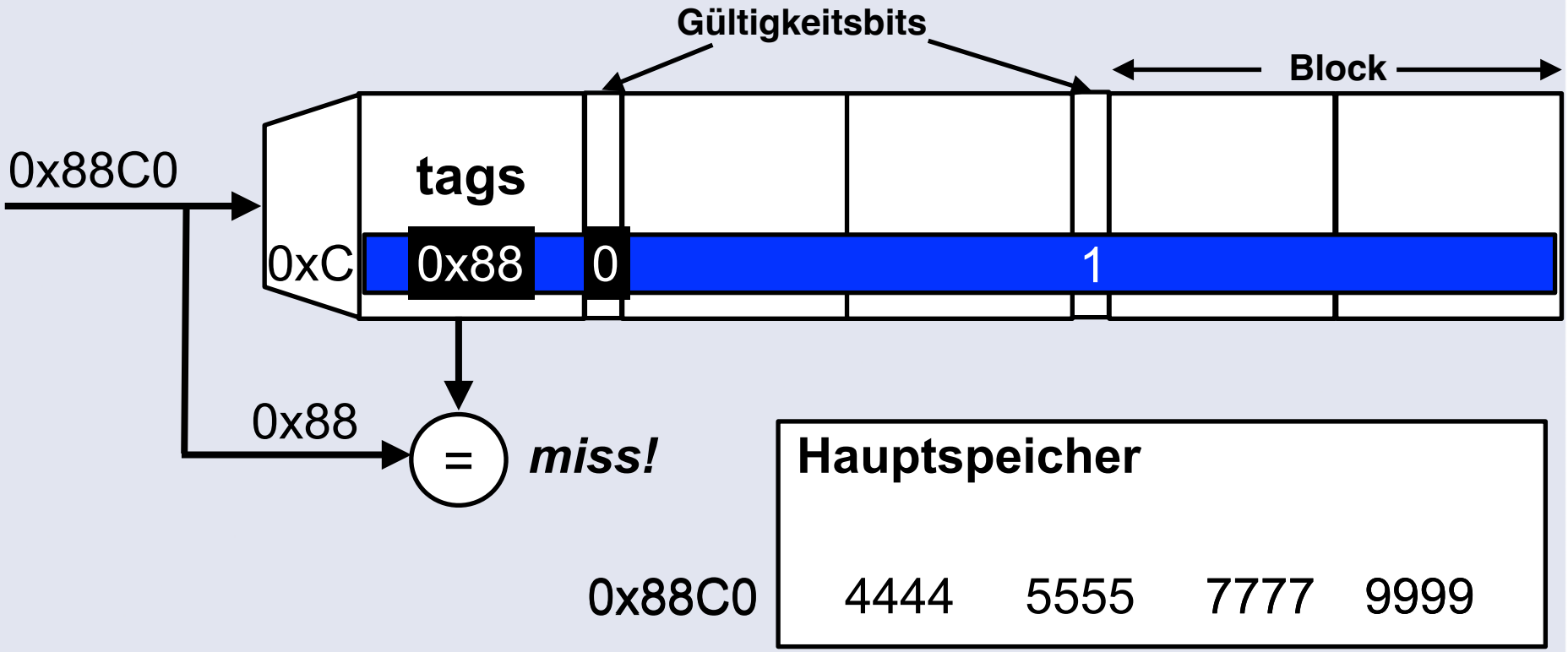
Cache-Blöcke (2)

- Wenn *block size* < *line size*, dann sind zusätzliche Gültigkeitsbits erforderlich
  - Beispiel: Blockgröße = *line size* / 2



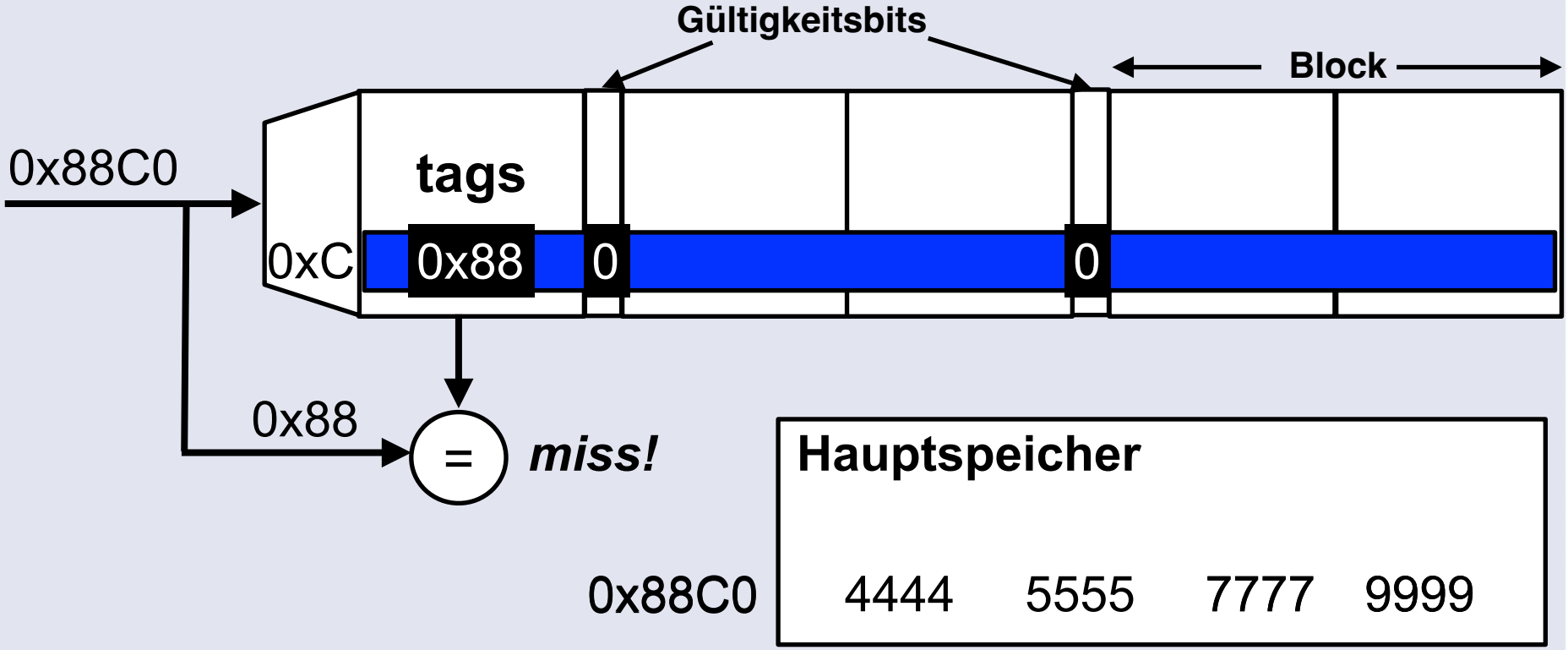
Cache-Blöcke (2)

- Wenn *block size* < *line size*, dann sind zusätzliche Gültigkeitsbits erforderlich
  - Beispiel: Blockgröße = *line size* / 2



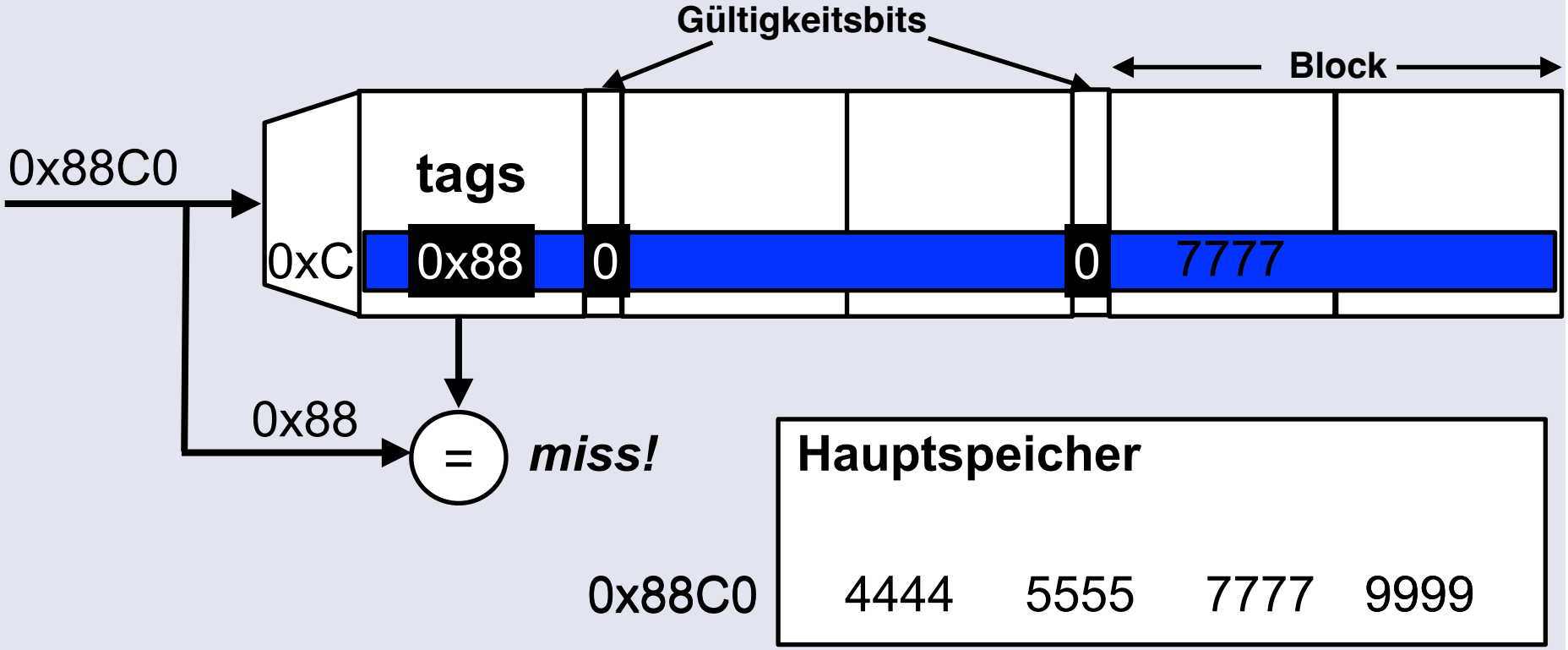
Cache-Blöcke (2)

- Wenn *block size* < *line size*, dann sind zusätzliche Gültigkeitsbits erforderlich
  - Beispiel: Blockgröße = *line size* / 2



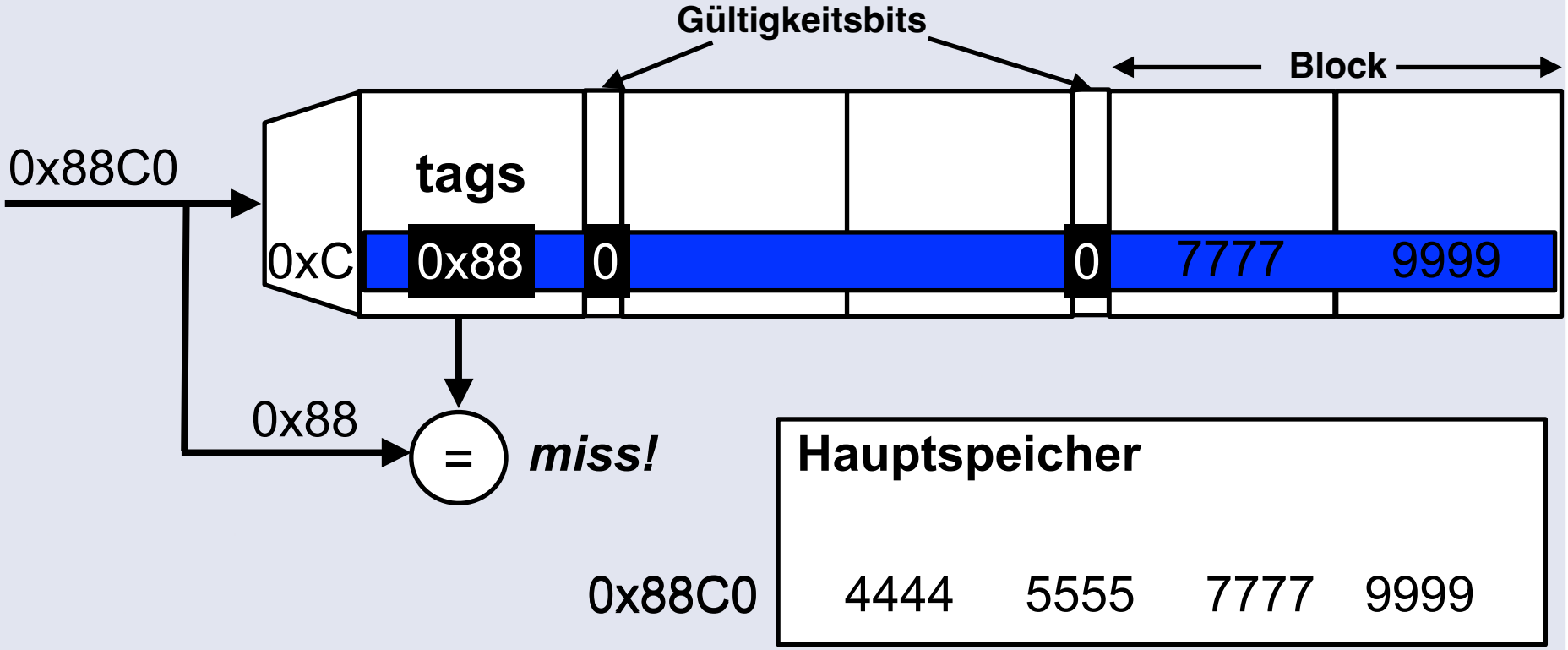
Cache-Blöcke (2)

- Wenn *block size* < *line size*, dann sind zusätzliche Gültigkeitsbits erforderlich
  - Beispiel: Blockgröße = *line size* / 2



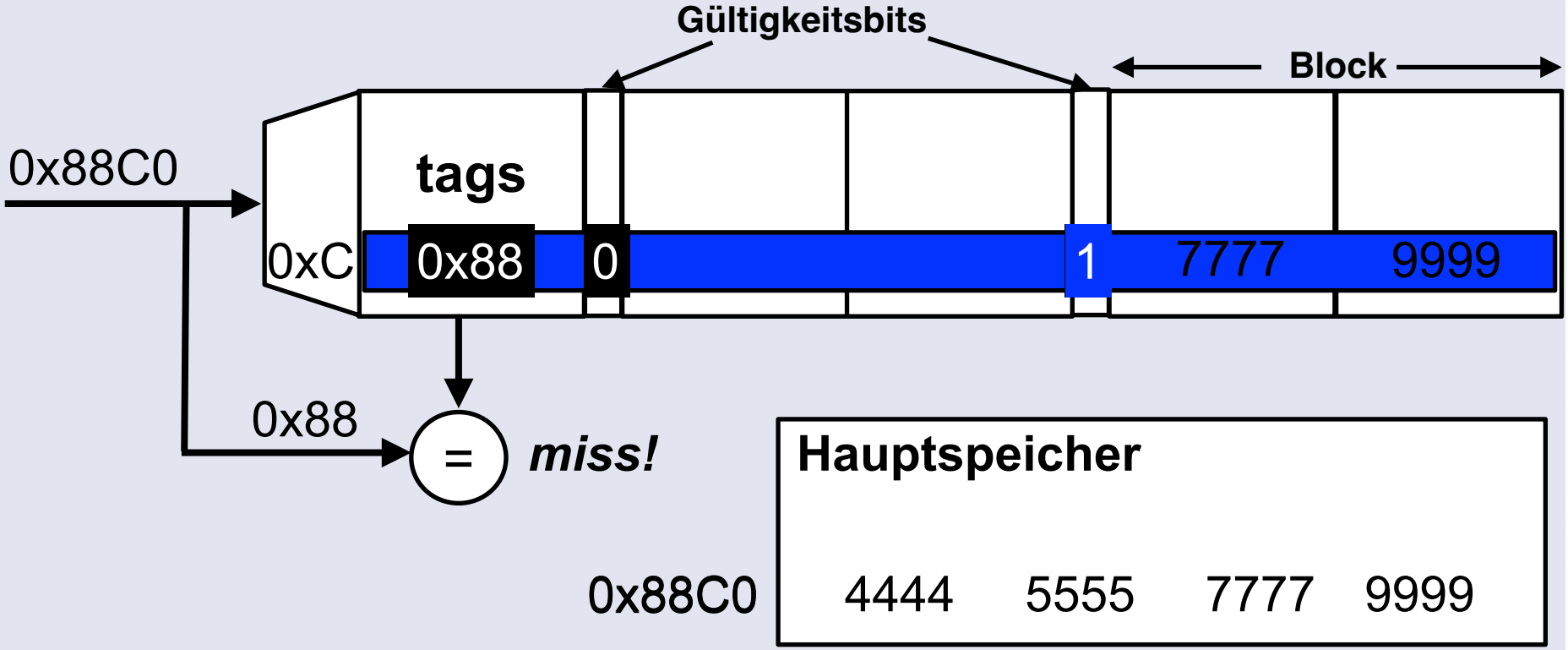
Cache-Blöcke (2)

- Wenn *block size* < *line size*, dann sind zusätzliche Gültigkeitsbits erforderlich
  - Beispiel: Blockgröße = *line size* / 2



Cache-Blöcke (2)

- Wenn *block size* < *line size*, dann sind zusätzliche Gültigkeitsbits erforderlich
  - Beispiel: Blockgröße = *line size* / 2

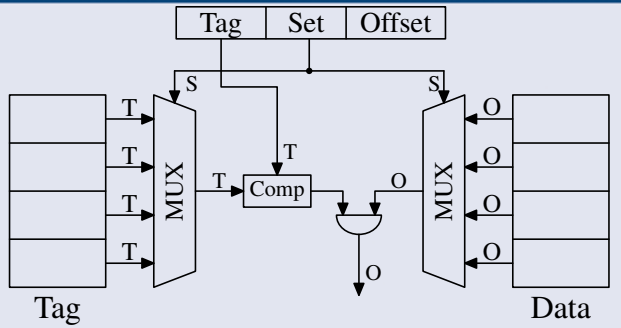


Cache-Blöcke (3)

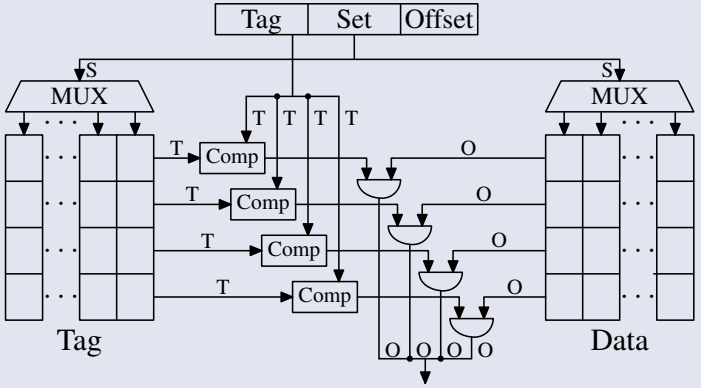
- Wenn *block size* > *line size*, dann werden bei jedem *miss* mehrere Zeilen nachgeladen
  - Stets wird zuerst das gesuchte Wort, dann der Rest des Blocks geladen
- Verbindung Speicher/Cache so entworfen, dass der Speicher durch das zusätzliche Lesen nicht langsamer wird
- Methoden dazu:
  1. Schnelles Lesen aufeinanderfolgender Speicherzellen (nibble mode, block/page mode der Speicher)
  2. Interleaving (mehrere Speicher-ICs mit überlappenden Zugriffen)
  3. Block-Mode beim Buszugriff, Page-Mode beim Speicherzugriff
  4. Fließbandzugriff auf den Speicher (EDO-RAM, SDRAM)
  5. breite Speicher, die mehrere Worte parallel übertragen können

Organisationsformen von Caches

- **Direct mapping** (direkte Abbildung)
  - Jedes Tag passt nur zu *genau einem* Cacheeintrag
  - Für caching von Befehlen besonders sinnvoll, weil bei Befehlen aliasing sehr unwahrscheinlich ist



- **Set associative mapping** (mengenassoziative Abbildung)
  - Tag und Daten in Mengen (Größe 2^n) unterteilt
  - Ein passendes Tag auf Menge abgebildet, ein Element der Menge *kann* matchen
  - Sehr häufige Organisationsform, mit Set-Größe=2 oder 4, selten 8



- **Fully Associative mapping** (vollassoziative Abbildung)
  - Jede Cachezeile kann den Inhalt einer beliebigen Speicheradresse enthalten
  - Wegen der Größe eines Caches kommt diese Organisationsform kaum in Frage

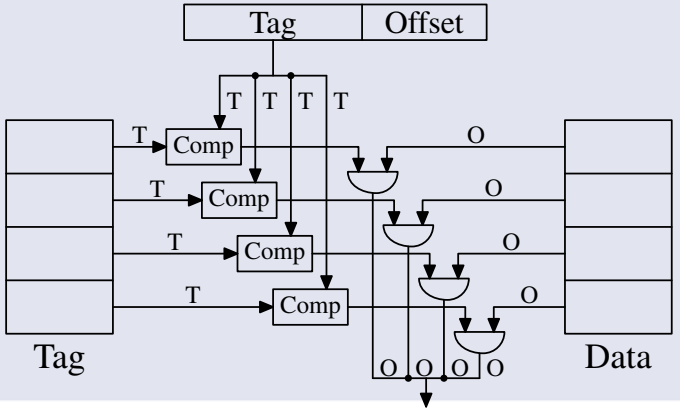


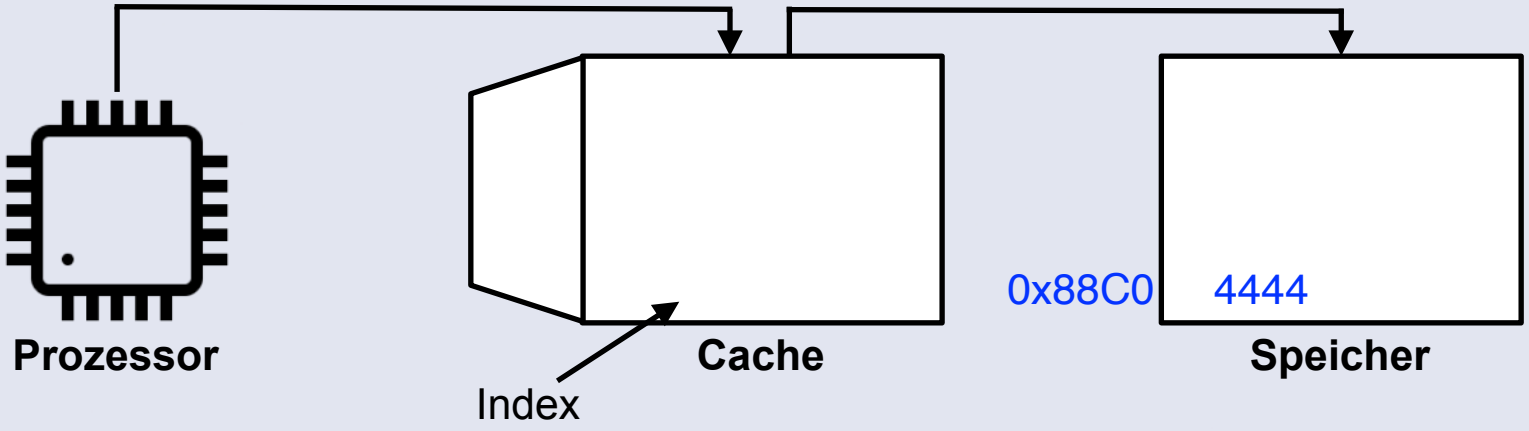
Abb. aus [3]

Schreibverfahren

- Strategien zum Rückschreiben Cache → (Haupt-) Speicher

  1. Write-Through (durchschreiben)

  - Jeder Schreibvorgang in den Cache führt zu einer unmittelbaren Aktualisierung des (Haupt-) Speichers



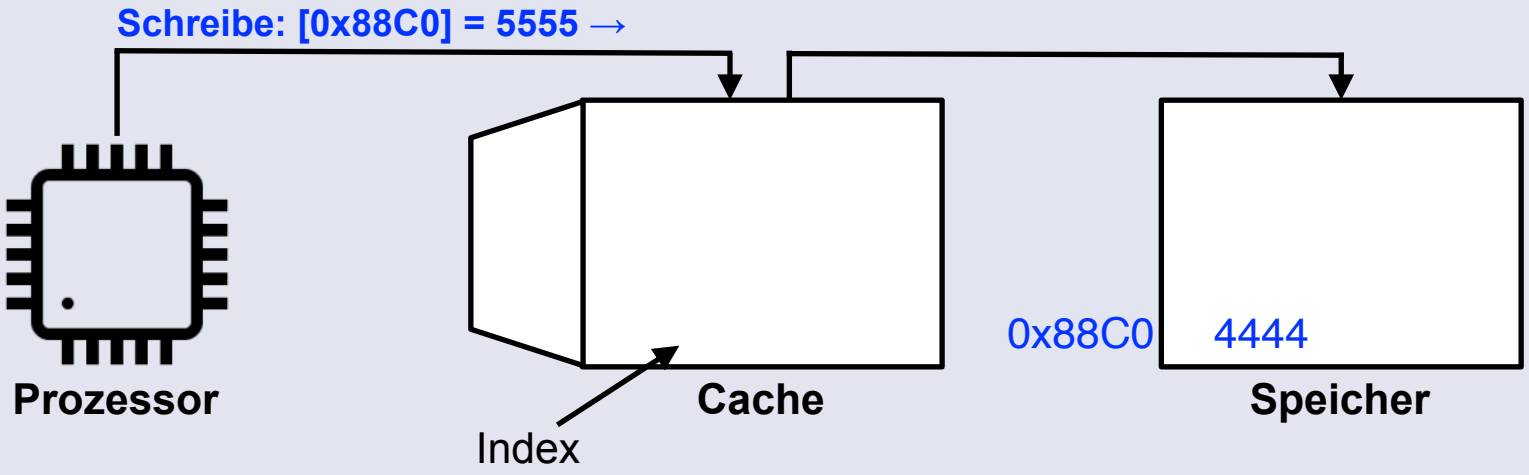
Speicher wird Engpass, es sei denn, der Anteil an Schreiboperationen ist klein oder der (Haupt-)Speicher ist nur wenig langsamer als der Cache

Schreibverfahren

- Strategien zum Rückschreiben Cache → (Haupt-) Speicher

  1. Write-Through (durchschreiben)

  - Jeder Schreibvorgang in den Cache führt zu einer unmittelbaren Aktualisierung des (Haupt-) Speichers



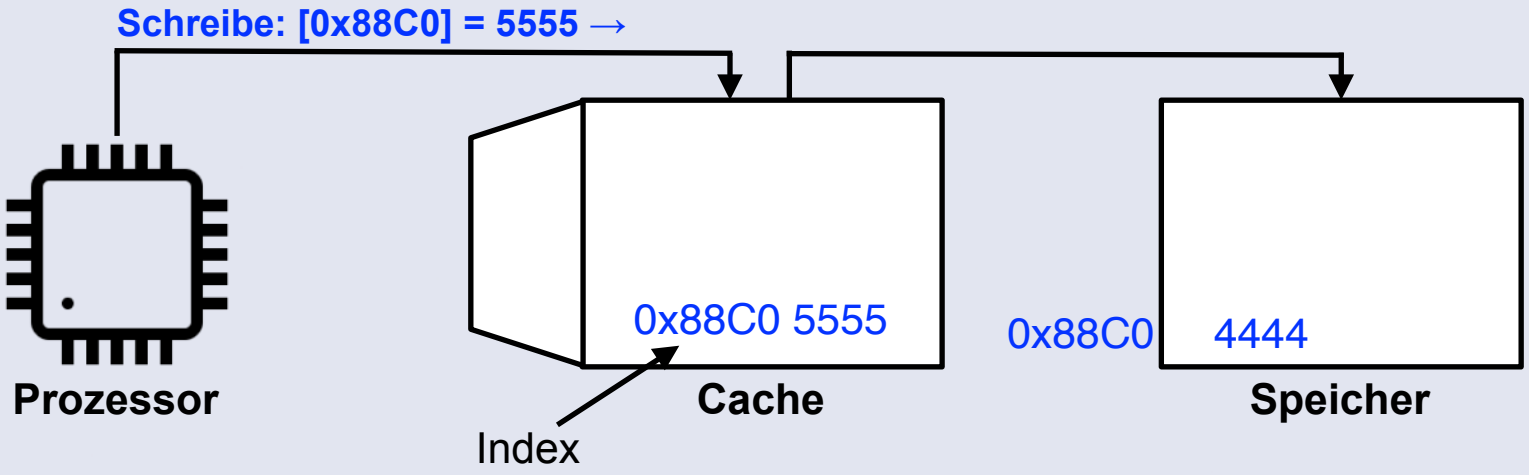
Speicher wird Engpass, es sei denn, der Anteil an Schreiboperationen ist klein oder der (Haupt-)Speicher ist nur wenig langsamer als der Cache

Schreibverfahren

- Strategien zum Rückschreiben Cache → (Haupt-) Speicher

  1. Write-Through (durchschreiben)

  - Jeder Schreibvorgang in den Cache führt zu einer unmittelbaren Aktualisierung des (Haupt-) Speichers



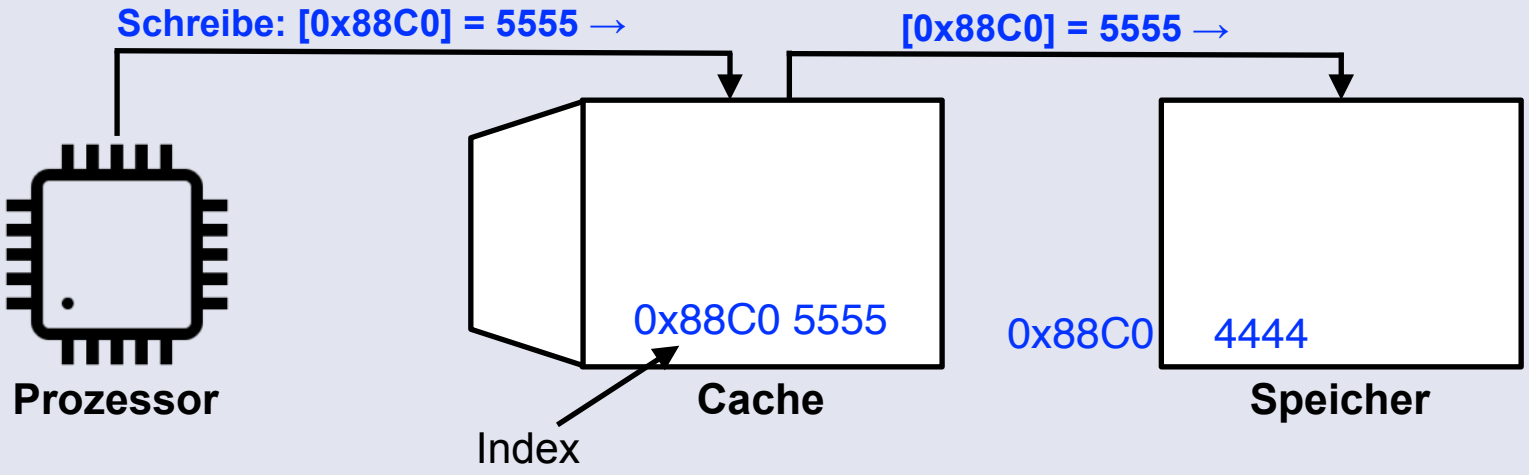
Speicher wird Engpass, es sei denn, der Anteil an Schreiboperationen ist klein oder der (Haupt-)Speicher ist nur wenig langsamer als der Cache

Schreibverfahren

- Strategien zum Rückschreiben Cache → (Haupt-) Speicher

  1. Write-Through (durchschreiben)

  - Jeder Schreibvorgang in den Cache führt zu einer unmittelbaren Aktualisierung des (Haupt-) Speichers



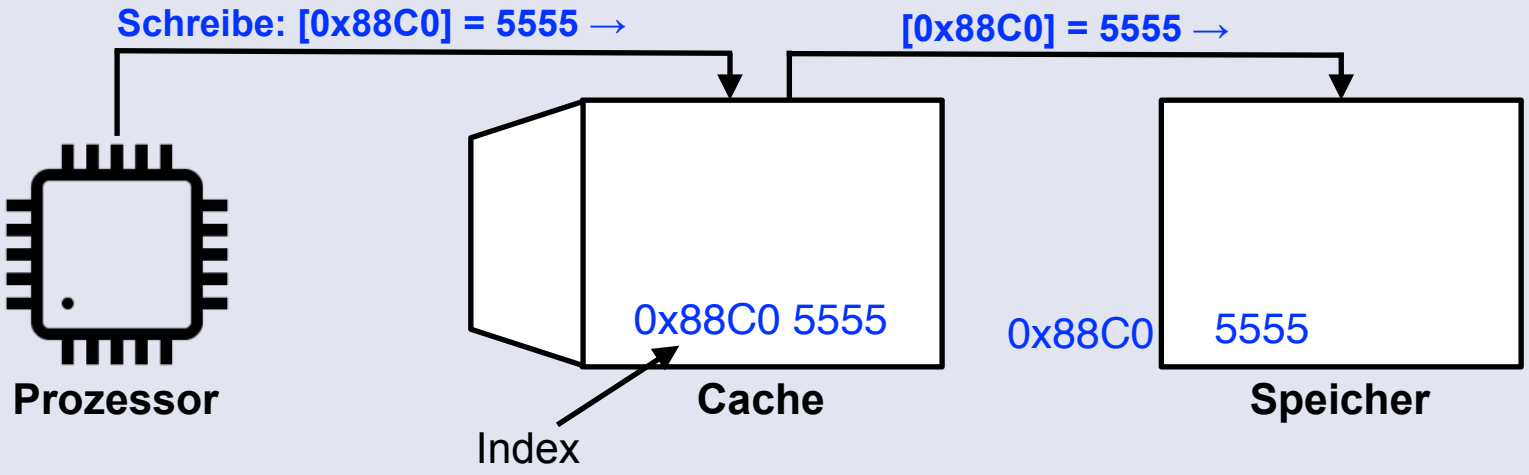
Speicher wird Engpass, es sei denn, der Anteil an Schreiboperationen ist klein oder der (Haupt-)Speicher ist nur wenig langsamer als der Cache

Schreibverfahren

- Strategien zum Rückschreiben Cache → (Haupt-) Speicher

  1. Write-Through (durchschreiben)

  - Jeder Schreibvorgang in den Cache führt zu einer unmittelbaren Aktualisierung des (Haupt-) Speichers



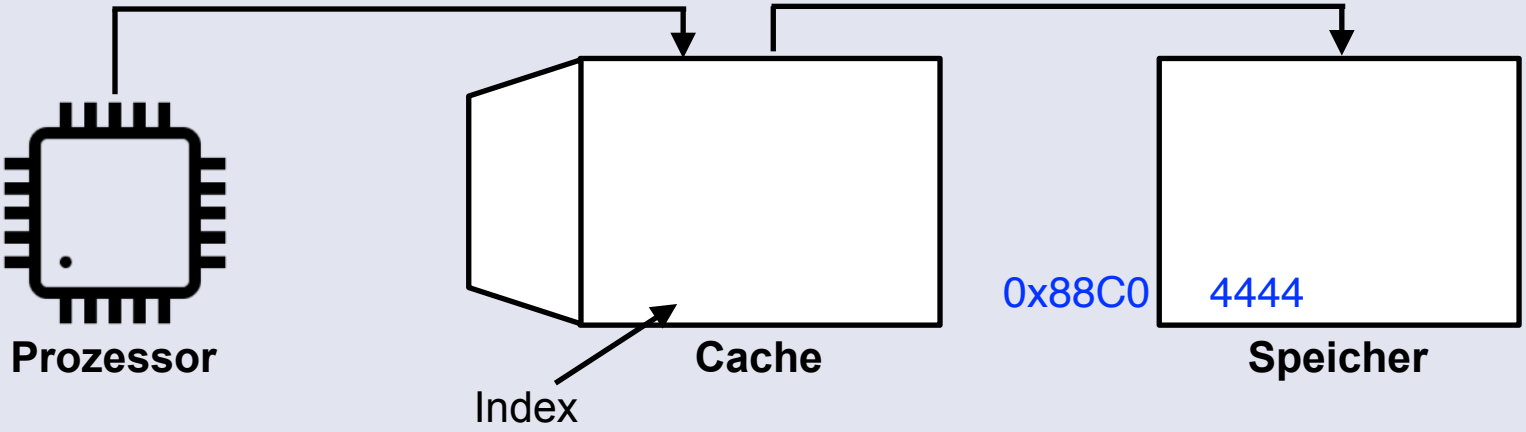
Speicher wird Engpass, es sei denn, der Anteil an Schreiboperationen ist klein oder der (Haupt-)Speicher ist nur wenig langsamer als der Cache

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird

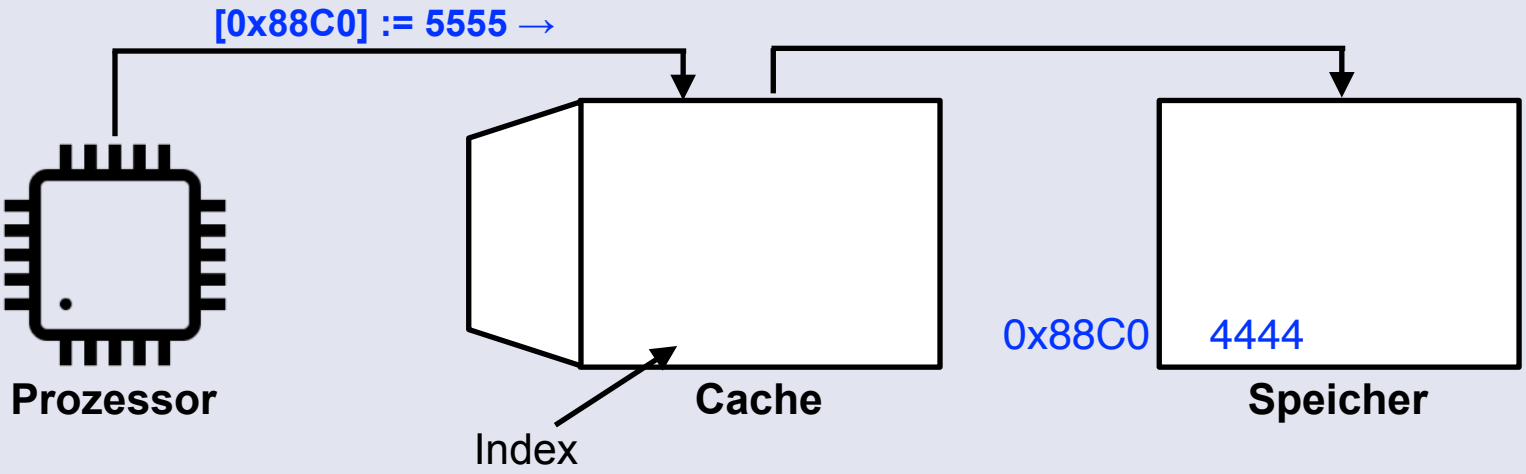


Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird

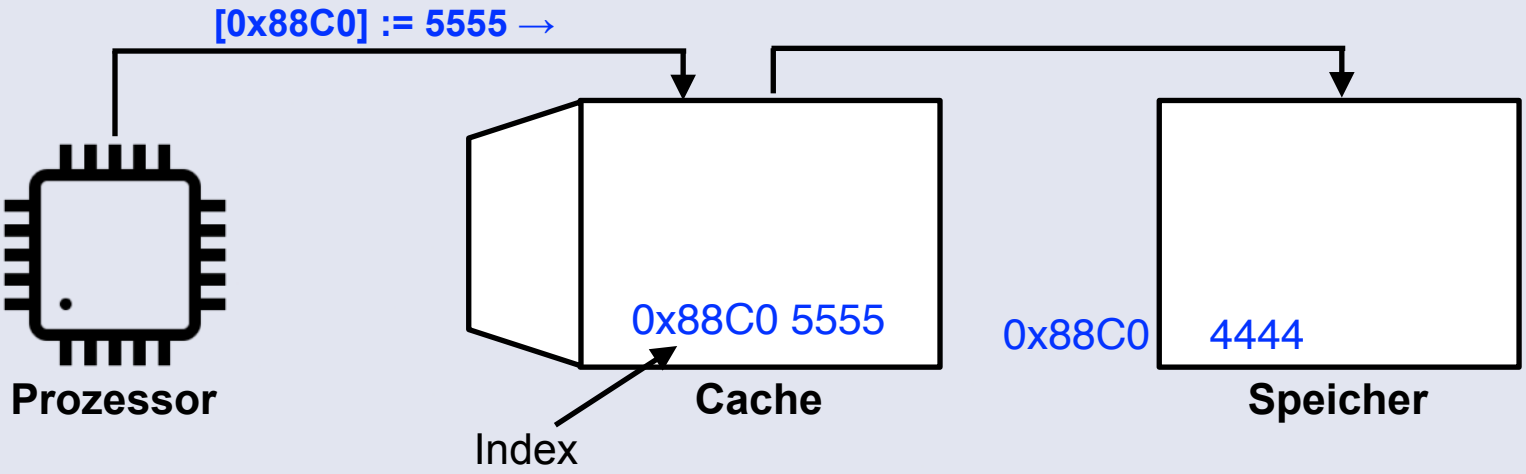


Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird

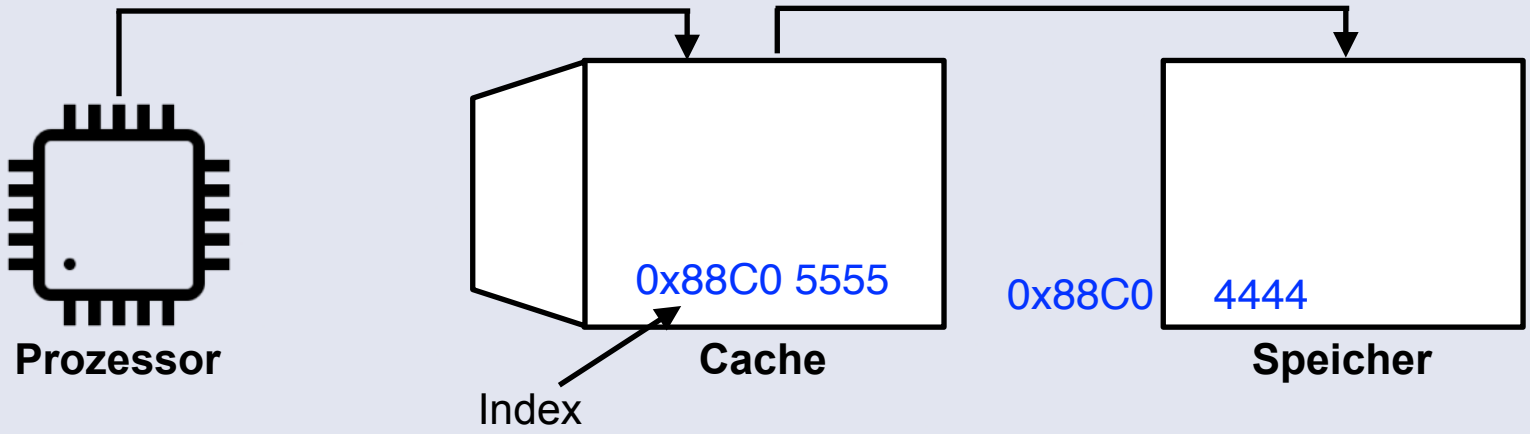


Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



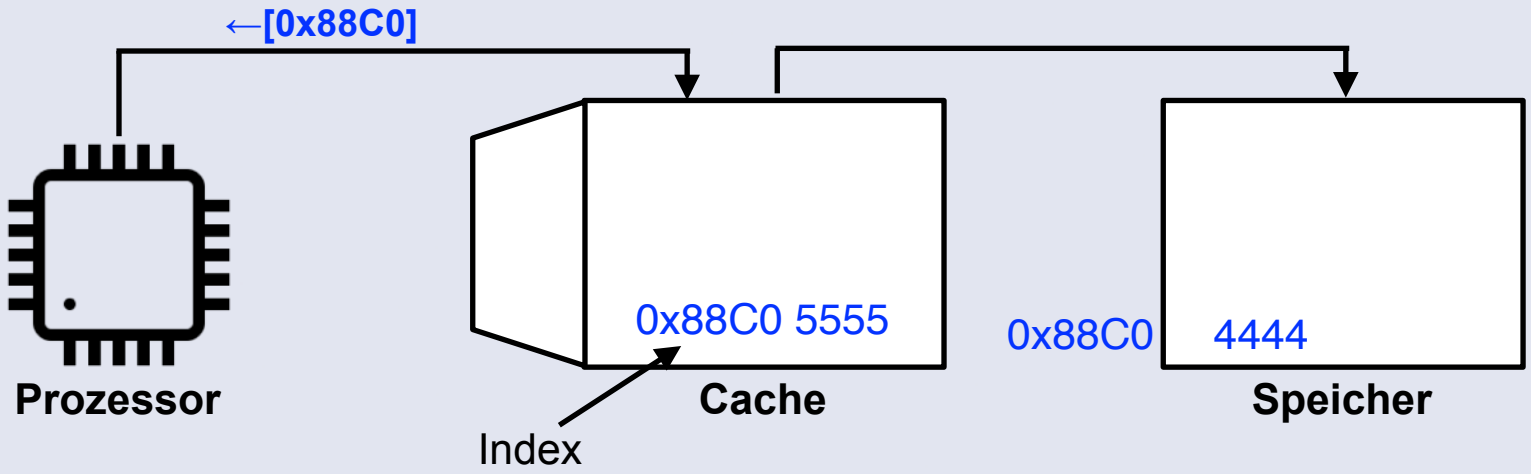
Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird

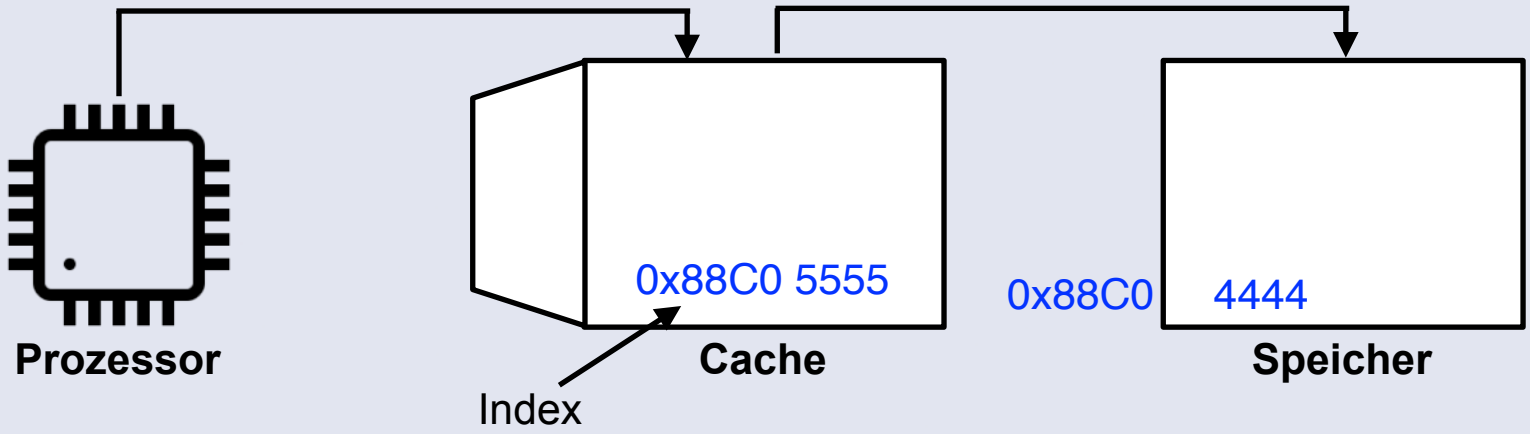


Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



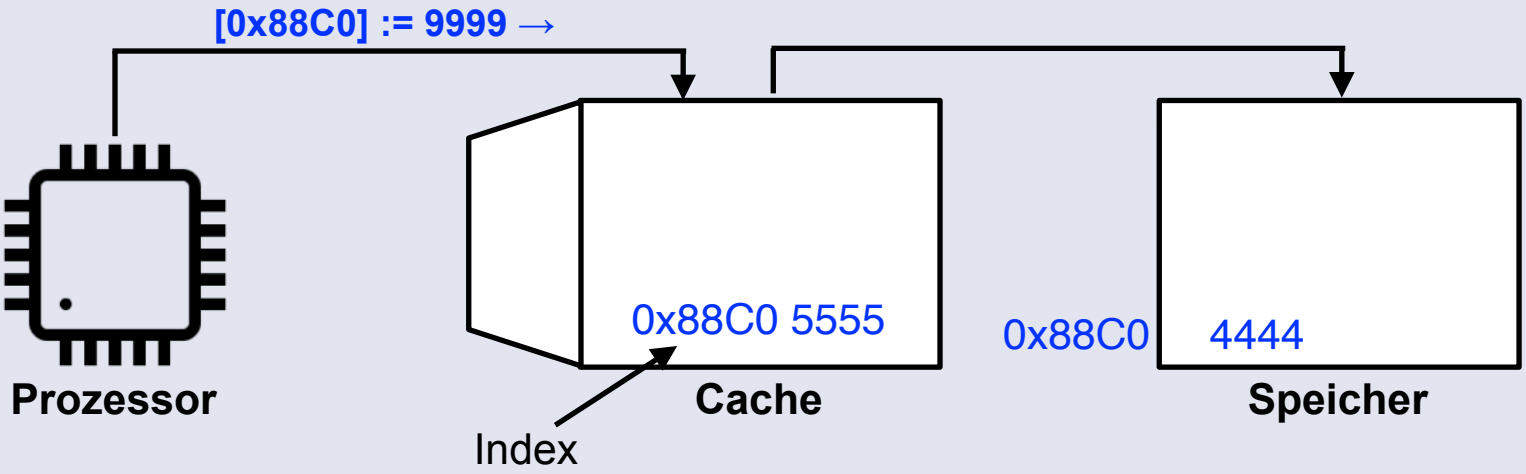
Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



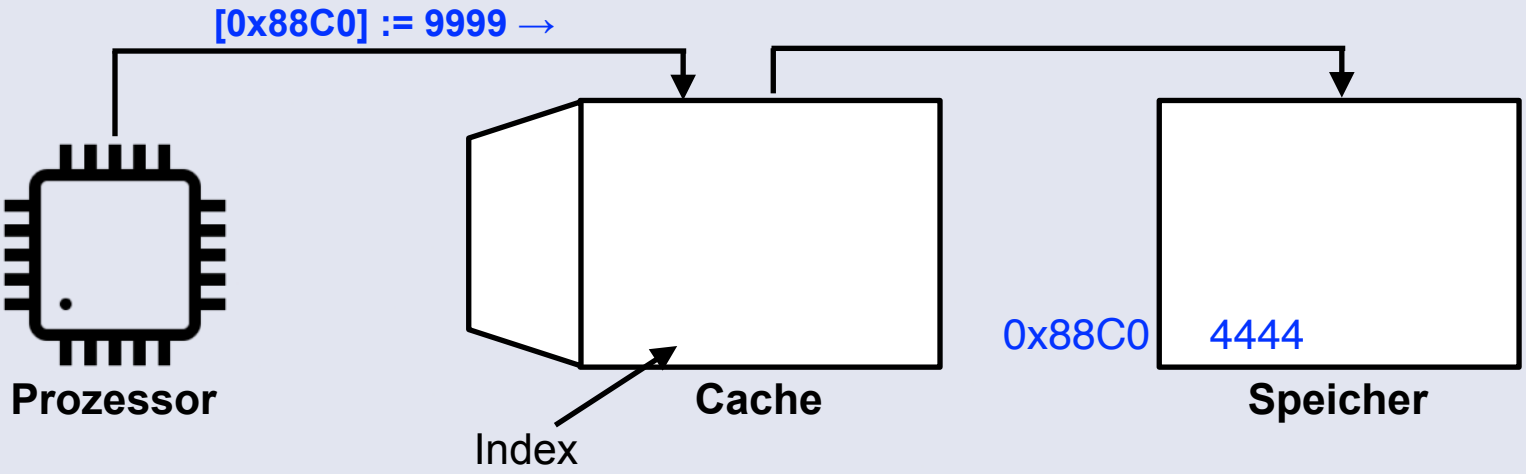
Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



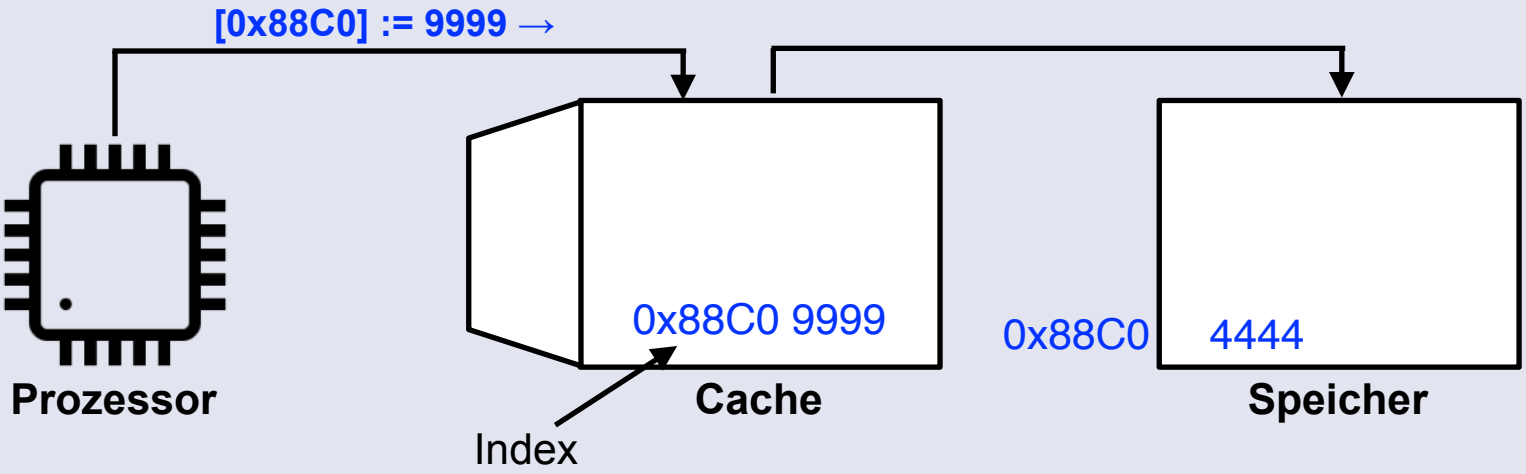
Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



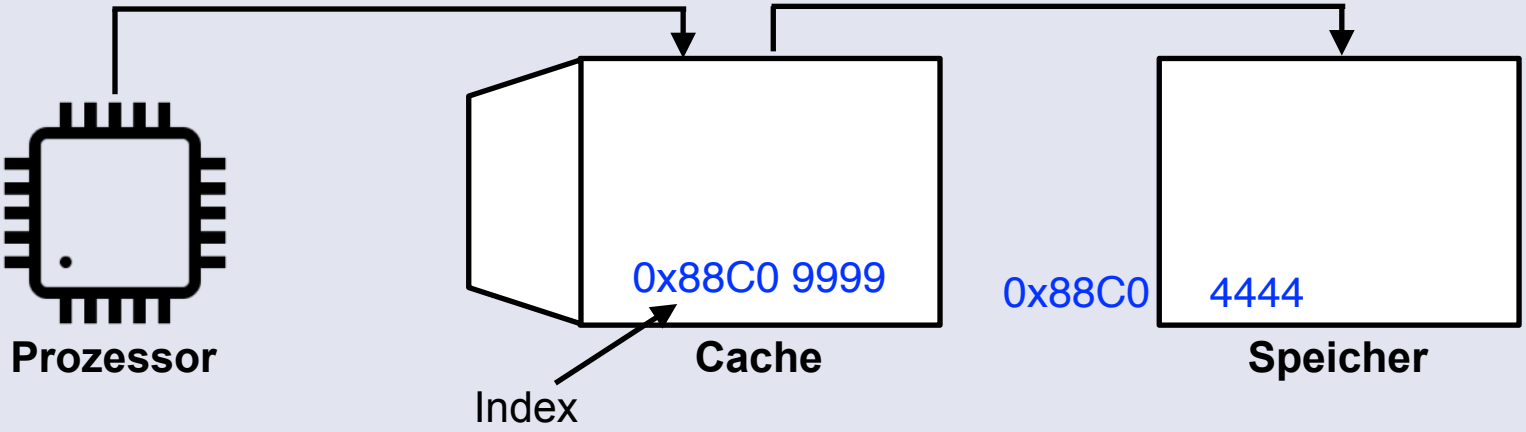
Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird

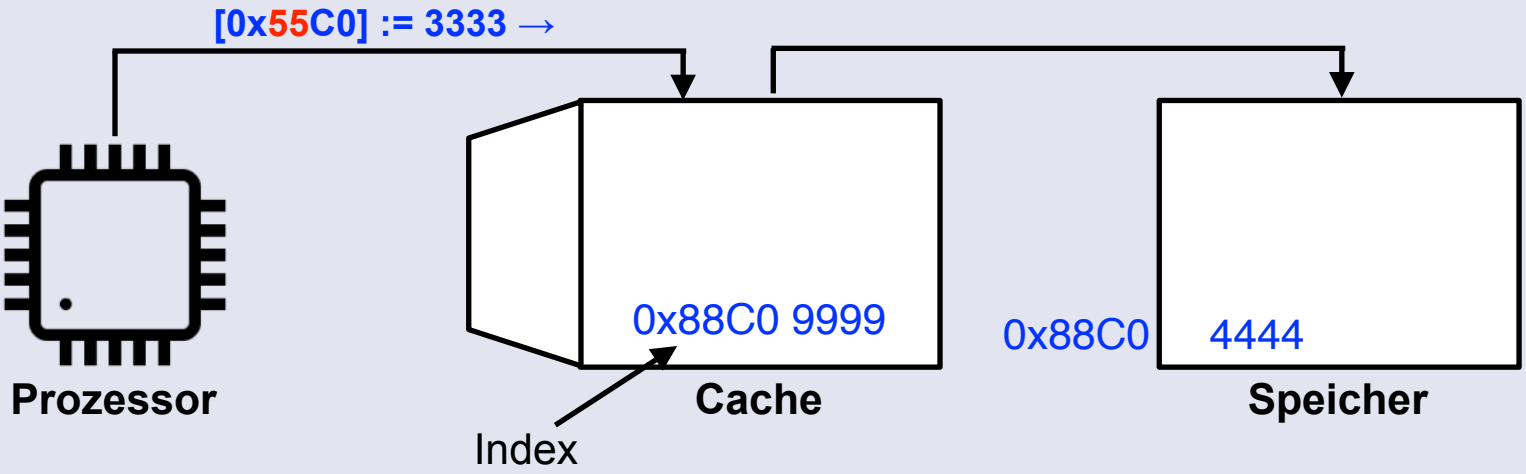


Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



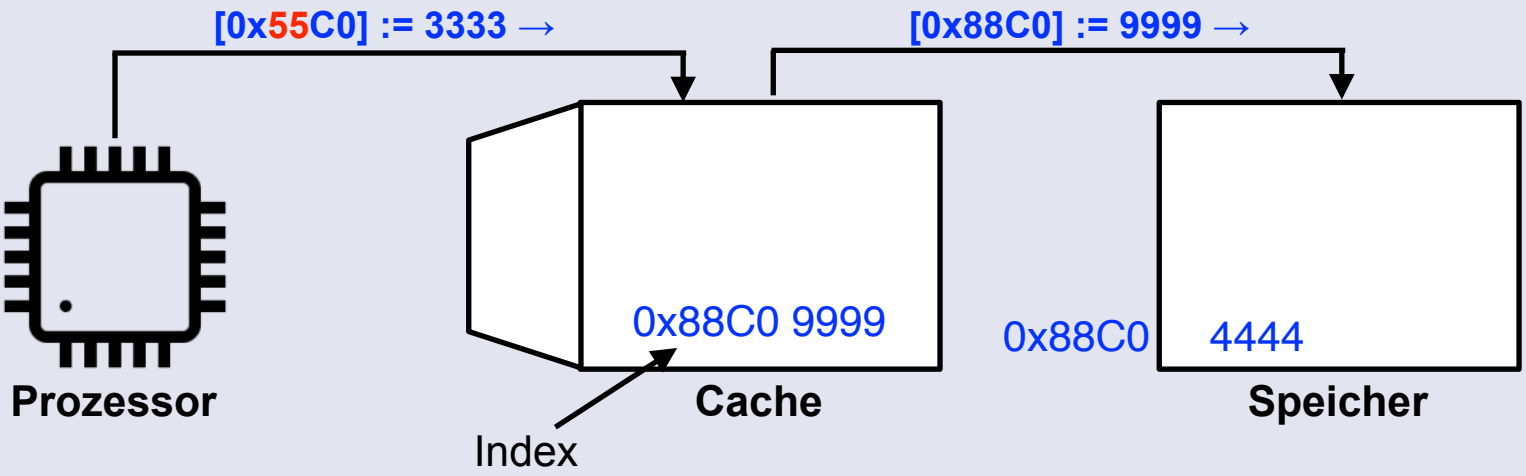
Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



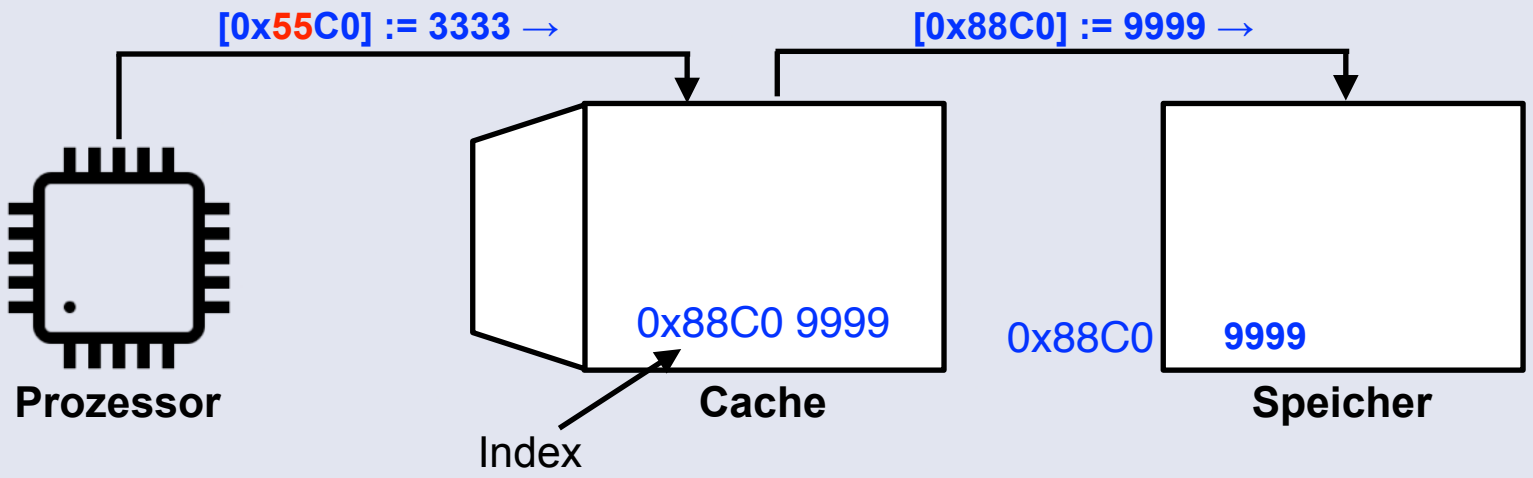
Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird

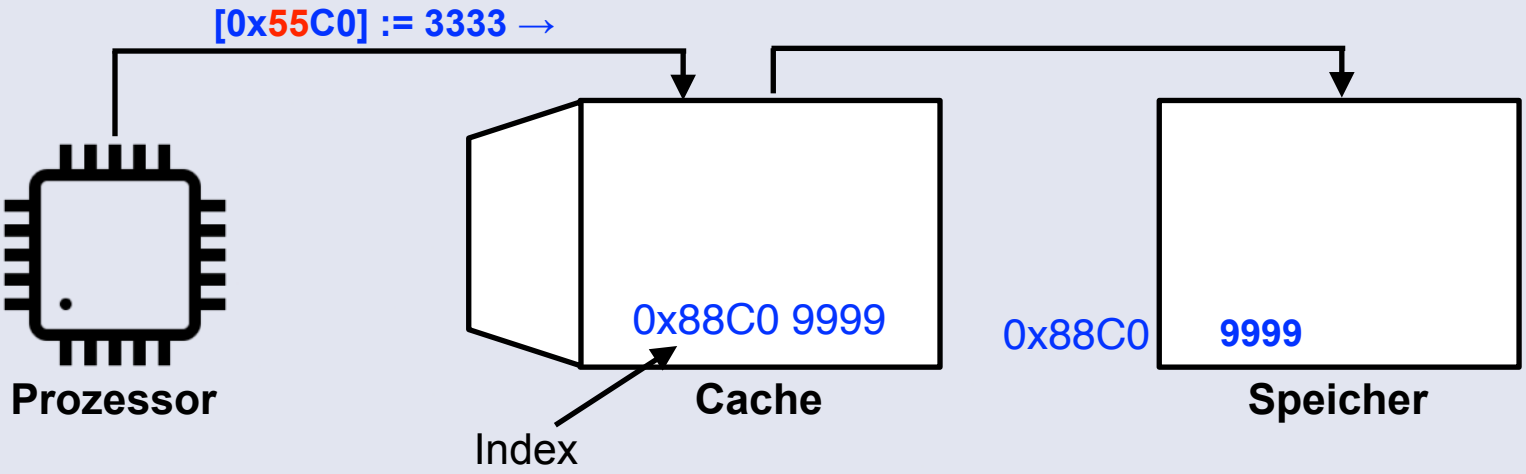


Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



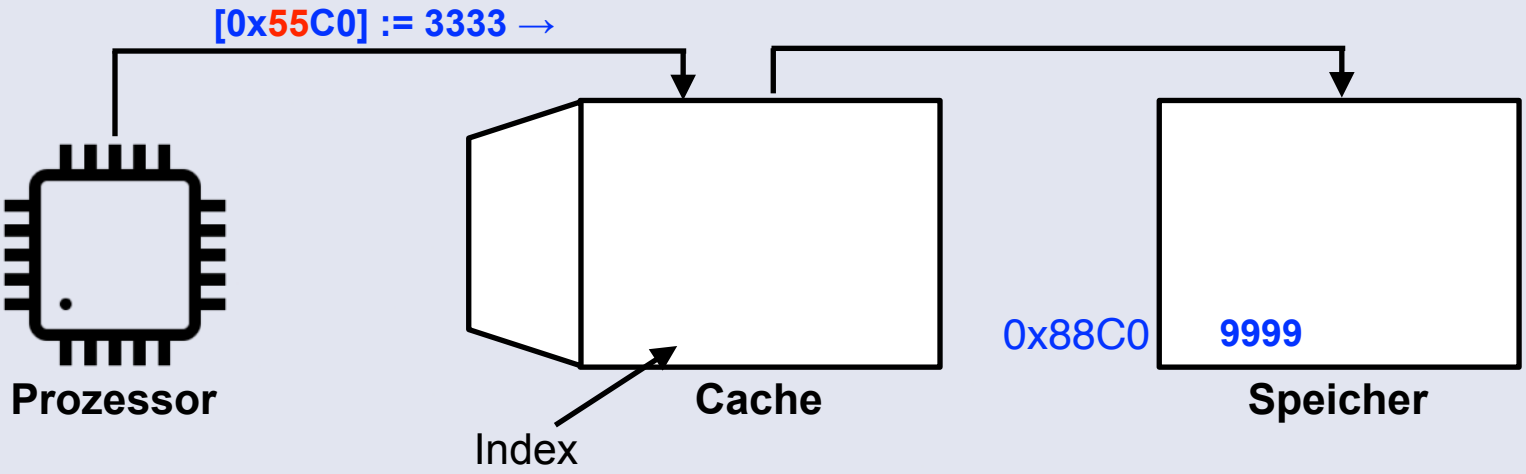
Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



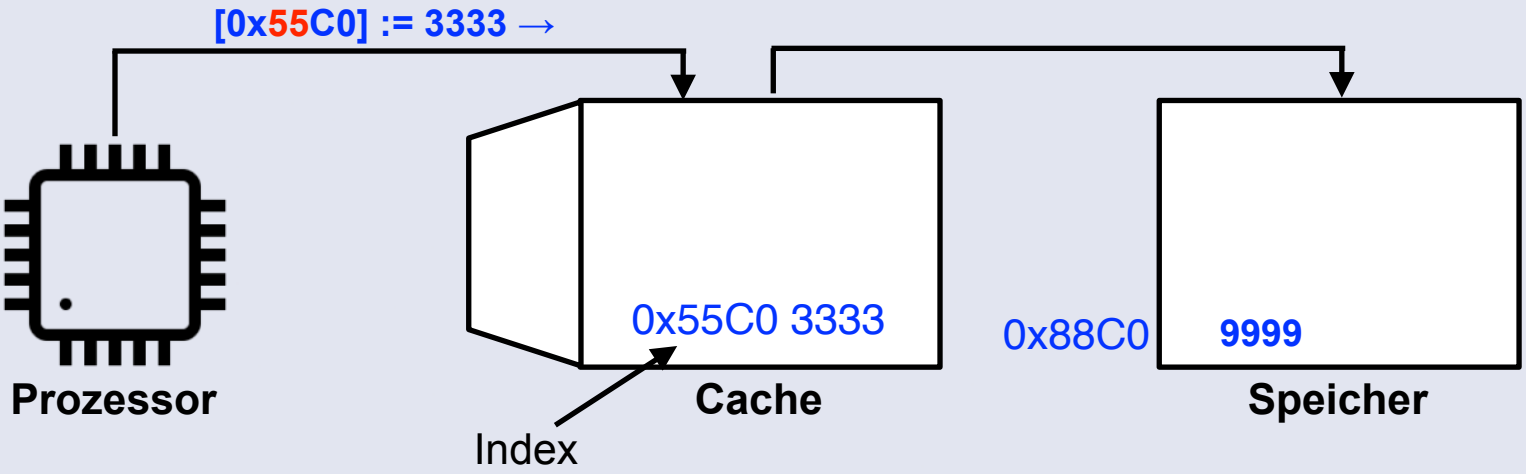
Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



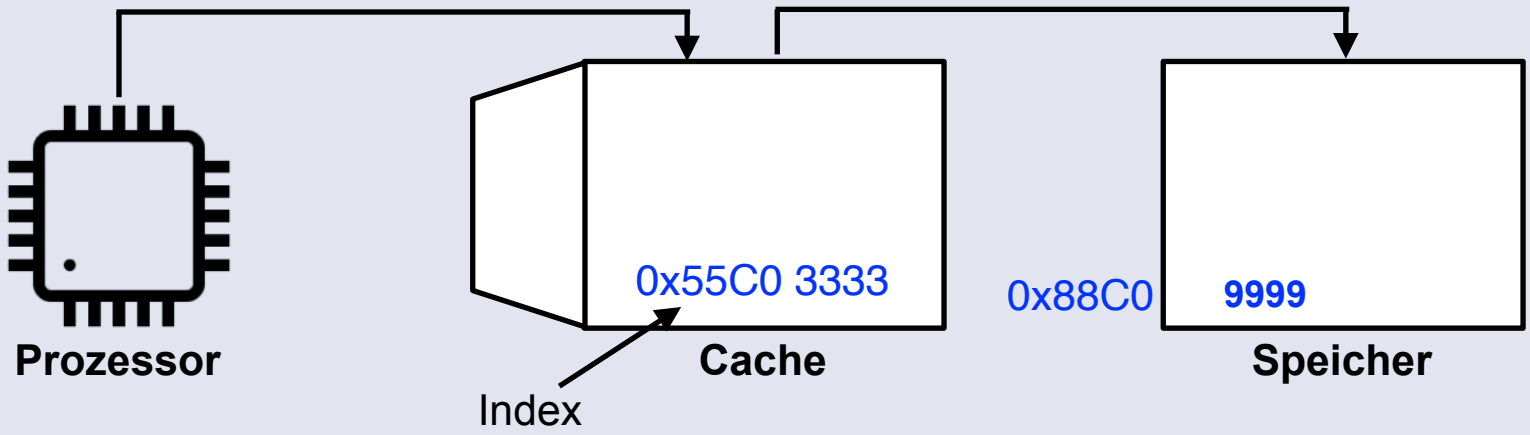
Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

Universität Bamberg

2. Copy-Back, conflicting use write back:

- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird

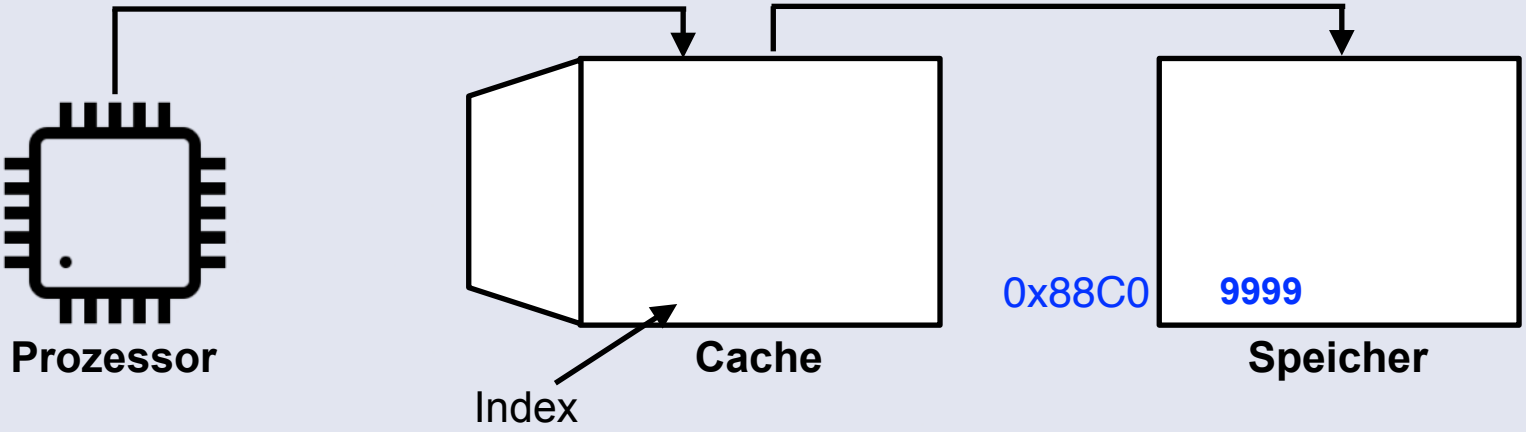


Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Schreibverfahren (2)

2. Copy-Back, conflicting use write back:

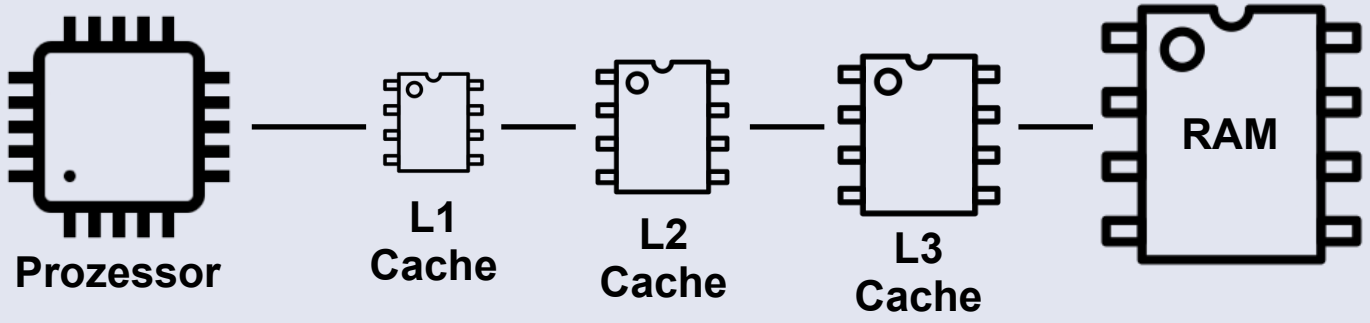
- Rückschreiben erfolgt erst, wenn die Cache-Zeile überschrieben wird



Funktioniert auch bei großen Geschwindigkeitsunterschieden zwischen Cache und Speicher. Vorkehrungen erforderlich, damit keine veralteten Werte aus dem Speicher kopiert werden

Systeme mit mehreren Caches

- Die Speicherhierarchie wird oft durch mehrere Ebenen (*levels*) von Caches erweitert
  - Level 1 (L1): nahe an der CPU, kleinster und schnellster Cache
  - Level 2 (L2): größer und langsamer als L1, zwischen L1 und L2 bzw. Hauptspeicher
    - L1- und L2-Caches sind meist pro CPU-Kern verfügbar
  - Level 3 (L3): größer und langsamer als L2, zwischen L2 und Hauptspeicher
    - Nicht bei allen Architekturen realisiert
    - L3-Caches sind oft geteilt zwischen allen Prozessorkernen



Beispiele für Cachegrößen

Hersteller	SoC	CPU	L1	L2	L3	L2+L3
Apple	M4 Max	12P +4E	192/128 kB	16 MB + 16 MB + 4 MB	–	36 MB
Apple	M4 Pro	10P +4E	128/64 kB	16 MB + 16 MB + 4 MB	–	36 MB
Apple	M4	4P+6E	128/64 kB	16 MB + 4 MB	–	20 MB
AMD	Ryzen 9950X	16	80 kB	8 MB + 8 MB	32 MB + 32 MB	80 MB
AMD	Ryzen 9900X	12	80 kB	6 MB + 6 MB	32 MB + 32 MB	76 MB
AMD	Ryzen 9 HX 370	4 + 8C	80 kB	4 MB + 8 MB	16 MB + 8 MB	36 MB
Intel	Core Ultra 9 285K	8P + 16E	192 kB	24 MB + 16 MB	36 MB	76 MB
Intel	Lunar Lake	4P+4E	192 kB	10 MB + 4 MB	12 MB	26 MB
Qualcomm	Snapdragon X Elite	12P	128 kB	12 MB + 12 MB + 12 MB	–	36 MB
Apple	M3 Max	12P +4E	192/128 kB	16 MB + 16 MB + 4 MB	–	36 MB

Austauschverfahren

Was geschieht, wenn der Cache voll ist (alle Einträge belegt)?

Überschreiben ("Verdrängen") von Einträgen in Caches:

1. **Random- bzw. Zufallsverfahren**
   - Ein zufällig ausgewählter Eintrag wird überschrieben
2. **NRU, not recently used**
   - Einteilung von Einträgen in 4 Klassen:
     A. **Nicht benutzte, nicht modifizierte Einträge**
     B. **Nicht benutzte, modifizierte Seiten**
        Benutzt-Kennzeichnung wird nach einiger Zeit zurückgesetzt.
     C. **Benutzte, nicht modifizierte Seiten**
     D. **Benutzte, modifizierte Seiten**

   Überschrieben wird ein zufällig ausgewählter Eintrag der niedrigsten Klasse, die nicht-leer ist
3. **LRU=least recently used**
   - Überschreiben des Eintrags, der am längsten nicht benutzt wurde

Effizienz von Caches

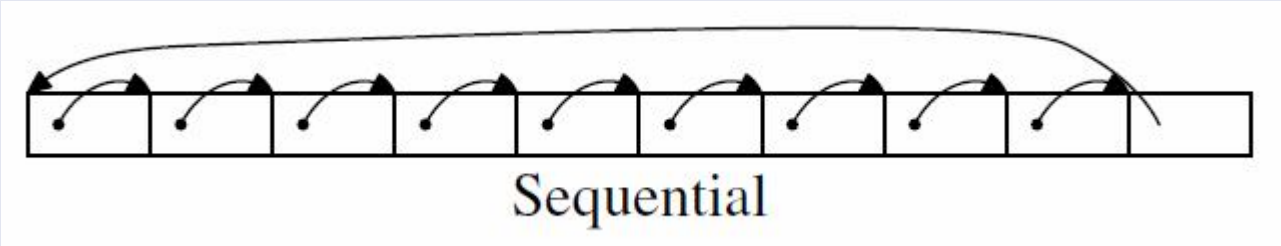
- Annahmen:
  - Dauer Hauptspeicherzugriff: 200 Zyklen
  - Dauer Cache-Zugriff: 15 Zyklen

- Beispiel: 100 x Zugriff auf 100 Datenelemente:
  - 200 * 100 * 100 = **2.000.000 Zyklen**,
    wenn Daten im Hauptspeicher
  - (100 * 200) Zyklen + 99 * (100 * 15) Zyklen = **168.500 Zyklen**,
    wenn Daten im Cache
    - Erster Zugriff auf Hauptspeicher: 100 * 200 Zyklen
    - Folgende 99 Zugriffe auf den Cache: je 100 * 15 Zyklen

- Performance-**Verbesserung** um **91,5%**, wenn alle Elemente in den Cache passen!

Abhängigkeit von Cache auf die Laufzeit

- Untersuchung der Laufzeit beim Durchlauf durch eine im Array abgelegte verkettete Liste mit Einträgen von NPAD*8 Bytes



- Evaluationsplattform: Pentium P4
  - 16 kB L1 Daten-Cache mit 4 Zyklen/Zugriff
  - 1 MB L2 Cache mit 14 Zyklen/Zugriff
  - Hauptspeicher mit 200 Zyklen/Zugriff
  - Details in [3]

Zyklen/Zugriff als Funktion der Listengröße

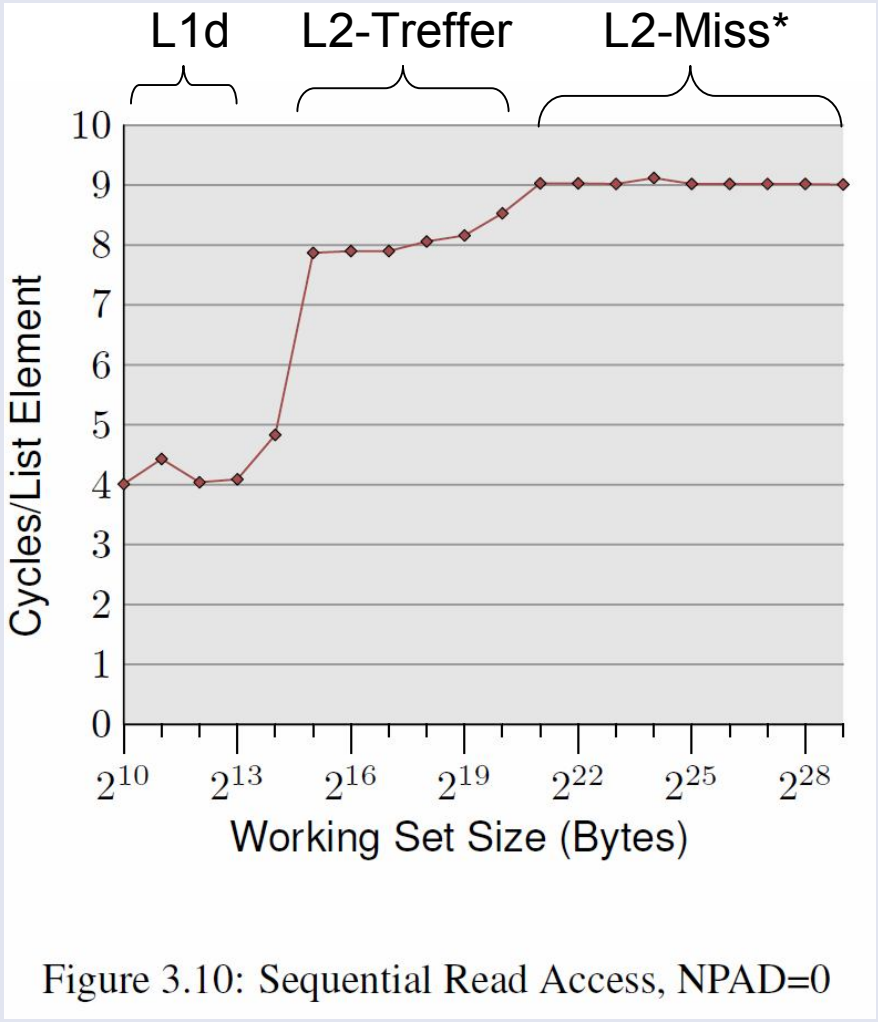
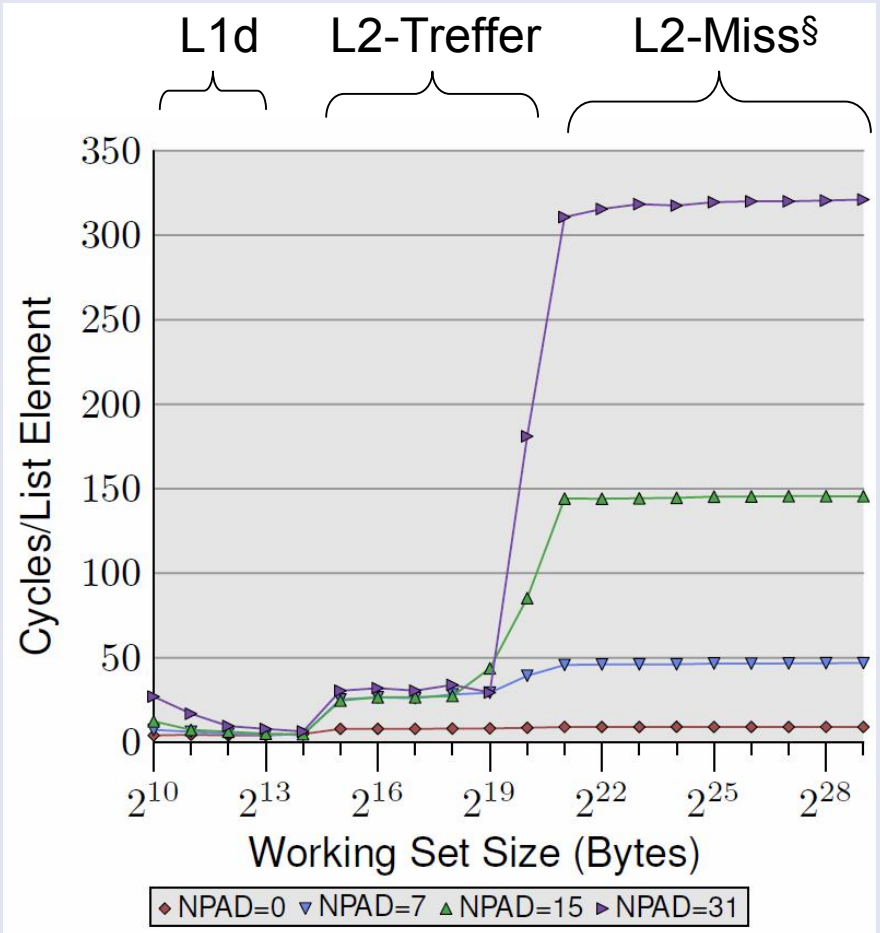


Figure 3.10: Sequential Read Access, NPAD=0



* Funktionierendes Prefetching

§ Prefetching schlägt fehl

Einfluß der Cachegröße

- Größere Caches verschieben die Stufen

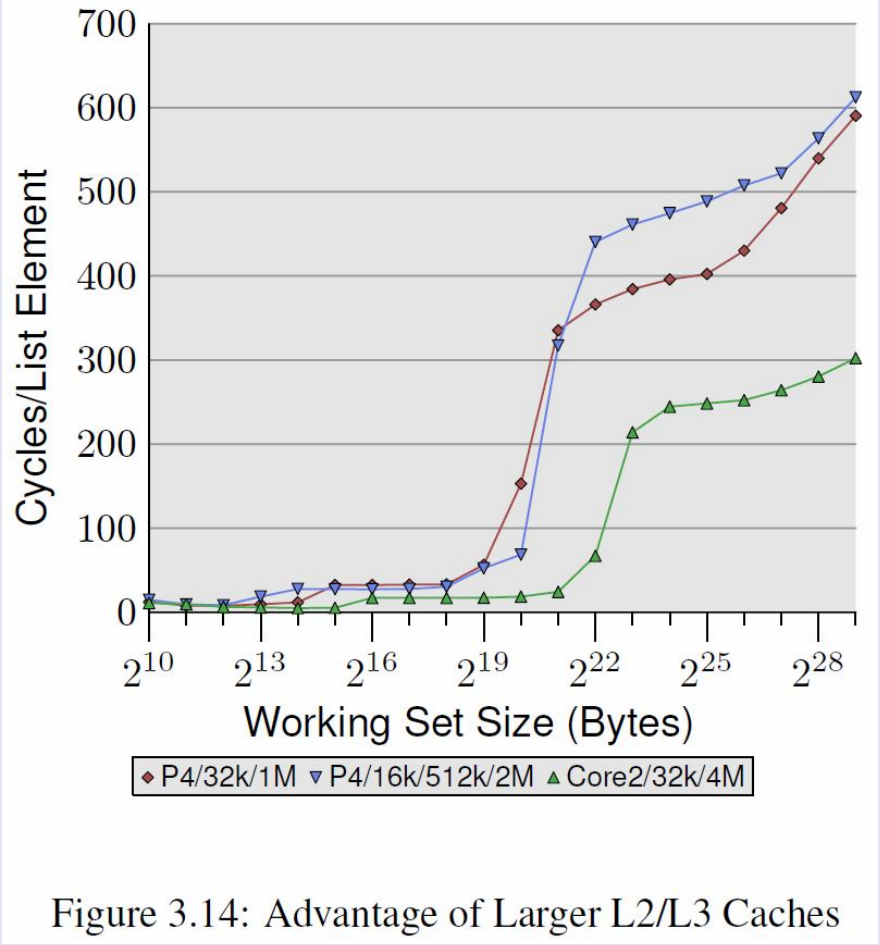


Figure 3.14: Advantage of Larger L2/L3 Caches

Referenzen

[1] Frank Slomka, Michael Glaß
Grundlagen der Rechnerarchitektur
Von der Schaltung zum Prozessor

[2] John L. Hennessy, David A. Patterson
Computer Architecture: A Quantitative Approach
Morgan Kaufmann, 7th edition 2025
ISBN-13: 978-0443154065

[3] Ulrich Drepper
What every programmer should know about memory
RedHat 2007
https://akkadia.org/drepper/cpumemory.pdf

[4] Peter J. Denning
The working set model for program behavior
In: Communications of the ACM Volume 11, Issue 5, 1968, S. 323–333