

Speicherhierarchie

Speicherhierarchie zum Ausgleich der unterschiedlichen Zugriffszeiten der CPU und des Hauptspeichers.

2 Strategien:

- ❑ **Cache-Speicher:**

Kurze Zugriffszeiten → Beschleunigung des Prozessorzugriffs

- ❑ **Virtueller Speicher:**

Vergrößerung des tatsächlich vorhandenen Hauptspeichers (z. B. bei gleichzeitiger Bearbeitung mehrerer Prozesse)



Speicherhierarchie

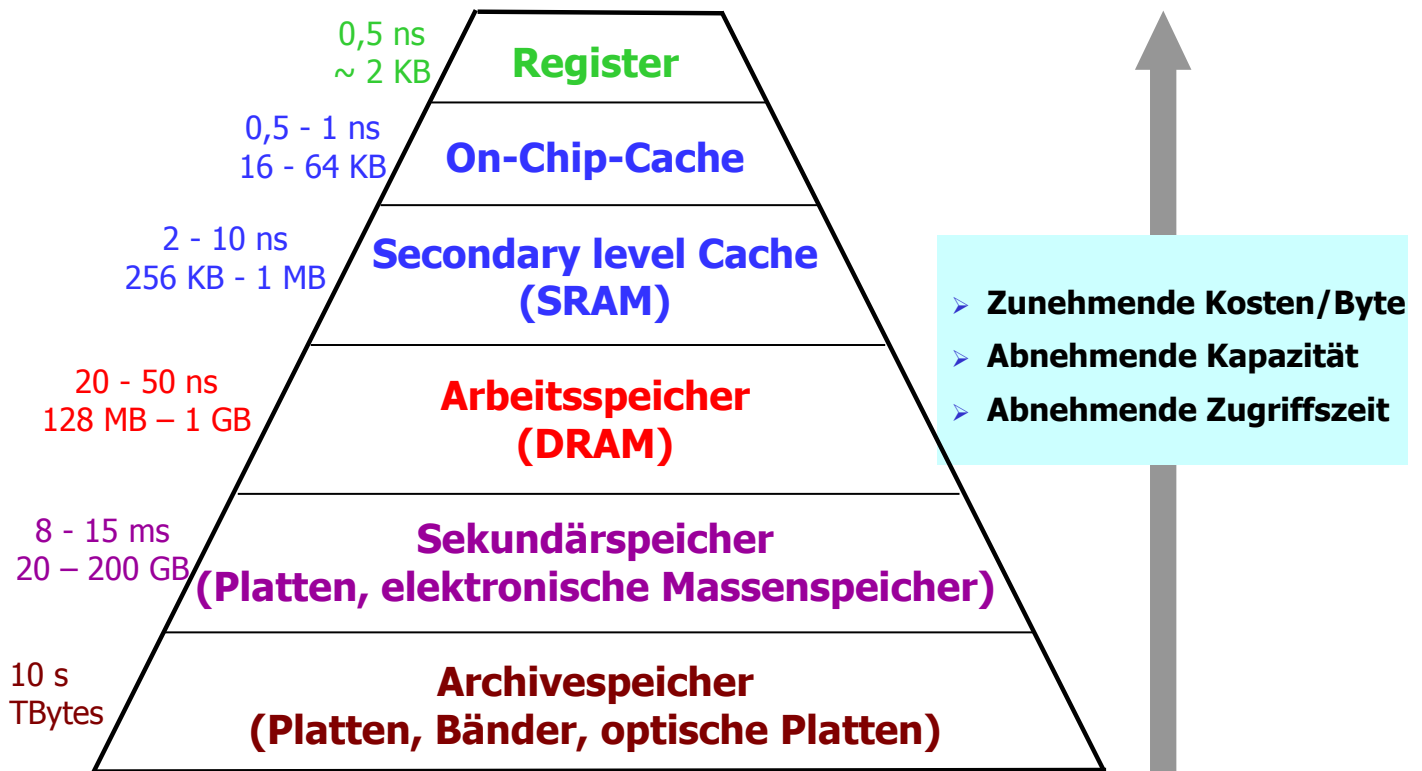
Ein technologisch einheitlicher Speicher mit **kurzer Zugriffszeit** und **großer Kapazität** ist aus **Kostengründen** i. A. nicht realisierbar

Lösung:

Schichtenweise Anordnung verschiedener Speicher und Verschiebung der Information zwischen den Schichten (Speicherhierarchie)



Speicherhierarchie



Speicherhierarchie

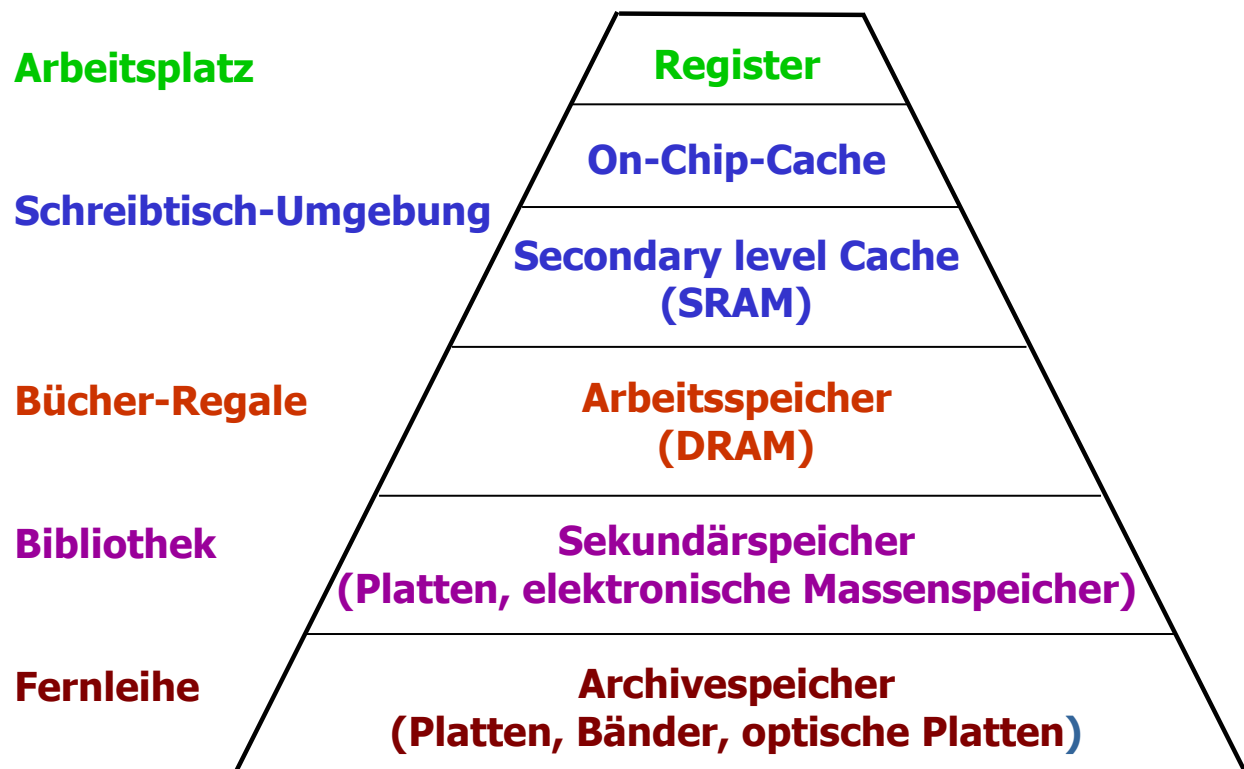
Wirkung: wie ein großer und schneller Speicher, wenn

- Lokalitätsverhalten der Programmverarbeitung,
- Umlagerung der Information rechtzeitig (Umlagerungsstrategien),
- Inhomogenität des Speichersystems für Benutzer nicht sichtbar ist (Virtueller Speicher)

Leistungsfähigkeit der Hierarchie ist bestimmt durch die Eigenschaften der Speichertechnologien (Zugriffsart, Zugriffszeiten, ...), Adressierung der Speicherplätze und Organisation des Betriebs



Speicherhierarchie



Cache-Speicher

Problem:

die Buszykluszeit moderner Prozessoren ist erheblich kürzer als die Zykluszeit preiswerter, großer DRAM-Bausteine

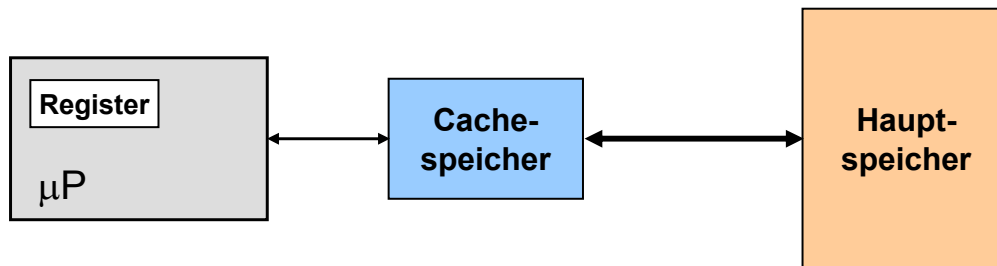
- dies zwingt zum Einfügen von Wartezyklen.
SRAM-Bausteine hingegen, die ohne Wartezyklen betrieben werden können, sind jedoch klein, teuer und leistungsintensiv.
- nur kleine Speicher können so aufgebaut werden.



Cache-Speicher

Lösung des Problems:

zwischen den Prozessor und den relativ langsamen, aber billigen Hauptspeicher aus DRAM-Bausteinen legt man einen kleinen, schnellen Speicher aus SRAM-Bausteinen, den sogenannten **Cache-Speicher**.

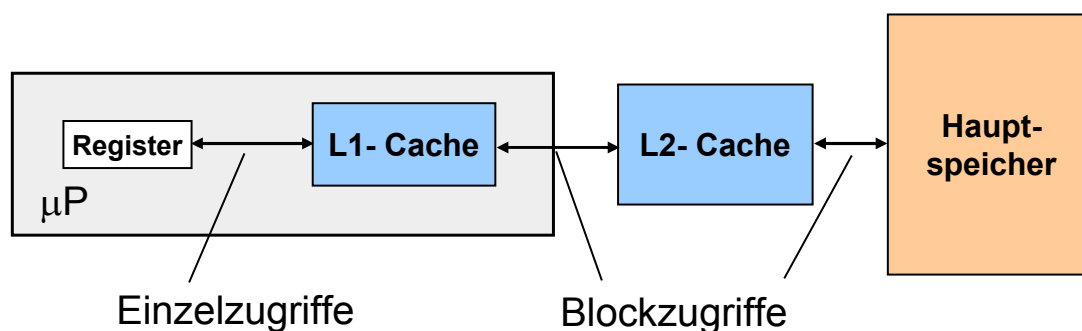


Auf den Cache-Speicher soll der Prozessor fast so schnell wie auf seine Register zugreifen können.



Cache-Speicher

- **On-Chip-Cache:** integriert auf dem Prozessorchip
 - Sehr kurze Zugriffszeiten (wie die der prozessorinternen Register)
 - Aus technologischen Gründen begrenzte Kapazität
- **Off-Chip-Cache:** prozessorextern (SRAM-Speicher)



Cache-Speicher

Anwendungsbeispiele:

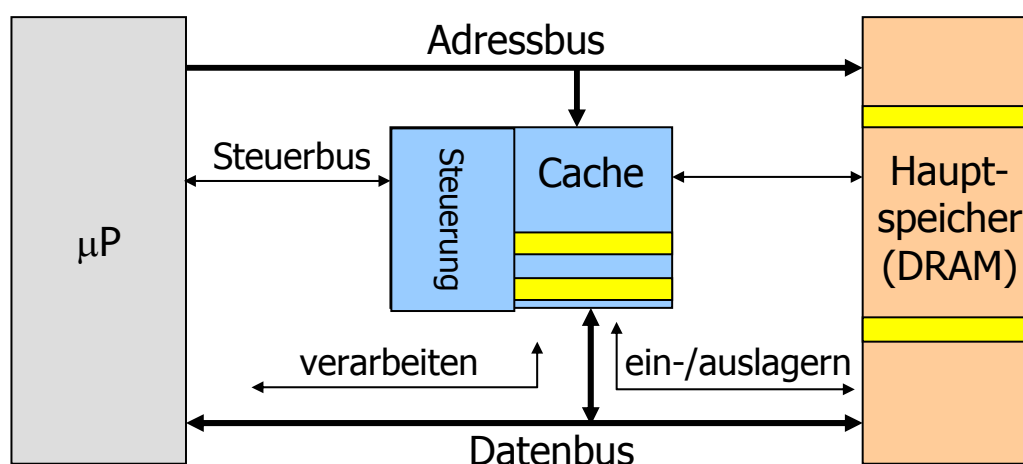
- Verbesserung der Zugriffszeit des Hauptspeichers eines Prozessors durch einen Cache zur Vermeidung von Wartezyklen (**CPU-Cache**, **Befehls- und Daten-Cache**).
- Verbesserung der Zugriffszeit von Plattenspeichern durch einen Cache (**Plattencache**)

Hier soll im wesentlichen der erste Fall näher betrachtet werden



Cache-Speicher

Unter einem **CPU-Cache-Speicher** versteht man einen kleinen, schnellen Pufferspeicher, in dem Kopien derjenigen Teile des Hauptspeichers bereitgehalten werden, auf die aller Wahrscheinlichkeit nach von der CPU als nächstes zugegriffen wird.



Cache-Speicher

Ein CPU-Cache-Speicher bezieht seine Effizienz im wesentlichen aus der **Lokalitätseigenschaft** von Programmen (*locality of reference*), d. h. es werden bestimmte Speicherzellen bevorzugt und wiederholt angesprochen (z. B. Programmschleifen)

- ❑ **Zeitliche Lokalität:** Die Information, die in naher Zukunft angesprochen wird, ist mit großer Wahrscheinlichkeit schon früher einmal angesprochen worden (Schleifen).
- ❑ **Örtliche Lokalität:** Ein zukünftiger Zugriff wird mit großer Wahrscheinlichkeit in der Nähe des bisherigen Zugriffs liegen.



Cache-Speicher

Die **Cachespeicherverwaltung** (Steuerung) sorgt dafür, dass die am häufigst benötigten Daten sich im Cache befinden

→ die CPU kann mit hoher Wahrscheinlichkeit das nächste Datum aus dem schnellen Cache und nicht aus dem langsamen Arbeitsspeicher holen.

Dies wird erreicht, indem automatisch durch eine **Hardware-Steuerung (Cache-Controller)** alle Daten in den Cache kopiert werden, auf die der Prozessor zugreift.

Weniger häufig benötigte Daten werden nach verschiedenen Strategien aus dem Cache verdrängt.



Funktionsweise eines Caches

Lesezugriffe:

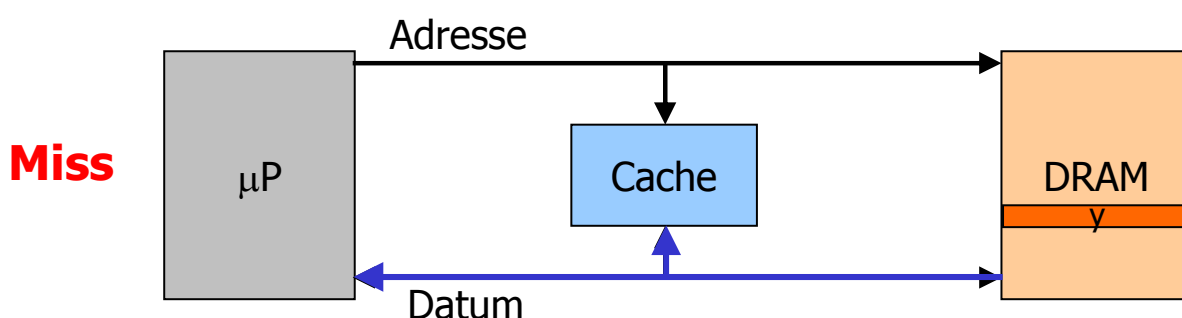
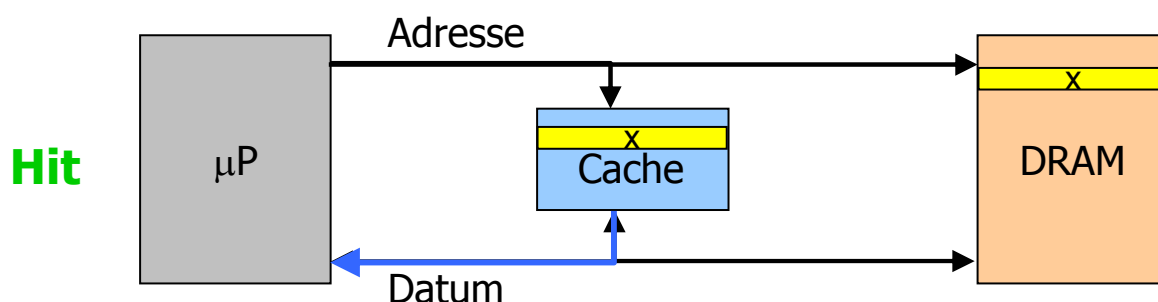
Vor jedem Lesezugriff prüft der μP , ob das Datum im Cache steht

Wenn ja: **Treffer (Cache Hit)**
das Datum kann ohne Wartezyklen aus dem Cache entnommen werden.

Wenn nein: **kein Treffer (Cache Miss)**
das Datum wird mit Wartezyklen aus dem Arbeitsspeicher gelesen und gleichzeitig in den Cache eingefügt.



Funktionsweise eines Caches



Begriffe

Die **Hit-Rate** bezeichnet die Trefferquote im Cache:

$$\text{Hit-Rate} = \text{Anzahl Treffer} / \text{Anzahl Zugriffe}$$

Die **mittlere Zugriffszeit** berechnet sich annähernd wie folgt:

$$t_{\text{Access}} = (\text{Hit-Rate}) * t_{\text{Hit}} + (1 - \text{Hit-Rate}) * t_{\text{Miss}}$$

mit t_{Hit} : Zugriffszeit des Caches
 t_{Miss} : Zugriffszeit ohne den Cache



Funktionsweise eines Caches

Schreibzugriffe:

Liegt beim Schreiben ein Cache-Miss vor, wird das Datum sowohl in den Arbeitsspeicher als auch in den Cache geschrieben.

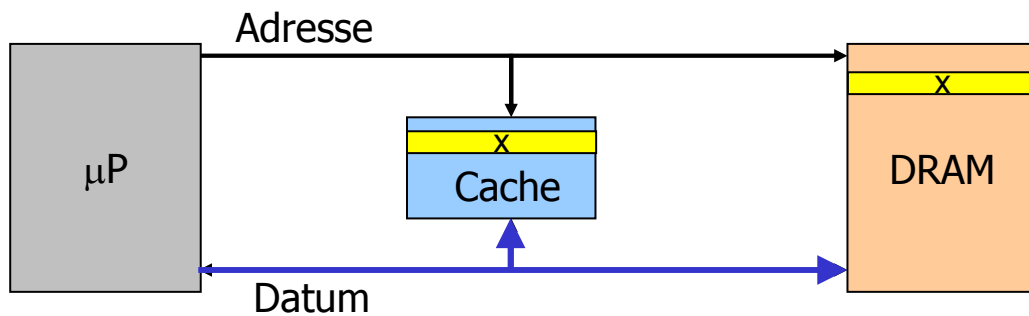
Liegt beim Schreiben jedoch ein Cache-Hit vor, d. h. ein im Cache stehendes Datum wird durch den Prozessor verändert, so existieren verschiedene Organisationsformen:



Schreibzugriffe

➤ **Durchschreibverfahren:** (*write through policy*)

Ein Datum wird von der CPU immer gleichzeitig in den Cache- und in den Arbeitsspeicher geschrieben.



Schreibzugriffe

Vorteil:

garantierte Konsistenz zwischen Cache- und Arbeitsspeicher.

Nachteil:

Schreibzugriffe benötigen immer die langsame Zykluszeit des Hauptspeichers und belasten den Systembus.



Schreibzugriffe

➤ **Gepuffertes Durchschreibverfahren:** (*buffered write through policy*)

Variante des Durchschreibverfahrens

Zur Milderung des Nachteils beim Durchschreibverfahren wird ein kleiner Schreib-Puffer verwendet, der die zu schreibenden Daten temporär aufnimmt.

Diese Daten werden dann automatisch vom Cache-Controller in den Hauptspeicher übertragen, während der Prozessor parallel dazu mit weiteren Operationen fortfährt.



Schreibzugriffe

➤ **Rückschreib-Verfahren:** (*write back policy*)

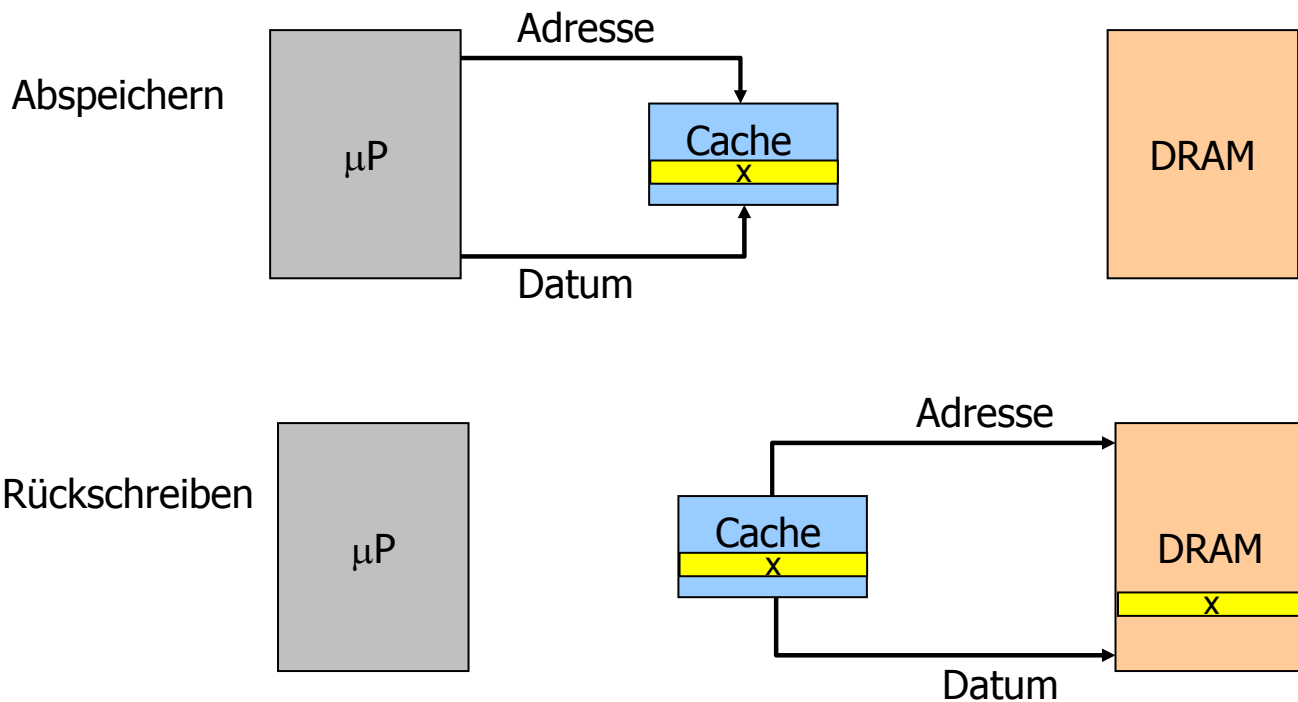
Ein Datum wird von der CPU nur in den Cachespeicher geschrieben und durch ein spezielles Bit (*altered bit, modified bit, dirty bit*) gekennzeichnet.

Der Arbeitsspeicher wird nur geändert, wenn ein so gekennzeichnetes Datum aus dem Cache verdrängt wird



Schreibzugriffe

Prinzip des Rückschreibverfahren:



Schreibzugriffe

Vorteil:

auch Schreibzugriffe können mit der schnellen Cache-Zykluszeit abgewickelt werden

Nachteil:

Konsistenzprobleme zwischen Cache- und Hauptspeicherspeicher.

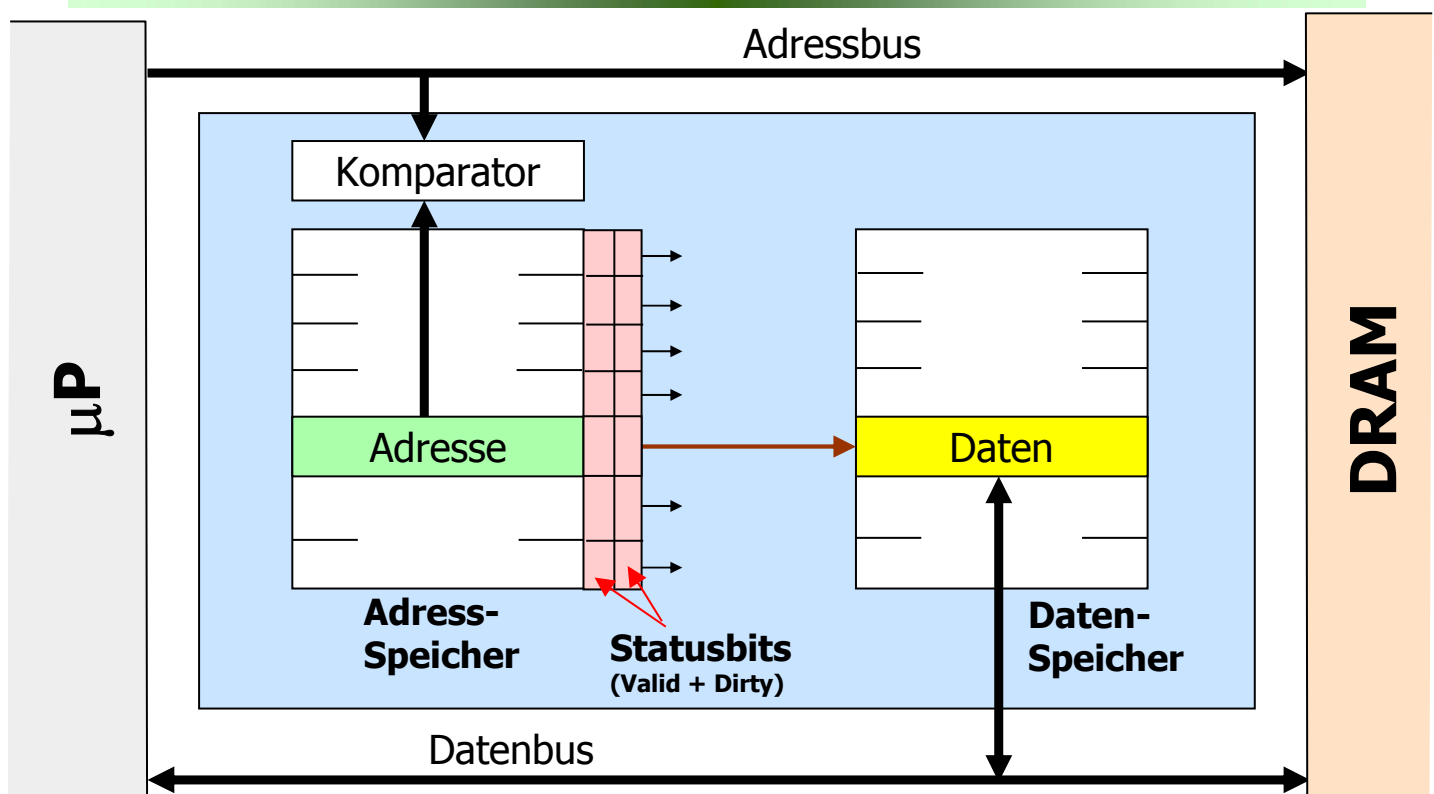


Beispiele für Konsistenzprobleme

- Andere Systemkomponenten, z. B. DMA-Controller, finden nun unter Umständen „veraltete Daten“ im Hauptspeicher vor, die von der CPU längst geändert, jedoch noch nicht in den Hauptspeicher übertragen wurden.
 - Ebenfalls können andere Systemkomponenten Daten im Hauptspeicher ändern, während die CPU noch mit den alten Daten im Cachespeicher arbeitet.
- ➔ aufwendige Verfahren bei der Cache-Steuerung zur Verhinderung solcher Inkonsistenzen sind erforderlich (z. B. muss die Cache-Steuerung über jede Datenänderung im Hauptspeicher informiert werden).



Aufbau eines Cache-Speichers



Aufbau eines Cache-Speichers

Ein Cache-Speicher besteht aus zwei Speicher-Einheiten:

- **Datenspeicher:**
enthält die im Cache abgelegten Daten
- **Adressspeicher:**
enthält die Adressen dieser Daten im Arbeitsspeicher

In älteren Mikrorechner-Caches:

jeder Dateneintrag besteht aus genau einem Wort des Hauptspeichers



Aufbau eines Cache-Speichers

Heute:

jeder Dateneintrag besteht aus einem ganzen Datenblock (line, bis 64 Byte).

Mit jedem Datum, auf das der Prozessor zugreift, wird die Umgebung miteingelagert (Hoffnung auf Lokalität von Programmen).

Im Adressspeicher wird die Basisadresse jedes Blocks abgelegt.



Aufbau eines Cache-Speichers

- Jede Cache-Zeile enthält ein (Adress, Daten)-Paar und Statusbits.
- Ein *(Daten)-Block* ist eine zusammengehörende Reihe von Speicherplätzen.
- Dazugehörig wird ein *Adresstikett (Index, Cache-Tag)* im Adress-Speicher ablegt.
- Das Cache-Tag enthält die Adresse des aktuellen Blocks im Hauptspeicher.
- Die Statusbits sagen aus, ob die Daten im Cache gültig sind.



Aufbau eines Cache-Speichers

Ein Komparator ermittelt, ob das zu einer auf dem Adressbus liegende Adresse gehörende Datum auch im Cache abgelegt worden ist

➔ Adressvergleich mit den Adressen im Adressspeicher

Dieser Adressvergleich muss **sehr schnell** gehen (*möglichst in einem Taktzyklus*), da sonst der Cachespeicher effektiv langsamer wäre als der Arbeitsspeicher.



Cache-Strukturen

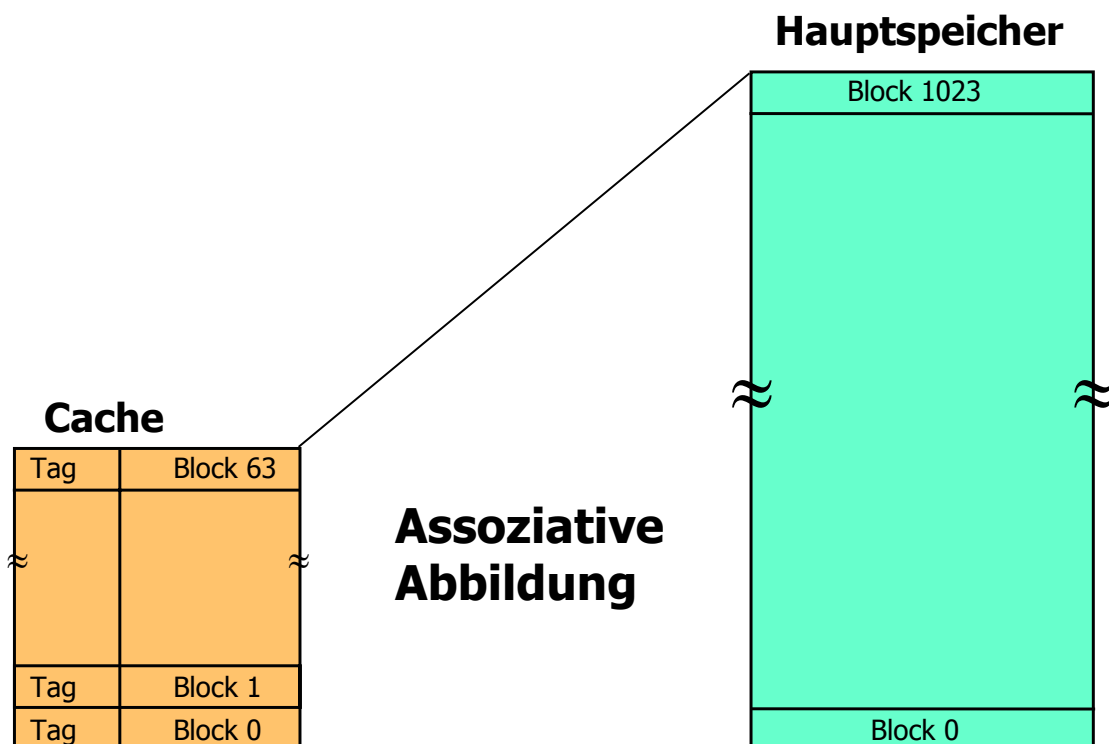
Wie wird festgestellt, ob die benötigten Daten im Cache sind und falls ja wie können diese gefunden werden?

3 Techniken für den Adressvergleich → 3 Cache-Typen:

- **Voll-Assoziativer Cache**
- **Direct Mapped Cache**
- **n-Way Set Associative Cache**



Voll-assoziativer Cache



Voll-Assoziativer Cache

Vollparalleler Vergleich aller Adressen im Adressspeicher in einem einzigen Taktzyklus

Vorteil:

- ein Datum kann an beliebiger Stelle im Cache abgelegt werden
- Optimale Cache-Ausnutzung, völlig freie Wahl der Strategie bei Verdrängungen

Nachteil:

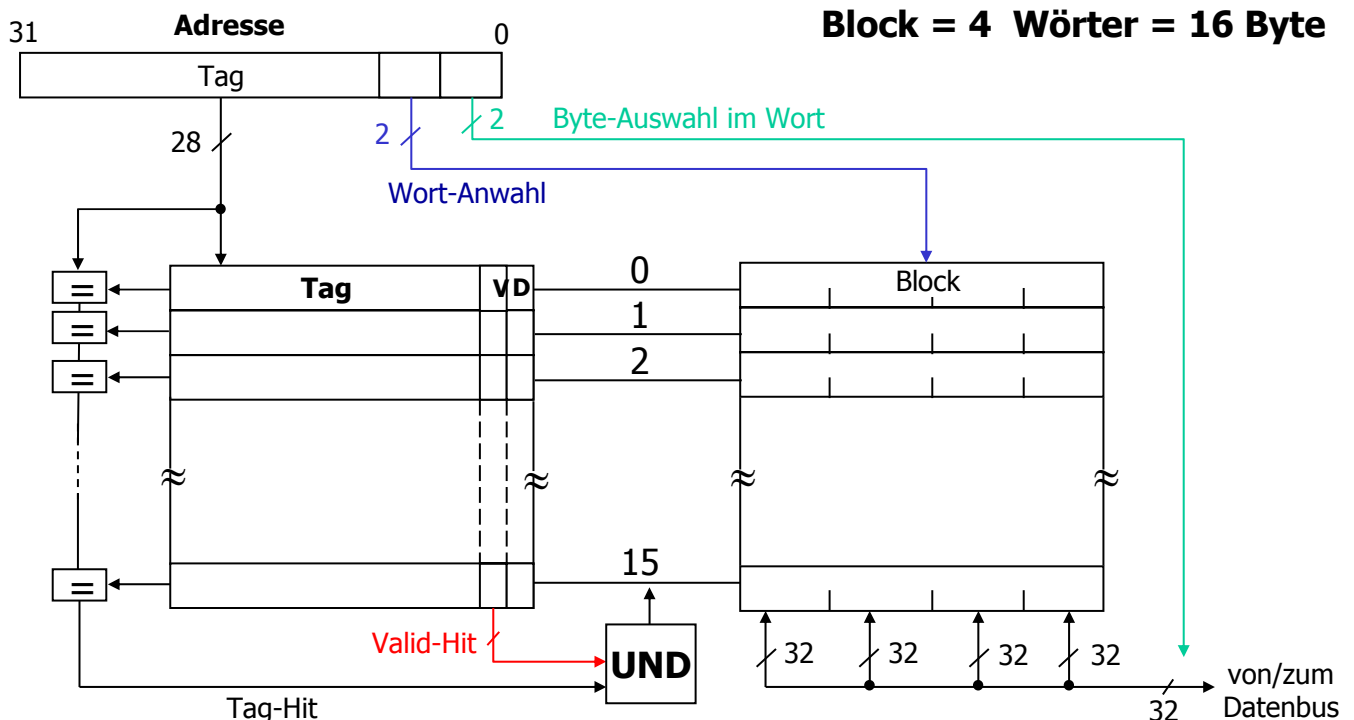
- Hoher Hardwareaufwand (für jede Cache-Zeile ein Vergleich)
➔ nur für sehr kleine Cachespeicher realisierbar
- Die große Flexibilität der Abbildungsvorschrift erfordert eine weitere Hardware, welche die **Ersetzungsstrategie** (welcher Block soll überschrieben werden, wenn der Cache voll ist) realisiert.



Beispiel: Vollassoziativer Cache

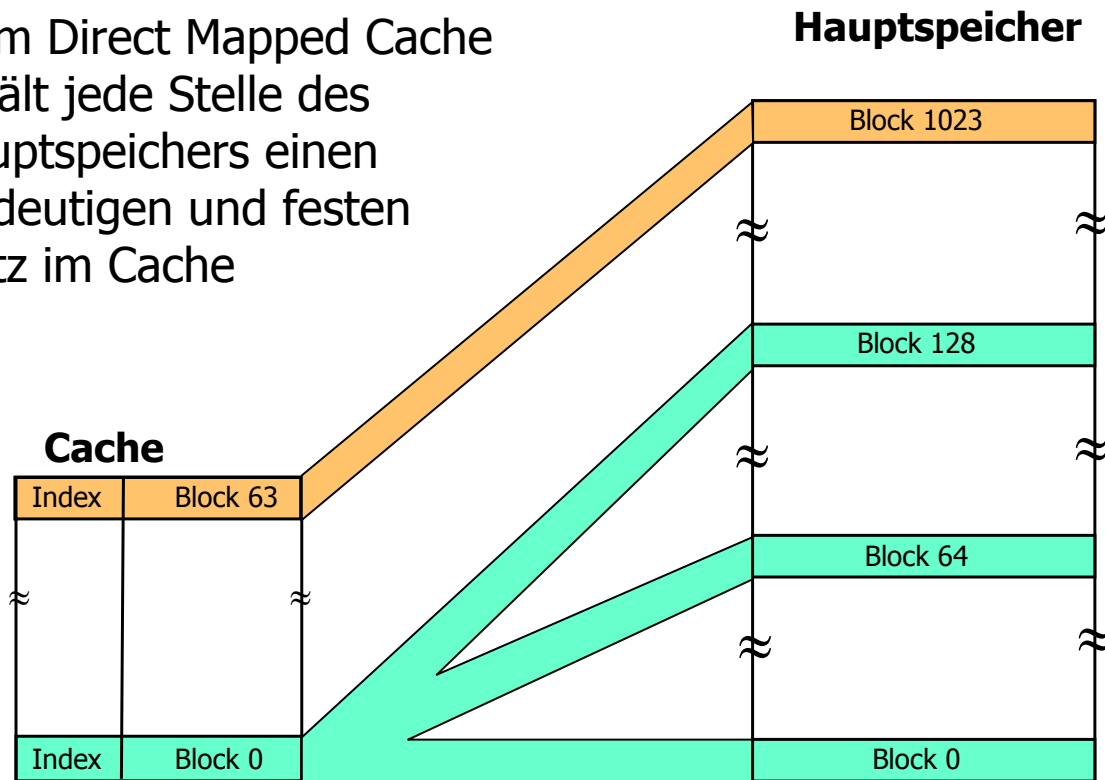
Kapazität: 256 Byte

Block = 4 Wörter = 16 Byte



Direct-mapped-Cache

Beim Direct Mapped Cache erhält jede Stelle des Hauptspeichers einen eindeutigen und festen Platz im Cache



Direct-mapped-Cache

Nachteile:

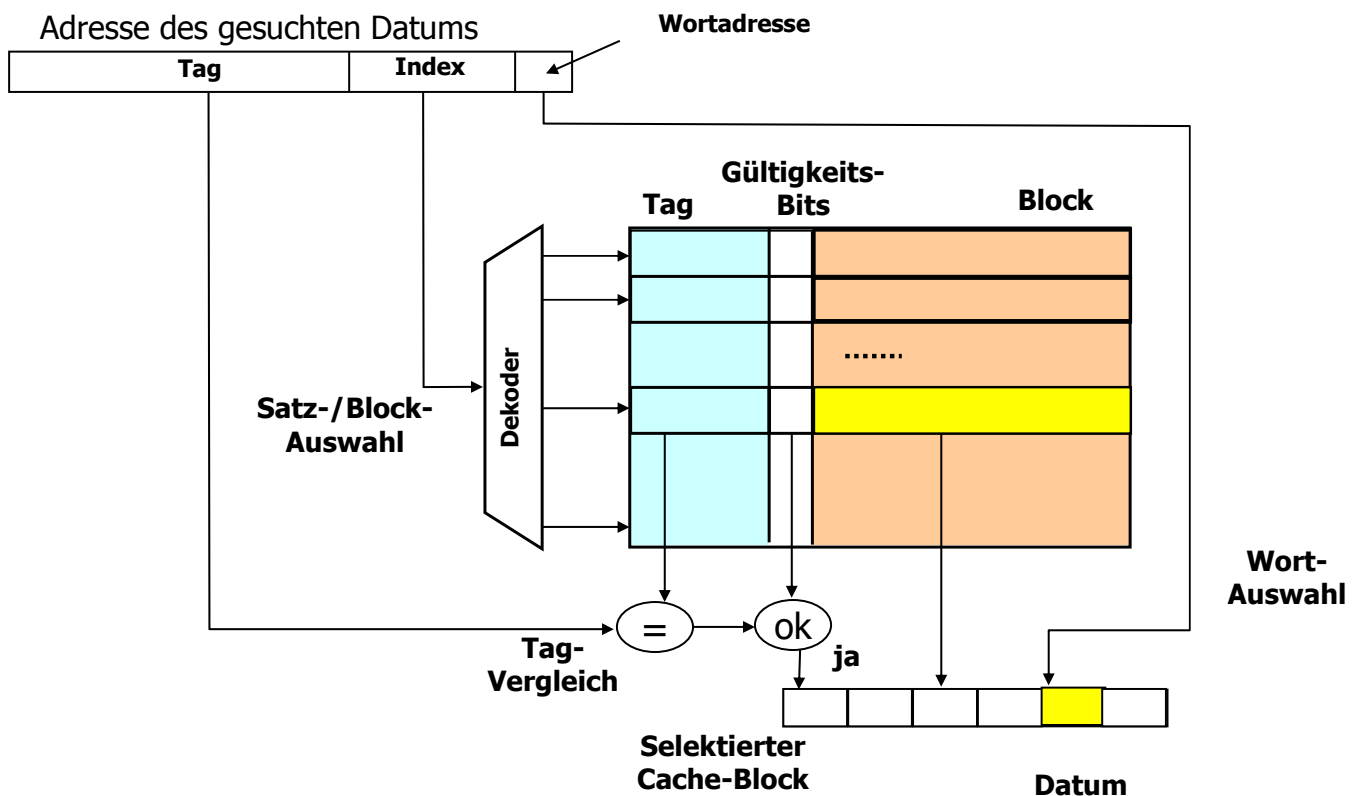
Ständige Konkurrenz der Blöcke (z. B. 0, 64, 128,...), obwohl andere Blöcke im Cache frei sein können.

Vorteile:

Geringer Hardwareaufwand für die Adressierung, da nur ein Vergleich für alle Tags benötigt wird.



Adressierung im Direct-Mapped Cache



Merkmale des Direct Mapped Cache

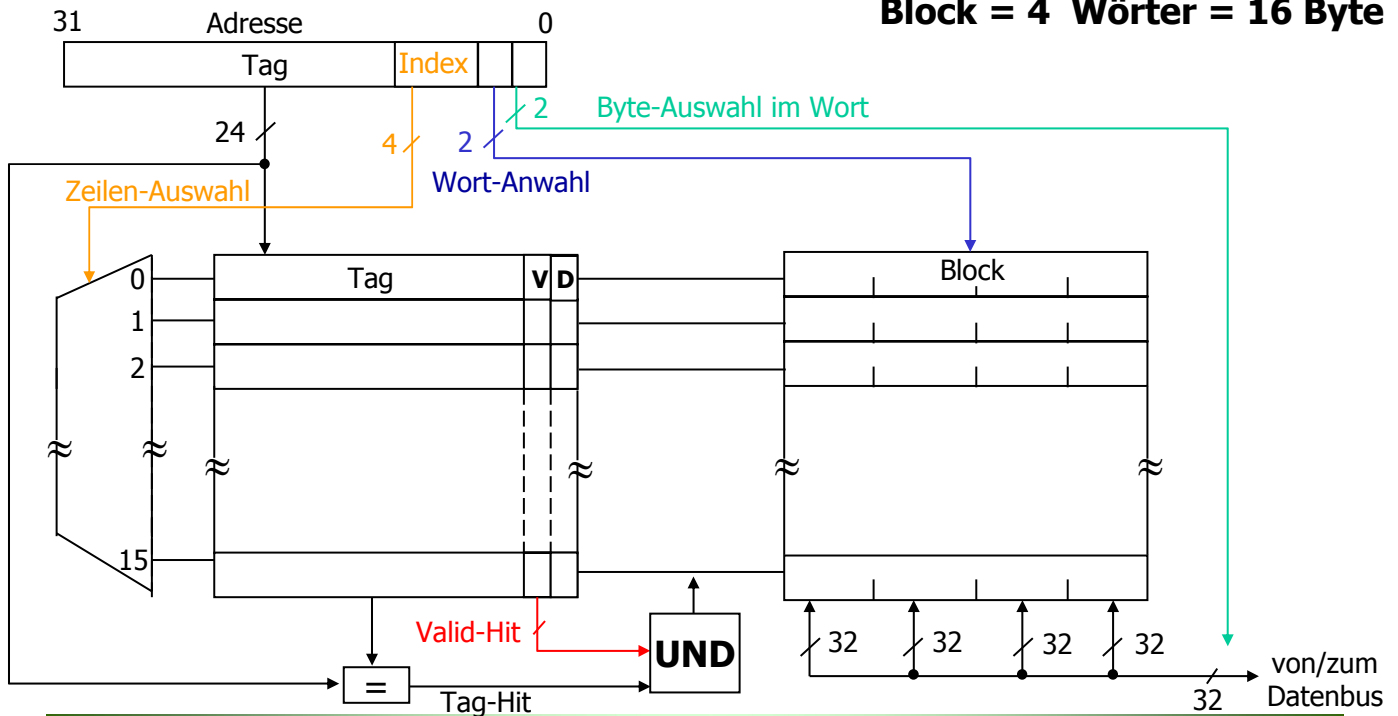
- Einfache Hardware-Realisierung (nur ein Vergleich und ein Tag-Speicher)
- Der Zugriff erfolgt schnell, weil das Tag-Feld parallel mit dem zugehörigen Block gelesen werden kann
- Es ist keine Ersetzungsstrategie erforderlich, weil die direkte Zuordnung keine Alternativen zulässt
- Auch wenn an anderer Stelle im Cache noch Platz ist, erfolgt wegen der direkten Zuordnung eine Ersetzung
- Bei einem abwechselnden Zugriff auf Speicherblöcke, deren Adressen den gleichen Index-Teil haben, erfolgt laufendes Überschreiben des gerade geladenen Blocks. Es kommt zum **"Flattern"** (trashing)



Beispiel: Direct-mapped-Cache

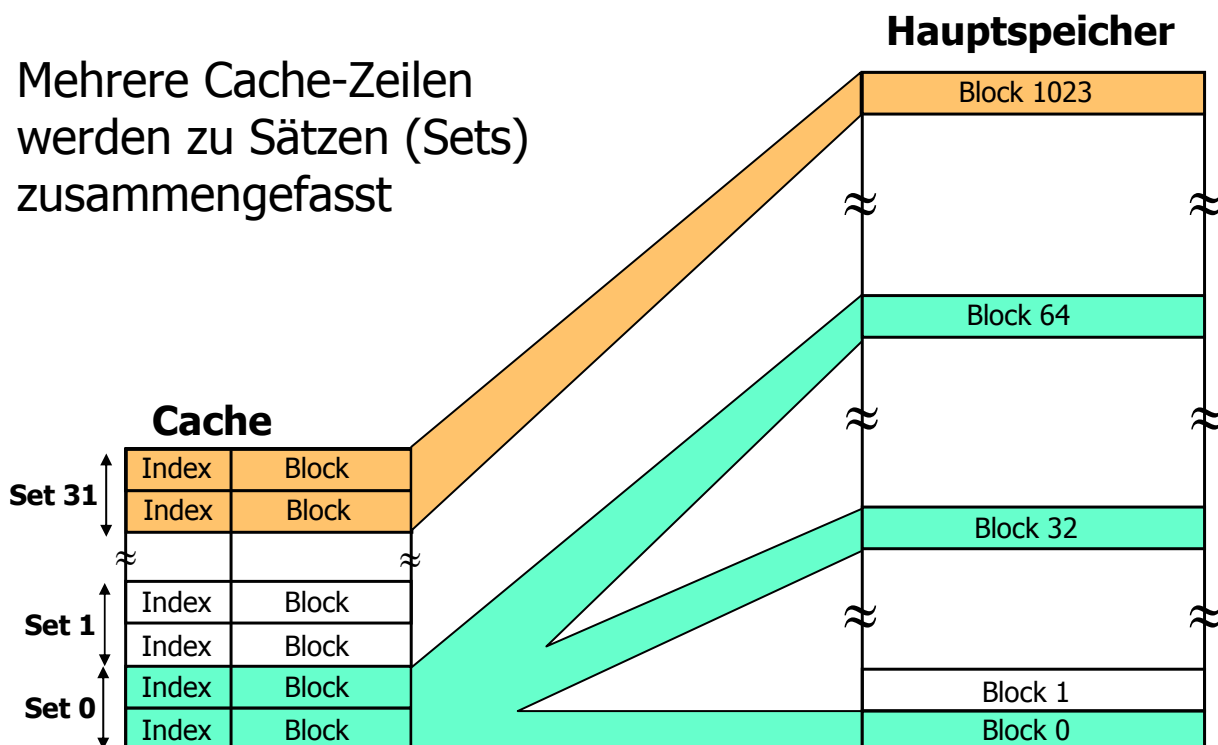
Kapazität: 256 Byte

Block = 4 Wörter = 16 Byte



n-way-set-assoziativer Cache

Mehrere Cache-Zeilen werden zu Sätzen (Sets) zusammengefasst



n-way-set-assoziativer Cache

- Kompromiss zwischen direct-mapped-Cache und vollassoziativen Cache.
- Verbesserte Trefferrate, da hier eine Auswahl möglich ist (der zu verdrängende Eintrag kann unter n ausgewählt werden).
- **Ersetzungsstrategie notwendig:**
 - Zyklisch (der zuerst eingelagerte Eintrag wird auch wieder verdrängt, FIFO-Strategie)
 - LRU-Strategie (*least recently used*) der am längsten nicht mehr benutzte Eintrag wird entfernt.
 - Zufällig (durch Zufallsgenerator)



n-Way Set Associative Cache

Zum Auffinden eines Datums müssen alle n Tags mit demselben Index parallel verglichen werden

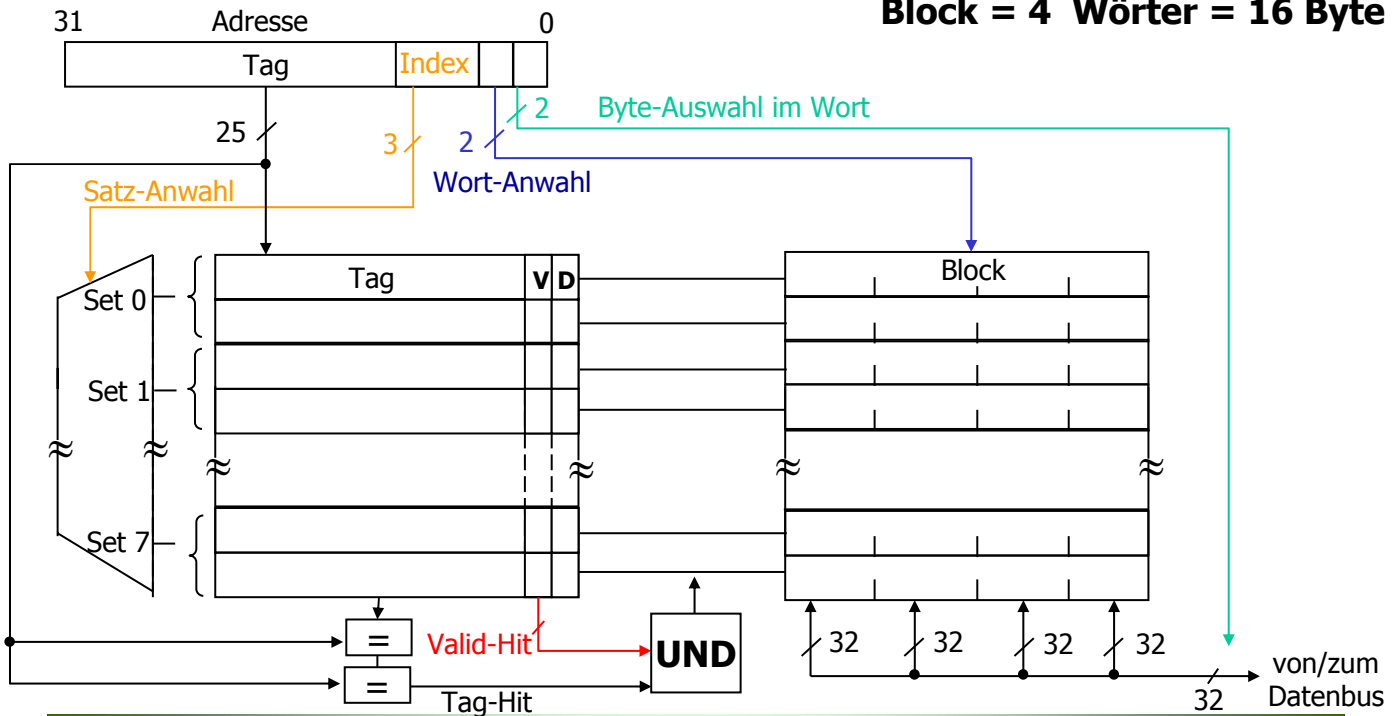
- ➔ der Aufwand steigt mit der Zahl n , für große n nähert sich der Aufwand den voll-assoziativen Caches
- ➔ Kompromiß zwischen Direct Mapped Cache und voll-assoziativem Cache



Beispiel: 2-way-set-assoziativer Cache

Kapazität: 256 Byte

Block = 4 Wörter = 16 Byte

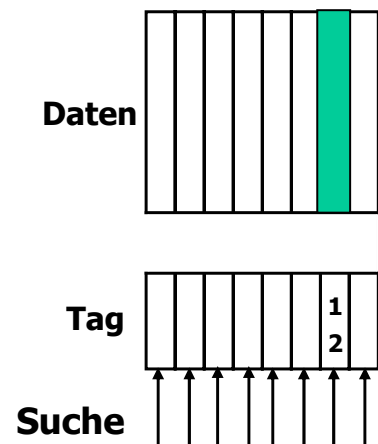
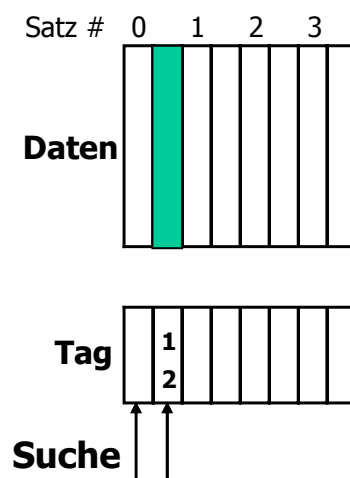
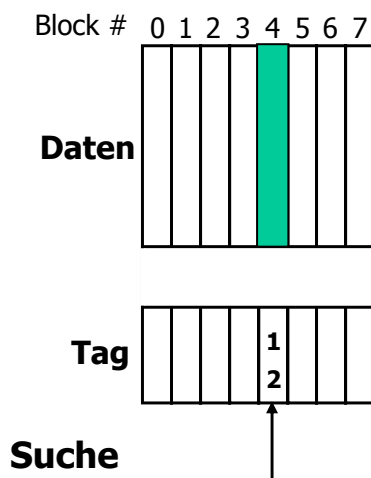


Beispiel: Organisation eines Caches mit 8 Speicherplätzen

Direct-mapped

Set-associative

fully associative



Direct Mapped:

Speicherblock 12 kann nur an einer Stelle stehen

2-Way Set Associative:

Speicherblock 12 kann an zwei Stellen stehen

Voll-Assoziativ:

Speicherblock 12 kann an allen Stellen stehen

