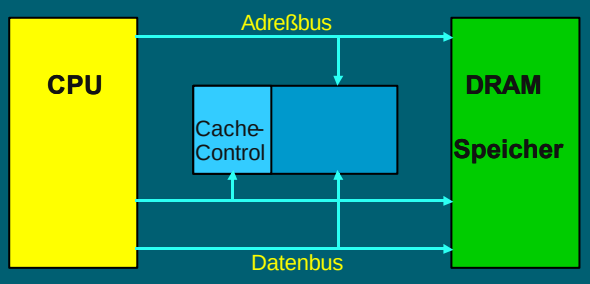
4.2 Caches

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

Lage des Caches



In der Regel besitzt ein Rechner einen getrennten Cache für Instruktionen (**Instruktionscache**) und für 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
  - d.h. zu jedem Zeitpunkt während eines Programmablaufs werden bestimmte Speicherzellen bevorzugt und wiederholt angesprochen (siehe z.B. Schleifen)

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**: Insbesondere bei Großrechnern wird mit jedem Datum auch dessen umgebender Block von Daten geladen in der Erwartung, daß folgende Zugriffe auf diese Daten erfolgen (hier aber nicht betrachtet).

Illustration des Lesezugriffs: cache hit

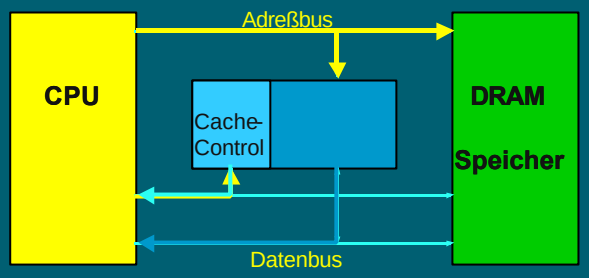
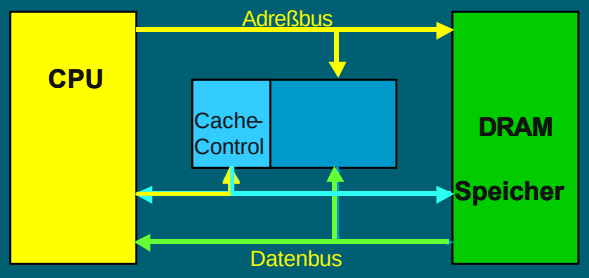


Illustration des Lesezugriffs: cache miss



... einige Wartezykeln

Mittlere Zugriffszeit beim Lesen

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

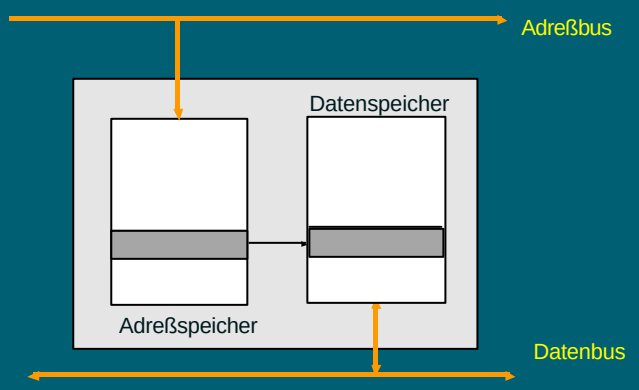
➔ durchschnittliche Zugriffszeit: $c + (1-h) \cdot m$

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)

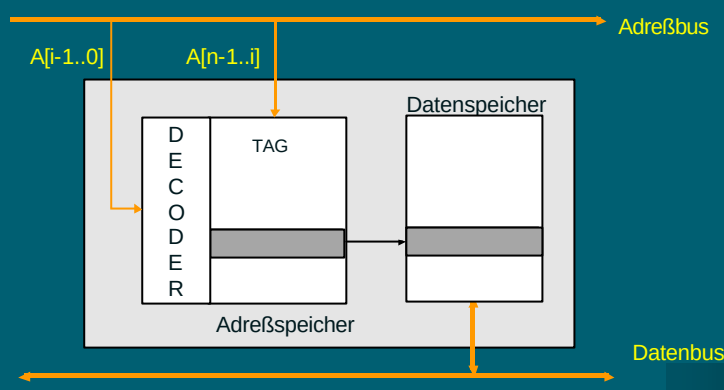
Denkbare Verdrängungsstrategien

- 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

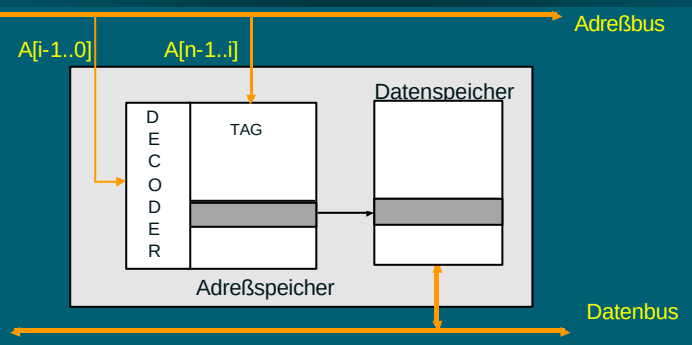
verlangt einen mehr oder weniger großen Aufwand

Direct Mapped Cache

- **Feste Abbildung** der Hauptspeicher-Adressen auf die Cache-Adressen
  - Kein assoziativer Speicher nötig
  - Keine Verdrängungsstrategien nötig

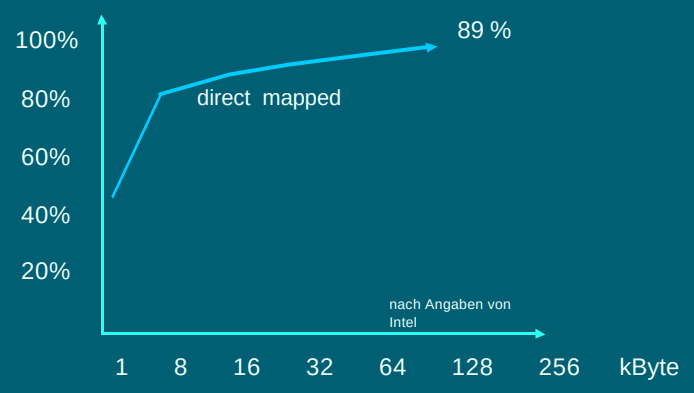


Direct Mapped Cache



- 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 (write-through-Verfahren)

Schreibe Datum in den Arbeitsspeicher unter Adresse a:

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

- **write-through Verfahren**
  - **cache miss:** CPU schreibt das Datum in die Hauptspeicherzelle mit Adresse a. Der Inhalt des Cache wird nicht verändert.
  - **cache hit:** die Kopie der Hauptspeicherzelle im Cache wird aktualisiert, die Hauptspeicherzelle selbst wird sofort aktualisiert

Prinzipielle Funktionsweise: Schreibzugriff (write-back-Verfahren)

Schreibe Datum in den Arbeitsspeicher unter Adresse a:

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

- **write-back Verfahren**
  - **cache miss:** CPU schreibt das Datum in die Hauptspeicherzelle mit Adresse a. Der Inhalt des Cache wird nicht verändert.
  - **cache hit:** die Kopie der Hauptspeicherzelle im Cache wird aktualisiert und durch „dirty bit" als verändert markiert, die Hauptspeicherzelle selbst wird erst später aktualisiert, nämlich wenn die Kopie aus dem Cache verdrängt wird

Prinzipielle Funktionsweise: Schreibzugriff (write-allocation-Verfahren)

Schreibe Datum in den Arbeitsspeicher unter Adresse a:

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

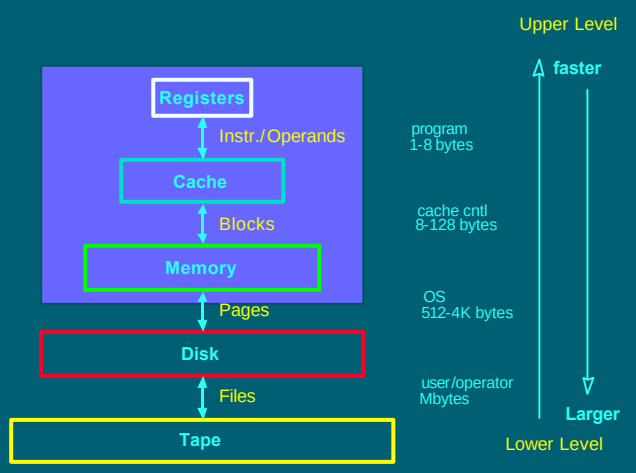
- **write-allocation Verfahren**
  - **cache miss:** CPU schreibt Datum in den Cache (markiert mit „dirty bit"), aber nicht in den Hauptspeicher (dort erst aktualisiert, wenn Kopie aus dem Cache verdrängt).
  - **cache hit:** die Kopie der Hauptspeicherzelle im Cache wird aktualisiert (markiert durch „dirty bit"), die Hauptspeicherzelle selbst wird erst später aktualisiert, nämlich wenn die Kopie aus dem Cache verdrängt wird

Write-through versus Write-back/Write-alloaction

Vorteile von Write-back/Write-alloaction

- 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-alloaction

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 Module veraltete Werte vorfinden und mit ihnen rechnen.

Speicherorganisation