

Modul Informatik-II

Kurs Informatik-3: Teil-5

www.engineering.zhaw.ch/de/engineering/studium/bachelor/informatik/studium-zurich.html

Prof. Dr. Olaf Stern
Leiter Studiengang Informatik
+41 58 934 82 51
olaf.stern@zhaw.ch

Lernziele: (Allgemein)

- Die Studierenden kennen die *grundlegende Architektur von Rechnern* und die wichtigsten *Architekturelemente*.
- Sie sind vertraut mit der *elementaren Arbeitsweise eines Computers* und der *hardwarenahen Programmierung*. Sie können diese an einfachen Beispiel erläutern.
- Die Studierenden kennen die grundsätzlichen Aufgaben eines *Betriebssystems*. Sie können die *typischen* Verfahren und *Algorithmen*, die bei der *Entwicklung* von *Betriebssystemen* zur Anwendung gelangen, beschreiben.

Lernziele: (Allgemein)

- Die Kurse der Module Informatik I und Informatik II (der Modulgruppen "Grundlagen der Informatik I+II") vermitteln den Studierenden die *Grundlagen der Informatik, die jede / jeder Studierende unabhängig von der Wahl der Wahlpflichtmodule im Fachstudium erlangen sollte.*
- Die vermittelten Grundlagen *werden in den Modulen im Fachstudium vorausgesetzt.*

Lernziele: Spezifisch Teil-5

- Die Studierenden kennen die Ziele für den Einsatz von Zwischenspeichern (**Cache**) und die Einordnung in der **Speicherhierarchie** eines Rechners; sie können insbesondere die Bedeutung eines **Caches** für die Leistung von Rechnern darlegen.
- Sie können den **grundlegenden Aufbau und die Funktionsweise eines Cache** (Lese- und Schreibvorgänge) erläutern und sind vertraut mit den verschiedenen elementaren Ansätzen **direktabbildend** und **(satz-) assoziativ**.
- Sie können an Beispielen den Einfluss der Parameter (Auslegung eines **Cache**) aufzeigen.

Lernziele: Spezifisch Teil-5

- Die Studierenden kennen den Aufbau und die Umsetzung eines *virtuellen Speichers*.

Themenüberblick Teil-5

Technische Informatik / Rechnerarchitektur

- Einführung / Übersicht
- Grundlegende Rechnerarchitektur
- Prozessoren
- Befehle – die „Wörter“ des Rechners
- „Mini-Power-PC“
- **Speicher**
 - Speicherarten und Speicheraufbau
 - Speicherhierarchie
 - **Cache (Puffer, Zwischenspeicher)**
- „Mini-Power-PC“ (Fortsetzung)

Lerninhalte Teil-5

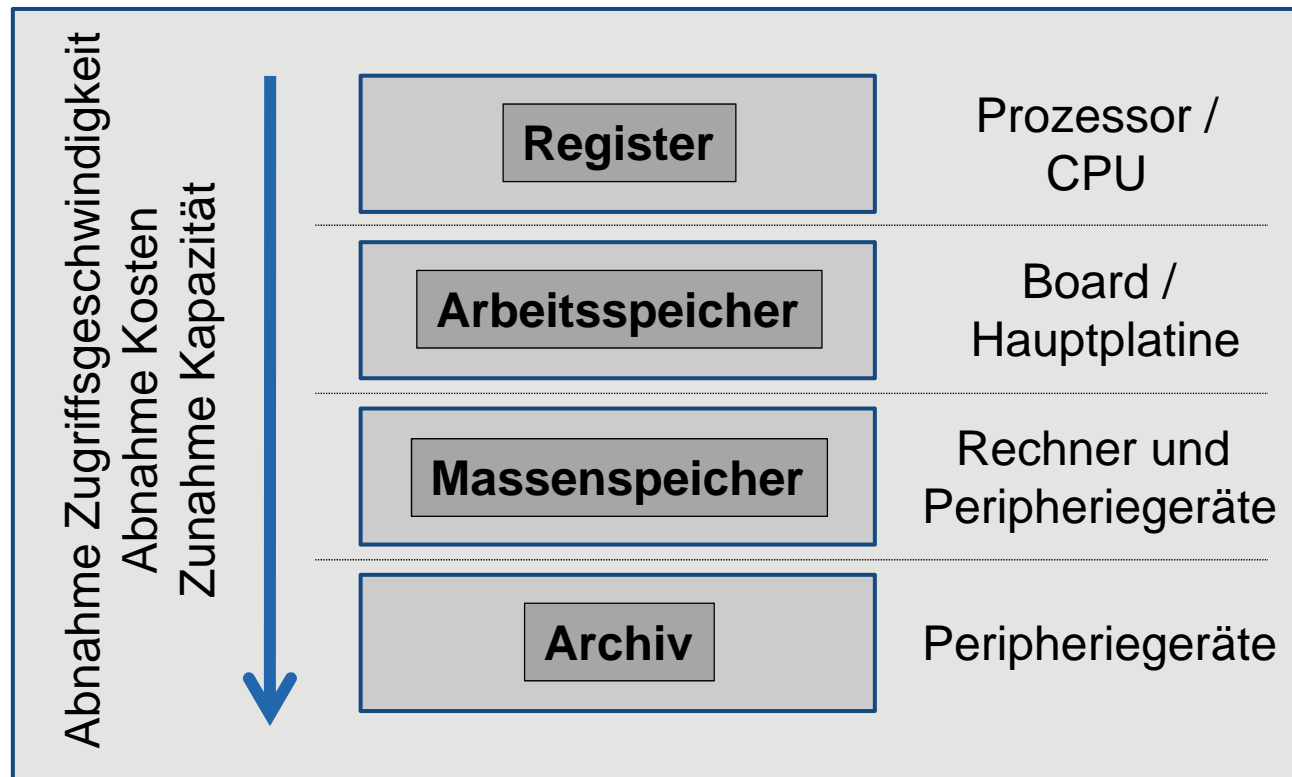
- **Cache (Zwischenspeicher)**
 - **Einführung / Motivation**
 - **Lokalitätsprinzip**
 - **Grundlegende Definitionen**
 - **Lesezugriff**
 - Assoziativer Cache
 - Direktabbildender Cache
 - Mischformen

Lerninhalte Teil-5

- **Cache (Zwischenspeicher)**
 - ...
 - **Schreibzugriff**
 - *Write Through*
 - *Write Puffer*
 - *Write Back*
 - **Optimierung und *Multi-Level-Cache***
 - ***Virtuelle Speicher***
 - Aufbau
 - Umsetzung

Architekturelemente - Speicher

Speicheraufbau – Speicherhierarchie



Architekturelemente - Speicher

Speicheraufbau – Speicherhierarchie

- Prozessoren (Rechen- / Steuerwerke) können Daten **sehr schnell** bearbeiten.
- Der Zugriff auf Daten im Arbeitsspeicher (heute **DRAMs**) ist dagegen **langsam** (unabhängig vom **Von-Neumann-Flaschenhals**).
- Nur **wenige Daten** können in den Registern vom(n) Rechen- und Steuerwerk(en) schnell bearbeitet werden.

Folge: **Der Prozessor muss sehr häufig auf Daten warten!**

Dieses wäre für sehr viele Anwendungen der Fall:

z. B. Bearbeitung von Texten, Matrizen, Audio, Video, Berechnung physikalischer oder chemischer Formeln, Näherungsverfahren, ...

Architekturelemente - Speicher

Speicheraufbau – Speicherhierarchie

- Lösung für schnellen Zugriff auf viele Daten für den Prozessor (Rechen- und Steuerwerke) ?

=> Einen Zwischenspeicher, „**Puffer-Speicher**“ auch „**Cache**“ genannt, nutzen.

Prinzip: Dem Nutzer (Rechner) „**viel**“ **kostengünstigen Speicher** mit **schnellen Zugriffszeiten** (vergleichbar der schnellsten Speichertechnologie) zur Verfügung zu stellen.

[Der Begriff **Cache** wurde in den 60er-Jahren ursprünglich für den Pufferspeicher zwischen Prozessor und Arbeitsspeicher geprägt.]

Architekturelemente - Speicher

Speicheraufbau – Speicherhierarchie

- **Grundideen für einen *Cache*:**
 - **Der Zugriff auf Daten im *Cache* ist:**
 - Zwar (ev.) **langsamer** als auf **Daten direkt** im **Register**
 - Aber **erheblich schneller** als auf **Daten** im **Arbeitsspeicher**.
 - **Die Grösse / Kapazität eines Caches ist**
 - Einerseits **deutlich grösser** als die der **Register**
 - Andererseits **deutlich kleiner** als der **Arbeitsspeicher**.

! Dieser Ansatz lässt sich auf alle(n) Speicherebenen übertragen / nutzen!

Architekturelemente - Speicher

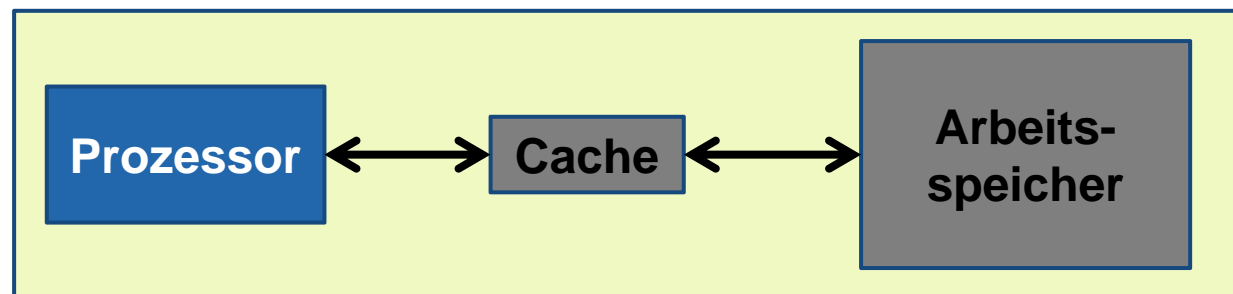
Speicheraufbau – Speicherhierarchie

- **Grundideen für einen *Cache*:** (Forts.)

- Statt auf den Arbeitsspeicher greift der Prozessor auf die Daten im *Cache* zu:

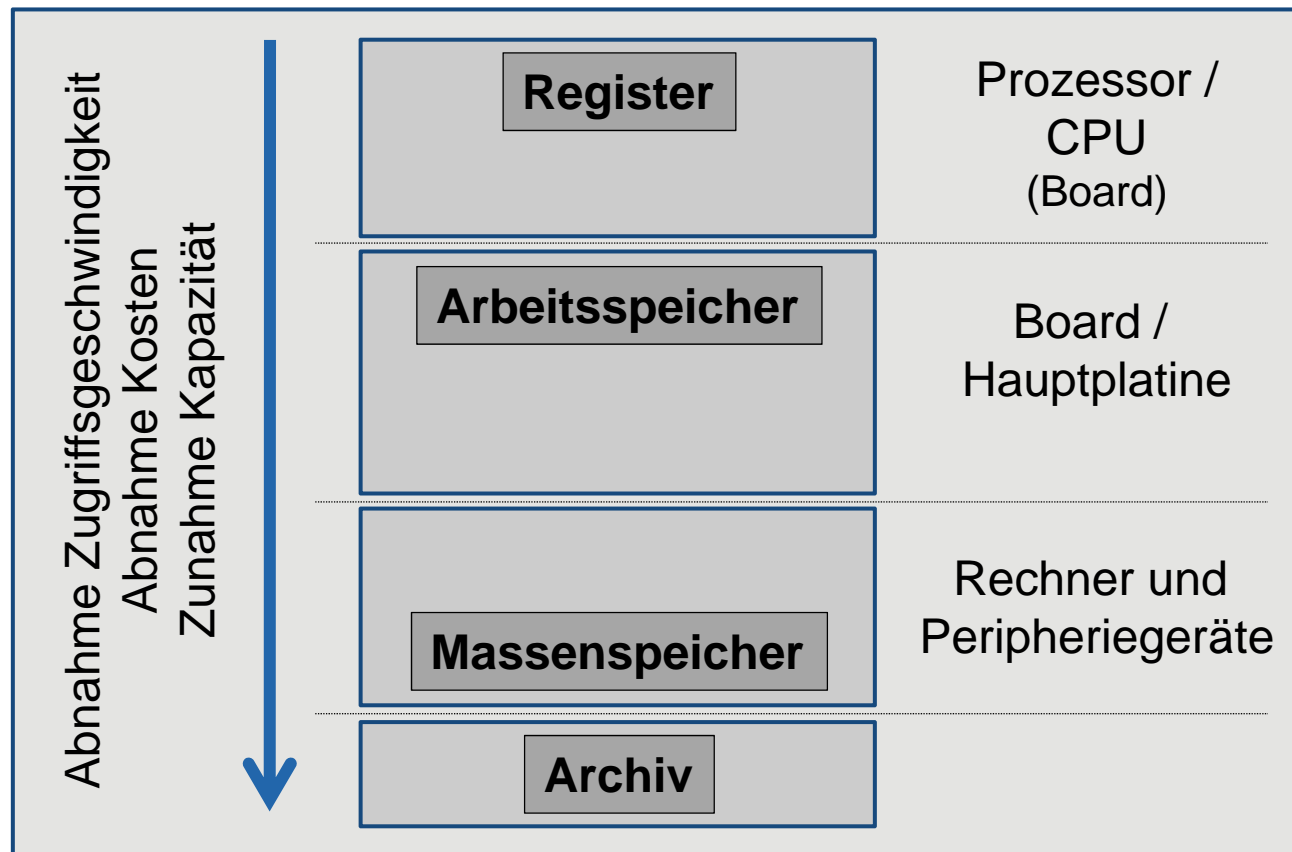
=> Dafür müssen die Daten zuvor „nur“ aus dem **Arbeitsspeicher** in den *Cache* kopiert worden sein.

- Grobschema:

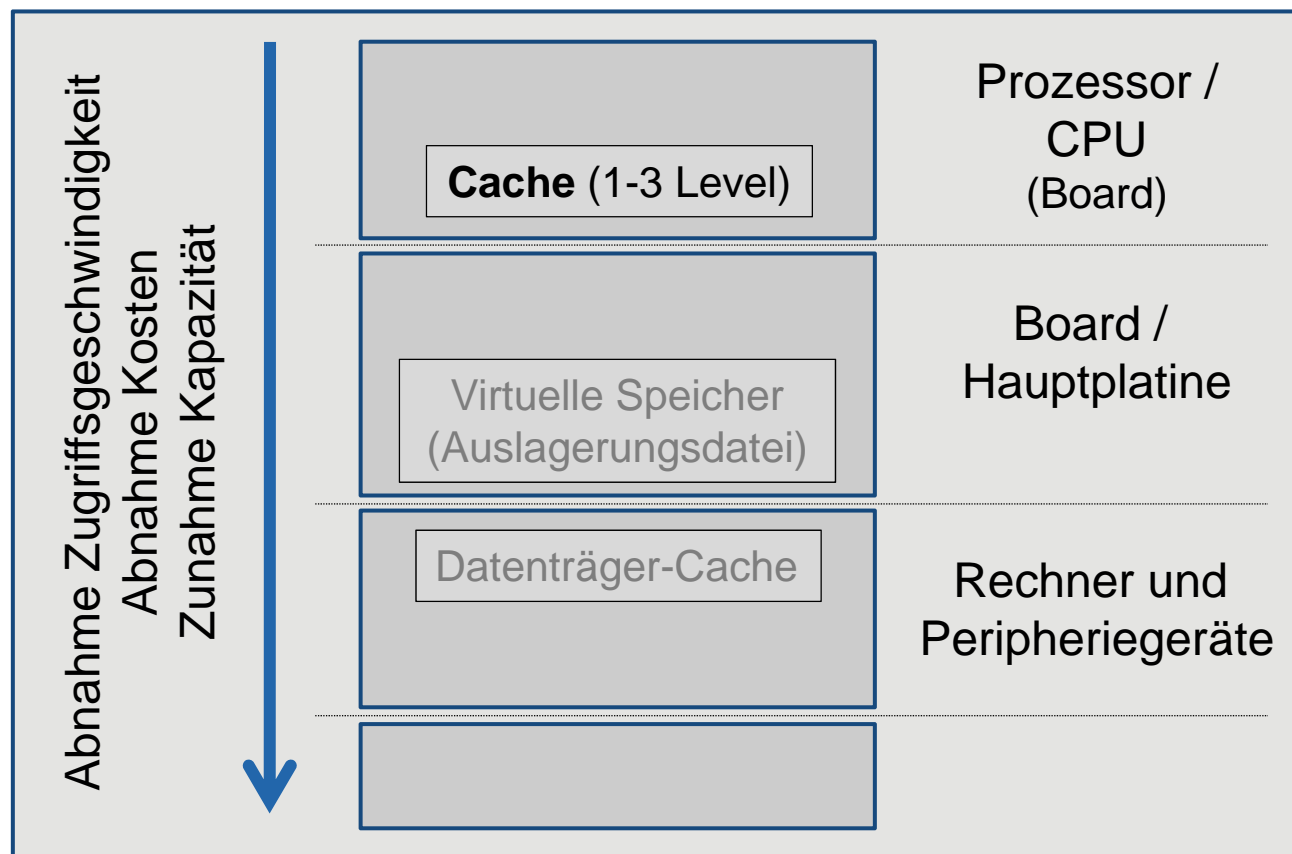


Architekturelemente - Speicher

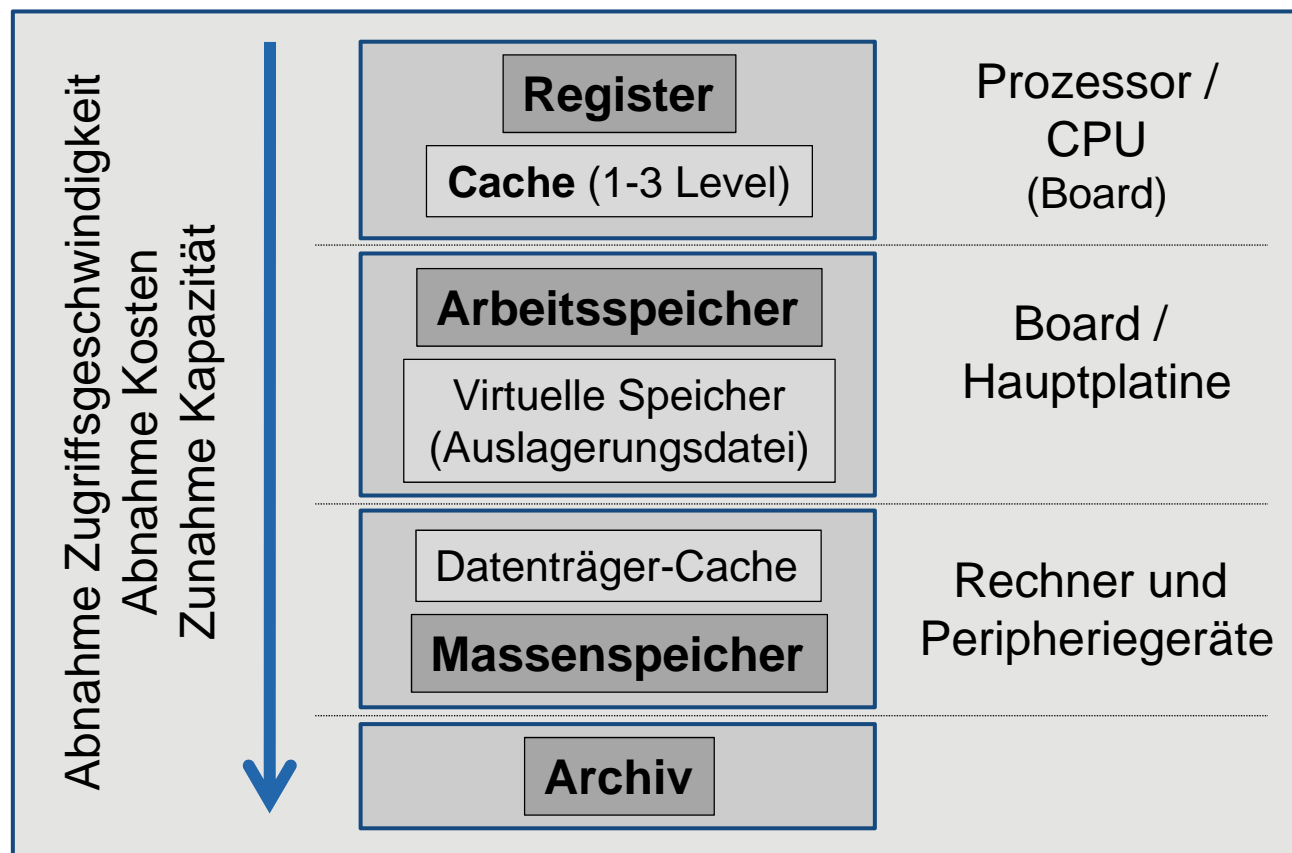
Speicheraufbau – Speicherhierarchie



Zwischenspeicher in der Speicherhierarchie



Typische Speicherhierarchie eines Rechners



Architekturelemente - Speicher

Speicheraufbau – verwendete Technologien

- Register: sehr schnelle Hardware, **SRAM**
- Cache: **SRAM**, **Assoziativ**-Speicher
- Arbeitsspeicher: **DRAM** (heute fast ausschliesslich)
- Auslagerungsdatei / Virtuelle Speicher: **DRAM**, **Flash**, schnelle Festplatten, **Solid State Disk**
- Datenträger-Cache: **DRAM**, **SRAM** (im Controller der Festpl.)
- Massenspeicher: Festplatten (magnetisch, optisch), **Solid State Disk**, **Flash** (**USB-Stick**), ...
- Archiv: Festplatten, CD, DVD, **Solid State Disk**, **Flash** (**USB-Stick**), (Magnet-)Bänder, Disketten, ...

Architekturelemente - Speicher

Speicheraufbau – Speicherhierarchie

- **Grundideen** für einen **Cache**: (Forts.)
 - „... **deutlich kleiner** als der Arbeitsspeicher“.
(muss ja **deutlich kleiner** sein, sonst können die **Kosten nicht reduziert** werden!)
 - Zudem werden die **Strukturen erheblich komplexer !**
(= zeitaufwendiger)
 - **Mehr Bauteile**
 - **Daten** müssen vom Register und Arbeitsspeicher zum **Cache übertragen** werden.
 - ...

Architekturelemente - Speicher

Speicheraufbau – Speicherhierarchie

- **Grundideen für einen *Cache*: (Forts.)**
 - „... **deutlich kleiner** als der Arbeitsspeicher“.
(muss ja **deutlich kleiner** sein, sonst können die **Kosten nicht reduziert** werden!)
 - Zudem werden die **Strukturen erheblich komplexer !**
(= zeitaufwendiger)

Wieso funktioniert ein *Cache* erwiesenermassen dennoch sehr effizient?

Anmerkung: Konkret haben die Entwicklungen von *Caches* wesentlich zur **Steigerung der Rechenleistung** in den letzten Jahren beigetragen!

Architekturelemente - Speicher

Speicheraufbau – Caches

Wieso funktioniert ein *Cache* erwiesenermassen
dennoch sehr effizient?

- Lösung: *Lokalität*

- a) Zeitliche *Lokalität*

- Daten, die **zuletzt genutzt wurden**, werden mit **grosser Wahrscheinlichkeit** auch wieder in **naher Zukunft** genutzt
 - Bsp.: Befehle in Programmschleifen, Variablen, ...

- b) Räumliche *Lokalität*

- Mit sehr grosser Wahrscheinlichkeit wird nach dem Zugriff auf ein Datum in der Folge auf ein Datum **zugegriffen**, das in der „**Nähe**“ abgespeichert ist.
 - Bsp.: Programmschleifen, Feld-Elemente, Textsuche, ...

Architekturelemente - Speicher

Speicheraufbau – Caches

Wieso funktioniert ein *Cache* erwiesenermassen
dennoch sehr effizient?

- Lösung: *Lokalität* (zeitliche und räumliche)

Die **Wahrscheinlichkeit** ist **sehr gross** ist, ein Datum zu verwenden, dass

- **bereits kürzlich verwendet wurde** (zeitliche Lokalität)
- und/oder **nahe zum bisherigen Datum abgespeichert ist** (räumliche Lokalität)⁽⁺⁾

und daher **bereits im *Cache* abgespeichert ist.**

⁽⁺⁾ Blockansatz => folgt

Architekturelemente - Speicher

Caches

■ **Definition:**

„A safe place for hiding or storing things“

[Webster's New World Dictionary of the American Language, 3rd Edition, 1988]

„Cache bezeichnet in der EDV einen schnellen Puffer-Speicher, der Zugriffe auf ein langsames Hintergrundmedium oder zeitaufwendige Neuberechnungen nach Möglichkeit vermeidet.

- **Meist werden hierzu Inhalte/Daten gepuffert, die bereits einmal verwendet und/oder berechnet wurden, um beim nächsten Zugriff schneller zur Verfügung zu stehen, oder/und**
- **vermutlich bald benötigte Daten vorab vom langsamen Hintergrundmedium geladen und bereitgestellt.**

Caches sind als Puffer-Speicher realisiert, die Kopien zwischenspeichern. Sie können als Hardware- oder Softwarestruktur ausgebildet sein.“

[<http://de.wikipedia.org/wiki/Cache>, Stand Aug. 2011]

Architekturelemente - Speicher

Caches

■ Grundbegriffe: [nach Patterson/Hennessy]

– „**Hit Rate**“ R_{hit} (**Trefferrate**):

Anteil der Speicherzugriffe auf einen **Cache**, die zu einem **Treffer** führen (**Daten** befinden sich im **Cache**).

– „**Miss Rate**“ R_{miss} (**Fehlzugriffsrate**):

Anteil der Speicherzugriffe auf einen **Cache**, die **nicht** zu einem **Treffer** führen (**Daten** befinden sich **nicht** im **Cache**).

Es gilt: $R_{miss} = 1 - R_{hit}$

Architekturelemente - Speicher

Caches

■ Grundbegriffe: (Forts.)

- „**Hit Time**“ t_{hit} (**Zugriffszeit bei Treffer**):

Erforderliche Zeit für den **erfolgreichen Zugriff** auf ein **Datum**⁽⁺⁾ im **Cache**, inkl. der Zeit, die für den **Test** erforderlich ist, ob das **Datum** im **Cache** vorhanden ist.

- „**Miss Penalty**“ t_{miss} (**Fehlzugriffsaufwand**):

Erforderliche Zeit für den **Austausch eines Blocks**⁽⁺⁾ im **Cache** aus der **nächsten Ebene** (z. B. dem Arbeitsspeicher), inkl. der Zeit, **diesen Block Nutzer / Prozessor zur Verfügung zu stellen**.

⁽⁺⁾Mit einem Datum wird im folgenden ein **Wort** assoziiert; Blockansatz => folgt

Architekturelemente - Speicher

Caches

▪ Grundbegriffe: Bedeutung

Da der **Fehlzugriffsaufwand** sehr hoch ist (Faktoren grösser als Zugriffszeit bei einem Treffer), reichen schon **wenige Fehlzugriffe** aus, um den **Nutzen eines Caches** deutlich zu reduzieren.

=> Es muss das primäre Ziel der Architektur eines Speichersystems mit **Cache** sein, die „**Miss Rate**“ zu minimieren ...
... und die „**Hit Time**“ zu optimieren; natürlich auch den **Fehlzugriffsaufwand** zu minimieren.

(Dieses gilt ebenso für das Betriebssystem und die Compiler / Programme)

Architekturelemente - Speicher

Caches

▪ Grundbegriffe: Bedeutung – Beispiel (vereinfacht)

Beispieldaten für Prozessor ohne *Cache*:

- Die **Zykluszeit t_c** für einen **Befehl** beträgt **1 Zeiteinheit** (mit **CPI = 1**).
- Der **Zugriffszeit t_s** für den Zugriff auf ein **Datum** direkt im **Arbeitsspeicher** beträgt **50 Zeiteinheiten**.
- Durchschnittlich **40%** (Anteil **A_s**) **aller Befehle** eines Programms erfordern den Zugriff auf den **Arbeitsspeicher** (Lesen oder Schreiben).

Architekturelemente - Speicher

Caches

■ Grundbegriffe: Bedeutung – Beispiel (vereinfacht)

Beispieldaten für Prozessor ohne *Cache*:

– ...

Beispieldaten für Prozessor mit *Cache*:

- Die **Hit Time t_{hit}** beträgt **2** Zeiteinheiten.
- Der **Fehlzugriffsaufwand t_{miss}** beträgt **100** Zeiteinheiten:
 - a) für **Test**, ob **Datum** im **Cache** steht,
 - b) um Datum im **Arbeitsspeicher** zu adressieren / anzusprechen und
 - c) um **Datum/Bock** in den **Cache** zu übertragen.
- Die **Hit Rate R_{hit}** beträgt a) **65%** bzw. b) **98%**.

Architekturelemente - Speicher

Caches

■ Grundbegriffe: Bedeutung – Beispiel (vereinfacht)

Um wie viel % steigert der **Cache** die Rechenleistung?

- Durchschnittliche Rechenzeit ohne **Cache** t_{ohneC} :

$$t_{ohneC} = (1 - A_S) * t_C + A_S * t_S$$

- Durchschnittliche Rechenzeit mit **Cache** t_{mitC} :

$$t_{mitC} = (1 - A_S) * t_C + A_S * (R_{hit} * t_{hit} + (1 - R_{hit}) * t_{miss})$$

- Steigerung der Rechenleistung: $(t_{ohneC} / t_{mitC}) - 1$

[%]
//

A_S = Anteil Befehle,
die RAM benötigen

t_C = Zykluszeit

t_S = Zugriffszeit Daten
(@RAM)

Anmerkung: Sehr vereinfachte Berechnung – u. a. keine Unterscheidung zwischen Lese- und Schreibzugriffen.

Architekturelemente - Speicher

Caches

■ Grundbegriffe: Bedeutung – Beispiel (vereinfacht)

Um wie viel % steigert der **Cache** die Rechenleistung?

- Durchschnittliche Rechenzeit ohne **Cache** t_{ohneC} :

$$t_{ohneC} = 0.6 * 1 + 0.4 * 50 = 20.6 \text{ [Zeiteinheiten]}$$

- Durchschnittliche Rechenzeit mit **Cache** t_{mitC} :

a) $t_{mitC-65} = 0.6 * 1 + 0.4 * (0.65 * 2 + 0.35 * 100) = 15.12 \text{ [Zeiteinheiten]}$

Steigerung der Leistung: $(t_{ohneC} / t_{mitC-65}) - 1 = (20.6 / 15.12) - 1 \cong 30\%$

b) $t_{mitC-98} = 0.6 * 1 + 0.4 * (0.98 * 2 + 0.02 * 100) = 2.184 \text{ [Zeiteinheiten]}$

Steigerung de Leistung: $(t_{ohneC} / t_{mitC-98}) - 1 = (20.6 / 2.184) - 1 \cong 845\%$

Architekturelemente - Speicher

Caches

▪ Beispiel – Schlussfolgerungen

- ☞ Mit einem **gut entworfenen Cache** kann die **Rechenleistung** eines Rechners **deutlich gesteigert** werden.
- ☞ Die **Verbesserung der Leistungsfähigkeit** des **Caches** bringt **wesentlich mehr** als die **Steigerung** der „**reinen**“ **Rechengeschwindigkeit** eines Prozessors (wie z. B. die Erhöhung der **Zyklusrate**).
- ☞ **Compiler** und **Programme**, die das **Prinzip der Lokalität** unterstützen, können **erheblich** den **Programmablauf beschleunigen**.
- ☞ Ein „**schlechter**“ **Compiler / Programmierer** kann jeden Prozessor „**ausbremsen**“.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

Allgemeine Vorgehensweise:

- 1) Der **Prozessor** fordert ein **Datum** an (über **Adresse**).
- 2) Das **Datum** wird im **Cache** gesucht.
 - a) **Datum** ist im **Cache** (und **gültig**):
 - **Datum** wird an den **Prozessor** zurückgegeben.
 - b) **Datum** ist nicht im **Cache** (oder **ungültig**):
 - **Adresse** wird an den **Arbeitsspeicher** gegeben.
 - **Datum** (bzw. **Block**) wird **dort gelesen** und in den **Cache** geschrieben.
 - mit 2a fortgesetzt

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

Da der *Cache* üblicherweise deutlich kleiner als der Hauptspeicher ist, können sich nicht alle Daten im *Cache* befinden:

- Woher weiss man nun, ob sich ein Datum im *Cache* befindet?
- Und wenn es sich im *Cache* befindet, wie finden wir es?

Grundansätze / Konzepte:

- Assoziativer* Zugriff (voll)
- Direktabbildend*
- Satzassoziativ* (*Satzorientiert*)

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

a) *Assoziativer* Zugriff – Schema

- Bei einem *Assoziativspeicher* wird die **Adresse** des **gesuchten Datums** mit **allen Adressen** der sich im **Speicher** befindenden **Daten parallel verglichen**

und

Falls eine **Adresse** übereinstimmt das **Datum unmittelbar gelesen / geschrieben**.

- Dazu muss zu **jedem Datum zusätzlich** die **Adresse abgespeichert werden** (wird allgemein als *Tag* bezeichnet).

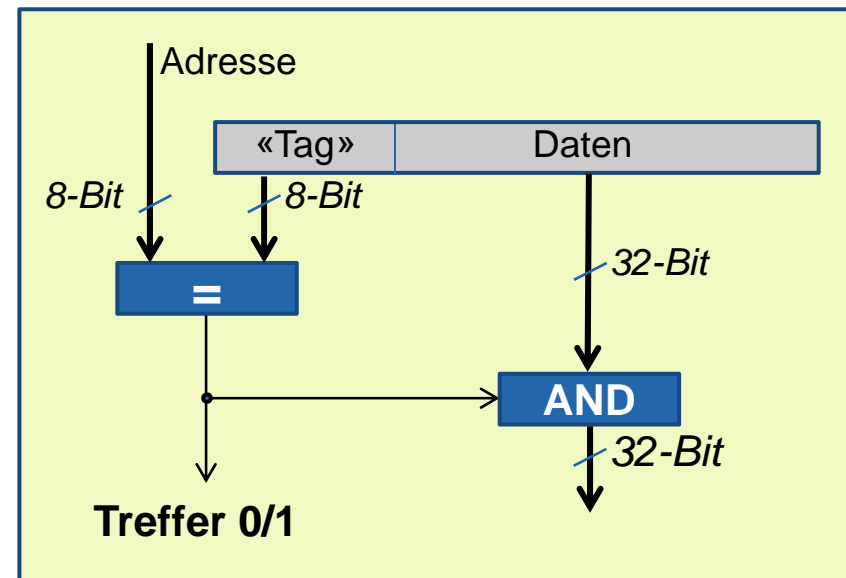
Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

a) *Assoziativer* Zugriff – Grundprinzip (Bsp. Lesen)

Bsp. für einen **assoziativen** Speicher mit einem Adressbereich von **8 Bit** (2^8 Adressen) und einer Blockgrösse von **32 Bit**.

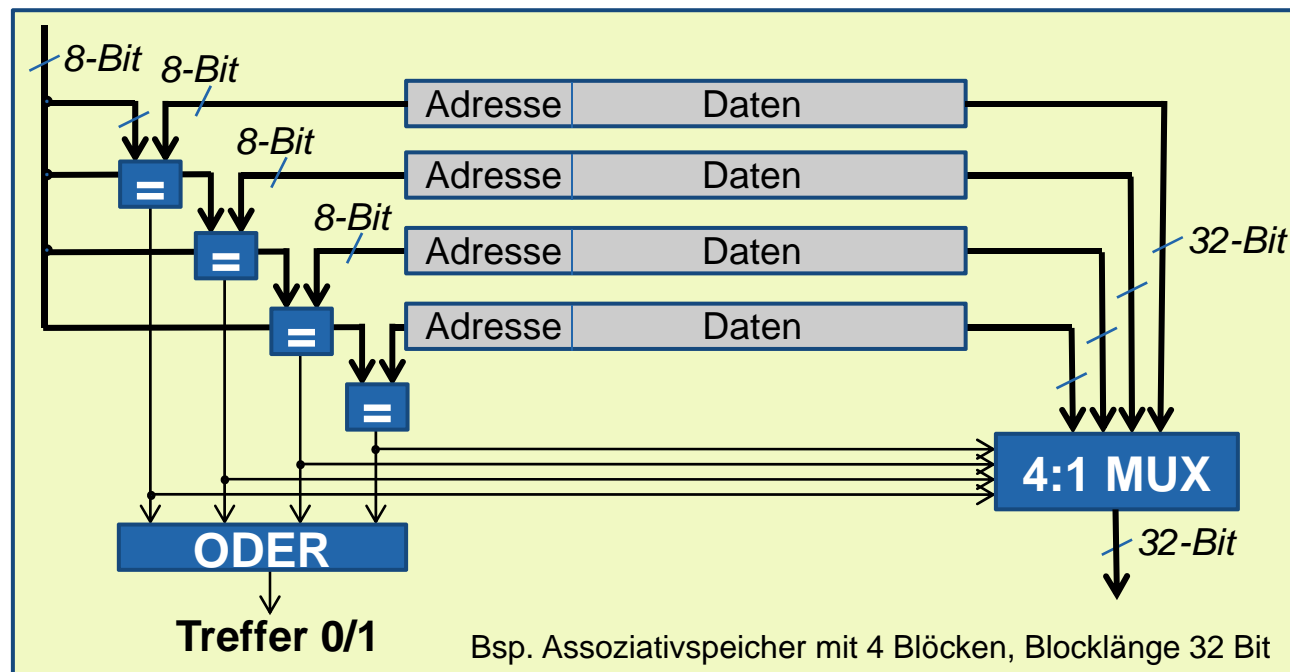
Über einen **Vergleicher** müssen die **8 Bit** der Adressen verglichen werden.



Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- Lesezugriffe – Grundansätze / Konzepte:
 - a) *Assoziativer* Zugriff – Grundprinzip (Bsp. Lesen)



Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- Lesezugriffe – Grundansätze / Konzepte:

- a) *Assoziativer* Zugriff

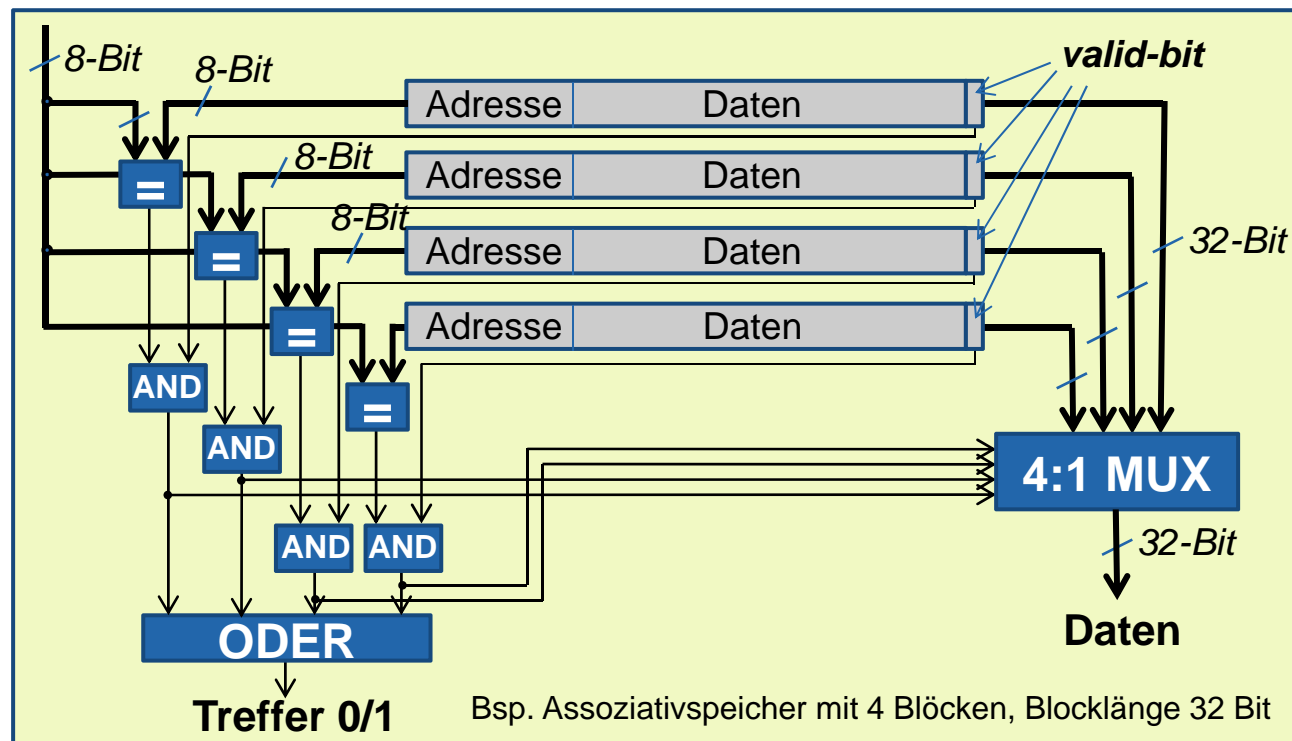
Frage: Wie wird erkannt, ob ein Datum (Block) im *Cache* gültig (*valid*) ist?

=> Jeder Block beinhaltet ein zusätzliche Bit (*valid bit*), das angibt, ob die Daten gültig sind oder nicht.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- Lesezugriffe – Grundansätze / Konzepte:
 - a) *Assoziativer* Zugriff – Grundprinzip (Bsp. Lesen)



Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) *Direktabbildend* – Schema:

- Bei einem *direktabbildenden Speicher* wird jeder **Adresse** im **Speicher** genau **eine Adresse (Position)** im *Cache* zugeordnet.

Die **Abbildung** ist **sehr einfach** – in der Regel über die „*modulo*“-Funktion realisiert:

$$\text{Cache-Adresse} = (\text{Block-Adresse}) \bmod (\text{Anzahl Blöcke im Cache})$$

(Grösse von **Speicher** und *Cache* sind i. d. R. ein **Vielfaches** von 2.)

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- **Lesezugriffe – Grundansätze / Konzepte:**
 - b) *Direktabbildend* – Grundprinzip (Bsp. Abbildung)
 - Arbeitsspeicher mit 32 Blöcken
 - Cache mit 8 Blöcken
 - Block z. B. 4 Byte

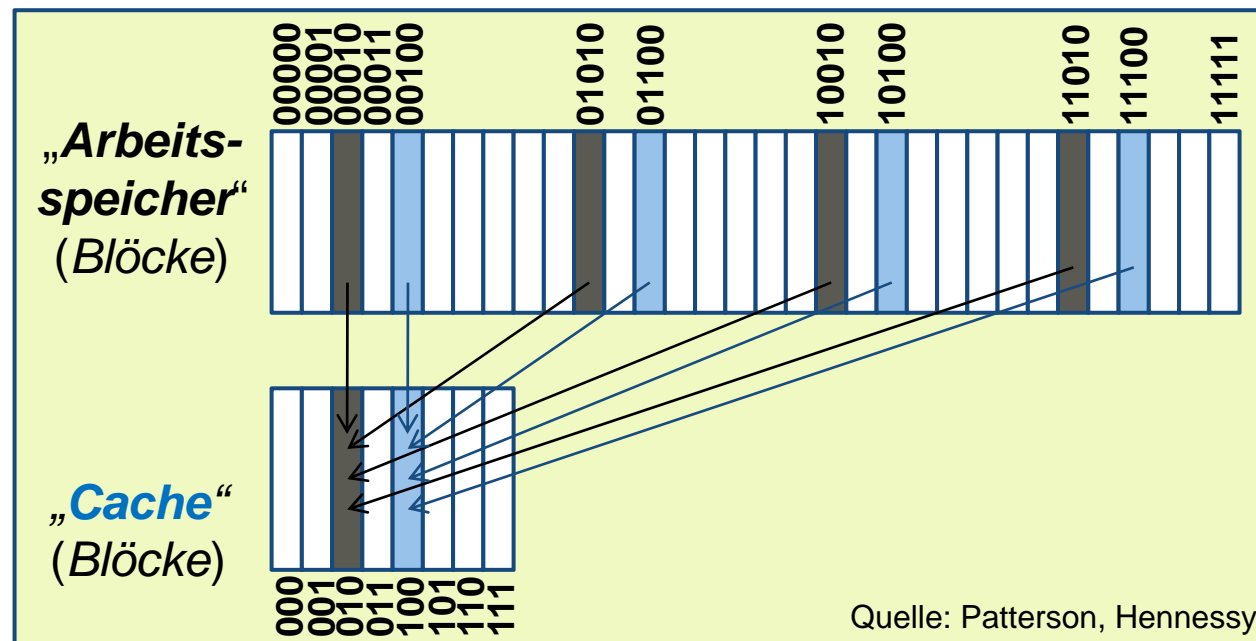
Die *Blockgrösse* ist i. d. R. ein **Vielfaches** der **Wortgrösse**:

$$\text{Blockgrösse} = \text{Wortgrösse} * 2^m, m \in \mathbb{N}$$

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- Lesezugriffe – Grundansätze / Konzepte:
 - b) *Direktabbildend* – Grundprinzip (Bsp. Abbildung)



Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) *Direktabbildend* – Grundprinzip (Bsp. Abbildung)

Der **Block** im **Arbeitsspeicher** mit der **Adresse 00010** wird auf den **Block** im **Cache** mit der **Adresse 010** abgebildet, ebenso die **Blöcke 01010, 10010** und **11010**.

- Um zu erkennen, ob ein **Block** im **Cache** den **gesuchten Daten entspricht**, wird wiederum ein **zusätzlicher „Tag“** mit **abgespeichert**, der einen Teil der **Blockadresse beinhaltet** (im Bsp. die Bits **0** und **1** der Blockadresse), und beim **Test abgeglichen**.
- Die **vollständige** Blockadresse ist **nicht notwendig**, da diese **partiell** im **Index** schon enthalten ist (im Bsp. die Bits **2, 3** und **4** der Blockadresse).

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) *Direktabbildend* – Grundprinzip (Bsp. Abbildung)

Cache mit *Tags* (des Beispiels):

- Gesucht der Eintrag für die Block-Adresse: „0 1 0 1 1“ Index

Beispiel für eine
Belegung im *Cache*

<i>Index</i>	<i>Tag</i>	<i>Datenblöcke</i>
0 0 0	0 0	„Daten: 32 Bit“
0 0 1	0 1	„Daten: 32 Bit“
0 1 0	1 1	„Daten: 32 Bit“
0 1 1	0 1	„Daten: 32 Bit“
1 0 0	0 1	„Daten: 32 Bit“
1 0 1	1 1	„Daten: 32 Bit“
1 1 0	1 1	„Daten: 32 Bit“
1 1 1	0 0	„Daten: 32 Bit“

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) *Direktabbildend* – Grundprinzip (Bsp. Abbildung)

Cache mit *Tags* (des Beispiels):

– Gesucht der Eintrag für die Block-Adresse: „0 1 0 1 1“

Tag

Index

Beispiel für eine
Belegung im *Cache*

Berechnung vom Index:

$$01011_2 \bmod 1000_2 = 011_2$$

Index	Tag	Datenblöcke
0 0 0	0 0	„Daten: 32 Bit“
0 0 1	0 1	„Daten: 32 Bit“
0 1 0	1 1	„Daten: 32 Bit“
0 1 1	0 1	„Daten: 32 Bit“
1 0 0	0 1	„Daten: 32 Bit“
1 0 1	1 1	„Daten: 32 Bit“
1 1 0	1 1	„Daten: 32 Bit“
1 1 1	0 0	„Daten: 32 Bit“

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) *Direktabbildend* – Beispiel Adressberechnung

Gegeben sei ein **Cache** mit **32 Blöcken** von je **4 Bytes**

- Auf **welchen Block** (**Tag**) im **Cache** wird das **Byte** im **Arbeitspeicher** mit der **Adresse 2410** abgebildet? (Beginn bei Adr. 0)

$$\text{Blockadresse} = \lfloor \text{Byteadresse} / (\text{Bytes pro Block}) \rfloor$$

$$\text{Blockadresse} = \lfloor (2410+1) / 4 \rfloor = \lfloor 602.75 \rfloor = 602$$

$$\text{Cache-Blocknummer (Tag)} = (\text{Blockadresse}) \bmod (\text{Anzahl Blöcke im Cache})$$

$$\text{Cache-Blocknummer (Tag)} = 602 \bmod 32 = 26$$

(es ist dort das **4. Byte**; **2407** ist das **1. Byte**)

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- Lesezugriffe – Grundansätze / Konzepte:

- b) *Direktabbildend* – Beispiel Grössenberechnung

Gegeben sei für einen **Arbeitsspeicher** mit 2^{32} Byte ein *direktabbildender Cache* mit **4 KiB Daten⁽⁺⁾**, einer **Blockgrösse** von **4 Wörtern** und einer **Wortlänge** von **32 Bit (4 Byte)**.

➤ **Wie gross** ist die **tatsächliche Anzahl** von **Bits** für den *Cache*?

(+)Zur Erinnerung: **4 KiB** bedeuten exakt $2^{12} = 4096$ Byte, leider wird umgangssprachlich auch häufig **4 KB** gesagt.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) *Direktabbildend* – Beispiel Grössenberechnung

➤ Wie gross ist die tatsächliche Anzahl von Bits für den *Cache*?

=> *Blockanzahl* * (*Blockgrösse* + *Tag*-Grösse + *Flag*-Bits)

▪ *Blockgrösse*: $4 * 4 \text{ Byte} = 2^4 \text{ Byte} = 16 \text{ Byte} (= 2^7 \text{ Bit})$

▪ *Blockanzahl* (= Nr. Indexe): $2^{12} / 2^4 \text{ Byte} = 2^8 \text{ Byte} = 256 \text{ Blöcke}$

▪ Wörter pro Block: $4 \text{ Byte} = 2^2 \text{ Byte}$

▪ Byte pro Wort: $4 \text{ Byte} = 2^2 \text{ Byte}$

=> Grösse *Tag*-Feld: $32 - 8 - 2 - 2 \text{ Bit} = 20 \text{ Bit}$

▪ *Valid*-Bit pro Block : 1 Bit

=> $ca. 2^8 * (2^7 + 20 + 1) \text{ Bit} = 38'144 \text{ Bit} = 4'768 \text{ Byte}$

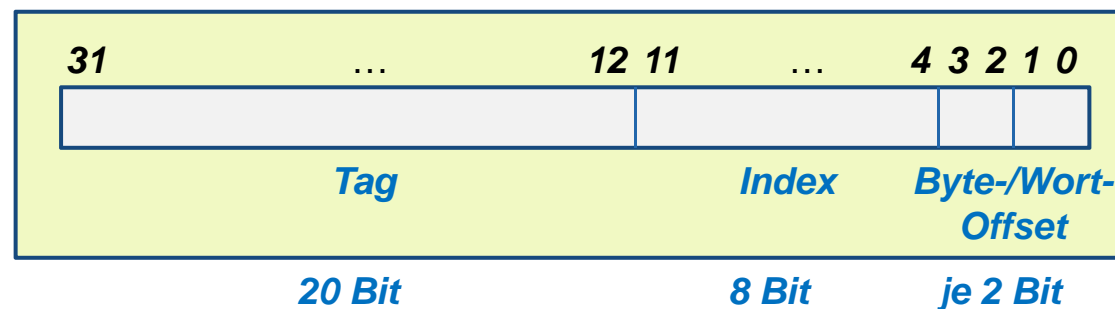
Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) *Direktabbildend* – Beispiel Grössenberechnung

➤ Wie gross ist die tatsächliche Anzahl von Bits für den *Cache*?

Blockgrösse: „*Aufteilung der Adresse*“: (für das vorherige Beispiel)



Architekturelemente - Speicher

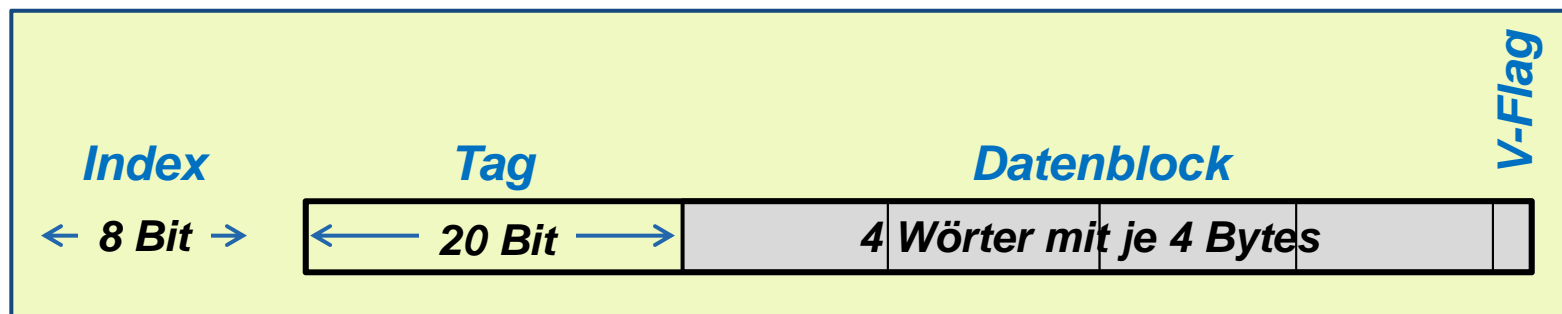
Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) *Direktabbildend* – Beispiel Grössenberechnung

- Wie gross ist die **tatsächliche Anzahl** von **Bits** für den *Cache*?

Aufbau eines Datenblocks: (für das vorherige Beispiel)



Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

a) + b) Vergleich: *Assoziativ* ↔ *Direktabbildend*

Cachestruktur	assoziativ	direktabbildend
Zugriffszeit (bei Treffer)	höher	geringer
Trefferrate (im Durchschnitt)	höher	geringer
Komplexität = Kosten	gross	geringer

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

c) *Satzassoziativ*

- Ist eine **Mischform** von einem *direktabbildenden Cache* und *einem assoziativen Cache*.
- Ein **Block** kann **statt auf genau einem Platz**, wie bei einem direktabbildenden *Cache*, auf eine **vorgegebene Anzahl von Positionen** (i. d. R. Vielfaches von **2**, wird als **Satz** bezeichnet) **beliebig abgelegt werden**.
- **Suche in einem *satzassoziativen Cache*:**
 - Ein **Block** wird *direkt* auf einen **Satz** abgebildet und
 - **innerhalb der Positionen eines Satzes *assoziativ* gesucht**.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

c) *Satzassoziativ*: Beispiel

- 2 Positionen (*2-fach satzassoziativer Cache*)
- 4 Sätze (zu Blöcken von je 32 Bit)

Satz-Index	Tag	Datenblöcke	v-Bit
0	0 0	„Daten: 32 Bit“	
	0 1	„Daten: 32 Bit“	
1	1 1	„Daten: 32 Bit“	
	0 1	„Daten: 32 Bit“	
2	0 1	„Daten: 32 Bit“	
	1 1	„Daten: 32 Bit“	
3	1 1	„Daten: 32 Bit“	
	0 0	„Daten: 32 Bit“	

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

c) *Satzassoziativ*

- Typische Satzgrösse: 2 oder 4 Positionen
(*2-* bzw. *4-fach satzassoziativer Cache*)

=> Mehrkosten bzgl. Zeit und zusätzlicher Hardware
vertretbar ...

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

c) *Satzassoziativ*

- Typische Satzgrösse: 2 oder 4 Positionen
(*2-* bzw. *4-fach satzassoziativer Cache*)

... und heuristisch wurde gezeigt, dass eine grössere Assoziativität ev. nur noch wenig bringt!

Quelle (Daten):
Hennessy / Patterson
[2003]

<i>n-fach satzassoziativ</i>	Datenfehl- zugriffsrate	Verbesserung (abs. / rel.)
1	10.3 %	-
2	8.6 %	16.5 % (-)
4	8.3 %	19.4 % (3.5 %)
8	8.1 %	21.4 % (2.5 %)

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

c) *Satzassoziativ*

- Mehrkosten für *n-fach satzassoziativen Cache* entstehen insb. durch die erforderlichen **Vergleicher** (Chip-Fläche für Hardware) und **längere Laufzeiten**, da der **kritische Pfad grösser** wird.
- Mit neuen **CAM** (**Content Addressable Memory**) auch teilweise *8/16-fach satzassoziativer Cache* implementiert.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

c) *Satzassoziativ*

- Ein *1-fach satzassoziativer Cache* ist ein Sonderfall und entspricht einem *Cache* mit *direktem Zugriff*.
- Ein *n-fach satzassoziativer Cache* bei einem *Cache* mit insgesamt *n* Blöcken ist ebenfalls ein Sonderfall und entspricht einem *Cache* mit *voll-assoziativem Zugriff*.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) + c) Verdrängung bei *assoziativ* und *satzassoziativ*

- Wenn **alle Blöcke** in einem *assoziativen Cache* (alle **Blöcke** in einem **Satz**) **belegt** sind, **muss**, wenn ein **neues Datum** in den *Cache* aufgenommen werden soll, ein **bestehender Eintrag** (Blockeintrag) „**verdrängt**“ (d. h. überschrieben) werden.

=> Dafür gibt es grundsätzliche drei Ansätze:

- *Least recently used (LRU)*
- *Least frequently used (LFU)*
- *First in first out (FIFO)*

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) + c) Verdrängung bei *assoziativ* und *satzassoziativ*

– *Least Recently Used (LRU)*:

Es wird der **Block entfernt** (überschrieben), dessen **Zugriff am längsten zurückliegt**.

Realisierung: z. B. über einen **Zugriffszähler**

– *Least Frequently Used (LFU)*

Es wird der **Block entfernt** (überschrieben), auf den am **wenigsten häufig zugegriffen** wurde.

Realisierung: ebenfalls z. B. über einen **Zugriffszähler**

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

b) + c) Verdrängung bei *assoziativ* und *satzassoziativ*

– *First In First Out (FIFO)*:

Es wird der **Block entfernt** (überschrieben), der zeitlich am **längsten** im *Cache* liegt.

Realisierung: z. B. einfach über ein *Flag*

– Weitere: *Random, Climb, Clock, ...*

Auch wenn *FIFO* nicht so hohe Trefferraten wie die anderen Ansätze erreicht, wird dieser Ansatz dennoch **sehr häufig genutzt**.

=> Der Grund ist die einfache Realisierung!

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

c) *Satzassoziativ* – Beispiel Grössenberechnung

Gegeben sei für einen **Arbeitsspeicher** mit 2^{32} Byte ein *4-fach satzassoziativer Cache* mit **4 KiB Daten**, einer **Blockgrösse** von **4 Wörtern**, einer **Wortlänge** von **32 Bit** und *FIFO* als **Verdrängungsmethode** (= 1 zusätzliches Bit pro Block).

➤ Wie gross ist die **tatsächliche Anzahl** von **Bits** für den *Cache*?

=> **Satzanzahl** * (**Satzgrösse** + *Tag-Grösse* + *Flag-Bits*)

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

c) *Satzassoziativ* – Beispiel Grössenberechnung

➤ **Wie gross** ist die **tatsächliche Anzahl** von **Bits** für den **Cache?**
(Näherungsweise)

- Blockgrösse: $4 * 32 \text{ Bit} = 2^7 \text{ Bit} = 128 \text{ Bit} (= 2^4 \text{ Byte})$
- **Satzgrösse:** $4 * 4 * 32 \text{ Bit} = 2^9 \text{ Bit} = 512 \text{ Bit} (= 2^6 \text{ Byte})$
- (Blockanzahl: $2^{12} \text{ Byte} / 2^4 \text{ Byte} = 2^8 = 256 \text{ Blöcke}$)
- **Satzanzahl:** $2^{12} \text{ Byte} / 2^6 \text{ Byte} = 2^6 = 64 \text{ Sätze}$
- **Wörter pro Block:** $4 \text{ Byte} = 2^2 \text{ Byte}$
- **Byte pro Wort:** $4 \text{ Byte} = 2^2 \text{ Byte}$

=> Tag-Feld: $32 - 6 - 2 - 2 \text{ Bit} = 22 \text{ Bit}$

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Grundansätze / Konzepte:

c) *Satzassoziativ* – Beispiel Grössenberechnung

➤ **Wie gross** ist die **tatsächliche Anzahl** von **Bits** für den **Cache?**
(Näherungsweise)

- **Valid-Bit** pro Block: **1 Bit**
- **Flag** für **FIFO**-Realisierung pro Block: **1 Bit**

$$\begin{aligned} \Rightarrow & \text{Satzanzahl} * (\text{Satzgrösse} + \text{Tag-Grösse} + 1 + 1) \\ & = 2^8 * (2^7 + 22 + 1 + 1) \text{ Bit} = 38'912 \text{ Bit} = \text{ca. } 4'864 \text{ Byte} \end{aligned}$$

(Vereinfacht - in der Praxis zuzüglich weiterer **Flags**)

[Zur Erinnerung: Die eigentliche **Mehrfläche** wird durch die notwendige zusätzliche Logik (Vergleicher) verursacht.]

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Optimierung:

– „*Early-Restart*“

Ziel: Reduktion des Fehlzugriffsaufwands

- Nach einem **Fehlzugriff** wird die **Programmausführung bereits fortgesetzt**, sobald das **angeforderte Wort** des **Blocks** geladen wurde (statt auf das **Laden** des **ganzen Blocks** zu warten); die folgenden Wörter werden dann „*just in time*“ geliefert.
- Funktioniert bei **Zugriffen** auf **Befehle** und **grosser Blockgrösse recht gut, weniger gut** allgemein für **Datenzugriffe** (Zugriffe auf Befehle erfolgen häufig **sequentiell**).

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Lesezugriffe – Optimierung:

- „*Requested Word First*“ bzw. „*Critical Word First*“

Ziel: Reduktion des Fehlzugriffsaufwands

- **Komplexer als *Early-Restart*** insofern, dass zusätzlich zum Ansatz von *Early-Restart* nach einem **Fehlzugriff** das **angeforderte Wort zuerst geladen** wird und **anschliessend** erst die **restlichen Wörter des Blocks**.
- **Schneller als *Early-Restart***, wenn das **angeforderte Wort** erst **später im Block** auftritt.
- **Dafür aufwendiger zu implementieren** (zusätzliche Logik und Speicherorganisation).

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Schreibzugriffe – Grundansätze / Konzepte:

➤ **Schreibzugriffe sind grundsätzlich komplexer als Lesezugriffe!**

➤ Wenn nach einem **Schreibzugriff auf Daten** diese **nur** in den **Cache** geschrieben werden, **enthalten Speicher und Cache unterschiedliche Daten**.

=> Cache und Speicher sind *inkonsistent*.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- **Schreibzugriffe – Grundansätze / Konzepte:**

- Einfachste Bearbeitung von Schreibzugriffen:

- „**Write-Through**“ (*Durchschreibetechnik*)

- Bei **jedem Schreibvorgang** werden die **Daten sowohl** in den **Cache** als auch in den **Speicher** geschrieben.

- => **Cache** und **Speicher** sind immer **konsistent** !

- **Sehr einfach zu realisieren**, jedoch **sehr schlechte Performance**, da bei **jedem Schreibzugang** auf den **Arbeitsspeicher** zugegriffen werden muss!

[Bei einem **Fehlzugriff** wird zuerst der **Block aus dem Speicher** in den **Cache** geladen und anschliessend sowohl in den **Cache** wie auch in den **Speicher** geschrieben.]

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- **Schreibzugriffe – Grundansätze / Konzepte:**

- **Effizientere Bearbeitung von Schreibzugriffen:**

Verwendung von „**Write Puffer**“ (*Schreibpuffer*)

- Bei einem **Schreibvorgang** werden die **Daten** in den **Cache** und einen in **Schreibpuffer** geschrieben und das **Programm dann bereits fortgesetzt**.
 - Aus dem **Schreibpuffer** werden die **Daten** dann im **Hintergrund** in den **Speicher geschrieben** und anschliessend der **Eintrag** aus dem **Schreibpuffer entfernt**.
 - ...

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- **Schreibzugriffe – Grundansätze / Konzepte:**

- **Effizientere Bearbeitung von Schreibzugriffen:**

Verwendung von „**Write Puffer**“ (*Schreibpuffer*) – Forts.

- Das **Schreiben** in den **Schreibpuffer** erfolgt **deutlich schneller** als in den **Speicher**.
- **Grösse des Schreibpuffers** ist für **Performance** wichtig.
- Bei sehr vielen **unmittelbar aufeinanderfolgenden Schreibzugriffen** („*bursts*“) kann der **Schreibpuffer** diese **nicht** mehr aufnehmen und muss „**warten**“.
- [**Konflikte** mit „**DMA**“-Zugriffen müssen berücksichtigt werden.]

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- **Schreibzugriffe – Grundansätze / Konzepte:**
 - Weiterer Ansatz: „**Write Back**“ (*Rückschreibetechnik*)
 - Bei einem **Schreibvorgang** werden die **Daten** nur in den **Cache** geschrieben und das **Programm dann bereits fortgesetzt** und der **Block** mit einem **Flag**, „**dirty bit**“, markiert.
 - Erst wenn der **Block im Cache überschrieben** wird, erfolgt (unmittelbar zuvor) das **Schreiben in den Speicher** (ev. mit **Schreibpuffer**).

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Schreibzugriffe – Grundansätze / Konzepte:

– Weitere Unterscheidung:

- *Write Allocation* und
- *No Write Allocation*

Wenn es bei einem **Schreibvorgang** zu einem **Fehlzugriff** kommt, wird **direkt** in den **Speicher geschrieben**, **nicht** aber in den *Cache*.

Hintergrund ist

- Die **bessere Performance** (und die einfachere Handhabung), wenn **ganze Seiten im Speicher ausgetauscht** werden.
- Im *Cache* werden keine anderen, ev. wichtigeren Einträge **verdrängt**.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- Anmerkungen und allgemeine Optimierungen
 - Kombination der verschiedenen Ansätze für **Lese-** und **Schreibzugriffe**
 - Unterschiedliche Handhabung (insb.) der **Schreibzugriffe**

ACHTUNG: **DMA-Zugriffe** müssen mit **einbezogen** bzw. **berücksichtigt** werden.

DMA (**Direct Memory Access**) gestattet **Peripheriegeräten** den **direkten Zugriff** auf **Daten** im **Speicher**, ohne dass der **Prozessor** damit belastet wird, z. B. Kopie von **Daten** im **Speicher** auf eine **CD**.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

- Anmerkungen und allgemeine Optimierungen

- Beobachtung zur **Blockgrösse**:

Fehlzugriffsrate sinkt zunächst bei grösseren Blöcken,

=> räumliche Lokalität wird **besser genutzt**

steigt aber bei zu grossen Blöcken wieder an, weil

=> bei sehr grossen Blöcken (bei gleichgrossem **Cache**) die Anzahl der Blöcke **abnimmt** und ein Block häufiger im **Cache verdrängt** wird,

=> eventuell überhaupt **nicht** auf alle Wörter eines Blocks zugegriffen wird (zuvor schon wieder verdrängt).

Zudem **steigt** der Fehlzugriffsaufwand mit der Blockgrösse.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

■ Anmerkungen und allgemeine Optimierungen

– Getrennte *Caches* für Befehle und Daten:

- Unterschiedliche Ansätze für **Optimierung** der **Lese-** und **Schreibzugriffe** einfach realisierbar.
- Unterschiedliche Wortbreite (z. B. *Cache* für Befehle kleiner) und **Kapazitäten / Grössen**.
- Positiver Nebeneffekt: **Erhöhung** der **Bandbreite**
- Cachekapazität muss **aufgeteilt** werden.

=> Führt zu geringfügig erhöhten Trefferraten.

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Anmerkungen und allgemeine Optimierungen

– **Mehrstufige Caches** (heute häufig **zweistufig** realisiert):

- Ein **kleiner, sehr schneller Cache**; **direkt im Prozessorkern implementiert** (einfache Realisierung).

=> **Optimierung** der **Hit Time**

[Arbeiten i. d. R. synchron (1:1) zur Prozessorgeschwindigkeit, so dass bei einem Treffer überhaupt nicht gewartet werden muss.]

- Ein **grösserer** etwas **langsamerer zweiter Cache** auf dem Mainboard, der häufig **deutlich komplexer** ist.

=> **Optimierung** des **Fehlzugriffsaufwandes**

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Beispiel für **Cache**-Leistung (a)

- **Prozessor mit 4 GHz** (entspricht 0.25 ns Zykluszeit)
- **CPI = 1** für Befehle, die nicht auf den Speicher zugreifen (im \emptyset),
CPI = 50 für Speicherbefehle (**Load / Store**) ($= 50 * 0.25\text{ ns} = 12.5\text{ ns}$).
- Bei ca. **40 %** der Befehle wird auf den Speicher zugegriffen (im \emptyset).
- „**On-Chip**“-**Cache**:
 - **2-fach satzassoziativ** (sehr schnell, aber teuer)
 - **Hit-Rate** von **96%** (statistisch ermittelt)
 - **Hit-Time** von **einem Taktzyklus**
 - **Fehlzugriffsaufwand: 100 Zyklen** ($= 100 * 0.25\text{ ns} = 25\text{ ns}$)

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Beispiel für **Cache**-Leistung (a) (Forts.)

Wie lange dauert die Bearbeitung eines Befehls im Durchschnitt?

- Ohne **Cache**: $t_{ohneC} = (1 - A_S) * T_C + A_S * T_S$
 $= (0.6 * 0.25 \text{ ns} + 0.4 * 12.5 \text{ ns}) = 5.15 \text{ ns}$
- Mit **Cache**: $t_{mitC} = (1 - A_S) * T_C + A_S * (R_{hit} * t_{hit} + (1 - R_{hit}) * t_{miss})$
 $= 0.6 * 0.25 \text{ ns} + 0.4 * (0.96 * 0.25 \text{ ns} + 0.04 * 25 \text{ ns})$
 $= 0.646 \text{ ns}$

=> Leistungssteigerung durch den **Cache**: $(5.15 / 0.646) - 1 \cong 800\%$

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Beispiel für **Cache**-Leistung (b)

- Derselbe Prozessor wird nun mit bei identischen Gesamtkosten mit zwei **Caches** ausgestattet (fiktives Beispiel):

- **Level-1-Cache** („**On-Chip**“): **Direktabbildend** (klein, schnell)
 - **Hit-Rate** von „nur“ noch **92%** (statistisch ermittelt); u. a. da kleiner
 - **Hit-Time** von einem Taktzyklus
- **Level-2-Cache** („**On-Board**“): **4-fach satzassoziativ** (gross, langs.)
 - **Hit-Rate** von **80%** (statistisch ermittelt) für die **Fehlzugriffe** des **Level-1-Cache**
 - **Hit-Time** von **6 Taktzyklen** ($= 6 * 0.25 \text{ ns} = 1.5 \text{ ns}$)
 - **Fehlzugriffsaufwand** von **100 Zyklen** ($= 100 * 0.25 \text{ ns} = 25 \text{ ns}$)

Architekturelemente - Speicher

Caches – Realisierung / Umsetzung

▪ Beispiel für **Cache**-Leistung (b) (Forts.)

Wie lange dauert die Bearbeitung eines Befehls im Durchschnitt?

$$\begin{aligned}
 \text{– Mit } \text{Cache}_{2\text{-Level}}: t_{\text{mitC2}} &= (1 - A_S) * T_C + A_S * (R_{\text{hit-C1}} * t_{\text{hit-C1}} + (1 - R_{\text{hit-C1}}) \\
 &\quad * (R_{\text{hit-C2}} * t_{\text{hit-C2}} + (1 - R_{\text{hit-C2}}) * t_{\text{miss-C2}})) \\
 &= 0.6 * 0.25 \text{ ns} + 0.4 * (0.92 * 0.25 \text{ ns} + 0.08 * \\
 &\quad (0.8 * 1.5 \text{ ns} + 0.2 * 25 \text{ ns})) \\
 &= 0.4404 \text{ ns}
 \end{aligned}$$

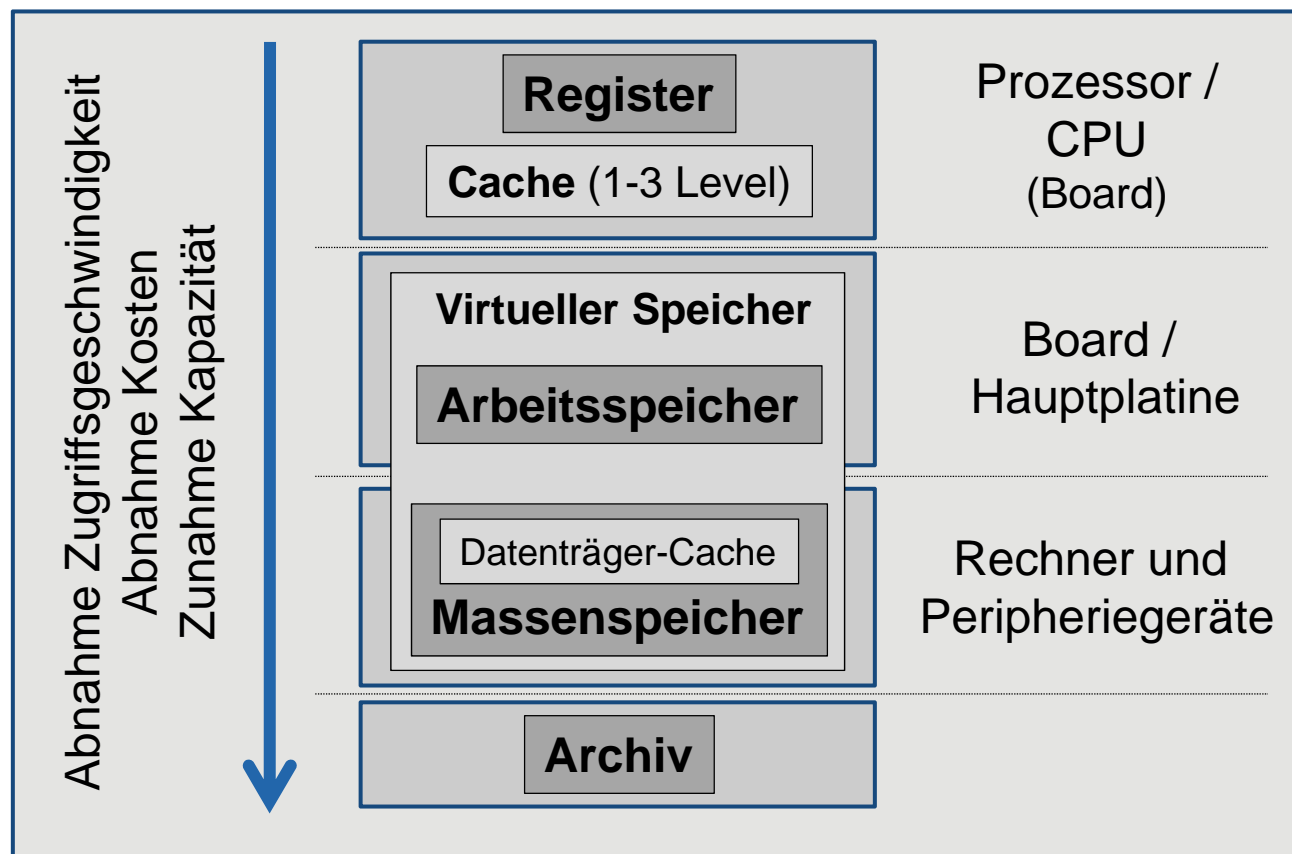
=> Leistungssteigerung gegenüber 1-stufigem **Cache**:

$$(0.646 \text{ ns} / 0.4404 \text{ ns}) - 1 \cong 47\%$$

[bzw. $(5.15 \text{ ns} / 0.4404 \text{ ns}) - 1 \cong 1170\%$ gegenüber keinem **Cache**]

Architekturelemente - Speicher

Caches – Virtueller Speicher: Einordnung



Architekturelemente - Speicher

Caches – Virtueller Speicher

- Analog der Funktion eines *Caches* für den *Arbeitsspeicher* kann der *Arbeitsspeicher* als „*Cache*“ für den *Massenspeicher* dienen !
- Wesentlicher Unterschied:
 - Dieser „*Cache*“ wird primär durch den Arbeitsspeicher selber, Software und wenig zusätzlicher Hardware realisiert – es gibt keinen physikalischen *Cache*.

Architekturelemente - Speicher

Caches – Virtueller Speicher

- Analog der Funktion eines *Caches* für den *Arbeitsspeicher* kann der *Arbeitsspeicher* als „*Cache*“ für den *Massenspeicher* dienen !
- Wesentlicher Unterschied:
 - Dieser „*Cache*“ wird primär durch den Arbeitsspeicher selber, Software und wenig zusätzlicher Hardware realisiert – es gibt keinen physikalischen *Cache*.
 - Die Entwicklung wurde – neben dem Ziel eines schnelleren Zugriffs – auch durch „weitere Aufgaben / Funktionen“ ausgelöst und vorangetrieben.

Architekturelemente - Speicher

Caches – Virtueller Speicher

- **Weiteren Aufgaben / Funktionen:**

- Die gemeinsame **sichere Nutzung** des Speichers durch **viele** Programme (gleichzeitig).

Auf einem Rechner sind immer mehrere Programme (Prozesse) gleichzeitig aktiv: Jedes Programm darf nur den Teil des Speicher lesen und insbesondere verändern, der ihm zugeordnet ist.

Architekturelemente - Speicher

Caches – Virtueller Speicher

- **Weiteren Aufgaben / Funktionen:** (Forts.)
 - Die **Vergrößerung** des **physikalischen** Speichers

a) Für ein Programm:

Ein **Programm** kann für seine **Ausführung** mehr **Speicherplatz** erfordern, als **physikalisch zur Verfügung steht** – dieses war in der Vergangenheit häufig der Fall und musste durch den Programmentwickler mühsam durch die „**overlay**“-Technik umgangen werden.

Architekturelemente - Speicher

Caches – Virtueller Speicher

- **Weiteren Aufgaben / Funktionen:** (Forts.)
 - Die **Vergrößerung** des **physikalischen** Speichers

- b) Für mehrere Programme:

Der **erforderliche Speicher** für **viele hundert** oder **gar tausend Programme** ist schnell **größer** als der **tatsächlich vorhandene physikalische Arbeitsspeicher**.

Sie gaben dem **Ansatz / der Technik** auch den heute üblicherweise verwendeten Begriff: „**virtueller Speicher**“ ...

... auch wenn die **Grundansätze und -methoden** identisch mit denen für einen **Cache für den Arbeitsspeicher** sind !

Architekturelemente - Speicher

Caches – Virtueller Speicher

▪ „Bezeichnungen“

- Statt „**Cache**“ heisst es „**virtueller Speicher**“
- Ein **virtueller Speicherblock** wird „**Seite**“ (engl. „**page**“) genannt.
- Ein Speicherfehlzugriff auf eine Seite entsprechend „**Seitenfehler**“ (engl. „**page fault**“).
- Adressen des **virtuellen Speichers** heissen „**virtuelle**“ **Adressen** und müssen für den tatsächlichen Zugriff in eine „**physikalische Adresse**“ übersetzt werden:
 - => Dieser Prozess heisst: **Adressabbildung** bzw. **Adress-übersetzung** (engl. „**paging**“).

Architekturelemente - Speicher

Caches – Virtueller Speicher

- „**Paging**“ und „**Segmentierung**“

- a) „**Segmentierung**“

- Es werden Blöcke variabler Länge im Arbeitsspeicher genutzt; diese Blöcke heissen „**Segmente**“.
 - Die **virtuelle Adresse** setzt sich aus einer **virtuellen Segment-Nummer** und einem **Segment-Offset** zusammen.

=> Bei der **Abbildung** auf die **physikalische Adresse** müssen **Segmentnummer** und **Segment-Offset** übersetzt / berücksichtigt werden, da das **Offset** unterschiedlich **gross** sein kann und die **Grenzen überprüft** werden müssen.

Architekturelemente - Speicher

Caches – Virtueller Speicher

■ „Paging“ und „Segmentierung“

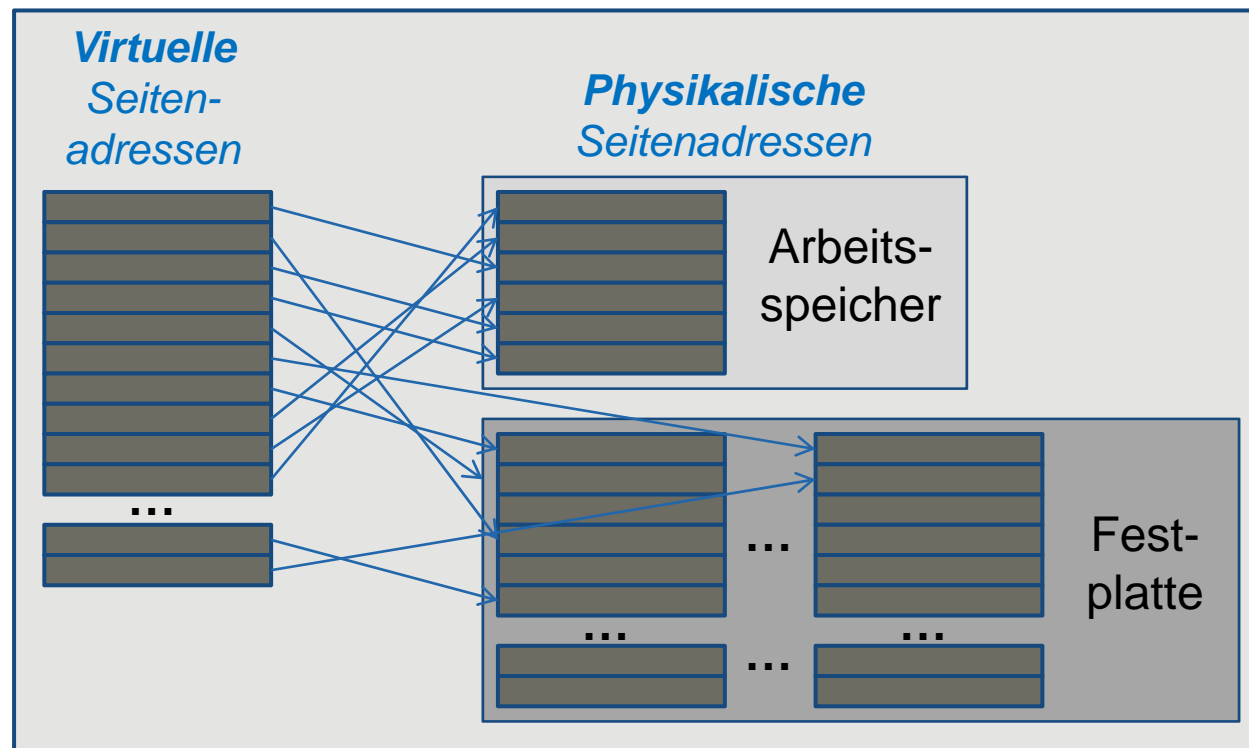
b) „Paging“

- Es werden Blöcke fester Länge im *Arbeitsspeicher* genutzt; diese Blöcke heissen „(Page) Frames“.
- Die *virtuelle* Adresse setzt sich aus einer virtuellen *Seitenadresse* und einem *Seiten-Offset* zusammen.
 - => Bei der **Abbildung** auf die **physikalische Adresse** muss nur die *Seitenadresse* übersetzt werden, da das *Seiten-Offset* immer gleich gross ist.
- Eine „*Fragmentierung*“ des Speichers ist ausgeschlossen.
 - => Heute fast ausschliesslich Nutzung von „Paging“.

Architekturelemente - Speicher

Caches – Virtueller Speicher

- Schema der *Adressabbildung* („Paging“)



Architekturelemente - Speicher

Caches – Virtueller Speicher

- „**Reallocation**“:

Ein Programm (Prozess) muss nicht mehr fortlaufend im **Arbeitspeicher** (physikalisch) stehen, es muss nur über (auf) ausreichend viele **Seiten** im **Arbeitsspeicher** verfügen (zugreifen können), die im **virtuellen Speicher** fortlaufend als ein Block „geladen“ werden.

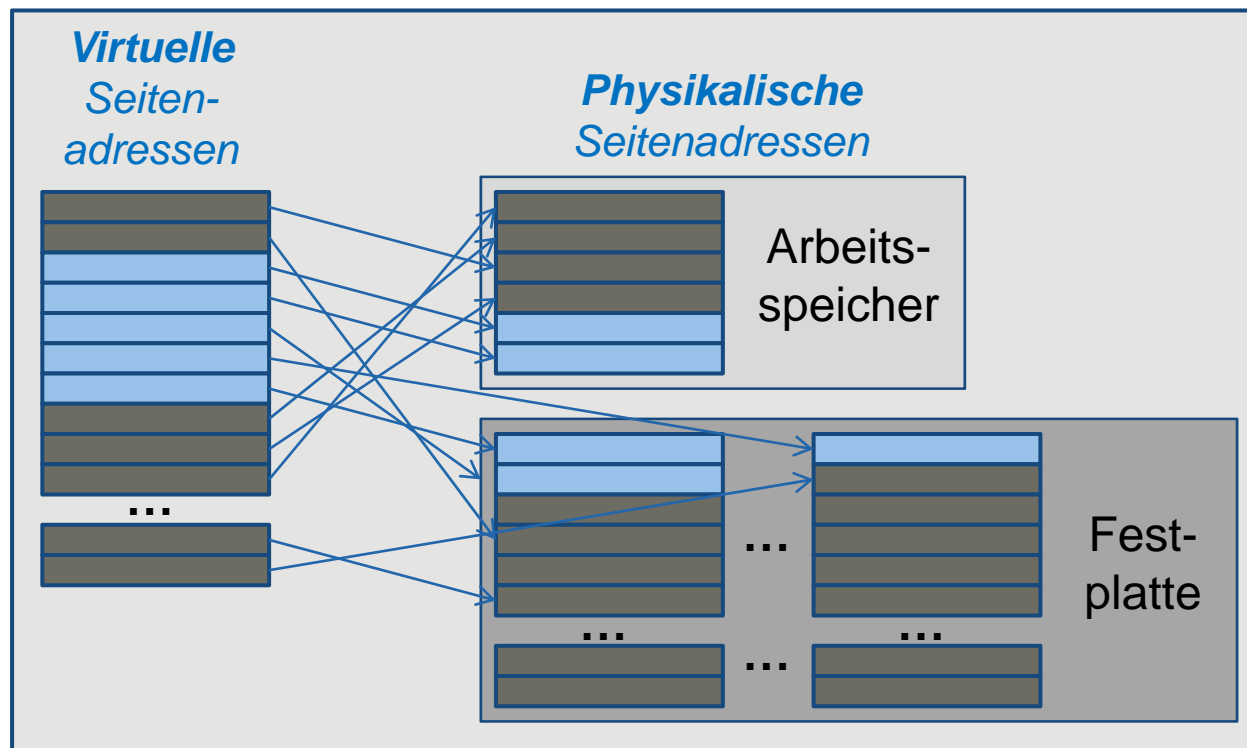
=> Laden eines Programms in jeden beliebigen Teil des **Arbeitsspeichers**.

(Für den Programmierer *transparent*)

Architekturelemente - Speicher

Caches – Virtueller Speicher

▪ Beispiel zu *Reallocation*



Architekturelemente - Speicher

Caches – Virtueller Speicher

- Beim Entwurf ist insbesondere zu berücksichtigen, dass ein Fehlzugriff (**Seitenfehler**) **sehr hohe Kosten** zur Folge hat !
(über 10^6 Taktzyklen!)
 - **Grosse Seiten**, um die **langen Zugriffszeiten** für das (Nach-) **Laden** zu „**nutzen**“ – nicht dass nach wenigen Zugriffen erneut geladen werden muss.
 - **Assoziatives Konzept** für das **Laden** und **Suchen** der **Seiten**, um die **Seitenfehlerrate** zu reduzieren (voll- oder zumindest sehr hoch assoziativ).
 - **Steuerung** (auch für **Seitenfehler**) **kann mit Software realisiert werden** (Zugriff auf Festplatte deutlich länger).
 - **Nutzung** von **Rückschreibetechniken** (Durchschreibetechniken wäre um Grössenordnungen zu langsam).

Architekturelemente - Speicher

Caches – Virtueller Speicher

- „**Seitentabelle**“ (engl. „**Page table**“)
 - Wie kann bei einem vollständig **assoziativen** Platzieren eine **Seite** im **virtuellen Speicher** gefunden werden?

(die Seite kann an jeder Position stehen und eine sequentielle Suche ist sicherlich nicht effizient).

=> In einer „**Seitentabelle**“ (individuell für jeden Prozess) wird diese Zuordnung festgehalten.

Bei modernen Rechnern kann die **Seitentabelle** mehrstufig (bis zu **vier Stufen**) aufgebaut sein, um sehr effizient mit **kleinen Tabellen** auch sehr **grosse virtuelle Speicher** zu verwalten.

Architekturelemente - Speicher

Caches – Virtueller Speicher

- „**Seitentabelle**“ (engl. „**Page table**“)
 - Die **Seitentabelle** selber ist auch im **virtuellen Speicher** abgelegt.

Wie wird die nun gefunden?

=> In einem ausgezeichneten Register, dem „**Seitentabellenregister**“, wird festgehalten, wo für das gerade ausgeführte Programm (Prozess) sich die **Seitentabelle** im **virtuellen Speicher** befindet.

Als positiver „**Nebeneffekt**“ kann so sehr einfach und effizient ein **laufendes Programm** unterbrochen werden:

=> Es müssen nur das **Seitentabellenregister**, der **Befehlszähler** und die weiteren **Register abgespeichert** werden (nicht die ganze **Seitentabelle** oder gar der ganze genutzte **Arbeitsspeicher**!).

Architekturelemente - Speicher

Caches – Virtueller Speicher

- „**Translation Lookaside Buffer**“ (TLB)

- Da auf die **Seitentabellen** recht häufig zugegriffen wird, sollte die dafür erforderliche Zugriffszeit so gering wie möglich sein!

=> In einem speziellen **Cache**, dem „**Translation Lookaside Buffer**“, kann die **Zugriffzeit** auf einen **Prozessorzyklus** im **Idealfall reduziert** werden!

Beim **Translation Lookaside Buffer** handelt es sich um einen „**echten**“, in Hardware realisierten Cache.

Architekturelemente - Speicher

Caches – Virtueller Speicher

- „**Austauschspeicher**“ (engl. „**swap space**“)
 - Für jedes Programm (Prozess) wird beim Start für alle erforderlichen Seiten des Programms entsprechender Platz auf der Festplatte reserviert, der als **Austauschspeicher** bezeichnet wird.

Architekturelemente - Speicher

Caches – Virtueller Speicher

▪ Austausch von Seiten

- Strategie für den Austausch von *Seiten*:
 - => Rückschreibetechnik mit *Least recently used*
- Rückschreiben von *Seiten* ist effizienter als das einzelne Wörter.
(Festplatte => Suchzeit ist sehr viel grösser als Übertragungszeit.)
- Es wird nur zurückgeschrieben, wenn *dirty bit* gesetzt.

Architekturelemente - Speicher

Zürcher Hochschule
für Angewandte Wissenschaften

zhaw School of
Engineering

