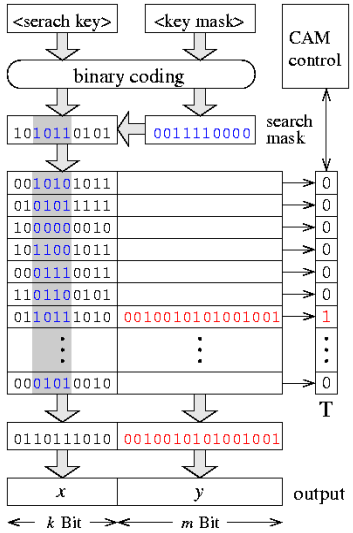
6 Exkurs: Assoziativspeicher

- alternative Möglichkeit der Speicherung von Informationen in einem Computer: **Assoziativspeicher** (**inhaltsadressierbarer Speicher** bzw. **CAM** = *Content Addressable Memory*) :
  - bei einem RAM muss die **Adresse** bekannt sein, um ein bestimmtes Datum aus dem Speicher zu holen
  - ein CAM kann alle gespeicherten Daten ermitteln, von denen ein Teil des **Inhalts** bekannt ist
  - Konzept bereits 1943 von **Zuse** vorgestellt, jedoch erst mit Aufkommen der Halbleitertechnik in Hardware realisiert
  - ein CAM ermöglicht die schnelle Realisierung von **Suchverfahren** (z.B. in Datenbanken oder auf dem Gebiet der Künstlichen Intelligenz)
- heute weitgehend ersetzt durch schnelle Algorithmen (z. B. Haching-Techniken), die auf konventionellem Speicher arbeiten

6 Exkurs: Assoziativspeicher (2)

- allgemeiner Aufbau eines CAM:
  - jede CAM-Speicherzeile besteht aus
    1) *k*-Bit **Schlüssel** *x* (***key***)
    2) *m*-Bit **Datenfeld** *y*
  - *k*-Bit Suchschlüssel (***search key***)
  - Suchmaske (***key mask***) bestimmt relevanten Teil
  - Suche erfolgt **teil-** oder **vollparallel** im nicht ausmaskierten Teil des Schlüsselfelds mittels zusätzlicher Logik
  - **Trefferregister T** zeigt an, in welchen Zeilen Übereinstimmung vorlag
  - sequentielles Auslesen/Verarbeiten der Treffer durch Steuerlogik



7 Caches

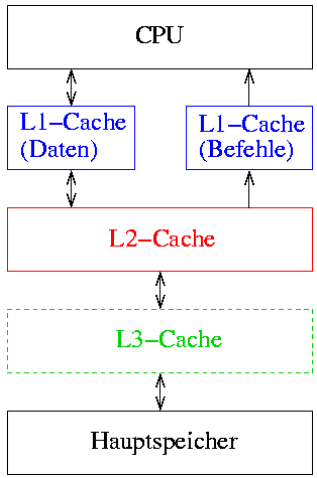
- aufgrund immer höherer Taktfrequenzen, der superskalaren Prozessor-Architektur und „*Out-of-Order*“ Befehlsausführung nehmen die Anforderungen moderner CPUs an den Speicher ständig zu; benötigt werden:
  - **kurze Zugriffszeit**
  - **hohe Transferrate**
- Speichermodule aus aktuellen DRAM-Varianten erreichen zwar eine relativ hohe Transferrate; die Zugriffszeit ist bei wahlfreier Adressierung ist jedoch völlig unzureichend!

⇒ durch Einsatz einer **Speicherhierarchie** (aus gestaffelt schnellen Speichern) soll ein Speicher aus DRAM-Bausteinen ähnlich schnell werden wie ein Speicher aus SRAM-Bausteinen

7.1 Speicherhierarchie

heutige Rechner verfügen über eine mehrstufige Speicherhierarchie:

- prozessorinterne L1 Caches für Code und Daten; Zugriff in 1-3 Takten; typische Größe von 8–64 KByte, SRAM
- prozessorexterner, aber jedoch auf dem gleichen Chip integrierter (für Code und Daten gemeinsamer) L2 Cache; Zugriff in 4-8 Takten; typische Größe von 256 KByte bis 16 MByte, SRAM
- in einigen Systemen zusätzlicher externer L3 Cache; typische Größe 4-64 MByte
- Hauptspeicher (DRAM) von 64 MByte bis zu einigen GByte; Zugriff in ca. 40 ns

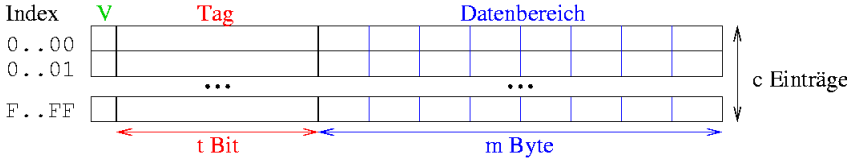


7.1 Speicherhierarchie (2)

- Cache auf Ebene $i+1$ ist größer und langsamer als auf Ebene i
- jeder Cache arbeitet wie ein **Assoziativspeicher** für Speichereinträge des Hauptspeichers:
  - der **Schlüssel** entspricht der Speicheradresse
  - das **Datenfeld** enthält den Speicherinhalt
- bei jedem Speicherzugriff wird beginnend beim Cache auf der Ebene $i = 1$ überprüft, ob die Speicheradresse als Schlüssel gespeichert ist:
  - falls vorhanden (***Cache Hit***) , enthält der Cache der Ebene i eine Kopie des Speicherinhaltes
  - falls nicht vorhanden (***Cache Miss***), wird der Cache auf Ebene $i+1$ bzw. der Hauptspeicher konsultiert und die adressierten Daten werden für spätere Zugriffe im Cache der Ebene i gespeichert

7.2 Cache-Aufbau

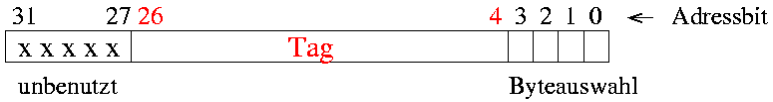
- ein Cache besteht aus $c = 2^k$ Cache-Zeilen
- jede Cache-Zeile (*Cache Line* oder *Cache Block*) besteht aus einem *t*-Bit Identifikator (*Tag*), einem Datenbereich, einem Index und Gültigkeits-Flag V (*Valid Bit*):



- Datenbereich besteht aus $m = 2^d$ Bytes (typisch: $m = 8, 16, 32$); m wird Eintragsgröße (*Block Size* oder *Line Size*) genannt
- der *Tag* enthält einen Teil der Speicheradresse
- Größe des Cache-Speichers: $S_c = c \cdot m$

7.3 Vollassoziativer Cache

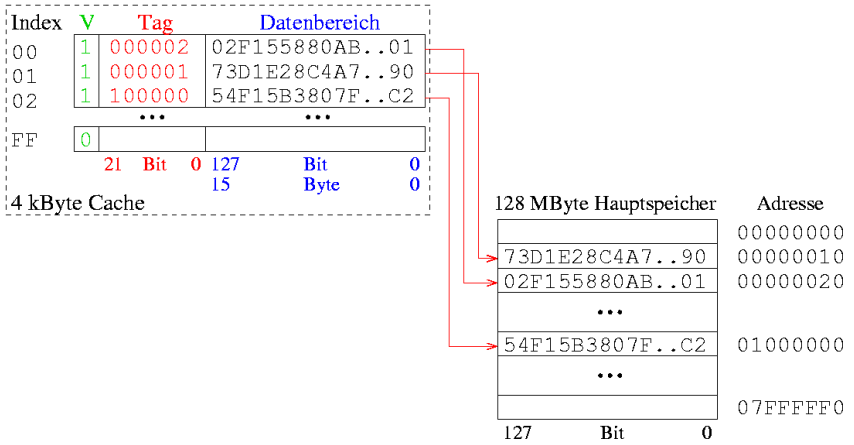
- ein Hauptspeicherinhalt mit $m = 2^d$ Bytes ab Adresse a wird in **beliebiger** Cache-Zeile abgelegt
  - es muss gelten: $a \bmod m = 0$
  - bei $a \bmod m \neq 0$ würde Speicherung in zwei Cache-Zeilen erfolgen
- für einen Hauptspeicher der Größe $S_M = 2^w$ Byte ist ein *Tag* von $t = w - d$ Bit erforderlich
  - Beispiel: 32-Bit Adressen, 128 MByte Hauptspeicher ($\Rightarrow$ w = 27), Cache der Größe 4 KByte mit Zeilen aus je $m = 16$ Bytes ($\Rightarrow$ $d = 4$)



- niedrige d Bit einer Adresse nicht im *Tag* enthalten; sie dienen der **Auswahl eines Bytes** aus Cache-Zeile

7.3 Vollassoziativer Cache (2)

- Aufbau und Arbeitsweise eines vollassoziativen 4 KByte Caches mit 256 Zeilen à 16 Byte bei einem 128 MByte Hauptspeicher:

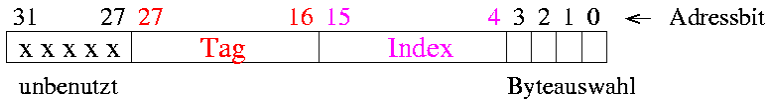


7.3 Vollassoziativer Cache (3)

- bei Zugriff auf Adresse *a* wird das *Tag* aller gültigen Cache-Zeilen i.a. gleichzeitig (**vollparallel**) mit den entsprechenden Adress-Bits von *a* verglichen werden
- Aufwand: $(w-d) \cdot c$ Bit-Vergleicher
  ein c-Bit Treffer-Register
  sowie Baum aus UND- und ODER-Gattern zur Bestimmung eines *Cache Hit* Signals für CPU aus Treffer-Registern
- alternativ kann der Vergleich auch mit geringerem Aufwand **teilparallel** erfolgen (d.h. sequentiell über alle gültigen Zeilen *oder* über alle *Tag*-Spalten); ist jedoch bedeutend langsamer!
- vollassoziativer Cache wird **nur für kleine Caches** verwendet!
  (ansonsten ist Aufwand zu hoch und die Trefferbestimmung aufgrund der aufwendigen Logik zu langsam)

7.4 Direkt abbildender Cache

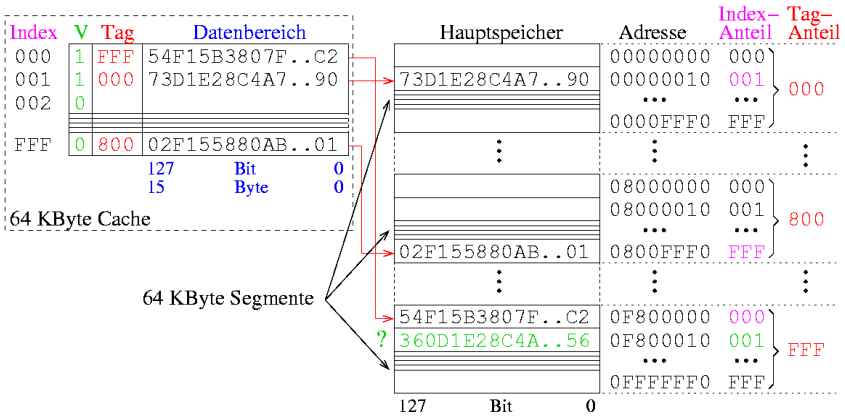
- Hauptspeicher wird in **gleich große Segmente** unterteilt:
  - Größe des Hauptspeichers $S_M = 2^w$ Byte
  - Aufteilung in $s = 2^t$ der Cachegröße S_C entsprechende Segmente
  - es muss gelten: $S_M = s \cdot S_C = s \cdot c \cdot m$
- bei einem Hauptspeicher der Größe $S_M = 2^w$ Byte und einem Cache mit $c = 2^k$ Zeilen aus jeweils $m = 2^d$ Byte ist ein *Tag* von $t = w - k - d$ Bit erforderlich
  - Beispiel: 32-Bit Adressen, 256 MByte Hauptspeicher ($\Rightarrow$ w = 28), 64 KByte Cache mit 4096 Zeilen ($\Rightarrow k = 12$) aus je 16 Bytes ($\Rightarrow d = 4$)



- Index-Anteil gibt **direkt** die Cache-Zeile an!

7.4 Direkt abbildender Cache (2)

- Aufbau und Arbeitsweise eines direkt abbildenden 64 KByte Caches mit 4096 Zeilen à 16 Byte bei einem 256 MByte großen Hauptspeicher:



7.4 Direkt abbildender Cache (3)

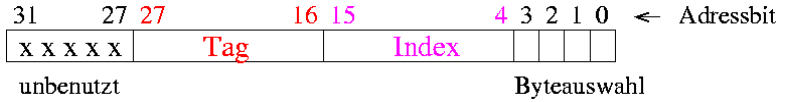
- jede Hauptspeicherzeile kann nur **direkt** auf **eine bestimmte** Cache-Zeile (*direct mapped cache*) abgebildet werden
- *s* Hauptspeicherzeilen mit gleichem Indexanteil (d.h. mit Adress-Distanz 2^{w-t}) **konkurrieren** um eine einzige Cache-Zeile, in der sie gespeichert werden können!
  - bei einem Konflikt muss entsprechende Zeile erst freigegeben werden, bevor sie erneut belegt wird
- bei Eintrag der *i*-ten Hauptspeichereile des *j*-ten Segmentes
  - entspricht *i* dem **Index**
  - entspricht *j* dem ***Tag***
- bei Cache-Zugriff muss nur ein *t*-Bit *Tag* mit den zugehörigen *t* Bit der Adresse verglichen werden
  (Aufwand: *t* Bit-Vergleicher, kein Assoziativspeicher!)

7.5 *n*-Wege teilassoziativer Cache

- **Mischform** aus einem **vollassoziativen** und einem **direkt abbildenden** Cache:
  - Hauptspeicher ist wie beim direkt abbildendem Cache in $s = 2^t$ Segmente der Größe S_M / s unterteilt
  - auch der Cache wird in n Partitionen (*Sets*) der Größe S_C / n unterteilt (*n-Way Set-Associative Cache*)
  - jede Partition ist ein direkt abbildender Cache; assoziativer Zugriff auf entsprechende Zeilen aller n Partitionen
- jede Hauptspeicherzeile kann in einer **beliebigen** Partition, dort aber nur in einer **bestimmten** Cache-Zeile abgespeichert werden
  - bei einer Kollision kann eine andere Partition ausgewählt werden, sofern dort in der entsprechenden Zeile ein Eintrag kollisionsfrei möglich ist

7.5 *n*-Wege teilassoziativer Cache (2)

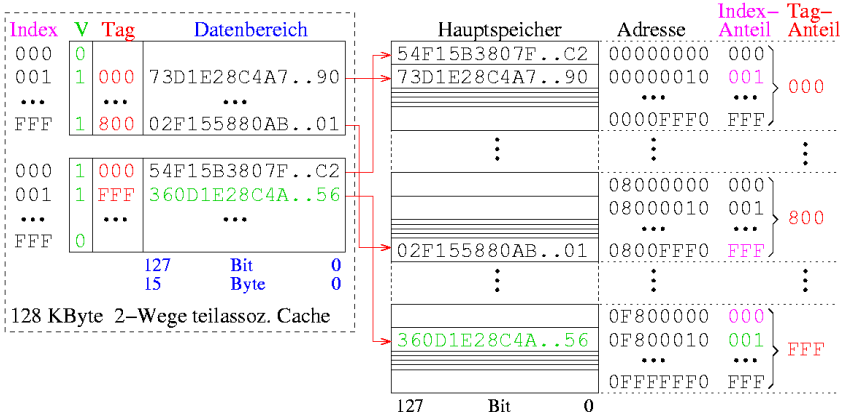
- bei einem Hauptspeicher der Größe $S_M = 2^w$ Byte und einem n-Wege teilassoziativem Cache mit $c = 2^k$ Zeilen je Partition und Cache-Zeilen aus $m = 2^d$ Byte ist ein *Tag* von $t = w - k - d$ Bit erforderlich
  - resultierende Cache-Gesamtgröße: $S_C = n \cdot c \cdot m$ Byte $= n \cdot 2^k \cdot 2^m$ Byte
  - Beispiel: 32-Bit Adressen, 256 MByte Hauptspeicher ($\Rightarrow$ w = 28), 2-Wege teilassoziativer Cache der Größe 128 KByte, d.h. 2 Partitionen à 64 KByte mit jeweils $c = 4096$ Zeilen ($\Rightarrow k = 12$) und Cache-Zeilen aus $m = 16$ Bytes ($\Rightarrow d = 4$)



- Adressbildung erfolgt wie bei einem direkt abbildendem Cache der Größe S_C / n

7.5 *n*-Wege teilassoziativer Cache (3)

- Aufbau und Arbeitsweise eines 2-Wege teilassoziativen Caches der Größe 128 KByte mit Zeilen à 16 Byte bei einem 256 MByte Hauptspeicher:



7.5 *n*-Wege teilassoziativer Cache (4)

- sämtliche n Zeilen mit gleichem Index stellen jeweils einen **Assoziativspeicher** dar
- s Hauptspeicherzeilen mit gleichem Indexanteil (d.h. mit Adress-Distanz 2^{w-t}) **konkurrieren** um n Cache-Zeilen!
  (Allgemein gilt $n \ll s$, d.h. Konflikte sind möglich, aber seltener als beim direkt abbildenden Cache)
- beim Eintrag der i-ten Hauptspeichereile des j-ten Segmentes
  - entspricht i dem **Index**
  - wird eine Partition ermittelt, in der eine Cache-Zeile mit **Index** i frei ist
  - wird dort j als ***Tag*** eingetragen
- bei einem Cache-Zugriff müssen die t-Bit *Tags* aller n Zeilen mit den zugehörigen t Bit der Adresse verglichen werden
  (n parallele Vergleiche, Aufwand: $n \cdot t$ Bit-Vergleicher)

7.6 Lese- und Schreibzugriffe

- Folgende Situationen können bei einem **Lesezugriff** auf eine Cache-Zeile auftreten:
  - ***Read Hit***: bei einem erfolgreichen Lesezugriff (d.h. *Tag* und die zugehörigen Adress-Bits stimmen überein)
    - wird ein Datum aus typisch 1, 2, 4, 8 oder *m* Byte aus Cache-Zeile in den Befehlspuffer bzw. in ein Register der CPU geladen
  - ***Read Miss***: bei einem erfolglosen Lesezugriff auf Adresse *a*
    - wird stets eine **komplette Cache-Zeile** mit *m* Bytes (Burst-Modus !) ab der Adresse $a - a \bmod m$ aus dem Hauptspeicher bzw. aus dem Cache der nächsten Hierarchie-Stufe eingelesen
    - muss ggf. eine Cache-Zeile ersetzt werden, z.B. gemäß der **LRU-Strategie** (*Least Recently Used*)

7.6 Lese- und Schreibzugriffe (2)

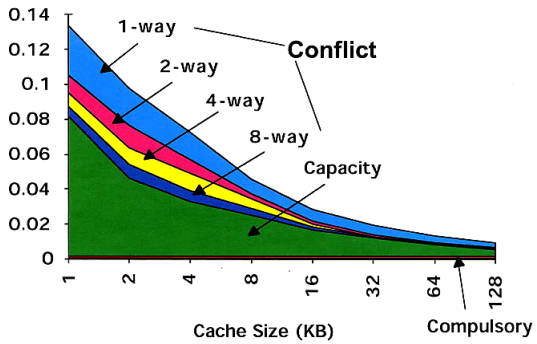
- Folgende Situationen können bei einem **Schreibzugriff** auf eine Cache-Zeile auftreten:
  - ***Write Hit***: bei erfolgreichem Schreibzugriff auf Adresse *a* wird ein Datum aus typisch 1, 2, 4 oder 8 Byte im Cache aktualisiert und die **komplette Cache-Zeile** wird entweder
    - unmittelbar in den Hauptspeicher zurück geschrieben (***Write Through***)
    - oder durch ein zusätzliches Flag (*Dirty Bit*) markiert und erst später bei Verdrängung in den Hauptspeicher zurück geschrieben (***Write Back***)
  - ***Write Miss***: bei erfolglosem Schreibzugriff wird entweder
    - der Eintrag zunächst aus dem Speicher geholt und dann wie bei einem *Write Hit* aktualisiert (***Fetch on Write***)
    - oder der Eintrag wird nur im Hauptspeicher ohne Modifikation des Caches aktualisiert (***Write Around***)

7.7 Verhalten von Caches

- Faustregeln:
  - ein **2-Wege teilassoziativer** Cache hat typischerweise eine *Miss Rate* wie ein **doppelt** großer **direkt abbildender** Cache!
  - ein **8-Wege teilassoziativer** Cache weist für die meisten Anwendungen ungefähr eine *Miss Rate* wie ein **vollassoziativer** Cache auf!
- mittlere Speicher-Zugriffszeit:
  - mit L1-Cache: $t_{access} = t_{accessL1} + \mu_{L1} \cdot t_{accessMemory}$
  - mit L1- und L2-Cache: $t_{access} = t_{accessL1} + \mu_{L1} \cdot (t_{accessL2} + \mu_{L2} \cdot t_{accessMemory})$
    (wobei μ_{L1} und μ_{L2} die jeweiligen Cache-Fehlzugriffsraten darstellen)
- 3 Ursachen für Cache-Fehlzugriffen (3 „*C*“s):
  - bei **Erstbelegung** nach Programmstart (*Compulsory*)
  - wenn wegen zu geringer **Kapazität** Verdrängungen benötigter Zeilen auftreten (*Capacity*)
  - wenn benötigte Zeilen wegen **Konflikten** verdrängt werden (*Conflict*)

7.7 Verhalten von Caches (2)

- typische Fehlzugriffsrate in Abhängigkeit von Ursache, Cache-Größe und -Typ
  (ermittelt für Spec92 Benchmark, Patterson 1998)



- Fehlzugriffsrate sinkt deutlich bei Vergrößerung des Caches!
- Fehlzugriffe durch Erstbelegung sind für viele Anwendungen vernachlässigbar!

7.7 Verhalten von Caches (3)

- **Fazit**: eine gut dimensionierte Cache-Hierarchie kann durch Ausnutzung
  - von zeitlicher und räumlicher Lokalität in Programmen
  - des Burst-Modus bei ausreichend hoher Eintragsgröße

  die Nachteile von langsamen DRAM-Bausteinen weitgehend verdecken!

7.8 Cache-Optimierungen im Programm

- Einfügen von **Prefetch-Instruktionen**, d.h. benötigte Daten werden schon vorab in Cache geholt
  (automatisch vom Compiler oder manuell im Assemblerprogramm)
- **Erhöhung der Lokalität** beim Zugriff auf Daten:

Beispiel 1: *Merging*

```
/* vorher: */
String[] Name;
int[] PersonalNummer;

/* nachher: */
class Person {
  String Name;
  int PersonalNummer;
}
Person[] Personal;
```

Beispiel 2: *Loop Interchange*

```
/* vorher: */
for (j=0;j<100;j=j+1)
  for (i=0;i<5000;i=i+1)
    x[i][j] = 2*x[i][j]

/* nachher: */
for (i=0;i<5000;i=i+1)
  for (j=0;j<100;j=j+1)
    x[i][j] = 2*x[i][j]
```

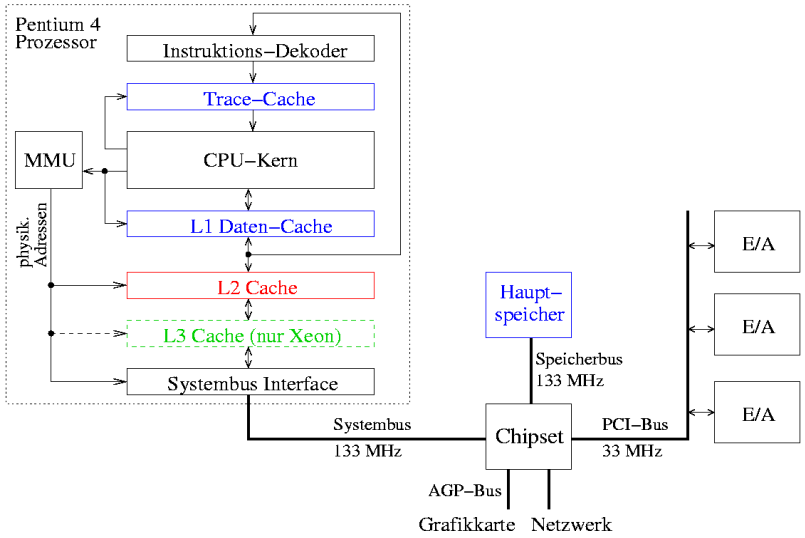

7.8 Cache-Optimierungen im Programm (2)

- **Vermeidung von Cache-Konflikten**
  - die Wahrscheinlichkeit für Verdrängungen durch Cache-Konflikte kann z.B. steigen, wenn
    - die Größe einer Dimension eines mehrdimensionalen Feldes einer **Zweierpotenz** entspricht
    - die Elemente mehrerer Felder, deren Größe jeweils einer **Zweierpotenz** entspricht, verknüpft werden
  - Lösung: Einfügen von **Füllworten** (*Padding* bzw. *Array Padding*), z.B.
    - durch Erhöhen der Feldgröße auf einen Wert, der keine Zweierpotenz darstellt
    - durch Einfügen zusätzlicher Variablen zwischen der Deklaration von Feldern
  - einige Compiler für Höchstleistungscomputer können ein *Padding* auch automatisch durchführen

7.8 Implementierung von Caches

- Cache-Zugriff erfolgt i.a. mit internem **Pipelining**
  - je Takt kann ein neuer Zugriff gestartet werden
  - Dauer eines Zugriffs beträgt jedoch mehrere Takte
- Caches können **logisch/virtuell** oder **physikalisch** adressiert werden
  - L1-Caches werden zumeist logisch/virtuell adressiert
    - Vorteil: sehr schnell, da keine Abbildung der Adressen bei Zugriff erforderlich
    - Nachteil: bei Prozesswechsel muss L1-Cache gelöscht werden
  - L2- oder L3-Cache werden zumeist physikalisch adressiert
    - Vorteil: keine Veränderung bei Prozesswechsel
    - Nachteil: Adressen müssen vor Zugriff erst abgebildet werden

7.9 Fallstudie: Speicherarchitektur eines PC



7.9 Fallstudie: Speicherarchitektur eines PC (2)

- ein **Trace-Cache**
  - enthält nur die tatsächlich **ausgeführten** und bereits **dekodierten** Befehle (µ-Befehle) in der **zeitlichen Reihenfolge** des Auftretens im Programm
  - ersetzt hier einen internen L1 Code-Cache
- Cache-Hierarchie des **Pentium 4** Prozessors:

	Trace-Cache	L1 Daten	L2
Cache-Typ	?	4-Wege teilassoz.	8-Wege teilassoz.
Cache-Größe	12 K µ-Befehle	8 KByte	256 KByte
Eintragsgröße	6 µ-Befehle	64 Bytes	128 Bytes
Schreibstrategie	—	*Write Through*	*Write Back*
Zugriffszeit / Durchsatz	2 Takte / 3 µ-Befehle je Takt	2 Takte / 1 je Takt	7 Takte / 1 je zwei Takte

8 Lernziele

- **Begriffe**: SRAM, DRAM, SDRAM, DDR-SDRAM, RDRAM, EPROM, EEPROM, Flash-Speicher, ...
- Verständnis der Arbeitsweise einer **SRAM-Zelle,** einer **DRAM-Zelle** sowie einer **EPROM-Zelle**
- Möglichkeiten der Beschleunigung bei DRAMs
- **Organisation** von Speicherbausteinen; Aufbau eines Speichers aus vorgegebenen Bausteinen
- Unterschiede in **Aufbau**, **Adressierung** und **Arbeitsweise** eines vollassoziativen, eines direkt abbildenden sowie eines *n*-Wege teilassoziativen Cache
- **Verhalten** von Caches und Auswirkungen von Caches auf die Programmlaufzeit; Cache-Optimierungen im Programm