Inhaltsverzeichnis

3. Speicherkonzepte

3.1 Einleitung

Moderne Computersysteme benötigen wegen ihrer hohen internen Parallelität und ihrer hohen Taktraten komplexe und wohl-organisierte Speicherstrukturen mit *kurzen* Zugriffszeiten, um die immer schneller werdenden Prozessoren effizient mit Befehlen und Daten versorgen zu können. Da der Preis für Halbleiterspeicher überproportional zur Zugriffsgeschwindigkeit steigt, muss eine wirtschaftliche Lösung des Problems durch spezielle Speicherarchitekturen erreicht werden.

Die schnelle Weiterentwicklung der Mikroprozessoren hat dazu geführt, dass mittlerweile selbst Personal Computer (PCs) eine mehrstufige Speicherhierarchie benötigen, um bei wirtschaftlich vertretbaren Kosten einen ausreichenden Datendurchsatz zu erzielen. Bei steigender Verarbeitungsgeschwindigkeit der CPUs wird der Datentransport zunehmend zum Flaschenhals im System. Der großzügige Einsatz extrem schneller Speicher verbietet sich aus Kostengründen, sodass eine praktikable Lösung des Problems in der Verwendung verschiedener Speichermedien, kombiniert mit einer effizienten Datenorganisation liegt.

Neben der *schnellen* Versorgung der Prozessoren mit Befehlen und Daten ensteht auch das Problem der Speichermenge. Offensichtlich kann heute der Speicherbedarf eines einzelnen Programms mit seinen während der Ausführung erzeugten Zwischendaten häufig sehr groß werden, sodass dieses die Kapazität des Arbeitsspeichers sprengt. Ideal wäre natürlich ein einstufiges Speicherkonzept, bei dem mit jedem Prozessortakt auf jedes Speicherwort zugegriffen werden kann. Das ist technologisch für Prozessoren hoher Leistung jedoch nicht möglich, Große Speicher existieren nur mit relativ langsamem Zugriff, während Speicherbausteine mit hoher Zugriffsgeschwindigkeit in ihrer Speicherkapazität beschränkt und sehr teuer sind.

3.1.1 Einführung einer Speicherhierarchie

Aus technologischer Sicht wird die Lücke zwischen der Verarbeitungsgeschwindigkeit des Prozessors und der Zugriffsgeschwindigkeit der Speicherbausteine des Hauptspeichers immer größer. Die hohen Prozessortaktraten und die Fähigkeit superskalarer Mikroprozessoren, mehrere Operationen pro Takt auszuführen, erzeugen von Seiten des Prozessors einen immer größeren Bedarf nach Programmcode und Daten aus dem Speicher. Die Geschwindigkeit dynamischer Speicherbausteine hat hingegen über die Jahre hinweg deutlich weniger zugenommen. Eine Ausführung der Programme aus dem Hauptspeicher hätte zur Folge, dass der Prozessor nur mit einem Bruchteil seiner maximalen Leistung arbeiten könnte. Deshalb muss der Prozessor seine Befehle vorwiegend aus dem auf dem Prozessor befindlichen Code-Cache-Speicher und seine Daten aus den Registern und den weiteren internen und externen Cache-Speichern erhalten.

Unter Caches sind in diesem Zusammenhang vom Zugriff her sehr schnelle, aber im Vergleich zum Hauptspeicher auch sehr kleine Zwischenspeicher zu verstehen, die ausgewählte Teile von Inhalten des Hauptspeichers als Kopie enthalten. Auf der anderen Seite werden große Datenmengen, die weniger häufig benötigt werden, auf vom Zugriff her langsameren Hintergrund- oder Massenspeichern abgelegt. Diese hierarchische Folge aus Registern, Caches, Hauptspeicher und Plattenspeicher nennt man *Speicherhierarchie*.

Das Hauptproblem einer einerseits zugriffsschnellen und andererseits effizienten Speicherhierarchie ist die Datenorganisation, d.h. wann werden welche Daten oder Programmcode auf welchem Medium gespeichert. Grundlage dieser Problemlösung bei einer Speicherhierarchie bildet das sog. **Lokalitäts-**

prinzip, dessen Gesetzmäßigkeit sowohl die Befehle als auch die Daten eines Programms weitgehend gehorchen. Man unterscheidet die *zeitliche Lokalität* und die *räumliche Lokalität*. Zeitliche Lokalität bedeutet, dass im Laufe einer Programmausführung auf dasselbe Code- oder Datenwort mehrfach zugegriffen wird. Dies ist zum Beispiel der Fall wenn bei der Ausführung einer Schleife ein bestimmtes Datenwort immer wieder gelesen oder die entsprechende Speicherstelle immer wieder neu beschrieben bzw. aktualisiert wird. Räumliche Lokalität bedeutet, dass im Verlaufe einer Programmausführung auch die zu einer häufig benutzten Speicherstelle benachbarten Code- oder Datenwörter häufiger benötigt werden als andere. Ein Beispiel dafür ergibt sich bei der Ausführung eines im Hauptspeicher abgelegten Programmes. Da im Programm ein Befehl nach dem anderen ausgeführt wird (wenn nicht gerade ein Sprungbefehl vorliegt) und der Programmcode sequentiell im Hauptspeicher abgelegt ist, werden im Hauptspeicher fortlaufend Speicherzellen mit aufsteigenden Adressen gelesen.

Diese zeitliche und räumliche Lokalität kann sinnvoll genutzt werden, indem man Befehle und Daten, auf die wahrscheinlich als nächstes zugegriffen werden muss, möglichst nahe am Prozessor platziert. Andererseits werden Befehle und Daten, die wahrscheinlich in nächster Zeit nicht benötigt werden, auf entfernteren Speichermedien abgelegt. Nahe beim Prozessor verstehen wir dabei im Sinne von zugriffsschnell und bei Medien mit schnellen Zugriff handelt es sich auch um teuren und damit stark begrenzt zur Verfügung stehenden Speicher. Zu diesem Zweck werden Daten, auf die in wenigen Takten wieder zugegriffen wird, in CPU-internen Registern bereitgehalten und ständig aktualisiert. Register nützen dabei die zeitliche Lokalität. In der nächsten Ebene verdrängt ein CPU-interner Daten-Cache-Speicher einmal geholte Daten erst wieder, wenn sie durch neuere Zugriffe ersetzt werden müssen. Dieser nutzt sowohl zeitliche als auch räumliche Lokalität, da nach einem Cache-Fehlzugriff nicht nur das betreffende Datenwort, sondern ein ganzer aus mehreren Datenworten bestehender Cache-Block im Cache-Speicher bereitgestellt wird.

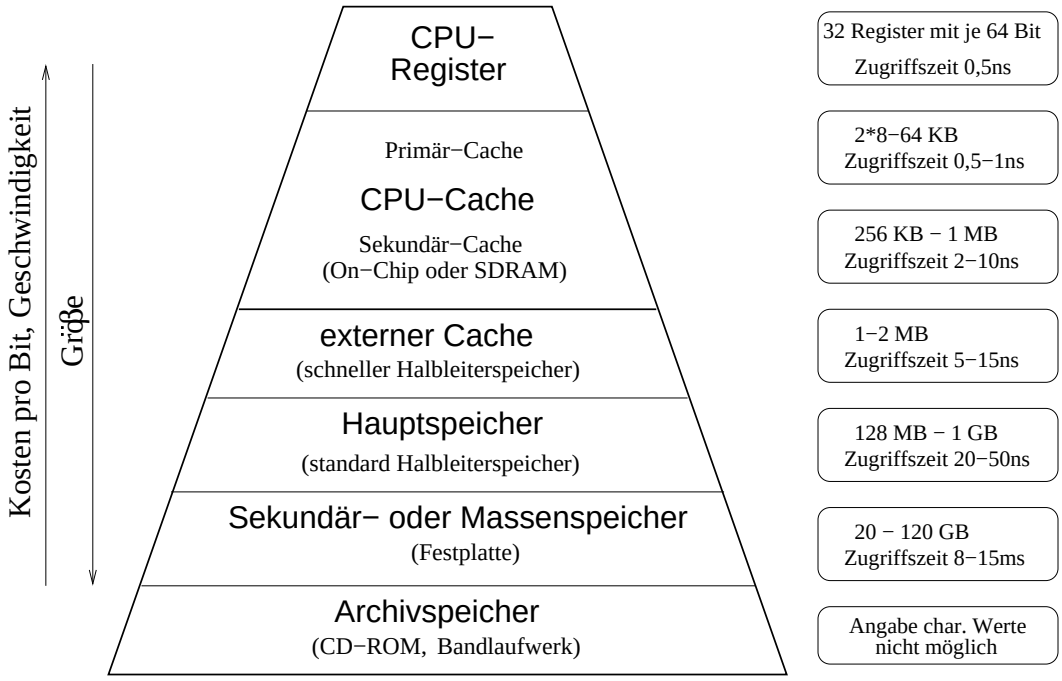


Abbildung 3.1: Speicherhierarchie (Stand ca. 2005)

Heutige Rechner besitzen eine **Speicherhierarchie**, die sich – nach absteigenden Zugriffsgeschwindigkeiten und aufsteigenden Speicherkapazitäten geordnet – aus Registern, Primär-Cache-Speicher, Sekundär-Cache-Speicher, Hauptspeicher und Sekundärspeicher zusammensetzt. Abb. 3.1 zeigt eine solche Speicherhierarchie mit einigen beispielhaften Zugriffszeiten und Speicherkapazitäten. Speichermedien in höheren Ebenen der Hierarchie sind kleiner, schneller, aber auch pro Byte teurer als solche in den tieferen Ebenen. Speicherinhalte werden bei Bedarf von einer tieferen zu einer höheren und damit prozessornäheren Speicherebene oder umgekehrt übertragen. Eine Anforderung kann bei-

spielsweise durch einen Ladebefehl entstehen, der einen sog. Cache-Fehlzugriff (Miss) zur Folge hat, d.h. der gesuchte Speicherinhalt befindet sich nicht im Cache. Eine andere Ursache für eine Übertragung kann ein sog. Seitenfehler, bei dem die benötigte Seite sich gerade nicht im Hauptspeicher (HSP) befindet und von einer Festplatte geladen wird, sein. In der Regel sind die Speicherinhalte in den höheren Ebenen Kopien der Speicherinhalte in den unteren Ebenen, sodass sich insgesamt eine hohe Redundanz ergibt. Eine grundlegende Speicherorganisationsfrage ist, ob und wann bei einem Schreibzugriff auf ein Speicherelement höherer Ebene auch eine Modifikation des entsprechenden Speicherelementes in einem der Speicher tieferer Ebene erfolgen muss. Ein sofortiges „Durchschreiben“, das die Konsistenz in allen Kopien sicherstellt, ist nicht effizient implementierbar.

3.1.2 Virtueller Speicher und Caching

Neben dem schnellen Zugriff ergibt sich ein anderes Problem bei der effizienten Nutzung des Speichers aus den verschiedenen Adressräumen einzelner Speichermedien wie z.B. dem Hauptspeicher und der Festplatte. Der gesamte in einem Anwenderprogramm benutzte Teil des Speichers aus der Hierachie soll natürlich vom Anwenderprogramm aus einheitlich und transparent über einen gemeinsamen Adressraum ansprechbar sein. Dieses Problem wird durch die sog. **virtuelle Speicherverwaltung** gelöst. Diese verwaltet virtuelle und physikalische Adressen, rechnet diese ineinander um und vermittelt dabei dem Prozessor bzw. dem Anwender den Eindruck, als würde er ausschließlich auf dem Hauptspeicher arbeiten. Virtuelle Adressen sind die Adressen, über die das Anwenderprogramm die benötigten Speicherzellen anspricht. Physikalische Adressen sind die tatsächlichen Adressen innerhalb der einzelnen Speichermedien.

Ein weiterer gewollter Effekt der virtuellen Speicherverwaltung ist, dass der mit Hilfe der virtuellen Adressierung nutzbare Arbeitsspeicher eines Rechners erheblich größer erscheint als der tatsächliche vorhandene Hauptspeicher (HSP). Die virtuelle Speicherverwaltung stellt dem Anwender mit Hilfe der virtuellen Adressen einen großen Speicher zur Verfügung, ohne dass der HSP entsprechend ausgebaut ist. Der virtuelle Adressraum kann dabei um ein Vielfaches größer sein als der tatsächlich vorhandene aus Halbleiterbausteinen bestehende Hauptspeicher. Zu diesem Zweck wird ein Großteil der Speicherinhalte auf dem Hintergrundspeicher abgelegt und über sog. virtuelle Adressen angesprochen. Der fehlende Speicherplatz wird also von der Speicherverwaltung mit Hilfe von Massenspeichern aufgefüllt, sodass sich immer nur ein Teil des virtuellen Speicherraums für direkten Zugriff im Halbleiterspeicher befindet. Die ausgelagerten Teile müssen bei Bedarf vom Massenspeicher geladen werden.

Beide Konzepte, der virtuelle Speicher als auch das Caching, versuchen mit Hilfe eines kleineren, aber dafür schnelleren Speichermediums einen großen Adressraum optimal zu bedienen. So gibt es auch Gemeinsamkeiten in der Verwaltung des virtuellen Speichers und des Caches. Cache-Blöcke korrespondieren mit den Speicherblöcken, d.h. den Seiten (*pages*) oder Segmenten des virtuellen Speichers. Ein Cache-Fehlzugriff entspricht einem Seiten- oder Segmentfehlzugriff.

Ein großer Unterschied besteht jedoch in der Größe der Cache- und der Speicherblöcke als auch im Zeitverlust, die ein Fehlgriff verursacht. Der Unterschied in der Geschwindigkeit zwischen Zugriffen auf Cache- und Hauptspeicher liegt im Verhältnis 1:20, meist sogar günstiger. Das Verhältnis der Zugriffszeiten zwischen Hauptspeicher und Massenspeicher liegt jedoch oft bei 1:100 000. Die obige Tabelle gibt einen Überblick über die unterschiedlichen Größenordnungen verschiedener Parameter, bei denen sich Caches und virtuelle Speicher unterscheiden.

Parameter	**Primär-Cache**	**virtueller Speicher**
Block-/Seitengröße	16 – 128 Byte	4 – 64 KB
Zugriffszeit	1 – 2 Takte	40 – 100 Takte
Fehlzugriffsaufwand	8 – 100 Takte	70000 – 6000 000 Takte
Fehlzugriffsrate	0,5 – 10%	0,00001 – 0,001%
Speichergröße	8 – 64 KB	16 MB – 8 GB

Beim virtuellen Speicher wird der Adressraum für Anwendungsprogramme scheinbar vergrößert, während beim Cache eine Beschleunigung des Zugriffs ohne Veränderung des Adressraumes erreicht wird. Ein wesentlicher Unterschied zwischen Cache und virtuellem Speicher ist, dass infolge des Cachings bestimmte Speicherinhalte mehrfach abspeichert werden. Für das Caching müssen also zusätzliche Speicherbausteine installiert werden, die virtuelle Speicherverwaltung hingegen basiert auf den vorhandenen Speichereinheiten wie Arbeitsspeicher und Festplatte. Während die Cache-Speicherverwaltung vollständig in Hardware ausgeführt ist, wird die virtuelle Speicherverwaltung innerhalb des Betriebssystems, also in Software, ausgeführt. Die virtuelle Speicherverwaltung erhält von der Hardware Unterstützung durch die sog. **Speicherverwaltungseinheit** (*Memory Management Unit – MMU*) und durch spezielle Maschinenbefehle.

Die Adressbreite des Prozessors bestimmt die maximale Größe des virtuellen Adressraums und damit des virtuellen Speichers. Die Größe des oder der Cache-Speicher ist davon unabhängig. Die Cache-Speicherverwaltung verschiebt Cache-Blöcke zwischen den Cache-Speichern verschiedener Hierarchieebenen und dem Hauptspeicher. Die virtuelle Speicherverwaltung verschiebt Speicherblöcke zwischen Hauptspeicher und Sekundärspeicher, d.h. in der Regel einer Festplatte.

Durch die virtuelle Speicherverwaltung können auf einem System Programme benutzt werden, die aufgrund Ihrer Größe nicht in den vorhandenen Halbleiterspeicher passen. Der Aufbau großer Programme rechtfertigt eine solche Vorgehensweise aus folgenden Gründen:

- Viele in einem Programm eingebaute Funktionen werden vom Anwender nur selten oder nie benutzt. Der Programmcode dieser selten benutzten Funktionen muss nicht ständig im Hauptspeicher verfügbar sein.
- Programme enthalten oft Fehlerbehandlungsroutinen für selten auftretende Fehler. Dieser Kode wird daher nur selten zur Ausführung benötigt.
- In einem Datenbank-System beispielsweise wird in der Regel nur auf einem kleinen Teil der Datenbank gearbeitet, sodass zu einem bestimmten Zeitpunkt nur ein Teil der kompletten Datenmenge im Speicher gehalten werden muss.

Wir wollen in dieser Kurseinheit die beiden Speicherkonzepte anhand von Organisation und Ausführung ausführlich behandeln. Wir beginnen dabei mit der in der logischen Hierarchie höher angesiedelten virtuellen Speicherverwaltung.

3.2 Virtuelle Speicherverwaltung

3.2.1 Ziele der virtuellen Speicherverwaltung

Mehrbenutzer- und Multitasking-Betriebssysteme stellen neben dem schnellen Zugriff auf Code und Daten sowie der effizienten Nutzung des Speichers weitere Anforderungen an die Speicherverwaltung. Programme, die in einer Mehrprozessumgebung lauffähig sein sollen, müssen relozierbar, d.h. innerhalb des Speichers beliebig verschiebbar, sein. Das bedeutet, dass die geladenen Programme und ihre Daten nicht an festgelegte, physikalische Speicheradressen gebunden sein dürfen. Weiterhin soll für jeden Prozess ein eigener großer Adressraum sowie geeignete Schutzmechanismen zwischen den Prozessen (unter Prozess verstehen wir hier allgemeiner eine lauffähige Anwendung) bereitgestellt werden. Zur Durchführung von Schutzmaßnahmen müssen zusätzliche Information über Zugriffsrechte und Gültigkeit der Speicherwörter (zum Beispiel nach dem Überschreiben) abgelegt werden.

Die wichtigsten Ziele bei der Realisierung einer virtuellen Speicherverwaltung (VS) sind demnach:

- Dem Anwenderprozess soll ein großer Adressraum mit einer Abbildung von beispielsweise 2^{32} bis 2^{64} virtuellen Adressen auf beispielsweise 2^{28} physikalische Adressen für physikalisch vorhandene Hauptspeicherzellen zur Verfügung gestellt werden.
- Für den Anwenderprozess soll dieser große virtuelle Adressraum, welcher der Adressbreite des Prozessors entspricht, völlig transparent sein, d.h. der Anwenderprozess arbeitet mit einem großen virtuellen Hauptspeicher (HSP) ohne zu merken dass dieser tatsächlich nur virtuell existiert.
- Durch die VS soll ein schneller Programmstart ermöglicht werden, da nicht alle Code- und Datenbereiche zu Programmbeginn im Hauptspeicher vorhanden sein müssen.
- Es soll eine einfache Relozierbarkeit (*relocation*) ermöglicht werden, die es erlaubt, ein bestimmtes Programm in beliebige Bereiche des Hauptspeichers zu laden oder zu verschieben.
- Die VS soll die Verwendung von Schutzbits unterstützen, die beim Zugriff geprüft werden und es ermöglichen, unerlaubte Zugriffe abzuwehren.
- Die VS soll eine automatische (vom Betriebssystem organisierte) Verwaltung des Haupt- und Sekundärspeichers, die den Programmierer von diesen Aufgaben entlastet, zur Verfügung stellen.

Bei der Umsetzung dieser Ziele in der Organisation einer virtuellen Speicherverwaltung sind dann die folgenden Fragestellungen von Bedeutung:

- *Wo kann ein Speicherblock im Hauptspeicher platziert werden?*
  Wegen der linearen Adressierung des Hauptspeichers durch physikalische Adressen und der festen Seitenlänge ist der Speicherort einer Seite im Hauptspeicher beliebig wählbar. Die freie Auswahl der Platzierung mindert insbesondere die Zahl der Verdrängungen einer Seite aus dem HSP durch eine andere gerade benötigte und reduziert somit den Einfluss des außerordentlich hohen Fehlzugriffsaufwandes. Allerdings sollte bei den Segmenten wegen ihrer potenziell unterschiedlichen Segmentlängen auf eine möglichst große Verdichtung geachtet werden, um nicht durch Fragmentierung (auf dieses Problem kommen wir später noch) des Freispeichers (sog. *externe Fragmentierung*) so kleine freie Hauptspeicherbereiche zu erhalten, dass diese nicht mehr belegt werden können.
- *Welcher Speicherblock sollte bei einem Fehlzugriff ersetzt werden?*
  Die übliche Ersetzungsstrategie ist die LRU-Strategie (*Least Recently Used*), die anhand von Zusatzbits für jeden Speicherblock feststellt, auf welchen Speicherblock am längsten nicht mehr zugegriffen worden ist, und diesen ersetzt.
- *Wann muss ein verdrängter Speicherblock in den Sekundärspeicher zurückgeschrieben werden?*

Falls der zu verdrängende Speicherblock im Hauptspeicher nicht modifiziert, d.h. neu beschrieben wurde, so kann dieser einfach überschrieben werden. Andernfalls muss der verdrängte Speicherblock in den Sekundärspeicher zurückgeschrieben werden.

- *Wie wird ein Speicherblock aufgefunden, der sich in einer höheren Hierarchiestufe befindet?*
  Eine Seiten- oder Segmenttabelle kann sehr groß werden. Um die Tabellengröße zu reduzieren, werden meistens Hash-Verfahren auf den virtuellen Seiten-/Segmentadressen durchgeführt. Dies führt dazu, dass die Tabellengröße deutlich verringert werden kann. Diese ist meist sehr viel kleiner als die Zahl der virtuellen Seiten. Man spricht dann von einer invertierten Seitentabelle (*inverted page table*).
- *Welche Seitengröße sollte gewählt werden?*
  Es gibt gute Gründe für große Seiten: Die Größe der Seitentabelle ist umgekehrt proportional zur gewählten Seitengröße. Größere Seiten sparen Speicherplatz, da eine kleinere Seitentabelle genügt. Weiterhin kann ein Transport großer Seiten zwischen Haupt- und Sekundärspeicher effizienter organisiert werden als ein häufiger Transport kleiner Seiten. Die Zahl der Einträge im Adressumsetzungspuffer (TLB) der Speicherverwaltungseinheit ist beschränkt. Große Seiten erfassen mehr Speicherplatz und führen damit zu weniger TLB-Fehlzugriffen (siehe Abschnitt 3.2.7). Für kleine Speicherseiten spricht die geringere Fragmentierung des Speichers (sog. *interne Fragmentierung*) und ein schnellerer Prozessstart. Die Lösung ist meist eine Hybridtechnik, die es erlaubt, mehrere unterschiedliche Seitengrößen zuzulassen oder Segmente und Seiten zu kombinieren.

3.2.2 Grundprinzip der virtuellen Speicherverwaltung

Das Grundkonzept der **virtuellen Speicherverwaltung** (virtuell bedeutet „scheinbar“) besteht in der Entkopplung zwischen den Adressen, die das auf dem Prozessor ablaufende Programm benutzt (**logische** bzw. **virtuelle Adressen**), und den tatsächlichen Adressen im Halbleiterspeicher (**physikalische Adressen**). Abbildung 3.2 zeigt vereinfacht das Grundprinzip.

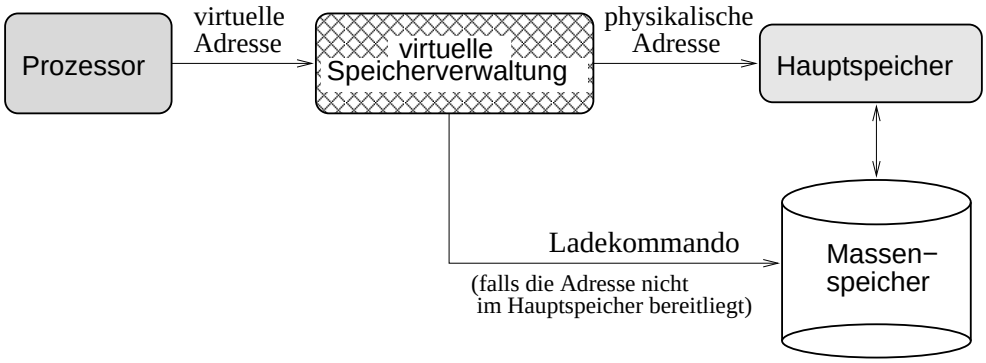


Abbildung 3.2: Prinzip der virtuellen Speicherverwaltung

Die vom Prozessor ausgeführte Anwendung adressiert einfach einen großen *virtuellen* Adressraum und die virtuelle Speicherverwaltung übersetzt die virtuellen Adressen in entsprechende physikalische Adressen. Genauer gesagt wird aus den Speicherreferenzen in den Maschinenbefehlen durch die Adressrechnung zunächst eine effektive Adresse (siehe KE2 Abschnitt über Adressierungsarten) berechnet. Diese effektive Adresse wird als sog. **logische Adresse** durch die virtuelle Speicherverwaltung zuerst in eine **virtuelle Adresse** und dann in eine **physikalische Speicheradresse** transformiert. Häufig wird in der Literatur nicht zwischen logischen und virtuellen Adressen unterschieden, die logische Adresse ist die vom Anwender oder Programmierer sichtbare und benutzte Adresse, die virtuelle Adresse ist die vom Prozessor benutzte Adresse.

Die virtuelle Speicherverwaltung unterteilt den physikalisch zur Verfügung stehenden Speicher in Speicherblöcke, die als Seiten mit fester Größe oder/oder Segmente mit variabler Größe organisiert

sind und bindet diese an die im Rechner ablaufenden Prozesse. Damit ergibt sich die Möglichkeit einen Schutzmechanismus für die einzelnen Blöcke einzurichten, der den Speicherzugriff eines Prozesses auf seine dadurch zugeordneten Speicherblöcke beschränkt.

Die virtuelle Speicherverwaltung stellt während der Programmausführung fest, welche Daten gerade gebraucht werden, transferiert die angeforderten Speicherbereiche zwischen Sekundärspeicher (z.B. Festplatte) und Hauptspeicher und aktualisiert die Referenzen zwischen virtueller und physikalischer Adresse in einer Übersetzungstabelle. Die Speicherverwaltung kann Bereiche fester Länge, sog. Seiten, und/oder variabler Länge, sog. Segmente, verwenden. Falls eine Referenz nicht im physikalischen Speicher, dem Hauptspeicher, gefunden werden kann, wird dies als **Seitenfehler** (*page fault*) bezeichnet. Der entsprechende Speicherblock (Seite) muss dann durch eine Betriebssystemroutine in den Hauptspeicher nachgeladen werden. Das Benutzerprogramm bemerkt diese Vorgänge nicht. Die Zeit, die für den Umladevorgang benötigt wird, verringert jedoch die für den Benutzer erreichbare Verarbeitungsgeschwindigkeit. Die Lokalität von Code und Daten und eine geschickte Ersetzungsstrategie für jene Seiten, die im Hauptspeicher durch Umladen überschrieben werden, gewährleistet dennoch eine hohe Wahrscheinlichkeit, dass die Daten, die durch den Prozessor angefordert werden, im physikalischen Hauptspeicher zu finden sind.

Die Übersetzung einer logischen in eine physikalische Adresse geschieht in der Regel nicht über *eine*, sondern über *mehrere* Übersetzungstabellen. Grundsätzlich werden zwei Übersetzungen eingesetzt, die Segmentierung und die Seitenübersetzung. Dementsprechend werden zwei Übersetzungstabellen – eine Segmenttabelle und eine Seitentabelle – benutzt. Bei der Speichersegmentierung wird der gesamte Speicher in ein oder mehrere Segmente variabler Länge eingeteilt. Für jedes Segment können Zugriffsrechte getrennt vergeben werden, die dann bei der Adressübersetzung nachgeprüft werden. Ebenso wird beim Übersetzen einer logischen Adresse getestet, ob die erzeugte virtuelle Adresse überhaupt innerhalb des vom Betriebssystem reservierten Speicherbereiches liegt. Zusammen mit einer prozessspezifischen Festlegung der Zugriffsrechte kann damit ein Speicherschutzmechanismus realisiert werden.

Zusätzlich zur Speichersegmentierung wird die Seitenübersetzung (Speicherseitenverwaltung oder auch Paging genannt) eingesetzt. Dabei steht die Segmentierung in der logischen Hierarchie über der Seitenübersetzung, da in einer ersten Übersetzungsphase über die Segmentierung aus der *logischen Adresse* eine *virtuelle Adresse* bestimmt wird. In der zweiten Übersetzungsphase wird dann mit Hilfe der Seitenübersetzung aus der *virtuelle Adresse* die physikalische Adresse bestimmt. Bei der Seitenübersetzung wird der Adressraum in Seiten fester Größe zerlegt (Größenordnung 1 bis 64 KByte, häufig 4 KByte). Die virtuelle Adresse wird in eine virtuelle Seitenadresse und eine virtuelle Seitennummer (Offset-Adresse) aufgeteilt. Bei der Übersetzung einer virtuellen in eine physikalische Adresse wird die virtuelle Seitenadresse mit Hilfe von möglicherweise mehrstufigen Übersetzungstabellen in eine physikalische Seitenadresse umgewandelt, während die Offset-Adresse unverändert übernommen wird. Wir werden die beiden Konzepte der Segmentierung und der Seitenübersetzung in den folgenden Abschnitten noch etwas genauer betrachten.

3.2.3 Speicherseitenverwaltung (Paging)

Beim Verfahren der Speicherseitenverwaltung (Seitenübersetzung) wird der gesamte Adressraum, sowohl der virtuelle als auch der physikalische Adressraum, in gleichgroße Abschnitte, die sog. **Seiten** (*Pages*), aufgeteilt. Der Abschnitt des Hauptspeichers (HSP), an dem eine Seite abgelegt werden kann, wird als **Seitenrahmen** (*pageframe*) bezeichnet. Typische Seitengrößen liegen im Bereich von 512 bis 4096 Byte, sind aber immer von der Größe 2^n ($n = 1, 2, \ldots$).

Durch die Seitenübersetzung wird die virtuelle Adresse in zwei Teile zerlegt: In eine Seitennummer und in die Adresse innerhalb einer Seite, die **Verschiebung** (Distanzbetrag) oder der **Offset**.

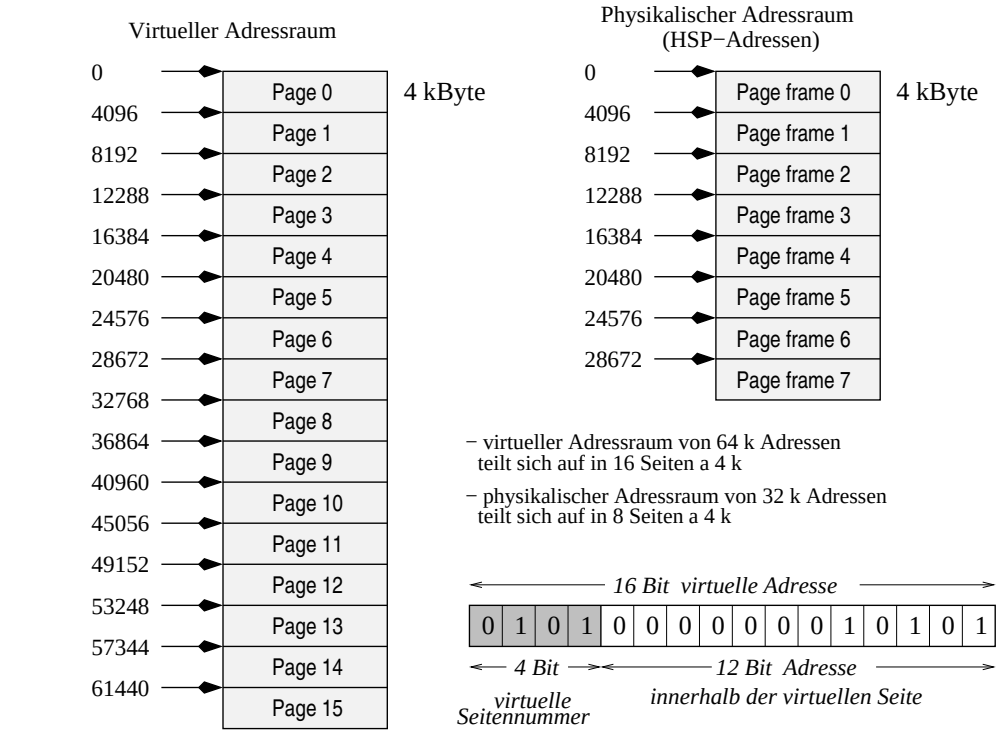
Die Seitennummer dient als Zeiger innerhalb einer **Seitentabelle** (*Pagetable*), in der zu jeder Page- bzw. Seitennummer die dazugehörige physikalische Anfangsadresse dieser Seite gespeichert ist. Da-

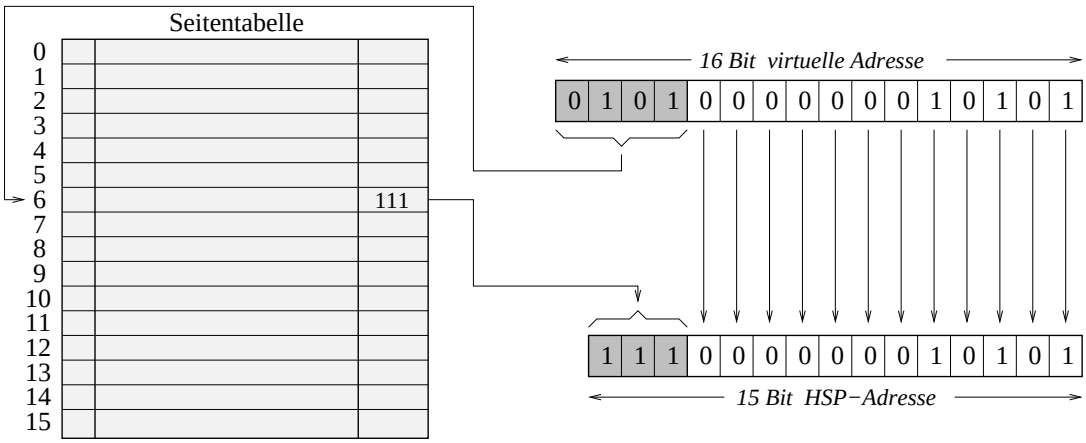
Seitennummer	Offset

Abbildung 3.3: Aufteilung einer virtuellen Adresse

durch ist es möglich, die Seiten in einen beliebigen Seitenrahmen (Pageframe) im Hauptspeicher zu kopieren. Zudem wird bei den meisten Prozessoren die Seitennummer in mehrere Teile zerlegt, sodass eine hierarchische Struktur mit mehreren kleinen Tabellen entsteht. Durch die Aufteilung in viele kleine Einheiten können auch die Seitentabellen beliebig im Speicher verteilt werden. Außerdem kann bei einer hierarchischen Struktur jeder Prozess in einem Mehrprozesssystem ohne zusätzlichen Aufwand eine eigene Seitentabelle zugeteilt bekommen.

Beispiel 3.1 *Speicherseitenverwaltung*
Angenommen wir haben einen Prozessor mit einem 16 Bit breiten Adressbus. Der virtuelle Adressraum besteht damit aus $2^{16} = 65536$ Adressen. Ein Prozess, der auf diesem Prozessor abläuft, kann also Adressen zwischen 0 und 65535 ansprechen. Bei einem Byte-adressierten Speicher und einer Seitengröße von 4096 Byte besteht der virtuelle Speicher aus 16 Seiten. Nehmen wir weiter an, dass der Hauptspeicher 32 KByte groß ist, dann besitzt dieser 8 Seitenrahmen. Die folgende Abb. zeigt eine solche Aufteilung des Speichers in 4 KByte große Seiten.





Bei einem Speicherzugriff wird die 16-Bit-Adresse der angeforderten Speicherzelle in eine virtuelle 4-Bit-Seitennummer (Page Number) und eine 12-Bit-Adresse innerhalb der selektierten Seite aufgeteilt. Ein Zugriff auf die Speicherzelle mit der dezimalen Adresse 20501 beispielsweise resultiert in einem Zugriff auf die Zelle mit der Adresse 21 innerhalb der virtuellen Seite 5.

Seitentabelle

0	1		100
1	1		110
2	0		
3	0		
4	1		001
5	1		101
6	0		
7	1		111
8	0		
9	1		000
10	1		010
11	1		011
12	0		
13	0		
14	0		
15	0		

HSP

Page 9	Frame 0
Page 4	Frame 1
Page 10	Frame 2
Page 11	Frame 3
Page 0	Frame 4
Page 5	Frame 5
Page 1	Frame 6
Page 7	Frame 7

Abbildung Seitentabelle <---> HSP

Nachdem die Seitennummer bestimmt ist, wird festgestellt ob die gesuchte Seite sich im bereits im HSP befindet. Dies passiert anhand der Seitentabelle, die für jede virtuelle Seite einen Eintrag enthält. Die Seitentabelle des Beispiels enthält also 16 Einträge mit je drei Feldern. Das erste Feld enthält eine 1 wenn sich die Seite im HSP befindet und ansonsten eine 0. Das zweite Feld enthält die physikalische Adresse der Seite auf dem Hintergrundspeicher. Diese wird nur benötigt wenn die Seite sich nicht im HSP befindet oder später aufgrund einer Modifikation zurückgeschrieben werden muss. Befindet sich die gesuchte Seite im HSP, dann enthält das dritte Feld die Nummer des Seitenrahmens, indem die Seite im HSP gespeichert ist. Bei 8 Seitenrahmen ist Rahmennummer 3 Bit lang, die mit den restlichen 12 Bit aus der virtuellen Adresse zur physikalischen HSP-Adresse mit 15 Bit ($2^{15} = 32768$ HSP-Adressen) zusammengesetzt wird. □

Die meisten realen Rechensysteme laden eine Seite nur auf Anforderung. Dieses Verfahren wird als **Demand Paging** bezeichnet. Beim Starten eines Programms wird der Programmcode nur im virtuellen Speicher vorgesehen, d.h. die Seitentabelle wird initialisiert, aber die Seiten verbleiben auf dem Massenspeicher. Erst wenn der Prozess beim Versuch, einen Befehl auszuführen, einen Seitenfehler auslöst, wird die entsprechende Seite in den Hauptspeicher geladen. Als Folge dieser Vorgehensweise häufen sich beim Programmstart die Seitenfehler solange, bis die benötigten Informationen geladen wurden. Zu jedem Zeitpunkt t eines Programmlaufs gibt es daher eine Menge von Seiten, die während der letzten k Speicherzugriffe benutzt wurden. Man nennt diese Menge Seiten w den **Arbeitsbereich** (*Working Set*) und beschreibt ihn als Funktion von k und t. Der Arbeitsbereich steigt zu Beginn der Programmausführung stark an, danach flacht der Anstieg ab, und die Kurve nähert sich asymptotisch ihrem Endwert, dem maximal erforderlichen Speicher für diesen Programmlauf. Bild 3.4 zeigt den prinzipiellen Verlauf der Arbeitsbereichskurve ab einem bestimmten Zeitpunkt t.

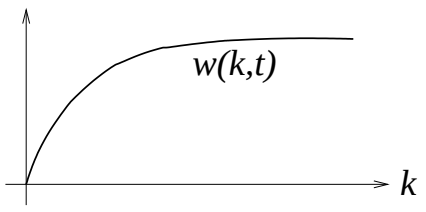
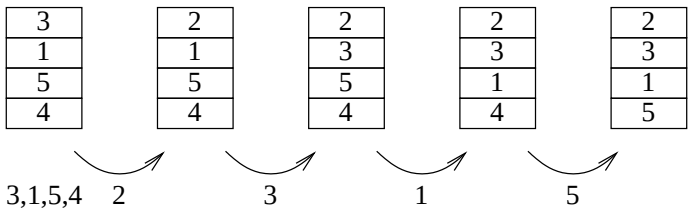


Abbildung 3.4: Verlauf des Arbeitsbereichs $w(k,t)$

Ein ähnlicher Kurvenverlauf kann für $w(k,t)$ zu jedem beliebigen Zeitpunkt t gemessen werden. Nach einer gewissen Anzahl von Ausführungen vergrößert sich der Arbeitsbereich nur noch geringfügig, d.h. es werden nur noch selten Seitenfehler auftreten, da die benötigten Seiten bereits geladen sind.

Falls jedoch der Hauptspeicher zu klein ist, um den benötigten Arbeitsbereich (*Working Set*) aufzunehmen, werden die zuletzt geladenen Seiten andere benötigte Seiten verdrängen, sodass die Anzahl der Seitenfehler auf hohem Niveau stagniert. Dieses Verhalten wird in der Literatur als **Flattern** (*Thrashing*) bezeichnet. Die Ausführungszeit des Programms wird stark ansteigen im Vergleich zur Ausführungsdauer auf einem vergleichbaren System mit größerem Hauptspeicher.

Beispiel 3.2 *Thrashing*
Ein Programm bekommt vom Betriebssystem 4 Seitenrahmen zugewiesen. Die Reihenfolge der Zugriffe sei gegeben durch: `3,1,5,4,2,3,1,5`. Bei Anwendung der LRU-Strategie (Least Recently Used) (siehe nächster Abschnitt 3.2.4) ergibt sich folgender Verlauf des Working-Sets:



Jeder Zugriff löst einen Seitenfehler aus, da der Arbeitsbereich des Programms größer ist als 4 Seiten. Es kommt zu *Thrashing*, und die Ausführungsgeschwindigkeit sinkt um mehrere Größenordnungen.

Die Situation verbessert sich sofort, wenn dem Programm ein Seitenrahmen zusätzlich zur Verfügung gestellt wird. Nach der Initialisierung befindet sich der Arbeitsbereich komplett im Speicher, und es tritt kein weiterer Seitenfehler auf. □

Falls mehrere Prozesse im Zeitscheibenverfahren bearbeitet werden, sollte das Ein- und Auslagern (**Swapping**) von kompletten Prozessdaten beim Ablauf einer Zeitscheibe vermieden werden. Um Thrashing zu vermeiden, sollte also jeder aktive Prozess möglichst einen ausreichend großen Bereich des Hauptspeichers zur Verfügung haben, aus dem er nur in Ausnahmefällen verdrängt werden kann. Andererseits sollte der zugewiesene Bereich auch nicht zu großzügig bemessen werden, damit möglichst viele Prozesse gleichzeitig im Hauptspeicher gehalten werden können. Die Größe dieses Bereiches kann mit Hilfe der Überlegungen zum Arbeitsbereich dynamisch angepasst werden. Folgender Algorithmus realisiert einen Abgleich:

1. Jeder Prozess darf seinen kompletten Arbeitsbereich im Speicher halten. Falls der Hauptspeicher dafür nicht ausreicht, werden einige Prozesse komplett ausgelagert.
2. Nach jedem Seitenfehler wird die neue Seite zum Arbeitsbereich hinzugefügt.
3. Falls der aktuelle Arbeitsbereich Seiten enthält, die während eines Zeitfensters t nicht angesprochen wurden, wird nach LRU-Strategie (siehe nächster Abschnitt 3.2.4) eine solche Seite verdrängt. Falls jedoch alle Seiten des bisherigen Arbeitsbereichs innerhalb von t angesprochen wurden, wird der Arbeitsbereich um die neue Seite erweitert.
4. Damit der Arbeitsbereich nicht beliebig wächst, werden die beiden am längsten nicht benutzten Seiten außerhalb des Zeitfensters t aus dem Arbeitsbereich entfernt.

Kritisch bei diesem Verfahren ist vor allem das Festlegen der Größe des Zeitfensters t. Das Fenster sollte groß genug gewählt werden, um den meisten Programmen einen ausreichend großen Arbeitsbereich zur Verfügung zu stellen. Die Ermittlung dieses Wertes erfolgt meist experimentell. Selbstverständlich bezieht sich t auf die *lokale* Bearbeitungszeit eines Prozesses, d.h. für einen inaktiven oder wartenden Prozess steht die Zeit still. Anstelle eines Zeitfensters kann auch eine bestimmte Anzahl von Instruktionen als Kriterium verwendet werden.

Als Alternative zum oben beschriebenen Algorithmus kann auch das **Seitenfehlerfrequenzverfahren** (*Page-Fault-Frequency-Method*) verwendet werden. Die Idee basiert auf der Beobachtung, dass ein

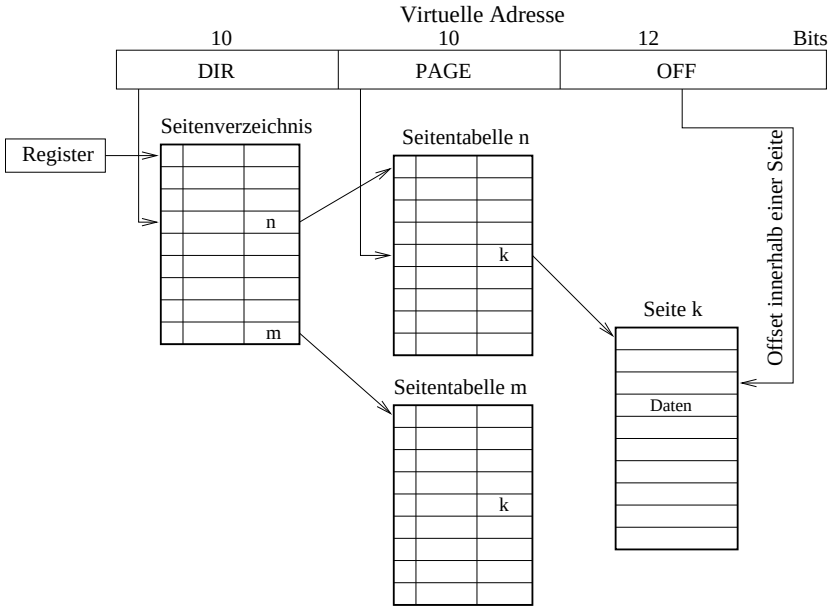
Programm in unregelmäßigen Abständen seinen Arbeitsbereich grundlegend verändert. Dieser Umbruch wird durch ein gehäuftes Auftreten von Seitenfehlern angezeigt. In einer solchen Situation kann der gesamte bisherige Arbeitsbereich verdrängt werden. Diese Überlegungen führen zu folgendem Algorithmus:

1. Beim Auftreten eines Seitenfehlers wird die Seitenfehlerfrequenz geschätzt. Eine grobe Schätzung kann durch $1/(t_1 - t_0)$ erfolgen, wobei t_1 die aktuelle Zeit und t_0 den Zeitpunkt des letzten Seitenfehlers markiert. Durch Mittelwertbildung über mehrere Seitenfehler kann die Genauigkeit erhöht werden.
2. Falls die geschätzte Frequenz einen vordefinierten Schwellwert Θ überschreitet, dann befindet sich der Prozess entweder im Umbruch oder der Arbeitsbereich ist zu klein. Die neue Seite wird daher zum Arbeitsbereich hinzugefügt.
3. Falls Θ nicht überschritten wird, befindet sich der Prozess in einer stabilen Phase. Die neue Seite verdrängt dann eine Seite des Arbeitsbereichs nach LRU-Strategie.
4. Falls Θ über einen längeren Zeitraum nicht erreicht wird, werden einige Seiten nach LRU-Strategie aus dem Arbeitsbereich entfernt.

Bedingt durch Regel 2 wird der Arbeitsbereich im Falle eines Umbruchs zunächst vergrößert. Sobald der Umbruch jedoch abgeschlossen ist, wird Regel 4 den Arbeitsbereich wieder auf das benötigte Volumen reduzieren.

Um die Zugriffszeiten beim Übersetzen von der virtuellen in die physikalische Adresse so gering wie möglich zu halten, wird zum Speichern der Seitentabelle oft ein sog. **Adressübersetzungspuffer** (*Translation-Lookaside Buffer*) (**TLB**) verwendet (siehe Abschnitt 3.2.7).

Beispiel 3.3 *Lineare Adressabbildung bei den Intel Prozessoren 80x86*
Bei den Intel Prozessoren 80x86 wird die Seitennummer in mehrere Teile zerlegt, sodass eine zweistufige Struktur des linearen virtuellen Speichers entsteht. Jede einzelne Seitentabelle enthält 1024 Einträge. Die folgende Abbildung zeigt die Struktur der Adressübersetzung.



Ein globales Register des 80x86 zeigt auf ein Seitenverzeichnis (Pagedirectory). Mit Hilfe der oberen 10 Bit der virtuellen Adresse wird ein Eintrag im Seitenverzeichnis (n) ausgewählt der dann auf eine Seitentabelle zeigt. Die mittleren 10 Bit der virtuellen Adresse markieren den Zeiger in der Seitentabelle (k), sodass nun die physikalsche Adresse des Seitenrahmens bestimmt ist. Beim 80x86 ist

die Größe eines Seitenrahmens auf 4 KByte festgelegt, dementsprechend hat der Offset einen Umfang von 12 Bit. Im Gegensatz zum 80x86 ist beim 68030 von Motorola die Seitengröße zwischen 256 Byte und 32 KByte in Zweierpotenzen wählbar. □

3.2.4 Dynamische Speicherverwaltung

Idealerweise sollte das Working-Set sich vollständig im Haupspeicher befinden. Da der virtuelle Speicherbereich des Prozessors größer ist als der tatsächlich vorhandene physikalische Speicher und verschiedene auf dem Prozessor ablaufende Prozesse miteinander konkurrieren, wird der Prozessor manchmal auf eine Seite zugreifen, die sich momentan nicht im Arbeitsspeicher (HSP) befindet. Dieses Ereignis wird als **Seitenfehler** (*Page-Fault*) bezeichnet. Bei der Lösung dieses Problems ist das Phänomen der **referentiellen Lokalität** bei der Programmausführung eine wesentliche Voraussetzung zum Erreichen einer guten Leistung. Untersuchungen anhand verschiedener Programmtypen haben gezeigt, dass sich die Zugriffswahrscheinlichkeit an einer bestimmten Stelle im Programm auf einige wenige Adressbereiche konzentriert (Abb. 3.5). Hierbei fällt jedoch auf, dass die Bereiche mit hoher Zugriffswahrscheinlichkeit im gesamten Adressraum eines Prozessors verteilt sind. Bereiche mit hoher Zugriffswahrscheinlichkeit sind beispielsweise die nachfolgenden Adressen des gerade ausgeführten Befehls, da ein Programm mit großer Wahrscheinlichkeit sequentiell abgearbeitet wird. Dieses Merkmal kann der im Abschnitt 3.1.1 beschriebenen *räumlichen* Lokalität zugeordnet werden.

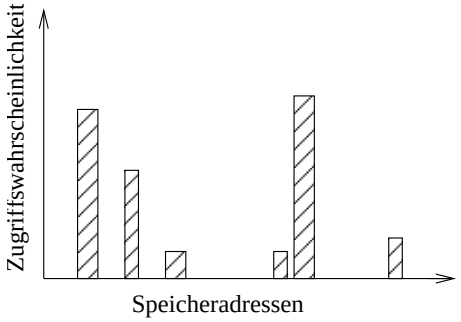


Abbildung 3.5: Beispiel für lokales Verhalten von Programmen

Eine hohe Zugriffswahrscheinlichkeit erreicht auch der Speicherbereich für globale Daten, allerdings kann dieser Bereich an einer völlig anderen Stelle im Speicher plaziert sein. Diese ungleichmäßige Verteilung der Zugriffswahrscheinlichkeiten kann zu einer kostengünstigen Leistungssteigerung genutzt werden. Es ist offensichtlich nicht erforderlich, den gesamten Hauptspeicher für die volle Prozessorgeschwindigkeit auszulegen, sondern es genügt, wenn die wenigen Bereiche mit hoher Wahrscheinlichkeit für einen schnellen Zugriff bereitgehalten werden.

Die Speicherverwaltung kann einen Seitenfehler an einem speziellen Statusbit in der Seitentabelle erkennen. In einem solchen Fall muss die gesuchte Seite vom Massenspeicher in den Hauptspeicher geladen werden. Die Dauer des Ladevorgangs kann speziell bei hoch getakteten RISC-Prozessoren die Größenordnung von 1 Million Befehlen erreichen. Es verbietet sich daher, die CPU bis zur Durchführung des Ladevorgangs warten zu lassen. Statt dessen meldet die Speicherverwaltung das Problem an das Betriebssystem. Dieses verwaltet die Seitentabellen und initiiert den notwendigen Ladevorgang. Danach wird der zuletzt bearbeitete Prozess für die Dauer des Ladevorgangs stillgelegt. Die freie Zeit kann nun bei Bedarf zur Bearbeitung eines anderen Prozesses genutzt werden.

Beim Auftreten eines Seitenfehlers stellt sich die Frage, welche Seite aus dem Hauptspeicher verdrängt werden soll. In diesem Punkt entspricht der Hauptspeicher einem voll assoziativen, d.h. völlig wahlfreien Speicher, da jede Seite an einen beliebigen Seitenrahmen kopiert werden kann. Es kommen daher verschiedene Verdrängungsstrategien in Betracht. Das wichtigste Prinzip stellt das sog. **LRU**-Verfahren dar (LRU=least recently used). Dazu wird eine Liste geführt, in der die Seiten nach dem Zeitpunkt ihrer letzten Aktivität geordnet sind. Falls nun eine Seite entfernt werden muss, dann trifft es die Seite, dessen letzte Aktivität am weitesten zurückliegt.

Im Gegensatz zu einem Cache-Miss beim Caching verursacht ein Seitenfehler beim virtuellem Speicher einen um Größenordnungen höheren Zeitaufwand zum Nachladen, d.h. selbst kleinste Verbesserungen der Trefferrate erhöhen die Gesamtleistung merklich. Andererseits steht bei einem Seitenfehler viel mehr Zeit zur Bearbeitung eines Ersetzungsalgorithmus zur Verfügung. Betriebssysteme mit virtueller Speicherverwaltung erlauben meist die quasi-nebenläufige Bearbeitung von mehreren Prozessen. Dies muss bei der Ersetzungsstrategie berücksichtigt werden, da eine gute Strategie für einzelne Prozesse nicht unbedingt die Gesamtleistung des Systems verbessert.

Die LRU-Strategie funktioniert in der Regel bis auf einige pathologische Fälle (siehe Beispiel 3.2) sehr gut. Eine effiziente Implementierung von LRU muss jedoch durch die Hardware unterstützt werden, da aus Zeitgründen nicht bei jedem Speicherzugriff eine Verwaltungsroutine zur Dokumentierung vergangener Zugriffe aktiviert werden kann. Als Alternative kann daher auch ein „First-In-First-Out“ (**FIFO**) Verfahren angewendet werden, bei dem immer die zuerst geladene Seite verdrängt wird, unabhängig von der Anzahl der darauf ausgeführten Zugriffe. Diese Strategie ist zwar einfacher zu realisieren als LRU, da beim Ladevorgang ohnehin ein Verwaltungsaufwand erforderlich ist, dafür ist die Leistung jedoch meist schlechter. Besonders bemerkenswert ist, dass FIFO zu den Algorithmen gehört, welche die Anzahl der Seitenfehler eventuell erhöhen, wenn einem Prozess mehr Seitenrahmen zur Verfügung gestellt werden. Dieses Verhalten ist auch als **Beladys Anomalie** bekannt. Abb. 3.6 stellt dieses Phänomen graphisch dar.

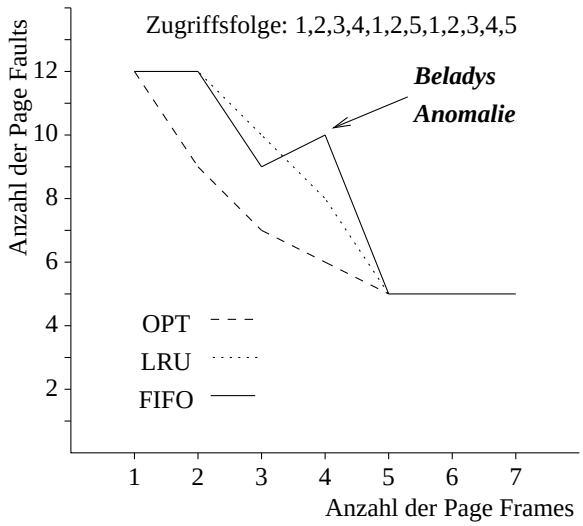


Abbildung 3.6: Seitenfehler-Kurve für verschiedene Algorithmen

Bei der Ersetzungsstrategie **OPT** handelt es sich um die theoretisch optimale, praktisch jedoch nicht realisierbare Ersetzungsstrategie. OPT führt stets zu einer minimalen Anzahl von Seitenfehlern, da OPT immer die Seite verdrängt, deren Bedarf am weitesten in der Zukunft liegt. Und genau aus diesem Grund ist diese Ersetzungsstrategie OPT nicht realisierbar, die Anwendung von OPT erfordert das vollständige Wissen über alle *zukünftigen* Seitenzugriffe.

Neben den bisher beschriebenen Verfahren kann auch eine *globale* LRU-Strategie zur Seitenverwaltung verwendet werden. Hierbei entfällt dann die Zuteilung von Speicherbereichen an einzelne Prozesse, statt dessen verdrängt eine neue Seite die am längsten nicht benutzte Seite des Systems ohne Rücksicht auf die Prozesszugehörigkeit. Allerdings besteht dann bei knapp dimensionierter Hauptspeichergröße die Gefahr der Verdrängung wartender Prozesse. Als Grundvoraussetzung für eine gute Systemleistung gilt daher bei allen Algorithmen, dass die Hauptspeichergröße ausreichend zum Speichern der jeweiligen Arbeitsbereiche sein muss.

Aufgabe 3.1 *Ersetzungsstrategien zur Seitenverwaltung*
Ein Computersystem habe 16 Seiten virtuellen Adressraum, aber nur 4 Seitenrahmen. Die Zugriffsfolge der virtuellen Seiten betrage: `0,7,2,7,5,8,9,2,4,5,9`.
Wieviele Seitenfehler entstehen bei OPT, LRU und FIFO Strategie? ◇

Aufgabe 3.2 *Reduktion von Seitenfehlern*
Welche der folgenden Maßnahmen tragen in der Regel dazu bei, die Häufigkeit von Seitenfehlern zu reduzieren?

a) Vergrößerung des Hauptspeichers

b) Vergrößerung des Sekundärspeichers

c) Verbesserung der Verdrängungsstrategie bei der Auslagerung von Seiten. ◇

3.2.5 Fragmentierung des Speichers

In der Regel wird ein Anwenderprogramm samt den dazugehörigen Daten keine ganze Zahl an Seiten füllen, die letzte Seite wird mehr oder weniger nur teilweise genutzt. Abhängig von der Seitengröße ist dieser Verlust mehr oder weniger groß. Die Aufteilung des Speichers in Seiten bringt zwangsläufig eine gewisse interne **Fragmentierung** des Speichers mit sich, ein Teil des Speichers bleibt stets ungenutzt, da Programme und Daten meist nicht genau den Seitenrahmen füllen.

Zur Auswahl einer geeigneten Seitengröße gibt es keine allgemein gültige Regel, sondern der Designer muss auch hier versuchen, einen Kompromiss zwischen sich widersprechenden Randbedingungen zu finden. Bei einer Seitengröße von s Datenworten werden daher im Mittel $s/2$ Datenworte pro Informationsblock (Programm oder eigenständige Datenstruktur) ungenutzt bleiben (*Wort* bezeichnet hier allgemein die kleinste adressierbare Einheit). Der Effekt der Fragmentierung kann durch eine kleine Seitengröße reduziert werden, da dadurch der vorhandene Hauptspeicher besser ausgenutzt werden kann. Dagegen spricht allerdings, dass bei kleinerer Seitengröße die Anzahl der Seitenrahmen steigt, und damit auch die Größe der Seitentabelle. Weiter wird dadurch der Zugriff auf den Plattenspeicher vergrößert, da die Suche länger dauert. Außerdem erhöht sich das Risiko eines Fehlgriffs im TLB (Adressübersetzungspuffer), wodurch die mittlere Zugriffsgeschwindigkeit auf den HSP sinkt.

Unabhängig von den bisherigen Überlegungen muss auch die Charakteristik des Massenspeichers berücksichtigt werden. Bei einer Festplatte zum Beispiel benötigt die Kopfpositionierung den größten Teil der Zugriffszeit. Wenn also mit einer Positionierung ein größerer Datenblock kopiert werden kann, so erhöht sich dadurch der Gesamtdurchsatz, da die längere Datenübertragungsdauer nur geringfügig zur Gesamtzeit beiträgt. Allerdings muss auch hierbei das Problem der Fragmentierung des Plattenspeichers beachtet werden.

3.2.6 Speichersegmentierung

Die in den vorherigen Abschnitten betrachtete Seitenadressierung fasst den virtuellen Speicher als eindimensionalen Adressraum, beginnend bei Adresse 0 und linear steigend bis zu einer maximalen Adresse, auf. Da in heutigen Rechnern mehrere Prozesse oder Anwenderprogramme parallel bearbeitet werden, ist es notwendig dem virtuellen Speicher eine Struktur zu geben um damit den Speicher flexibler einteilen zu können. Jedem Prozess i kann ein eigener virtueller Adressraum von Adresse 0 bis n_i zugewiesen werden. Diese prozessbezogenen Adressen werden als *logische Adressen* bezeichnet. Ein solcher einzelner Datenbereich oder logischer Adressraum wird als **Segment** bezeichnet, wobei die Größe eines Segments im allgemeinen vom Prozess selbst gewählt und dynamisch angepasst werden kann. Da nun z.B. die virtuelle Adresse 0 mehrfach auftritt, muss das Betriebssystem für jeden Prozess einen eigenen Segmentzeiger verwalten, sodass keine Überschneidung auftreten kann.

Ein Segment wird als logische Einheit mit definierter Funktion angesehen. Jedes Programm (oder auch jede einzelne Prozedur) kann ein eigenes Segment mit dem logischen Adressbereich von 0 bis n bekommen und wird dadurch völlig unabhängig von der Position im linearen virtuellen Adressraum und von der Position im physikalischen Speicher, was eine einfache Relozierbarkeit ermöglicht. Weiter können für jedes Segment bestimmte Schutzbits festgelegt werden, die dessen Benutzung durch verschiedene andere Prozesse festlegen. Das Konzept der Segmentierung ist also prinzipiell zu unterscheiden von einer Speicherverwaltung mit Seitenaufteilung.

Darüber hinaus muss das Konzept der Segmentierung im Gegensatz zum Paging auch vom Programmierer unterstützt werden. Die Möglichkeit der Funktionszuordnung an einzelne Segmente erlaubt die einfache Realisierung von Schutzmaßnahmen, z.B. kann ein bestimmtes Segment mit einem Nur-Lese-Status versehen werden, um Manipulationen zu verhindern. Auch die gemeinsame Nutzung von Datenbereichen kann durch Segmentierung optimiert werden. Jeder Prozess hat seine eigene individuelle Sicht auf den Speicher, ohne auf globale Randbedingungen Rücksicht nehmen zu müssen.

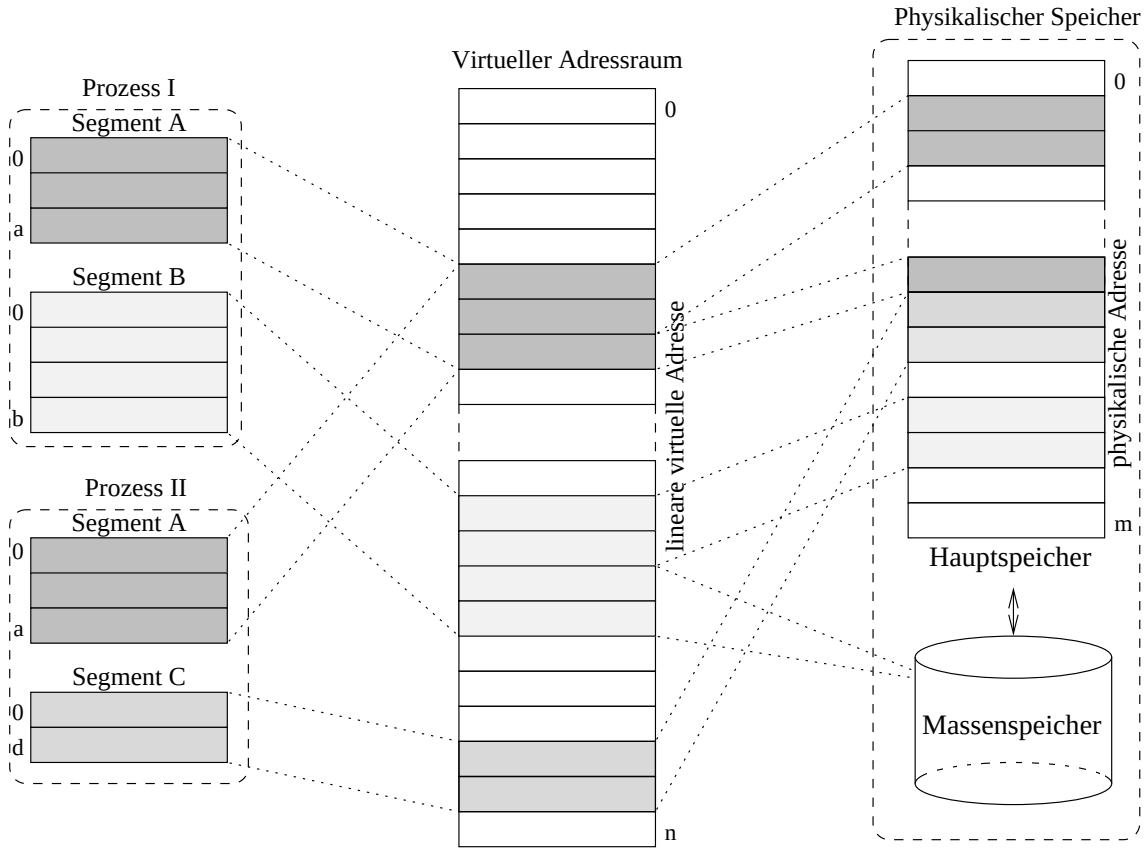
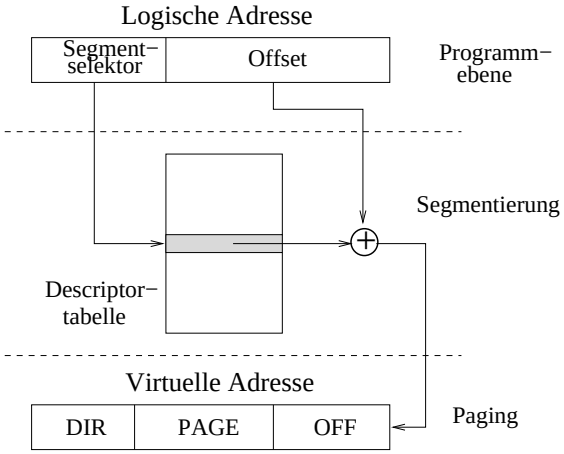


Abbildung 3.7: Kombination von Segmentierung und Seitenverwaltung

Bild 3.7 verdeutlicht graphisch die Kombination von segmentiertem und linear virtuellem Speicher. Aus Sicht der dargestellten Prozesse I und II besteht der Speicher aus jeweils zwei Segmenten. Die Adressierung innerhalb eines Segments, der Segmentoffset, beginnt immer bei Adresse 0. Die Speicherverwaltung erkennt, dass beide Prozesse das gleiche Segment A benutzen und hat deshalb die entsprechenden Blöcke nur einmal im Speicher vorgesehen. Diese Situation kommt häufig dadurch zustande, dass gleiche Programme auf unterschiedlichen Datensätzen arbeiten. Das gemeinsame Segment A wäre also jeweils der Programmcode, wobei sich die einzelnen Prozesse der Mehrfachbenutzung nicht bewusst sind. Zusätzlich unterhält jeder Prozess noch ein Datensegment unterschiedlicher Größe.

Auf dem Weg zur physikalischen Adresse wird zunächst aus der Kombination von Segmentoffset und Segmentanfangsadresse die lineare virtuelle Adresse ermittelt. Um die Berechnungszeiten zu minimieren, wird die Segmentierung in der Regel durch spezielle CPU Register unterstützt, sodass die lineare virtuelle Adresse ohne zusätzlichen Speicherzugriff direkt zur Verfügung steht. Diese wird dann nach dem bereits geschilderten Paging-Verfahren mit Hilfe von Seitentabellen in eine physikalische Speicheradresse übersetzt.

Beispiel 3.4 *Segmentierung bei den Intel Prozessoren 80x86*
Bei den Intel 80x86-Prozessoren besteht eine logische Adresse aus einem *Segmentselektor* und einer *Verschiebung* (Offset). In der Segmentierungsphase wird mit dem Segmentselektor aus einer *Deskriptortabelle* die Segment-Basisadresse gelesen.



Die Segment-Basisadresse ergibt zusammen mit dem Offset eine virtuelle Adresse. Aus der virtuellen Adresse wird nach dem in Beispiel 3.3 gezeigten Verfahren die entsprechende physikalische Adresse ermittelt. □

3.2.7 Adressübersetzungspuffer (Translation-Lookaside-Buffer, TLB)

Das softwaremäßige Durchlaufen der Übersetzungstabellen dauert verhältnismäßig lange, sodass eine Hardwareunterstützung wichtig ist. Eine schnelle Adressumsetzung wird durch eine Hardwaretabelle erreicht, welche die wichtigsten Adressumsetzungen enthält und als **Adressübersetzungspuffer** (*Translation Look Aside Buffer – TLB*) bezeichnet wird. Es handelt sich hierbei um eine Art Cache, der einen Teil der Einträge der Seitentabelle für den schnellen Zugriff bereithält. In diesem TLB wird die virtuelle Seitennummer als Adresse in einer Tabelle verwendet. Als Ergebnis des Zugriffs wird die physikalische Adresse der Seite geliefert, sofern diese Seite bereits in den Hauptspeicher geladen wurde. Der TLB enthält in der Regel 32 bis 128 Einträge für die zuletzt durchgeführten Adressumsetzungen und wird vollassoziativ organisiert, d.h. die Einträge können an beliebiger Stelle platziert werden. Der Vergleich der zu übersetzenden Adresse geschieht simultan in einem Takt mit allen im TLB gespeicherten Einträgen. Meist ist der TLB um zusätzliche Logik zur Seitenverwaltung und für den Zugriffsschutz erweitert. Man spricht dann von einer **Speicherverwaltungseinheit** (*Memory Management Unit* – MMU).

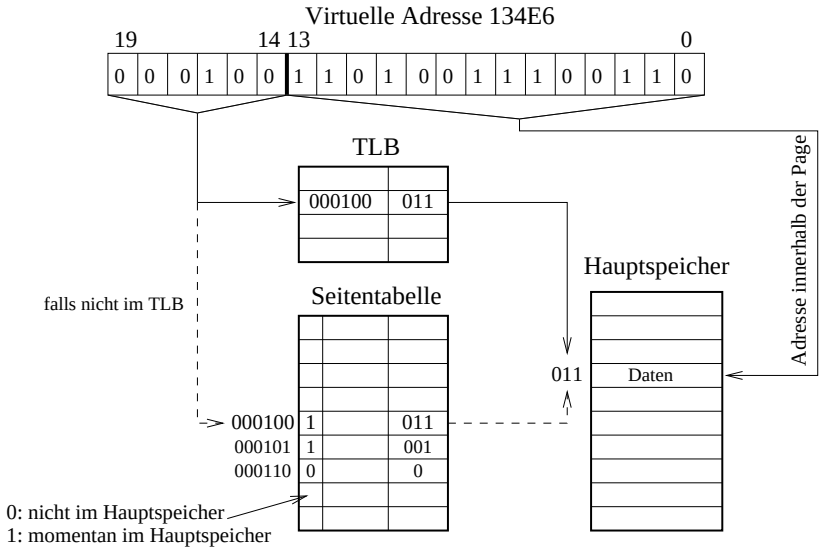
Ist ein gesuchter Eintrag im TLB nicht vorhanden – also ein TLB-Fehlzugriff (*TLB-Miss*) –, so wird eine Unterbrechung ausgelöst, welche die Übersetzungstabellen der virtuellen Speicherverwaltung durchläuft, um die physikalische Adresse herauszufinden, und dann das Adresspaar in den TLB einträgt. Ein TLB-Eintrag besteht aus einem sog. **Tag** (Etikett), das einen Teil der virtuellen Adresse enthält, einem Datenteil mit einer physikalischen Seitennummer (*physical page frame number*) sowie weiteren Verwaltungs- und Schutzbits für:

- die Seite befindet sich im Hauptspeicher (*resident*),
- der Zugriff ist nur dem Betriebssystem erlaubt (*supervisor*),
- Schreibzugriff ist verboten (*read-only*),
- die Seite wurde verändert (*dirty*) oder
- ein Zugriff auf die Seite ist erfolgt (*referenced*).

Bei heutigen Prozessoren gibt es auf dem Prozessor-Chip getrennte MMUs mit eigenen TLBs für den Befehls- und den Daten-Cache. Damit kann die Adressübersetzung für Code und Daten parallel zueinander durchgeführt werden. Die TLB-Zugriffe geschehen meist im gleichen Takt, in dem auch auf die Primär-Cache-Speicher zugegriffen wird. Eine Erweiterung stellt der *Tagged* TLB dar, bei dem jeder TLB-Eintrag zusätzlich einen Prozess-Tag enthält, mit dem die Adressräume der verschiedenen Prozesse unterschieden werden können. Im Falle eines Prozesswechsels müssen die TLB-Einträge nicht gelöscht werden (*TLB flush*), sondern werden nach Bedarf verdrängt.

Beispiel 3.5 *Übersetzung einer virtuellen Adresse mit TLB*
Die folgende Abbildung zeigt die Übersetzung einer virtuellen Adresse mit Hilfe eines TLB. Gesucht wird der Speicherinhalt, der unter der virtuelle Adresse $134E6|_{16}$ abgelegt ist.
Die oberen 6 Bit dienen als Zeiger innerhalb der Seitentabelle und ermöglichen sozusagen als Ektikett (Tag) alleine die vollständig Zuordnung zur physikalischen Adresse. Ein Teil der Seitentabelle wird zusätzlich für schnellen Zugriff im TLB bereitgehalten.

Um die Übersetzung beschleunigt auszuführen, wird der gewünschte Eintrag 000100 zuerst in der TLB gesucht. Ist der Eintrag vorhanden, dann führt dieser direkt auf die Seitenanfangsadresse im HSP (durchgezogene Linie).
Befindet sich der gewünschte Eintrag nicht in der TLB, dann wird dieser aus der Seitentabelle selektiert (gestrichelte Linie). Da der virtuelle Adressraum größer ist als der Hauptspeicher, enthält die Seitentabelle ein Bit zur Markierung, ob sich die jeweilige Seite momentan im Hauptspeicher befindet oder nicht. Ferner wird anschließend die TLB mit dem gewünschten Eintrag aktualisiert. Die physikalische Hauptspeicheradresse zur virtuellen Adresse $134E6|_{16}$ ergibt sich also zu $0F4E6|_{16}$.



□

3.2.8 Seitenverlust durch Segmentierung mit Seitenübersetzung

Angenommen in einem Rechensystem werde Segmentierung mit Seitenübersetzung angewendet. Aufgrund der bearbeiteten Prozesse stelle sich eine mittlere Segmentgröße von s Datenworten ein. Da die sich einstellende Segementgröße s im allgemeinen kein Vielfaches einer bestimmten, gewählten Seitengröße von p Speicherworten ist, können in der letzten Seite jedes Segments einige Speicherworte nicht ausgenutzt werden. Außerdem wird zusätzlicher Speicherplatz für die Seitentabelle eines jeden Segments benötigt.

Aus dieser Argumentation folgt, dass je kleiner die Seitengröße gewählt wird, um so kleiner ist der Verlust an nicht nutzbaren Speicherworten in der letzten Seite eines Segments. Andererseits befinden sich dann aber mehr Seiten innerhalb eines Segments und umso mehr Einträge besitzt die Seitentabelle. Der Speicherverlust S_v durch die Segmentierung mit Seitenübersetzung besteht aus zwei Anteilen, dem Verschnitt in der letzten Seite und dem Verlust durch die Seitentabelle, die zueinander gegenläufig sind.

Bei einer mittleren Segmentgröße von s Worten und einer Seitengröße von p Worten enthält ein Segment im Mittel s/p Seiten. Nehmen wir an, dass innerhalb eines Segments für einen Eintrag in der Seitentabelle, d.h. pro Seite x Datenworte benötigt werden, dann beträgt der Speicherverlust durch die Seitentabelle

$$x \cdot \frac{s}{p} \quad \text{Datenworte.}$$

Der Speicherverlust durch den Verschnitt in der letzten Seite ist gleichverteilt zwischen Null und einer ganzen Seite, er beträgt daher im Mittel eine halbe Seitengröße $p/2$. Die Gesamtverschwendung S_v ist somit:

$$S_v = \frac{1}{2}\,p + x \cdot \frac{s}{p} \quad \text{Datenworte.}$$

Da die beiden summierten Effekte für den Speicherverlust gegenläufig sind, muss sich in Abhängigkeit von der gewählten Seitengröße p ein Minimalwert ergeben. Diesen Minimalwert können wir durch Differentiation von S_v nach der Seitengröße p ermitteln:

$$\frac{\partial S_v}{\partial p} = \frac{1}{2} + x \cdot \frac{-s}{p^2}$$

Nullsetzen der Ableitung ergibt:

$$\frac{\partial S_v}{\partial p} = \frac{1}{2} + x \cdot \frac{-s}{p^2} \stackrel{\text{def}}{=} 0 \quad \Longrightarrow \quad \frac{1}{2} = x \cdot \frac{s}{p^2} \quad \Longrightarrow \quad p = \sqrt{2xs}$$

Der Seitenverlust wird also minimiert, wenn eine Seitengröße von $\sqrt{2xs}$ Worten gewählt wird.

Bild 3.7 verdeutlicht aber auch noch eine weitere Problematik des Segmentierens. Die einzelnen Segmente benötigen jeweils einen zusammenhängenden Adressraum, wobei es ausreicht einen virtuellen Adressraum zu reservieren. Trotzdem muss für jedes neu angeforderte Segment ein unbenutzter Bereich in der erforderlichen Größe gefunden werden. Hierzu gibt es verschiedene Möglichkeiten:

- Beim **first fit** Verfahren wird der Bereich gewählt, der als erstes in ausreichender Größe gefunden wird. Die Suchzeit ist demnach minimal.
- Mit **best fit** wird der kleinstmögliche Bereich in ausreichender Größe benutzt. Da die geforderte Größe jedoch so gut wie nie genau in den freien Bereich passt, bleiben viele Bereiche, die zu klein sind um weitere Anfragen zu befriedigen, ungenutzt.
- **worst fit** dagegen verwendet immer den größten noch freien Bereich. Hierbei behalten auch die Reststücke eine ausreichende Größe. Allerdings kann manchmal eine sehr große Anfrage nicht mehr bearbeitet werden, da keine sehr großen freien Bereiche verbleiben.
- Die beste Methode besteht darin, die benutzten Segmente in regelmäßigen Abständen neu zu einem Block zusammenzupacken. Der Zeitaufwand für eine solche **Verschiebung** ist allerdings nur vertretbar wenn wie in Abb. 3.7 zusätzlich zur Segmentierung auch noch eine Seitenaufteilung benutzt wird. Beim Zusammenpacken werden dann keine Daten im Hauptspeicher hin- und herkopiert, sondern lediglich die Einträge in der Seitentabelle verändert.

Die Segmentierung des Speicherbereichs kann auch ohne den Zwischenschritt über einen linearen virtuellen Adressraum erfolgen, wie das nachfolgende Beispiel zeigt.

Beispiel 3.6 *Segmentierung beim Prozessor Intel 80286*
Der Prozessor 80286 hatte eine Struktur, bei der vier verschiedene Segmente unterschieden wurden. Ein Datensegment DS enthielt die zu bearbeitenden Daten, und das Codesegment CS enthielt den Programmcode. Darüberhinaus gab es ein spezielles Stacksegment SS zur Aufnahme der Stackdaten und ein sog. Extrasegment ES. Jede Segmentanfangsadresse wurde in einem speziellen CPU-Register bereitgehalten.
Die physikalische Hauptspeicheradresse wurde durch die Kombination des Segmentoffsets (hier *Insegment-Adresse* genannt) mit einem Segmentregister ermittelt. Problematisch war beim 80286 jedoch die maximale Segmentgröße von 64 KByte, wobei physikalisch bis zu 1 Mbyte (mit Schaltungskniff auch 4 MByte) direkt angesprochen werden können. Eine Aufteilung des Speichers in einzelne Seiten war beim 80286 nicht vorgesehen. □

Aufgabe 3.3 *Segmentierung*
In Abb. 3.7 wird jedes Segment als kompakter Block im virtuellen Adressraum repräsentiert. Gibt es eine Konstruktion, bei der auch die einzelnen Segmente *zerteilt* in kleine Blöcke im virtuellen Adressraum liegen können? ◇

Beispiel 3.7 *Segmentierung bei Intel 80x86-Prozessoren*
Bei den Intel 80x86-Prozessoren besteht eine logische Adresse aus einem *Segmentselektor* und einer *Verschiebung* (Offset). Es gibt eine sog. *Deskriptortabelle*, in der für jedes Segment ein sog. *Segmentdeskriptor* angelegt wird. Dieser Segmentdeskriptor enthält für ein bestimmtes Segment die Informationen über die virtuelle Segment-Basisadresse, unter der das Segment im Speicher beginnt und die Länge des Segments sowie über die erforderlichen Zugriffsrechte.

In der ersten Phase der Adressübersetzung wird die logische Adresse in eine virtuelle Adresse übersetzt. Zu diesem Zweck wird der Segmentselektor als Zeiger in der Deskriptortabelle verwendet, um einen Segmentdeskriptor auszuwählen, der dann die Basisadresse und die Länge des zu diesem Segmentselektor gehörenden Segments liefert. Die Bildung der virtuellen Adresse erfolgt durch Addition der virtuellen Segment-Basisadresse und der Verschiebung, die in der logischen Adresse enthalten ist.

In diesem Beispiel seien drei Segmente im Speicher angelegt, ein Segment CS für den Programmcode, ein Segment DS für Datenstrukturen und ein Segment SS für den Stapel (Stack). Bei drei Segmenten enthält die Deskriptortabelle drei Zeilen, jede Zeile stellt einen Segmentdeskriptor dar:

Segment	virtuelle Segment-Basisadresse	Länge (hexadezimal)
CS	3A17C000	1000
DS	D29E4000	4000
SS	51F60000	1000

Der Prozessor verfügt also über die drei Segmentregister CS, DS und SS.

Die Seitengröße betrage 4096 Worte, wobei der Hauptspeicher maximal 4 Seiten aufnehmen kann. Weiter nehmen wir an, dass Seiten der CS- und der SS-Segmente vom Betriebssystem nicht ausgelagert werden können. Als Ersetzungsstrategie kommt LRU zum Einsatz.
Auf diesem Prozessor soll der folgende Algorithmus ausgeführt werden:

```
(01)  FOR i = 0 TO 2^13 – 1
(02)  a = 0;
(03)  FOR j = 0 TO 2
(04)  a = a + x [ i+j ];
(05)  END
(06)  y [ i ] = a;
(07)  END
```

In diesem Programm wird die skalare Variable *a*, die Variablen *i* und *j* für ganze Zahlen sowie die Felder *X* und *Y* verwendet. Das Feld *X* beginne im Speicher an der logischen Adresse mit der Verschiebung 2000 im Datensegment DS, das gleiche gilt für das Feld *Y*, jedoch mit der Verschiebung 0. Auf die Werte der Variablen *i*, *j* und *a* kann beliebig zugegriffen werden, deren Zugriff wird hier nicht weiter betrachtet.

Die Seitentabelle enthalte zu Beginn folgende Einträge:

Seitenrahmen	logische Adresse		virtuelle
	dezimal	hexadezimal	Adresse (32 Bit)
0	24576 – 28671	6000 – 6FFF	`3A17C000 - 3A17CFFF`
1	28672 – 32768	7000 – 7FFF	`51F60000 - 51F60FFF`
2	08192 – 12287	2000 – 2FFF	`D29E6000 - D29E6FFF`
3	12288 – 16383	3000 – 3FFF	`D29E7000 - D29E7FFF`

Anhand der virtuellen Adressen in der Seitentabelle ist zu erkennen, dass im Seitenrahmen 0 des HSPs die Seite mit dem Codesegment CS (virtuelle Adressen `3A17C000 - 3A17CFFF`) und im Seitenrahmen 1 die Seite mit dem Stapelsegment SS (virtuelle Adressen `51F60000 - 51F60FFF`) eingelagert ist. Weiter ist in den Seitenrahmen 2 und 3 die zweite Hälfte des Datensegments DS (virtuelle

Adressen D29E6000 - D29E7FFF) eingelagert. Die erste Hälfte des Datensegments DS befindet sich momentan nicht im HSP.
Das das Feld X an der logischen Adresse mit der Verschiebung 2000 im DS beginnt, befinden sich die ersten 8192 Elemente von X im HSP, aber keine Elemente des Feldes Y.

Die Aufgabe besteht nun darin, ein Protokoll der ein- und ausgelagerten Seiten während der Ausführung des Programms zu erstellen. Dazu müssen wir den Zugriff auf die einzelnen Feldelemente $x(i)$ und $y(i)$ verfolgen. Bei der Ausführung des Programmes wird in der 4. Anweisung jeweils auf ein Feldelement von X und in der 6. Anweisung auf ein Element von Y zugegriffen.
Die äußere Schleife beginnt mit $i=0$ und läuft bis $i=2^{13}-1$. Sie umfasst eine innere Schleife, die jeweils mit $j=0$ beginnt und bis $j=2$ läuft und die 4. Anweisung enthält und nach dieser inneren Schleife wird die 6. Anweisung ausgeführt.
Im ersten Durchlauf der äußeren Schleife mit $i=0$ wird also in der inneren Schleife auf die Feldelemente $x(0)$, $x(1)$ und $x(2)$, und anschließend in der äußeren Schleife auf das Feldelement $y(0)$ zugegriffen.
Im zweiten Durchlauf der äußeren Schleife mit $i=1$ wird in der inneren Schleife auf die Feldelemente $x(1)$, $x(2)$ und $x(3)$, und anschließend in der äußeren Schleife auf das Feldelement $y(1)$ zugegriffen.
Im k-ten Durchlauf der äußeren Schleife mit $i=k-1$ wird in der inneren Schleife auf die Feldelemente $x(k-1)$, $x(k)$ und $x(k+1)$, und anschließend in der äußeren Schleife auf das Feldelement $y(k-1)$ zugegriffen.
Im letzten Durchlauf ($2^{13}=8192$) der äußeren Schleife mit $i=8191$ wird in der inneren Schleife auf die Feldelemente $x(8191)$, $x(8192)$ und $x(8193)$, und anschließend in der äußeren Schleife auf das Feldelement $y(8191)$ zugegriffen.

Die Feldelemente $x(0)$, $x(1)$ und $x(2)$ besitzen die logischen Adressen 2000, 2001 und 2002, die beim Programmstart im Seitenrahmen 2 des HSPs eingelagert sind. Das Feldelement $y(0)$ besitzt die logische Adresse 0000, die sich beim Programmstart nicht im HSP befindet und daher wird der Zugriff auf $y(0)$ im ersten Durchlauf einen Seitenfehler erzeugen. Da vorher in der inneren Schleife mit den Zugriffen auf $x(0)$, $x(1)$ und $x(2)$ auf den Seitenrahmen 2 zugegriffen wurde, wird nach LRU der Seitenrahmen 3 mit den derzeitigen logischen Adressen 3000 – 3FFF (D29E7000 - D29E7FFF) durch die Seite mit den logischen Adressen 0000 – 0FFF (D29E4000 - D29E4FFF) neu belegt.

Zeitpunkt	Seitenrahmen 2	Seitenrahmen 3
Beginn	2000 – 2FFF (D29E6000 - D29E6FFF)	3000 – 3FFF (D29E7000 - D29E7FFF)
1. Austausch im 1. Durchlauf, $i=0$	2000 – 2FFF (D29E6000 - D29E6FFF)	0000 – 0FFF (D29E4000 - D29E4FFF)

Nun kann die äußeren Schleife solange durchlaufen werden, bis auf die logische Adresse 3000 zugegriffen wird. Dies passiert im Durchlauf mit der hexadezimalen Nummer $i=$0FFE, da dabei in der inneren Schleife für $i+j$ ($j=2$) die hexadezimale Zahl 1000 erreicht wird. Da im Durchlauf der inneren Schleife davor ($i=$0FFE, $j=1$) auf die Seite mit den logischen Adressen 2000 – 2FFF zugegriffen wurde, wird nach LRU die Seite mit den logischen Adressen 0000 – 0FFF (D29E4000 - D29E4FFF) gegen die Seite mit den logischen Adressen 3000 – 3FFF (D29E7000 - D29E7FFF) im Seitenrahmen 3 ausgetauscht.

Zeitpunkt	Seitenrahmen 2	Seitenrahmen 3
Beginn	2000 – 2FFF (D29E6000 - D29E6FFF)	3000 – 3FFF (D29E7000 - D29E7FFF)
1. Austausch im 1. Durchlauf $i=0$	2000 – 2FFF (D29E6000 - D29E6FFF)	0000 – 0FFF (D29E4000 - D29E4FFF)
2. Austausch im Durchlauf mit $i=$0FFE, $j=2$, $i+j=$1000	2000 – 2FFF (D29E6000 - D29E6FFF)	3000 – 3FFF (D29E7000 - D29E7FFF)
3. Austausch im Durchlauf mit $i=$0FFE, $j=2$, $i+j=$1000	0000 – 0FFF (D29E4000 - D29E4FFF)	3000 – 3FFF (D29E7000 - D29E7FFF)

Anschließend wird im weiteren Verlauf dieser äußeren Schleife auf y(`0FFE`) zugegriffen, was dazu führt, dass nach LRU die Seite mit den logischen Adressen 2000 – 2FFF (`D29E6000 - D29E6FFF`) gegen die Seite mit den logischen Adressen 0000 – 0FFF (`D29E4000 - D29E4FFF`) im Seitenrahmen 2 ausgetauscht wird. Nach diesen Durchläufen enthält die Seitentabelle folgende Einträge:

Seitenrahmen	logische Adresse		virtuelle
	dezimal	hexadezimal	Adresse (32 Bit)
0	24576 – 28671	6000 – 6FFF	`3A17C000 - 3A17CFFF`
1	28672 – 32768	7000 – 7FFF	`51F60000 - 51F60FFF`
2	00000 – 04095	0000 – 0FFF	`D29E4000 - D29E4FFF`
3	12288 – 16383	3000 – 3FFF	`D29E7000 - D29E7FFF`

Bei Anwendung der LRU-Strategie wurden bis dahin 6 Seiten ein- bzw. ausgelagert. Die durchgeführten Aktionen können in einer Tabelle zusammengefasst werden:

	Programm-zeile	i	j	Aktion	Seiten-rahmen
1	6	`0000`	-	Seite `D29E7` auslagern, Seite `D29E4` anlegen	3
2	4	`0FFE`	2	Seite `D29E4` auslagern, Seite `D29E7` einlagern	3
3	6	`0FFE`	-	Seite `D29E6` auslagern, Seite `D29E4` einlagern	2

□

Aufgabe 3.4 *Segmentierung bei Intel 80x86-Prozessoren*
Verfolgen Sie die Ausführung des in Beispiel 3.7 gegebenen Programmes auf dem Prozessor ab dem Durchlauf der äußeren Schleife mit der Nummer $i =$ `0FFF` weiter bis eine Zugriffsverletzung auftritt. Die Seitentabelle zeigt zu diesem Zeitpunkt folgende Einträge:

Seitenrahmen	logische Adresse		virtuelle
	dezimal	hexadezimal	Adresse (32 Bit)
0	24576 – 28671	6000 – 6FFF	`3A17C000 - 3A17CFFF`
1	28672 – 32768	7000 – 7FFF	`51F60000 - 51F60FFF`
2	00000 – 04095	0000 – 0FFF	`D29E4000 - D29E4FFF`
3	12288 – 16383	3000 – 3FFF	`D29E7000 - D29E7FFF`

Geben Sie jeweils die Iteration und die Zeilennummer an, bei der ein Seitenfehler auftritt. Wie viele Seitenwechsel werden insgesamt durchgeführt? ◇

Aufgabe 3.5 *Segmentierung bei Intel 80x86-Prozessoren*
Wir betrachten weiterhin die Ausführung des in Beispiel 3.7 gegebenen Programmes unter der dort gegebenen Umgebung. Die Frage lautet:
Gibt es eine Strategie zur Seitenersetzung, die besser ist als LRU?

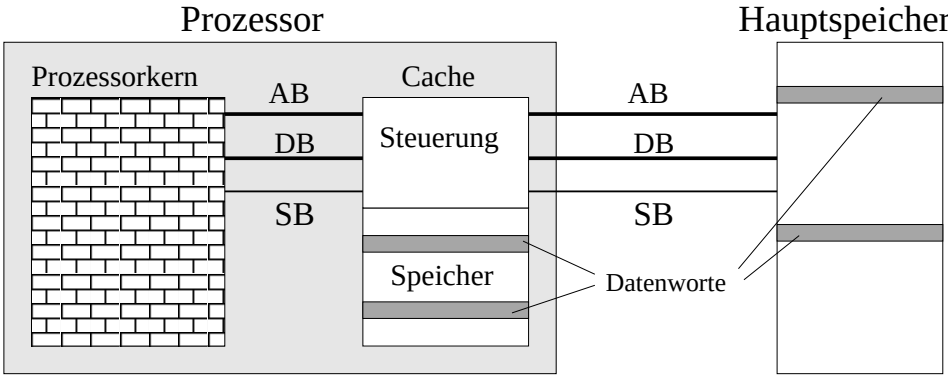
Um diese Frage zu beantworten, müssen Sie den Programmablauf verfolgen und damit immer die Seiten austauschen, die so weit wie möglich in der Zukunft nicht benötigt werden.

Wenn es für das gegebene Programm eine solche Strategie gibt, wie viele und welche Ein-/Auslagerungen müssen dann durchgeführt werden? ◇

3.3 Aufbau und Funktion von Cache-Speichern

„Cache“ bedeutet in wörtlicher Übersetzung „Versteck“, man könnte also eine verdeckte oder versteckte, transparente Vorratskammer damit meinen. Mit dem Prozessor-Cache beispielsweise soll aber ein zu langsamer Hauptspeicher neutralisiert werden. Unter einem **Cache** oder **Cache-Speicher** versteht man einen kleinen, schnellen Pufferspeicher, in dem Kopien derjenigen Teile des Hauptspeichers oder eines Hintergrundspeichers gepuffert werden, auf die aller Wahrscheinlichkeit nach vom Prozessor als nächstes zugegriffen wird.

Die Festlegung und Aktualisierung der Inhalt in diesem kleinen, aber schnellen Pufferspeicher basiert direkt auf dem in Abschnitt 3.1.1 betrachteten Lokalitätsprinzip, das sich in die zeitliche und räumliche Lokalität aufsplitten lässt. Beide Prinzipien können unabhängig voneinander beim Caching realisiert werden. Die zeitliche Lokalität wird realisiert, indem die aktuell oder in der nahen Vergangenheit verwendeten Speicherinhalte im Cache abgelegt werden. Die räumliche Lokalität wird realisiert, indem die zu den aktuell oder in der nahen Vergangenheit verwendeten Speicherinhalten adresslich benachbarten Speicherinhalte ebenfalls im Cache abgelegt werden.



AB=Adressbus, DB=Datenbus, SB=Steuerbus

Abbildung 3.8: Hierarchie aus Cache und Hauptspeicher

Auf den Cache-Speicher soll der Prozessor fast so schnell wie auf seine Register zugreifen können. Er ist deshalb bei heutigen Mikroprozessoren als sog. **Primär-Cache** (*first-level-*, *on-chip-*, *primary-cache*) direkt auf dem Prozessor-Chip angelegt oder als **Sekundär-Cache** (*secondary-level-cache*, *L2-Cache*) entweder ebenfalls auf dem Prozessor-Chip oder in der schnellen und teuren SRAM-Technologie realisiert. Daneben gibt es häufig noch einen L3-Cache, der meist zur Synchronisation der logisch darüber liegenden Caches in Systemen mit mehreren Prozessoren bzw. Prozessorkernen verwendet wird. Meist werden heute als Primär-Caches getrennte **Code-** und **Daten-Cache-Speicher** mit jeweils eigenen Speicherverwaltungseinheiten angewandt, um die Zugriffe durch die Befehlsholeeinheit und die Lade-/Speichereinheit zu entkoppeln. Die Cache-Speicher besitzen nur einen Bruchteil der Kapazität des mit dynamischen Speicherbausteinen (DRAM-Technologie) aufgebauten Hauptspeichers. Durch die Verwendung von Cache-Speichern wird die Lücke zwischen der hohen Zugriffsgeschwindigkeit auf Register und dem langsameren Zugriff auf den Hauptspeicher teilweise überbrückt. Hauptzweck eines Cache-Speichers ist es, die mittlere Zugriffszeit auf den Hauptspeicher zu verringern.

Aus Sicht des Prozessors arbeitet der Cache völlig transparent, d.h. der Prozessor benutzt nach wie vor ausschließlich die physikalischen Adressen des Hauptspeichers. Der Prozessor bemerkt lediglich eine Erhöhung der mittleren Zugriffsgeschwindigkeit. Die Organisation des Cache-Speichers, d.h. die Aktualisierung der Inhalte, erfolgt durch die **Cache-Speicherverwaltung**. Sie sorgt dafür, dass der Inhalt des Cache-Speichers in der Regel das Speicherwort enthält, auf das der Prozessor als nächstes zugreift. Die Cache-Speicherverwaltung muss sehr schnell sein und ist deshalb vollständig in Hardware realisiert. Durch eine Hardware-Steuerung (*Cache-Controller*) werden automatisch die Speicherwörter in den Cache kopiert, auf die der Prozessor zugreift. Die Cache-Speicherverwaltung muss

von der für die virtuelle Speicherverwaltung zuständigen Speicherverwaltungseinheit unterschieden werden, wenn auch beide meist koordiniert arbeiten.

3.3.1 Definition wichtiger Cache-Parameter

Man spricht von einem **Cache-Treffer** (*Cache-Hit*), falls das angeforderte Speicherwort im Cache-Speicher vorhanden ist, und von einem **Cache-Fehlzugriff** (*Cache-Miss*), falls das angeforderte Speicherwort nur im Hauptspeicher steht. Im Folgenden wird der Einfachheit halber von „Speicherwort“ gesprochen, wenn allgemein ein Speicherzugriff des Prozessors gemeint ist. Der Zugriff kann in gleicher Weise ein in ein Register zu ladendes oder aus einem Register zu speicherndes Datenwort als auch einen bzw. im Falle der Superskalarprozessoren sogar gleich mehrere aufeinander folgende Befehle betreffen.

Im Falle eines Cache-Fehlzugriffs wird eine bestimmte Speicherportion, die das angeforderte Speicherwort umfasst, aus dem Hauptspeicher in den Cache-Speicher geladen. Dies geschieht durch die Cache-Speicherverwaltung unabhängig vom Prozessor bzw. Prozessorkern. Der Prozessor hat die Illusion, immer auf den Hauptspeicher zuzugreifen, und merkt den Unterschied nur daran, dass der Zugriff im Falle eines Cache-Fehlzugriffs länger dauert. Für das Anwenderprogramm bleibt die Verwendung eines Cache-Speichers transparent.

Die **Cache-Zugriffszeit** t_{Hit} bei einem Treffer ist die Zeit bzw. die Anzahl der Takte, die benötigt wird, um ein Speicherwort im Cache zu identifizieren, die Verfügbarkeit und Gültigkeit zu prüfen und das Speicherwort der nachfolgenden Pipeline-Stufe zur Verfügung zu stellen. Da der Cache-Speicher nur einen Bruchteil des Hauptspeicherinhalts fasst, wird es zwangsläufig zu Anfragen der CPU nach Speicherinhalten kommen, die sich nicht im Cache befinden. Die **Fehlzugriffsrate** (*Miss-Rate*) ist der Prozentsatz der Cache-Fehlzugriffe bezogen auf alle Cache-Zugriffe. Die **Trefferrate** (*Hit-Rate*) ist der Anteil bzw. der Prozentsatz der Treffer beim Cache-Zugriff bezogen auf alle Cache-Zugriffe.

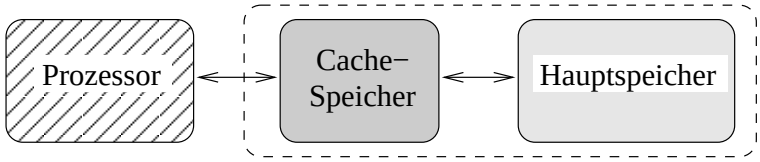
Der **Fehlzugriffsaufwand** (*Miss-Penalty*) t_{Miss} ist die Zeit, die benötigt wird, um einen Cache-Block von einer tiefer gelegenen Hierarchiestufe in den Cache zu laden und das Speicherwort dem Prozessor zur Verfügung zu stellen. Die **Zugriffszeit** (*access-time*) zur unteren Hierarchiestufe ist abhängig von der Latenz des Zugriffs auf der unteren Hierarchiestufe. Die **Übertragungszeit** (*transfer-time*) ist die Zeit, um den Block zu übertragen und hängt von der Übertragungsbandbreite und der Blocklänge ab.

Um den Nutzen eines Cache-Speichers quantitativ auszudrücken, muss die effektive **Zugriffszeit** T_{eff} aus Sicht der CPU ermittelt werden. Da die Zugriffszeit von dem zufälligen Ereignis abhängt, ob sich das gesuchte Datum im Cache befindet, ist es sinnvoll eine *mittlere* Zugriffszeit anzugeben. Dazu gewichten wir die entstehende Zugriffszeit t_{Hit} für den Fall, dass sich der benötigte Speicherinhalt im Cache befindet, mit der Wahrscheinlichkeit, dass nur auf den Cache zugegriffen werden muss. Diese Wahrscheinlichkeit ergibt sich direkt aus der Trefferrate. Die entstehende Zugriffszeit t_{Miss} für den Fall, dass sich der benötigte Speicherinhalt nicht im Cache befindet wird mit der komplementären Wahrscheinlichkeit, der Fehlzugriffsrate, gewichtet. Die **effektive Zugriffszeit** (*Average-Memory-Access-Time*) T_{eff} (in *ns* oder in Prozessortakten) wird also definiert als:

$$\begin{aligned} T_{\text{eff}} &= \text{Trefferrate} * \text{Cache-Zugriffszeit} + \text{Fehlzugriffsrate} * \text{Fehlzugriffsaufwand} \\ &= \text{Trefferrate} * t_{Hit} + (1 - \text{Trefferrate}) * t_{Miss} \end{aligned} \tag{3.1}$$

Je höher die Trefferquote, desto mehr verringert sich die mittlere Zugriffszeit. Dadurch beschleunigt sich sowohl der Zugriff auf den Programmcode – Programmschleifen stehen häufig vollständig im Cache – als auch der Zugriff auf die Daten.

Beispiel 3.8 *Effektive Zugriffszeit*
Die folgende Abb. zeigt den Aufbau eines Prozessorsystems mit einem einfachen Cache-Speicher. Der Anteil an Speicherzugriffen der aus dem Cache bedient werden kann, sei durch die **Trefferrate** h gegeben, wobei für die entsprechende Trefferwahrscheinlichkeit $0 \leq h \leq 1$ gilt. T_{HS} bezeichne die Zugriffszeit des Hauptspeichers und T_{CA} die Zugriffszeit auf den Cache. Gesucht ist die effektive Zugriffszeit T_{eff}.



Wir wollen zwei Fälle unterscheiden. Im ersten Fall nehmen wir an, dass der Zugriff auf HSP und Cache sequentiell erfolgt wie dies bei früheren Prozessoren der Fall war. Dabei wird zuerst auf den Cache zugegriffen, und erst nachdem festgestellt wurde, dass sich das gesuchte Datum nicht im Cache befindet, wird auf den HSP zugegriffen. Der Fehlzugriffsaufwand ergibt sich dann aus der Addition $T_{\mathrm{CA}}+T_{\mathrm{HS}}$. Für die mittlere effektive Zugriffszeit folgt somit:

$$T_{eff} = h \cdot T_{CA} + (1-h) \cdot (T_{CA} + T_{HS})$$

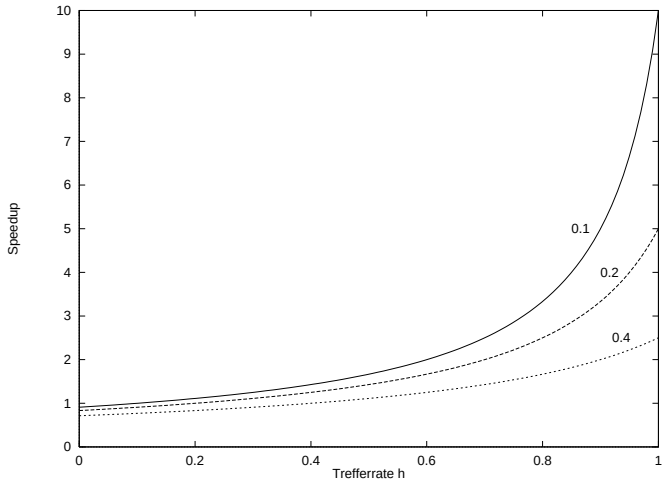
Der erste Ausdruck $h \cdot T_{CA}$ beschreibt den Einfluss der Cache-Treffer. Der zweite Ausdruck, $(1-h) \cdot (T_{CA} + T_{HS})$ berücksichtigt die restlichen Zugriffe, bei denen zuerst im Cache und danach im HSP gesucht wird. Eine einfache algebraische Umformung ergibt:

$$T_{eff} = h \cdot T_{CA} + T_{CA} + T_{HS} - h \cdot T_{CA} - h \cdot T_{HS} = T_{CA} + (1-h)\, T_{HS}$$

Erfolgt der Zugriff auf HSP und Cache parallel, d.h. es wird stets unabhängig davon, ob sich das Datum im Cache befindet vorsichtshalber auch auf den HSP zugegriffen, dann ergibt sich die effektive Zugriffszeit als:

$$T_{eff} = h \cdot T_{CA} + (1-h) \cdot T_{HS}$$

Da beide Gleichungen zahlreiche Einflussgrößen, wie zum Beispiel die Unterscheidung in Lese- und Schreibzugriffe, bei der Berechnung der effektiven Zugriffszeit vernachlässigen, entsteht eine einfache lineare Gleichung. Diese ist aber trotzdem geeignet, um den elementaren Zusammenhang zwischen der Trefferrate *h* und der tatsächlichen Beschleunigung des Systems darzustellen.



Die effektive Beschleunigung des Systems wird als **Speedup** *S* bezeichnet und ist definiert als das Verhältnis von bisheriger Zugriffszeit zur Zugriffszeit mit Cache:

$$S = \frac{T_{HS}}{T_{eff}}$$

Die folgende Abb. zeigt den Zusammenhang zwischen der Trefferrate und der Beschleunigung bei verschiedenen Verhältnissen $\frac{T_{CA}}{T_{HS}}$. Man beachte, dass auch ein Speedup kleiner 1 möglich ist, falls die Trefferrate sehr weit absinkt. In einem solchen Fall bewirkt die zusätzliche Zeit zur Verwaltung des Cache eine Verschlechterung im Vergleich zu einem System ohne Cache-Speicher. Besonders auffällig ist die überproportionale Beschleunigung bei hohen Trefferraten. Selbst wenn der Cache-Speicher 10-mal schneller ist als der Hauptspeicher ($\frac{T_{CA}}{T_{HS}} = 0,1$), müssen etwa 3/4 aller Zugriffe im Cache abgewickelt werden, damit eine Beschleunigung um den Faktor 3 erreicht wird. Offensichtlich ist eine merkliche Geschwindigkeitssteigerung erst ab einer Trefferrate von $0,8$ möglich. □

3.3.2 Funktionsweise des Caches

Bei einem Cache-Speicher versteht man unter einem **Blockrahmen** (*Block-Frame*) eine Anzahl von Speicherplätzen, dazu ein Adressetikett und Statusbits. Ein **Cache-Block** (*Cache-Block*, *Cache-Line*) ist die Speicherportion, die in einen Blockrahmen passt. Dies ist gleichzeitig auch die Speicherportion, die bei Bedarf auf einmal zwischen Cache- und Hauptspeicher übertragen wird. Im Vergleich mit der Speicherseitenverwaltung (Paging) entspricht ein Cache-Block einer Seite (Page), die aus einer Menge von in der Regel 2^n Speicherzellen besteht. Unter der **Blocklänge** (*block size*, *line size*), auch Blockgröße genannt, versteht man die Anzahl der Speicherplätze in einem Blockrahmen. Die Größe eines solchen Blockrahmens wird so ausgelegt, dass sie sich immer durch eine Zweierpotenz ausdrücken lässt:

$$\text{Blockgröße in Byte} = 2^{\#r}, \qquad 2^1 = 2,\ 2^2 = 4,\ 2^3 = 8, \text{ usw.} \tag{3.2}$$

Hierbei liegt $\#r$ üblicherweise im Bereich $2 \leq \#r \leq 6$, abhängig von der Breite des Datenbusses und der Organisation des Speichers. Typische Blocklängen sind 16 oder 32 Byte.

Da ein Blockrahmen im Cache zu verschiedenen Zeitpunkten verschiedene Blöcke aus dem Hauptspeicher repräsentieren kann, muss zusätzlich zur Dateninformation auch noch eine Adressinformation im Cache gespeichert werden. Diese Adressinformation, das **Adressetikett** (*address tag*, *cache tag*), das auch einfach **Tag** genannt wird, enthält einen Teil der Blockadresse, also einen Teil der Speicheradresse des ersten Speicherwortes der aktuell im Blockrahmen gespeicherten Speicherwörter. Es ermöglicht die Zuordnung eines Cache-Eintrags zu einer Adresse im Hauptspeicher. Die Größe dieses Adressfeldes hängt neben der Adressbreite im verwendeten Computersystem auch noch von der Cache-Organisation ab. Zusätzliche Statusbits im Etikett geben an, ob der Cache-Block „gültig“ bzw. „ungültig“ ist und ob ein Teil des Cache-Blocks geändert wurde. In diesem Fall wird das sog. *Dirty-Bit* gesetzt.

Die Adresse einer Hauptspeicherzelle gliedert sich bei einer Cache-Speicherzelle in einen **Tag-Teil**, einen **Indexteil** und eine **Wortadresse**. Tag- und Indexteil der Adresse ergeben die Blockadresse, während die Wortadresse das Speicherwort innerhalb des Blocks identifiziert.

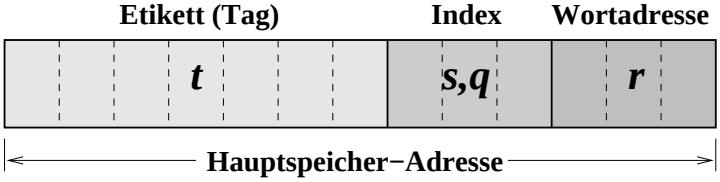


Abbildung 3.9: Adressaufteilung bei direkter Zuordnung

Der Indexteil wird durch den Cache-Speicher hardwaremäßig festgelegt. Ein Cache mit beispielsweise $2^8 = 256$ Speicherrahmen besitzt hexadezimale Indexadressen von 00 bis FF. In diesen 256 Speicher- bzw. Blockrahmen können entsprechend der Cache-Organisation, auf die wir später kommen, Datenblöcke abgespeichert werden. Zu jedem Datenblock gehört dabei ein Tag (Etikett), das den in einem Blockrahmen unter einem bestimmten Index abgelegten HSP-Block eindeutig identifiziert. Ein Cache-Speicher besteht also aus dem Datenspeicher, in dem die Datenblöcke abgelegt werden und einem Adressspeicher, in dem die dazugehörigen Tags abgelegt werden. Der Index dient zum Auffinden eines bestimmten Blockrahmens innerhalb des Caches. Die Wortadresse bezeichnet das Speicherwort innerhalb eines Cache-Blocks, ist also von der Blocklänge und der Art der Speicheradressierbarkeit (Byte- oder Wort-adressiert) abhängig. Im Falle eines Byte-adressierbaren Speichers und einer Blockgröße von 16 Bytes benötigt die Wortadresse vier Bits.

Die Behandlung von Lesezugriffen

Auf den Cache-Speicher kann natürlich sowohl lesend als auch schreibend zugegriffen werden. Beide Zugriffe werden durch einen Zugriff des Prozessors auf den Hauptspeicher (HSP) ausgelöst. Bei jedem Lesezugriff des Prozessors auf eine Speicherzelle im HSP wird gleichzeitig auch im Cache nachgeschaut, ob diese Speicherzelle und damit das gesuchte Wort (Byte) dort bereits abgelegt wurde.

Bei einem Treffer (**Read-Hit**) wird das gesuchte Wort aus dem Cache entnommen und bleibt natürlich für weitere Zugriffe dort erhalten.

Bei einem Fehlzugriff (**Read-Miss**) wird das gesuchte Datenwort aus dem Hauptspeicher in den Prozessor geladen. Daneben wird die vom Prozessor benötigte HSP-Speicherzelle in die Cache-Speicherzelle mit dem entsprechenden Index übertragen. Dazu wird sowohl das gesuchte Wort bzw. der Block, in dem sich das gesuchte Wort befindet, in den Datenspeicher des Caches geladen, als auch die entsprechenden Tags im Adressspeicher aktualisiert. In der Regel wird dabei ein älterer länger nicht mehr benötigter Block, der unter diesem Index abgelegt war, verdrängt.

Die Behandlung von Schreibzugriffen

Bei jedem Schreibzugriff des Prozessors auf eine Speicherzelle im HSP wird gleichzeitig auch im Cache nachgeschaut, ob diese Speicherzelle dort bereits abgelegt wurde. Bei einem Treffer (**Write-Hit**) kann die neu zu beschreibende Speicherzelle im Cache aktualisiert werden. Dazu müsste aber nur der Datenspeicher aktualisiert werden, das entsprechende Tag bleibt unverändert. Wenn aber die betreffende Speicherzelle nur im Cache aktualisiert wird, dann ist die entsprechende Speicherzelle im HSP veraltet und unbrauchbar. An dieser Stelle entsteht also ein Problem, die Inkonsistenz zwischen Cache und HSP.

Bei einem Fehlzugriff (**Write-Miss**) müsste die neu zu beschreibende Speicherzelle im Cache sozusagen erst eingerichtet werden. Dazu müsste sowohl Datenspeicher als auch Tag unter dem entsprechenden Index im Cache aktualisiert werden. Zusätzlich ergibt sich das Problem der Inkonsistenz. Im einfachsten Fall wird bei einem Write-Miss nur der HSP aktualisiert und damit die Inkonsistenz umgangen. Um diesem Problem der Inkonsistenz generell zu begegnen, gibt es jedoch verschiedene Schreibstrategien, die wir im nächsten Abschnitt betrachten wollen.

3.3.3 Die Inkonsistenz bei Schreibzugriffen

Jede mehrschichtige Speicherarchitektur beinhaltet ein Konsistenzproblem, da bei Schreibzugriffen auf einzelne Ebenen eine Inkonsistenz zu anderen, bisher noch nicht aktualisierten Schichten entsteht. Der Grund dafür ist, dass beim Schreiben einfacher weise stets immer nur die aktuelle Ebene überschrieben wird, die anderen nicht. Wird beispielsweise ein Datum im Cache durch einen Schreibbefehl aktualisiert, so muss dieses in die nächsthöhere Ebene, hier den HSP, zurückgeschrieben werden, falls die betroffene Speicherzelle im Cache erneut aktualisiert werden soll. Im Folgenden wollen wir daher verschiedene Verfahren zur Behandlung von Schreibzugriffen behandeln.

Durchschreibestrategie

Bei der **Durchschreibestrategie** (*Write-Through-Verfahren*, *Store-Through-Cache-Policy*) wird ein Speicherwort vom Prozessor immer gleichzeitig in den Cache- und in den Hauptspeicher geschrieben. Der Vorteil ist eine garantierte Konsistenz der Inhalte von Cache- und Hauptspeicher, da der Hauptspeicher immer die zuletzt geschriebene, also gültige Kopie enthält. Das ist besonders dann wichtig, wenn mehrere Verarbeitungselemente (beispielsweise mehrere Prozessoren) auf ein Speicherwort zugreifen wollen, das sich in einem Cache-Speicher befindet.

Der Nachteil ist, dass der sehr schnelle Prozessor bei Schreibzugriffen mit dem langsamen Hauptspeicher synchronisiert werden muss und somit erheblich an Verarbeitungsgeschwindigkeit verliert. Um diesen Nachteil zu mildern, wird in Verbindung mit einer Durchschreibestrategie ein kleiner **Schreibpuffer** (*write buffer*) verwendet, der das zu schreibenden Speicherwort temporär aufnimmt und sukzessive in den Hauptspeicher überträgt, während der Prozessor parallel dazu mit weiteren Operationen fortfährt. Diese Technik wird als **gepufferte Durchschreibestrategie** (*buffered write-through*) bezeichnet. Selbst in diesem Fall wird aber der Bus zwischen Prozessor und Speicher belegt, was zum Engpass führen kann.

Rückschreibestrategie

Um auch bei Schreibzugriffen die höhere Geschwindigkeit des Cache-Speichers auszunutzen, wird bei der **Rückschreibestrategie** (*Write-Back-Verfahren*, *Copy-Back-Cache-Policy*, *store-in-cache*) ein Speicherwort nur in den Cache-Speicher geschrieben, ohne den Hauptspeicher zu aktualisieren. Alle im Cache befindlichen Daten können beliebig oft mit hoher Geschwindigkeit gelesen und verändert werden, wobei jedoch die Veränderungen dann zu einem späteren Zeitpunkt vom Cache zum Hauptspeicher transportiert werden müssen. Ein Geschwindigkeitsvorteil gegenüber dem Write-Through-Verfahren entsteht also dann, wenn die gleiche Adresse mehrfach geändert wird *bevor* der Transport zum Hauptspeicher erfolgt.

Damit die Cache-Verwaltung jederzeit feststellen kann, ob ein Block verändert wurde, bekommt das Etikett (Tag) zusätzlich zur Adressinformation noch ein weiteres Bit, das sog. **Dirty-Bit**. Falls dieses Bit gesetzt ist, wurde mindestens eine Stelle innerhalb des Blocks modifiziert. Dieses *Dirty-Bit* muss bei jedem Schreibzugriff auf den Cache gesetzt werden. Es wird zurückgesetzt, wenn der inkonsistente Block in den HSP zurückgeschrieben wird. Um bei einem Cache-Miss durch Nachladen und Verdrängen keine Änderungen im Cache zu überschreiben, muss vorher immer das Dirty-Bit des zu verdrängenden Blocks geprüft werden.

Die Rückschreib-Strategie, die zwar einen höheren Verwaltungsaufwand bei einem Lesezugriff verlangt, benötigt beim Schreibzugriff durch den Prozessor keine Synchronisation mit dem Hauptspeicher und ist damit schneller. Die Strategie ist komplexer – aber vorteilhaft. Sie wird bei heutigen Rechnern vorwiegend angewandt. Da jedoch der Datendurchsatz auch sehr stark vom Programmverhalten abhängt, ist die Strategie bei einigen Prozessoren variabel ausgelegt. So kann zum Beispiel der interne Daten-Cache des Intel Pentium wahlweise für Write-Back oder für Write-Through konfiguriert werden.

Fehlzugriffe beim Schreiben

Bei einem *lesenden Fehlzugriff*, einem **Read-Miss**, wird die im Cache nicht vorhandene gesuchte Speicherzelle immer mit dem entsprechenden Cache-Block aus dem HSP in den Cache geladen, der Cache wird aktualisiert.
Bei einem *schreibenden Fehlzugriff*, einem **Write-Miss**, hingegen gibt es zwei unterschiedliche Strategien. Bei der Strategie **Write-Allocation** wird ein **Write-Miss** genauso behandelt wie ein **Read-Miss**, die gesuchte Speicherzelle wird mit dem entsprechenden Cache-Block aus dem HSP in den Cache geladen. Dann wird die Speicherzelle neu beschrieben und beim *Write-Back-Verfahren* das die Inkonsistenz anzeigende *Dirty-Bit* gesetzt. Diese sehr aufwändige Strategie ist in vielen Fällen, insbesondere bei Mehrprozessorsystemen, wegen der damit verbundenen großen Komplexität unwirtschaftlich.

Deswegen gibt es daneben die sehr viele einfachere Strategie **Write-Around**, bei der die zu schreibende Speicherzelle im Cache bei einem **Write-Miss** garnicht erst gesucht und auch nicht eingetragen wird. Wegen der zunehmenden Möglichkeiten an Zugriffen von verschiedenen Stellen aus auf den HSP bzw. den Cache (zunehmende Zahl an Prozessoren, die zusammenarbeiten, DMA usw.) geht der Trend mehr zu dieser Strategie hin. Neben der Reduzierung der Komplexität spielt auch die Überlegung eine Rolle das man nach einem schreibenden Zugriff auf eine Speicherzelle mit großer Wahrscheinlichkeit wieder schreiben wird bevor diese gelesen wurde und sich dadurch kein Vorteil aus der Aktualisierung des Caches ziehen lässt.

Der Einfachheit halber gehen wir in diesem Kurs in allen Erläuterungen, Beispielen und den Aufgaben von der Strategie **Write-Around** aus ohne dies jeweils besonders zu erwähnen.

Aufgabe 3.6 *Schreibvarianten*
Warum kann bei einem Pentium-Prozessor nur der Daten-Cache frei konfiguriert werden? ◇

3.4 Organisation von Cache-Speichern

Die gesamte Menge der Block- bzw. Speicherrahmen eines Cache-Speichers wird in sog. **Sätze** (*sets*) unterteilt. Die Anzahl der Blockrahmen n in einem Satz wird als **Assoziativität** (*associativity, degree of associativity, set size*) bezeichnet. Jeder Datenblock aus dem HSP kann im Cache nur in einem bestimmten Satz, aber innerhalb des Satzes in einem beliebigen Blockrahmen gespeichert werden. Die Gesamtzahl c der Blockrahmen in einem Cache-Speicher entspricht immer dem Produkt aus der Anzahl s der Sätze und der Assoziativität n, also $c = s \cdot n$.

Beim Entwurf einer (Cache-)Speicherhierarchie müssen die folgenden vier wesentlichen Entscheidungen getroffen werden:

1. Die erste Entscheidung betrifft die Platzierung eines HSP-Blocks innerhalb des Caches: Hier besteht die Wahl zwischen einer *direkt abgebildeten*, *vollassoziativen* oder *satzassoziativen* Organisation.
2. Die zweite Entscheidung betrifft die Adressaufteilung der HSP-Adresse in Index, Tag und Wort-Bits. Diese Entscheidung steht mit der Größe des Caches und der Blöckgröße in Zusammenhang.
3. Die dritte Entscheidung betrifft die Verdrängungsstrategie, die festlegt welcher Block nach einem Fehlzugriff aus dem Cache verdrängt wird.
4. Die vierte Entscheidung betrifft die Organisation der Schreibzugriffe, wobei im wesentlichen die Durchschreib- und die Rückschreibstrategie zur Auswahl stehen.

Wir werden im Folgenden alle vier Entscheidungen noch ausführlicher betrachten.

Ein Cache-Speicher mit c Blockrahmen heißt:

- **direkt abgebildet** (*Direct-Mapped*), falls jeder Satz nur einen einzigen Blockrahmen enthält ($n=1$, $s=c$),
- **vollassoziativ** (*Fully-Associative*), falls er nur aus einem einzigen Satz besteht ($s=1$, $n=c$) und
- ***n*-fach satzassoziativ** (*n-Way set-Associative*), falls allgemein $s = c/n$ gilt.

Abbildung 3.10: Zuordnung von HSP-Blöcken zu Blockrahmen im Cache

Abb. 3.10 demonstriert diese Unterscheidung am Beispiel eines Cache-Speichers mit einer Kapazität von acht Cache-Blöcken ($c=8$). Unter der Annahme einer Abbildungsfunktion „Blockadresse modulo Anzahl Sätze“ kann beispielsweise ein Block mit der Adresse 12 in einem der grau markierten Blockrahmen gespeichert werden. In Abb. 3.10 sind an den linken Seiten jeweils die Satznummern angegeben. Im Falle des vollassoziativen Cache-Speichers gibt es nur einen Satz und der HSP-Block kann an einer beliebigen Stelle im Cache abgelegt werden. Im Falle der zweifach satzassoziativen Verwaltung gibt es vier Sätze mit je zwei Blockrahmen. Der HSP-Block mit Adresse 12 muss also in einem der beiden Blockrahmen des Satzes 0 ($12\,Mod\,4 = 0$) abgelegt werden. Im Falle der direkt

abgebildeten Organisation gibt es 8 Sätze mit je einem Blockrahmen. Der Block muss also im Satz 4 ($12 \mathit{Mod}\ 8 = 4$) gespeichert werden.

Wie wir im vorherigen Abschnitt schon gesehen haben, wird eine HSP-Adresse beim Cache in eine Wortadresse der Länge *r*, einen Indexteil der Länge *s* und einen Tag der Länge *t* aufgeteilt. Der Indexteil wird benötigt, um in einem direkt abgebildeten oder *n*-fach satzassoziativen Cache-Speicher (s.u.) den Satz ausfindig zu machen. Die Länge des Index ist also von der Anzahl der Sätze im Cache-Speicher abhängig. Unter Annahme von 128 Cache-Sätzen benötigt der Indexteil sieben Bits der Speicherwortadresse. Der Rest der Adresse wird zum Tag-Teil, d.h. die Tag-Längen im Cache-Speicher hängen entscheidend von der Blocklänge, der Assoziativität und der Größe des Cache-Speichers ab. Bei gleich bleibender Cache-Größe verringert eine höhere Assoziativität die Indexlänge und vergrößert die Tag-Länge.

3.4.1 Cache mit direkter Zuordnung (Direct-Mapped-Cache)

Die einfachste Organisationsform stellt der Cache-Speicher mit **direkter Zuordnung** (*Direct-Mapped-Cache*) dar. Beim direkt abgebildeten Cache gibt es so viele Sätze wie Blockrahmen, es besteht eine Eins-zu-Eins-Abbildung von HSP-Blöcken zu Blockrahmen im Cache. Der Inhalt einer Adresse des HSP kann nur an einer einzigen Position im Cache-Speicher liegen, aus dem Indexteil der HSP-Adresse ergibt sich also direkt der Blockrahmen im Cache, in dem dieser HSP-Block abgelegt sein kann. Dieser Zusammenhang ist in Abb. 3.11 an einem einfachen Beispiel erläutert. Die gestrichelten Linien deuten den Zusammenhang zwischen einer Adresse im Hauptspeicher und der möglichen Position im Cache-Speicher an. In diesem Beispiel soll jeder Block 4 Byte enthalten, also gilt $\#r = 2$. Jede HSP-Adresse sei 8 Bit breit, der Hauptspeicher enthalte $2^8 = 256$ Byte. Der Cache-Speicher enthalte 4 Blockrahmen, insgesamt also 16 Byte. Die Anzahl der Blöcke im Cache-Speicher ist $2^{\#q} = 4$ gegeben, wobei $\#q = 2$.

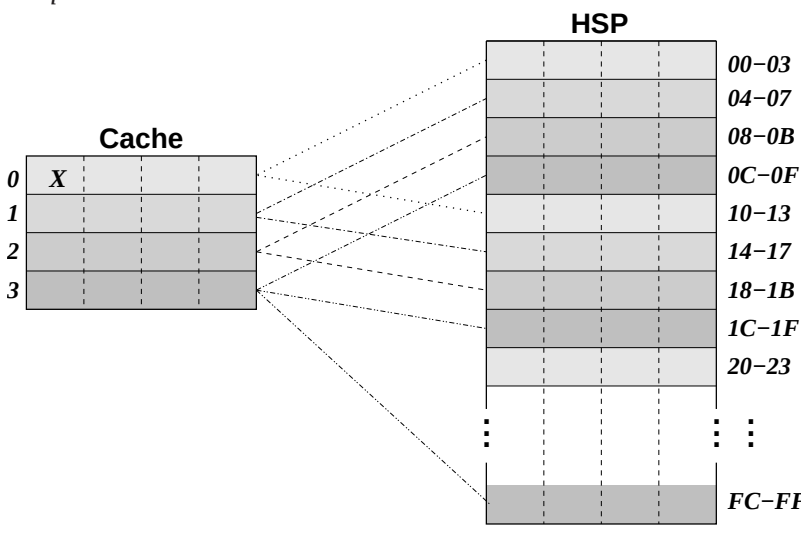
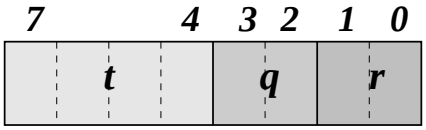


Abbildung 3.11: Cache-Speicher mit direkter Zuordnung

Die Größe des Datenspeichers eines *Direct-Mapped-Caches* in Bytes ist gegeben durch:

$$C = \text{Cachegröße} = \text{Anzahl Blöcke} \times \text{Blockgröße} = 2^{\#q} \cdot 2^{\#r} = 2^{\#q+\#r} \tag{3.3}$$

Es stellt sich nun die Frage, welcher Teil der Hauptspeicheradresse als Tag im Cache gespeichert werden muss, um eine eindeutige Zuordnung zu ermöglichen. Bei diesem einfachen Beispiel kann man leicht erkennen, dass an der mit „*X*“ markierten Stelle im Cache-Speicher nur jeweils die Inhalte der Hauptspeicheradressen „00, 10, 20, …“ abgelegt werden können. Man sieht weiterhin, dass alle diese Adressen auf den niederwertigen Bitpositionen die 0 enthalten. Da diese Stellen immer gleich sind, brauchen sie nicht im Tag gespeichert werden. Für jeden Block brauchen also nur 4 Bit, nämlich die oberen 4 Bit der Adresse, gespeichert werden. Diese Adressaufteilung ist in Abb. 3.12 dargestellt.



Tag=höherwertigen 4 Bit der Adresse

Abbildung 3.12: Adressaufteilung bei direkter Zuordnung

Falls der Prozessor nun beispielsweise den Inhalt der Adresse $16_{16} = 00010110_2$ lesen möchte, so zeigen die Bitpositionen 2 und 3 dieser Adresse (Wert 01), dass sich der gesuchte Wert nur im Cache-Rahmen Nr. 1 befinden kann (in Abb. 3.11 durch ein „*X*“ markiert). Die oberen 4 Bit der Adresse werden nun mit dem für Block 1 gespeicherten Tag verglichen. Bei Übereinstimmung kann der Wert direkt aus dem Cache gelesen werden, sonst muss der Block zuerst aus dem Hauptspeicher geladen werden.

Die Anzahl der Bits in einem einzelnen Etikett wird durch die Variable #*t* dargestellt. Für einen *Direct-Mapped-Cache* gilt:

$$\#t = \text{Adressbreite} - \#q - \#r \tag{3.4}$$

Die Gesamtgröße des erforderlichen Tag-Speichers kann durch den Ausdruck $2^{\#q} \cdot \#t$ berechnet werden. Im vorliegenden Fall wären demnach $2^2 \cdot 4 = 16$ Bit erforderlich.

An diesem einfachen Beispiel des Caches mit Bockgröße 4 und 4 Sätzen bzw. 4 Blöcken wird auch ein Hauptvorteil des *Direct-Mapped-Cache* deutlich: Es muss nur ein einziges Tag (Etikett) mit einem Teil der Adresse verglichen werden, um zu entscheiden, ob sich der gesuchte Wert im Cache-Speicher befindet. Eine solche Funktion kann mit relativ geringem Hardwareaufwand realisiert werden. Allerdings muss diese einfache Struktur durch eine geringe Flexibilität bezahlt werden, da jede Hauptspeicheradresse nur an eine einzige Stelle in den Cache kopiert werden kann. Falls zwei häufig benutzte Adressen auf den gleichen Block im Cache abgebildet werden, sinkt die Trefferrate deutlich, da sich die Einträge immer gegenseitig verdrängen.

Hinweis: Die Beispiele zur Erläuterung der Adressaufteilung beziehen sich auf eine Byte-adressierte Architektur. Bei Wort- oder Langwort-adressierten Maschinen verschiebt sich die Adressaufteilung entsprechend.

In der Adressaufteilung der Abb. 3.9 ist angegeben, dass man die *höherwertigen* Bitstellen der Adresse als Tag (Etikett) und die *niederwertigen* Bitstellen als Index verwendet. Logisch wäre doch die umgekehrte Aufteilung, da man bei der Suche eines Speicherinhalts unter dem Index nach schaut, ob dort das gesuchte Tag vorhanden ist. Der Grund für die umgekehrte Aufteilung ist, dass man damit eine bessere Ausnutzung der räumlichen Lokalität erhält, da im HSP benachbarte Speicherzellen im Cache gerade nicht benachbart sind und umgekehrt. Dies führt zu weniger häufigeren Verdrängungen.

Beispiel 3.9 *Cache-Speicher mit direkter Zuordnung*
Folgende Systemdaten seien vorgegeben: 16 MB Hauptspeicher mit 24 Bit Adressbus,
Cachegröße 32 KB, Blockgröße 16 Byte.
Gesucht sind die Adressaufteilung, die Zahl der Blockrahmen, und die Größe des Tag-Speichers.

Bei einem Byte-adressierten Speicher ergibt sich die Zahl #*r* der Wort-Bits unmittelbar aus der Blockgröße:

$$16\,\text{Byte} = 2^{\#r} \quad \Rightarrow \quad \#r = 4$$

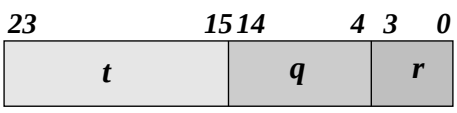
Aus der Cachegröße ergibt sich dann der Anzahl der Index-Bits #*q*:

$$32\,\text{KB} = 2^{15} = 2^{\#q+\#r} \quad \Rightarrow \quad \#q = 11$$

Die Anzahl der Tag-Bits #*t* ergibt sich aus dem Rest der Adresse:

$$\#t = \text{Adressbreite} - \#q - \#r = 9$$

Die Adressaufteilung ergibt sich somit zu:



Die Anzahl der Sätze bzw. Blöcke im Cache ergibt sich aus der Zahl #*q* der Index-Bits:
$2^{\#q} = 2^{11} = 2048$ oder aus der Cachegröße: $32 \cdot 2^{10}\,\text{Byte} / 16\,\text{Byte} = 2 \cdot 2^{10} = 2048$.
Die Größe des Tag-Speichers ergibt sich aus der Zahl #*q* der Index-Bits und der Zahl #*t* der Tag-Bits:
$2^{\#q} \cdot \#t = 18432\,\text{Bit} = 2304\,\text{Byte}$ □

Aufgabe 3.7 *Verhalten des Cache-Speichers mit direkter Zuordnung*
Welche Veränderungen treten an den Inhalten von Tag- und Datenspeicher eines *Direct-Mapped*-Cache sowie im Hauptspeicher des Rechners bei den folgenden Ereignissen auf:

1) *Write-Hit* 2) *Write-Miss* 3) *Read-Hit* 4) *Read-Miss*

Erklären Sie das Verhalten jeweils für die Strategien A) *Write-Through* und B) *Write-Back*. ◇

Aufgabe 3.8 *Verbesserung der Zugriffszeit*
Angenommen Sie würden die Cache-Bausteine eines Prozessors durch neuere mit halber Zugriffszeit ersetzen. Um welchen Faktor kann sich die vom Prozessor beobachtete Speicherzugriffszeit maximal verbessern? Unter welchen Bedingungen tritt dieser Optimalfall ein? ◇

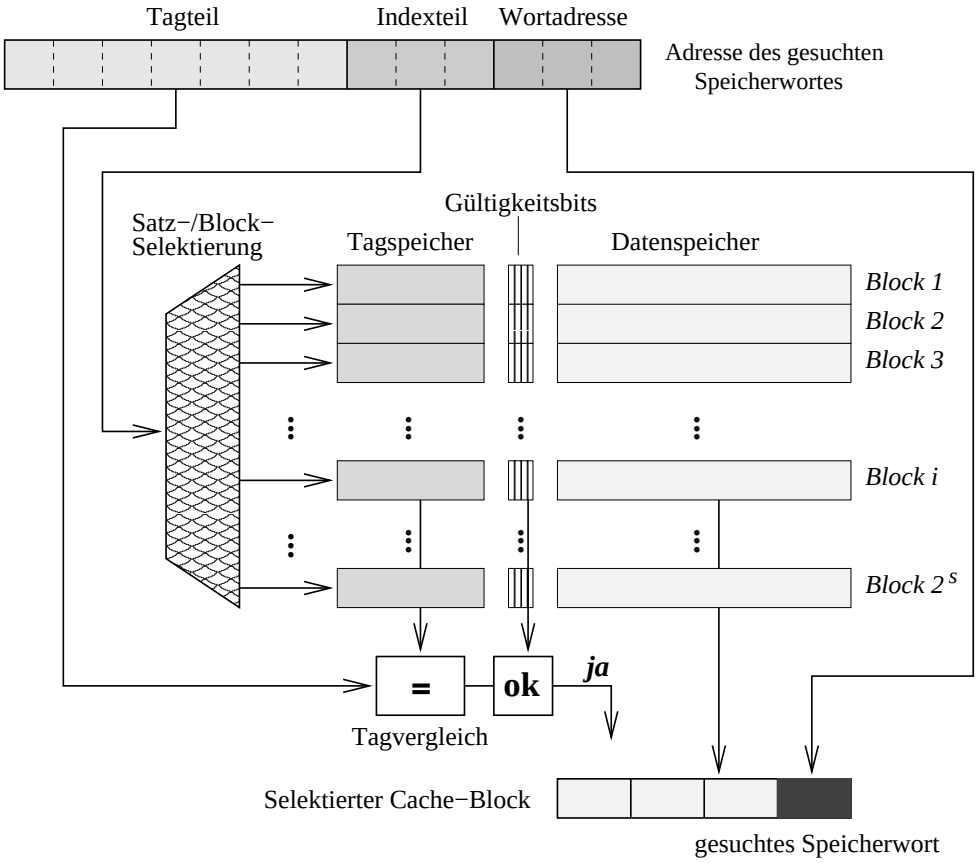


Abbildung 3.13: Realisierung einer direkt abgebildeten Cache-Speicherverwaltung

Abb. 3.13 zeigt die prinzipielle Realisierung einer direkt abgebildeten Cache-Speicherverwaltung. Jeder Blockrahmen besteht im Direct-Mapped-Cache aus dem Tag (Adressteil des enthaltenen Blocks), einigen Statusbits und dem gespeicherten Cache-Block. Der Indexteil *x* der angelegten Adresse wird einer Abbildungsfunktion f unterworfen und adressiert mit Hilfe des Satz-/Block-Decoders einen Satz, der hier gleichzeitig auch einem Block entspricht. Die meist verwendete Abbildungsfunktion ist $f(x) = x \,Mod\, s$, wobei *s* eine Zweierpotenz ist. Ein Vergleicher überprüft die Gleichheit von Tag-Teil der angelegten Adresse und Tag (Adressteil des Blocks, der im selektierten Blockrahmen gespeichert ist). Bei Gleichheit wird noch der Status überprüft. Ist auch dieser in Ordnung, so haben wir einen Cache-Treffer und es wird anhand der Wortadresse das gesuchte Speicherwort aus dem Block ausgewählt.

Zusammenfassend weist der direkt abgebildete Cache-Speicher folgende Eigenschaften auf:

- Die Hardwarerealisierung ist einfach. Es wird ein Tag-Feld pro Block und nur insgesamt ein Vergleicher benötigt.
- Der Zugriff erfolgt schnell, weil auf das Tag-Feld parallel mit dem Block zugegriffen werden kann.
- Auch wenn an anderer Stelle im Cache noch Platz ist, erfolgt wegen der direkten Zuordnung eine Ersetzung, so dass die Speicherkapazität oft schlecht ausgenutzt wird.
- Das Ersetzungsverfahren ist einfacher als bei *n*-fach satzassoziativen Cache-Speichern, da bei einem direkt abgebildeten Cache-Speicher jeder Satz nur *einen* Cache-Block enthält.

Bei einem abwechselnden Zugriff auf Cache-Blöcke, deren Adressen den gleichen Indexteil haben, erfolgt laufendes Überschreiben des gerade geladenen Blocks. Es kommt zum sog. **Flattern** (*Thrashing*). Davon ist der direkt abgebildete Cache-Speicher besonders häufig betroffen. Dieses Flattern wird bei einem *n*-fach satzassoziativen Cache-Speicher um so mehr vermieden, je größer die Assoziativität *n* ist.

Beispiel 3.10 *Zugriffe beim Cache-Speicher mit direkter Zuordnung*
Gegeben sei ein 8-Bit-Prozessor mit einem 16 Bit breiten Adressbus und einem *Direct-Mapped* Cache. Der Zugriff zum Speicher erfolge nach der *Write-Through*-Strategie durch die folgenden Assemblerbefehle (0000\leq$ ADR $\leq$$FFFF):

LDA ADR	Lade den Akkumulator mit dem Inhalt der Speicherstelle ADR
STA ADR	Speichere den Inhalt des Akkumulators unter der Adresse ADR

Der Hauptspeicher sei Byte-adressiert und die Blockgröße sei der Einfachheit genau ein Byte. Die folgende Tabelle zeige einen unzusammenhängenden und unsortierten Ausschnitt aus der aktuellen Belegung des Arbeitsspeichers (HSP):

Adresse	Datum	Adresse	Datum	Adresse	Datum	Adresse	Datum
FFFF	*03*	8F7F	*D2*	E4D0	*CD*	8729	*BC*
D000	*35*	7FD0	*3B*	CF7F	*D0*	6330	*5D*
CD84	*00*	21FE	*B1*	A530	*A7*	0430	*7F*
A0B7	*D3*	01FF	*47*	9800	*7F*	0000	*36*

Die folgende Abbildung zeige einen wiederum unzusammenhängenden Ausschnitt aus der Belegung des Caches mit 256 Einträgen zu Beginn eines Beobachtungszeitraumes, wobei die links notierte HEX-Zahl den Index des jeweiligen Cache-Rahmens (1 Byte) angibt.

	Tag	**Datum**
FF	**FF**	***03***
D0	**E4**	***CD***
B7	**A0**	***D3***
84	**CD**	***00***
7F	**8F**	***D2***
30	**A5**	***A7***
29	**87**	***BC***
00	**D0**	***35***

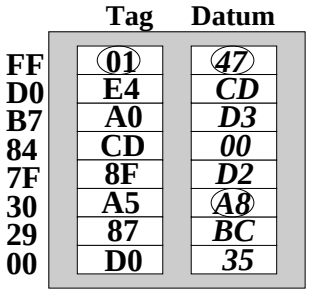
1. LDA $CD84
2. STA $A530 ACCU = $*A*8
3. LDA $01FF
4. LDA $01FF
5. STA $FFFF ACCU = $04

Man gebe für die in der Tabelle angegebenen Speicherzugriffe an, ob dabei ein *Hit* oder ein *Miss* vorliegt. Dabei soll die aktuelle Hauptspeicheradresse (HSP), der Inhalt des *Akkus* sowie der *Index*, das *Tag* (Etikett) und das *Datum* des aktuellen Cache-Blockes nach Ausführung jedes Befehls angegeben werden. ACCU=$A8 bedeute, dass der Inhalt des Registers A (Akkumulator) unabhängig von den vorherigen Anweisungen aktuell $A8 betrage.

Wie oft wird bei der Ausführung dieser Anweisungen der Eintrag mit dem Index **FF** modifiziert und wie oft wird dabei das *Tag* verändert? Wie oft wird insgesamt im Cache ein *Datum* verändert?

						Cache		
Befehl	Hit	Miss	Akku	Datum	HSP-Adr.	Index	Tag	Datum
1. LDA $CD84	R		00	00	CD84	84	CD	00
2. STA $A530	W		A8	A8	A530	30	A5	A8
3. LDA $01FF		R	47	47	01FF	FF	01	47
4. LDA $01FF	R		47	47	01FF	FF	01	47
5. STA $FFFF		W	04	04	FFFF	FF	01	47

Wir beginnen mit der Bearbeitung des ersten Befehls LDA $CD84. LDA $CD84 ist ein Lesebefehl, es soll der Inhalt der Adresse CD84 in den Akkumulator geladen werden. Die Adresse CD84 besteht aus dem Index 84 und dem Tag CD. Daher wird im Cache nachgesehen, ob dort für den Index 84 das Tag CD eingetragen ist. Unter dem Index 84 findet der Prozessor das Tag CD, es handelt sich also um ein Read-Hit. Der Prozessor liest das Datum 00 aus dem Cache, der Cache selbst bleibt unverändert.
Beim zweiten Befehl STA $A530 handelt es sich um einen Schreibbefehl auf die HSP-Zelle mit der Adresse A530, der Index dieser Adresse ist 30, das Tag ist A5. Unter dem Index 30 findet der Prozessor das Tag A5, es handelt sich also um ein Write-Hit. Der Prozessor schreibt das Datum A8 (den Akkuinhalt) in die entsprechende Zelle des Datenspeichers im Cache und aufgrund der Schreibstrategie Write-Through gleichzeitig auch in den HSP.
Beim dritten Befehl LDA $01FF handelt es sich um einen Lesebefehl auf die HSP-Zelle mit der Adresse 01FF, der Index dieser Adresse ist FF, das Tag ist 01. Unter dem Index FF ist aber das Tag FF eingelagert, so dass es sich bei diesem Zugriff um ein Read-Miss handelt. Bei einem Read-Miss wird das entsprechende Datum aus dem HSP gelesen und die entsprechende Cache-Zelle wird aktualisiert. Dazu wird unter dem Index FF das gerade benötigte Tag 01 und das dazugehörige Datum 47 eingetragen, der alte Inhalt wird überschrieben.
Beim vierten Befehl LDA $01FF handelt es sich wieder um einen Lesebefehl auf die HSP-Zelle mit der Adresse 01FF. Hierbei handelt es sich nun natürlich um ein Read-Hit. Der Prozessor liest das Datum 47 aus dem Cache, der Cache selbst bleibt unverändert.
Schließlich handelt es sich beim letzten Befehl STA $FFFF um ein Write-Miss, da zu diesem Zeitpunkt unter dem Index FF das Tag 01 eingelagert ist. Der Prozessor schreibt das Datum in den HSP, der Cache bleibt unverändert.

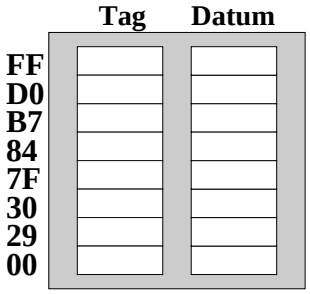


Das nebenstehende Bild zeigt den Cache-Inhalt nach Ausführung der fünf Anweisungen. Der Cache-Eintrag mit dem Index FF wird einmal modifiziert. Mit der 3. Anweisung wird der *Tag* zu 01 und das *Datum* zu 47. Das Tag wird also auch einmal modifiziert. Bei der 5. Anweisung handelt es sich um einen *Write-Miss*, und dabei werden die Daten nur in den HSP geschrieben, der Cache wird aber nicht aktualisiert. Insgesamt wird zweimal ein *Datum* im Cache verändert. □

Aufgabe 3.9 *Zugriffe beim Cache-Speicher mit direkter Zuordnung*

						Cache		
Befehl	Akku	Hit	Miss	Akku	HSP-Adr.	Index	Tag	Datum
6. LDA $8F7F								
7. LDA $FFFF								
8. STA $FFFF	ACCU=$05							
9. LDA $FFFF								
10. LDA $0000								
11. STA $A0B7	ACCU=$D4							

Gegeben sei der 8-Bit-Prozessor mit 16 Bit breiten Adressbus und *Direct-Mapped*-Cache aus Beispiel 3.10. Es gelten die nach der fünften ausgeführten Anweisung entstandenen HSP- und Cache-Inhalte. Nach diesen fünf Anweisungen sollen die folgenden sechs Anweisungen ausgeführt werden.



Geben Sie für die in der Tabelle angegebenen Speicherzugriffe die geforderten Daten an. Wie oft wurde bei der Ausführung dieser elf Anweisungen der Eintrag mit dem Index **FF** modifiziert und wie oft wurde dabei das *Tag* verändert? Wie oft wurde insgesamt im Cache ein *Datum* verändert? Geben Sie den Cache-Inhalt nach diesen elf Anweisungen an.

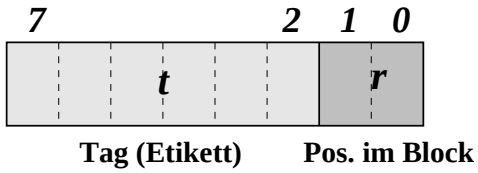
◇

3.4.2 Cache mit voller Zuordnungsfreiheit

Bei einer Organisation mit **voller Zuordnungsfreiheit** (*Fully-Associative-Cache*) kann ein Datenblock aus dem Hauptspeicher in einen *beliebigen* Blockrahmen im Cache-Speicher kopiert werden. Der vollassoziative Cache besitzt keinen Index, wegen der völligen Wahlfreiheit wird die Adresse nur in das Tag und die Wort-Bits aufgeteilt.

Durch die volle Zuordnungsfreiheit wird zwar ein Maximum an Flexibilität erreicht, aber der erforderliche Hardwareaufwand steigt gleichzeitig enorm an. Bei jedem Speicherzugriff des Prozessors muss die Cache-Verwaltung *alle* gespeicherten Tags (Etiketten) mit der gerade gesuchten Adresse vergleichen, um zu entscheiden, ob der gesuchte HSP-Block sich im Cache befindet. Um den Geschwindigkeitsvorteil des Cache-Speichers auszunutzen, muss diese Suche parallel erfolgen, d.h. alle Tags werden *gleichzeitig* mit der gesuchten Adresse verglichen. Diese Architektur benötigt also 2^q Vergleicher (einen für jeden Blockrahmen). Im Vergleich dazu ist der Aufwand für einen Direct-Mapped-Cache verschwindend gering, da hier nur ein einziger Vergleicher erforderlich ist.

Zur Darstellung der Adressaufteilung betrachten wir wieder das einfache Beispiel aus dem vorherigen Abschnitt 3.4.1 mit einer 8 Bit breiten Adresse und einem 16 Byte großen Cache mit 4 Blockrahmen von je 4 Byte.



Tag =höherwertigen 6 Bit der Adresse

Abbildung 3.14: Adressaufteilung bei voller Zuordnungsfreiheit

Im Vergleich zu der Aufteilung für einen Direct-Mapped-Cache (siehe Abb. 3.12) fällt sofort das größere Tag (Etikett) für den Fully-Associative-Cache auf. Dies resultiert aus der Tatsache, dass nun die Position im Cache keinen Hinweis mehr auf die Adresse geben kann. Dadurch ergibt sich für die Größe des Tags:

$$\#t = \text{Adressbreite} - \#r \tag{3.5}$$

Für einen Cache mit voller Zuordnungsfreiheit steigt also nicht nur die Anzahl der Vergleicher, sondern auch die Größe jedes einzelnen Vergleichers sowie die Größe des Tag-Speichers. Im vorliegenden Fall sind zum Speichern der Etiketten $2^{\#q} \cdot \#t = 2^2 \cdot 6 = 24$ Bit erforderlich.

Aufgrund des großen Schaltungsaufwands zur Verwaltung des Caches werden vollassoziative Cache-Speicher nur für Spezialfälle mit kleinem Speichervolumen realisiert.

Aufgabe 3.10 *Cache mit voller Zuordnungsfreiheit*
Ermitteln Sie Adressaufteilung und Größe des Etikett-Speichers bei folgenden Systemdaten:
16-Bit Adressbus, 512 Byte Cachegröße, 16 Byte Blockgröße. ◇

3.4.3 Cache mit mehrfach satzassoziativer Zuordnung

Zum Erreichen einer hohen Flexibilität bei vertretbarem Schaltungsaufwand wird als Mittelweg zwischen den bisher beschriebenen Organisationen der ***n*-fach satzassoziative Cache** (*n-Way-Set-Associative-Cache*) eingesetzt. Hierbei beschreibt „*n*“ die Anzahl der möglichen Blockrahmen im Cache, in die ein bestimmter HSP-Block gespeichert werden kann. Der *n*-fach satzassoziative Cache besteht im Prinzip aus *n* nebeneinander liegenden Cache mit direkter Zuordnung, wobei man bei einem fehlgeschlagenen Leseversuch (Read-Miss) entscheiden kann, in welchen der Teil-Caches ein aktueller HSP-Block geschrieben werden soll. Dabei ist diese Entscheidung natürlich von grundlegender Bedeutung für die Trefferrate.

Der Cache-Speicher mit direkter Zuordnung ist demnach ein Spezialfall eines satzassoziativen Caches mit $n = 1$. Man sieht sofort, dass *n* gleichzeitig die Anzahl der benötigten Vergleicher angibt, da ja alle möglichen Positionen gleichzeitig geprüft werden müssen. Gängige Werte für *n* liegen zwischen 1 und 8, wobei der Wert meist durch eine Zweierpotenz darstellbar ist. So hatte z.B. der PowerPC 601 einen 8-fach satzassoziativen On-Board Cache-Speicher.

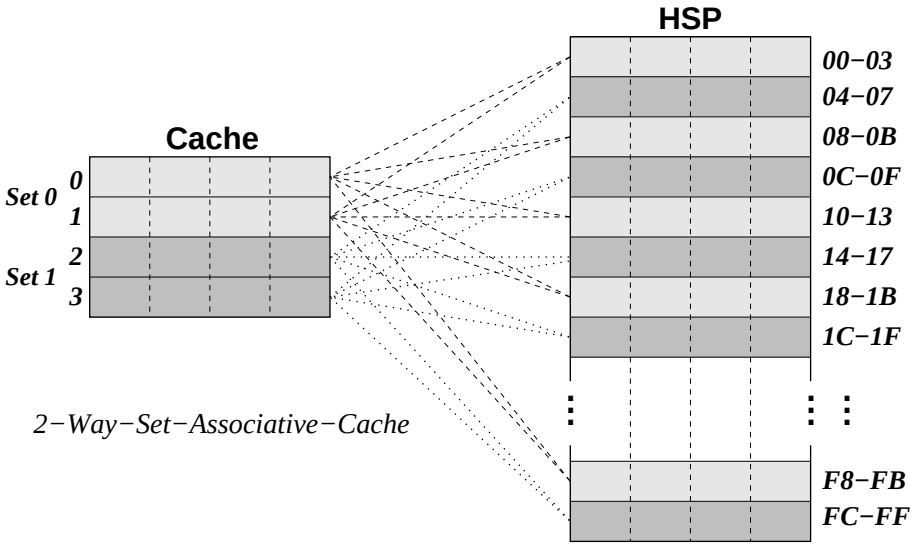


Abbildung 3.15: Cache-Speicher mit 2-fach satzassoziativer Zuordnung

Der Cache wird in gleichgroße **Sätze** (Sets) aufgeteilt, wobei jeder Satz aus *n* Blockrahmen besteht. Jeder Block des Hauptspeichers kann in einen beliebigen Blockrahmen innerhalb eines bestimmten Satzes kopiert werden, wobei ein Teil der Adresse benutzt wird, um genau ein Satz zu selektieren. Die Suche im Cache kann also auf einen Satz beschränkt werden, so dass genau *n* Vergleicher erforderlich sind. Man beachte in Abb. 3.15 insbesondere die Verschachtelung der Sätze im Hauptspeicher (markiert durch unterschiedliche Schraffuren). Durch die hier dargestellte Anordnung werden aufeinander folgende Blöcke des Hauptspeichers in unterschiedliche Sätze abgebildet. Der Zusammenhang zwischen Hauptspeicheradressen und möglichen Positionen im Cache ist in Abb. 3.15 für ein einfaches Beispiel mit 2-fach satzassoziativer Zuordnung dargestellt. Adressaufteilung für dieses Beispiel zeigt Abb. 3.16. Im Beispiel wurden 2 Sets mit jeweils 2 Blöcken verwendet.

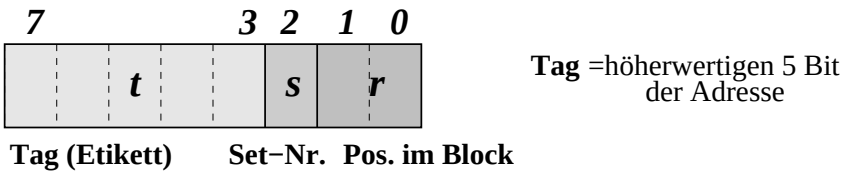


Abbildung 3.16: Adressaufteilung bei 2-fach satzassoziativer Zuordnung

In der Adressaufteilung für den 2-fach satzassoziativen Cache (Abb. 3.16) gibt der rechte Teil nach wie vor die Position innerhalb eines Cache-Blocks an und belegt 2 Bit bei einer Blockgröße von 4

Byte. Das Feld in der Mitte bestimmt die Nummer des Satzes. Da dieses Minimalbeispiel nur 2 Sätze enthält, genügt ein Bit. Der Rest der Adresse muss als Tag (Etikett) gespeichert werden.

Bezeichnet man mit $2^{\#q}$ die Anzahl Blöcke pro Satz, dann berechnet sich die Anzahl der Index-Bits $\#s$ nach folgender Gleichung:

$$2^{\#q} = n \cdot 2^{\#s} \qquad \Rightarrow \qquad \#s = \#q - \log_2 n \tag{3.6}$$

$2^{\#s}$ beschreibt die Anzahl der Sätze im Cache, durch Umstellen von (3.6) ergibt sich $2^{\#s} = 2^{\#q}/n$. Die Berechnung der Anzahl Tag-Bits $\#t$ erfolgt für einen n-fach satzassoziativen Cache nach der Formel:

$$\#t = \text{Adressbreite} - \#s - \#r \tag{3.7}$$

Die Größe des Datenspeichers eines n-fach satzassoziativen Caches in Bytes ist gegeben durch:

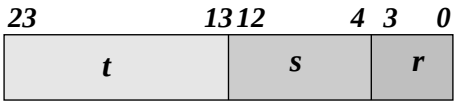
$$C = \text{Cachegröße} = n \times \text{Anzahl Sets} \times \text{Blockgröße} = n \cdot 2^{\#s} \cdot 2^{\#r} = n \cdot 2^{\#s+\#r} \tag{3.8}$$

Beispiel 3.11 *Pufferspeicher mit 4-fach satzassoziativer Zuordnung*
Folgende Systemdaten seien vorgegeben:
16 MB Hauptspeicher mit 24-Bit Adressbus, Cachegröße 32 KByte, Blockgröße 16 Byte.

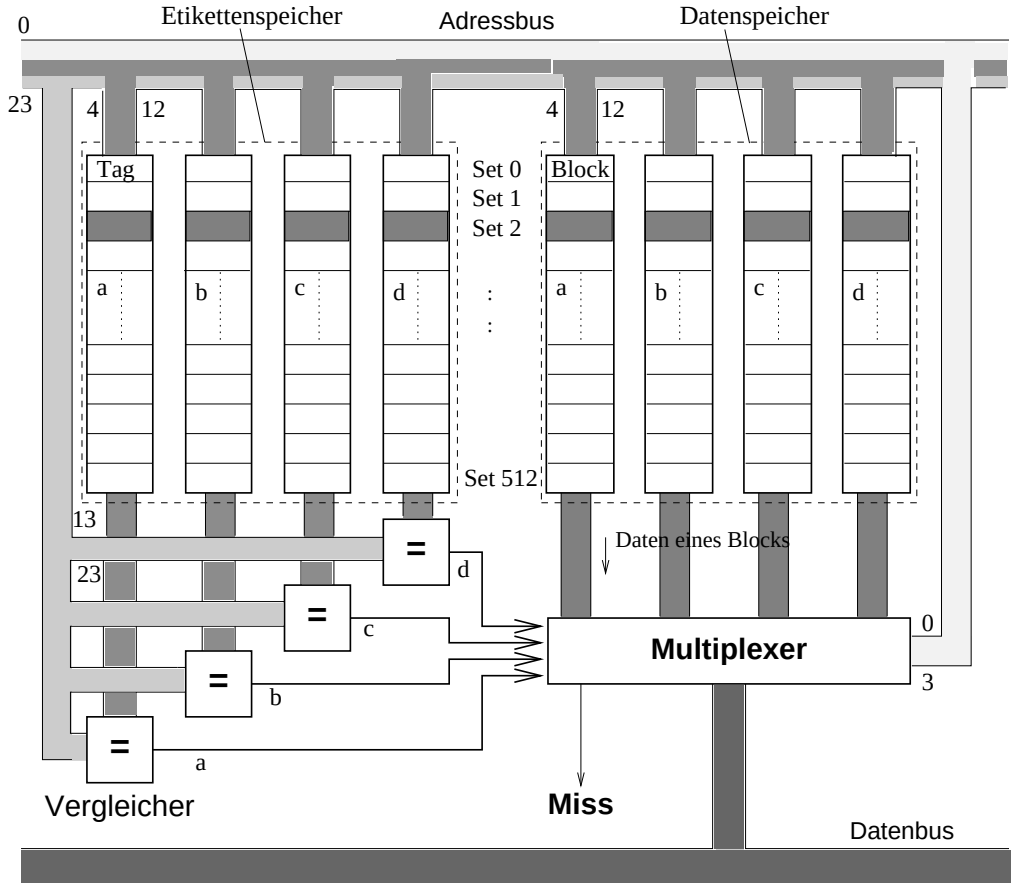
Aus diesen Angaben kann die Adressaufteilung direkt berechnet werden:

$$\begin{aligned}
&\text{Blockgröße} = 16\,\text{Byte} = 2^{\#r} \quad \Rightarrow \quad \#r = 4 && \text{(Adressbits A0–A3)}\\
&C = 32\,\text{KB} = 2^{15} = 2^{\#q+\#r} \quad \Rightarrow \quad \#q = 11 &&\\
&\#s = \#q - \log_2 \#n, \quad \text{mit } n = 4 \text{ folgt } \#s = 9 && \text{(Adressbits A4–A12)}\\
&\#t = \text{Adressbreite} - \#s - \#r = 11 && \text{(Adressbits A13–A23)}
\end{aligned}$$

Die Adressaufteilung ergibt sich somit zu:



Anzahl der Blöcke im Cache: $2^{\#q} = 2^{11} = 2048$
Größe des Tag-Speichers: $2^{\#q} \cdot \#t = 22528\,\text{Bit} = 2816\,\text{Byte}$



Die Abb. zeigt den schematischen Aufbau dieses 4-fach satzassoziativen Caches. Der Cache enthält 512 Sätze mit jeweils 4 Blöcken, wobei ein Block aus 16 Byte besteht, also $512 \cdot 4 \cdot 16\,\text{Byte} = 32768\,\text{Byte} = 32\,\text{KB}$. Man erkennt die vier parallelen Datenpfade für die jeweils 4 Blöcke in einem Satz. Bei einem Lesezugriff des Prozessors wird durch die Adressleitungen A4–A12 ein Satz bestimmt, beispielsweise Satz 2 (im Bild grau markiert). Da Daten- und Etikettspeicher parallel arbeiten, stehen sämtliche Informationen nach Ablauf der Cache-Zugriffszeit gleichzeitig an den vier Vergleichern und an der Auswahllogik (Multiplexer) zur Verfügung. Die vier Vergleicher prüfen nun, ob eines der gespeicherten Tags mit dem oberen Teil (A13–A23) der gesuchten Adresse übereinstimmt. Das Ergebnis des Vergleichs wird von der Auswahllogik (Multiplexer) zur Auswahl eines Blockes benutzt. Mit Hilfe des niederwertigen Adressteils (A0–A3) wird zuletzt die gesuchte Information innerhalb eines Blockes selektiert und dem Datenbus zugeführt. Falls ein Cache-Miss auftritt, erkennt die Auswahllogik (Multiplexer), dass keiner der Vergleicher aktiviert wurde und meldet den Misserfolg an die Cache-Verwaltung. □

Aufgabe 3.11 *Cache mit 8-fach satzassoziativer Zuordnung*
Ermitteln Sie Adressaufteilung und Größe des Etikett-Speichers bei folgenden Systemdaten:
16-Bit Adressbus, Cachegröße 8 KB, Blockgröße 2 Byte. ◇

Wie beim direkt abgebildeten Cache wird der Indexteil x der Adresse einer Abbildungsfunktion f unterworfen und adressiert mit Hilfe eines Satz-Decoders einen Satz. Während beim direkt abgebildeten Cache hier nur das Tag eines Blockes auf Gleichheit geprüft wird, müssen hier alle Tags des selektierten Satzes geprüft werden. Bei Gleichheit eines Tags liegt ein Treffer vor, dann wird noch der Status geprüft und anhand der Wortadresse das gesuchte Speicherwort aus dem Block ausgewählt. Falls festgestellt wird, dass der gesuchte Block in dem Satz nicht vorhanden ist, ergibt sich ein Cache-Fehlzugriff. Im Falle eines Cache-Fehlzugriffs wird ein Cache-Block nach einer **Ersetzungsstrategie** durch den durch $f(x)$ selektierten Block ersetzt. Die Ersetzungsstrategie wählt einen der $2^{\#q-\#s}$ im betreffenden Satz abgelegten Blöcke zur Verdrängung aus. Daher besitzt die Verdrängungsstrategie erheblichen Einfluss auf die Trefferrate und stellt damit eine weitere wichtige Entwurfsentscheidung dar. Wir werden diese im nächsten Abschnitt betrachten.

Bei Cache-Speichern mit geringer Assoziativität müssen nur wenige Blockrahmen pro Satz beim Zugriff durchsucht werden, was die Implementierung vereinfacht. Der Einsatz eines Cache-Speichers mit mehrfach assoziativer Zuordnung ermöglicht eine Erhöhung der Flexibilität und dadurch auch eine Steigerung der Trefferrate bei geringem Schaltungsmehraufwand. Aus diesem Grund sind die heutigen Cache-Speicher fast immer in dieser Organisation ausgeführt. Bei Primär-Cache-Speichern kommt meist eine zwei- bis vierfach satzassoziative Cache-Speicherorganisation zum Einsatz.

Zuordnung	**direkt**	***n*-fach satzassoziativ**	**vollassoziativ**
$\#r =$	$\log_2(\text{Blockgröße})$		
$\#q =$	$\log_2(\text{Cachegröße}) - \#r$		
$\#t =$	Adressbreite $-\#s - \#r$		
$\#s =$	$\#q$	$\#q - \log_2 n$	0
Adressaufteilung	t | q | r	t | s | r	t | r

Tabelle 3.1: Adressaufteilung verschiedener Cache-Organisationen im Überblick

Aufgabe 3.12 *Cache-Design*
In einem Computer mit einem 32-Bit breiten Adressbus wird ein 128 KB großer *8-fach satzassoziativer* Cache mit 32 Byte Blockgröße verwendet.

a) Geben Sie die Adressaufteilung an.
b) Geben Sie die Größe des Tag-Speichers für einen Cache mit *Write-Back* Strategie an!

◇

3.4.4 Ersetzungsstrategien

Bei einem mehrfach assoziativen Cache muss nach einem Fehlzugriff ein konkurrierender Block aus dem Cache entfernt werden, falls kein freier Blockrahmen im Cache mehr zur Verfügung steht. Es stellt sich die Frage, welcher Block aus dem Cache entfernt werden soll. Dieser Block wird durch die **Verdrängungsstrategie** ausgewählt.

Die Verdrängungsstrategie ist in Hardware realisiert, da sie sehr schnell reagieren muss. Für vollassoziative Cache-Speicher bietet sich eine Zufallsstrategie an, da diese den kleinsten Hardware-Aufwand verursacht. Bei den meisten *n*-fach satzassoziativen Caches wird die sog. **LRU-Strategie** (*Least-Recently-Used*) angewendet, die den Cache-Block verdrängt, auf den am längsten nicht mehr zugegriffen wurde. Dazu wird für jeden Satz eine Liste geführt, in der die Blöcke nach dem Zeitpunkt ihrer letzten Aktivität geordnet sind. Falls nun ein Block entfernt werden muss, trifft es den Block, dessen letzte Aktivität am weitesten in der Vergangenheit liegt. Die Verwaltung dieser Liste erzeugt keine zeitlichen Probleme, da eine Veränderung der Einträge nur im Falle eines Cache-Miss notwendig ist. In einem solchen Fall muss der Cache-Speicher aber ohnehin auf den langsameren Hauptspeicher warten und hat deshalb reichlich Zeit zur Verwaltung der Liste. Speziell bei vierfach satzassoziativen Cache-Speichern wird sogar meist die einfachere **Pseudo-LRU-Strategie** implementiert, die mit drei statt vier Bits auskommt, jedoch gelegentlich auch den zweitletzten Cache-Block überschreibt.

Um die Effektivität einer Strategie zu beurteilen, wird als Vergleichsmaßstab die theoretisch optimale, praktisch jedoch nicht realisierbare Ersetzungsstrategie (**OPT**) auf eine Folge von Adressreferenzen angewendet. OPT ist deshalb nicht realisierbar, weil die Anwendung dieser Strategie das vollständige Wissen über alle *zukünftigen* Speicherzugriffe erfordert. OPT ergibt immer die minimale Anzahl von Cache-Misses für eine spezielle Adressfolge. Die Regel beim Anwenden von OPT lautet: Entferne immer den Block aus dem Cache, deren nächste Aktivität am weitesten in der Zukunft liegt.

Warum ist OPT tatsächlich optimal? Angenommen, es gäbe eine bessere Strategie, genannt MEGA-OPT. Beide Strategien bearbeiten die gleiche Adressfolge und starten mit gleichen Speicherzuständen. Um tatsächlich besser zu sein, müsste zu irgendeinem Zeitpunkt ein Cache-Miss bei OPT und gleichzeitig ein Hit bei MEGA-OPT auftreten. Angenommen dieser Hit betreffe Block *A*. OPT hätte jedoch nicht mehr Block *A* im Cache sondern stattdessen Block *B*. Dies bedeutet, dass *B* **vorher** angesprochen wurde, sonst wäre *B* von OPT entfernt worden. Zu diesem früheren Zeitpunkt muss demnach bei MEGA-OPT ein Cache-Miss aufgetreten sein. OPT war also zum Zeitpunkt des Miss beim Zählen von Cache Treffern bereits in Führung und MEGA-OPT kann mit seinem Hit bestenfalls ausgleichen. Man sieht, OPT ist tatsächlich die bestmögliche Strategie!

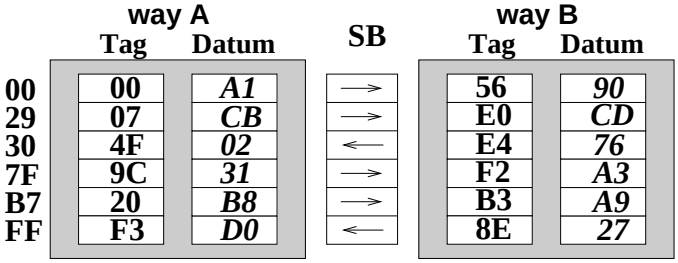
Bei einem direkten Vergleich mit OPT erreicht LRU insbesondere bei verschachtelten kompakten Programmschleifen nahezu die optimale Leistung. Falls jedoch längere sequentielle Abschnitte auftreten, sinkt die Leistung der LRU-Strategie stark ab. Verschiedene Versuche haben gezeigt, dass auch eine Selektion nach dem Zufallsprinzip eine gute Leistung erreichen kann, insbesondere bei Rechnern, die mit stark unterschiedlichen Programmen arbeiten.

Beispiel 3.12 *Zugriffe beim Cache-Speicher mit 2-fach satzassoziativer Zuordnung*
Gegeben sei ein 8-Bit-Prozessor mit einem 16 Bit breiten Adressbus und einem *Two-Way-Set-Associative*-Cache. Der Zugriff zum Speicher erfolge nach der *Write-Through*-Strategie durch die folgenden Assemblerbefehle (0000\leq$ ADR $\leq$$FFFF):

LDA ADR	Lade den Akkumulator mit dem Inhalt der Speicherstelle ADR
STA ADR	Speichere den Inhalt des Akkumulators unter der Adresse ADR

Der Hauptspeicher sei Byte-adressiert und die Blockgröße sei der Einfachheit genau ein Byte. Die folgende Tabelle zeige einen unzusammenhängenden Ausschnitt aus der aktuellen Belegung des Arbeitsspeichers (HSP). Die darauf folgende Abbildung zeige einen wiederum unzusammenhängenden Ausschnitt aus der Belegung des Caches mit 512 Einträgen zum Beginn eines Beobachtungszeitraumes, wobei die links notierte HEX-Zahl den Index des jeweiligen Cache-Rahmens (1 Byte) angibt.

Adresse	Datum	Adresse	Datum	Adresse	Datum	Adresse	Datum
0729	*CB*	6A00	*C5*	907F	*10*	A1B7	*76*
1484	*8A*	7D30	*4B*	A17F	*5B*	F3FF	*D0*
4F30	*02*	8EFF	*27*				



1. LDA $7D30
2. STA $A1B7
3. LDA $4F30
4. LDA $F3FF

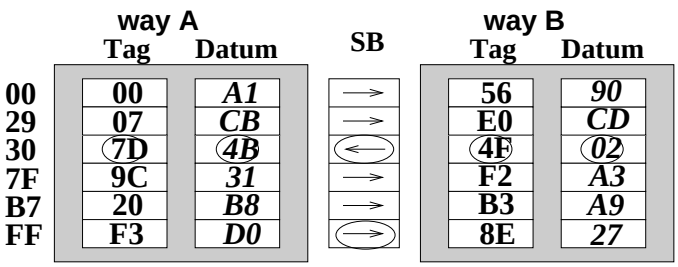
Der Einfachheit halber ist in der Mitte zwischen den beiden Blöcken eines Cache-Satzes das Ersetzungsbit in Form eines Pfeiles angegeben. Der Pfeil zeige stets auf den im nächsten Zugriff eventuell zu verdrängenden Eintrag. Dieser Eintrag werde nach dem LRU-Verfahren ausgewählt, d.h. es werde jeweils derjenige Eintrag verdrängt, auf den am längsten nicht mehr zugegriffen wurde.

Man gebe für die in der Tabelle angegebenen Speicherzugriffe an, ob dabei ein *Hit* oder ein *Miss* vorliegt. Dabei soll die aktuelle Hauptspeicheradresse (HSP), der Inhalt des *Akkus* sowie der *Index*, das *Tag* (Etikett) und die *Daten* des aktuellen Cache-Satzes (*way A*, *SB-Bit*, *way B*) nach Ausführung jedes Befehls angegeben werden.

					Cache					
						way A			**way B**	
Befehl	Hit	miss	Akku	HSP-Adr.	Index	Tag	Datum	**SB**	Tag	Datum
1. LDA $7D30		R	4B	7D30	30	7D	4B	→	E4	76
2. STA $A1B7		W	4B	A1B7	B7	20	B8	→	B3	A9
3. LDA $4F30		R	02	4F30	30	7D	4B	←	4F	02
4. LDA $F3FF	R		D0	F3FF	FF	F3	D0	→	8E	27

Beim ersten Befehl LDA $7D30 handelt es sich um einen Ladebefehl auf die HSP-Zelle mit der Adresse 7D30, der Index dieser Adresse ist 30, das Tag ist 7D. Folglich wird im Cache unter dem Index 30 das Tag 7D gesucht. Unter dem Index 30 sind aber die Tags 4F und E4 eingelagert, so dass es sich um ein Read-Miss handelt. Read-Miss bedeutet, dass der Cache aktualisiert wird, und da das SB-Bit auf den way-A zeigt, wird die entsprechende Zelle im Cache mit Tag und Datum überschrieben, sodass sich die in der Tabelle gezeigte Zeile ergibt. Dabei wird das SB-Bit invertiert.
Beim zweiten Befehl STA $A1B7 handelt es sich um ein Write-Miss, der Cache bleibt unverändert, das Datum wird in den HSP geschrieben. Beim dritten Befehl ergibt sich wie im ersten Befehl ein Read-Miss, der Cache wird aktualisiert. Im vierten Befehl wird bei einem Read-Hit nur das SB-Bit verändert.

Nach Ausführung dieser vier Befehle ergibt sich die folgende Cache-Belegung:

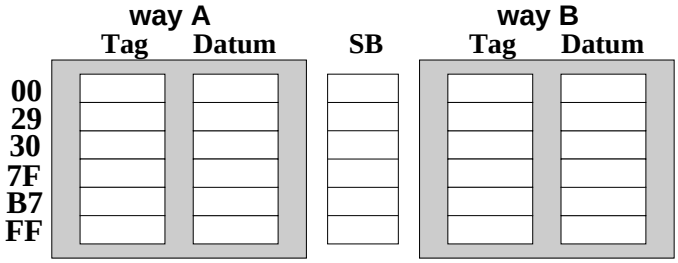


□

Aufgabe 3.13 *Zugriffe beim Cache-Speicher mit 2-fach satzassoziativer Zuordnung*
Gegeben sei der 8-Bit-Prozessor mit 16 Bit breiten Adressbus und *2-Way-Set-Associative*-Cache aus Beispiel 3.12. Es gelten die nach der vierten ausgeführten Anweisung entstandenen HSP- und Cache-Inhalte. Nach diesen vier Anweisungen sollen die folgenden drei Anweisungen ausgeführt werden.

					Cache					
						way A			way B	
Befehl	Hit	Miss	Akku	HSP-Adr.	Index	Tag	Datum	**SB**	Tag	Datum
5. LDA $0729										
6. STA $8EFF										
7. LDA $8EFF										

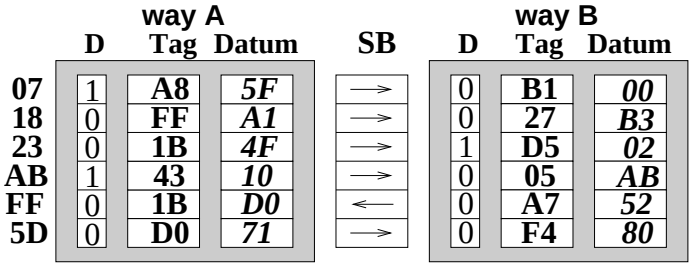
Cache-Belegung nach dem letzten Befehl:



◇

Beispiel 3.13 *Zugriffe beim Cache-Speicher mit 2-fach satzassoziativer Zuordnung*
Gegeben sei der 8-Bit-Prozessor mit 16 Bit breiten Adressbus und *2-Way-Set-Associative*-Cache aus Beispiel 3.12. Der Zugriff zum Speicher erfolge jedoch nach der *Write-Back*-Strategie. Die folgende Tabelle zeige einen Ausschnitt aus der aktuellen Belegung des Arbeitsspeichers (HSP) und die nachfolgende Abb. einen Ausschnitt aus der Belegung des Caches.

Adresse	Datum	Adresse	Datum	Adresse	Datum	Adresse	Datum
05FF	*03*	70A2	*BC*	B094	*A1*	D823	*18*
D05D	*71*	80A6	*5D*	D098	*00*	1BFF	*D0*
C323	*AB*	10AA	*7F*	B307	*D2*	FF18	*A1*



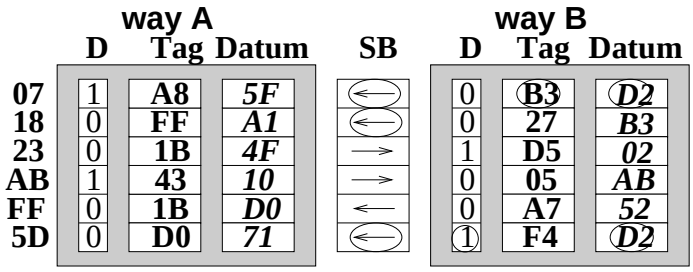
1. LDA $2718
2. STA $B3AB
3. LDA $B307
4. STA $F45D

D kennzeichne das *Dirty-Bit*. Ein nicht gesetztes *Dirty-Bit* (d.h. *D=0*) zeigt die Konsistenz zwischen HSP und Cache an während ein gesetztes Dirty-Bit (d.h. *D=1*) eine mögliche Inkonsistenz zwischen HSP und Cache anzeigt.
Man gebe für die vier Speicherzugriffe an, ob ein *Hit* oder ein *Miss* vorliegt sowie die aktuelle HSP-Adresse (HSP), den Inhalt des *Akkus*, den *Index*, die *Tags* (Etiketten), das *Dirty-Bit* und die *Daten* des aktuellen Cache-Satzes nach Ausführung der Befehle.

Beim ersten Befehl LDA $2718 handelt es sich um einen Lesebefehl auf die HSP-Zelle 2718, der Index ist 18, das Tag ist 27. Unter dem Index 18 befindet sich im way-B der gesuchte Eintrag mit den Tag 27, folglich liegt ein Read-Hit vor, das Dirty-Bit ist nicht von Bedeutung. Ebenso beim zweiten Befehl, bei einem Write-Miss wird das Datum direkt in den HSP zurückgeschrieben.

					Cache							
						way A				way B		
Befehl	Hit	Miss	Akku	HSP-Adr.	Index	**D**	Tag	Datum	**SB**	**D**	Tag	Datum
1. LDA $2718	R		B3	2718	18	0	FF	A1	⟵	0	27	B3
2. STA $B3AB		W	B3	B3AB	AB	1	43	10	⟶	0	05	AB
3. LDA $B307		R	D2	B307	07	1	A8	5F	⟵	0	B3	D2
4. STA $F45D	W		D2	F45D	5D	0	D0	71	⟵	1	F4	D2

Beim dritten Befehl, einem Read-Miss, ist der Eintrag unter dem Index 07 im way-B zu ersetzen. Dazu muss nun vorher das Dirty-Bit geprüft werden. Ist dies gesetzt, dann muss der entsprechende Eintrag vor dem Überschreiben in den HSP zurückgeschrieben werden. Ist das Dirty-Bit wie im vorliegenden Fall nicht gesetzt, dann kann der Eintrag überschrieben werden. Beim vierten Befehl STA $F45D, einem Write-Hit, wird das Datum im Rückschreibeverfahren nur im Cache eingetragen und erst bei Bedarf in den HSP zurückgeschrieben. Da sich hier eine Inkonsistenz zwischen Cache und HSP ergibt, wird das Dirty-Bit gesetzt. Nach Ausführung dieser vier Befehle ergibt sich die folgende Cache-Belegung:

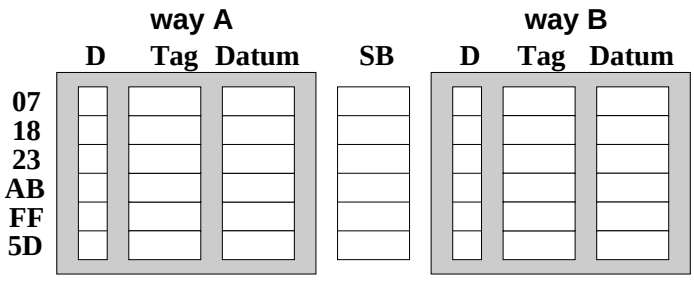


□

Aufgabe 3.14 *Zugriffe beim Cache-Speicher mit 2-fach-satzassoziativer Zuordnung*
Gegeben sei der 8-Bit-Prozessor mit 24 Bit breiten Adressbus und *2-Way-Set-Associative*-Cache aus Beispiel 3.13. Es gelten die nach der vierten Anweisung entstandenen HSP- und Cache-Inhalte. Danach sollen die folgenden vier Anweisungen ausgeführt werden.

					Cache							
						way A				way B		
Befehl	Hit	Miss	Akku	HSP-Adr.	Index	**D**	Tag	Datum	**SB**	**D**	Tag	Datum
5. LDA $05FF												
6. STA $D05D												
7. LDA $C323												
8. STA $D823												

Cache-Inhalt nach Ausführung der letzten Anweisung:



◇

Aufgabe 3.15 *Vergleich der drei Cache-Organisationen*
Bei einem Byte-adressierten Rechner sollen vergleichsweise ein direkt abgebildeter Cache, ein 2-fach satzassoziativer Cache und ein vollassoziativer Cache eingesetzt werden. Die drei Cache-Speicher haben jeweils eine Speicherkapazität von 32 Bytes und werden in Blöcken von je vier Bytes geladen. Die Hauptspeicheradresse umfasst 32 Bits.

a) Geben Sie für die drei Cache-Speicher an, wie viele Bits zur Verwaltung (Tag- und Zustandsbits) eines Cache-Blocks benötigt werden. Dabei sollen für den Zustand des Cache-Blocks zwei Statusbits verwendet werden (Valid-Bit und Dirty-Bit).

b) Betrachten Sie die Folge der Lesezugriffe auf die folgenden HSP-Adressen:
`$0046 $0009 $0044 $00B9 $0011 $00FB $0055 $00F9 $005C $0006`
Nehmen Sie an, die Caches seien zu Beginn leer. Ermitteln Sie, ob es sich beim Lesezugriff auf die jeweiligen Adressen um einen Treffer (Read-Hit) oder einen Fehlzugriff (Read-Miss) handelt. Falls notwendig, werde die LRU-Ersetzungsstrategie verwendet. Geben Sie die Tag-Zustände der Cache-Speicher nach dem letzten Zugriff an.

◇

3.5 Cache Analyse

3.5.1 Cache Analyse durch Tracing

Um die zu erwartende Leistung eines Cache-Speichers bei verschiedenen Organisationen schon in der Entwurfs-Phase abzuschätzen, kann der Designer die Bearbeitung von typischen Adressfolgen simulieren. Diese Adressfolgen (**Traces**) werden entweder von einer speziellen Hardware in einem realen System protokolliert oder durch Simulation bestimmt. Das Protokollieren ist jedoch nur für Maschinen bis zu mittleren Geschwindigkeitsbereichen möglich, da die spezielle Hardware ein Mehrfaches schneller sein muss als das zu beobachtende System. Die Simulatoralternative ist zwar technologisch leicht zu beherrschen, dafür liegt die Geschwindigkeit aber mindestens um den Faktor 1000 unter der Hardware-Lösung.

Aufgrund der anfallenden Datenmengen entstehen dabei häufig Probleme beim Erzeugen von Adressreferenzen:

1. Der protokollierte oder simulierte Programmabschnitt ist nicht unbedingt repräsentativ für das Gesamtverhalten.
2. Die Initialisierungsphase, in der der Pufferspeicher zu Beginn einer Simulation zum ersten mal mit Daten gefüllt wird, kann das Ergebnis überproportional verfälschen.
3. Selbst eine Adressfolge von einigen Millionen Referenzen könnte noch zu kurz sein, um ein realistisches Bild vom Verhalten des Cache zu vermitteln.

Insbesondere das zweite Problem kann merkwürdige Effekte erzeugen. So wird sich zum Beispiel bei gleicher Trace-Länge das Hit-zu-Miss Verhältnis bei einer Verdoppelung der Cachegröße nicht unbedingt verbessern, da der größere Pufferspeicher einen viel größeren Teil der Adressfolge zur Initialisierung benötigt. Als Richtwert gilt daher, dass die Trace-Länge mit der 1,5-fachen Potenz zur Cache-Vergrößerung wachsen sollte. Bei Verdoppelung der Cachegröße muss demnach die Trace-Länge nahezu um das Dreifache wachsen. Ein Beispiel verdeutlicht diesen Effekt.

Beispiel 3.14 *Initialisierungsfehler*
Bei der Simulation eines Direct-Mapped-Cache mit 512 Blöcken wurden mit einer Folge von 100 000 Adressen 600 Misses ermittelt. Dies ergibt eine Trefferrate von 0,994. Der gleiche Trace wurde für ein verändertes Cache-Design gleicher Gesamtgröße, jedoch mit 2048 Blöcken verwendet. Hierbei wurden 1500 Misses gemessen, was einer Trefferrate von 0,985 entspricht.

Bei genauerer Betrachtung fällt jedoch auf, dass der zweite Cache zu maximal 3/4 gefüllt sein kann, da bei jedem Miss nur ein Block geladen wird. Die Vergrößerung der Zahl der Fehlzugriffe kann demnach völlig auf die Initialisierung zurückgeführt werden, so dass keine Aussage zum tatsächlichen Leistungsvergleich möglich ist. □

Bei einem direkt abgebildeten Cache kann der Initialisierungsfehler dadurch eliminiert werden, dass ein Cache-Miss nur dann gezählt wird, wenn der gleiche Block bereits mindestens einmal geladen wurde. Die effektive Trace-Länge wird dadurch meist nur geringfügig verkürzt. Das gleiche Verfahren kann auch bei einem n-fach satzassoziativen Pufferspeicher angewendet werden. Allerdings kann hierbei die Messung erst beginnen, wenn mindestens n Zugriffe pro Satz erfolgt sind, da sich erst dann die tatsächliche Größe des Caches bemerkbar macht. Die benötigte Trace-Länge zum Messen der gleichen Anzahl an Speicherzugriffen kann dadurch für $n = 4$ um über 25% ansteigen. Diese Überlegungen können zur Abschätzung einer Trace-Mindestlänge genutzt werden.

Beispiel 3.15 *Abschätzung einer Trace-Mindestlänge*
Ein 4-fach satzassoziativer Cache-Speicher der Größe 32 KB enthalte 512 Sätze bei einer Blockgröße von 16 Byte. Falls eine Trefferrate von 0,99 angenommen wird, sind 5 Millionen Referenzen erforderlich, um im Mittel 100 Misses pro Satz zu erzeugen. Bei einem 4-mal größeren Cache halbiert sich die Miss-Rate ungefähr, und der Cache enthält die 4-fache Anzahl an Sätzen, d.h. die Trace-Länge muss um den Faktor 8 steigen, um die gleiche statistische Genauigkeit zu erzielen. □

Aufgabe 3.16 *Abschätzung einer Trace-Mindestlänge*
Wie viele Referenzen werden zur Initialisierung eines vollassoziativen Cache-Speichers mit 512 Blöcken bei einer Trefferrate von 0,95 benötigt? ◇

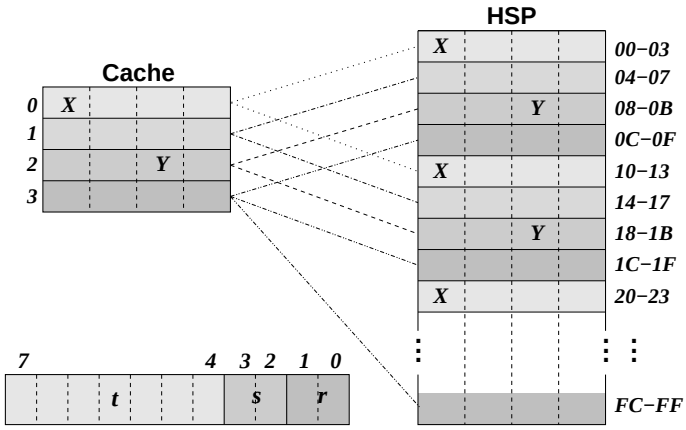
Damit der Simulationsaufwand nicht jeden wirtschaftlich und zeitlich sinnvollen Rahmen sprengt, wurden Verfahren zur Verkürzung der benötigten Adressfolgen entwickelt, ohne die Genauigkeit zu reduzieren. Die einfachste Regel hierfür lautet:

Produziere eine reduzierte Adressfolge durch Speicherung aller Cache-Misses, die entstehen, wenn der komplette Trace auf einen einfach assoziierten Cache mit n Sätzen und der Blockgröße L angewendet wird. Die Simulation verschiedener Cache-Designs mit der Blockgröße L und mindestens n Sätzen ergibt bei Verwendung der reduzierten Adressfolge die gleiche Anzahl an Cache-Misses wie bei Verwendung der kompletten Folge.

Die reduzierte Adressfolge erreicht meist weniger als 20% der ursprünglichen Länge. Bei der Berechnung der Trefferrate müssen selbstverständlich die entfernten Zugriffe mitgezählt werden, da diese bei jedem anderen Design mit gleichem *L* einen Treffer erzielt hätten.

Beispiel 3.16 *Cache Simulation*
Wir betrachten den *direkt abgebildeten (Direct-Mapped)-Cache* aus Abb. 3.11 mit 4 Sätzen und Blockgröße 4 Byte, wobei die 4 Sätze direkt den 4 Cacherahmen entsprechen. Die folgende Adressfolge soll durch Anwendung der obigen Regel reduziert werden.

```
$0E $1B $01 $A7 $05 $0E $1B $04 $A7 $07 $0E $1B $2F $E3
$05 $0E $1B $4F $FD $85 $0E $1B $01 $AA $04 $0E $07 $85
```



Bei 2 Byte-Adressen, 4 Sätzen (#q=2) (4 Blöcken) und einer Blockgröße 4 Byte (#r=2) gibt es folgende Zuordnung von Bytes zu Blöcken und Indexen 0 bis *F*:

Block-rahmen	Index	Adress-Blöcke							
0	0-3/00	00-03	10-13	20-23	30-33	40-43	50-53	60-63	70-73
		80-83	90-93	A0-A3	B0-B3	C0-C3	D0-D3	E0-E3	F0-F3
1	4-7/01	04-07	14-17	24-27	34-37	44-47	54-57	64-67	74-77
		84-87	94-97	A4-A7	B4-B7	C4-C7	D4-D7	E4-E7	F4-F7
2	8-B/10	08-0B	18-1B	28-2B	38-3B	48-4B	58-5B	68-6B	78-7B
		88-8B	98-9B	A8-AB	B8-BB	C8-CB	D8-DB	E8-EB	F8-FB
3	E-F/11	0C-0F	1C-1F	2C-2F	3C-3F	4C-4F	5C-5F	6C-6F	7C-7F
		8C-8F	9C-9F	AC-AF	BC-BF	CC-CF	DC-DF	EC-EF	FC-FF

Nach dem Zugriff auf die ersten acht Adressen ergeben sich bei einem Direct-Mapped-Cache folgende Cache-Belegungen:

$0E ⇒ Tag=0, Index=E ⇒ Tag 0 nach Block 3 geladen (Miss)
$1B ⇒ Tag=1, Index=B ⇒ Tag 1 nach Block 2 geladen (Miss)
$01 ⇒ Tag=0, Index=1 ⇒ Tag 0 nach Block 0 geladen (Miss)
$A7 ⇒ Tag=A, Index=7 ⇒ Tag A nach Block 1 geladen (Miss)
$05 ⇒ Tag=0, Index=5 ⇒ Tag 0 nach Block 1 geladen (Miss)
$0E ⇒ Tag=0, Index=E ⇒ Tag 0 bereits in Block 3 (Hit)
$1B ⇒ Tag=1, Index=B ⇒ Tag 1 bereits in Block 2 (Hit)
$04 ⇒ Tag=0, Index=4 ⇒ Tag 0 bereits in Block 1 (Hit)

Block	Index	Tag des Cache-Inhalts		
0	0-3/00	0(M)		
1	4-7/01	A(M)	0(M)	0(H)
2	8-B/10	1(M)	1(H)	
3	C-F/11	0(M)	0(H)	

Steht beispielsweise im Cache-Rahmen 0 der ein HSP-Inhalt mit dem Tag 0, dann stehen in diesem Cache-Rahmen auch die drei benachbarten Zelleninhalte mit den Adressen 01, 02 und 03. Diese vier Bytes werden jeweils gleichzeitig im Cache ein- und ausgelagert. Aufgrund des Verlaufs der 28 Zugriffe auf die komplette Adressfolge ergeben sich die folgenden Cache-Belegungen (x(M) steht für die Belegung mit Tag x bei einem Miss, x(H) für die Belegung mit Tag x bei einem Hit):

Block	Index	Tag des Cache-Inhalts
0	0-3/00	0(M) E(M) 0(M)
1	4-7/01	A(M) 0(M) 0(H) A(M) 0(M) 0(H) 8(M) 0(M) 0(H) 8(M)
2	8-B/10	1(M) 1(H) 1(H) 1(H) 1(H) A(M)
7	C-F/11	0(M) 0(H) 0(H) 2(M) 0(M) 4(M) F(M) 0(M) 0(H)

Die Adressfolge weist die unterstrichenen Treffer auf:

`$0E $1B $01 $A7 $05 $0E $1B $04 $A7 $07 $0E $1B $2F $E3`
`$05 $0E $1B $4F $FD $85 $0E $1B $01 $AA $04 $0E $07 $85`

Nach Eliminierung der Treffer erhalten wir als reduzierte Adressfolge:

`$0E $1B $01 $A7 $05 $A7 $07 $2F $E3 $0E $4F $FD $85 $0E $01 $AA $04 $85`

Anwendung der Original-Adressfolge zur Simulation eines 2-fach satzassoziativen Caches gleicher Blockgröße mit 8 Sätzen und LRU-Stretegie

`$0E $1B $01 $A7 $05 $0E $1B $04 $A7 $07 $0E $1B $2F $E3`
`$05 $0E $1B $4F $FD $85 $0E $1B $01 $AA $04 $0E $07 $85`

Die Original-Adressfolge führt auf die folgenden Cache-Belegungen:

Block	Index		Tag des Cache-Inhalts	SB
0	0-3/00	way A	0(M) 0(H)	01
		way B	E(M)	
1	4-7/01	way A	A(M) A(H) 8(M) 8(H)	010101
		way B	0(M) 0(H) 0(H) 0(H) 0(H) 0(H)	
2	8-B/10	way A	1(M) 1(H) 1(H) 1(H) 1(H)	0
		way B	A(M)	
3	C-F/11	way A	0(M) 0(H) 0(H) 0(H) F(M) 0(M)	0101
		way B	2(M) 4(M)	

Die Adressfolge weist die unterstrichenen Treffer auf:

`$0E $1B $01 $A7 $05 $0E $1B $04 $A7 $07 $0E $1B $2F $E3`
`$05 $0E $1B $4F $FD $85 $0E $1B $01 $AA $04 $0E $07 $85`

Nach Eliminierung der Treffer erhalten wir als reduzierte Adressfolge:

`$0E $1B $01 $A7 $05 $2F $E3 $4F $FD $85 $0E $AA`

Die Anwendung der durch den Direct-Mapped-Cache reduzierten Adressfolge

`$0E $1B $01 $A7 $05 $A7 $07 $2F $E3 $0E $4F $FD $85 $0E $01 $AA $04 $85`

zur Simulation eines 2-fach satzassoziativen Caches gleicher Blockgröße mit 8 Sätze und LRU-Stretegie ergibt die folgenden Cache-Belegungen:

Block	Index		Tag des Cache-Inhalts	SB
0	0-3/00	way A	0(M) 0(H)	01
		way B	E(M)	
1	4-7/01	way A	A(M) A(H) 8(M) 8(H)	01010
		way B	0(M) 0(H) 0(H)	
2	8-B/10	way A	1(M)	0
		way B	A(M)	
3	C-F/11	way A	0(M) 0(H) F(M) 0(M)	0101
		way B	2(M) 4(M)	

Die Adressfolge weist die unterstrichenen Treffer auf:

`$0E $1B $01 $A7 $05` <u>`$A7`</u> <u>`$07`</u> `$2F $E3` <u>`$0E`</u> `$4F $FD $85 $0E` <u>`$01`</u> `$AA` <u>`$04`</u> <u>`$85`</u>

Nach Eliminierung der Treffer erhalten wir als reduzierte Adressfolge:

`$0E $1B $01 $A7 $05 $2F $E3 $4F $FD $85 $0E $AA`

Das ist exakt die gleiche Liste wie sie bei Anwendung der Original-Adressfolge entsteht. Es ergibt sich demnach für den 2-fach satzassoziativen Cache in diesem Beispiel eine Miss-Rate von $12/28 = 0,429$, dies entspricht einer Trefferrate von $0,571$. Die geringe Trefferrate resultiert aus der Tatsache, dass die Regel zur Mindestlänge eines Trace grob missachtet wurde. □

Aufgabe 3.17 *Ersetzungsstrategie*
Welche Trefferrate könnte mit den Daten aus Beispiel 3.16 bei Anwendung der OPT-Strategie erreicht werden? ◇

3.5.2 Analytische Berechnung der Zugriffszeit

Wir wollen nun eine kurze Einführung in die Berechnung der mittleren Zugriffszeit unter Berücksichtigung der Architekturparameter geben. Eine einfache Gleichung zum Abschätzen der **Zugriffszeit** wurde bereits in Abschnitt 3.3.1 mit Gl. (3.1) vorgestellt. Nun soll nun die Berechnung der effektiven Zugriffszeit unter Berücksichtigung verschiedener Einflussgrößen durch die getrennte Betrachtung der möglichen Zugriffsarten etwas genauer ausgeführt werden. Wir unterscheiden zunächst zwischen Schreib- und Lesezugriffen, wobei die Wahrscheinlichkeit für einen Schreibzugriff durch p_{wr} ausgedrückt wird. Da die Summe der Wahrscheinlichkeiten immer 1 ergeben muss und es keine weiteren Alternativen gibt, folgt für die Wahrscheinlichkeit eines Lesezugriffs $1-p_{wr}$.

Weiterhin kann in jeder Zugriffsart ein Cache-Miss oder ein Cache-Hit auftreten, ausgedrückt durch die Trefferrate *h*. Die mittlere Gesamtzugriffszeit kann also durch folgende vier Komponenten ausgedrückt werden:

	Hit	Miss
Read	t_{RH}	t_{RM}
Write	t_{WH}	t_{WM}

Die effektive Zugriffszeit ergibt sich durch die gewichtete Addition der vier möglichen Fälle:

$$T_{eff} = (1-p_{wr})\,[h \cdot t_{RH} + (1-h)\,t_{RM}] + p_{wr}[h \cdot t_{WH} + (1-h)\,t_{WM}] \tag{3.9}$$

Die Bestimmung der Einzelkomponenten hängt stark vom Cache-Design sowie von sonstigen Randbedingungen des Systems ab. Die Vorgehensweise wird deshalb für die zwei gängigen Strategien *Write-Through* und *Write-Back* vorgestellt.

1. Zugriffszeit bei Write-Through

Bei einem Cache mit Write-Through können die einzelnen Komponenten nach folgenden Überlegungen ermittelt werden:

Read-Hit: Hierfür ist nur die Zugriffszeit des Cache-Speichers t_{PS} erforderlich.

Read-Miss: Bei einem Read-Miss muss der gesuchte Block aus dem Hauptspeicher geladen werden. Zusätzlich zur vergeblichen Suche im Cache kommt also noch die erforderliche Transportzeit zum Kopieren eines Blocks aus dem Hauptspeicher. Hierbei muss berücksichtigt werden, dass zum Transportieren eines Blockes möglicherweise mehrere Zugriffe auf den Hauptspeicher erforderlich sind, je nach Breite von Datenbus und Blockgröße. Die Anzahl der erforderlichen Zugriffe für einen **Transportzyklus** wird durch z ausgedrückt.

Write-Hit: Beim Cache mit Write-Through erfordert jeder Schreibzugriff die volle Zugriffszeit des Hauptspeichers, da der Wert immer direkt im Hauptspeicher aktualisiert wird. Das schnellere Aktualisieren des Cache kann parallel erfolgen.

Write-Miss: Es ist nur die Zugriffszeit des Hauptspeichers erforderlich unter der Annahme, dass der betroffene Block *nicht* in den Cache geladen wird.

Es ergeben sich also die folgenden Ausdrücke:

	Hit	Miss
Read	t_{PS}	$t_{PS} + z \cdot t_{HS}$
Write	t_{HS}	t_{HS}

Zusammengesetzt entsteht:

$$T_{eff} = (1 - p_{wr})\,[h \cdot t_{PS} + (1-h)\,(t_{PS} + z \cdot t_{HS})] + p_{wr}[h \cdot t_{HS} + (1-h)\,t_{HS}]$$

Und daraus ergibt sich durch Umformung:

$$T_{eff} = (1 - p_{wr})\,t_{PS} + [\,(1 - p_{wr})(1-h)\,z + p_{wr}]\,t_{HS} \tag{3.10}$$

2. Zugriffszeit bei Write-Back

Bei einem Cache mit Write-Back werden die Komponenten bestimmt durch:

Read-Hit: Hierfür ist nur die Zugriffszeit des Cache-Speichers t_{PS} erforderlich.

Read-Miss: Zusätzlich zur erforderlichen Ladezeit könnte der Fall eintreten, dass der zu verdrängende Block modifiziert ist und deshalb zuerst gespeichert werden muss. Die Wahrscheinlichkeit hierfür werde durch p_{dirty} ausgedrückt.

Write-Hit: Bei einem Write Hit ist nur die Cache Zugriffszeit erforderlich.

Write-Miss: Nach der erfolglosen Suche im Cache, dem Laden des Blockes und einem eventuellen Speichern des verdrängten Blocks muss noch der Schreibzugriff auf den Cache-Speicher erfolgen.

Daraus ergeben sich die folgenden Terme:

	Hit	Miss
Read	t_{PS}	$t_{PS} + z(1 + p_{dirty})\,t_{HS}$
Write	t_{PS}	$2\,t_{PS} + z(1 + p_{dirty})\,t_{HS}$

Zusammengesetzt entsteht:

$$\begin{aligned} T_{eff} \;=\; & (1 - p_{wr}) \cdot \{\, h \cdot t_{PS} + (1-h)\,[\,t_{PS} + z(1 + p_{dirty})\,t_{HS}\,]\,\} \\ & + p_{wr}\{\, h \cdot t_{PS} + (1-h)\,[\,2\,t_{PS} + z(1 + p_{dirty})\,t_{HS}\,]\,\} \end{aligned}$$

und nach Umformung:

$$T_{eff} = [\,1 + (1-h)\,p_{wr}\,]\,t_{PS} + (1-h)\,[\,z(1 + p_{dirty})\,]\,t_{HS} \tag{3.11}$$

Aufgabe 3.18 *Berechnung der mittleren Zugriffszeit*
Bei einer speziellen Rechnerarchitektur könnte der Cache die Schreibzugriffe des Prozessors wahlweise im *Write-Through*- oder im *Write-Back*-Verfahren abwickeln, je nach Voreinstellung durch den Benutzer. (Ein Umschalten im laufenden Betrieb ist nicht vorgesehen und löst einen Reset aus!).

Im *Write-Through*-Betrieb werden Schreibvorgänge direkt im Hauptspeicher durchgeführt, der betroffene Block werde nicht in den Cache geladen. Falls sich der Block bereits im Cache befindet, werde der Wert im Cache *gleichzeitig* mit dem Schreibvorgang zum HSP aktualisiert. Bei diesem Verfahren werde eine Trefferrate von $0,9$ erzielt.

Im *Write-Back*-Betrieb werden Schreibvorgänge des Prozessors nur im Cache durchgeführt, der betroffene Block muss daher eventuell in den Cache geladen werden. Bei diesem Verfahren werde nur eine Trefferrate von $0,8$ erzielt, da der Cache öfter verändert wird.

Folgenden Systemparameter sind vorgegeben:

$$t_{PS} = 25\,ns \qquad t_{HS} = 110\,ns \qquad p_{dirty} = 0,3 * p_{wr} \qquad z = 2$$

Geben Sie an, für welchen Wertebereich von p_{wr} das *Write-Through*-Verfahren die bessere Leistung (kleinere effektive Zugriffszeit T_{eff}) bietet! ◇

Aufgabe 3.19 *Untersuchung der Trefferraten verschiedener Cache-Organisationen*
Gegeben seien ein direkt abgebildeter Cache, ein 4-fach satzassoziativer Cache und ein vollassoziativer Cache. Alle drei Cache-Speicher werden durch 32 Bit adressiert und haben eine Kapazität von 16 Blöcken, wobei ein Block jeweils vier Datenwörter umfasst. Bei den assoziativen Caches wird die LRU-Ersetzungsstrategie verwendet.

Die folgende Sequenz zeigt die hexadezimalen Adressen der ersten zwanzig Lesezugriffe eines Programms zur Multiplikation zweier Matrizen mit je 10000 Zeilen/Spalten:

```
1F296FFA, 378CF121, 1F296FFB, 378D1831, 1F296FFC,
378D3F41, 1F296FFD, 378D6651, 1F296FFE, 378D8D61,
1F296FFF, 378DB471, 1F297000, 378DDB81, 1F297001,
378E0291, 1F297002, 378E29A1, 1F297003, 378E50B1
```

a) Geben Sie unter Berücksichtigung der jeweiligen Cache-Organisationsform an, wie viele Adressbits für den Tag- bzw. den Index-Teil benötigt werden. Wie breit ist die Wortadresse?

b) Bestimmen Sie für alle drei Cache-Typen, bei welchen Zugriffen auf die oben genannten Adressen es sich um Cache-Treffer handelt. Führen Sie dazu Buch über die Belegungen der einzelnen Cache-Blöcke. Nehmen Sie an, dass die Caches zu Beginn leer sind. Wie hoch sind die Trefferraten?

◇

3.6 Maßnahmen zur Beschleunigung des Speicherzugriffs

Da die mittlere Zugriffszeit von der Fehlzugriffsrate, dem Fehlzugriffsaufwand und der Cache-Zugriffszeit bei einem Treffer abhängt, kann die Verarbeitungsleistung eines Cache-Speichers auf drei Arten verbessert werden:

- durch Verringern der Fehlzugriffsrate,
- durch Verringern des Fehlzugriffsaufwandes und
- durch Verringern der Cache-Zugriffszeit bei einem Treffer.

Mit dieser Thematik wollen wir uns in diesem Abschnitt beschäftigen.

3.6.1 Verringern der Fehlzugriffsrate

Die mögliche Verbesserung Fehlzugriffsrate hängt entscheidend von einer Analyse der Ursachen für die Fehlzugriffe ab. Diese lassen sich in drei Klassen einteilen:

- **Erstzugriff** (*compulsory* – obligatorisch): Beim ersten Zugriff auf einen Cache-Block befindet sich dieser noch nicht im Cache-Speicher und muss erstmals geladen werden. Diese Art von Fehlzugriffen lässt sich auch als Kaltstartfehlzugriffe (*cold start misses*) oder Erstbelegungsfehlzugriffe (*first reference misses*) bezeichnen. Sie würden sogar in einem unendlich großen Cache-Speicher auftreten und sind unvermeidbar.
- **Kapazität** (*capacity*): Falls der Cache-Speicher nicht alle benötigten Cache-Blöcke aufnehmen kann, müssen Cache-Blöcke verdrängt und eventuell später wieder geladen werden. Fehlzugriffe wegen mangelnder Cache-Kapazität sind von der Cache-Speicherorganisation unabhängig und würden auch in einem vollassoziativen Cache mit entsprechend beschränkter Größe auftreten.
- **Konflikt** (*conflict*): Bei einer satzassoziativen oder direkt abgebildeten Cache-Speicherorganisation können zusätzlich zu den ersten beiden Fehlzugriffsarten Konfliktfehlzugriffe auftreten, da ein Cache-Block potenziell verdrängt und später wieder geladen wird, falls zu viele Cache-Blöcke auf denselben Satz abgebildet werden. Diese Fehlzugriffe werden auch Kollisionsfehlzugriffe (*collision misses*) oder Interferenzfehlzugriffe (*interference misses*) genannt. Kollisionsfehlzugriffe treten nur bei direkt abgebildeten oder satzassoziativen Cache-Speichern beschränkter Größe auf.

Die Gesamtzahl der Fehlzugriffe hängt also von den *3 Cs – Compulsory*, *Capacity*, *Conflict* – ab. Die Zahl der Fehlzugriffe, die durch einen Erstzugriff bedingt sind, kann nur durch die Wahl eines größeren Cache-Blocks und/oder durch eine bessere Verteilung der Daten im Speicher durch den Compiler verringert werden. In beiden Fällen geht es darum, die räumliche Lokalität mehr auszunutzen und mit einem Speicherzugriff mehr erforderliche Daten bereitzustellen. Fehlzugriffe wegen mangelnder Cache-Speicherkapazität lassen sich nur durch größere Caches verringern.

Die Anzahl der durch Konflikte ausgelösten Fehlzugriffe kann jedoch unter der Annahme eines festen Budgets an Cache-Speicherplatz auf dem Chip durch die folgenden Maßnahmen verringert werden:

1. *Größere Cache-Blöcke*: Größere Cache-Blöcke erhöhen die Ausnutzung der räumlichen Lokalität und können somit die Zahl der Erstzugriffs- als auch der Konfliktfehlzugriffe verringern. Allerdings bedeutet das Laden größerer Cache-Blöcke auch einen größeren Fehlzugriffsaufwand und potenziell wiederum eher Konflikte, da der Cache weniger Cache-Blöcke umfasst. Simulationen zeigten, dass die optimale Blockgröße bei 32 – 128 Bytes liegt.
2. *Höhere Assoziativität*: Die Zahl der Konflikte lässt sich durch eine größere Assoziativität entscheidend verringern. Allerdings lässt sich die Assoziativität nicht beliebig steigern, da der Hardware-Aufwand und die Zugriffszeit ansteigen und im Falle eines On-Chip-Cache-Speichers die Taktrate des Prozessors beeinträchtigt wird. Üblich sind heute Assoziativitäten von vier- und achtfach.

3. *Victim-Cache*: Im Falle eines direkt abgebildeten Cache-Speichers tritt häufig das sog. Cache-Flattern (*Thrashing*) ein, das durch mehrfache Speicherzugriffe auf zwei Cache-Blöcke, die auf denselben Cache-Satz abgebildet werden, bedingt ist. Die Folge ist, dass die Cache-Blöcke ständig zwischen dem Cache-Speicher und der nächsten Speicherhierarchieebene ausgetauscht werden. Die Fehlzugriffsrate kann in diesem Fall durch einen sog. Victim-Cache verringert werden. Ein *Victim-Cache* ist ein kleiner Pufferspeicher, auf den genauso schnell wie auf den Cache zugegriffen werden kann und der die zuletzt aus dem Cache geworfenen Cache-Blöcke aufnimmt. Der Victim-Cache wurde in Alpha- und HP-Prozessoren angewendet.

4. *Pseudo-Assoziativität*: Diese dient dazu, bei einem direkt abgebildeten Cache-Speicher die Anzahl der Konflikte zu verringern. Der Cache wird in zwei Bereiche geteilt, auf die mit verschiedenen Geschwindigkeiten zugegriffen wird. Im Fall eines Fehlzugriffs im ersten Cache-Bereich wird im zweiten Cache-Bereich gesucht. Falls das Speicherwort dort gefunden wird, so spricht man von einem Pseudo-Treffer.

   Das Verfahren eignet sich nicht gut für Primär-Cache-Speicher, da ein Cache-Zugriff, der ein oder zwei Takte benötigt, nur schwer in die Befehls-Pipeline eingebaut werden kann. Die Technik eignet sich jedoch für Sekundär-Cache-Speicher, da dann schon ein Cache-Fehlzugriff in der Prozessor-Pipeline ausgelöst wurde und sich nur die Zahl der Wartetakte erhöht. Die Technik wird bei den Sekundär-Cache-Speichern des MIPS R10000 und in ähnlicher Weise beim UltraSPARC angewandt.

5. *Vorabladen per Hardware* (*hardware-prefetching*): Die Hardware lädt nach einem bestimmten Schema Speicherwörter spekulativ in den Cache-Speicher, ohne dass diese durch einen Fehlzugriff angefordert wurden. Spekulatives Laden benötigt, wie alle Arten von Spekulation, freie Ressourcen als Voraussetzung. Vorabladen ergibt nur dann Sinn, wenn zusätzliche Speicherbandbreite zur Verfügung steht, die durch eine normale Programmausführung nicht ausgenutzt wird. Die einfachste Form des spekulativen Vorabladens ist es, bei einem Cache-Fehlzugriff nicht einen, sondern gleich noch den darauf folgenden Cache-Block in den Cache-Speicher zu laden. Das geschieht bei einem Code-Cache-Fehlzugriff des Alpha-21064-Prozessors, der den zweiten Cache-Block allerdings in einen *Stream Buffer* genannten Pufferspeicher schreibt. Beim nächsten Cache-Fehlzugriff wird dann auf den Stream Buffer zugegriffen. Das Vorabladen per Hardware wird von vielen heutigen Prozessoren unterstützt. Beispielsweise erkennt der Pentium-4-Prozessor ständig wiederkehrende Zugriffsmuster und führt ein Vorabladen per Hardware durch seine Hardware-Data-Prefetch-Einheit durch.

6. *Vorabladen per Software* (*software-data-prefetching*): Die Software lädt durch spezielle Vorabladebefehle (*cache-prefetch, cache-touch*) Daten spekulativ in den Daten-Cache-Speicher. Das Verfahren wird von vielen Prozessorarchitekturen unterstützt, wie beispielsweise bei den PowerPC-, SPARC-V9-, MIPS IV- und IA-64-Architekturen. Die Vorabladebefehle stellen eine Form der Software-Spekulation dar und erzeugen keine Ausnahmen. Zu beachten ist weiterhin, dass die Vorabladebefehle Befehle sind, die zusätzlich zum zugehörigen Ladebefehl in der Pipeline ausgeführt werden. Voraussetzung ist deshalb wiederum, dass ansonsten ungenutzte Ressourcen zur Verfügung stehen. Das betrifft neben der Speicherbandbreite auch die Befehlslade-, Zuordnungs- und Ausführungsbandbreite des Prozessors. Für heutige Superskalarprozessoren stellt die Verwendung von Vorabladebefehlen durch den Compiler neben der Sprungspekulation eine weitere Methode dar, um die Prozessorressourcen besser auszulasten.

7. *Compile-Optimierungen*: Die Befehle und Daten werden vom Compiler so im Speicher angeordnet, dass Konflikte möglichst vermieden werden und die Kapazität besser ausgenutzt wird. Ziel ist es, dass möglichst nur Speicherwörter, die auch benötigt werden, in den Cache-Speicher geladen werden. Durch Hintereinanderanordnung von Daten, auf die nacheinander zugegriffen wird, lässt sich auch die Anzahl der Fehlzugriffe beim Erstzugriff verringern. Mögliche Techniken dafür sind das Zusammenfassen von Array-Datenstrukturen und die Manipulation der Schleifenanordnung: Beispiele dafür sind das Vertauschen geschachtelter Schleifen oder die

Verschmelzung von Schleifen, um auf die Daten in der gleichen Ordnung zuzugreifen, in der sie im Speicher stehen.

3.6.2 Verringern des Fehlzugriffsaufwandes

Es gibt verschiedene Techniken, um den **Fehlzugriffsaufwand** zu verringern. Die Grundidee ist meist, dem Prozessor das zu ladende Speicherwort so schnell als möglich zur Verfügung zu stellen.

1. *Ladezugriffe vor Speicherzugriffen beim Cache-Fehlzugriff*: Im Falle einer Durchschreibestrategie werden die in den Speicher zu schreibenden Cache-Blöcke in einem Schreibpuffer zwischengespeichert. Dabei kann das Laden neuer Cache-Blöcke vor dem Speichern von Cache-Blöcken ausgeführt werden, um die Wartezeit des Prozessors auf die zu ladenden Speicherwörter zu verringern. Um sicherzugehen, dass bei einem vorgezogenen Laden keine Datenabhängigkeit zu einem der passierten Speicherwörter verletzt wird, müssen die Ladeadressen mit den Adressen der zu speichernden Daten verglichen werden. Im Falle einer Übereinstimmung muss die Ladeoperation ihren Wert aus dem Schreibpuffer und nicht aus der nächst tieferen Speicherhierarchieebene erhalten.

   Auch bei Rückschreib-Cache-Speichern wird bei einem Cache-Fehlzugriff der verdrängte „Dirty"-Block zurückgespeichert. Normalerweise würde das Rückschreiben vor dem Holen des neuen Cache-Blocks erfolgen. Um den zu ladenden Wert dem Prozessor so schnell als möglich zur Verfügung stellen zu können, wird der verdrängte Cache-Block ebenfalls im Schreibpuffer zwischengespeichert und erst der Ladezugriff vor dem anschließenden Rückschreibzugriff auf dem Speicher durchgeführt.

2. *Zeitkritisches Speicherwort zuerst* (*critical-word-first*): Das Laden eines Cache-Blocks in den Cache kann mehrere Takte dauern. Der Prozessor benötigt jedoch nur das Speicherwort dringend, dessen Ladezugriff den Fehlzugriff ausgelöst hat. Sobald dieses Speicherwort im Cache ankommt, wird es sofort dem Prozessor zur Verfügung gestellt und dieser kann weiterarbeiten (*early restart*). Darüber hinaus kann das angeforderte Speicherwort auch als erstes vom Speicher zum Cache übertragen werden (*critical word first, wrapped fetch*) und anschließend werden die nicht zeitkritischen, restlichen Speicherwörter des Cache-Blocks übertragen. Beide Verfahren sind nur bei einem großen Cache-Block effizient und bringen keinen Vorteil, wenn auch sofort auf die benachbarten Speicherwörter zugegriffen wird.

3. *Nichtblockierender Cache-Speicher* (*non-blocking cache*, *lockup-free cache*): Im Falle eines Cache-Fehlzugriffs können nachfolgende Ladeoperationen, die nicht denselben Cache-Block benötigen, auf dem Cache-Speicher durchgeführt werden, ohne dass auf die Ausführung der den Fehlzugriff auslösenden Lade- oder Speicheroperation gewartet werden muss. Ein Fehlzugriff blockiert damit keine nachfolgenden Ladebefehle. Konsequenterweise kann das Verfahren auch auf mehrere Cache-Fehlzugriffe ausgedehnt werden, also Ladezugriffe bei mehreren ausstehenden Cache-Fehlzugriffen erlauben. Die Komplexität der Cache-Speicherverwaltung wird dadurch erheblich erhöht, da mehrere ausstehende Speicherzugriffe verwaltet werden müssen. Dennoch wird das Verfahren heute allgemein angewandt. Bereits der Pentium-Pro-Prozessor erlaubte beispielsweise vier ausstehende Speicherzugriffe. Probleme ergeben sich jedoch für die Speicherkonsistenz, falls Prozessoren mit nicht-blockierenden Cache-Speichern in Multiprozessorsystemen eingesetzt werden.

4. *Einsatz eines Sekundär-Cache-Speichers* (*L2-Cache*): Wie schon erwähnt, befinden sich auf dem Prozessor-Chip getrennte Code- und Daten-Cache-Speicher als Primär-Caches. Auf diese kann mit der Taktrate des Prozessors zugegriffen werden. Sie sind deshalb mit 8 bis 64 KB verhältnismäßig klein. Durch die Trennung erreicht man einen parallelen Zugriff auf Programm und Daten, wodurch die hohen Anforderungen der heutigen Superskalar-Prozessoren an die Speicherbandbreiten erfüllt werden können. Dabei wird auf dem Chip eine Harvard-Architektur (KE2) realisiert, bei der Programm und Daten in getrennten Speichern abgelegt werden. Falls

nur ein einziger Cache-Speicher für Code und Daten vorhanden ist, spricht man dagegen von einem einheitlichen Cache-Speicher.

Neben den Primär-Caches befindet sich häufig ein weiterer, heute etwa 256 KB bis 512 KB großer Sekundär-Cache auf dem Prozessor-Chip. Dieser sorgt dafür, dass bei einem Fehlzugriff auf den Primär-Cache die Daten schnell nachgeladen werden können. Beim Austausch von Daten zwischen Hauptspeicher und Sekundär-Cache bzw. Sekundär- und Primär-Cache werden komplette Blöcke oder Teilblöcke in einer Aktion (*Burst*) übertragen.
Beim Vorhandensein eines Sekundär-Cache-Speichers wird der Primär-Cache meist als Durchschreibespeicher betrieben, so dass alle Daten im Primär-Cache auch im Sekundär-Cache stehen (Inklusionsprinzip). Der Sekundär-Cache arbeitet dagegen in der Regel mit dem Rückschreibeverfahren.
Der Sekundär-Cache kann als sog. *Look-Aside-Cache* parallel zum Hauptspeicher an den Bus angeschlossen werden. Der Prozessor kann so direkt auf Cache und Hauptspeicher zugreifen. Allerdings ist dies nur bei Ein-Prozessorsystemen sinnvoll, da es bei jedem Speicherzugriff zu einem Buszugriff und damit zu einer erheblichen Zusatzbelastung des Speicherbusses kommt.

3.6.3 Verringern der Cache-Zugriffszeit bei einem Treffer

Als weitere Cache-Optimierung kann die Zugriffszeit beim Cache-Treffer verringert werden.

1. *Primär-Cache-Speicher auf dem Prozessor-Chip*: Wie schon erwähnt, sollte der Primär-Cache-Speicher auf dem Prozessor-Chip untergebracht und klein sowie einfach genug strukturiert sein, um mit der Geschwindigkeit der Prozessortaktrate zugreifen zu können. Bereits der Alpha-21164-Prozessor hatte beispielsweise je 8 KByte Primär-Caches für Befehle und Daten und einen Sekundär-Cache-Speicher von 96 KBytes auf dem Prozessor-Chip. Ein direkt abgebildeter Cache ist im Zugriff schneller als ein mehrfach satzassoziativer Cache. Jedoch finden sich heute bis zu achtfach satzassoziative Daten-Caches als Primär-Caches auf dem Prozessor-Chip. Die Primär-Cache-Größen variieren von 8 KB beim Pentium-4- über 16 KB beim Pentium-III- bis zu 64 KB beim Athlon-Prozessor.

2. *Virtueller Cache*: Die Zugriffszeit kann auch durch ein Vermeiden von Adressübersetzungen erhöht werden. Der Cache kann mit der virtuellen Adresse (virtueller Cache) statt mit der physikalischen Adresse (physikalischer Cache) angesprochen werden. Je nachdem, ob die Adressumsetzung durch eine Speicherverwaltungseinheit (MMU) vor oder hinter dem Cache vorgenommen wird, spricht man von einem **virtuell** oder **physikalisch adressierten Cache-Speicher**.

   Der virtuell adressierte Cache hat den Vorteil, dass auf die Daten schneller zugegriffen werden kann, da bei einem Cache-Treffer keine Adress-umsetzung notwendig ist. Jedoch muss bei einem virtuellen Cache bei jedem Prozesswechsel ein Löschen der Cache-Inhalte erfolgen, da ansonsten falsche Treffer geschehen. Die zusätzlichen Kosten sind dann die Zeit für das Leeren des Caches plus die Zeit für die Erstbelegungsfehlzugriffe, da nach einem Prozesswechsel immer mit einem leeren Cache gestartet wird.

3. *Pipeline-Verarbeitung für Schreibzugriffe auf den Cache*: Die Tag-Prüfung eines Cache-Zugriffs und die Cache-Modifikation durch den vorherigen Schreibzugriff können als separate Pipeline-Stufen implementiert und überlappt zueinander ausgeführt werden. Durch eine solche zusätzliche Pipeline-Verarbeitung wird die Zugriffsgeschwindigkeit bei hohen Prozessortaktraten erhöht.

3.7 Cache-Kohärenz und Konsistenz

Falls in einem Multiprozessorsystem mehrere Prozessoren mit jeweils eigenen Cache-Speichern unabhängig voneinander auf Speicherwörter des Hauptspeichers zugreifen können, entstehen Gültigkeitsprobleme. Mehrere Kopien des gleichen Speicherwortes können sich in den Cache-Speichern der verschiedenen Prozessoren befinden und müssen miteinander in Einklang gebracht werden. Eine Cache-Speicherverwaltung heißt **Cache-kohärent**, falls ein Lesezugriff immer den Wert des zeitlich letzten Schreibzugriffs auf das entsprechende Speicherwort liefert. **Kohärenz** bedeutet also das korrekte Voranschreiten des Systemzustands durch ein abgestimmtes Zusammenwirken der Einzelzustände. Im Zusammenhang mit einem Cache-Speicher muss das System dafür sorgen, dass immer die aktuellen und nicht die veralteten Daten gelesen werden.

Ein System ist **konsistent**, wenn alle Kopien eines Speicherwortes im Hauptspeicher und den verschiedenen Cache-Speichern *identisch* sind. Dadurch ist auch die Kohärenz sichergestellt. Eine Inkonsistenz zwischen Cache-Speicher und Hauptspeicher entsteht dann, wenn ein Speicherwort nur im Cache-Speicher und nicht gleichzeitig im Hauptspeicher verändert wird. Solche Inkonsistenzen entstehen bei Anwendung des Rückschreibeverfahrens im Gegensatz zur Anwendung des Durchschreibeverfahrens. Um alle Kopien eines Speicherwortes immer konsistent zu halten, müsste ein hoher Aufwand getrieben werden, der zu einer Leistungseinbuße führen würde. Es kann nun im begrenzten Umfang die Inkonsistenz der Daten zugelassen werden, wenn ein geeignetes **Cache-Kohärenzprotokoll** dafür sorgt, dass die Cache-Kohärenz gewährleistet ist. Das Protokoll muss sicherstellen, dass immer die aktuellen und nicht die veralteten Daten gelesen werden. Dabei gibt es zwei prinzipielle Ansätze für Cache-Kohärenzprotokoll:

- *Write-Update-Protokoll*: Beim Verändern einer Kopie in einem Cache-Speicher müssen alle Kopien in anderen Cache-Speichern ebenfalls verändert werden, wobei die Aktualisierung auch verzögert (spätestens beim Zugriff) erfolgen kann.
- *Write-Invalidate-Protokoll*: Vor dem Verändern einer Kopie in einem Cache-Speicher müssen alle Kopien in anderen Cache-Speichern für „ungültig“ erklärt werden.

Üblicherweise wird bei symmetrischen Multiprozessoren ein *Write-invalidate*-Cache-Kohärenzprotokoll mit Rückschreibeverfahren angewandt.

3.7.1 Busschnüffeln und MESI-Protokoll

Weiterhin wird bei symmetrischen Multiprozessoren, bei denen mehrere Prozessoren mit lokalen Cache-Speichern über einen Systembus an einen gemeinsamen Hauptspeicher angeschlossen sind, das sog. **Bus-Schnüffeln** (*Bus-Snooping*) angewandt. Die Schnüffel-Logik jedes Prozessors „hört“ am Speicherbus die Adressen mit, die von den anderen Prozessoren auf den Bus gelegt werden. Die Adressen auf dem Bus werden mit den Adressen der Cache-Blöcke im Cache-Speicher verglichen. Bei Übereinstimmung der auf dem Systembus erschnüffelten Adresse mit einer der Adressen der Cache-Blöcke im Cache-Speicher geschieht Folgendes:

- Im Fall eines erschnüffelten Schreibzugriffs wird der im Cache gespeicherte Cache-Block für „ungültig“ erklärt, sofern auf den Cache-Block nur lesend zugegriffen wurde.
- Wenn ein Lese- oder Schreibzugriff erschnüffelt wird und die Datenkopie im Cache-Speicher verändert wurde, dann unterbricht die Schnüffel-Logik die Bustransaktion. Die „schnüffelnde“ Hardware-Logik übernimmt den Bus und schreibt den betreffenden Cache-Block in den Hauptspeicher. Dann wird die ursprüngliche Bustransaktion erneut durchgeführt. Alternativ könnte auch ein direkter Cache-zu-Cache-Transfer durchgeführt werden (*Snarfing* genannt).

Als Cache-Kohärenzprotokoll in Zusammenarbeit mit dem Bus-Schnüffeln hat sich das sog. **MESI-Protokoll** durchgesetzt (Abb. 3.17). Dieses ist als Write-Invalidate-Kohärenzprotokoll einzuordnen.

Das MESI-Protokoll[1] ordnet jedem Cache-Block einen der folgenden vier Zustände zu:

- *Exclusive-Modified*: Der Cache-Block wurde durch einen Schreibzugriff geändert und befindet sich ausschließlich in diesem Cache.
- *Exclusive-Unmodified*: Der Cache-Block wurde für einen Lesezugriff übertragen und befindet sich nur in diesem Cache.
- *Shared-Unmodified*: Kopien des Cache-Blocks befinden sich für Lesezugriffe in mehr als einem Cache.
- *Invalid*: Der Cache-Block ist ungültig.

Bei der Erläuterung der Funktionsweise des MESI-Protokolls beginnen wir mit einem Lesezugriff auf ein Datenwort in einem Cache-Block, der sich im Zustand *Invalid* befindet. Bei einem solchen Lesezugriff wird ein Cache-Fehlzugriff ausgelöst, und der Cache-Block, der das Datenwort enthält, wird aus dem Hauptspeicher in den Cache-Speicher des lesenden Prozessors übertragen.

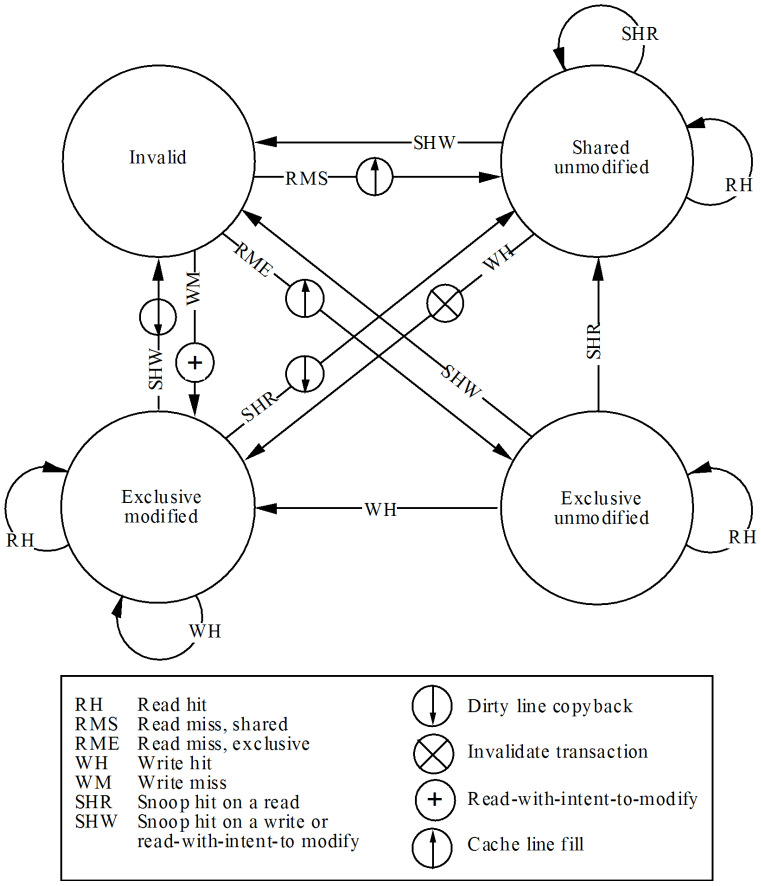


Abbildung 3.17: MESI-Cache-Kohärenzprotokoll

Je nachdem, ob der Cache-Block bereits in einem anderen Cache-Speicher steht, muss man folgende Fälle unterscheiden:

- Der Cache-Block stand in keinem Cache-Speicher eines anderen Prozessors: Dann wird ein RME (*Read miss exclusive*) durchgeführt und der übertragene Cache-Block erhält das Attribut *Exclusive unmodified*. Dieses Attribut bleibt so lange bestehen, wie sich der Cache-Block in keinem anderen Cache-Speicher befindet und nur Lesezugriffe des einen Prozessors stattfinden (RH, *Read-Hit*).

[1] Das Akronym „MESI“ wurde aus Anfangsbuchstaben der Zustände *exclusive Modified, Exclusive-Unmodified, Shared-Unmodified* und *Invalid* gebildet.

- Falls beim Lesezugriff erkannt wird, dass der Cache-Block bereits in einem (oder mehreren) anderen Cache-Speichern steht und dort den Zustand *Exclusive-Unmodified* (bzw. *Shared-Unmodified*) besitzt, wird das Attribut auf *Shared-Unmodified* gesetzt (RMS, *Read-Miss-Shared*) und der andere Cache-Speicher ändert sein Attribut ebenfalls auf *Shared-Unmodified*. Das geschieht durch die Schnüffelaktion SHR (*Snoop-Hit on a Read*).
- Falls beim Lesezugriff ein anderer Cache-Speicher den Cache-Block als *Exclusive-Modified* besitzt, so erschnüffelt dieser die Adresse auf dem Bus (SHR, *Snoop-Hit on a Read*), er unterbricht die Bustransaktion, schreibt den Cache-Block in den Hauptspeicher zurück (*Dirty-Block-Copyback*) und setzt in seinem Cache-Speicher den Zustand auf *Shared-Unmodified*. Danach wird die Leseaktion auf dem Bus wiederholt.

Falls ein Prozessor ein Datenwort in seinen Cache-Speicher schreibt, so kann dies nur im Zustand *Exclusive-Modified* des betreffenden Cache-Blocks ohne Zusatzaktion geschehen (WH, *Write-Hit*). Der geänderte Cache-Block wird wegen des Rückschreibeverfahrens nicht sofort in den Hauptspeicher zurückgeschrieben. Ist der vom Schreibzugriff betroffene Cache-Block nicht im Zustand *Exclusive-Modified*, so wird die Adresse des Code-Blocks auf den Bus gegeben. Dabei muss man folgende Fälle unterscheiden:

- Der Cache-Block war noch nicht im Cache vorhanden (Cache-Fehlzugriff, Zustand *Invalid*): Es wird ein *Read-With-Intent-to-Modify* auf den Bus gegeben, alle anderen Cache-Speicher erschnüffeln die Transaktion (SHW, *Snoop-Hit on a Write*) und setzen ihren Cache-Blockzustand auf *Invalid*, falls dieser vorher *Shared-Unmodified* oder *Exclusive-Unmodified* war. In diesen Cache-Speichern kann nun nicht mehr lesend auf den Cache-Block zugegriffen werden, ohne dass ebenfalls ein Cache-Fehlzugriff ausgelöst wird. Der Cache-Block wird aus dem Hauptspeicher übertragen und erhält das Attribut *Exclusive-Modified*. War der Cache-Block jedoch in einem anderen Cache-Speicher mit dem Attribut *Exclusive-Modified*, so geschieht (wie im obigen Fall eines Lesezugriffs) ein Unterbrechen der Bustransaktion und Rückschreiben des Cache-Blocks in den Hauptspeicher.
- Der Cache-Block ist im Cache-Speicher vorhanden, aber im Zustand *Exclusive-Unmodified*: Hier genügt eine Änderung des Attributs auf *Exclusive-Modified*.
- Der Cache-Block ist im Cache-Speicher vorhanden, aber im Zustand *Shared-Unmodified*: Hier müssen zunächst wieder über eine *Invalidate*-Transaktion der oder die anderen betroffenen Cache-Speicher benachrichtigt werden, dass ein Schreibzugriff durchgeführt wird, so dass diese ihre Kopien des Cache-Blocks invalidieren können.

Ein Cache-Block wird erst dann in den Hauptspeicher zurückgeschrieben, wenn er gemäß der Verdrängungsstrategie ersetzt wird oder einer der anderen Prozessoren darauf zugreifen will.

3.7.2 Speicherkonsistenz

Die Lade-/Speichereinheit eines Prozessors führt alle Datentransfer-Befehle zwischen den Registern und dem Daten-Cache-Speicher durch. Cache-Kohärenz wird erst dann wirksam, wenn ein Lade- oder Speicherzugriff durch die Lade-/Speichereinheit des Prozessors ausgeführt wird, also ein Zugriff auf den Cache-Speicher geschieht. Cache-Kohärenz sagt nichts über mögliche Umordnungen der Lade- und Speicherbefehle *innerhalb* der Lade-/Speichereinheit aus. Heutige Mikroprozessoren führen jedoch die Lade- und Speicherbefehle nicht mehr unbedingt in der Reihenfolge aus, wie sie vom Programm her vorgeschrieben sind.

Die Prinzipien der vorgezogenen Ladezugriffe und des nichtblockierenden Cache-Speichers steigern die Verarbeitungsgeschwindigkeit des Prozessors, da neue Daten so schnell wie möglich in die Register geladen werden, damit nachfolgende Verarbeitungsbefehle ausgeführt werden können – das Rückspeichern erscheint demgegenüber nicht so zeitkritisch. Wesentlich ist dabei bei beiden Prinzipien, dass aus Speichersicht die Programmordnung nicht mehr eingehalten wird. Unter der angegebenen Randbedingung – dem lokalen Adressabgleich der betroffenen Datenwerte – bleibt jedoch die

Ausführung außerhalb der Programmreihenfolge ohne Auswirkung auf das Ergebnis, sofern nur ein einzelner Prozessor beteiligt ist.

Dies gilt jedoch nicht mehr bei einer parallelen Programmausführung auf einem Multiprozessor. Bei speichergekoppelten Multiprozessoren können im Prinzip zu jedem Zeitpunkt mehrere Zugriffe auf denselben Speicher erfolgen. Bei symmetrischen Multiprozessoren können trotz Cache-Kohärenz durch Verwendung von vorgezogenen Ladezugriffen oder durch nichtblockierende Cache-Speicher die Speicherzugriffsoperationen in anderer Reihenfolge als durch das Programm definiert am Speicher wirksam werden.

Aus Sicht des Programmierers ist weniger die implementierungstechnische Seite der Cache-Kohärenz interessant, als das zugrundeliegende **Konsistenzmodell**, das etwas über die zu erwartende Ordnung der Speicherzugriffe durch parallel arbeitende Prozessoren aussagt. Genauer gesagt:

*Ein **Konsistenzmodell** spezifiziert die Reihenfolge, in der Speicherzugriffe eines Prozesses von anderen Prozessen gesehen werden.*

Im Normalfall setzt der Programmierer die **sequenzielle Konsistenz** voraus, die jedoch zu starken Einschränkungen bei der Implementierung führt. Ein Multiprozessorsystem ist **sequenziell konsistent**, wenn das Ergebnis einer beliebigen Berechnung dasselbe ist, als wenn die Operationen aller Prozessoren auf einem Einprozessorsystem in einer sequenziellen Ordnung ausgeführt würden. Dabei ist die Ordnung der von den Prozessoren ausgeführten Operationen die des jeweiligen Programms. Insbesondere verbietet die sequenzielle Konsistenz die Prinzipien der vorgezogenen Ladezugriffe und des nichtblockierenden Cache-Speichers.

Die sequenzielle Konsistenz stellt zwar die korrekte Ordnung der Speicherzugriffe sicher, nicht jedoch die Korrektheit der Zugriffe auf gemeinsam benutzte Datenobjekte durch parallele Kontrollfäden. Letzteres ist durch geeignete Synchronisationen des Datenzugriffs oder des Prozessablaufs sicherzustellen, was nach dem heutigen Stand Sache des Programmierers ist und in Zukunft vielleicht Sache eines parallelisierenden Compilers sein wird.

Wenn sich der Programmierer über die Auswirkungen einer möglichen Verletzung der Zugriffsordnung nicht im Klaren ist, so wird er solche Verletzungen unter allen Umständen zu vermeiden suchen. Das heißt, er wird vom System die Einhaltung der sequenziellen Konsistenz fordern und dafür dann den dadurch entstehenden hohen, systemimmanenten Synchronisationsaufwand in Kauf nehmen müssen.

Weiß der Programmierer, dass in gewissen Phasen der Programmausführung eine Verletzung der Zugriffsordnung toleriert werden kann, so kann er die Forderung nach einer sequenziellen Konsistenz aufgeben und sich statt dessen mit einer **schwachen Konsistenz** (*weak consistency*) begnügen, um die Effizienz der Programmausführung zu steigern. Schwache Konsistenz heißt, dass die Konsistenz des Speicherzugriffs nicht mehr zu allen Zeiten gewährleistet ist, sondern nur zu bestimmten vom Programmierer in das Programm eingesetzten Synchronisationspunkten. Praktisch bedeutet dies die Einführung von *kritischen Bereichen*, innerhalb derer Inkonsistenzen zugelassen werden. Die Synchronisationspunkte sind dabei die Ein- oder Austrittspunkte der kritischen Bereiche.

Aufgabe 3.20 *Cache-Kohärenz bei einem Multiprozessorsystem*
Gegeben sei ein speichergekoppeltes Multiprozessorsystem mit zwei Prozessoren, die über einen Bus mit einem globalen Speicher verbunden sind. Zur Wahrung der Cache-Kohärenz werde das MESI-Protokoll verwendet. Die Caches der beiden Prozessoren haben je eine Größe von drei Cache-Blocks, die genau ein Speicherwort aufnehmen können. Die Caches werden vom niedrigsten Cache-Block aufwärts gefüllt, falls noch freie Blöcke zur Verfügung stehen. Im Falle eines voll besetzten Caches werden sie nach der Strategie LRU (least recently used) überschrieben. Ergänzen Sie die folgende Tabelle, wobei die Buchstaben die Zustände: M=Exclusive-Modified, E=Exclusive-Unmodified, S=Shared-Unmodified, I=Invalid repräsentieren. Die Zahlen hinter den Aktionen und Zuständen bezeichnen Speicheradressen.

Prozessor	**Aktion**	**Cache 1 Block 1**	**Cache 1 Block 2**	**Cache 1 Block 3**	**Cache 2 Block 1**	**Cache 2 Block 2**	**Cache 2 Block 3**
-	-	E/8	E/12	I/-	E/6	I/-	I/-
2	READ 10						
1	WRITE 8						
1	READ 10						
2	READ 8						
1	WRITE 8						
1	WRITE 8						
1	READ 18						
2	WRITE 10						
2	WRITE 18						

◇

Aufgabe 3.21 *Wiederholungsfragen zu Speicherverwaltung und Cache*

a) Geben Sie die Speicherhierarchie für heutige PCs geordnet nach absteigenden Zugriffsgeschwindigkeiten und aufsteigenden Speicherkapazitäten an.

b) Was ist die Voraussetzung für eine effiziente Speicherhierarchie?

c) Wozu dient ein Translation-Look-Aside-Buffer (TLB)?

d) Welcher technologische Unterschied besteht zwischen Cache und Hauptspeicher?

e) Wodurch unterscheidet sich die Rückschreibe- von der Durchschreibe-Strategie?

f) Welche Vor- und Nachteile weisen virtuell bzw. physikalisch adressierte Cache-Speicher auf?

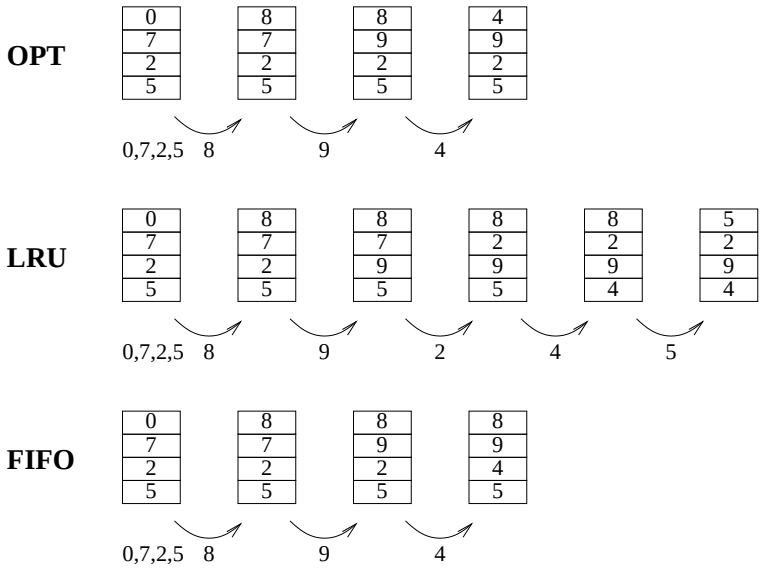
g) Was bedeutet Cache-Kohärenz? Nennen Sie ein Cache-Kohärenzprotokoll?

h) Welche Vor- und Nachteile hat das Durchschreib-Verfahren (Write-Through-Policy) gegenüber dem Rückschreibverfahren (Write-Back-Policy) bei einem Cache-Speicher?

◇

3.8 Lösungen zu den Selbsttestaufgaben

Aufgabe 3.1



Demnach entstehen folgende Seitenfehler:

OPT: 4+3=7 **LRU:** 4+5=9 **FIFO:** 4+3=7

Aufgabe 3.2

Maßnahmen zur Reduzierung von Seitenfehlern

a) Vergrößerung des Hauptspeichers

Lösung: Ja, die Vergrößerung des Hauptspeichers führt dazu dass sich mehr Seiten im HSP befinden und vergrößert damit die Wahrscheinlichkeit, dass sich eine aktuelle gesuchte Seite im HSP befindet.

b) Vergrößerung des Sekundärspeichers

Lösung: Nein, der Prozessor greift immer auf die virtuellen Adressen zu, die den HSP und einen geringen, begrenzten Teil einer Festplatte, den sog. Swap-Bereich, belegen. Aber nur ein Fehlzugriff auf den HSP erzeugt einen Seitenfehler.

c) Verbesserung der Verdrängungsstrategie bei der Auslagerung von Seiten

Lösung: Ja, eine bessere Verdrängungsstrategie sorgt dafür, dass die Wahrscheinlichkeit dafür, dass eine demnächst benötigte Seite sich im HSP befindet, größer wird. Damit sinkt die Wahrscheinlichkeit für einen Seitenfehler.

Aufgabe 3.3

Die Darstellung in Abb. 3.7 basiert auf einem Verfahren, bei dem zu jedem Segment lediglich eine Anfangsadresse im virtuellen Adressraum gespeichert wird. Durch Addition des Offsets kann so leicht die virtuelle Adresse berechnet werden, allerdings benötigt das Segment einen zusammenhängenden Block.

Alternativ könnte für jedes Segment eine eigene Seitentabelle zum Transfer in den virtuellen Adressraum verwendet werden. Dann könnten die einzelnen Seiten frei im virtuellen Adressraum verteilt werden, allerdings steigt der Aufwand beim Übersetzen der Adresse. Außerdem erlaubt die Kombination von Segmentierung und Paging ein effizientes Verschieben der Blöcke, sodass der Mehraufwand von zusätzlichen Seitentabellen meist nicht gerechtfertigt ist.

Aufgabe 3.4

Wir haben den Durchlauf der äußeren Schleife mit der Nummer $i = \texttt{0FFE}$ abgeschlossen und dabei 6 Seiten ein- bzw. ausgelagert.

Zeitpunkt	Seitenrahmen 2	Seitenrahmen 3
Beginn	2000 – 2FFF (D29E6000 - D29E6FFF)	3000 – 3FFF (D29E7000 - D29E7FFF)
1. Austausch im 1. Durchlauf $i = 0$	2000 – 2FFF (D29E6000 - D29E6FFF)	0000 – 0FFF (D29E4000 - D29E4FFF)
2. Austausch im Durchlauf mit $i = \texttt{0FFE}$, $j = 2$, $i + j = \texttt{1000}$	2000 – 2FFF (D29E6000 - D29E6FFF)	3000 – 3FFF (D29E7000 - D29E7FFF)
3. Austausch im Durchlauf mit $i = \texttt{0FFE}$, $j = 2$, $i + j = \texttt{1000}$	0000 – 0FFF (D29E4000 - D29E4FFF)	3000 – 3FFF (D29E7000 - D29E7FFF)

Die Seitentabelle hat den folgenden Stand erreicht:

Seitenrahmen	logische Adresse		virtuelle
	dezimal	hexadezimal	Adresse (32 Bit)
0	24576 – 28671	6000 – 6FFF	3A17C000 - 3A17CFFF
1	28672 – 32768	7000 – 7FFF	51F60000 - 51F60FFF
2	00000 – 04095	0000 – 0FFF	D29E4000 - D29E4FFF
3	12288 – 16383	3000 – 3FFF	D29E7000 - D29E7FFF

Im nächsten Durchlauf der äußeren Schleife mit der Nummer $i = \texttt{0FFF}$ findet nun sofort wieder ein Austausch statt, da im ersten Durchlauf der inneren Schleife auf die Seite mit den logischen Adressen 2000 – 2FFF zugegriffen wird. Dabei wird nach LRU im Seitenrahmen 3 die Seite mit den logischen Adressen 3000 – 3FFF (D29E7000 - D29E7FFF) gegen die Seite 2000 – 2FFF (D29E6000 - D29E6FFF) zurück getauscht (4. Austausch). Im zweiten Durchlauf der inneren Schleife wird auf die Seite mit den logischen Adressen 3000 – 3FFF zugegriffen, sodass nach LRU im Seitenrahmen 2 die Seite mit den logischen Adressen 0000 – 0FFF (D29E4000 - D29E4FFF) gegen die Seite 3000 – 3FFF (D29E7000 - D29E7FFF) getauscht wird (5. Austausch).

Anschließend wird im weiteren Verlauf dieser äußeren Schleife auf $y(\texttt{0FFF})$ zugegriffen, was dazu führt, dass nach LRU die Seite mit den logischen Adressen 2000 – 2FFF (D29E6000 - D29E6FFF) gegen die Seite mit den logischen Adressen 0000 – 0FFF (D29E4000 - D29E4FFF) im Seitenrahmen 3 ausgetauscht wird (6. Austausch).

Zeitpunkt	Seitenrahmen 2	Seitenrahmen 3
Beginn	2000 – 2FFF (D29E6000 - D29E6FFF)	3000 – 3FFF (D29E7000 - D29E7FFF)
1. Austausch im 1. Durchlauf $i = 0$	2000 – 2FFF (D29E6000 - D29E6FFF)	0000 – 0FFF (D29E4000 - D29E4FFF)
2. Austausch im Durchlauf mit $i = \texttt{0FFE}$, $j = 2$, $i + j = \texttt{1000}$	2000 – 2FFF (D29E6000 - D29E6FFF)	3000 – 3FFF (D29E7000 - D29E7FFF)
3. Austausch im Durchlauf mit $i = \texttt{0FFE}$, $j = 2$, $i + j = \texttt{1000}$	0000 – 0FFF (D29E4000 - D29E4FFF)	3000 – 3FFF (D29E7000 - D29E7FFF)
4. Austausch im Durchlauf mit $i = \texttt{0FFF}$, $j = 0$, $i + j = \texttt{0FFF}$	0000 – 0FFF (D29E4000 - D29E4FFF)	2000 – 2FFF (D29E6000 - D29E6FFF)
5. Austausch im Durchlauf mit $i = \texttt{0FFF}$, $j = 1$, $i + j = \texttt{1000}$	3000 – 3FFF (D29E7000 - D29E7FFF)	2000 – 2FFF (D29E6000 - D29E6FFF)
6. Austausch im Durchlauf mit $i = \texttt{0FFF}$, $j = 2$, $i + j = \texttt{1001}$	3000 – 3FFF (D29E7000 - D29E7FFF)	0000 – 0FFF (D29E4000 - D29E4FFF)
7. Austausch im Durchlauf mit $i = \texttt{1000}$, $j = 2$, $i + j = \texttt{1002}$	3000 – 3FFF (D29E7000 - D29E7FFF)	1000 – 1FFF (D29E5000 - D29E5FFF)

Im nächsten Durchlauf der äußeren Schleife mit der Nummer $i = 1000$ wird in Anweisung 6 auf die Seite mit den logischen Adressen 1000 – 1FFF (`D29E5000 - D29E5FFF`) zugegriffen, sodass nach LRU im Seitenrahmen 3 die Seite mit den logischen Adressen 3000 – 3FFF (`D29E7000 - D29E7FFF`) gegen die Seite 1000 – 1FFF (`D29E5000 - D29E5FFF`) getauscht wird (7. Austausch).

Nun kann die äußere Schleife solange durchlaufen werden, bis im Durchlauf mit der hexadezimalen Nummer $i =$ `1FFE` in der inneren Schleife mit $j = 2$, $i + j = 2000$ auf die logische Adresse 4000 zugegriffen werden soll. Dabei müsste nach LRU im Seitenrahmen 2 die Seite mit den logischen Adressen 3000 – 3FFF (D29E7000 – D29E7FFF) 0000 – 0FFF (D29E4000 – D29E4FFF) gegen die Seite 4000 – 4FFF (D29E8000 – D29E8FFF) getauscht werden. Dieser Austausch wird aber nicht ausgeführt, da wegen der begrenzten Segmentgröße von 4000 (Hex) (Zugriffsverletzung) auf die virtuelle Adresse D29E8000 nicht mehr zugegriffen kann.

Bei Anwendung der LRU-Strategie werden also insgesamt 14 Seiten ein- bzw. ausgelagert. Die durchgeführten Aktionen können in einer Tabelle zusammengefasst werden:

	Programm-zeile	i	j	Aktion	Seiten-rahmen
1	6	`0000`	-	Seite `D29E7` auslagern, Seite `D29E4` anlegen	3
2	4	`0FFE`	2	Seite `D29E4` auslagern, Seite `D29E7` einlagern	3
3	6	`0FFE`	-	Seite `D29E6` auslagern, Seite `D29E4` einlagern	2
4	4	`0FFF`	0	Seite `D29E7` auslagern, Seite `D29E6` einlagern	3
5	4	`0FFF`	1	Seite `D29E4` auslagern, Seite `D29E7` einlagern	2
6	6	`0FFF`	-	Seite `D29E6` auslagern, Seite `D29E4` einlagern	3
7	6	`1000`	-	Seite `D29E4` auslagern, Seite `D29E5` einlagern	3
8	6	`1FFE`	2	Seite `D29E8` Zugriffsverletzung	

Aufgabe 3.5

Verbesserung der Seitenersetzungs-Strategie gegenüber LRU

	Programm-zeile	i	j	Aktion	Seiten-rahmen
1	6	`0000`	-	Seite `D29E7` auslagern, Seite `D29E4` anlegen	3
2	4	`0FFE`	2	Seite `D29E4` auslagern, Seite `D29E7` einlagern	3
3	6	`0FFE`	-	Seite `D29E6` auslagern, Seite `D29E4` einlagern	2
4	4	`0FFF`	0	Seite `D29E7` auslagern, Seite `D29E6` einlagern	3
5	4	`0FFF`	1	Seite `D29E4` auslagern, Seite `D29E7` einlagern	2
6	6	`0FFF`	-	Seite `D29E6` auslagern, Seite `D29E4` einlagern	3
7	6	`1000`	-	Seite `D29E4` auslagern, Seite `D29E5` einlagern	3
8	6	`1FFE`	2	Seite `D29E8` Zugriffsverletzung	

Wie man aus der Tabelle der Ein-/Auslagerungen für LRU erkennt, wird die Seite mit den logischen Adressen 0000 – 0FFF als auch die Seite mit den logischen Adressen 3000 – 3FFF im Segment DS mehrmals verdrängt und anschließend wieder eingelagert.
Um die LRU-Strategie zu verbessern, müsste die Speicherverwaltung bei der Festlegung der auszulagernden Seiten mehr in die Zukunft anstatt in die Vergangenheit schauen. Zu Beginn steht die Seite mit den logischen Adressen 2000 – 2FFF im Seitenrahmen 2 und die Seite mit den logischen Adressen 3000 – 3FFF im Seitenrahmen 3. Wir betrachten den Ablauf der Zugriffe noch einmal im Überblick:

$i = 0, \ldots,$ `0FFD`: je Durchlauf der äußeren Schleife gibt es
j=0,1,2 $\Longrightarrow$ 3 Zugriffe auf Seite 2, 1 Zugriff auf Seite 0
Austausch nach LRU: Seite 3 in Rahmen 3 durch Seite 0 ersetzen
optimaler Austausch: Seite 3 in Rahmen 3 durch Seite 0 ersetzen

$i = $ `0FFE`:
j=0,1 $\Longrightarrow$ 2 Zugriffe auf Seite 2, j=2 $\Longrightarrow$ 1 Zugriff auf Seite 3, 1 Zugriff auf Seite 0
Austausch nach LRU: zuerst wird die Seite 0 im Seitenrahmen 3 durch die Seite 3 ersetzt, dann die Seite 2 im Rahmen 2 durch Seite 0
optimaler Austausch: Seite 2 in Rahmen 2 durch Seite 3 ersetzen

$i = $ `0FFF`:
j=0 $\Longrightarrow$ 1 Zugriffe auf Seite 2, j=1,2 $\Longrightarrow$ 2 Zugriffe auf Seite 3, 1 Zugriff auf Seite 0
Austausch nach LRU: zuerst wird die Seite 3 im Rahmen 3 durch die Seite 2 ersetzt, dann die Seite 0 im Rahmen 2 durch Seite 3 und dann die Seite 2 im Rahmen 3 durch Seite 0
optimaler Austausch: erst Seite 3 in Rahmen 2 durch Seite 2 ersetzen, dann wieder die Seite 2 in Rahmen 2 durch Seite 3 ersetzen

$i = $ `1000`, ..., `1FFD`: je Durchlauf der äußeren Schleife gibt es
j=0,1,2 $\Longrightarrow$ 3 Zugriffe auf Seite 3, 1 Zugriff auf Seite 1
Austausch nach LRU: Seite 0 in Rahmen 3 durch Seite 1 ersetzen
optimaler Austausch: Seite 0 in Rahmen 3 durch Seite 1 ersetzen

Zeitpunkt	Seitenrahmen 2	Seitenrahmen 3
Beginn	2000 – 2FFF (`D29E6000 - D29E6FFF`)	3000 – 3FFF (`D29E7000 - D29E7FFF`)
1. Austausch im 1. Durchlauf $i = 0$	2000 – 2FFF (`D29E6000 - D29E6FFF`)	0000 – 0FFF (`D29E4000 - D29E4FFF`)
2. Austausch im Durchlauf mit $i = $ `0FFE`, $j = 2$, $i + j = $ `1000`	3000 – 3FFF (`D29E7000 - D29E7FFF`)	0000 – 0FFF (`D29E4000 - D29E4FFF`)
3. Austausch im Durchlauf mit $i = $ `0FFF`, $j = 0$, $i + j = $ `0FFF`	2000 – 2FFF (`D29E6000 - D29E6FFF`)	0000 – 0FFF (`D29E4000 - D29E4FFF`)
4. Austausch im Durchlauf mit $i = $ `0FFF`, $j = 1$, $i + j = $ `1000`	3000 – 3FFF (`D29E7000 - D29E7FFF`)	0000 – 0FFF (`D29E4000 - D29E4FFF`)
5. Austausch im Durchlauf mit $i = $ `1000`, $j = 2$, $i + j = $ `1002`	3000 – 3FFF (`D29E7000 - D29E7FFF`)	1000 – 1FFF (`D29E5000 - D29E5FFF`)

	Programm-zeile	i	j	Aktion	Seiten-rahmen
1	6	`0000`	-	Seite `D29E7` auslagern, Seite `D29E4` anlegen	3
2	4	`0FFE`	2	Seite `D29E6` auslagern, Seite `D29E7` einlagern	2
3	4	`0FFF`	0	Seite `D29E7` auslagern, Seite `D29E6` einlagern	2
4	4	`0FFF`	1	Seite `D29E6` auslagern, Seite `D29E7` einlagern	2
5	6	`1000`	-	Seite `D29E4` auslagern, Seite `D29E5` einlagern	3
6	6	`1FFE`	2	Seite `D29E8` Zugriffsverletzung	

Bei dieser otimalen, aber fiktiven Ersetzungsstrategie reduziert sich die Anzahl der Ein-/Auslagerungen auf 10 Seiten.

Aufgabe 3.6

In den Code-Cache wird niemals geschrieben, Befehle werden ausschließlich gelesen!

Aufgabe 3.7

Der *Index* kann bei einem Cache nicht geändert werden. Er ist durch den hardwaremäßigen Aufbau des Caches festgelegt. Aktualisiert werden das *Tag* und das *Datum*.

1. **Write-Hit:** Der *Tag* bleibt unverändert.
   Bei A) (*write-through*) werden die Daten sowohl im Cache als auch im Hauptspeicher (HSP) aktualisiert.

Bei B) (*Write-Back*) werden die Daten nur im Cache eingetragen und im Gegensatz zum write-through nicht im HSP aktualisiert.
Es wird ein *Dirty-* bzw. *Modified-Bit* für den geschriebenen Cache-Eintrag gesetzt, das die Inkonsistenz der Daten im Cache und im HSP anzeigt.

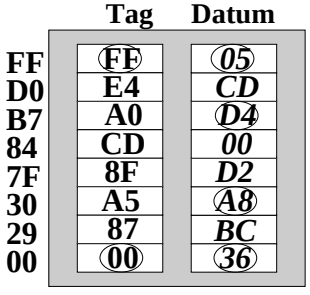
2. **Write-Miss:** Bei A) als auch bei B) keine Veränderung im Cache.
   Die Daten werden direkt im HSP aktualisiert.
3. **Read-Hit:** Bei A) als auch bei B) keine Veränderung im Cache und im HSP.
   Der Prozessor liest das Datum aus dem Cache.
4. **Read-Miss:** Keine Veränderung im HSP.
   Bei A) werden die aus dem Hauptspeicher gelesenen und an den Prozessor weitergegebenen Daten in den Cache eingetragen.
   Dabei wird der aus der Adresse des Datums bestimmte *Tag* zusammen mit den Daten in den Cache-Blockrahmen mit dem entsprechenden *Index* geschrieben.
   Bei B) wird im Unterschied zu A) bei der erforderlichen Verdrängung eines Eintrages aus dem Cache das *Dirty-Bit* geprüft.
   Ist das *Dirty-Bit* gesetzt, wird vor der Verdrängung der im Cache befindliche Eintrag in den HSP geschrieben. In einer solchen Situation kommt es also zunächst zu einem Schreibzugriff auf den HSP, obwohl der Prozessor eigentlich Daten lesen will.

Aufgabe 3.8

Der optimale Fall ergibt sich natürlich, wenn nur Lesezugriffe auf Daten stattfinden, die sich bereits im Cache befinden, d.h. die Cache-Hit-Rate beträgt 100%. In diesem Fall halbiert sich mit den neuen Bausteinen die vom Prozessor beobachtete Zugriffszeit.

Aufgabe 3.9

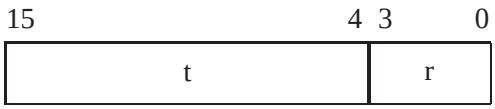
						Cache		
Befehl	Hit	Miss	Akku	Datum	HSP-Adr.	Index	Tag	Datum
6. LDA $8F7F	R		D2	D2	8F7F	7F	8F	D2
7. LDA $FFFF		R	04	04	FFFF	FF	FF	04
8. STA $FFFF	W		05	05	FFFF	FF	FF	05
9. LDA $FFFF	R		05	05	FFFF	FF	FF	05
10. LDA $0000		R	36	36	0000	00	00	36
11. STA $A0B7	W		D4	D4	A0B7	B7	A0	D4



Das nebenstehende Bild zeigt den Cacheinhalt nach Ausführung der elf Anweisungen. Der Cache-Eintrag mit dem Index FF wird insgesamt dreimal modifiziert. Mit der 7. Anweisung wird das *Tag* zu FF und das *Datum* zu 04, mit der 8. Anweisung bleibt das *Tag* FF und das *Datum* wird zu 05. Das *Tag* mit dem Index FF wird also insgesamt zweimal verändert. Aus den beiden Tabellen erkennt man, dass insgesamt sechsmal ein *Datum* im Cache verändert wird.

Aufgabe 3.10

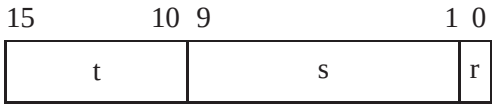
16 Byte Blockgröße $\Rightarrow$ $2^{\#r} = 16$ $\Rightarrow$ $\#r = 4$
512 Byte Cachegröße $\Rightarrow$ $2^{\#q+\#r} = 2^9$ $\Rightarrow$ $\#q = 9-4 = 5$
$\#t =$ Adressbreite $- \#r = 16-4 = 12$
Die Adressaufteilung ergibt sich zu:



Anzahl der Blöcke im Cache: $2^{\#q} = 2^5 = 32$
Größe des Tag-Speichers: $2^{\#q} \cdot \#t = 2^5 \cdot 12 = 384\,\text{Bit} = 48\,\text{Byte}$

Aufgabe 3.11

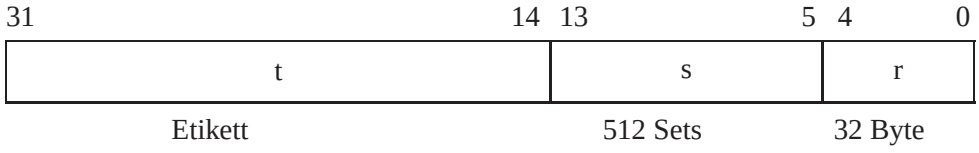
2 Byte Blockgröße $\Rightarrow$ $2^{\#r} = 2$ $\Rightarrow$ $\#r = 1$
8 KB Cachegröße $\Rightarrow$ $2^{\#q+\#r} = 8 \cdot 2^{10} = 2^3 \cdot 2^{10} = 2^{13}$ $\Rightarrow$ $\#q = 13-1 = 12$
$\#s = \#q - \log_2(n) = 12 - \log_2(8) = 12-3 = 9$
$\#t =$ Adressbreite $- (\#s + \#r) = 16-(9+1) = 6$
Die Adressaufteilung ergibt sich zu:



Anzahl der Blöcke im Cache: $2^{\#q} = 2^{12} = 4096$
Größe des Tag-Speichers: $2^{\#q} \cdot \#t = 2^{12} \cdot 6 = 24576\,\text{Bit} = 3072\,\text{Byte}$

Aufgabe 3.12

32 Byte Blockgröße $\Rightarrow$ $2^{\#r} = 32$ $\Rightarrow$ $\#r = 5$
128 KB Cachegröße $\Rightarrow$ $2^{\#q+\#r} = 128 \cdot 2^{10} = 2^7 \cdot 2^{10} = 2^{17}$ $\Rightarrow$ $\#q = 17-5 = 12$
$\#s = \#q - \log_2(n) = 12 - \log_2(8) = 12-3 = 9$
$\#t =$ Adressbreite $- (\#s + \#r) = 32-(9+5) = 18$
Die Adressaufteilung ergibt sich zu:

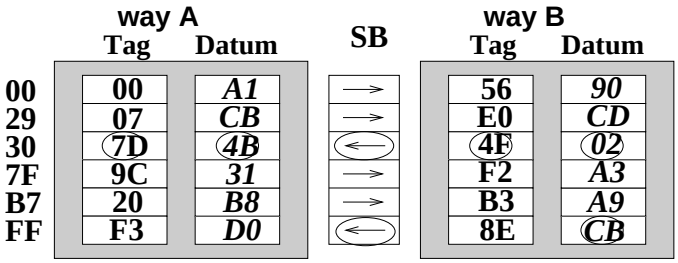


Anzahl der Blöcke im Cache: $2^{\#q} = 2^{12} = 4096$
Größe des Tag-Speichers: $2^{\#q} \cdot \#t = 2^{12} \cdot (18+1) = 77824\,\text{Bit} = 9728\,\text{Byte} = 9{,}5\,\text{KB}$
(Taggröße=19 Bit, 18 Bit für den Adressteil, 1 Bit für das *Dirty*-Bit!)

Aufgabe 3.13

					Cache					
						way A			**way B**	
Befehl	Hit	miss	Akku	HSP-Adr.	Index	Tag	Datum	**SB**	Tag	Datum
5. LDA $0729	R		CB	0729	29	07	CB	$\longrightarrow$	E0	CD
6. STA $8EFF	W		CB	8EFF	FF	F3	D0	$\boxed{\longleftarrow}$	8E	$\boxed{\text{CB}}$
7. LDA $8EFF	R		CB	8EFF	FF	F3	D0	$\longleftarrow$	8E	CB

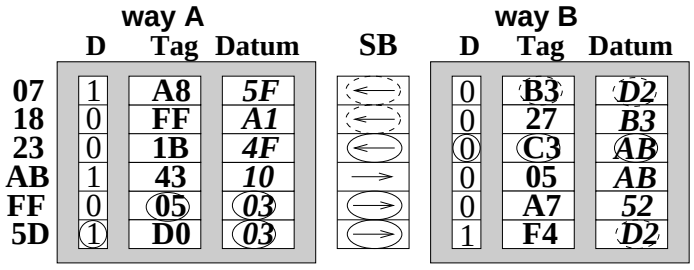
Nach Ausführung der sieben Befehle ergibt sich die folgende Cache-Belegung:



Aufgabe 3.14

					Cache							
						way A				way B		
Befehl	Hit	Miss	Akku	HSP-Adr.	Index	**D**	Tag	Datum	**SB**	**D**	Tag	Datum
5. LDA $05FF		R	03	05FF	FF	0	05	03	→	0	A7	52
6. STA $D05D	W		03	D05D	5D	1	D0	03	→	1	F4	D2
7. LDA $C323		R	AB	C323	23	0	1B	4F	←	0	C3	AB
8. STA $D823		W	AB	D823	23	0	1B	4F	←	0	C3	AB

Nach Ausführung der acht Befehle ergibt sich die folgende Cache-Belegung:



Veränderung im Hauptspeicher

Der HSP wurde bei der Ausführung der acht Befehle insgesamt 3-mal modifiziert, und zwar bei der zweiten, siebten und achten Anweisung.

		HSP-Adresse	geschriebenes Datum
2. Befehl	*Write-Miss*	$B3AB	*B*3
7. Befehl	*Read-Miss*	$D523	02
8. Befehl	*Write-Miss*	$D823	*AB*

Bei der 2. Anweisung wird aufgrund eines *Write-Miss* das Datum *B*3 in den HSP in die Zelle mit der Adresse $B3AB zurückgeschrieben. Bei der 8. Anweisung wird aufgrund eines Write-Miss das Datum *AB* in den HSP in die Zelle mit der Adresse $D823 zurückgeschrieben.

Bei der 7. Anweisung handelt es sich zwar um einen reinen Lesebefehl, da jedoch bei dem *Read-Miss* in der entsprechenden Cache-Line das *dirty-bit* gesetzt ist, muss der entsprechende Inhalt 02 zuerst in den HSP in Zelle $D523 zurückgeschrieben werden.

Bei der 4. und 6. Anweisung handelt es sich ebenfalls um Schreibbefehle. Hier bleibt der Arbeitsspeicher (HSP) jedoch unbeeinflusst, da es sich jeweils um einen *Write-Hit* handelt. Bei diesem *Write-Hit* wird im Write-Back-Modus nur der Cache aktualisiert und dort das *dirty-bit* gesetzt, was die Inkonsistenz der Daten zwischen HSP und Cache anzeigt.

Ein Eintrag mit dem *Index* **5D** wurde im Cache 2-mal modifiziert, und zwar bei der 4. und 6. Anweisung. Der *Tag* wurde dabei nicht modifiziert, da es sich in beiden Fällen um einen *Write-Hit* handelte. Ein *Datum* wurde im Cache insgesamt 5-mal verändert. Ein *SB-Bit* wurde im Cache insgesamt 6-mal verändert.

Aufgabe 3.15

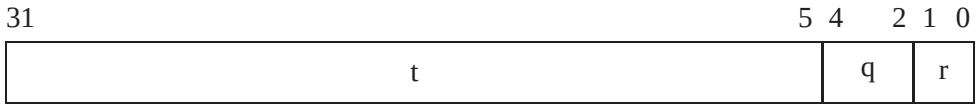
a) Adressaufteilung bei 4 Byte Cache-Blöcken, Cachegröße 32 Byte und 32-Bit-HSP-Adressen

Bei einer Cachegröße von 32 Byte und 4 Byte großen Cache-Blöcken enthält jeder Cache 32/4=8 Blöcke.

Direkt abgebildeter Cache (DM):

4 Byte Blockgröße	$\Rightarrow 2^{\#r} = 4 \Rightarrow \#r = 2$	Wort-Bits 0-1
32 Byte Cachegröße	$\Rightarrow 2^{\#q+\#r} = 2^5 \Rightarrow \#q = 5-2 = 3$	Index-Bits 2-4
$\#t =$ Adressbreite $-(\#q+\#r) = 32-(3+2) = 27$		Tag-Bits 5-31

Die Adressaufteilung ergibt sich zu:

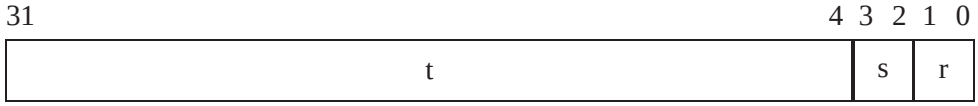


Zusammen mit den beiden Zustandsbits (Valid und Dirty) werden damit 29 Bits für die Verwaltung eines Cacheblocks benötigt.

2-fach satzassoziativer Cache (A2):

4 Byte Blockgröße	$\Rightarrow 2^{\#r} = 4 \Rightarrow \#r = 2$	Wort-Bits 0-1
32 Byte Cachegröße	$\Rightarrow 2^{\#q+\#r} = 2^5 \Rightarrow \#q = 5-2 = 3$	
$\#s = \#q - \log_2(n) = 3-\log_2(2) = 3-1 = 2$		Index-Bits 2-3
$\#t =$ Adressbreite $-(\#s+\#r) = 32-(2+2) = 28$		Tag-Bits 4-31

Die Adressaufteilung ergibt sich zu:

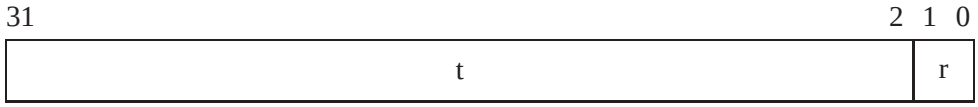


Zusammen mit den beiden Zustandsbits (Valid und Dirty) werden damit 30 Bits für die Verwaltung eines Cacheblocks benötigt.

Vollassoziativer Cache (AV):

4 Byte Blockgröße	$\Rightarrow 2^{\#r} = 4 \Rightarrow \#r = 2$	Wort-Bits 0-1
$\#t =$ Adressbreite $- \#r = 32-2 = 30$		Tag-Bits 2-31

Beim AV-Cache entfällt der Index, so dass die Adressierung vollständig über das Tag abläuft.
Die Adressaufteilung ergibt sich zu:



Zusammen mit den beiden Zustandsbits (Valid und Dirty) werden damit 32 Bits für die Verwaltung eines Cacheblocks benötigt.

Beim DM-Cache ist jeder Datenblock eindeutig genau einem Cacheblock zugeordnet und da das Tag 27 Bit lang ist, wird ein einziger 27-Bit-Vergleicher benötigt. Beim A2-Cache kann jeder Datenblock in zwei Cacheblöcken abgelegt sein und da das Tag 28 Bit lang ist, sind zwei 28-Bit-Vergleicher erforderlich. Beim AV-Cache kann jeder Datenblock in acht Cacheblöcken abgelegt sein und da das Tag 30 Bit lang ist, sind acht 30-Bit-Vergleicher erforderlich.

Beim DM-Cache wird durch den Index nur eine einzige Zeile adressiert, die hier auszuwerten ist. Beim A2-Cache werden durch jeden Index je zwei Zeilen adressiert, die zum entsprechenden Satz gehören. Innerhalb des Satzes sind hier zwei Zeilen auszuwerten. Beim AV-Cache ohne Index müssen alle 8 Zeilen ausgewertet werden.

b) Verlauf der Zugriffe (Cache-Hit oder Cache-Miss)

`$0046 $0009 $0044 $00B9 $0011 $00FB $0055 $00F9 $005C $0006`

Um den Ablauf bei den einzelnen Zugriffen besser zu überschauen, geben wir zunächst eine Zuordnung von HSP-Adressen zu HSP-Blöcken an:

HSP-Adresse	00-03	04-07	08-0B	0C-0F	10-13	14-17	18-1B	1C-1F
HSP-Block	0	1	2	3	4	5	6	7

HSP-Adresse	20-23	24-27	28-2B	2C-2F	30-33	34-37	38-3B	3C-3F
HSP-Block	8	9	10	11	12	13	14	15

HSP-Adresse	40-43	44-47	48-4B	4C-4F	50-53	54-57	58-5B	5C-5F
HSP-Block	16	17	18	19	20	21	22	23

HSP-Adresse	60-63	64-67	68-6B	6C-6F	70-73	74-77	78-7B	7C-7F
HSP-Block	24	25	26	27	28	29	30	31

Da die 10 gegebenen Adressen sich zwischen \$00 und \$FF bewegen, müssen nur die niederwertigen 8 Bit betrachtet werden. Für die Abbildung auf den DM- und den A2-Cache muss die hexadezimale Adresse in eine binäre Adressen gewandelt und diese in Tag, Index und Wort-Bits aufgeteilt werden.

Zugriff	Adressen		DM-Cache			A2-Cache		
	Hexadezimal	Dual	Tag	Index	Offset	Tag	Index	Offset
1	`$46`	0100 0110	010	001	10	0100	01	10
2	`$09`	0000 1001	000	010	01	0000	10	01
3	`$44`	0100 0100	010	001	00	0100	01	00
4	`$B9`	1011 1001	101	110	01	1011	10	01
5	`$11`	0001 0001	000	100	01	0001	00	01
6	`$FB`	1111 1011	111	110	11	1111	10	11
7	`$55`	0101 0101	010	101	01	0101	01	01
8	`$F9`	1111 1001	111	110	01	1111	10	01
9	`$5C`	0101 1100	010	111	00	0101	11	00
10	`$06`	0000 0110	000	001	10	0000	01	10

Damit ergibt sich beim DM-Cache der folgende Zugriffs- bzw. Belegungsverlauf:

Cache-Rahmen	Index	HSP-Adresse, Tag, Hit/Miss		
		1. Belegung	2. Belegung	3. Belegung
Block 0	0 000			
Block 1	1 001	`$46`, 010, Miss	⇒ `$44`, 010, Hit	⇒ `$06`, 000, Miss
Block 2	2 010	`$09`, 000, Miss		
Block 3	3 011			
Block 4	4 100	`$11`, 000, Miss		
Block 5	5 101	`$55`, 010, Miss		
Block 6	6 110	`$B9`, 101, Miss	⇒ `$FB`, 111, Miss	⇒ `$F9`, 111, Hit
Block 7	7 111	`$5C`, 010, Miss		

Entsprechend ergibt sich beim A2-Cache der folgende Zugriffs- bzw. Belegungsverlauf:

Cache-Rahmen		Index	HSP-Adresse, Tag, Hit/Miss 1. Belegung	2. Belegung	3. Belegung
Set 0	way-A way-B	0 00	$11, 0001, Miss		
Set 1	way-A way-B	1 01	$46, 0100, Miss $55, 0101, Miss	⇒ $44, 0100, Hit	⇒ $06, 0000, Miss
Set 2	way-A way-B	2 10	$09, 0000, Miss $B9, 1011, Miss	⇒ $FB, 1111, Miss	⇒ $F9, 1111, Hit
Set 3	way-A way-B	3 11	$5C, 0101, Miss		

Beim AV-Cache ergibt sich der folgende Zugriffs- bzw. Belegungsverlauf:

Cache-Rahmen		HSP-Adresse, Hit/Miss 1. Belegung	2. Belegung
Set 0	way-A	$46, Miss	⇒ $44, Hit
	way-B	$09, Miss	
	way-C	$B9, Miss	
	way-D	$11, Miss	
	way-E	$FB, Miss	⇒ $F9, Hit
	way-F	$55, Miss	
	way-G	$5C, Miss	
	way-H	$06, Miss	

Damit ergibt sich die folgende Hit (X) / Miss (-) Verteilung:

Adresse	$46	$09	$44	$B9	$11	$FB	$55	$F9	$5C	$06	Hits
Block-Nr.	17	02	17	46	04	62	21	62	23	01	
DM-Cache	–	–	*X*	–	–	–	–	*X*	–	–	2
Index	1	2	1	6	4	6	5	6	7	1	
A2-Cache	–	–	*X*	–	–	–	–	*X*	–	–	2
Index	1	2	1	2	0	2	1	2	3	1	
AV-Cache	–	–	*X*	–	–	–	–	*X*	–	–	2

DM

Index	Tag	HSP-Block
0	—	–
1	000	01
2	000	02
3	—	–
4	000	04
5	010	21
6	111	62
7	010	23

A2

Index	Tag	HSP-Block
0	0001	04
	—-	–
1	0000	01
	0101	21
2	0000	62
	1011	46
3	0101	23
	—-	–

AV

Tag	HSP-Block
0100 01	17
0000 10	02
1011 10	46
0001 00	04
1111 10	62
0101 01	21
0101 11	23
0000 01	01

Aufgabe 3.16

Der vollassoziative Cachepeicher hat nur ein Set, daher sind genau 512 Misses zur vollständigen Initialisierung erforderlich. Bei $h = 0,95$ ergibt das 512/(1-0,95)=10240 Adressreferenzen. Da die Missrate mit 1,0 beginnt und während der Initialisierung ständig abnimmt, sind mit großer Wahrscheinlichkeit weniger Referenzen erforderlich.

Aufgabe 3.17

Die OPT-Strategie würde die HSP-Adresse `0E` nicht aus dem Cache entfernen, dadurch ergibt sich ein Treffer mehr, Hitrate: 0,607.

Aufgabe 3.18

$$\begin{aligned} \text{Write-Back:} \quad T_{eff} &= t_{PS} + (1-h)\left[z(1+np_{wr})\,t_{HS} + p_{wr}\,t_{PS}\right] \\ T_{eff} &= t_{PS} + (1-h)zt_{HS} + (1-h)\left(znt_{HS} + t_{PS}\right)p_{wr} \end{aligned}$$

mit $0,3\,p_{wr} = p_{dirty}$ folgt nach Einsetzen der Werte:

$$T_{eff} = 69ns + 18.2ns * p_{wr}$$

$$\begin{aligned} \text{Write-Through:} \quad T_{eff} &= (1-p_{wr})\left[(t_{PS} + (1-h)zt_{HS}) + p_{wr}\,t_{HS}\right] \\ T_{eff} &= t_{PS} + (1-h)zt_{HS} + \left[(h-1)z+1\right)t_{HS} - t_{PS}]\,p_{wr} \end{aligned}$$

nach Einsetzen der Werte folgt:

$$T_{eff} = 47ns + 63ns * p_{wr}.$$

Gleichsetzen ergibt:

$$69ns + 18.2ns * p_{wr} = 47ns + 63ns * p_{wr} \qquad \Rightarrow p_{wr} = 0,491$$

Für $0 \leq p_{wr} \leq 0,491$ ist das *Write-Through*-Verfahren besser!

Aufgabe 3.19

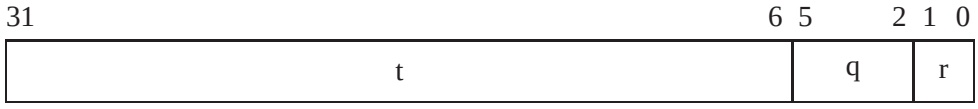
a) Adressaufteilungen

HSP ist wortadressiert, 1 Block umfasst 4 Datenworte, Cachegröße 16 Blöcke, entspricht 64 Datenworten, 32-Bit-HSP-Adressen

Direkt abgebildeter Cache (DM):

4 Datenworte Blockgröße	$\Rightarrow 2^{\#r} = 4 \Rightarrow \#r = 2$	Wort-Bits 0-1
64 Datenworte Cachegröße	$\Rightarrow 2^{\#q+\#r} = 2^6 \Rightarrow \#q = 6-2 = 4$	Index-Bits 2-5
$\#t =$ Adressbreite $-(\#q+\#r) = 32-(4+2) = 26$		Tag-Bits 6-31

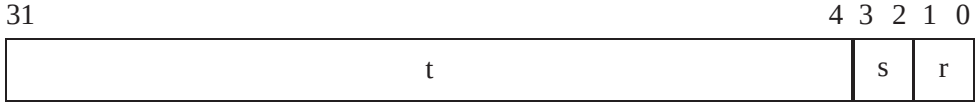
Die Adressaufteilung ergibt sich zu:



2-fach satzassoziativer Cache (A2):

4 Datenworte Blockgröße	$\Rightarrow 2^{\#r} = 4 \Rightarrow \#r = 2$	Wort-Bits 0-1
32 Datenworte Cachegröße	$\Rightarrow 2^{\#q+\#r} = 2^6 \Rightarrow \#q = 6-2 = 4$	
$\#s = \#q - \log_2(n) = 4 - \log_2(4) = 4-2 = 2$		Index-Bits 2-3
$\#t =$ Adressbreite $-(\#s+\#r) = 32-(2+2) = 28$		Tag-Bits 4-31

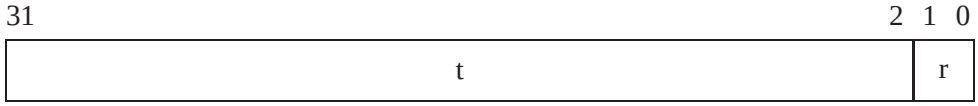
Die Adressaufteilung ergibt sich zu:



Vollassoziativer Cache (AV):

4 Datenworte Blockgröße $\Rightarrow 2^{\#r} = 4 \Rightarrow \#r = 2$ Wort-Bits 0-1
$\#t =$ Adressbreite $- \#r = 32 - 2 = 30$ Tag-Bits 2-31

Beim AV-Cache entfällt der Index, so dass die Adressierung vollständig über das Tag abläuft.
Die Adressaufteilung ergibt sich zu:



Cache-Organisation	Sätze	Blöcke/Satz	Index-Bits	Tag-Bits
direkt abgebildet	16	1	4	26
4-fach satzassoziativ	4	4	2	28
vollassoziativ	1	16	0	30

b) Verlauf der Zugriffe (Cache-Hit oder Cache-Miss)

`1F296FFA, 378CF121, 1F296FFB, 378D1831, 1F296FFC,`
`378D3F41, 1F296FFD, 378D6651, 1F296FFE, 378D8D61,`
`1F296FFF, 378DB471, 1F297000, 378DDB81, 1F297001,`
`378E0291, 1F297002, 378E29A1, 1F297003, 378E50B1`

Für die Abbildung auf den DM-, den A4-, und den AV-Cache müssen die hexadezimalen Adressen in binäre Adressen gewandelt werden. Da Index- und Wort-Bits zusammen maximal beim direkt abgebildeten Cache 6 Bits ergeben, stellen wir nur das niederwertigste Byte der Adressen binär dar.

Zugriff	Adressen		DM-Cache			A2-Cache		
			Tag	Index	Offset	Tag	Index	Offset
1	1F296FFA	1F296F 1111 1010	1F296F	11 1110	10	1F296F	1111 10	10
2	378CF121	378CF1 0010 0001	378CF1	00 1000	01	378CF1	0010 00	01
3	1F296FFB	1F296F 1111 1011	1F296F	11 1110	11	1F296F	1111 10	11
4	378D1831	378D18 0011 0001	378D18	00 1100	01	378D18	0011 00	01
5	1F296FFC	1F296F 1111 1100	1F296F	11 1111	00	1F296F	1111 11	00
6	378D3F41	378D3F 1000 0001	378D3F	10 0000	01	378D3F	1000 00	01
7	1F296FFD	1F296F 1111 1101	1F296F	11 1111	01	1F296F	1111 11	01
8	378D6651	378D66 0101 0001	378D66	01 0100	01	378D66	0101 00	01
9	1F296FFE	1F296F 1111 1110	1F296F	11 1111	10	1F296F	1111 11	10
10	378D8D61	378D8D 0110 0001	378D8D	01 1000	01	378D8D	0110 00	01
11	1F296FFF	1F296F 1111 1111	1F296F	11 1111	11	1F296F	1111 11	11
12	378DB471	378DB4 0111 0001	378DB4	01 1100	01	378DB4	0111 00	01
13	1F297000	1F2970 0000 0000	1F2970	00 0000	00	1F2970	0000 00	00
14	378DDB81	378DDB 1000 0001	378DDB	10 0000	01	378DDB	1000 00	01
15	1F297001	1F2970 0000 0001	1F2970	00 0000	01	1F2970	0000 00	01
16	378E0291	378E02 1001 0001	378E02	10 0100	01	378E02	1001 00	01
17	1F297002	1F2970 0000 0010	1F2970	00 0000	10	1F2970	0000 00	10
18	378E29A1	378E29 1010 0001	378E29	10 1000	01	378E29	1010 00	01
19	1F297003	1F2970 0000 0011	1F2970	00 0000	11	1F2970	0000 00	11
20	378E50B1	378E50 1011 0001	378E50	10 1100	01	378E50	1011 00	01

Damit ergeben sich die folgenden Zugriffs- bzw. Belegungsverläufe:

Cache-Rahmen	Index	HSP-Adresse, Tag, Hit/Miss
Block 0	0000	378D3F41, 378D3F10, Miss ⇒ 1F297000, 1F297000, Miss ⇒ 378DDB81, 378DDB10, Miss ⇒ 1F297001, 1F297000, Miss ⇒ 1F297002, 1F297000, Hit ⇒ 1F297003, 1F297000, Hit
Block 1	0001	
Block 2	0010	
Block 3	0011	
Block 4	0100	378D6651, 378D6601, Miss ⇒ 378E0291, 378E0210, Miss
Block 5	0101	
Block 6	0110	
Block 7	0111	
Block 8	1000	378CF121, 378CF100, Miss ⇒ 378D8D61, 378D8D01, Miss ⇒ 378E29A1, 378E2910, Miss
Block 9	1001	
Block 10	1010	
Block 11	1011	
Block 12	1100	378D1831, 378D1800, Miss ⇒ 378DB471, 378DB401, Miss ⇒ 378E50B1, 378E5010, Miss
Block 13	1101	
Block 14	1110	1F296FFA, 1F296F11, Miss ⇒ 1F296FFB, 1F296F11, Hit
Block 15	1111	1F296FFC, 1F296F11, Miss ⇒ 1F296FFD, 1F296F11, Hit ⇒ 1F296FFE, 1F296F11, Hit ⇒ 1F296FFF, 1F296F11, Hit

Cache-Rahmen		Index	HSP-Adresse, Tag, Hit/Miss
Set 0	way-A	00	378CF121, 378CF12, Miss ⇒ 378D8D61, 378D8D6, Miss ⇒ 378E0291, 378E029, Miss
	way-B		378D1831, 378D183, Miss ⇒ 378DB471, 378DB47, Miss ⇒ 378E29A1, 378E29A, Miss
	way-C		378D3F41, 378D3F4, Miss ⇒ 1F297000, 1F29700, Miss ⇒ 1F297001, 1F29700, Hit ⇒ 1F297002, 1F29700, Hit ⇒ 1F297003, 1F29700, Hit
	way-D		378D6651, 378D665, Miss ⇒ 378DDB81, 378DDB8, Miss ⇒ 378E50B1, 378E50B, Miss
Set 1	way-A	01	
	way-B		
	way-C		
	way-D		
Set 2	way-A	10	1F296FFA, 1F296FF, Miss ⇒ 1F296FFB, 1F296FF, Hit
	way-B		
	way-C		
	way-D		
Set 3	way-A	11	1F296FFC, 1F296FF, Miss ⇒ 1F296FFD, 1F296FF, Hit ⇒ 1F296FFE, 1F296FF, Hit ⇒ 1F296FFF, 1F296FF, Hit
	way-B		
	way-C		
	way-D		

Cache-Rahmen		HSP-Adresse, Tag, Hit/Miss
Set 0	way-A	1F296FFA, 1F296FF 10, Miss ⇒ 1F296FFB, 1F296FF 10, Hit
	way-B	378CF121, 378CF12 00, Miss
	way-C	378D1831, 378D183 00, Miss
	way-D	1F296FFC, 1F296FF 11, Miss ⇒ 1F296FFD, 1F296FF 11, Hit ⇒ 1F296FFE, 1F296FF 11, Hit ⇒ 1F296FFF, 1F296FF 11, Hit
	way-E	378D3F41, 378D3F4 00, Miss
	way-F	378D6651, 378D665 00, Miss
	way-G	378D8D61, 378D8D6 00, Miss
	way-H	378DB471, 378DB47 00, Miss
	way-I	1F297000, 1F29700 00, Miss ⇒ 1F297001, 1F29700 00, Hit ⇒ 1F297002, 1F29700 00, Hit ⇒ 1F297003, 1F29700 00, Hit
	way-J	378DDB81, 378DDB8 00, Miss
	way-K	378E0291, 378E029 00, Miss
	way-L	378E29A1, 378E29A 00, Miss
	way-M	378E50B1, 378E50B 00, Miss
	way-N	
	way-O	
	way-P	

Cache-Belegung nach jeweils allen Zugriffen:

DM

Index	Tag
0000	1F2970 00
0001	---- –
0010	---- –
0011	---- –
0100	378E02 10
0101	---- –
0110	---- –
0111	---- –
1000	378E29 10
1001	---- –
1010	---- –
1011	---- –
1100	378E50 10
1101	---- –
1110	1F296F 11
1111	1F296F 11

A2

Index	Block	Tag
00	A	378E029
	B	378E29A
	C	1F29700
	D	378E50B
01	A	-----
	B	-----
	C	-----
	D	-----
10	A	1F296FF
	B	-----
	C	-----
	D	-----
11	A	1F296FF
	B	-----
	C	-----
	D	-----

AV

Block	Tag
A	1F296FF 10
B	378CF12 00
C	378D183 00
D	1F296FF 11
E	378D3F4 00
F	378D665 00
G	378D8D6 00
H	378DB47 00
I	1F29700 00
J	378DDB8 00
K	378E029 00
L	378E29A 00
M	378E50B 00
N	----- –
O	----- –
P	----- –

Für die Trefferraten gilt:

6/20=0,30 7/20=0,35 7/20=0,35

Aufgabe 3.20

Zur besseren Übersichtlichkeit sind in der folgenden Tabelle Einträge, die sich gegenüber dem vorherigen Takt nicht verändert haben, leer.

Prozessor	**Aktion**	**Cache 1 Block 1**	**Cache 1 Block 2**	**Cache 1 Block 3**	**Cache 2 Block 1**	**Cache 2 Block 2**	**Cache 2 Block 3**
-	-	E/8	E/12	I/-	E/6	I/-	I/-
2	READ 10					E/10	
1	WRITE 8	M/8					
1	READ 10			S/10		S/10	
2	READ 8	S/8					S/8
1	WRITE 8	M/8					I/-
1	WRITE 8						
1	READ 18		E/18				
2	WRITE 10			I/-		M/10	
2	WRITE 18		I/-				M/18

Aufgabe 3.21

Fragen zu Speicherverwaltung und Cache

a) Speicherhierarchie für PCs
Register –> Primär-Cache –> Sekundär-Cache –> Hauptspeicher –> Sekundärspeicher.

b) Voraussetzung für eine effiziente Speicherhierarchie
Die Nutzung zeitlicher und räumlicher Lokalität und damit ein Programmablauf, der dies zulässt.

c) Adressübersetzungspuffer (TLB)
Der TLB dient dazu, den Zugriff auf Speicherwörter (Befehle oder Daten) bei Verwendung einer virtuellen Speicherverwaltung zu beschleunigen.

d) Technologische Unterschied zwischen Cache und HSP
Cache-Speicher ist auf dem Prozessor-Chip oder als SRAM-Speicher implementiert, während für den HSP DRAM-Speicher verwendet werden. SRAM wird durch Transistorschaltungen realisiert, ist schnell und platzaufwendig, während DRAM mittels kleiner Kondensatorladungen implementiert wird und damit im Zugriff langsamer ist, aber mehr Bits pro Chip-Fläche unterbringen kann.

e) Unterschied Rückschreibe- und Durchschreibe-Strategie
Bei der Rückschreibestrategie wird ein zu speicherndes Wort zunächst nur in den Cache geschrieben und erst auf Anforderung, z.B. durch das Cache-Kohärenzverfahren oder beim Ersetzen des Cache-Blocks in den HSP übertragen. Beim Durchschreibeverfahren wird jedes zu speichernde Wort in den Cache und sofort in den Hauptspeicher geschrieben.

f) Vor- und Nachteile virtuell bzw. physikalisch adressierter Cache-Speicher
Bei virtuell adressiertem Cache wird die Zugriffsgeschwindigkeit durch vermeiden der Adressübersetzung erhöht, während beim physikalisch adressierten Cache vor dem Zugriff ein TLB-Zugriff nötig ist. Nachteilig ist beim virtuell adressierten Cache die Notwendigkeit des Löschens des gesamten Cache-Inhalts vor einem Prozesswechsel sowie die Notwendigkeit einer zusätzlichen physikalischen Adresstabelle für das Schnüffeln auf dem Bus zur der Implementierung der Cache-Kohärenz (auf dem Bus gibt es natürlich nur physikalische Adressen).

g) Cache-Kohärenz
Eine Cache-Speicherverwaltung heißt *cache-kohärent,* falls ein Lesezugriff immer den Wert des zeitlich letzten Schreibzugriffs auf das entsprechende Speicherwort liefert. Ein Cache-Kohärenzprotokoll ist das MESI-Protokoll.

h) Vor- und Nachteile des Durchschreib-Verfahrens gegenüber dem Rückschreibverfahren
Vorteil Durchschreib-Verfahren: garantiert die Datenkonsistenz zwischen Cache und Arbeitsspeicher, da jeder Schreibzugriff auf den Cache gleichzeitig auch im Arbeitsspeicher ausgeführt wird.
Nachteil Durchschreib-Verfahren: Schreibzugriffe benötigen dann die langsame Zykluszeit auf den Arbeitsspeicher und belasten den Systembus.

Vorteil Rückschreib-Verfahren: auch Schreibzugriffe können mit der schnellen Cache-Zykluszeit abgewickelt werden. Hierbei wird ein Datum erst dann in den Arbeitsspeicher zurückgeschrieben, wenn es im Cache durch ein neues verdrängt werden soll. Dadurch wird der Systembus auch entlastet.
Nachteil Durchschreib-Verfahren: Problem der Datenkonsistenz zwischen Cache und Arbeitsspeicher, wenn andere Komponenten des Systems, z.B. ein DMA-Controller, auf den Arbeitspeicher zugreifen können.