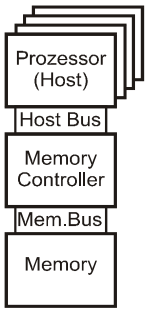
# 7 Der Datenverkehr von/zu den Prozessoren

Personal Computer werden überwiegend mit Ein-Prozessor-Motherboards ausgestattet. Doppel-Prozessor-Motherboards sind nicht sehr verbreitet. Mehr-Prozessor-Strukturen sind im Anwendungsfeld der Hochleistungs-Server die Regel.

Es ist selbstverständlich, dass ein Host-Bus, der zwei und mehr Prozessoren zu „bedienen“ hat, anders aufgebaut sein muss, als ein Host-Bus, der nur einen einzigen Prozessor zu bedienen hat.
Der Hauptspeicher ist allen gemeinsam. D.h., dass es einen „Nadelöhreffekt“ gibt, je mehr Prozessoren gleichzeitig zugreifen.




Gibt es eine Möglichkeit, den Nadelöhreffekt zu entschärfen?

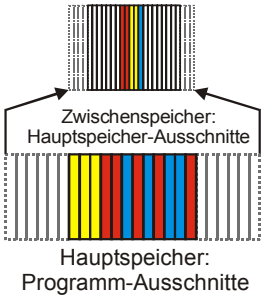
## 7.1 Das Cache-Konzept

### Der Nutzen einer geeigneten Speicherhierarchie

In der Regel greift der Prozessor beim Programmablauf nicht nur einmal zu einem Befehl oder einer Variablen im Hauptspeicher zu, sondern mehrfach: im ersten Fall, weil z.B. eine Schleife mehrfach durchlaufen wird, im zweiten Fall, weil eine Variable mehrfach abgefragt oder verändert wird. Findet man den Befehl bzw. die Variable beim wiederholten Zugriff in einem schnellen Zwischenspeicher, spart man Zugriffszeit.

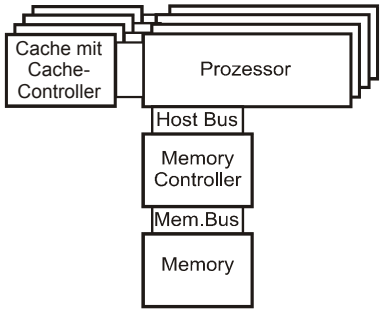
Der Nutzen für den Programmdurchsatz ist zweifach:
- Durch die schnellere Zugriffszeit ergibt sich eine kürzere Programmablaufzeit,
- man hat weniger Zugriffe zum Hauptspeicher und damit in Multiprozessor-Strukturen weniger Verdrängungen am Hauptspeicher.

Das Ziel des Konzeptes eines schnellen Zwischenspeichers muss sein, geeignete Ausschnitte des gerade ablaufenden Programmes bereitzuhalten. Die Ausschnitte müssen mit dem Programmfortschritt aktualisiert werden.




Aus der Sicht eines Prozessors muss verdeckt bleiben, ob der Speicherzugriff zum Zwischenspeicher oder zum Hauptspeicher stattfindet, was die Bezeichnung **Cache** für den schnellen Zwischenspeicher nahe legt.

Aus der Sicht eines Prozessors interessiert die kürzeste Zeit, die vom Beginn bis zum Ende eines Speicherzugriffs vergeht. Je kürzer diese Zeit ist, umso schneller kann der nächste Zugriff erfolgen. Deshalb müssen zwei Zugriffsvorgänge gleichzeitig beginnen: der zum Zwischenspeicher und der zum Hauptspeicher. Der schnellere Zwischenspeicher beendet den Zugriff zum langsameren Hauptspeicher vorzeitig, wenn ein gültiger Hauptspeicherausschnitt im Zwischenspeicher vorliegt. Deshalb gibt es zwei unabhängige Zugriffspfade: einen zum Zwischenspeicher und einen zum Hauptspeicher.

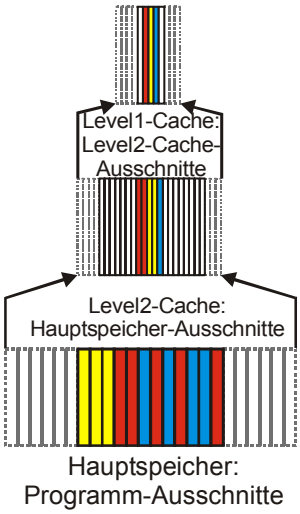


Den schnellsten Zugriff zum Zwischenspeicher hat man, wenn man ihn auf dem Prozessor-Chip realisiert.

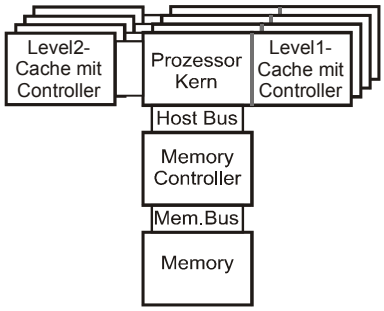
Der Entwurf der Prozessor-Schaltungen erlaubt nun einen weiteren Gewinn, wenn man den Cache selbst wieder so gliedert, dass ein kleiner, kernnaher Cache einen Ausschnitt eines kernferneren großen Cache enthält.

Der Cache, der am nächsten am Prozessor-Kern liegt, wird **Level1-Cache** genannt.
Der Cache, aus dem die Ausschnitte des Level1-Cache gebildet werden, heißt **Level2-Cache**.

Der Level1-Cache enthält eine Untermenge der Speicherwörter des Level2-Caches, der eine Untermenge der Speicherwörter des Hauptspeichers enthält.




Das Cache-Konzept bewirkt, dass deutlich mehr Speicherzugriffe zu den „privaten“ Caches der Prozessoren als zum gemeinsamen Hauptspeicher erfolgen.




**Die Cache-Struktur in modernen Prozessoren**

In modernen Prozessoren sind der Level1 und der Level2-Cache zusammen mit dem Prozessorkern auf einem einzigen Die realisiert.

Das erkennt man an der Basisstruktur eines modernen Prozessors (Bild 7.1).

- Er braucht eine geeignete Bus Unit, damit er über den System-Bus mit der „Außenwelt“ zusammenwirken kann.

- Das interne Cache-System ist der Puffer im Datenverkehr des Prozessorkerns mit dem Hauptspeicher. Es wird damit zu einem der entscheidenden Faktoren für die Geschwindigkeit des Programmablaufes. Je mehr Speicherzugriffe zu den schnellen internen Zwischenspeichern als zum langsamen Hauptspeicher erfolgen, umso besser für einen kurzen Programmablauf. Wenn man beim Entwurf eines neuen schnelleren Prozessortyps eine kürzere Zykluszeit plant, dann muss man auch realisieren, ob das Cachesystem die schnellere Nachfrage befriedigen kann.

- Er braucht eine Steuereinheit, die aus den Maschinenbefehlen geeignete Sequenzen von internen Steueroperationen macht, die die Register und die operativen Einheiten (Execution Units) zur Ausführung der gewünschten Maschinenbefehle veranlasst.

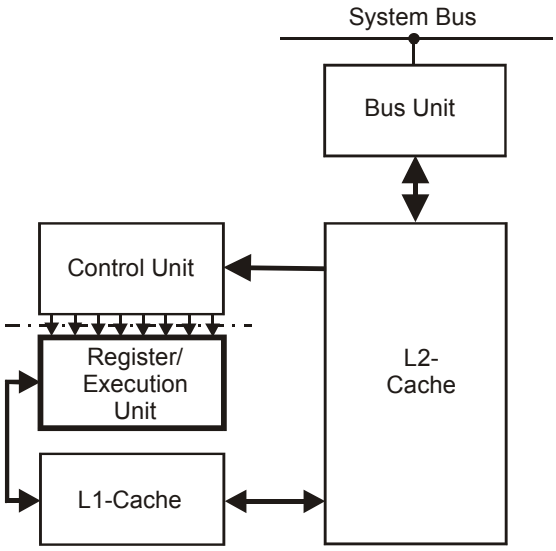



Bild 7.1: Basis-Struktur eines modernen Prozessors

Im Folgenden wird die Cache-Struktur diskutiert.

## 7.2 Basisverfahren für Caches

Im Cache können sich nur Ausschnitte des Hauptspeichers befinden. Die Frage ist, in welchen Einheiten man diese Ausschnitte bildet.

Die Einheit kann nur ein Mehrfaches der „normalen“ Speicherwortlänge sein, in den aktuellen Computersystemen also ein Mehrfaches von 64Bit, das durch die Anzahl der Speicherzugriffe pro Burst festgelegt ist.

Ein Burst von 4 Speicherwörtern ergibt eine Gesamtwortlänge von 256 Bit. Das ist ein üblicher Wert in realisierten Cache-Konzepten. Ein solches Groß-Wort ist ein cachebares Wort.

Für die Adressierung der cachebaren Wörter sind die Hostadressbits $HA_5$ bis $HA_n$ zuständig.


Cachebare Wörter können nach Bedarf in den Cache „kopiert“ werden. Dies sind dann die Hauptspeicherauschnitte, mit denen der Prozessor arbeitet.

Im folgenden werden cachebare Wörter im Hauptspeicher und im Cache zwischengespeicherte Wörter auch als **Cachelines** bezeichnet.

Mit welchem Verfahren kann man aus der Menge der cachebaren Wörter im Hauptspeicher diejenigen aussuchen, die im Cache zwischengespeichert werden sollen?

Das Verfahren muss gewährleisten, dass bei jedem Adressierungsvorgang unverzögert - d.h. mit der Antwortzeit des schnellen Zwischenspeichers - herauskommt, ob das adressierte Wort im Zwischenspeicher steht oder nicht.

Wenn man sofort beim Adressierungsvorgang eines Cache herausfinden will, ob das adressierte cachebare Hauptspeicherwort als Kopie im Cache steht, dann muss das im Cache stehende Wort ein Merkmal enthalten, das es eindeutig erkennbar macht.

Die Adresse ist der eindeutige Identifikator eines Hauptspeicherwortes. Also ist es sinnvoll, die Adresse als Merkmal zu verwenden.

Im folgenden soll die gesamte Adresse als Anwesenheitsmerkmal gespeichert werden, was in Bild 7.2 an einem beispielhaften Szenario gezeigt wird.

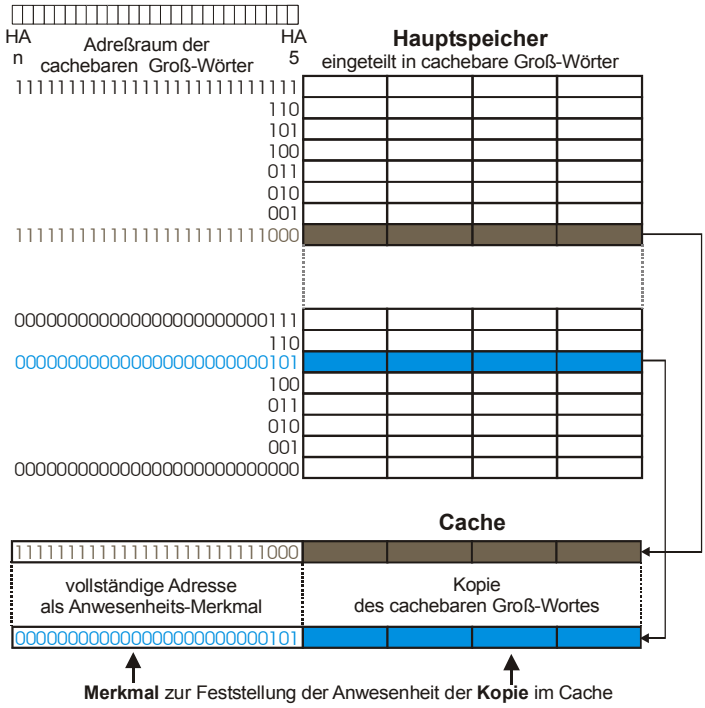



Bild 7.2: Cache mit Zeilen für die Kopie eines cachebaren Groß-Wortes und seiner vollständigen Adresse als Anwesenheitsmerkmal; zwei willkürliche beispielhafte Einträge

Es ist selbstverständlich, dass der Cache nur eine beschränkte Anzahl von Zeilen hat, d.h. dass die Größe des Cache begrenzt ist. Der Cache soll ja nur einen Ausschnitt des Hauptspeichers bilden.

Wie kann man nun mit Hilfe eines schnellen Vergleiches feststellen, ob eine Kopie vorhanden ist oder nicht?

Einen schnellen Vergleich kann man auf der Basis von besonderen Speichern realisieren, die inhaltsbezogen adressieren und **assoziative Speicher** heißen. Bei der Adressierung werden die Speicherwörter nicht durch direkt angegebene Adressen ausgewählt, sondern die Auswahl geschieht durch Vergleich der angelegten Adresse mit demjenigen Teil der Speicherwörter, der den Adresswert enthält. Sobald Gleichheit festgestellt wird (Treffer, match), wird das entsprechende Wort aus dem Speicher ausgegeben bzw. in den Speicher geschrieben. Der Cache füllt sich beim Gebrauch. Wenn er voll ist, müssen Einträge überschrieben werden. Wörter, die beim Gebrauch durch den Prozessor verändert wurden, müssen in den Hauptspeicher zurückgeschrieben werden, bevor sie überschrieben werden.

Assoziative Speicher sind sehr komplex und haben eine verhältnismäßig geringe Speicherkapazität. Sie haben sich für den Masseneinsatz nicht durchgesetzt.

Das beschriebene Basis-Verfahren ist zwar geeignet, eine bestimmte Anzahl beliebig ausgewählter cachebarer Speicherwörter im Cache zu speichern, aber auf Kosten eines komplexen Verfahrens der Auswahl.

Die „Unbegrenztheit“ des Basisverfahrens erkennt man daran, dass in jeder „Cache-Zeile“ ein beliebiges - aus dem gesamten Hauptspeicher ausgewähltes - cachebares Speicherwort stehen kann. Die Vereinfachung des Verfahrens kann dort ansetzen. In einer Cache-Zeile soll nicht mehr ein beliebiges cachebares Groß-Wort aus dem Gesamtspeicher stehen, sondern ein beliebiges aus einem kleineren geeigneten Ausschnitt des Hauptspeichers.

## 7.3 Setadressierte Verfahren

### Prinzip der Abbildung des Hauptspeichers auf den Cache

Angenommen, man teilt die Adresse in einen Teil mit niederwertigen und einen Teil mit höherwertigen Adressbits.

Dann ist das übliche Verfahren, mit den höherwertigen Adressbits den Hauptspeicher in einzeln adressierbare Speicherblöcke einzuteilen und mit den niederwertigen Adressbits die Adresse eines einzelnen Speicherwortes innerhalb des Speicher-blockes. Das gemeinsame Merkmal der Menge der Speicherwörter in einem Speicherblock ist derselbe Adresswert in den höherwertigen Adressbits.

Ganz andere Mengen werden gebildet, wenn man annimmt, dass die Speicherwörter, die zur Menge gehören, denselben niederwertigen Adresswert haben.

Es ist nun die Frage, welche der beiden Mengenbildungen geeigneter für die Cache-Anwendung ist.

Der Sinn der Mengenbildung ist prinzipiell, die „Abbildung“ des Hauptspeichers in den Cache zu systematisieren. Diese Abbildung ordnet einer genau definierten Menge von Groß-Wörtern im Hauptspeicher einen genau definierten Speicherplatz für ein einziges Groß-Wort (oder einige wenige Groß-Wörter) aus dieser Menge zu. Das ist der statische Aspekt des Problems. Der dynamische Aspekt liegt darin, wie beim Programmablauf die Groß-Worte im Hauptspeicher bzw. Cache aktuell gehalten werden.

Angenommen man unterscheidet die Mengen mit Hilfe der höherwertigen Adressbits. Dann müsste es für jede Menge mindestens einen Speicherplatz im Cache geben, der ein Groß-Wort daraus aufnehmen kann. Angenommen, man sieht einen einzigen Speicherplatz vor. Wenn nun ein Programm linear durchlaufen wird, werden die Speicheradressen fortlaufend durch Inkrementierung aktualisiert und in der Regel ändern sich nur die niederwertigen Adressbits. Um das Groß-Wort, das im Cache steht und wieder verwendbar sein soll, aktuell zu halten, muss das jeweils zuletzt gebrauchte dort abgelegt werden, was eine fortlaufende Aktualisierung des Speicherplatzes im Cache nach sich ziehen würde. Das wäre eine sinnlose Lösung, auch dann, wenn man mehr Speicherplätze pro Menge im Cache zur Verfügung stellt.
Diese Mengenbildung ist nicht geeignet. Nun soll die Alternative an einem Beispiel untersucht werden.

Angenommen, man nutzt die drei niederwertigen Adressbits der Groß-Wörter dafür, die Mengen (**Sets**) zu definieren. Dann ergeben sich 8 Sets mit je einem Speicherplatz (für ein Groß-Wort aus diesem Set) im Cache (Bild 7.3).

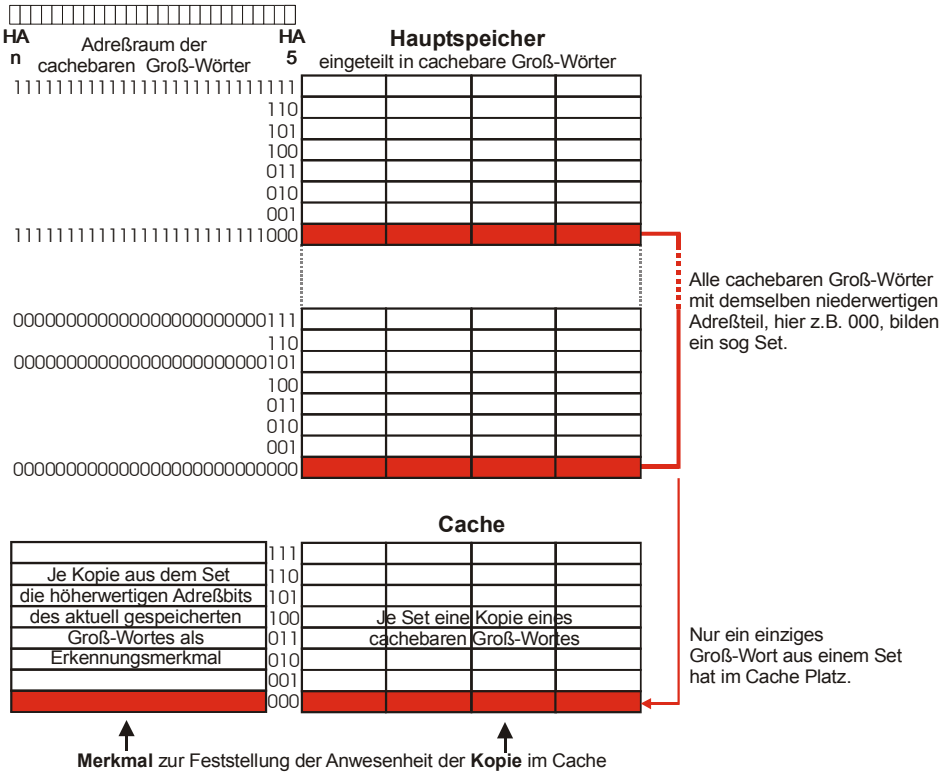



Bild 7.3: Bildung von 8 Sets und Zuordnung je eines Speicherplatzes im Cache für ein Groß-Wort jedes Sets

Wenn der Prozessor einen Speicherzugriff mit einer bestimmten Hauptspeicher-adresse ausführt, dann muss mit der angelegten Adresse eindeutig entscheidbar sein, ob das angeforderte Groß-Wort im Cache ist oder nicht. Das muss wie im Basisverfahren auch hier gelten.

Der niederwertige Adressteil (die drei Bit im Bild 7.3) macht den Speicherplatz des Set eindeutig erkennbar, der das cachebare Großwort enthält. Welches von den cachebaren aus dem Set dort tatsächlich steht, kann man nur erkennen, wenn der höherwertige Adressteil des tatsächlich gespeicherten Groß-Wortes hinzugefügt wird. Man nennt diesen Adressteil **Tag**.

Beim Speicherzugriff vergleicht man die vom Prozessor angelegten höherwertigen Adressbits mit dem gespeicherten Tag des Set. Eine Übereinstimmung heißt **Hit**, eine Nichtübereinstimmung **Miss**.

### Das Füllen des Cache und die wiederholte Nutzung des Cache-Inhaltes

Man kann sich am Bild 7.4 plausibel machen, was im Falle eines wiederholten linearen Programmablaufes geschieht. Angenommen, der Ablauf startet bei der Hostadresse 0. Zuerst wird Set0, dann Set1 usw. angesprochen. Also werden nacheinander die entsprechenden Speicherplätze der Sets im Cache aktualisiert. Wenn beispielsweise bei der Hostadresse 7 eine Verzweigung zur Hostadresse 0, d.h. wenn eine Wiederholung erfolgt, stehen die cachebaren Groß-Wörter schon im Cache.

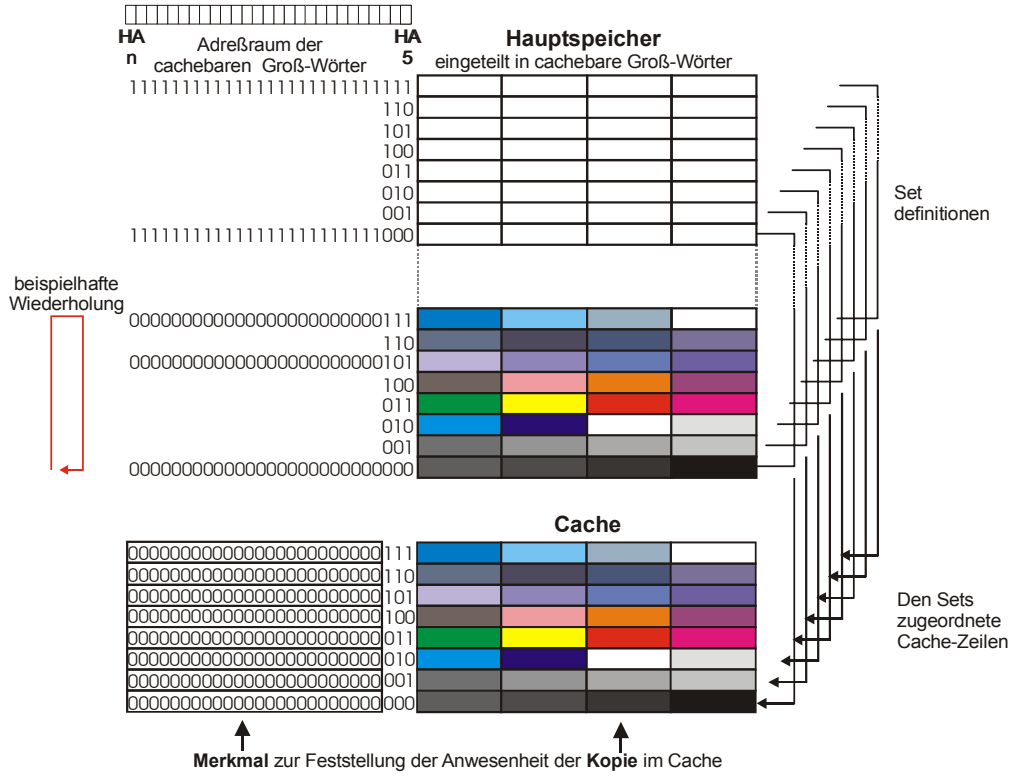



Bild 7.4: Zustand des Cache beim Durchlaufen einer beispielhaften Schleife

Man erkennt sofort, dass eine Vergrößerung der Zahl der Sets, also eine Vergrößerung des Cache, die Chance dafür erhöht, wiederholbare Befehle zu finden. Das gilt nicht nur für Befehle, sondern auch für Variable.

Das Cachekonzept nennt man **setadressierter Cache**.

Das Laden einer Cacheline aus dem Hauptspeicher heißt **Cacheline-Fill**.

**Die Konsistenz zwischen Hauptspeicher- und Cache-Inhalt**

Das Überschreiben einer Cache-Zeile beim Cacheline-Fill kann nicht bedingungslos erfolgen. Es hängt von der Strategie ab, die der Prozessor beim Schreiben in den Cache bzw. Hauptspeicher verfolgt.

In jedem Fall muss man dafür sorgen, dass keine Änderung verloren geht.

Eine einfache Lösung wäre, sowohl die Kopie in der Cache-Zeile als auch das Original im Hauptspeicher im gleichen Zugriff zu schreiben. Das wird **Write Through** genannt.

Mit dem Write-Through-Verfahren gibt es zu keiner Zeit eine Ungleichheit zwischen den Kopien im Cache und den Originalen im Hauptspeicher.

Diese Eigenschaft der ununterbrochen geltenden Konsistenz zwischen Hauptspeicher und Cache erkauft man sich durch Verzicht auf einen Zeitgewinn, den man bei einer anderen Strategie hätte. Man könnte die veränderte Kopie im Cache erst dann in den Hauptspeicher schreiben, wenn die veränderte Kopie durch ein Cacheline-Fill überschrieben wird.

Dieses Verfahren heißt **Write Back**. Es lässt eine zeitweilige Inkonsistenz zu, die aber keinen Datenverlust bewirkt. Vielmehr spart man Zeit. Bei mehrfachen schreibenden Zugriffen auf eine Cache-Zeile - zwischen einem Cacheline-Fill und dem nächsten überschreibenden Cacheline-Fill - addieren sich die kürzeren Zugriffszeiten zum Cache, bis beim nächsten überschreibenden Cacheline-Fill das längere Zurückschreiben in den Hauptspeicher erzwungen wird.

Das Verfahren ist nur dann anwendbar, wenn man für jede Cache-Zeile getrennt erkennen kann, ob sie modifiziert worden ist oder nicht. Nur dann kann man eine bedingte Reaktion erzeugen. Das bedeutet, dass man in jeder Cache-Zeile ein entsprechendes Zustandsmerkmal braucht.

Man braucht noch ein zusätzliches Zustandsmerkmal. Beim Hochfahren eines Computersystemes sind die Cache-Zeilen in einem undefinierten Zustand und man muss durch ein entsprechendes Zustandsmerkmal ein Cacheline-Fill erzwingen. Danach ist das nicht mehr notwendig. Auch dieses bedingte Verhalten erzwingt ein entsprechendes Zustandsmerkmal pro Cache-Zeile.

Die in den Cache-Zeilen anzugebenden Merkmale müssen also (mindestens) folgende Zustände verschlüsseln: Ungültig; Gültig/nicht modifiziert; Gültig/modifiziert.

Für die Verschlüsselung von drei Zuständen braucht man 2 Bits. Man kann dafür zwei explizite Merkmalbits mit den Funktionen (valid, modified) vorsehen und z.B. wie folgt zuordnen: (0,0) = ungültig; (1,0) = gültig, nicht modifiziert; (1,1) = gültig, modifiziert.
Man kann aber auch davon ausgehen, dass man mit 2 Bits 4 Zustände frei kodieren kann. Es gibt (etwas komplexere als die hier vorgestellten) Cache-Strategien, die das nutzen.

In jedem Fall braucht man also zwei zusätzliche Bits pro Cachezeile, um darin den Zustand der Cache-Zeile angeben bzw. erkennen zu können (Bild 7.5).

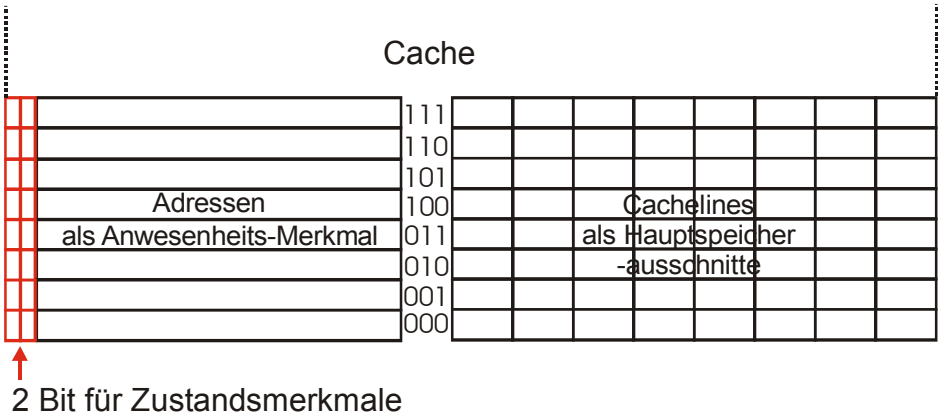



Bild 7.5: Zustandsmerkmale pro Cacheline

**Chance für Hits und Cache-Größe**

Kleinere Sets bedeuten mehr Cache-Zeilen, längere Setadressen und kürzere Tags. Je mehr Cache-Zeilen ein Cache hat, d.h. je größer der Cache ist, umso größer wird die Chance für Cache-Hits (Treffer), umso größer wird der Gewinn für den Durchsatz. Auf der anderen Seite nähert man sich immer mehr dem Fall, dass man den Hauptspeicher „verdoppelt“.

Die angemessene Aufteilung im Sinne eines guten Preis/Leistungsverhältnisses ist eine anspruchsvolle Aufgabe für den Computerentwurf.

## 7.4 Die hierarchische Cache-Struktur

### Prinzipien einer Struktur für den größten Zeitgewinn

Nun soll die hierarchische Struktur mit einem Level2 und einem Level1-Cache untersucht werden. Eine solche Struktur muss so konzipiert werden, dass bei ihrer Nutzung kein Datenverlust entsteht.

Ausgangspunkt jeder Aktion ist der Speicherzugriff durch den Prozessor, der die Adresse und die Übertragungsrichtung angibt. Im Einfachsystem, das bisher diskutiert wurde, bilden ein Cache und der Hauptspeicher die möglichen „Targets“ eines Zugriffs.

Im System mit einem hierarchischen Cache gibt es den Level1-, den Level2-Cache (in manchen Systemen auch ein Level3-Cache) und den Hauptspeicher als mögliche „Targets“. Ein solche Struktur hat nur dann einen Sinn, wenn der Level1-Cache früher reagiert als der Level2-Cache und wenn der Level2-Cache wiederum früher reagiert als der Hauptspeicher.

> Die Steuerung des Ablaufes eines Speicherzugriffs vollzieht sich im Prozessorkern nach den Regeln des internen Buszyklus - ein Entwurfsgeheimnis der Prozessorhersteller. Dieser entscheidet beim Entwurf, ob die zeitlich gestaffelten Antwortzeiten möglich sind und welchen Gewinn sie bringen.

Das hierarchische Cache-Konzept soll also bei einem Speicherzugriff immer den günstigsten Speicherort herausfinden: das ist derjenige mit der kürzesten Zugriffszeit. Angenommen, man geht vom Lesen aus. Bei einem Miss im Cache, der dem Prozessor am nächsten ist, soll der weiter entfernte Cache liefern, bei einem Miss dort der noch weiter entfernte Cache, bis der Hauptspeicher als endgültige und langsamste Quelle liefert.

In einem solchen Konzept entsteht ein Cacheline-Fill, also die Tatsache, dass der Hauptspeicher den Speicherzugriff befriedigt, dann, wenn keiner der „gestaffelten“ Caches einen Hit hat. Die Strategie der Ausführung des Cacheline-Fill in den gestaffelten Cache ist nun maßgeblich dafür, dass die gewünschte gestaffelte Antwort möglich ist.

Für eine gestaffelte Antwort ist es notwendig, dass die mögliche Antwort in allen gestaffelten Caches zur Verfügung steht.

Außer dem Cacheline-Fill soll es keine weitere Cache-Operation geben, die bestimmt, welche Einträge in die Caches der einzelnen Ebenen erfolgen.

Das bedeutet, dass die (beim Cacheline-Fill vom Hauptspeicher gelieferte) Cacheline nicht nur den Prozessor erreicht, sondern dass sie auch in alle gestaffelten Caches eingetragen wird.

Diese Maßnahme erzeugt eine grundsätzliche Eigenschaft: die Menge der Einträge einer bestimmten Speicher-Ebene ist eine Untermenge der Einträge in der Speicher-Ebene, die eine Stufe weiter vom Prozessor entfernt ist.

Nach dem Cacheline-Fill sind alle Cache-Einträge der adressierten Cacheline identisch. Wenn nun der Prozessor mit der eingetragenen Cacheline arbeitet und den Eintrag, der z.B. eine Variable realisiert, verändern will, dann stehen zwei Strategien zur Verfügung.
Beim Write-Through-Verfahren wird die Cacheline, die vom Prozessor adressiert wird, in jeder Ebene des hierarchischen Systems aktualisiert, d.h. geschrieben. Cachelines mit derselben Hauptspeicheradresse bleiben damit zu jeder Zeit in allen Speicherebenen gleich. Da beim Write-Through-Verfahren immer der langsamste, nämlich der Hauptspeicher, die Zugriffszeit bestimmt, entsteht beim Schreiben kein Zeitgewinn. Der Zeitgewinn entsteht nur bei den Lese-Vorgängen im Cache.

Das alternative Write-Back-Verfahren bietet die Möglichkeit, den Zeitgewinn auch beim Schreiben zu nutzen. Zunächst einmal schreibt der Prozessor nur in den Cache, der ihm am nächsten ist. Sooft er weiterhin dort zugreift, entsteht die kurze Zugriffszeit. Erst, wenn dieser veränderte Eintrag beim Cacheline-Fill überschrieben werden soll, wird der veränderte Wert in eine der weiter entfernten Speicherebenen zurückgeschrieben, und zwar vor dem Cacheline-Fill-Vorgang.

Beim Zurückschreiben wird diejenige Speicherebene gesucht, die dem Prozessor am nächsten ist und die nicht durch das nachfolgende Cacheline-Fill überschrieben wird. Solange beim Zurückschreiben eine verwendbare Speicherebene im Cache gefunden wird, ist Zeit für den Speicherzugriff kürzer als beim Zurückschreiben in den Hauptspeicher.

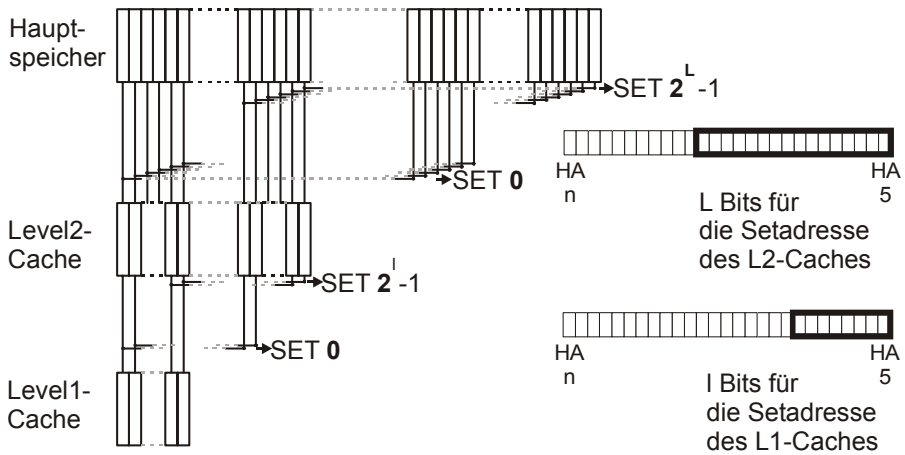
**Beispielhafte Szenarien**

Das Ergebnis eines solchen Entwurfes soll an beispielhaften Abläufen dargestellt werden.

Jede Cache-Realisierung bildet konkrete Sets mit ihren zugeordneten Cache-Zeilen.

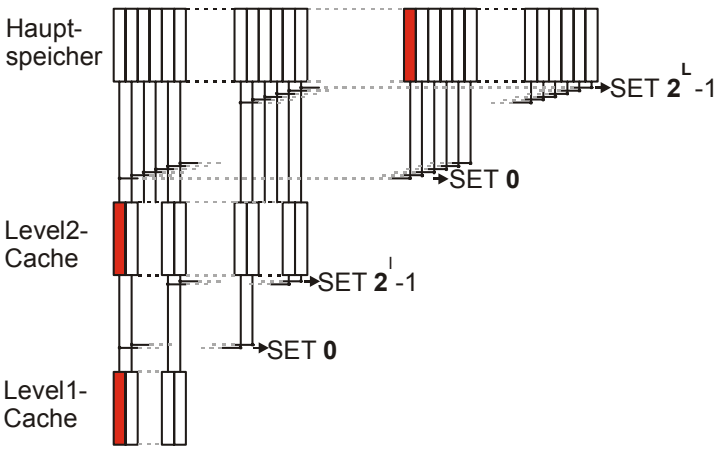
Der Level2-Cache enthält so viele Cache-Zeilen, wie die Setadressbits für den L2-Cache vorgeben.

Das gilt auch für die Sets des L1-Cache, wobei seine Sets im Level2-Cache gebildet werden.

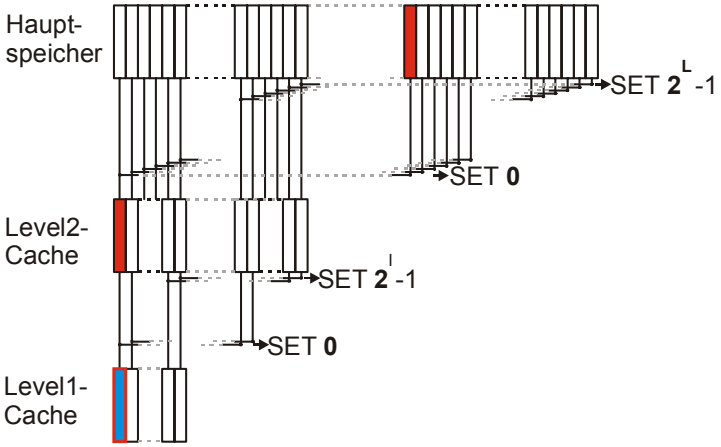


(1) Ein Cacheline-Fill aus dem Hauptspeicher sei der Beginn. Original und die Kopien in den Caches sind gleich.

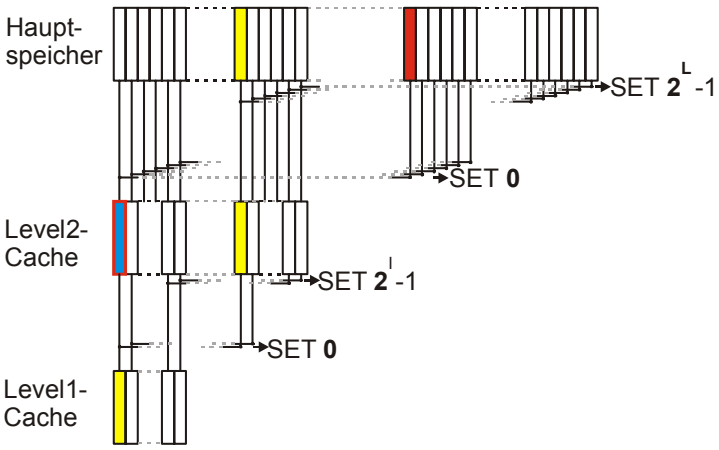
Wenn nun ein Wort im Level1-Cache geändert wird, gibt es mit Sicherheit noch die zweite Original-Kopie im Level2-Cache.



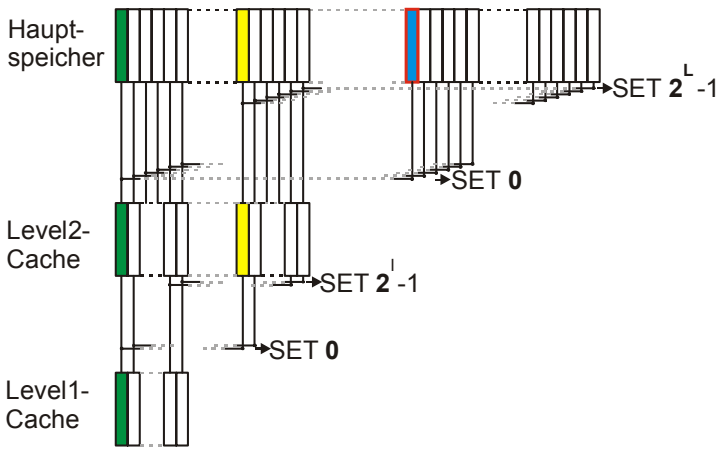

(2) Der Prozessor kann nun das Wort im Level1-Cache schreibend verändern.



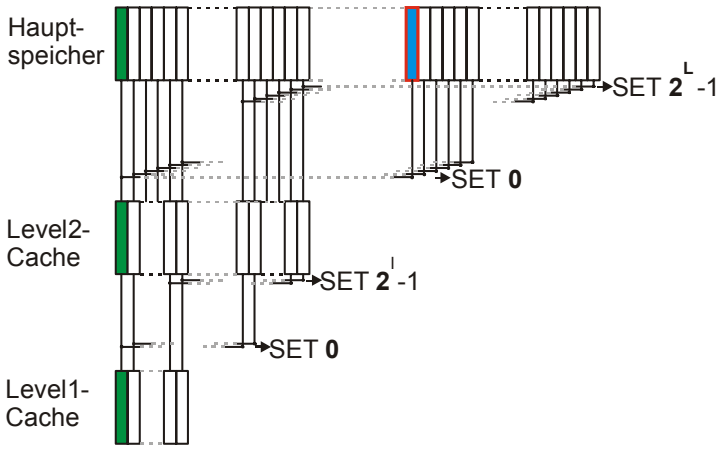

(3) Beim nächsten notwendigen Cacheline-Fill, der die geänderte Cacheline im Level1-Cache verändert, kann man also in diese Kopie zurückschreiben, aber nur dann, wenn das nächste Cacheline-Fill (wie im Beispiel angenommen) das zurückgeschriebene Wort nicht überschreiben wird.



(4) Wenn sich beim nächsten Cacheline-Fill im Level2-Cache herausstellt, dass ein zurückgeschriebenes Wort in der Cache-Zeile steht, dann wird zuerst das zurückgeschriebene Wort in den Hauptspeicher zurückgeschrieben, bevor es mit dem Cacheline-Fill verändert wird.




(5) Wenn dieser letzte grün markierte Zugriff vor dem gelb markierten in (3) gekommen wäre, dann hätte das veränderte blau markierte Wort sofort in den Hauptspeicher zurück geschrieben werden müssen, weil der Cacheline-Fill auch die Cache-Zeile im Level2-Cache überschreibt.



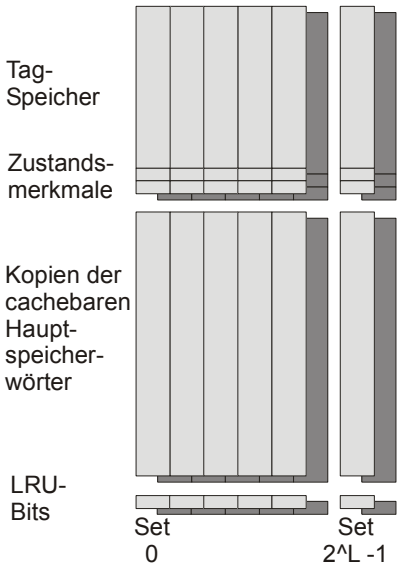

Man kann in allen Zustandsbeispielen feststellen, dass **der untergeordnete Cache immer eine Untermenge des übergeordneten Speichers enthält**. Das ist das wesentliche Merkmal dafür, dass keine Widersprüchlichkeit durch Datenverlust entsteht. Man spricht auch von der **Konsistenz** der Speicherwörter.

**Mehrwege-Caches**

Um die Chance zu erhöhen, eine Cacheline im Cache zu finden, kann man für jedes Set mehr als eine Cache-Zeile vorsehen. Man spricht dann von einem **Mehrwege-Cache**.

Mit einer Verdopplung verdoppelt sich die Chance für einen Cache-Hit.

Natürlich ist das mit Schaltungs-Mehraufwand verbunden. Beim Cacheline-Fill muss man entscheiden, welche der beiden Cachelines bei einer gegebenen Tag-Adresse überschrieben werden soll.



Man unterstützt diese Entscheidung durch die Hinzufügung eines Statusbits zu jeder Cache-Zeile. Dieses **Statusbit** gibt an, ob die Adressierung der entsprechenden Cacheline zeitlich weiter zurückliegt als die der anderen. Sobald ein Cache-Zugriff erfolgt, wird das Statusbit der adressierten Cacheline auf Null gesetzt, das Statusbit der anderen wird auf Eins gesetzt, d.h. als "**least recently used**" gekennzeichnet. Beim Cacheline-Fill wird die least recently used Cacheline überschrieben.

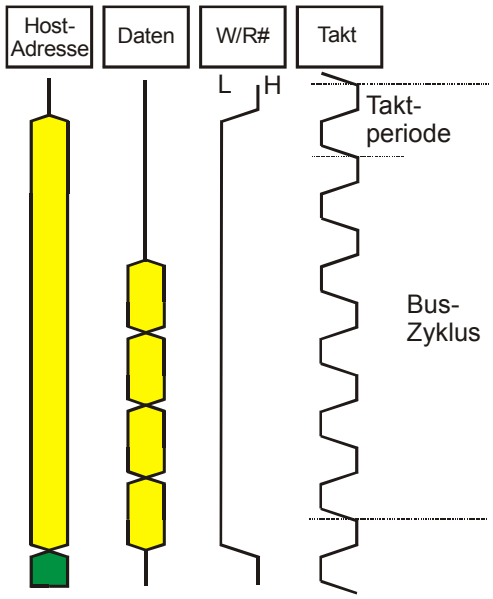
Wenn man mehr als 2 Wege vorsehen würde, müsste man den LRU-Algorithmus entsprechend anpassen.

## 7.5 Host-Bus-Konzepte

Das Host-Bus-Konzept hängt maßgeblich davon ab, ob es für ein Ein-Prozessor-System oder ein Multiprozessor-System konzipiert wird.

**Konzept mit unteilbaren Buszyklen**

Für Einprozessor-Systeme ist es typisch, dass ihr Hostbus-Protokoll auf „unteilbaren" Buszyklen aufbaut. Die Signalfolge wird so festgelegt, dass ein Einzelwort oder ein Burst von Datenwörtern in beiden Übertragungs-richtungen möglich ist.




Alle Busaktionen sind taktsynchron.

Erst wenn ein Buszyklus fertig ausgeführt ist, kann ein neuer beginnen. Das wird hier als „unteilbarer" Buszyklus bezeichnet.

In PCs ist die Host-Bridge des Chipsatzes das Partner-Target für den Prozessor. Sie bildet bei Bursts die inkrementierten Adressen, wenn das für das Speichersystem notwendig ist.

Den realen Hintergrund des Schemas kann man am Signal-Zeit-Diagramm eines der letzten großen Vertreter dieses Typs von Host-Bus-Protokoll erkennen, dem Pentium-Prozessor (Bild 7.6).

Da das Host-Bus-Protokoll für ein Mehrprozessorsystem gelten soll, muss eine Arbitrierungsphase entscheiden, welcher Prozessor den gemeinsamen Bus benutzen darf.

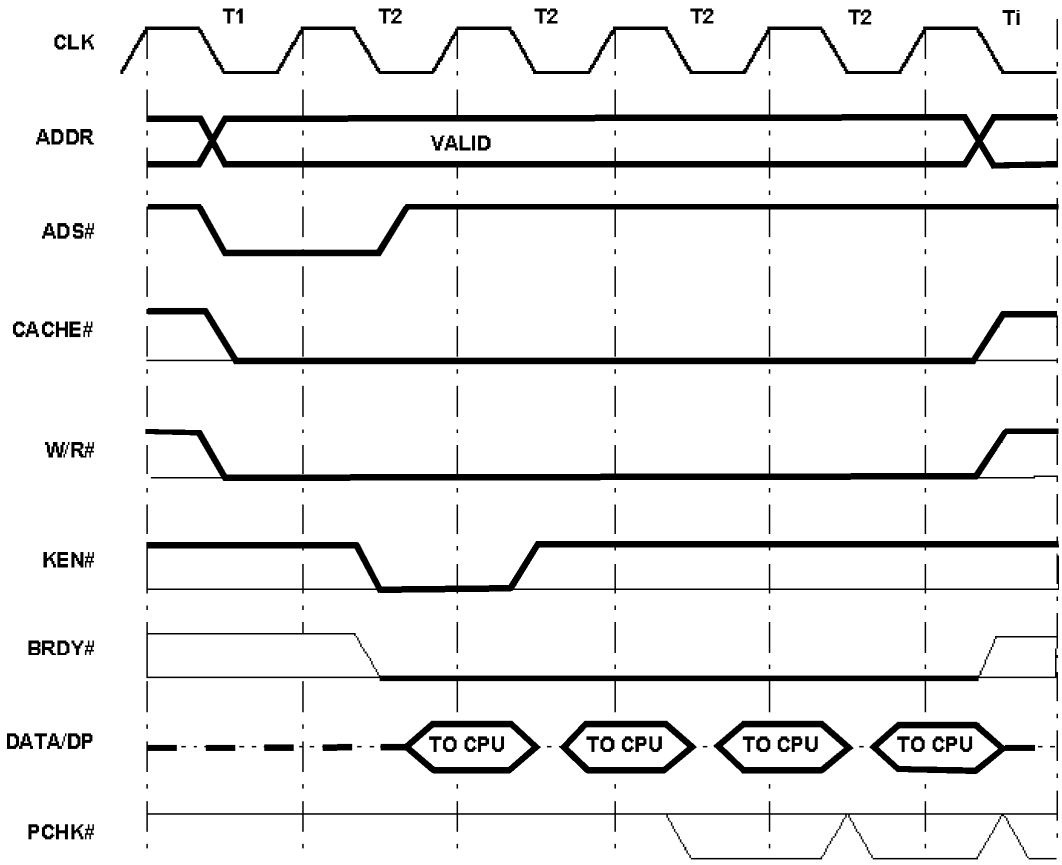



Bild 7.6: Pentium-Burst mit 4 Lese-Transfers (Datenblatt-Auszug)
Von den Steuersignalen wird nur BRDY# (bus ready) herausgegriffen: bei einem Buszyklus, der vom Prozessor erzeugt wird, kann das Partner-Target (also die Host-Bridge) BRDY# deaktivieren und solange Warte-Zyklen des Prozessors erzwingen, wie es braucht.

### Konzept mit Transaktionen und Pipeline-Struktur

Die Leistungsanforderungen für Mehrprozessorsysteme machten neue Konzepte für die Übertragung auf dem Host-Bus notwendig. Sie führten zu einer taktsynchronen, befehlsgesteuerten Arbeitsweise mit einer besonderen Verarbeitungsstruktur. Man verwendet eine Verarbeitungsstruktur, die der eines Fließbandes in der Fertigungsindustrie vergleichbar ist.

An einem Fließband wird ein Objekt nacheinander in einzelnen Fertigungsstufen bearbeitet. Die Bearbeitungszeiten der Stufen sind gleich und bestimmen den Takt, in dem ein Objekt von Stufe zu Stufe weitergereicht wird.

Im Falle der Übertragung kann man die Aktionen, die bei der Übertragung eines einzelnen Wortes oder eines Datenblockes notwendig sind, wie die Bearbeitung eines Objektes bei einem Fließband in aufeinander folgende, möglichst gleiche Phasen gliedern. Jede Phase wird von einer eigenständigen Schaltungsstufe ausgeführt, die synchron zu einem Takt beginnt und endet.

Die Phasen gliedern also den Ablauf eines elementaren Übertragungsvorganges in Abschnitte, die vom ersten bis zum letzten ablaufen müssen, um ihn insgesamt auszuführen. Der Übertragungsvorgang soll als Ganzes im folgenden **Transaktion** heißen. Beispielhaft wird jetzt eine Gliederung der Transaktionen vorgenommen, wobei davon auszugehen ist, dass mehrere Prozessoren am Host-Bus Transaktionen initiieren können (Bild 7.7).

Da das Host-Bus-Protokoll für ein Mehrprozessorsystem gelten soll, muss eine Arbitrierungsphase entscheiden, welcher Prozessor den gemeinsamen Bus benutzen darf.

Dann kann die Übertragung des Lese/Schreib-Befehls oder eines anderen folgen.
Eine Fehlerbehandlung sichert die Aktionen der Befehlsübertragung.
Ein Hit im L2/L1-Caches muss gemeldet werden.
Die Daten werden zusammen mit einer Statusmeldung des Partner-Targets übertragen.

Das Beispiel geht davon aus, dass die ersten beiden Phasen zwei Takte benötigen. Der Ansatz, alle Stufen in einer Taktzeit fertig werden zu lassen, ist nicht immer realisierbar.

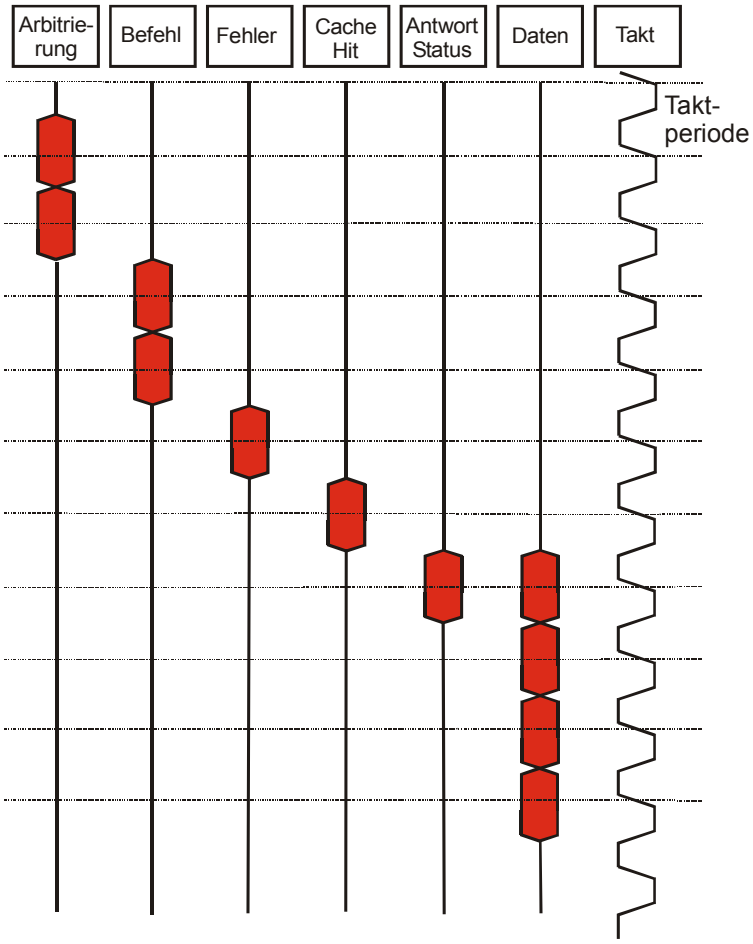



Bild 7.7: Funktionale und taktbezogene Aufteilung der Aktionen auf einem Hostbus während eines Speicherzugriffs des Host-Prozessors mit der Übertragung von vier konsekutiven Speicherwörtern

Die Schaltungsstufen werden nun im Sinne einer Fließbandstruktur (**Pipeline**-Struktur) so konzipiert, dass jede Stufe sofort eine neue Transaktion bearbeiten kann, wenn sie ihre Taktabschnitte für die aktuelle beendet hat.

Die Objekte dieses Fließbandes sind also Transaktionen. Dadurch entsteht ein Gewinn, der am größten wird, wenn die maximal mögliche Datenübertragungsrate auf dem Host-Bus entsteht. Diese entsteht, wenn zwischen den Übertragungen (einzel oder burst) keine Pausen entstehen (Bild 7.8).

Das Überlagerungsschema zeigt die Überlagerung von zwei aufeinander folgenden Transaktionen, so dass keine Pause bei der Übertragung der Datenwörter entsteht.

Würden die Anforderungen der Prozessoren mit der Regelmäßigkeit von vier Takten gestellt und würde jeder Befehl einen Burst von vier Wörtern pro Transaktion bedeuten und würden alle Antwortphasen in der kürzest möglichen Antwortzeit (wie gezeigt) beginnen, dann ergäbe sich eine Fortsetzung der gezeigten Überlagerung mit der maximal möglichen Übertragungsrate.

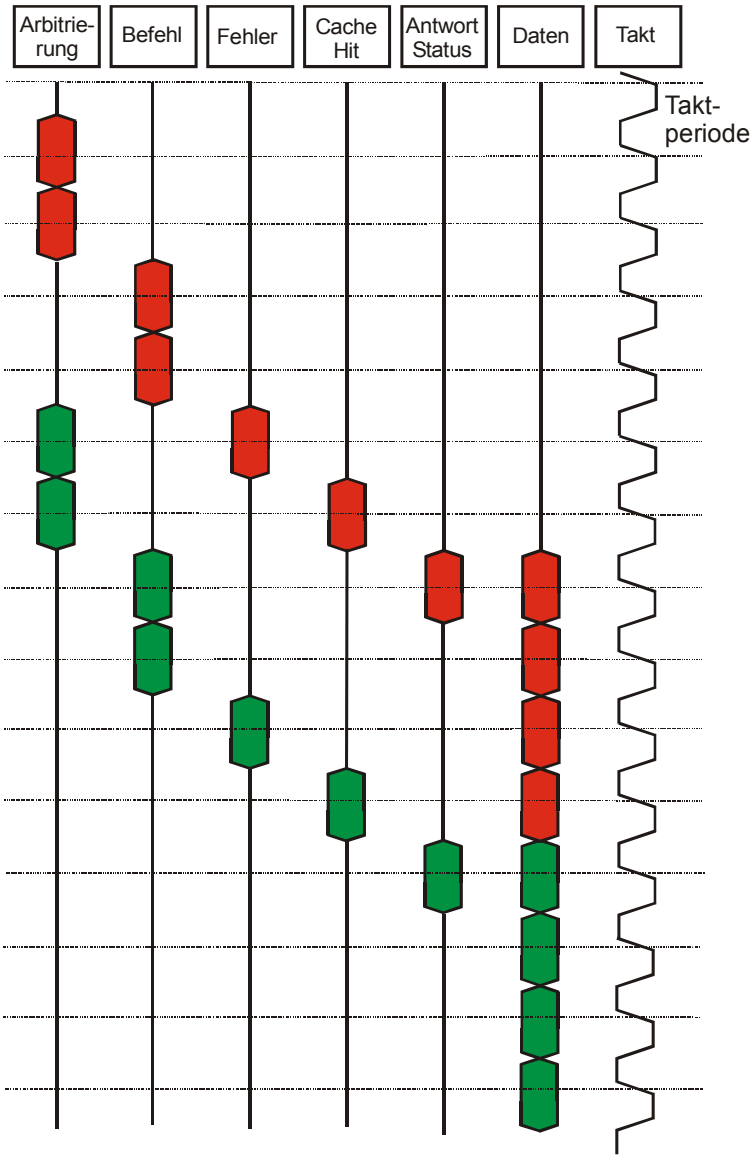



Bild 7.8: Simultanarbeit der einzelnen Pipeline-Stufen

## Die Funktionen der Signale für die einzelnen Pipelinestufen (P6-Konzept)

Um einen Eindruck von den Aktionen zu erhalten, die in den einzelnen Phasen ablaufen, sollen die Signale gruppenweise vorgestellt und ihre Verwendung erläutert werden. Das P6-Konzept gliedert in fünf sequentielle Phasen.

### Arbitration-Phase

| Pin/Signal Name | Pin Mnemonic | Signal Mnemonic |
|---|---|---|
| Symmetric Agent Bus Request | BR[1:0]# | BREQ[1:0]# |
| Priority Agent Bus Request | BPRI# | BPRI# |
| Block Next Request | BNR# | BNR# |
| Lock | LOCK# | LOCK# |

Mit den BREQ-Signalen (bus request) melden sich die Prozessoren untereinander die Anforderungen, den gemeinsamen Host-Bus zugeteilt zu bekommen. Die übrigen Signale dienen dazu, den „normalen“ Entscheidungsalgorithmus für bestimmte Zwecke abzuschalten.

### Request-Phase:

| Pin Name | Pin Mnemonic | Signal Name | Signal Mnemonic |
|---|---|---|---|
| Address Strobe | ADS# | Address Strobe | ADS# |
| Request Command | REQ[4:0]# | Request | REQ[4:0]# |
| Address | A[35:3]# | Address | A[35:3]# |
| Address Parity | AP[1:0]# | Address Parity | AP[1:0]# |
| Request Parity | RP# | Request Parity | RP# |

Die Aktivierung des Strobe-Signales ADS# (address strobe) zeigt den Beginn einer neuen Transaktion an.
Im ersten Phasentakt werden die Host-Adressbits HA3 bis HA31, im zweiten Phasentakt die Bus-Enable-Bit BE0 bis BE7 übertragen.
Die Hostadressbits sind durch Parität gesichert, und zwar sichert AP1# die Bits A[35:24]# und AP0# die Bits A[23:3]#.
Das Request Command REQ[4:0] gibt die gewünschte Busoperation an. Es wird in der ersten Transaktionsphase gesendet und kann in der zweiten ergänzt werden. Die Sicherung erfolgt mit RP#.

### Completion oder Snoop-Phase

| Type | Signal Names |
|---|---|
| Keeping a Non-Modified Cache Line | HIT# |
| Hit to a Modified Cache Line | HITM# |
| Defer Transaction Completion | DEFER# |

Der Memory-Controller in der Host-Bridge wird darüber informiert, ob ein Read- oder Write-Hit im Prozessormodul vorliegt. Er kann seine Aktionen entsprechend einstellen.
Das Defer-Signal dient einem speziellen Verzögerungskonzept

**Response-Phase**

| Type | Signal Names |
|---|---|
| Data Ready | DRDY# |
| Data Bus Busy | DBSY# |
| Data | D[63:0]# |
| Data ECC Protection | DEP[7:0]# |

In der Request-Phase wird ein Partner-Target adressiert. Dieses antwortet in der Reponse-Phase, indem es den Bearbeitungszustand der Transaktion in RS[2:0] meldet.
DREADY# gibt dem adressierten Partner-Target die Möglichkeit, die Übernahme von Daten zu verzögern.
Die Datenbits sind durch einen 1-Fehler-korigierbaren Kode gesichert DEP[7:0].
Mit DRDY# ist eine Verzögerung möglich.
DBSY# gibt dem augenblicklichen Busagenten die Möglichkeit, die Kontrolle über den Host-Bus zu behalten.

## 7.6 Die Host-Bus-Arbitrierung

**Prinzipien der Hostbus-Arbitrierung für Mehrprozessorsysteme (P6-Konzept)**

Man hat die Wahl zwischen einem Verfahren mit zentraler Entscheidungsschaltung (wie z.B. beim PCI-Bus) oder einer verteilten Entscheidung in jedem Prozessor.

Bei einem zentralen Verfahren hat man immer den Nachteil der Signallaufzeit zwischen der Arbiterschaltung und dem Anforderer. Da es schnell gehen soll, sucht man nach einer alternativen Lösung, die die mit einem Hand-Shake verbundenen Signallaufzeiten vermeidet.

Die Alternative zur zentralen Arbitrierungsschaltung ist die dezentrale Arbitrierung durch funktional gleiche Arbiterschaltungen in den Prozessoren. Jeder Prozessor enthält eine Einheit, die für ihn die Benutzung des Host-Bus organisiert. Diese enthält die Arbiterschaltung. Die Einheit wird im folgenden **Bus-Agent** genannt.

Damit kein zeitlicher Versatz zwischen den unabhängig voneinander wirkenden Arbiterschaltungen entsteht, müssen sie ihre Arbeitsphasen streng synchron halten. Das bedeutet, dass sie sich auf ein gemeinsames Taktsignal synchronisieren.

Folgende Bedingungen für die Funktionen des Arbiters sollen gelten:
Jede Arbiterschaltung soll durch entsprechende Signale wissen, welche Requests anliegen.
Jede Arbiterschaltung soll daraus eine eigene Entscheidung bilden. In fehlerfreien Fall wird eine einzige Arbiterschaltung auf Buszugriff entscheiden, die anderen werden auf Nicht-Zugriff entscheiden. Ein Widerspruch ist ein entdeckter Fehler.
Die Arbiterschaltungen und damit die Prozessoren sollen beim Zugriff die gleiche Chance erhalten.

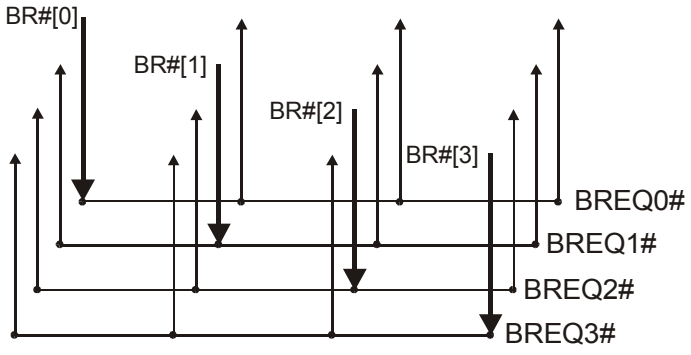
Die Arbiterschaltungen sollen der Reihe nach die Chance zur Busbenutzung bekommen. Dann werden sie gleich behandelt. Es muss also eine zyklische Chancenvergabe realisiert werden.

Der Realisierungsansatz geht von einem modulo 2^n -Dualzähler aus. Dieser hat die Eigenschaft, nach 2^n Zyklen wieder von vorne zu beginnen.
Man kann 2^n solcher Zähler gleichzeitig betreiben, wobei alle zum gleichen Zeitpunkt weiter zählen sollen.
Gleichgültig mit welchen Anfangswerten man sie startet: jeder wird nach 2^n Takten wieder die Anfangskombination erreichen.
Sind die Anfangswerte zyklisch verschoben, dann schieben sich die Anfangswerte durch die Kette, wie folgende Tabelle beispielhaft zeigt, die die Zählerzustande in vier Busagenten mit den Nummern 0 bis 3 taktsynchron angibt.

| Takt | Zähler 0 | Zähler 1 | Zähler2 | Zähler3 |
|---|---|---|---|---|
| Anfang | 0 | 3 | 2 | 1 |
| 1.Takt | 1 | 0 | 3 | 2 |
| 2.Takt | 2 | 1 | 0 | 3 |
| 3.Takt | 3 | 2 | 1 | 0 |
| 4.Takt | 0 | 3 | 2 | 1 |

Damit hat man ein Verfahren der Buszuteilung, das einige der gestellten Bedingungen erfüllt. Angenommen, dass der Bus zugeteilt wird, wenn eine Anforderung anliegt und der Zählerstand 0 ist. Dann erkennt man aus dem Schema der obigen Tabelle, dass jeder Bus-Agent zyklisch einmal eine Zugriffs-Chance erhält.

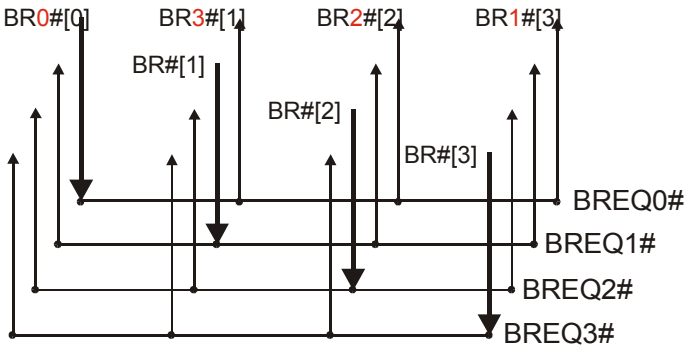
Die signaltechnische Umsetzung muss nun für die Erfüllung der anderen Bedingungen sorgen.
Wenn jeder Bus-Agent von jedem anderen wissen soll, ob er eine Anforderung hat, dann erzwingt das einen entsprechenden Signalaustausch.




Lies: BR#[i] Bus Request, den Agent i sendet,
BREQi# Signalleitung Bus Request i, die den eingeprägten Bus Request i verteilt.

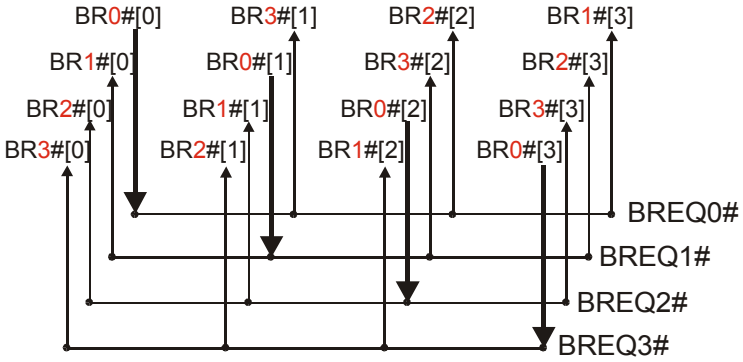
In jedem Takt soll jeder Bus-Agent genau wissen, welcher gerade einen Request haben muss.
Beispiel: In dem Takt, in dem der Agent 0 den Zählerstand 0 hat, hat Agent0 die Chance zur Busbelegung. Agent 1 hat den Zählerstand 3, Agent 2 hat den Zählerstand 2 und Agent 3 hat den Zählerstand 1. Das Requestsignal von Agent 0 soll nun so zu den Signaleingängen der anderen Agenten geführt werden, dass diese vom eigenen Zählerstand exakt auf den Eingang schließen können, wo das Requestsignal des Agenten 1 ankommt.

Agent 1 fragt den Eingang ab, der beim Zählerstand 3 abzufragen ist und den Bus Request 0 erfasst, hier BR3#[1] genannt.
Agent 2 fragt den Eingang ab, der beim Zählerstand 2 abzufragen ist und den Bus Request 0 erfasst, hier BR2#[2] genannt.
Agent 3 fragt den Eingang ab, der beim Zählerstand 1 abzufragen ist und den Bus Request 0 erfasst, hier BR1#[3] genannt.



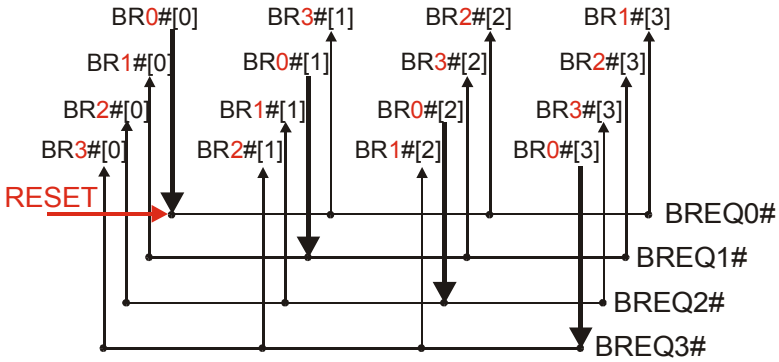

Die anderen Kombinationen ergeben sich sinngemäß.

Jetzt kann jeder Bus-Agent in jedem Takt eindeutig die Requests aller anderen empfangen und einen Widerspruch feststellen.




Es bleibt noch das Problem zu lösen, wie man nach dem Reset die Zählerstände der Agenten einstellt.

Das P6-Konzept löst das so, dass die Reset-Schaltung die Signalleitung BREQ0# aktiviert. Da die Eingänge implizit bestimmten Zählerständen zugeordnet sind, kann man daraus die einzustellenden Zählerstände erkennen und einstellen.




Jede Arbitrierungsschaltung „weiß“ in jedem Takt genau,
wie der Anforderungszustand der anderen ist und
welche andere zu diesem Zeitpunkt den Bus belegen darf und
wann sie frühestens in der Reihenfolge bedient werden kann.
Wie diese gegenseitige Information im P6-Konzept zur Überwachung auf Ablauffehler genutzt wird, bleibt Geheimnis des Herstellers Intel.

Die gegenseitige Meldung des Anforderungszustandes ist aber auch für den normalen Betrieb sehr wichtig: nur so bemerkt jeder, dass ein bestimmter Bus-Agent keine Anforderung hat. Das gibt für den Schaltungsentwurf die Möglichkeit, dass alle beim nächsten Takt sofort zur Abfrage des nächsten übergehen.

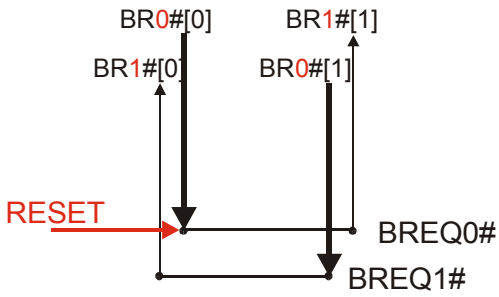
### Vereinfachung für Doppelprozesorsysteme

In Doppelprozessor-Systemenwird eine auf 2 reduzierte Form der Arbitrierung eingesetzt.

| Takt | Zähler 0 | Zähler 1 |
|---|---|---|
| Anfang | 0 | 1 |
| 1.Takt | 1 | 0 |
| 2.Takt | 0 | 1 |

Die Prinzipien sind die gleichen wie beim 2^2-System.

Man stellt die gleiche rotierende Priorität wie dort fest.




### Ausnahmen von der normalen Arbitrierung

Das Signalschema in Bild 7.9 ergänzt die Ausnahmesignale, wobei der Einfachheit halber das Signalschema für ein Doppelsystem angenommen wird.

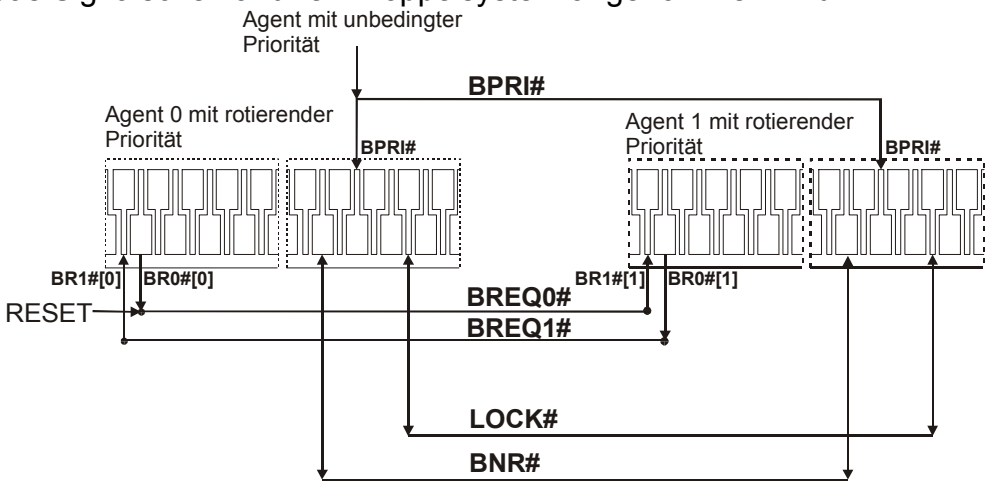



Bild 7.9: Signalschema für die Arbitrierungssignale in Doppel-Prozessor-Systemen

Die erste Ausnahme ist der „unbedingte“ Vorrang beim Arbitrieren, der durch das Signal BPRI# (bus priority) angezeigt wird. Derjenige Busagent, der es sendet, kann damit das normale Verfahren zu seinen Gunsten abschalten.

Wenn ein Prozessor den Host-Bus belegt hat und eine nicht unterbrochene Folge von Bustransaktionen durchführen will (z.B. beim Lesen und Verändern einer Semaphore), kann er die Arbitrierung durch Aktivierung von LOCK# kurz aussetzen.

Jeder Prozessor kann das Signal BNR# (block next request) sozusagen als „Not-bremse“ aktivieren. Wie gleich ausführlich gezeigt wird, kann es ein Ungleichgewicht zwischen den geforderten und den ausführbaren Transaktionen geben, das ein Warteschlangenkonzept erfordert. Dabei können Warteschlangenpuffer vollständig belegt werden. Dann muss eine weitere Belegung blockiert werden, bis wieder freier Platz da ist. Das Blockieren wird mit BNR# gesteuert.

**Das Warteschlangenkonzept bei der Belegung des P6-Host-Bus**

Anforderungen zur Belegung des Host-Bus entstehen gemäß dem Fortschritt des Programmablaufes in den einzelnen Prozessoren, d.h. sie entstehen kaum regelmäßig. Die größten Anforderungen stellt der Fall dar, dass alle Prozessoren gleichzeitig Anforderungen stellen.

Das Problem soll am vereinfachten Beispielfall mit zwei Anforderungen deutlich gemacht werden (Bild 7.10).

Wenn zwei Prozessoren eine Anforderung auf Belegung des Host Bus haben und die erste Anforderung wird erfasst, dann wird die entsprechende Transaktion begonnen. Sobald die Arbitrierungsstufe wieder frei ist, wird die Anforderung des in der Reihenfolge folgenden erfasst und die entsprechende Transaktion wird auch begonnen. Diese zweite Transaktion wird in dem Moment nicht mehr fortsetzbar sein, wenn ihre Antwortphase beginnen soll. Denn sie trifft auf die noch nicht beendete Antwortphase der vorhergehenden Transaktion.

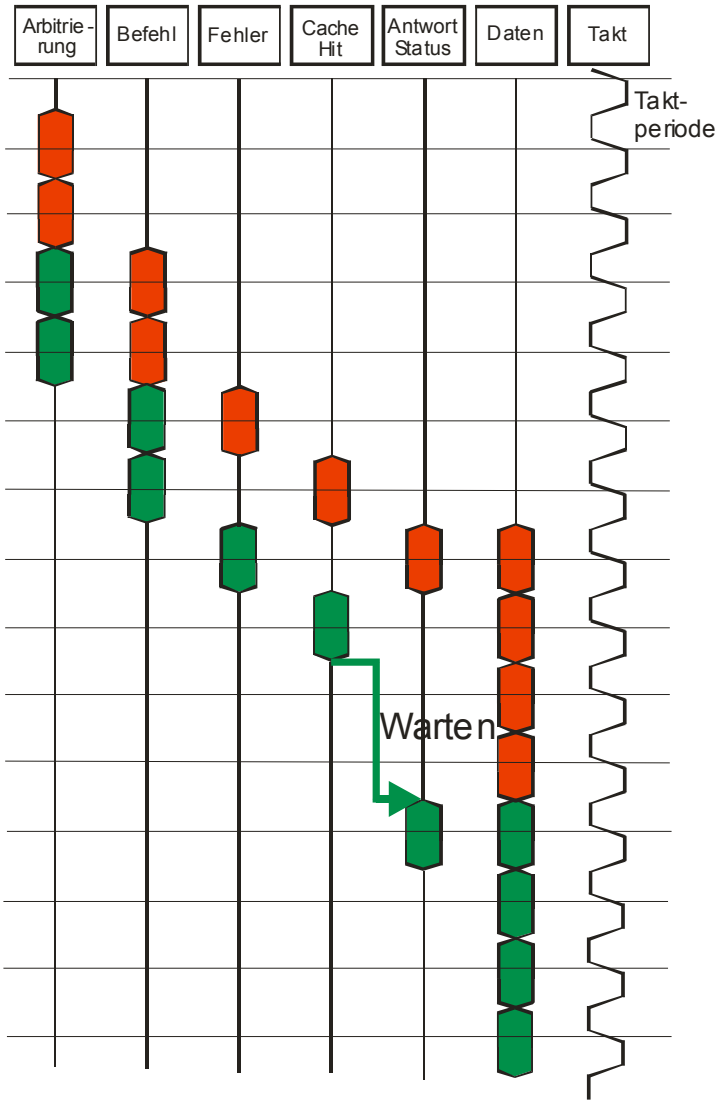



Bild 7.10: Warten innerhalb einer „aufgelaufenen“ Transaktion

Die nachfolgende Transaktion muss mit ihrer Antwortphase warten, bis die vorhergehende abgeschlossen ist. Die Warteschlange enthält ein Element.

Würden noch eine dritte und eine vierte Transaktion begonnen, weil die entsprechenden Prozessoren eine Anforderung haben, würde die Warteschlange zwei weitere Elemente aufnehmen.

Die Busagenten müssen sich also die wartenden Transaktionen in der Reihenfolge ihrer Blockade als **Warteschlange** merken.

Das geschieht mit Hilfe von geeigneten Puffern, die die Merkmale der wartenden Transaktionen enthalten. Wenn die erste wartende Transaktion deblockiert und damit fortgesetzt wird, wird der entsprechende Platz im Puffer frei.

Wenn die Puffer voll sind, muss eine weitere Aufnahme von Transaktionsmerkmalen verhindert werden. Das geschieht mit Hilfe des Signales *Block Next Request,* bis belegbarer Pufferplätze frei werden.

## 7.7 Cache-Kohärenz bei dma-Zugriffen

Das Problem wird in Anlehnung an http://www.iam.unibe.ch/~rvs/lectures/ra/ra_7.pdf diskutiert.

### Das Cache-Kohärenzproblem

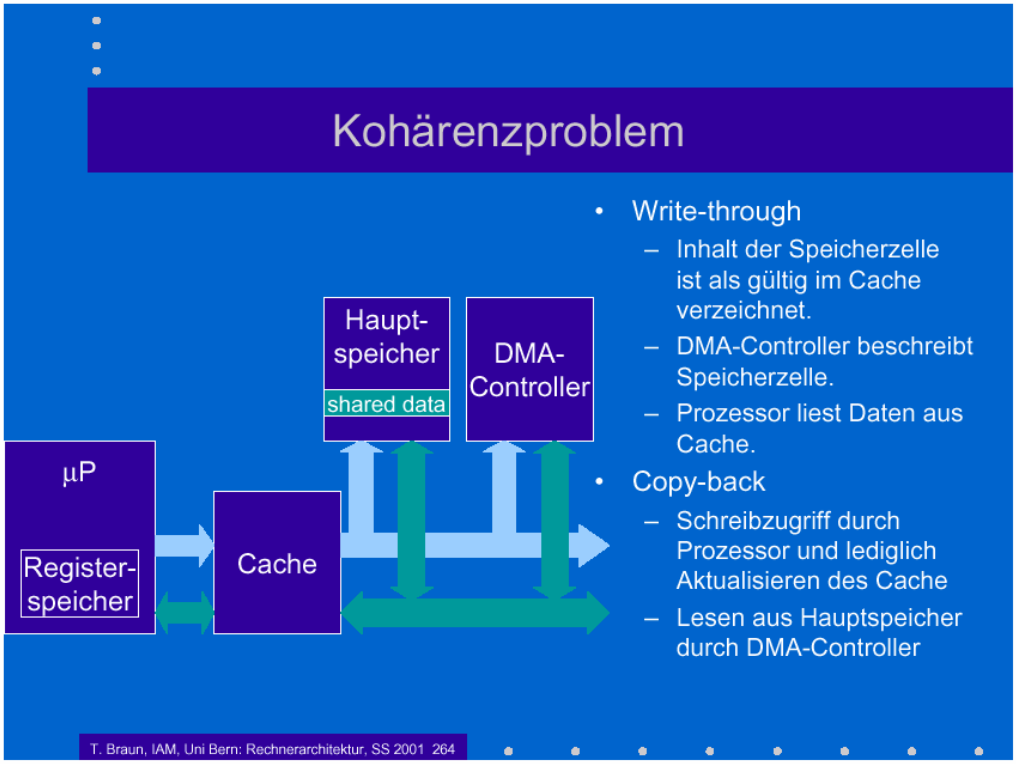



Bild 7.11: Bedingungen für Inkohärenz
Write-back wird hier als Copy-back bezeichnet. - Mit DMA-Controller wird hier die Einheit bezeichnet, die die den direkten Speicherzugriff in der zentralen Verteilereinheit ausführt, die man mit Memory Controller Hub oder Host-Bridge bezeichnet.

Bild 7.11 gibt zwei Inkohärenz-Fälle an:

- Wenn der Cache-Controller im Write-Through-Betrieb arbeitet, werden alle schreibenden Veränderungen von Variablen sowohl im Cache als auch im Hauptspeicher ausgeführt. Eine schreibende Veränderung durch einen dma-Zugriff wird aber nur im Hauptspeicher ausgeführt. Ein lesender Zugriff bekommt den „veralteten“ Wert aus dem Cache und nicht den aktuelleren Wert aus dem Hauptspeicher.
- Wenn der Cache-Controller im Write-Back-Betrieb arbeitet, werden alle schreibenden Veränderungen von Variablen nur im Cache ausgeführt. Ein lesender dma-Zugriff zum Hauptspeicher bekommt nicht den aktuellen Wert, der in der Kopie im Cache steht, sondern den „veralteten Wert“ im Hauptspeicher.

Es gibt mehrere Alternativen, das Kohärenz-Problem zu lösen:

- Man schließt Hauptspeicherbereiche (hier Seiten genannt, weil der Hauptspeicher in gleichlange Seiten eingeteilt wird), die der dma-Änderung unterliegen, von der Zwischenspeicherung im Cache aus – und verzichtet damit auf die Vorteile der Zwischenspeicherung, wenn diese Seiten adressiert werden.
- Im Falle von dma-Zugriffen macht man die Kopien im Cache ungültig (bei write through) bzw. man räumt den Cache zuerst aus (bei write back).
- Man hört auf dem Host-Bus mit und meldet den Zustand der betroffenen Cachelines, so dass in anderen Bus-Agenten eine evt. Inkohärenz festgestellt werden kann: bus snooping.

Folgende Tabelle stellt die Alternativen zur Lösung des Cache-Kohärenzproblems zusammen:

| Lösungen des Kohärenzproblems: | |
|---|---|
| 1. Kennzeichnung der gemeinsamen Seiten als non-cacheable | |
| | Cache-Controller wird für die gekennzeichneten Seiten nicht aktiv |
| | Kennzeichnung auch für den Adressbereich von E/A-Controllern (Register) |
| 2. Cache-Clear/Cache-Flush | |
| 3. Bus-Snooping | |

Das Bus-Snooping-Verfahren wird in den Intel-Prozessoren angewendet, die hier als Beispiel dienen. Deshalb wird es hier weiter diskutiert.

**Bus-Snooping**

Die Erklärung wird ganz allgemein für Multiprozessorsysteme gegeben. Jeder Prozessor verfügt dabei über eigene Caches.

Aller Datenverkehr zwischen Prozessoren und zwischen diesen und dem Hauptspeicher bzw. der Peripherie läuft über den Hostbus.
Sobald ein Prozessor zum Hauptspeicher zugreift, ist nicht sicher, ob ein anderer Prozessor in seinem Cache eine Kopie der adressierten Cacheline hat. Wenn er sie hat und sie ist im Zustand modifiziert, dann muss diese modifizierte Cacheline zum nachfragenden Prozessor geliefert werden und nicht die nicht modifizierte im Hauptspeicher.

Der andere Prozessor kann aber nur dann eine Aussage darüber machen, dass er die adressierte Cacheline hat, wenn er die Adresse empfangen und die Übereinstimmung zwischen dem Tag und entsprechenden Hostadressbits feststellen kann. Den Zustand, dass er die Adressierung über den Hostbus mitbekommt, nennt man Snooping-Zustand.

Am Hostbus des P6-Konzeptes sind alle Prozessoren, die nichts mit der gerade auf dem Host-Bus ablaufenden Transaktion zu tun haben, im Snooping-Zustand.

Die dma-Controller-Funktion ist eine Teilfunktion der Verteilereinheit (Host-Bridge), die neben den Prozessoren mit ihren Bus-Agenten auch ein Teilnehmer am Host-Bus ist. Wie die Prozessoren mit ihren Bus-Agenten hat auch die Verteilereinheit ein Interface zum Hostbus.

Das bedeutet, dass sie wie ein Prozessor ein Memory Read oder ein Memory Write auf dem Hostbus erzeugen kann, während die übrigen Prozessoren im Snooping-Zustand sind.

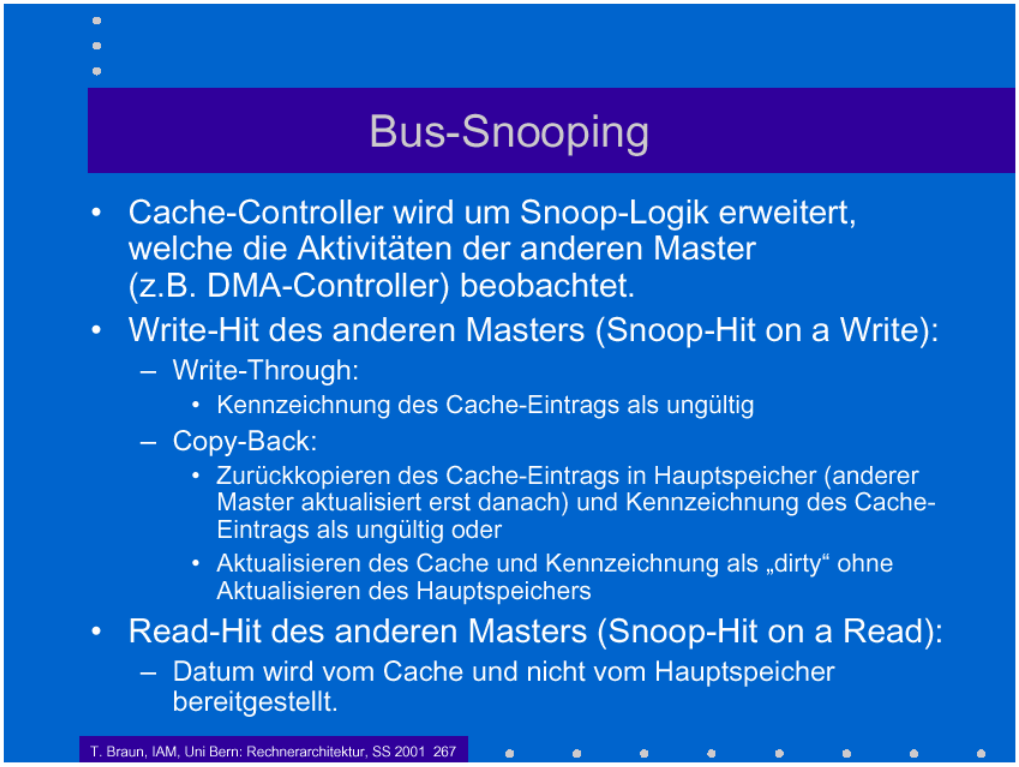



Bild 7.12: Merkmale des Bus-Snooping im Überblick

Die Merkmale des Bus-Snooping kann man an folgenden Fällen erläutern:

2. Fall: Die dma-Controller-Funktion in der Verteilereinheit führt ein Write aus, weil z.B. ein PCI-Controller einen dma-Transfer von der Disk in den Hauptspeicher ausführt. In jedem Fall kann er die Speicherzelle adressieren, in die er speichern kann. Das wird oben als „Write hit des anderen Masters“ ausgedrückt. Gleichzeitig zum Hauptspeicherzugriff wird mit derselben Adresse eine Write-Transaktion auf dem Hostbus ausgeführt. Wenn nun einer der am Hostbus „snoopenden“ Prozessoren eine Cacheline hat, die zu der angelegten Adresse passt (snoop write hit), und sie ist modifiziert, entsteht ohne besondere Maßnahmen eine Inkohärenz.
Es werden nun zwei Vorschläge gemacht, das zu beheben.
Erst erfolgt ein Write-back, dann erfolgt das Aktualisieren durch den dma-Controller, dann erfolgt ein Ungültig-Machen der Cacheline im betroffenen Prozessor, oder
Es erfolgt das Aktualisieren der Cacheline im betroffenen Prozessor mit dem Setzen des Modifiziert-Zustandes. Dadurch besteht später die Möglichkeit zum Write back.

2. Fall: Die dma-Controller-Funktion in der Verteilereinheit führt ein Read aus, weil z.B. ein PCI-Controller einen dma-Transfer vom Hauptspeicher zur Disk ausführt. In jedem Fall kann er die Speicherzelle adressieren, aus der er lesen will. Das wird oben als „Read hit des anderen Masters“ ausgedrückt. Gleichzeitig zum Hauptspeicherzugriff wird mit derselben Adresse eine Read-Transaktion auf dem Hostbus ausgeführt. Wenn nun einer der am Hostbus „snoopenden“ Prozessoren eine Cacheline hat, die zur Adresse passt (snoop read hit) und sie ist modifiziert, entsteht ohne besondere Maßnahmen Inkohärenz.

Die einfache Lösung: der dma-Controller verwirft das Ergebnis des Hauptspeicherzugriffs und gibt als Leseergebnis das weiter, was über den Hostbus kommt.

Das Protokoll des P6-Hostbus sieht nach der Request- und Error-Phase eine spezielle Snoop-Phase vor. In dieser Phase dienen zwei Signale der Meldung des Cacheline-Zustandes: HIT, wenn eine nicht modifizierte Cacheline da ist, und HITM, wenn eine modifizierte Cacheline da ist. Das Signal HITM, das in der Snoop-Phase geliefert wird, ist also maßgebend für die Maßnahmen in der Verteilereinheit, die Cache-Kohärenz zu gewährleisten.

Damit die Verteilereinheit die Host-Bus-Transaktion, die den dma-Zugriff begleitet, möglichst unverzögert ausführen kann, muss ihr Request anders behandelt werden als die Requests der Prozessoren. Sie muss außerhalb der Arbitrierungsordnung für die Prozessoren anfordern und bewilligt werden können.
Dazu dient im P6-Host-Bus-Konzept das Signal BPRI (*bus priority*). Die Verteilereinheit kann damit den unbedingten Zugriff zum Host-Bus erzwingen, wenn der Bus frei geworden ist.

**Steuerung der Kohärenz im Cache-Controller eines Prozessors**

Das Verhalten der Cache-Controller in den beispielhaften Intel-Prozessoren wird durch das sog. MESI-Protokoll festgelegt.

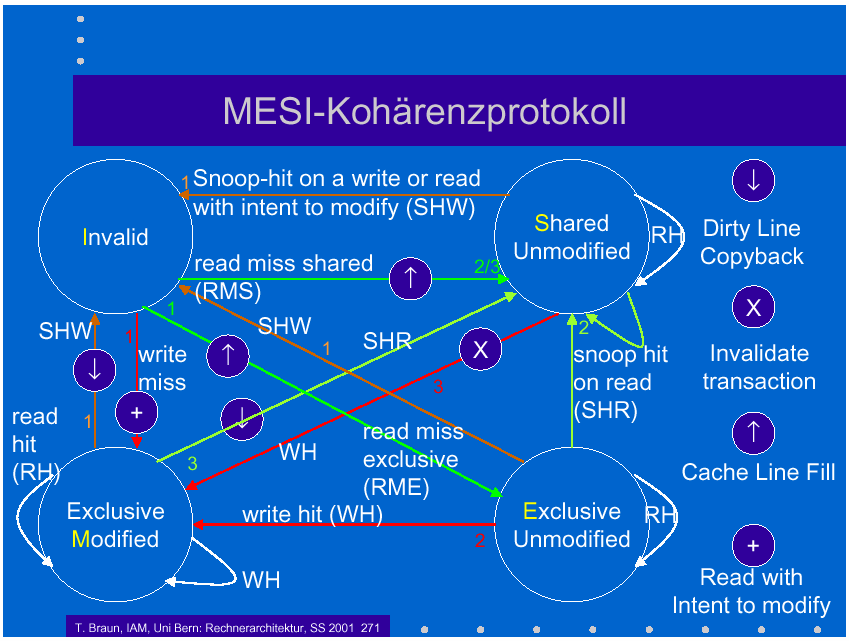



Bild 7.13: Zustände und Zustandsübergänge gemäß MESI-Protokoll

Inkohärenz, die mit dma-Zugriffen zusammenhängt, entsteht dann, wenn modifizierte Cachelines betroffen sind. Man kann sich also auf den Zustand *modified* konzentrieren. Dort interessieren nur die Übergänge, die mit einem Snoop Hit zusammenhängen, als SHW (snoop hit on a write) und SHR (snoop hit on a read).

Im Bild 7.13 kann man sich auf den Übergang SHW (1orange) und den Übergang SHR (3grün) am Zustand *Exclusive Modified* konzentrieren.

Bei beiden Übergängen ist angegeben, dass zuerst ein Write-back erfolgt, bezeichnet als *Dirty Line Copyback*.

Das hat Konsequenzen für das Konzept die dma-Controller-Funktion in der Verteilereinheit:
Wenn die Verteilereinheit die Transaktion ausführt, die einen dma-Zugriff begleitet, und in der Cache/Snooping-Phase wird ein HITM aktiviert, dann wird in der Antwort-Phase die modifizierte Cacheline geliefert.

- Bei einer Write-Transaktion kann die Verteilereinheit das, was geliefert wird, einfach verwerfen, weil der gültige Wert derjenige ist, den die Peripherie liefert.
  Wichtig ist eigentlich nur das, was im Cache des Prozessors geschieht, nämlich die Umschaltung der Cacheline in den *invalid*-Zustand.

- Bei einer Lese-Transaktion ist die in der Antwort-Phase gelieferte modifizierte Cacheline die entscheidende Größe. Die Verteilereinheit muss diese Cacheline sowohl zur Peripherie als auch in den Hauptspeicher weitergeben.

Im Prozessor erfolgt beim Lesen das Umschalten der Cacheline in den Zustand *unmodified* (weil ja das Write-back in den Hauptspeicher endgültig erfolgte) kombiniert mit *shared*, weil man in diesem Protokoll davon ausgeht, dass der Prozessor die Cacheline sowohl für eine Verteilereinheit als auch für einen anderen Prozessor liefern kann.

Bus-Snooping erfordert ein angepasstes Verhalten sowohl im Cache-Controller der Prozessoren als auch in den Controller-Funktionen in der Verteilereinheit.