3.7 Die Speicherhierarchie

Zur Vorlesung Rechenanlagen

SS 2019

Allgemeines

Bisher haben wir nur sehr kleine, schnelle Speicher für Programme und Daten betrachtet (Harvard Architektur).

Problem:

Große Speicher haben zu hohe Zugriffszeiten,

Kleine Speicher schränken den Raum möglicher Anwendungen zu sehr ein.

Ziel:

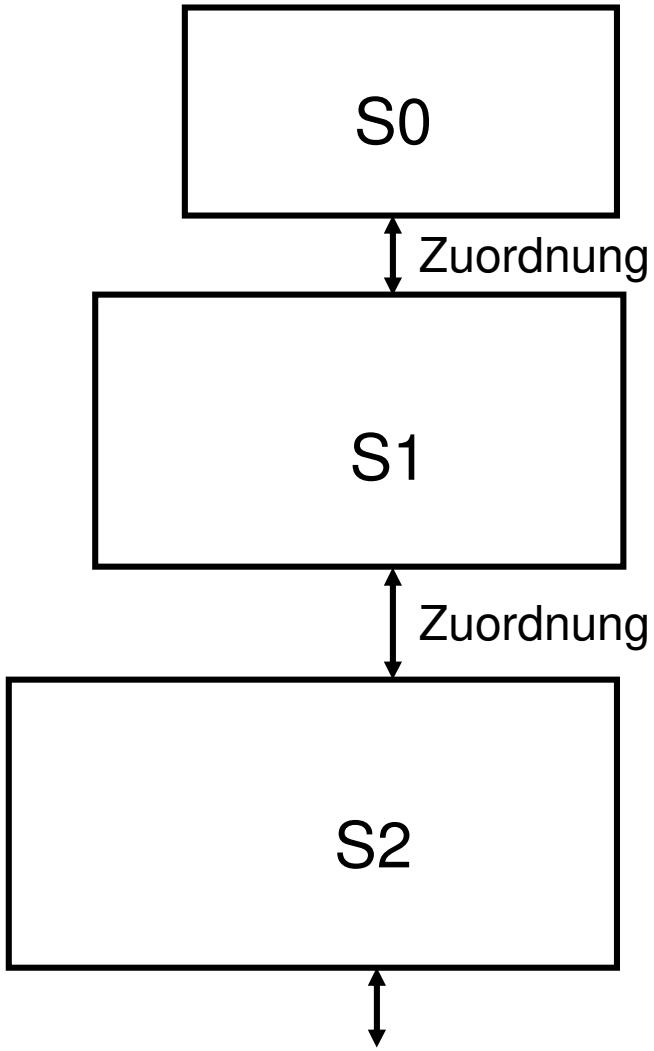
Realisiere schnelle Befehlsausführung (kleine Speicher)

und unterstütze viele Anwendungen (großer Speicher)

Lösung:

Konstruiere eine Hierarchie von Speichern

Speicherhierarchie



Schnelle, teure Speicher:
SRAM Caches

Langsamere, billigere Speicher:
DRAM Hauptspeicher

Langsame, billige Massenspeicher:
Festplatten, CD RW, DAT, ...

Speicherhierarchie ff

Mit zunehmender Tiefe in der Hierarchie, wächst die Zugriffszeit und sinken die Kosten pro Bit Speicherplatz

Probleme:

- Erhalte eine konsistente Zuordnung zwischen den Speichern
- Erziele eine möglichst niedrige, mittlere Zugriffszeit
- Benutze möglichst einheitliche Adressen.

Zeitliche und räumliche Lokalität

Wie erreicht man eine niedrige „mittlere“ Zugriffszeit?

Wir legen dem Verhalten typischer Programme einfach folgende Annahme zugrunde:

Annahme: (Prinzip der zeitlichen und räumlichen Lokalität)

Daten und Befehle, die in einem kurzen Zeitintervall in der Maschine bearbeitet werden liegen im Speicher meistens nahe benachbart beieinander.

Solange man durch die zu konstruierende Zuordnung nicht verhindert, dass nah benachbarte Daten oder Befehle nicht gleichzeitig in einem Speicher niedriger Hierarchie liegen dürfen, sollte die Zugriffszeit im Mittel sehr niedrig sein.

Zeitliche und räumliche Lokalität ff

Bemerkungen:

Man kann in Kenntnis der Zuordnung natürlich Programme stets vorsätzlich so schreiben bzw. organisieren, dass sie möglichst hohe Zugriffszeiten liefern.

Es wird offenkundig, dass dem Compiler eine wichtige Aufgabe beim Planen und der Vergabe des Speichers zukommt, und eine genaue Kenntnis der Speicherorganisation der Zielmaschine erforderlich ist.

Architektur und Compiler stehen bei der Entwicklung neuer Maschinen in untrennbarer Wechselwirkung.

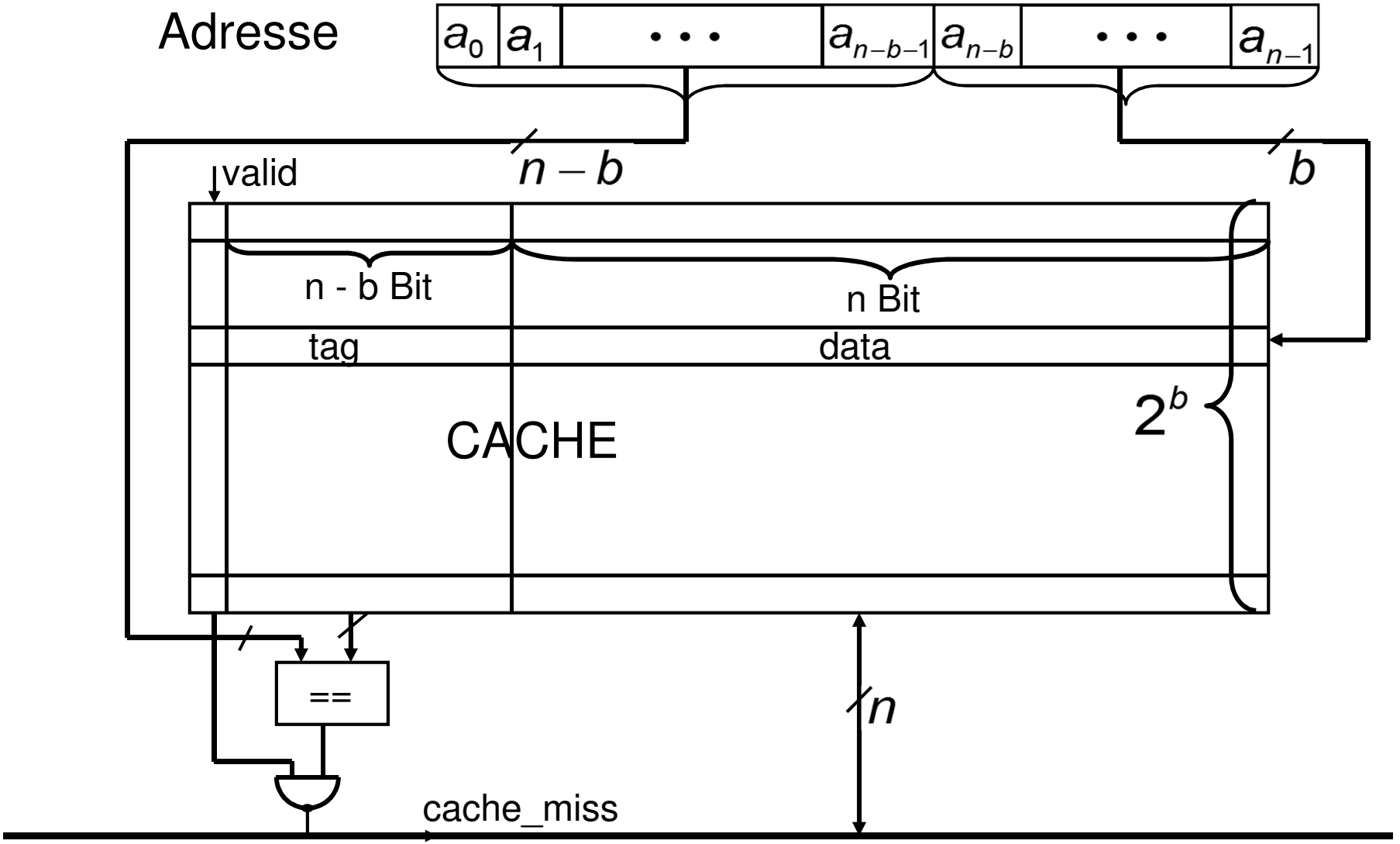
Caches

Wir wollen das Zusammenwirken zwischen Caches und Hauptspeicher zunächst in den Vordergrund stellen.

Eine sehr billige und weitverbreitete Technik ist die der sogenannten **direkten Zuordnung**.

Gegeben sei ein wortorientierter Cache der Adressbreite $b < n$. Wir realisieren die direkte Zuordnung nach folgendem Schema:

Cache: direkte Zuordnung



Direkte Zuordnung ff

Wir speichern neben einem Datenwort auch die führenden *n-b* Bit seiner Adresse als sogenanntes tag field im Cache. Bei einem Zugriff auf den Cache mit *b*-bit Adressraum vergleichen wir stets das tag Feld mit den führenden *n-b* Bits der aktuellen Adresse, um sicher zu gehen, ob der Eintrag für diese Adresse auch gültig ist. Um beim Start mit leerem Cache eine gültige Zuordnung zu haben, nehmen wir noch ein weiteres Bit, das valid Bit hinzu, das anzeigt, ob das aktuelle tag auch gültig ist.

Damit können im Cache an der Stelle *r* alle Speicherzellen stehen, für deren Adresse *a* gilt:

$$a \bmod 2^b = r$$

Solange also zwei Zellen den Abstand < 2^b haben, können sie gleichzeitig im Cache aufbewahrt werden.

Direkte Zuordnung ff

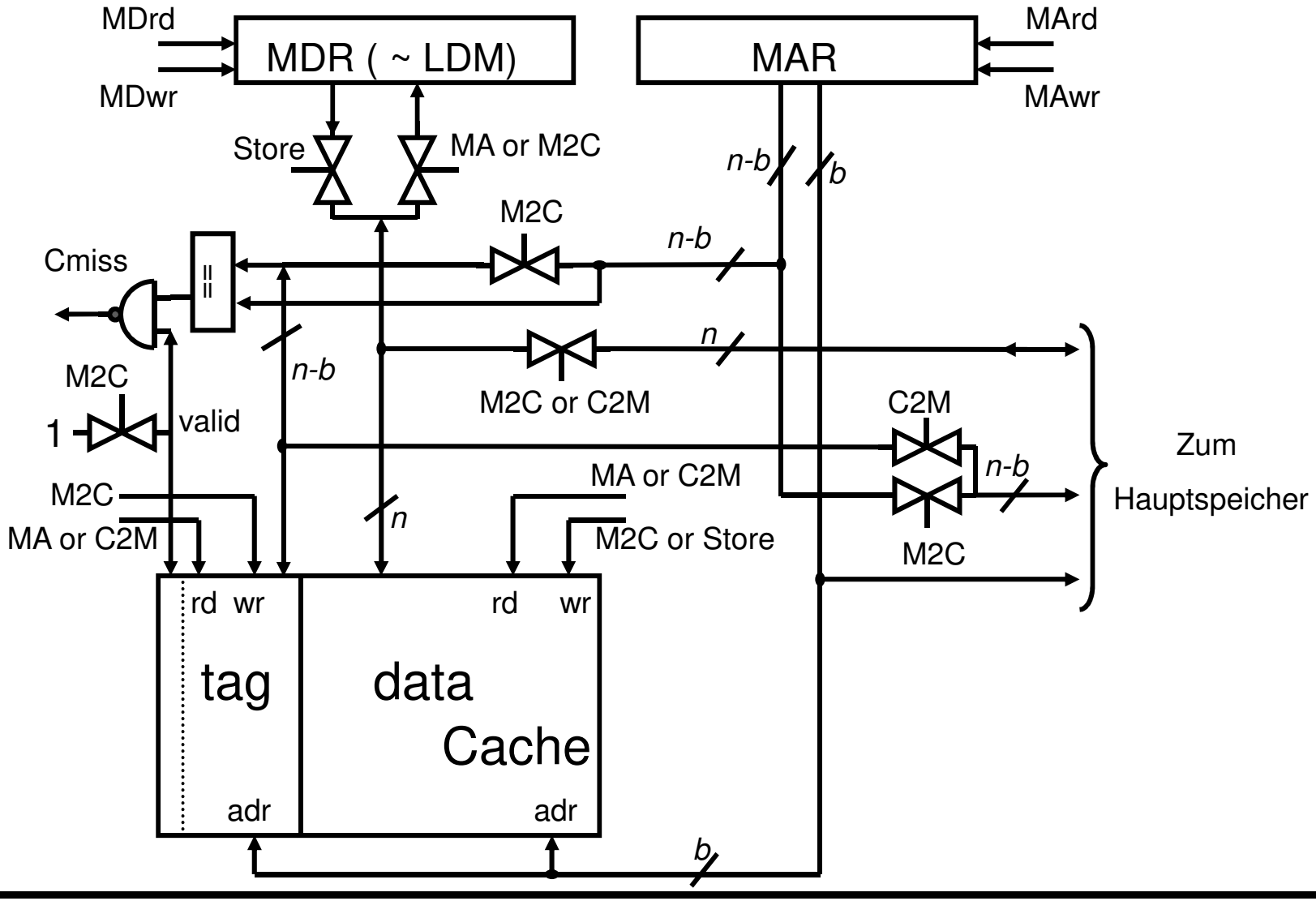
Lokalität ist also gewahrt.

Beispiel:

Betrachten wir eine Maschine mit 128 MB Hauptspeicher und 512 kB Cache. Dann benötigen wir 27-bit Adressbreite und 19-bit Adressbreite für den Cache. D.h. das tag kostet uns ein zusätzliches Byte Speicher pro Zelle. Hinzu kommt noch ein Bit für das valid bit.

Diese Lösung lässt sich alleine mit einem Vergleicher, einem Gatter und Speicherbausteinen realisieren. Hinzu kommt noch eine Erweiterung der Kontrolle des Prozessors, die cache misses in den entsprechenden Zuständen, in denen sie vorkommen können (IF,MEM) behandelt. Wir betrachten dazu zunächst die Anbindung zwischen Cache und Hauptspeicher:

Anbindung an den Hauptspeicher



Erläuterungen zur Anbindung

Wir haben nun einige Kontrollsignale dazugenommen, die Ansteuerung des Hauptspeichers selbst lassen wir weg:

- Cmiss: zeigt ein Cache miss an
- MA: zeigt Memory Access Zyklus an
- M2C: Transport zwischen Hauptspeicher und Cache
- C2M: Transport zwischen Cache und Hauptspeicher
- Store: Wegspeichern des MDR (Memory Data Register) im Cache (Store Befehl)
- MDrd,MDwr: Lese/Schreibkontrollsignale des Datenregisters
- MArd,MAwr: Kontrollsignale des Adressregisters

Wir skizzieren nun die Kontrolle:

Kontrolle der Cache/HS Anbindung

Ideen:

Wir führen im MEM Zyklus stets einen Lesezugriff auf den Cache aus.

Datum gültig: Dann können wir einen **`LOAD`** Befehl auch schon als ausgeführt betrachten, wenn wir das MDR bei einem **`LOAD`** schon im MEM-Zyklus beschreiben.

Datum ungültig: Nun müssen wir erst

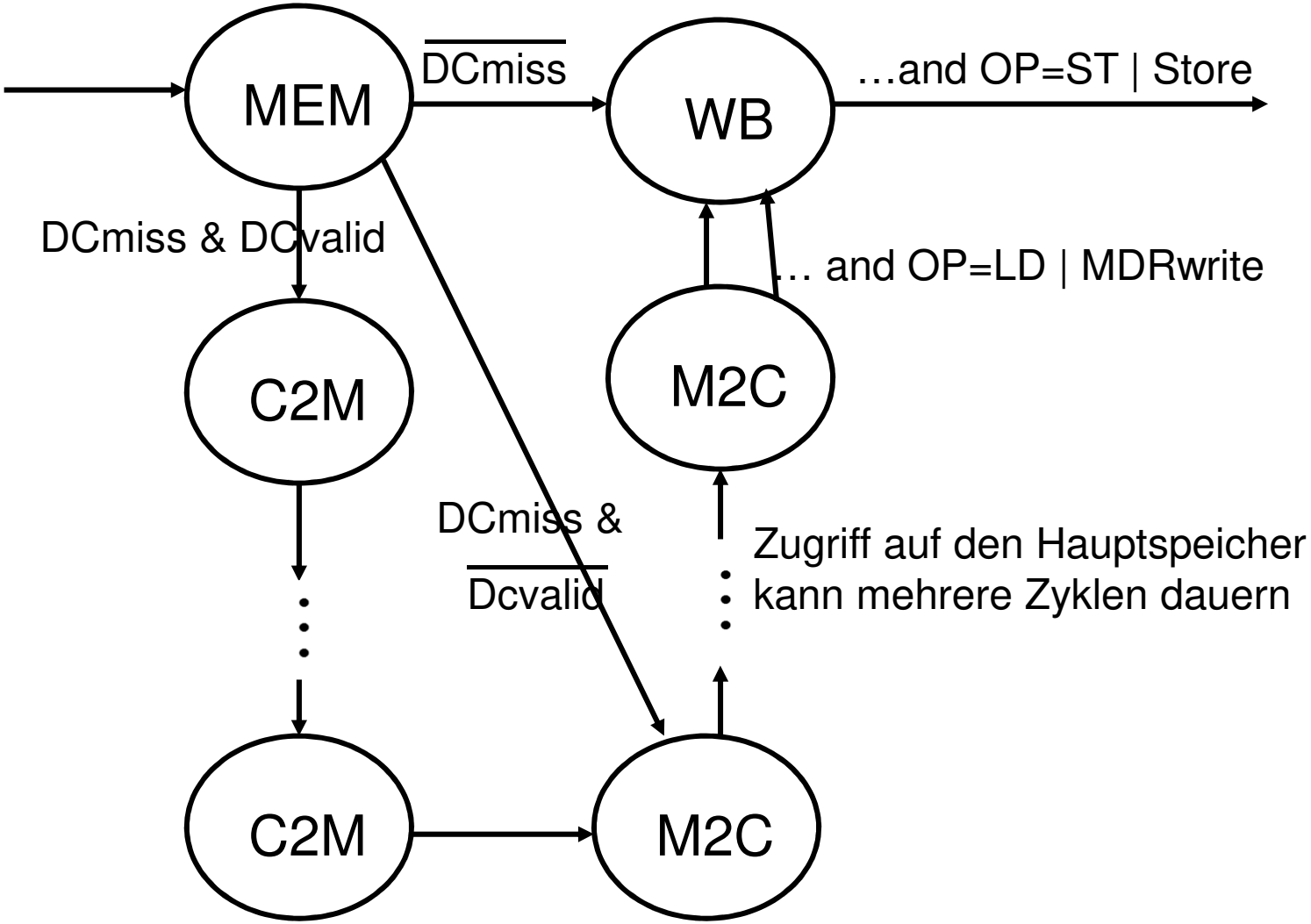
das ungültige Datum retten → **C2M**

das gültige Datum laden → **M2C**
(dabei das tag ersetzen!)

`Load:` wir können beim Laden des gültigen Datums das MDR überschreiben.

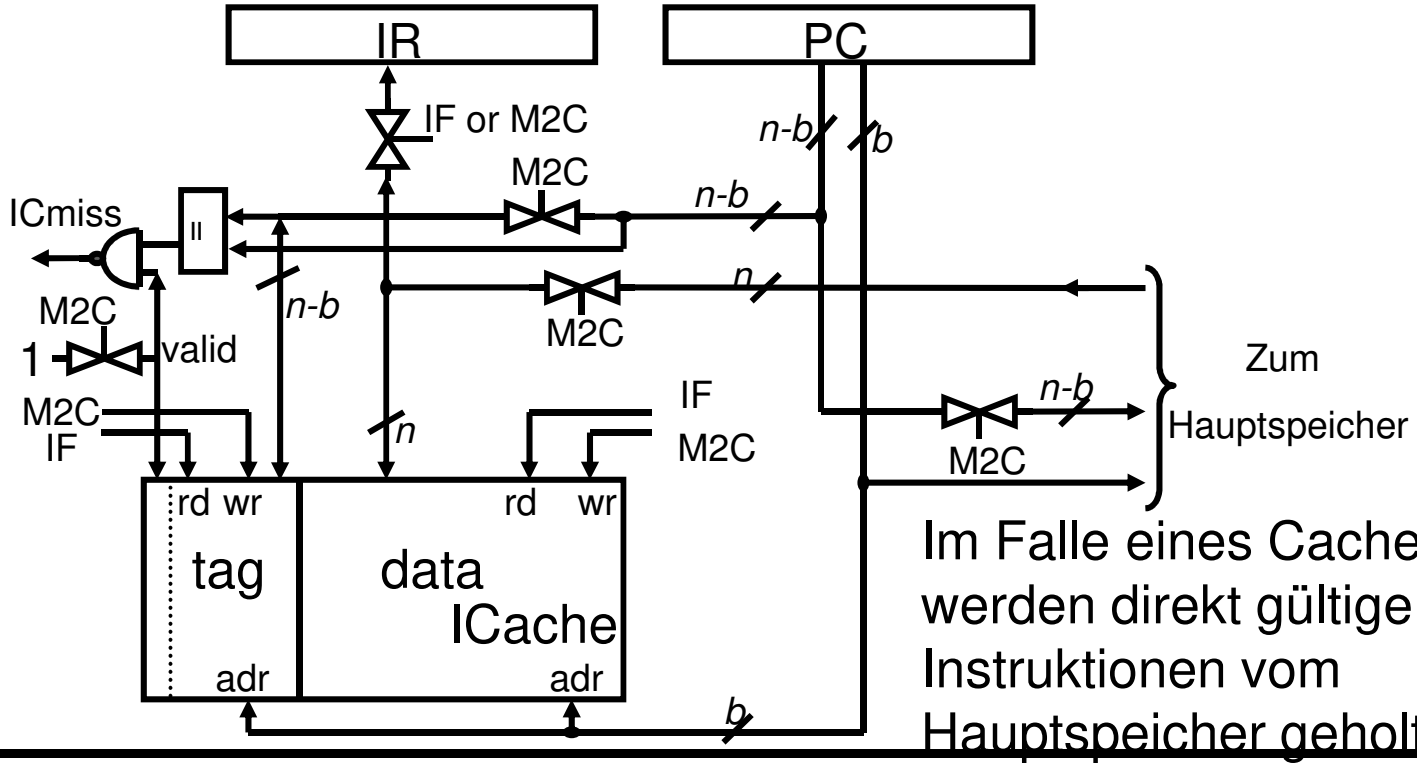
`Store:` wir verschieben das eigentliche Überschreiben in den WB-Zyklus.

Kontrolle für Datencache



Instruction Cache

Hier ist die Anbindung sehr viel einfacher, weil nur lesend zugegriffen wird. Im Wesentlichen entfallen einfach nur Teile der Ansteuerung und der Kontrolle zum Retten des Inhalts eines ungültigen Eintrags:



Im Falle eines Cache miss werden direkt gültige Instruktionen vom Hauptspeicher geholt!

Virtuelle Adressierung

Der Umgang mit Maschinenworten als Hauptspeicher-adressen ist problematisch:

- Man kann selten den ganzen Adressraum als RAM Speicher realisieren (32 Bit ~ 4,3 GB, 64 Bit ~ niemals)
- Start von Programmen in unterschiedlichen Speicherbereichen, oder unterschiedlichen Ausbaustufen von Maschinen (!?)
- Unterstützung von multi tasking, oder gar multi user Systemen

Jedes vernünftige Betriebssystem sollte den quasi-parallelen Betrieb mehrerer Programme unterstützen (Peripherie, Netzwerk,…) und für Fairness und Schutz der Programme untereinander sorgen!

Virtuelle Adressierung ff

Idee:

jedes Programm rechnet virtuell auf dem gesamten Adressraum der Maschine. Dazu müssen

Maschinenworte = **virtuelle Adressen**

in

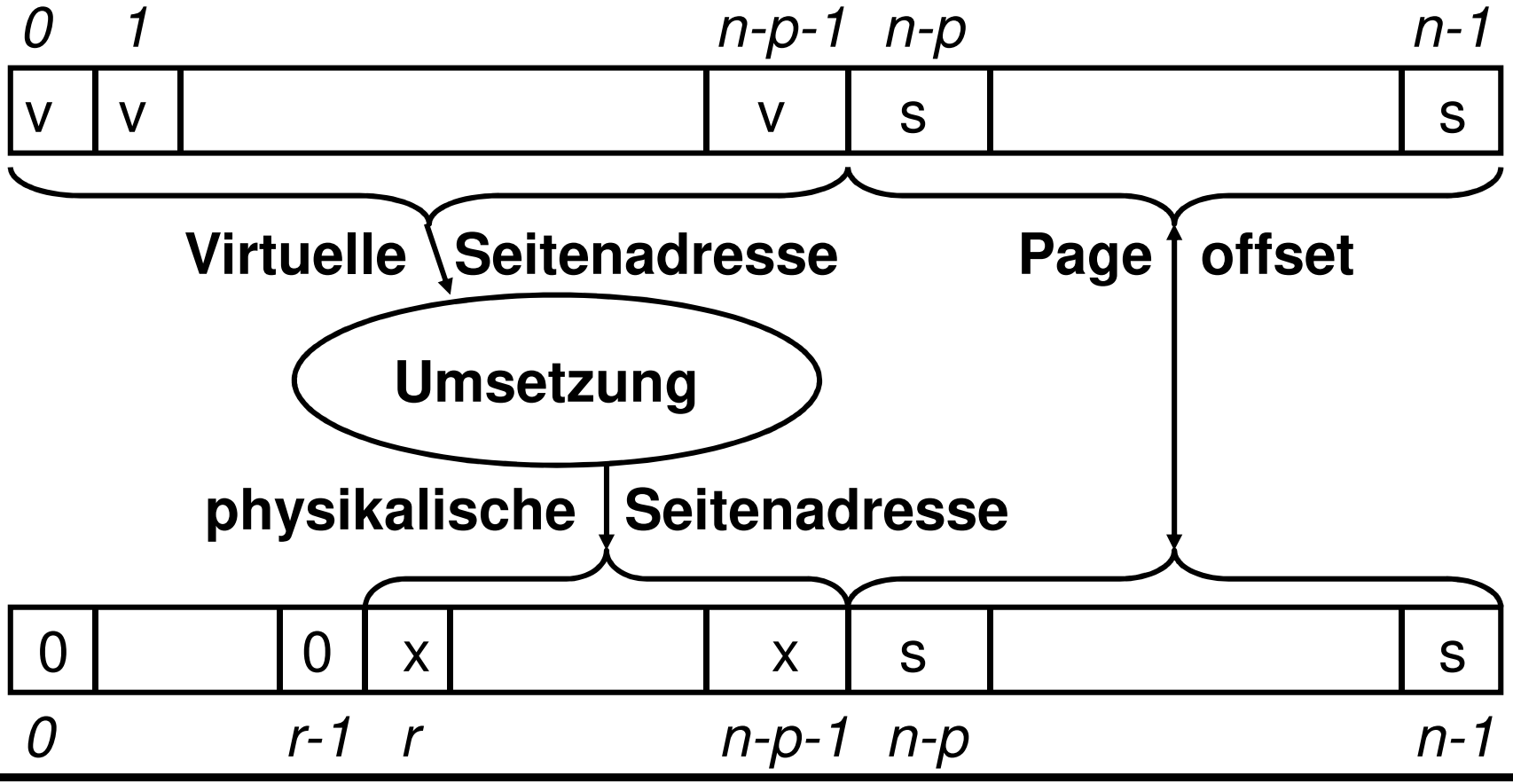
physikalische Adressen von *[0:M-1],*

wo *M* die Speichergröße ist, umgerechnet werden. Diese Umsetzung übernimmt ein privilegiertes Programm, das Betriebssystem.

Bei einem solchen Vorgehen sind alle Programme vor Übergriffen geschützt, wenn die Zuteilung und Umsetzung des physischen Speichers korrekt ist.

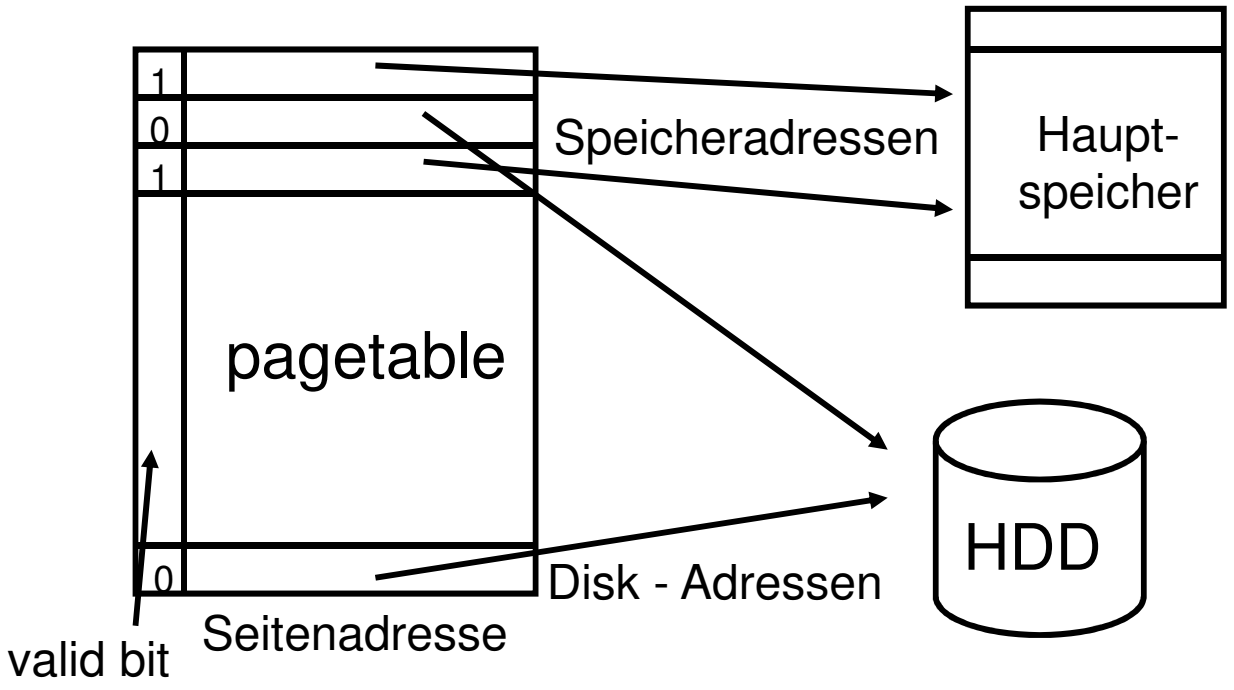
Virtuelle Adressierung ff

Zur Umsetzung gehen wir ähnlich vor wie bei Caches, d.h. wir teilen den Speicher in Seiten (pages) ein:



Virtuelle Adressierung ff

Die Umsetzung und Vergabe physischer Seiten steuert das Betriebssystem. Es muss auch Buch führen, ob eine Seite im Hauptspeicher liegt, oder ob sie zwischenzeitlich auf langsamere Massenspeicher ausgelagert wurde. Diese Umsetzung verwaltet das System in der **Pagetable**:



Virtuelle Adressierung ff

Zahlenbeispiele:

Bei einem virtuellen Adressraum vom *28* Bit (256MB) hätten wir bei

Pagegröße *2 kB* → *11 Bit* offset und *17 Bit* Pageadresse, ergibt *17 Bit* Adressraum zu *3* Byte = *384 kB* Pagetable

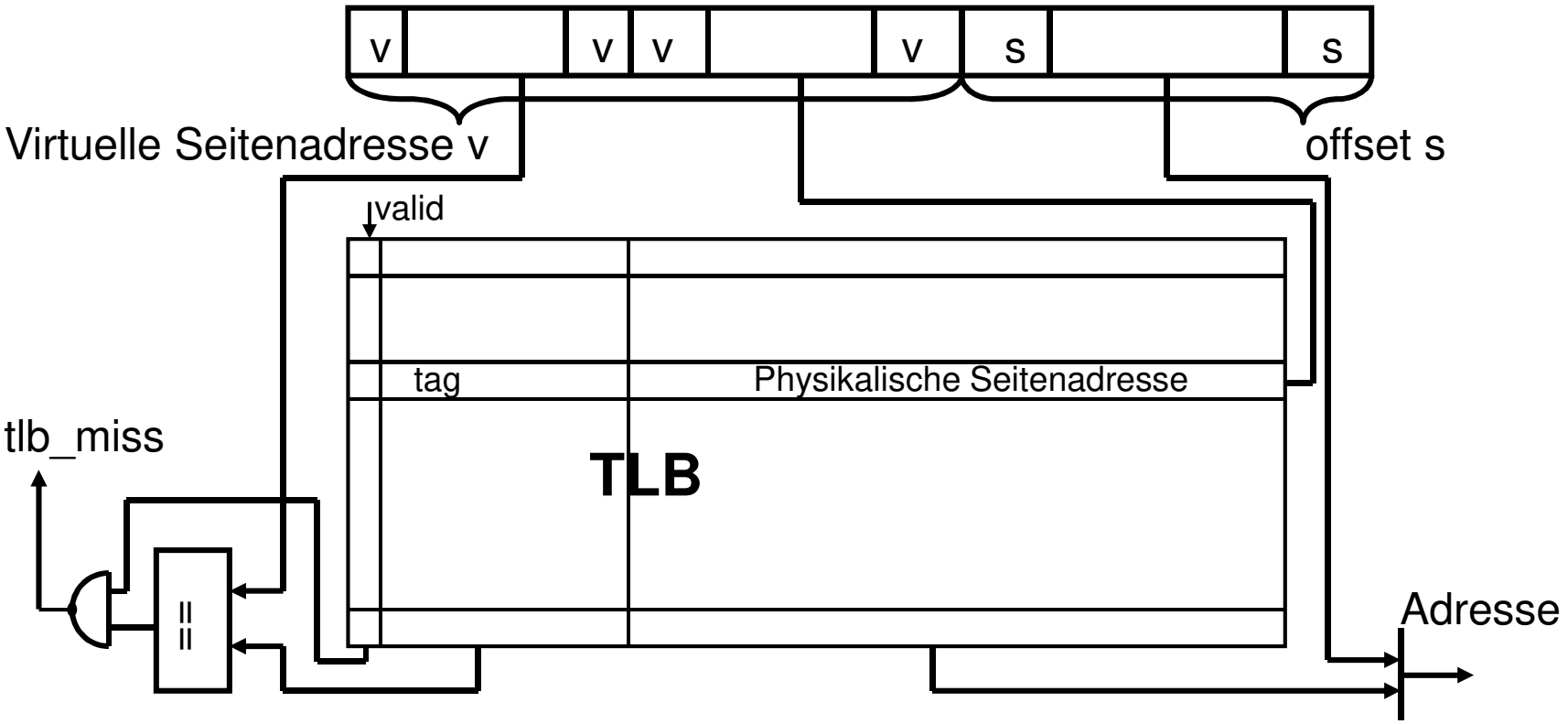
Pagegröße *16 kB* → *14 Bit* offset und *14 Bit* Pageadresse, ergibt *14 Bit* Adressraum zu *2* Byte = *32 kB* Pagetable

Problem:

Nun würden Cache misses ja ewig dauern, weil zunächst das Betriebssystem (in festem Adressbereich arbeitend) aufgerufen werden müsste, um die Umsetzung nachzuschlagen. Geht das überhaupt?

Translation Look aside Buffer -- TLB

Lösung: Wir halten auch Teile der Pagetable in einem kleinen, schnellen Speicher, dem **translation look aside buffer**, in dem die Umsetzung nachgeschlagen wird. Die Zuordnung kann wie bei Caches geschehen.



Translation Look aside Buffer ff

Diese Umsetzung über den **translation look aside buffer** wird nun bei jedem Transport zwischen Cache und Hauptspeicher nachgeschlagen.

Fehlt ein gültiger Umsetzungseintrag (TLB-miss) muss die Kontrolle zunächst in der Pagetable die geforderte Umsetzung nachschlagen. Dazu kann man z.B. ein spezielles Register einführen, das **Pagetable Register**, das die (physikalische Anfangsadresse) der Pagetable des aktuell laufenden Programmes im Hauptspeicher enthält. Dieses Register darf nur im privilegierten Modus beschrieben werden.

Typische Größen für den TLB sind *32 -1024* Einträge.

Was wir nicht geschafft haben

- CISC Architekturen, Mikroprogrammierung
- Busse
- Ein-/Ausgabegeräte
- Coprozessoren
- Pipelining
- Superscalare Architekturen
- Parallelrechner