

# Vorlesung Rechnerstrukturen

---

- **Kapitel 2: Architektur und Mikroarchitektur von Mikroprozessoren**
  - ◆ **2.9 Sprungvorhersage und spekulative Ausführung**

# Sprungvorhersage

---

- Behandlung von Sprungbefehlen und Verzweigungen ist sehr wichtig für superskalare Prozessoren
  - ◆ Bereitstellen möglichst vieler Kandidaten für die Zuweisung.
  - ◆ Programmanalyse
    - ◆ Jeder 7. Befehl ist Sprungbefehl, der den Programmfluss unterbrechen kann.
  - ◆ Sprungvorhersage (Branch Prediction)
    - ◆ Vorhersage des Verhaltens von Verzweigungen

# Sprungvorhersage

---

- Behandlung von Verzweigungen
  - ◆ Frühes Bestimmen der Verzweigungsrichtung (branch resolution)
  - ◆ Puffern des Sprungziels im BTAC nach dem erstmaligem Berechnen und bei Bedarf sofortiges Laden des PCs aus dem BTAC.
  - ◆ Sprungvorhersage (branch prediction) mit hoher Genauigkeit und der Möglichkeit, Befehle spekulativ auszuführen.
  - ◆ Zulassen mehrerer Spekulationsebenen
  - ◆ Schneller Rückrollmechanismus bei falscher Vorhersage (geringe misprediction penalty)

# Sprungvorhersage

---

- Kosten einer falschen Vorhersage (misprediction penalty)
  - ◆ Abhängig von Pipeline-Organisation
    - ◆ Lange Pipelines verursachen höhere Kosten.
    - ◆ Löschen von als ungültig erkannten Befehlen aus der Pipeline oft nicht möglich: Warten bis Retire notwendig.
    - ◆ Anzahl spekulativer Befehle im Befehlsfenster oder im Reorder Buffer:
      - ◆ Es können nur wenige Befehle pro Zyklus gelöscht werden.
  - ◆ Beispiele:
    - ◆ Alpha 21164: 4-9 Taktzyklen
    - ◆ Pentium II: 11 oder mehr Taktzyklen

# Sprungvorhersage

---

- ❑ Statische Sprungvorhersage
- ❑ Dynamische Vorhersage
  - ◆ Berücksichtigung des Programmverhaltens
  - ◆ Genauere Vorhersage möglich
  - ◆ Hoher Hardware-Aufwand!

# Dynamische Sprungvorhersage

---

- ❑ Sprungziel-Cache: Branch Target Address Cache (BTAC), Branch Target Buffer (BTB)
  - ◆ Speichert die Adresse der Verzweigung und das entsprechende Sprungziel
  - ◆ Steht in Verbindung mit IF-Phase

Adresse der Verzweigung	Sprungziel-adresse

# Dynamische Sprungvorhersage

---

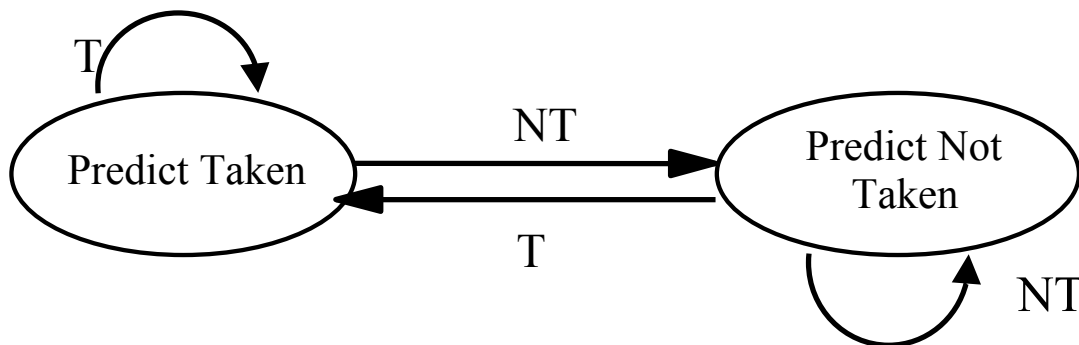
- Branch Prediction Buffer, Branch History Table (BHT)
  - ◆ Speichert die Adresse der Verzweigung und das entsprechende Sprungziel
  - ◆ Weiteres Feld für Vorhersagebit

Adresse der Verzweigung	Sprungziel-adresse	Vorher-sagebits

# Dynamische Sprungvorhersage

## □ Ein-Bit Predictor

- ◆ Branch Prediction Buffer, Branch History Table
- ◆ Vorhersagebit:
  - ◆ Wenn das Bit gesetzt ist, wird angenommen, dass der Sprung ausgeführt wird.
  - ◆ Wenn das Bit nicht gesetzt ist, wird angenommen, dass der Sprung nicht ausgeführt wird.
  - ◆ Bei einer Fehlannahme: Invertieren des Bits



# Dynamische Sprungvorhersage

---

## □ Ein-Bit Predictor

- ◆ Korrekte Vorhersage eines Sprungs am Ende einer Schleife, solange die Schleife ausgeführt wird.
- ◆ In geschachtelten Schleifen kommt es in der inneren Schleife zu falschen Vorhersagen:
  - ◆ Am Ende der inneren Schleife wird das Austreten aus der Schleife falsch vorhergesagt.
- ◆ Vermeiden des Problems durch 2-Bit Predictor

# Dynamische Sprungvorhersage

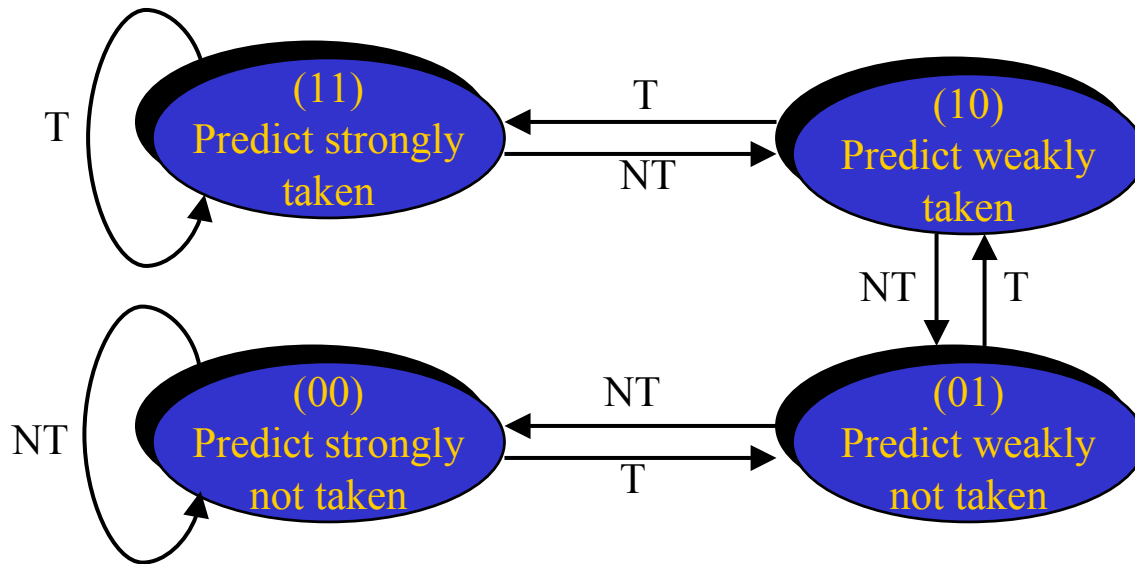
---

## □ Zwei-Bit Predictor

- ◆ Zwei Bit pro Eintrag für die Kodierung der Vorhersage → vier Zustände:
  - *Sicher genommen (strongly taken)*
  - *Vielleicht genommen (weakly taken)*
  - *Vielleicht nicht genommen (weakly not taken)*
  - *Sicher nicht genommen (strongly not taken)*
- ◆ In einem sicheren Zustand sind zwei aufeinanderfolgende Fehlannahmen notwendig, um die Vorhersageannahme umzudrehen.

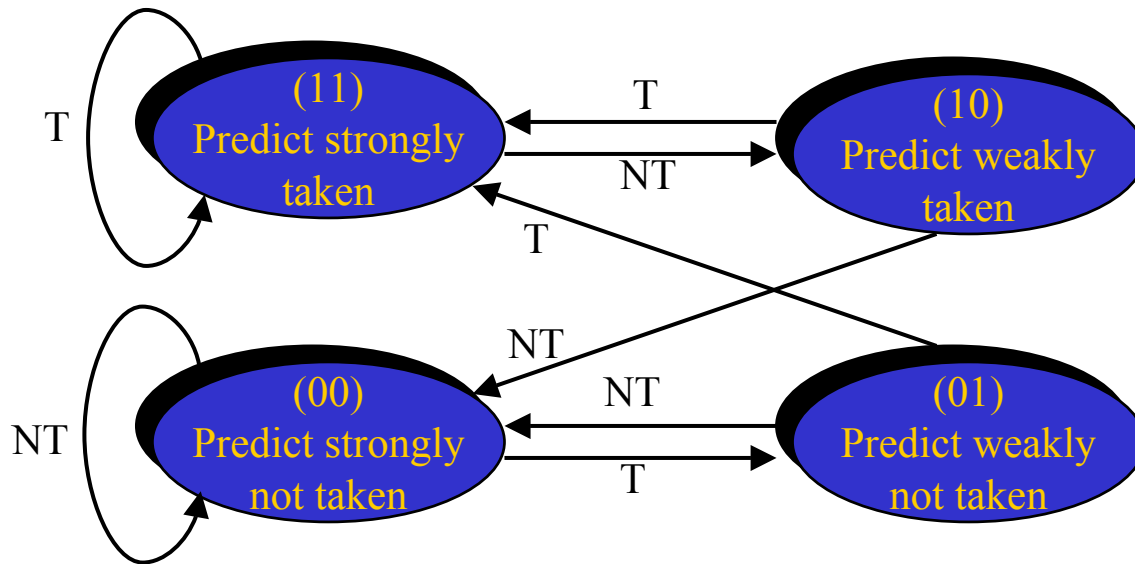
# Dynamische Sprungvorhersage

- Zwei-Bit Predictor mit Sättigungszähler (Two Bit Predictor with Saturation Scheme)



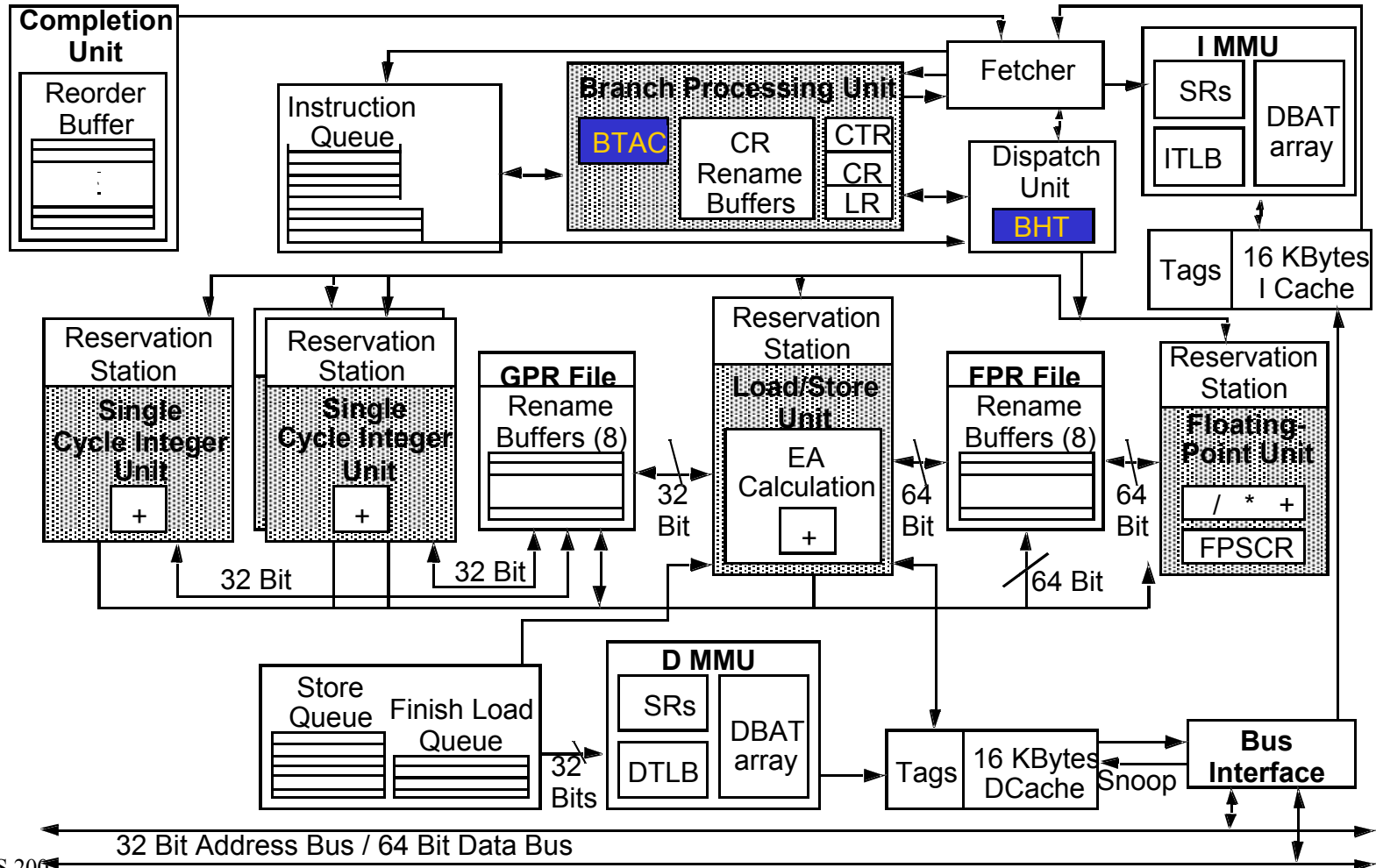
# Dynamische Sprungvorhersage

- Zwei-Bit Predictor mit Hysterese methode (Two Bit Predictor with Hysteresis Scheme)



# Dynamische Sprungvorhersage

- Beispiel eines superskalaren Prozessors: Motorola PowerPC 604



# Dynamische Sprungvorhersage

---

## □ Zwei-Bit Predictor

- ♦ Erweiterbar auf  $n$  Bit
  - ♦ Experimente haben gezeigt, dass kaum Verbesserungen erzielbar sind.
- ♦ Implementierbar im Branch Target Address Cache
- ♦ Neben BTAC Verwendung einer Branch History Table als Vorhersagetabelle
- ♦ Fehlannahmen:
  - ♦ Falsche Annahme für Verzweigung
  - ♦ Durch die Indizierung wurde die Vergangenheit eines anderen Sprungbefehls betrachtet.
- ♦ Gute Vorhersagen in technisch-wissenschaftlichen Programmen (Schleifen).
- ♦ Hohe Fehlannahmerate bei Programmen, in denen die Sprünge miteinander in Beziehung stehen.

# Dynamische Sprungvorhersage

---

- ❑ Korrelations-Prediktoren (Correlation-Based Predictors)
  - ◆ Auswerten des Verhaltens anderer Sprünge zusätzlich zu der Verzweigung, für welche die Vorhersage zu treffen ist.
  - ◆ Neben der Auswertung der Vorgeschichte des aktuellen Sprungs, auch Auswertung der Vorgeschichte abhängiger Sprünge.
  
- ❑  $(m,n)$ -Predictors:
  - ◆ Benutzt das Verhalten der letzten  $m$  Sprünge für die Auswahl aus  $2^m$  Prediktoren, wobei jeder Prediktor ein  $n$ -Bit Prediktor für einen Sprung ist.

# Dynamische Sprungvorhersage

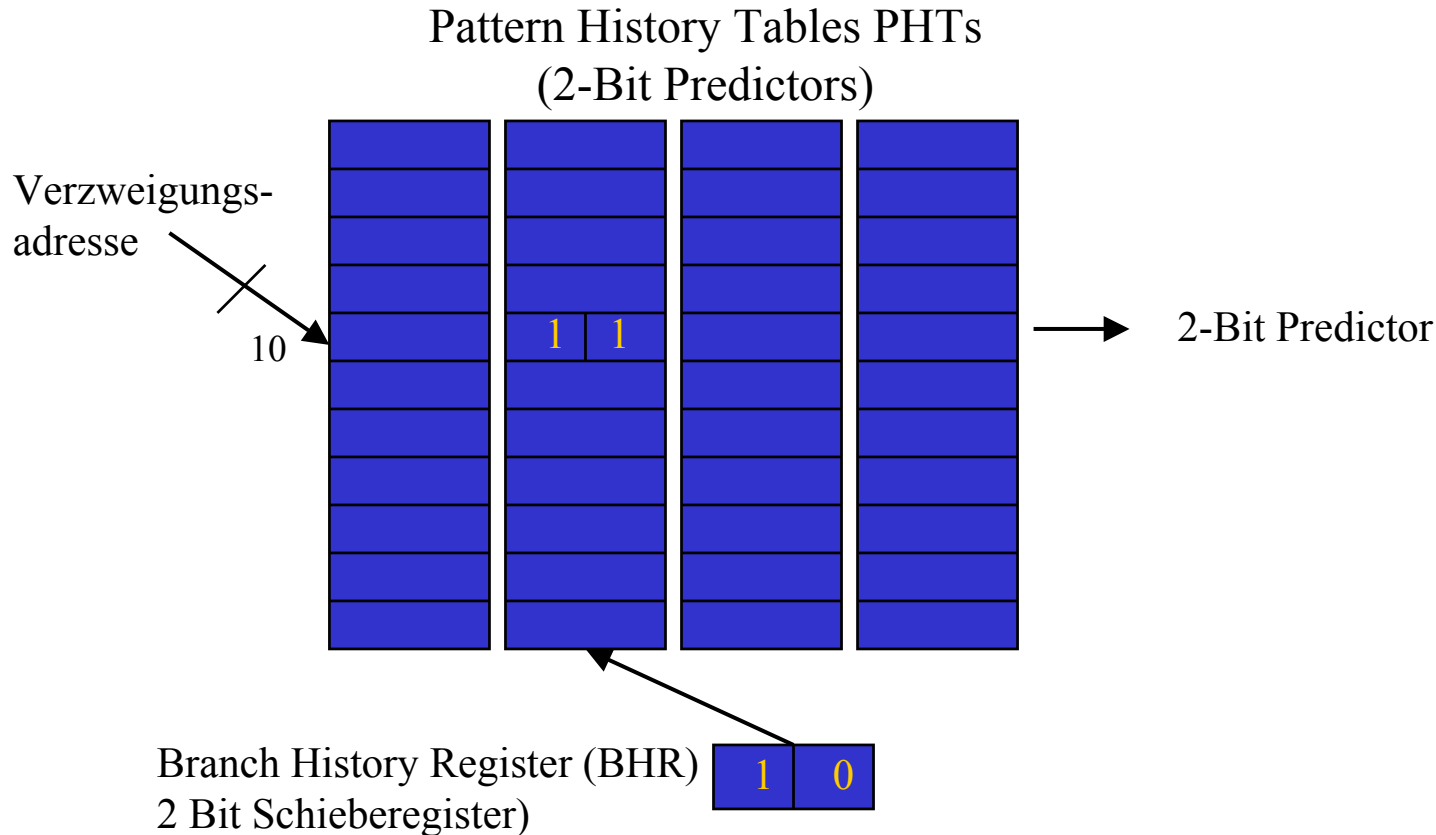
---

## □ (m,n)-Predictors:

- ◆ Benutzt das Verhalten der letzten  $m$  Sprünge für die Auswahl aus  $2^m$  Prediktoren, wobei jeder Prediktor ein  $n$ -Bit Prediktor für einen Sprung ist.
- ◆ **Branch History Register (BHR)**
  - ◆ M-Bit Schieberegister
  - ◆ Globale Vorgeschichte der letzten  $m$  Sprünge
  - ◆ Zustand der Bits zeigt an, ob die letzten Sprünge genommen worden sind oder nicht.
  - ◆ Nach jedem Sprung wird der Sprungausgang in BHR geschoben.
  - ◆ Inhalt des BHR bildet Index in sogenannte **Pattern History Table (PHT)**

# Dynamische Sprungvorhersage

## □ Beispiel: (2,2)-Predictor:



# Zusammenfassung: Sprungvorhersage

---

## □ Sprungvorhersage

- ◆ Für superskalare Prozessoren ist eine möglichst genaue Vorhersagetechnik notwendig.

## □ Statische Vorhersage

- ◆ Hardware- oder Compiler-Techniken

## □ Dynamische Vorhersage

- ◆ Berücksichtigung der Vorgeschichte
- ◆ Hohe Genauigkeit erreichbar
- ◆ Hoher Hardware-Aufwand

# Zusammenfassung: Sprungvorhersage

---

## □ Dynamische Vorhersage

- ◆ Branch Target Buffer (BTB), Branch Target Address Cache (BTAC)
  - ◆ Festhalten des Sprungziels
- ◆ Branch History Table
  - ◆ Festhalten der Vorgeschichte eines Sprungbefehls
  - ◆ 1-Bit Predictor
  - ◆ 2-Bit Predictor
- ◆ (m,n) Predictors
  - ◆ Festhalten der Vorgeschichte des Sprungbefehls
  - ◆ Korrelation mit abhängigen Sprüngen

# Bedingte Befehle

---

- ❑ Elimination von Sprüngen
- ❑ Die Ausführung eines bedingten Befehls hängt von einer Bedingung ab, die mit dem Befehl assoziiert ist.
  - ◆ Die Auswertung der Bedingung erfolgt mit der Ausführung des Befehls
  - ◆ Wenn die Bedingung erfüllt ist, dann wird die Instruktion normal ausgeführt und das Ergebnis gültig.
  - ◆ Wenn die Bedingung nicht erfüllt ist, dann verhält sich die Instruktion wie eine Leeroperation.
- ❑ Wandeln Steuerabhängigkeiten in Datenabhängigkeiten

# Bedingte Befehle

---

□ Beispiel:

if (A=0) {S=T} →

BNEZ R1, L2

MOV R2,R3

L2:

**CMOVZ R2,R2,R1**

**Bedingter Befehl:**

# Bedingte Befehle

---

□ Beispiel:  $A = \text{abs}(B)$

if  $(B < 0)$  then  $\{A = -B\}$  else  $\{A = B\}$

kann als Paar von bedingten Datentransportoperationen implementiert werden, oder

als unbedingter Transport  $(A = B)$  und als bedingter Transport  $(A = -B)$

# Bedingte Befehle

---

- Beispiele von Prozessoren mit bedingten Befehlen:
  - ◆ Alpha: Conditional Move
  - ◆ MIPS: Conditional Move
  - ◆ PowerPC: Conditional Move
  - ◆ SPARC Conditional Move
  - ◆ Intel Pentium II: Conditional Move
  - ◆ Intel Itanium: Bedingte Ausführung von Befehlen, Predication

# Vorlesung Rechnerstrukturen

---

## □ **Kapitel 2: Architektur und Mikroarchitektur von Mikroprozessoren**

- ◆ **2.9 Registerorganisation**
- ◆ **2.10 Cache-Speicher**
- ◆ **2.11 Speicherorganisation**

# Vorlesung Rechnerstrukturen

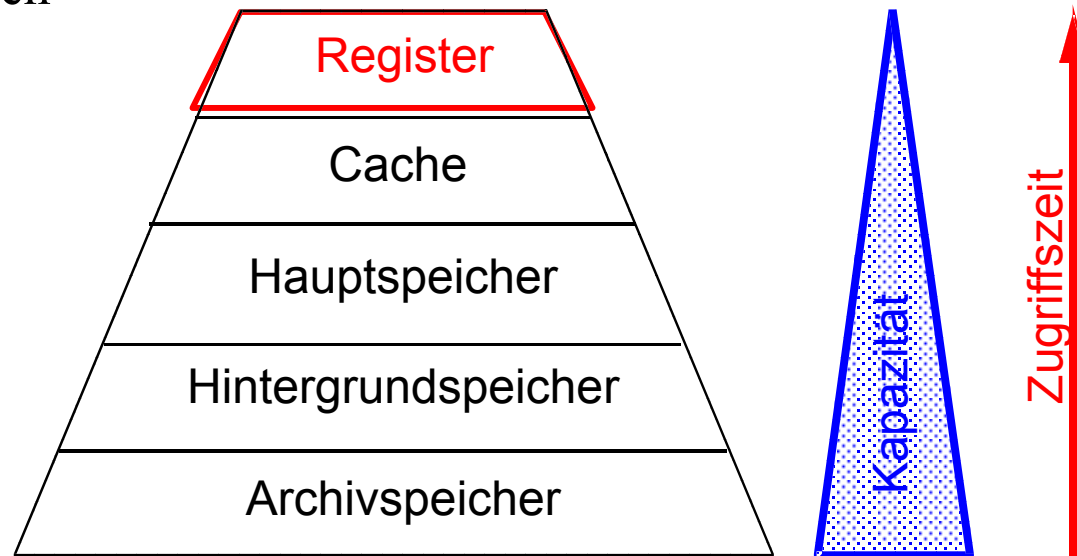
---

- **Kapitel 2: Architektur und Mikroarchitektur von Mikroprozessoren**
  - ◆ **2.9 Registerorganisation und Befehlsklassen**

# Motivation

## □ Speicherhierarchie

- ◆ Ausnützen der Lokalitätseigenschaft von Programmen
- ◆ Kompromiss zwischen Preis und Leistungsfähigkeit
- ◆ Hierarchie von Speicherkomponenten
  - ◆ Speicherkomponenten mit unterschiedlichen Geschwindigkeiten und Kapazitäten



# Register

---

## □ Register

- ◆ Schnelle Speicherplätze, auf die in der Operandenholphase des Befehlszyklus in einem Takt zugegriffen werden kann. In der Rückschreibphase werden die Ergebnisse gespeichert.
- ◆ Unterscheidung:
  - ◆ **Allgemeine (Allzweck-) Register**
    - ◆ Beispiel: 32 32-Bit Register (R0 - R31)
    - ◆ R0 ist oft festverdrahtet und mit dem Wert 0 belegt.
  - ◆ **Gleitkomma-Register**
    - ◆ Beispiel: 32 64 Bit Register (IEEE-Format)
    - ◆ Beispiel: 8 80 Bit Register (Intel FPU-Einheit)
  - ◆ **Multimedia-Register**
    - ◆ 64, oder 128 Bit Register
  - ◆ **Systemregister**
    - ◆ Befehlsregister, Statusregister, Steuerregister

# Befehlsklassen

---

## ❑ Lade-/Speicher-Architektur

- ♦ Nur die Lade- und Speicherbefehle führen Datentransport zwischen Register und Hauptspeicher durch.

## ❑ Dreiadressbefehle

- ♦ Angabe von zwei Operanden-Registern und einem Zielregister (bei arithmetisch-logischen Befehlen)

## ❑ Zweiadressbefehle

- ♦ Ein Quelloperand-Register liegt implizit fest.
- ♦ Überdeckung

## ❑ Einadressbefehle

- ♦ Akkumulator-Register dient implizit als Quell- und Zielregister.

## ❑ Nulladressbefehle

- ♦ Stack-Architekturen
- ♦ Verwalten der Operanden auf Stack.

# Registerfensterprinzip

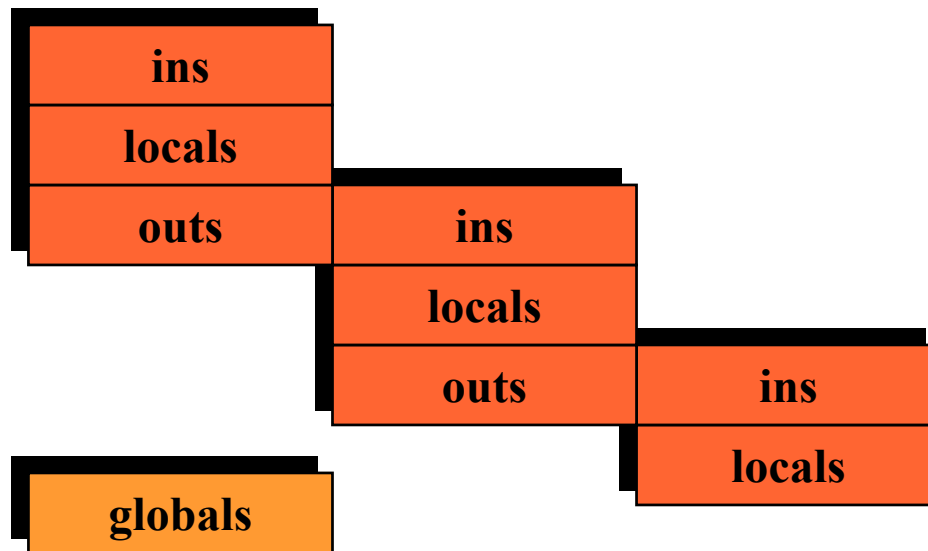
---

- ❑ Gründe für die Verwaltung von Registerfenstern
  - ◆ Forderung nach der Reduzierung der Speicherzugriffe
  - ◆ Beobachtung des Speicheraufwandes bei Prozeduraufrufen durch Sicherung des Registersatzes
  - ◆ Parameterüberwachung möglichst über Register
  - ◆ Viele Register sind technisch möglich.
  
- ❑ Konzept der überlappenden Registerfenster
  - ◆ Entwickelt für die Berkeley-RISC-Architektur
  - ◆ Findet sich in SPARC-Architekturen

# Registerfensterprinzip

---

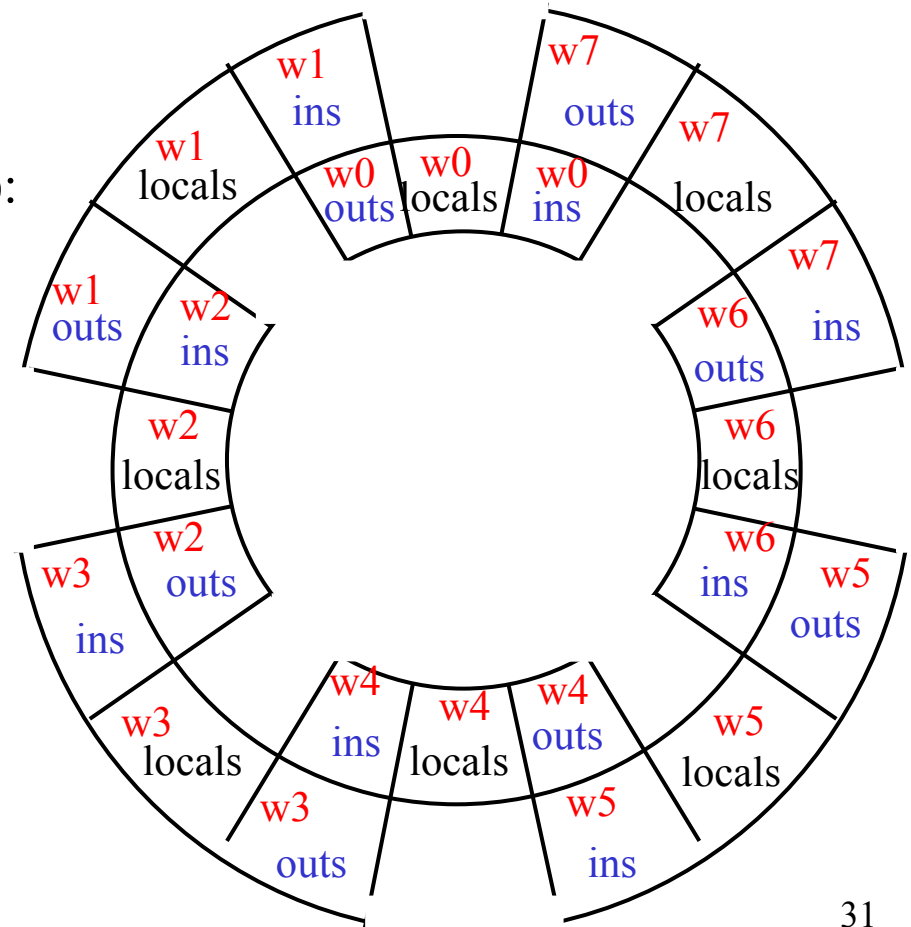
- Überlappende Registerfenster bei der SPARC-Architektur
  - ◆ 8 ins-Register, 8 locals-Register, 8 outs-Register



# Registerfensterprinzip

## □ Umlaufspeicherorganisation am Beispiel des SPARC-Prozessors mit 8 Registerfenstern

- ◆ Registerfenster:  $w0-w7$
- ◆ Current Window Pointer (CWP): zeigt das aktuelle Fenster an.
- ◆ Registerüber- und Unterlauf.
- ◆ SAVE-Befehl dekrementiert CWP
- ◆ RESTORE-Befehl inkrementiert CWP



# Registerfensterprinzip

---

## □ Eigenschaften von Umlaufspeichern

- ◆ Ein Umlaufspeicher mit  $n$  Registerfenstern kann  $n-1$  verschachtelte Prozeduraufrufe behandeln.
- ◆ Eine relativ kleine Anzahl von Registerfenstern reicht für eine effiziente Prozedurbehandlung aus.
  - ◆ Berkeley-RISC: 8 Fenster mit 16 Registern
  - ◆ SPARC-Prozessoren: 7 oder 8 Fenster, plus 8 globale Register
  - ◆ Am29000: 128 lokale Register, 64 globale Register, variable Registerfenster
- ◆ Problem: hoher Aufwand bei Prozesswechsel!

# Vorlesung Rechnerstrukturen

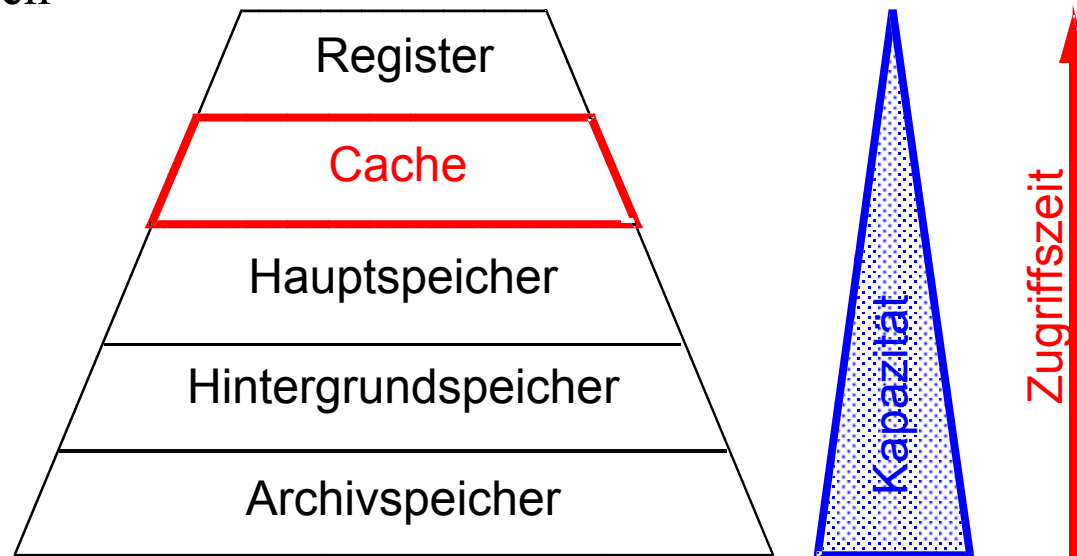
---

- **Kapitel 2: Architektur und Mikroarchitektur von Mikroprozessoren**
  - ◆ **2.10 Cache-Speicher**

# Motivation

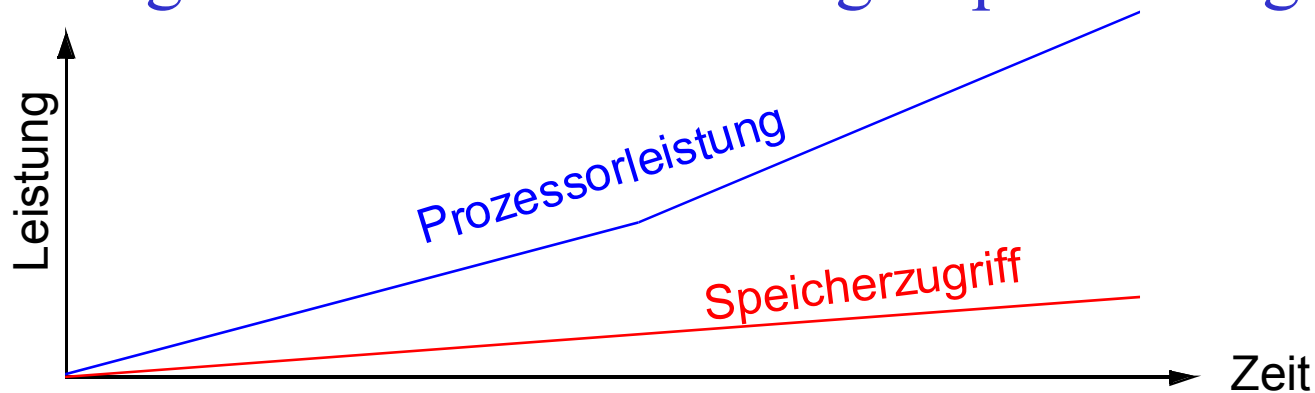
## □ Speicherhierarchie

- ◆ Ausnützen der Lokalitätseigenschaft von Programmen
- ◆ Kompromiss zwischen Preis und Leistungsfähigkeit
- ◆ Hierarchie von Speicherkomponenten
  - ◆ Speicherkomponenten mit unterschiedlichen Geschwindigkeiten und Kapazitäten



# Motivation

## □ Vergleich Prozessorleistung / Speicherzugriff



### ◆ Leistung von Prozessoren:

- ◆ hohe Taktfrequenz
- ◆ Beispiel: AMD Athlon 1,2 GHz, Pentium 4: 1,5 GHz
- ◆ Feinkörniger Parallelismus (Instruction Level Parallelism, ILP)

### ◆ Speicherzugriff:

- ◆ Zugriffszeit bei SRAM:  $\sim 4\text{ns}$  (Blockzugriff)

# Definitionen und Eigenschaften

---

## □ Cache-Speicher (Definition):

- ◆ Pufferspeicher mit schnellem Zugriff
- ◆ wichtigste Anwendung:
  - ◆ Pufferspeicher zwischen Hauptspeicher und Prozessor
- ◆ Aufgabe:
  - ◆ stellen die während einer Programmausführung jeweils aktuellen Hauptspeicherinhalte für Prozessorzugriffe als Kopien möglichst schnell zur Verfügung.

# Definitionen und Eigenschaften

---

## □ Cache-Speicher-Verwaltung

- ◆ Sorgt dafür, dass der Cache-Speicher in der Regel das Datum enthält, auf der Prozessor als nächstes zugreift.

## □ Cache-Controller (Hardware-Steuerung)

- ◆ Kopiert automatisch die Daten in den Cache, auf die der Prozessor zugreift.

# Definitionen und Eigenschaften

---

## □ Eigenschaften:

- ♦ geringere Kapazität im Vergleich zum Hauptspeicher
- ♦ besondere Strategien für das
  - ♦ Laden
  - ♦ das Aktualisieren und
  - ♦ das Adressieren des Inhalts
- ♦ Speicherung von Befehlen und Daten
  - ♦ gemeinsam in einem Cache (Befehls-/Daten-Cache)
  - ♦ getrennt jeweils in einem Befehls- und einem Daten-Cache: Harvard-Architektur

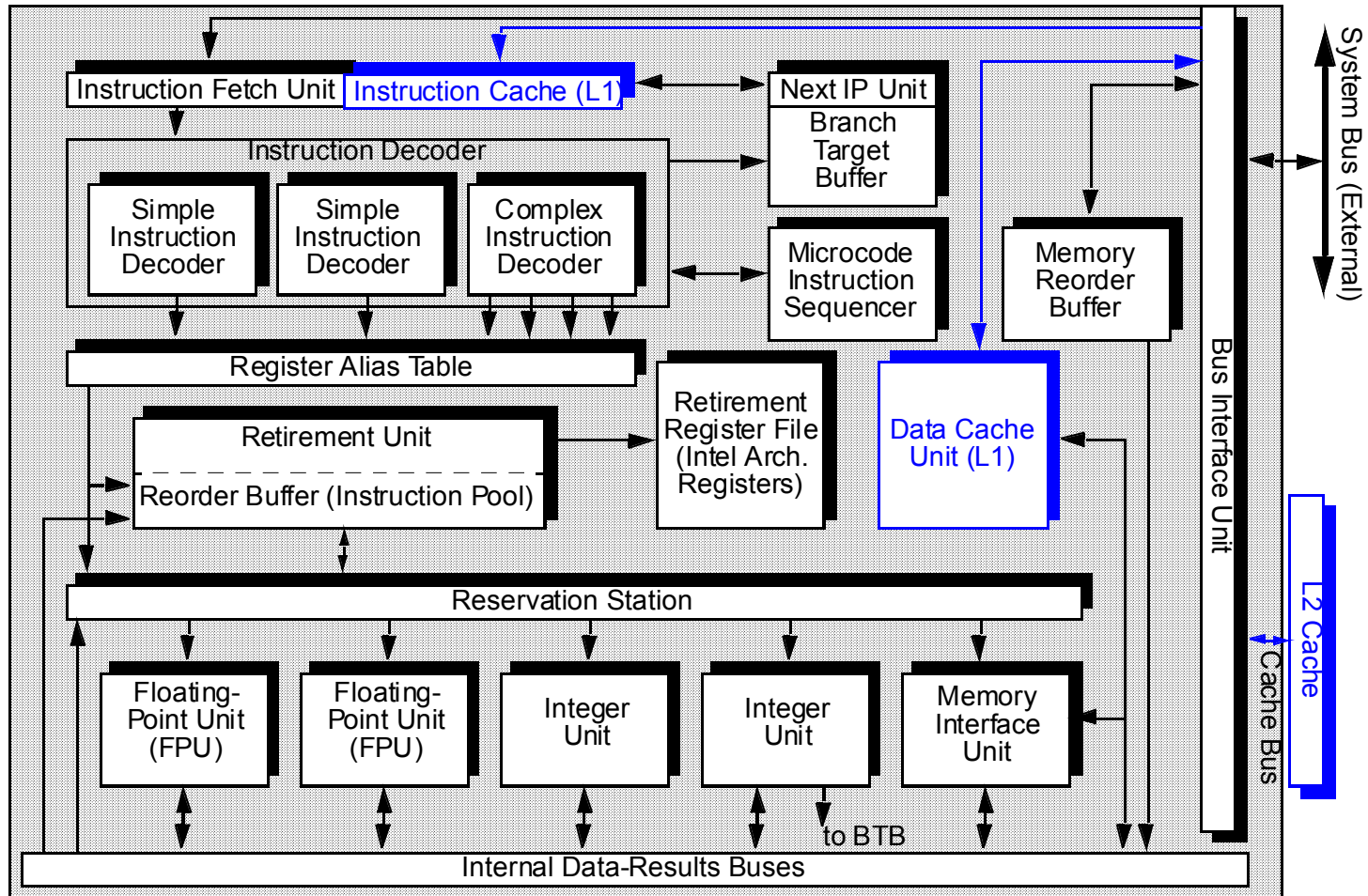
# Definitionen und Eigenschaften

---

## □ Eigenschaften:

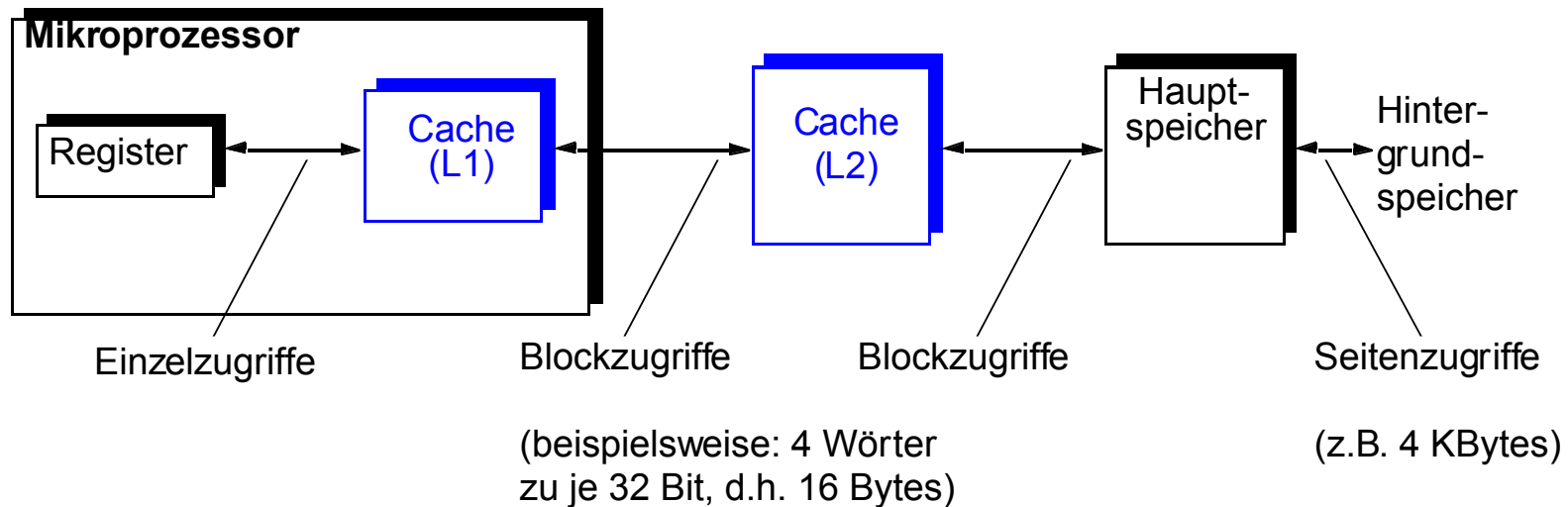
- ◆ Platzierung
  - ◆ **On-Chip-Cache**: integriert auf Prozessorbaustein
  - ◆ sehr kurze Zugriffszeiten (wie sie prozessorinterne Register haben)
  - ◆ begrenzte Kapazität (aus technologischen Gründen)
  - ◆ **Off-Chip-Cache**: prozessorextern
    - ◆ Im Allgemeinen in SRAM-Technik realisiert.

# Fallstudie: Intel Pentium II



# Definitionen und Eigenschaften

## □ Speicherhierarchie mit Übertragungsmodi



## □ Cache-Hierarchie:

- ◆ Cache-Speicher auf der ersten Stufe (Level 1, L1),
- ◆ der zweiten (Level 2, L2) Stufe,
- ◆ dritten Stufe (Level 3, L3) Stufe
- ◆ Inklusionseigenschaft:  $L1 < L2 < L3$

# Definitionen und Eigenschaften

---

## □ Blockrahmen (Block-Frame, Cache-Line)

- ◆ Reihe von Speicherplätzen im Cache-Speicher
- ◆ Adresstikett (Adress-Tag):
  - ◆ Verbunden mit Blockrahmen
  - ◆ Enthält den gemeinsamen Adressteil der in einem Block gespeicherten Datenkopien.
- ◆ Statusbits
- ◆ Blocklänge
  - ◆ Anzahl der Speicherplätze in einem Blockrahmen

# Laden des Cache-Speichers

---

## □ Initialisierung

### ◆ Cache-Speicher abgeschaltet

- ◆ verhindert, dass zufällig im Cache-Speicher vorhandene Inhalte sich negativ auswirken
- ◆ An- und Abschalten des Cache-Speichers über Steuerbits im Steuerregister
- ◆ Löschen der (immer noch) falschen Inhalte durch speziellen Befehl (**cache-clear**)
- ◆ **Zurücksetzen des Gültigkeitsbits (Valid-Bits)** in den Cache-Zeilen
- ◆ Cache-Speicher gilt als „leer“.

# Laden des Cache-Speichers

---

## □ Arbeitsweise

- ◆ Cache-Steuerung prüft bei Speicherzugriffen des Mikroprozessors, ob
  - ◆ der zur Speicheradresse gehörende Hauptspeicherinhalt als Kopie im Cache steht (Bedingung 1) und
  - ◆ dieser Cache-Eintrag durch das Valid-Bit als gültig gekennzeichnet ist (Bedingung 2).
  - ◆ Prüfung führt zu Cache-Treffer oder zu Fehlzugriff.

# Laden des Cache-Speichers

---

## □ Cache-Fehlzugriff (cache-miss):

- ◆ eine oder beide Bedingungen sind nicht erfüllt!
- ◆ **Aktionen bei Lesezugriffen (read-miss):**
  - ◆ Lesen des Datums aus dem Hauptspeicher und Laden des Cache-Speichers
  - ◆ Kennzeichnen der Cache-Eintrages als gültig (Setzen des Valid-Bits)
  - ◆ Speichern der Adressinformation im Adressteil des Cache-Speichers

# Laden des Cache-Speichers

---

## □ Cache-Fehlzugriff (cache-miss):

### ◆ Aktionen bei Schreibzugriffen (write-miss):

- ◆ Aktualisierungsstrategie bestimmt, ob
- ◆ der entsprechende Block in Cache geladen und dann mit dem zu schreibenden Datum aktualisiert wird, oder ob
- ◆ nur der Hauptspeicher aktualisiert wird und der Cache unverändert bleibt.

# Laden des Cache-Speichers

---

## □ Cache-Treffer (Cache hit, read-hit, write-hit)

- ◆ Bedingungen 1 und 2 sind erfüllt!
- ◆ Zugriff erfolgt auf Cache-Speicher;

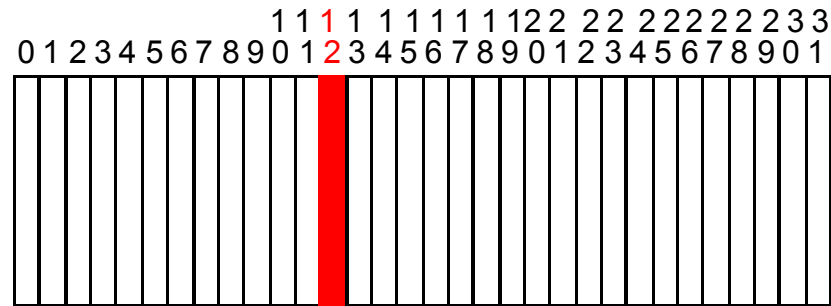
# Cache-Strukturen

## □ Wohin wird ein Block im Cache geladen?

### Hauptspeicher

Block  $B_j$   $j = 0, 1, \dots, (n-1)$

Kapazität:  $n * b = 2^{s+w}$  Wörter



### Cache

Blockrahmen  $Z_i$  (Cache-Zeile)

$i = 0, 1, \dots, (m-1)$

Kapazität:  $m * b = 2^{r+w}$  Wörter

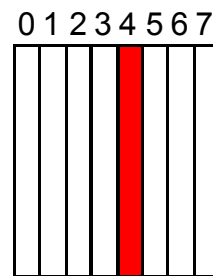


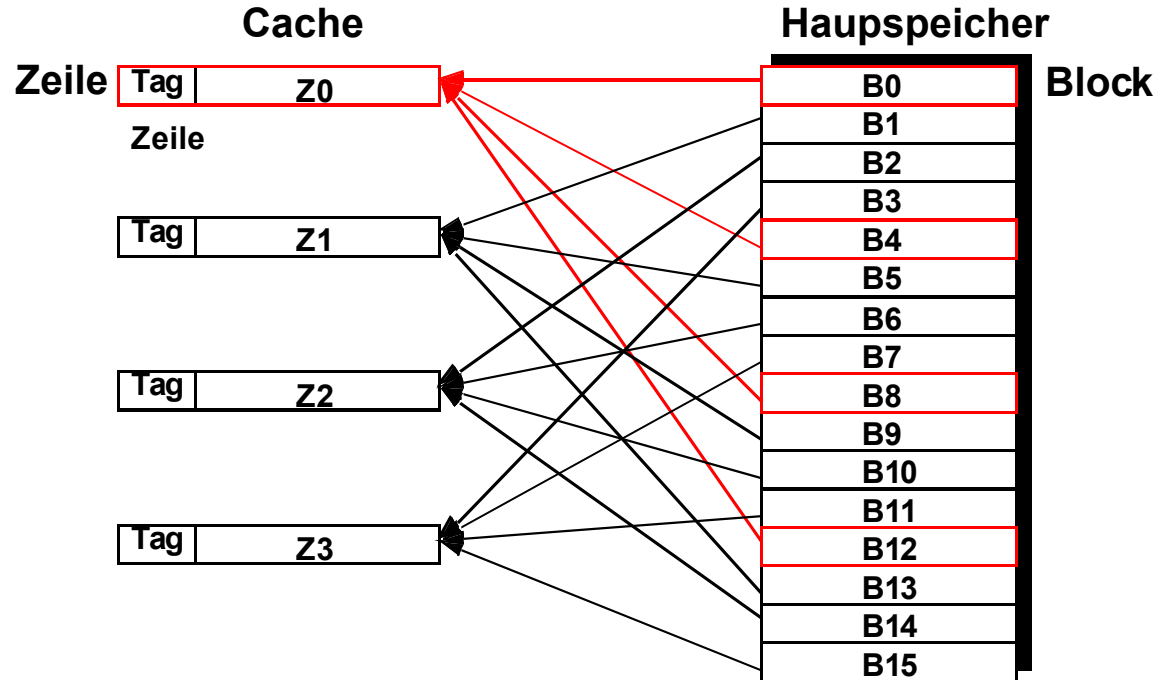
Abbildung von  $\{B_j\}$  nach  $\{Z_i\}$

$n \gg m, n = 2^s, m = 2^r$   
 jeder Block enthält  $b$  Wörter  
 mit  $b = 2^w$

# Cache-Strukturen

## □ Direkt-abgebildete Cache-Speicherverwaltung (Direct-Mapped Cache)

- ◆ direkte Abbildung von  $n/m = 2^{s-r}$  Speicherblöcken in eine Cache-Zeile:  $B_j \rightarrow Z_i$ , mit  $i = j \bmod m$





# Cache-Strukturen

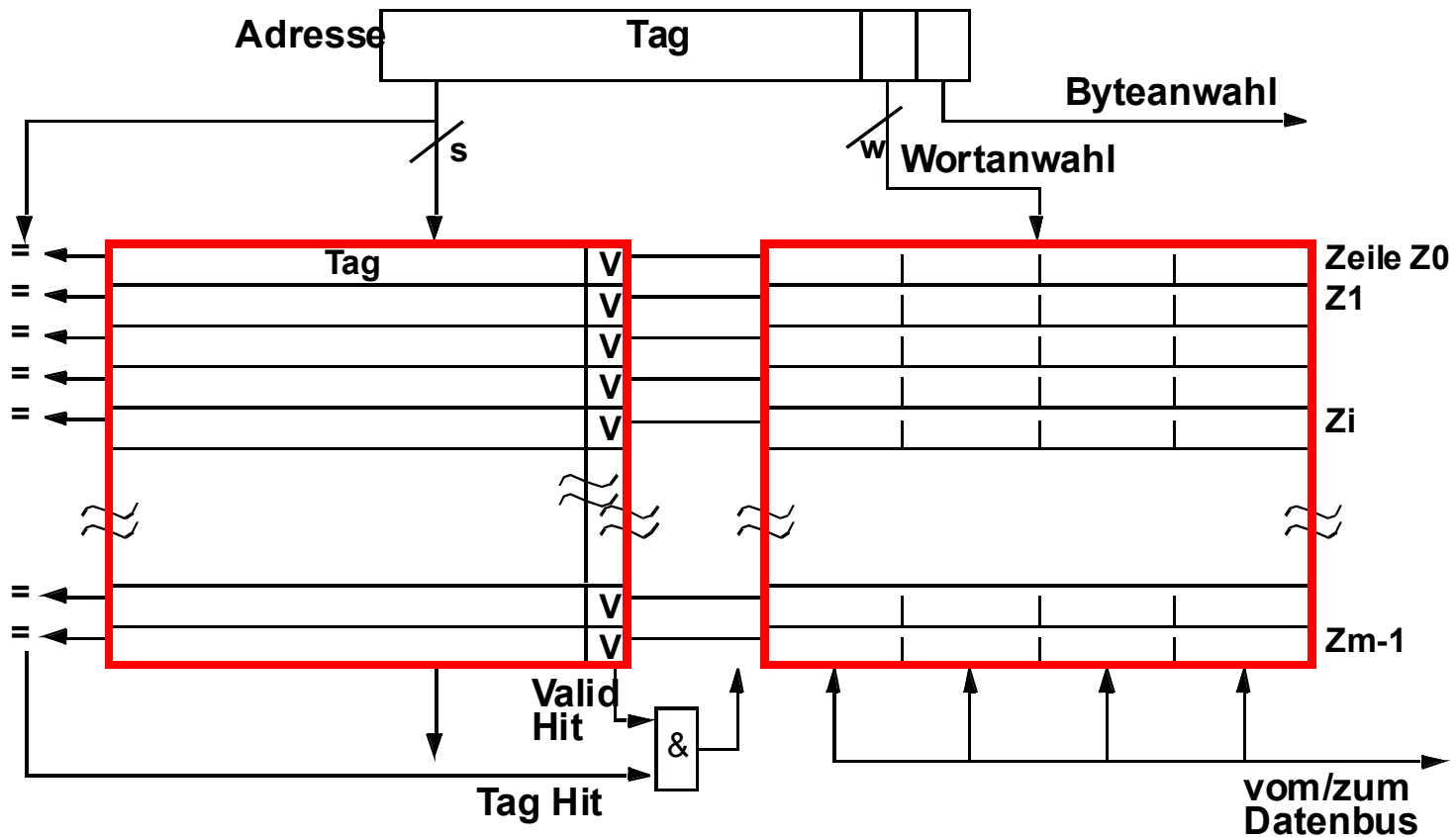
---

## □ Vollassoziativer Cache

- ◆ jeder Block des Hauptspeichers kann auf jede Zeile des Cache-Speichers abgebildet werden (Flexibilität)
- ◆ Ersetzungsstrategie gibt vor, welche Zeile beim Laden ersetzt werden soll (z.B. Least-Recently-Used)
- ◆ hoher Hardware-Aufwand (Anzahl Vergleicher = Anzahl Zeilen)

# Cache-Strukturen

## □ Vollassoziativer Cache



# Cache-Strukturen

---

## □ Mengenassoziativer oder satzassoziativer Cache

- ◆ Kompromiss zwischen Direct-Mapped- und vollassoziativem Cache;
- ◆  $k$ -fach mengenassoziativer Cache:
  - ◆ Zusammenfassen von  $k$  Zeilen zu einer Menge (set)
  - ◆ Aufteilen der  $m$  Cache-Zeilen in  $v = m/k$  Mengen zu je  $k$  Zeilen;
  - ◆ jede Menge wird über eine  $d$  Bit breite Nummer identifiziert:  $2^d = v$ ;
  - ◆ ein Block  $B_j$  kann in eine der verfügbaren Zeilen  $Z_f$  in einer Menge  $S_i$  abgebildet werden:
  - ◆  $B_j \rightarrow Z_f \in S_i$ , mit  $j(\bmod v) = i$ .

# Cache-Strukturen

## □ 2-fach mengenassoziativer Cache

