64-189
Projekt: Entwurf eines Mikrorechners

http://tams.informatik.uni-hamburg.de/lectures/2013ws/projekt/mikrorechner

– Rechnerarchitekturen: grundlegende Konzepte –

Andreas Mäder

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

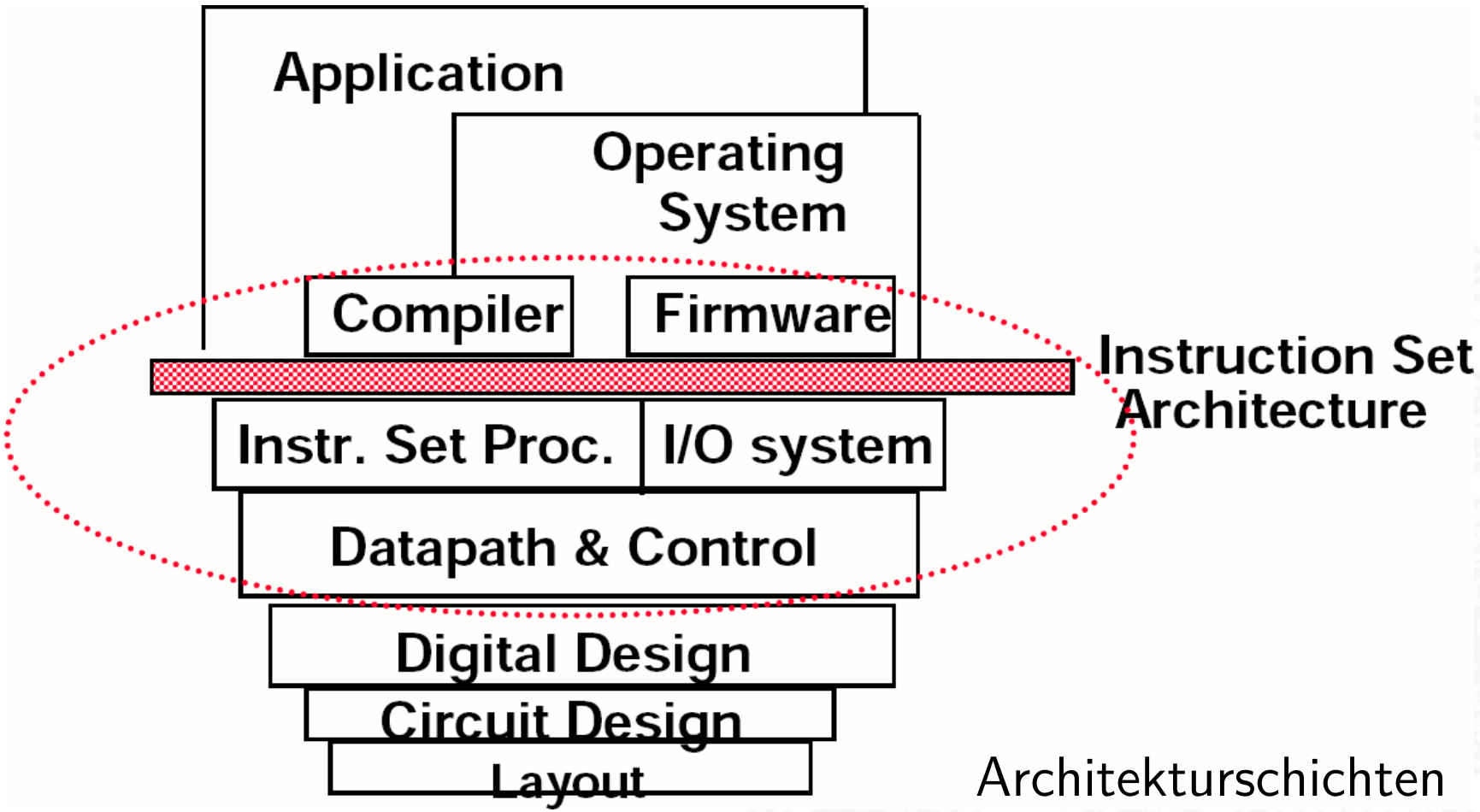
17. Oktober 2013

Die folgenden Folien sind ein Auszug aus den Unterlagen der Vorlesung 64-613 Rechnerarchitekturen und Mikrosystemtechnik vom Wintersemester 2011/2012.

Das komplette Material findet sich auf den Web-Seiten unter
http://tams.informatik.uni-hamburg.de/lectures/2011ws/vorlesung/ram

Gliederung

Schnittstellen zu anderen Inhalten des Studiums



Architekturschichten

Schnittstellen zu anderen Inhalten des Studiums (cont.)

Sehr starke Wechselwirkungen der Rechnerarchitektur mit anderen Bereichen der Informatik

- Rechnerorganisation
- Betriebssysteme
- Compilerbau
- Warteschlangentheorie

Der Bereich der Rechnerarchitektur liefert genug Stoff für mehrere eigene Vorlesungen, deshalb kann hier nur einführende Übersicht folgen

Schnittstellen zu anderen Inhalten des Studiums (cont.)

Literatur – besonders diese

- D. Patterson, J. Hennessy: *Rechnerorganisation und -entwurf; Die Hardware/Software-Schnittstelle* [PH11]
- J. Hennessy, D. Patterson: *Computer architecture; A quantitative approach* [HP11]
- A. Tanenbaum: *Computerarchitektur; Strukturen, Konzepte, Grundlagen* [Tan06]
- `http://de.wikipedia.org` und `http://en.wikipedia.org`
- C. Märtin: *Einführung in die Rechnerarchitektur: Prozessoren und Systeme* [Mär03]
- W. Oberschelp, G. Vossen: *Rechneraufbau und Rechnerstrukturen* [OV06]

Was ist Rechnerarchitektur?

Definitionen

1. *The term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behaviour, as distinct from the organization and data flow and control, the logical and the physical implementation.* [Amdahl, Blaauw, Brooks]

2. *The study of computer architecture is the study of the organization and interconnection of components of computer systems. Computer architects construct computers from basic building blocks such as memories, arithmetic units and buses.*

Was ist Rechnerarchitektur? (cont.)

From these building blocks the computer architect can construct anyone of a number of different types of computers, ranging from the smallest hand-held pocket-calculator to the largest ultra-fast super computer. The functional behaviour of the components of one computer are similar to that of any other computer, whether it be ultra-small or ultra-fast.

By this we mean that a memory performs the storage function, an adder does addition, and an input/output interface passes data from a processor to the outside world, regardless of the nature of the computer in which they are embedded. The major differences between computers lie in the way of the modules are connected together, and the way the computer system is controlled by the programs. In short, computer architecture is the discipline devoted to the design of highly specific and individual computers from a collection of common building blocks. [Stone]

Was ist Rechnerarchitektur? (cont.)

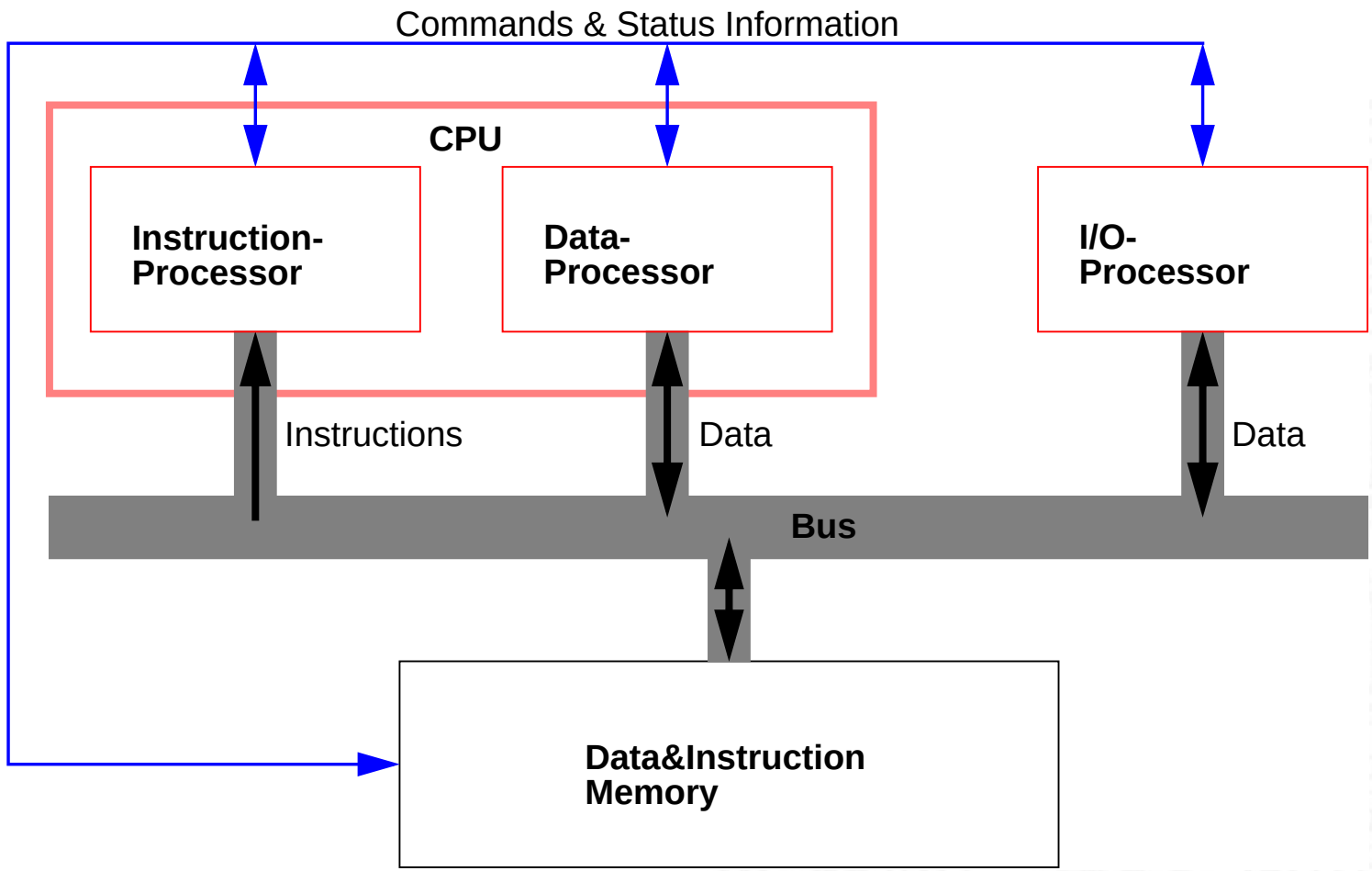
Zwei Aspekte der Rechnerarchitektur

1. Operationsprinzip: das funktionelle Verhalten der Architektur
   - = Programmierschnittstelle
   - = ISA – **I**nstruction **S**et **A**rchitecture
2. Hardwarestruktur: beschrieben durch Art und Anzahl der Hardware-Betriebsmittel sowie die sie verbindenden Kommunikationseinrichtungen
   - = Mikroarchitektur, beispielsweise „von-Neumann“ Architektur

von-Neumann Architektur

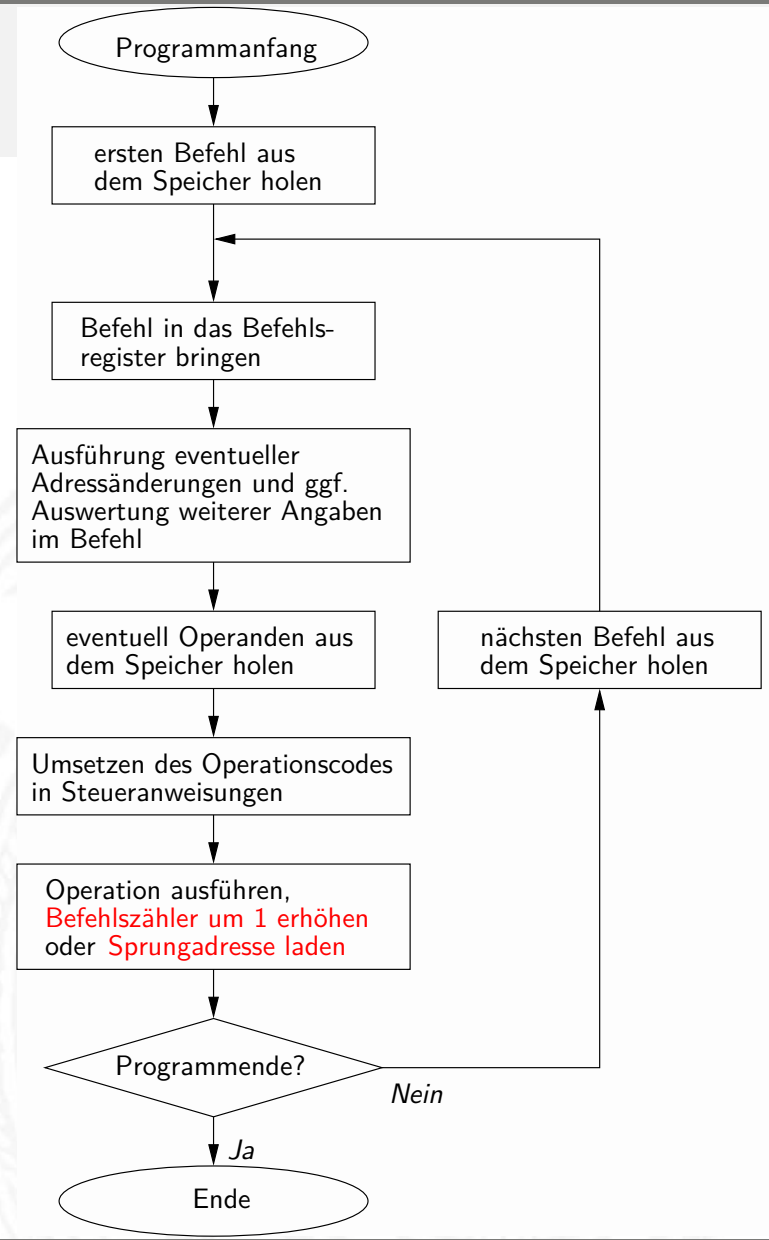
- Historie: 1945 entwickelt (John von Neumann)
- Abstrakte Maschine mit minimalem Hardwareaufwand
- Komponenten
  - zentralen Recheneinheit: CPU
    - logisch unterteilt in
    1. Datenprozessor / Rechenwerk / Operationswerk
    2. Befehlsprozessor / Leitwerk / Steuerwerk
  - Speicher für Daten und Befehle, fortlaufend adressiert
  - Ein/Ausgabe-Einheit zur Anbindung peripherer Geräte
  - Bussystem(e) verbinden diese Komponenten
  - die Struktur ist unabhängig von dem Problem, das Problem wird durch austauschbaren Speicherinhalt (Programm) beschrieben

von-Neumann Architektur (cont.)



von-Neumann Architektur (cont.)

- Paradigma der Programmverarbeitung
  - Programm als Sequenz elementarer Anweisungen (Befehle)
  - als Bitvektoren im Speicher codiert
  - Interpretation (Operanden, Befehle und Adressen) ergibt sich aus dem Kontext (der Adresse)
  - zeitsequenzielle Ausführung der Instruktionen



von-Neumann Architektur (cont.)

– „von-Neumann Flaschenhals“: Zugriff auf Speicher

... vieles davon gilt Heute noch, außer

- *parallele*, statt sequentieller Befehlsabarbeitung
  Stichwort: superskalare Prozessoren
- dynamisch veränderte Abarbeitungsreihenfolge
  Stichwort: *„out-of-order execution“*
- häufig getrennte Daten- und Instruktionsspeicher
  Stichwort: *Harvard-Architektur*
- *Speicherhierarchie*, Caches etc.

Gliederung

Kriterien beim Entwurf

- Architekt sucht beste Lösung im Suchraum möglicher Entwürfe
- Kriterien „guter“ Architekturen:
  - hohe Rechenleistung
  - zuverlässig, robust
  - modular, skalierbar
  - einfach handhabbar, programmierbar
  - orthogonal
  - ausgewogen
  - wirtschaftlich, adäquat
  - ...

Kriterien beim Entwurf (cont.)

Skalierbarkeit Hinzufügen weiterer Module (ohne zusätzliche Änderungen) verbessert das System
⇒ Erweiterbarkeit, Wirtschaftlichkeit

Orthogonalität Jedes Modul hat eine definierte Funktionalität; keine zwei Module bieten die gleiche Funktionalität
⇒ Wartbarkeit, Kosten, Handhabbarkeit

Adäquatheit Die Kosten eines Moduls sind adäquat zur Funktion
⇒ Performance, Kosten

Virtualität Elimination physikalischer Grenzen: virtueller Prozessor, virtueller Speicher, virtuelle Kanäle...
⇒ skalierbar, ausbaubar, einfache Programmierung

Kriterien beim Entwurf (cont.)

Transparenz unwichtige Details werden verborgen
⇒ einfache Programmierung

Performance-Transparenz Änderung der System-Konfiguration ohne die Funktionalität zu beeinflussen
⇒ Skalierbarkeit, Wartbarkeit, Zuverlässigkeit

Größentransparenz Erweiterung des System, so dass sich die Performance verbessert
⇒ Skalierbarkeit, Kosten

Fehlertransparenz System verbirgt, maskiert oder toleriert Fehler
⇒ Zuverlässigkeit

- Die Punkte sind hier für die *Mikroarchitektur* formuliert, gelten aber gleichermaßen für die *ISA*

Kriterien zur Architekturbewertung

Kenngrößen zur Bewertung

- Taktfrequenz
- Werte die sich aus Eigenschaften der Architektur ergeben
- Ausführungszeiten von Programmen
- Durchsätze
- statistische Größen
- ...

Die Wahl der Kenngrößen hängt entscheidend von der jeweiligen Zielsetzung ab

Kriterien zur Architekturbewertung (cont.)

Verfahren zur Bestimmung der Kenngrößen

- *Benchmarking*: Laufzeitmessung bestehender Programme
  - Standard Benchmarks
    SPEC Standard Performance Evaluation Corporation
    `http://www.spec.org`
    TPC Transaction Processing Performance Council
    `http://www.tpc.org`
  - profilspezifische Benchmarks: SysMark, PCmark, Winbench etc.
  - benutzereigene Anwendungsszenarien
- *Monitoring*: Messungen während des Betriebs
- Modelltheoretische Verfahren: Analytische Modelle, Simulation...

Kenngrößen

Taktfrequenz

- In den letzten Jahren erfolgreich beworben!

⇒ *für die Leistungsbewertung aber völlig ungeeignet*

theoretische Werte

- MIPS – **M**illion **I**nstructions **P**er **S**econd
- MFLOPS – **M**illion **Fl**oating Point **O**perations **P**er **S**econd
  - keine Angabe über die Art der Instruktionen und deren Ausführungszeit
  - nicht direkt vergleichbar
- innerhalb einer Prozessorfamilie sinnvoll

Kenngrößen (cont.)

Ausführungszeit

- Benutzer: *Wie lange braucht mein Programm?*
- Gesamtzeit: Rechenzeit +
  Ein-/Ausgabe, Platten- und Speicherzugriffe...
- CPU-Zeit: Unterteilung in System- und Benutzer-Zeit

Unix `time`-Befehl: `597.07u 0.15s 9:57.61 99.9%`

`597.07`	user CPU time [sec.]
`0.15`	system CPU time
`9:57.61`	elapsed time
`99.9`	CPU/elapsed [%]

Kenngrößen (cont.)

Theoretische Berechnung der CPU-Zeit (user CPU time)

- CPU-Zeit $= IC \cdot CPI \cdot T$

IC	Anzahl auszuführender Instruktionen	**I**nstruction **C**ount
CPI	mittlere Anzahl Takte pro Instruktionen **C**ycles **p**er **I**nstruction	
T	Taktperiode	

- IC kleiner: weniger Instruktionen
  - bessere Algorithmen
  - bessere Compiler
  - mächtigerer Befehlssatz

Kenngrößen (cont.)

- *CPI* kleiner: weniger Takte pro Instruktion
  - parallel Befehle ausführen: VLIW...
  - parallel Teile der Befehle bearbeiten: Pipelining, Superskalar...
- *T* kleiner: höhere Taktfrequenz
  - Technologie
- genauere Untersuchung wenn CPI über die Häufigkeiten und Zyklenanzahl einzelner Befehle berechnet wird
- so lassen sich beispielsweise alternative Befehlssätze miteinander vergleichen

Kenngrößen (cont.)

CPU-Durchsatz

- RZ-Betreiber
  - *Wie viele Aufträge kann die Maschine gleichzeitig verarbeiten?*
  - *Wie lange braucht ein Job im Mittel?*
  - *Wie viel Arbeit kann so pro Tag erledigt werden?*

$\Rightarrow$ Latenzzeit: *Wie lange dauert es, bis mein Job bearbeitet wird?*

$\Rightarrow$ Antwortzeit: *Wie lange rechnet mein Job?*

- Modellierung durch Warteschlangentheorie: Markov-Ketten, stochastische Petri-Netze...

Kenngrößen (cont.)

statistische Werte zur Zuverlässigkeit

- Betriebssicherheit des Systems: „Quality of Service“
- Fehlerrate: Fehlerursachen pro Zeiteinheit
  Ausfallrate: Ausfälle pro Zeiteinheit
  - *Fault:* Fehlerursache
  - *Error:* fehlerhafter Zustand
  - *Failure:* ein Ausfall ist aufgetreten
- MTTF Mean Time To Failure
  MTBF Mean Time Between Failures
  MTTR Mean Time To Repair
  MTBR Mean Time Between Repairs
- ...

Bewertung von Maßnahmen

Wie wirken sich Verbesserungen der Rechnerarchitektur aus?

- Speed-Up: Verhältnis von Ausführungszeiten *vor* und *nach* der Verbesserung

  Speed-Up $= T_{\mathbf{v}orVerbesserung} / T_{\mathbf{n}achVerbesserung}$
- werden nur Teile der Berechnung beschleunigt, Faktor F:

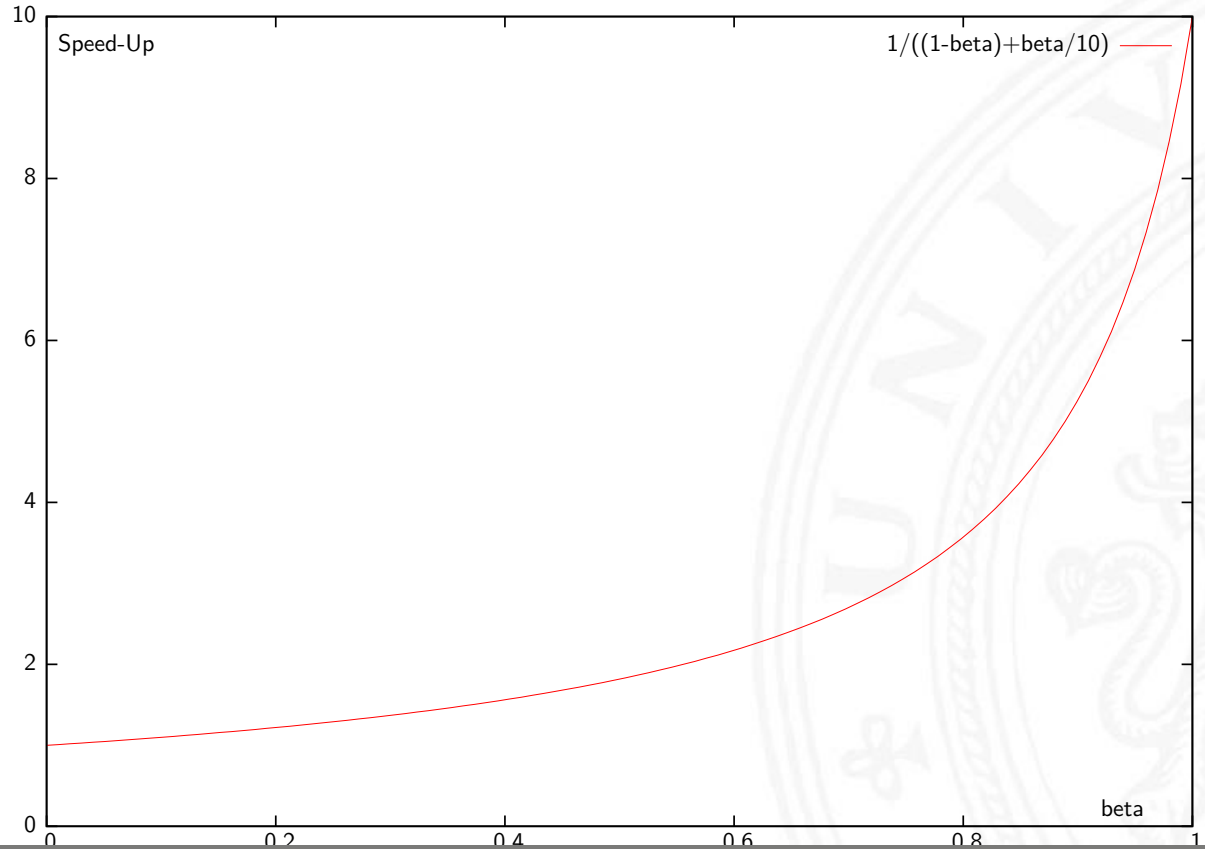
$$\begin{aligned} T &= T_{\mathbf{o}hneEffekt} + T_{\mathbf{m}itEffekt} \\ T_n &= T_{v,o} + T_{v,m}/F_{Verbesserung} \end{aligned}$$

$\Rightarrow$ den „Normalfall“, den häufigsten Fall beschleunigen, um den größten Speed-Up zu erreichen

Bewertung von Maßnahmen (cont.)

- Amdahlsches Gesetz (s.u.)

$$\text{Speed-Up} = \frac{1}{(1-\beta)+\beta/F_{Verbesserung}} \text{ mit } \beta = T_{v,m}/T_v$$



$F_{Verbesserung} = 10$

Bewertung von Maßnahmen (cont.)

Amdahlsches Gesetz

- Beschleunigung der Bearbeitung durch Parallelausführung mit N Prozessoren (Gene Amdahl '67)
- Speed-Up $= \frac{1}{(1-\beta)+f_k(N)+\beta/N} \leq \frac{1}{(1-\beta)}$

N	# Prozessoren als Verbesserungsfaktor
β	Anteil parallelisierbarer Berechnung
$1-\beta$	Anteil nicht parallelisierbarer Berechnung
$f_k()$	Kommunikationsoverhead zwischen den Prozessoren

  - Aufgaben verteilen
  - Arbeit koordinieren
  - Ergebnisse zusammensammeln

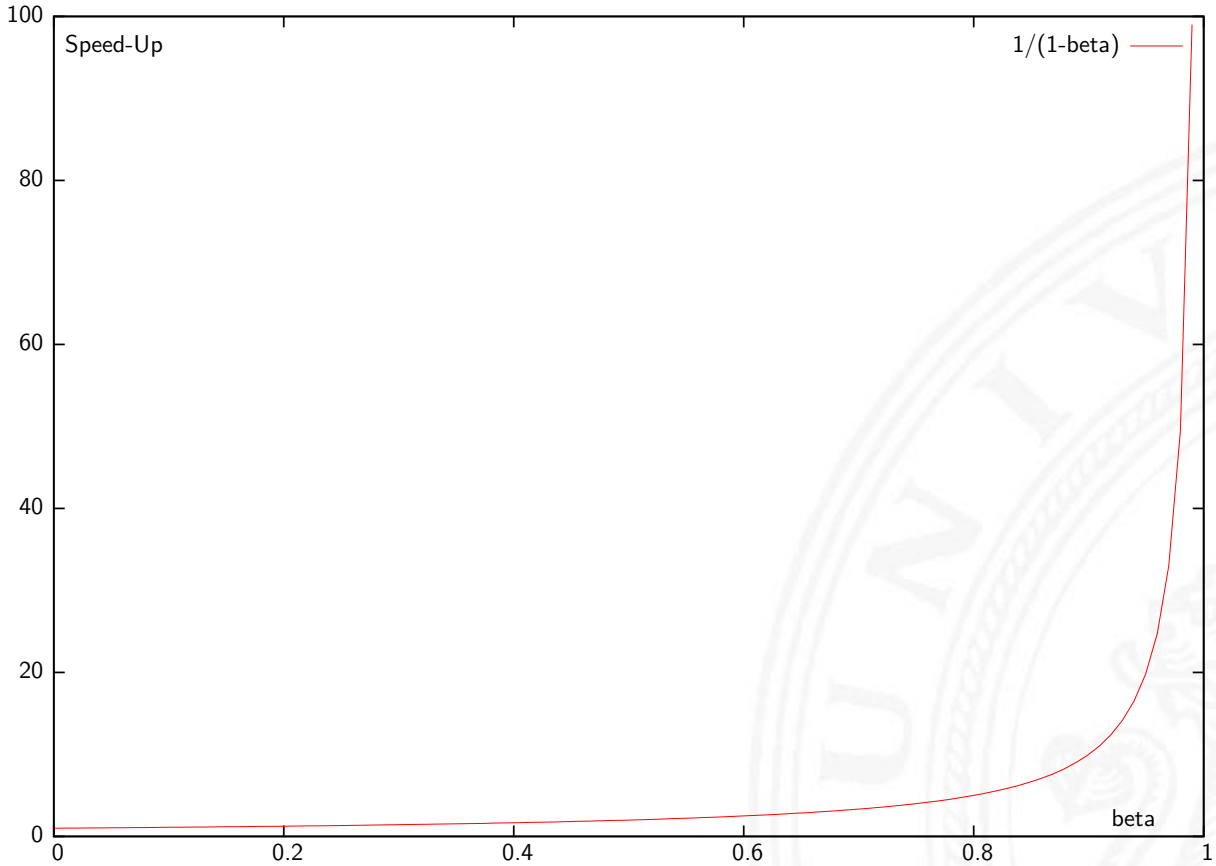
Bewertung von Maßnahmen (cont.)

- *Welche Auswirkungen hat das in der Praxis?*

N	β	Speed-up
2	0,40	1,25
4	0,40	1,43
4536	0,80	5,00
9072	0,99	98,92

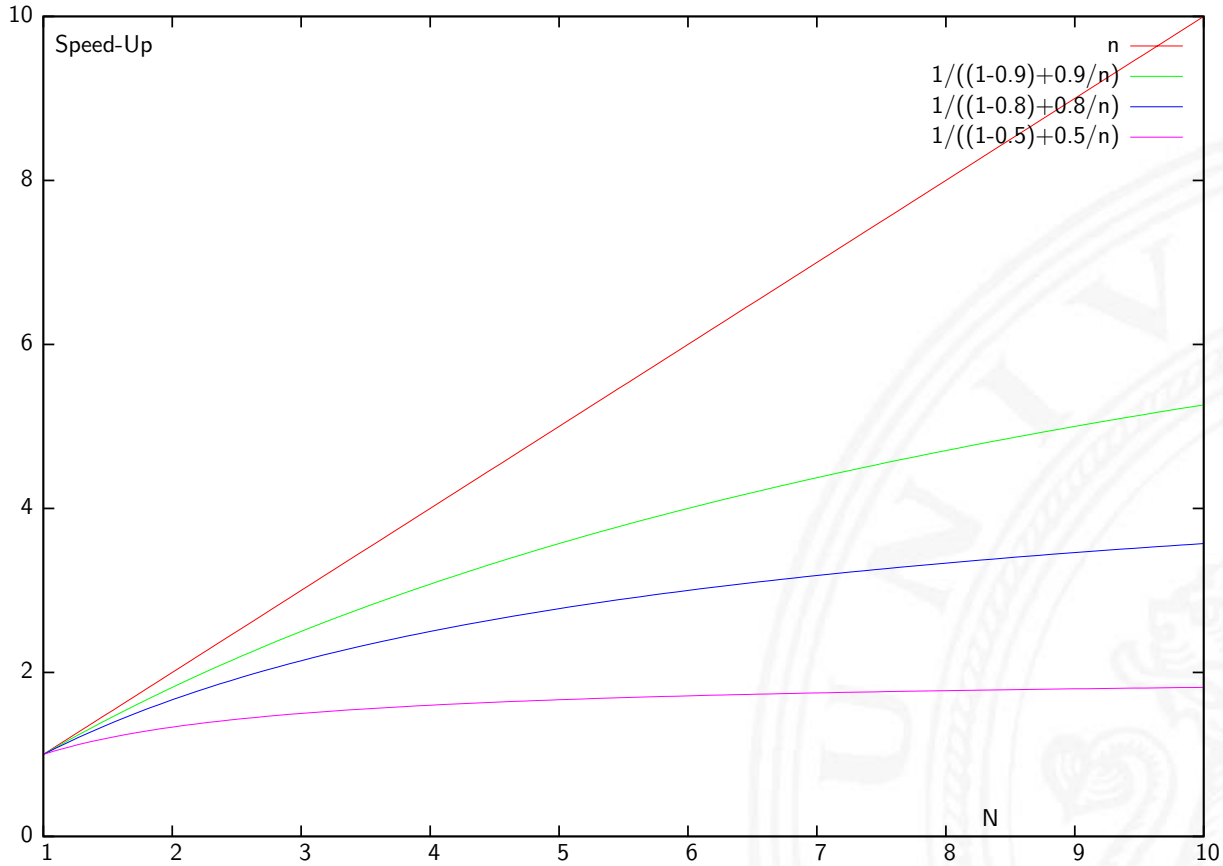
  - Enttäuschend: zwei Prozessoren sind *nicht* doppelt so schnell
  - . . . immerhin bei Multitasking und mehreren Prozessen kommt man auf große β

Bewertung von Maßnahmen (cont.)



$$F_{Verbesserung} = \infty$$

Bewertung von Maßnahmen (cont.)



Gliederung

Instruction Set Architecture

ISA – **I**nstruction **S**et **A**rchitecture

- Schnittstelle zur Hardware für Programmierer/Compiler
- Definiert durch
  - konzeptionelle Struktur
  - funktionales Verhalten

Compilerschnittstelle

- legt fest, was Compiler ausdrücken kann
- „Wortschatz des Compilers“
- definiert was Hardware ausführen können muss

Instruction Set Architecture (cont.)

Programmierer-Sicht des Computers

- Wie werden Daten und Datenstrukturen repräsentiert?
- Wo können Daten gespeichert werden?
- Wie kann auf die Daten zugegriffen werden?
- Welche Operationen können auf den Daten ausgeführt werden?
- Wie sind die Instruktionen codiert?
- Wie ist die Organisation der (sichtbaren) Register?
- ...

Instruction Set Architecture (cont.)

Anwendungsprogrammierer / User-Level

- direkte I/O-Befehle nicht erlaubt
- kein Zugriff auf Speicherverwaltung (Segmentierung / Paging)
- kein Zugriff auf Interrupt-, Exception- und System-Call-Handling
- stattdessen *System-Calls*

(Betriebs-) Systemprogrammierer / System-Level

- direkte I/O-Befehle erlaubt
- Zugriff auf Segmentierung / Paging
- Zugriff auf Interrupt-, Exception- und System-Call-Handling
- System-Calls nicht möglich

Instruction Set Architecture (cont.)

Kompatibilität von ISA

- ▶ Erweiterung bestehender ISA (Bildung von Obermengen)
- \+ Abwärtskompatibilität
  Prozessorfamilien: Intel, Sparc, MIPS...
- \+ lange Software Lebensdauer
- – neue (bessere) Architekturen sind schwierig einzuführen

Inhalte einer ISA

- Syntax
  - Argumente der Befehle und Adressierungsarten
  - Codierung
- Semantik
  - Definition der Auswirkung bei Ausführung
- Daten
  - Binärdarstellung
  - Interpretation
  - Speicherung
- Befehle
  - Formate der Befehle
  - Befehlsklassen
  - Adressierungsarten

ISA – Daten

Datenworte

- Architekturabhängige Speicherwortlängen
Bit	0, 1	
Nibble	4 Bits	
Byte	8 Bits	
Half Word		16 Bits
Word	16 Bits	32 Bits
Long Word	32 Bits	64 Bits
Quad Word	64 Bits	
...		

ISA – Daten (cont.)

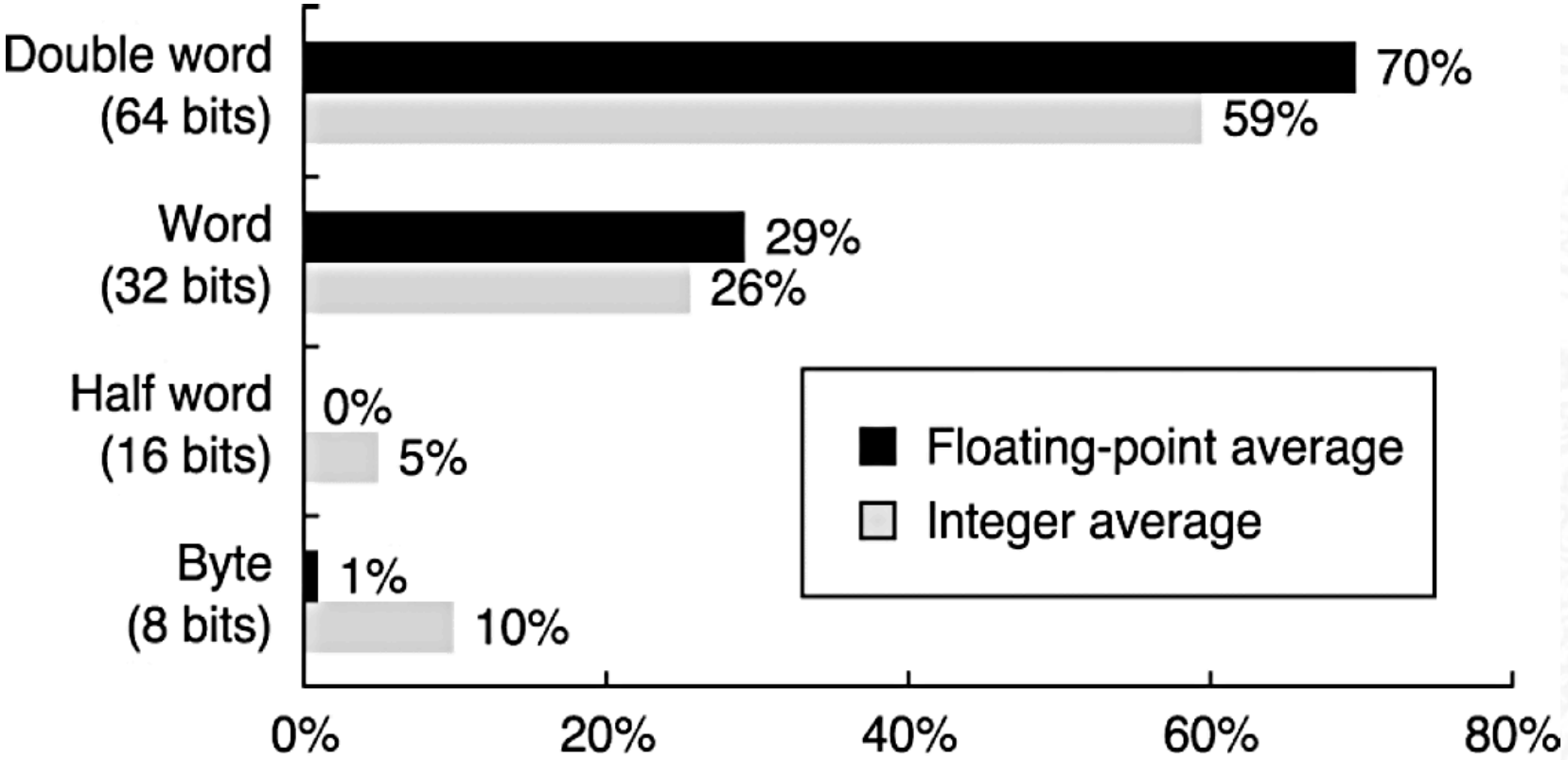
Datentypen

- Interpretation der Worte
Character	ASCII 7 Bit Code. . .
Decimal	BCD-Ziffern: paarweise in 8 Bits. . .
Integer	2er-Komplement. . .
Floating-Point	IEEE 754: Fliesskommastandard
	einfache, doppelte, erweiterte Genauigkeit. . .

  . . .

ISA – Daten (cont.)

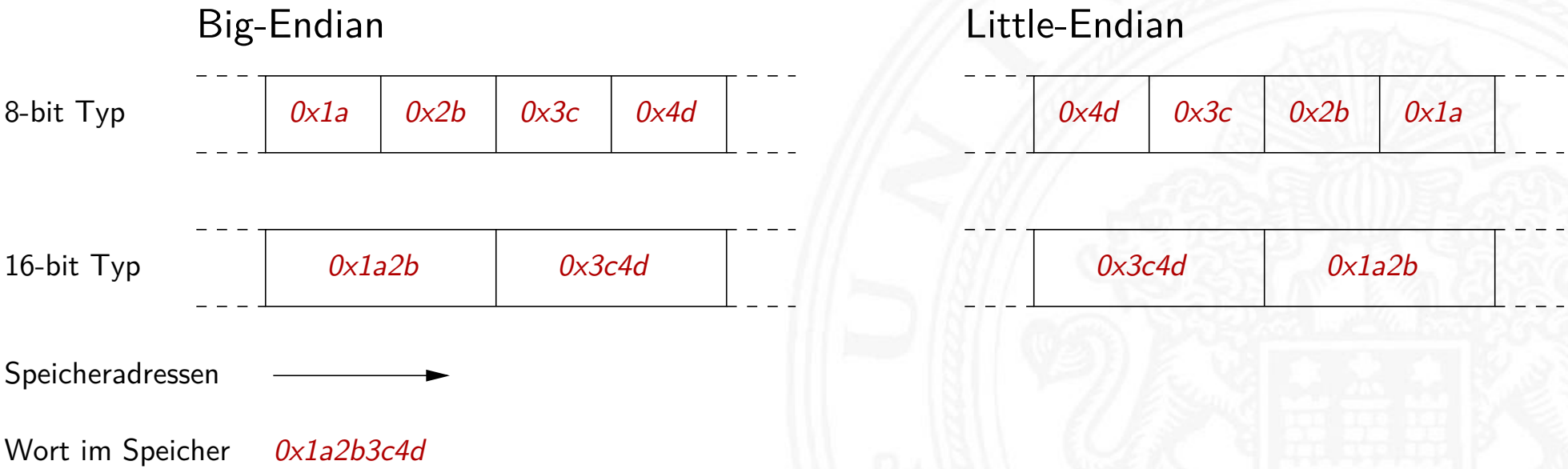
- Verteilung von Wortlängen und Datentypen



ISA – Daten (cont.)

Zuordnung der Datentypen zu Speicheradressen

- Hauptspeicher meist Byte-adressiert
  aber größere Wortbreite (z.B. 32-bit)
- Byte Reihenfolge / „*Endian*“



ISA – Befehle

Befehlsformate / Anzahl der Speicherworte

- variabel: befehlsabhängige Anzahl von Worten
  - CISC-Befehlssätze
  - \+ mächtige Befehle möglich
  - – komplexe Kontrollstruktur
  - – Pipelining schwierig
- fest: einheitliche Formatgröße
  - RISC-Befehlssätze
  - \+ einheitliches Konzept
  - \+ Befehlsabarbeitung sehr gut in Pipeline integrierbar

ISA – Befehle (cont.)

Befehlsklassen

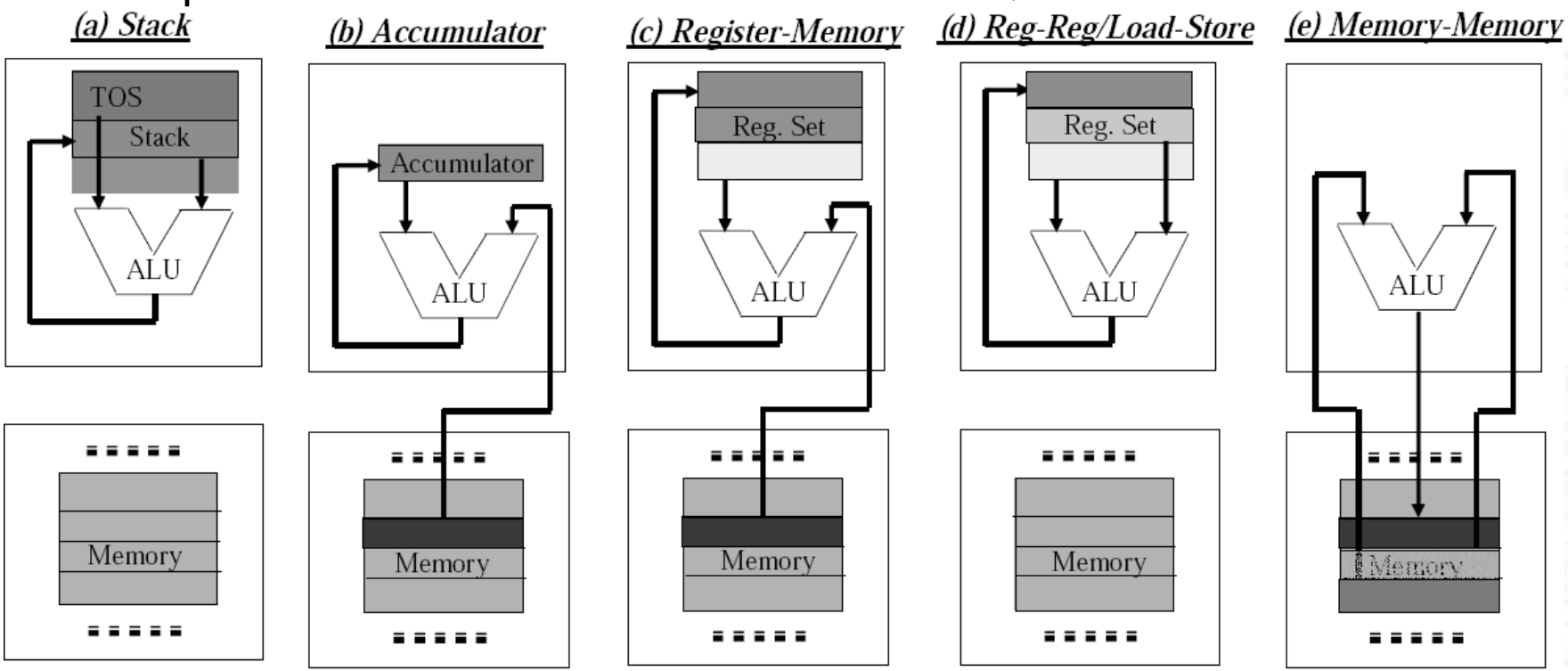
- Datentransfer: Speicher $\leftrightarrow$ Register, Ein-/Ausgabe
- arithmetische und logische Operationen
- Ablaufsteuerung: unbedingte/bedingte Sprünge, Unterprogrammaufrufe
- Systembefehle

Anzahl der Adressen im Befehl

- 0 Stack Operationen
  1 definiertes Akkumulator-Register und $\langle op1 \rangle$
  2 zwei Operanden: $\langle op1 \rangle$, $\langle op2 \rangle$
  3 drei Operanden: $\langle op1 \rangle$, $\langle op2 \rangle$, $\langle op3 \rangle$
- Operanden $\langle op \rangle$ jeweils Register oder Speicher

ISA – Befehle (cont.)

- Beispiel: 0- bis 3-Adress Befehle $C \leftarrow A + B$




```
Push A
Push B
Add
Pop C
```

```
Load  A
Add   B
Store C
```

```
Load  R1,A
Add   R1,B
Store R1,C
```

```
Load  R1,A
Load  R2,B
Add   R3,R1,R2
Store R3,C
```

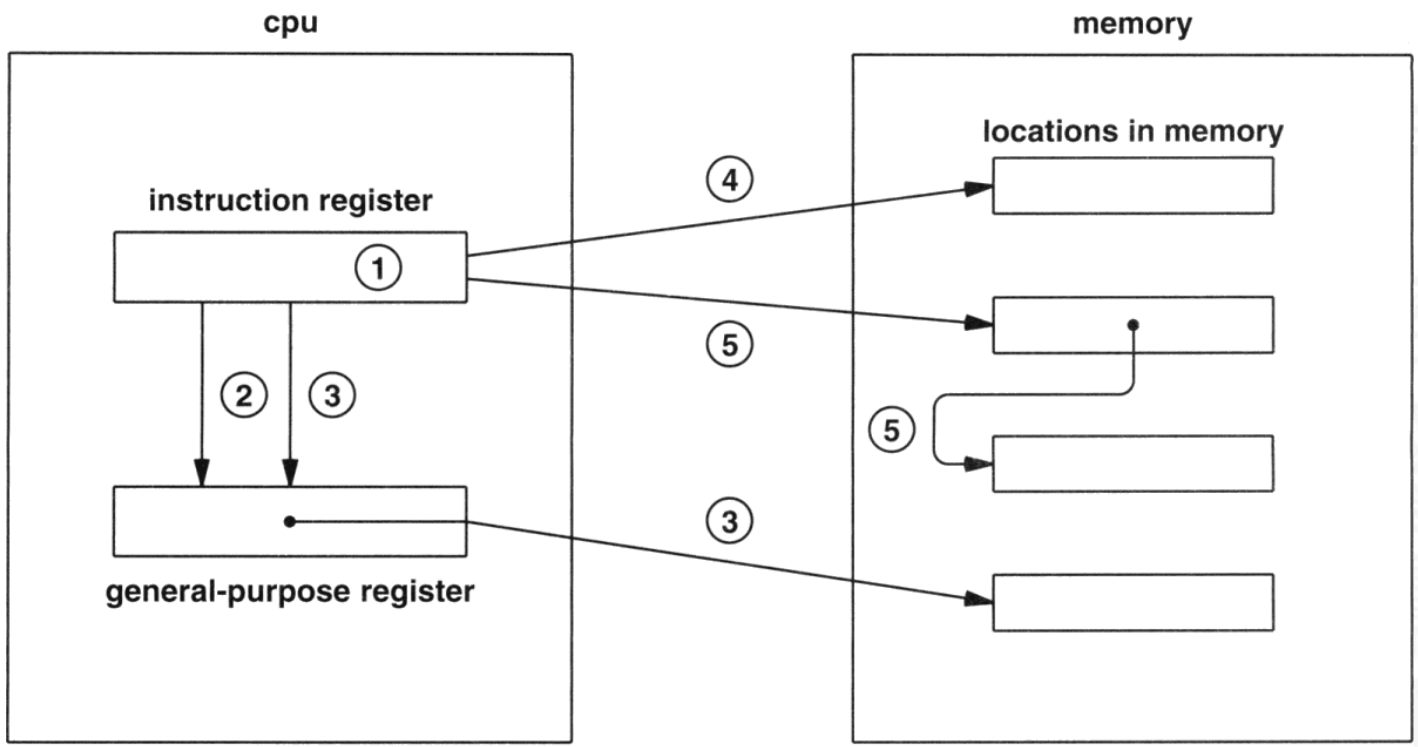
```
Add C,A,B
or Add A,B
```

ISA – Befehle (cont.)

Adressierungsarten

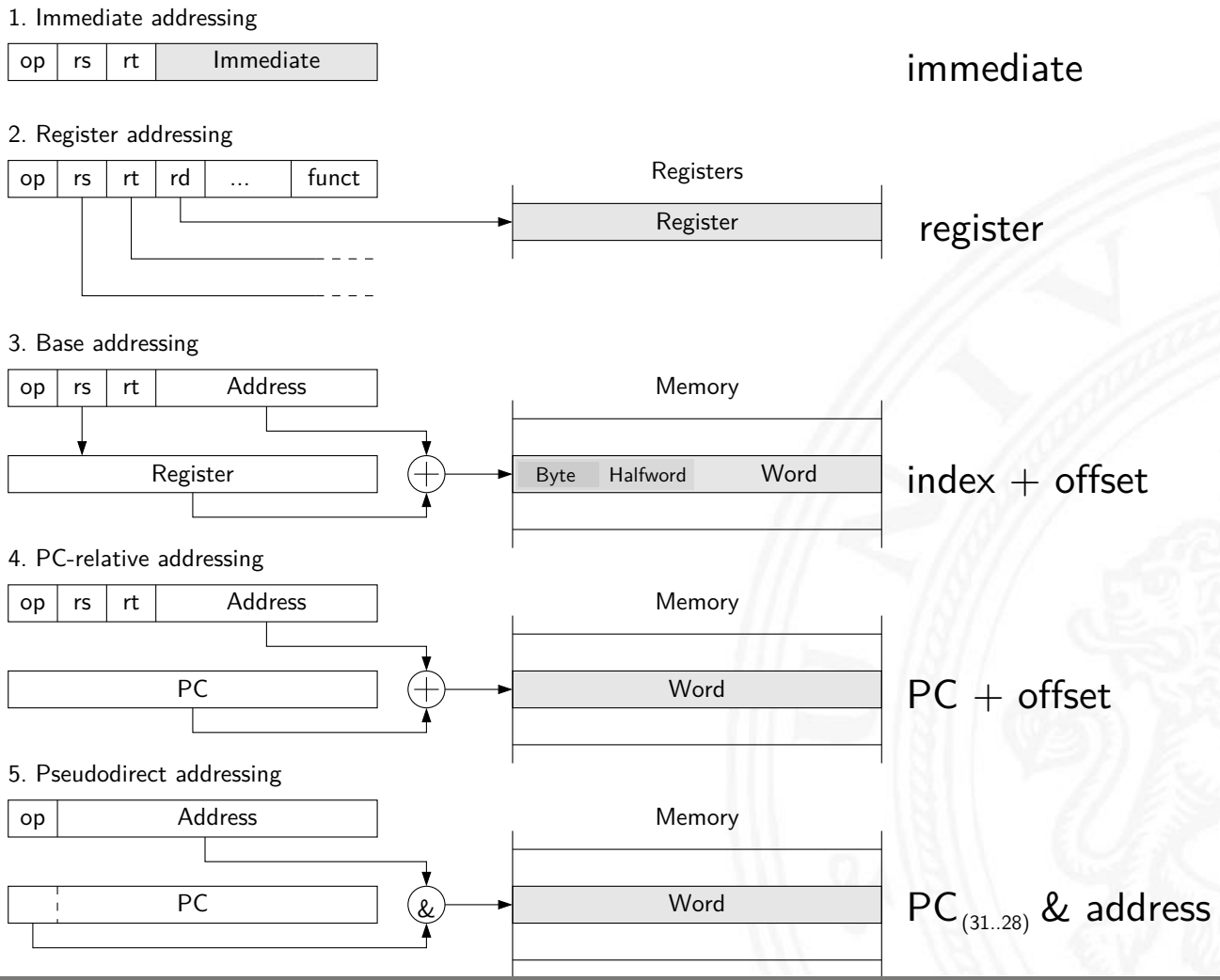
- Speicheradressierung
  - \+ entspricht Programmiermodell
  - – langsam
  - – Operandenlänge
- Registeradressierung
  - \+ Zugriffsgeschwindigkeit
  - – begrenzte Anzahl
  - – Verwaltung
- von grundlegender Architektur abhängig: RISC/CISC
- verschiedene Varianten
  - immediate Operand
  - Memory direct
  - Register direct
  - Indirect
  - Indirect with Offset / with Scale Factor
  - ...

ISA – Befehle (cont.)



1 Immediate value (in the instruction)
2 Direct register reference
3 Indirect through a register
4 Direct memory reference
5 Indirect memory reference

ISA – Befehle (cont.)



ISA – Befehle (cont.)

Ein-/Ausgabe

- direkter Zugriff auf E/A-Kanäle
  - \+ getrennte Adressräume für Speicher und E/A
  - – spezielle Instruktionen notwendig
- *Memory-mapped* Mechanismen
  - \+ Zugriff über „normale“ Load-/Store-Befehle
  - – gemeinsamer Adressraum

Bewertung der ISA

Kriterien für einen *guten* Befehlssatz

- vollständig: alle notwendigen Instruktionen verfügbar
- orthogonal: keine zwei Instruktionen leisten das Gleiche
- symmetrisch: z.B. Addition $\Leftrightarrow$ Subtraktion
- adäquat: technischer Aufwand entsprechend zum Nutzen
- effizient: kurze Ausführungszeiten

Statistiken zeigen: Dominanz der einfachen Instruktionen

Bewertung der ISA (cont.)

- x86-Prozessor

	Anweisung	Ausführungshäufigkeit %
1.	load	22 %
2.	conditional branch	20 %
3.	compare	16 %
4.	store	12 %
5.	add	8 %
6.	and	6 %
7.	sub	5 %
8.	move reg-reg	4 %
9.	call	1 %
10.	return	1 %
Total		96 %

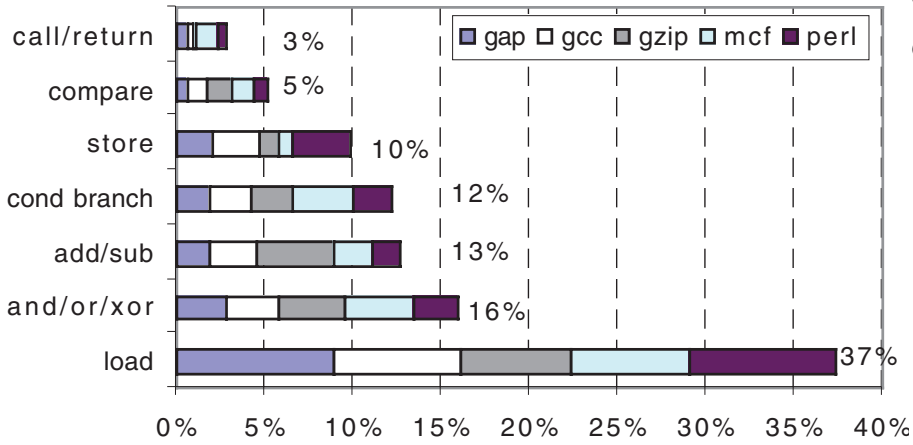
Bewertung der ISA (cont.)

Instruction	compress	eqntott	espresso	gcc (cc1)	li	Int. average
load	20.8%	18.5%	21.9%	24.9%	23.3%	22%
store	13.8%	3.2%	8.3%	16.6%	18.7%	12%
add	10.3%	8.8%	8.15%	7.6%	6.1%	8%
sub	7.0%	10.6%	3.5%	2.9%	3.6%	5%
mul				0.1%		0%
div						0%
compare	8.2%	27.7%	15.3%	13.5%	7.7%	16%
mov reg-reg	7.9%	0.6%	5.0%	4.2%	7.8%	4%
load imm	0.5%	0.2%	0.6%	0.4%		0%
cond. branch	15.5%	28.6%	18.9%	17.4%	15.4%	20%
uncond. branch	1.2%	0.2%	0.9%	2.2%	2.2%	1%
call	0.5%	0.4%	0.7%	1.5%	3.2%	1%
return, jmp indirect	0.5%	0.4%	0.7%	1.5%	3.2%	1%
shift	3.8%		2.5%	1.7%		1%
and	8.4%	1.0%	8.7%	4.5%	8.4%	6%
or	0.6%		2.7%	0.4%	0.4%	1%
other (xor, not, . . .)	0.9%		2.2%	0.1%		1%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
compare FP						0%
mov reg-reg FP						0%
other (abs, sqrt, . . .)						0%

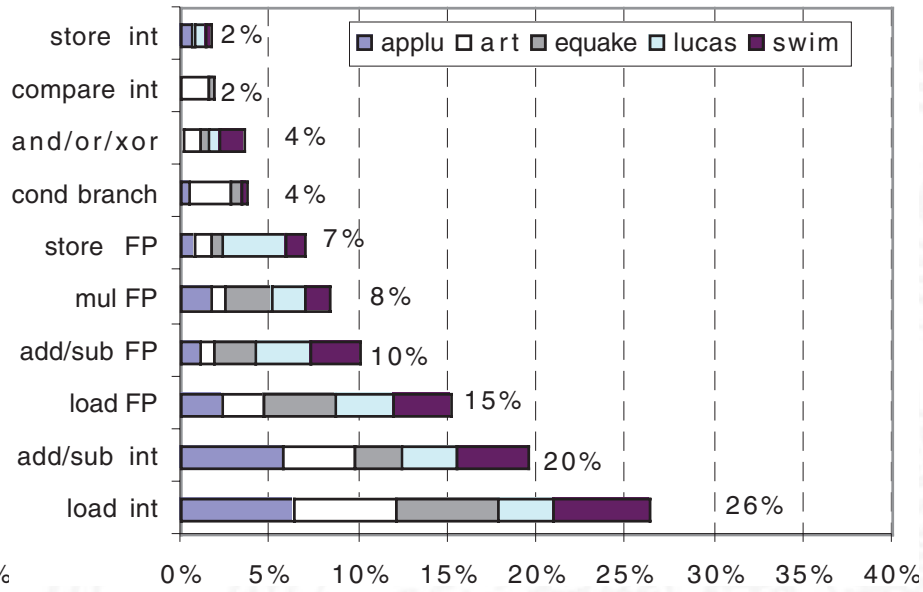
Figure D.15 80x86 instruction mix for five SPECint92 programs.

Bewertung der ISA (cont.)

- MIPS-Prozessor



SPECint2000 (96%)



SPECfp2000 (97%)

Bewertung der ISA (cont.)

- ca. 80 % der Berechnungen eines typischen Programms verwenden nur ca. 20 % der Instruktionen einer CPU
- am häufigsten gebrauchten Instruktionen sind einfache Instruktionen: load, store, add...

$\Rightarrow$ Motivation für RISC

CISC

CISC – **C**omplex **I**nstruction **S**et **C**omputer

- Motivation
  - aus der Zeit der ersten Großrechner, 60er Jahre
  - Programmierung auf Assemblerebene
  - Komplexität durch sehr viele (mächtige) Befehle umgehen
- Eigenschaften
  - Instruktionssätze mit mehreren hundert Befehlen (> 300)
  - sehr viele Adressierungsarten, -Kombinationen
  - verschiedene Instruktionsformate
  - viele Befehle können auf den Speicher zugreifen

CISC (cont.)

- Konsequenzen
  - \+ nah an der Programmiersprache, einfacher Assembler
  - \+ kompakter Code: weniger Befehle holen, kleiner I-Cache
  - – Pipelining schwierig
  - – Ausführungszeit abhängig von: Befehl, Adressmodi...
  - – Instruktion holen schwierig, da variables Instruktionsformat
  - – Speicherhierarchie schwer handhabbar: Adressmodi
- Mikroprogrammierung
  - ein Befehl kann nicht in einem Takt abgearbeitet werden
  - $\Rightarrow$ Unterteilung in Mikroinstruktionen ($\varnothing$ 5...7)
  - Ablaufsteuerung durch endlichen Automaten
  - meist als ROM (RAM) implementiert, das *Mikroprogrammworte* beinhaltet

RISC

RISC – **R**educed **I**nstruction **S**et **C**omputer

- Grundidee: Komplexitätsreduktion in der CPU
- Historie
  - seit den 80er Jahren: „RISC-Boom“
  - wegen Hochsprachen und Compilereinsatz besteht kein Bedarf mehr für mächtige Assemblerbefehle
  - wegen schnellem Speicher und der Speicherhierarchie muss nicht mehr „möglichst viel“ lokal in der CPU gerechnet werden (CISC, mikroprogrammiert)
- Eigenschaften
  - reduzierte Anzahl der Instruktionen (z.B. 128)
  - nur Load- und Store-Instruktionen können auf Speicher zugreifen
  - alle anderen Operationen arbeiten auf Registern

RISC (cont.)

- Befehlsformate der meisten RISC ISA
  - 3-Adress-Instruktionen
  - Instruktionen in einem Wort (32 Bits) codiert
- Konsequenzen
  - \+ fest-verdrahtete Logik, kein Mikroprogramm
  - \+ einfache Instruktionen, wenige Adressierungsarten
  - \+ Cycles per Instruction = 1
  - \+ Pipelines
  - – längerer Maschinencode
  - – viele Register notwendig
  - optimierende Compiler nötig / möglich
  - High-performance Speicherhierarchie

Instruction Level Parallelism

ILP – **I**nstruction **L**evel **P**arallelism

- Instruktionen in einer *Pipeline* überlappend ausführen
- Instruktionen auf verschiedene Einheiten verteilen
  - zur Laufzeit: Superskalare Prozessoren
  - statisch durch den Compiler: VLIW Prozessoren

\+ viele Optimierungsmöglichkeiten: Loop unrolling, pipeline scheduling, scoreboarding, register renaming, branch prediction, multiple instruction issue, instruction reordering, software pipelining, trace scheduling, speculative execution...

Einige werden im folgenden Abschnitt über Pipelining ab Seite 61 genauer behandelt

Mikroarchitektur

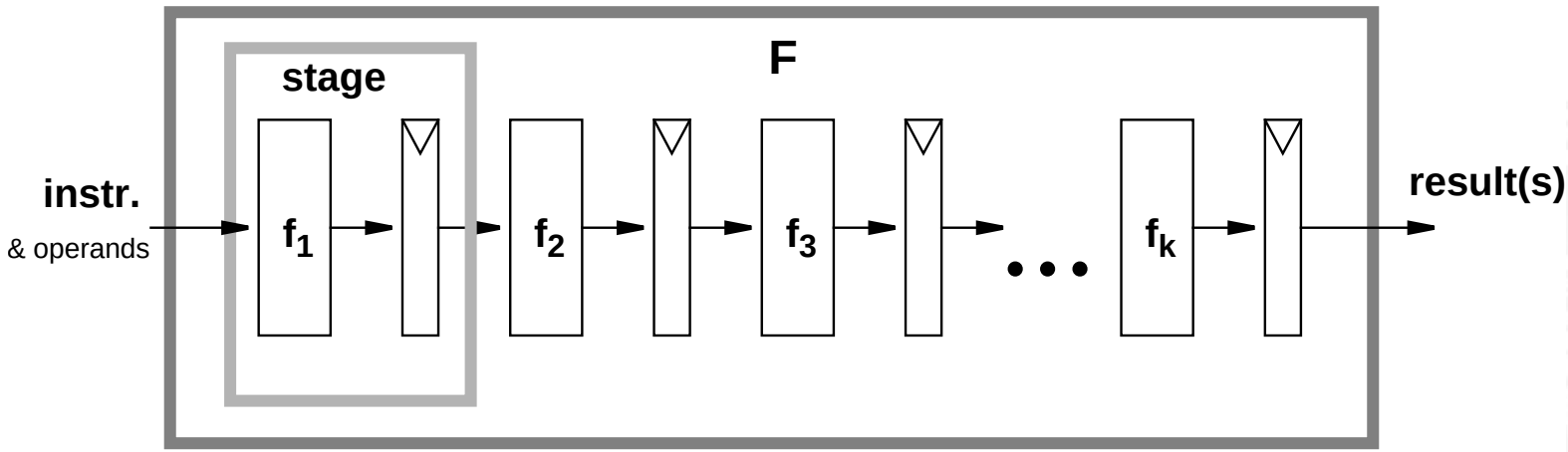
ISA als *Verhaltensbeschreibung*

Mikroarchitektur als *Strukturbeschreibung*

- funktionale Unterteilung: Daten- und Befehlsprozessor
- Hardwareeinheiten
  - Register und Register-Files
  - Arithmetische Einheiten: Integer-, Fliesskomma-ALUs ...
  - Datenpfade: Busse, Multiplexer, Demultiplexer ...
- Steuerung des Ablaufs = Implementierung der ISA
  - CISC: Mikroprogramm
  - RISC: Steuerung der Pipeline durch Kontrollautomat(en)

Gliederung

Pipelining / Fließbandverarbeitung

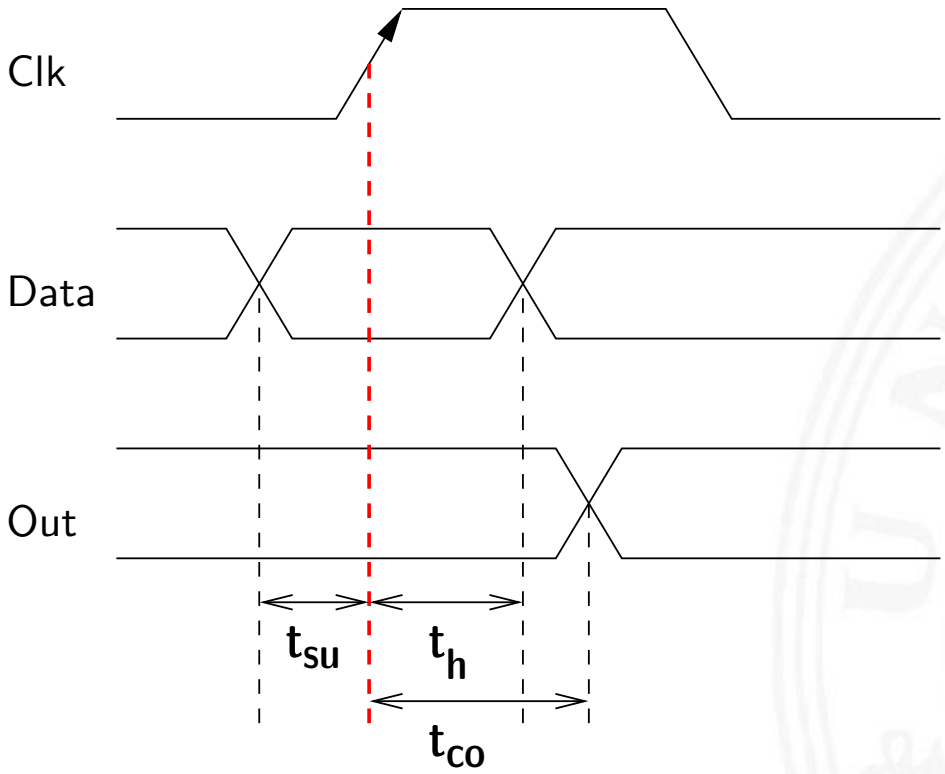


Grundidee

- Operation F kann in Teilschritte zerlegt werden
- jeder Teilschritt f_i braucht ähnlich viel Zeit
- alle Teilschritte f_i können parallel zueinander ausgeführt werden
- Trennung der Pipelinestufen („*stage*“) durch Register

Pipelining / Fließbandverarbeitung (cont.)

- Verglichen mit der Zugriffszeit auf die Register (t_{co}) dauert der Teilschritt f_i lang



Pipelining / Fließbandverarbeitung (cont.)

Arithmetische Pipelines

- Idee: lange Berechnung in Teilschritte zerlegen
  wichtig bei komplizierteren arithmetischen Operationen
  - die sonst sehr lange dauern (weil ein großes Schaltnetz)
  - die als Schaltnetz extrem viel Hardwareaufwand erfordern
  - Beispiele: Multiplikation, Division, Fließkommaoperationen...

\+ Erhöhung des Durchsatzes, wenn Berechnung mehrfach hintereinander ausgeführt wird

(RISC) Prozessorpipelines

- Idee: die Phasen der von-Neumann Befehlsabarbeitung (Befehl holen, Befehl decodieren ...) in eine Pipeline integrieren

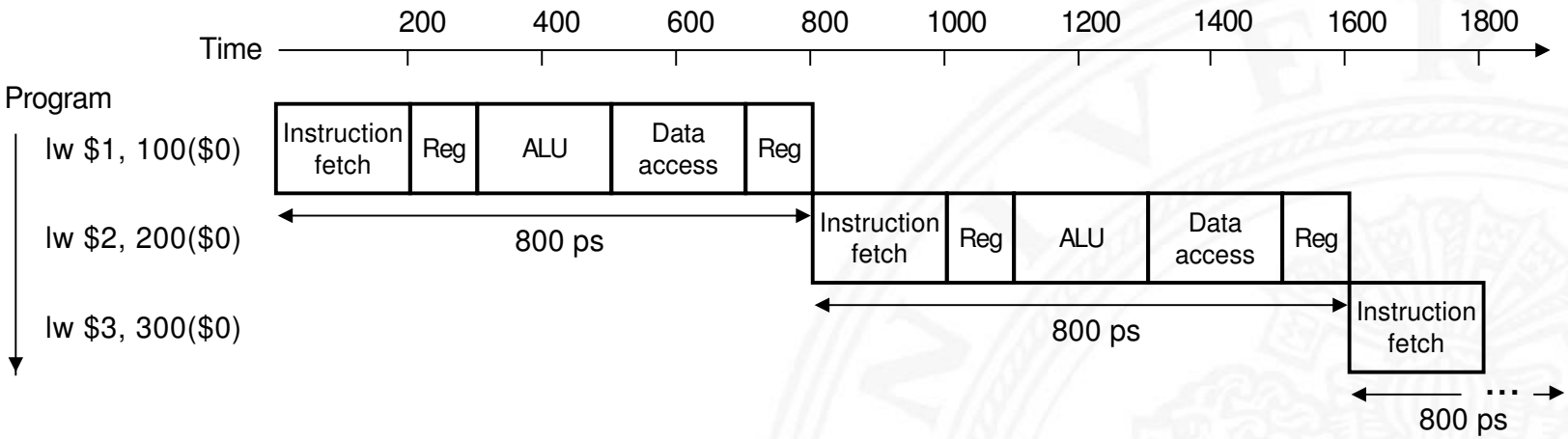
RISC Pipelining

Schritte der RISC Befehlsabarbeitung (von ISA abhängig)

- IF — **I**nstruction **F**etch
  Instruktion holen, in Befehlsregister laden
- ID — **I**nstruction **D**ecode
  Instruktion decodieren
- OF — **O**perand **F**etch
  Operanden aus Registern holen
- EX — **Ex**ecute
  ALU führt Befehl aus
- MEM — **Mem**ory access
  Speicherzugriff bei Load-/Store-Befehlen
- WB — **W**rite **B**ack
  Ergebnisse in Register zurückschreiben

RISC Pipelining (cont.)

- je nach Instruktion sind 3-5 dieser Schritte notwendig
- Beispiel *ohne* Pipelining:

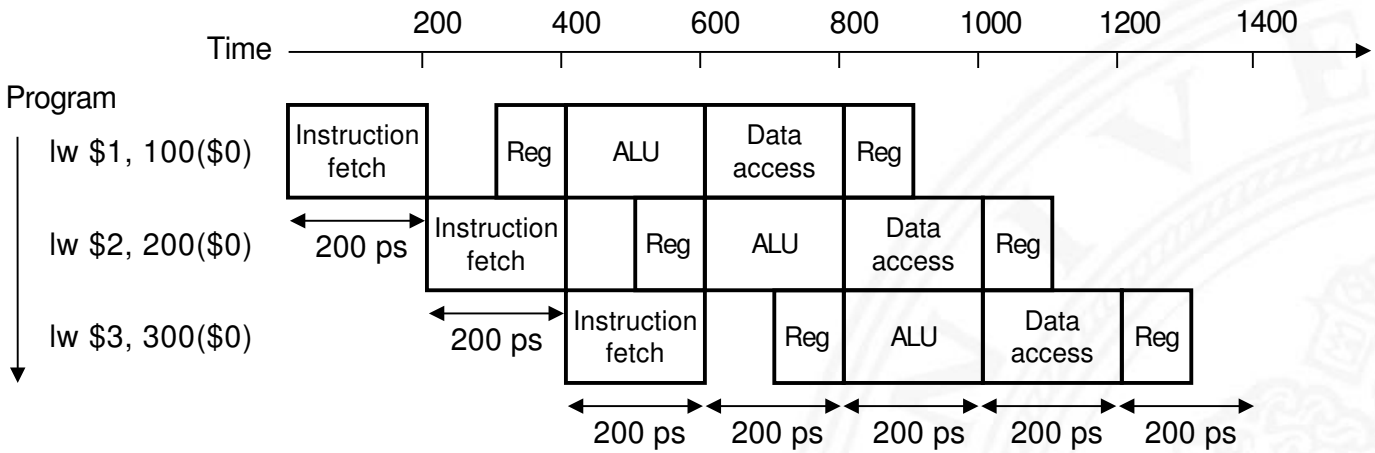


[PH12]

RISC Pipelining (cont.)

Pipelining in Prozessoren

- Beispiel *mit* Pipelining:



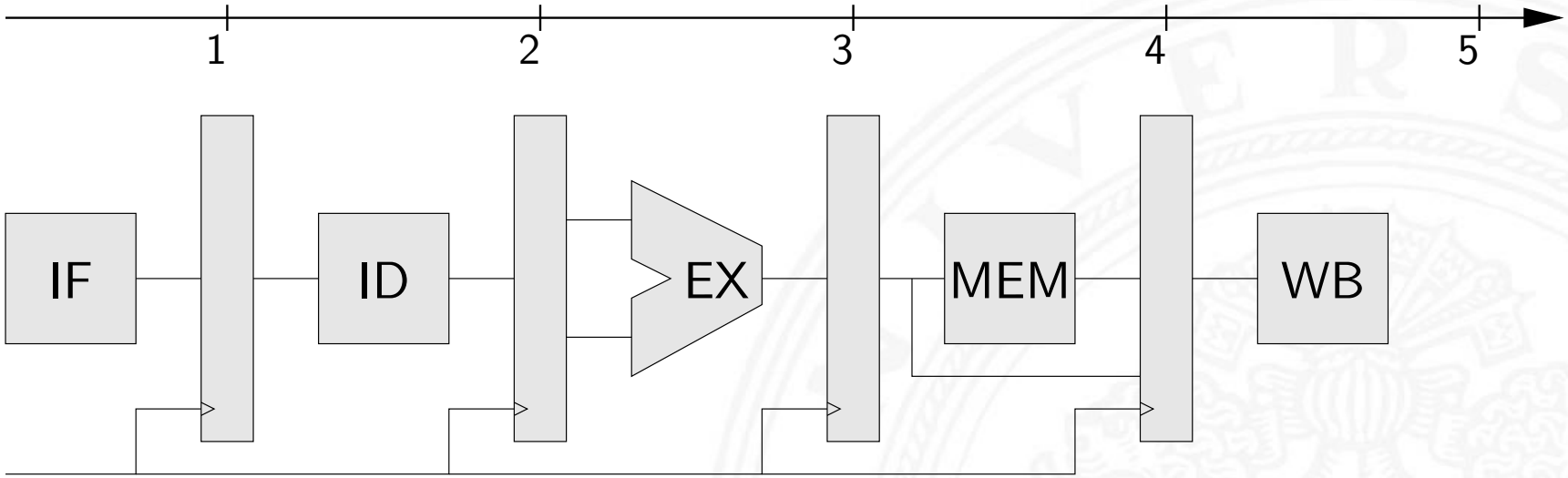
[PH12]

- Befehle überlappend ausführen
- Register trennen Pipelinestufen

RISC Pipelining (cont.)

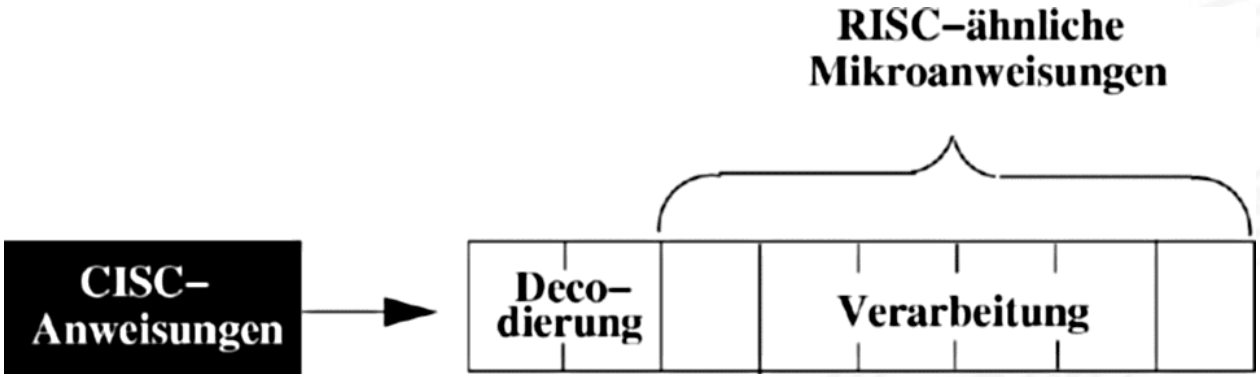
- RISC ISA: Pipelining wird direkt umgesetzt

Pipelinestufen



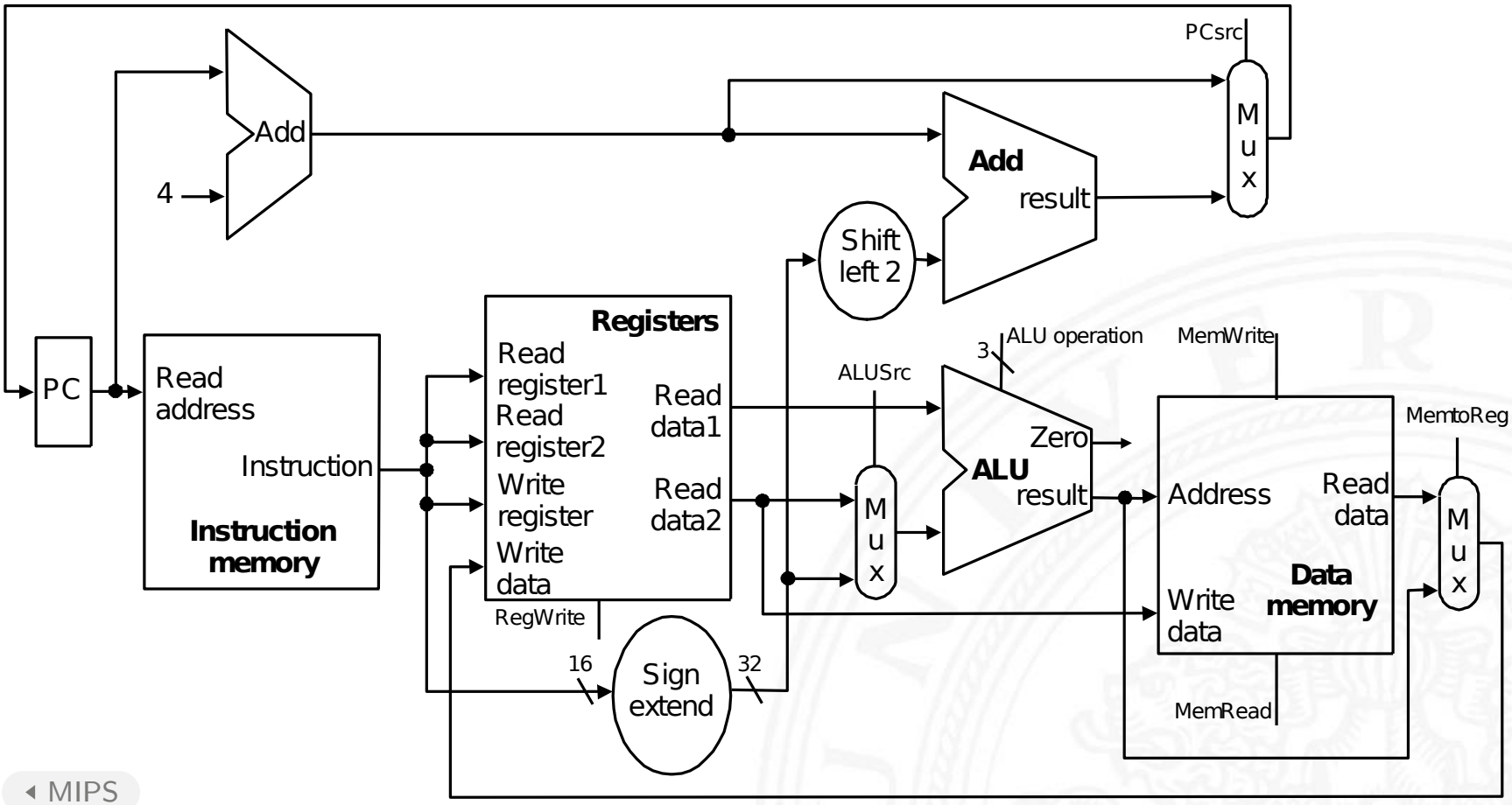
RISC Pipelining (cont.)

- CISC ISA: Umsetzung der CISC Befehle in Folgen RISC-ähnlicher Anweisungen

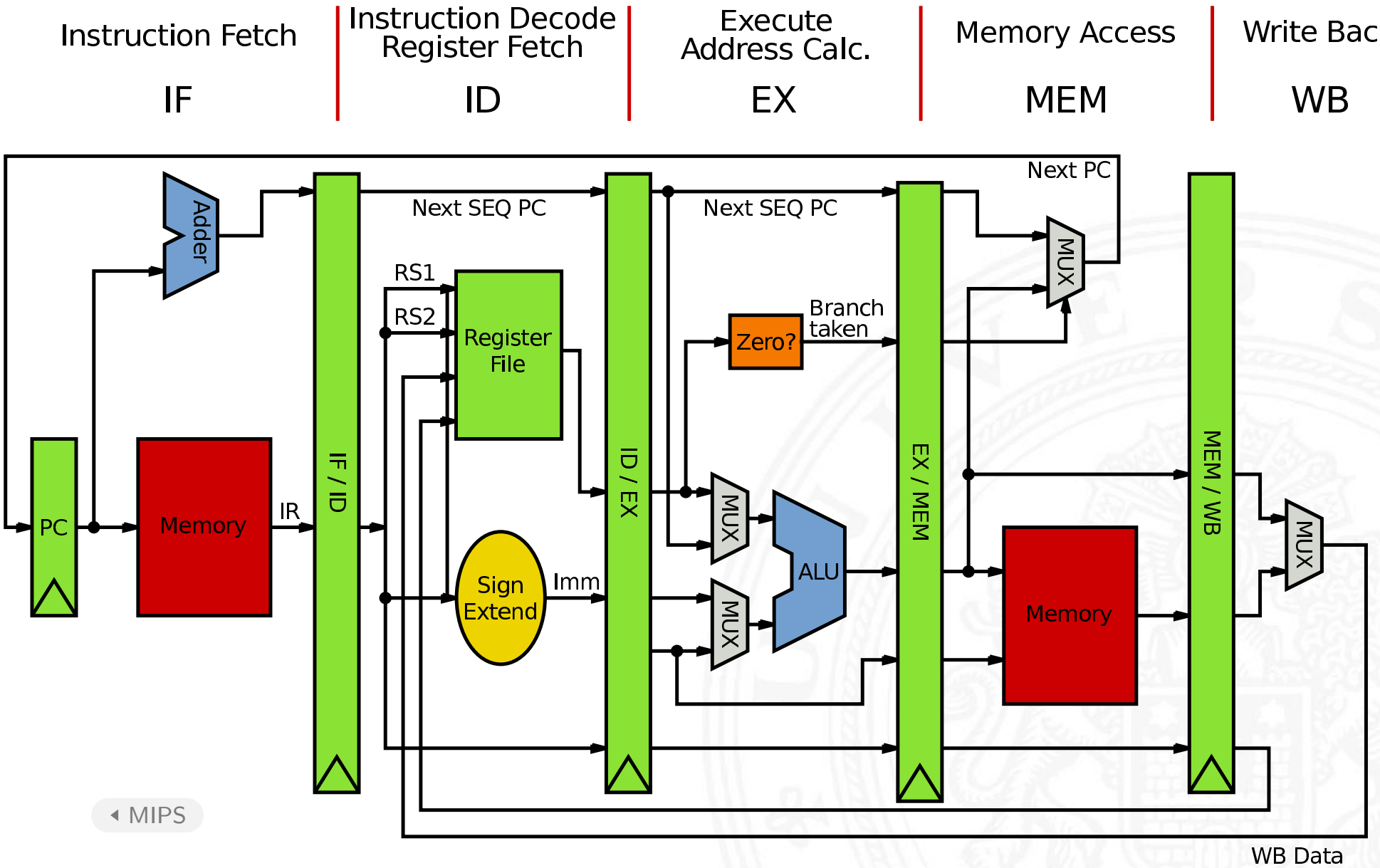


  - \+ CISC-Software bleibt lauffähig
  - \+ Befehlssatz wird um neue RISC Befehle erweitert
- MIPS-Architektur (aus Patterson, Hennessy [PH11, PH12])

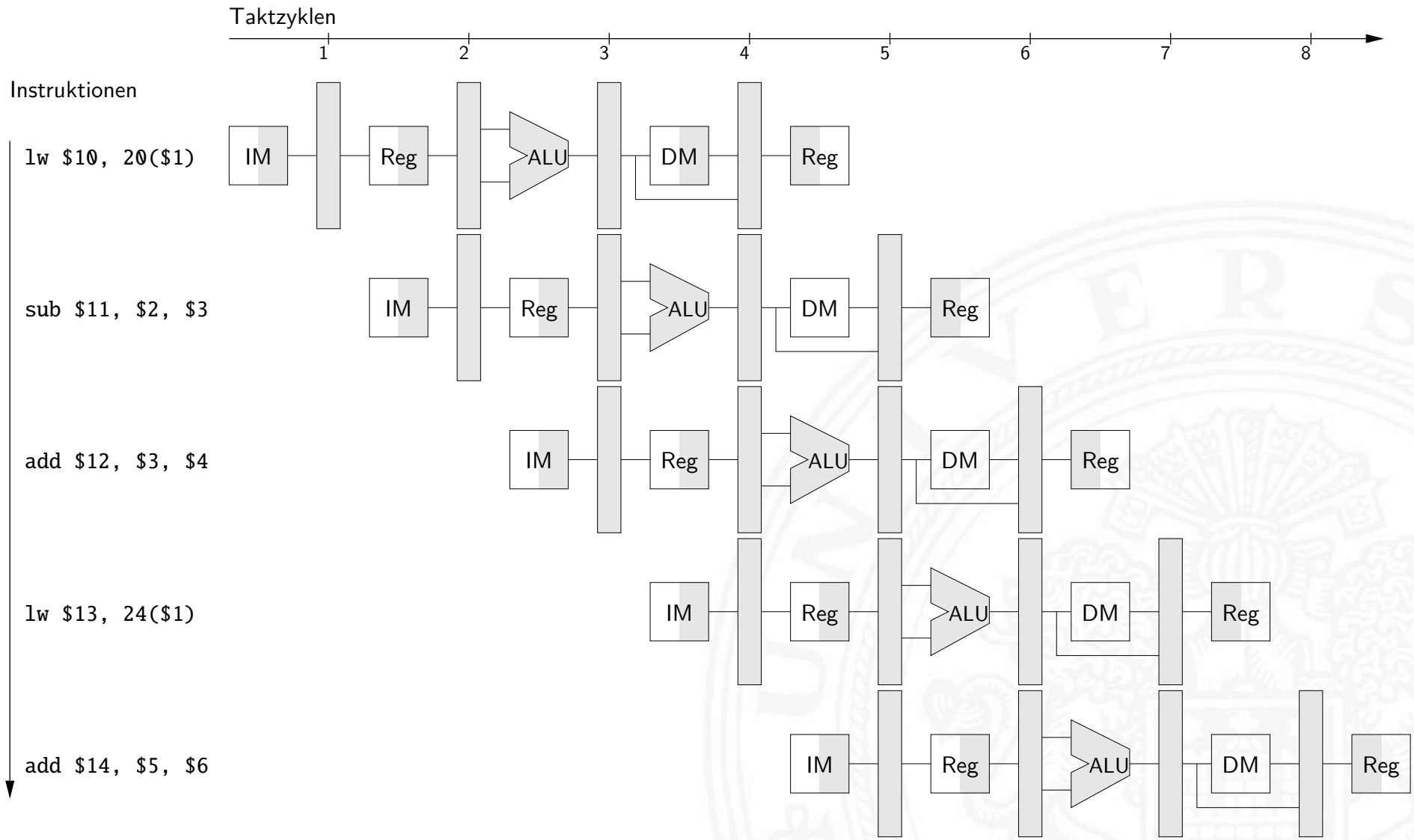
▸ MIPS ohne Pipeline ▸ MIPS Pipeline ▸ Pipeline Schema


PCsrc
Add
4
M
u
x
Add
result
Shift
left 2
PC
Read
address
Instruction
Instruction
memory
Registers
Read
register1
Read
register2
Write
register
Write
data
Read
data1
Read
data2
RegWrite
ALUSrc
M
u
x
3
ALU operation
MemWrite
Zero
ALU
result
Address
Read
data
Write
data
Data
memory
MemRead
MemtoReg
M
u
x
16
Sign
extend
32

◂ MIPS


Instruction Fetch
IF
Instruction Decode
Register Fetch
ID
Execute
Address Calc.
EX
Memory Access
MEM
Write Back
WB
Adder
Next SEQ PC
Next SEQ PC
Next PC
RS1
RS2
Register
File
Zero?
Branch
taken
MUX
PC
Memory
IR
IF / ID
ID / EX
EX / MEM
MEM / WB
Sign
Extend
Imm
ALU
MUX
MUX
Memory
MUX
WB Data

◂ MIPS


Taktzyklen
1
2
3
4
5
6
7
8
Instruktionen
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
IM
Reg
ALU
DM
Reg

◂ MIPS

Prozessorpipeline – Begriffe

Begriffe

- Pipeline-Stage: einzelne Stufe der Pipeline
- Pipeline Machine Cycle: Instruktion kommt einen Schritt in Pipeline weiter
- Durchsatz: Anzahl der Instruktionen, die in jedem Takt abgeschlossen werden
- Latenz: Zeit, die eine Instruktion benötigt, um alle Pipelinestufen zu durchlaufen

Prozessorpipeline – Bewertung

Vor- und Nachteile

+ Pipelining ist für den Programmierer nicht sichtbar!
+ höherer Instruktionsdurchsatz $\Rightarrow$ bessere Performanz
– Latenz wird nicht verbessert, bleibt bestenfalls gleich
– Pipeline Takt limitiert durch langsamste Pipelinestufe
  unausgewogene Pipelinestufen reduzieren den Takt und damit die Performanz
– zusätzliche Zeiten, um Pipeline zu füllen bzw. zu leeren

Prozessorpipeline – Speed-Up

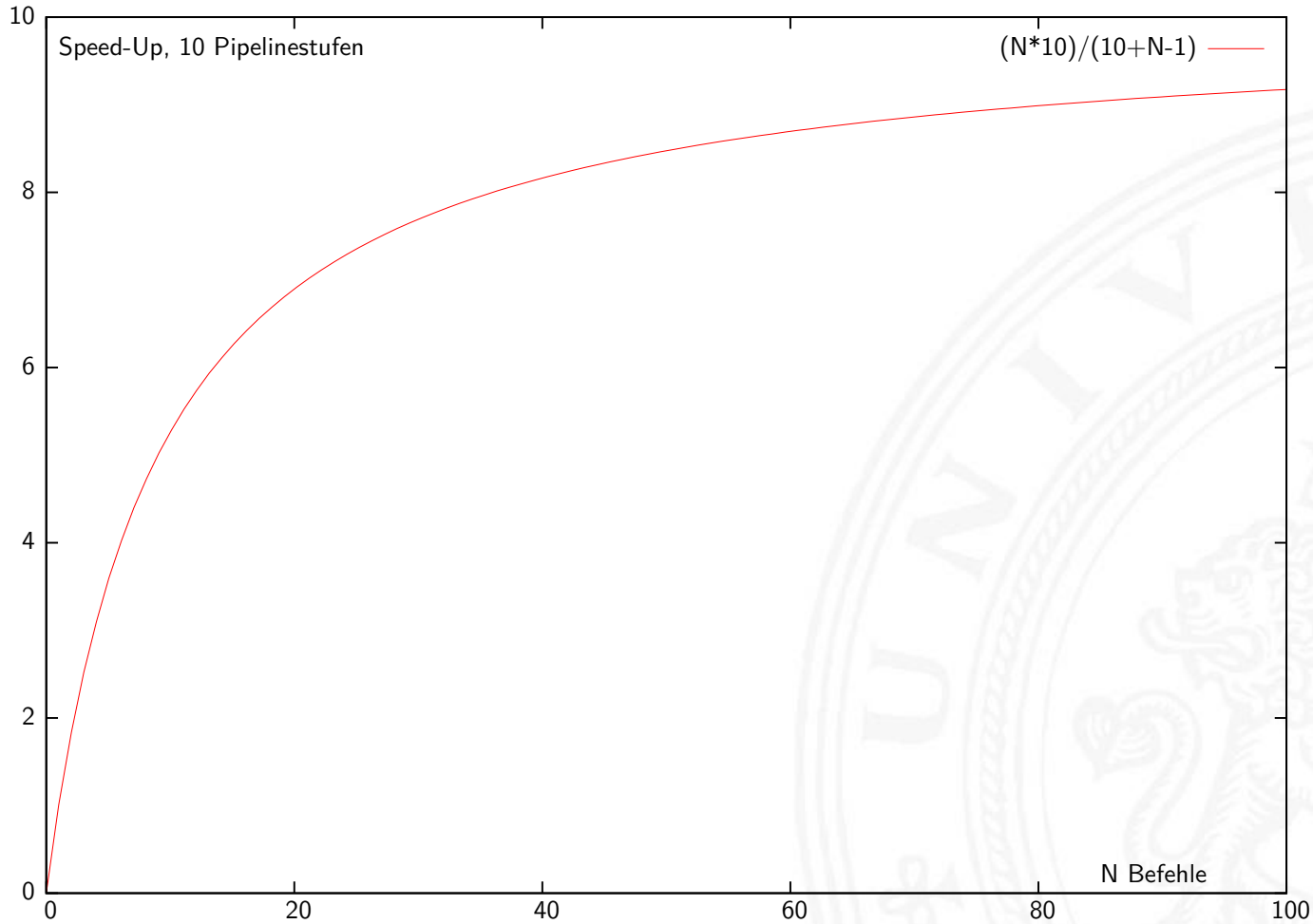
Pipeline Speed-Up

- N Instruktionen; K Pipelinestufen
- ohne Pipeline: $N \cdot K$ Taktzyklen
- mit Pipeline: $K + N - 1$ Taktzyklen
- Speed-Up $= \frac{N \cdot K}{K+N-1}$, $\lim_{N \to \infty} S = K$

$\Rightarrow$ ein großer Speed-Up wird erreicht durch

1. große Pipelinetiefe: K
2. lange Instruktionssequenzen: N

Prozessorpipeline – Speed-Up (cont.)



Prozessorpipeline – Dimensionierung

Dimensionierung der Pipeline

- Längere Pipelines
- Pipelinestufen in den Einheiten / den ALUs (*superskalar*)

$\Rightarrow$ größeres K wirkt sich direkt auf den Durchsatz aus

$\Rightarrow$ weniger Logik zwischen den Registern, höhere Taktfrequenzen

- Beispiele

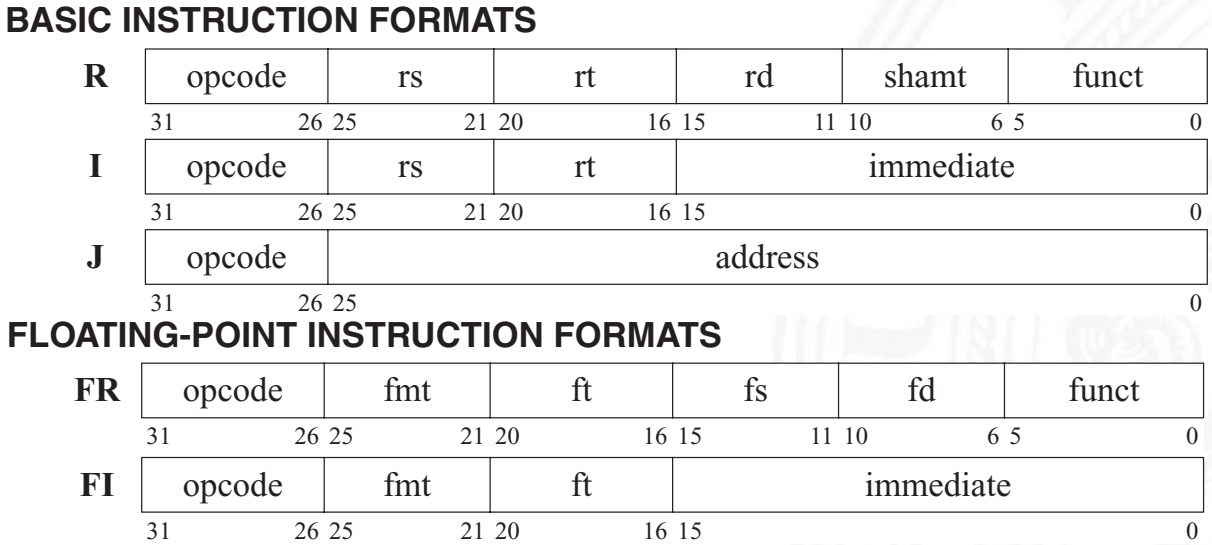
CPU	Pipelinestufen	Taktfrequenz [*MHz*]
Pentium	5	300
Motorola G4	4	500
Motorola G4e	7	1000
Pentium II/III	12	1400
Athlon XP	10/15	2500
Athlon 64, Opteron	12/17	$\leq$ 3000
Pentium 4	20	$\leq$ 5000

Prozessorpipeline – Auswirkungen

Architekturentscheidungen, die sich auf das Pipelining auswirken

gut für Pipelining

- gleiche Instruktionslänge
- wenige Instruktionsformate
- Load/Store Architektur



MIPS-Befehlsformate [PH11, PH12]

Prozessorpipeline – Auswirkungen (cont.)

schlecht für Pipelining: *Pipelinekonflikte / -Hazards*

- Strukturkonflikt: gleichzeitiger Zugriff auf eine Ressource durch mehrere Pipelinestufen
- Datenkonflikt: Ergebnisse von Instruktionen werden innerhalb der Pipeline benötigt
- Steuerkonflikt: Sprungbefehle in der Pipelinesequenz

sehr schlecht für Pipelining

- Unterbrechung des Programmkontexts: Interrupt, System-Call, Exception...
- (Performanz-) Optimierungen mit „Out-of-Order-Execution“ etc.

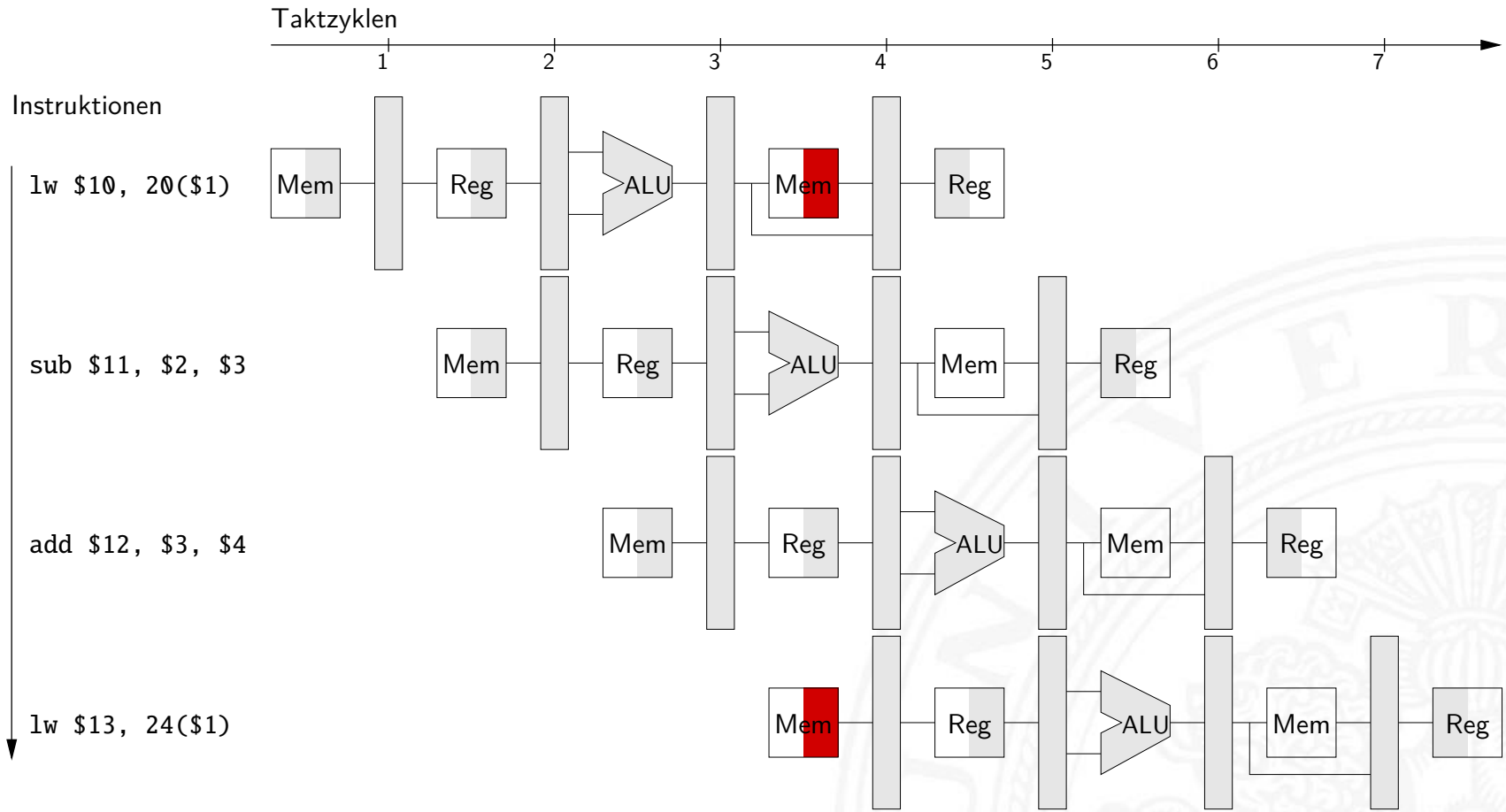
Pipeline Strukturkonflikte

Strukturkonflikt / Structural Hazard

- mehrere Stufen wollen gleichzeitig auf eine Ressource zugreifen
- Beispiel: gleichzeitiger Zugriff auf Speicher ▸ Beispiel

$\Rightarrow$ Mehrfachauslegung der betreffenden Ressourcen

- Harvard-Architektur vermeidet Strukturkonflikt aus Beispiel
- Multi-Port Register
- mehrfach vorhandene Busse und Multiplexer...



◂ Strukturkonflikte

Pipeline Datenkonflikte

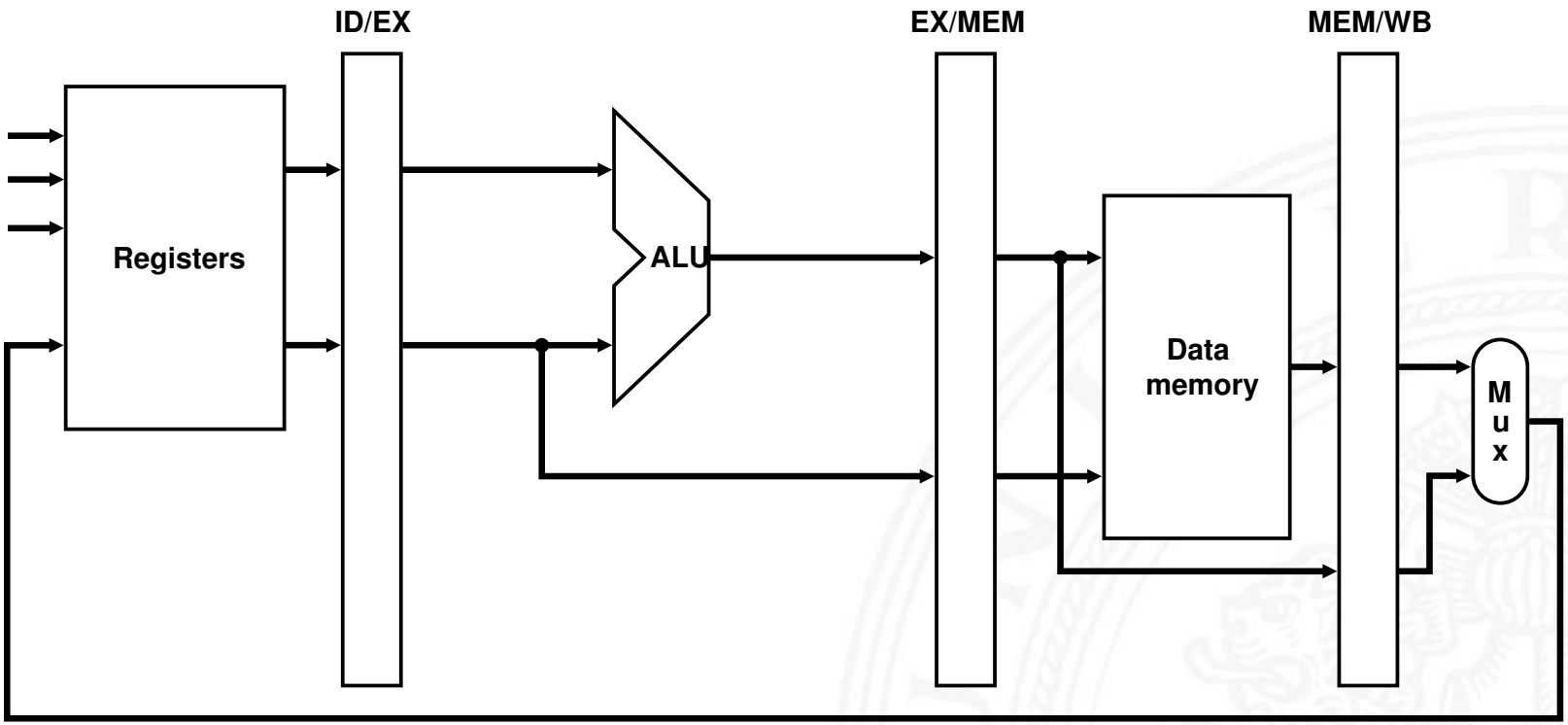
Datenkonflikt / Data Hazard

- eine Instruktion braucht die Ergebnisse einer vorhergehenden, diese wird aber noch in der Pipeline bearbeitet
- Datenabhängigkeiten der Stufe „Befehl ausführen“ ▸ Beispiel

Forwarding

- kann Datenabhängigkeiten auflösen, s. Beispiel
- extra Hardware: „*Forwarding-Unit*“
- Änderungen in der Pipeline Steuerung
- neue Datenpfade und Multiplexer

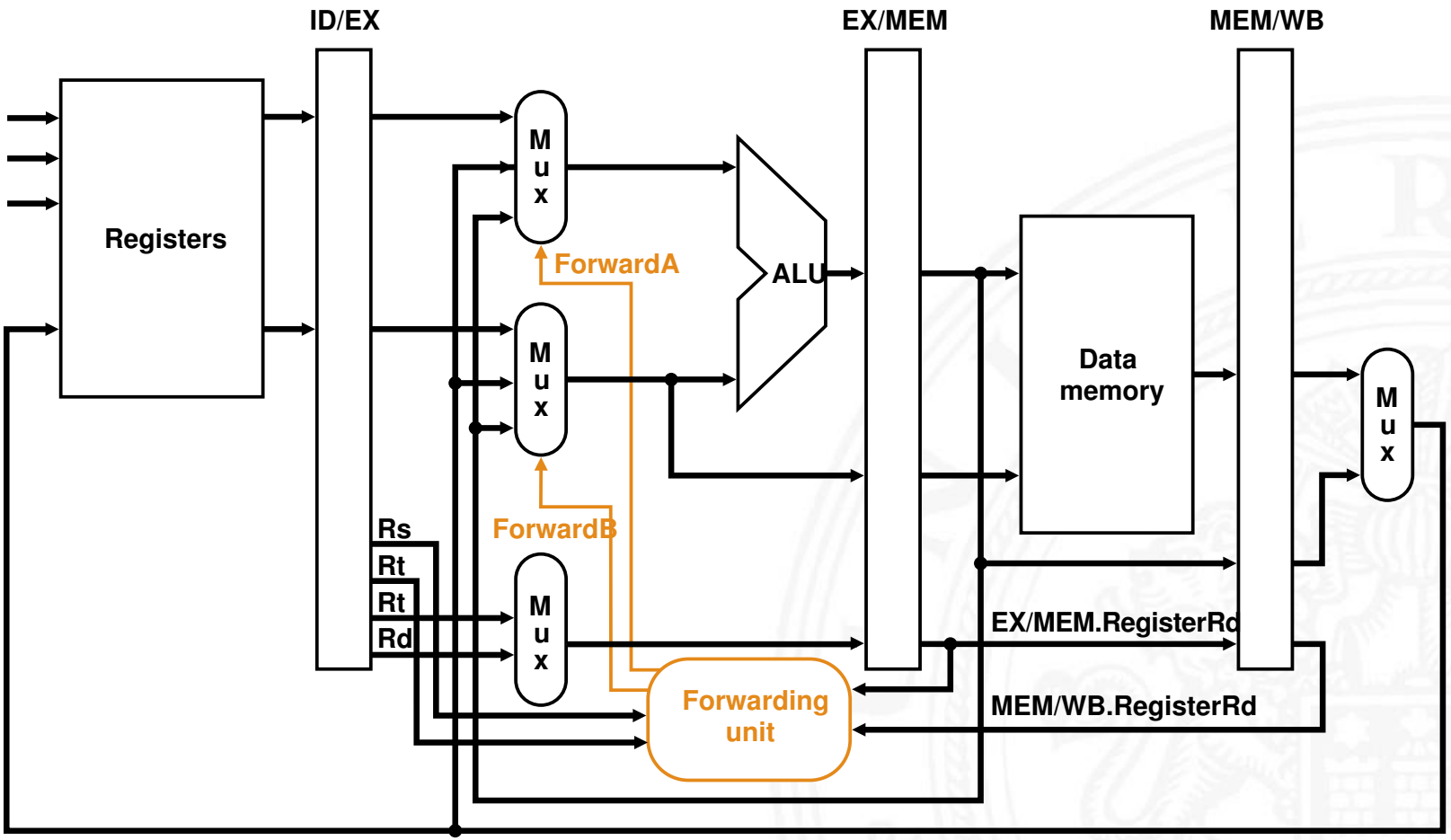
Pipeline Datenkonflikte (cont.)



MIPS Datenpfad

[PH12]

Pipeline Datenkonflikte (cont.)



MIPS Forwarding

[PH12]

Pipeline Datenkonflikte (cont.)

Rückwärtsabhängigkeiten

- spezielle Datenabhängigkeit ▸ Beispiel
- Forwarding-Technik funktioniert nicht, da die Daten erst *später* zur Verfügung stehen
  - bei längeren Pipelines
  - bei Load-Instruktionen (s.u.)

Auflösen von Rückwärtsabhängigkeiten

1. Softwarebasiert, durch den Compiler, Reihenfolge der Instruktionen verändern ▸ Beispiel
   - andere Operationen (ohne Datenabhängigkeiten) vorziehen
   - nop-Befehl(e) einfügen

Pipeline Datenkonflikte (cont.)

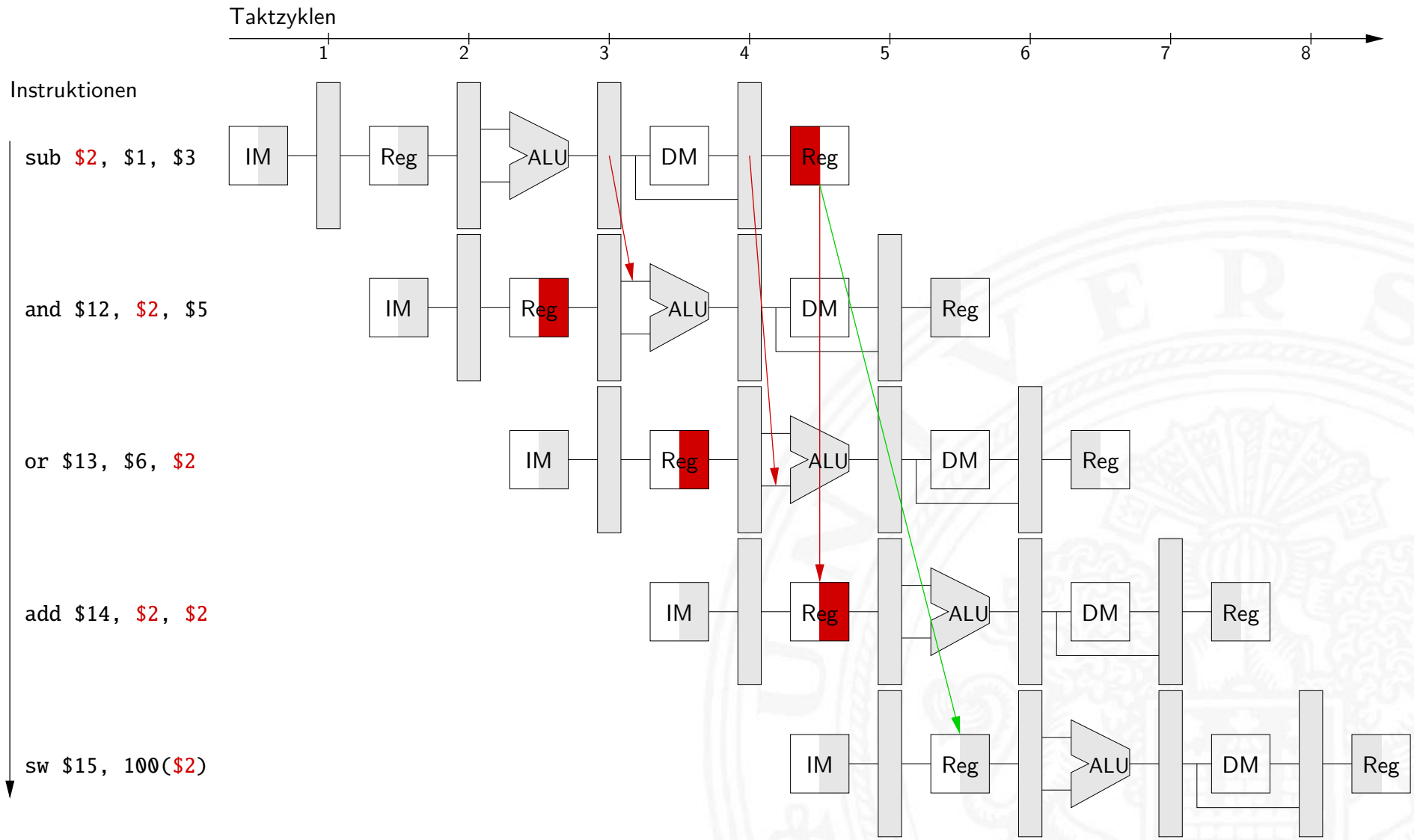
2. *„Interlocking“*

▸ Beispiel

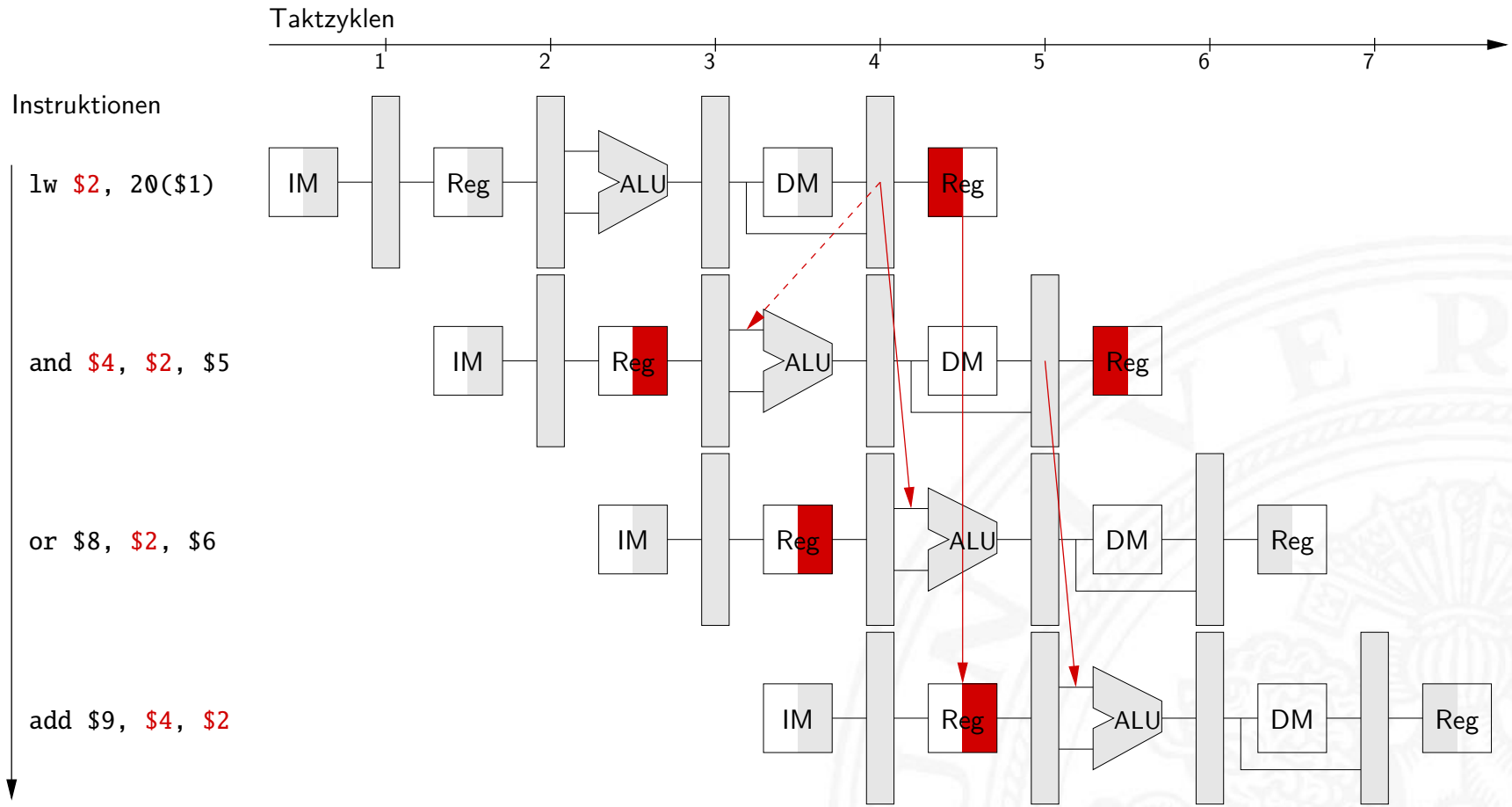
   - zusätzliche (Hardware) Kontrolleinheit
   - verschiedene Strategien
   - in Pipeline werden keine neuen Instruktionen geladen
   - Hardware erzeugt: Pipelineleerlauf / *„pipeline stall“*

„Scoreboard“

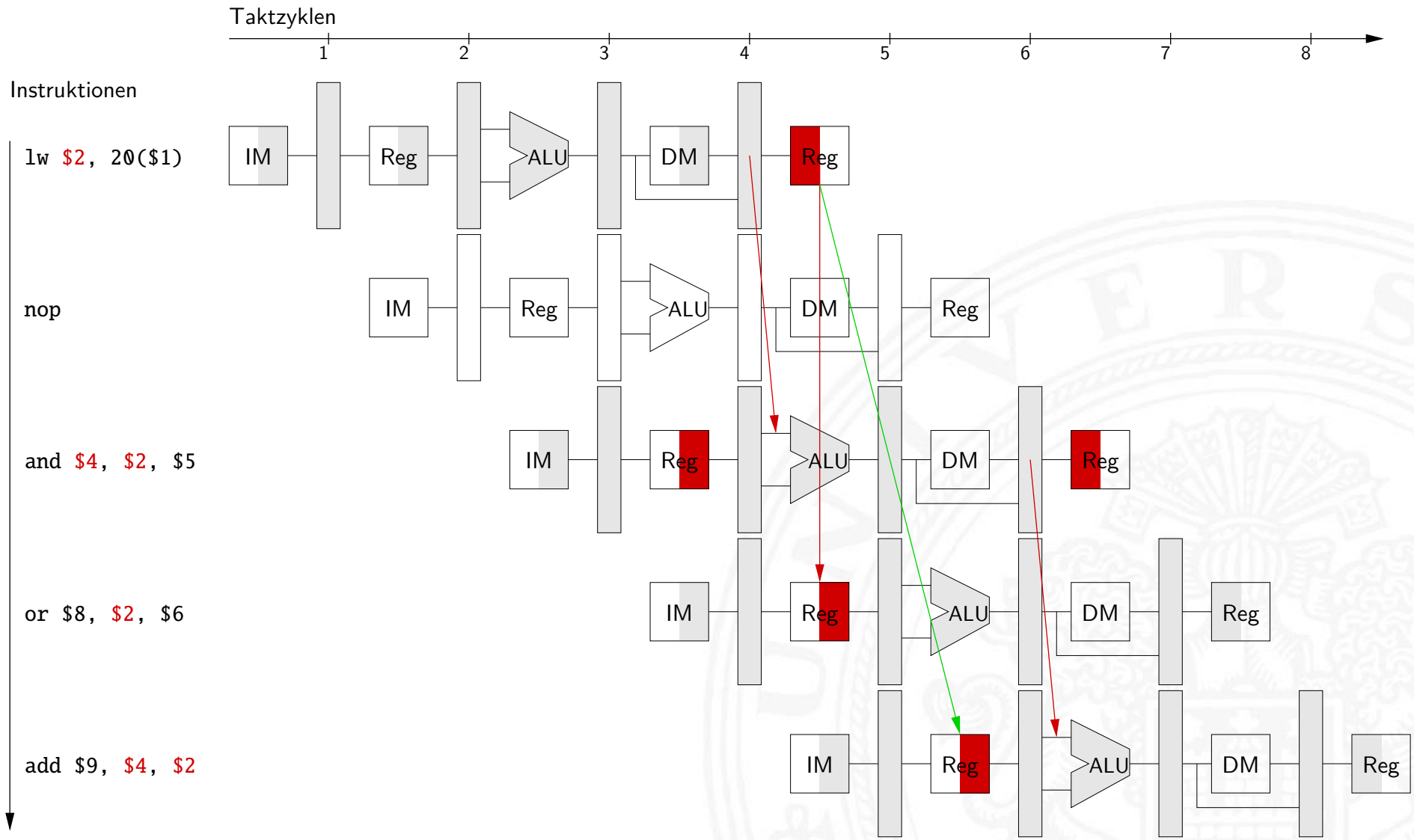
- Hardware Einheit zur zentralen Hazard-Erkennung und -Auflösung
- Verwaltet Instruktionen, benutzte Einheiten und Register der Pipeline



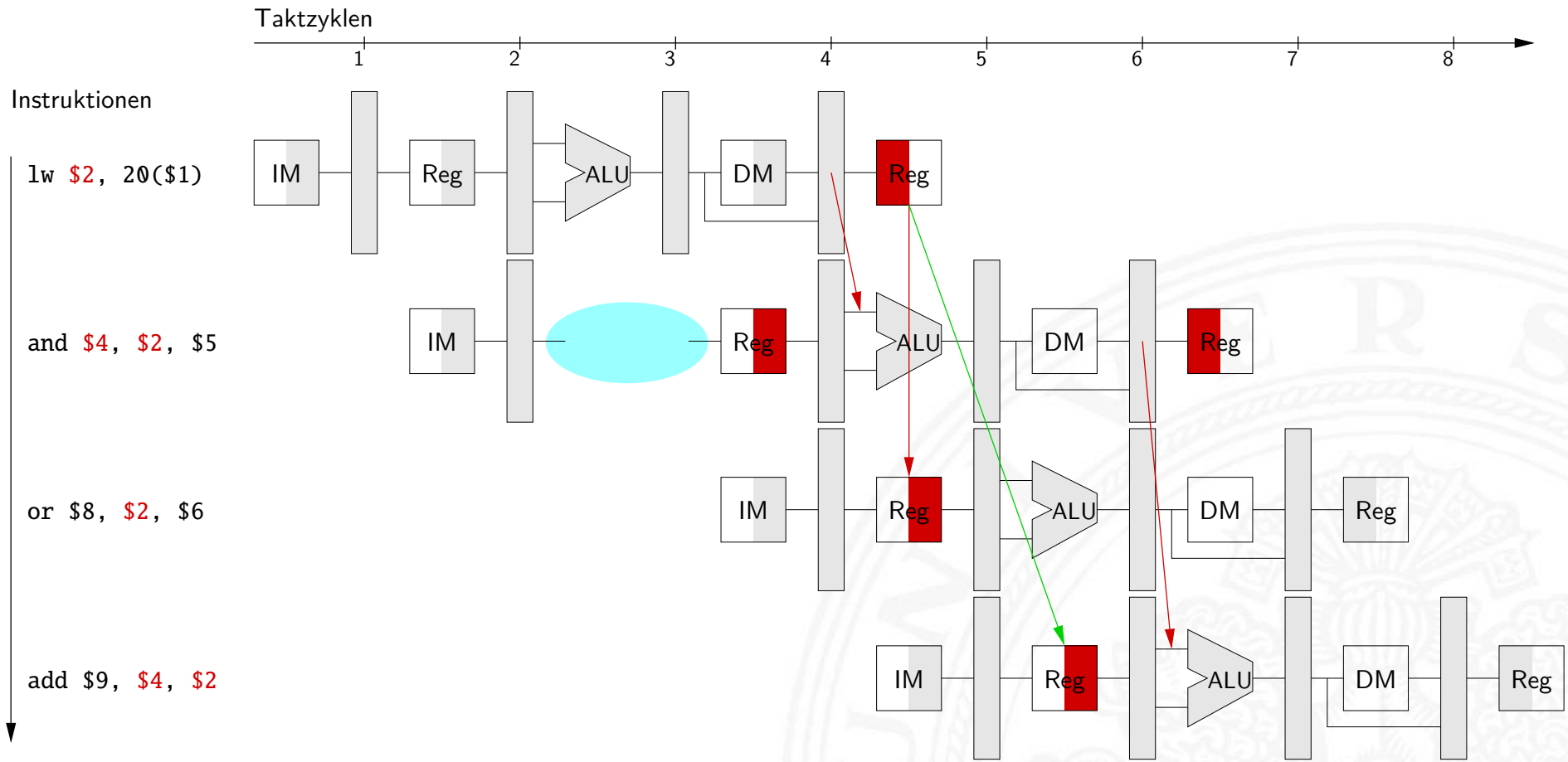
◂ Datenkonflikte



◂ Datenkonflikte



◂ Datenkonflikte



◂ Datenkonflikte

Pipeline Steuerkonflikte

Steuerkonflikt / Control Hazard

- Sprungbefehle unterbrechen den sequenziellen Ablauf der Instruktionen
- Problem: Instruktionen die auf (bedingte) Sprünge folgen, werden in die Pipeline geschoben, bevor bekannt ist, ob verzweigt werden soll
- Beispiel: bedingter Sprung

▸ Beispiel

Pipeline Steuerkonflikte (cont.)

Lösungsmöglichkeiten für Steuerkonflikte

- ad-hoc Lösung: „Interlocking“ erzeugt Pipelineleerlauf
  - – ineffizient: ca. 19 % der Befehle sind Sprünge

1. Annahme: nicht ausgeführter Sprung / „untaken branch“
   - \+ kaum zusätzliche Hardware
   - – im Fehlerfall
     - Pipelineleerlauf
     - Pipeline muss geleert werden / „flush instructions“
2. Sprungentscheidung „vorverlegen“
   - Software: Compiler zieht andere Instruktionen vor Verzögerung nach Sprungbefehl / „delay slots“
   - Hardware: Sprungentscheidung durch Zusatz-ALU (nur Vergleiche) während Befehlsdecodierung (z.B. MIPS)

Pipeline Steuerkonflikte (cont.)

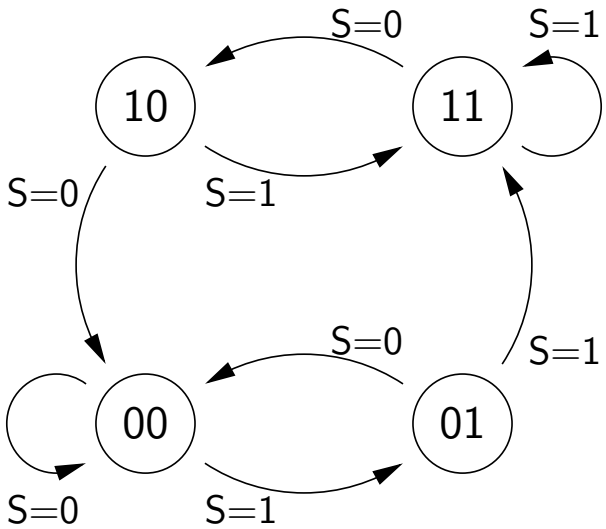
3. Sprungvorhersage / *„branch prediction“*
   - Beobachtung: ein Fall tritt häufiger auf:
     Schleifendurchlauf, Datenstrukturen durchsuchen etc.
   - mehrere Vorhersageverfahren; oft miteinander kombiniert
   + hohe Trefferquote: bis 90 %

   Statische Sprungvorhersage (softwarebasiert)
   - Compiler erzeugt extra Bit in Opcode des Sprungbefehls
   - Methoden: Codeanalyse, Profiling...

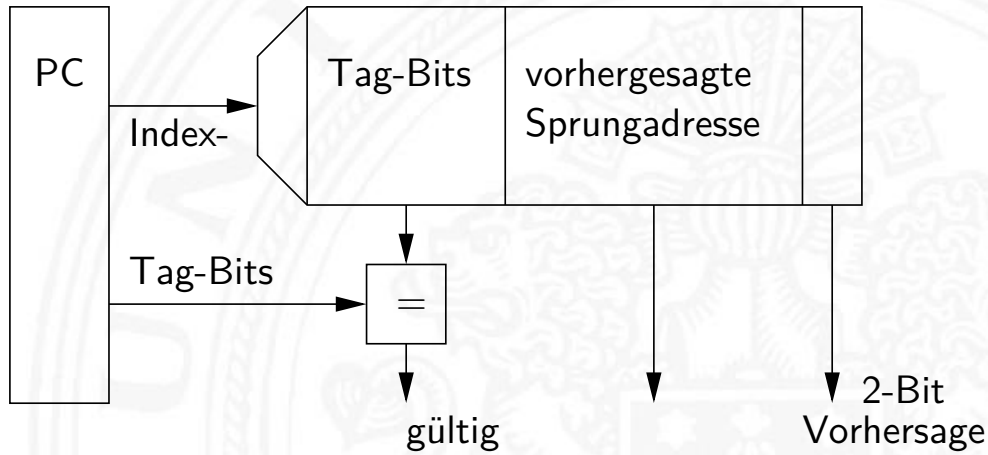
   Dynamische Sprungvorhersage (hardwarebasiert)
   - Sprünge durch Laufzeitinformation vorhersagen:
     Wie oft wurde der Sprung in letzter Zeit ausgeführt?
   - viele verschiedene Verfahren:
     History-Bit, 2-Bit Prädiktor, korrelationsbasierte Vorhersage,
     Branch History Table, Branch Target Cache...

Pipeline Steuerkonflikte (cont.)

- Beispiel: 2-Bit Sprungvorhersage + Branch Target Cache

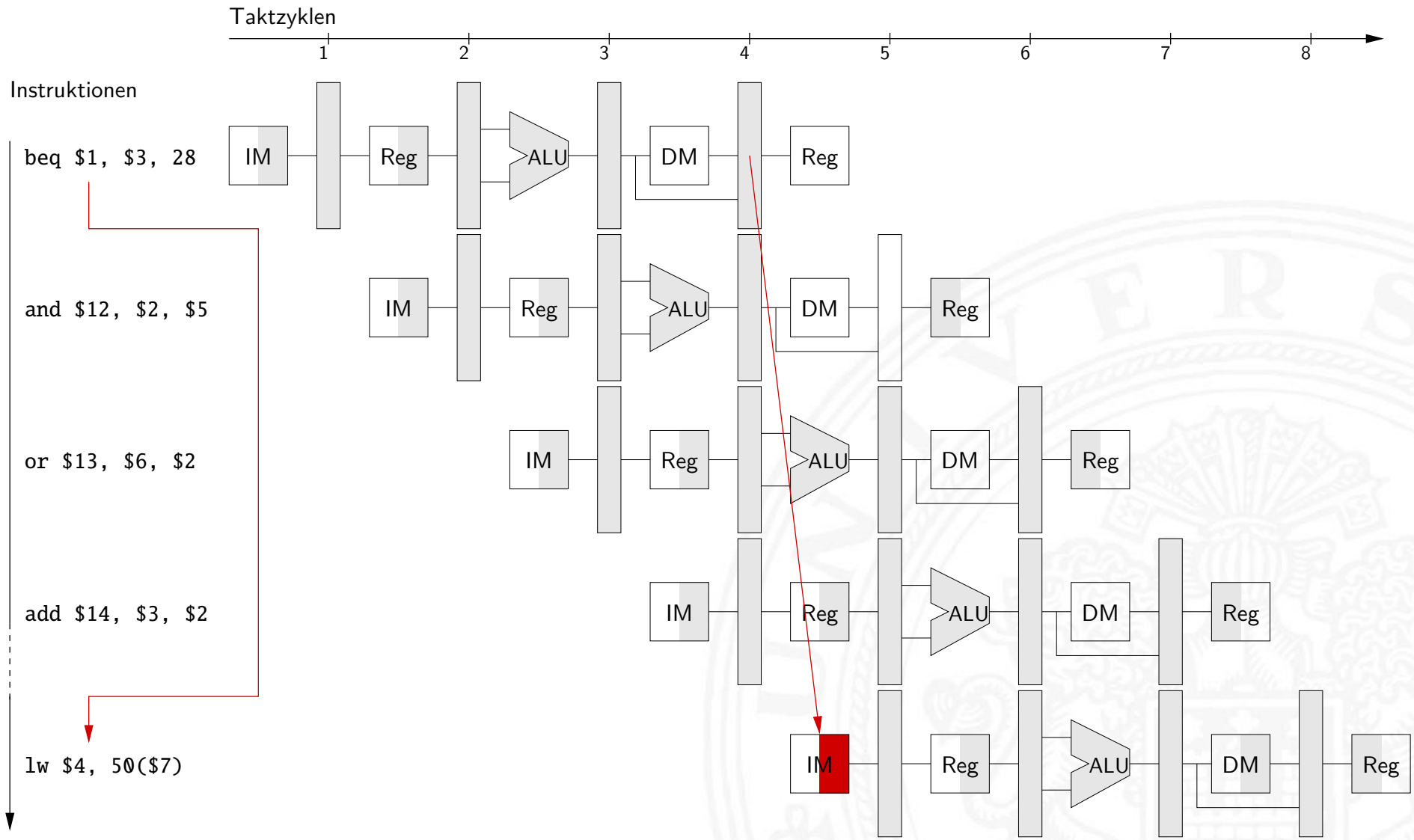


VH **V**orhersage bit
Historie bit
S=0/1 **S**prung ausgeführt



Pipeline Steuerkonflikte (cont.)

- Schleifen abrollen / „*Loop unrolling*“
  - zusätzliche Maßnahme zu allen zuvor skizzierten Verfahren
  - bei statische Schleifenbedingung möglich
  - Compiler iteriert Instruktionen in der Schleife (teilweise)
  - – längerer Code
  - \+ Sprünge und Abfragen entfallen
  - \+ erzeugt sehr lange Codesequenzen ohne Sprünge
    $\Rightarrow$ Pipeline kann optimal ausgenutzt werden



◂ Steuerkonflikte

Superskalare Prozessoren

- Superskalare CPUs besitzen mehrere Recheneinheiten: 4. . . 10
- In jedem Takt werden (dynamisch) mehrere Instruktionen eines konventionell linearen Instruktionsstroms abgearbeitet: $CPI < 1$
  ILP (**I**nstruction **L**evel **P**arallelism) ausnutzen!
- Hardware verteilt initiierte Instruktionen auf Recheneinheiten
- Pro Takt kann *mehr als eine* Instruktion initiiert werden
  Die Anzahl wird dynamisch von der Hardware bestimmt:
  0. . . „*Instruction Issue Bandwidth*“

\+ sehr effizient, alle modernen CPUs sind superskalar

– Abhängigkeiten zwischen Instruktionen sind der Engpass, das Problem der Hazards wird verschärft

Superskalar – Datenabhängigkeiten

Datenabhängigkeiten

- RAW – **R**ead **A**fter **W**rite
  Instruktion I_x darf Datum erst lesen, wenn I_{x-n} geschrieben hat
- WAR – **W**rite **A**fter **R**ead
  Instruktion I_x darf Datum erst schreiben, wenn I_{x-n} gelesen hat
- WAW – **W**rite **A**fter **W**rite
  Instruktion I_x darf Datum erst überschreiben, wenn I_{x-n} geschrieben hat

Superskalar – Datenabhängigkeiten (cont.)

Datenabhängigkeiten superskalarer Prozessoren

- RAW: echte Abhängigkeit; Forwarding ist kaum möglich und in superskalaren Pipelines extrem aufwendig
- WAR, WAW: „*Register Renaming*“ als Lösung

„*Register Renaming*“

- Hardware löst Datenabhängigkeiten innerhalb der Pipeline auf
- Zwei Registersätze sind vorhanden
  1. Architektur-Register: „logische Register“ der ISA
  2. viele Hardware-Register: „Rename Register“
     - dynamische Abbildung von ISA- auf Hardware-Register

Superskalar – Datenabhängigkeiten (cont.)

- Beispiel

Originalcode

```
tmp  = a + b;
res1 = c + tmp;
tmp  = d + e;
res2 = tmp - f;
```

nach Renaming

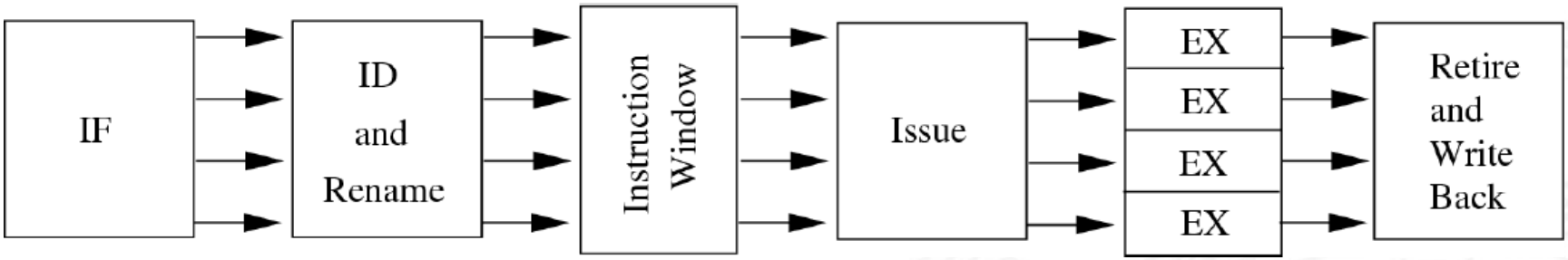
```
tmp1 = a + b;
res1 = c + tmp1;
tmp2 = d + e;
res2 = tmp2 - f;
tmp  = tmp2;
```

Parallelisierung des modifizierten Codes

```
tmp1 = a + b;      tmp2 = d + e;
res1 = c + tmp1;   res2 = tmp2 - f;   tmp = tmp2;
```

Superskalar – Pipeline

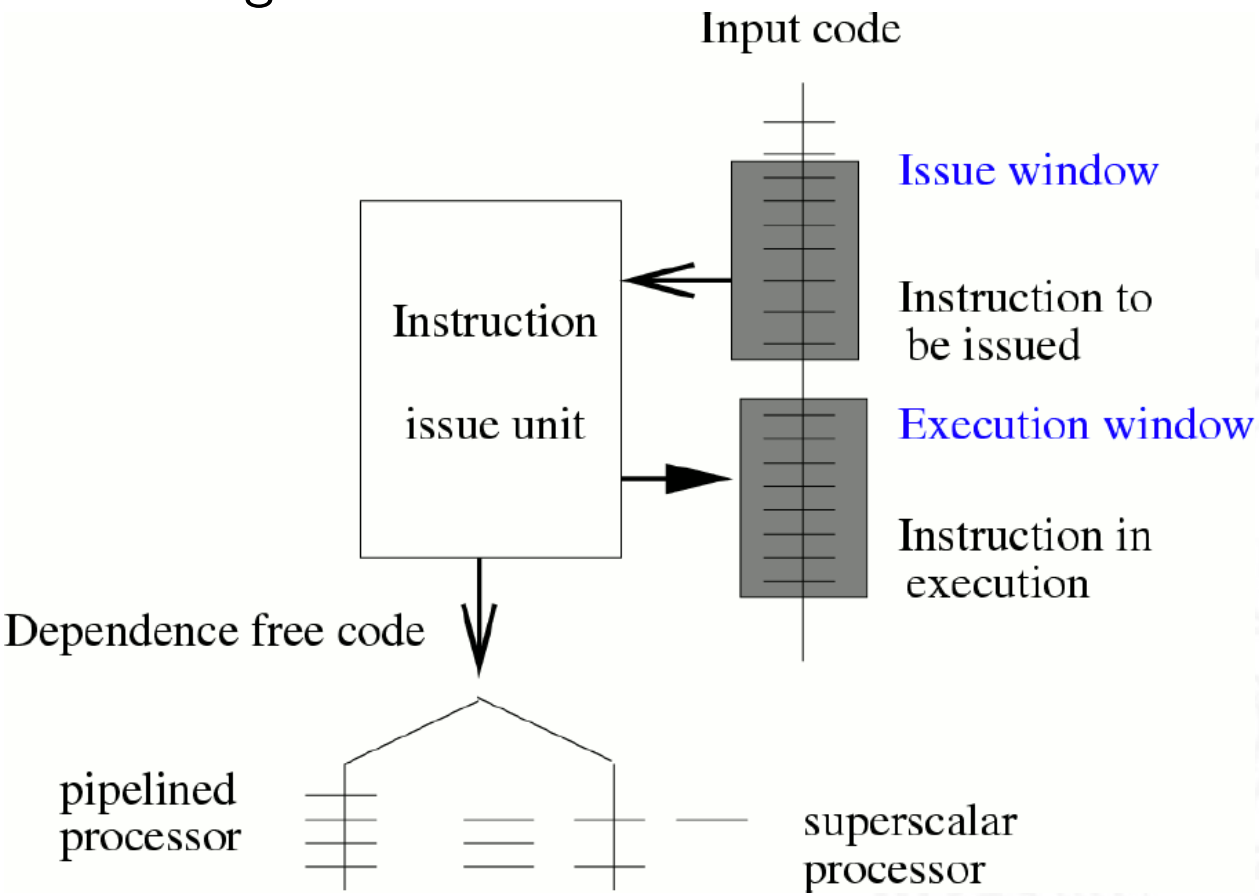
Aufbau der superskalaren Pipeline



- lange Pipelines mit vielen Phasen: Fetch (Prefetch, Predecode), Decode / Register-Renaming, Issue, Dispatch, Execute, Retire (Commit, Complete / Reorder), Write-Back
- je nach Implementation unterschiedlich aufgeteilt
- entscheidend für superskalare Architektur sind die Schritte
  vor den ALUs: Issue, Dispatch $\Rightarrow$ *out-of-order* Ausführung
  nach -"- : Retire $\Rightarrow$ *in-order* Ergebnisse

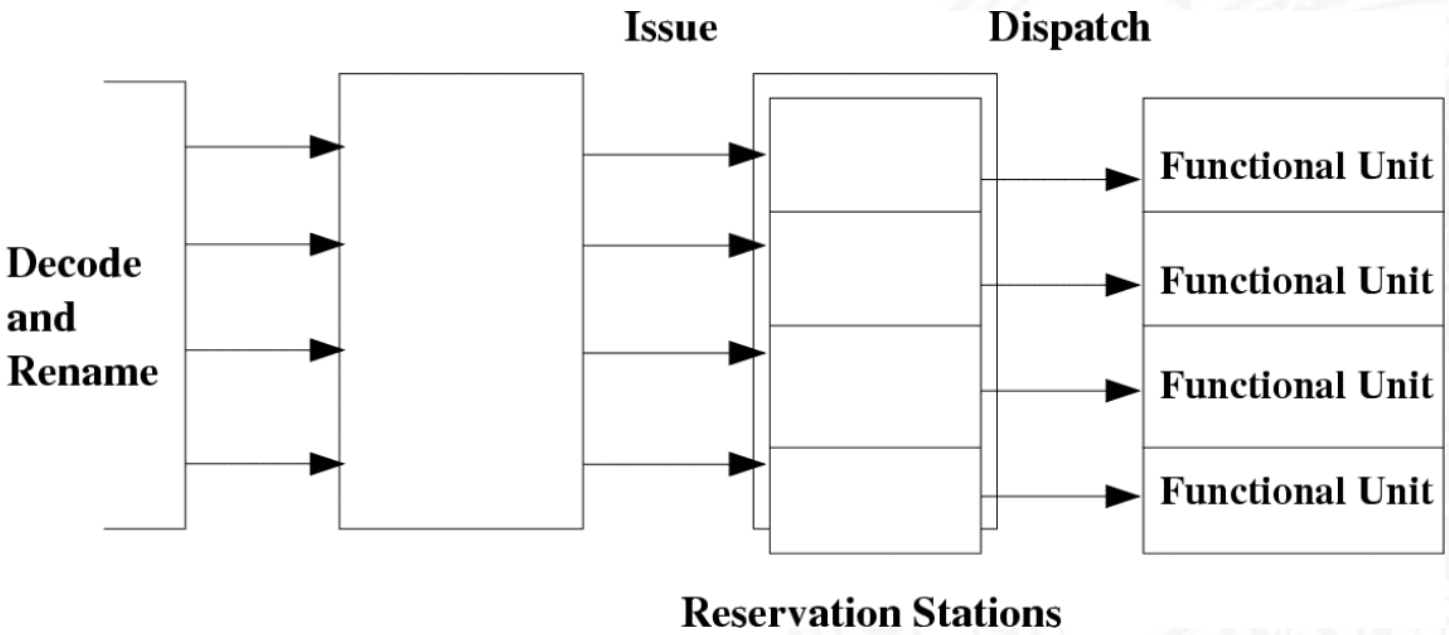
Superskalar – Pipeline (cont.)

Scheduling der Instruktionen



Superskalar – Pipeline (cont.)

- Dynamisches Scheduling erzeugt *out-of-order* Reihenfolge der Instruktionen
- Issue: globale Sicht
  Dispatch: getrennte Ausschnitte in „Reservation Stations“



Superskalar – Pipeline (cont.)

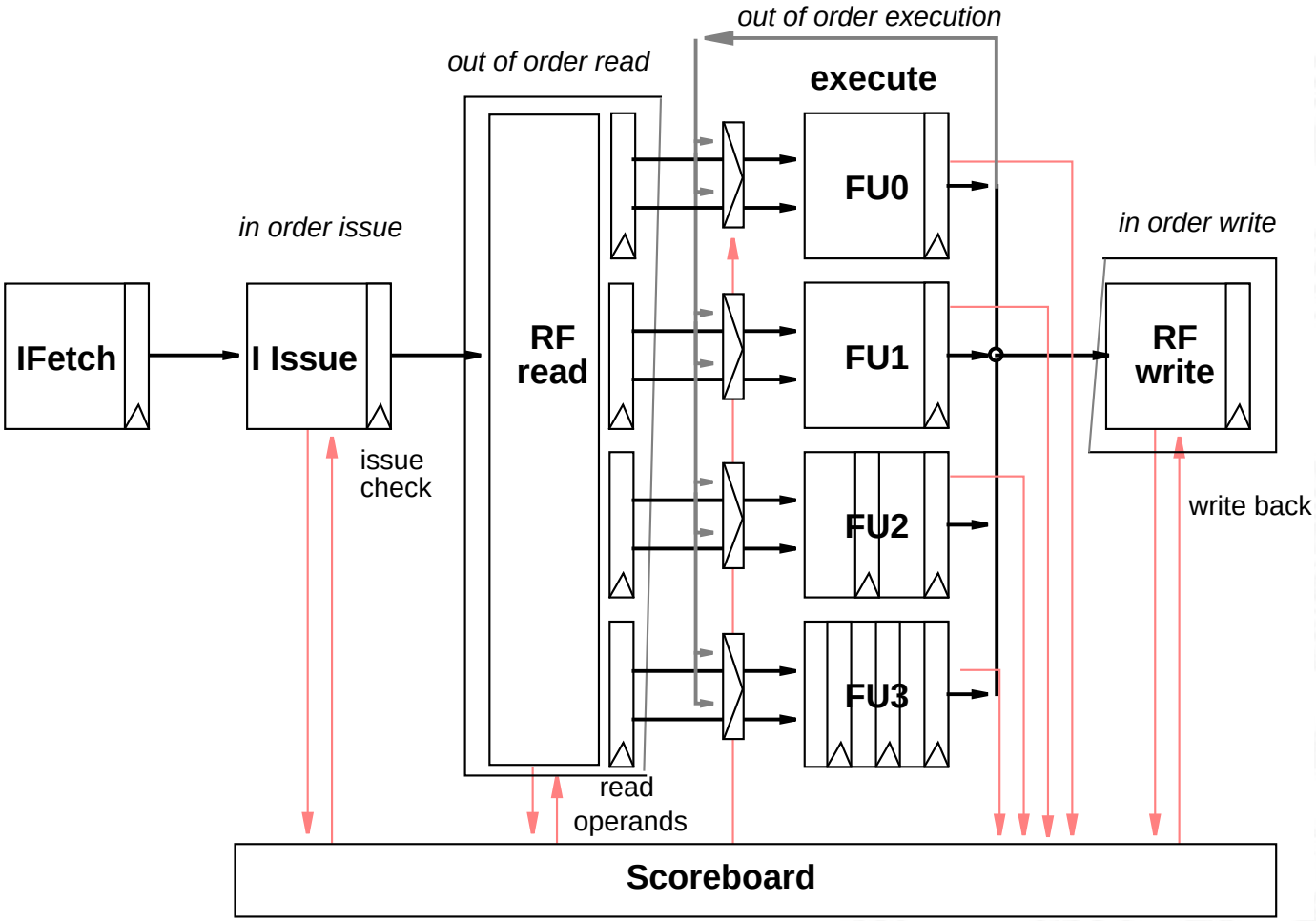
Reservation Station für jede Funktionseinheit

- speichert: initiierte Instruktionen die auf Recheneinheit warten
- –"– zugehörige Operanden
- –"– ggf. Zusatzinformation
- Instruktion bleibt blockiert, bis alle Parameter bekannt sind und wird dann an die zugehörige ALU weitergeleitet

- Dynamisches Scheduling: zuerst '67 in IBM 360 (Robert Tomasulo)
  - Forwarding
  - Registerumbenennung und Reservation Stations

Superskalar – Pipeline (cont.)

Zentrale Verwaltungseinheit: „Scoreboard“



Superskalar – Pipeline (cont.)

Scoreboard erlaubt das Management mehrerer Ausführungseinheiten

- out-of-order Ausführung von Mehrzyklusbefehlen
- Auflösung aller Struktur- und Datenkonflikte: RAW, WAW, WAR

Einschränkungen

- single issue (nicht superskalar)
- in-order issue
- keine Umbenennungen; also Leerzyklen bei WAR- und WAW-Konflikten
- kein Forwarding, daher Zeitverlust bei RAW-Konflikten

Superskalar – Pipeline (cont.)

Retire-Stufe

- erzeugt wieder *in-order* Reihenfolge
- FIFO: Reorder-Buffer
- commit: „richtig ausgeführte“ Instruktionen gültig machen
- abort: Sprungvorhersage falsch
  Instruktionen verwerfen

Superskalar – Pipeline (cont.)

Spezielle Probleme superskalarer Pipelines

- weitere Hazard-Möglichkeiten
  - die verschiedenen ALUs haben unterschiedliche Latenzzeiten
  - Befehle „warten“ in den Reservation Stations

  $\Rightarrow$ Datenabhängigkeiten können sich mit jedem Takt ändern
- Kontrollflussabhängigkeiten: Anzahl der Instruktionen zwischen bedingten Sprüngen limitiert Anzahl parallelisierbarer Instruktion

$\Rightarrow$ „Loop Unrolling“ wichtig

$+$ optimiertes (dynamisches) Scheduling: Faktor 3 möglich

Superskalar – Pipeline (cont.)

Software Pipelining

- Codeoptimierungen beim Compilieren: Ersatz für, bzw. Ergänzend zu der Pipelineunterstützung durch Hardware
- Compiler hat „globalen“ Überblick
  $\Rightarrow$ zusätzliche Optimierungsmöglichkeiten
- symbolisches Loop Unrolling
- Loop Fusion
- ...

Superskalar – Interrupts

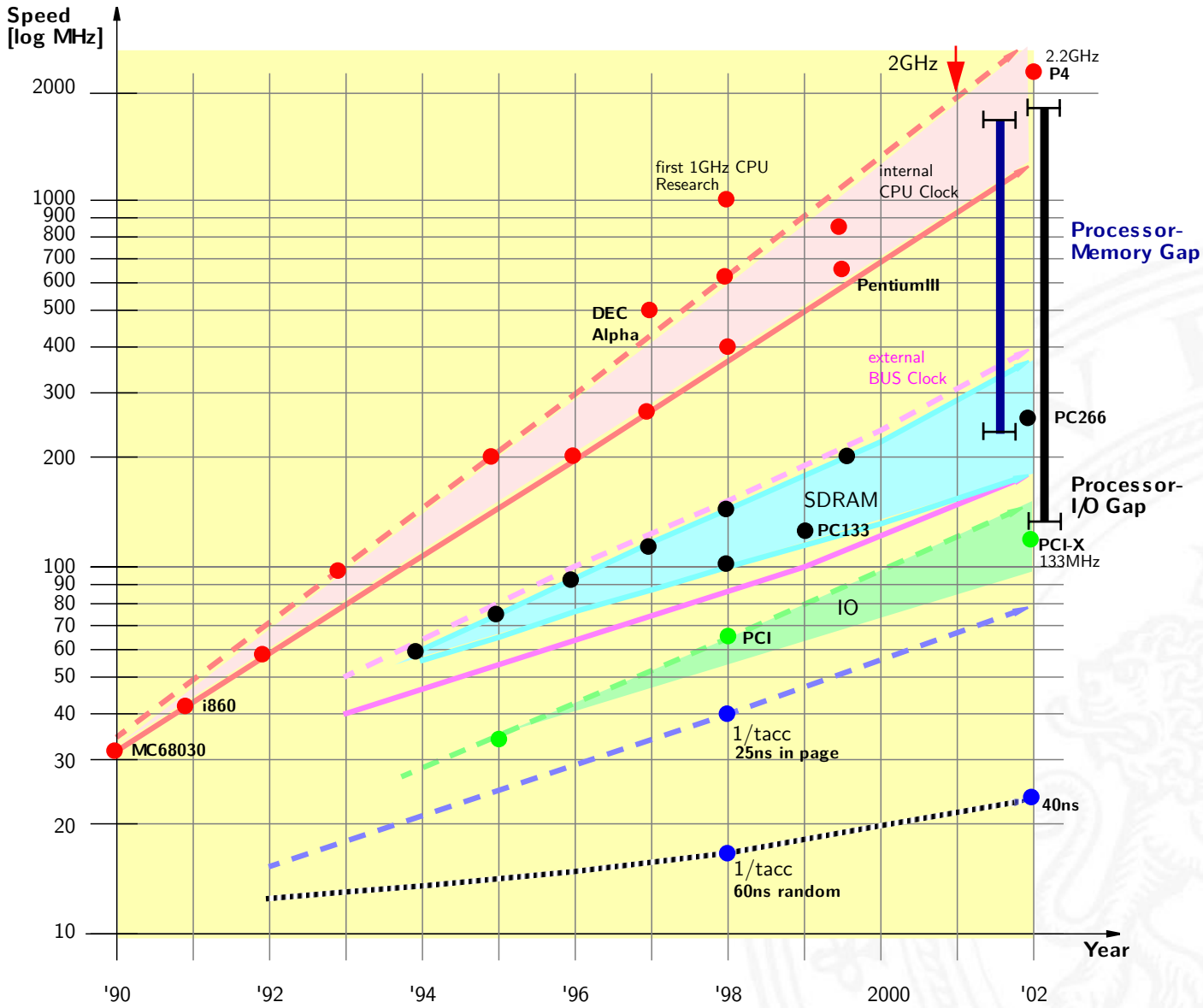
Exceptions, Interrupts und System-Calls

- Interruptbehandlung ist wegen der Vielzahl paralleler Aktionen und den Abhängigkeiten innerhalb der Pipelines extrem aufwendig
  - da unter Umständen noch Pipelineaktionen beendet werden müssen, wird *zusätzliche Zeit* bis zur Interruptbehandlung benötigt
  - wegen des Register-Renaming muss sehr viel *mehr Information* gerettet werden als nur die ISA-Register
- Prinzip der Interruptbehandlung
  - keine neuen Instruktionen mehr initiieren
  - warten bis Instruktionen des Reorder-Buffers abgeschlossen sind

Superskalar – Interrupts (cont.)

- Verfahren ist von der „Art“ des Interrupt abhängig
  - Precise-Interrupt: Pipelineaktivitäten komplett Beenden
  - Imprecise-Interrupt: wird als verzögerter Sprung (Delayed-Branching) in Pipeline eingebracht
    Zusätzliche Register speichern Information über Instruktionen die in der Pipeline nicht abgearbeitet werden können (z.B. weil sie den Interrupt ausgelöst haben)
- Definition: Precise-Interrupt
  - Programmzähler (PC) zur Interrupt auslösenden Instruktion ist bekannt
  - Alle Instruktionen bis zur PC-Instruktion wurden vollständig ausgeführt
  - Keine Instruktion nach der PC-Instruktion wurde ausgeführt
  - Ausführungszustand der PC-Instruktion ist bekannt

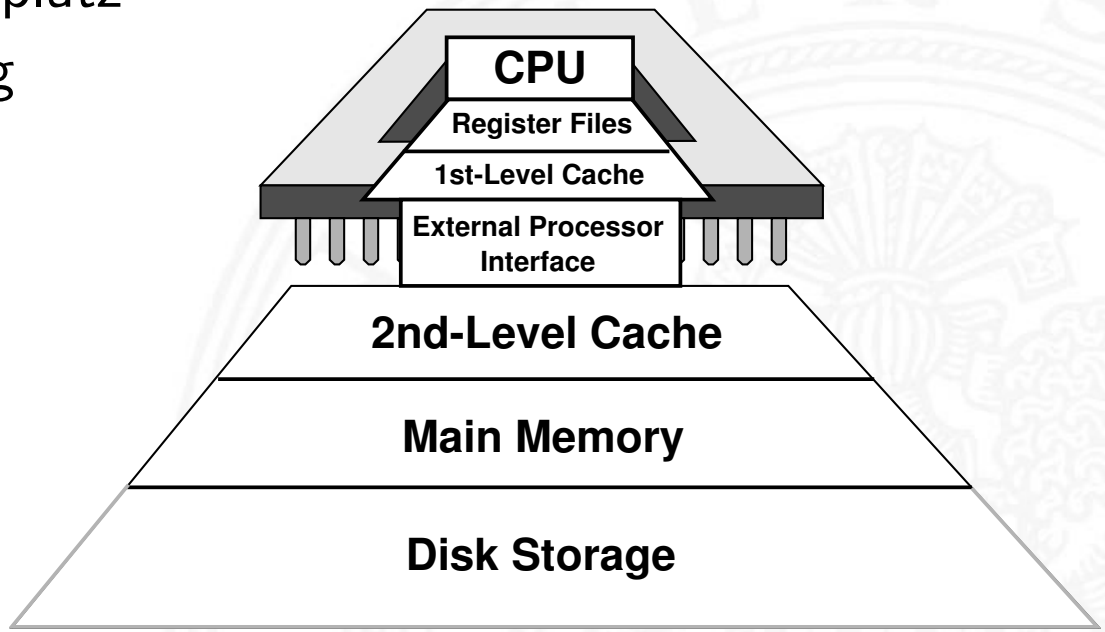
Gliederung


Speed
[log MHz]
2000
1000
900
800
700
600
500
400
300
200
100
90
80
70
60
50
40
30
20
10
2GHz
2.2GHz
P4
first 1GHz CPU
Research
internal
CPU Clock
PentiumIII
DEC
Alpha
external
BUS Clock
PC266
Processor-
Memory Gap
SDRAM
PC133
Processor-
I/O Gap
PCI-X
133MHz
IO
PCI
i860
MC68030
1/tacc
25ns in page
40ns
1/tacc
60ns random
Year
'90
'92
'94
'96
'98
2000
'02

Speicherhierarchie

Motivation

- Geschwindigkeit der Prozessoren
- Kosten für den Speicherplatz
- permanente Speicherung
  - magnetisch
  - optisch
  - mechanisch

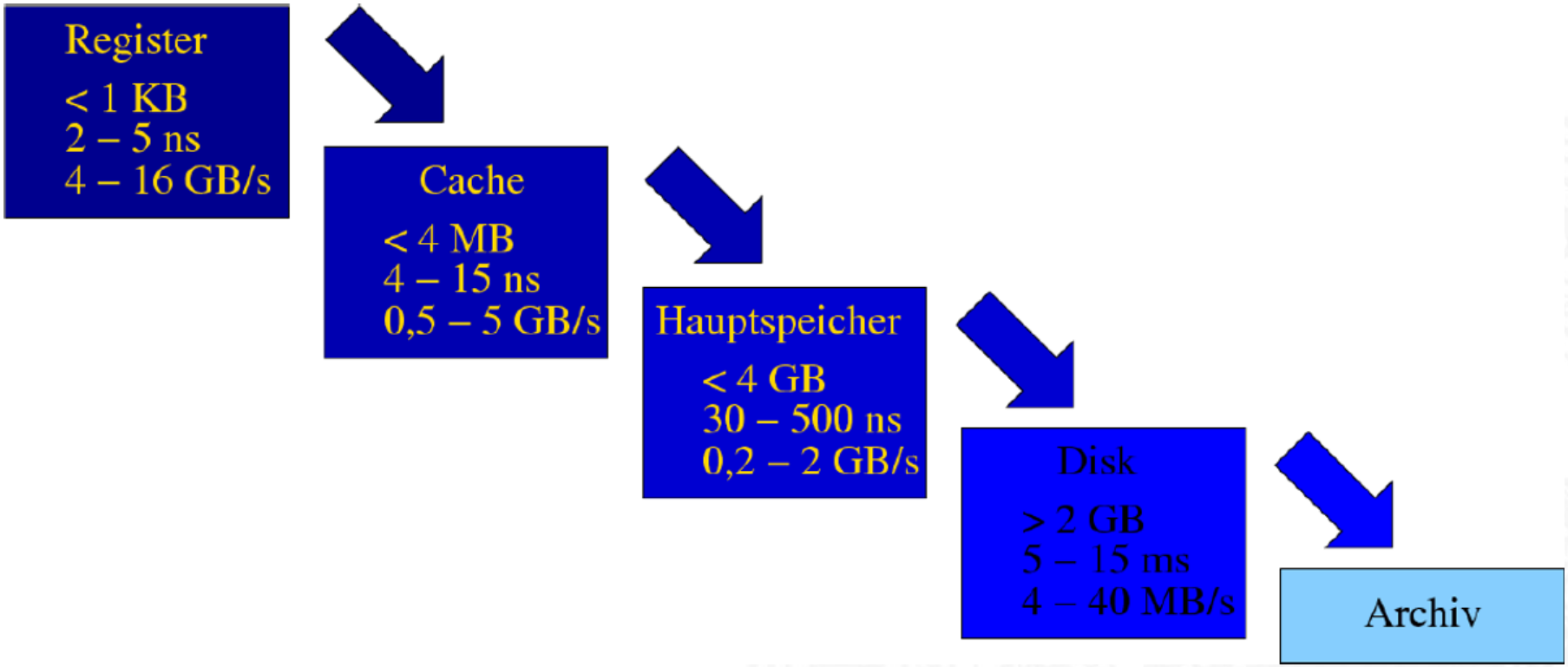


Speicherhierarchie (cont.)

Speichertypen

Speicher	Vorteile	Nachteile
Register	sehr schnell	sehr teuer
SRAM	schnell	teuer, große Chips
DRAM	hohe Integration	Refresh nötig, langsam
Platten	billig, Kapazität	sehr langsam, mechanisch
Beispiel	Hauptspeicher	Festplatte
Latenz	8 ns	4 ms
Bandbreite	$\approx$ 38,4 GByte/sec (triple Channel)	$\approx$ 750 MByte/sec (typisch $<$ 300)
Kosten	1 GByte, 5 €	1 TByte, 40 € (4 Ct./GByte)

Speicherhierarchie (cont.)



Speicherhierarchie (cont.)

Verwaltung der Speicherhierarchie

- Register $\leftrightarrow$ Memory
  - Compiler
  - Assembler-Programmierer
- Cache $\leftrightarrow$ Memory
  - Hardware
- Memory $\leftrightarrow$ Disk
  - Hardware und Betriebssystem (Paging)
  - Programmierer (Files)

Register

- Speicherung von Werten in der CPU
- Aufgaben
  - „*General-purpose*“ Register: für den (Assembler-) Programmierer sichtbare Register für Operanden und Speicheradressen
  - „*Flag*“ Register: Statusanzeigen im Datenprozessor (Status nach ALU Operationen) und Befehlsprozessor (Programm-, Interruptstatus. . .)
  - Befehlszähler, spezielle Adressierungsregister im Befehlsprozessor
  - Pipelineregister
  - . . .

Register (cont.)

- Zugriff mit *voller* Prozessorgeschwindigkeit
  . . . also möglichst viele Register
  - Kontextwechsel sind zeitaufwendig
  - mehr Bits für Adressierung

⇒ virtuelle Register

- beschränkte Anzahl von Registern (pro Kontext)
- Umschalten zwischen Registersätzen

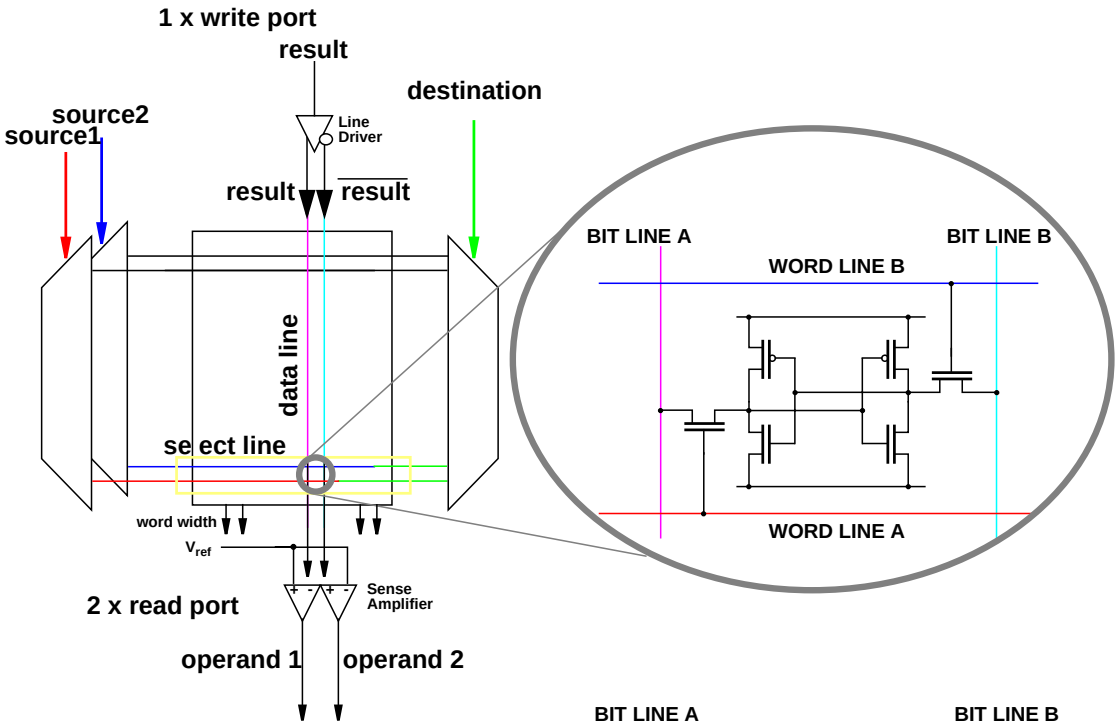
⇒ Registerfenster

- virtuelle Register dürfen sich überlappen
- effiziente Unterprogrammaufrufe: Parameterübergabe über solche Bereiche (sonst über Speicher)

Registerbänke

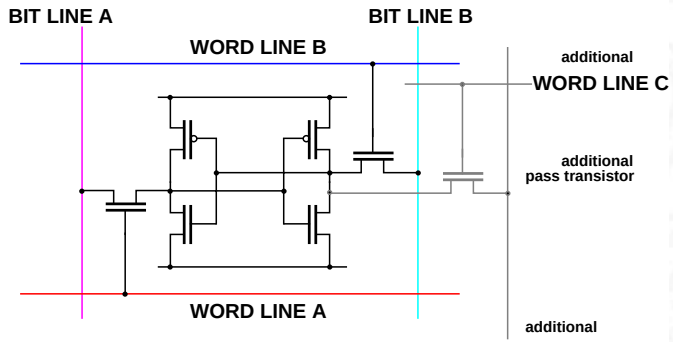
- (große) Arrays von einzeln adressierbaren Registern
- meist mit mehreren Ein- und Ausgabeports
- Realisierung
  - Register aus Flipflops und Multiplexerlogik
  - SRAM Realisierung große „*Register Files*“

Registerbänke (cont.)



Register-File

figure b



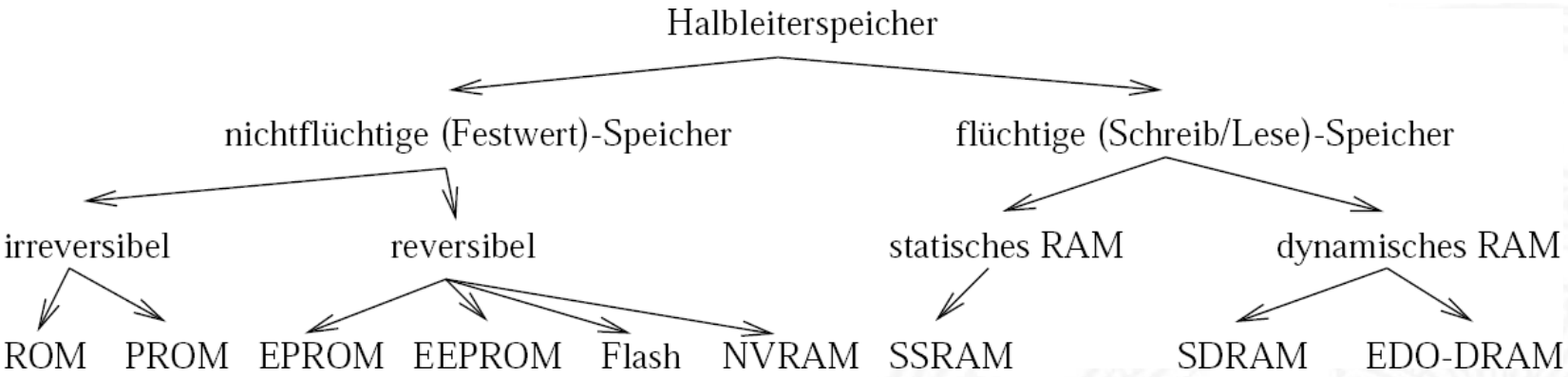
Hauptspeicher

Speichertypen

	SRAM	DRAM
Zugriffszeit	5...50 ns	60...100 ns t_{rac}
		20...300 ns t_{cac}
		110...180 ns t_{cyc}
Leistungsaufnahme	200...1300 mW	300...600 mW
Speicherkapazität	$<$ 72 Mbit	$<$ 4 Gbit
Preis	10 €/Mbit	0,2 Ct./Mbit
	$\Rightarrow$ Cache	$\Rightarrow$ Hauptspeicher

Hauptspeicher (cont.)

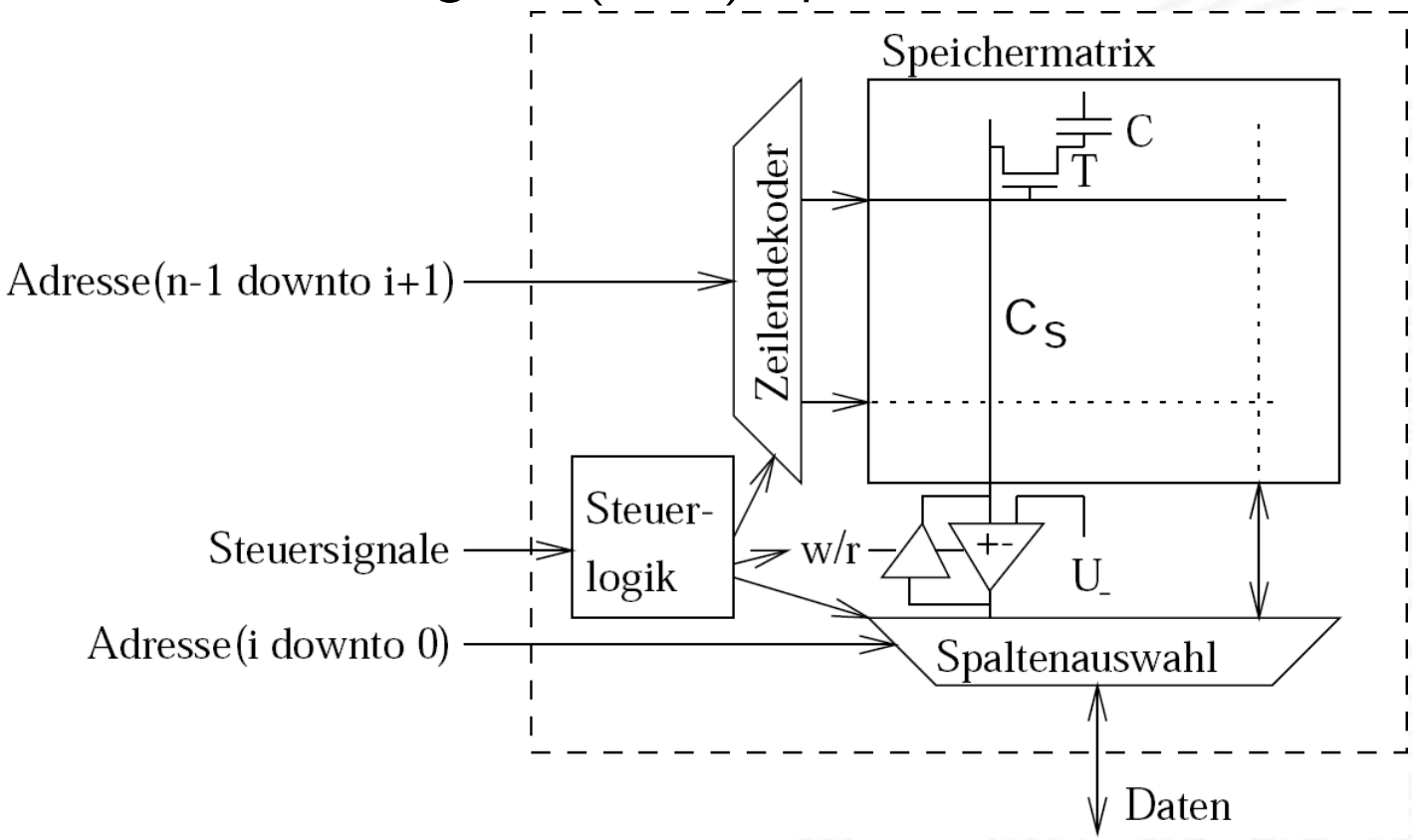
Arten von Halbleiterspeicher



Hauptspeicher: DRAM

Dynamische RAMs

- Matrix-Anordnung der (1-bit) Speicherzellen

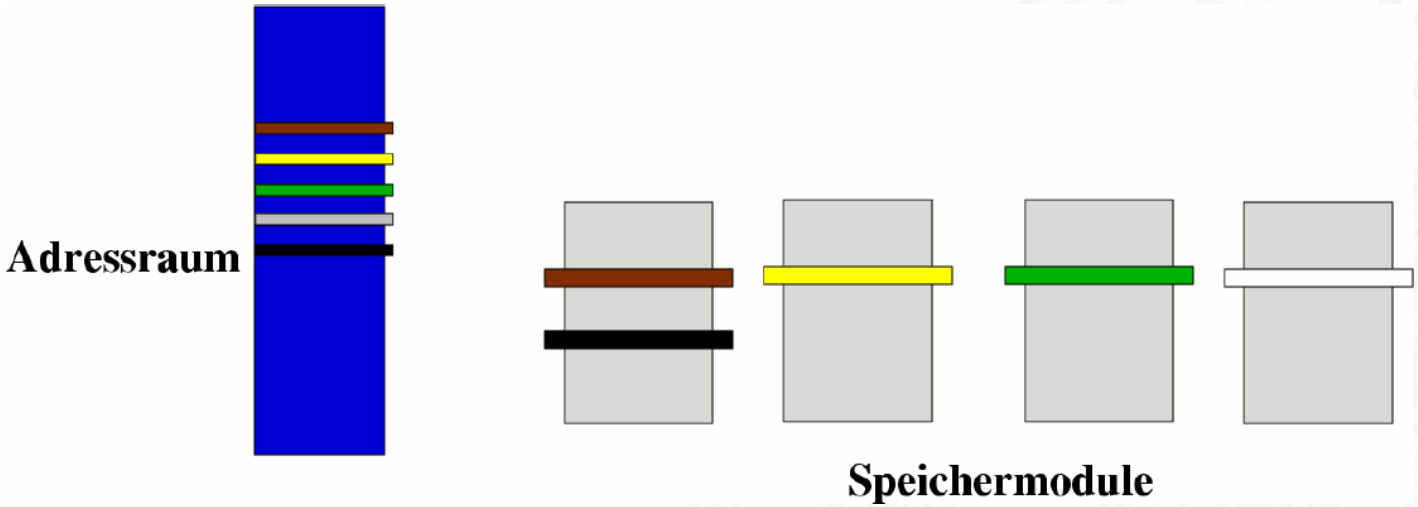


Hauptspeicher: DRAM (cont.)

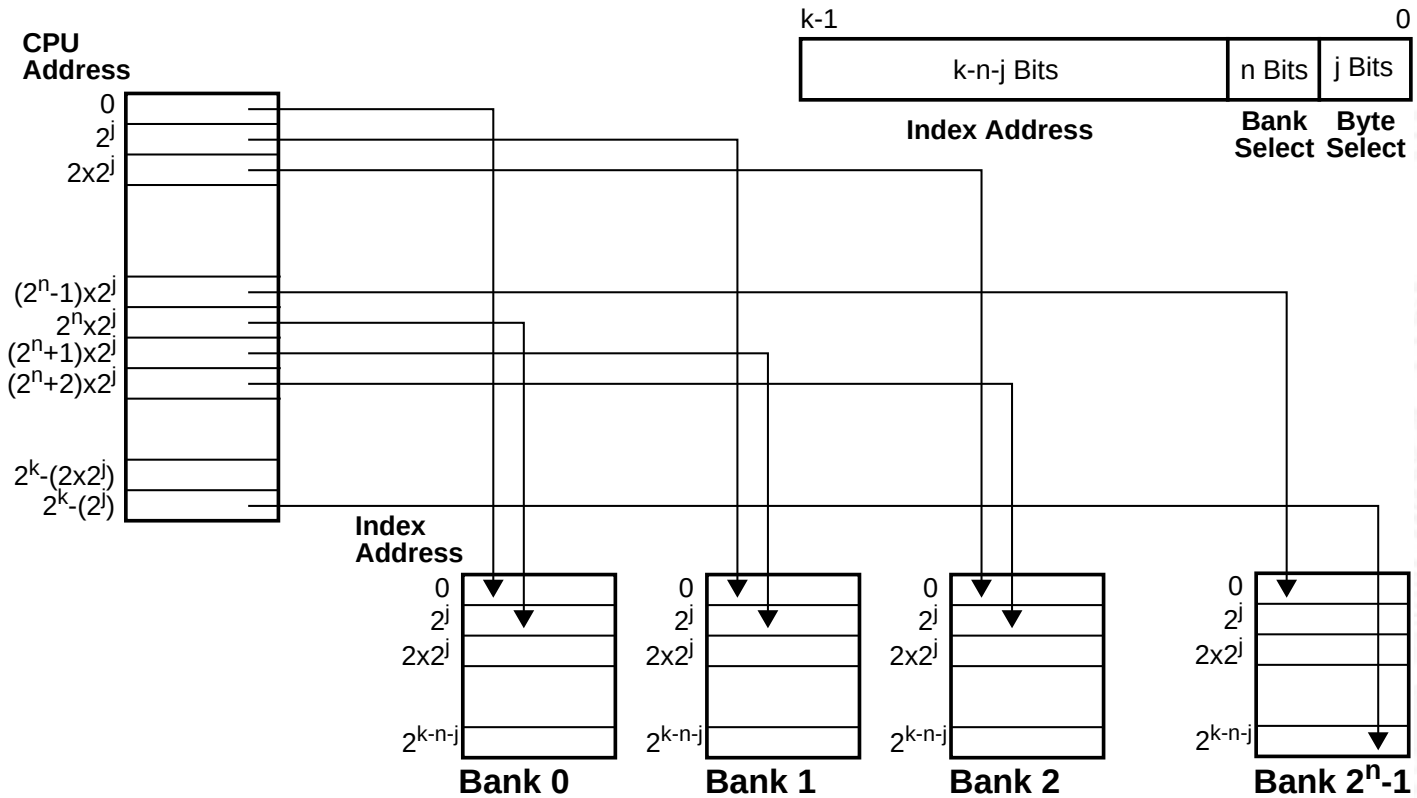
- serielle Zeilen- und Spaltenadressierung
  - \+ weniger Adressleitungen
- Speicher benötigen „Refresh“ nach Zugriff
- Zugriffszeit: Adresse und Daten senden
  Zykluszeit: minimale Zeit vom Beginn eines Zugriffs bis zum nächsten Speicherzugriff

Hauptspeicher: DRAM (cont.)

- „*Interleaving*“: Speichermodule verschränkt Adressieren
  - \+ bei Lokalität der Speicheradressen kann so die Zykluszeit verborgen werden



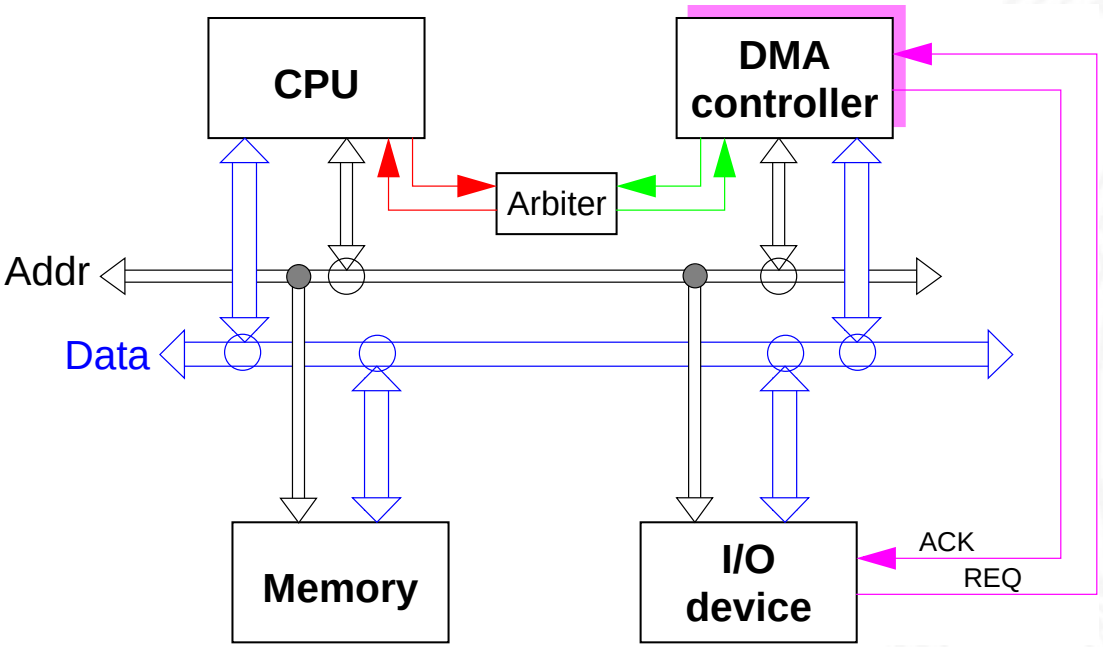
Hauptspeicher: DRAM (cont.)



Hauptspeicher: DMA

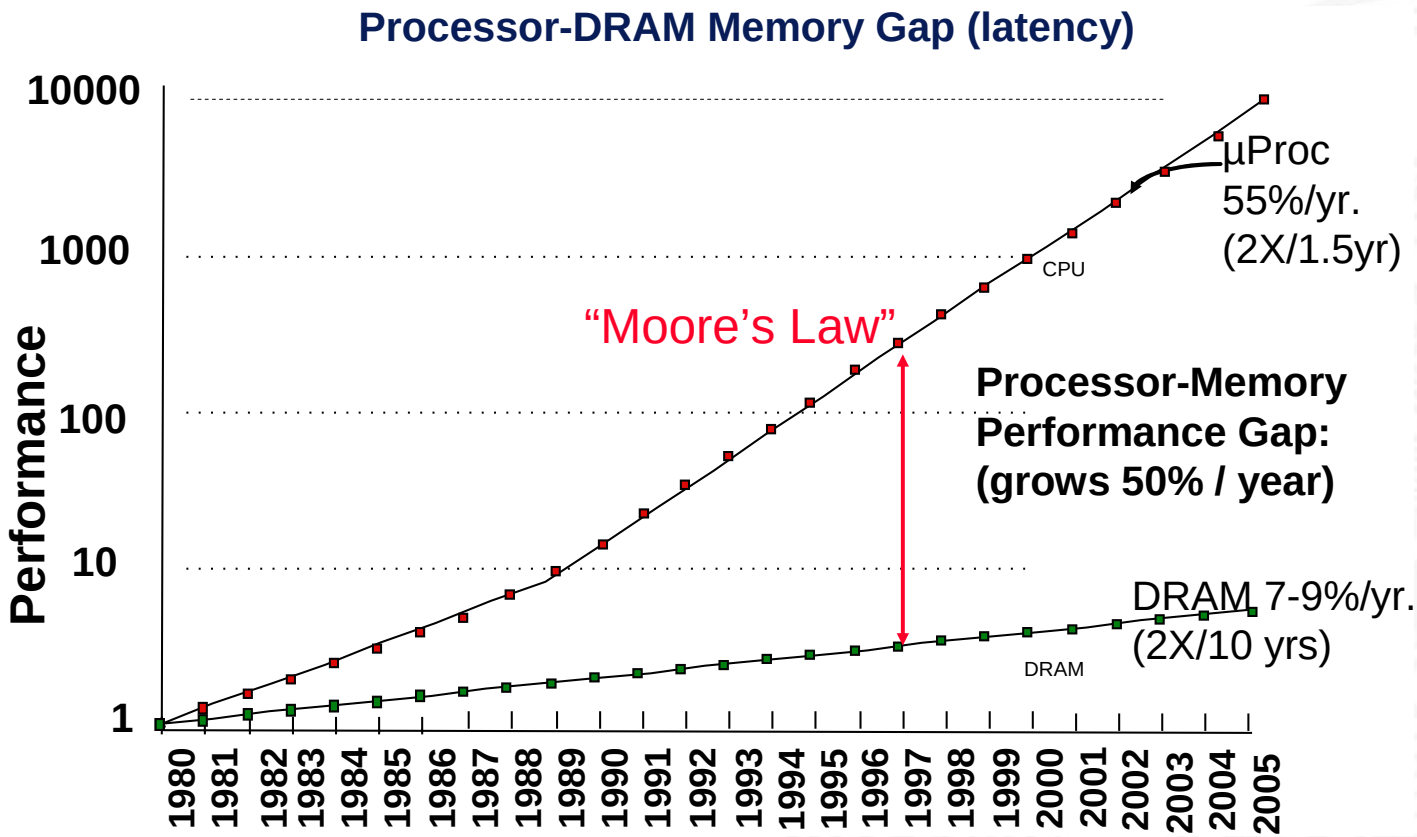
DMA – **D**irect **M**emory **A**ccess

- eigener Controller zum Datentransfer
- \+ Speicherzugriffe unabhängig von der CPU
- \+ CPU kann lokal (Register und Cache) weiterrechnen



Cache

- „*Memory Wall*“: DRAM zu langsam für CPU



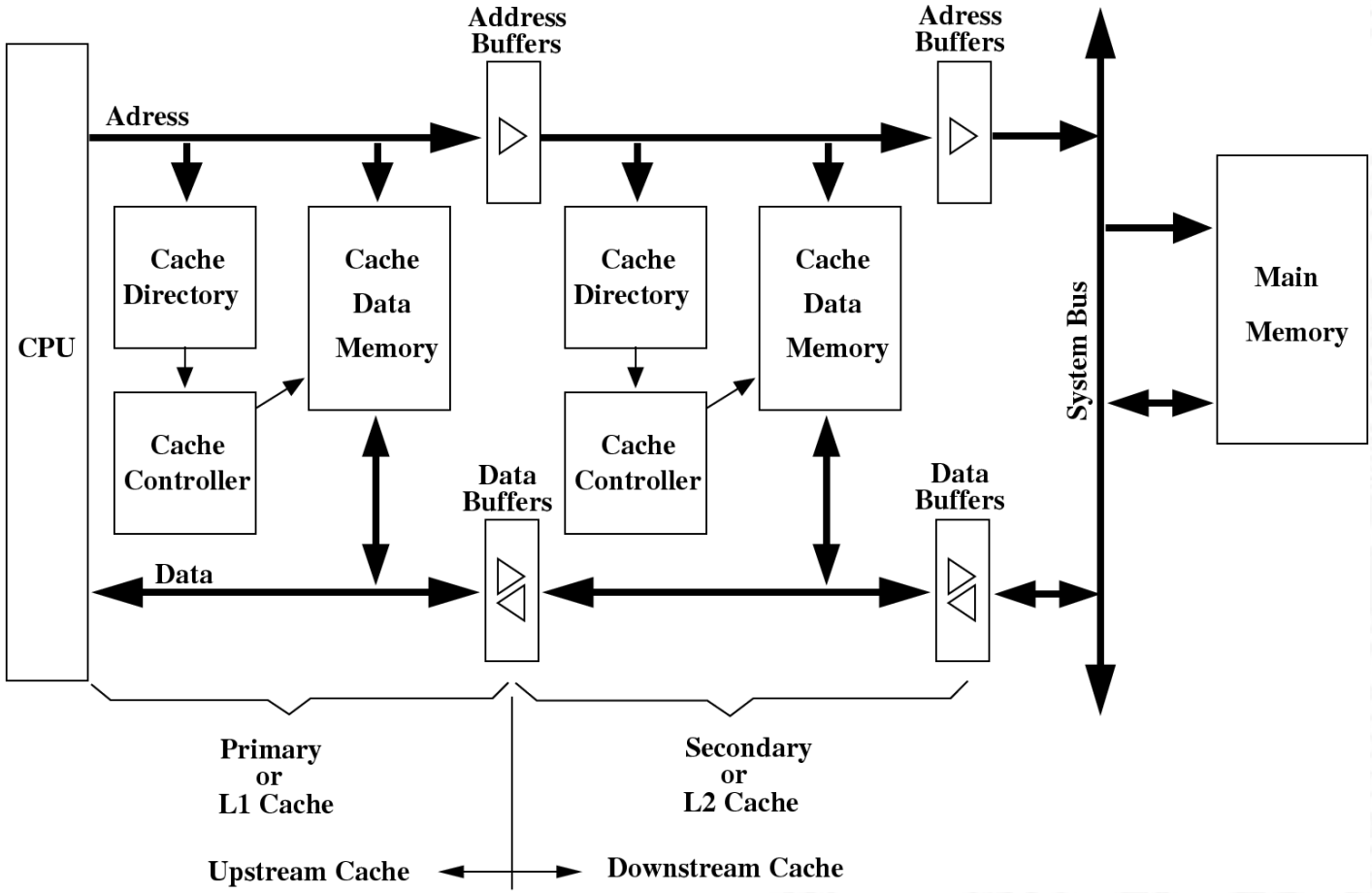
Cache (cont.)

⇒ Cache als schneller Zwischenspeicher zum Hauptspeicher

- technische Realisierung: SRAM
- transparenter Speicher:
  Cache ist für den Programmierer nicht sichtbar!
- `http://de.wikipedia.org/wiki/Cache`
  `http://en.wikipedia.org/wiki/Cache`

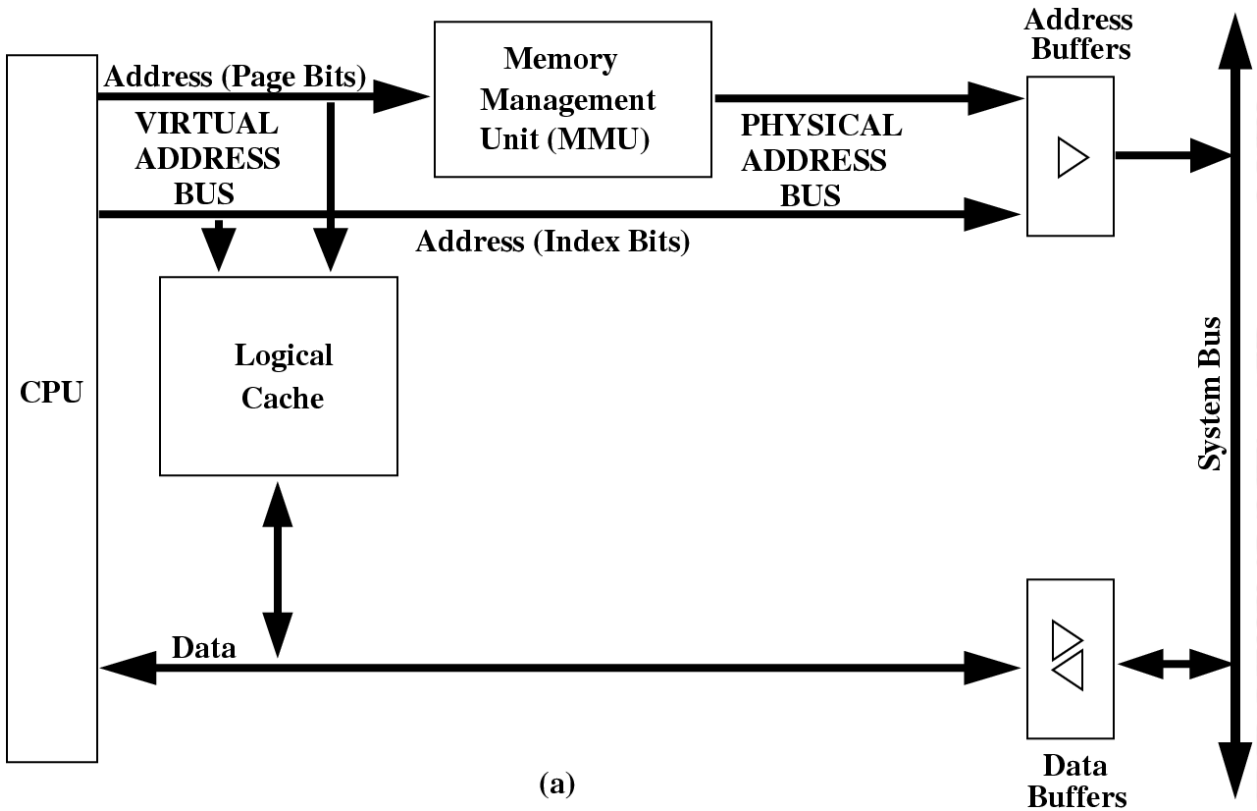
Cache – Position

- First- und Second-Level Cache



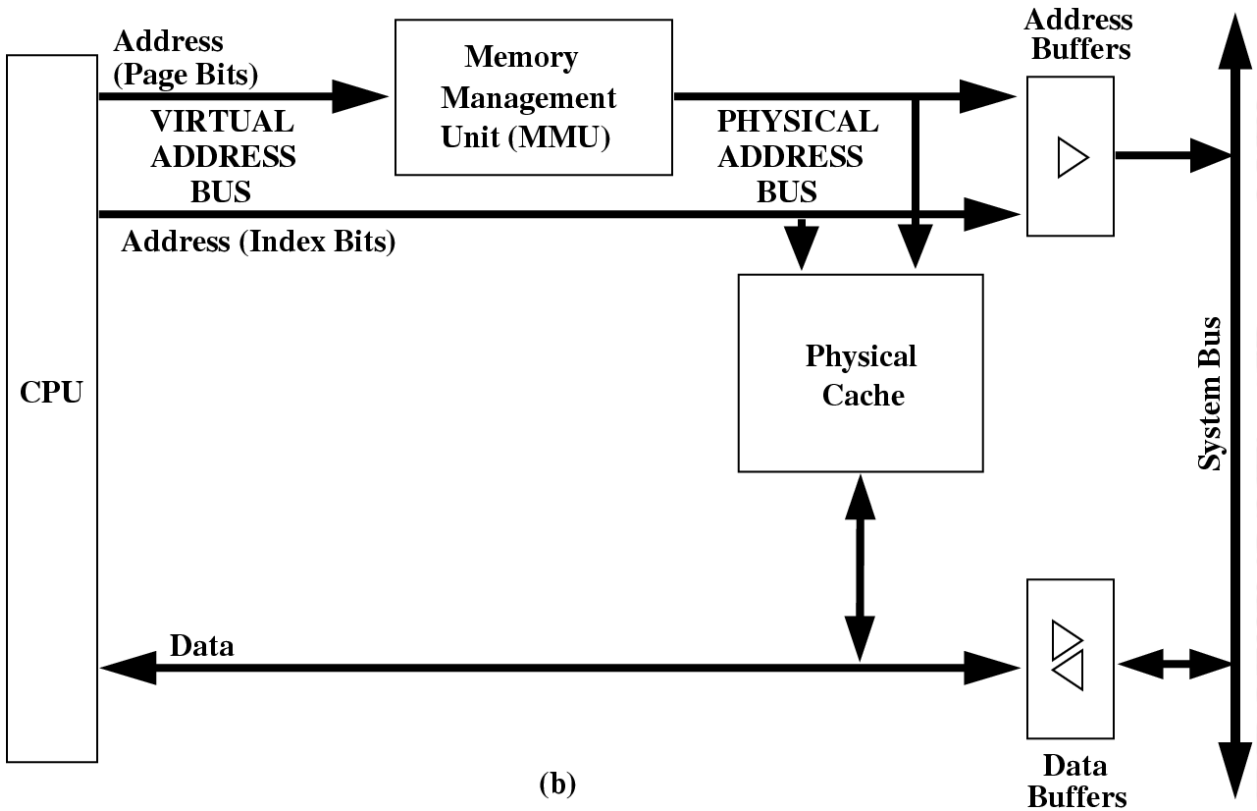
Cache – Position (cont.)

- Virtueller Cache
  - \+ Adressumrechnung durch MMU oft nicht nötig
  - – Cache leeren bei Kontextwechseln



Cache – Position (cont.)

- Physikalischer Cache
  - \+ Cache muss nie geleert werden
  - – Adressumrechnung durch MMU immer nötig



Cache – Strategie

Cachestrategie: *Welche Daten sollen in den Cache?*
Diejenigen, die bald wieder benötigt werden!

- *temporale Lokalität*:
  die Daten, die zuletzt häufig gebraucht wurden
- *räumliche Lokalität*:
  die Daten, die nahe den zuletzt gebrauchten liegen
- verschiedene Platzierungs-, Ersetzungs- und Rückschreibestrategien für den Cache

Cache – Performanz

Cacheperformanz

- Begriffe

Treffer (Hit)		Zugriff auf Datum, ist bereits im Cache
Fehler (Miss)		–"– ist nicht –"–
Treffer-Rate	R_{Hit}	Wahrscheinlichkeit, Datum ist im Cache
Fehler-Rate	R_{Miss}	$1 - R_{Hit}$
Hit-Time	T_{Hit}	Zeit, bis Datum bei Treffer geliefert wird
Miss-Penalty	T_{Miss}	zusätzlich benötigte Zeit bei Fehler

- Mittlere Speicherzugriffszeit $= T_{Hit} + R_{Miss} \cdot T_{Miss}$
- Beispiel

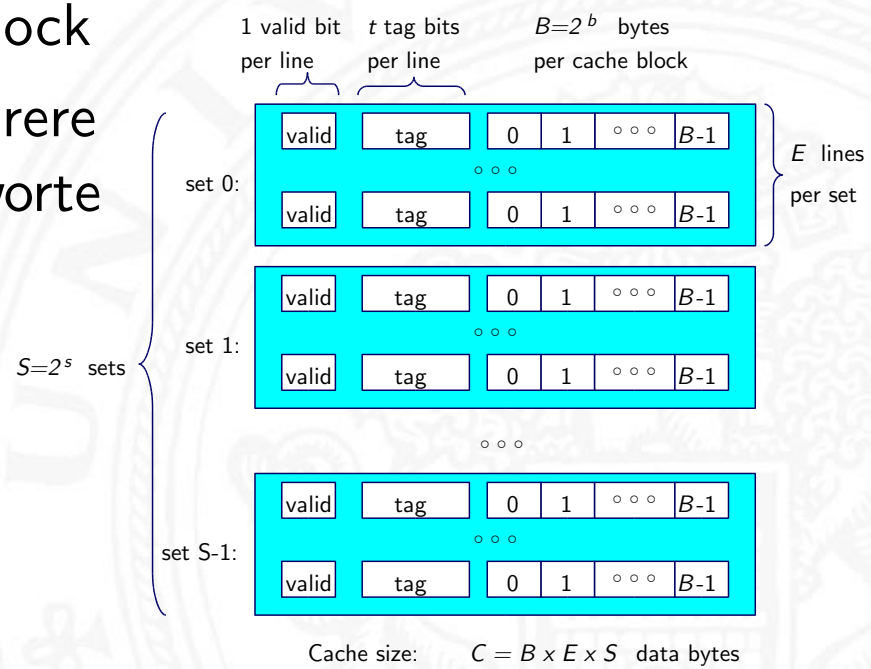
  $T_{Hit} = 1\,Takt,\ T_{Miss} = 20\,Takte,\ R_{Miss} = 5\,\%$

  Mittlere Speicherzugriffszeit $= 2\,Takte$

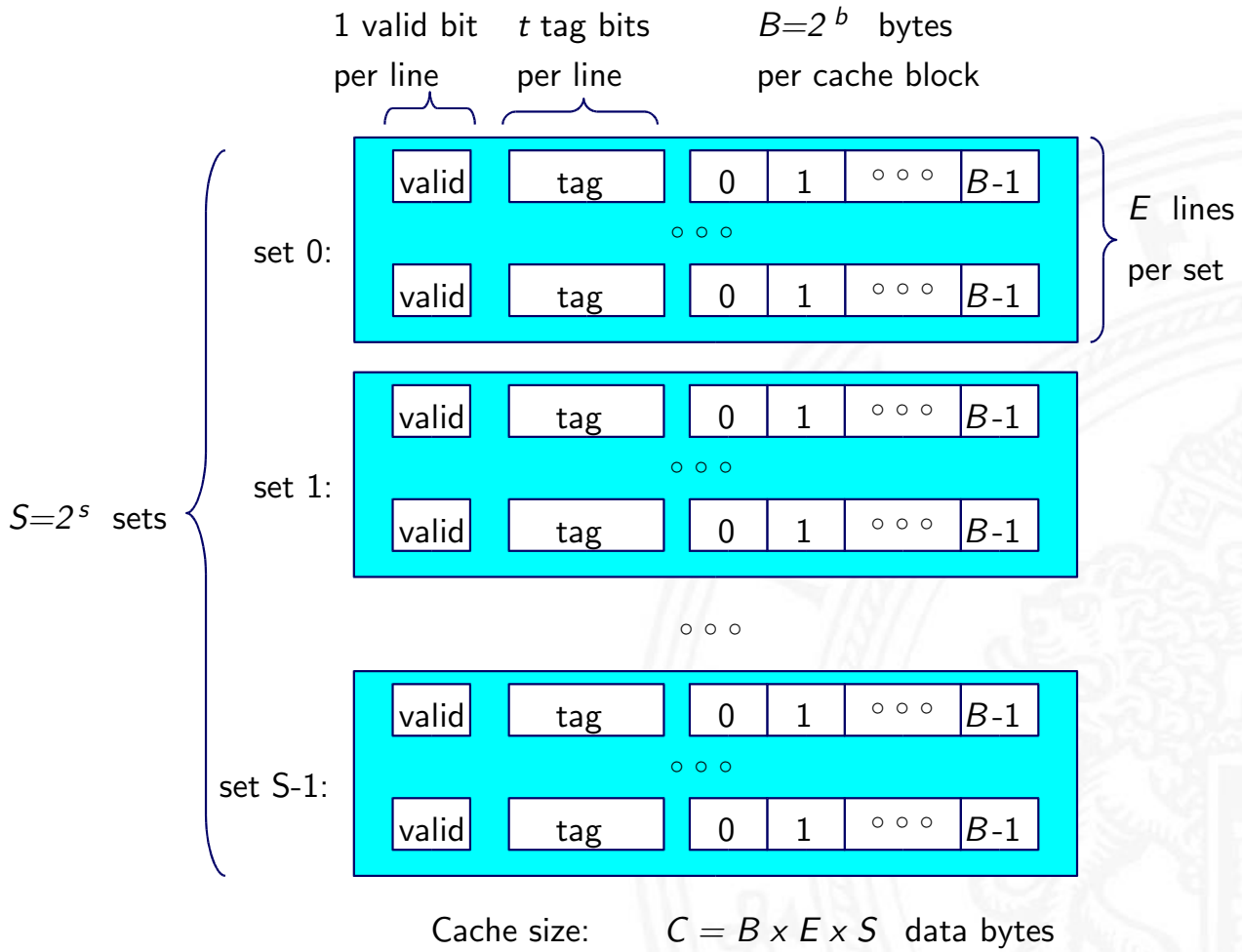
Cache – Organisation

Allgemeine Organisation eines Cache Speichers

- Cache ist ein Array von Speicher-Bereichen („*sets*“)
- Jeder Bereich enthält eine oder mehrere Cachezeilen
- Jede Zeile enthält einen Datenblock
- Jeder Datenblock beinhaltet mehrere (aufeinander folgende) Speicherworte



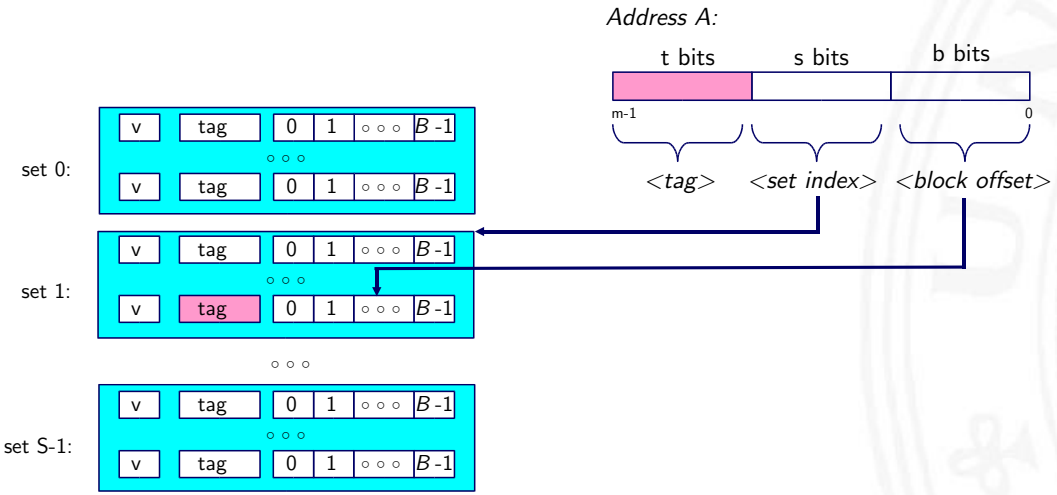
Cache – Organisation (cont.)



Cache – Organisation (cont.)

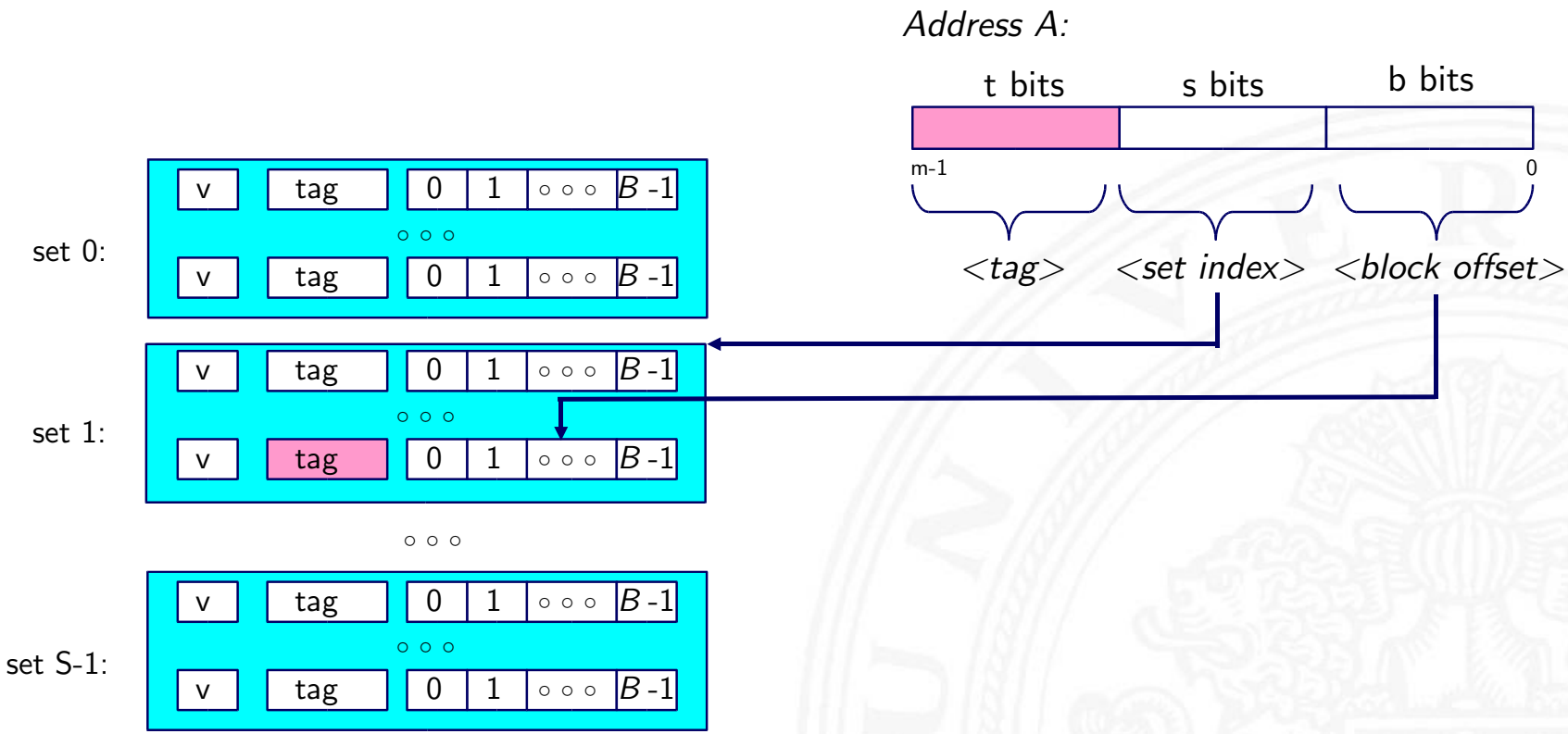
Cache Zugriff

- Das Wort der Adresse A liegt im Cache, wenn die „*tag*“ *Bits* in einer gültigen Zeile $\langle valid \rangle$ im aktuellen Bereich $\langle set\ index \rangle$ mit dem $\langle tag \rangle$ der Adresse A übereinstimmen
- Das gesuchte Wort steht im Cache an der Stelle $\langle block\ offset \rangle$ hinter dem Anfang des Blocks



◂ Direct Mapped
◂ Set Associative

Cache – Organisation (cont.)



Cache – Adressierung

- *Welchen Platz im Cache belegt ein Datum des Hauptspeichers?*
- drei Verfahren

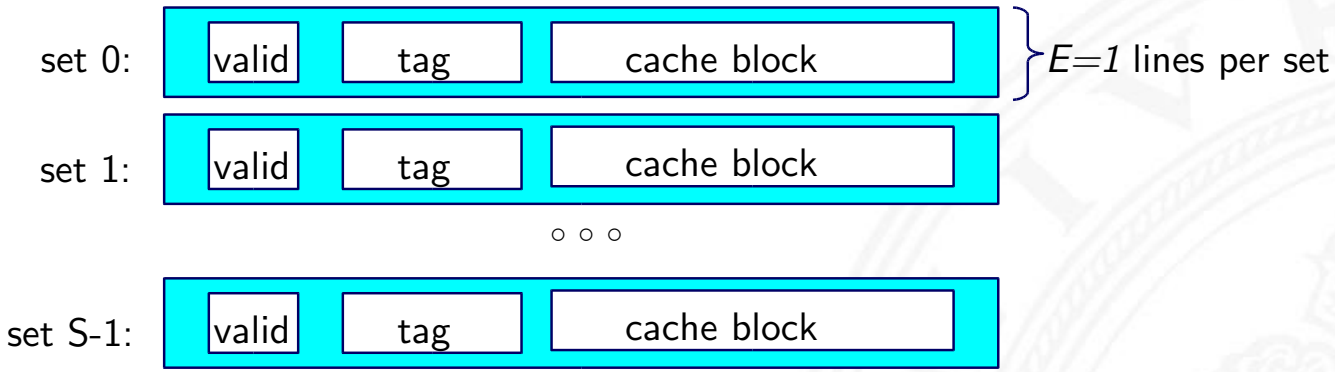
  Direkt abgebildet / direct mapped jeder Speicheradresse ist genau eine Cache-Speicherzelle zugeordnet

  n-fach Bereichsassoziativ / set associative jeder Speicheradresse ist eine von E möglichen Cache-Speicherzellen zugeordnet

  Voll-Assoziativ jeder Speicheradresse kann jede beliebige Cache-Speicherzelle zugeordnet werden

Cache: direkt abgebildet

- Jeder Adresse ist genau eine Speicherzelle im Cache zugeordnet
- Verfügt über genau 1 Zeile pro Bereich S Bereiche (**S**ets)



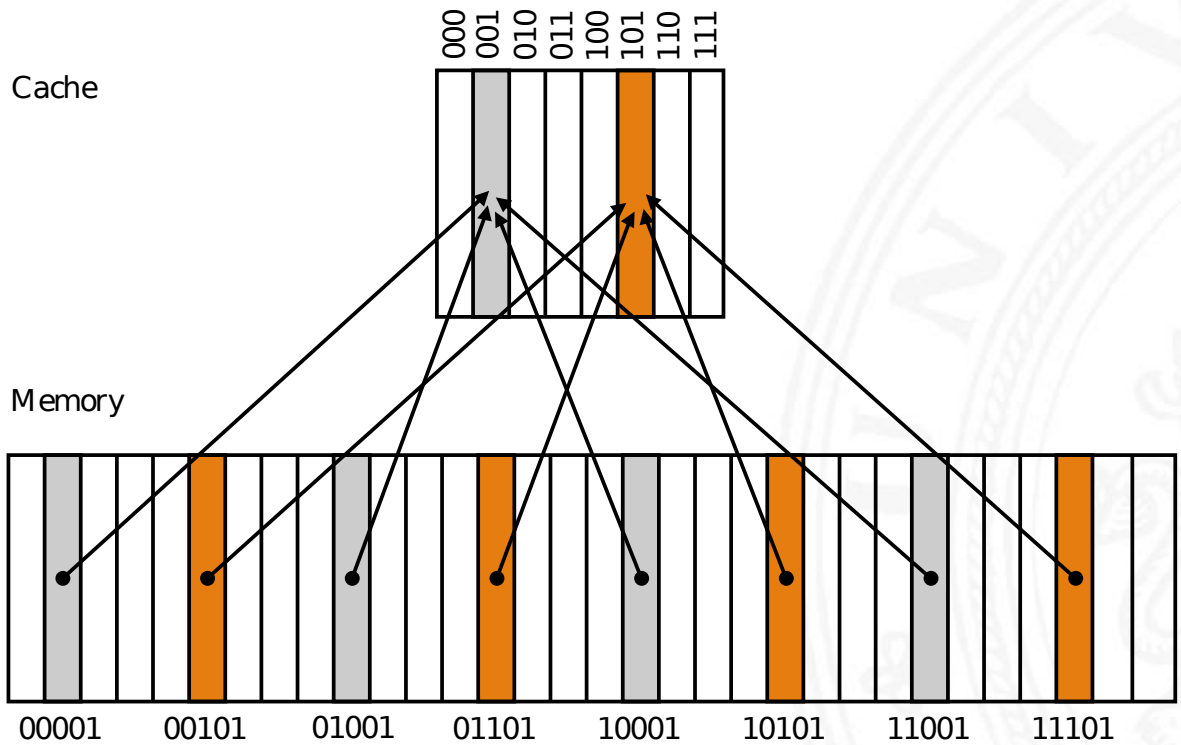
+ einfachste Cache-Art
+ große Caches möglich
– Effizienz, z.B. Zugriffe auf A, $A + n \cdot S \ldots$

Cache: direkt abgebildet (cont.)

Adressabbildung über Bereichsausschnitte

▸ Cache Zugriff

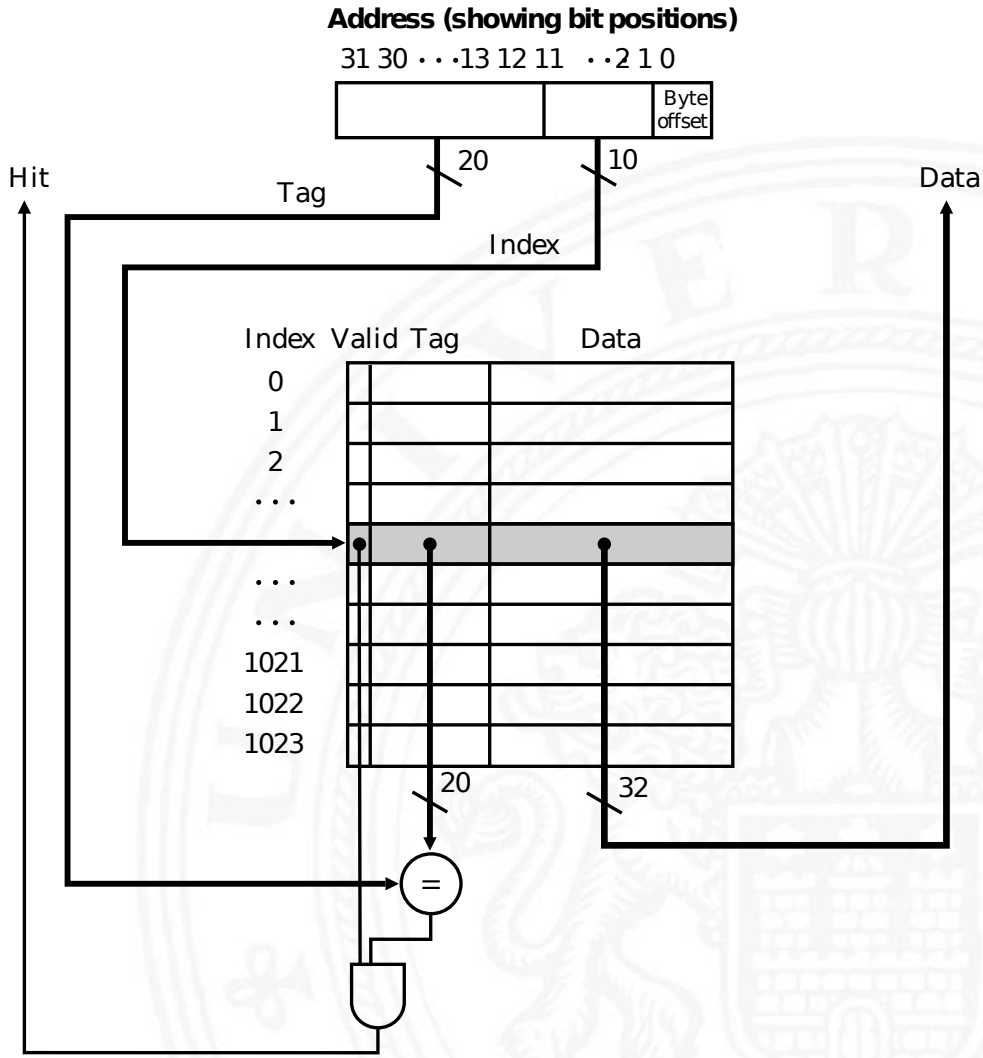
- $A_{Cache} = A_{Memory} \bmod S$, S Cache-Einträge
  $Tag = A_{Memory} \;/\; S$



Cache: direkt abgebildet (cont.)

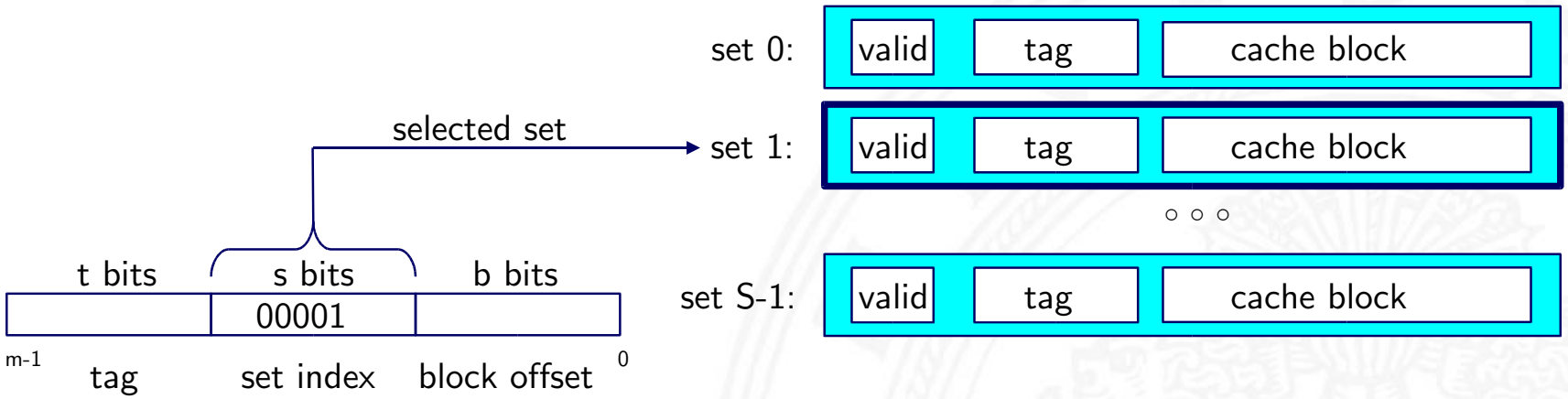
Direct Mapped Cachezugriff

- Rückabbildung:
  A_{Memory} aus *Tag* und
  A_{Cache} bestimmen



Cache: direkt abgebildet (cont.)

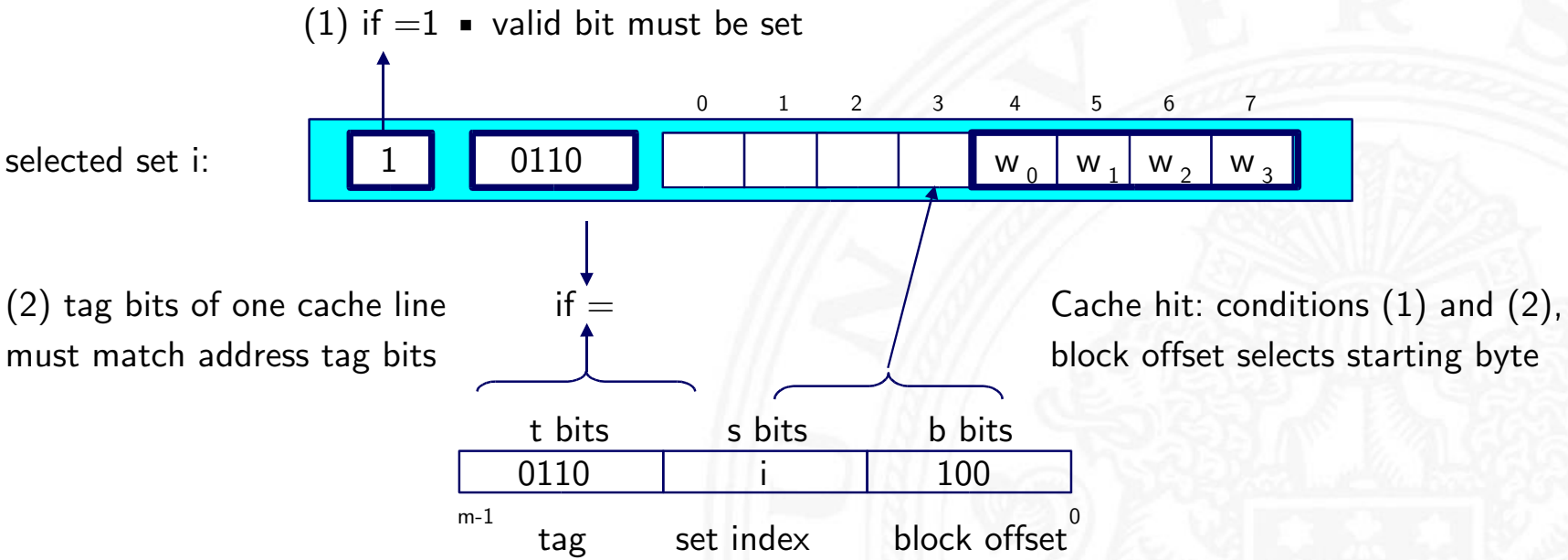
1. Bereichsauswahl
   - Bereichsindex-Bits $\langle set\ index \rangle$ legen den Bereich (die Cachezeile A_{Cache}) fest



Cache: direkt abgebildet (cont.)

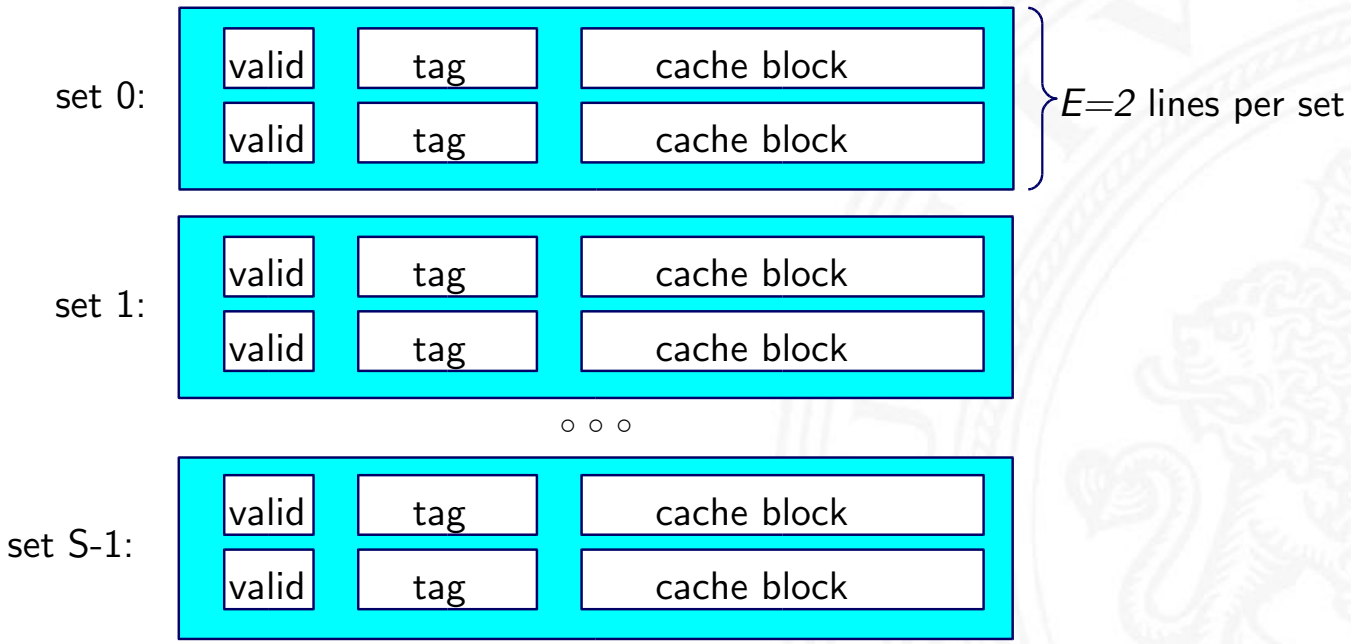
2. „Line matching“ und Wortselektion

- „Line matching“: gültige Zeile bei übereinstimmendem $\langle tag \rangle$
- Wortselektion: dann extrahiere das Wort $\langle block\ offset \rangle$



Cache: bereichsassoziativ

- Jeder Adresse des Speichers ist ein Cachebereich S mit E Speicherzellen zugeordnet
- Verfügen über mehr als eine Zeile pro Bereich (z.B.: $E = 2$)



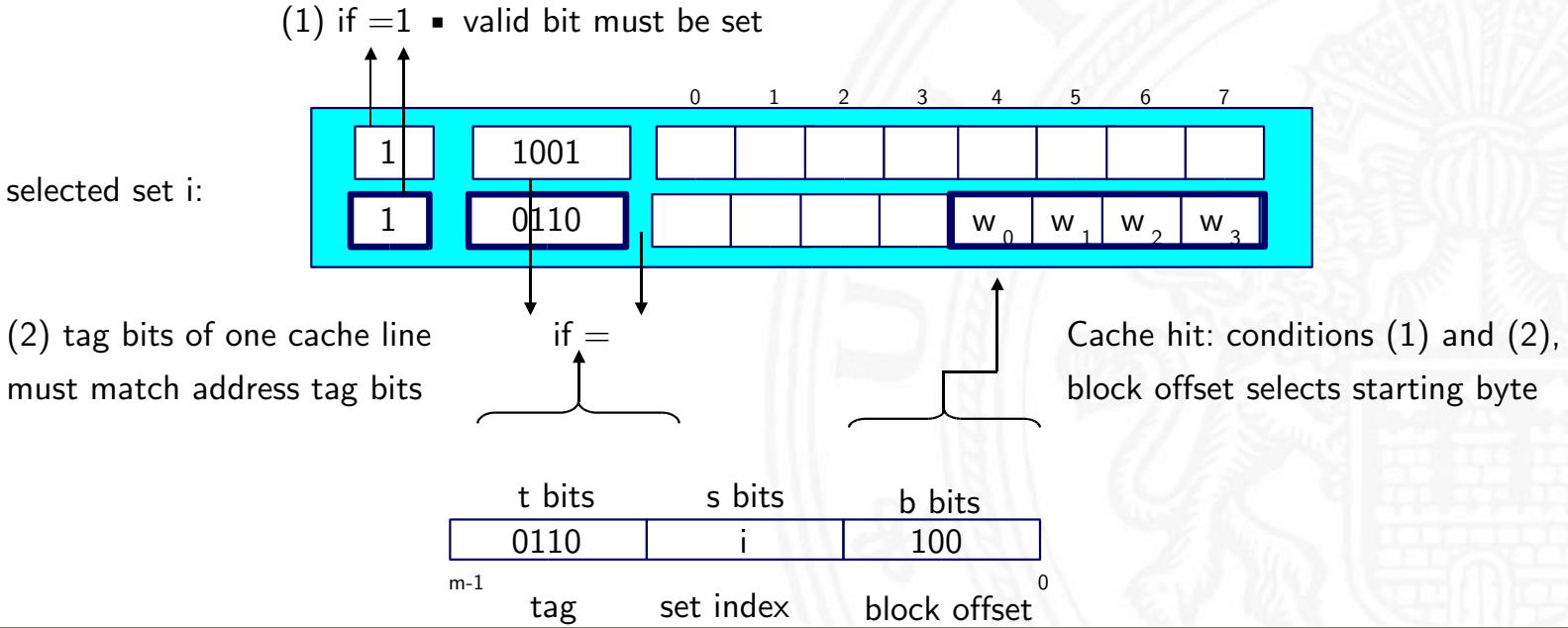
Cache: bereichsassoziativ (cont.)

Adressabbildung über Bereichsausschnitte

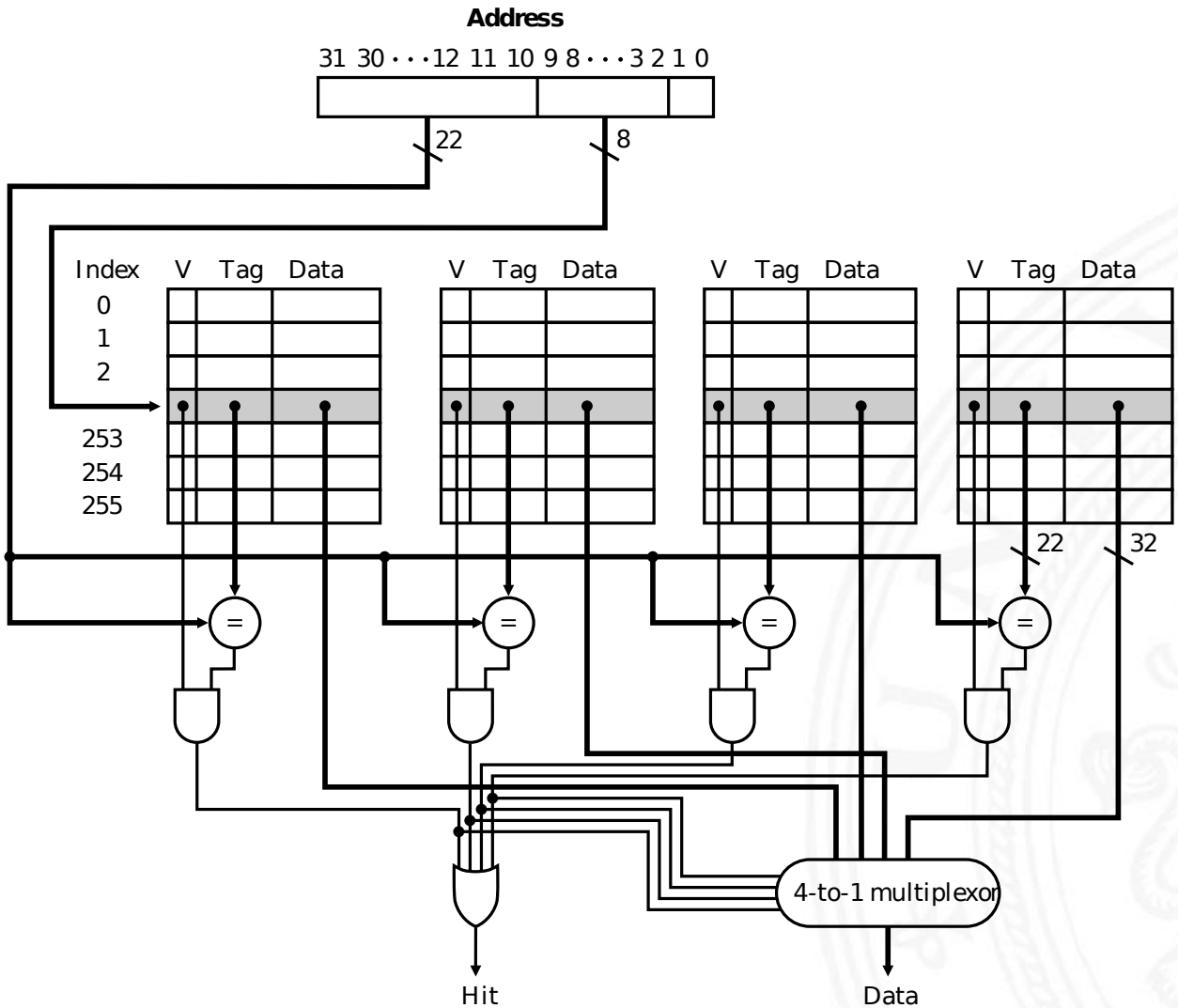
▸ Cache Zugriff

Set Associative Cachezugriff

1. Bereichsauswahl, wie bei „direkt abgebildetem Cache“
2. „Line matching“ und Wortselektion
   - mehrere Tags vergleichen: erhöhter Hardwareaufwand



Cache: bereichsassoziativ (cont.)

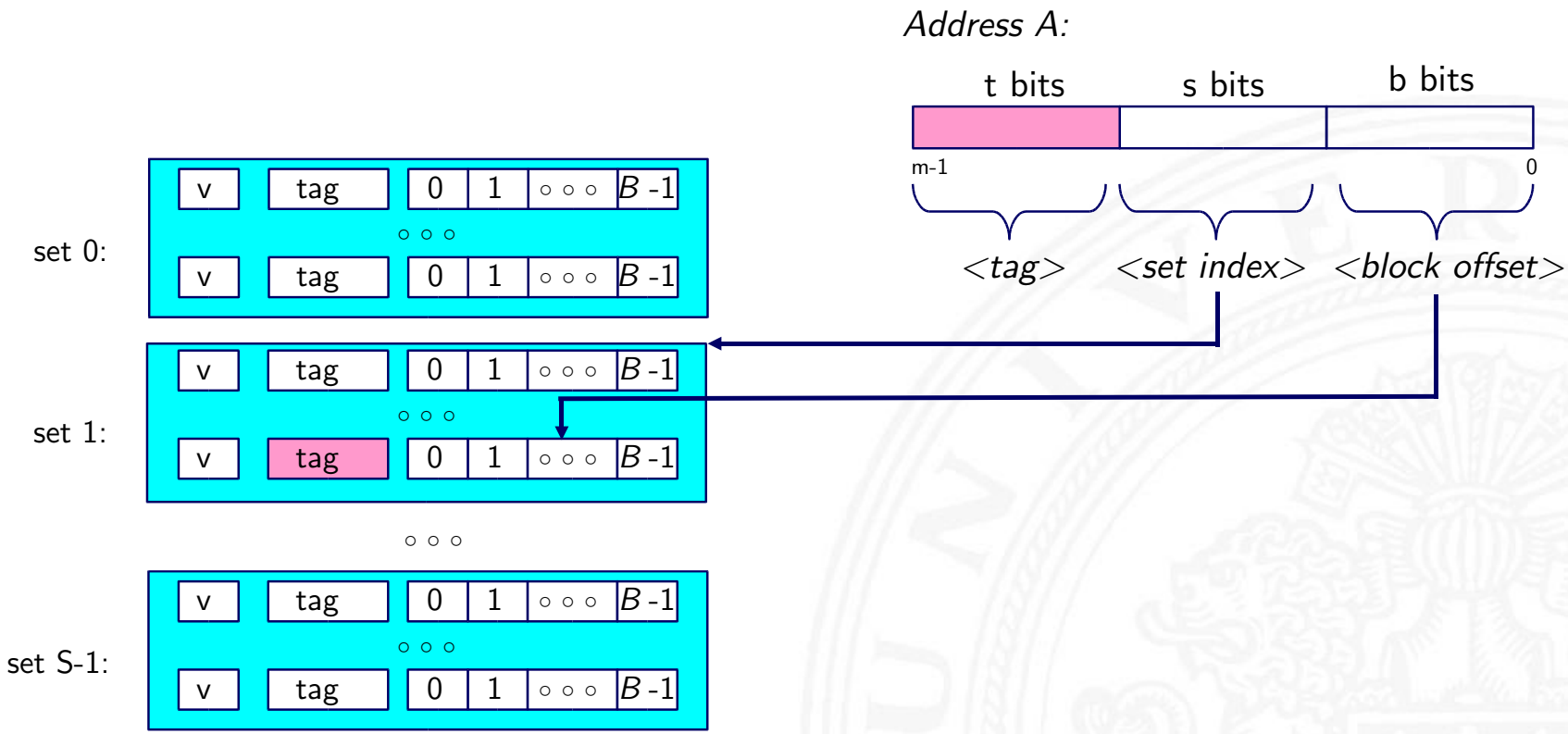


Cache: voll-assoziativ

- Jeder Adresse des Speichers kann jede beliebige Speicherzelle des Caches zugeordnet werden
- Spezialfall: nur ein Cachebereich S

– benötigt E-Vergleicher
– nur für sehr kleine Caches realisierbar

Cache – Dimensionierung



- Parameter: S, B, E

Cache – Dimensionierung (cont.)

- Cache speichert immer größere Blöcke / „*Cache-Line*“
- Wortauswahl durch $\langle$*block offset*$\rangle$ in Adresse

\+ nutzt räumliche Lokalität aus

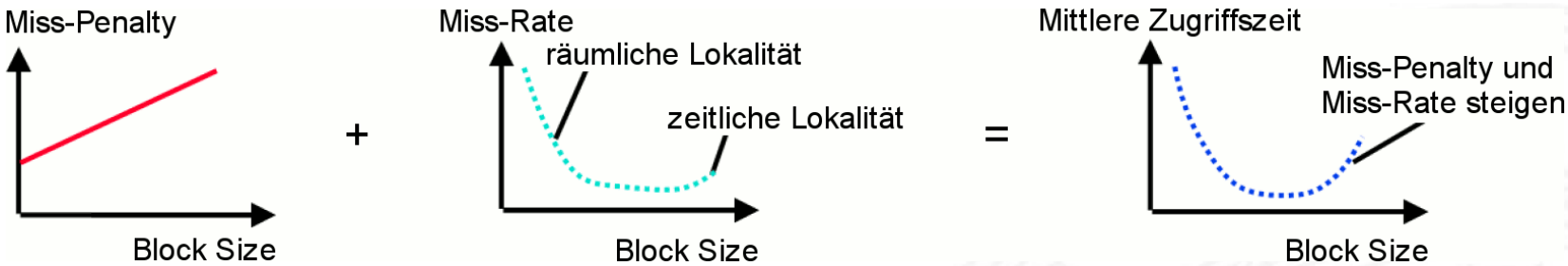
\+ Breite externe Datenbusse

\+ nutzt Burst-Adressierung des Speichers: Adresse nur für erstes Wort vorgeben, dann automatisches Inkrement

\+ kürzere interne Cache-Adressen

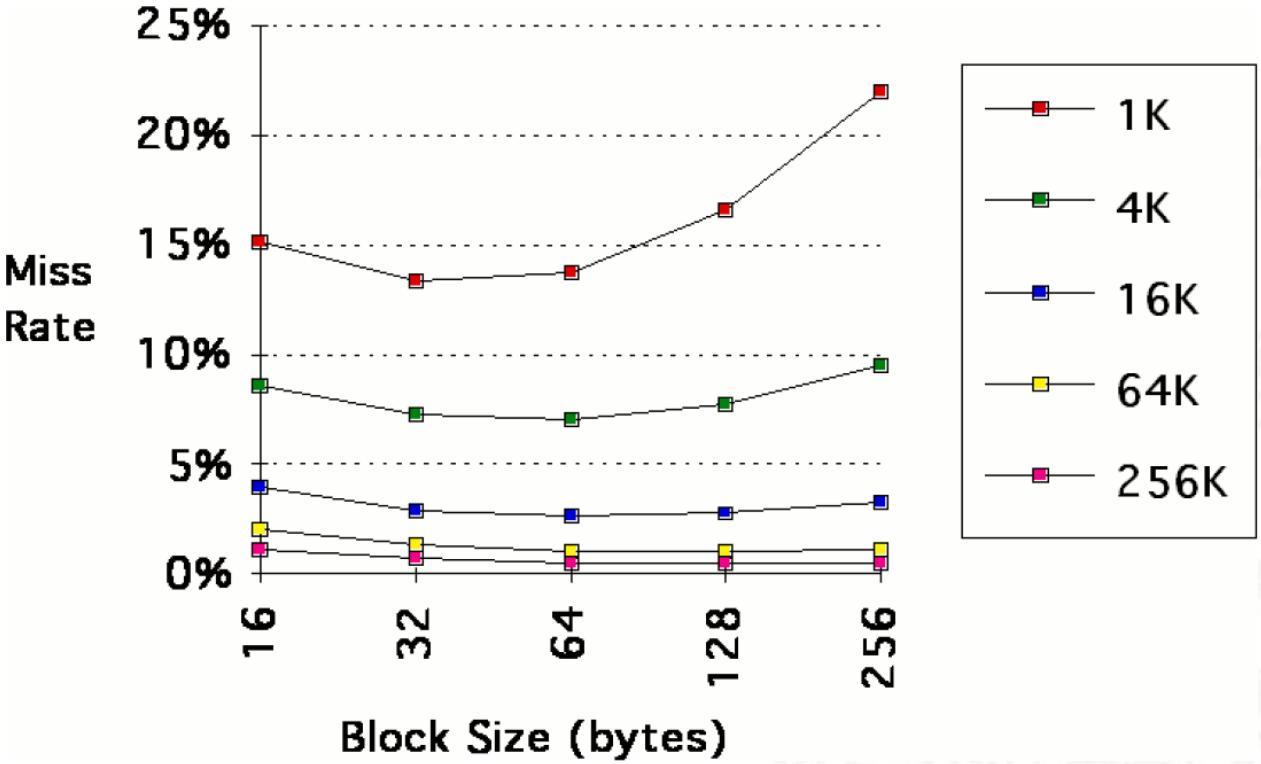
Cache – Dimensionierung (cont.)

Cache- und Block-Dimensionierung



- Blockgröße klein, viele Blöcke
  - \+ kleinere Miss-Penalty
  - \+ temporale Lokalität
  - – räumliche Lokalität
- Blockgröße groß, wenig Blöcke
  - – größere Miss-Penalty
  - – temporale Lokalität
  - \+ räumliche Lokalität

Cache – Dimensionierung (cont.)



- Block-Size: 32...128 Byte
  L1-Cache: 4...256 KByte
  L2-Cache: 256...4096 KByte

Cache – Dimensionierung (cont.)

- zusätzliche Software-Optimierungen
  - Ziel: Cache Misses reduzieren
  - Schleifen umsortieren, verschmelzen...
  - Datenstrukturen zusammenfassen...

Cache Ersetzungsstrategie

Wenn der Cache gefüllt ist, welches Datum wird entfernt?

- zufällige Auswahl
- LRU (**L**east **R**ecently **U**sed):
  der „älteste“ nicht benutzte Cache Eintrag
  - echtes LRU als Warteschlange realisiert
  - Pseudo LRU mit baumartiger Verwaltungsstruktur:
    Zugriff wird paarweise mit einem Bit markiert,
    die Paare wieder zusammengefasst usw.
- LFU (**L**east **F**requently **U**sed):
  der am seltensten benutzte Cache Eintrag
  - durch Zugriffszähler implementiert

Cache Schreibstrategie

Wann werden modifizierte Daten des Cache zurückgeschrieben?

- Write-Through: beim Schreiben werden Daten sofort im Cache und im Hauptspeicher modifiziert
  - \+ andere Bus-Master sehen immer den „richtigen“ Speicherinhalt: *Cache-Kohärenz*
  - – Werte werden unnötig oft in Speicher zurückgeschrieben
- Write-Back: erst in den Speicher schreiben, wenn Datum des Cache ersetzt werden würde
  - \+ häufig genutzte Werte (z.B. lokale Variablen) werden nur im Cache modifiziert
  - – Cache-Kohärenz ist nicht gegeben
  - ⇒ spezielle Befehle für „Cache-Flush“
  - ⇒ „non-cacheable“ Speicherbereiche

Cache Kohärenz

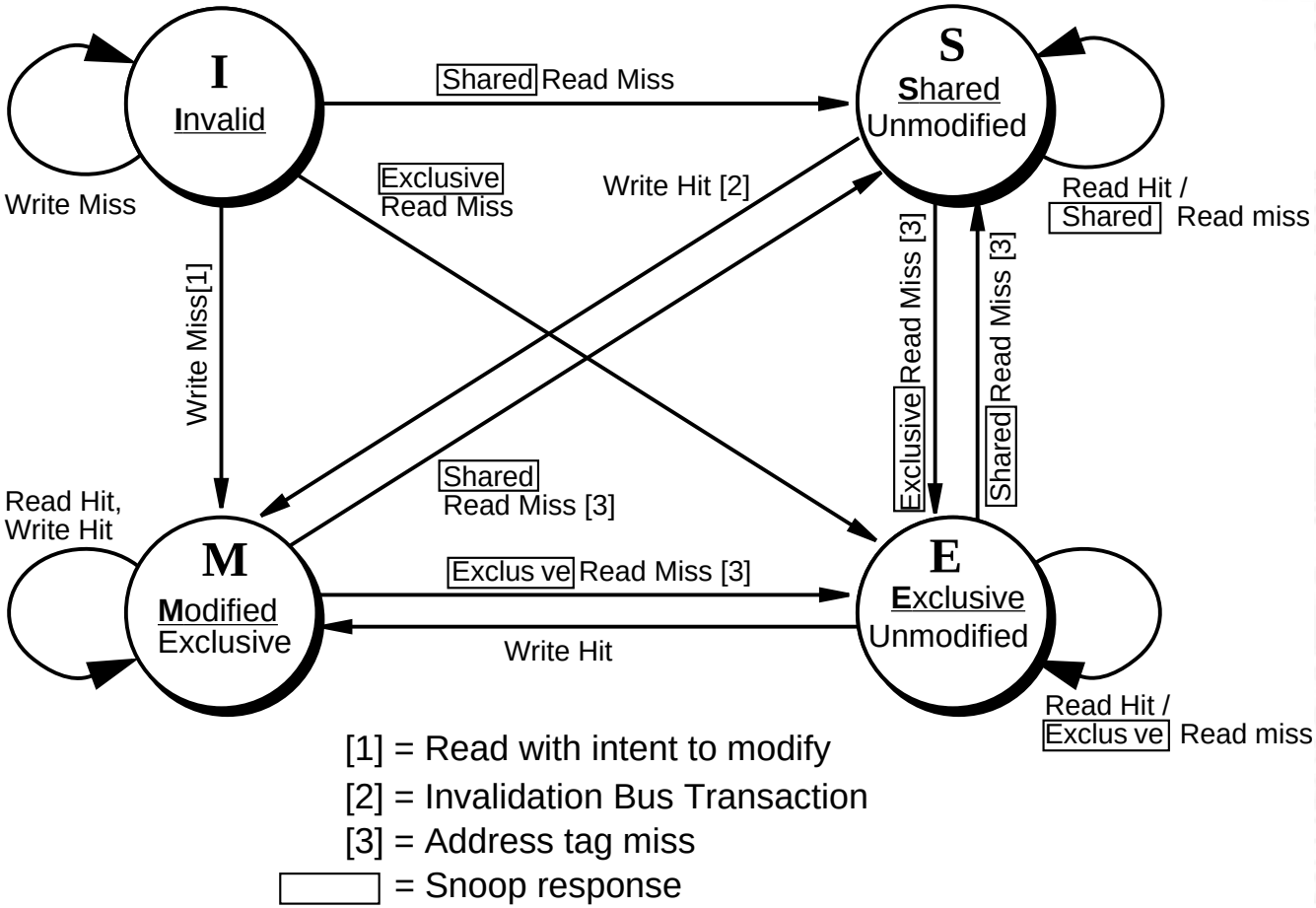
- Daten zwischen Cache und Speicher konsistent halten
- notwendig wenn auch andere Einheiten (Bus-Master) auf Speicher zugreifen können
- Harvard-Architektur hat getrennte Daten- und Instruktions-Speicher: einfacherer Instruktions-Cache
  - Instruktionen sind read-only
  - kein Cache-Kohärenz Problem
- „Snooping“
  - Cache „lauscht“ am Speicherbus
  - externer Schreibzugriff $\Rightarrow$ Cache aktualisieren / ungültig machen
  - externer Lesezugriff $\Rightarrow$ Cache liefert Daten

Cache Kohärenz (cont.)

- MESI Protokoll
  - besitzt zwei Statusbits für die Protokollzustände
  - **M**odified: Inhalte der Cache-Line wurden modifiziert
  - **E**xclusive unmodified: Cache-Line „gehört“ dieser CPU, nicht modifiz.
  - **S**hared unmodified: Inhalte sind in mehreren Caches vorhanden
  - **I**nvalid: ungültiger Inhalt, Initialzustand

Cache Kohärenz (cont.)

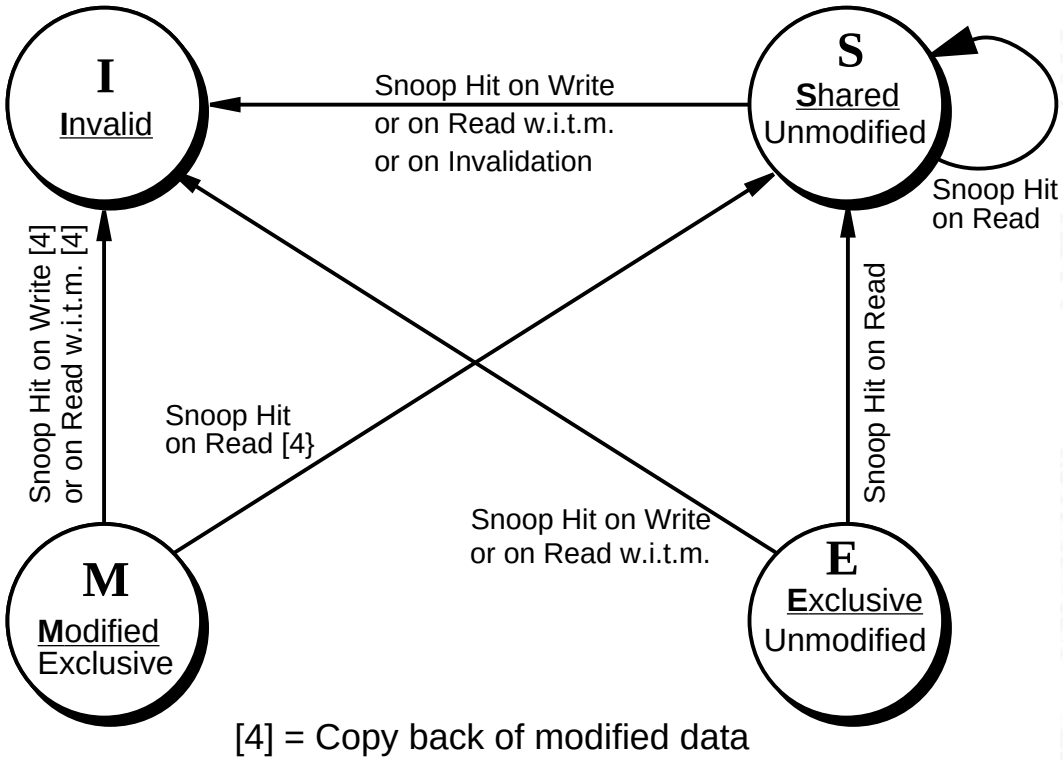
- Zustandsübergänge für „Bus Master“ CPU



[1] = Read with intent to modify
[2] = Invalidation Bus Transaction
[3] = Address tag miss
□ = Snoop response

Cache Kohärenz (cont.)

- Zustandsübergänge für “Snooping“ CPU



[4] = Copy back of modified data

Optimierung der Cachezugriffe

- Mittlere Speicherzugriffszeit $= T_{Hit} + R_{Miss} \cdot T_{Miss}$

$\Rightarrow$ Verbesserung der Cache Performanz durch kleinere T_{Miss} am einfachsten zu realisieren

- mehrere Cache Ebenen
- Critical Word First: bei großen Cache Blöcken (mehrere Worte) gefordertes Wort zuerst holen und gleich weiterleiten
- Read-Miss hat Priorität gegenüber Write-Miss
  $\Rightarrow$ Zwischenspeicher für Schreiboperationen (Write Buffer)
- Merging Write Buffer: aufeinander folgende Schreiboperationen zwischenspeichern und zusammenfassen
- Victim Cache: kleiner voll-assoziativer Cache zwischen direct-mapped Cache und nächster Ebene „sammelt“ verdrängte Cache Einträge

Optimierung der Cachezugriffe (cont.)

⇒ Verbesserung der Cache Performanz durch kleinere R_{Miss}
- größere Caches (– mehr Hardware)
- höhere Assoziativität (– langsamer)

⇒ weitere Optimierungstechniken
- Software Optimierungen
- Prefetch: Hardware (Stream Buffer)
  Software (Prefetch Operationen)
- Cache Zugriffe in Pipeline verarbeiten
- Trace Cache: im Instruktions-Cache werden keine Speicherinhalte, sondern ausgeführte Sequenzen (*trace*) einschließlich ausgeführter Sprünge gespeichert

  Beispiel: Pentium IV

Virtueller Speicher und Memory Management

Speicher-Paradigmen

- Programmierer
  - ein großer Adressraum
  - linear adressierbar
- Betriebssystem
  - eine Menge laufender Tasks / Prozesse
  - read-only Instruktionen
  - read-write Daten

Virtueller Speicher und Memory Management (cont.)

Aufgaben des Betriebssystems

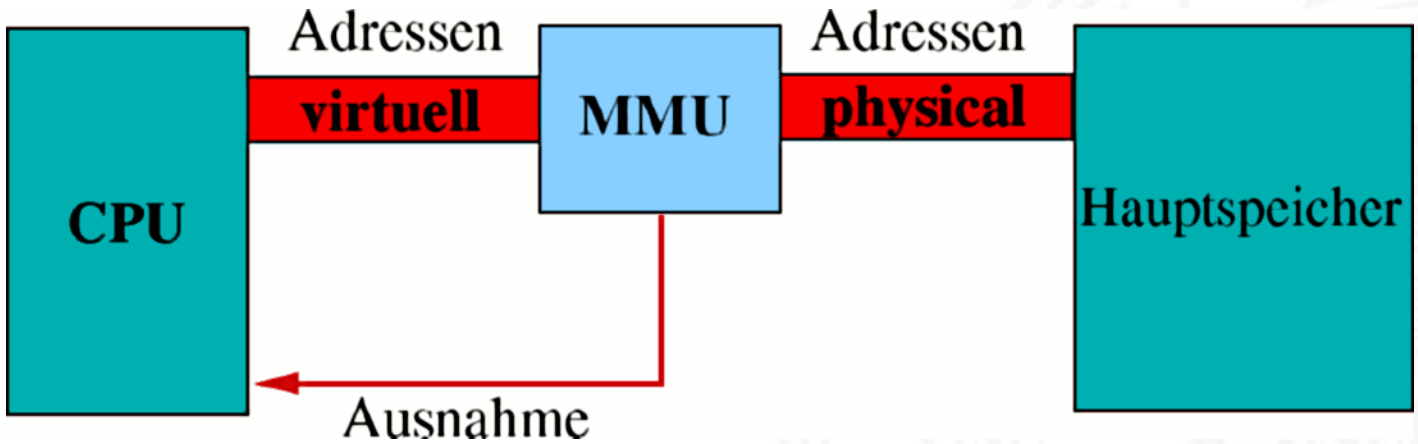
- effiziente Nutzung des Speichers gewährleisten
  - Prozess benötigt oft nicht alle Teile seines Adressraums
  - Speicherbedarf aller Prozesse kann größer als der verfügbare Hauptspeicher sein

  $\Rightarrow$ nicht benötigte Teile des Adressraums auf Hintergrundspeicher auslagern
- Speicherschutz
  - Prozesse haben nur Zugriff auf eigenen Daten

Virtueller Speicher und Memory Management (cont.)

Prinzip des virtuellen Speichers

- jeder Prozess besitzt seinen eigenen virtuellen Adressraum
- MMU – **M**emory **M**anagement **U**nit



- Umsetzung von virtuellen zu physikalischen Adressen, Programm-Relokation

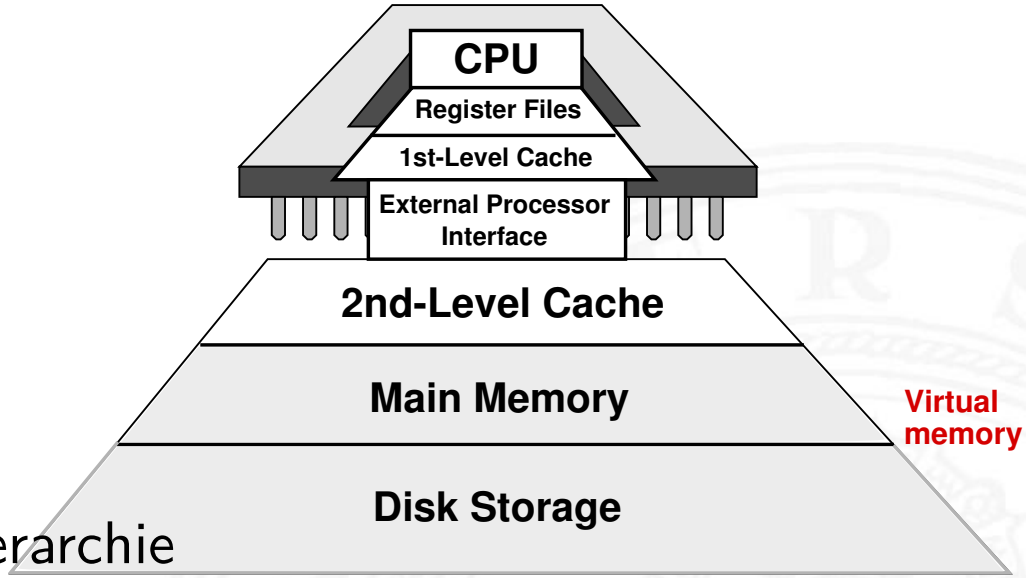
Virtueller Speicher und Memory Management (cont.)

- Umsetzungstabellen werden vom Betriebssystem erzeugt und aktualisiert
- wegen des Speicherbedarfs der Tabellen beziehen sich diese auf größere Speicherblöcke

- Umgesetzt wird nur die Anfangsadresse, der Offset innerhalb des Blocks bleibt unverändert
- Blöcke dieses virtuellen Adressraums können durch Betriebssystem ausgelagert werden
- Methoden zur Implementation virtuellen Speichers
  - *Segmentierung*
  - *Paging*, Speicherzuordnung durch *Seiten*
  - gemischte Verfahren (Standard bei: Desktops, Workstations...)

Virtueller Speicher und Memory Management (cont.)

- Hauptspeicher als Cache für den Plattenspeicher

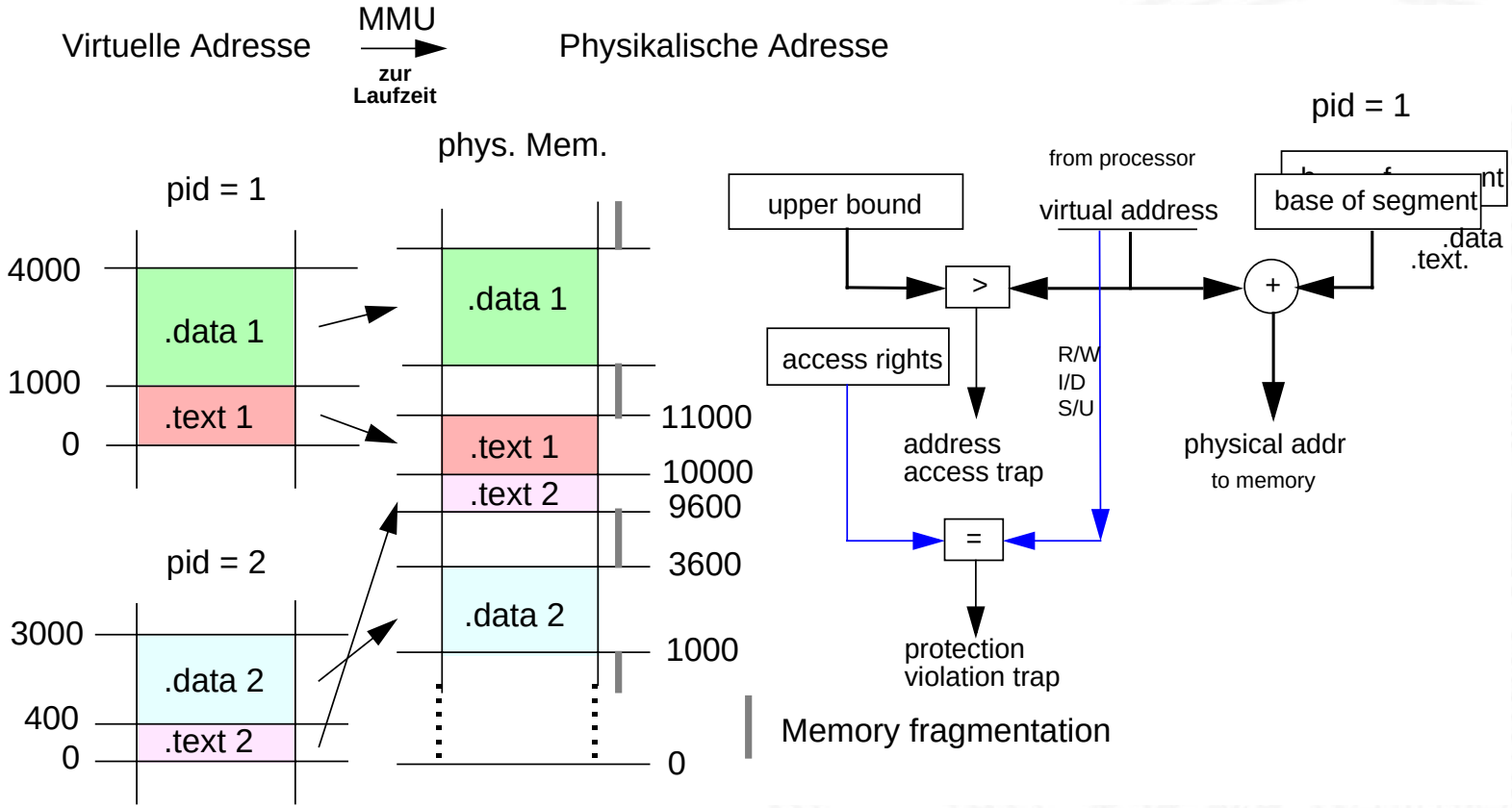


- Parameter der Speicherhierarchie

	1st-Level Cache	virtueller Speicher
Blockgröße	16-128 Byte	4-64 kByte
Hit-Dauer	1-2 Zyklen	40-100 Zyklen
Miss Penalty	8-100 Zyklen	70.000-6.000.000 Zyklen
Miss Rate	0,5-10 %	0,00001-0,001 %
Speichergröße	8-64 kByte	16-8192 MByte

Virtueller Speicher: Segmentierung

- Unterteilung des Adressraums in kontinuierliche Bereiche *variabler* Größe



Virtueller Speicher: Segmentierung (cont.)

- Idee: Trennung von Instruktionen, Daten und Stack

$\Rightarrow$ Abbildung von *Programmen* in den *Hauptspeicher*

$+$ Inhalt der Segmente: logisch zusammengehörige Daten

$+$ getrennte Zugriffsrechte, Speicherschutz

$+$ exakte Prüfung der Segmentgrenzen

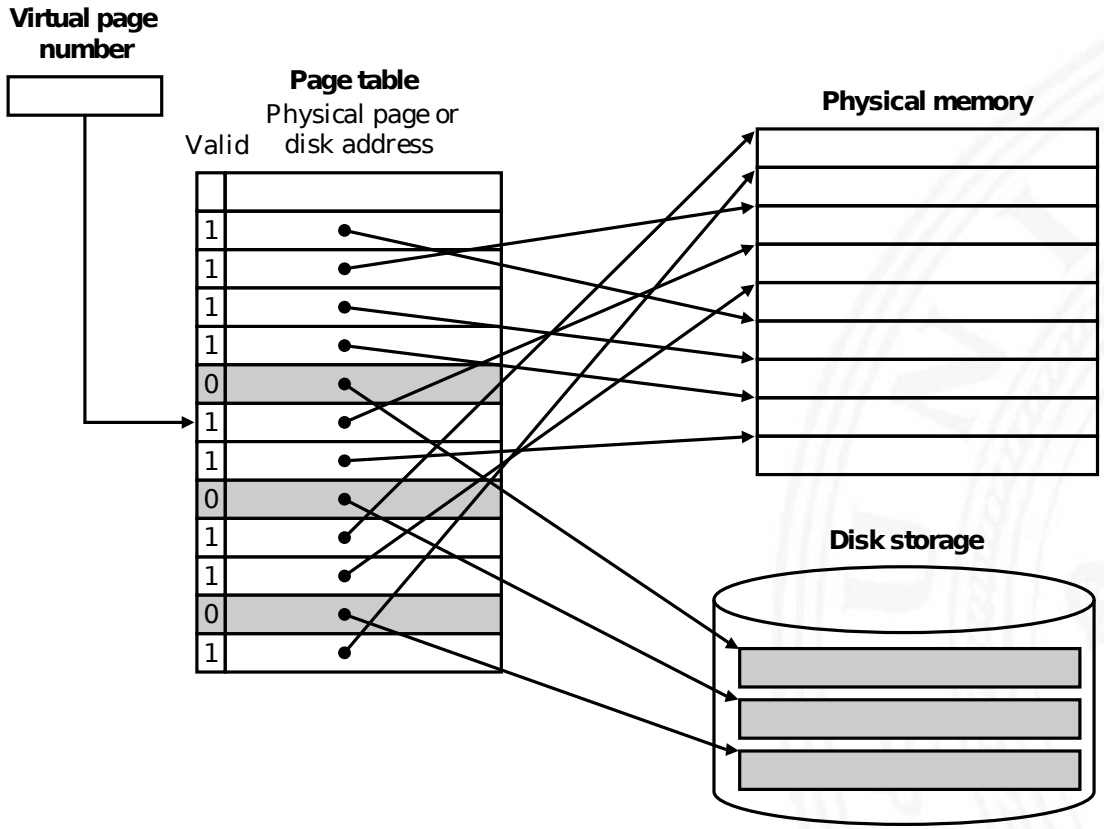
$-$ Segmente könne sehr groß werden

$-$ Ein- und Auslagern von Segmenten kann sehr lange dauern

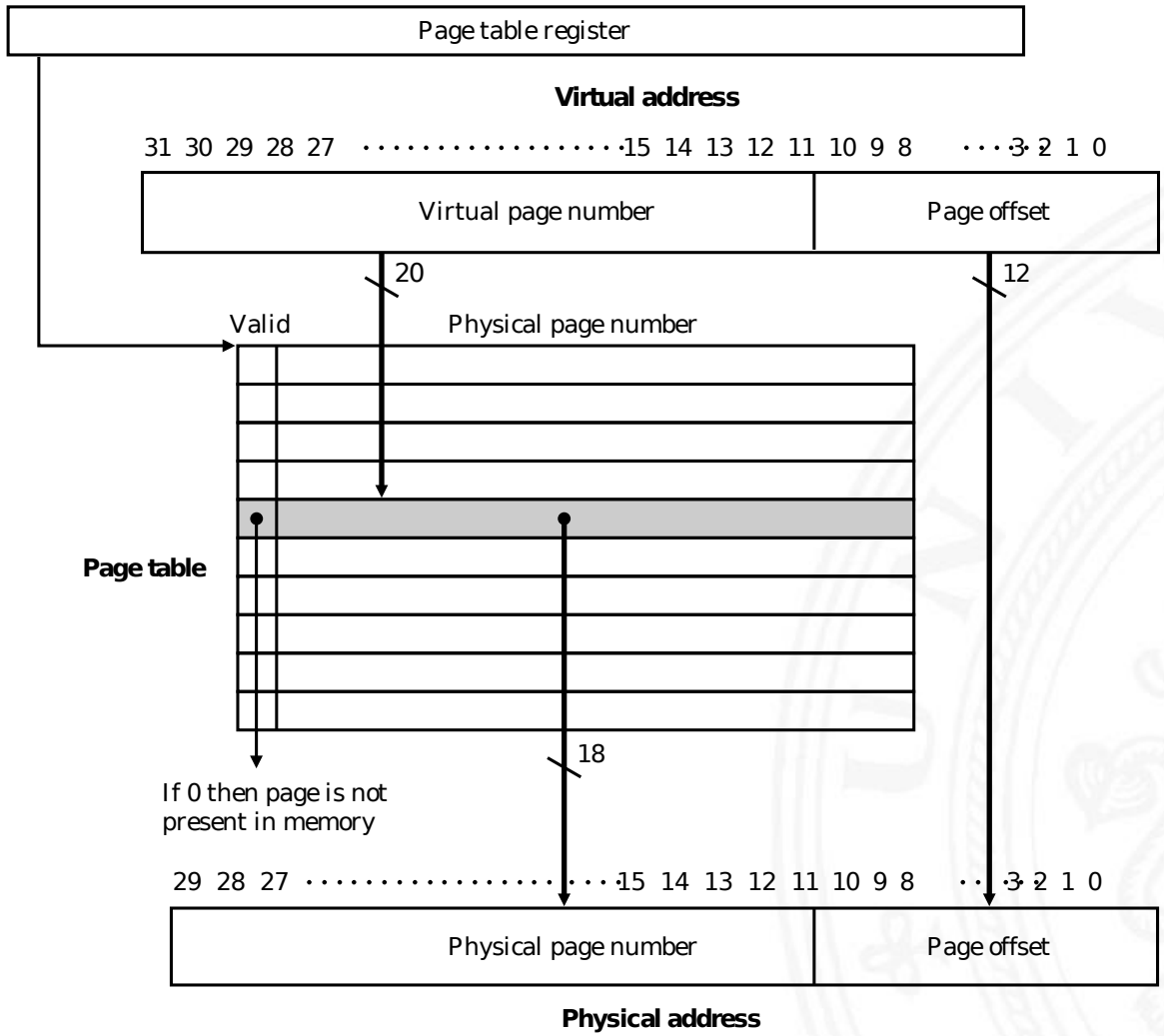
$-$ „Verschnitt“, **Memory Fragmentation**

Virtueller Speicher: Paging / Seitenadressierung

- Unterteilung des Adressraums in Blöcke *fester* Größe = Seiten
  Abbildung auf Hauptspeicherblöcke = Kacheln



Virtueller Speicher: Paging / Seitenadressierung (cont.)



Virtueller Speicher: Paging / Seitenadressierung (cont.)

⇒ Abbildung von *Adressen* in den *virtuellen Speicher*

\+ Programme können größer als der Hauptspeicher sein

\+ Programme können an beliebige physikalischen Adressen geladen werden, unabhängig von der Aufteilung des physikalischen Speichers

\+ feste Seitengröße: einfache Verwaltung in Hardware

\+ Zugriffsrechte für jede Seite (read/write, User/Supervisor)

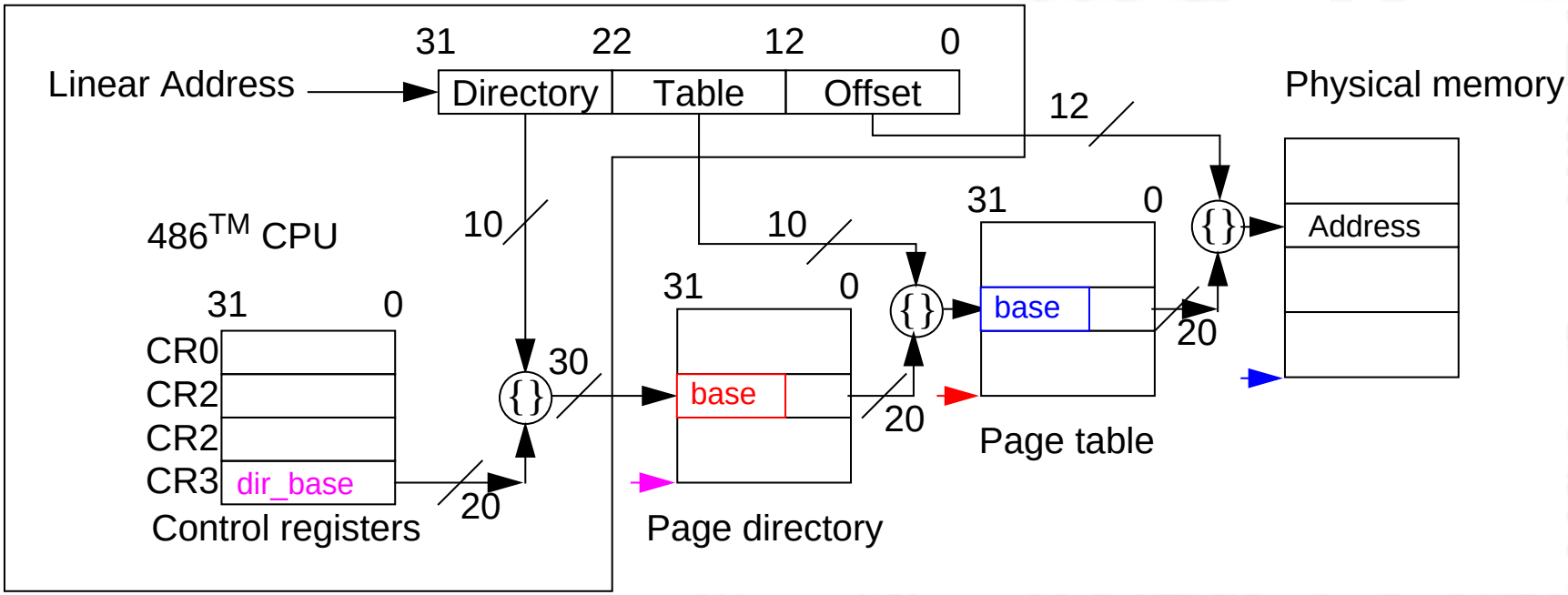
▸ große Miss-Penalty (Nachladen von der Platte)
⇒ Seiten sollten relativ groß sein: 4 oder 8 kByte

– Speicherplatzbedarf der Seitentabelle
viel virtueller Speicher, 4 kByte Seitengröße
⇒ große Pagetable

Virtueller Speicher: Paging / Seitenadressierung (cont.)

⇒ Hash-Verfahren (*inverted page tables*)

⇒ mehrstufige Verfahren

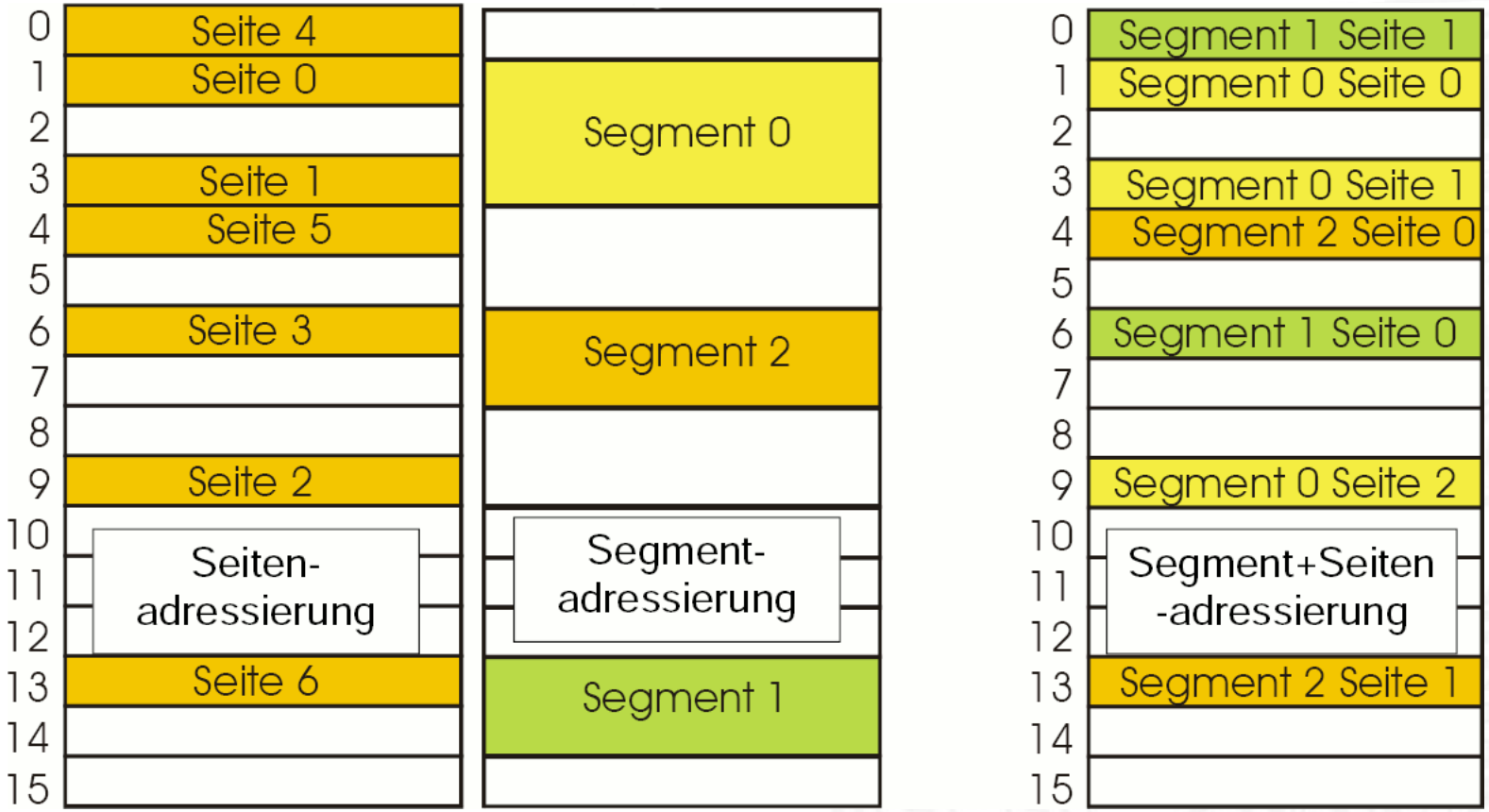


Virtueller Speicher: Paging / Seitenadressierung (cont.)

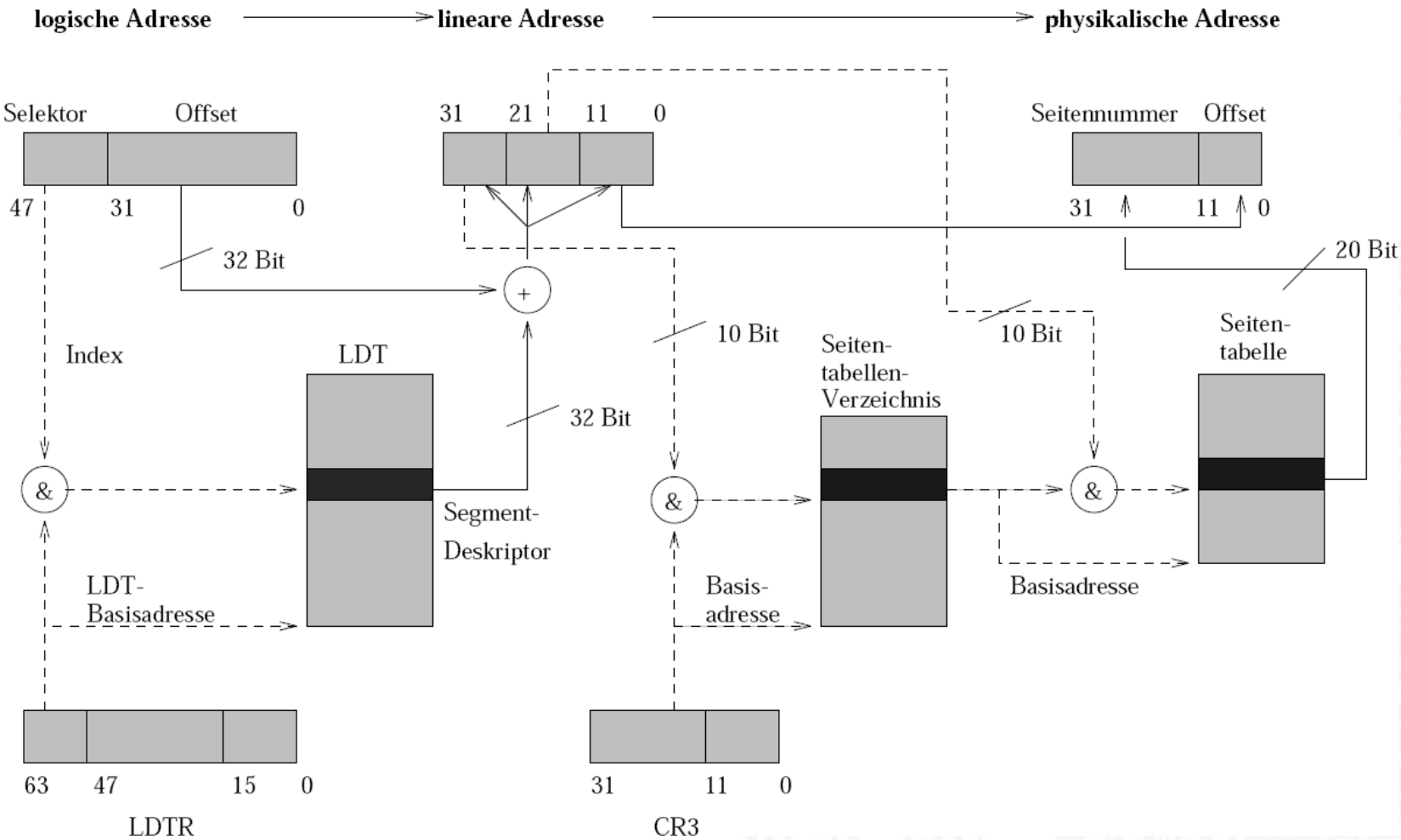
- Schreibstrategie
  - Write-Back (Write-Through wegen Plattenzugriffs zu langsam)
- Ersetzungsstrategie
  - Not recently used
  - Least recently used
  - FIFO...
- Behandlung von Seitenfehlern („*Page-Faults*“) in Software, durch Betriebssystem
  1. Umschalten auf anderen laufbereiten Prozess
  2. Daten in freie Page im Hauptspeicher laden (DMA)
  3. Interrupt wenn DMA fertig ist
  4. Interrupt-Handler: Page-Tabelle des Prozesses updaten
     - beim Zurückschalten auf den ursprünglichen Prozess sind die Daten dann vorhanden

Virtueller Speicher: Segmentierung + Paging

Gemischte Verfahren: Segmentierung und Paging (Beispiel: I386)



Virtueller Speicher: Segmentierung + Paging (cont.)



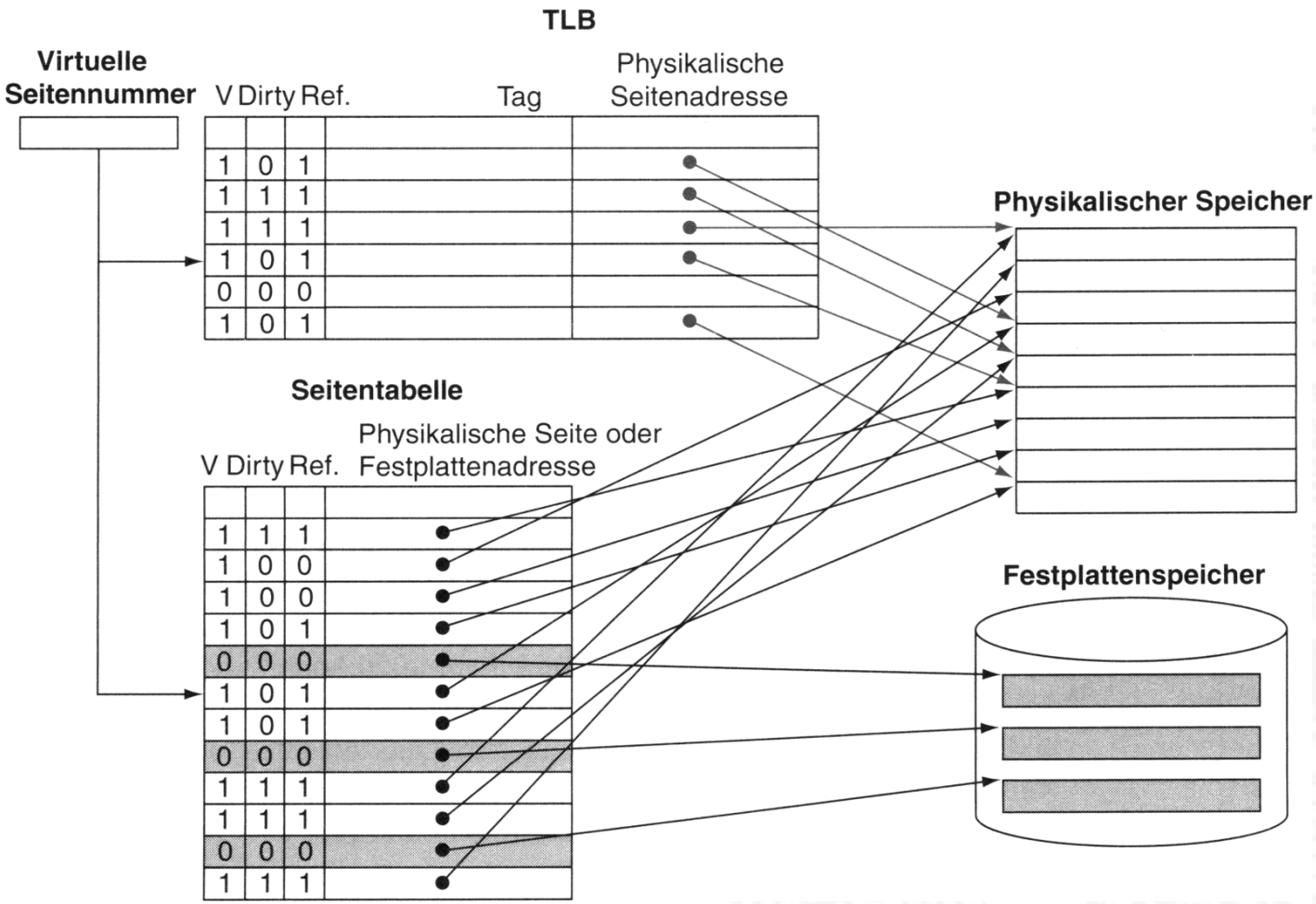
Virtueller Speicher: Segmentierung + Paging (cont.)

- Problem der Adressübersetzung
  - mehrstufige Verfahren
    Beispiel: Segmentierung + 2-stufige Seitenadressierung
  - die jeweiligen Tabellen sind selber im Speicher abgelegt
  - – mehrere Speicherzugriffe

⇒ TLB – **T**ranslation **L**ookaside **B**uffer

- Cache für Seitentabelle

Virtueller Speicher: Segmentierung + Paging (cont.)



Literaturliste

[PH11] David A. Patterson, John L. Hennessy:
Rechnerorganisation und -entwurf – Die Hardware/Software-Schnittstelle.
Oldenbourg; München, 2011.
ISBN 978–3–486–59190–3

[PH12] David A. Patterson, John L. Hennessy:
Computer Organization and Design – The Hardware/Software Interface.
Morgan Kaufmann Publishers Inc.; San Mateo, CA, 2012.
ISBN 978–0–12–374750–1

Literaturliste (cont.)

[HP11] John L. Hennessy, David A. Patterson:
Computer architecture – A quantitative approach.
Morgan Kaufmann Publishers Inc.; San Mateo, CA, 2011.
ISBN 978–0–12–383872–8

[Tan06] Andrew S. Tanenbaum:
Computerarchitektur – Strukturen, Konzepte, Grundlagen.
Pearson Studium; München, Boston..., 2006.
ISBN 3–8273–7151–1

Literaturliste (cont.)

[Mär03] Christian Märtin:
Einführung in die Rechnerarchitektur – Prozessoren und Systeme.
Fachbuchverlag Leipzig im Carl Hanser Verlag; Leipzig, 2003.
ISBN 3–446–22242–1

[OV06] Walter Oberschelp, Gottfried Vossen:
Rechneraufbau und Rechnerstrukturen.
Oldenbourg; München, 2006.
ISBN 3–486–57849–9