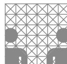
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MIN-Fakultät
Fachbereich Informatik



Rechnerarchitektur: ISA / Pipelining / Speicher

https://tams.informatik.uni-hamburg.de

Andreas Mäder

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Technische Aspekte Multimodaler Systeme

November 2020

Die folgenden Folien sind ein Auszug aus den Unterlagen der Vorlesung 64-613 Rechnerarchitekturen und Mikrosystemtechnik vom Wintersemester 2011/2012 und 64-040 Rechnerstrukturen und Betriebssysteme vom WS 2019/2020.

Das komplette Material findet sich auf den Web-Seiten unter
https://tams.informatik.uni-hamburg.de/lectures/2011ws/vorlesung/ram, bzw.
https://tams.informatik.uni-hamburg.de/lectures/2019ws/vorlesung/rsb

Literaturübersicht

Quellen – besonders diese

- D. Patterson, J. Hennessy: *Computer Organization and Design – The Hardware/Software Interface – MIPS Edition* [PH20]
- J. Hennessy, D. Patterson: *Computer architecture – A quantitative approach* [HP17]
- A. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner* [TA14]
- W. Stallings: *Computer Organization and Architecture – Designing for Performance* [Sta19]
- `https://de.wikipedia.org` und `https://en.wikipedia.org`

Gliederung

1. Rechnerarchitektur
   - Motivation
   - von-Neumann Architektur
   - Wie arbeitet ein Rechner?
2. Bewertung von Architekturen und Rechnersystemen
3. Instruction Set Architecture
4. Pipelining
5. Speicherhierarchie

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

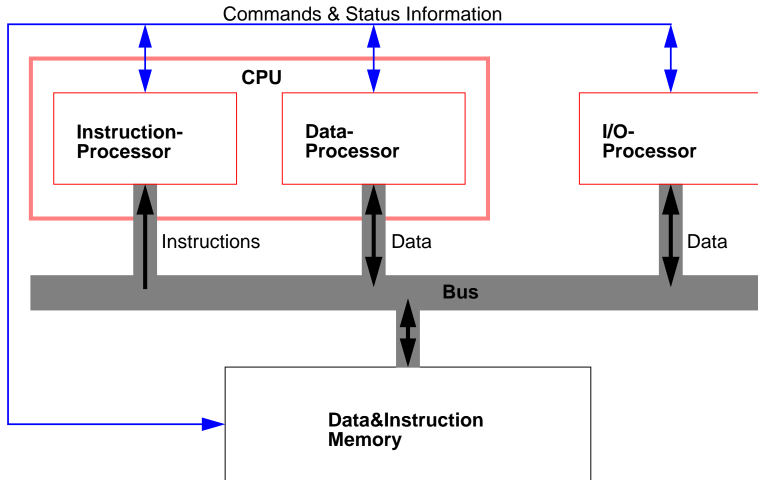
1. Operationsprinzip:
   das funktionelle Verhalten der Architektur — Befehlssatz
   - = Programmierschnittstelle
   - = ISA – **I**nstruction **S**et **A**rchitecture
   - = Maschinenorganisation: *Wie werden Befehle abgearbeitet?*

2. Hardwarearchitektur:
   der strukturelle Aufbau des Rechnersystems — Mikroarchitektur
   - = Art und Anzahl der Hardware-Betriebsmittel + die Verbindungs- / Kommunikationseinrichtungen
   - = (technische) Implementierung

von-Neumann Architektur

- J. Mauchly, J.P. Eckert, J. von-Neumann 1945
- abstrakte Maschine mit minimalem Hardwareaufwand
- die Struktur ist unabhängig von dem Problem, das durch Speicherinhalt (Programm) beschrieben wird
- Hardwarekomponenten
  - zentrale Recheneinheit: CPU, (logisch) unterteilt in
    1. Datenprozessor / Rechenwerk / Operationswerk
    2. Befehlsprozessor / Leitwerk / Steuerwerk
  - gemeinsamer Speicher für Programme und Daten
    - fortlaufend adressiert
    - Programme können wie Daten manipuliert werden
    - Daten können als Programm ausgeführt werden
  - Ein/Ausgabe-Einheit zur Anbindung peripherer Geräte
  - Bussystem(e) verbinden diese Komponenten

von-Neumann Architektur (cont.)

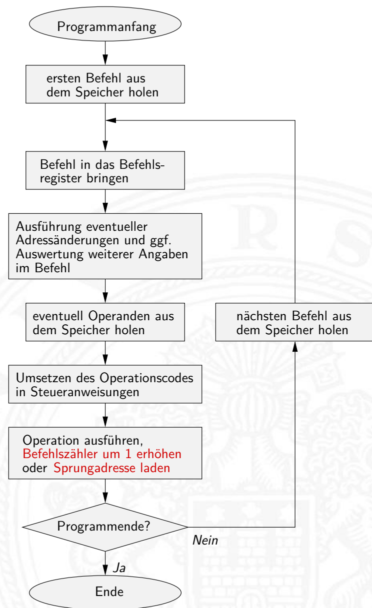


von-Neumann Architektur (cont.)

- Prozessor (CPU) = Steuerwerk + Operationswerk
- Steuerwerk: zwei zentrale Register
  - Befehlszähler (*program counter PC*)
  - Befehlsregister (*instruction register IR*)
- Operationswerk (Datenpfad, *data-path*)
  - Rechenwerk (*arithmetic-logic unit ALU*)
  - Universalregister (mind. 1 *Akkumulator*, typisch 8..64 Register)
  - evtl. Register mit Spezialaufgaben
- Speicher (*memory*)
  - Hauptspeicher/RAM: *random-access memory*
  - Hauptspeicher/ROM: *read-only memory* zum Booten
  - Externspeicher: Festplatten, CD/DVD, Magnetbänder
- Peripheriegeräte (Eingabe/Ausgabe, *I/O*)

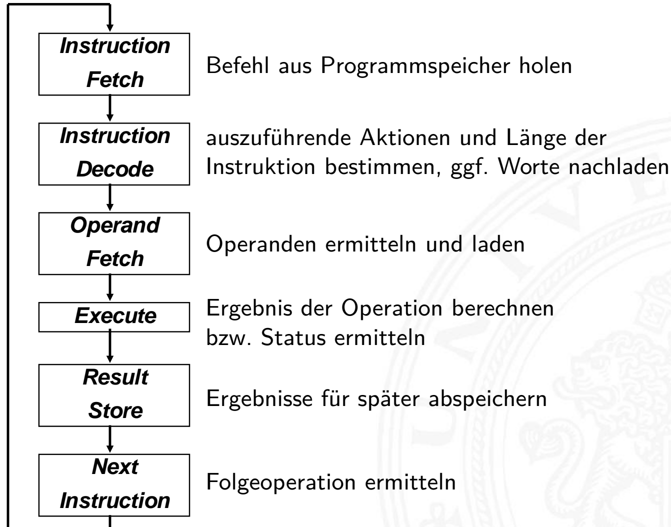
Programmverarbeitung

- von-Neumann Konzept
  - 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



Programmverarbeitung (cont.)

- Ausführungszyklus



Entwicklung der Prozessorarchitekturen

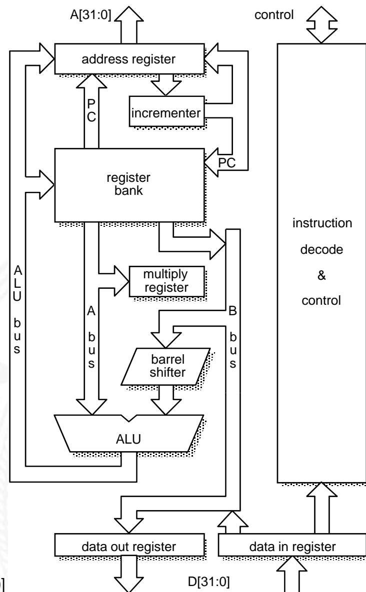
- „von-Neumann Flaschenhals“: Zugriff auf Speicher

⇒ Erweiterungen aktueller Systeme

  - parallele, statt sequenzieller Befehlsabarbeitung
    ⇒ *Pipelining*
  - mehrere Ausführungseinheiten
    ⇒ *superskalare Prozessoren, Mehrkern-Architekturen*
  - dynamisch veränderte Abarbeitungsreihenfolge
    ⇒ *„out-of-order execution“*
  - getrennte Daten- und Instruktionsspeicher
    ⇒ *Harvard-Architektur*
  - *Speicherhierarchie*, Caches etc.

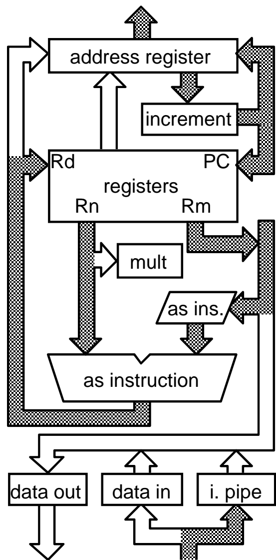
Kompletter Prozessor: ARM 3

- Registerbank (inkl. Program Counter)
- Inkrementer
- Adress-Register
- ALU, Multiplizierer, Shifter
- Speicherinterface (Data-In / -Out)
- Steuerwerk
- bis ARM 7: 3-stufige Pipeline
  fetch, *decode*, *execute*

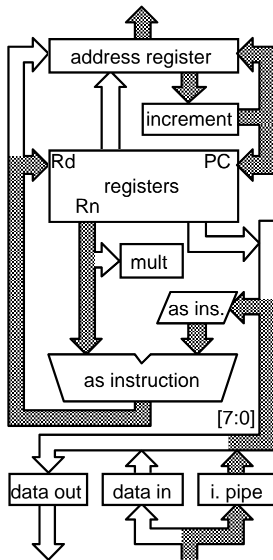


[Fur00]

ARM Datentransfer: Register-Operationen



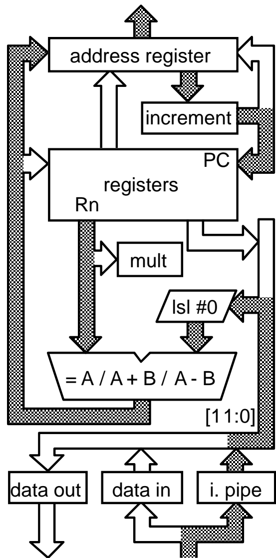
(a) register – register operations



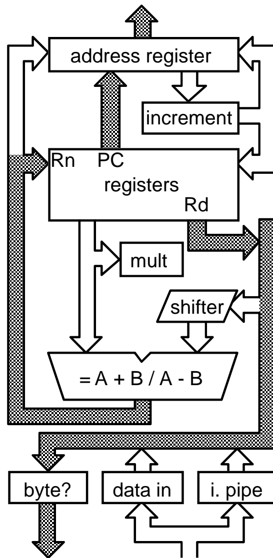
(b) register – immediate operations

[Fur00]

ARM Datentransfer: Store-Befehl



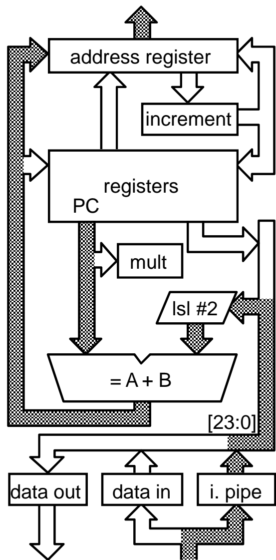
(a) 1st cycle - compute address



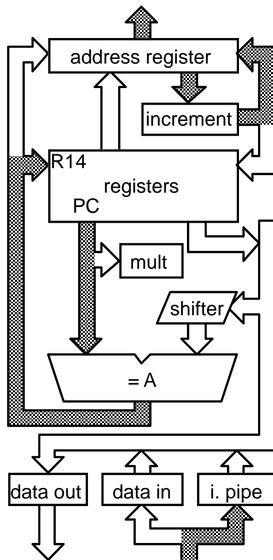
(b) 2nd cycle – store & auto-index

[Fur00]

ARM Datentransfer: Funktionsaufruf/Sprungbefehl



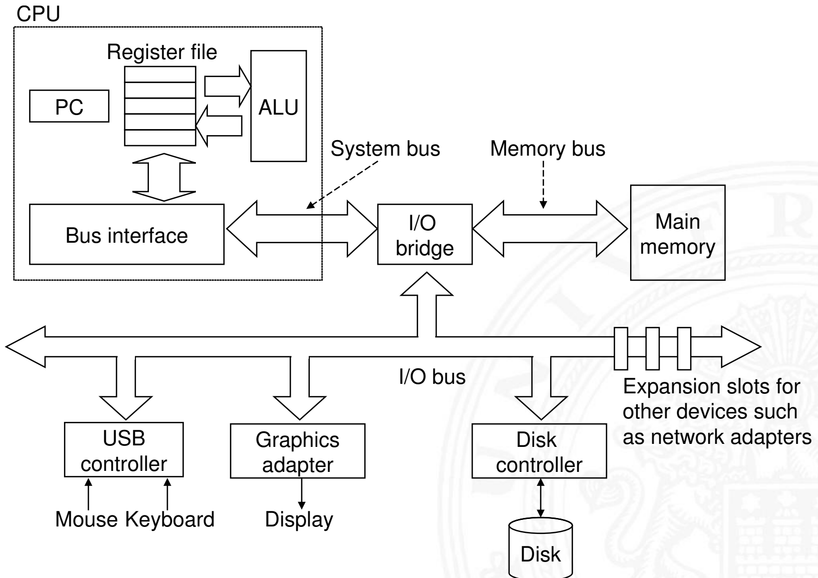
(a) 1st cycle - compute branch target



(b) 2nd cycle - save return address

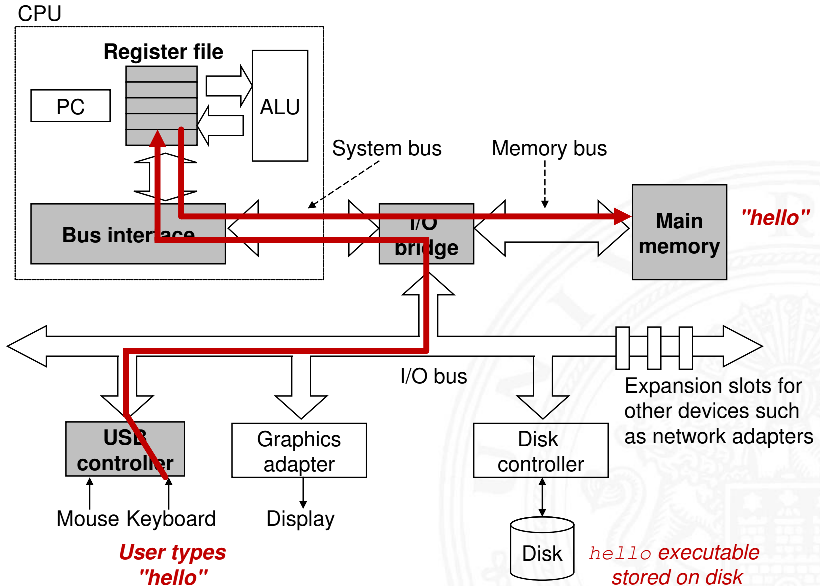
[Fur00]

Hardwareorganisation eines typischen Systems



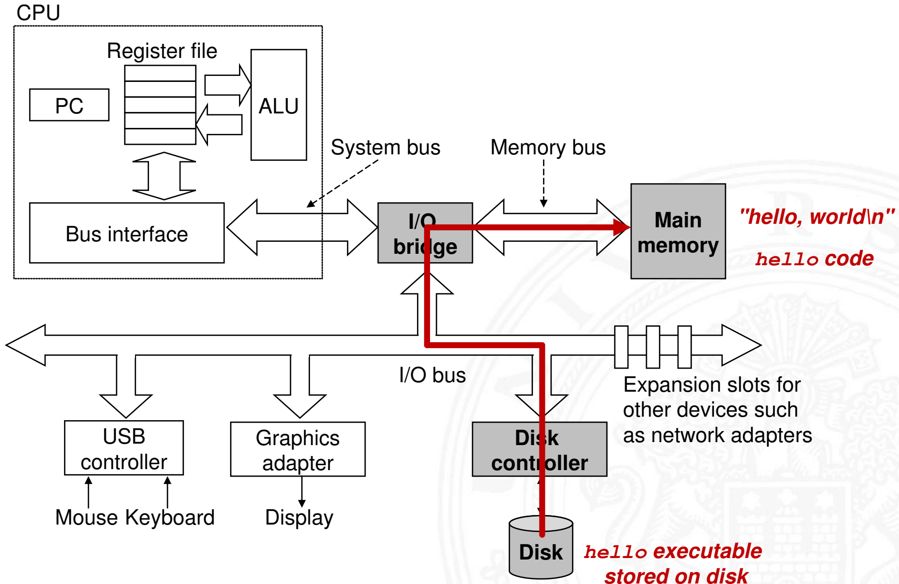
[BO15]

Programmausführung: 1. Benutzereingabe



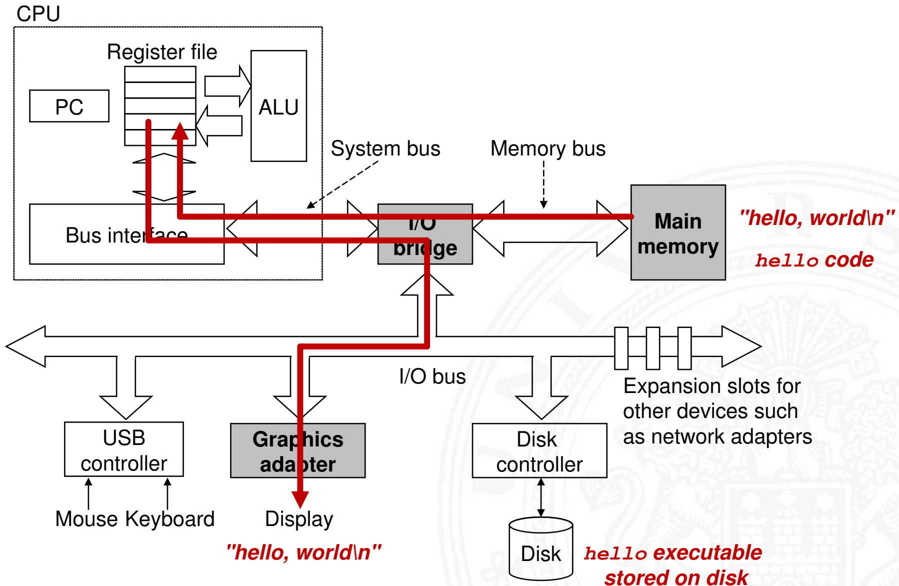
[BO15]

Programmausführung: 2. Programm laden



[BO15]

Programmausführung: 3. Programmlauf

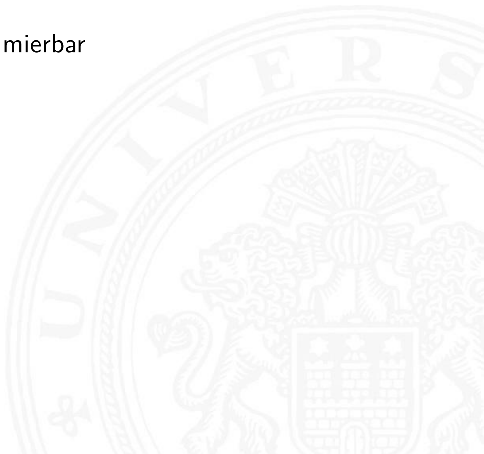


Gliederung

1. Rechnerarchitektur
2. Bewertung von Architekturen und Rechnersystemen
   - Entwurfskriterien
   - Architekturbewertung
   - Kenngrößen
   - Amdahl's Gesetz
3. Instruction Set Architecture
4. Pipelining
5. Speicherhierarchie

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.)

Begriffe, gelten für die *Mikroarchitektur* (Hardwarekomponenten) und den *Befehlssatz* (ISA)

Skalierbarkeit Zusätzliche Hardware/Befehle verbessert das System
$\Rightarrow$ Erweiterbarkeit, Performanz, Wirtschaftlichkeit

Orthogonalität Jedes Modul/jeder Befehl hat eine definierte Funktionalität; keine zwei gleichartigen Module/Befehle
$\Rightarrow$ Wartbarkeit, Kosten, Handhabbarkeit

Adäquatheit Kosten eines Moduls/Befehls entsprechen dessen Nutzen bzw. Funktionalität
$\Rightarrow$ Kosten, Performanz

Virtualität und **Transparenz** Virtuelle Hardware/ISA eliminiert physikalische Grenzen, (unwichtige) Details werden verborgen
$\Rightarrow$ skalierbar, Zuverlässigkeit, einfache Programmierung

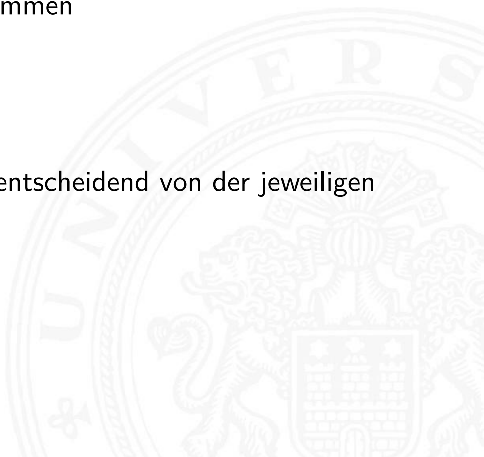
Fehlertransparenz System verbirgt, maskiert oder toleriert Fehler
$\Rightarrow$ Zuverlässigkeit

Bewertung von Architekturen

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



Bewertung von Architekturen (cont.)

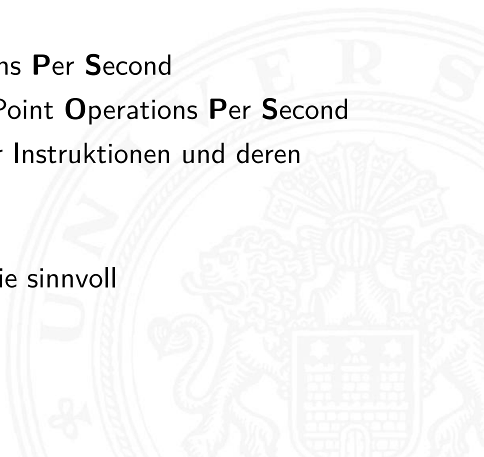
Verfahren zur Bestimmung der Kenngrößen

- *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
- Theoretische Verfahren: analytische Modelle, Simulation ...

Kenngrößen



Taktfrequenz

- Seit Jahren 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 Instruktion	**C**ycles **p**er **I**nstruction
T	Taktperiode	

+ IC kleiner: weniger Instruktionen
  - bessere Algorithmen
  - bessere Compiler
  - mächtigere Befehle (CISC)
+ CPI kleiner: weniger Takte pro Instruktion
  - einfachere Befehle (RISC)
  - parallel Befehle ausführen: VLIW ...
  - parallel Teile der Befehle bearbeiten: Pipelining, superskalare Architekturen ...

Kenngrößen (cont.)

+ T kleiner: höhere Taktfrequenz
  - bessere Technologie
- genauer, wenn CPI über die Häufigkeiten und Zyklenanzahl einzelner Befehle berechnet wird
- so lassen sich beispielsweise alternative Befehlssätze miteinander vergleichen

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?*

⇒ Latenzzeit: *Wie lange dauert es, bis mein Job bearbeitet wird?*

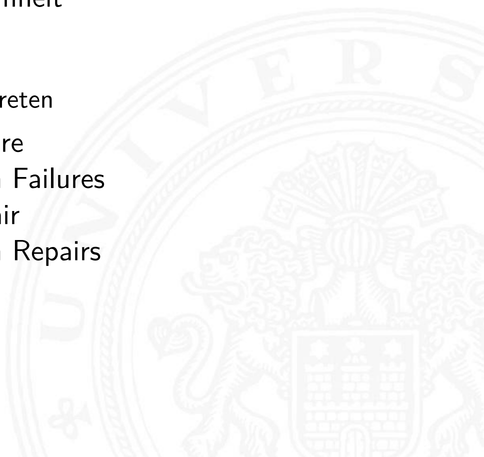
⇒ Antwortzeit: *Wie lange rechnet mein Job?*

- 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
- ...



Speedup

Wie wirken sich Verbesserungen der Rechnerarchitektur aus?

- Speedup: Verhältnis von Ausführungszeiten T *vor* und *nach* der Verbesserung

$$\text{Speedup} = T_{\mathbf{v}orVerbesserung} / T_{\mathbf{n}achVerbesserung}$$

- Teile der Berechnung ($0 \leq \beta \leq 1$) werden um Faktor F beschleunigt:

$$T = T_{\mathbf{o}hneEffekt} + T_{\mathbf{m}itEffekt}$$

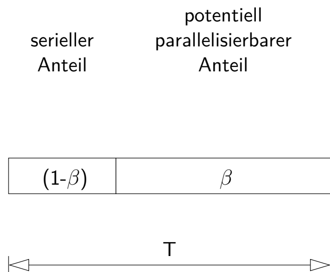
$$T_n = T_{v,o} + T_{v,m} / F_{Verbesserung}$$

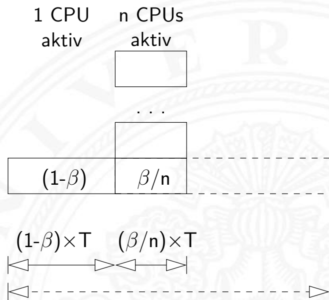
⇒ möglichst großer Anteil β

⇒ den „Normalfall", den häufigsten Fall beschleunigen, um den größten Speedup zu erreichen

Amdahl's Gesetz

- Gene Amdahl, Architekt der IBM S/360, 1967
- ursprüngliche Idee
  - Parallelrechner mit n-Prozessoren $(= F)$
  - Parallelisierung der Aufgabe, bzw. einer Teilaufgabe





Amdahl's Gesetz (cont.)

- Amdahl's Gesetz

  Speedup $= \frac{1}{(1-\beta)+f_k(n)+\beta/n} \leq \frac{1}{(1-\beta)}$, mit $\beta = [0,1]$

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

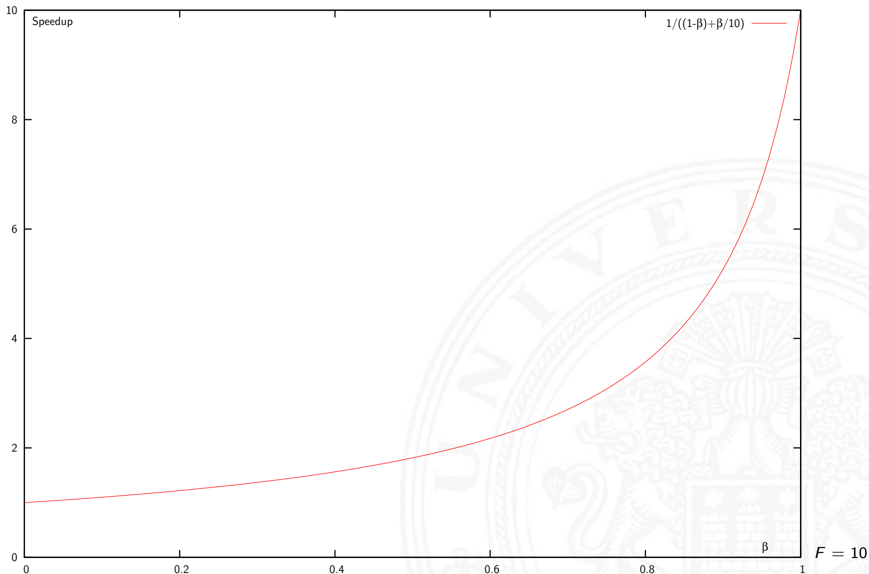
  - Aufgaben verteilen
  - Arbeit koordinieren
  - Ergebnisse zusammensammeln

Amdahl's Gesetz: Beispiele

n	β	*Speedup*		
10	0,1	$1/(0{,}9 + 0{,}01)$	$=$	1,1
2	0,5	$1/(0{,}5 + 0{,}25)$	$=$	1,33
2	0,9	$1/(0{,}1 + 0{,}45)$	$=$	1,82
1,1	0,98	$1/(0{,}02 + 0{,}89)$	$=$	1,1
4	0,5	$1/(0{,}5 + 0{,}125)$	$=$	1,6
4 536	0,8	$1/(0{,}2 + 0{,}0\ldots)$	$=$	5,0
9 072	0,99	$1/(0{,}01 + 0{,}0\ldots)$	$=$	98,92

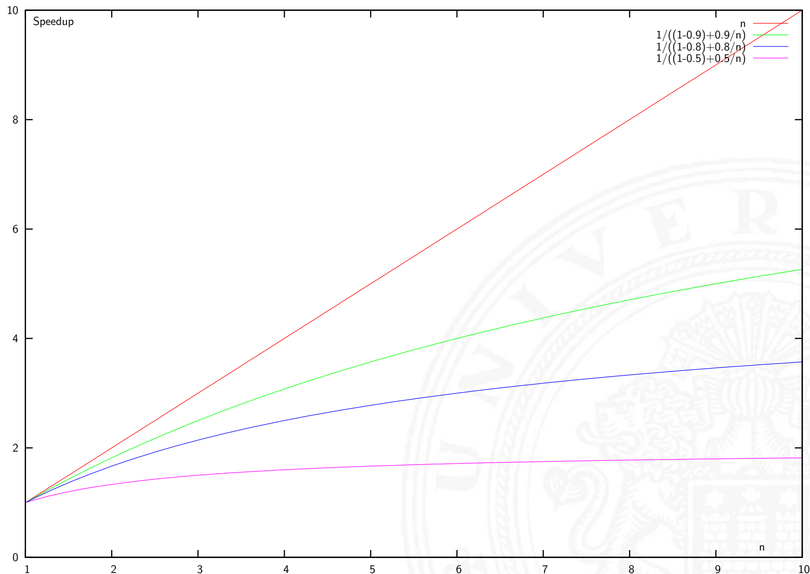
- Optimierung bringt nichts, wenn der nicht beschleunigte „serielle" Anteil $(1 - \beta)$ eines Programms überwiegt
- n-Prozessoren (große F) wirken *nicht linear*
- Parallelität in Hochsprachen-Programmen ist meist gering, typischerweise: *Speedup* ≤ 4
  Multitasking und mehrere Prozesse: große β

Amdahl's Gesetz: Beispiele (cont.)

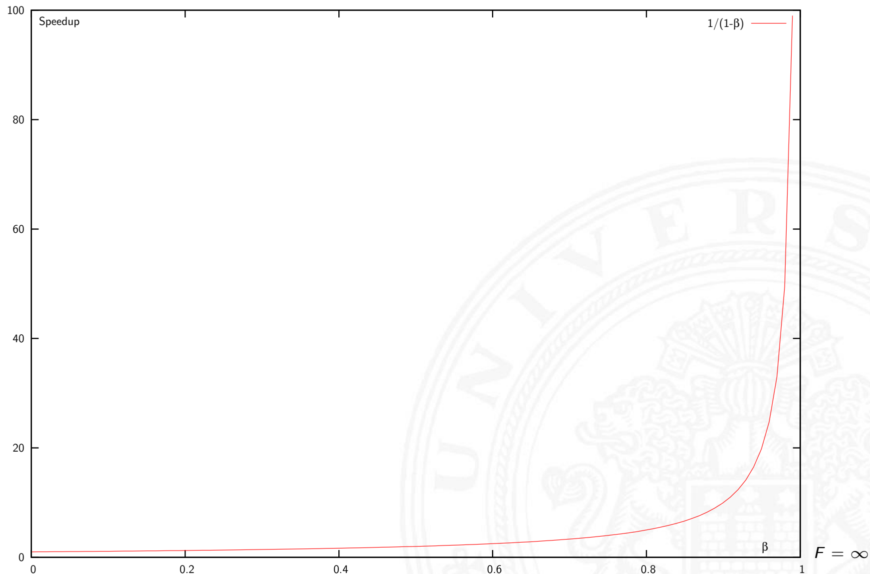


$F = 10$

Amdahl's Gesetz: Beispiele (cont.)



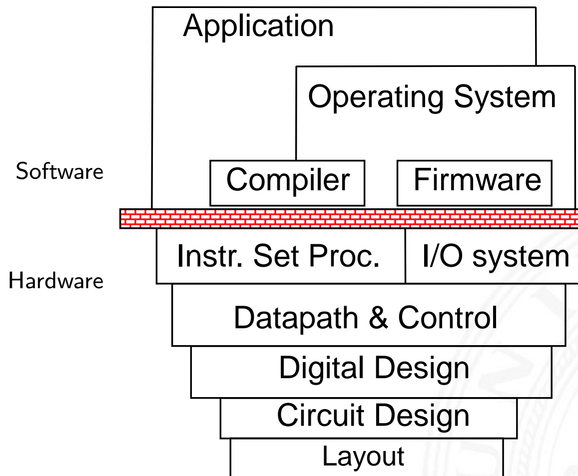
Amdahl's Gesetz: Beispiele (cont.)



Gliederung

1. Rechnerarchitektur
2. Bewertung von Architekturen und Rechnersystemen
3. Instruction Set Architecture
   - Speicherorganisation
   - Befehlssatz
   - Befehlsformate
   - Adressierungsarten
   - Befehlssätze
4. Pipelining
5. Speicherhierarchie

Schnittstelle von Hardware und Software

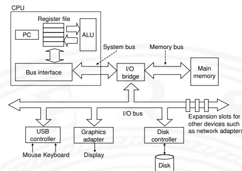


Instruction Set Architecture

[PH20]

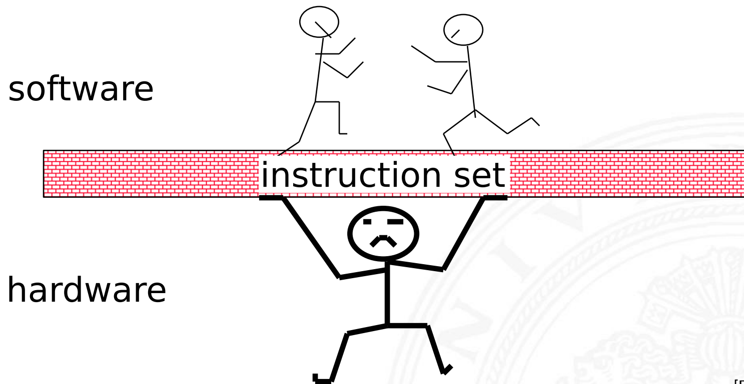
ISA – **I**nstruction **S**et **A**rchitecture

⇒ alle für den Programmierer sichtbaren Attribute eines Rechners



- der (konzeptionellen) Struktur
  - Funktionseinheiten der Hardware: Recheneinheiten, Speicher, Verbindungssysteme
- des Verhaltens
  - Organisation des programmierbaren Speichers
  - Datentypen und Datenstrukturen: Codierungen und Darstellungen
  - Befehlssatz
  - Befehlsformate
  - Modelle für Befehls- und Datenzugriffe
  - Ausnahmebedingungen

- Befehlssatz: die zentrale Schnittstelle



[PH20]

Merkmale der Instruction Set Architecture

▶ Speichermodell	Wortbreite, Adressierung …
▶ Rechnerklasse	Stack-/Akku-/Registermaschine
▶ Registersatz	Anzahl und Art der Rechenregister
▶ Befehlssatz	Definition aller Befehle
▶ Art, Zahl der Operanden	Anzahl/Wortbreite/Reg./Speicher
▶ Ausrichtung der Daten	Alignment/Endianness

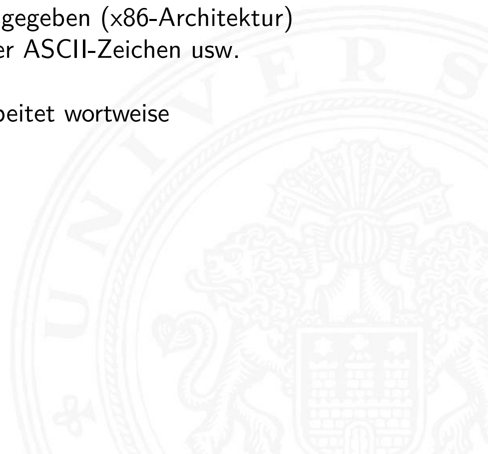
- Ein- und Ausgabe, Unterbrechungsstruktur (Interrupts)

▶ Systemsoftware	Loader, Assembler, Compiler, Debugger

Speicherorganisation

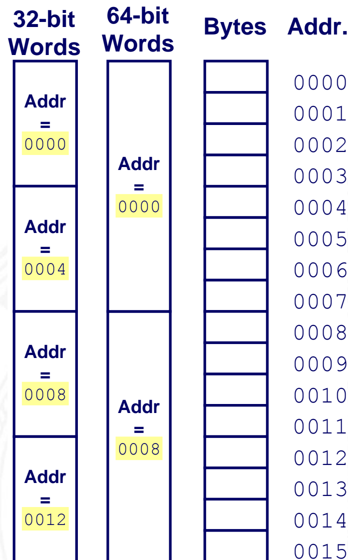
- Datentypen: Abspeichern von Zahlen, Zeichen, Strings?
  - kleinster Datentyp üblicherweise ein Byte (8-bit)
  - andere Daten als Vielfache: 16-bit, 32-bit, 64-bit ...
- Organisation und Adressierung des Speichers?
  - Adressen typisch in Bytes angegeben (x86-Architektur)
  - erlaubt Adressierung einzelner ASCII-Zeichen usw.

  - aber Maschine/Prozessor arbeitet wortweise



Speicherorganisation (cont.)

- Speicher Wort-orientiert
- Adressierung Byte-orientiert
  - die Adresse des ersten Bytes im Wort
  - Adressen aufeinanderfolgender Worte unterscheiden sich um 4 (32-bit Wort) oder 8 (64-bit)
  - Adressen normalerweise Vielfache der Wortlänge
  - verschobene Adressen „in der Mitte“ eines Worts oft unzulässig



[BO15]

Byte-Order

- *Wie sollen die Bytes innerhalb eines Wortes angeordnet werden?*
- Speicher wort-basiert ⇔ Adressierung byte-basiert

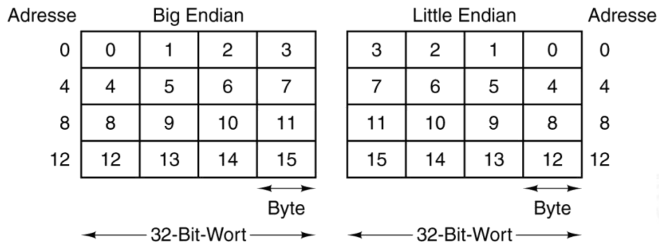
Zwei Möglichkeiten / Konventionen:

- **Big Endian**: Sun, Mac usw.
  das MSB (*most significant byte*) hat die kleinste Adresse
  das LSB (*least significant byte*) hat die höchste –"–

- **Little Endian**: Alpha, x86
  das MSB hat die höchste, das LSB die kleinste Adresse

satirische Referenz auf Gulliver's Reisen (Jonathan Swift)

Big- vs. Little Endian

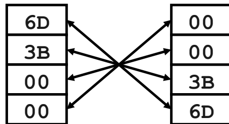


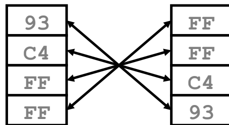
- Anordnung einzelner Bytes in einem Wort (hier 32 bit)
  - Big Endian ($n \dots n+3$): MSB ... LSB „String“-Reihenfolge
  - Little Endian ($n \dots n+3$): LSB ... MSB „Zahlen“-Reihenfolge
- beide Varianten haben Vor- und Nachteile
- ggf. Umrechnung zwischen beiden Systemen notwendig

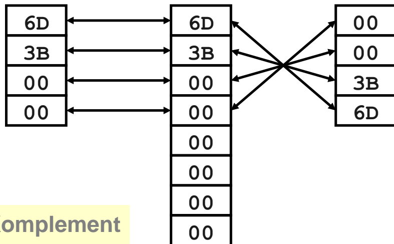
Byte-Order: Beispiel


```
     int A =  15213;
     int B = -15213;
long int C =  15213;
```

Dezimal:	15213			
Binär:	0011	1011	0110	1101
Hex:	3	B	6	D







2-Komplement
Big Endian
Little Endian

[BO15]

Byte-Order: Beispiel Datenstruktur


```
/* JimSmith.c - example record for byte-order demo */

typedef struct employee {
  int      age;
  int   salary;
  char    name[12];
} employee_t;

static employee_t jimmy = {
  23,                   // 0x0017
  50000,                // 0xc350
  "Jim Smith",          // J=0x4a i=0x69 usw.
};
```

Byte-Order: Beispiel x86 und SPARC

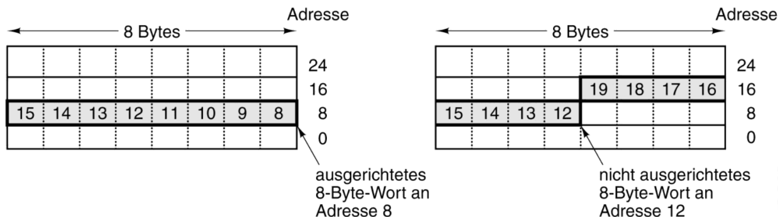

```
tams12> objdump -s JimSmith.x86.o
JimSmith.x86.o:     file format elf32-i386

Contents of section .data:
 0000 17000000 50c30000 4a696d20 536d6974  ....P...Jim Smit
 0010 68000000                             h...

tams12> objdump -s JimSmith.sparc.o
JimSmith.sparc.o:     file format elf32-sparc

Contents of section .data:
 0000 00000017 0000c350 4a696d20 536d6974  .......PJim Smit
 0010 68000000                             h...
```

„Misaligned“ Zugriff



[TA14]

- Beispiel: 8-Byte-Wort in Little Endian Speicher
  - „*aligned*“ bezüglich Speicherwort
  - „*non aligned*“ an Byte-Adresse 12
- Speicher wird (meistens) Byte-weise adressiert
  aber Zugriffe lesen/schreiben jeweils ein ganzes Wort

$\Rightarrow$ was passiert bei „krummen“ (*misaligned*) Adressen?

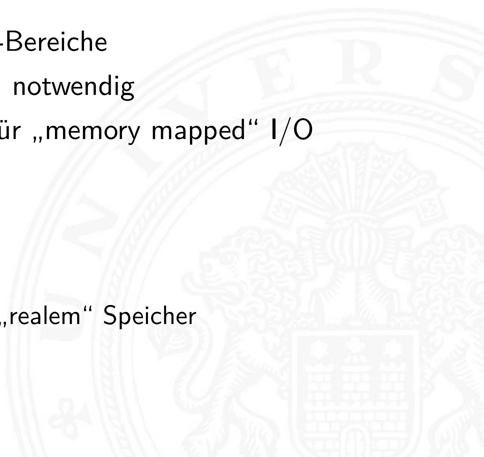
- automatische Umsetzung auf mehrere Zugriffe (x86)
- Programmabbruch (SPARC)

Memory Map

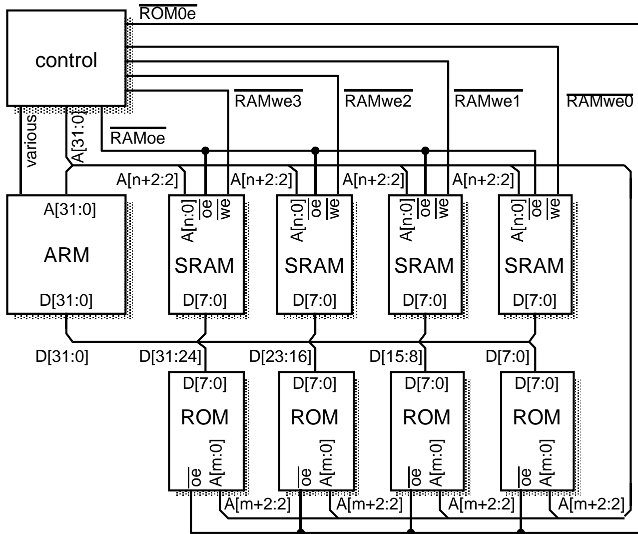
- CPU kann im Prinzip alle möglichen Adressen ansprechen
- in der Regel: kein voll ausgebauter Speicher
  32 bit Adresse entsprechen 4 GiB Hauptspeicher, 64 bit ...

- Aufteilung in RAM und ROM-Bereiche
- ROM mindestens zum Booten notwendig
- zusätzliche Speicherbereiche für „memory mapped" I/O

⇒ „Memory Map"
  - Adressdecoder
  - Hardwareeinheit
  - Zuordnung von Adressen zu „realem" Speicher



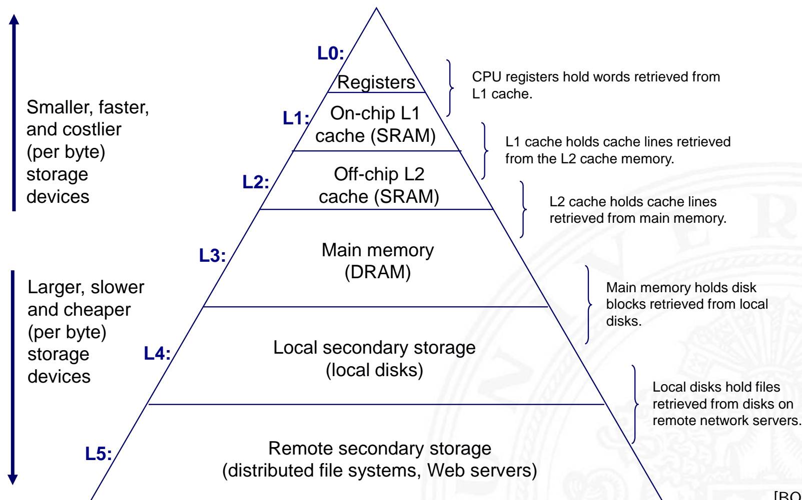
Memory Map (cont.)



32-bit ARM Proz.
4 × 8-bit SRAMs
4 × 8-bit ROMs

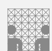
[Fur00]

Speicherhierarchie



[BO15]

später mehr . . .



ISA-Merkmale des Prozessors

- Befehlszyklus
- Befehlsklassen
- Registermodell
- n-Adress Maschine
- Adressierungsarten

Befehlszyklus

- Prämisse: von-Neumann Prinzip
  - Daten und Befehle im gemeinsamen Hauptspeicher
- Abarbeitung des Befehlszyklus in Endlosschleife
  - Programmzähler PC adressiert den Speicher
  - gelesener Wert kommt in das Befehlsregister IR
  - Befehl decodieren
  - Befehl ausführen
  - nächsten Befehl auswählen
- benötigte Register

Steuerwerk

PC	Program Counter	Adresse des Befehls
IR	Instruction Register	aktueller Befehl

Rechenwerk

R0 ... R31	Registerbank	Rechenregister (Operanden)
ACC	Akkumulator	= Minimalanforderung

Instruction Fetch
„Befehl holen“ Phase im Befehlszyklus

1. Programmzähler (PC) liefert Adresse für den Speicher
2. Lesezugriff auf den Speicher
3. Resultat wird im Befehlsregister (IR) abgelegt
4. Programmzähler wird inkrementiert (ggf. auch später)

- Beispiel für 32 bit RISC mit 32 bit Befehlen
  - $\text{IR} = \text{MEM}[\text{PC}]$
  - $\text{PC} = \text{PC} + 4$
- bei CISC-Maschinen evtl. weitere Zugriffe notwendig, abhängig von der Art (und Länge) des Befehls

Instruction Decode
„Befehl decodieren" Phase im Befehlszyklus

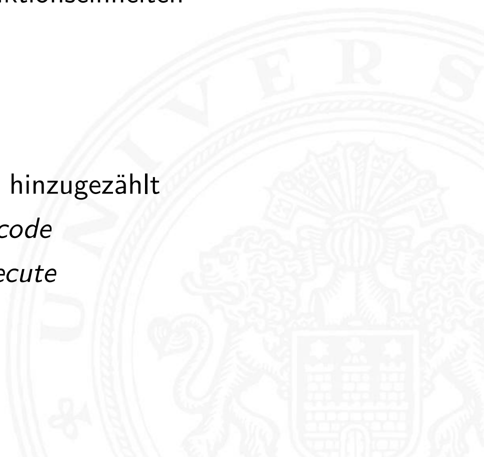
- Befehl steht im Befehlsregister IR

1. Decoder entschlüsselt Opcode und Operanden
2. leitet Steuersignale an die Funktionseinheiten

Operand Fetch

- wird meist zu anderen Phasen hinzugezählt
  RISC: Teil von *Instruction Decode*
  CISC: –"– *Instruction Execute*

1. Operanden holen



Instruction Execute

„Befehl ausführen“ Phase im Befehlszyklus

- ▷ Befehl steht im Befehlsregister IR
- ▷ Decoder hat Opcode und Operanden entschlüsselt
- ▷ Steuersignale liegen an Funktionseinheiten

1. Ausführung des Befehls durch Aktivierung der Funktionseinheiten
2. ggf. Programmzähler setzen/inkrementieren

- Details abhängig von der Art des Befehls
- Ausführungszeit –"–
- Realisierung
  - fest verdrahtete Hardware
  - mikroprogrammiert

Welche Befehle braucht man?

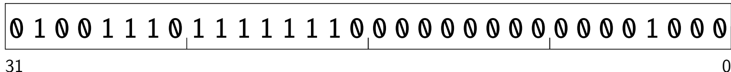
Befehlsklassen	Beispiele
▶ arithmetische Operationen	add, sub, inc, dec, mult, div
logische Operationen	and, or, xor
schiebe Operationen	shl, sra, srl, ror
▶ Vergleichsoperationen	cmpeq, cmpgt, cmplt
▶ Datentransfers	load, store, I/O
▶ Programm-Kontrollfluss	jump, jmpeq, branch, call, return
▶ Maschinensteuerung	trap, halt, (interrupt)

⇒ Befehlssätze und Computerarchitekturen (Details später)

CISC – Complex Instruction Set Computer
RISC – Reduced Instruction Set Computer

Befehls-Decodierung

- Befehlsregister IR enthält den aktuellen Befehl
- z.B. einen 32-bit Wert



Wie soll die Hardware diesen Wert interpretieren?

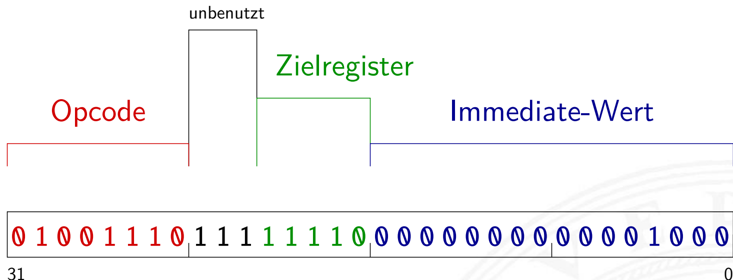
- direkt in einer Tabelle nachschauen (Mikrocode-ROM)
- Problem: Tabelle müsste 2^{32} Einträge haben

⇒ Aufteilung in Felder: Opcode und Operanden

⇒ Decodierung über mehrere, kleine Tabellen

⇒ unterschiedliche Aufteilung für unterschiedliche Befehle: Befehlsformate

Befehlsformate



- Befehlsformat: Aufteilung in mehrere Felder
  - Opcode — eigentlicher Befehl
  - ALU-Operation — add/sub/incr/shift/usw.
  - Register-Indizes — Operanden / Resultat
  - Speicher-Adressen — für Speicherzugriffe
  - Immediate-Operanden — Werte direkt im Befehl
- Lage und Anzahl der Felder abhängig vom Befehlssatz

Befehlsformat: Beispiel MIPS

- festes Befehlsformat
  - alle Befehle sind 32 Bit lang
- Opcode-Feld ist immer 6-bit breit
  - codiert auch verschiedene Adressierungsmodi

nur 3 Befehlsformate: R, I, J

- R-Format
  - Register-Register ALU-Operationen
- I-/J-Format
  - Lade- und Speicheroperationen
  - alle Operationen mit unmittelbaren Operanden
  - Jump-Register
  - Jump-and-Link-Register

MIPS: Übersicht

„Microprocessor without Interlocked Pipeline Stages“

- entwickelt an der Univ. Stanford, seit 1982
- Einsatz: eingebettete Systeme, SGI Workstations/Server

- klassische 32-bit RISC Architektur
- 32-bit Wortbreite, 32-bit Speicher, 32-bit Befehle
- 32 Register: R0 ist konstant Null, R1 . . . R31 Universalregister
- Load-Store Architektur, nur base+offset Adressierung

- sehr einfacher Befehlssatz, 3-Adress-Befehle
- keinerlei HW-Unterstützung für „komplexe“ SW-Konstrukte
- SW muss sogar HW-Konflikte („Hazards“) vermeiden
- Koprozessor-Konzept zur Erweiterung

MIPS: Registermodell

- 32 Register, R0 . . . R31, jeweils 32-bit
- R1 bis R31 sind Universalregister
- R0 ist konstant Null (ignoriert Schreiboperationen)
  - R0 Tricks

R5 = -R5	sub	R5, R0, R5
R4 = 0	add	R4, R0, R0
R3 = 17	addi	R3, R0, 17
if (R2 == 0)	bne	R2, R0, label

- keine separaten Statusflags
- Vergleichsoperationen setzen Zielregister auf 0 bzw. 1

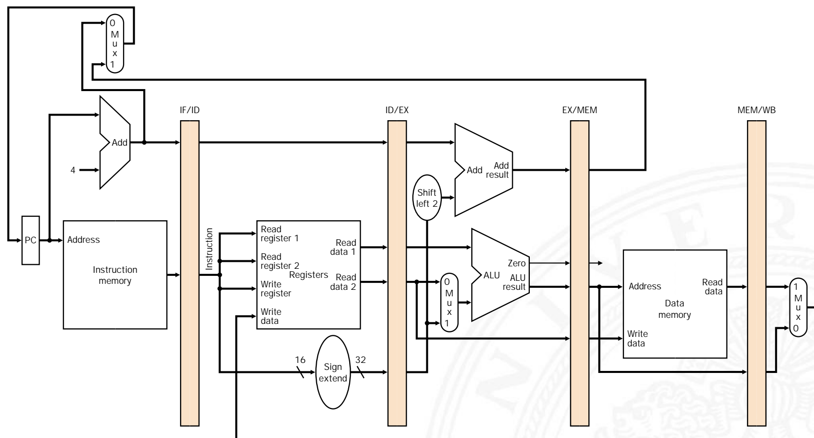
R1 = (R2 < R3)	slt	R1, R2, R3

MIPS: Befehlssatz

- Übersicht und Details: [PH20, PH16]
  David A. Patterson, John L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle*
- dort auch hervorragende Erläuterung der Hardwarestruktur

- klassische fünf-stufige Befehlspipeline
  - Instruction-Fetch — Befehl holen
  - Decode — Decodieren und Operanden holen
  - Execute — ALU-Operation oder Adressberechnung
  - Memory — Speicher lesen oder schreiben
  - Write-Back — Resultat in Register speichern

MIPS: Hardwarestruktur



[PH20]

PC
I-Cache

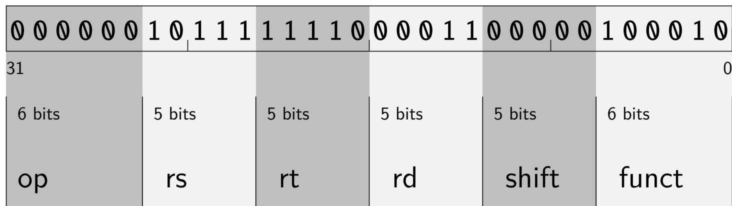
Register
(R0 .. R31)

ALUs

Speicher
D-Cache

MIPS: Befehlsformate

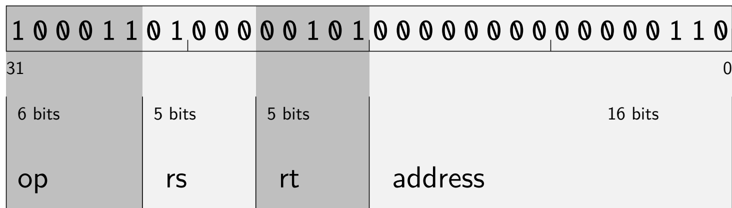
Befehl im R-Format



▶ op:	Opcode	Typ des Befehls	`0` = „alu-op“
rs:	source register 1	erster Operand	`23` = „r23“
rt:	source register 2	zweiter Operand	`30` = „r30“
rd:	destination register	Zielregister	`3` = „r3“
shift:	shift amount	(optionales Shiften)	`0` = „0“
funct:	ALU function	Rechenoperation	`34` = „sub“

⇒ r3 = r23 - r30 `sub r3, r23, r30`

MIPS: Befehlsformate
Befehl im I-Format



- op: Opcode — Typ des Befehls — 35 = „lw“
  rs: base register — Basisadresse — 8 = „r8“
  rt: destination register — Zielregister — 5 = „r5“
  addr: address offset — Offset — 6 = „6“

$\Rightarrow$ r5 = MEM[r8+6] `lw r5, 6(r8)`

Adressierungsarten

- Woher kommen die Operanden / Daten für die Befehle?
  - Hauptspeicher, Universalregister, Spezialregister
- Wie viele Operanden pro Befehl?
  - 0- / 1- / 2- / 3-Adress Maschinen
- Wie werden die Operanden adressiert?
  - immediate / direkt / indirekt / indiziert / autoinkrement / usw.

$\Rightarrow$ wichtige Unterscheidungsmerkmale für Rechnerarchitekturen

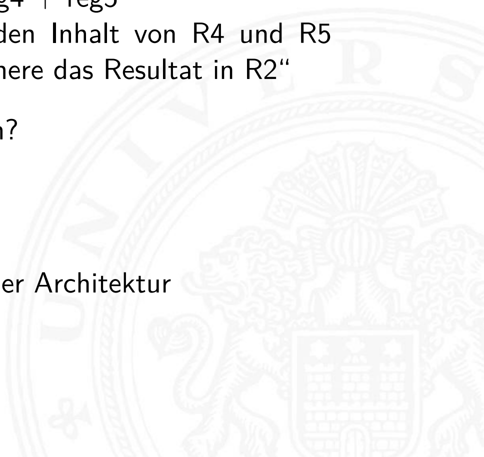
- Zugriff auf Hauptspeicher: $\approx 100 \times$ langsamer als Registerzugriff
  - möglichst Register statt Hauptspeicher verwenden (!)
  - „load/store“-Architekturen

Beispiel: Add-Befehl

- Rechner soll „rechnen“ können
- typische arithmetische Operation nutzt 3 Variablen
  Resultat, zwei Operanden: $X = Y + Z$

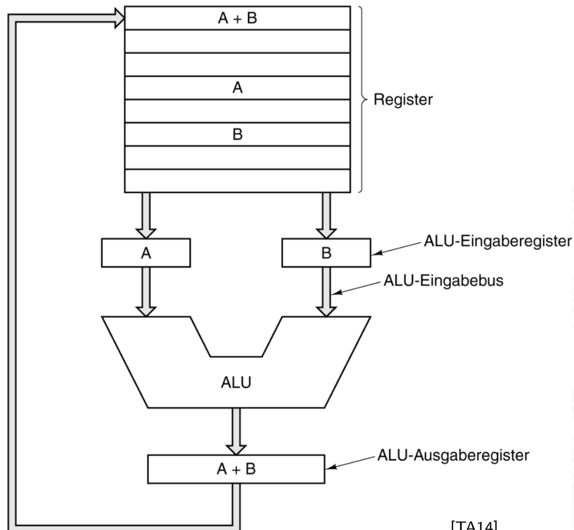
  `add r2, r4, r5` reg2 = reg4 + reg5
  „addiere den Inhalt von R4 und R5 und speichere das Resultat in R2“

- woher kommen die Operanden?
- wo soll das Resultat hin?
  - Speicher
  - Register
- entsprechende Klassifikation der Architektur



Beispiel: Datenpfad

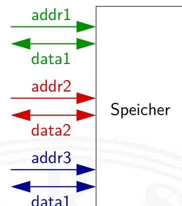
- Register (-bank)
  - liefern Operanden
  - speichern Resultate
- interne Hilfsregister
- ALU, typ. Funktionen:
  - add, add-carry, sub
  - and, or, xor
  - shift, rotate
  - compare
  - (floating point ops.)



[TA14]

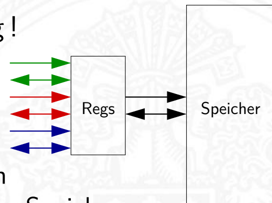
Woher kommen die Operanden?

- typische Architektur
  - von-Neumann Prinzip: alle Daten im Hauptspeicher
  - 3-Adress-Befehle: zwei Operanden, ein Resultat



⇒ „Multiport-Speicher“ mit drei Ports ?
  - sehr aufwändig, extrem teuer, trotzdem langsam

⇒ Register im Prozessor zur Zwischenspeicherung !
  - Datentransfer zwischen Speicher und Registern
    Load reg = MEM[addr]
    Store MEM[addr] = reg
  - RISC: Rechenbefehle arbeiten *nur* mit Registern
  - CISC: gemischt, Operanden in Registern oder im Speicher



n-Adress Maschine $n = \{3 \dots 0\}$

3-Adress Format
- $X = Y + Z$
- sehr flexibel, leicht zu programmieren
- Befehl muss 3 Adressen codieren

2-Adress Format
- $X = X + Z$
- eine Adresse doppelt verwendet:
  für Resultat und einen Operanden
- Format wird häufig verwendet

1-Adress Format
- $ACC = ACC + Z$
- alle Befehle nutzen das Akkumulator-Register
- häufig in älteren / 8-bit Rechnern

0-Adress Format
- $TOS = TOS + NOS$
- Stapelspeicher: *top of stack*, *next of stack*
- Adressverwaltung entfällt
- im Compilerbau beliebt

Beispiel: n-Adress Maschine

Beispiel: `Z = (A-B) / (C + D*E)` Hilfsregister: `T`

3-Adress Maschine

```
sub Z, A, B
mul T, D, E
add T, C, T
div Z, Z, T
```

2-Adress Maschine

```
mov  Z, A
sub  Z, B
mov  T, D
mul  T, E
add  T, C
div  Z, T
```

1-Adress Maschine

```
load  D
mul   E
add   C
stor  Z
load  A
sub   B
div   Z
stor  Z
```

0-Adress Maschine

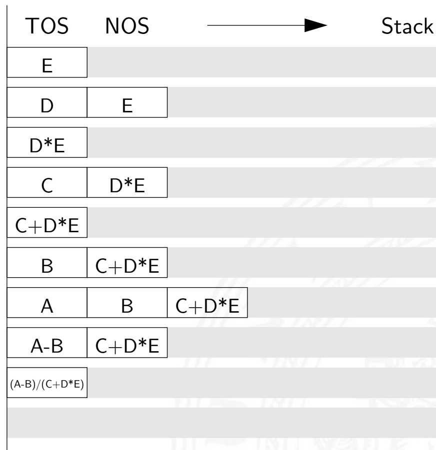
```
push  E
push  D
mul
push  C
add
push  B
push  A
sub
div
pop   Z
```

Beispiel: Stack-Maschine / 0-Adress Maschine

Beispiel: `Z = (A-B) / (C + D*E)`

0-Adress Maschine

```
push  E
push  D
mul
push  C
add
push  B
push  A
sub
div
pop   Z
```



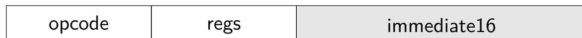
Adressierungsarten

- „immediate“
  - Operand steht direkt im Befehl
  - kein zusätzlicher Speicherzugriff
  - aber Länge des Operanden beschränkt
- „direkt“
  - Adresse des Operanden steht im Befehl
  - keine zusätzliche Adressberechnung
  - ein zusätzlicher Speicherzugriff
  - Adressbereich beschränkt
- „indirekt“
  - Adresse eines Pointers steht im Befehl
  - erster Speicherzugriff liest Wert des Pointers
  - zweiter Speicherzugriff liefert Operanden
  - sehr flexibel (aber langsam)

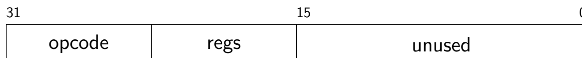
Adressierungsarten (cont.)

- „register“
  - wie Direktmodus, aber Register statt Speicher
  - 32 Register: benötigen 5 bit im Befehl
  - genug Platz für 2- oder 3-Adress Formate
- „register-indirekt“
  - Befehl spezifiziert ein Register
  - mit der Speicheradresse des Operanden
  - ein zusätzlicher Speicherzugriff
- „indiziert“
  - Angabe mit Register und Offset
  - Inhalt des Registers liefert Basisadresse
  - Speicherzugriff auf (Basisadresse+offset)
  - ideal für Array- und Objektzugriffe
  - Hauptmodus in RISC-Rechnern (auch: „Versatz-Modus“)

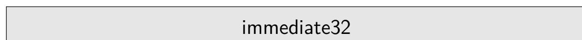
Immediate Adressierung



1-Wort Befehl

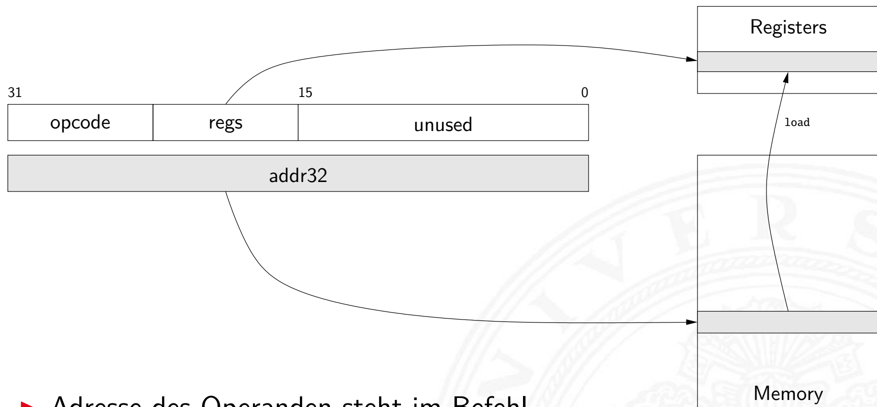


2-Wort Befehl



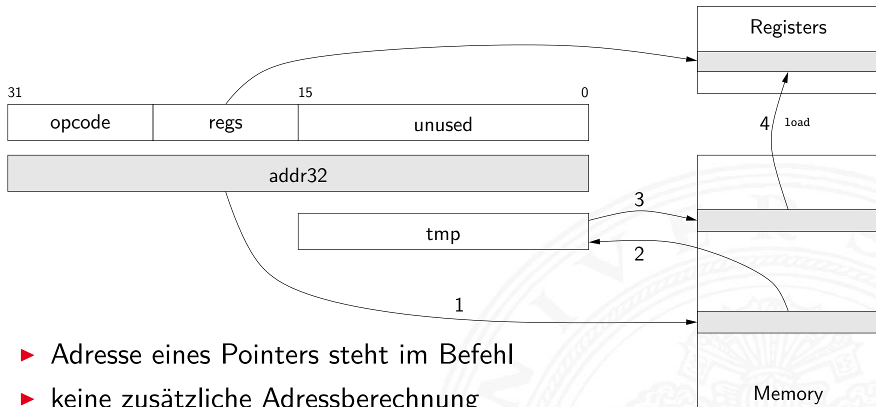
- Operand steht direkt im Befehl, kein zusätzlicher Speicherzugriff
- Länge des Operanden $<$ (Wortbreite - Opcodebreite)
- Darstellung größerer Zahlenwerte
  - 2-Wort Befehle (x86)
    zweites Wort für Immediate-Wert
  - mehrere Befehle (MIPS, SPARC)
    z.B. obere/untere Hälfte eines Wortes
  - Immediate-Werte mit zusätzlichem Shift (ARM)

Direkte Adressierung



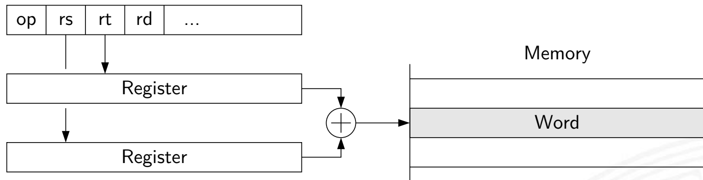
- Adresse des Operanden steht im Befehl
- keine zusätzliche Adressberechnung
- ein zusätzlicher Speicherzugriff: z.B. R3 = MEM[addr32]
- Adressbereich beschränkt, oder 2-Wort Befehl (wie Immediate)

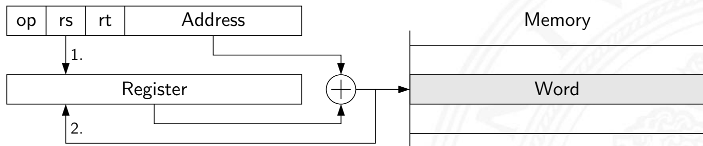
Indirekte Adressierung



- Adresse eines Pointers steht im Befehl
- keine zusätzliche Adressberechnung
- zwei zusätzliche Speicherzugriffe:
  z.B. tmp = MEM[addr32]   R3 = MEM[tmp]
- typische CISC-Adressierungsart, viele Taktzyklen
- kommt bei RISC-Rechnern nicht vor

Indizierte Adressierung





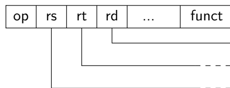
- indizierte Adressierung, z.B. für Arrayzugriffe
  - addr $= \langle \textit{Sourceregister} \rangle + \langle \textit{Basisregister} \rangle$
  - addr $= \langle \textit{Sourceregister} \rangle +$ offset;
    Sourceregister $=$ addr

Beispiel: MIPS Adressierungsarten

1. Immediate addressing

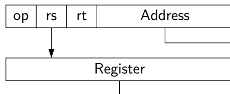
immediate

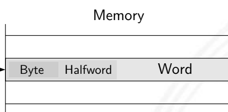
2. Register addressing



register

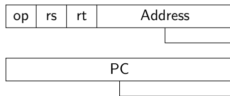
3. Base addressing

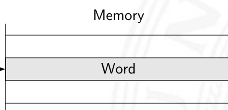




index + offset

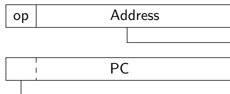
4. PC-relative addressing

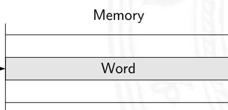




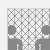
PC + offset

5. Pseudodirect addressing





$PC_{(31..28)}$ & address



typische Adressierungsarten

welche Adressierungsarten / -Varianten sind üblich?

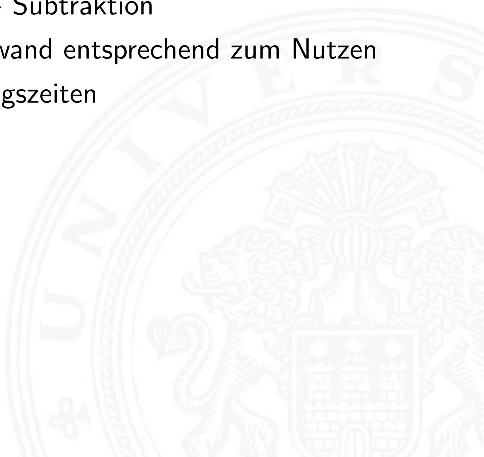
- 0-Adress (Stack-) Maschine — Java virtuelle Maschine
- 1-Adress (Akkumulator) Maschine — 8-bit Mikrocontroller
einige x86 Befehle
- 2-Adress Maschine — 16-bit Rechner
einige x86 Befehle
- 3-Adress Maschine — 32-bit RISC

- CISC Rechner unterstützen diverse Adressierungsarten
- RISC meistens nur indiziert mit Offset
- siehe `en.wikipedia.org/wiki/Addressing_mode`

Bewertung der ISA

Kriterien für einen *guten* Befehlssatz

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



Bewertung der ISA (cont.)

Statistiken zeigen: Dominanz der einfachen Instruktionen

- 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.)

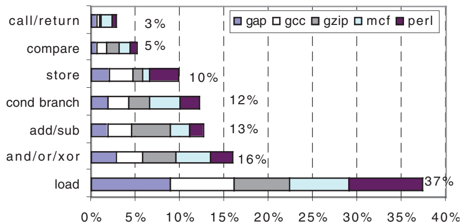
Instruction	compress	eqntott	espresso	gcc (cc1)	li	Int. average
load	20.8%	18.5%	21.9%	24.9%	23.3%	22%
store	13.8%	3.2%	8.3%	16.6%	18.7%	12%
add	10.3%	8.8%	8.15%	7.6%	6.1%	8%
sub	7.0%	10.6%	3.5%	2.9%	3.6%	5%
mul				0.1%		0%
div						0%
compare	8.2%	27.7%	15.3%	13.5%	7.7%	16%
mov reg-reg	7.9%	0.6%	5.0%	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%

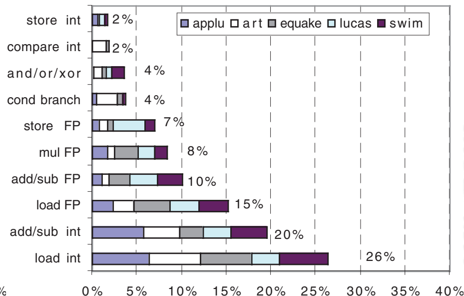
Figure D.15 **80x86 instruction mix for five SPECint92 programs.**

[HP17]

Bewertung der ISA (cont.)

- MIPS-Prozessor [HP17]





SPECint2000 (96%) SPECfp2000 (97%)

- 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 . . .

⇒ Motivation für RISC

CISC – Complex Instruction Set Computer

Rechnerarchitekturen mit irregulärem, komplexem Befehlssatz und (unterschiedlich) langer Ausführungszeit

- aus der Zeit der ersten Großrechner, 60er Jahre
- Programmierung auf Assemblerebene
- Komplexität durch sehr viele (mächtige) Befehle umgehen

typische Merkmale

- Instruktionssätze mit mehreren hundert Befehlen (> 300)
- unterschiedlich lange Instruktionsformate: 1 . . . n-Wort Befehle
  - komplexe Befehlscodierung
  - mehrere Schreib- und Lesezugriffe pro Befehl
- viele verschiedene Datentypen

CISC – **C**omplex **I**nstruction **S**et **C**omputer (cont.)

- sehr viele Adressierungsarten, -Kombinationen
  - fast alle Befehle können auf Speicher zugreifen
  - Mischung von Register- und Speicheroperanden
  - komplexe Adressberechnung
- Unterprogrammaufrufe: über Stack
  - Übergabe von Argumenten
  - Speichern des Programmzählers
  - explizite „Push“ und „Pop“ Anweisungen
- Zustandscodes („*Flags*“)
  - implizit gesetzt durch arithmetische und logische Anweisungen

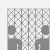
CISC – Complex Instruction Set Computer (cont.)

Vor- / Nachteile

+ nah an der Programmiersprache, einfacher Assembler
+ kompakter Code: weniger Befehle holen, kleiner I-Cache
– Befehlssatz vom Compiler schwer auszunutzen
– Ausführungszeit abhängig von: Befehl, Adressmodi ...
– Instruktion holen schwierig, da variables Instruktionsformat
– Speicherhierarchie schwer handhabbar: Adressmodi
– Pipelining schwierig

Beispiele

- Intel x86 / IA-64, Motorola 68 000, DEC Vax



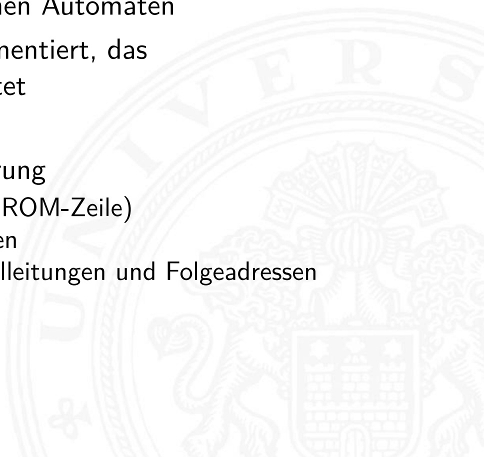
CISC – Mikroprogrammierung

- ein Befehl kann nicht in einem Takt abgearbeitet werden

⇒ Unterteilung in Mikroinstruktionen ($\varnothing$ 5 . . . 7)

- Ablaufsteuerung durch endlichen Automaten
- meist als ROM (RAM) implementiert, das *Mikroprogrammworte* beinhaltet

1. horizontale Mikroprogrammierung
   - langes Mikroprogrammwort (ROM-Zeile)
   - steuert direkt alle Operationen
   - Spalten entsprechen: Kontrollleitungen und Folgeadressen



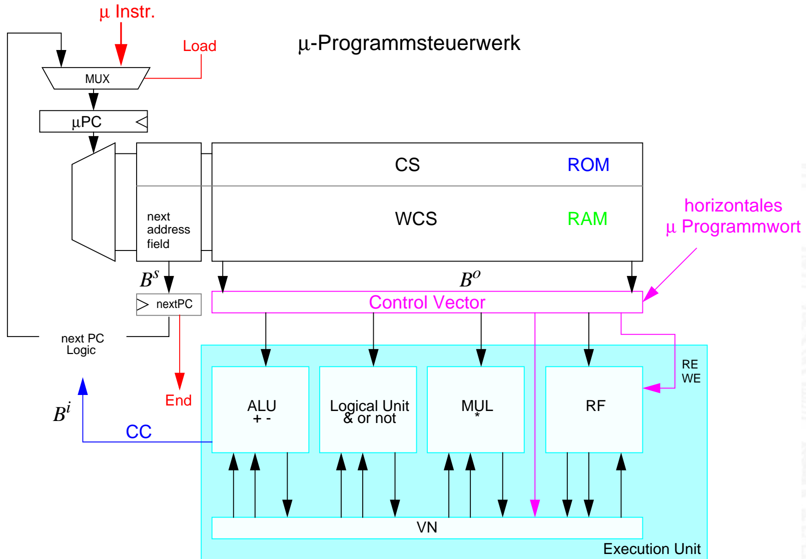
CISC – Mikroprogrammierung (cont.)

2. vertikale Mikroprogrammierung
   - kurze Mikroprogrammworte
   - Spalten enthalten Mikrooperationscode
   - mehrstufige Decodierung für Kontrollleitungen

+ CISC-Befehlssatz mit wenigen Mikrobefehlen realisieren
+ bei RAM: Mikrobefehlssatz austauschbar
− (mehrstufige) ROM/RAM Zugriffe: zeitaufwändig

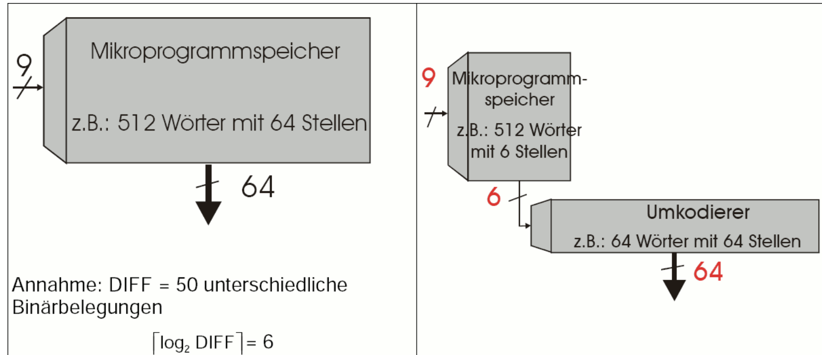
▸ horizontale Mikroprog. ▸ vertikale Mikroprog.

horizontale Mikroprogrammierung



Mikroprogrammierung

vertikale Mikroprogrammierung



◂ Mikroprogrammierung

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

oft auch: „**R**egular **I**nstruction **S**et **C**omputer“

- Grundidee: Komplexitätsreduktion in der CPU
- seit den 80er Jahren: „RISC-Boom“
  - internes Projekt bei IBM
  - von Hennessy (Stanford) und Patterson (Berkeley) publiziert
- Hochsprachen und optimierende Compiler
  - $\Rightarrow$ kein Bedarf mehr für mächtige Assemblerbefehle
  - $\Rightarrow$ pro Assemblerbefehl muss nicht mehr „möglichst viel“ lokal in der CPU gerechnet werden (CISC Mikroprogramm)

Beispiele

- IBM 801, MIPS, SPARC, DEC Alpha, ARM

typische Merkmale

- reduzierte Anzahl einfacher Instruktionen (z.B. 128)
  - benötigen in der Regel mehr Anweisungen für eine Aufgabe
  - werden aber mit kleiner, schneller Hardware ausgeführt

RISC – Reduced Instruction Set Computer (cont.)

- reguläre Struktur, z.B. 32-bit Wortbreite, 32-bit Befehle
- nur ein-Wort Befehle
- alle Befehle in gleicher Zeit ausführbar $\Rightarrow$ Pipeline-Verarbeitung
- Speicherzugriff *nur* durch „Load“ und „Store“ Anweisungen
  - alle anderen Operationen arbeiten auf Registern
  - keine Speicheroperanden
- Register-orientierter Befehlssatz
  - viele universelle Register, keine Spezialregister ($\geq$ 32)
  - oft mehrere (logische) *Registersätze*: Zuordnung zu Unterprogrammen, Tasks etc.
- Unterprogrammaufrufe: über Register
  - Register für Argumente, „Return“-Adressen, Zwischenergebnisse
- keine Zustandscodes (*„Flags“*)
  - spezielle Testanweisungen
  - speichern Resultat direkt im Register
- optimierende Compiler statt Assemblerprogrammierung

RISC – Reduced Instruction Set Computer (cont.)

Vor- / Nachteile

- \+ fest-verdrahtete Logik, kein Mikroprogramm
- \+ einfache Instruktionen, wenige Adressierungsarten
- \+ Pipelining gut möglich
- \+ Cycles per Instruction = 1
  in Verbindung mit Pipelining: je Takt (mind.) ein neuer Befehl
- – längerer Maschinencode
- – viele Register notwendig
- ▶ optimierende Compiler nötig / möglich
- ▶ High-performance Speicherhierarchie notwendig

CISC vs. RISC

ursprüngliche Debatte

- streng geteilte Lager
- pro CISC: einfach für den Compiler; weniger Code Bytes
- pro RISC: besser für optimierende Compiler;
  schnelle Abarbeitung auf einfacher Hardware

aktueller Stand

- Grenzen verwischen
  - RISC-Prozessoren werden komplexer
  - CISC-Prozessoren weisen RISC-Konzepte oder gar RISC-Kern auf
- für Desktop Prozessoren ist die Wahl der ISA kein Thema
  - Code-Kompatibilität ist sehr wichtig!
  - mit genügend Hardware wird alles schnell ausgeführt
- eingebettete Prozessoren: eindeutige RISC-Orientierung
  - \+ kleiner, billiger, weniger Leistungsverbrauch

ISA Design heute

- Restriktionen durch Hardware abgeschwächt
- Code-Kompatibilität leichter zu erfüllen
  - Emulation in Firm- und Hardware
- Intel bewegt sich weg von IA-32
  - erlaubt nicht genug Parallelität

  hat IA-64 eingeführt („Intel Architecture 64-bit“)

  $\Rightarrow$ neuer Befehlssatz mit expliziter Parallelität (EPIC)

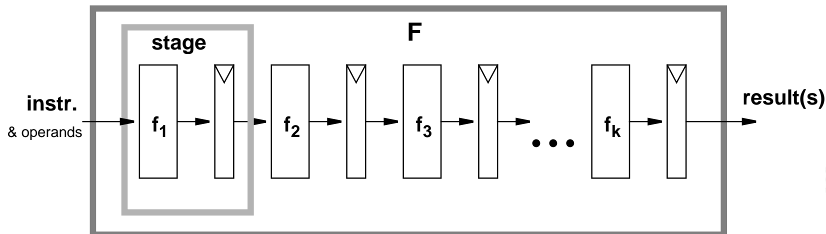
  $\Rightarrow$ 64-bit Wortgrößen (überwinden Adressraumlimits)

  $\Rightarrow$ benötigt hoch entwickelte Compiler

Gliederung

1. Rechnerarchitektur
2. Bewertung von Architekturen und Rechnersystemen
3. Instruction Set Architecture
4. Pipelining
   - Motivation und Konzept
   - Befehlspipeline
   - MIPS
   - Bewertung
   - Hazards
   - Superskalare Rechner
5. Speicherhierarchie

Pipelining / Fließbandverarbeitung



Grundidee

- Operation F kann in Teilschritte zerlegt werden
- jeder Teilschritt f_i braucht ähnlich viel Zeit
- Teilschritte $f_1..f_k$ können parallel zueinander ausgeführt werden
- Trennung der Pipelinestufen („*stage*“) durch Register
- Zeitbedarf für Teilschritt $f_i \gg$ Zugriffszeit auf Register (t_{FF})

Pipelining / Fließbandverarbeitung (cont.)

Pipelining-Konzept

- Prozess in unabhängige Abschnitte aufteilen
- Objekt sequenziell durch diese Abschnitte laufen lassen
  - zu jedem Zeitpunkt werden zahlreiche Objekte bearbeitet
  - –"– sind alle Stationen ausgelastet

Konsequenz

- Pipelining lässt Vorgänge gleichzeitig ablaufen
- reale Beispiele: Autowaschanlagen, Fließbänder in Fabriken

Pipelining / Fließbandverarbeitung (cont.)

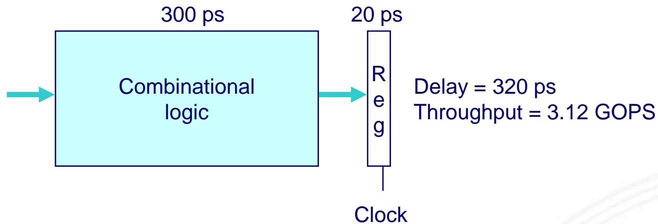
Arithmetische Pipelines

- 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

Befehlspipeline im Prozessor

- Idee: die Phasen der von-Neumann Befehlsabarbeitung
  (Befehl holen, Befehl decodieren ...) als Pipeline implementieren

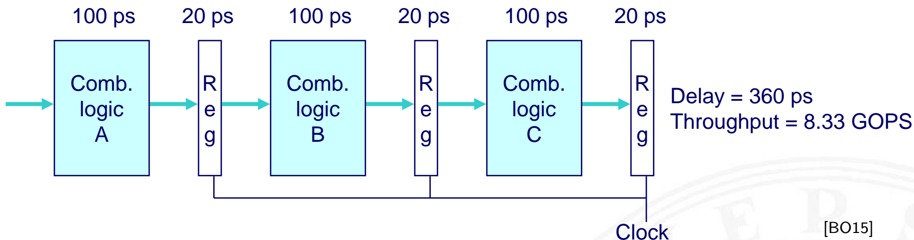
Beispiel: Schaltnetz ohne Pipeline



[BO15]

- Verarbeitung erfordert 300 ps
- weitere 20 ps um das Resultat im Register zu speichern
- Zykluszeit: mindestens 320 ps

Beispiel: Version mit 3-stufiger Pipeline

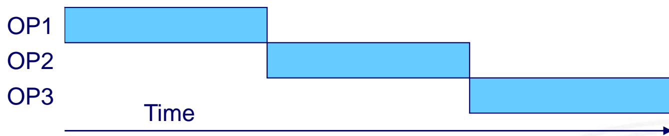


[BO15]

- Kombinatorische Logik in 3 Blöcke zu je 100 ps aufgeteilt
- neue Operation, sobald vorheriger Abschnitt durchlaufen wurde
  $\Rightarrow$ alle 120 ps neue Operation
- allgemeine Latenzzunahme
  $\Rightarrow$ 360 ps von Start bis Ende

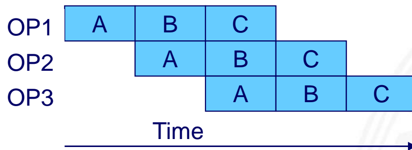
Prinzip: 3-stufige Pipeline

- ohne Pipeline

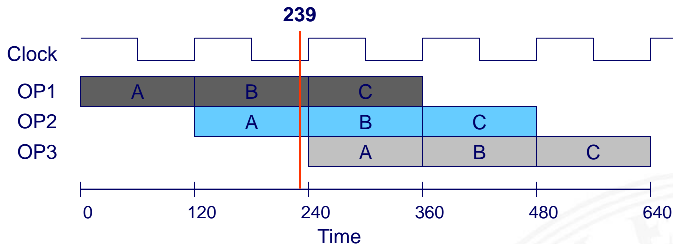


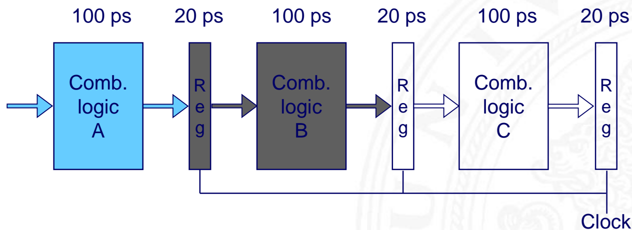
- 3-stufige Pipeline

[BO15]



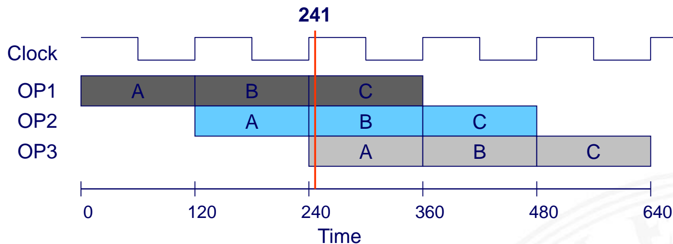
Timing: 3-stufige Pipeline

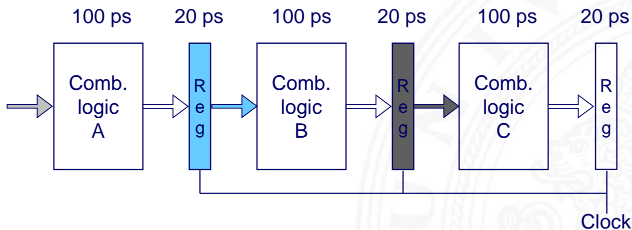




[BO15]

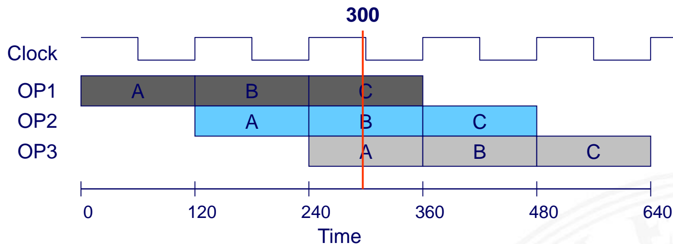
Timing: 3-stufige Pipeline

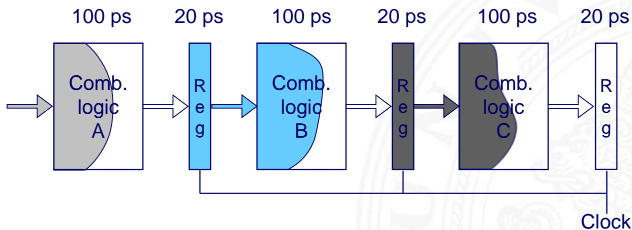




[BO15]

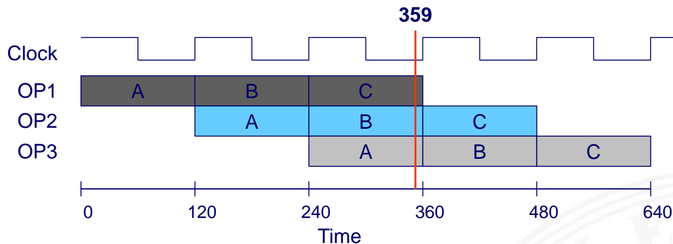
Timing: 3-stufige Pipeline

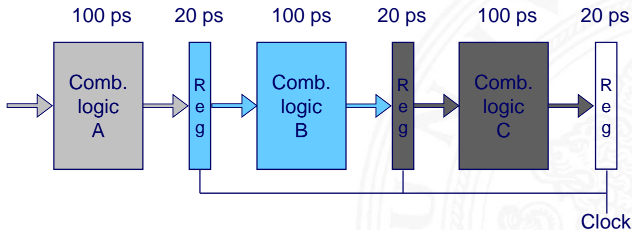




[BO15]

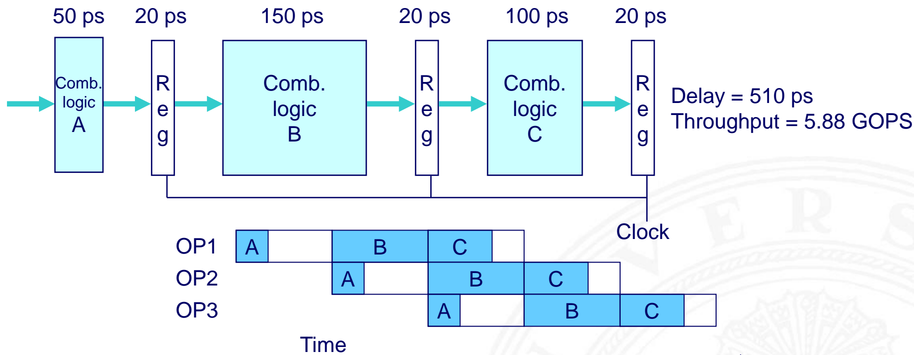
Timing: 3-stufige Pipeline





[BO15]

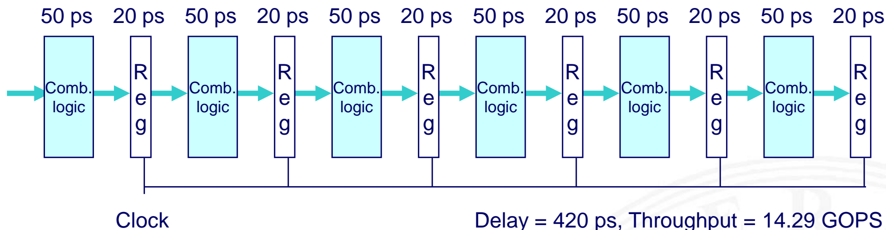
Limitierungen: nicht uniforme Verzögerungen



[BO15]

- Taktfrequenz limitiert durch langsamste Stufe
- Schaltung in möglichst gleich schnelle Stufen aufteilen

Limitierungen: Register „Overhead“

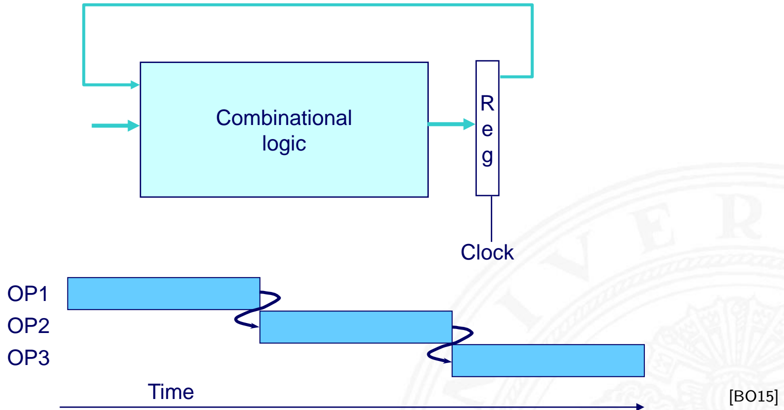


[BO15]

- registerbedingter Overhead wächst mit Pipelinelänge
- (anteilige) Taktzeit für das Laden der Register

		Overhead	Taktperiode
1-Register:	6,25%	20 ps	320 ps
3-Register:	16,67%	20 ps	120 ps
6-Register:	28,57%	20 ps	70 ps

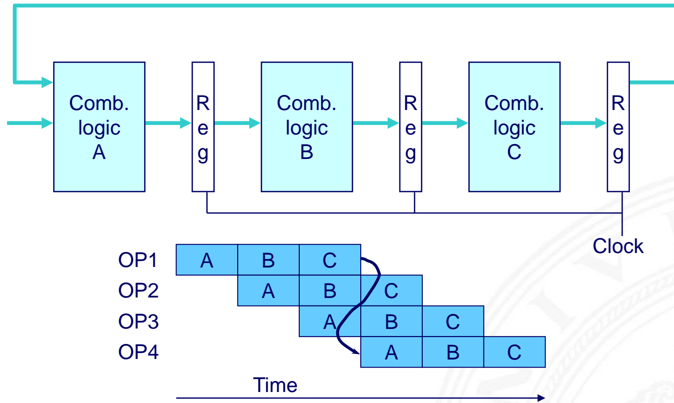
Limitierungen: Datenabhängigkeiten



[BO15]

- jede Operation hängt vom Ergebnis der Vorhergehenden ab

Limitierungen: Datenabhängigkeiten (cont.)



[BO15]

⇒ Resultat-Feedback kommt zu spät für die nächste Operation

⇒ Pipelining ändert Verhalten des gesamten Systems

von-Neumann Befehlszyklus

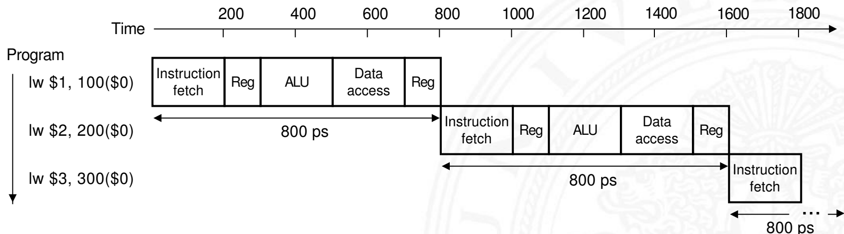
typische Schritte der Befehlsabarbeitung (von ISA abhängig)

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: Daten laden/abspeichern
WB	**W**rite **B**ack Ergebnis in Register zurückschreiben

von-Neumann Befehlszyklus (cont.)

- je nach Instruktion sind 3-5 dieser Schritte notwendig
  - *nop*: nur Instruction-Fetch
  - *jump*: kein Speicher-/Registerzugriff
- Schritte können auch feiner unterteilt werden (mehr Stufen)

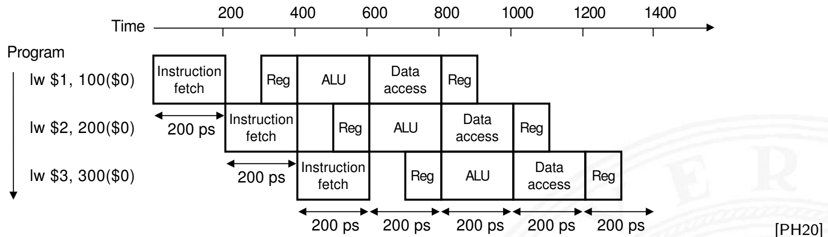
serielle Bearbeitung ohne Pipelining



[PH20]

von-Neumann Befehlszyklus (cont.)

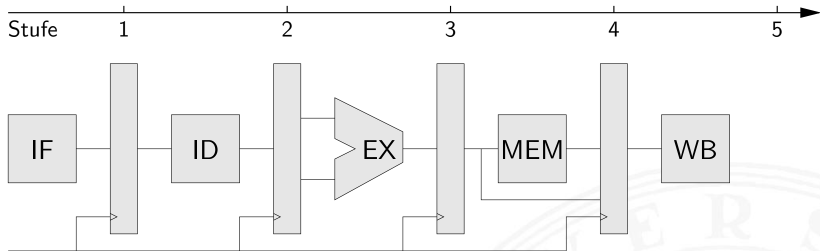
Pipelining für die einzelnen Schritte der Befehlsausführung



[PH20]

- Befehle überlappend ausführen: neue Befehle holen, dann decodieren, während vorherige noch ausgeführt werden
- Register trennen Pipelinestufen

Klassische 5-stufige Pipeline

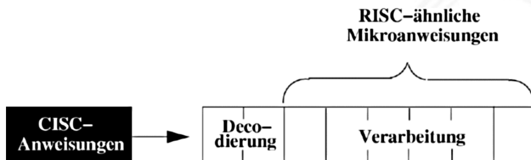


- Grundidee der ursprünglichen RISC-Architekturen
- + Durchsatz ca. $3 \dots 5 \times$ besser als serielle Ausführung
- + guter Kompromiss aus Leistung und Hardwareaufwand

- MIPS-Architektur (aus Patterson, Hennessy [PH16])
  - MIPS ohne Pipeline
  - MIPS Pipeline
  - Pipeline Schema

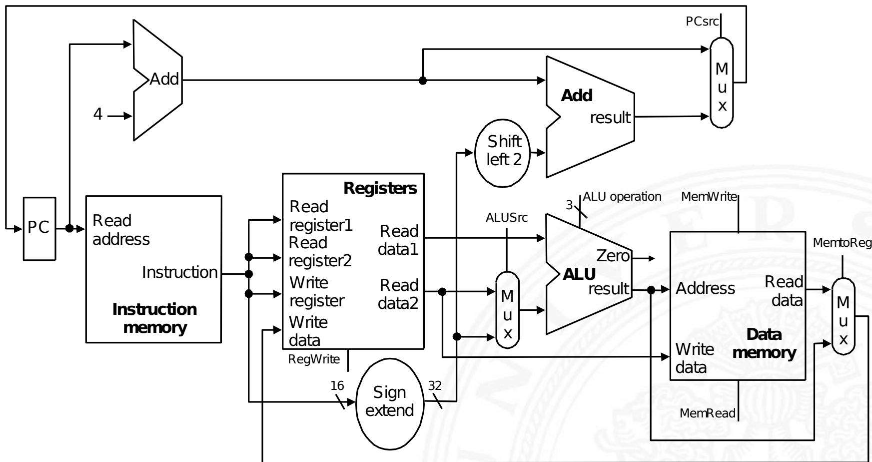
Klassische 5-stufige Pipeline (cont.)

- RISC ISA: Pipelining wird direkt umgesetzt
  - Befehlssätze auf diese Pipeline hin optimiert
  - IBM-801, MIPS R-2000/R-3000 (1985), SPARC (1987)
- CISC-Architekturen heute ebenfalls mit Pipeline
  - Motorola 68020 (zweistufige Pipeline, 1984), Intel 486 (1989), Pentium (1993) ...
  - Befehle in Folgen RISC-ähnlicher Anweisungen umsetzen



\+ CISC-Software bleibt lauffähig
\+ Befehlssatz wird um neue RISC Befehle erweitert

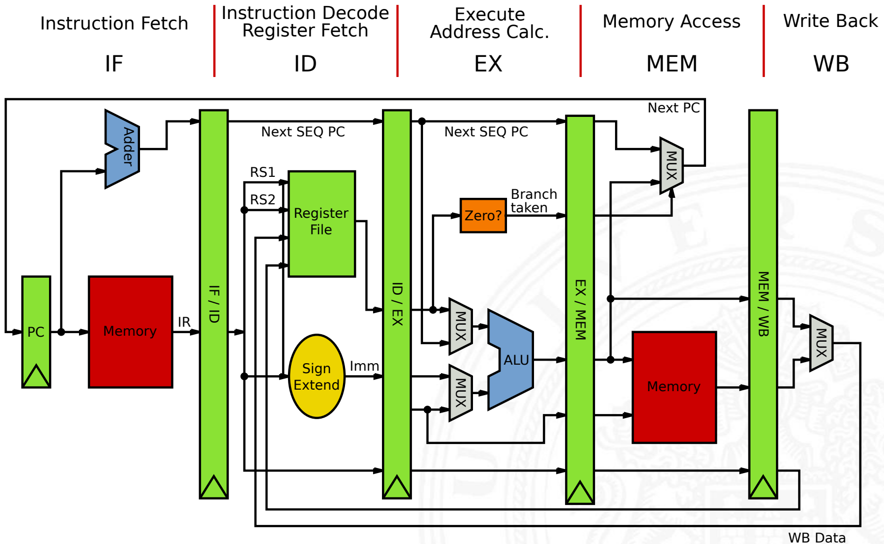
MIPS: serielle Realisierung ohne Pipeline



längster Pfad: PC - IM - REG - MUX - ALU - DM - MUX - PC/REG [PH20]

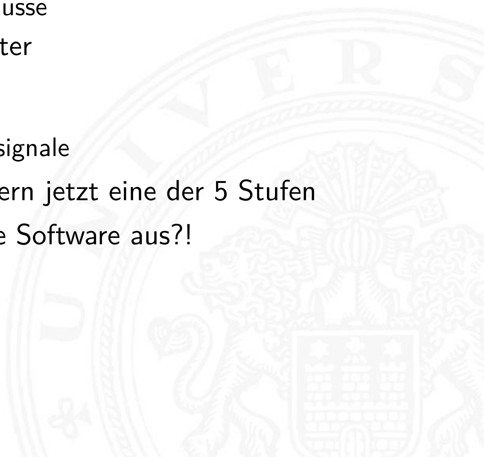
◂ RISC Pipelining

MIPS: mit 5-stufiger Pipeline



◂ RISC Pipelining

MIPS: mit 5-stufiger Pipeline (cont.)

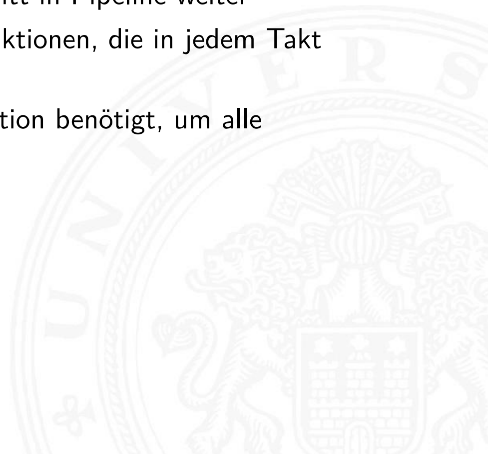


- die Hardwareblöcke selbst sind unverändert
  - PC, Addierer fürs Inkrementieren des PC
  - Registerbank
  - Rechenwerke: ALU, sign-extend, zero-check
  - Multiplexer und Leitungen/Busse
- vier zusätzliche Pipeline-Register
  - die (decodierten) Befehle
  - alle Zwischenergebnisse
  - alle intern benötigten Statussignale
- längster Pfad zwischen Registern jetzt eine der 5 Stufen
- aber wie wirkt sich das auf die Software aus?!

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

+ Schaltnetze in kleinere Blöcke aufgeteilt ⇒ höherer Takt
+ im Idealfall ein neuer Befehl pro Takt gestartet ⇒ höherer Durchsatz, bessere Performanz
+ geringer Zusatzaufwand an Hardware
+ Pipelining ist für den Programmierer nicht direkt sichtbar!
  – Achtung: Daten-/Kontrollabhängigkeiten (s.u.)
– 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 – Bewertung (cont.)

Analyse

- N Instruktionen; K Pipelinestufen
- ohne Pipeline: $N \cdot K$ Taktzyklen
- mit Pipeline: $K + N - 1$ Taktzyklen

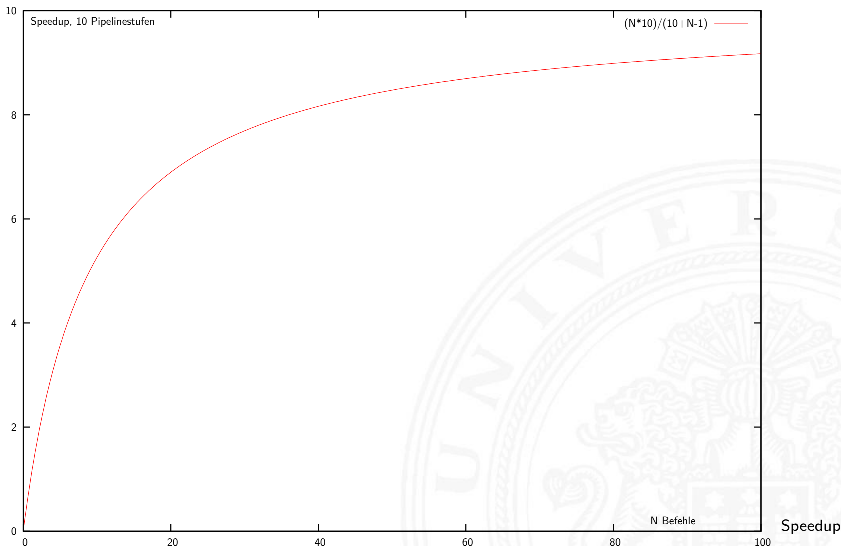
- „Speedup“ $S = \frac{N \cdot K}{K+N-1}$, $\lim_{N \to \infty} S = K$

$\Rightarrow$ ein großer Speedup wird erreicht durch

- große Pipelinetiefe: K
- lange Instruktionssequenzen: N

- wegen Daten- und Kontrollabhängigkeiten nicht erreichbar
- außerdem: Register-Overhead nicht berücksichtigt

Prozessorpipeline – Bewertung (cont.)



Speedup

Prozessorpipeline – Dimensionierung

- größeres K wirkt sich direkt auf den Durchsatz aus
- weniger Logik zwischen den Registern, höhere Taktfrequenzen
- zusätzlich: technologischer Fortschritt (1985 . . . 2010)
- Beispiele

CPU	Pipelinestufen	Taktfrequenz [MHz]
80 386	1	33
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

BASIC INSTRUCTION FORMATS

	31–26	25–21	20–16	15–11	10–6	5–0
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate (15–0)		
J	opcode	address (25–0)				

FLOATING-POINT INSTRUCTION FORMATS

	31–26	25–21	20–16	15–11	10–6	5–0
FR	opcode	fmt	ft	fs	fd	funct
FI	opcode	fmt	ft	immediate (15–0)		

MIPS-Befehlsformate [PH20]

Prozessorpipeline – Auswirkungen (cont.)

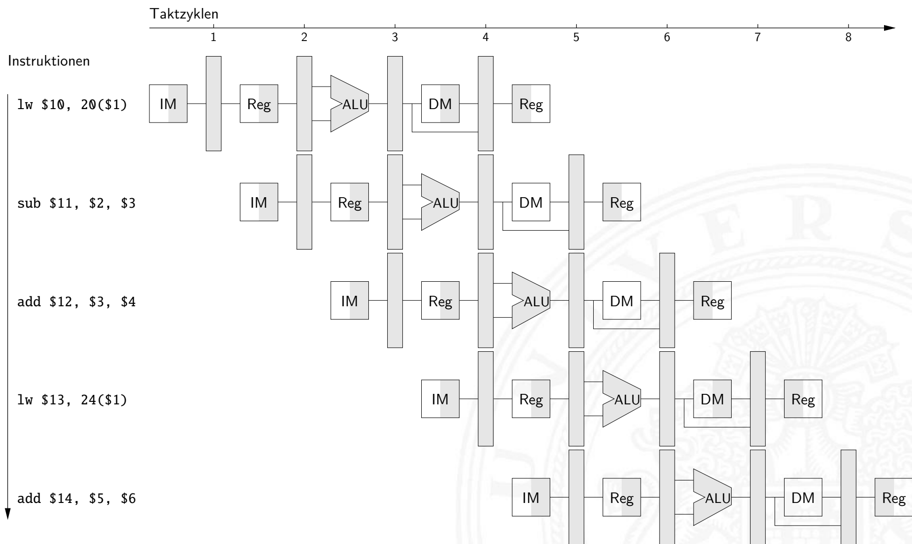
schlecht für Pipelining: *Pipelinekonflikte / -Hazards*

- 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 Schema



◂ RISC Pipelining

Pipeline Strukturkonflikte

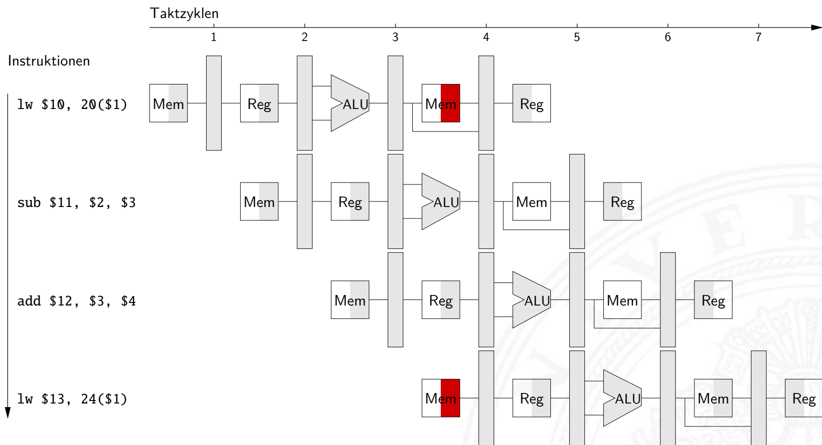
Strukturkonflikt / Structural Hazard

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

▸ Beispiel

⇒ Mehrfachauslegung der betreffenden Ressourcen
  - Harvard-Architektur vermeidet Strukturkonflikt aus Beispiel
  - Multi-Port Register
  - mehrfach vorhandene Busse und Multiplexer ...

Beispiel: Strukturkonflikt



◂ Strukturkonflikte

gleichzeitigen Laden aus *einem* Speicher, zwei verschiedene Adressen

Pipeline Datenkonflikte

Datenkonflikt / Data Hazard

- eine Instruktion braucht die Ergebnisse einer vorhergehenden, diese wird aber noch in der Pipeline bearbeitet
- Datenabhängigkeiten aufeinanderfolgender Befehle ▸ Beispiel
  - Operanden während ID-Phase aus Registerbank lesen
  - Resultate werden erst in WB-Phase geschrieben
  - ⇒ aber: Resultat ist schon nach EX-/MEM-Phase bekannt

Forwarding

- zusätzliche Hardware („*Forwarding-Unit*") kann Datenabhängigkeiten auflösen
- Änderungen in der Pipeline Steuerung
- neue Datenpfade und Multiplexer ▸ ohne Forwarding ▸ mit Forwarding

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“*

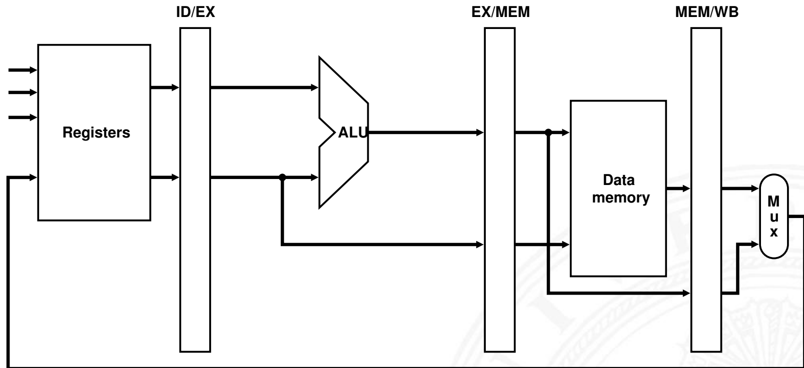
   ▸ Beispiel

   - zusätzliche (Hardware) Kontrolleinheit: komplexes Steuerwerk
   - automatisches Stoppen der Pipeline, bis die benötigten Daten zur Verfügung stehen – Strategien:
     - in Pipeline werden keine neuen Instruktionen geladen
     - Hardware erzeugt: Pipelineleerlauf / *„pipeline stall“*

„Scoreboard“

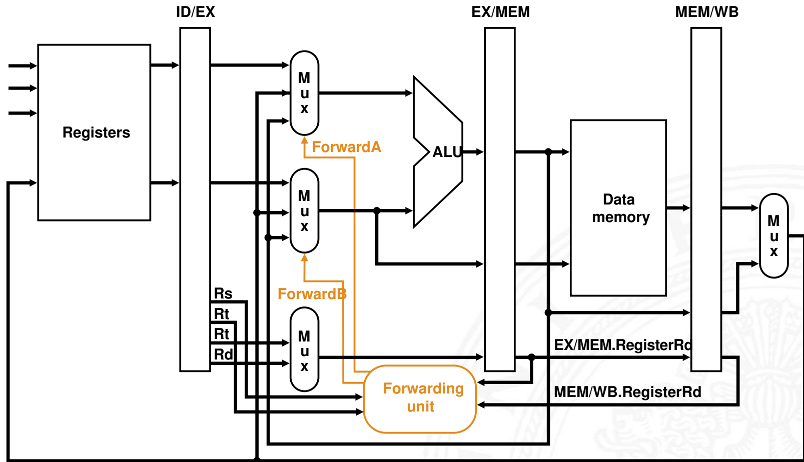
- Hardware Einheit zur zentralen Hazard-Erkennung und -Auflösung
- Verwaltet Instruktionen, benutzte Einheiten und Register der Pipeline (siehe *„Superskalare Rechner“*, ab Folie 151)

Beispiel: MIPS Datenpfad



◂ Forwarding

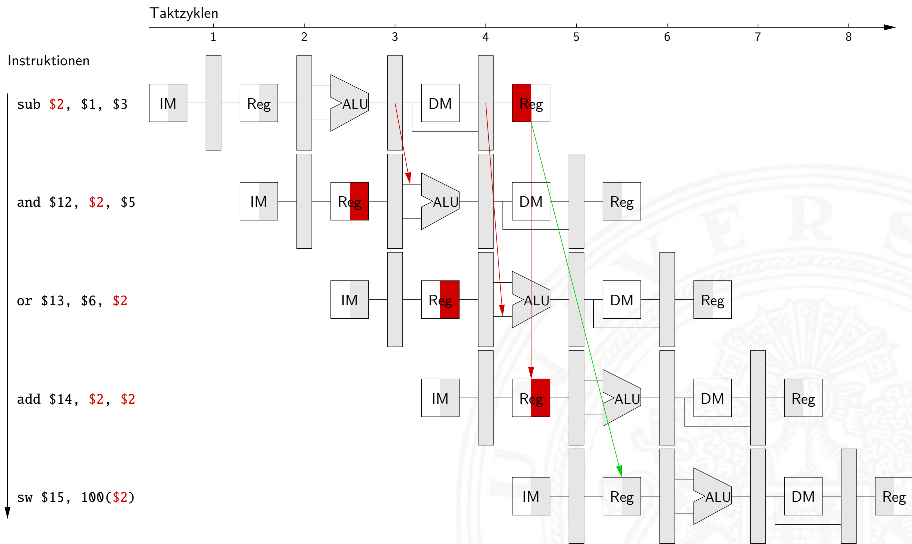
Beispiel: MIPS Forwarding



[PH20]

◂ Forwarding

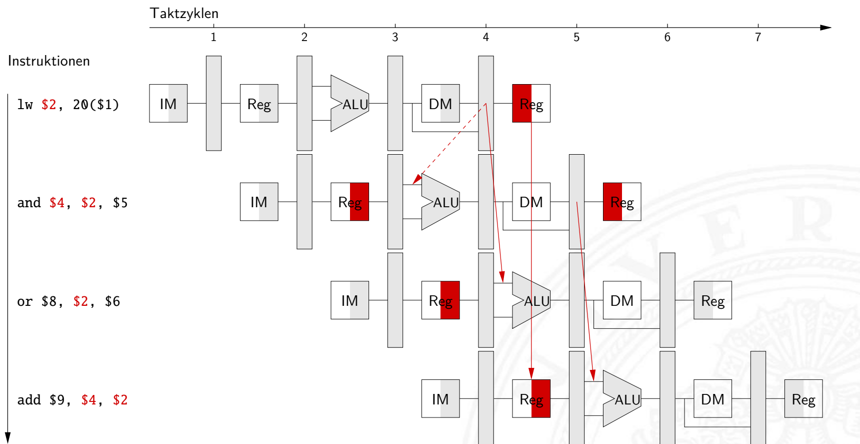
Beispiel: Datenkonflikt



Datenkonflikte

Befehle wollen R2 lesen, während es noch vom ersten Befehl berechnet wird

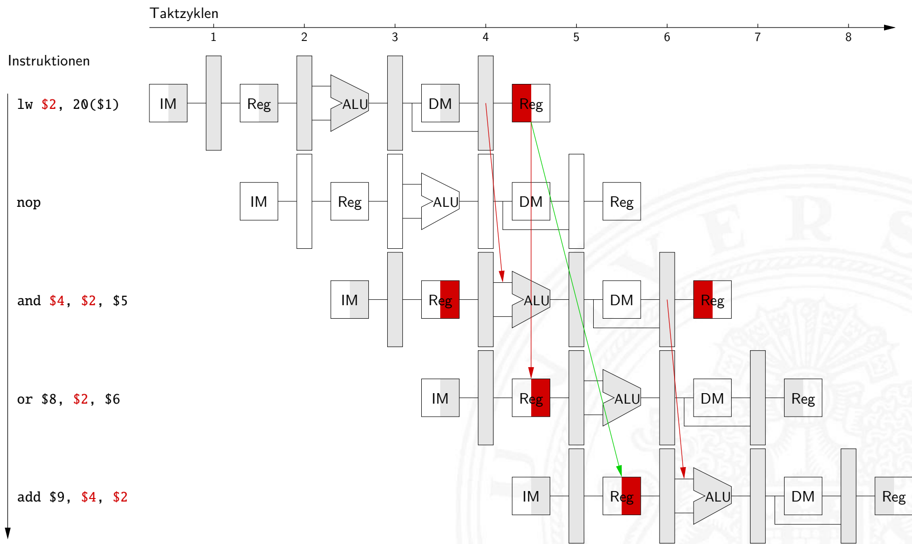
Beispiel: Rückwärtsabhängigkeit



◂ Datenkonflikte

Befehle wollen R2 lesen, bevor es aus Speicher geladen wird

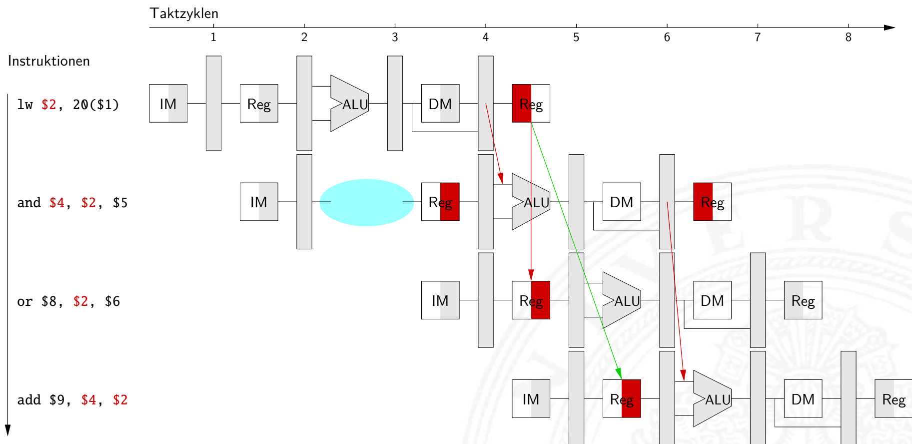
Beispiel: nop



◂ Datenkonflikte

Compiler kennt Hardware und hat einen nop-Befehl eingefügt

Beispiel: „bubbles“



Datenkonflikte

Hardware verzögert Bearbeitung, bis Konflikte beseitigt sind („*bubbles*“)

Pipeline Steuerkonflikte

Steuerkonflikt / Control Hazard

Beispiel

- Unterbrechung des sequenziellen Ablaufs durch Sprungbefehle und Unterprogrammaufrufe: `call` und `ret`
  - Instruktionen die auf (bedingte) Sprünge folgen, werden bereits in die Pipeline geschoben
  - Sprungadresse und Status (*taken*/*untaken*) sind aber erst am Ende der EX-Phase bekannt
  - einige Befehle wurden bereits teilweise ausgeführt, Resultate eventuell „ge-forwarded“

– alle Zwischenergebnisse müssen verworfen werden
  - inklusive aller Forwarding-Daten
  - Pipeline an korrekter Zieladresse neu starten
  - erfordert sehr komplexe Hardware

– jeder (ausgeführte) Sprung kostet enorm Performanz

Pipeline Steuerkonflikte (cont.)

Lösungsmöglichkeiten für Steuerkonflikte

- ad-hoc Lösung: *„Interlocking“*
  Pipeline prinzipiell bei Sprüngen leeren
  - – ineffizient: ca. 19 % der Befehle sind Sprünge

1. Annahme: (nicht) ausgeführter Sprung *„(un)taken branch“*
   - \+ kaum zusätzliche Hardware
   - – im Fehlerfall muss Pipeline 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 ...

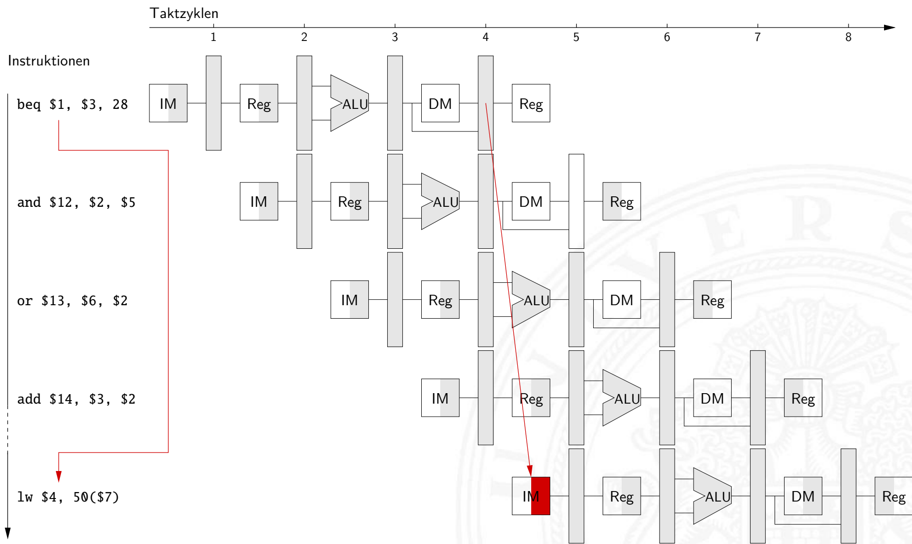
Pipeline Steuerkonflikte (cont.)

- Schleifen abrollen / „*Loop unrolling*“
  - zusätzliche Maßnahme zu allen zuvor skizzierten Verfahren
  - bei statischer 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

Beispiel: Steuerkonflikt

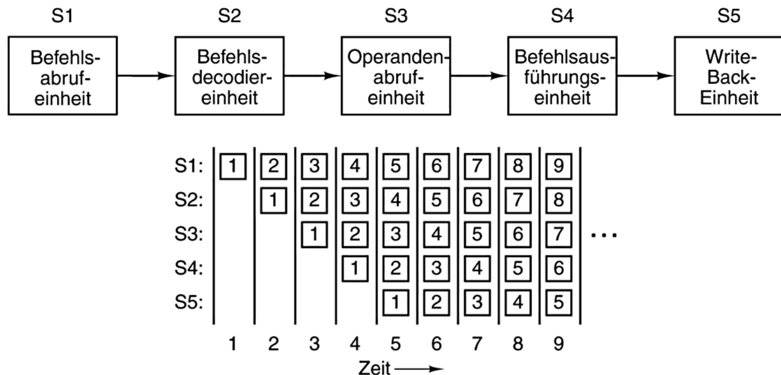


◂ Steuerkonflikte

Pipeline – Zusammenfassung

- von-Neumann Zyklus auf separate Phasen aufteilen
- überlappende Ausführung von mehreren Befehlen
  - einfachere Hardware für jede Phase $\Rightarrow$ höherer Takt
  - mehrere Befehle in Bearbeitung $\Rightarrow$ höherer Durchsatz
  - 5-stufige RISC-Pipeline: IF$\rightarrow$ID/OF$\rightarrow$Exe$\rightarrow$Mem$\rightarrow$WB
  - mittlerweile sind $9 \ldots 20$ Stufen üblich
- Struktur-, Daten- und Steuerkonflikte
  - Lösung durch mehrfache/bessere Hardware
  - Data-Forwarding umgeht viele Datenabhängigkeiten
  - Sprungbefehle sind ein ernstes Problem
- Pipelining ist prinzipiell unabhängig von der ISA
  - einige Architekturen basiseren auf Pipelining (MIPS)
  - Compiler/Tools/Progammierer sollten CPU Pipeline kennen

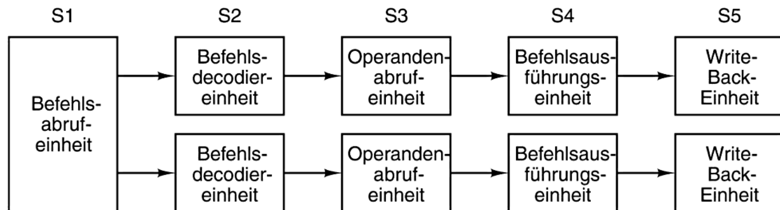
Befehlspipeline



[TA14]

- Befehl in kleinere, schnellere Schritte aufteilen $\Rightarrow$ höherer Takt
- mehrere Instruktionen überlappt ausführen $\Rightarrow$ höherer Durchsatz

Parallele Pipelines



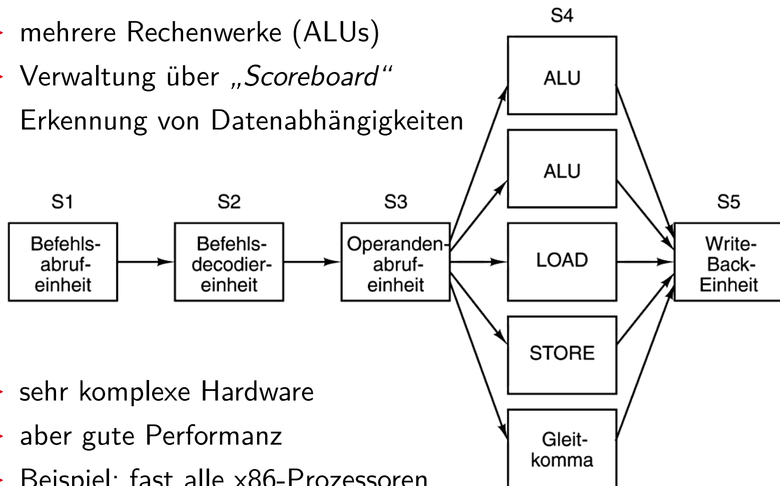
[TA14]

- im Bild jeweils zwei Operationen pro Pipelinestufe
- parallele („superskalare“) Ausführung

- komplexe Hardware (Daten- und Kontrollabhängigkeiten)
- Beispiel: Pentium

Superskalarer Prozessor

- mehrere Rechenwerke (ALUs)
- Verwaltung über „*Scoreboard*“
  Erkennung von Datenabhängigkeiten



- sehr komplexe Hardware
- aber gute Performanz
- Beispiel: fast alle x86-Prozessoren seit Pentium II

[TA14]

Superskalar

- Superskalare CPUs besitzen mehrere Recheneinheiten: 4 ... 10
- in jedem Takt werden (dynamisch) mehrere Instruktionen eines konventionell linearen Instruktionsstroms abgearbeitet

⇒ ILP (**I**nstruction **L**evel **P**arallelism)

- 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

Datenabhängigkeiten superskalarer Prozessoren

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

Superskalar – Datenabhängigkeiten (cont.)

„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

- Beispiel
  - Originalcode

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

nach Renaming

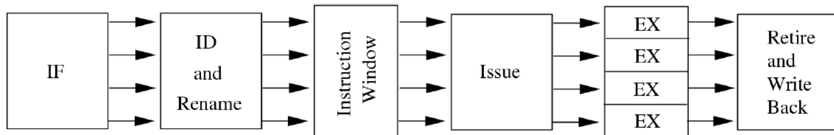
```
tmp1 = a + b;
res1 = c + tmp;
tmp2 = d + e;
res2 = tmp2 - f;
tmp  = tmp2;
```

  - Parallelisierung des modifizierten Codes

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

Superskalar – Pipeline

Aufbau der superskalaren Pipeline

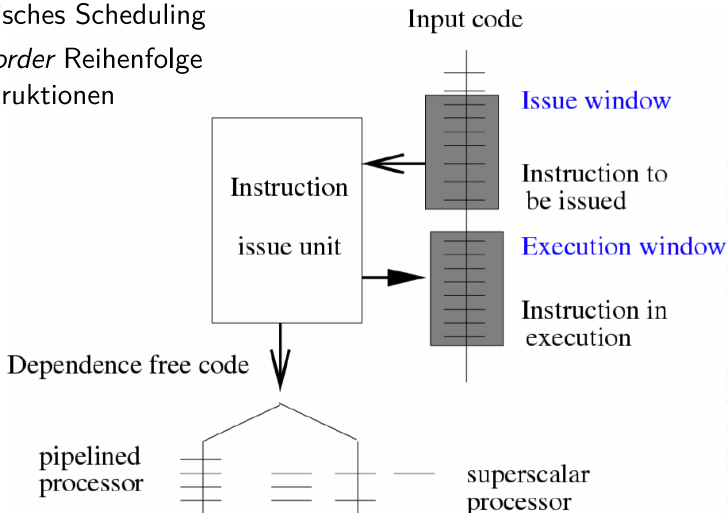


- lange Pipelines mit vielen Phasen: Fetch (Prefetch, Predecode), Decode / Register-Renaming, Issue, Dispatch, Execute, Retire (Commit, Complete / Reorder), Write-Back
- je nach Implementation unterschiedlich aufgeteilt
- entscheidend für superskalare Architektur sind die Schritte
  vor den ALUs: Issue,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

Superskalar – Pipeline (cont.)

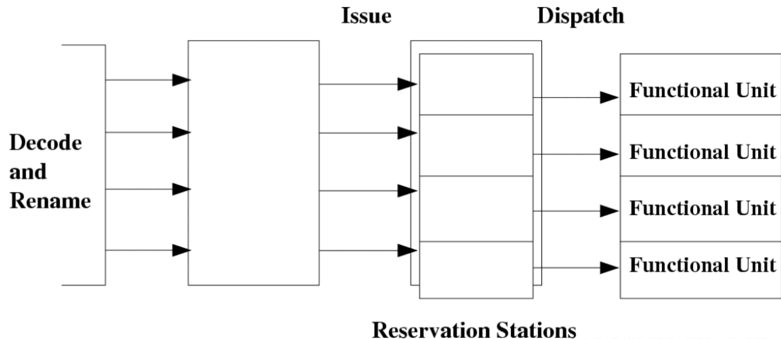
- Dynamisches Scheduling

⇒ *out-of-order* Reihenfolge der Instruktionen



Superskalar – Pipeline (cont.)

- 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
- ggf. „*Retire*"-Stufe
  - Reorder-Buffer: erzeugt wieder *in-order* Reihenfolge
  - commit: „richtig ausgeführte" Instruktionen gültig machen
abort: Instruktionen verwerfen, z.B. Sprungvorhersage falsch
- Dynamisches Scheduling: zuerst '67 in IBM 360 (R. Tomasulo)
  - Forwarding
  - Registerumbenennung und Reservation Stations

Superskalar – Probleme

Spezielle Probleme superskalarer Pipelines

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

  ⇒ Datenabhängigkeiten können sich mit jedem Takt ändern
- Kontrollflussabhängigkeiten:
  Anzahl der Instruktionen zwischen bedingten Sprüngen
  limitiert Anzahl parallelisierbarer Instruktionen

⇒ „*Loop Unrolling*“ besonders wichtig
  + optimiertes (dynamisches) Scheduling: Faktor 3 möglich

Superskalar – Probleme (cont.)

Softwareunterstützung für Pipelining superskalarer Prozessoren
„Software Pipelining“

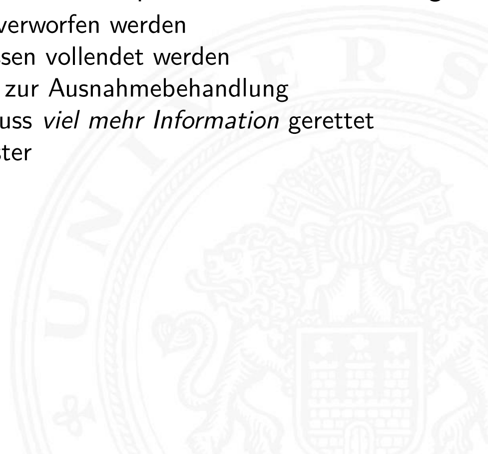
- Codeoptimierungen beim Compilieren als Ersatz/Ergänzung zur Pipelineunterstützung durch Hardware
- Compiler hat „globalen“ Überblick

⇒ zusätzliche Optimierungsmöglichkeiten

Superskalar – Ausnahmebehandlung

Interrupts, System-Calls und Exceptions

- Pipeline kann normalen Ablauf nicht fortsetzen
- Ausnahmebehandlung ist wegen der Vielzahl paralleler Aktionen und den Abhängigkeiten innerhalb der Pipelines extrem aufwändig
  - einige Anweisungen können verworfen werden
  - andere Pipelineaktionen müssen vollendet werden
    benötigt *zusätzliche Zeit* bis zur Ausnahmebehandlung
  - wegen Register-Renaming muss *viel mehr Information* gerettet werden als nur die ISA-Register



Superskalar – Ausnahmebehandlung (cont.)

Prinzip der Interruptbehandlung

- keine neuen Instruktionen mehr initiieren
- warten bis Instruktionen des Reorder-Buffers abgeschlossen sind
- 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 auslösenden Instruktion ist bekannt
  - alle Instruktionen bis zur PC-Instr. wurden vollständig ausgeführt
  - keine Instruktion nach der PC-Instr. wurde ausgeführt
  - Ausführungszustand der PC-Instruktion ist bekannt

Beispiel: Pentium 4 / NetBurst Architektur

- superskalare Architektur (mehrere ALUs)
- CISC-Befehle werden dynamisch in „μOPs“ (1 . . . 3) umgesetzt
- Ausführung der μOPs mit „Out of Order“ Maschine, wenn
  - Operanden verfügbar sind
  - funktionelle Einheit (ALU) frei ist
- Ausführung wird durch „Reservation Stations“ kontrolliert
  - beobachtet die Datenabhängigkeiten zwischen μOPs
  - teilt Ressourcen zu
- „Trace“ Cache
  - ersetzt traditionellen Anweisungscache
  - speichert Anweisungen in decodierter Form: Folgen von μOPs
  - reduziert benötigte Rate für den Anweisungsdecoder
- „Double pumped“ ALUs (2 Operationen pro Taktzyklus)

Beispiel: Pentium 4 / NetBurst Architektur (cont.)

- große Pipelinelänge ⇒ sehr hohe Taktfrequenzen

Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

- umfangreiches Material von Intel unter:
  `ark.intel.com`, `www.intel.com`

Beispiel: Pentium 4 / NetBurst Architektur (cont.)

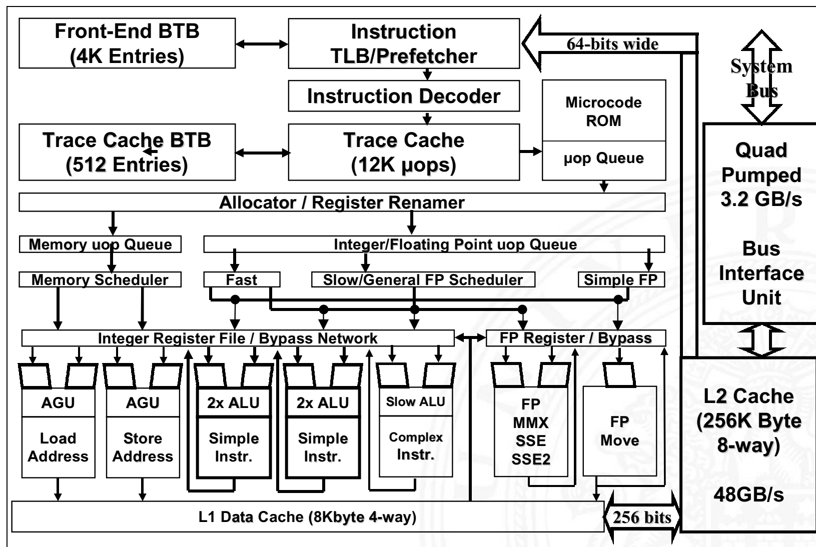
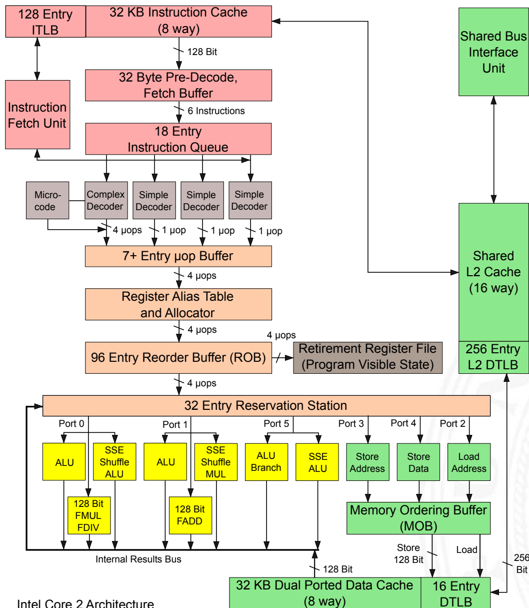


Figure 4: Pentium® 4 processor microarchitecture

Intel: Q1, 2001

Beispiel: Core 2 Architektur

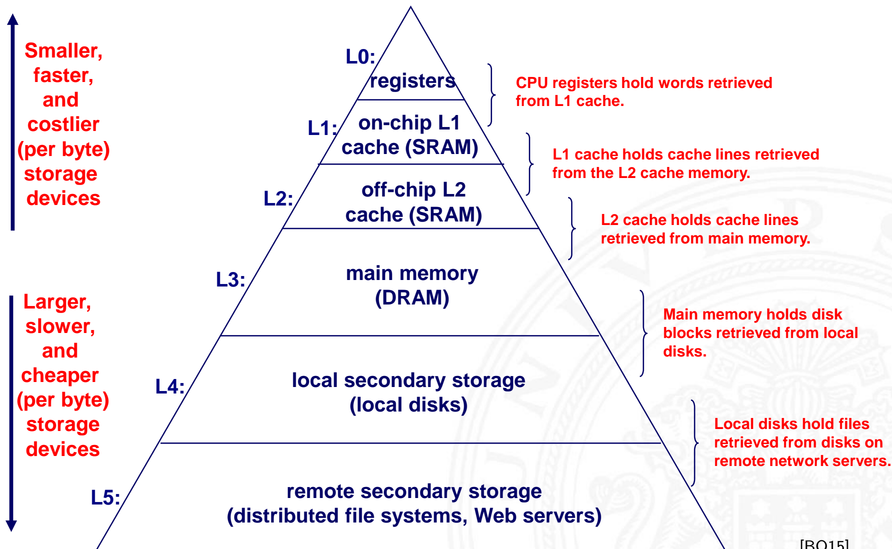


Intel Core 2 Architecture

Gliederung

1. Rechnerarchitektur
2. Bewertung von Architekturen und Rechnersystemen
3. Instruction Set Architecture
4. Pipelining
5. Speicherhierarchie
    - Speichertypen
    - Halbleiterspeicher
    - spezifische Eigenschaften
    - Motivation
    - Cache Speicher
    - Virtueller Speicher

Speicherhierarchie: Konzept



Speicherhierarchie: Konzept (cont.)

Gesamtsystem kombiniert verschiedene Speicher

- wenige KByte Register (-bank) im Prozessor
- einige MByte SRAM als schneller Zwischenspeicher
- einige GByte DRAM als Hauptspeicher
- einige TByte Festplatte als nichtflüchtiger Speicher
- Hintergrundspeicher (CD/DVD/BR, Magnetbänder)
- das WWW und Cloud-Services

Kompromiss aus Kosten, Kapazität, Zugriffszeit

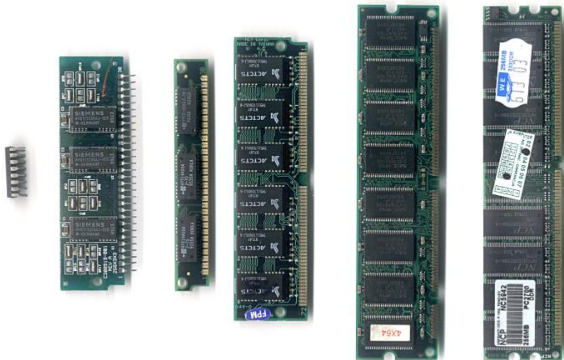
- Illusion aus großem schnellem Speicher
- funktioniert nur wegen räumlicher/zeitlicher Lokalität

L0: Register

- Register im Prozessor integriert
  - Program-Counter und Datenregister für Programmierer sichtbar
  - ggf. weitere Register für Systemprogrammierung
  - zusätzliche unsichtbare Register im Steuerwerk
- Flipflops oder Registerbank mit 6 Trans.-Speicherzellen
  - Lesen und Schreiben in jedem Takt möglich
  - ggf. mehrere parallele Lesezugriffe in jedem Takt
  - Zugriffszeiten ca. 100 ps
- typ. Größe einige KByte, z.B. 16 Register á 64-bit *x86-64*

L1-L3: Halbleiterspeicher RAM

- „*Random-Access Memory*“ (RAM) aufgebaut aus Mikrochips
- Grundspeichereinheit ist eine Zelle (ein Bit pro Zelle)
- SRAM (6T-Zelle) oder DRAM (1T-Zelle) Technologie
- mehrere RAM Chips bilden einen Speicher



L4: Festplatten

- dominierende Technologie für nichtflüchtigen Speicher
- hohe Speicherkapazität, derzeit einige TB
  - Daten bleiben beim Abschalten erhalten
  - aber langsamer Zugriff
  - besondere Algorithmen, um langsamen Zugriff zu verbergen

  - Einsatz als Speicher für dauerhafte Daten
  - Einsatz als erweiterter Hauptspeicher („*virtual memory*“)

- FLASH/SSD zunehmend als Ersatz für Festplatten
  - Halbleiterspeicher mit sehr effizienten multibit-Zellen
  - Verwaltung (derzeit) wie Festplatten
  - signifikant schnellere Zugriffszeiten

L5: Hintergrundspeicher

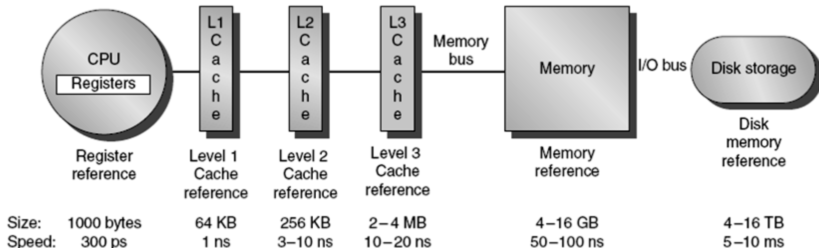
- enorme Speicherkapazität
- langsame Zugriffszeiten

- Archivspeicher und Backup für (viele) Festplatten
  - Magnetbänder
  - RAID-Verbund aus mehreren Festplatten
  - optische Datenspeicher: CD-ROM, DVD-ROM, BlueRay

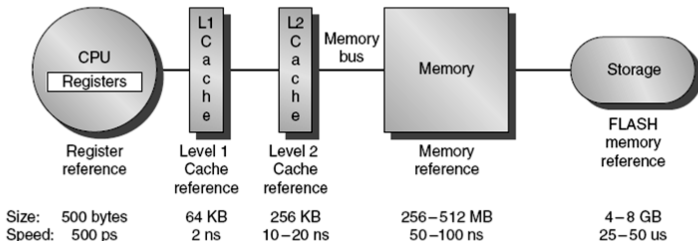
- WWW und Internet-Services, Cloud-Services
  - Cloud-Farms ggf. ähnlich schnell wie L4 Festplatten, da Netzwerk schneller als der Zugriff auf eine lokale Festplatte

- in dieser Vorlesung nicht behandelt

Speicherhierarchie: zwei Beispiele



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

[HP17]

Random-Access Memory / RAM

SRAM „statisches RAM“

- jede Zelle speichert Bit mit einer 6-Transistor Schaltung
- speichert Wert solange er mit Energie versorgt wird
- unanfällig für Störungen wie elektrische Brummspannungen
- schneller und teurer als DRAM

DRAM „dynamisches RAM“

- jede Zelle speichert Bit mit 1 Kondensator und 1 Transistor
- der Wert muss alle 10-100 ms aufgefrischt werden
- anfällig für Störungen
- langsamer und billiger als SRAM

SRAM vs. DRAM

	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	1 €/Mbit	0,1 Ct./Mbit

	Tran. per bit	Access time	Persist?	Sensitive?	Cost	Applications
SRAM	6	1X	Yes	No	100x	cache memories
DRAM	1	10X	No	Yes	1X	Main memories, frame buffers

[BO15]

Nichtflüchtige Speicher

- DRAM und SRAM sind flüchtige Speicher
  - Informationen gehen beim Abschalten verloren
- nichtflüchtige Speicher speichern Werte selbst wenn sie spannungslos sind
  - allgemeiner Name ist „Read-Only-Memory“ (ROM)
  - irreführend, da einige ROMs auch verändert werden können
- Arten programmierbarer ROMs
  - PROM: „Programmable ROM“
  - EPROM: „Eraseable Programmable ROM“ UV Licht Löschen
  - EEPROM: „Electrically Eraseable PROM“ elektrisch Löschen
  - Flash Speicher (hat inzwischen die meisten PROMs ersetzt)

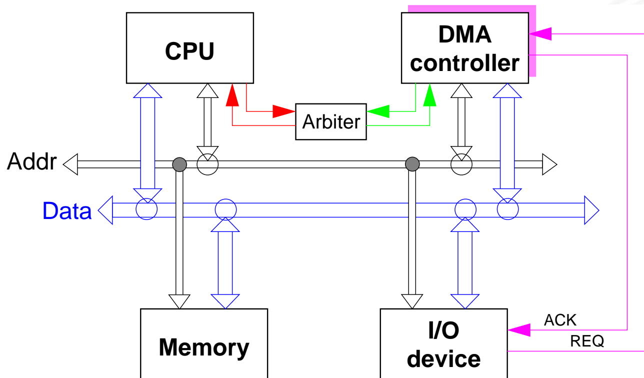
Anwendungen für nichtflüchtigen Speicher

- Firmware
- Programm wird in einem ROM gespeichert
  - Boot Code, BIOS („Basic Input/Output System“)
  - Grafikkarten, Festplattencontroller
  - Eingebettete Systeme

Speicheranbindung – DMA

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

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

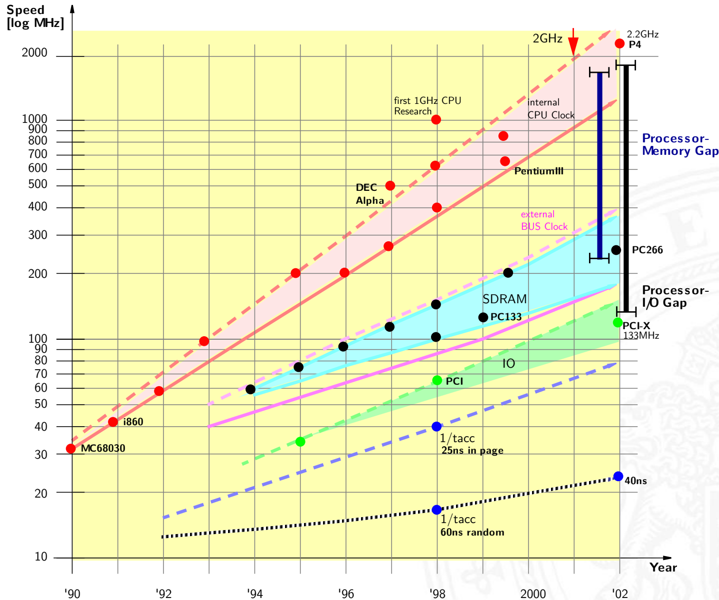


Eigenschaften der Speichertypen

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	SSD
Latenz	8 ns	4 ms	0,2/0,4 ms
Bandbreite	25,6 GB/sec (pro Kanal, bis 4)	250 MB/sec	3/2 GB/sec (r/w)
Kosten/GB	8 €	3 ct. 1 TB: 30 €	30 ct.

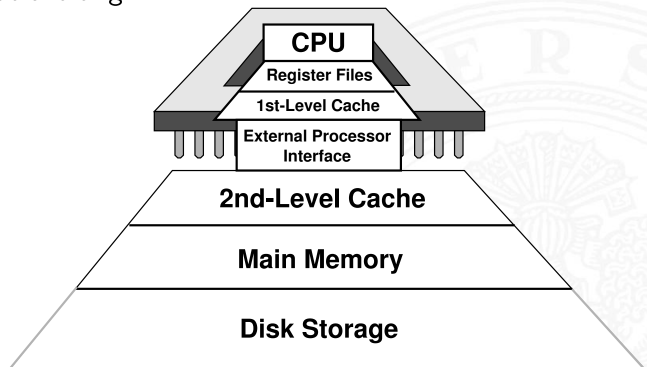
Prozessor-Memory Gap



Speicherhierarchie

Motivation

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



Speicherhierarchie (cont.)

- schnelle vs. langsame Speichertechnologie

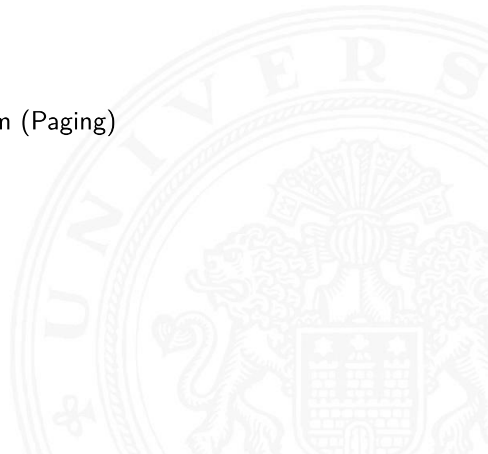
schnell	: hohe	Kosten/Byte	geringe Kapazität
langsam	: geringe	–"–	hohe –"–

- wachsender Abstand zwischen CPU und Speichergeschwindigkeit
  - Prozessor läuft mit einigen GHz Takt
  - Register können mithalten, aber nur einige KByte Kapazität
  - DRAM braucht 60 . . . 100 ns für Zugriff: 100 × langsamer
  - Festplatte braucht 10 ms für Zugriff: 1 000 000 × langsamer
- Lokalität der Programme wichtig
  - aufeinanderfolgende Speicherzugriffe sind meistens „lokal“
  - gut geschriebene Programme haben meist eine gute Lokalität

⇒ Motivation für spezielle Organisation von Speichersystemen **Speicherhierarchie**

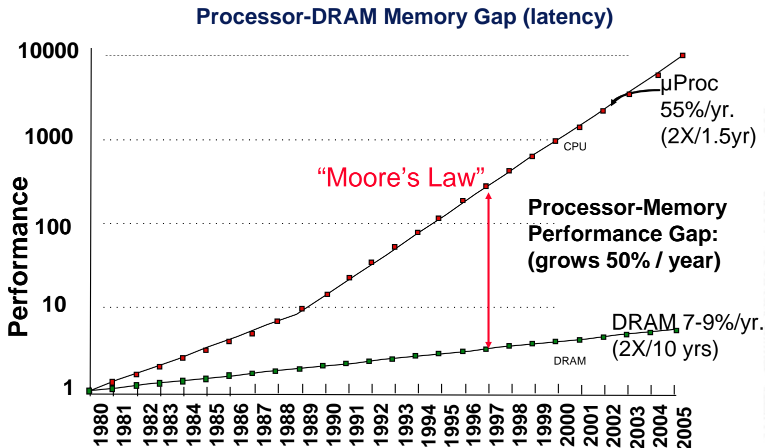
Verwaltung der Speicherhierarchie

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



Cache

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



[PH20]

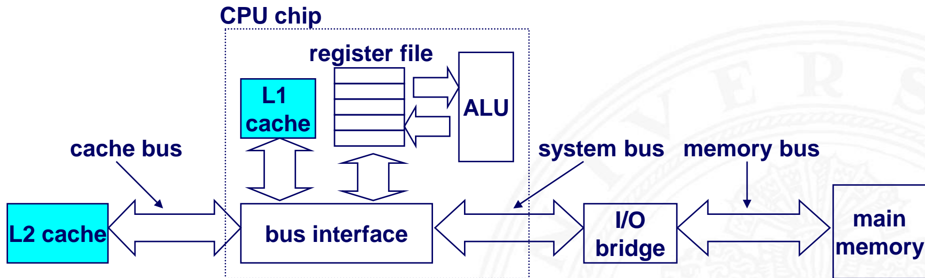
⇒ Cache als schneller Zwischenspeicher zum Hauptspeicher

Cache (cont.)

- technische Realisierung: SRAM
- transparenter Speicher
  - Cache ist für den Programmierer nicht sichtbar!
  - wird durch Hardware verwaltet
- ggf. getrennte Caches für Befehle und Daten
- enthält Hauptspeicherblöcke mit erhöhter Zugriffswahrscheinlichkeit
- basiert auf Prinzip der Lokalität von Speicherzugriffen durch ein laufendes Programm
  - ca. 80% der Zugriffe greifen auf 20% der Adressen zu
  - manchmal auch 90% / 10% oder noch besser
- https://de.wikipedia.org/wiki/Cache
  https://en.wikipedia.org/wiki/CPU_cache
  https://en.wikipedia.org/wiki/Cache_(Computing)

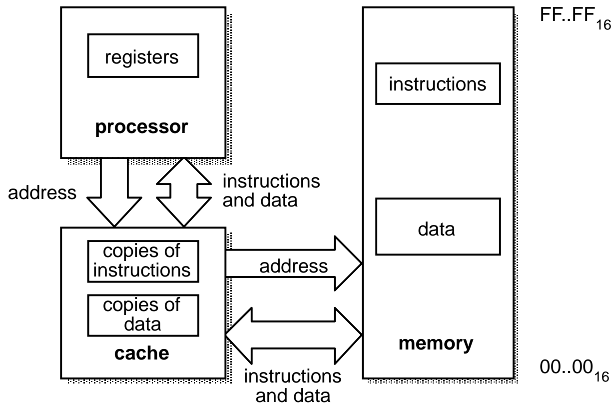
Cache (cont.)

- CPU referenziert Adresse
  - parallele Suche in L1 (level 1), L2... und Hauptspeicher
  - erfolgreiche Suche liefert Datum, Abbruch laufender Suchen



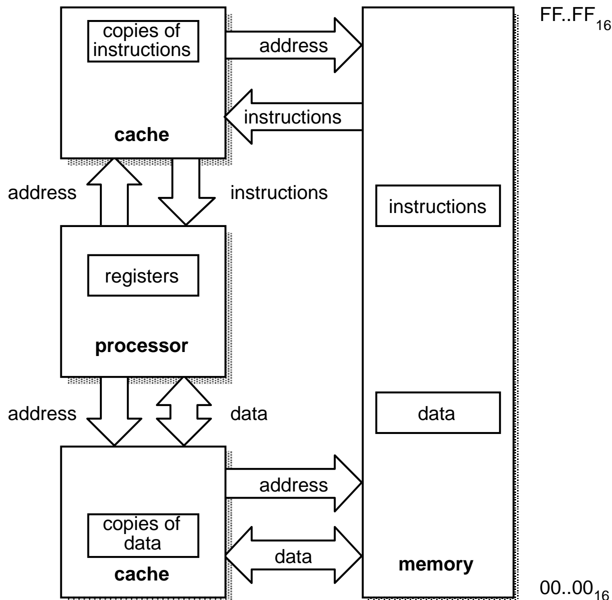
[BO15]

gemeinsamer Cache / „unified Cache“



[Fur00]

separate Instruction-/Data Caches

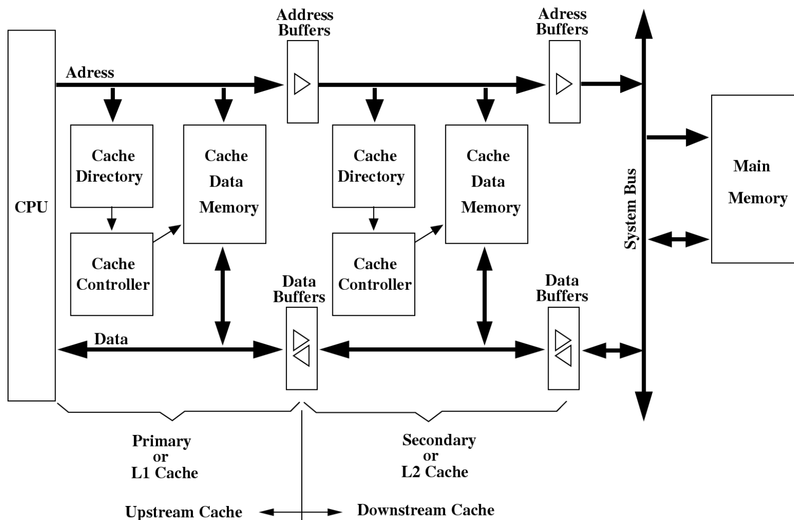


FF..FF$_{16}$

00..00$_{16}$ [Fur00]

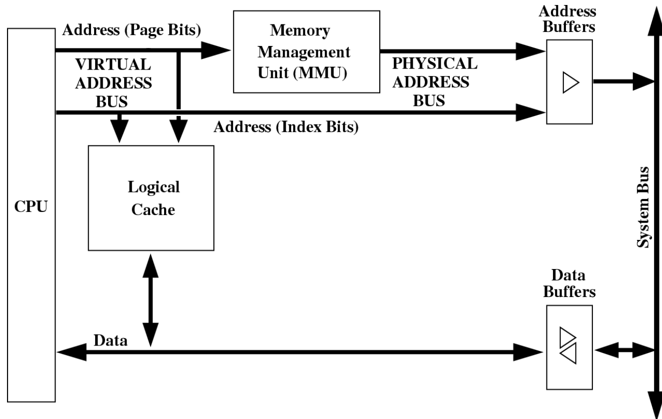
Cache – Position

- First- und Second-Level Cache



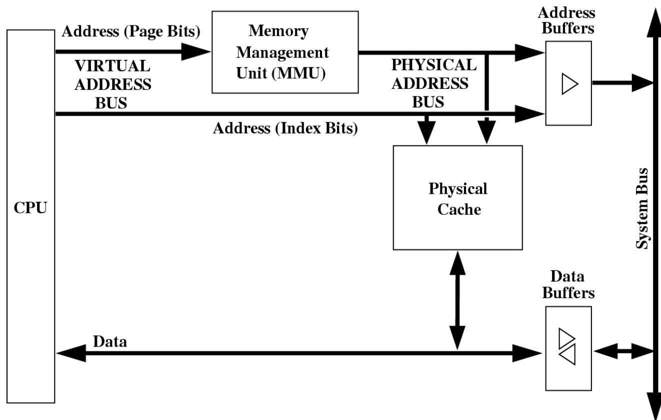
Cache – Position (cont.)

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



Cache – Position (cont.)

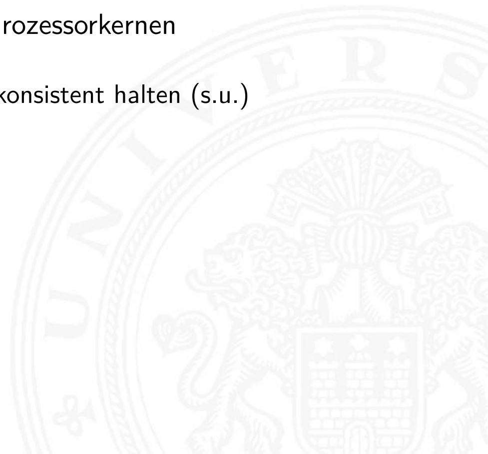
- Physikalischer Cache
  - + Cache muss nie geleert werden
  - – Adressumrechnung durch MMU immer nötig



Cache – Position (cont.)

- typische Cache Organisation
  - First-Level Cache: getrennte Instruktions- und Daten-Caches
  - Second-Level Cache: gemeinsamer Cache je Prozessorkern
  - Third-Level Cache: gemeinsamer Cache für alle Prozessorkerne
- bei mehreren Prozessoren / Prozessorkernen

  ⇒ Cache-Kohärenz wichtig
  - gemeinsam genutzte Daten konsistent halten (s.u.)

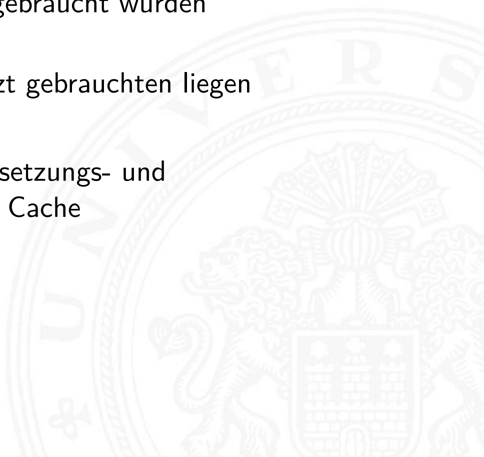


Cache – Strategie

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 Rückschreibestategien für den Cache



Cache – Performanz

Cacheperformanz

- Begriffe

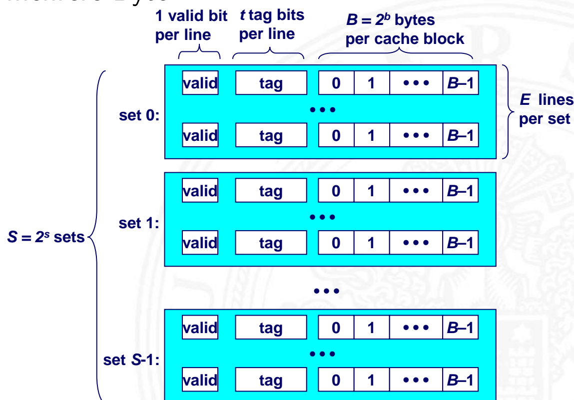
Treffer (Hit)		Zugriff auf Datum, ist bereits im Cache
Fehler (Miss)		–"– ist nicht –"–
Treffer-Rate	R_{Hit}	Wahrscheinlichkeit, Datum ist im Cache
Fehler-Rate	R_{Miss}	$1 - R_{Hit}$
Hit-Time	T_{Hit}	Zeit, bis Datum bei Treffer geliefert wird
Miss-Penalty	T_{Miss}	zusätzlich benötigte Zeit bei Fehler

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

  $T_{Hit} = 1\,\text{Takt},\ T_{Miss} = 20\,\text{Takte},\ R_{Miss} = 5\,\%$
  $\Rightarrow$ Mittlere Speicherzugriffszeit $= 2$ Takte

Cache Organisation

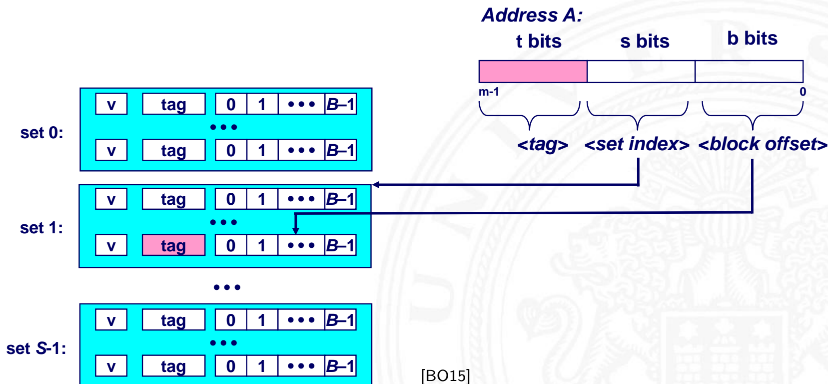
- Cache ist ein Array von Speicher-Bereichen („sets“)
- jeder Bereich enthält eine oder mehrere Zeilen
- jede Zeile enthält einen Datenblock
- jeder Block enthält mehrere Byte



[BO15]

Adressierung von Caches

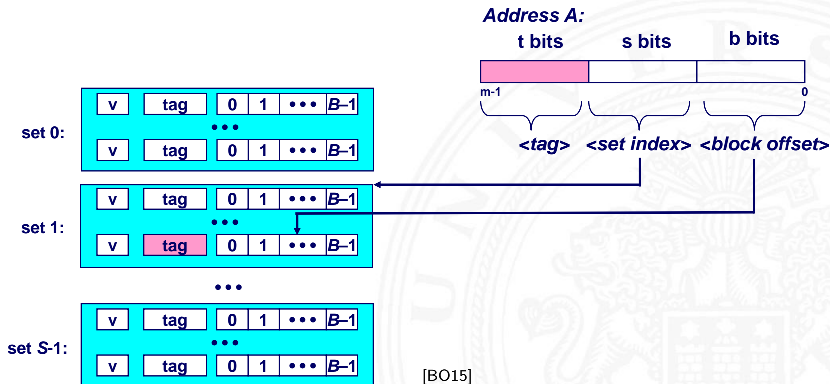
- Adressteil ⟨*set index*⟩ von *A* bestimmt Bereich („set“)
- Adresse *A* ist im Cache, wenn
  1. Cache-Zeile ist als gültig markiert („valid“)
  2. Adressteil ⟨*tag*⟩ von $A =$ „tag“ Bits des Bereichs



[BO15]

Adressierung von Caches (cont.)

- Cache-Zeile („cache line“) enthält Datenbereich von 2^b Byte
- gesuchtes Wort mit Offset $\langle$*block offset*$\rangle$



[BO15]

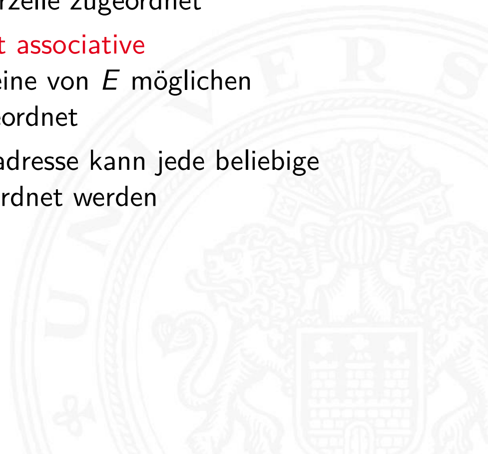
Cache – Organisation

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

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

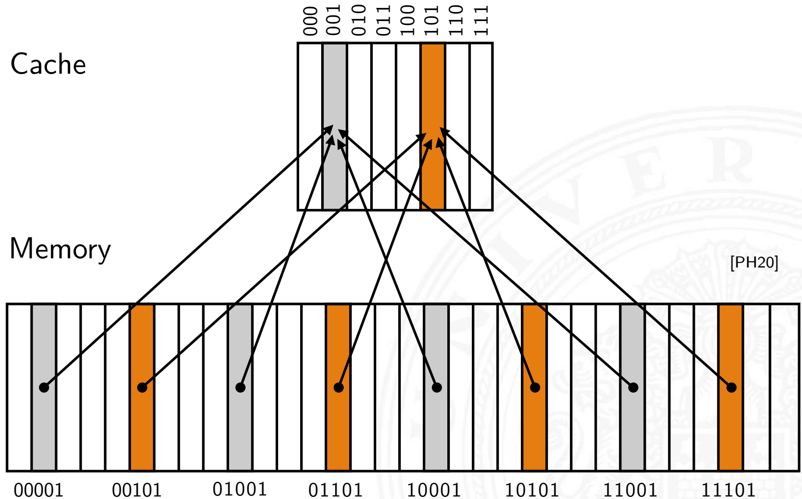
n-fach bereichsassoziativ / set associative
: jeder Speicheradresse ist eine von E möglichen Cache-Speicherzellen zugeordnet

voll-assoziativ
: jeder Speicheradresse kann jede beliebige Cache-Speicherzelle zugeordnet werden



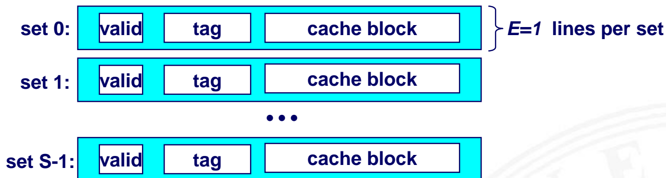
Cache: direkt abgebildet / „direct mapped“

- jeder Adresse ist genau eine Speicherzelle im Cache zugeordnet



Cache: direkt abgebildet / „direct mapped“ (cont.)

- verfügt über genau 1 Zeile pro Bereich S Bereiche (**S**ets)



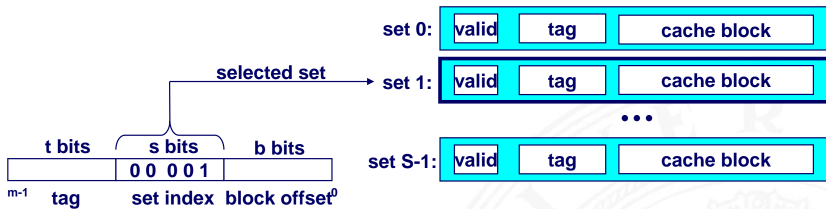
[BO15]

+ einfachste Cache-Art
+ große Caches möglich
− Effizienz, z.B. Zugriffe auf A, $A + n \cdot S$ …
  $\Rightarrow$ *„Cache Thrashing“*

Cache: direkt abgebildet / „direct mapped“ (cont.)

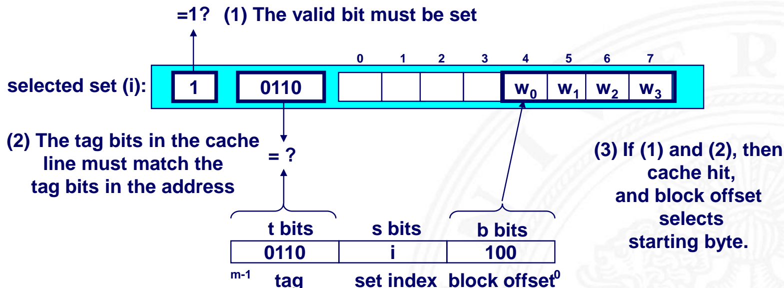
Zugriff auf direkt abgebildete Caches

1. Bereichsauswahl durch Bits ⟨*set index*⟩



[BO15]

2. ⟨*valid*⟩: sind die Daten gültig?
3. „Line matching“: stimmt ⟨*tag*⟩ überein?
4. Wortselektion extrahiert Wort unter Offset ⟨*block offset*⟩

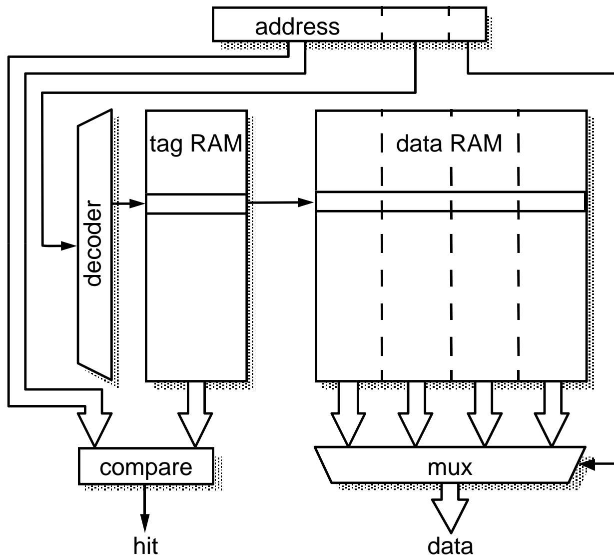


[BO15]

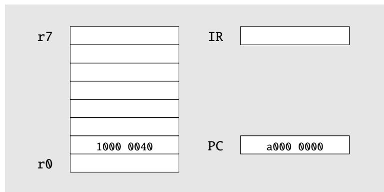
Cache: direkt abgebildet / „direct mapped“ (cont.)

Prinzip

[Fur00]

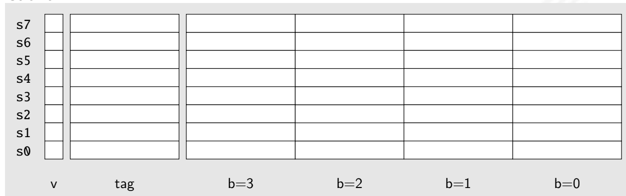


Direct mapped cache: Beispiel – leer

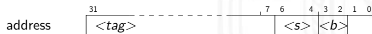


CPU

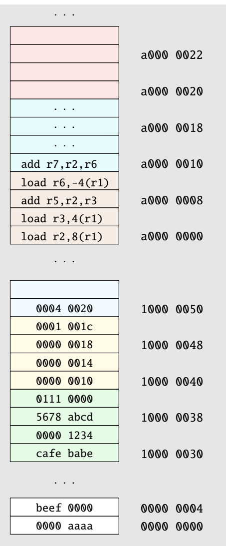
Cache



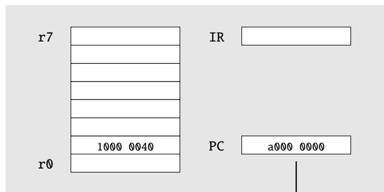
address

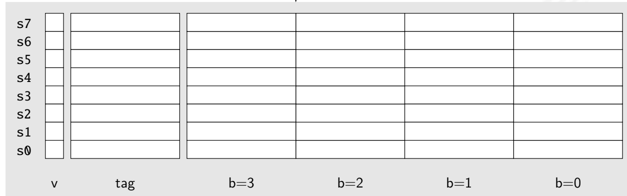


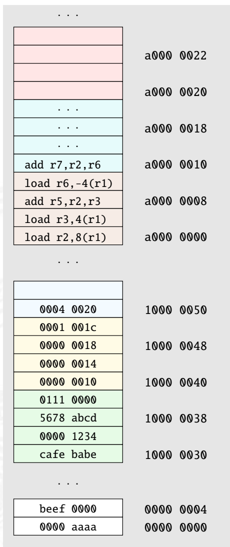
Memory



Direct mapped cache: Beispiel – fetch miss

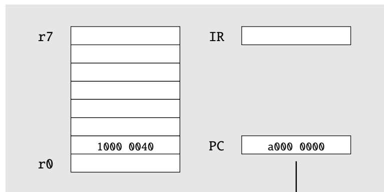


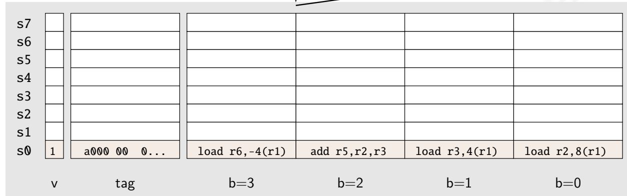


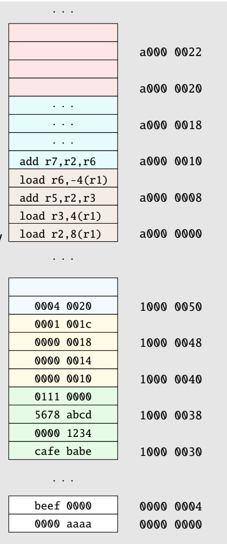


`load r2, 8(r1)` fetch cache miss (empty, all invalid)

Direct mapped cache: Beispiel – fetch fill

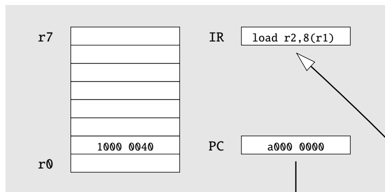


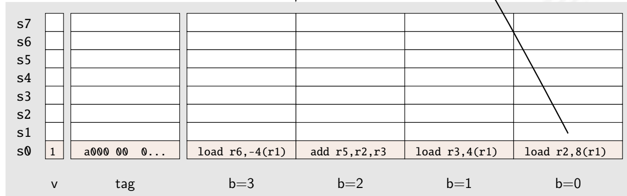


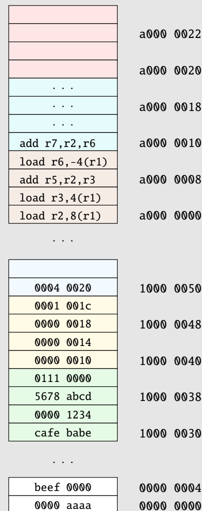


load r2, 8(r1)   fetch   fill cache set s0 from memory

Direct mapped cache: Beispiel – fetch



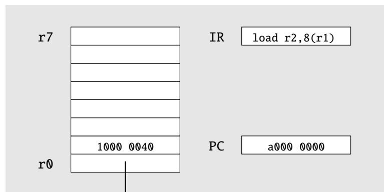




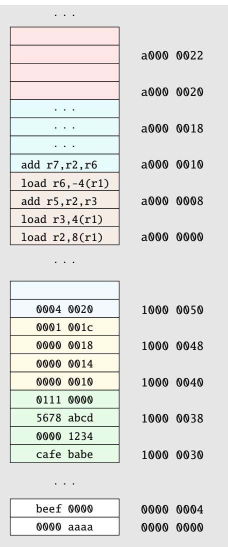
load r2, 8(r1) fetch load instruction into IR

Direct mapped cache: Beispiel – execute miss

CPU



Memory

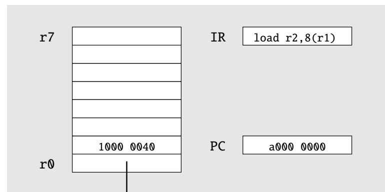


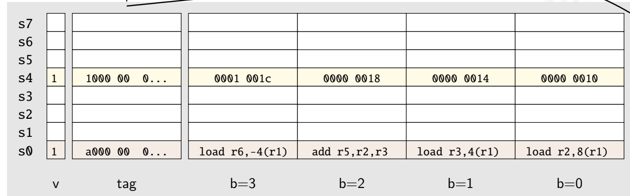
Cache

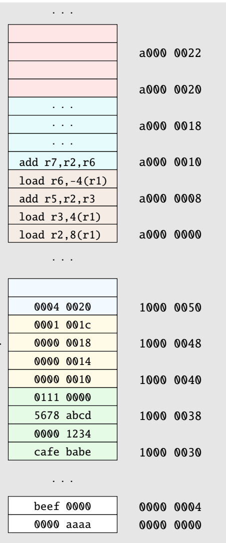
	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4						
s3						
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

load r2, 8(r1)	fetch	load instruction into IR
	execute	cache miss

Direct mapped cache: Beispiel – execute fill

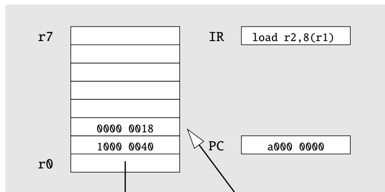






`load r2, 8(r1)`	fetch	load instruction into IR
	execute	fill cache set s4 from memory

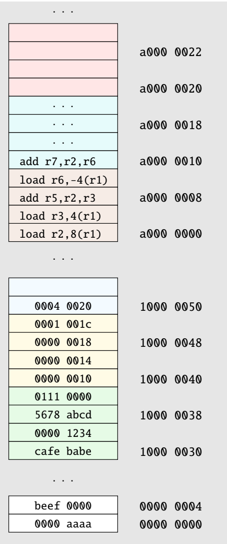
Direct mapped cache: Beispiel – execute



CPU

1000 0048

Memory

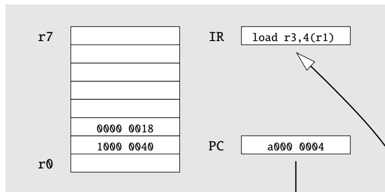


Cache

	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3						
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

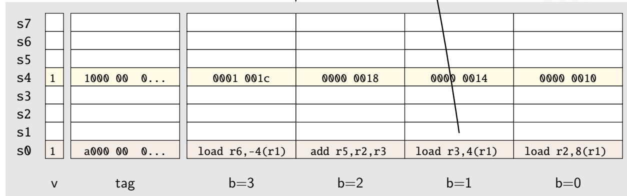
`load r2, 8(r1)`	fetch	load instruction into IR
	execute	load value into r2

Direct mapped cache: Beispiel – fetch hit

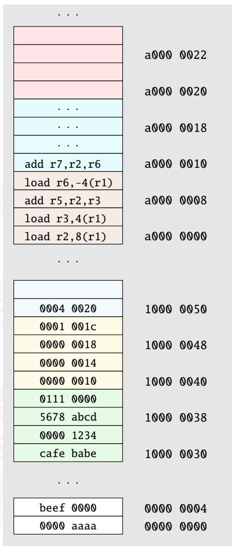


CPU

Cache

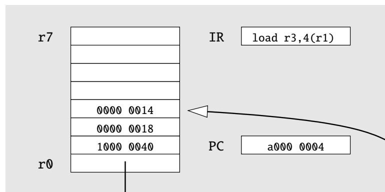


Memory



load r3, 4(r1)   fetch   cache hit, load instruction into IR

Direct mapped cache: Beispiel – execute hit



CPU

Cache

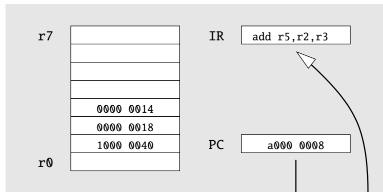
	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3						
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

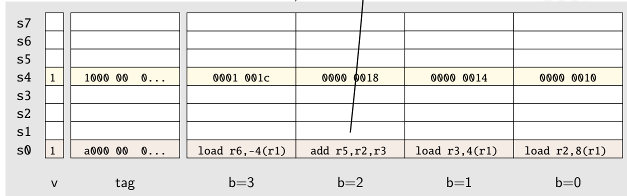
Memory

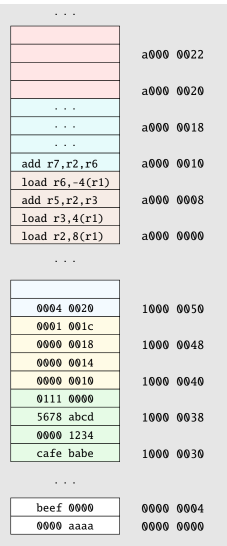
Content	Address
. . .	
	a000 0022
	a000 0020
. . .	
. . .	a000 0018
. . .	
add r7,r2,r6	a000 0010
load r6,-4(r1)	
add r5,r2,r3	a000 0008
load r3,4(r1)	
load r2,8(r1)	a000 0000
. . .	
0004 0020	1000 0050
0001 001c	
0000 0018	1000 0048
0000 0014	
0000 0010	1000 0040
0111 0000	
5678 abcd	1000 0038
0000 1234	
cafe babe	1000 0030
. . .	
beef 0000	0000 0004
0000 aaaa	0000 0000

load r3, 4(r1)	fetch	cache hit, load instruction into IR
	execute	cache hit, load value into r3

Direct mapped cache: Beispiel – fetch hit

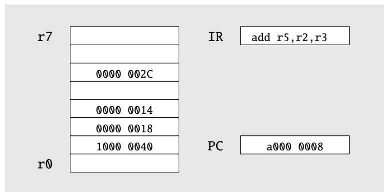




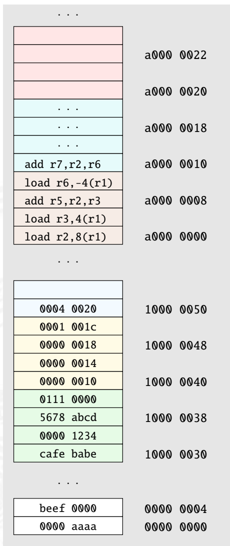


add r5,r2,r3 fetch cache hit, load instruction into IR

Direct mapped cache: Beispiel – execute hit



CPU

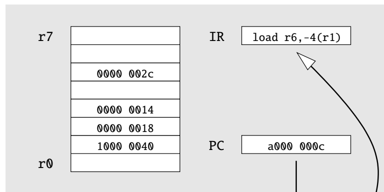


Cache

	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3						
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

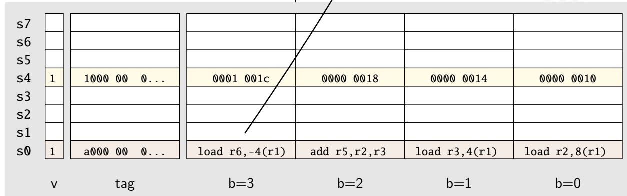
add r5,r2,r3	fetch	cache hit, load instruction into IR
	execute	no memory access

Direct mapped cache: Beispiel – fetch hit

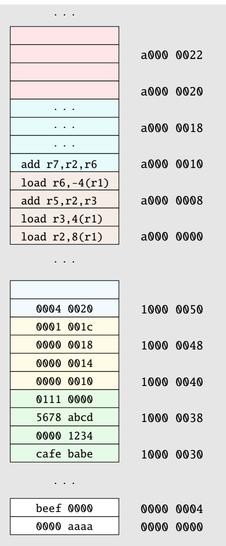


CPU

Cache

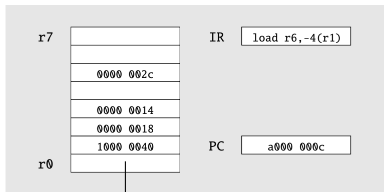


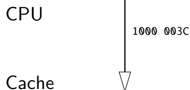
Memory



load r6,-4(r1) fetch cache hit, load instruction into IR

Direct mapped cache: Beispiel – execute miss





Cache

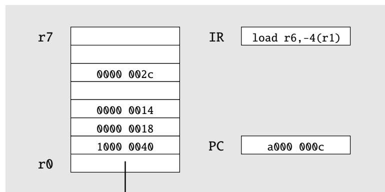
	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3						
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

Memory

Inhalt	Adresse
. . .	
	a000 0022
	a000 0020
. . .	
. . .	a000 0018
. . .	
add r7,r2,r6	a000 0010
load r6,-4(r1)	
add r5,r2,r3	a000 0008
load r3,4(r1)	
load r2,8(r1)	a000 0000
. . .	
0004 0020	1000 0050
0001 001c	
0000 0018	1000 0048
0000 0014	
0000 0010	1000 0040
0111 0000	
5678 abcd	1000 0038
0000 1234	
cafe babe	1000 0030
. . .	
beef 0000	0000 0004
0000 aaaa	0000 0000

load r6,-4(r1)	fetch	cache hit, load instruction into IR
	execute	cache miss

Direct mapped cache: Beispiel – execute fill



CPU

1000 003C

Cache

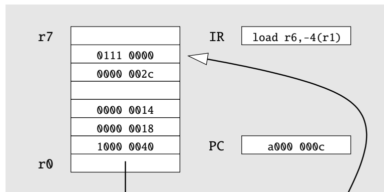
	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3	1	1000 00 0...	0111 0000	5678 abcd	0000 1234	cafe babe
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

Memory

Content	Address
...	
	a000 0022
	a000 0020
...	
...	a000 0018
...	
add r7,r2,r6	a000 0010
load r6,-4(r1)	
add r5,r2,r3	a000 0008
load r3,4(r1)	
load r2,8(r1)	a000 0000
...	
0004 0020	1000 0050
0001 001c	
0000 0018	1000 0048
0000 0014	
0000 0010	1000 0040
0111 0000	
5678 abcd	1000 0038
0000 1234	
cafe babe	1000 0030
...	
beef 0000	0000 0004
0000 aaaa	0000 0000

load r6,-4(r1)	fetch	cache hit, load instruction into IR
	execute	fill cache set s3 from memory

Direct mapped cache: Beispiel – execute



CPU

1000 003C

Cache

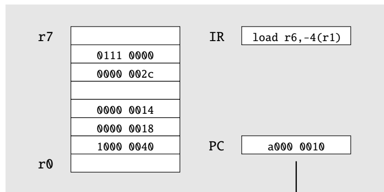
	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3	1	1000 00 0...	0111 0000	5678 abcd	0000 1234	cafe babe
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

Memory

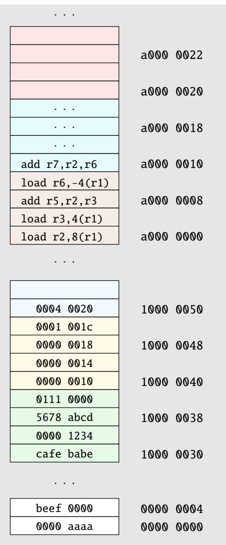
Inhalt	Adresse
...	
	a000 0022
	a000 0020
...	
...	a000 0018
...	
add r7,r2,r6	a000 0010
load r6,-4(r1)	
add r5,r2,r3	a000 0008
load r3,4(r1)	
load r2,8(r1)	a000 0000
...	
0004 0020	1000 0050
0001 001c	
0000 0018	1000 0048
0000 0014	
0000 0010	1000 0040
0111 0000	
5678 abcd	1000 0038
0000 1234	
cafe babe	1000 0030
...	
beef 0000	0000 0004
0000 aaaa	0000 0000

load r6,-4(r1)	fetch	cache hit, load instruction into IR
	execute	load value into r6

Direct mapped cache: Beispiel – fetch miss



CPU

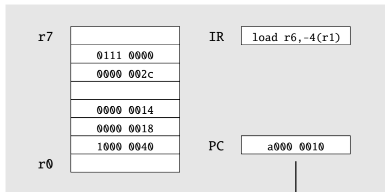


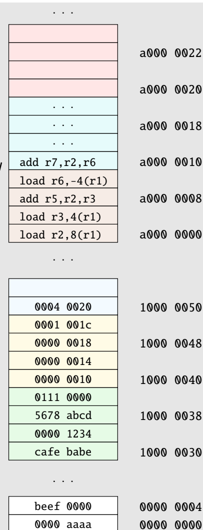
Cache

	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3	1	1000 00 0...	0111 0000	5678 abcd	0000 1234	cafe babe
s2						
s1						
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

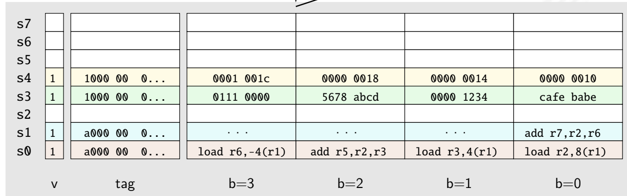
add r7,r2,r6 fetch cache miss

Direct mapped cache: Beispiel – fetch fill





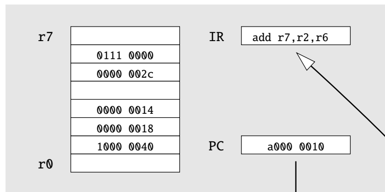
Cache



	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3	1	1000 00 0...	0111 0000	5678 abcd	0000 1234	cafe babe
s2						
s1	1	a000 00 0...	...	...	...	add r7,r2,r6
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

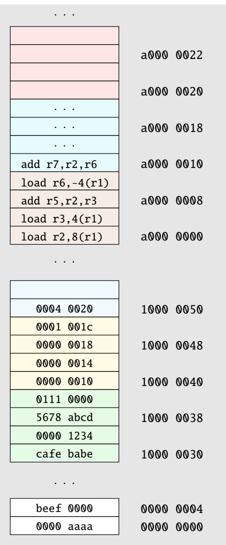
add r7,r2,r6 fetch fill cache set s1 from memory

Direct mapped cache: Beispiel – fetch



CPU

Memory



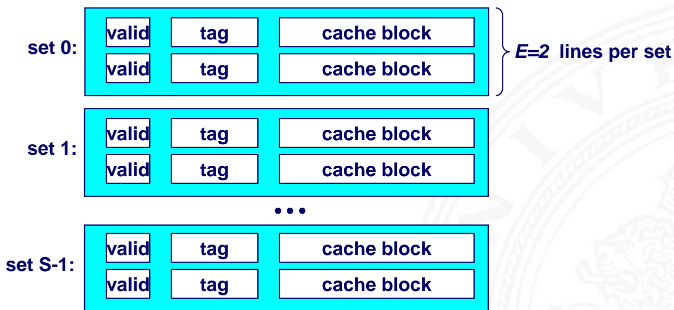
Cache

	v	tag	b=3	b=2	b=1	b=0
s7						
s6						
s5						
s4	1	1000 00 0...	0001 001c	0000 0018	0000 0014	0000 0010
s3	1	1000 00 0...	0111 0000	5678 abcd	0000 1234	cafe babe
s2						
s1	1	a000 00 0...	. . .	. . .	. . .	add r7,r2,r6
s0	1	a000 00 0...	load r6,-4(r1)	add r5,r2,r3	load r3,4(r1)	load r2,8(r1)

add r7,r2,r6 fetch load instruction into IR

Cache: bereichsassoziativ / „set associative“

- jeder Speicheradresse ist ein Bereich S mit mehreren (E) Cachezeilen zugeordnet
- n-fach assoziative Caches: E=2, 4 …
  „2-way set associative cache“, „4-way …“

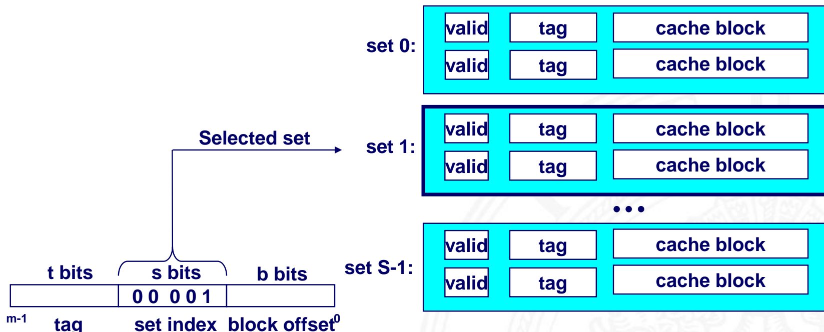


[BO15]

Cache: bereichsassoziativ / „set associative“ (cont.)

Zugriff auf n-fach assoziative Caches

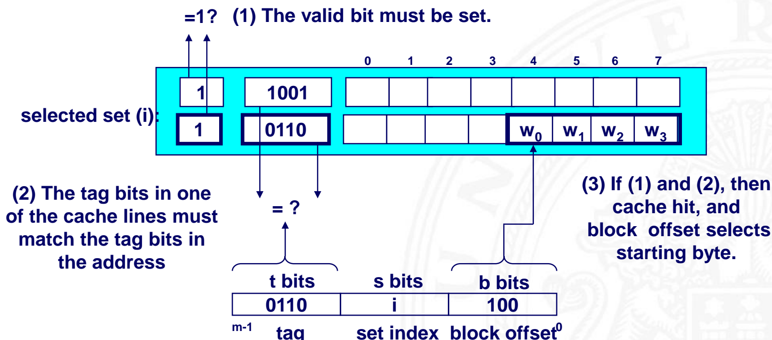
1. Bereichsauswahl durch Bits ⟨*set index*⟩



[BO15]

Cache: bereichsassoziativ / „set associative“ (cont.)

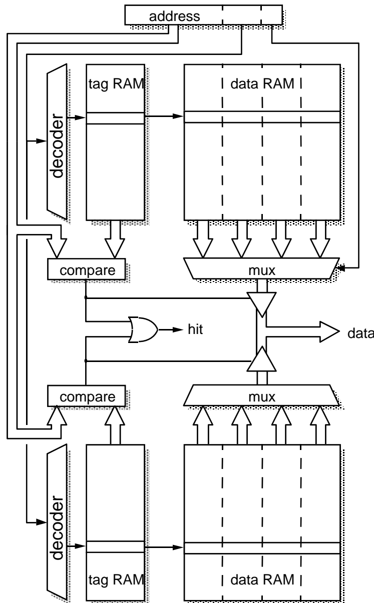
2. ⟨*valid*⟩: sind die Daten gültig?
3. „Line matching“: Cache-Zeile mit passendem ⟨*tag*⟩ finden? dazu Vergleich aller „tags“ des Bereichs ⟨*set index*⟩
4. Wortselektion extrahiert Wort unter Offset ⟨*block offset*⟩



[BO15]

Cache: bereichsassoziativ / „set associative“ (cont.)

Prinzip

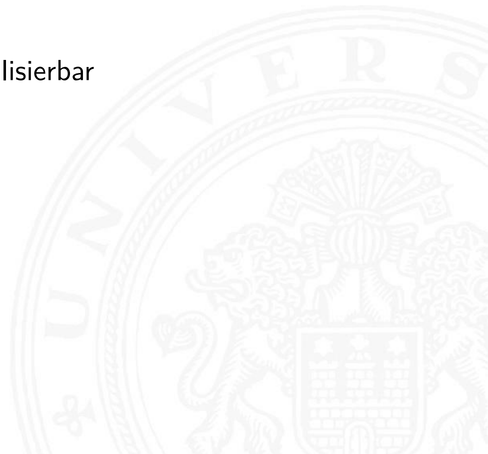


[Fur00]

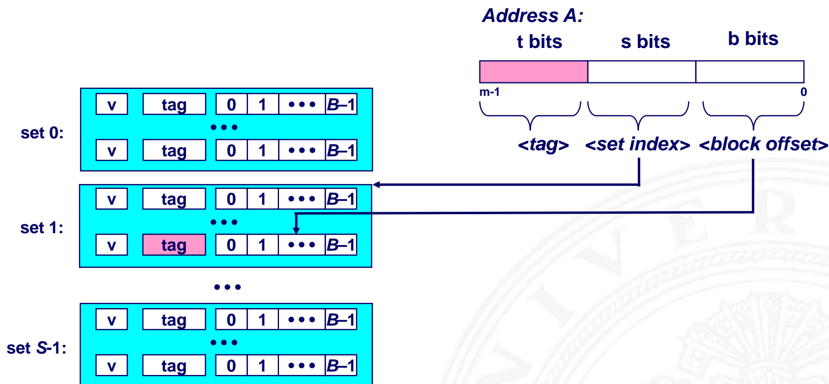
Cache: voll-assoziativ

- jeder Adresse des Speichers kann jede beliebige Cachezeile zugeordnet werden
- Spezialfall: nur ein Cachebereich S

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



Cache – Dimensionierung



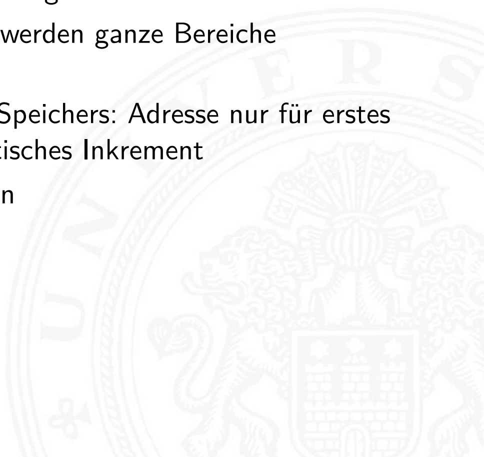
[BO15]

- Parameter: S, B, E
- Cache speichert immer größere Blöcke / „*Cache-Line*“
- Wortauswahl durch ⟨*block offset*⟩ in Adresse

Cache – Dimensionierung (cont.)

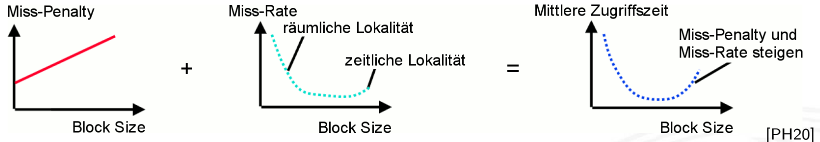
Vor- und Nachteile des Cache

- \+ nutzt „räumliche Lokalität“ aus
  Speicherzugriffe von Programmen (Daten und Instruktionen) liegen in ähnlichen/aufeinanderfolgenden Adressbereichen
- \+ breite externe Datenbusse, es werden ganze Bereiche übertragen
- \+ nutzt Burst-Adressierung des Speichers: Adresse nur für erstes Wort vorgeben, dann automatisches Inkrement
- \+ kürzere interne Cache-Adressen
- – Hardwareaufwand und Kosten



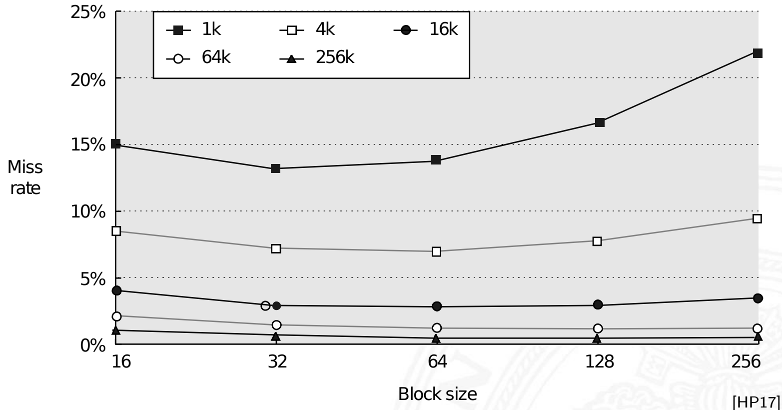
Cache – Dimensionierung (cont.)

Cache- und Block-Dimensionierung



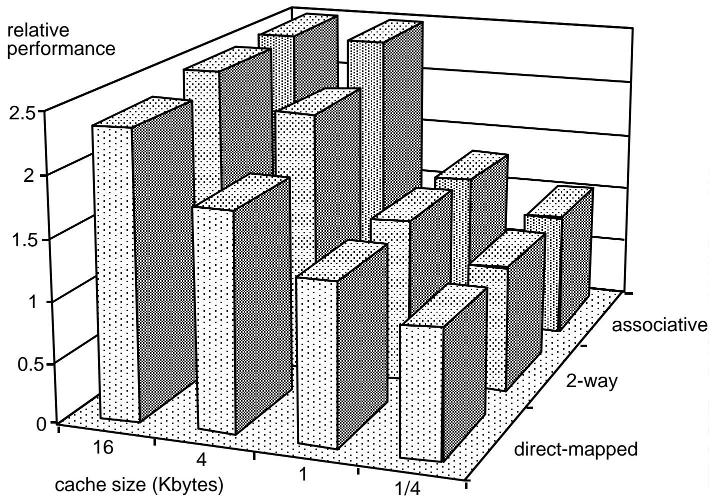
- Blockgröße klein, viele Blöcke
  - + kleinere Miss-Penalty
  - + temporale Lokalität
  - – räumliche Lokalität
- Blockgröße groß, wenig Blöcke
  - – größere Miss-Penalty
  - – temporale Lokalität
  - + räumliche Lokalität

Cache – Dimensionierung (cont.)



- Block-Size: 32... 128 Byte
  L1-Cache: 4... 256 KiByte
  L2-Cache: 256... 4 096 KiByte

Cache – Dimensionierung: relative Performanz



[Fur00]

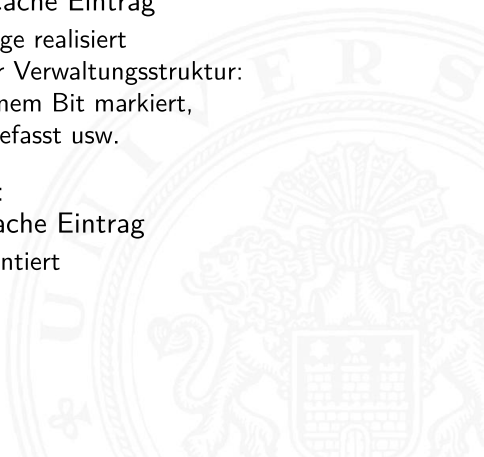
Drei Typen von Cache-Misses

- cold miss
  - Cache ist (noch) leer
- conflict miss
  - wenn die Kapazität des Cache eigentlich ausreicht, aber unterschiedliche Daten in den selben Block abgebildet werden
  - Beispiel für „Trashing“ beim direct-mapped Cache mit S=8: abwechselnder Zugriff auf Blöcke 0, 8, 0, 8, 0, 8 ... ist jedesmal ein Miss
- capacity miss
  - wenn die Menge der aktiven Blöcke („working set“) größer ist als die Kapazität des Cache

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 mehrere Einheiten (Bus-Master: Prozessor, DMA-Controller) auf Speicher zugreifen können: wichtig für *„Symmetric Multiprocessing“*
- Harvard-Architektur hat getrennte Daten- und Instruktions-Speicher
  - Instruktionen sind read-only
  - ⇒ einfacherer Instruktions-Cache
  - ⇒ kein Cache-Kohärenz Problem
- Cache-Kohärenz Protokolle und *„Snooping“*
  - alle Prozessoren(P_1, P_2 ...) überwachen alle Bus-Transaktionen Cache „schnüffelt“ am Speicherbus
  - Prozessor P_2 greift auf Daten zu, die im Cache von P_1 liegen
    P_2 Schreibzugriff ⇒ P_1 Cache aktualisieren / ungültig machen
    P_2 Lesezugriff ⇒ P_1 Cache liefert Daten
  - *Was ist mit gleichzeitige Zugriffen von P_1, P_2?*

Cache-Kohärenz (cont.)

- viele verschiedene Protokolle: Hersteller- / Prozessor-spezifisch
  - SI („*Write Through*“)
  - MSI, MOSI,
  - MESI: *Modified*, *Exclusive*, *Shared*, *Invalid*
  - MOESI: *Modified* (exclusive), *Owned* (Modified shared), *Exclusive*, *Shared*, *Invalid*
  - ...

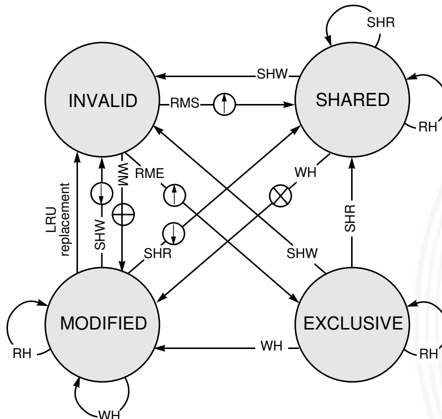
siehe z.B.: `en.wikipedia.org/wiki/Cache_coherence`

MESI Protokoll

- Caches enthalten Wert, Tag und zwei Statusbits für die vier Protokollzustände
  - **M**odified: gültiger Wert, nur in diesem Cache, gegenüber Hauptspeicher-Wert verändert
  - **E**xclusive: gültiger Wert, nur in diesem Cache nicht verändert (unmodified)
  - **S**hared: gültiger Wert, in mehreren Caches vorhanden nicht verändert (unmodified)
  - **I**nvalid: ungültiger Inhalt, Initialzustand
- alle Prozessoren überwachen alle Bus-Transaktionen
- bei Speicherzugriffen Aktualisierung des Status'
- Zugriffe auf „modified“-Werte werden erkannt:
  1. fremde Bus-Transaktion unterbrechen
  2. eigenen (=modified) Wert zurückschreiben
  3. Status auf shared ändern
  4. unterbrochene Bus-Transaktion neu starten

MESI Protokoll (cont.)

- erfordert spezielle Snoop-Logik im Prozessor
- garantiert Cache-Kohärenz
- gute Performanz, aber schlechte Skalierbarkeit
- Zustandsübergänge: MESI Protokoll



PowerPC 604 RISC Microprocessor User's Manual [Motorola / IBM]

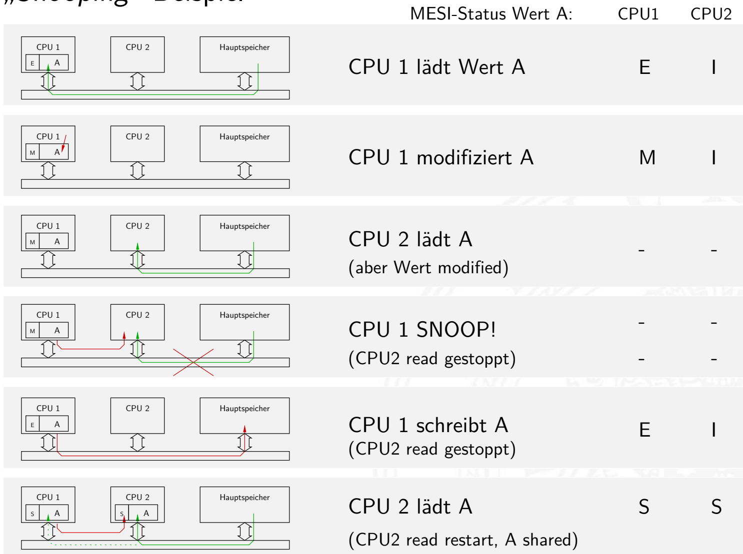
RH = Read hit
RMS = Read miss, shared
RME = Read miss, exclusive
WH = Write hit
WM = Write miss
SHR = Snoop hit on a read
SHW = Snoop hit on a write or read-with-intent-to-modify

Bus Transactions

(↓) = Snoop push
(⊗) = Invalidate transaction
(⊕) = Read-with-intent-to-modify
(↑) = Read

MESI Protokoll (cont.)

- „*Snooping*“ Beispiel



Optimierung der Cachezugriffe

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

⇒ 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
    ⇒ Zwischenspeicher für Schreiboperationen (Write Buffer)
  - Merging Write Buffer: aufeinanderfolgende Schreiboperationen zwischenspeichern und zusammenfassen
  - Victim Cache: kleiner voll-assoziativer Cache zwischen direct-mapped Cache und nächster Ebene
    „sammelt“ verdrängte Cache Einträge

⇒ Verbesserung der Cache Performanz durch kleinere R_{Miss}

  - größere Caches (− mehr Hardware)
  - höhere Assoziativität (− langsamer)

Optimierung der Cachezugriffe (cont.)

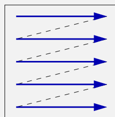
⇒ Optimierungstechniken

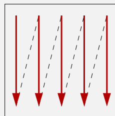
- Software Optimierungen
- Prefetch: Hardware (Stream Buffer)
  Software (Prefetch Operationen)
- Cache Zugriffe in Pipeline verarbeiten
- Trace Cache: im Instruktions-Cache werden keine Speicherinhalte, sondern ausgeführte Sequenzen (*trace*) einschließlich ausgeführter Sprünge gespeichert

  Beispiel: NetBurst Architektur (Pentium 4)

Cache Effekte bei Matrixzugriffen


```
public static double sumRowCol( double[][] matrix ) {
  int rows = matrix.length;
  int cols = matrix[0].length;
  double sum = 0.0;
  for( int r = 0; r < rows; r++ ) {
    for( int c = 0; c < cols; c++ ) {
      sum += matrix[r][c];
    }
  }
  return sum;
}
```

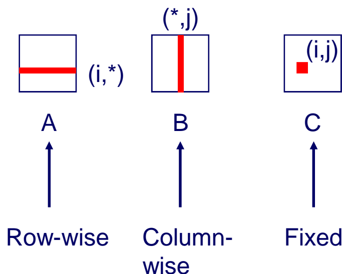




Matrix creation (5000×5000)	2105 msec.	
Matrix row-col summation	75 msec.	
Matrix col-row summation	383 msec.	⇒ 5 × langsamer
Sum = 600,8473695346258 / 600,8473695342268		⇒ andere Werte

Cache Effekte bei Matrixzugriffen (cont.)

```
/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```



Misses per Inner Loop Iteration:

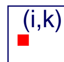
A	B	C
0.25	1.0	0.0

[BO15]

Cache Effekte bei Matrixzugriffen (cont.)

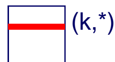

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Inner loop:



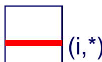
A

Fixed



B

Row-wise



C

Row-wise

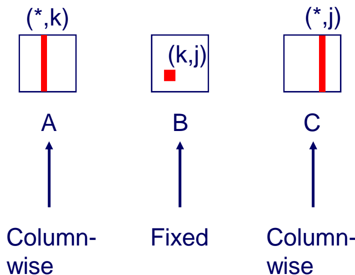
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

[BO15]

Cache Effekte bei Matrixzugriffen (cont.)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

[BO15]

Cache Effekte bei Matrixzugriffen (cont.)

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (i=0; i<n; i++)  {
   for (j=0; j<n; j++) {
      sum = 0.0;
      for (k=0; k<n; k++)
         sum += a[i][k] * b[k][j];
      c[i][j] = sum;
   }
}
```

kij (& ikj):

- 2 loads, 1 store
- misses/iter = **0.5**

```
for (k=0; k<n; k++) {
   for (i=0; i<n; i++) {
      r = a[i][k];
      for (j=0; j<n; j++)
         c[i][j] += r * b[k][j];
   }
}
```

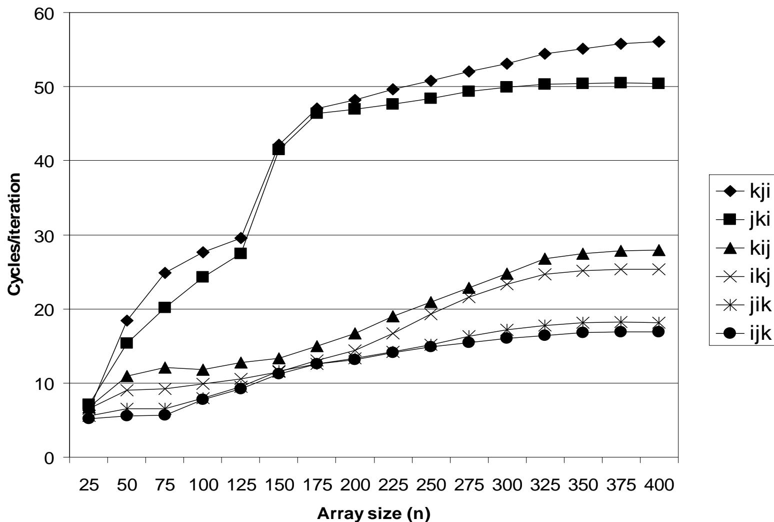
jki (& kji):

- 2 loads, 1 store
- misses/iter = **2.0**

```
for (j=0; j<n; j++) {
   for (k=0; k<n; k++) {
      r = b[k][j];
      for (i=0; i<n; i++)
         c[i][j] += a[i][k] * r;
   }
}
```

[BO15]

Cache Effekte bei Matrixzugriffen (cont.)



[BO15]

Cache vs. Programmcode

Programmierer kann für maximale Cacheleistung optimieren

- ▷ Datenstrukturen werden fortlaufend alloziert

1. durch entsprechende Organisation der Datenstrukturen
2. durch Steuerung des Zugriffs auf die Daten
   - Geschachtelte Schleifenstruktur
   - Blockbildung ist eine übliche Technik

Systeme bevorzugen einen *Cache-freundlichen* Code

- Erreichen der optimalen Leistung ist plattformspezifisch
  - Cachegrößen, Zeilengrößen, Assoziativität etc.
- generelle Empfehlungen
  - „working set" klein $\Rightarrow$ zeitliche Lokalität
  - kleine Adressfortschaltungen („strides") $\Rightarrow$ räumliche Lokalität

Virtueller Speicher – Motivation

- Wunsch des Programmierers
  - möglichst großer Adressraum, ideal 2^{32} Byte oder größer
  - linear adressierbar

- Sicht des Betriebssystems
  - verwaltet eine Menge laufender Tasks / Prozesse
  - jedem Prozess steht nur begrenzter Speicher zur Verfügung
  - strikte Trennung paralleler Prozesse
  - Sicherheitsmechanismen und Zugriffsrechte
  - read-only Bereiche für Code
  - read-write Bereiche für Daten

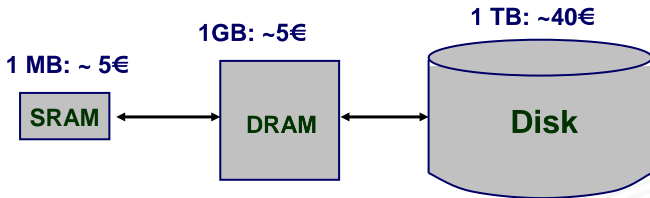
$\Rightarrow$ widersprüchliche Anforderungen

$\Rightarrow$ Lösung mit virtuellem Speicher und Paging

Virtueller Speicher – Motivation (cont.)

1. Benutzung der Festplatte als *zusätzlichen* Hauptspeicher
   - Prozessadressraum kann physikalische Speichergröße übersteigen
   - Summe der Adressräume mehrerer Prozesse kann physikalischen Speicher übersteigen
2. Vereinfachung der Speicherverwaltung
   - viele Prozesse liegen im Hauptspeicher
   - jeder Prozess mit seinem eigenen Adressraum $(0 \ldots n)$
   - nur *aktiver* Code und Daten sind tatsächlich im Speicher
     - bedarfsabhängige, dynamische Speicherzuteilung
3. Bereitstellung von Schutzmechanismen
   - ein Prozess kann einem anderen nicht beeinflussen
     - sie operieren in verschiedenen Adressräumen
   - Benutzerprozess hat keinen Zugriff auf privilegierte Informationen
   - jeder virtuelle Adressraum hat eigene Zugriffsrechte

Festplatte „erweitert“ Hauptspeicher

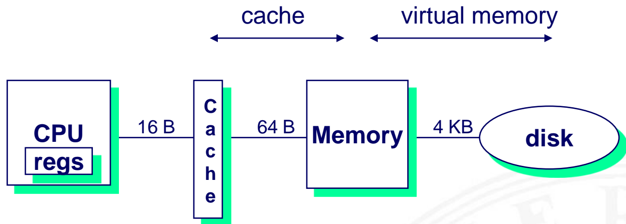


[BO15]

- Vollständiger Adressraum zu groß $\Rightarrow$ DRAM ist *Cache*
  - 32-bit Adressen: $\approx 4 \cdot 10^9$ Byte 4 Milliarden
  - 64-bit Adressen: $\approx 16 \cdot 10^{16}$ Byte 16 Quintillionen
- Speichern auf Festplatte ist $\approx 34 \times$ billiger als im DRAM
  - 1 TiB DRAM: $\approx 1\,000$ €
  - 1 TiB Festplatte: ≈ 30 €

$\Rightarrow$ kostengünstiger Zugriff auf große Datenmengen

Ebenen in der Speicherhierarchie



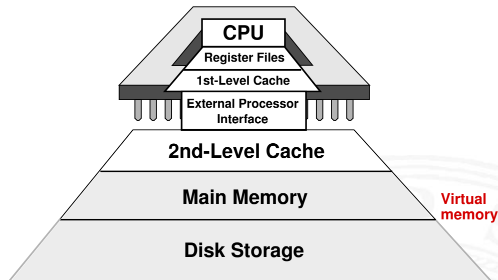
	Register	Cache	Memory	Disk Memory
size:	64 B	32 KB-12MB	8 GB	2 TB
speed:	300 ps	1 ns	8 ns	4 ms
$/Mbyte:		5€/MB	5€/GB	4 Ct./GB
line size:	16 B	64 B	4 KB	

larger, slower, cheaper

[BO15]

Ebenen in der Speicherhierarchie (cont.)

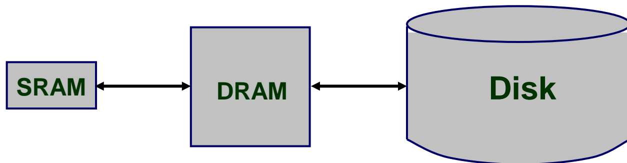
- Hauptspeicher als Cache für den Plattenspeicher



- Parameter der Speicherhierarchie

	1st-Level Cache	virtueller Speicher
Blockgröße	16-128 Byte	4-64 KiByte
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 %
Adressraum	14-20 bit	25-45 bit

Ebenen in der Speicherhierarchie (cont.)



[BO15]

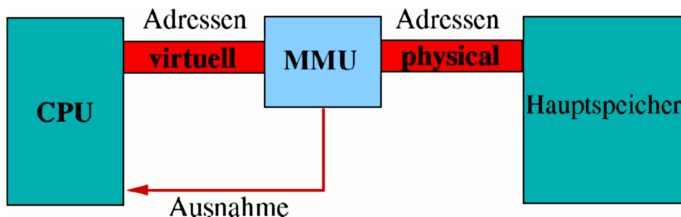
- DRAM vs. Festplatte ist extremer als SRAM vs. DRAM
  - Zugriffswartezeiten
    - DRAM $\approx 10\times$ langsamer als SRAM
    - Festplatte $\approx 500\,000\times$ langsamer als DRAM

$\Rightarrow$ Nutzung der räumlichen Lokalität wichtig

- erstes Byte $\approx 500\,000\times$ langsamer als nachfolgende Bytes

Prinzip des virtuellen Speichers

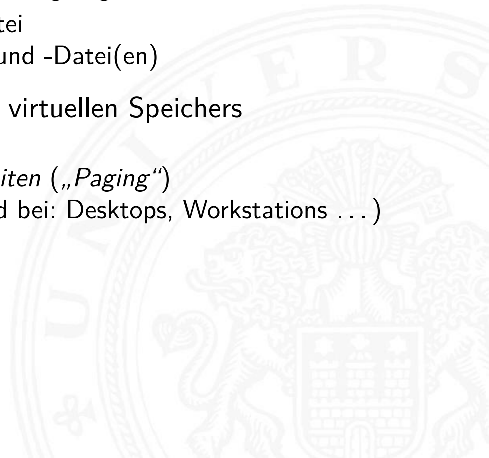
- jeder Prozess besitzt seinen eigenen virtuellen Adressraum
- Kombination aus Betriebssystem und Hardwareeinheiten
- MMU – **M**emory **M**anagement **U**nit



  - Umsetzung von virtuellen zu physikalischen Adressen, Programm-Relokation
  - Umsetzungstabellen werden vom Betriebssystem verwaltet
  - wegen des Speicherbedarfs der Tabellen beziehen sich diese auf größere Speicherblöcke (*Segmente* oder *Seiten*)

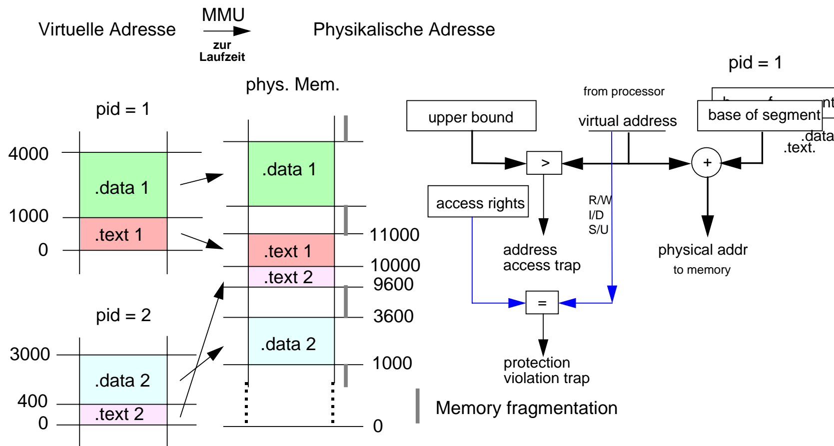
Prinzip des virtuellen Speichers (cont.)

- Umgesetzt wird nur die Anfangsadresse, der Offset innerhalb des Blocks bleibt unverändert
- Blöcke dieses virtuellen Adressraums können durch Betriebssystem auf Festplatte ausgelagert werden
  - Windows: Auslagerungsdatei
  - Unix/Linux: `swap` Partition und -Datei(en)
- Konzepte zur Implementation virtuellen Speichers
  - *Segmentierung*
  - Speicherzuordnung durch *Seiten* („*Paging*“)
  - gemischte Ansätze (Standard bei: Desktops, Workstations . . .)



Virtueller Speicher: Segmentierung

- Unterteilung des Adressraums in kontinuierliche Bereiche *variabler* Größe



Virtueller Speicher: Segmentierung (cont.)

- Idee: Trennung von Instruktionen, Daten und Stack

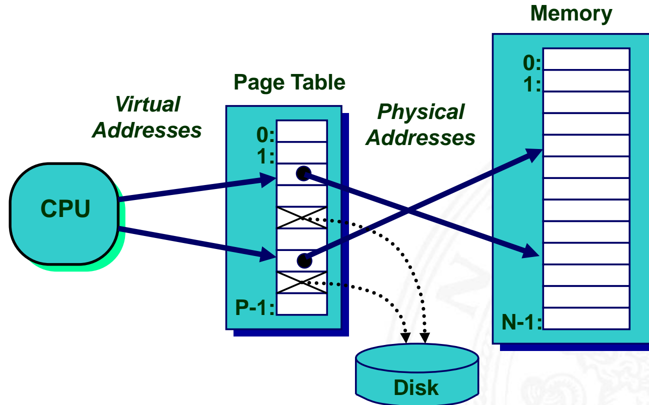
⇒ Abbildung von *Programmen* in den *Hauptspeicher*

+ 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



[BO15]

Virtueller Speicher: Paging / Seitenadressierung (cont.)

⇒ Abbildung von *Adressen* in den *virtuellen Speicher*: Hauptspeicher + Festplatte

\+ Programme können größer als der Hauptspeicher sein

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

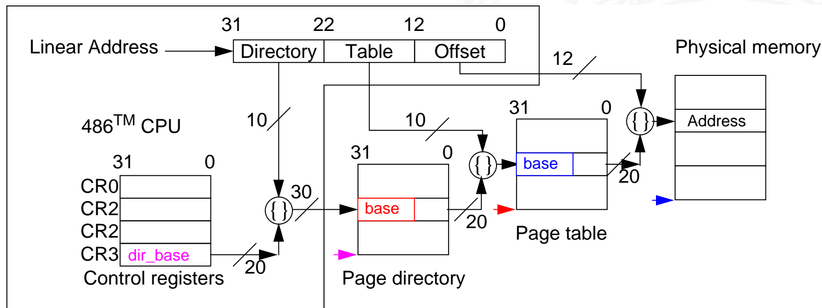
\+ feste Seitengröße: einfache Verwaltung in Hardware

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

\+ gemeinsam genutzte Programmteile/-Bibliotheken können sehr einfach in das Konzept integriert werden
- Windows: `.dll`-Dateien
- Unix/Linux: `.so`-Dateien

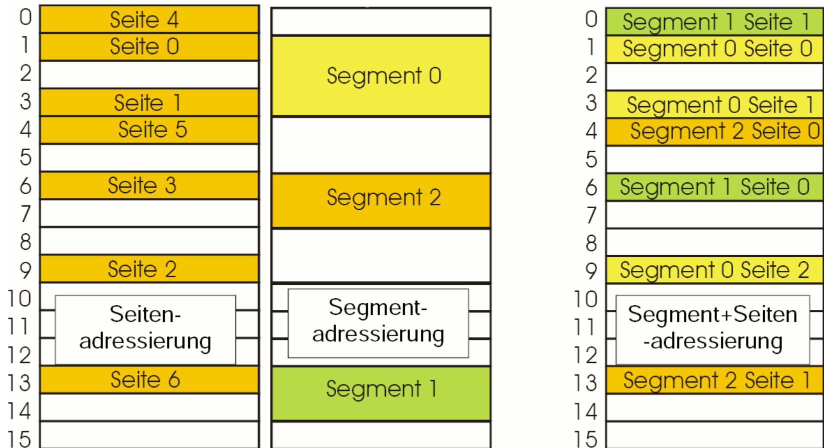
Virtueller Speicher: Paging / Seitenadressierung (cont.)

- große Miss-Penalty (Nachladen von der Platte)
  - ⇒ Seiten sollten relativ groß sein: 4 . . . 64 KiByte
- – Speicherplatzbedarf der Seitentabelle
  viel virtueller Speicher, 4 KiByte Seitengröße
  - = sehr große Pagetable
  - ⇒ Hash-Verfahren (*inverted page tables*)
  - ⇒ mehrstufige Verfahren



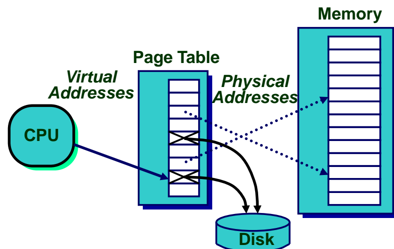
Virtueller Speicher: Segmentierung + Paging

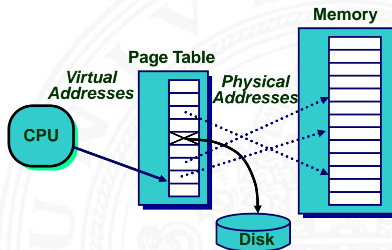
aktuell = Mischung: Segmentierung und Paging (seit I386)



Seitenfehler

- Seiten-Tabelleneintrag: Startadresse der virt. Seite auf Platte
- Daten von Festplatte in Speicher laden:
  Aufruf des „Exception handler“ des Betriebssystems
  - laufender Prozess wird unterbrochen, andere können weiterlaufen
  - Betriebssystem kontrolliert die Platzierung der neuen Seite im Hauptspeicher (Ersetzungsstrategien) etc.



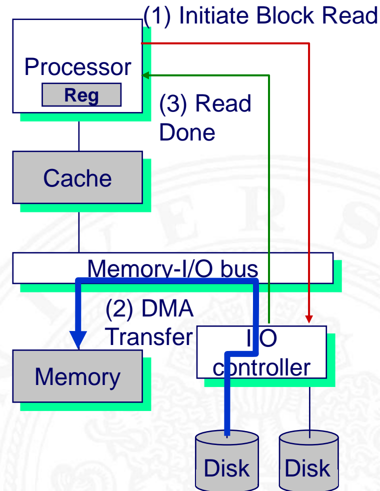


[BO15]

Seitenfehler (cont.)

Behandlung des Seitenfehlers

1. Prozessor signalisiert DMA-Controller
   - lies Block der Länge P ab Festplattenadresse X
   - speichere Daten ab Adresse Y in Hauptspeicher
2. Lesezugriff erfolgt als
   - Direct Memory Access (DMA)
   - Kontrolle durch I/O Controller
3. I/O Controller meldet Abschluss
   - Gibt Interrupt an den Prozessor
   - Betriebssystem lässt unterbrochenen Prozess weiterlaufen



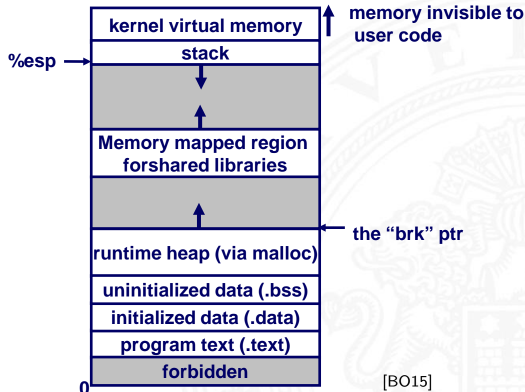
[BO15]

Separate virtuelle Adressräume

Mehrere Prozesse können im physikalischen Speicher liegen

- Wie werden Adresskonflikte gelöst?
- Was passiert, wenn Prozesse auf dieselbe Adresse zugreifen?

Linux x86
Speicherorganisation

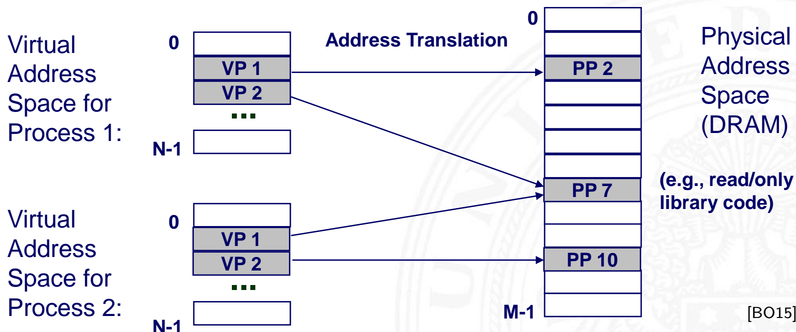


[BO15]

Separate virtuelle Adressräume (cont.)

Auflösung der Adresskonflikte

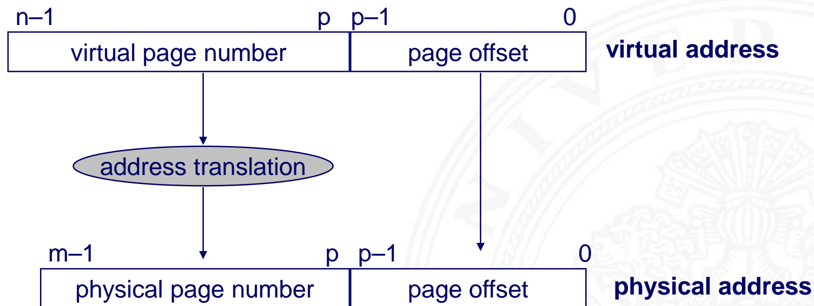
- jeder Prozess hat seinen eigenen virtuellen Adressraum
- Betriebssystem kontrolliert wie virtuelle Seiten auf den physikalischen Speicher abgebildet werden



[BO15]

Virtueller Speicher – Adressumsetzung

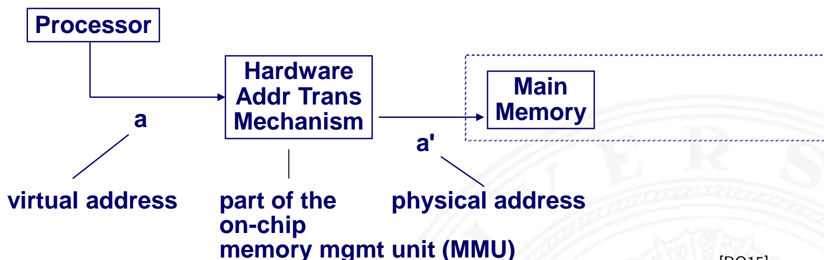
- Parameter
  - $P = 2^p$ = Seitengröße (Bytes)
  - $N = 2^n$ = Limit der virtuellen Adresse
  - $M = 2^m$ = Limit der physikalischen Adresse



[BO15]

Virtueller Speicher – Adressumsetzung (cont.)

- virtuelle Adresse: Hit

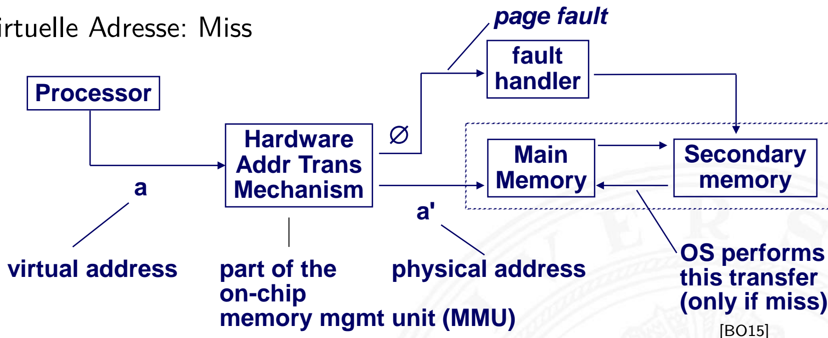


[BO15]

- Programm greift auf virtuelle Adresse a zu
- MMU überprüft den Zugriff, liefert physikalische Adresse a'
- Speicher liefert die zugehörigen Daten $d[a']$

Virtueller Speicher – Adressumsetzung (cont.)

- virtuelle Adresse: Miss



[BO15]

- Programm greift auf virtuelle Adresse *a* zu
- MMU überprüft den Zugriff, Adresse nicht in Hauptspeicher
- „page-fault" ausgelöst, Betriebssystem übernimmt

Seiten-Tabelle

Virtual Page Number

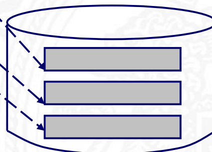
Memory resident page table (physical page or disk address)

Valid

Valid	
1	
1	
0	
1	
1	
1	
0	
1	
0	
1	

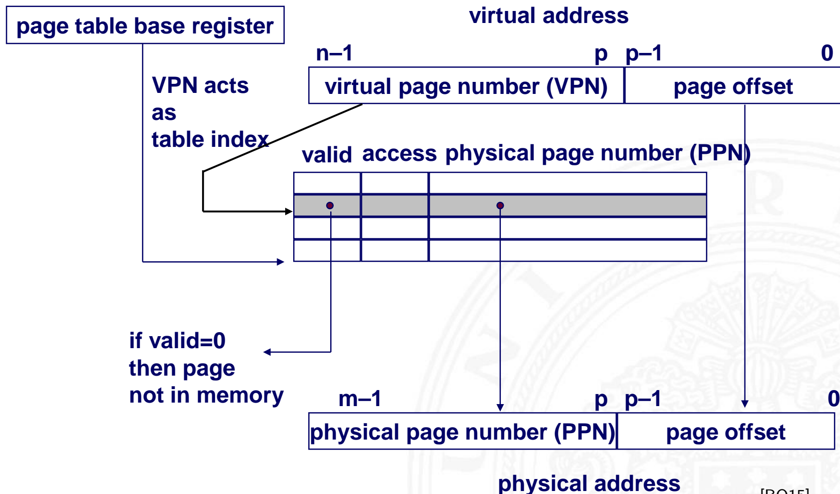
Physical Memory

Disk Storage (swap file or regular file system file)



[BO15]

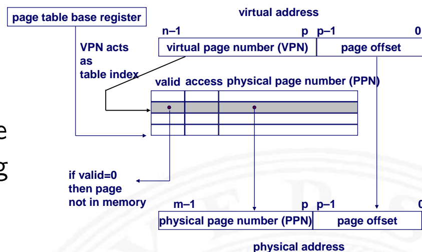
Seiten-Tabelle (cont.)



[BO15]

Seiten-Tabelle (cont.)

- separate Seiten-Tabelle für jeden Prozess
- VPN („*Virtual Page Number*“) bildet den Index der Seiten-Tabelle
  ⇒ zeigt auf Seiten-Tabelleneintrag
- Seiten-Tabelleneintrag liefert Informationen über die Seite
- Daten im Hauptspeicher: valid-Bit
  - valid-Bit = 1: die Seite ist im Speicher ⇒ benutze physikalische Seitennummer („Physical Page Number“) zur Adressberechnung
  - valid-Bit = 0: die Seite ist auf der Festplatte ⇒ Seitenfehler

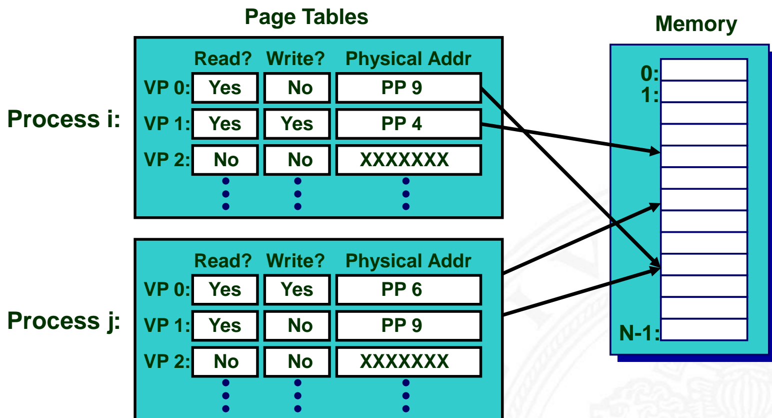


Zugriffsrechte

Schutzüberprüfung

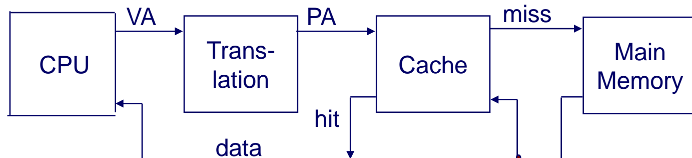
- Zugriffsrechtefeld gibt Zugriffserlaubnis an
  - typischerweise werden zahlreiche Schutzmodi unterstützt
  - Unterscheidung zwischen Kernel- und User-Mode
  - z.B. read-only, read-write, execute-only, no-execute
  - no-execution Bits gesetzt für Stack-Pages: Erschwerung von Buffer-Overflow-Exploits
- Schutzrechteverletzung wenn Prozess/Benutzer nicht die nötigen Rechte hat
- bei Verstoß erzwingt die Hardware den Schutz durch das Betriebssystem („Trap“ / „Exception“)

Zugriffsrechte (cont.)



[BO15]

Integration von virtuellem Speicher und Cache



[BO15]

Die meisten Caches werden *physikalisch adressiert*

- Zugriff über physikalische Adressen
- mehrere Prozesse können, gleichzeitig Blöcke im Cache haben
- –"– sich Seiten teilen
- Cache muss sich nicht mit Schutzproblemen befassen
  - Zugriffsrechte werden als Teil der Adressumsetzung überprüft

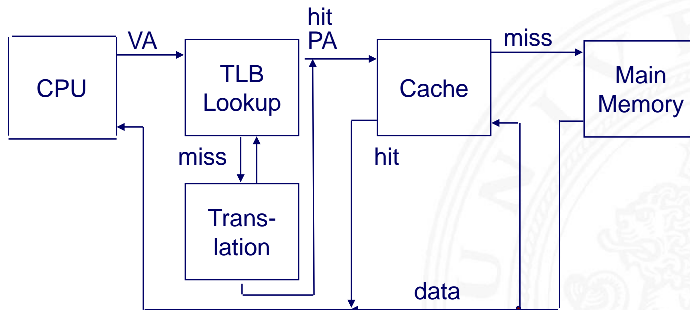
Die Adressumsetzung wird vor dem Cache „Lookup" durchgeführt

- kann selbst Speicherzugriff (auf den PTE) beinhalten
- Seiten-Tabelleneinträge können auch gecacht werden

TLB / „Translation Lookaside Buffer“

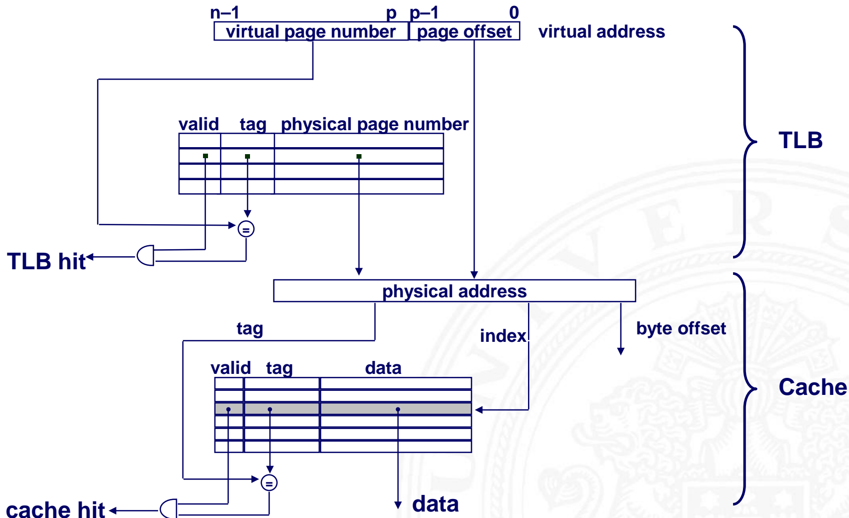
Beschleunigung der Adressumsetzung für virtuellen Speicher

- kleiner Hardware Cache in MMU (Memory Management Unit)
- bildet virtuelle Seitenzahlen auf physikalische ab
- enthält komplette Seiten-Tabelleneinträge für wenige Seiten



[BO15]

TLB / „Translation Lookaside Buffer“ (cont.)

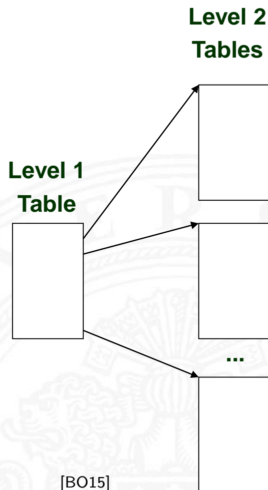


[BO15]

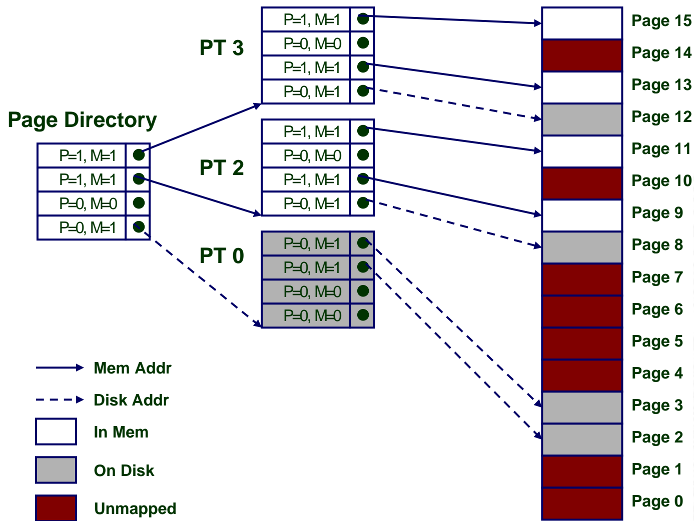
mehrstufige Seiten-Tabellen

- Gegeben
  - 4 KiB (2^{12}) Seitengröße
  - 32-bit Adressraum
  - 4-Byte PTE („Page Table Entry“)
    Seitentabelleneintrag
- Problem
  - erfordert 4 MiB Seiten-Tabelle
  - 2^{20} Bytes

⇒ übliche Lösung
  - mehrstufige Seiten-Tabellen („multi-level“)
  - z.B. zweistufige Tabelle (Pentium P6)
    - Ebene-1: 1024 Einträge → Ebene-2 Tabelle
    - Ebene-2: 1024 Einträge → Seiten



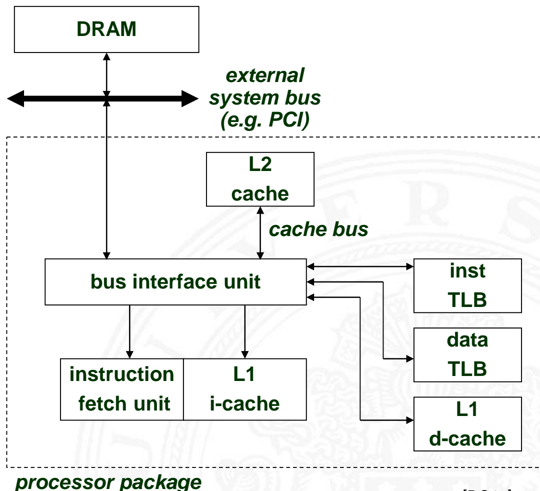
mehrstufige Seiten-Tabellen (cont.)



[BO15]

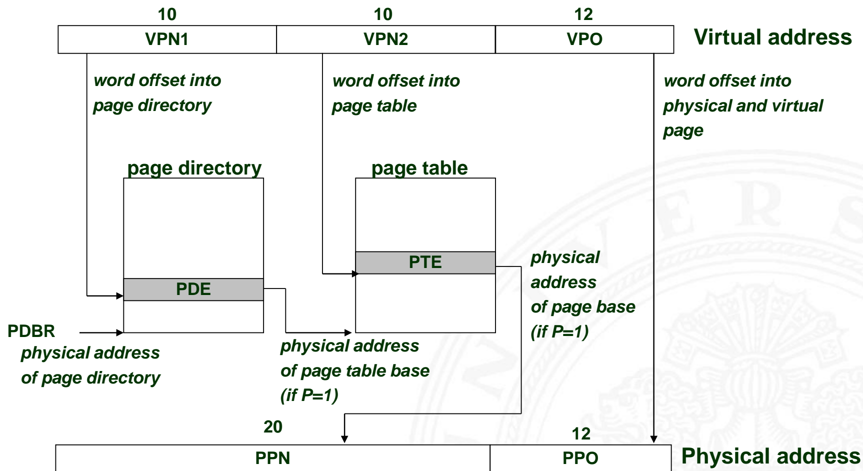
Beispiel: Pentium und Linux

- 32-bit Adressraum
- 4 KiB Seitengröße
- L1, L2 TLBs
  4fach assoziativ
- Instruktions TLB
  32 Einträge
  8 Sets
- Daten TLB
  64 Einträge
  16 Sets
- L1 I-Cache, D-Cache
  16 KiB
  32 B Cacheline
  128 Sets
- L2 Cache
  Instr.+Daten zusammen
  128 KiB . . . 2 MiB



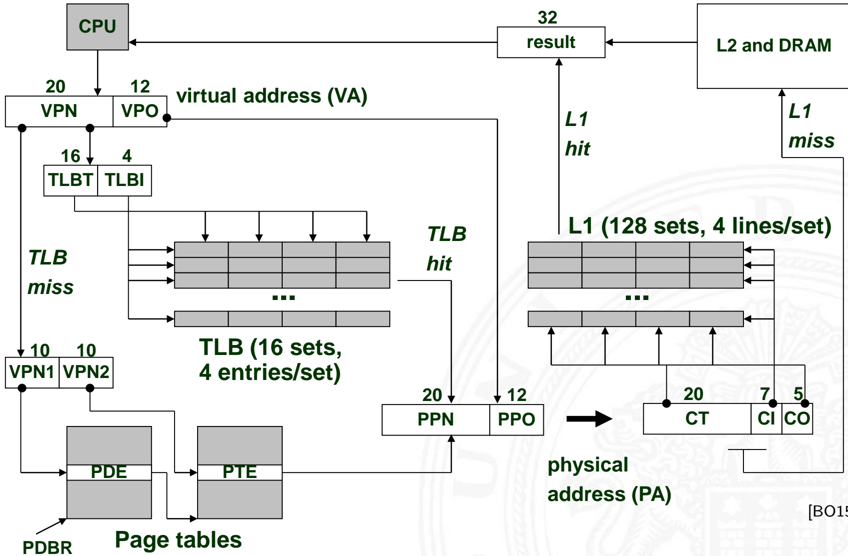
[BO15]

Beispiel: Pentium und Linux (cont.)



[BO15]

Beispiel: Pentium und Linux (cont.)



[BO15]

Zusammenfassung

Cache Speicher

- dient nur zur Beschleunigung
- unsichtbar für Anwendungsprogrammierer und OS
- komplett in Hardware implementiert

Virtueller Speicher

- ermöglicht viele Funktionen des Betriebssystems
  - größerer virtueller Speicher als reales DRAM
  - Auslagerung von Daten auf die Festplatte
  - Prozesse erzeugen („exec“ / „fork“)
  - Taskwechsel
  - Schutzmechanismen
- Implementierung mit Hardware und Software
  - Software verwaltet die Tabellen und Zuteilungen
  - Hardwarezugriff auf die Tabellen
  - Hardware-Caching der Einträge (TLB)

Zusammenfassung (cont.)

Sicht des Programmierers

- großer „flacher“ Adressraum
- Programm „besitzt“ die gesamte Maschine
  - hat privaten Adressraum
  - bleibt unberührt vom Verhalten anderer Prozesse

Sicht des Systems

- Adressraum von Prozessen auf Seiten abgebildet
  - muss nicht fortlaufend sein
  - wird dynamisch zugeteilt
  - erzwingt Schutz bei Adressumsetzung
- Betriebssystem verwaltet viele Prozesse gleichzeitig
  - jederzeit schneller Wechsel zwischen Prozessen
  - u.a. beim Warten auf Ressourcen (Seitenfehler)

Literaturliste

[PH20] David A. Patterson, John L. Hennessy:
Computer Organization and Design – The Hardware Software Interface – MIPS Edition.
Morgan Kaufmann Publishers Inc., 2020. ISBN 978–0–12–820109–1

[PH16] David A. Patterson, John L. Hennessy:
Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle.
5. Aufl.; Oldenbourg, 2016. ISBN 978–3–11–044605–0

[HP17] John L. Hennessy, David A. Patterson:
Computer architecture – A quantitative approach.
6th ed.; Morgan Kaufmann Publishers Inc., 2017. ISBN 978–0–12–811905–1

Literaturliste (cont.)

[Sta19] W. Stallings:
Computer Organization and Architecture – Designing for Performance.
11th edition, Pearson International, 2019. ISBN 978–0–13–499719–3

[TA14] Andrew S. Tanenbaum, Todd Austin:
Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner.
6. Aufl.; Pearson Deutschland GmbH, 2014. ISBN 978–3–86894–238–5

[BO15] Randal E. Bryant, David R. O'Hallaron:
Computer systems – A programmers perspective.
3rd global ed., Pearson Education Ltd., 2015. ISBN 978–1–292–10176–7. `csapp.cs.cmu.edu`

Literaturliste (cont.)

[Fur00] Steve Furber:
ARM System-on-Chip Architecture.
2nd edition, Pearson Education Limited, 2000. ISBN
978–0–201–67519–1

[Intel] Intel Corp.; Santa Clara, CA.
`www.intel.com` `ark.intel.com`