



Unterlagen zur Vorlesung

Rechnerarchitektur

Prof. Dr. Jürgen Neuschwander

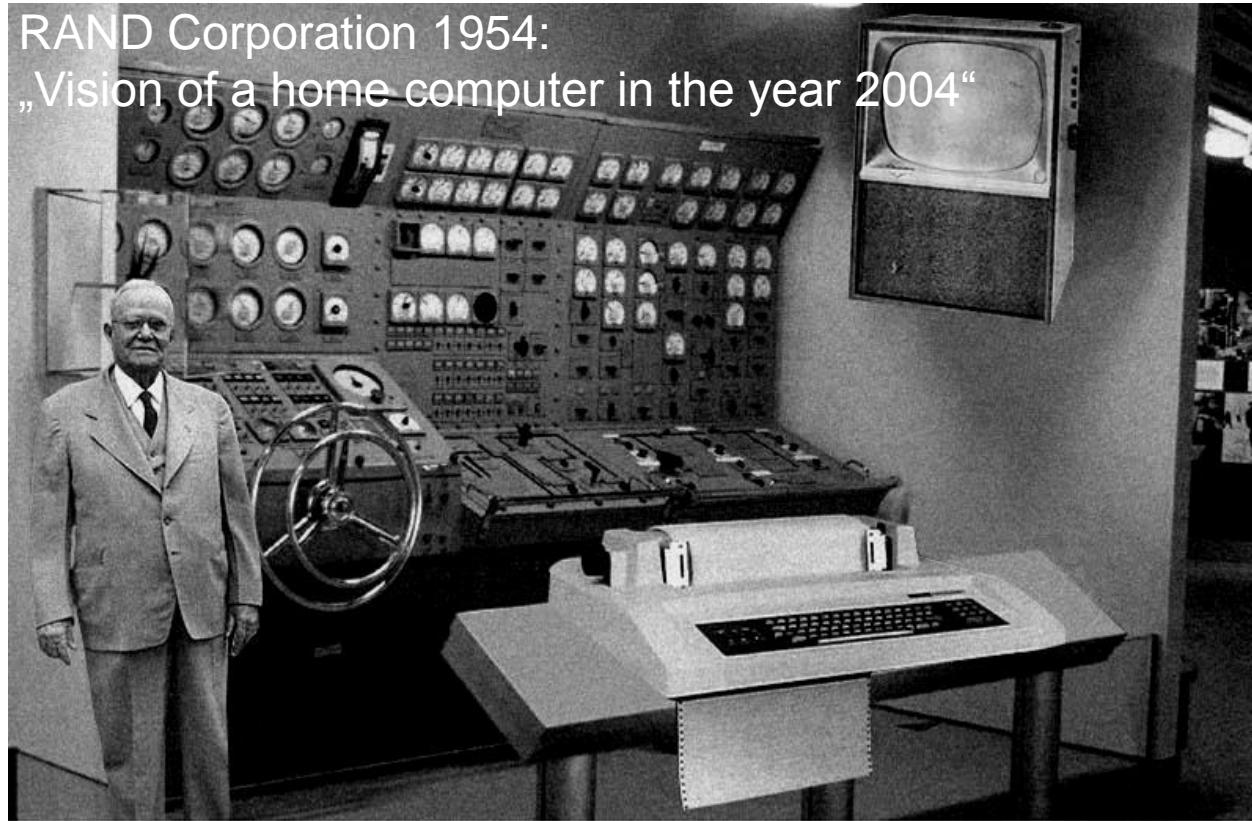
Allgemeine Informationen

- Prof. Dr. Jürgen Neuschwander
- Fakultät Informatik
- Gebäude O, Raum 304
- Telefon: 07531 206 501
- E-Mail: juergen.neuschwander@htwg-konstanz.de
- Kursunterlagen:
 - http://www-home.htwg-konstanz.de/~neuschwa/LV_Rechnerarchitektur.htm
 - oder auf Moodle
- Sprechstunden nach Vereinbarung

Dieser Kurs bietet eine Einführung in jene seltsamen Maschinen,
welche sich weigern, uns das Denken abzunehmen!

RAND Corporation 1954:

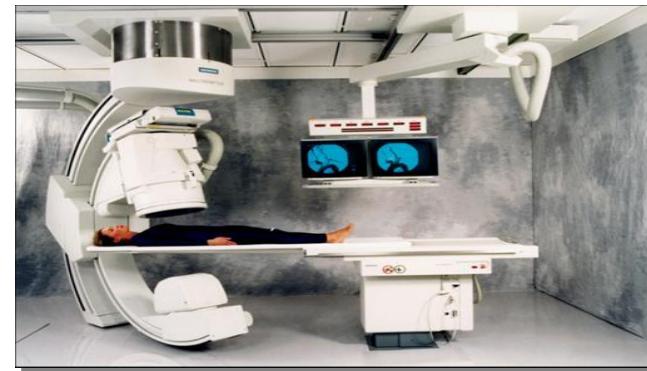
„Vision of a home computer in the year 2004“



Entwicklungstrends in der Computertechnik

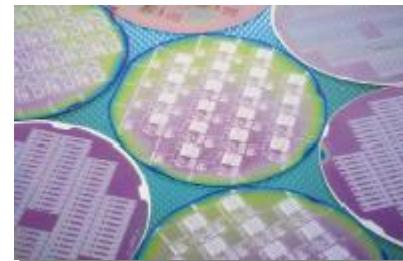
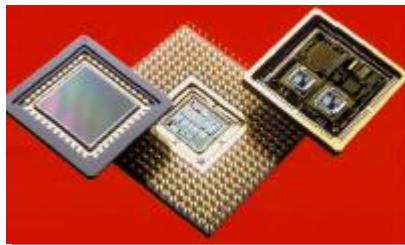
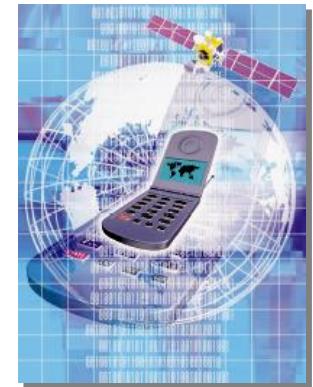
Entwicklungstrends in der Computertechnik

- Computerbasierte Systeme sind in alle Bereiche des Alltags eingedrungen, z. B.:
 - Verkehrsmittel
 - Kommunikation
 - medizinische Technik
 - „Intelligente Kleidung“ (NEC)



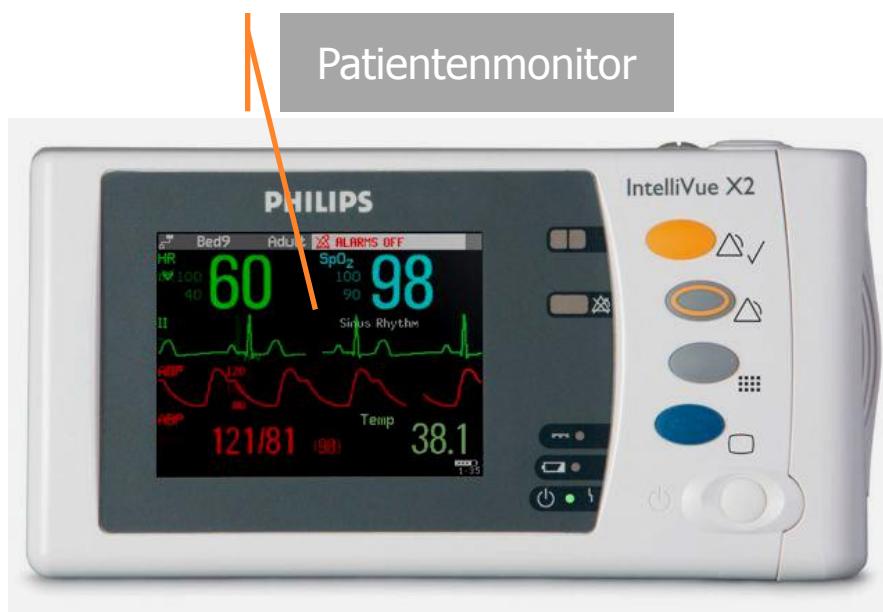
Moderne Entwicklung in Schlagworten

- Die Rechner sind „unsichtbar“ und nehmen an Zahl gewaltig zu
 - **Ubiquitäre Systeme**
- Alles wird mit allem verbunden
 - Internet, Next Generation Mobile Networks (LTE), Wireless LANs
 - Allways-on
- „Chips“ sind wesentliche Gestalter des wirtschaftlichen Geschehens (Mikroelektronik)



Embedded Systems

- **Eingebettete Systeme** sind Computersysteme, die auf einen Anwendungsbereich spezialisiert und in einen technischen Kontext eingebunden sind.



Patientenmonitor

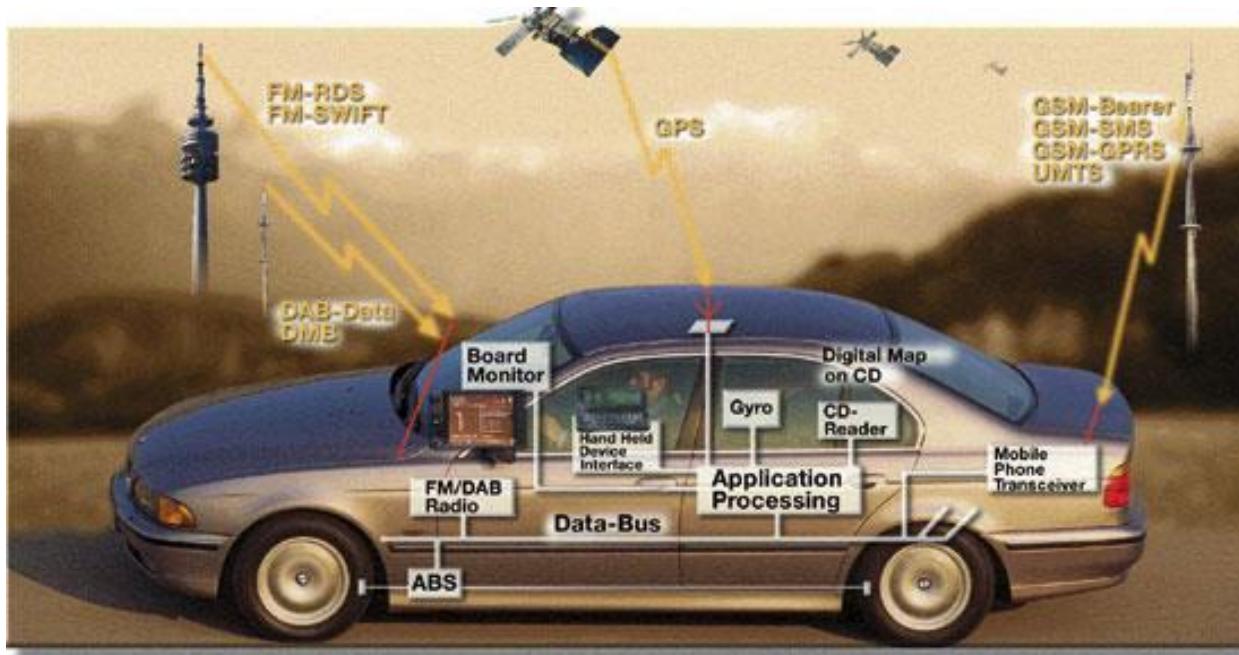


Benutzer-schnittstelle

Temperatur-steuerung

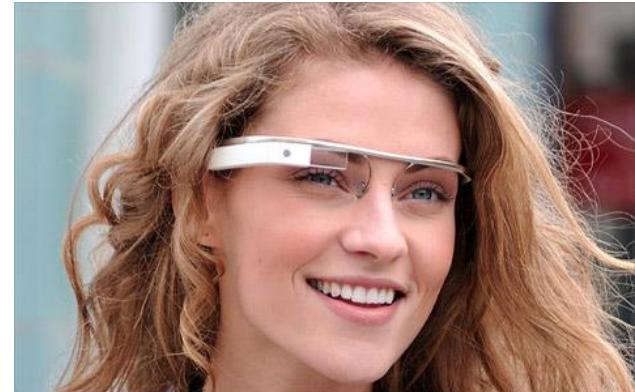
Embedded Systems

- Echtzeitsysteme sind eingebettete Systeme, die in zeitkritischen Anwendungen eingesetzt werden, bei denen eine Reaktion innerhalb bestimmter Zeitschränken erforderlich ist.
- Eingebettete Systeme sind verteilt implementiert. Die einzelnen Komponenten sind über ein Kommunikationssystem verbunden.



Entwicklungstrends - Ubiquitäre IT-Systeme

- Ubiquität („Allgegenwärtigkeit“)
 - Nichtgebundensein an einen Standort
 - Information als überall erhältliches Gut
 - Information Technology (IT) beyond the PC
- Persönliche Technologien
 - Zugang zu IT-Diensten mit sich herumtragen
 - Beispiele: Smartphone, wearable Devices
- Informationsumgebungen
 - Zugang zu IT-Diensten überall vorhanden
 - Beispiele: Intelligente, kommunikationsfähige Geräte/ Systeme, wie intelligente Kaffeetasse, aktive Gebäude
- Ubiquitäre Unterstützung wirkt im Hintergrund
 - Wird selbst aktiv, (teil-)autonom von Menschen



Beispiel: Intelligente Türschilder

- Intelligente Türschilder als Wegweiser
 - Pfeile in Richtung des gesuchten Büros
- Intelligente Türschilder als Rezeption
 - Arbeitssituation bei Anwesenheit
- Intelligente Türschilder als Stellvertreter
 - Auskunft über aktuellen Aufenthaltsort
 - Auskunft über voraussichtlich Rückkehr bei Abwesenheit
- Intelligente Türschilder als Informationsquelle
 - Wer war da?



Beispiel: Die MediaCup

- Der Boden der MediaCup enthält die Elektronik in einem abnehmbaren Gummiüberzieher.
- Die Elektronik wird kabellos mit Energie versorgt; ein 15 minütiger Aufladevorgang kann die Tasse etwa 10 Stunden mit Energie versorgen.
- Sensoren erkennen Temperatur und Bewegungszustand der Tasse.
- Diese Informationen wird von der Tasse in den Raum gesendet.



Beispiel: Die MediaCup

HotWatch

- Eine spezielle Uhr (HotWatch) gibt einen Pieps-Warnton von sich, wenn der Kaffee/Tee in der MediaCup zu heiß zum Trinken ist.



Beispiel: Die MediaCup

CoffeePump

- Eine erweiterte Kaffeemaschine (CoffeePump) brüht neuen Kaffee, falls der Kaffee leer ist und alle Tassen ausgetrunken sind.



Beispiel: Die MediaCup

- Mit Hilfe der MediaCup kann ein elektronische Türschild (SmartDoorplate) bestimmte Situationen erkennen und darauf reagieren.
- So kann das Türschild "Besprechung" angeben, wenn durch die Tassenkonstellation eine Besprechung im Zimmer erkannt wird.



Beispiel: Photoplethysmographie

- Spiegel mit integrierter Kamera, der auf einem in das Glas integrierten Display die Pulsrate der Person anzeigt, die vor dem Spiegel steht.



- Das System misst Helligkeits-Unterschiede des von der Haut reflektierten Lichtes und wertet diese aus. Während eines Herzschlags dehnen sich die Blutgefäße leicht, da der Druck steigt. Das verursacht einen Anstieg der optischen Absorption, und deshalb auch ein Absinken der Intensität des Lichtes, das vom Gesicht reflektiert wird. Die Genauigkeit liegt im Bereich von drei Schlägen pro Minute
- Demnächst lassen sich auch weitere Parameter wie Atemfrequenz oder Sauerstoff-Werte im Blut so ablesen (das auch bald mit Fotofunktion des Handys).

Beispiel: Fitness-Armbänder



Kommunikation zwischen Armband und Smartphone per Bluetooth.
Gespeichert werden die persönlichen Daten in der Cloud.

Vorlesungsinhalt

- Einleitung
 - Entwicklung der Rechnertechnik
 - Entwurfsprinzipien und Leistungsbewertung
 - Begriff Rechnerarchitektur
 - Amdahls Gesetz
- Die von Neumann-Maschine
 - Operationsprinzip, Informationskomponenten
 - Strukturelemente einer CPU und Prozesse
 - Komponenten eines Operationswerks (Beispiel M68000)
 - Datenzugriffe
 - Stacks und Queues (Unterprogrammtechnik)
 - Klassen von Architekturen
 - Programmiermodell

Vorlesungsinhalt

- Assemblerprogrammierung (Beispiel M68000)
 - Grundlagen
 - Befehlssatz / Adressierungsarten / Pseudobefehle
 - Technik der Assemblierung
 - Technik Unterprogramme und Parameterübergabe
 - Praktische Beispiele im Praktikum
- Steuerwerk eines Prozessors
 - Direkte und indirekte Realisierung
 - Mikroprogrammierte Steuerwerke
 - Bus-Arbitration / Priorisierung
- Bussysteme
 - Synchroner, semi-synchroner und asynchroner Bus / Bushierarchie
- Synchronisation von Prozessen
 - Binäre Semaphore

Vorlesungsinhalt

- Speicher, Speicherbausteine und Speicherzugriff
 - Struktur und Organisation von Speicherbausteinen (SRAM/DRAM)
 - Interleaving, Direct Memory Access
- CISC, RISC, Superskalarität
- Pipelineverarbeitung
 - Klassifikation nach Flynn
 - DLX-Pipeline (Struktur und zeitlicher Ablauf)
 - Leistungssteigerung durch Pipelines
 - Pipeline-Hemmisse, Datenabhängigkeiten und Konflikte
 - Steuerflusskonflikte
- Parallelle Pipelines und Sprungvorhersage
 - Superskalare Prozessoren /VLIW-Prozessoren
 - Superskalare Pipelines
 - Multithreading
 - Statische und dynamische Sprungvorhersage

Vorlesungsinhalt

- Cache Speicher
 - Prinzip und Funktionsweise, Systemstrukturen
 - Innerer Aufbau und Arbeitsweise
 - Trefferrate und Zugriffszeiten
 - Aktualisierungsstrategien
 - Varianten von Cache-Speichern
 - Full associative Caches
 - Direct Mapped Caches
 - N-way set associative Caches
 - Cache Kohärenz
- Virtuelle Speichererwaltung
 - Segmentbasierte Verwaltung
 - Seitenbasierte Verwaltung / Paging /mehrstufige Umsetzung
 - Probleme der virtuellen Speicherverwaltung und Lösungen

Vorlesungsinhalt

- Ausnahmeverarbeitung
 - Exceptions
 - Vektor- und Autovektor-Interrupts
 - Priorisierung

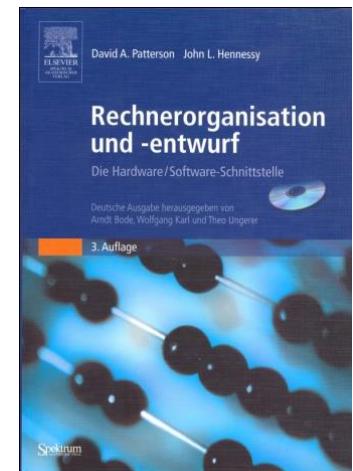
Literaturauswahl

- Th. Flik: *Mikroprozessortechnik*
Springer Verlag, 2008, (*Standardwerk*)
- A.S.Tanenbaum: *Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner*
(Pearson Studium - IT), 2014
- W.Oberschelp, G.Vossen: *Recheraufbau und Rechnerstrukturen*
Oldenbourg Verlag, 2006
- H. Bähring: *Mikrorechner-Technik, Band 1 und 2*
Springer Verlag, 2004
- Herold, Lurz, Wohlrab: *Grundlagen der Informatik*,
Pearson Studium, 2012



Literaturauswahl

- Brinkschulte, Ungerer: *Mikrocontroller und Mikroprozessoren*
Springer Verlag, 1. Auflage, 2002
- Patterson, Hennessy: *Rechnerorganisation und –entwurf*,
Verlag Spektrum, 2011 (*Standardwerk*)
- H. Malz: *Rechnerorganisation*
Vieweg Verlag, 2001



Die Leistungssteigerung bei Mikroprozessoren ist durch folgende Fortschritte erreicht worden:

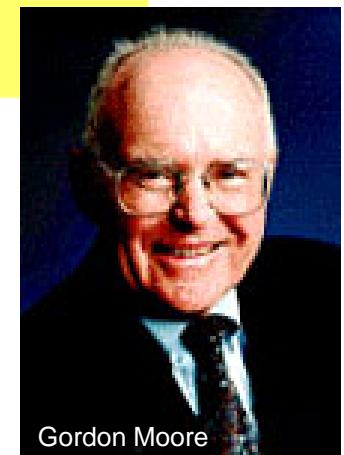
- durch Steigerung der Gatterzahl auf dem Chip,
- durch Steigerung der Taktrate und
- durch Fortschritte beim Hardware-Entwurf (Architektur, Mikroarchitektur, Entwurfswerkzeuge).

Das “Mooresche Gesetz” in der Mikroelektronik

Die Anzahl der Transistoren pro (Prozessor-)Chip verdoppelt sich alle zwei Jahre.

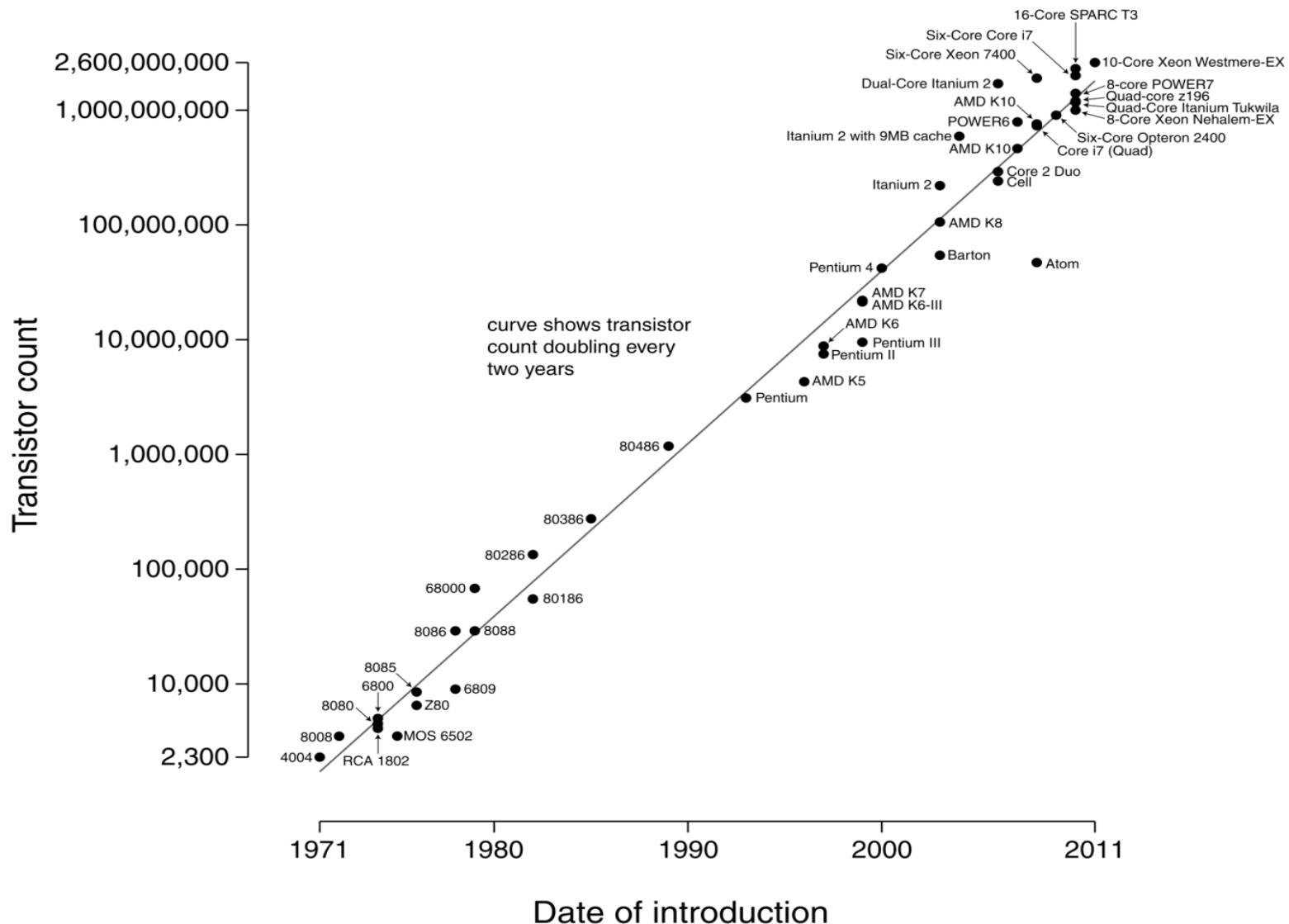
Die Verarbeitungsleistung der Hochleistungsprozessoren Verdoppelt sich alle 18 Monate.

Für den gleichen Preis liefert die Mikroelektronik die doppelte Leistung in weniger als zwei Jahren.

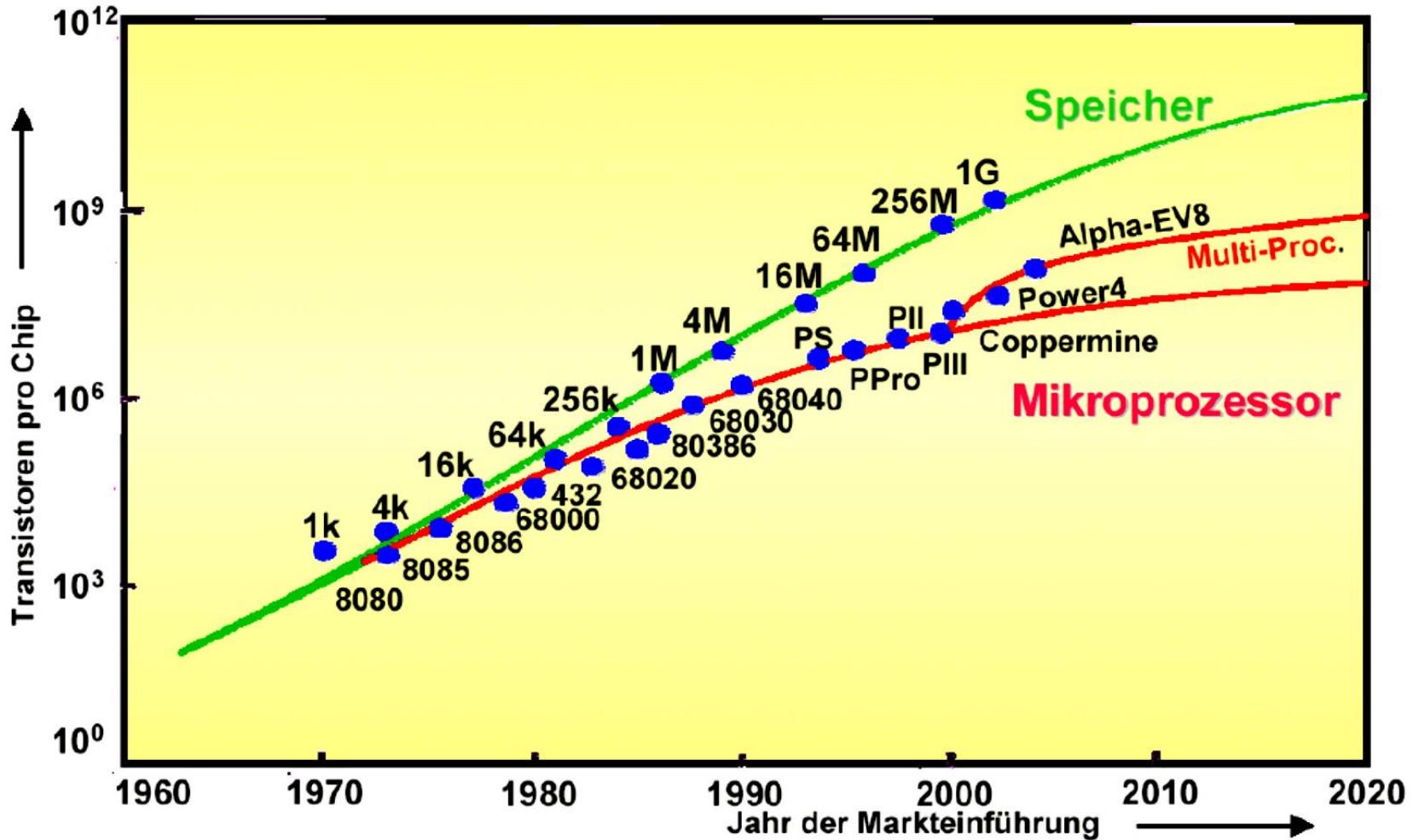


Gordon Moore

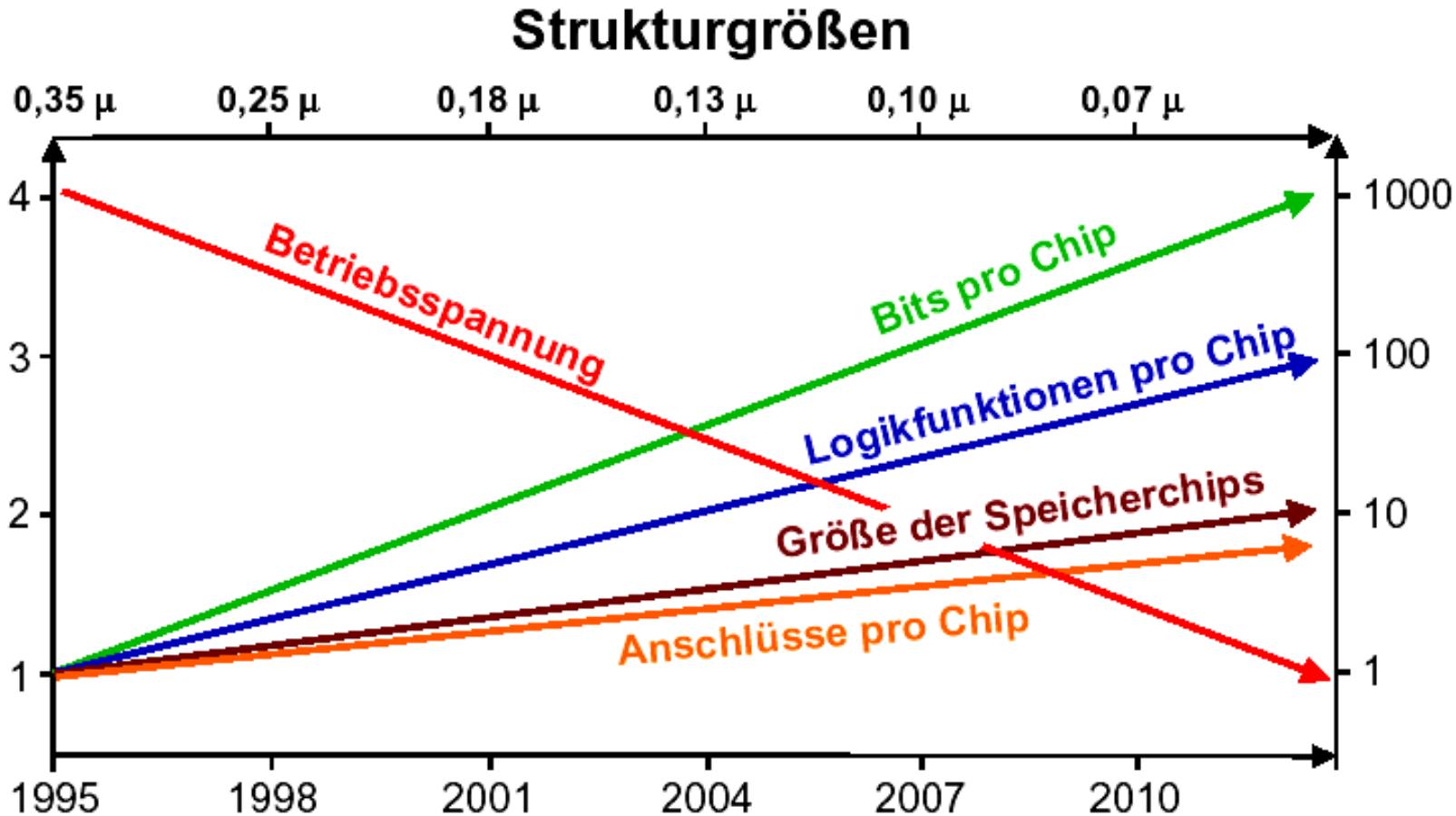
Microprocessor Transistor Counts 1971-2011 & Moore's Law



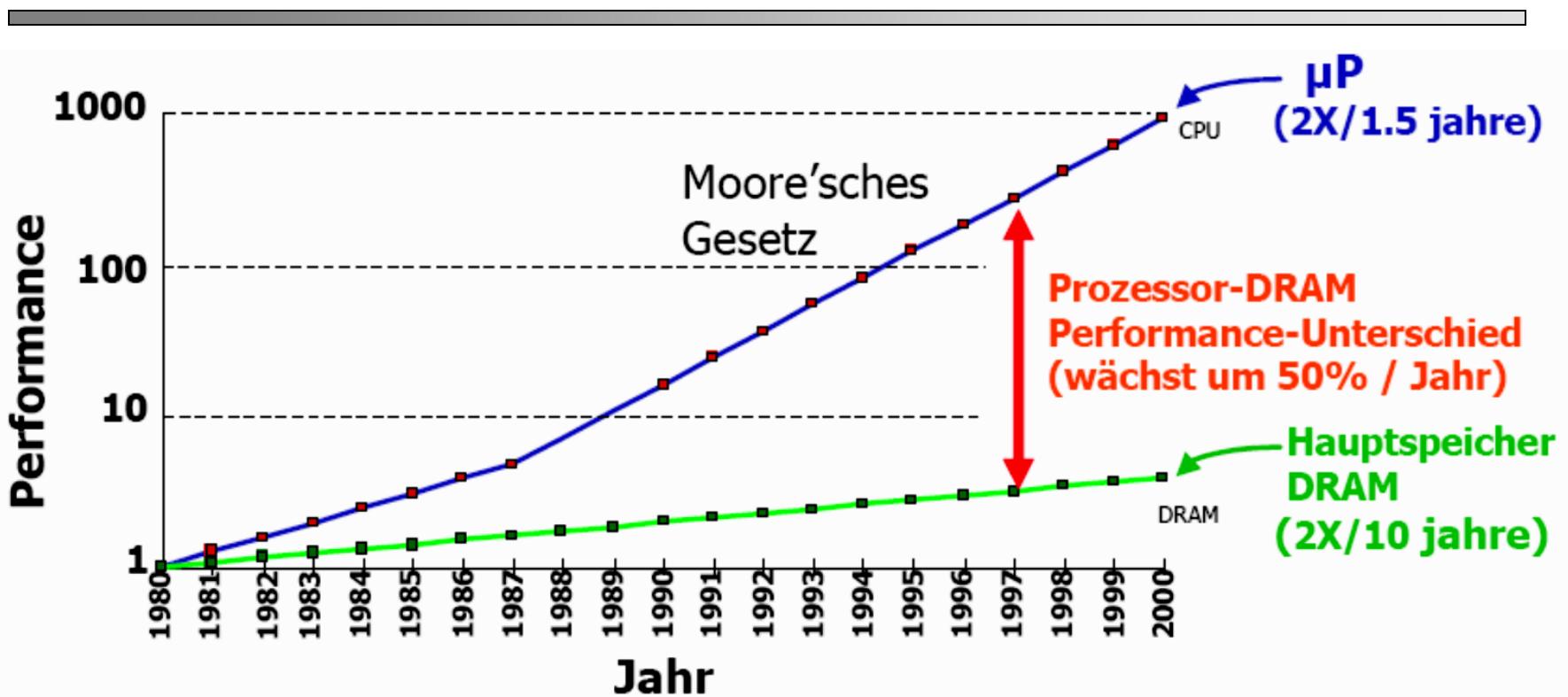
Mooresches Gesetz bei Halbleiterelementen



Leistungsparameter bei Prozessoren



Prozessor-Speicher-Performance-Unterschied



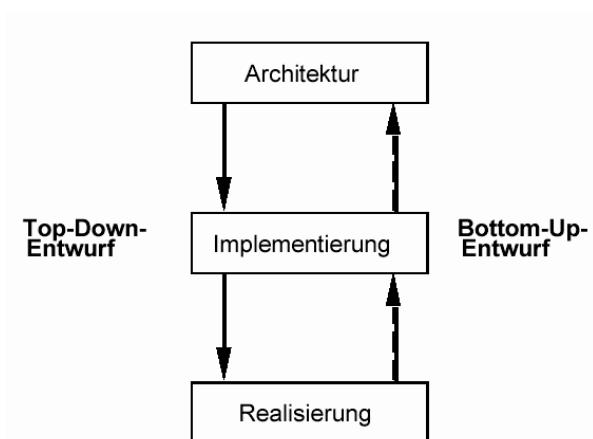
Immer größer werdende Lücke zwischen Verarbeitungsgeschwindigkeit von Prozessoren und Zugriffsgeschwindigkeit der DRAM-Speicherchips des Hauptspeichers

Der ultimative Mikroprozessor (Nature, Bd. 406, S.1047)

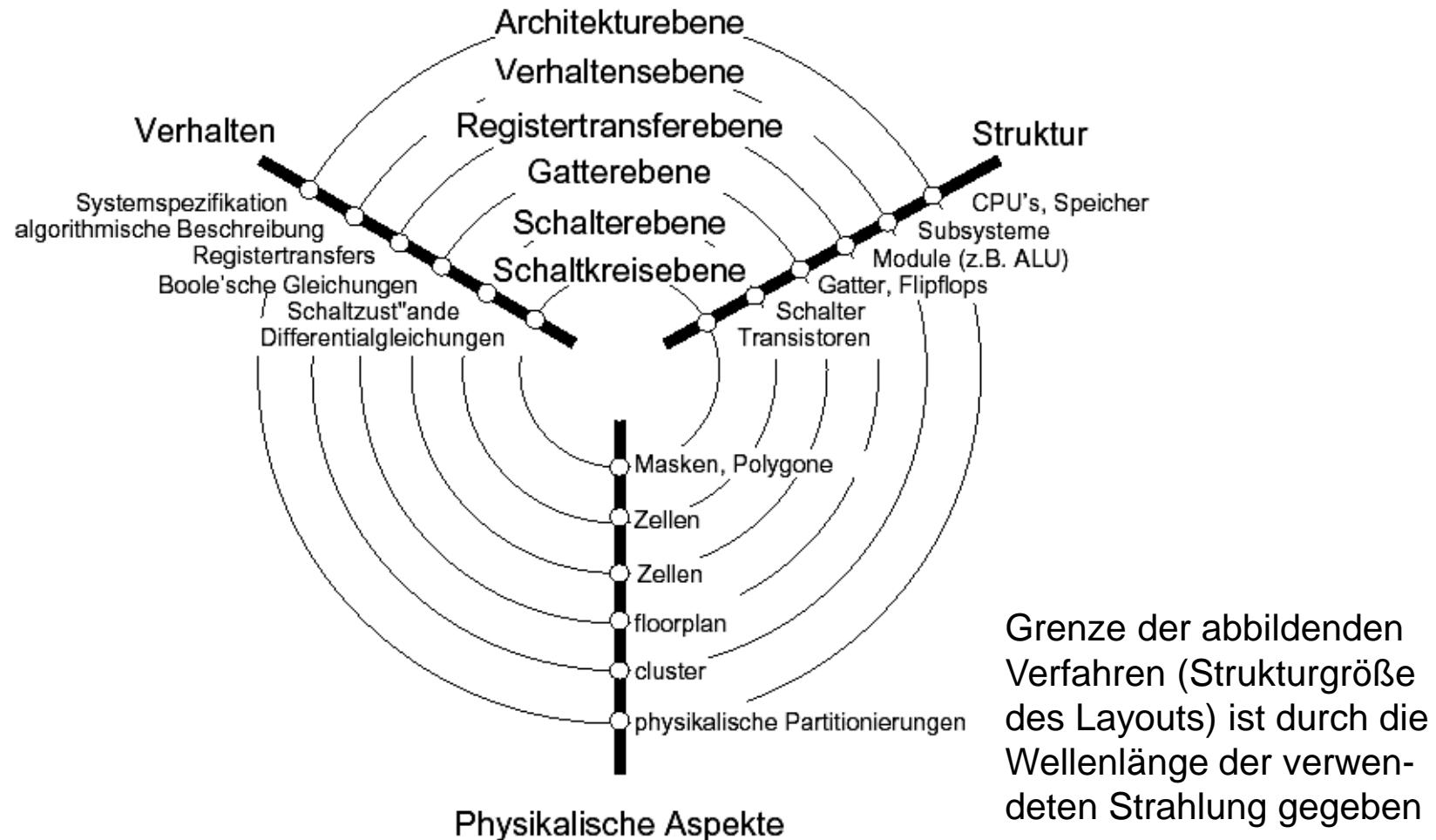
- Moores Gesetz 250 Jahre lang weitergeführt
- 10^{51} Maschinenbefehle pro Sekunde (MIPS)
- Einhundert Sextillionen Mal schneller als heutige Hochleistungsprozessoren
- Packungsdichte wie die Materie in einem schwarzen Loch
- Wird beim Einschalten heißer als die Sonne

Was wirkt noch begrenzend ?

- Maximale Anzahl der am Chipumfang anbringbaren Ein-/Ausgabe-Anschlüsse (z.B. Pentium 4 über 270 Pins)
- **Vollständige Testbarkeit ist nicht mehr erreichbar**
 - Würde bei rund 260 Pins (130 Eingänge, 130 Ausgänge) jede Funktion des Chips im Black Box-Modell in einer Nanosekunde getestet, würde der vollständige Test mehr als 10^{24} Jahre dauern (exhaustive testing).
- Leistungsaufnahme des Chips
- Mit welchen Entwurfsmethoden werden diese komplexen Chips entworfen ?
 - Idealvorstellung: **Silicon Compiler**



Dimensionen des Entwurfs (Y-Modell)



Historische Entwicklung der Rechenmaschinen

Historische Entwicklung der Rechenmaschinen

- **1642: Blaise Pascal**

Erste funktionierende Rechenmaschine
(Addition und Subtraktion);
rein mechanisch, betrieben mit einer Handkurbel

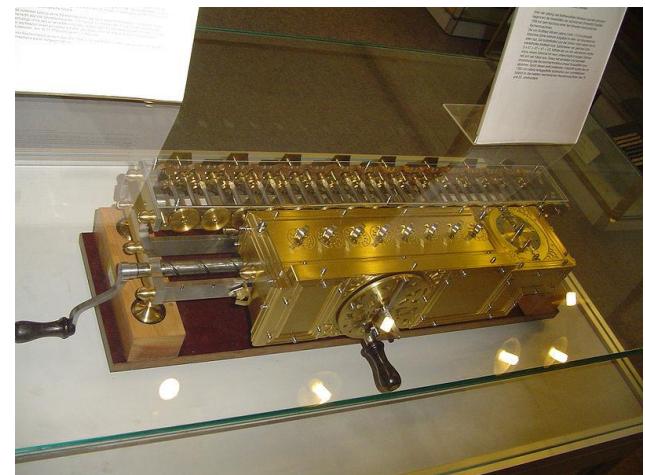


[http://commons.wikimedia.org/wiki/File:
Arts_et_Metiers_Pascaline_dsc03869.jpg](http://commons.wikimedia.org/wiki/File:Arts_et_Metiers_Pascaline_dsc03869.jpg)

- **1672: Gottfried Wilhelm Leibniz**

Rechenmaschine mit Staffelwalzen für die vier Grundrechenarten

Ab 1818 wurden die Rechenmaschinen serienmäßig hergestellt

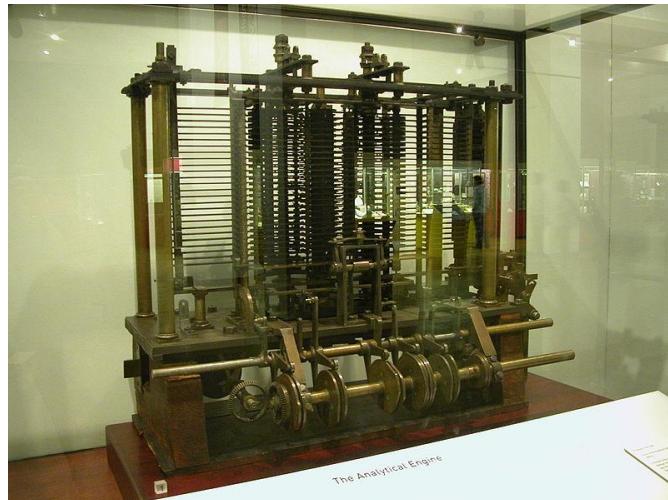


[http://commons.wikimedia.org/wiki/File:
Leibnitzrechenmaschine.jpg](http://commons.wikimedia.org/wiki/File:Leibnitzrechenmaschine.jpg)

Charles Babbage (1792-1871)

□ Difference Engine:

- Addition und Subtraktion
- Diente der Berechnung von Zahlentabellen für die Schiffnavigation
- Führte nur einen einzigen Algorithmus (Methode der finiten Differenzen mit Hilfe von Polynomen)
- Ergebnisse wurden auf einer Kupferplatte gestanzt.



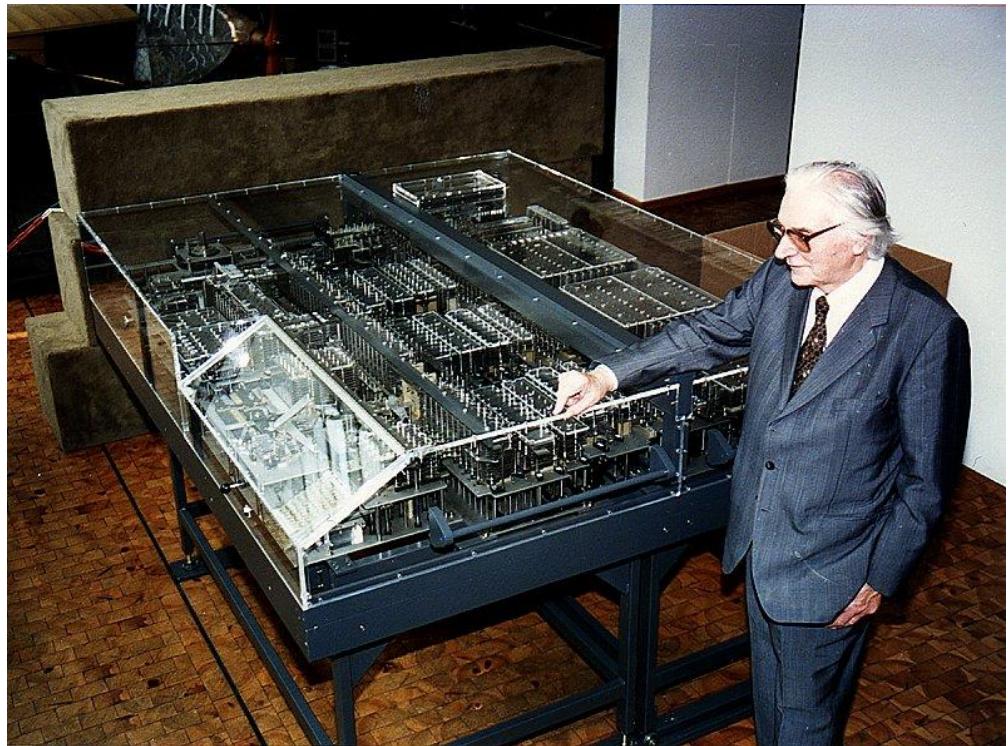
1936: Konrad Zuse

Baute eine Reihe von programmgesteuerten Rechenmaschinen mittels elektromagnetischer Relais

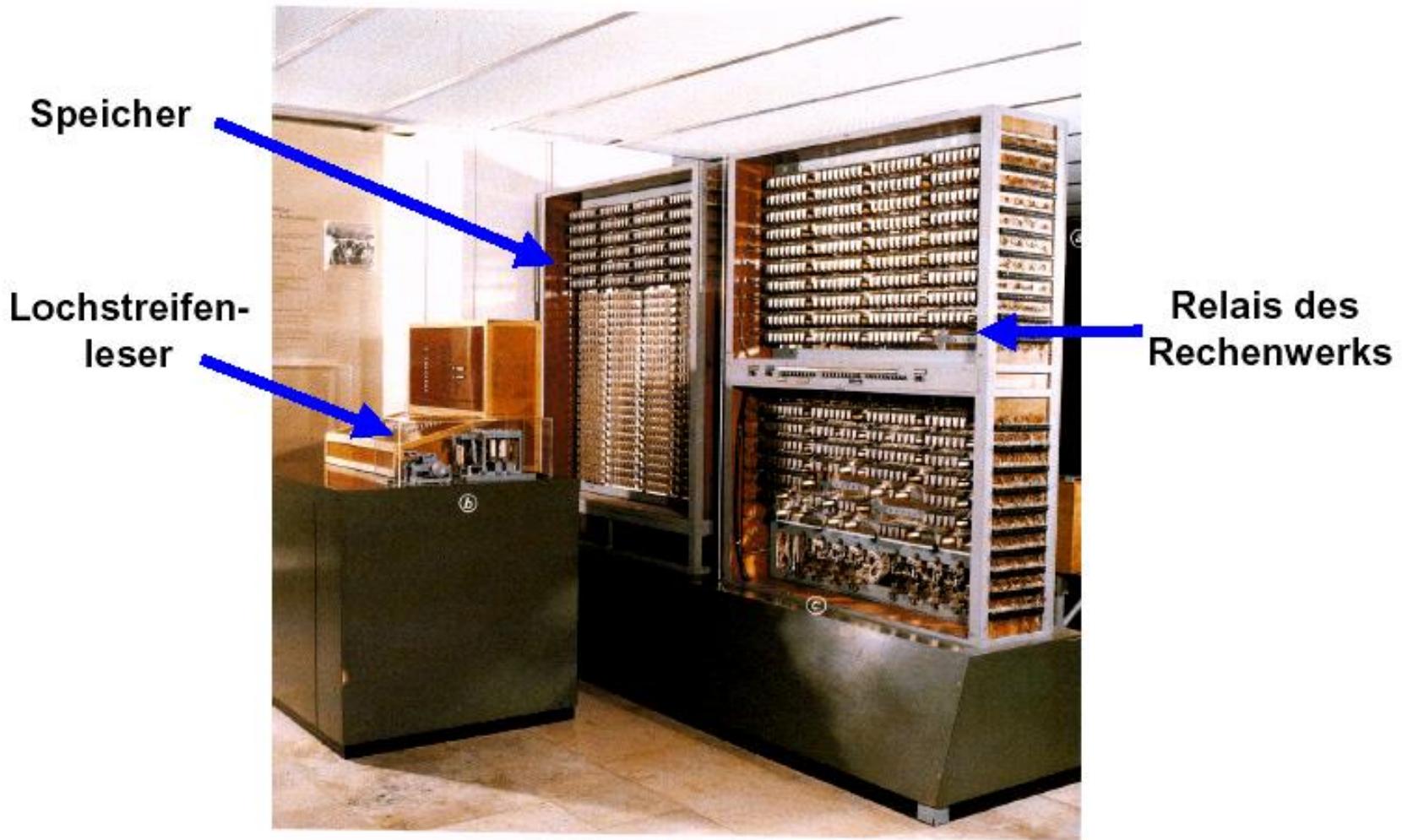
- Speicher, Eingabewerk, Rechenwerk, Plansteuerwerk und Ausgabewerk.
- Anwendung des Dualsystems und der halblogarithm. Zahlendarstellung (Gleitkommadarstellung) sowie des Aussagenkaküls
- Gebaute Maschinen (Z1, Z2, Z3 und Z4)
- Die Maschinen wurden 1944 zerstört
Nachbau der Z3 steht im Deutschen Museum in München

Historische Entwicklung der Rechenmaschinen

Konrad Zuse



Historische Rechenmaschinen - die Z3



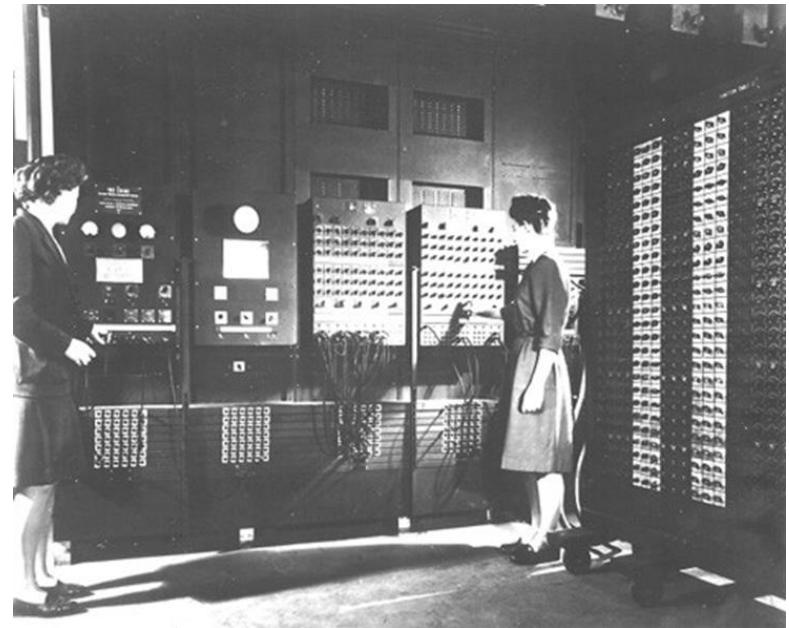
1938: Howard Aiken

- Erster programmgesteuerter Rechenautomat der USA
(Harvard Mark I)
- Dezimales Zählrad-Prinzip
- Sehr große Maschine
- Relativ schnell

Addition von 23-stelligen Dezimalzahlen in 0,3 sec

Multiplikation in 65 sec und Division in 115 sec

- Zur Ein- und Ausgabe wurden gelochte Paperbänder benutzt
- Mark II: Aikens Nachfolgermodell

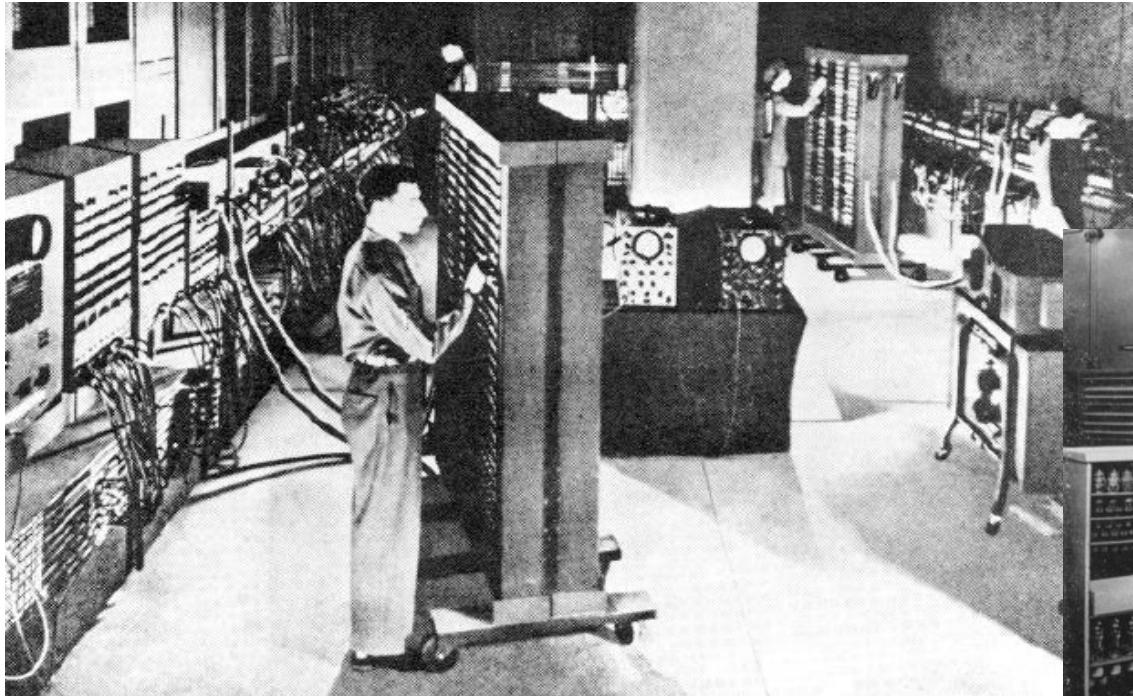


1943-1949: P. Eckert, J.W. Mauchly

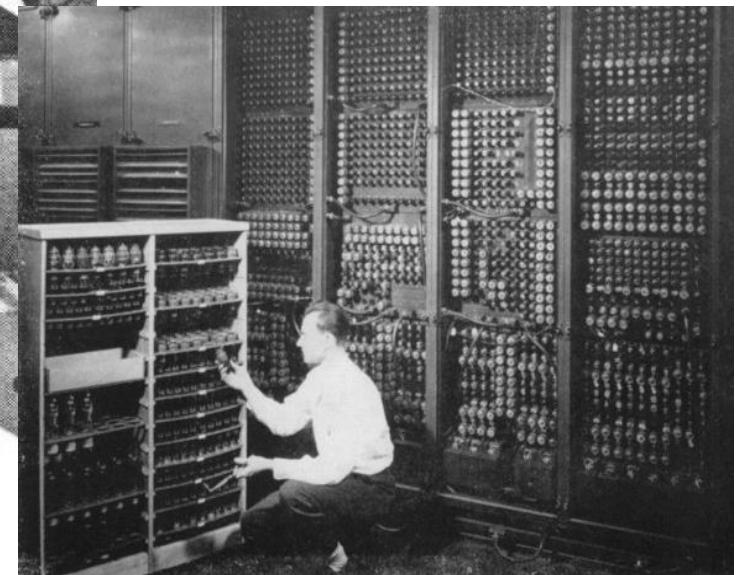


- Bau des Rechenautomaten **ENIAC** (**E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer)
- Erstmals Anwendung elektronischer Schaltelemente
- 17468 Elektronenröhren, 1500 Relais
- Gewicht: 30 Tonnen, Leistungsverbrauch: 174 KW
- Addition von 10-stelligen Zahlen in 0,2 msec
Multiplikation in 2,8 msec
- Programmierung durch Verschalten von Schalttafeln
(Sehr umständlich und fehleranfällig)

Historische Entwicklung der Rechenmaschinen



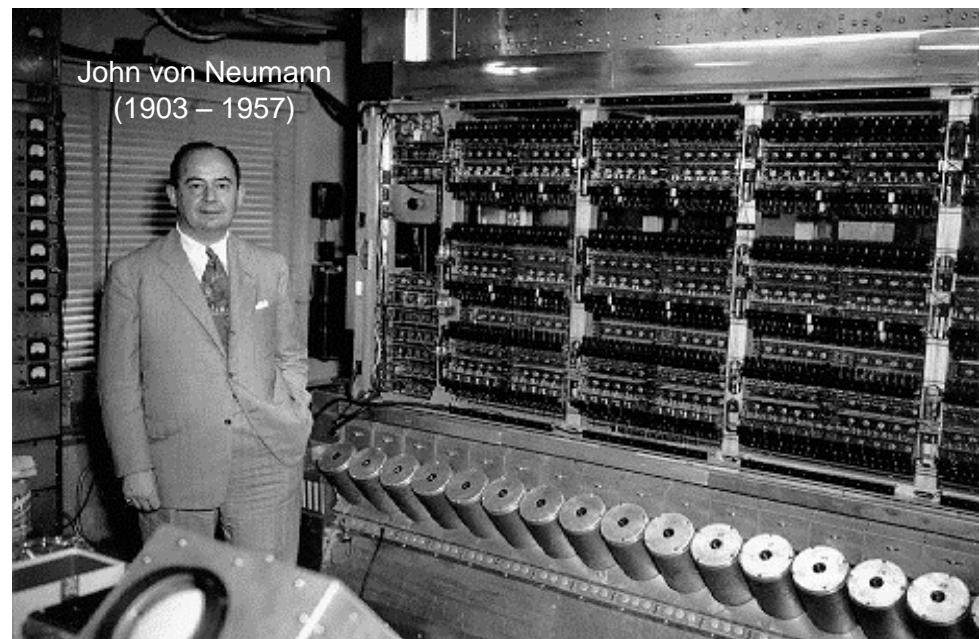
ENIAC 1946



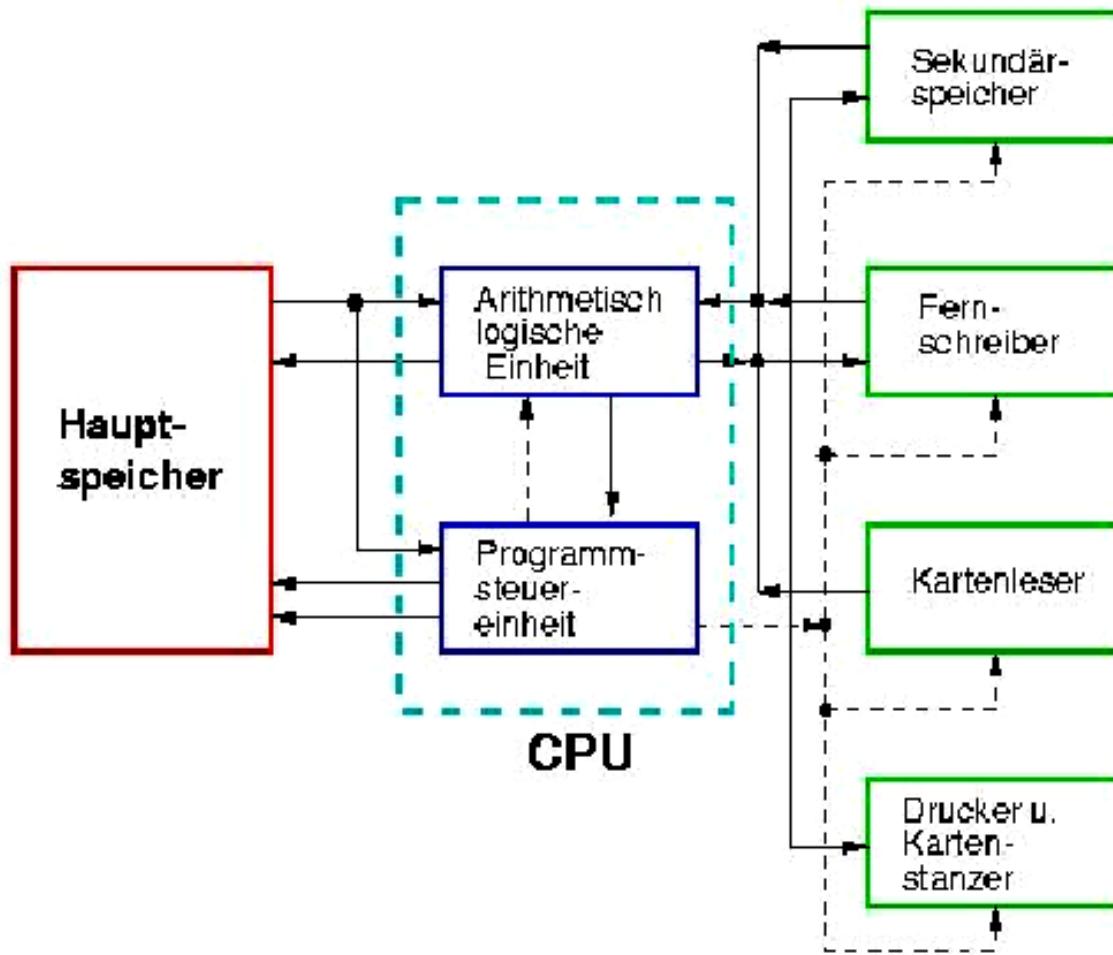
Lebensdauer einer Röhre: ca. 1000h, d.h. alle 3,3 min ein Ausfall!

1944-1946: Von Neumann, A.W. Burks, H.H. Goldstine

- Bau des Rechenautomaten **EDVAC** (**E**lectronic **D**iscrete **V**ariable **A**utomatic **C**omputer)
- Anwendung elektronischer Schaltelemente
- Programm mit Befehlen und Adressen wurde erstmals intern gespeichert und in der gleichen Art kodiert und gespeichert.
- Adressen und Befehle konnten von der Maschine selbst verändert werden
- Aufgrund bedingter Befehle war die Maschine in der Lage, den Programmablauf in Abhängigkeit von Zwischenergebnissen zu ändern



Von Neumann, A.W. Burcks, H.H. Goldstine: EDVAC



The Second Generation – Transistors (1955–1965)

- Transistor invented at Bell Labs in 1948
 - Within 10 years vacuum tube computers were obsolete
- The first minicomputer PDP1 was built by DEC in 1961
 - It costs \$120.000 compared to millions for the twice as fast transistorized computer of IBM, the IBM 7090
- CDC introduced CDC 6600 (Seymour Gray)
 - It was 10 times faster than IBM 7090
 - The CPU had multiple functional units that could run in parallel
- Integrated circuits were invented by Robert Noyce in 1958

The Third Generation – Integrated Circuits (1965–1980)

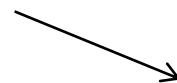
- IBM build the System/360
 - It was the first machine with multiprogramming for better CPU utilization
 - It could emulate other machines, e.g., the IBM 7094, via special microcode
 - It had a huge address space of 2^{24} bytes which was sufficient until mid 1980s
- DEC developed the PDP-11

The Fourth Generation – VLSI (1980-?)



Apple II ,1977

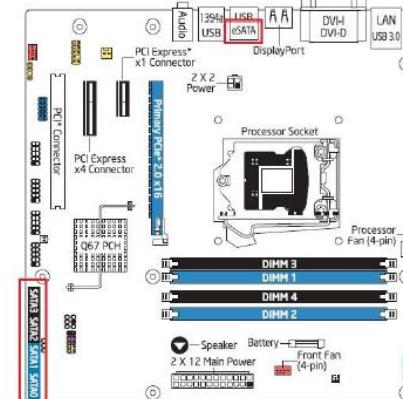
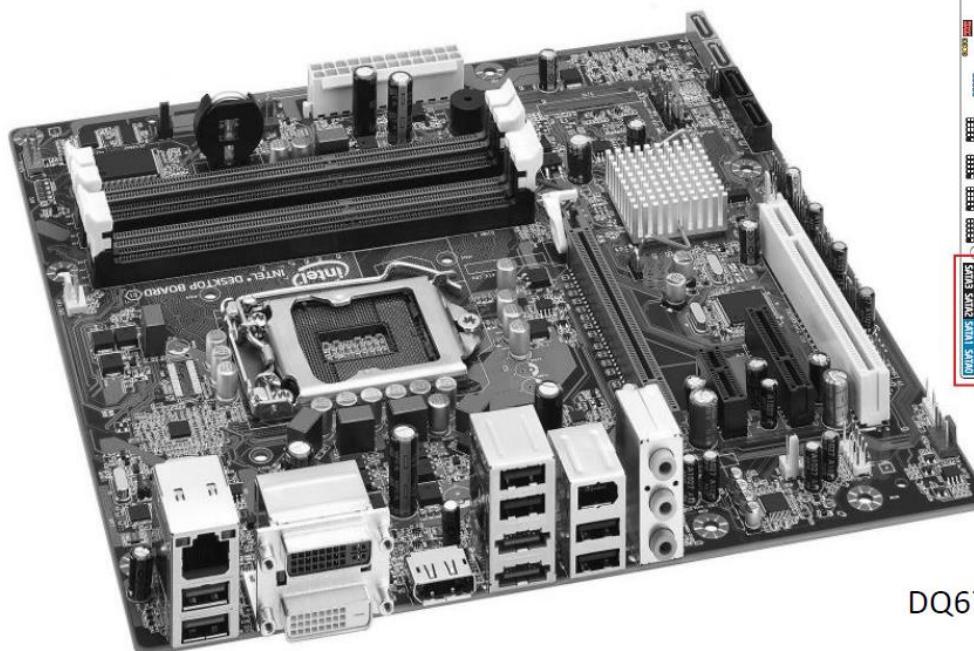
- In 1984 the Apple Macintosh was the first computer with a Graphical User Interface



- 1992 DEC produced the first 64-bit RISC processor, the Alpha processor

The Fourth Generation – VLSI (1980-?)

- Personal computer started due to price drop
 - Apple, Commodore, Atari
 - IBM PC in 1981 based on Intel CPU

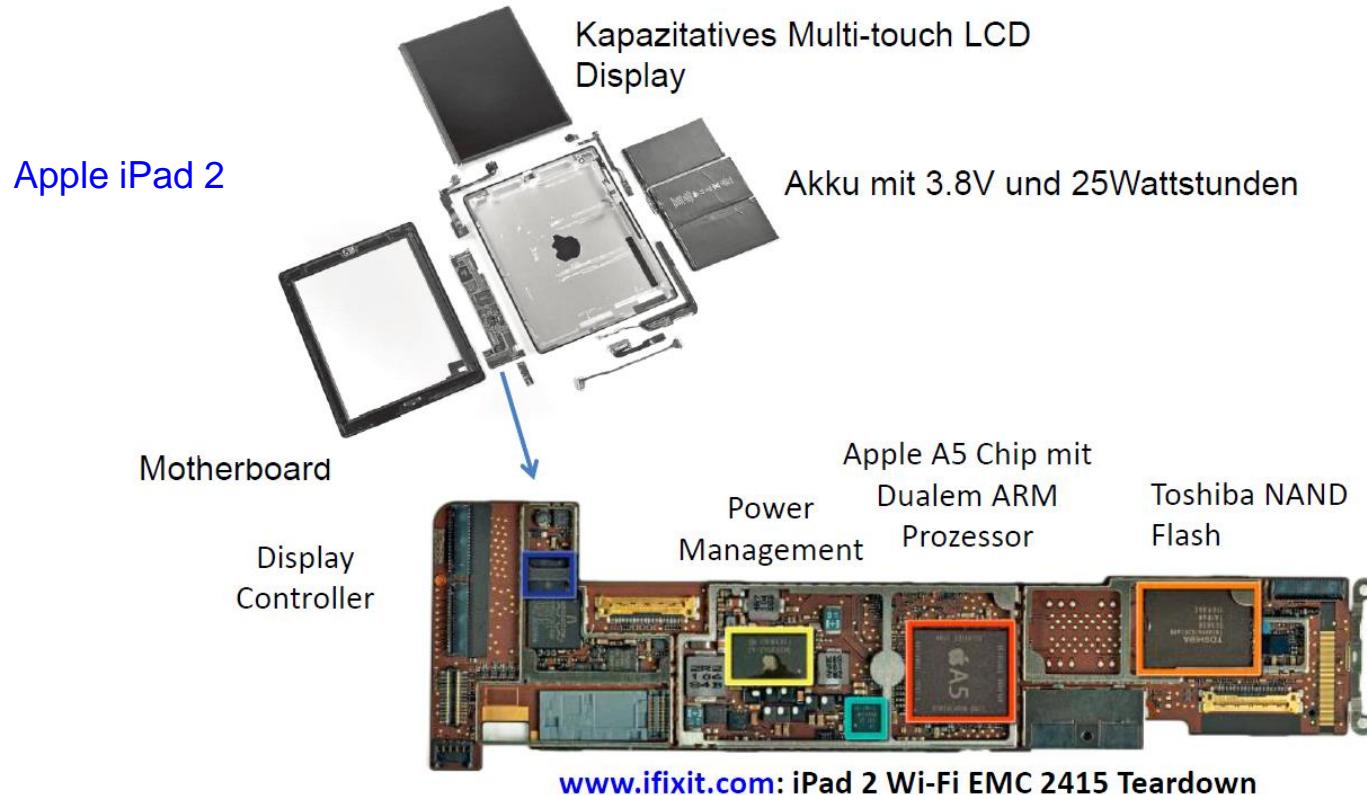


DQ67SW von Intel (Tanenbaum)

- Mid 1980s RISC was born

The Fifth Generation – Invisible Computers

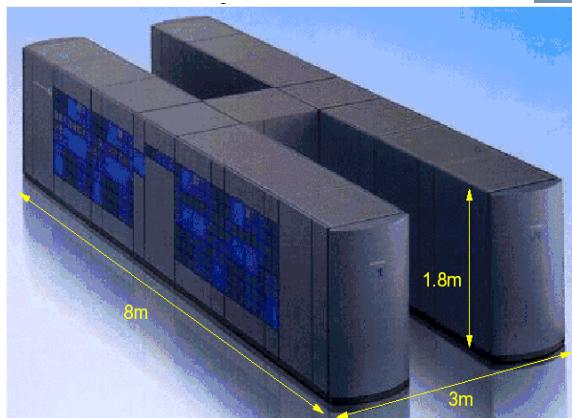
- Small embedded processors are changing the world



- Going towards **Ubiquitous Computing** or pervasive computing

Heutige Höchstleistungsrechner

Hitachi SR 8000 am
LRZ München



Heutige Höchstleistungsrechner

- **IBM JUQUEEN**
 - Platz 8 der 500 schnellsten Rechner der Welt
 - 5,9 Petaflops = rund 6 Billiarden Rechenoperationen pro Sekunde
 - 458.752 Rechenkerne in 28 Racks
 - entspricht ungefähr 100.000 modernen PCs

Zum Vergleich

- IBM Jugene (2008)
 - 825,5 TFlops
 - 294.912 Prozessoren
 - in 72 Racks

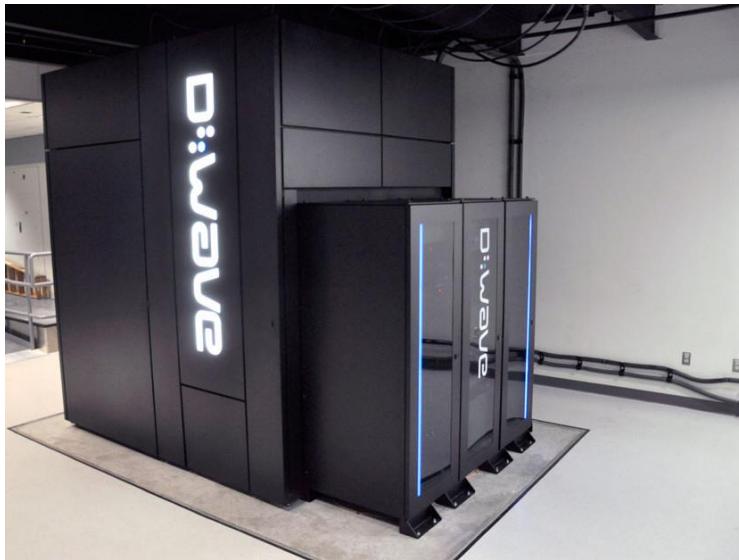


Superrechner im Forschungszentrum Jülich

Zukünftige Entwicklung

Quantencomputer

- Theorie basiert auf den Gesetzen der Quantenmechanik, statt Bits -- Qubits
- Es existieren heute kleine Prototypen für bestimmte Problemklassen
- Universeller Einsatz aber noch lange nicht in Sicht



- Video von Google (Google and NASA's Quantum Artificial Intelligence Lab):
<http://www.youtube.com/watch?v=CMdHDHEuOUE>

Zitate

- “I think there is a world market for maybe five computers.”

Thomas Watson, Chairman of IBM 1943

- “Computer in the future may weigh no more than 1.5 tons.”

Popular Mechanics, 1949

- “There is no reason for any individual to have a computer in their home.”

Ken Olson, President, Chairman and Founder
of the Digital Equipment Corp., 1977

- “640K ought to be enough for anybody.”

Bill Gates, 1981
(though today he denies he said it)

Historische Entwicklung von Mikroprozessoren

1. Generation

Jap. Firma Busicom beauftragt die gerade 3 Jahre alte Firma Intel, einen Chipsatz für Tischrechner zu entwickeln

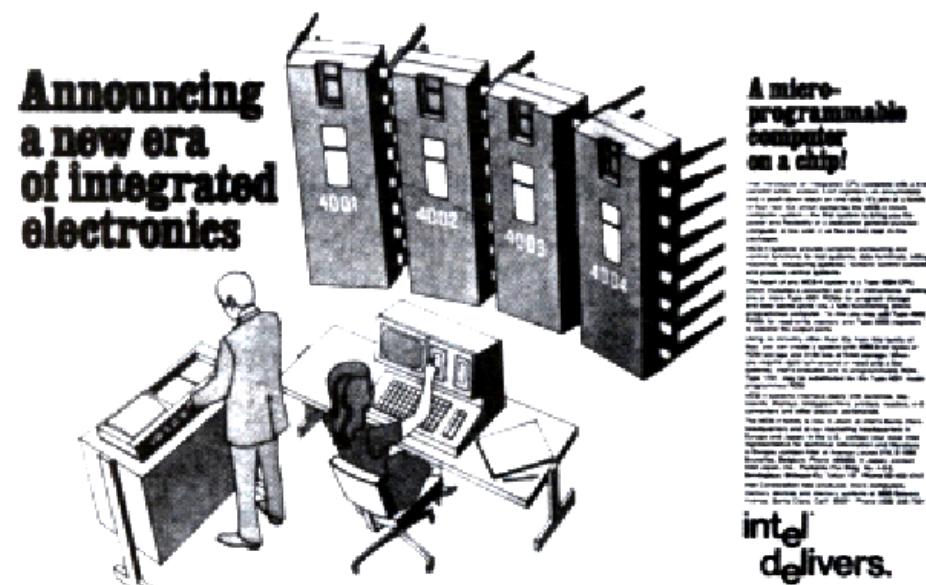
Da Intel bisher Speicher entwickelt (erstes EPROM 1701 im Jahr 1969), wurde hierfür als Konzept keine festverdrahtete Logik, sondern eine programmierbare Logik gewählt

- Intel entwickelt einen Chipsatz bestehend aus 3 Chips mit je ca. 2000 MOS-Transistoren.

Komponenten:

Festwertspeicher, RAM, Zentraleinheit (CPU), 4 Bit BCD ALU, 4 Bit Datenbus, 12 Bit Adressbus, 45 Befehle

Entwicklungszeit ca. 9 Mannmonate



1. Generation

Die Zentraleinheit wurde 1971 unter der Bezeichnung **Intel 4004** als Kernstück eines Mikrorechnersystems (MCS-4) angeboten.

Fachzeitschrift: *Electronics News*

Historische Entwicklung von Mikroprozessoren

2. Generation

1972: Intel 8080

- 8 Bit Prozessor mit erhöhter Rechengeschwindigkeit
- Instruktionszeit ca. 2 Mikrosekunden
- ca. 5000 Transistoren
- ab 1974 in NMOS-Technologie (einfache 5 Volt Versorgungsspannung)
- ca. 75 Befehle (dazu Unterstützung externer Unterbrechungen und Unterprogrammsprünge)
- 8 Bit Datenbus, 16 Bit Adressbus (64kByte)
- Industriestandard

3. Generation

1978: 8086 von Intel

- Erster 16 Bit Prozessor von Intel
- HMOS Technologie (High Density MOS)
- ca. 27000 Transistoren (aber 30% mehr Fläche als ein 8080)
- virtuelle Speicherverwaltung
- 16 Bit Datenbus, 20 Bit Adressbus (1MByte)

Historische Entwicklung von Mikroprozessoren

3. Generation

1979: 68000 von Motorola

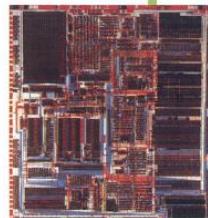
- ▷ 16 Bit Prozessor, intern jedoch 32 Bit Registersatz
- ▷ HMOS Technologie
- ▷ ca. 68000 Transistoren
- ▷ 24 Bit Adressbus (16 MByte)
- ▷ orthogonaler Befehlsatz wie bei Minicomputern üblich

1979: Z8000 von Zilog

16 Bit Prozessor, Nachfolger des Z80

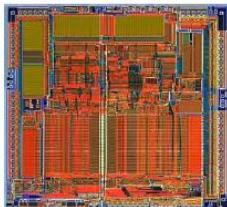
1985: 80386 von Intel

- 32 Bit Prozessor
- CMOS Technologie
- 275 000 Transistoren
- virtuelle Speicherverwaltung, Segmentierung, Paging



1986: 68020 von Motorola

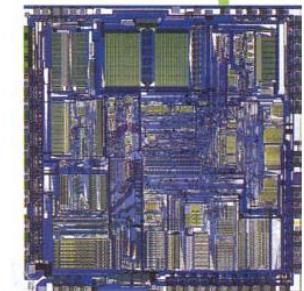
- 32 Bit Prozessor
- ca. 200 000 Transistoren
- virtueller Adressraum



1982:

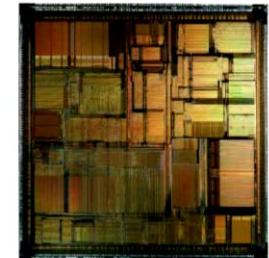
80286 von Intel

- Nach dem 80186 der zweite Nachfolger des 8086
- ca. 130000 Transistoren
- Erweiterter Adressraum (16 Mbyte)
- Mehrere Betriebsarten
- Multitasking-Unterstützung
- Jahrelang in vielen Personal Computern (z. B. IBM AT) eingesetzt



RISC Mikroprozessoren:

- Advanced Micro Devices Am29000 (~1987)
- Sun Microsystems SPARC (April 1987):
 - 32 Bit CPU mit über 55.000 Transistoren
 - Alle Befehle waren 32 Bit Breit
 - Ausführungszeit lag bei nur 1.3 Takt pro Befehl
 - Eine Sun war 3 mal schneller als 386er Rechner bei einem Fünftel dessen Komplexität.
 - Der SPARC verfügte in der ersten Version über 128 Register
- MIPS technologies (MIPS R2000 / MIPS R3000)



Sun Microsystems SPARC

Historische Entwicklung von Mikroprozessoren

4. Generation

1992: Pentium von Intel

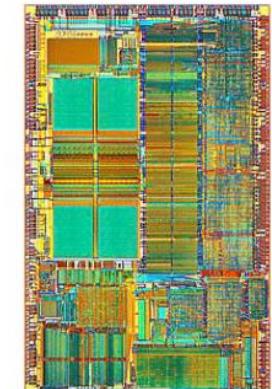
- ▷ Nachfolger des 80486
- ▷ ca. 3 100 000 Transistoren
- ▷ intern teilweise 64 Bit Architektur
- ▷ 2 fach Superskalar, Code und Datencache

1992-95: Power PC's MPC601, MPC603, MPC604, MPC620 von Motorola/IBM/Apple

- ▷ RISC Architektur, teilweise 64 Bit Architektur (Daten)
- ▷ Superskalar
- ▷ ca. 4 000 000 Transistoren (MPC620)

➤ 1989: 80486 von Intel

- Erweiterung des 80386 um integrierten Cache und integrierten numerischen Coprozessor
- ca. 1 200 000 Transistoren
- Multiprozessor-Unterstützung

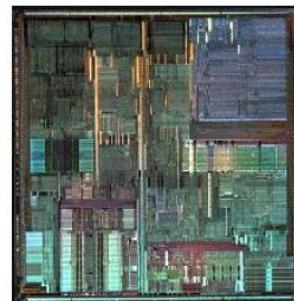


➤ 1990: 68040 von Motorola

- Nach 68030 zweiter Nachfolger des 68000
- ca. 1 200 000 Transistoren

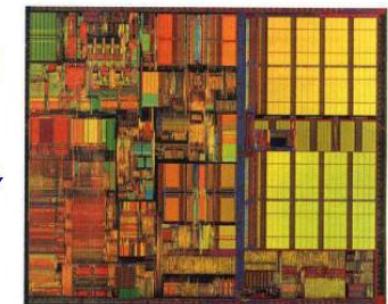
➤ 1995: Pentium Pro von Intel

- Nachfolger des Pentium
- ganz anderer interner Aufbau
- 3-5 fach Superskalar
- ca. 14stufige Befehlspipeline
- 5 500 000 Transistoren
- Zwei eingebaute Cache-Speicher-Ebenen
- speculative execution, dynamic branch prediction



➤ 1998: Pentium III

- Nachfolger vom Pentium II mit Internet Streaming SIMD Extension (ISSE)
- (*SIMD = Single Instruction, Multiple Data*)
- 16 KByte Daten- und Befehls-Cache mit vollem Prozessortakt.
- 2nd-Level-Cache mit halbem Prozessortakt
- Anbindung an die Außenwelt über einen mit 100 - 133 MHz arbeitenden Systembus.

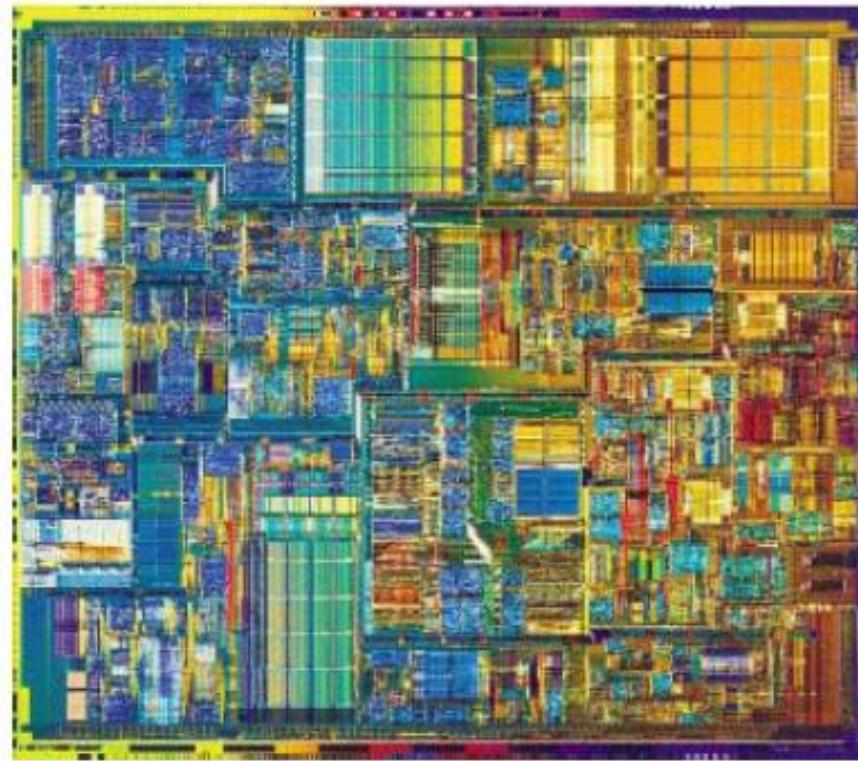


Historische Entwicklung von Mikroprozessoren

4. Generation

2000 – 2008: Pentium 4:

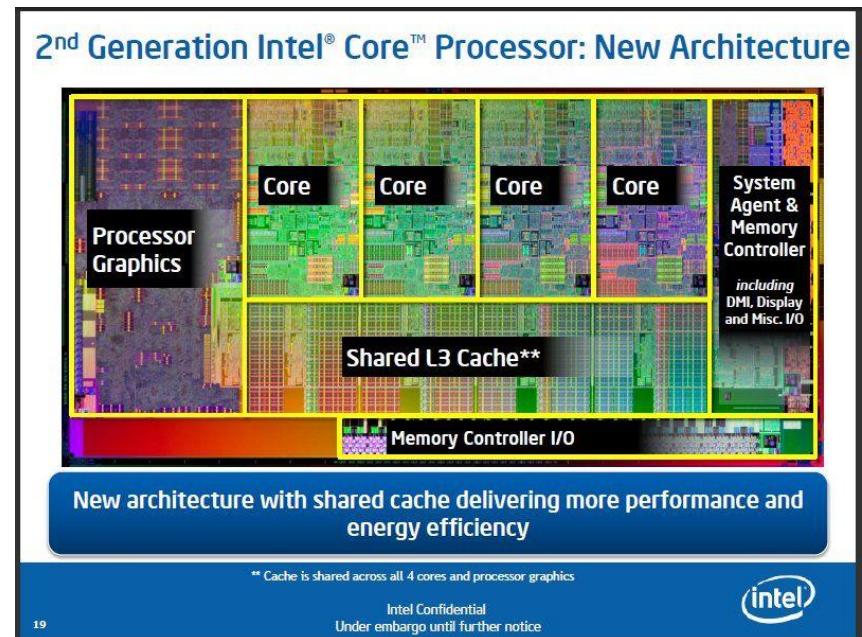
- komplette Neuentwicklung
- Intel® NetBurst™ micro-architecture
- Nachfolger vom Pentium III mit Internet Streaming SIMD Extensions 2
- Enhanced floating point/multimedia
- Advanced dynamic execution
- Hyper-threading technology
- Execution trace cache and advanced transfer cache
- 400MHz System Bus



Historische Entwicklung von Mikroprozessoren

Ab 2010: Core i5

- Intel Core-Mikroarchitektur löste Netburst-Architektur ab
- 4 Prozessorkerne auf einem Die
- Front Side Bus (FSB) wird durch Quick-Path-Interconnect (QPI) ersetzt (Punkt-zu-Punkt, besserer Durchsatz bei Verbindung CPU-Chipsatz)
- Anbindung des Arbeitsspeichers über integrierten Speichercontroller
- Simultaneous Multithreading (SMT)
- Dreistufige Cache-Architektur
- 14-stufige Pipeline



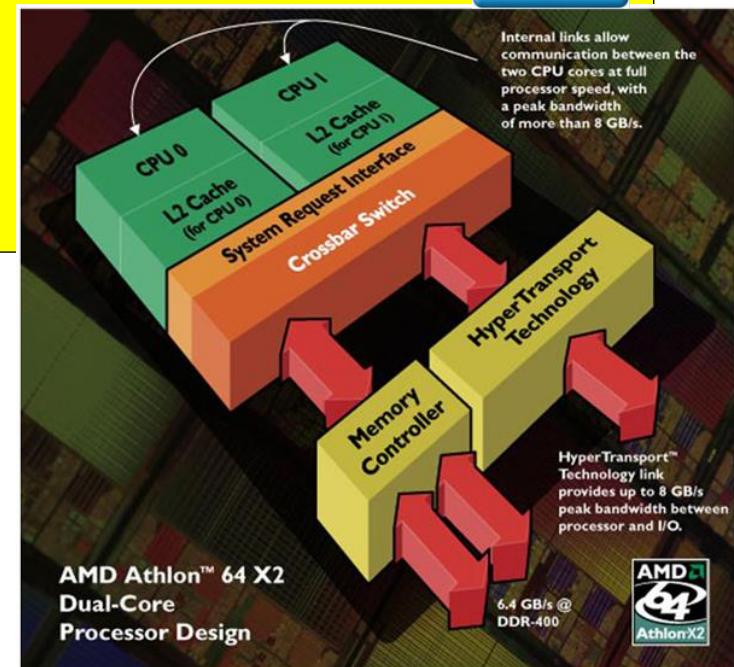
Entwicklung von Mikroprozessoren

- Higher clock rates
 - Increase power consumption
 - Energy proportional to f and U^2
 - Built higher frequency needs higher voltage
 - Increase heat output and cooling requirements
 - Limit chip size (speed of light)

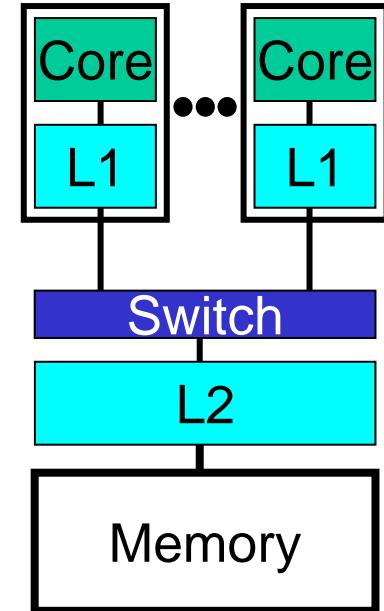
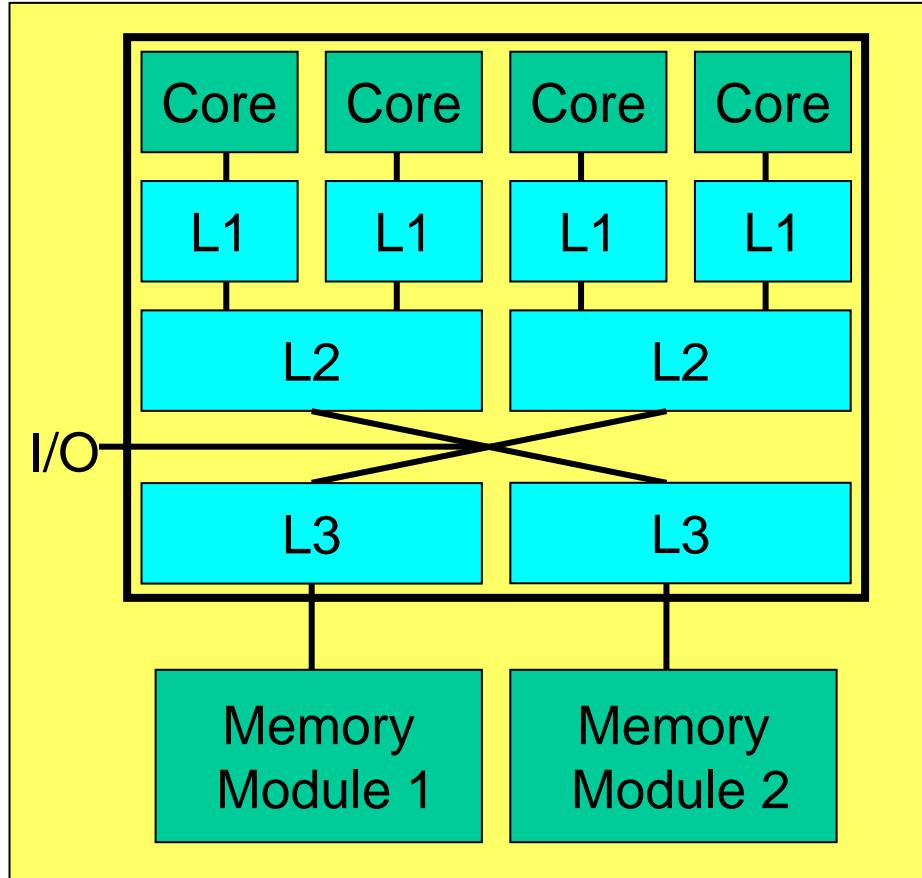
- More parallelism
 - Increased bit width (now: 64 bit architectures)
 - Instruction Level Parallelism (ILP)
 - Exploits parallelism found in a instruction stream
 - Limited by data/control dependencies
 - Can be increased by speculation
 - Average of ILP in typical programs: 6-7

Entwicklung von Mikroprozessoren

- More parallelism
 - Thread Level Parallelism (TLP)
 - Hardware multithreaded (e.g. SMT: Hyperthreading)
 - Better exploitation of superscalar execution units
 - Multiple cores
 - Legacy software must be parallelized
 - Challenge for whole software industry

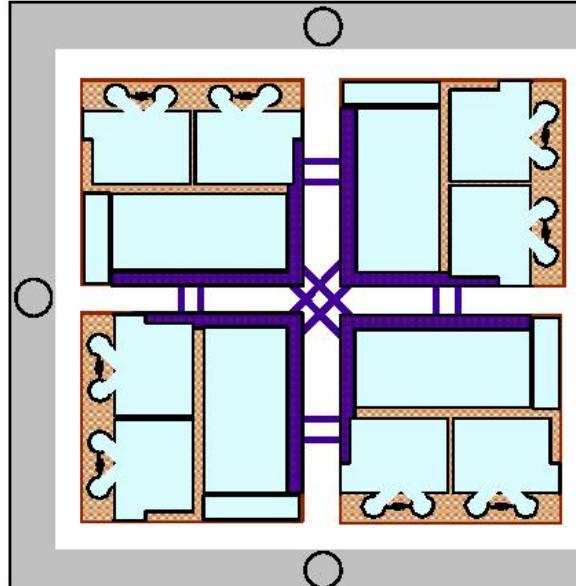


Entwicklung von Mikroprozessoren



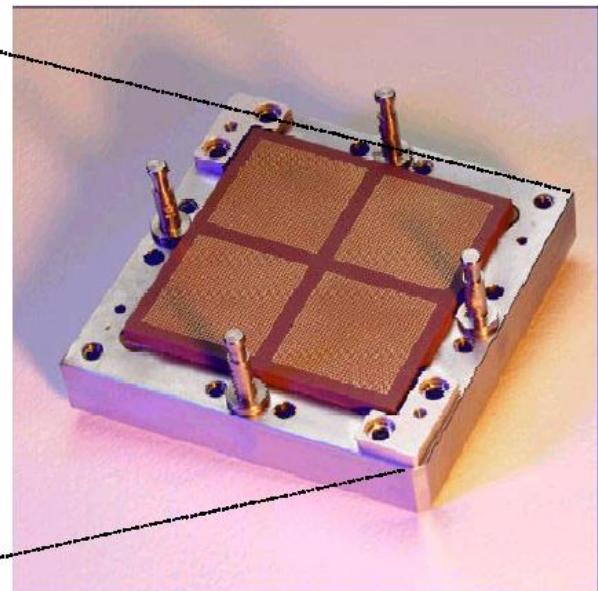
Homogeneous with
shared caches
and cross bar

8 – Prozessor - Multichipmodul



8-processor Multi-Chip Module

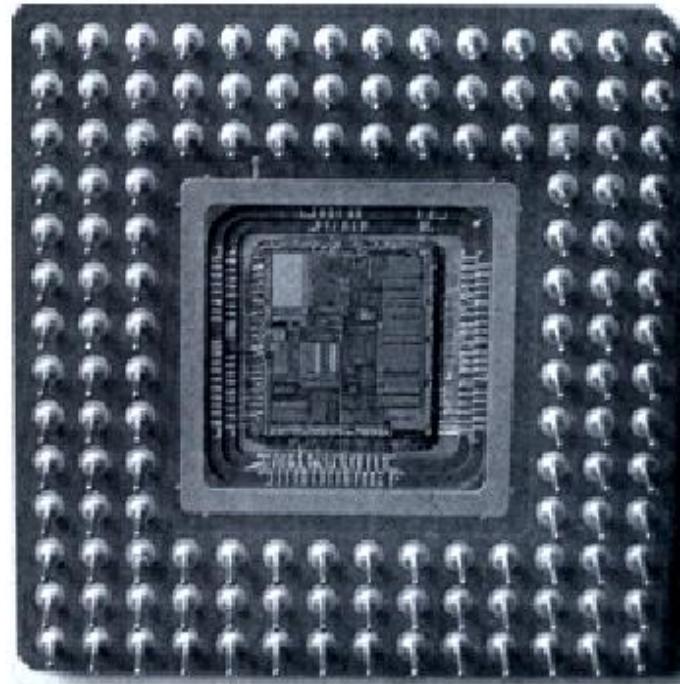
- Chips rotated to allow faster interconnection



4.5" square, glass-ceramic construct

Gehäuse und Anschlüsse

Pin Grid Gehäuse mit 132 Anschlüssen, 100 mm² großer Chip:



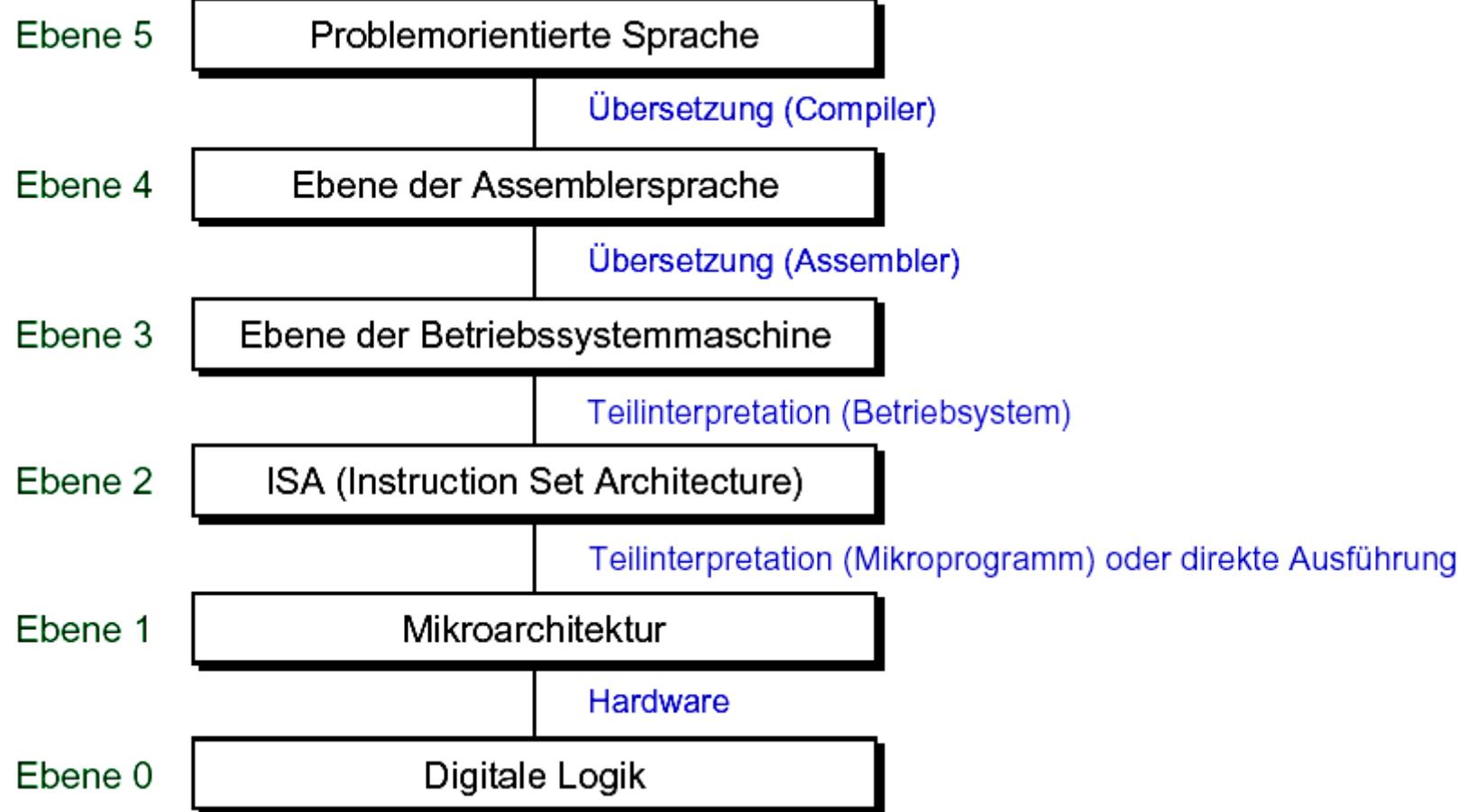
Die Verbindungen der Chip-Anschlüsse an die Pins erfolgt mittels Golddrähten (bonding)

Der Begriff „Rechnerarchitektur“

Ebenenmodell eines Rechners

- Computer können beispielsweise durch ein Ebenenmodell (oder Schichtenmodell) abstrakt beschrieben werden. Jede Ebene baut auf der jeweils vorhergehenden auf.
- Die auf jeder **Ebene verfügbaren Datentypen, Operationen, Elemente und Merkmale nennt man Architektur.**
Die Architektur betrifft jeweils die Aspekte, die für den Benutzer auf der jeweiligen Ebene sichtbar sind. Technologische Details, wie etwa die verwendete Chiptechnologie, sind nicht Teil der Architektur.
- Im folgenden ist die Modellierung eines Rechners mit 6 Ebenen dargestellt (*nach Tanenbaum*).

Ebenenmodell eines Rechners mit 6 Ebenen



Bedeutung der verschiedenen Ebenen

- Ebene 0: Digitale Logik
 - Bauelemente der Digitaltechnik
 - Gatter, PLDs, Multiplexer, Demultiplexer, Register
- Ebene 1: Mikroarchitektur
 - Strukturen werden aus Bauelementen der Ebene 0 gebildet, wie z.B. ALU, Datenpfade, Pipelines,.. Der Fluss der Daten wird durch Programme (Mikroprogramme) oder direkt durch Hardware (RISC) gesteuert. Ein Mikroprogramm ist ein Interpreter für Anweisungen der Ebene 2.
- Ebene 2: Instruction Set Architecture
 - Menge der Instruktionen, die von einem Prozessor direkt interpretativ ausgeführt werden können (Maschinensprache). Diese Ebene ist prozessorspezifisch.

Bedeutung der verschiedenen Ebenen

- Ebene 3: Betriebssystemmaschine
 - Erweiterung der Ebene 2 Funktionen durch neue Funktionen, wie Speicherorganisation, Mehrprogrammbetrieb,... sogenannte Betriebssystem-Funktionen. Diese werden von einem Interpreter ausgeführt, der auf Ebene 2 läuft.
- Ebene 4-5: Assembler- und problemorientierte Sprachen
 - Werden durch Übersetzung (Assembler, Compiler) oder Interpreter unterstützt. Übersetzung als Dienstleistung für die unteren Schichten

Übersetzung und Interpretation

Übersetzung

- Ein **L_i -Compiler** ist eine Software, die ein in der Sprache L_i geschriebenes Programm einliest und ein äquivalentes Programm der Sprache L_j ($j < i$) ausgibt
- Ist L_j die Maschinensprache, so spricht man von einem **L_i -Vollcompiler**.

Ausführung eines L_i -Programms

- Transformation in ein äquivalentes L_j -Programm ($j < i$)
- Ausführung des L_j -Programms

Interpretation

cmd := erste Instruktion des L_i -Programms;

repeat

transformiere *cmd* in eine Befehlssequenz *cmd2* der Sprache L_j ;

führe *cmd2* auf Ebene j aus;

cmd := nächste auszuführende Instruktion des L_i -Programms;

until L_i -Programm abgearbeitet;

Begriff Rechnerarchitektur

Eine heute noch weitverbreitete Auffassung beruht auf der Definition:

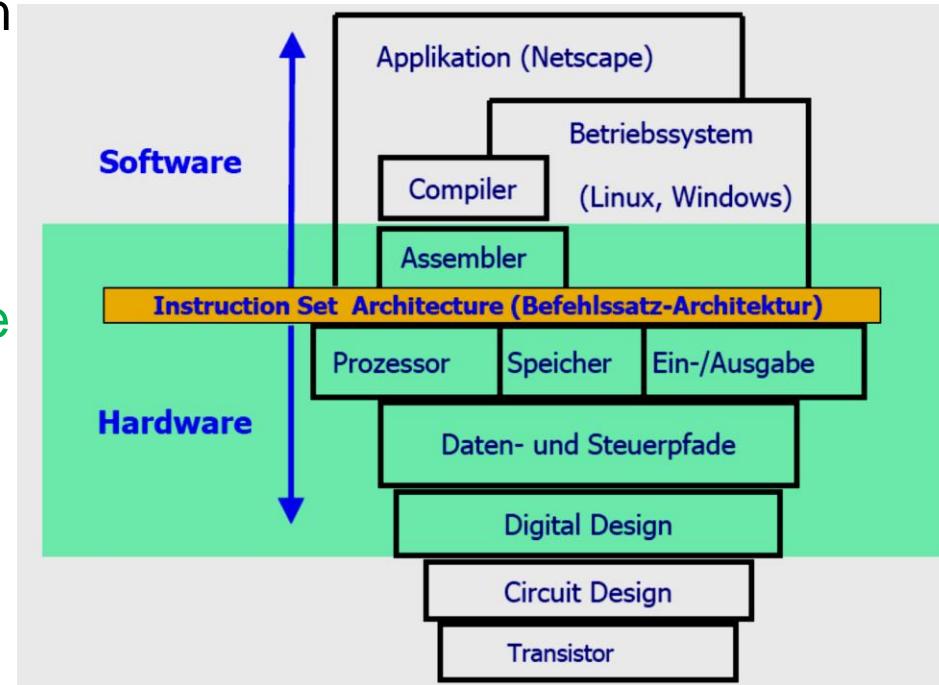
- *Computer architecture is defined as the attributes and behavior of a computer as seen by a machine-language programmer. This definition includes the instruction set, instruction formats, operation codes, addressing modes, and all registers and memory locations that may be directly manipulated by a machine language programmer.* (Amdahl, Blaauw und Brooks (IBM, 1967))
- *The architecture must therefore carefully describe the behavior that a machine-language programmer sees, but must not describe the means by which a particular implementation achieves that behavior.* (Architekten der DEC Alpha (1992))
- Diese Definition behandelt nur das äußere Erscheinungsbild des Rechners (Verhalten).

Instruction Set Architecture (ISA)

- Die Sichtweise eines (Maschinen-)Programmierers kann mit den folgenden Fragen beschrieben werden
 - Welche Datenformate gibt es ?
 - Wie können die Daten gespeichert werden ?
 - Wie kann auf die Daten zugegriffen werden ?
 - Welche Befehle können auf die Daten angewandt werden ?
 - Wie sind die Befehle codiert ?
- Die Beantwortung dieser Fragen führt zu der sogenannten **Instruction Set Architecture (ISA)**.
- ISA beschreibt das funktionale Verhalten eines Prozessors aus der Sichtweise eines Maschinen-Programmierers (Abstraktion).

Instruction Set Architecture (ISA)

- Zur Spezifikation der ISA gehören
 - Befehlssatz
 - Befehlsformat
 - Adressierungsarten
 - Datentypen und Datenformate
 - Register und Speichermodell
 - Unterbrechungssystem
- Die Instruction Set Architecture betrifft jedoch keine Details der Hardware und der technischen Ausführung, sondern beschreibt eine abstrakte Schnittstelle zwischen der Hardware und der untersten Softwareebene der Maschine.



Mikro-Architektur

- Die Mikro-Architektur betrifft die aktuelle Hardware-Struktur, den Entwurf der Hardware-Logik und der Datenpfade einer speziellen Verkörperung der Architektur - also einen konkreten Mikroprozessor.
- Implementierungstechniken:
 - Die Art und Stufenzahl des Befehlspipelining
 - der Grad der Verwendung der Superskalar-Technik
 - Art und Anzahl der internen Ausführungseinheiten eines Mikroprozessors sowie
 - Einsatz und Organisation von Primär-Cache-Speichern

Leistungsbewertung von Rechnern

Bewertung der Leistungsfähigkeit

- Wo wird dies gebraucht ?
 - Auswahl einer Rechenanlage, Vergleich von Rechnern
 - Veränderung der Konfiguration einer bestehenden Anlage (Tuning), Entwurf von Rechenanlagen
- Auswertung von Hardware-Leistungsparametern
 - Taktfrequenz, MIPS, MFLOPS
- Laufzeitmessungen bestehender Programme
 - Benchmarking
- Messungen während des Betriebs
 - Monitoring
- Modelltheoretische Verfahren
 - Analytische Modelle (Warteschlangenmodelle), Simulation

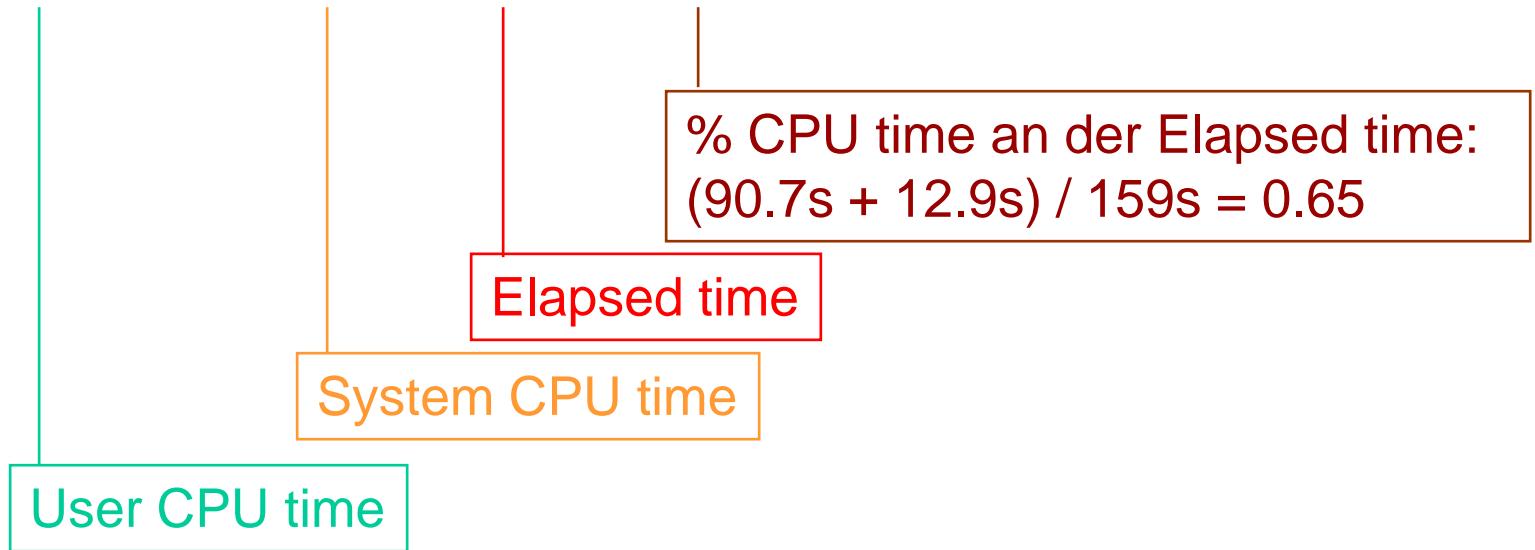
Bewertung der Leistungsfähigkeit

Hardware-Parameter

- Zykluszeit (Taktperiode der CPU)
- Befehlsausführungszeit, Cycles per Instruction
- Speicherzugriffszeit
- Speicherbandbreite
 - Enpgaß Speicherzugriff:
 - 1988: Takt: 25 MHz, Periode: 40 ns
 - Speicherzugriff: 200 ns = 5 Taktzyklen
 - 2001: Takt: 1000 MHz, Periode: 1 ns
 - Speicherzugriff: 100 ns = 100 Taktzyklen

Programmausführungszeiten: z. B. Unix Time Kommando

90.7s, 12.9s, 2:39 65%



- **Elapsed Time:** Latenzzeit für die Ausführung einer Aufgabe, schließt den Speicher- und Plattenzugriff, Ein-/ Ausgabe etc. mit ein.
- **User CPU Time:** Zeit, in der die CPU ein Programm ausführt
- **System CPU Time:** Zeit, in der die CPU Betriebssystemaufgaben ausführt, die von einem Programm angefordert werden.

Programmausführungszeiten

- Die **Gesamtausführungszeit eines Programms P** lässt sich ermitteln, indem man die Gesamtzahl der für das Programm P erforderlichen Taktzyklen aller Befehle mit der Taktzykluszeit der CPU multipliziert.
- Die **Taktfrequenz ist der Kehrwert der Taktzykluszeit.**

$$CPU_{\text{Programmausführungszeit}} = CPU_{\text{Anzahl Taktzyklen } P} \times \text{Taktzykluszeit}_{\text{CPU}}$$

$$CPU_{\text{Programmausführungszeit}} = \frac{CPU_{\text{Anzahl Taktzyklen } P}}{\text{Taktfrequenz}_{\text{CPU}}}$$

- Man sieht, dass die Programmausführungszeit entweder durch die Verringerung der Taktzykluszeit oder durch die Verringerung der erforderlichen Anzahl der Taktzyklen verbessert werden kann.

Programmausführungszeiten

- Die Anzahl der erforderlichen CPU Taktzyklen für ein Programm P lässt sich bestimmen, indem man die Anzahl der Befehle von P mit der Anzahl der Taktzyklen, die jeder Befehl benötigt, multipliziert.

$$CPU_{\text{Anzahl Taktzyklen } P} = \text{Anzahl Befehle pro Programm} \times \text{Anzahl Taktzyklen pro Befehl}$$

$$CPU_{\text{Anzahl Taktzyklen } P} = \text{Anzahl Befehle pro Programm} \times CPI$$

CPI: Clock Cycles per Instruction

- Der CPI-Wert ist ein Mittelwert über alle Befehle. Mit diesem Wert hat man die Möglichkeit verschiedene Implementierungen der ISA miteinander zu vergleichen (da die Anzahl der Befehle gleich ist)
- Liegen die Befehle in unterschiedlichen Befehlsklassen (jeweils anderer CPI-Wert), muss eine gewichtete Summe für die Summe der Taktzyklen der CPU gebildet werden (siehe Übung).

Programmausführungszeiten

- Aus den bisherigen Betrachtungen lässt sich die **Ausführungszeit eines Programms P** durch die **CPU** damit auch folgendermaßen ausdrücken:

$$CPU_{\text{Programmausführungszeit}} = \text{Anzahl der Befehle} \times \text{Taktzyklen pro Befehl} \times \text{Taktzykluszeit}$$

$$CPU_{\text{Programmausführungszeit}} = \frac{\text{Anzahl der Befehle} \times CPI}{\text{Taktfrequenz}}$$

Befehlsausführungsrate

- Eine weitere Größe, die öfters verwendet wird, ist der MIPS-Wert (MIPS: *Million Instructions per Second*).

$$\text{MIPS} = \frac{\text{Anzahl der Befehle}}{\text{Ausführungszeit} \times 10^6}$$

- MIPS-Wert entspricht der Befehlsausführungsrate, d.h. schnellere Computer haben tendenziell einen höheren MIPS-Wert.
- MIPS-Wert berücksichtigt nicht die Leistungsfähigkeit der Befehle
 - Computer mit unterschiedlichen Befehlssätzen können nicht miteinander verglichen werden
- MIPS-Werte auf demselben Computer sind von Programm zu Programm unterschiedlich
 - Nicht ein MIPS-Wert für alle Programme

Standardisierungsorganisationen

SPEC Standard Performance Evaluation Corporation

- Seit 1989, mehr als 40 Rechnerhersteller haben sich zusammen geschlossen, Festlegung von Richtlinien für eine gemeinsame Rechnerbewertung
- Ziel: Vergleichbarkeit von Rechnern
- Benchmarks
 - Bewertung der Leistungsfähigkeit aufgrund von Messungen
 - Programm oder Programmsammlung im Quellcode
 - Übersetzung notwendig
 - Messung der Ausführungszeiten
 - In die Bewertung fließt „Güte“ des Compilers und Betriebsssoftware ein

SPEC-Benchmarks

- Zahlreiche Benchmark Suites, z.B.
 - SPEC95, SPECweb96,
 - JavaSPEC,
 - GPC (graphic performance characterization)
- „The goal of SPEC is to ensure that the marketplace has a fair and useful set of metrics to differentiate candidate systems. The basic SPEC methodology is to provide the benchmarker with a standardized suite of source code based upon existing applications that has already been ported to a wide variety of platforms by its membership. The benchmarker then takes this source code, compiles it for the system in question and then can tune the system for the best results“.

SPECint95: 8 Integer-Test-Programme (ANSI C)

- go: Go-Spiel, das drei Spiele gegen sich selbst spielt,
- m88ksim: ein Simulator für den 88110-Mikroprozessor,
- gcc: der GNU-C-Compiler,
- compress: Komprimierprogramm,
- li: LISP-Interpreter,
- jpeg: JPEG-Komprimierprogramm,
- perl: PERL-Interpreter,
- vortex: Transaktions-Benchmark mit einer objektorientierten Einbenutzer-Datenbank von 40 Mbyte Größe.

Beispiel zu SPEC95

- 600 MHz Pentium III:
 $\text{SPECint95} = 24$
 $\text{SPECfp95} = 15.9$
- 450 MHz Sun UltraSPARC-II:
 $\text{SPECint95} = 19.7$
 $\text{SPECfp95} = 27.9$
- Ziel: vergleichbare Angaben für unterschiedliche Systeme
- Aber: meist nur erster Anhaltspunkt für eine Rechnerauswahl

Verbesserung der Rechnerleistung

- Fragestellung
 - Welche Auswirkungen auf die Gesamtleistung hat die Verbesserung einer Komponente des Rechners?
 - Beispiel
 - Gleitkomma-Einheit wird doppelt so schnell
 - Wie viel mal schneller werden dadurch die Programme?
 - Sicher abhängig vom Anteil der Gleitkomma-Operationen
- Gesetz von Amdahl liefert eine quantitative Antwort

Verbesserung der Rechnerleistung

- „Beschleunigung“:
- Man verwendet den Begriff der Beschleunigung um auszudrücken, wie sich die Verarbeitungsleistung einer Rechnerarchitektur verändert hat, wenn verschiedene Verbesserungen daran vorgenommen wurden.

$$B = \frac{t_A \text{ vorher}}{t_A \text{ nachher}}$$

t_A nachher: Ausführungszeit nach der Verbesserung

t_A vorher: Ausführungszeit vor der Verbesserung

- **Beispiel:** Wenn ein Programm, das vor der architektonischen Verbesserung 25 s für die Ausführung braucht, nachher nur noch 15 s läuft, beträgt die Verbesserung 1,67.

Amdahls Gesetz

- Beim Entwurf leistungsfähiger Rechnersysteme gilt die Regel: **Je häufiger ein Verarbeitungsvorgang auftritt, umso performanter muss man ihn auslegen.**
- Qualitativ bedeutet dies, dass der Einfluss einer bestimmten Leistungsverbesserung auf die Gesamtleistung sowohl davon abhängt, wie stark die bestimmte Leistung steigt, als auch davon, wie häufig sie eingesetzt wird. Dies drückt das **Amdahlsche Gesetz** aus.

$$t_{A \text{ neu}} = t_{A \text{ alt}} \left[T_{\text{Ant unbenutzt}} + \frac{T_{\text{Ant benutzt}}}{B_{\text{benutzt}}} \right]$$

$t_{A \text{ neu}}$: Neue Ausführungszeit

$t_{A \text{ alt}}$: Alte Ausführungszeit

$T_{\text{Ant unbenutzt}}$: Zeitanteil, in der die Verbesserung nicht benutzt wird

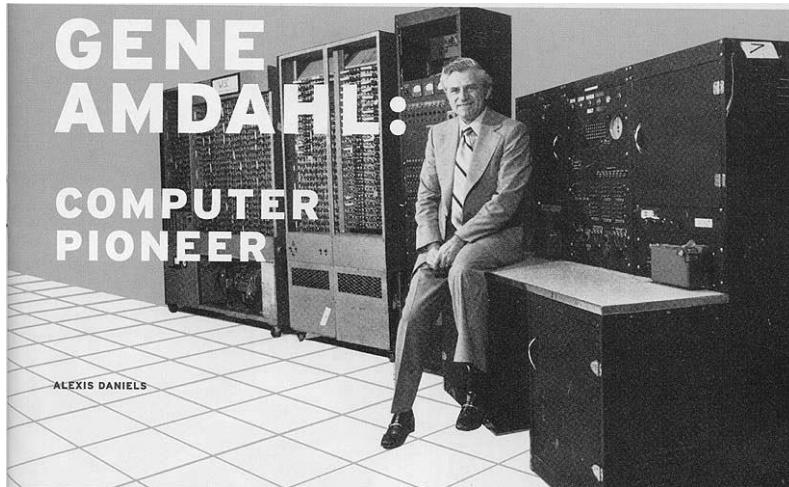
$T_{\text{Ant benutzt}}$: Zeitanteil, in der die Verbesserung benutzt wird

B_{benutzt} : Beschleunigung für den Anteil der Leistungsverbesserung

Amdahls Gesetz

- Amdahls Gesetz lässt sich mit Hilfe der Definition der Beschleunigung auch folgendermaßen umformulieren:

$$B = \frac{t_A \text{ vorher}}{t_A \text{ nachher}} = \frac{1}{T_{\text{Ant unbenutzt}} + \frac{T_{\text{Ant benutzt}}}{B_{\text{benutzt}}}}$$

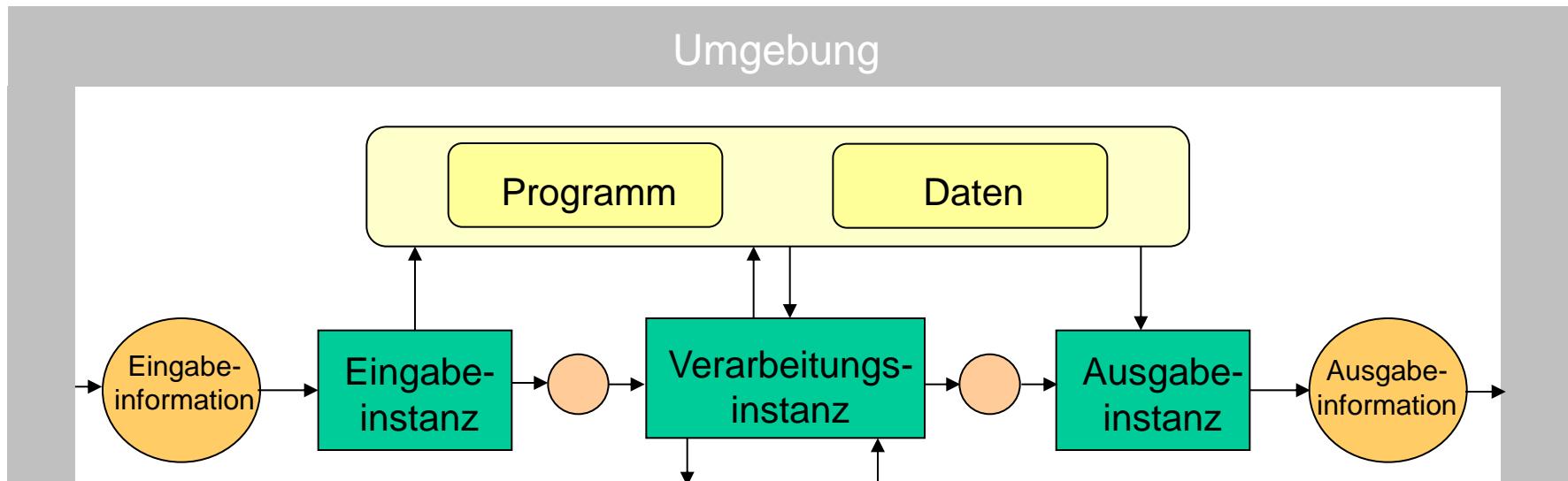


Von Neumann - Maschine

Von Neumann - Architektur

- Das Programm besteht aus einer sequentiellen Folge von Befehlen
 - Befehle beschreiben einen sequenziellen Prozess (Ablauf)
 - Befehle sind von der Maschine in Aufschreibungsreihenfolge auszuführen
 - Abweichen von dieser Reihenfolge durch Sprünge möglich
 - Zu jedem Zeitpunkt führt der Prozessor nur jeweils einen einzigen Befehl aus
 - Ein Befehl bearbeitet (höchstens) einen Datenwert
 - Operanden und Ergebnis werden durch Speicheradressen repräsentiert

Die von Neumann-Maschine



Komponenten eines von Neumann - Rechners

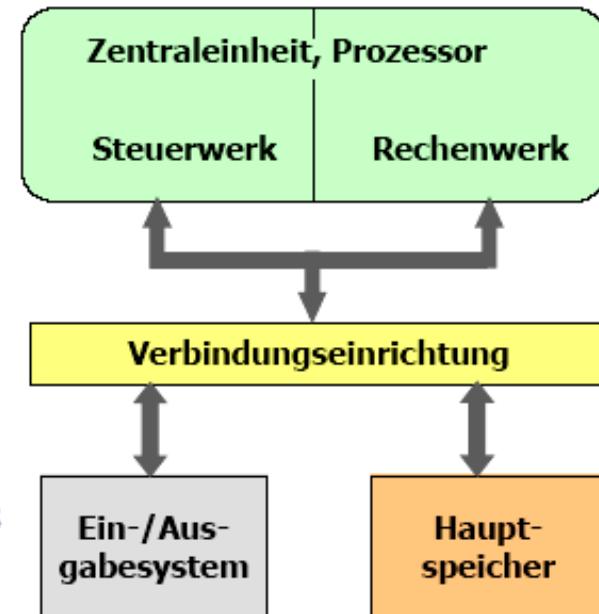
□ Zentraleinheit

(central processing unit,
CPU, Prozessor)

Verarbeitet Daten gemäß eines
Programms. Sie besteht aus
Leitwerk und Rechenwerk:

➤ **Leitwerk** (Steuerwerk,
control unit, CU)

- Holt die Befehle eines Programms
aus dem Speicher
- entschlüsselt sie und
- steuert ihre Ausführung in der
verlangten Reihenfolge durch
Steuer- und Synchronisier-Signale.

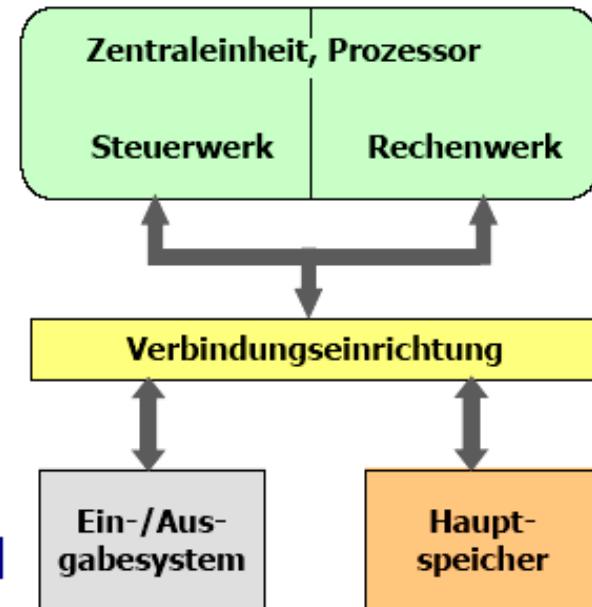


Komponenten eines von Neumann - Rechners

□ Zentraleinheit

Verarbeitet Daten gemäß eines Programms. Sie besteht aus Leitwerk und Rechenwerk:

- **Rechenwerk (Operationswerk, Ausführungseinheit, ALU)**
 - Führt arithmetisch/logische Operationen aus.
 - Wird durch Steuersignale des Leitwerks beeinflußt und liefert seinerseits Meldesignale an das Leitwerk zurück.



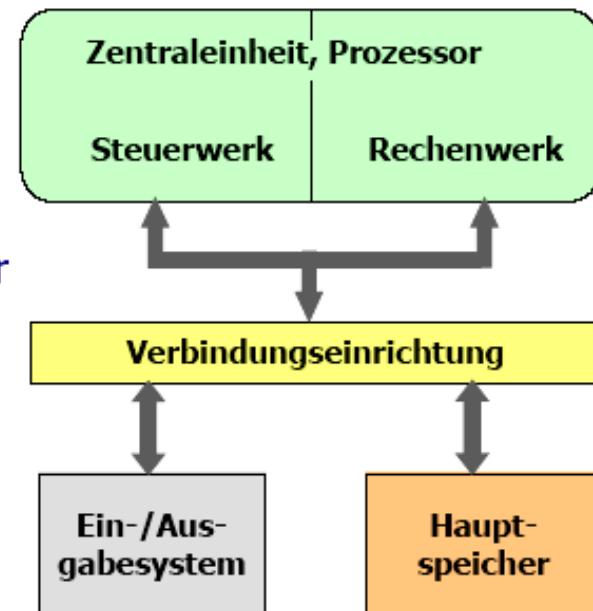
Komponenten eines von Neumann - Rechners

□ Verbindungsstruktur (BUS)

- **Adreßleitungen:** Leitungen, auf denen die Adressinformation transportiert wird (unidirektional).
- **Datenleitungen:** Transportieren Daten und Befehle von/zum Prozessor (bidirektional).
- **Steuerleitungen:** Geben Steuerinformationen von/zum Prozessor (uni- oder bidirektional).

□ Bus (Sammelschiene)

Systembus = Adreßbus + Datenbus + Steuerbus



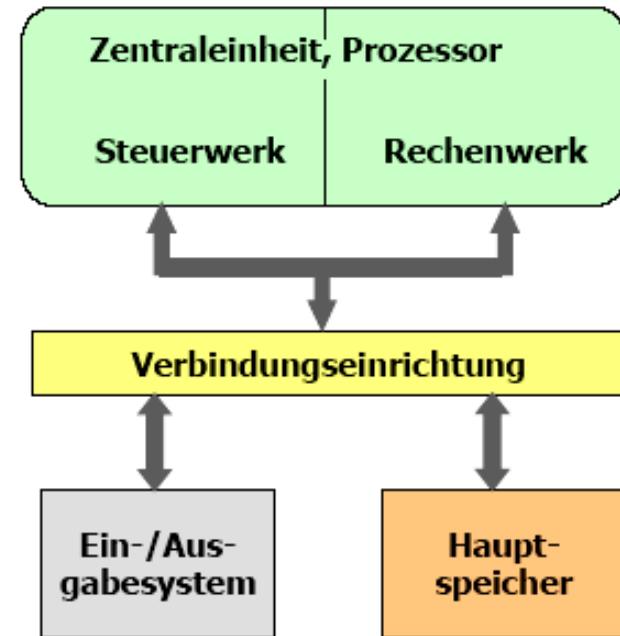
Verbindungseinrichtung: Bus

- Ein **Bus** ist ein gemeinsamer Datenpfad (Leitungsbündel), der mehrere (≥ 2) Einheiten eines Rechners verbindet.
- Die Einheiten heißen **Busteilnehmer**
 - Aktive Busteilnehmer (**Bus Master**) können Datentransfers einleiten.
 - Passive Busteilnehmer (**Slaves**) reagieren nur auf Anfragen.
- Ein Bus ist ein exklusives Betriebsmittel
 - Zu jedem Zeitpunkt kann nur ein Busteilnehmer den Bus aktiv steuern.
- Ein Bus ist ein Broadcast-Medium
 - Daten auf dem Bus können von allen Busteilnehmern gleichzeitig gelesen werden.

Komponenten eines von Neumann - Rechners

□ Ein-/Ausgabesystem (Peripheriegeräte)

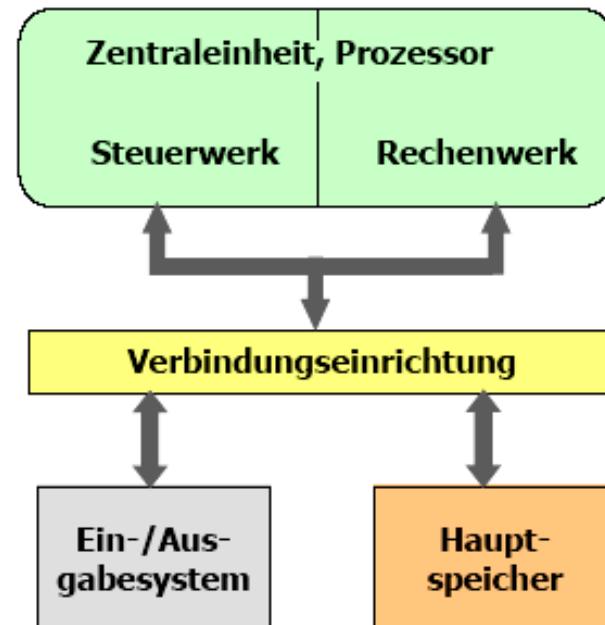
- Geräte zur Eingabe von Daten und Programmen und zur Ausgabe der verarbeiteten Daten (Bildschirme, Drucker, Terminals, ...)
- Diese Geräte sind über Ein-/Ausgabe-Schnittstellen mit dem Rechner verbunden.
- Die Verbindung der Schnittstellen mit dem Prozessor (und zu den Peripheriegeräten) geschieht durch Adreß-, Daten- und Steuerleitungen.



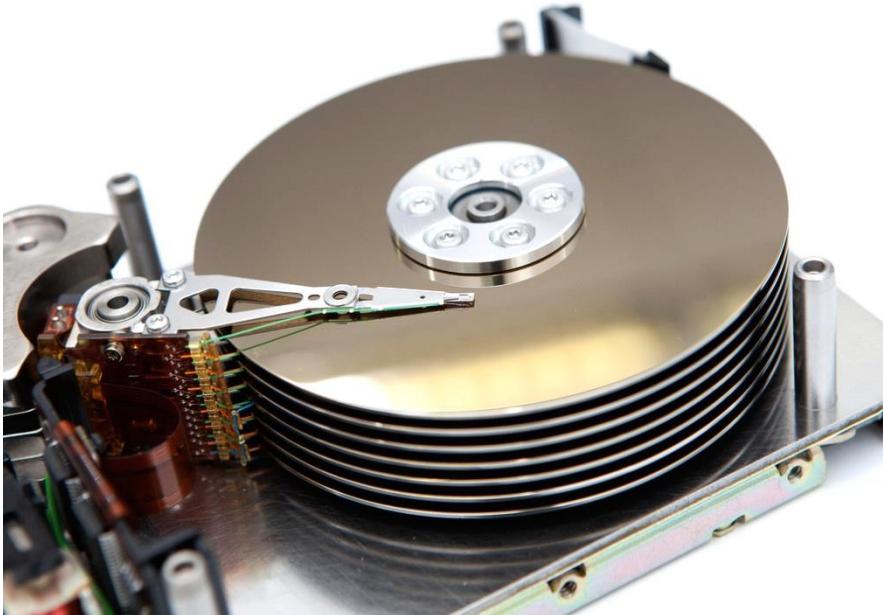
Komponenten eines von Neumann - Rechners

□ Hauptspeicher

- Jede Speicherzelle ist eindeutig durch ihre Nummer (Adresse) identifizierbar.
- Dort werden Programme und Daten aufbewahrt (von-Neumann-Konzept).
- Den einzelnen Speicherzellen ist nicht anzusehen, welchen Typ von Information sie enthält
- Alternativ: **Harvard-Architektur** mit getrenntem Programm- und Datenspeicher.
- Inhalt nach Abschaltung des Rechners flüchtig.



Magnetfestplatte und Solid State Disk (SSD)



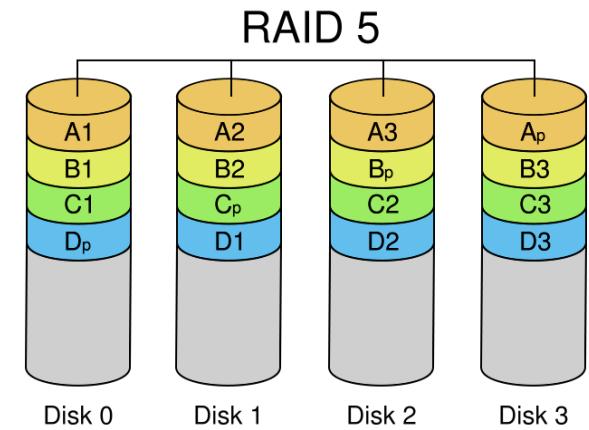
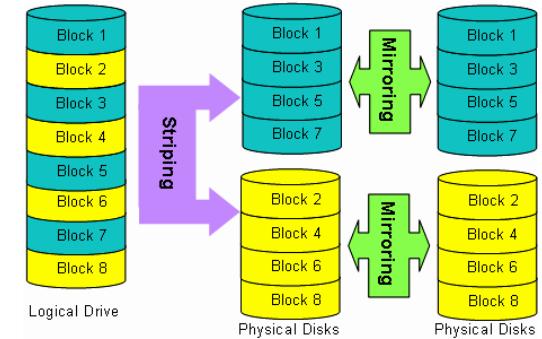
- Magnetische Speicherung von Daten (nicht flüchtig)
- Schreib-/Lese-Kopf „fliegt über die Platte“
- Drehzahlen 5.400 U/min oder 7.200 U/min



- Nicht-flüchtiger Speicher
- Wahlfreier Zugriff
- Schnelle Zugriffszeiten
- Kleine Speichermenge
- Teuer

Redundant Array of Independent Disks (RAID)

- RAID 0+1
 - Gesamter Datenbestand wird auf zwei unabhängigen Festplatten gespiegelt
 - mind. zwei Festplatten notwendig
- RAID 5
 - Gesamter Datenbestand wird ergänzt über Prüfsummen gespeichert
 - mind. drei Festplatten notwendig

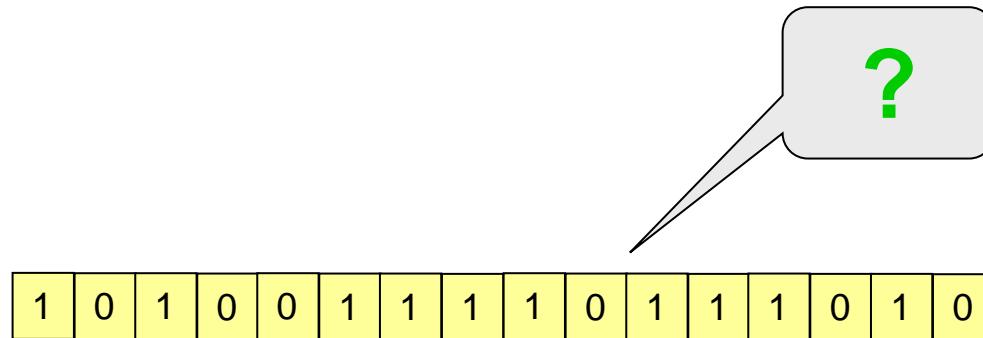


Charakteristika der von Neumann-Maschine

- **Informationsstruktur:**

Beschreibt die Typen der vorkommenden Informationskomponenten

- Mit welchen Objekten arbeitet die Maschine ?
 - Bitketten (Binärvokabeln)
- Wie ist ihre Interpretation (Semantik) ?
 - Befehle
 - Daten
 - Adressen



Maschinen-Datentypen

- Wie erfolgt die Darstellung der Informationskomponenten in der Maschine ?
 - Durch Maschinen-Datentypen
 - Bit (Zustandsgröße, 1 Bit)
 - BCD (BCD-Zahlen, 4 Bit)
 - Byte (Zahl, ASCII-Zeichen, 8 Bit)
 - Wort (Zahl, Adresse, Befehl, 16 Bit)
 - Langwort (Zahl, Adresse, Befehl, 32 Bit)
- Alle Informationskomponenten werden als Maschinen-Datentypen durch Binärvokabeln dargestellt. Maschinen-Datentypen können durch die Maschine direkt verarbeitet werden. Sie sind nicht typbehaftet und nicht strukturiert.

Interpretation

- Interpretation der Informationskomponenten
 - Wie weiß die Maschine, wie sie eine gegebene Bitkette zu interpretieren hat ?
→ Innerer Zustand der Maschine zum Zeitpunkt der Interpretation
- Interpretation setzt bestimmte Darstellungsformen voraus
 - Lineare Teilordnung über der Menge der Befehle
 - Darstellungsform für Anweisungen
 - Maschinenbefehl
 - Operator
 - Operand oder
 - Operandenreferenz
 - » explizit
 - » implizit

Speicherstruktur

- Code und Daten stehen ohne Unterscheidung im selben Speicher. Es existieren keine speichertechnischen Maßnahmen, um ungerechtfertigten Zugriff zu verhindern.

→ Wie könnte denn so ein Schutz aussehen?

- Die Interpretation eines Speicherinhalts hängt nur vom aktuellen Kontext ab.

→ Was müsste man tun, damit man einer Speicherzelle ansieht, was sie enthält?

Speicherstruktur

- Die Menge der Befehle eines Programms ist partiell geordnet.
Auf der Menge der Daten eines Programms gibt es keine für die Maschine erkennbare Strukturierung (Operanden werden durch Zeiger referenziert).

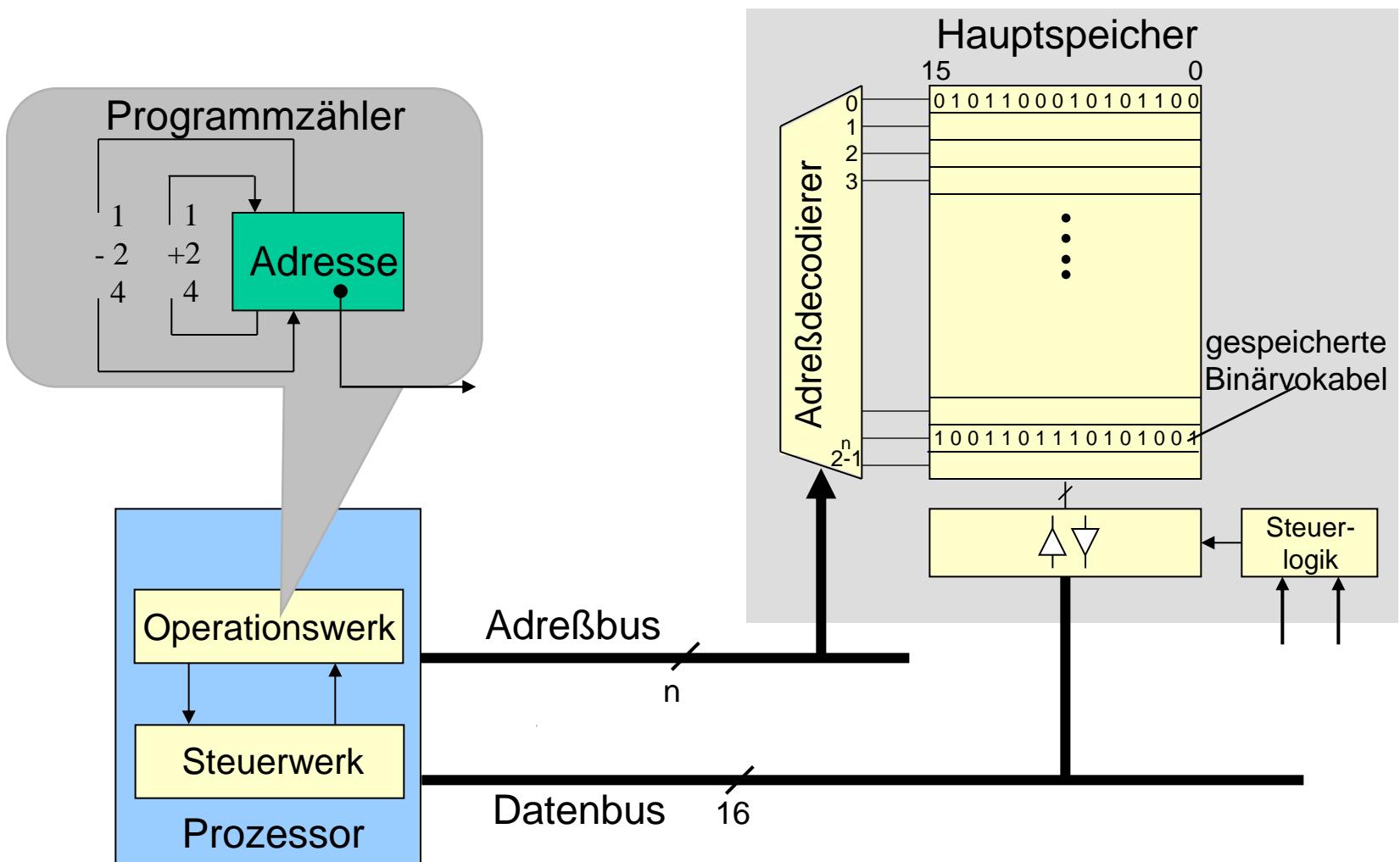
→ Welche Alternative gäbe es denn zur linearen Ordnung?
- Die lineare Anweisungsstruktur erlaubt auf der Hardwareebene der Prozessorrealisierung die Anwendung des Befehlszählern-Prinzips.

→ Prinzip des minimalen Hardwareaufwands

Charakteristika der von Neumann-Maschine

- Zwei-Phasen-Konzept der Befehlsverarbeitung
 - In der Interpretationsphase wird aufgrund der durch den Befehlszähler angezeigten Adresse der Inhalt einer Speicherzelle geholt und als Befehl interpretiert.
 - In der Ausführungsphase wird aufgrund der im Befehl enthaltenen Adresse der Inhalt einer weiteren Speicherzelle geholt und als Datenwert verarbeitet.
- Der Befehlsablauf folgt einer sequentiellen Befehlsfolge und ist selbst streng sequentiell.
- Zu jedem Zeitpunkt führt der Prozessor nur einen einzigen Befehl aus und dieser kann höchstens einen Datenwert bearbeiten (SISD).

Programmzählerprinzip und Adressierung



Nachteile der von-Neumann-Architektur

Die Verbindungseinrichtung (Hauptspeicher \leftrightarrow CPU) stellt einen Engpass dar: „von-Neumann-Flaschenhals“

Die Programmierung muss den sequentiellen Verkehr durch den von-Neumann-Flaschenhals planen

→ Auswirkungen auf höhere Programmiersprachen
("intellektueller Flaschenhals")

Geringe Strukturierung der Daten

Maschinenbefehl bestimmt Operandentyp

→ semantische Lücke

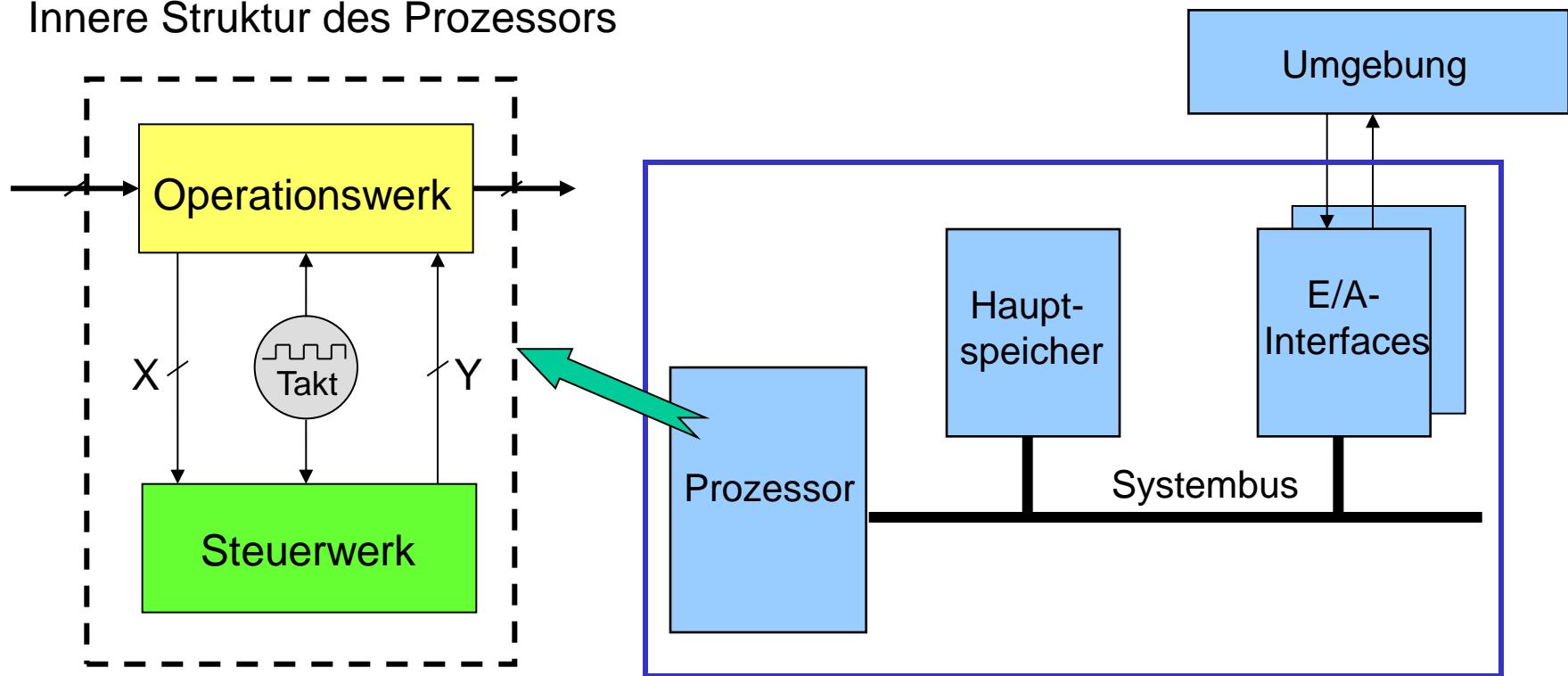


Heute: Änderungen am Operationsprinzip

- Interne Befehlsausführung nicht mehr streng sequentiell
 - Erhöhung der Performance durch Pipelining und Superskalarprinzip
 - Aber: Nach außen hin „Illusion der streng sequentiellen Abarbeitung“
 - Verwendung mehrerer Prozessorkerne
- Problem von-Neumann-Flaschenhals
 - Einführung einer Speicherhierarchie, effiziente Speicheranbindung
- Adressmodifikation
 - Programmadressen werden modifiziert
 - Durch das Programm selbst: in eine Prozessadresse (effektive Adresse, virtuelle Adresse)
 - Durch Maschine und Betriebssystem: in eine physische Adresse
- Mehrere Register im Rechenwerk statt nur ein Akkumulator
- Unterbrechungen der Programmablaufs möglich (Interrupts)

Detaillierung der Prozessorstruktur

Innere Struktur des Prozessors



Taktfrequenzen: CPU - und Bussystem und Busbreite

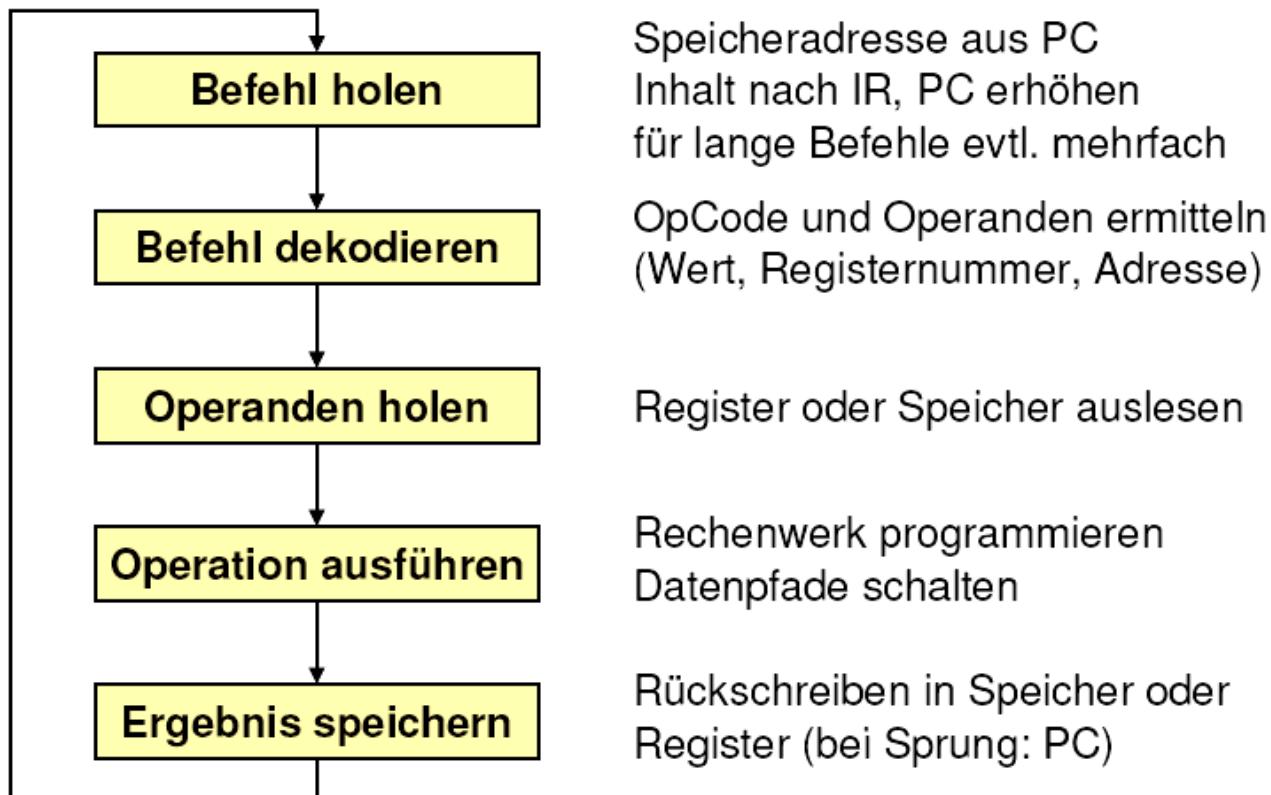
- CPU und Bussystem arbeiten mit unterschiedlichen **Taktfrequenzen**. Bus-Takt wird durch Frequenzteilung aus dem CPU-Takt abgeleitet.
- Beispiel:
 - $f_{\text{cpu}} = 800 \text{ MHz} \implies T_{\text{cpu}} = 1,25 \text{ ns}$
 - $f_{\text{Bus}} = 100 \text{ MHz} \implies T_{\text{Bus}} = 10 \text{ ns}$

Prozessortyp	Datenbreite (in Bit)	Adressbreite (in Bit)
8086	16	20
80286	16	24
80386, -486, Pentium	32	32
Itanium	64	64
MC68000	16	32
PowerPC	32	32

Befehlszyklus

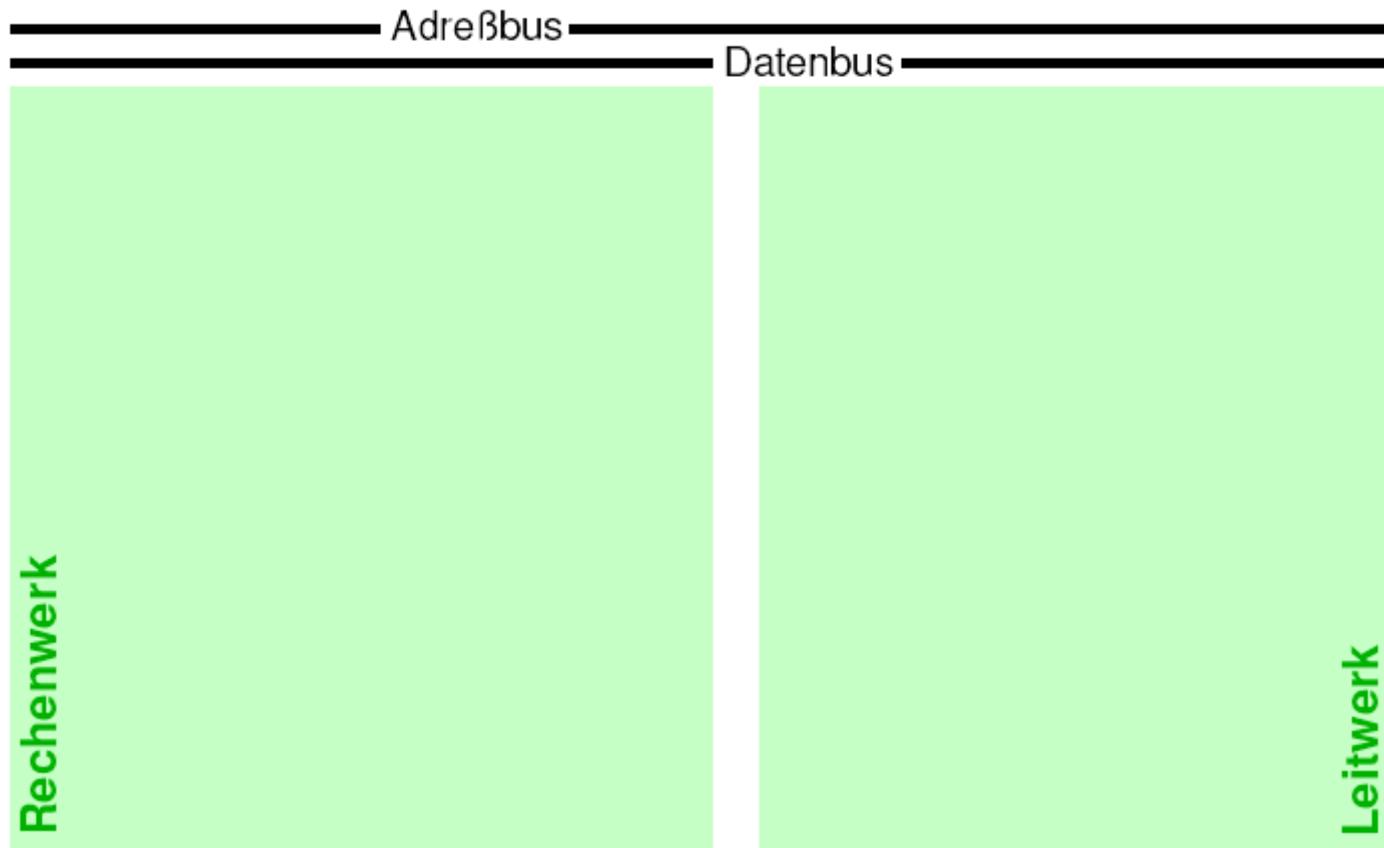
- Im Folgenden soll der systematische funktionale Aufbau einer einfachen CPU betrachtet werden. Basis ist dabei der Prozess der Befehlsbearbeitung.

Ein einfacher Befehlszyklus



Strukturelemente einer einfachen CPU

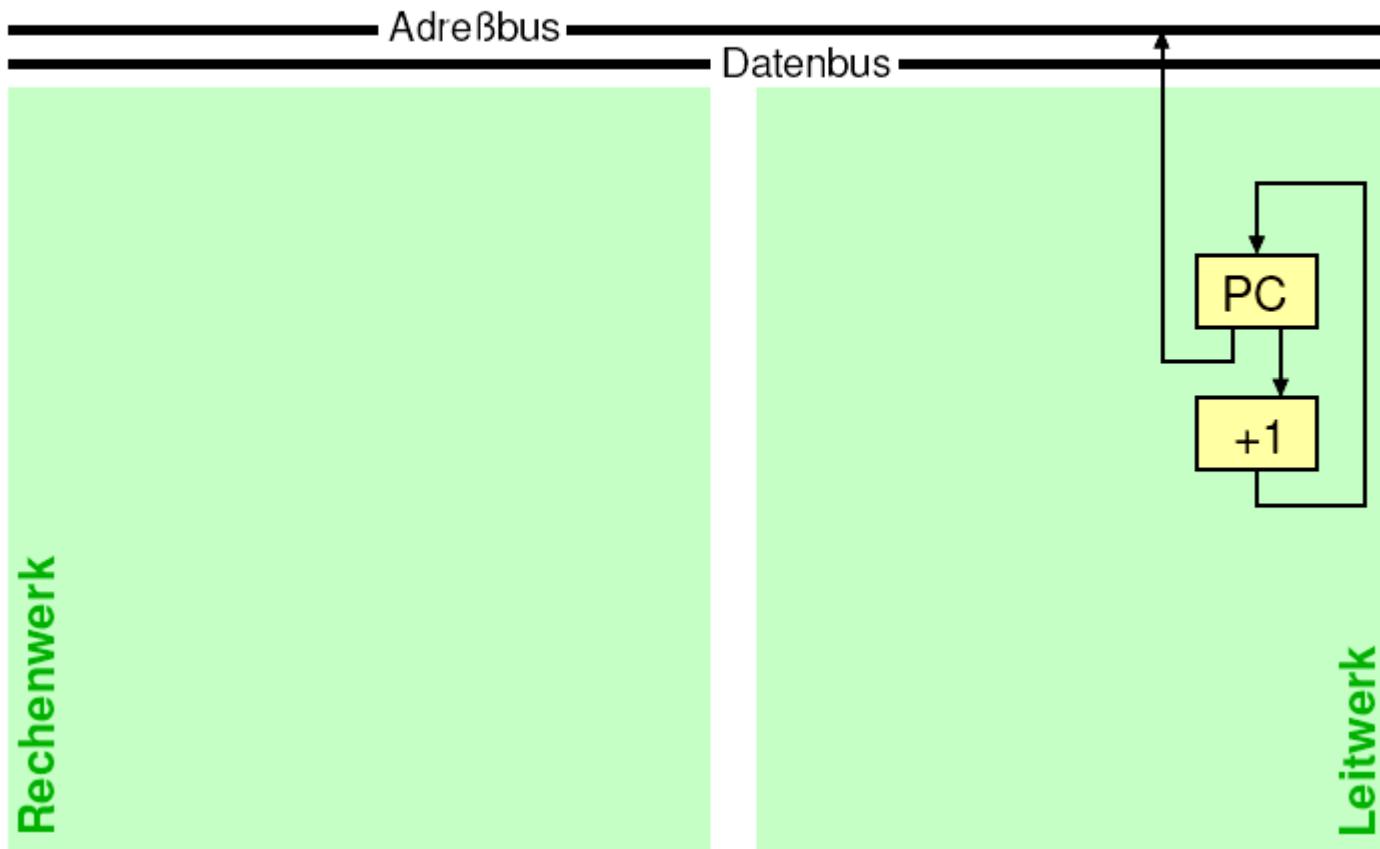
Aufbau einer einfachen CPU



Strukturelemente einer einfachen CPU

Aufbau einer einfachen CPU

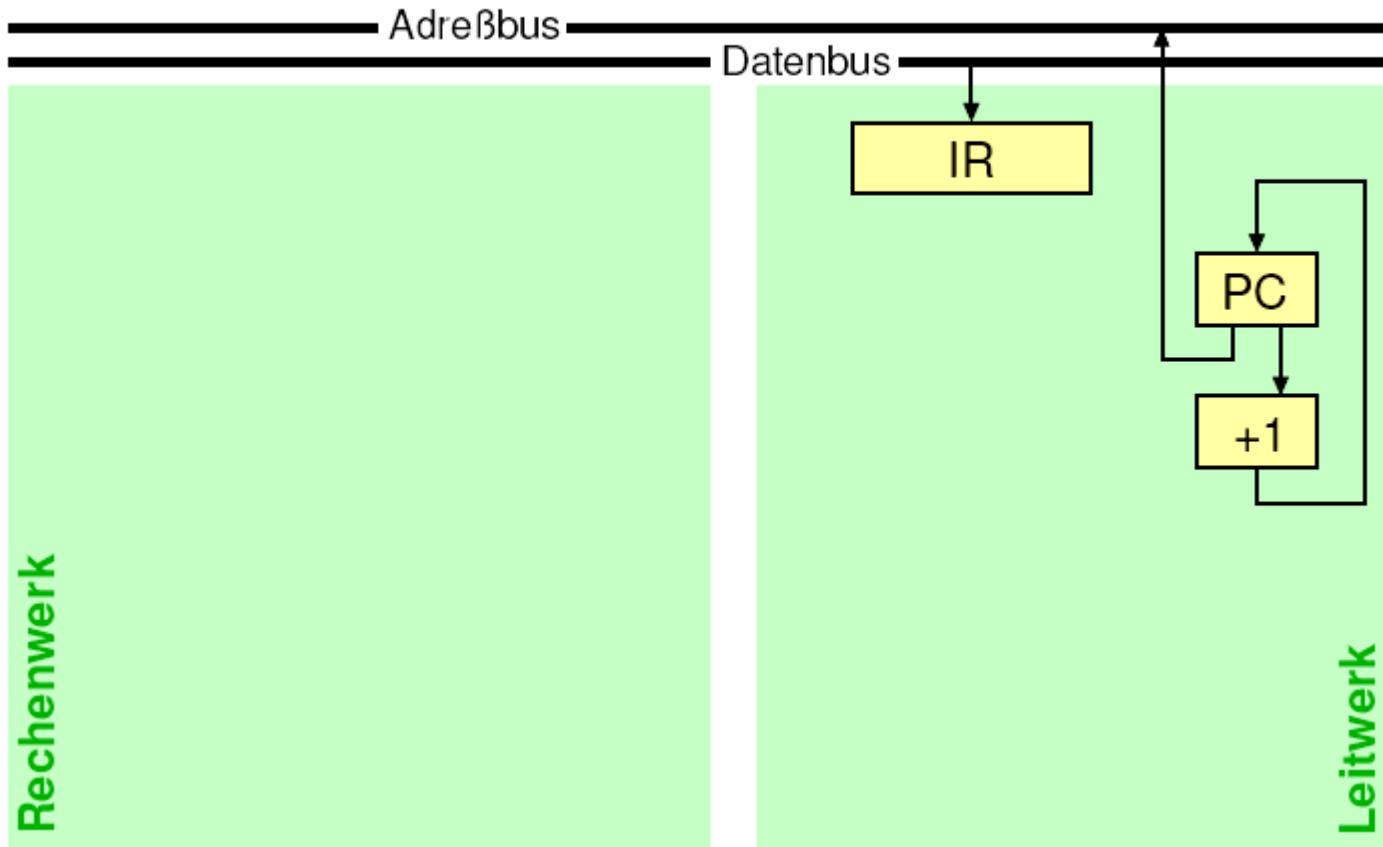
Befehlszähler



Strukturelemente einer einfachen CPU

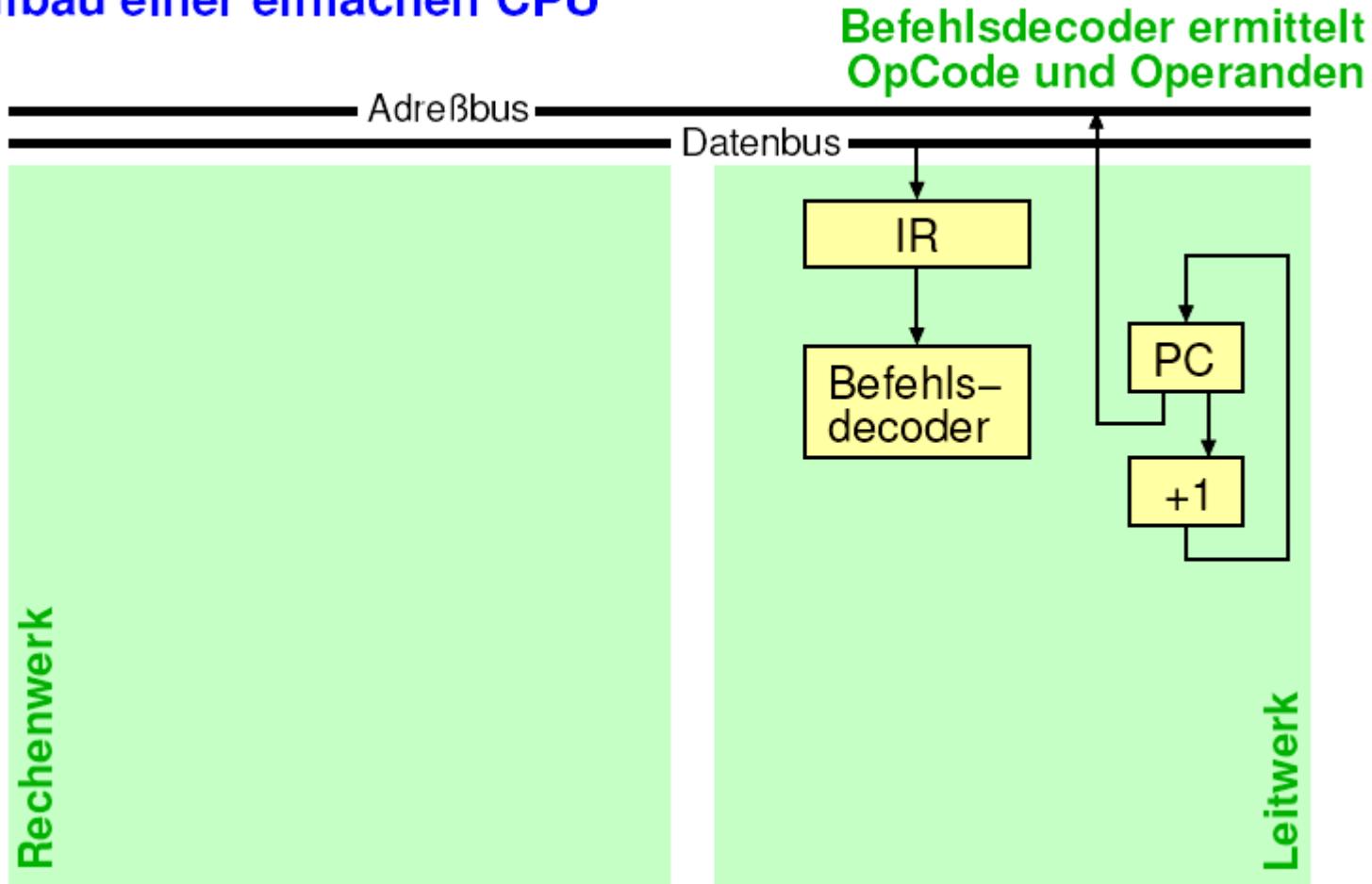
Aufbau einer einfachen CPU

Instruktionsregister



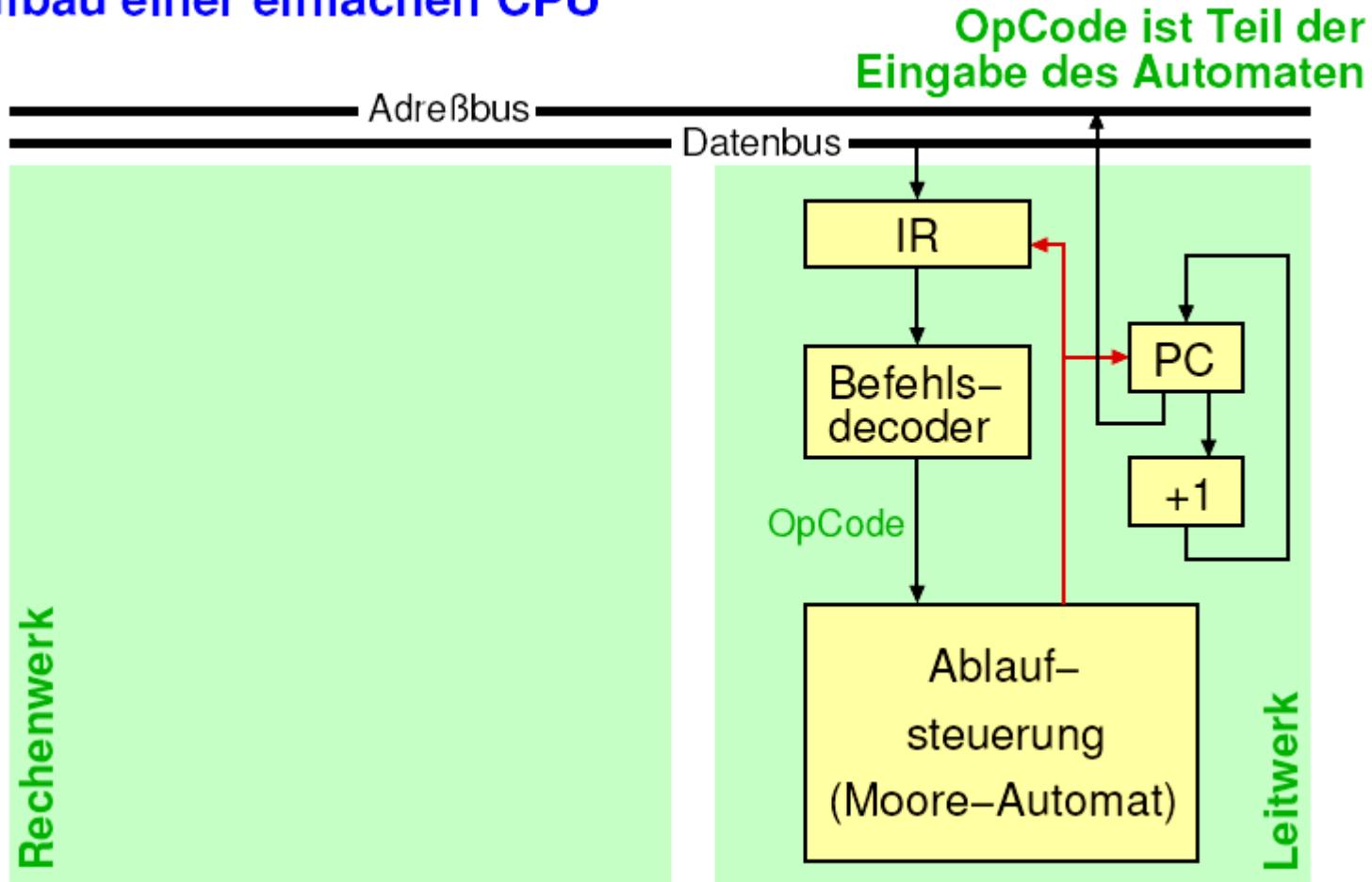
Strukturelemente einer einfachen CPU

Aufbau einer einfachen CPU



Strukturelemente einer einfachen CPU

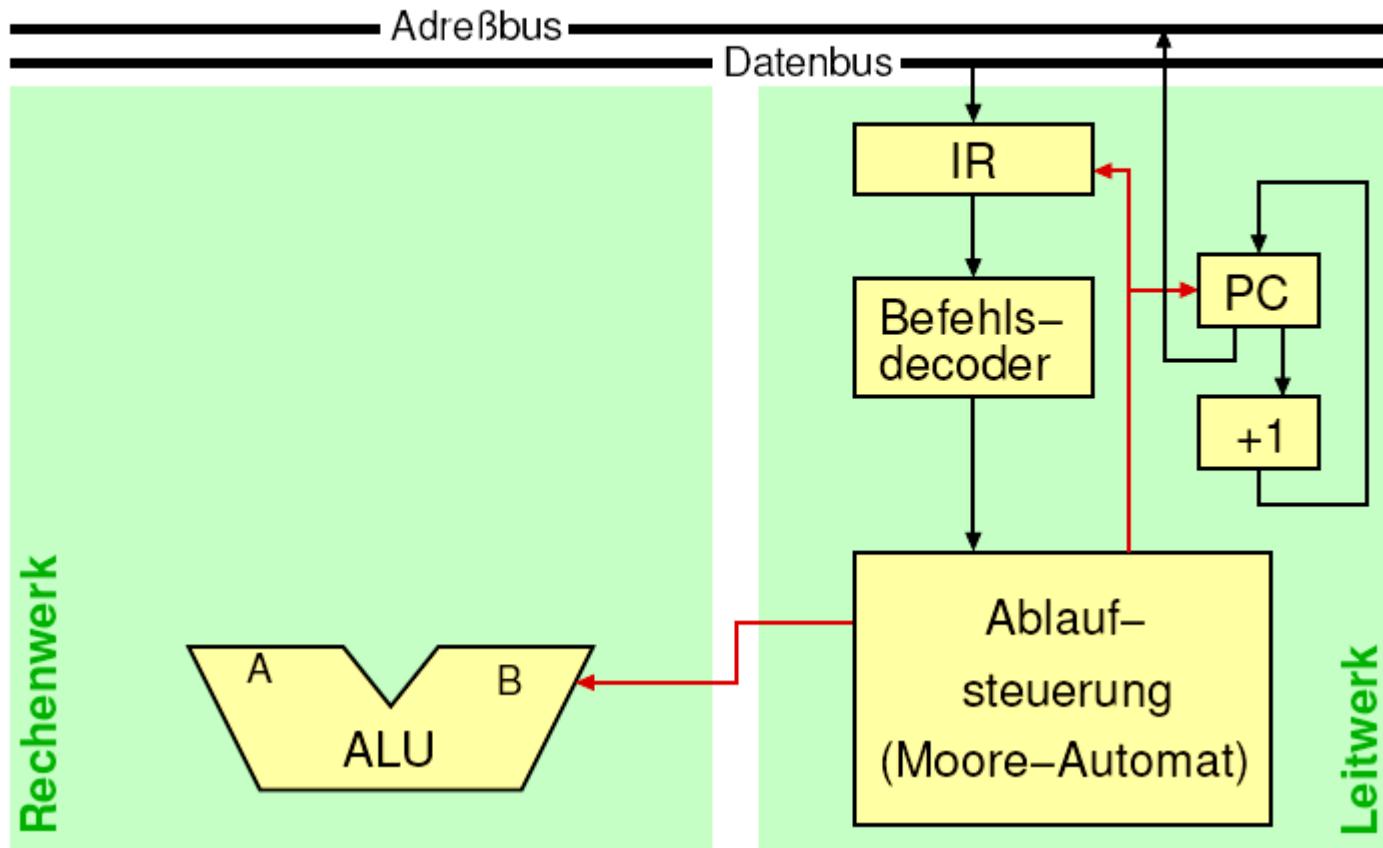
Aufbau einer einfachen CPU



Strukturelemente einer einfachen CPU

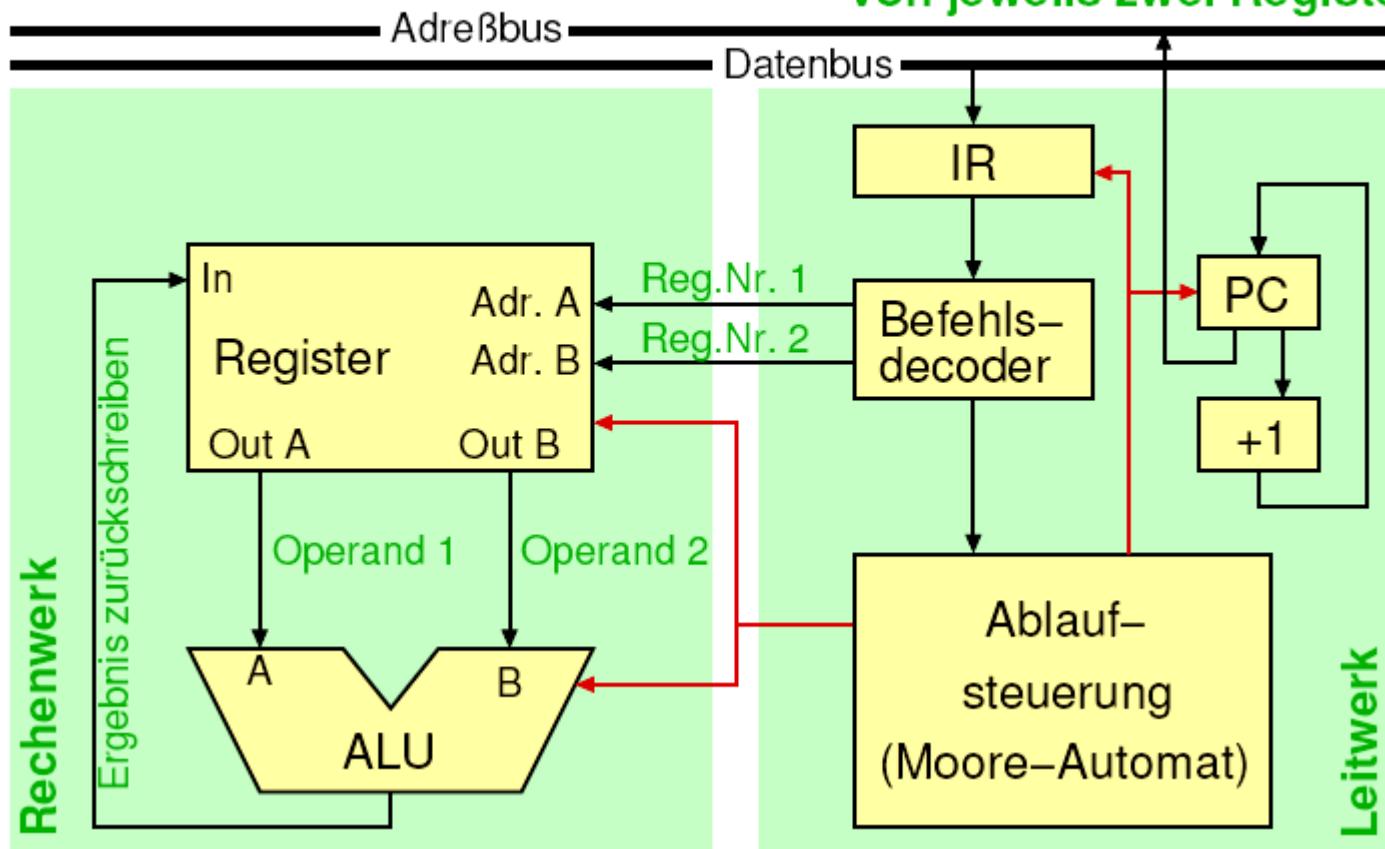
Aufbau einer einfachen CPU

Arithmetisch/logische Einheit



Strukturelemente einer einfachen CPU

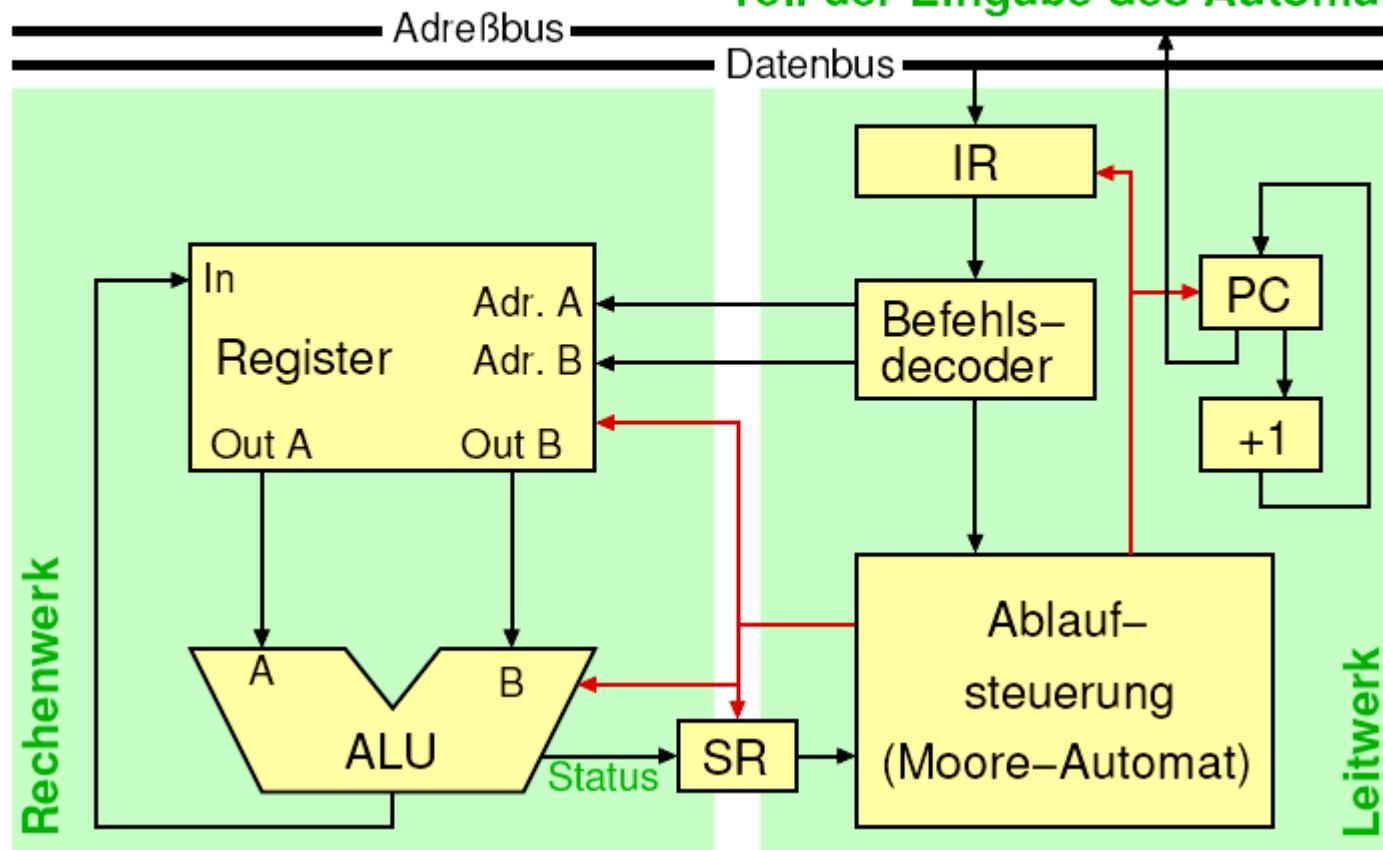
Aufbau einer einfachen CPU



Strukturelemente einer einfachen CPU

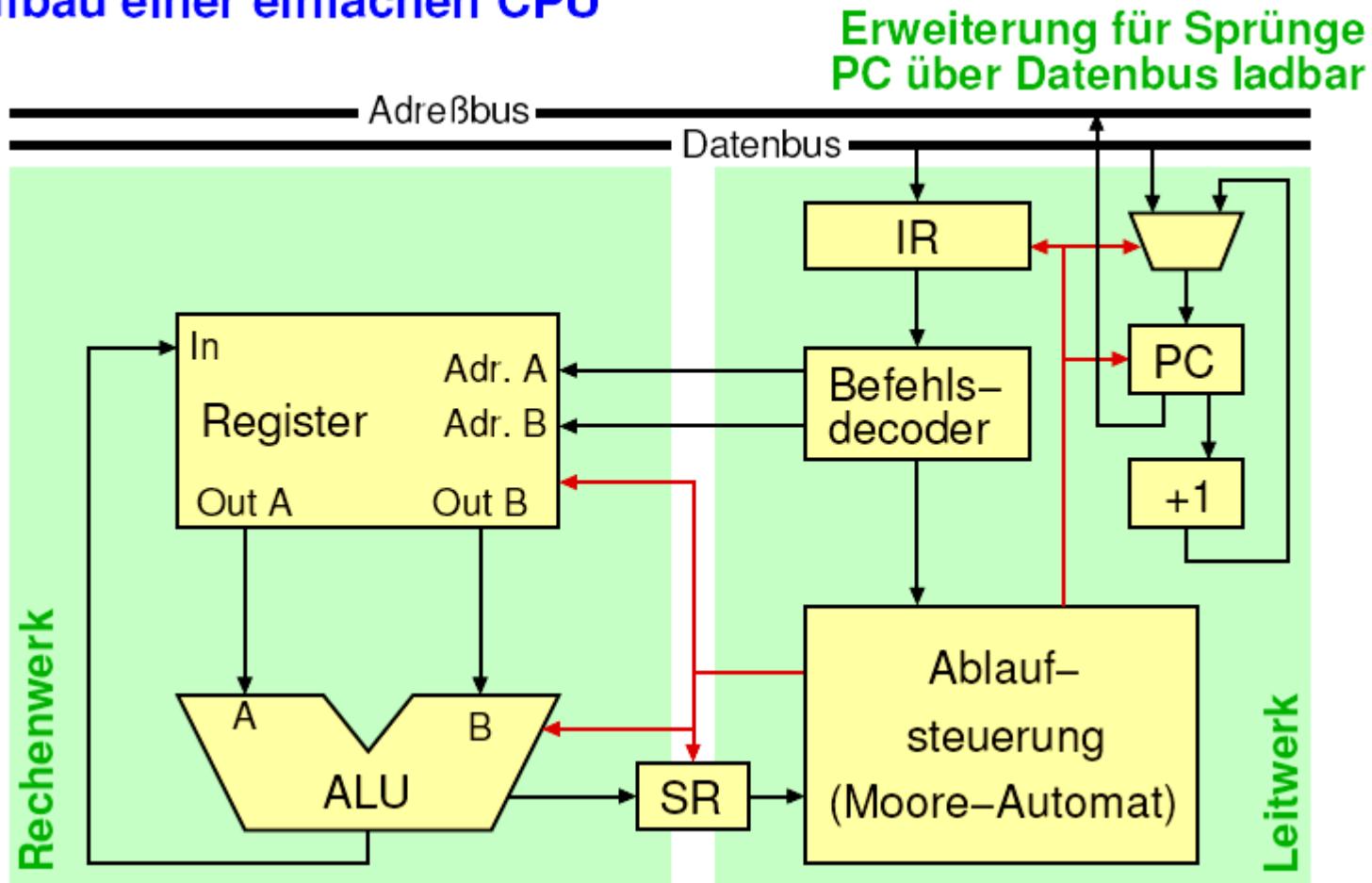
Aufbau einer einfachen CPU

Statusregister: Statussignale sind Teil der Eingabe des Automaten



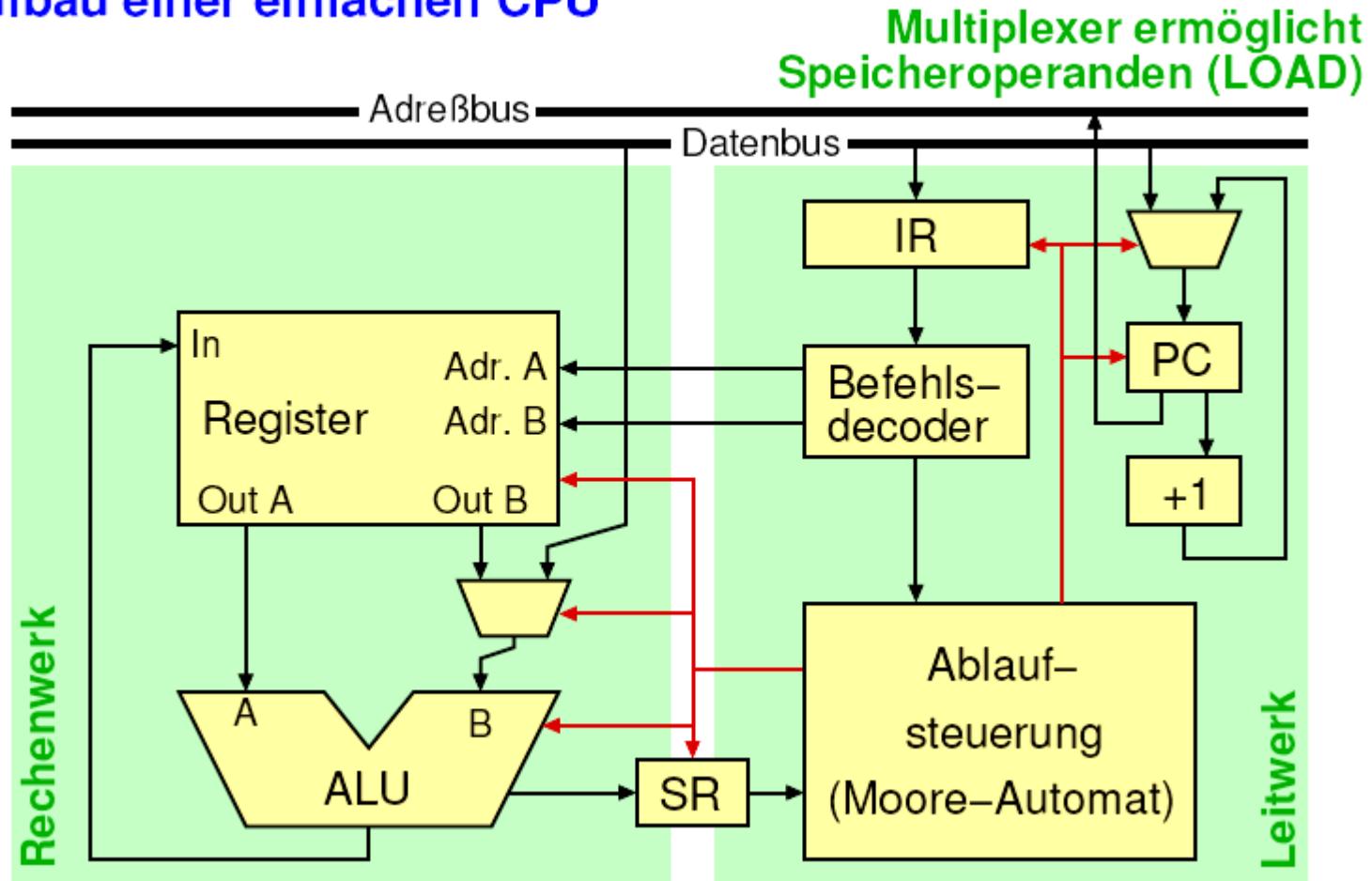
Strukturelemente einer einfachen CPU

Aufbau einer einfachen CPU



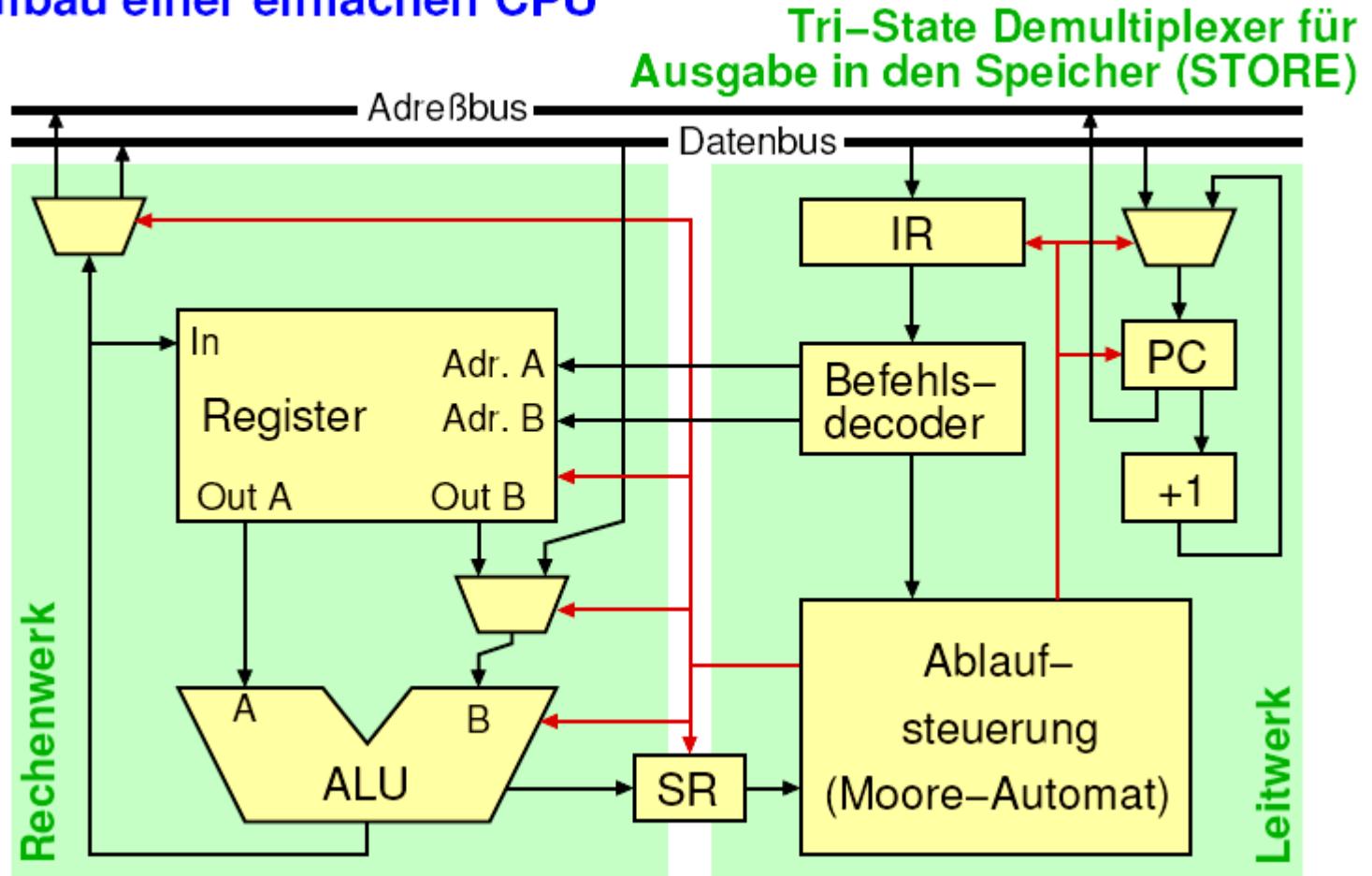
Strukturelemente einer einfachen CPU

Aufbau einer einfachen CPU



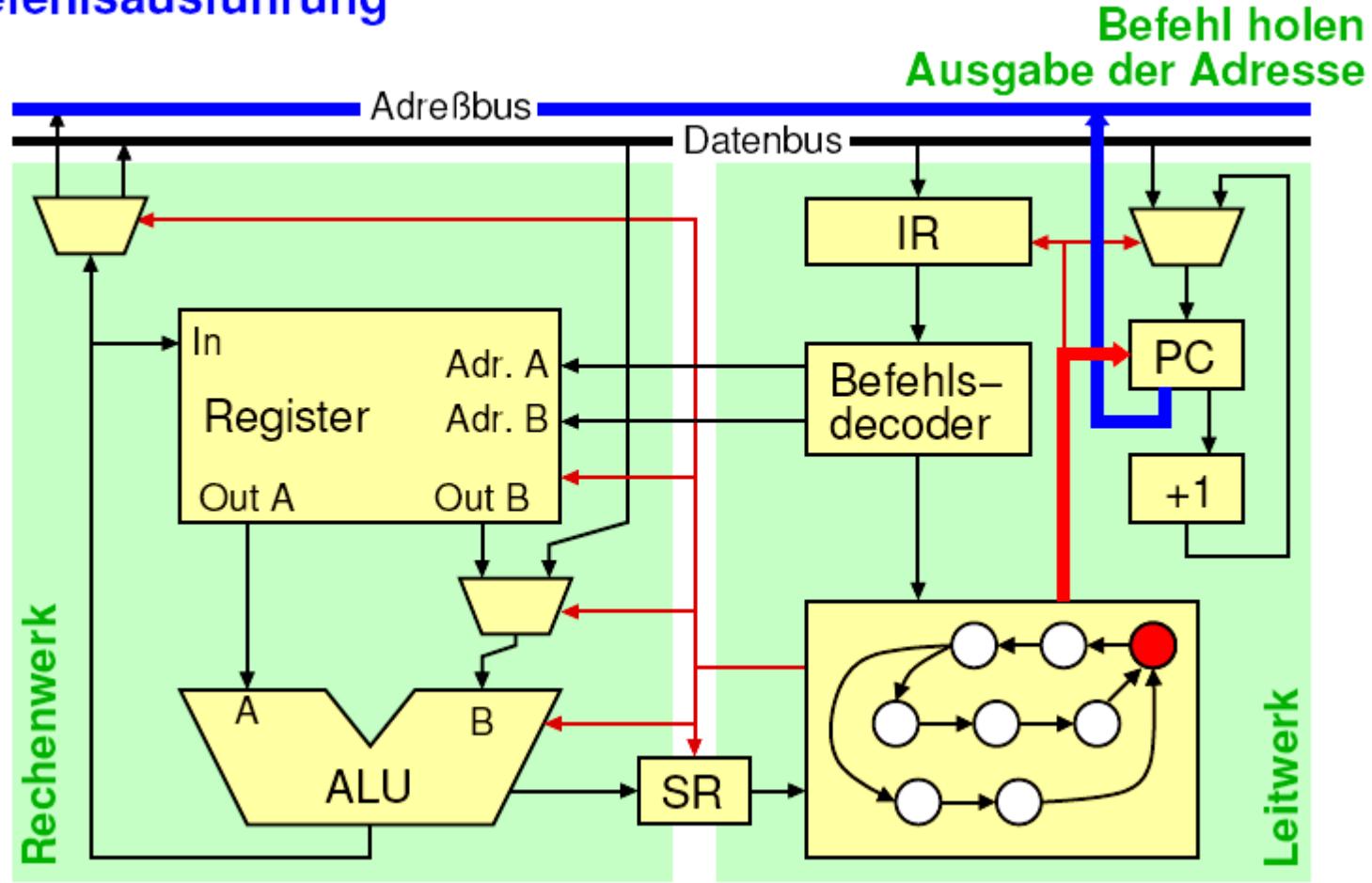
Strukturelemente einer einfachen CPU

Aufbau einer einfachen CPU



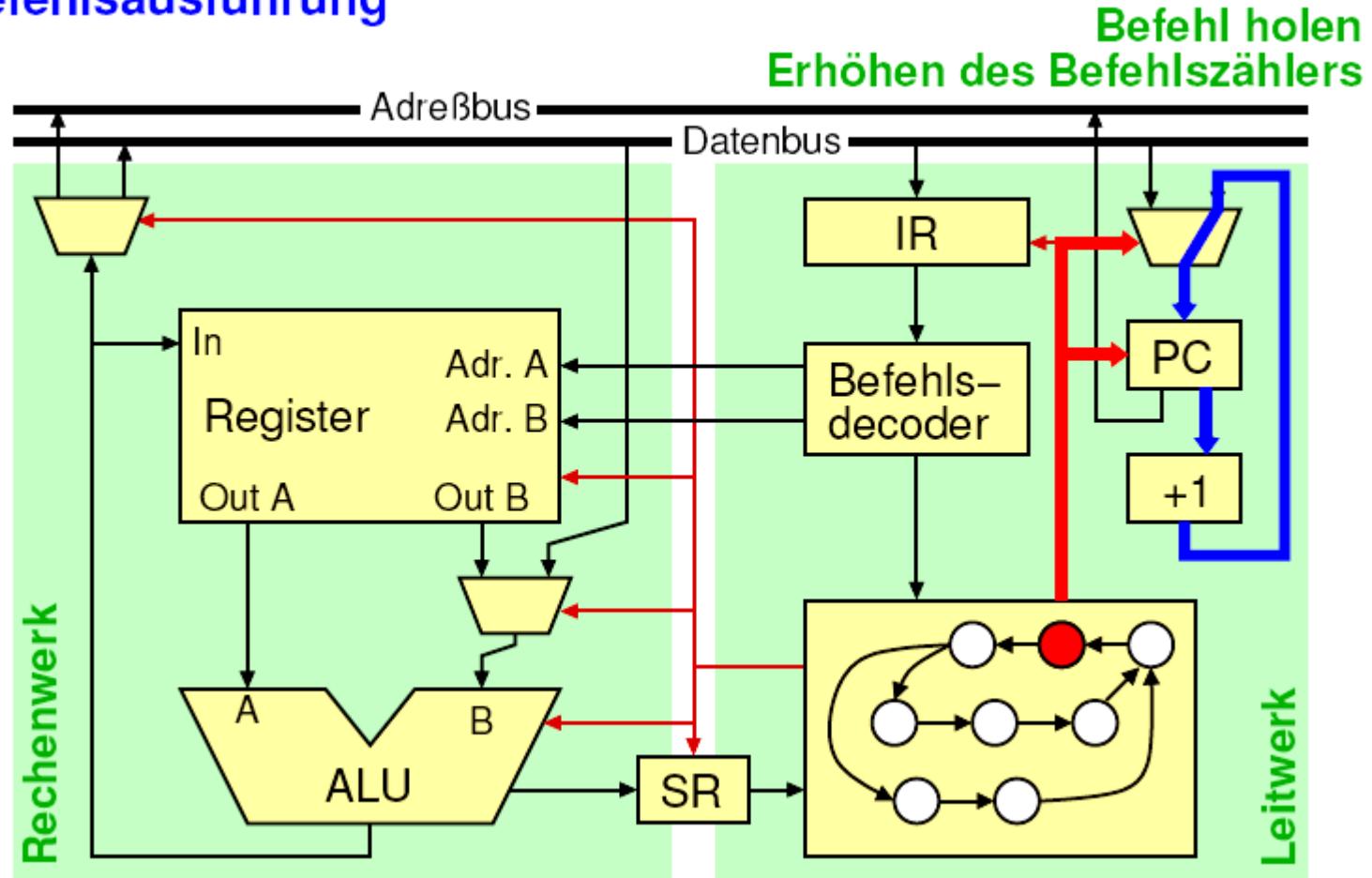
Prozess: Befehlsausführung in einer CPU

Befehlsausführung

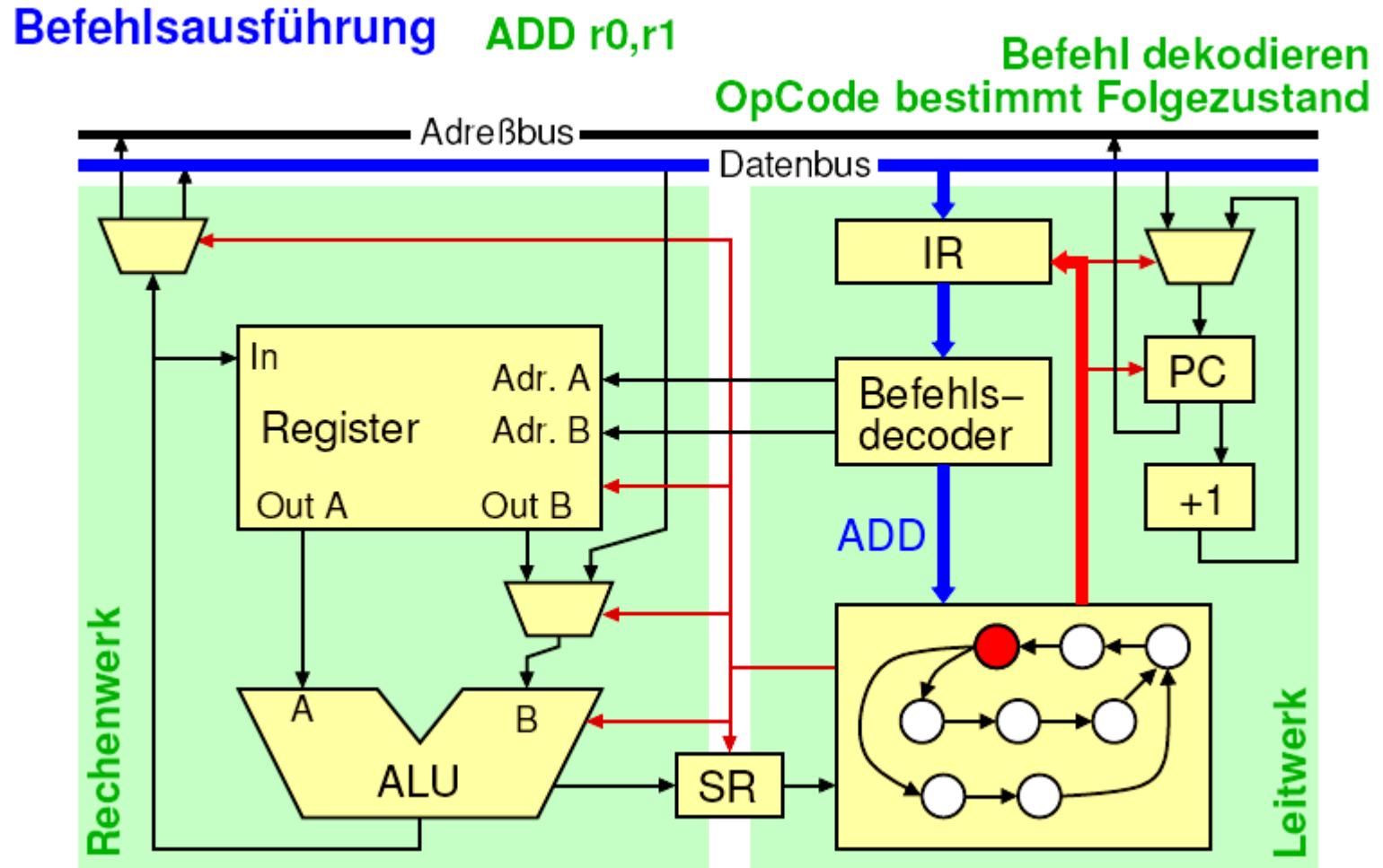


Prozess: Befehlsausführung in einer CPU

Befehlsausführung



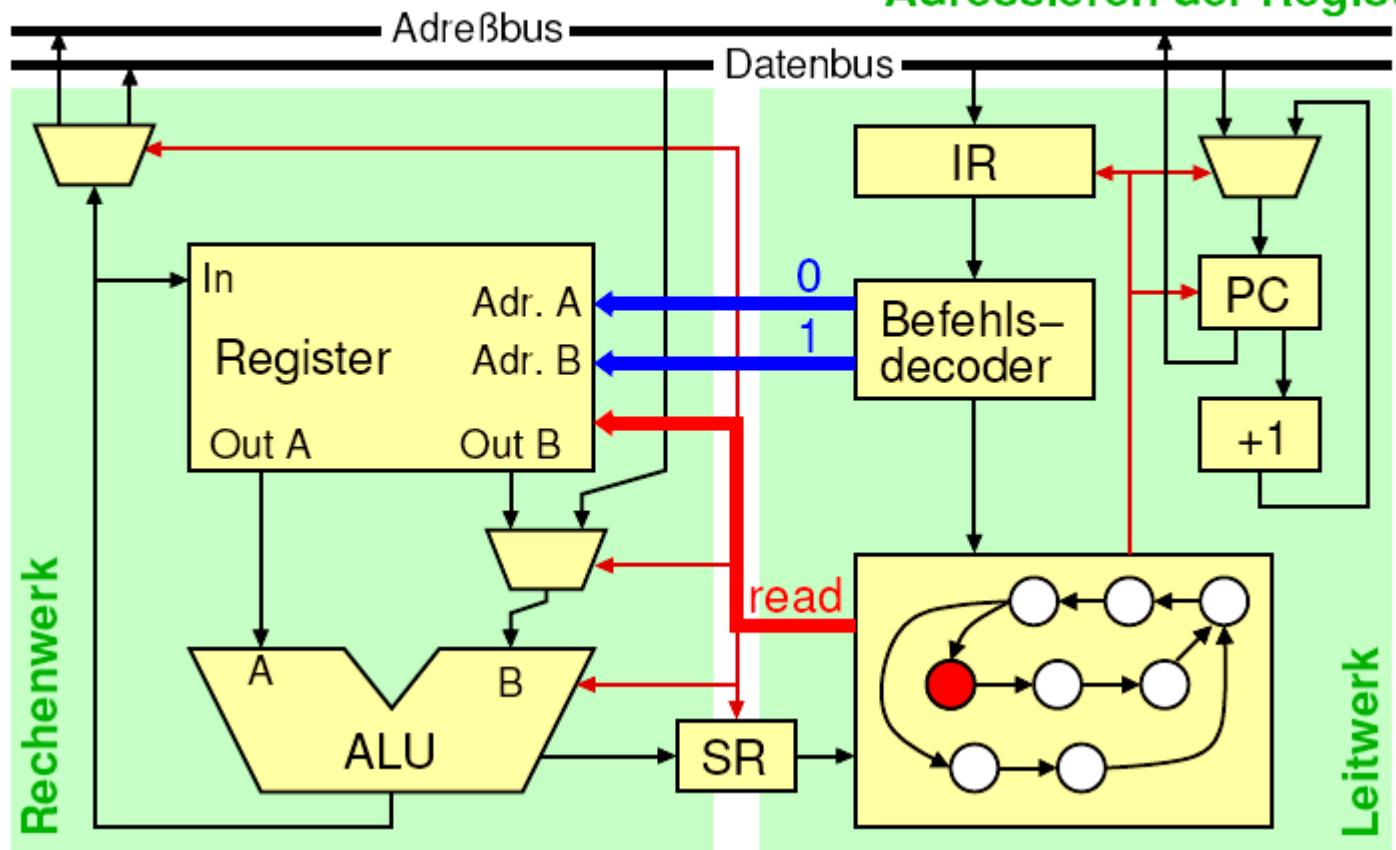
Prozess: Befehlsausführung in einer CPU



Prozess: Befehlsausführung in einer CPU

Befehlsausführung ADD r0,r1

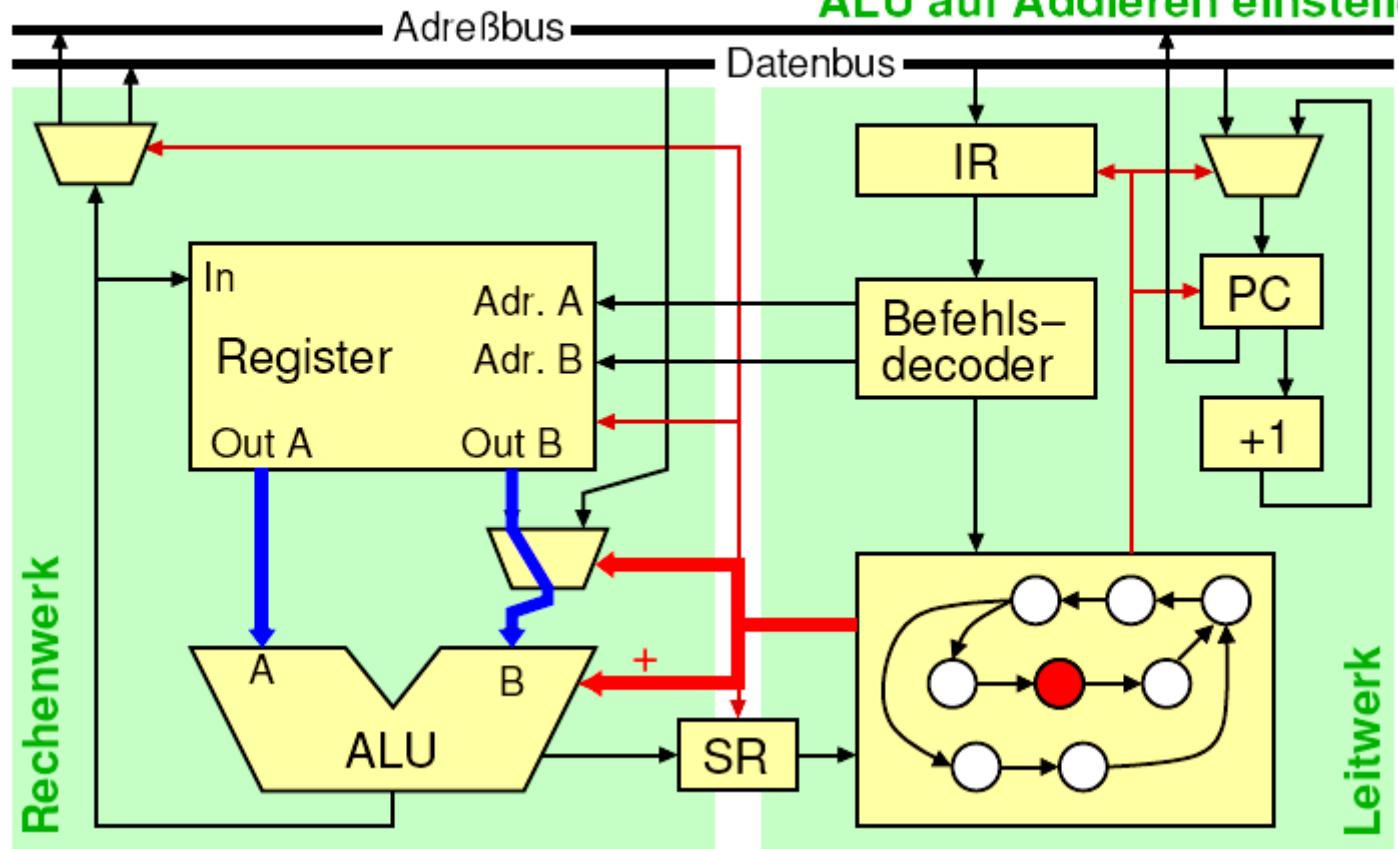
Operanden holen
Adressieren der Register



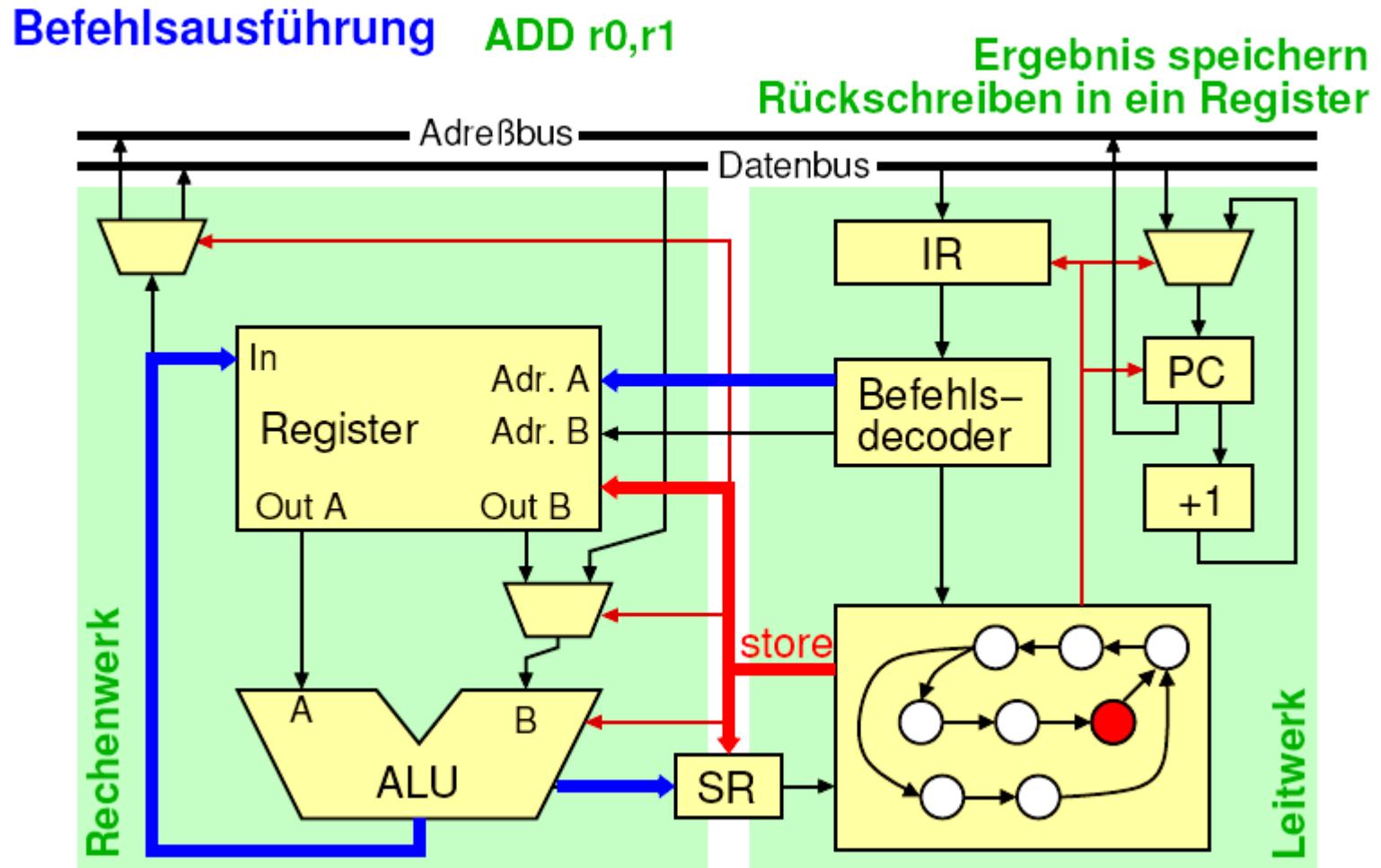
Prozess: Befehlsausführung in einer CPU

Befehlsausführung ADD r0,r1

Operation ausführen
ALU auf Addieren einstellen

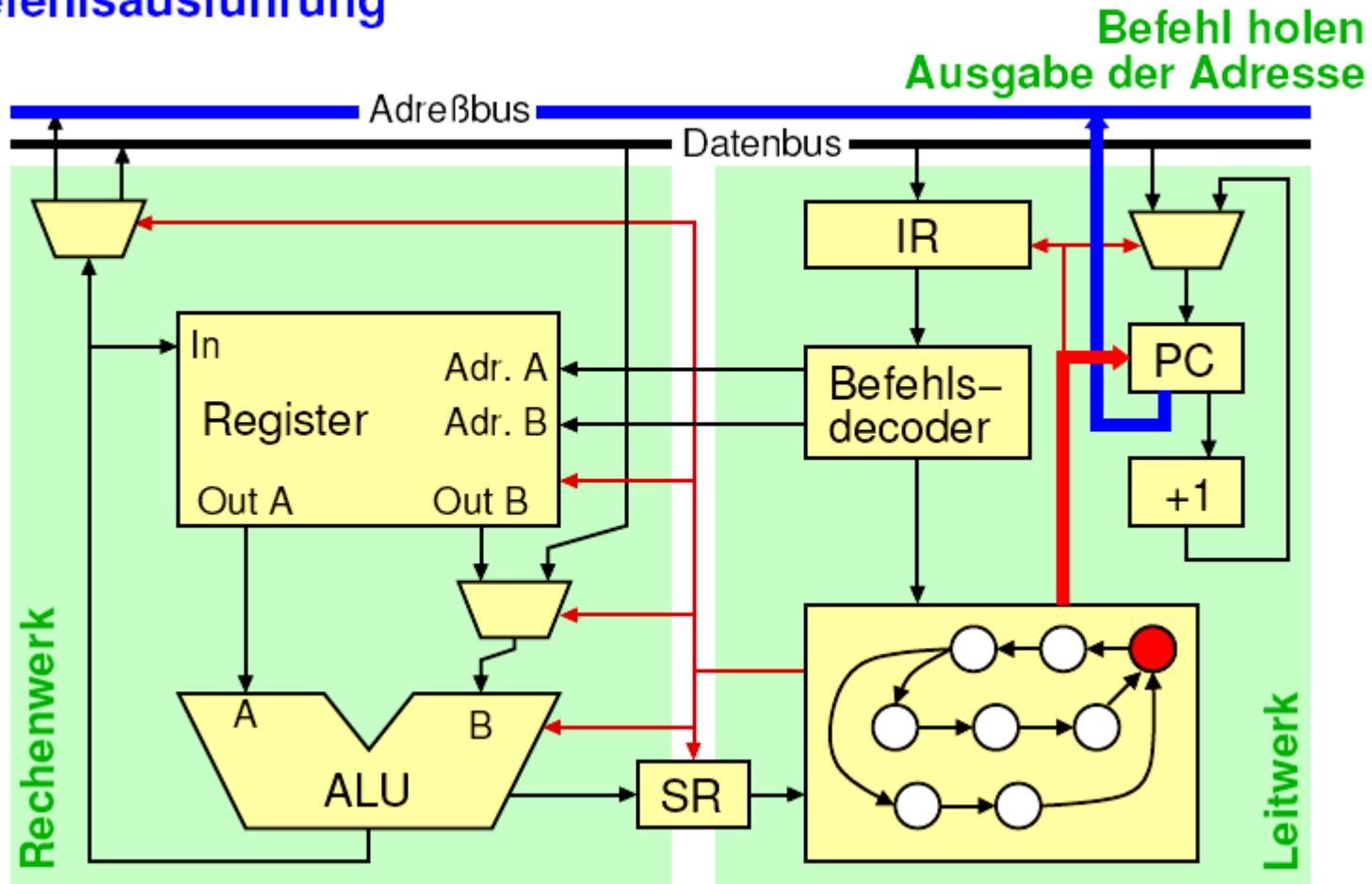


Prozess: Befehlsausführung in einer CPU



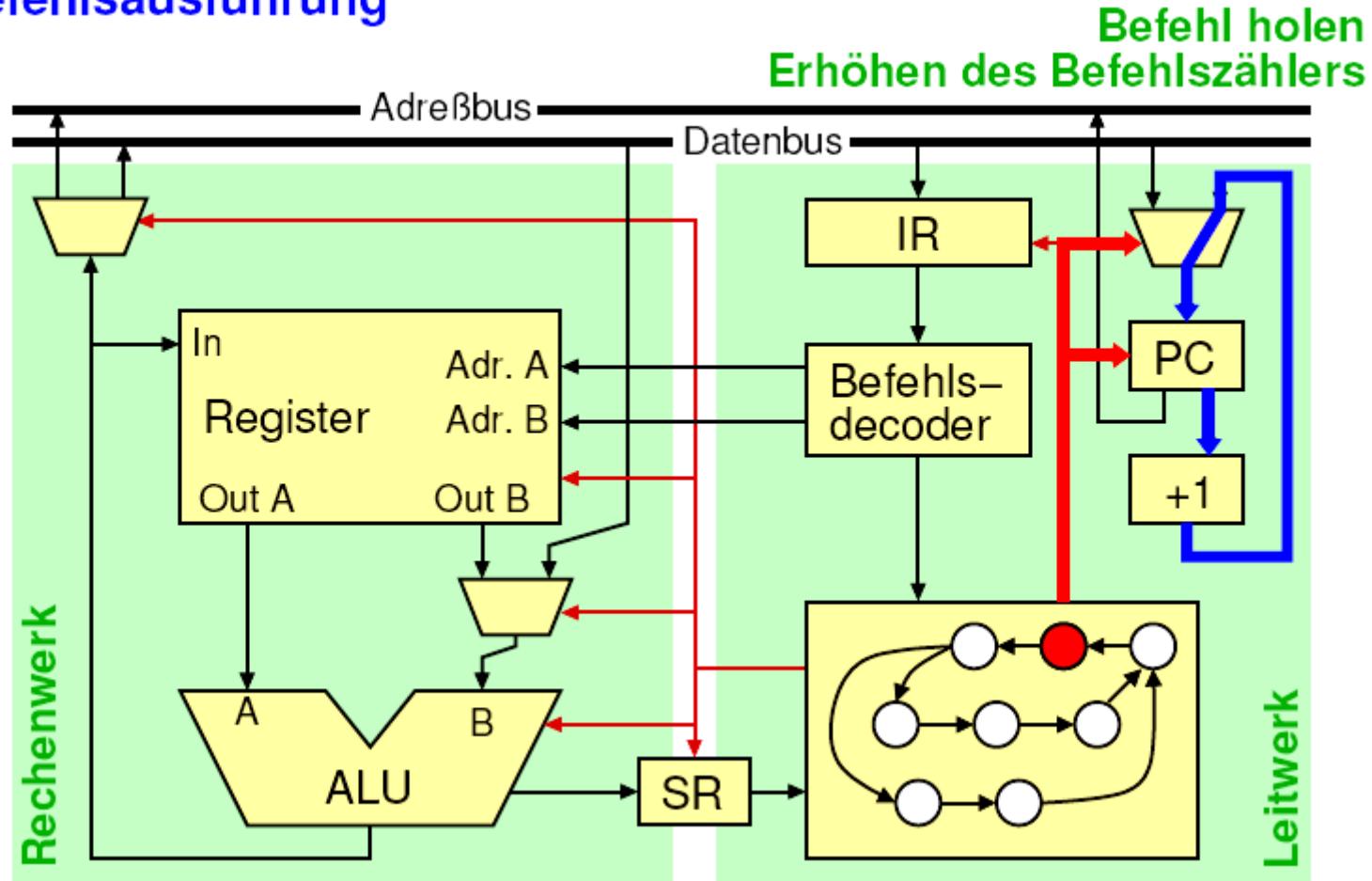
Prozess: Befehlsausführung in einer CPU

Befehlsausführung

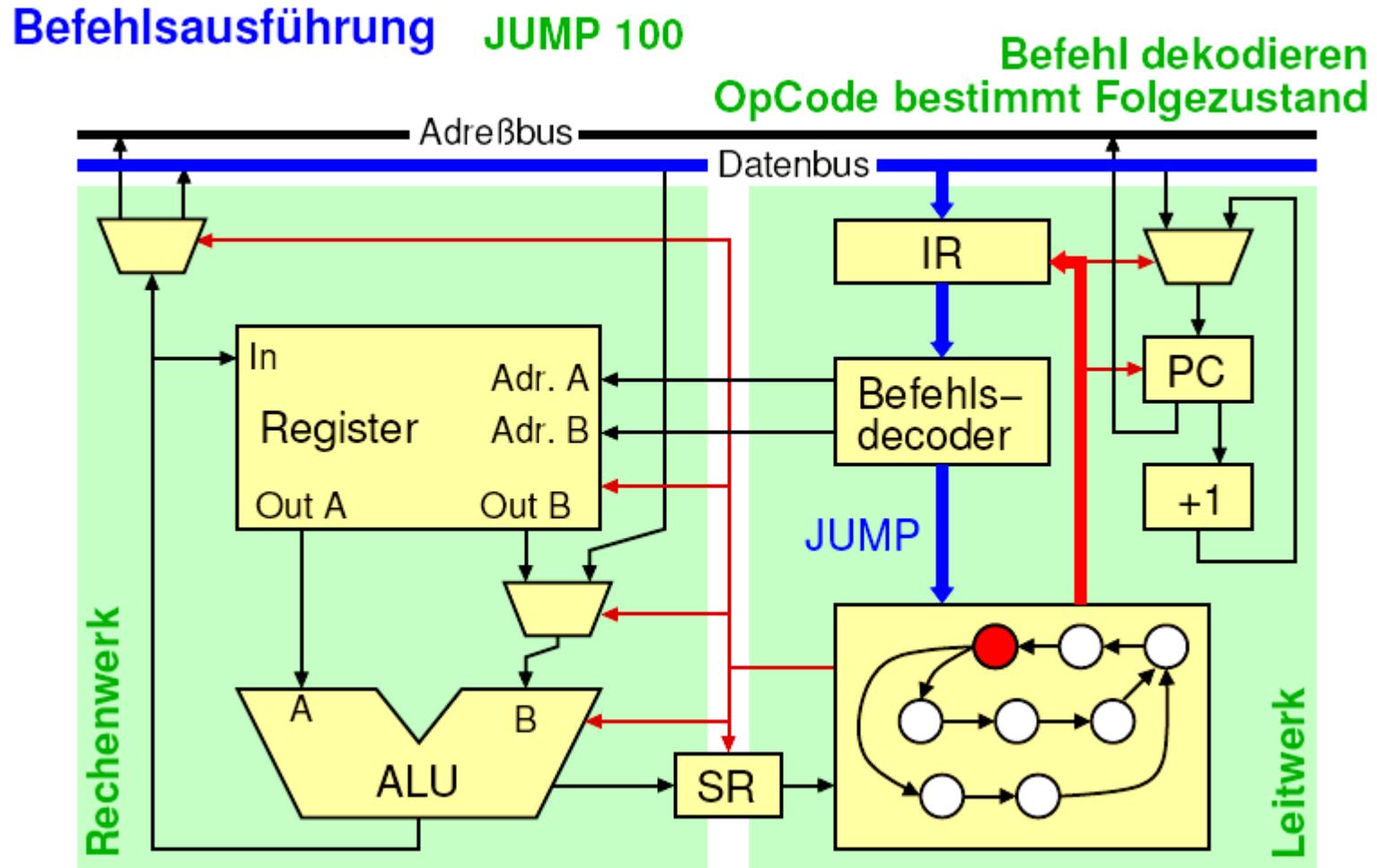


Prozess: Befehlsausführung in einer CPU

Befehlsausführung



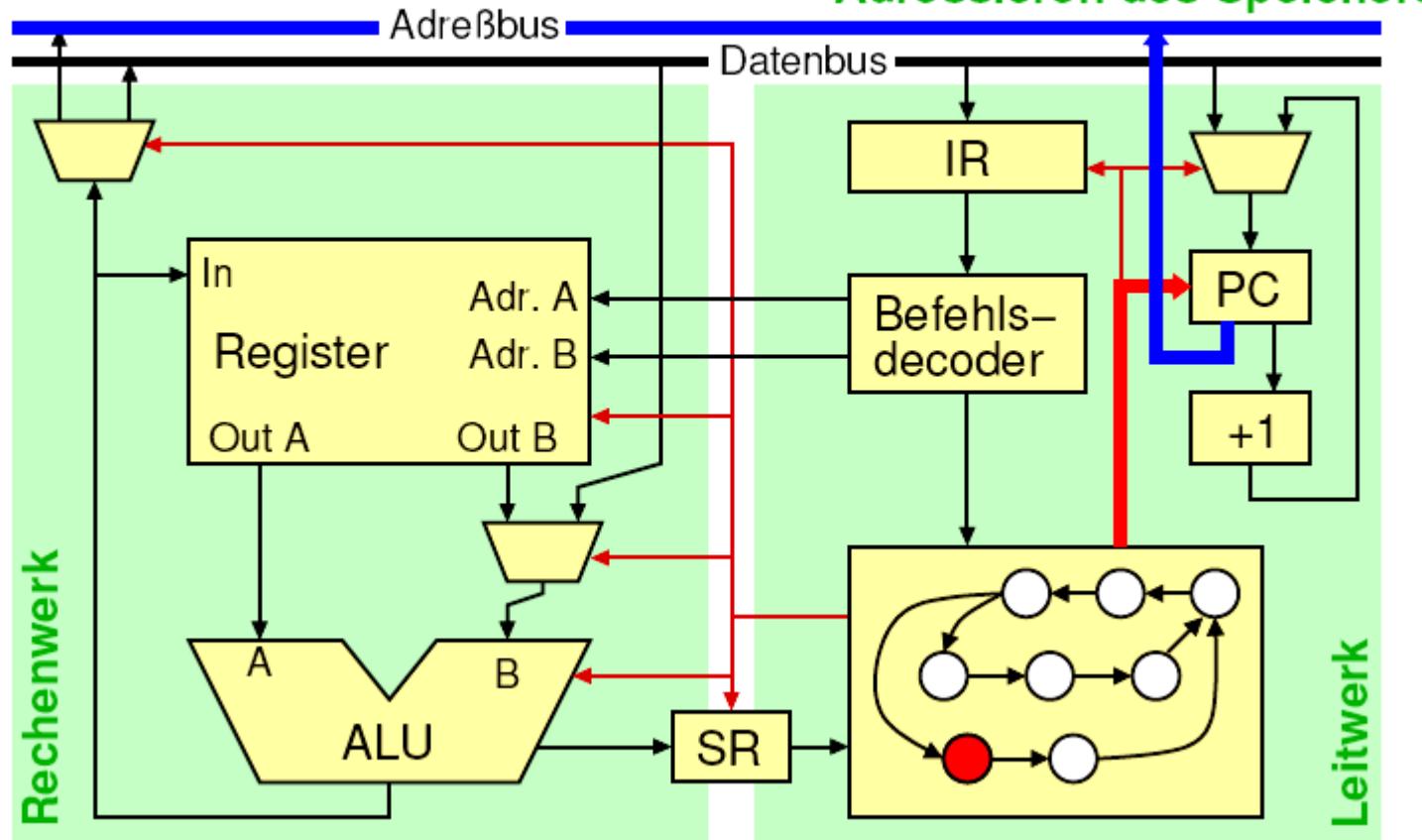
Prozess: Befehlsausführung in einer CPU



Prozess: Befehlsausführung in einer CPU

Befehlsausführung JUMP 100

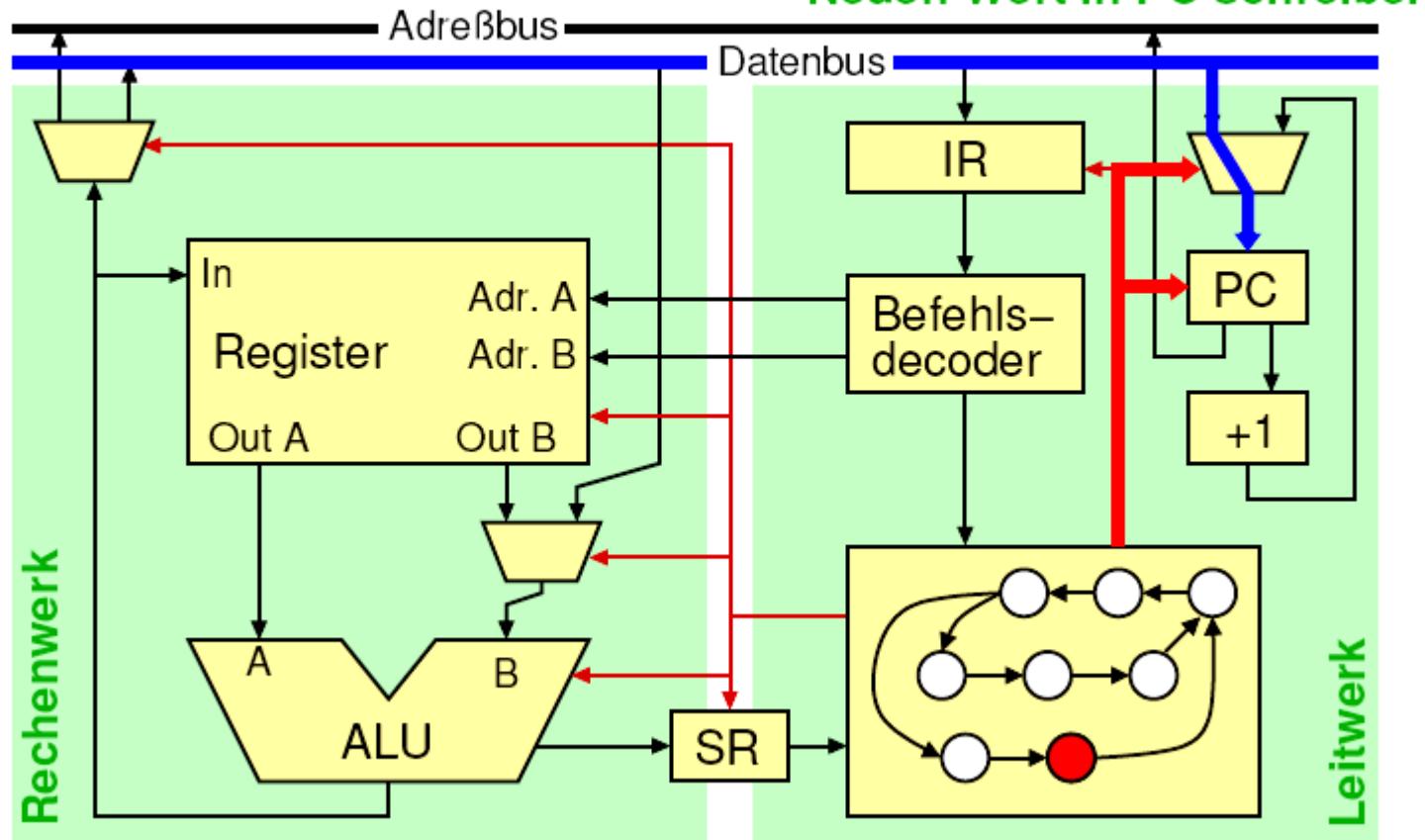
Operanden holen
Adressieren des Speichers



Prozess: Befehlsausführung in einer CPU

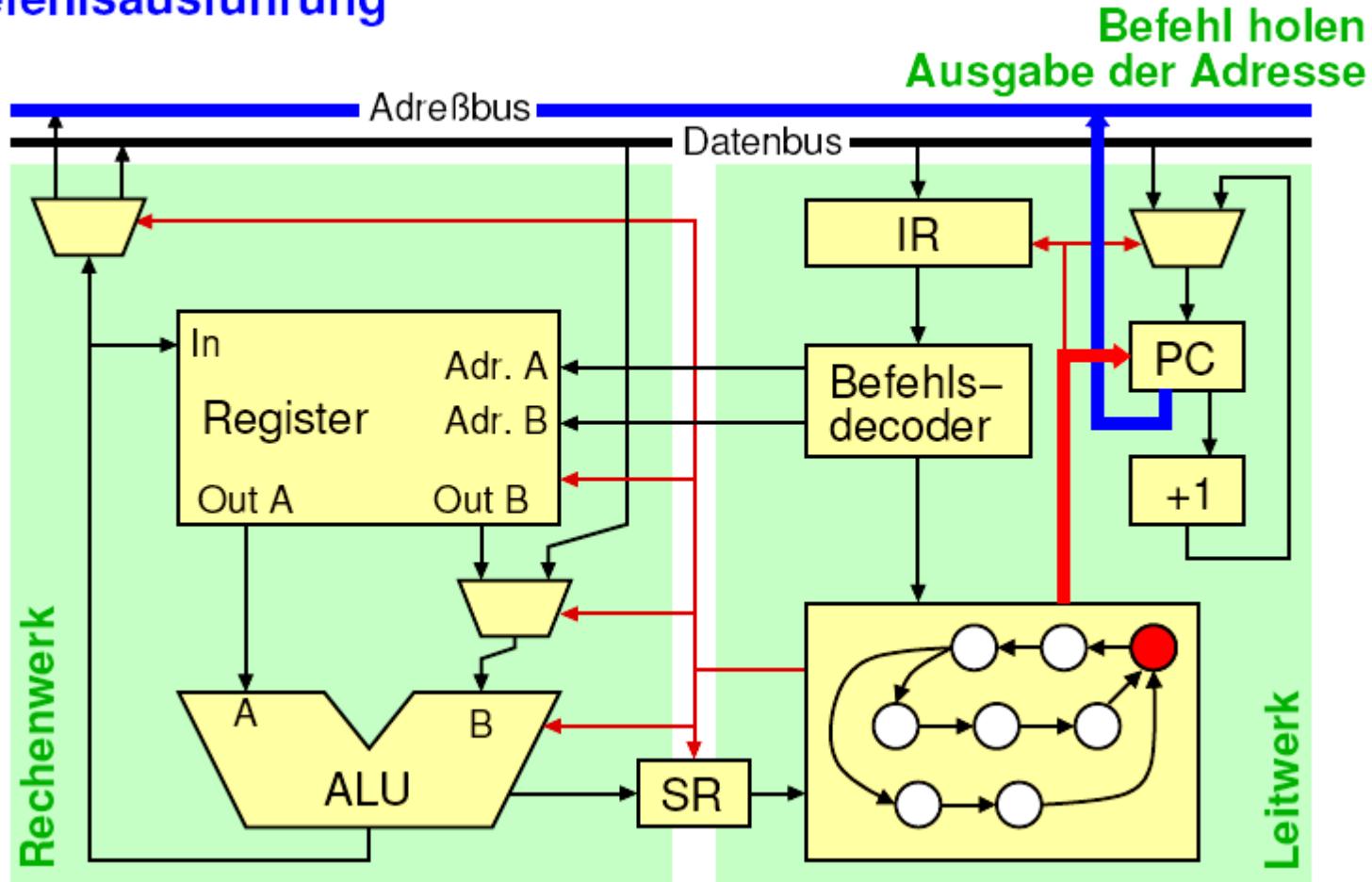
Befehlsausführung JUMP 100

Ergebnis speichern
Neuen Wert in PC schreiben



Prozess: Befehlsausführung in einer CPU

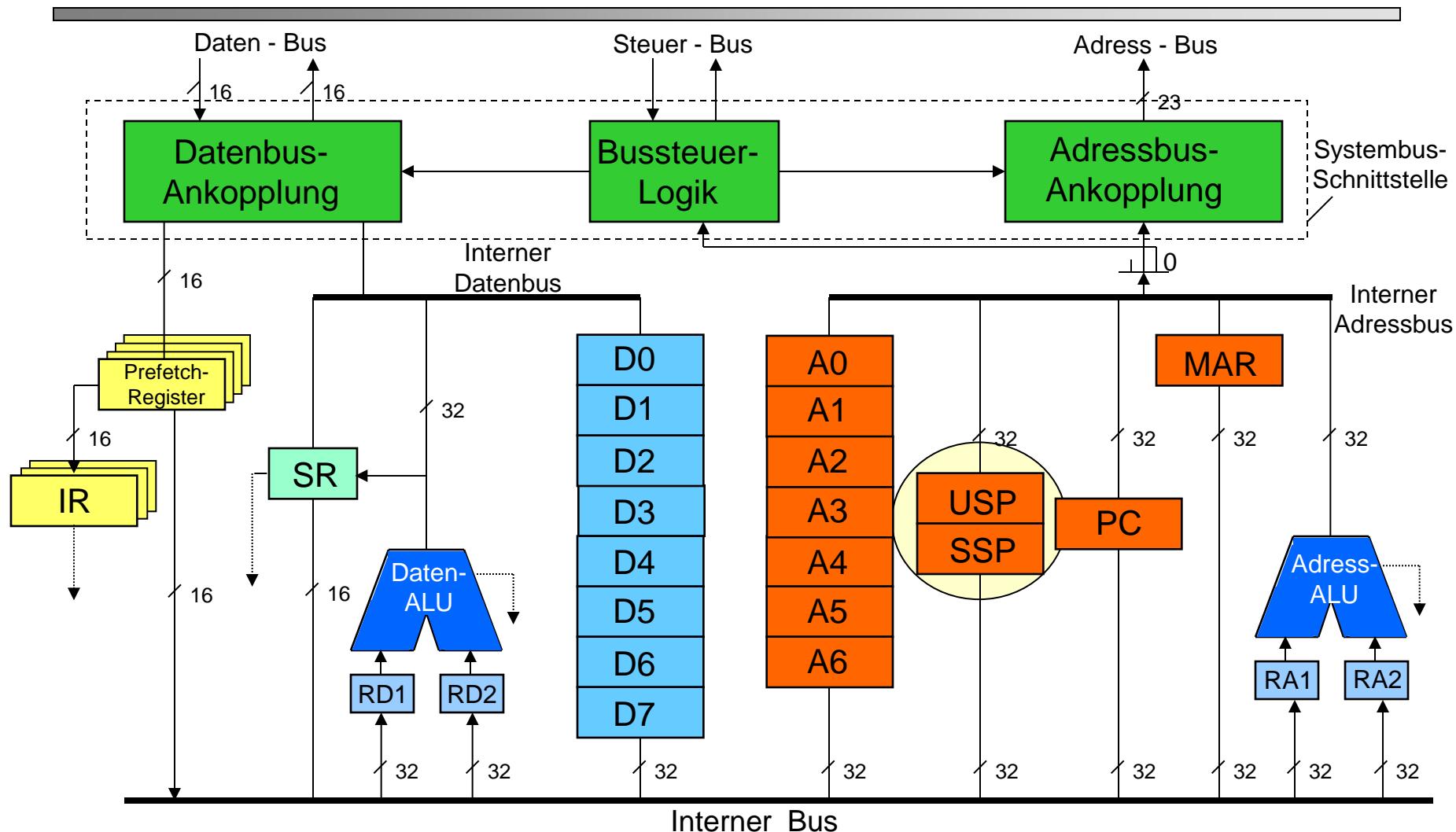
Befehlsausführung



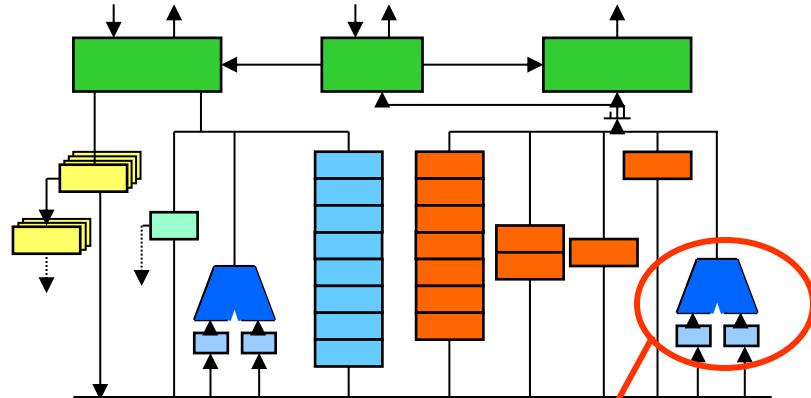
Der Mikroprozessor M68000

- Im Folgenden wollen wir exemplarisch einen realen Prozessor als Modellprozessor betrachten: Den **Mikroprozessor M68000 von Motorola**.
- Er eignet sich didaktisch in besonderer Weise für einen Einführungskurs, da er alle wesentlichen Komponenten einer CPU besitzt und klar strukturiert ist. Das Praktikum zur Vorlesung im Bereich der maschinennahen Programmierung wird auf diesem Prozessor ablaufen.

Operationswerk des M68000



Operationswerk des M68000



Operationswerk:

- ⇒ Führt die vom Steuerwerk verlangten logischen und arithmetischen Operationen aus

Adresswerk:

- ⇒ Berechnet nach den Vorschriften des Steuerwerks die Adresse eines Befehls oder eines Operanden
- ⇒ Früher häufig Bestandteil des Rechenwerks
- ⇒ Heute sehr komplex (viele verschiedene komplexe Adressierungsarten) und deshalb eigenständig

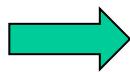
Operationswerk - Registersatz

Register:

Speicherzellen mit kleiner Zugriffszeit (wenige ns)

- Auswahl einzelner Register durch individuelle Steuerleitungen
kleine Register-Anzahl ➔ keine Adressdecoder erforderlich ➔ Zeit der Adreßdecodierung entfällt

- Register sind auf dem Prozessorchip untergebracht
➔ zeitraubendes Umschalten auf externe Daten- und Adresswege entfällt



Datenregister und Adressregister

Operationswerk - Registersatz

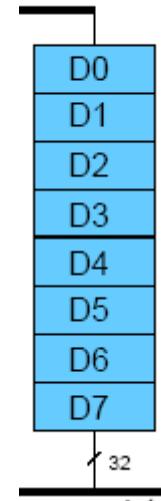
- getrennte Ein-/Ausgänge ➔ Dual Port Speicher, zwischen Eingangs- und Ausgangsbus gehängt
 - ➔ Schreiben eines Registers und gleichzeitiges Lesen eines anderen Registers möglich

Heutige superskalar-Prozessoren: pro Takt
4 allgemeine Register schreiben und bis zu 8
allgemeine Register lesen
- oft dynamische Speicherzellen ➔ Refresh erforderlich
- Register mit Zusatzfunktionen:
Inkrementieren/Dekrementieren/auf Null setzen/Inhalt verschieben

Eigenschaften der Register (M68000)

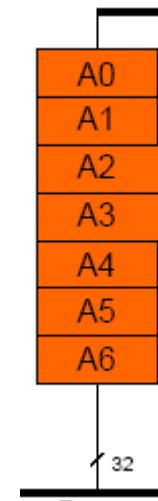
- **Datenregister**

- Mit Bezeichnungen D0,...,D7 in Befehlen ansprechbar
- Verarbeitung von BCD-, Byte-, Wort- und Langwortverarbeitung (32 Bit), wobei Bytes an den niedrigwertigsten acht Stellen des Datenregisters (0,...,7) gespeichert werden.
- Der nicht angesprochene Inhalt des Registers ändert sich nicht



Eigenschaften der Register (M68000)

- **Adressregister**
 - Mit Bezeichnungen A0,...,A7 in Befehlen ansprechbar
 - Vorzeichenrichtige Erweiterung der Operanden im Register (sign extended)
 - Bit- und Byteverarbeitung nicht erlaubt
 - Flags im Statusregister werden nicht beeinflusst
 - Nur die niederwertigsten 24 Bit werden auf den Adressbus nach extern gegeben
 - Adressbereich bis 16 MByte

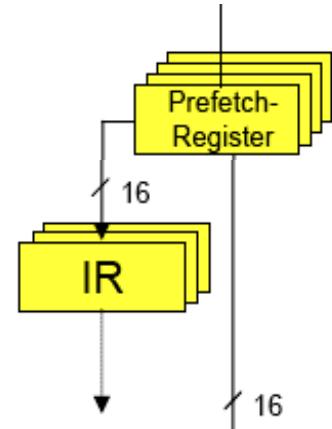


Befehlsregister (Instruction Register IR)

Das Befehlsregister besteht aus mehreren Registern:

Gründe:

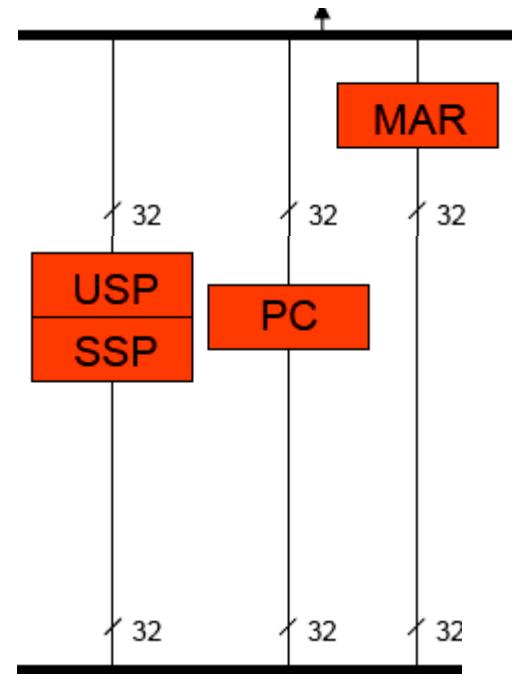
- ⇒ unterschiedlich lange Befehlsformate:
verschiedene Befehle sind unterschiedlich lang
(1-Wort-Befehle, 2-Wort-Befehle, 3-Wort-Befehle, ...)
- ⇒ Vorabladen von Befehlen (*Opcode-Prefetching*):
zur Steigerung der Verarbeitungsgeschwindigkeit
werden bereits mehrere folgende Befehle in das
Befehlsregister geladen, während der aktuelle Befehl
gerade dekodiert wird



Opcode prefetch queue, Warteschlange, Pipelining

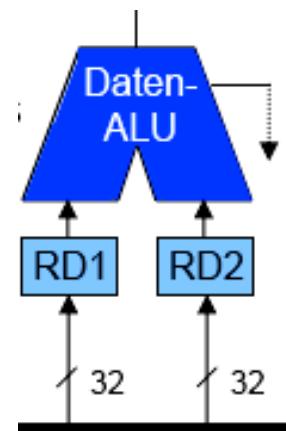
Operationswerk - spezielle Register

- Programmzähler
 - Program Counter PC
- Speicher-Adressregister
 - Memory Adress Register MAR
- Stackregister
 - User-Stackpointer USP
 - Supervisor-Stackpointer SSP



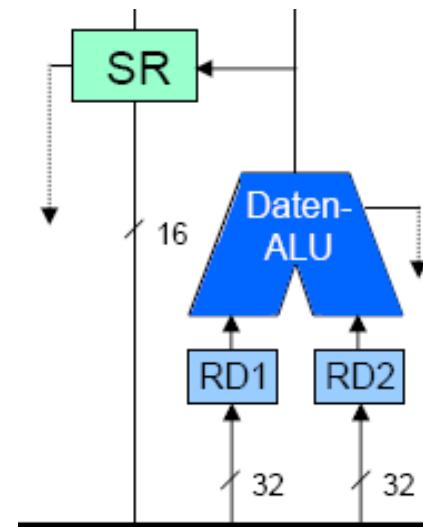
Das „Rechenwerk“ – die ALU

- 2 Eingangsbusse (2 Operanden) und 1 Ausgangsbus (Ergebnis). Sie entsprechen in der Breite dem internen Prozessor-Bus und sind mit diesem verbunden
- Die ALU selbst ist ein reines Schaltnetz
- Vor die ALU sind Hilfsregister zur Zwischenspeicherung von Operanden geschaltet
- Die Ergebnisse werden entweder in Prozessor-Registern gespeichert oder über den externen Datenbus an andere Systemkomponenten übertragen
- Steuer-Eingänge zur Steuerung der ALU-Operationen



ALU - Funktionalität

- Arithmetische Operationen:
 - Vier Grundrechenarten, Inkrementieren, Dekrementieren, Zweierkomplementbildung, Vergleichsoperationen
- Logische Funktionen
 - Negation, UND, ODER, Antivalenz
- Schiebeoperationen
 - Links- und Rechtsschieben, Rotieren
- In Abhängigkeit von den Ergebnissen der Verarbeitung in der Daten-ALU wird das *Statusregister* beeinflusst.



Schiebeoperationen

- Dem logischen Linksschieben entspricht die Multiplikation mit 2
- Dem logischen Rechtschieben entspricht die Division durch 2
- Dies gilt jedoch nur für positive Zahlen. Bei negativen Zahlen im Zweierkomplement muß zur Vorzeichenerhaltung das höchstwertigste Bit in sich selbst zurückgeführt werden.

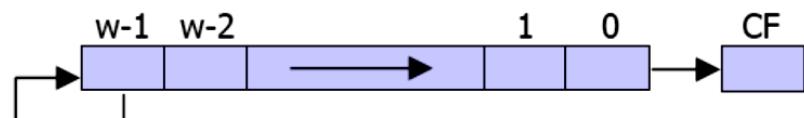
a) logisches und arithmetisches Verschieben nach links



logisches Verschieben nach rechts



b) arithmetisches Verschieben nach rechts



→ Unterschied logisches/arithmetisches Rechtsschieben

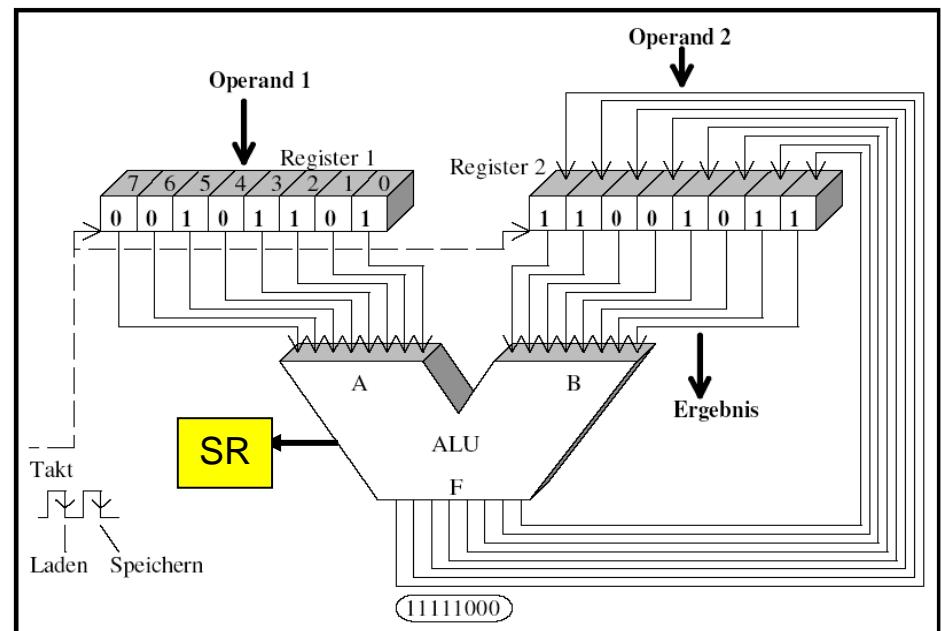
Arithmetisch logische Einheit (ALU)

- Allgemeine Gleichungen müssen in einfache Gleichungen mit maximal zwei Operanden zerlegt werden.

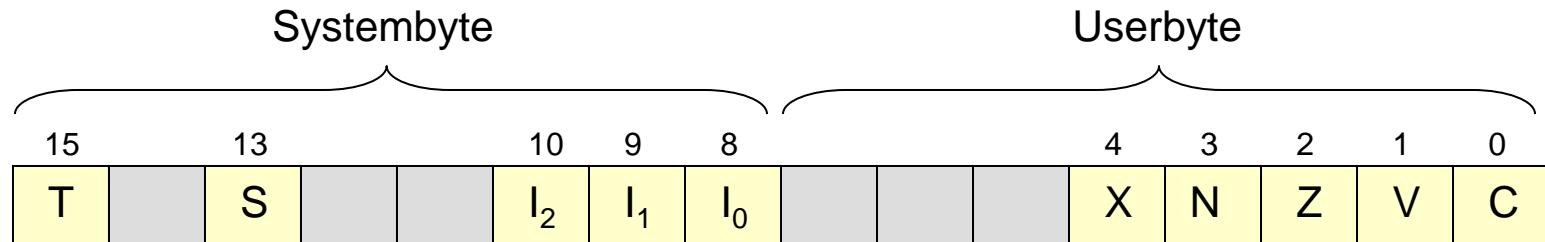
Beispiel: $x = (a + b) * c^2 \rightarrow x_1 = a + b$
 $\rightarrow x_2 = c * c$
 $\rightarrow x = x_1 * x_2$

oder:
 $\rightarrow x_1 = a + b$
 $\rightarrow x_2 = x_1 * c$
 $\rightarrow x = x_2 * c$

- Beispiel:
Parallele Addition zweier Zahlen



Statusregister beim M68000



- C Übertragsbit (Carry)
 - V Überlaufbit (Overflow)
 - Z Nullbit (Zero)
 - N Negativbit
 - X Erweiterungsbit (Extended)
-
- I₂ - I₀ Interrupt-Maske
 - S Supervisorbit (Modus)
 - T Einzelschrittbetrieb (Trace)

Statusregister beim M68000

- **Carrybit C:**
 - Das Carrybit (Übertragsbit) wird gesetzt ($C=1$), wenn bei arithmetischen Operationen mit Bytes, Worten oder Langworten ein Übertrag auftritt.
- **Zerobit Z:**
 - Das Zerobit (Nullbit) wird gesetzt ($Z=1$), wenn das Ergebnis einer Operation Null ist.
- **Negativbit N:**
 - Das Negativbit wird gesetzt ($N=1$), wenn das Ergebnis einer Operation eine negative Zahl im Zweierkomplement darstellt ($MSB=1$).
- **Extendbit X:**
 - Das Extendbit (Erweiterungsbit) verhält sich wie das Carrybit. Das C-Bit dient als Verzweigungsentscheidung, das X-Bit stellt das Übertragsbit für erweiterte Additions- und Subtraktionsbefehle dar.

Statusregister beim M68000

- **Overflowbit V:**
 - Das Overflowbit (Überlaufbit) wird gesetzt ($V=1$), wenn bei einer arithmetischen Operation eine Zahlenbereichsüberschreitung (Zweierkomplement) stattgefunden hat.
 - Ein Overflow ist dann eingetreten, wenn beide Operanden gleiches Vorzeichen haben und das Ergebnis hat das andere Vorzeichen.
- **Tracebit T:**
 - Das Tracebit dient der Realisierung des Einzelschrittbetriebs per Software. $T=1$ bedeutet, dass der Prozessor nach dem Abarbeiten jedes einzelnen Befehls in eine Trace Routine verzweigt (Testen von Programmen).

Statusregister beim M68000

- **Supervisorbit S**
 - Hardwaremäßige Unterstützung der Trennung von Anwendungsprogrammen (User Mode) und Systemprogrammen (Supervisor Mode)
 - $S = 0$: User Mode
 - $S = 1$: Supervisor Mode
 - Die Umschaltung vom User Mode in den Supervisor Mode erfolgt immer bei einer Ausnahmeverarbeitung, z.B. Interrupts oder Betriebssystem-Aufruf.
- **Interruptmaske I2, I1, I0:**
 - Die Interruptmaske legt die Laufpriorität des Prozessors fest. Nur dann, wenn die Anforderungspriorität höher als die aktuelle Laufpriorität ist, wird einer Unterbrechung stattgegeben.

Bedeutung des Statusregisters

```
if (Bedingung) {  
    Befehlsfolge  
}
```

```
do {  
    Befehlsfolge  
} while (Bedingung)
```

```
if (Bedingung) {  
    Befehlsfolge 1  
} else {  
    Befehlsfolge 2  
}
```

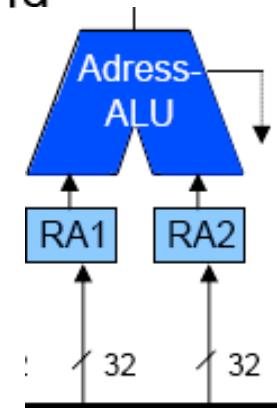
```
Switch (condition) {  
    case 0:  
    case 1:  
    case 2:  
    break  
}
```

- Alle in einer Hochsprache formulierten Verzweigungsentscheidungen müssen auf der Ebene des Prozessors in **Werte der Variablen des Statusregisters** abgebildet werden.

Das Adresswerk

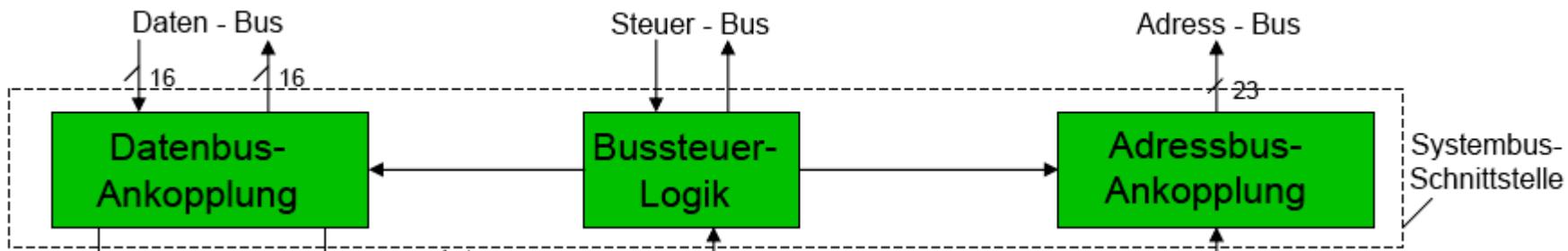
Das Adresswerk hat die Aufgabe, nach den Vorgaben des Steuerwerks aus den Inhalten bestimmter Speicherzellen und Register die Adresse eines Operanden zu bilden.

- ⇒ Adressberechnung findet parallel zu den Aktivitäten des Operationswerks statt!
- ⇒ Führt neben einfacher Adressrechnung auch die Adressrechnung zur virtuellen Speicherverwaltung durch
- ⇒ Früher oft separater Baustein, heute in den Prozessor integriert.
- ⇒ Komplexe Adresswerke zur virtuellen Speicherverwaltung (später!)

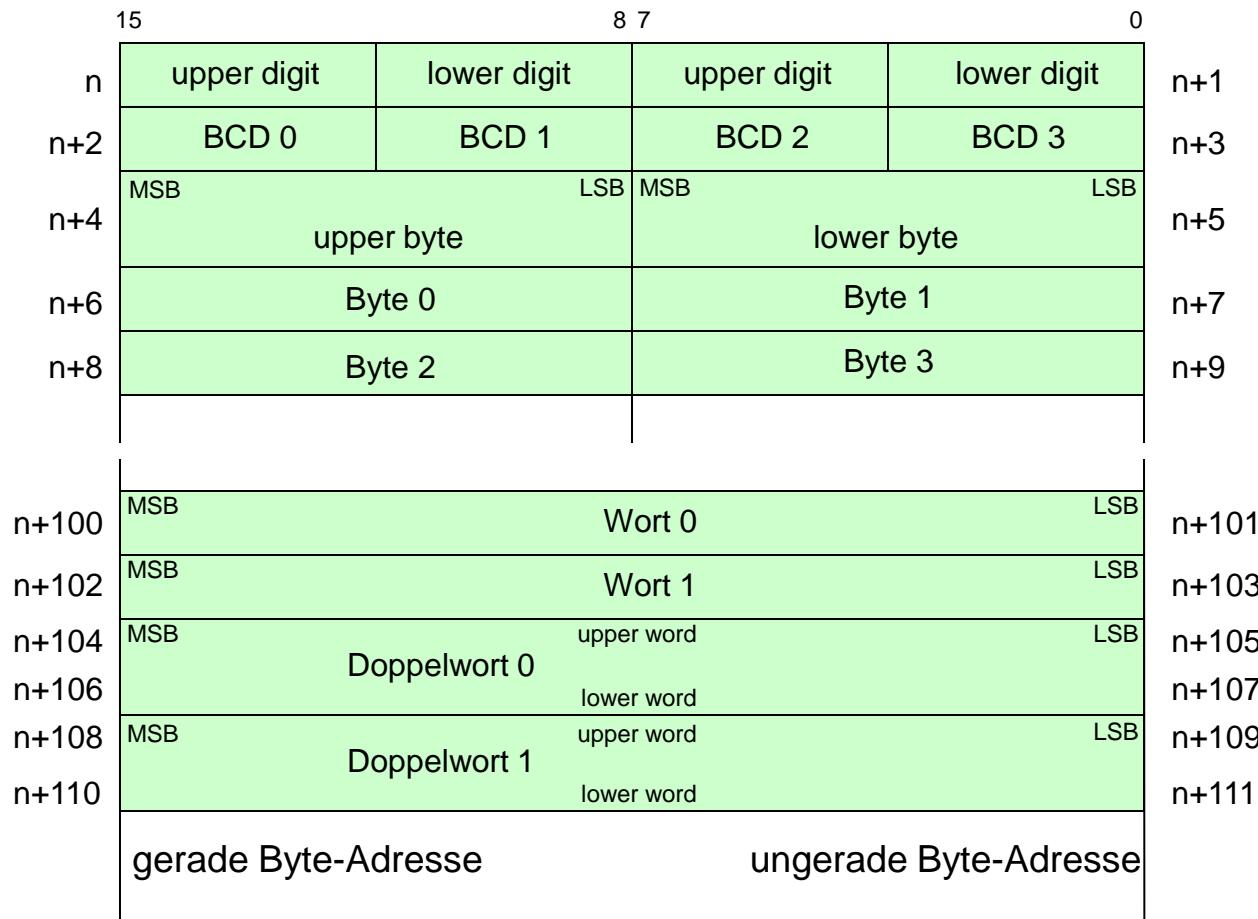


Systembus - Schnittstelle

- ⇒ Enthält diverse Zwischenspeicher-Register (Puffer) zur kurzfristigen Aufbewahrung von Adressen und Daten, z.B.
 - ⇒ **Programmzähler:** Adresse des nächsten Befehls (Instruction Pointer, Program Counter)
 - ⇒ **Adresspuffer:** Adresse des gewünschten Operanden im Hauptspeicher
 - ⇒ **Datenbuspuffer:** Wert des gerade bearbeiteten Operanden
- ⇒ Enthält Aus- und Eingangstreiber (Tristate-Treiber)

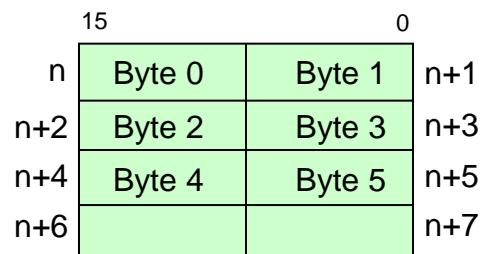


Speicherorganisation am Beispiel des M68000



Datentypen und Datenformate im Speicher/Register

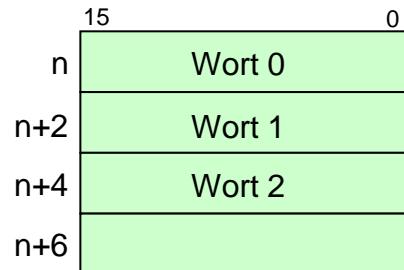
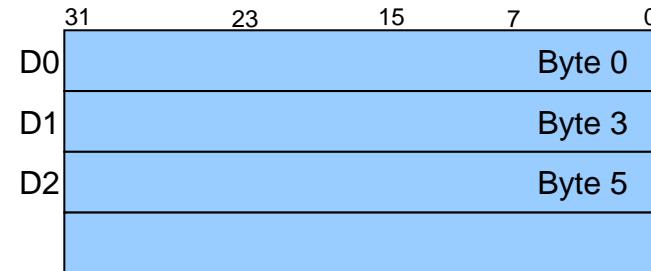
Speicheradressierung



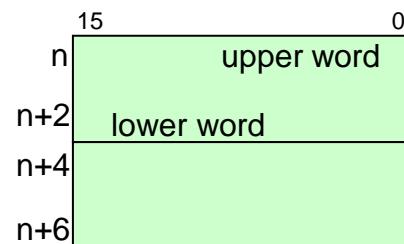
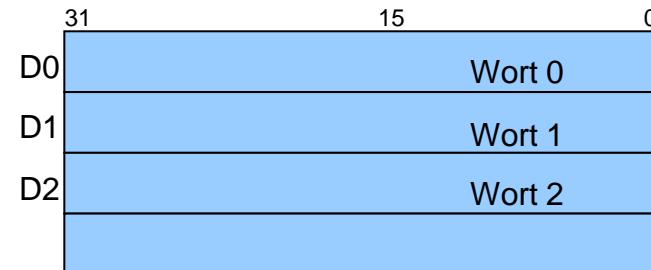
Datenformat

Byte

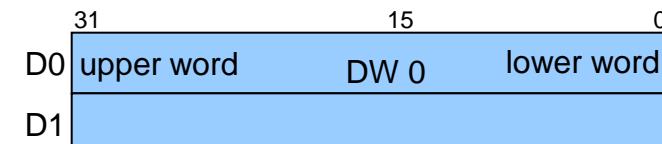
Registeradressierung



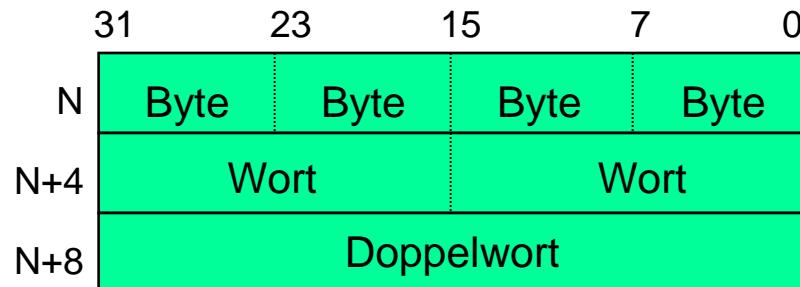
Wort



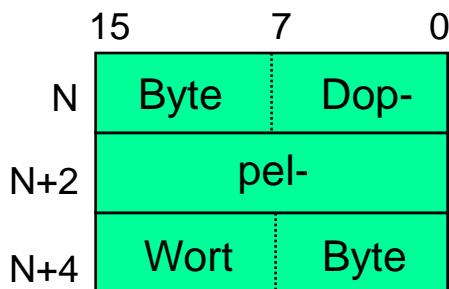
Doppel-Wort



Data-Alignment und Data-Misalignment



Data-Alignment bei 32-Bit Prozessor
im Doppelwort-orientierten Speicher



Data-Misalignment bei 32-Bit Prozessor
im wortorientierten Speicher

Data-Misalignment

- Vorteil: lückenloses Füllen möglich
- Nachteil: zusätzliche Speicherzugriffe erforderlich

Datenzugriff - Byte Ordering

Byte Ordering:

Zählweise der Bytes in einem Speicherwort:

- Von links nach rechts: "*big endian*" (beginnend beim höchstwertigen Bit),
Beispiele: IBM /370, Motorola.
- Von rechts nach links: "*little endian*" (beginnend beim niederwertigen Bit).
Beispiele: DEC VAX, Intel.

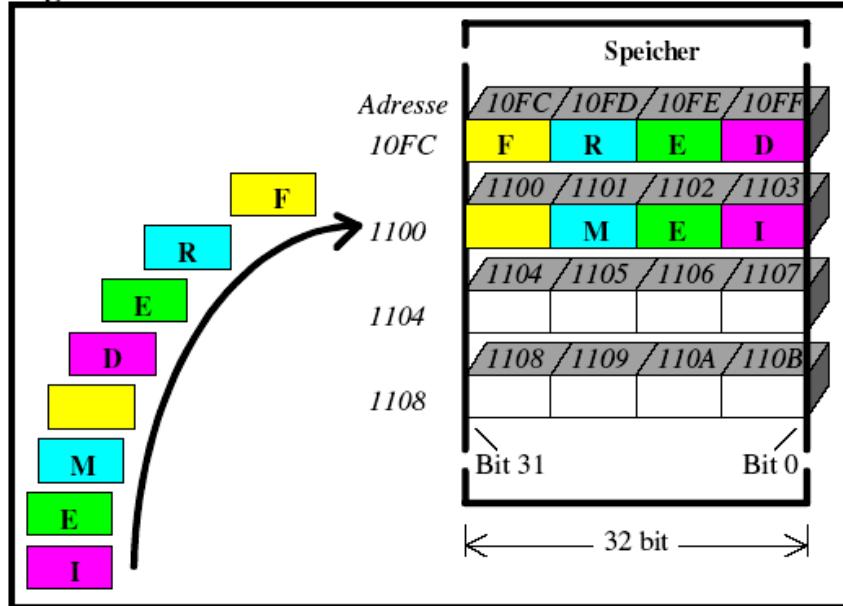
Woher der Name:

Jonathan Swift: Gulliver's Reisen (1726):

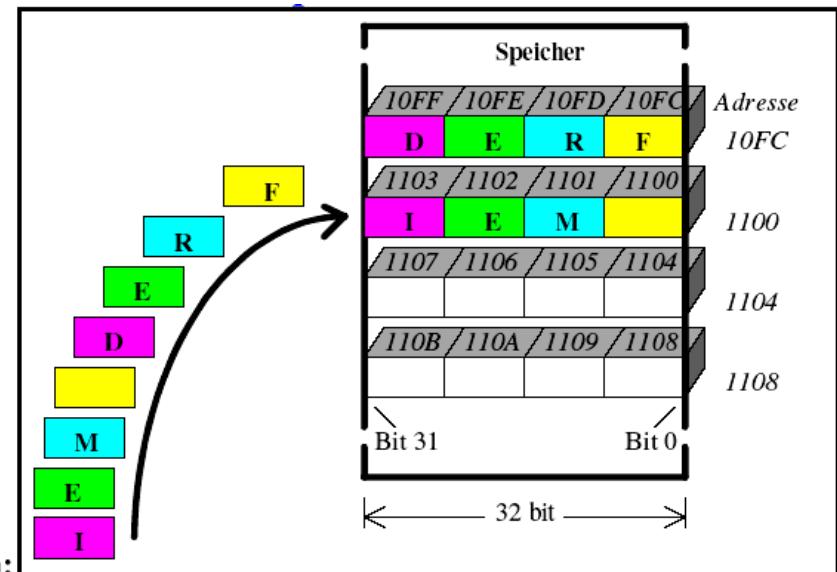
„Gulliver gelangt in das Land Liliput, dessen Einwohner nur 6 Zoll groß sind. Dort war es verboten, ein Ei an der stumpfen Seite zu öffnen, weil sich der Prinz des Landes dabei in den Finger geschnitten hatte. Ein Teil der Bevölkerung, die BIGENDIANS, wollte das nicht hinnehmen, es gab 6 Rebellionen, ein Kaiser starb, einer musste gehen. Die BIGENDIANS wurden aus dem öffentlichen Dienst ausgeschlossen, 100 BIGENDIAN-Bücher verboten. Viele BIGENDIANS retteten sich in das benachbarte Blefescu. Es kam zum Krieg, der 3 Jahre dauerte und Liliput 30000 Soldaten, vierzig große und noch mehr kleinere Schiffe kostete. Die Invasion von Liliput konnte nur dadurch verhindert werden, dass Gulliver („der Menschenberg“) die feindliche Flotte raubte....“

Datenzugriff - Byte Ordering

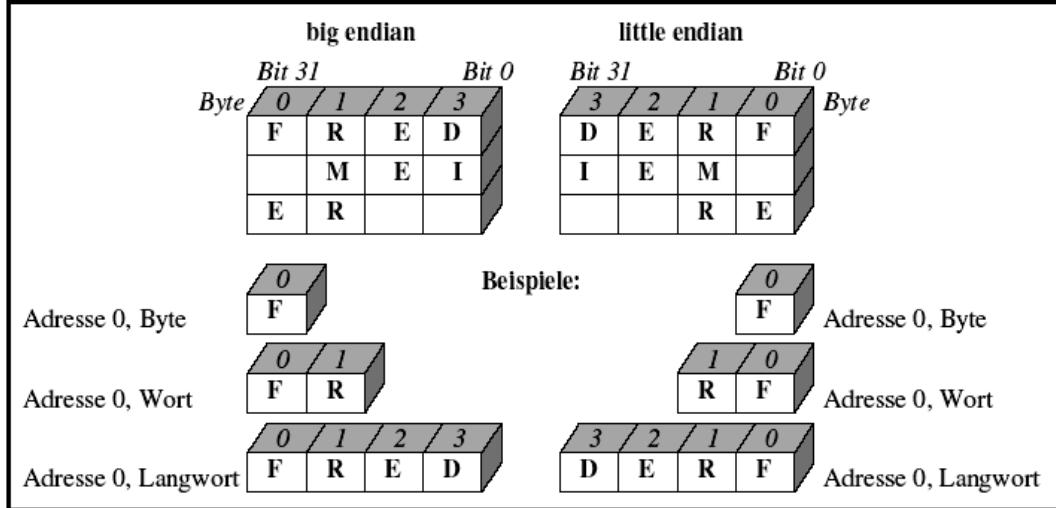
Big Endian:



Little Endian:



Ablage der Daten im Speicher



Ablage der Zeichenfolge

Beispiel: '25l₁₀'

0	0	0	0	19
---	---	---	---	----

big endian

0	0	0	0	19
---	---	---	---	----

little endian

Beispiel: Personaldaten: Name "FRED MEIER" und Alter "25"

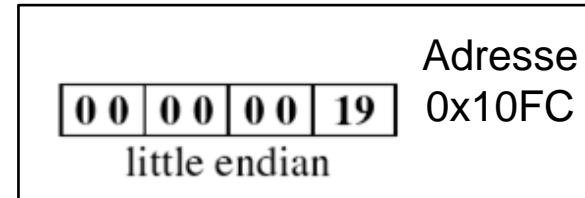
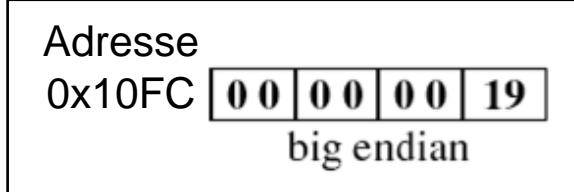
F	R	E	D
	M	E	I
E	R		
0	0	0	0
19			

big endian

D	E	R	F
I	E	M	
	R	E	
0	0	0	0
19			

little endian

Ablage der Daten im Speicher



BigEndian Byte Ordering

Daten in Formaten, die größer als ein Byte sind, werden so im Speicher abgelegt, dass das *niedrigstwertige Datenbyte* an der *höchstwertigen Adresse* und das *höchstwertige Datenbyte* an der *niedrigstwertigen Adresse* steht.

LittleEndian Byte Ordering

Daten in Formaten, die größer als ein Byte sind, werden so im Speicher abgelegt, dass das *niedrigstwertige Datenbyte* an der *niedrigstwertigen Adresse* und das *höchstwertige Datenbyte* an der *höchstwertigen Adresse* steht.

Funktionen und lokale Variablen

- Alle Programmiersprachen unterstützen das Konzept der Prozeduren oder Funktionen.
- Rekursive Funktionsaufrufe sind zulässig, d.h. eine Funktion darf sich selbst aufrufen.
- Beispiel: Fakultätsberechnung

```
fakultaet(int n) {  
    if ( n == 1 )  
        return(1);  
    else  
        return( n * fakultaet(n-1) );
```

→ Funktionen erfordern: Konzept der lokalen Variablen

Globale und lokale Variablen

- **Globale Variablen** sind im gesamten Programm einschließlich aller Unterprogramme bekannt. (Sie sollten vermieden werden)
- **Lokale Variablen** sind innerhalb einer Funktion oder eines „Blockes“ deklarierte Variable.

```
#include <stdio.h>

int global = 0;          /* Das ist eine globale Variable */

main () {
    int lokal = 1;        /* Das ist eine Lokale Variable in main */
    printf("Global %d, Lokal %d\n", global, lokal);
    {
        int lokal = 2;   /* Das ist eine lokale Variable im Block */
        printf("Global %d, Lokal %d\n", global, lokal);
    }
    printf("Global %d, Lokal %d\n", global, lokal);
}
```

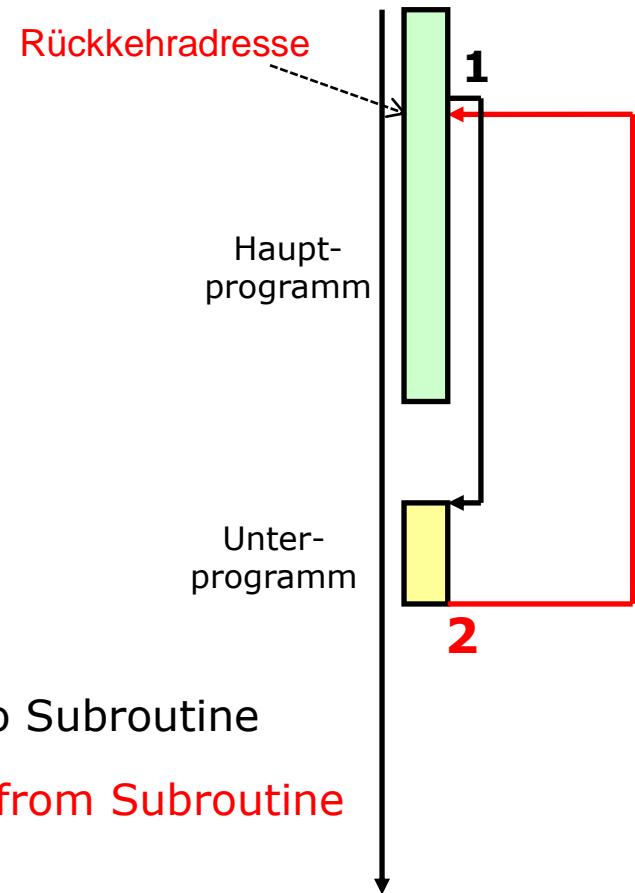
→ Wo werden lokale Variablen gespeichert?

Besonderer Speicherbereich: Der Stack

- Der **Stack** ist ein besonderer Speicherbereich, der im Arbeitsspeicher angelegt wird und nach dem **Stapelprinzip** (Kellerprinzip) organisiert ist (*LIFO – Last-in-first-out*).
- **Funktion:**
 - Der Stack dient zum Speichern lokaler Variablen einer Prozedur (Funktion).
 - Ort zur Parameterübergabe
 - Abspeichern des Befehlszählerstands beim Aufruf eines Unterprogramms.
 - → **Beispiel**
- Bei modernen Prozessoren werden mehrere getrennte Stackspeicher verwendet:
 - System Stack, User Stack, Data Stack

Aufruf von Unterprogrammen

- Ein **Unterprogramm (subroutine)** ist ein eigenständiges, ablauffähiges Programm, das von einem übergeordneten Programm aus beliebig oft aufgerufen werden kann. Das Unterprogramm erscheint nur einmal im Speicher.



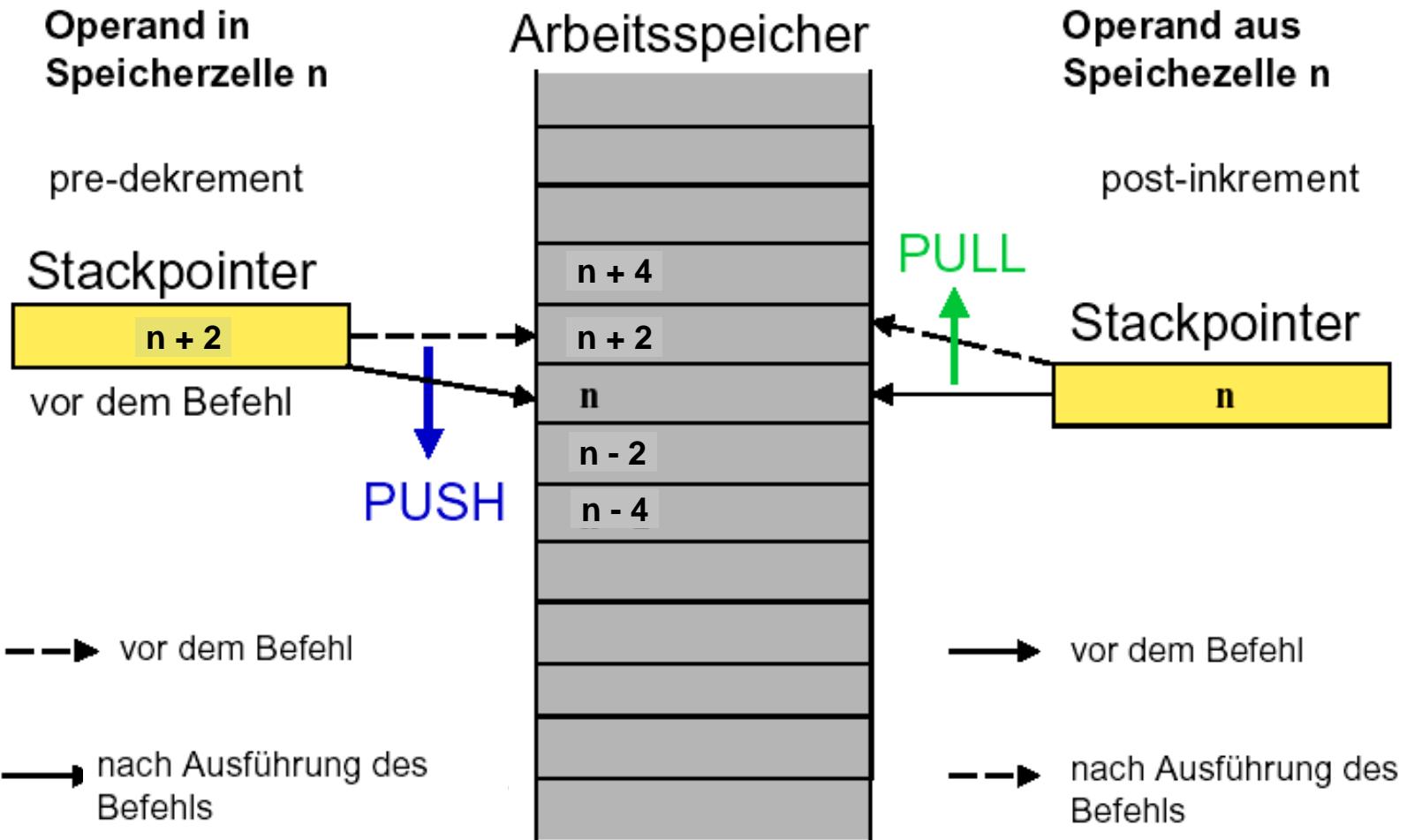
1: Jump to Subroutine

2: Return from Subroutine

Hardware-Unterstützung des Stacks

- Zur Adressierung des Stacks wird ein spezielles Prozessorregister verwendet: Der **Stackpointer SP (Stapelzeiger)**.
 - Der SP enthält immer die Adresse des zuletzt auf dem Stack eingetragenen Datenworts (*Top-of-stack*).
- Für die Übertragung von Daten auf den bzw. aus dem Stack werden zwei symbolische Befehle bereitgestellt:
 - **Push**
 - Transportiert Daten vom Prozessorregister auf den Stack
 - **Pull**
 - Transportiert Daten vom Stack in ein Prozessorregister
- Die zuletzt auf den Stack gebrachte Information wird auch als erste wieder geholt (*Last-in-First-out*). Der Stack wächst per Definition von hohen zu niedrigeren Adressen.

Hardware-Unterstützung des Stacks



Implementierung der Stackoperationen

- Die *symbolischen Befehle* PUSH und PULL zur Manipulation des Stackpointers müssen bei einem M68000-System durch Standard Assemblerbefehle nachgebildet werden.
- Abbildung der Stackoperationen auf M68000-Befehle:
 - Push: MOVE.X < eA >, - (A7)
 - Pull: MOVE.X (A7) + , < eA >

Prinzip von Warteschlangen

- Bei der Kommunikation zwischen zwei Prozessen benötigt man u.U. einen Zwischenpuffer, bei dem die zuerst abgelegten Daten auch wieder als erste geholt werden:

Queue oder Warteschlange

- Ist dies auch eine Struktur von Typ LIFO?
- Wie kann man die Verwaltung dieser Datenstruktur, d.h. den Zugriff auf einzelne Elemente organisieren?

Klassen von Architekturen – eine Übersicht

□ **Klassen von Architekturen, Ausführungsmodelle**

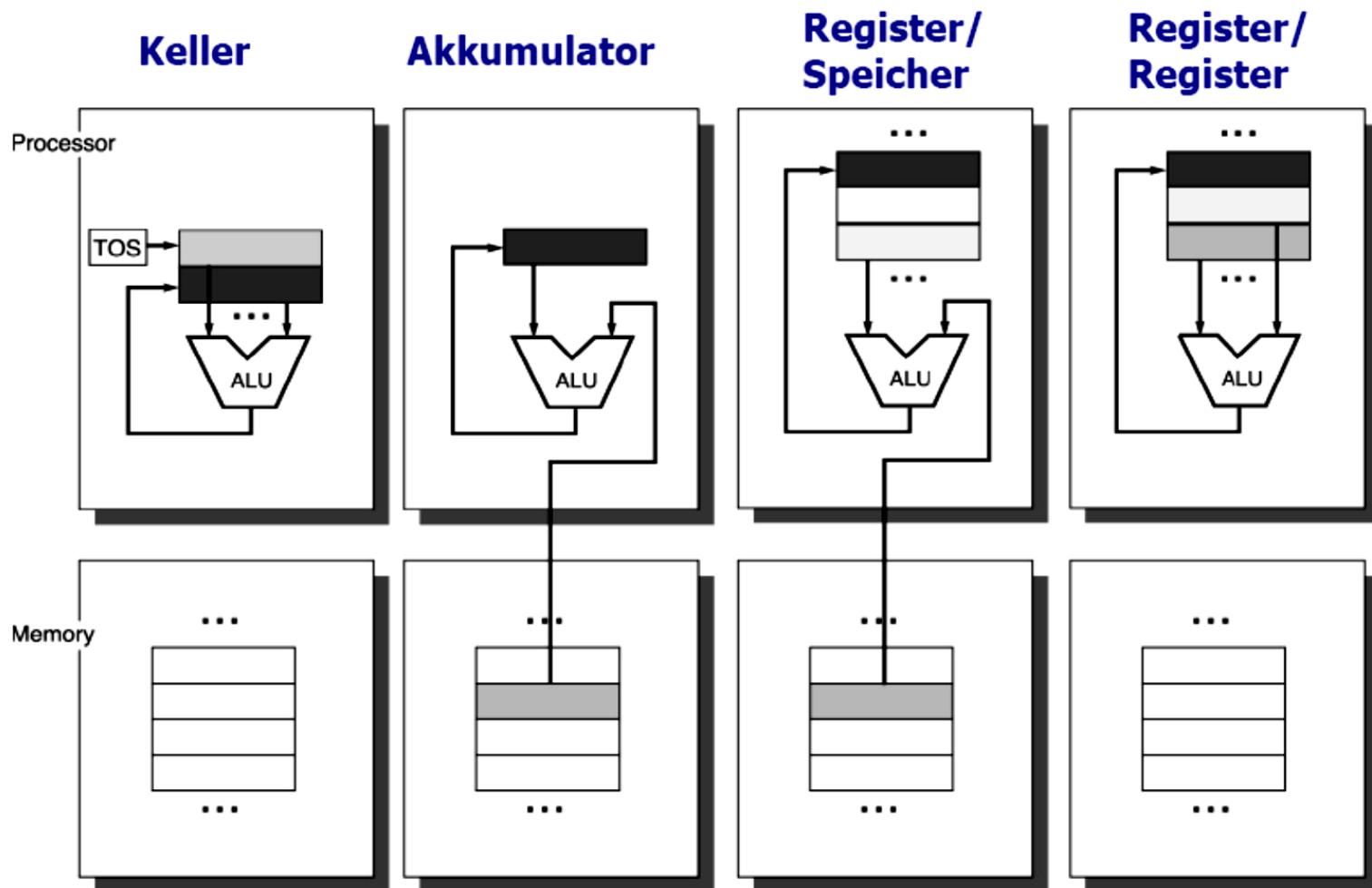
- Für eine zweistellige Operation, d. h. bei einer Operation, bei der die zwei Operanden miteinander verknüpft werden, sind folgende Angaben notwendig
 - Art der Operation
 - Adresse des ersten Operanden
 - Adresse des zweiten Operanden
 - Adresse des Resultats

Adressformate für Operationen

- ❑ monadische (einstellige) Operationen $A := op B$
 - ❑ dyadische (zweistellige) Operationen $A := C op B$

Adressformate für dyadische Operationen:

Klassen von Architekturen



Klassen von Architekturen

□ Allzweckregister-Architekturen

➤ Register-Register-Modell

- Alle Operanden und das Ergebnis stehen in Allzweckregistern

add R1 ,R2 ,R3 $R1 \leftarrow R2 + R3$

- Load/Store-Architektur

- Ausgezeichnete Befehle holen die Operanden aus dem Hauptspeicher und schreiben die Inhalte von Registern in den Speicher

load R2 ,A $R2 \leftarrow \text{mem}[A]$

load R3 ,B $R3 \leftarrow \text{mem}[B]$

add R1 ,R2 ,R3 $R1 \leftarrow R2 + R3$

store C ,R1 $\text{mem}[C] \leftarrow R1$

- Dreiadressformat

Klassen von Architekturen

□ Allzweckregister-Architekturen

➤ Register-Register-Modell

- Vorteil:

- Einfaches und festes Befehlsformat
 - Einfaches Code-Generierungsmodell
 - Etwa gleiche Ausführungszeit der Befehle

- Nachteil:

- Höhere Anzahl von Befehlen im Vergleich zu Architekturen mit Speicherreferenzen
 - Mehr Instruktionen und geringere Befehlsdichte führen zu längeren Programmen

- Beispiele

- Alpha, ARM, MIPS, PowerPC, SPARC, Trimedia TM5200

Klassen von Architekturen

□ Allzweckregister-Architekturen

➤ Register-Speicher-Modell

- Ein Operand steht im Speicher, der zweite Operand steht im Register; das Ergebnis steht entweder im Speicher oder im Register

- **add A,R1** $\text{mem}[A] \leftarrow \text{mem}[A] + R1$
 - **add R1,A** $R1 \leftarrow R1 + \text{mem}[A]$

- explizite Adressierung mit/ohne Überdeckung

Überdeckung: ROR #2, D1 ;
Quell- und Zieladresse fallen zusammen

- Zweiaadressformat

- Befehlsformat sieht zwei explizite Adressangaben vor
 - » Registerspeicher (prozessorintern)
 - » Hauptspeicher (prozessorextern)
 - Überdeckung einer Quelladresse mit einer Zieladresse

Klassen von Architekturen

□ Allzweckregister-Architekturen

➤ Register-Speicher-Modell

- Vorteil:
 - Auf die Daten kann ohne vorherige Lade-Operation zugegriffen werden
 - Kodierung im Befehlsformat führt zu höherer Code-Dichte
- Nachteile:
 - Operanden können nicht gleich behandelt werden, wenn eine Überdeckung vorliegt
 - Anzahl der Taktzyklen pro Instruktion variiert in Abhängigkeit der Adressrechnung
- Beispiele:
 - IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x

Klassen von Architekturen

□ Akkumulator-Register-Architektur

- Ein ausgezeichnetes Register (**Akkumulator**)
 - Der Akkumulator wird bei einer zweistelligen Operation als Quelle einer der beiden Operanden angesprochen, gleichzeitig dient der Akkumulator als Ziel für das Resultat
 - **add A** $acc \leftarrow acc + mem[A]$
 - **addx A** $acc \leftarrow acc + mem[A+x]$
 - **add r1** $acc \leftarrow acc + r1$
- Implizite und überdeckte Adressierung
- Spezielle Befehle ermöglichen den Transport von Operanden
- Einadressformat

Klassen von Architekturen

□ Keller-Architektur

- Die beiden Operanden einer zweistelligen Operation stehen auf den beiden obersten Kellerelementen
- Das Ergebnis wird wieder auf dem Keller abgelegt

- add

tos \leftarrow **tos** + **next**

- Implizite Adressierung
 - über den Kellerzeiger (tos)
- Überdeckung
- Nulladressformat
- Beispiele
 - Burroughs B5000 (1968), 80x86 Gleitkomma-Architektur, Java Virtual Machine (JVM)

Klassen von Architekturen

□ Speicher-Speicher-Architektur

- Die beiden Operanden einer zweistelligen Operation sowie das Ergebnis stehen im Speicher

– `add A, B, C mem[A] ← mem[B] + mem[C]`

- Explizite Adressierung
- Dreiaadressformat
- Nachteil:
 - Speicherzugriffe führen zu Speicherengpass
- Beispiele
 - DEC VAX

Programm $C=A+B$; $D=C-B$; in den vier Befehlsformatsarten codiert

Register-Register

```
load Reg1,A  
load Reg2,B  
Add Reg3,Reg1,Reg2  
store C,Reg3  
load Reg1,C  
load Reg2,B  
sub Reg3,Reg1,Reg2  
store D,Reg3
```

Register-Speicher

```
load Reg1,A  
add Reg1,B  
store C,Reg1  
load Reg1,C  
sub Reg1,B  
store D,Reg1
```

Akkumulator

```
load A  
add B  
store C  
load C  
sub B  
store D
```

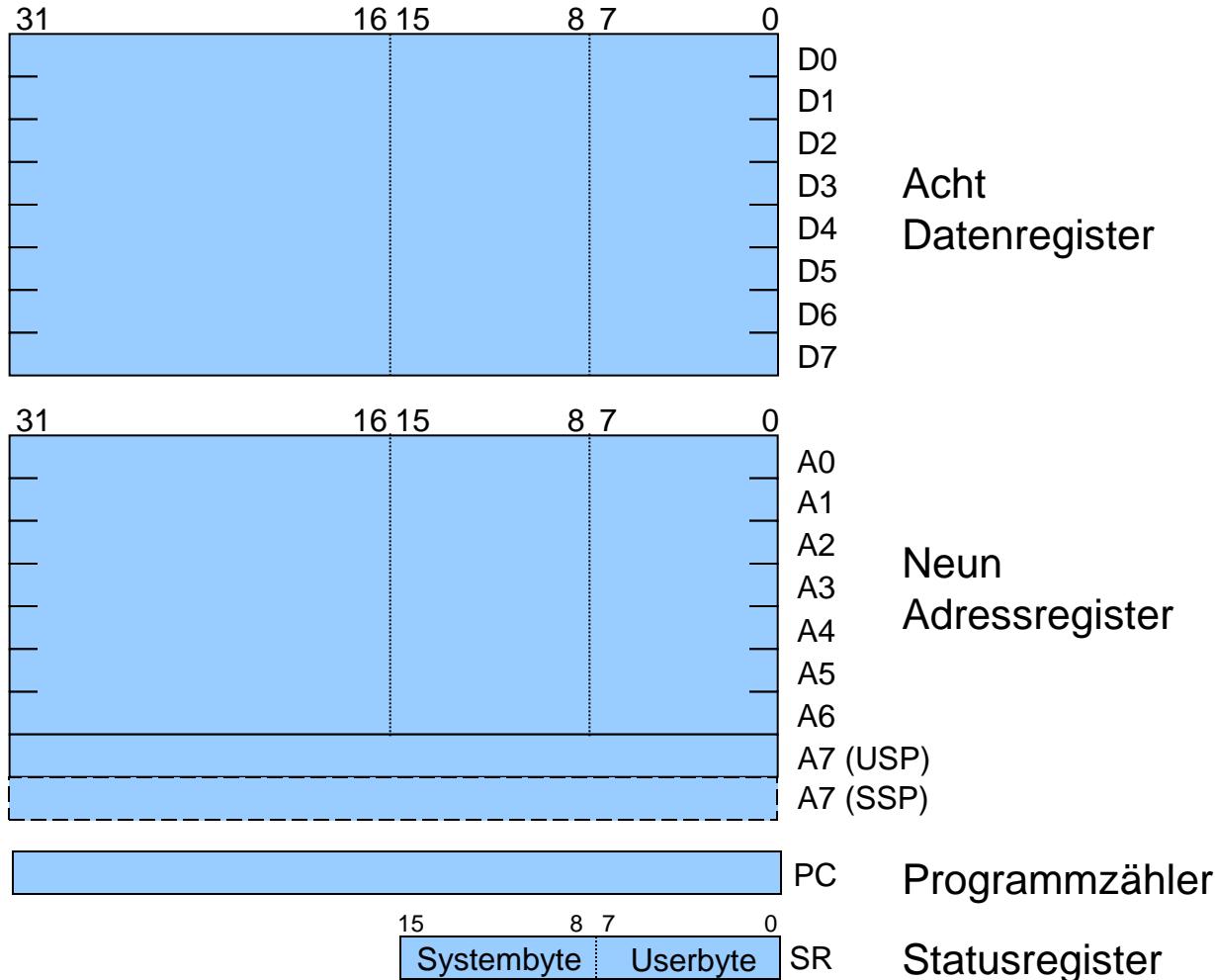
Keller

```
push B  
push A  
add  
pop C  
push C  
push B  
sub  
pop D
```

Assemblerprogrammierung

Technik der Assemblierung

M68000 - Programmiermodell



Assemblerprogrammierung

Maschinensprache: Repräsentation von Anweisungen, die für einen Mikroprozessor unmittelbar verständlich sind, z.B.

00000001100010001100000100001

Assemblersprache: eine symbolische Repräsentation der Maschinensprache, die für den Menschen verständlich und anschaulich ist, z.B.

MOVE D1, D5

Symbolischer Befehl \equiv Maschinen-Befehl

Assemblerbefehl

- Besteht aus einprägsamer Abkürzung für den Befehl und meist Operanden
⇒ Mnemo-Code für den Befehl
 - ADD addiere
 - SUB subtrahiere
 - MOVE transportiere
- Ist nur eine andere Darstellung für den binär kodierten Maschinenbefehl
- Spezifisch für einen Prozessor bzw. eine Prozessorfamilie

Assemblerbefehle - Beispiele

MOVE QADR, ZADR

Inhalt von QADR (Quelladresse)
nach ZADR (Zieladresse)

SUB QADR, ZADR

Subtrahiere den Inhalt von QADR
vom Inhalt von ZADR und schreibe
das Ergebnis nach ZADR

CMP ADR1, ADR2

Vergleiche die mit ADR1 und ADR2 adressierten
Operanden. Ergebnis in den CC-Bits des Prozessor-
Statusregisters (Dual- und 2-Komplement-Zahlen)

Assemblerbefehle - Beispiele

BNE SPRADR

Bedingter Sprung:

lade den Programmzähler mit der Sprungadresse SPRADR, sofern das Prozessor-Statusregister den Zustand "ungleich" zeigt, sonst wird der nächste Befehl im Programm ausgeführt

JMP SPRADR

Unbedingter Sprung: lade den Programmzähler mit der Sprungadresse SPRADR

MC68000 - Assemblerbefehle (Auswahl)

- ADD.B Addition zweier Operanden (Byte-Operation)
- ADD.W Addition zweier Operanden (Wort-Operation)
- ADD.L Addition zweier Operanden (Langwort-Operation)
- SUB.X Subtraktion zweier Operanden (X ist entw. B, W oder L)
- BRA Sprung auf Marke (Branch Always / verzweige immer)
- BEQ Sprung, wenn Z-Bit = 1
- BMI Sprung, wenn N-Bit = 1
- BCC Sprung, wenn C-Bit = 0
- CLR.X Löschen eines Registers
- CMP.X Vergleich zweier Operanden
- MOVE.B Transfer eines Byte
- MOVE.W Transfer eines Wortes (2 Byte)
- MOVE.L Transfer eines Langwortes (4 Byte)
- NEG.X Bildung 2-er Komplement (Negierung)

Befehlsklassen des M68000

Die 77 Befehle des M68000 sind in die folgenden 8 Befehlsklassen unterteilt. In den einzelnen Klassen werden Befehle mit ähnlichem funktionellem Verhalten zusammengefasst.

- Daten-Transportbefehle
- Arithmetische Verknüpfungen (Festkomma-Arithmetik)
- BCD - Arithmetik
- Logische Verknüpfungen
- Schiebe- und Rotationsbefehle
- Bitmanipulationsbefehle
- Programmsteuerbefehle
 - Bedingte Sprungbefehle und Sprungbedingungen
- Systembefehle

MC68000 - Assemblerbefehle (Befehlssatz-Liste)

1) Datentransport-Befehle:

Assembler-Schreibweise	Datentyp	Operation	Condition Code Reg. X N Z V C	Mögliche Adressierungsarten								#
				Dn	An	(An)	(An)+ -(An)	d(An)	\$XXXX	d(PC)	d(PC,Xn)	
MOVE.X < eA >, < eA >	B, W, L	< Quelle > → Ziel	- * * 0 0	Q/Z	Q/-	Q/Z	Q/Z	Q/Z	Q/Z	Q/-	Q/-	Q/-
MOVEA.X < eA >, An	W, L	< Quelle > → An	- - - - -	Q/-	Q/Z	Q/-	Q/-	Q/-	Q/Z	Q/-	Q/-	Q/-
MOVEQ.X #Konstante,Dn	B	Konstante → Dn	- * * 0 0	-/Z	-/-	-/-	-/-	-/-	-/-	-/-	-/-	Q/-
SWAP Dn	W	Dn[31:16] ↔ Dn[15:0]	- * * 0 0	Q/Z	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-

2) Arithmetische Befehle:

Assembler-Schreibweise	Datentyp	Operation	Condition Code Reg. X N Z V C	Mögliche Adressierungsarten								#
				Dn	An	(An)	(An)+ -(An)	d(An)	\$XXXX	d(PC)	d(PC,Xn)	
ADD.X < eA >, Dn	B, W, L	< Quelle > + < Dn > → Dn	C * * * *	Q/Z	Q/-	Q/-	Q/-	Q/-	Q/Z	Q/-	Q/-	Q/-
ADD.X Dn, < eA >	B, W, L	< Dn > + < Ziel > → Ziel	C * * * *	Q/-	-/-	-/Z	-/Z	-/Z	-/Z	-/-	-/-	-/-
ADDA.X < eA >, An	W, L	< Quelle > + < An > → An	- - - - -	Q/-	Q/Z	Q/-	Q/-	Q/-	Q/Z	Q/-	Q/-	Q/-

Programm - Übersetzung (Assemblierung)

Assembler:

Programm, das einen Quellcode in Assemblersprache in eindeutiger Weise in Maschinensprache übersetzt

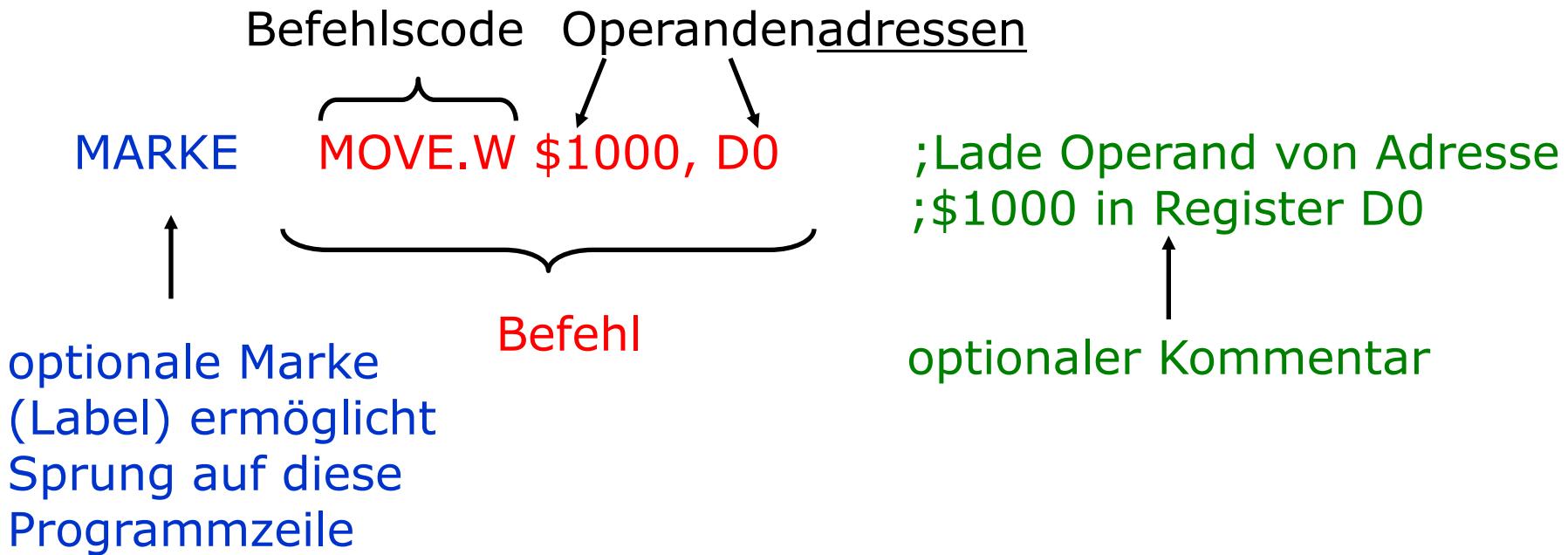
Die Regeln zur symbolischen Programmierung ergeben sich aus der Definition einer Assemblersprache

Format einer Programmzeile:

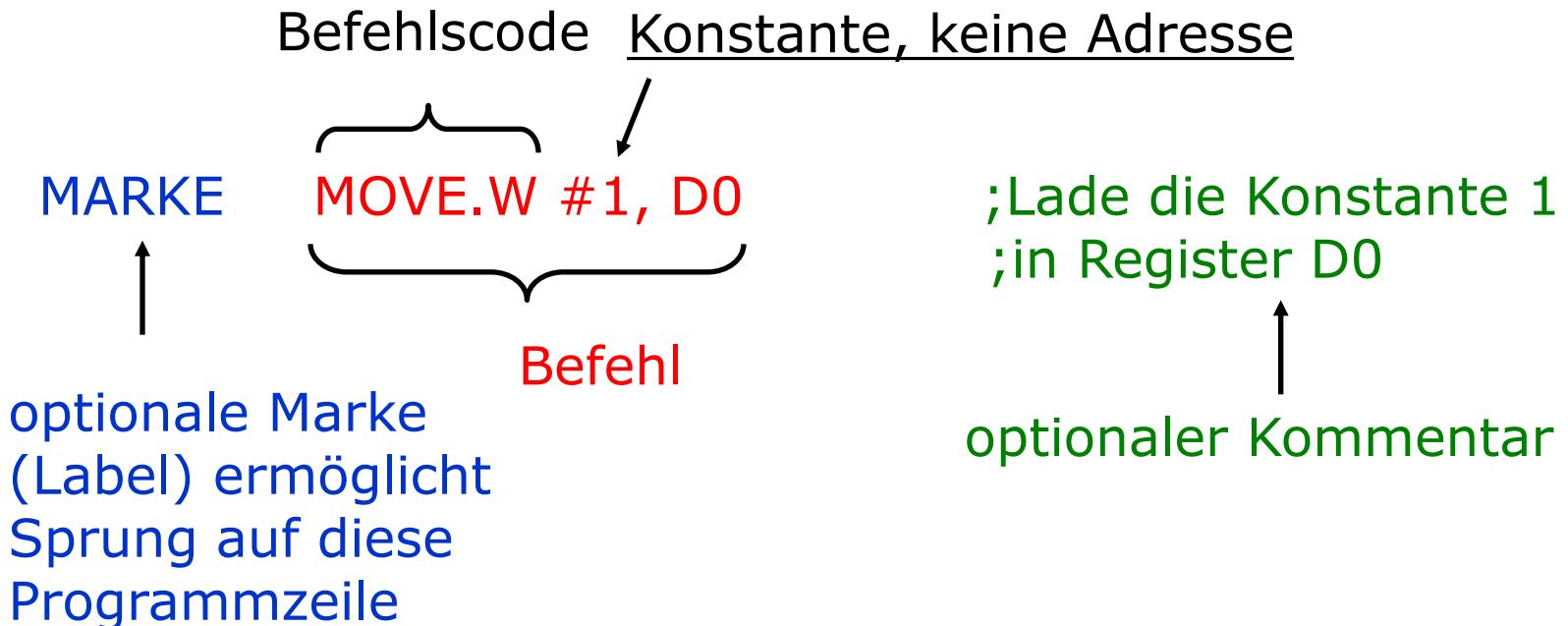
Namensfeld (label)	Operationsfeld (OpCode)	Adreßfeld (Operanden)	Kommentarfeld
symbolische Adressierung einer Programmzeile	symbolischer OpCode	symbolische Adreßangaben	

Aufbau einer Assemblerzeile – Move-Befehl (1)

Allgemeine Befehlssyntax:
MOVE.X QADR ZADR



Aufbau einer Assemblerzeile – Move-Befehl (2)

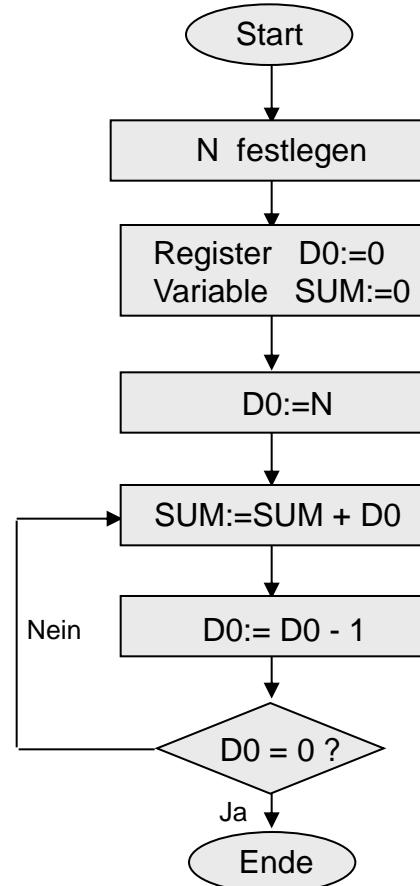


Beispiel Assemblerprogramm (1)

Aufgabe: Berechne die Summe aller ganzen Zahlen von 1 bis N

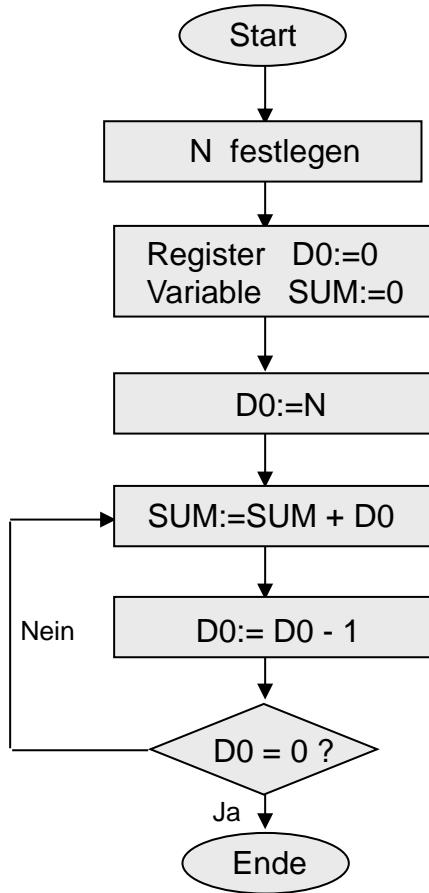
$$\text{SUM} = \sum_{i=1}^N i$$

Pseudocode:
Gib N an
Lösche Inhalt von SUM
For (i = N; i > 0; i --)
 SUM:= SUM + i
END



Beispiel Assemblerprogramm (2)

Aufgabe: Berechne die Summe aller ganzen Zahlen von 1 bis N



Marke	Op-Code	Operanden	Kommentar
	CLR.L	D0	;Lösche D0
	CLR.L	SUM	;Lösche SUM
LOOP	MOVE.W	#N,D0	;Initialisiere D0 mit N
	ADD.L	D0,SUM	;Addiere D0 und SUM
	SUB.W	#1,D0	;Erniedrige D0 um 1
	CMP	#0,D0	;Vergleiche D0 mit 0
	BNE	LOOP	;Verzweige, falls z=0 ist
	END		

Aber wie definiert man die Konstante N und die Variable SUM?

Assemblerdirektiven (Pseudobefehle)

Pseudobefehle sind Befehle für den Assembler

- ↳ Erzeugung von Konstanten
- ↳ Wertzuweisung an Operanden
- ↳ Reservierung von Speicherplatz
- ↳ Steuerung des Assemblierungsvorgangs
- ↳ erzeugen bei der Übersetzung nicht immer Binärkode (im Gegensatz zu Maschinenbefehlen)
- ↳ sie unterliegen dem Format einer Programmzeile

Assemblerdirektiven (Pseudobefehle)

[symbol]	ORG	c	origin of program or data
[symbol]	DS	c	define storage
[symbol]	DC	c	define constant
symbol	EQU	c	equate
	END		end of program

Im Beispiel:

```
        ORG  0
EAREG  EQU  32678
NULL   DC   0
EINS   DC   1
ZEITK  DS   1
```

Assemblerdirektiven (Pseudobefehle)

- Adressfestlegung: **ORG Adresse**
 - Dient der Adresssteuerung für die Assemblierung und legt die Adresse des nachfolgenden Befehls fest.
 - Z.B.: **ORG \$2000**
- Wertzuweisung: **Symbol EQU Ausdruck**
 - Equate (EQU) weist dem *Symbol* im Namensfeld den Wert von „*Ausdruck*“ im Operandenfeld zu.
 - Z.B.: **REG1 EQU 32100; Setze REG1 dem Wert 32100 gleich**

Assemblerdirektiven (Pseudobefehle)

- Konstantenvereinbarungen: *Symbol DC.X Operand*
 - Die DC-Anweisungen (Define Constant) dienen der **Erzeugung von Konstanten**, also der Definition von Speicherzelleninhalten. Der erste Operand belegt den durch den Adresszählerstand (Wert von Symbol) bestimmten Speicherplatz. X kann Byte, Wort oder Langwortformat definieren.

Z.B.: **WERT DC.W \$15**; WERT ist die Adresse der Konstanten 21
(Konstanten, die im Programm referiert werden, werden durch ihre Adresse referiert)
- Endeanweisung: **END**
 - Zeigt dem Assembler das Ende des zu übersetzenden Quellcodes an.

Assemblerdirektiven (Pseudobefehle)

- Reservierung von Speicherplatz: *Symbol DS.X Operand*
 - Mit Define Storage (DS) wird **Speicherplatz im Byte, Wort oder Langwortformat reserviert**, ohne das der Speicherplatz mit definierten Werten belegt wird. Die Anzahl der Speicherplätze gibt „*Operand*“ an. Ein *Symbol* im Namensfeld bezeichnet die Adresse des ersten reservierten Speicherplatzes.
 - Z.B.: **BLOCK DS.L 12**; BLOCK ist die Startadresse des Datenblocks

Beispiel Assemblerprogramm (3)

Marke	Op-Code	Operanden	Kommentar
	ORG	\$1000	
	BRA	START	
N	EQU	100	;Konstante N definiert
SUM	DS.L	1	;Reserviert 1 Langwort
START	CLR.L	D0	;Lösche D0
	CLR.L	SUM	;Lösche SUM
	MOVE.W	#N,D0	;Initialisiere D0 mit N
LOOP	ADD.L	D0,SUM	;Addiere D0 und SUM
	SUB.W	#1,D0	;Erniedrige D0 um 1
	CMP	#0,D0	;Vergleiche D0 mit 0
	BNE	LOOP	;Verzweige, falls z=0 ist
	END		

Listing eines Assemblerprogramm

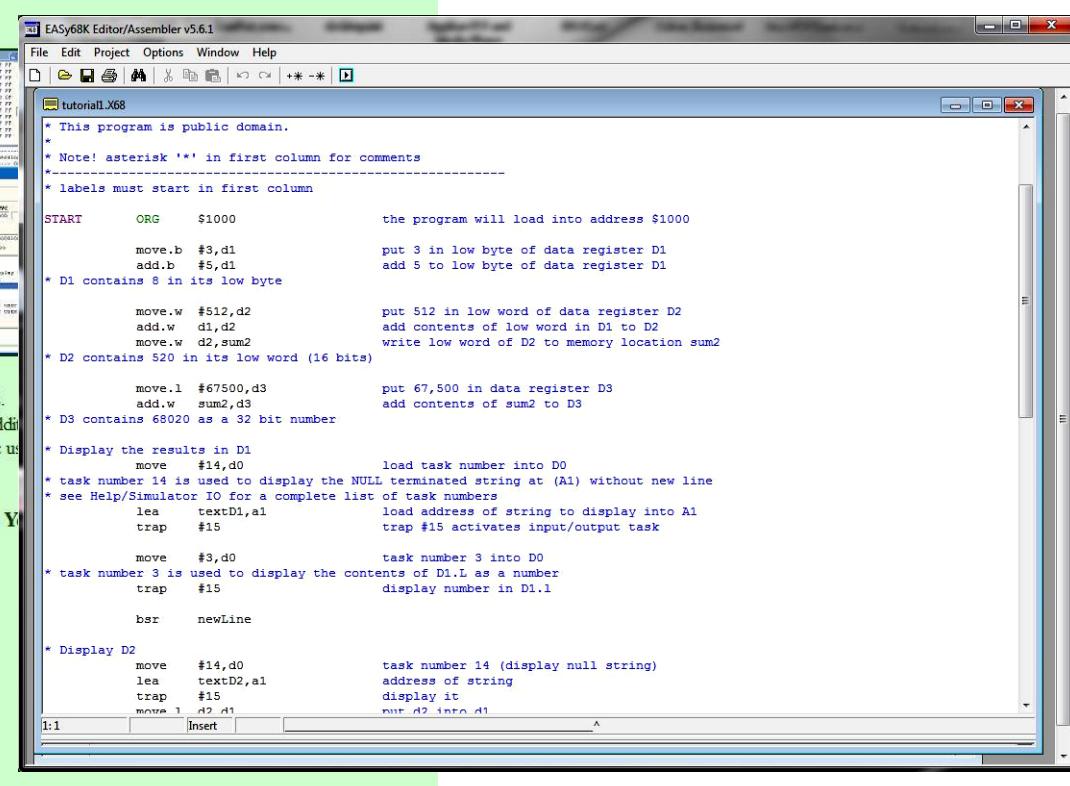
Speicher- stelle	Maschinenbefehl / Speicherinhalt	Marke	Assemblerbefehl	Kommentar
		1	ORG	\$3000
00003000		2	RESULT DS.W	\$0002
00003004	0001	3	ONE	DC.W \$0001
00003006	000A	4	TEN	DC.W \$000A
00003008	4280	5	START	CLR.L D0 ; Loeſche D0
0000300A	4281	6	CLR.L D1	
0000300C	303C 0001	7	MOVE.W ONE,D0	;D0 := 1
00003010	3200	8	MOVE.W D0,D1	;D1 := D0
00003012	5240	9	LOOP	ADDI.W ONE,D0 ;D0 := D0+1
00003014	D240	10	ADD.W D0,D1	;D1 := D1+D0
00003016	0C40 000A	11	CMP.W TEN,D0	;D0=10?
0000301A	66F6	12	BNE LOOP	;Nein, dann Loop
0000301C		13	MOVE.W D1,RESULT	;RESULT := D1
		14	END START	

M68000 – Simulator (Praktikum)

[EASy68K Home Page](http://www.easy68k.com/)

[Editor/Assembler/Simulator for the 68000](http://www.easy68k.com/)
Includes S-Record and Binary file utility.

<http://www.easy68k.com/>



Welcome to the EASy68K home page. EASy68K is a 68000 Structured Assembly Language IDE. EASy68K allows you to edit, assemble and run 68000 programs on a Windows PC or [Wine](#). No additional hardware is required. EASy68K is an open source project distributed under the GNU general public license.

EASy68K, the #1 [68000 Assembler](#) and [68000 Simulator](#) according to [Google](#).

Download

Check the [Forum](#) for latest version information.

Current Build

[SetupEASy68K.exe](#) Executable with installer
[EASy68K.zip](#) Portable App with no installer
[EASy68Ksource.zip](#) Source Code, Requires Borland C++ Builder 6.0 or newer to compile.

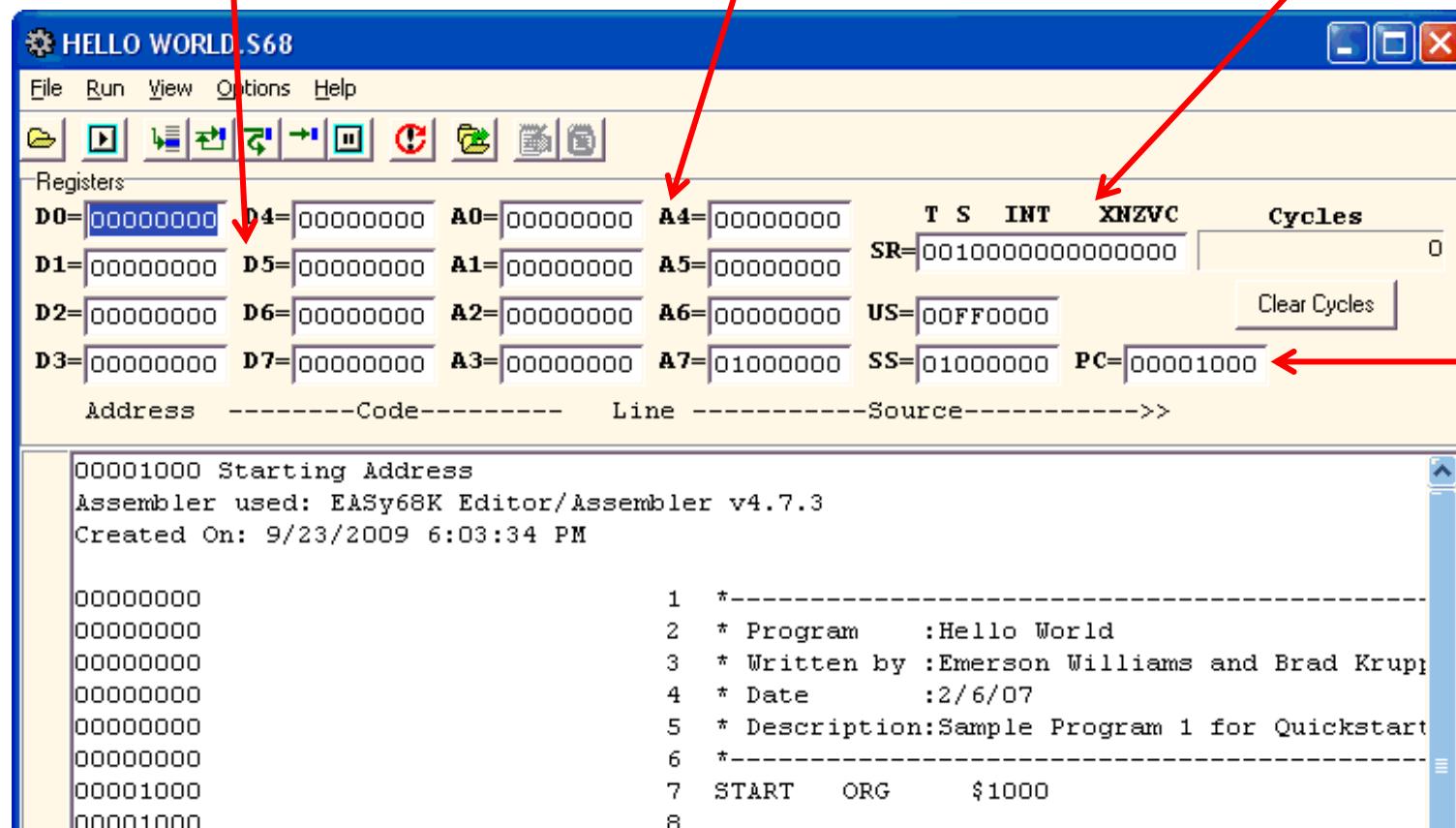
Red arrow pointing to the SetupEASy68K.exe link.

M68000 – Simulator (Praktikum)

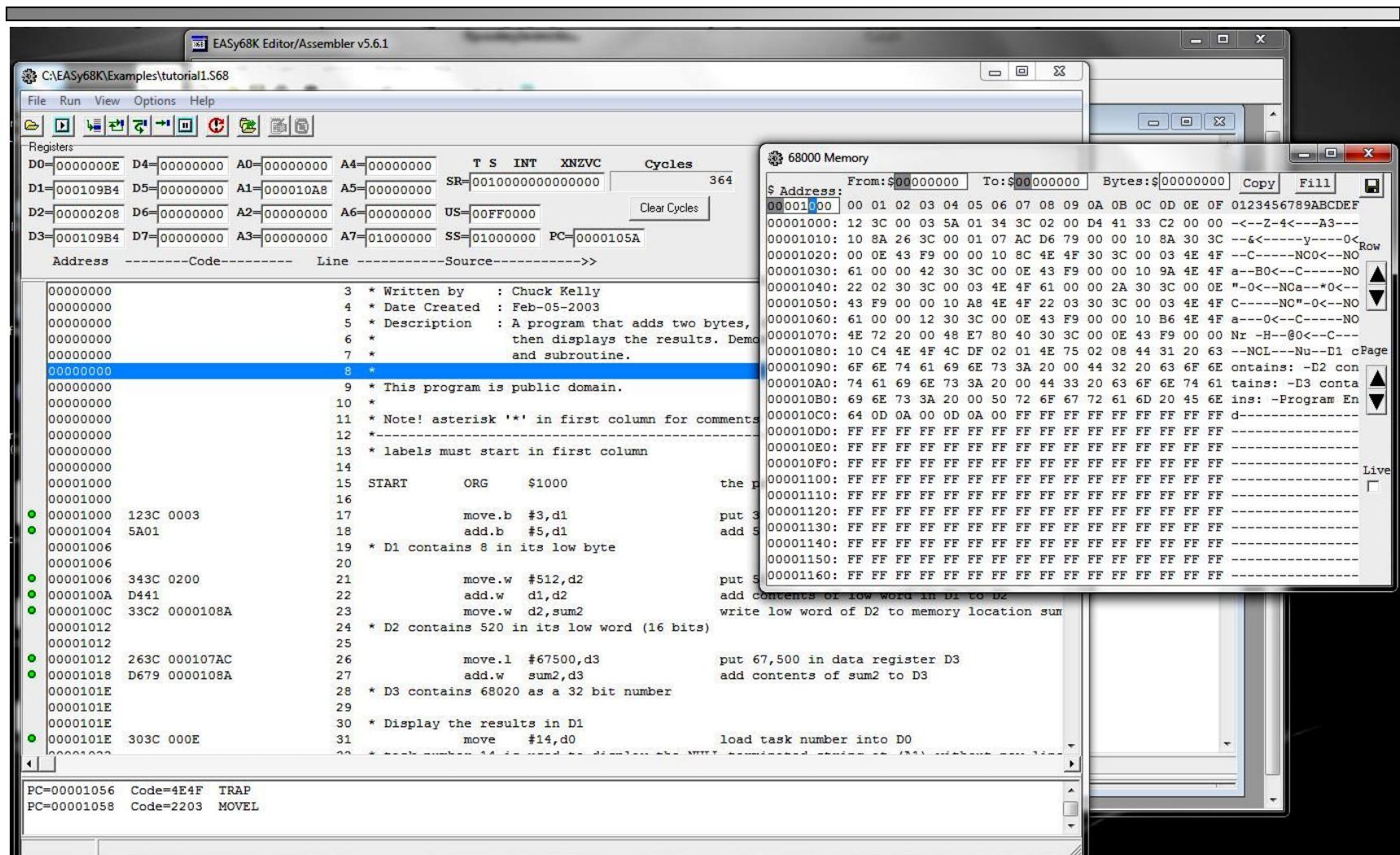
Prozessor-Datenregister

Prozessor-Adressregister

Statusregister



M68000 – Simulator (Praktikum)





Adressierungsarten

M68000

Adressierungsarten

Adressierungsarten:

die verschiedenen Möglichkeiten eines Prozessors die Adresse eines Operanden oder eines Sprungziels im Speicher zu berechnen

Früher: Adresse der Operanden und Sprungziele absolut im Befehl vorgegeben

Nachteile:

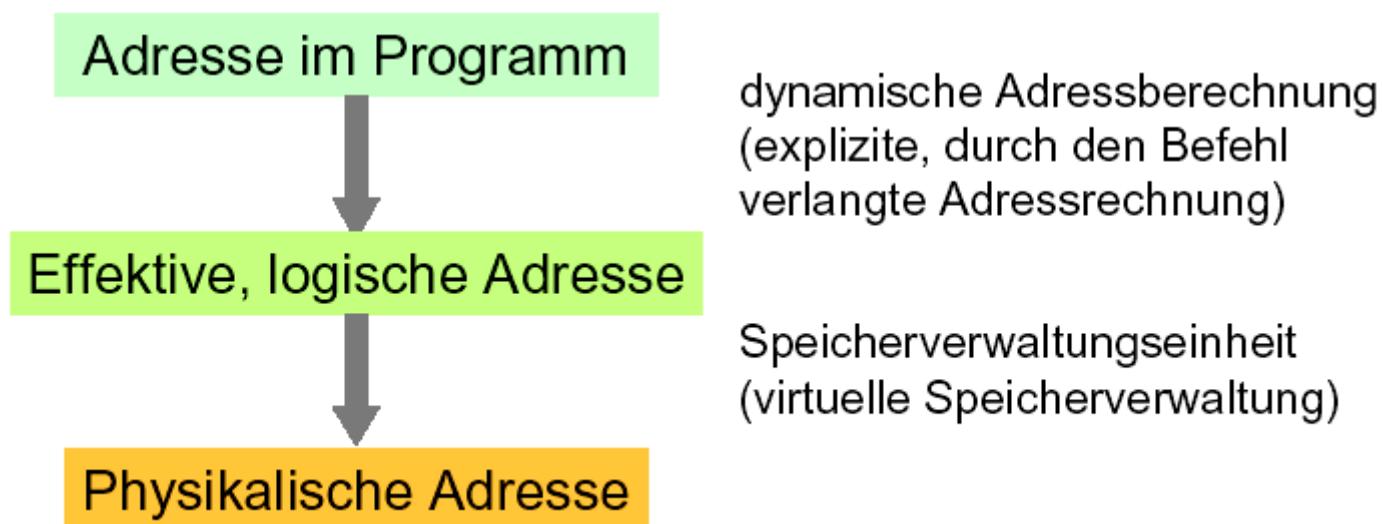
- ⇒ absolute Adressen müssen bereits zur Programmierzeit festgelegt werden ➔ Programme sind lageabhängig im Speicher

Adressierungsarten

Abhilfe:

Adresse wird zur Laufzeit berechnet (dynamische Adressberechnung)

Ablauf der Adressberechnung



Adressierungsarten

Effektive Adresse

- Ist die durch die Adressierungsart spezifizierte Speicheradresse im Hauptspeicher. Sie entsteht im Prozessor nach Ausführung der Adressrechnung.

Gründe für eine dynamische Adressberechnung:

- Die Adresse eines Operanden ergibt sich häufig erst zur Laufzeit aus Berechnungen durch das Programm.
- Die Adresse einer Datenstruktur setzt sich z.B. additiv aus der Anfangsadresse der Datenstruktur und der Distanz innerhalb der Datenstruktur zusammen, wobei die Distanz erst zur Laufzeit bekannt ist.

Registerdirekte Adressierung

MOVE.L D1,D0

Assemblierbefehl
für Transfer eines
Langwortes

Quell-
Register

Ziel-
Register

Vor Befehlsausführung
(Beispiel)

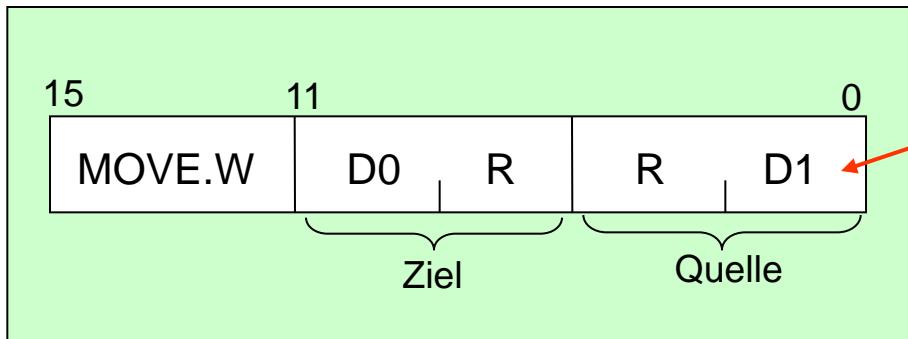
	31		0
D0	12	34	56
D1	AA	BB	CC

Nach Befehlsausführung

	31		0
D0	AA	BB	CC
D1	AA	BB	CC

Abbildung auf Maschinenebene

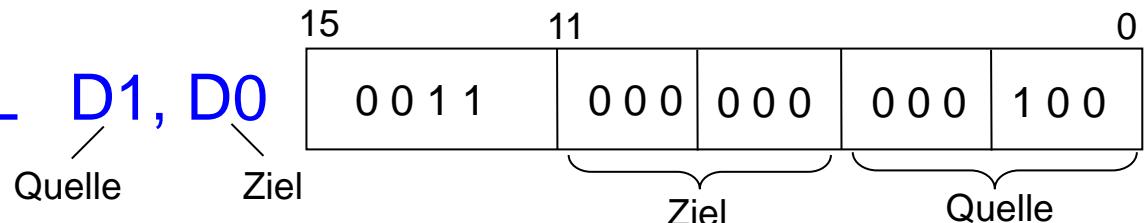
Registerdirekte Adressierung R



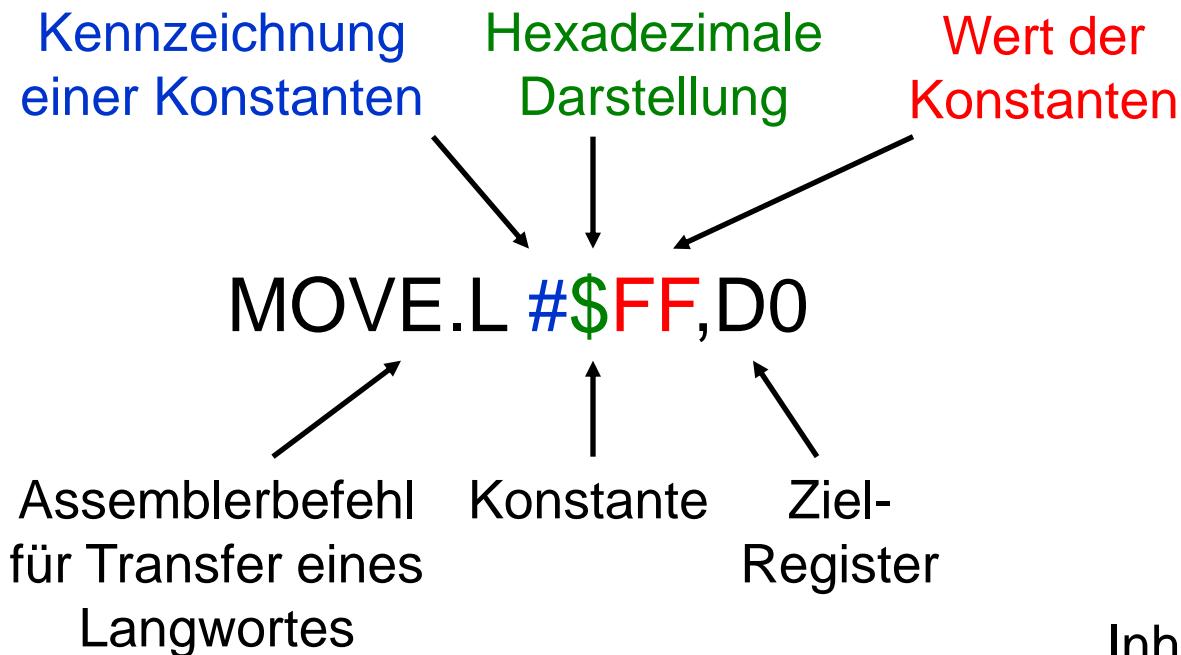
Befehlsformat auf
Maschinenebene

R: spezifiziert die Adressierungsart, hier R = (000)
Bit 12 – 15 codiert Befehl und Format, hier (0010)

Beispiel: MOVE.L D1, D0



Immediate Adressierung (Konstanten)



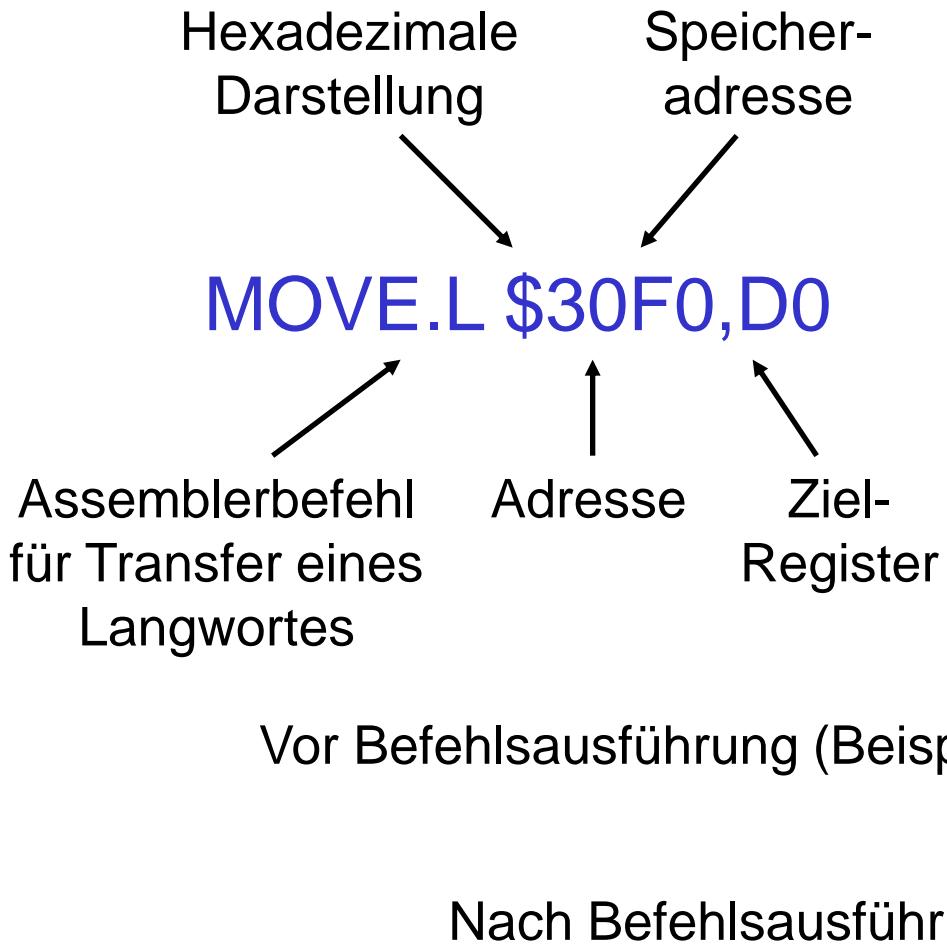
Vor Befehlsausführung (Beispiel)

31	0
12	34

Nach Befehlsausführung

31	0
00	FF

Direkte Adressierung



Hauptspeicherinhalt :

\$30F0	15	56	78	0	\$30F1
\$30F2		9A	BC		\$30F3

Inhalt von D0 :

31	12	34	56	78	0
31	56	78	9A	BC	0

Byte- und Wortadressierung

Datenregister D0

31	0
12	34
56	78

Befehl

MOVE.B D0,\$1000

Speicher

15	0
78	

31	0
12	34
56	78

MOVE.B D0,\$1001

15	0
	78

31	0
12	34
56	78

MOVE.W D0,\$1000

15	0
56	78

Wort- und Langwortadressierung

31	0
12	34
56	78

MOVE.W D0,\$1001

Nicht erlaubt !!!!

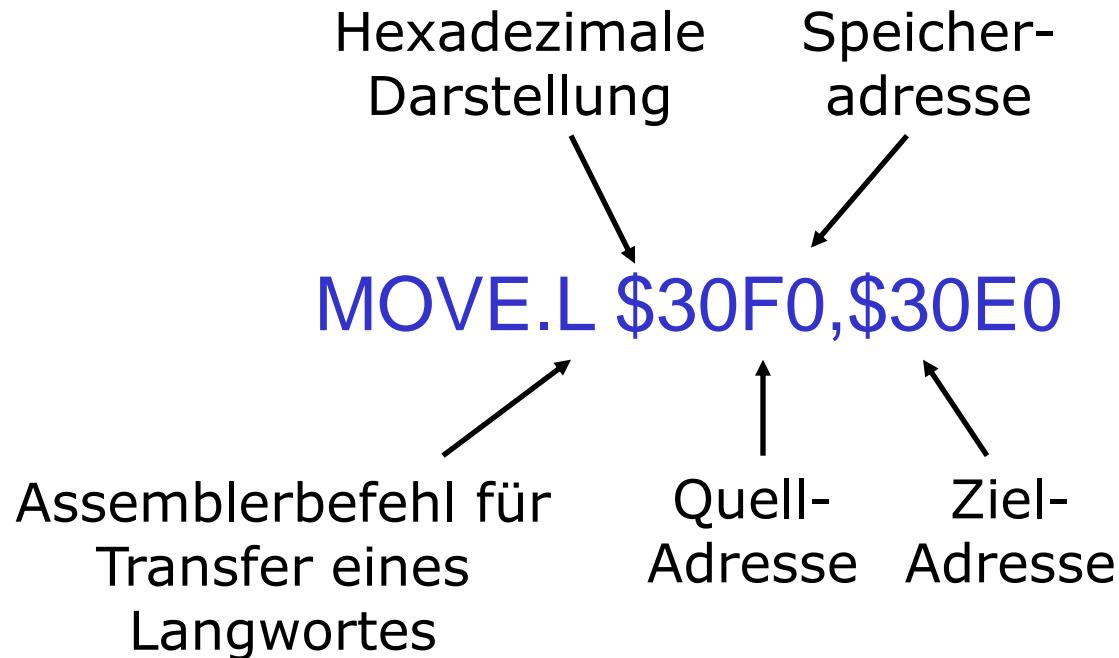
15	0
	\$1001
	\$1002
	\$1003

31	0
12	34
56	78

MOVE.L D0,\$1000

15	0
12	\$1001
56	\$1002
78	\$1003

Direkte Adressierung im Speicher



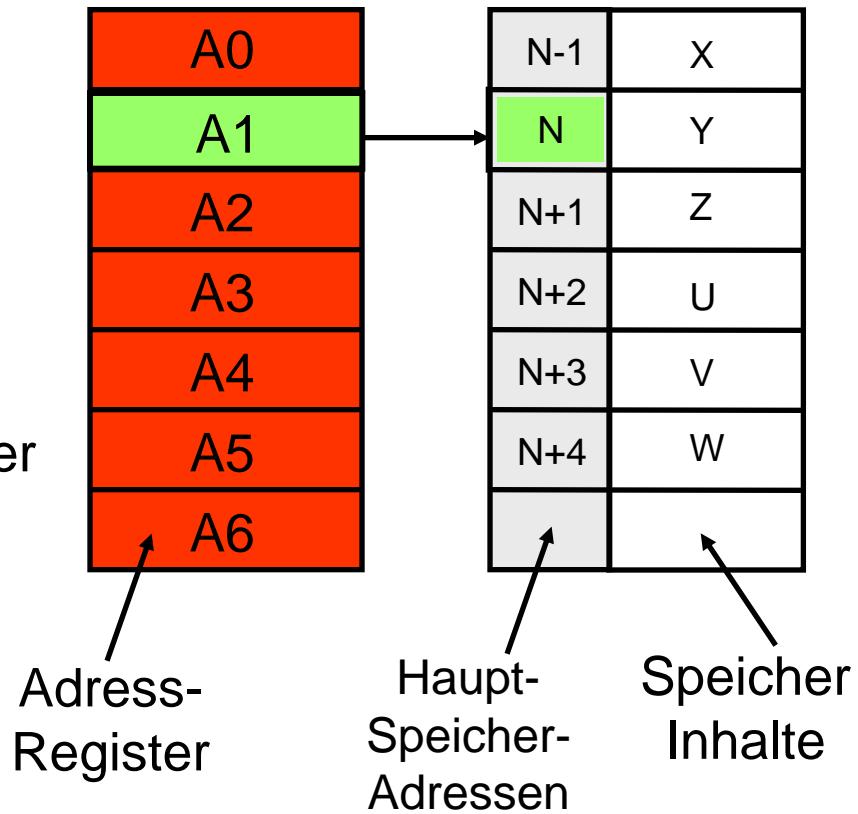
Register-indirekte Adressierung

MOVE.L (A1),D0

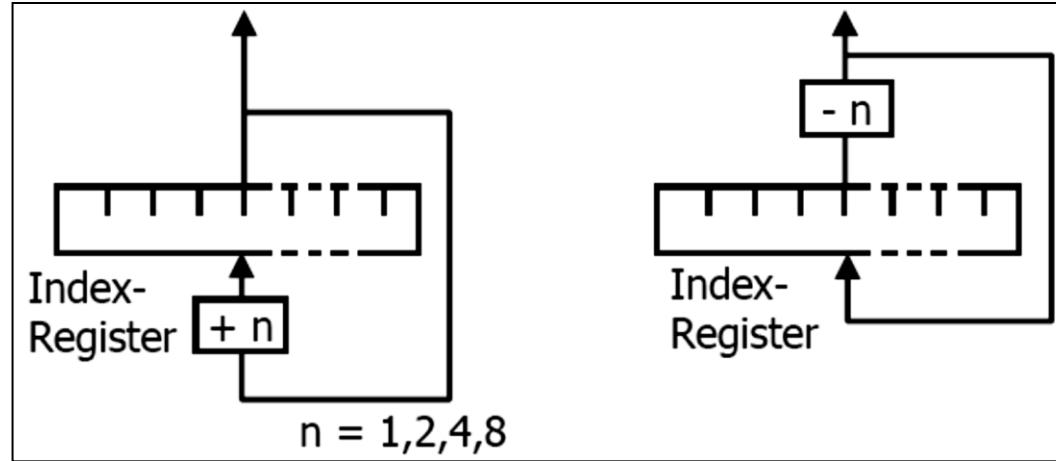
Assemblerbefehl für
Transfer eines
Langwortes

Pointer
auf Quell-
Adresse
(hier auf
Speicheradresse N)

Ziel-
Register



Automatische Modifikation von Registern



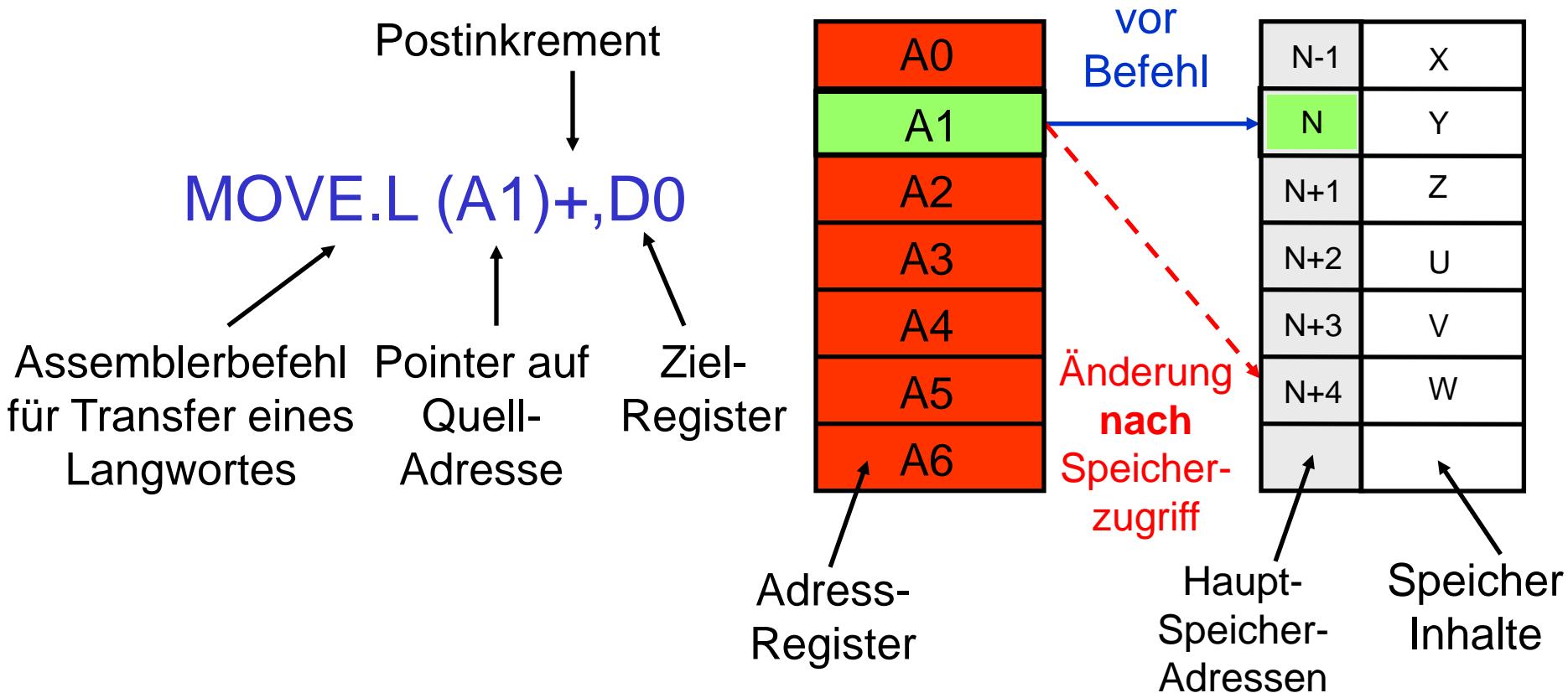
a) Post-Inkrement:

automatische Erhöhung des Registerwerts um $+n$ nach Adressierung einer Speicherzelle

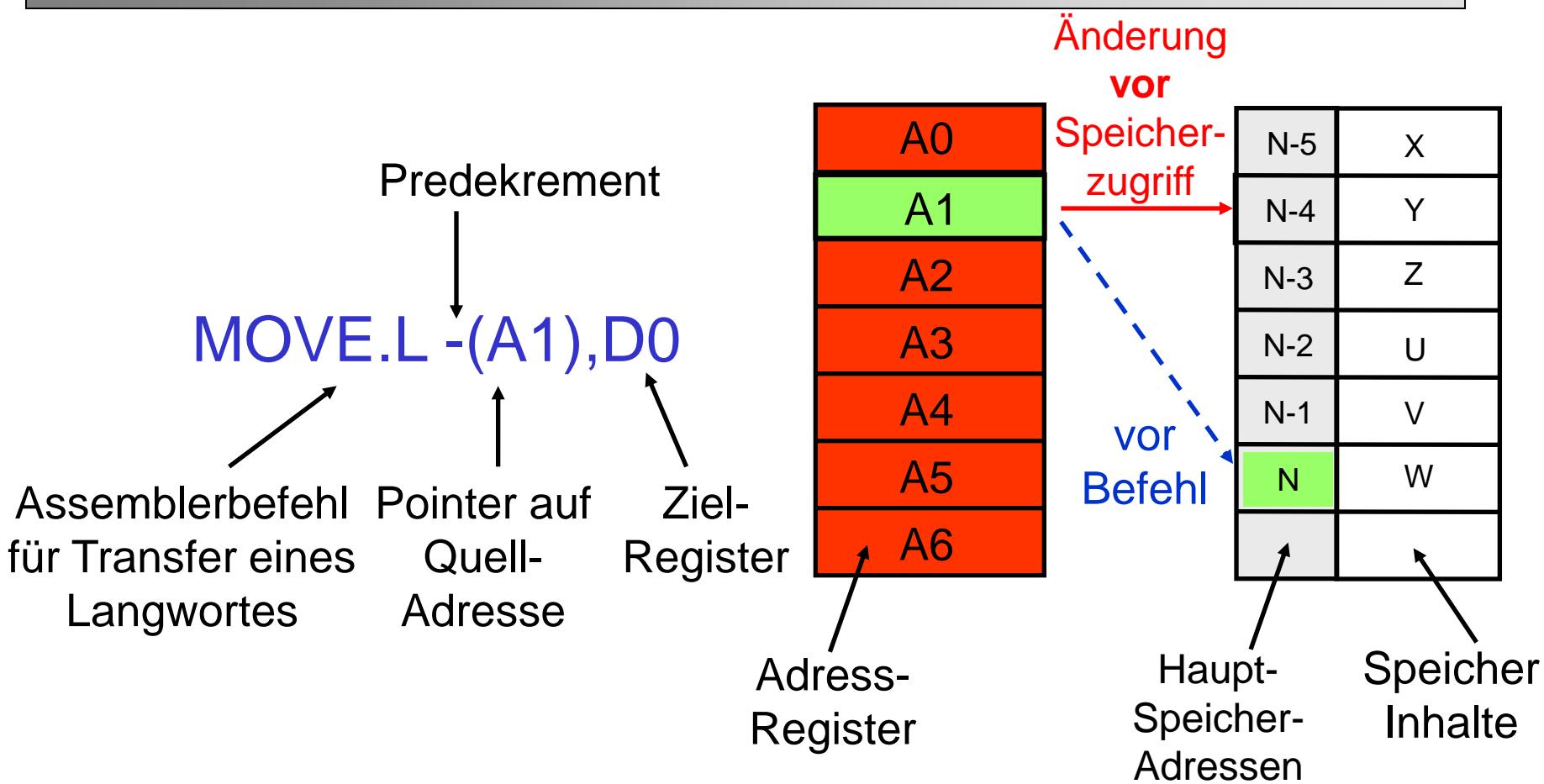
b) Pre-Dekrement:

automatische Erniedrigung des Registerwerts um $-n$ vor Adressierung einer Speicherzelle

Register-indirekte Adressierung mit Post-Inkrement



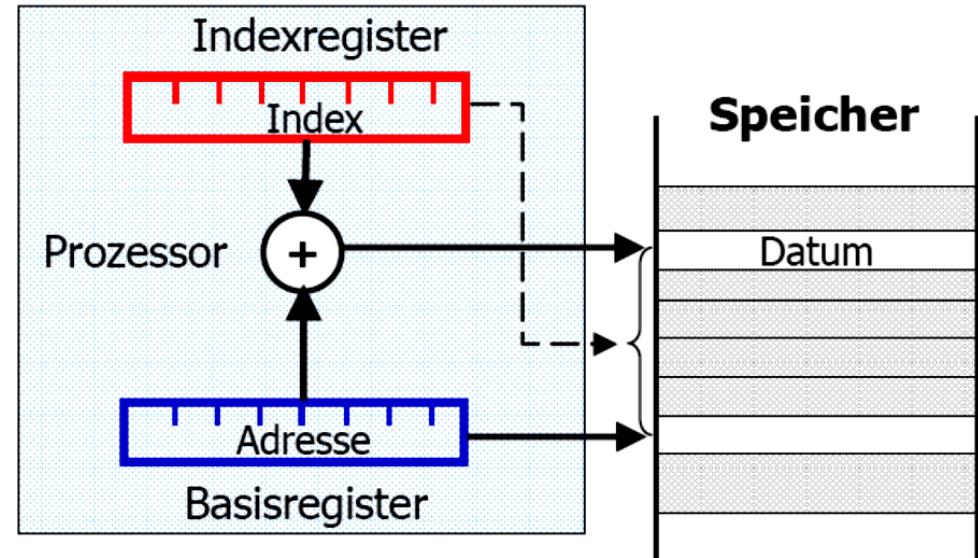
Register-indirekte Adressierung mit Pre-Dekrement



Allgemein: Funktion von Basis- und Indexregister

Basisregister enthält die Anfangsadresse eines Speicherbereichs.

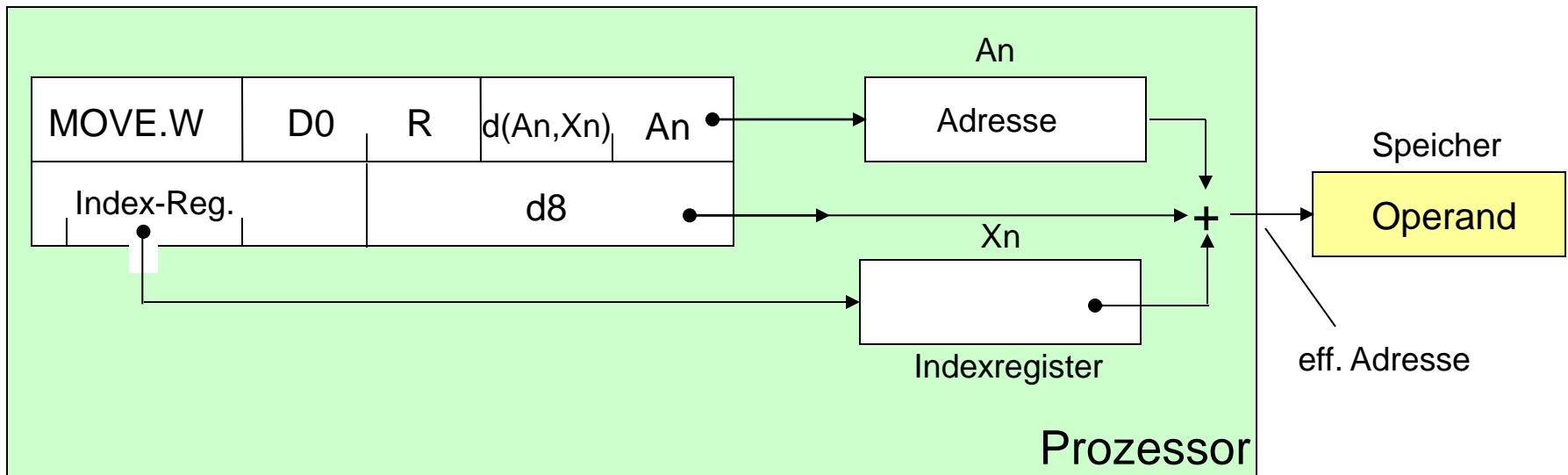
Diese bleibt während der Bearbeitung des Speicherbereichs unverändert.



Indexregister enthält eine Distanz (Offset, Displacement) zu einer Basisadresse und dient zur Auswahl eines bestimmten Datums des Speicherbereichs.

Adressierungsarten beim M68000

Registerindirekte Adressierung mit Index d(An, Xn.x)



Effektive Adresse: $(A_n) + (X_n) + d_8$

Allgemein: $MOVE.x \quad d8 \quad (A_n, X_n.x), <EA>z$

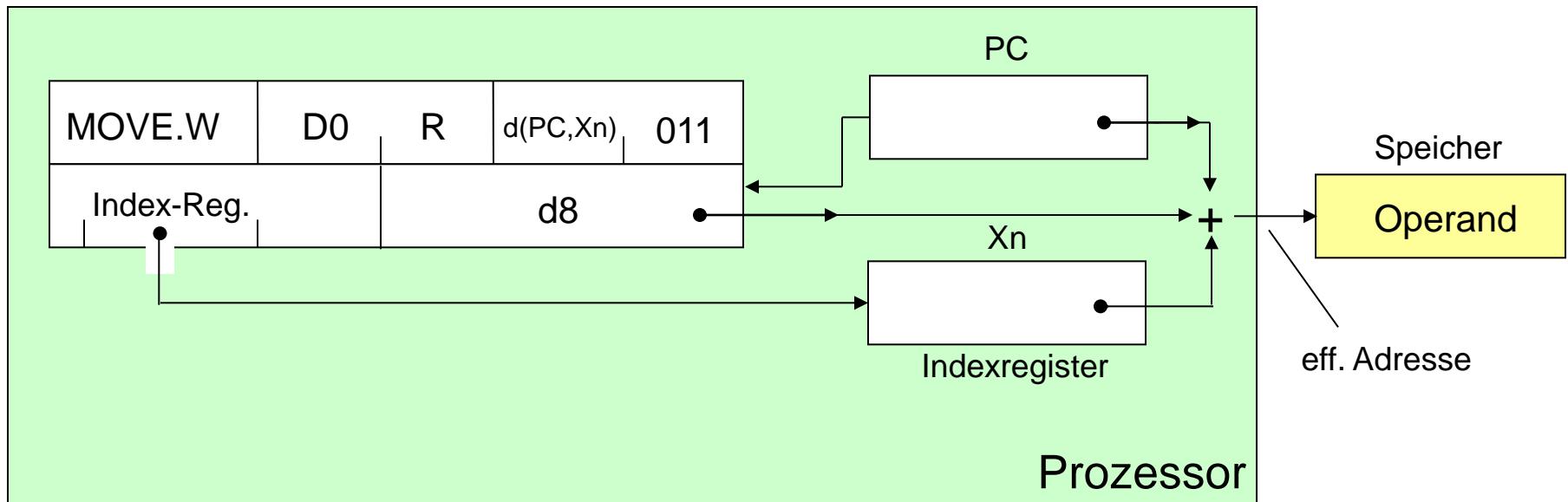
Beispiel: **MOVE.W 4(A3,D1.W), D0**

Programmzähler relative Adressierung



Adressierungsarten beim M68000

Programmzählerrelative Adressierung mit Verschiebung und Index $d(PC, Xn.x)$



Effektive Adresse: $(PC) + (Xn) + d8$

d8: sign extended

Allgemein: $MOVE.x \quad d8 \quad (PC, Xn.x), <EA>z$

Beispiel: **MOVE.W 4(PC,A3.W), D0**

Befehlsausführungszeiten des M68000

- Befehlsausführungszeiten
 - In der folgenden Tabelle sind auszugsweise die Befehlsausführungszeiten für den Befehl MOVE angegeben, in Vielfachen von externen Taktperioden (siehe Motorola-M68000-Handbuch).
 - Unter der Annahme, dass der Speicher schnell genug ist, benötigt ein **Schreib- und Lesezyklus jeweils 4 Taktperioden (Clocks)**. Die erste Zahl eines jeden Werts bezeichnet die Gesamtausführungszeit in Taktperioden (CLKs). Die in Klammern angegebenen Werte (r/w) beziehen sich auf die Anzahl der Lese (r)- und Schreib (w)-Zyklen.
 - Beispielsweise enthält ein angegebener Wert von 18 (3/1) drei Lesezyklen und einen Schreibzyklus und demzufolge werden 2 CLKs für prozessorinterne Funktionen benötigt.

Befehlsausführungszeiten des M68000

Befehlsausführungszeiten beim MOVE – Befehl (.W)

Source	Destination								
	Dn	An	(An)	(An)+	-(An)	(d16, An)	(d8, An, Xn)*	(xxx).W	(xxx).L
Dn	4(1/0)	4(1/0)	8(1/1)	8(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
An	4(1/0)	4(1/0)	8(1/1)	8(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
(An)	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
(An)+	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
-(An)	10(2/0)	10(2/0)	14(2/1)	14(2/1)	14(2/1)	18(3/1)	20(3/1)	18(3/1)	22(4/1)
(d 16, An)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
(d 8, An, Xn)*	14(3/0)	14(3/0)	18(3/1)	18(3/1)	18(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
(xxx).W	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
(xxx).L	16(4/0)	16(4/0)	20(4/1)	20(4/1)	20(4/1)	24(5/1)	26(5/1)	24(5/1)	28(6/1)
(d 16, PC)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
(d 8, PC, Xn)*	14(3/0)	14(3/0)	18(3/1)	18(3/1)	18(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
#<data>	8(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)

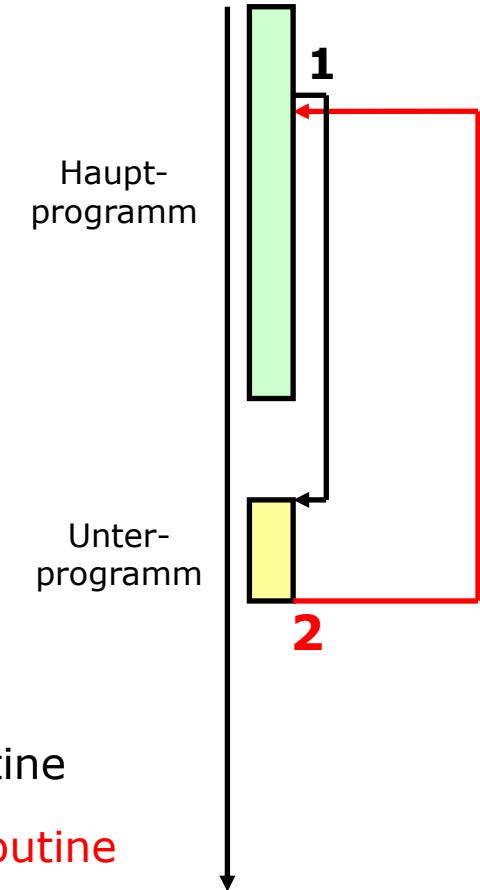
*The size of the index register (Xn) does not affect execution time.

Weitere Befehlsausführungszeiten können aus dem Prozessor Handbuch von Motorola entnommen werden.

Unterprogrammtechnik

Programmiertechnik - Unterprogramme

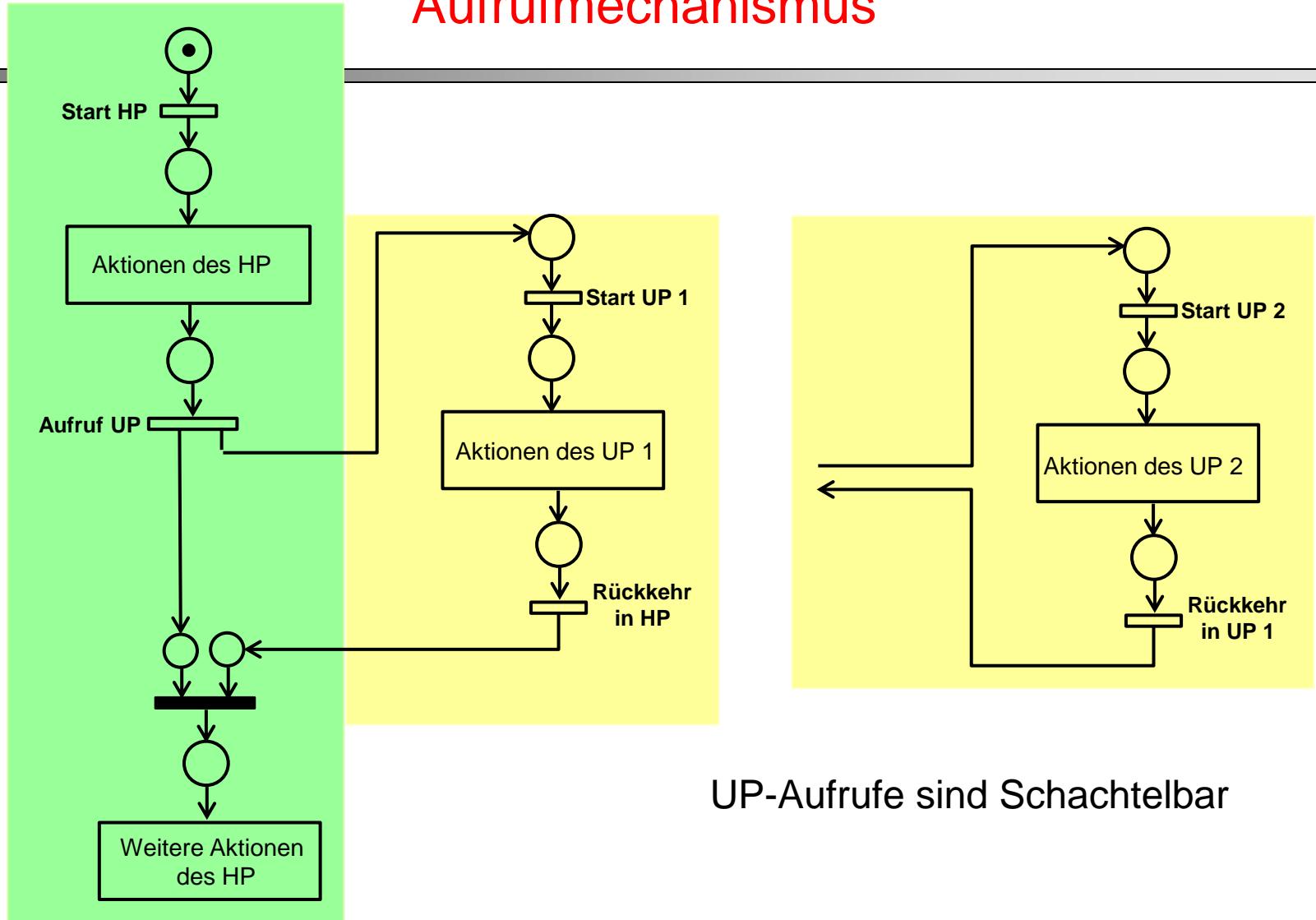
- Ein **Unterprogramm (subroutine)** ist ein eigenständiges, ablauffähiges Programm, das von einem übergeordneten Programm aus beliebig oft aufgerufen werden kann. Das Unterprogramm erscheint nur einmal im Speicher.



Unterprogramm-Mechanismus

- Aufrufmechanismus:
 - *Branch to subroutine:* $BSR < eA >$ (1)
- Durch den Unterprogrammaufruf wird der PC mit der effektiven Adresse des Unterprogramms geladen und die Programmabwicklung dort fortgesetzt. Der alte Wert des PC wird dabei **automatisch auf den Stack** (USP) gerettet.
- Den Abschluss eines Unterprogramms bildet immer der Befehl:
 - *Return from subroutine:* RTS (2)
 - RTS lädt den PC wieder mit der Rückkehradresse vom Stack

Aufrufmechanismus



UP-Aufrufe sind Schachtelbar

Einfacher Unterprogramm-Aufruf

Hauptprogramm

```
MOVE.W #5,D7  
BSR UP1  
MOVE.W D0,D1
```

Unterprogramm

UP1	MOVE.W	SR,-(A7)
	MOVEM.W	D0-D3,-(A7)
Befehle des Unterprogramms	(Übergabewert in D7)	
	MOVEM.W	(A7)+,D0-D3
	MOVE.W	(A7)+,CCR
	RTS	

Beispiel in C

```
void Punkte_zeichnen(int anzahl) {  
    for (int i = 0; i < anzahl; i++) {  
        putchar('.');  
    }  
}
```

Beispiel eines Funktionsaufrufs in C:

```
Punkte_zeichnen(5);
```

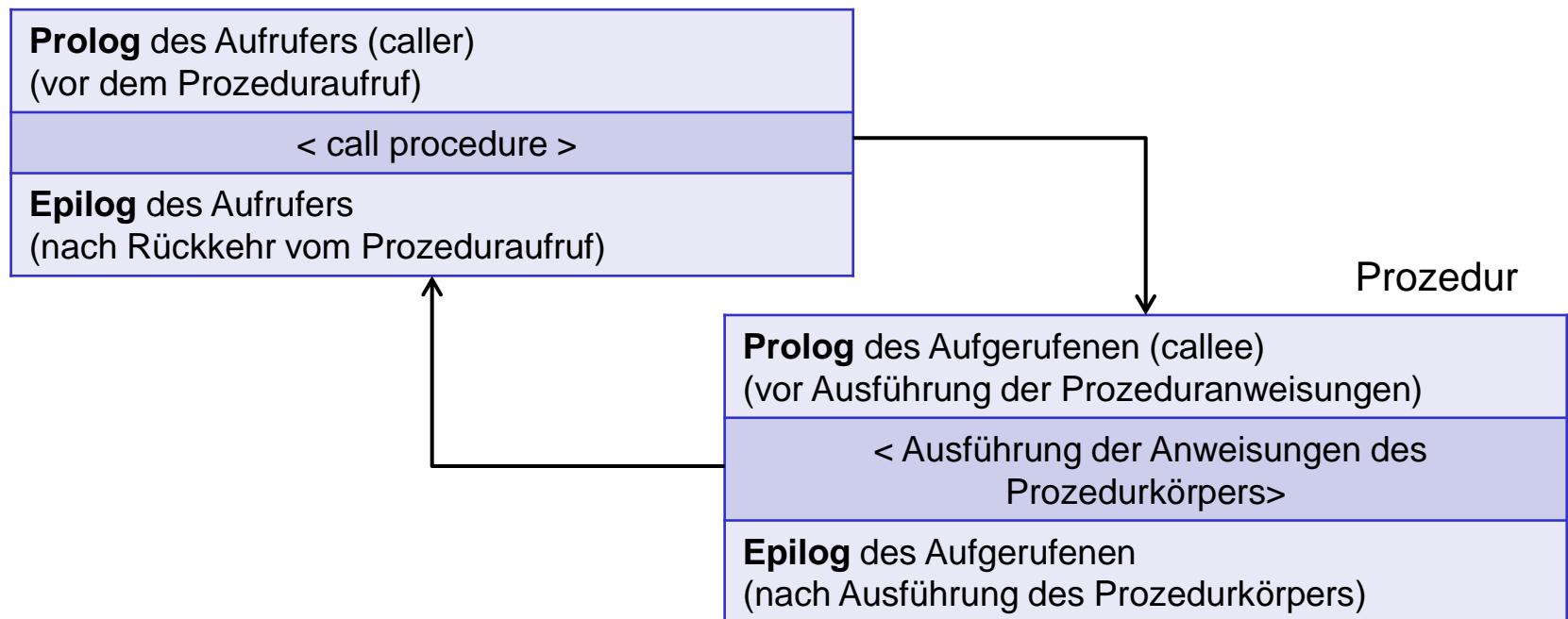
Quelle: Wikipedia

Unterprogramme - Parameterübergabe

- Parameterübergabe
 - Art des Parameters
 - Call by value (Wertübergabe)
 - Call by reference (Adressübergabe)
 - Ort der Übergabe
 - Register
 - Stack
 - Datenbereich
- Unterprogrammtypen:
 - geschachtelte Unterprogramme
 - rekursive Unterprogramme

Prozedurrahmen und Aufruf-Konventionen

- Speicherung aller lokalen Variablen und Daten einer Prozedur, die nicht in die verfügbaren Register passen, auf dem Stack.
- Das Stacksegment mit allen Daten eines Prozeduraufrufs heißt Prozedurrahmen (stack frame)
- Ein Rahmenzeiger f_p zeigt auf den Anfang des Prozedurrahmens.



Prozeduraufruf und -rücksprung

Vor einem Prozederaufruf

1. Sicherung benötigter Register
2. Bereitstellung der Argumente für Prozederaufruf
3. Prozederaufruf mittel BSR-Befehls



Aktionen am Beginn einer Prozedur

1. Sicherung aller Register, die vor Rücksprung restauriert werden müssen
2. Einrichten eines Framepointers fp

Aktionen am Ende einer Prozedur

1. Rückgabewerte speichern
2. Restaurierung aller Register, die zu Beginn gesichert wurden
3. Anpassen des Stackpointers
4. Rücksprung mittels RTS

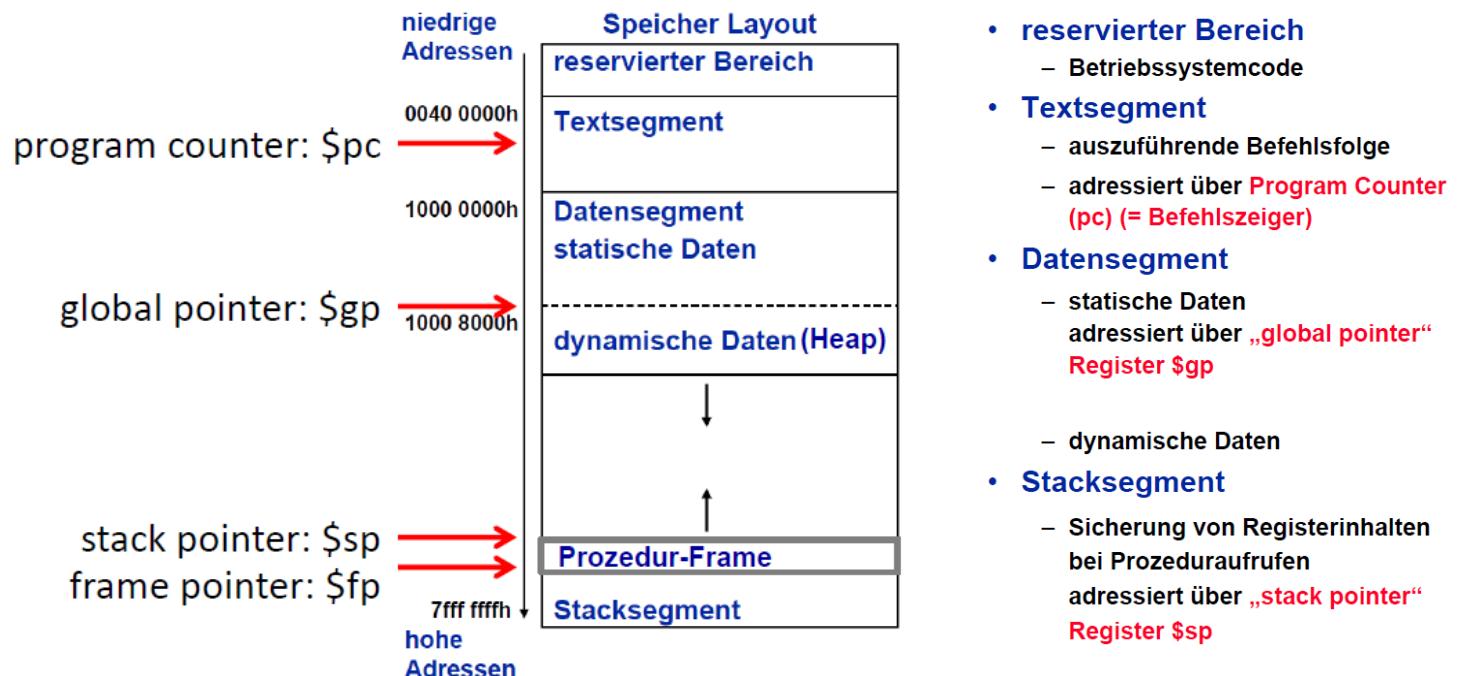
Nach einem Prozederaufruf

1. Rückspeicherung vorher gesicherter Register
2. Anpassen des Stackpointers sp



Mögliche Konvention einer Speicheraufteilung

- Beispiel: Der Speicher kann folgendermaßen strukturiert sein:
Das *Textsegment* beinhaltet den Programm Code. Das *Datensegment* enthält einen Teil für statische Daten wie Konstanten und Variablen fester Größe sowie einen Teil für dynamische Daten, der *Heap* genannt wird. Dynamische Daten sind alle Daten, für die während der Laufzeit Speicher reserviert wird (`malloc`, `new`), wie z.B. Listen oder Arrays mit variabler Größe. Das Stack-Segment wird für die Sicherung von Registern bei Prozeduraufrufen sowie für lokale Variablen genutzt.



Vom Quellcode zum ausführbaren Programm

- **Assembler:**

Programm, das einen Quellcode in Assemblersprache in eindeutiger Weise in Maschinensprache übersetzt

- **Objektcode:** Repräsentation eines Maschinenprogramms,

in dem noch ungelöste Referenzen auf externe Unterprogramme oder Speicherbereiche enthalten sind

- Zusätzlich können im Objektcode Informationen

enthalten sein, die die Fehlersuche mit einem **Debugger** ermöglichen

- **Binder (Linker):** Programm, das die ungelösten

Referenzen mehrere Objektcode-Module auflöst und sie zu einem ausführbaren Programm verbindet

Technik der Übersetzung von Quellprogrammen

Compiler

- Übersetzung von Hochsprachen in Binärcode
- Hochsprachenprogrammierung wird verwendet,
 - da die Programmereffizienz hier ungleich höher ist als bei der Verwendung der Maschinensprache,
 - da Programme in Hochsprachen (mehr oder weniger) maschinenunabhängig sind.
- Moderne Compiler erzeugen Assemblercode, der so gut optimiert ist, dass er manuell kaum besser erzeugt werden könnte.
- Manuelle Assembler-Programmierung
 - zur Optimierung spezieller Codesequenzen
 - z.B. um Reaktionszeit einer Bremse zu garantieren
 - wenn es keinen (guten) Compiler gibt
- Während höhere Programmiersprachen (weitestgehend) maschinenunabhängig sind, muss für jede Zielarchitektur ein spezieller Compiler bereit gestellt werden.

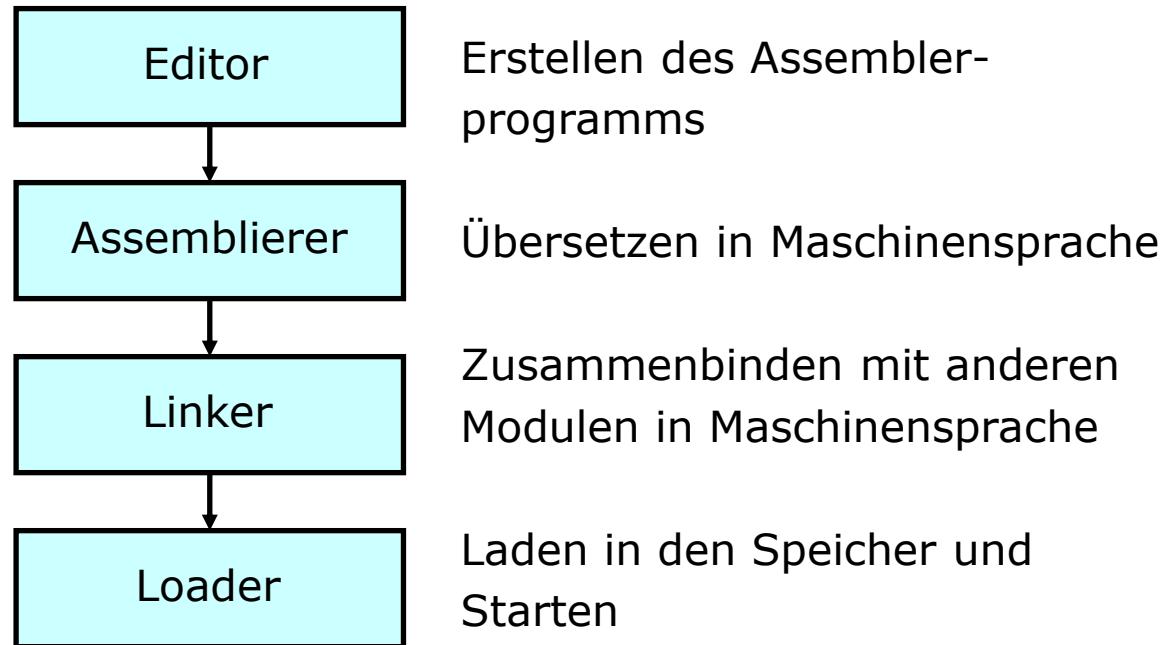
Ablauf der Assemblierung

Programm-Darstellung

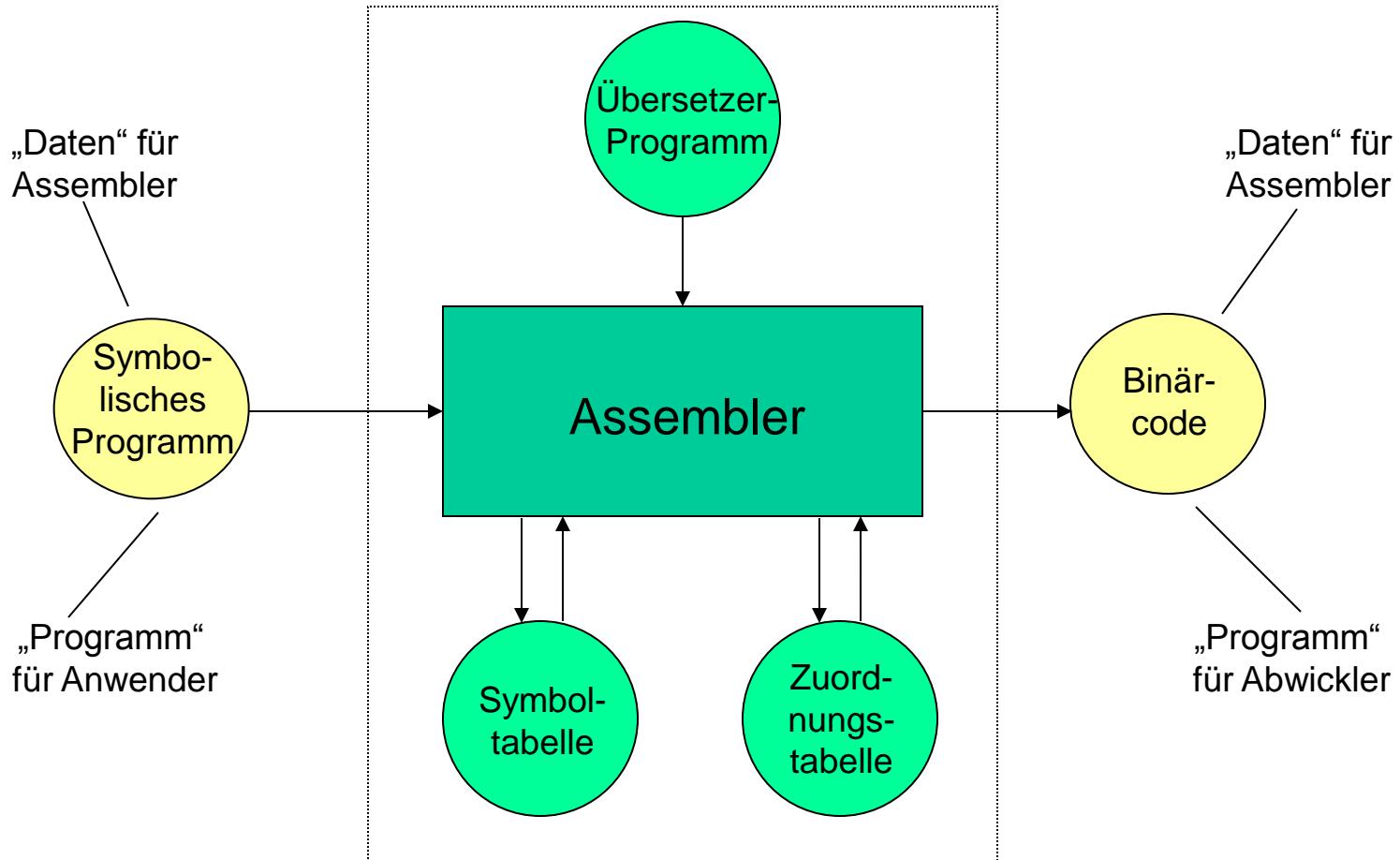
- Symbolische Darstellung
- Maschinencode-Darstellung

Programm-Übersetzung

- Assemblersprache
- Assembleranweisungen
- Assemblierung



Technik der Assemblierung



Technik der Assemblierung

Der Assembler verwendet zur Programmübersetzung zwei Tabellen:

- **Zuordnungstabelle**
 - Enthält die Binärcodes für die mnemotechnischen Operationscodewörter und für die Register des Prozessors

Zum Beispiel

Symbol	Binärcode
MOVE	0000 0001
SUB	0000 0010
CMP	0010 0001
BNE	0000 1100
JMP	0000 0101

Technik der Assemblierung

- **Symboltabelle**
 - enthält die Zuordnung der Symbole zu ihren Werten

Die Ersetzung symbolischer Adressen durch numerische Adressen ergibt sich aus der Lage des Programms im Hauptspeicher.

Zur Feststellung der jeweiligen Programmzeilenadresse verwendet der Assembler den **Zuordnungszähler**, mit dessen Hilfe sich auch Adressdistanzen berechnen lassen.

Zum Beispiel

Symbol	Adresse					
					Dual	Dezimal
MARKE1	0000	0000	0000	1000	8	0008
MARKE2	0000	0000	0000	1010	10	000A
ZEITK	0000	0000	0001	0111	23	0017
NULL	0000	0000	0001	1000	24	0018
EINS	0000	0000	0001	1001	25	0019
EAREG	1000	0000	0000	0000	32768	8000

Beispiel zur Assemblierung

- Auszug aus einem Programmbeispiel

Adresse	Maschinencode	Anzahl Bytes	Assembler-Schreibweise		
0	4241	2	CLR.W	D1	
2	4242	2	CLR.W	D2	
4	10387D64	4	MARKE1:	MOVE.B SM,D0	
8	E208	2		LSR.B #1,D0	
A	64F8	2		BCC MARKE1	
C	10387D65	4		MOVE.B DM,D0	

- Statusregister SM liegt im Adressraum auf Adresse 32100 (\$7D64)
- Der Adresswert des Zuordnungszählers hängt von der Anzahl der Bytes pro Befehl ab.
- In Zeile 10 (Verzweigungsbefehl) muss, wenn C=0 ist, relativ zum Programmzähler, um 8 Bytes zurück gesprungen werden.

Beispiel zur Assemblierung

Programmauszug:

SM	EQU	32100
DM	EQU	32101
SK	EQU	32200
DK	EQU	32201

Die **Symboltabelle** zeigt den Stand nach der Übersetzung des Befehls BEQ MARKE2.

			Symbol	Adreßwert	verwendet	definiert
			SM	32100	*	*
START	CLR.L	D1				
	CLR.L	D2	DM	32101	*	*
MARKE1	MOVE.B	SM,D0	SK	32200	*	*
	LSR.B	#1,D0				
	BCC	MARKE1	DK	32201	*	*
	MOVE.B	DM,D0	MARKE1	0004	*	*
	ANDI.B	#1,D0				
	OR.B	D0,D1	MARKE2		*	
	ADDQ.B	#1,D2				
	CMP1.B	#B,D2				
	BEQ	MARKE2				

Problem: Vorwärtsadressbezug

Arbeitsweise des Assemblers (Summary)

- **Aufgaben des Assemblers**
 - Auflösen von Pseudo-Instruktionen und Markos
 - Auflösen von symbolischen Adressen (Labels)
- **Globale und lokale Labels**
 - lokal: Label in gleicher Datei/Code-Segment
 - global, extern: Label in referenzierter Datei/Code-Segment
- **Forward Reference Problem**
 - Sprünge zu einem Label, das sich „weiter vorne“ im Code befindet
 - Assembler kann das Label nicht auflösen, ohne die „Nummer“ der Code-Zeile des Labels zu kennen
- **Der Assembler braucht zwei Durchgänge durch den Assembler-Code, aufgrund von Vorwärts-Adressverzweigungen.**

Arbeitsweise des Assemblers (Summary)

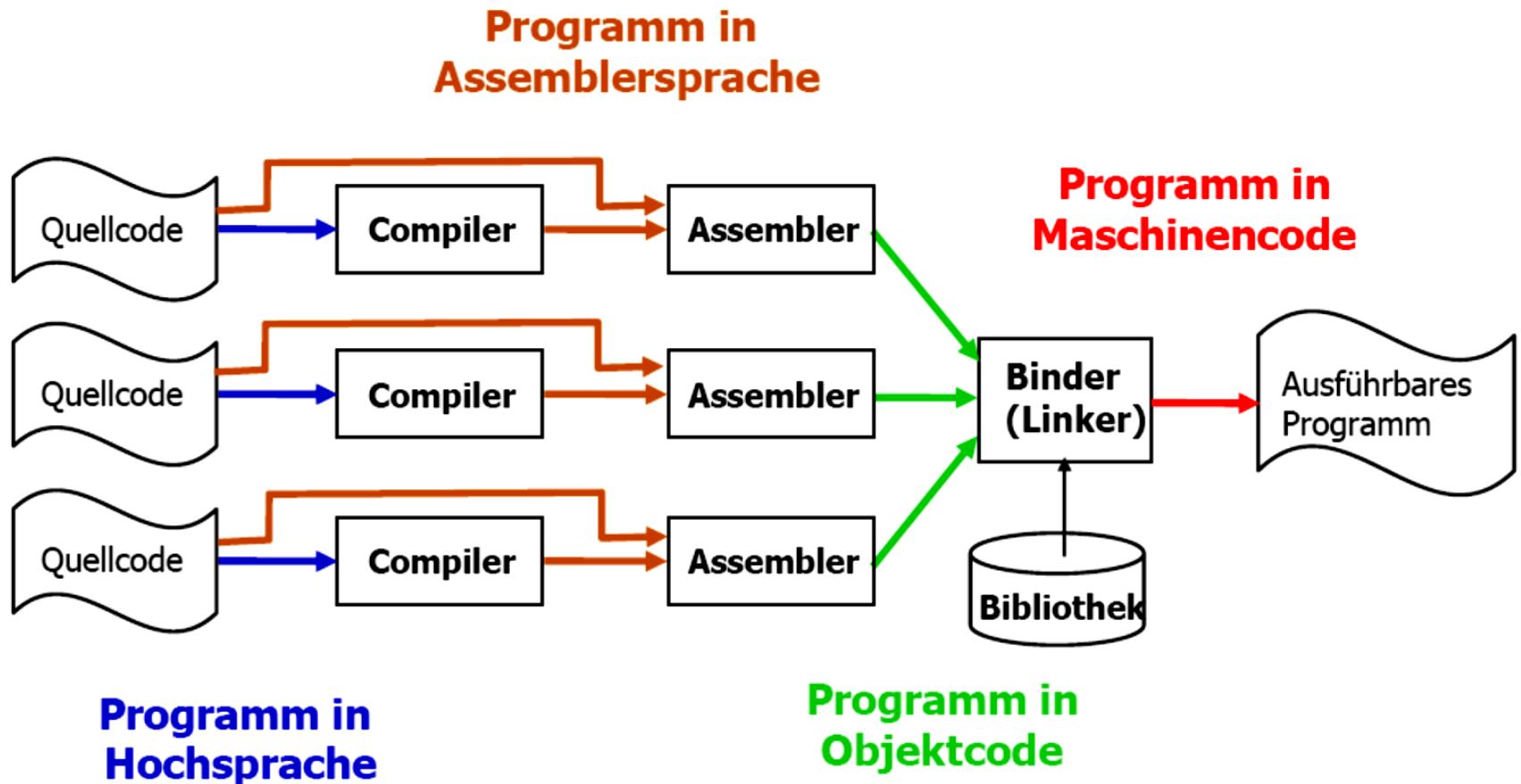
- Erster Durchlauf (First Pass)
 - Auflösung von Pseudo-Instruktionen und Makros
 - Produziert Symboltabelle: Zuordnung von Labels zu Speicheradressen/Code-Zeilen
 - Speicheradressen von „statischen“ Daten werden bestimmt
 - Bei Erreichen der END-Anweisung müssen alle Symbole definiert sein.
- Zweiter Durchlauf (Second Pass)
 - Produziert Code in Maschinensprache
 - Ersetzt lokale Label durch Speicheradressen

Arbeitsweise des Assemblers (Summary)

- Objektfiles enthalten alle Informationen, die für die weitere Bearbeitung erforderlich sind. Beispiel: Unix-Objektfiles
- Struktur
 - Header: Beschreibt die Größe und Position der übrigen Teile des Objektfiles
- Code und Daten
 - Textsegment: Code in Maschinensprache, kann nicht aufgelöste Referenzen (externe Labels) enthalten
 - Datensegment: Statische Daten, die während der Programmlaufzeit allokiert werden, dynamische Daten, die während der Programmlaufzeit ihre Größe verändern
- Informationen für den Linker
 - Relocationsinformation: Identifikation von Instruktionen und Daten, die von absoluten Speicheradressen abhängig sind
 - Symboltabelle: noch undefinierte Labels (z.B. externe Sprungziele)
 - Debugging Information: Informationen, die dem Debugger ermöglichen, Maschinenbefehle mit C-Sourcecode in Verbindung zu bringen

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information
--------------------	--------------	--------------	------------------------	--------------	-----------------------

Vom Quellcode zum ausführbaren Programm



Linker

- Um eine weitgehende Wiederverwendung von Programmteilen zu ermöglichen, werden Programme Prozedur für Prozedur übersetzt.
- Für jedes einzelne der so entstehenden Programmstücke müssen die relativen Positionen innerhalb des Speichers bestimmt werden.
- Aufgaben des Linkers
 - Zusammenbinden der übersetzten Prozeduren zu einem ausführbaren Binärprogramm und symbolische Platzierung von Code und Daten im Speicher.
 - Bestimmung der Adressen von Daten-und Instruktionslabels
 - „Patchen“ von internen und externen Referenzen, d.h. Einsetzen der ermittelten Sprungweiten bzw. Sprungziele
 - Das Executable hat dasselbe Format wie das Objektfile, nur mit aufgelösten Referenzen
- Im Executable müssen alle Referenzen aufgelöst sein
 - Ausnahme: DLL (Dynamic Linked Libraries)

Linker

- Der Linker erzeugt aus den relativen Programm-Modulen des Assemblers ein relatives Gesamtprogramm, wobei i.A. Adressen neu berechnet werden müssen. Die Adressbezüge zwischen den einzelnen Modulen sind dann festgelegt, das gebundene Gesamtprogramm ist im Adressraum aber noch verschiebbar.
- Sonderfall: Im Modul1 erfolgt ein Aufruf eines Modul2 mit Namen SUBR (z.B.: BRA SUBR). Das Symbol SUBR kommt in Modul1 sonst nicht mehr vor. Dies würde zu einem Fehler in der 1. Phase der Assemblierung führen (Symboltabelle). Deshalb muss man dem Assembler kenntlich machen, dass dieses Symbol extern definiert wird (globales Symbol):

GLBL Symbol1 [, Symbol2,

- Diese Information nutzt der Linker zur Adressberechnung beim Binden der Module.

Beispiel: Aufgaben des Linkers

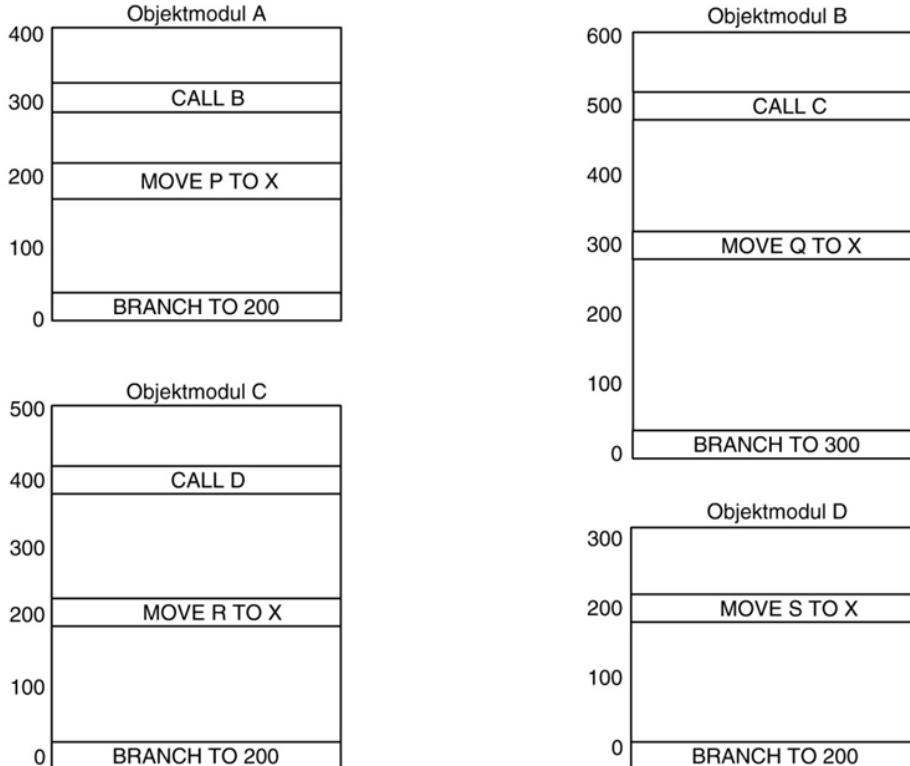
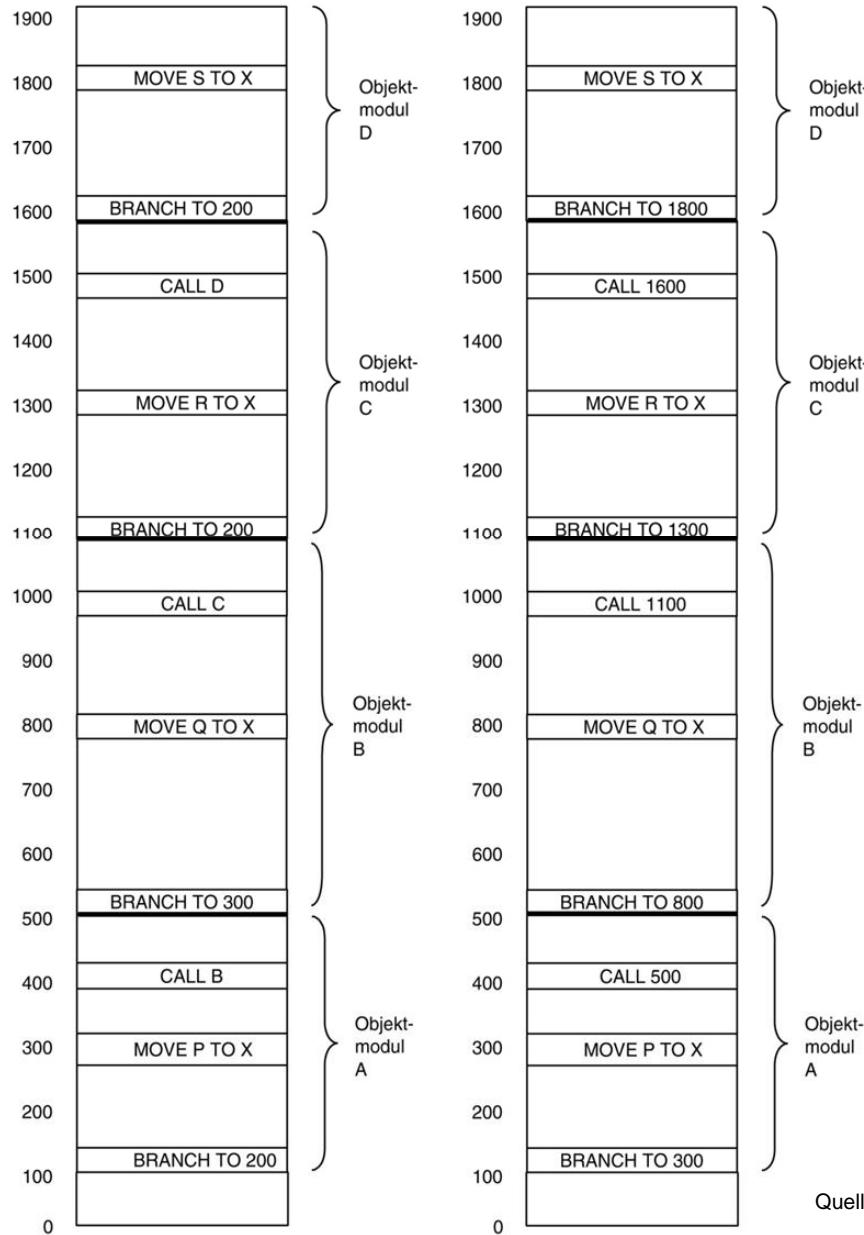


Abbildung 7.3: Jedes Modul hat seinen eigenen Adressraum, der jeweils bei 0 beginnt

Quelle: A.S. Tanenbaum: Rechnerarchitektur, 6. Auflage

Beispiel:

- Gegeben sind 4 Objektmodule, deren Startadresse jeweils 0 (Null) beträgt.
- Der Linker hat die Aufgabe, die getrennt übersetzten Prozeduren zusammen zu binden, damit sie sich als ausführbares Binärprogramm (Executable) ausführen lassen.
- Jedes Modul beginnt mit einem BRANCH-Befehl zu einem MOVE-Befehl im selben Modul. Innerhalb eines Moduls kann ein anderes Modul aufgerufen werden (CALL).



Beispiel: Aufgaben des Linkers

Beispiel:

- Um das Programm ausführen zu können, erstellt der Linker ein Abbild des (virtuellen) Adressraums und bringt alle Objektmodule an die richtigen Positionen.
- Als Startadresse für das Programm ist die willkürlich gewählte Adresse 100 angegeben.
- Die linke Abbildung zeigt die zusammengesetzten Prozeduren ohne Adresskorrekturen. In der rechten Abbildung sind die Adressen und Sprungziele an die neuen Adressen angepasst worden.

Quelle: A.S. Tanenbaum: Rechnerarchitektur, 6. Auflage

Abbildung 7.4: (a) Die Objektmodule von Abbildung 7.3, nachdem der Linker sie im binären Abbild positioniert, aber noch bevor er sie umgeordnet und gebunden hat. (b) Die gleichen Objektmodule nach dem Binden und der Relokation

Dynamic Linked Libraries (DLL)

- Eine DLL ist eine Bibliothek mit einer Sammlung von nützlichen Prozeduren, die sich in den Speicher laden und von mehreren Prozessen gleichzeitig nutzen lassen.

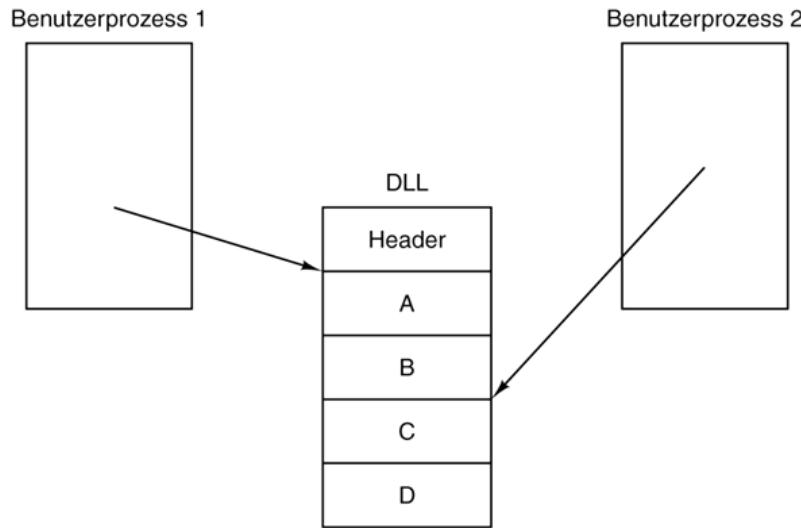


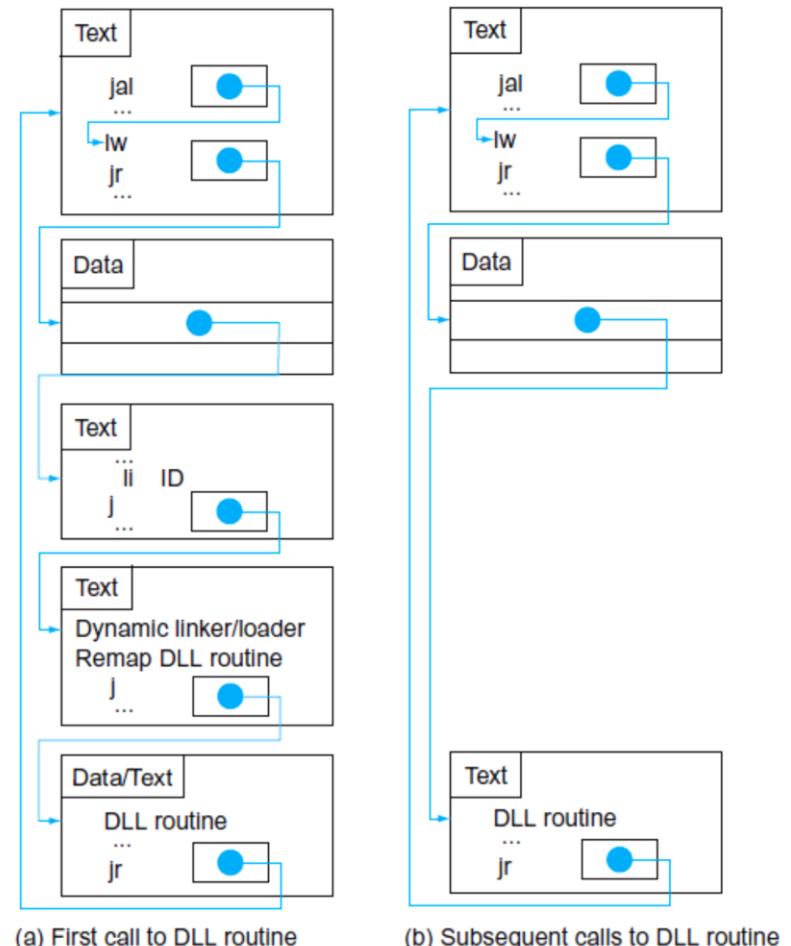
Abbildung 7.8: Zwei Prozesse nutzen eine DLL gemeinsam

- Betriebssysteme unterstützen **dynamisches Binden** zum Zeitpunkt des Prozedurauftrufs. Dies ermöglicht die gemeinsame Nutzung von Bibliotheken durch mehrere Prozesse. Die DLL wird dabei nur einmal in den Speicher geladen.
- Eine DLL (.dll) kann, im Gegensatz zu einem ausführbaren Binärprogramm, nicht eigenständig starten und ausgeführt werden.

Quelle: A.S. Tanenbaum: Rechnerarchitektur, 6. Auflage

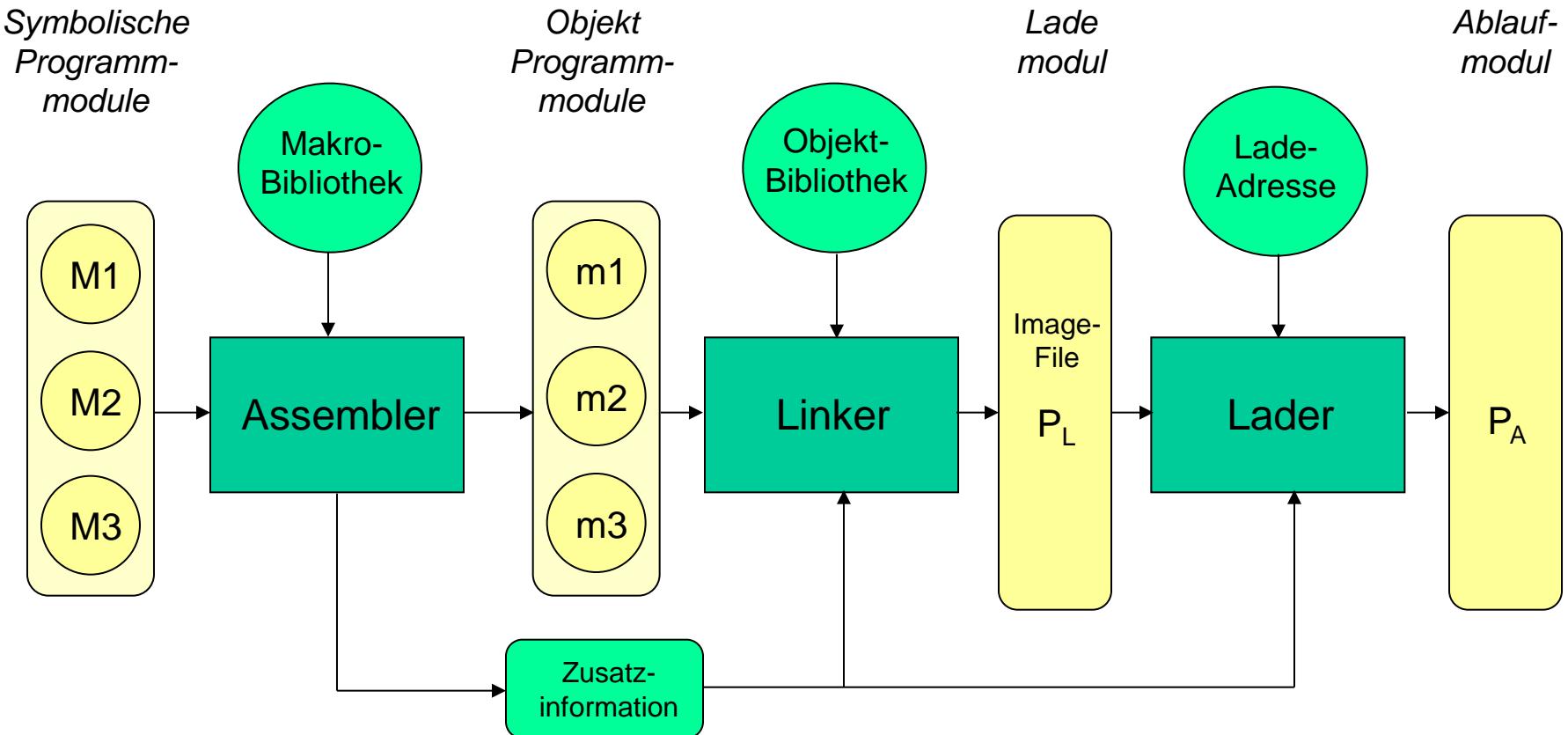
Dynamic Linked Libraries (DLL)

- Library-Routinen werden nicht vom Loader eingebunden sondern während der Laufzeit nachgeladen.
 - Es müssen nicht alle Routinen einer Library genutzt werden.
 - Bug Fixes für Libraries können vorgenommen werden, ohne Programme zu ändern.
- Prinzip:
 - Routine wird beim ersten Aufruf dem Programm-Code hinzugefügt und die Code-Adresse an entsprechende Stelle im Speicher eingefügt.



Quelle: Patterson, Hennessy: Rechnerorganisation und -entwurf: Rechnerarchitektur, 3. Auflage

Zusammenspiel: Assembler, Linker und Lader



Lader

- Betriebssystemroutine, die ein Anwendungsprogramm zur Ausführung auf einem Zielrechner bringt.
- Dabei werden z. B. in Unix die folgenden Schritte ausgeführt:
 - Lesen des Programms und Bestimmen der Programm- und Datensegmentgrößen.
 - Bereitstellen eines hinreichend großen Adressraums für Programm und Daten.
 - Kopieren der Instruktionen und Daten aus dem Programm in den Arbeitsspeicher.
 - Initialisierung der Maschinenregister; Setzen des Stackpointers auf die erste freie Position.
 - Sprung zu einer Start-up-Routine, die zur main-Prozedur des Programms verzweigt.

Lader

- Beim Assembler-Entwicklungssystem:
Der Lader lädt das Programm entweder an die Startadresse, die durch die **ORG-Anweisung** festgelegt wurde oder er nimmt eine Verschiebung des Programms um eine vorzugebende Ladedistanz vor (relocating loader).
- Insbesondere im Bereich der **Multitasking Systeme** ist es unumgänglich, dass alle Module oder Prozesse (Tasks) im Adressraum **verschieblich** sind. Da aufgrund der Freispeicherverwaltung des Betriebssystems die aktuelle Ladeadresse für eine Task variieren kann, ist es zum Zeitpunkt der Programmerstellung nicht möglich, diese Ladeadresse anzugeben.

Bisher betrachtet:

- Assemblerbefehle des Prozessors
 - Adressierungsarten
 - Pseudobefehle
 - Technik der Assemblierung (Zuordnungstabelle, Symboltabelle)
 - Globale Namen (Linker)
 - Konzept der lokalen Variablen (Stackbereich und Verwaltung)
 - Unterscheidung LIFO (Stack) und FIFO (Queue)
 - Parameterübergabe über den Stack
 - Unterprogrammtechnik (JSR, RTS)
-
- Jetzt noch: Synchronisation

Synchronisation von Prozessen

- **Sequentielle Abläufe** spiegeln den Kausalzusammenhang von Prozessen wieder. Sie bilden eine Ereigniskette oder eine Folge von Aktionen ab. Sequentielle Abläufe lassen sich gut mit einer sequentiellen Programmabwicklung implementieren.
- **Parallele Abläufe** spiegeln kausal nebenläufige Abläufe wider. Die Implementierung echter Parallelität erfordert auch mehrere Prozessoren, die parallel arbeiten können. In Monoprozessorsystemen erfolgt eine Zwangs-Sequenzialisierung der parallelen Prozesse.
- Die Modellierung solcher Prozesse erfolgt zweckmäßigerweise mit **Petri-Netzen**.

Synchronisation von Prozessen

- Zwei Prozesse sind **nebenläufig (concurrent)**, wenn es mindestens ein Zeitintervall gibt, zu dem beide begonnen, aber nicht abgeschlossen sind. (In Monoprozessorsystemen können konkurrente Prozesse nur abschnittsweise sequentiell abgearbeitet werden).
- Nebenläufige Prozesse können kommunizierend und auch kooperierend sein. Greifen 2 Prozesse „gleichzeitig“ auf gemeinsame Daten zu, kann dies zu undefinierten und inkorrekten Zuständen führen.
- Prozessabschnitte, deren Ausführung sich mit der Ausführung anderer Prozesse nicht überlappen dürfen, nennt man „**Kritische Abschnitte**“ oder „**kritische Bereiche**“ (critical regions).

Synchronisation von Prozessen

- Spezielle Steuerungsmechanismen sollen verhindern, dass beim Bearbeiten eines Prozesses in einen kritischen Abschnitt kein anderer Prozess diesen Abschnitt betreten darf. Man nennt dies „**gegenseitiger Ausschluss**“ (mutual exclusion).
- Der einfachste Mechanismus zum gegenseitigen Ausschluss ist der „**Semaphor-Mechanismus**“ (Dijkstra).
- Einfachster Fall: Semaphor entspricht boolescher Variablen.
- Die beiden Werte („0 ↔ frei“ oder „1 ↔ belegt“) zeigen den Zustand eines kritischen Abschnitts an

Synchronisation von Prozessen

- Gegeben seien zwei Prozeduren $P(S)$ und $V(S)$, durch die der Wert eines Semaphors S verändert werden kann, und zwar:

```
P(S):      begin
            if S=1 then < wait until S=0 >
            S:=1
        end;
```

```
V(S):      begin
            S:= 0
        end;
```

Synchronisation von Prozessen

- Wenn sich zwei Prozesse P1 und P2 gegenseitig aus einem kritischen Bereich ausschließen sollen, muss gelten:

P1

S: semaphor

•
•

P(S)

<Anweisungen des
kritischen Abschnitts,
Datenzugriff auf globale
Variable>

V(S)

•
•

P2

S: semaphor

•
•

P(S)

<Anweisungen des
kritischen Abschnitts,
Datenzugriff auf globale
Variable>

V(S)

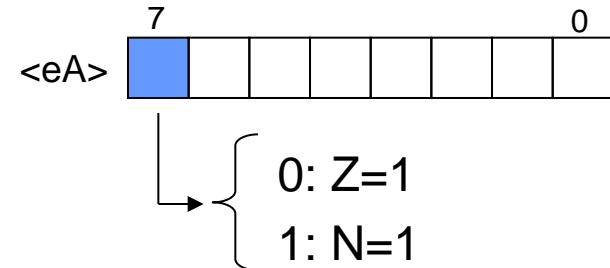
•
•

Unteilbarkeit
der
Operation
 $P(S)$
gefordert!

Synchronisation von Prozessen

- Die Semaphor-Operationen sind leicht zu verstehen und zu implementieren. Aber wie wird die Unteilbarkeit der Operation erreicht?
- Auf der Ebene des Prozessors und seines Befehlssatzes gibt es den Befehl: „**TAS**“. TAS operiert mit einem binären Semaphor.

- Test-and-Set: **TAS <eA>**
 - Destination tested → CC
 - 1 → Bit Nr. 7 von <eA>



Auf der Hardwareebene der Prozessorrealisierung wird ein sogenannter **Read-Modify-Write-Zyklus** ausgeführt.

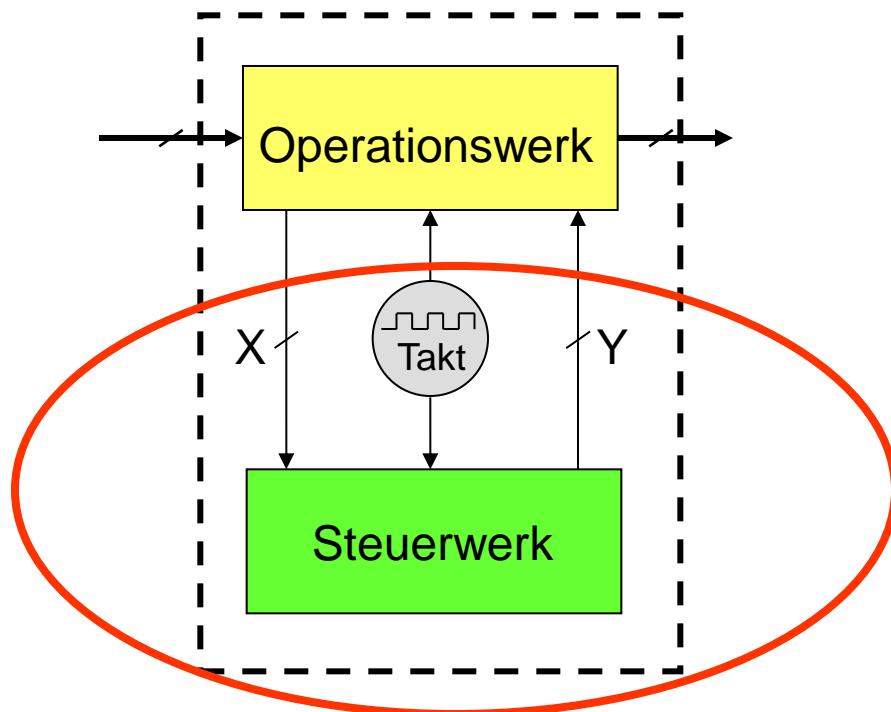
Synchronisation von Prozessen

- Semaphor-Mechanismen sind unsichere Konstrukte:
 - Überspringen von P(S) führt dazu, dass der Mechanismus außer Kraft gesetzt wird.
 - Überspringen von V(S) führt zur Blockierung der anderen Prozesse
 - Mechanismus kann ganz vergessen werden
- Es handelt sich um ein Low-Level-Konstrukt. Es existieren sicherere Mechanismen, z.B. das Monitor-Konzept. Weitere Details dann im Kurs Betriebssysteme.

Steuerwerk des Prozessors

Steuerwerk des Prozessors

Innere Struktur des Prozessors

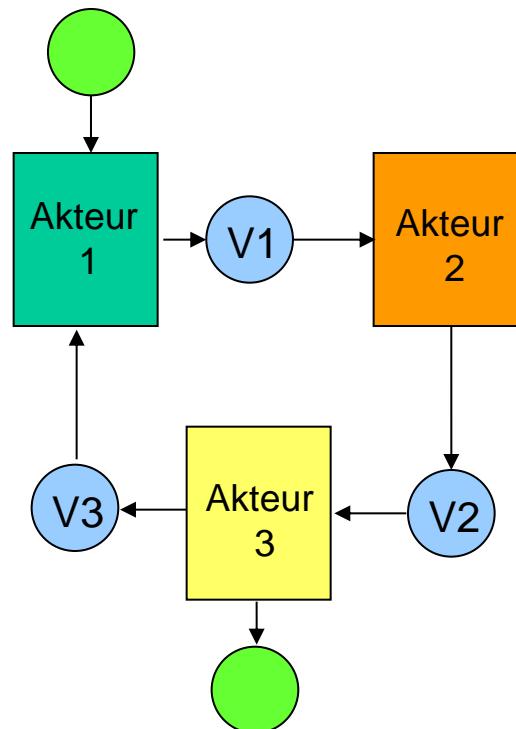


Wie erfolgt die Interpretation der Befehle und wie wird die zur Abwicklung der Befehle im Operationswerk benötigte Steuerinformation (Y) bereitgestellt ?

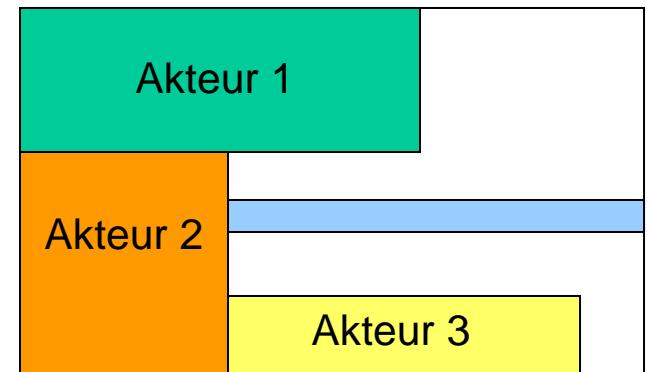
- ↗ Steuert die Systemkomponenten

Direkte und indirekte Realisierung einer Funktionseinheit (Akteur)

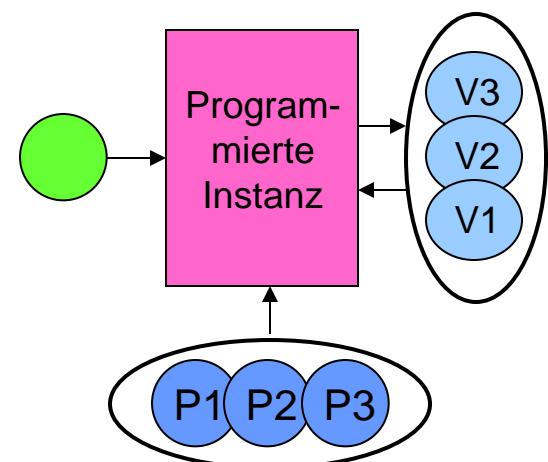
Block-Diagramm



Direkte
Realisierung



Indirekte
Realisierung



Direkte und indirekte Realisierung einer Funktionseinheit (Akteur)

- Direkte Realisierung bedeutet, dass eine physische Struktur derart partitioniert werden kann, dass sich eine 1:1-Abbildung zwischen den Akteuren und Variablen des Block-Diagramms und den Partitionsblöcken der physischen Struktur ergibt.
- Unter der indirekten Realisierung versteht man die Mehrfachausnutzung von Komponenten der physischen Struktur zur Realisierung von Komponenten des Block-Diagramms. Damit ändert sich die funktionale Zuordnung mit der Zeit. Dieses zeitliche Nacheinander der Zuordnung (Zeitmultiplex) bewirkt eine Zwangssquentialisierung der einzelnen Akteure.
- Werden die Akteure des Steuerwerks nicht direkt, sondern indirekt realisiert, spricht man vom Aktionsträgermultiplex. Zur Realisierung dient ein mikroprogrammiertes Steuerwerk, das sequentiell die einzelnen Programme der Akteure ausführt.

Realisierung des Steuerwerks

Zwei Techniken zur Implementierung von Befehlen im Rechner

➤ **Direkt durch Hardware**

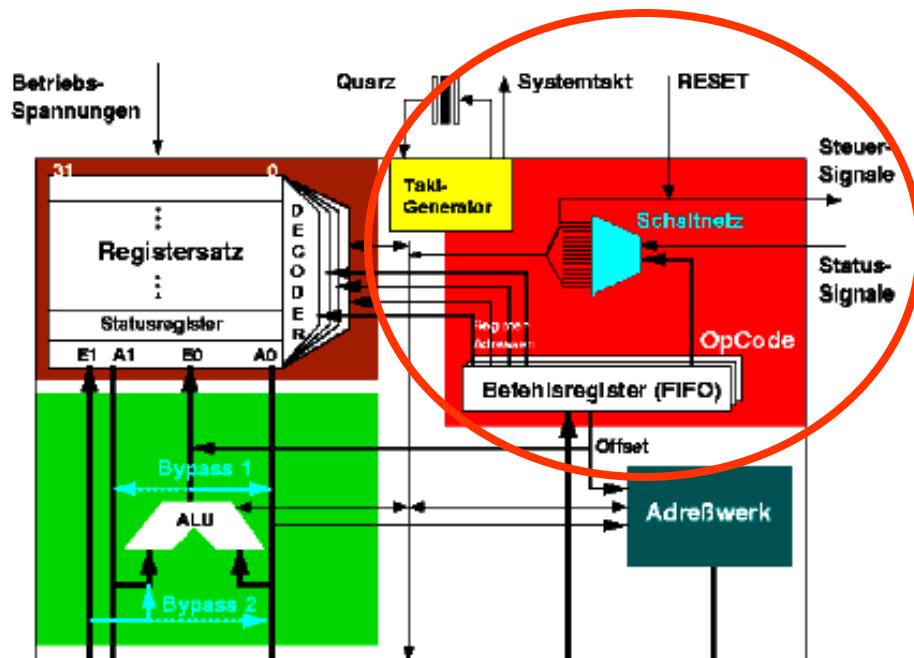
- aufwendige Schaltnetze und Schaltwerke bei großer Anzahl von Befehlen (200-300)

➤ **Mikroprogramme als Befehlsinterpretierer**

- Mikroprogrammspeicher im Steuerwerk, der neu geladen werden kann
- verschiedene Befehlssätze können implementiert werden

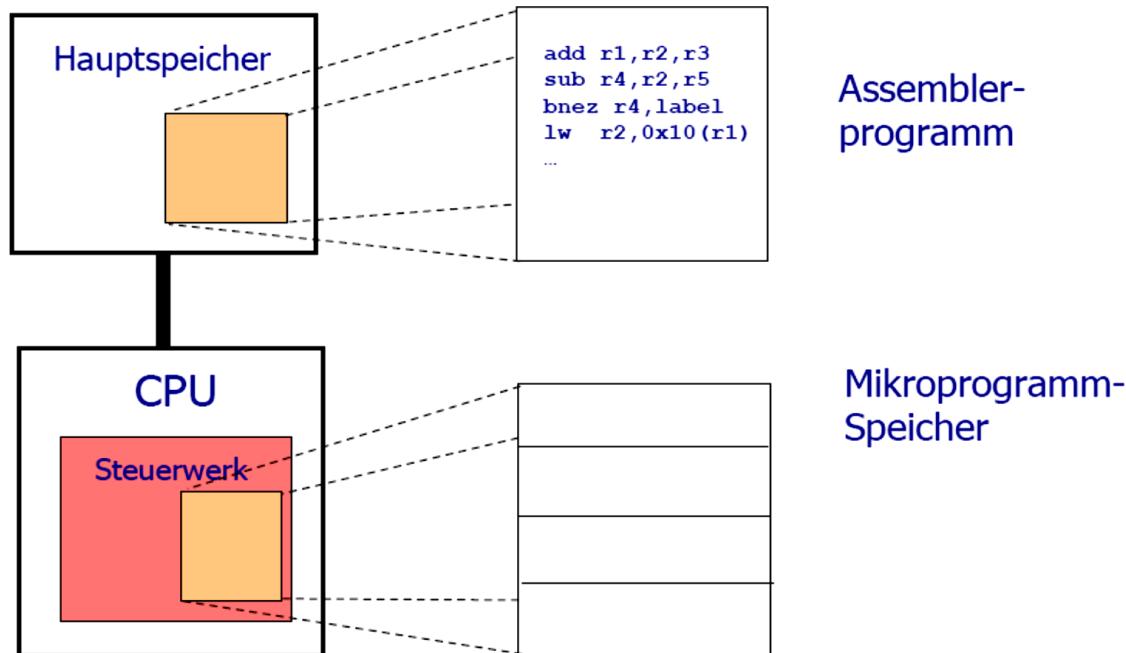
Realisierung des Steuerwerks

- Realisierung als kombinatorisch sequentielles Netz
 - „hardwired control unit“ --> RISC
 - schnellstmögliche Ausführung von Operationen
 - starre Festlegung der Steuervorgänge
 - größerer Platzbedarf wegen inhomogener Struktur



Realisierung des Steuerwerks

- Realisierung als programmierbare Steuerung
 - „microprogrammed control unit“ --> CISC
 - leichte Erweiterbarkeit
 - Flexibilität
 - Langsamere Ausführung (Speicher, Decodierung)



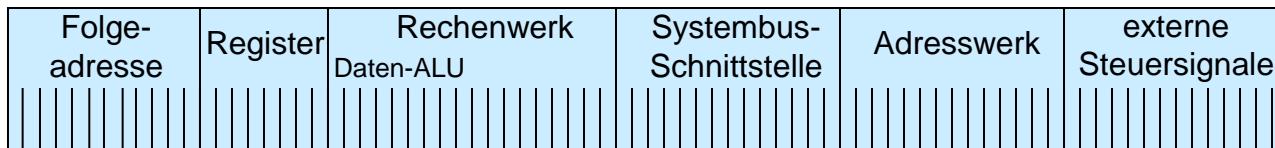
Steuerwerk als mikroprogrammiertes Schaltwerk

Mikroprogrammsteuerwerk

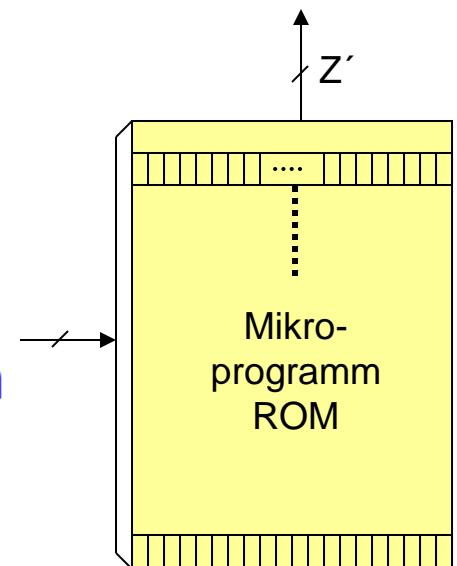
Im Festwertspeicher liegt für jeden Befehl ein Mikroprogramm

Mikroprogramm \equiv Folge von Mikrobefehlen

Aufbau eines Mikrobefehls:



Einzelne Bits eines Mikrobefehls \equiv Mikrooperationen \equiv Auswahl- und Freigabesignale für die benötigten Komponenten



Steuerwerk als mikroprogrammiertes Schaltwerk

- Das Mikroprogramm-ROM enthält alle Sequenzen, die zur Steuerung der Abwicklung von Maschinenbefehlen benötigt werden. Die Adressierung des Mikro-ROMs erfolgt durch den Mikro-PC. Zu jedem Operator eines Maschinenbefehls der ISA-Ebene gehört eine Mikroprogrammsequenz. Die Zuordnung erfolgt durch das Mapping-ROM. Ferner gibt es z.B. die Befehlsholsequenz oder die Sequenz zur Steuerung des Stackpointers als Mikroprogramm.
- Mikroprogramme können vom Benutzer nicht geändert werden, d.h. CISC-Prozessoren sind meist mikroprogrammiert, aber nicht mikroprogrammierbar.
- Erinnerung - Schichtenmodell eines Rechners: Das Mikroprogramm, als Bestandteil der Mikroarchitekturebene, ist ein Interpreter für die Maschinenbefehle der ISA-Ebene.

Steuerwerk als mikroprogrammiertes Schaltwerk

Die Abwicklung des Mikroprogramms

Holphase

der nächste Befehl in das Befehlsregister laden

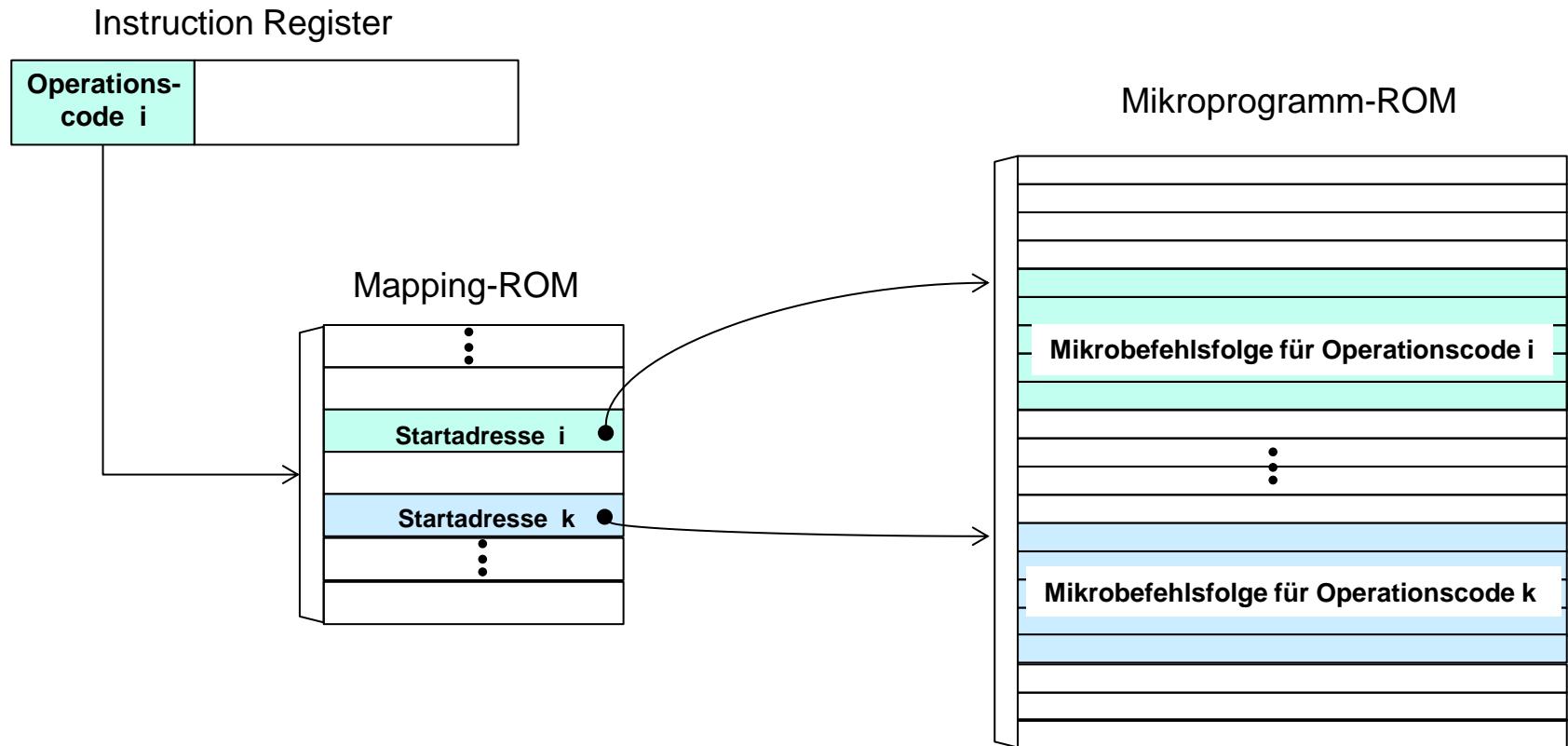
Decodierphase

der Befehlsdecoder ermittelt die Startadresse des Mikroprogramms, welches den Befehl ausführt

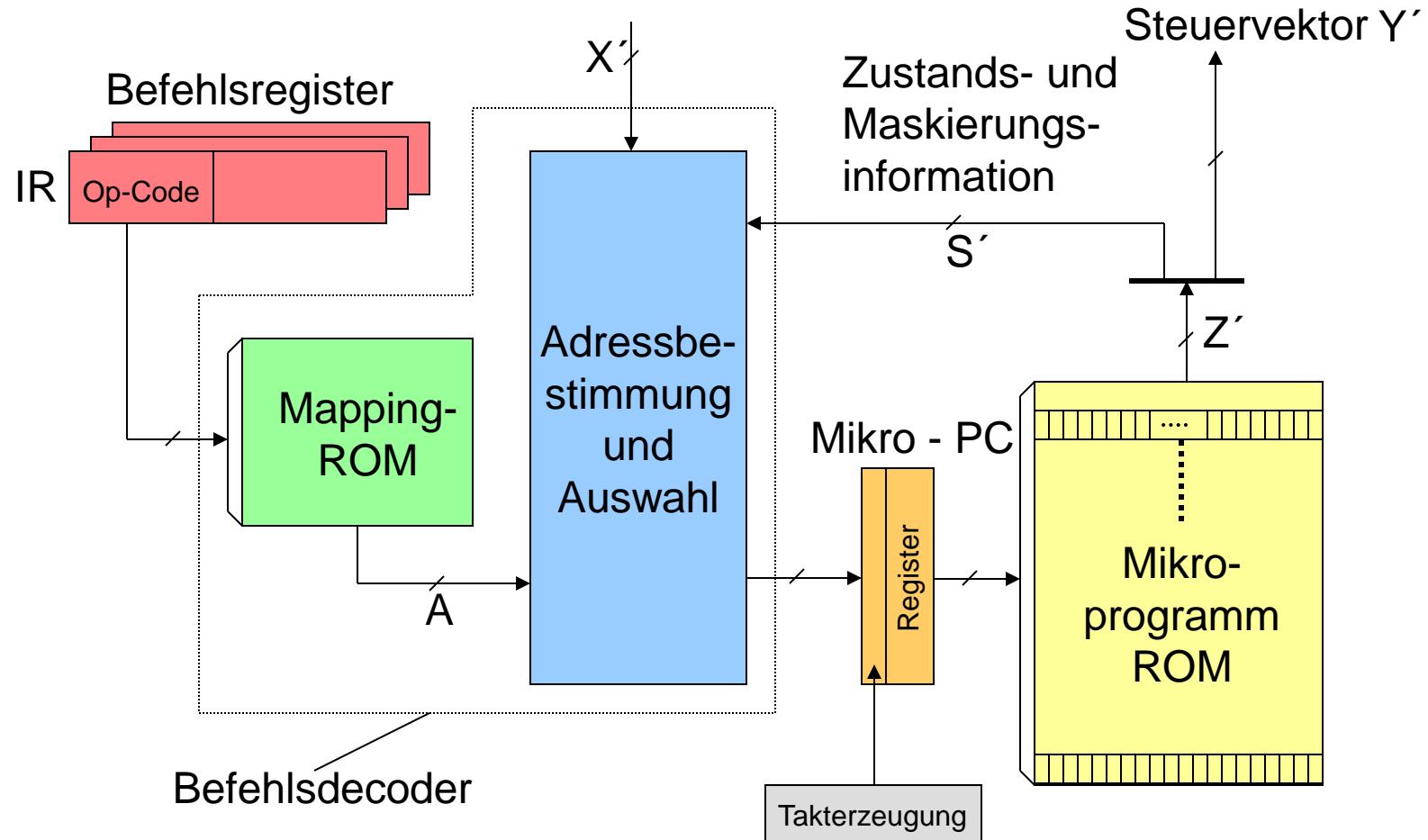
Ausführungsphase

das Mikroprogramm steuert die Befehlausführung, indem es entsprechende Signalfolgen an die anderen Prozessorkomponenten übermittelt und Meldesignale auswertet

Vom Operationscode zum Mikroprogramm



Steuerwerk als mikroprogrammiertes Schaltwerk



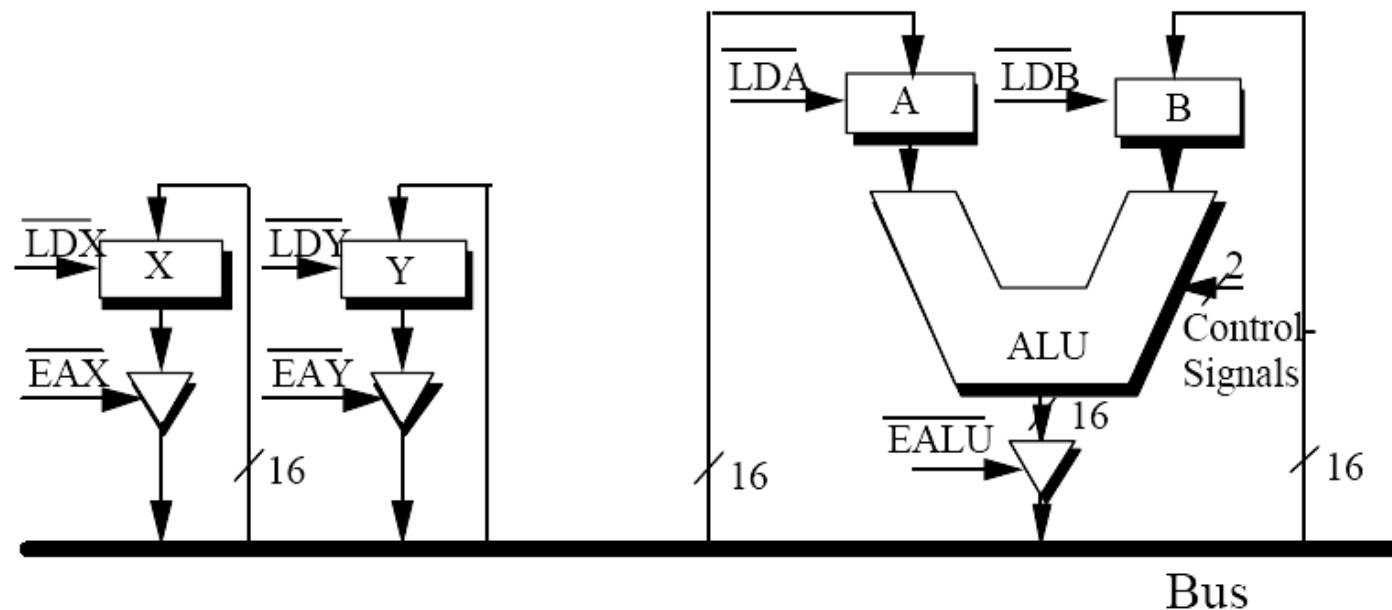
Steuerwerk als mikroprogrammiertes Schaltwerk

- Das Mikroprogramm-ROM, als Teil des Steuerwerks, muss auf dem Prozessorchip integriert werden, deshalb soll der Speicherumfang möglichst gering sein (Chipfläche). Als eine Konsequenz daraus muss man die Anzahl der Adressleitungen (Eingangsvariablen) verkleinern.
- Bei einem Schaltwerk sind in einem Zustand nie alle Entscheidungsvariablen (X-Vektor), die die Fortschaltung des Zustands sowie die Ausgabe beeinflussen, relevant. Praktische Beispiele zeigen, dass meist nur ein kleiner Teil der Eingangsinformation ausgewertet werden muss ==> Separation relevanter Entscheidungsvariablen. Dazu dienen die Adressbildungsnetze. Die sog. Maskierungsinformation dient dazu, nur die im aktuellen Zustand relevanten Eingangsvariablen auszuwählen.
- Solche Mikroprogramm-Kerne gibt es auch als hochintegrierte Bausteine, z.B. MARC (Minimized Address ROM Based Controller)

Beispiel einer einfachen Mikroprogrammierung

Gegeben:

Eine mikroprogrammierbare Schaltung besteht aus einem Bus, 4 Registern, einer ALU und einigen Tristate-Treibern



Beispiel einer einfachen Mikroprogrammierung

Gesucht:

Mikroprogramme für die folgenden Operationen 1-5
durch Programmierung eines Datenpfades:

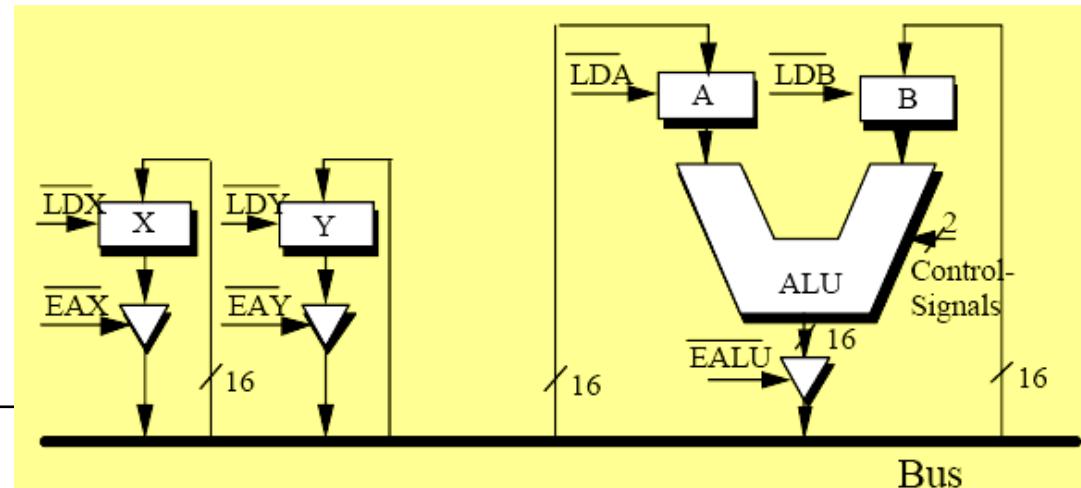
		Controlsignale	Operation
1.	$x = x + y$	$C_1 \ C_0$ 0 0	A + B
2.	$x = x - y$	0 1	A - B
3.	$x = x \text{ and } y$	1 0	A and B
4.	$x = x \text{ or } y$	1 1	A or B
5.	$y = x$ (move x to y)		

Beispiel einer einfachen Mikroprogrammierung

Lösung:

Mikroprogramme für die Operationen 1-4:

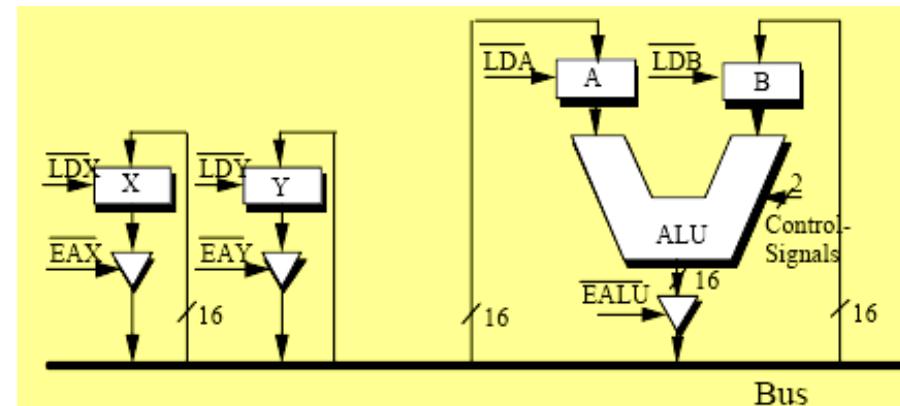
- X auf dem Bus legen. Warten bis die Daten stabil anliegen.
- X ins Register A laden.
- Y auf dem Bus legen. Warten bis die Daten stabil anliegen.
- Y ins Register B laden.
- Ergebnis auf den Bus. Warten bis es stabil anliegt.
- Ergebnis ins Register X laden.



Beispiel einer einfachen Mikroprogrammierung

Lösung:

<u>EAX</u>	0	0	1	1	1	1
<u>LDX</u>	1	1	1	1	1	0
<u>EAY</u>	1	1	0	0	1	1
<u>LDY</u>	1	1	1	1	1	1
<u>LDA</u>	1	0	1	1	1	1
<u>LDB</u>	1	1	1	0	1	1
<u>EALU</u>	1	1	1	1	0	0
C_1	-	-	-	0	0	0
C_0	-	-	-	0	0	0



$$x = x + y$$

$$x = x - y$$

$$x = x \text{ and } y$$

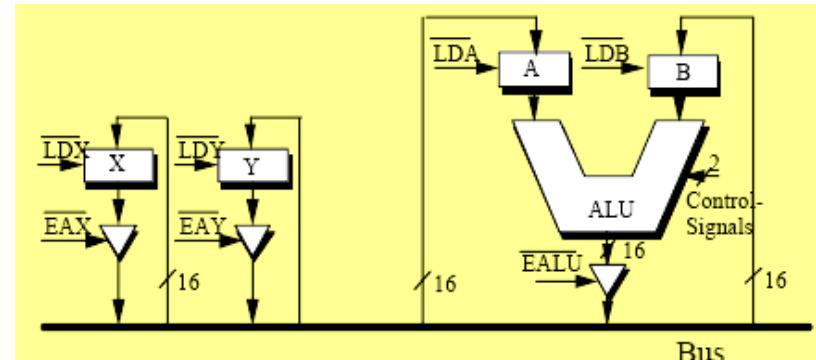
$$x = x \text{ or } y$$

Beispiel einer einfachen Mikroprogrammierung

Mikroprogramm für die Operation 5:

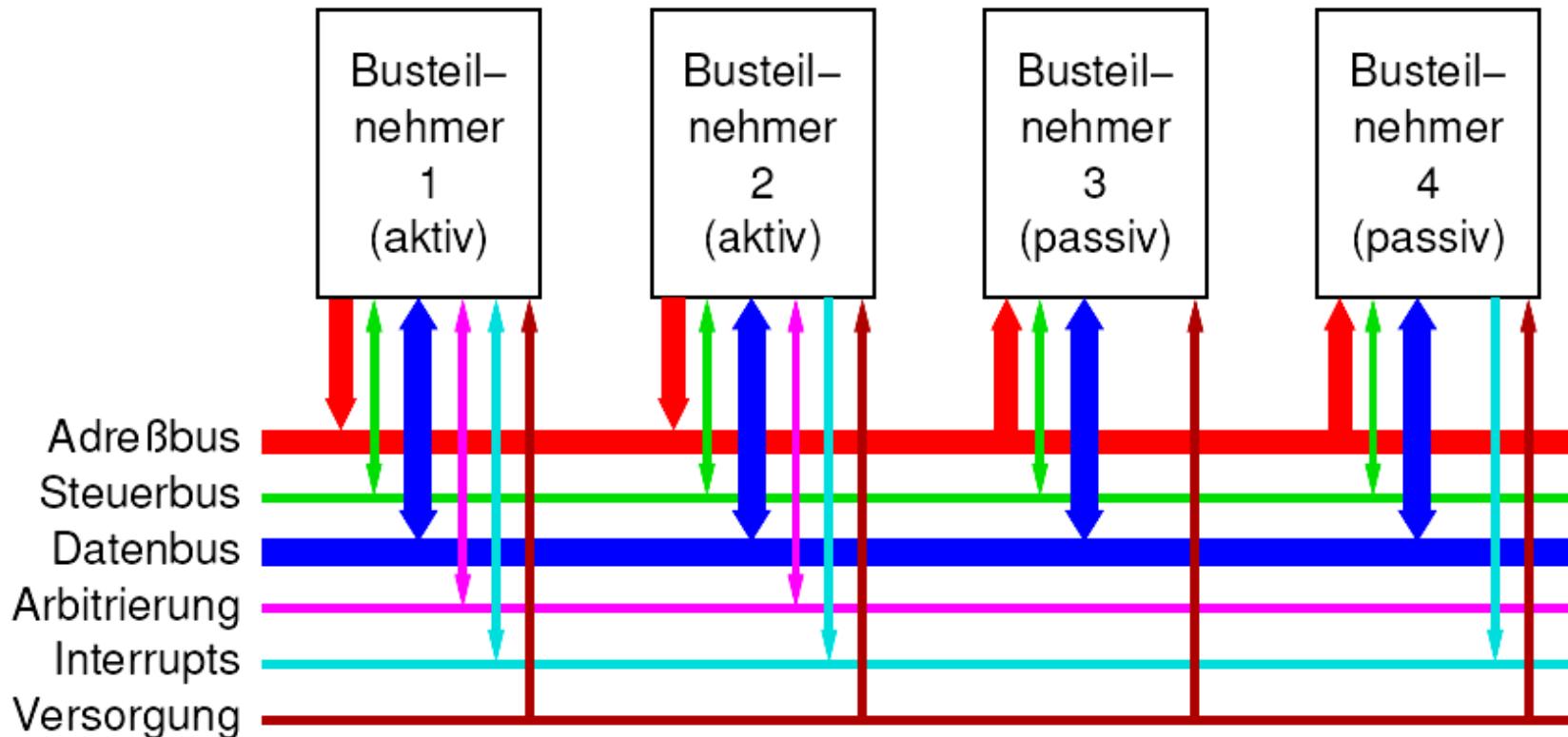
- X auf dem Bus legen. Warten bis die Daten stabil anliegen.
- Y laden.

<u>EAX</u>	0	0
<u>LDX</u>	1	1
<u>EAY</u>	1	1
<u>LDY</u>	1	0
<u>LDA</u>	1	1
<u>LDB</u>	1	1
<u>EALU</u>	1	1



Systembus

Typischer Aufbau eines parallelen Busses



Systembus

Systembus-Zuteilung:

meist müssen mehrere Komponenten eines Mikroprozessor-Systems aktiv auf den Systembus zugreifen

- Systembuszuteilung (Bus Arbitration) erforderlich

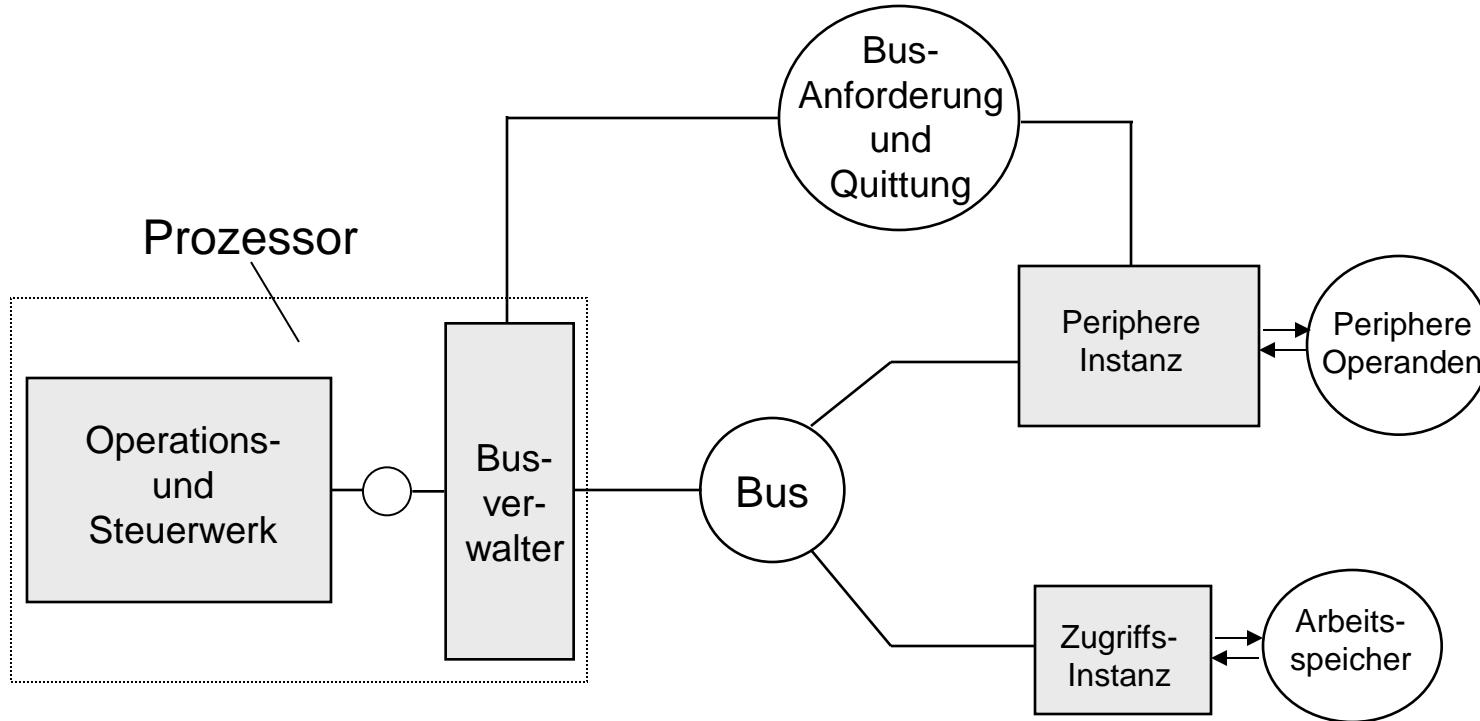
Bus - Arbiter

Ein Systemsteuer-Baustein wird benötigt, wenn mehrere Komponenten selbstständig auf den Systembus zugreifen und so zum *Bus Master* werden. Z.B.

- **In einem Mikrocomputer:**
Co-Prozessoren, DMA-Controller, DRAM-Controller ... usw.
können *Bus Master* werden
- **In einem speichergekoppelten Multiprozessorsystem:**
mehrere Prozessoren sind über den Systembus mit
einem gemeinsamen Speicher verbunden.

Der Bus-Arbiter regelt hierbei die Hierarchie der Zugriffsberechtigungen und gewährt den Zugriff auf den Systembus.

Lokale Bus-Arbitration



- Bus-Anforderung: BR
- Bus-Vergabe: BG
- Bus-Belegt: BGACK

Busvergabe - 3-Leitungs-Handshake

□ Busanforderung, (*bus request*): BR

Dieses Signal teilt dem Prozessor mit, dass eine andere Systemkomponente den Systembus belegen will

□ Busfreigabe (*bus grant*): BG

Mit diesem Signal teilt der Prozessor mit, daß der Systembus nun zur Verfügung steht. Alle Prozessorausgänge werden vorher auf hochohmig geschaltet

□ Busübernahme-Bestätigung:

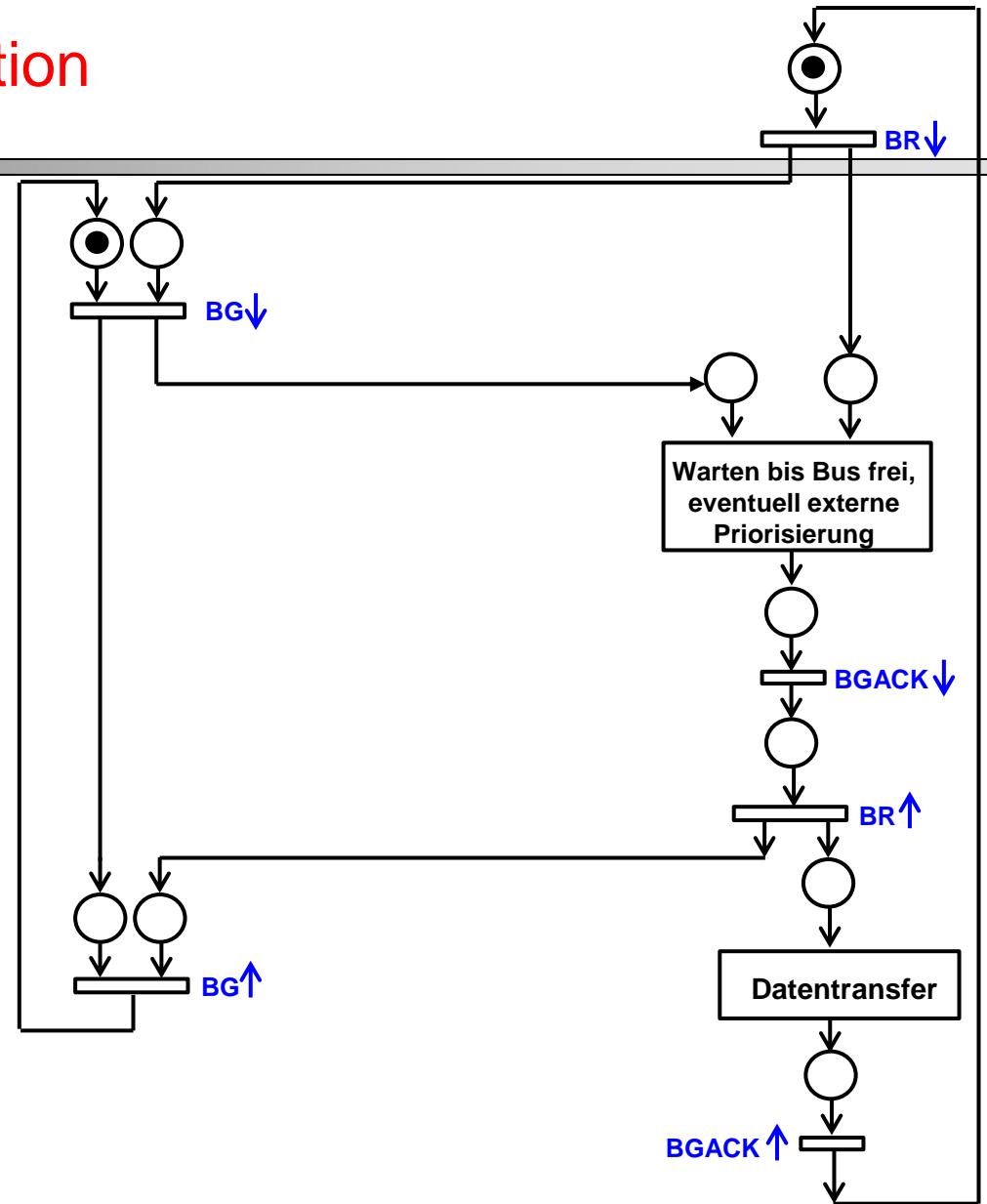
(*Bus grant acknowledge* BGACK)

Mit diesem Signal bestätigt eine Komponente dem Prozessor, daß sie nun den Bus übernommen hat (neuer Bus-Master)

Bewerben sich mehrere Komponenten gleichzeitig um den Bus, so muß der Bus-Arbiter eine hiervon auswählen

Protokoll der Busarbitration

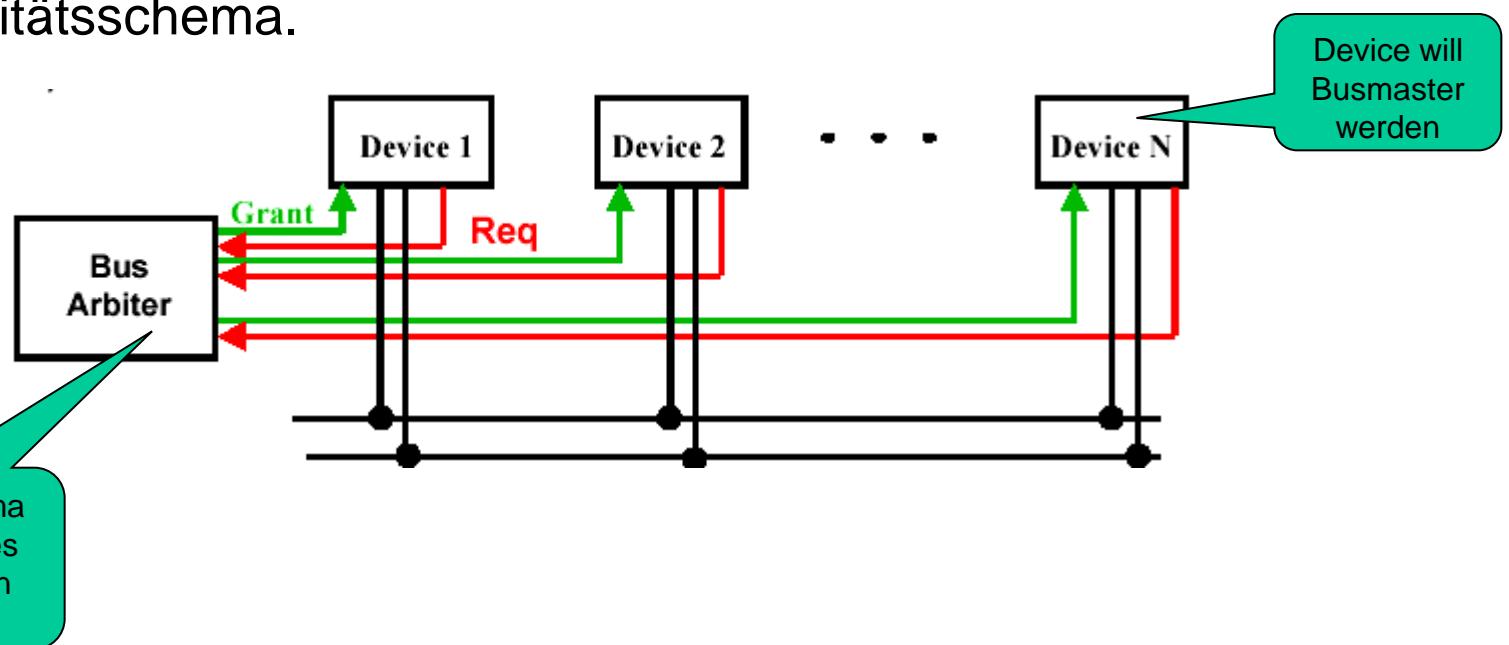
Das Petrinetz modelliert das Kommunikationsprotokoll zur Vergabe des Systembus. Die anfordernde Instanz kommuniziert mit dem Busverwalter des Prozessors



Priorisierung mehrerer Master

- **Zentrales Verfahren**

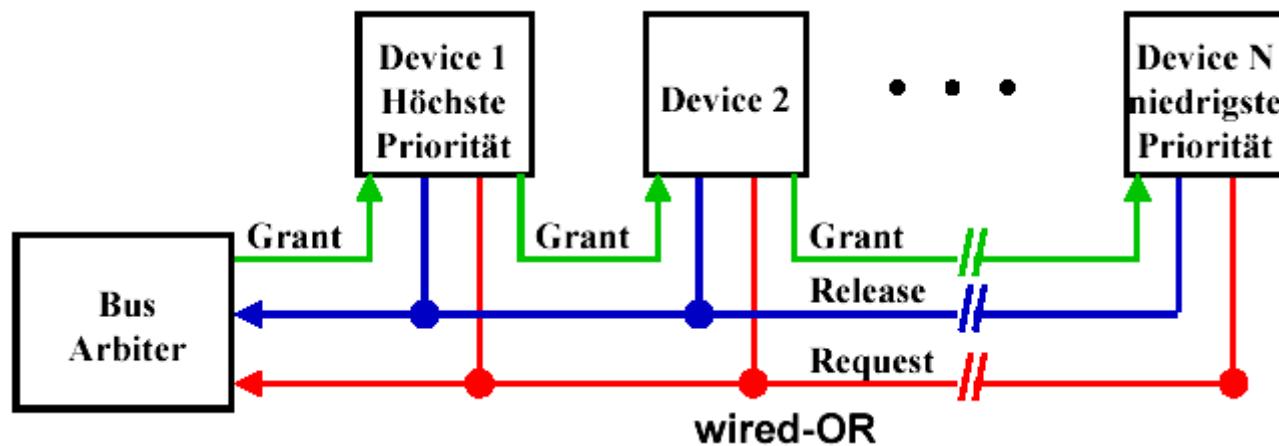
Jede masterfähige Instanz fordert den Zugriff auf den Systembus von einem zentralen Bus-Arbiter an. Dieser vergibt den Bus bei mehreren gleichzeitigen Anforderungen nach einem Prioritätsschema.



Priorisierung mehrerer Master

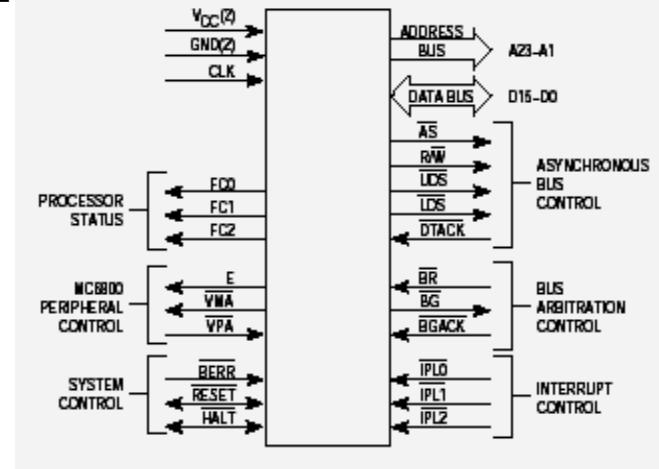
- Dezentrales Verfahren (Daisy Chain)

Alle masterfähigen Instanzen werden kettenförmig miteinander verbunden. Jede Instanz schiebt den „*Bus Grant*“, den sie selbst nicht benötigt (kein eigener „*Bus Request*“), an die nächste weiter. Die erste Instanz in der Kette hat die höchste Priorität („[geografische Priorität](#)“).



Ein- / Ausgabesignale des Steuerwerks

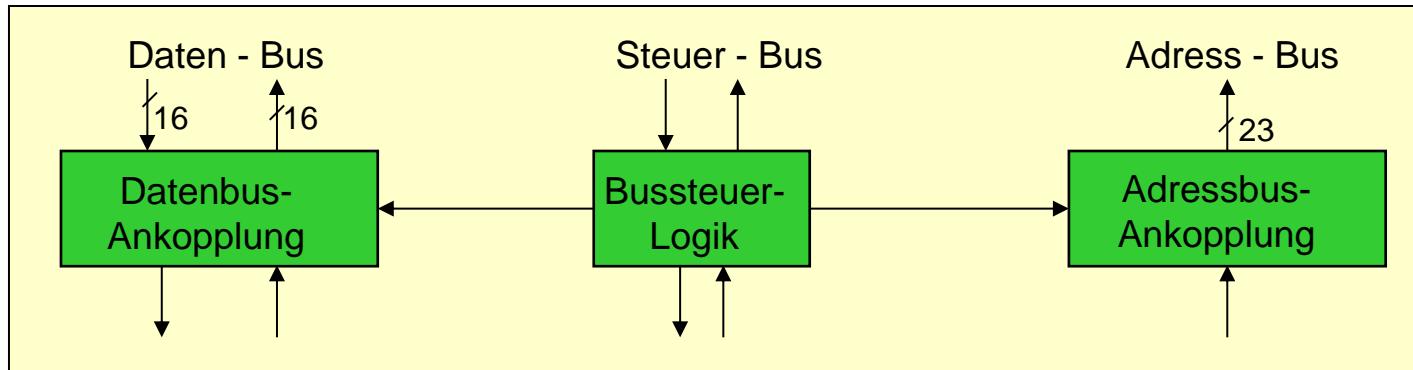
- Unterbrechungsanforderungen (Interrupt request)
 - Anfragen zur Unterbrechung des laufenden Programms, bei Akzeptanz wird eine Interrupt-Service-Routine gestartet.
- Fehlermeldungen der Systemkomponenten zum Prozessor (BERR)
 - Ausbleibendes Quittungssignal
 - Datenübertragungsfehler auf dem Bus
- Statusinformation, die der Prozessor ausgibt
 - Zugriff auf Daten/Befehle, User-, Supervisor-Mode
- Systembus-Steuersignale
 - Art und Richtung des Datentransports



Systembus

Die Systembus - Schnittstelle

Die Systembus-Schnittstelle (Bus Interface Unit, BIU) stellt die Verbindung des Mikroprozessors zu seiner Umwelt (Komponenten des Mikrorechnersystems) dar.



Aufgaben:

- ⇒ kurzfristiges Zwischenspeichern (Puffern) von Adressen und Daten
- ⇒ elektrische Anpassung der Signalpegel

Zeitverhalten von Systembus - Signalen

- ❑ Synchroner Systembus
- ❑ Semi-synchroner Systembus
- ❑ Asynchroner Systembus

- ❑ Beispiele für asynchronen Systembus:
Motorola 68000 - 68030
- ❑ Beispiele für semi-synchronen Systembus:
Intel-Prozessoren, Motorola 68040

Synchroner Bus

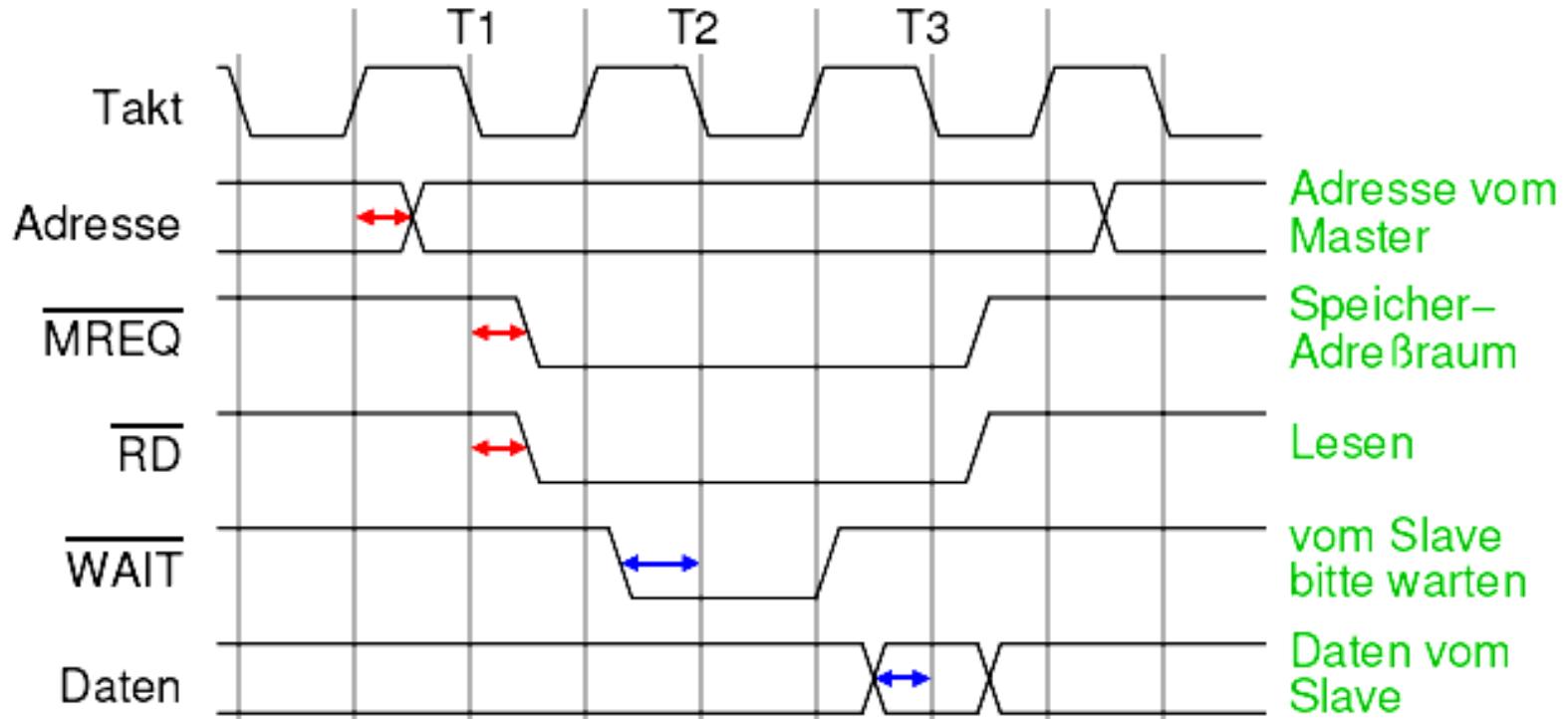
- Beim synchronen Bus existiert für einen Bustransfer zwischen zwei Funktionskomponenten ein festes Zeitraster, innerhalb dessen der Transfer abgewickelt werden muss.
 - Alle Bustransaktionen werden durch zentralen Takt gesteuert
- Das Kommunikationsverhalten am Bus ist vom Typ „Master-Slave“.
- Beim synchronen Betrieb setzt der Master beim Slave feste Reaktionszeiten voraus.
 - Nachteil: die langsamste Funktionseinheit am Bus bestimmt die maximale Transfersgeschwindigkeit.
 - Vorteil: einfacher Kommunikationsmechanismus: Master muss seine Signale für eine bestimmte Anzahl von Taktzyklen bereitstellen und kann sich darauf verlassen, dass der Slave reagiert.

Semi - synchroner Systembus

- Bedingt durch höhere Taktfrequenzen dauern Schreib- und Lesezyklen oft mehrere Taktzyklen. Kann die angesprochene Funktionseinheit nicht direkt die Daten liefern, fordert sie **Wartezyklen (wait states)** beim Prozessor an (Signal WAIT). Damit können unterschiedlich schnelle Speicher und Geräte am Systembus angeschlossen werden.
- Der Bus ist immer noch synchron, aber die Dauer eines Buszyklus ist nicht mehr fest, sondern ist in Vielfachen von Taktzyklen variierbar.
=> **Semi-synchroner Systembus**

Semi-Synchroner Bus

Beispiel für synchrones Busprotokoll



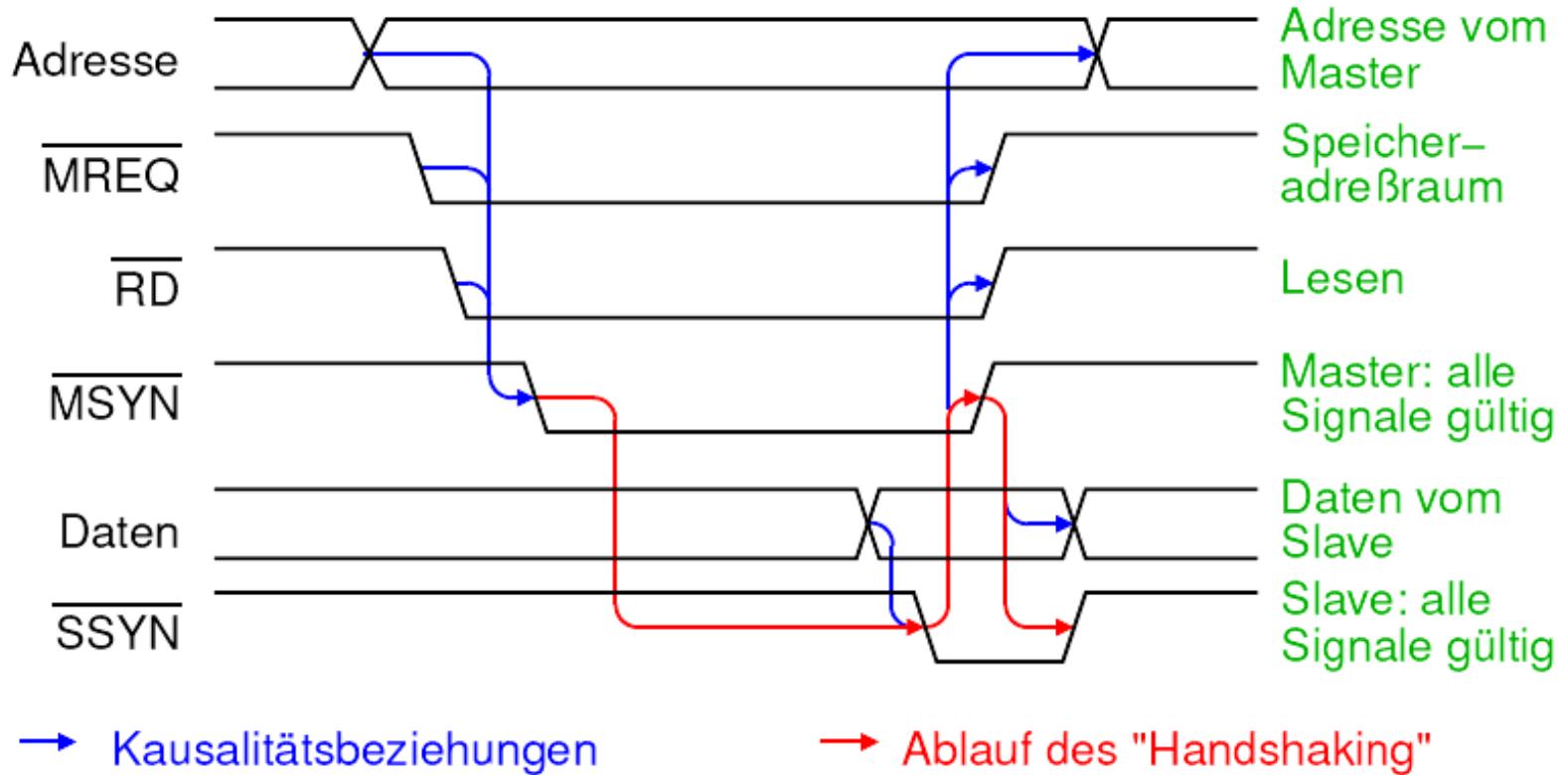
- Signal innerhalb vorgeschriebener Zeit nach der Taktflanke gültig sein
- Signal muß eine bestimmte Zeit vor der Taktflanke gültig sein

Asynchroner Systembus

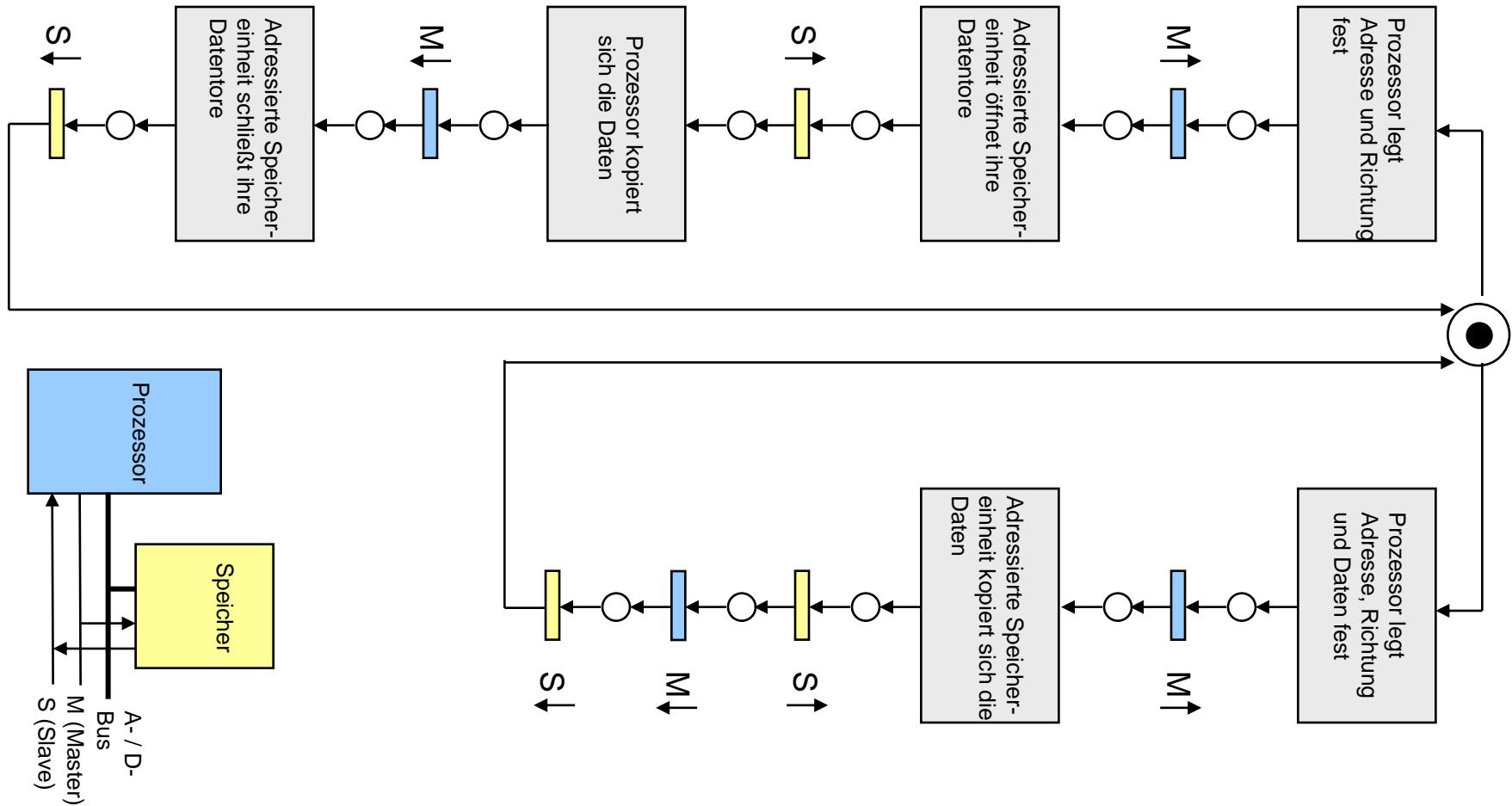
- Die zeitlichen Abläufe am Bus werden durch **Handshake-Signale** gesteuert.
- Der Systemtakt spielt keine Rolle mehr für die Synchronisation der Übertragung. Beim asynchronen Bus wird jeder Bustransfer durch die Reaktionsgeschwindigkeit der angesprochenen Einheit bestimmt.

Asynchroner Systembus

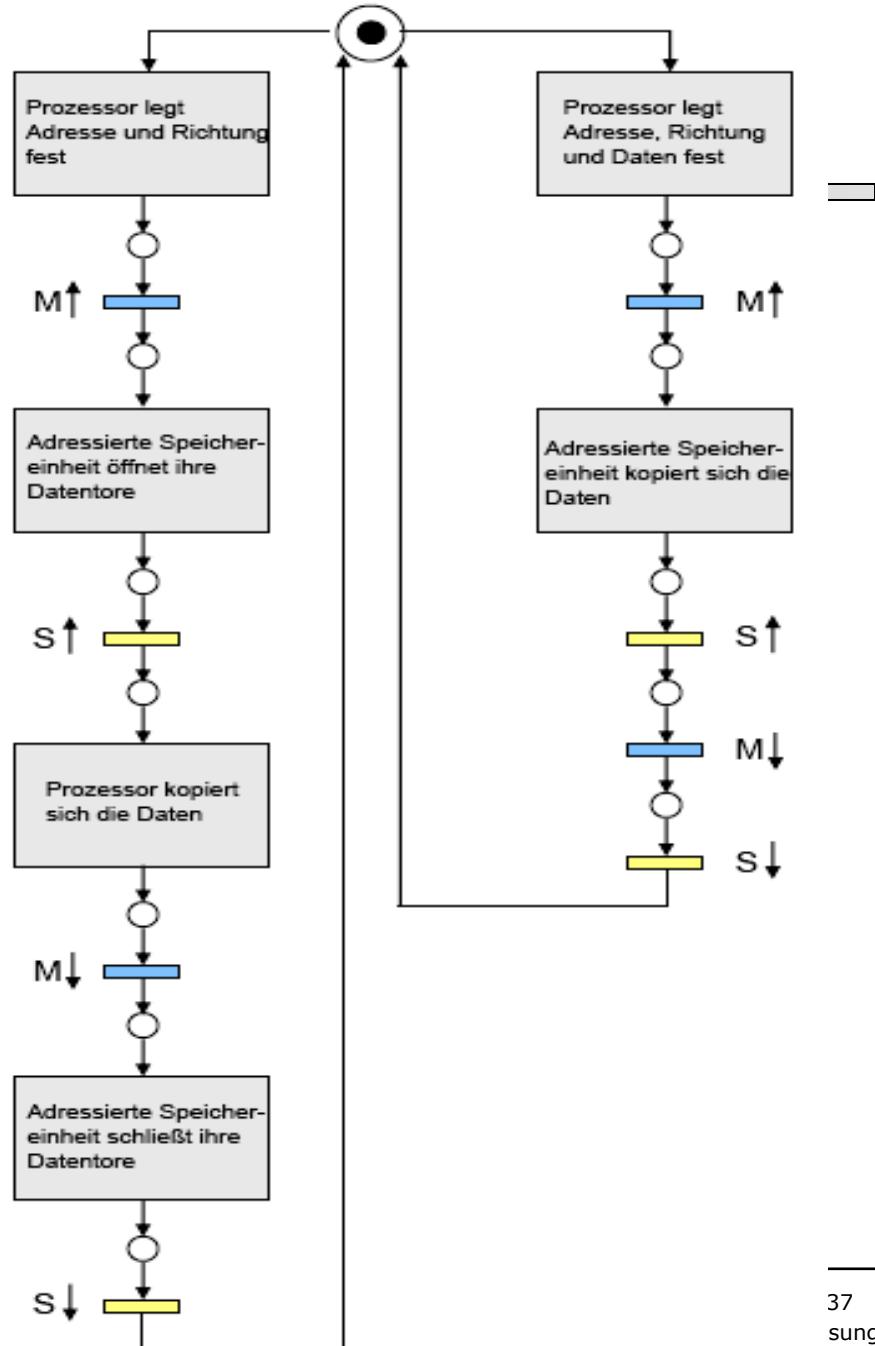
Beispiel für asynchrones Busprotokoll



Kommunikationsprotokoll am asynchronen Systembus



Kommunikationsprotokoll am asynchronen Systembus



Bushierarchie in Rechnern

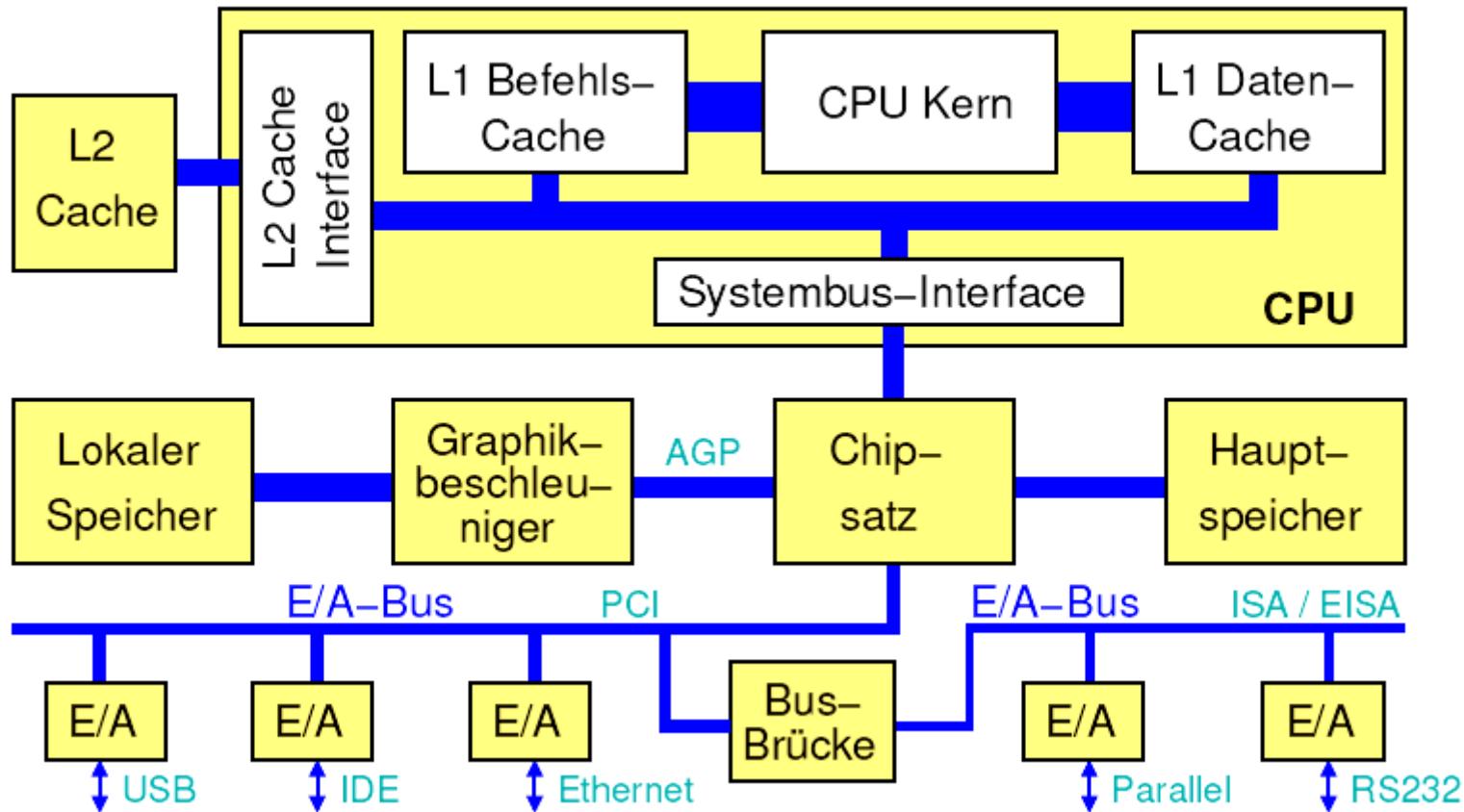
- Heutige Rechner: **Hierarchische Busorganisation**
 - Schnelle, breite Busse zu Caches / Speicher
 - Langsamere Busse für Anbindung von E/A-Geräten
 - Verbindung der Busse über Busbrücken

Insbesondere bei PC-Architekturen:

- Prozessorspezifische Chipsätze realisieren
 - Speicheranbindung (DRAM-Controller)
 - Brücken zu Peripheriebussen

Bushierarchie in Rechnern

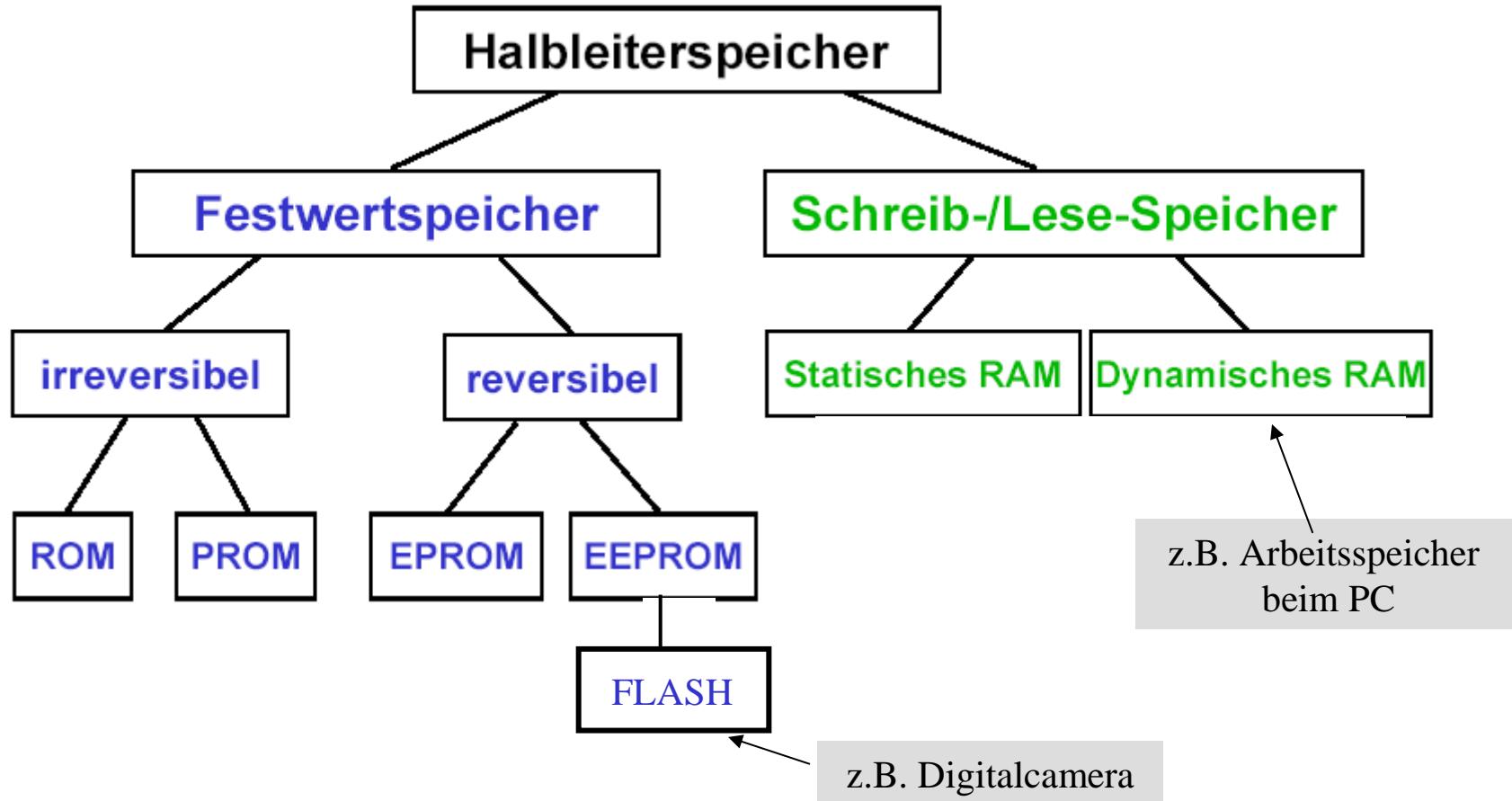
Beispiel: Bushierarchie in einem PC



Aufbau und Organisation des Arbeitsspeichers

Zugriffsmethoden auf den Speicher

Klassifizierung von Halbleiterspeichern



Schreib-/Lesespeicher (RAM)

Inhalt ist jeder Zeit lesbar und schreibbar.

Inhalt ist flüchtig (*volatile*) , d.h. er geht bei Abschalten der Versorgungsspannung verloren.

□ **Statische Schreib/Lese-Speicher (SRAM)**

Flipflops als Speicherzellen, Inhalt stabil, solange Versorgungsspannung vorhanden ist

□ **Dynamische Schreib/Lese-Speicher (DRAM)**

Information wird als elektrische Ladung in einem Kondensator gespeichert.

Das Lesen bewirkt meist Entladung (*destructive read*)

► nach dem Lesen muss wieder eingeschrieben werden

Die Ladung geht nach einiger Zeit auch durch Leckströme verloren ► periodische Auffrischung erforderlich (*refresh*)

Aufbau und Organisation des Arbeitsspeichers

□ Kapazität:

Informationsmenge (in Anzahl Bit), die im Speicher untergebracht werden kann: **n • m Bit**

□ Größen zur Charakterisierung der Arbeitsgeschwindigkeit eines Speicherbausteins:

⇒ Zugriffszeit (*access time*)

maximale Zeitdauer, die vom Anlegen einer Adresse an den Speicher bis zur Ausgabe der gewünschten Daten vergeht

⇒ Zykluszeit (*cycle time*):

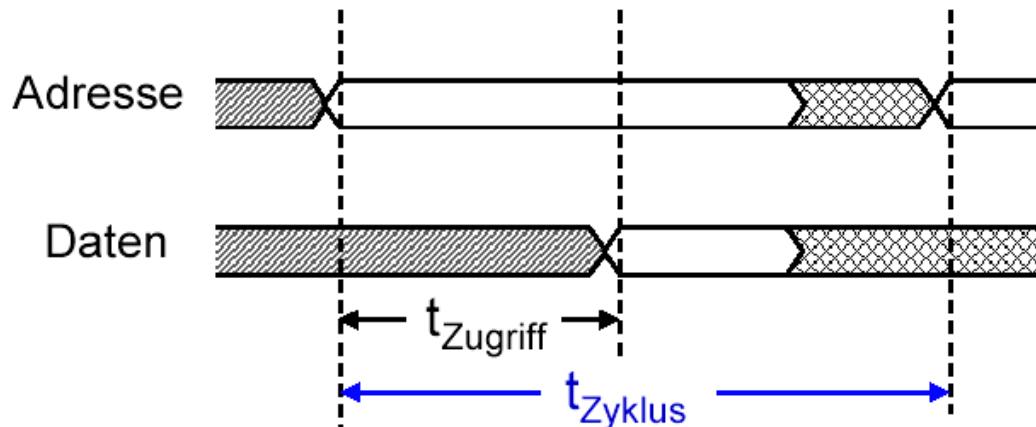
minimale Zeitdauer, die zwischen zwei hintereinander folgenden Aufschaltungen von Adressen an den Speicher vergehen muß

Zugriffszeit / Zykluszeit

Die Zykluszeit kann erheblich länger als die Zugriffszeit sein!

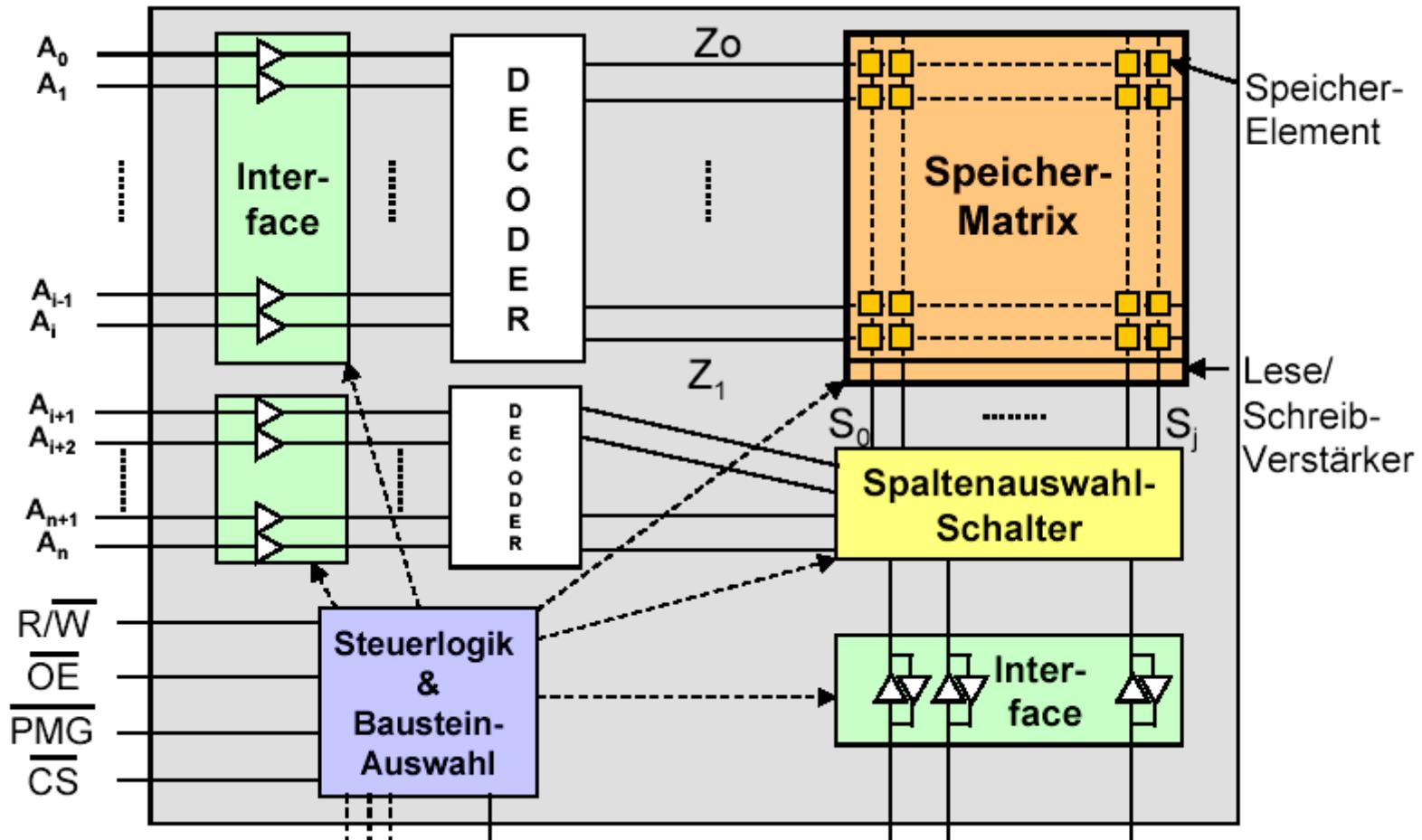
Gründe:

- ⇒ Speicherzelle muß sich nach einem Zugriff "erholen"
- ⇒ Bei einigen Speicherarten wird die Information durch das Auslesen zerstört und muß erst wieder eingeschrieben werden (refresh)



Idealfall: Zykluszeit = Zugriffszeit

Organisation von Speicherbausteinen (Prinzip)



Organisation von Speicherbausteinen

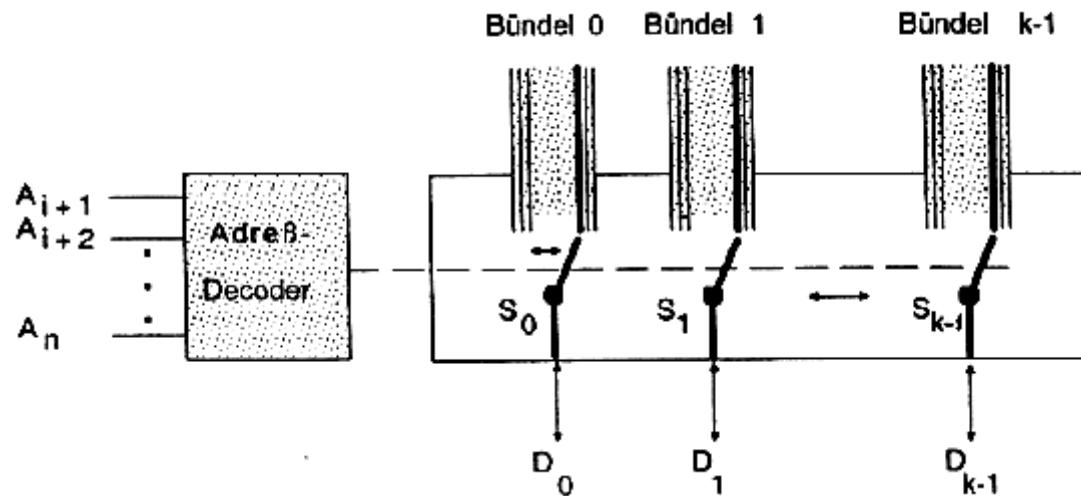
Gewinnung der Zeilen- und Spaltenauswahlleitungen aus den Adressleitungen:

Durch die Auswahl einer ganzen Zeile in einer möglichst quadratischen Matrix werden i.A. erheblich mehr Speicherzellen angesprochen als aktuell ausgelesen werden müssen.

- ➡ Auswahl der anzusprechenden bzw. auszulesenden Spalten durch einen Spaltenauswahl-Schalter
Dieser wird über Treiber und Decoder von den höherwertigen Adressbits gesteuert

Organisation von Speicherbausteinen

Funktionsweise der Spaltenauswahl-Schalter:



- ⇒ Zusammenfassung zu k Bündeln
(k = äußere Speicherbreite)
- ⇒ Auswahl über k synchron arbeitende Schalter
- ⇒ Ansteuerung über Adreßdekoder

Vergleich: Statische und dynamische RAMs

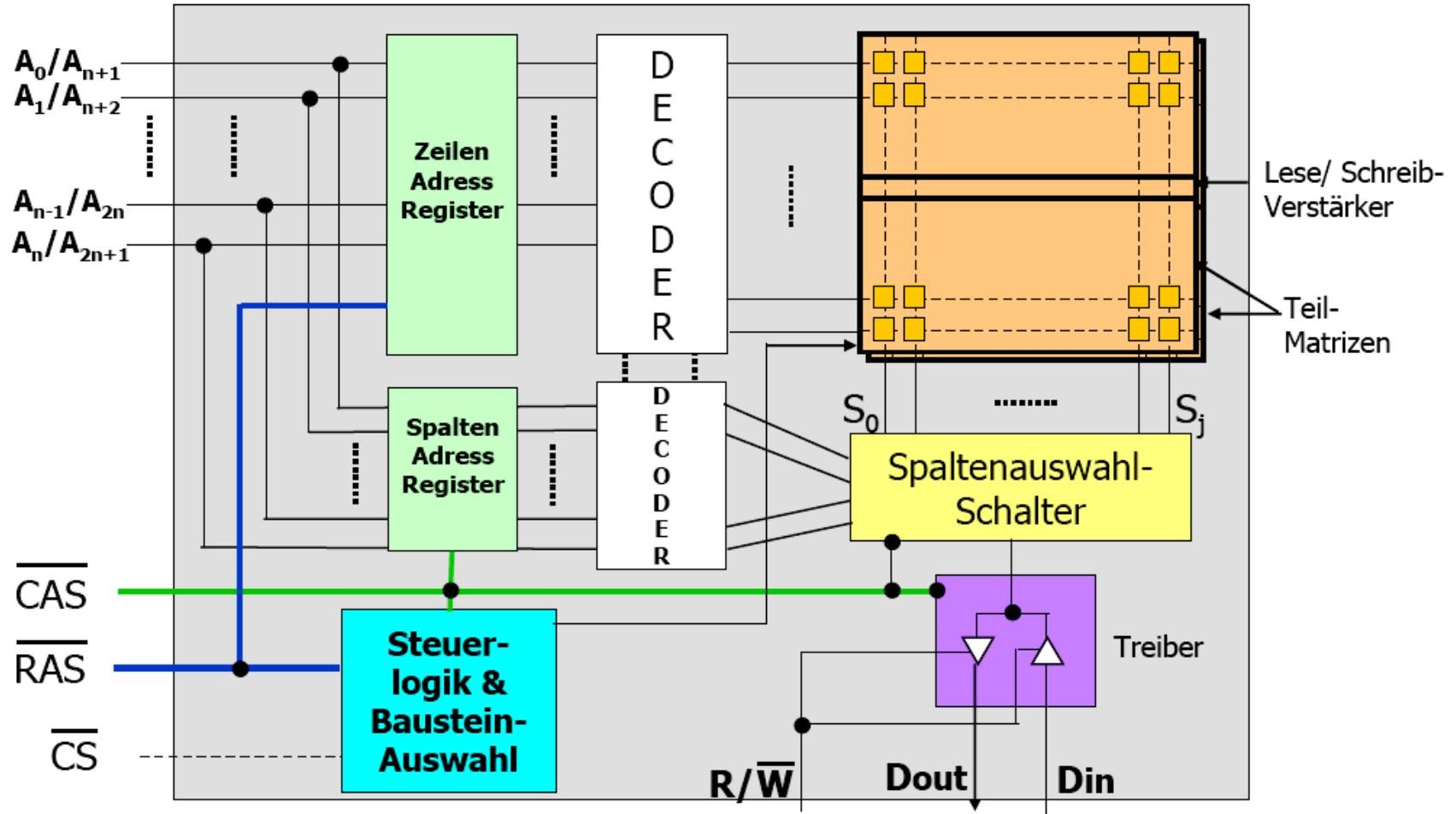
Statische RAMs

- sehr schneller Zugriff
- relativ teuer
- reduzierte Speicherkapazität
(bzgl. gleich großem DRAM)
- Zugriffszeiten: 5.....30 ns
- Speicherzellenmodell:
dauerhafte Speicherung -
Flipflop

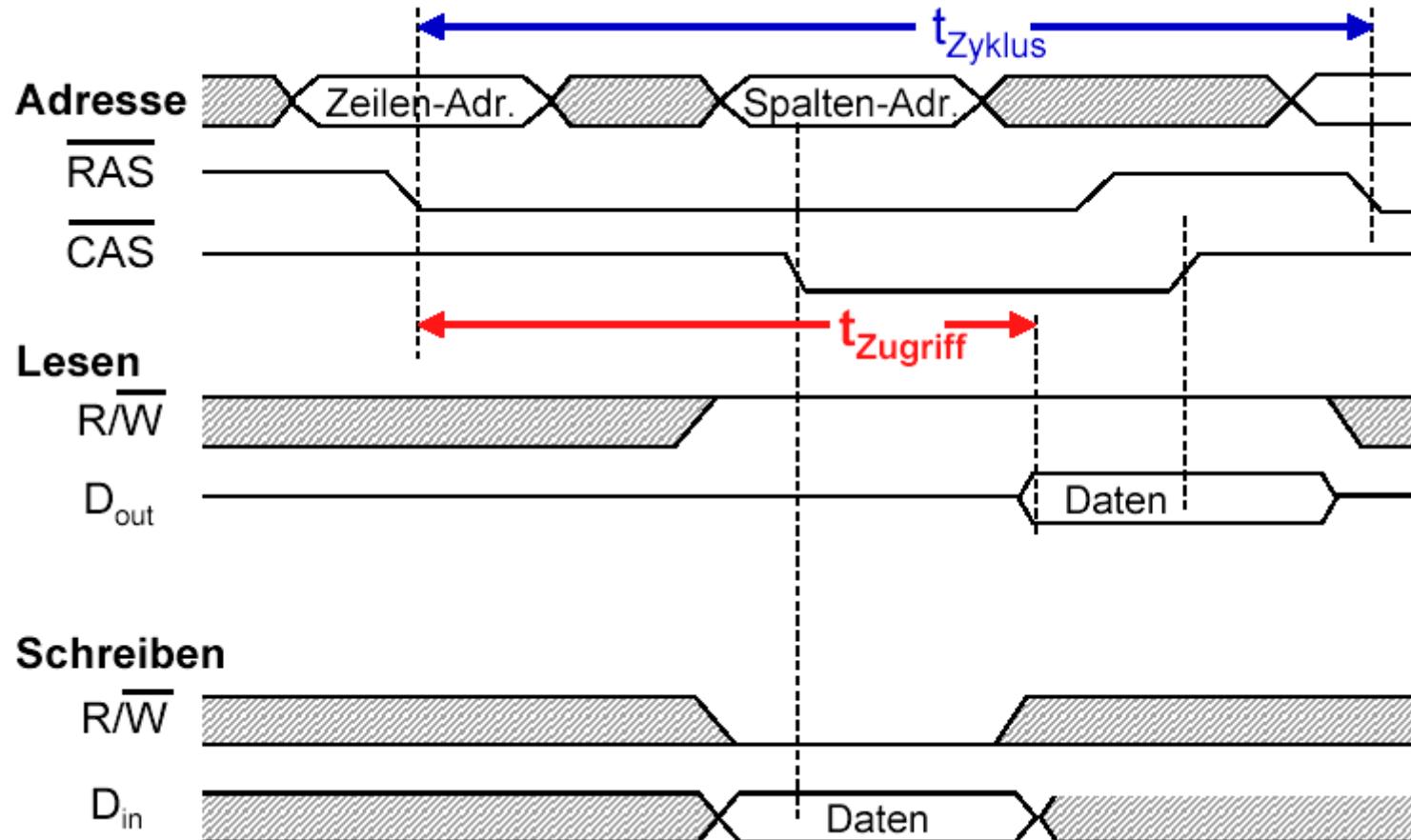
Dynamische RAMs

- langsamerer Zugriff
- kostengünstig
- große Speicherkapazität
(ca. 4 x SRAM)
- benötigen Erholzeit (Refresh)
- Zugriffszeit: ca. 40 ns,
Zykluszeit: ca. 80 ns
- Speicherzellenmodell:
kapazitive Ladungsspeicherung

Dynamische RAM - Bausteine



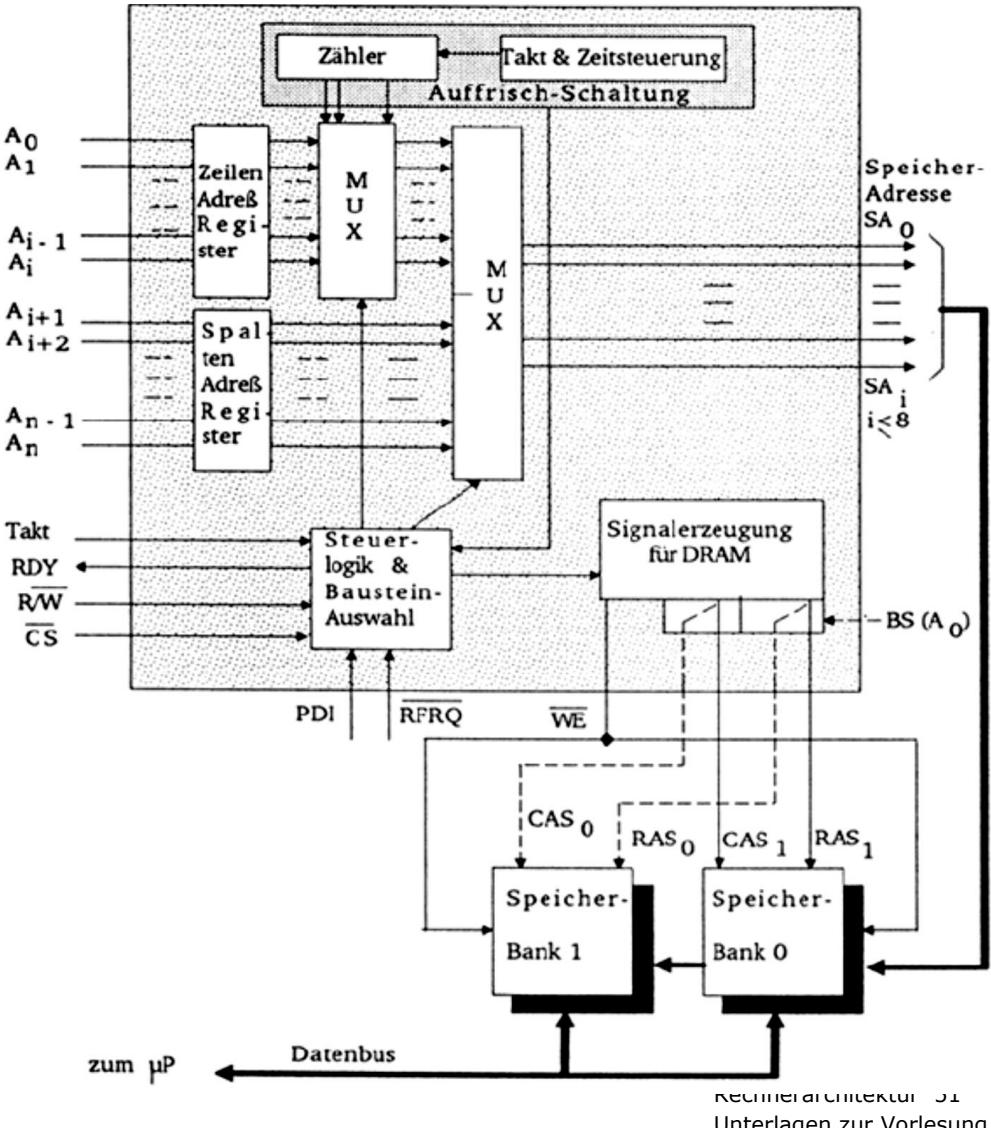
Adressierung eines dynamischen RAM-Bausteins



DRAM - Controller

Aufgabe:

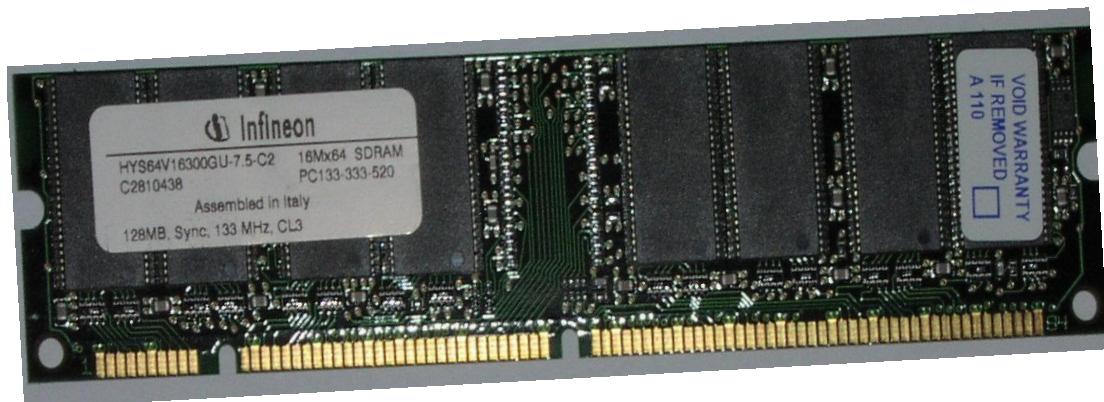
- ↳ Ansteuerung der DRAM-Bausteine
- ↳ Erzeugen der RAS- und CAS-Signale
- ↳ Multiplexen der Adressen
- ↳ Speicher-Refresh



Bausteintyp: DDR - SDRAM

□ DDR-SDRAM (*Double-Data-Rate-SDRAM*)

Die DDR-SDRAMs entsprechen in Bauform und Funktionsweise den "normalen" SDRAM-Modulen, jedoch werden im Gegensatz zu diesen die Speicherzellen zweimal pro Takt ausgelesen bzw. geschrieben. Dadurch erreichen die DDR-SDRAM Module den doppelten Datendurchsatz.



Speicherzugriffe durch Blockbuszyklen

- Heutige Prozessoren führen Datentransporte mit Speichern nicht mehr als Einzelübertragung (single cycles) sondern fast ausschließlich als 4-Langwort-Blockübertragung (4 x 32 Bit) mittels **Blockbuszyklen (burst cycles)** durch.
- Durch die Aufeinanderfolge der Langwortadressen muss der Prozessor nur noch die Adresse des ersten Langworts angeben. Die weitere Adressfortschaltung erfolgt dann prozessorextern (durch die Speicheransteuerung, Chipsatz) in kürzerer Zeit als der erste Langworttransport.
- Fall des minimalen Buszyklus (ohne wait states)
 - 2 Takte für den ersten Transport
 - 1 Takt für die Folgetransporte
 - → (2-1-1-1)-Burst
- Können Speicher dem Blockburst nicht so schnell folgen
 - Müssen Wartezyklen vom Prozessor eingelegt werden
 - Müssen Strukturmaßnahmen auf der Speicherseite erfolgen (z.B. Interleaving)

Prinzip des Interleaving

- Speicherseitige Strukturmaßnahme: alternierender Zugriff auf zwei oder mehr Speicherbänke. Diese Maßnahme bezeichnet man als **Verschränkung von Speicherbänken oder Interleaving**.

Beispiel (2-fache Verschränkung, Prinzip):

Datenbus (64 Bit)

Bank 1:
Worte 1,3,5, ...

Speicher
(64 Bit breit)

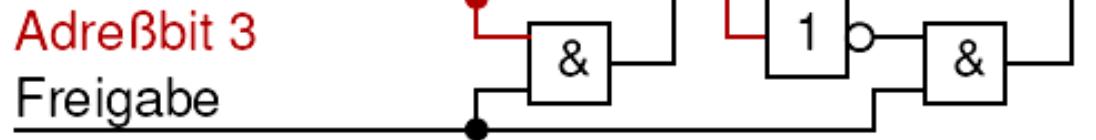
Speicher
(64 Bit breit)

Bank 0:
Worte
0,2,4,...

Adreßbus (Bits 4...n)

Adreßbit 3
Freigabe

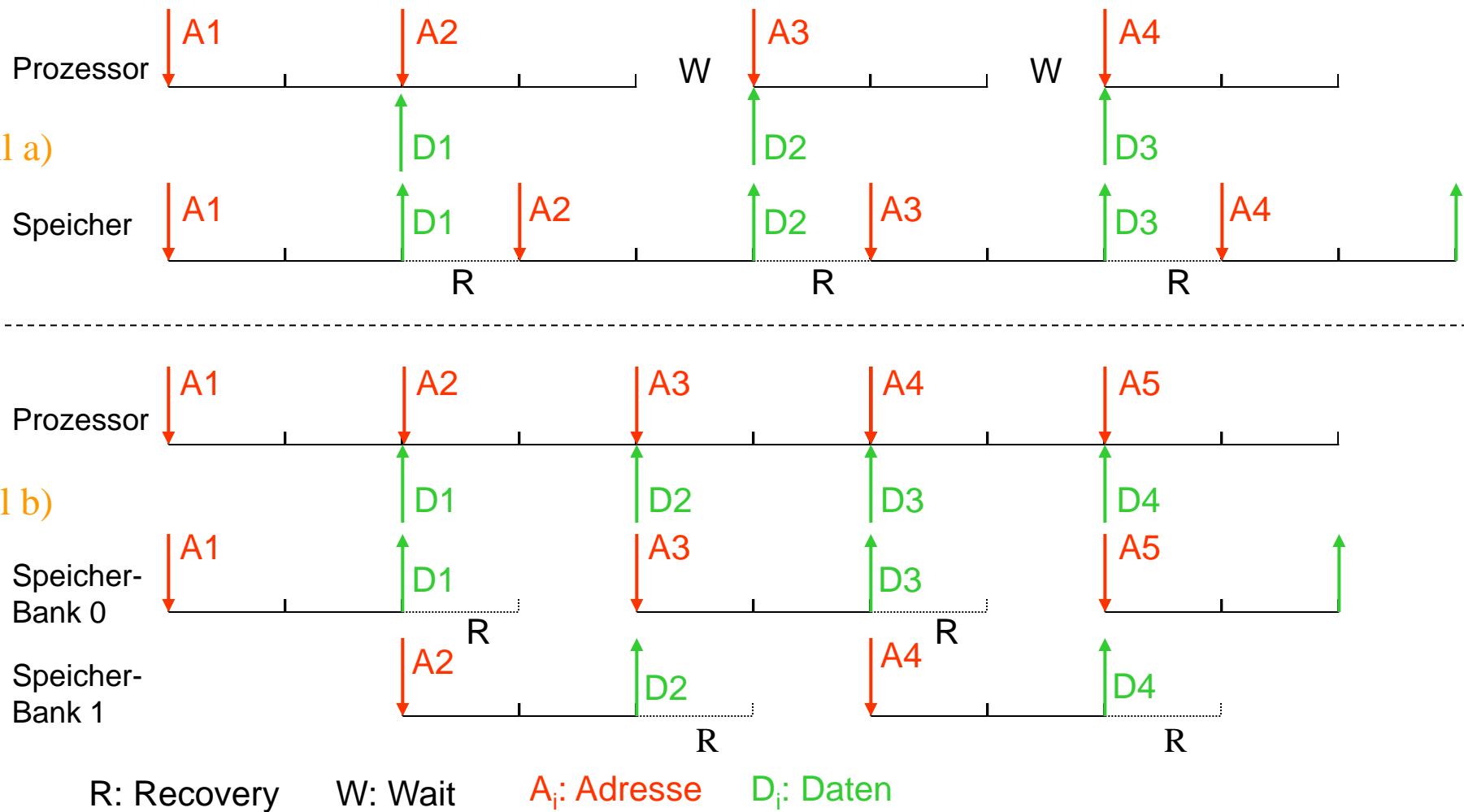
Freigabe



Prinzip des Interleaving

- Das **nachfolgende Beispiel** verdeutlicht den Bankwechsel bei zwei DRAM-Speicherbänken.
 - Im **Fall a)** werden direkt aufeinanderfolgende Lesezugriffe des Prozessors auf einen DRAM-Speicher betrachtet, bei nur einer Speicherbank.
 - Im **Fall b)** werden dieselben Zugriffe auf zwei verschränkt adressierte Speicherbänke betrachtet.
 - Es gelten die folgenden Voraussetzungen:
 - minimale Buszykluszeit des Prozessors: 2 Bustakte
 - Speicherzugriffszeit: 2 Bustakte
 - Speicherzykluszeit: 3 Bustakte

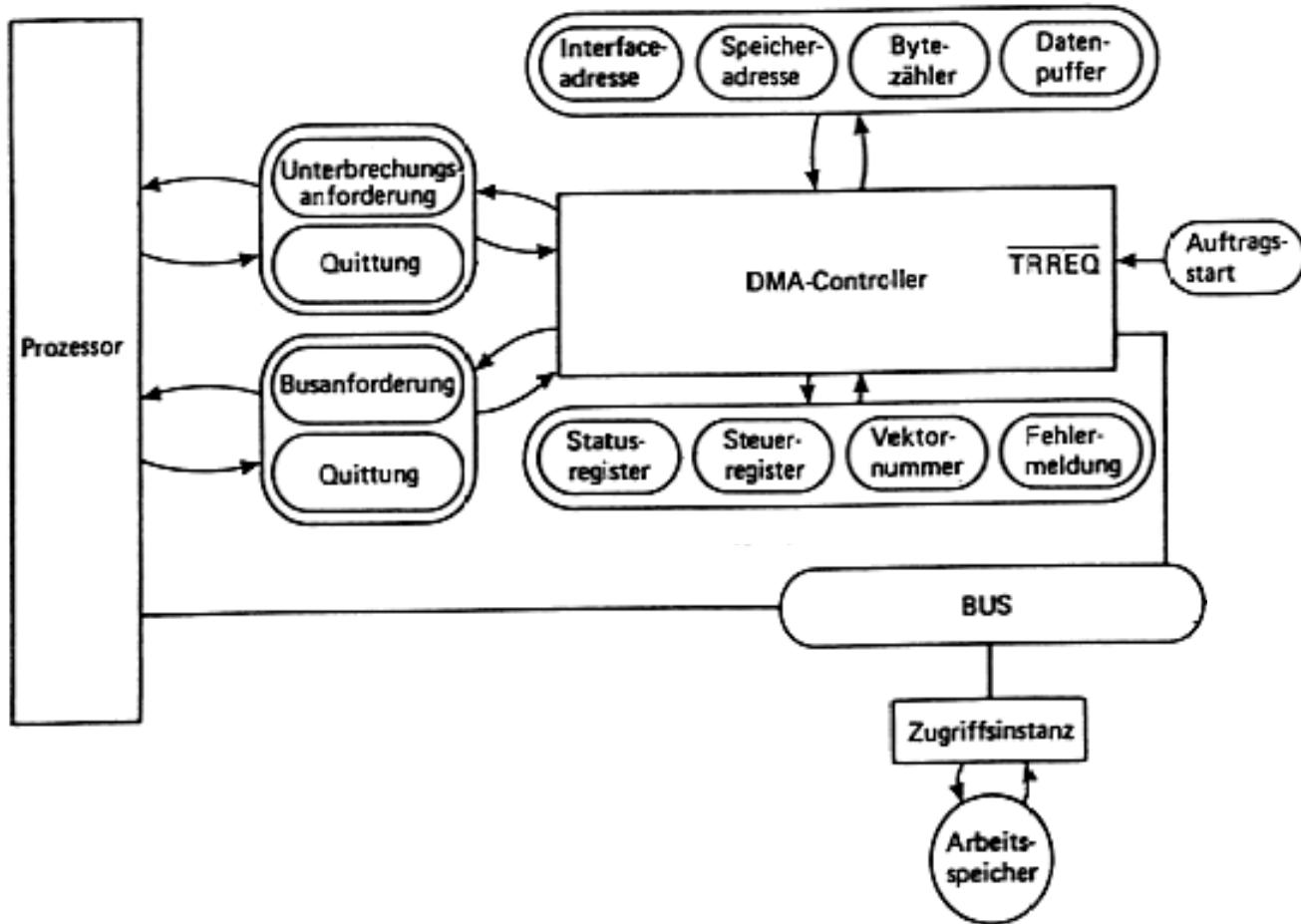
Prinzip des Interleaving



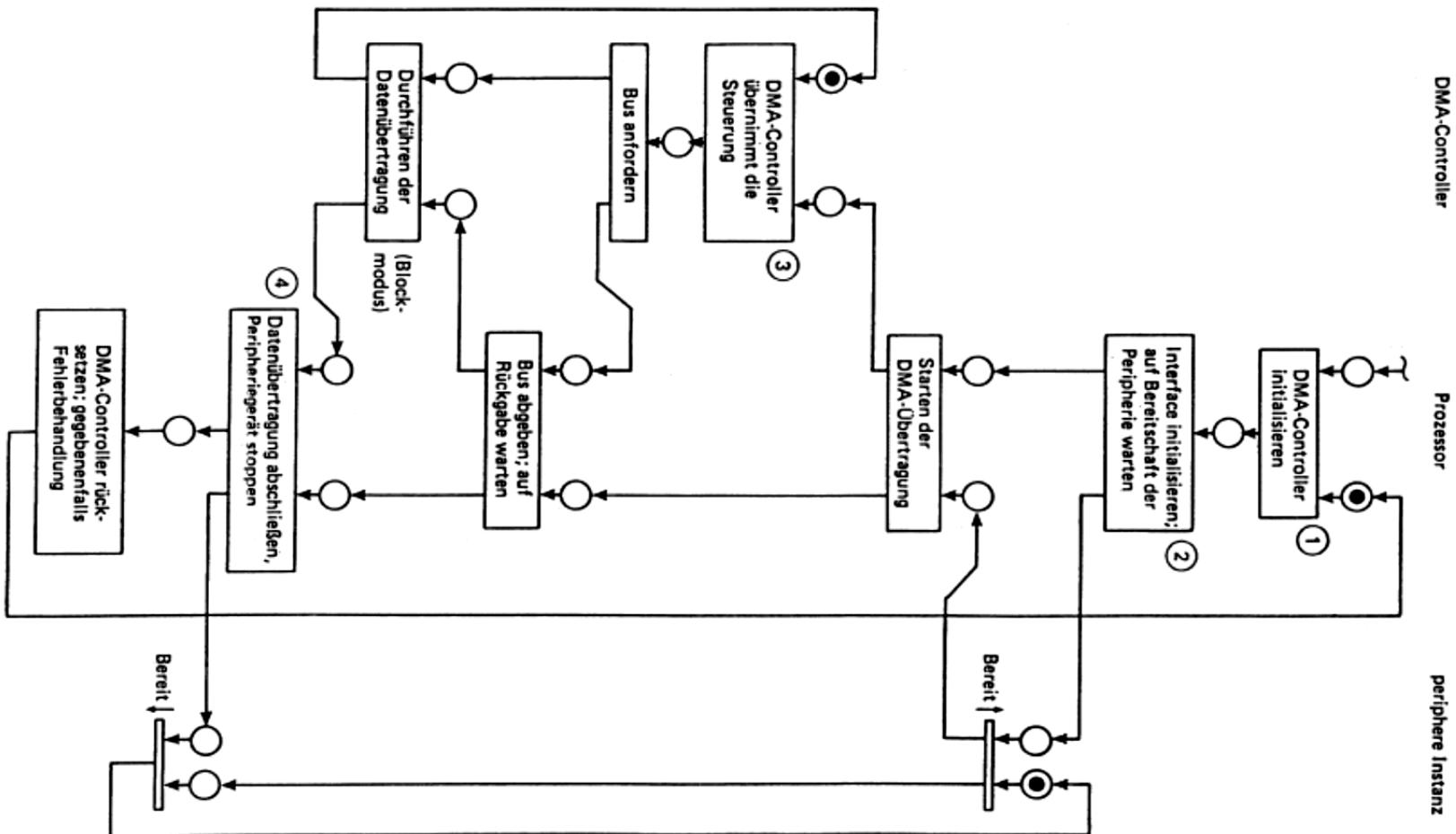
Direkter Speicherzugriff (DMA)

- Bisher erfolgte ein Datentransfer mit dem Speicher oder der Peripherie immer über den Prozessor.
 - **Nachteil:** Langsamer Transfer, Prozessor ist belastet
-
- Abhilfe: **Direkter Speicherzugriff (Direct Memory Access, DMA)**
 - Hierbei erfolgt ein Datentransfer direkt zwischen den beteiligten Komponenten, „ohne“ Beteiligung des Prozessors.
 - Dazu dient ein spezieller Baustein: **DMA-Controller**
 - Der DMA-Controller koordiniert den Datentransfer
 - **Einzeltransfer (single transfer mode)**
 - **Block-Transfer (burst mode)**
 - Der DMA-Controller muss dazu allerdings vom Prozessor konfiguriert werden

DMA - Controllerbaustein



DMA - Kommunikationsprotokoll

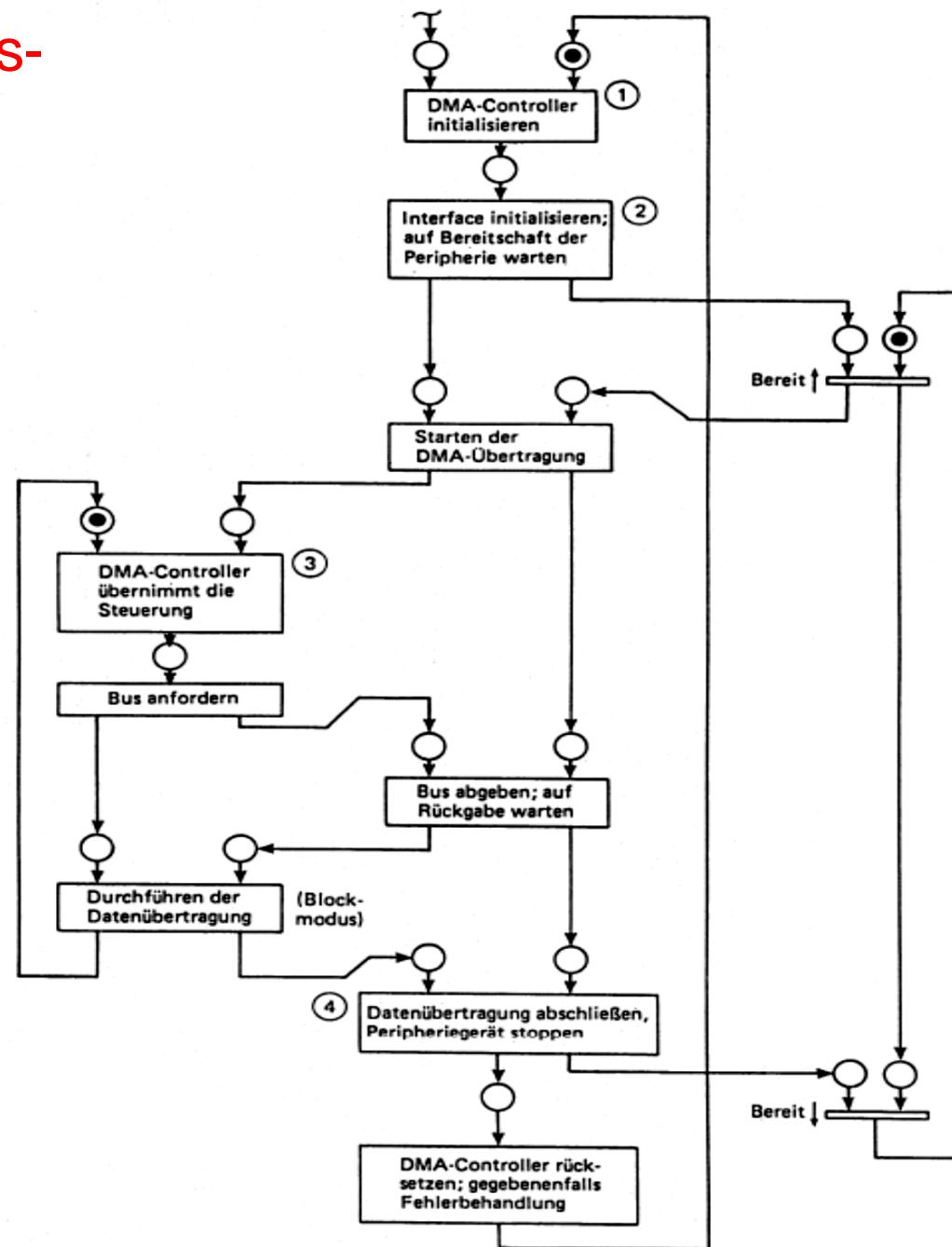


DMA – Kommunikations-protokoll

DMA-Controller

Prozessor

periphere Instanz



Memory - Mapped und Isolierte Adressierung

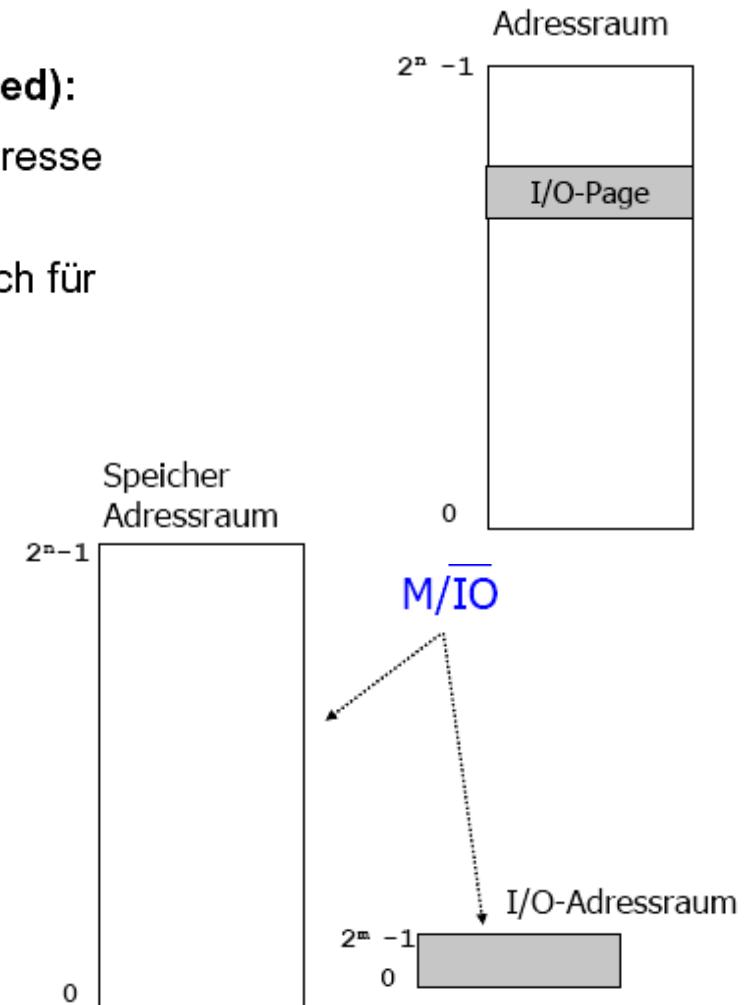
□ Speicherbezogene Adressierung (memory-mapped):

kein Unterschied zwischen Speicheradresse und Adresse eines Registers eines Peripherie-Bausteins,
häufig wird ein zusammenhängender Speicherbereich für Peripherie-Bausteine verwendet: I/O-Page

□ Isolierte Adressierung:

getrennte Adressräume für Speicher und Peripherie (eigener I/O-Adressraum)

Auswahl des Adressraums durch M/IO-Signal (memory/input-output)



CISC- und RISC-Prozessoren

Pipeline - Verarbeitung

RISC und CISC - Befehlssätze

- Grundsätzliche Frage: Wie soll ein Befehlssatz aussehen?
- Komfortabel und relativ mächtig
 - CISC: Complex Instruction Set Computer
 - Vorteil: einfache Programmierbarkeit
 - Nachteile: komplexe (langsame) Implementierung,
 - evtl. viele Funktionen ungenutzt
- Minimalistisch, auf das absolut notwendige beschränkt
 - RISC: Reduced Instruction Set Computer
 - Vorteil: einfache, effiziente, schnelle Implementierung
 - Nachteil: schwierigere Programmierbarkeit
 - Aber: Das erledigt der Compiler

Entstehungsgründe für CISC-Rechner

Entstehungsgründe für umfangreiche Maschinenbefehlssätze:

- Geschwindigkeitsunterschied zwischen CPU und Hauptspeicher
 - möglichst viel innerhalb des Prozessors rechnen
- Mikroprogrammierung
 - Flexibilität führte zu immer neuen Befehlsstrukturen
- Kompakter Code
 - Komplexe Befehle führten zu kürzeren Programmen
- Unterstützung höherer Programmiersprachen
 - Verkleinerung der semantischen Lücke
- Aufwärtskompatibilität
 - Familienkonzept der Rechnerhersteller
- Marktstrategie
 - alte Programme müssen weiterhin unterstützt werden

CISC: Grenzen der Weiterentwicklung

Grenzen der Weiterentwicklungs möglichkeiten bei CISC-Rechnern:

- ⇒ Schnellere Speicher
- ⇒ Ausnutzung nur eines Teils des Maschinen-Befehlssatzes
- ⇒ Verlängerte Entwurfszeit
- ⇒ Sehr komplexe Steuerwerke (> 50% der Chipfläche)
- ⇒ Sehr umfangreiche Mikroprogramme
- ⇒ Größere Fehlerhäufigkeit auf der Mikroprogramm ebene
- ⇒ Schwieriger Compilerbau

Befehlsausnutzung und CPI

Systemprogramme in XPL auf IBM/360:

90 % aller ausgeführten Befehle: 10 verschiedene Befehle

95 % aller ausgeführten Befehle: 21 verschiedene Befehle

99 % aller ausgeführten Befehle: 30 verschiedene Befehle

COBOL-Programme auf IBM/370:

90,28 % aller ausgeführten Befehle: 26 verschiedene Befehle

99,08 % aller ausgeführten Befehle: 48 verschiedene Befehle

(nur 84 verschiedene Befehle wurden überhaupt benutzt)

CISC:

Benutzungshäufigkeit von Befehlen



▫ Befehlsausnutzung (80/20 Regel):

viele mächtige Befehle, komplexes Befehlsformat,
Mikroprogrammierung, nur 20 % der Befehle werden
überwiegend benutzt

▫ Kritisches Problem: Anzahl der Zyklen pro Instruktion (CPI)

bei allen heutigen CISC Architekturen ist CPI >> 2
MC 68030: CPI = 4-6, Intel 80386 CPI = 4-5

RISC-Entwurfsziele

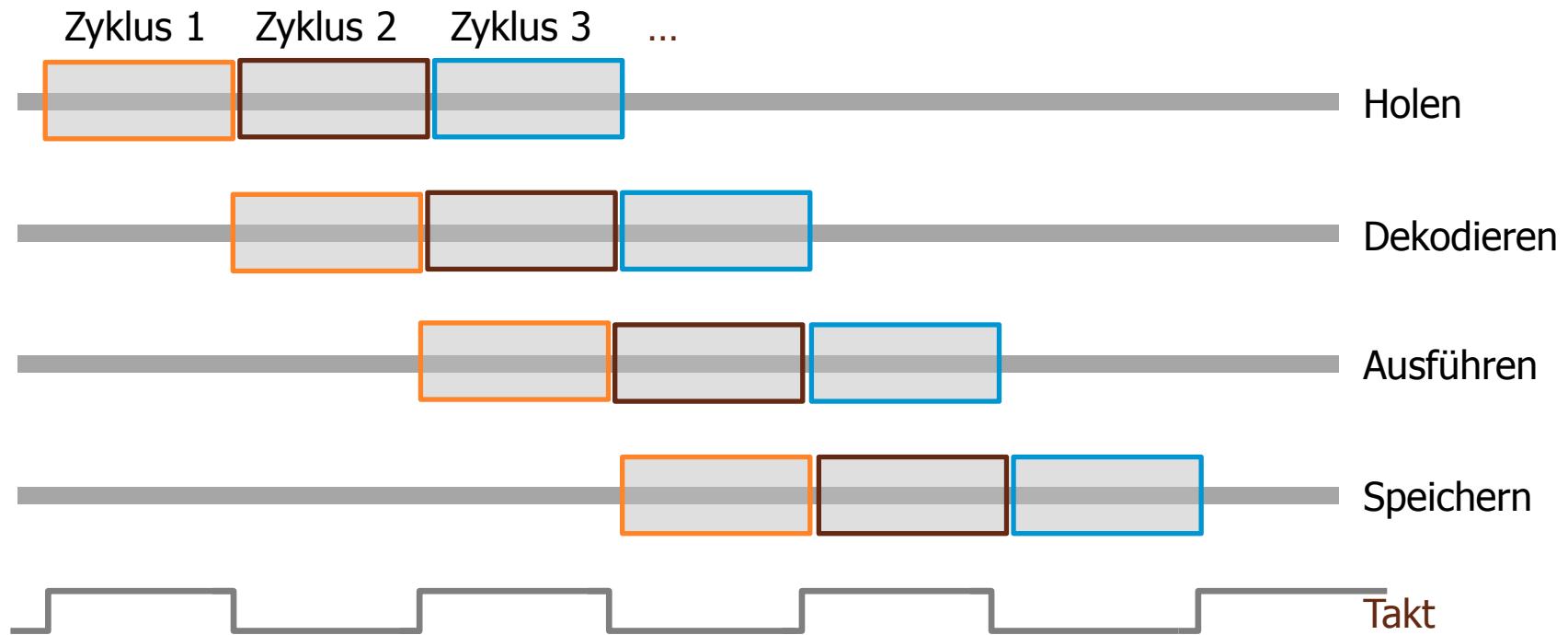
- Befehle so einfach wie möglich gestalten, einheitliche Befehlsformate garantieren **schnelle Decodierung**
- Ausführung eines Befehls in **einem Taktzyklus**
- Programme werden länger, Ausführungszeit wird trotzdem kürzer
- Nur **Load-Store Befehle** und **Register-Registerbefehle**
- Operanden werden möglichst in großen Registersätzen gehalten, garantiert schnellen Zugriff
- **Keine Mikroprogrammierung**, HW-Decodierung der Befehle
- **Pipelining** anwenden, wo immer möglich
- Konflikte sollen möglichst durch Compiler behoben werden
- Coprozessoren für komplexe Befehle

Befehlszyklus eines RISC - Prozessors

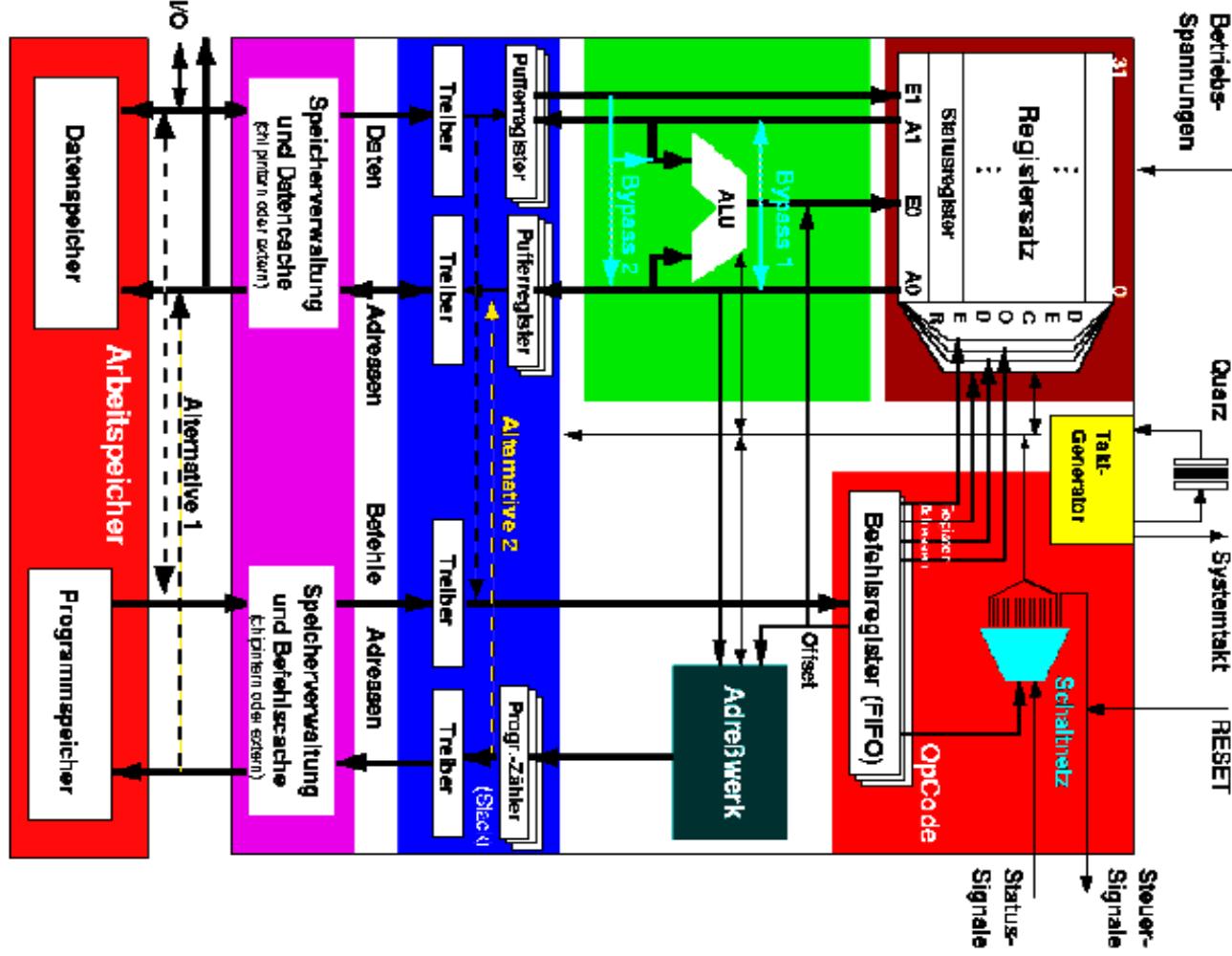
Reduced Instruction Set Computer

- Jede Operation = gleiche Taktlänge
- Problematisch: Sprungbefehle

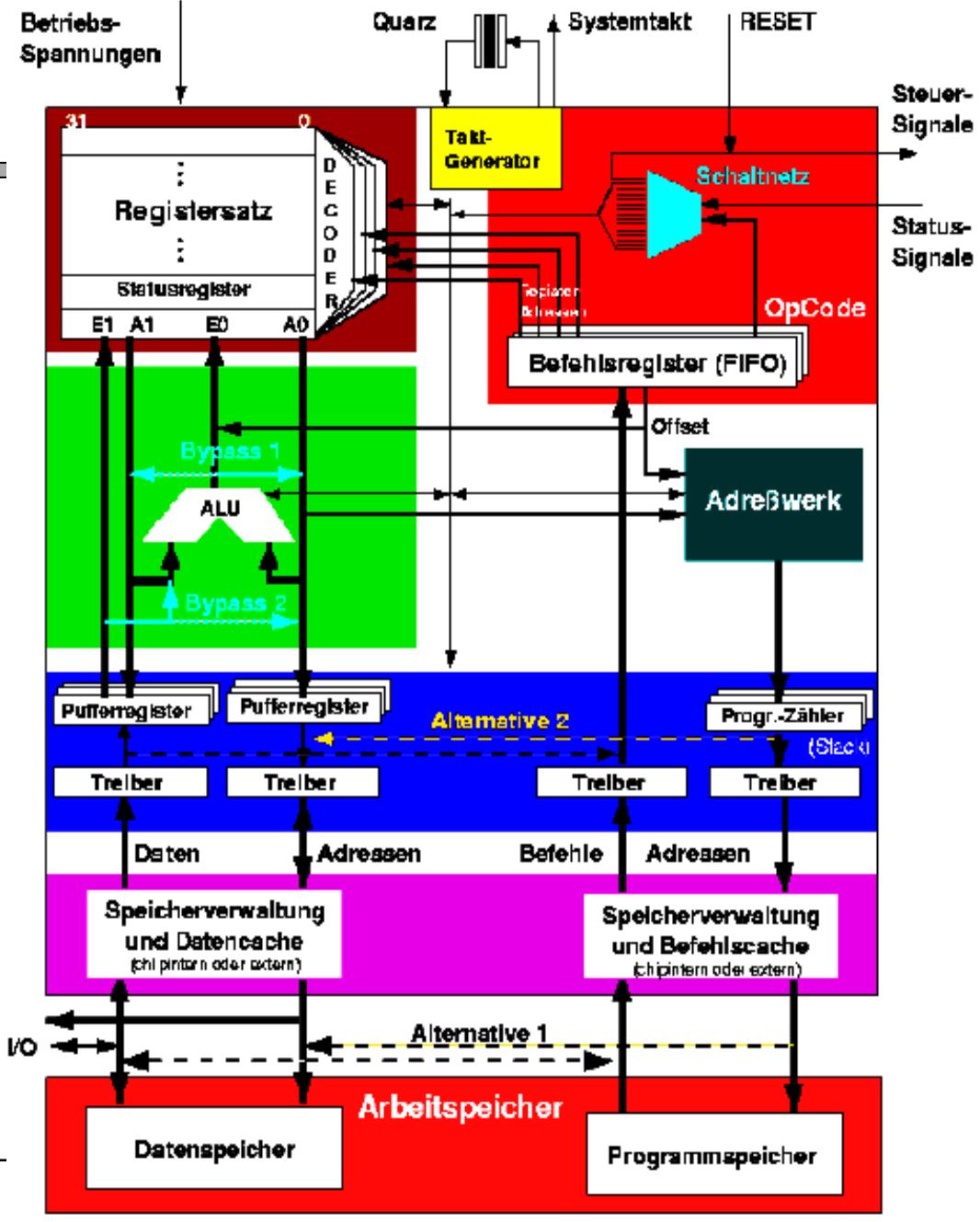
Einsatz: Tablet, Smartphone
ARM=*Advanced RISC Machines.*



Aufbau eines RISC - Prozessors



Aufbau eines RISC - Prozessors



Unterschiede zwischen RISC- und CISC- Prozessoren

Havard Architektur:

getrennter Programm- und Datenspeicher, deshalb
zwei Adress- und Datenbusse

► paralleles Holen von Operanden und Instruktionen

Vereinfachende Varianten:

1. zwei getrennte Bussysteme bis zu den Cache-Speichern, jedoch nur ein Arbeitsspeicher (niedrigere Kosten)
2. nur ein Bussystem wie bei Standard-Mikroprozessoren

Unterschiede zwischen RISC- und CISC- Prozessoren

Systembusschnittstelle:

enthält Registerblocks sowohl für Daten als auch für Adressen (gleichzeitiges Lesen eines Datums und Zwischenspeichern eines Ergebnisses)

Programmzähler:

ist manchmal als Hardware-Stack ausgebildet
(beschleunigt Unterprogrammaufrufe)

Unterschiede zwischen RISC- und CISC- Prozessoren

Steuerwerk:

- ⇒ festverdrahtet
- ⇒ Das Befehlsregister als Warteschlange (FIFO) realisiert
- ⇒ Für jede Pipeline-Stufe ist dort ein Register vorhanden
- ⇒ Die OpCodes jeder Stufe können vom Schaltnetz des Steuerwerks ausgewertet werden

Registersatz:

- ⇒ besteht aus einer großen Zahl von Registern
- ⇒ erlaubt gleichzeitige Auswahl von 3 bis 4 Registern
(z.B. 4 Port Registersatz, gleichzeitiges Schreiben
(E0, E1) und Lesen (A0, A1) von jeweils 2 Registern)

Unterschiede zwischen RISC- und CISC- Prozessoren

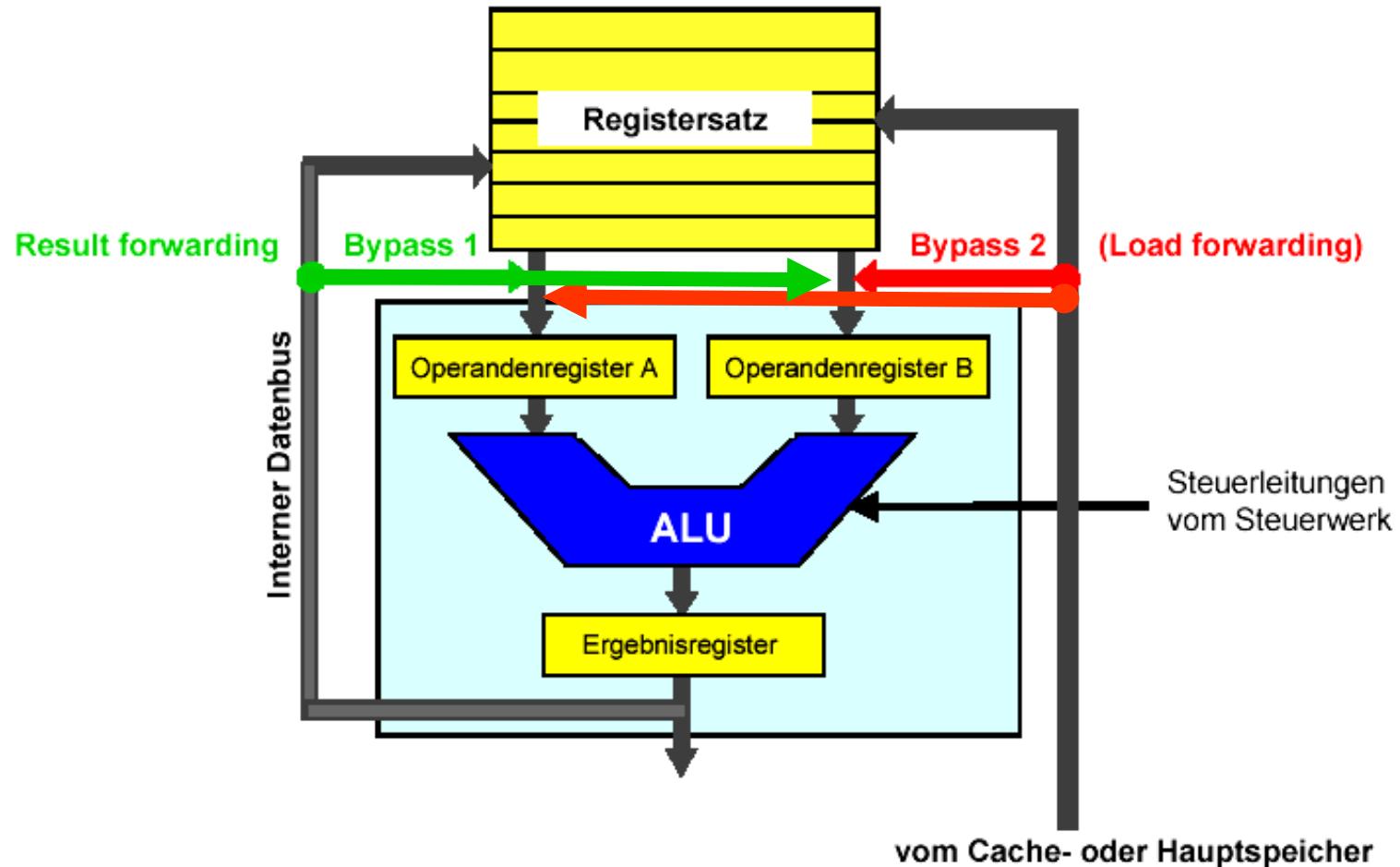
Rechenwerk:

besitzt eine Load/Store-Architektur. Die Operanden werden über 2 Operandenbusse aus dem Registersatz herbeigeführt, das Ergebnis (noch im selben Taktzyklus) über den Ergebnisbus in den Registersatz geschrieben.

Normalerweise gibt es keine direkte Verbindung zwischen ALU und Systemdatenbus, Datentransfer läuft über die Register (Load/Store-Architektur)

Ausnahme: Register-Bypasses zur Vermeidung von Pipeline-Hemmrisiken (forwarding techniques)

Bypass - Techniken



Unterschiede zwischen RISC- und CISC- Prozessoren

□ Bypass 1

Folgen in der Pipeline zwei Befehle direkt hintereinander, bei denen das Ergebnis der Vorgänger-Operation als Operand der Nachfolger-Operation benötigt wird

- ⇒ Zwischenspeichern im Registersatz würde den Ablauf verzögern
- ⇒ Bypass 1 erlaubt gleichzeitig zum Abspeichern des Ergebnisses auch dessen direkte Rückkopplung zum Eingang der ALU (result forwarding)

Unterschiede zwischen RISC- und CISC- Prozessoren

□ Bypass 2

Wurde ein Operand einer Pipeline-Operation erst im unmittelbar vorangehenden Taktzyklus vom Speicher in den Registersatz übertragen

- ⇒ der Weg über den Registersatz würde ebenfalls eine Verzögerung bedeuten
- ⇒ Bypass 2 erlaubt ein aus dem Speicher in ein Register geladenen Operanden gleichzeitig auch an den ALU-Eingang zu führen (load forwarding)

Befehlsverarbeitung in RISC- Prozessoren

Einfacher Befehlssatz von RISC-Prozessoren

- Maschinenprogramme sind länger als bei CISC Prozessoren

(komplexe Befehle und Adressierungsarten müssen aus den einfachen RISC-Befehlen zusammengesetzt werden)

Trotzdem arbeitet ein RISC-Prozessor meist schneller als ein CISC-Prozessor. Der Grund liegt in der nahezu **vollständigen Parallelarbeit aller Komponenten** eines RISC-Prozessors (extreme Pipeline-Verarbeitung)

Es wird mit großer Wahrscheinlichkeit **in jedem Taktzyklus ein Befehl beendet**

Beispiel: A5 Prozessor Chip



- Zwei ARM9 Core Prozessoren
- Grafikprozessor mit 4 Kernen
- Interfaces zu
 - Speicher (DDR-SDRAM, *Double Data Rate Synchronous Dynamic Random Access Memory*)
 - I/O
 - WiFi
 - USB
 - Video/Audio

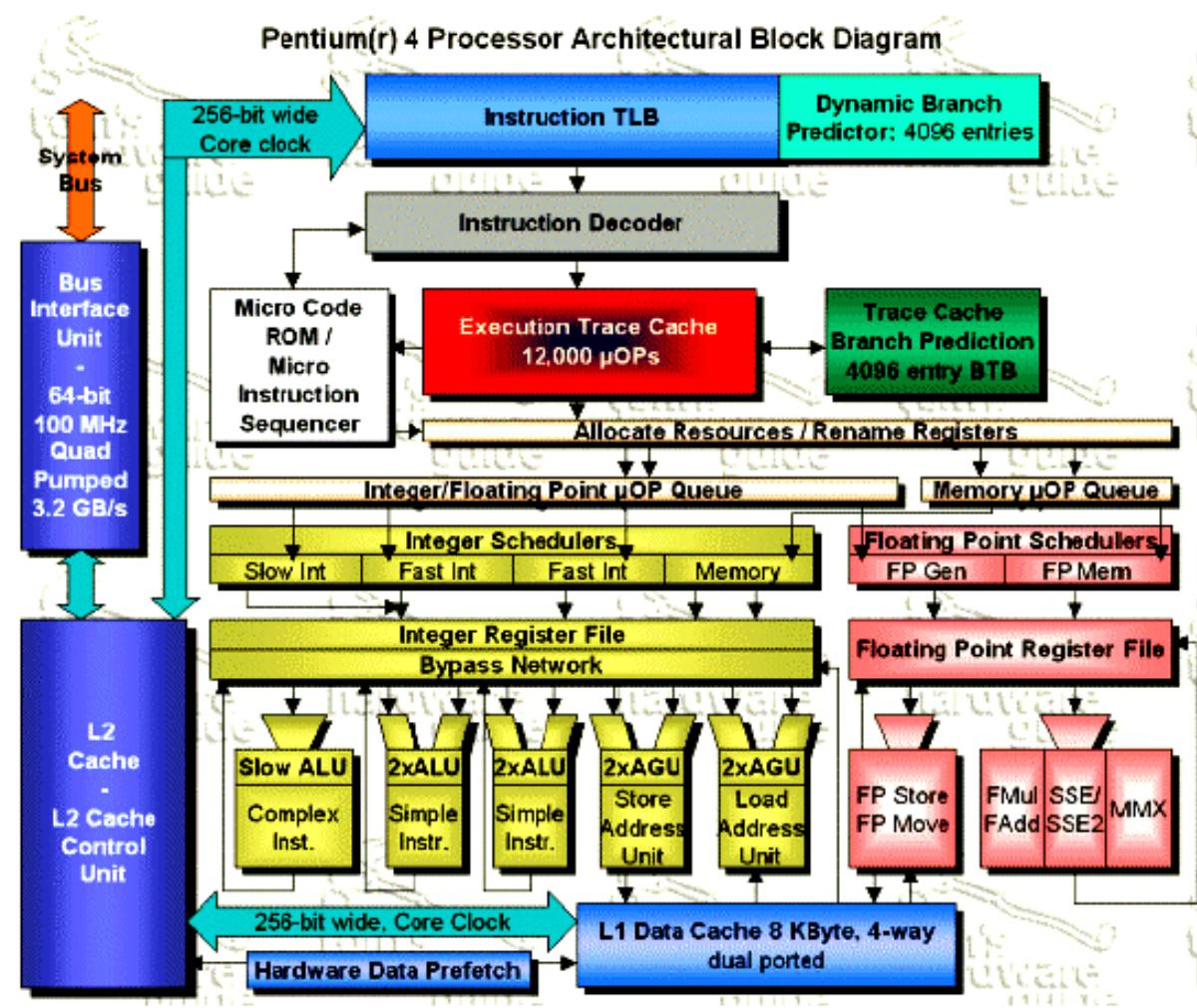
Gegenüberstellung RISC - CISC

CISC	RISC
Komplexe Befehle, Ausführung in mehreren Taktzyklen	Einfache Befehle, Ausführung in einem Taktzyklen
Jeder Befehl kann auf den Speicher zugreifen	Nur Lade- und Speicherbefehle greifen auf den Speicher zu
Wenig Pipelining	Intensives Pipelining
Befehle werden von einem Mikroprogramm interpretiert	Befehle werden durch festverdrahtete Hardware ausgeführt
Befehlsformat variabler Länge	Alle Befehle mit fester Länge
Die Komplexität liegt im Mikroprogramm	Die Komplexität liegt im Compiler
Einfacher Registersatz	Mehrere Registersätze

RISC - superskalar

- ❑ RISC-Prozessoren, die das Entwurfsziel von durchschnittlich einer Befehlsausführung pro Takt (CPI – *cycles per instruction* oder IPC – *instructions per cycle* von eins) erreichen, werden als **skalare RISC-Prozessoren** bezeichnet.
- ❑ Die Superskalar-Technik ermöglicht es heute, pro Takt bis zu vier Befehle den Ausführungseinheiten zuzuordnen und eine gleiche Anzahl von Befehlsausführungen pro Takt zu beenden.
- ❑ Solche Prozessoren werden als **superskalare (RISC)-Prozessoren** bezeichnet, da die oben definierten RISC-Charakteristika auch heute noch weitgehend beibehalten werden.
- ❑ Heutige Mikroprozessoren nutzen Befehlsebenenparallelität durch die Pipelining- und Superskalartechnik.

RISC - superskalar



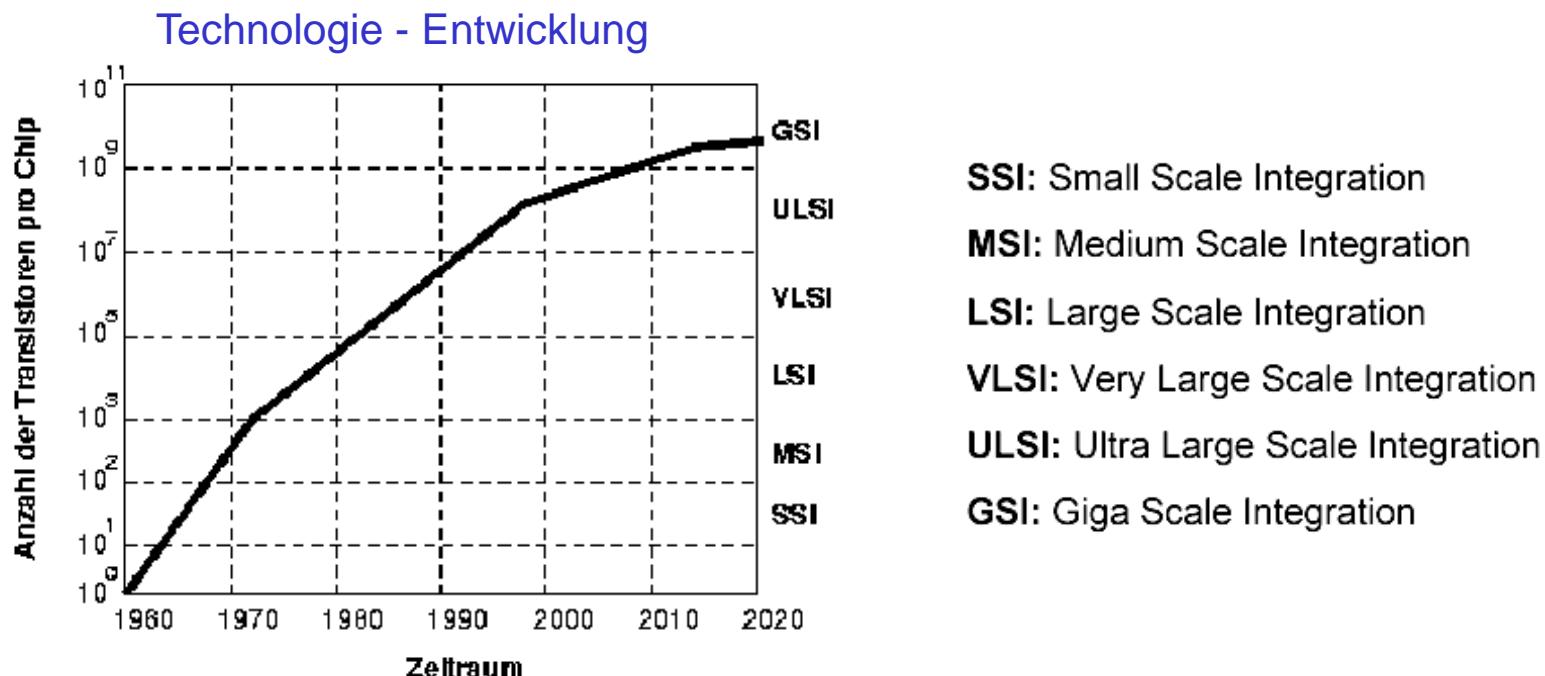
Leistungssteigerung in Rechnern

Pipelining

Leistungssteigerung

- Leistungssteigerung von Prozessoren

- Durch Halbleitertechnologie (Erhöhung der Taktfrequenz)
- Durch Mikroarchitektur (Verringerung des CPI-Werts)

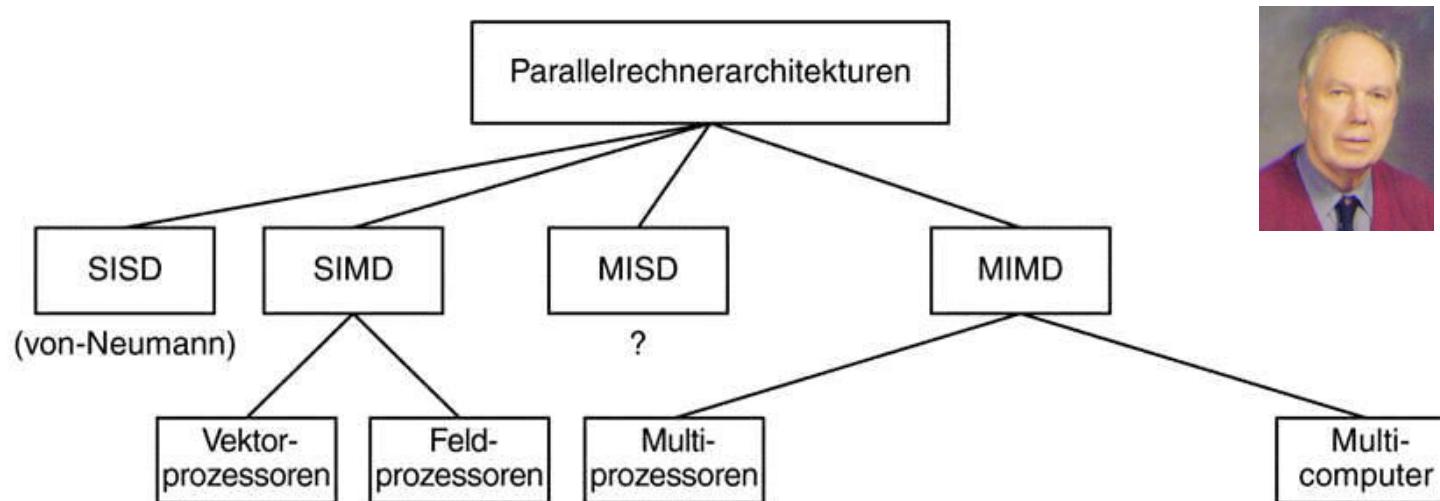


Leistungssteigerung

- Techniken zur Parallelverarbeitung in Prozessoren
 - Pipelining, superskalare Prozessoren
 - ISA: ein sequentieller Befehlsstrom
 - VLIW, EPIC
 - ISA: ein sequentieller Strom parallel ausführbarer Befehlsgruppen
 - Multithreaded Prozessoren
 - mehrere Befehlsströme

Strukturelle Maßnahmen

- Klassifizierung von Parallelrechnern nach Flynn
 - Konzept: Befehlsströme und Datenströme, die gewissermaßen als unabhängig voneinander betrachtet werden.
 - Z.B. ein System mit n CPUs besitzt n Befehlszähler und somit n Befehlsströme. Ein Datenstrom besteht aus einer Operandenmenge.
 - Im unteren Bild ist die Flynn'sche Klassifikation um eine Ebene erweitert worden



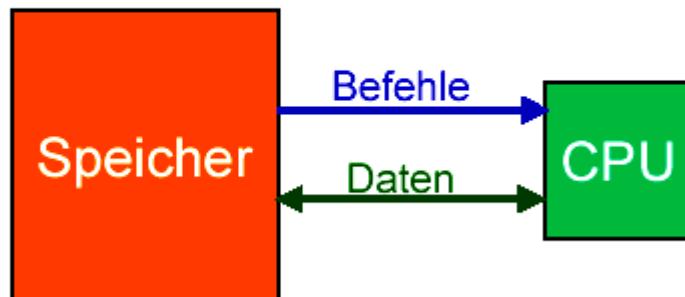
Strukturelle Maßnahmen

Klassifikation von Rechnerstrukturen nach Flynn:

Unterscheidung bezüglich der gleichzeitig bearbeiteten Befehls- und Datenströme:

- SISD (Single Instruction Single Data):

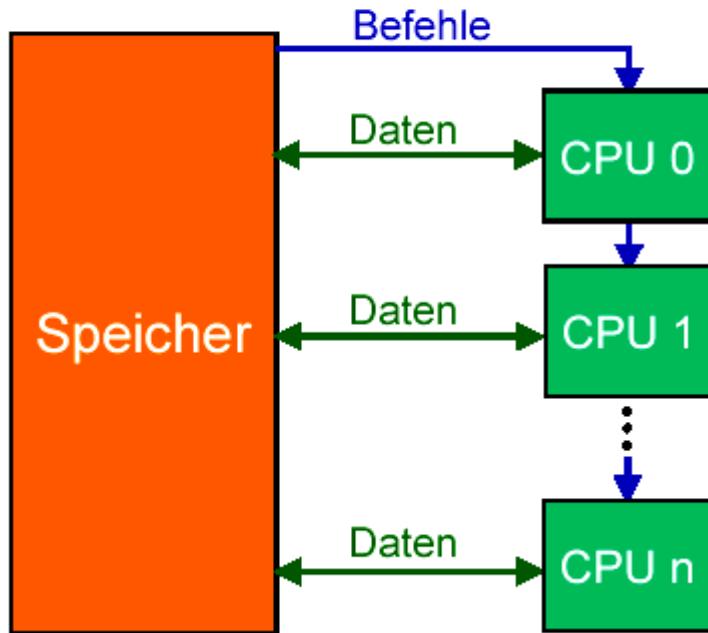
Ein Datenstrom wird entsprechend einer seriellen Befehlsfolge verarbeitet (von-Neumann-Rechner)



IBM-PC, IBM 370,
Micro-VAX von DEC

Strukturelle Maßnahmen

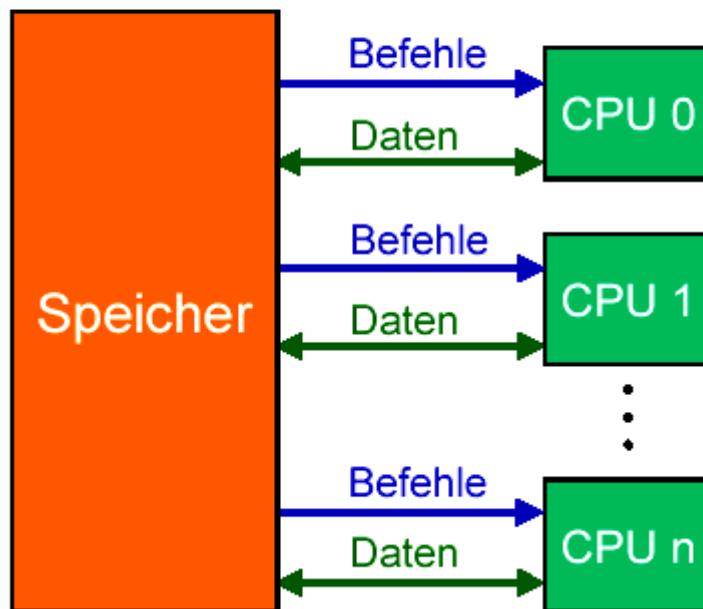
- SIMD (Single Instruction Multiple Data):
Alle Prozessoren führen gleichzeitig **dieselben Befehle** auf **verschiedenen Daten** aus (Array-Prozessoren).



Bildverarbeitung:
jedem Prozessor
wird ein Bildausschnitt
zugeordnet.

Strukturelle Maßnahmen

- ❑ MIMD (Multiple Instruction Multiple Data):
Alle Prozessoren führen gleichzeitig **verschiedene** Befehle auf **verschiedenen Daten** aus.



Multiprozessor-Systeme:
IBM 3084, Cray-2

Strukturelle Maßnahmen

□ MISD (Multiple Instruction Single Data):

Es wird nur ein Datenstrom bearbeitet. Bestimmte Ausführungseinheiten übernehmen die Ausführung bestimmter Teile einer Operation (Pipeline-Verarbeitung)

Parallelität auf Befehlsebene.

Diese Darstellung für MISD ist allerdings umstritten.
Mehrzahl der Experten ist der Meinung, dass es z.Z. keine MISD Maschinen gibt.

Techniken zur Parallelverarbeitung in Prozessoren

Fließbandverarbeitung (Pipelining)

- Vorausgesetzt wird ein sequenzieller Befehlsstrom (ISA). Es wird die Tatsache ausgenutzt, dass jede Befehlausführung aus mehreren Teilen besteht, die nacheinander in entsprechenden Stufen des Fließbandes abgearbeitet werden. Bei RISC-Pipeline-Architekturen werden diese Stufen mit verschiedenen Befehlen nun gleichzeitig überlappend ausgeführt.
- Zunächst ein Beispiel aus dem täglichen Leben.....

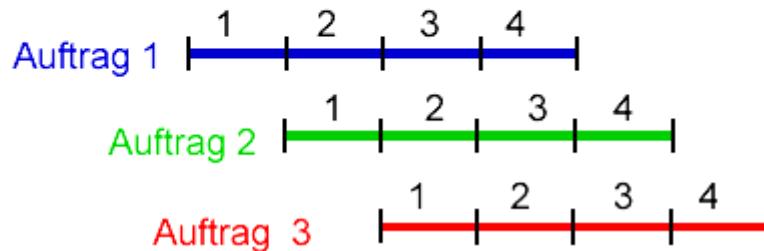
Fließband - Verarbeitung

Ausführung von 3 gleichartigen Verarbeitungsaufträgen in 4 Teilverarbeitungsschritten:

Serielle Verarbeitung:



Pipeline-Verarbeitung:

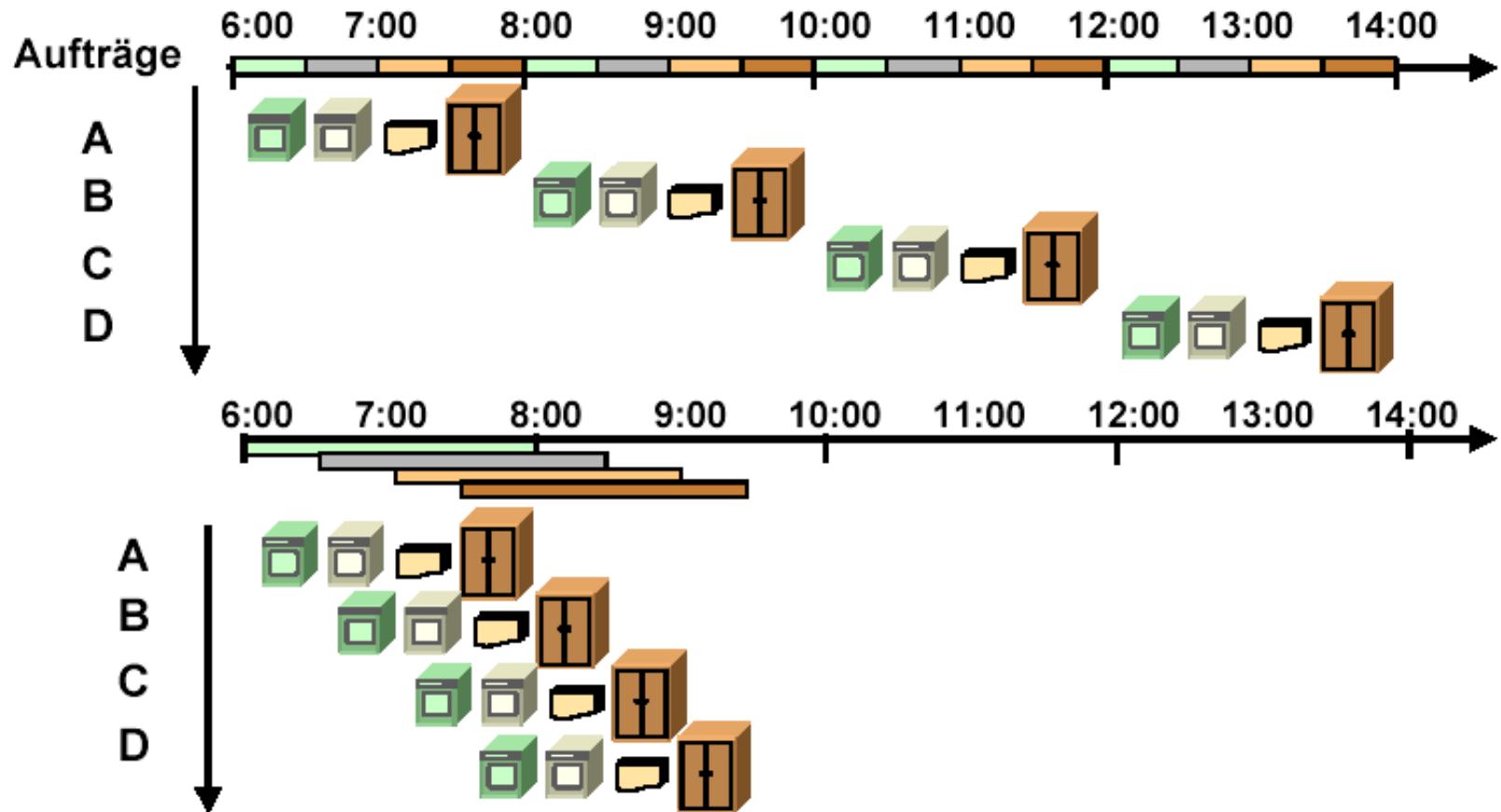


Beispiel: „Wäsche - Pipelining“

Ein Wäsche-Vorgang kann in 4 Teiltätigkeiten unterteilt werden:

- ⇒ Schmutzige Wäsche in die Waschmaschine
- ⇒ Nasse Wäsche in den Trockner
- ⇒ Falten, Bügeln, ...
- ⇒ Kleider in den Schrank

Beispiel: „Wäsche - Pipelining“



Definition: Pipeline - Verarbeitung

- Unter dem Begriff **Pipelining** versteht man die Zerlegung einer Maschinenoperation in mehrere Phasen oder Suboperationen, die dann von hintereinander geschalteten Verarbeitungseinheiten taktsynchron bearbeitet werden, wobei jede Verarbeitungseinheit genau eine spezielle Teiloeration ausführt
- Die Gesamtheit dieser Verarbeitungseinheiten nennt man eine **Pipeline**.
- Bei einer **Befehlspipeline** (Instruction Pipeline) wird die Ausführung eines Maschinenbefehls in verschiedene Phasen unterteilt, aufeinanderfolgende Maschinenbefehle werden jeweils um einen Taktzyklus versetzt ausgeführt

5-stufige Befehlspipeline (DLX)



IF: Befehl holen

ID: Befehl dekodieren

EX: Befehl ausführen

MA: Speicherzugriff

WB: Zurückschreiben

5-stufige Befehlspipeline (DLX)

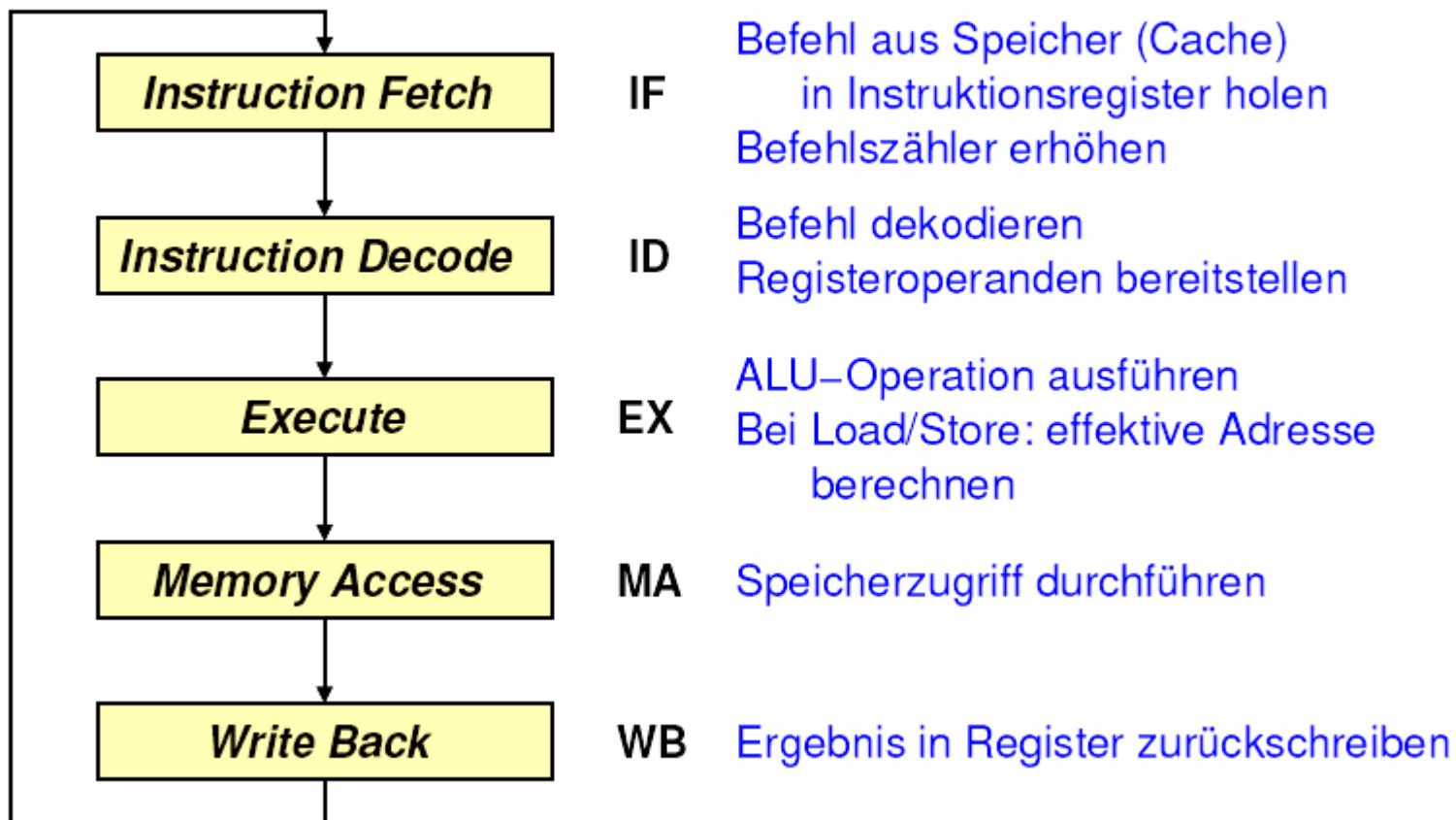
- Befehlsbereitstellungsphase (Instruction Fetch)
 - Der Befehl, der durch den Befehlszähler adressiert ist, wird aus dem Hauptspeicher oder dem Cache-Speicher in einen Befehlspuffer geladen.
 - Der Befehlszähler wird weitergeschaltet.
- Decodier- und Operandenbereitstellungsphase (Decode/Operand Fetch)
 - Aus dem Operationscode des Maschinenbefehls werden prozessorinterne Steuersignale erzeugt.
 - Die Operanden werden aus Registern bereitgestellt.
- Ausführungsphase (ALU Operation)
 - Die Operation wird auf den Operanden ausgeführt. Bei Lade-/Speicherbefehlen wird die effektive Adresse berechnet.

5-stufige Befehlspipeline (DLX)

- Speicherzugriffsphase (memory access)
 - Der Speicherzugriff wird durchgeführt.
- Resultatspeicherphase (Write Back)
 - Das Ergebnis wird in ein Register geschrieben.
- Anforderungen an die Befehlsstruktur
 - Einfache Maschinenbefehle
 - Einheitliches und festes Befehlsformat
 - Load / Store Architektur
 - Befehle arbeiten auf Registeroperanden
 - Load und Store Befehle greifen auf Speicher zu

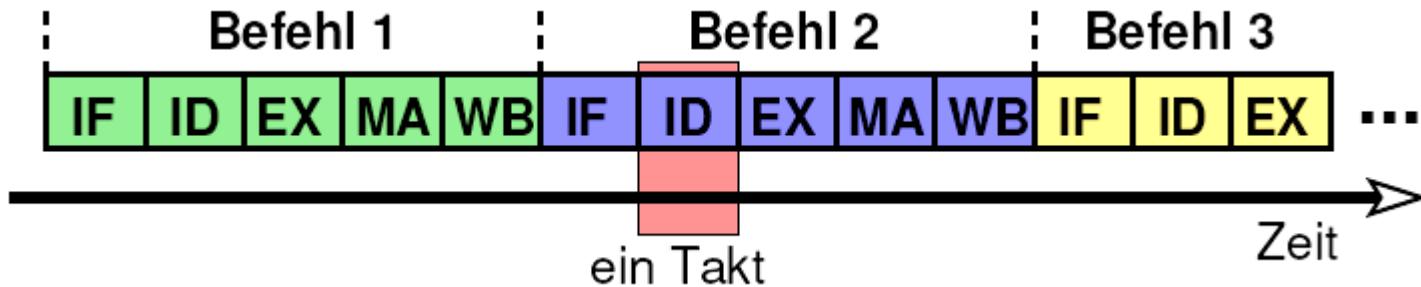
Befehlszyklus für die Pipeline

Erinnerung: ein einfacher Befehlszyklus (leicht modifiziert)



Zeitlicher Ablauf des Pipelining

- Zeitlicher Ablauf ist sequentiell
- Ein einfacher Befehlsstrom (vorerst ohne Sprünge)



- Erkenntnis: Die Schritte des Befehlszyklus werden durch verschiedene Einheiten ausgeführt
 - jede Einheit wird nur in einem von fünf Takten benutzt

Zeitlicher Ablauf des Pipelining

Befehlsausführung mit Pipelining



Zeitlicher Ablauf des Pipelining

Befehlsausführung mit Pipelining



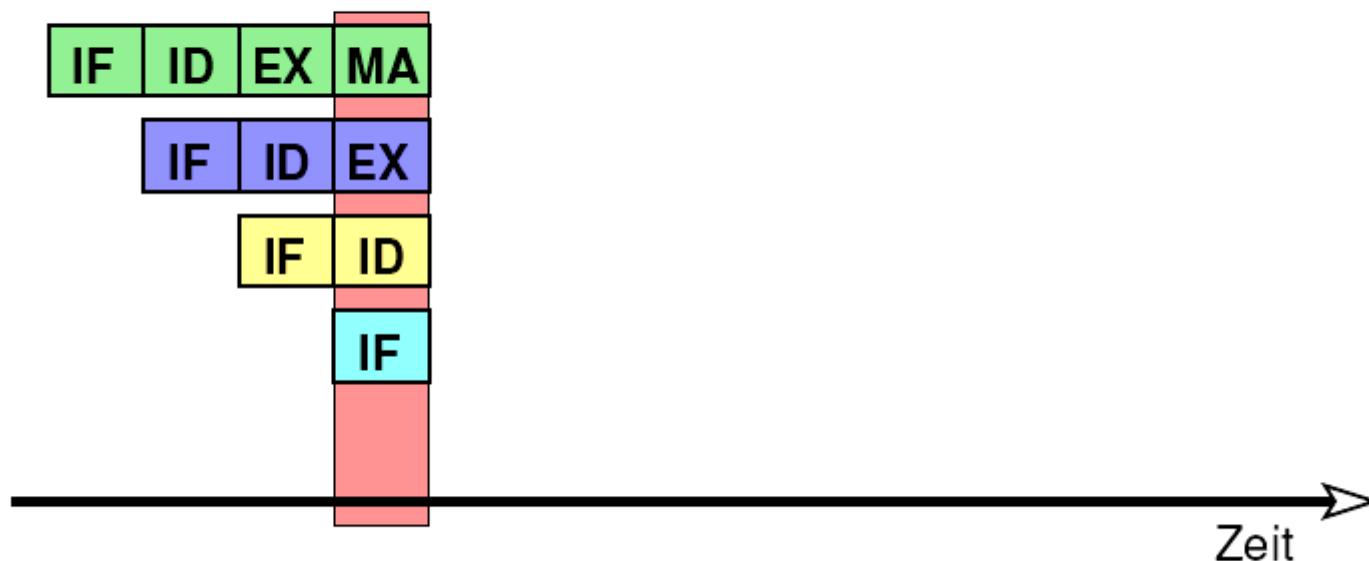
Zeitlicher Ablauf des Pipelining

Befehlsausführung mit Pipelining



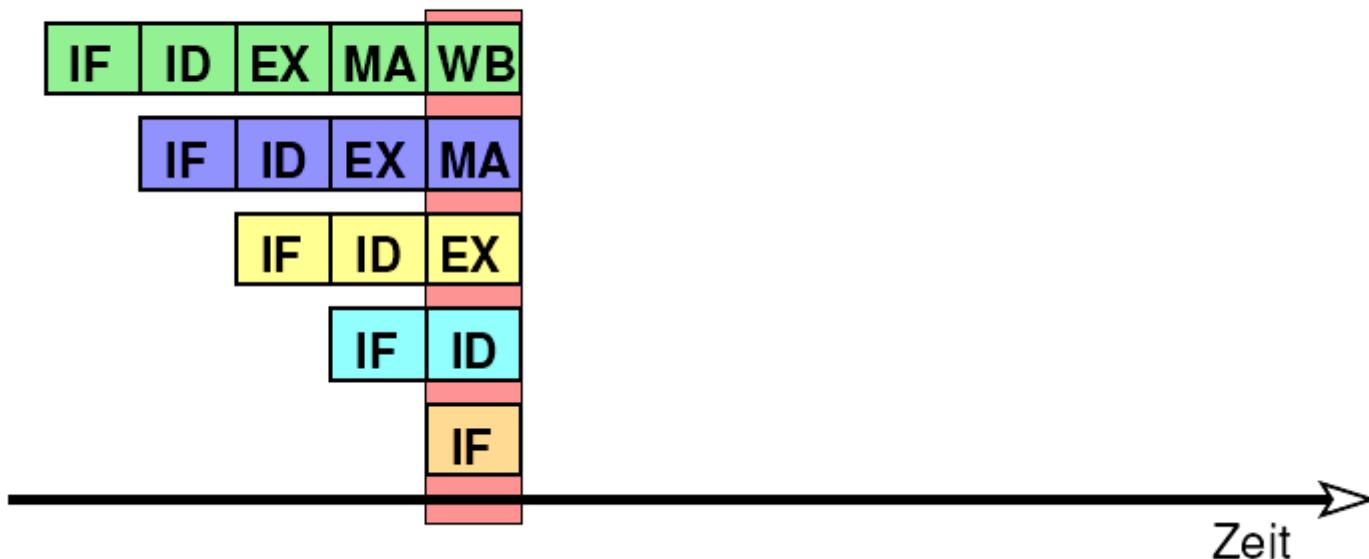
Zeitlicher Ablauf des Pipelining

Befehlsausführung mit Pipelining



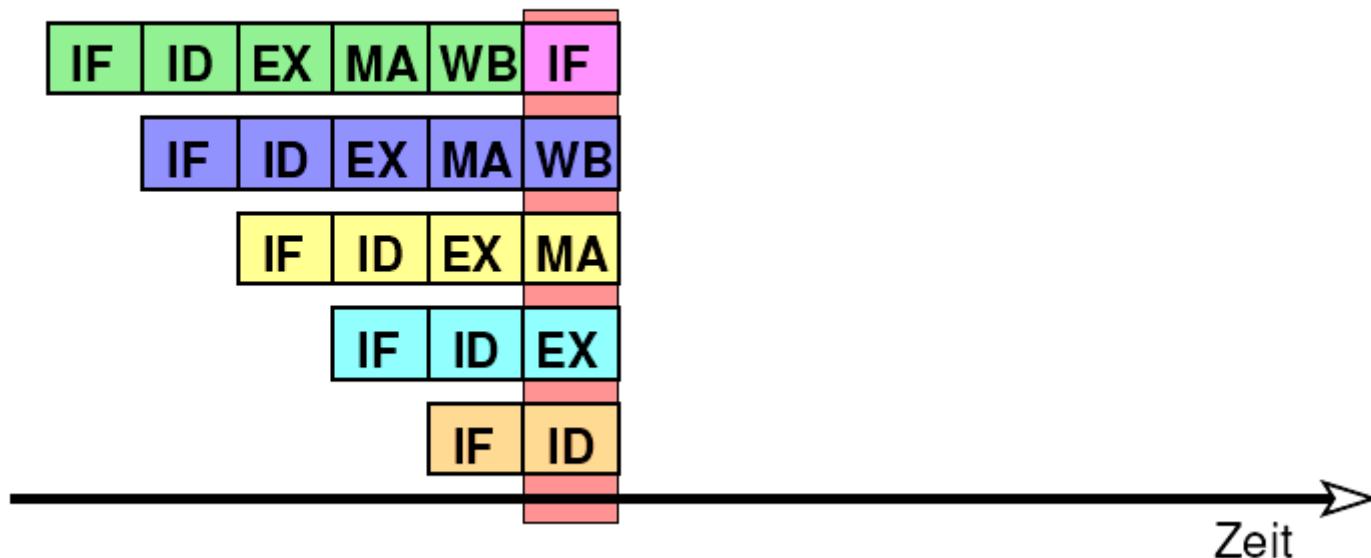
Zeitlicher Ablauf des Pipelining

Befehlsausführung mit Pipelining



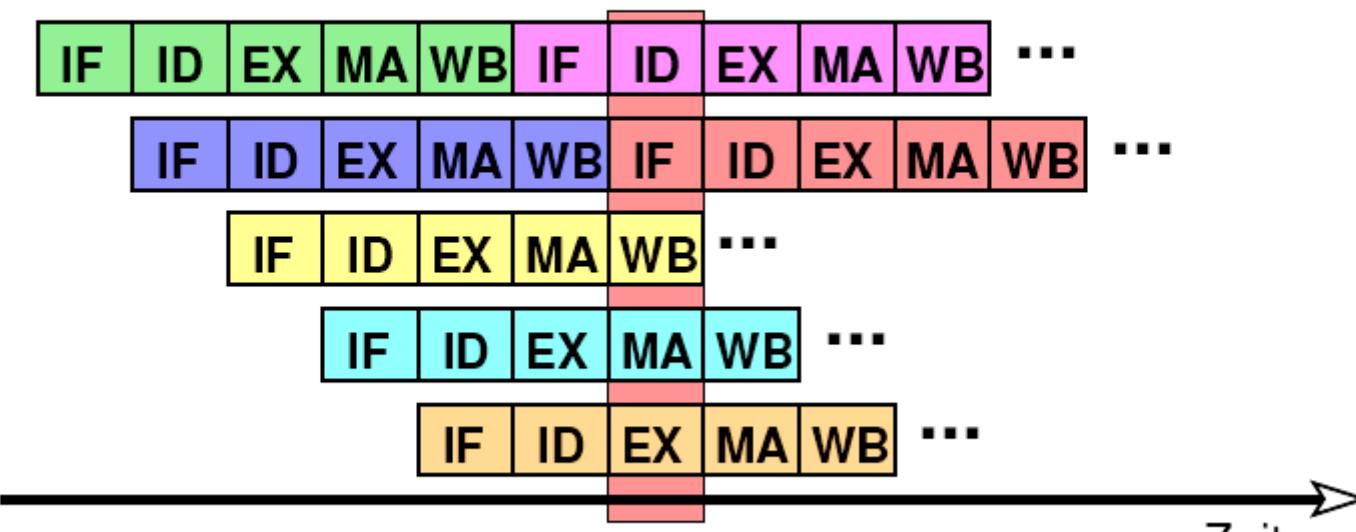
Zeitlicher Ablauf des Pipelining

Befehlsausführung mit Pipelining



Zeitlicher Ablauf des Pipelining

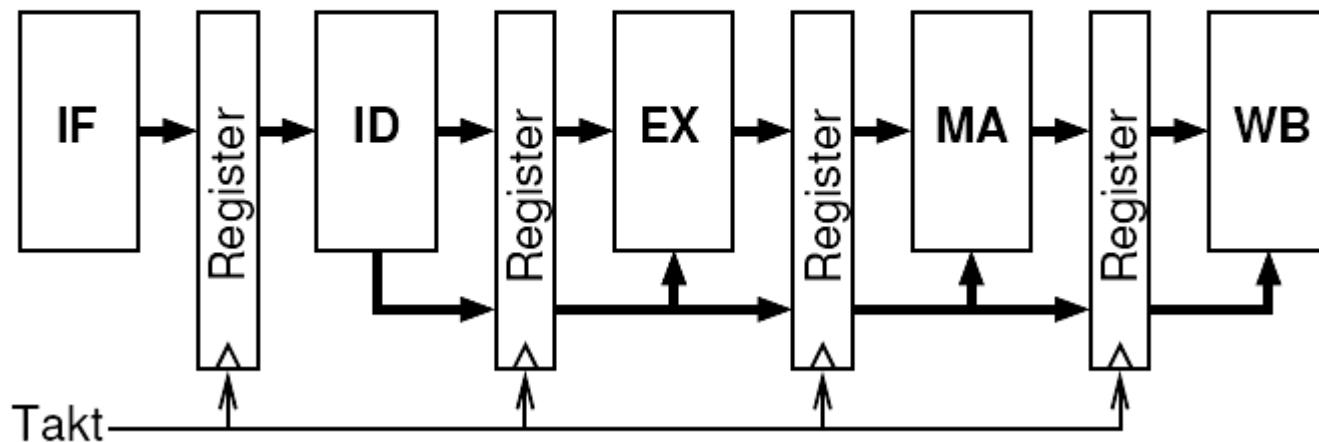
Befehlsausführung mit Pipelining



- Analogie: Fließband in der KFZ-Industrie
 - Ausführung eines Befehls braucht immer noch 5 Takte
 - trotzdem: im Mittel wird pro Takt ein Befehl fertig ($CPI = 1$)

Hardwarestruktur und Belegung der Stufen

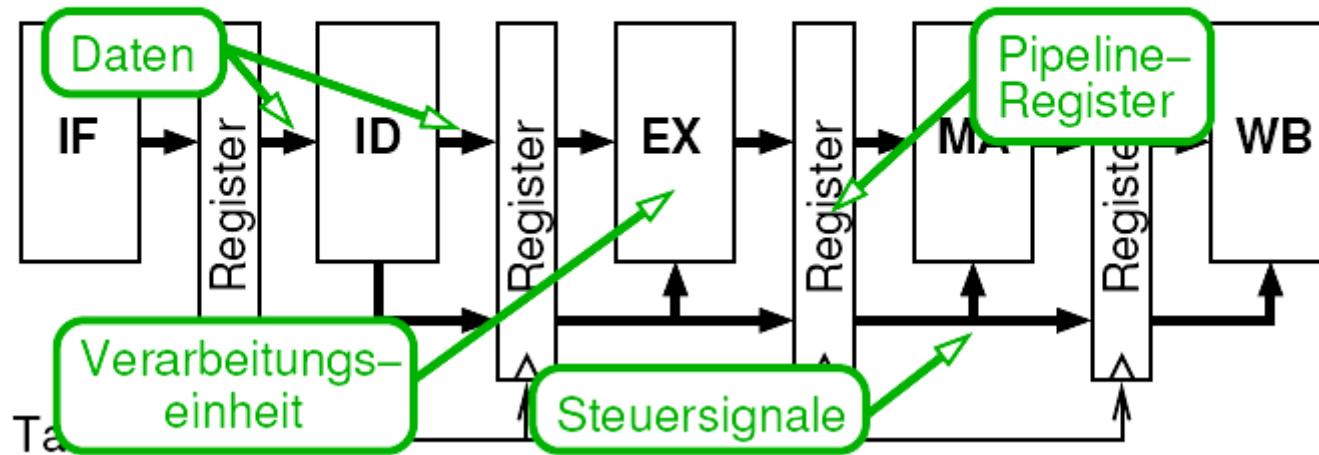
Realisierung des Pipelining in Hardware



- ➔ Pipelinestufen sind durch Pipeline-Register verbunden
 - ➡ Pipeline-Register speichern Daten und Steuersignale
- ➔ In jedem Takt: „weiterschieben“ zur nächsten Stufe

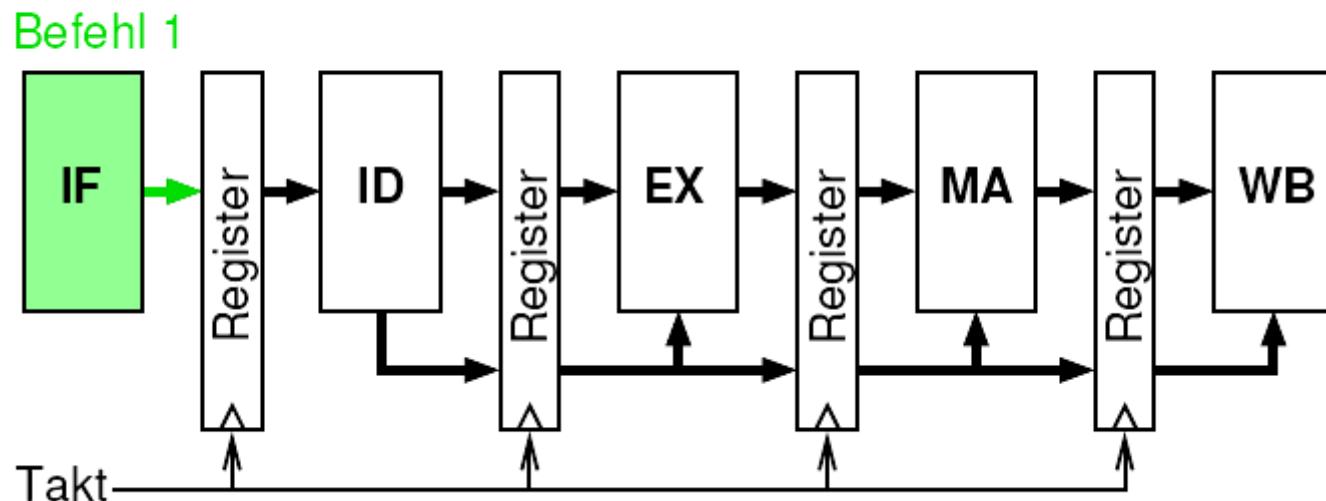
Hardwarestruktur und Belegung der Stufen

Realisierung des Pipelining in Hardware



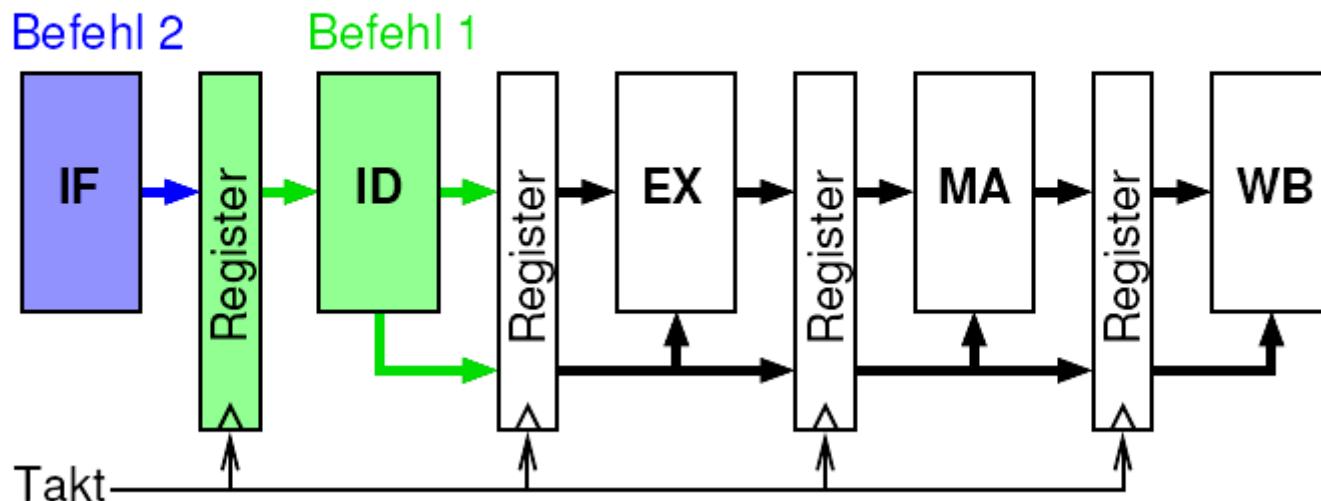
Hardwarestruktur und Belegung der Stufen

Realisierung des Pipelining in Hardware



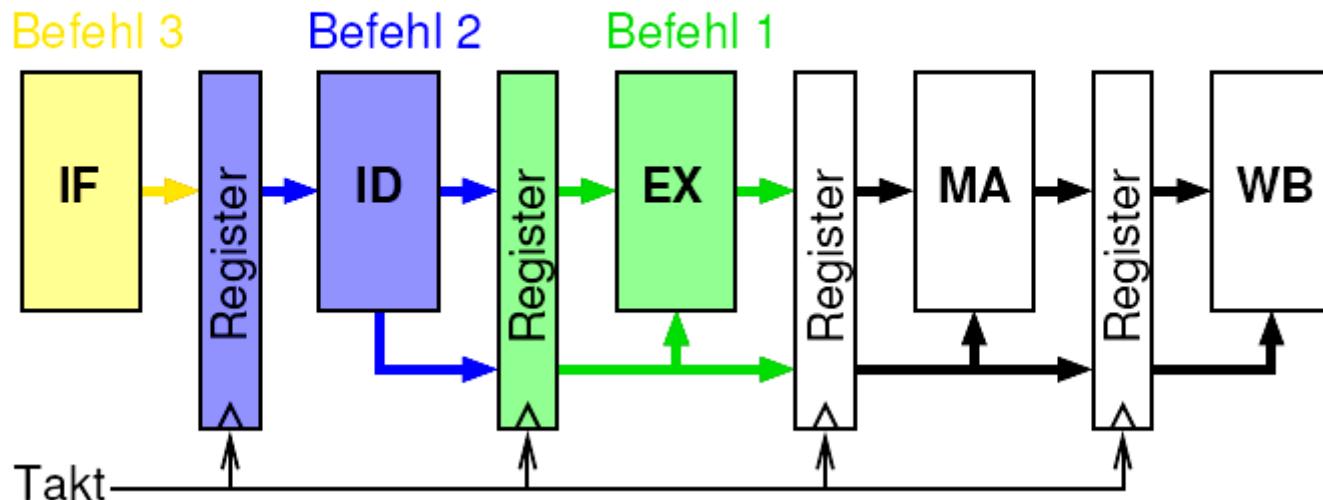
Hardwarestruktur und Belegung der Stufen

Realisierung des Pipelining in Hardware



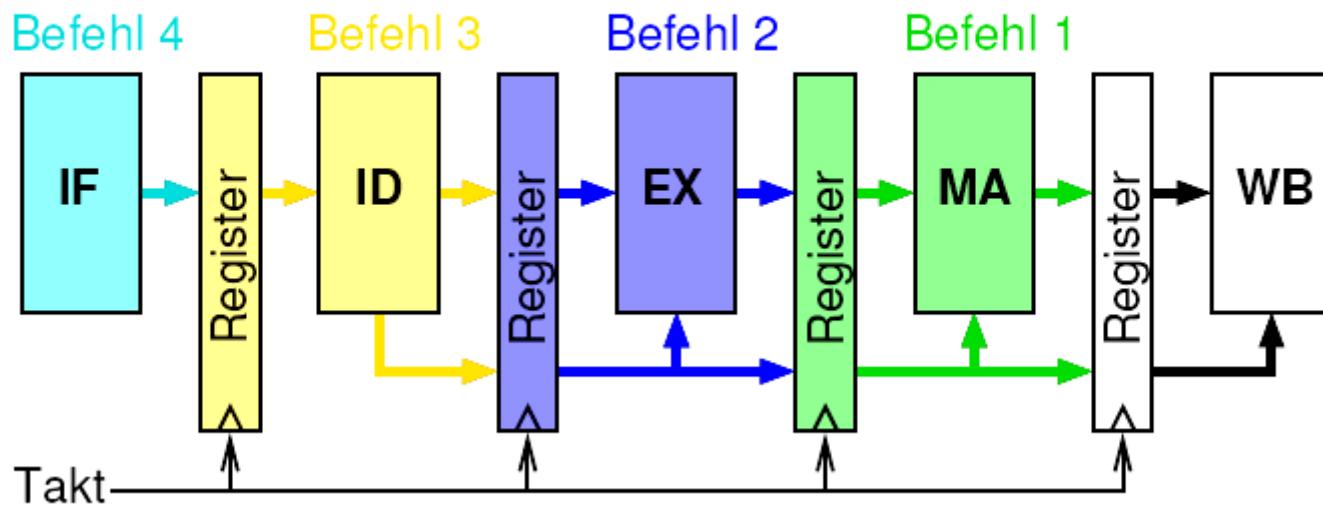
Hardwarestruktur und Belegung der Stufen

Realisierung des Pipelining in Hardware



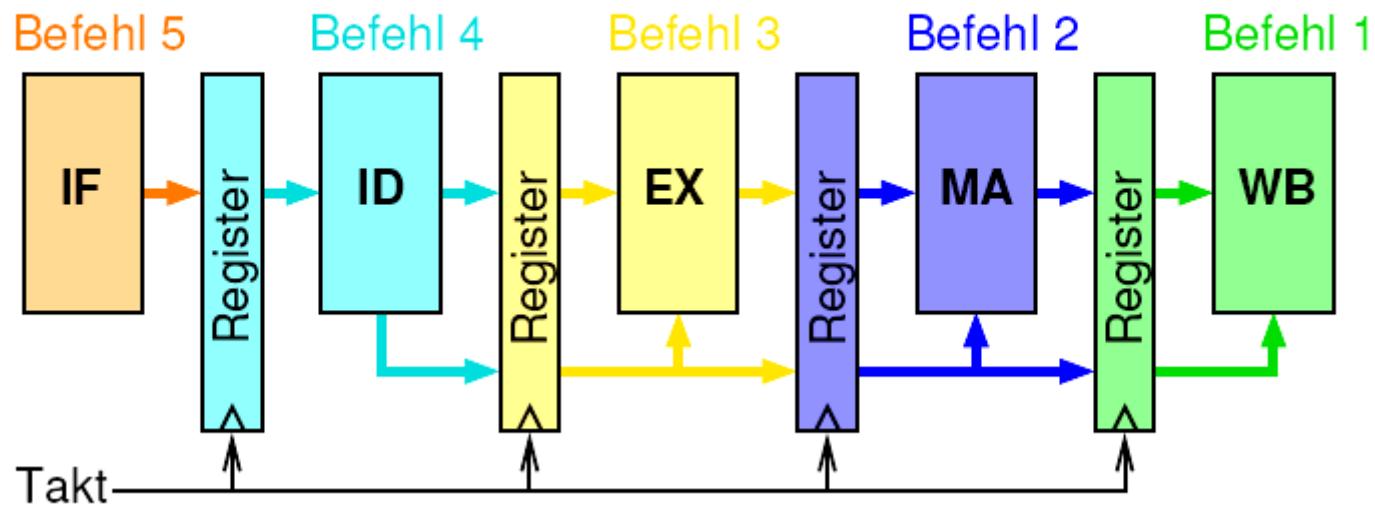
Hardwarestruktur und Belegung der Stufen

Realisierung des Pipelining in Hardware



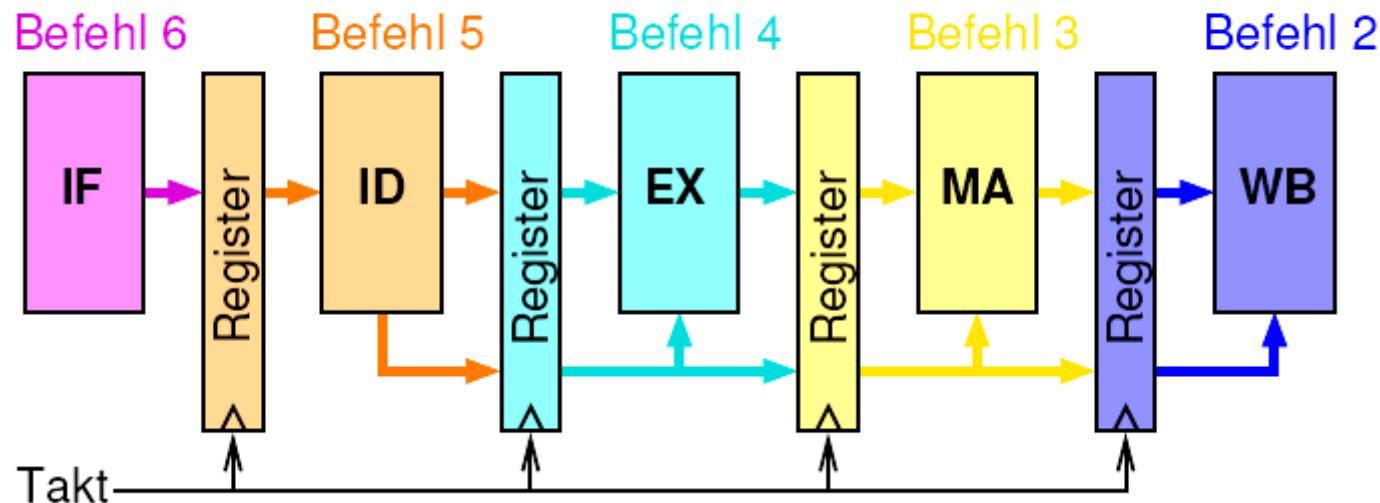
Hardwarestruktur und Belegung der Stufen

Realisierung des Pipelining in Hardware



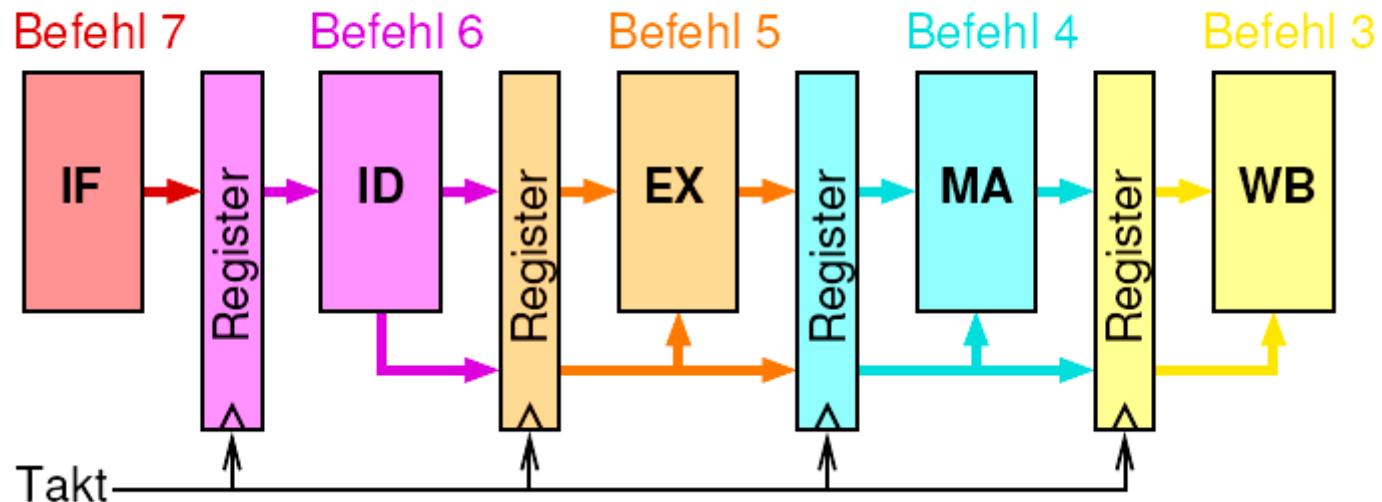
Hardwarestruktur und Belegung der Stufen

Realisierung des Pipelining in Hardware



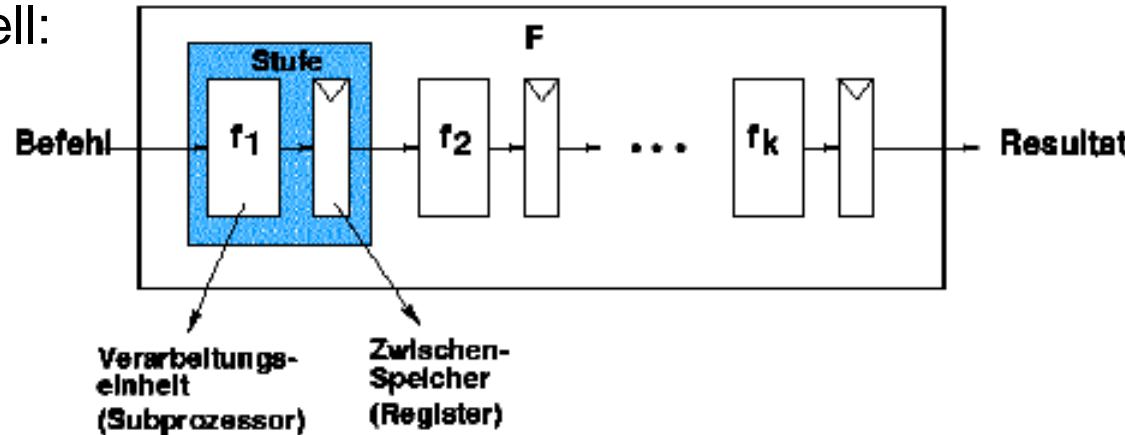
Hardwarestruktur und Belegung der Stufen

Realisierung des Pipelining in Hardware



Realisierung: Lineare Pipeline mit k Stufen

Modell:



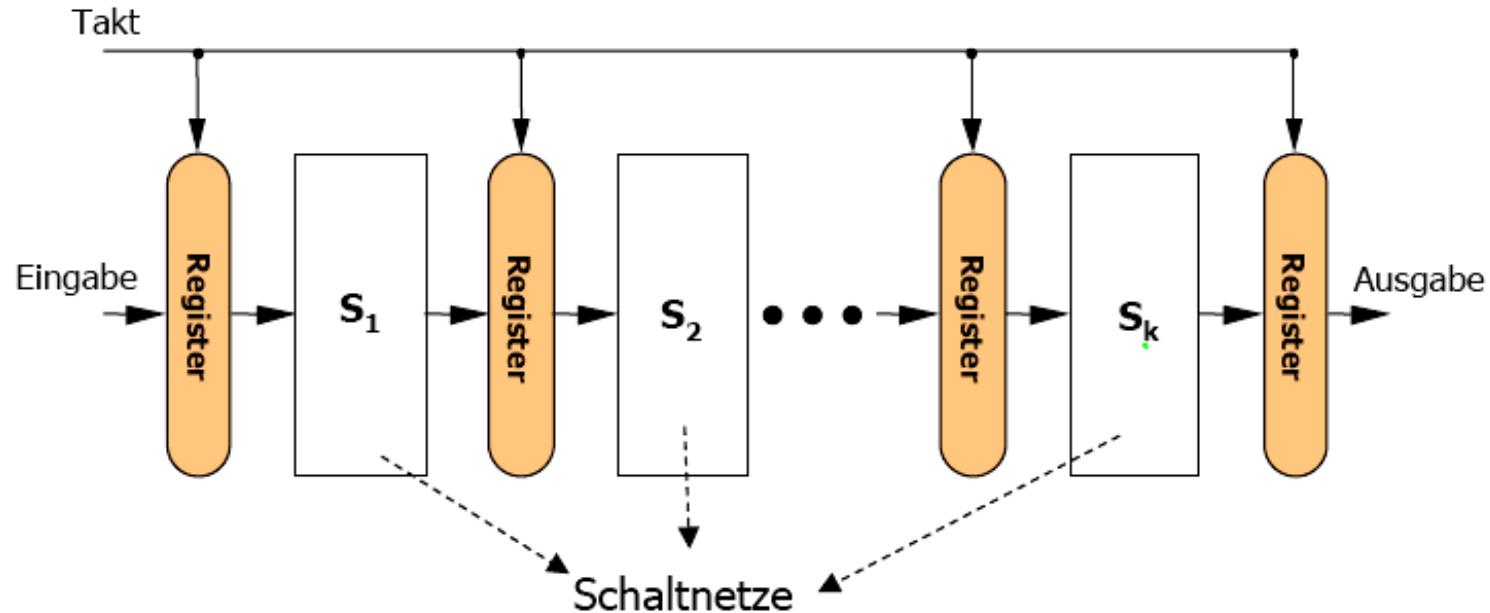
Annahmen für die Pipeline-Verarbeitung:

- Die Operation F kann in Suboperationen f_i zerlegt werden
- Die Bearbeitung der einzelnen Suboperationen f_i benötigt etwa die gleiche Zeit
- Die Ausführungszeit einer Suboperation ist lang gegenüber der Register-Zugriffszeit

Pipelining - Definitionen

- ❑ Jede Stufe der Pipeline heisst **Pipeline-Stufe**
- ❑ Pipeline-Stufen werden durch getaktete **Pipeline-Register** (auch *latches* genannt) getrennt.
- ❑ Ein Pipeline-Maschinentakt ist die Zeit, die benötigt wird, um einen Befehl eine Stufe weiter durch die Pipeline zu schieben.
- ❑ Idealerweise wird ein Befehl in einer **k-stufigen Pipeline** in k Takten von k Stufen ausgeführt.
- ❑ Wird in jedem Takt ein neuer Befehl geladen, dann werden zu jedem Zeitpunkt unter idealen Bedingungen k Befehle gleichzeitig behandelt und jeder Befehl benötigt k Takte, bis zum Verlassen der Pipeline.

Lineare Pipeline mit k Stufen



Verzögerungszeiten:

- der Schaltnetze: τ_i ($i = 1, \dots, k$)
- der Pipeline-Register: τ_{reg}

Länge eines Taktzyklus:

$$\tau = \max\{\tau_1, \tau_2, \dots, \tau_k\} + \tau_{reg}$$

Pipelining - Definitionen

- Man definiert die **Latenz** als die Zeit, die ein Befehl benötigt, um alle k Pipeline-Stufen zu durchlaufen.

Ideale Verhältnisse:

- Ausführung eines Befehls in k Takten.
- Es werden gleichzeitig k Befehle bearbeitet.

- Der **Durchsatz** einer Pipeline wird definiert als die Anzahl der Befehle, die eine Pipeline pro Takt verlassen können. Dieser Wert spiegelt die Rechenleistung einer Pipeline wider.

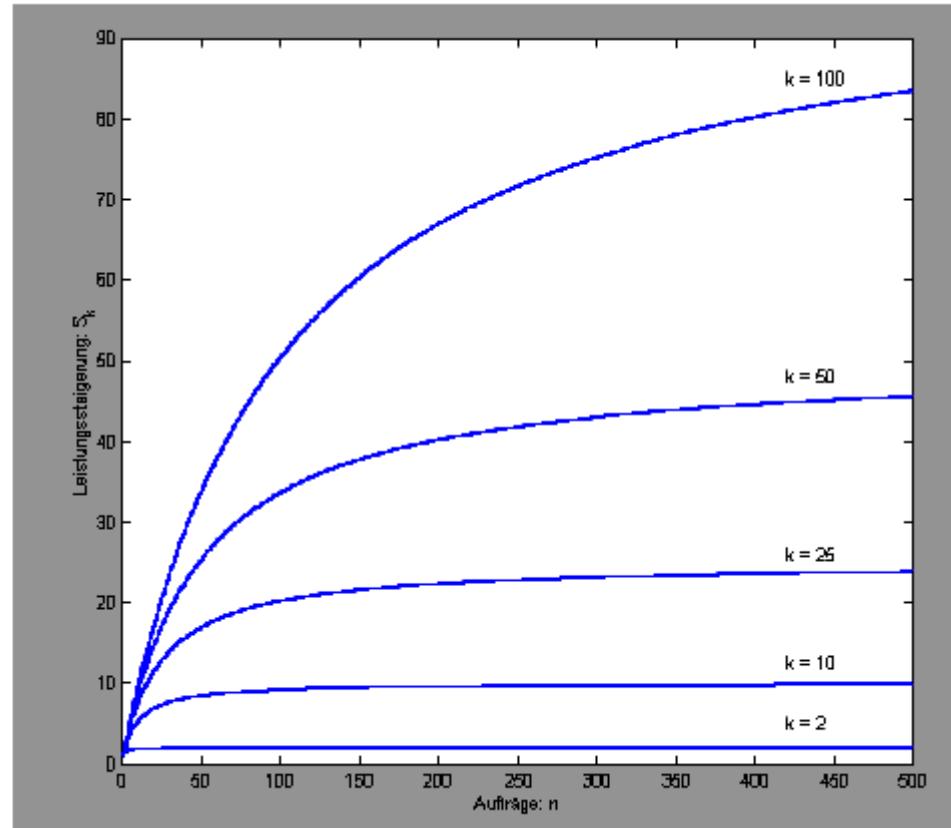
Leistungssteigerung durch Pipelines

- Im **Idealfall** werden n Befehle in einer Pipeline mit k Stufen in $T_k = k + (n-1)$ Taktzyklen ausgeführt.
 - k Taktzyklen, um die Pipeline zu füllen
 - $(n-1)$ Taktzyklen, um die restlichen $(n-1)$ Befehle durchzuführen.
- In einem hypothetischen Prozessor ohne Pipeline würde man zur Durchführung von n Befehlen $n * k$ Taktzyklen benötigen (unter der Annahme idealer Bedingungen mit einer Latenz von k Takten und einem Durchsatz von 1)
- Leistungssteigerung S_k (*speedup*) eines Pipeline-Prozessors mit k Stufen gegenüber einem „äquivalenten“ Prozessor ohne Pipeline ist definiert als die dimensionslose Zahl S_k :

Leistungssteigerung durch Pipelines

$$S_k = \frac{n * k}{k + (n-1)}$$

$$\lim_{n \rightarrow \infty} S_k = k$$



Leistungssteigerung durch Pipelines

- Eine Erhöhung der Pipelinestufenzahl bewirkt eine Beschleunigung der Verarbeitungsgeschwindigkeit (idealer Fall, die Beschleunigung nähert sich asymptotisch k).
- Die **Erhöhung der Pipelinestufenzahl** bewirkt, dass jede einzelne Stufe weniger komplex wird (kleinere Gattertiefe) und damit im Prinzip höher getaktet werden kann. Also kann eine längere Pipeline eine höhere Verarbeitungsgeschwindigkeit besitzen als eine kürzere Pipeline (höhere Taktrate).
- Problem: Die Verarbeitung/Lösung von Pipelinekonflikten wird umso komplizierter und zeitlich aufwendiger, je länger die Pipeline ist.

Pipeline - Hemmnisse

Die Verarbeitung eines Befehlstroms in einer Pipeline kann durch **Pipelinekonflikte** gebremst werden → Unterbrechung des taktsynchronen Durchlauf eines Befehls durch die Pipelinestufen.

Pipelinekonflikte werden hervorgerufen durch

- ❑ Daten- und Kontrollabhängigkeiten im Programm
- ❑ Nichtverfügbarkeit von Ressourcen
(Ausführungseinheiten, Register, ... usw.)

Abhängigkeiten können, falls sie nicht erkannt und behandelt werden, zu fehlerhaften Datenzuweisungen führen. Die Situationen, die zu Konflikten führen, werden **Pipeline-Hemmnisse** (Pipeline Hazards) genannt.

Pipeline - Hemmnisse

□ Datenkonflikte

- ♦ Treten auf, wenn ein Operand nicht verfügbar ist;
- ♦ Beispiel: Befehl benötigt das Ergebnis eines vorhergehenden und noch nicht abgeschlossenen Befehls;
- ♦ Datenabhängigkeiten zwischen Befehlen im Befehlsstrom

□ Strukturkonflikte

- ♦ Treten bei einigen Befehlskombinationen aufgrund von Ressourcenkonflikten auf;
- ♦ Beispiel: Gleichzeitiger Schreibzugriff zweier Befehle auf Registerdatei mit nur einem Schreibeingang.

□ Steuerkonflikte

- ♦ Bei Sprung- und Steuerflussbefehlen.
- ♦ Beispiel: Bei einem bedingten Sprung muss der Befehlsstrom in der Pipeline unterbrochen und das Sprungziel geholt werden.

Bestimmung von Datenabhängigkeiten

Man betrachte zwei Befehle S_i ($i=1,2$), wobei

- ⇒ DEF_i sei die Menge der Variablen, auf die in S_i schreibend zugegriffen wird
- ⇒ USE_i die Menge der Variablen, auf die in S_i lesend zugegriffen wird

Eine hinreichende Bedingung für eine parallele Ausführung von S_1 und S_2 ist dann:

$$(\text{DEF}_1 \cap \text{USE}_2) \cup (\text{USE}_1 \cap \text{DEF}_2) \cup (\text{DEF}_1 \cap \text{DEF}_2) = \emptyset$$

Diese Bedingung drückt eine völlige Datenunabhängigkeit zwischen den beiden Anweisungen S_1 und S_2 aus

Klassifizierung von Datenabhängigkeiten

- Es besteht eine **echte Abhängigkeit** (*true Dependence*) von S_1 nach S_2 ($S_1 \xrightarrow{\delta'} S_2$), falls eine Variable $v \in \text{DEF}_1 \cap \text{USE}_2$ existiert
(S_2 liest eine Variable v , die in S_1 geschrieben wurde)

S_1 schreibt in ein Register, das von S_2 gelesen wird
(S_2 würde einen veralteten Registerwert lesen)

- Es besteht eine **Gegenabhängigkeit** (*Anti Dependence*) von S_1 nach S_2 ($S_1 \xrightarrow{\delta^a} S_2$), falls eine Variable $v \in \text{USE}_1 \cap \text{DEF}_2$ existiert
(S_1 liest eine Variable v , die in S_2 geschrieben wurde)
(S_1 sollte eigentlich den alten Registerwert verwenden, aber S_2 hat schon den neuen Wert ins Register geschrieben).

Klassifizierung von Datenabhängigkeiten

- Es besteht eine **Ausgabeabhängigkeit** (*Output Dependence*) von S_1 nach S_2 ($S_1 \xrightarrow{\delta^o} S_2$), falls eine Variable $v \in \text{DEF}_1 \cap \text{DEF}_2$ existiert

(S_1 und S_2 greifen beide schreibend auf die Variable v zu)

(Da S_1 und S_2 parallel arbeiten, hängt der Wert im Ausgangsregister davon ab, wer zuerst fertig wird)

Die obigen Ausführungen gelten allgemein. Für „benachbarte Befehle“ können diese Konflikte aufgrund der Abhängigkeiten entstehen. Manche dieser Fälle können jedoch in unserer einfachen linearen Pipeline nicht auftreten (aber z.B. bei superskalaren Prozessoren (siehe später)).

Klassifizierung von Datenabhängigkeiten

Beispiel:

S₁: add r1,r2,2 # r1 := r2 + 2

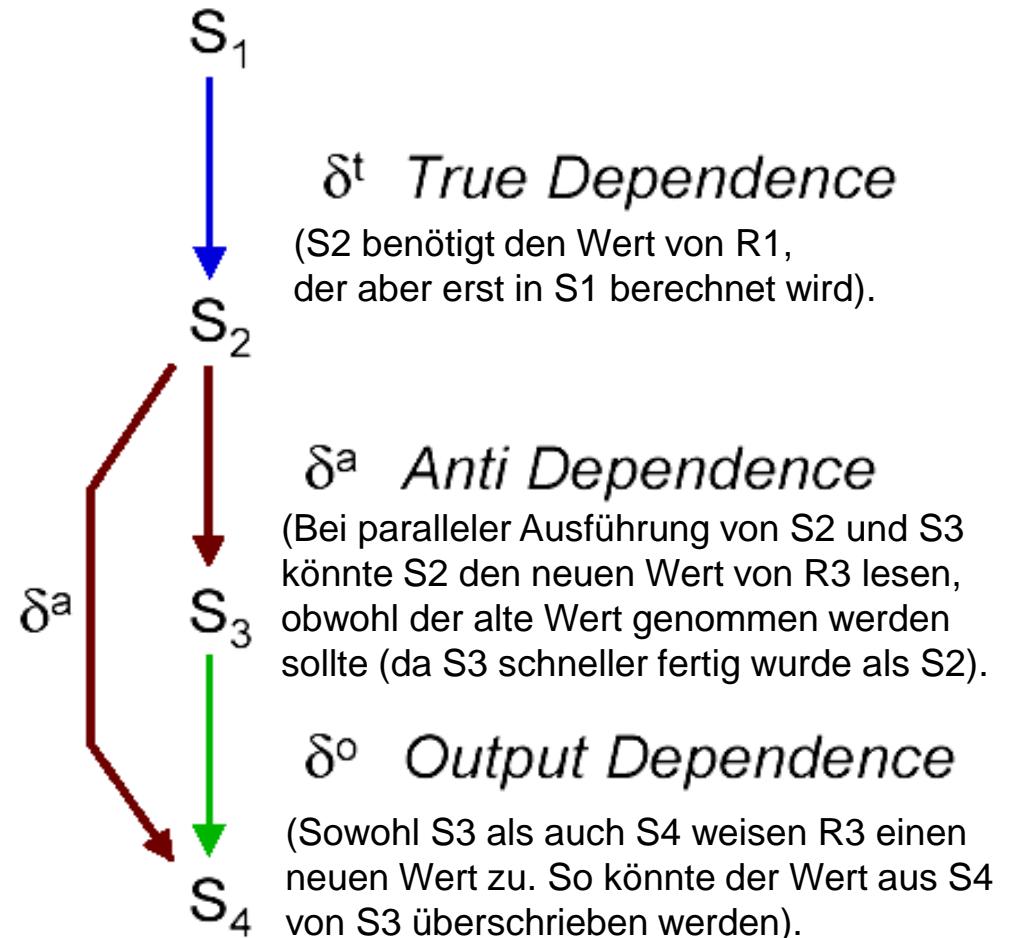
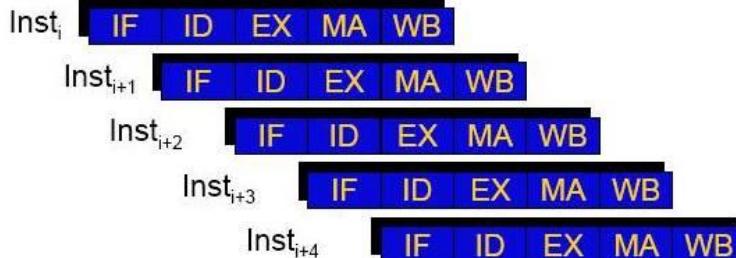
S₂: add r4,r1,r3 # r4 := r1 + r3

S₃: mul r3,r5,3 # r3 := r5 * 3

S₄: mul r3,r6,3 # r3 := r6 * 3

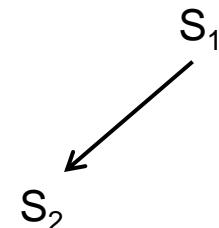
Klassifizierung von Datenabhängigkeiten

$S_1:$	add	r1, r2, 2
$S_2:$	add	r4, r1, r3
$S_3:$	mul	r3, r5, 3
$S_4:$	mul	r3, r6, 3



Datenabhängigkeiten - Definitionen

- An instruction j is data dependend on instruction i if either of the following holds:
 - Instruction i produces a result that may be used by instruction j , or
 - Instruction j is data dependant on instruction k , and instruction k is data dependant on instruction i .
- Representation by arrows:
 - The arrow points from an instruction that must precede the instruction that the arrowhead points to.



Datenabhängigkeiten und Datenkonflikte

- Datenabhängigkeiten können Datenkonflikte verursachen, wenn die entsprechenden Befehle so nah beieinander sind, dass ihre überlappende Bearbeitung in der Pipeline ihre Zugriffsreihenfolge verändert.
- Es bestehen also keine Datenkonflikte, wenn die datenabhängigen Befehle „weit genug“ voneinander entfernt sind. Dies hängt aber von der Struktur der Pipeline ab.

Die beschriebenen Datenabhängigkeiten können zu folgenden Datenkonflikten führen:

- **Read-After-Write-Konflikt (RAW)**
 - wird durch True Dependence erzeugt
 - Tritt auf, wenn Befehl j sein Quellregister liest, bevor Befehl i das Ergebnis geschrieben hat ($j > i$).

Datenabhängigkeiten und Datenkonflikte

Die beschriebenen Datenabhängigkeiten können zu folgenden Datenkonflikten führen:

- Write-After-Read-Konflikt (WAR)
 - wird durch Anti Dependence erzeugt
 - Tritt auf, wenn Befehl j sein Zielregister beschreibt, bevor Befehl i den Operanden gelesen hat ($j > i$).
 - kann nur in einer Pipeline auftreten, wenn sich die Befehle vor der Operandenbereitstellung überholen können (vergl. Beispiel S2/S3), dies ist bei superskalaren Prozessoren möglich.
- Write-After-Write-Konflikt (WAW)
 - wird durch Output Dependence erzeugt
 - tritt nur in Pipelines auf, die in mehr als einer Stufe in ein Register schreiben können
- WAR und WAW-Konflikte können in unserer Pipeline nicht auftreten.

Pipeline-Konflikte

Datenabhängigkeiten

→ Echte Abhängigkeit (*true dependence, flow dependence*)

Inst1: $a = b + c$

...
 δt

Inst 2: $x = a * b$

Inst 1 schreibt sein Ergebnis in ein Register (oder Speicher), das von Inst 2 gelesen wird.

→ Antiabhängigkeit (*anti dependence*)

Inst1: $a = b + c$

...
 δa

Inst 2: $b = x * y$

Inst 1 liest einen Wert aus einem Register (oder Speicher), das von Inst 2 überschrieben wird.

Pipeline-Konflikte

Datenabhängigkeiten ...

→ Ausgabeabhängigkeit (*output dependence*)

Inst1: $a = b + c$
↓
 $\delta^0 \dots$

Inst 2: $a = x * y$

Inst 1 und Inst 2 beschreiben
dasselbe Register (oder dieselbe
Speicherzelle)

Datenkonflikt

- Ein **Datenkonflikt** tritt auf, wenn die überlappte Abarbeitung zweier Befehle in der Pipeline eine Datenabhängigkeit verletzt, d.h. die Reihenfolge der beteiligten Zugriffe vertauschen würde.

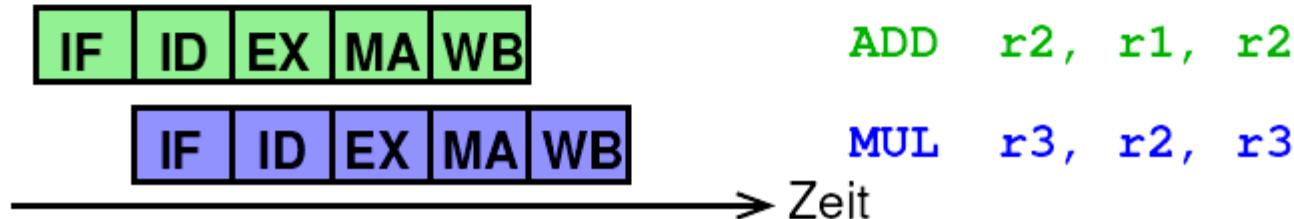
Pipeline-Konflikte

Datenkonflikte

- Konflikte, die durch den Datenfluß zwischen Befehlen ausgelöst werden
- Beispiel: Ein Befehl benötigt das Ergebnis des vorigen:

ADD r2, r1, r2 ; $r2 = r1 + r2$ Datenabhängigkeit
MUL r3, r2, r3 ; $r3 = r2 * r3$

- Problem bei Pipelining:



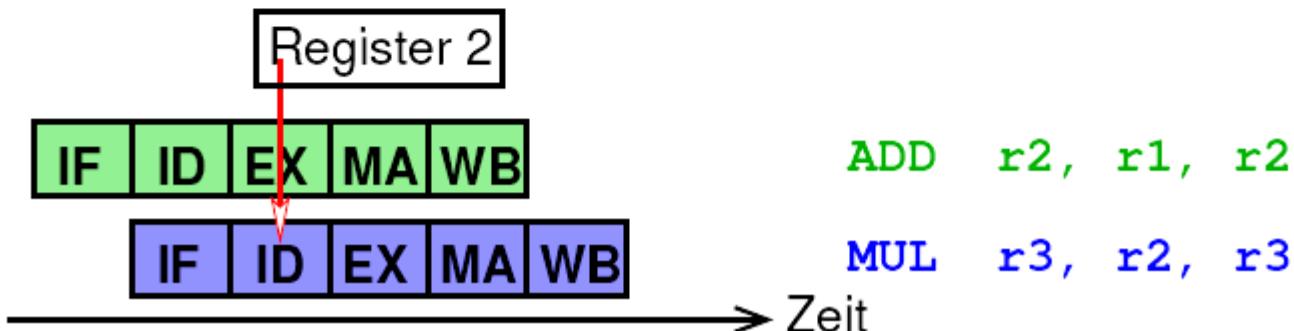
Pipeline-Konflikte

Datenkonflikte

- Konflikte, die durch den Datenfluß zwischen Befehlen ausgelöst werden
- Beispiel: Ein Befehl benötigt das Ergebnis des vorigen:

ADD r2, r1, r2 ; r2 = r1 + r2 Datenabhängigkeit
MUL r3, r2, r3 ; r3 = r2 * r3

- Problem bei Pipelining:



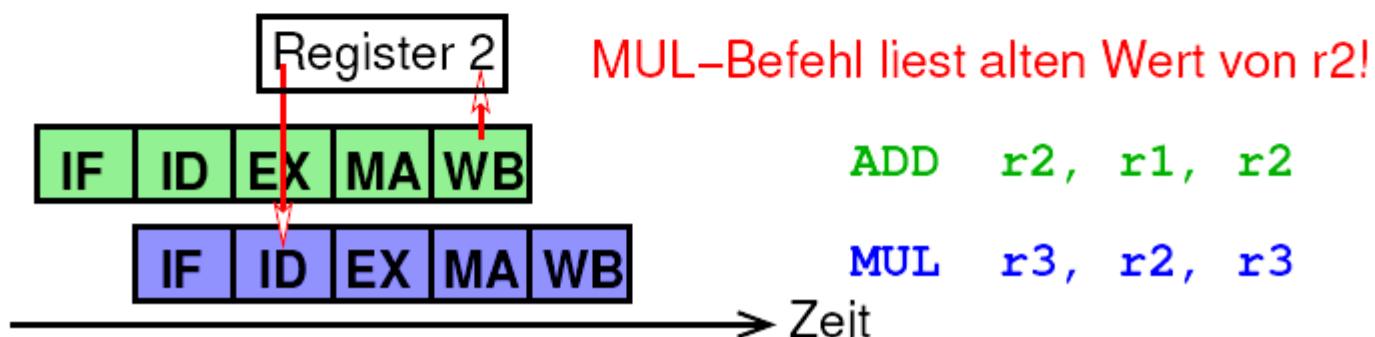
Pipeline-Konflikte

Datenkonflikte

- Konflikte, die durch den Datenfluß zwischen Befehlen ausgelöst werden
- Beispiel: Ein Befehl benötigt das Ergebnis des vorigen:

ADD r2, r1, r2 ; $r2 = r1 + r2$ Datenabhängigkeit
MUL r3, r2, r3 ; $r3 = r2 * r3$

- Problem bei Pipelining:



Lösung der Datenkonflikte

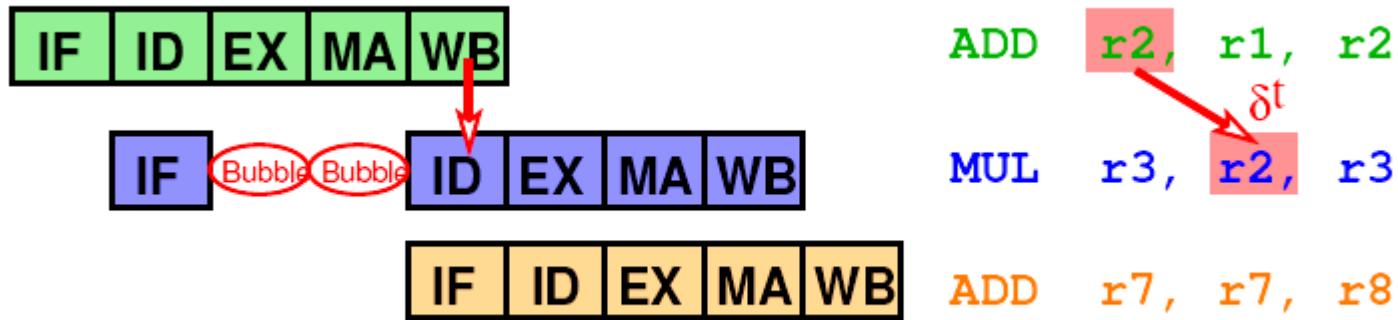
Auflösung von Datenkonflikten

- Hardware-Lösungen ([dynamische Verfahren](#))
 - benötigen entsprechende Logik zur Konflikterkennung
 - z.B. Scoreboarding (wird in diesem Kurs nicht behandelt)
- Techniken:
 - Leerlauf der Pipeline (Interlocking, Stalling)
 - Forwarding
 - Forwarding mit Interlocking

Lösung der Datenkonflikte

Auflösung von Datenkonflikten: Interlocking

- Die Bearbeitung eines Befehls (und aller nachfolgenden!) wird in der Pipeline für einige Takte angehalten
 - Einfügen von Pipeline-Leerzyklen



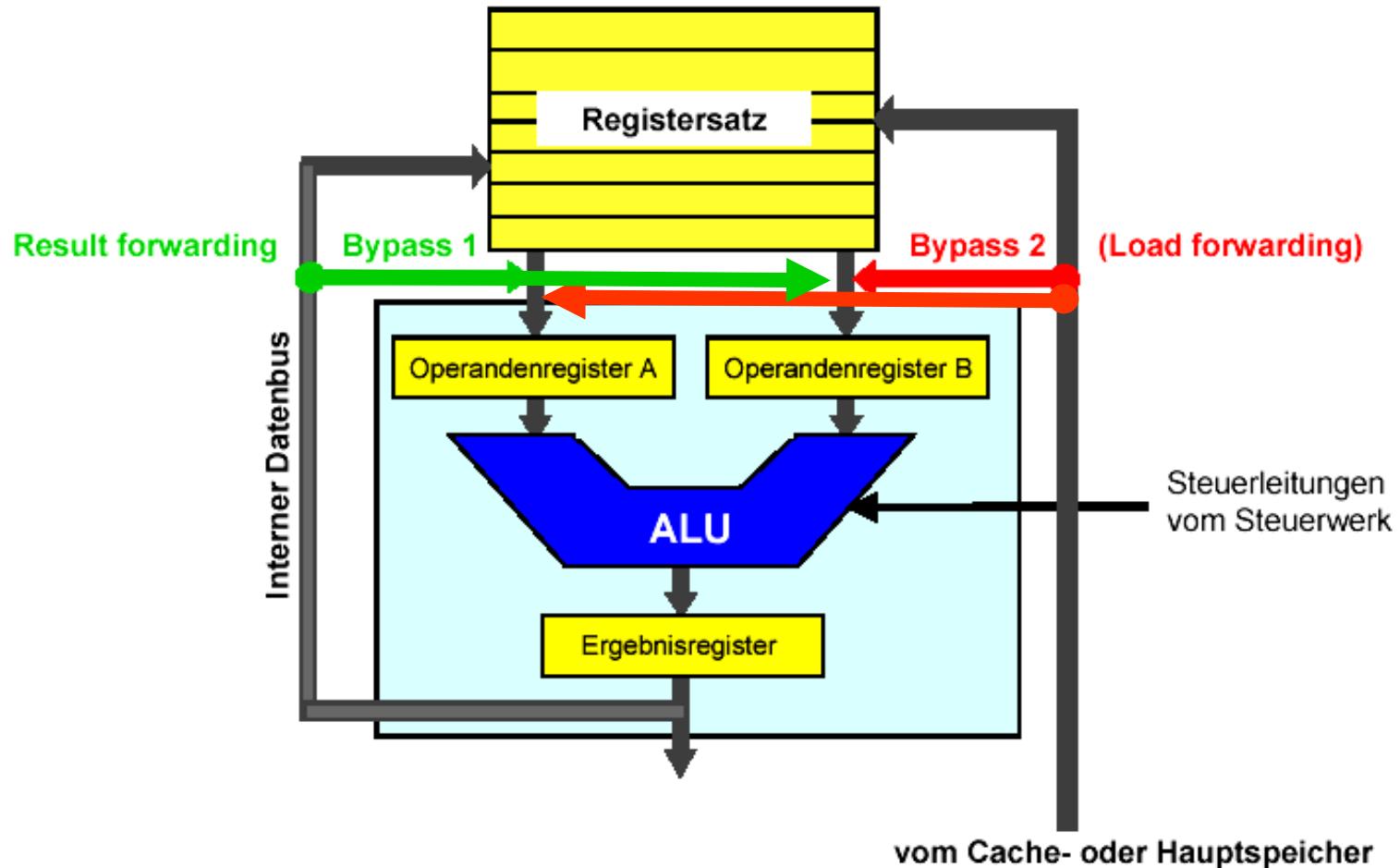
Lösung der Datenkonflikte

Auflösung von Datenkonflikten: Forwarding

- Einführung von "Abkürzungspfaden" in der Pipeline
- Erhöhter Hardware-Aufwand, aber kein Warten notwendig
- Beispiel bei unserer Pipeline:
 - Operanden werden spätestens zu Beginn der EX-Phase benötigt
 - ALU-Ergebnis liegt am Ende der EX-Phase vor
 - Daher ist folgende Ausführung prinzipiell möglich:



Bypass - Techniken



Lösung der Datenkonflikte

Auflösung von Datenkonflikten: Forwarding mit Interlocking

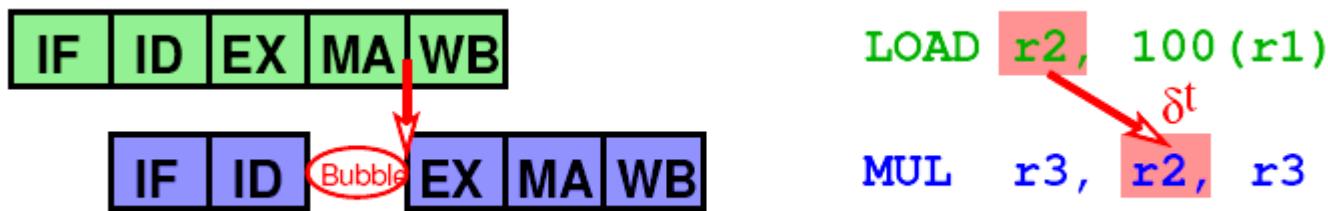
- Nicht alle Konflikte lassen sich mit Forwarding auflösen
- Beispiel: Load-Befehle
- Problem: Wert ist erst am Ende der MA-Phase verfügbar



Lösung der Datenkonflikte

Auflösung von Datenkonflikten: Forwarding mit Interlocking

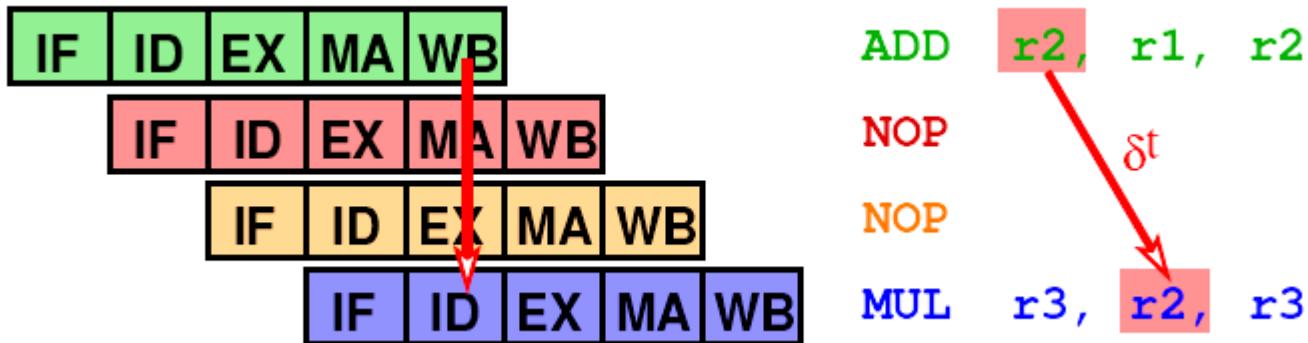
- Nicht alle Konflikte lassen sich mit Forwarding auflösen
- Beispiel: Load-Befehle
- Lösung: Kombination aus Forwarding und Interlocking



Lösung der Datenkonflikte

Auflösung von Datenkonflikten

- Software-Lösung: durch den Compiler (**statische Verfahren**)
 - Compiler muss Datenkonflikte erkennen
 - Einfügen von Leerbefehlen, um Konflikte zu beseitigen



- Optimierung: Umordnen des Programms so, dass statt der Leerbefehle nützliche Befehle verwendet werden (**Instruction Scheduling, Pipeline Scheduling**).

Kurzes Summary

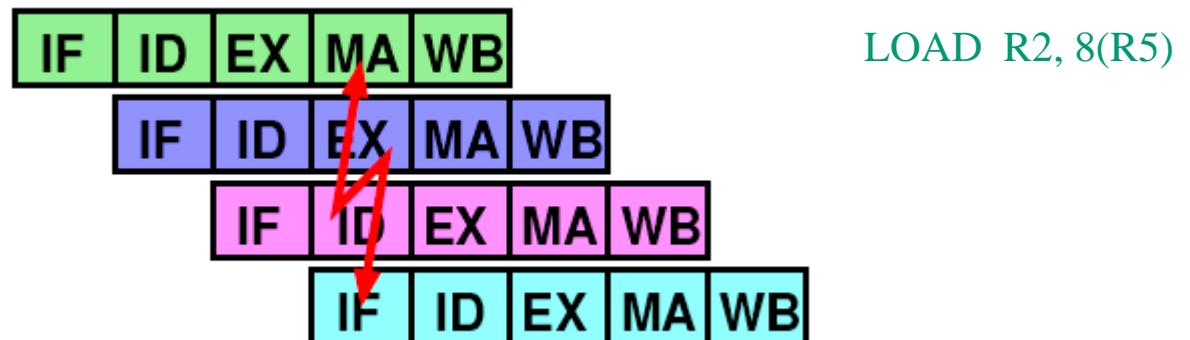
Was haben wir bisher behandelt?

- Prinzip des Pipelining
 - Befehlsausführung in Phasen unterteilen und dabei
 - unterschiedliche Phasen aufeinanderfolgender Befehle gleichzeitig bearbeiten
- Im Idealfall hoher Leistungsgewinn
- Problem: Pipeline-Konflikte
- Datenkonflikte durch Datenabhängigkeiten
- => Konfliktauflösung:
 - Statisch: Compiler ist verantwortlich
 - Dynamisch: Prozessor ist verantwortlich
 - Interlocking: Einführen von Leerzyklen
 - Forwarding: Abkürzungs-Datenpfade

Strukturkonflikte der Pipeline

Strukturkonflikte

- Ein **Strukturkonflikt (Ressourcenkonflikt)** tritt auf, wenn bei der überlappten Ausführung zweier Befehle gleichzeitig auf ein nur einfach vorhandenes Betriebsmittel zugegriffen wird.
- Beispiel: Pipeline mit nur einem Speicher für Befehle und Daten
 - IF und MA greifen beide auf diesen Speicher zu
- Konfliktsituation:



Strukturkonflikte der Pipeline

Strukturkonflikte: Lösungen

- Arbitrierung mit Interlocking
 - Auflösung des Konflikts durch die Hardware
 - Anhalten des späteren, am Konflikt beteiligten Befehls
 - Einfügen von Pipeline-Leerzyklen
- ”
- Ressourcenreplizierung
 - Ressourcen werden mehrfach vorgesehen
 - Z.B. getrennte Daten- und Befehlsspeicher
 - Speicher erlauben mehrere gleichzeitige Zugriffe (Multiport)

Steuerflusskonflikte der Pipeline

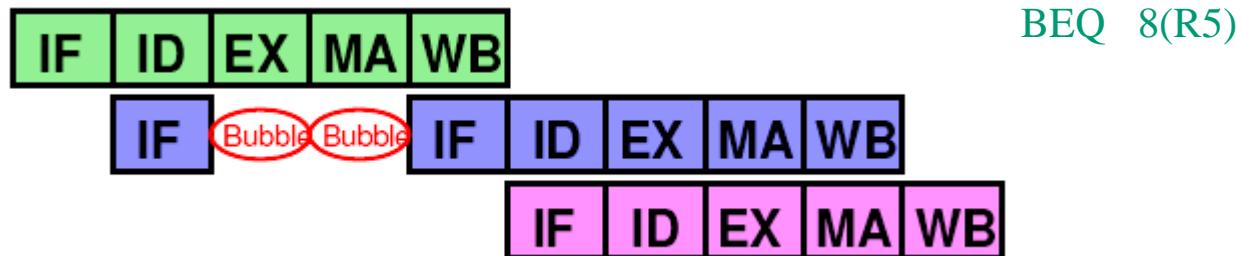
Steuerflusskonflikte

- Ein **Steuerflusskonflikt** tritt nach dem Laden eines Sprung- oder Verzweigungsbefehls auf, da vor der Bestimmung der Zieladresse nicht feststeht, von wo die nächsten Befehle geladen werden müssen.
- **Beispiel: Unsere Pipeline**
 - Berechnung der Sprungbedingung und des Sprungziels in der EX-Phase
 - Zieladresse wird in der MA-Phase in den PC geschrieben
 - d.h.: erst drei Takte nach dem Laden eines Sprungs steht fest, ab welcher Adresse die nachfolgenden Befehle zu laden sind.

Steuerflusskonflikte der Pipeline

Steuerflusskonflikte: Einfachste Lösung

- Anhalten der Pipeline, sobald Sprung erkannt wurde (ID-Phase)



- Alter Wert von IF wird gelöscht, nach Berechnen des Sprungziels (MA-Phase) muss IF wiederholt werden (neue Sprungziel-Adresse). Bis dahin Pipeline 2 Takte anhalten.
 - Großer Leistungsverlust, besonders bei langen Pipelines

Steuerflusskonflikte der Pipeline

- Möglichst frühe Berechnung von Sprungziel und Bedingung
 - ID Phase wäre geeignet
- Probleme bei Berechnung in ID-Phase:
 - Strukturkonflikt: Sprungzielberechnung benötigt ALU
 - zusätzliche ALU (Addierer) in ID-Phase notwendig
- In unserer betrachteten DLX-Pipeline **werden Steuerflusskonflikte durch die Hardware weder erkannt noch behandelt**, d. h. die drei Befehle hinter einem Sprungbefehl werden immer ausgeführt.
- Techniken zur Konfliktauflösung
 - Software-Techniken
 - Hardware-Techniken

Lösung der Steuerflusskonflikte

Software-Lösung

- **Verzögerte Sprungtechnik (delayed branch technique)**
 - Ausfüllen der Verzögerungszeitschlitte (bubbles) mit Leerbefehlen (noop)
 - Unsere Pipeline: Drei Leerbefehle nach jedem Programmsteuerbefehl
- **Statische Befehlsanordnung:**
 - Der Compiler verschiebt Befehle, die in der logischen Programmreihenfolge vor dem Sprungbefehl liegen, in die Verzögerungszeitschlitte
 - Nur dann möglich, wenn die verschobenen Befehle keinen Einfluss auf die Sprungrichtung haben
 - Gibt es keine Befehle, die in die Verzögerungszeitschlitte verschoben werden können, müssen Leerbefehle eingefügt werden
- **Nachteil: Code wird abhängig von der Pipeline-Struktur**

Lösung der Steuerflusskonflikte

Hardware-Lösung (dynamische Technik)

- Pipeline-Leerlauf:
 - Einfachste und ineffizienteste Methode
 - Hardware erkennt Verzweigungsbefehle (in der ID-Stufe) und lädt keine weiteren Befehle in die Pipeline, bis die Zieladresse berechnet und im Falle bedingter Sprungbefehle die Sprungentscheidung getroffen ist.
 - Der eine Befehl, der bereits ins Pipeline-Register der IF-Stufe geladen wurde, muss gelöscht werden.
- Spekulation auf nicht genommene bedingte Sprünge:
 - Einfachste Technik der statischen Sprungvorhersagen
 - Annahme: Sprung wird nicht „genommen“. Nachfolgende Befehle werden in die Pipeline geladen. Falls der Sprung „genommen“ wird, müssen die drei Befehle gelöscht werden (Pipeline flushing).
 - Nachteil: Ineffizient, da bedingt durch die in Programmen häufig auftretenden Schleifen die Mehrzahl der Sprünge genommen wird.

Lösung der Steuerflusskonflikte

Weitere Technik zur Vermeidung von Wartezyklen:

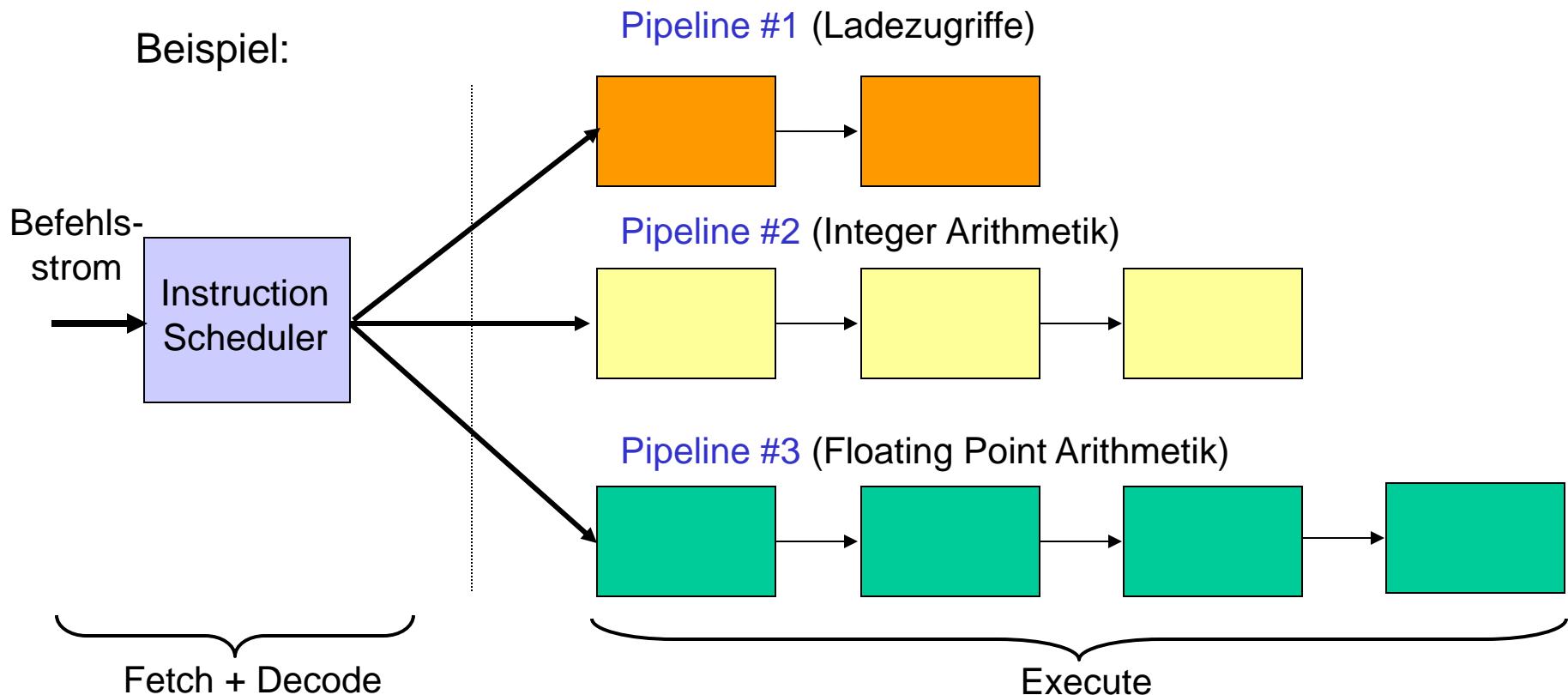
- Sprungvorhersage (Branch Prediction)
 - Compiler und/oder Hardware machen eine Vorhersage des wahrscheinlichsten Sprungziel
 - spekulative Ausführung der Befehle, die dem Sprung folgen oder die am Sprungziel stehen.
- Nach Auswertung der Sprungbedingung:
 - Fortfahren mit der Ausführung ohne Verzögerung bei korrekter Vorhersage.
 - Verwerfen der geholten Befehle bei falscher Vorhersage

Superskalarprinzip und Sprungvorhersage

Parallel - Pipelines

- Komplexere Mikroprozessoren verfügen über mehr als eine Pipeline.

Beispiel:



Parallel - Pipelines

- Aus der Struktur der gezeigten Pipelines ist ersichtlich, dass sich Befehle in der Abwicklung prinzipiell „überholen“ können. Obwohl der Scheduler die Befehle in der vorgegebenen Reihenfolge holt (**in-order-issue**), können sie außerhalb dieser Ordnung beendet werden (**out-of-order processing/completion**), damit können Pipeline-Hemmisse entstehen.
- Aufgrund der dargestellten Struktur kann höchstens 1 Befehl pro Zyklus in die Pipeline fließen (Fetch + Decode des Instruction Schedulers). Damit wird auch nur 1 Befehl im Mittel pro Zyklus fertig gestellt: **IPC ≈ 1** (Instructions per Cycle).
- Will man den IPC erhöhen, muss man mehr als einen Befehl parallel in die Pipelines einspeisen und bearbeiten können.

Problematik

- Die ununterbrochene Versorgung der „Execute Units“ ist i. A. aber schwer erreichbar, da das Programm dann so geartet sein müsste, dass immer alle Pipelinestufen parallel gespeist werden können (z.B. Integer - und Floating-Point-Pipeline).
- **Beispiel:** Intel i860 besitzt
 - Pipeline für Gleitpunkt-Multiplikation
 - Pipeline für Gleitpunkt-Addition
 - Pipeline für Integer-Operation

Idealfall: $IPC = 3$, jedoch gemessene Werte: $IPC \approx 1,1$
(d.h. im Mittel sind 2 der 3 Rechenwerke nicht beschäftigt)

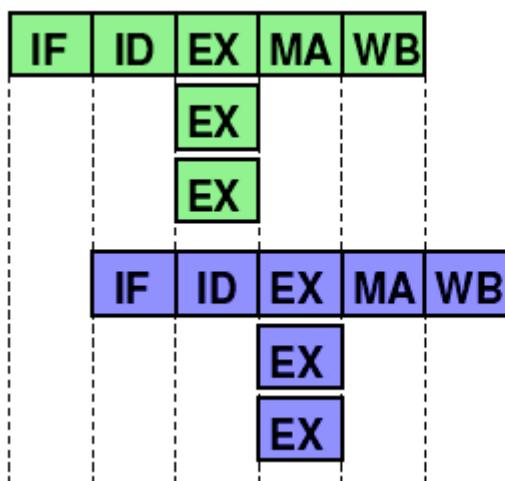
Idee der VLIW-Prozessoren

- In einem Befehlswort werden mehrere parallel ausführbare Operationen kodiert
- Befehlsworte sind dadurch typischerweise sehr lang
- Befehlsworte werden sequentiell geladen, dekodiert und ausgeführt (Pipelining)
- Ausführung eines Befehls erfolgt parallel und synchron in allen Funktionseinheiten
 - Die Latenzzeiten und Initiierungsintervalle der einzelnen Funktionseinheiten müssen im Code berücksichtigt werden.

Rechenwerk	Latenz	Initiierungs-intervall
Integer ALU	1	1
FP Add	4	1
FP Multiply	7	1
FP Divide	25	25

VLIW - Prozessoren

- Prozessoren mit einer großen Anzahl gleichzeitig initialisierbarer Befehle werden als VLIW-Prozessoren (Very Long Instruction Word) bezeichnet. Diese bestehen aus einer Anzahl von Ausführungseinheiten, die jeweils mehrere Maschinenoperationen eines VLIW-Befehlspakets taktsynchron zueinander ausführen können (Anzahl der Befehle pro VLIW-Paket = Anzahl der verfügbaren Ausführungseinheiten).
 - Befehlsworte dadurch typischerweise sehr lang



Beispiel:
VLIW-Prozessor mit
drei Ausführungs-
einheiten

VLIW - Prozessoren

- VLIW bezeichnet also eine Architekturvariante, bei der ein Compiler eine feste Anzahl von einfachen, voneinander unabhängigen Befehlen in einem Maschinenbefehlswort fester Länge zusammenpackt (**128 bis 1024 Bits**). Die Ausführung eines Befehls erfolgt parallel und synchron in allen Ausführungs-einheiten.
- Alle Befehle innerhalb eines VLIW-Befehlspakets müssen unabhängig voneinander sein. Wenn die volle Breite eines VLIW-Befehlsworts nicht ausgenutzt werden kann, muss mit Leerbefehlen aufgefüllt werden. Es wird nur eine statische Befehlszuordnung (per Compiler) unterstützt.
- Beispiele:
 - TriMedia (MP3-, DVD-Player, Camcorder)
 - Itanium-Prozessor
 - Cray-Prozessoren

VLIW – Prozessorstruktur und Befehlszuteilung

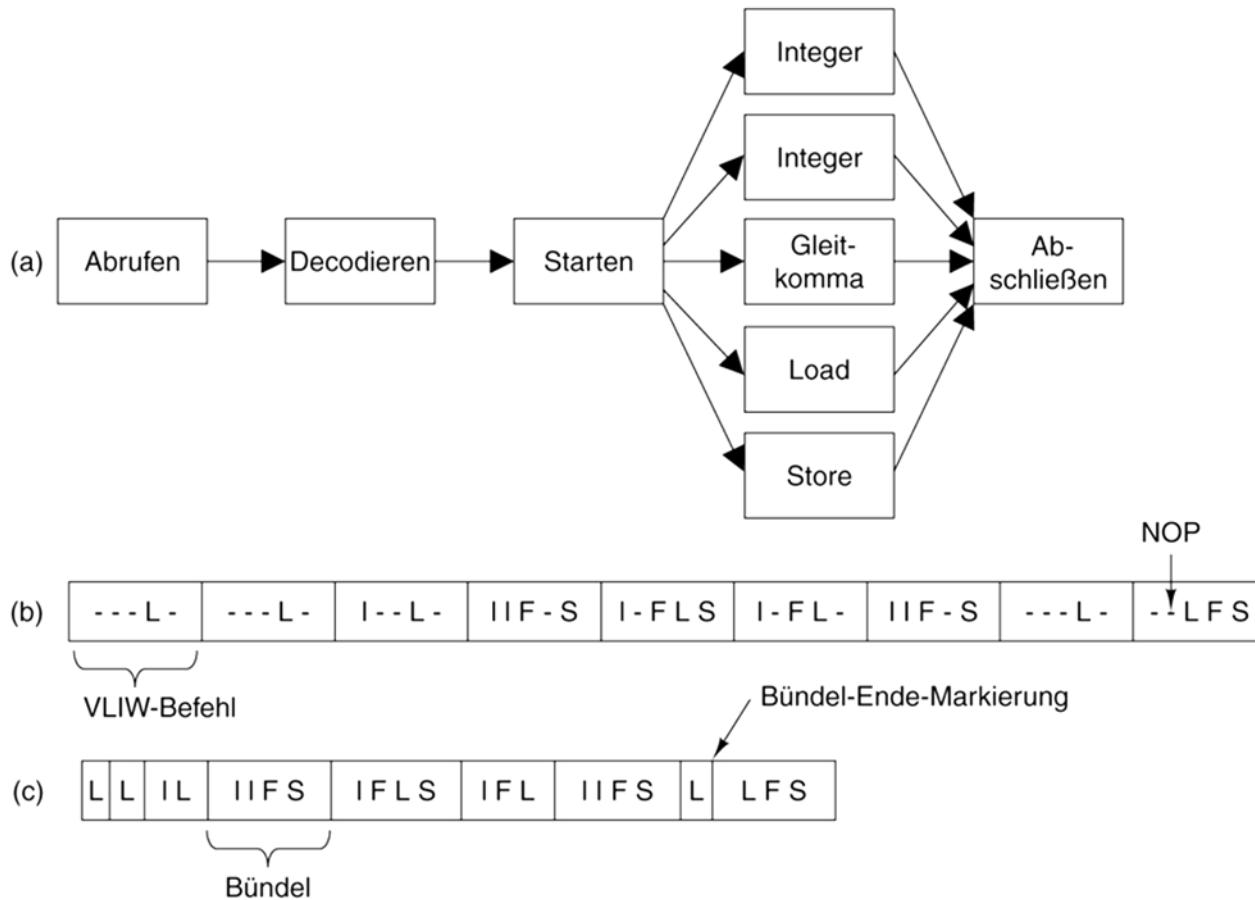


Abbildung 8.2: (a) Eine CPU-Pipeline (b) eine Sequenz von VLIW-Befehlen (c) ein Befehsstrom, in dem die Bündel markiert sind

Ein optimierender Compiler muss „kompatible“ Befehle finden und als Bündel vorbereiten. Der Prozessor bearbeitet dann Befehlsbündel ab (Parallelität auf Befehlsebene).

Quelle: A.S. Tanenbaum: Rechnerarchitektur, 6. Auflage

Wie wird die Befehlszuordnung vorgenommen?

- Compiler sucht parallel ausführbare Befehle
 - Spart Hardware auf dem Prozessorchip
 - Compiler sieht komplette Module bzw. Unterprogramme. Dies erhöht die Chance, unabhängige Befehle zu finden.
 - Parallel Abarbeitung ist in der ISA sichtbar und muss explizit berücksichtigt werden (Compiler kennt möglichen Parallelitätsgrad der Befehle).
 - Keine Binärkompatibilität zu alten Prozessoren möglich
- Genaue Aufgaben des Compilers
 - Code-Analyse für Parallelisierung
 - Kontrollflussanalyse
 - Datenflussanalyse
 - Datenabhängigkeitsanalyse
 - Schleifenparallelisierung
 - Loop Unrolling
 - Scheduling
 - Packen der voneinander unabhängigen Befehle in breite Befehlswörter

Kurzer Einschub: Loop Unrooling

- Bei einer Schleife muss bei jedem Durchlauf die Schleifenbedingung geprüft werden, ob ein weiterer Durchlauf stattfinden soll. Diese Prüfung kann einen großen Anteil der Laufzeit ausmachen. So verbringt der Prozessor im Beispiel etwa die Hälfte der Zeit nur mit Kontrollanweisungen.

```
for( i=0 ; i<8 ; i=i+1 )  
    dest[i] = src[i];
```

- Eine mögliche Optimierung hinsichtlich der Ausführungsgeschwindigkeit ist es, die Schleife durch n Kopien ihres Schleifenkörpers zu ersetzen (vollständiges Entrollen). Schleifenkontrollanweisungen und -zähler entfallen dann ganz.

```
dest[0] = src[0];  
dest[1] = src[1];  
dest[2] = src[2];  
dest[3] = src[3];  
dest[4] = src[4];  
dest[5] = src[5];  
dest[6] = src[6];  
dest[7] = src[7];
```

Quelle: Wikipedia

Zusammenfassung VLIW und EPIC

- VLIW-Architekturen vermeiden Hardwarekomplexität zur Bearbeitung expliziter paralleler Operationen
- Compiler ist verantwortlich Parallelität zu erkennen
- Weiterentwicklung: EPIC = *Explicitly Parallel Instruction Computing*
 - EPIC vermeidet Probleme klassischer VLIW-Architekturen
 - Instruktionsformat ist unabhängig von der Zahl der Funktionseinheiten
 - Flexible Gruppierung parallel ausführbarer Operationen,
 - Keine starre Zuordnung zu Funktionseinheiten wie bei VLIW
 - damit: Anzahl der Funktionseinheiten nicht in ISA sichtbar
 - ➔skalierbare Mikroarchitektur möglich
 - Kontroll- und Datenspekulation

Superskalare Pipelines

Superskalarprinzip

- **Superskalare Prozessoren**

- Ein superskalarer Prozessor verfügt im Vergleich zu einem Prozessor mit sequentieller Pipeline über die **n-fache Anzahl von Ausführungs-einheiten** (es können bis zu **n Befehle gleichzeitig** ausgeführt werden).

- **Voraussetzung**

- In jedem Takt müssen genügend (also n) ausführbare Befehle bereitstehen.

- Als **Superskalaritätsgrad** bezeichnet man die Anzahl der maximal pro Takt zu initialisierenden Befehle (z.B. 2 bis 6).

Superskalaritätsprinzip

- Herausforderung

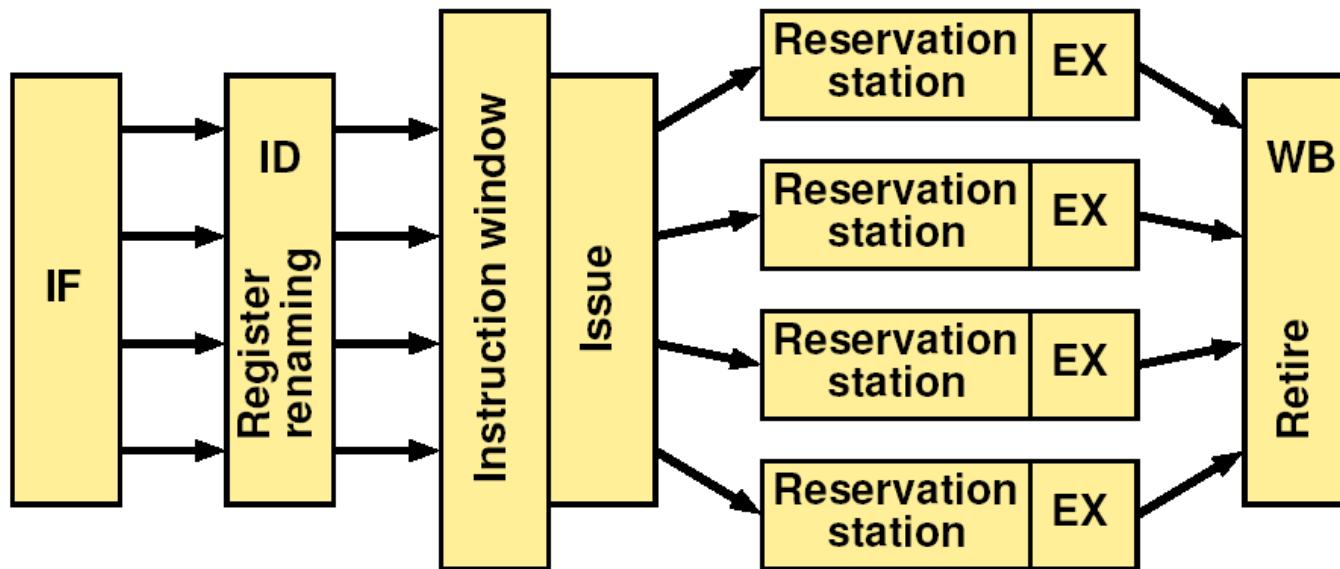
- Semantik der sequentiellen Befehlsabarbeitung wird nach außen hin beibehalten
- Daten- / Struktur- und Steuerungskonflikte müssen von der Hardware behandelt werden,
- Vorarbeiten zur effizienten Konfliktauflösung zur Übersetzungszeit sind erwünscht (Compileroptimierungen)
- Der Abwicklungsprozess (parallele Befehle) muss intern koordiniert werden
- Speicherbandbreite (für Befehle und Daten) muss vorhanden sein

Wie findet man parallel ausführbare Befehle?

- Bei superskalaren Prozessoren: Hardware sucht parallel ausführbare Befehle
 - Hoher Hardware-Aufwand
 - Relativ kleiner Suchraum (Befehlsfenster)
 - Parallel Abarbeitung von außen (in ISA) nicht sichtbar
- Dynamic Scheduling
 - Die Steuerung der parallelen Ausführung von Maschinenbefehlen erfolgt mit Hilfe von dynamischen Techniken
 - D.h. für die Zuweisung und die Konfliktauflösung ist nur der Prozessor verantwortlich
- Alternative (VLIW): Compiler sucht parallel ausführbare Befehle

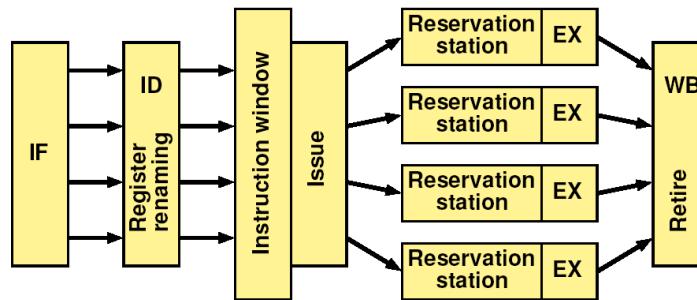
Superskalare Pipeline

Prinzipieller Aufbau einer superskalaren Pipeline



Superskalare Pipeline – IF-Phase

- Holen mehrerer Befehle in den Befehlsholpuffer (IF-Einheit)
 - Mindestens so viele, wie Ausführungseinheiten vorhanden sind
 - Führt Branch-Prediction (Sprungzielvorhersage) durch, d.h. holt die Befehle von der vorhergesagten Programmspeicheradresse
 - Speichert die Befehle der Reihe nach in den Befehlsholpuffer (Instruction Queue)
 - Befehlsholpuffer entkoppelt IF- und ID-Phase

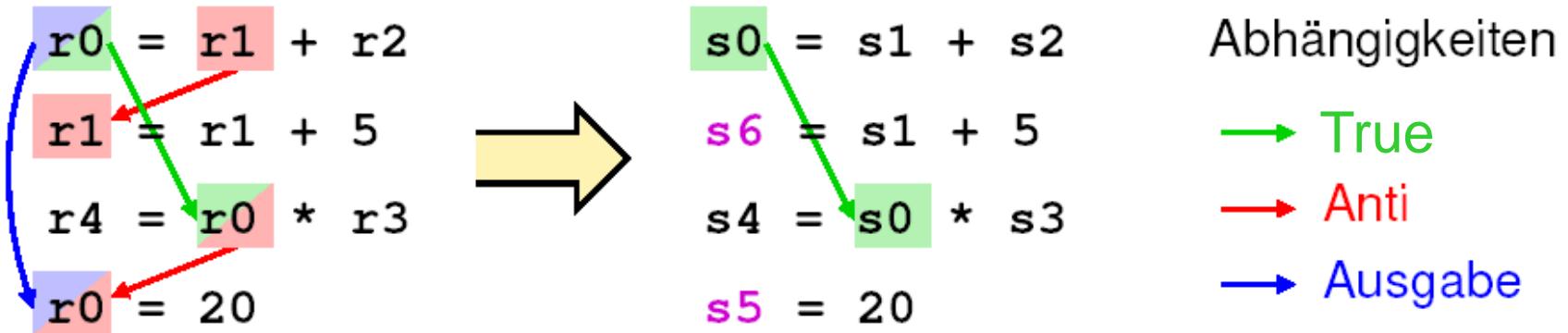


Superskalare Pipeline – ID-Phase

- Dekodierung der Befehle
 - Mindestens so viele gleichzeitig, wie Ausführungseinheiten vorhanden sind (Erhöhung der Auswahlmöglichkeiten für die Zuordnung)
- Aber: Datenabhängigkeiten superskalarer Prozessoren
 - RAW: echte Abhängigkeit, im besten Fall ist Forwarding möglich, in superskalaren Pipelines jedoch extrem aufwändig
 - WAR, WAW: „Register Renaming“ als Lösung
- Register Renaming (Umbenennung von Registern in Reservation Stations)
 - Hardware löst Datenabhängigkeiten innerhalb der Pipeline auf
 - Zwei Registersätze sind vorhanden
 - 1. Architektur-Register: „Logische Register“ der ISA
 - 2. Viele Hardware-Register: „Rename Register“ (physische Prozessorregister)
 - Es erfolgt eine dynamische Abbildung von ISA- auf Hardware-Register

Superskalare Pipeline – ID-Phase

- Registerumbenennung
 - Auflösung von Anti- und Output-Abhängigkeiten (Dependencies)



- Registertabelle
 - Zuweisung : Architekturregister → physische Register
 - Physisches Register muss frei sein

Superskalare Pipeline – ID-Phase

- Weiteres Beispiel
 - Register-Umbenennung und Parallelisierung des Codes

Originalcode

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

nach Renaming

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

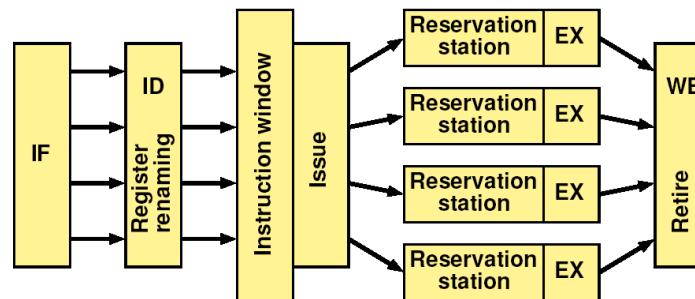
Parallelisierung des modifizierten Codes

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

Quelle: Universität Hamburg, Informatik

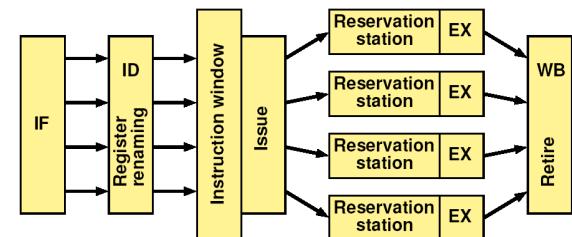
Superskalare Pipeline – ID-Phase

- Schreiben der dekodierten Befehle in das Befehlsfenster
 - Befehle im Befehlsfenster (instruction window) sind frei von Anti- und Ausgabeabhängigkeiten (wegen Registerumbenennung)
 - Unterscheidung: Ohne oder mit spekulativer Ausführung von Operationen
 - Entweder werden Steuerkonflikte bereits hier aufgelöst, oder Instruktionen werden spekulativ ausgeführt.
 - Einfache Annulierung von spekulativ ausgeführten Befehlen bei falscher Sprungzielvorhersage



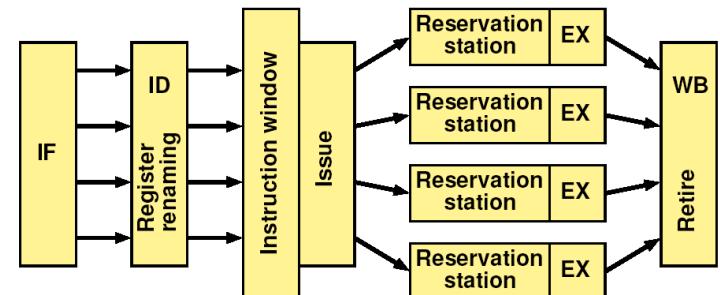
Superskalare Pipeline – Instruction Issue

- Ziel: Zuordnen von Befehlen zu Ausführungseinheiten
 - Befehle werden zunächst von Umordnungspuffern (**Reservation Stations**) aufgenommen. Reservation Stations sind Puffer mit ein oder mehreren Einträgen.
 - Weiterleitung an die Ausführungseinheit, wenn alle Operanden verfügbar sind
 - Out-of-Order execution
 - Keine Berücksichtigung der Ordnung im Programm
 - Nur Berücksichtigung von echten Datenabhängigkeiten
- Ablegen der logischen Programmreihenfolge der Befehle in den Rückordnungspuffer (**Reorder Buffer**), wichtig für WB-Phase.
- Auflösung von RAW Datenkonflikten
 - Befehle warten in Reservation Station, bis alle Operanden verfügbar sind.



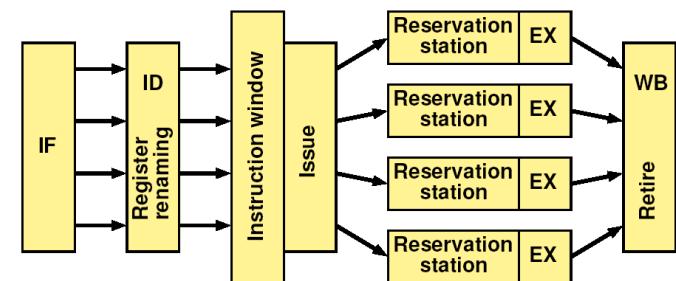
Superskalare Pipeline - Dispatch

- Befehle werden gestartet (Execution), sobald die Operanden verfügbar und die Ausführungseinheit frei ist.
 - Dispatch: Übergang vom Warten zur Ausführung
- Befehle können sich null oder mehrere Takte in der Reservation Station aufhalten.
 - Falls bei Zuordnung (Issue) die Ausführungseinheit frei und Operanden schon verfügbar sind, kann die Ausführung sofort beginnen.
- Dispatch und Ausführung müssen nicht gemäß der Programmordnung (in-order) erfolgen.



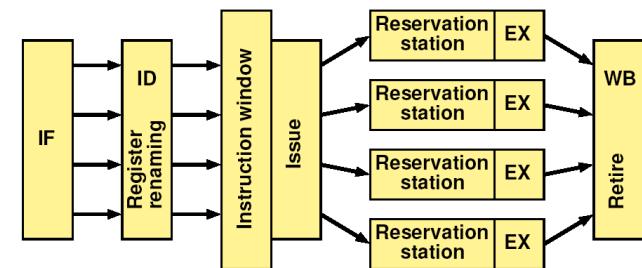
Superskalare Pipeline – Completion

- Befehlsausführung ist vollständig (*complete*), wenn:
 - Befehl die Ausführungseinheit nicht mehr belegt
 - Ergebnis ggf. über Forwarding für andere Befehle bereitsteht
 - Completion ist unabhängig von der Programmordnung
- Abschluss:
 - Bereinigung der Reservierungstabelle in den Reservation Stations
 - Aktualisierung des Zustands im Reorder Buffer. Der Zustand zeigt u.a. an, ob
 - Eine Unterbrechung (Ausnahme) aufgetreten ist
 - Der vollständige Befehl noch von einer Spekulation abhängt



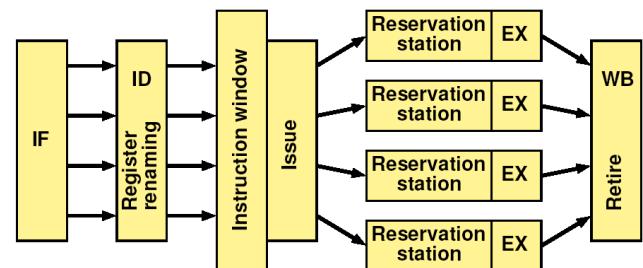
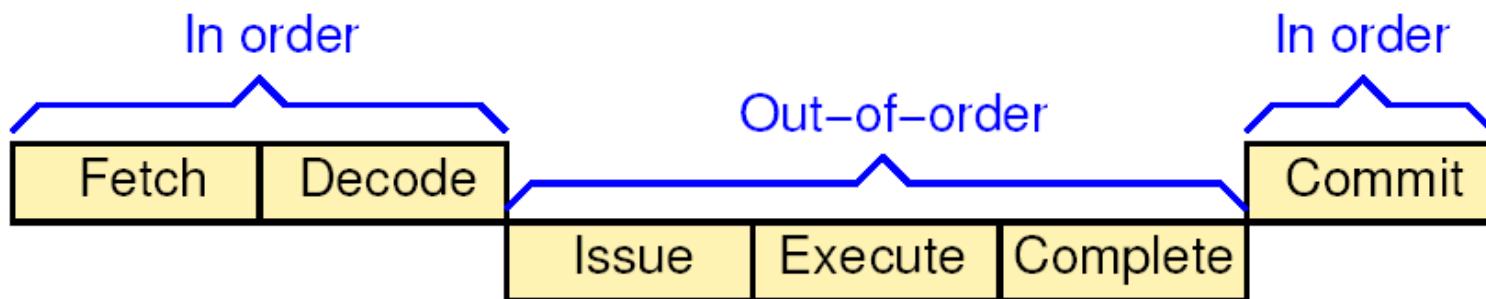
Superskalare Pipeline – Commitment

- Bedingungen für Commitment
 - Die Befehlsausführung ist vollständig
 - Alle Befehle, die in der Programmordnung vor dem Befehl stehen, haben ihre Bearbeitung beendet (sind *committed*) oder beenden sie im selben Takt
 - Es ist während der Ausführung keine Unterbrechung aufgetreten
 - Der Befehl hängt nicht mehr von einer Spekulation ab
- Mit dem Commitment werden die Ergebnisse der Befehle wieder in den Architekturregistern abgelegt
 - Durch Rückschreiben aus den Umbenennungsregistern (Schattenregistern)
- Die Programmreihenfolge der Befehle ist im Rückordnungspuffer (Reorder Buffer, WB) abgelegt
 - Wird benötigt in der WB-Phase



Superskalare Pipeline – Retirement

- Freigeben des Platzes im Reorder Buffer
 - Nach Commitment: Ergebnis ist dauerhaft
 - Nach Verwerfen des Befehls: ohne dauerhafte Zustandsänderung
- Die Abarbeitungsreihenfolge der Befehle ist wieder hergestellt



Leistungssteigerung durch Multithreading

- Ein Thread ist ein eigenständiger Handlungsfaden (Kontrollfaden, Befehlsstrom) innerhalb eines Programms.
- Threads teilen sich (im Gegensatz zu Prozessen) sämtliche Systemressourcen mit Ausnahme der Prozessorregister, d.h. gemeinsamer Speicher, gemeinsame Dateien, ...
 - Anwendungsbeispiele:
 - Zur einfachen Programmierung nebenläufiger Vorgänge, z.B. Internet-Browser: Laden der Seite vom Netzwerk / Bedienung von Benutzereingaben
 - Zur parallelen Abarbeitung von Prozesssträngen

Multithreading im Prozessor

- Was bringt Multithreading für den Prozessor?
 - Überbrückung von Wartezeiten (Latency Hiding), z.B. bei Speicherzugriff
 - Aktueller Thread ist blockiert
 - Aber: andere Threads können in dieser Zeit ausgeführt werden
 - Bessere Ausnutzung der Funktionseinheiten erreichbar
- Im Prozessor nötig:
 - Mehrere Registersätze (inklusive Befehlszähler)
 - Steuerung des Threadwechsels / Kontrolle und Administration

Varianten des Multithreading

1. Cycle-by-cycle Interleaving (*fine-grain multithreading*)

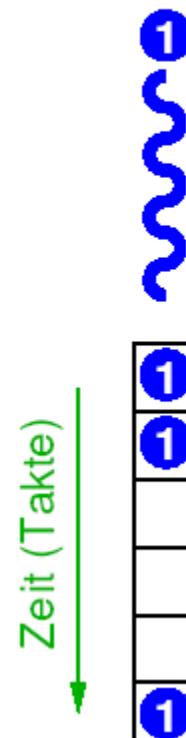
- Eine Anzahl von Threads ist geladen (Zustand *Ready*)
- Der Prozessor wählt in jedem Takt einen der ausführungs bereiten Threads aus.
- Der nächste Befehl in der Befehlsreihenfolge des ausgewählten Threads wird zur Ausführung gebracht.

2. Block Interleaving (*coarse-grain multithreading*)

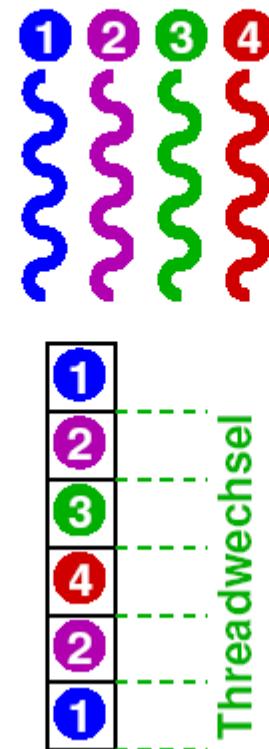
- Befehle eines Threads werden so lange ausgeführt, bis ein Befehl mit langer Latenzzeit ausgeführt wird
- Dann Wechsel zu einem anderen Thread (zur Kompensation der Latenzzeit)

Vergleich der Varianten für Single Scalar Prozessor

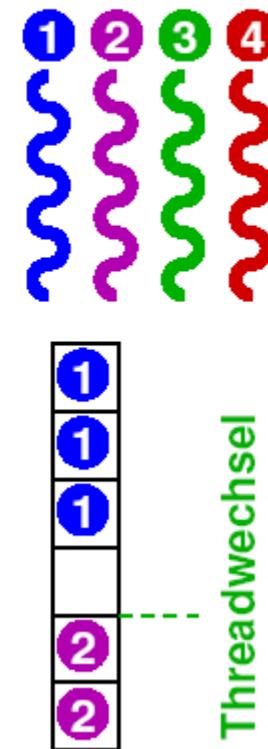
Single-threaded scalar



Cycle-by-cycle interleaving multithreaded scalar

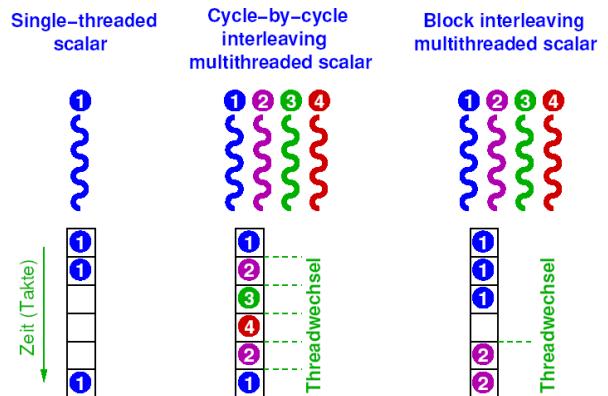


Block interleaving multithreaded scalar



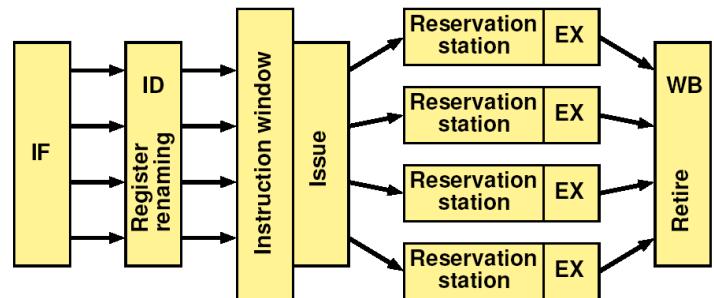
Bewertung für Single Scalar Prozessor

- Single-Threaded scalar
 - Leerzyklen behindern den Thread
- Cycle-by-cycle
 - Voraussetzung: Es sind mindestens so viele Threads vorhanden wie Pipelinestufen.
 - Alle Befehle sind unabhängig
- Block-Interleaving
 - Bestimmte Operationen (z.B. Load/Store) können zu mehreren Leerzyklen führen.
 - Umschalten erfolgt erst beim Auftreten dieser Operationen.
 - Es kommt nur zu einem oder wenigen Leerzyklen.



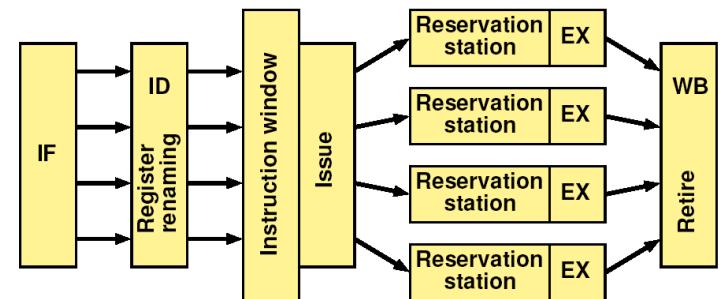
Simultaneous Multithreading

- **Simultaneous Multithreading (SMT)**
 - Wird angewendet bei superskalaren Prozessoren
 - Die Ausführungseinheiten werden über eine Zuordnungseinheit aus mehreren Befehlspuffern (Reservation Stations) versorgt
 - Jeder Befehlspuffer liefert den Befehlsstrom eines anderen Threads
 - Jedem Befehlsstrom ist ein eigener Registersatz zugeordnet
- **Beispiele:**
 - Intel P4 Hyperthreading
 - IBM Power 5, 2 CPUs auf einem Chip, jede mit zwei Threads



Bewertung – Superskalarer Prozessor

- **Simultaneous Multithreading**
 - Threads werden nicht nur scheinbar sondern tatsächlich simultan ausgeführt.
 - Instruktionen mehrerer Threads füllen die Funktionseinheiten im gleichen Takt (multi-Issue).
- **Insgesamt**
 - Einzelner Thread wird nicht schneller
 - Aber der Durchsatz steigt



Vergleich für Multi-Issue-Prozessoren

- Static Scheduling: Compiler bestimmte Reihenfolge der Ausführung
- Dynamic Scheduling: Hardware bestimmt Reihenfolge der Ausführung
- VLIW (Very Long Instruction Word): Compiler packt mehrere Instruktionen zusammen
- EPIC: Explicit Parallel Instruction Computing

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Cortex-A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Sprungvorhersage

Problem der Verzweigungsentscheidungen

- In „typischen Programmen“ sind etwa
 - 20% aller Befehle Verzweigungsbefehle
 - 66% davon führen die Verzweigung auch durch (Branch Taken), d.h. es kommt zur Programmflussänderung
- Mögliche Lösung (z.B. MC68040): Zwei parallele Pipelines, die für die beiden möglichen Sprungziele die Folgebefehle bereithalten (Branch Taken / Not Taken). Sobald der endgültige Programmablauf feststeht, wird auf die „richtige“ Pipeline umgeschaltet.
 - Hoher Hardwareaufwand (dual-ported Caches für Simultanzugriffe), wird sehr komplex bei aufeinander folgenden Branch-Befehlen (dann eventuell Hardware Interlocking).
- Man benötigt deshalb Verfahren, die die Entscheidung für Verzweigungsbefehle vorhersagen, diese nennt man **Sprungvorhersagen (Branch Prediction)**.

Sprungvorhersage (Branch Prediction)

- Die Auflösung von **Steuerflusskonflikten** in superskalaren Prozessoren erfolgt meist mittels Sprungvorhersage.
- Beim Auftreten einer Verzweigung:
 - Vorhersage des Sprungziels
- Spekulatives Füllen der Verzögerungsphasen mit Befehlen,
 - Die dem Sprung folgen oder
 - Die am Sprungziel stehen
- Nach Auswertung der Sprungbedingung:
 - Fortfahren mit der Ausführung ohne Verzögerung bei korrekter Vorhersage (Vermeidung der aufwändigen Reorganisation der Pipeline)
 - Verwerfen der geholten Befehle bei falscher Vorhersage

Sprungvorhersage

- **Statische Sprungvorhersage**
 - Die Richtung der Vorhersage ist für einen Befehl immer gleich
- **Dynamische Sprungvorhersage**
 - Die Vorhersage hängt von der Vorgeschichte ab

Statische Sprungvorhersage

- **Hardware**
- Sprungvorhersage im Prozessor fest verdrahtet. Einfachster Fall: alle Verzweigungsbefehle werden gleichartig vorhergesagt (alle „branch taken“ oder „branch not taken“).
- Statistische Untersuchungen zeigen, dass Rückwärtsverzweigungen in etwa 90% der Fälle ausgeführt werden. Diese Sprungzielvorhersage liefert also eine gute Trefferquote, während Vorwärtsverzweigungen nicht im Vorgriff ausgeführt werden sollten.
- Bei diesen Verfahren wird in der Regel nur ein Sprung vorhergesagt. Tritt ein weiterer Verzweigungsbefehl auf, wird das Laden der Befehlswarteschlange angehalten.

Statische Sprungvorhersage

- Detektiert der Compiler eine Anweisung wie

`for (i=0; i < 1000000; i++) {... }`

Ist klar, dass der Sprung am Ende der Schleife fast jedes Mal erfolgt. Bei n Schleifendurchläufen ist die Vorhersage $(n-1)$ -mal erfüllt. Dies kann im Opcode des Sprungbefehls vermerkt werden.

- Compiler

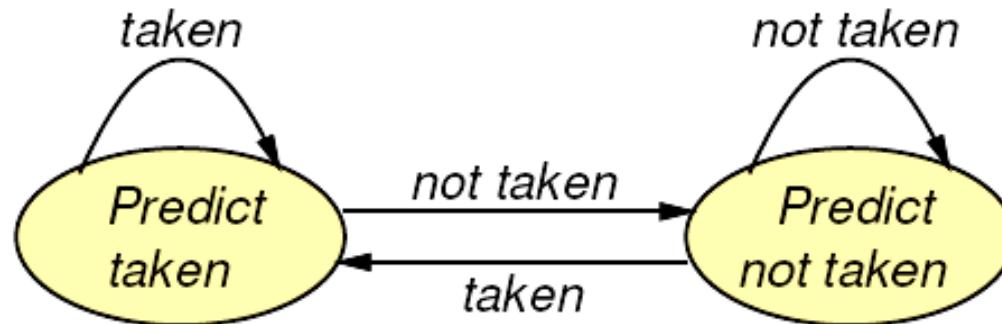
- Kodierung der Vorhersage in einem Bit des Opcodes möglich
- Holen der Befehle vom festgelegten Sprungziel
- Bei falscher Vorhersage werden die geholten Befehle wieder verworfen.
- Vorhersage aufgrund Programmanalyse

Dynamische Sprungvorhersage

- Berücksichtigung des Programmverhaltens
- Diese Verfahren versuchen die Vorhersage des Sprungziels aus der „Vorgeschichte“ des Verzweigungsbefehls zu ermitteln.
- Die Sprungvorhersage erfolgt mittels eines Prädiktor-Modells

Sprungvorhersage - Ein-Bit-Prädiktor

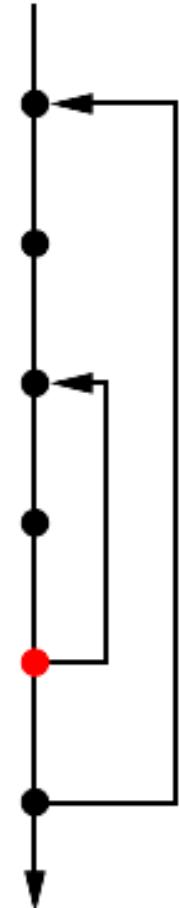
- Ein-Bit-Prädiktor
 - Nur ein Vorhersagebit
 - Falls gesetzt: Annahme, dass Sprung genommen wird
 - Falls gelöscht: Annahme, dass Sprung nicht genommen wird
 - Bei Fehlannahme: Invertieren des Bits



- Vorhersage: Sprung macht dasselbe wie beim letzten Mal“

Sprungvorhersage - Ein-Bit-Prädiktor

- Ein-Bit-Prädiktor
- Schleifensprünge werden korrekt vorhergesagt, solange die Schleife ausgeführt wird
- Aber: schlechte Vorhersage bei geschachtelten Schleifen
 - Im jeweils ersten Durchlauf der innersten Schleife ist die Vorhersage falsch:
 - Letzte Ausführung des Schleifensprungs war bei Austritt aus der Schleife
 - Vorhersage daher: sofortiger Austritt aus der Schleife
- Vermeidung des Problems durch Zwei-Bit-Prädiktor



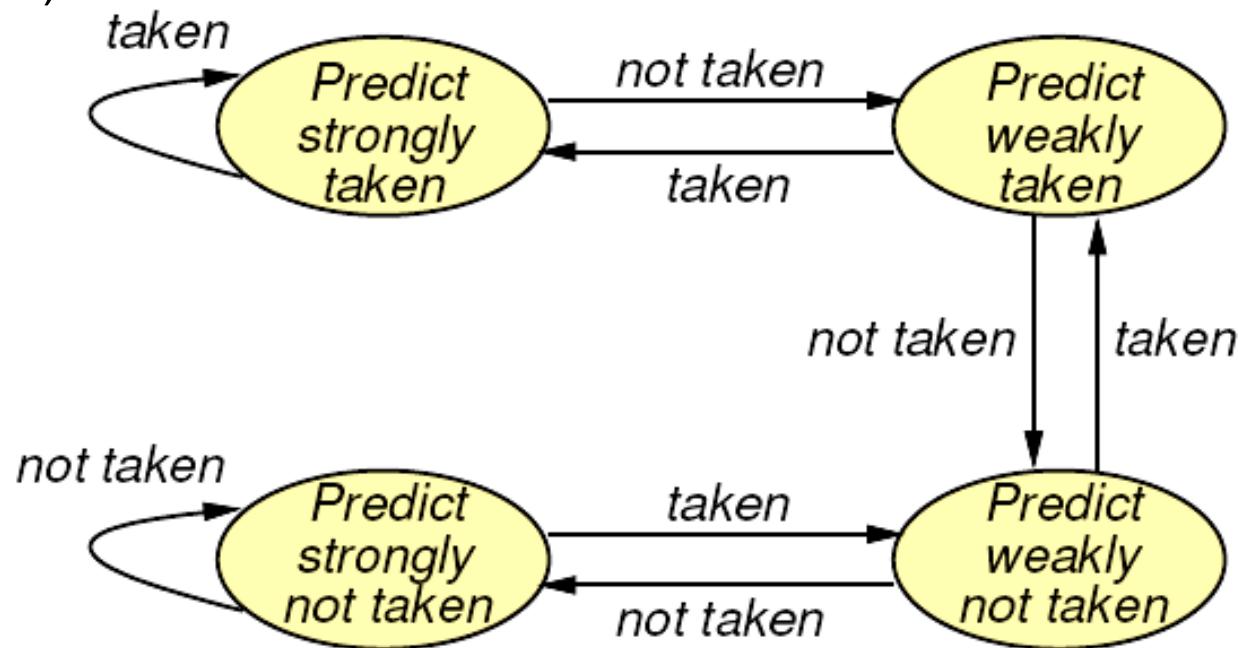
Sprungvorhersage - Zwei-Bit-Prädiktor

- **Zwei-Bit-Prädiktor**
- Zwei Bit pro Eintrag codieren vier mögliche Zustände:
 - sicher genommen (strongly taken)
 - vielleicht genommen (weakly taken)
 - vielleicht nicht genommen (weakly not taken)
 - sicher nicht genommen (strongly not taken)
- In einem „strongly“ Zustand:
 - erst **zwei** Fehlvorhersagen kehren die Vorhersage um
- Der Anfangszustand „strongly taken“ ist die günstigste Wahl für Branch-Befehle am Ende einer Schleife.

Beispiel: Zwei - Bit - Prediktor

Branch Prediction mit 2 Vorhersagebits:

- Jedes Mal, wenn sich (nachträglich) herausstellt, dass die Verzweigungsbedingung nicht erfüllt war, findet im Zustandsdiagramm ein Übergang nach links statt (im anderen Fall nach rechts).



Dynamische Sprungvorhersage

- Eine Hardware-Logik beobachtet zu jedem Branch-Befehl die Verzweigungsentscheidung (Vorhersagebits) und zeichnet diese in den **Branch History Bits** auf, die Teil des **Branch-Target-Buffers** sind. Diese „History“ dient als Entscheidungshilfe bei einer erneuten Verzweigungsentscheidung. Die History wird modifiziert, wenn die Voraussage falsch war. Als Ausgangszustand für den ersten Branch-Befehl werden die History-Bits vorbelegt.
- Dadurch kann in der Decodierphase der Pipeline die Sprungzieladresse vorhergesagt werden.

Implementierung der dynamischen Sprungvorhersage

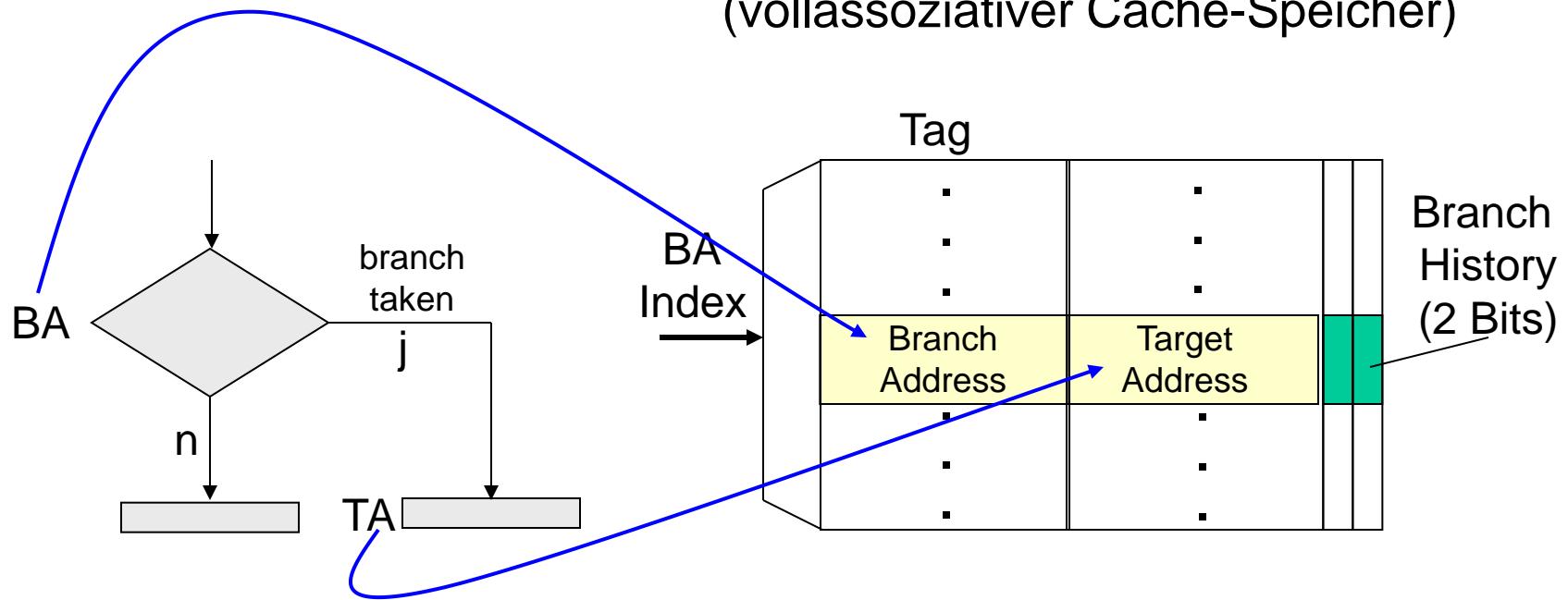
- Die Vorhersagebits und die Sprungzieladressen werden in einem gemeinsamen Cache-Speicher, dem **Branch Target Buffer (BTB)** abgelegt.

Der **BTB** realisiert eine Tabelle, die

- in der 1. Spalte die Adressen der zuletzt ausgeführten Branch-Befehle (Branch Address, BA) und
- in der 2. Spalte die zu den durchgeföhrten Verzweigungen gehörenden Sprungzieladressen (Target Address, TA) enthält.
- In weiteren Spalten werden die 2 History Bits gespeichert, die die aktuelle, aus früheren Durchläufen berechnete Sprungvorhersage enthalten.
- Typische BTBs umfassen 64 bis 512 Einträge.

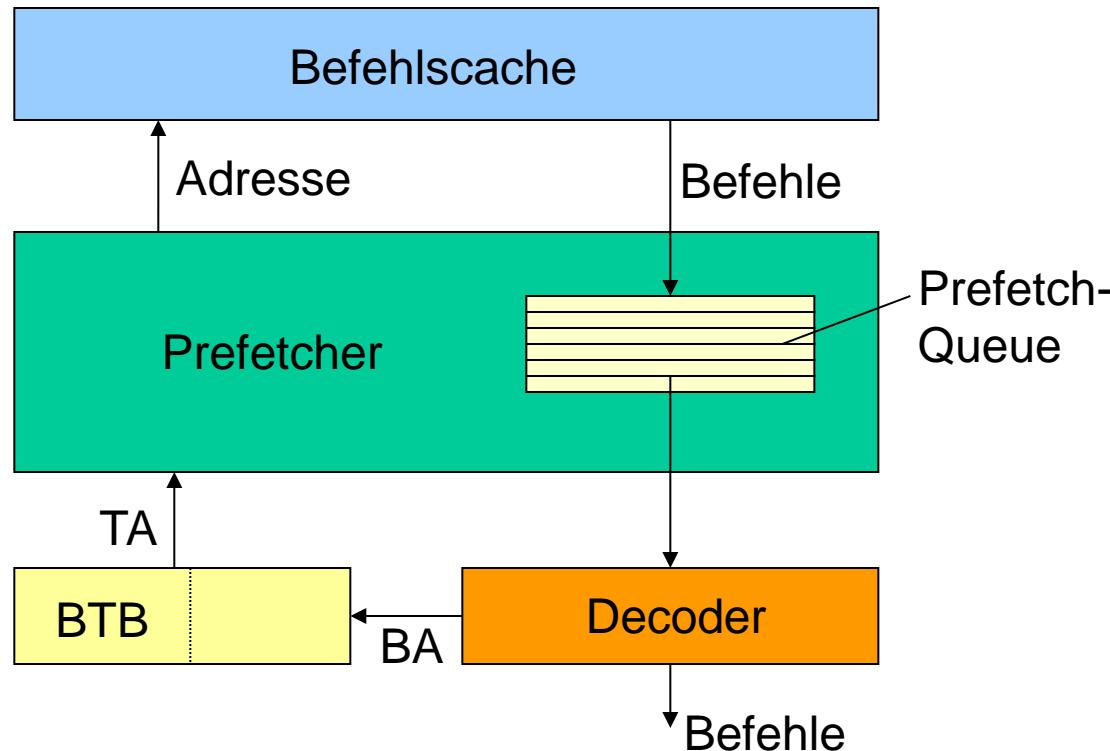
Branch Target Buffer (BTB)

Branch Target Buffer (BTB)
(vollassoziativer Cache-Speicher)



Implementierung der dynamischen Sprungvorhersage

- Sobald der Decoder aus der Prefetch-Queue einen Verzweigungsbefehl liefert bekommt, reicht er dessen Befehlsadresse (BA) an den BTB weiter.



Implementierung der dynamischen Sprungvorhersage

- Adresse ist nicht im BTB enthalten (BTB Miss):
 - Nach berechneter Sprungzieladresse in BTB eintragen
 - History Bits initialisieren (mit Ergebnis)

=> Ablauf erfolgt ohne Einflussnahme des BTB
- Befehlsadresse wird im BTB gefunden (BTB Hit):
 - bei Vorhersage „Branch not taken“ wird ohne Beeinflussung durch den BTB weitergearbeitet
 - bei Vorhersage „Branch taken“ wird Prefetch-Queue nach Verzweigungsbefehl gelöscht. TA im BTB wird an den Prefetcher weitergereicht, der lädt die neue Befehlsfolge in die Prefetch-Queue.
- History Bits werden, abhängig von der wirklichen Entscheidung (Execution phase der Pipeline) im Programm entsprechend aktualisiert.

Implementierung der dynamischen Sprungvorhersage

- Superskalare Prozessoren übergeben pro Taktzyklus 2 bis 6 Befehle an die Pipelines. Da typischerweise 20% Verzweigungsbefehle sind, besteht eine hohe Wahrscheinlichkeit, dass mehrere Branch-Befehle gleichzeitig in Bearbeitung sind. Die maximal mögliche Anzahl ist vom Prozessor abhängig.
- Die Sprungvorhersage bei superskalaren Prozessoren macht nur Sinn, wenn die Befehle an den Sprungzielen bereits ausgeführt werden können, bevor die Sprungvorhersage validiert ist. Die Ausführung „auf Verdacht“ wird **spekulative Befehlsverarbeitung (speculative execution)** genannt. Stellt sich die Vorhersage als falsch heraus, müssen die bis dahin berechneten Ergebnisse der spekulativen, ausgeführten Befehle rückgängig gemacht werden (**branch recovery**).

Nachtrag

Probleme der IA-32 Architektur

- IA-32 besitzt veraltete ISA, harmoniert nicht mit moderner Technologie
- CISC-ISA mit Befehlen unterschiedlicher Länge und vieler verschiedener Formate (unterschiedliche Adressierungsarten)
 - D.h. IA-32 Befehle müssen zur Laufzeit in RISC-ISA-Mikrooperationen zerlegt werden (kostet HW und Zeit).
- IA-32 ist eine speicherorientierte Zweiadressmaschine
 - Schlecht, da Speicherreferenzen viel Zeit kosten (LOAD-/STORE-ISA's mit Drei-Adress-Registerbefehlen sind deutlich besser)
- Die geringe Registerzahl erfordert die Ablage von Zwischenergebnissen im Speicher (daher Zeitprobleme)
- Geringe Registerzahl verursacht viele Abhängigkeiten (Umbenennung von Registern im Umordnungspuffer erforderlich, WAR)
- Wegen der Schnelligkeit der Abarbeitung ist tiefe Pipeline erforderlich, damit viele Taktzyklen. Dies erfordert sehr genaue Sprungvorhersage, da Fehlvorhersagen hohe Kosten verursachen .
- IA-32 beschränkt den linearen Adressraum auf 4 GB.

Probleme der IA-32 Architektur

- Situative Beschreibung:

- Ein Großteil der Transistoren der Intel-NetBurst-Mikroarchitektur oder auch des Core i7 Prozessors sind weitgehend nur damit beschäftigt
- CISC-Befehle zu zerlegen
- Möglichkeiten zur Parallelisierung zu finden
- Konflikte aufzulösen und die
- Folgen falscher Vorhersagen zu korrigieren.

→ Die IA-32 Architektur ist ausgereizt!

→ Abhilfe sollte die IA-64 Architektur bringen

IA-64 - Architektur

- Gemeinsames Projekt von Hewlett-Packard und Intel
- Ziel: 64 Bit Architektur IA-64
 - Explizite Spezifikation des Parallelismus im Maschinencode
 - Bedingte Ausführung von Befehlen (Predication)
 - Spekulative Ausführung von Ladeoperationen (Control and Data Speculation)
 - Großer Registersatz
 - Sinnvolles Zusammenwirken zwischen Compiler und Hardware erforderlich
- *Itanium*: erster IA-64-Prozessor

Speicherhierarchie

Cache - Speicher

Speicherhierarchie

Ein technologisch einheitlicher Speicher mit ***kurzer Zugriffszeit*** und ***großer Kapazität*** ist aus ***Kostengründen*** i.a. nicht realisierbar

Lösung:

Schichtenweise Anordnung verschiedener Speicher und Verschiebung der Information zwischen den Schichten (Speicherhierarchie)

Speicherhierarchie zum Ausgleich der unterschiedlichen Zugriffszeiten der CPU und des Hauptspeichers.

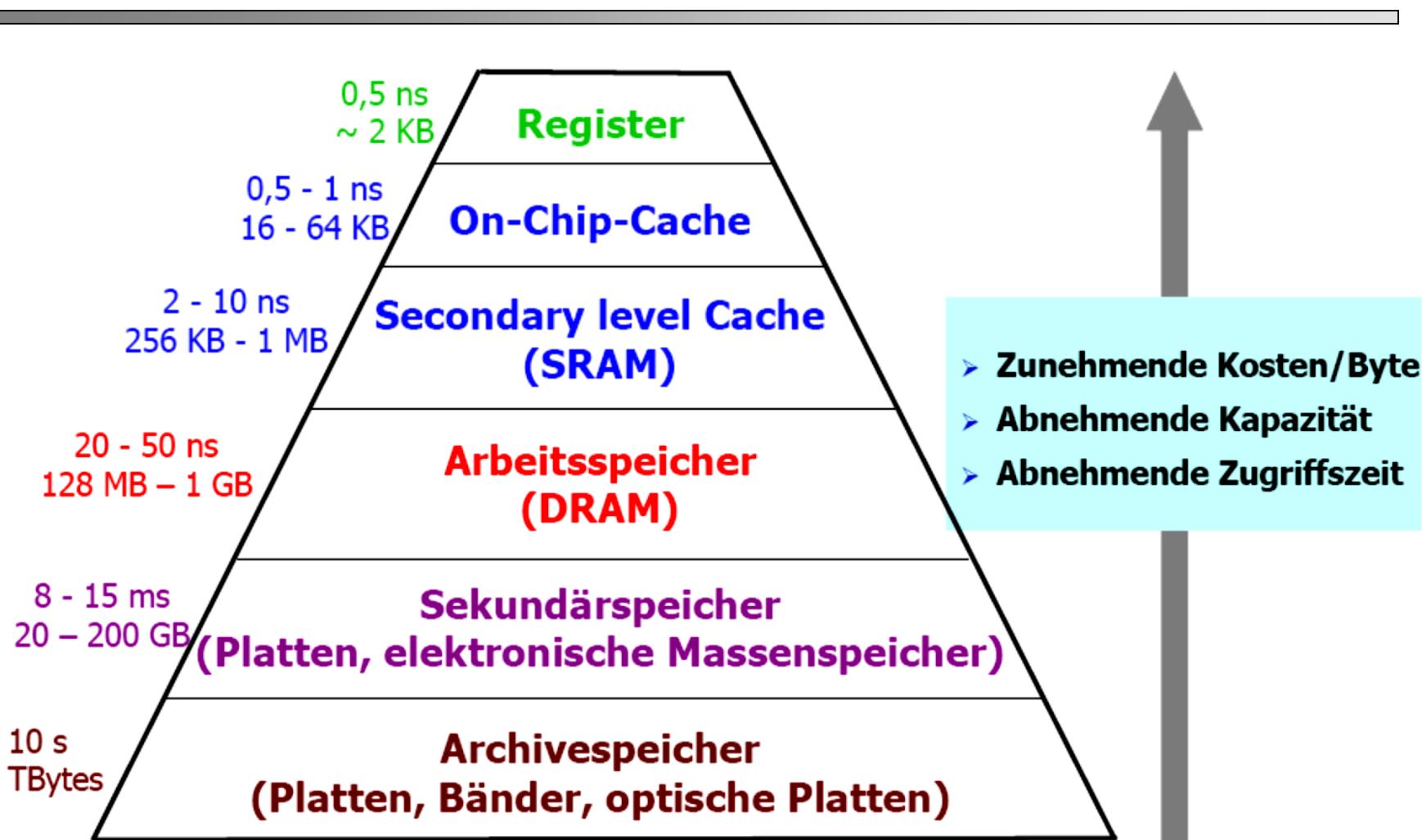
Speicherhierarchie

Wirkung: wie ein großer und schneller Speicher, wenn

- ❑ Lokalitätsverhalten der Programmverarbeitung,
- ❑ Umlagerung der Information rechtzeitig
(Umlagerungsstrategien),
- ❑ Inhomogenität des Speichersystems für Benutzer
nicht sichtbar ist (Virtueller Speicher)

Leistungsfähigkeit der Hierarchie ist bestimmt durch die Eigenschaften der Speichertechnologien (Zugriffsart, Zugriffszeiten, ...), Adressierung der Speicherplätze und Organisation des Betriebs

Speicherhierarchie



Das Problem der Zykluszeit

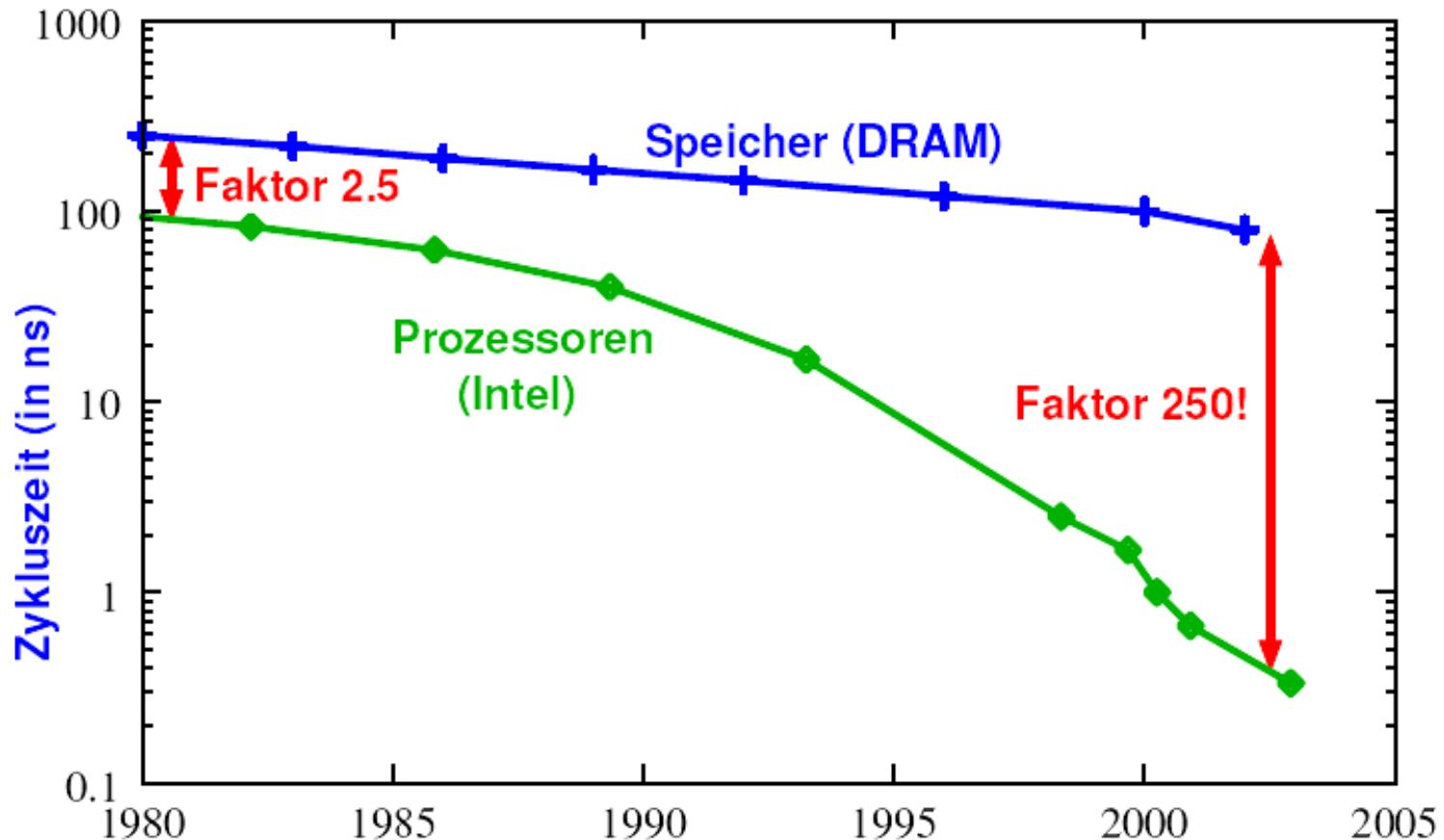
Problem:

die Buszykluszeit moderner Prozessoren ist erheblich kürzer als die Zykluszeit preiswerter, großer DRAM-Bausteine

- ▶ dies zwingt zum Einfügen von Wartezyklen. SRAM-Bausteine hingegen, die ohne Wartezyklen betrieben werden können, sind jedoch klein, teuer und leistungsintensiv.
- ▶ nur kleine Speicher können so aufgebaut werden.

Das Problem der Zykluszeit

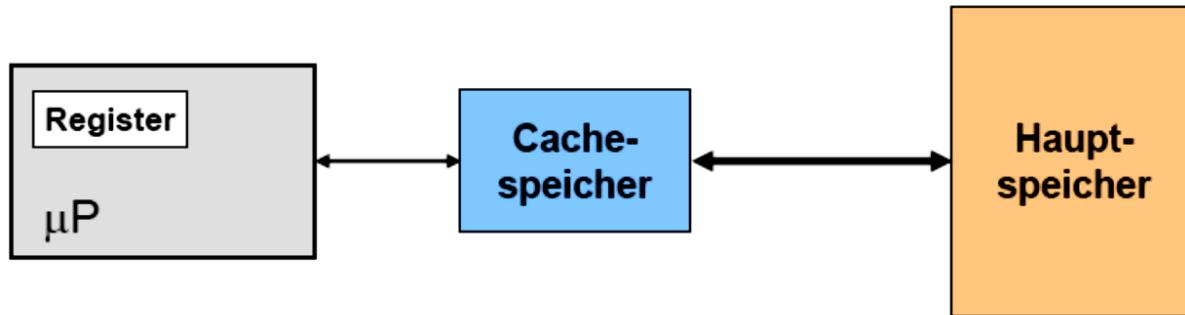
Motivation: Prozessor- versus Speichergeschwindigkeit



Cache-Speicher

Lösung des Problems:

zwischen den Prozessor und den relativ langsamen, aber billigen Hauptspeicher aus DRAM-Bausteinen legt man einen kleinen, schnellen Speicher aus SRAM-Bausteinen, den sogenannten **Cache-Speicher**.



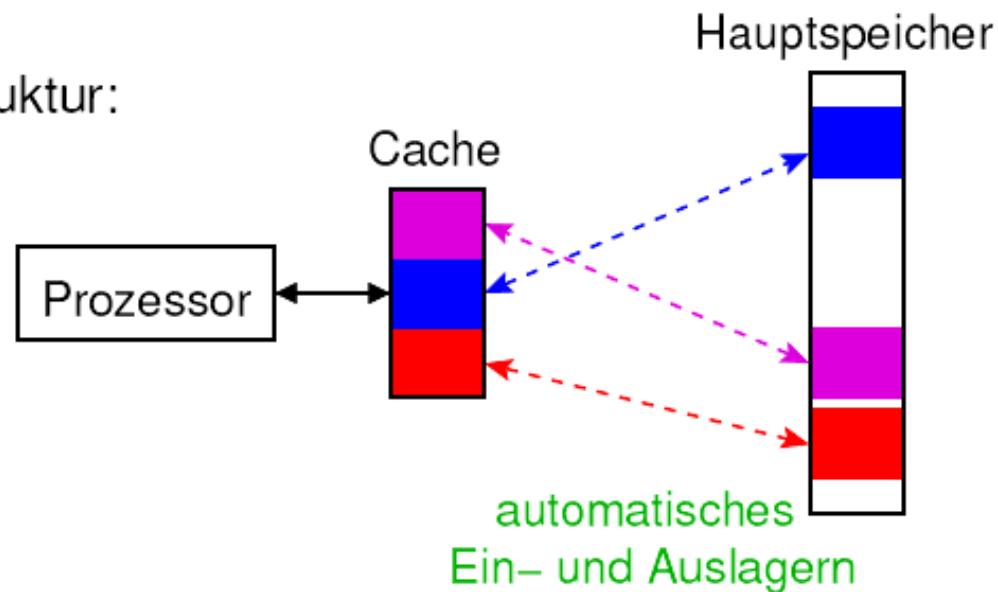
Auf den Cache-Speicher soll der Prozessor fast so schnell wie auf seine Register zugreifen können.

Grundidee des Cache

- Häufig benötigte Daten in kleinem, schnellem Pufferspeicher halten
- Geringe Kapazität im Vergleich zum Hauptspeicher
- Cache enthält immer Kopie eines Ausschnitts des Hauptspeichers
 - Alle Daten im Cache sind auch im Hauptspeicher vorhanden
 - Ausschnitt kann mit der Zeit wechseln

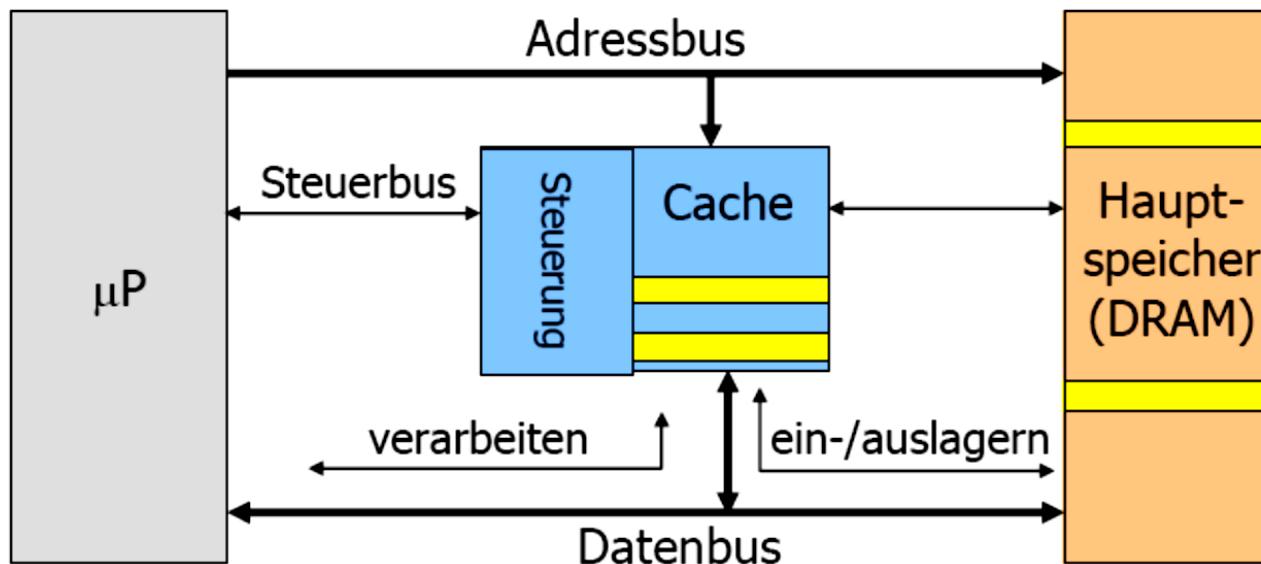
Grundidee ...

→ Logische Struktur:



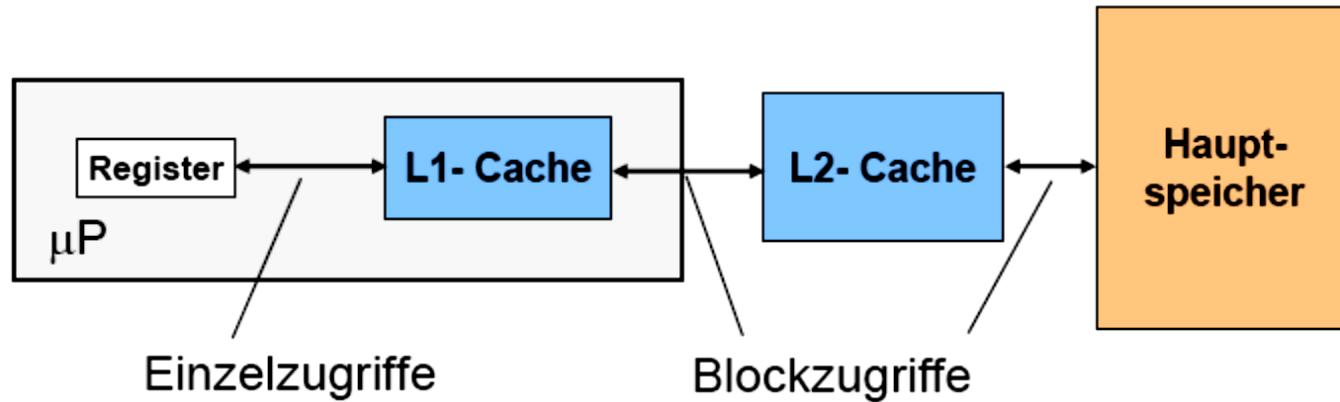
Cache-Speicher

Unter einem **CPU-Cache-Speicher** versteht man einen kleinen, schnellen Pufferspeicher, in dem Kopien derjenigen Teile des Hauptsspeichers bereitgehalten werden, auf die aller Wahrscheinlichkeit nach von der CPU als nächstes zugegriffen wird.



Cache-Speicher

- **On-Chip-Cache:** integriert auf dem Prozessorchip
 - Sehr kurze Zugriffszeiten (wie die der prozessorinternen Register)
 - Aus technologischen Gründen begrenzte Kapazität



Cache - Speicher

Ein CPU-Cache-Speicher bezieht seine Effizienz im wesentlichen aus der **Lokalitätseigenschaft** von Programmen (*locality of reference*), d.h. es werden bestimmte Speicherzellen bevorzugt und wiederholt angesprochen (z.B. Programmschleifen)

□ **Zeitliche Lokalität:**

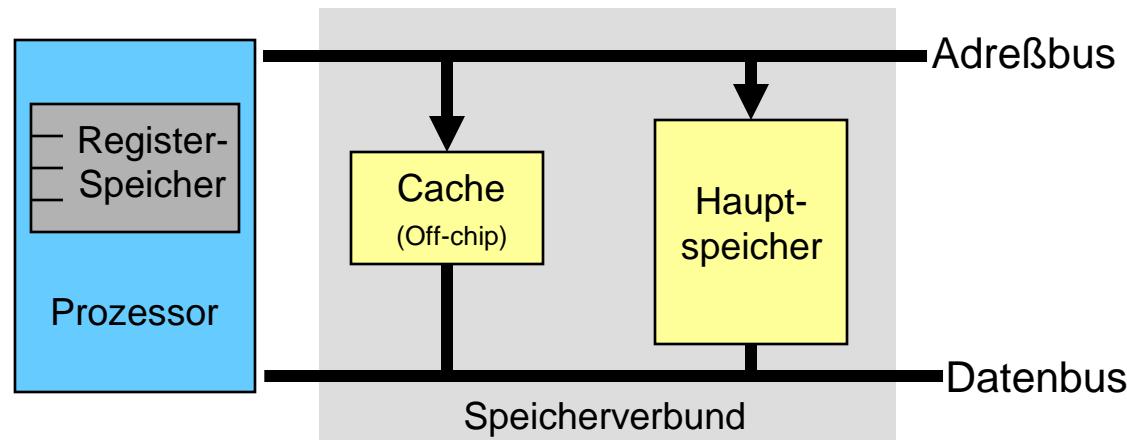
Die Information, die in naher Zukunft angesprochen wird, ist mit großer Wahrscheinlichkeit schon früher einmal angesprochen worden (Schleifen).

□ **Örtliche Lokalität:**

Ein zukunftiger Zugriff wird mit großer Wahrscheinlichkeit in der Nähe des bisherigen Zugriffs liegen.

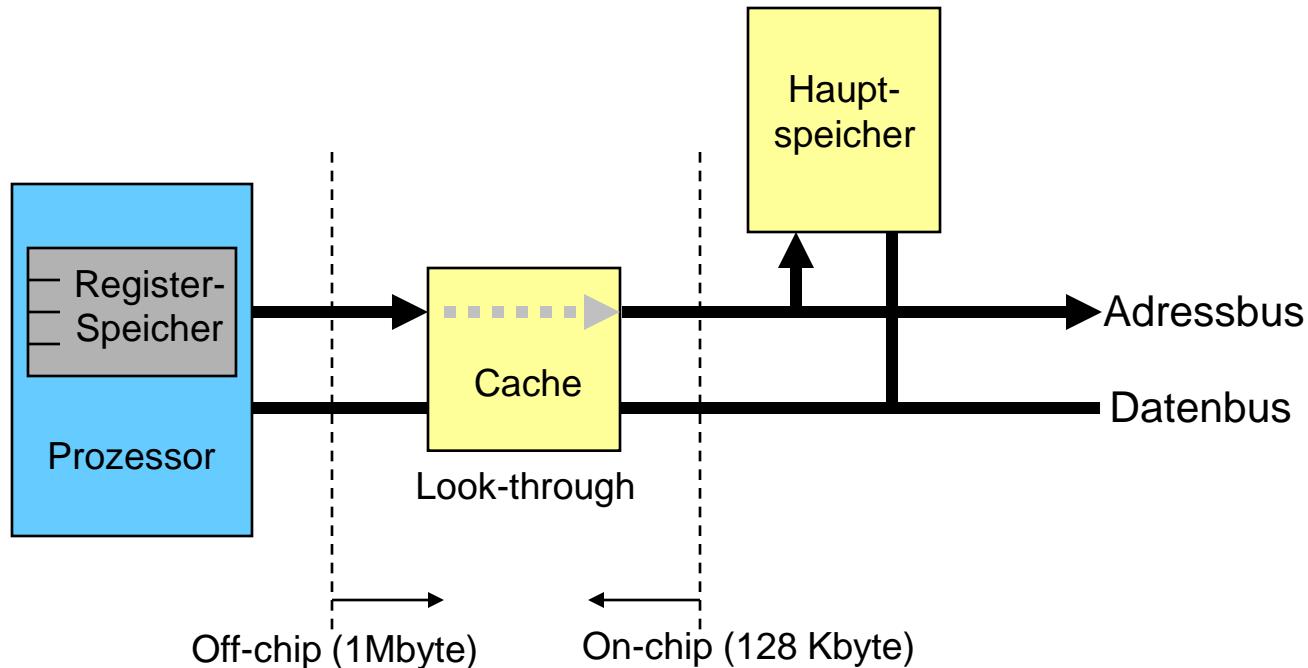
Systemstruktur: Look - Aside - Cache

- Wird „parallel“ am Systembus betrieben. Zugriffsanforderung des Prozessors geht gleichzeitig an den Cache und an den Hauptspeicher. Kann der Cache die Anforderungen befriedigen, wird der Hauptspeicherzugriff gestoppt, wenn nicht, ist der Speicherzugriff fasst ohne Verzögerung durch den Cache möglich.



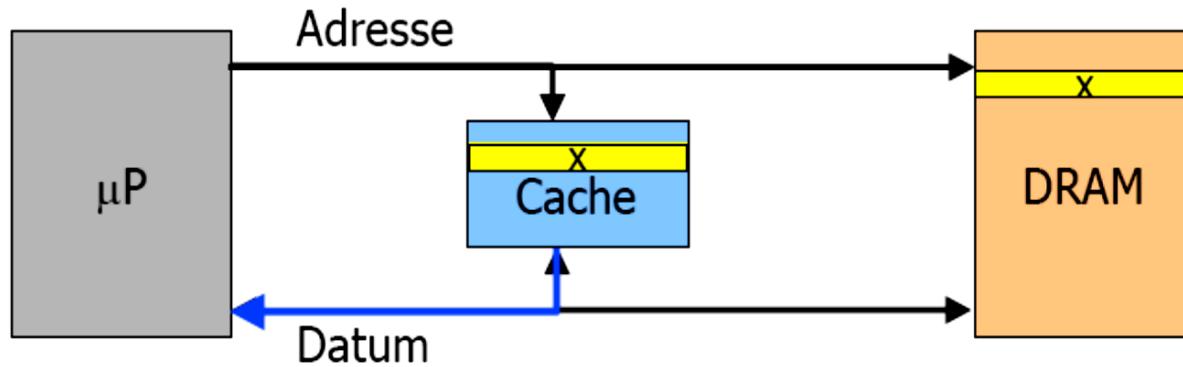
Systemstruktur: Look - Through - Cache

- Typisch für Multiprozessorsysteme
 - mehrere Prozessor-Cache-Einheiten greifen auf einen gemeinsamen Speicher zu
 - Zugriffsanforderungen des Prozessors, werden vom Cache-Controller gesteuert, um die Speicherbuszugriffe zu minimieren.



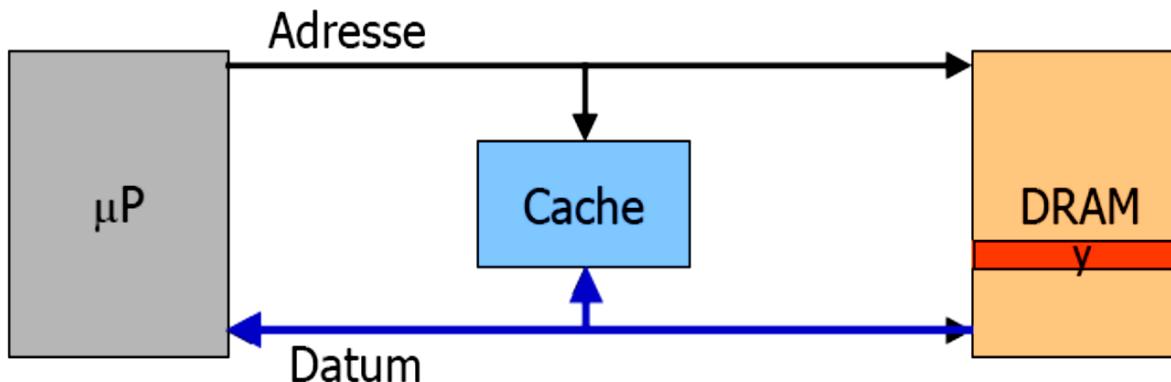
Cache - Speicher: Lesezugriffe

Hit



Treffer (Cache Hit)
das Datum kann ohne Wartezyklen dem Cache entnommen werden.

Miss



kein Treffer (Cache Miss)
das Datum wird mit Wartezyklen aus dem Arbeitsspeicher gelesen und gleichzeitig in den Cache eingefügt.

Cache - Speicher: Schreibzugriffe

Schreibzugriffe:

Liegt beim Schreiben ein Cache-Miss vor, wird das Datum in den Arbeitsspeicher geschrieben.

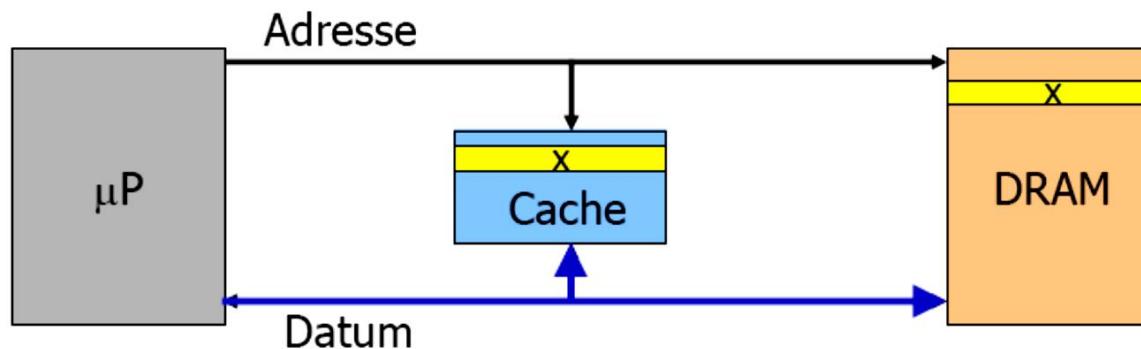
Liegt beim Schreiben jedoch ein Cache-Hit vor, d.h. ein im Cache stehendes Datum wird durch den Prozessor verändert, so existieren verschiedene Organisationsformen:

Schreibzugriffe: Bei Cache - Hit

□ Durchschreibverfahren:

(*write through policy*)

Ein Datum wird von der CPU immer gleichzeitig in den Cache- und in den Arbeitsspeicher geschrieben.



- Beispiel:
CPI = 1 (ohne write through)
- 10% Store-Befehle, das Schreiben eines Worts im Speicher dauert 100 Takte
- Dann ist die „effektive CPI“: $1 + 0,1 \times 100 = 11$

Schreibzugriffe: Bei Cache - Hit

□ Durchschreibverfahren:

(*write through policy*)

Ein Datum wird von der CPU immer gleichzeitig in den Cache- und in den Arbeitsspeicher geschrieben.

Vorteil:

garantierte Konsistenz zwischen Cache- und Arbeitsspeicher.

Nachteil:

Schreibzugriffe benötigen immer die langsame Zykluszeit des Hauptspeichers und belasten den Systembus.

(Variante beim Cache-Write-Miss: *Write-through with write-allocate*)

(In Hauptspeicher schreiben und dann den Block in den Cache laden)

Schreibzugriffe: bei Cache - Hit

□ Gepuffertes Durchschreibverfahren: *(buffered write through policy)*

Variante des Durchschreibverfahrens

Zur Milderung des Nachteils beim Durchschreibverfahren wird ein kleiner Schreib-Puffer verwendet, der die zu schreibenden Daten temporär aufnimmt.

*

Diese Daten werden dann automatisch vom Cache-Controller in den Hauptspeicher übertragen, während der Prozessor parallel dazu mit weiteren Operationen fortfährt.

* CPU kann direkt weiterarbeiten, muss aber stoppen, falls der Schreibpuffer voll ist.

Schreibzugriffe: bei Cache - Hit

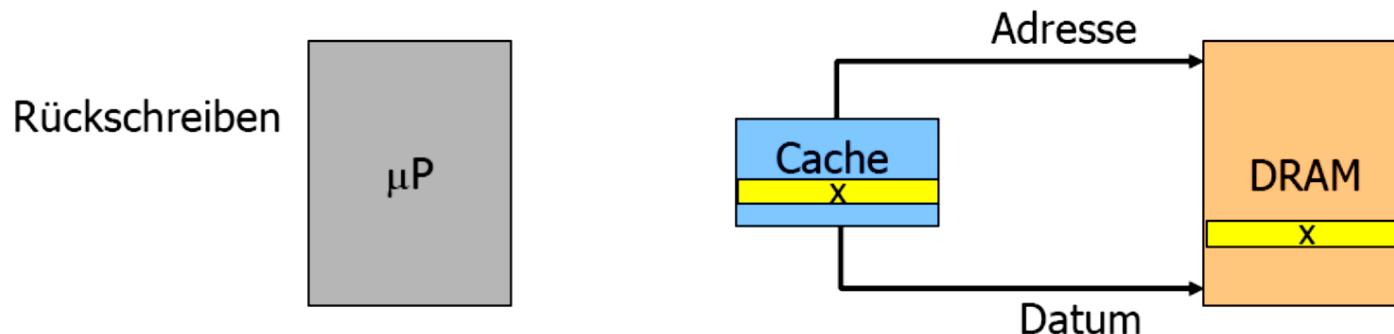
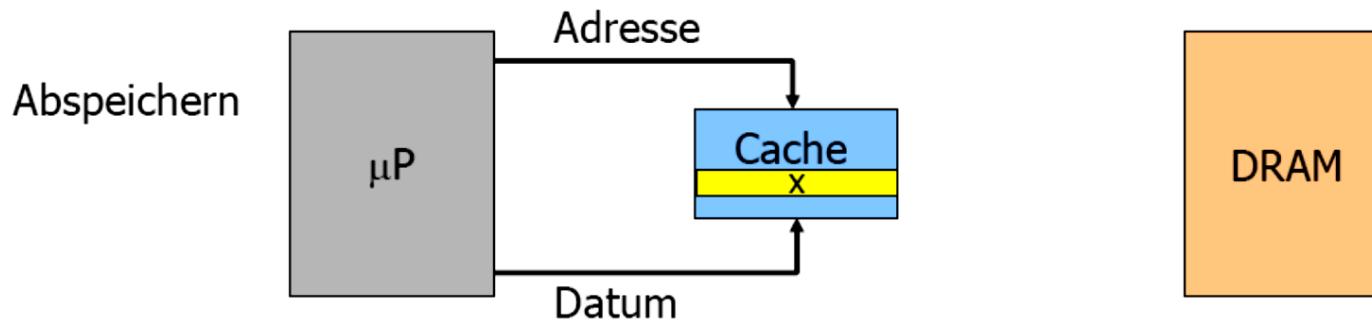
- **Rückschreib-Verfahren:**
(write back policy)

Ein Datum wird von der CPU nur in den Cachespeicher geschrieben und durch ein spezielles Bit (*altered bit, modified bit, dirty bit*) gekennzeichnet.

Der Arbeitsspeicher wird nur geändert, wenn ein so gekennzeichnetes Datum aus dem Cache verdrängt wird

Cache - Speicher: Schreibzugriffe

Prinzip des Rückschreibverfahrens:



Vorteil:

auch Schreibzugriffe können mit der schnellen Cache-Zykluszeit abgewickelt werden

Nachteil:

Konsistenzprobleme zwischen Cache- und Arbeitsspeicher.

Beispiele für Konsistenzprobleme

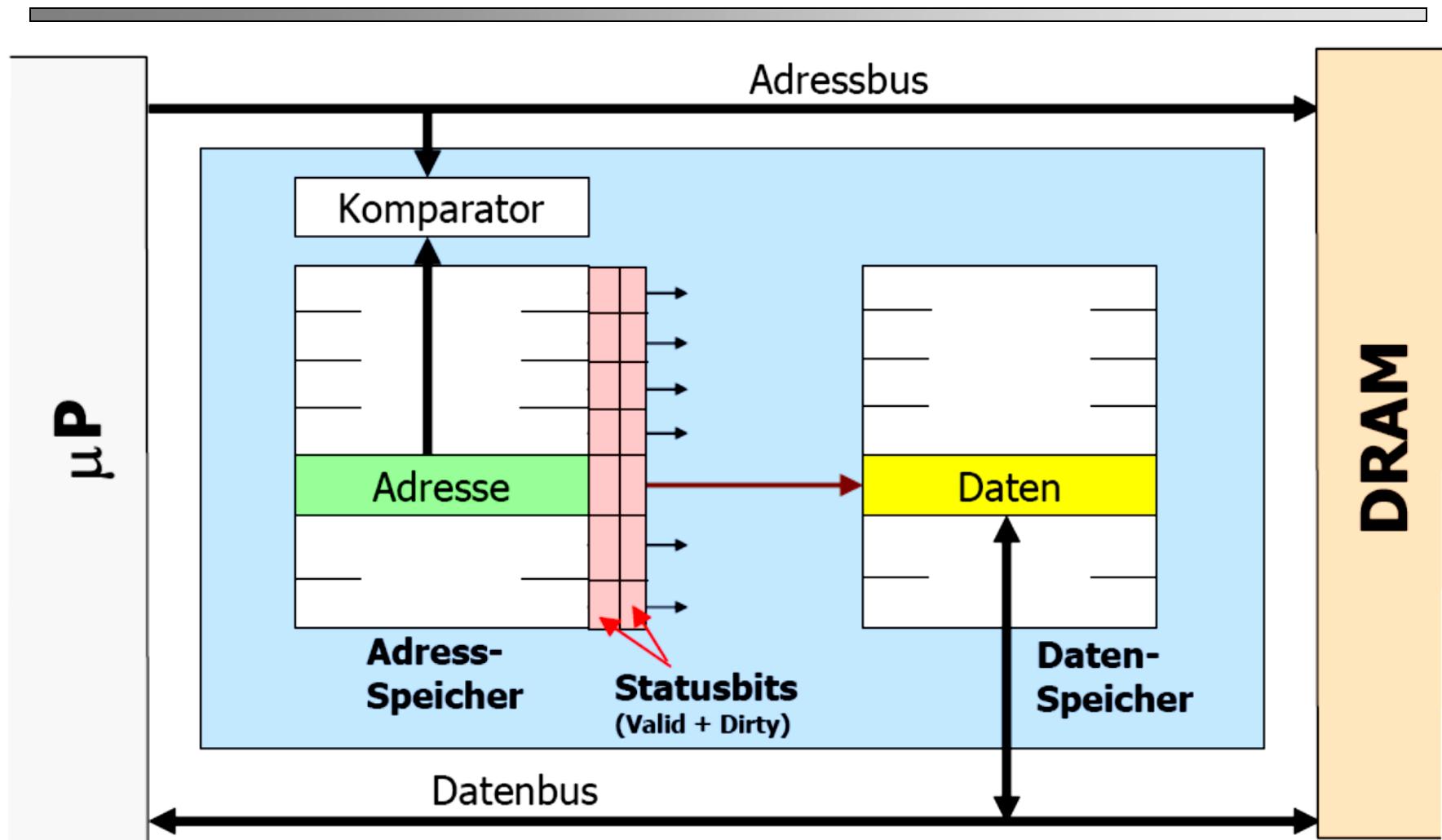
- Andere Systemkomponenten, z.B. DMA-Controller, finden nun unter Umständen „veraltete Daten“ im Arbeitsspeicher vor, die von der CPU längst geändert, jedoch noch nicht in den Arbeitsspeicher übertragen wurden.
- Ebenfalls können andere Systemkomponenten Daten im Hauptspeicher ändern, während die CPU noch mit den alten Daten im Cachespeicher arbeitet.
 - ➡ aufwendige Verfahren bei der Cache-Steuerung zur Verhinderung solcher Inkonsistenzen sind erforderlich (z.B. muss die Cache-Steuerung über jede Datenänderung im Hauptspeicher informiert werden).

Assoziative Speicherung von Daten

- Konventioneller Speicher
 - Adressen weisen zu den Informationen
- Assoziativer Speicher
 - gespeicherte Information selbst enthält den Suchbegriff

Kriterienreg.	Personalnr.	Name	Vorname	Geburtsdatum	Firmen-Eintritt
123 456	111 555	Mayer	Josef	1.4.67	1.1.89
	115 678	Adler	Ulrike	29.2.69	1.7.86
	123 456	Schmid	Karl	27.11.58	1.4.83
	124 963	Müller	Peter	24.12.72	1.5.92
	132 824	Pfleiderer	Inge	17.9.63	15.10.87

Erster Aufbau eines Cache - Speichers



Aufbau eines Cache - Speichers

Ein Cache-Speicher besteht aus zwei Speicher-Einheiten:

▫ **Datenspeicher:**

enthält die im Cache abgelegten Daten

▫ **Adressspeicher:**

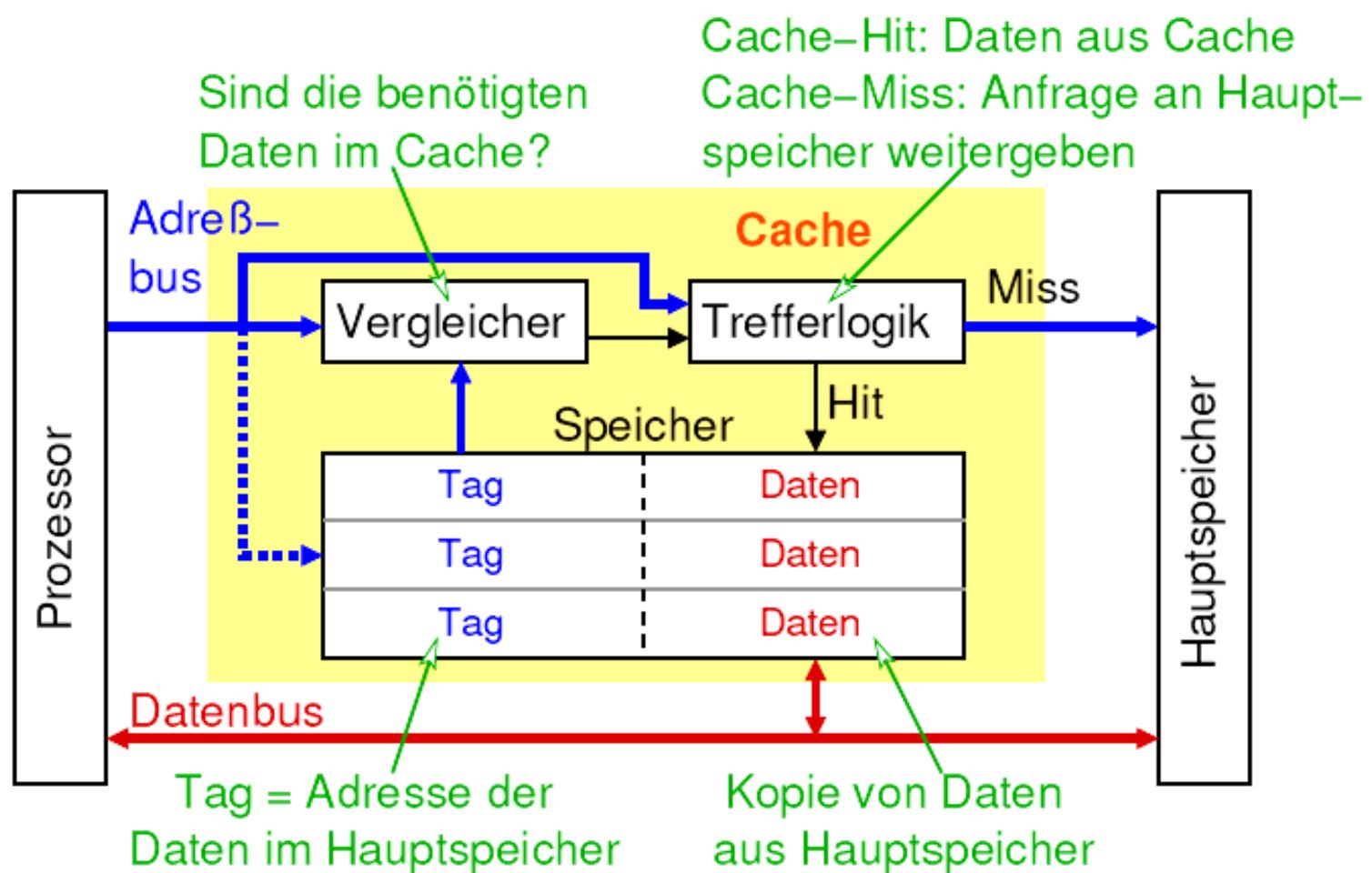
enthält die Adressen dieser Daten im Arbeitsspeicher

Ein Komparator ermittelt, ob das zu einer auf dem Adressbus liegende Adresse gehörende Datum auch im Cache abgelegt worden ist

► Adressvergleich mit den Adressen im Adressspeicher

Dieser Adressvergleich muss *sehr schnell* gehen (*möglichst in einem Taktzyklus*), da sonst der Cachespeicher effektiv langsamer wäre als der Arbeitsspeicher.

Prinzipieller Aufbau eines Cachespeichers



Aufbau Cache-Speicher

- Jede Cache-Zeile enthält ein (Adress, Daten)-Paar und Statusbits.
- Ein (*Daten*)-**Block** ist eine zusammengehörende Reihe von Speicherplätzen.
- Dazugehörig wird ein Adressetikett (Index, Cache-Tag) im Adress-Speicher ablegt.
- Das Cache-Tag enthält die Adresse des aktuellen Blocks im Hauptspeicher.
- Die Statusbits sagen aus, ob die Daten im Cache gültig sind.

Arbeitsweise des Cache - Speichers

- ❑ Cache-Steuerung prüft bei Zugriffen des µP, ob
 - Der zur Speicheradresse gehörende Hauptspeicherinhalt als Kopie im Cache steht (**Bedingung 1**) und
 - Dieser Cache-Eintrag durch das Gültigkeits-Bit (Valid-Bit) als gültig gekennzeichnet ist (**Bedingung 2**)
- ❑ Prüfung führt zu einem Cache-Treffer oder zu einem Fehlzugriff.
- ❑ Cache-Fehlzugriff (Cache-miss): eine der beiden Bedingungen ist nicht erfüllt.

Arbeitsweise des Cache - Speichers

- ❑ Cache-Fehlzugriff (Cache-miss): eine der beiden Bedingungen ist nicht erfüllt.

Lesezugriffe (read miss)

- Lesen des Datums aus dem Hauptspeicher und Laden des Cache-Speichers
- Kennzeichnen der Cache-Eintrag als gültig (V-Bit setzen)
- Speichern der Adressinformation im Adress-Speicher des Cache-Speichers

Arbeitsweise des Cache - Speichers

- ❑ Cache-Fehlzugriff (Cache-miss): eine der beiden Bedingungen ist nicht erfüllt.

Schreibzugriffe (write miss)

Aktualisierungsstrategie bestimmt, ob

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

Aktualisierungsstrategien - Zusammenfassung

↗ Aktualisierungsstrategie für Caches mit je einem Valid- und einem Dirty-Bit

Cache-Zugriff	Write-Through No-Write Alloc.	Write-Through Write-Alloc.	Copy-Back
Read-Hit	Cache-Datum --> CPU	Cache-Datum --> CPU	Cache-Datum --> CPU
Read-Miss	HS-Block, Tag --> Cache HS-Datum --> CPU 1 --> V	HS-Block, Tag --> Cache HS-Datum --> CPU 1 --> V	Cache-Zeile --> HS HS-Block, Tag --> Cache HS-Datum --> CPU 1 --> V, 0 --> D
Write-Hit	CPU-Datum --> Cache, HS	CPU-Datum --> Cache, HS	CPU-Datum --> Cache 1 --> D
Write-Miss	CPU-Datum --> HS	HS-Block, Tag --> Cache, 1 --> V CPU-Datum --> Cache, HS	Cache-Zeile --> HS HS-Block, Tag --> Cache 1 --> V CPU-Datum --> Cache 1 --> D

Verdrängungsstrategie

- **Verdrängungsstrategie** gibt an, welcher Teil des Cachespeichers nach einem Cache-Miss durch eine neu geladene Speicherportion überschrieben wird.
- **Verdrängungsstrategie** nur bei voll- oder n-fach satzassoziativer Cachespeicherorganisation angewandt
- meist die sehr einfache Strategie gewählt, die am längsten nicht benutzte Speicherportion zu ersetzen (**LRU-Strategie**, *Least Recently Used*).

Trefferrate und Zugriffszeit

Die **Hit-Rate** bezeichnet die Trefferquote im Cache:

$$\text{Hit-Rate} = \text{Anzahl Treffer} / \text{Anzahl Zugriffe}$$

Die **mittlere Zugriffszeit** berechnet sich annähernd wie folgt:

$$t_{\text{Access}} = (\text{Hit-Rate}) * t_{\text{Hit}} + (1 - \text{Hit-Rate}) * t_{\text{Miss}}$$

mit

t_{Hit} : Zugriffszeit des Caches

t_{Miss} : Zugriffszeit ohne den Cache

Trefferrate und Zugriffszeit

- Trefferrate (*hit rate*) und Fehlzugriffsrate (*miss rate*)
 - Prozentsatz der Treffer beim Cachezugriff bzw. Prozentsatz der Cache-Fehlzugriffe bezogen auf alle Cachezugriffe.
- Cache-Zugriffszeit t_{Hit}
 - Anzahl der Takte (oder Zeit), die benötigt wird, um ein Speicherwort im Cache zu identifizieren, die Verfügbarkeit und Gültigkeit zu prüfen und das Speicherwort zur Verfügung zu stellen.
- Fehlzugriffszeit t_{Miss}
 - Zeit, die benötigt wird, um einen Cacheblock von einer tiefer gelegenen Hierarchiestufe in den Cache zu laden und das Speicherwort dem Prozessor zur Verfügung zu stellen.
- Effektive CPI = Basis-CPI + Speicherzugriffszyklen

1. Beispiel zur Trefferrate

- Arbeitsspeicher besteht aus DRAM-Bausteinen; Zykluszeit 200 ns.
- Cachespeicher besteht aus SRAM-Bausteinen; Zugriffszeit 50 ns
(In dieser Zeit sei die Hit-/Miss-Entscheidung enthalten).
- Zugriff des Prozessors:
 - Auf Cache: Ohne Wartezyklen
 - Auf Arbeitsspeicher: 3 Wartezyklen
- Trefferrate der Cachezugriffe liegt bei 80%.

- Wie groß ist die mittlere Zugriffszeit?
- Wie groß ist die durchschnittliche Anzahl der Wartezyklen?

2. Beispiel zur Trefferrate

- Ein Rechnersystem besitzt einen getrennten Befehls- und Datencache
 - Befehlscache: 2% miss rate
 - Datencache: 4% miss rate
 - Miss-Penalty: 100 Takte
 - Basis-CPI (bei 100% Hit rate): 2
 - Anteil der *load*- und *store*-Befehle: 36%.
-
- Mittlere Anzahl Memory-Miss-Cycles pro Instruktion?
 - Instruktions-Cache:
 - Daten-Cache:
 - Was ist der effektive CPI?
 - Effektiver CPI:
 - Ideale CPU bei 100% Hit Rate ist über zweieinhalb-mal schneller
 - Eine niedrige Hit-Rate vor allem im Befehls-Cache ist entscheidend für die Performance des Programms

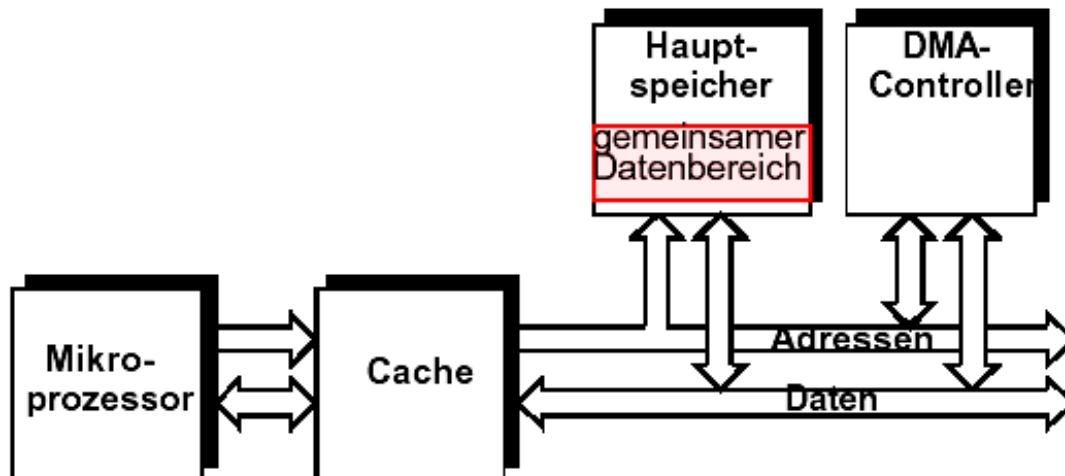
Performance von Cachespeichern

- Bei steigender CPU-Performance wird der Einfluss von Miss-Penalty signifikant.
 - Je niedriger der Basis-CPI desto größer wird der Anteil der Zeit, die in „Memory-Stalls“ verbraucht wird.
 - Höhere CPU-Taktfrequenz führt zu mehr Taktzyklen in Memory Stalls
 - Spekulative Programmausführung und Multi-Threading mindern die Auswirkungen von Misses, da während der Miss-Penalty andere Instruktionen ausgeführt werden können.
- Die Cache-Effizienz hat einen entscheidenden Einfluss auf die Gesamtperformance des Systems

Zusätzliche Master am Systembus

↗ Datenkonsistenz bei zusätzlichen Mastern ohne Cache

- ↗ Beispiel: Mikroprozessorsystem, das neben dem Prozessor mit Cache einen DMA-Controller als weiteren Master aufweist;
 - ↗ zusätzlicher Master hat wie Prozessor Zugriff auf Hauptspeicher
 - ↗ beide teilen sich **gemeinsamen Datenbereich (shared data)**



Cache - Konsistenzproblem

☞ Konsistenzproblem:

- ☞ beim Write-Through-Verfahren:
 - ☞ DMA-Controller beschreibt eine Speicherzelle, deren Inhalt im Cache als gültig eingetragen war, der Prozessor führt danach einen Lesezugriff mit der Adresse dieser Speicherzelle durch:
 - ☞ --> Prozessor liest veraltetes Datum
- ☞ beim Copy-Back-Verfahren:
 - ☞ der Prozessor führt Schreibzugriff mit der Adresse aus dem gemeinsamen Bereich aus und aktualisiert nur Cache; der DMA-Controller liest anschließend die Speicherzelle mit dieser Adresse:
 - ☞ --> der DMA-Controller liest veraltetes Datum (im Hauptspeicher);

Lösung des Konsistenzproblems

☞ Non-Cachable Data

- ☞ der vom Prozessor und dem zusätzlichen Master gemeinsam benutzte Speicherbereich wird von der Speicherung im Cache ausgeschlossen;
- ☞ Speicherverwaltung:
- ☞ Der Adreßbereich wird in seinem für die Speicherverwaltungseinheit bereitgestelltem Deskriptor als „*non-cacheable*“ gekennzeichnet.
- ☞ Die Cache-Steuerung wird bei Zugriffen auf den so gekennzeichneten Bereich nicht aktiv.
- ☞ Es werden auch die für Schnittstellen und Controller reservierten Adreßbereiche als „*non-cacheable*“ gekennzeichnet, um den direkten Zugriff auf deren Daten-; Steuer- und Statusregister zu gewährleisten.

Lösung des Konsistenzproblems

- ☒ **Cache-Clear, Cache-Flush**
- ☒ Die Zugriffe von Prozessor und DMA-Controller auf den gemeinsamen Datenbereich werden von zwei unterschiedlichen Tasks ausgeführt;
- ☒ In diesem Fall kann die Task, die den DMA-Vorgang auslöst, dafür sorgen, daß der Cache gelöscht wird, d.h nachfolgende Prozessorzugriffe führen zu einem Neuladen des Cache;
 - ☒ Write-Through: Cache-Clear:
 - ☒ Die Cache-Einträge werden auf ungültig gesetzt.
 - ☒ Copy-Back: Cache-Flush:
 - ☒ Alle mit „dirty“ gekennzeichneten Einträge im Cache werden in den Hauptspeicher zurückgeschrieben, danach werden Cache-Einträge auf ungültig gesetzt.

Lösung des Konsistenzproblems

↗ Bus-Snooping

- ↗ Die Cache-Steuerung beobachtet den Bus hinsichtlich der Speicherzugriffe anderer Master:
 - ↗ Write-Through
 - ↗ bei Write-Hit des anderen Masters wird der entsprechende Cache-Eintrag ungültig gesetzt.
 - ↗ Copy-Back
 - ↗ Der Cache-Eintrag wird bei Write-Hit entweder als ungültig gekennzeichnet (Write Invalidate) oder er wird aktualisiert und als „dirty“ gekennzeichnet (Write Update);

Strukturen von Cache - Speichern

Cache - Strukturen

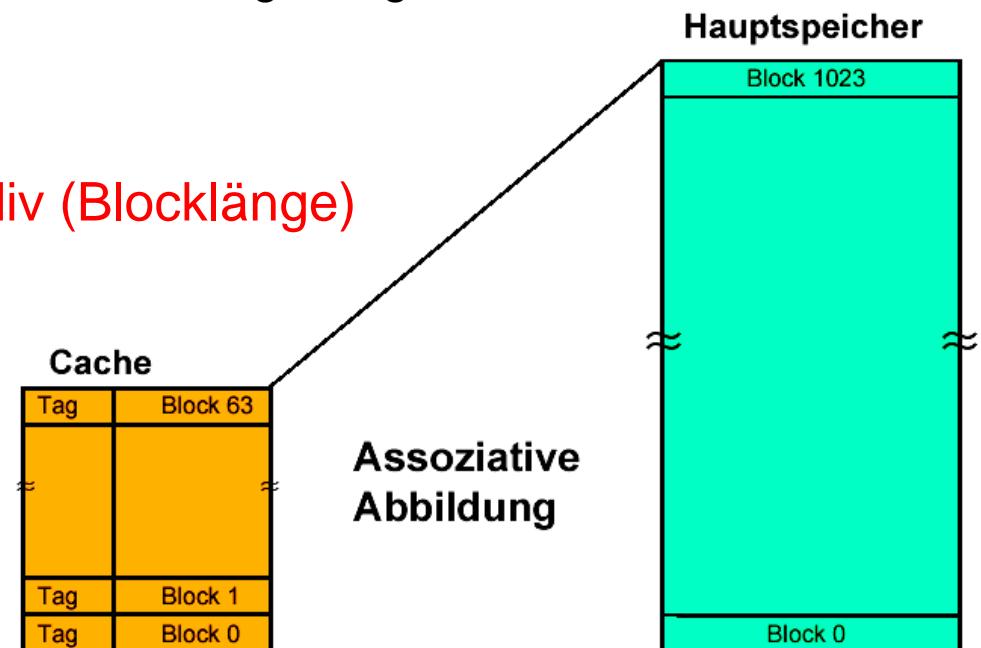
Anhand der für den Adressvergleich verwendeten Technik lassen sich verschiedene Cache-Typen unterscheiden:

- **Voll-Assoziativer Cache**
- **Direct Mapped Cache**
- **n-Way Set Associative Cache**

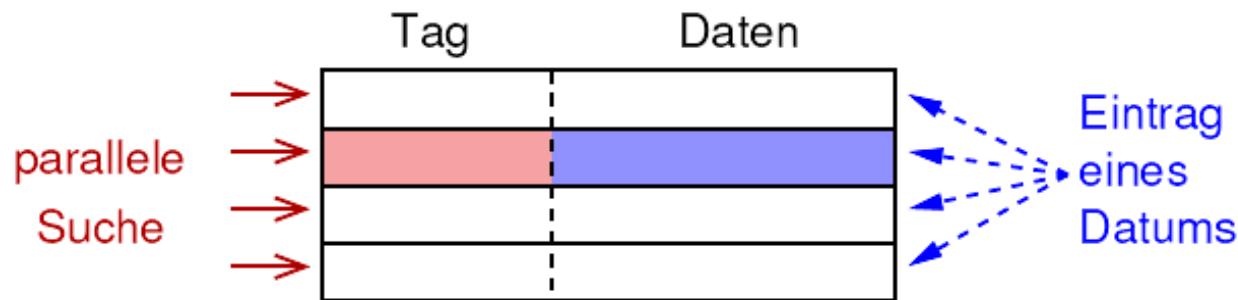
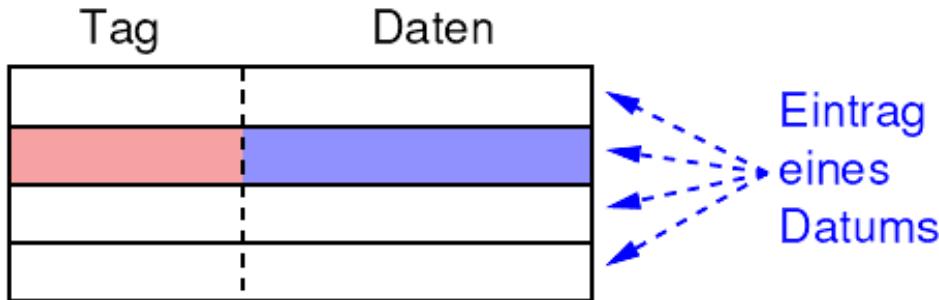
Voll - assoziativer Cache (VA)

- Datenblock kann prinzipiell in jede Cacheline eingetragen werden
- Größtmögliche Flexibilität (und damit Trefferrate)
- Aufwand für Tag-Vergleich sehr hoch
 - Adresse wird gleichzeitig mit allen Tags verglichen

Block-Nr.:= (HS-Adresse) div (Blocklänge)



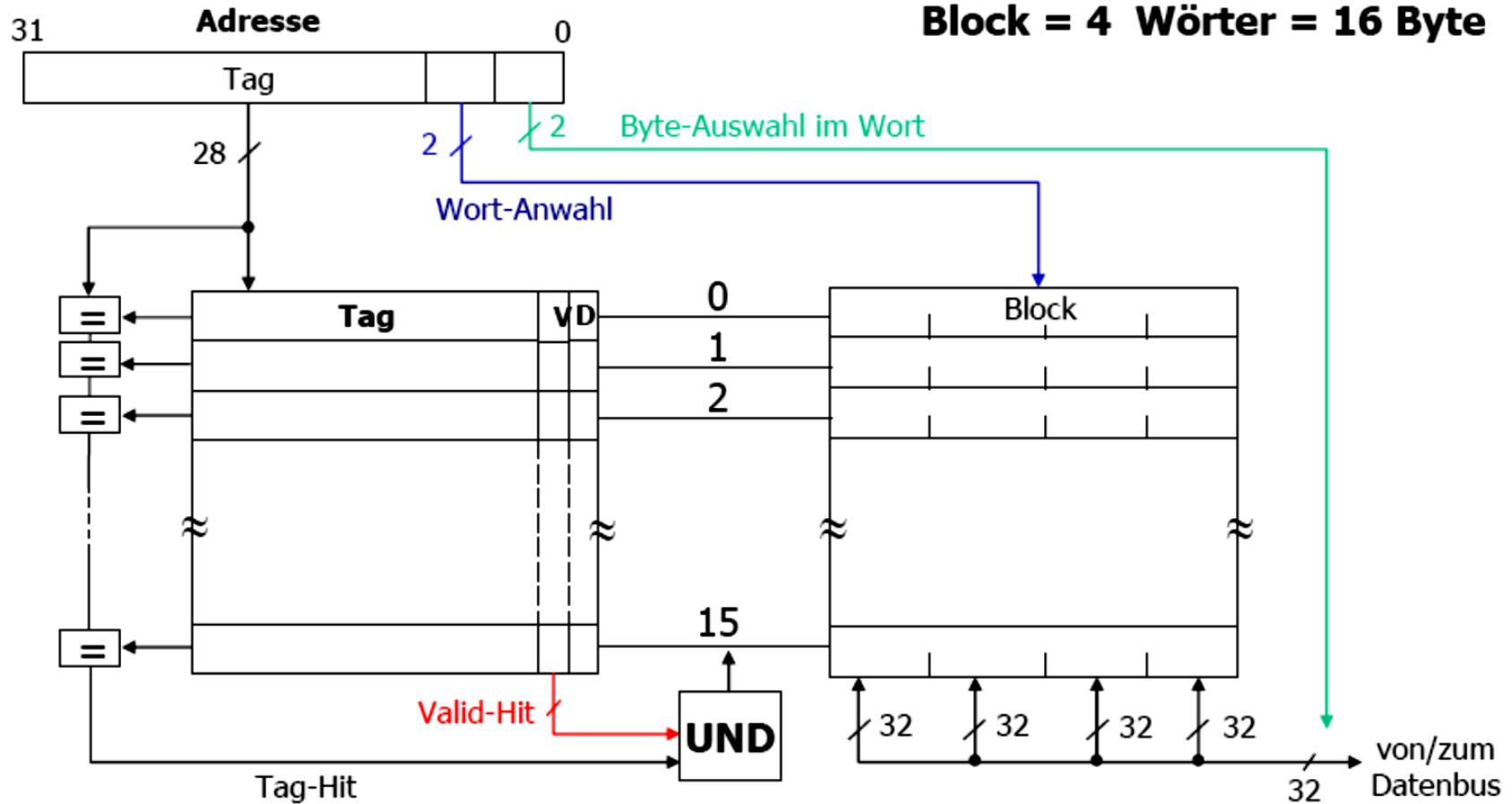
Platzierung und Ersetzungsstrategie (VA)



VA-Cache: Hardwarestruktur

Kapazität: 256 Byte

Block = 4 Wörter = 16 Byte



Voll - assoziativer Cache

Vollparalleler Vergleich aller Adressen im Adressspeicher
in einem einzigen Taktzyklus

Vorteil:

- ein Datum kann an beliebiger Stelle im Cache abgelegt werden
- Optimale Cache-Ausnutzung, völlig freie Wahl der Strategie bei Verdrängungen

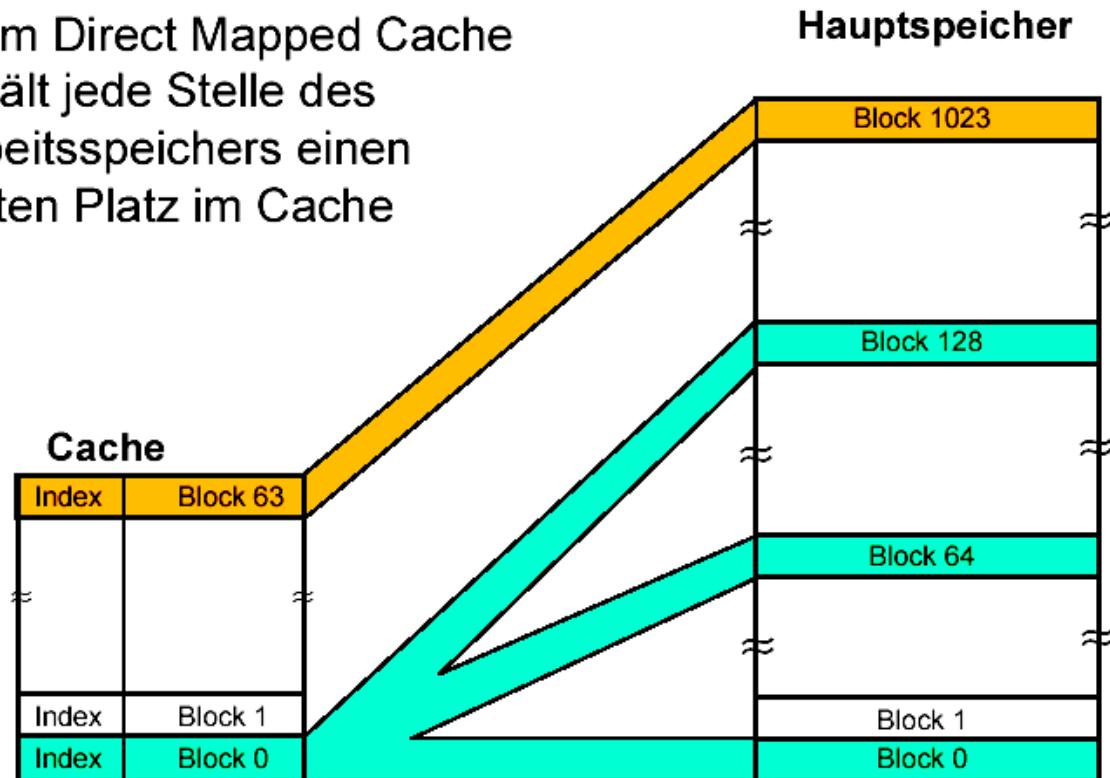
Nachteil:

- Hoher Hardwareaufwand (für jede Cache-Zeile ein Vergleicher)
 - ➡ nur für sehr kleine Cachespeicher realisierbar
- Die große Flexibilität der Abbildungsvorschrift erfordert eine weitere Hardware, welche die Ersetzungsstrategie (welcher Block soll überschrieben werden, wenn der Cache voll ist) realisiert.

Direct - Mapped Cache (DM)

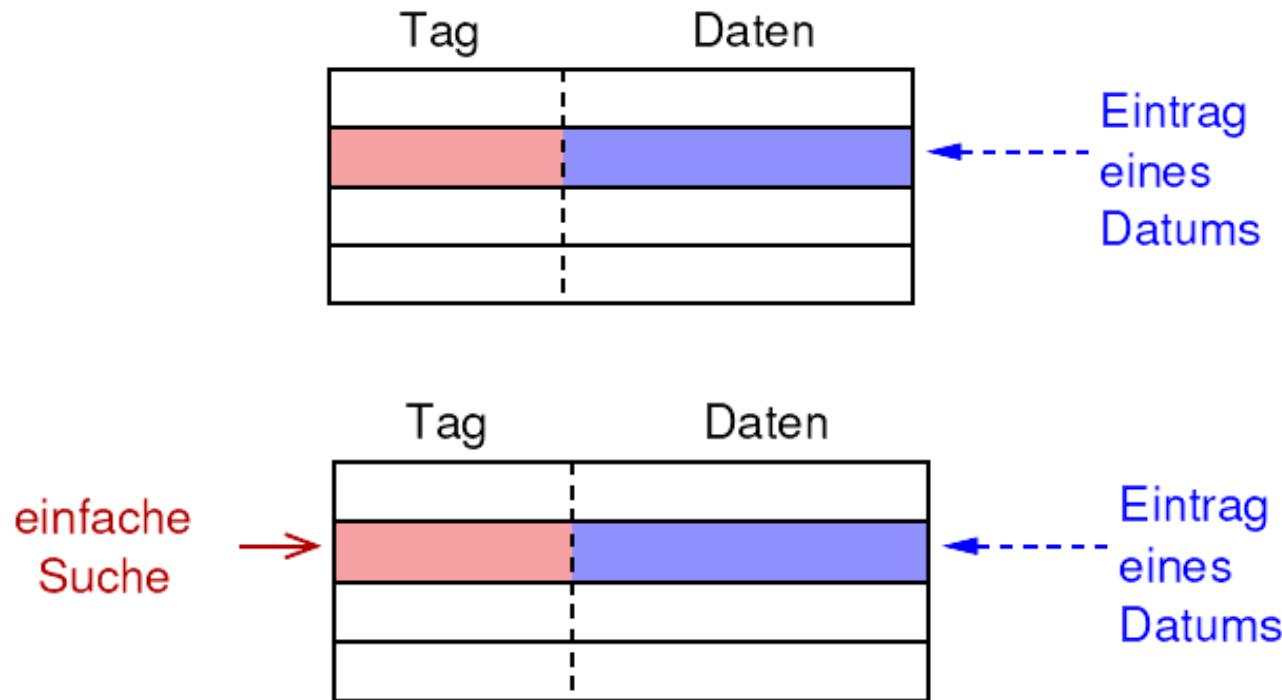
- Datenblock ergibt sich eindeutig aus der Hauptspeicheradresse der Cachezeile

Beim Direct Mapped Cache erhält jede Stelle des Arbeitsspeichers einen festen Platz im Cache

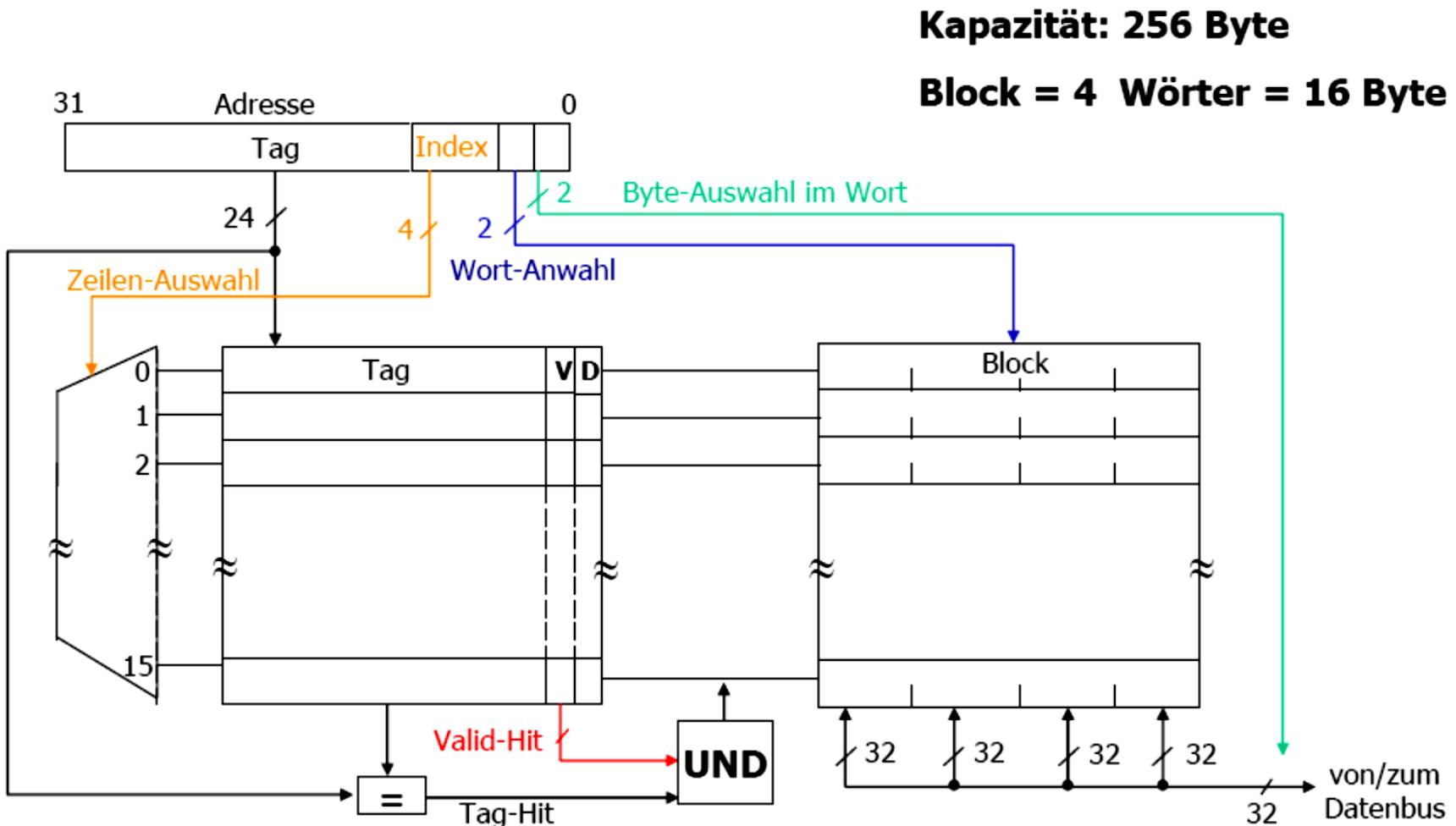


$$\text{Zeilen-Nr.} := (\text{Block-Nr.}) \bmod (\text{Zeilenzahl})$$

Platzierung und Ersetzungsstrategie (DM)



Direct - Mapped Cache (DM)

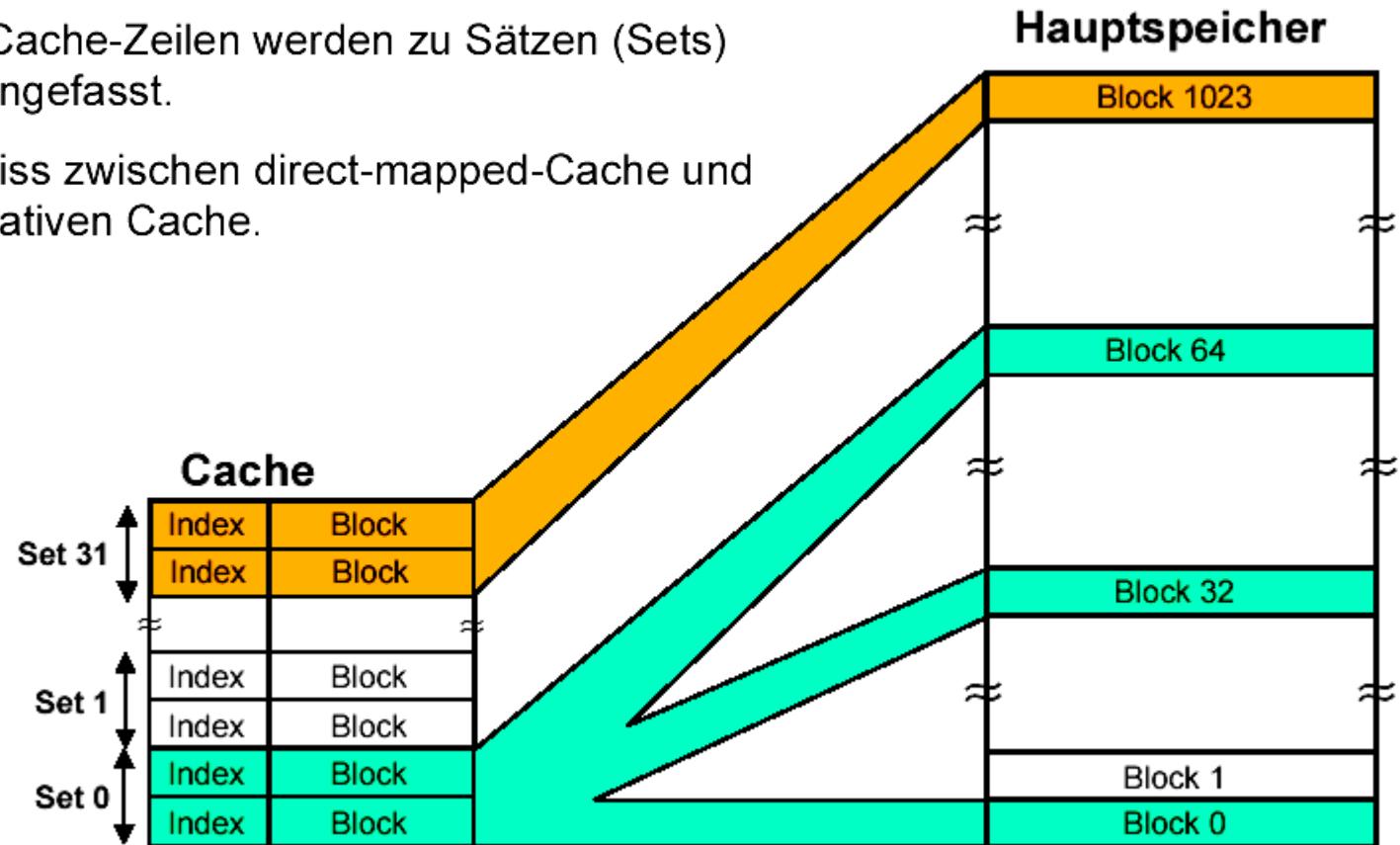


Direct - Mapped Cache (DM)

- ❑ Einfache Hardware-Realisierung (nur ein Vergleicher und ein Tag-Speicher)
- ❑ Der Zugriff erfolgt schnell, weil das Tag-Feld parallel mit dem zugehörigen Block gelesen werden kann
- ❑ Es ist kein Verdrängungsalgorithmus erforderlich, weil die direkte Zuordnung keine Alternativen zulässt
- ❑ Auch wenn an anderer Stelle im Cache noch Platz ist, erfolgt wegen der direkten Zuordnung eine Ersetzung
- ❑ Bei einem abwechselnden Zugriff auf Speicherblöcke, deren Adressen den gleichen Index-Teil haben, erfolgt laufendes Überschreiben des gerade geladenen Blocks. Es kommt zum "Flattern" (trashing)

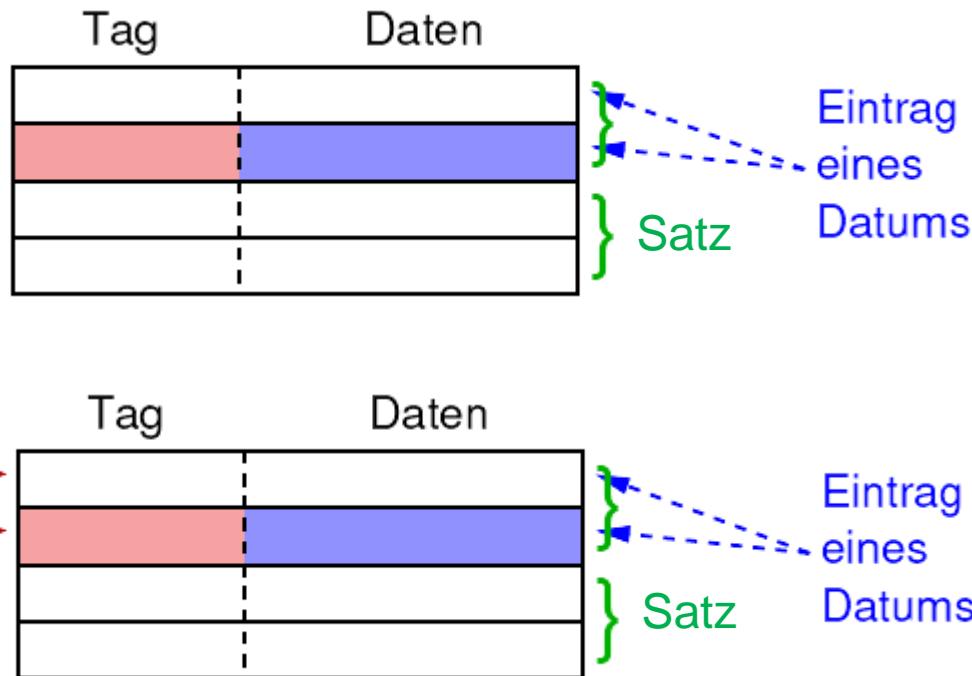
n - way - set - associative Cache

- Mehrere Cache-Zeilen werden zu Sätzen (Sets) zusammengefasst.
- Kompromiss zwischen direct-mapped-Cache und vollassoziativen Cache.

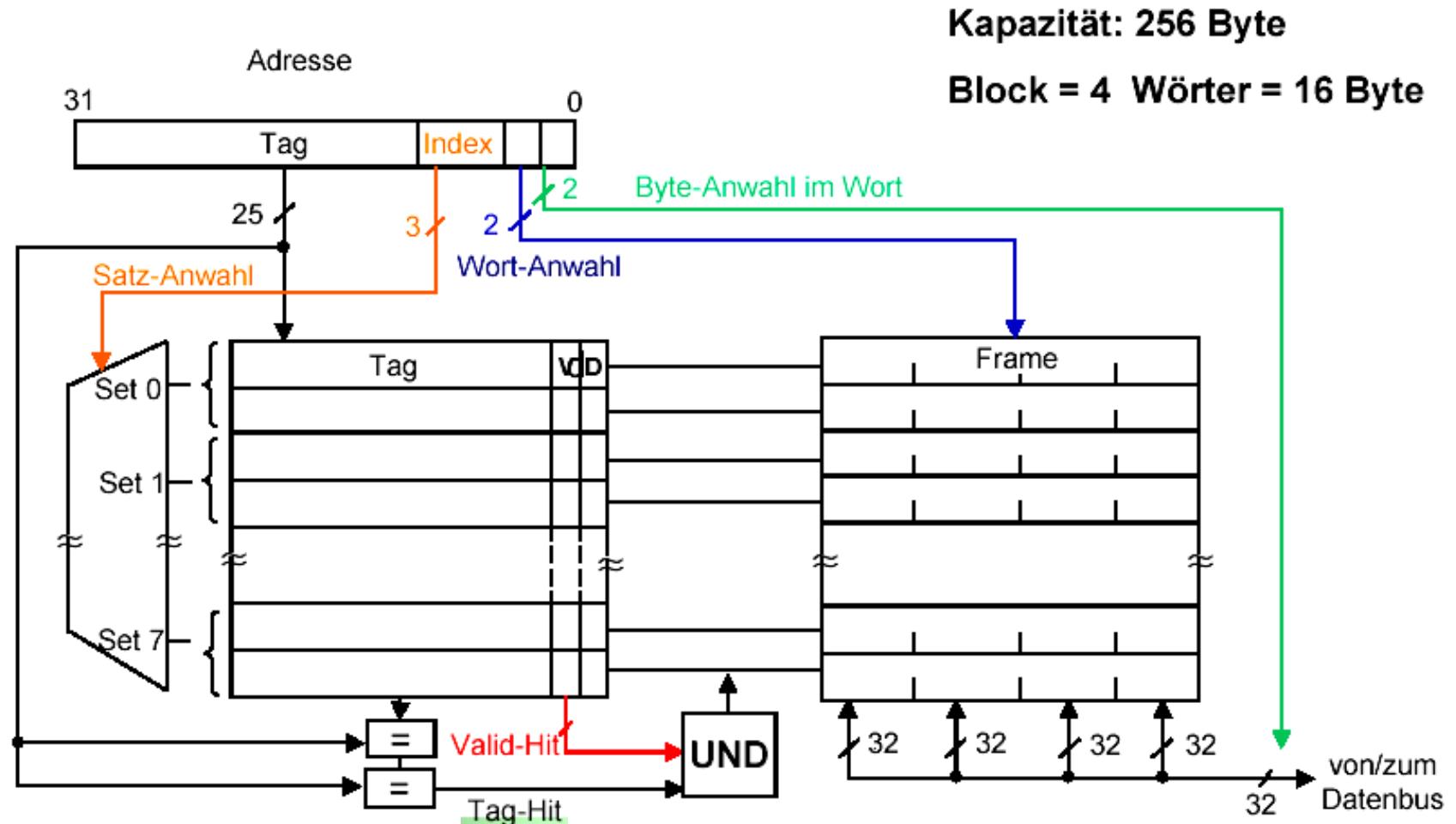


Satz-Nr.:= (Block-Nr.) mod (Satzanzahl)

Platzierung und Ersetzungsstrategie (A2)



n - way - set - associative Cache



n - way - set - associative Cache

Verbesserte Trefferrate, da hier eine Auswahl möglich ist
(der zu verdrängende Eintrag kann unter n ausgewählt werden)

Auswahlstrategien (Verdrängungsstrategien):

- ⇒ Zyklisch (der zuerst eingelagerte Eintrag wird auch wieder verdrängt, FIFO-Strategie)
- ⇒ Zufällig (durch Zufallsgenerator)
- ⇒ LRU-Strategie (*least recently used*) der am längsten nicht mehr benutzte Eintrag wird entfernt.

n - way - set - associative Cache

Zum Auffinden eines Datums müssen alle n Tags mit demselben Index parallel verglichen werden

- der Aufwand steigt mit der Zahl n,
für große n nähert sich der Aufwand den voll-assoziativen Caches

- Kompromiß zwischen Direct Mapped Cache und voll-assoziativem Cache

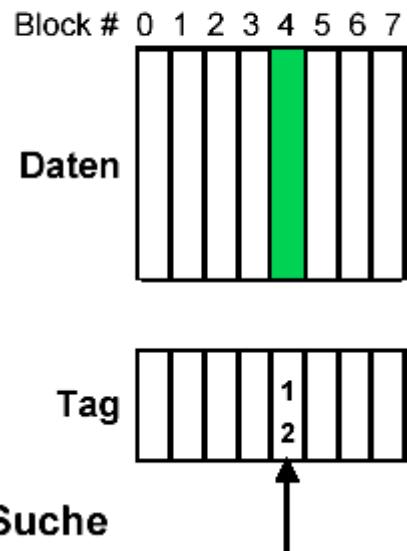
Finden der Einträge im Cache - Zusammenfassung

Feststellen, ob und wo sich die Daten zu einer Speicheradresse im Cache befinden (abhängig von der Cache-Organisation).

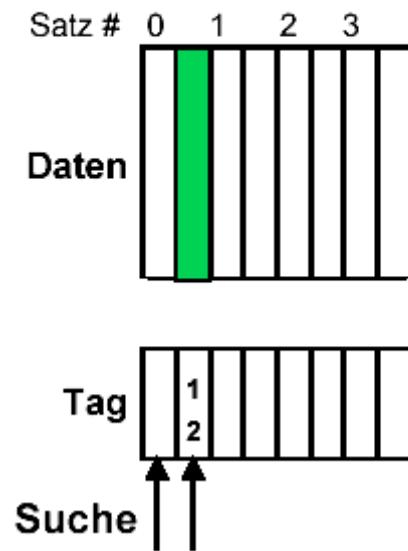
Assoziativität	Methode zur Lokalisierung des Cache-Eintrags	Anzahl der Tag-Vergleiche
Direct-mapped	Index	1
N-way-set-associative	Satz-Index, dann Suche in allen Einträgen im Satz	n
Fully-associative	Suche in allen Einträgen	#Einträge

Beispiel: Cache Organisation mit 8 Speicherplätzen

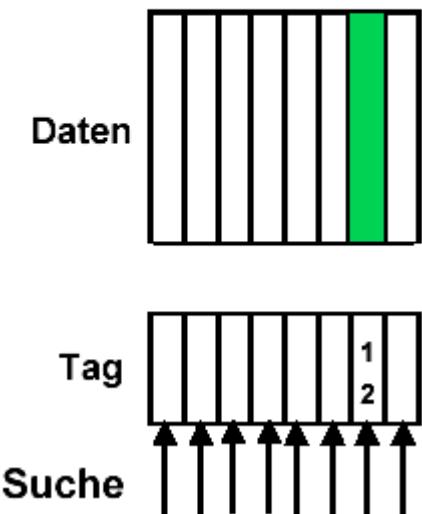
Direct-mapped



Set-associative



fully associative



Direct Mapped:

Speicherblock 12 kann nur an einer Stelle stehen

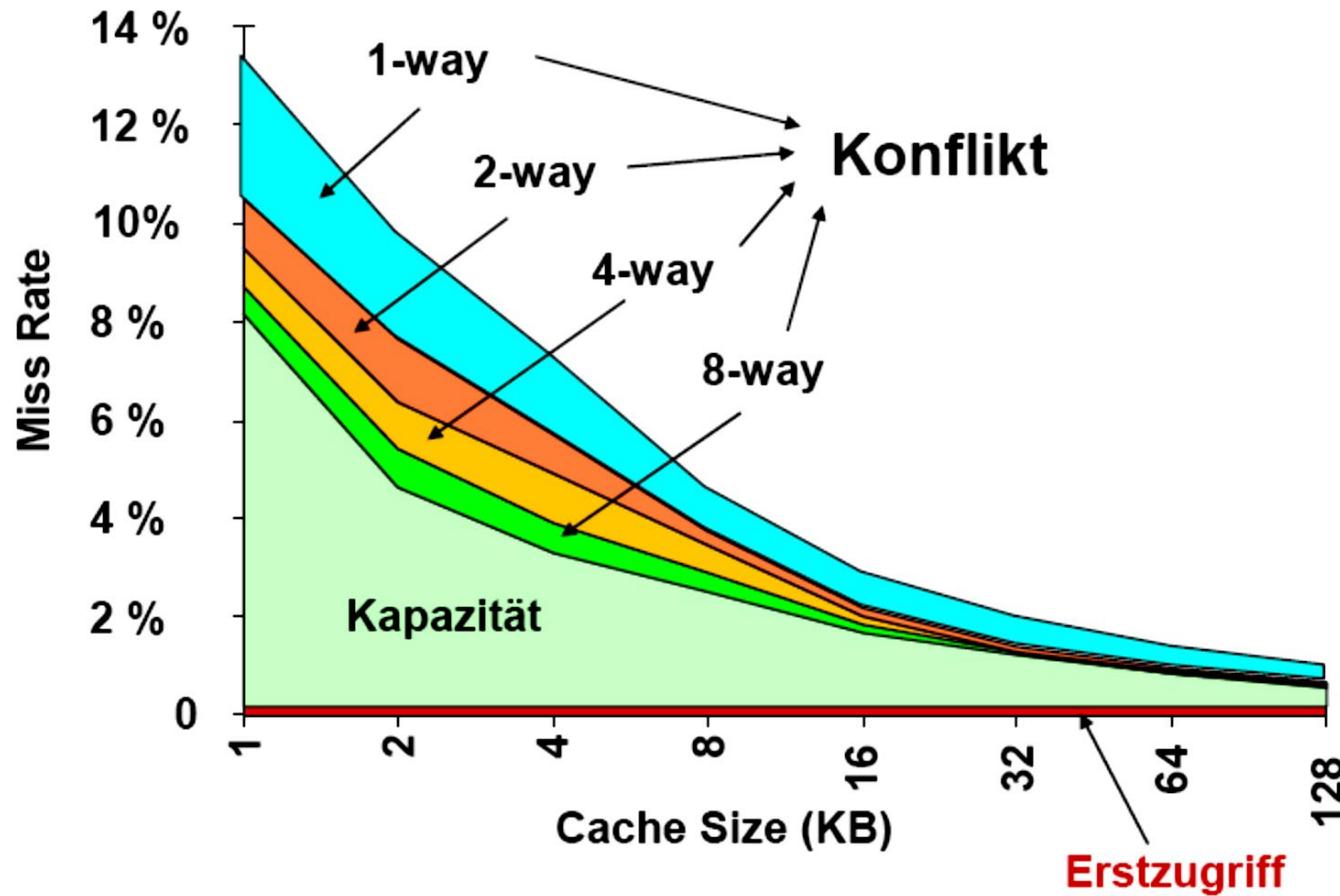
2-Way Set Associative:

Speicherblock 12 kann an zwei Stellen stehen

Voll-Assoziativ:

Speicherblock 12 kann an allen Stellen stehen

Erzielbare Cache - Trefferquoten



Verringern der Fehlerzugriffsrate

- **Erstzugriff (compulsory):**
 - Beim ersten Zugriff auf einen Cacheblock muss dieser erstmals geladen werden (*cold start misses*), sind unvermeidbar, egal wie groß der Cache ist.
- **Kapazität (capacity):**
 - Da nicht alle benötigten Cache-Blöcke aufgenommen werden können, müssen Cache-Blöcke verdrängt und später ggf. wieder geladen werden. Fehlzugriffe wegen mangelnder Kapazität sind von der Cache-Organisationsform unabhängig.
- **Konflikt (conflict):**
 - Bei satzassoziativen und direkt-abbildenden Caches können Konfliktfehlzugriffe auftreten, da ein Cacheblock potenziell verdrängt und später wieder geladen wird, falls zu viele Cache-Blöcke auf denselben Satz abgebildet werden (*collision misses*).

Cache Optimierung

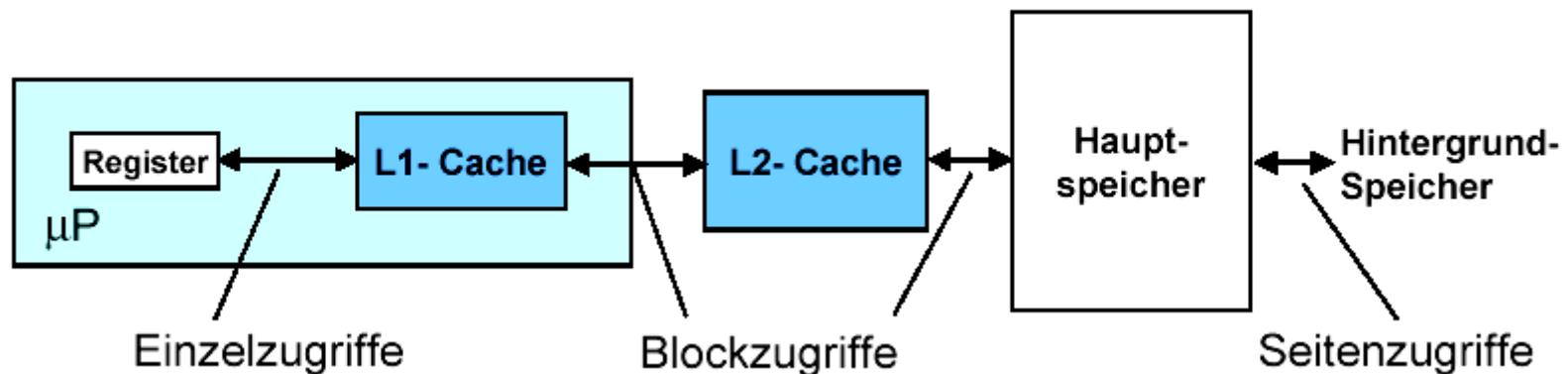
Veränderung	Auswirkung auf Miss Rate	Negative Auswirkung
Größerer Cache	reduziert Capacity-Misses	kann Zugriffszeit vergrößern
Höherer Grad an Assoziativität	reduziert Conflict-Misses	kann Zugriffszeit vergrößern
Größere Blöcke	reduziert Compulsory-Misses	größere Miss-Penalty

- Compulsory-Miss: Miss bei erstem Zugriff auf einen Block
- Capacity-Miss: Miss durch beschränkte Cache-Kapazität, auf ersetzen Block wird noch einmal zugegriffen
- Conflict-Miss: Miss durch beschränkte Anzahl Plätze in einem Set, würde in einem fully-associative Cache nicht auftreten

Verwendung mehrerer Caches

Oft findet sich eine mehrstufige Cache-Organisation:

- ⇒ Auf dem Prozessor-Chip befindet sich der sogenannte ***First-Level-Cache (On-Chip-Cache)***
- ⇒ Auf dem Prozessor-Chip befindet sich außerdem ein weiterer, größerer Cache, der sogenannte ***Secondary-Level-Cache (On-Board-Cache)***, 64 - 1024 KByte groß)



Multi-Level On-Chip Caches

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	-	Unified (instruction and data)
L3 cache size	-	8 MiB, shared
L3 cache associativity	-	16-way set associative
L3 replacement	-	Approximated LRU
L3 block size	-	64 bytes
L3 write policy	-	Write-back, Write-allocate
L3 hit time	-	35 clock cycles

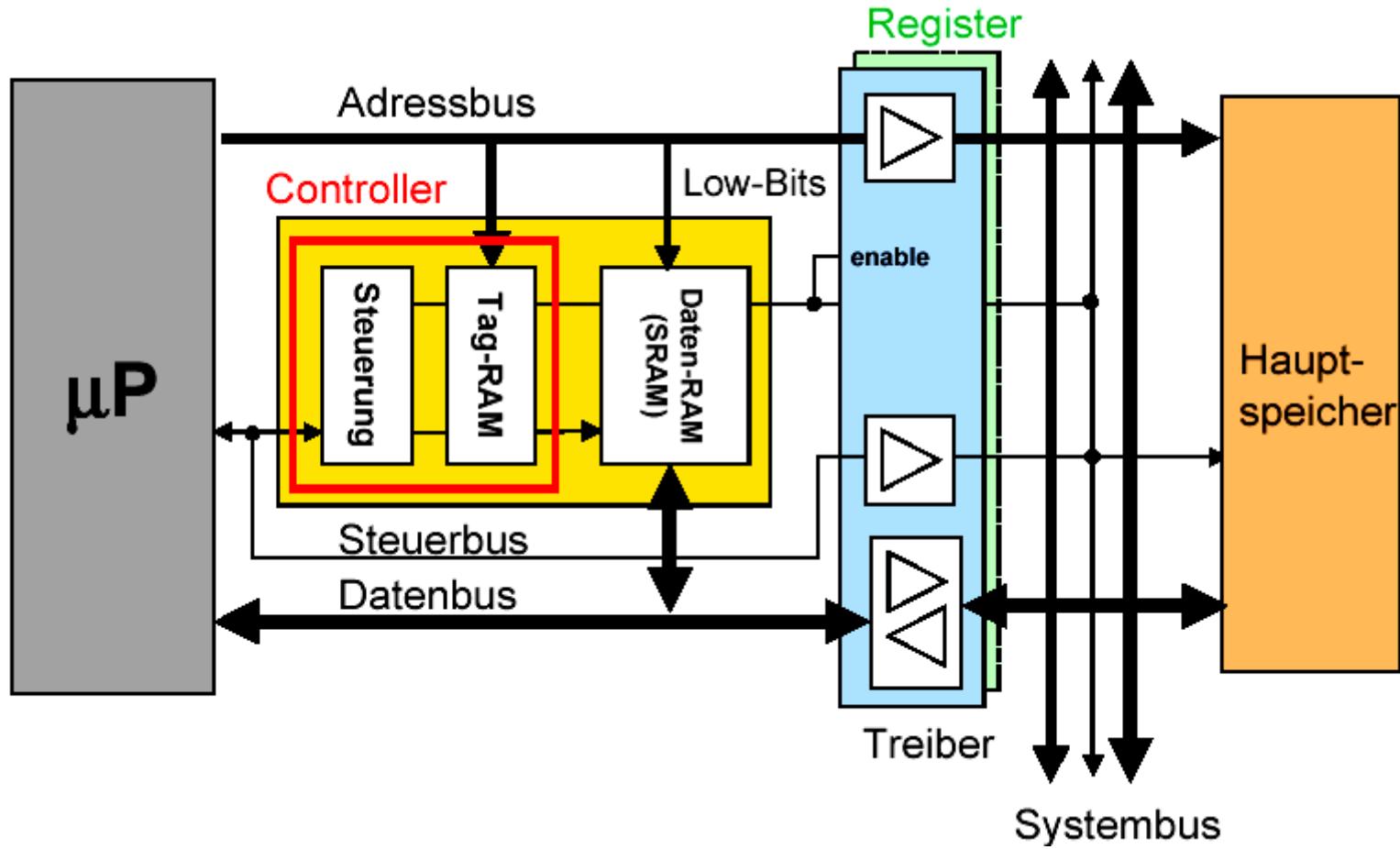
Verwendung mehrerer Caches

- ⇒ Der Secondary-Level-Cache kann parallel zum Hauptspeicher an den Bus angeschlossen werden (**Look-Aside-Cache**). Er sorgt dafür, daß bei einem First-Level-Cache-Miss die Daten schnell nachgeladen werden können

Havard-Architektur

- ⇒ Häufig getrennte On-Chip-Caches: **Befehlscache** für die Befehle und **Datencache** für die Daten.
 - paralleler Zugriff auf Programm und Daten, wodurch die hohen Anforderungen bei heutigen Superskalar-Prozessoren an die Speicherbandbreiten erfüllt werden können

Anbindung des Caches an den Systembus



Anbindung des Caches an den Systembus

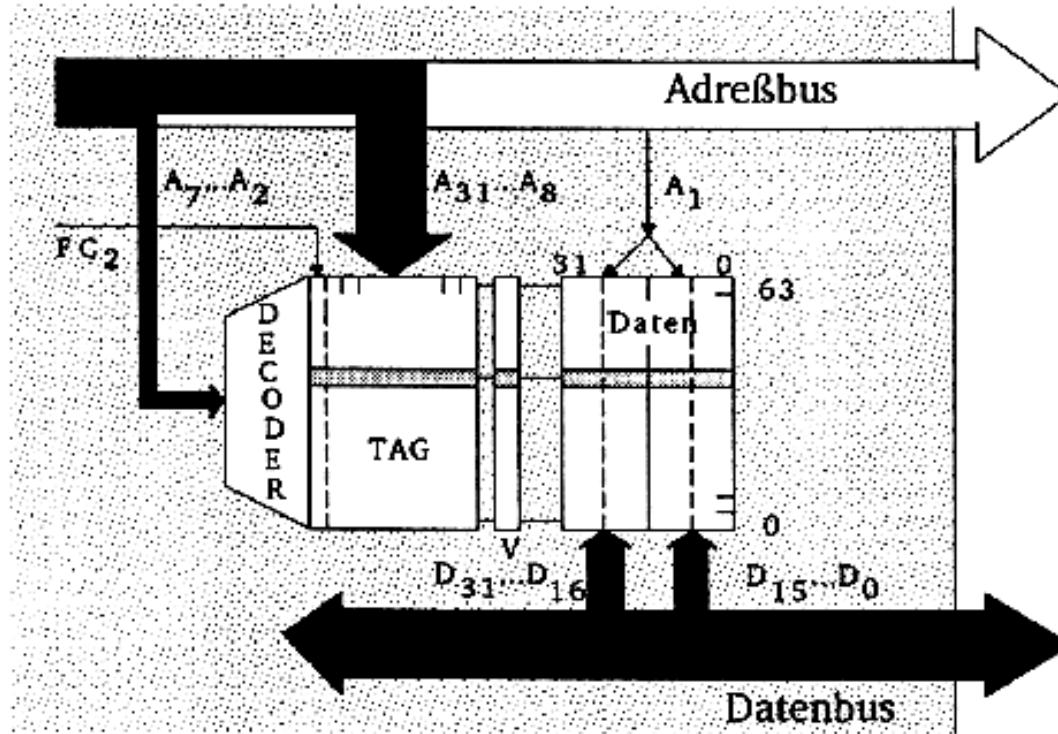
- **Cache-Controller:**

Tag-RAM + Steuerung + Tag-Komparator

Da dieser sehr schnell sein muß ➔ auf einem Chip integriert

- Cachespeicher selbst ist separat mit SRAM-Bausteinen aufgebaut
- Cache-Controller übernimmt die Steuerung der Treiber zum Systembus (Systembuszugriff nur bei Cache-Miss, sonst ist der Systembus für andere Komponenten frei), sowie der Systembussignale zur Einfügung von Wartezyklen bei Cache-Miss

Befehlscache des Motorola 68020



Auf dem Prozessorchip integriert. Reiner Befehlscache
64 Einträge, jeder Eintrag umfaßt 4 Bytes

Befehlscache des Motorola 68020

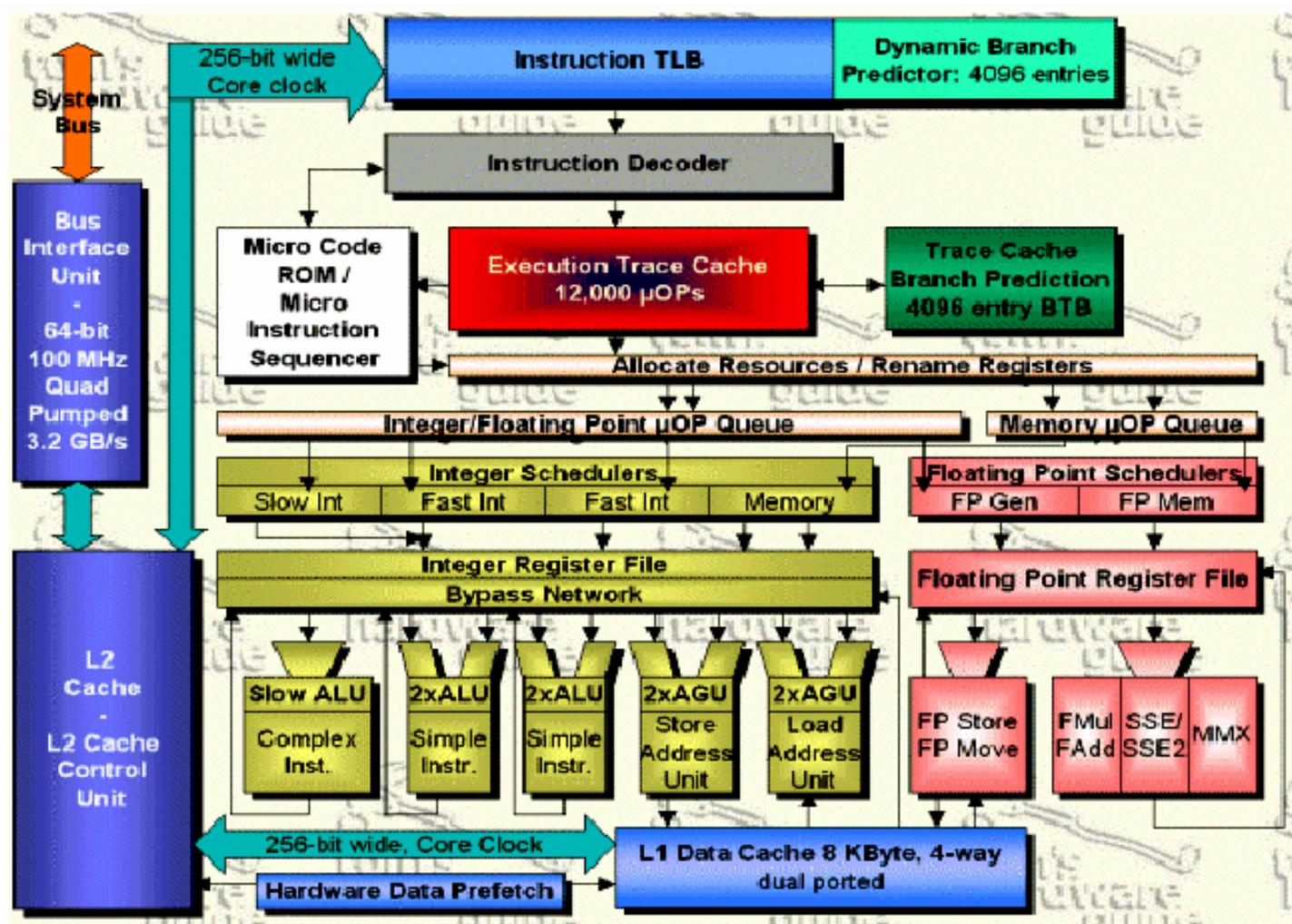
Da Befehle eines Programms während der Ausführung nicht verändert werden ➔ keine Konsistenzprobleme

Cache-Adressierung:

- Auswahl eines 16-Bit Cache-Wortes aus einem 32-Bit Cache-Block durch A1
- Adressierung eines Cache-Blocks durch A2 - A7
- Tag-Information: A8 - A31, zusätzlich FC2 zur Kennzeichnung, ob Adresse des Benutzer- oder Betriebssystembereichs vorliegt

Statusbit V kennzeichnet die Gültigkeit eines Eintrags

Cache Speicher im Pentium 4



Cache Speicher im Pentium 4

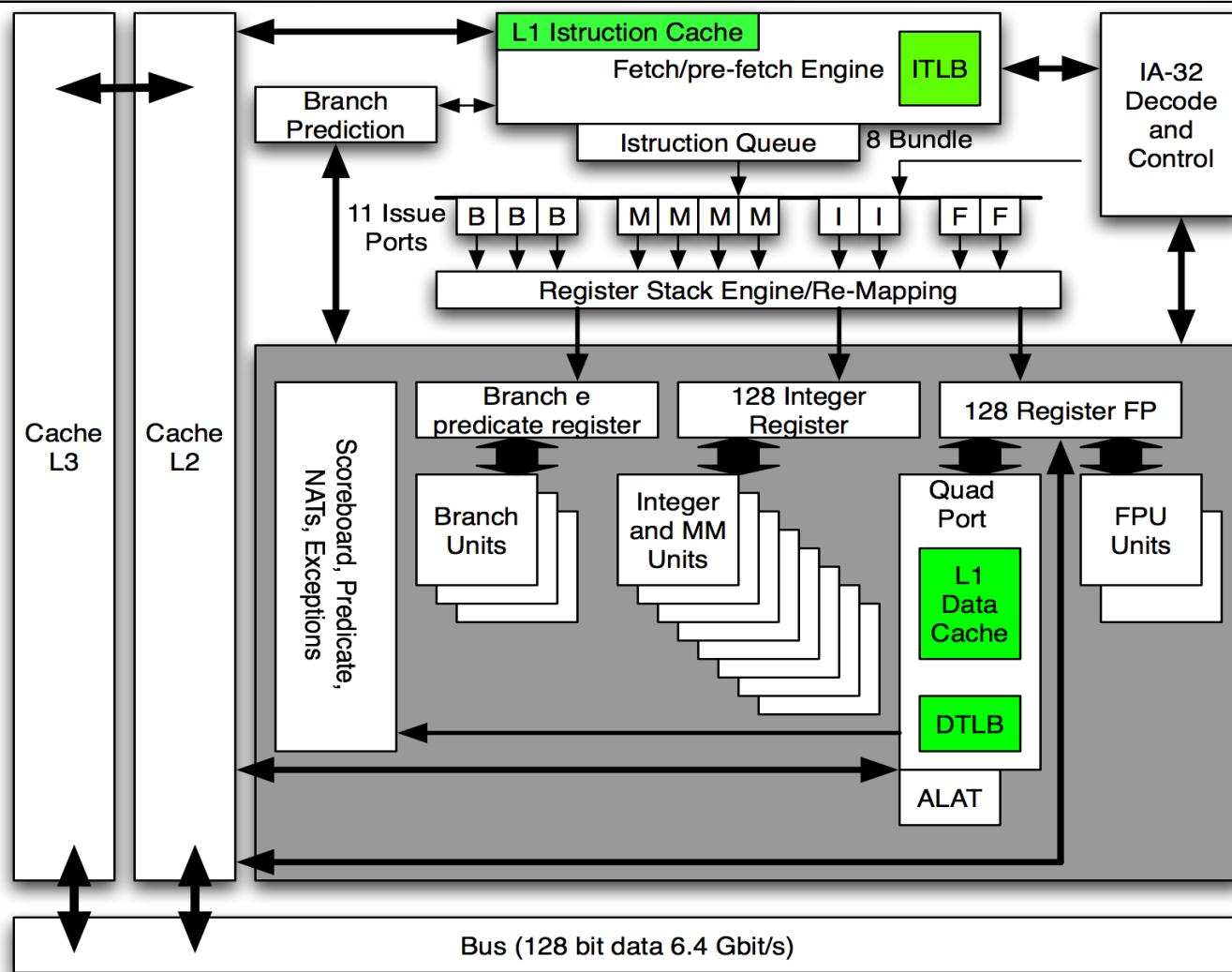
□ L1-Cache:

- 8 Kbyte Daten-Cache (Bei Pentium III: 16 Kbyte; bei Athlon: 64 Kbyte)
- 4-way set associative Daten-Cache
- Cache-lines mit 64 Byte (dual-pot-Architektur)
- Write-Through

□ L2-Cache:

- 256 Kbyte
- 8-way set associative
- Cache-lines mit 128 Byte
- Write back
- Bandbreite 44,8 Gbyte/s (16 Gbyte/s bei Pentium III)

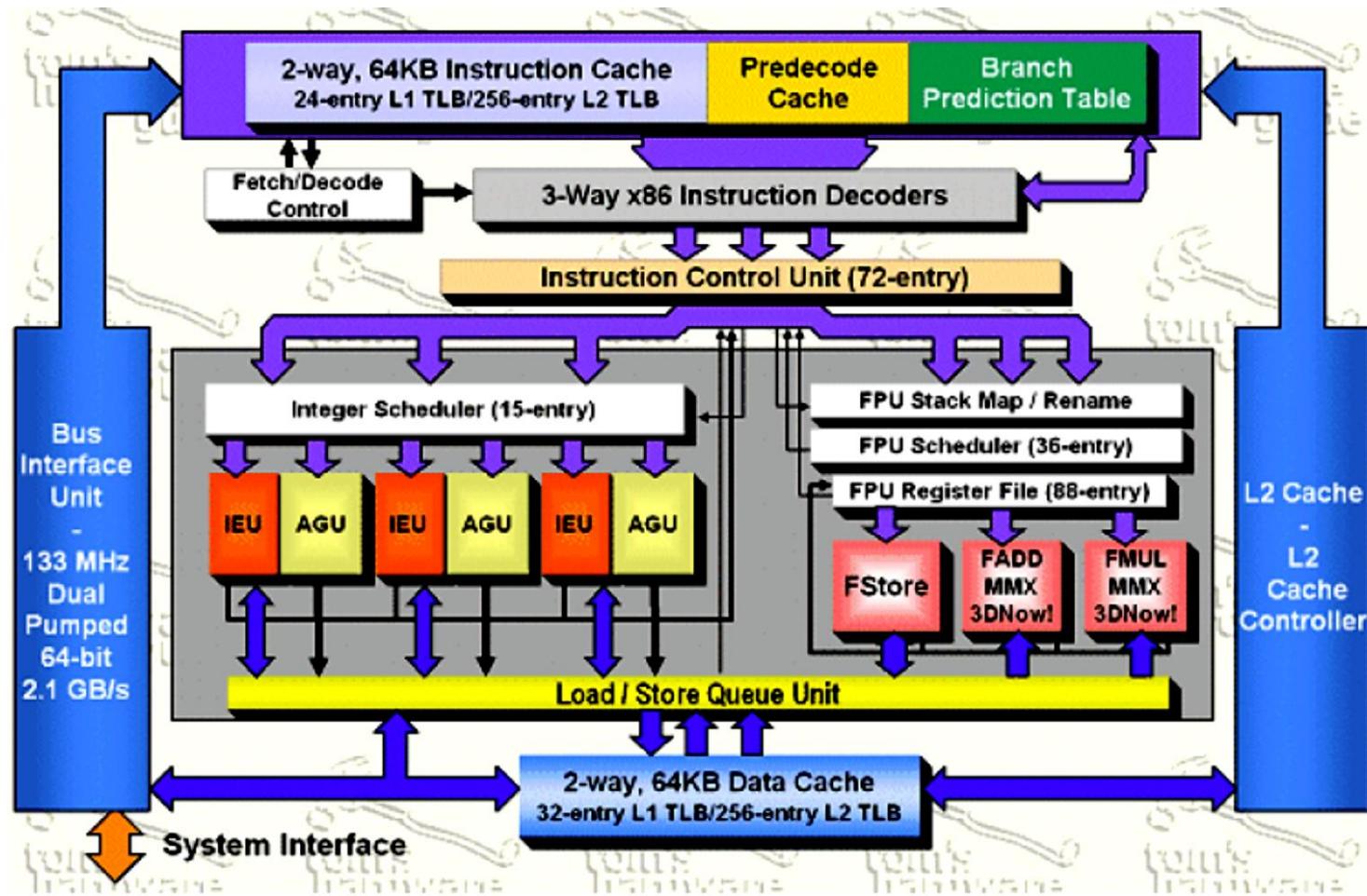
Beispiel: Itanium 2



Beispiel: Itanium 2

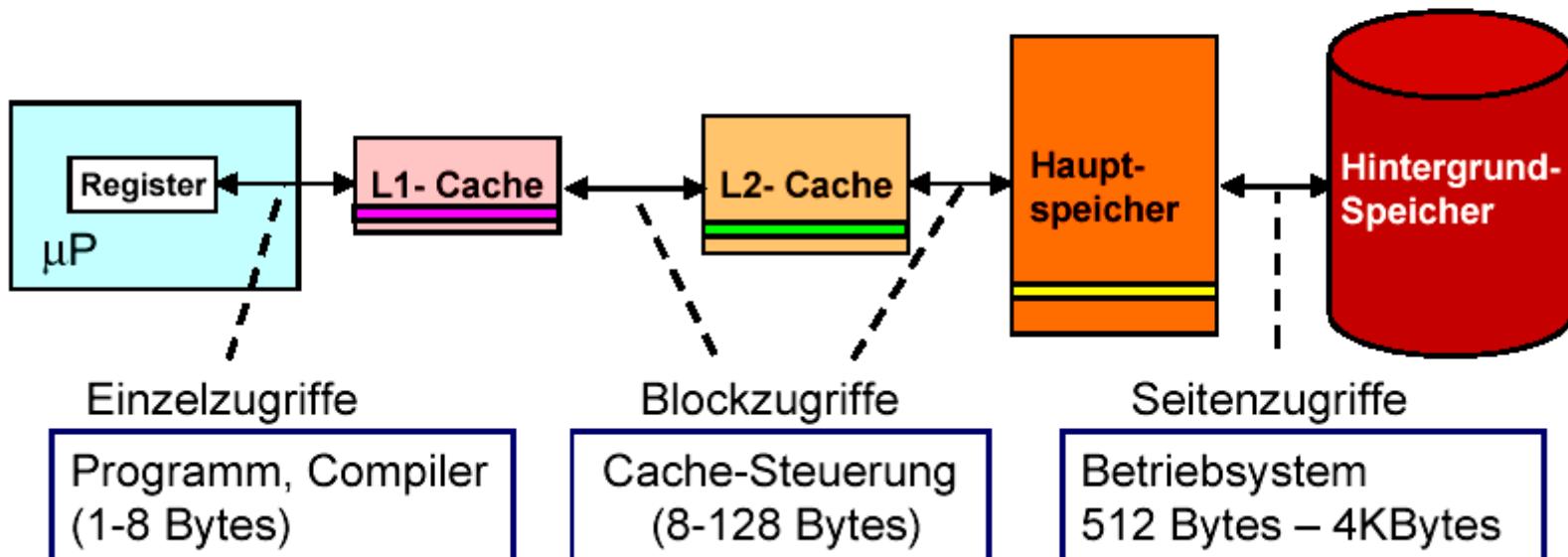
- Dreistufige Cache-Hierarchie
- L1-Cache: *on-chip*, getrennt für Daten und Befehle
 - je 16 KByte, 4-Wege satzassoziativ, Cachezeilen mit 32 Byte
- L2-Cache: *on-chip*
 - 96 KByte, 6-Wege satzassoziativ, Cachezeilen mit 64 Byte
- L3-Cache: *off-chip*, aber noch im Prozessormodul
 - 4 MByte, 4-Wege satzassoziativ, Cachezeilen mit 64 Byte

Cachespeicher bei Athlon



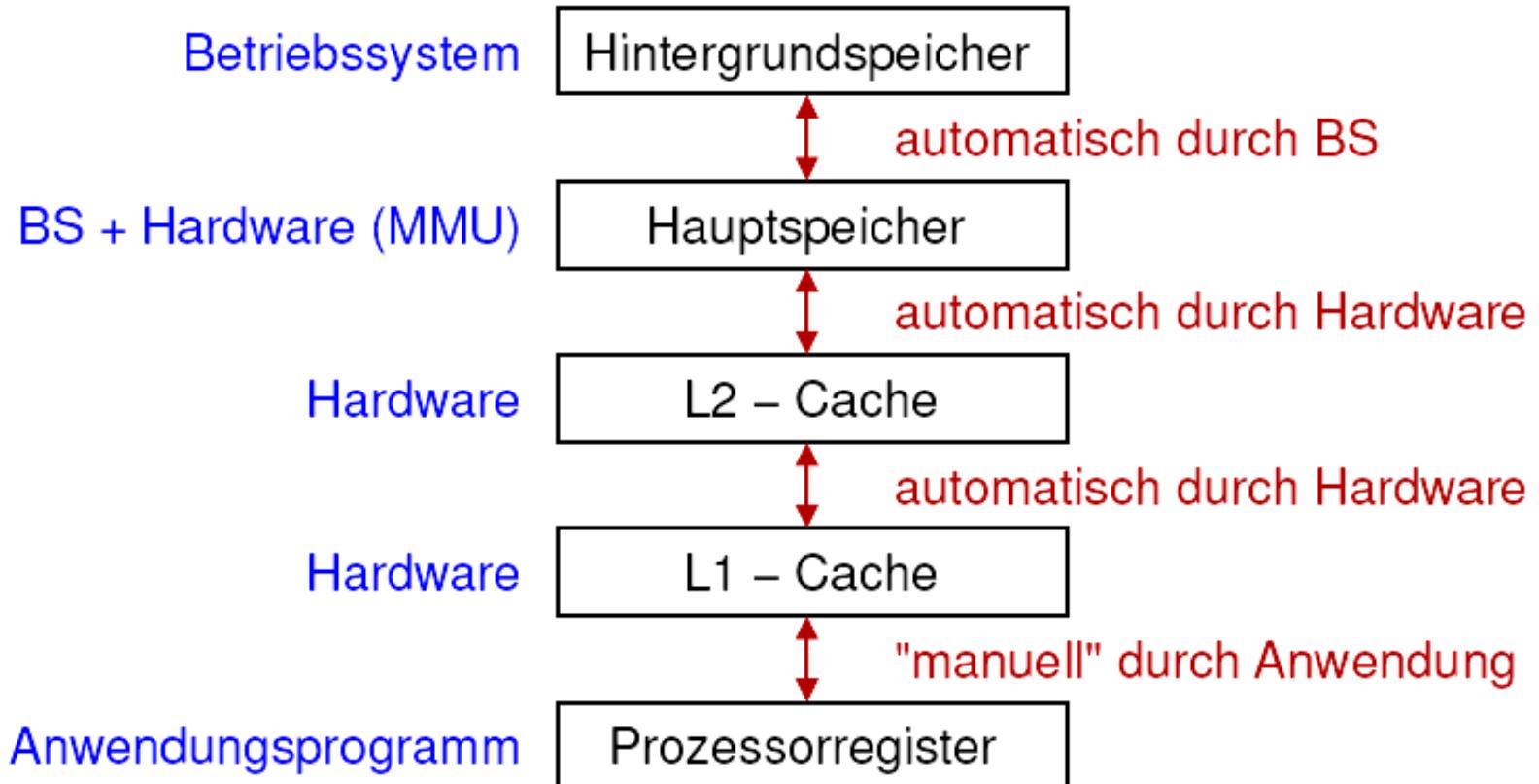
Zusammenfassung: Speicherhierarchie

Daten werden nur zwischen aufeinanderfolgenden Ebenen der Speicherhierarchie kopiert.



Zusammenfassung: Speicherhierarchie

Verwaltung durch:



Datentransport:

Virtuelle Speicherverwaltung

Spezielle Aufgaben des Betriebssystems

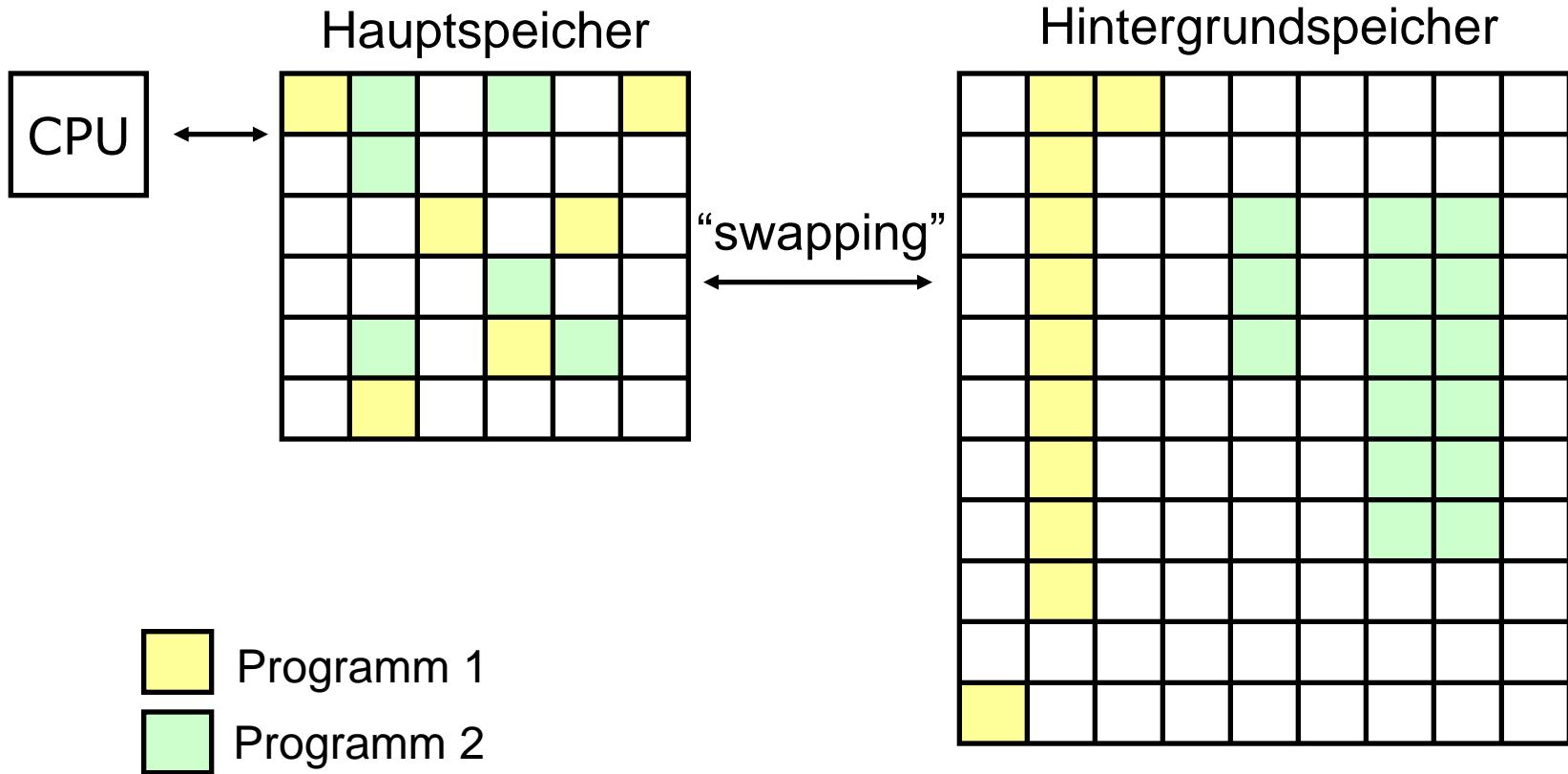
□ Speicherverwaltung (*memory management*)

Durch immer größer werdende Programme sowie mehrerer quasi gleichzeitig laufender Programme

- der zur Verfügung stehende Arbeitsspeicher wird schnell zu klein

Abhilfe: Nur die gerade benötigten Teile der aktiven Programme werden wirklich im Arbeitsspeicher gehalten, der Rest befindet sich im Hintergrundspeicher und wird bei Bedarf geladen (*swapping, paging*).

Grundstruktur virtueller Speicherverwaltung

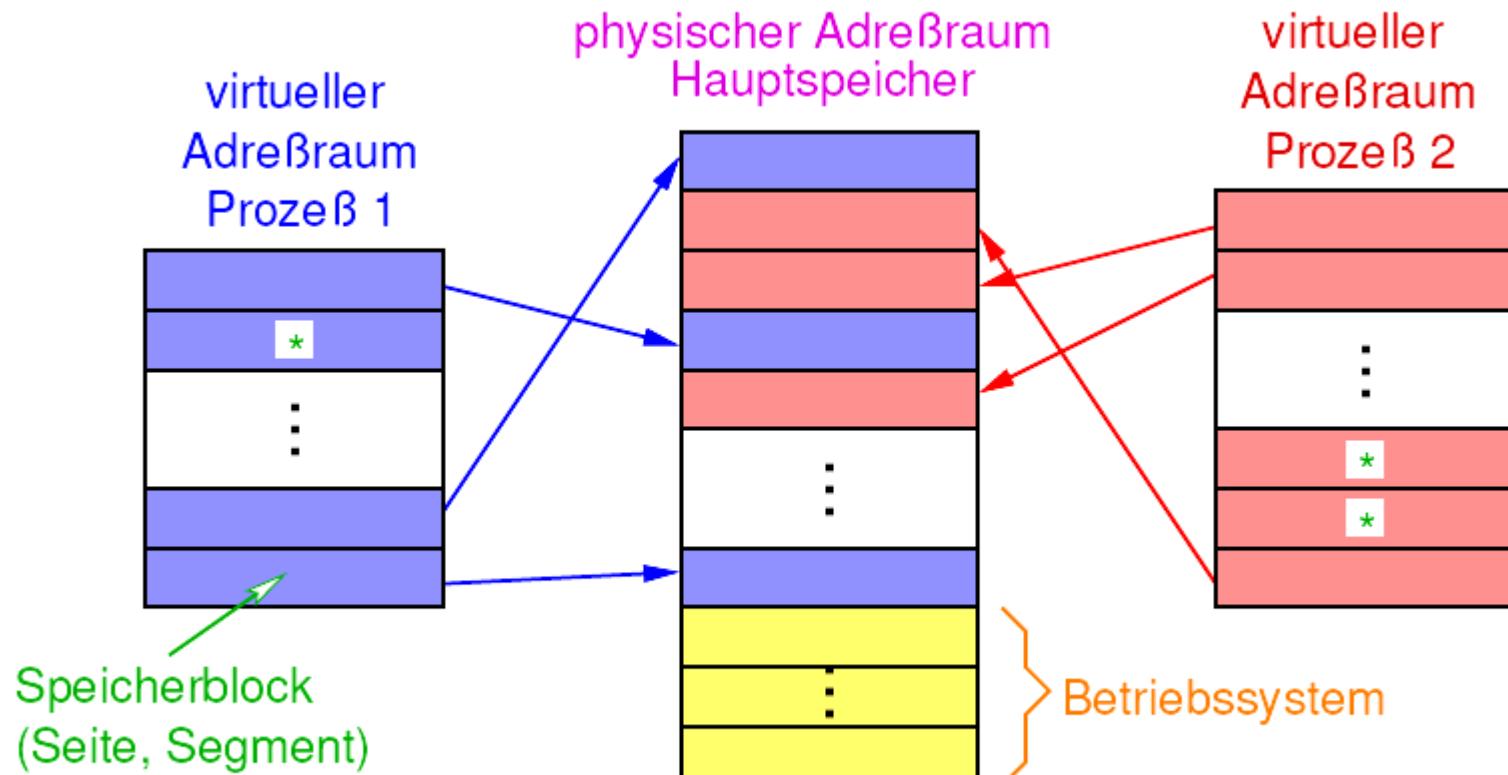


Prinzip der virtuellen Speicherverwaltung

- Dieser Vorgang bleibt dem Anwender völlig verborgen, d.h. der Arbeitsspeicher erscheint dem Anwender wesentlich größer, als er in Wahrheit ist.
- Ein nach diesem Konzept verwalteter Speicher heißt deshalb virtueller Speicher.
- Von modernen Prozessoren wird die Verwaltung dieses virtuellen Speichers hardwaremäßig durch eine MMU (memory management unit) unterstützt.
- Hauptaufgabe dieser virtuellen Speicherverwaltung:
Umsetzung virtueller (logischer) Adressen in physikalische Adressen

Prinzip des virtuellen Speichers

Speicherabbildung bei virtueller Speicherorganisation



- * Diese Blöcke sind nicht in den Hauptspeicher abgebildet
Sie könnten z.B. auf Hintergrundspeicher ausgelagert sein.

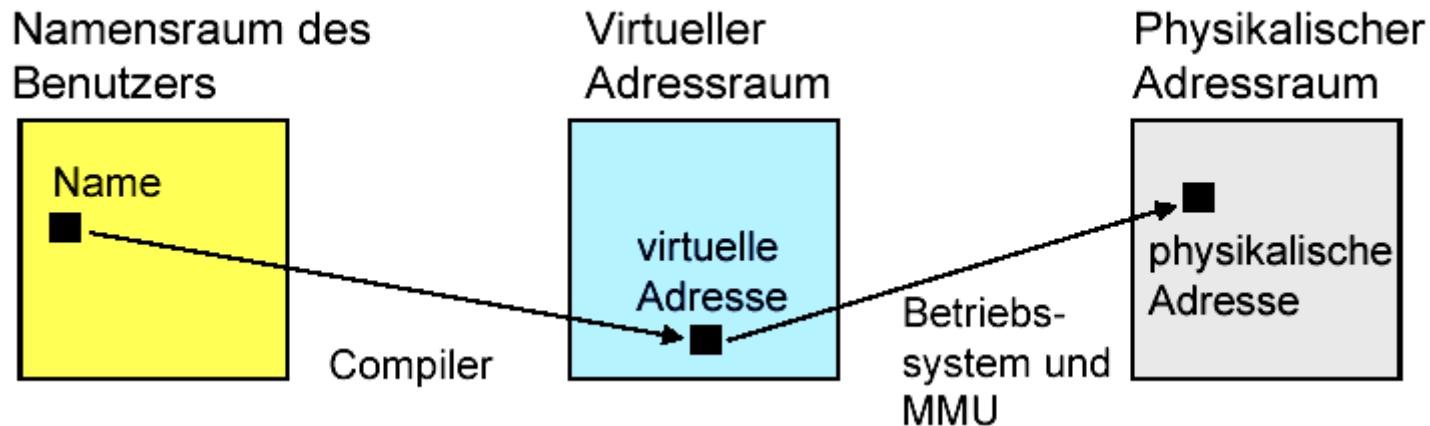
Prinzip der virtuellen Speicherverwaltung

- Zur Erinnerung: Die Auswertung eines Adressausdrucks in einem Befehl liefert eine **logische (bzw. virtuelle) Adresse**.
- Prozessoren unterstützen eine virtuelle Speicherorganisation, um die **Mehrprozeß- und Mehrbenutzerfähigkeit** von Betriebssystemen zu ermöglichen.
 - Rechner arbeitet zu einem Zeitpunkt mehrere Prozesse verschiedener Benutzer (quasi-)gleichzeitig ab.
- Die virtuelle Adresse wird nach Vorgaben des Betriebssystems in die **physische Adresse** umgesetzt, mit der der Hauptspeicher adressiert wird.
- Virtuelle Speicherorganisation erleichtert / ermöglicht diese Aufgaben des Betriebssystems.

Prinzip der virtuellen Speicherverwaltung

- Jeder Prozess besitzt seinen eigenen virtuellen Adressraum
 - Ein Prozess adressiert seinen Speicher mit virtuellen Adressen
- Betriebssystem kann dem Prozess physischen Speicher an beliebiger Stelle im Hauptspeicher zuweisen
 - Betriebssystem erzeugt dazu entsprechende Umsetzungstabellen
 - Bei Adressumsetzung: MMU prüft auch Zugriffsrechte bei Rechteverletzung, ggf. Erzeugung einer Ausnahme.
- Betriebssystem kann Blöcke des virtuellen Adressraums auslagern
 - Rückspeichern der Blöcke auf den Hintergrundspeicher
 - Bei erneutem Zugriff durch den Prozess: Ausnahme
 - Ausnahmebehandlung: Einlagern des benötigten Blocks, Aktualisierung der Umsetzungstabellen, Fortsetzung des Prozesses

Virtuelle Speicherverwaltung

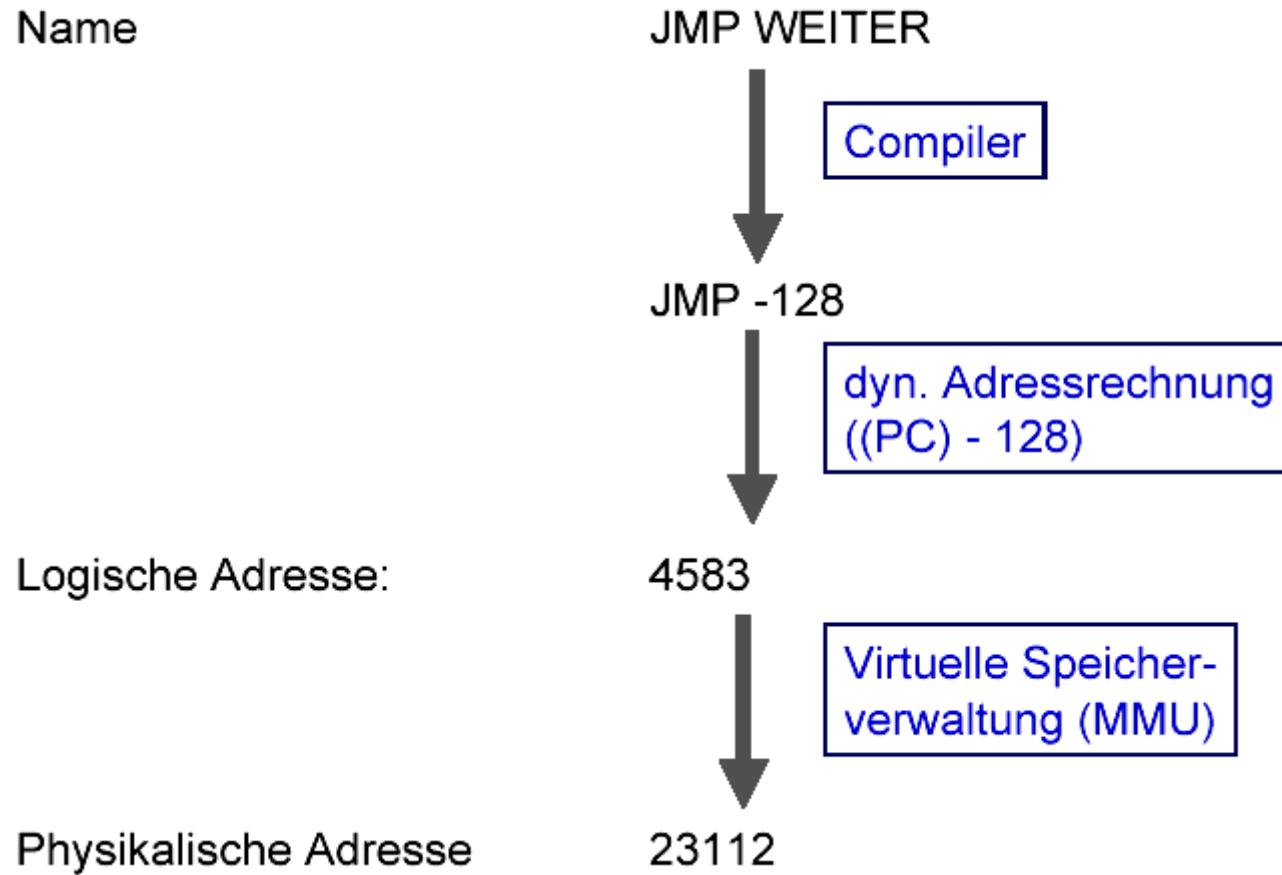


Benutzer: kennzeichnet seine Objekte (Programme, Unterprogramme, Variablen, ...) durch Namen

Compiler: übersetzt diese Namen in virtuelle (logische) Adressen.

Virtuelle Speicherverwaltung: wandelt diese Adressen zur Laufzeit je nach gerade gegebener Speicherbelegung in die physikalische Adresse (wirklicher Ort des Objekts im Hauptspeicher) um

Virtuelle Speicherverwaltung - Beispiel



Virtuelle Speicherverwaltung

- Das **Betriebssystem** führt Buch über die freien Speicherbereiche und lagert bei nicht ausreichendem Freiplatz im Hauptspeicher diejenigen Speicherinhalte auf den Hintergrundspeicher aus, die gegenüber ihrem Orginalen auf dem Hintergrundspeicher verändert worden sind.
- Die eindeutige Abbildung des großen virtuellen Speichers auf die effektive Hauptspeicherkapazität wird von der **Hardware** durch **Speicherverwaltungseinheiten** unterstützt (*memory management units, MMU*)
- Die erforderliche Abbildungsinformation stellt das Betriebssystem in Form einer oder mehrerer **Übersetzungstabellen** zur Verfügung.

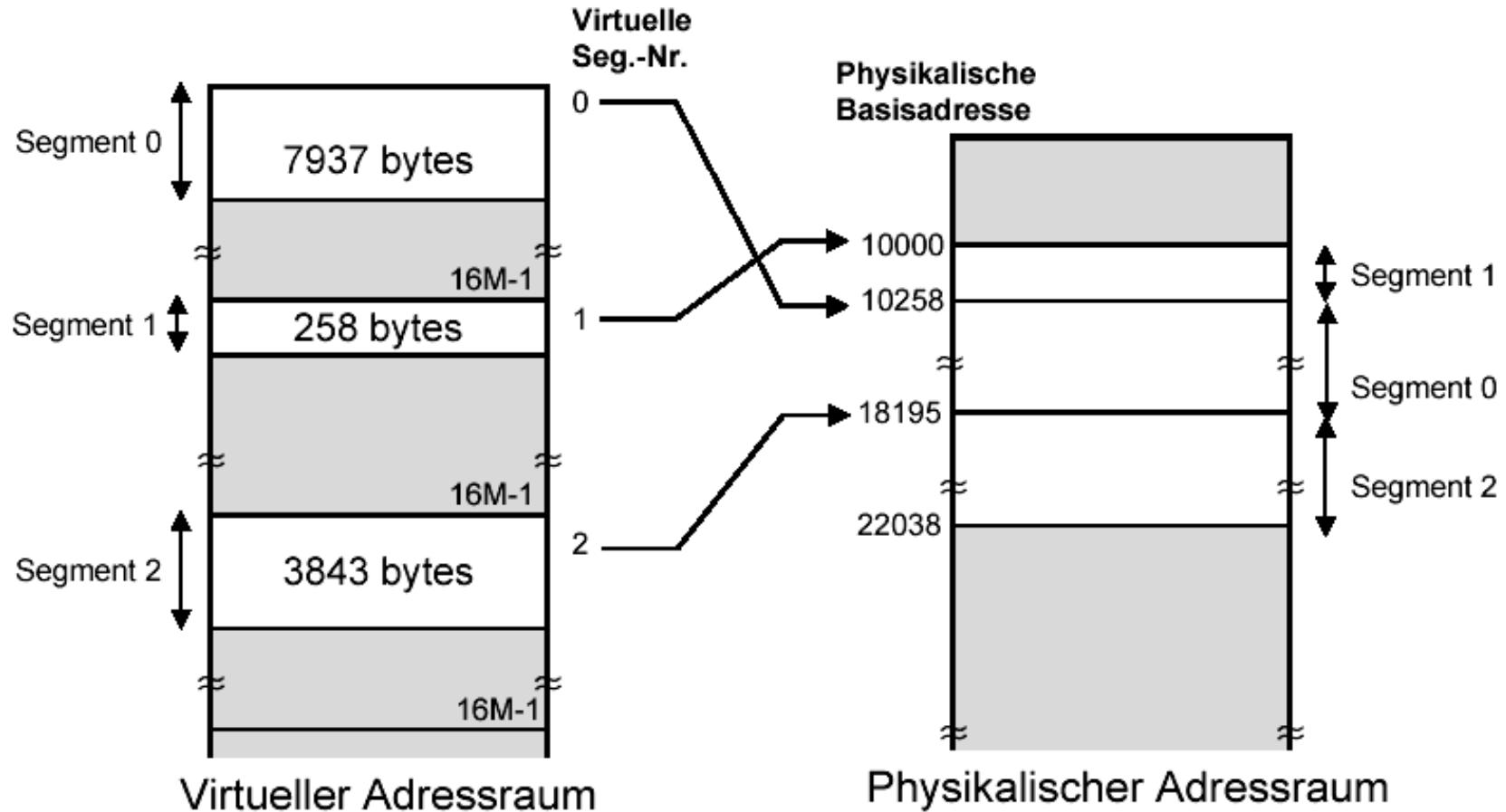
Segmentierungs- und Seitenwechselverfahren

Es existieren zwei grundlegende Verfahren zur virtuellen Speicherverwaltung (Segmentierung und Seitenwechsel)

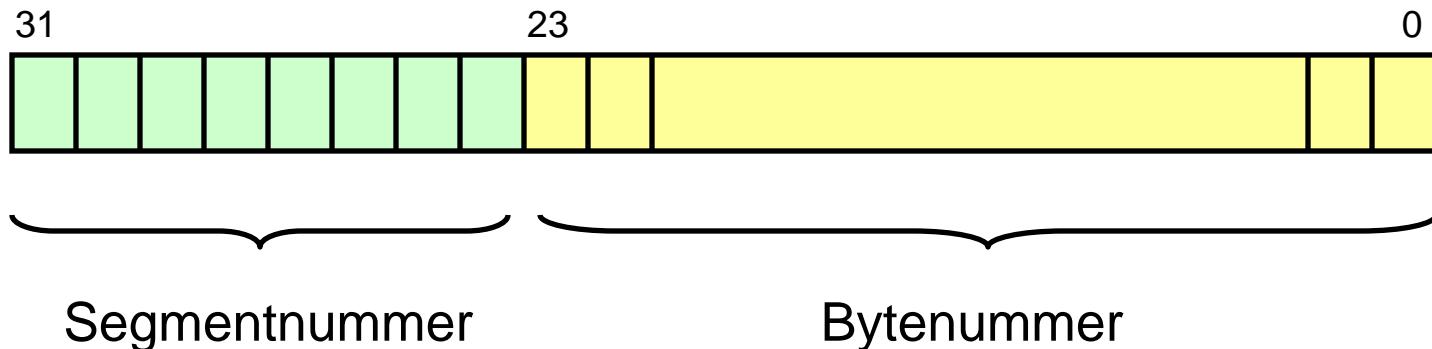
Segmentierung

- Hierbei wird der virtuelle Adressraum in Segmente verschiedener Länge zerlegt.
- Jedem Prozess sind ein oder mehrere Segmente z.B. für den Programmcode und die Daten, zugeordnet.
- Die einzelnen Segmente enthalten logisch zusammenhängende Informationen (Programm- und Datenmodule) und können relativ groß sein.

Virtueller und realer Adressraum bei Segmentierung



Segmentbasierte Speicherverwaltung

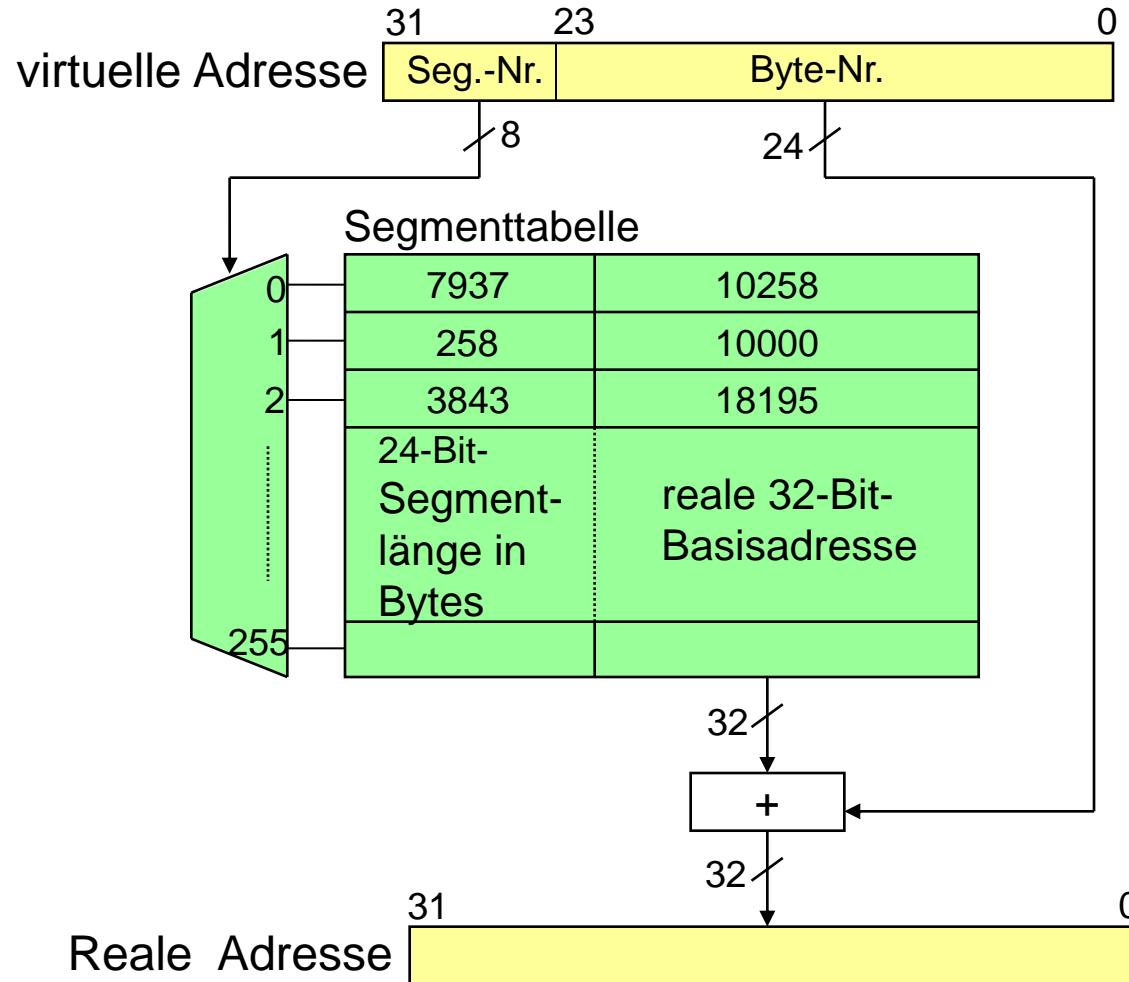


- Die virtuelle Adresse wird in zwei Anteile aufgeteilt:
 - Segmentnummer
 - Bytenummer

Segmentbasierte Speicherverwaltung

- Virtuelle Adresse wird in eine Segmentnummer (**n** höherwertige Bits der virtuellen Adresse) als Kennung eines Segments und in eine Bytenummer (verbleibenden **m** Bits der Adresse) als Abstand zum Segmentanfang.
- Maximale virtuelle Segmentanzahl = 2^n , Maximale Segmentgröße = 2^m
- Die Adressabbildung erfolgt über eine Segmenttabelle (im Registerspeicher der MMU).
- Reale Adresse ergibt sich aus der Segmentbasisadresse, zu der die virtuelle Bytenummer addiert wird.
- Segmentlängenangaben in der Segmenttabelle, um segmentüberschreitende Zugriffe feststellen und ggf. verhindern zu können.
- Bei Segmenten mit einer geringeren Größe als 2^m , gilt der ungenutzte Raum als Verschnitt.
- Gute Ausnutzung des Hauptspeichers, wenn man die Segmentgrenzen an jeder Byteadresse zulässt.

Segmentbasierte Speicherverwaltung



Segmentbasierte Speicherverwaltung

Der Speicher wird in ein oder mehrere physikalische Segmente (zusammenhängende Speicherbereiche variabler Länge) unterteilt

Oft verschiedene Segmenttypen (für Code, Daten, ...)

Jeder Segment-Deskriptor beschreibt das zugehörige Segment durch folgende Attribute:

- die Segment-Basisadresse (*base address*)
- die Segment-Größe in Bytes (*limit*)
- die Zugriffsrechte auf das Segment (*access rights*)

Segmentbasierte Speicherverwaltung

- **Vorteile:**

- Daten in den Segmenten sind logisch zusammengehörig
- Segment ist sinnvolle Einheit für Rechtevergabe
 - Z.B. Schreibverbot auf Code-Segment, Ausführungsverbot für Daten- und Kellersegmente
- Einhaltung der Segmentgrenzen exakt überwachbar

- **Nachteile:**

- Segmente können sehr groß werden
- Speicherverwaltung muss ein genügend großes, zusammenhängendes Stück physischen Speicher finden
- Ein-/Auslagern eines Segments kann sehr lange dauern
- Keine partielle Auslagerung eines Segments möglich

Segmentierungs- und Seitenwechselverfahren

Aufteilung in Seiten

- Hierbei wird der logische und der physikalische Adressraum in "Segmente fester Länge", die sogenannten Seiten (pages) unterteilt.
- Die Seiten sind relativ klein (256 Byte - 4 kByte)
- Ein Prozess wird auf viele dieser Seiten verteilt (keine logischen Zusammenhänge wie bei der Segmentierung)

Im Unterschied zu Segmenten können Seiten nicht an beliebiger Stelle im Speicher beginnen, sondern nur in einem festen an der Seitengröße orientierten Raster

Seitenaufteilung

Vorteile:

- ⇒ durch kleine Seiten wird nur der wirklich benötigte Teil eines Programms eingelagert.
- ⇒ geringerer Verwaltungsaufwand als Segmentierung

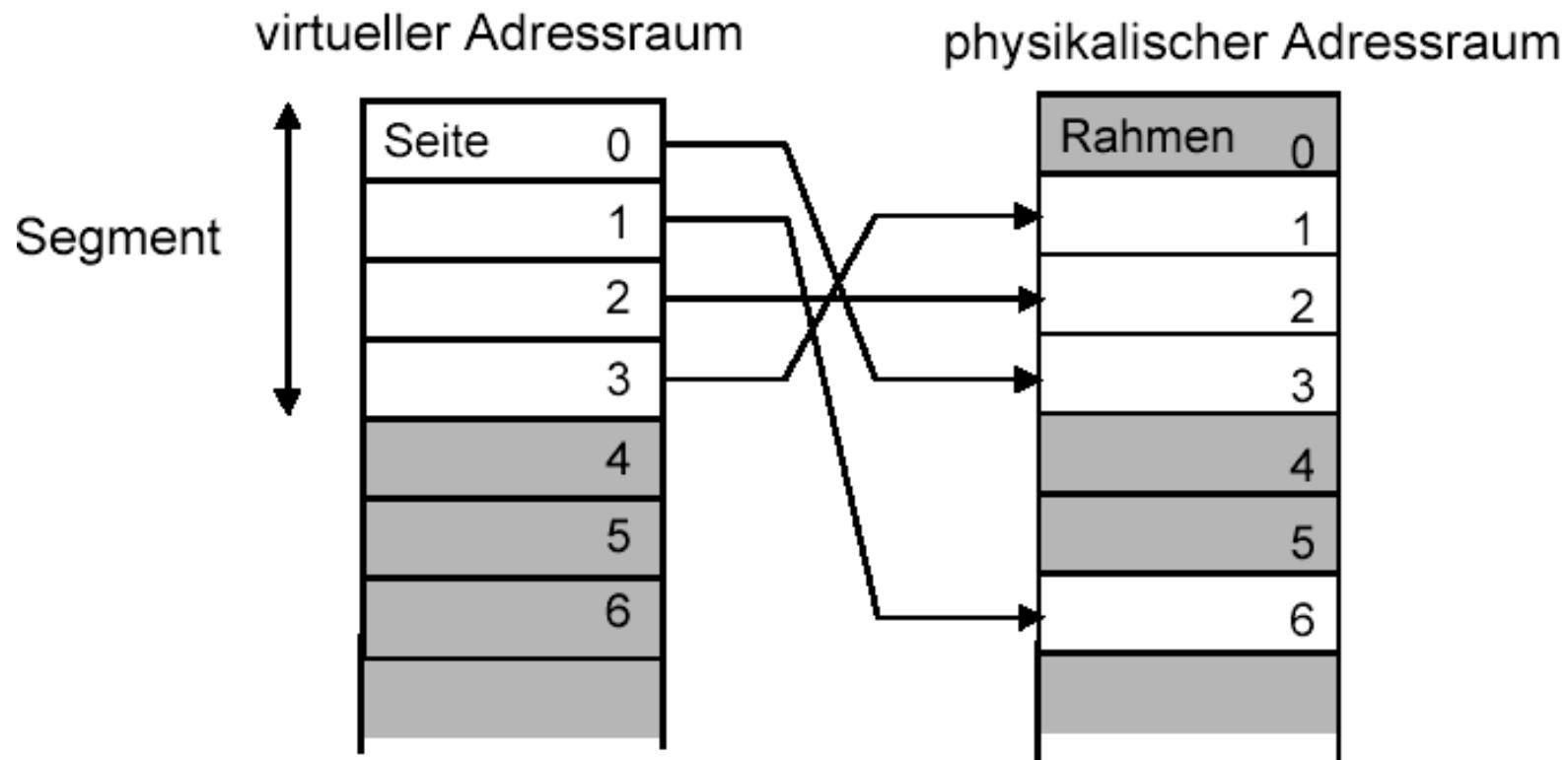
Nachteil:

- ⇒ häufiger Datentransfer

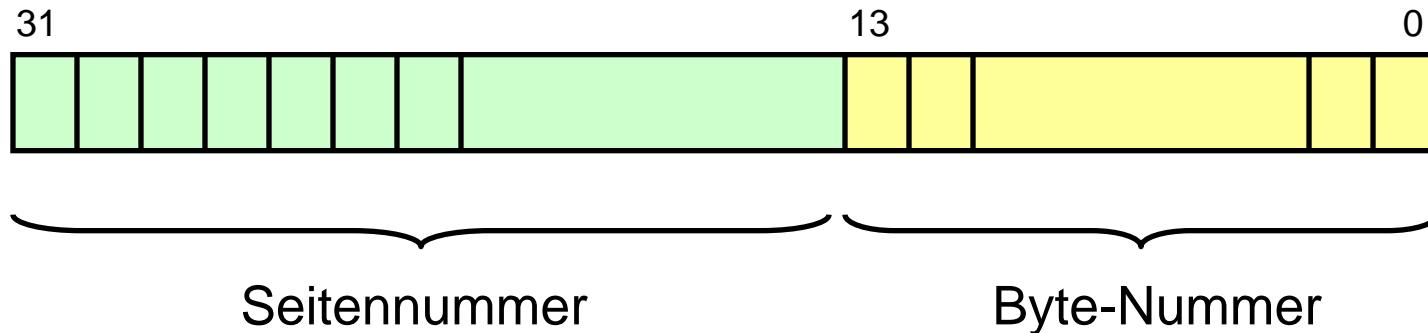
Ältere Prozessoren unterstützten jeweils nur ein Verfahren, z.B. Segmentierung bei 80286, 68010 und Seitenwechsel bei Z8003/4, National 32000

Heutige Mikroprozessoren, z.B. 80386, 80486, Pentium, ... unterstützen beide Verfahren

Virtueller und realer Adressraum beim Paging

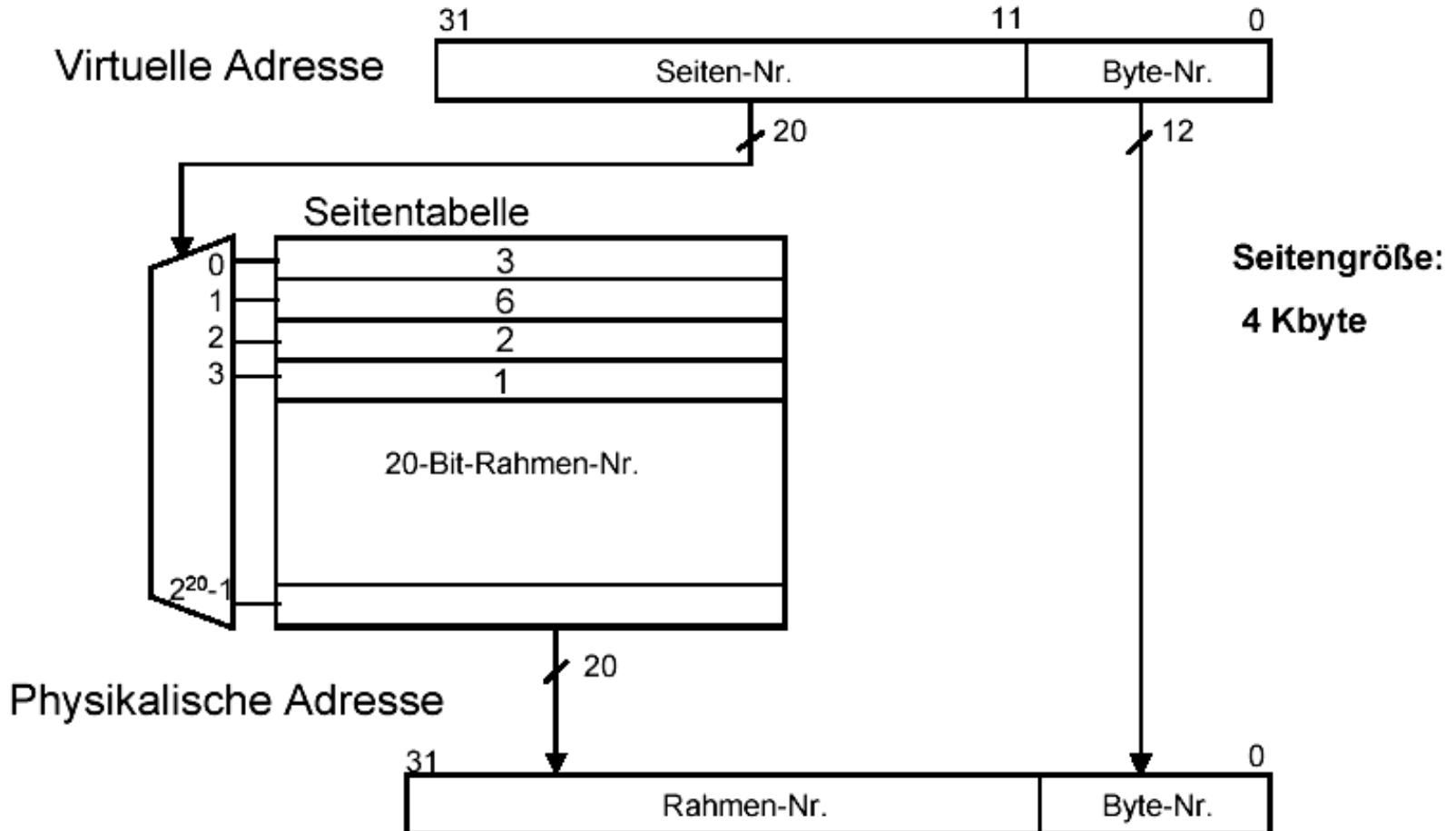


Seitenorientierte Speicherverwaltung



- Die virtuelle Adresse wird in zwei Anteile aufgeteilt:
 - Seitennummer
 - Bytenummer

Seitenorientierte Speicherverwaltung



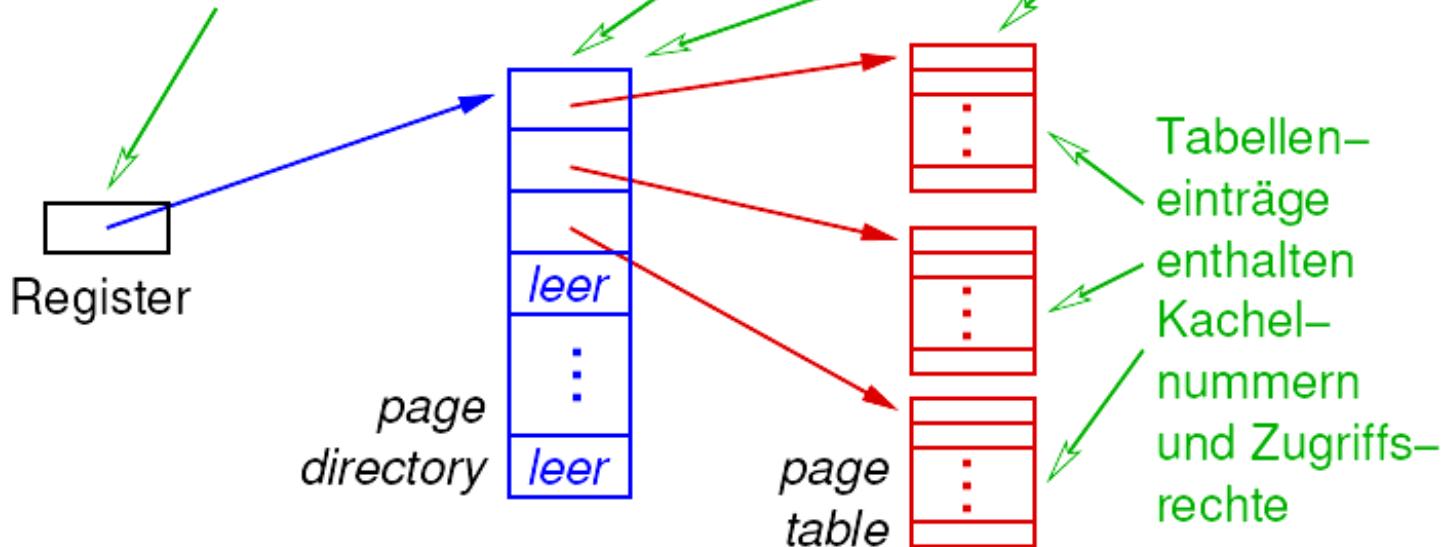
Paging: Seiten/Kachel - Tabellen

- Problem: Speicherplatzbedarf der Tabelle, falls für jede Seite des virtuellen Adressraums ein Eintrag vorgesehen wird
 - Z.B. bei virtuellen Adressen mit 32 Bit: 2^{20} Seiten à 4 Byte
 - Für jeden Prozess wäre eine 4 MByte große Tabelle nötig
 - Bei 64-Bit Adressen sogar 2^{52} Seiten, d.h. 9000 TByte!
- Aber: ein Prozess nutzt den virtuellen Adressraum i.a. nicht vollständig
 - Daher: mehrstufige Tabellen sinnvoll
 - Z.B. zweistufige Tabelle bei IA-32 (32-Bit Adressen)
 - 1. Stufe: page directory, 1 Kachel, 1024 Einträge
 - 2. Stufe: ≤ 1024 page tables, je 1 Kachel zu 1024 Einträgen

Paging: zweistufige Umsetzung

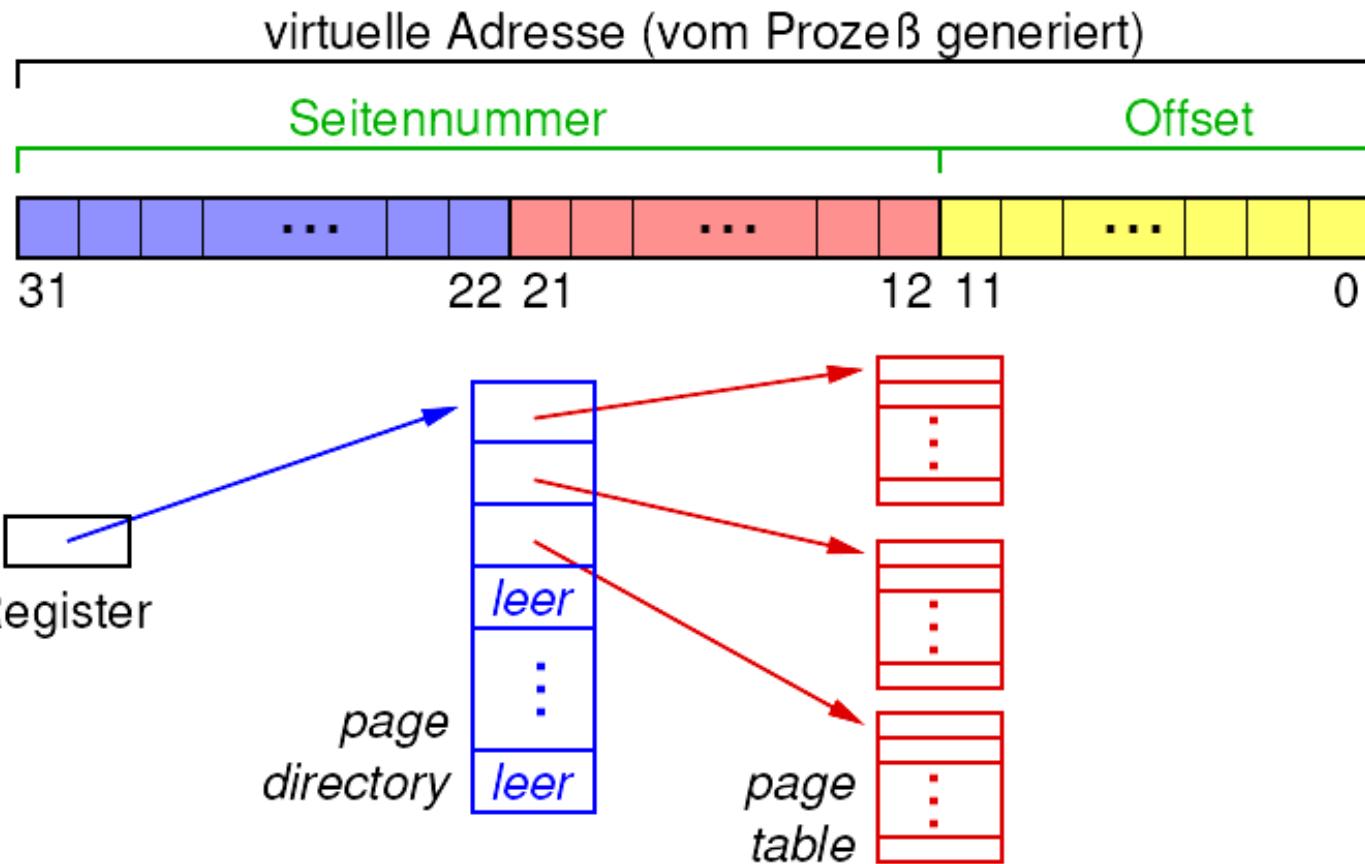
Paging: Zweistufige Seiten/Kacheltabellen ...

Spezialregister enthält
Adresse des *page
directory*
(wird bei Prozeß-
wechsel umgeladen)



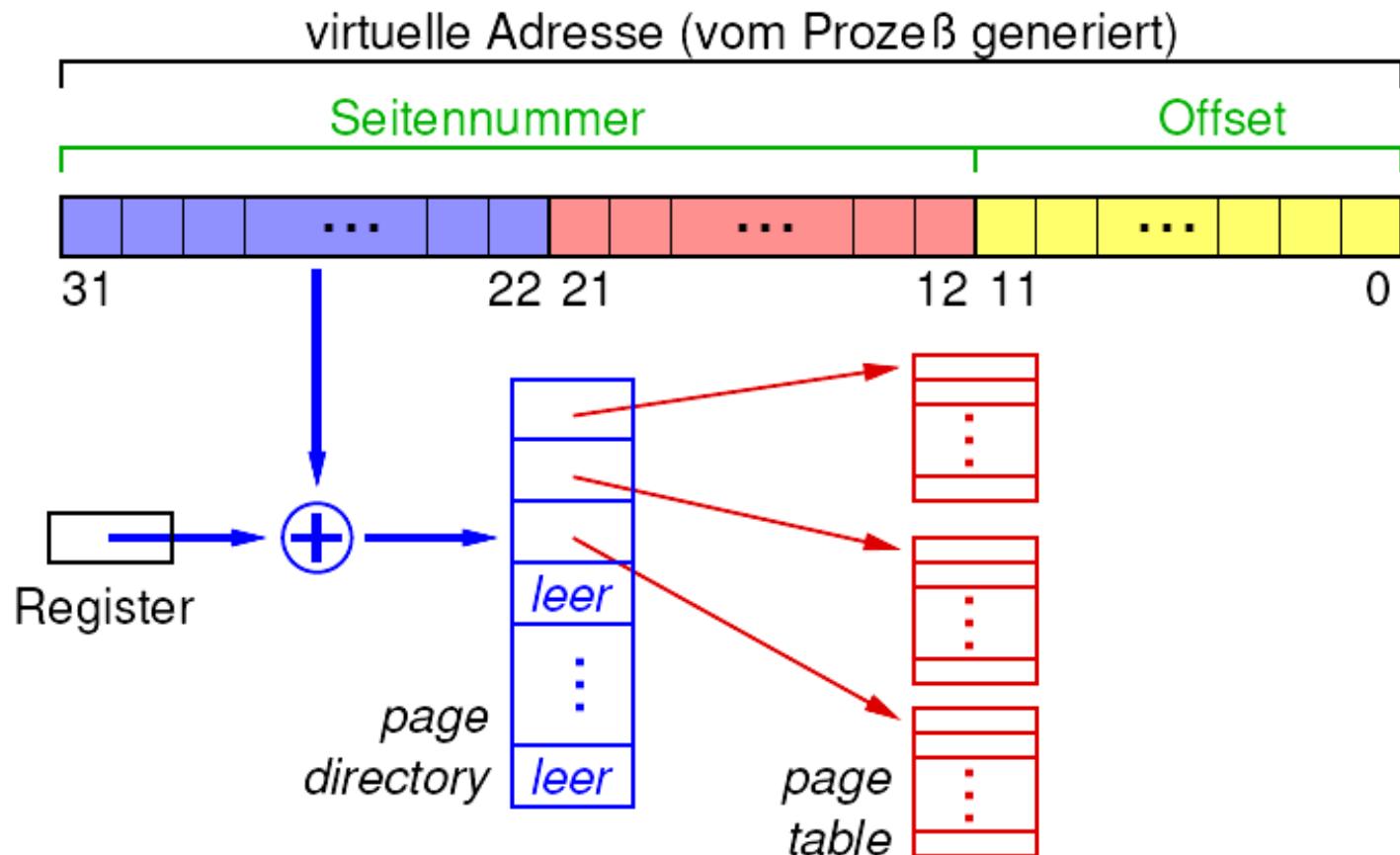
Paging: zweistufige Umsetzung

Paging: Zweistufige Seiten/Kacheltabellen ...



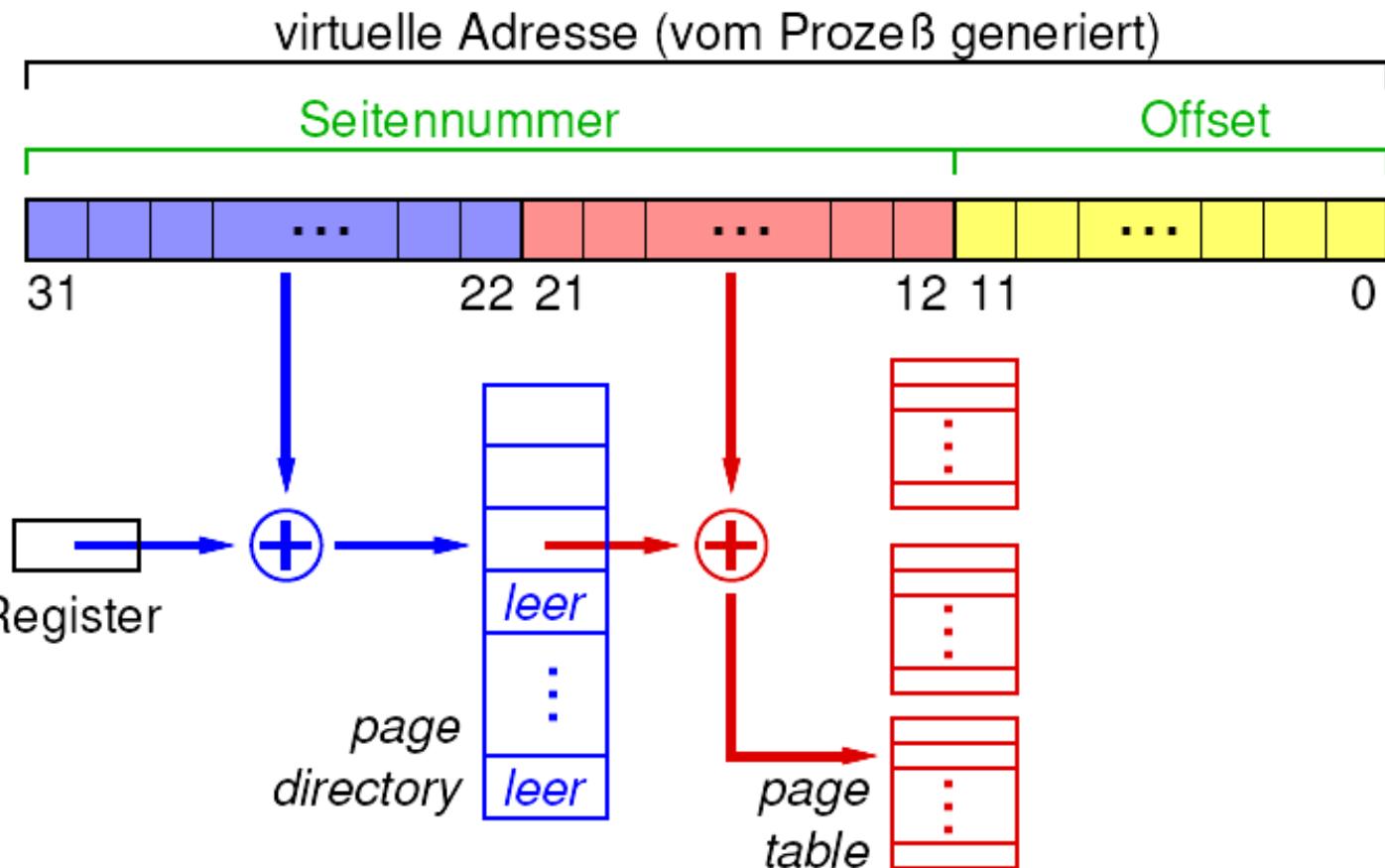
Paging: zweistufige Umsetzung

Paging: Zweistufige Seiten/Kacheltabellen ...



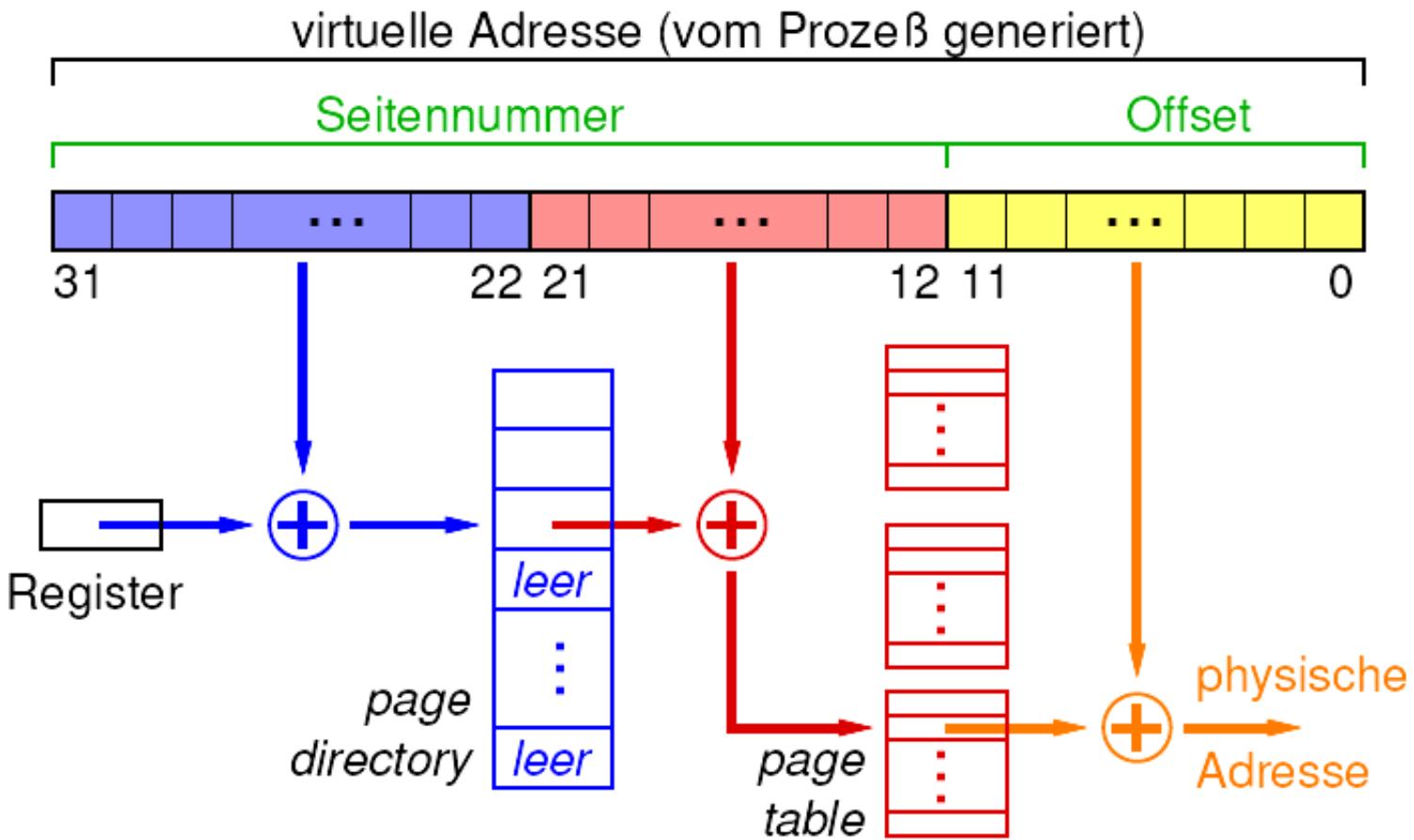
Paging: zweistufige Umsetzung

Paging: Zweistufige Seiten/Kacheltabellen ...



Paging: zweistufige Umsetzung

Paging: Zweistufige Seiten/Kacheltabellen ...

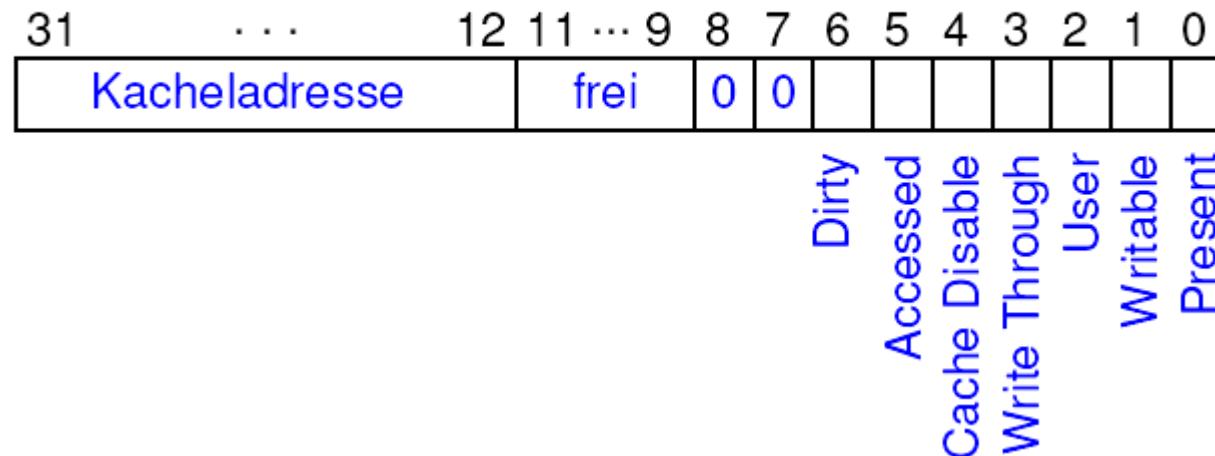


Paging: Aufbau eines Tabelleneintrags

- Zu jedem Tabelleneintrag werden noch bestimmte Attribute einer Seite gespeichert, die dem BS erlauben, Zugriffe auf einen Frame zu verifizieren.

Beispiel IA-32: Paging

Aufbau eines Tabelleneintrags (bei 4 KB Seiten):



Probleme der virtuellen Speicherverwaltung

Beim Austausch von Daten zwischen Arbeits- und Hintergundspeicher ergeben sich drei Problemkreise:

1. Der Einlagerungszeitpunkt

Wann werden Segmente oder Seiten in den Arbeitsspeicher eingelagert ?

Gängiges Verfahren:

Einlagerung auf Anforderung (Demand Paging bei Seitenverfahren)

Hierbei werden Daten eingelagert, sobald auf sie zugegriffen wird, sie sich aber nicht im Arbeitsspeicher befinden.

Der Zugriff auf ein nicht im Arbeitsspeicher vorhandenes Segment oder Seite heißt **Segment- oder Seiten-Fehler** (*segment fault, page fault*).

Probleme der virtuellen Speicherverwaltung

2. Das Zuweisungsproblem

An welche Stelle des Arbeitsspeichers werden die Seiten oder Segmente eingelagert ?

Bei Segmentierungsverfahren:

Hier muss eine ausreichend große Lücke im Arbeitsspeicher gefunden werden.

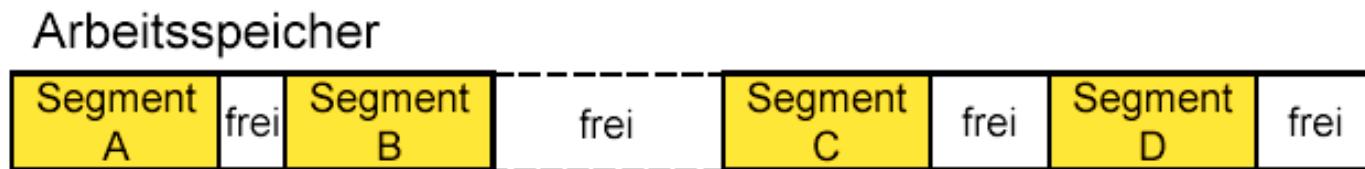
Drei Strategien:

- ⇒ *first-fit*: erste passende Lücke wird genommen
- ⇒ *best-fit* : kleinste passende Lücke wird genommen
- ⇒ *worst-fit*: größte passende Lücke wird genommen

Probleme der virtuellen Speicherverwaltung

Problem bei allen drei Verfahren:

Der Speicher zerfällt nach einiger Zeit in belegte und unbelegte Speicherbereiche ➔ externe Fragmentierung.



Die unbelegten Speicherbereiche sind hierbei oft zu klein, um Segmente aufnehmen zu können

Probleme der virtuellen Speicherverwaltung

Zuweisungsproblem bei Seitenwechselverfahren

Hier taucht dieses Problem nicht auf, da alle Seiten gleich groß sind und somit immer “passende Lücken” entstehen ➔ keine externe Fragmentierung.

Jedoch: Problem der **internen Fragmentierung**

Diese entsteht bei der Aufteilung eines Programms auf die Seiten.

Einheitliche Seitengröße ➔ auf der letzten Seite jedes Programm-Moduls entsteht mit hoher Wahrscheinlichkeit ein ungenutzter Leerraum.

Probleme der virtuellen Speicherverwaltung

3. Das Ersetzungsproblem

Welche Segmente oder Seiten müssen ausgelagert werden, um Platz für neu benötigte Daten zu schaffen ?

Bei Segmentierungsverfahren:

Meist wird die Anzahl der gleichzeitig von einem Prozeß benutzbaren Segmente limitiert:

- bei Einlagerung eines neuen Segments wird ein zuvor für diesen Prozeß benutztes Segment ausgelagert

Es ist jedoch auch eine der im folgenden für Seitenwechsel-Verfahren beschriebenen Methoden möglich

Probleme der virtuellen Speicherverwaltung

Ersetzungsproblem bei Seitenwechselverfahren

3 gängigste Strategien zum Ersetzen einer Seite:

- ⇒ **FIFO (*first-in-first-out*)**: die sich am längsten im Arbeitsspeicher befindende Seite wird ersetzt
- ⇒ **LIFO (*last-in-first-out*)**: die zuletzt eingelagerte Seite wird ersetzt
- ⇒ **LRU (*least recently used*)**: die Seite, auf die am längsten nicht zugegriffen wurde, wird ersetzt

Daneben werden bevorzugt solche Seiten ersetzt, die nicht verändert wurden → kein Rückschreiben der geänderten Seite erforderlich.

Beschleunigung der Adressberechnung

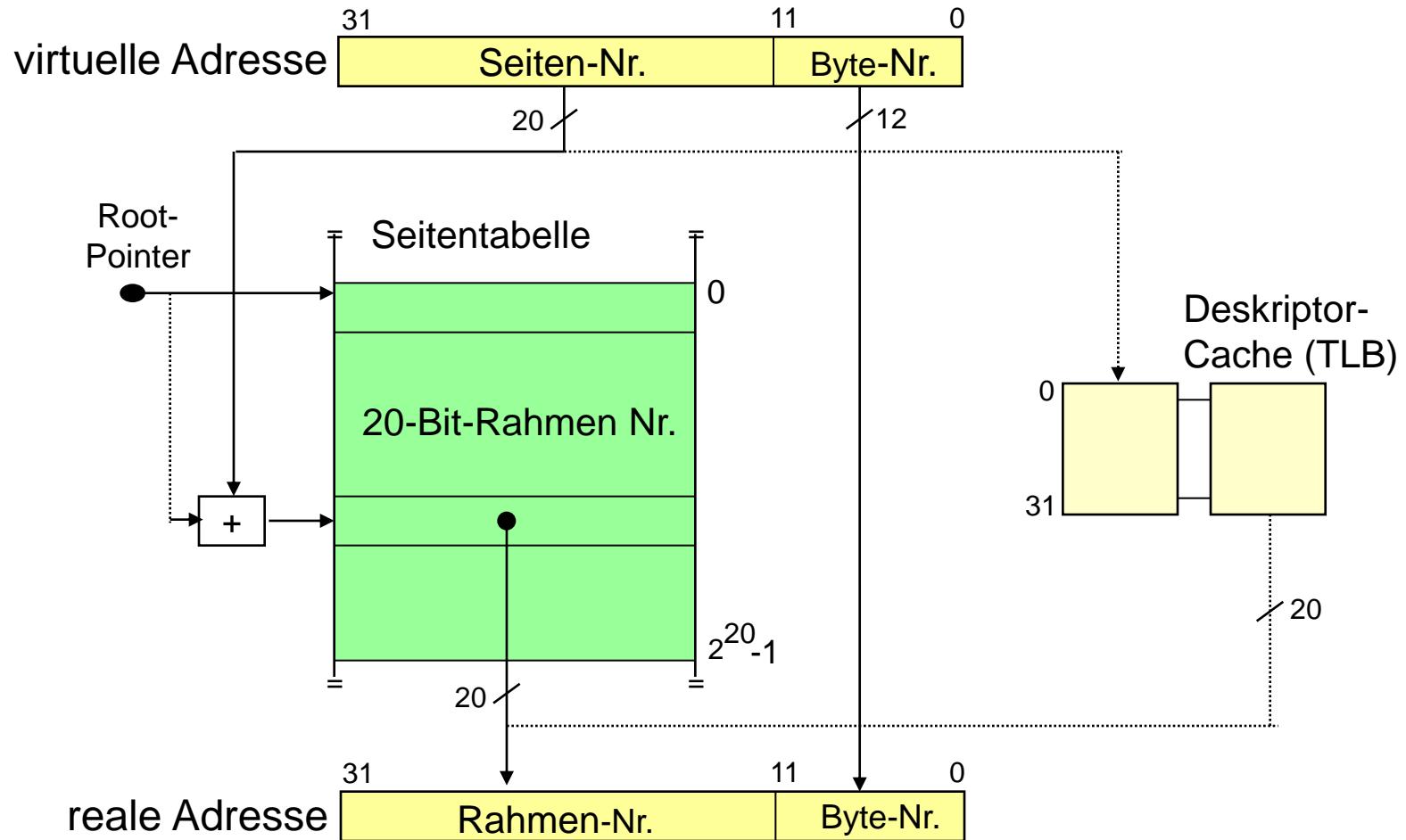
Beschleunigung der Adressberechnung durch einen Cache

Fallbeispiel Intel 80386:

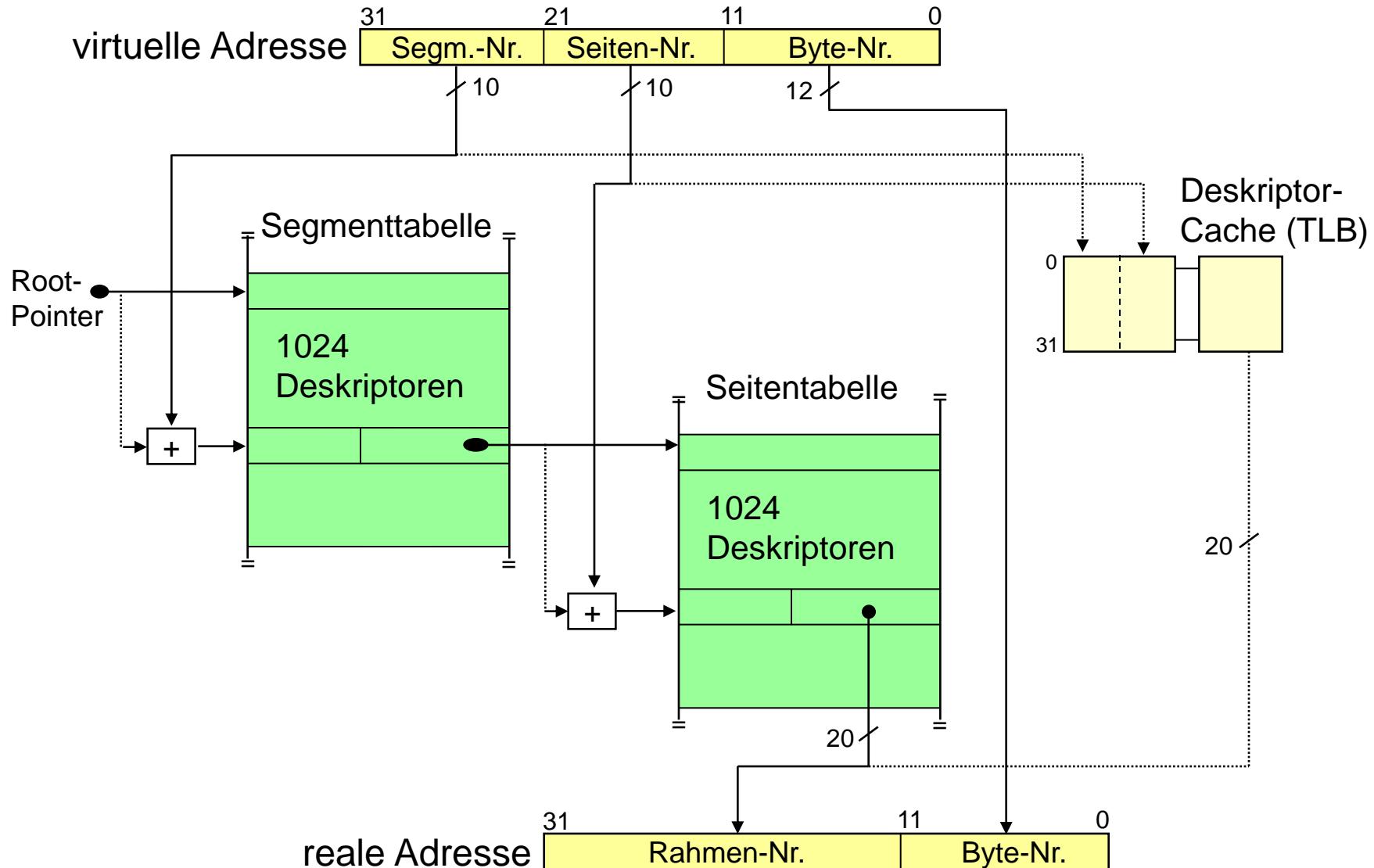
Ein schneller voll-assoziativer Cache (*Translation Lookaside Buffer, TBL*) speichert automatisch die 32 zuletzt benutzen Einträge aus Seitentabellenverzeichnis und Seitentabellen

- ➔ Im Trefferfall (Trefferquote ca. 90 %) muß nicht auf die im Hauptspeicher liegenden Tabellen zugegriffen werden

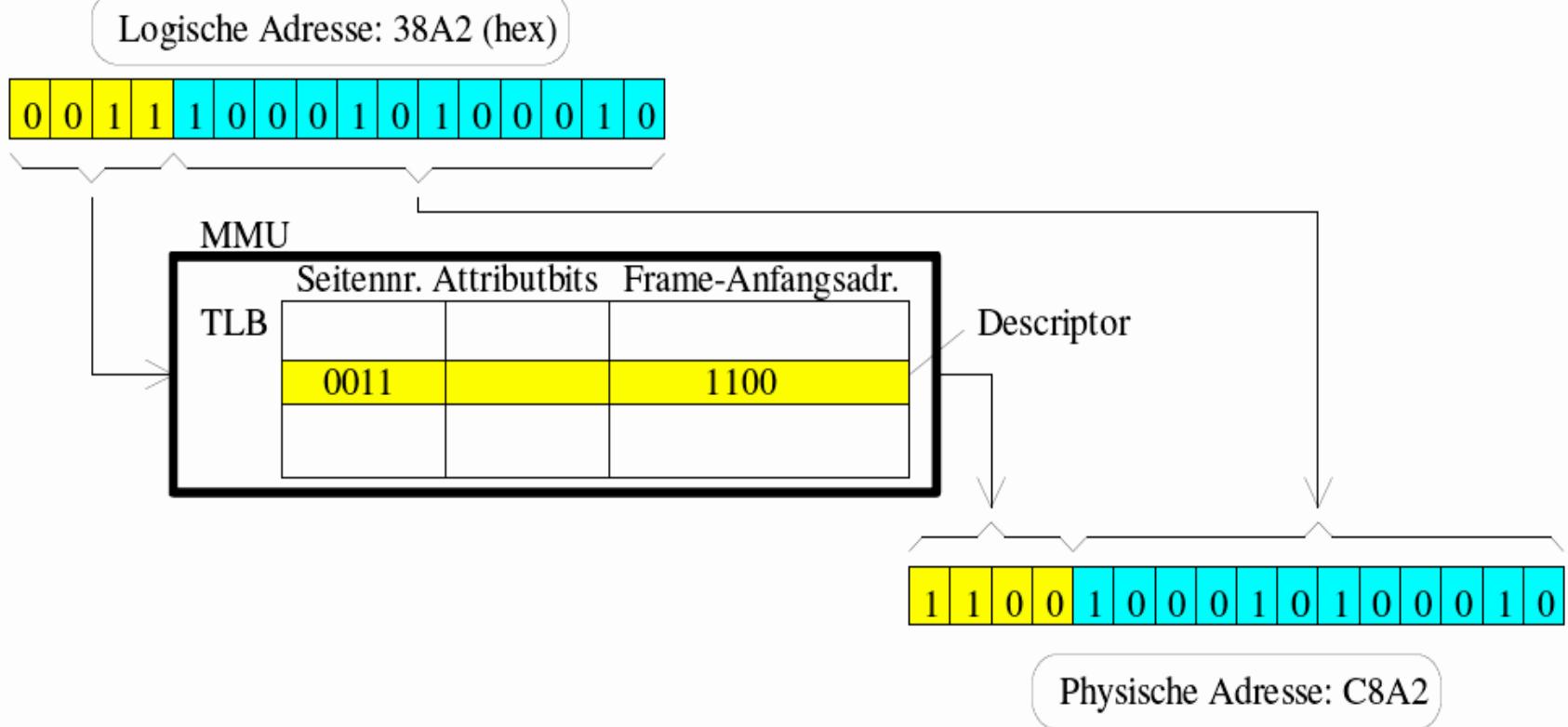
Beschleunigung der Adressberechnung



Zweistufige Adressumsetzung: Segment- und Seitenverwaltung



Adressumsetzung durch MMU



Anmerkungen

Sowohl bei segmentorientierter wie bei seitenorientierter Speicherverwaltung gilt:

Befindet sich eine Seite oder ein Segment nicht im Hauptspeicher, so löst der Prozessor eine Unterbrechung aus, um die Seite oder das Segment durch das Betriebssystem zu laden
(Seiten- oder Segmentfehler)

Erkennung eines Segmentfehlers:

Bit im Segment-Deskriptor zeigt an, ob das Segment im Hauptspeicher ist oder nicht

Erkennung eines Seitenfehlers:

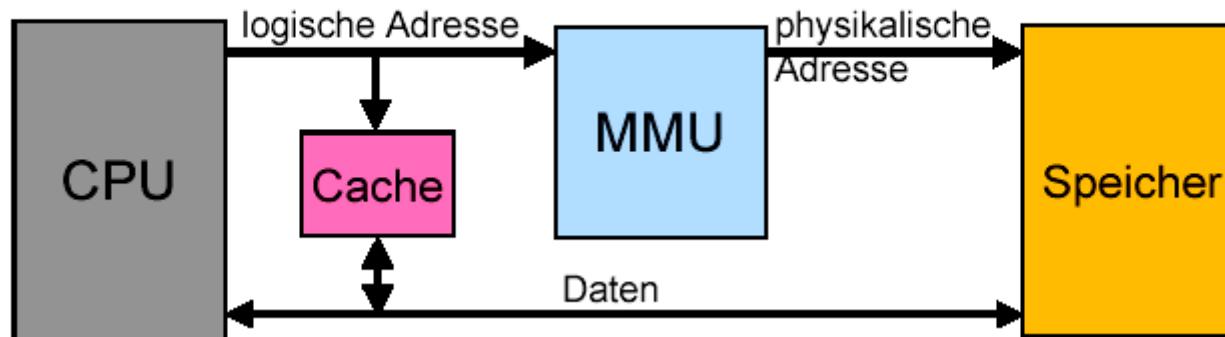
Seitennummer befindet sich nicht in der Seitentabelle
(Seitenfehler)

Cache und Speicherverwaltungseinheit

Zwei Möglichkeiten der Cache-Einbindung bei virtueller Speicherverwaltung:

- **Virtueller Cache:**

wird zwischen CPU und MMU gelegt. Die höherwertigen Bits der logischen Adressen als Tags abgelegt.



Virtueller Cache

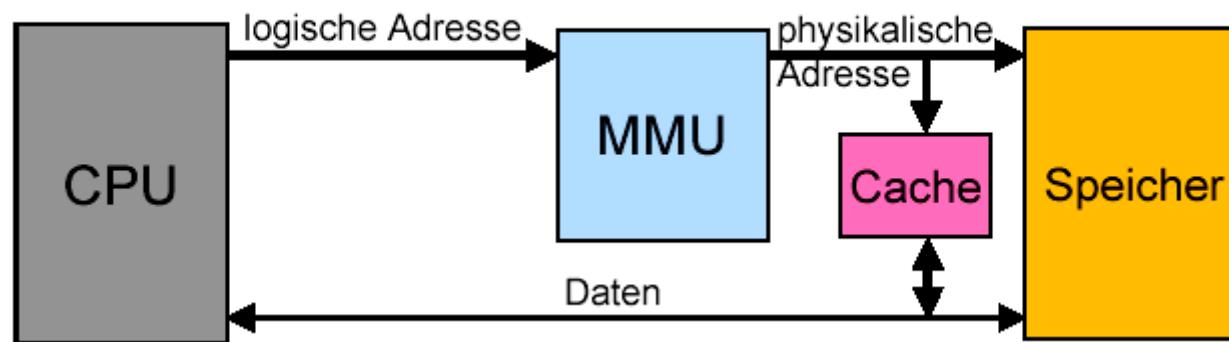
- Cache arbeitet mit virtuellen, vom Prozessor ausgegebenen Adressen
- Vorteil
 - Bei Treffern wird die MMU nicht benötigt
- Probleme
 - Kohärenzprobleme bei mehreren Mastern
 - Shared Data-Bereiche: Snoop-Logik kann Treffer nicht detektieren, da z.B. DMA-Controller mit realen Adressen arbeitet, aber Cache mit virtuellen.
 - Abhilfe: Shared Data ist non-cacheable
 - Task Wechsel beim Multitasking
 - Virtuelle Adressen überdecken sich
 - Abhilfe: Cache-clear, cache-flush-Operation bei Taskwechsel oder Tag-Speicher um Task-ID erweitern

Cache und Speicherverwaltungseinheit

□ Physischer Cache

wird zwischen MMU und Speicher gelegt.

Die höherwertigen Bits der physikalischen Adressen werden als Tags abgelegt



Physischer Cache

- Cache-Adressierung kann erst erfolgen, wenn MMU Adressabbildung durchgeführt hat.
- **Abhilfe**
 - Bei On-Chip MMU's wird Adressumsetzung bei Fließbandverarbeitung im Vorgriff durchgeführt.
 - Bei direct-mapped und n-fach satzassoziativen Caches wird die durch den Zeilen- oder Satz-Index erfolgte Cache-Anwahl parallel zur Adressumsetzung der MMU durchgeführt.
- **Probleme**
 - Kohärenzprobleme bei mehreren Mastern lassen sich mit den gängigen Verfahren lösen.
 - Probleme beim Taskwechsel treten nicht auf.