

Rechnerarchitektur I

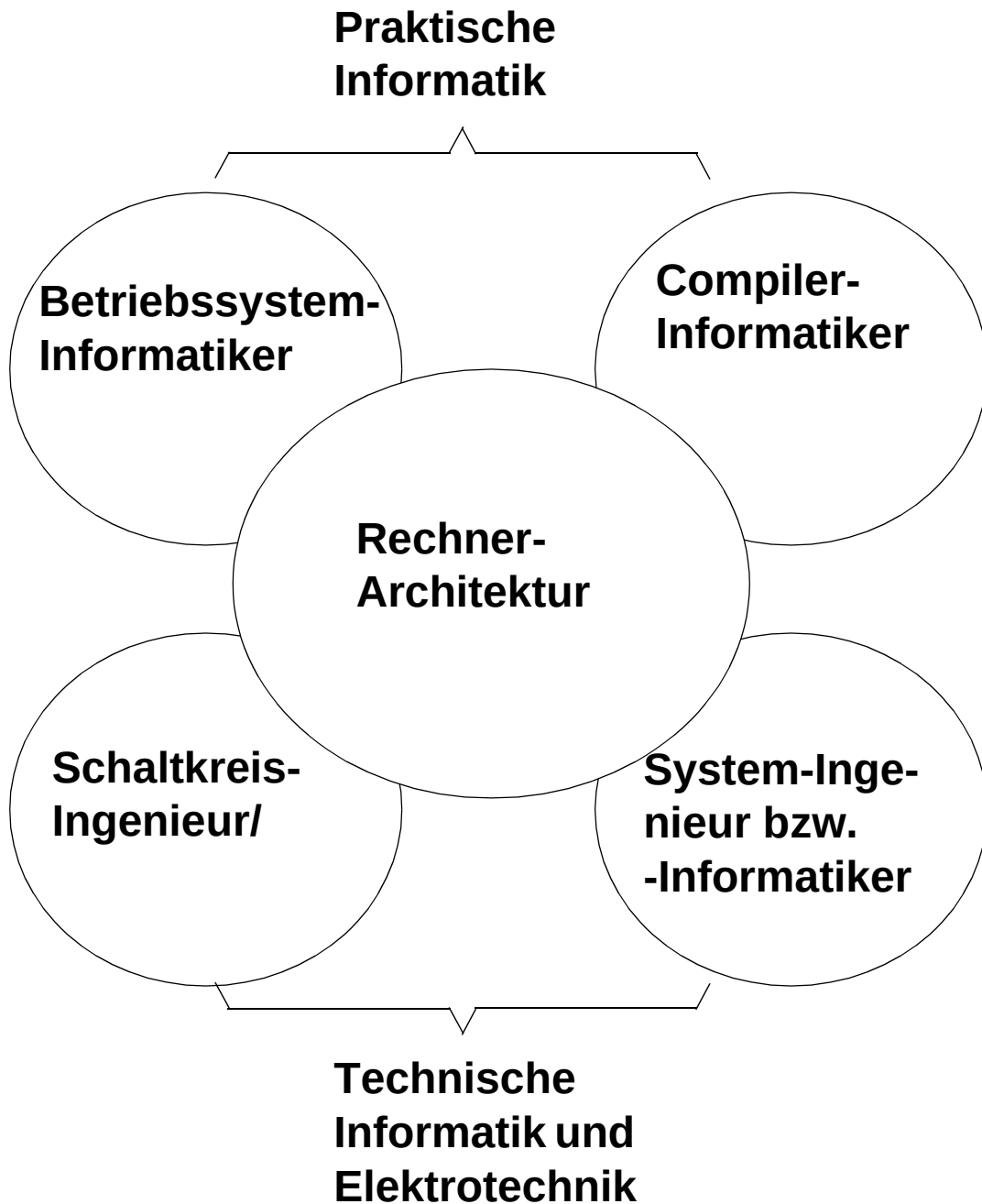
Einleitung und Überblick

Definition der Rechnerarchitektur

- Aus R. Sites, *Alpha Architecture Reference Manual*, Digital Equipment Corporation, Maynard, Massachusetts, 1992:
- „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
- Implementation is defined as the actual hardware structure, logic design, and data-path organization of a particular embodiment of the architecture
- Thus, the architecture is a document that describes the behavior of all possible implementations
- An implementation is typically a single computer chip“

to compute: rechnen

Wer beschäftigt sich mit Rechnerarchitektur?



Vereine und Verbände für Rechnerarchitekturen

- In Deutschland: Gesellschaft für Informatik (GI); international: IFIP, IEEE und ACM
- IFIP International Federation of Information Processing (gegründet 1945!)
- IEEE Computer Society, Sitz in Los Alamitos, <http://www.ieee.org>
- ACM Association for Computing Machinery; Sitz in New York, <http://www-acs.ucsd.edu/~acm>
- Es gibt in diesen großen Vereinen Unterabteilung speziell für Rechnerarchitektur, z.B.:
- GI-Fachbereich "Technische Informatik und Architektur von Rechensystemen"
http://www.informatik.unibw-muenchen.de/GI_FB3
- Special Interest Group on Computer Architecture der ACM (SIGARCH)
- Darüberhinaus gibt es sehr viele Web-Seiten zu Rechnerarchitektur, z.B.:

Computer architecture Homepage der Universität von Wisconsin (<http://www.cs.wisc.edu/~arch/www>)

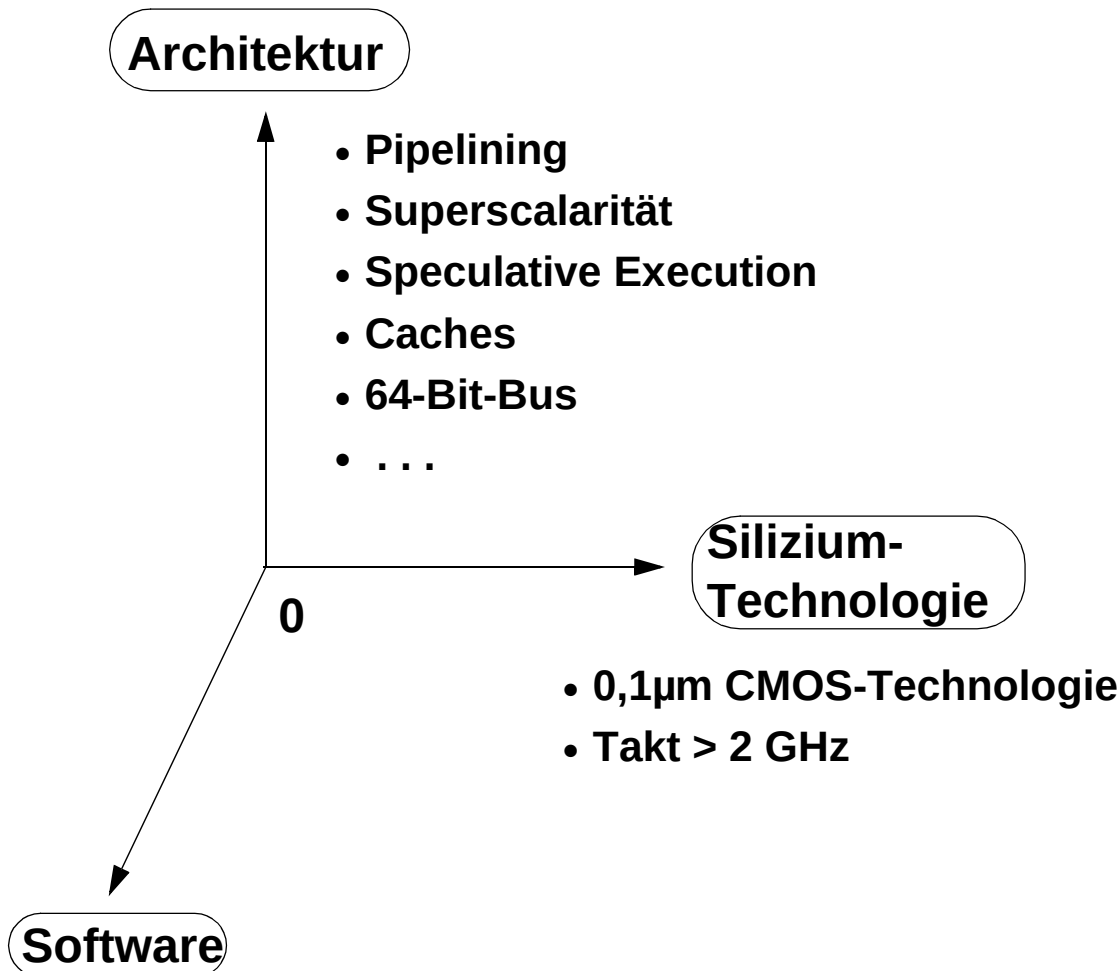
Vielfalt der Rechnerarchitekturen

Computertypen - Überblick

- **Personal Computer PC (= persönlicher Rechner)**
- **Workstation WS (Arbeitsplatzrechner)**
- **Server-/ Client-“Architekturen“**
- **Mainframes**
- **Supercomputer**
- **Vektorrechner**
- **Minicomputer z. B. PDP 11, VAX 11-780]**
- **Parallelrechner**
- **Multiprozessorsystem, Multirechnersystem**
- **Laptop (Lap = Schoß)**
- **Notebook, -pad**
- **Personal Communicator, Personal Digital Assistant (PDA)**
- **Workstation Cluser (WSC), Network-Of-Workstations (NOWs) und Client-Server-Computing**
- **Netzcomputer (NC)**
- **Metacomputer**
- **Embedded Computer**
- **diverse Spezialrechner z. B. für Echtzeit, Signalverarbeitung usw.**
- **Handys**

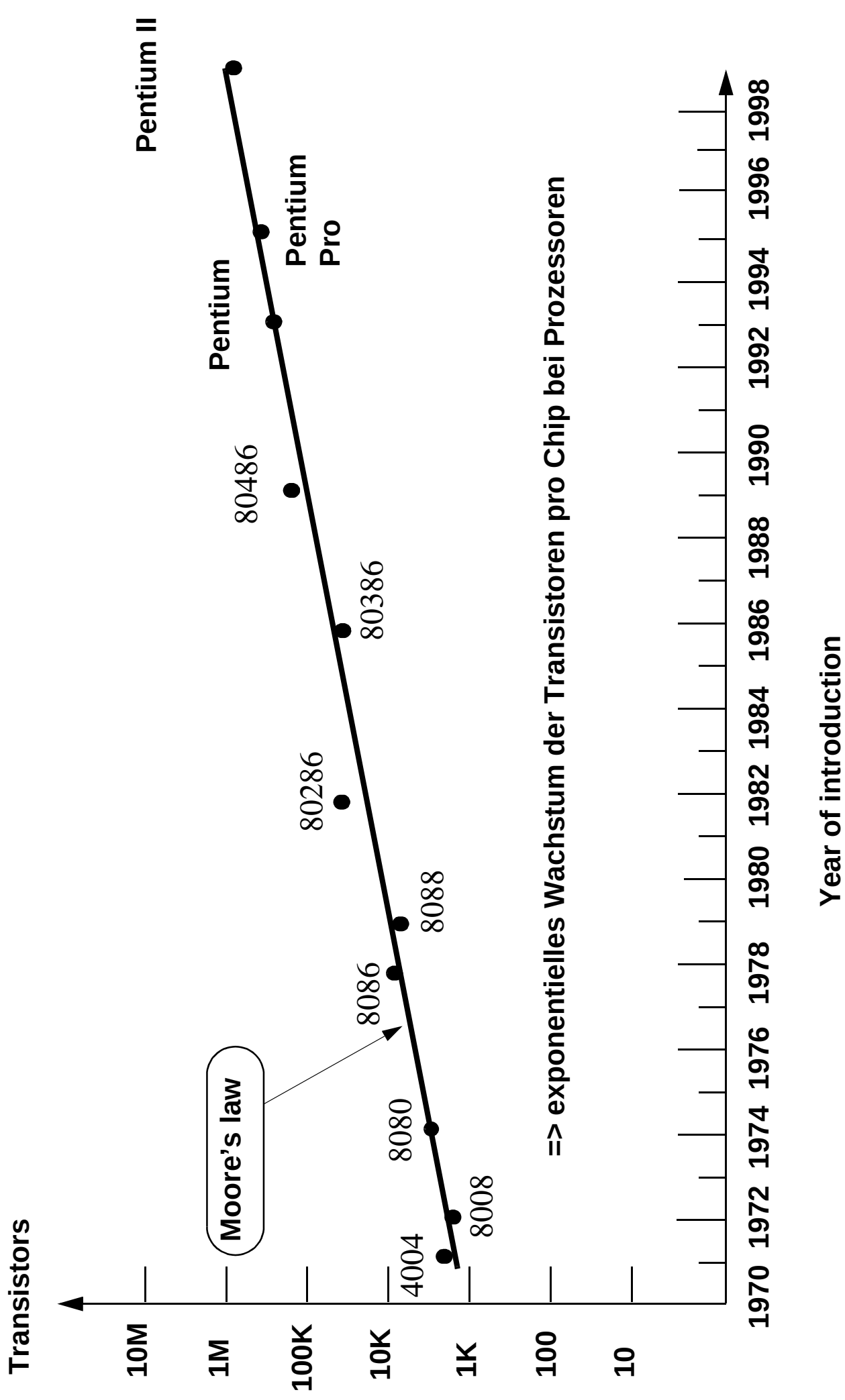
personal = persönlich, d. h. für einen selbst
personnel = Personal

Welche Komponenten erbringen die Leistung in einem Rechner?



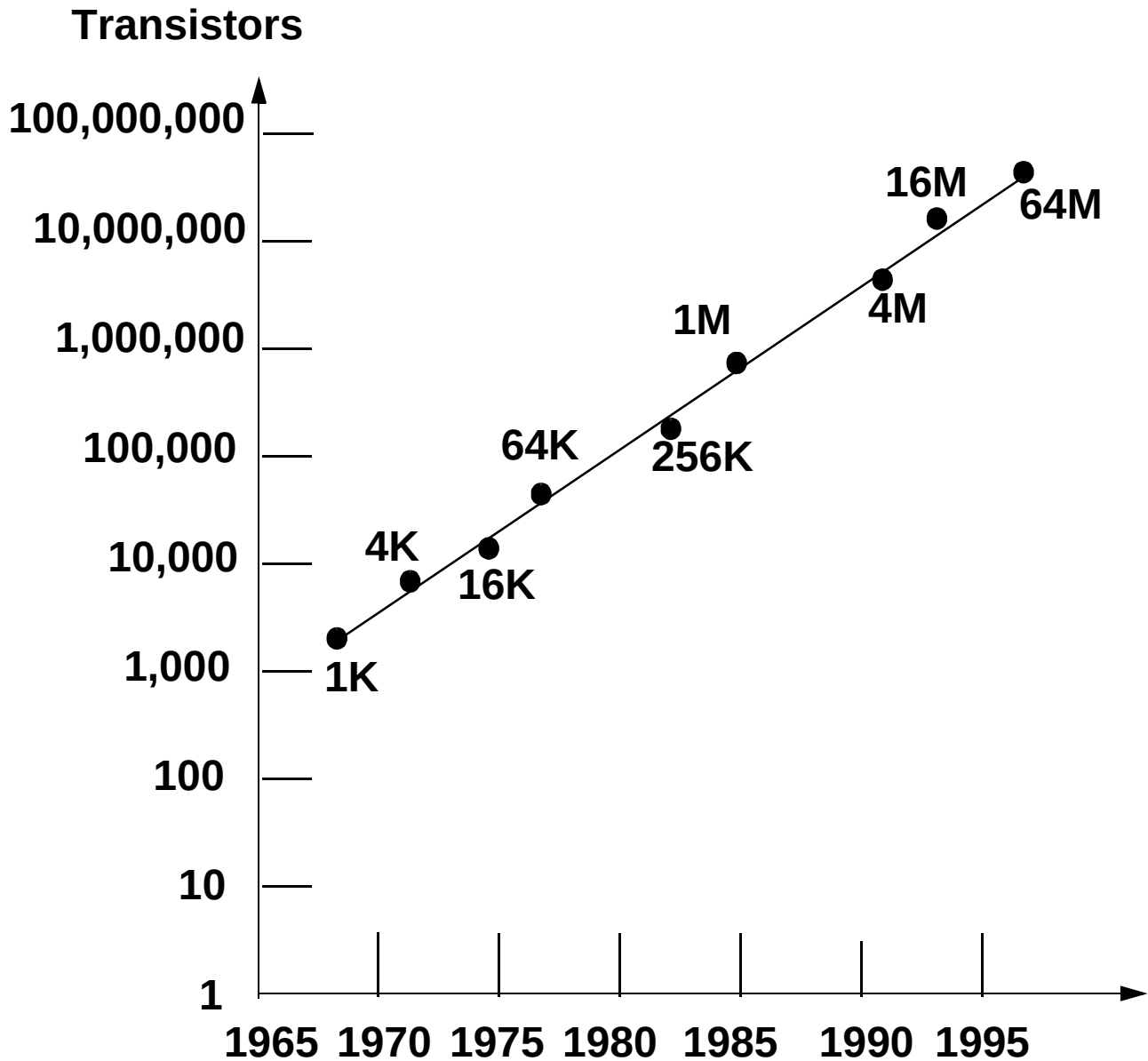
- sog. optimierende Compiler = optimiertes Zusammenspiel zwischen Compiler und Prozessor bei der Programmausführung
- Hardware/Software-Codesign = Berücksichtigung der Verzahnung zwischen Prozessor und späteren Basisbetriebsprogrammen (z.B. „BIOS“) bei der Entwicklung neuer Prozessoren („Was in HW- Was in SW?“)

Zahl der Transistoren bei Mikroprozessoren



Fortschritte in der Silizium-Technologie

Zahl der Transistoren bei Speichern



=> exponentielles Wachstum der Transistoren
pro Chip bei Speichern

CPU – Entwicklung: Aufwand u. Produktivität

Jahr	Projekt	Komplexität [Transistoren]	Entwickler [Anzahl]	Dauer [Monate]	Produktivität [Transistoren pro Entwickler]
1971	Intel 4004	$1 \cdot 10^3$	4	9	28
1982	Intel 286	$100 \cdot 10^3$	20	12	417
1992	Intel Pen- tium	$3 \cdot 10^6$	100	20	1500
2002	geschätzt:	$150 \cdot 10^6$	1000	40	3750

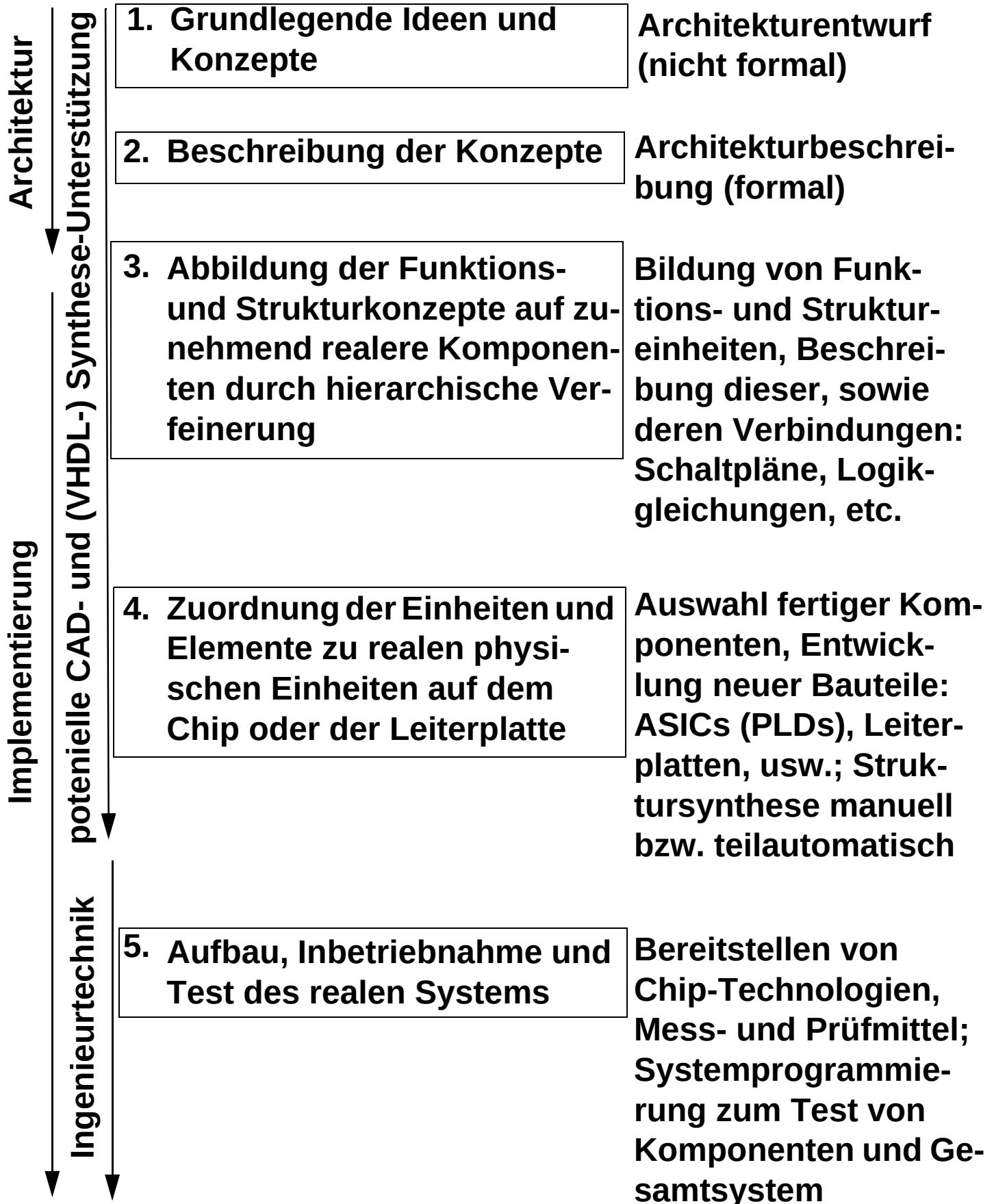
(aus K. T. Hagen: Abstrakte Modellierung digitaler Schaltungen,
Springer Verlag Berlin, 1995)

=> exponentielles Wachstum der Produktivität bei der
Chip-Entwicklung

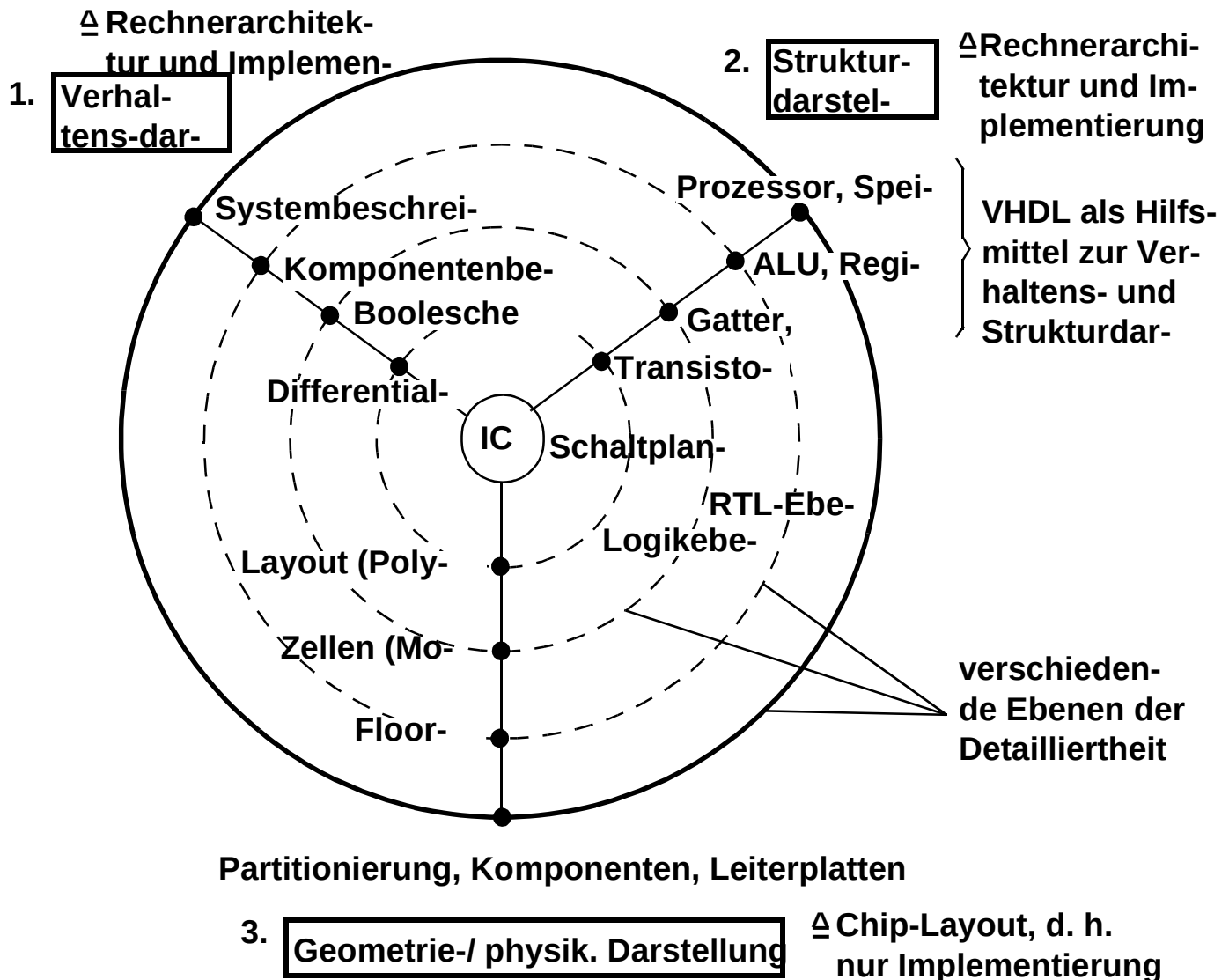
Beachte:

Viele 16-Bit-Prozessoren, z. B. Motorola 68000 und Zilog
Z8000 wurden noch vollständig ohne Simulation entworfen!

Phasen des Rechnerentwurfs



Sichtweisen beim Rechnerentwurf (Y-Diagramm)



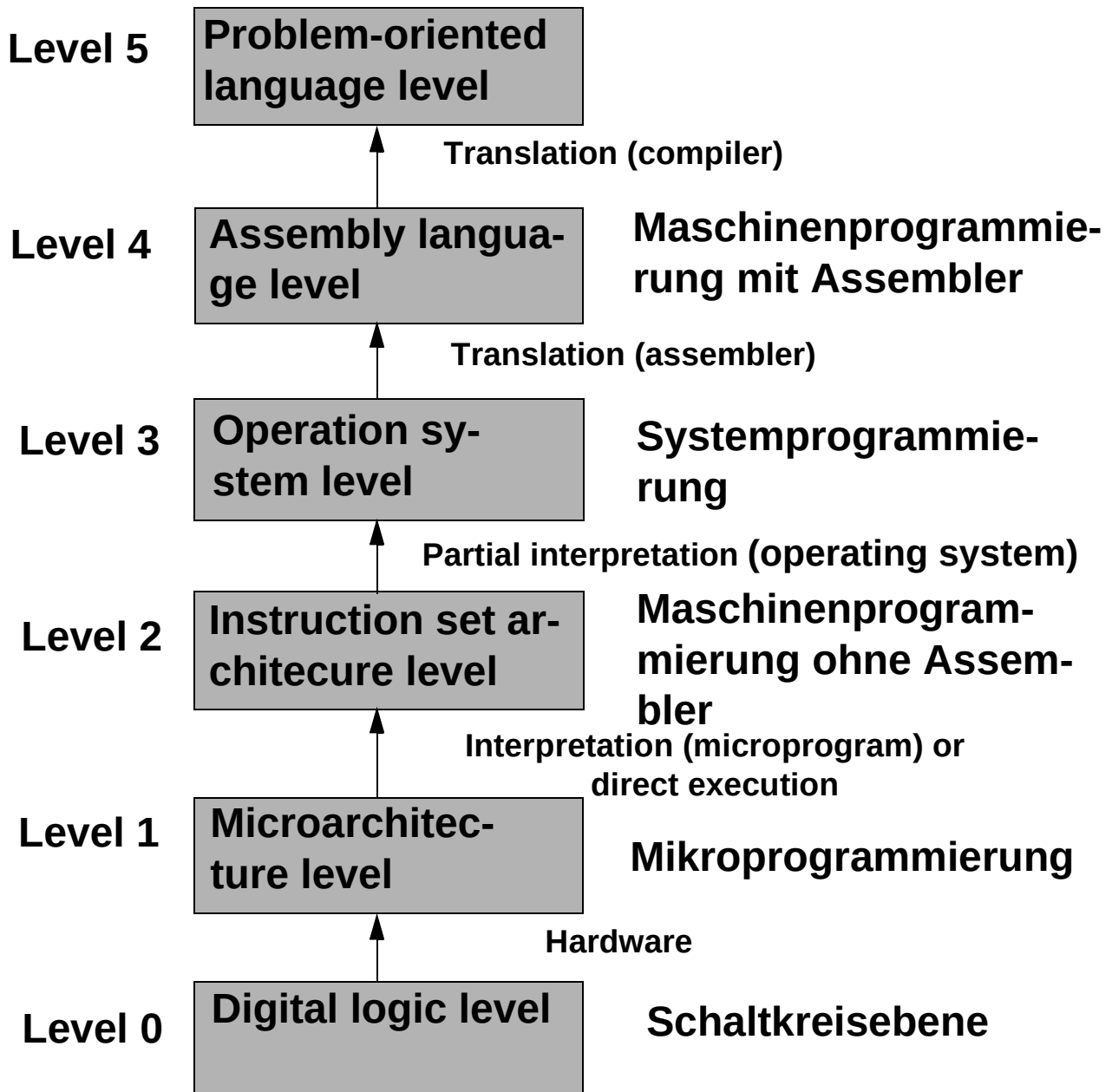
IC = Integrated circuit
 = Rechner-Chip
 RTL = Register-Transfer-Level
 VHDL = VLSI Hardware Description Language

Nach welchen Prinzipien können Rechner funktionieren?

- Es gibt ganz unterschiedliche Prinzipien
- Sie erlauben ebenso unterschiedliche Unterteilungen in einzelne Architekturklassen, z.B.:
 - Unterteilung nach der Art der Programmierung und der Art der Ausführung der Programme
 - von Neuman-Rechner: Steuerfluß-orientiert, d.h. arbeitet Algorithmen schrittweise ab („prozedural“ bzw. „imperativ“, „Tu Dies, Tu das“)
 - Datenfluß-orientierte Rechner, d.h. es geht immer dann von selber weiter, wenn alle Operanden für eine Operation „da“ sind (sog. Zündregeln ähnlich wie bei Petrinetzen)
 - Rechner, die eine funktionale Programmiersprache direkt in Hardware umsetzen (Funktionsaufrufe und Reduktionsregeln in Hardware)
 - Rechner, die eine deklarative Programmiersprache direkt in Hardware umsetzen; für die Programmierung wird nicht die Prozedur zur Erreichung der Lösung beschrieben, sondern es werden nur die Eigenschaften der Lösung mittels Prädikatenlogik beschrieben

- **Unterteilung nach der Zahl der pro Schritt verarbeitbaren Daten**
 - einzelne Skalare bzw. einzelne Buchstaben
 - Vektoren (Vektorrechner) bzw. Strings
 - mehrdimensionale Felder (Systolische Rechner) bzw. ganze Records
- **Unterteilung nach den physikalischen Prinzipien**
 - Assoziativ (wie bei einer Datenbank sowie grob ähnlich zum Gehirn)
 - Neuronal (grob ähnlich zum Gehirn)
 - Optisch (Photonen statt Elektronen)
 - Biologisch (DNA-Computing)
 - Molekular (Nano-Computing)
 - Quanten-Bits; QBits sind nicht nur in einem von zwei Zuständen, sondern gleichzeitig in unendlich vielen Zuständen (Quanten-Computing)

Ebenen in einem Rechner



to compile: zusammenstellen
to translate: übersetzen
to assemble: montieren

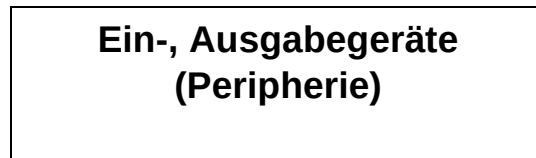
Der sog. von Neumann-Rechner (v. Neumann, A. Burks, H. Goldstine)

Der von-Neumann-Rechner hatte mehrere Vorläufer z.B.:

- **C. Babbage, Difference Engine, Großbritannien 1833, rein mechanisch (leider nicht funktionsfähig), programmierbar über Lochkarten, Ausgabe über Drucker !!**
- **Konrad Zuse, Z3, Deutschland 1941, elektromechanisch über Relais, binär, teil-programmierbar in der Sprache „Plankalkül“ (keine Sprünge)**
- **J. Atanasoff, USA 1942, elektronisch über Röhren, binär, Programm fest verdrahtet zum Lösen von Differentialgleichungen**
- **A. Turing, Colossus, Großbritannien, 1943, elektronisch über Röhren, Spezialrechner zum Knacken von Codes**
- **H. Aiken, Mark I, USA, 1944, elektronisch über Röhren, dezimal, für die Berechnung von Flugbahnen von Kanonenkugeln, programmierbar über Kabel, die auf einem Steckbrett hin- und hergesteckt werden**
- **K. Zuse, Z4, Deutschland 1945, elektromechanisch über Relais, binär**
- **Eckert, Mauchly, ENIAC, USA 1946, elektronisch über Röhren, programmierbar über Kabel auf Steckbrett**
- **von-Neumann, EDVAC, USA 1944-52, binär, Programme und Daten werden kodiert in einem Speicher abgelegt**

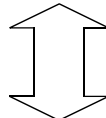
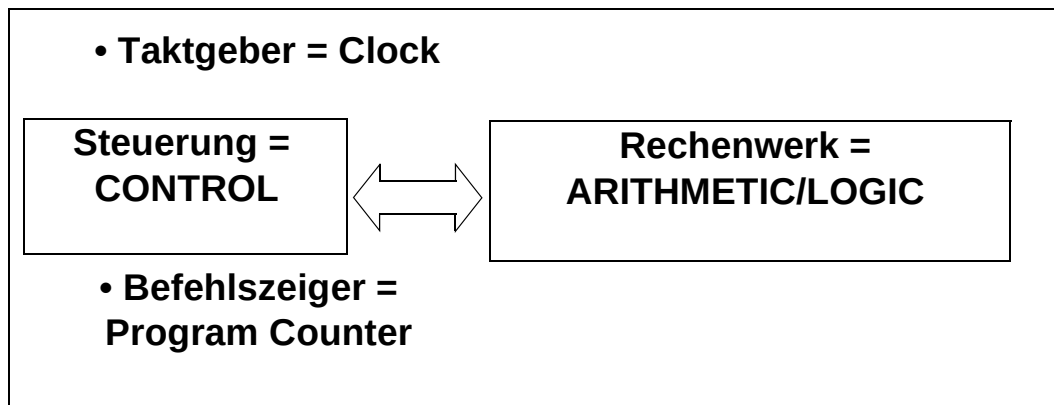
Die drei Hardware-Komponenten der von Neumann-Maschine

1.) Schnittstelle zum Bediener und der Umwelt



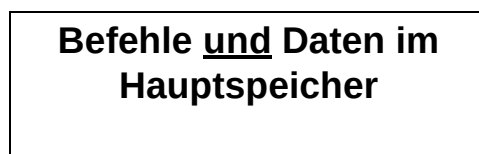
CPU liest und schreibt

2.) Zentral-einheit (CPU)



CPU liest und schreibt

3.) Speicher



**CPU = Central Processing Unit = Zentraleinheit
= Rechnerkern**

Control = Control Unit = Steuereinheit = Steuerwerk

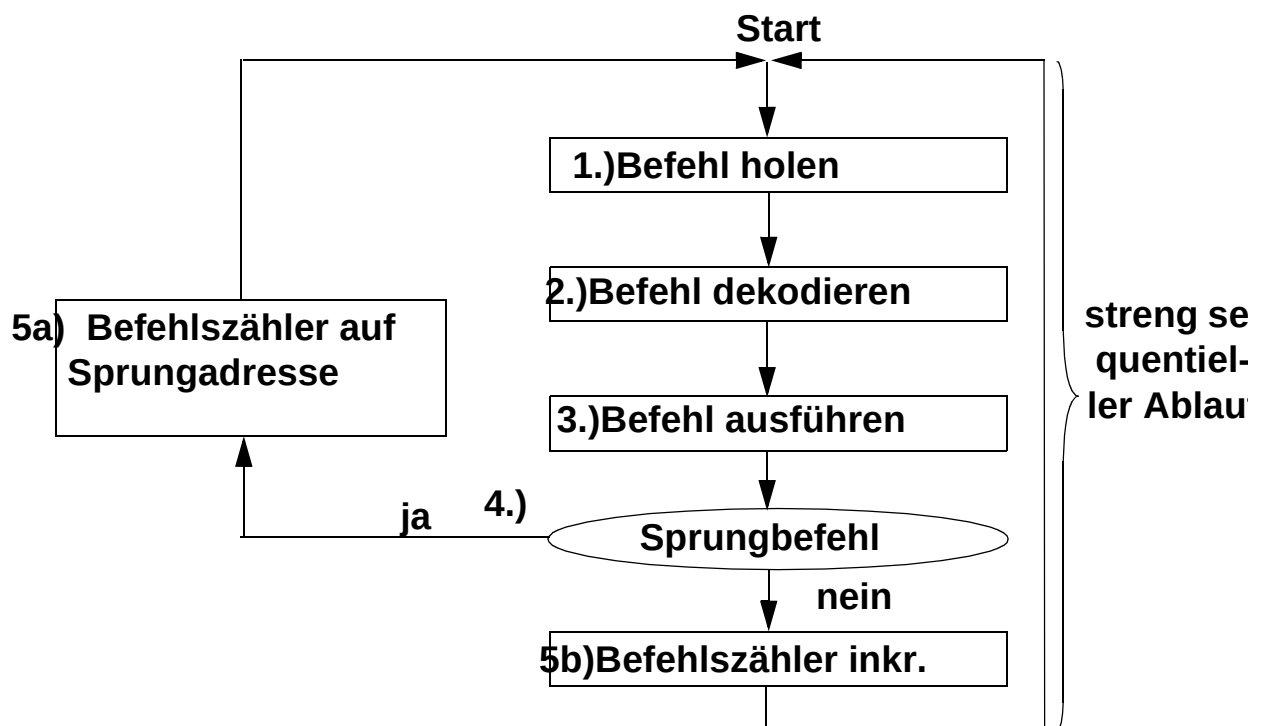
**Arithmetic/Logic = Arithmetic/Logical Unit = ALU = Re-
chenwerk**

Befehlszeiger = Program Counter

Wie arbeitet die von Neumann-Maschine?

Prinzip: Zentrale Befehlsschleife

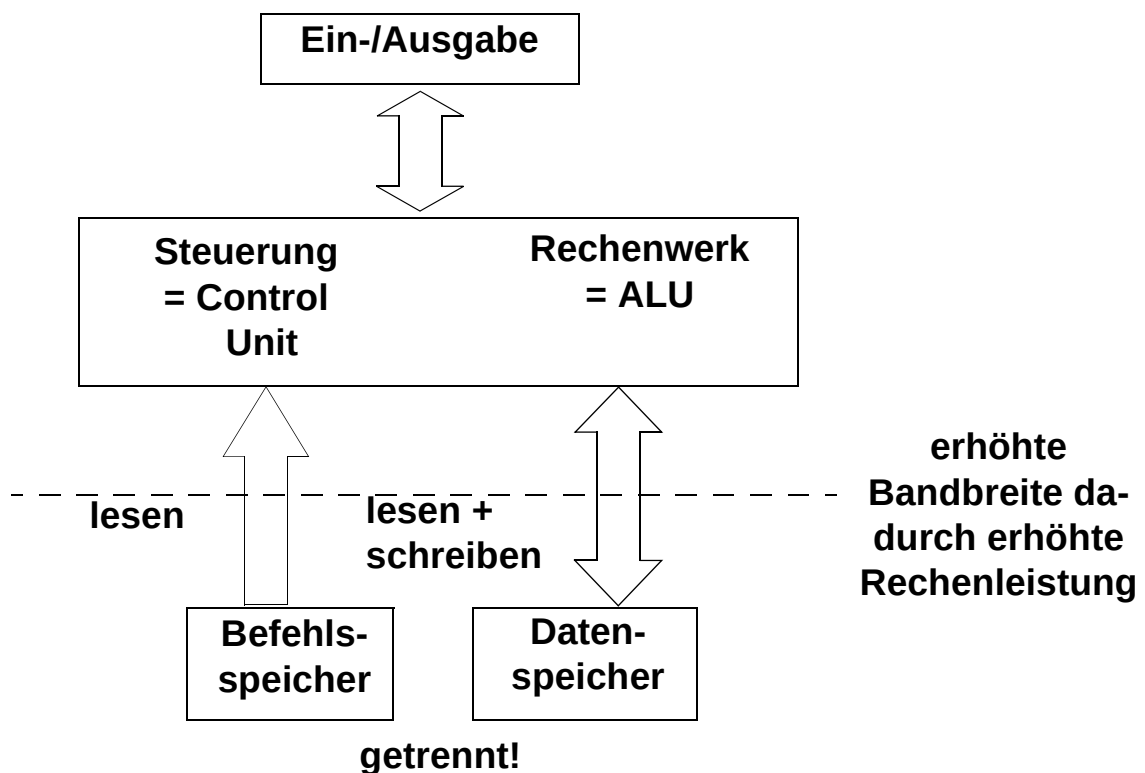
- Die zentrale Befehlsschleife implementiert einen seriellen, schrittweisen Ablauf bei der Programmausführung
- Sie verhindert, daß Dinge parallel bearbeitet werden



- Dazu kommt ein “datenabhängiger Sprung“ (bedingter Sprung), er ist grundlegend für die Universalität der Maschine. Nur dadurch ist die von Neumann-Maschine so berechnungsstark wie die Turing-Maschine
- Turing-Maschine = theoretische Modell eines Computers (Alan Turing 1936). Hat hohe Bedeutung für die Berechenbarkeits- und Komplexitätstheorie.

Alternative zum von Neumann-Rechner z. B. Harvard-Architektur

- Befehle- und Daten sind getrennt in verschiedenen Speichern untergebracht
- Befehle und Daten können gleichzeitig geholt werden.
- Erhöhter Aufwand, nur in sog. Echtzeitanwendungen gerechtfertigt
- Beispiel: „Signalprozessor“ TI 320C40 (TI = Texas Instruments = Prozessor-Hersteller)



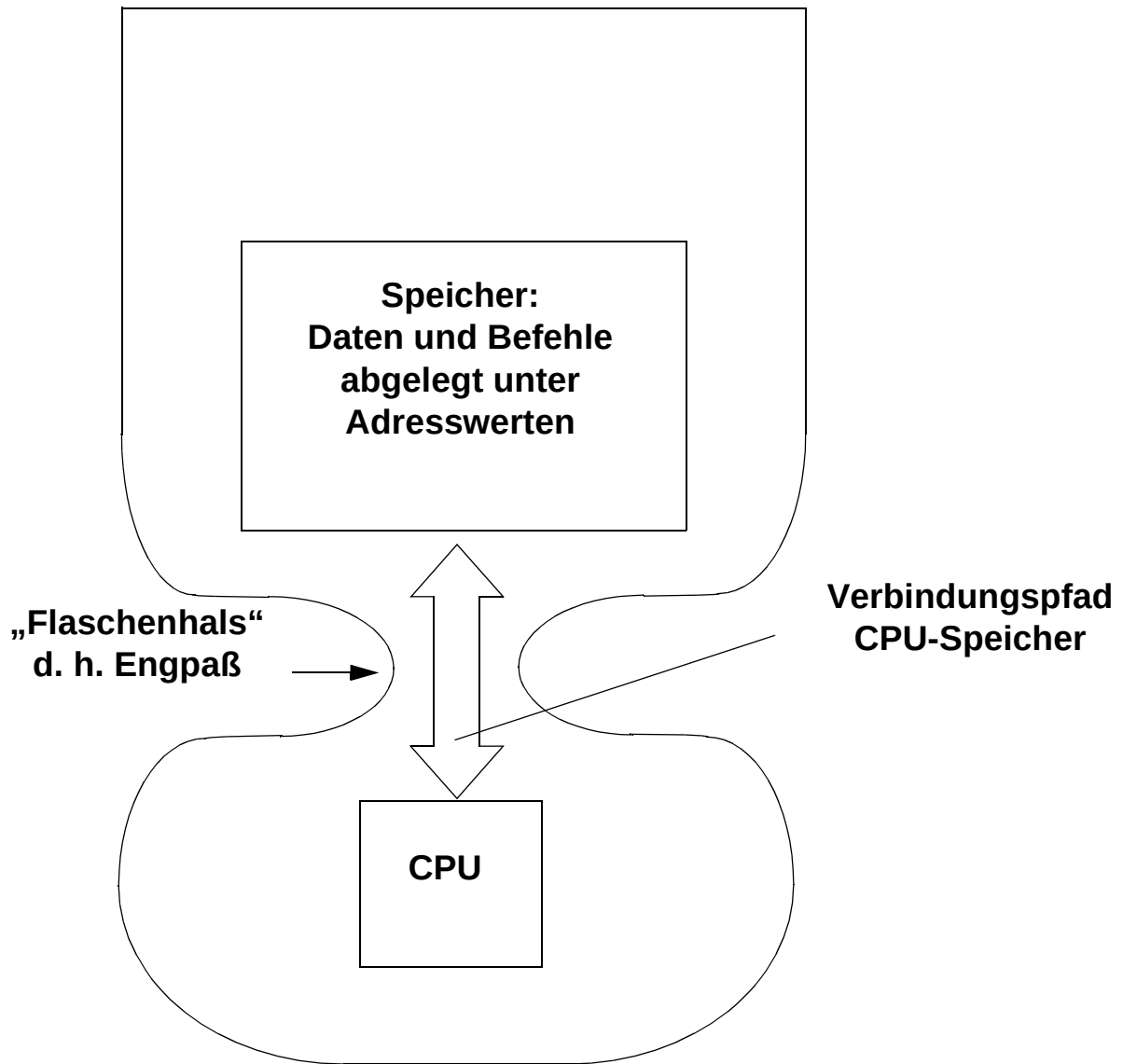
- Nachteil: Ist der Datenspeicher nicht ganz voll, kann er nicht zur Speicherung von Befehlen verwendet werden und umgekehrt

Bandbreite = Zahl der transferierten Bytes pro Sekunde

Vor- und Nachteile des von-Neumann-Prinzips

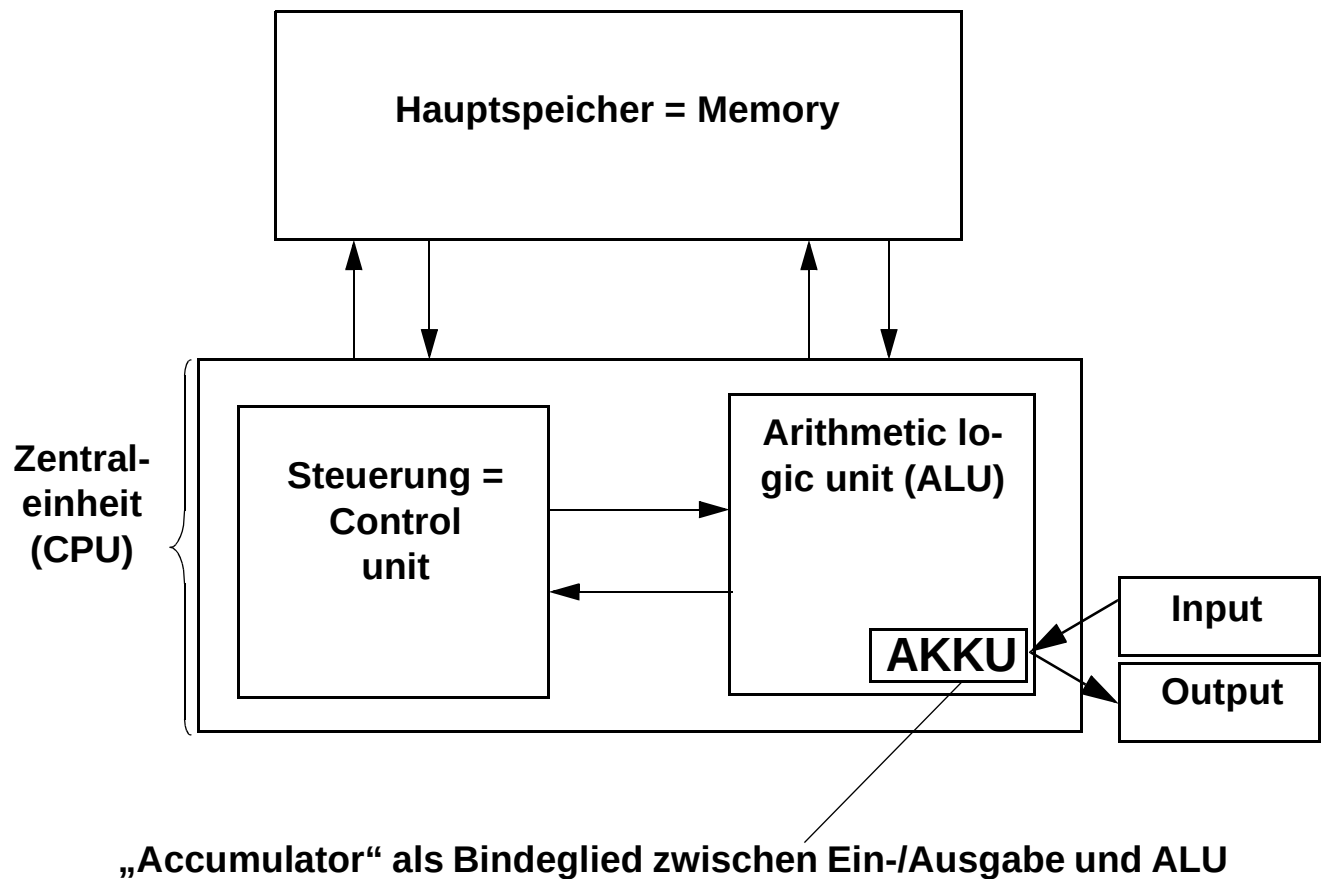
- **Vorteil: Prinzip des minimalen Hardware-Aufwandes.
Es kann nichts mehr weggelassen werden**
- **Nachteile:**
 - „von Neumann-Flaschenhals“ (Flaschenhals = Engpaß):
Zugriff ist langsam, da sequentiell (nur 1 Speicherport)
 - „Semantische Lücke“ zwischen Anwendungsprogramm
und Rechner, das heißt die Maschine kennt keine komplexen Datentypen und Operationen, so wie sie der Anwender braucht

Erläuterung des von Neumann-Flaschenhalses



- Der v. Neumannsche Flaschenhals ist bis heute eines der nur näherungsweise gelösten Probleme der Rechnerarchitektur
- Eine der Lösungsmöglichkeiten sind sog. Caches

Erweiterung des von Neumann-Rechner durch einen „Akkumulator“

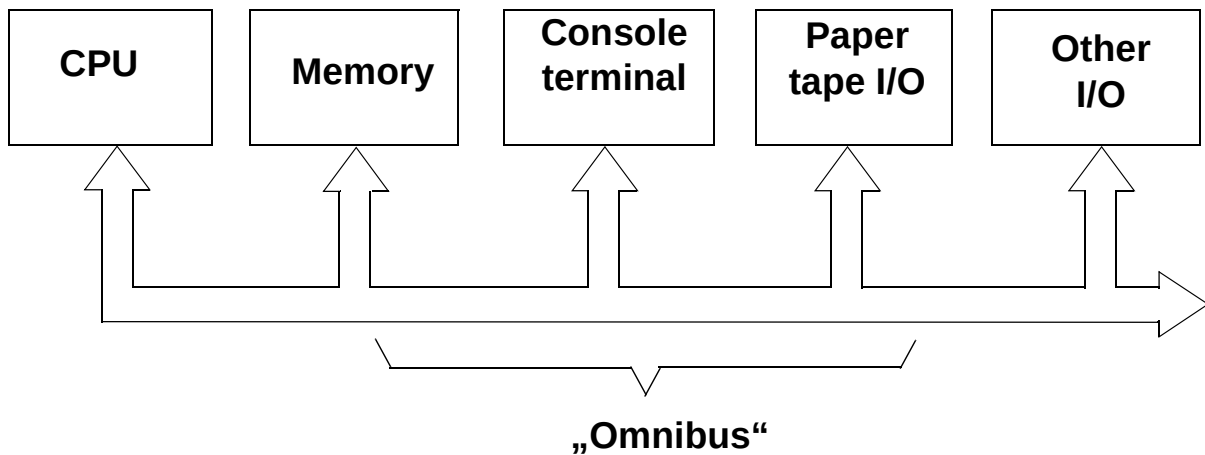


Kennzeichen:

- 1.) Programm und Daten werden in einem Speicher abgelegt (stored program computer)
- 2.) Der Speicher ist für beide gemeinsam
- 3.) Der Akku dient als CPU-interner, schneller Zwischenspeicher zum Rechnen und für die Ein-/Ausgabe

Erweiterung des von Neumann-Rechners durch einen „Bus“

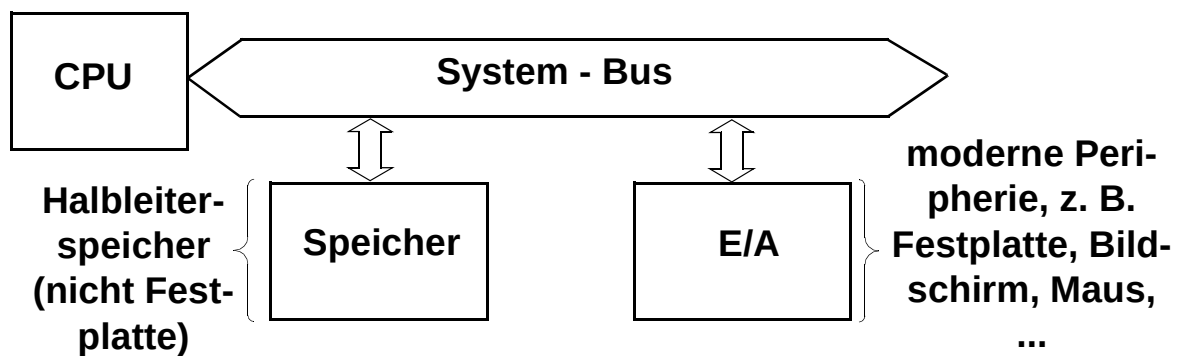
- Beispiel: Digital Equipment Corporation, PDP 8, 1968



- Aus Kosten- und Effizienzgründen wird ein gemeinsames Transportmedium eingeführt: der Bus.
- Omnibus lat. „für alle“; Bus ist die Abkürzung von Omnibus

Allgemeine Computer-Grundstruktur

- Ein typischer, konventioneller Computer besitzt folgende Grundstruktur:



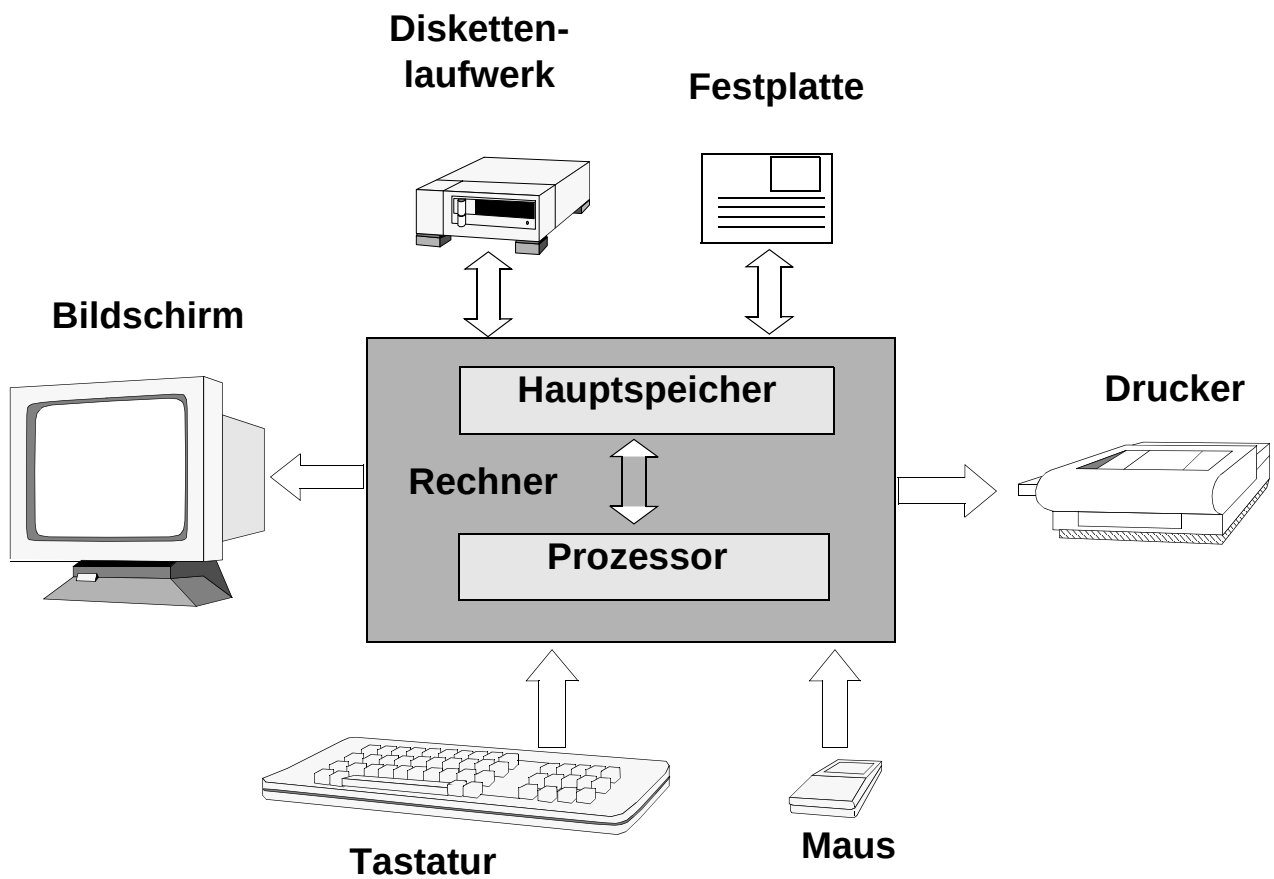
- Kennzeichnend ist, daß es nur einen zentralen Verbindungspfad für die drei Hauptkomponenten CPU, Speicher und E/A — den sog. Systembus — gibt

(Hinweis: Eine **Schnittstelle** verbindet *genau zwei* Seiten bzw. Teilnehmer miteinander; ein **Bus** verbindet *mehr als zwei* Teilnehmer.)

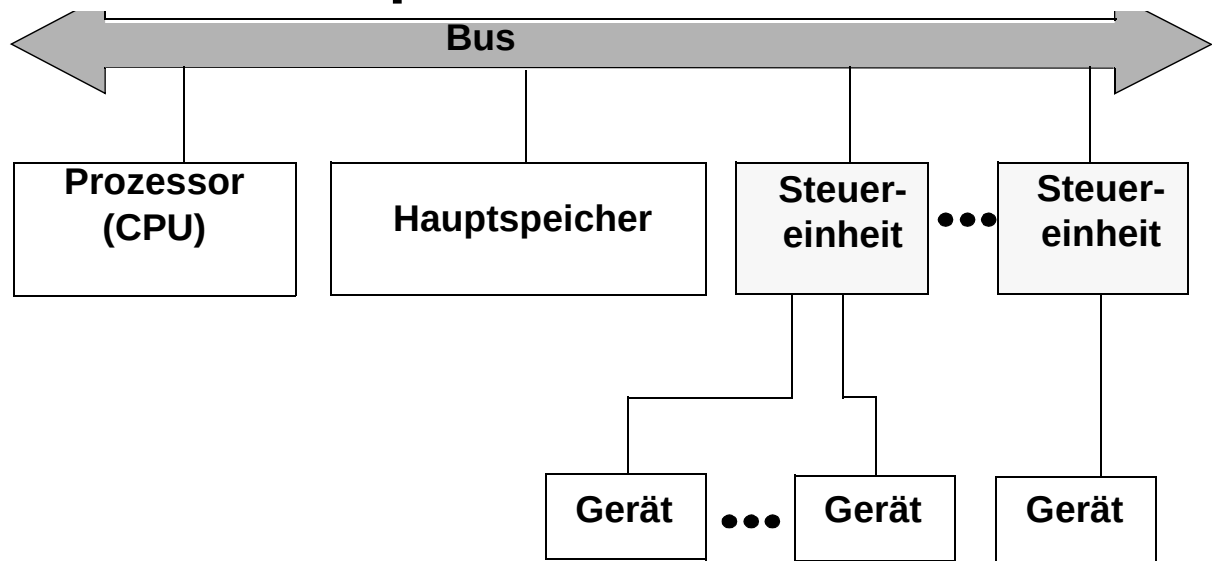
=> Problem des von Neumann-Flaschenhalses ist verstärkt, da auch die Ein-/Ausgaben über den gemeinsamen Bus gehen.

Bemerkung: Zur Leistungssteigerung werden häufig 2 Busse eingesetzt: 1 Systembus und 1 Ein-/Ausgabebus (z.B. PCI-Bus)

Ein moderner Rechner mit Peripherie



Grobarchitektur eines Universalrechners mit Peripherie/ ohne E/A-Bus

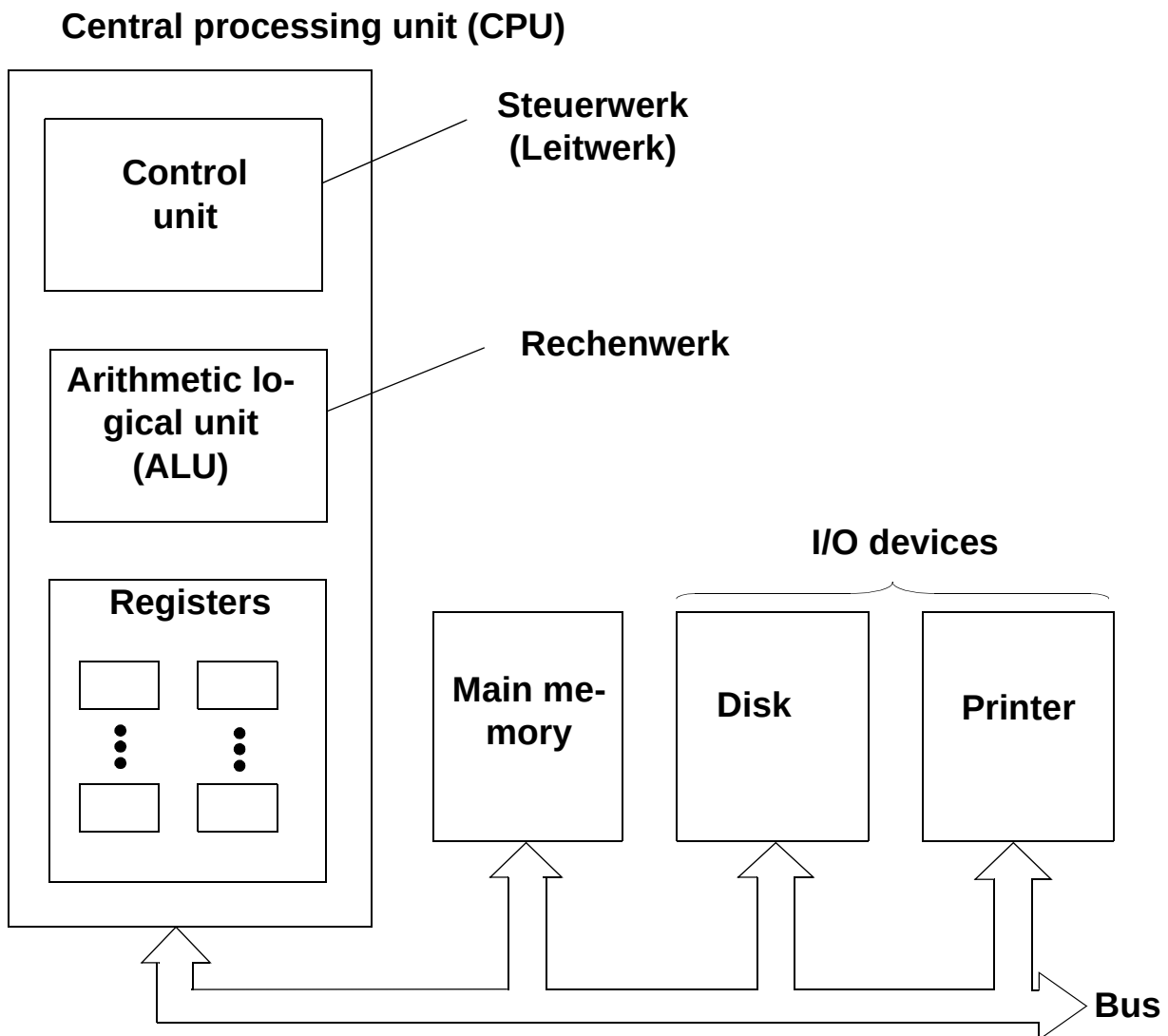


- Steuereinheiten (Controller) können intelligente Bausteine bzw. Spezialprozessoren für die Peripherie sein. Steuereinheit ist hier ein Oberbegriff für E/A-Module unterschiedlicher Leistungsfähigkeit vom einfachen seriellen Schnittstellen-Modul bis zum komplexen E/A-Prozessor (EA = Ein-/Ausgabe)

Beispiele:

Bezeichnung	Gerät	
serielle Schnittstelle (RS 232C, V.24)	Maus, Modem	einfach
parallele Schnittstelle (Centronics)	Drucker	
SCSI	Platte, Scanner	komplex
Ethernet-Adapter	Lokales Netz	

Zusammenfassung heutiger Rechner (nicht sehr leistungsstark)



Kennzeichen:

Architektur wie beim von Neumann-Rechner plus folgende Innovationen:

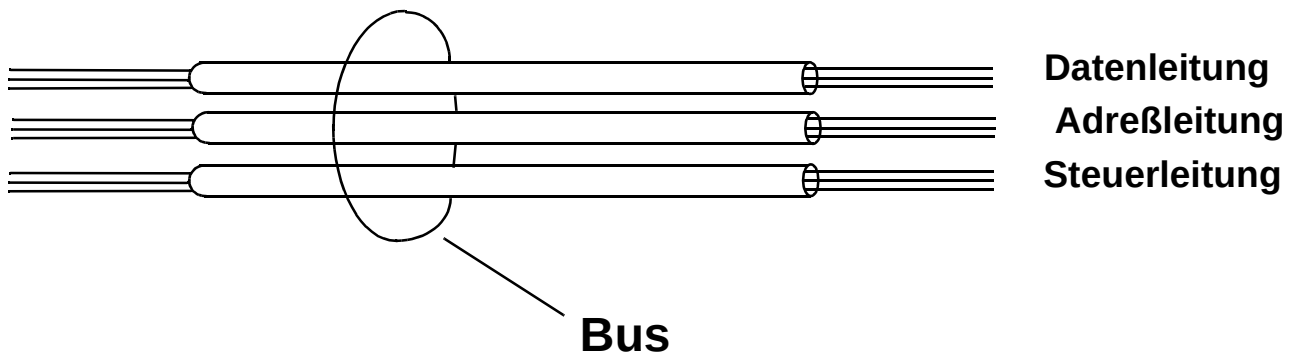
- gemeinsamer Bus
- Register als Datenspeicher in der CPU

Bemerkung: zur Vereinfachung kein getrennter System- und Ein-/Ausgabebus (= leistungsschwach)

Überblick über die wichtigsten funktiona- len Komponenten eines Rechners

Der Bus

- Ein Bus ist ein Übertragungsmedium, das von mehreren Einheiten gemeinsam benutzt wird
- Er besteht aus einer Menge paralleler Leitungen, die in drei Gruppen untergliedert werden können



- Ansonsten unterscheiden sich Busse erheblich bezüglich der Daten und Adreßbreite, der Art der Zugangsregelung, der Anzahl der anschließbaren Geräte u. a.
- Ein typischer heutiger Systembus (z. B. VME, PCI) verfügt über 32 Daten- und 32 Adressleitungen und hat eine Bandbreite von 150 MB/s bis 1,5 GB/s

Hauptspeicher

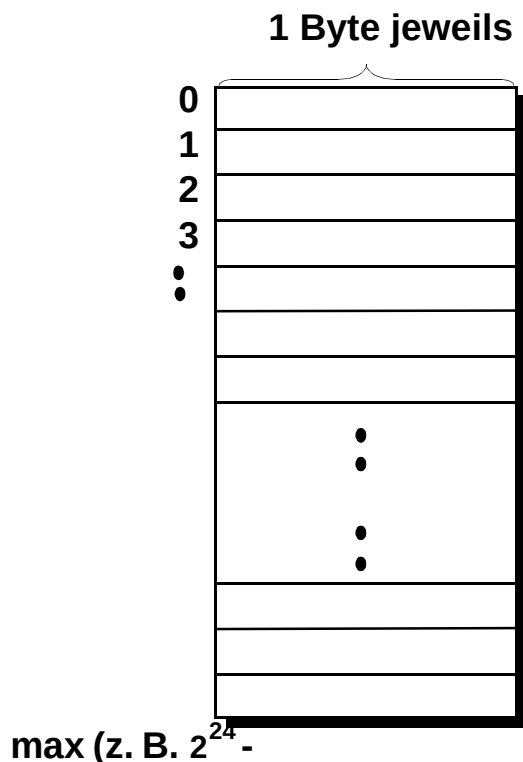
Der Hauptspeicher dient der temporären Speicherung der gerade aktiven Programme und ihrer Daten

Er ist ein Halbleiterspeicher, der durch Siliziumchips realisiert ist

Logisch gesehen besteht er aus einer Folge von Bytes, die einzeln gelesen und geschrieben werden können (RAM = Random Access Memory)

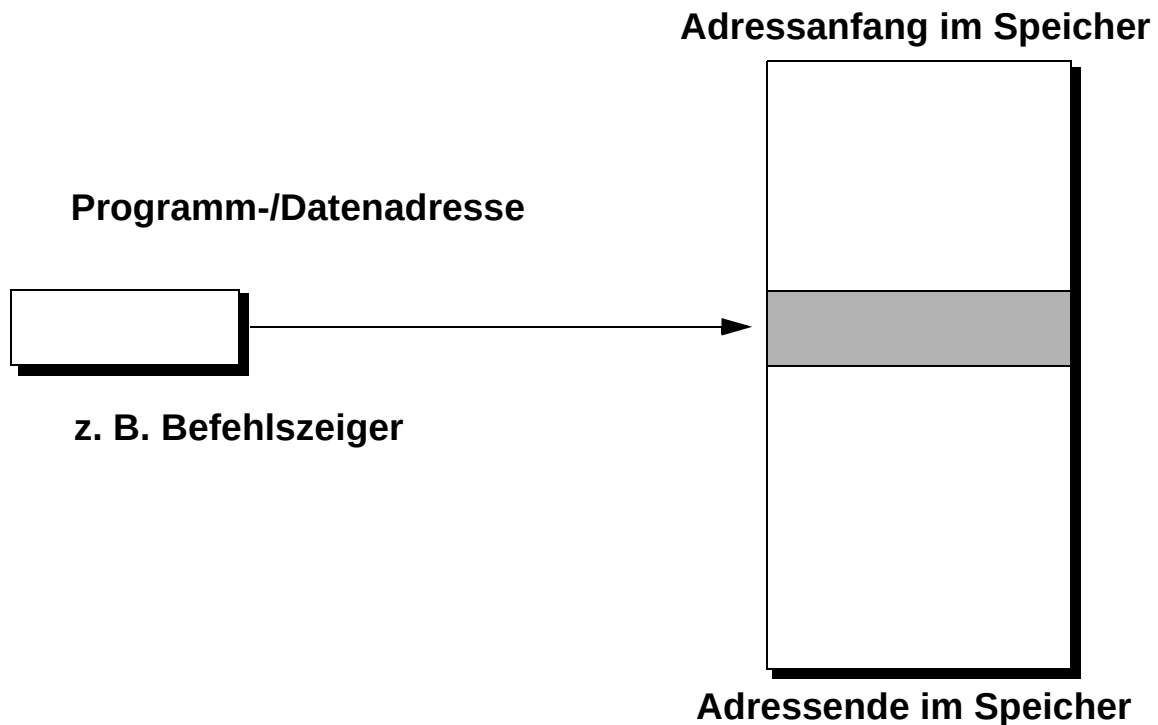
Sein Inhalt geht beim Ausschalten des Rechners verloren. Nach dem Einschalten wird das Betriebssystem geladen

Typische Größe (bei einem heutigen PC): 256 MByte



Hauptspeicher und Adressierung

Direkte Adressierung (= einfachster Fall der Adressierung)

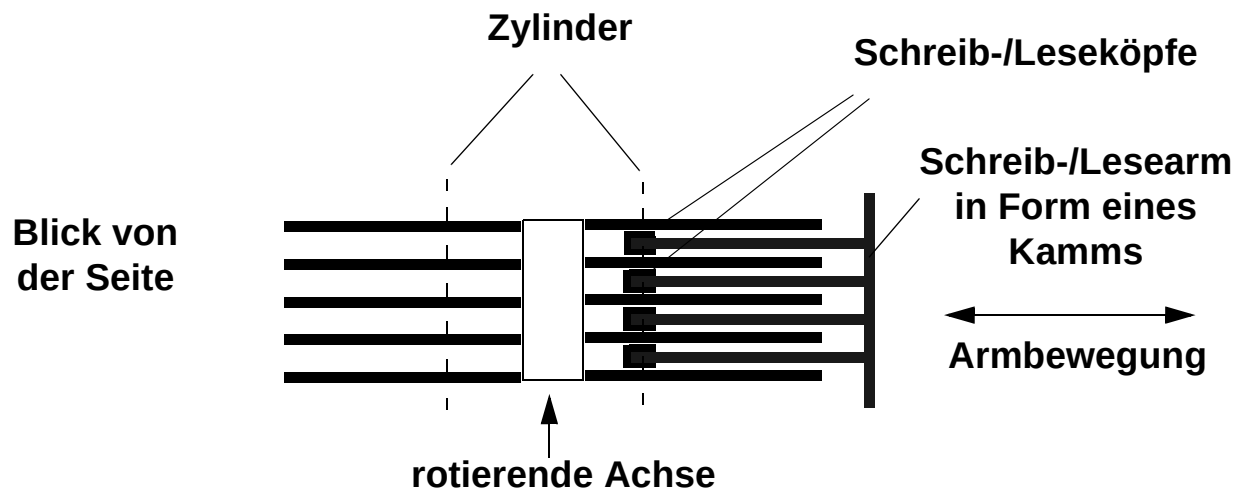


- Die Adressen, die in den Befehlen stehen, können direkt als Speicheradressen verwendet werden (Programmadresse = Speicheradresse)
- Sie werden bereits vom Übersetzer (= Compiler bzw. Binder = Loader) festgelegt. (Adressbindung zur Übersetzungszeit)
- Die Lage des Programms im Hauptspeicher ist daher vorgegeben

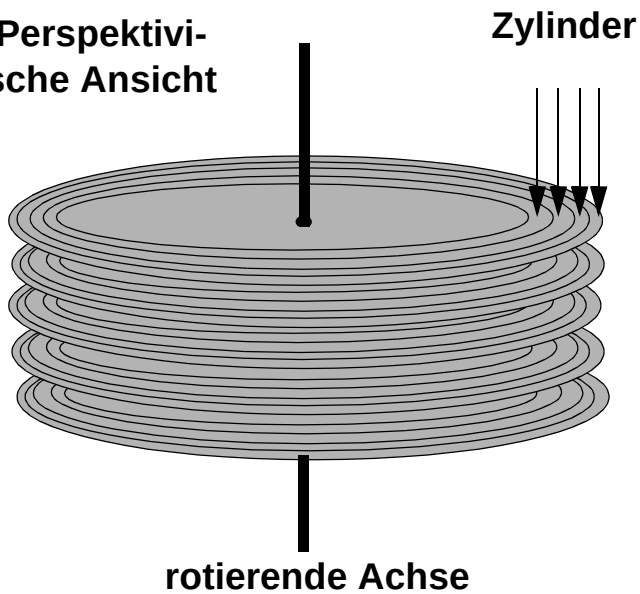
Plattenspeicher

- Der Plattenspeicher dient der *permanenten* Speicherung von Programmen und Daten. Typische Kapazität (in einem heutigen PC): 10 - 100 Gbyte
- Ein Plattenspeicher besteht aus einer oder mehreren Platten oder Scheiben, deren Oberflächen magnetisierbar sind und die mit hoher Geschwindigkeit rotieren (> 4000 Upm)
- Für jede magnetisierbare Oberfläche existiert ein Schreib-/Lesekopf, der hin- und herbewegt werden kann, um die gesamte Oberfläche zu erreichen. Die Einrichtung, an der die Köpfe befestigt sind, heißt Plattenarm.
Besteht der Plattenspeicher aus mehreren Einzelplatten, so hat der Plattenarm die Form eines Kamms. (Jeder Zacken den Kamms ist ein Schreib-(Lesekopf).
- Jede Oberfläche besteht aus konzentrischen Spuren (500 - 2000), auf denen die Information aufgezeichnet ist.
- Jede Spur wiederum ist in mehrere Sektoren (32 - 128) aufgeteilt. Diese Sektoren sind die kleinsten les- oder schreibbaren Einheiten (z. b. 512 Bytes), sog. Blöcke
- Die Gesamtheit aller Spuren, die mit einer festen Armposition gelesen werden kann, wird auch Zylinder genannt.

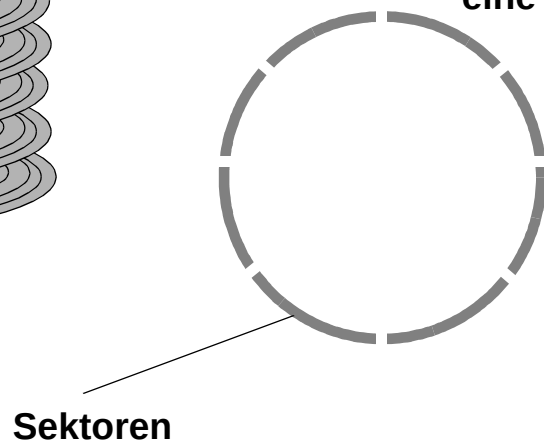
Organisation einer Festplatte



**Perspektivi-
sche Ansicht**



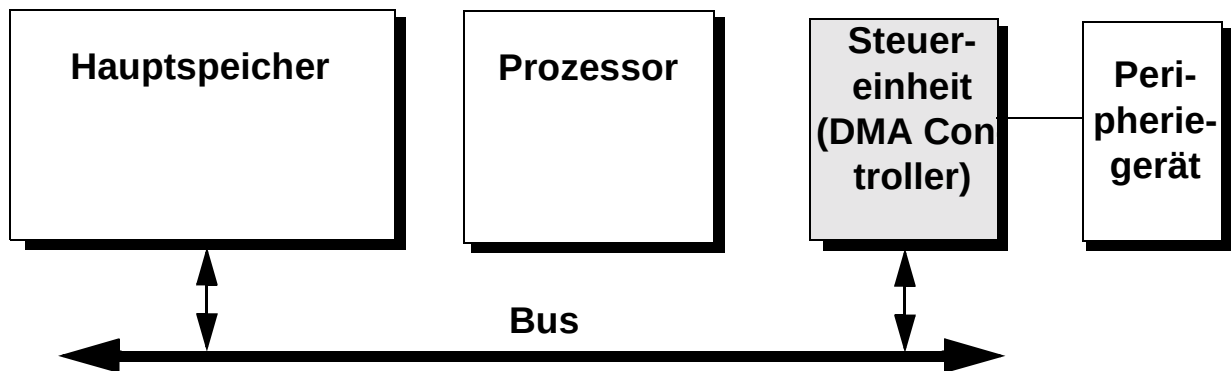
**Blick von
oben auf
eine Spur**



Datentransport von/zu Peripherieeinheiten

Der Datentransport zwischen Ein/Ausgabegeräten und dem Hauptspeicher wird heute üblicherweise per DMA (direct memory access) durchgeführt.

DMA bedeutet, dass die Steuereinheit des Geräts in der Lage ist, selbstständig auf den Hauptspeicher zuzugreifen. (selbstständig = ohne CPU)



Der Prozessor muss zum Auslösen des DMA-Transfers entsprechende Register in der jeweiligen Steuereinheit laden, um z. B. Quelle, Ziel und Umfang des Datentransfers anzugeben.

Nach Beendigung der E/A muß der Prozessor informiert werden. Dazu gibt es zwei Möglichkeiten:

- **Polling:**

Der Prozessor fragt periodisch das Statusregister der Steuereinheit ab (In den meisten Fällen zu ineffizient)

- **Unterbrechung (Interrupt):**

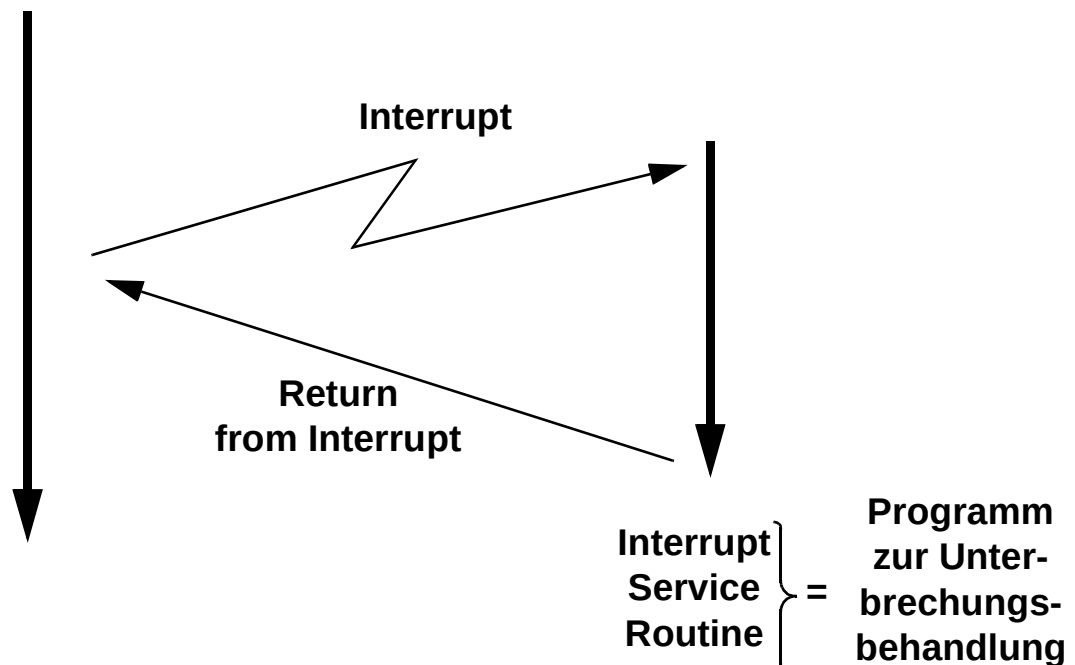
Der Prozessor wird vom Ende der Übertragung durch ein spezielles Signal informiert (= übliches Verfahren)

Unterbrechungen (Interrupts)

- Der Bus verfügt über (mindestens) eine Unterbrechungsleitung, die an den Prozessor angeschlossen ist.
- Nach jedem Befehl prüft der Prozessor, ob an dieser Leitung ein Signal (Spannung) anliegt.
- Falls ja, wird augenblicklich eine Prozedur angesprungen, die die Unterbrechung auswertet und dann die erforderlichen Aktionen durchführt oder veranlasst

Normale Programmausführung

Unterbrechungsbehandlung



Darstellung von Gleitkommazahlen in einem Rechner

Zahldarstellung für Kommazahlen mit großem Wertebereich (sog. Gleitkommazahlen)

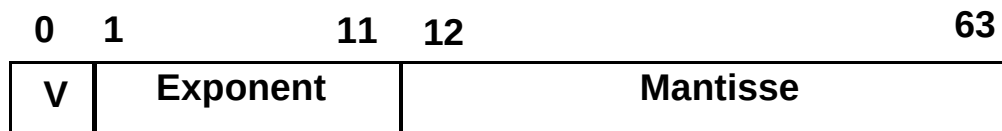
- Für solche Zahlen wird ein anderes Format (= floating point format) verwendet als für Festkommazahlen. Es wird die Zahl in eine normalisierte Mantisse und einen Exponenten zerlegt.

Beispiel: 1.746.399,1242 wird dargestellt als:

$$\underbrace{0,17463991242}_{\text{Mantisse}} \cdot \underbrace{10^7}_{\text{Exponent}} \quad (\text{im Dezimalsystem})$$

Üblicherweise wird das IEEE-Standardformat (double precision) mit Länge von 64 Bit verwendet:

Im
Dual-
system



↓
11 Bit für Exponent
52 Bit für Mantisse
1 Bit für Vorzeichen
der Mantisse

Größte darstellbare Zahl: ca. $2^{1024} \approx 10^{308}$,

Größte Genauigkeit: ca. $2^{-52} \approx 10^{-16}$ d. h.
16 Dezimalstellen

- Vor jeder arithmetischen Operation muss die Mantisse angeglichen werden. Was über die 16 Stellen hinausgeht, wird abgeschnitten.

Beispiel:

Die Summe

65.749.230.736,16372 + 47.828,00829834098

führt zur Rechnung:

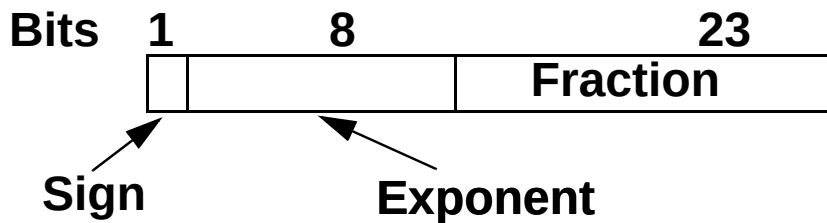
$$\begin{array}{r} 0,6574923073616372 \\ 0,0000004782800929834098 \\ \hline 0,6574927856417201 \end{array}$$

fällt weg, da länger als 16 Stellen

Gleitkommazahlen nach IEEE-Standard 754

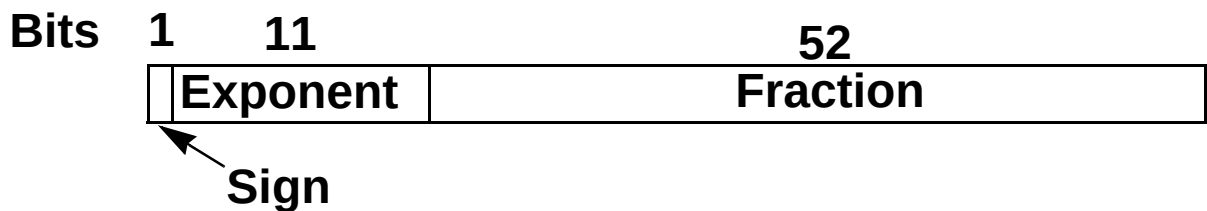
Es gibt 2 Wertebereiche: single und double precision

(a)



32 Bit single precision => Wertebereich $\pm (10^{-38}$ bis $10^{+38})$

(b)



64 Bit double precision => Wertebereich $\pm (10^{-308}$ bis $10^{+308})$

- Beim IEEE floating point-Standard 754 wird bei single precision der Exponent mit einem Offset von 127 dargestellt, d. h. positive Exponenten sind ≥ 127 und negative < 127 . Bei double precision ist der Offset 1023.
- Die Mantisse liegt bei IEEE 754 per Definition im Bereich zwischen 1.0 und 1.999 ..., wobei nur die Nachkommastellen abgespeichert werden (= „normalisiert“) Achtung: andere Definition von „normalisiert“ bei nicht- IEEE-Zahlenformaten

Darstellung beliebiger Zahlen, wenn die Mantisse als Zahl zwischen 1.0 und 1.999... interpretiert wird

- Satz: man kann jede rationale Zahl mit endlicher Stellenzahl zerlegen in ein Produkt zweier Zahlen, wobei die eine zwischen 1 und 2 liegt und die andere 2^{**x} ist, mit x aus der Menge der pos. und neg. ganzen Zahlen

=> Beliebige Zahlen werden dargestellt, indem man die darzustellende Zahl y zerlegt in $y = (y/2^{**x}) * 2^{**x}$

- Für Zahlen $y < 1.0$ ist $x < 0$ derart, daß $1.0 \leq (y/2^{**x}) \leq 1.999...$ gilt
- für Zahlen $y > 1.999...$ ist $x > 0$ derart, daß $1.0 \leq (y/2^{**x}) \leq 1.999...$ gilt
- für Zahlen $1.0 \leq y \leq 1.999$ ist $x = 0$, weil so $1.0 \leq (y/2^{**x}) \leq 1.999...$ gilt
- Beispiele: 0,5 ist nach single precision IEEE 754 = 3F000000; 1,0 = 3F800000; 1,5 = 3FC00000

Zahlentypen nach IEEE-Gleitkommastandard 754

Normalized	±	$0 < \text{Exp} < \text{Max}$	Any bit pattern „1111...“ = 1,9999...	Für normalen Wertebereich
Denormalized	±	0	Any nonzero bit pattern	Für erweiterten Wertebereich
Zero	±	0	0	± 0
Infinity	±	1 1 1 ... 1	0	$\pm \bullet$
Not a number	±	1 1 1 ... 1	Any nonzero bit pattern	

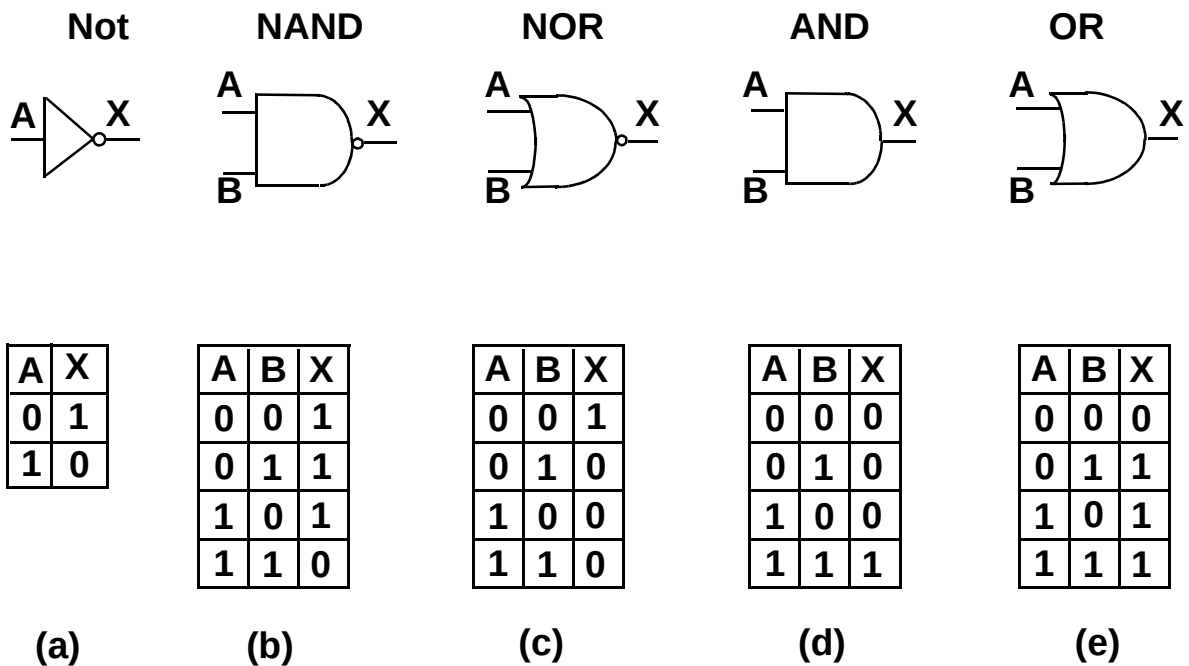


- Not a number: dient zur Anzeige, daß bei einer Gleitkommaoperation etwas schief gegangen ist, z.B. durch den Versuch, die Wurzel aus -1 zu ziehen
- Denormalisierte Zahlen erlauben das Rechnen in einem erweiterten Zahlenbereich $\pm 10^{-45}$ statt $\pm 10^{-38}$ bzw. $\pm 10^{-324}$ statt $\pm 10^{-308}$ bei gleichzeitig reduzierter Genauigkeit
- Darüberhinaus gibt es $\pm \infty$ als Zahl, mit der „wie mit jeder anderen“ gerechnet werden kann $\infty + \infty = \infty$;
 $x:\infty = 0$; $x:0 = \infty$

Gatter- und Registerebene in der Zentraleinheit (CPU) eines Rechners

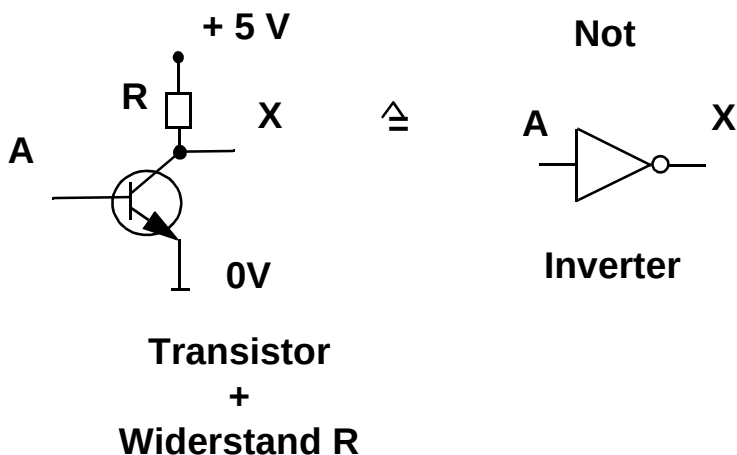
= unterste Hardwareebene des Prozessors

Elementare Gatterschaltungen (1)



- a) Invertierer (NOT)
- b) invertiertes UND-Gatter (NAND)
- c) invertiertes ODER-Gatter (NOR)
- d) UND-Gatter (AND)
- e) ODER-Gatter (OR)

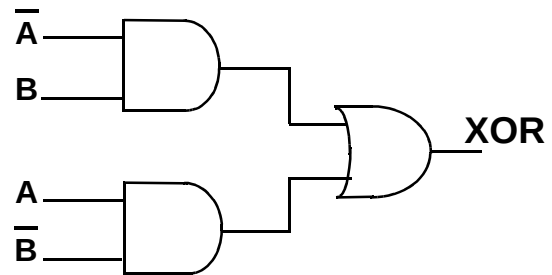
Beispielimplementierung eines Gatters:



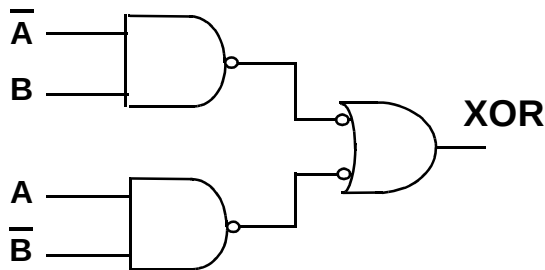
Elementare Gatterschaltungen (2) XOR

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

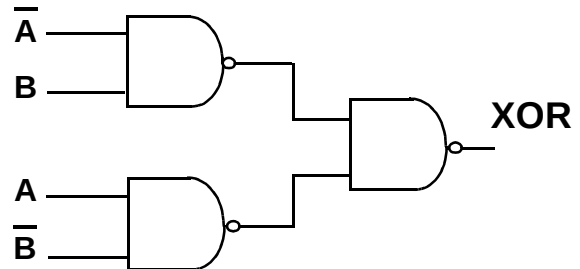
(a)



(b)



(c)

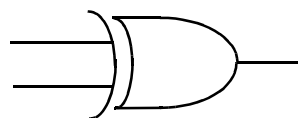


(d)

a) Wertetafel des ausschließenden ODER
(Excluxive OR, EXOR, XOR)

b)- d) drei äquivalente Implementierung von EXOR

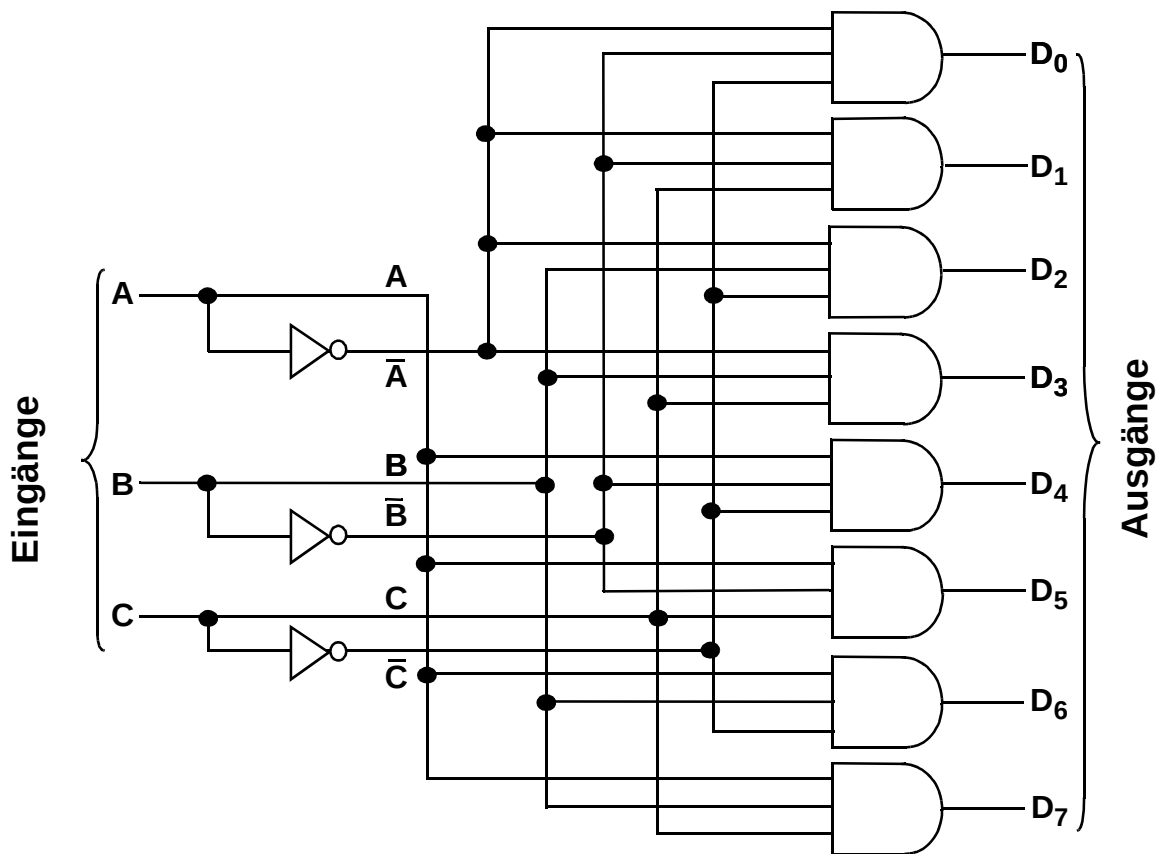
EXOR - Symbol



- Mit EXOR kann man einen der beiden Eingänge invertiert auf den Ausgang geben. Diese Funktionalität wird oft in der Praxis gebraucht.

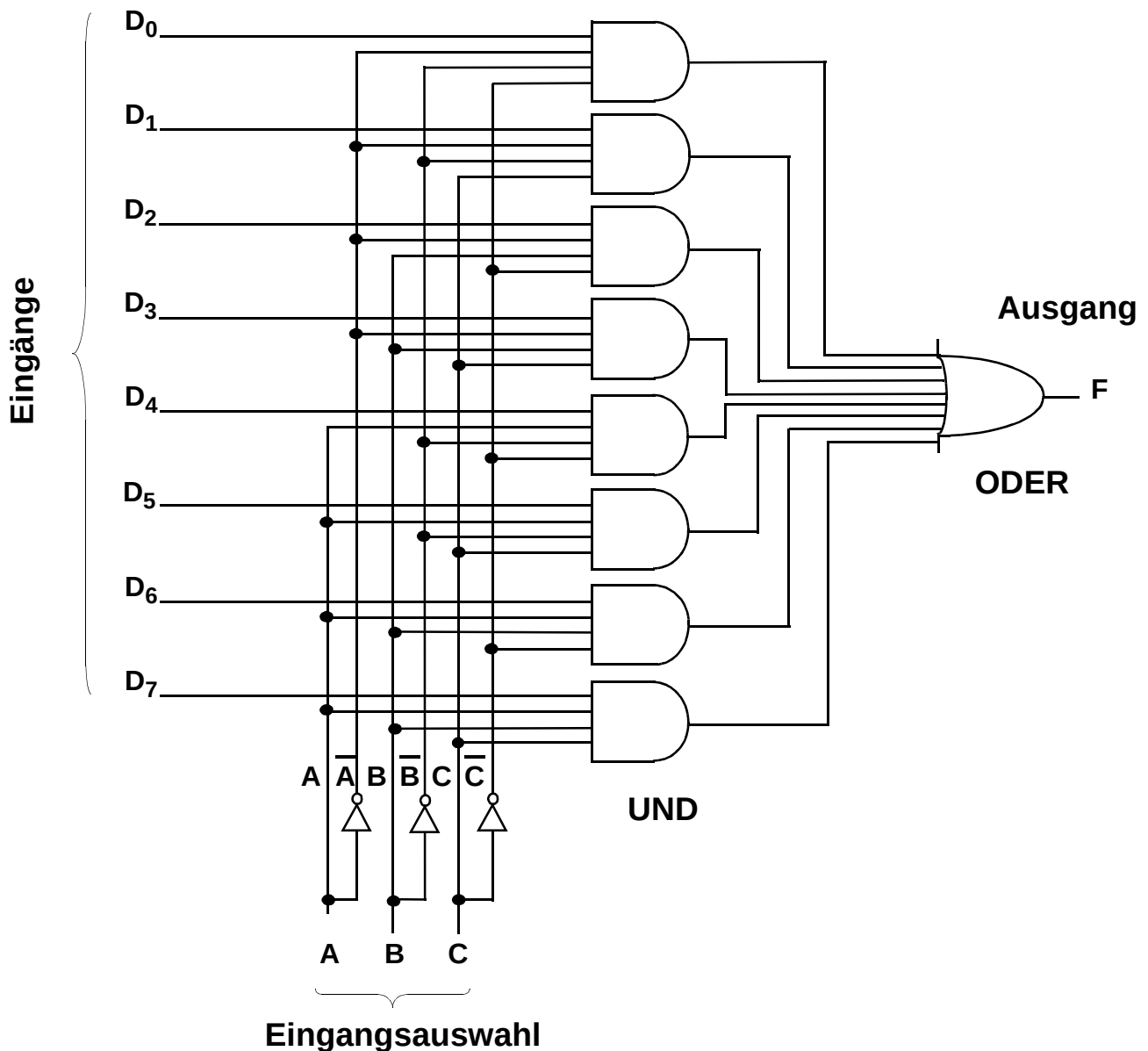
Beispiel einer Gatterzusammenschaltung

3 → 8-Dekodierer



- setzt einen von 8 Ausgängen auf 1
- Anwahl des Eingangs erfolgt im Binärkode
- wird verwendet zur Dekodierung der Befehls Worte in der CPU

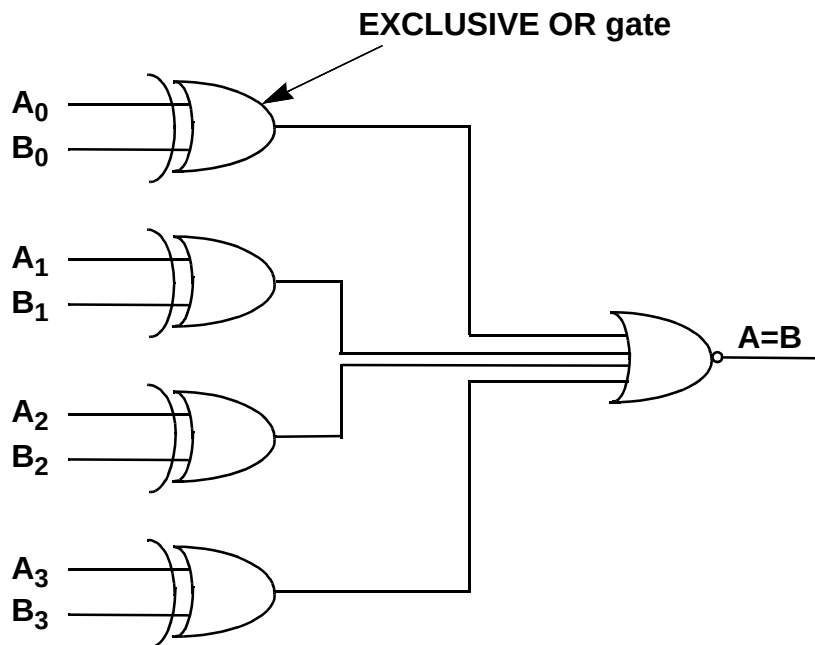
8 → 1-Multiplexer



- wählt einen von 8 Eingängen aus und gibt ihn auf den Ausgang
- Auswahl des Eingangs erfolgt im Binärcode
- wird verwendet, um CPU-Register od. Rechenwerk aus verschiedenen Quellen mit Operanden zu versorgen

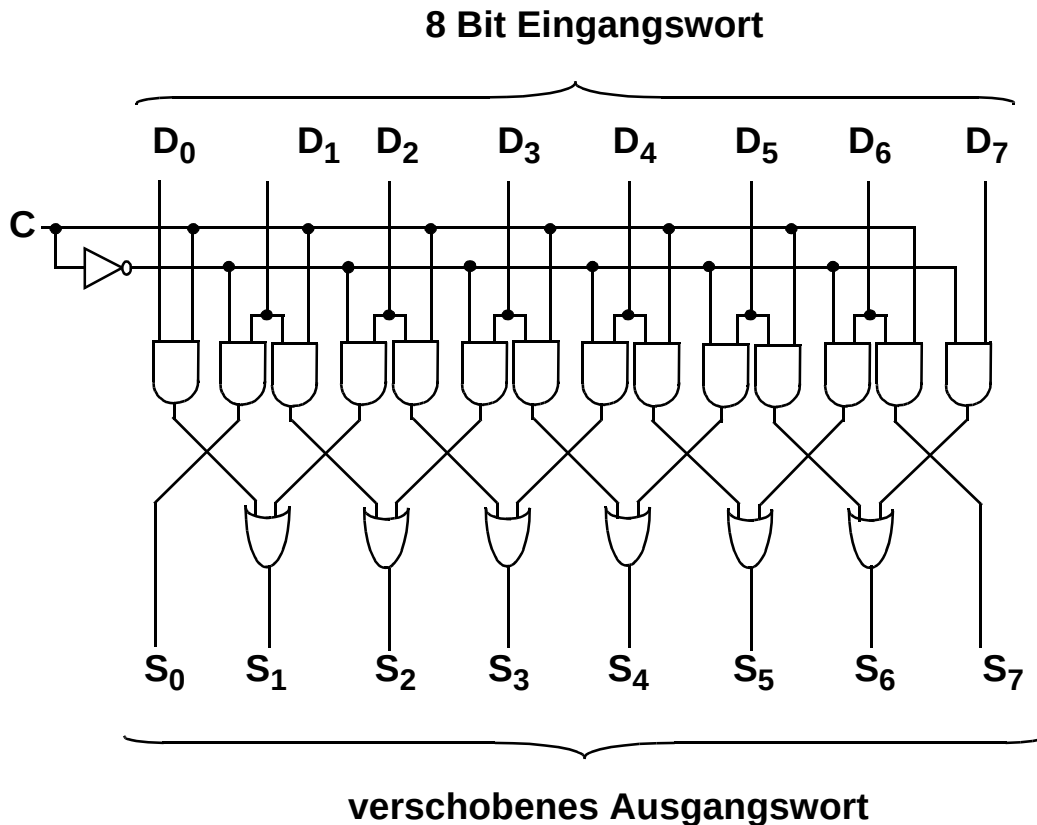
4-Bit-Komparator

Gate = Gatterschaltung



- vergleicht zwei 4-Bit-Worte miteinander
- wird im Rechenwerk für die Ausführung des Assembler-Befehls COMPARE A,B verwendet

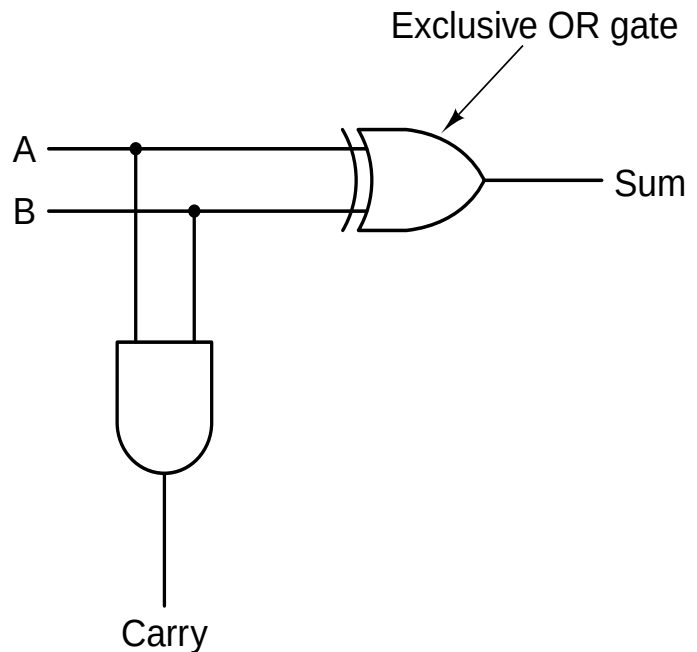
1-Bit-Links/Rechtsschieber



- Verschiebt alle Bits eines 8-Bit-Wortes nach links oder rechts um ein Bitposition
- $C = 1$ = Rechtsschieben, $C = 0$ = Linksschieben, C = Control
- verschobenes Ausgangswort enthält 7 Bit des Eingangsworts + 1 Null-Bit, das neu erzeugt wird
- wird im Rechenwerk für die Ausführung des Assembler-Befehls SHIFTLOGICAL A,n verwendet

1-Bit-Halbaddierer

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

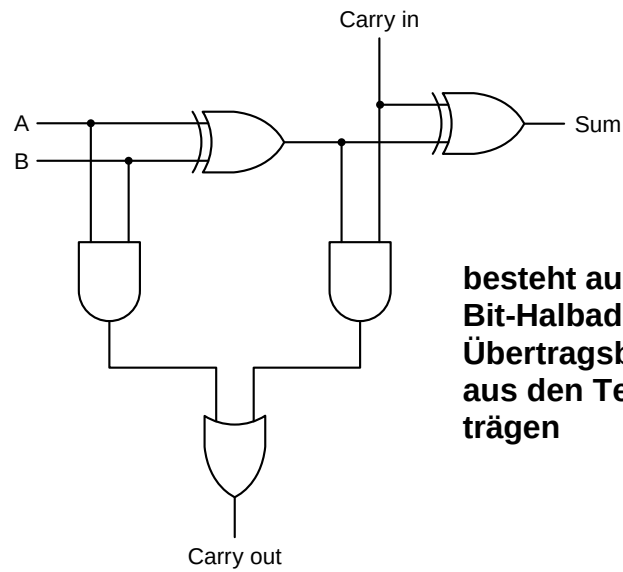


- Addiert zwei 1-Bit-Worte und bildet einen Übertrag
- Ist nicht kaskadierbar, da der Übertrag einer vorangegangenen Addition nicht verwertet werden kann
- Wird deswegen als Halbaddierer bezeichnet

1-Bit-Volladdierer

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

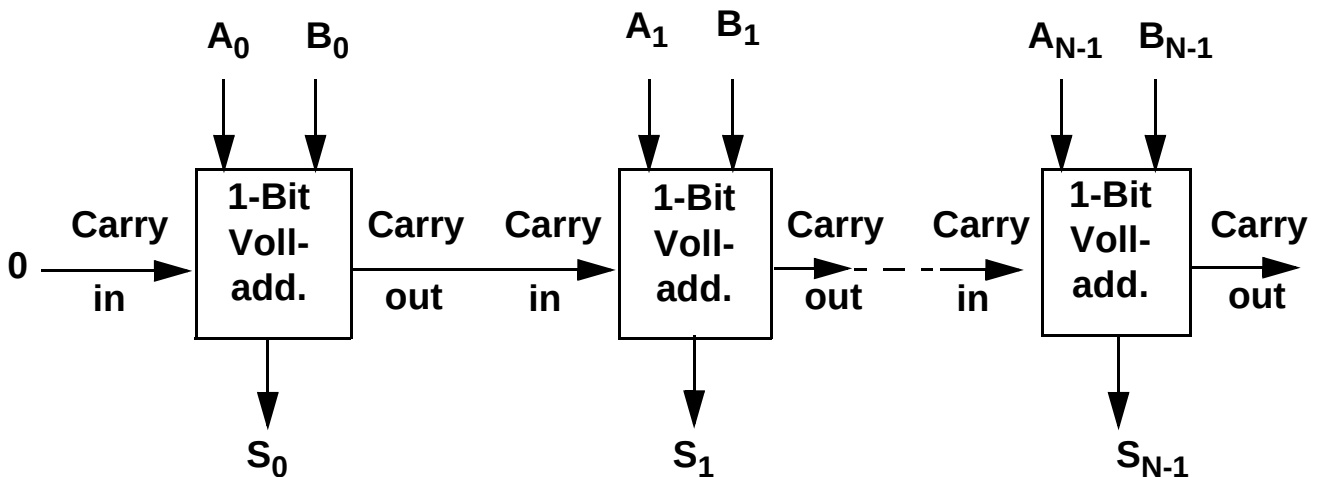
(a)



(b)

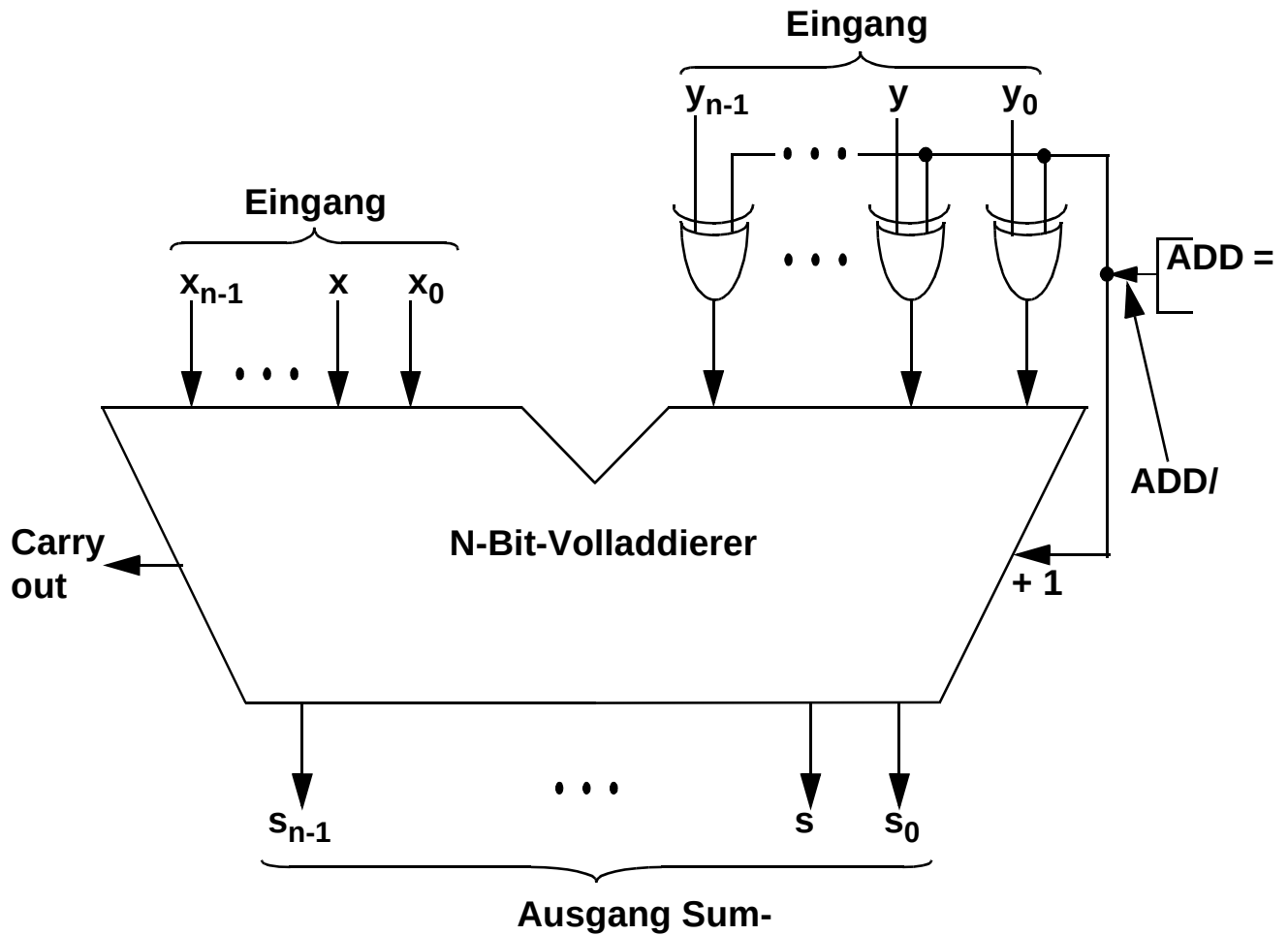
- berücksichtigt zusätzlich den Übertrag einer vorangegangenen Addition
- mehrere 1-Bit Volladdierer können zu einem N-Bit-Volladdierer kaskadiert werden! (= Syntheseschritt)
- wird so ähnlich im Rechenwerk für die Ausführung des Assembler-Befehls ADD A,B verwendet

N-Bit Volladdierer (carry propagation adder)



- An die $2N$ Eingänge werden die je N Bits der Worte A und B angelegt
- Am Ausgang erscheint die Summe S incl. Übertrag (carry)
- Der Übertrag entsteht durch Ausbreitung des Teilübertrags von links nach rechts (= carry propagation)
- Nachteil: carry propagation hat Zeitkomplexität $O(\text{Zahl der Bits pro Addition})$

N-Bit-Addierer/Substrahierer



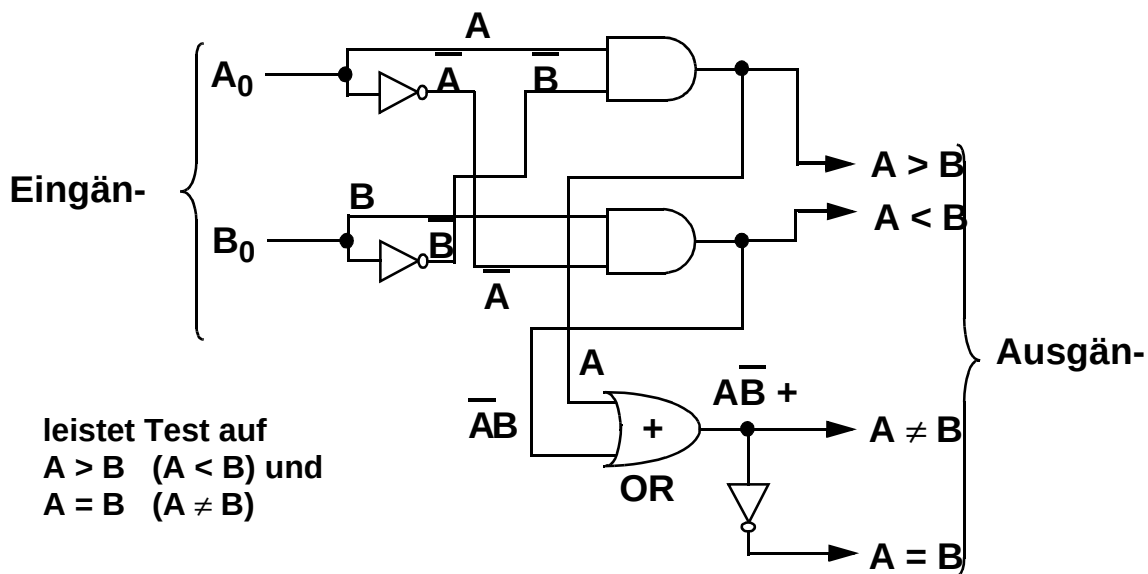
- Die Steuerleitung ADD/SUB wählt aus, ob addiert oder subtrahiert werden soll
- Die EXOR-Gatter dienen als steuerbare Inverter

$$\text{ADD: } S = x + y$$

$$\text{SUB: } S = x + \overline{y} + 1 \hat{=} x - y$$

2er-Komplement von y

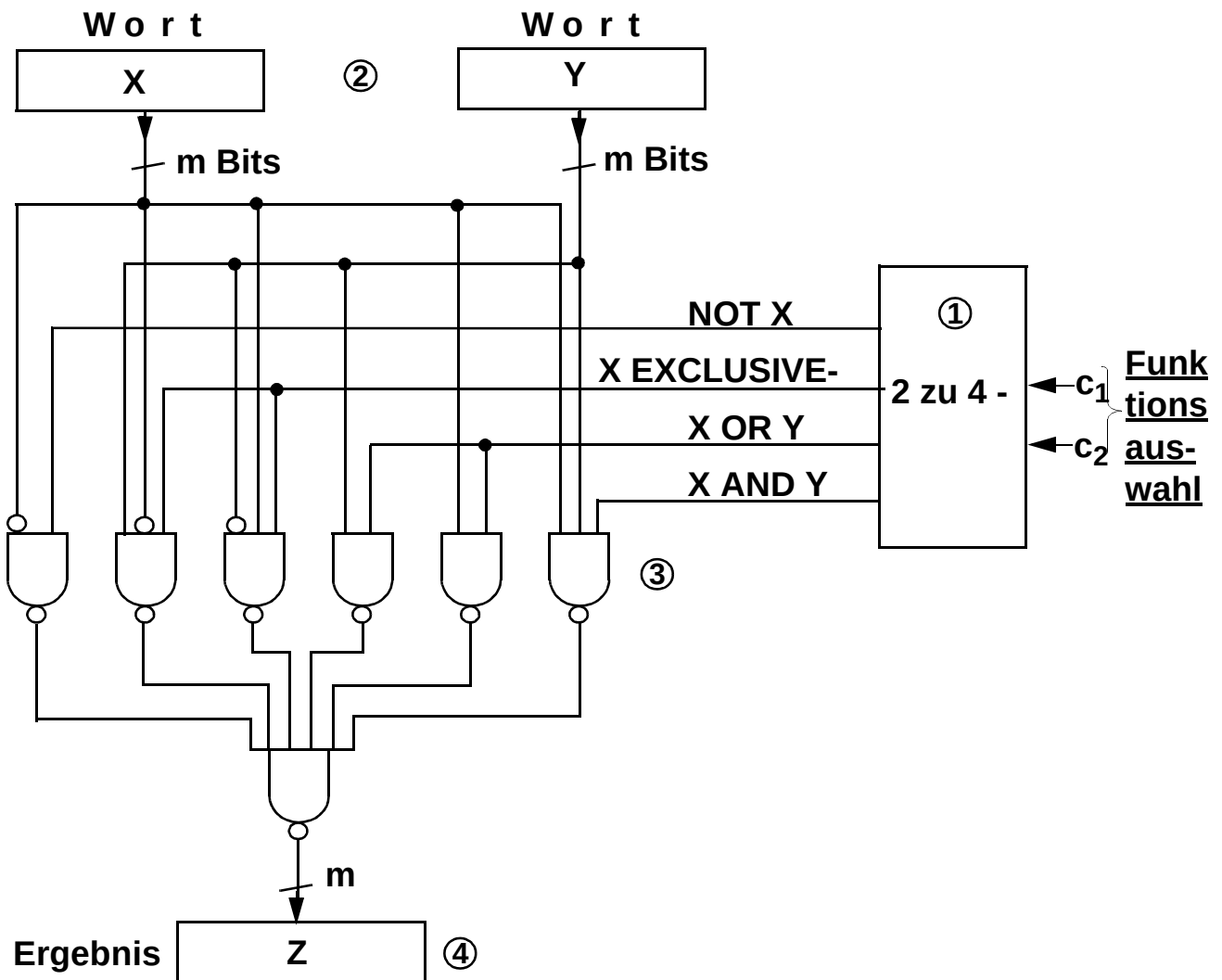
Einfache arithmetische Einheit (arithmetic unit)



- wird im Rechenwerk für die Ausführung des Assembler-Befehls CMPGT A,B (compare on greater than) und CMPEQ A,B (compare on equal) verwendet

-

Einfache logische Einheit (logic unit)



-  ist ein NAND-Gatter mit invertiertem Eingang

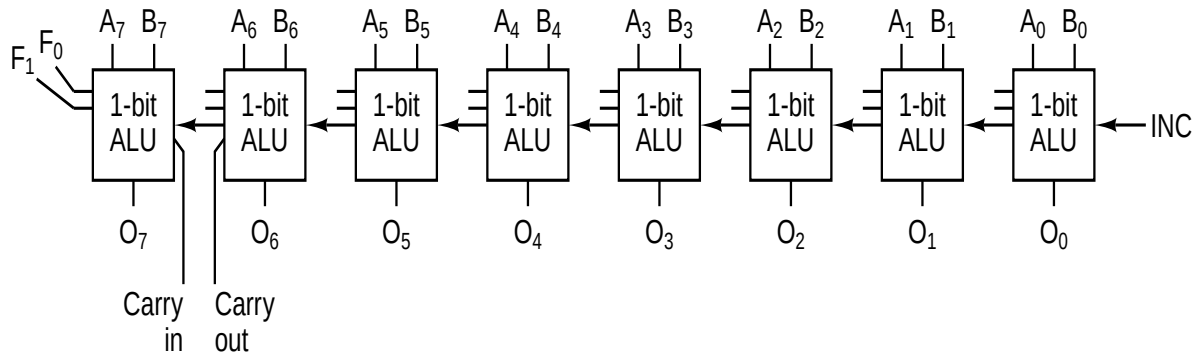
- Implementierung der vier logischen Befehle
AND, OR, EXCLUSIVE-OR, and NOT

$$Z = \underbrace{\bar{c}_1 \bar{c}_2 XY}_{\text{AND}} + \underbrace{\bar{c}_1 c_2 X + \bar{c}_1 c_2 Y}_{\text{OR}} + \underbrace{c_1 \bar{c}_2 \bar{X} Y + c_1 \bar{c}_2 X \bar{Y}}_{\text{EX-OR}} + \underbrace{c_1 c_2 \bar{X}}_{\text{NOT}}$$

- Die Funktionsauswahl legt fest, welche logische Funktion ausgeführt wird

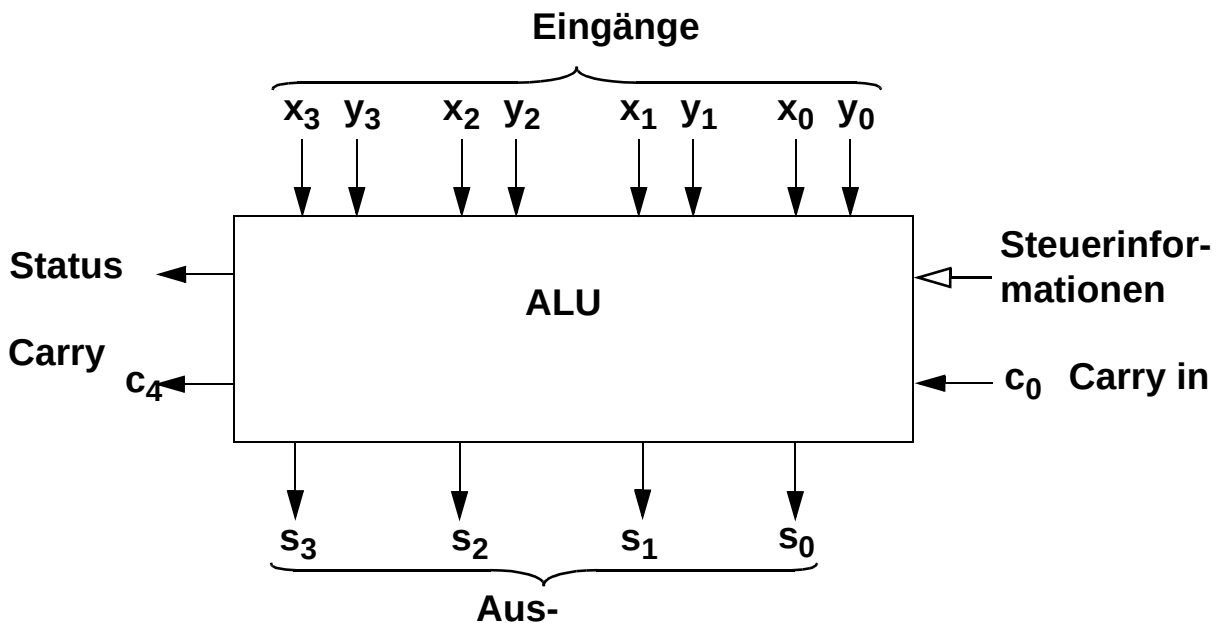
unit = Baustein, Einheit

8-Bit-Rechenwerk (ALU)



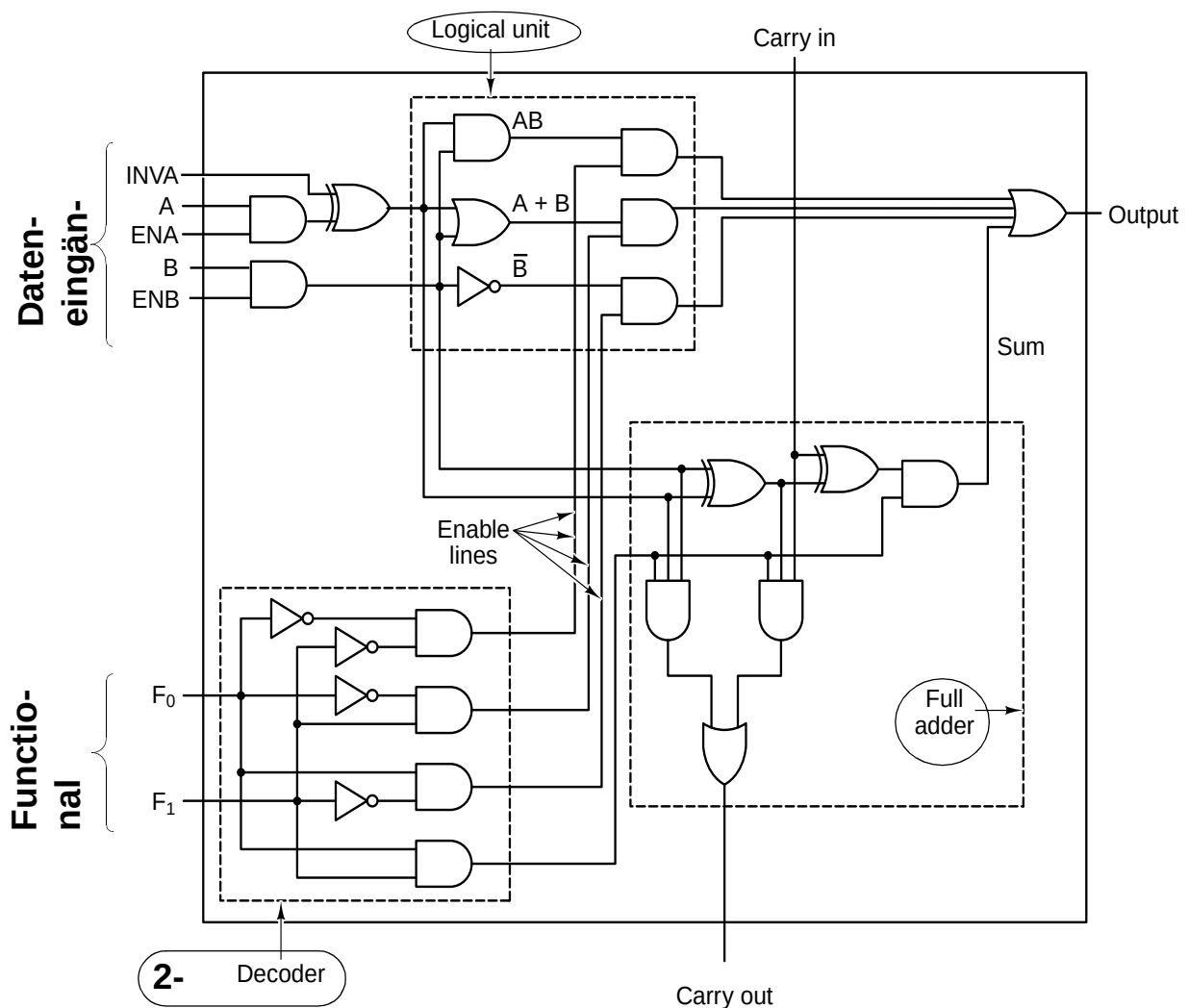
- **Erweiterung des 1-Bit-Rechenwerks auf 8 Bit (grössere Wortbreiten analog)**
- **Kaskadierung erfolgt über 8 1-Bit-Rechenwerke, die über Carry out -> Carry in gekoppelt sind (=Syntheseschritt)**
- **ALU = Arithmetic/Logical Unit = Rechenwerk**
- **Nachteil von carry-out-carry-in-Kopplung: O(n)-Zeitkomplexität**

Verallgemeinertes Rechenwerk (ALU)



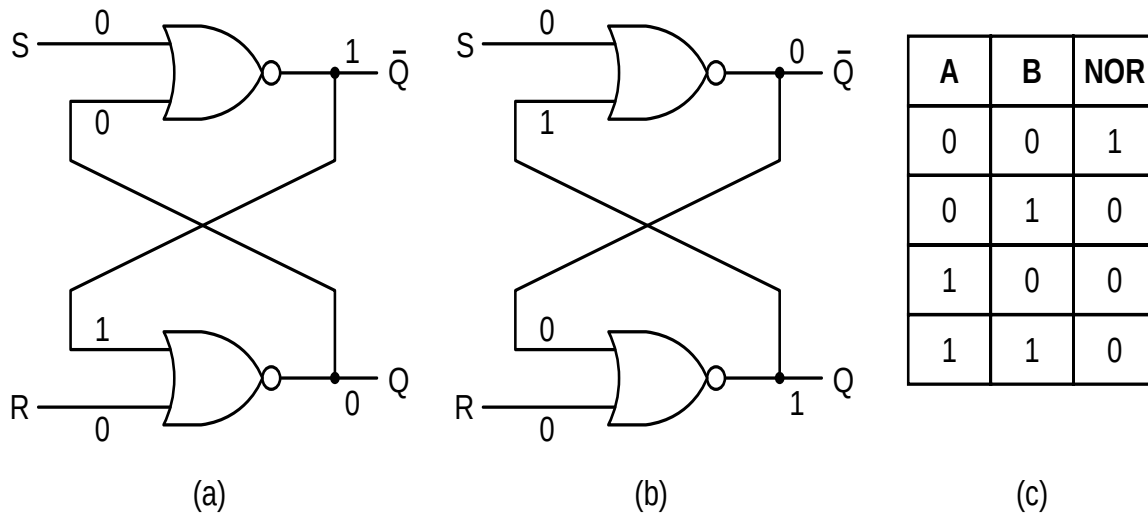
- Bei einer allgemeinen ALU gibt ein Bündel von Steuerleitungen an, welche Funktion von der ALU ausgeführt werden soll
- Neben dem Ergebnis am Ausgang gibt die ALU noch zusätzlich eine Statusmeldung aus, die Auskunft über Erfolg oder Mißerfolg der ausgeführten Operation gibt

1-Bit-Rechenwerk (Arithmetisch-Logische-Einheit, ALU)



- kann auf Wunsch eine von vier Funktionen ausführen: A AND B, A OR B und A ADD B sowie B
- Der Inhalt des A-Eingangs kann mit Hilfe des INVA-Signals zusätzlich invertiert werden
- ENA, ENB = Freigabe des Eingangs (Enable)

1-Bit-Speicher (SR-Latch) mit Setzen/ Rücksetzen



a) SR-Latch zurückgesetzt ($Q=0$)

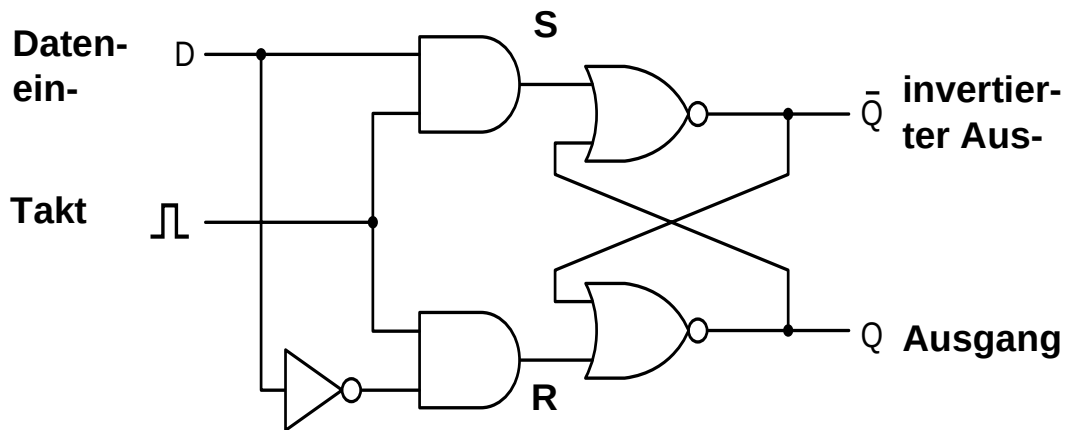
b) SR-Latch gesetzt ($Q=1$)

c) Wahrheitstafel eines NOR-Gatters

- der Latch-Ausgang Q hat zwei stabile Zustände: 0 oder 1
- der Latch-Ausgang Q wird gesetzt ($=1$) mit einer 1 am Setzeingang S
- der Latch-Ausgang Q wird zurückgesetzt ($=0$) mit einer 1 am Rücksetzeingang R

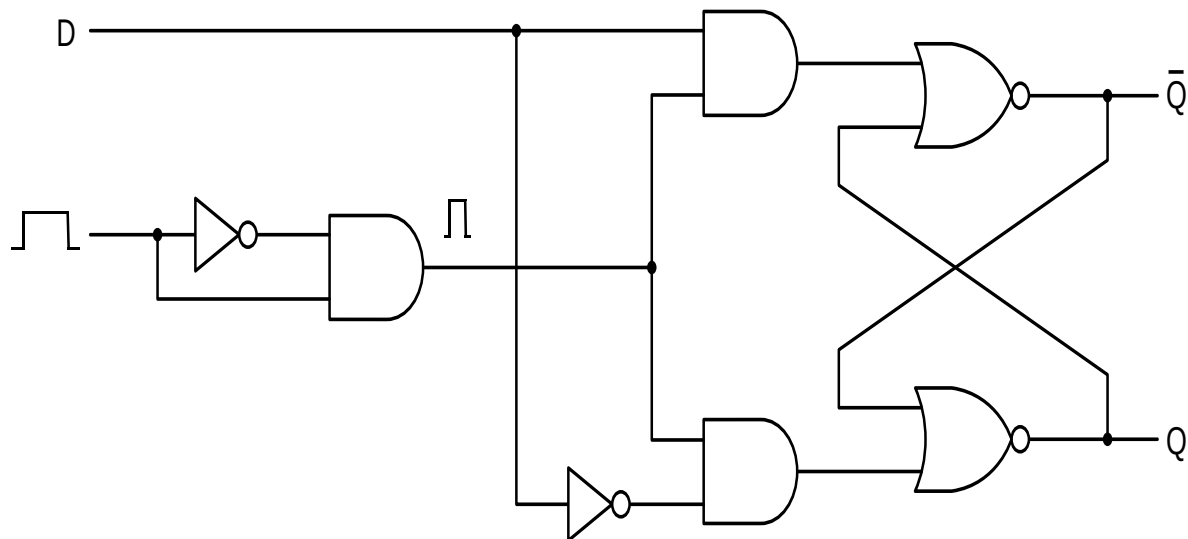
S,R = Set, Reset; Latch = (engl.), Schnappschloß

Taktgesteuerter 1-Bit-Datenspeicher (D-Latch)



- Erweiterung des SR-Latches um einen Takteingang
- Sobald der Takteingang aktiviert ist (=1) kann der Zustand des SR-Latches verändert werden
- der Zustand des SR-Latches wird von dem Wert am Dateneingang D bestimmt ($D=0 \Rightarrow Q=0$, $D=1 \Rightarrow Q=1$), d.h. der Ausgang Q speichert den Wert am Eingang D
- Der Vorteil ist, daß der Ausgang sich nur gleichzeitig mit dem Takt (=synchron) ändern kann
- Taktsynchrone Schaltungen sind leichter beherrschbar als asynchrone, deswegen sind CPUs taktsynchron

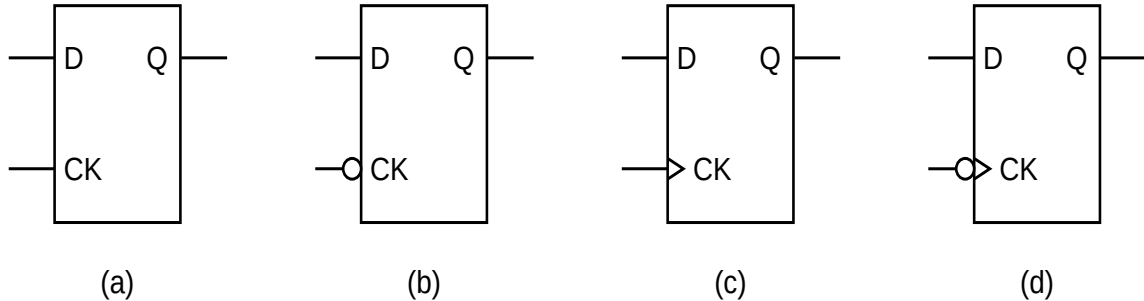
Taktflankengesteuerter 1-Bit-Speicher (D-Flip-Flop)



- Erweiterung des D-Latches so, daß nicht der Pegel am Takteingang, sondern ein Impuls beliebiger Länge genügt, um den Wert vom Eingang D zu speichern
- Der Taktimpuls wird durch die Gatterdurchlaufzeit des Inverters zu einem kurzen Impuls fester Länge umgewandelt
- D-Flip-Flops sind die wichtigsten statischen Speicherelemente

Zusammenfassung

D-Latches und D-Flip-Flops



CK = Takteingang (Clock, CK)

D = Dateneingang

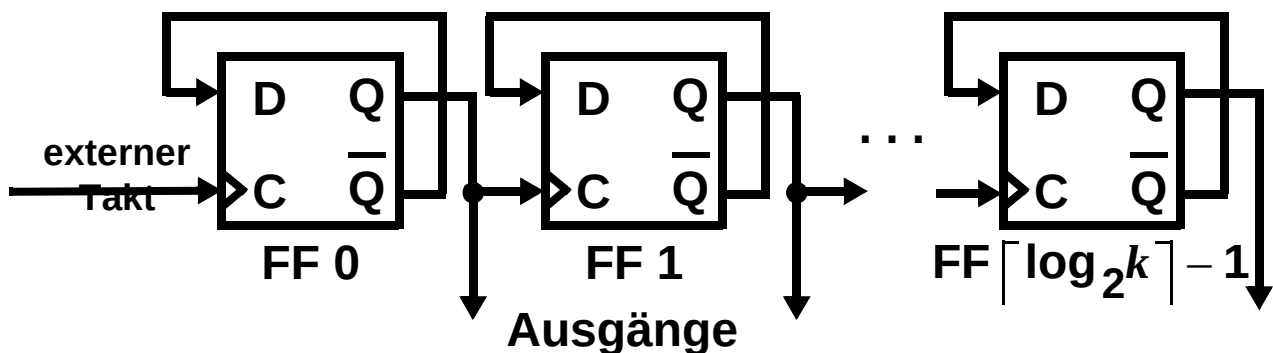
Q = Speicherausgang

- a) 1-Pegel am Takteingang speichert den Inhalt am Dateneingang im D-Latch**
- b) D-Latch mit invertiertem Takteingang**
- c) 1-Impuls beliebiger Länge am Takteingang speichert den Inhalt des Dateneingang im D-Flip-Flop**
- d) D-Flip-Flop mit invertiertem Takteingang**

Modulo-k-Zähler

- Mehrere D-Flip Flops können zu einem sog. Zähler zusammengeschaltet werden
- Modulo-k-Zähler zählen anhand eines äußeren Taktimpulses von 0 bis $k-1$ und beginnen dann von vorne
- Die FF-Ausgänge bilden ein Binärwort (=Zählerzustand)
- Für k verschiedene Zählerzustände, sind $\lceil \log_2 k \rceil$ FFs erforderlich
- Zähler können in den Varianten Synchron- und Asynchrone Zähler realisiert werden

Asynchroner Modulo-k-Zähler



- Durch die Rückkopplung von $\bar{Q}$ -Ausgang zu D-Eingang ändert sich mit jedem Taktimpuls der Q-Ausgang und damit auch Q gemäß: $Q_{\text{neu}} := Q_{\text{alt}}$

- Da $Q, \bar{Q} \in \{0, 1\}$, geht Q mit jedem Taktimpuls in seinen komplementären Zustand $\bar{Q}$ über

=> Q erzeugt die Folge 0,1,0,1,0,... (Anfangszustand von Q sei 0)

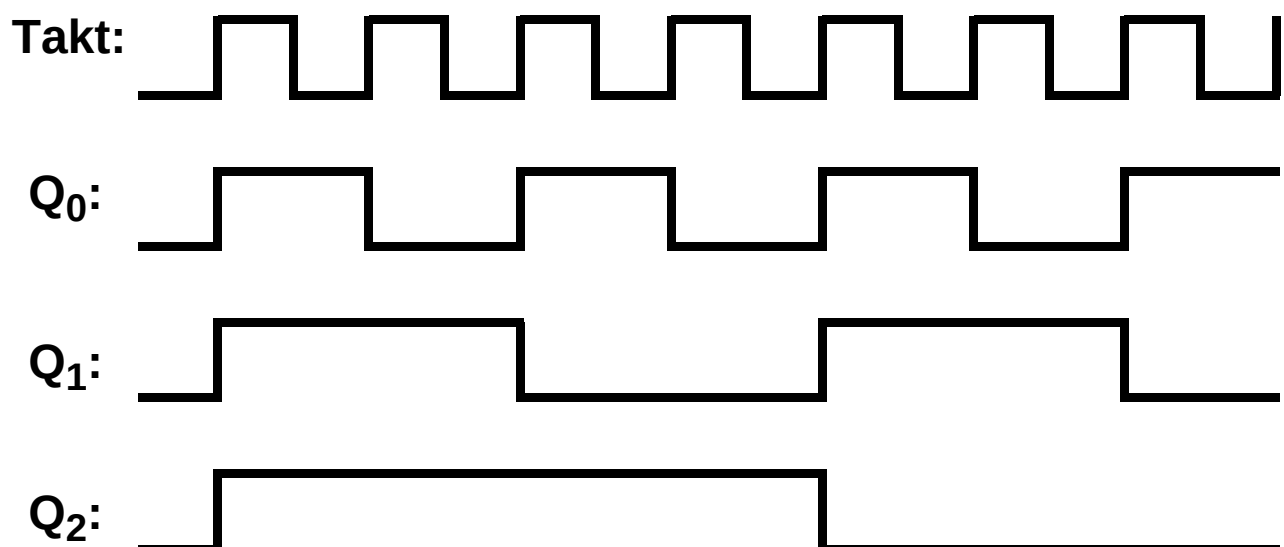
- Die 0/1-Folge von Q_i liefert den Taktimpuls für den Eingang C_{i+1} des nachfolgenden FFs ($0 \leq i \leq k-2$)

- Jede Periode von Q_i (= 1 Hin- und Her-Wechsel) liefert genau einen Taktimpuls für C_{i+1}

- Für jede Periode von Q_i sind seinerseits 2 Taktimpulse bei C_i erforderlich

=> $C_i = 2 C_{i+1}$, d.h. FF_{i+1} ändert seinen Ausgang Q_{i+1} nur halb so schnell wie FF_i Q_i => FF-Kette hat die Funktion eines "Frequenzteilers"

- Beispiel: $k = 3$ und rückwärts zählen; zu Beginn: $Q_i = 0$

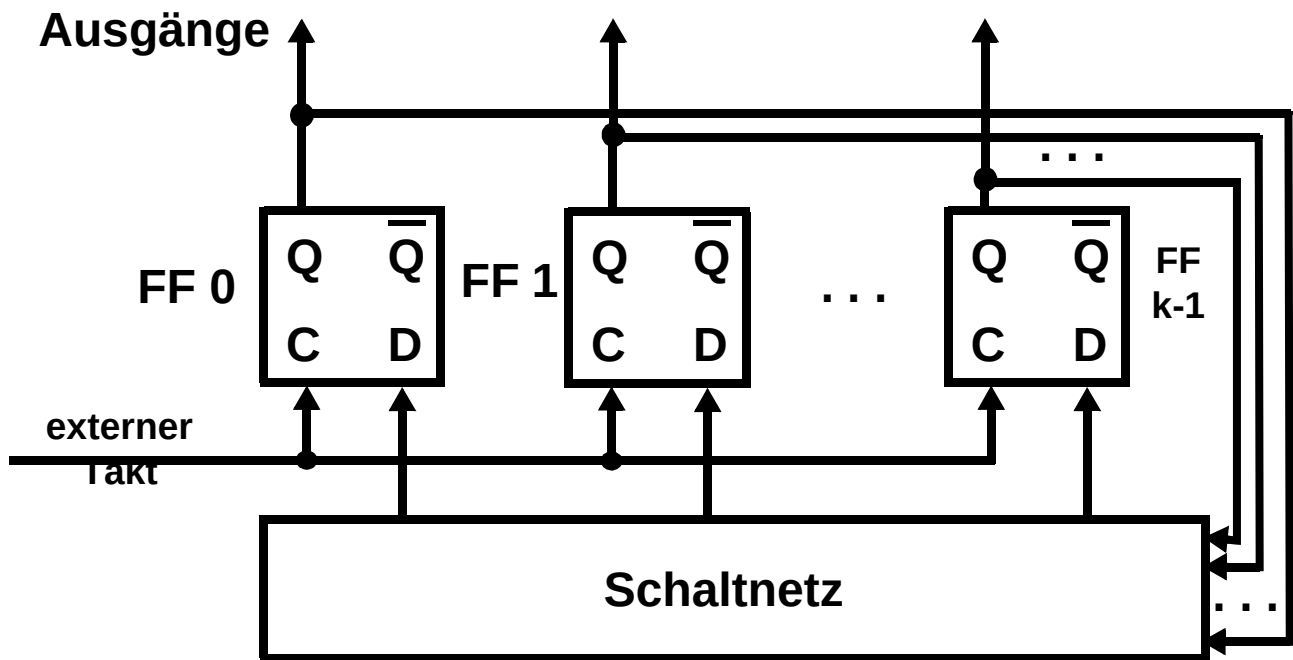


- **Nachteil von Asynchron-Zählern:**
- **Jedes FF hat intern eine Schaltzeit T_s zwischen D-Eingang und Q-Ausgang**
- => Ausgang Q_{k-1} ändert sich erst kT_s Schaltzeiten später als Taktimpuls C_0 , da k FFs in Serie geschaltet sind**
- => FF-Ausgänge ändern sich asynchron zueinander und zum äußeren Takt, daher die Bezeichnung Asynchron-Zähler**
- **Anwendungen von Asynchron-Zählern:**
 - als Frequenzteiler
 - als einfaches Leitwerk in einem Prozessor (nur solange wie kT_s klein gegenüber der Periode des Takts ist)

Synchroner Modulo-k-Zähler

- Zur Vermeidung der Summierung der Schaltverzögerungen werden synchrone Zähler eingesetzt
 - Bei Synchronzählern ändern sich alle FF-Ausgänge gleichzeitig
 - Dazu ist es erforderlich, daß alle FF-Eingänge D_i ebenfalls gleichzeitig mit dem Wert beaufschlagt werden, den der Ausgang Q_i mit dem nächsten Taktimpuls annehmen soll
 - Definition: zum Zeitpunkt $T = n$ sei der Zählerzustand Z^n mit $Z^n = (Q_{k-1}, Q_{k-2}, \dots, Q_0)^n$, ($Z^n =$ „z oben n“)
 - Die Werte D_i für den nachfolgenden Zählerzustand Z^{n+1} müssen einen Zustand vorher (d.h. in Z^n) berechnet und an die Dateneingänge D_i angelegt werden
- => Mit dem jedem Taktimpuls können sich alle FFs gleichzeitig ändern, da die Verkettung $Q_{i-1}D_i$ entfällt
- Zur Vorausberechnung der D_i -Werte ist eine extra Hardware erforderlich
 - Als Hardware wird ein sog. Schaltnetz (=kombinatorische Logik) verwendet

- **Definition:** Ein Schaltnetz besteht aus einer Menge von UND-, ODER und NICHT-Gattern ohne Speicherfunktion. Rückkopplungen, so daß ein FF entsteht, sind verboten.
- **Prinzipschaubild Synchronzähler:**



- Das Schaltnetz bildet eine Rückkopplung von Q_i -Ausgängen zu D_i -Eingängen zur Verarbeitung des jeweiligen Zählerzustandes Z^n zur Zeit $T = n$ für alle n
 - **Definition:** Ein Schaltwerk besteht aus einer Menge von UND-, ODER, NICHT-Gattern und FFs inclusive von Rückkopplungen
- => Ein Schaltwerk hat u.a. eine Speicherfunktion
- => Ein Synchronzähler ist ein Beispiel für ein Schaltwerk

- Die Frage ist, wie das Schaltnetz intern aufgebaut sein muß, um die D_i -Eingänge zu jedem Zeitpunkt t richtig setzen zu können
- Diese Frage wird durch die sog. Schaltnetzsynthese beantwortet

Beispiel: Schaltnetzsynthese für Synchronzähler mit $k=3$

- Schritt 1: Aufstellen der Wahrheitstabellen für Zählerzustand Z^n und Z^{n+1}

Z^n			Z^{n+1}		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

- Dabei wird die Z^n -Tabelle als Eingangsinformation für das Schaltnetz und die Z^{n+1} -Tabelle als Ausgangsinformation verwendet: $Z^n \rightarrow Z^{n+1}$

- Mit jedem Takt gibt es einen Übergang $z^n \rightarrow z^{n+1}$
- Das Schaltnetz berechnet die sog. Übergangsfunktion so, daß gilt $(D_2, D_1, D_0)^n := (Q_2, Q_1, Q_0)^{n+1}$ (Gl. 1)

Schritt 2: Bestimmen der Übergangsfunktion entweder durch Aufstellen der Booleschen Gleichungen oder der Karnaugh-Diagramme gemäß (Gl. 1)

$$D_0 = \overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0} + \overline{Q_2} \cdot Q_1 \cdot \overline{Q_0} + Q_2 \cdot \overline{Q_1} \cdot \overline{Q_0} + Q_2 \cdot Q_1 \cdot \overline{Q_0}$$

$$D_1 = \overline{Q_2} \cdot \overline{Q_1} \cdot Q_0 + \overline{Q_2} \cdot Q_1 \cdot \overline{Q_0} + Q_2 \cdot \overline{Q_1} \cdot Q_0 + Q_2 \cdot Q_1 \cdot \overline{Q_0}$$

$$D_2 = \overline{Q_2} \cdot Q_1 \cdot Q_0 + Q_2 \cdot \overline{Q_1} \cdot \overline{Q_0} + Q_2 \cdot \overline{Q_1} \cdot Q_0 + Q_2 \cdot Q_1 \cdot \overline{Q_0}$$

Karnaugh-Diagramme:

Für D_0 :

		Q_0	
$Q_2 Q_1$		0	1
00		1	0
01		1	0
10		1	0
11		1	0

Für D_1 :

		$Q_1 Q_0$			
		00	01	10	11
Q_2	0	0	1	1	0
	1	0	1	1	0

Für D_2 :

		$Q_1 Q_0$			
		00	01	10	11
Q_2	0	0	0	0	1
	1	1	1	1	0

- **Schritt 3:** Optimieren der Gleichungen bzw. Zusammenfassen von Zuständen in den Karnaugh-Diagrammen zur Schaltnetzvereinfachung

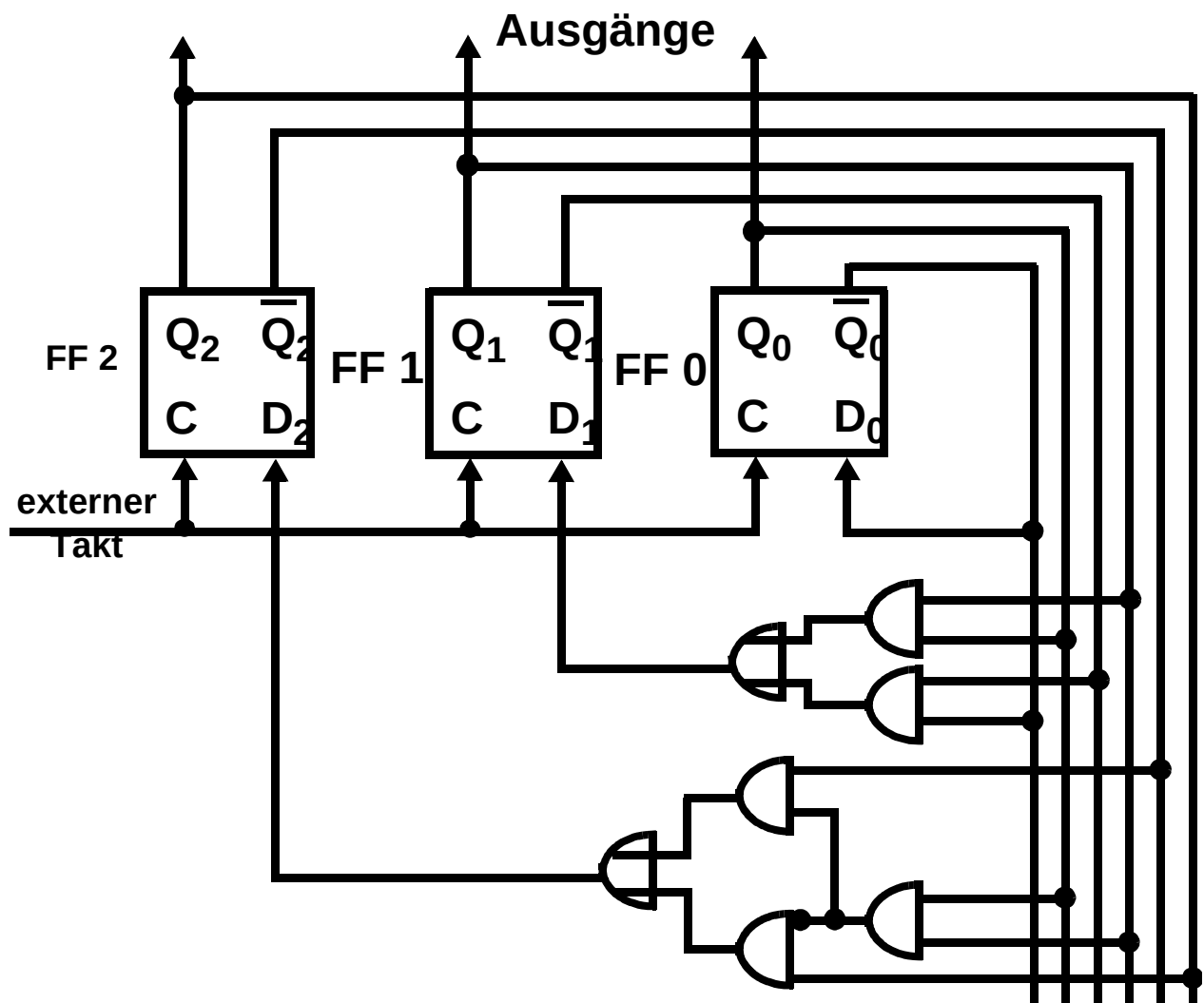
Resultat der Vereinfachung im Beispiel:

$$D_0 = \overline{Q_0}$$

$$D_2 = \overline{Q_2} \cdot Q_1 \cdot Q_0 + Q_2(\overline{Q_1} \cdot \overline{Q_0})$$

$$D_1 = \overline{Q_1} \cdot Q_0 + Q_1 \cdot \overline{Q_0}$$

- Ergebnis der Schaltnetzsynthese:



- Anwendungen von Synchron-Zählern:

- als Frequenzteiler
- als einfaches Leitwerk in einem Prozessor mit hohen Taktfrequenzen
- **Nachteil:** nicht immer ist die natürliche Zählreihenfolge $0,1,\dots,k-1$ gewünscht sondern eine Permutation davon

Synchron-Zähler mit permutierten Zählfolgen

- **Fakt:** Übergangsfunktion eines Synchronzählers wird mittels Tabellen für die Zählerzustände Z^n spezifiziert (für alle n)
 - Jede Übergangsfunktion $Z^n \rightarrow Z^{n+1}$ mit endlicher Grundmenge ist mittels einer Tabelle spezifizierbar
 - Diese kann mit Hilfe der gezeigten Schaltnetzsynthese in eine Hardware umgesetzt werden
- => Für jede beliebige endliche Übergangsfunktion gibt es ein dazugehöriges Schaltnetz**

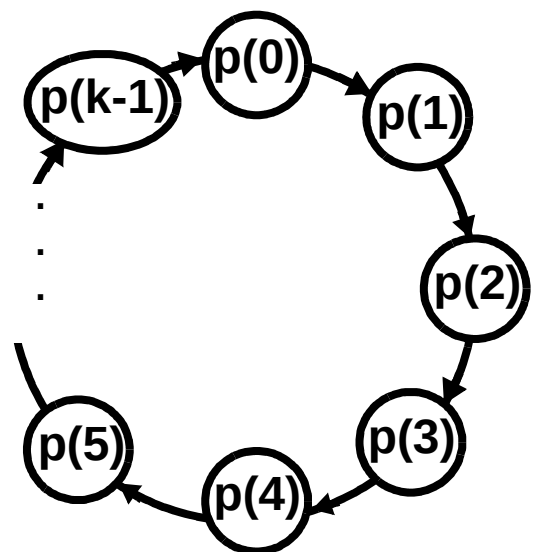
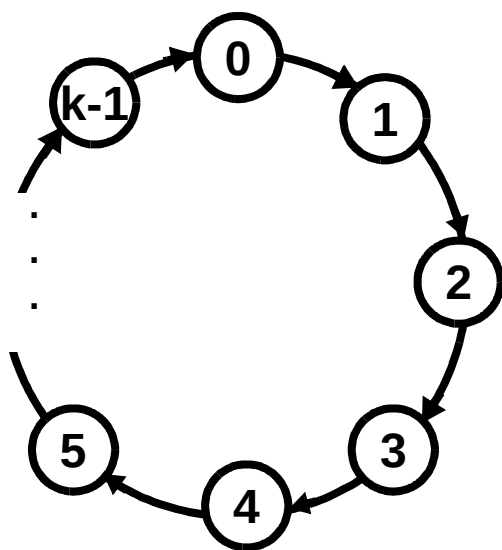
Beispiel: Ein Synchronzähler soll so konstruiert werden, daß er in einer permutierten Reihenfolge zählt

- Sei $p(0), p(1), \dots, p(k-1)$ eine permutierte Zählweise mit $p(i)$ als umkehrbar eindeutige Funktion (= Permutationsfunktion)

- Dann ersetze für alle i die Tabelleneinträge i von Z^n und Z^{n+1} durch $p(i)$, und erhalte so das Schaltnetz für die permutierte Zählweise
- => Die Schaltnetzsynthese für beliebige Übergangsfunktionen erfordert dieselben Schritte wie beim Zähler der Folge $0,1,\dots,k-1$

Endliche Automaten

- Zum besseren Vergleich verschiedener Zähler dient das formale Hilfsmittel der endlichen Automaten
- **Definition:** Ein endlicher Automat (EA) ist ein Graph aus Knoten und gerichteten Kanten, wobei die Knoten die Zählerzustände und die Kanten die Übergänge zwischen den Zuständen repräsentieren
- Beispiel: EAs von Zählern für natürliche und permutierte Zählreihenfolge

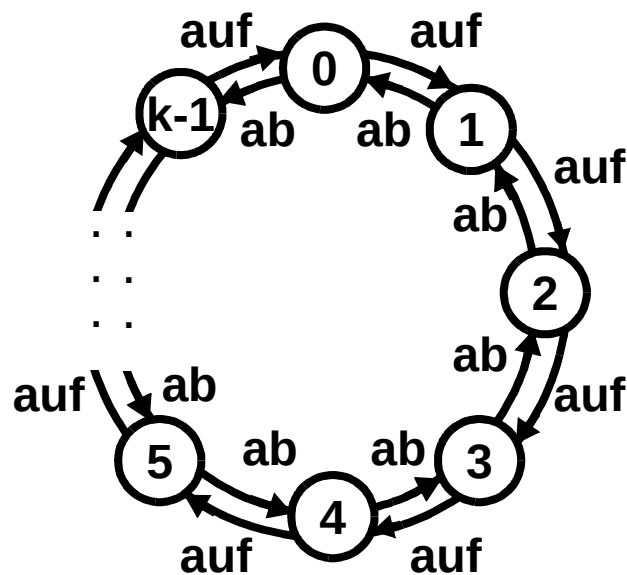


- **Definition:** Die textuellen Bezeichnungen der Knoten des Graphen heißen **Knotenmarkierungen**
- Ebenso sind sog. **Kantenmarkierungen** zulässig
- Kantenmarkierungen im EA-Graphen haben die Bedeutung eines "endlichen Automaten mit Eingabe"

Endliche Automaten mit Eingabe

- Die EAs der gezeigten Zähler wechseln den Zustand mit jedem Taktimpuls
 - Sie reagieren damit auf kein anderes äußeres Signal als den Takt
 - EAs können dahingehend **erweitert** werden, daß nur dann ein Übergang zwischen zwei benachbarten Knoten möglich ist, wenn zusätzlich **eine Bedingung** erfüllt ist
 - Als Bedingung für das "Weiterschalten" können externe Signale verwendet werden
- => Die externen Signale dienen als Eingabe in den EA**

- Beispiel: Zähler, der aufwärts und abwärts zählen kann

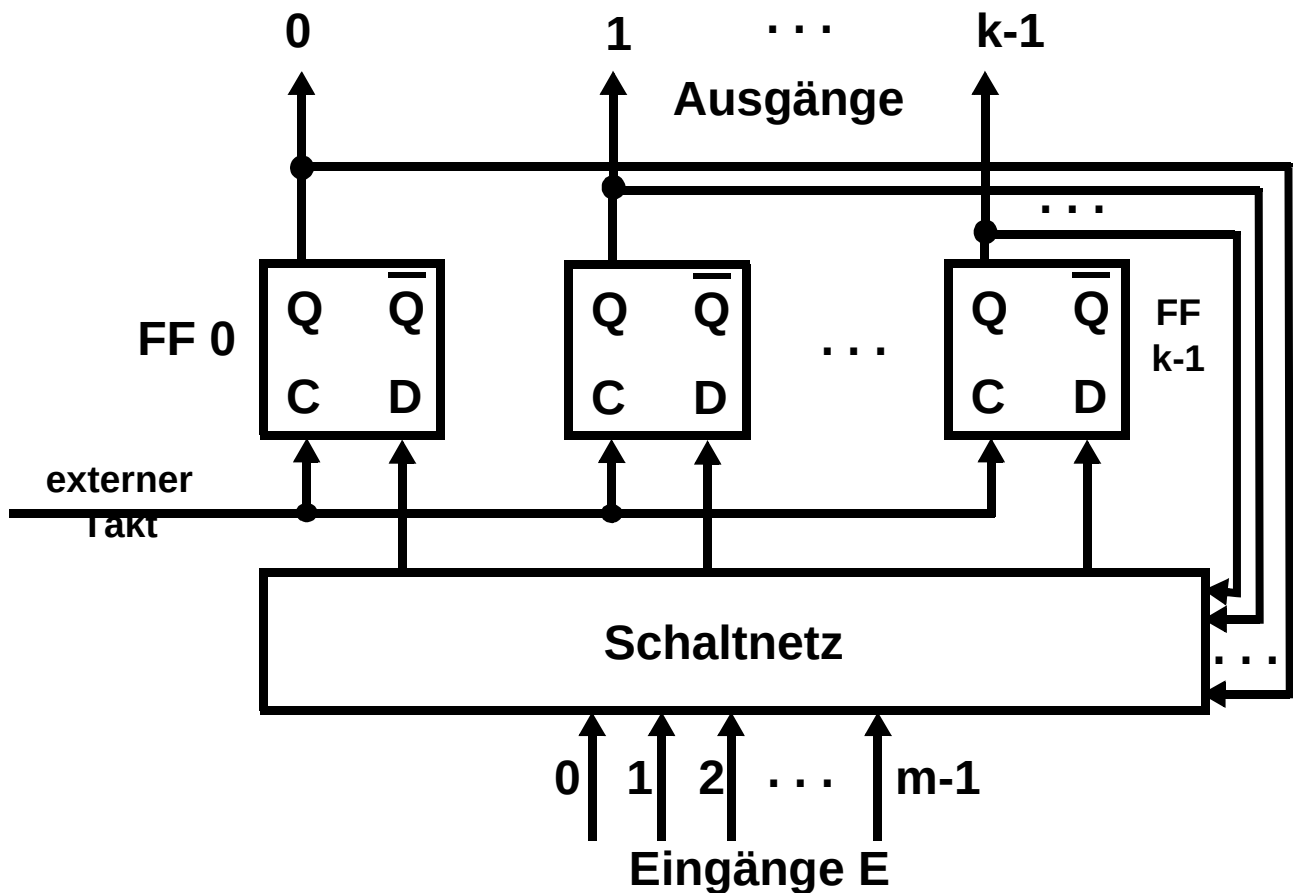


- Der EA des Auf-/Ab-Zählers wertet ähnlich wie ein Lift die beiden Signale "auf" und "ab" zur Steuerung aus
- Sind beide gesetzt oder keines davon, wird kein Zustandsübergang durchgeführt

Implementierung von endlichen Automaten mit Eingabe

- EAs mit Eingabe können ebenfalls durch einen Synchronzähler mit Schaltnetz implementiert werden
 - Dazu muß das Schaltnetz nicht nur die Zählerzustände Z, sondern auch die Eingangssignale E auswerten
- => Ein Synchronzähler mit Schaltnetz, das über zusätzliche Steuereingänge verfügt, implementiert einen EA mit Eingabe

- Prinzipschaubild Synchronzähler mit Steuereingängen:



Schaltnetzsynthese beim Synchronzähler mit Steuereingängen

- Die Frage beim Synchronzähler mit Steuereingängen ist, wie die Schaltnetzsynthese abläuft
- Dazu müssen die Tabellen für die Zählerzustände Z^n und Z^{n+1} um neue Spalten für die Werte der Eingangssignale E erweitert werden

=> Die Übergangsfunktion wird durch E, Z^n und Z^{n+1} spezifiziert: $(E, Z^n) \rightarrow Z^{n+1}$

- Wie zuvor erfüllt dann die Übergangsfunktion die Gleichung $(D_{k-1}, \dots, D_1, D_0)^n = (Q_{k-1}, \dots, Q_1, Q_0)^{n+1}$ für alle n
- Ein Synchronzähler mit Steuereingängen kann durch tabellarische Spezifikation der Übergangsfunktion $(E, Z^n) \rightarrow Z^{n+1}$ jeden endlichen Automaten mit Eingabe implementieren

Endliche Automaten mit Ein- und Ausgabe

- Ein EA mit Ein-/und Ausgabe kann ein Ausgangssignal erzeugen, das mehr ist als die Ausgabe seines momentanen Zustandswertes
- Es müssen zwei Varianten unterschieden werden:

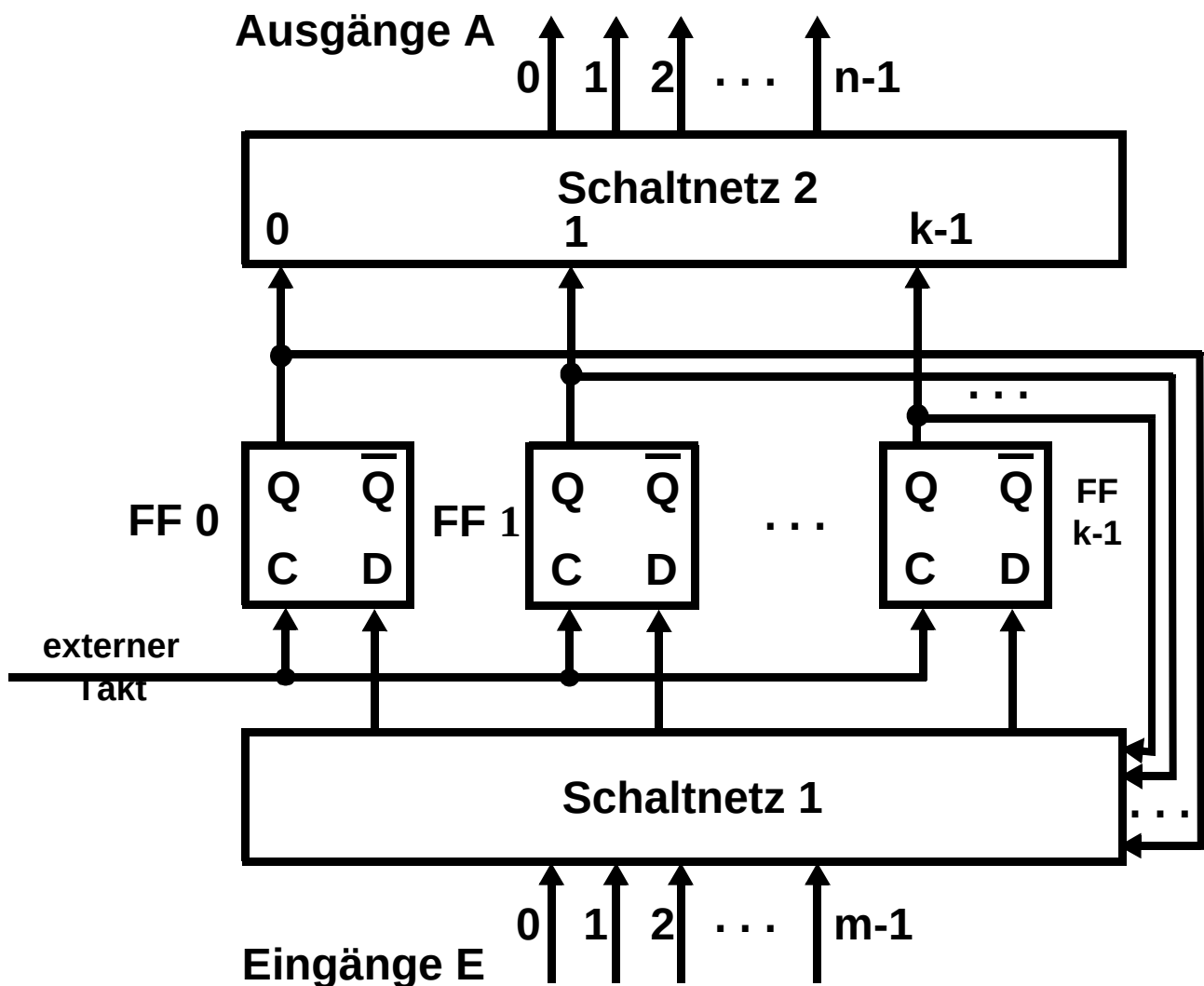
Mealy-Automat und Moore-Automat

- Beim Moore-Automat ergibt sich der Ausgabewert als Funktion des jeweiligen Zustandswertes.
 - Die Ausgabe ist synchron zum Takt
 - Beim Mealy-Automat ergibt sich der Ausgabewert als Funktion des Zustandswertes und der Eingabewerte
 - Die Ausgabe kann zu jedem Zeitpunkt auf Eingabeänderungen reagieren
- => Die Ausgabe ist asynchron zum Takt

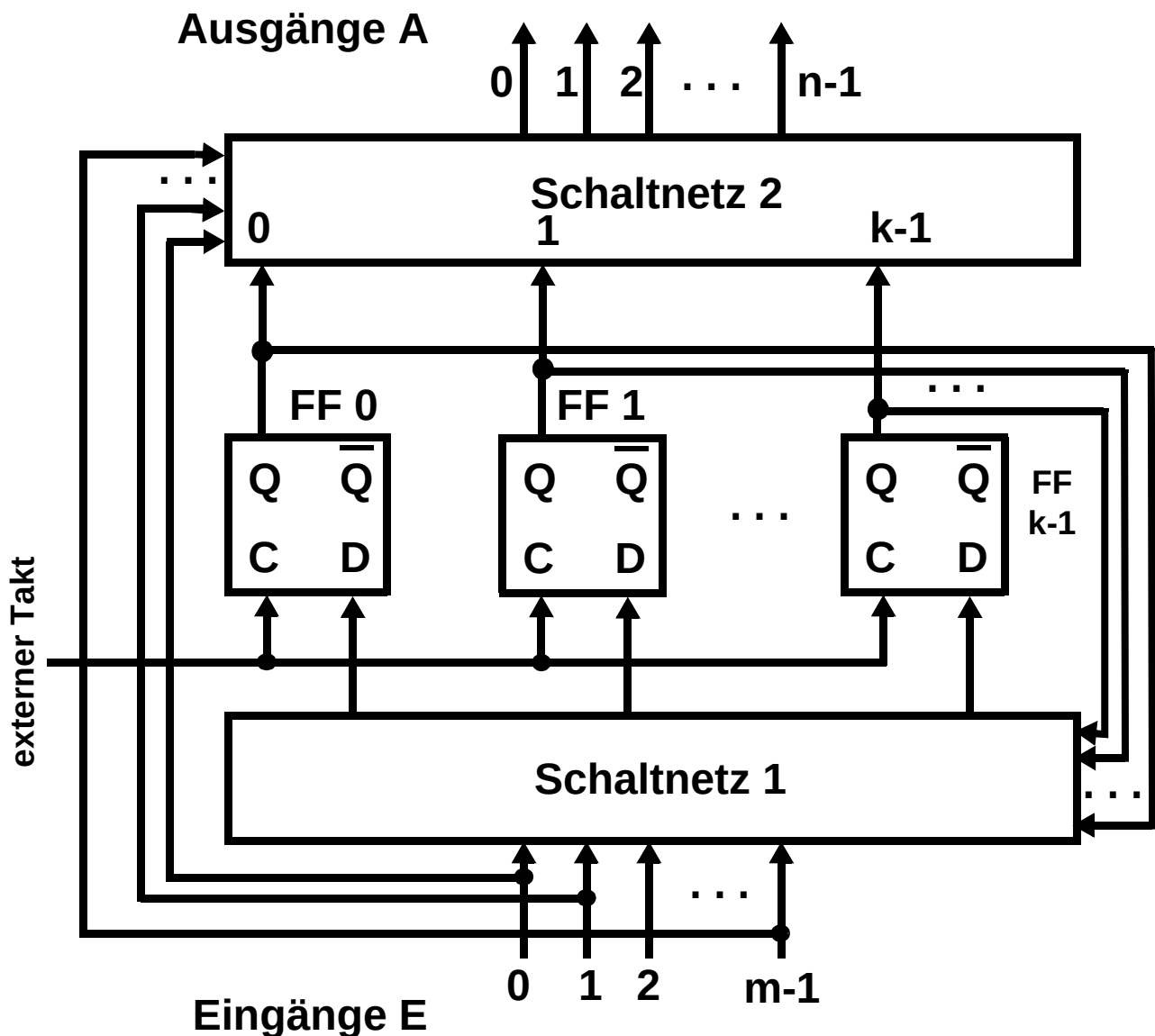
- Formal wird der Graph eines EA mit Ein-/und Ausgabe um eine Knotenmarkierung erweitert, die den Ausgabe-wert darstellt

Implementierung von endlichen Automaten mit Ein- und Ausgabe

- EAs mit Ein- und Ausgabe können durch einen Synchro-nzähler mit extra Schaltnetz implementiert werden
- Prinzipschaubild Moore-Automat:



- Die Synthese des Ausgabeschaltnetzes erfolgt ähnlich zu der des Eingabeschaltnetzes
- Es wird tabellarisch die Ausgabefunktion als Abbildung des Zählerzustands Z auf die n Ausgänge spezifiziert
- Daraus werden die Booleschen Gleichungen oder die Karnaugh-Diagramme für jeden Ausgang aufgestellt und anschließend optimiert
- Prinzipschaubild Mealy-Automat:



- Durch die direkte Verbindung von Eingängen E des Synchronzählers zum 2. Schaltnetz ergibt sich eine "Bypass"-Funktion
- Die Bypass-Funktion erlaubt, auf jede Änderung am Eingang E unmittelbar mit einer Änderung am Ausgang A zu reagieren

=> Mealy-Automat arbeitet mit asynchroner Ausgabe

- Vorteil von Mealy- gegenüber Moore-Automat:

- Mealy-A. enthält Moore-A. als Teilmenge

=> alles, was Moore-A. kann, kann auch Mealy-A.

- Mealy-A. reagiert schneller auf Eingaben als Moore-A.

- Nachteil von Mealy-A.

- asynchrone Ausgaben sind beim Zusammenspiel mehrerer EAs mit Ein- und Ausgabe schlechter zu beherrschen als synchrone, d. h. die Kopplung mehrerer Mealy-A. ist i. A. schwierig

Anwendungen von EAs mit Ein-/Ausgabe

- Als Steuerwerk in einem Prozessor
- Als Herzstück von Steuerungen beliebiger Art
- Allgemein: Zur Implementierung von Algorithmen einer bestimmten Klasse komplett in Hardware

Berechnungsstärke der EAs

- Aus der theoretischen Informatik ist bekannt, daß viele (aber nicht alle!) Algorithmen formal mit Hilfe von endlichen Automaten spezifiziert und damit von EAs mit Ein-/Ausgabe implementiert werden können
- Die mittels EAs spezifizierten Algorithmen sind alternativ auch mit Hilfe einer "regulären Sprache" angebar = deterministisch kontextfreie Sprache
- Berechnungsstärker als die endlichen Automaten sind (in dieser Reihenfolge):
 - endliche Automaten mit Kellerspeicher (sog. Kellerautomaten) => kontextfreie Sprachen
 - Turingmaschine mit endlichem Speicherband => kontext-sensitive Sprachen
 - Turingmaschine mit unendlichem Speicherband => Turing-entscheidbare/Turing-erkennbare Sprachen
- Jeder dieser Maschinen spezifiziert eine Klasse von Sprachen, die die jeweils vorangehende umfaßt (= Chomsky-Hierarchie)

Berechnungsstärke von EAs mit Ein- und Ausgabe und zusätzlichem Speicher

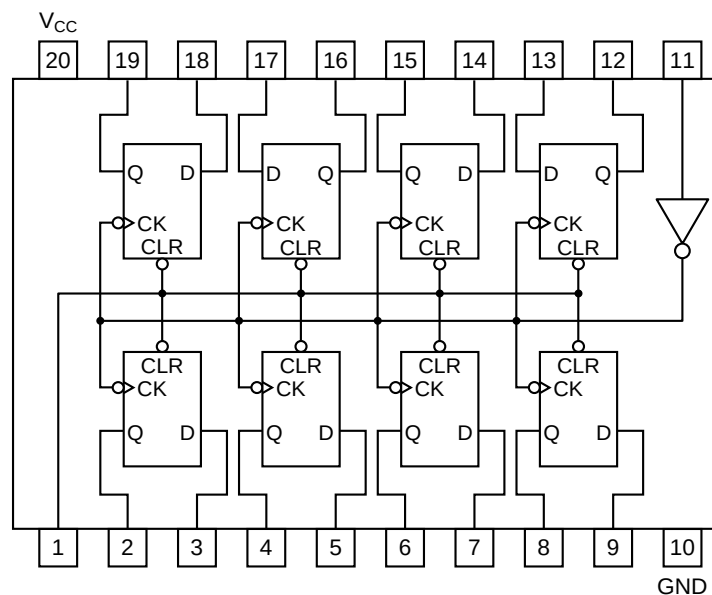
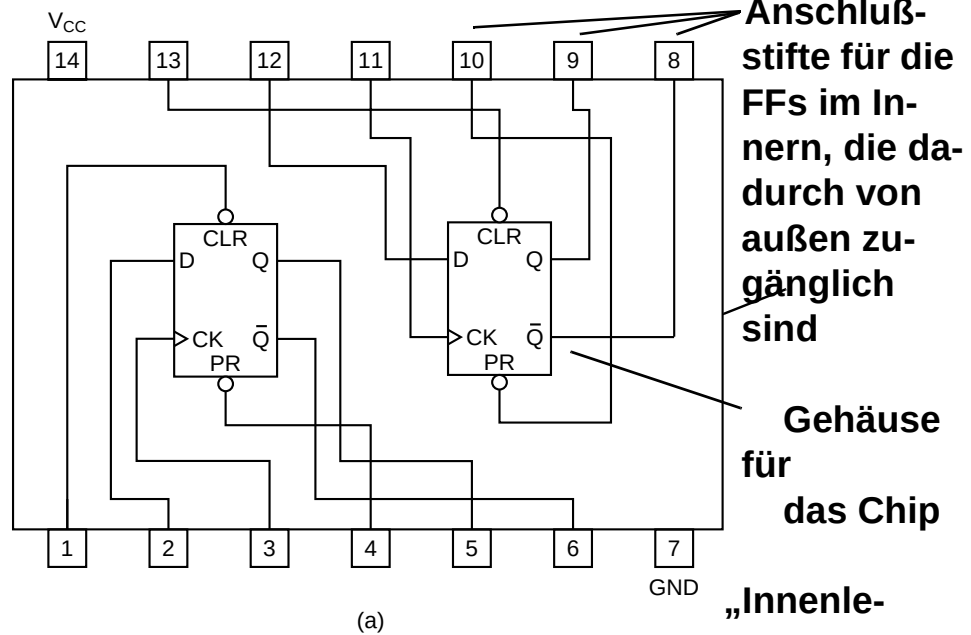
- ein EA mit Ein- und Ausgabe sowie zusätzlichem Speicher entspricht von der Berechnungsstärke her einem üblichen Rechner
 - Das zusätzliche Rechenwerk, über das der Rechner verfügt, trägt nichts zu dessen Berechnungsstärke bei
 - ein EA mit Ein- und Ausgabe implementiert die Steuerung, die für einen Kellerautomaten oder eine Turingmaschine benötigt wird
 - ein Speicher mit wahlfreiem Zugriff (RAM) implementiert das endliche Speicherband einer entsprechenden Turingmaschine
- => ein Rechner hat die Berechnungsstärke einer Turingmaschine mit endlichem Speicherband**
- der Kellerautomat ist berechnungsschwächer als ein üblicher Rechner, weil dessen Speicher hinsichtlich des Zugriffs eingeschränkt ist (nur jeweils das oberste Stack-Element)

Registerspeicher

**(= Vektor von 1-Bit-FFs zur Speicherung
von Datenworten in der CPU)**

Beispiele für 2-Bit- und 8-Bit-Register

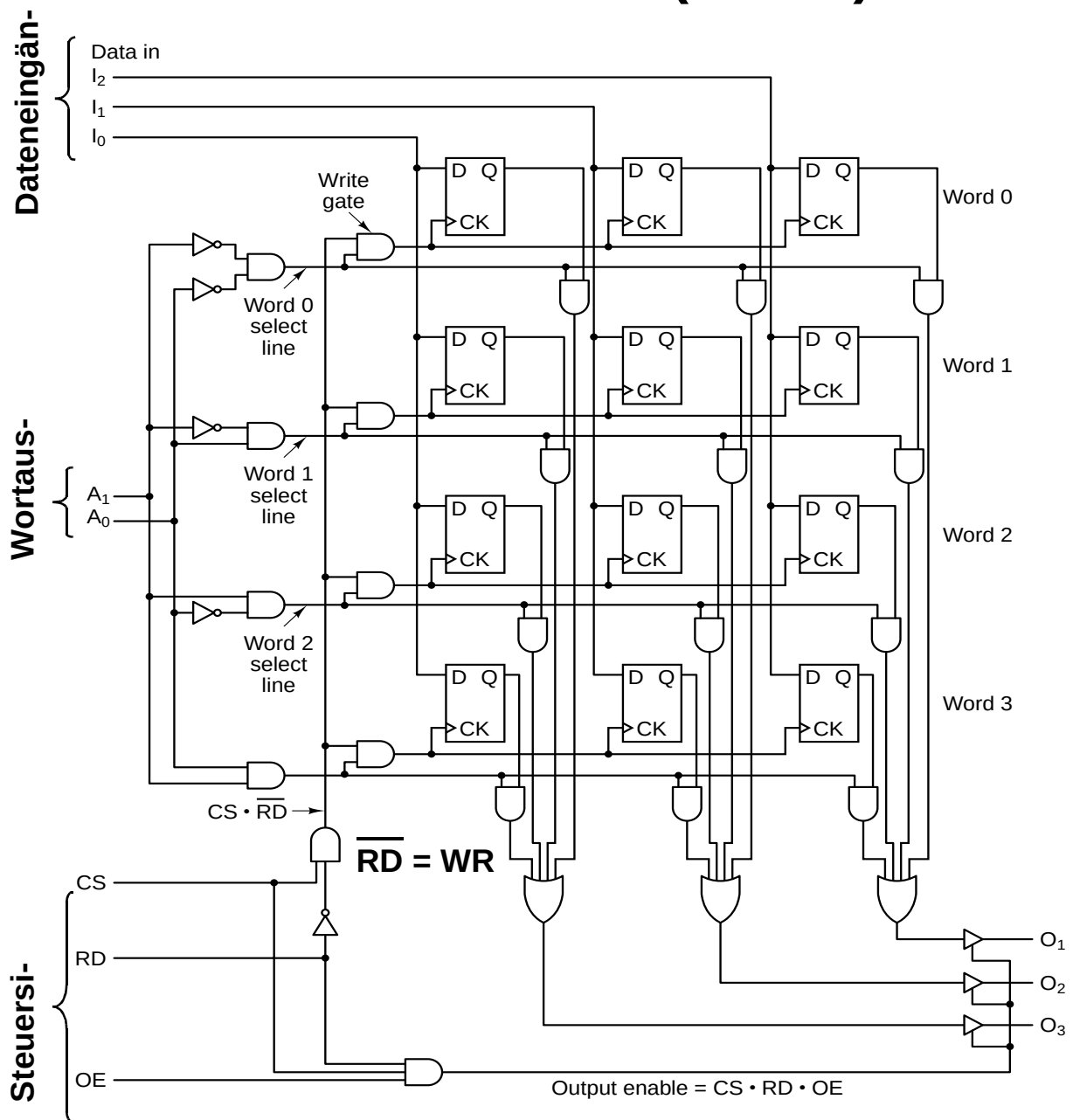
Beispiele für Register, die als integrierte Schaltungen kommerziell erhältlich sind. Es gilt folgende Sprachregelung: Chip + Gehäuse für das Chip = integrierte Schaltung = integrated Circuit = IC



- Aus mehreren D-Flip-Flops wird ein Register aufgebaut. (Ein Register speichert mehrere Bits, sog. Bitvektor)
- Die gezeigten Register sind in einen Baustein (Chip) mit 14 bzw. 20 Anschlußstiften integriert und kommerziell erhältlich (Serie 7400 von Texas Instruments)
- Die Eingänge PR = Preset (setzt Q auf 1) und CLR = Clear (setzt Q auf 0) sind invertiert = „active low“

chip = Splitter, Span; hier: kleines Blättchen aus Silizium

Statisches RAM (SRAM)

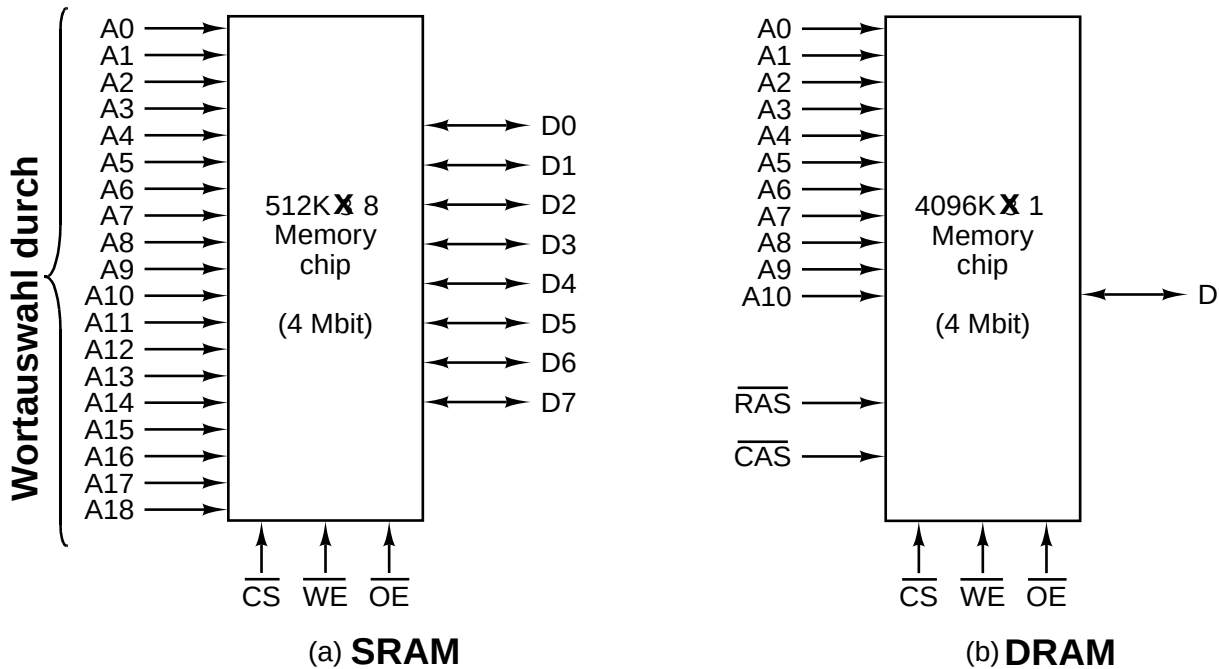


- mehrere Register speichern eine Matrix aus Bits (Jede Registerzeile heißt „Wort“)

Im Beispiel werden 4 Worte zu je 3 Bit gespeichert (I=Dateneingang, O=Datenausgang, A0A1=Wortauswahl, CS=Bausteinaktivierung, OE=Wort ausgeben)

I = Input, O = Output, CS = chip select; OE = output enable
RAM = Random Access Memory = Schreib-/Lesespeicher

SRAM und DRAM (Dynamisches RAM)



- Speicher für 512 K x 8-Bit bzw. 4096 K x 1-Bit-Worte
- A_0 - A_{18} = 18 Adressbits zur Wortauswahl im SRAM. A_0 - A_{10} im DRAM stellen gemultiplexte Adressbits dar.
- D0-D7 = Datenein- und ausgänge (bidirektional!) bzw. D alleine
- $\overline{CS}$ = invertierter Chip Select (Bausteinaktivierung)
- $\overline{OE}$ = invertierter Output Enable (Wortausgabe)
- $\overline{WE}$ = invertiertes Write Enable (Schreibaktivierung)

Der Strich bei einem Bezeichner heißt, dass das betreffende Signal invertiert wird. Das Signal wird durch Anlegen einer „0“ aktiviert.

Unterschiede zwischen SRAM und DRAM

- SRAM speichert jedes Bit in einem eigenen Flip Flop
 - DRAM speichert jedes Bit in einem sehr kleinen Kondensator: keine el. Ladung = „0“, Ladung = „1“
 - die Kondensatoren zur Bit-Speicherung verlieren ihre Ladung aus technischen Gründen rasch (nach ca. 10 ms).
 - zur dauerhaften Bitspeicherung muß bei jeder „1“ die Ladung nach ca. 1-2 ms immer wieder nachgefüllt werden = sog. „refresh“
 - wegen der notwendigen refresh-Zyklen spricht man von dynamischem RAM
 - Vorteil von DRAMs gegenüber SRAMs: schlechte Kondensatoren lassen sich viel leichter und viel dichter auf einem Chip integrieren als Flip Flops, die aus mehreren Transistoren bestehen
 - um den Preis weiter zu reduzieren, spart man durch Multiplexen der Adreßbits Anschlußstifte
 - z.B. ergeben 10 Adreßanschlußstifte 20 Adreßbits, die zu je 10 Bits in 2 Schritten an die Stifte angelegt werden
- => Preis von DRAMs ist niedriger und Speicherkapazität höher (ca. 4-fache Kapazität)

Die wichtigsten Register in einer CPU

- 1.) Der Befehlszeiger (PC, program counter)
Er enthält die Hauptspeicheradresse des jeweils auszuführenden Befehls.
- 2.) Das Instruktionsregister (JR, instruction register)
Es enthält den jeweils auszuführenden Befehl, nachdem dieser aus dem Hauptspeicher gelesen wurde.
- 3.) Das Speicheradressenregister (MAR, memory address register)
Es enthält die Hauptspeicheradresse der Zelle, die gerade gelesen oder geschrieben wird.
- 4.) Das Speicherdatenregister (MDR, memory data register)
Es enthält den Inhalt der Zelle, deren Adresse im MAR steht.
(= MAR enthält die Adresse einer Zelle,
MDR enthält den Inhalt dieser Zelle)

Weitere wichtige Register der CPU

1.) Der Akkumulator (ACCU)

Er dient als temporärer Zwischenspeicher der ALU und enthält Resultatwerte der ALU.

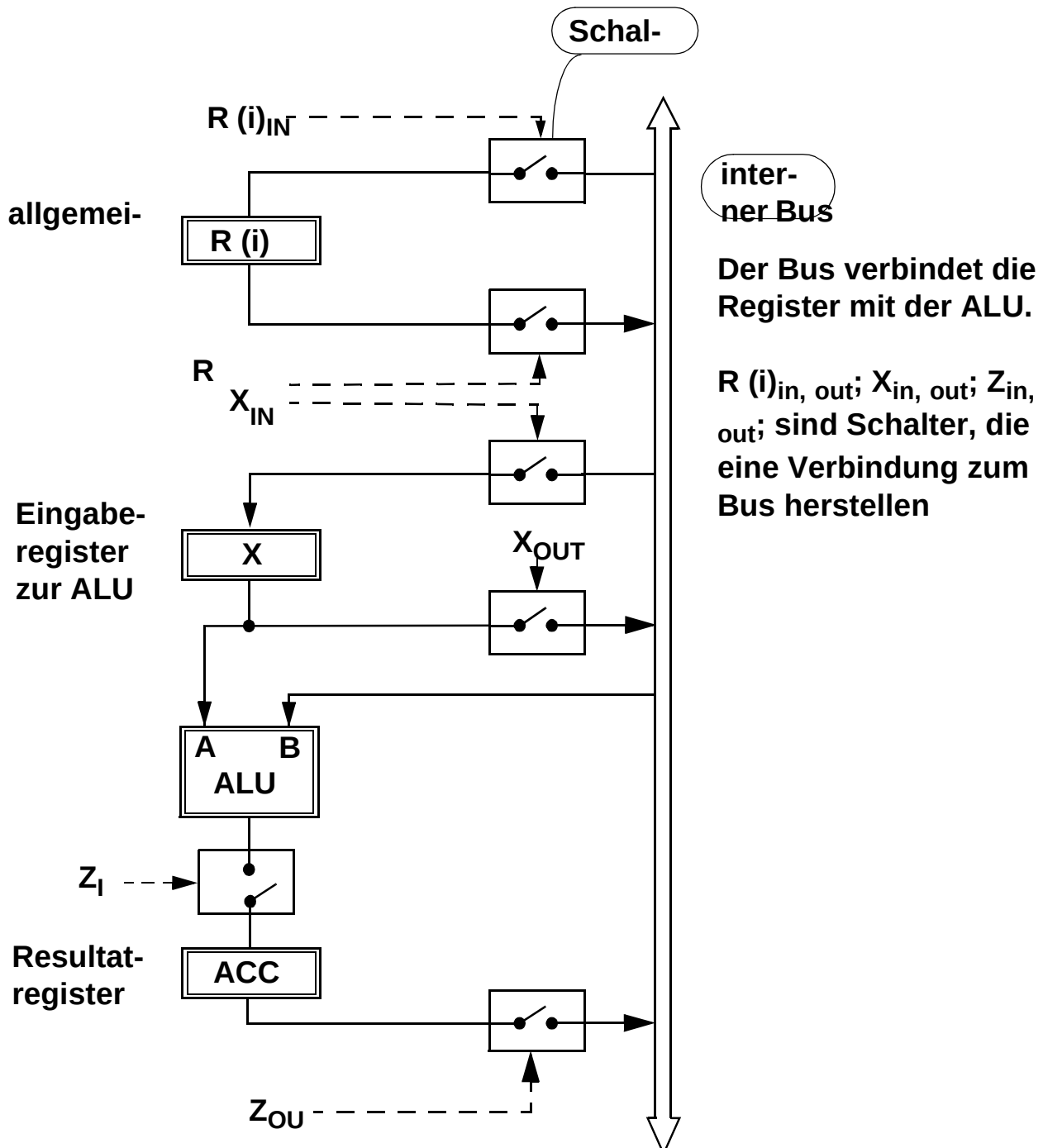
2.) Das ALU-Eingangsregister

Es dient ebenfalls als temporärer Zwischenspeicher, um ein Operand für die ALU aufzunehmen. Der 2. ALU-Operand, der z. B. für die Operation $A + B$ nötig ist, kommt entweder von einem 2. ALU-Eingangsregister oder direkt vom ACCU.

3.) Allgemeine Rechenregister $R_0, R_1, \dots, R_{n-1}$

4.) Spezialregister wie z. B. Stack Pointer (Stapelzeiger)

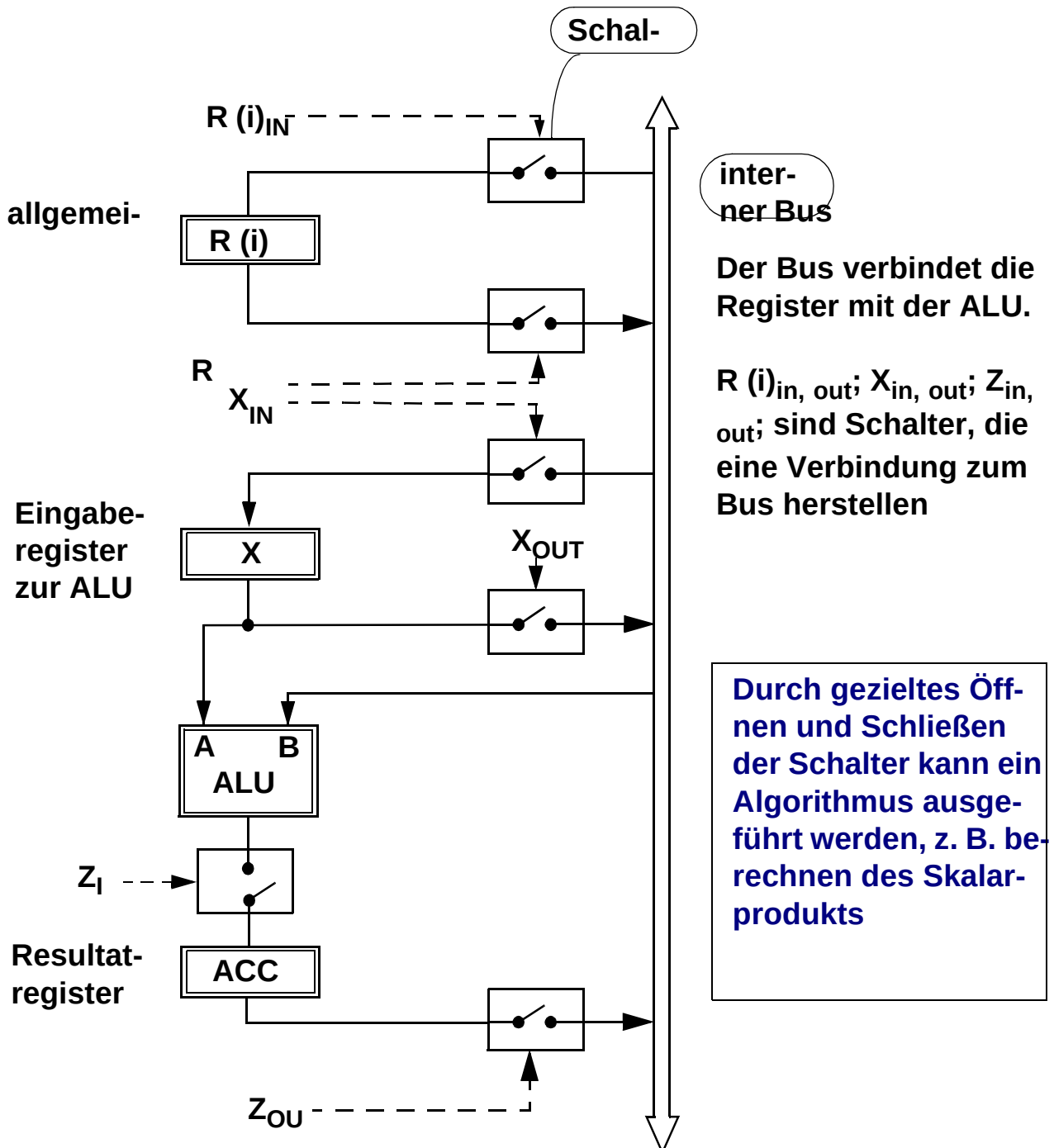
Ein wichtiger Teil eines Prozessors (CPU): die ALU mit Registern



- Alle Signale $R(i)_{in}$, $R(i)_{out}$, X_{in} , X_{out} , Z_{in} , Z_{out} werden gesteuert durch eine Steuereinheit (= Leitwerk, Control unit)
- Das Leitwerk gibt an, wann die verschiedenen Schalter geschlossen werden sollen.

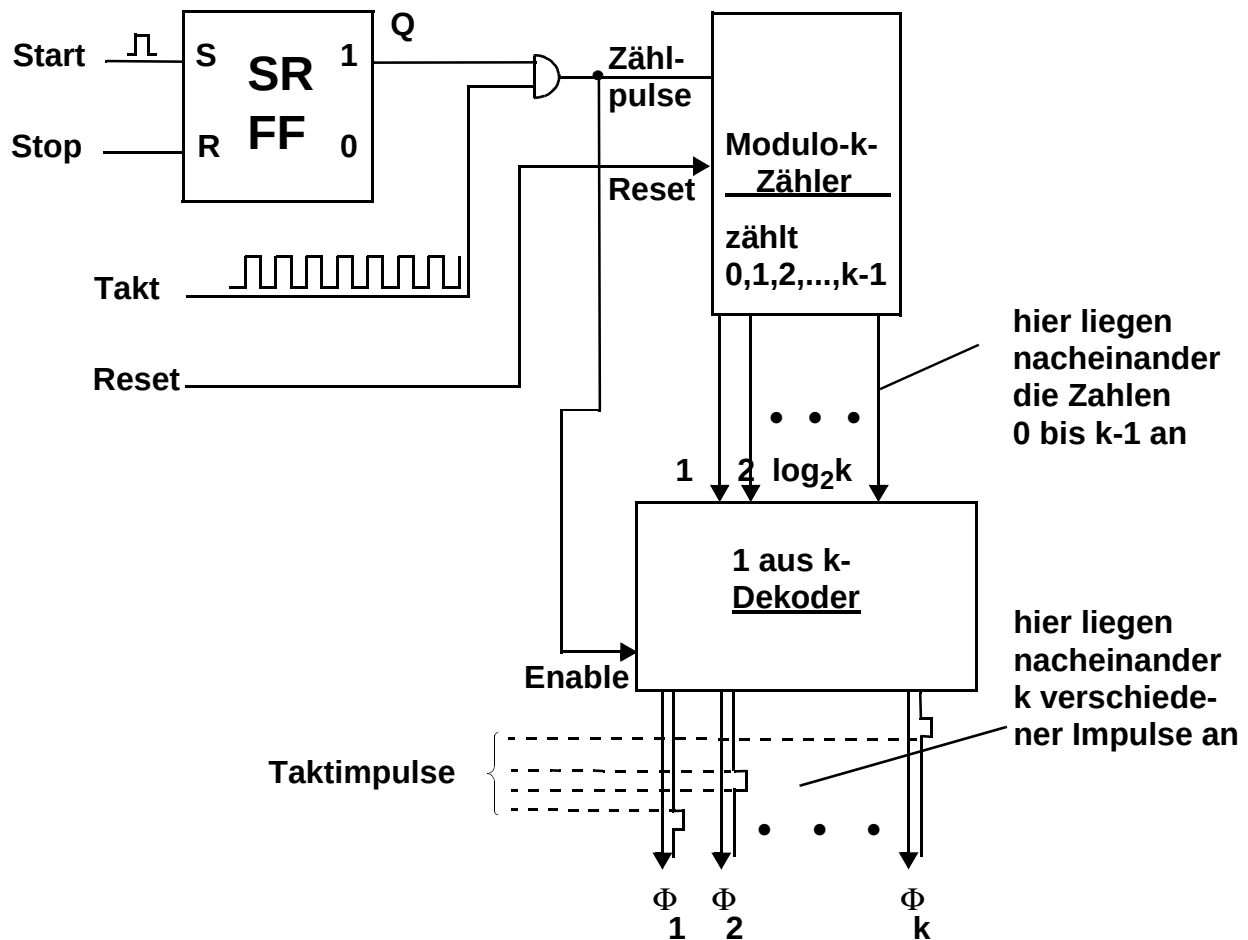
Das Leitwerk (Steuerwerk) eines Rechners

Wozu dient das Steuerwerk? (erläutert am Beispiel einer ALU mit Registern)



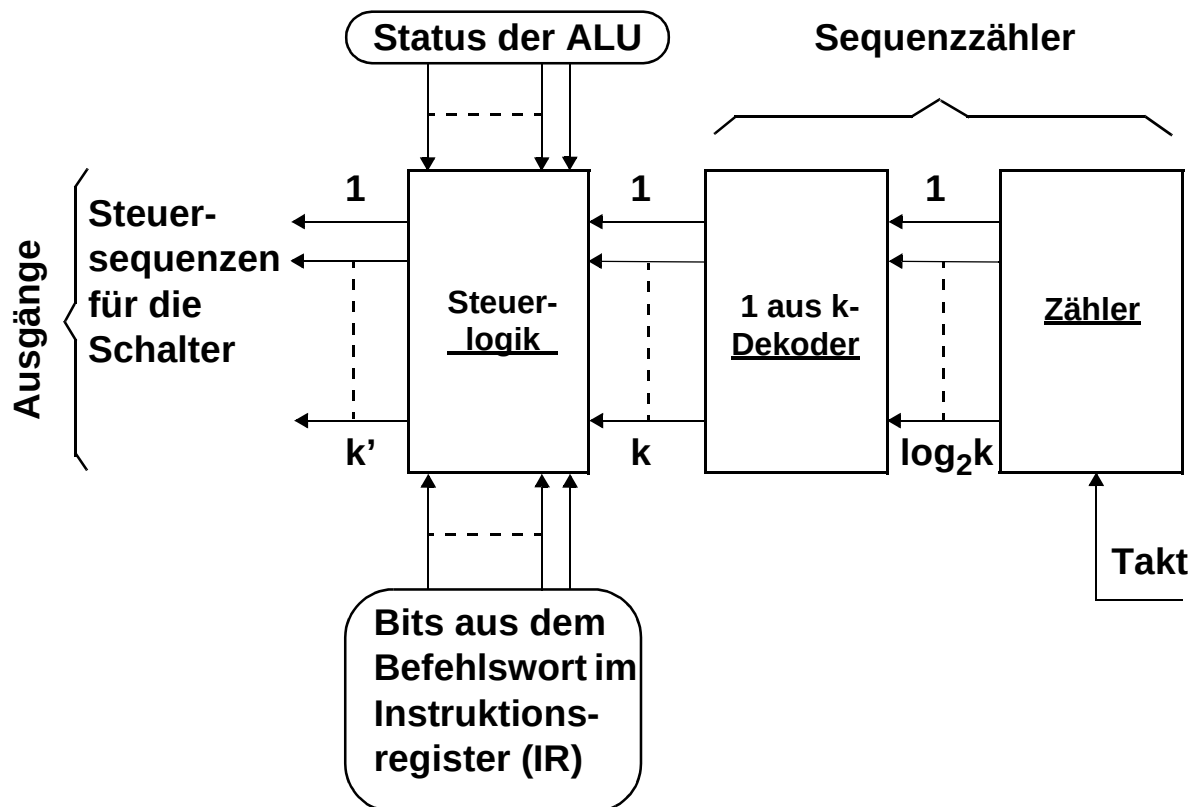
- Alle Signale $R(i)_{in}$, $R(i)_{out}$, X_{in} , X_{out} , Z_{in} , Z_{out} werden gesteuert durch eine Steuereinheit (= Leitwerk, control unit)
- Das Leitwerk gibt an, wann die verschiedenen Schalter geschlossen werden sollen

Ein einfaches Leitwerk: der Sequenzzähler



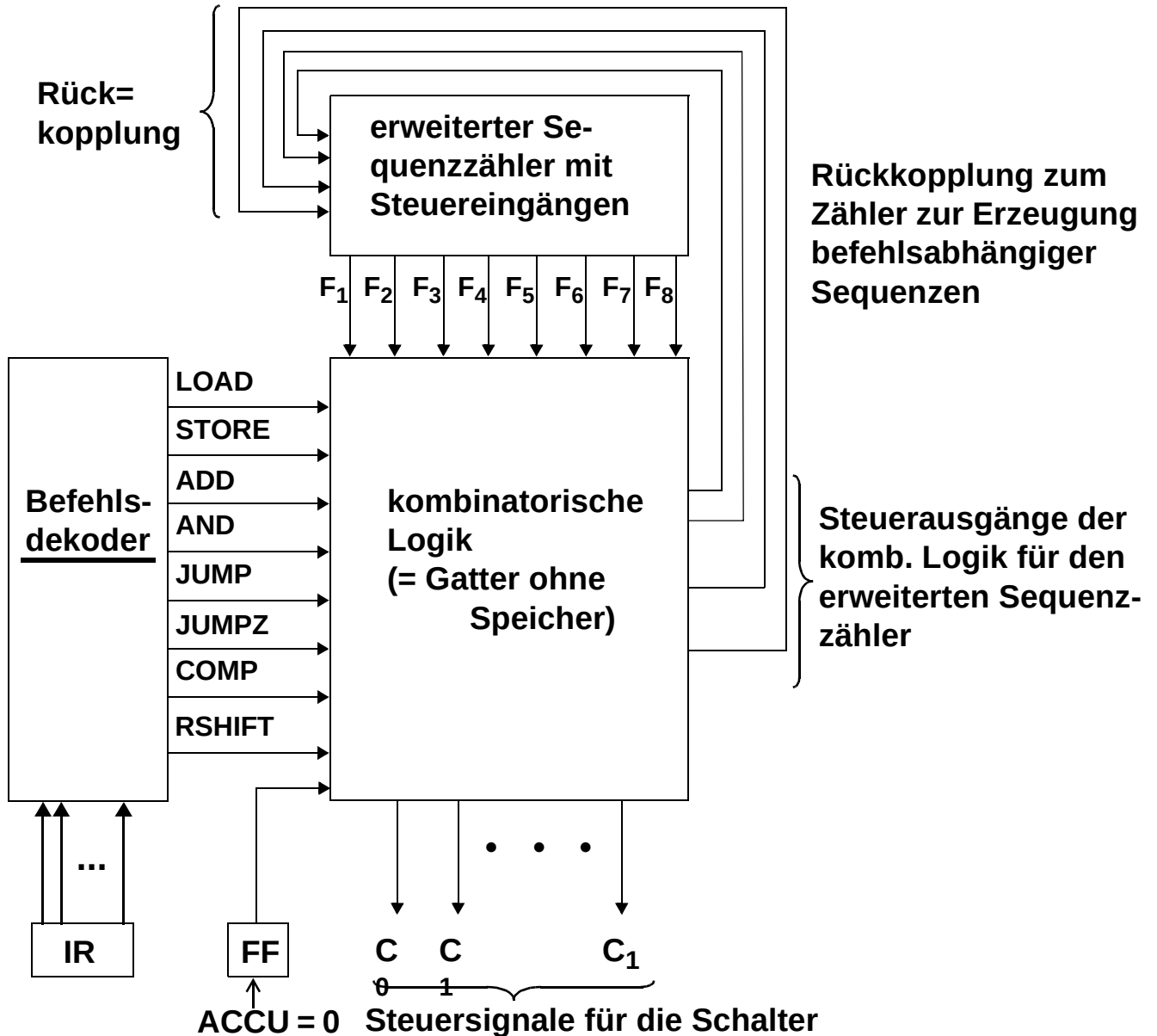
- Der Sequenzzähler wird implementiert durch einen EA mit Ein-/Ausgabe
 - Er erzeugt nacheinander verschiedene Impulse $\Phi_1, \Phi_2, \dots, \Phi_k$,
 - Jeder Impuls erscheint auf einer anderen Ausgangsleitung des 1 aus k-Dekoders
 - Alle Impulse sind gleich lang und haben den gleichen zeitlichen Abstand zueinander
 - Enable-Eingang am Dekoder erlaubt Pulse in derselben Pulsbreite wie beim Taktsignal
- => Sie definieren ein Raster aus Zeitpunkten (=Signalimpulse)
- Die Signalimpulse steuern die Schalter im Rechenwerk nacheinander an => Sequenz von Impulsen

Ein richtiges Leitwerk: Sequenzzähler mit-nachgeschalteter Logik



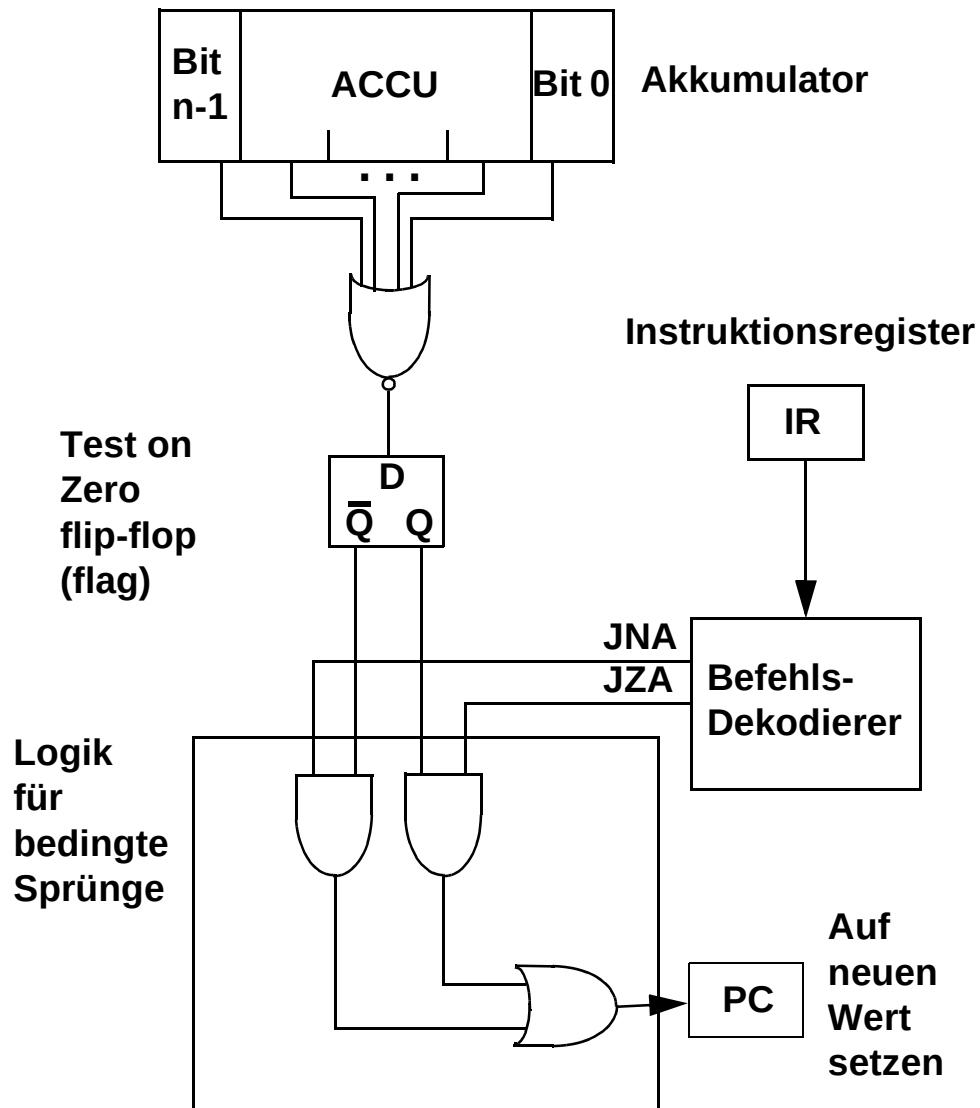
- Die nachgeschaltete Logik (= Steuerlogik) ist eine Erweiterung des Sequenzzählers. Sie erlaubt, Impulse unterschiedlicher Länge in unterschiedlichem Abstand zueinander zu erzeugen. Das ist das, was ein reales Rechenwerk braucht.
- Zusätzlich verarbeitet die Steuerlogik Statusbits aus der ALU und die Bits des Befehlswordes
- Alles zusammen wird implementiert als EA mit Ein-/Ausgabe (Moore Automat)
- Nachteil: feste Zählreihenfolge (keine Permutationen)

Ein richtiges Leitwerk (2): Sequenzzähler mit Logik sowie mit Rückkopplung und Befehlsdekoder



- Die Rückkopplung sorgt dafür, daß abhängig vom jeweiligen Befehl eine individuelle Sequenz von Steuersignalen erzeugt wird
- Der Befehlsdekoder aktiviert dazu pro Befehl einer seiner Ausgangsleitungen als Steuersignal für die komb. Logik
- Der Sequenzzähler hat Steuereingänge, die es erlauben in verschiedenen Reihenfolgen zu zählen => Permutationen

- Beispiel für kombinatorische Logik: „Jump-Logik“



- Implementierung der bedingten Sprünge
JZA (jump on zero accumulator)
und JNA (jump on nonzero accumulator)

**Das Leitwerk (Steuerwerk)
realisiert durch die
MIKROPROGRAMMIERUNG**

Mikroprogrammierung (1)

- Das gezeigte Leitwerk aus Sequenzzähler mit Logik, Befehlsdekoder und Rückkopplung ist nur eine Möglichkeit der Realisierung
- Bei der Generation der CISC-Rechner (1971 - ca. 1990) wurde das Leitwerk anders realisiert: als eigener kleiner Prozessor
- Dieser Prozessor heißt Mikroprogramm-Sequenzer und stellt einen Prozessor innerhalb der CPU dar
- Der Sequenzer führt ein Programm aus, das als Mikroprogramm bezeichnet wird und das in einem Festwertspeicher innerhalb der CPU gespeichert ist (sog. Mikroprogrammspeicher)
- Die einzelnen Befehle des Mikroprogramms werden von dem Mikrobefehlszähler des Sequenzers adressiert

Mikroprogrammierung (2)

- Der Trick ist, daß der Sequenzer wesentlich weniger Befehle beherrscht, als der Prozessor, in dem er sitzt. Dadurch ist er einfacher zu realisieren.
- Da das Leitwerk im Falle des Sequenzers programmgesteuert ist, bewirkt jeder einzelne Befehl, den der Prozessor ausführt, die Ausführung eines kompletten Mikroprogramms durch den Sequenzer
- Macht man den Mikroprogrammspeicher sehr groß (einige KB), kann der Sequenzer komplexe Folgen von Steuersignalen erzeugen
- Daraus resultieren komplexe Befehle des Prozessors. Sie geben dieser Prozessorgeneration den Namen:
CISC
= Complex Instruction Set Computer

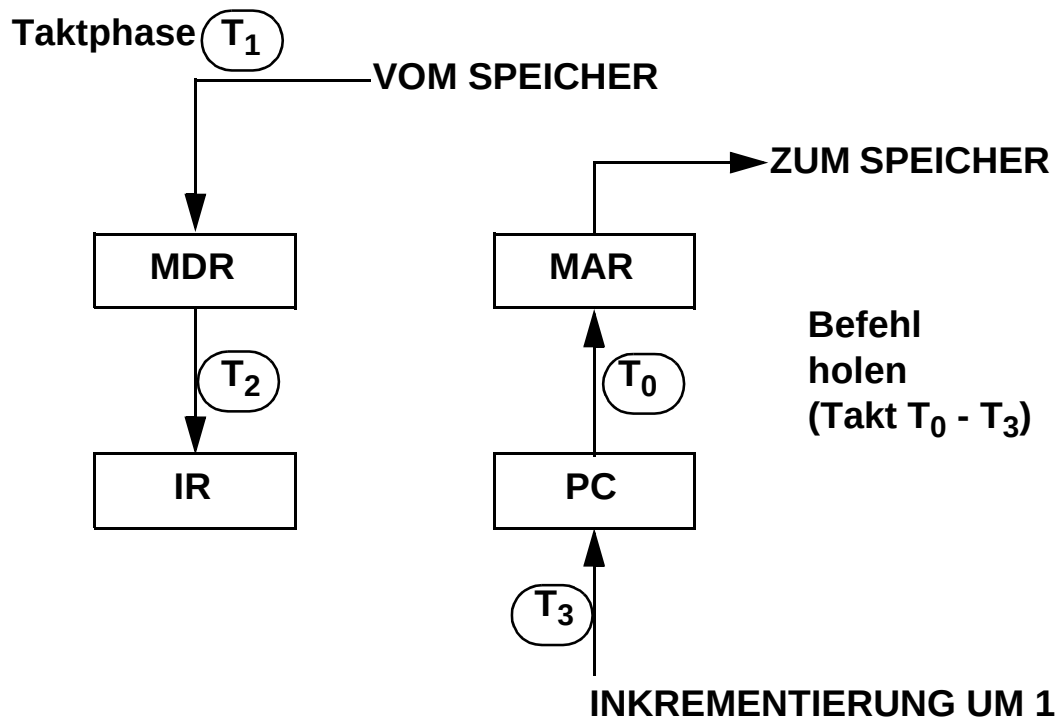
Mikroprogrammierung (3)

- Das Prinzip des Prozessors im Prozessor kann rekursiv fortgesetzt werden. Dann wird das Leitwerk des Sequenzers nicht hardwaremäßig ausgeführt, sondern als Sequenzer.
 - Man spricht dann von Nanoprogramm-Sequenzer bzw. Nanoprogramm
Bsp.: Die Prozessoren der Motorola 68 000er- Serie sind mikro- und nanoprogrammiert.
 - Nachteil der CISC-Prozessoren ist, daß sie langsam sind, weil für jeden Befehl mehrere Mikrobefehle ausgeführt werden müssen und für diese u. U. weitere Nanobefehle
 - Vorteil ist, daß komplexe Befehle sich mit wenig Hardware-Aufwand in Software implementieren lassen
- => Prozessor kann hinsichtlich seiner Befehle relativ schnell an neue Erfordernisse angepasst werden (neuer Befehlssatz). Eine Änderung der Hardware ist nicht erforderlich.

Interaktion zwischen den funktionellen Einheiten eines Prozessors

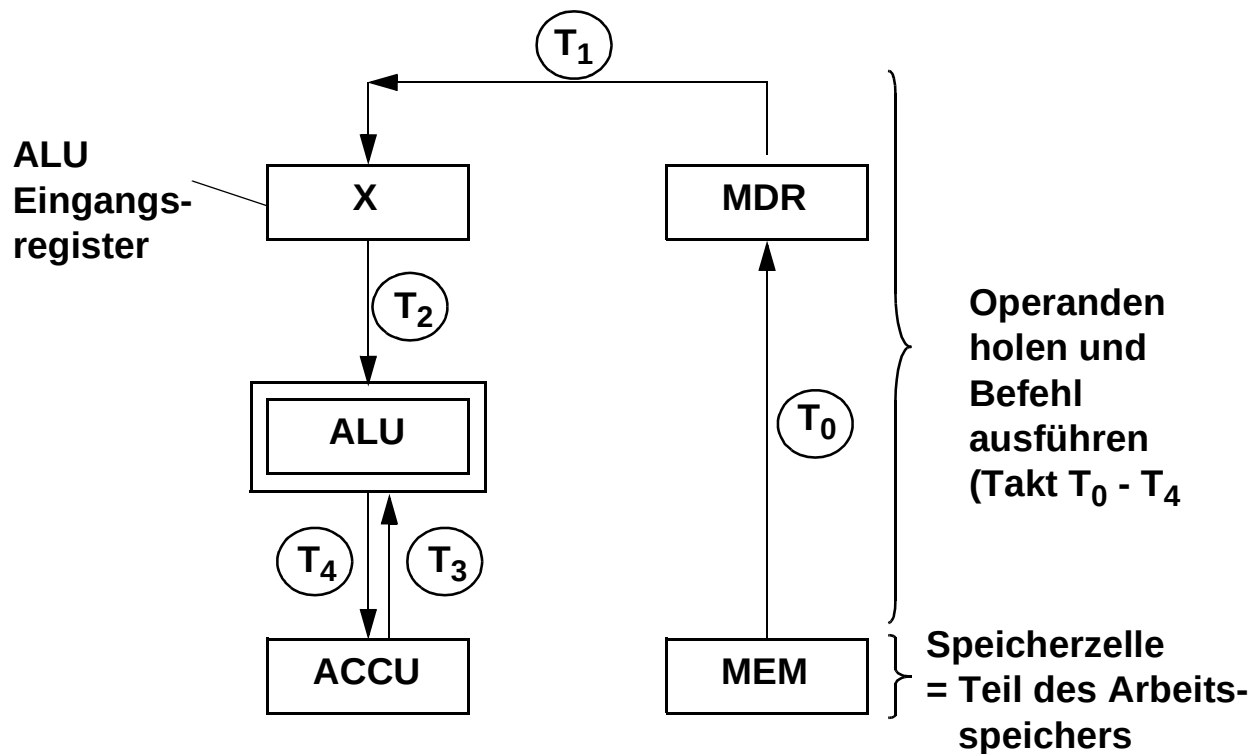
„Wie funktioniert ein Prozessor?“

Die Befehlsholphase der CPU (Zusammenspiel PC, MAR, MDR und IR)



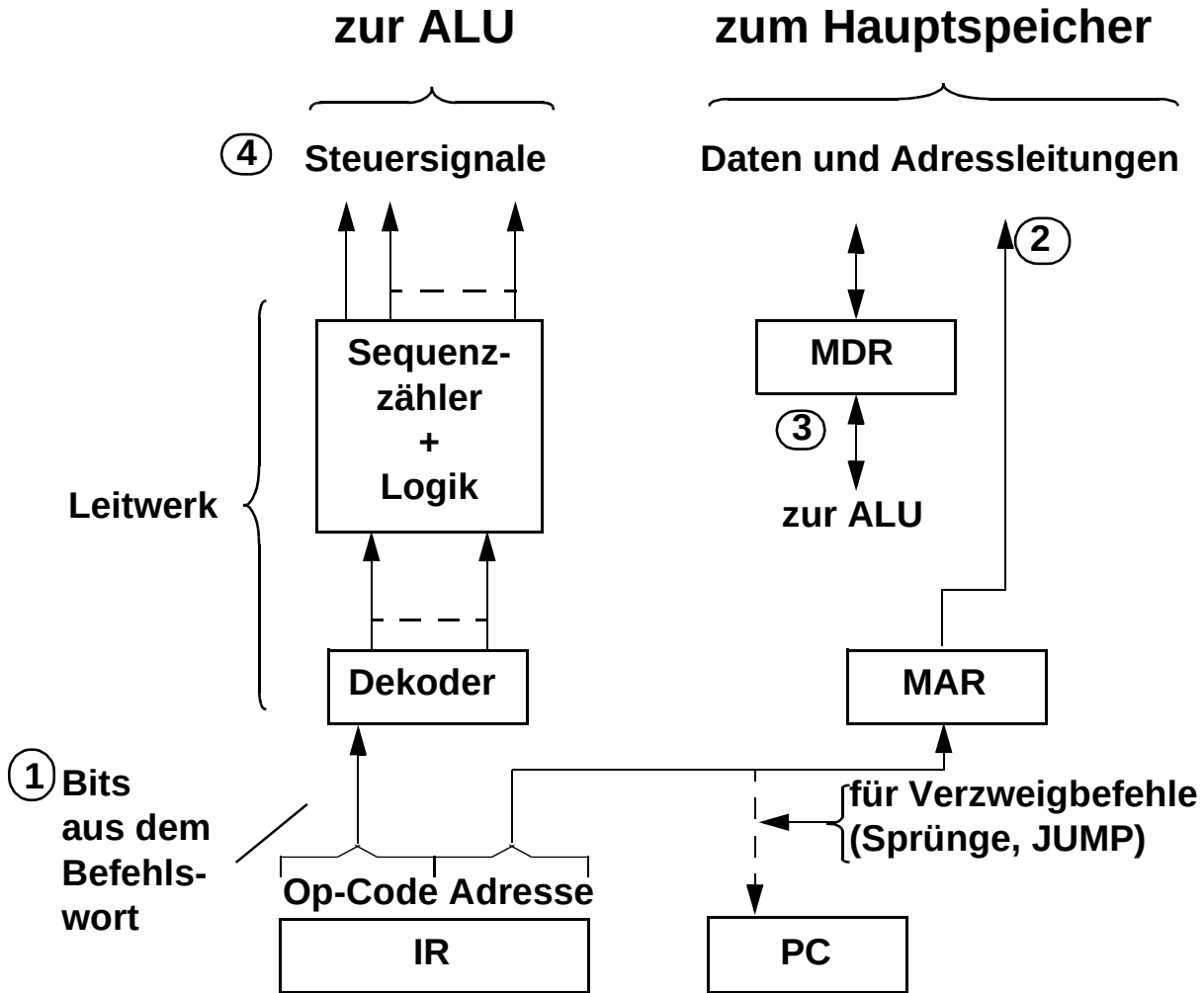
- Befehlsholphase hat 4 Unterphasen
- Jede Unterphase muss einzeln in Gang gesetzt werden
- Das Leitwerk erzeugt die Steuerimpulse T₀ - T₃
- Es gibt damit den Ablauf innerhalb der Befehlsholphase an
- Dieser Ablauf wird auch Sequenz von Aktionen bezeichnet

Die Phasen „Operand holen“ und „Befehl ausführen“ (Zusammenspiel ALU, ACCU, MDR)



- Operand holen und Befehl ausführen hat in diesem Beispiel 5 Unterphasen
- wiederum steuert das Leitwerk die Sequenz der Aktionen (T₀ - T₄), die für die Phasen „Operand holen“ und „Befehl ausführen“ nötig sind

Zusammenspiel von Sequenzzähler mit Logik, PC, IR, MAR und MDR



- Der Inhalt von IR wird in Befehlscode (op-Code) und in Operandadresse(n) aufgespalten

PC = program counter (Befehlszähler)

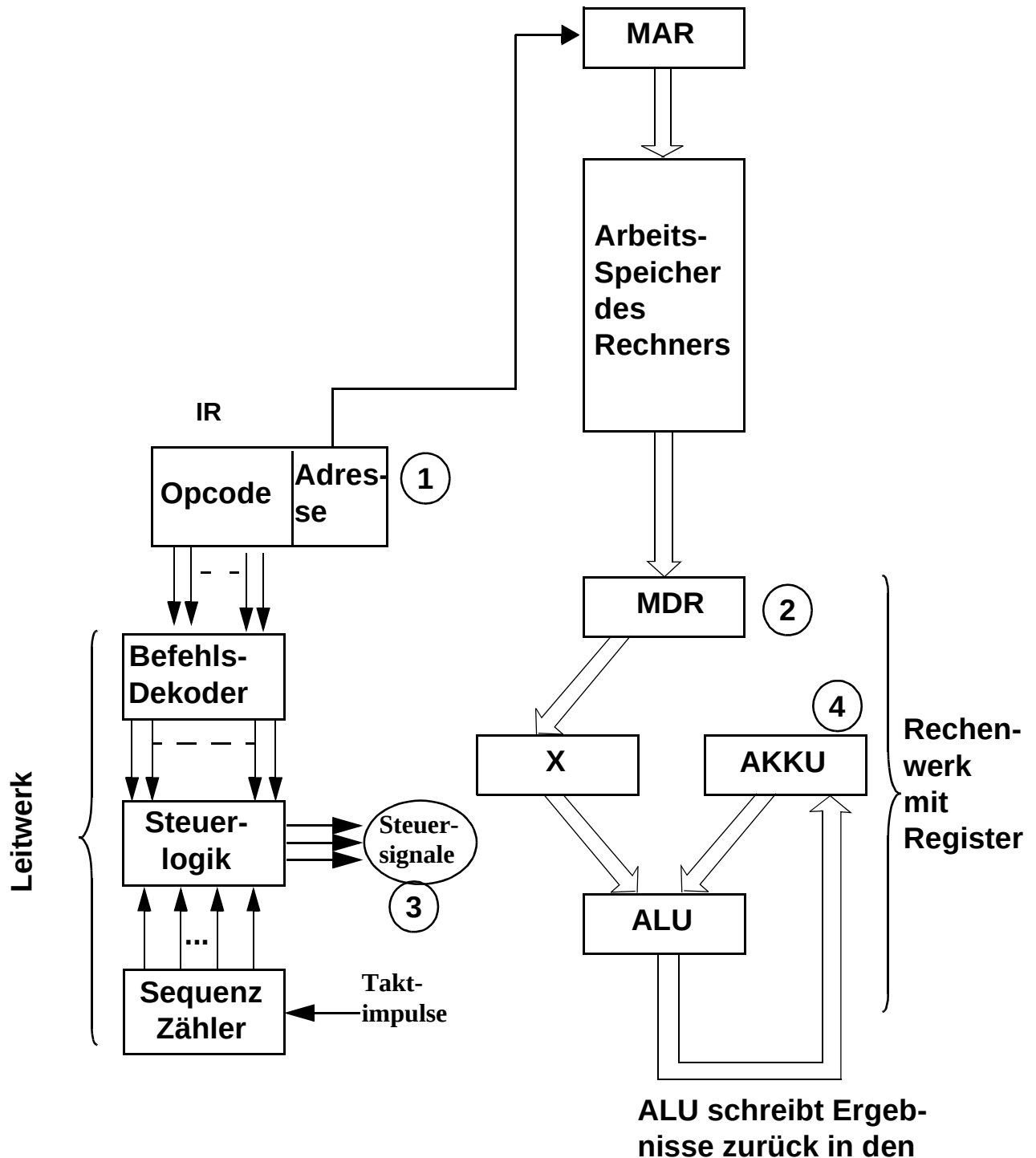
IR = instruction register (Instruktionsregister)

MAR = memory address register

MDR = memory data register

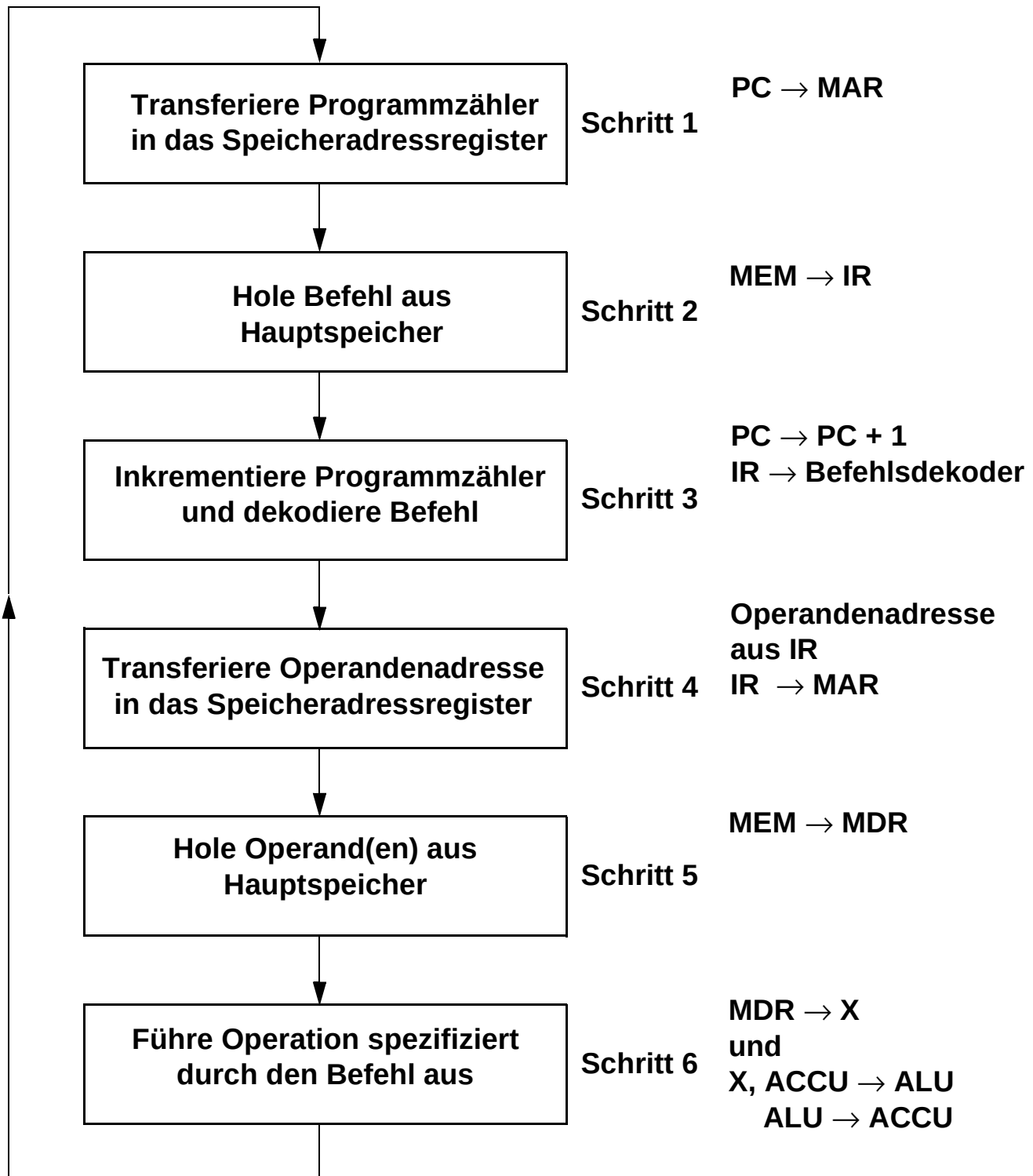
Op-Code = Operation Code = Befehlscode

Zusammenspiel von Leitwerk, Rechenwerk und Register

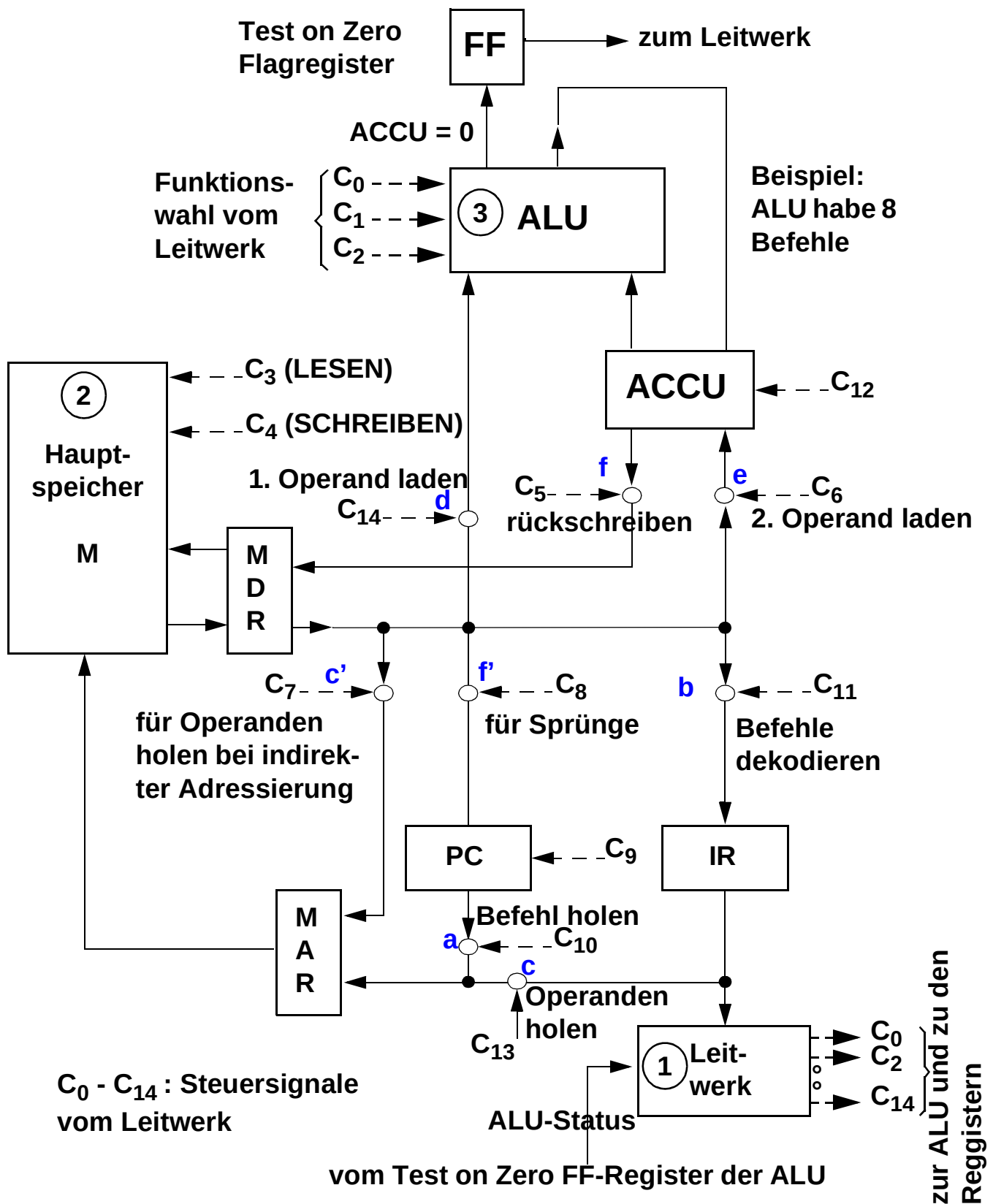


- IR = Instruktionsregister
- MAR = Speicheradressenregister
- MDR = Speicherdatenregister
- X = ALU - Eingangsregister (Hilfsregister)
- AKKU = ALU - Eingangs- und Ausgangsregister

Die Hauptschleife in der CPU



Zusammenspiel zw. Rechenwerk, Leitwerk, Speicher, Register und Schalter



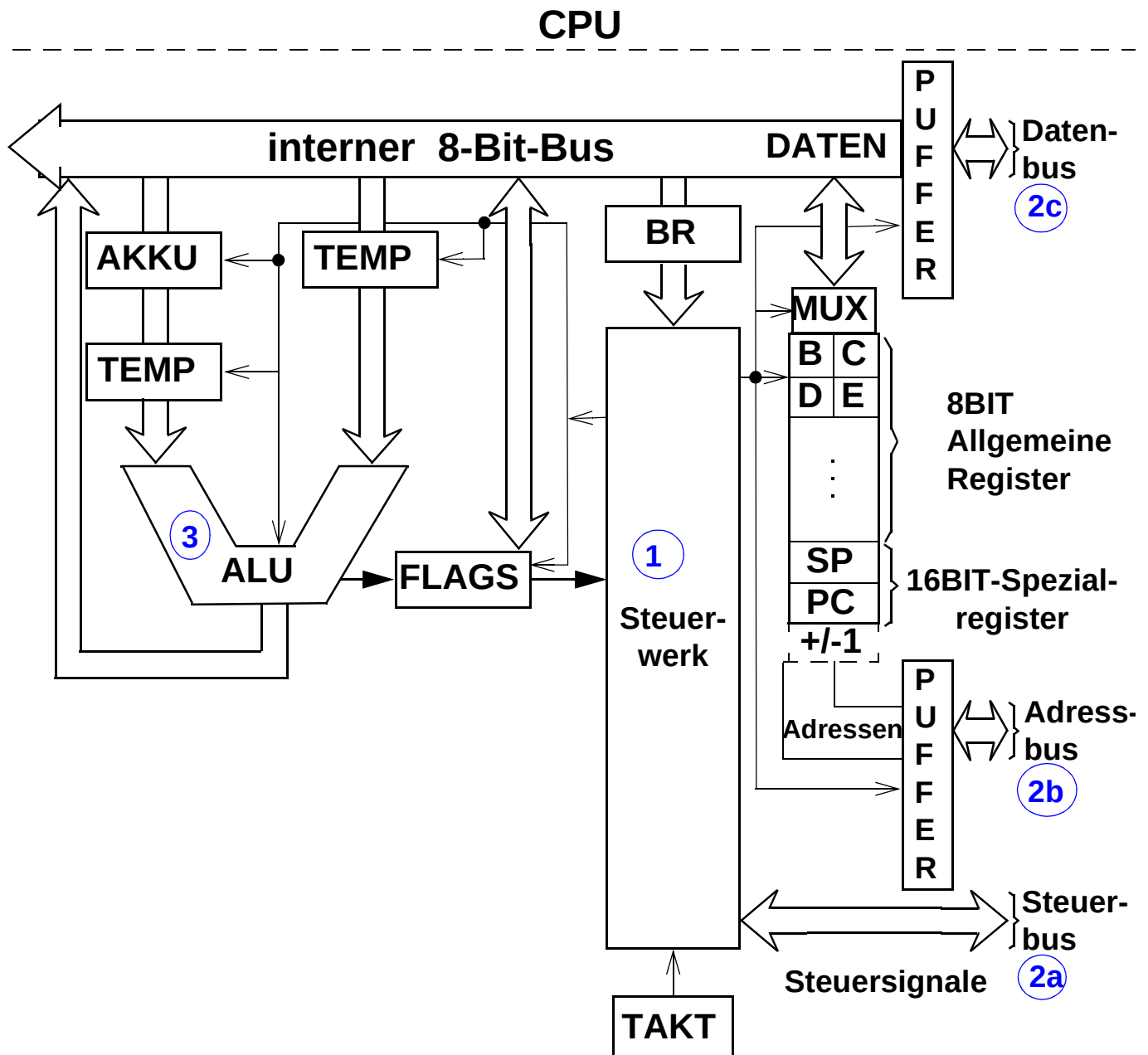
CISC-Prozessoren

CISC = Complex Instruction Set Computer

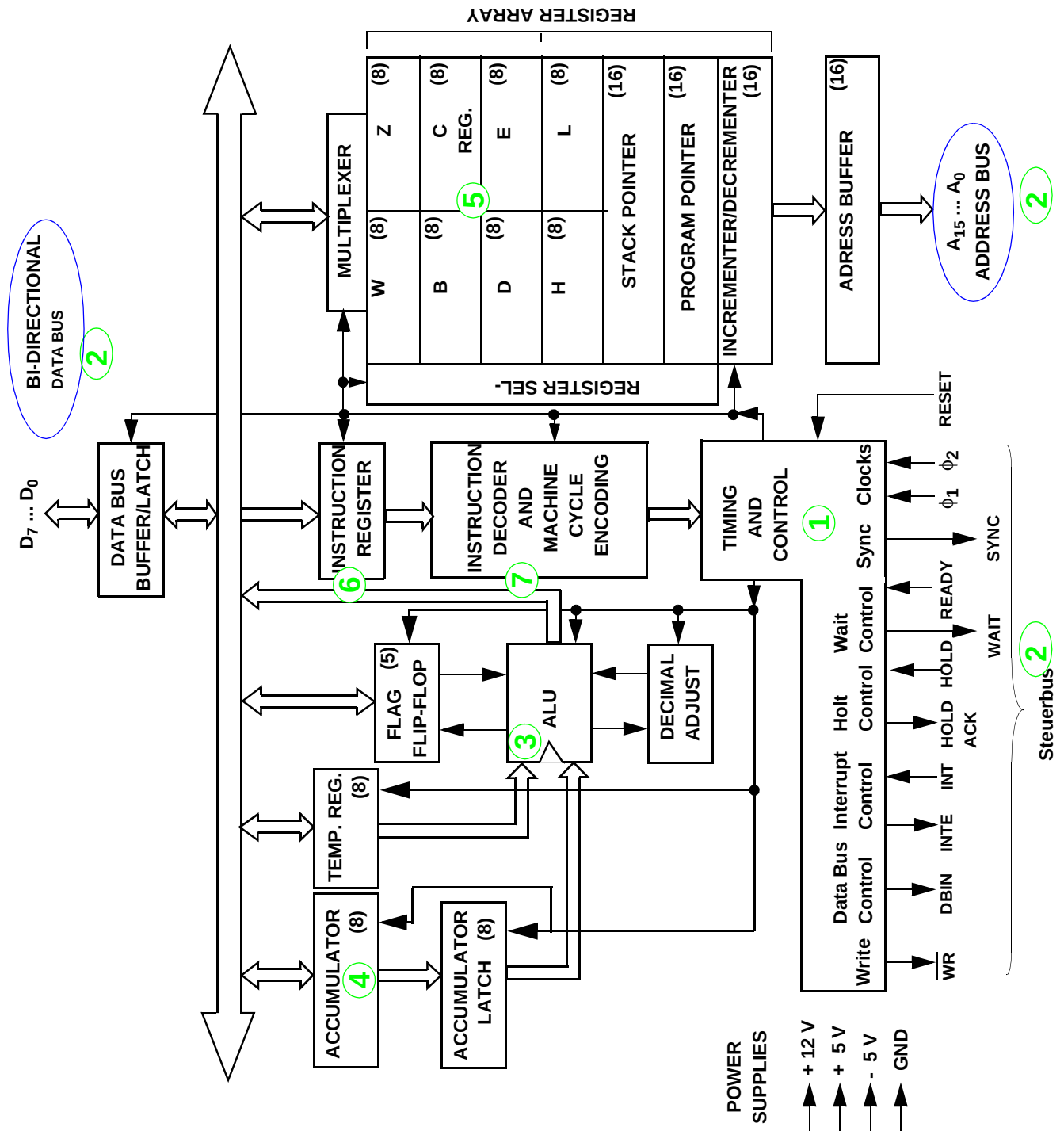
**= Bezeichnung für Prozessoren,
wie sie von 1971 - ca. 1990 gebaut wurden**

Beispiel eines 8-Bit-CISC-Mikroprozessors

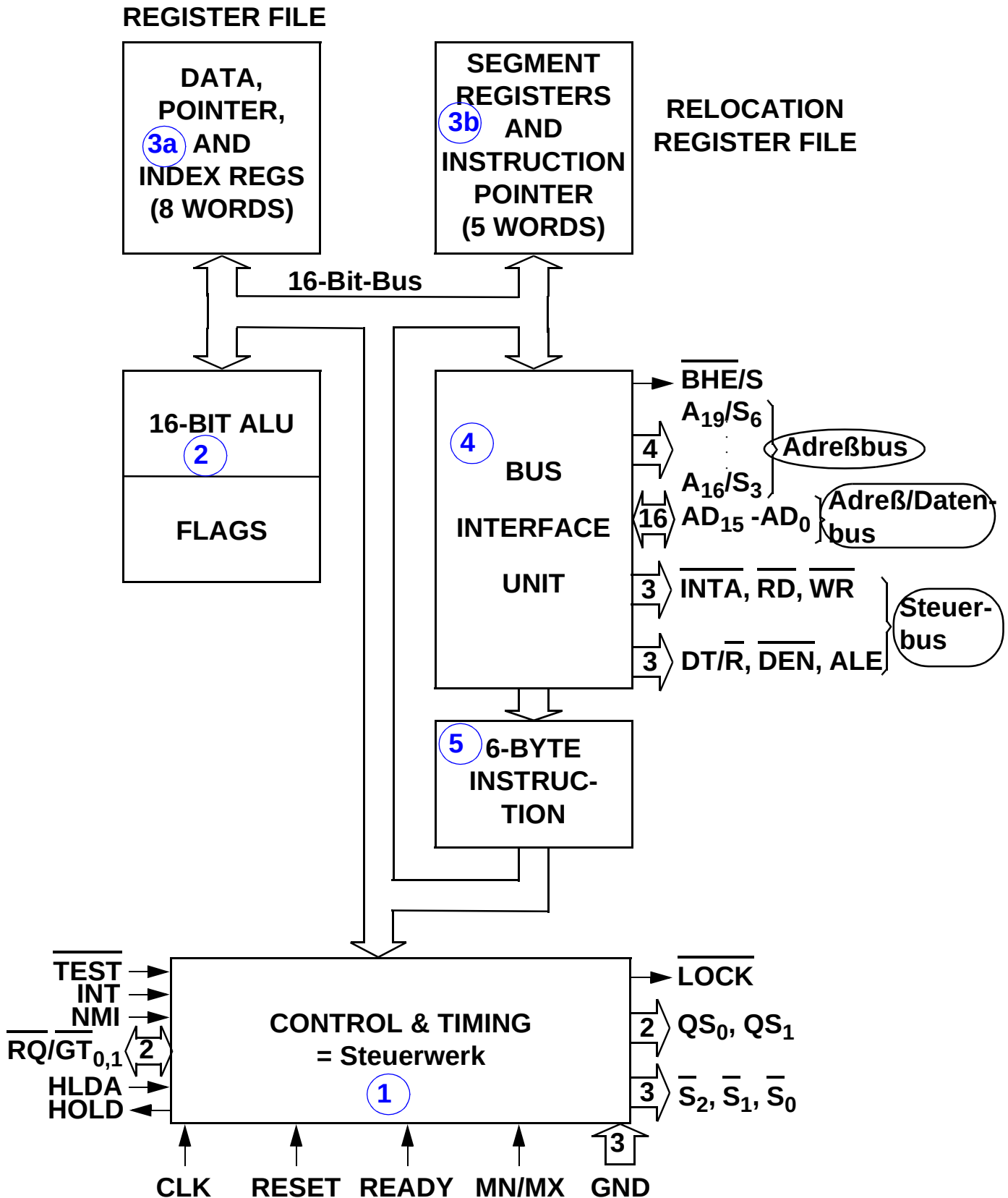
- **8-Bit-Architektur** heißt, daß interne Datenbusse und Register 8 Bit breit sind



Der erste 8-Bit Mikroprozessor „Intel 8080“



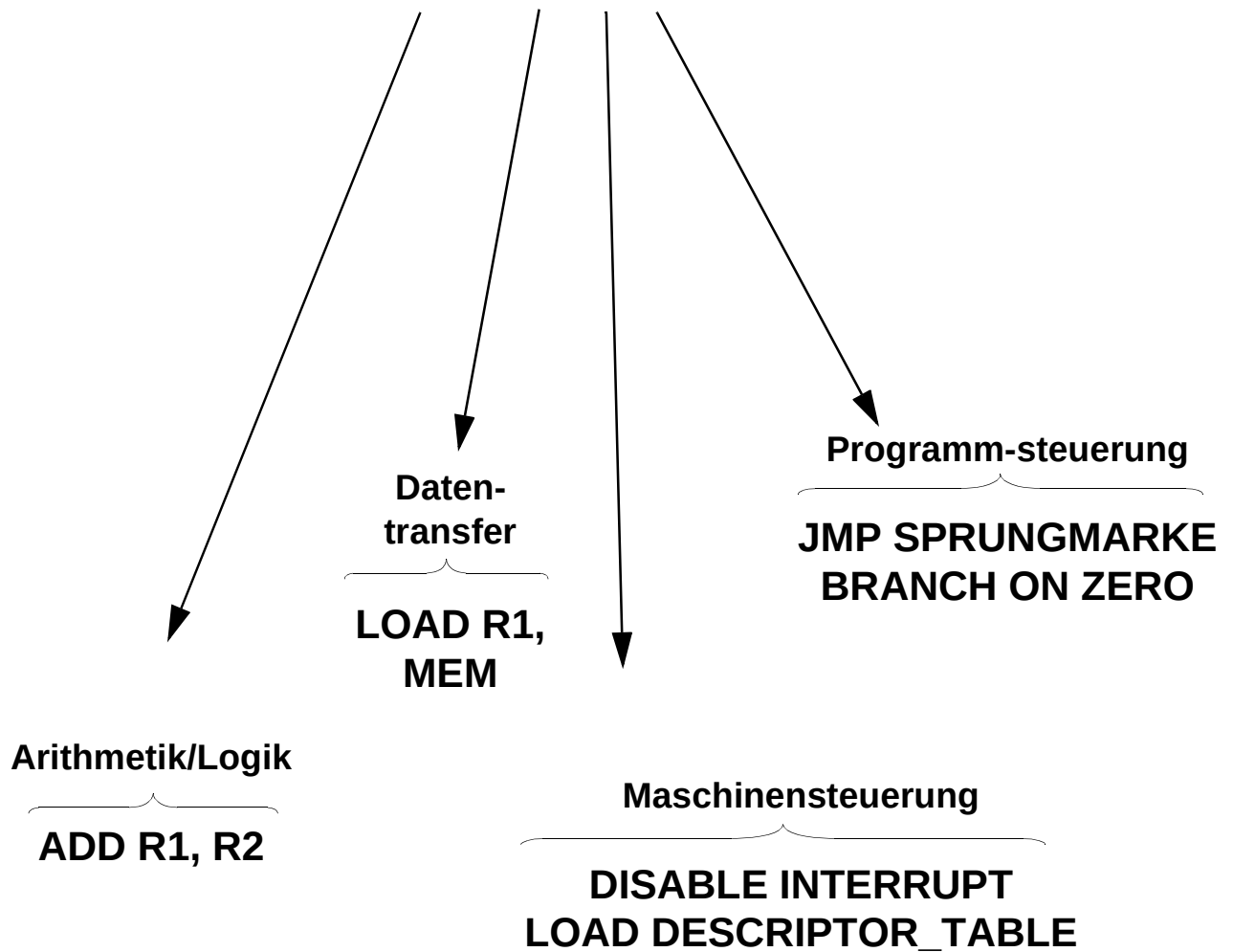
Der erste 16-Bit-Mikroprozessor Intel 8086



Der Befehlssatz eines Prozessors

- ♦ **Grundlegende Befehlsklassen**
- ♦ **Exemplarischer Befehlssatz**
- ♦ **Befehle des Pentium II**
- ♦ **Adressierungsarten**

Grundlegende Befehlsklassen



Exemplarischer Befehlssatz eines CISC-Prozessors

Typ	Operationsname(n)	Beschreibung
DATEN TRANSFER	MOVE (TRANSFER)	Transferiert Wort oder Block von Quelle nach Ziel
	STORE	Transferiert Wort vom <u>Prozes-</u> <u>sor</u> zum externen Speicher
	LOAD (FETCH)	Transferiert Wort vom exter- nen Speicher <u>zum</u> Prozessor
	EXCHANGE	Vertauscht Inhalt von Quelle und Ziel
	CLEAR (RESET)	Transferiert Null-Wort zum Ziel
	SET	Transferiert Wort aus 1en zum Ziel
	PUSH	Transferiert Wort von der Quelle zur Spitze des Stapel- speichers
	POP (PULL)	Transferiert Wort von der Spit- ze des Stapelspeichers zum Ziel

ARITHMETIK	ADD	Berechnet die Summe zweier Operanden
	SUBTRACT	Berechnet die Differenz zweier Operanden
	MULTIPLY	Berechnet das Produkt zweier Operanden
	DIVIDE	Berechnet den Quotient zweier Operanden
	ABSOLUTE	Ersetzt den Operanden durch seinen Absolutwert
	NEGATE	Ändert das Vorzeichen des Operanden
	INCREMENT DECREMENT	Addiert 1 zum Operanden Subtrahiert 1 vom Operanden
LOGISCH	AND OR NOT (COMPLEMENT)	Führt die angegebenen logischen Operationen bitweise aus
	EXCLUSIVE-OR EQUIVALENCE	Test auf Bitgleichheit
	SHIFT	Links- (rechts-) verschieben des Operanden, Einführung von Konstanten am Ende
	ROTATE	Links- (rechts-) shift des Operanden entlang eines geschlossenen Weges
	CONVERT (EDIT)	Ändert Datenformat, z. B. von binär nach dezimal

Grundbefehlssatz

PROGRAMM= STEUERUNG (flow control)	JUMP (BRANCH)	Unbedingter Sprung; lädt PC mit bestimmter Adresse
	JUMP CONDITIONAL	Teste bestimmte Bedingung; abhängig von Bedingung, entweder PC mit bestimmter Adresse laden oder nichts tun
	JUMP TO SUBROUTINE = (CALL, BRANCH-AND- LINK)	Speichere aktuelle Programmkontrollinformation (PC, Status Register, usw.) an bestimmte Stelle, z. B. auf den Stapelspeicher; springe zu angegebener Adresse
	RETURN	Ersetze Inhalte des PC, Status- Register, usw., mit Information von entsprechender Stelle, z. B. von der Spitze des Stapelspeichers

	EXECUTE	Hole Operand von angegebener Stelle und führe ihn als Befehl aus. Beachte, dass der PC nicht modifiziert wird
	SKIP	Inkrementiere PC, um nächsten Befehl zu überspringen
	SKIP CONDITIONAL	Teste bestimmte Bedingung; abhängig von Bedingung, entweder PC inkrementieren oder nichts tun
	TEST	Teste bestimmte Bedingung; setze flag(s) entsprechend
	COMPARE	Mache logische oder arithmetische Vergleiche von zwei oder mehr Operanden; setze flags entsprechend
	SET CONTROL- VARIABLES	Umfangreiche Klasse von Befehlen für Sicherheitskontrollen verschiedene Zwecke, Unterbrechungsbehandlung, Uhrenkontrolle usw. (oft privilegierte Befehle)
	HALT	Stoppe Programmausführung
	WAIT (HOLD)	Stoppe Programmausführung; teste bestimmte Bedingung fortlaufend, wenn Bedingung zutrifft, Ausführung fortsetzen
	NO OPERATION	Keinen Befehl ausführen, Programm wird fortgesetzt

Eingabe	INPUT (READ)	Transferiert Daten von bestimmtem IO-Port oder Gerät an bestimmtes Ziel, z. B. Hauptspeicher oder Prozessor-Register
	OUTPUT (WRITE)	Transferiert Daten von bestimmter Quelle an IO-Port oder Gerät
Ausgabe	START IO	Transferiert Befehle an IOP um IO-Operation zu starten
	TEST IO	Transferiert Status Information von IO-System an bestimmtes Ziel
	HALT IO	Transferiert Befehle zur IOP um IO-Operation zu beenden

Auswahl von Pentium II-Instruktionen

Moves

MOV DST, SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1, DS2	Exchange DS1 and DS2
LEA DST, SRC	Load effective addr of SRC into DST
CMOV DST, SRC	Conditional move

Arithmetic

ADD DST, SRC	Add SRC to DST
SUB DST, SRC	Subtract DST from SRC
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST, SRC	Add SRC to DST, then add carry bit
SBB DST, SRC	Subtract DST & carry from SRC
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST, SRC	Boolean AND SRC into DST
OR DST, SRC	Boolean OR SRC into DST
XOR DST, SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST, #	Shift DST left/right # bits
SHL/SHR DST, #	Logical shift DST left/right # bits
ROL/ROR DST, #	Rotate DST left/right # bits
RCL/RCR DST, #	Rotate DST through carry # bits

Test/compare

TST SRC1, SRC2	Boolean AND operands, set flags
CMP SRC1, SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT ADDR	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE, LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL, PORT	Input a byte from PORT to AL
OUT PORT, AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination

= shift/rotate count
LV = # locals

Adressierungsarten

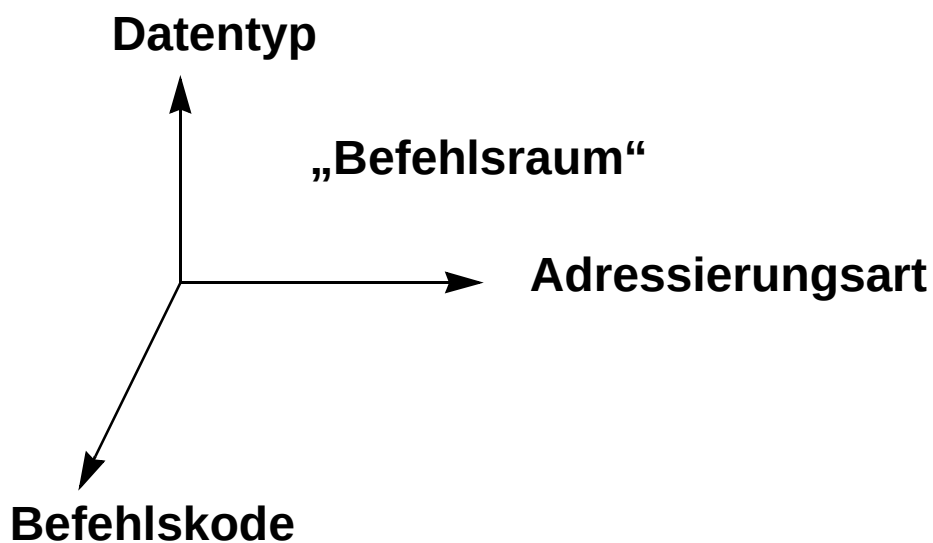
- Adressierungsarten sollen für den Compiler und für Assembler-Programmierer den Zugriff auf Daten und strukturierte Datentypen vereinfachen
- Beispiele eines strukturierten Datentyps: 2D-Array

Vor- kom- men	Art	Anwendung	Beispiel	Wirkung
RISC + CISC	Immediate	(einmalige) Konstante	ADD R1,#4	$R1 \leftarrow R1 + 4$
RISC + CISC	Direkt (<u>Absolut</u>)	statische Da- ten	ADD R1,(100)	$R1 \leftarrow R1 + \langle 100 \rangle$
RISC + CISC	Register	Wert ist im Register	ADD R1,R2	$R1 \leftarrow R1 + R2$
RISC + CISC	Register- <u>Indirekt</u>	Registerwert ist Pointer	ADD R1,(R2)	$R1 \leftarrow R1 + \langle R2 \rangle$
RISC + CISC	Displace- ment (<u>Bas- ed</u>)	lokale Var. stat. Felde- lem.	ADD R1,10(R2)	$R1 \leftarrow R1 + \langle R2 + 10 \rangle$
CISC	Basis- <u>Indiziert</u>	dyn. be- stimmte Feldelemen- te	ADD R1,(R2)(R3)	$R1 \leftarrow R1 + \langle R2 + R3 \rangle$
CISC	Auto-Inkr./ <u>Dekrement</u>	Feldzugriff in Loops	ADD R1,(R2+/-)	$R1 \leftarrow R1 + \langle R2 \rangle$ $R2 \leftarrow R2 \pm \text{Element-}$ länge
CISC	Speicher- <u>Indirekt</u>	Speicher- platz ist Pointer	ADD R1,@(R2)	$R1 \leftarrow R1 + \langle \langle R2 \rangle \rangle$
CISC	Skaliert	Feldzugriff mit Datentyp der Länge d	ADD R1,100(R2)[R3]	$R1 \leftarrow$ $R1 + \langle 100 + R2 + d \cdot R3 \rangle$

<z>: Inhalt des Speicherplatzes, der durch z gegeben ist.

Orthogonaler Befehlssatz

- Ein Befehlssatz heißt orthogonal, wenn Befehlskode, Adressierungsarten und Datentypen (und Anzahl der Operanden) unabhängig miteinander kombiniert werden können.



- Orthogonalität soll die Befehlssatznutzung vereinfachen
- Orthogonalität führt schnell zu "riesigen" Befehlssätzen, wenn es viele Adressierungsarten und Datentypen gibt. (>1000 Befehle)
- Bsp.: VAX von der Fa. DEC (sog. Supermini-Computer, 1977) besaß ein hohes Maß an Orthogonalität

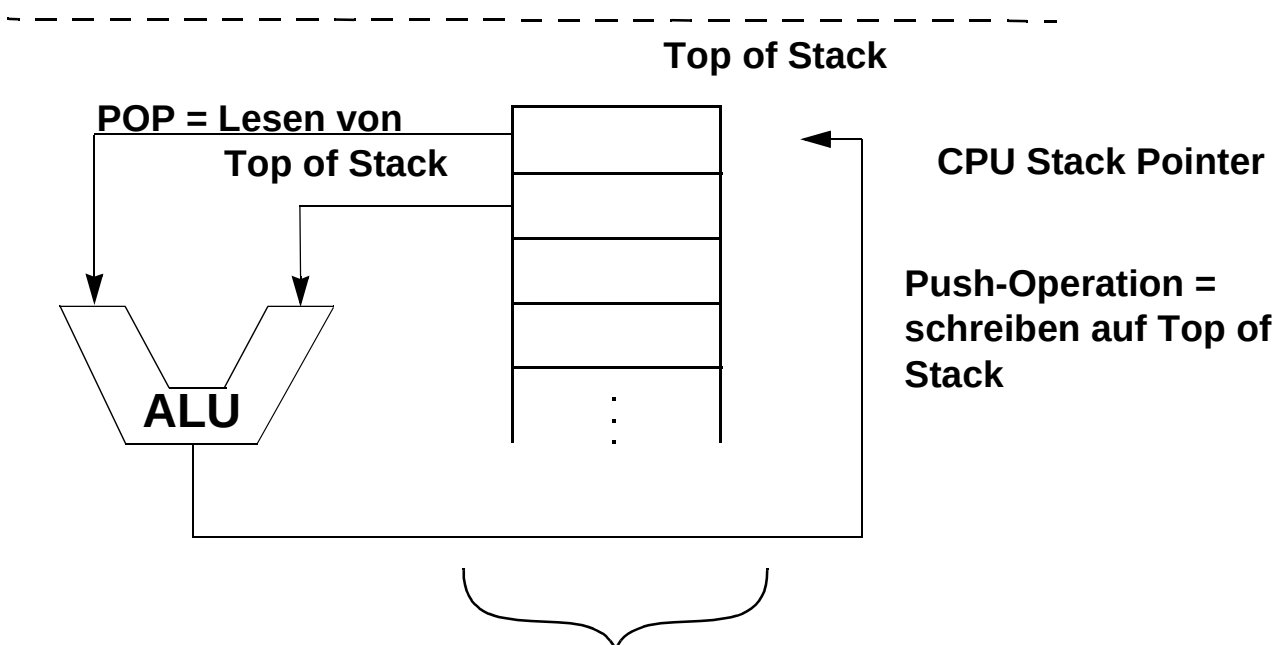
Grundlegende Architekturen bei CISC-Prozessoren

- Stack-Architektur**
- Akkumulator-Architektur**
- Register-Architektur**
- Speicher-Architektur**

Stack-Architektur

- Verknüpfungsoperationen benötigen keine Adressangaben (Nulladreßmaschine) für ihre Operanden.
- Quell- und Ergebnisoperanden befinden sich auf fest vereinbarten - impliziten - Plätzen an der Spitze von einem (Register-) Operanden-Stack.

normaler Arbeitsspeicher,
falls Stack CPU extern ist



Der Operandenstack ist ein stack-artig organisiertes Registerfile, z. B. aus 3 Registern, oder ein im Speicher organisierter Operandenstack. D. h. der Stack ist entweder CPU-intern oder extern im Speicher.

- Registerfile: mehrere Register am Stück z. B. $R_0 - R_{15}$

Stackarchitektur-Ablaufbeispiel

Befehl

**Speicheradres-
sen**

Inhalte

**$z := x + y$
mit $x = 4$**

x-Adresse 100

**2
4**

**Arbeits-
speicher**

Operanden-

**Arbeits-
speicher**

PUSH <100>, PUSH <101>,

POP <102>

100

.
2
4
.

101

2
.
.

4
2
.

6
.
.

2
4
6

102

**Register-/Speicher
Operanden-Stack**

Bewertung der Stackarchitektur

- Stackarchitekturen unterstützen die Ausführung gewisser Anweisungen in höheren Programmiersprachen, z. B.:
 - die Abarbeitung arithmetischer Ausdrücke (umgekehrte polnische Notation)
 - die Kontrolle des Gültigkeitsbereiches von Variablennamen in Verbindung mit einer Blockstruktur der Sprache = „Scope“
 - den Prozedurmechanismus (Unterprogramme) und
 - die Parameterübergabe. (Parameter kommen auf den Stack)
- Stackarchitekturen können speicherplatzsparenden Kode ergeben, da bei Stackoperationen keine Adressangabe nötig ist (Nulladreibmaschine).
- Flexibilität und Eignung für allgemeine Datenverarbeitungsaufgaben werden kritisch beurteilt:
 - kein wahlfreier Zugriff auf beliebige Stackelemente
 - Stack(top) stellt Flaschenhals dar (Pipelining)
- Stackarchitekturen stellen keine Alternative zu Rechnern mit Registerstruktur (Universalregister-Maschinen) dar.
- Populäres Beispiel einer Stack-Architektur: Java Virtual Maschine (JVM)

Auszug aus dem Befehlssatz der Java Virtual Machine

Hex	Mnemonic	Meaning
0x10	BIPUSH <i>byte</i>	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO <i>offset</i>	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ <i>offset</i>	Pop word from stack and branch if it is zero
0x9B	IFLT <i>offset</i>	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ <i>offset</i>	Pop two words from stack; branch if equal
0x84	IINC <i>varnum const</i>	Add a constant to a local variable
0x15	ILOAD <i>varnum</i>	Push local variable onto stack
0xB6	INVOKEVIRTUAL <i>disp</i>	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE <i>varnum</i>	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W <i>index</i>	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack

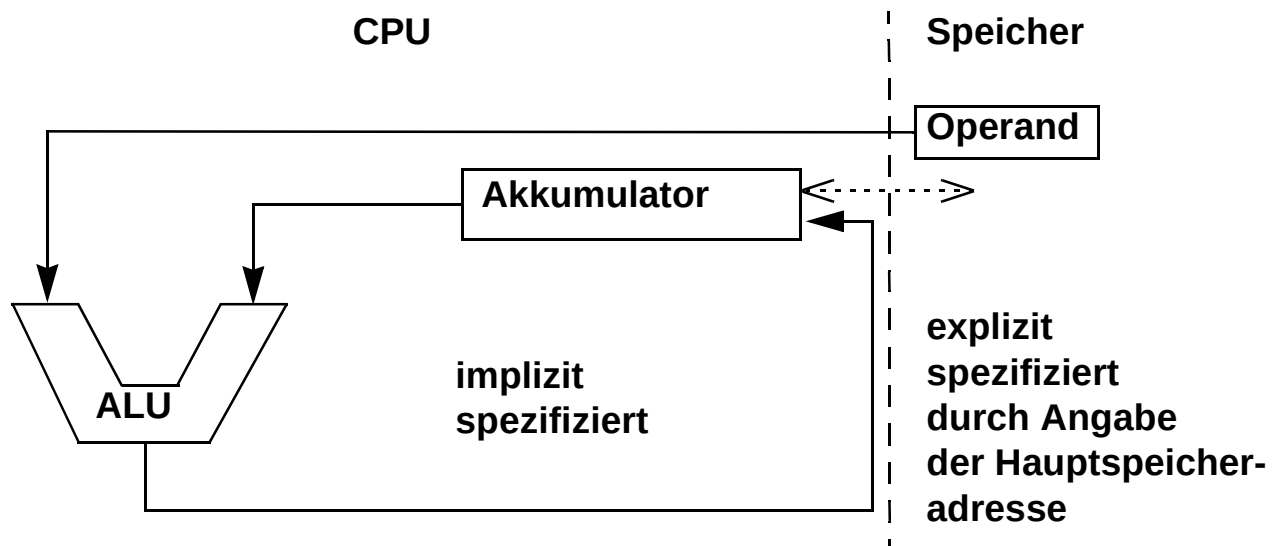
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Figure 4-11. The IJVM instruction set. The operands *byte*, *const*, and *varnum* are 1 byte. The operands *disp*, *index*, and *offset* are 2 bytes.

Akkumulator-Architekturen

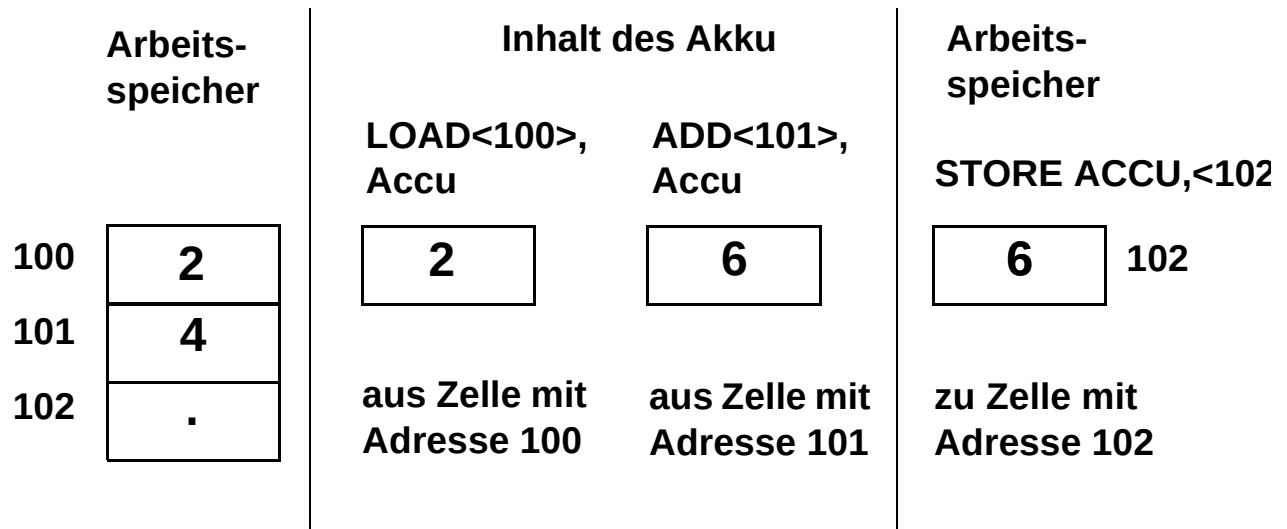
- Bei Verknüpfungsoperationen enthält das Akkumulator-Register implizit einen Quelloperanden und letztlich auch den Ergebnisoperanden.
- Außer dem Akkumulator-Register gibt es i. d. R. kein weiteres (für den Programmierer zugängliches) Register.
- Der zweite Operand liegt typisch im Hauptspeicher und

muß explizit adressiert werden. (Einadress-Maschine)



Akkumulator-Arch.-Ablaufbeispiel

Befehl	Speicheradressen	Inhalte
z := x + y	x-Adresse 100	2
	y-Adresse 101	4
	z-Adresse 102	. / 6 (nach STORE)

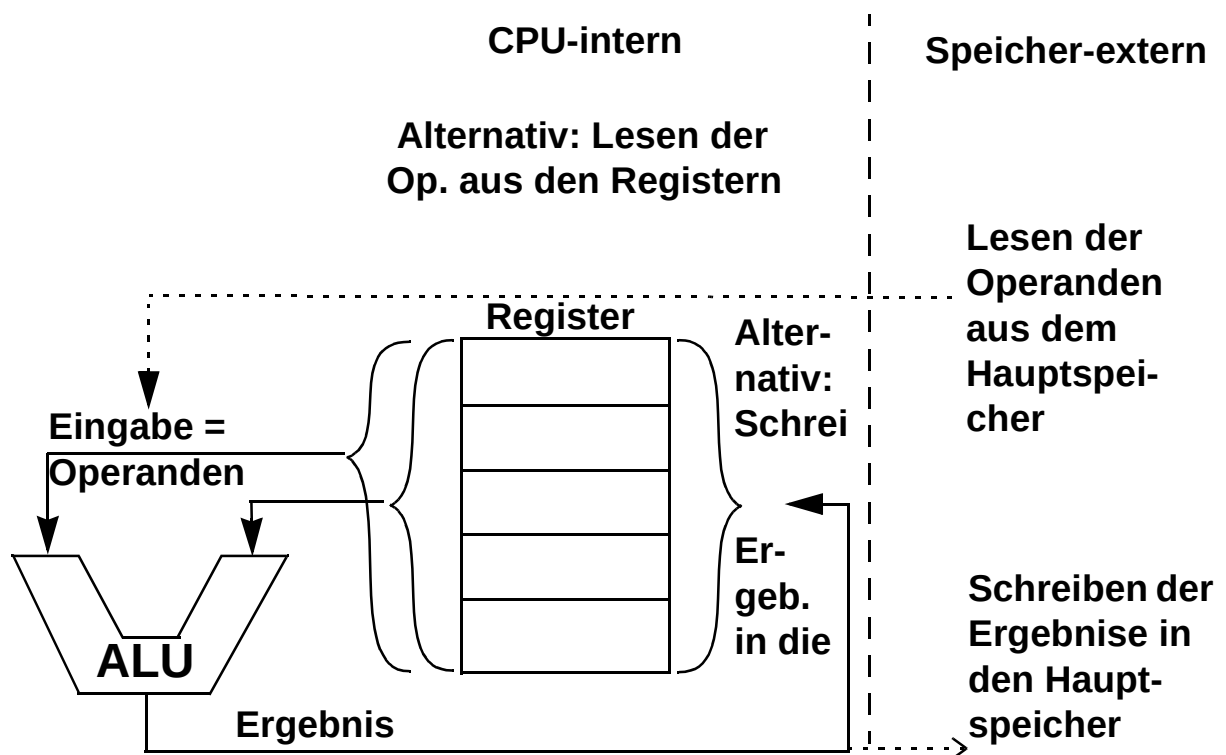


Bewertung der Akkumulator-Architektur

- Minimaler Aufwand, da nur ein internes Register, der Akkumulator, benötigt wird
- Höherer Speicherverkehr, da der Akkumulator der einzige interne temporäre Speicher ist. (vom und zum Speicher oft)
- Der interne Maschinenzustand (sog. Kontext) ist minimal
- Die ersten 8-Bit-Mikroprozessoren hatten eine Akkumulator-Architektur

(Universal-)Register-Architektur

- Ein Satz mehr oder weniger gleichberechtigter Register kann für das Halten von Variablen bzw. Operanden genutzt werden (Register-File)
- Im Op-Code sind typisch mehrere Operanden explizit anzugeben (Zweiadressmaschine, 3-Adressmaschine usw.)



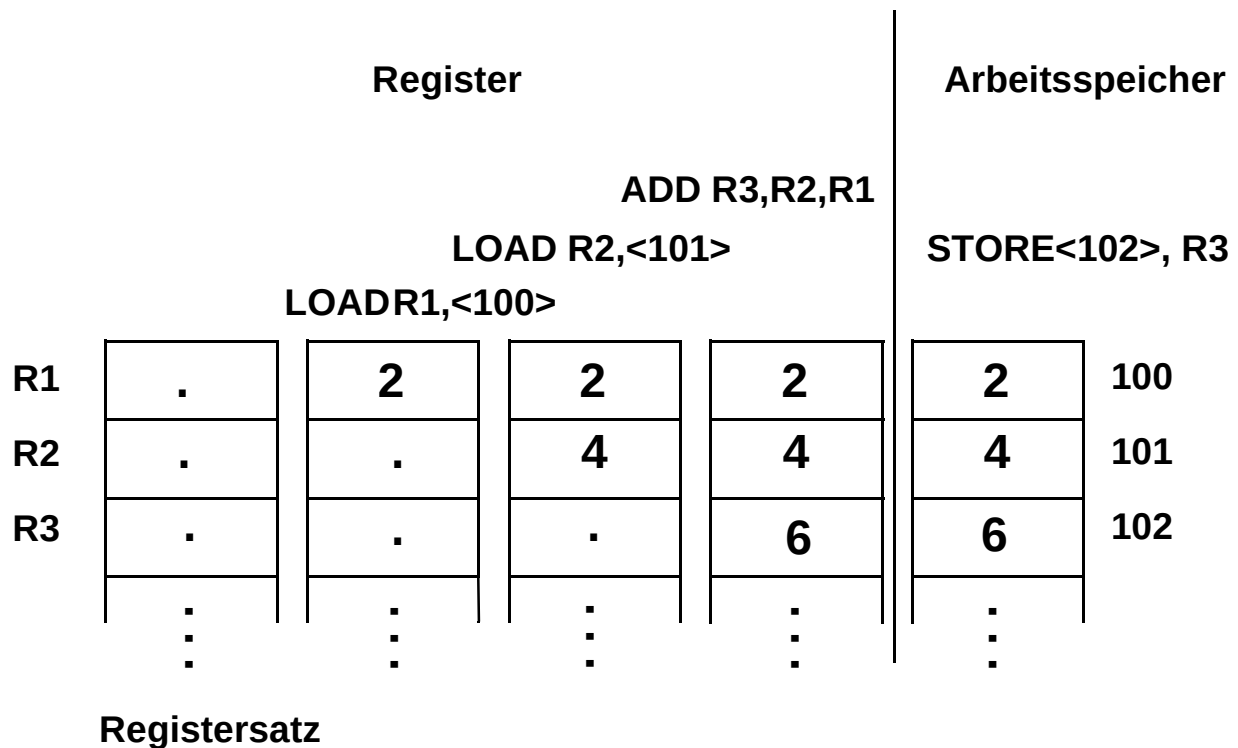
- Operanden werden durch explizite Angabe ihrer Adressen aus dem Hauptspeicher in die ALU kopiert oder aber aus den Register geholt

Ergebnis wird entweder in ein Register zurückgeschrieben oder durch Angabe einer weiteren Adresse in den Arbeitsspeicher zurücktransportiert

Registerarchitekturen - Ablaufbeispiel

- 3-Adressmaschine:

Befehl	Speicheradressen	Inhalte
	x-Adresse 100	2
$z := x + y$	y-Adresse 101	4
	z-Adresse 102	. / 6 (nach STORE)



Bewertung der Register-Architektur

Vorteile:

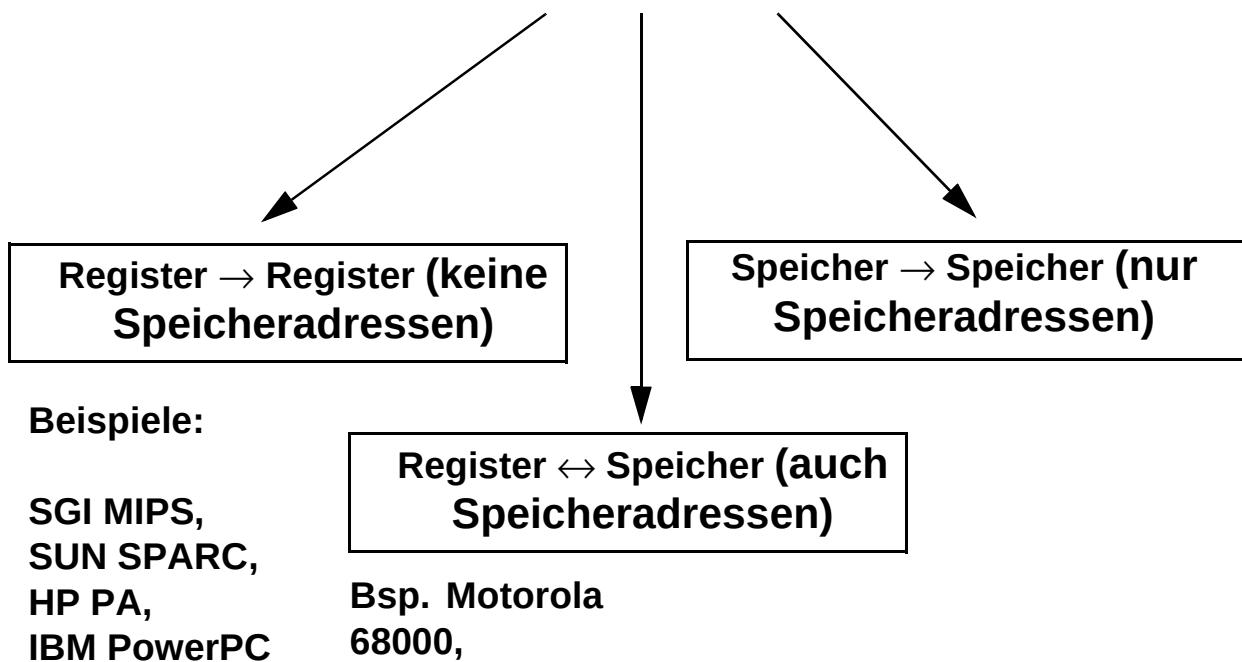
- Der Hauptvorteil liegt in der freien und effektiven Anwendbarkeit der Register durch den Compiler, vorausgesetzt, es gibt keine *impliziten Registerfestlegungen*
- Ausdrucksberechnungen können in beliebiger Reihenfolge erfolgen, was Pipelining unterstützt
- Das Halten von Variablen in Registern verringert den Speicherverkehr, beschleunigt die Ausführung, verbessert die Code-Dichte und ermöglicht den effektiven Einsatz von Superskalartechniken (Register-renaming; unabhängiger, paralleler Registerzugriff)

Nachteile:

- Bei Kontextwechsel z.B. aufgrund von Interrupt oder Time Sharing sind große Registersätze auszutauschen
- Alle Operanden müssen adressiert werden, was zu längeren Befehlen führt

Verallgemeinerte Register–Architekturen

Klassifikation nach Art der Verwendung von Speicheradressen pro ALU-Befehl



- Die Register - Architekturen sind die wichtigsten Architekturen sowohl bei CISC - als auch bei RISC - Rechnern
- RISC-Rechner unterstützen ALU-Operationen nur auf Registeradressen, nicht auf Speicheradressen
- CISC-Rechner unterstützten ALU-Operationen sowohl auf Register als auch Speicheradressen

Register–Speicher–Typ

- In Verknüpfungsbefehlen können auch Speicheroperanden adressiert werden
 - Vorteil:
 - Alle Daten können referenziert werden, ohne sie vorher in ein Register laden zu müssen
 - Nachteil:
 - Befehlslänge und die Taktzyklen pro Befehl variieren in Abhängigkeit von der Anzahl der übergebenen Speicheroperanden
- => wird nur bei CISC-Rechnern verwendet

Speicher–Speicher–Typ

Vorteile:

- Im Extremfall gibt es nur temporäre interne Register und der Programmierer “sieht” nur Speicheroperanden
- Die Programmierung wird transparenter, da nicht zwischen Register- und Speicheroperanden unterschieden werden muss
 - CISC-Intention

Nachteile:

- Hoher Speicherverkehr
- Variierende Befehlslänge und Befehlstaktzahl, wenn neben dem Nur-Speicheroperanden-Modell der Befehlssatz aus Flexibilitätsgründen auch Registeroperanden zulässt
 - (siehe: Orthogonaler Befehlssatz, CISC-Befehlssatz.)
 - => wird nur bei CISC verwendet

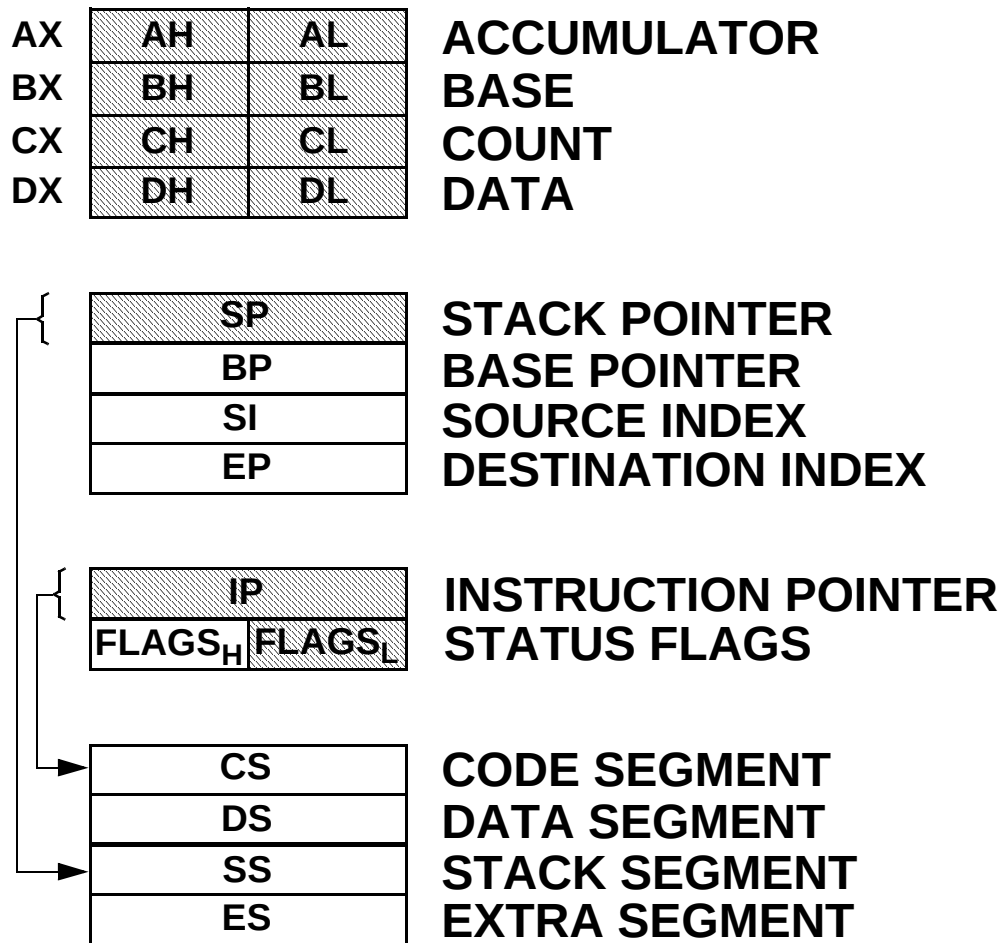
Register–Register–Typ

- Verknüpfungsoperationen der ALU verwenden ausschliesslich Register (-adressen) für Quell- und Zieloperanden
- Nur in Lade- und Speicherbefehlen sind Speicheradressangaben zu finden
- Der Speicherverkehr erfolgt nur mittels Load- und Store-Befehlen von Speicher in die Register und umgekehrt
 - Load / Store-Architektur
- Vorteil:
 - Ein Verknüpfungsbefehl tritt nicht “mal mit, mal ohne” Speicheroperand auf
 - einfache Befehlskodierungen fester Länge, daraus folgt:
 - einheitliche Taktzeiten je Befehl
 - Pipelining-Unterstützung
- Nachteil:
 - Alle Speichertransfers sind durch einen separaten Befehl auszuweisen
 - Code-Expansion (Programm wird größer)
- wird exklusiv von RISC-Rechnern eingesetzt

Beispiele für Register-Architekturen nach dem Register-Speicher-Typ

- Intel i8086**
- Intel i80386**
- Intel Pentium II**

Register des 8086



8086 Register Model (8080 Registers Shaded)

- gestrichelt sind die Register, die es auch schon beim 8080 gab.

Register des i386

Befehls-/Stapelzeiger

31 16 15
0

EIP		IP
-----	--	----

EFLAG-Register

31 16 15
0

EFLAG		FLAG
-------	--	------

Allgemeine Register

31 16 15
0

EAX		AH	AL
EBX		BH	BL
ECX		CH	CL
EDX		DH	DL
ESI		SI	
EDI		DI	

Segmentregister

15
0

CS	
SS	
DS	
ES	
FS	
GS	

Speicherverwaltungsregister

15 0 31 0
19 0

TR	TSS Wähler	TSS Basisadresse	TSS Limit
LDTR	LDT Wähler	LDT Basisadresse	LDT Limit
	IDTR	IDT Basisadresse	IDT Limit
	GDTR	GDT Basisadresse	GDT Limit

Steuerregister

31 16 15
0

CR3		
CR2		
CR1		

Debug-Register

31 16 15
0

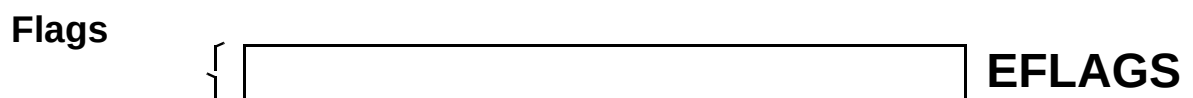
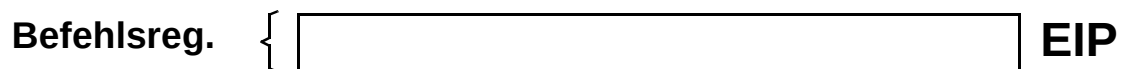
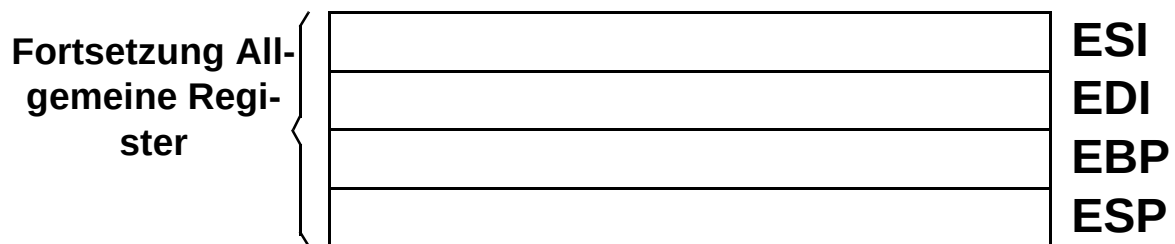
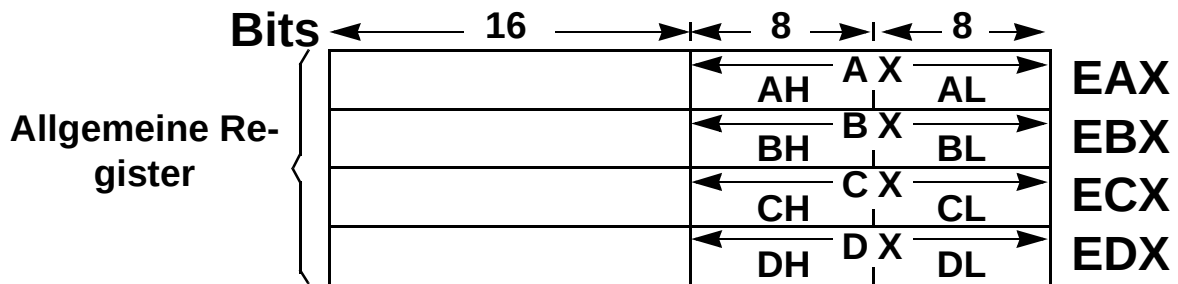
DR7		
DR6		
DR5		
DR4		
DR3		
DR2		
DR1		

Testregister

31 16 15
0

TR7		
-----	--	--

Die wichtigsten Register des Pentium II

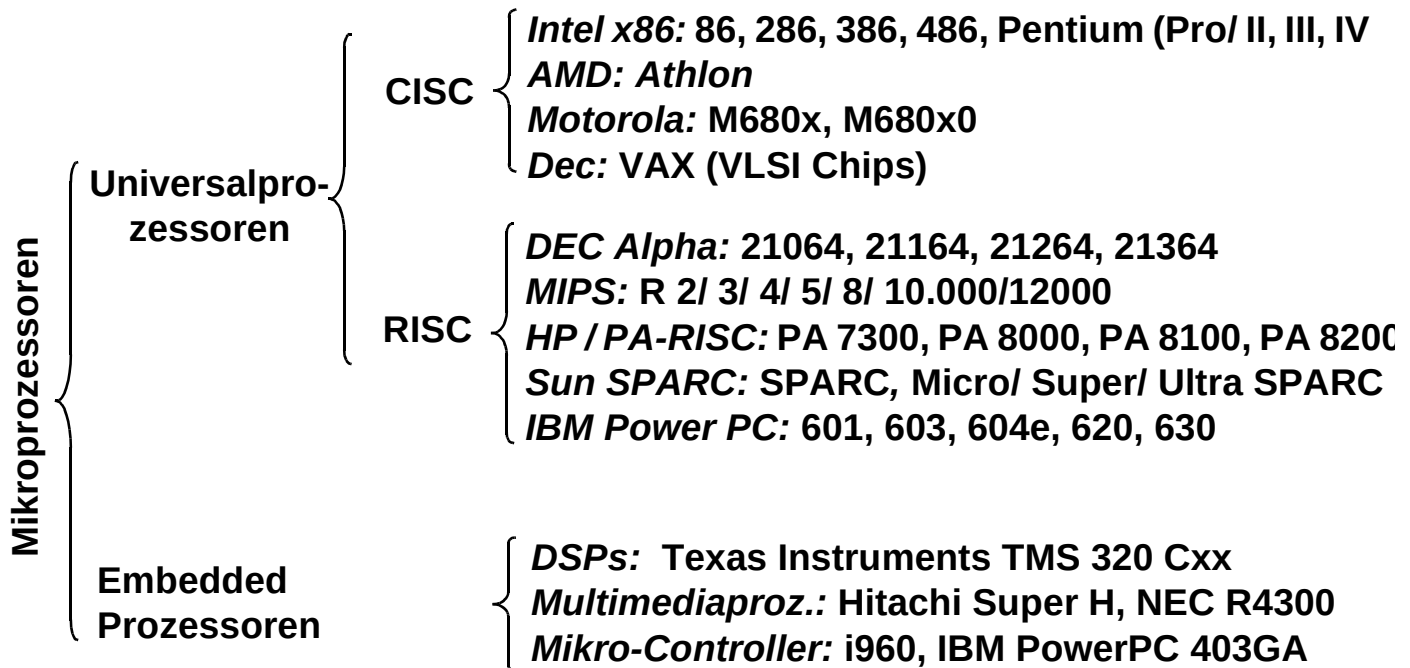


Heutiger Status der CISC-Prozessoren

- Heutzutage werden Prozessoren mit CISC-Architektur nicht mehr hergestellt
- Trotzdem gibt es sehr viele Prozessoren mit typischem CISC-Befehlssatz (alle aus der Pentium-Familie, bzw. kompatibel dazu)
- Pentium-Prozessoren weisen intern eine RISC-ähnliche Architektur auf
- Eine spez. Einheit im Pentium, der Micro opsequenzer sorgt für die Abbildung des CISC-Befehlssatzes auf die RISC-Befehle des Prozessorkerns
- Der Pentium-RISC-Kern beruht auf typischen RISC-Merkmalen wie „Superpipelining“, „Superakalarität“ und „out-of-order execution“

Prozessorfamilien

Beispiele:



DSP = digitaler Signalprozessor; DSPs sind in jedem Handy und in jedem Modem,

Multimedia-Prozessoren sind in jeder Spieleconsole,

Mikrocontroller sind in jeder Automobil-Motorsteuerung eingebaut (i.e. „eingebettet“)

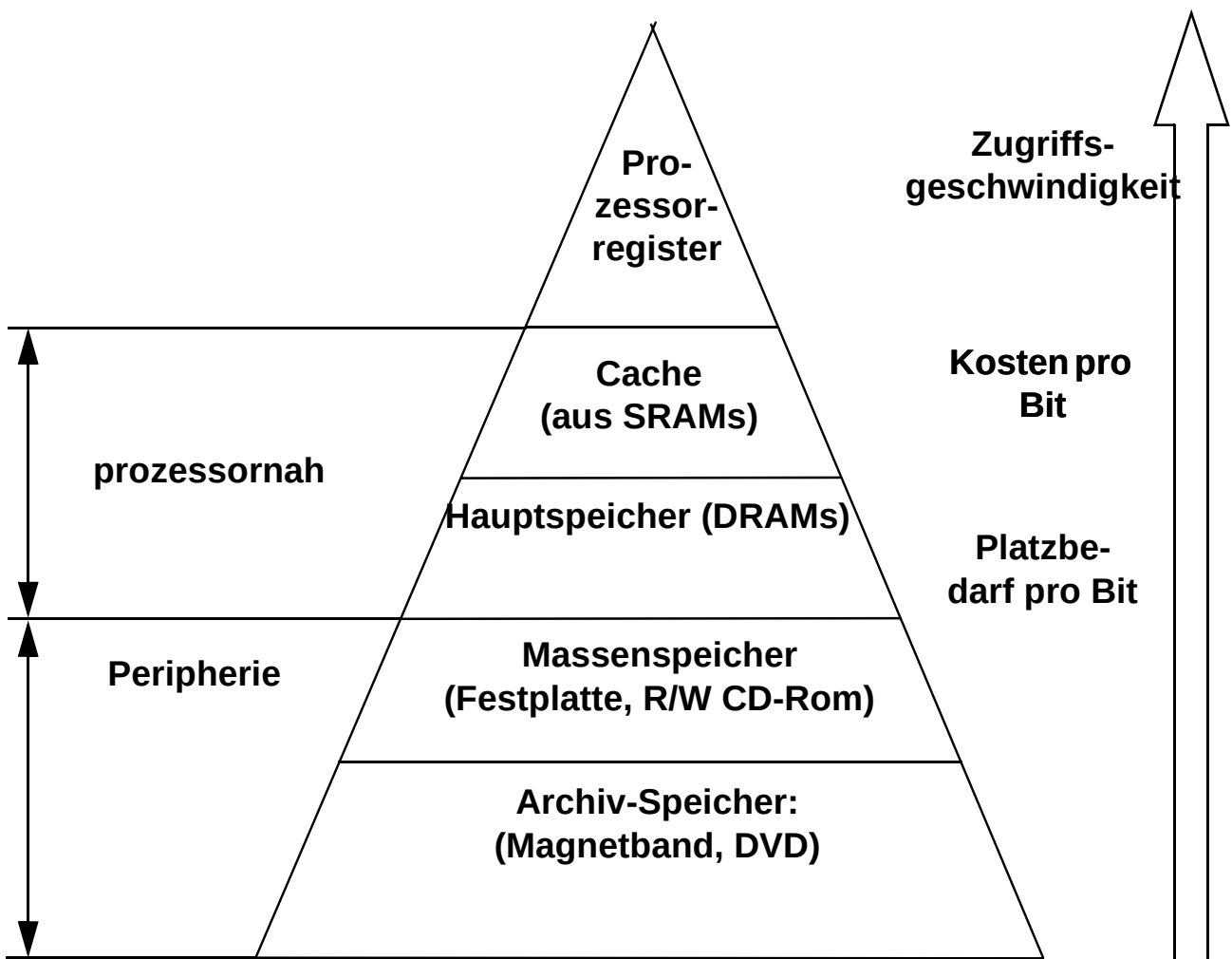
Caches

- Speicherhierarchie
- Lokalitätsprinzip
- Depot von Daten/Befehlen
- Aufbau und Arbeitsweise
- Ersetzungsstrategien
- Das Konsistenzproblem
- Leistung des Caches

Speicherhierarchie

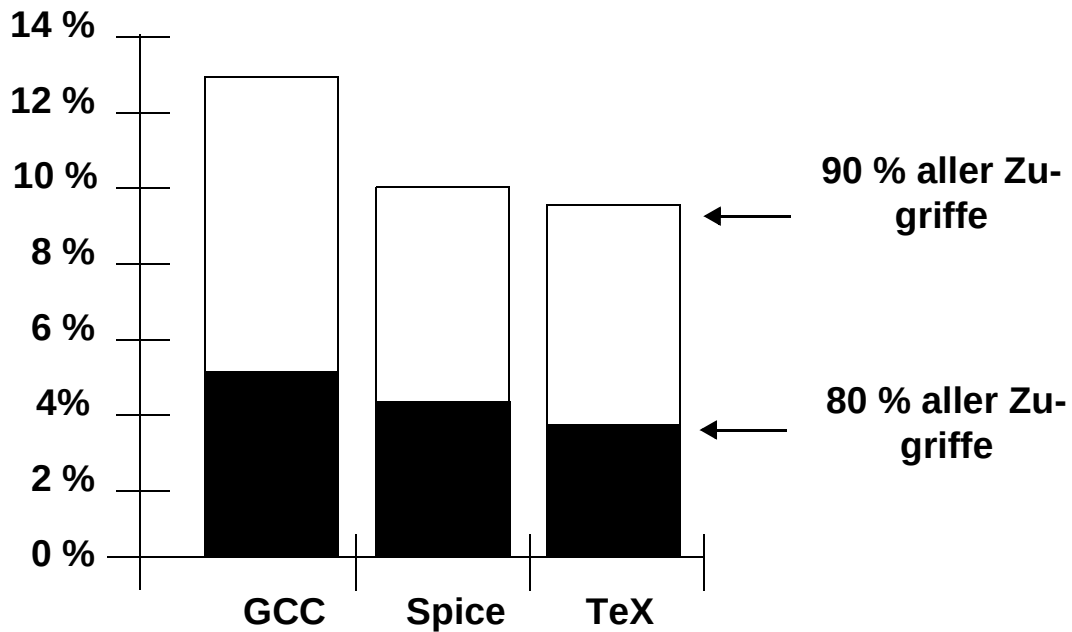
- Ein hierarchischer Speicheraufbau ist ein guter Kompromiss bezüglich Zugriffszeit und Kosten pro Bit

=> Caches sind ein wichtiger Bestandteil der Hierarchie



Befehlslokalität

- Prozentsatz von Befehlen, der für 80 % bzw. 90 % der abgearbeiteten Befehle verantwortlich ist.



GCC= GNU C-Compiler
Spice = Schaltkreissimulator
TeX= Textsystem

[Grafik nach Hennessy/Patterson]

- Faustregel:
Ein Programm verbraucht 90 % seiner Zeit für 10 % des Codes!

Programmeigenschaft „Lokalität“

- **Programme tendieren zur wiederholten Nutzung von Befehlen und Daten**
- **Es gibt Befehlslokalität und Datenlokalität**
- **Für die Lokalität von Befehlen und Daten gibt es zwei Unterfälle:**

1.) Zeitliche Lokalität

Auf ein gerade zugegriffenes Speicherwort wird häufig auch in naher Zukunft zugegriffen werden. (wegen Schleifen)

2.) Räumliche Lokalität

Auf Speicherworte deren Adressen benachbart sind, wird häufig auch in naher Zukunft zugegriffen. (wg. linearer Codeabarbeitung)

- **Datenzugriffe zeigen eine geringere Lokalität als Befehlszugriffe**

=> Aufgrund der Lokalitätseigenschaft macht es Sinn, ein kleines Depot für Befehle und/oder Daten einzurichten

=> Ein benötigter Befehl/Datum steht aufgrund des Lokalitätsprinzips oft im Depot und braucht nicht aus dem langsamen Hauptspeicher geholt werden

- Der Cache ist ein Depot für Befehle und Daten, auf das die CPU schnell zugreifen kann [Cache (engl.) = Depot]
- Da der Cache nur Kopien enthält, muss es ständig aktualisiert werden

Aufbau und Arbeitsweise von Caches

Prinzip:

Es werden beim Cache nicht nur Kopien von Hauptspeicherworten gespeichert, sondern Paare von Daten der Art

(Kennung, Hauptspeicherwort)

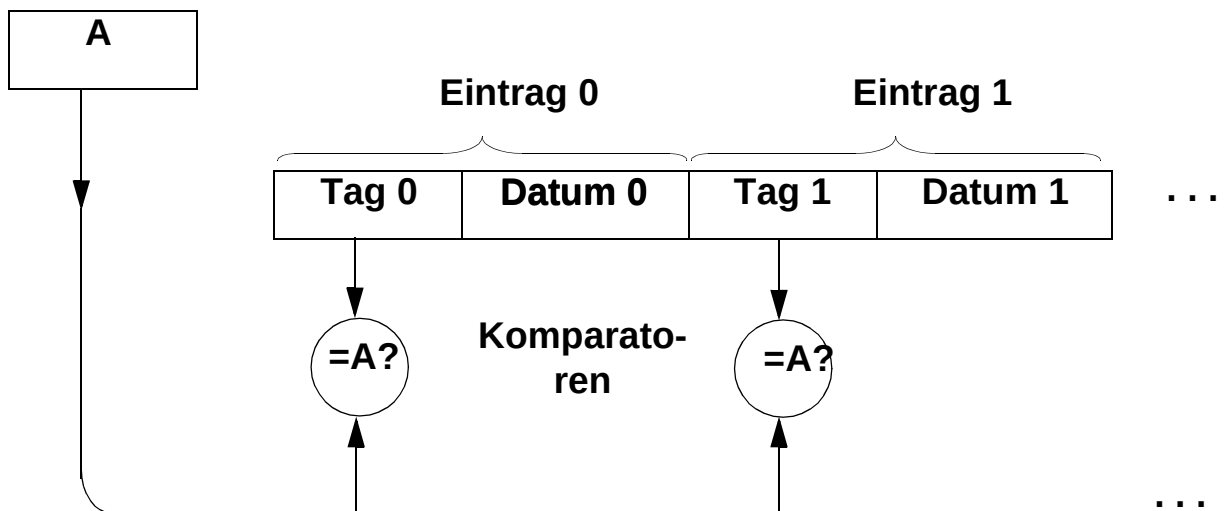
(Kennung, Hauptspeicherwort)

...

u. s. w.

Vollassoziativer Cache

- Ein vollassoziativer Cache enthält eine Folge von Einträgen
 - Jeder der n Einträge besteht aus einem „Tag“-Feld (= Kennung) und einem Datenfeld (Tag = Etikett)
 - Das Tagfeld enthält die Kennung eines gegebenen Eintrages, das Datenfeld die entsprechenden Daten
- gesuch-



- Als Kennung wird bei Rechner-Caches immer direkt oder indirekt die Hauptspeicheradresse des Original-Speicherwortes verwendet, das als Kopie im Cache ist
- => $A \approx$ Hauptspeicheradresse des Datums aus dem Hauptspeicher
- Die gesuchte Kennung wird simultan von allen Komparatoren mit den Tags verglichen
 - Ist ein Tag identisch mit der gesuchten Kennung, wird das dazugehörige Datum ausgegeben

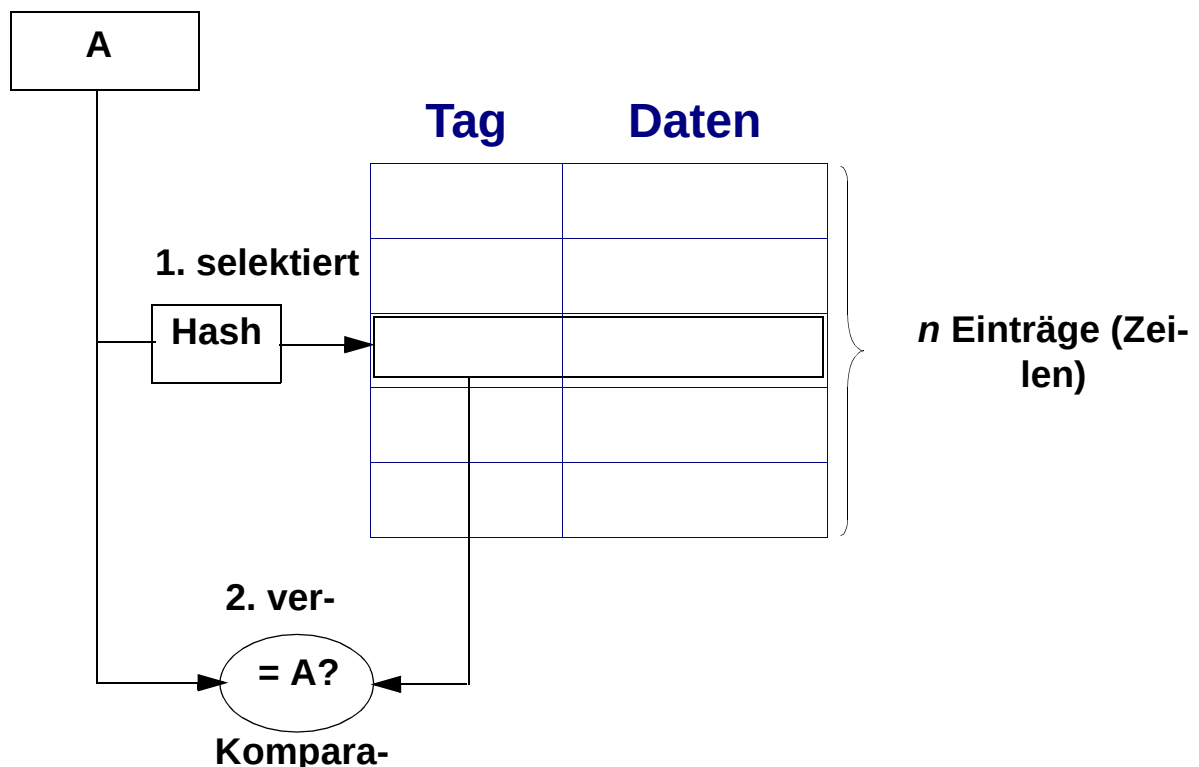
Vollassoziativer Cache - Arbeitsweise

- Bei einem lesenden Cachezugriff wird die Adresse A mit allen n Tags gleichzeitig verglichen.
 - Stimmt ein Tag überein (*Hit*), so wird das betreffende Datenfeld verwendet.
 - Anderenfalls, bei einem *Miss*, muss geeignet reagiert werden, z. B. durch Löschung eines Eintrages und Einspeicherung des gewünschten Datums mit der Hauptspeicheradresse A in dem Cache.
 - Vorteil:
Solange der Cache nicht voll ist, treten beim Schreiben von Einträgen in den Cache keine Probleme auf, da jeder Eintrag an beliebiger Stelle im Cache stehen kann $\Rightarrow$ keine Konflikte
 - Nachteil:
Dieser Cachetyp ist sehr aufwendig, da n Komparatoren nötig sind
-

Direct-Mapped Cache

- Der direct-mapped Cache ist eine Vereinfachung des vollassoziativen Cache
- Jeder der n Einträge besteht aus einem Tag-Feld und einem Datenfeld, genau wie beim vollassoziativen Cache
- Eine Hash-Funktion bildet einen Index I aus einem Eingabewert (= Adresse A). Der Index I selektiert eine Cache-Zeile aus n Zeilen

gesuch-



to hash: zerhak-

Direct-Mapped Cache - Arbeitsweise

- Die Adresse A wird mit dem Tag verglichen, welches in der selektierten Zeile gespeichert ist
- Stimmen Tag und Adresse überein, haben wir einen Hit und das Datenfeld kann verwendet werden
- Anderenfalls, bei einem Miss, muss wiederum geeignet reagiert werden: durch Ersetzen der selektierten Zeile durch das Hauptspeicherdatum, das die Adresse A hat
- Es werden die hinteren Bits von A als Hash-Funktion verwendet
 - ⇒ Die vorderen Bits von A werden im Tagfeld anstelle der vollständigen Adresse gespeichert, d. h. die Bits von A werden geteilt (hashing)
- Vorteil: Dieser Cache ist aufwandsarm. Bei gegebener Chipfläche gestattet er die größte Cache-Kapazität, da nur 1 Komparator nötig
- Nachteil: Beim Einspeichern in den Cache können Schreibkonflikte entstehen, da mehrere Einträge auf ein und denselben Index (Zeile) abgebildet werden
 - ⇒ Nichteindeutigkeit der Hash-Funktion, da modulo-Funktion zum Hashing verwendet wird
- Das Abschneiden der vorderen Bits von A durch das Hashing entspricht einer Modulo-Abbildung

Beispiel für hash-Funktion zur Abbildung von Hauptspeicheradressen auf Cache-Adressen bei Direct-mapped Caches

- 1.) Ein Cache habe Platz für 128 Einträge 0,1, ..., 127 und speichere pro Eintrag jeweils ein Hauptspeicherwort

=>

- Das Datum aus Hauptspeicheradresse 0 oder 128 oder 256 oder 512 oder ... wird in die Cache-Adresse 0 gespeichert
- Das Datum aus Hauptspeicheradresse 1 oder 129 oder 257 oder 513 oder ... wird in die Cache-Adresse 1 gespeichert
- Das Datum aus HS-Adresse 127, 255, 511, ... wird bei 127 gespeichert

=> $\text{Cache-Adresse} = \text{Hauptspeicheradresse} \text{ MOD Zahl der Cache-Einträge}$
 $\text{Cache-Kennung} = \text{Hauptspeicheradresse} \text{ DIV Zahl der Cache-Einträge}$

(DIV = Ganzzahldivision, MOD = Rest nach Division)

2.) Ein Cache habe n Einträge und speichere pro Eintrag jeweils k Hauptspeicherworte

=> Es werden jeweils ganze Blöcke aus k Worten zwischen Hauptspeicher und Cache transportiert.

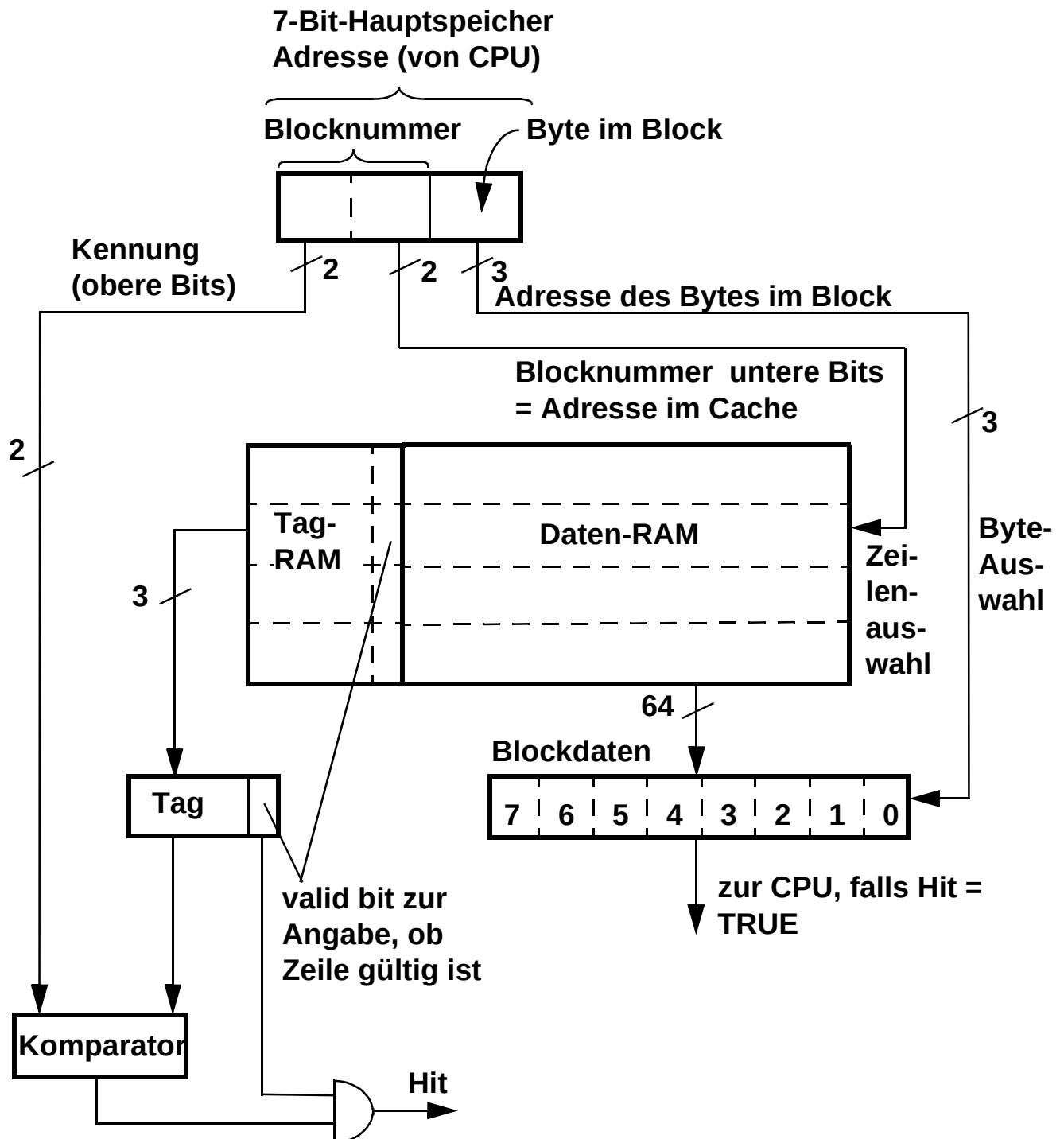
=> der Hauptspeicher bekommt gedanklich eine blockweise Einteilung in Blockadressen

=> Blockadresse = Hauptspeicheradresse DIV k

=>	Cache-Adresse = Blockadresse MOD n Cache-Kennung = Blockadresse DIV n	= die hinteren Bits
		= die vorderen Bits der Blockadresse

Beispiel für den technischen Aufbau eines Direct-mapped Cache

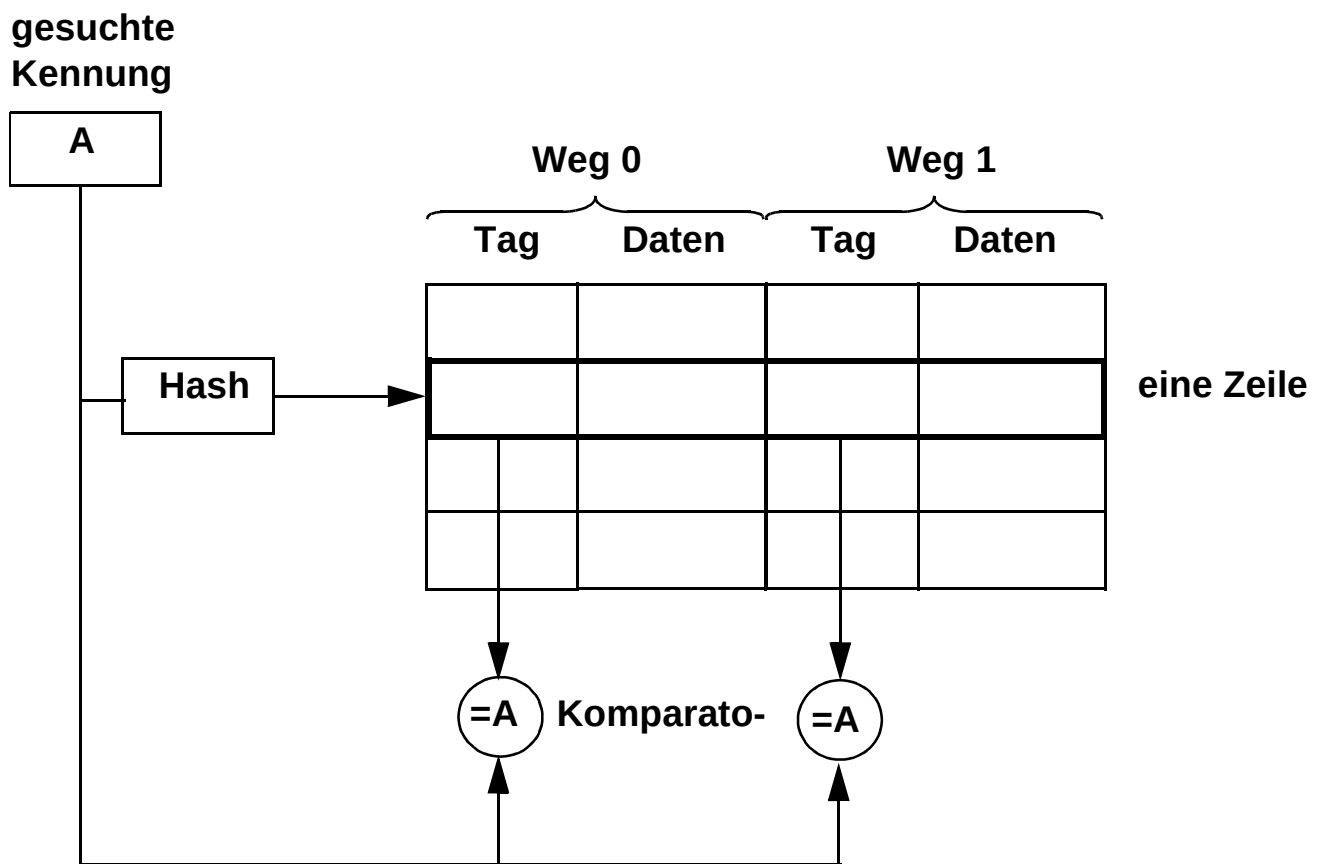
- Block-Größe = 8 bytes, Hauptspeichergröße = 16 Blöcke, Cache-Größe = 4 Blöcke; CPU möchte lesen



m-Wege Mengenassoziativer Cache

- Ein *m*-Wege mengenassoziativer Cache besteht im Prinzip aus *m* kleinen Direct-mapped Caches
- Er ist eine Kreuzung aus vollassoziativem und direktabbildendem Cache

Beispiel: 2 -Wege Mengenassoziativer Cache

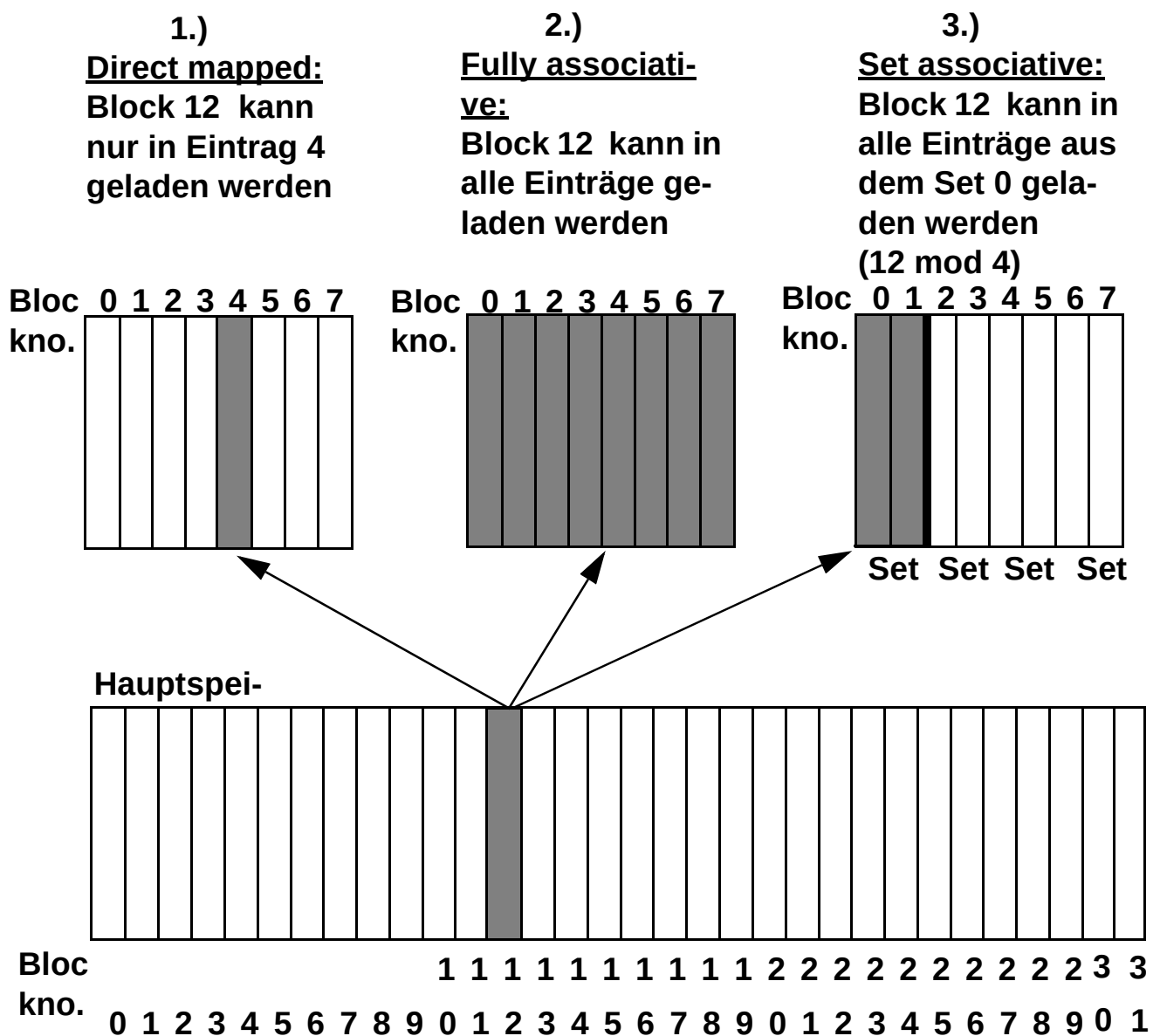


Mengenassoziativer Cache - Arbeitsweise

- Die Hash-Funktion selektiert eine Menge von m Einträgen gleichzeitig $\Rightarrow$ m Möglichkeiten, das gesuchte Datum zu finden
- Alle m Tags werden gleichzeitig mit der Adresse A verglichen
- Bei n Zeilen und m Wegen enthält der Cache $n * m$ Einträge
- Es gibt m Wege, um ein Datum mit Hauptspeicheradresse A in eine Zeile einzuspeichern, weshalb man auch vom m -fach Mengenassoziativen Cache spricht
- Vorteil:
Die m -fache Einspeichermöglichkeit reduziert die Wahrscheinlichkeit für einen Speicherkonflikt auf $\frac{1}{m}$ im Vergleich zu einem Direct-Mapped Cache
- Nachteil:
Im Vergleich zum Direct-Mapped Cache benötigt er mehr Chipfläche, da m Komparatoren nötig sind

Vergleich zwischen Direct Mapped-, Voll Assoziativem- und Mengenassoziativem Cache

- Gegeben seien 3 Caches mit je 8 Einträgen
- Es soll der Block Nr. 12 aus dem Hauptspeicher in den Cache geschrieben werden
- An welchen Stellen kann Block Nr. 12 stehen?



Wann und wie werden Daten zwischen Hauptspeicher und Cache transferiert?

- Mechanismen für den Transfer zwischen Cache und Hauptspeicher
 - Der Hauptspeicher wird logisch in Blöcke unterteilt
 - Der Cache wird in eine kleine Menge von Blöcken eingeteilt, die die gleiche Größe wie die Hauptspeicherblöcke haben
 - Ein Cache-Block ist entweder leer oder enthält eine Kopie eines Hauptspeicherblocks
 - Der Cache enthält ein Verzeichnis der aktuell im Cache enthaltenen Hauptspeicherblöcke in Form der Kennungen (Tags)
- Wenn ein Hauptspeicherwort von der CPU gelesen werden soll:
 - wird aus der Adresse die Hauptspeicher-Blocknummer ermittelt
 - nach dieser Blocknummer im Cache-Verzeichnis nachgesehen (= Vergleich mit dem Tag/den Tags)
 - wenn eine Kopie des Blockes gerade im Cache ist, werden die Daten aus dem entsprechenden Cache-Block ermittelt: wir haben einen Hit
 - sonst: wir haben ein Miss: eine Kopie des Blockes wird vom Hauptspeicher in einen Cache-Block transferiert (in einen Cache-Block passt genau ein Hauptspeicherblock)

Problemfall:

Es gibt einen Cache-Miss bei lesenden CPU-Zugriff, und der entsprechende Eintrag im Cache ist nicht frei zur Einspeicherung des gewünschten Hauptspeicherblocks.

Was ist zu tun?

- Wenn der Cache-Block, der den gesuchten Hauptspeicherblock aufnehmen soll, nicht leer ist, muss er überschrieben werden**
- Dabei sind 2 Fälle zu unterscheiden:**
 - 1.) wenn die Daten in zu verwendenden Cache-Block nicht modifiziert werden, brauchen sie nicht weiter betrachtet zu werden und können überschrieben werden**
 - 2.) sonst muss eine Kopie in den Hauptspeicher vorher zurückgeschrieben werden, bevor überschrieben wird, sonst sind die Daten verloren**

Ersetzungsstrategien bei vollassoziativem Cache und mengenassoziativem Cache

- Was tun, wenn Cache voll ist?
Welcher Cache-Eintrag soll überschrieben werden?

⇒ „Ersetzungsstrategien“ für die Wahl des richtigen Cache-Blockes
- Beim Direct-mapped Cache stellt sich diese Frage nicht, da die hash-Funktion keine Wahlmöglichkeit für die Einspeicherung von Hauptspeicherblöcken in Cache-Blöcke hat
- Beim mengenassoziativen- und beim vollassoziativen Cache hingegen gibt es Freiheitsgrade für die „Blockersetzung“

Die Ersetzungsstrategie entscheidet, wohin die Kopie eines Blockes innerhalb des Caches gehen soll

- Zufällige Ersetzung: irgendeiner der möglichen Cache-Blöcke wird durch den neuen Block ersetzt
- LRU (least recently used)-Ersetzung:
 - Idee: wenn ein Cache-Block eine Weile nicht mehr benutzt worden ist, wird er wohl auch nicht mehr bald benutzt werden
- Für kleine Caches ist LRU besser, aber teuer: normalerweise wird eine Näherungslösung anstelle von LRU in den Cache eingebaut
- Für Caches, die groß genug sind, gibt es kaum einen Leistungsunterschied zwischen zufälliger Ersetzung und LRU, so dass die erstere gewählt wird, da einfacher zu implementieren

Das Konsistenzproblem **zwischen Cache und Hauptspeicher**

- Angenommen, die CPU möchte eine Speicherzelle beschreiben, die sich in einem Block befindet, der zuvor als Kopie in den Cache geladen wurde
- Wohin soll die Information geschrieben werden?
 - a) in den Cache
 - b) in den Hauptspeicher
 - c) in beides
- In den Fällen a) und b) gibt es zwei Blöcke, deren Inhalt nicht mehr deckungsgleich ist (= Konsistenzproblem)
- Man spricht in dem Fall, dass zunächst nur der Block im Cache modifiziert wird (= Fall a) von einem „dirty block“, da er inkonsistent mit dem Originalblock im Hauptspeicher ist

Schreibstrategien in Cache und Hauptspeicher

- Wenn der Hauptspeicherblock, in den geschrieben werden soll, als Kopie in Cache vorliegt, gibt es 2 Möglichkeiten:

1.) **Write through**: der Hauptspeicher wird jedesmal wenn der Cache modifiziert wird, ebenfalls aktualisiert

- Um die langsamen Hauptspeicherzugriffe zu verringern, genügt es, in einen schnellen „Schreibpuffer“ zu schreiben, der anschließend langsam in den Hauptspeicher übertragen wird

2.) **Write back**: Daten werden nur in den Cache geschrieben; der Hauptspeicher wird nur aktualisiert, wenn der Inhalt eines „dirty“ Blocks aufgrund einer Ersetzungsstrategie überschrieben werden muss

- Wenn der Hauptspeicherblock, in den geschrieben werden soll, nicht als Kopie im Cache vorliegt, gibt es auch 2 Möglichkeiten:

1.) **Fetch on write**:

Der Block wird zuerst in den Cache kopiert und dann dort modifiziert. Dies wird typischerweise eingesetzt, wenn auch die write-back-Strategie gewählt wurde

2.) **Write around**:

Die Daten werden direkt in den Hauptspeicher geschrieben; der entsprechende Block wird nicht in den Cache kopiert

Die Leistung des Caches

- hängt ab von:

- dem auszuführenden Programm, d.h. der Anwendung (= engl. application)
- Cachegröße und Blockgröße
- Organisation (direct, set associative, associative)
- Cache-Strategien bei Ersetzung und bei Schreiben

- Leistungskriterien sind z. B.

- Hit-Rate
 - * nur ein Aspekt der Cache-Leistung
 - * wegen seiner Einfachheit benutzt
- Effektive d. h. tatsächlich wirksame Speicherzugriffszeit: Realität ist komplizierter
 - * hängt von Hauptspeicher- und Busparametern ab
 - * fügt weitere Parameter und Komplexität bei der Untersuchung von Caches hinzu

Cache-Größe als Parameter

- kleiner Cache:

- geringer Hardware-Aufwand (billig)
- viele kapazitäts-und konfliktbedingte Nicht-Treffer (Misses)
- sehr kleine Caches arbeiten unabhängig von ihrer Organisation nur sehr schlecht

⇒ *zu kleine Cache bringen nichts*

- großer Cache:

- hoher Hardware-Aufwand (teuer)
 - Leistung hängt von der Größe und der Ersetzungsstrategie ab
 - sehr große Caches arbeiten auch unabhängig von ihrer Organisation gut
- für Caches mittlerer Größe gilt: 2- oder 4-Wege-Set assoziative Caches liegen sehr dicht an der Leistung von voll assoziativen Caches, wobei set-assoziative Caches sehr viel weniger Hardware erfordern)

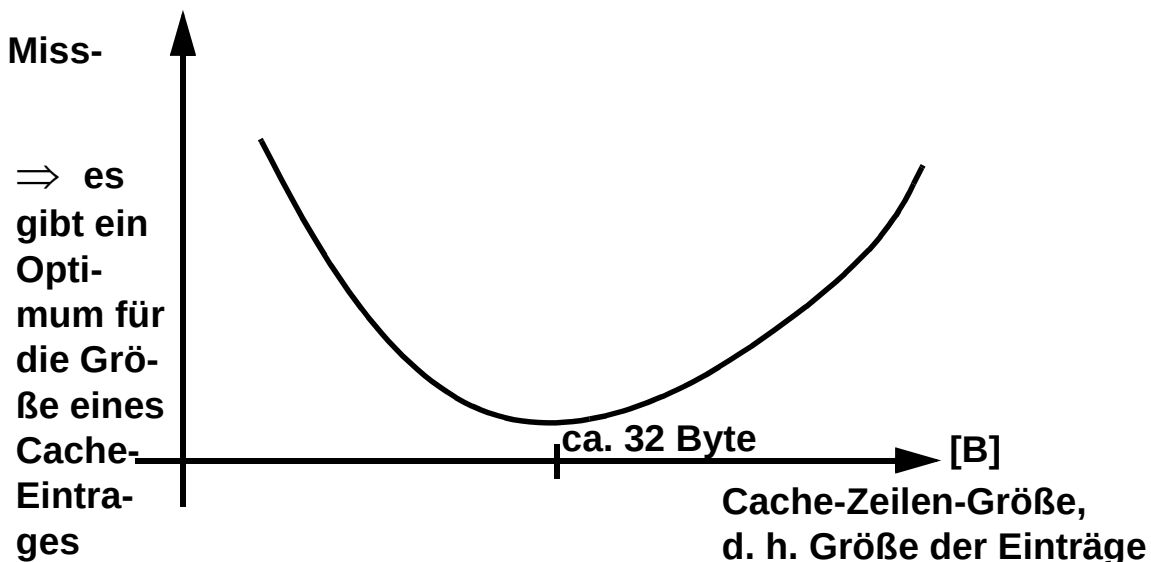
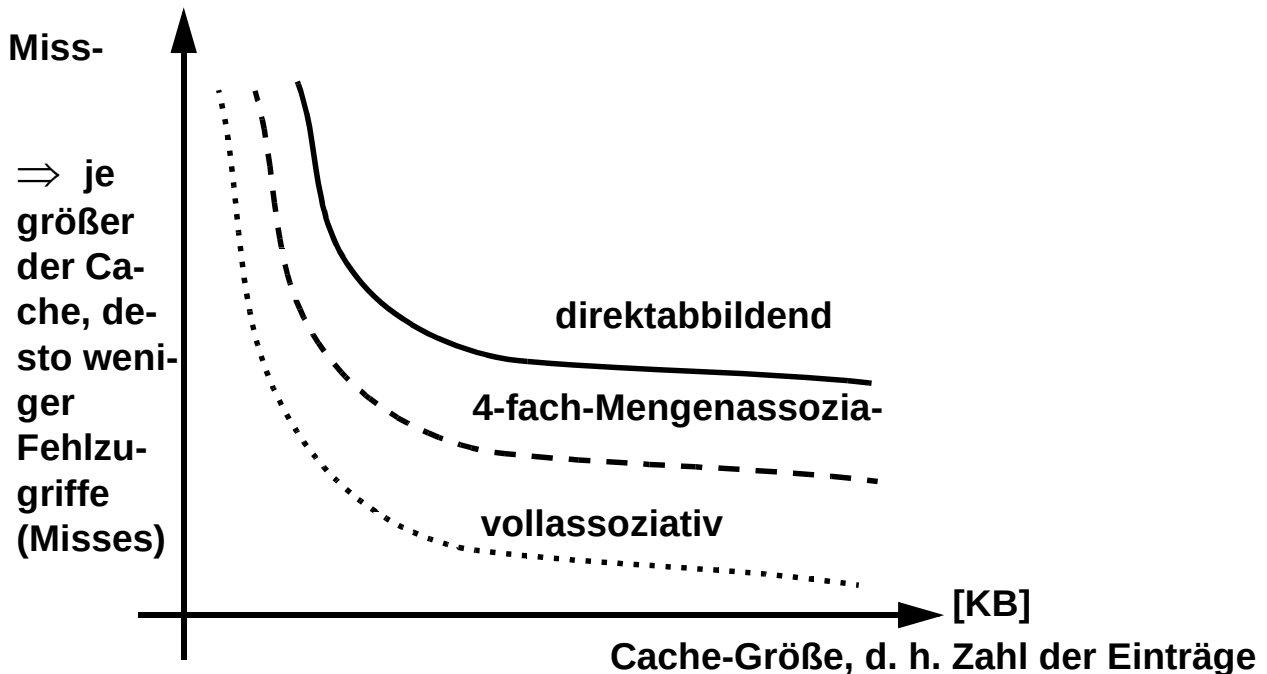
⇒ *mengenassoziative Caches werden am meisten eingesetzt*

Gründe für Cache Misses beim Lesen

- **Es gibt zwei Möglichkeiten, warum ein Datum nicht im Cache steht:**
 - 1.) Der Block, in dem das Datum enthalten ist, war bisher noch nie im Cache eingespeichert**
 - 2.) Der Block ist nicht mehr im Cache, weil er infolge fehlender Cache-Kapazität überschrieben wurde**
- **Ein wichtiger Spezialfall zu 2.) ist der folgende:**
 - Der Block ist nicht mehr im Cache, weil er abwechselnd mit einem anderen Block immer wieder ein- und ausgelagert wird**
 - Dieser Effekt der wechselseitigen, periodischen Verdrängung ist analog zu dem Effekt des „Seitenflatterns“, bei dem sich zwei Pages wechselseitig aus dem Hauptspeicher verdrängen**

Miss-Rate

- Cache-Misses bei typischen Programmen



- heutige Caches haben meistens 32 Byte pro Cache-Eintrag (= 1 Cache Line)

Cache: Effekt auf die Geschwindigkeit

- Der wirksame durchschnittliche Geschwindigkeitsgewinn S liegt bei

$$S = \frac{T_{HS}}{T_E}$$

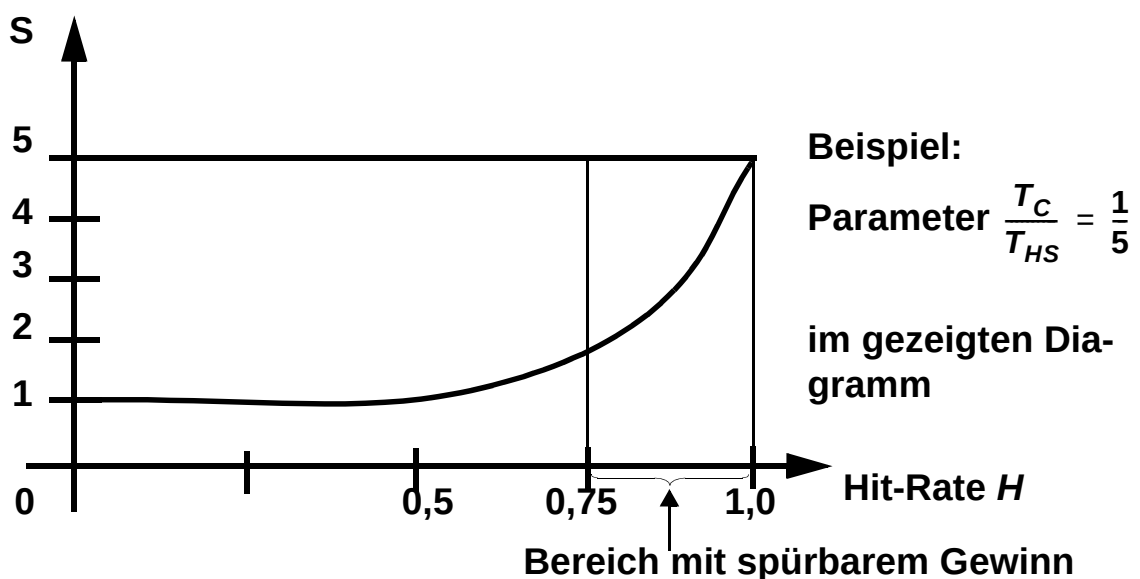
(S = Speedup, T_{HS} = Hauptspeicherzugriffszeit, T_E = Effektive Cache-Zugriffszeit)

mit
$$T_E = H * T_C + (1-H) T_{HS}$$

(T_C = Cache-Zugriffszeit und H = Hit-Rate zwischen 0...1)

ergibt sich

$$S = \frac{T_{HS}}{HT_C + (1-H)T_{HS}} = \frac{1}{1 - H\left(1 - \frac{T_C}{T_{HS}}\right)}$$



Split Caches (Teilung des Cache)

- Einige Implementierungen teilen den Cache auf in:
 - Daten-Cache
 - Befehls-Cache
- Vorteile der Aufteilung des Caches:
 - „verdoppelt“ die Bandbreite der CPU
 - Gelegenheit, jeden Cache unter Berücksichtigung der unterschiedlichen Charakteristiken des Daten- und Befehlsflusses zu optimieren
 - * Befehle müssen nur gelesen werden
 - * Befehle haben eine bessere örtliche und zeitliche Lokalität als Daten
 - => Befehlscache ist anders organisiert als Datencache bzgl. Ersetzungsstrategien, Schreibstrategien, etc.
- Nachteil der Cache-Aufteilung
 - Aufrechterhaltung der Konsistenz zwischen den beiden Caches (bei selbst-modifizierendem Code)

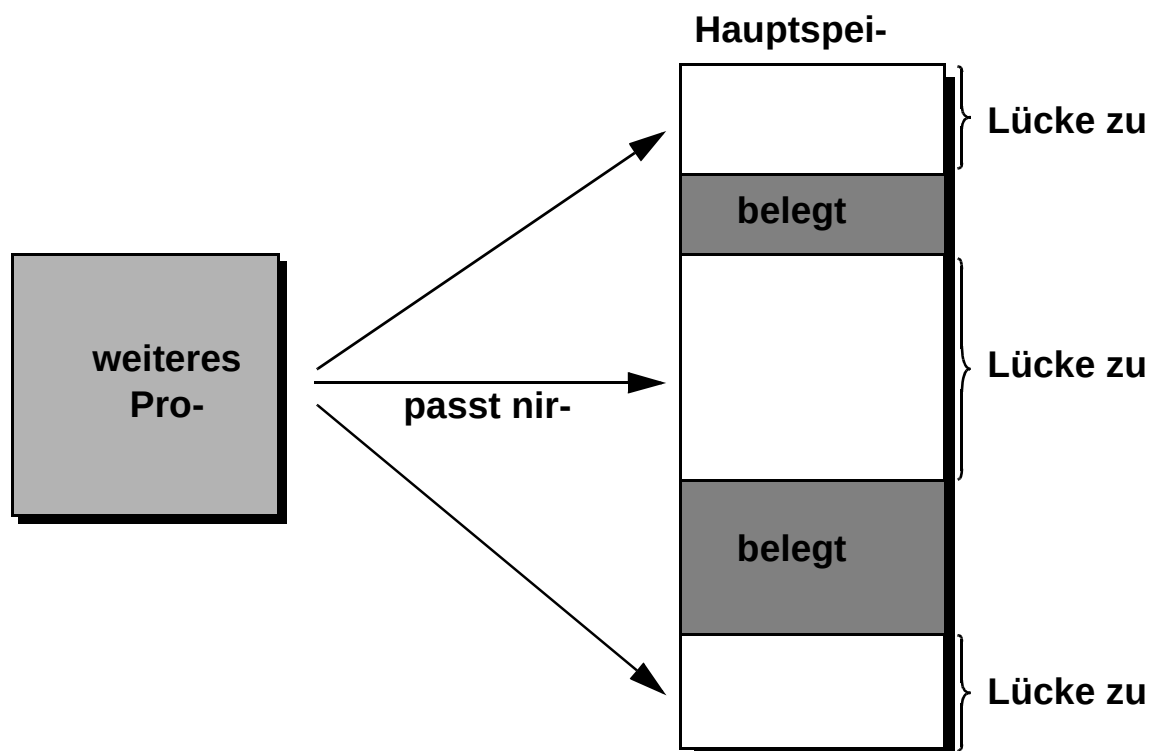
Hauptspeicher und MMU

MMU = Memory Management Unit

= Speicherverwaltungseinheit

Problem der nicht zusammenhängenden Adressräume

- Im Laufe des Betriebs entstehen Lücken im Speicher, die nicht genutzt werden können (sog. Verschnitt)
- Ein Verschieben der Speicherbereiche ist zu aufwendig
- Beispiel:



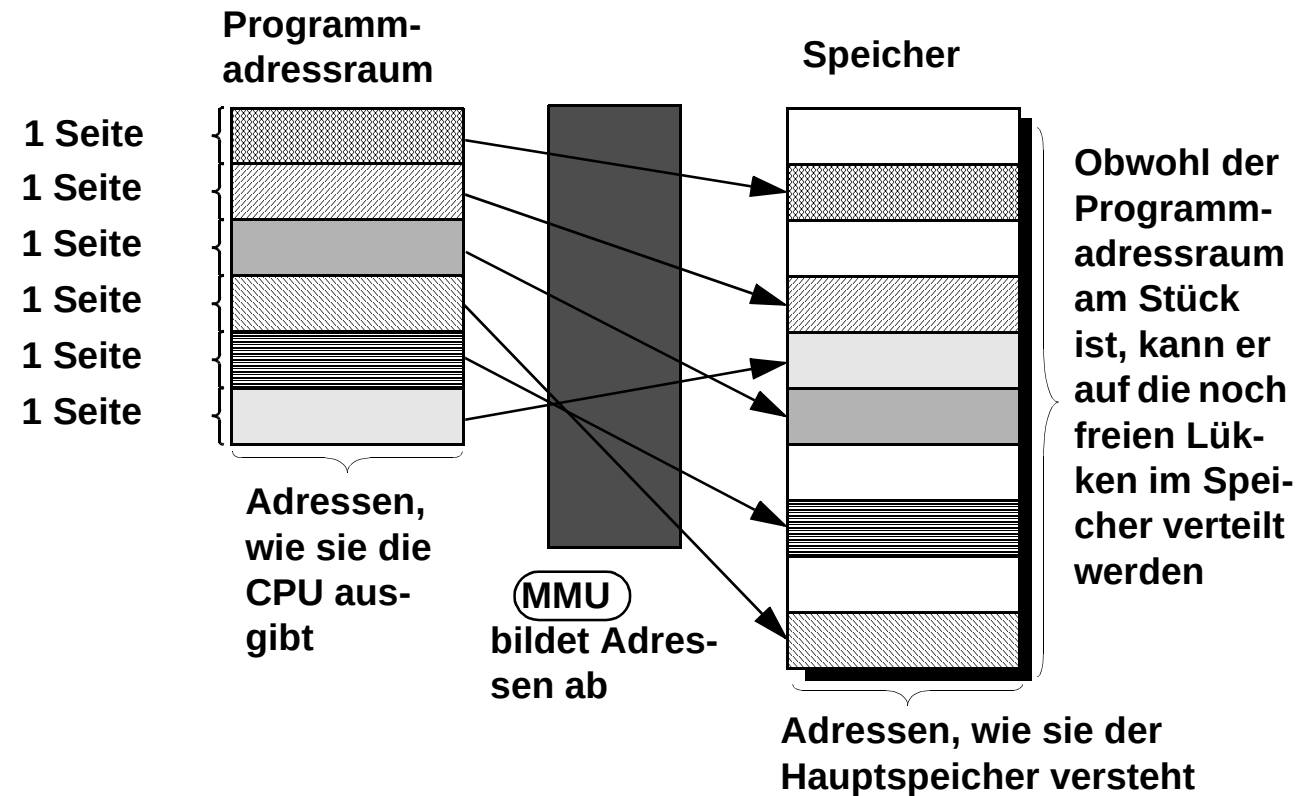
Lösung des Problems: Speicheraufteilung in Seiten (pages)

- Der Programmadressraum wird in kleinere Stücke von jeweils gleicher Größe (sog. Seiten) zerlegt, die unabhängig voneinander im Speicher abgelegt werden
- Für die Zerlegung müssen die Prozessoradressen von den Hauptspeicheradressen getrennt werden

=> Man spricht von dynamischer Adressumsetzung

- Eigenschaften der dynamischen Adressumsetzung:
 - Geringerer Verschnitt (Positiv)
 - Höherer Aufwand beim Adressieren (Negativ)
 - Die Adressen im Programm („logische Adressen“) werden durch eine spezielle Einrichtung des Prozessors in „physikalische Adressen“ umgesetzt.
 - Die Einrichtung heisst Speicherabbildungseinheit, (*Memory Management Unit, MMU*)
 - Die Unterscheidung zwischen Prozessoradressen und Adressen, wie sie der Hauptspeicher versteht, hat sich bei praktisch allen Prozessoren durchgesetzt, auf denen ein Plattenbetriebssystem ausgeführt werden kann
- => Dynamische Adressumsetzung durch eine MMU ist Standard

Adressabbildung durch die MMU



Erweiterung der dynamischen Adressumsetzung zum virtuellen Speicher

- Aufgrund der Lokalitätseigenschaft von Programmen reicht es aus, nur den Teil des Programmadressraums im Hauptspeicher zu halten, der gerade benötigt wird
- Der große Rest des Programms verbleibt auf der Festplatte in einer speziellen Datei, der Auslagerungsdatei (pagefile.sys) oder einer Partion (swap partition)

Virtueller Hauptspeicher

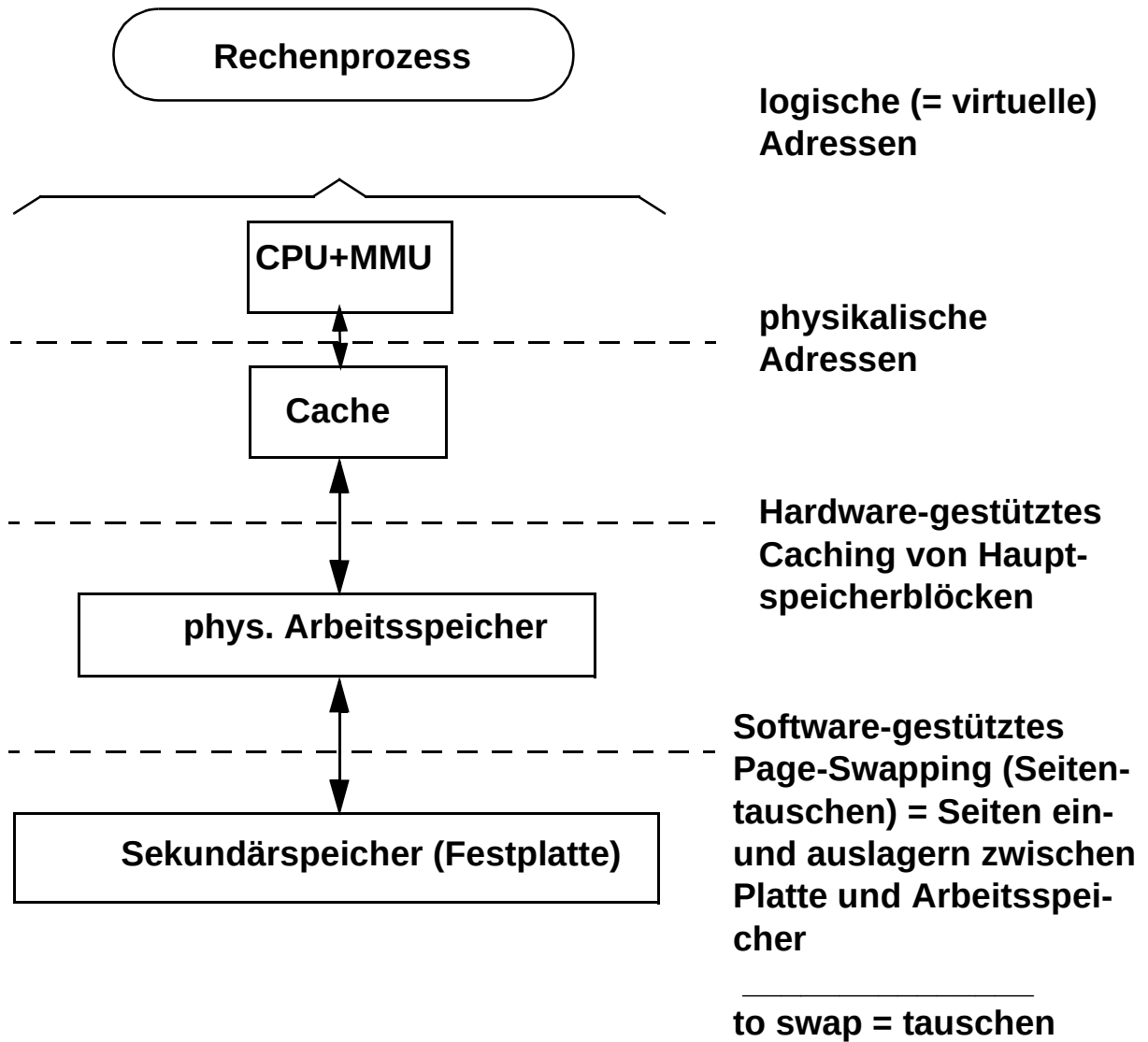
- Das Programm und der Hauptspeicher werden in gleichlange Stücke (Seiten bzw. pages) eingeteilt
- Wird auf eine Seite zugegriffen, die nicht im Hauptspeicher ist, sendet die MMU ein Signal zum Betriebssystem, die „page fault exception“ (Seitenfehlen)
- Anschliessend wird der Prozess, dem die Seite fehlt, „descheduled“
- Das Betriebssystem lädt die fehlende Seite von der Festplatte nach (= „paging“ oder „swapping“)
- Sobald die Seite im Hauptspeicher ist, wird der Prozess „rescheduled“
- Ist im Speicher kein Platz frei, wird eine andere Seite auf den Plattenspeicher ausgelagert
- Es existieren für die Verdrängung einer Seite dieselben Ersetzungsstrategien wie bei den Caches

Vorteile des virtuellen Hauptspeichers

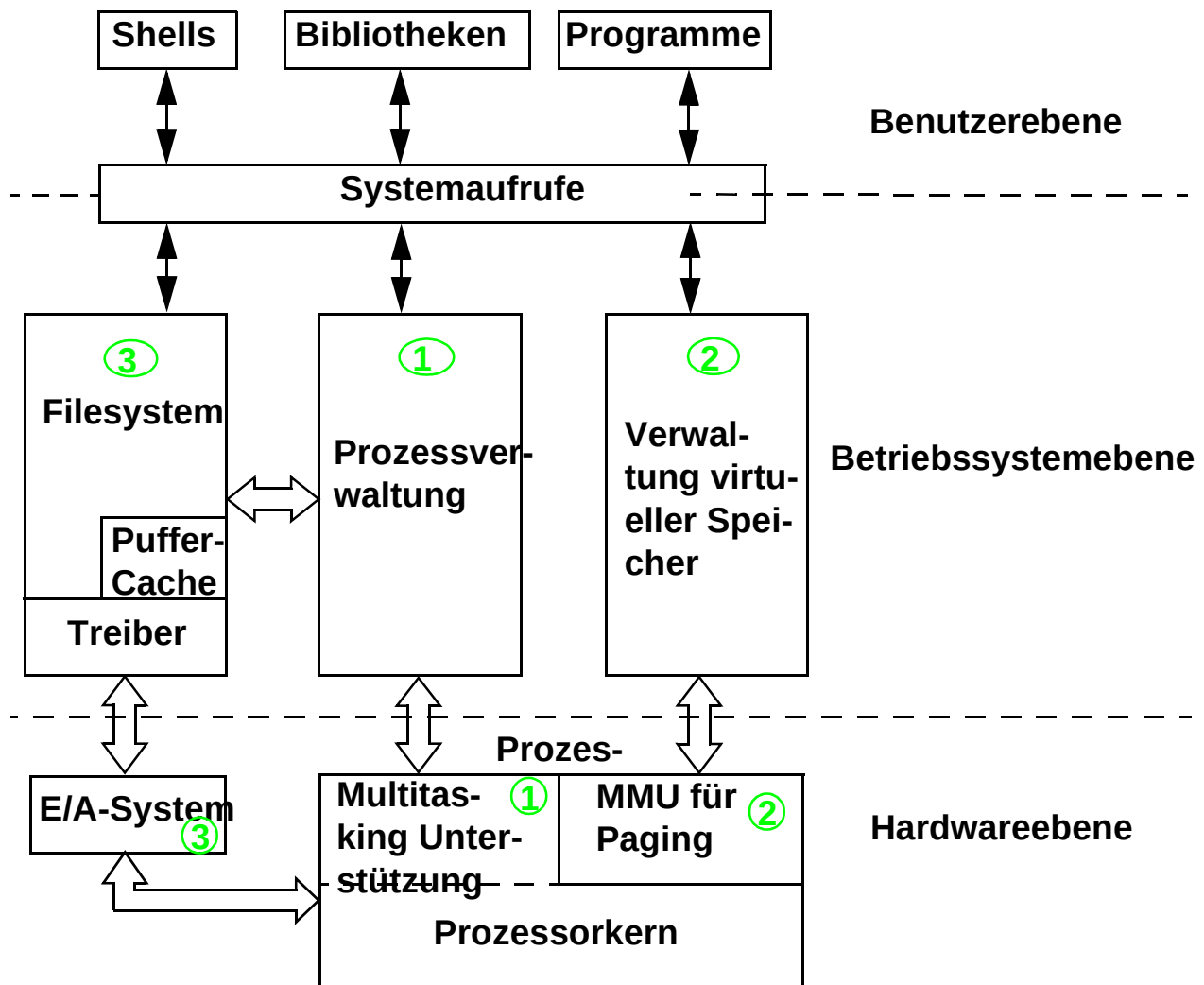
- 1.) Es können grosse Programme ausgeführt werden, die mehr Platz benötigen, als vorhanden ist
- 2.) Es können im timesharing mehr Programme ausgeführt werden als in den Hauptspeicher passen

- Die MMU erweitert den realen Hauptspeicher um die Grösse der Auslagerungsdatei
 - Die Grösse der Auslagerungsdatei wird nur durch den zur Verfügung stehenden Plattenplatz begrenzt
 - Beides zusammen, pages im Hauptspeicher und auf der Festplatte, wird als virtueller Speicher bezeichnet, da der reale Hauptspeicher grösser erscheint als er ist
 - Praktischerweise entspricht die Seitengröße des virtuellen Speichers der Grösse, in der Daten zwischen Plattenspeicher und Hauptspeicher transportiert werden (z. B. 4 KByte)
- => Grösse einer Seite im Hauptspeicher = Grösse eines Records auf der Platte

DIE CPU sieht nur „logische“ Prozeßadressen



Einordnung der MMU in die Prozessor-Hardware und in die Betriebssystem-Software

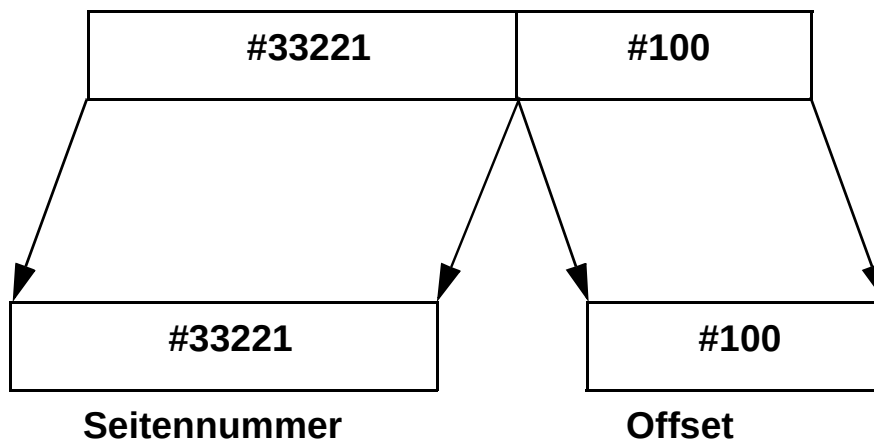


shell (engl.) = Schale; Muschel

Adressumsetzung

- Eine virtuelle Prozessoradresse wird durch eine Abbildungsfunktion - einer Mapping-Funktion - auf eine physikalische Adresse abgebildet. (Mittels sog. MMU, Seitentabellen und dem Betriebssystem)
- Eine beliebige Adresse kann nicht beliebig abgebildet werden. Nur ganze Blöcke sind frei abbildbar. (=“Seite, page“)
- Typische Seitengrößen sind 4 bzw. 8 KB.
- Eine virtuelle Adresse besteht aus 2 Komponenten: einer *virtuellen Seitennummer* und dem *Offset (Versatz)*

Prozessoradresse z. B. :
=Virtuelle Adresse #33221100



- In eine physikalische Adresse ist nur die virtuelle Seitennummer umzuwandeln. Der Offset bleibt unverändert.

- Der virtuellen Seitennummer muss zur Laufzeit eine physikalische Seitennummer zugeordnet werden.
- Die Zuordnung wird einer Seitentabelle (Map/Pagetabelle) entnommen. (= Listenstruktur im Hauptspeicher). Dazu liest ein Programm des Betriebssystems die Tabelle.

Virtuelle Adresse von der CPU

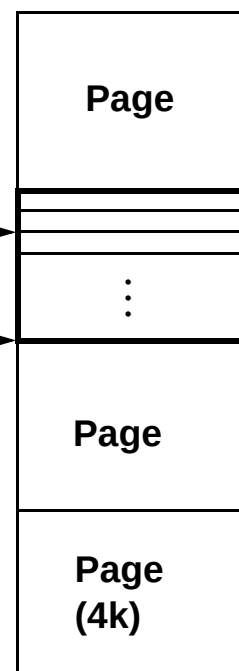
Seitennr.	Offset
-----------	--------

Valid-
Bit

Page Table

	virtuelle Adr.	phys. Adresse
		⋮

Physikalischer Speicher
= Hauptspeicher



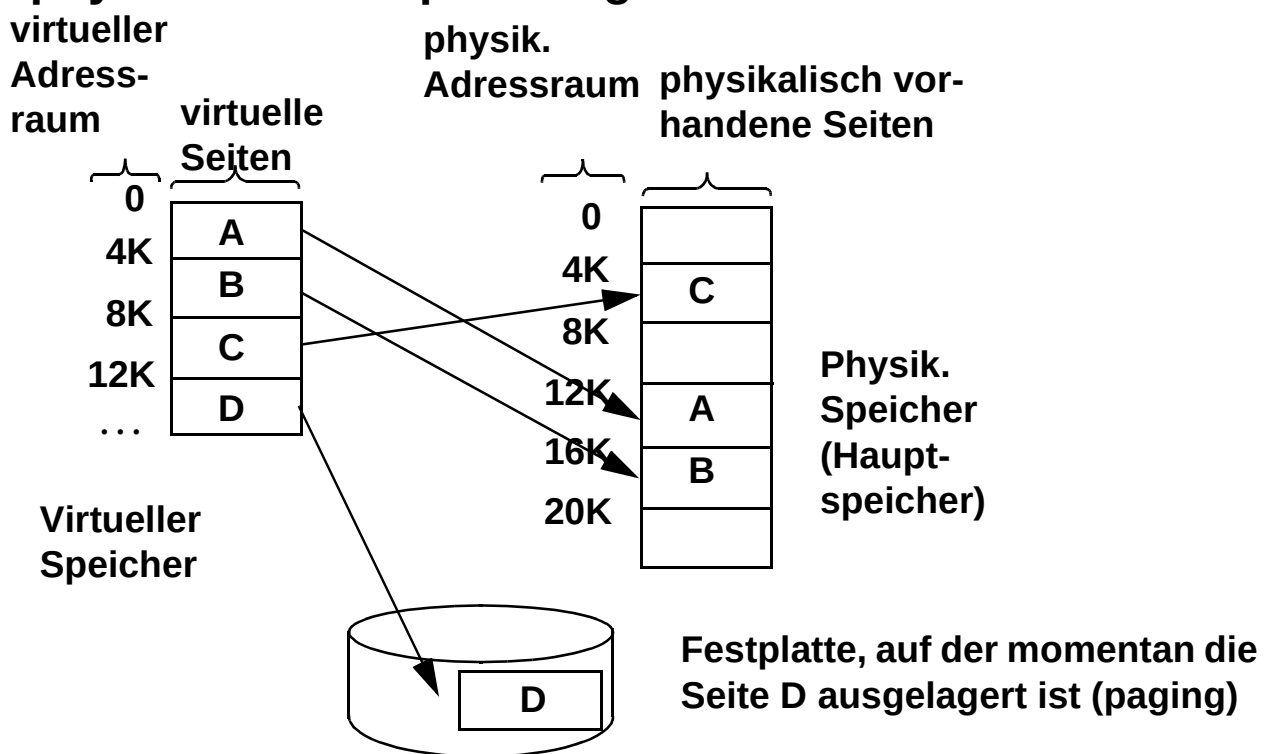
Wortan-
fang

Seiten-
anfang

Das Betriebssystem lädt Seite von
Platte, wenn Valid-Bit gelöscht ist

Virtueller Adressraum

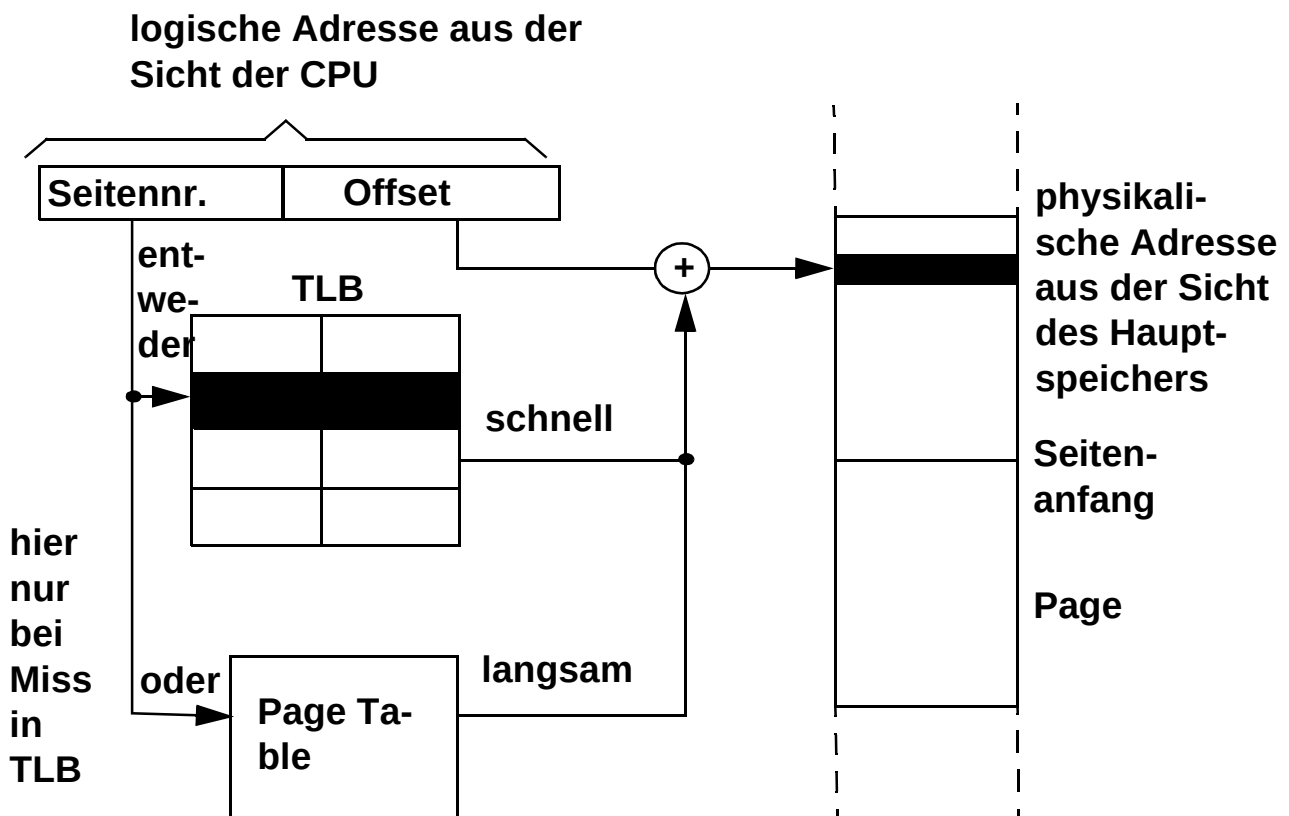
- Die Adressumsetzung erlaubt es, dass der Speicher, den die CPU adressiert, größer als der physikalische sein kann (= virtueller Adressraum)
- Gibt es zu einer virtuellen Seite momentan keine zugehörige physikalische Seite, wird diese durch ein zusätzliches Flag - das *valid bit* - gekennzeichnet (valid = FALSE)
- Werden fehlende Seiten angesprochen, muss der „Zugriff“ warten bis das Betriebssystem die Seite in den physikalischen Speicher geladen hat.



- Die MMU leistet Unterstützung bei der Adressabbildung
- Wichtigster Teil der MMU ist der Translation Lookaside Buffer.

Translation Lookaside Buffer

- Die Seitentabelle wird vom Betriebssystem im Hauptspeicher und auf der Festplatte verwaltet
- Um schnelle Abbildungen zu ermöglichen, wird die Seitentabelle in Teilen in einem kleinen, schnelleren, auf der CPU befindlichen Zwischenspeicher - dem sogenannten „Translation Lookaside Buffer“ (TLB) - als Kopie gehalten
- Der TLB ist vollassoziativer Cache organisiert. Er sorgt für die Umsetzungsinformationen bei aktuell benötigten Seiten
- Nur bei Cache Miss wird die Pagetable konsultiert



Mindestanforderungen für den Aufbau von Page Table und TLB

- mind. 2-spaltige Einträge in die Page Table für virtuelle/ physikalische Seitenadressen
- Ein *Valid*-Bit für jeden Eintrag, d.h. jede Seite zur Verwaltung der Information „Page present in memory“
- Zusätzliche muss die CPU haben:
 - Ein Trap bzw. Exception-Mechanismus, ausgelöst im „Page not present“-Fall
 - Wiederaufsetzbare Befehle, im Fall eines Seiten Fehlens (d. h. Befehle können ohne störende Seiteneffekte vorzeitig abgebrochen und danach wiederholt werden)

Erweiterte Anforderungen für den Aufbau der TLB

- Zusätzlich sollte die TLB für Speicherschutz sorgen
- Speicherschutz ist unerlässlich für Multitasking und/ oder Multiuser-Betrieb
- Ist Speicherschutz vorhanden, spricht man von einer Memory Management Unit (MMU), die als wichtigste Komponente die TLB hat

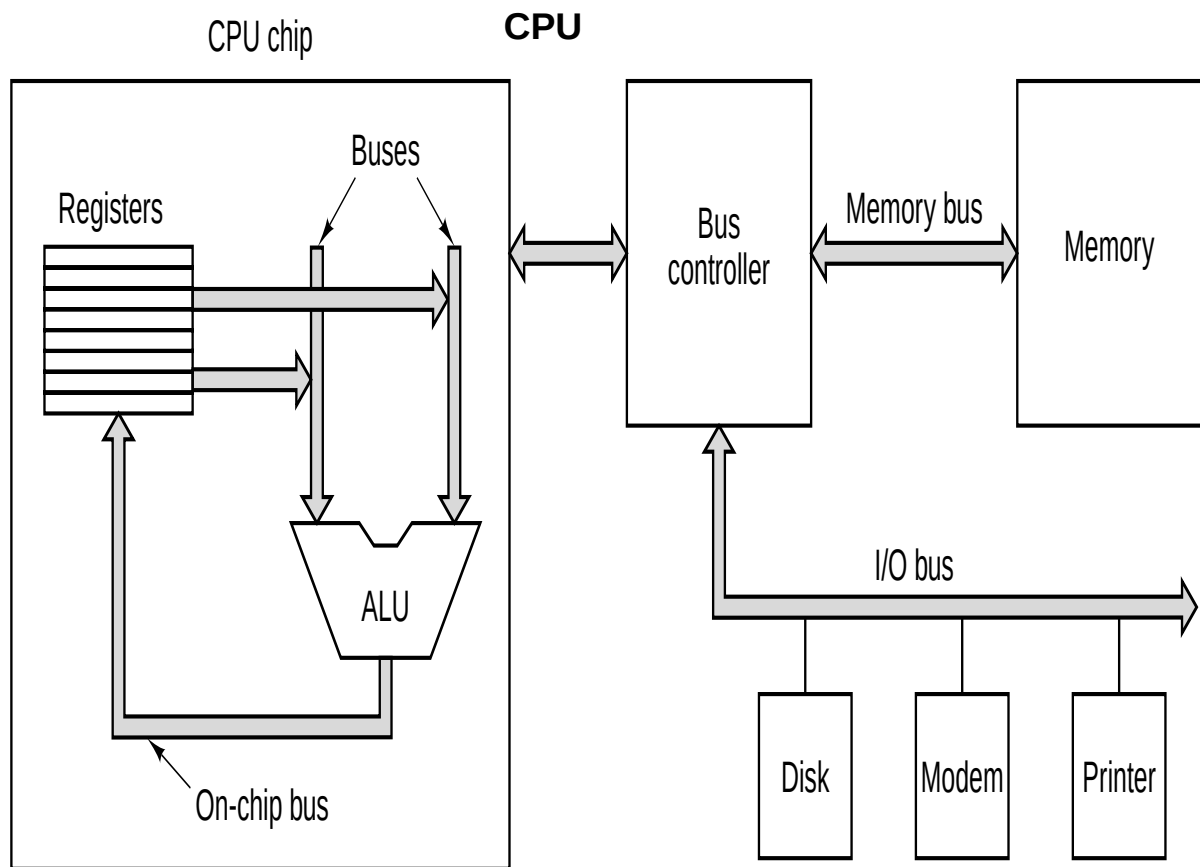
-
- Trap = Falle; exception = Ausnahme

Speicherschutz durch das Betriebssystem mit Unterstützung der MMU

- Bei Multitasking/Multiuser-Systemen ist notwendig:
 - a) Schutz durch das Betriebssystem vor unberechtigten Anwenderzugriffen, d. h.
 - unberechtigten Datentransfer (Lesen, Schreiben) und
 - unberechtigten Control-Transfer (Prozeduraufrufen, Eintrittspunkte)
 - b) Schutz vor Zugriffen zwischen Tasks (Task-Isolation)
- Dazu sind in der MMU für jeden Seiteneintrag zusätzlich Bits enthalten, die vom Betriebssystem geschrieben und von der MMU automatisch bei jedem Seitenzugriff überprüft werden. (=“Schutzbits“)
- Diese Bits enthalten Privilegien und Zugriffsinformationen für alle Seiten eines Prozesses
- Sie sind zusätzlich zum Valid-Bit im TLB in weiteren Spalten pro Eintrag gespeichert
- Privilegien sind Rechte, die einzelnen Benutzern oder Prozessen zugewiesen werden. BSP.: Administratorrechte = Recht zur Ausführung eines Systemprozesses

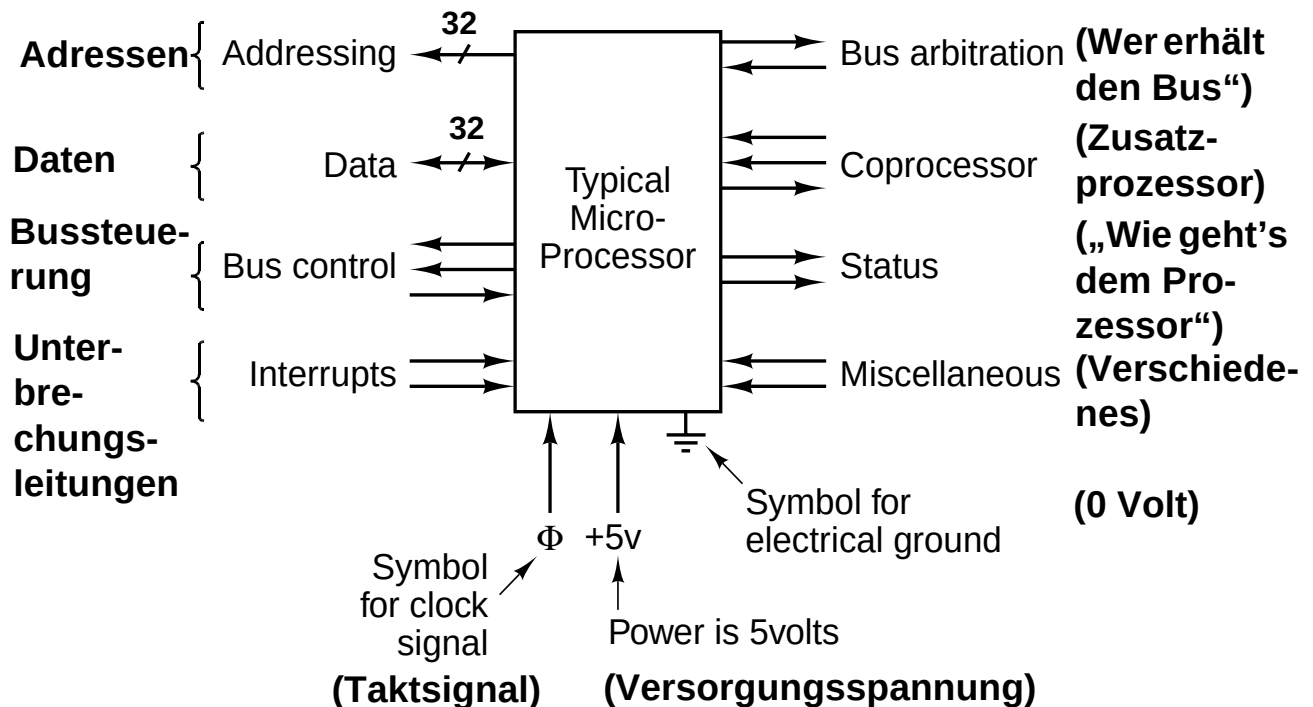
Peripherieeinheiten eines Prozessors

Busse in einem Rechner



- Sowohl innerhalb der Zentraleinheit (CPU) als auch im ganzen Rechner werden verschiedene Busse eingesetzt
- Diese dienen zur Verbindung von funktionalen Einheiten innerhalb der CPU bzw. zwischen CPU und den Peripheriegeräten bzw. Speicher

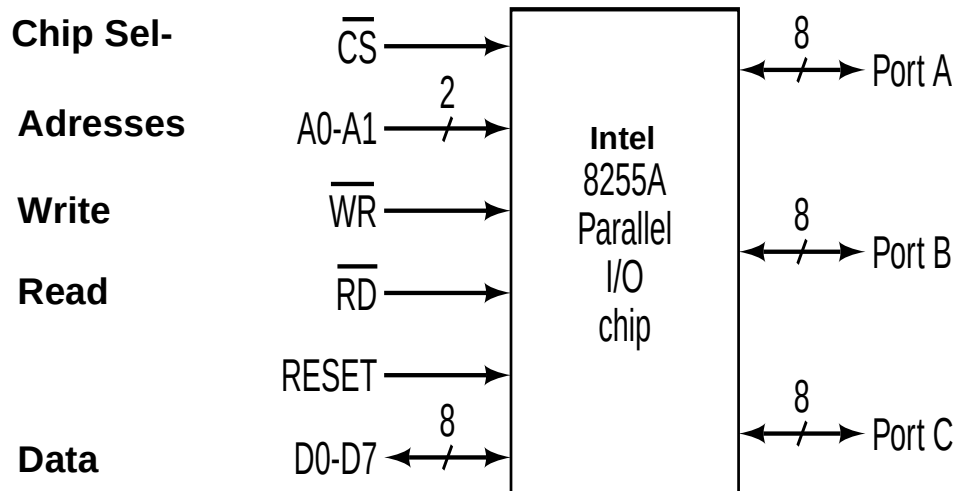
Beispiel eines Mikroprozessors „von außen“§ gesehen



- Pfeile geben die Richtung des Datenflusses an
- Doppelpfeile geben bidirektionale Leitungen an
- kleine Schrägstriche mit Zahlenangabe bedeuten Bus
- Die Zahl über dem Schrägstrich gibt die Zahl der Leitungen an, aus denen der Bus besteht

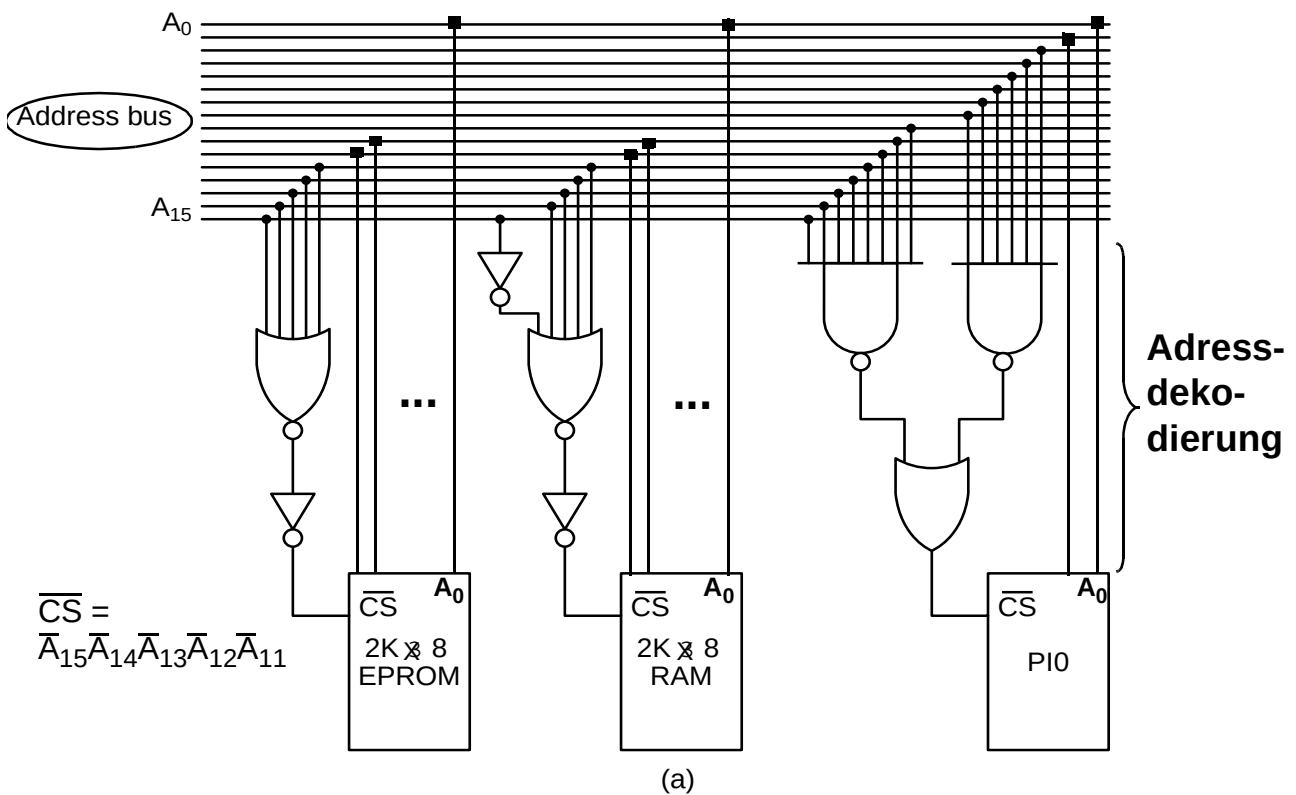
Peripheriebausteine

Beispiel: Baustein für parallele Ein-/Ausgabe (PIO)



- Baustein besteht aus drei Bussen zu je 8 Bit, die für Ein-/ und Ausgabe verwendet werden können (PortA-C)
- Die eingelesenen bzw. auszugebenden Daten sind am Datenbus D0-D7 zugänglich bzw. müssen dort angelegt werden
- Die Bausteinaktivierung erfolgt mit Chip Select $\overline{CS}$
- interne Register werden mit A0-A1 ausgewählt
- interne Register werden mit $\overline{WR}$ geschrieben und mit $\overline{RD}$ gelesen
- Ein vollständiges Rücksetzen aller interner Speicher der PIO erfolgt mit RESET
- Die PIO ist auf einem einzigen Chip integriert

Adressierung durch Chip-Select bei Peripherie- und Hauptspeicher-Chips



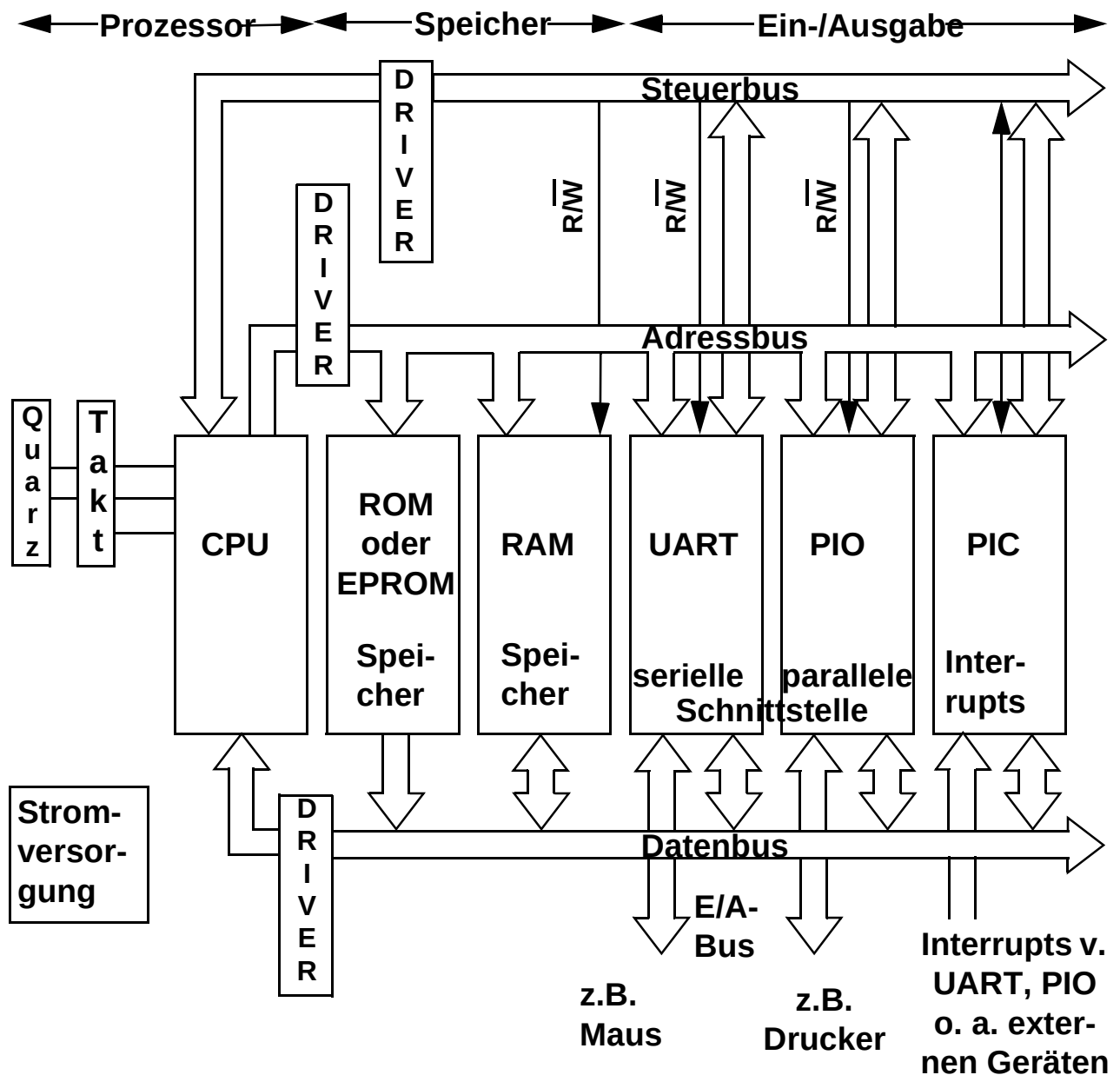
- EPROM = Erasable Programmable Read Only Memory = Festwertspeicher
- EPROMs speichern Daten auch ohne Strom dauerhaft
- ein EPROM ist erneut beschreibbar, nachdem es per UV-Licht oder Ansteuerimpulse gelöscht wurde
- der Löschvorgang ist langsam (einige ms)
- RAM = Random Access Memory = Speicherbaustein, der ohne vorheriges Löschen gelesen und beschrieben werden kann. RAMs benötigen Strom zur Datenspeicherung.

PIO

(Parallel I/O, Datenein-/ausgabebaustein)

- PIOs verfügen über 8, 16 oder 32 parallele Ein-/ausgabeleitungen, die bidirektional sind
- die Auswahl, welcher PIO-Baustein oder welcher Speicherbaustein gemeint ist, erfolgt in mehreren Phasen:
 - 1.) statische Festlegung, für welche Kombination aus höherwertigen Bits (MSBs) auf dem Adreßbus der betreffende integrierte Schaltkreis aktiviert werden soll
 - 2.) permanente Beobachtung des Adreßbusses durch einen Adreßdekoder, ob die zuvor festgelegte Adresse auf dem Bus anliegt
 - 3.) Aktivierung des Ausgangs des Adreßdekoders, sobald die gewünschte Bitkombination angelegt wurde
 - 4.) Aktivierung der Chip Select-Leitung des integrierten Schaltkreises durch den Adreßdekoder
 - 5.) Verwendung der niederwertigen Adreßbits (LSBs) zur Adressierung innerhalb des Bausteins
 - 6.) Lesen von oder Schreiben an der so ausgewählten Stelle des aktivierten Bausteins
- Jeder Peripheriebaustein hat ein ganzes Bündel von Kombinationen höherwertiger Bit zu seiner Aktivierung
- Das Bündel besteht aus zusammenhängenden Adressen, dem sog. „Adreßraum“

Standard-Mikroprozessorsystem



UART = Universal Asynchronous Receiver & Transmitter; für serielle Schnittstellen wie z.B. V24.

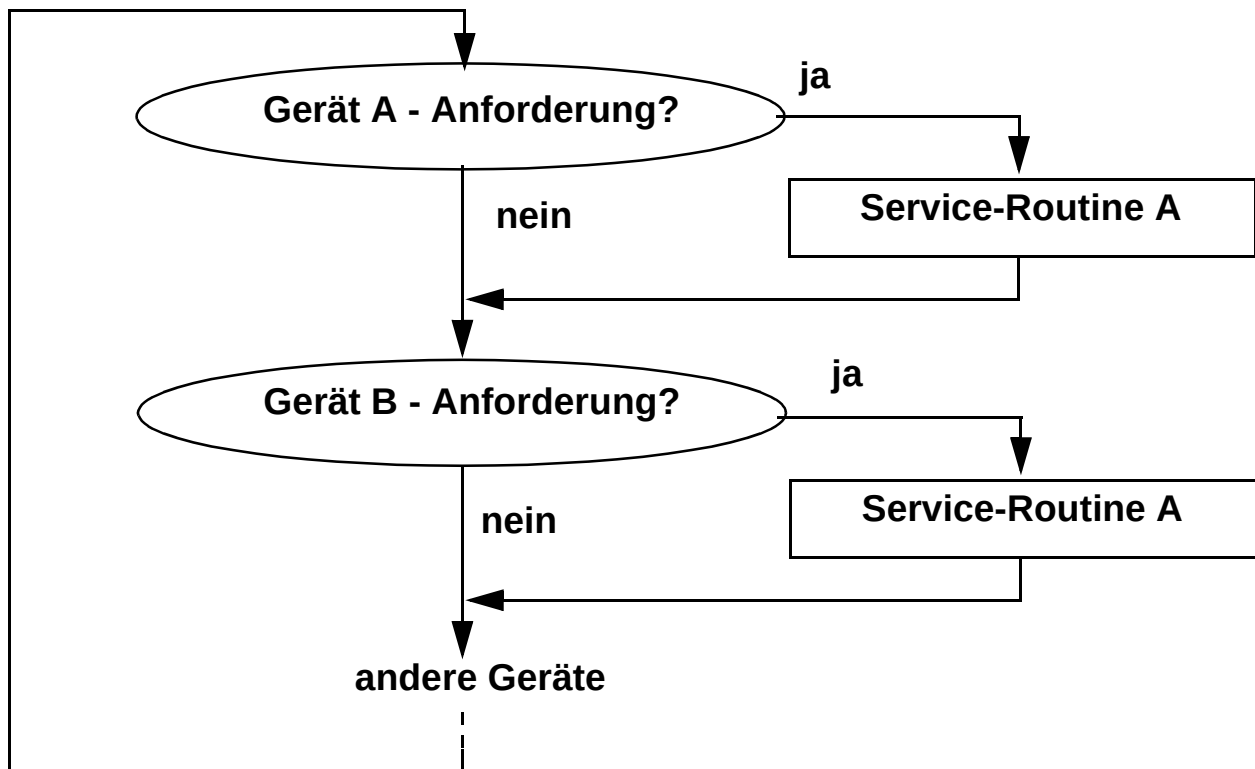
PIC = Programmable Interrupt Controller

PIO = Parallel I/O; für parallele Schnittstellen wie z.B. Centronics

Polling

- Unter Polling versteht man eine zyklische programmierte (softwaretechnische) Abfrage von einem oder mehreren E/A-Geräten zur Feststellung einer Kommunikationsbereitschaft von Kommunikationswünschen.
- Typisch wird dazu ein die Kommunikationsbereitschaft anzeigendes Flag in einem E/A-Steuerregister eines E/A-Gerätes zyklisch per READ-Befehl gelesen und ausgewertet.
- Wird eine Kommunikationsanforderung erkannt, wird auf der Basis von READ-und WRITE-Befehlen ein geeignetes Protokoll zum Datentransfer (bzw. zum Service) abgewickelt.

Software-Schleife



Polling: Vorteile - Nachteile

Vorteile:

- Polling erfordert minimalen Hardware-Aufwand.
- Feststellung und Bedienung einer Kommunikationsanforderung erfolgen *synchron* zum Programmlauf, d. h. sind explizit vorhersehbar - im Gegensatz zu asynchronen Interrupts.

Nachteile:

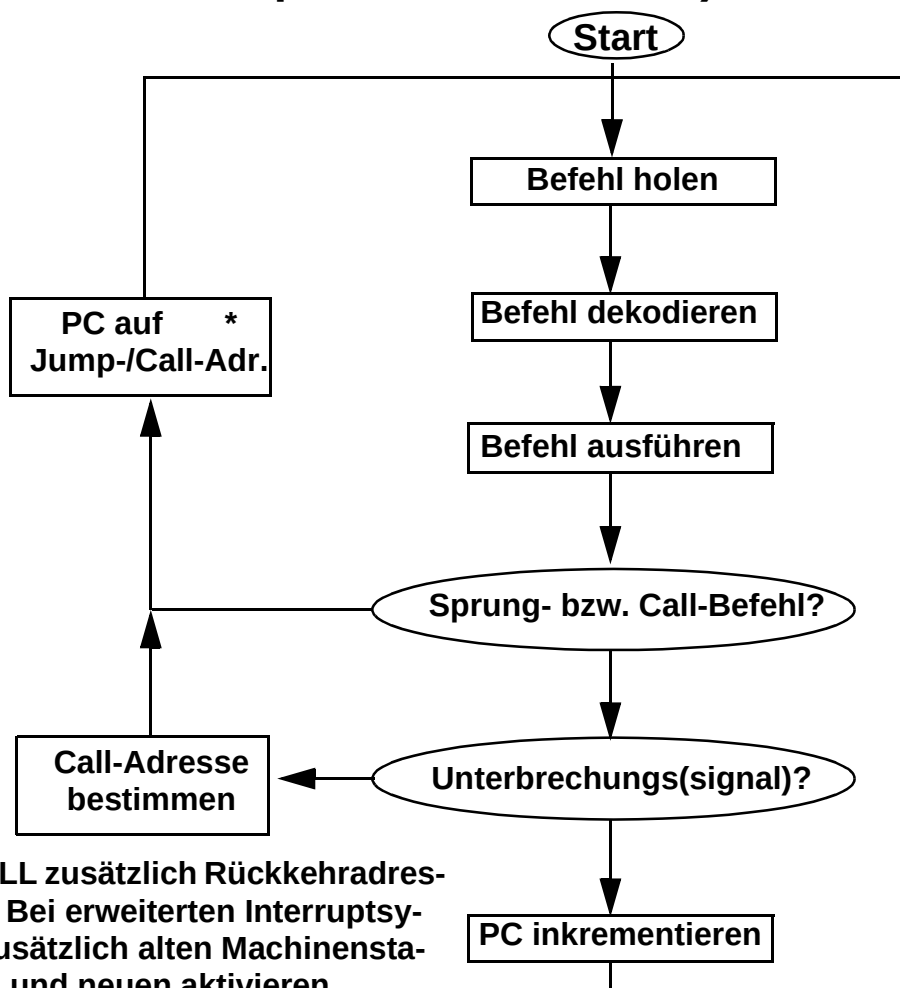
- Polling erzeugt einen hohen Programm-Overhead.
- Die meisten Anfragen erweisen sich als unnötig (Zeitverlust)
- Mit zunehmender Gerätezahl steigt die Reaktionszeit.
- Priorisierungen bei gleichzeitigen Anfragen erfordern weitere Zeit.

Folgerung:

- Die asynchrone Kommunikation mit externen Geräten sollte durch Hardware unterstützt werden $\Rightarrow$ *Interrupts*.

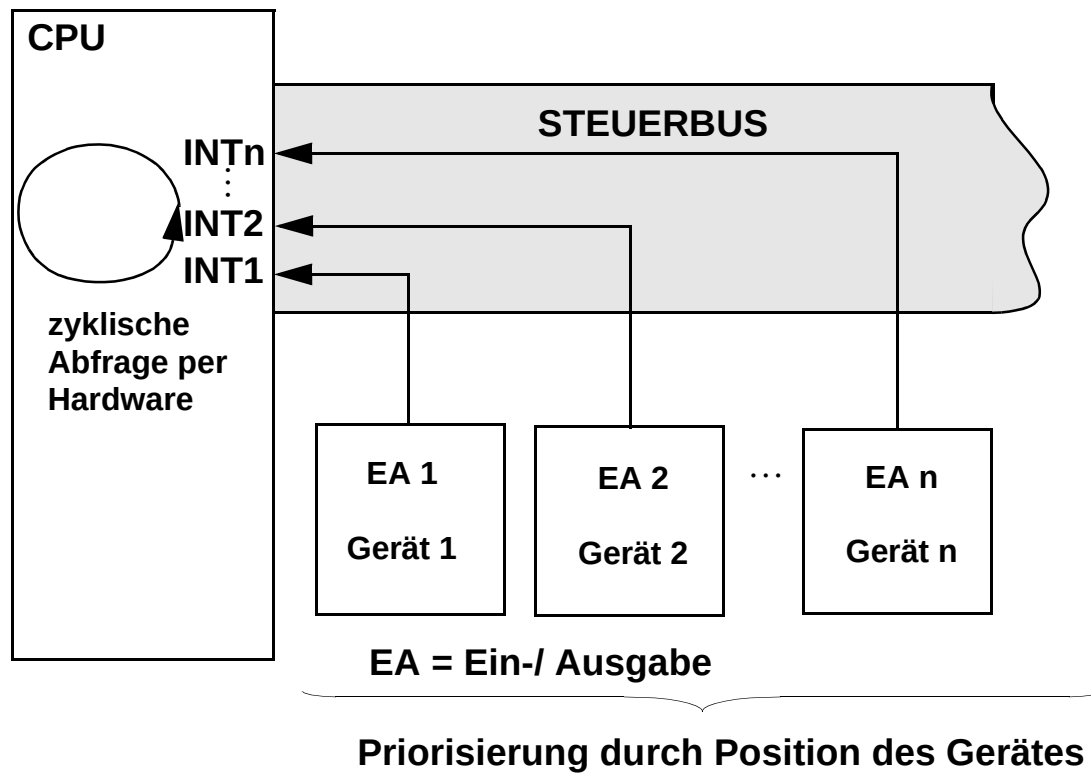
Interrupt-Prinzip

- Das Feststellen einer externen Kommunikationsanforderung kann auch durch eine Art „hardware-gestütztes Polling“ von sogenannten Interrupt-Signalleitungen (INT1, INT2, ...) erfolgen.
- Der Hardware-Steuermechanismus, der die „zentrale Befehlsschleife“ in der CPU organisiert, wird dazu um eine Unterbrechungsabfrage erweitert.
- Eine Kommunikationsanforderung wird typisch in eine Programmverzweigung vom Typ CALL „umgeleitet“. (Ruf einer Interrupt-Service-Routine)



(*) Bei CALL zusätzlich Rückkehradresse retten. Bei erweiterten Interruptsystemen zusätzlich alten Maschinenstatus retten und neuen aktivieren

Interruptsystem - Prinzipielle Struktur



Hinweis: Adress- und Datenbus sind nicht dargestellt.

Interrupt-CALL-Adresse Zuordnung

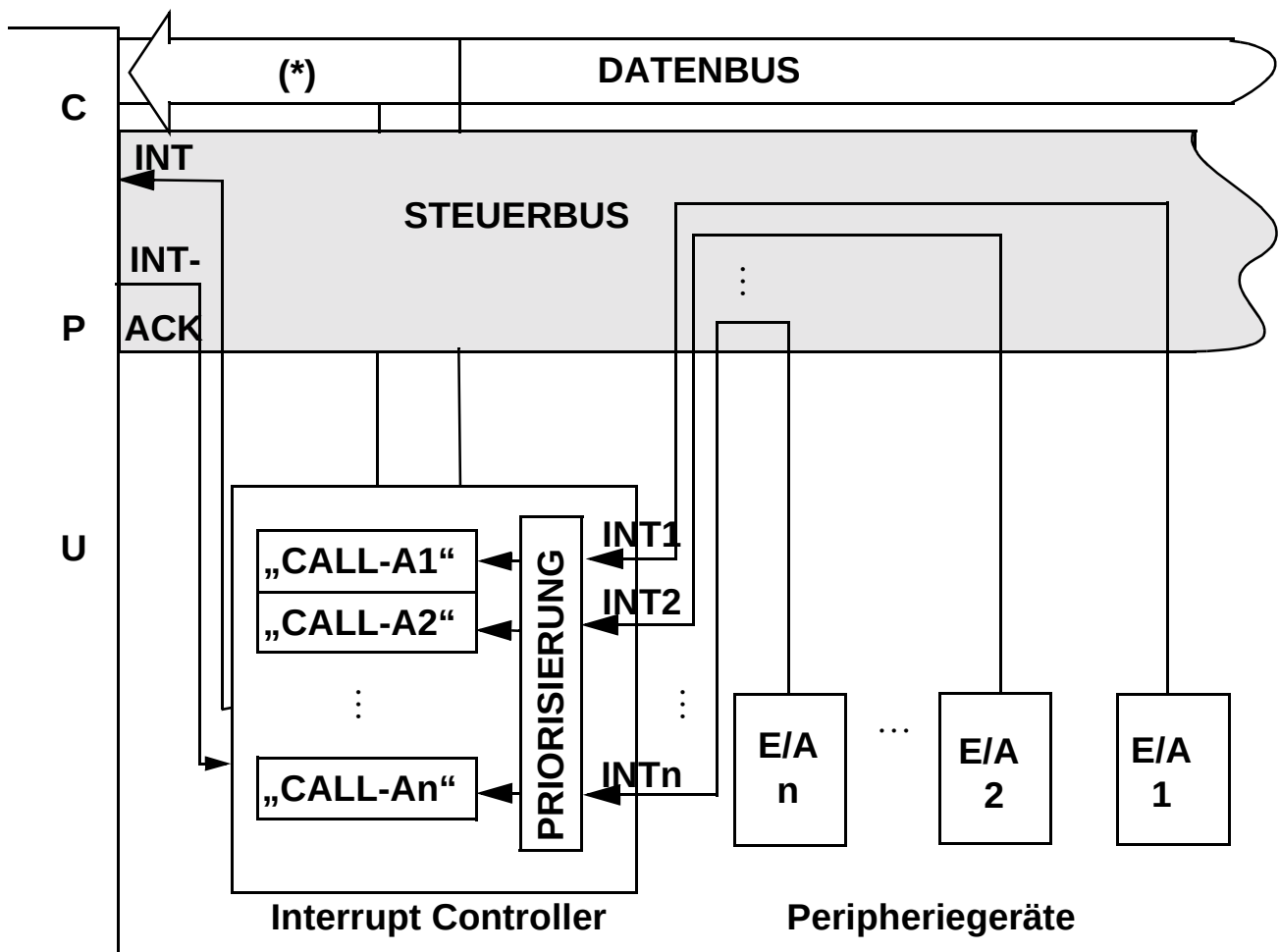
- Es gibt 2 grundsätzliche Zuordnungsverfahren

Variante A Einem bestimmten Signal wird durch die CPU eine feste Adresse zugeordnet: der sogenannte NICHTVEKTORISIERTE INTERRUPT

Variante B Einem Signal wird dynamisch eine wahlfreie Adresse zugeordnet, welche von der CPU automatisch nach einem fest vereinbarten Protokoll vom Datenbus gelesen wird: der sogenannte VEKTORISIERTE INTERRUPT

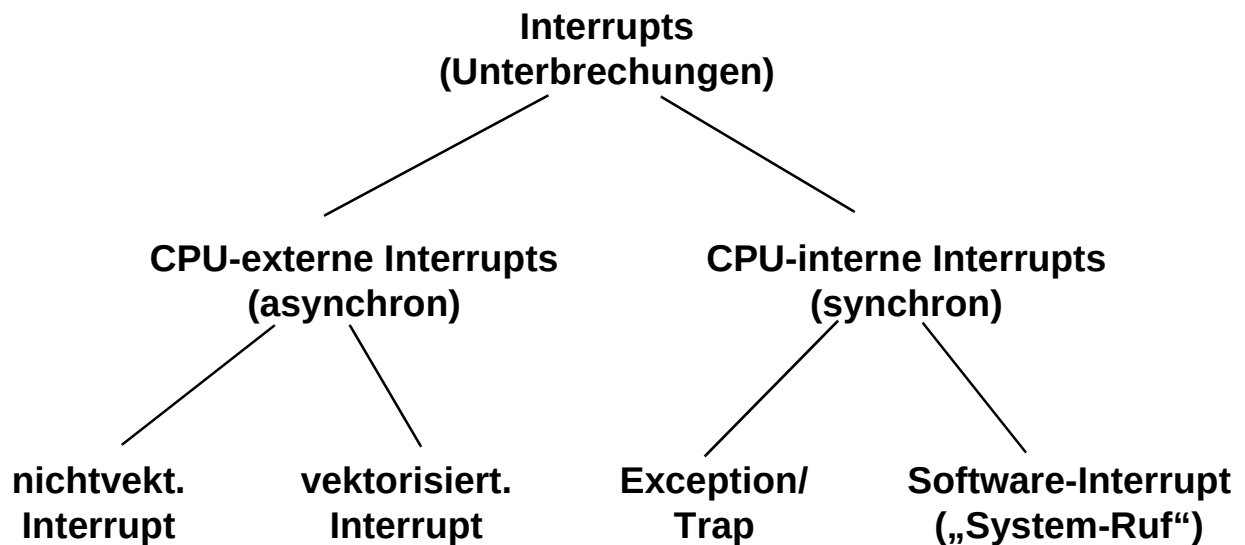
Vektorisierter Interrupt

- In Reaktion auf ein Interruptsignal wird die Verzweigungsadresse auf dem Datenbus erwartet bzw. von dort gelesen, was vermittelt eines Interrupt-Akknahmebestätigungs-Signals (INTACK) angezeigt wird
- Eine typisch verwendete Hardware ist die „Zentrale Prioritäts- und Adresszuweisungslogik“ (= Interrupt Controller)



(*) „CALL-Adresse“ ist bei entwickelten Interruptsystemen nur ein *Indexwert*, der als solcher benutzt wird, um aus einer CPU-internen (Interrupt) Tabelle letztlich die eigentliche Zieladresse zu bestimmen.

Erweitertes Interruptsystem



- **Exceptions** bzw. **Traps** sind typische Reaktionen auf interne Fehlerzustände, wie z.B. FPU-Error, Page-Fault usw. (FPU = Floating Point Unit = Gleitkommarechenwerk)
- **Software-Interrupts** sind bestimmte Befehle, die quasi ein internes Unterbrechungssignal auslösen, und in denen die „CALL-Adresse“ - bzw. der Interrupttabellen-Index, der auf diese Adresse zeigt - im Befehl selbst enthalten ist
- Der (Software)Interrupt-Mechanismus wird häufig mit einer **Maschinenstatus-Umschaltung** kombiniert, z.B. Umschalten auf System-Privilegien (= kernel mode bei UNIX)

Vor- und Nachteile von Interrupts

- Interruptgesteuerte E/A befreit die CPU vom Warten auf jedes E/A-Ereignis, jedoch benötigen vektorisierte Interruptprotokolle viele Taktzyklen
- Der Overhead des Protokolls sowie des Kontext-Retens innerhalb der Interrupt-Service-Routine wirkt sich besonders bei der Übertragung kleiner Datenmengen (einzelne Bytes) relativ stark aus
- Hinzu kommt, dass Interrupts typisch erst am Ende eines Befehlszyklus' anerkannt werden, was die Interrupt-Reaktionszeit erhöht (Problem Echtzeitfähigkeit)
- Bei RISC-Prozessoren bringen Interrupts aufgrund von Pipeline-Neustarts allgemeine Geschwindigkeitsverluste mit sich

Direct Memory Access (DMA)

- DMA dient zur Entlastung der CPU
- DMA wird über eine eigene Hardware, den DMA-Controller, abgewickelt
- Das Prinzip ist, dass die CPU den DMA-Controller so instruiert, dass dieser selbstständig Daten von der Peripherie in den Hauptspeicher transferieren kann und umgekehrt
- Dazu müssen von der CPU verschiedene Register in dem DMA-Controller gesetzt werden, bevor der DMA startet (Initialisierung):
 - Woher die Daten kommen (Adresse des Datenblocks im Hauptspeicher bzw. Adresse des Peripheriegeräts)
 - Wohin die Daten gehen sollen
 - Wie viele Daten zu übertragen sind
 - Interruptvektor

Eigenschaften von DMA

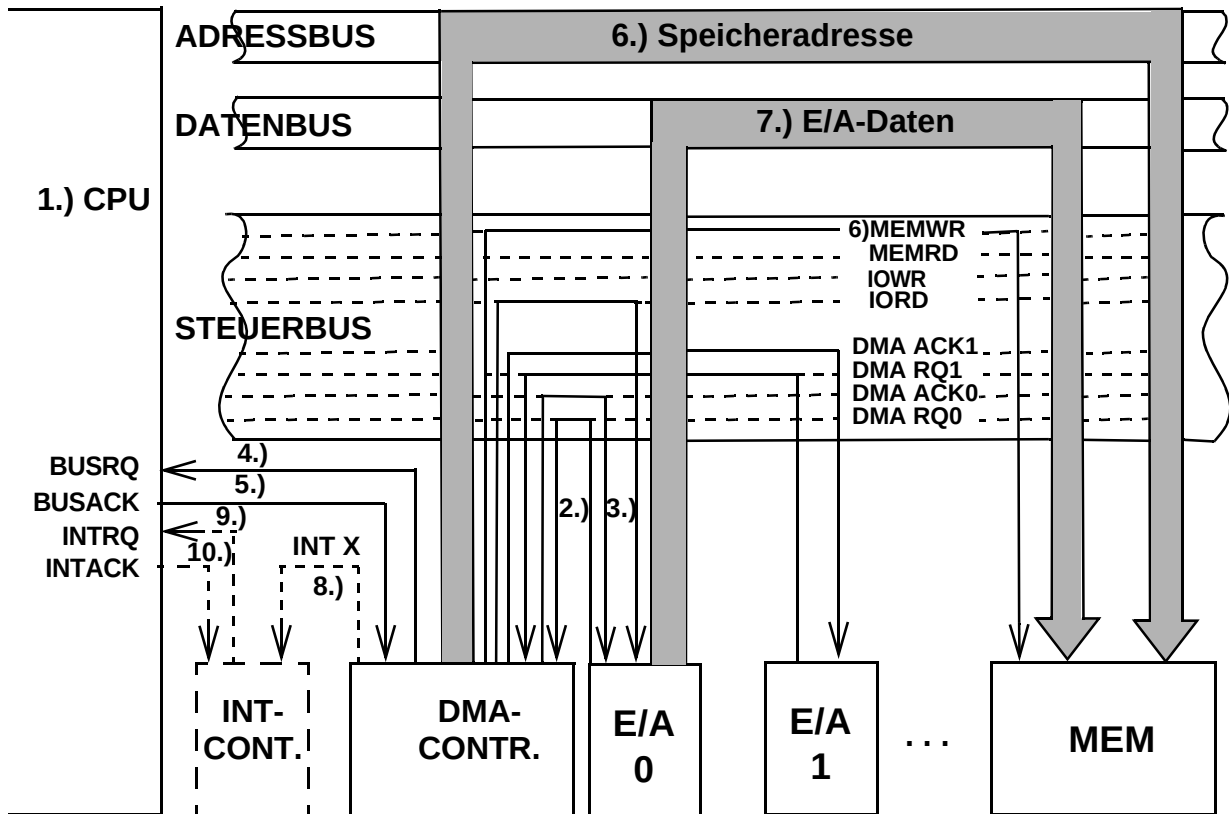
- Ein „direkter Speicherzugriff“ eines Peripheriegerätes erspart die Datenzwischenspeicherung in der CPU und entlastet diese
- DMA-Transfers erfordern zwar die Busfreigabe durch die CPU, jedoch kann diese währenddessen mit CPU-internen Daten weiterarbeiten (Parallelität)
- DMA-Transfers werden zweckmäßig bei *häufig wiederkehrenden, Echtzeit-erfordernden* (Block-Übertragungen) angewandt, z. B. Transfer von Plattenblöcken, Multimedia-Daten, Mess- und Steuerdaten usw.
- Häufig arbeitet der DMA-Controller mit einem Interrupt Controller zusammen, der wiederum bei der CPU einen Interrupt auslöst, sobald der DMA fertig ist

Ablauf eines DMA-Transfers

Beispiel: Peripheriegerät schreibt Daten direkt in den Hauptspeicher (ohne CPU als Zwischenglied)

- 1.) CPU initialisiert DMA-Controller und Interrupt Controller mit Herkunft- und Zieladresse der Daten, ihrer Länge und dem Interrupt-Vektor
- 2.) Peripherie fordert einen DMA-Transfer an (DMA Request)
- 3.) DMA-Controller bestätigt den DMA-Transfer (DMA Acknowledge)
- 4.) DMA-Controller fordert bei der CPU den Systembus an (BUS-REQUEST)
- 5.) CPU bestätigt die Busanforderung (BUS ACKNOWLEDGE)
- 6.) DMA-Controller gibt die passende Hauptspeicheradresse aus und aktiviert die Speicherschreibleitung (MEMORY WRITE)
- 7.) Peripheriegerät transferiert ein Datenwort in den Hauptspeicher. Die Hauptspeicheradresse wird automatisch vom DMA-Controller um 1 erhöht. Danach wiederholt sich Schritt 6 und 7 bis alles transferiert ist.
- 8.) Nach Ende des DMA-Transfers aktiviert der DMA-Controller einen Interrupt beim Interrupt Controller
- 9.) Der Interrupt Controller unterbricht die CPU (INT REQ)
- 10.) Die CPU geht in die Interrupt-Behandlungsroutine (INT ACK) und kann dort auf die Daten im Hauptspeicher zugreifen

DMA-Steuerereinheit (Controller)



Signalzusammenspiel beim Transfer E/A<->Speicher

Zwei Implementierungsvarianten von DMA

- A Eine E/A-Einheit besitzt selbst einen DMA-Controllerteil und kann somit Busmaster werden.
(Oftmals wird in ihr soviel „Intelligenz“ installiert - z. B. in Gestalt eines I/O-Prozessors - , dass man von einem „Intelligenten Kanal“ spricht.)

- B Eine *spezielle DMA-Steuereinheit (Blocktransferprozessor)* steht zentral zur Verfügung und kann von verschiedenen E/A-Einheiten genutzt werden. Vor ihrer Nutzung wird diese entsprechend den Bedürfnissen des zu nutzenden Gerätes (von der CPU) *programmiert*.

Memory-Mapped I/O

- Ein E/A-Interface besteht typischerweise aus einigen (wenigen) Registern, z. B. Sende- und Empfangsdatenregister und Status- und Steuerregister.
- Für die Adressierung dieser Register gibt es zwei grundsätzliche Varianten:
 - *Memory-Mapped-I/O* bedeutet, dass Teile des normalen Speicher-Adressraumes I/O-Geräten zugeordnet werden.
 - Getrennter I/O-Adressraum wird in Verbindung mit speziellen *E/A-Befehlen* realisiert, bei deren Ausführung von der CPU eine spezielle Leitung aktiviert wird, mit der auf dem *Bus* eine *E/A-Adresse* angezeigt wird.

Mem.-Mapped vs. Getr. Adressraum

- Getrennter I/O-Adressraum

- Historische Variante aus Zeiten, bei der Adressraum knapp war und Speicher- im Gegensatz zu E/A-Zugriffen rein synchron, d. h. ohne Handshake abliefen
- Typ. Vertreter: Intel 80 x 86 Familie
- Vorteile:
 - * Der Speicheradressraum wird nicht reduziert oder zerstückelt
 - * Der E/A-Adressraum kann kleiner als der Speicheradressraum sein, weshalb Dekodieraufwand entfällt
 - * E/A-Befehle können mit Supervisor-Privileg ausgestattet werden
 - * 2 getrennte Adressräume gestatten einfache „zentrale DMA-Steuerlösungen“
- Nachteile:
 - * Notwendigkeit spezieller E/A-Lese/Schreibbefehle

- Memory-Mapped I/O

- Von den meisten modernen RISC-Prozessoren, wie MIPS oder SPARC genutzte Variante
 - * Adressraum ist ausreichend vorhanden
 - * Aufgrund der „heutigen“ Speichertechnologie ist die Möglichkeit einer asynchronen Speicherzugriffssteuerung ohnehin erforderlich bzw. gegeben
- Vorteile: einfach
- Nachteile: keine

RISC-Prozessoren

**RISC= Reduced
Instruction
Set
Computer**

Reduced Instruction Set Computer

- in 80-er Jahren:
statt anwendungsnahe, komplexe Befehlssätze (CISC)
jetzt maschinennahe, einfache Befehlssätze (RISC)
 - Grundlage der Veränderungen:
 - 1.) Speicher war nicht mehr knapp und langsam
 - Befehlskodelesen ging schnell
 - Komplexe Befehle (viele Operationen mit *wenig* Code) waren nicht mehr notwendig
 - 2.) mehr Logikgatter und Designtools verfügbar
 - Nachdenken über Formen der Leistungssteigerung
 - große Schaltnetze/werke leichter beherrschbar
 - 3.) fortgeschrittene Compiler (optimierend und schnell)
 - breiter Trend zur Hochsprachenprogrammierung
 - Compiler verhindert die Konfrontation des Anwenders mit einfachen Befehlssätzen, welche zu unübersichtlichem Kode führen
 - 4.) verstärkte Nutzung von „Pipeline“-Techniken
 - Pipeline-Effizienz erfordert für die Befehle Einfachheit, Kodelängengleichheit, Bearbeitungszeitgleichheit
- => Ergebnis: wesentliche Steigerung der Rechenleistung

Pipelining = Fließbandverarbeitung

Prozessorleistung

- Die Prozessorleistung ist umgekehrt proportional zur Ausführungszeit eines bestimmten Programmes (Algorithmus, Benchmark, Task)
- Die Ausführungszeit kann als Produkt von 3 Faktoren beschrieben werden:

$$\text{Ausführungszeit} = C \cdot T \cdot I$$

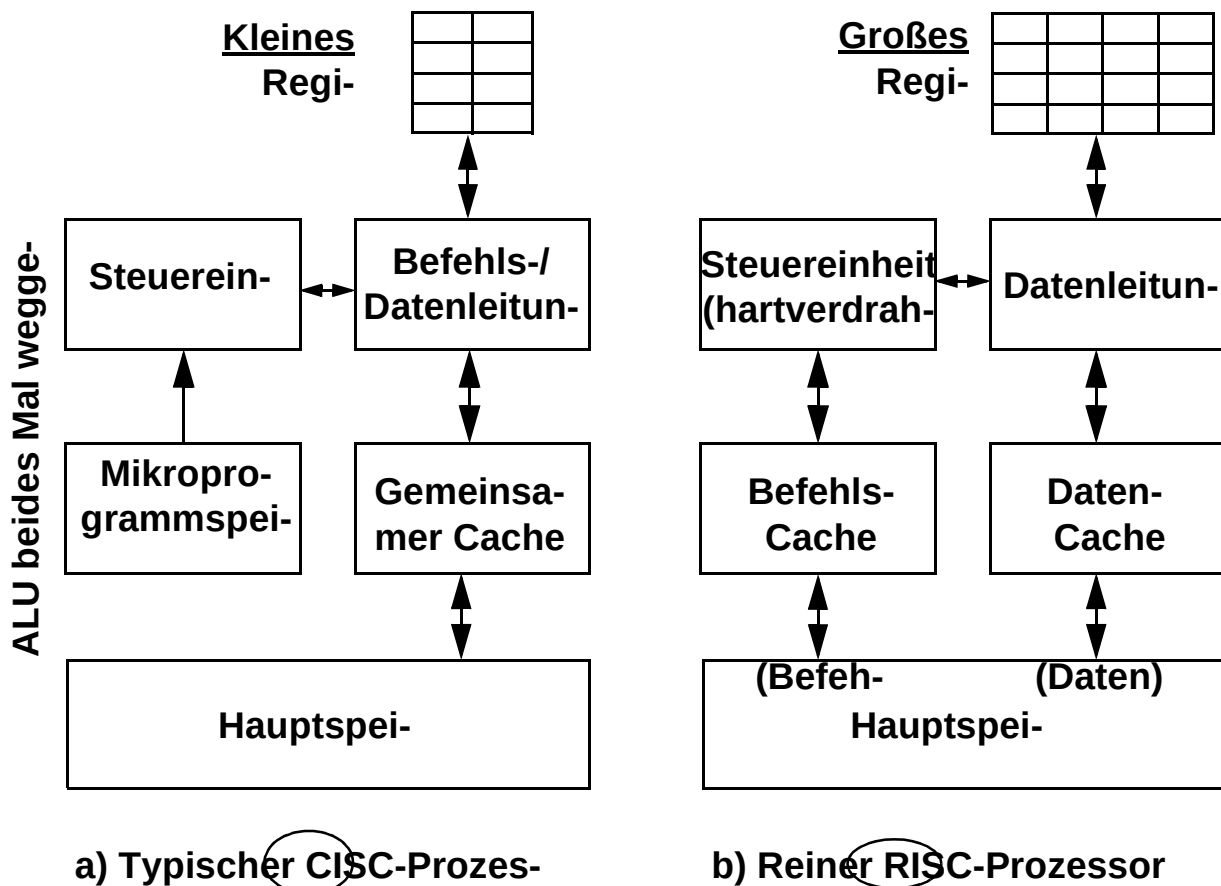
C = Cycles per Instruction (CPI-Wert)

T = Taktzeit = $1 / \text{Taktfrequenz}$

I = Instructions per Task

- C wird wesentlich durch die Architektur bestimmt,
 T durch die Technologie
und
 I durch den Compiler
- RISC-Architekturen versuchen, C und T zu minimieren, was meist eine Erhöhung von I mit sich bringt
- Optimierende Compiler und andere Techniken versuchen, den Einfluss der I -Erhöhung abzuschwächen
- Unter dem Strich lohnen sich RISC-Architekturen hinsichtlich der Ausführungszeit

Architektonische Unterschiede RISC-CISC



Cache = „Depot“

kleiner Speicher, der die Inhalte der am häufigsten benutzten Speicheradressen enthält (als Kopie)

Vorteil des Cache: wesentlich schneller als der Hauptspeicher, idealerweise so schnell wie die CPU

Vergleich RISC vs. CISC

- CISC

- variabler und großer Befehlssatz ($\gg 200$ Befehle)
- vielfältige Adressierungsarten; insbesondere indirekte Adressierungen (> 10)
- meist mikroprogrammiert
- mehrere Zyklen pro Befehl ($CPI > 1$) notwendig
- komplexe Befehle zur Unterstützung von Hochsprachen
- kleine Registersätze (< 10)

- RISC

- kleiner Befehlssatz (< 150)
- weniger Befehlsformate (< 4)
- wenige, wichtige Adressierungsarten (< 6)
- alle Operationen werden nur mit Registern ausgeführt
- nur zwei Befehlstypen, Load und Store, referieren den Hauptspeicher
- optimierte, festverdrahtete Steuerlogik statt Mikroprogrammierung
- Ausführung aller oder der meisten Befehle in einem einzigen Taktzyklus ($CPI \rightarrow 1$) (bei Pipelining nur 1 Takt pro Pipelinestufe)
- große Registersätze (≥ 32)

CPI = Clock Cycles Per Instruction
= Zahl der Takte pro Befehl

Moderne RISC-Prozessoren

R 14000	SGI / MIPS
α -Prozessor 21364	HP
UltraSPARC I,II,III	SUN
PA-8400	HP
PowerPC	Motorola, IBM, Apple

Bemerkung:

- Die sog. „INTEL-IA-32-Architektur“ des Pentium, Pentium Pro, PII, PIII, PIV ist weder rein RISC noch rein CISC sondern eine hybride Lösung

Zusammenfassung

RISC — Design Prinzipien

- einfache Maschinenbefehle
 - einfache und wenige Adressierungsarten
 - einheitliches Befehlsformat, gleiche Länge und Ausführungszeit (= wichtig bei Pipelining)
 - Load / Store-Architektur (Trennung Datenadressierung und Datenmanipulation)
 - Einzyklus-Maschinenbefehle (CPI $\rightarrow$ 1) (bei Pipelining 1 Taktzeit pro Stufe)
- große, universelle Registersätze
 - schneller Zugriff auf (Register) Variable
 - optimierte Zuweisung und Behandlung von Registervariablen durch den Compiler
- Verzahnung Architektur und Compiler
 - (prozessortypisch) optimierende Compiler
- optimierte VLSI-Chipfläche
 - einfache, platzsparende Steuerwerke schaffen Fläche für größere Registersätze und weitere leistungsoptimierende Techniken, wie Pipelining, Register-Windows, Branch Prediction, Superskalar-Techniken

Pipelining	= Fließbandverarbeitung
Branch Prediction	= Vorhersage des Sprungziels
Superskalar	= mehr als ein Befehl pro Takt
Register-Windows	= Parameter für Unterprogramme werden über Register übergeben, nicht über Stack

Verfahren zur Beschleunigung der Befehlsausführung bei RISC-Rechnern

- Pipelining**
- Superskalarität**
- Dynamische Befehlsausführung**
- Spekulative Befehlsausführung**

Pipelining

- In der Hauptbefehlsschleife eines Prozessors werden für jeden Befehl immer wieder dieselben elementaren Schritte durchgeführt:
- Beispiel 1: 5 Schritte bei CISC-Rechnern
 - 1.) Befehl aus dem Speicher lesen (= Fetch Phase)
 - 2.) Befehl dekodieren (= Decode Phase)
 - 3.) Operanden aus dem Hauptspeicher holen (= Operand Phase)
 - 4.) Befehl ausführen (=Execute Phase)
 - 5.) Ergebnis Rückschreiben (=Write Back Phase)
- Beispiel 2: 4 Schritte bei RISC-Rechnern
 - 1.) Fetch
 - 2.) Decode
 - 3.) Execute
 - 4.) Write Back
- Aufgrund der Register-Architektur entfällt bei RISC-Rechnern der Hauptspeicherzugriff
- Für Hauptspeicherzugriffe muß ein extra Befehl ausgeführt werden, nicht eine extra Phase pro Befehl (=Load/Store-Architektur)
- Ein Spezialfall ist die Ausführung bedingter Sprünge

- Beispiel 3: bedingter Sprung bei CISC-Rechnern

- 1.) Fetch = Sprungbefehl holen**
- 2.) Decode = Sprungbefehl dekodieren**
- 3.) potentielle Zieladresse berechnen (= wohin wird vielleicht gesprungen?)**
- 4.) Execute = Sprungbedingung auswerten (=Wird gesprungen?)**
- 5.) Befehlszähler auf Nachfolgeadresse setzen**

- Beispiel 4: bedingter Sprung bei RISC-Rechnern

- 1.) Fetch = Sprungbefehl holen**
- 2.) Decode = Sprungbefehl dekodieren und potentielle Zieladresse berechnen**
- 3.) Execute = Sprungbedingung auswerten**
- 4.) Befehlszähler auf Nachfolgeadresse setzen**

- RISC-Rechner fassen beim Sprung i.d.R. Schritt 2 und 3 der CISC-Rechner zusammen, so dass nur ein Schritt nötig ist (Parallelverarbeitung)

- Die Idee ist jetzt, das immer gleiche Schema der Befehlshauptschleife im Fließbandverfahren (Pipelining) abzuarbeiten

Pipeline-Prinzip

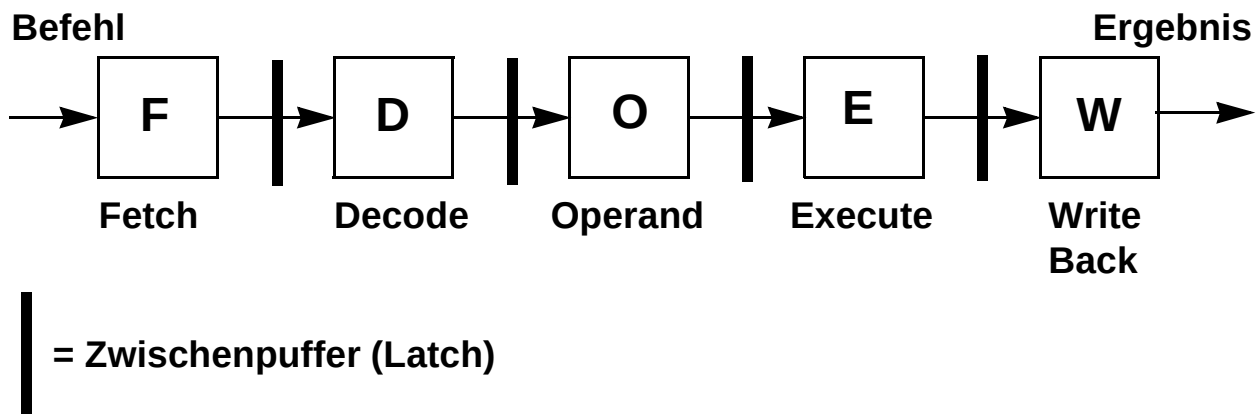
- Pipelining (Fließbandverarbeitung) ist eine grundlegende, typisch bei RISC-Prozessoren angewandte Implementierungstechnik, die Leistung eines Prozessors zu verbessern, indem mehrere Befehle zeitlich überlappt ausgeführt werden.

Mit und ohne Pipelining

- a) sequentieller Fall
 - Verarbeitung durch eine einzige, komplexe Logik



- b) Beispiel: 5-stufige Pipeline (CISC-Fall)
- Verarbeitung durch 5 in Reihe geschaltete Stufen
- um die Zwischenergebnisse einer Pipeline-Stufe aufzufangen und an die nächste Stufe weiterzureichen, ist ein Zwischenspeicher nötig
- alle Zwischenspeicher der Pipeline werden als Latch ausgeführt



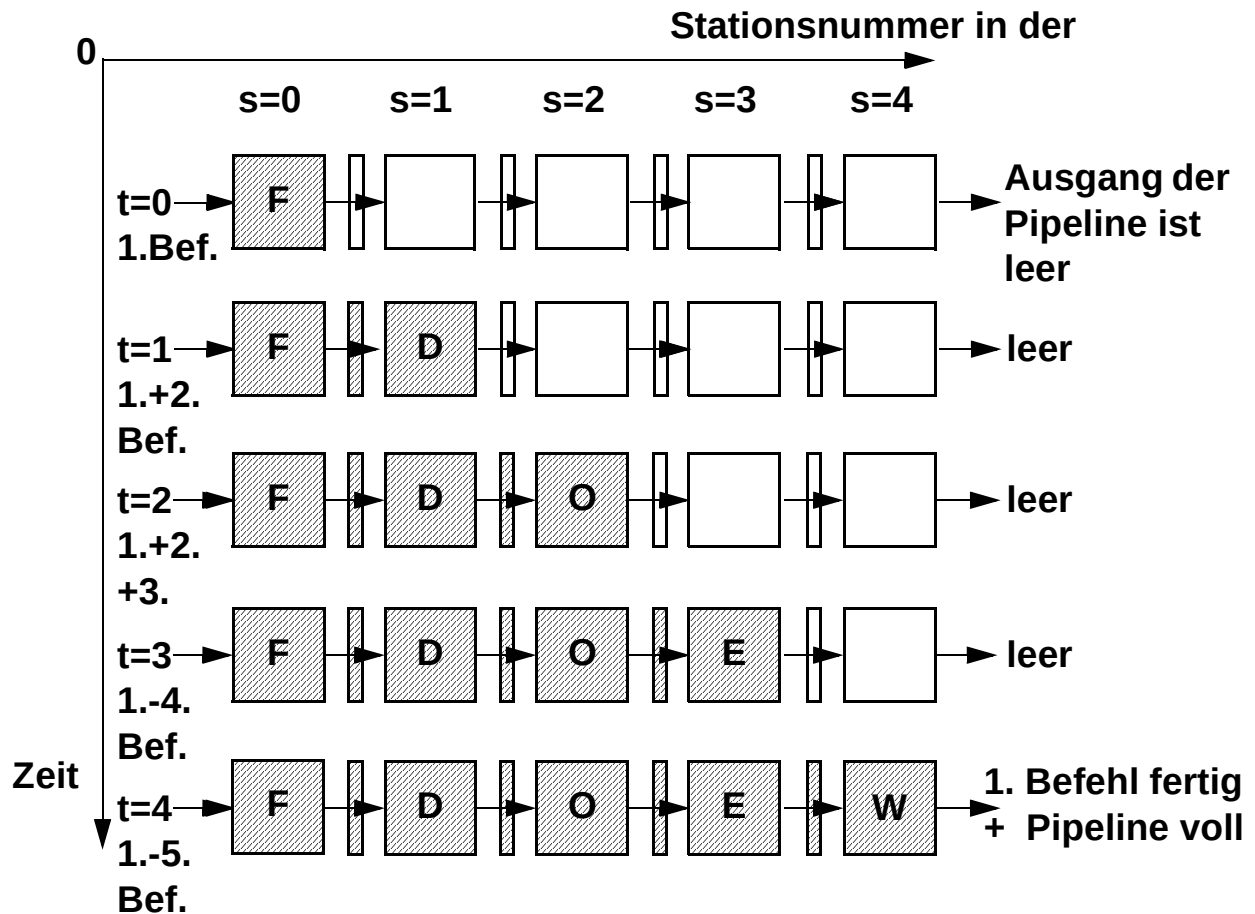
=> Bearbeitung wird zerlegt in kleinere Schritte (Divide et Impera)

- Jeder kleine Schritt kann schneller ausgeführt werden als ein großer
- Viele kleine Schritte überlappend ausgeführt kosten nicht mehr Zeit als ein kleiner Schritt
- Ein-/Ausspeichern in die Latches kostet jedoch zusätzliche Zeit (= Effizienzverlust)
- Wichtig ist, daß alle Teilphasen ungefähr gleich lang dauern

=> Die logischen Phasen einer Befehlsabarbeitung werden je nach den tatsächlichen benötigten Ausführungszeiten so zusammengefaßt, daß etwa gleich lange Teilphasen entstehen.

=> Die Teilphasen werden in physikalisch voneinander unabhängigen Teilwerken ausgeführt, die ihre Teilergebnisse synchronisiert über Latches weiterleiten.

Raum-Zeitdiagramm für den Durchlauf eines Befehls bei Pipelining

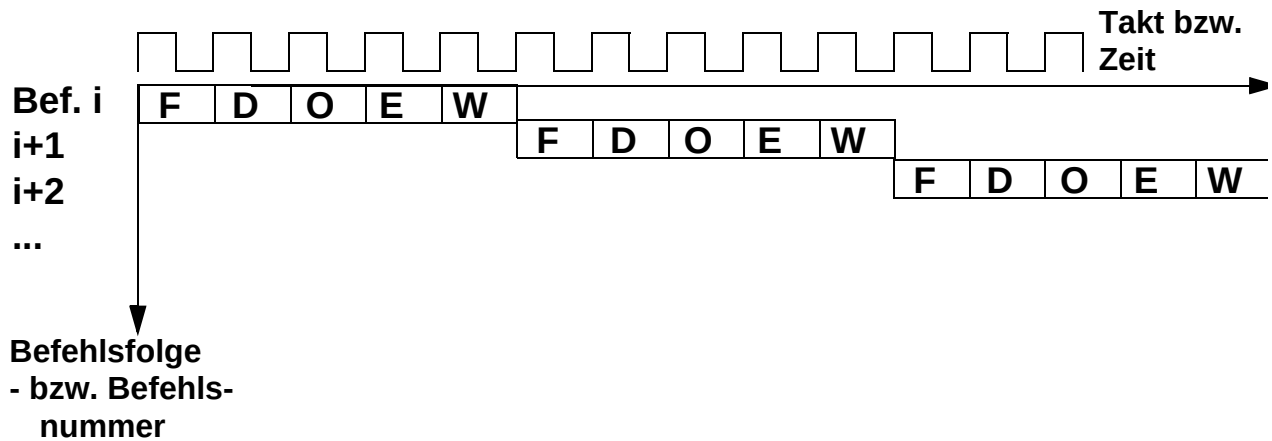


- Beginnt bei einem Befehl die 2. Phase, kann der nachfolgende bereits in die 1. Phase gehen (zeitl. Überlappung)
- Rückt mit Ende einer Phase stets ein neuer Befehl nach, “füllt“ sich die Pipeline schließlich

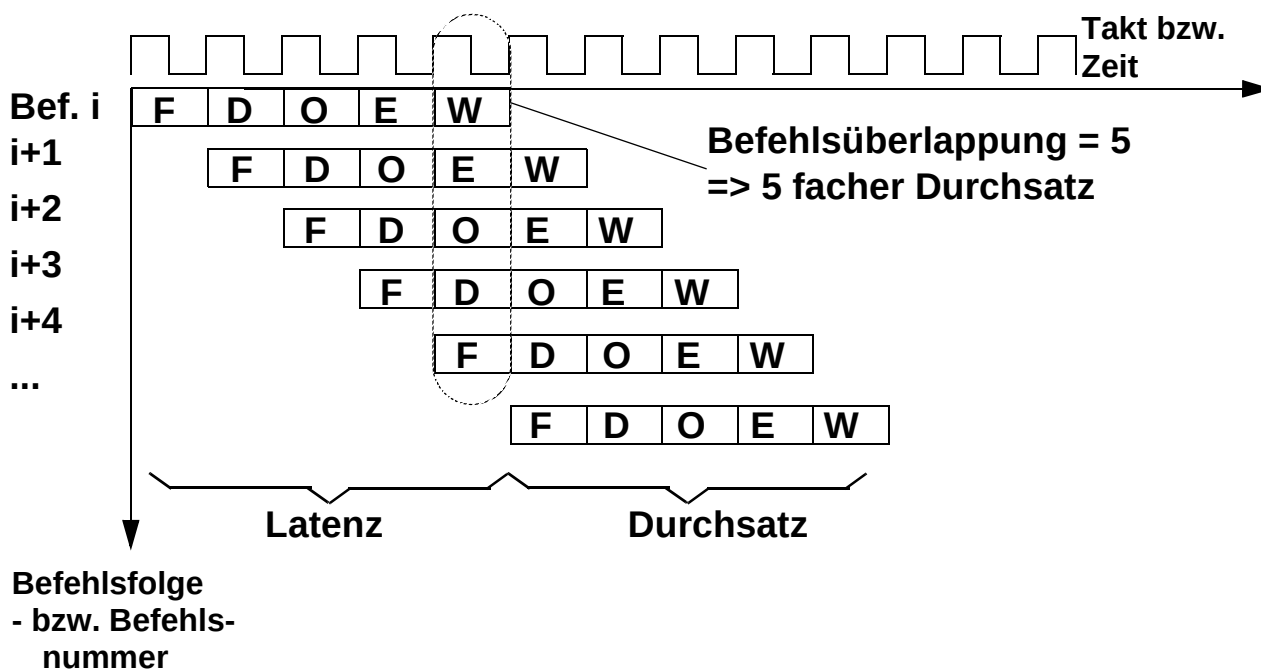
- Nach Abschluß der letzten, d. h. der 5. (n-ten) Teilphase, ist ein Befehl fertig
- Sein Nachfolger wird eine Teilphase später fertig
- Bei gefüllter Pipeline wird nach jedem Takt (=Zeit einer Phase) ein Befehl fertig
- Der Durchsatz beträgt dann 1 Befehl pro Takt
- Die Wirkung ist, als ob jeder Befehl nur einen Takt zur Ausführung benötigen würde
- Man spricht von Einzyklus-Maschinenbefehlen bzw. von einem "Cycles Per Instruction-Wert" (CPI) gleich 1

Vergleich ohne/mit Pipelining

- a) ohne Pipelining (Achtung: Abbildung ist in Befehl/Zeitdarstellung, nicht in Raum/Zeitdarstellung)



- b) mit Pipeline von 5 Stufen (in Befehl/Zeit-Darstellung)



=> Rechenleistung ist 5-fach verbessert

Maße bei einer Pipeline

- Stufenzahl k der Pipeline
- Taktzeit τ
- Latenz L = Zeit vom Eintritt eines Befehls bis zu seiner Fertigstellung (=Durchlaufzeit durch die Pipeline)
- $L = \tau k$, d.h. L ist proportional zu k
- Durchsatz D = Zahl der überlappend bearbeiteter Befehle
- $D = k$, sobald der eingeschwungene Zustand erreicht ist
- wichtig: Pipelining erhöht den Befehls-Durchsatz, reduziert jedoch nicht die Ausführungszeit eines Befehls

Speedup (Beschleunigung)

- Bei einem Rechner ohne Pipelining sei T die durchschnittliche Ausführungszeit eines Befehls
- Die Befehlsausführung wird durch eine k -stufige Pipeline in k Teilphasen zerlegt
- Jede Teilphase benötigt im Idealfall dieselbe Ausführungszeit T / k

- Dann beträgt die Beschleunigung von n Befehlen bei einer Pipeline:

$$\text{Beschleunigung} = \frac{\text{Ausführungszeit von } n \text{ Befehlen ohne Pipelining}}{\text{Ausführungszeit derselben Bef. mit Pipelining}}$$

- bzw.:

$$S = \frac{nT}{T + (n - 1) \cdot \frac{T}{k}} = \frac{nk}{k + n - 1}$$

- mit S = Beschleunigung (Speedup), n = Zahl der Befehle, T = Taktzeit ohne Pipelining und k = Anzahl der Pipelineinstufen

- Spezialfälle:

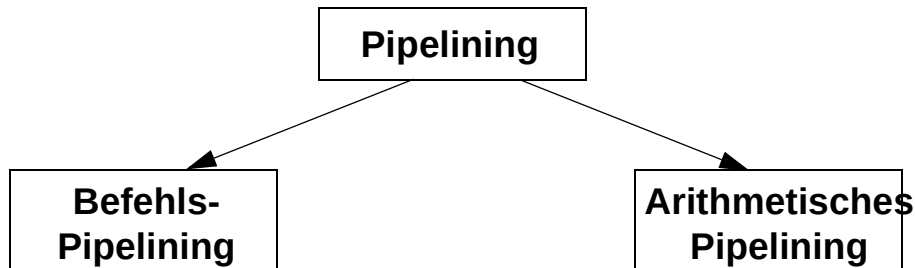
- $n = 1 \Rightarrow S = 1$

- $n \rightarrow \infty \Rightarrow S \rightarrow k$ (= eingeschwungener Zustand)

- Eine Pipeline mit k Stufen erreicht max. einen Speedup von k

Funktions- und Befehls-Pipelining

- Es gibt 2 Arten von Pipelining:



- Befehls-Pipelining:

- Die Abarbeitung einer Folge mehrerer Befehle wird in Pipeline-Form organisiert

- Arithmetisches Pipelining:

- Die Befehlsausführungsphase eines einzelnen Befehls wird in Pipeline-Form organisiert (Bsp.: Execute Phase bei Gleitkommadivision)

- Pipelining ist eine sehr alte Technologie

- Sie wurde bereits in frühen Mainframes eingesetzt (z.B. IBM 360/91)

Superpipelining

- Superpipelining wurde bei RISC-Rechnern erstmals verwendet
- Superpipelining heißt, dass die Hauptphasen der Pipeline im Leitwerk (=Befehls-Pipeline) in weitere Unterphasen unterteilt werden
- Beispiel 1:
 - Die Befehlsausführungsphase (Execute Phase) wird in Unterphasen aufgeteilt und als Pipeline organisiert
 - Diese Pipeline heißt arithmetische Pipeline und wird als Sub-Pipeline in die Haupt-Pipeline „eingehängt“
- Beispiel 2:
 - Die Rückschreibephase wird in Unterphasen aufgeteilt und als Pipeline organisiert
- Die Einbindung von arithmetischer Pipeline und/oder Rückschreibe-Pipeline in die Befehls-Pipeline führt zu Superpipelining

- Beispiel 3:
- Die Befehls-Pipeline des RISC-Prozessors R4000 wird in jeweils 2 Halbphasen unterteilt, so daß sich insgesamt 8 Stufen ergeben:

- 1.) Befehl holen 1. Teil
- 2.) Befehl holen 2. Teil
- 3.) Register-Operanden holen
- 4.) Befehl ausführen
- 5.) Daten-Cache-Zugriff 1. Teil
- 6.) Daten-Cache-Zugriff 2. Teil
- 7.) Tag-Prüfung
- 8.) Ergebnis rückschreiben

- Superpipelining heißt ca. 8 - 16 Pipeline-Stufen
- Heutige Prozessoren verwenden alle Superpipelining
- Beispiel: der Pentium II-Prozessor z. B. hat bis zu 14 Subphasen

Vorteil von Superpipelining

- höhere Taktrate möglich, weil jede Subphase kürzer als eine Hauptphase der Befehls-Pipeline ist

Nachteil von Superpipelining

- „Lange“ Pipelines sind schwerer ununterbrochen gefüllt zu halten als kurze, da sie schneller die Operanden/ Befehle abarbeiten („Nachschubmangel“)

Blasenfreie Pipeline

- Idealerweise sollten Befehls-/ und Arithmetik-Pipeline ununterbrochen gefüllt sein
- Dies wird als „blasenfreie“ Pipeline bezeichnet

Grenzen des Speedups einer Pipeline

- Die Gleichung für den Speedup gilt nur, wenn Befehls-/ und Arithmetik-Pipeline ständig mit neuen Befehlen und Operanden versorgt werden
- In der Praxis ist dies nicht der Fall, vielmehr kann es zu sog. Steuerfluß- und Datenflußkonflikten kommen

Datenflußkonflikte

- Ein Datenflußkonflikt tritt z. B. auf, wenn ein Ergebnis von einem früheren Befehl als Eingabe benötigt wird (Operanden-Phase), das Ergebnis aber noch nicht in das Zielregister zurückgeschrieben ist (Write back-Phase noch nicht abgeschlossen)
- => Datenflußkonflikte treten z.B. zwischen der Operanden-Hol-Stufe und der Rückschreibe-Stufe der Befehls-Pipeline auf

Steuerflußkonflikte

- Ein Steuerflußkonflikt tritt auf, wenn der Prozessor einen bedingten oder unbedingten Sprung ausführt
 - Hierbei wird zum Füllen der Befehls-Pipeline das Sprungziel bereits benötigt, bevor es berechnet ist!
 - Die Fetch-Stufe, die den jeweils nächsten Befehl holen soll, weiß nicht, wo dieser steht
- => Steuerflußkonflikte treten in der Fetch-Stufe der Befehls-Pipeline auf

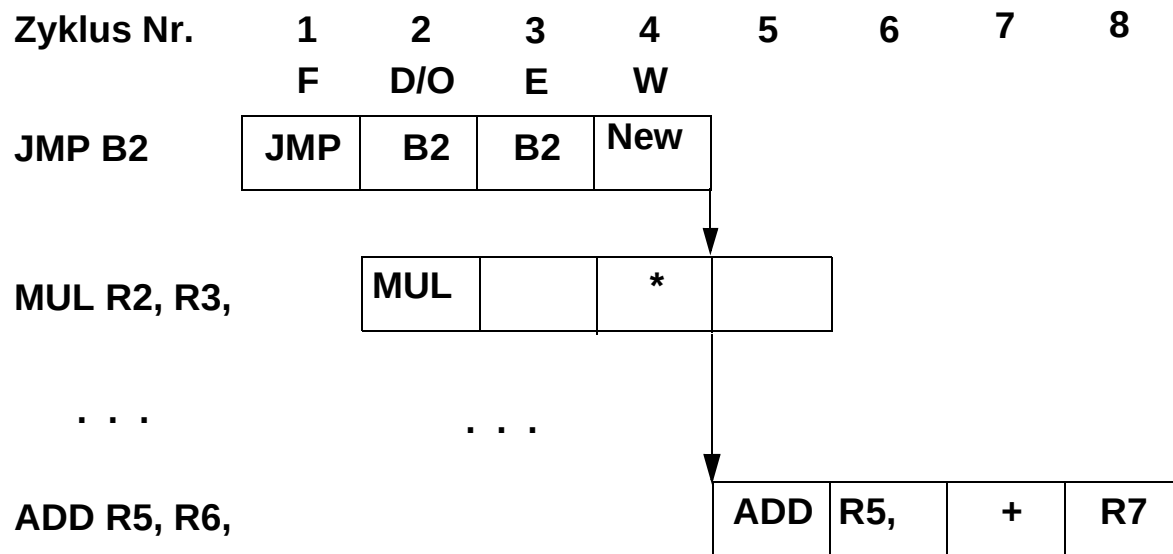
Steuerflußkonflikt bei *unbedingtem Sprung*

- Beispiel:

- B1: JMP B2 -- springe nach Befehl B2
- B1+1: MUL R4, R2, R3 -- R4 := R2 * R3
-
- B2: ADD R7, R5, R6 -- R7 := R5 + R6

- 1.) In der Fetch-Phase wird der JMP-Befehl gelesen
- 2.) In der D/O-Phase wird der JMP-Befehl dekodiert
- 3.) In der Ausführungsphase wird die Nachfolgeadresse („Branch Target“) berechnet
- 4.) In der Rückschreibephase wird der PC auf den neuen Wert gesetzt

=> PC steht erst am Ende der Phase 4 zur Verfügung



=> Write Back-Phase von MUL (Befehl B1+1) muß abgebrochen werden

=> Fetch- und Decode-Phase von Befehl B1+2 sowie Fetch-Phase von Befehl B1+3 müssen abgebrochen werden

=> Fetch-Phase von ADD (Befehl B2) wird um 3 Takte verzögert => Absinken der Rechenleistung

- Der Abbruch der Ausführung der Befehls-Pipeline wird als *Pipeline-Invalidierung* bezeichnet
- Nach einer Invalidierung muß die Pipeline neu in Gang gesetzt werden
- Der Nachfolgebefehl muß von der richtigen Adresse gelesen werden

Steuerflußkonflikt bei bedingtem Sprung

- Beispiel: bedingter Sprung wird durchgeführt

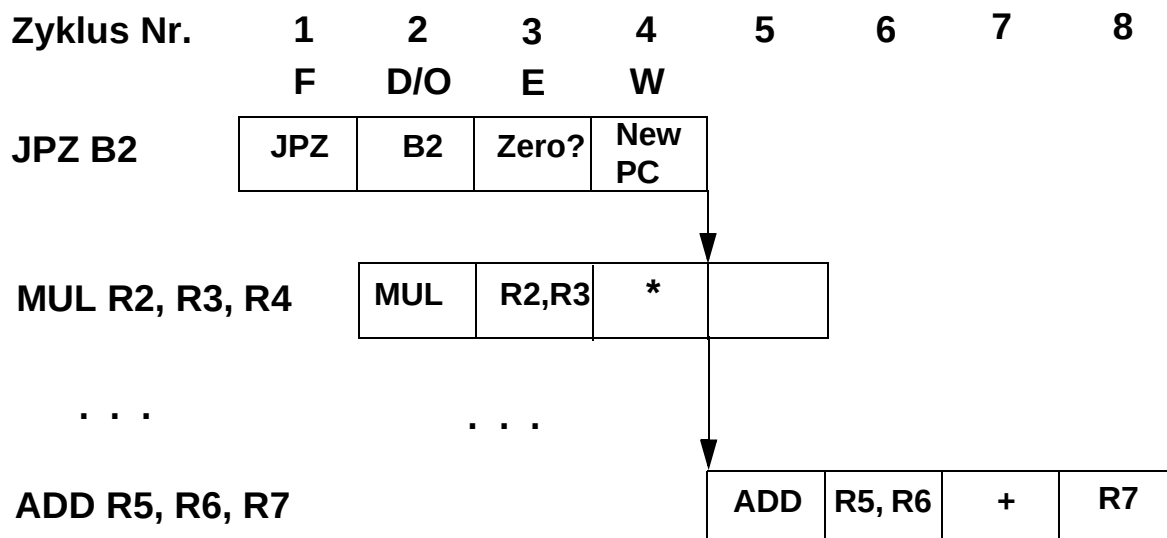
- B1: JPZ B2 -- Jump on Zero Flag
- B1+1: MUL R4, R2, R3 -- R4 := R2 + R3
-
- B2: ADD R7, R5, R6 -- R7 := R5 + R6

1.) In der Fetch-Phase wird der JPZ-Befehl gelesen

2.) In der Dekodier-/Operanden-Phase wird der JPZ-Befehl dekodiert

3.) In der Ausführungsphase wird die Sprungbedingung ausgewertet und dann die Nachfolgeadresse berechnet

4.) In der Rückschreibephase wird der PC auf den neuen Wert gesetzt



=> PC steht erst am Ende der Phase 4 zur Verfügung

=> Rückschreibphase des MUL-Befehls muß abgebrochen werden, sowie Fetch- und Decode-Phasen von B1+2 bzw. B1+3

=> Fetch-Phase des ADD-Befehls wird wie beim unbedingten Sprung um 3 Takte verzögert

- Wenn ein bedingter Sprungbefehl ausgeführt wird (= “branch taken“), steht der nachfolgende Befehl an der Stelle, die angesprungen werden soll**
- Wenn ein bedingter Sprung nicht ausgeführt wird (= “branch not taken“), steht der nachfolgende Befehl unmittelbar nach dem Sprungbefehl**
- ob ein bedingter Sprung taken oder not taken ist, wird erst nach Auswertung der Sprungbedingung klar, d.h. nach der Execute-Phase**
- Bis dahin weiß die Fetch Unit nicht, von welcher Speicherzelle sie den nächsten Befehl holen soll = Steuerflußkonflikt**
- Zur Verringerung der Zeitverluste bei einem Steuerflußkonflikt kann man versuchen, vorauszusagen, ob ein bedingter Sprung genommen werden wird**
- Dies heißt Sprungvorhersage (Branch prediction)**

Sprungvorhersage (Branch Prediction)

- Das Prinzip der Sprungvorhersage ist, vorauszusagen, ob ein bedingter Sprung vermutlich durchgeführt werden wird oder nicht
- Die Voraussage ist nur mit einer gewissen Wahrscheinlichkeit richtig
- bei modernen RISC-Prozessoren liegt die Trefferquote der Schätzung bei über 90%
- Damit sich die Voraussage lohnt, muß die Prognose vor der Ausführung der Sprungbedingung, d.h. vor Ende der Execute Phase vorliegen
- die Sprungvorhersage versucht, die Schätzung, ob gesprungen wird, in die Dekodierphase oder sogar in die Fetch-Phase vorzuverlegen
- Dazu ist im Prozessor eine aufwendige Spezialeinheit für Sprünge erforderlich, die sog. Branch Unit

Branch Unit

- Das Arbeitsprinzip der Branch Unit beruht darauf, daß ein endlicher Automat durch Sprünge, die in der Vergangenheit ausgeführt wurden, auf Voraussagen für die Zukunft „trainiert“ wird
- Durch eine Branch Unit kann der Zeitverlust durch Steuerflußkonflikte bei erfolgreicher Schätzung auf 1 Takt verringert werden

- Durch weitere Optimierungen sind im besten Falle sogar 0 Takte Verzögerung möglich
- Alle modernen Prozessoren sind mit einer Branch Unit ausgestattet. Sie trägt wesentlich zur Leistungsfähigkeit eines RISC-Prozessors bei
- bei einem unbedingten Sprung kommt es nicht darauf an, zu schätzen, ob der Sprung genommen wird
- Hier ist es wichtig, möglichst früh zu wissen, wie die Adresse des Nachfolgebefehls lautet
- Dazu existiert eine zweite Einheit im Prozessor, die die Nachfolgeadresse schätzt: der Sprungzielspeicher (Branch Target Cache)

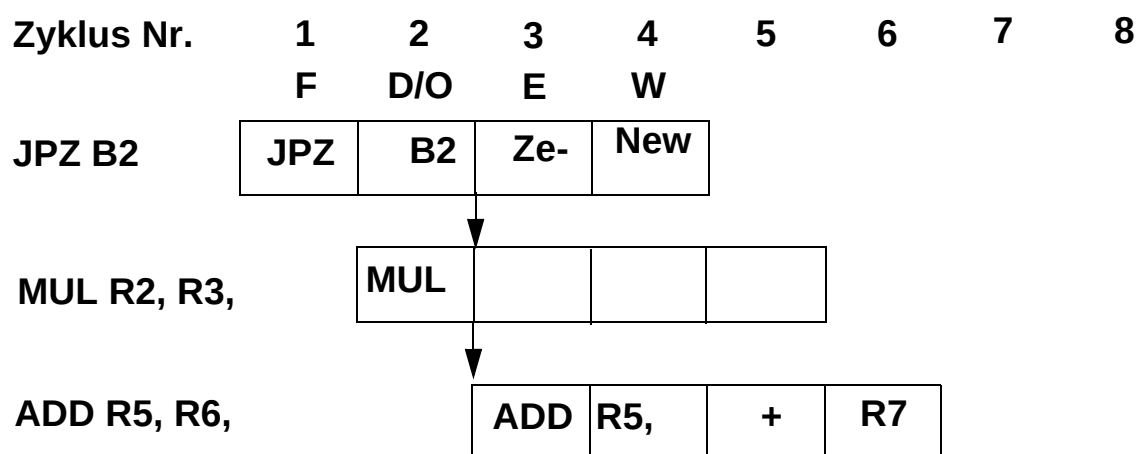
Branch Target Cache

- Das Schätzen der Nachfolgeadresse wird als Sprungzielvorhersage bezeichnet
- Bei bedingten Sprüngen ist der Sprungzielspeicher ebenfalls nützlich:
- Die Branch Unit schätzt, ob der Sprung genommen wird
- Der Branch Target Cache schätzt vor, zeitgleich mit oder nach der Branch Unit die mutmaßliche Nachfolgeadresse
- Im branch-not-taken-Fall wird sie wieder verworfen

- Bei einigen RISC-Prozessoren wird der Sprungzielspeicher als ein fester Bestandteil der Branch Unit angesehen und nicht mehr extra im Blockschaltbild aufgeführt
- In diesem Fall ist die Branch Unit für bedingte und für unbedingte Sprünge zuständig

Beispiel: bedingter Sprung mit Sprungvorhersage und Sprungzielvorhersage

- B1: JPZ B2 -- Jump on Zero Flag, branch taken
- B1+1: MUL R2, R3, R4
-
- B2: ADD R5, R6, R7



1.) In der Fetch-Phase wird der JPZ-Befehl gelesen

2.) In der Dekodier-/Operanden-Phase wird

- der JPZ-Befehl dekodiert
- der Ausgang der Sprungbedingung geschätzt (taken/not taken)

- im taken-Fall die Nachfolgeadresse vom Branch Target Cache geschätzt, sowie die Pipeline invalidiert und einen Takt später eine neue Fetch-Phase bei der mutmaßlichen Nachfolgeadresse begonnen

3.) In der Ausführungsphase wird

- die Sprungbedingung ausgewertet
- bei Fehlprognose die Pipeline nochmal invalidiert

4.) In der Rückschreibephase wird

- der PC auf die berechnete Nachfolgeadresse gesetzt

- Die Fetch-Phase des ADD-Befehls wird bei richtiger Vorhersage nur um 1 Takt verzögert (D/O-Phase)

- Bei falscher Vorhersage, muß die Pipeline nach Phase 3 (Execute) invalidiert und neu geladen werden

=> Fetch-Pase des ADD-Befehls (B2) wird um 2 Takte verzögert (D/O-Phase + Execute-Phase)

- Bei einem bedingten Sprung ist die Vorverlegung von Sprungvorhersage und Sprungzielvorhersage in die Dekodierphase kompliziert, weil

1.) Die Nachfolgeadresse real erst nach Auswertung der Sprungbedingung zur Verfügung steht, d.h. erst nach der Execute-Phase

2.) Die Sprungbedingung nicht von der Branch Unit sondern von der ALU ausgewertet wird. Die Branch Unit kann mangels eigener Arithmetikfähigkeiten nur schätzen.

3.) die Pipeline bei geschätztem taken-Fall und Fehler in der Schätzung insgesamt 2 Mal invalidiert werden muß

- Dennoch kann unter bestimmten Umständen auch bei bedingten Sprüngen eine Branch Unit die Taktverluste auf bis zu 0 reduzieren = maximale Beschleunigung der Befehlsausführung

Vorteil von Sprungvorhersage + Sprungzielvorhersage

- Die Sprungvorhersage steigert zusammen mit der Sprungzielvorhersage die Effizienz der Befehls-Pipeline bei bedingten Sprüngen auf > 95%

Nachteil von Sprungvorhersage + Sprungzielvorhersage

- Wird wegen einer falschen Sprung-/Sprungzielvorhersage der falsche Nachfolgebefehl geholt, gibt es nach der Bedingungsauswertung eine 2. Pipe-Invalidierung
- Zu den Zeitverzögerungen durch falsche Schätzungen addieren sich die Takte, die dadurch verloren gehen, daß die Pipeline wieder neu gefüllt werden muß
- Bei n Pipeline-Stufen gehen zusätzlich n-1 Takte verloren, bis die Pipeline mit den richtigen Nachfolgebefehlen gefüllt ist

=> erhöhtes Absinken der Effizienz bei Fehlprognose („Fehler sind teuer“)

Maximale Beschleunigung der Befehlsausführung

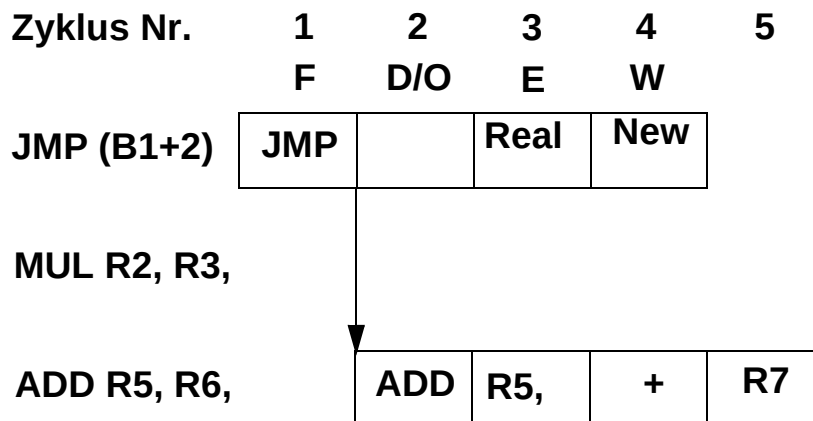
- Idealerweise sollte möglichst frühzeitig folgendes vorausgesagt werden:
 - 1.) wie die Auswertung eines bedingten Sprungs voraussichtlich ausgehen wird und
 - 2.) wie vermutlich die Hauptspeicheradresse des nächsten Befehls nach dem Sprung lauten wird
- Je nach dem, wie frühzeitig die Sprungzielvorhersage vorgenommen wird, beträgt die Zahl der verlorenen Takte bei richtigen Schätzungen 1 bzw. 0
- Bei 0 Takten Verlust hat man eine maximale Beschleunigung der Befehlsausführung
- Man erhält die maximale Beschleunigung, wenn man die Sprungvorhersage nicht abwartet, sondern die Sprungzielvorhersage vor der Sprungvorhersage vornimmt
- Die vorgezogene Sprungzielvorhersage wird erst nach Ende der Sprungvorhersage und nur im taken-Fall relevant. Andernfalls wird sie verworfen.
- Die vorgezogene Sprungzielvorhersage erfolgt bereits in der Fetch-Phase, d.h. bevor der Prozessor überhaupt weiß, daß es sich um einen Sprungbefehl handelt!

- Gleichzeitig mit dem Holen eines neuen Befehls wird bereits die Sprungzielvorhersage durchgeführt
- Die Sprungzielvorhersage benützt als Information zur Prognose die Hauptspeicheradresse, von der der nächste Befehl geholt wird
- Zur Prognose schaut der Branch Target Cache bei sich nach, ob es einen Eintrag unter dieser Hauptspeicheradresse gibt
- Es ist Aufgabe des Branch Target Cache zu speichern, unter welchen Hauptspeicheradressen in der Vergangenheit Sprungbefehle ausgeführt wurden
- Der Branch Target Cache speichert die Hauptspeicheradressen bereits ausgeführter Sprünge zusammen mit deren Befehls-Nachfolgeadressen, wie sie in der Vergangenheit auftraten
- Hat der Branch Target Cache unter der Adresse des zu holenden Befehls einen Eintrag, steht in einer zweiten Spalte des Eintrags die frühere Nachfolgeadresse
- Diese Nachfolgeadresse wird von der Fetch-Stufe verwendet, um von dort einen Takt nach dem Befehls-Fetch den mutmaßlichen Nachfolgebefehl zu holen
- Dazu ist es nicht nötig zu wissen, ob der aktuelle Befehl überhaupt ein Sprungbefehl ist

- Wenn es einen Eintrag im Sprungzielspeicher gibt und dieser richtig ist, gehen 0 Takte verloren

Beispiel 1: unbedingter Sprung mit Sprung- und Sprungzielvorhersage und maximaler Beschleunigung der Befehlsausführung

- B1: JMP (B1+2) -- Jump to Label B1+2
- B1+1: MUL R2, R3, R4
- B1+2: ADD R5, R6, R7



1.) In der Fetch-Phase wird

- der nächste Befehl von der Hauptspeicheradresse geholt, auf die der PC zeigt
- gleichzeitig wird im Branch Target Cache nachgeschaut, ob diese Hauptspeicheradresse einen Eintrag hat
- falls ja, gibt der Branch Target Cache im selben Takt die mutmaßliche Nachfolgeadresse an die Fetch-Stufe weiter

2.) In der Dekodier-/Operanden-Phase wird

- **der JMP-Befehl dekodiert. Ab hier weiß die Pipeline, daß es sich um einen Jump-Befehl handelt**
- **parallel dazu der nächste Befehl von der Fetch-Stufe von der mutmaßlichen Nachfolgeadresse geholt**

3.) In der Ausführungsphase wird

- **die tatsächliche Nachfolgeadresse berechnet**
- **bei Fehlprognose die Pipeline invalidiert**

4.) In der Rückschreibephase wird der PC auf den richtigen Wert gesetzt

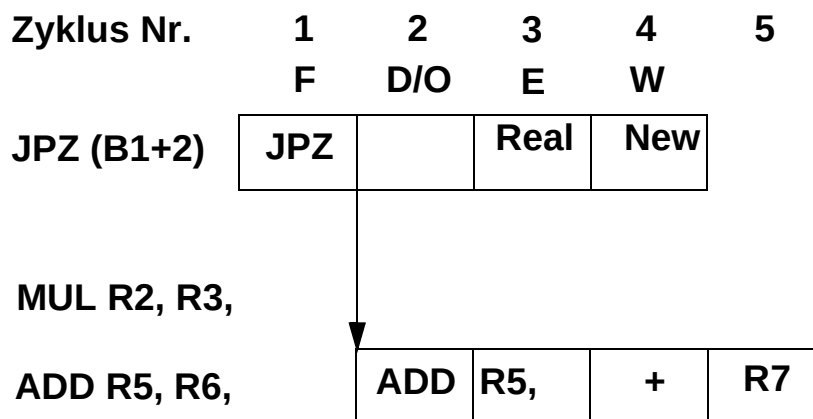
=> Fetch-Phase des ADD-Befehls wird bei richtiger Vorhersage nicht verzögert

=> Die Befehls-Pipeline kann den Nachfolgebefehl bereits in der nächsten Befehlsholphase lesen und bleibt so ununterbrochen gefüllt

- **Bei falscher Vorhersage wird der Beginn des Neufüllens der Pipeline um 3 Takte verzögert, da erst am Ende der Execute-Phase die Pipeline neu gefüllt werden kann**
- **Das Neufüllen kostet 3 weitere Takte, bis die Pipeline wieder voll arbeitet**

Beispiel 2: bedingter Sprung
mit Sprungvorhersage und Sprungzielvorhersage
und maximaler Beschleunigung der Befehlsausführung

- B1: JPZ (B1+2) -- Jump on Zero
- B1+1: MUL R2, R3, R4
- B1+2: ADD R5, R6, R7



1.) In der Fetch-Phase wird

- der nächste Befehl von der Hauptspeicheradresse geholt, auf die der PC zeigt
- gleichzeitig im Branch Target Cache nachgeschaut, ob diese Hauptspeicheradresse einen Eintrag hat
- falls ja, gibt der Branch Target Cache im selben Takt die mutmaßliche Nachfolgeadresse an die Fetch-Stufe weiter

2.) In der Dekodier-/Operanden-Phase wird

- der JPZ-Befehl dekodiert. Ab hier weiß die Pipeline, daß es sich um einen bedingten Sprungbefehl handelt
- parallel dazu der nächste Befehl von der Fetch-Stufe von der mutmaßlichen Nachfolgeadresse geholt

- geschätzt, ob es ein taken- oder ein not taken-Fall ist, falls es sich um einen bedingten Sprungbefehl handelt
- die Sprungzielvorhersage verworfen und die Pipeline invalidiert, um die Verluste möglichst klein zu halten, falls der Branch Target Cache einen Eintrag gefunden hatte, die Sprungvorhersage aber einen not taken-Fall geschätzt hat. D.h. die Pipeline wird nur bei einem bedingten Sprungbefehl mit mutmaßlichem taken-Fall fortgeführt.

3.) In der Ausführungsphase wird

- die Sprungbedingung ausgewertet
- die tatsächliche Nachfolgeadresse berechnet und an die Fetch-Stufe weitergegeben
- bei falscher Sprungvorhersage oder falscher Sprungzielvorhersage die Pipeline invalidiert

4.) In der Rückschreibephase wird der PC auf den richtigen Wert gesetzt

=> Fetch-Phase des ADD-Befehls wird bei richtiger Vorhersage nicht verzögert

- Andernfalls wird der Beginn des Neufüllens um 2-3 Takte verzögert, da erst am Ende der D/O-Phase oder sogar der Execute-Phase damit begonnen werden kann

Sprungzielspeicher

- Der Sprungzielspeicher enthält die Sprungziele aller zuletzt ausgeführten unbedingten/bedingten Sprünge
- Beim unbedingten bzw. bedingten Sprung, der genom-

men wird, wird vom linearen Kontrollfluß abgewichen

- Die Ziele bedingter Sprungbefehle werden nur gespeichert, wenn sie beim letzten Mal verzweigten**
 - Bei nicht verzweigenden bedingten Sprüngen (not taken) ist die Nachfolgeadresse trivial**
 - Sprungzielspeicher haben ca. 64-512 Einträge**
 - Sie sind sehr klein, dafür aber auch sehr schnell**
 - Eine RISC-CPU kann oft noch im selben Takt den Sprungzielspeicher adressieren und auslesen**
 - Wird der Sprungzielspeicher in der Befehlsholphase simultan mit dem Befehls-Cache adressiert, wird die maximale Beschleunigung erzielt**
 - Wenn es einen Eintrag im Sprungziel-Cache gibt, liegt am Ende der 1. Fetch-Phase die Adresse des mutmaßlichen Nachfolgebefehls vor**
- => Ein im Sprungzielspeicher eingetragener Sprung erlaubt, die Pipeline ohne Leerlauf fortzuführen**
- Es gibt einen Eintrag im Sprungzielspeicher wenn:**
 - 1.) der Befehl bereits einmal ausgeführt wurde und**
 - 2.) der Befehl ein Sprungbefehl ist und**

- 3.) wenn er nicht durch einen anderen Sprungbefehl aus dem Sprungzielspeicher verdrängt worden ist

Probleme bei der Sprungzielvorhersage

- Bei der Sprungzielvorhersage durch einen Sprungzielspeicher gibt es Problemfälle:
 - 1.) Es gibt keinen Eintrag im Sprungzielspeicher, weil ein Sprungbefehl zum ersten Mal aus dem Arbeitsspeicher geholt wird (erster Fetch)
 - 2.) Es gibt keinen Eintrag im Sprungzielspeicher, weil der Cache voll ist, und der Eintrag verdrängt worden ist
 - 3.) Der Eintrag im Sprungzielspeicher ist falsch, weil es der letzte Sprung einer Schleife ist. Beim nächsten Mal muß aus der Schleife herausgesprungen werden
 - 4.) Der Eintrag im Sprungzielspeicher ist falsch, weil der Sprung ein alternierendes Verhalten wie „verzweigend“, „nicht verzweigend“, „verzweigend“ hat

Zusammenfassung

- Sowohl bei Datenfluß- als auch bei Steuerflußkonflikten steht das Ergebnis eines Befehls (Operand bzw. neuer PC-Wert) nicht rechtzeitig zur Verfügung
- Die Pipeline muß angehalten/invalidiert werden, weil es an „Nachschub“ von Operanden bzw. Befehlen mangelt

- Dabei fällt die Prozessorleistung u. U. drastisch ab
- Bleibt bei einer Superpipeline der Nachschub an Operanden/Befehlen aus, sind mehr Takte als bei einer "kurzen" Pipeline nötig, um diese wieder zu füllen

=> Der Leistungsabfall aufgrund von schlecht gefüllter Pipeline ist bei Superpipelining größer

- Durch eine Branch Unit kann der Zeitverlust bei Steuerflußkonflikten (Sprüngen) verringert werden
- Die Branch Unit führt eine Sprungvorhersage durch (Branch Prediction)
- Die Sprungvorhersage schätzt, ob ein bedingter Sprung durchgeführt wird oder nicht
- Die Sprungvorhersage erfolgt, bevor die Sprungbedingung von der ALU ausgewertet ist, um möglichst eine blasenfreie Pipeline zu ermöglichen
- Zusätzlich gibt es eine Sprungzielvorhersage, die vom Sprungzielspeicher durchgeführt wird
- Der Sprungzielspeicher führt für bedingte Sprünge, die in der Vergangenheit genommen wurden und für unbedingte Sprünge eine Sprungzielvorhersage durch
- Bemerkung:
- Datenflußkonflikte können nicht durch eine Branch Unit oder einen Sprungzielspeicher gemildert werden

- Vielmehr werden dafür andere Techniken, wie z.B. das sog. Scoreboard eingesetzt

Beispiel für einen Datenflußkonflikt: der sog. RAW-Datenflußkonflikt

- Bei einer Befehls-Pipeline gibt es unter bestimmten Bedingungen eine sog. RAW-Abhängigkeit zwischen zwei Pipeline-Stufen
- Die RAW-Abhängigkeit bewirkt einen RAW-Datenflußkonflikt und verhindert, daß eine Befehls-Pipeline ohne Zusatzeinrichtungen korrekte Resultate liefert
- Diese Zusatzeinrichtungen sind ein sog. Scoreboard in der einfachen oder vollen Ausbaustufe oder ein „Tomasulo-Mechanismus“

Read After Write-Abhängigkeit (RAW)

- RAW ist die häufigste Form der Datenflußabhängigkeit
- RAW spiegelt den logischen Zusammenhang zwischen zwei Befehlen wider (Ursache-Wirkung-Beziehung)
- Beispiel:

B1: ADD R2, R3, R4 -- $R2 = R3 + R4$

B2: ADD R5, R2, R1 -- $R5 = R2 + R1$

- Wenn B2 zuerst R2 liest, bevor B1 geschrieben hat, wird ein ungültiges Ergebnis produziert

- Korrekt ist nur die Reihenfolge R, A, W (= lesen nach dem schreiben)
- Man spricht von einem RAW-Konflikt, wenn die Gefahr besteht, daß die RAW-Reihenfolge nicht eingehalten wird
- Bei einer Befehls-Pipeline kann eine RAW-Abhängigkeit zu einem RAW-Konflikt (= Datenflußkonflikt) führen
- Beispiel:

B1:	ADD R2, R3, R4	F	D	O	E	W
B2:	ADD R5, R2, R1	F	D	O	E	W

- R2 wird von B1 erst beschrieben, nachdem B2 R2 gelesen hat => Fehler
- Der Datenflußkonflikt wird dadurch beseitigt, daß eine geeignete Pipelinestufe in ihrer Dauer verlängert wird
- Der RAW-Konflikt ist eine prinzipielle Schwäche von Befehls-Pipelines
- Sie rührt daher, daß die Rückschreibephase eines Befehl später erfolgt als die Operandenholphase eines Nachfolgebefehls
- Zur Konfliktauflösung muß in obigem Beispiel die Dekodierphase solange verlängert werden, bis R2 beschrieben ist

ADD R2, R3, R4	F	D	O	E	W			
ADD R5, R2, R1		F	D	D	D	O	E	W

- Beispiel für einen RAW-Konflikt, bei dem die Befehle nicht unmittelbar aufeinanderfolgen:

ADD R3, R1, R2	F	D	O	E	W	
ADD R5, R4, R1		F	D	O	E	W

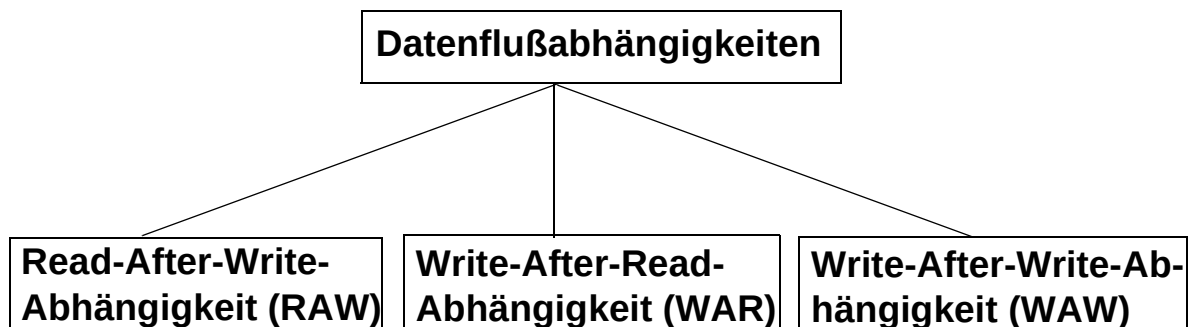
- Mit Verlängerung der Dekodierphase:

ADD R3, R1, R2	F	D	O	E	W	
ADD R5, R4, R1		F	D	O	E	W

- Je mehr Pipeline-Stufen es gibt, desto größer ist die Gefahr eines RAW-Konflikts
- Selbst weit entfernte Befehle können bei tiefen Pipelines hinsichtlich Rückschreibe- und Operandenholphase kollidieren
- Scoreboard oder Tomasulo-Mechanismus erkennen RAW-Konflikte und sorgen für eine geeignete Verlängerung der Dekodierphase (bzw. für eine sog. Befehlsumordnung)

Typen von Datenflußabhängigkeiten

- Insgesamt lassen sich Datenflußabhängigkeiten in 3 Gruppen einteilen:



Write-After-Read-Abhängigkeit (WAR)

- Hier handelt es sich um eine völlig andere Klasse von Datenflußabhängigkeit als bei der RAW-Abhängigkeit

- Beispiel:

B1: ADD R2, R3, R4 -- $R2 = R3 + R4$

B2: ADD R4, R5, R6 -- $R4 = R5 + R6$

- Wenn B2 vor B1 ausgeführt wird, wird ein ungültiges Ergebnis produziert (B2 beschreibt R4, bevor B1 R4 gelesen hat)
- Korrekt ist die Reihenfolge „schreiben nach dem lesen“

- Man spricht von einem WAR-Konflikt, wenn die WAR-Reihenfolge nicht eingehalten wird
- Die WAR-Abhängigkeit ist bei einer Befehls-Pipeline unter folgenden Voraussetzung konfliktfrei:
 - 1.) Zwei WAR-abhängige Befehle werden in der Reihenfolge ausgeführt, die der Programmierer vorgesehen hat
 - 2.) Die Rückschreibe-Phase der Pipeline kann die Operandenholphase nicht „überholen“
- => Die WAR-Abhängigkeit führt bei üblichen Befehls-Pipelines zu keinem WAR-Konflikt
- Bei einer speziellen Befehlsabarbeitungsform hingegen, der sog. out-of-order execution (Befehlsumordnung) kann eine WAR-Abhängigkeit zu einem Datenflußkonflikt führen
- Out-of-order execution verletzt die Voraussetzung 1

Write After Write-Abhängigkeit (WAW)

- Beispiel:

B1: ADD R2, R3, R4 --R2 = R3 + R4
B2: ADD R2, R5, R6 --R2 = R5 + R6

- Wenn B2 vor B1 ausgeführt wird, wird ein ungültiges Ergebnis produziert (B2 beschreibt R2, bevor B1 ge-

schrieben hat)

- Korrekt ist die Reihenfolge „schreiben nach dem schreiben“**
- Man spricht von einem WAW-Konflikt, wenn die WAW-Reihenfolge nicht eingehalten wird**
- Diese Gefahr besteht nur bei out-of-order execution**

=> WAW-Konflikte treten bei normalen Befehls-Pipelines nicht auf

- Bemerkung: Auch wenn die Konflikte hier nur auf der Assembler-Ebene diskutiert werden, heißt das nicht, daß es diese auf der Hochsprachenebene nicht gibt**
- Auf Hochsprachenebene treten Steuerfluß- und Datenflußkonflikte sowie sog. Ressourcenkonflikte in verdeckter Form auf, da jede Hochsprache in Maschinensprache übersetzt wird**

Ressourcenkonflikte

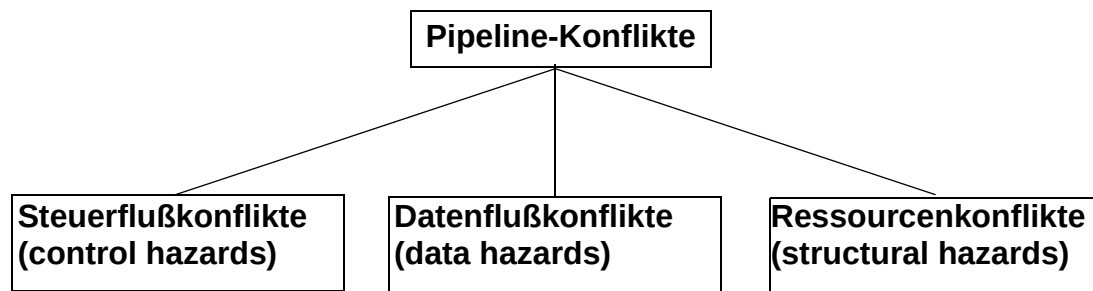
- Ressourcenkonflikte treten auf, wenn mehrere Befehle gleichzeitig dasselbe Betriebsmittel (Speicher, ALU, ...) benötigen

- Beispiel:

B1:	LOAD R1, (R2)	F	D	→	O	E	W	
B2:	ADD R5, R3, R4		F	→	D	O	E	W
B3:	SUB R8, R6, R7			→	F	D	O	E W

- Sowohl in der Operandenholphase von B1 als auch in der Befehlsholphase von B3 muß auf den Hauptspeicher zugegriffen werden
- Ein einfacher Hauptspeicher kann jedoch nur einen Zugriff zu einer Zeit bewerkstelligen (= Engpaß)
- Bei Befehls-Pipelines werden Ressourcen-Engpässe folgendermaßen behoben:
 - Die Fetch-Stufe wird für einen oder mehrere Takte angehalten
 - oder es werden Leerbefehle (NOOPs) eingefügt (hilft nicht bei Hauptspeicherengpaß)
- Ressourcen-Konflikte werden vom Scoreboard der vollen Ausbaustufe erkannt und behoben

Klassifikation der Pipeline-Konflikte



- Ist ein Konflikt aufgetreten, muß
 - eine Pipeline-Stufe in Ihrer Dauer verlängert werden (pipeline stall)
 - oder die ganze Pipeline invalidiert werden (flush)
 - oder es müssen Leer-Befehle (NOOPs) in den Befehlsstrom eingefügt werden, so daß die Pipeline „leerläuft“
- Pipeline-Konflikte können durch verschiedene Maßnahmen vermindert werden, z.B.:
 - Befehlsabarbeitung mit Sprung-/Sprungzielvorhersage; dies beugt Steuerflußkonflikten vor
 - Befehlsabarbeitung unter Aufsicht eines sog. Tomasulo-Mechanismus; beugt RAW-Datenflußkonflikten vor
 - Befehlsabarbeitung unter Aufsicht eines „Scoreboards“ der vollen Ausbaustufe; beugt allen Datenfluß- und Ressourcenkonflikten vor

Grenzen des Pipelinings

1.) Ungleichverteilung der Arbeit:

- Die Arbeit pro Pipeline-Stufe läßt sich nie perfekt ausbalancieren => Die langsamste Stufe bestimmt den Takt

2.) Atomizität der Arbeit:

- Die Arbeit läßt sich nicht in beliebig feine Portionen zerlegen. Nach Unterteilung in ca. 10-20 Schritte ist Schluß.

3.) Ein- und Ausspeicherzeiten in den Latches:

- Damit jede Stufe autonom arbeiten kann, sind Latches zur Entkopplung nötig. Latches kosten zusätzlich Zeit

4.) Steuerflußkonflikte:

- Bei falscher Vorhersage eines bedingten Sprungbefehls holt die Befehls-Pipeline die falschen Nachfolgebefehle

5.) Datenflußkonflikte:

- Bei sequentiellen Programmen ist es schwer, Befehle parallel auszuführen, weil diese voneinander abhängen

6.) Ressourcenkonflikte:

- Diese können an verschiedenen Stellen auftreten:
- Beispiel 1: Wenn eine Pipeline-Stufe den Hauptspeicher braucht, muß sie warten (von Neumann Flaschenhals)
- Beispiel 2: Wenn mehr als eine Pipeline-Stufe dasselbe Rechenwerk braucht, gibt es einen Engpaß
- Mit zunehmender Stufenzahl nehmen solche Konflikte stark zu, da immer mehr Stufen miteinander kollidieren können

=> In der Praxis ist die Zahl der Pipeline-Takte zwischen je zwei fertiggestellten Befehlen > 1

Scoreboard (einfache Ausbaustufe)

- Ein Scoreboard (= Punktetafel beim Sportwettkampf) wurde erstmals Anfang der 60er Jahre bei Mainframes eingesetzt**
- Heutzutage wird es bei allen Mikroprozessoren (in einer vollen Ausbaustufe) verwendet**
- Das Scoreboard ist eine Steuerlogik im Prozessor, die während der Befehlsausführung alle Datenfluß- und Ressourcenkonflikte erkennt und auflöst**
- In der einfachen Ausbaustufe erkennt das Scoreboard nur RAW-Datenflußkonflikte (RAW = erst lesen, nachdem geschrieben wurde)**
- Das einfache Scoreboard wurde bei Befehls-Pipelining eingesetzt**
- Prinzip des einfachen Scoreboards:**
 - 1.) Jedes Register hat ein extra Kennzeichnungsbit für „Registerinhalt gültig/ungültig“ (valid bit)**
 - 2.) Für jeden von der Dekodierstufe ausgegebenen Befehl stellt das Scoreboard fest, in welches Register der Befehl sein Resultat zurückschreiben wird**

- 3.) Das zukünftige Resultatregister eines Befehls wird nach der Decode-Phase als „ungültig“ markiert, da der Inhalt des Registers sich ändern wird, sobald der Befehl fertig ausgeführt ist**
- 4.) Nach der Writeback-Phase wird das Kennzeichnungsbit auf „gültig“ zurückgesetzt**
- 5.) Versucht ein anderer Befehl ein als ungültig markiertes Register zu lesen, wird der Befehl solange verzögert, bis der Registerinhalt gültig ist**

=> Die Erkennung eines RAW-Konflikts geschieht über die Kennzeichnungsbits der Register

- Die Auflösung eines RAW-Konflikts geschieht im Falle von Befehls-Pipelining durch Verlängern der Dauer einer Pipeline-Stufe (Dekodierstufe)**
- Die Dekodier-Phase wird solange verlängert, bis das benötigte Register auf gültig gesetzt ist**
- Bemerkung: Bei einem RISC-Prozessor mit 4 stufiger-Pipeline ist die Operanden-Hol-Phase mit der Dekodierphase vereinigt**

Tomasulo-Mechanismus

- **Der Tomasulo-Mechanismus wurde erstmals Mitte der 60er Jahre bei Mainframes eingesetzt**
- **Er erlaubt genau wie das einfache Scoreboard RAW-Datenflußkonflikte zu erkennen und aufzulösen**
- **Zusätzlich trägt er durch das sog. Operand Forwarding zu einer wesentlichen Steigerung der Ausführungsgeschwindigkeit bei**
- **Tomasulo wird bei Befehls-Pipelining, auch in Kombination mit Arithmetik-Pipelining eingesetzt, um RAW-Konflikte zu beheben**
- **Er löst damit das Problem, welcher Maschinenbefehl in einem Programmausschnitt von einem anderen logisch abhängt (RAW-Abhängigkeit)**
- **Der Tomasulo-Mechanismus stellt in folgenden Punkten eine Erweiterung des einfachen Scoreboards dar:**
 - **Zusätzlich zu den Kennzeichnungsbits „Registerinhalt gültig/ungültig“ hat jedes Register weitere Adreßbits**
 - **Daneben gibt es spezielle Reservierungsregister für die Rechenwerke, sowie einen gemeinsam genutzten Datenbus (Common Data Bus, CDB)**

- Die Kennzeichnungsbits der Register werden ähnlich wie beim einfachen Scoreboard verwendet
- Die Reservierungsregister (Reservation Stations) dienen als „Parkplatz“ für ALU-Operanden
- Der gemeinsam genutzte Datenbus dient schnell benötigten ALU-Operanden als „Überholspur“ außerhalb der normalen Datenpfade
- Zusätzlich zu den Adreßbits in den Registern hat auch jedes Reservierungsregister seine eigenen Adreßbits
- In den Adreßbits wird vermerkt, von welchem Befehl der Inhalt eines Registers bzw. einer Reservation Station abhängt (logische Abhängigkeit)
- Die Menge aller Adreßbits erlaubt, eine CPU-interne, verkettete Liste der Abhängigkeiten von Befehlen aufzustellen
- Die Liste stellt eine partielle Ordnung mit der Ordnungsrelation „ist RAW-abhängig von“ dar
- Das Gültig-Bit dient zur Erkennung von RAW-Datenflußkonflikten
- Die Adreßbits dienen zusammen mit den Reservierungsregistern und dem CDB zur Programmbeschleunigung

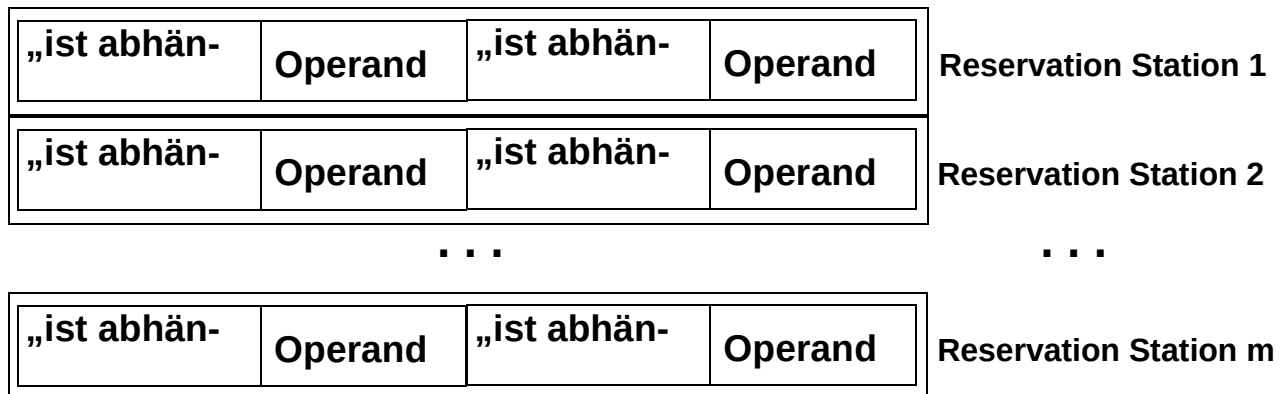
- Die Register nach Tomasulo haben folgenden Aufbau:

Tomasulo-Register

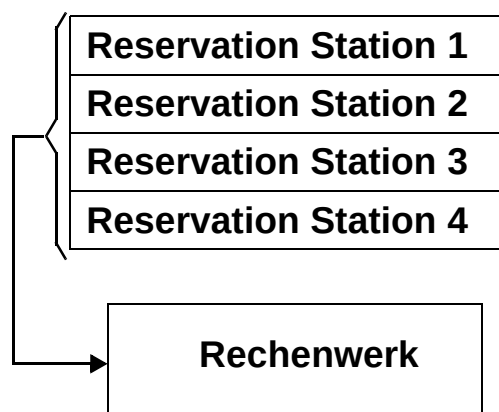
Gültig-Bit	„ist-abhän-	Registerinhalt	R1
Gültig-Bit	„ist-abhän-	Registerinhalt	R2
...			...
Gültig-Bit	„ist-abhän-	Registerinhalt	Rn

- Man geht bei Tomasulo davon aus, daß oft zwei Operanden zur Ausführung eines Befehls benötigt werden (dyadische Operation)
- Operanden sind entweder zur Compile-Zeit bekannte Konstante oder zur Laufzeit berechnete Variable
- Innerhalb eines betrachteten Programmausschnitts wird bei Tomasulo für jeden Operanden eines Befehls ein Platz in einem Reservierungsregister freigehalten
- Sobald ein Operand zur Verfügung steht, wird sein Wert in seinen Platz im Reservierungsregister eingetragen
- Solange ein Operand noch nicht zur Verfügung steht, wird anstelle seines Wertes vermerkt, durch welchen Befehl er berechnet werden wird
- Dies geschieht durch Eintragen der „ist-abhängig-von-Adresse“ in das Reservierungsregister

- Mehrere Reservierungsregister werden zu einem sog. Registerfile zusammengeschlossen
- Die Reservierungsregister haben folgende Struktur:



- Typischerweise besteht ein Registerfile aus 4-8 Reservation Stations
- Das Registerfile ist integraler Bestandteil der ALU
- Beispiel: ALU mit vier Eingangsregistern



- Die Idee der Reservierungsregister ist, daß die ALU die Operanden für die nächsten auszuführenden Befehle selbstständig darin „sammeln“ kann
- Ein Befehl wird dann von der Befehls-/Arithmetik-Pipeline der CPU ausgeführt
 - 1.) wenn er an der Reihe ist und
 - 2.) wenn alle benötigten Operanden in einer Reservation Station „eingesammelt“ sind
- Dies wird als Befehlsausgabe und Befehlsabarbeitung in sequentieller Reihenfolge bezeichnet (in-order-issue, in-order-execution)
- Das einzige, was berücksichtigt werden muß, sind RAW-Konflikte zwischen den Befehlen
- Dies wird mit Hilfe der „Gültig-Bits“ und der „ist abhängig von“-Adreßbits erreicht
- Dazu erhalten alle normalen Register und alle Reservierungsregister eine eindeutige Nummer (Unit-Adresse), auf die sich die Adreßbits beziehen

Beispiel 1:

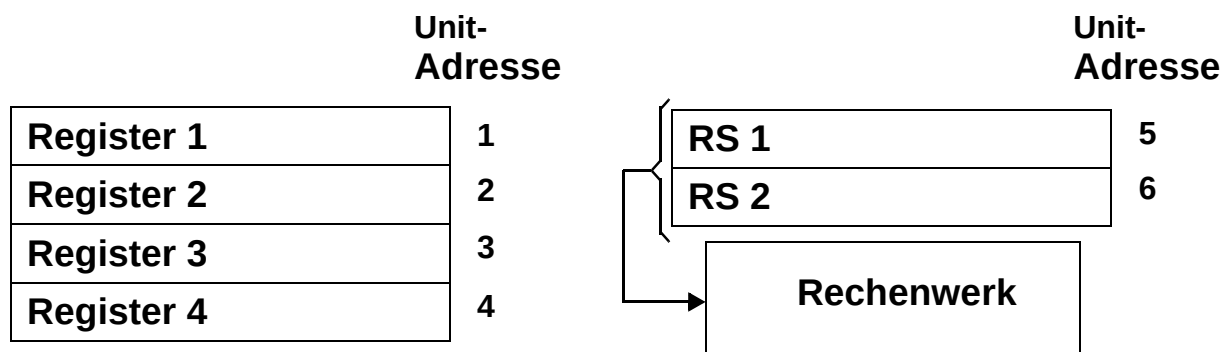
Es gebe eine RAW-Abhängigkeit so, daß zuerst ein Befehl B1 ausgeführt werden muß, dessen Ergebnis in R1 abgelegt wird, bevor der Befehl B2 ausgeführt werden kann, der sein Ergebnis wiederum in R2 ablegt.

Dann ist der grobe Ablauf der folgende:

- Im Adreßfeld des Registers R2 wird die Unit-Adresse von R1 eingetragen
- Das Gültig-Bit von R1 und R2 wird auf ungültig gesetzt
- Nach der Ausführung von B1 wird das R1-Gültig-Bit gesetzt. Genauso nach der Ausführung von B2.

Beispiel 2:

Gegeben sei eine CPU mit 4 Registern, einer ALU und zwei Reservierungsregistern. Die CPU sei als Befehls-Pipeline organisiert.



- Diese CPU soll die folgenden zwei Befehle ausführen

B1: ADD R2, R3, R4 -- $R2 = R3 + R4$
B2: ADD R1, R2, R1 -- $R1 = R2 + R1$

- Zwischen Instruktion 1 und 2 besteht eine RAW-Datenflußabhängigkeit wg. R2

- Der Ablauf für B1 und B2 ist der folgende:

1.) Das Addierwerk beginnt mit der 1. Addition, indem der Inhalt von R3 und R4 in die Operandenfelder 1 und 1' des Reservierungsregisters RS1 kopiert wird

2.) In das Adreßfeld von R2 wird 5, d.h. die Adresse von RS1 eingetragen

- Damit wird vermerkt, daß der Inhalt von R2 von dem Befehl abhängt, dessen Operanden in RS1 stehen

- Gleichzeitig wird das R2-Bit auf ungültig gesetzt

3.) Das Addierwerk beginnt mit der 2. Addition, indem der Inhalt von R1 in das Operandenfeld 2 des Reservierungsregisters RS2 kopiert wird

- Gleichzeitig stellt der Tomasulo-Mechanismus fest, daß in das Operandenfeld 2' der Inhalt von R2 noch nicht kopiert werden kann, weil R2 auf ungültig steht

4.) Die 2. Addition wird angehalten (Pipeline Stall)

- Gleichzeitig wird anstelle des Inhalts von R2 der Inhalt des Adreßfeldes von R2, d.h. 5, in das Adreßfeld des Operandenfelds 2' kopiert

- Damit steht die 5 (=RS1) im Adreßfeld von R2 und RS2

5.) Die 1. Addition wird abgeschlossen, das Ergebnis nach R2 zurückgeschrieben, und das R2-Bit auf „gültig“ gesetzt

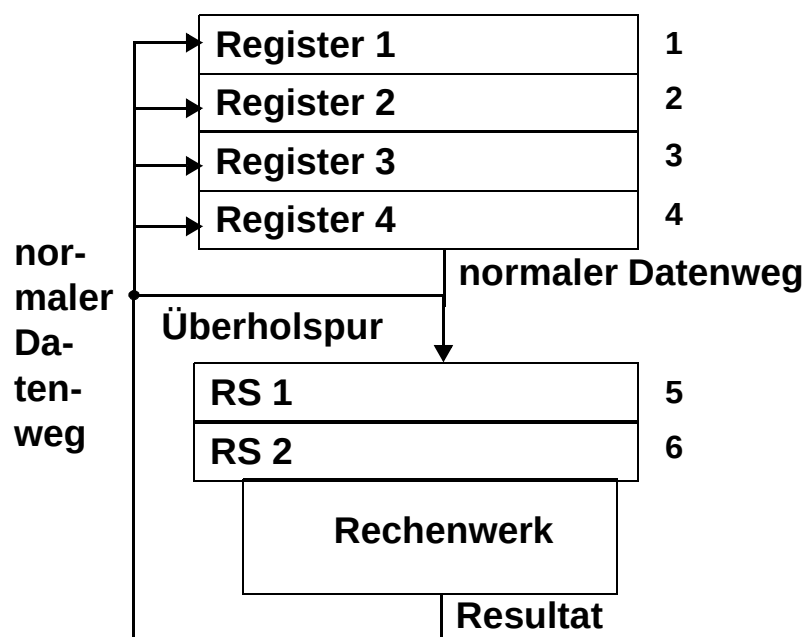
- Gleichzeitig kopiert der Tomasulo-Mechanismus den Inhalt von R2 ins Operandenfeld 2' und setzt das Adreßfeld des Operandenfelds 2' von 5 auf 0 zurück
 - 0 als Eintrag im Adreßfeld heißt, daß der Operand nicht von einer vorangehenden Alu-Operation abhängt, sondern bereits da ist
- 6.) Es wird in das Adreßfeld von R1 die 6 eingetragen, d.h. die RS2-Adresse, weil vom Resultat der RS2-Operanden R1 beschrieben werden wird
- Gleichzeitig wird das R1-Bit auf „ungültig“ gesetzt
- 7.) Die 2. Addition wird fortgesetzt, das Ergebnis nach R1 geschrieben, und das R1-Bit auf „gültig“ gesetzt
- Gleichzeitig wird das Adreßfeld von R1 von 6 auf 0 zurückgesetzt

Common Data Bus (CDB)

- Der vorige Beispielablauf kann beschleunigt werden, indem der CDB als Abkürzungspfad zwischen ALU-Ausgang und Reservierungsregister verwendet wird
- Der CDB wirkt dabei wie eine „Überholspur“
- Über den CDB kann das von der ALU berechnete Ergebnis von B1 direkt ins Operandenfeld 2' kopiert werden, noch bevor es ins R2-Ergebnisregister zurückgeschrieben ist

=> Der Befehl B2 kann direkt nach der Ausführungsphase von B1, d.h. noch vor dessen Rückschreibephase begonnen werden

- Dieses Verfahren wird als „Operand Forwarding“ bezeichnet
- Mit CDB sieht die CPU folgendermaßen aus:



- Damit das Ergebnis von B1 die Überholspur nehmen kann, wird es von der ALU mit der Zusatzinformation „5“, d.h. mit der Adresse von RS1 versehen
- Dies geschieht, weil B2 das Resultat von B1 braucht und RS1 den Befehl B1 widerspiegelt
- Die oben gezeigte Architektur ist Stand der Technik

- **ALU-Resultate werden stets zusammen mit der Unit-Adresse des Reservierungsregisters, das den zugehörigen Befehl widerspiegelt, als Broadcast auf den CDB gegeben**
- **Diejenigen Register/Reservierungsregister, die die auf dem CDB übertragene Unit-Adresse in einem Adreßfeld stehen haben, lesen das Datum vom CDB und tragen es ins zugehörige Register/Operandenfeld ein**
- **In unserem Beispiel lesen R2 und RS2 gleichzeitig das von RS1 bewirkte Ergebnis und speichern es ebenfalls gleichzeitig**

=> Zeitersparnis bei Operand Forwarding