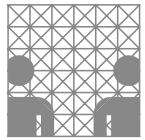


64-040 Modul InfB-RSB
Rechnerstrukturen und Betriebssysteme

https://tams.informatik.uni-hamburg.de/lectures/2019ws/vorlesung/rsb

Andreas Mäder

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

Wintersemester 2019/2020

Gliederung

Gliederung (cont.)

14. Rechnerarchitektur II
15. Betriebssysteme

Gliederung

Gliederung (cont.)

8. Schaltfunktionen
9. Schaltnetze
10. Schaltwerke
11. Rechnerarchitektur I
12. Instruction Set Architecture
13. Assembler-Programmierung
14. Rechnerarchitektur II
15. Betriebssysteme

Brockhaus-Enzyklopädie: „*Informatik*“

Die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Digitalrechnern (→ Computer). . . .

Brockhaus-Enzyklopädie: „*Informatik*“

Die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Digitalrechnern (→ Computer). . . .

system. Verarbeitung: von-Neumann Konzept

- Wie löst eine Folge elementarer Befehle (Programm) ein Problem?

⇒ Softwareentwicklung, Programmierung

Brockhaus-Enzyklopädie: „*Informatik*“

Die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Digitalrechnern (→ Computer). . . .

system. Verarbeitung: von-Neumann Konzept

- Wie löst eine Folge elementarer Befehle (Programm) ein Problem?

⇒ Softwareentwicklung, Programmierung

Digitalrechner: das technische System dazu (Rechnerarchitektur)

- Wie wird Information (Zahlen, Zeichen) repräsentiert/codiert?
- Wie werden Befehle effizient auf der Hardware abgearbeitet?

⇒ Hardwareentwicklung

Trennung von Hardware und Software

unterschiedliche Paradigmen

- SW: Hardware ist „notwendiges Übel“
  - Abstraktion von der Hardware
  - Entwicklung in Hochsprachen (Produktivität)
- HW: Optimierungsziel sind technische Werte

  = Taktfrequenz, Latenz, Durchsatz, Leistungsaufnahme etc.
  - Maschinenbefehl wird auf Hardwarearchitektur ausgeführt
  - technische Entwicklung: *Moore's Law*

  ...

Trennung von Hardware und Software

unterschiedliche Paradigmen

- SW: Hardware ist „notwendiges Übel“
  - Abstraktion von der Hardware
  - Entwicklung in Hochsprachen (Produktivität)
- HW: Optimierungsziel sind technische Werte

  = Taktfrequenz, Latenz, Durchsatz, Leistungsaufnahme etc.
  - Maschinenbefehl wird auf Hardwarearchitektur ausgeführt
  - technische Entwicklung: *Moore's Law*

- dies funktioniert seit Jahren! . . .

Trennung von Hardware und Software

unterschiedliche Paradigmen

- SW: Hardware ist „notwendiges Übel“
  - Abstraktion von der Hardware
  - Entwicklung in Hochsprachen (Produktivität)
- HW: Optimierungsziel sind technische Werte

  = Taktfrequenz, Latenz, Durchsatz, Leistungsaufnahme etc.
  - Maschinenbefehl wird auf Hardwarearchitektur ausgeführt
  - technische Entwicklung: *Moore's Law*

- dies funktioniert seit Jahren . . . bis Ende 2017

. . .

Trennung von Hardware und Software

unterschiedliche Paradigmen

- SW: Hardware ist „notwendiges Übel“
  - Abstraktion von der Hardware
  - Entwicklung in Hochsprachen (Produktivität)
- HW: Optimierungsziel sind technische Werte

  = Taktfrequenz, Latenz, Durchsatz, Leistungsaufnahme etc.
  - Maschinenbefehl wird auf Hardwarearchitektur ausgeführt
  - technische Entwicklung: *Moore's Law*

- dies funktioniert seit Jahren . . . bis Ende 2017

. . .

Problem

verschiedene Sichtweisen

⇒ Vorlesung RSB: *Wie funktioniert ein Digitalrechner?*

Motivation

Wie funktioniert ein Digitalrechner?

Warum ist das überhaupt wichtig?

- Informatik ohne Digitalrechner undenkbar
- Grundverständnis der Interaktion von SW und HW
  - für „performante“ Software
  - Sicherheitsaspekte
  - ...
- Systemsicht/Variantenvielfalt von Mikroprozessorsystemen
  - Supercomputer, Server, Workstations, PCs ...
  - Medienverarbeitung, Mobile Geräte ...
  - RFID-Tags, Wegwerfcomputer ...

$\Rightarrow$ Informatik Basiswissen

$\Rightarrow$ Bewertung von Trends und Perspektiven

$\Rightarrow$ Chancen und Grenzen der Miniaturisierung

Fortschritt

1. ständige technische Fortschritte in Mikro- und Optoelektronik mit einem weiterhin *exponentiellen* Wachstum (50 % . . . 100 % pro Jahr)
   - Rechenleistung von Prozessoren („Performanz“)
   - Speicherkapazität Hauptspeicher (DRAM, SRAM, FLASH)
   - Speicherkapazität Langzeitspeicher (Festplatten, FLASH)
   - Bandbreite (Netzwerke)
2. neue Entwurfsparadigmen und -werkzeuge

$\Rightarrow$ Möglichkeiten und Anwendungsfelder

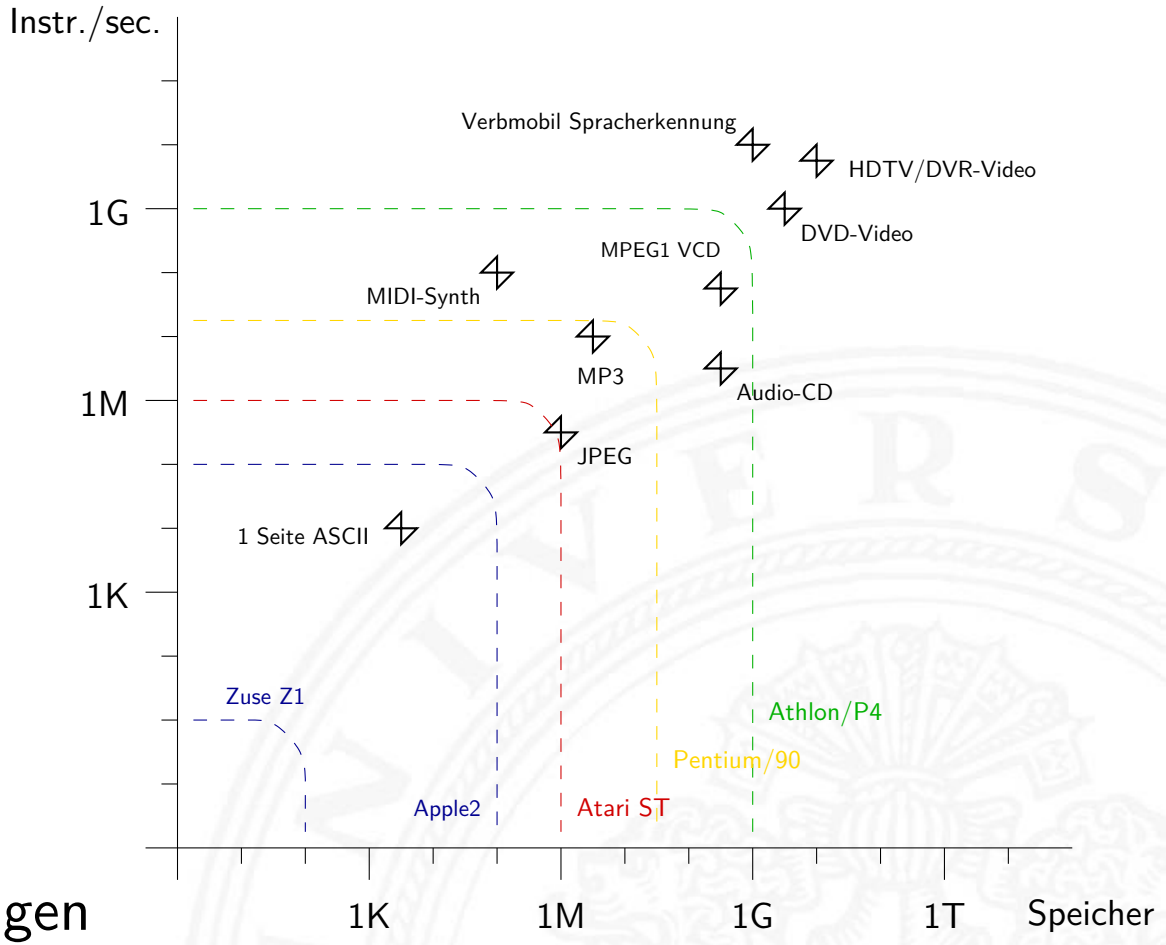
$\Rightarrow$ Produkte und Techniken

Fortschritt (cont.)

Kriterien / Maßgrößen

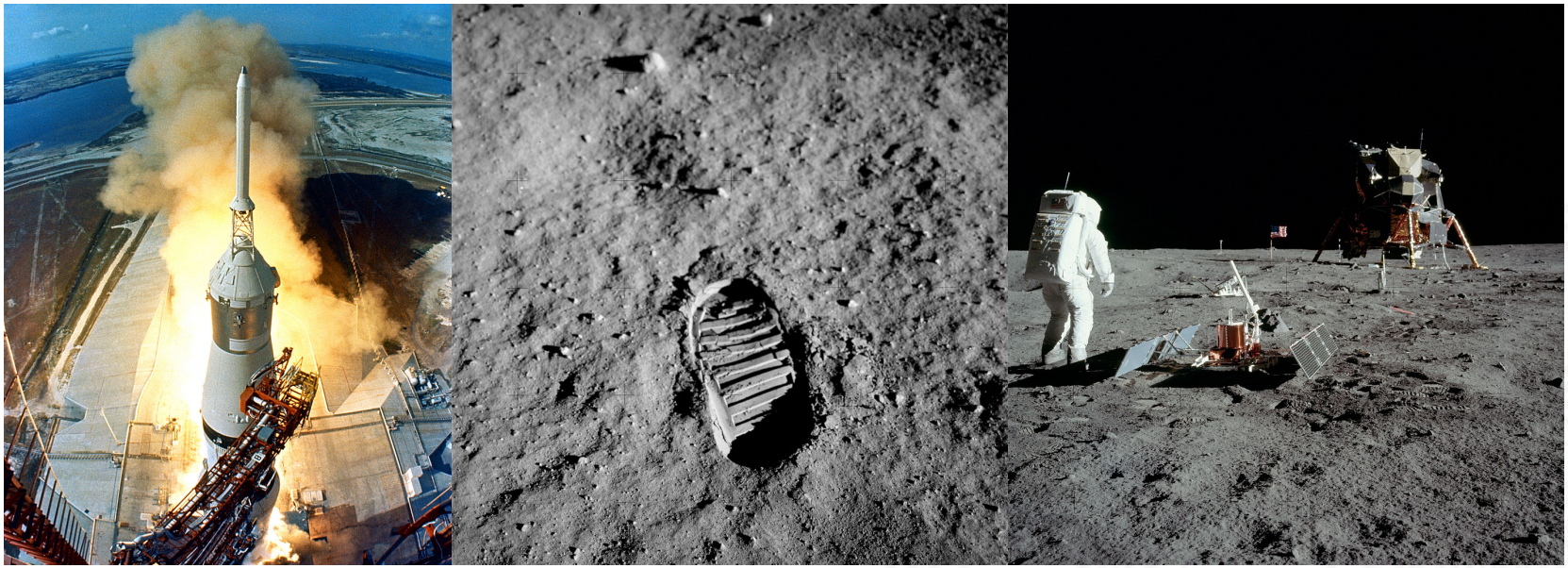
- Rechenleistung: MIPS
- MBytes (RAM, HDD)
- Mbps
- MPixel

⇒ jede Rechnergeneration erlaubt neue Anwendungen



vor 50 Jahren

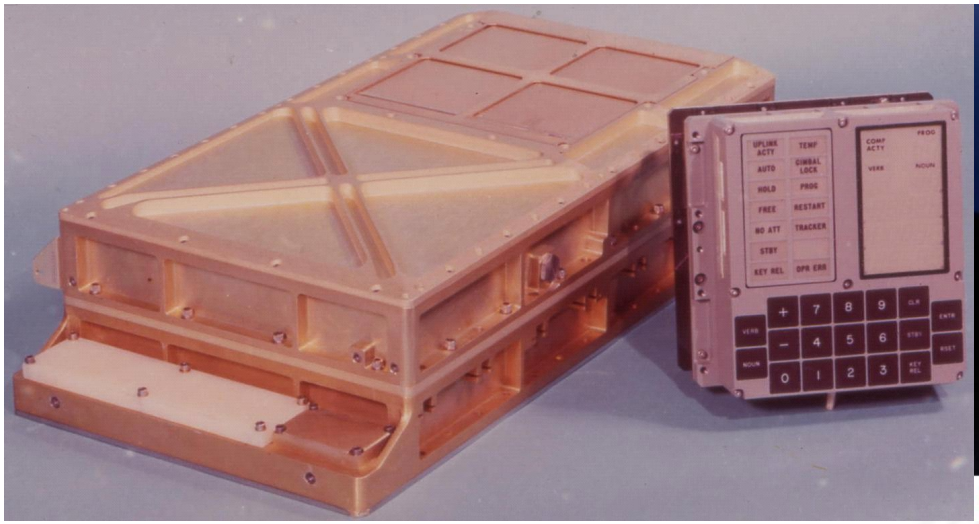
Beispiel: Apollo 11 (1969)



- bernd-leitenberger.de/computer-raumfahrt1.shtml
- www.hq.nasa.gov/office/pao/History/computers/Compspace.html
- en.wikipedia.org/wiki/Apollo_Guidance_Computer
- en.wikipedia.org/wiki/IBM_System/360

vor 50 Jahren (cont.)

1. Bordrechner: AGC (Apollo Guidance Computer)



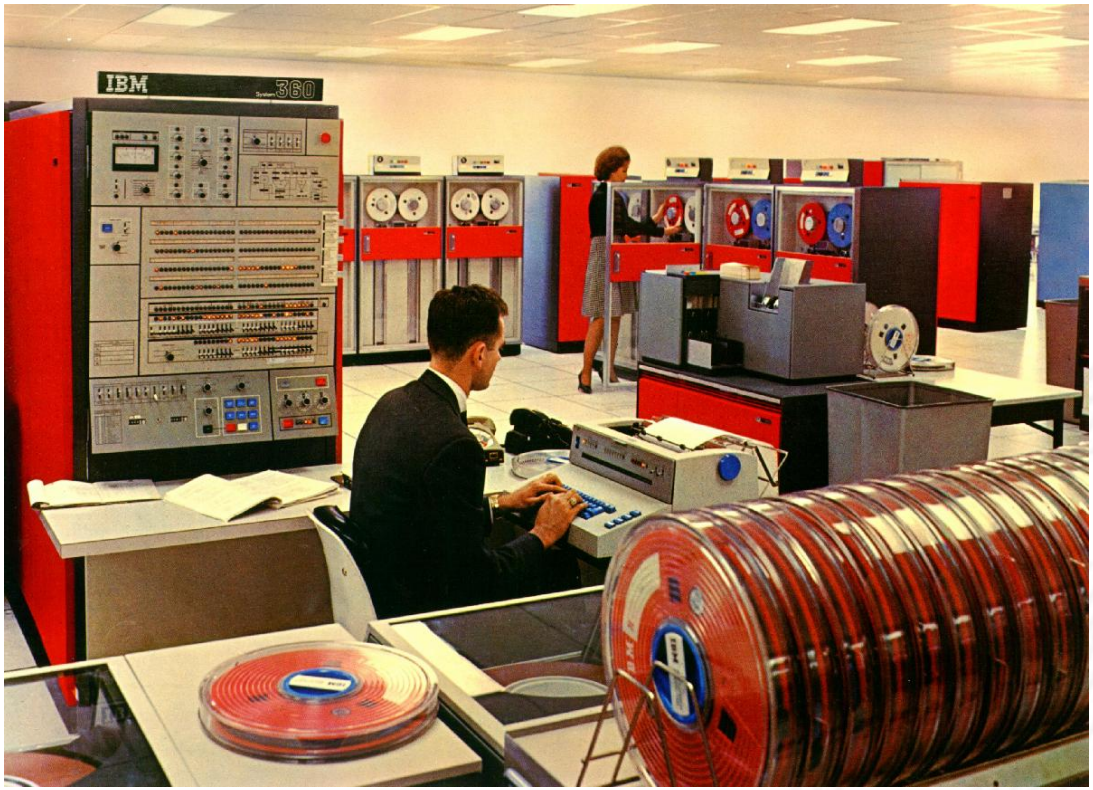
Dryden Flight Research Center EC96-43408-1 Photographed 1996
Apollo computer interface box used in the F-8 Digital
Fly-By-Wire program. NASA photo by Dennis Taylor

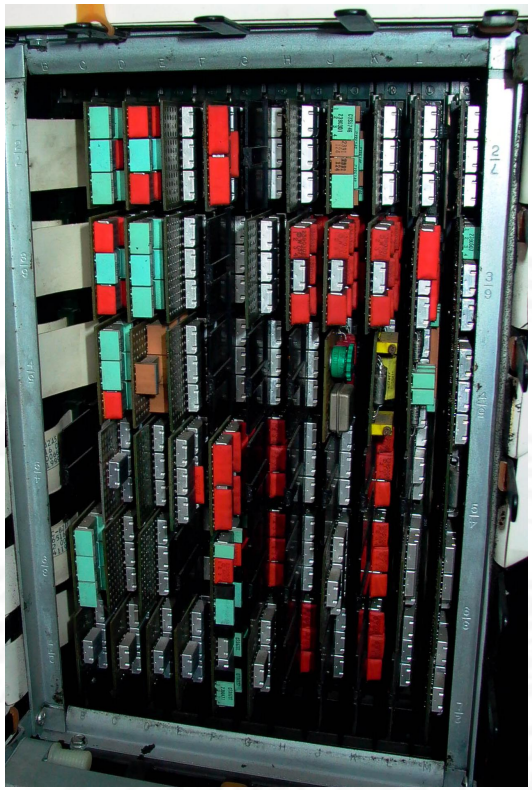
- Dimension $61 \times 32 \times 15{,}0\,\text{cm}$ $31{,}7\,\text{kg}$
  $20 \times 20 \times 17{,}5\,\text{cm}$ $8{,}0\,\text{kg}$
- Taktfrequenz: $1{,}024\,\text{MHz}$
- Addition $20\,\mu\text{s}$
- 16-bit Worte, nur Festkomma
- Speicher ROM 36 KWorte 72 KByte
  RAM 2 KWorte 4 KByte

Zykluszeit 11,7 ms (85 Hz)

vor 50 Jahren (cont.)

2. mehrere Großrechner: IBM System/360 Model 75s





vor 50 Jahren (cont.)

- je nach Ausstattung: Anzahl der „Schränke“
- Taktfrequenz: bis 5 MHz
- 32-bit Worte, 24-bit Adressraum (16 MByte)
- Speicherhierarchie: bis 1 MByte Hauptspeicher (1,3 MHz Zykluszeit)
- (eigene) Fließkomma Formate
- Rechenleistung: 0,7 Dhrystone MIPS

vor 50 Jahren (cont.)

- je nach Ausstattung: Anzahl der „Schränke“
- Taktfrequenz: bis 5 MHz
- 32-bit Worte, 24-bit Adressraum (16 MByte)
- Speicherhierarchie: bis 1 MByte Hauptspeicher (1,3 MHz Zykluszeit)
- (eigene) Fließkomma Formate
- Rechenleistung: 0,7 Dhrystone MIPS

- . . . und 2016

	CPU	Cores	[DMIPS]	F_{clk}[GHz]
Smartphone	Exynos 8890	8	47 840	2,3
Desktop PC	Core i7 6950X	10	317 900	3,0

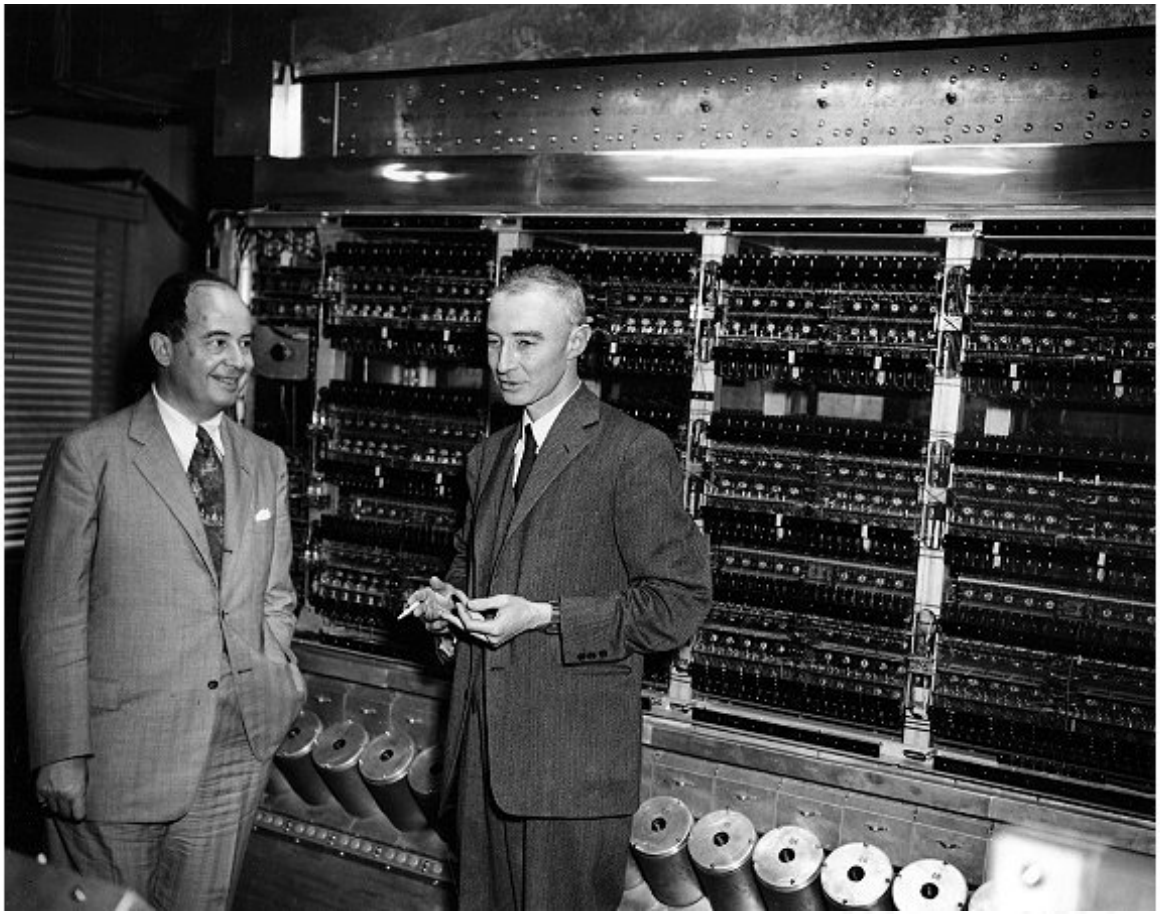
von-Neumann Konzept

- J. Mauchly, J.P. Eckert, J. von-Neumann 1945
- Abstrakte Maschine mit minimalem Hardwareaufwand
  - System mit Prozessor, Speicher, Peripheriegeräten
  - die Struktur ist unabhängig von dem Problem, das Problem wird durch austauschbaren Speicherinhalt (Programm) beschrieben
- gemeinsamer Speicher für Programme und Daten
  - fortlaufend adressiert
  - Programme können wie Daten manipuliert werden
  - Daten können als Programm ausgeführt werden
- Befehlszyklus: Befehl holen, decodieren, ausführen

$\Rightarrow$ enorm flexibel

- **alle** aktuellen Rechner basieren auf diesem Prinzip
- aber vielfältige Architekturvarianten, Befehlssätze usw.

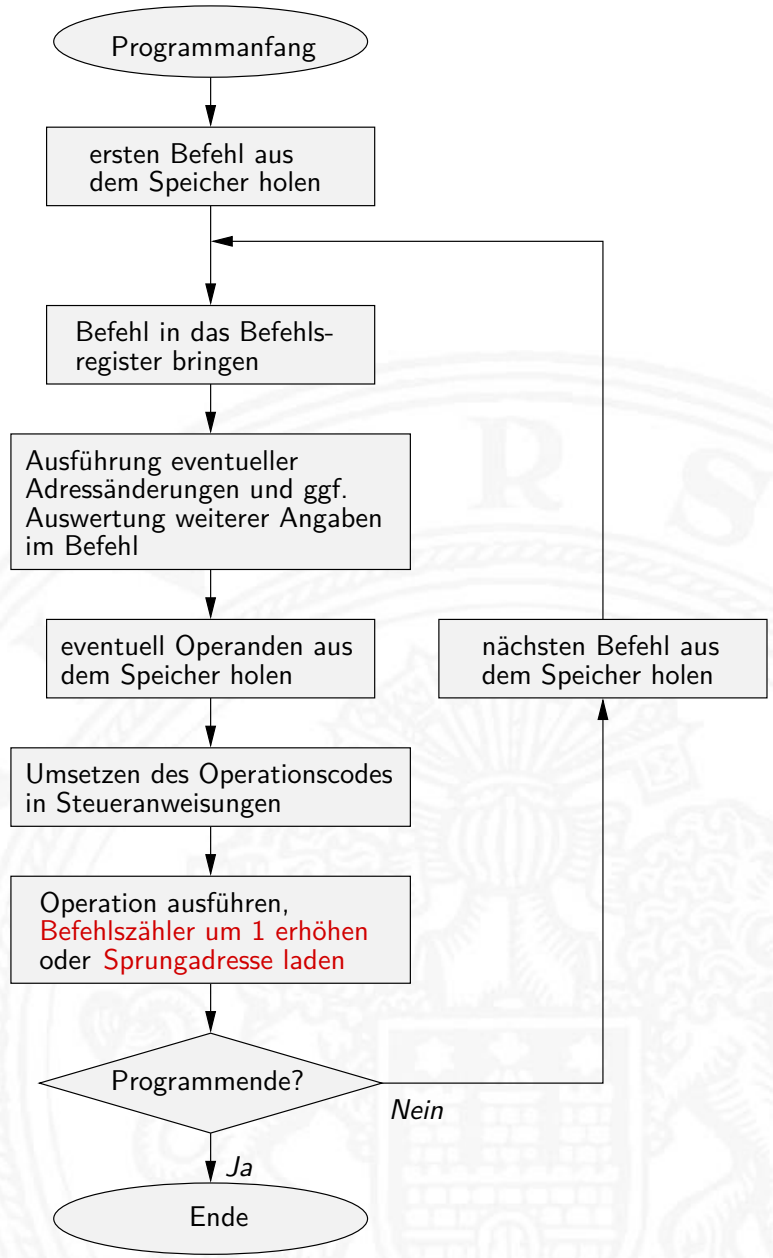
von-Neumann Rechner: IAS Computer



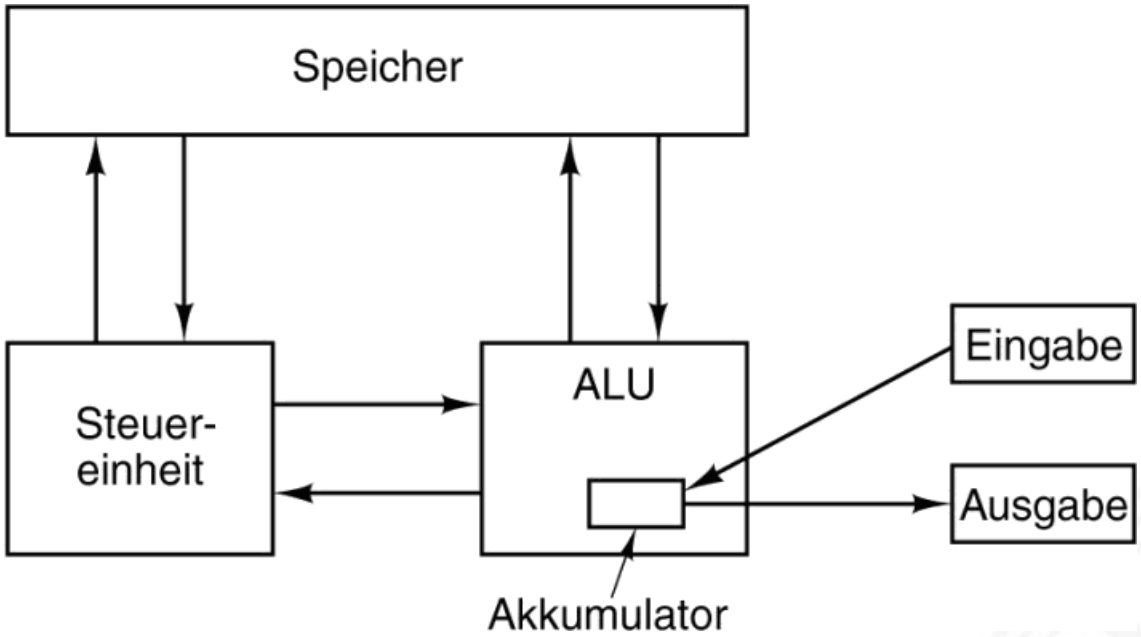
John von Neumann, R. J. Oppenheimer, IAS Computer Princeton `www.computerhistory.org`

Programmverarbeitung

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



von-Neumann Rechner



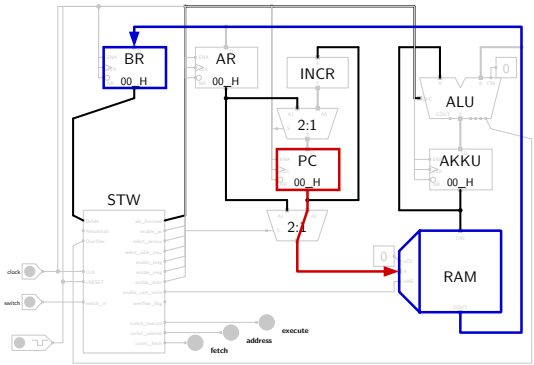
[TA14]

Fünf zentrale Komponenten:

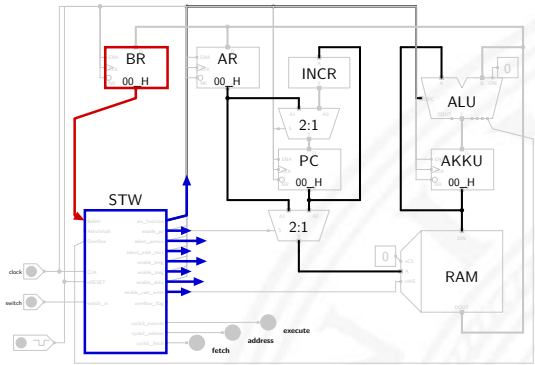
- Prozessor mit **Steuerwerk** und **Rechenwerk** (ALU, Register)
- **Speicher**, gemeinsam genutzt für Programme und Daten
- **Eingabe-** und **Ausgabewerke**
- verbunden durch Bussystem

- Verschaltung der Hardwarekomponenten für alle mögl. Datentransfers
- abhängig vom Befehl werden nur bestimmte Pfade aktiv
- Ausführungszyklus

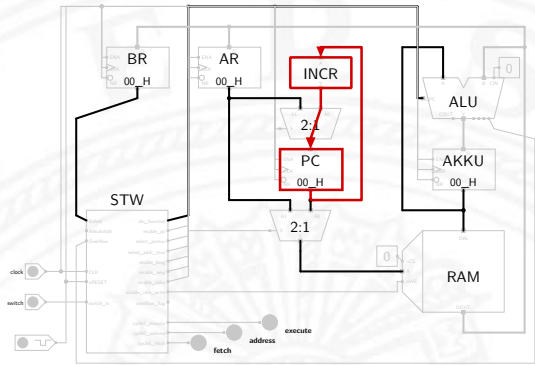
Befehl holen

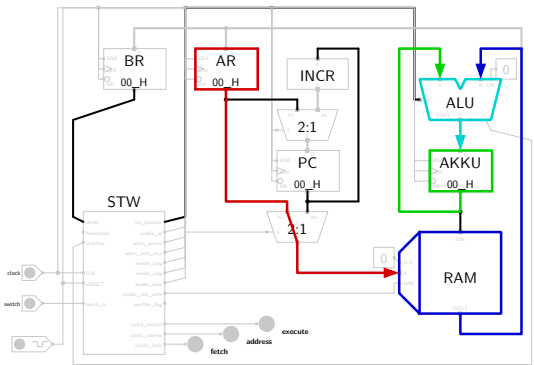


decodieren

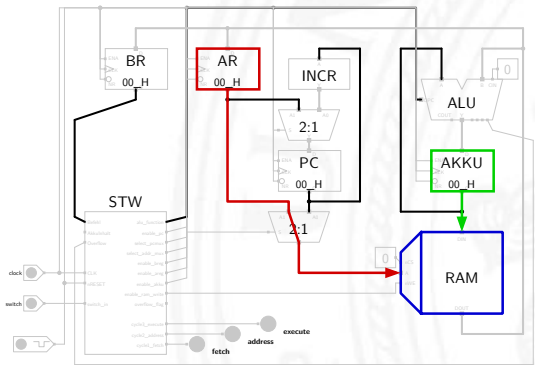


PC inkrementieren

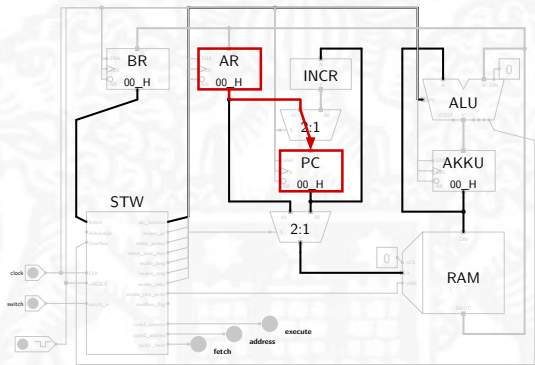




rechnen



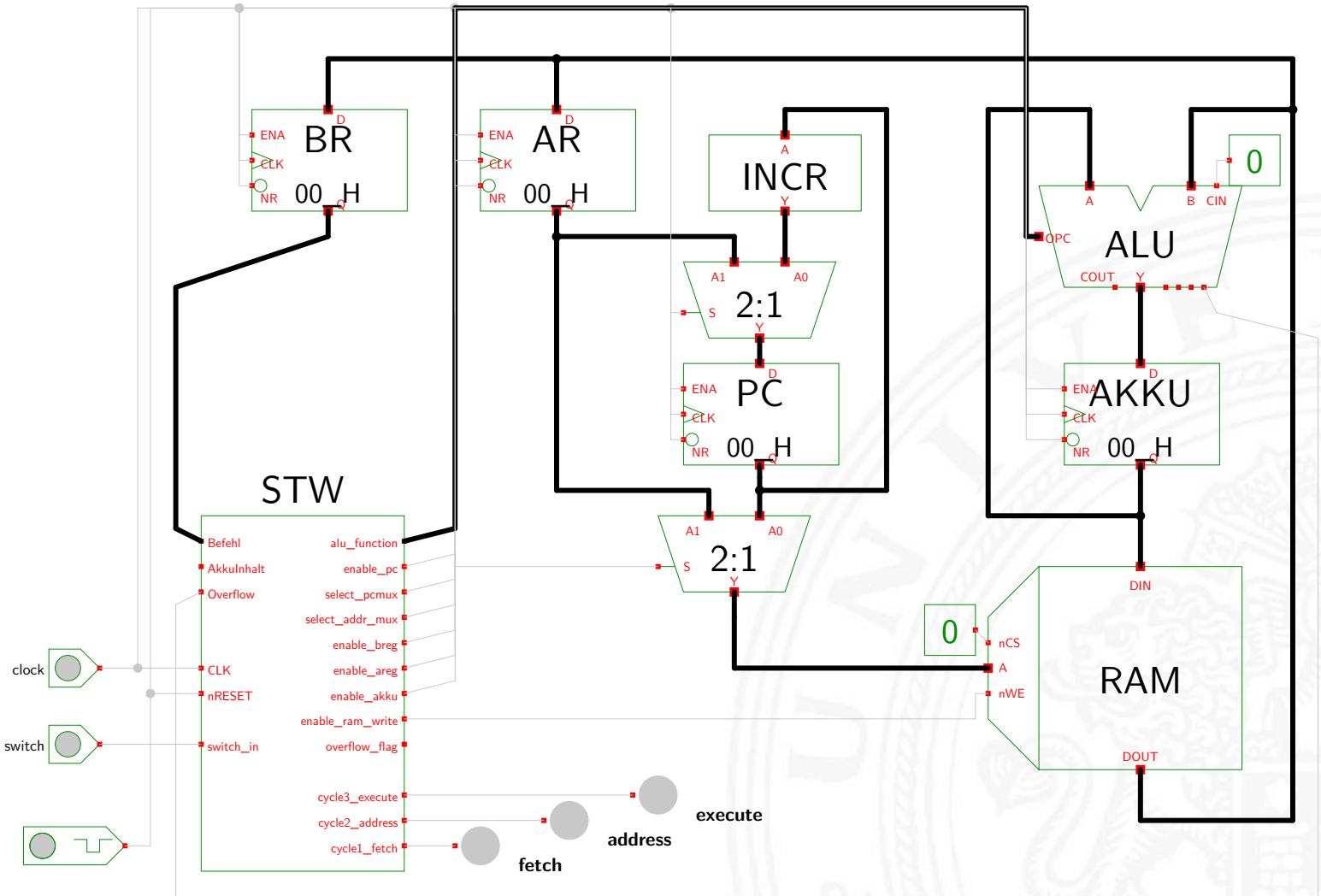
speichern



springen

Beispiel: PRIMA (die primitive Maschine)

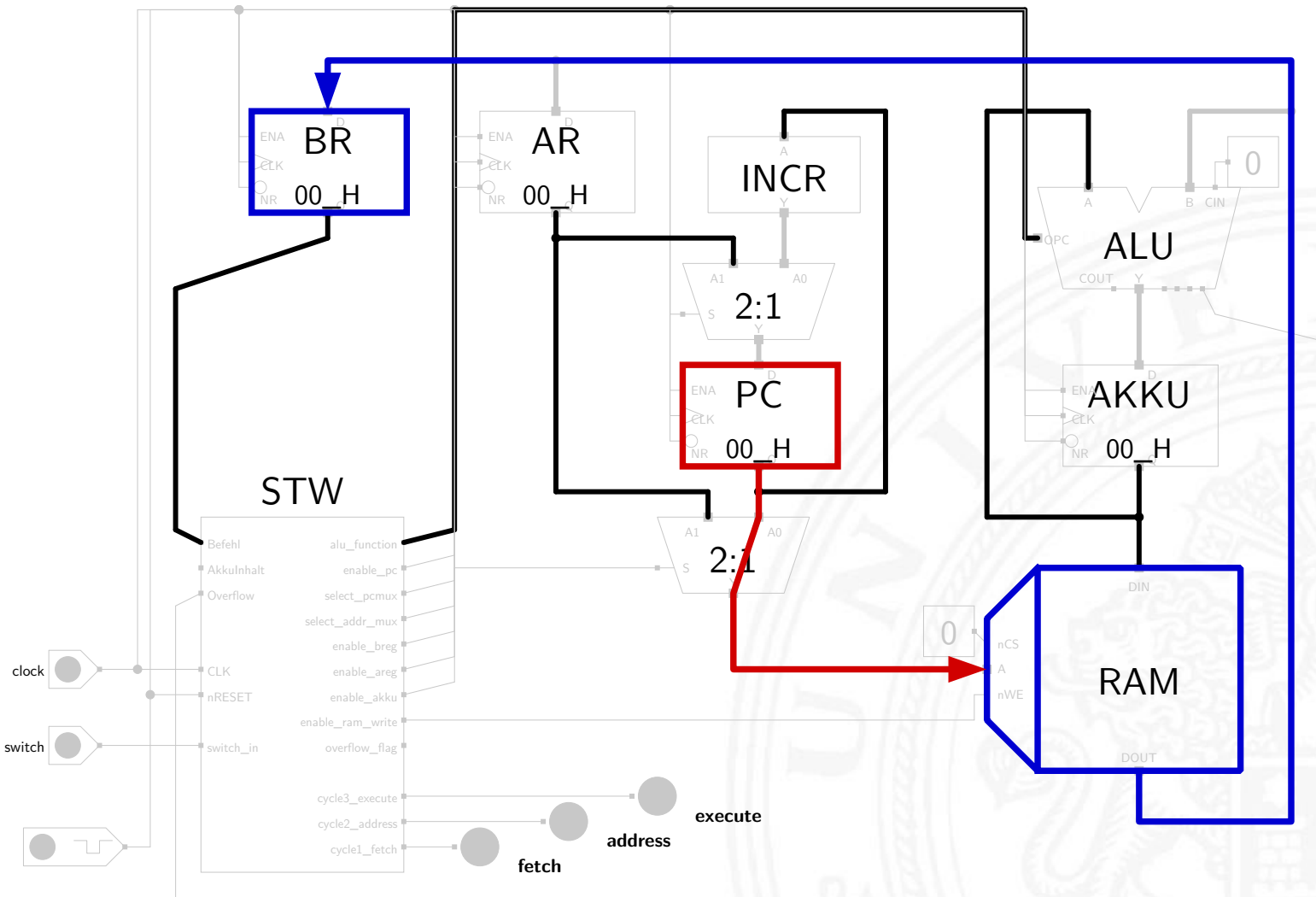
- ein (minimaler) 8-bit von-Neumann Rechner



[HenHA] Hades Demo: `50-rtlib/90-prima/prima`

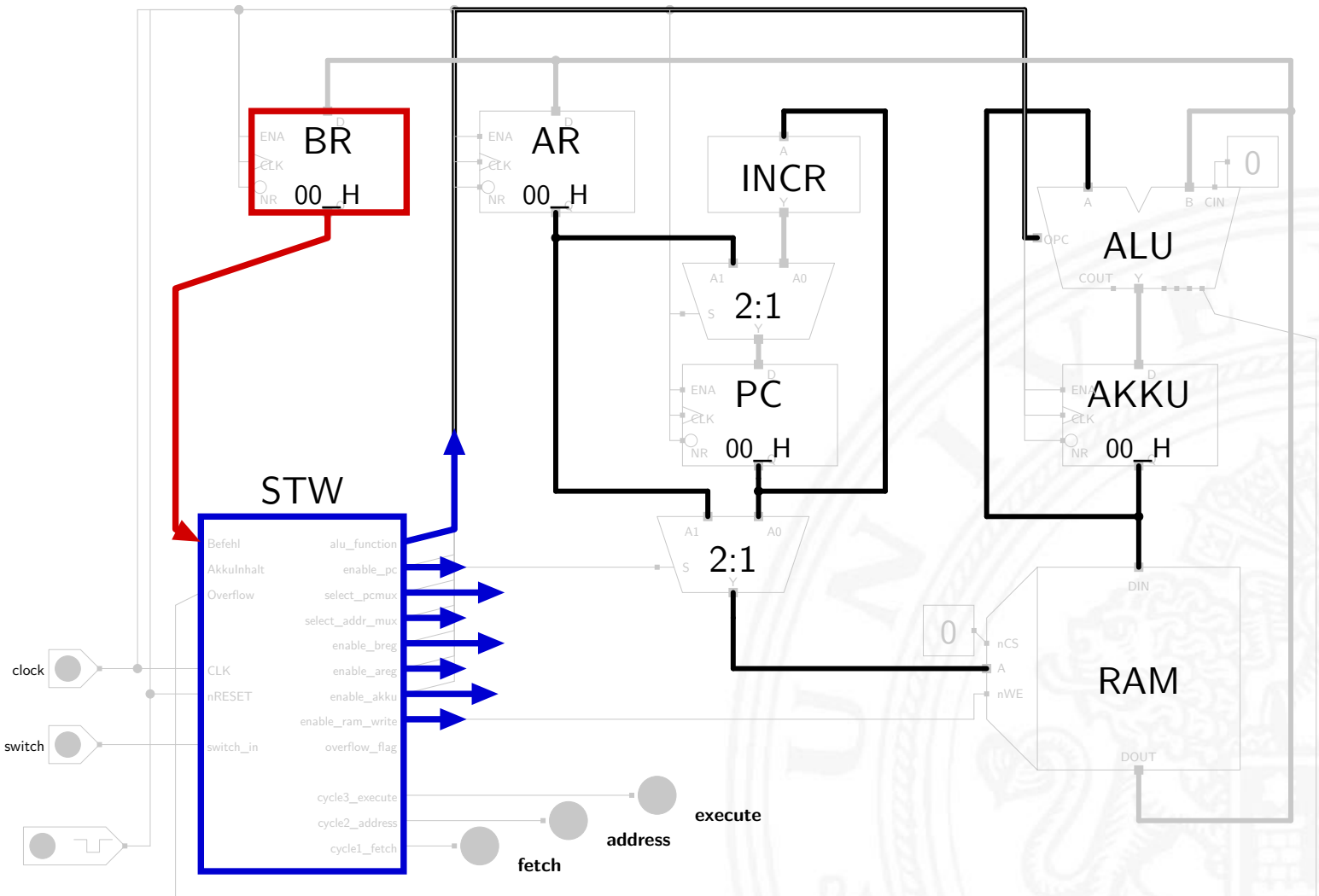
PRIMA: Befehl holen

BR = RAM[PC]



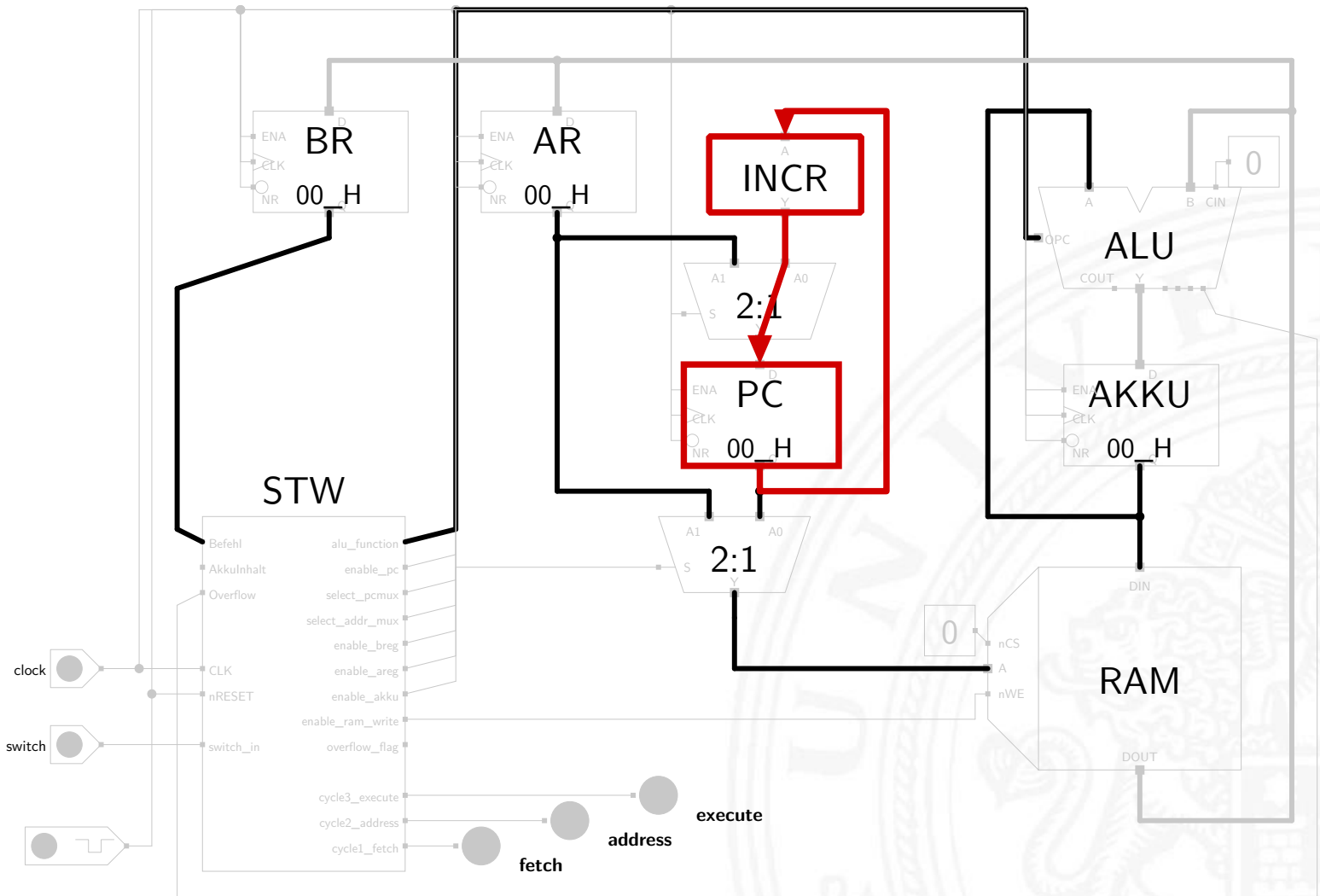
PRIMA: decodieren

Steuersignale = decode(BR)



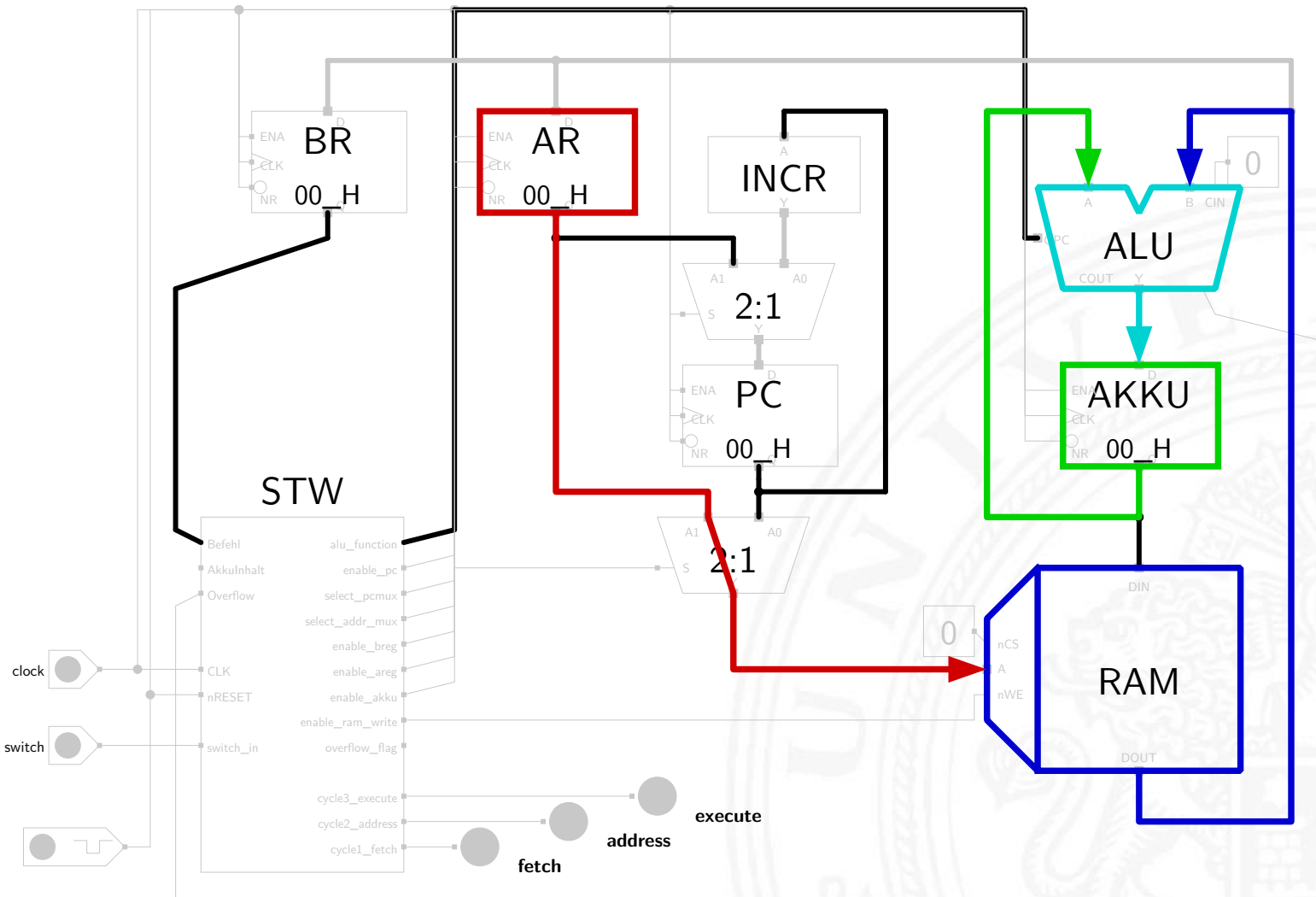
PRIMA: PC inkrementieren

PC = PC+1

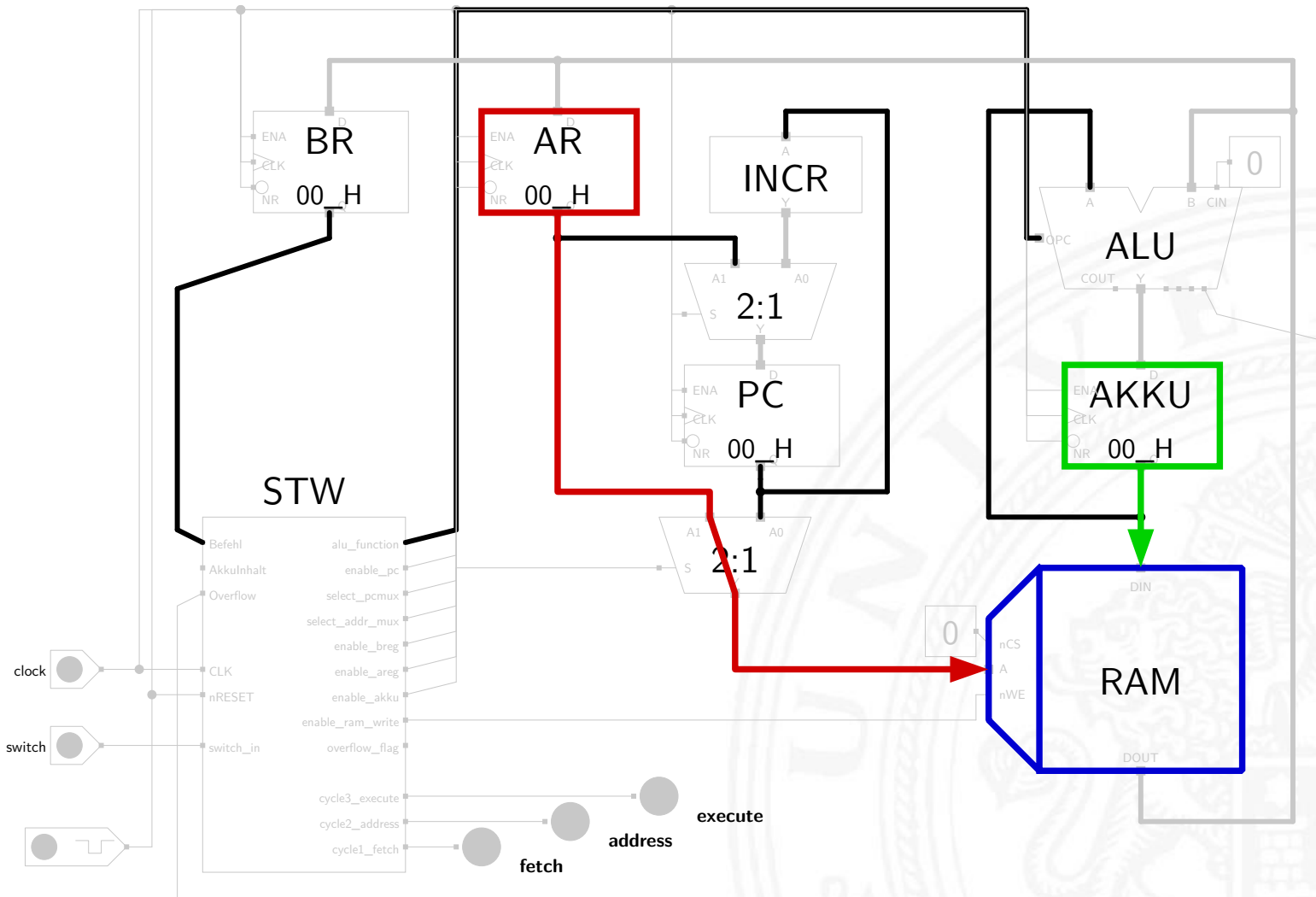


PRIMA: rechnen

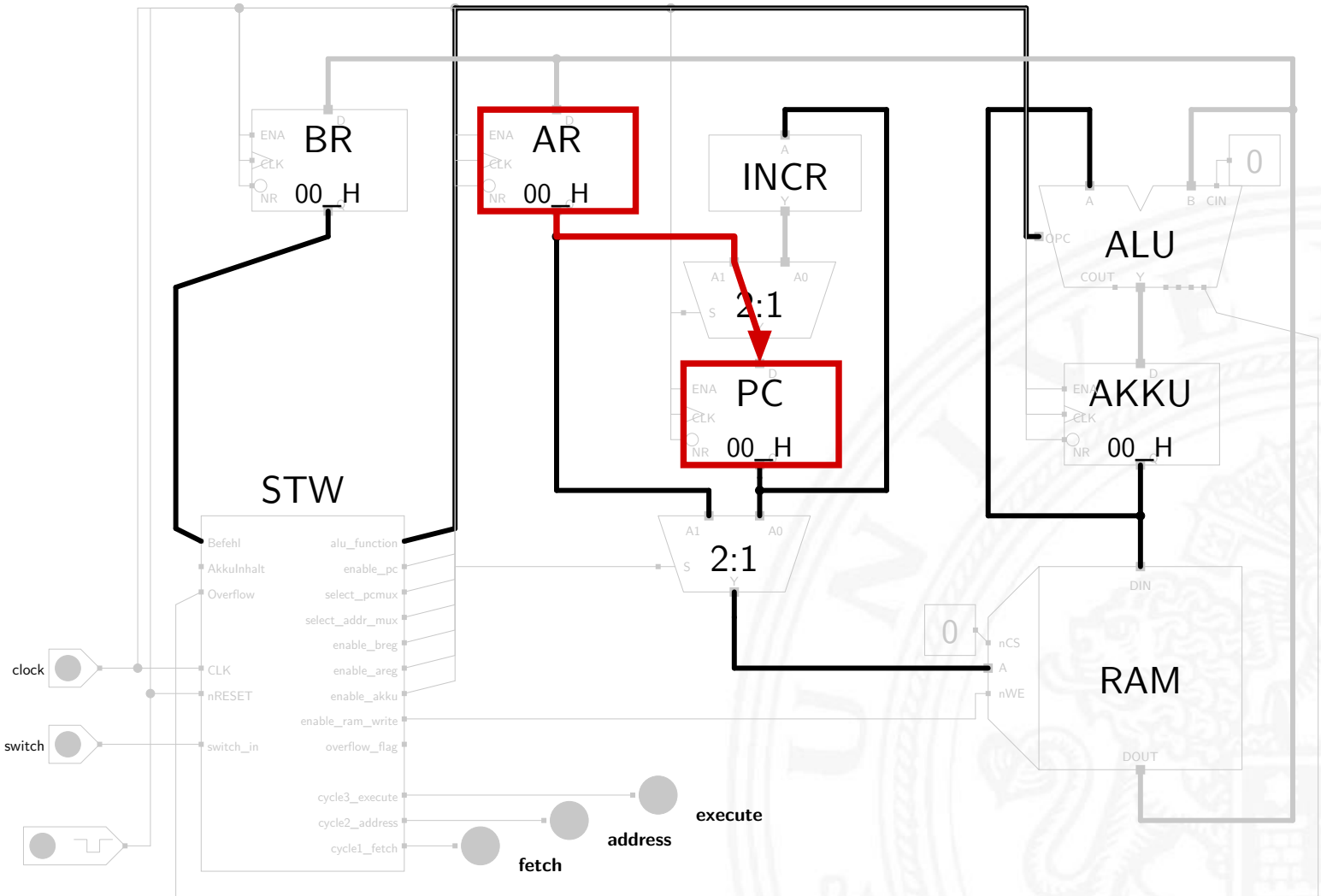
Akku = Akku + RAM[AR]



PRIMA: speichern
RAM[AR] = Akku



PRIMA: springen
PC = AR



später dazu mehr. . .

Timeline: Vorgeschichte

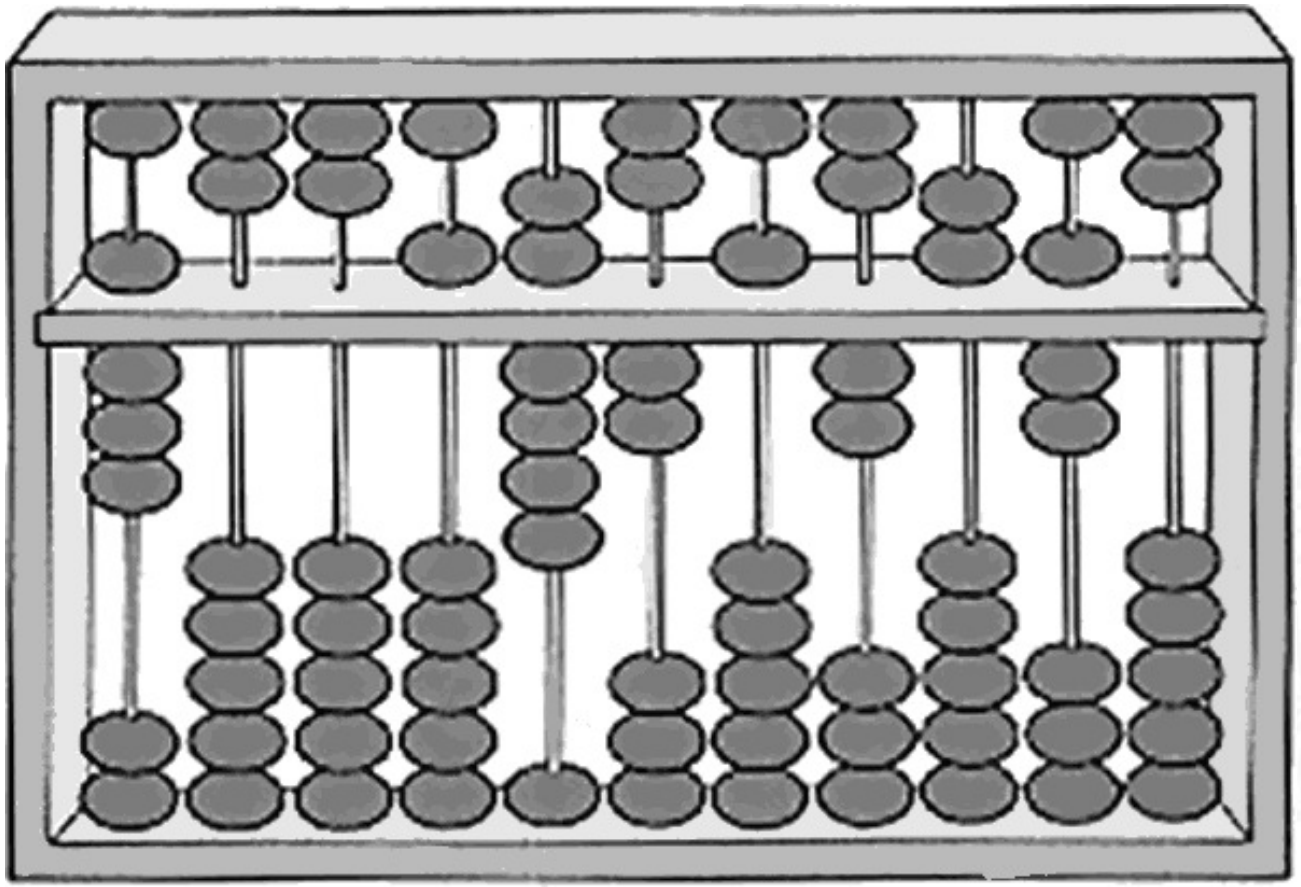
????	Abakus als erste Rechenhilfe
1642	Pascal: Addierer/Subtrahierer
1671	Leibniz: Vier-Operationen-Rechenmaschine
1837	Babbage: Analytical Engine
1937	Zuse: Z1 (mechanisch)
1939	Zuse: Z3 (Relais, Gleitkomma)
1941	Atanasoff & Berry: ABC (Röhren, Magnettrommel)
1944	Mc-Culloch Pitts (Neuronenmodell)
1946	Eckert & Mauchly: ENIAC (Röhren)
1949	Eckert, Mauchly, von Neumann: EDVAC (erster speicherprogrammierter Rechner)
1949	Manchester Mark-1 (Indexregister)

Abakus

Wert in Spalte	8	0	0	5	14	2	5	2	10	7	0

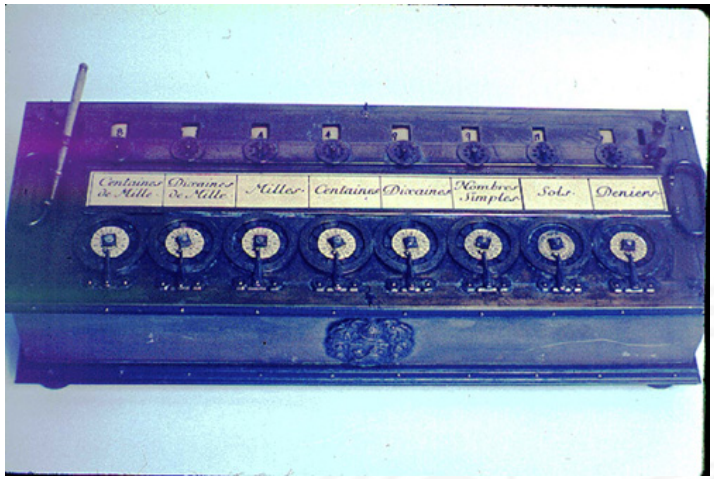
Kugel = 5

Kugel = 1



Zehnerpotenz der Spalte	10^{10}	10^{9}	10^{8}	10^{7}	10^{6}	10^{5}	10^{4}	10^{3}	10^{2}	10^{1}	10^{0}

Mechanische Rechenmaschinen



1623 Schickard: Sprossenrad, Addierer/Subtrahierer

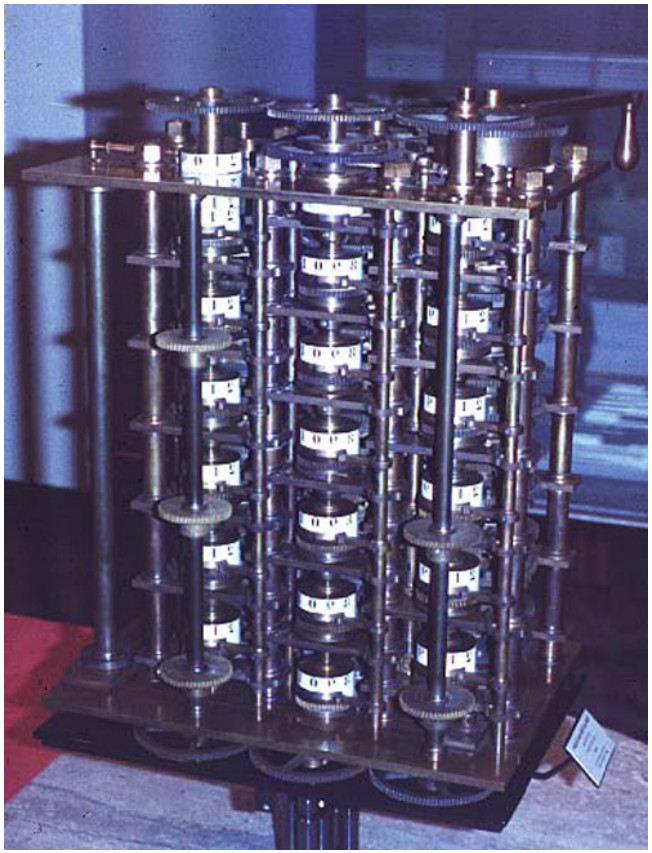
1642 Pascal: „Pascalene“

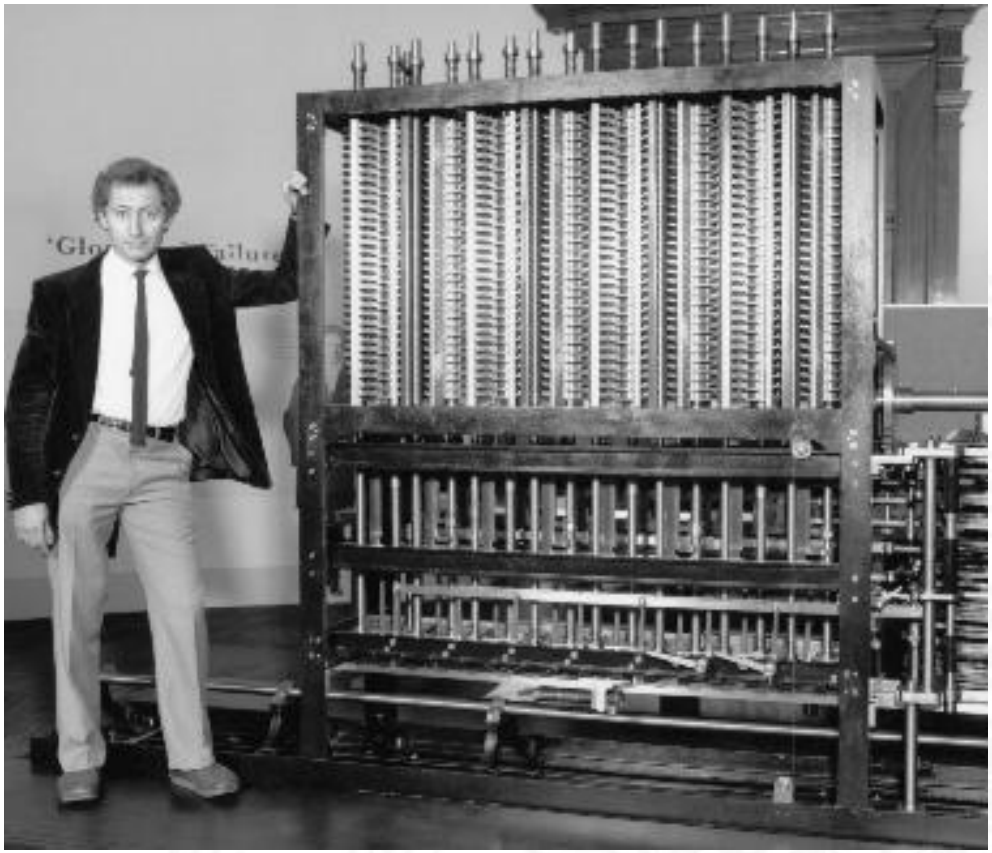
1673 Leibniz: Staffelwalze, Multiplikation/Division

1774 Philipp Matthäus Hahn: erste gebrauchsfähige „4-Spezies“-Maschine

Difference Engine

Charles Babbage 1822: Berechnung nautischer Tabellen

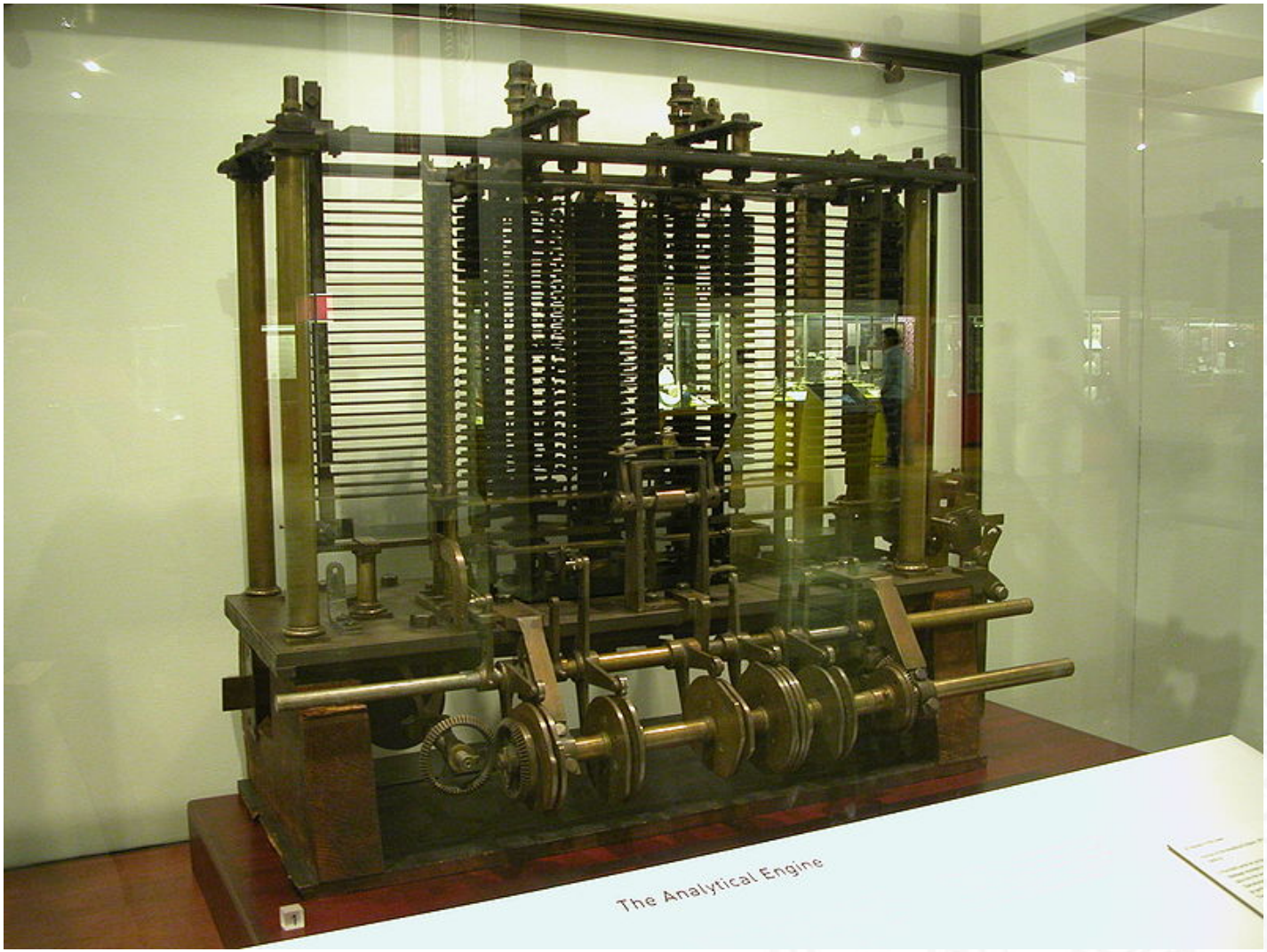




Original von 1832 und Nachbau von 1989, London Science Museum

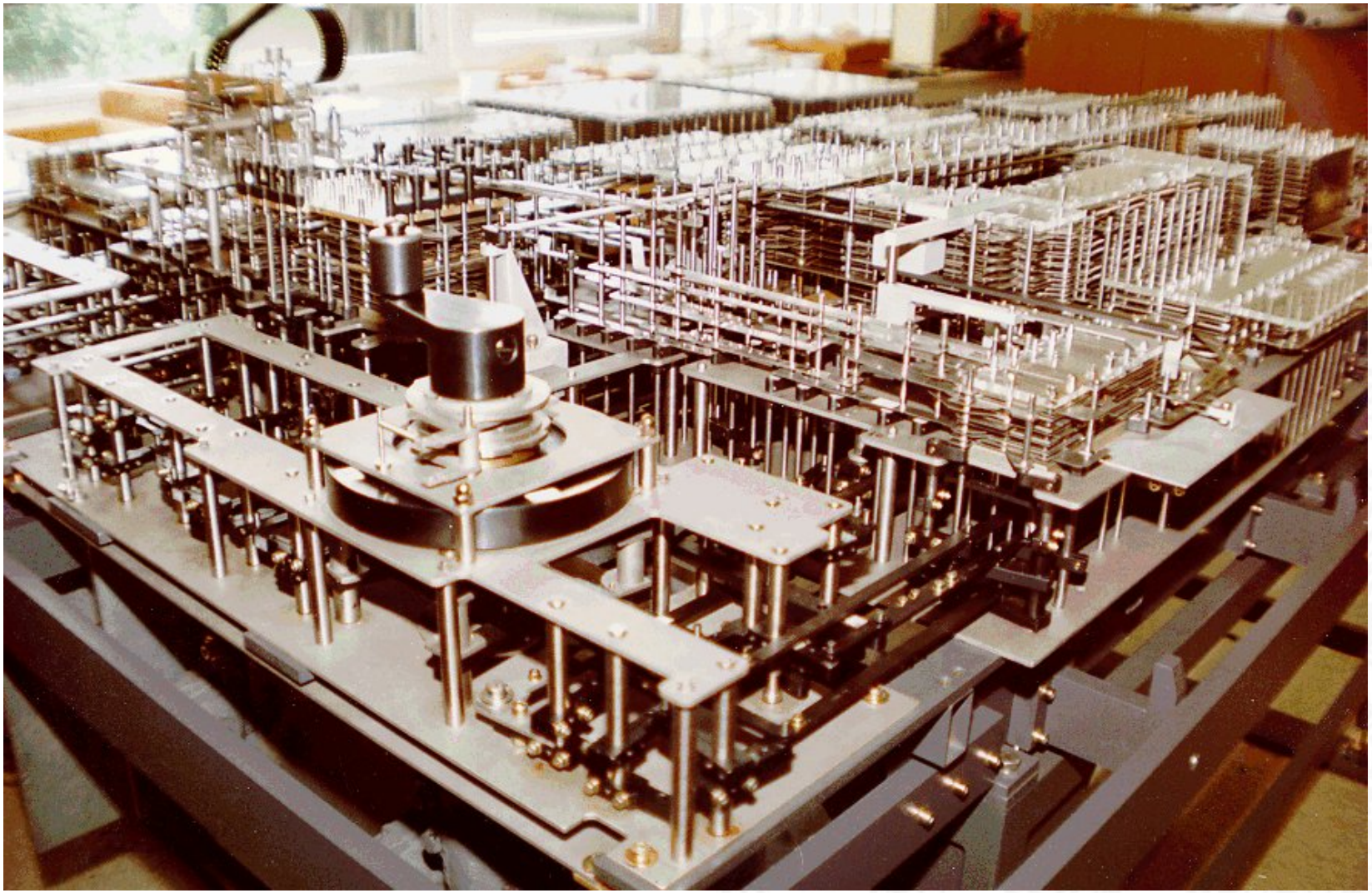
Analytical Engine

Charles Babbage 1837-1871: frei programmierbar, Lochkarten, unvollendet



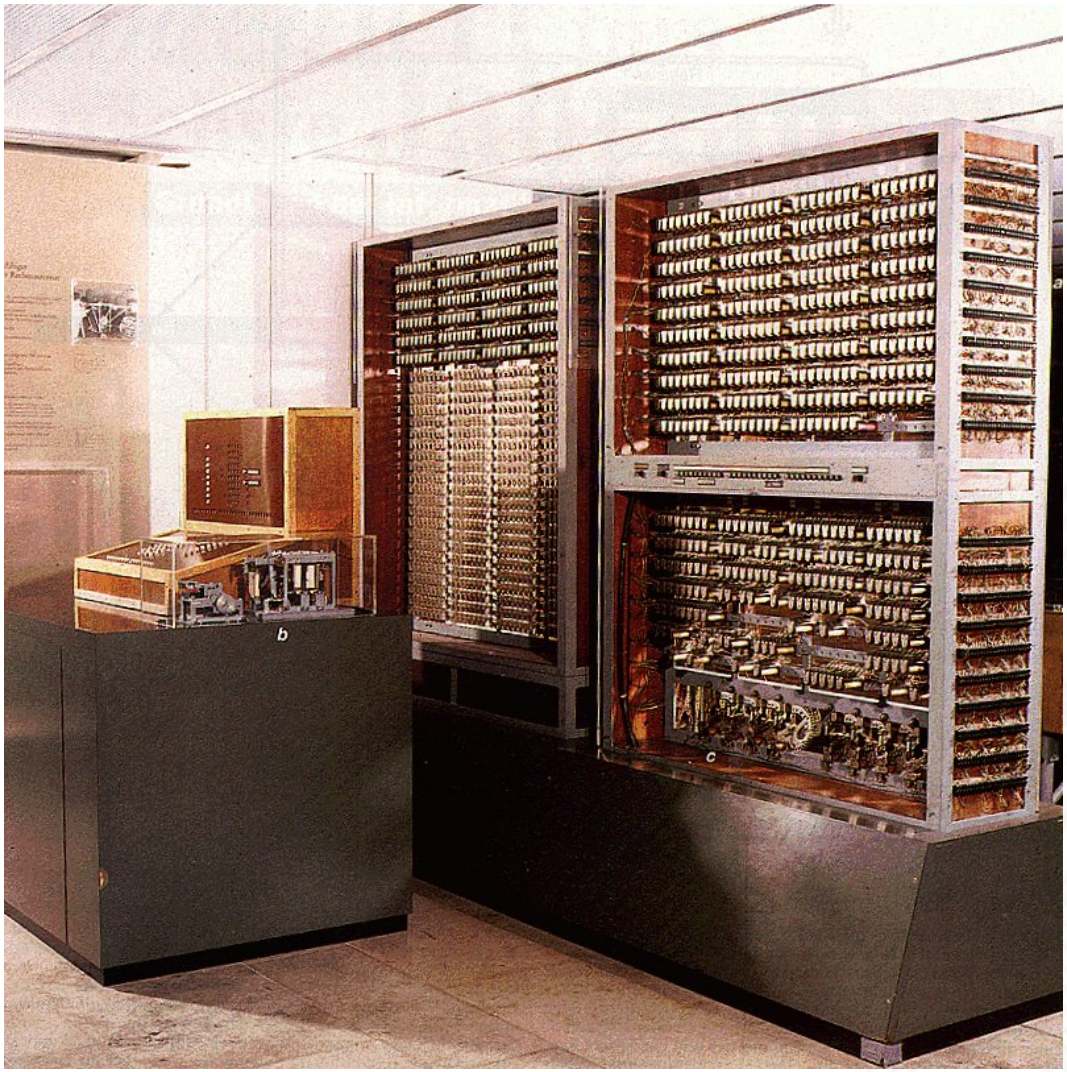
Zuse Z1

Konrad Zuse 1937: 64 Register, 22-bit, mechanisch, Lochfilm



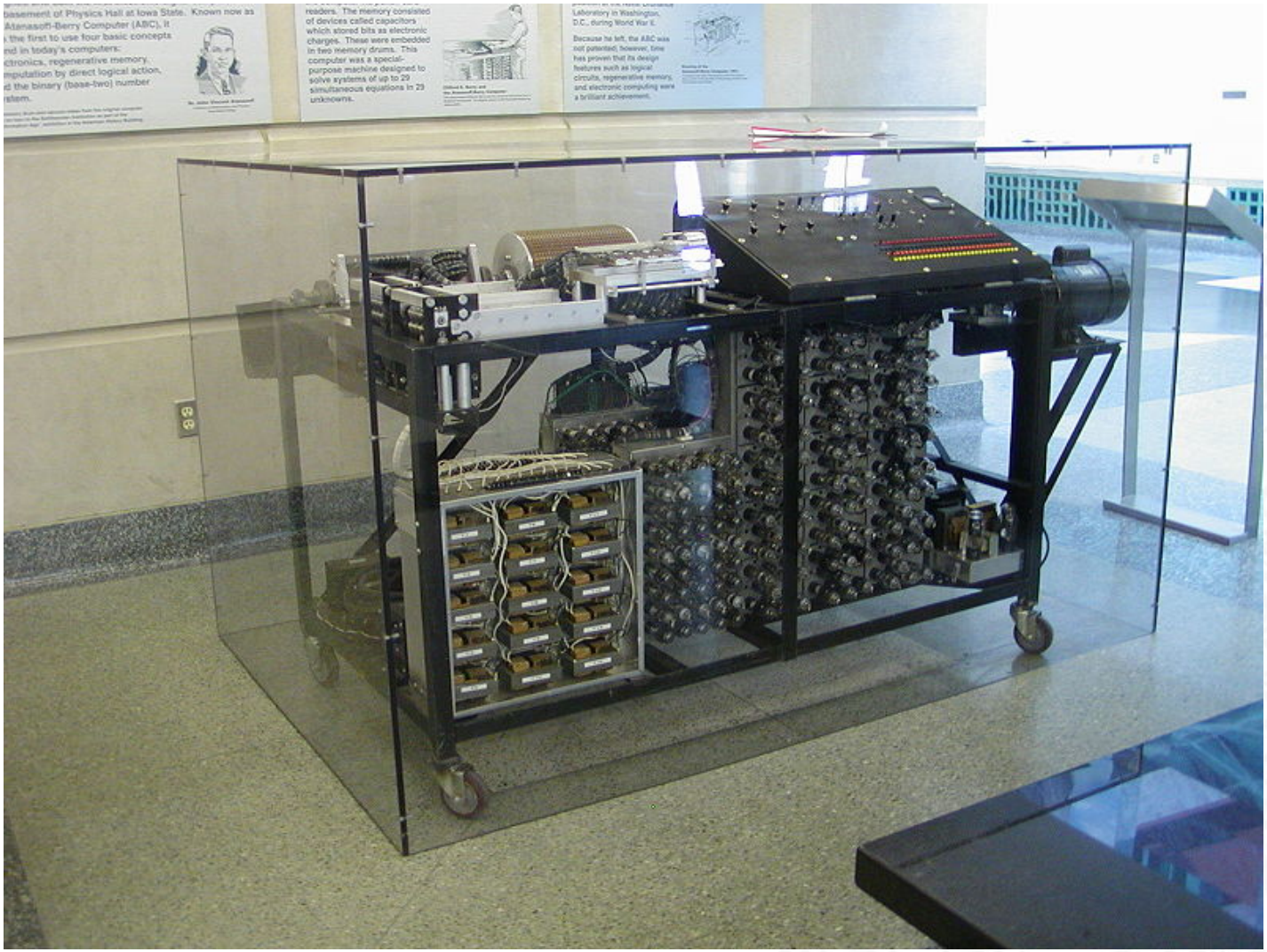
Zuse Z3

Konrad Zuse 1941, 64 Register, 22-bit, 2000 Relays, Lochfilm



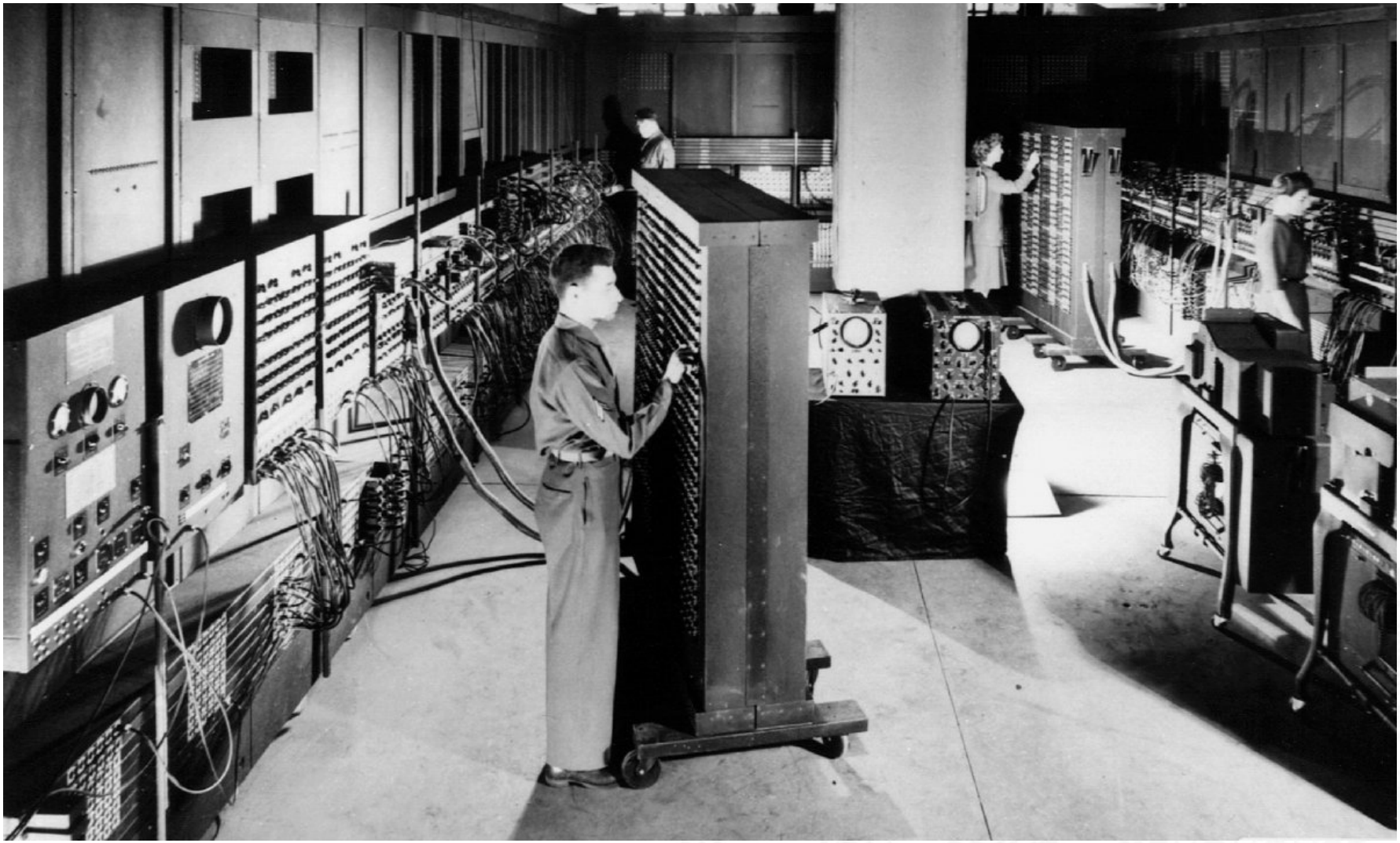
Atanasoff-Berry Computer (ABC)

J.V.Atanasoff 1942: 50-bit Festkomma, Röhren und Trommelspeicher, fest programmiert

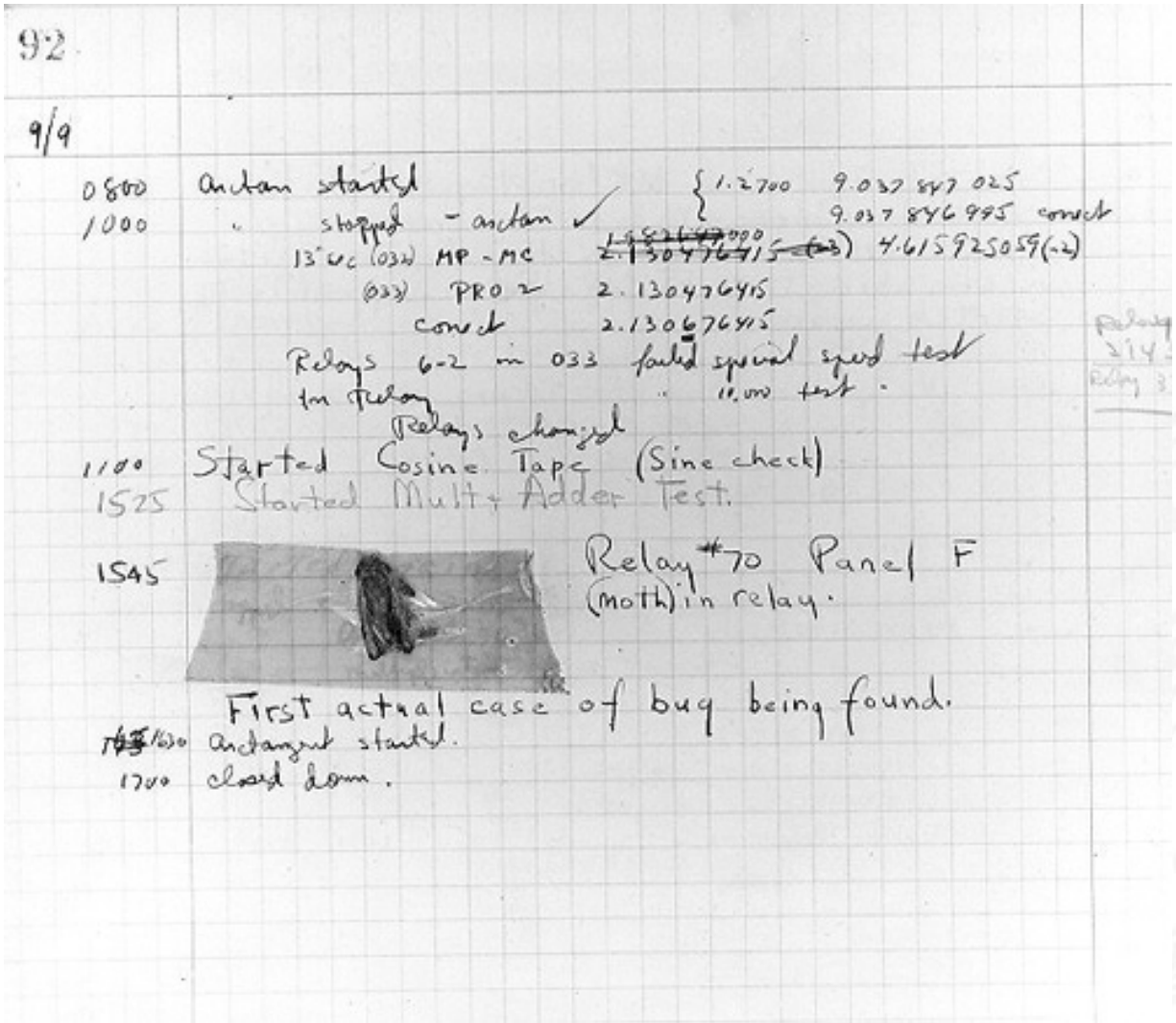


ENIAC – Electronic Numerical Integrator and Computer

Mauchly & Eckert, 1946: Röhren, Steckbrett-Programm



First computer bug



92

9/9

0800 Arctan started

1000 " stopped - arctan ✓

13 u/c (032) MP - MC

(033) PRO 2

correct

Relays 6-2 in 033 failed special speed test

In relay

Relays changed

1.2700 9.037 847 025

9.037 846 995 correct

4.615925059(-2)

2.130476415

2.130676415

1100 Started Cosine Tape (Sine check)

1525 Started Multy Adder Test.

1545 Relay #70 Panel F

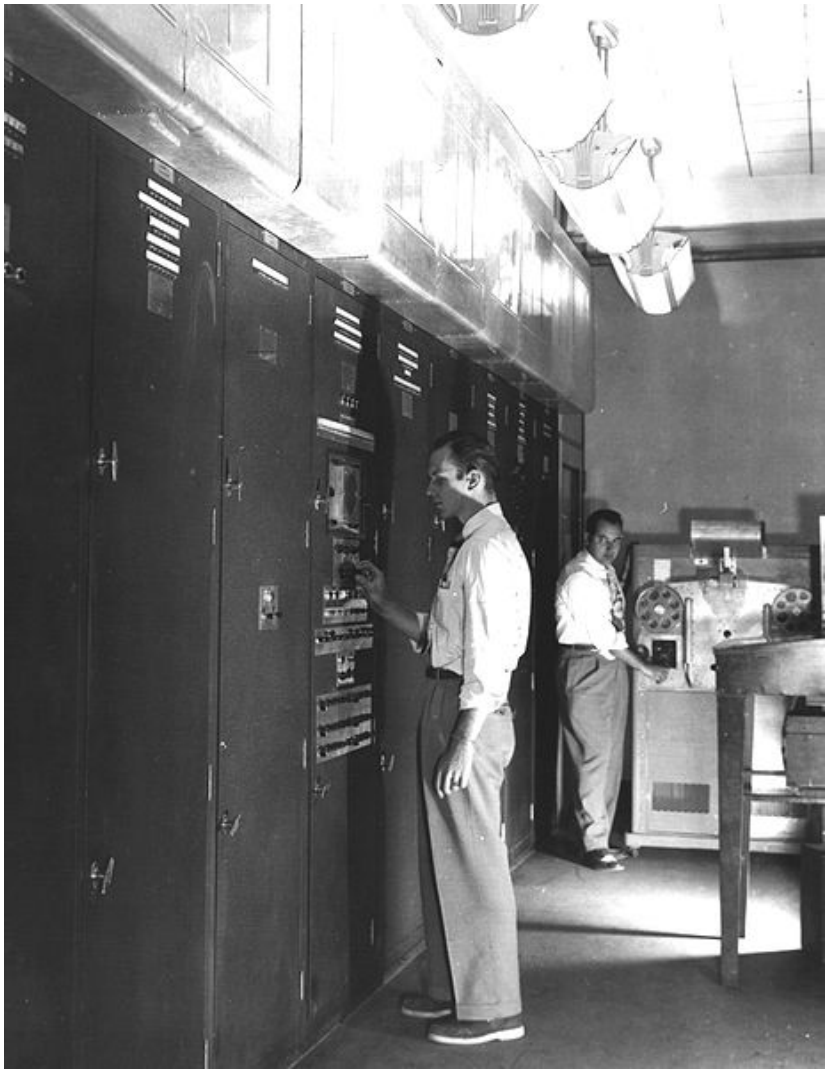
(moth) in relay.

First actual case of bug being found.

1700 closed down.

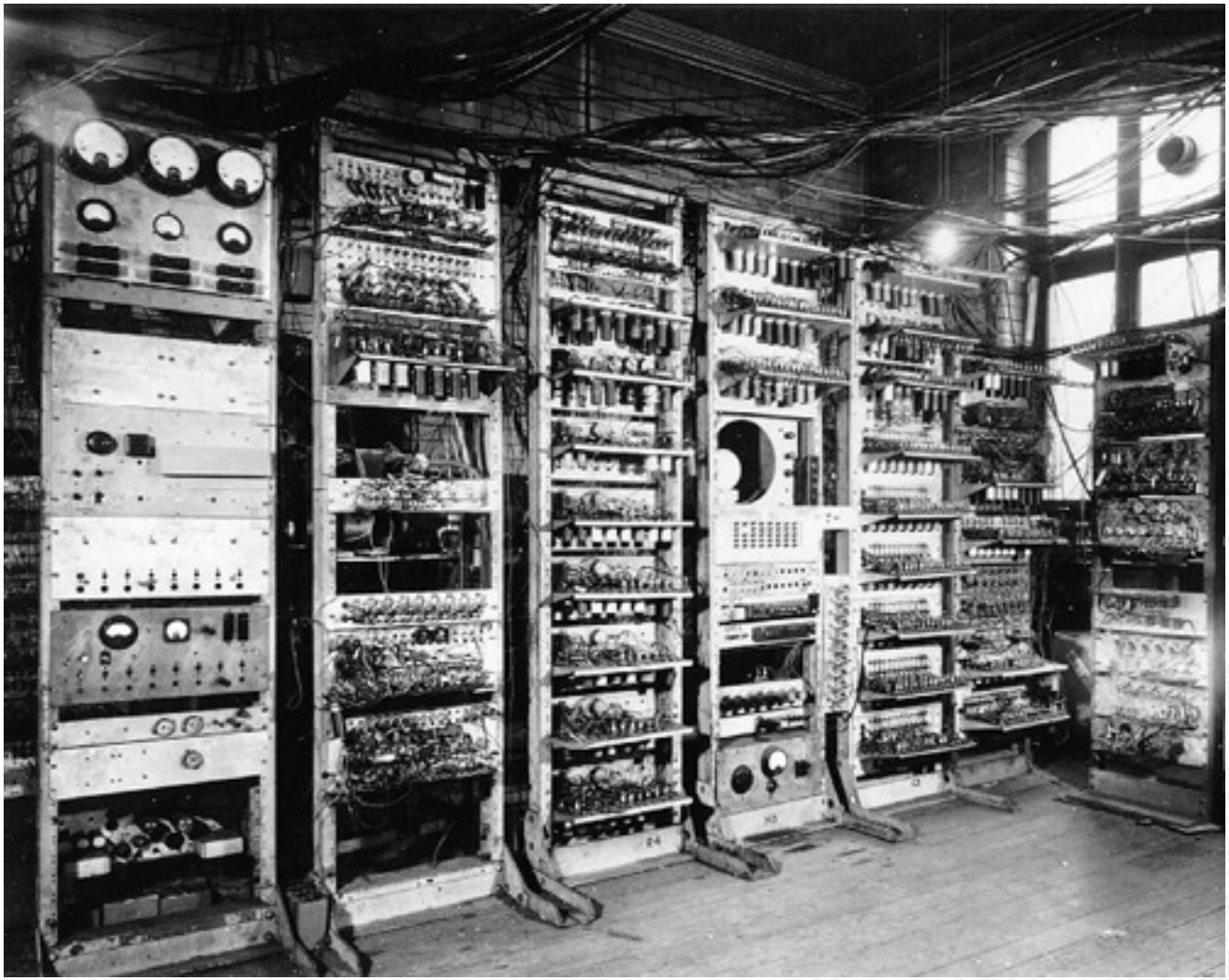
EDVAC

Mauchly, Eckert & von Neumann, 1949: Röhren, speicherprogrammiert



Manchester Mark-1

Williams & Kilburn, 1949: Trommelspeicher, Indexregister



Manchester EDSAC

Wilkes 1951: Mikroprogrammierung, Unterprogramme, speicherprogrammiert

erste Computer, ca. 1950

- zunächst noch kaum Softwareunterstützung
- nur zwei Schichten:

1. Programmierung in elementarer Maschinensprache (ISA level)
2. Hardware in Röhrentechnik (device logic level)
   - Hardware kompliziert und unzuverlässig

Mikroprogrammierung (Maurice Wilkes, Cambridge, 1951):

- Programmierung in komfortabler Maschinensprache
- Mikroprogramm-Steuerwerk (Interpreter)
- einfache, zuverlässigere Hardware
- Grundidee der sog. **CISC**-Rechner (68000, 8086, VAX)

erste Betriebssysteme

- erste Rechner jeweils nur von einer Person benutzt
- Anwender = Programmierer = Operator
- Programm laden, ausführen, Fehler suchen usw.

$\Rightarrow$ Maschine wird nicht gut ausgelastet

$\Rightarrow$ Anwender mit lästigen Details überfordert

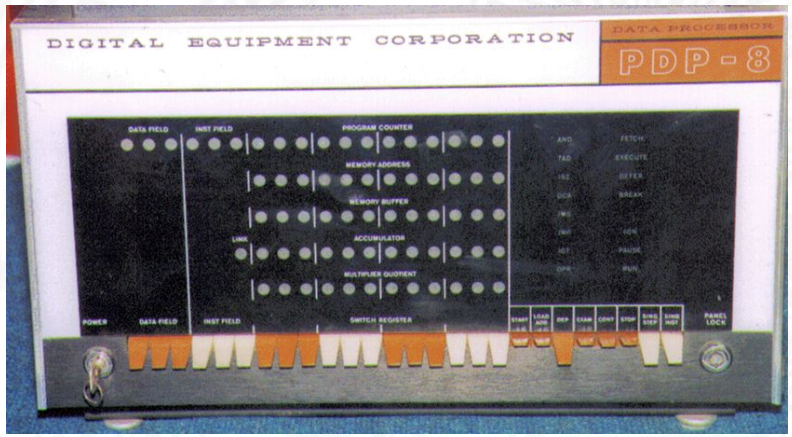
Einführung von **Betriebssystemen**

- „system calls“
- Batch-Modus: Programm abschicken, warten
- Resultate am nächsten Tag abholen

zweite Generation: Transistoren

- Erfindung des Transistors 1948 J. Bardeen, W. Brattain, W. Shockley
- schneller, zuverlässiger, sparsamer als Röhren
- Miniaturisierung und dramatische Kostensenkung

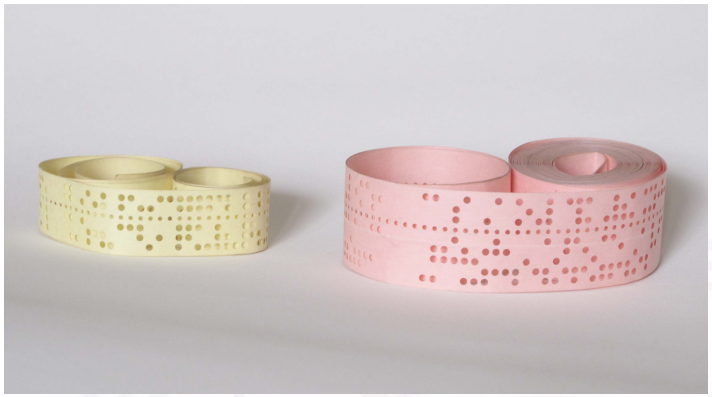
- Beispiel Digital Equipment Corporation PDP-1 (1961)
  - 4Ki Speicher (4096 Worte á 18-bit)
  - 200 KHz Taktfrequenz
  - 120 000 $
  - Grafikdisplay: erste Computerspiele
- Nachfolger PDP-8: 16 000 $
  - erstes Bussystem
  - 50 000 Stück verkauft



Festplatten

Massenspeicher bei frühen Computern

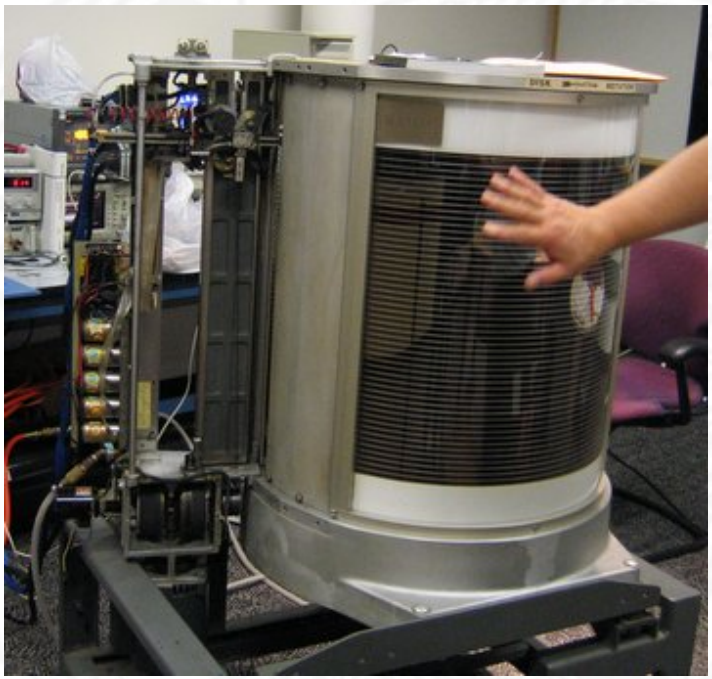
- Lochkarten
- Lochstreifen
- Magnetband



- Magnettrommel
- Festplatte

  IBM 350 RAMAC (1956)

  5 MByte, 600 ms Zugriffszeit



https://de.wikibooks.org/wiki/Computerhardware_für_Anfänger

dritte Generation: ICs

- Erfindung der integrierten Schaltung 1958 (Noyce, Kilby)
- Dutzende. . . Hunderte. . . Tausende Transistoren auf einem Chip

- IBM Serie-360: viele Maschinen, ein einheitlicher Befehlssatz
- volle Softwarekompatibilität

Eigenschaft	**Model 30**	**Model 40**	**Model 50**	**Model 65**
Rel. Leistung [Model 30]	1	3,5	10	21
Zykluszeit [ns]	1 000	625	500	250
Max. Speicher [KiB]	64	256	256	512
Pro Zyklus gelesene Byte	1	2	4	16
Max. Anzahl von Datenkanälen	3	3	4	6

vierte Generation: VLSI

- VLSI = *Very Large Scale Integration*
- ab 10 000 Transistoren pro Chip

- gesamter Prozessor passt auf einen Chip
- steigende Integrationsdichte erlaubt immer mehr Funktionen

1972 Intel 4004: erster Mikroprozessor

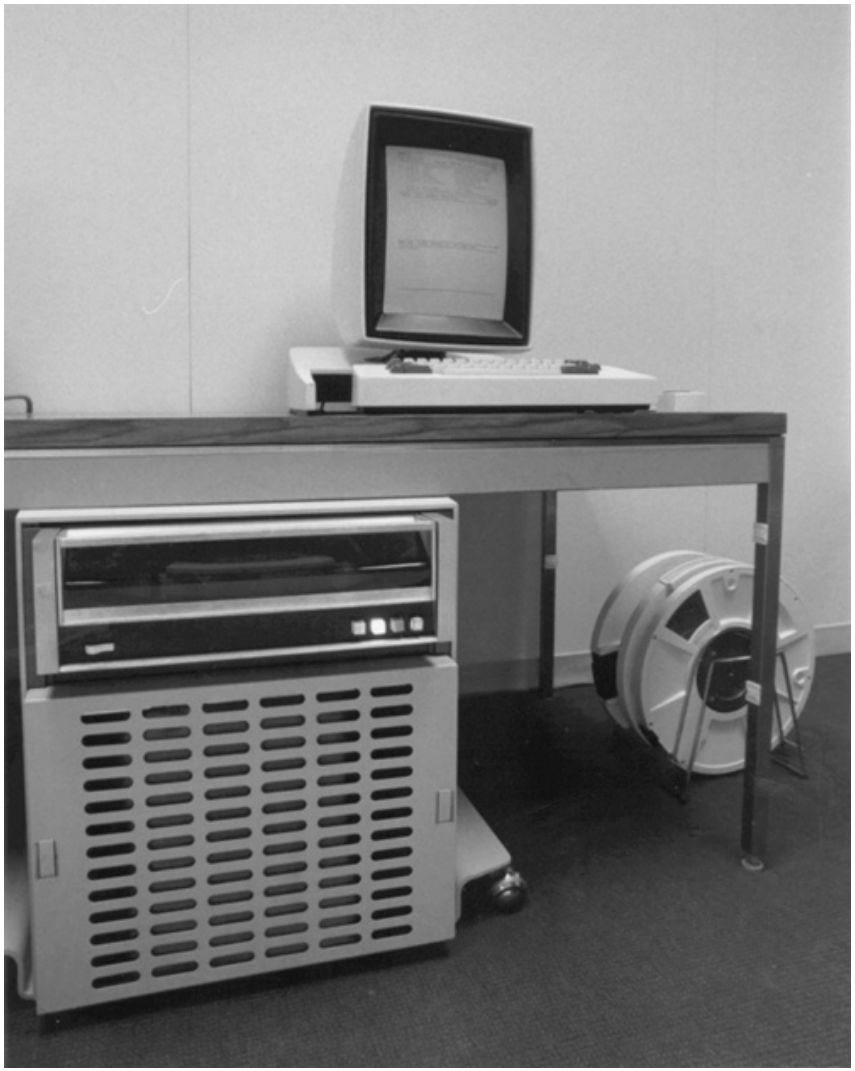
1975 Intel 8080, Motorola 6800, MOS 6502 ...

1981 IBM PC („personal computer“) mit Intel 8088

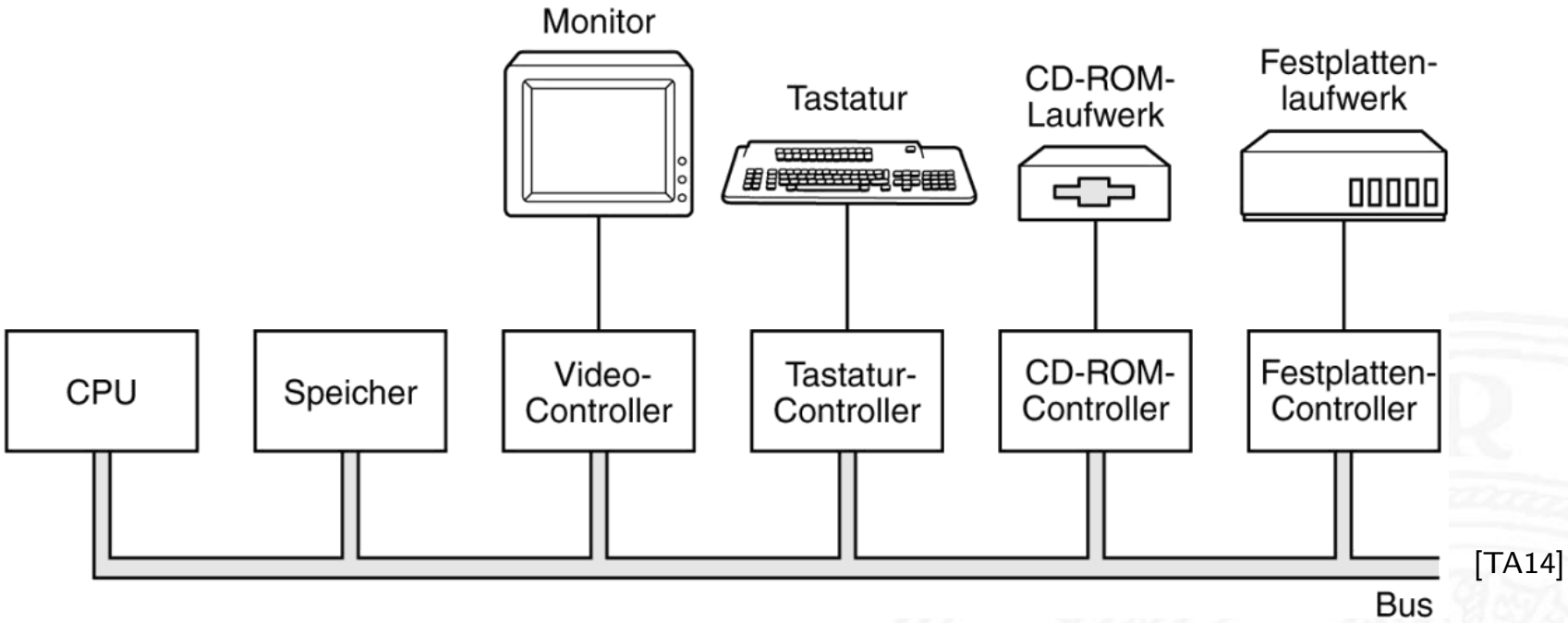
...

- Massenfertigung erlaubt billige Prozessoren (< 1\$)
- Miniaturisierung ermöglicht mobile Geräte

Xerox Alto: first workstation



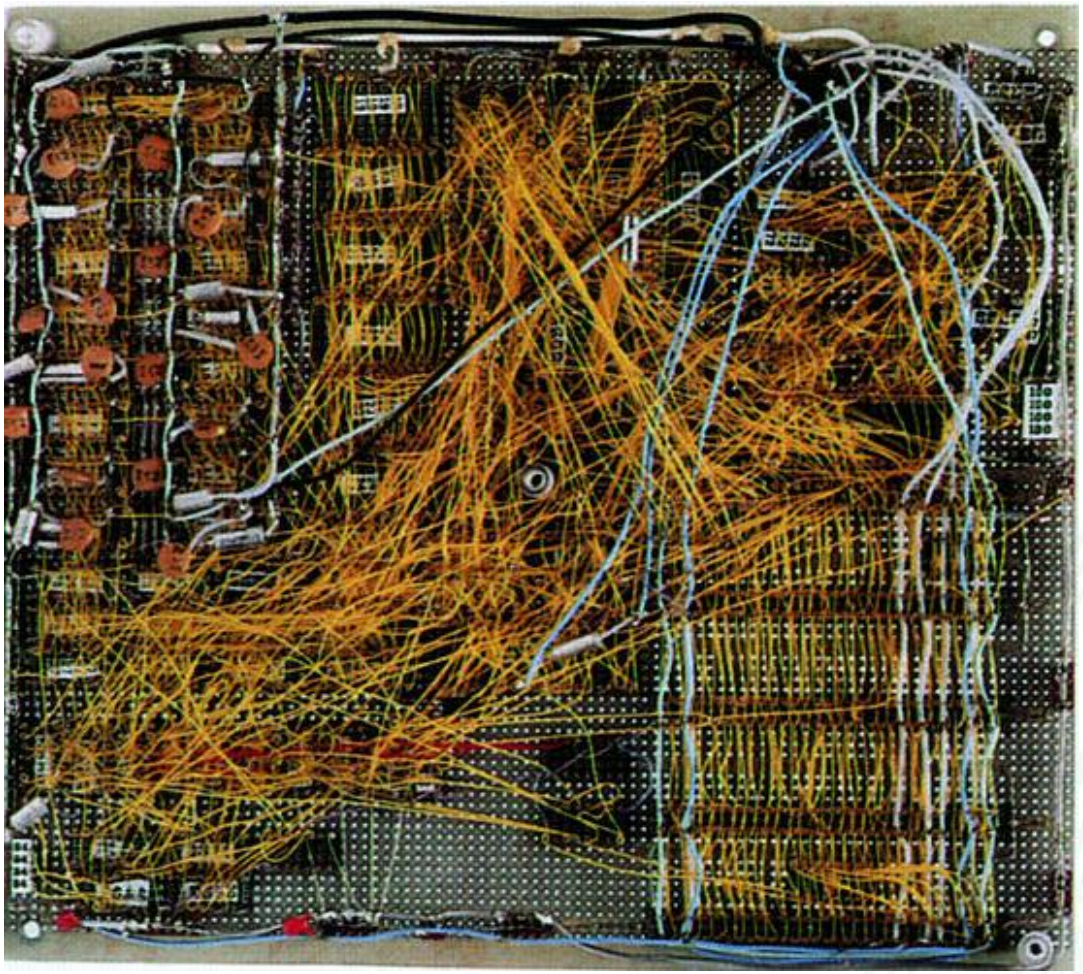
Personal Computer: Aufbau des IBM PC (1981)

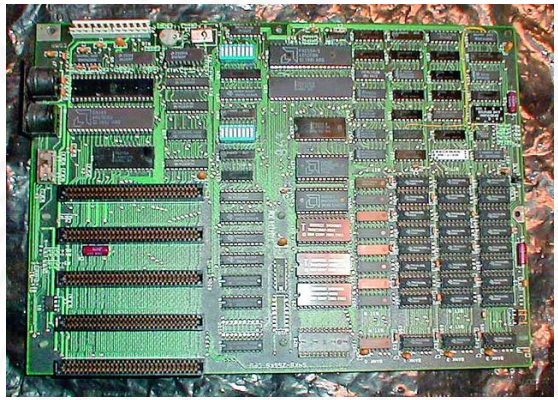


[TA14]

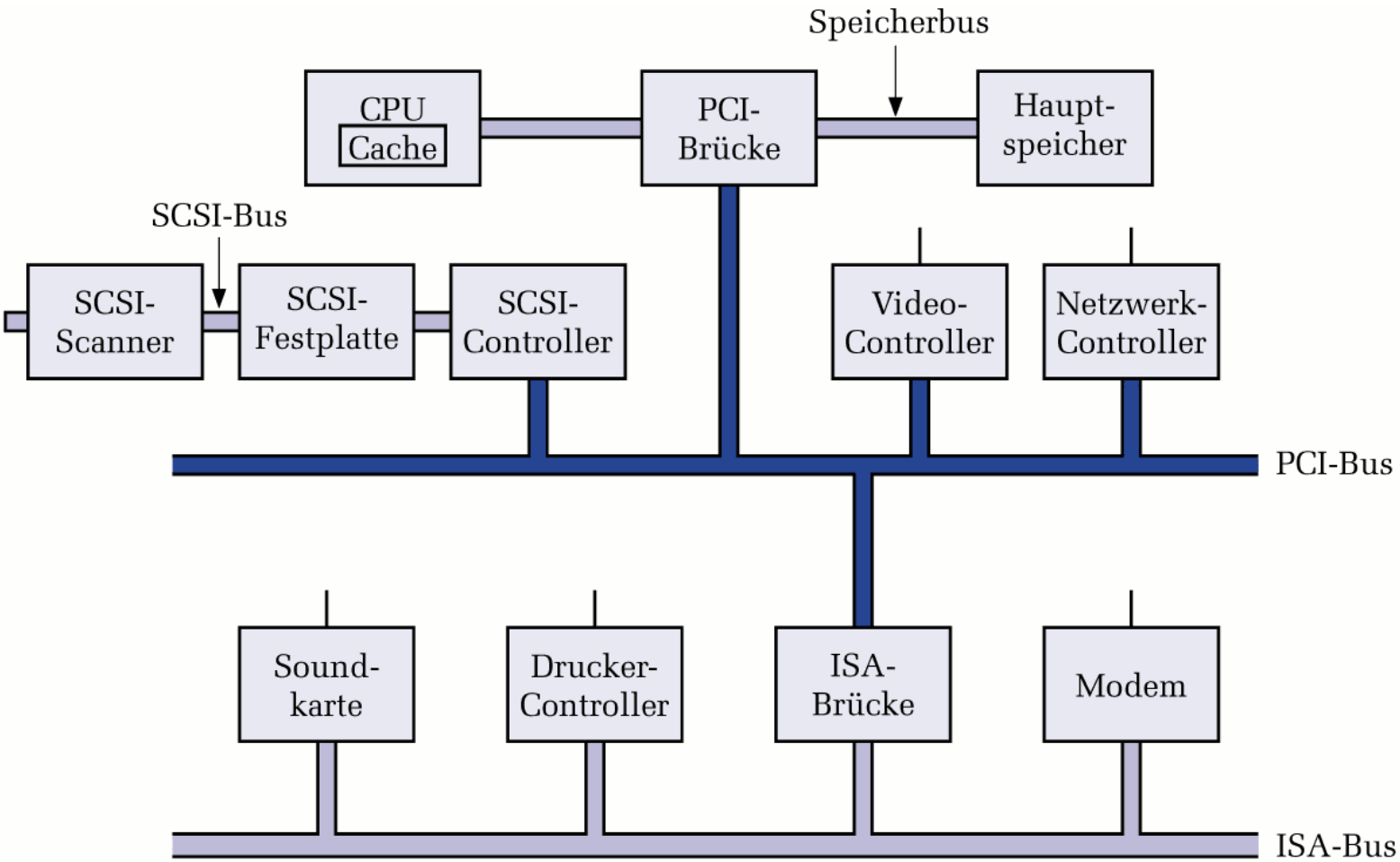
- Intel 8086/8088, 512 KByte RAM, Betriebssystem MS-DOS
- alle Komponenten über den zentralen (ISA-) Bus verbunden
- Erweiterung über Einsteckkarten

PC Prototyp (1981) und Hauptplatine



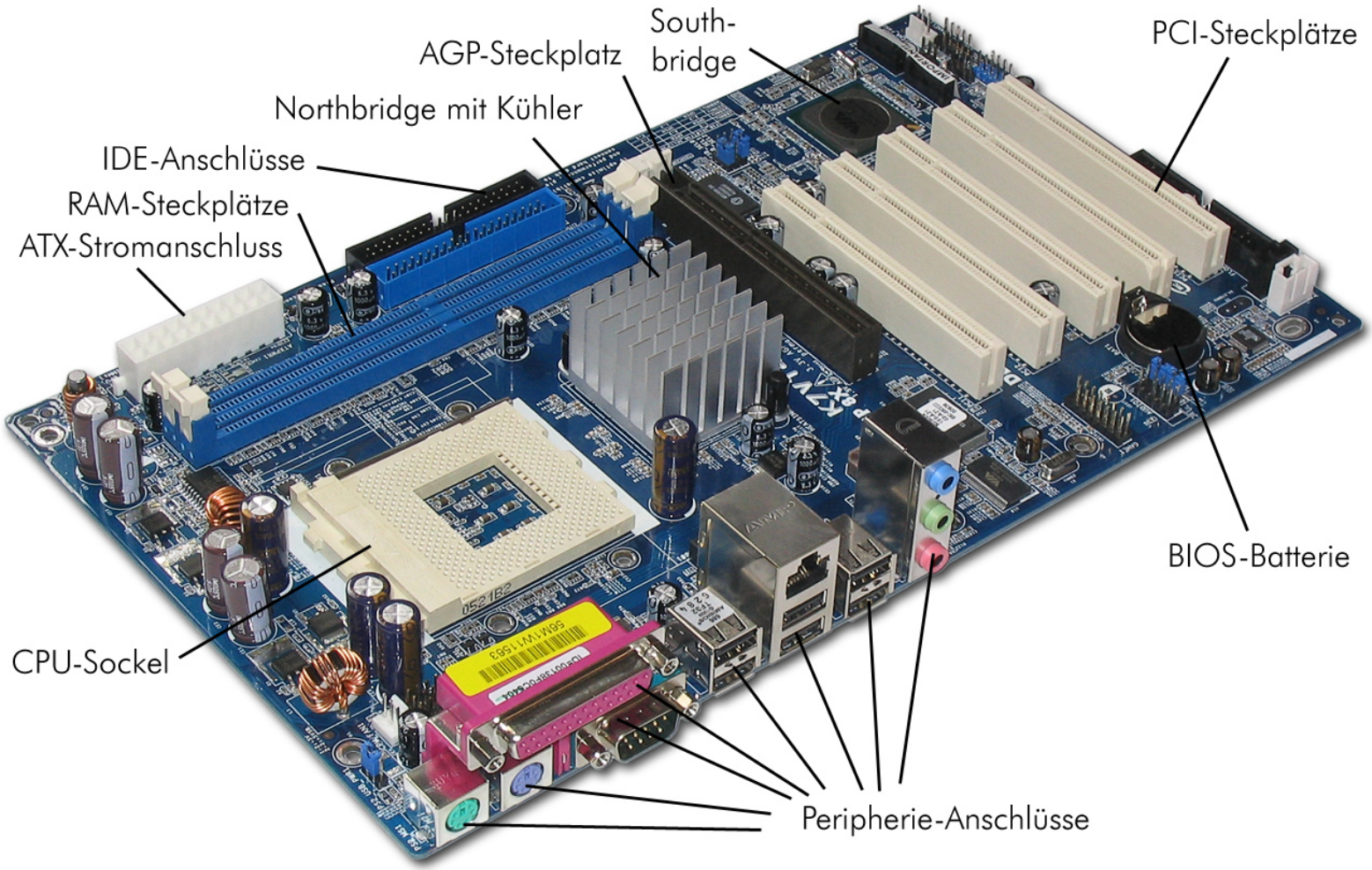


Aufbau mit PCI-Bus (2000)



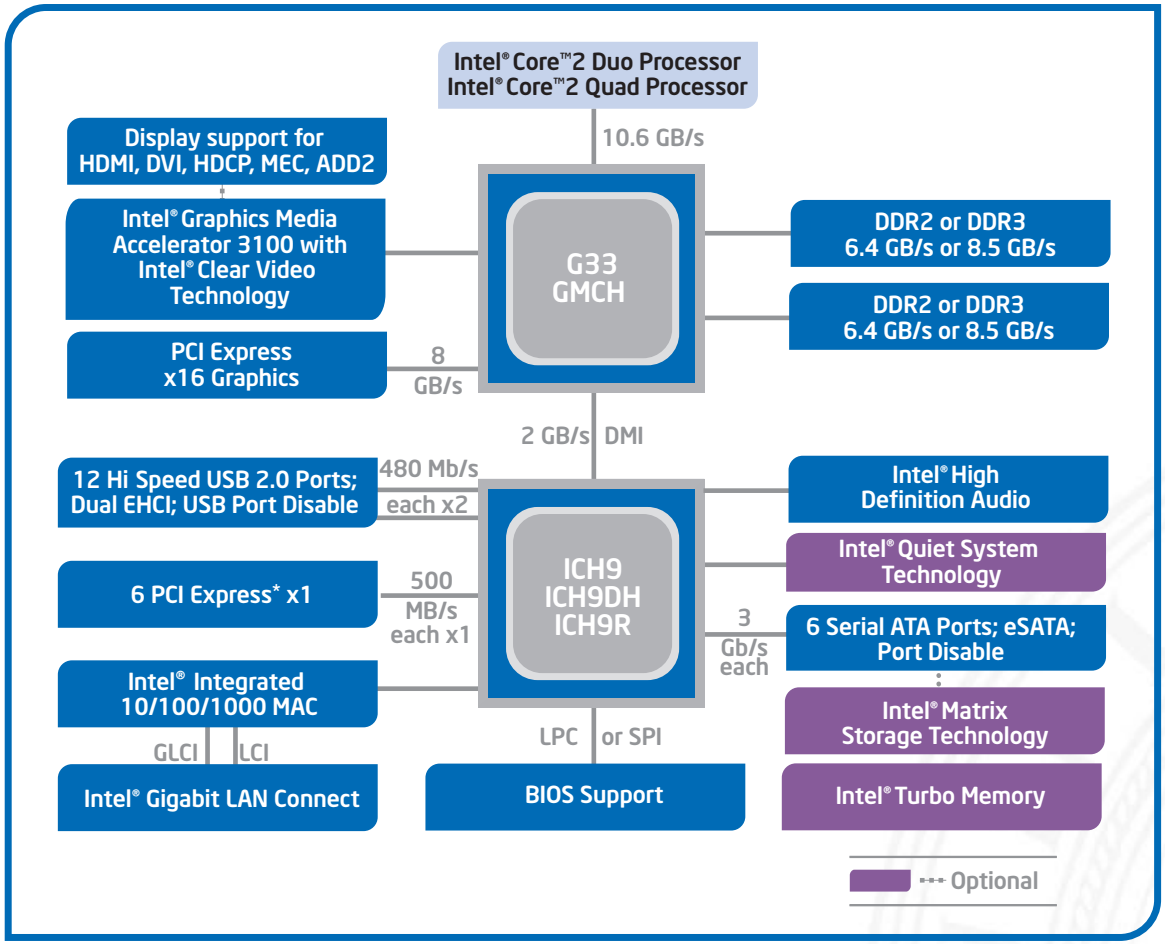
[TA14]

Hauptplatine (2005)



de.wikibooks.org/wiki/Computerhardware_für_Anfänger

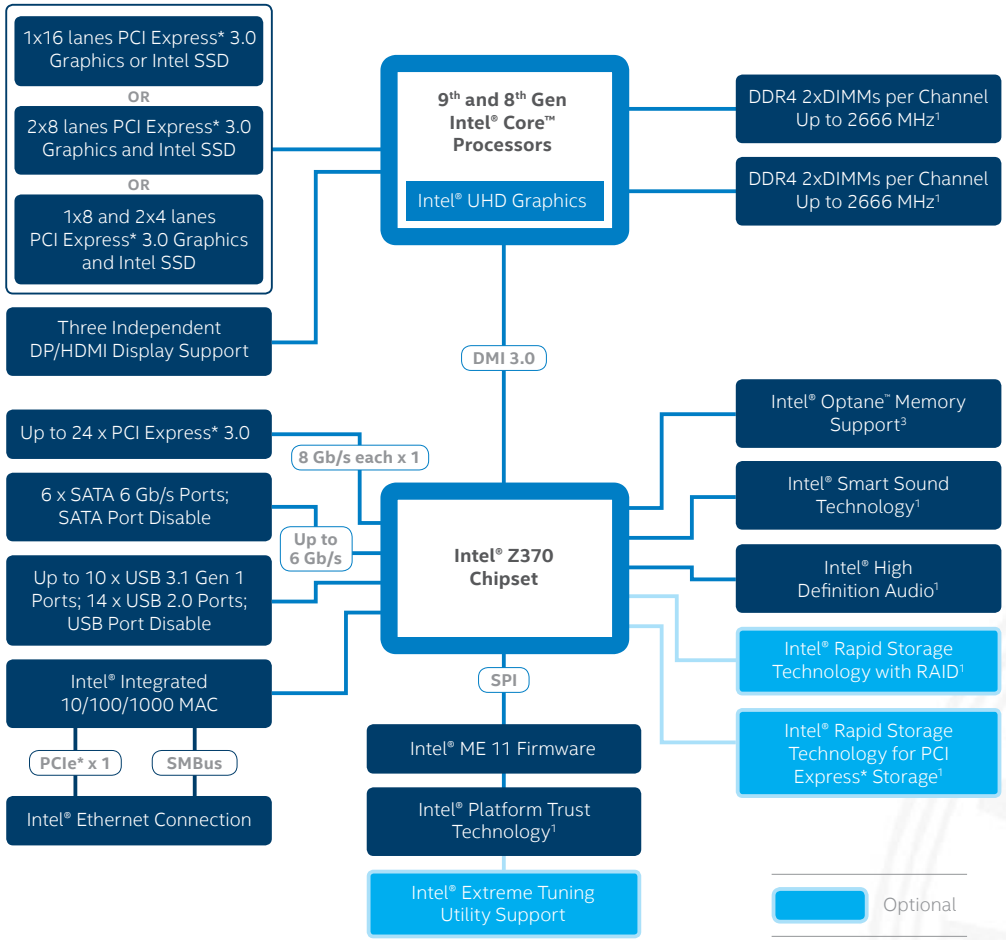
Aufbau (2010)



Intel `ark.intel.com`

- Mehrkern-Prozessoren („dual-/quad-/octa-core“)
- schnelle serielle Direktverbindungen statt PCI/ISA Bus

Aufbau (2017)



Intel `ark.intel.com`

- Speichercontroller und externe Anbindung (PCI Express) in CPU
- Grafikprozessor in CPU

Rechner-Spektrum

- Anzahl an Systemen / Prozessoren – weltweit

System	Anzahl (geschätzt!)
PCs, Workstation, Server	2 Milliarden
Tablets	1,3 Milliarden
Smartphones	4,8 Milliarden
„Embedded Systems“	75-100 Milliarden

- Preis des Prozessors

Typ	Preis [$]	Beispielanwendung
Wegwerfcomputer	0,5	Glückwunschkarten
Mikrocontroller	5	Uhren, Geräte, Autos
Mobile Computer und Spielkonsolen	50	Smartphones, Tablets, Heimvideospiele
Personalcomputer	500	Desktop- oder Notebook-Computer
Server	5 000	Netzwerkserver
Workstation Verbund	50 000 – 500 000	Abteilungsrechner (Minisupercomp.)
Großrechner (Mainframe)	5 Millionen	Batch-Verarbeitung in einer Bank
Supercomputer	$>$ 50 Millionen	Klimamodelle, Simulationen

Moore's Law

- bessere Technologie ermöglicht immer kleinere Transistoren
- Materialkosten sind proportional zur Chipfläche

$\Rightarrow$ bei gleicher Funktion kleinere und billigere Chips

$\Rightarrow$ bei gleicher Größe leistungsfähigere Chips

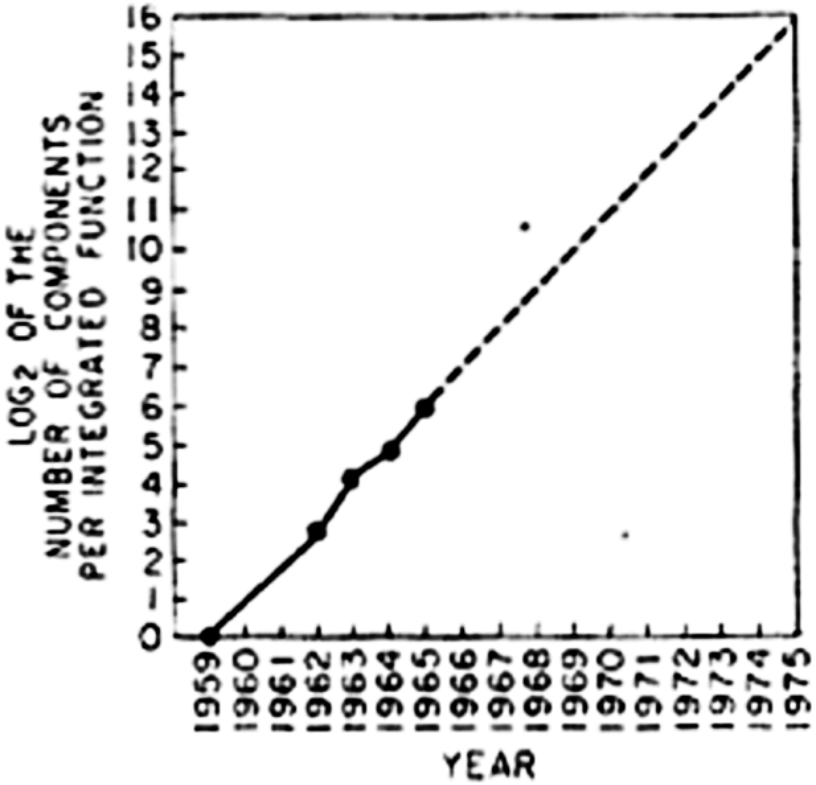
Moore's Law Gordon Moore, Mitgründer von Intel, 1965

Speicherkapazität von ICs vervierfacht sich alle drei Jahre

$\Rightarrow$ schnelles exponentielles Wachstum

- klares Kostenoptimum bei hoher Integrationsdichte
- trifft auch auf Prozessoren zu

Moore's Law (cont.)

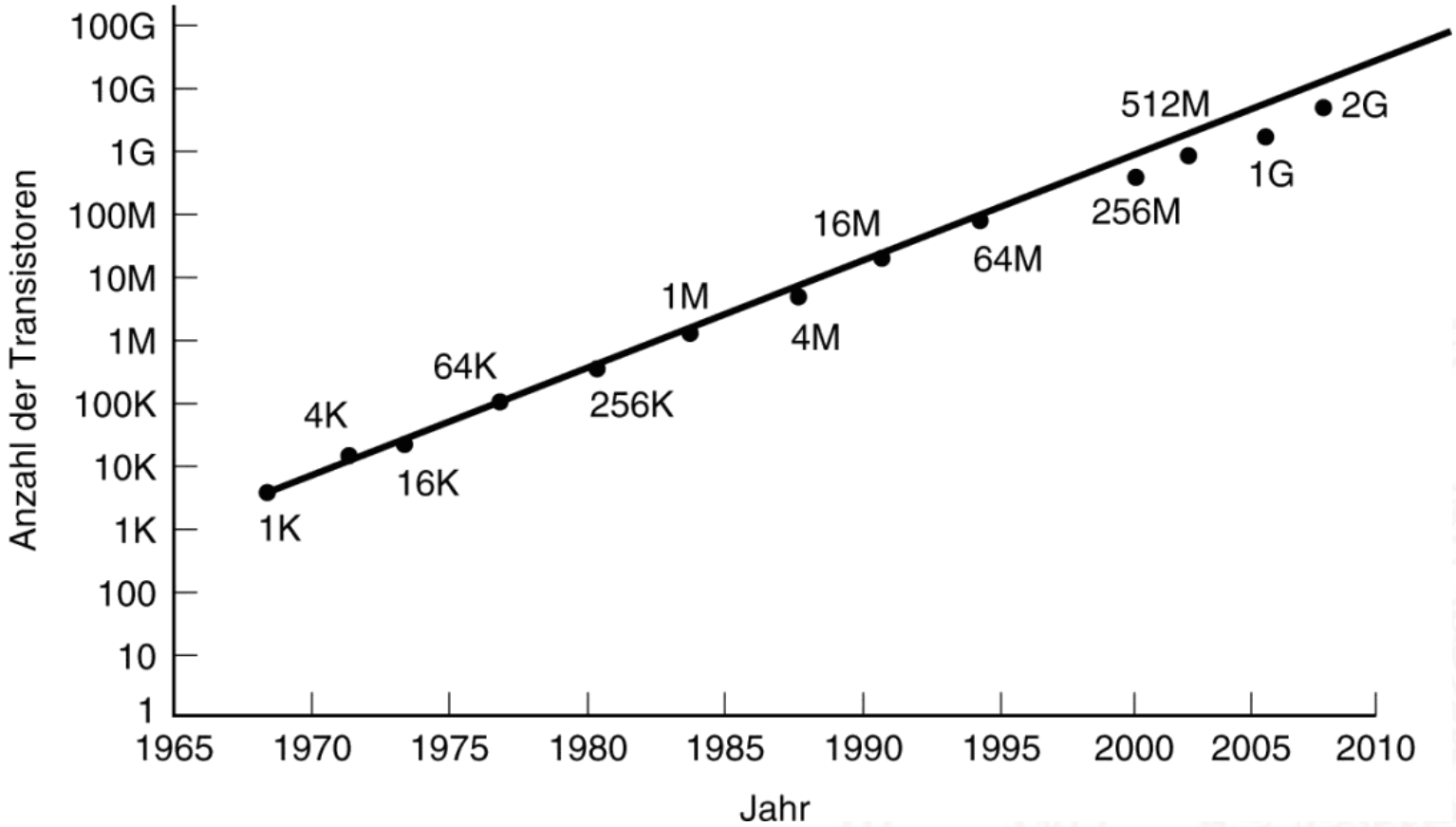


Gordon Moore, 1965, [Moo65]:
Cramming more components onto integrated circuits

Wird das so weitergehen?

- Vorhersage gilt immer noch
- „*IRDS*“ Prognosen bis zum Jahr 2034 [IRDS18]

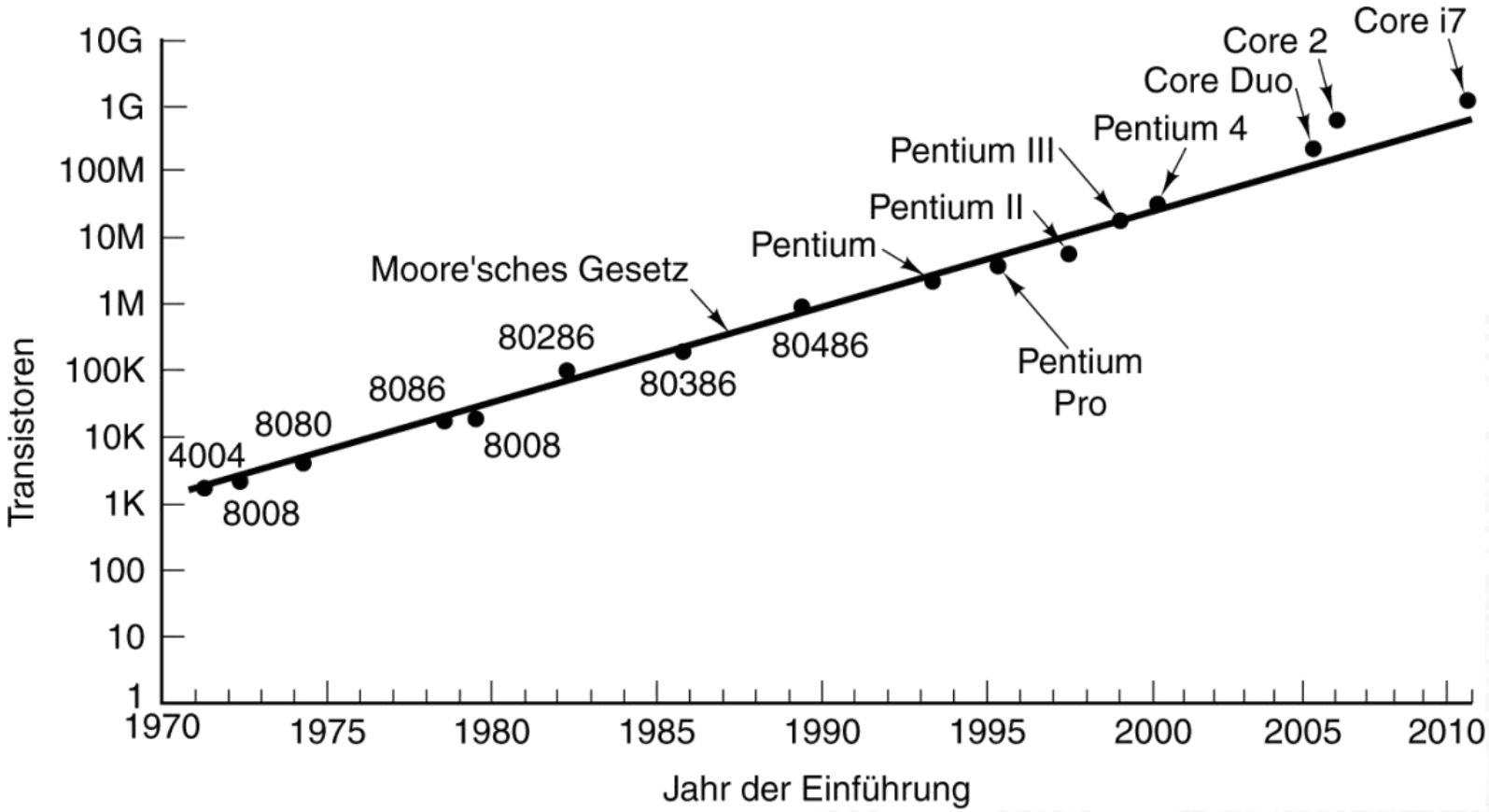
Moore's Law: Transistoren pro Speicherchip



[TA14]

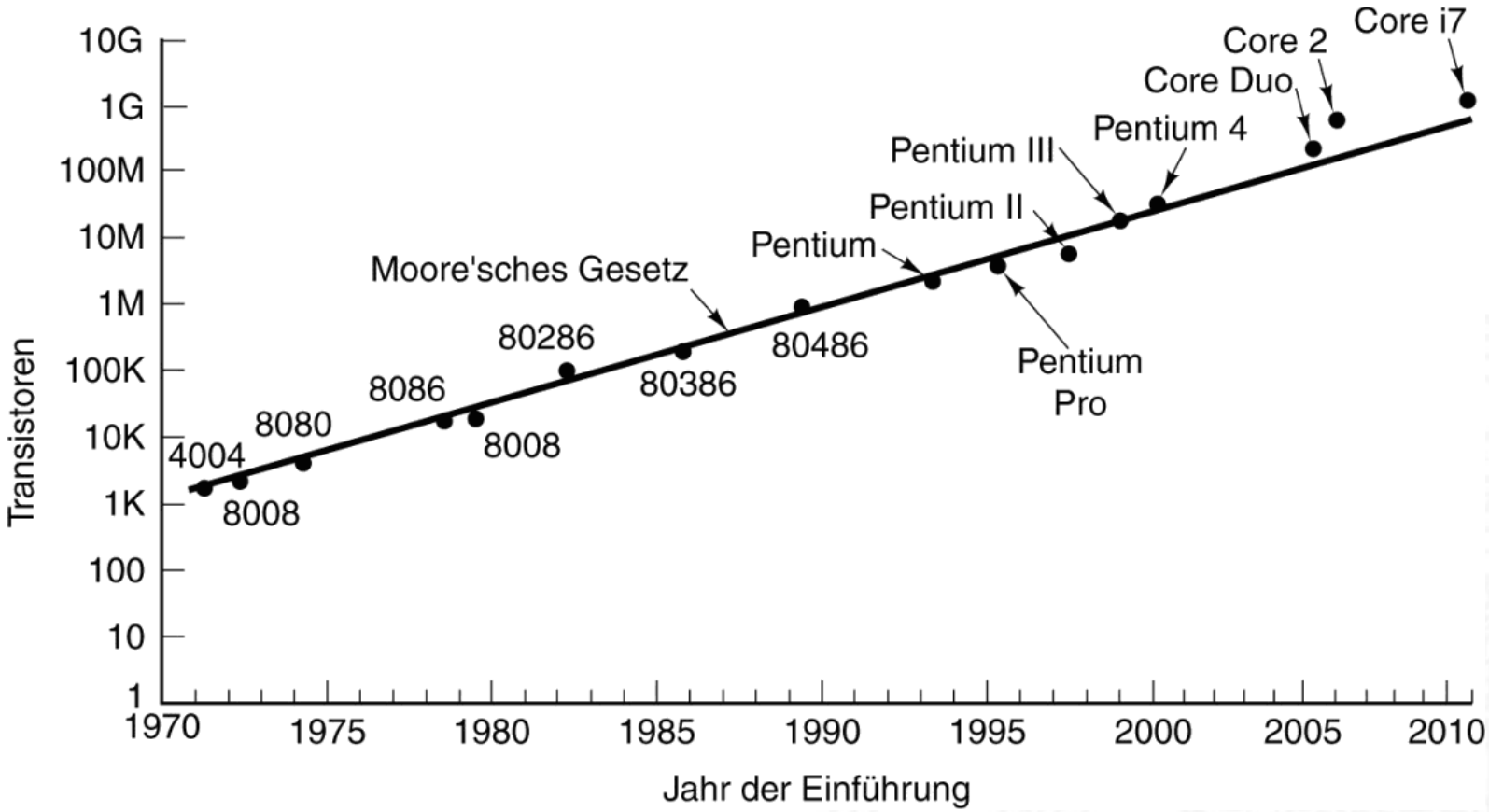
- Vorhersage: 60% jährliches Wachstum der Transistoranzahl pro IC

Moore's Law: Evolution der Prozessoren



[TA14]

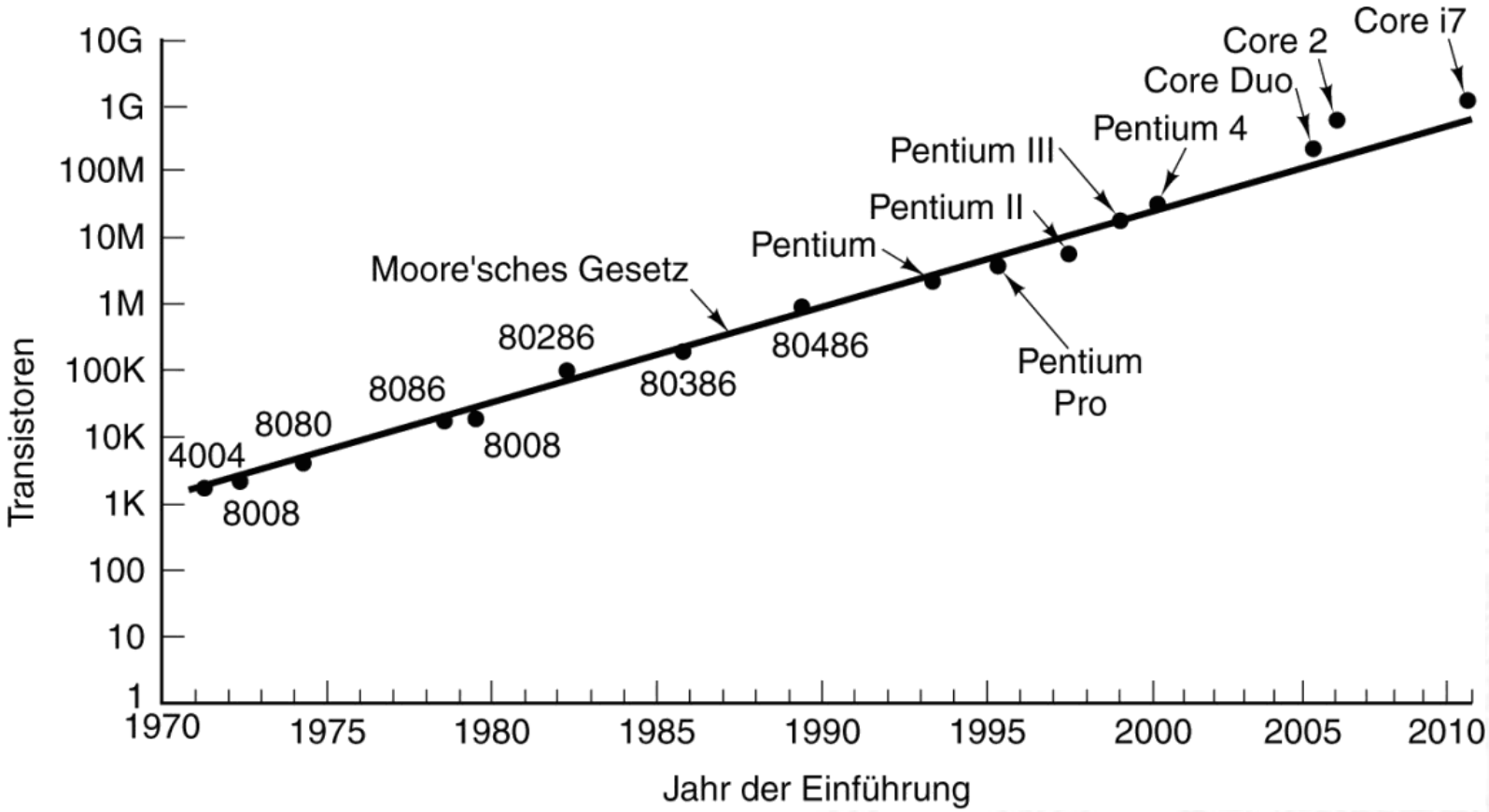
Moore's Law: Evolution der Prozessoren



[TA14]

Modell		Typ	Jahr	# Trans.
Xeon Platinum 8180	Intel	CPU	2017	8,0 Mrd.

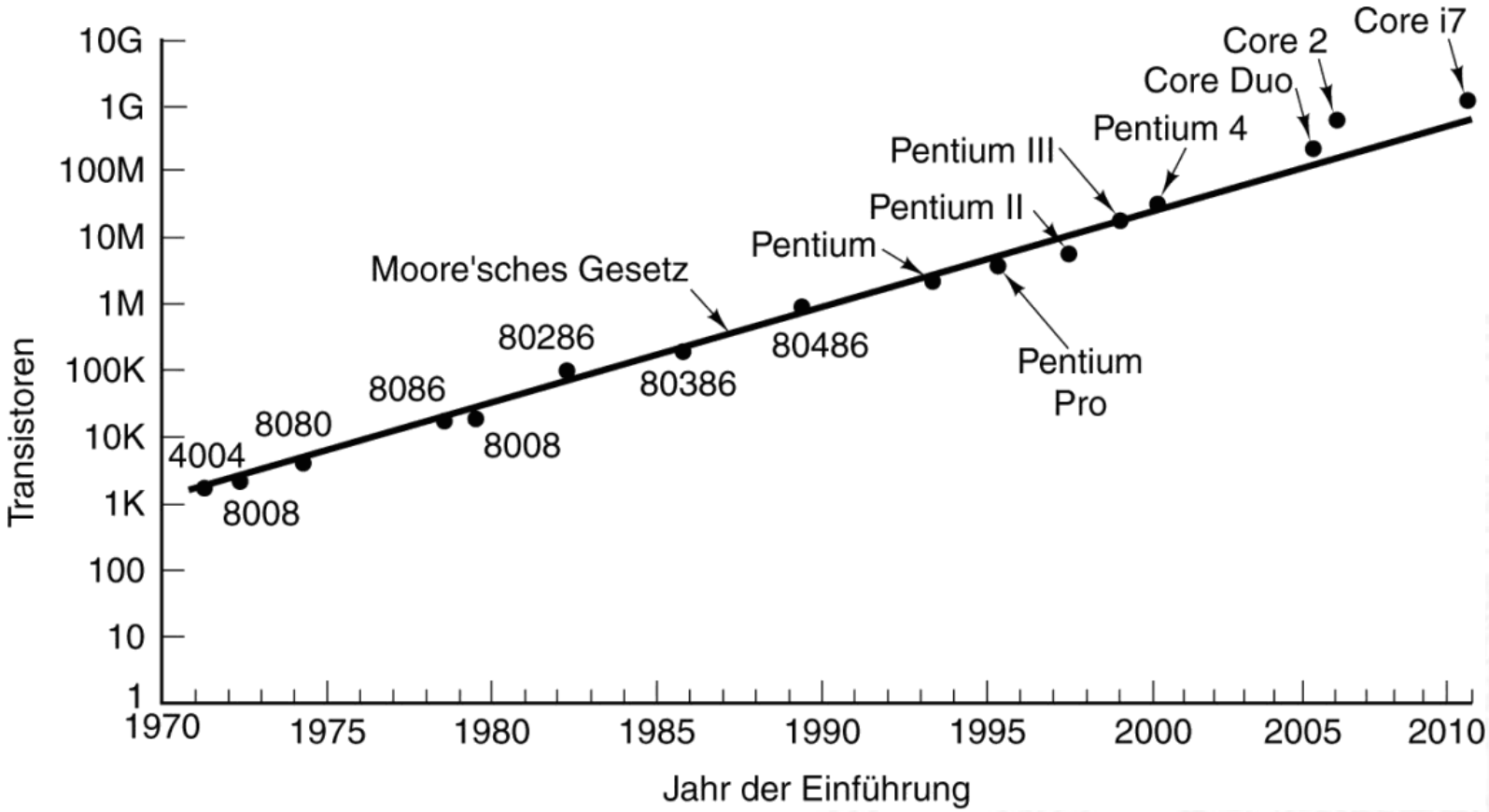
Moore's Law: Evolution der Prozessoren



[TA14]

Modell		Typ	Jahr	# Trans.
Xeon Platinum 8180	Intel	CPU	2017	8,0 Mrd.
A12X Bionic	Apple	SOC	2018	10,0 Mrd.

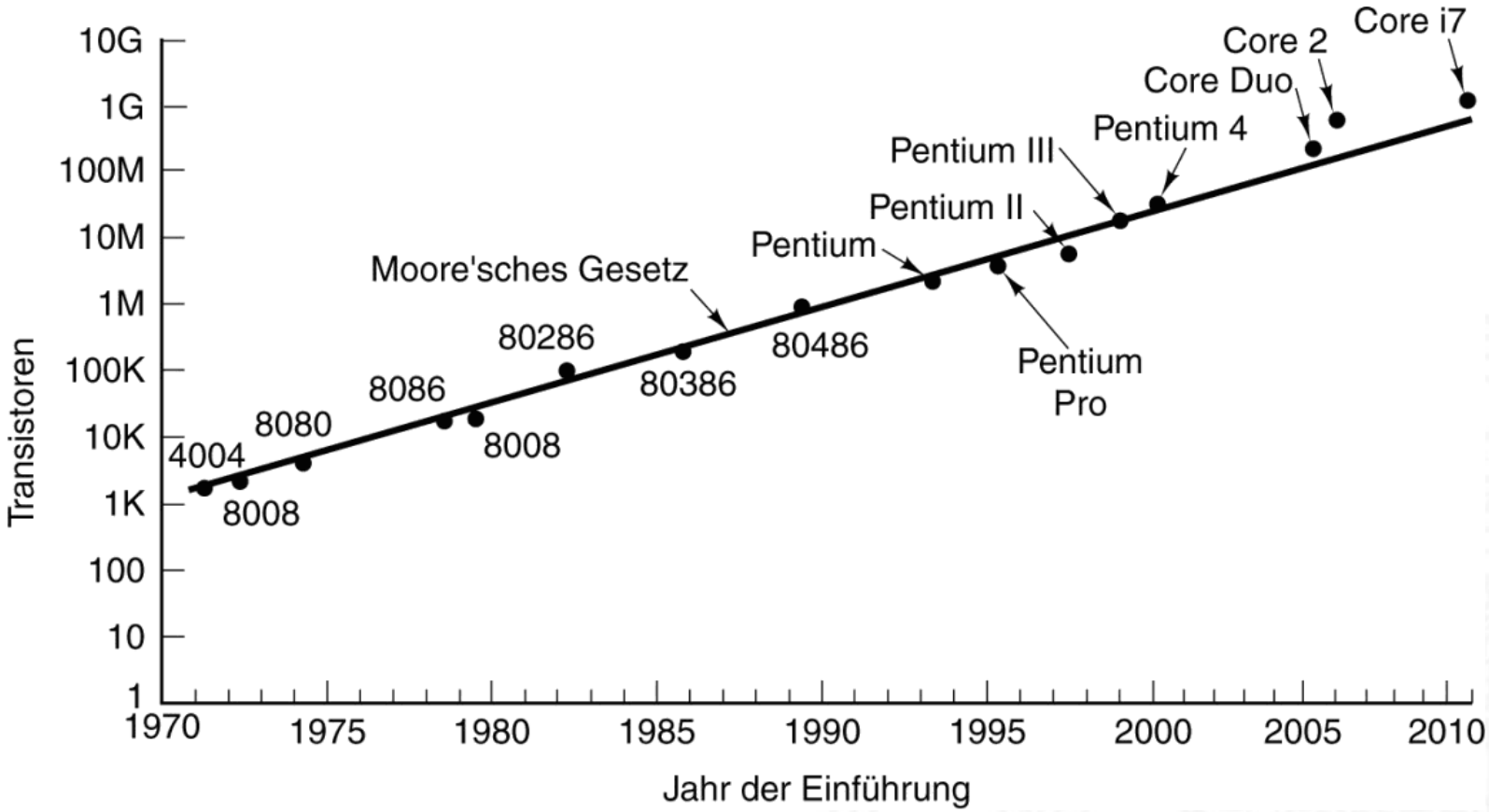
Moore's Law: Evolution der Prozessoren



[TA14]

Modell		Typ	Jahr	# Trans.
Xeon Platinum 8180	Intel	CPU	2017	8,0 Mrd.
A12X Bionic	Apple	SOC	2018	10,0 Mrd.
GV100 Volta	Nvidia	GPU	2017	21,1 Mrd.

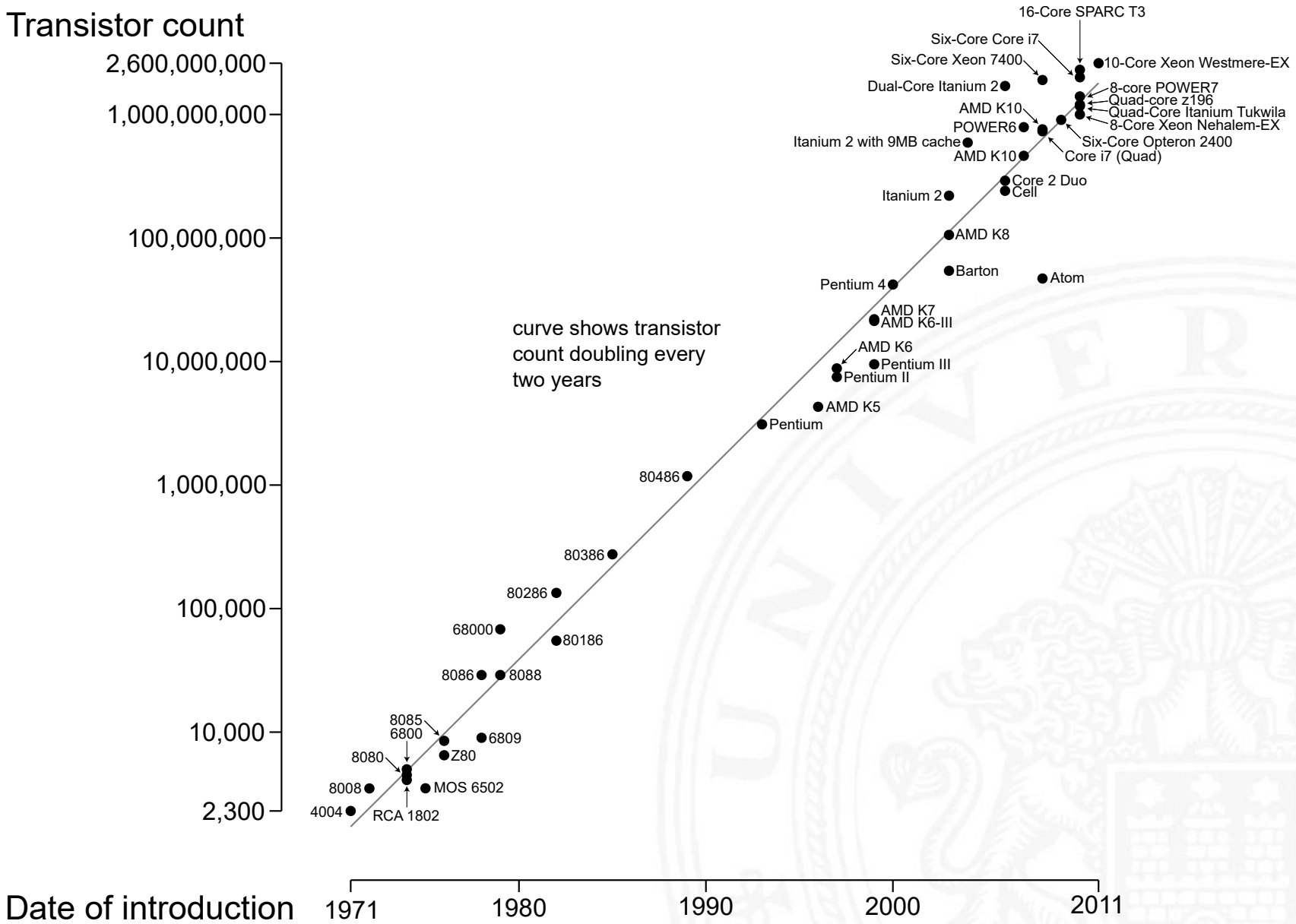
Moore's Law: Evolution der Prozessoren



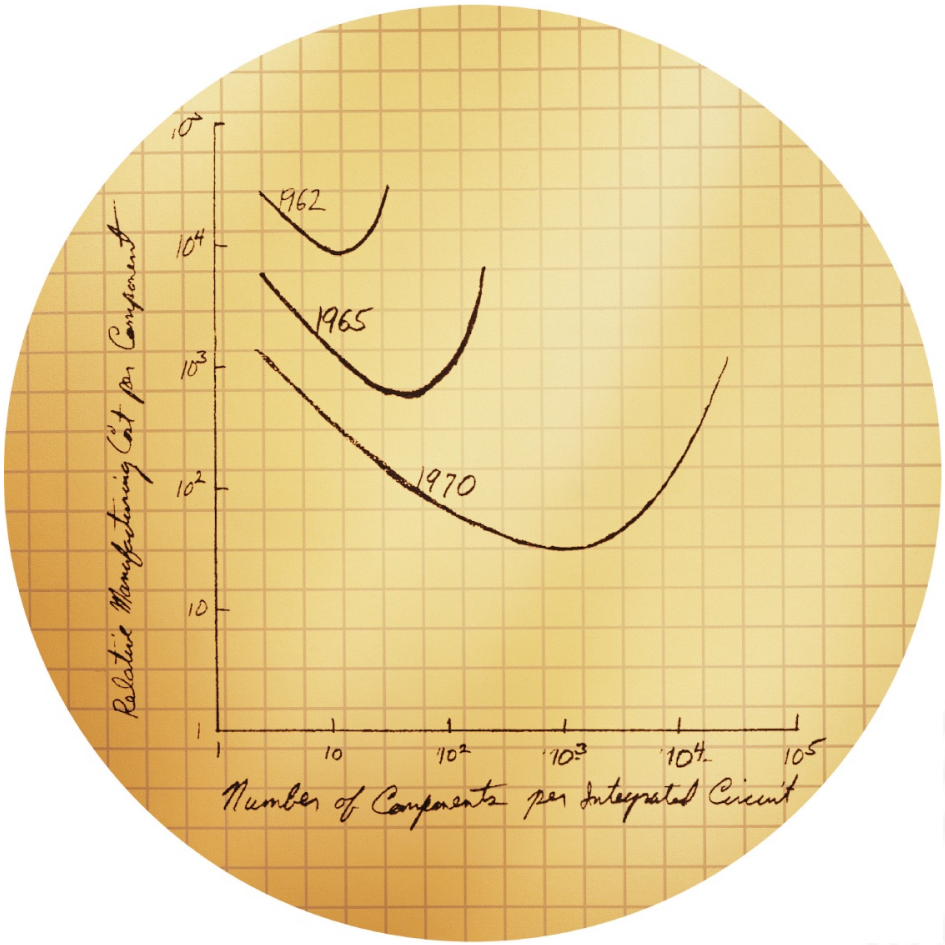
[TA14]

Modell		Typ	Jahr	# Trans.
Xeon Platinum 8180	Intel	CPU	2017	8,0 Mrd.
A12X Bionic	Apple	SOC	2018	10,0 Mrd.
GV100 Volta	Nvidia	GPU	2017	21,1 Mrd.
Versal VC1902	Xilinx	FPGA	2019	37,0 Mrd.

Moore's Law: Evolution der Prozessoren (cont.)



Moore's Law: Kosten pro Komponente



Originalskizze von G. Moore [Intel]

Moore's Law: Formel und Beispiele

$$L(t) = L(0) \cdot 2^{t/18}$$

mit: $L(t)$ = Leistung zum Zeitpunkt t,
$L(0)$ = Leistung zum Zeitpunkt 0,
und Zeit t in Monaten.

Einige Formelwerte:

Jahr 1:	1,5874
Jahr 2:	2,51984
Jahr 3:	4
Jahr 5:	10,0794
Jahr 6:	16
Jahr 7:	25,3984
Jahr 8:	40,3175

Leistungssteigerung der Spitzenrechner seit 1993

www.top500.org de.wikipedia.org/wiki/Supercomputer

Jahr	Rechner	CPU	Linpack [TFlop/s]	Prozessoren
1993	TMC CM-5/1024	(SuperSparc 32MHz)	0,0597	1 024
1994	Intel XP/S140	(80860 50MHz)	0,1434	3 680
1995	Fujitsu NWT	(105 MHz)	0,17	140
1996	Hitachi SR2201/1024	(HARP-1E 120MHz)	0,2204	1 024
1997	Intel ASCI Red	(Pentium Pro 200MHz)	1,068	7 264
1999	Intel ASCI Red	(Pentium Pro 333MHz)	2,121	9 472
2001	IBM ASCI White	(Power3 375MHz)	7,226	8 192
2002	NEC Earth Simulator	(NEC 1GHz)	35,86	5 120
2005	IBM BlueGene/L	(PowerPC 440 2C 700MHz)	136,8	65 536
2006	IBM BlueGene/L	(PowerPC 440 2C 700MHz)	280,6	131 072
2008	IBM Roadrunner	(Opteron 2C 1,8GHz + IBM Cell 9C 3,2 GHz)	1 026,0	122 400
2010	Cray XT5-HE Jaguar	(Opteron 6C 2,6GHz)	1 759,0	224 162
2011	Fujitsu K computer	(SPARC64 VIIIfx 2.0GHz)	8 162,0	548 352
2012	IBM SuperMUC	(Xeon E5-2680 8C 2,7GHz)	2 897,0	147 456
2012	IBM BlueGene/Q Sequoia	(Power BQC 16C 1,6GHz)	16 324,8	1 572 864
2013	IBM BlueGene/Q JUQUEEN	(Power BQC 16C 1,6GHz)	5 008,9	458 752
2013	NUDT Tianhe-2	(Xeon E5-2692 12C 2,2 GHz + Xeon Phi 31S1P)	33 862,7	3 120 000
2016	Sunway TaihuLight	(Sunway SW26010 260C 1,45 GHz)	93 014,6	10 649 600
2018	SuperMUC-NG	(Xeon Platinum 8174 24C 3,1GHz)	19 476,6	305 856
2018	Summit	(IBM Power 9 22C 3,07 GHz + NVIDIA GV100)	148 600,0	2 414 592

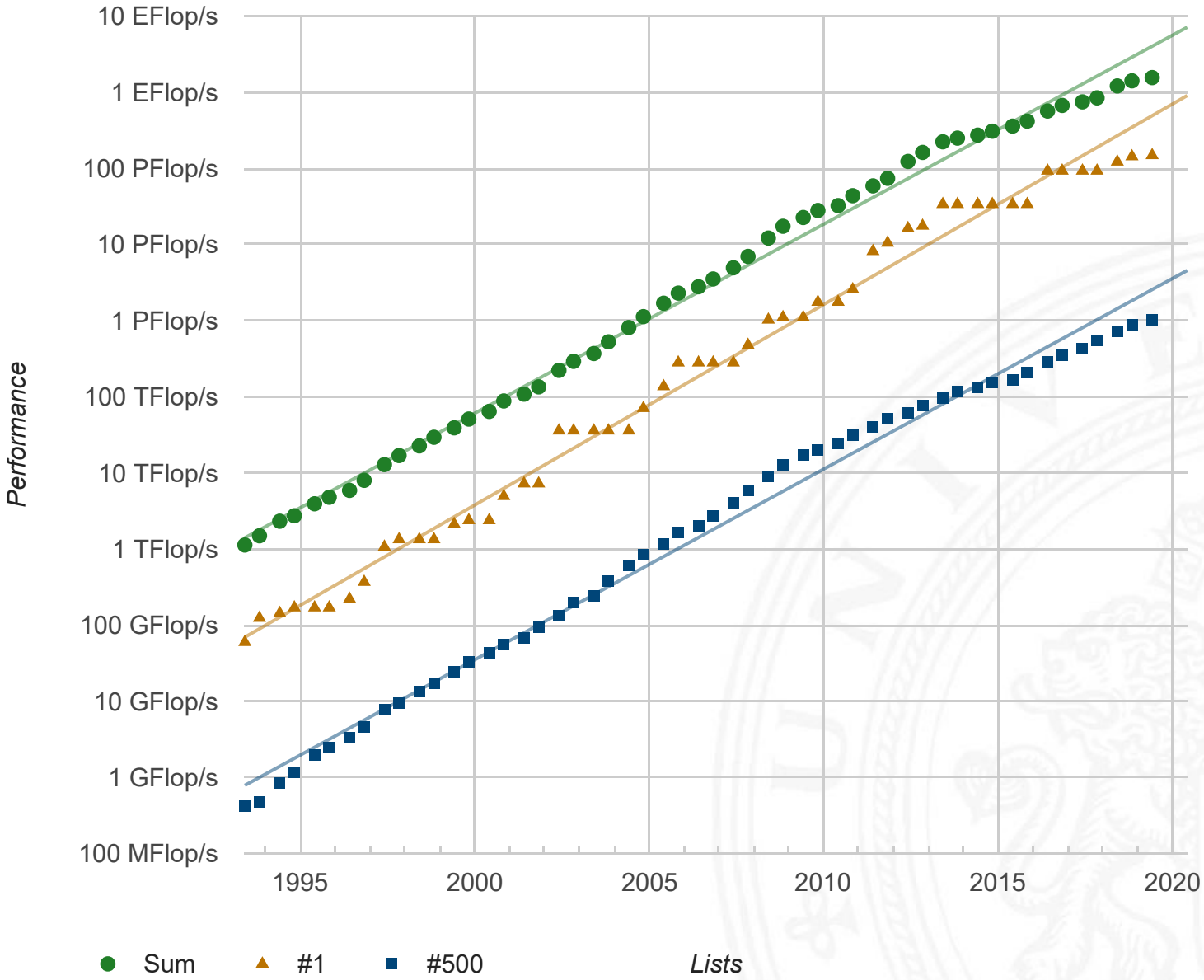
Leistungssteigerung der Spitzenrechner seit 1993

www.top500.org de.wikipedia.org/wiki/Supercomputer

Jahr	Rechner	CPU	Linpack [TFlop/s]	Prozessoren	Power [KW]
1993	TMC CM-5/1024	(SuperSparc 32MHz)	0,0597	1 024	
1994	Intel XP/S140	(80860 50MHz)	0,1434	3 680	
1995	Fujitsu NWT	(105 MHz)	0,17	140	
1996	Hitachi SR2201/1024	(HARP-1E 120MHz)	0,2204	1 024	
1997	Intel ASCI Red	(Pentium Pro 200MHz)	1,068	7 264	
1999	Intel ASCI Red	(Pentium Pro 333MHz)	2,121	9 472	
2001	IBM ASCI White	(Power3 375MHz)	7,226	8 192	
2002	NEC Earth Simulator	(NEC 1GHz)	35,86	5 120	3 200
2005	IBM BlueGene/L	(PowerPC 440 2C 700MHz)	136,8	65 536	716
2006	IBM BlueGene/L	(PowerPC 440 2C 700MHz)	280,6	131 072	1 433
2008	IBM Roadrunner	(Opteron 2C 1,8GHz + IBM Cell 9C 3,2 GHz)	1 026,0	122 400	2 345
2010	Cray XT5-HE Jaguar	(Opteron 6C 2,6GHz)	1 759,0	224 162	6 950
2011	Fujitsu K computer	(SPARC64 VIIIfx 2.0GHz)	8 162,0	548 352	9 899
2012	IBM SuperMUC	(Xeon E5-2680 8C 2,7GHz)	2 897,0	147 456	3 423
2012	IBM BlueGene/Q Sequoia	(Power BQC 16C 1,6GHz)	16 324,8	1 572 864	7 890
2013	IBM BlueGene/Q JUQUEEN	(Power BQC 16C 1,6GHz)	5 008,9	458 752	2 301
2013	NUDT Tianhe-2	(Xeon E5-2692 12C 2,2 GHz + Xeon Phi 31S1P)	33 862,7	3 120 000	17 808
2016	Sunway TaihuLight	(Sunway SW26010 260C 1,45 GHz)	93 014,6	10 649 600	15 371
2018	SuperMUC-NG	(Xeon Platinum 8174 24C 3,1GHz)	19 476,6	305 856	
2018	Summit	(IBM Power 9 22C 3,07 GHz + NVIDIA GV100)	148 600,0	2 414 592	10 096

Leistungssteigerung der Spitzenrechner seit 1993 (cont.)

www.top500.org de.wikipedia.org/wiki/Supercomputer



Moore's Law: Aktuelle Trends

- Miniaturisierung schreitet weiter fort
- aber Taktraten erreichen physikalisches Limit
- steigender Stromverbrauch, zwei Effekte:
  1. Leckströme
  2. proportional zu Taktrate

Entwicklungen

- $>$ 4 GByte Hauptspeicher sind Standard
- 64-bit Adressierung

$\Rightarrow$ seit 2011: CPU plus Grafikeinheit

$\Rightarrow$ Integration mehrerer CPUs auf einem Chip (2-. . . 32-Cores)

$\Rightarrow$ Cache Speicher (SRAM) auf dem Die

$\Rightarrow$ Integration von Peripheriegeräten (Speicherinterface, PCIe, . . .)

$\Rightarrow$ **SoC**: „System on a chip“

SoC: System on a chip

Gesamtes System auf einem Chip integriert:

- ein oder mehrere Prozessoren, z.T. verschiedene Typen
  - hohe Rechenleistung
  - energieeffizient

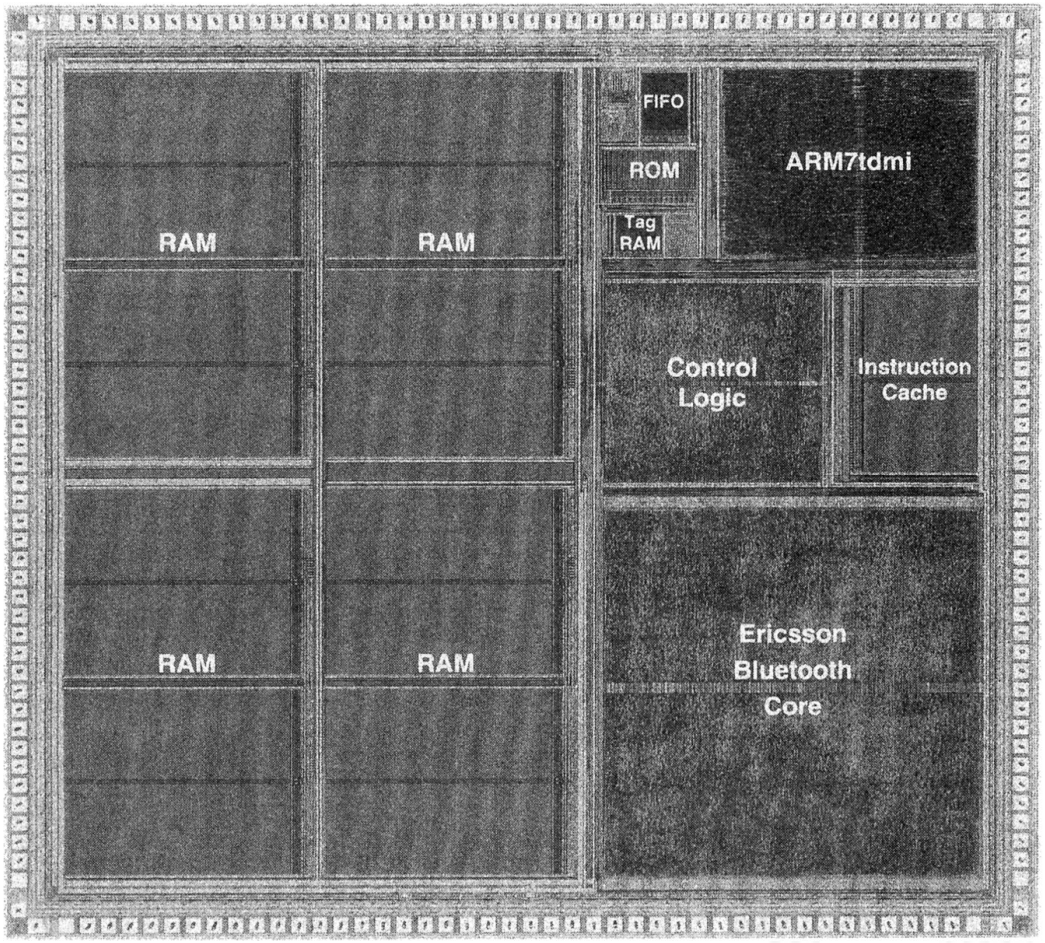
  $\Rightarrow$ z.B. ARM mit *big.LITTLE* Konzept
- Cache Hierarchie: 1-Level D- und I-Cache / 2-Level
- dedizierte Prozessoren: Grafik, Video(de)codierung, DSP ...
- Hauptspeicher (evtl. auch extern), Speichercontroller
- weitere Speicher für Medien/Netzwerkoperationen

SoC: System on a chip (cont.)

- Peripherieblöcke nach Kundenwunsch konfiguriert:
  - Displayansteuerung: DP, HDMI ...
  - A/V-Schnittstellen: Kamera, Mikrofone, Audio ...
  - serielle und parallele Schnittstellen, SPI, I/O-Pins ...
  - Feldbusse: I^2C, CAN ...
  - PC-like: USB, Firewire, SATA ...
  - Netzwerk kabelgebunden (Ethernet)
  - Funkschnittstellen: WLAN, Bluetooth, 4G ...
- Smartphones, Tablet-Computer, Medien-/DVD-Player, WLAN-Router, NAS-/Home-Server ...

SoC Beispiele

- Bluetooth-Controller (2000)



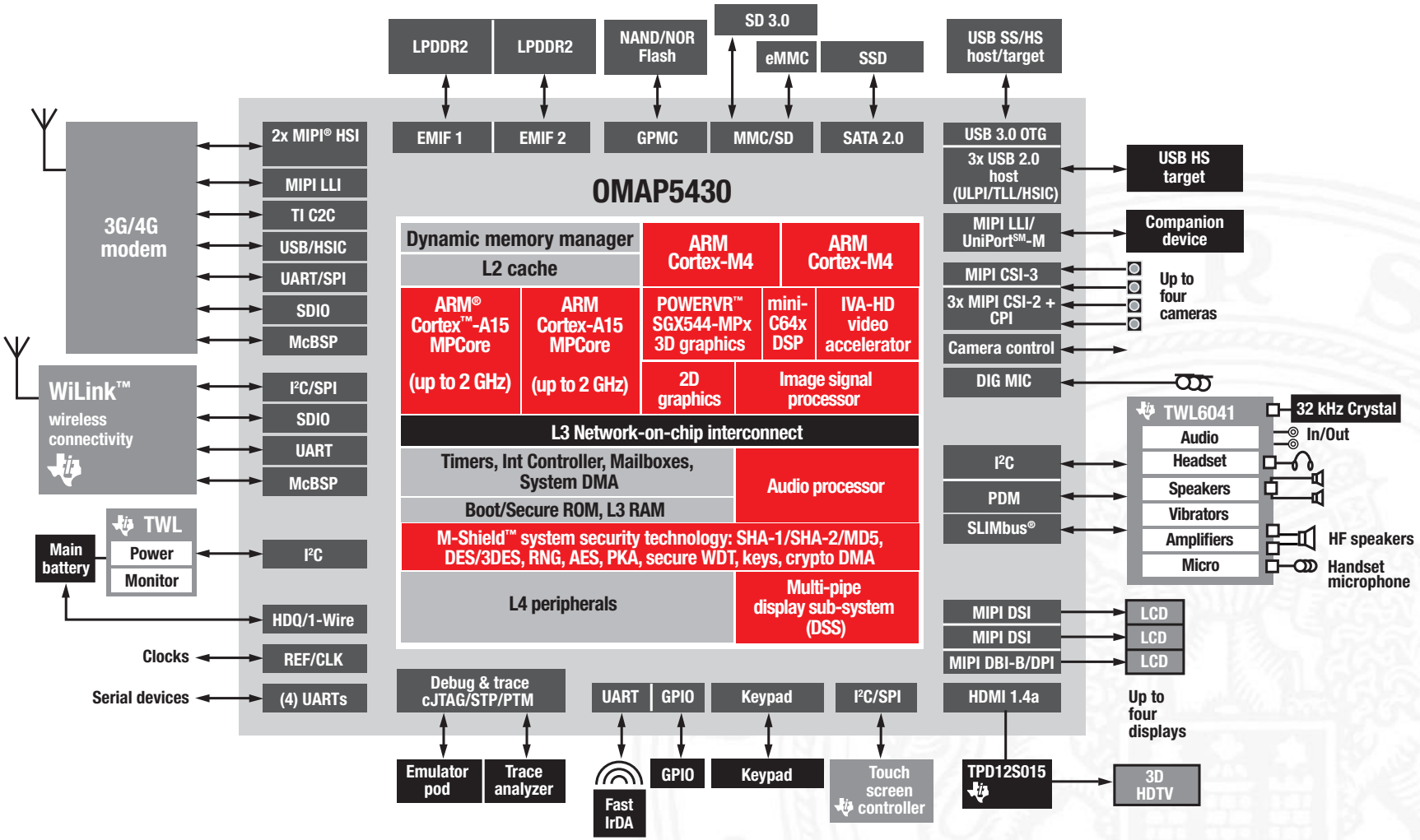
© VLSI Technology, Inc.

[Fur00]

Prozess	0,25 μm
Metall	3-Layer
V_{DD}	2,5 V
Transistoren	4,3 Mill.
Chipfläche	20 mm^2
Taktrate	0 . . . 13 MHz
MIPS	12
Power	75 mW
MIPS/W	160

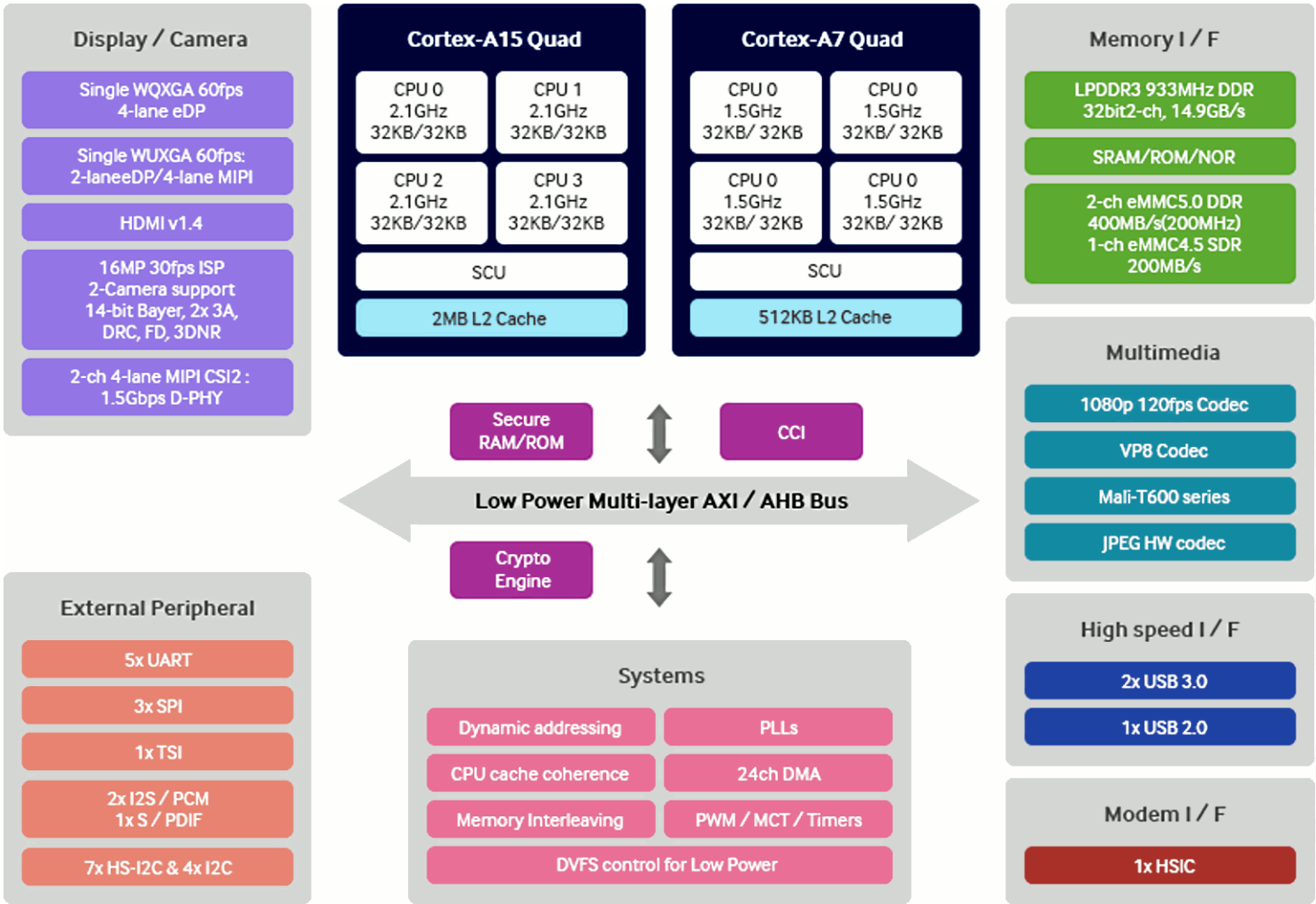
SoC Beispiele (cont.)

- Texas Instruments OMAP 5430 (2011) [TI]



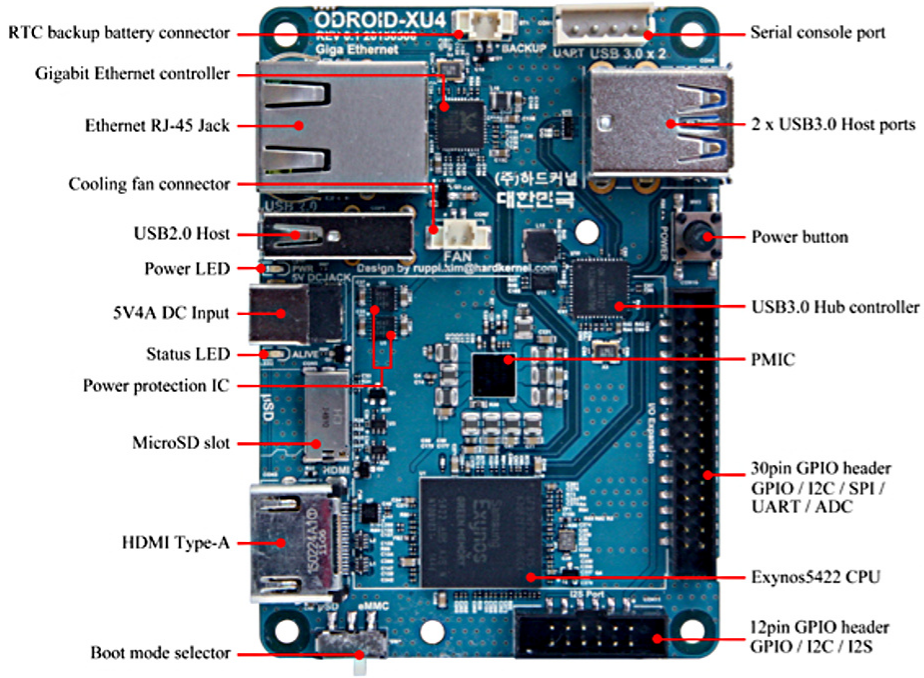
SoC Beispiele (cont.)

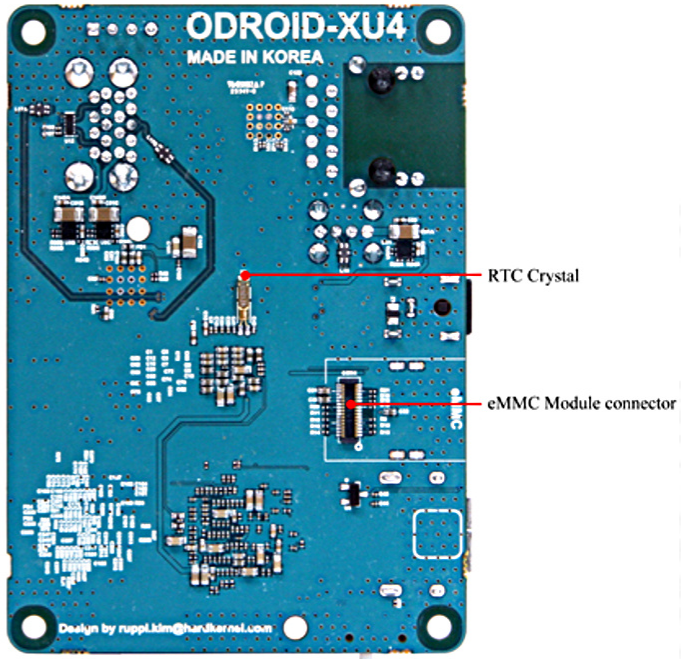
- Samsung Exynos-5422 (2014) [Samsung]



SoC Beispiele (cont.)

- Beispiel: Odroid XU4 [HK]





- vollständiger 8-Kern Microcomputer
- Betriebssystem: Android oder Linux

Grenzen des Wachstums

- Jeder exponentielle Verlauf stößt irgendwann an natürliche oder wirtschaftliche Grenzen
- Beispiel: physikalische Limits
  - Eine DRAM-Speicherzelle speichert etwa 200 Elektronen (2012) Skalierung: es werden mit jeder neuen Technologiestufe weniger
  - Offensichtlich ist die Grenze spätestens dann erreicht, wenn nur noch ein einziges Elektron gespeichert würde
  - Ab diesem Zeitpunkt gibt es bessere Performanz nur noch durch bessere Algorithmen / Architekturen!

$\Rightarrow$ Annahme: 50 % Skalierung pro Jahr, 200 Elektronen/Speicherzelle
gesucht: $x \widehat{=}$Jahre Fortschritt

$\Rightarrow$ $200/(1,5^x) \geq 1$
$x = \ln(200)/\ln(1,5) \approx 13$ Jahre

$a^b = \exp(b \cdot \ln a)$

Roadmap: IRDS

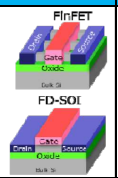
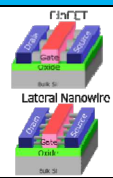
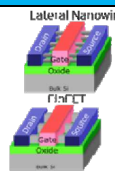
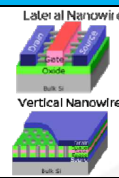
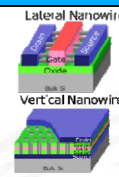
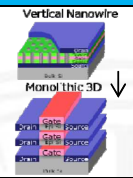
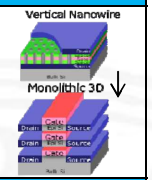
IEEE **I**nternational **R**oadmap for **D**evices and **S**ystems

`https://irds.ieee.org/editions/2018`

- IEEE: Institute of Electrical and Electronics Engineers
- Beteiligung von
  - Halbleiterherstellern
  - Geräte-Herstellern
  - Universitäten und Forschungsinstituten
  - Fachverbänden aus USA, Europa, Asien
- Publikation von langjährigen Vorhersagen
- Zukünftige Entwicklung der Halbleitertechnologie
- Prognosen zu Fertigungsprozessen, Modellierung, Simulation, Entwurf etc.
- für Chips (Speicher, Prozessoren, SoC . . .) und Systeme

Roadmap: IRDS (cont.)

Table MM01 – More Moore – Logic Core Device Technology Roadmap (Ausschnitt, 2017)

YEAR OF PRODUCTION	2017	2019	2021	2024	2027	2030	2033
	P54M36	P48M28	P42M24	P36M21	P32M14	P32M14T2	P32M14T4
Logic industry "Node Range" Labeling (nm)	"10"	"7"	"5"	"3"	"2.1"	"1.5"	"1.0"
IDM-Foundry node labeling	i10-f7	i7-f5	i5-f3	i3-f2.1	i2.1-f1.5	i1.5-f1.0	i1.0-f0.7
Logic device structure options	finFET FDSOI	finFET LGAA	LGAA finFET	LGAA VGAA	LGAA VGAA	VGAA, LGAA 3DVLSI	VGAA, LGAA 3DVLSI
Logic device mainstream device	finFET	finFET	LGAA	LGAA	LGAA	VGAA	VGAA
DEVICE STRUCTURES							
							
LOGIC TECHNOLOGY ANCHORS							
Patterning technology inflection for Mx interconnect	193i, EUV	193i, EUV DP	193i, EUV DP	193i, High-NA EUV	193i, High-NA EUV+(DSA)	193i, High-NA EUV+(DSA)	193i, High-NA EUV+(DSA)
Channel material technology inflection	Si	SiGe25%	SiGe50%	Ge, IIIV (TFET?), 2D Mat	Ge, IIIV (TFET?), 2D Mat	Ge, IIIV (TFET?), 2D Mat	Ge, IIIV (TFET?), 2D Mat
Process technogy inflection	Conformal deposition	Conformal Doping, Contact	Channel, RMG	Stacked-device Non-Cu Mx	Stacked-device Non-Cu Mx	Steep-SS, 3D	Steep-SS, 3D
Stacking generation inflection	2D	2D	3D-stacking: W2W D2W	3D-device: P-over-N Hetero	3D-device: Mem-on-Logic Hetero	3D-device: Mem-on-Logic Hetero	3D-device: Logic-on-Logic Hetero
LOGIC TECHNOLOGY INTEGRATION CAPACITY							
Design scaling factor for standard cell	-	0,98	1,09	0,96	1,03	2,00	1,00
Design scaling factor for SRAM (111) bitcell	-	1,00	1,00	1,00	1,00	1,25	1,00
▪ ▪ ▪							
POWER AND PERFORMANCE SCALING FACTORS							
Vdd (V)	0,75	0,70	0,65	0,65	0,65	0,60	0,55
Physical gate length for HP Logic (nm)	20,0	18,0	16,0	14,0	12,0	12,0	12,0
Datapath speed improvement at Vdd - relative	1,00	1,19	1,21	1,34	1,56	1,60	1,70
▪ ▪ ▪							
Power density of logic path cube at fmax - relative	1,00	1,20	1,21	1,82	2,69	4,49	8,00
fmax of a single CPU core at Vdd (GHz)	2,5	3,0	3,0	3,3	3,9	4,0	4,2
favg at constant power density and Vdd (GHz)	2,50	2,48	2,51	1,84	1,45	0,89	0,53
CPU SiP throughput at fmax (TFLOPS/sec)	0,16	0,27	0,46	0,79	1,34	2,27	3,86
▪ ▪ ▪							
INTERCONNECT TECHNOLOGY							
Conductor	Cu, non-Cu	Cu, non-Cu	Cu, non-Cu	Cu, non-Cu	Cu, non-Cu	Cu, non-Cu	Cu, non-Cu
Number of wiring layers	14	16	18	20	20	20	20
▪ ▪ ▪							

Moore's Law

Beispiel für die Auswirkung von Moore's Law

Angenommen die Lösung einer Rechenaufgabe dauert derzeit vier Jahre und die Rechenleistung wächst jedes Jahr um 60 %.

Wie lösen wir das Problem ?

Moore's Law: Schöpferische Pause

Beispiel für die Auswirkung von Moore's Law

Angenommen die Lösung einer Rechenaufgabe dauert derzeit vier Jahre und die Rechenleistung wächst jedes Jahr um 60 %.

Ein mögliches Vorgehen ist dann das folgende:

- Wir warten drei Jahre, kaufen dann einen neuen Rechner und erledigen die Rechenaufgabe in einem Jahr.
- *Wie das ?*

Moore's Law: Schöpferische Pause

Beispiel für die Auswirkung von Moore's Law

Angenommen die Lösung einer Rechenaufgabe dauert derzeit vier Jahre und die Rechenleistung wächst jedes Jahr um 60 %.

Ein mögliches Vorgehen ist dann das folgende:

- Wir warten drei Jahre, kaufen dann einen neuen Rechner und erledigen die Rechenaufgabe in einem Jahr.

$\Rightarrow$ Nach einem Jahr können wir einen Rechner kaufen, der um den Faktor 1,6 Mal schneller ist, nach zwei Jahren bereits $1{,}6 \cdot 1{,}6$ Mal schneller, und nach drei Jahren (also am Beginn des vierten Jahres) gilt $(1 + 60\,\%)^3 = 4{,}096$.

- Wir sind also sogar ein bisschen schneller fertig, als wenn wir den jetzigen Rechner die ganze Zeit durchlaufen lassen.

Wie geht es jetzt weiter?

Ab jetzt erst mal ein *bottom-up* Vorgehen:

Start mit grundlegenden Aspekten

- Informationsverarbeitung und -repräsentation
- Darstellung von Zahlen und Zeichen
- arithmetische und logische Operationen
- Schaltnetze, Schaltwerke, endliche Automaten

dann Kennenlernen aller Basiskomponenten des Digitalrechners

- Gatter, Flipflops ...
- Register, ALU, Speicher ...

und Konstruktion eines Rechners (HW)
mit seinen Betriebsmitteln (SW)

- Befehlssatz, -abarbeitung, Assembler
- Pipelining, Speicherhierarchie
- Dateisystem, Ein- / Ausgabe
- Prozesskontrolle, Locking, Interrrupts
- ...

Literatur

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*. 6. Auflage, Pearson Deutschland GmbH, 2014. ISBN 978–3–8689–4238–5

[HenHA] N. Hendrich: *HADES — HAmburg DEsign System*. Universität Hamburg, FB Informatik, Lehrmaterial. `tams.informatik.uni-hamburg.de/applets/hades/webdemos`

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

[Moo65] G.E. Moore: *Cramming More Components Onto Integrated Circuits*. in: *Electronics* 38 (1965), April 19, Nr. 8

Literatur (cont.)

[IRDS18] *International Roadmap for Devices and Systems (IRDS) 2018 Edition*. IEEE International Roadmap for Devices and Systems, 2018. irds.ieee.org/editions/2018

[Intel] Intel Corp.; Santa Clara, CA.
www.intel.com
www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html

[TI] Texas Instruments Inc.; Dallas, TX. www.ti.com

[Samsung] Samsung Electronics Co., Ltd.; Suwon, Südkorea.
www.samsung.com

[HK] Hardkernel co., Ltd.; AnYang, Südkorea.
www.hardkernel.com

Gliederung

Gliederung (cont.)

Definition: Digitalrechner

Tanenbaum, Austin: *Rechnerarchitektur* [TA14]

Ein Computer oder Digitalrechner ist eine Maschine, die Probleme für den Menschen lösen kann, indem sie die ihr gegebenen Befehle ausführt. Eine Befehlssequenz, die beschreibt, wie eine bestimmte Aufgabe auszuführen ist, nennt man ***Programm****.*
Die elektronischen Schaltungen eines Computers verstehen eine begrenzte Menge einfacher Befehle, in die alle Programme konvertiert werden müssen, bevor sie sich ausführen lassen. . . .

- Probleme lösen: durch Abarbeiten einfacher **Befehle**
- Abfolge solcher Befehle ist ein **Programm**
- Maschine versteht nur ihre eigene **Maschinensprache**

Befehlssatz und Semantic Gap

. . . verstehen eine begrenzte Menge einfacher Befehle . . .

Typische Beispiele für solche Befehle:

- addiere die zwei Zahlen in Register R1 und R2
- überprüfe, ob das Resultat Null ist
- kopiere ein Datenwort von Adresse 13 ins Register R4

$\Rightarrow$ extrem niedriges Abstraktionsniveau

- natürliche Sprache immer mit Kontextwissen
  Beispiel: „vereinbaren Sie einen Termin mit dem Steuerberater“

- **Semantic gap**:
  Diskrepanz zu einfachen elementaren Anweisungen
- Vermittlung zwischen Mensch und Computer erfordert zusätzliche Abstraktionsebenen und Software

Rechnerarchitektur bzw. -organisation

- Definition solcher Abstraktionsebenen bzw. Schichten
- mit möglichst einfachen und sauberen Schnittstellen
- jede Ebene definiert eine neue (mächtigere) **Sprache**

- diverse Optimierungs-Kriterien/Möglichkeiten:
  - Performanz, Größe, Leistungsaufnahme ...
  - Kosten: Hardware, Software, Entwurf ...
  - Zuverlässigkeit, Wartungsfreundlichkeit, Sicherheit ...

Achtung / Vorsicht:

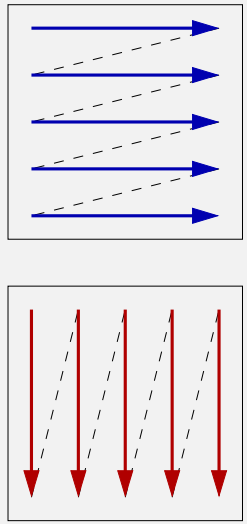
- Gesamtverständnis erfordert Kenntnisse auf allen Ebenen
- häufig Rückwirkung von unteren auf obere Ebenen

Rückwirkung von unteren Ebenen: Arithmetik


```
public class Overflow {
  ...
  public static void main( String[] args ) {
    printInt( 0 );                      //  0
    printInt( 1 );                      //  1
    printInt( -1 );                     // -1
    printInt( 2+(3*4) );                // 14
    printInt( 100*200*300 );            // 6000000
    printInt( 100*200*300*400 );        // -1894967296          (!)
    printDouble( 1.0 );                 // 1.0
    printDouble( 0.3 );                 // 0.3
    printDouble( 0.1 + 0.1 + 0.1 );     // 0.30000000000000004 (!)
    printDouble( (0.3) - (0.1+0.1+0.1) ); // -5.5E-17           (!)
  }
}
```

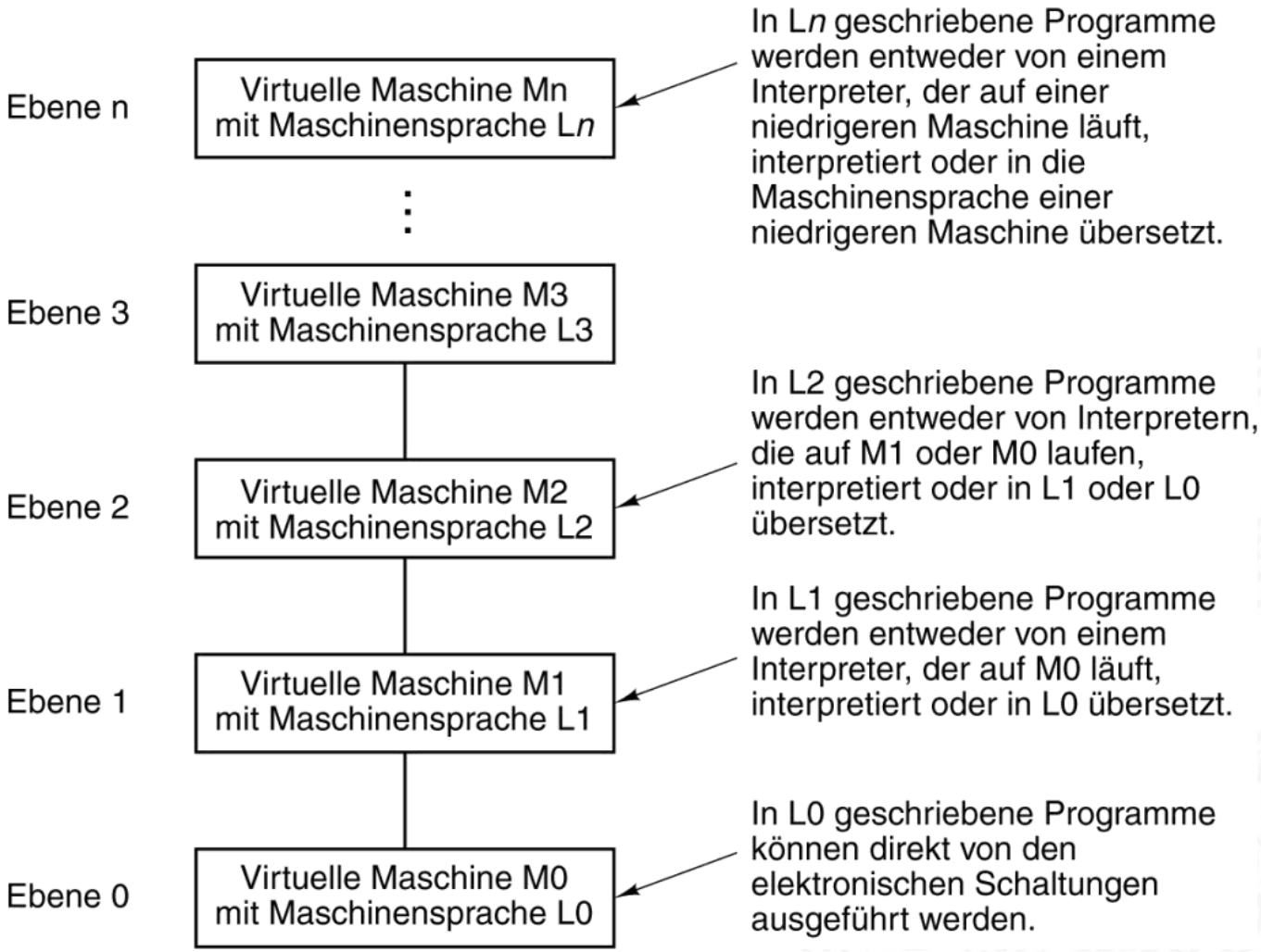
Rückwirkung von unteren Ebenen: Performanz


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



Matrix creation (5000×5000)	2105 ms	
Matrix row-col summation	75 ms	
Matrix col-row summation	383 ms	$\Rightarrow$ 5 × langsamer
Sum = 600,8473695346258 / 600,8473695342268		$\Rightarrow$ andere Werte

Maschine mit mehreren Ebenen



Tanenbaum: *Structured Computer Organization* [TA14]

Abstraktionsebenen und Sprachen

- jede Ebene definiert eine neue (mächtigere) Sprache
- Abstraktionsebene $\Longleftrightarrow$ Sprache
- L0 $<$ L1 $<$ L2 $<$ L3 $<$ ...

Software zur Übersetzung zwischen den Ebenen

- **Compiler**:
  Erzeugen eines neuen Programms, in dem jeder L1 Befehl durch eine zugehörige Folge von L0 Befehlen ersetzt wird
- **Interpreter**:
  direkte Ausführung der L0 Befehlsfolgen zu jedem L1 Befehl

Virtuelle Maschine

- für einen Interpreter sind L1 Befehle einfach nur Daten
- die dann in die zugehörigen L0 Befehle umgesetzt werden

$\Rightarrow$ dies ist gleichwertig mit einer:

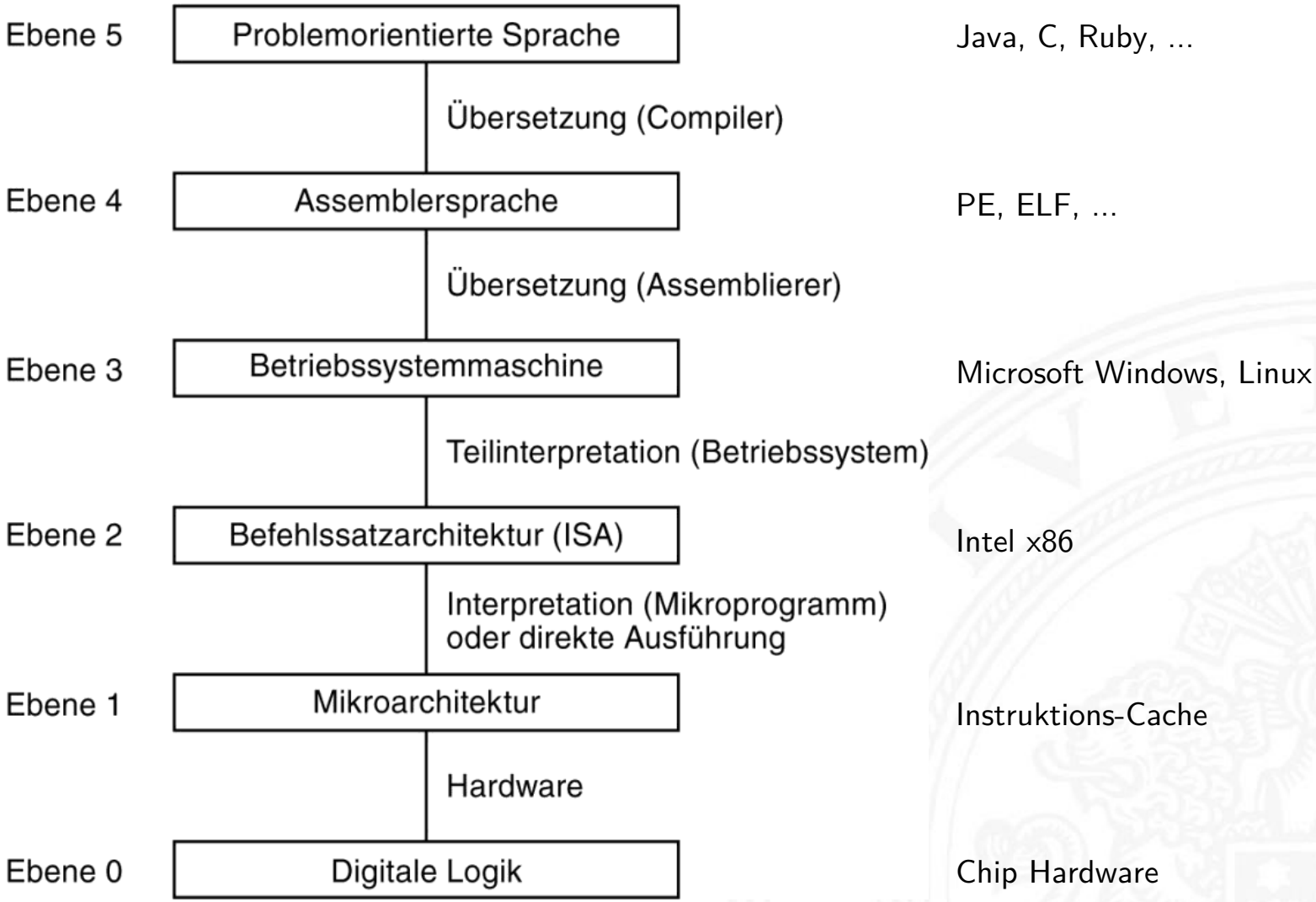
Virtuellen Maschine M1 für die Sprache L1

- ein Interpreter erlaubt es, jede beliebige Maschine zu simulieren
- und zwar auf jeder beliebigen (einfacheren) Maschine M0
- Programmierer muss sich nicht um untere Schichten kümmern
- Nachteil: die virtuelle Maschine ist meistens langsamer als die echte Maschine M1
- Maschine M0 kann wiederum eine virtuelle Maschine sein (!)
- unterste Schicht ist jeweils die Hardware

Übliche Einteilung der Ebenen

Anwendungsebene	Hochsprachen (Java, Smalltalk . . .)
Assemblerebene	low-level Anwendungsprogrammierung
Betriebssystemebene	Betriebssystem, Systemprogrammierung
Rechnerarchitektur	Schnittstelle zwischen SW und HW, Befehlssatz, Datentypen
Mikroarchitektur	Steuerwerk und Operationswerk: Register, ALU, Speicher . . .
Logikebene	Grundschaltungen: Gatter, Flipflops . . .
Transistorebene	Elektrotechnik, Transistoren, Chip-Layout
Physikalische Ebene	Geometrien, Chip-Fertigung

Beispiel: Sechs Ebenen



Hinweis: Ebenen vs. Vorlesungen im BSc-Studiengang

Anwendungsebene:	SE1+SE2, AD ...
Assemblerebene:	RSB
Betriebssystemebene:	RSB, VSS
Rechnerarchitektur:	RSB
Mikroarchitektur:	RSB
Logikebene:	RSB
Device-Level:	–

HelloWorld: Anwendungsebene Quellcode

```
/* HelloWorld.c - print a welcome message */

#include <stdio.h>

int main( int argc, char ** argv ) {
  printf( "Hello, world!\n" );
  return 0;
}
```

Übersetzung

```
gcc -S HelloWorld.c
gcc -c HelloWorld.c
gcc -o HelloWorld.exe HelloWorld.c
```

```
        .file   "HelloWorld.c"
        .section        .rodata
.LC0:
        .string "Hello, world!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
```

HelloWorld: Objectcode `od -x HelloWorld.o`

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000020 0001 003e 0001 0000 0000 0000 0000 0000
0000040 0000 0000 0000 0000 02b0 0000 0000 0000
0000060 0000 0000 0040 0000 0000 0040 000d 000a
0000100 4855 e589 8348 10ec 7d89 48fc 7589 bff0
0000120 0000 0000 00e8 0000 b800 0000 0000 c3c9
0000140 6548 6c6c 2c6f 7720 726f 646c 0021 4700
0000160 4343 203a 5528 7562 746e 2075 2e35 2e34
0000200 2d30 7536 7562 746e 3175 317e 2e36 3430
0000220 312e 2930 3520 342e 302e 3220 3130 3036
0000240 3036 0039 0000 0000 0014 0000 0000 0000
0000260 7a01 0052 7801 0110 0c1b 0807 0190 0000
0000300 001c 0000 001c 0000 0000 0000 0020 0000
0000320 4100 100e 0286 0d43 5b06 070c 0008 0000
0000340 0000 0000 0000 0000 0000 0000 0000 0000
0000360 0000 0000 0000 0000 0001 0000 0004 fff1
...
```

HelloWorld: Disassemblieren `objdump -d HelloWorld.o`

```
HelloWorld.o:     Dateiformat elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   55                          push   %rbp
   1:   48 89 e5                    mov    %rsp,%rbp
   4:   48 83 ec 10                 sub    $0x10,%rsp
   8:   89 7d fc                    mov    %edi,-0x4(%rbp)
   b:   48 89 75 f0                 mov    %rsi,-0x10(%rbp)
   f:   bf 00 00 00 00              mov    $0x0,%edi
  14:   e8 00 00 00 00              callq  19 <main+0x19>
  19:   b8 00 00 00 00              mov    $0x0,%eax
  1e:   c9                          leaveq
  1f:   c3                          retq
```

HelloWorld: Maschinencode `od -x HelloWorld.exe`

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000020 0002 003e 0001 0000 0430 0040 0000 0000
0000040 0040 0000 0000 0000 19e0 0000 0000 0000
0000060 0000 0000 0040 0038 0009 0040 001f 001c
0000100 0006 0000 0005 0000 0040 0000 0000 0000
0000120 0040 0040 0000 0000 0040 0040 0000 0000
0000140 01f8 0000 0000 0000 01f8 0000 0000 0000
0000160 0008 0000 0000 0000 0003 0000 0004 0000
0000200 0238 0000 0000 0000 0238 0040 0000 0000
0000220 0238 0040 0000 0000 001c 0000 0000 0000
0000240 001c 0000 0000 0000 0001 0000 0000 0000
0000260 0001 0000 0005 0000 0000 0000 0000 0000
0000300 0000 0040 0000 0000 0000 0040 0000 0000
0000320 070c 0000 0000 0000 070c 0000 0000 0000
0000340 0000 0020 0000 0000 0001 0000 0006 0000
0000360 0e10 0000 0000 0000 0e10 0060 0000 0000
...
```

Hardware: „*Versteinerte Software*“

- eine virtuelle Maschine führt L1 Software aus
- und wird mit Software oder Hardware realisiert

⇒ Software und Hardware sind logisch äquivalent
„Hardware is just petrified Software“ Karen Panetta Lentz
– jedenfalls in Bezug auf L1 Programmausführung

Entscheidung für Software- oder Hardwarerealisierung?

- abhängig von vielen Faktoren, u.a.
- Kosten, Performanz, Zuverlässigkeit
- Anzahl der (vermuteten) Änderungen und Updates
- Sicherheit gegen Kopieren ...

Information

- **Information** $\sim$ abstrakter Gehalt einer Aussage
- Die Aussage selbst, mit der die Information dargestellt bzw. übertragen wird, ist eine **Repräsentation** der Information
- im Kontext der Informationsverarbeitung / -übertragung: **Nachricht**
- Das Ermitteln der Information aus einer Repräsentation heißt **Interpretation**
- Das Verbinden einer Information mit ihrer Bedeutung in der realen Welt heißt **Verstehen**

Repräsentation: Beispiele

Beispiel: Mit der Information „25“ sei die abstrakte Zahl gemeint, die sich aber nur durch eine Repräsentation angeben lässt:

- Text deutsch: fünfundzwanzig
- Text englisch: twentyfive

  ...
- Zahl römisch: XXV
- Zahl dezimal: 25
- Zahl binär: 11001
- Zahl Dreiersystem: 221

  ...
- Morse-Code: ••−−− •••••

Interpretation: Information vs. Repräsentation

- Wo auch immer Repräsentationen auftreten, meinen wir eigentlich die Information, z.B.:

$$5 \cdot (2 + 3) = 25$$

- Die Information selbst kann man überhaupt nicht notieren (!)
- Es muss immer Absprachen geben über die verwendete Repräsentation. Im obigen Beispiel ist implizit die Dezimaldarstellung gemeint, man muss also die Dezimalziffern und das Stellenwertsystem kennen.

- Repräsentation ist häufig mehrstufig, z.B.

Zahl:	Dezimalzahl	347
Ziffer:	4-bit binär	0011 0100 0111 (BCD)
Bit:	elektrische Spannung	0,1V 0,1V 2,5V 2,5V ...

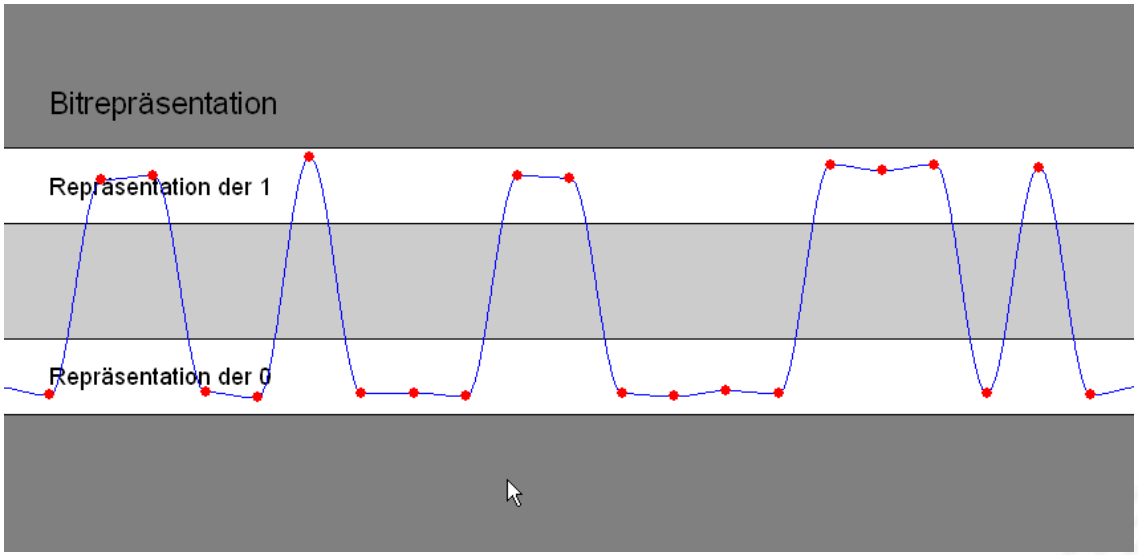
Repräsentation: Ebenen

In jeder (Abstraktions-) Ebene gibt es beliebig viele Alternativen der Repräsentation

- Auswahl der jeweils effizientesten Repräsentation
- unterschiedliche Repräsentationen je nach Ebene

- Beispiel: Repräsentation der Zahl $\pi = 3,1415\ldots$ im
  - x86 Prozessor — 80-bit Binärdaten, Spannungen
  - Hauptspeicher — 64-bit Binärdaten, Spannungen
  - Festplatte — codierte Zahl, magnetische Bereiche
  - CD-ROM — codierte Zahl, Land/Pits-Bereiche
  - Papier — Text, „3,14159265...“
  - ...

Repräsentation: digitale und analoge Welt



Beispiel: Binärwerte in 2,5 V CMOS-Technologie

K. von der Heide [Hei05]
Interaktives Skript T1, demobitrep

- Spannungsverlauf des Signals ist kontinuierlich
- Abtastung zu bestimmten Zeitpunkten
- Quantisierung über abgegrenzte Wertebereiche:
  - $0,0\,V \leq v(t) \leq 0,7\,V$: Interpretation als 0
  - $1,7\,V \leq v(t) \leq 2,5\,V$: Interpretation als 1
  - außerhalb und innerhalb: ungültige Werte

Information vs. Nachricht

- Aussagen
  - N1 Er besucht General Motors
  - N2 Unwetter am Alpenostrand
  - N3 Sie nimmt ihren Hut

- Alle Aussagen sind aber doppel/mehrdeutig:
  - N1 Firma? Militär?
  - N2 Alpen-Ostrand? Alpeno-Strand?
  - N3 tatsächlich oder im übertragenen Sinn?

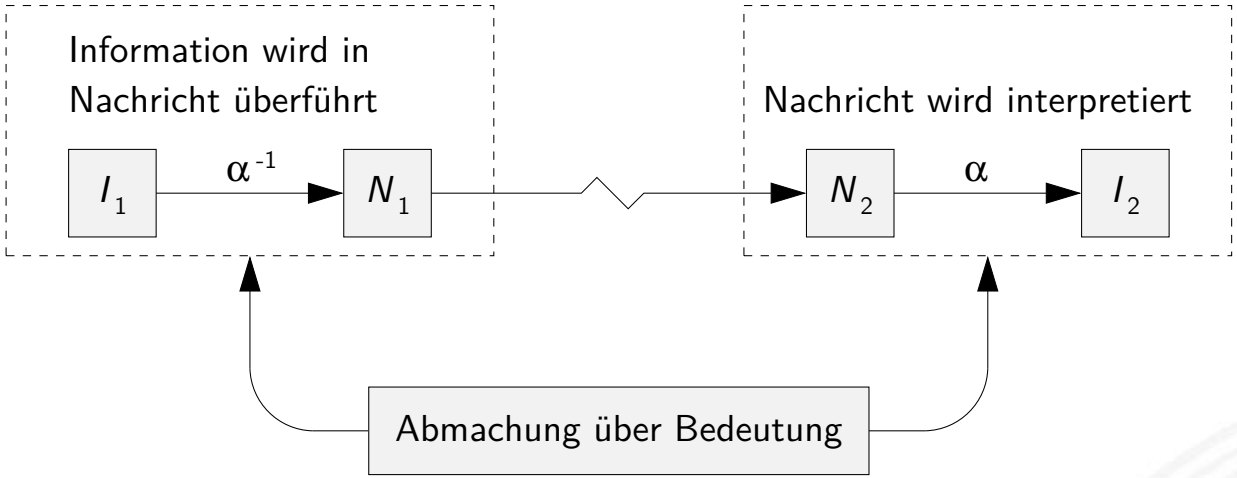
$\Rightarrow$ **Interpretation**: Es handelt sich um drei **Nachrichten**, die jeweils zwei verschiedene **Informationen** enthalten

Information vs. Nachricht (cont.)

- **Information**: Wissen um oder Kenntnis über Sachverhalte und Vorgänge – als Begriff nicht informationstheoretisch abgestützt, sondern an umgangssprachlicher Bedeutung orientiert
- **Nachricht**: Zeichen oder Funktionen, die Informationen zum Zweck der Weitergabe aufgrund bekannter oder unterstellter Abmachungen darstellen (DIN 44 300)

- Beispiel für eine Nachricht:
  Temperaturangabe in Grad Celsius oder Fahrenheit
- Die Nachricht ist also eine Darstellung von Informationen und nicht der Übermittlungsvorgang

Modell der Informationsübertragung

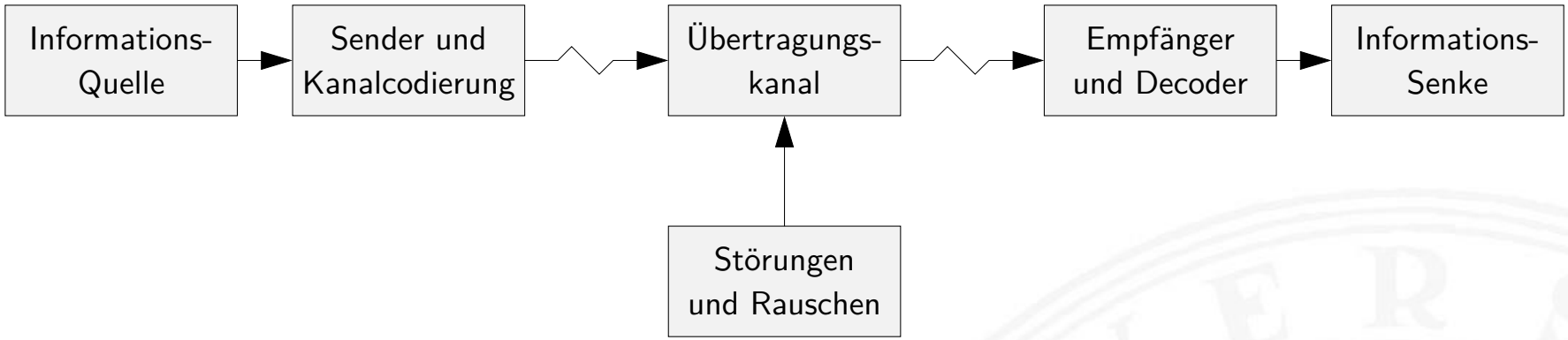


Beschreibung der **Informationsübermittlung**:

- Abbildung α^{-1} erzeugt Nachricht N_1 aus Information I_1
- Übertragung der Nachricht an den Zielort
- Interpretation α der Nachricht N_2 liefert die Information I_2

Modell der Informationsübertragung (cont.)

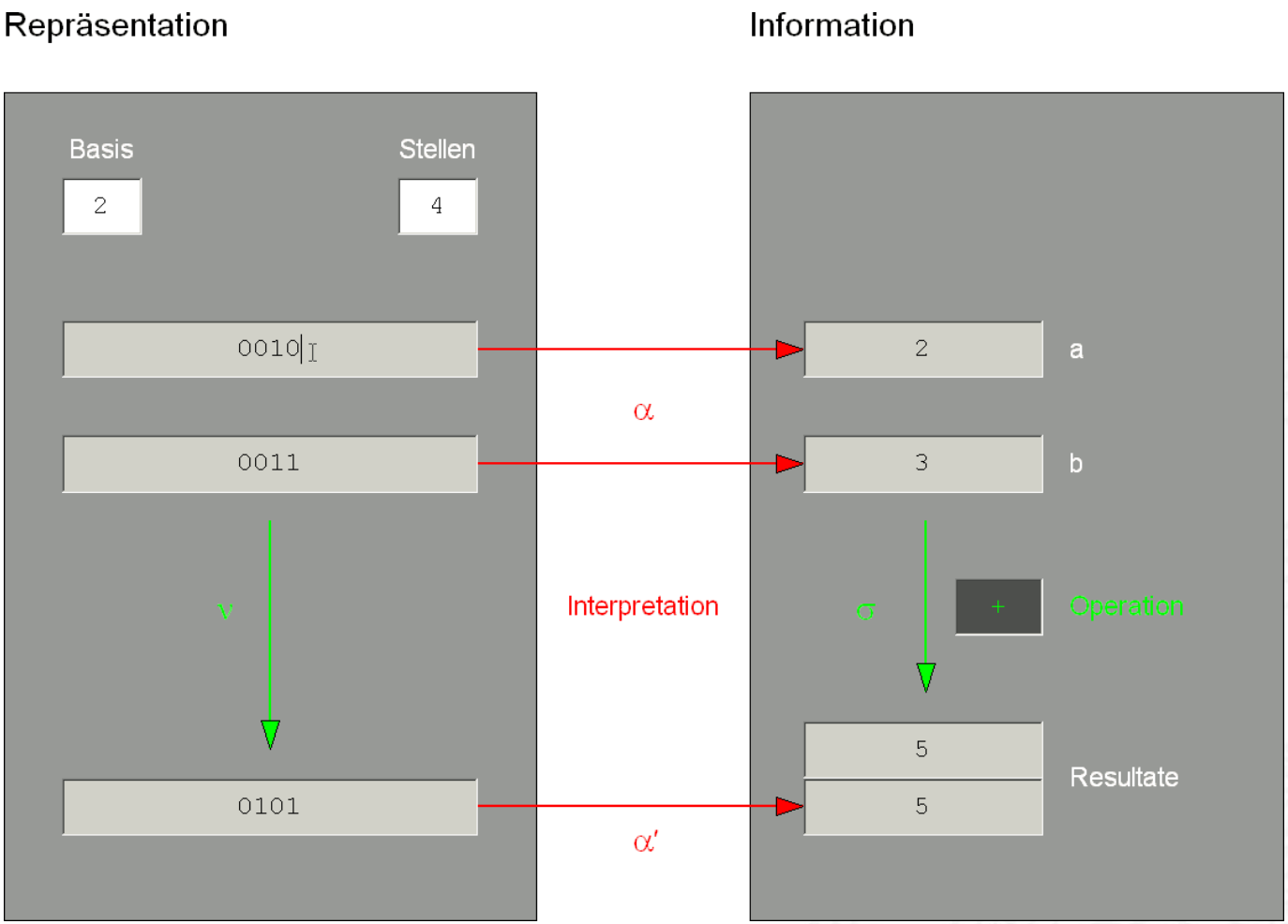
Nachrichtentechnisches Modell: **Störungen** bei der Übertragung



Beispiele

- Bitfehler beim Speichern
- Störungen beim Funkverkehr
- Schmutz oder Kratzer auf einer CD/DVD
- usw.

Verarbeitung von Information



Repräsentation natürlicher Zahlen durch Stellenwertsysteme

K. von der Heide [Hei05] Interaktives Skript T1, inforepres

Informationstreue

Ergibt α gefolgt von σ dasselbe wie ν gefolgt von α',
dann heißt ν **informationstreu** $\sigma(\alpha(r)) = \alpha'(\nu(r))$

- α' ist die Interpretation des Resultats der Operation ν
  häufig sind α und α' gleich, aber nicht immer

- ist σ injektiv, so nennen wir ν eine **Umschlüsselung**
  durch die Verarbeitung σ geht keine Information verloren
- ist ν injektiv, so nennen wir ν eine **Umcodierung**
- wenn σ innere Verknüpfung der Menge $\mathcal{J}$ und ν innere
  Verknüpfung der Menge $\mathcal{R}$, dann ist α ein **Homomorphismus**
  der algebraischen Strukturen $(\mathcal{J}, \sigma)$ und $(\mathcal{R}, \nu)$
- ist σ bijektiv, liegt ein **Isomorphismus** vor

Informationstreue (cont.)

Welche mathematischen Eigenschaften gelten bei der Informationsverarbeitung, in der gewählten Repräsentation?

Beispiele

- Gilt $x^2 \geq 0$?
  - float: ja
  - signed integer: nein
- Gilt $(x + y) + z = x + (y + z)$?
  - integer: ja
  - float: nein

  1.0E20 + (-1.0E20 + 3.14) = 0
- Details folgen später

Beschreibung von Information durch Zeichen

- **Zeichen**: engl. *character*
  Element z aus einer zur Darstellung von Information vereinbarten, einer Abmachung unterliegenden, endlichen Menge Z von Elementen

- Die Menge Z heißt **Zeichensatz** oder **Zeichenvorrat**
  engl. *character set*

- Beispiele
  - $\mathcal{Z}_1 = \{0, 1\}$
  - $\mathcal{Z}_2 = \{0, 1, 2, \ldots, 9, A, B, C, D, E, F\}$
  - $\mathcal{Z}_3 = \{\alpha, \beta, \gamma, \ldots, \omega\}$
  - $\mathcal{Z}_4 = \{CR, LF\}$

Beschreibung von Information durch Zeichen (cont.)

- **Numerischer Zeichensatz**: Zeichenvorrat aus Ziffern und/oder Sonderzeichen zur Darstellung von Zahlen
- **Alphanumerischer Zeichensatz**: Zeichensatz aus (mindestens) den Dezimalziffern und den Buchstaben des gewöhnlichen Alphabets, meistens auch mit Sonderzeichen (Leerzeichen, Punkt, Komma usw.)

Binärzeichen

- **Binärzeichen**: engl. *binary element, binary digit, bit*
  Jedes der Zeichen aus einem Vorrat / aus einer Menge von zwei Symbolen

- Beispiele
  - $\mathcal{Z}_1 = \{0, 1\}$
  - $\mathcal{Z}_2 = \{\text{high}, \text{low}\}$
  - $\mathcal{Z}_3 = \{\text{rot}, \text{grün}\}$
  - $\mathcal{Z}_4 = \{+, -\}$

Alphabet

- **Alphabet**: engl. *alphabet*
  Ein in vereinbarter Reihenfolge geordneter Zeichenvorrat $\mathcal{A} = \mathcal{Z}$

- Beispiele
  - $\mathcal{A}_1 = \{0, 1, 2, \ldots, 9\}$
  - $\mathcal{A}_2 = \{\text{Mo}, \text{Di}, \text{Mi}, \text{Do}, \text{Fr}, \text{Sa}, \text{So}\}$
  - $\mathcal{A}_3 = \{\text{A}, \text{B}, \text{C}, \ldots, \text{Z}\}$

Zeichenkette

- **Zeichenkette**: engl. *string*
  Eine Folge von Zeichen
- **Wort**: engl. *word*
  Eine Folge von Zeichen, die in einem gegebenen Zusammenhang als Einheit bezeichnet wird
- Worte mit 8 bit werden als **Byte** bezeichnet
- **Stelle**: engl. *position*
  Die Lage/Position eines Zeichens innerhalb einer Zeichenkette

- Beispiel
  - `s = H e l l o , w o r l d !`

Zahlen- und Zeichendarstellung in RSB

3. Natürliche Zahlen — engl. *integer numbers*
   Festkommazahlen — engl. *fixed point numbers*
   Gleitkommazahlen — engl. *floating point numbers*

4. Arithmetik

5. Aspekte der Textcodierung
   Ad-hoc Codierungen
   ASCII und ISO-8859-1
   Unicode

13. Pointer (Referenzen, Maschinenadressen)

Literatur

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*. 6. Auflage, Pearson Deutschland GmbH, 2014. ISBN 978–3–8689–4238–5

[Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript*. Universität Hamburg, FB Informatik, 2005. `tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1`

Gliederung

Gliederung (cont.)

Konzept der Zahl

„Das Messen ist der Ursprung der Zahl als Abstraktion der Anzahl von Objekten die man abzählen kann. . . “ [Ifr10]

Abstraktion zum:

- Zählen
- Speichern
- Rechnen

Eigenschaften eines Zahlensystems

- Zahlenbereich: kleinste und größte darstellbare Zahl?
- Darstellung negativer Werte?
- –"– gebrochener Werte?
- –"– sehr großer Werte?

- Unterstützung von Rechenoperationen?
  Addition, Subtraktion, Multiplikation, Division etc.
- Abgeschlossenheit unter diesen Operationen?

- Methode zur dauerhaften Speicherung/Archivierung?
- Sicherheit gegen Manipulation gespeicherter Werte?

Abstraktion: Verschiedene Symbole für eine Zahl

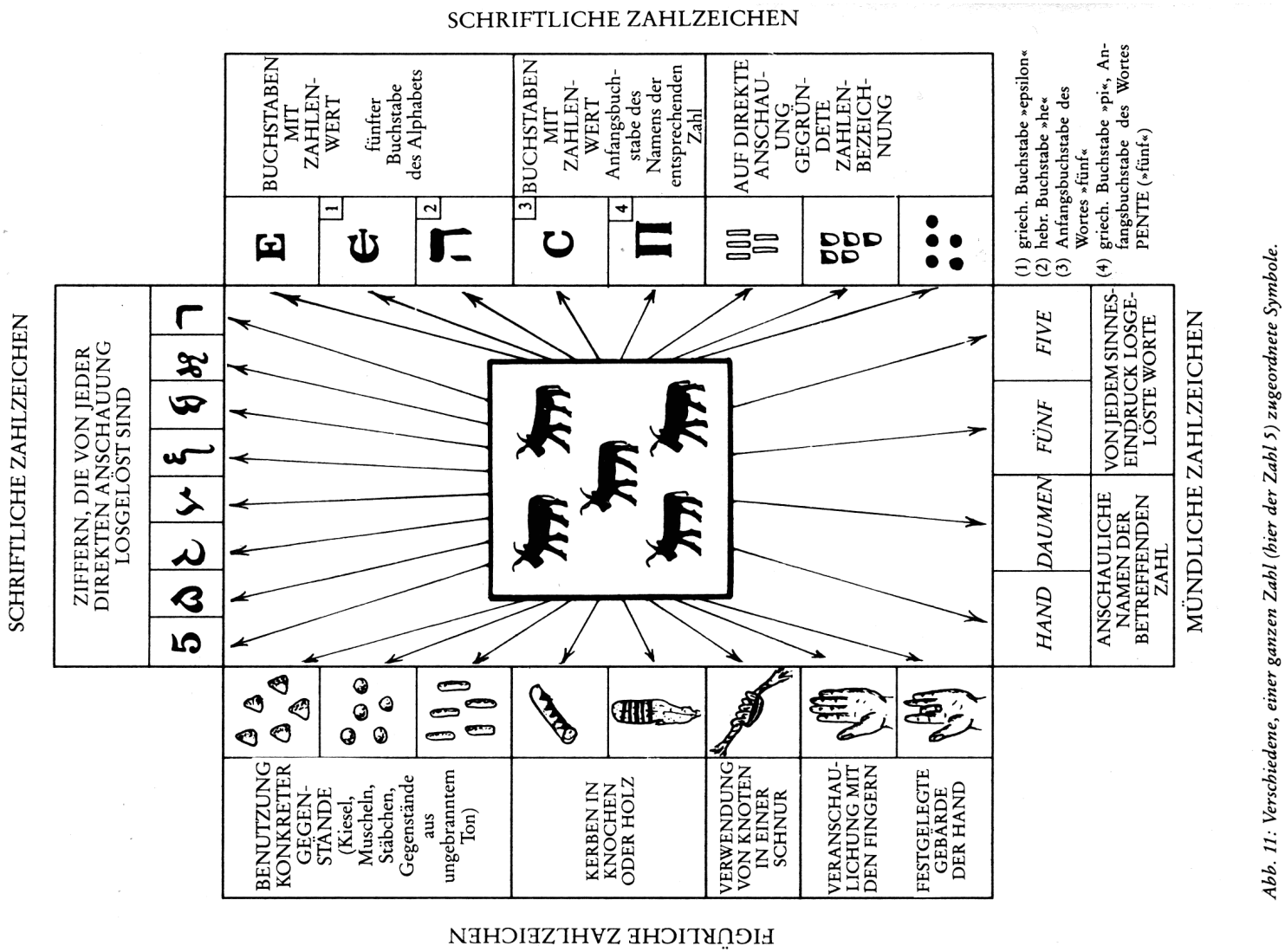


Abb. 11: Verschiedene, einer ganzen Zahl (hier der Zahl 5) zugeordnete Symbole.

[Ifr10]

Zählen mit den Fingern („digits“)

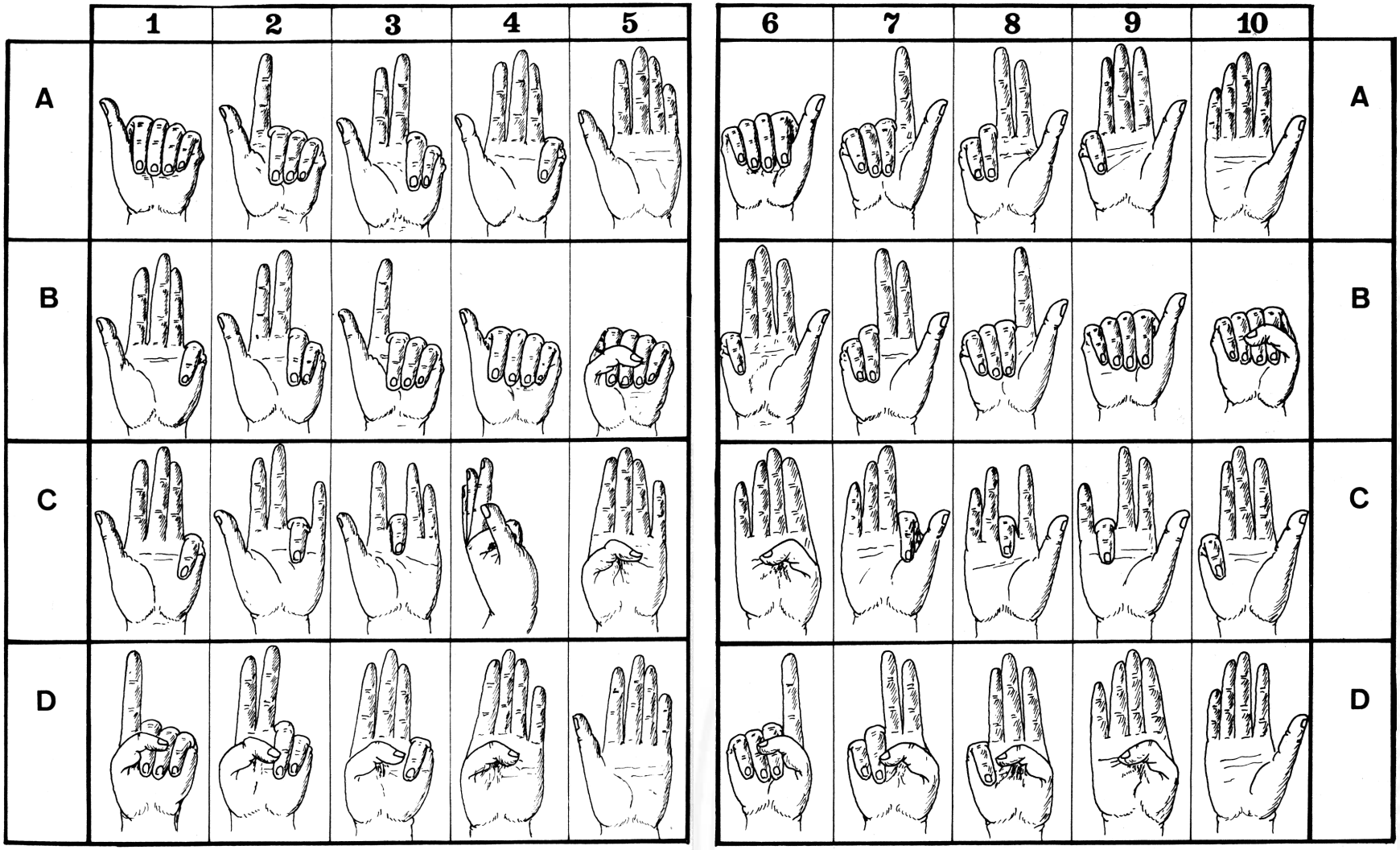


Abb. 12: Verschiedene Möglichkeiten des Zählens mit den Fingern.

[Ifr10]

Tonbörse: 15. Jh. v. Chr.

Gegenstände, Hammel und Ziegen betreffend
21 Mutterschafe
6 weibliche Lämmer
8 erwachsene Hammel
4 männliche Lämmer
6 Mutterziegen
1 Bock
(2) Jungziegen

Abb. 3: Eiförmige Tonbörse (46 mm × 62 mm × 50 mm), entdeckt in den Ruinen des Palastes von Nuzi (mesopotamische Stadt; ca. 15. Jh. v. Chr.).
(Harvard Semitic Museum, Cambridge. Katalognummer SMN 1854)

Kerbhölzer

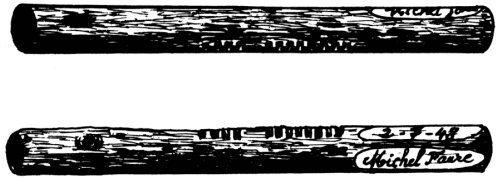


Abb. 58: Kerbhölzer aus Bäckereien in Frankreich, wie sie in kleinen Ortschaften auf dem Lande üblich waren.

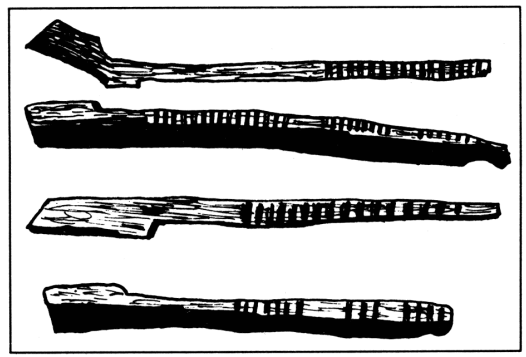


Abb. 59: Englische Kerbhölzer aus dem 13. Jahrhundert.
(Sammlung Society of Antiquaries, London; Zeichnung nach Menninger 1957/58, II, 42)

Knotenschnüre

[lfr10]

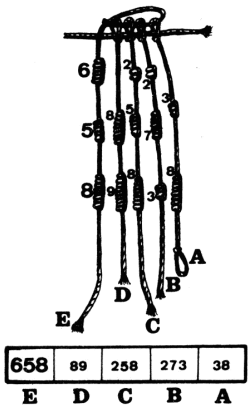


Abb. 66: Interpretation eines quipu: Die Zahl 658 auf der Schnur E ist gleich der Summe der Zahlen auf den Schnüren A, B, C und D. Dieses Bündel ist das erste an einem peruanischen quipu.
(American Museum of Natural History, New York, B 8713; vgl. Leland Locke 1923)

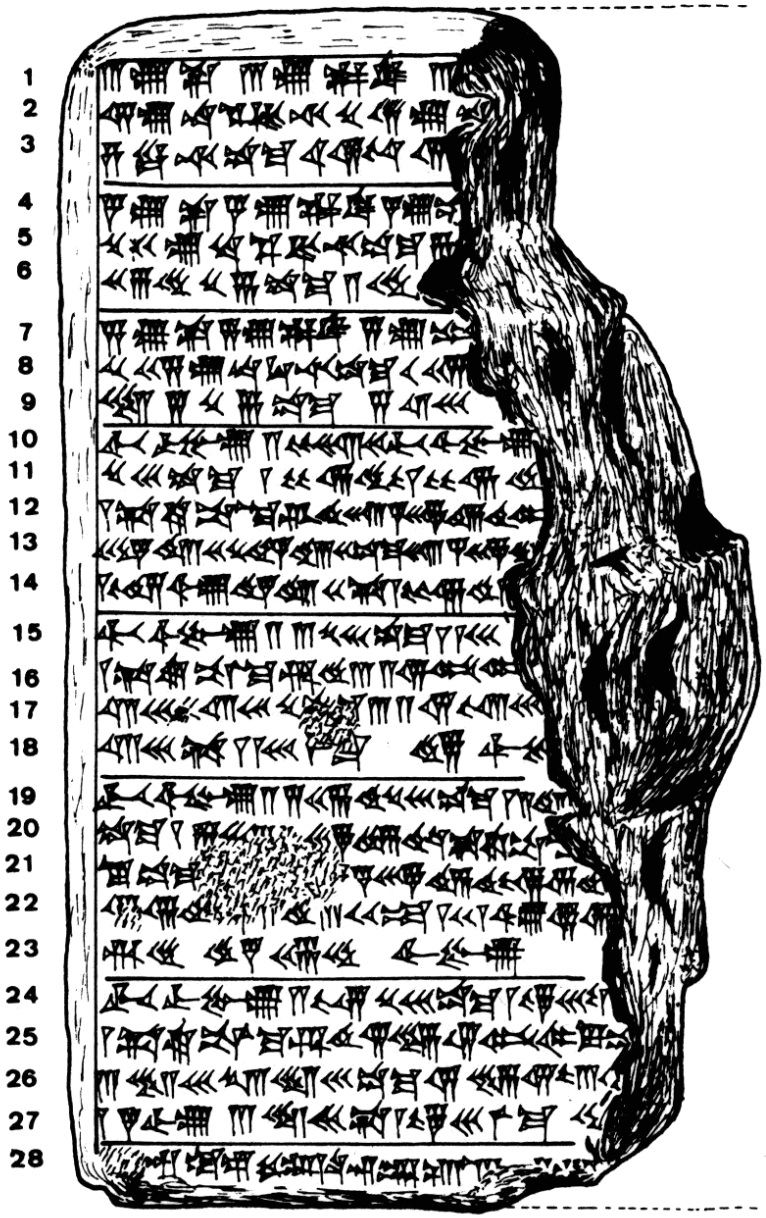
Rechnen: Römische Ziffern

- Ziffern: I=1, V=5, X=10, L=50, C=100, D=500, M=1000
- Werte eins bis zehn: I, II, III, IV, V, VI, VII, VIII, IX, X
- Position der Ziffern ist signifikant:
  - nach Größe der Ziffernsymbole sortiert, größere stehen links
  - andernfalls Abziehen der kleineren von der größeren Ziffer
  - IV=4, VI=6, XL=40, LXX=70, CM=900
- heute noch in Gebrauch: Jahreszahlen, Seitennummern usw.
  Beispiele: MDCCCXIII=1813, MMIX=2009

– keine Symbole zur Darstellung großer Zahlen

– Rechenoperationen so gut wie unmöglich

Babylon: Einführung der Null, 3 Jh. v. Chr.



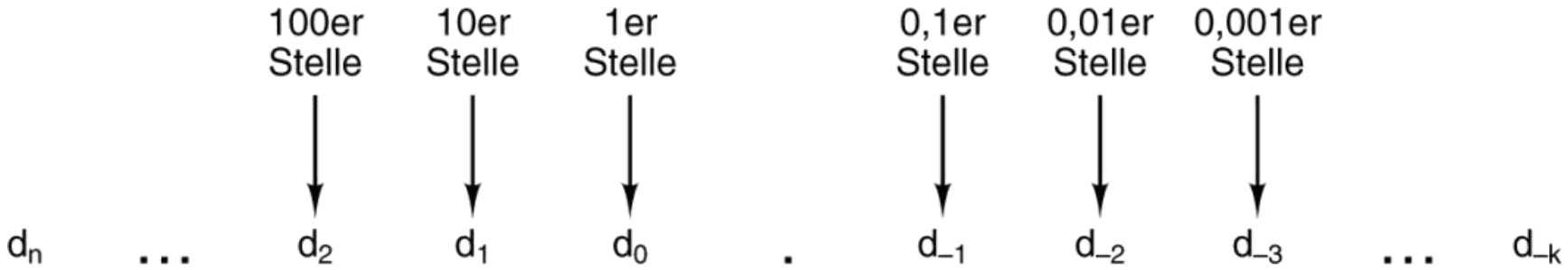
[Ifr10]

Abb. 289: Mathematische Tafel aus Uruk; sie wurde bei Schwarzgrabungen gefunden und stammt aus dem 2. oder 3. Jh. v. Chr. Es handelt sich um eines der ältesten bekannten Zeugnisse für die Verwendung der babylonischen Null.
(Musée du Louvre, Taf. AO 6484, Rückseite; Thureau-Dangin 1922, Nr. 33, Taf. 62; 1938, 76-81. Unveröffentl. Kopie d. Verf.)

Babylon: Sexagesimalsystem

- vor ungefähr 4 000 Jahren, erstes **Stellenwertsystem**
- Basis 60
- zwei Symbole: | $= 1$ und < $= 10$
- Einritzen gerader und gewinkelter Striche auf Tontafeln
- Null bekannt, aber nicht mitgeschrieben
  Leerzeichen zwischen zwei Stellen
- Beispiele
  - | | | | | 5
  - <<| | | 23
  - | <<< $90 = 1 \cdot 60 + 3 \cdot 10$
  - | <<| $3\,621 = 1 \cdot 3\,600 + 0 \cdot 60 + 2 \cdot 10 + 1$
- für Zeitangaben und Winkeleinteilung heute noch in Gebrauch

Dezimalsystem



$$\text{Zahl} = \sum_{i=-k}^{n} d_i \times 10^i$$

[TA14]

- das im Alltag gebräuchliche Zahlensystem
- Einer, Zehner, Hunderter, Tausender usw.
- Zehntel, Hundertstel, Tausendstel usw.

Stellenwertsystem („Radixdarstellung“)

- Wahl einer geeigneten Zahlenbasis b („Radix“)
  - 10: Dezimalsystem
  - 16: Hexadezimalsystem (Sedezimalsystem)
  - 2: Dualsystem
- Menge der entsprechenden Ziffern $\{0, 1, \ldots, b-1\}$
- inklusive einer besonderen Ziffer für den Wert Null
- Auswahl der benötigten Anzahl n von Stellen

$$|z| = \sum_{i=0}^{n-1} a_i \cdot b^i$$

b Basis a_i Koeffizient an Stelle i

- universell verwendbar, für beliebig große Zahlen

Dualsystem

- Stellenwertsystem zur Basis 2
- braucht für gegebene Zahl ca. dreimal mehr Stellen als Basis 10
- für Menschen daher unbequem
  besser Oktal- oder Hexadezimalschreibweise, s.u.

- technisch besonders leicht zu implementieren weil nur zwei Zustände unterschieden werden müssen

  z.B. zwei Spannungen, Ströme, Beleuchtungsstärken
  siehe: *2.6 Informationsverarbeitung – Binärzeichen*, Folie 110

+ robust gegen Rauschen und Störungen
+ einfache und effiziente Realisierung von Arithmetik

Dualsystem: Potenztabelle

Stelle	Wert im Dualsystem	Wert im Dezimalsystem
2^0	1	1
2^1	10	2
2^2	100	4
2^3	1000	8
2^4	1 0000	16
2^5	10 0000	32
2^6	100 0000	64
2^7	1000 0000	128
2^8	1 0000 0000	256
2^9	10 0000 0000	512
2^{10}	100 0000 0000	1 024
2^{11}	1000 0000 0000	2 048
2^{12}	1 0000 0000 0000	4 096
...	...	...

Dualsystem: Beispiele

- Basis 2
- Zeichensatz ist $\{0, 1\}$
- Beispiele:

$$
\begin{aligned}
0_2 &= 0_{10} \\
1_2 &= 1_{10} \\
11_2 &= 3_{10} && 2^1 + 2^0 \\
11\,0100_2 &= 52_{10} && 2^5 + 2^4 + 2^2 \\
1111\,1110_2 &= 254_{10} && 2^8 + 2^7 + \ldots + 2^2 + 2^1
\end{aligned}
$$

Addition im Dualsystem

- funktioniert genau wie im Dezimalsystem
- Addition mehrstelliger Zahlen erfolgt stellenweise
- Additionsmatrix:

+	0	1
0	0	1
1	1	10

- Beispiel

	1011	0011	= 179
+	0011	1001	= 57
Ü	11	11	11
	1110	1100	= 236

Multiplikation im Dualsystem

- funktioniert genau wie im Dezimalsystem
- $p = a \cdot b$ mit Multiplikator a und Multiplikand b
- Multiplikation von a mit je einer Stelle des Multiplikanten b
- Addition der Teilterme

- Multiplikationsmatrix ist sehr einfach:

$\cdot$	0	1
0	0	0
1	0	1

Multiplikation im Dualsystem (cont.)

- Beispiel

	10110011	·	1101	$= 179 \cdot 13$	$= 2327$
	10110011		1		$= 1001\,0001\,0111$
	10110011		1		$= 0x917$
	00000000		0		
	10110011		1		
Ü	11101111				
	100100010111				

Oktalsystem

- Basis 8
- Zeichensatz ist $\{0, 1, 2, 3, 4, 5, 6, 7\}$
- C-Schreibweise mit führender 0 als Präfix:
  - $0001 = 1_{10}$
    $0013 = 11_{10} = 1 \cdot 8 + 3$
    $0375 = 253_{10} = 3 \cdot 64 + 7 \cdot 8 + 5$
    usw.

  $\Rightarrow$ Hinweis: führende 0 in C für Dezimalzahlen unmöglich!
- für Menschen leichter lesbar als Dualzahlen
- Umwandlung aus/vom Dualsystem durch Zusammenfassen bzw. Ausschreiben von je drei Bits:

  $00 = 000, 01 = 001, 02 = 010, 03 = 011,$
  $04 = 100, 05 = 101, 06 = 110, 07 = 111$

Hexadezimalsystem

- Basis 16
- Zeichensatz ist $\{0, 1, 2, \ldots, 9, A, B, C, D, E, F\}$
- C-Schreibweise mit Präfix 0x – Klein- oder Großbuchstaben
  - $0x00000001 = 1_{10}$
    $0x000000fe = 254_{10} = 15 \cdot 16 + 14$
    $0x0000ffff = 65\,535_{10} = 15 \cdot 4\,096 + 15 \cdot 256 + 15 \cdot 16 + 15$
    $0xcafebabe = \ldots$ erstes Wort in Java Class-Dateien
    usw.
- viel leichter lesbar als entsprechende Dualzahl
- Umwandlung aus/vom Dualsystem durch Zusammenfassen bzw. Ausschreiben von je vier Bits:

  $0x0 = 0000, 0x1 = 0001, 0x2 = 0010, \ldots, 0x9 = 1001,$
  $0xA = 1010, 0xB = 1011, 0xC = 1100,$
  $0xD = 1101, 0xE = 1110, 0xF = 1111$

Beispiel: Darstellungen der Zahl 2019

Binär

$$\begin{array}{cccccccccccc}
1 & & 1 & & 1 & & 1 & & 1 & & 1 & & 0 & & 0 & & 0 & & 1 & & 1 \\
1\cdot 2^{10} & + & 1\cdot 2^{9} & + & 1\cdot 2^{8} & + & 1\cdot 2^{7} & + & 1\cdot 2^{6} & + & 1\cdot 2^{5} & + & 0\cdot 2^{4} & + & 0\cdot 2^{3} & + & 0\cdot 2^{2} & + & 1\cdot 2^{1} & + & 1\cdot 2^{0} \\
1\,024 & + & 512 & + & 256 & + & 128 & + & 64 & + & 32 & + & 0 & + & 0 & + & 0 & + & 2 & + & 1
\end{array}$$

Oktal

$$\begin{array}{cccccccc}
3 & & 7 & & 4 & & 3 \\
3\cdot 8^{3} & + & 7\cdot 8^{2} & + & 4\cdot 8^{1} & + & 3\cdot 8^{0} \\
1\,536 & + & 448 & + & 32 & + & 3
\end{array}$$

Dezimal

$$\begin{array}{cccccccc}
2 & & 0 & & 1 & & 9 \\
2\cdot 10^{3} & + & 0\cdot 10^{2} & + & 1\cdot 10^{1} & + & 9\cdot 10^{0} \\
2\,000 & + & 0 & + & 10 & + & 9
\end{array}$$

Hexadezimal

$$\begin{array}{cccccccc}
7 & & \mathrm{E} & & 3 \\
7\cdot 16^{2} & + & \mathrm{E}\cdot 16^{1} & + & 3\cdot 16^{0} \\
1\,792 & + & 224 & + & 3
\end{array}$$

Umrechnung Dual-/Oktal-/Hexadezimalsystem

- Beispiele

Hexadezimal		1		9		4		8	.	B		6			
Binär	0	0	0	1	1	0	0	1	0	1	0	0	1	0	0	0	.	1	0	1	1	0	1	1	0	0
Oktal				1		4		5		1		0	.	5		5		4								

Hexadezimal		7		B		A		3	.	B		C		4	
Binär	0	1	1	1	1	0	1	1	1	0	1	0	0	0	1	1	.	1	0	1	1	1	1	0	0	0	1	0	0
Oktal				7		5		6		4		3	.	5		7		0									

- Gruppieren von jeweils 3 bzw. 4 Bits
- bei Festkomma vom Dezimalpunkt aus nach außen

Umrechnung zwischen verschiedenen Basen

- Menschen rechnen im Dezimalsystem
- Winkel- und Zeitangaben auch im Sexagesimalsystem Basis: 60
- Digitalrechner nutzen (meistens) Dualsystem

- Algorithmen zur Umrechnung notwendig
- Exemplarisch Vorstellung von drei Varianten:

1. vorberechnete Potenztabellen
2. Divisionsrestverfahren
3. Horner-Schema

Umwandlung über Potenztabellen

Vorgehensweise für Integerzahlen

1.a Subtraktion des größten Vielfachen einer Potenz des Zielsystems von der umzuwandelnden Zahl
– gemäß der vorberechneten Potenztabelle

1.b Notation dieses größten Vielfachen (im Zielsystem)

- solange der der Rest der Zahl $\neq 0$, dann Wiederhole:

2.a Subtraktion wiederum des größten Vielfachen vom verbliebenen Rest

2.b Addition dieses Vielfachen (im Zielsystem)

Potenztabellen Dual/Dezimal

Stelle_2	Wert_{10}
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1 024
2^{11}	2 048
2^{12}	4 096
...	

Stelle_{10}	Wert_2
10^0	1
10^1	1010
10^2	110 0100
10^3	11 1110 1000
10^4	10 0111 0001 0000
10^5	0x1 86 A0
10^6	0xF 42 40
10^7	0x98 96 80
10^8	0x5 F5 E1 00
10^9	0x3B 9A CA 00
10^{10}	0x2 54 0B E4 00
10^{11}	0x17 48 76 E8 00
10^{12}	0xE8 D4 A5 10 00
...	

Potenztabellen: Beispiel

- Umwandlung Dezimal- in Dualzahl

$Z = (163)_{10}$

163			
$-\ 128$	2^7		$1000\,0000$
35			
$-\ 32$	2^5	$+$	$10\,0000$
3			
$-\ 2$	2^1	$+$	10
1			
$-\ 1$	2^0	$+$	1
0			$1010\,0011$

$Z = (163)_{10} \leftrightarrow (1010\,0011)_2$

Potenztabellen: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1010\,0011)_2$

1010 0011		
− 110 0100	$1 \cdot 10^2$	100
0011 1111		
− 11 1100	$6 \cdot 10^1$	+ 60
11		
− 11	$3 \cdot 10^0$	+ 3
0		163

$Z = (1010\,0011)_2 \leftrightarrow (163)_{10}$

Divisionsrestverfahren

- Division der umzuwandelnden Zahl im Ausgangssystem durch die Basis des Zielsystems
- Erneute Division des ganzzahligen Ergebnisses (ohne Rest) durch die Basis des Zielsystems, bis kein ganzzahliger Divisionsrest mehr bleibt
- Beispiel

$$
\begin{array}{rrll}
163 : 2 = & 81 & \text{Rest } 1 & 2^0 \\
81 : 2 = & 40 & \text{Rest } 1 & \vdots \\
40 : 2 = & 20 & \text{Rest } 0 & \\
20 : 2 = & 10 & \text{Rest } 0 & \\
10 : 2 = & 5 & \text{Rest } 0 & \\
5 : 2 = & 2 & \text{Rest } 1 & \uparrow \text{ Leserichtung} \\
2 : 2 = & 1 & \text{Rest } 0 & \vdots \\
1 : 2 = & 0 & \text{Rest } 1 & 2^7
\end{array}
$$

$$(163)_{10} \leftrightarrow (1010\,0011)_2$$

Divisionsrestverfahren: Beispiel

- Umwandlung Dual- in Dezimalzahl

$Z = (1010\,0011)_2$

$$
\begin{aligned}
(1010\,0011)_2 : (1010)_2 &= 1\,0000 \quad \text{Rest} \quad (11)_2 \mathrel{\widehat{=}} 3 \quad 10^0 \\
(1\,0000)_2 : (1010)_2 &= 1 \quad \text{Rest} \quad (110)_2 \mathrel{\widehat{=}} 6 \quad 10^1 \\
(1)_2 : (1010)_2 &= 0 \quad \text{Rest} \quad (1)_2 \mathrel{\widehat{=}} 1 \quad 10^2
\end{aligned}
$$

$Z = (1010\,0011)_2 \leftrightarrow (163)_{10}$

Hinweis: Division in Basis b folgt

Divisionsrestverfahren: Beispiel (cont.)

- Umwandlung Dezimal- in Dualzahl

$Z = (1492)_{10}$

$1492 : 2 =$	746	Rest 0	2^0
$746 : 2 =$	373	Rest 0	$\vdots$
$373 : 2 =$	186	Rest 1	
$186 : 2 =$	93	Rest 0	
$93 : 2 =$	46	Rest 1	
$46 : 2 =$	23	Rest 0	
$23 : 2 =$	11	Rest 1	
$11 : 2 =$	5	Rest 1	
$5 : 2 =$	2	Rest 1	$\uparrow$ Leserichtung
$2 : 2 =$	1	Rest 0	$\vdots$
$1 : 2 =$	0	Rest 1	2^{10}

$Z = (1492)_{10} \leftrightarrow (101\,1101\,0100)_2$

Divisionsrestverfahren: Algorithmus

Algorithmus | darzustellende Zahl x: 123 | Basis q: 2

rechentechnisch

```
a := x
while  a > 0
   y_n := a mod q
   a  := a div q
end
```

n = 1

a = 123

(a > 0) = 1

a mod 2 = 1

a div 2 = 61

Takt

0000000000000001

Resultat

K. von der Heide [Hei05] Interaktives Skript T1 stellen2stellen

Horner-Schema

- Darstellung einer Potenzsumme durch ineinander verschachtelte Faktoren

$$|z| = \sum_{i=0}^{n-1} a_i \cdot b^i = (\ldots((a_{n-1} \cdot b + a_{n-2}) \cdot b + a_{n-3}) \cdot b + \ldots + a_1) \cdot b + a_0$$

Vorgehensweise:

- Darstellung der umzuwandelnden Zahl im Horner-Schema
- Durchführung der auftretenden Multiplikationen und Additionen im Zielsystem

Horner-Schema: Beispiel

- Umwandlung Dezimal- in Dualzahl
  1. Darstellung als Potenzsumme
     $Z = (163)_{10} = (1 \cdot 10 + 6) \cdot 10 + 3$
  2. Faktoren und Summanden im Zielzahlensystem
     $(10)_{10} \leftrightarrow (1010)_2$
     $(6)_{10} \leftrightarrow (110)_2$
     $(3)_{10} \leftrightarrow (11)_2$
     $(1)_{10} \leftrightarrow (1)_2$
  3. Arithmetische Operationen

$$
\begin{array}{rl}
1 \cdot 1010 = & 1010 \\
+ & 110 \\
\hline
1\,0000 \cdot 1010 = & 1010\,0000 \\
+ & 11 \\
\hline
& 1010\,0011
\end{array}
$$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl
  1. Darstellung als Potenzsumme
     $Z = (1010\,0011)_2 =$
     $((((((1 \cdot 10_2 + 0) \cdot 10_2 + 1) \cdot 10_2 + 0) \cdot 10_2 + 0) \cdot 10_2 + 0) \cdot 10_2 + 1) \cdot 10_2 + 1$
  2. Faktoren und Summanden im Zielzahlensystem
     $(10)_2 \leftrightarrow (2)_{10}$
     $(1)_2 \leftrightarrow (1)_{10}$
     $(0)_2 \leftrightarrow (0)_{10}$

Horner-Schema: Beispiel (cont.)

3. Arithmetische Operationen

$$
\begin{array}{r}
1 \cdot 2 = 2 \\
+\,0 \\
\hline
2 \cdot 2 = 4 \\
+\,1 \\
\hline
5 \cdot 2 = 10 \\
+\;\;0 \\
\hline
10 \cdot 2 = 20 \\
+\;\;0 \\
\hline
20 \cdot 2 = 40 \\
+\;\;0 \\
\hline
40 \cdot 2 = 80 \\
+\;\;1 \\
\hline
81 \cdot 2 = 162 \\
+\;\;1 \\
\hline
163
\end{array}
$$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl
  $Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$Z = (1011\,1011\,0111)_2 = (2\,999)_{10}$

$1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,1\,1$

$1 + 2 \cdot 0 = 1$

$0 + 2 \cdot 1 = 2$

$1 + 2 \cdot 2 = 5$

$1 + 2 \cdot 5 = 11$

$1 + 2 \cdot 11 = 23$

$0 + 2 \cdot 23 = 46$

$1 + 2 \cdot 46 = 93$

$1 + 2 \cdot 93 = 187$

$0 + 2 \cdot 187 = 374$

$1 + 2 \cdot 374 = 749$

$1 + 2 \cdot 749 = 1\,499$

$1 + 2 \cdot 1\,499 = 2\,999$

Zahlenbereich bei fester Wortlänge

Anzahl der Bits	Zahlenbereich jeweils von 0 bis $(2^n - 1)$		
4-bit	2^4	=	16
8-bit	2^8	=	256
10-bit	2^{10}	=	1 024
12-bit	2^{12}	=	4 096
16-bit	2^{16}	=	65 536
20-bit	2^{20}	=	1 048 576
24-bit	2^{24}	=	16 777 216
32-bit	2^{32}	=	4 294 967 296
48-bit	2^{48}	=	281 474 976 710 656
64-bit	2^{64}	=	18 446 744 073 709 551 616

Präfixe

Für die vereinfachte Schreibweise von großen bzw. sehr kleinen Werten ist die Präfixangabe als Abkürzung von Zehnerpotenzen üblich. Beispiele:

- Lichtgeschwindigkeit: $300\,000\,\mathrm{Km/s} = 30\,\mathrm{cm/ns}$
- Ruheenergie des Elektrons: $0{,}51\,\mathrm{MeV}$
- Strukturbreite heutiger Mikrochips: $12\,\mathrm{nm}$
- usw.

Es gibt entsprechende Präfixe auch für das Dualsystem.
Dazu werden Vielfache von $2^{10} = 1024 \approx 1000$ verwendet.

Präfixe für Einheiten im Dezimalsystem

Faktor	Name	Symbol
10^{24}	yotta	Y
10^{21}	zetta	Z
10^{18}	exa	E
10^{15}	peta	P
10^{12}	tera	T
10^{9}	giga	G
10^{6}	mega	M
10^{3}	kilo	K
10^{2}	hecto	h
10^{1}	deka	da

Faktor	Name	Symbol
10^{-24}	yocto	y
10^{-21}	zepto	z
10^{-18}	atto	a
10^{-15}	femto	f
10^{-12}	pico	p
10^{-9}	nano	n
10^{-6}	micro	μ
10^{-3}	milli	m
10^{-2}	centi	c
10^{-1}	dezi	d

Präfixe für Einheiten im Dualsystem

Faktor	Name	Symbol	Langname
2^{80}	yobi	Yi	yottabinary
2^{70}	zebi	Zi	zettabinary
2^{60}	exbi	Ei	exabinary
2^{50}	pebi	Pi	petabinary
2^{40}	tebi	Ti	terabinary
2^{30}	gibi	Gi	gigabinary
2^{20}	mebi	Mi	megabinary
2^{10}	kibi	Ki	kilobinary

Beispiele:

1 kibibit	=	1 024 bit
1 kilobit	=	1 000 bit
1 megibit	=	1 048 576 bit
1 gibibit	=	1 073 741 824 bit

IEC-60027-2, Letter symbols to be used in electrical technology

Präfixe für Einheiten im Dualsystem

In der Praxis werden die offiziellen Präfixe nicht immer sauber verwendet. Meistens ergibt sich die Bedeutung aber aus dem Kontext. Bei Speicherbausteinen sind Zweierpotenzen üblich, bei Festplatten dagegen die dezimale Angabe.

- DRAM-Modul mit 4 GB Kapazität: gemeint sind 2^{32} Bytes
- Flash-Speicherkarte 32 GB Kapazität: gemeint sind 2^{35} Bytes

- Festplatte mit Angabe 2 TB Kapazität: typisch $2 \cdot 10^{12}$ Bytes
- die tatsächliche angezeigte verfügbare Kapazität ist oft geringer, weil das jeweilige Dateisystem Platz für seine eigenen Verwaltungsinformationen belegt.

Festkommadarstellung

Darstellung von **gebrochenen Zahlen** als Erweiterung des Stellenwertsystems durch Erweiterung des Laufindex zu negativen Werten:

$$|z| = \sum_{i=0}^{n-1} a_i \cdot b^i + \sum_{i=-\infty}^{i=-1} a_i \cdot b^i$$

$$= \sum_{i=-\infty}^{n-1} a_i \cdot b^i$$

mit $a_i \in N$ und $0 \leq a_i < b$.

- Der erste Summand bezeichnet den ganzzahligen Anteil, während der zweite Summand für den gebrochenen Anteil steht.

Nachkommastellen im Dualsystem

- $2^{-1} = 0{,}5$
  $2^{-2} = 0{,}25$
  $2^{-3} = 0{,}125$
  $2^{-4} = 0{,}0625$
  $2^{-5} = 0{,}03125$
  $2^{-6} = 0{,}015625$
  $2^{-7} = 0{,}0078125$
  $2^{-8} = 0{,}00390625$
  ...
- alle Dualbrüche sind im Dezimalsystem exakt darstellbar (d.h. mit endlicher Wortlänge)
- dies gilt umgekehrt nicht

Nachkommastellen im Dualsystem (cont.)

- gebrochene Zahlen können je nach Wahl der Basis evtl. nur als unendliche periodische Brüche dargestellt werden
- insbesondere erfordern viele endliche Dezimalbrüche im Dualsystem unendliche periodische Brüche
- Beispiel: Dezimalbrüche, eine Nachkommastelle

B=10	B=2
0,1	$0,0\overline{0011}$
0,2	$0,\overline{0011}$
0,3	$0,0\overline{1001}$
0,4	$0,\overline{0110}$
0,5	$0,1$
0,6	$0,\overline{1001}$
0,7	$0,1\overline{0110}$
0,8	$0,\overline{1100}$
0,9	$0,1\overline{1100}$

B=2	B=10
0,001	0,125
0,010	0,25
0,011	0,375
0,100	0,5
0,101	0,625
0,110	0,75
0,111	0,875

Umrechnung: Dezimalbruch nach Dual

Potenztabelle zur Umrechnung

- Potenztabelle

$2^{-1} = 0{,}5$	$2^{-7} = 0{,}0078125$
$2^{-2} = 0{,}25$	$2^{-8} = 0{,}00390625$
$2^{-3} = 0{,}125$	$2^{-9} = 0{,}001953125$
$2^{-4} = 0{,}0625$	$2^{-10} = 0{,}0009765625$
$2^{-5} = 0{,}03125$	$2^{-11} = 0{,}00048828125$
$2^{-6} = 0{,}015625$	$2^{-12} = 0{,}000244140625$

- Beispiel: Dezimal 0,3

  Berechnung durch Subtraktion der Werte

$$
\begin{aligned}
(0,3)_{10} &= 0\cdot 2^{-1} + 1\cdot 2^{-2} + 0\cdot 2^{-3} + 0\cdot 2^{-4} + 1\cdot 2^{-5} + 1\cdot 2^{-6} + \ldots \\
&= 2^{-2} + 2^{-5} + 2^{-6} + 2^{-9} + \ldots \\
&= (0,0\overline{1001})_2
\end{aligned}
$$

Umrechnung: Dezimalbruch nach Dual (cont.)

Divisionsrestverfahren

- statt Division: bei Nachkommastellen Multiplikation $\cdot 2$
  - man nimmt den Dezimalbruch immer mit 2 mal
  - Resultat < 1: eine 0 an den Dualbruch anfügen
    –"– ≥ 1: eine 1 –"–
    und den ganzzahligen Anteil streichen: $-1{,}0$
  - Ende, wenn Ergebnis 1,0 (wird zu 0)
    –"– wenn Rest sich wiederholt $\Rightarrow$ Periode
- Beispiel: Dezimal 0,59375

$$
\begin{array}{lll}
2 \cdot 0,59375 = 1,1875 & \rightarrow 1 & 2^{-1} \\
2 \cdot 0,1875 = 0,375 & \rightarrow 0 & \vdots \\
2 \cdot 0,375 = 0,75 & \rightarrow 0 & \downarrow \text{ Leserichtung} \\
2 \cdot 0,75 = 1,5 & \rightarrow 1 & \vdots \\
2 \cdot 0,5 = 1,0 & \rightarrow 1 & 2^{-5}
\end{array}
$$

$(0,59375)_{10} \leftrightarrow (0,10011)_2$

Darstellung negativer Zahlen

Drei gängige Varianten zur Darstellung negativer Zahlen

1. Betrag und Vorzeichen
2. Exzess-Codierung (Offset-basiert)
3. **Komplementdarstellung**

- Integerrechnung häufig im Zweierkomplement
- Gleitkommadarstellung mit Betrag und Vorzeichen
- –"– Exponent als Exzess-Codierung

Betrag und Vorzeichen

- Auswahl eines Bits als Vorzeichenbit
- meistens das MSB (engl. *most significant bit*)
- restliche Bits als Dualzahl interpretiert
- Beispiel für 4-bit Wortbreite:

0000	$+0$	1000	-0
0001	$+1$	1001	-1
0010	$+2$	1010	-2
0011	$+3$	1011	-3
0100	$+4$	1100	-4
0101	$+5$	1101	-5
0110	$+6$	1110	-6
0111	$+7$	1111	-7

– doppelte Codierung der Null: $+0$, -0

– Rechenwerke für Addition/Subtraktion aufwändig

Exzess-Codierung (Offset-basiert)

- einfache Um-Interpretation der Binärcodierung

$$z = c - offset$$

- z vorzeichenbehafteter Wert (Zahlenwert)
- c binäre Ganzzahl (Code)
- beliebig gewählter Offset

– Null wird also nicht mehr durch 000 . . . 0 dargestellt

\+ Größenvergleich zweier Zahlen bleibt einfach

- Anwendung: Exponenten im IEEE 754 Gleitkommaformat
- und für einige Audioformate

Exzess-Codierung: Beispiele

Bitmuster	Binärcode	Exzess-8	Exzess-6
0000	0	-8	-6
0001	1	-7	-5
0010	2	-6	-4
0011	3	-5	-3
0100	4	-4	-2
0101	5	-3	-1
0110	6	-2	0
0111	7	-1	1
1000	8	0	2
1001	9	1	3
1010	10	2	4
1011	11	3	5
1100	12	4	6
1101	13	5	7
1110	14	6	8
1111	15	7	9

$$z = c - offset$$

b-Komplement

Definition: das b-**Komplement** einer Zahl z ist

$$\begin{aligned} K_b(z) &= b^n - z, \quad \text{für} \quad z \neq 0 \\ &= 0, \quad \text{für} \quad z = 0 \end{aligned}$$

- b: die Basis (des Stellenwertsystems)
- n: Anzahl der zu berücksichtigenden Vorkommastellen
- mit anderen Worten: $K_b(z) + z = b^n$

- Stellenwertschreibweise

$$z = -a_{n-1} \cdot b^{n-1} + \sum_{i=-m}^{n-2} a_i \cdot b^i$$

- Dualsystem: 2-Komplement
- Dezimalsystem: 10-Komplement

b-Komplement: Beispiele

$$
\begin{array}{llll}
b = 10 & n = 4 & K_{10}(3\,763)_{10} & = 10^4 - 3\,763 = 6\,237_{10} \\
 & n = 2 & K_{10}(0,3763)_{10} & = 10^2 - 0,3763 = 99,6237_{10} \\
 & n = 0 & K_{10}(0,3763)_{10} & = 10^0 - 0,3763 = 0,6237_{10} \\
b = 2 & n = 2 & K_2(10,01)_2 & = 2^2 - 10,01_2 = 01,11_2 \\
 & n = 8 & K_2(10,01)_2 & = 2^8 - 10,01_2 = 1111\,1101,11_2
\end{array}
$$

$(b-1)$-Komplement

Definition: das $(b-1)$-**Komplement** einer Zahl z ist

$$\begin{aligned} K_{b-1}(z) &= b^n - z - b^{-m}, && \text{für} \quad z \neq 0 \\ &= 0, && \text{für} \quad z = 0 \end{aligned}$$

- b: die Basis des Stellenwertsystems
- n: Anzahl der zu berücksichtigenden Vorkommastellen
- m: Anzahl der Nachkommastellen
- mit anderen Worten: $K_{b-1}(z) + z + b^{-m} = b^n$

- Dualsystem: 1-Komplement
- Dezimalsystem: 9-Komplement

$(b-1)$-Komplement / b-Komplement: Trick

$$K_{b-1}(z) = b^n - b^{-m} - z, \quad \text{für} \quad z \neq 0$$

- im Fall $m = 0$ gilt offenbar $K_b(z) = K_{b-1}(z) + 1$

$\Rightarrow$ das $(b-1)$-Komplement kann sehr einfach berechnet werden:
es werden einfach die einzelnen Bits/Ziffern invertiert.

- Dualsystem:
  - 1-Komplement: 1100 1001
  - alle Bits invertieren: 0011 0110
- Dezimalsystem:
  - 9-Komplement: 24 453
  - alle Ziffern invertieren: 75 546
  - $0 \leftrightarrow 9$ $1 \leftrightarrow 8$ $2 \leftrightarrow 7$ $3 \leftrightarrow 6$ $4 \leftrightarrow 5$
  - Summe: $99\,999 = 100\,000 - 1$

$\Rightarrow$ das b-Komplement kann sehr einfach berechnet werden:
es werden einfach die einzelnen Bits/Ziffern invertiert
und 1 an der niedrigsten Stelle aufaddiert.

$(b-1)$-Komplement / b-Komplement: Trick (cont.)

- Dualsystem:

2-Komplement	1100 1001
Bits invertieren $+1$	0011 0111
Summe:	1 0000 0000

- Dezimalsystem:

10-Komplement	24 453
Ziffern invertieren $+1$	75 547
$0\leftrightarrow9$ $1\leftrightarrow8$ $2\leftrightarrow7$ $3\leftrightarrow6$ $4\leftrightarrow5$	
Summe:	100 000

Subtraktion mit b-Komplement

- bei Rechnung mit fester Stellenzahl n gilt:

$$K_b(z) + z = b^n \quad = 0$$

  weil b^n gerade nicht mehr in n Stellen hineinpasst
- also gilt für die Subtraktion auch:

$$x - y = x + K_b(y)$$

$\Rightarrow$ Subtraktion kann also durch Addition des b-Komplements ersetzt werden

- und für Integerzahlen gilt außerdem

$$x - y = x + K_{b-1}(y) + 1$$

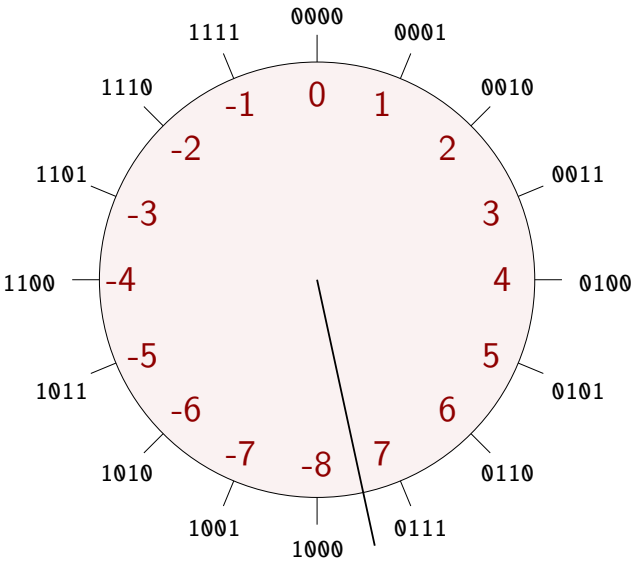
Subtraktion mit Einer- und Zweierkomplement

- Subtraktion ersetzt durch Addition des Komplements

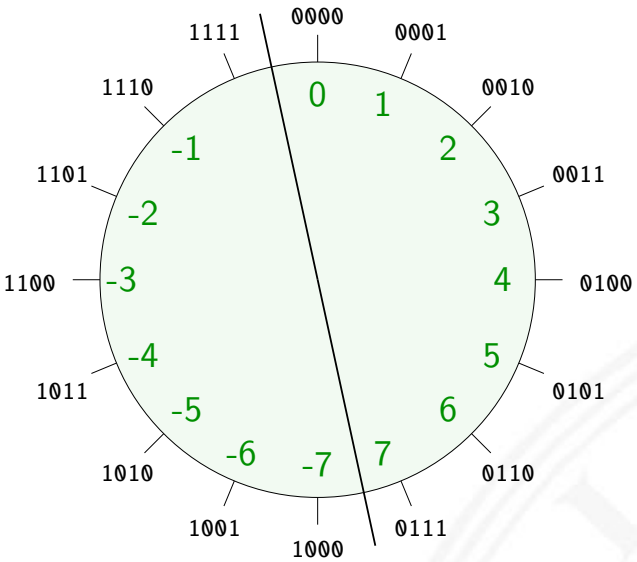
Dezimal	1-Komplement	2-Komplement
10	0000 1010	0000 1010
+(-3)	1111 1100	1111 1101
+7	1 0000 0110	1 0000 0111
Übertrag:	addieren +1	verwerfen
	0000 0111	0000 0111

Veranschaulichung: Zahlenkreis

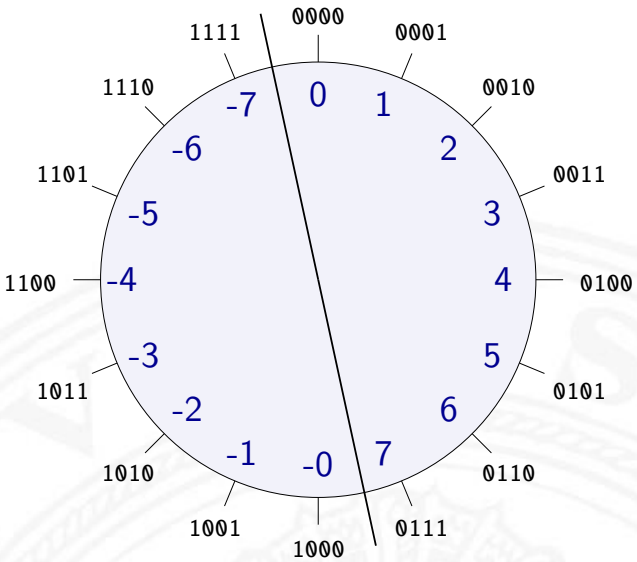
Beispiel für 4-bit Zahlen



2-Komplement



1-Komplement



Betrag+Vorzeichen

- Komplement-Arithmetik als Winkeladdition (siehe *4 Arithmetik*)
- Web-Anwendung: *Visualisierung im Zahlenkreis* (JavaScript, see: [Kor16])

Darstellung negativer Zahlen: Beispiele

N Dezimal	+N Binär	-N VZ+Betrag	-N 1-Komplement	-N 2-Komplement	-N Exzess-128
1	0000 0001	1000 0001	1111 1110	1111 1111	0111 1111
2	0000 0010	1000 0010	1111 1101	1111 1110	0111 1110
3	0000 0011	1000 0011	1111 1100	1111 1101	0111 1101
4	0000 0100	1000 0100	1111 1011	1111 1100	0111 1100
5	0000 0101	1000 0101	1111 1010	1111 1011	0111 1011
6	0000 0110	1000 0110	1111 1001	1111 1010	0111 1010
7	0000 0111	1000 0111	1111 1000	1111 1001	0111 1001
8	0000 1000	1000 1000	1111 0111	1111 1000	0111 1000
9	0000 1001	1000 1001	1111 0110	1111 0111	0111 0111
10	0000 1010	1000 1010	1111 0101	1111 0110	0111 0110
20	0001 0100	1001 0100	1110 1011	1110 1100	0110 1100
30	0001 1110	1001 1110	1110 0001	1110 0010	0110 0010
40	0010 1000	1010 1000	1101 0111	1101 1000	0101 1000
50	0011 0010	1011 0010	1100 1101	1100 1110	0100 1110
60	0011 1100	1011 1100	1100 0011	1100 0100	0100 0100
70	0100 0110	1100 0110	1011 1001	1011 1010	0011 1010
80	0101 0000	1101 0000	1010 1111	1011 0000	0011 0000
90	0101 1010	1101 1010	1010 0101	1010 0110	0010 0110
100	0110 0100	1110 0100	1001 1011	1001 1100	0001 1100
127	0111 1111	1111 1111	1000 0000	1000 0001	0000 0001
128	—	—	—	1000 0000	0000 0000
MSB	0	1	1	1	0

Gleitkommaformat

Wie kann man „wissenschaftliche“ Zahlen darstellen?

- Masse der Sonne $1,989 \cdot 10^{30}$ Kg
- Ladung eines Elektrons 0,000 000 000 000 000 000 16 C
- Anzahl der Atome pro Mol 602 300 000 000 000 000 000 000

...

Darstellung im Stellenwertsystem?

- gleichzeitig sehr große und sehr kleine Zahlen notwendig
- entsprechend hohe Zahl der Vorkomma- und Nachkommastellen
- durchaus möglich (Java3D: 256-bit Koordinaten)
- aber normalerweise sehr unpraktisch
- typische Messwerte haben nur ein paar Stellen Genauigkeit

Gleitkomma: Motivation

Grundidee: **halblogarithmische Darstellung einer Zahl**

- Vorzeichen ($+1$ oder -1)
- *Mantisse* als normale Zahl im Stellenwertsystem
- *Exponent* zur Angabe der Größenordnung

$$z = sign \cdot mantisse \cdot basis^{exponent}$$

- handliche Wertebereiche für Mantisse und Exponent
- arithmetische Operationen sind effizient umsetzbar
- Wertebereiche für ausreichende Genauigkeit wählen

Hinweis: rein logarithmische Darstellung wäre auch möglich, aber Addition/Subtraktion sind dann sehr aufwändig.

Gleitkomma: Dezimalsystem

$$z = (-1)^s \cdot m \cdot 10^e$$

- s Vorzeichenbit
- m Mantisse als Festkomma-Dezimalzahl
- e Exponent als ganze Dezimalzahl

- Schreibweise in C/Java: ⟨*Vorzeichen*⟩ ⟨*Mantisse*⟩ `E` ⟨*Exponent*⟩

`6.023E23`	$6,023 \cdot 10^{23}$	Avogadro-Zahl
`1.6E-19`	$1,6 \cdot 10^{-19}$	Elementarladung des Elektrons

Gleitkomma: Beispiel für Zahlenbereiche

Stellen		Zahlenbereich	
Mantisse	Exponent	$0 \leftarrow$	$\rightarrow \infty$
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

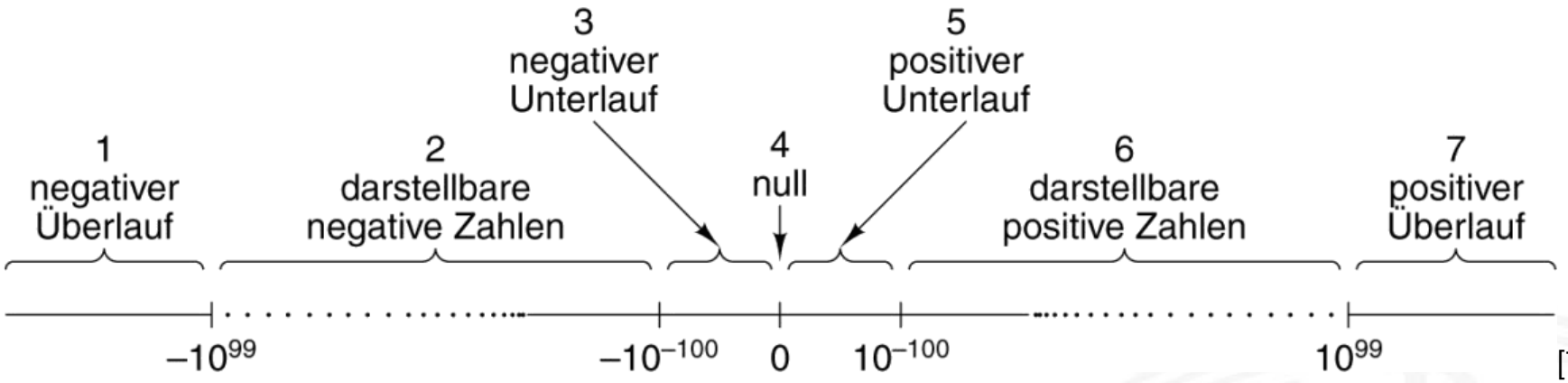
Gleitkomma: Historie

- 1937 Zuse: Z1 mit 22-bit Gleitkomma-Datenformat
- 195x Verbreitung von Gleitkomma-Darstellung für numerische Berechnungen
- 1980 Intel 8087: erster Koprozessor-Chip, ca. 45 000 Transistoren, ca. 50K FLOPS/s
- 1985 IEEE 754 Standard für Gleitkomma
- 1989 Intel 486 mit integriertem Koprozessor
- 1995 Java-Spezifikation fordert IEEE 754
- 1996 ASCI-RED: 1 TFLOPS (9 152 Pentium Pro)
- 2008 Roadrunner: 1 PFLOPS (12 960 Cell)
- 2018 Summit: 148,6 PFLOPS

...

FLOPS := Floating-Point Operations Per Second

Gleitkomma: Zahlenbereiche



[TA14]

- Darstellung üblicherweise als Betrag+Vorzeichen
- negative und positive Zahlen gleichberechtigt (symmetrisch)
- separate Darstellung für den Wert Null
- sieben Zahlenbereiche: siehe Grafik
- relativer Abstand benachbarter Zahlen bleibt ähnlich
  (vgl. dagegen Integer: 0/1, 1/2, 2/3, . . . , 65 535/65 536, . . .)

Gleitkomma: Normalisierte Darstellung

$$z = (-1)^s \cdot m \cdot 10^e$$

- diese Darstellung ist bisher nicht eindeutig:

  $123 \cdot 10^0 = 12,3 \cdot 10^1 = 1,23 \cdot 10^2 = 0,123 \cdot 10^3 = \quad \ldots$

normalisierte Darstellung

- Exponent anpassen, bis Mantisse im Bereich $1 \leq m < b$ liegt

$\Rightarrow$ Darstellung ist dann eindeutig

$\Rightarrow$ im Dualsystem: erstes Vorkommabit ist dann 1
und muss nicht explizit gespeichert werden

- evtl. zusätzlich sehr kleine Zahlen nicht-normalisiert

IEEE 754 Standard

bis 1985 ein Wildwuchs von Gleitkomma-Formaten:

- unterschiedliche Anzahl Bits in Mantisse und Exponent
- Exponent mit Basis 2, 10, oder 16
- diverse Algorithmen zur Rundung
- jeder Hersteller mit eigener Variante
- Numerische Algorithmen nicht portabel

1985: Publikation des Standards IEEE 754 zur Vereinheitlichung

- klare Regeln, auch für Rundungsoperationen
- große Akzeptanz, mittlerweile der universale Standard
- 2008: IEEE 754-2008 mit 16- und 128-bit Formaten

Details: unter anderem in `en.wikipedia.org/wiki/IEEE_754` oder in Goldberg [Gol91]

IEEE 754: *float* und *double*

- 32-bit Format: einfache Genauigkeit (*single precision*, *float*)

  Bits 1 8 23

- 64-bit Format: doppelte Genauigkeit (*double precision*, *double*)

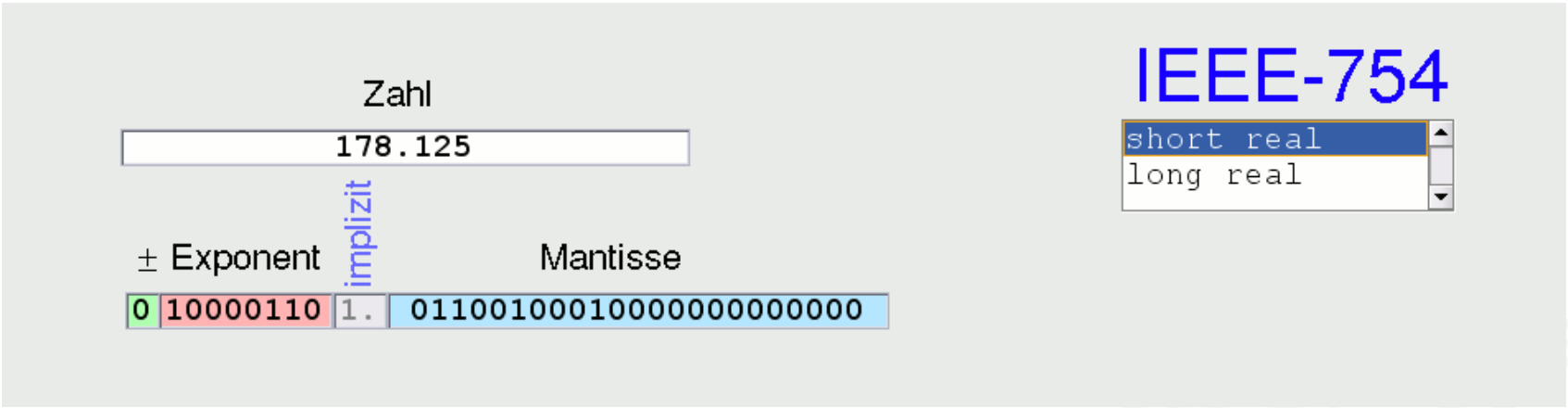
  Bits 1 11 52

- Mantisse als normalisierte Dualzahl: $1 \leq m < 2$
- Exponent in Exzess-127 bzw. Exzess-1023 Codierung
  Exp.-Bits: $00\ldots001$ bis $11\ldots110$
- einige Sonderwerte: Null ($+0$, -0), NaN, Infinity
  Exp.-Bits: $00\ldots000$ und $11\ldots111$

IEEE 754: Zahlenbereiche

Eigenschaft	einfache	doppelte Genauigkeit
Bits im Vorzeichen	1	1
Bits im Exponenten	8	11
Bits in der Mantisse	23	52
Bits insgesamt	32	64
Exponentensystem	Exzess-127	Exzess-1023
Exponentenbereich	$-126 \ldots +127$	$-1022 \ldots +1023$
kleinste normalisierte Zahl	2^{-126}	2^{-1022}
größte –"–	$\approx 2^{128}$	$\approx 2^{1024}$
$\widehat{=}$ Dezimalbereich	$\approx 10^{-38} \ldots 10^{38}$	$\approx 10^{-308} \ldots 10^{308}$
kleinste nicht normalisierte Zahl	$\approx 10^{-45}$	$\approx 10^{-324}$

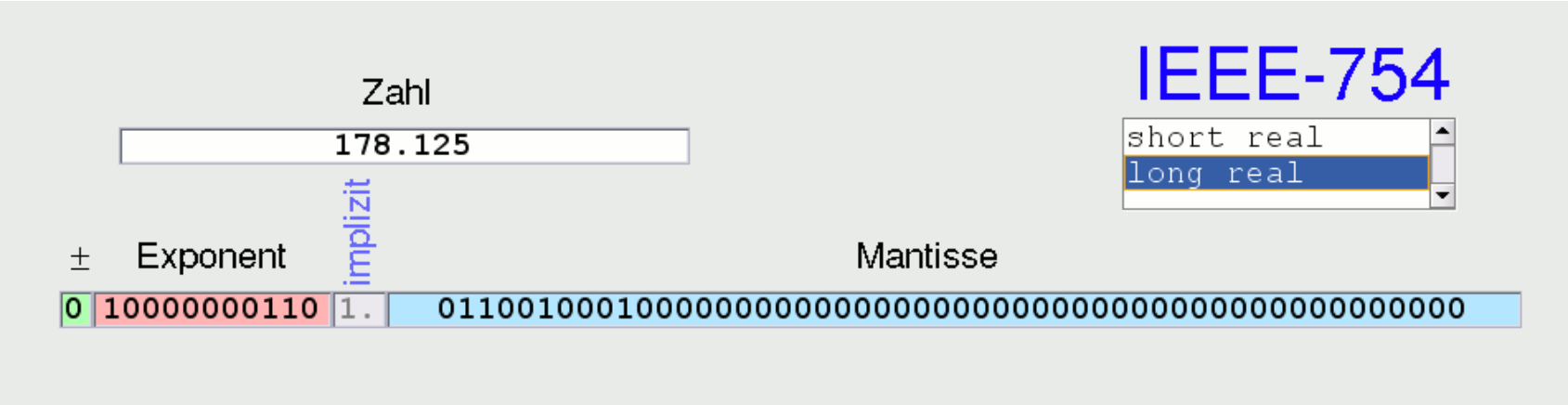
Matlab-Demo: demoieee754



- Zahlenformat wählen (float=short real, double=long real)
- Dezimalzahl in oberes Textfeld eingeben
- Mantisse/Exponent/Vorzeichen in unteres Textfeld eingeben
- andere Werte werden jeweils aktualisiert

K. von der Heide [Hei05], Interaktives Skript T1, demoieee754

Matlab-Demo: demoieee754 (cont.)



- Genauigkeit bei float: 23+1 bits, ca. 6 . . . 7 Dezimalstellen
- Genauigkeit bei double: 52+1 bits, ca. 16 Dezimalstellen

Erinnerung: $\log_2(10) = \ln(10)/\ln(2) \approx 3,322$

Beispiele: float

- 1-bit Vorzeichen 8-bit Exponent (Exzess-127), 23-bit Mantisse

  $z = (-1)^s \cdot 2^{(eeee\,eeee - 127)} \cdot 1, mmmm\,mmmm\,mmmm\ldots mmm$

- 1 1000 0000 1110 0000 0000 0000 0000 000

  $z = -1 \cdot 2^{(128-127)} \cdot (1 + 0,5 + 0,25 + 0,125 + 0)$

  $= -1 \cdot 2 \cdot 1,875 = -3,750$

- 0 1111 1110 0001 0011 0000 0000 0000 000

  $z = +1 \cdot 2^{(254-127)} \cdot (1 + 2^{-4} + 2^{-7} + 2^{-8})$

  $= 2^{127} \cdot 1,07421875 = 1,8276885 \cdot 10^{38}$

Beispiele: float (cont.)

$$z = (-1)^s \cdot 2^{(eeee\ eeee - 127)} \cdot 1, mmmm\ mmmm\ mmmm \ldots mmm$$

- 1 0000 0001 0000 0000 0000 0000 0000 000

$$z = -1 \cdot 2^{(1-127)} \cdot (1 + 0 + 0 + \ldots + 0)$$

$$= -1 \cdot 2^{-126} \cdot 1,0 = -1,17549435 \cdot 10^{-38}$$

- 0 0111 1111 0000 0000 0000 0000 0000 001

$$z = +1 \cdot 2^{(127-127)} \cdot (1 + 2^{-23})$$

$$= 1 \cdot (1 + 0,00000012) = 1,00000012$$

Gleitkomma: Addition, Subtraktion

Addition von Gleitkommazahlen $y = a_1 + a_2$

- Skalierung des betragsmäßig kleineren Summanden
- Erhöhen des Exponenten, bis $e_1 = e_2$ gilt
- gleichzeitig entsprechendes Skalieren der Mantisse $\Rightarrow$ schieben
- Achtung: dabei verringert sich die effektive Genauigkeit des kleineren Summanden

- anschließend Addition/Subtraktion der Mantissen
- ggf. Normalisierung des Resultats

- Beispiele in den Übungen

Gleitkomma-Addition: Beispiel

$a = 9,725 \cdot 10^7 \quad b = 3,016 \cdot 10^6$

$$
\begin{aligned}
y &= (a + b) \\
&= (9,725 \cdot 10^7 + 0,3016 \cdot 10^7) && \text{Angleichung der Exponenten} \\
&= (9,725 + 0,3016) \cdot 10^7 && \text{Distributivgesetz} \\
&= (10,0266) \cdot 10^7 && \text{Addition der Mantissen} \\
&= 1,00266 \cdot 10^8 && \text{Normalisierung} \\
\\
&= 1,003 \cdot 10^8 && \text{Runden bei fester Stellenzahl}
\end{aligned}
$$

- normalerweise nicht informationstreu !

Achtung: Auslöschung

Probleme bei Subtraktion/Addition zweier Gleitkommazahlen

Fall 1 Exponenten stark unterschiedlich

- kleinere Zahl wird soweit skaliert, dass von der Mantisse (fast) keine gültigen Bits übrigbleiben
- kleinere Zahl geht verloren, bzw. Ergebnis ist sehr ungenau
- Beispiel: `1.0E20 + 3.14159 = 1.0E20`

Fall 2 Exponenten gleich, Mantissen fast gleich

- fast alle Bits der Mantisse löschen sich aus
- Resultat hat nur noch wenige Bits effektiver Genauigkeit

Gleitkomma: Multiplikation, Division

Multiplikation von Gleitkommazahlen $y = a_1 \cdot a_2$

- Multiplikation der Mantissen und Vorzeichen

  Anmerkung: Vorzeichen s_i ist hier -1^{sBit}
  Berechnung als $sBit = sBit_1$ XOR $sBit_2$

- Addition der Exponenten
- ggf. Normalisierung des Resultats

$$y = (s_1 \cdot s_2) \cdot (m_1 \cdot m_2) \cdot b^{e_1+e_2}$$

Division entsprechend:

- Division der Mantissen und Vorzeichen
- Subtraktion der Exponenten
- ggf. Normalisierung des Resultats

$$y = (s_1 / s_2) \cdot (m_1 / m_2) \cdot b^{e_1-e_2}$$

IEEE 754: Infinity *Inf*, Not-a-Number *NaN*, ±0

- schnelle Verarbeitung großer Datenmengen
- Statusabfrage nach jeder einzelnen Operation unbequem
- trotzdem Hinweis auf aufgetretene Probleme wichtig

⇒ *Inf* (*infinity*): spezieller Wert für plus/minus Unendlich
Beispiele: $2/0, -3/0, arctan(\pi)$ usw.

⇒ *NaN* (*not-a-number*): spezieller Wert für ungültige Operation
Beispiele: $\sqrt{-1}, arcsin(2,0), Inf/Inf$ usw.

IEEE 754: Infinity *Inf*, Not-a-Number *NaN*, ±0 (cont.)

normalisiert	V	$0 < Exp < Max$	jedes Bitmuster
denormalisiert	V	00...0	jedes Bitmuster $\neq$ 00...0
0	V	00...0	00...0
Inf	V	11...1	00...0
NaN	V	11...1	jedes Bitmuster $\neq$ 00...0

- Rechnen mit *Inf* funktioniert normal: $0/Inf = 0$
- NaN für undefinierte Werte: sqrt(-1), arcsin(2.0) ...
- jede Operation mit *NaN* liefert wieder *NaN*

IEEE 754: Infinity *Inf*, Not-a-Number *NaN*, ±0 (cont.)

- Beispiele

```
 0 / 0                  = NaN
 1 / 0                  = Infinity
-1 / 0                  = -Infinity
 1 / Infinity           = 0.0
Infinity + Infinity     = Infinity
Infinity + -Infinity    = NaN
Infinity * -Infinity    = -Infinity
Infinity + NaN          = NaN
Infinity * 0            = NaN
sqrt(2)                 = 1.414213562373095
sqrt(-1)                = NaN
0 + NaN                 = NaN
NaN == NaN              = false                    Achtung
Infinity > NaN          = false                    Achtung
```

ULP: Unit in the last place

- die Differenz zwischen den beiden Gleitkommazahlen, die einer gegebenen Zahl am nächsten liegen
- diese beiden Werte unterscheiden sich im niederwertigsten Bit der Mantisse $\Rightarrow$ Wertigkeit des LSB
- daher ein Maß für die erreichbare Genauigkeit

- IEEE 754 fordert eine Genauigkeit von 0,5 ULP für die elementaren Operationen: Addition, Subtraktion, Multiplikation, Division, Quadratwurzel
  $=$ der bestmögliche Wert
- gute Mathematik-Software garantiert ≤ 1 ULP auch für höhere Funktionen: Logarithmus, Sinus, Cosinus usw.
- Progr.sprachenunterstützung, z.B. `java.lang.Math.ulp( double d )`

Rundungsfehler

- sorgfältige Behandlung von Rundungsfehlern essenziell
- teilweise Berechnung mit zusätzlichen Schutzstellen
- dadurch Genauigkeit ± 1 ULP für alle Funktionen
- ziemlich komplexe Sache

- in dieser Vorlesung nicht weiter vertieft
- beim Einsatz von numerischen Algorithmen essenziell

Datentypen in der Praxis: Maschinenworte

- die meisten Rechner sind für eine Wortlänge optimiert
- 8-bit, 16-bit, 32-bit, 64-bit ... Maschinen
- die jeweils typische Länge eines Integerwertes
- und meistens auch von Speicheradressen
- zusätzlich Teile oder Vielfache der Wortlänge unterstützt

- 32-bit Rechner
  - Wortlänge für Integerwerte ist 32-bit
  - adressierbarer Speicher ist 2^{32} Bytes (4 GiB)
  - bereits zu knapp für speicherhungrige Applikationen
- inzwischen sind 64-bit Rechner bei PCs/Laptops Standard
- kleinere Wortbreiten: *embedded*-Systeme (Steuerungsrechner), Mobilgeräte etc.

Datentypen auf Maschinenebene

- gängige Prozessoren unterstützen mehrere Datentypen
- entsprechend der elementaren Datentypen in C, Java ...
- `void*` ist ein **Pointer** (Referenz, Speicheradresse)
- Beispiel für die Anzahl der Bytes:

C Datentyp	DEC Alpha	typ. 32-bit	Intel IA-32 (x86)
int	4	4	4
long int	8	4	4
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	8	10/12
void *	8	4	4

Datentypen auf Maschinenebene (cont.)

Abhängigkeiten (!)

- Prozessor
- Betriebssystem
- Compiler

www.agner.org/optimize/calling_conventions.pdf

segment word size	16 bit			32 bit						64 bit			
compiler	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu v.3.x	Intel Linux	Microsoft	Intel Windows	Gnu	Intel Linux
bool	2	1	1	1	1	1	1	1	1	1	1	1	1
char	1	1	1	1	1	1	1	1	1	1	1	1	1
wchar_t		2		2	2	2	2	2	2	2	2	4	4
short int	2	2	2	2	2	2	2	2	2	2	2	2	2
int	2	2	2	4	4	4	4	4	4	4	4	4	4
long int	4	4	4	4	4	4	4	4	4	4	4	8	8
__int64				8	8			8	8	8	8	8	8
enum	2	2	1	4	4	4	4	4	4	4	4	4	4
float	4	4	4	4	4	4	4	4	4	4	4	4	4
double	8	8	8	8	8	8	8	8	8	8	8	8	8
long double	10	10	8	8	16	10	8	12	12	8	16	16	16
__m64				8	8				8		8	8	8
__m128				16	16				16	16	16	16	16
__m256					32				32		32		32
pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
far pointer	4	4	4										
function pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
data member pointer (min)	2	4	6	4	4	8	4	4	4	4	4	8	8
data member pointer (max)		4	6	12	12	8	12	4	4	12	12	8	8
member function pointer (min)	2	12	6	4	4	12	4	8	8	8	8	16	16
member function pointer (max)		12	6	16	16	12	16	8	8	24	24	16	16

Table 1 shows how many bytes of storage various objects use for different compilers.

Literatur

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

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–8689–4238–5

[Ifr10] G. Ifrah: *Universalgeschichte der Zahlen*.
Tolkemitt bei Zweitausendeins, 2010.
ISBN 978–3–942048–31–6

[Kor16] Laszlo Korte: *TAMS Tools for eLearning*.
Uni Hamburg, FB Informatik, 2016, BSc Thesis. `tams.informatik.uni-hamburg.de/research/software/tams-tools`

Literatur (cont.)

[Gol91] D. Goldberg: *What every computer scientist should know about floating-point*. in: *ACM Computing Surveys* 23 (1991), March, Nr. 1, S. 5–48.
`www.validlab.com/goldberg/paper.pdf`

[Knu08] D.E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 0, Introduction to Combinatorial Algorithms and Boolean Functions*.
Addison-Wesley Professional, 2008. ISBN 978–0–321–53496–5

[Knu09] D.E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams*.
Addison-Wesley Professional, 2009. ISBN 978–0–321–58050–4

Literatur (cont.)

[Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript*. Universität Hamburg, FB Informatik, 2005.
`tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1`
Float/Double-Demonstration: `demoieee754`

[Omo94] A.R. Omondi: *Computer Arithmetic Systems – Algorithms, Architecture and Implementations*. Prentice-Hall International, 1994. ISBN 0–13–334301–4

[Kor01] I. Koren: *Computer Arithmetic Algorithms*. 2nd edition, CRC Press, 2001. ISBN 978–1–568–81160–4.
`www.ecs.umass.edu/ece/koren/arith`

[Spa76] O. Spaniol: *Arithmetik in Rechenanlagen*. B. G. Teubner, 1976. ISBN 3–519–02332–6

Gliederung

Gliederung (cont.)

Wiederholung: Stellenwertsystem („Radixdarstellung“)

- Wahl einer geeigneten Zahlenbasis b („Radix“)
  - 10: Dezimalsystem
  - 16: Hexadezimalsystem (Sedezimalsystem)
  - 2: Dualsystem
- Menge der entsprechenden Ziffern $\{0, 1, \ldots, b-1\}$
- inklusive einer besonderen Ziffer für den Wert Null
- Auswahl der benötigten Anzahl n von Stellen

$$|z| = \sum_{i=0}^{n-1} a_i \cdot b^i$$

b Basis a_i Koeffizient an Stelle i

- universell verwendbar, für beliebig große Zahlen

Integer-Datentypen in C und Java

C:

- Zahlenbereiche definiert in Headerdatei
  `/usr/include/limits.h`
  `LONG_MIN`, `LONG_MAX`, `ULONG_MAX` etc.
- Zweierkomplement (signed), Ganzzahl (unsigned)
- die Werte sind plattformabhängig (!)

Java:

- 16-bit, 32-bit, 64-bit Zweierkomplementzahlen
- Wrapper-Klassen `Short`, `Integer`, `Long`

```
Short.MAX_VALUE   =       32767
Integer.MIN_VALUE = -2147483648
Integer.MAX_VALUE =  2147483647
Long.MIN_VALUE    = -9223372036854775808L
...
```

- Werte sind für die Sprache fest definiert

Addition im Dualsystem

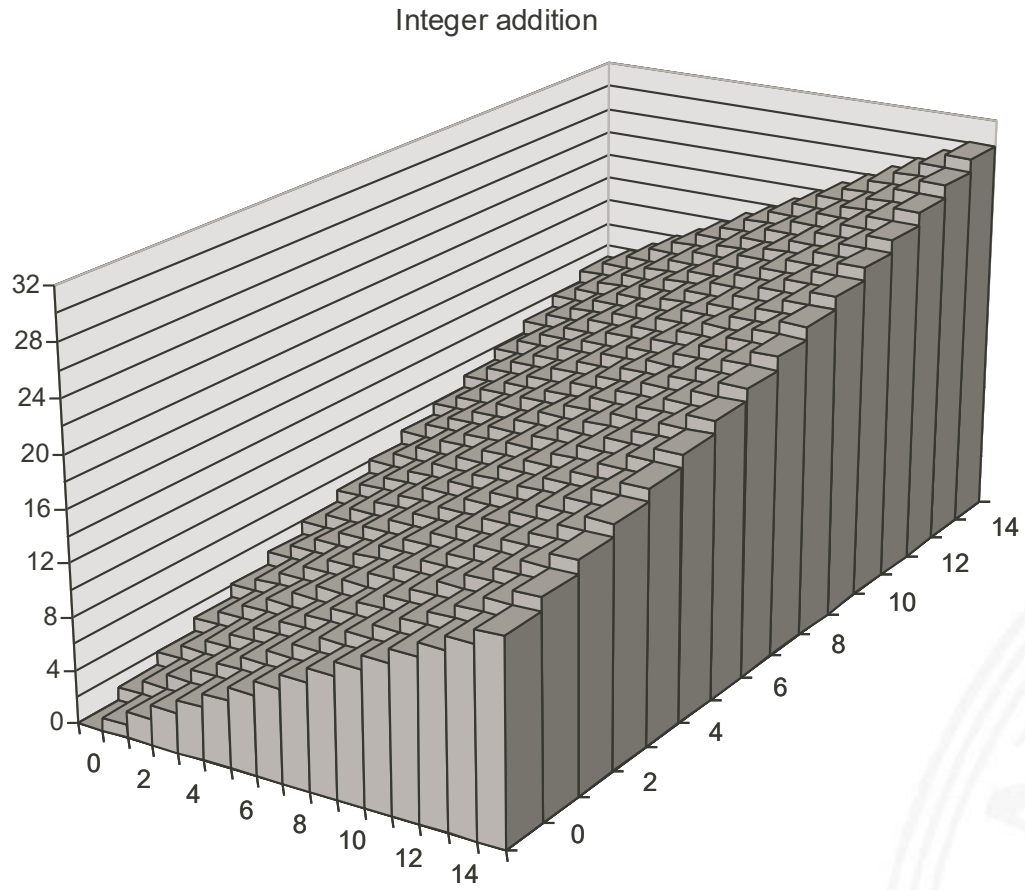
- funktioniert genau wie im Dezimalsystem
- Addition mehrstelliger Zahlen erfolgt stellenweise
- Additionsmatrix:

+	0	1
0	0	1
1	1	10

- Beispiel

	1011	0011	= 179
+	0011	1001	= 57
Ü	11	11	11
	1110	1100	= 236

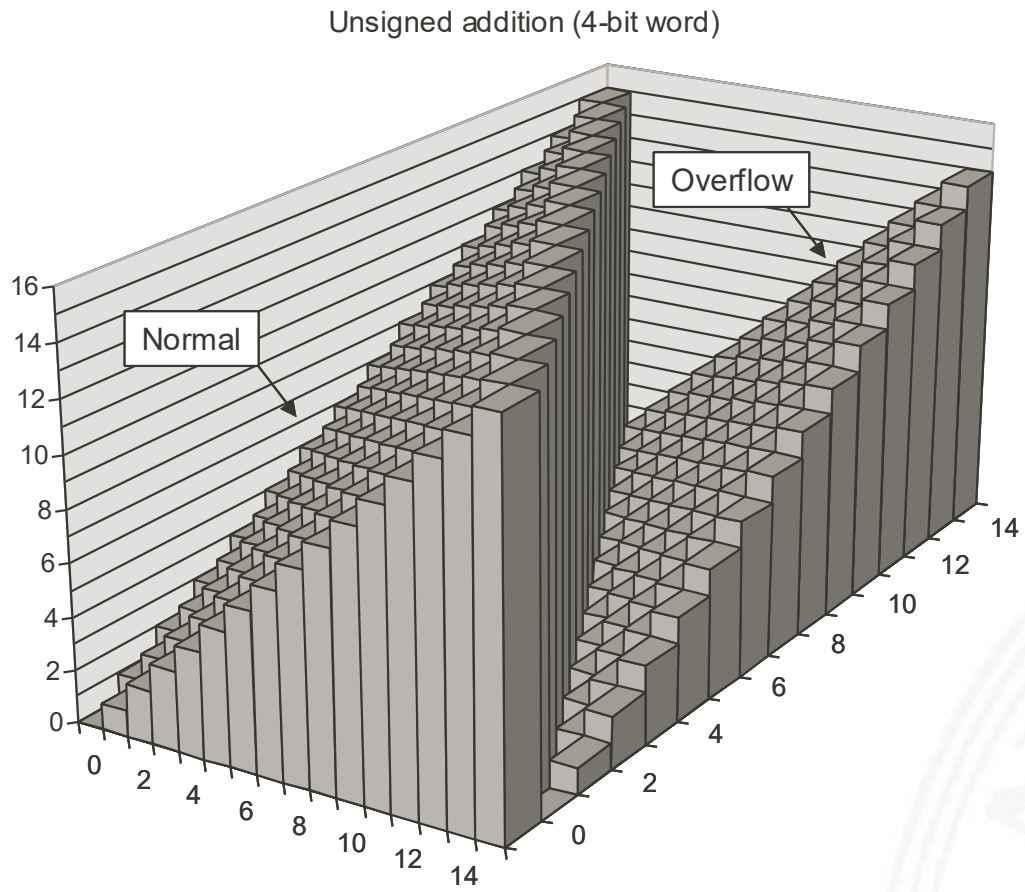
unsigned Addition: Visualisierung



[BO15]

- Wortbreite der Operanden ist w, hier 4-bit
- Zahlenbereich der Operanden x, y ist $0 \ldots (2^w - 1)$
- Zahlenbereich des Resultats s ist $0 \ldots (2^{w+1} - 2)$

unsigned Addition: Visualisierung (cont.)



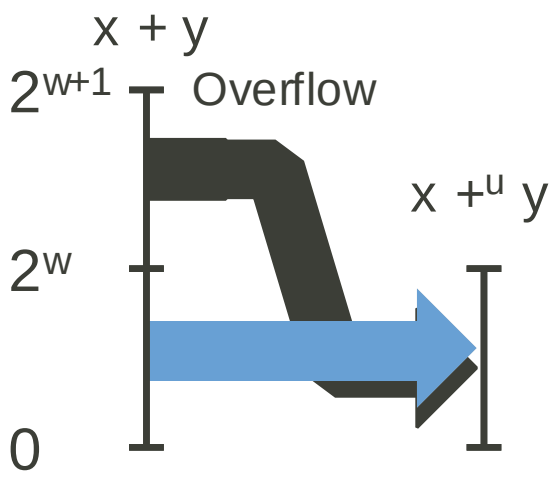
[BO15]

- Wortbreite der Operanden und des Resultats ist w

$\Rightarrow$ Überlauf, sobald das Resultat größer als $(2^w - 1)$

$\Rightarrow$ oberstes Bit geht verloren

unsigned Addition: Überlauf



- Wortbreite ist w
- Zahlenbereich der Operanden x, y ist $0 \ldots (2^w - 1)$
- Zahlenbereich des Resultats s ist $0 \ldots (2^{w+1} - 2)$
- Werte $s \geq 2^w$ werden in den Bereich $0 \ldots 2^w - 1$ abgebildet

Subtraktion im Dualsystem

- Subtraktion mehrstelliger Zahlen erfolgt stellenweise
- (Minuend - Subtrahend), Überträge berücksichtigen

- Beispiel

	1011 0011	= 179
-	0011 1001	= 57
Ü	1111	
	111 1010	= 122

- Alternative: Ersetzen der Subtraktion durch Addition des b-Komplements

Subtraktion mit b-Komplement

- bei Rechnung mit fester Stellenzahl n gilt:

$$K_b(z) + z = b^n \quad = 0$$

  weil b^n gerade nicht mehr in n Stellen hineinpasst
- also gilt für die Subtraktion auch:

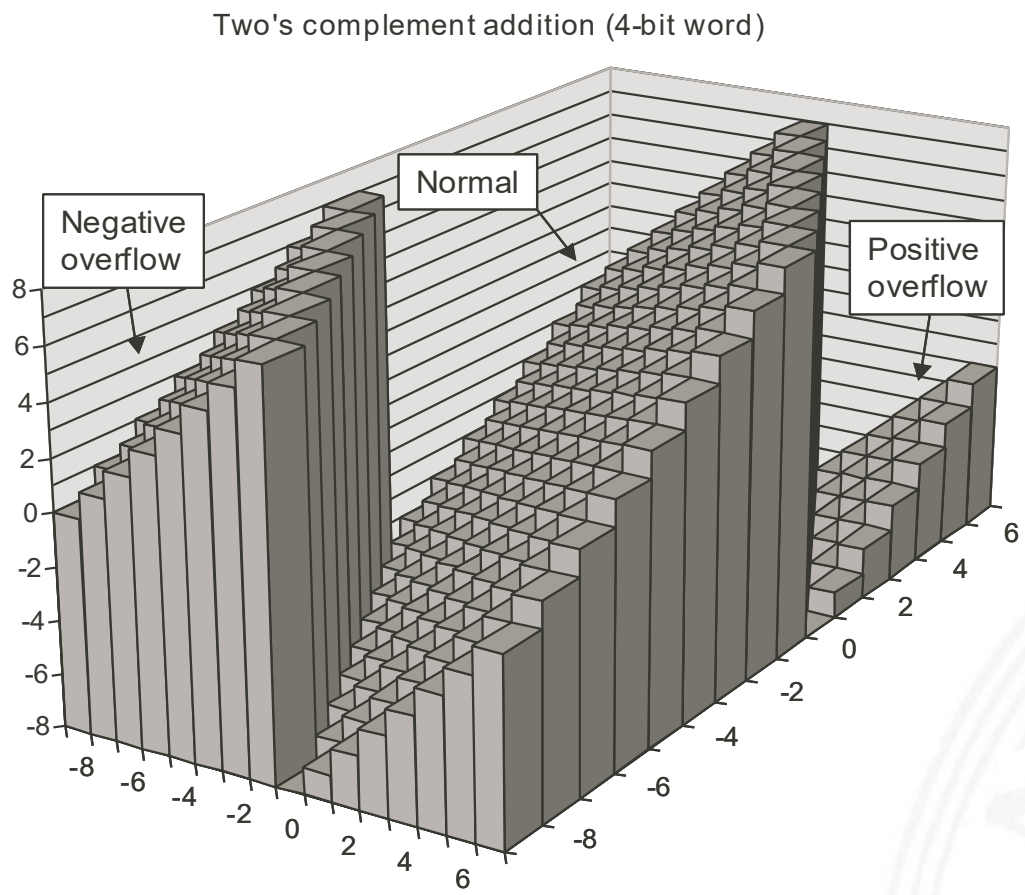
$$x - y = x + K_b(y)$$

$\Rightarrow$ Subtraktion kann also durch Addition des b-Komplements ersetzt werden

- und für Integerzahlen gilt außerdem

$$x - y = x + K_{b-1}(y) + 1$$

signed Addition: Visualisierung 2-Komplement

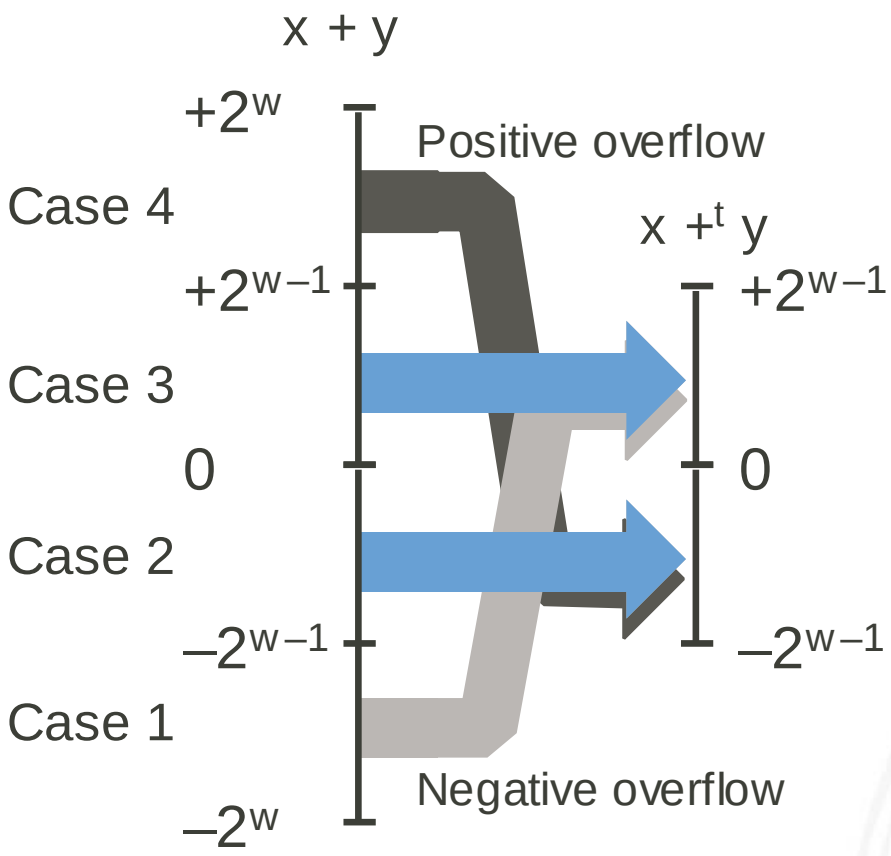


[BO15]

- Wortbreite der Operanden ist w, hier 4-bit
- Zahlenbereich der Operanden x, y ist $-2^{w-1} \ldots (2^{w-1} - 1)$
- Zahlenbereich des Resultats s ist $-2^{w} \ldots (2^{w} - 2)$

$\Rightarrow$ Überlauf in beide Richtungen möglich

signed Addition: Überlauf



- Wortbreite des Resultats ist w: Bereich $-2^{w-1} \ldots (2^{w-1}-1)$
- Überlauf positiv wenn Resultat $\geq 2^{w-1}$: Summe negativ
  –"– negativ –"– $< -2^{w-1}$: Summe positiv

Überlauf: Erkennung

- Erkennung eines Überlaufs bei der Addition?
- wenn beide Operanden das gleiche Vorzeichen haben und sich das Vorzeichen des Resultats unterscheidet

- Java-Codebeispiel

```
int a, b, sum;          // operands and sum
boolean ovf;            // ovf flag indicates overflow

sum = a + b;
ovf = ((a < 0) == (b < 0)) && ((a < 0) != (sum < 0));
```

Subtraktion mit Einer- und Zweierkomplement

- Subtraktion ersetzt durch Addition des Komplements

Dezimal	1-Komplement	2-Komplement
10	0000 1010	0000 1010
+(-3)	1111 1100	1111 1101
+7	1 0000 0110	1 0000 0111
Übertrag:	addieren +1	verwerfen
	0000 0111	0000 0111

- das b-**Komplement** einer Zahl z ist

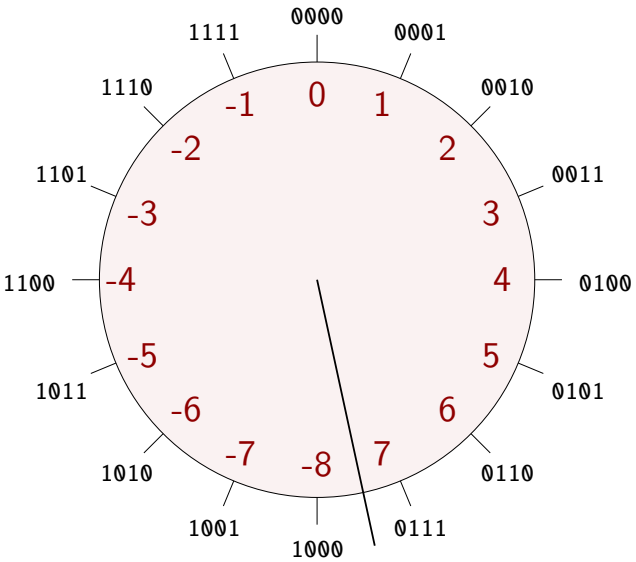
$$K_b(z) = b^n - z, \quad \text{für} \quad z \neq 0$$
$$= 0, \quad \text{für} \quad z = 0$$

- das $(b-1)$-**Komplement** einer Zahl z ist

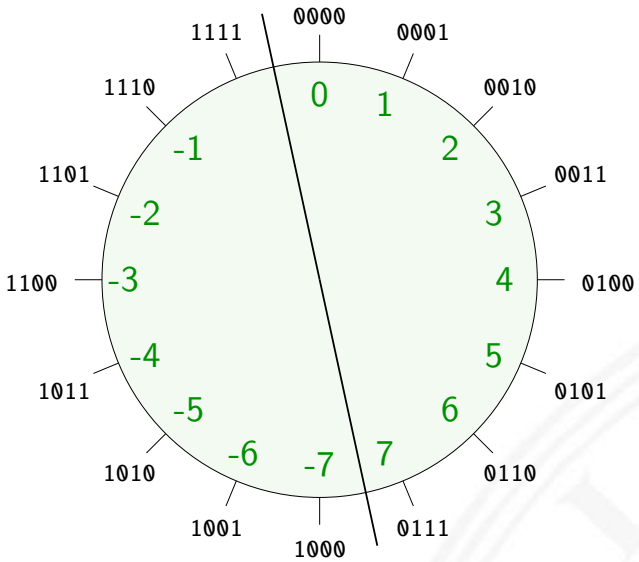
$$K_{b-1}(z) = b^n - z - b^{-m}, \quad \text{für} \quad z \neq 0$$
$$= 0, \quad \text{für} \quad z = 0$$

Veranschaulichung: Zahlenkreis

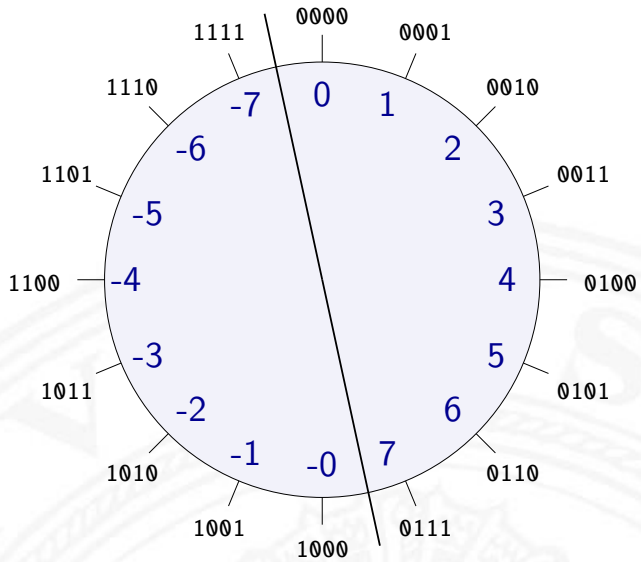
Beispiel für 4-bit Zahlen



2-Komplement



1-Komplement

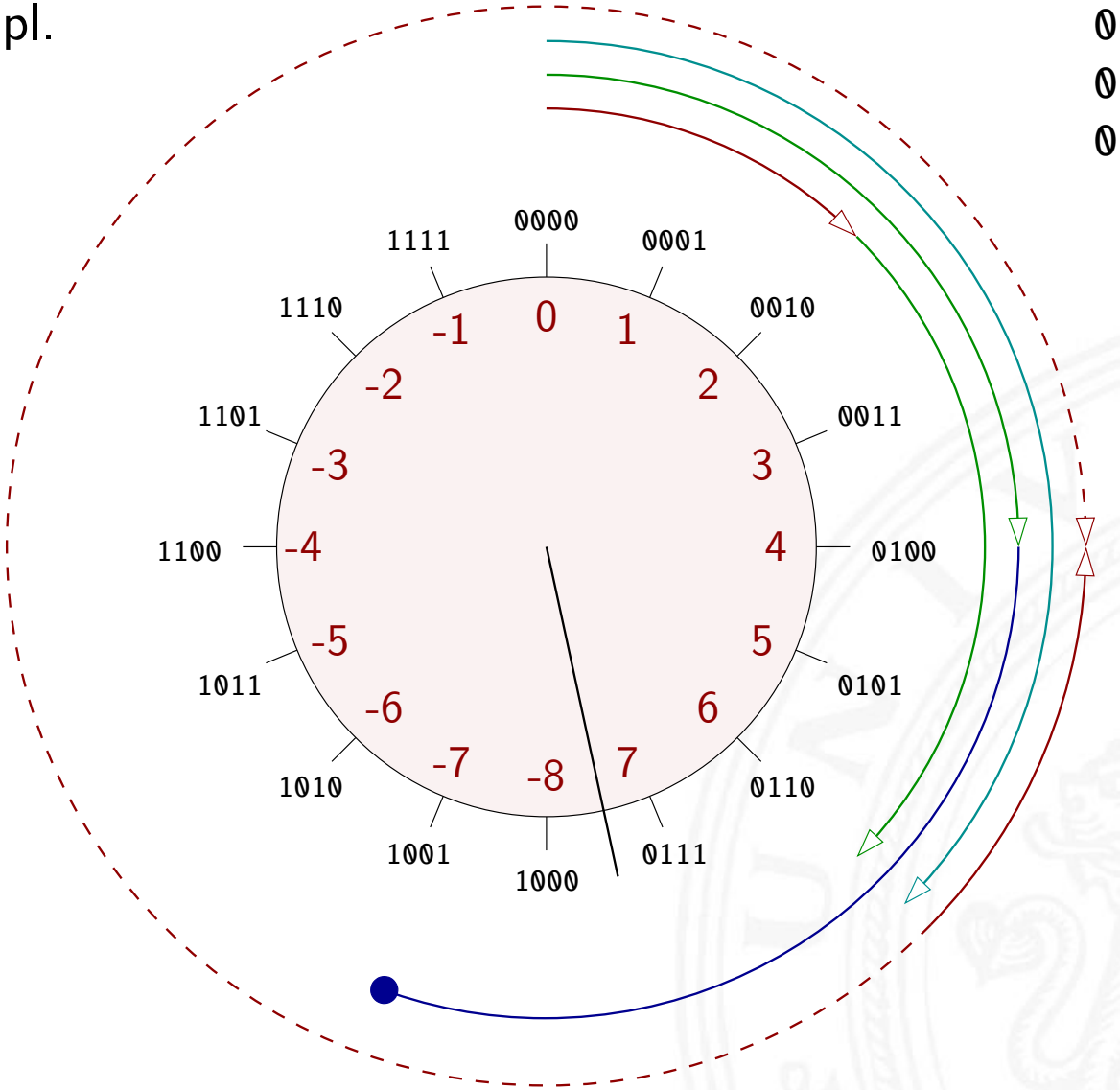


Betrag+Vorzeichen

- Komplement-Arithmetik als Winkeladdition

Zahlenkreis: Addition, Subtraktion

2-Kompl.

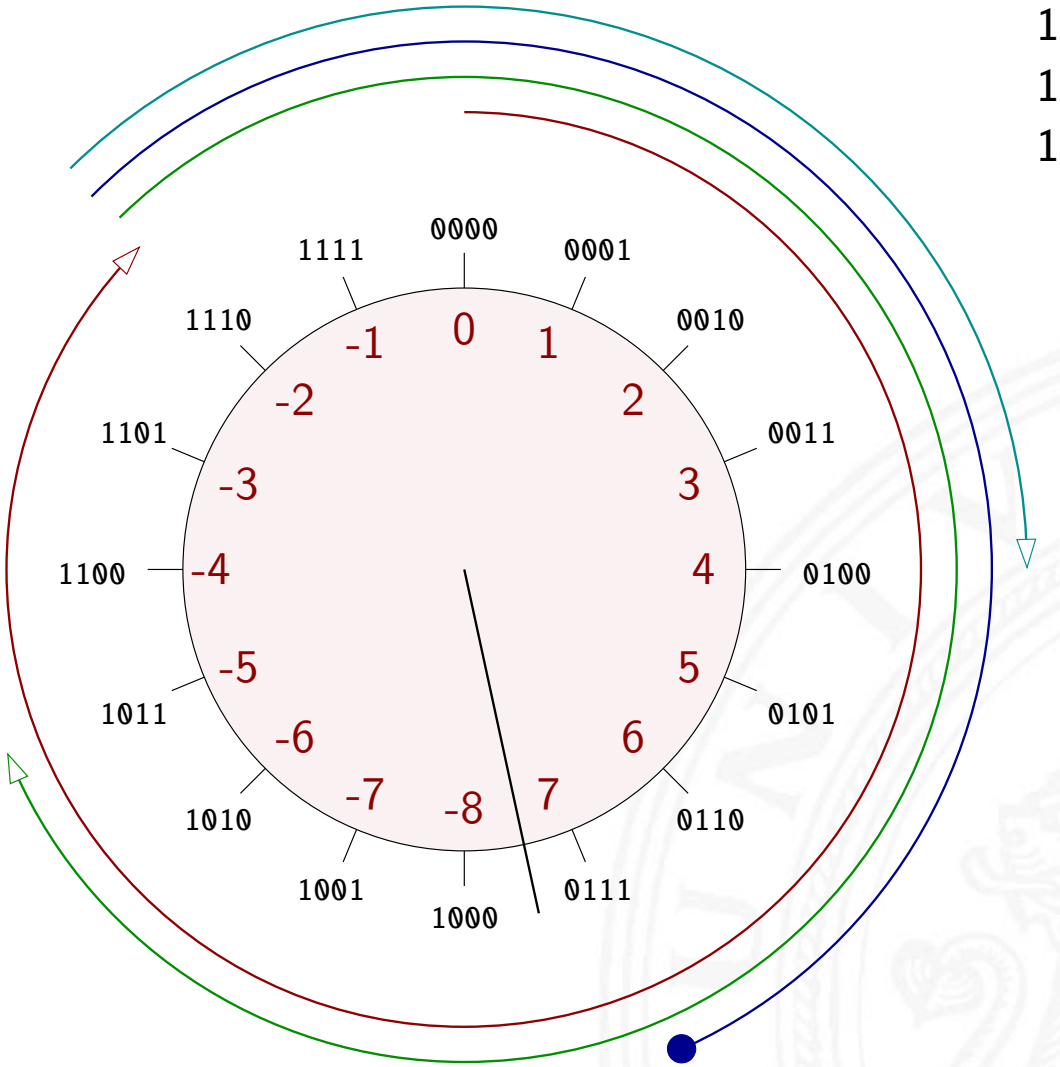


0010+0100=0110
0100+0101=1001
0110-0010=0100

0010 1110
0100
0101
0110

Zahlenkreis: Addition, Subtraktion (cont.)

2-Kompl.

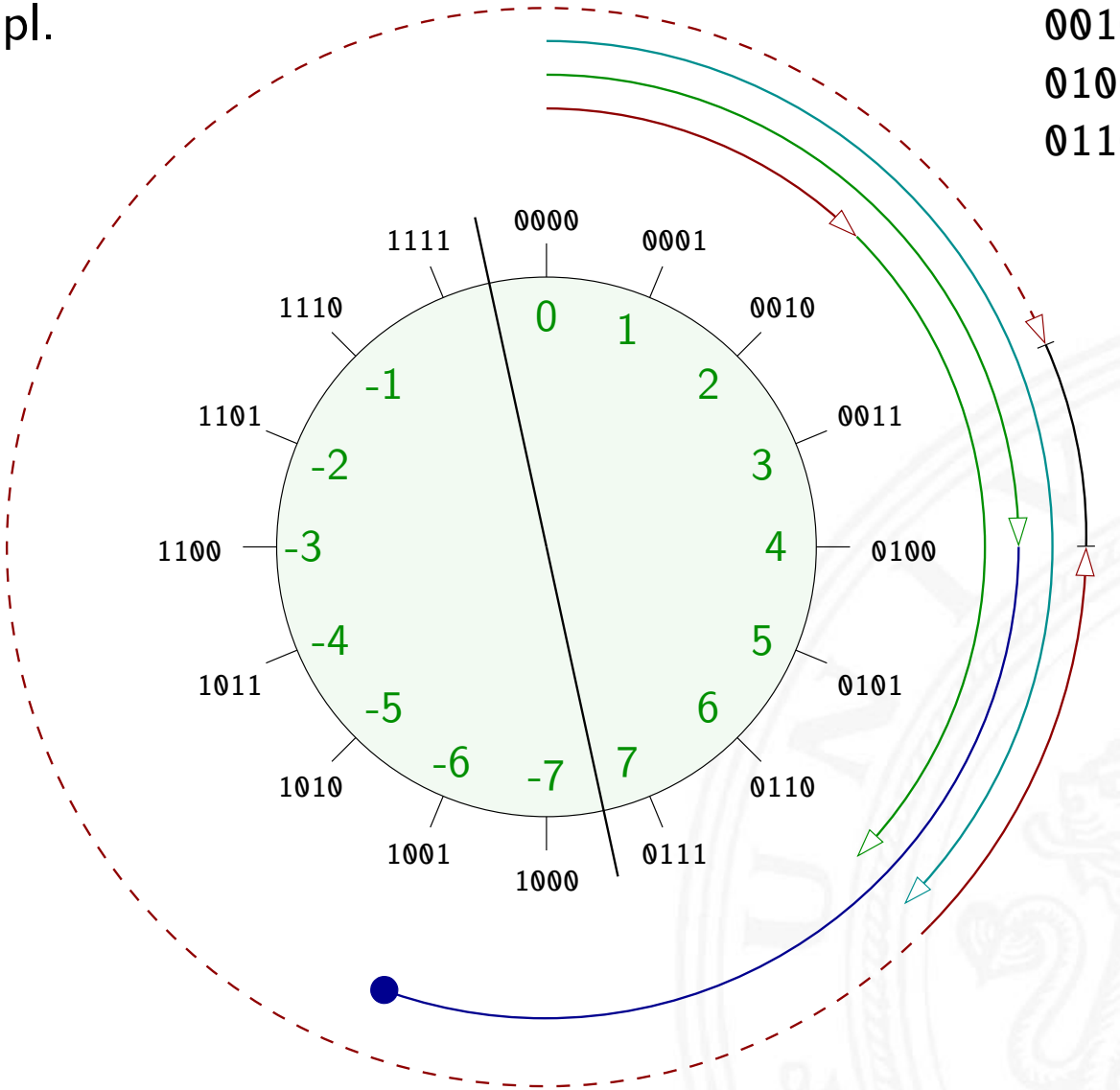


1110+1101=1011
1110+1001=0111
1110+0110=0100

1110
1101
1001
0110

Zahlenkreis: Addition, Subtraktion (cont.)

1-Kompl.

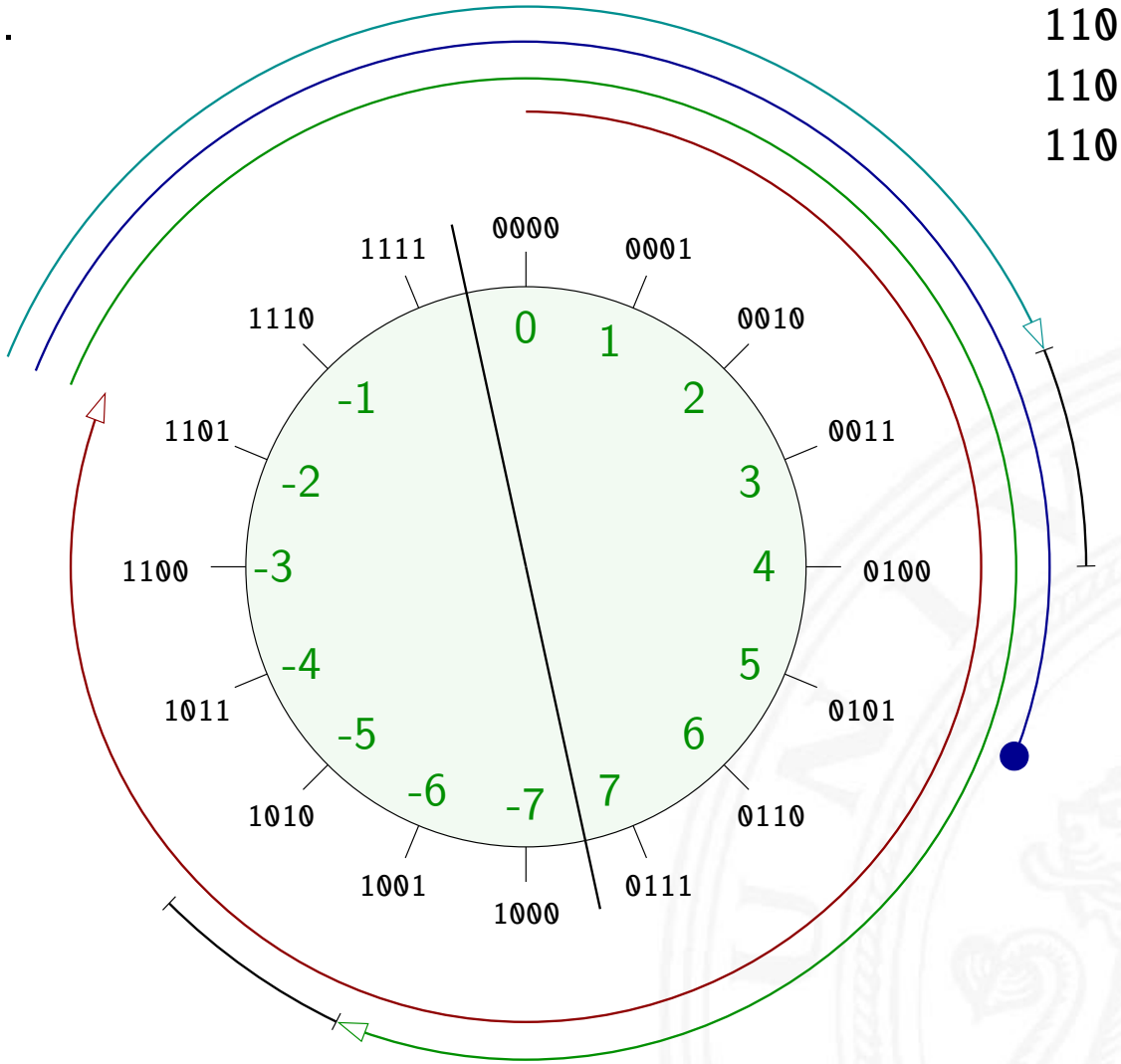


0010+0100=0110
0100+0101=1001
0110+1101+1=0100

0010 1101
0100
0101
0110

Zahlenkreis: Addition, Subtraktion (cont.)

1-Kompl.



1101+1100+1=1010
1101+1000=0101
1101+0110+1=0100

1101
1100
1000
0110

- für hardwarenahe Programme und Treiber
- für modulare Arithmetik („multi-precision arithmetic“)
- aber evtl. ineffizient (vom Compiler schlecht unterstützt)

- Vorsicht vor solchen Fehlern

```
unsigned int i, cnt = ...;
for( i = cnt-2; i >= 0; i-- ) {
  a[i] += a[i+1];
}
```

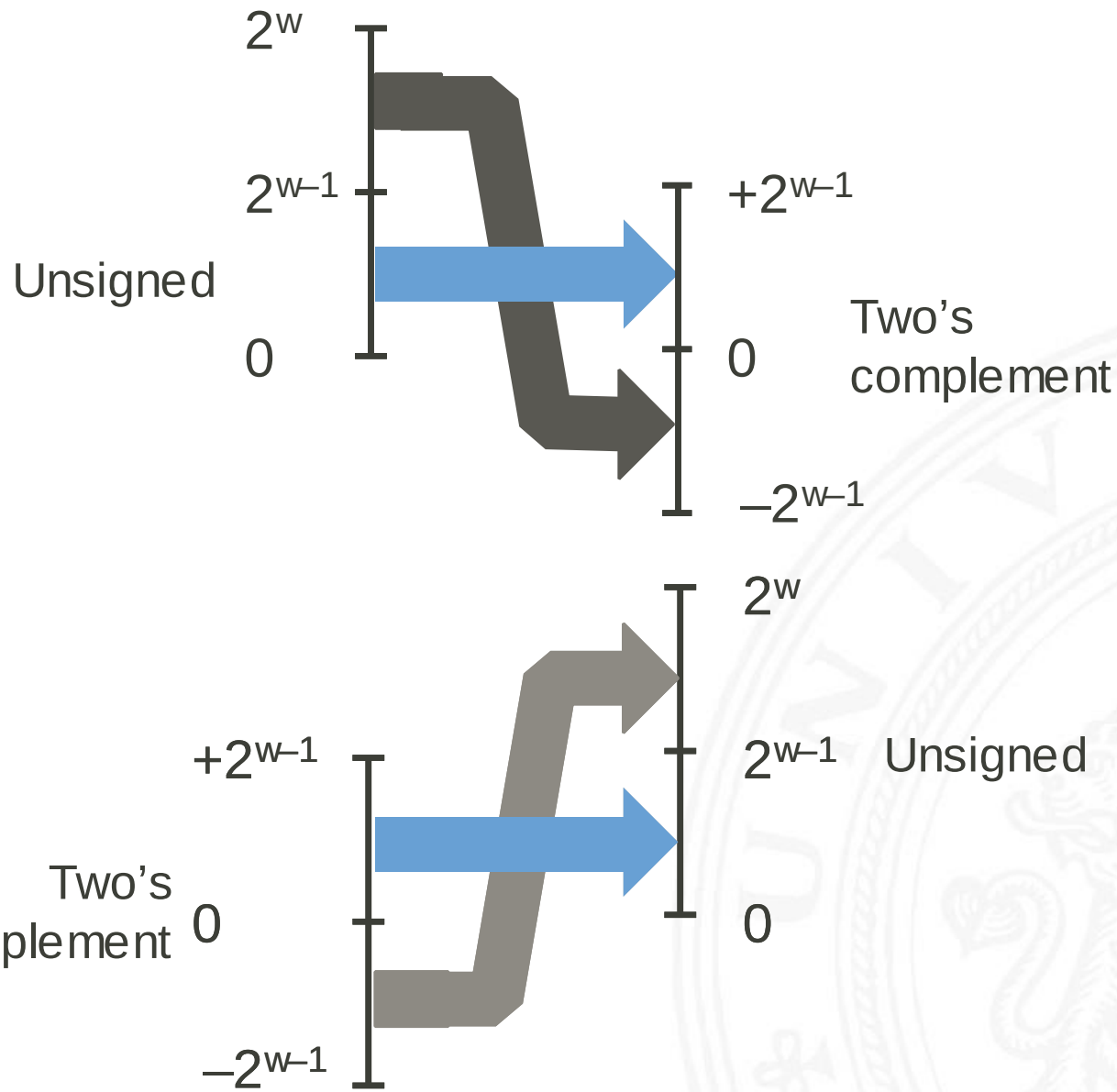
in C: Casting-Regeln

- Bit-Repräsentation wird nicht verändert
- kein Effekt auf positiven Zahlen
- Negative Werte als (große) positive Werte interpretiert

```
short int          x =  15213;
unsigned short int ux = (unsigned short) x; // 15213

short int          y = -15213;
unsigned short int uy = (unsigned short) y; // 50323
```

- Schreibweise für Konstanten:
  - ohne weitere Angabe: signed
  - Suffix „U“ für unsigned: `0U`, `4294967259U`

in C: unsigned / signed Interpretation



in C: Vorsicht bei Typumwandlung

- Arithmetische Ausdrücke:
  - bei gemischten Operanden: Auswertung als unsigned
  - auch für die Vergleichsoperationen <, >, ==, <=, >=
  - Beispiele für Wortbreite 32-bit:

Konstante 1	Relation	Konstante 2	Auswertung	Resultat
0	==	0U	unsigned	1
-1	<	0	signed	1
-1	<	0U	unsigned	0
2147483647	>	-2147483648	signed	1
2147483647U	>	-2147483648	unsigned	0
2147483647	>	(int) 2147483648U	signed	1
-1	>	-2	signed	1
(unsigned) -1	>	-2	unsigned	1

Fehler

Sign-Extension

- Gegeben: w-bit Integer x
- Umwandeln in $w + k$-bit Integer x' mit gleichem Wert?

- **Sign-Extension**: Vorzeichenbit kopieren

$$x' = x_{w-1}, \ldots x_{w-1}, x_{w-1}, x_{w-2}, \ldots x_0$$

- Zahlenbeispiele

0110	4-bit signed: +6
0000 0110	8-bit signed: +6
0000 0000 0000 0110	16-bit signed: +6
1110	4-bit signed: -2
1111 1110	8-bit signed: -2
1111 1111 1111 1110	16-bit signed: -2

```
public static void main( String[] args ) {
  System.out.println(
    Long.toHexString( 0x100000000L + 0xcafebabe ));
}
```

- Programm addiert zwei Konstanten, Ausgabe in Hex-Format
- Was ist das Resultat der Rechnung?

```
  0xffffffffcafebabe   (sign-extension!)
  0x0000000100000000
--------------------
  Ü 11111110
====================
    00000000cafebabe
```

Ariane-5 Absturz

Ariane-5 Absturz (cont.)

- Erstflug der Ariane-5 („V88“) am 04. Juni 1996
- Kurskorrektur wegen vermeintlich falscher Fluglage
- Selbstzerstörung der Rakete nach 36,7 Sekunden
- Schaden ca. 635 M€ (teuerster Softwarefehler der Geschichte?)

- bewährte Software von Ariane-4 übernommen
- aber Ariane-5 viel schneller als Ariane-4
- 64-bit Gleitkommawert für horizontale Geschwindigkeit
- Umwandlung in 16-bit Integer: dabei Überlauf

- `https://de.wikipedia.org/wiki/Ariane_V88`

Multiplikation im Dualsystem

- funktioniert genau wie im Dezimalsystem
- $p = a \cdot b$ mit Multiplikator a und Multiplikand b
- Multiplikation von a mit je einer Stelle des Multiplikanten b
- Addition der Teilterme

- Multiplikationsmatrix ist sehr einfach:

$\cdot$	0	1
0	0	0
1	0	1

Multiplikation im Dualsystem (cont.)

- Beispiel

	10110011	· 1101	$= 179 \cdot 13$	$= 2327$
	10110011	1		$= 1001\,0001\,0111$
	10110011	1		$= 0x917$
	00000000	0		
	10110011	1		
Ü	11101111			
	100100010111			

unsigned Multiplikation

- bei Wortbreite w bit
- Zahlenbereich der Operanden: $0 \ldots (2^w - 1)$
- Zahlenbereich des Resultats: $0 \ldots (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$

$\Rightarrow$ bis zu $2w$ bits erforderlich

- C: Resultat enthält nur die unteren w bits
- Java: keine unsigned Integer
- Hardware: teilweise zwei Register *high*, *low* für die oberen und unteren Bits des Resultats

signed Multiplikation 2-Komplement

- Zahlenbereich der Operanden: $-2^{w-1} \ldots (2^{w-1} - 1)$
- Zahlenbereich des Resultats: $-2^{w-1} \cdot (2^{w-1} - 1) \ldots (2^{2w-2})$

$\Rightarrow$ bis zu $2w$ bits erforderlich

- C, Java: Resultat enthält nur die unteren w bits
- Überlauf wird ignoriert

```
int i = 100*200*300*400; // -1894967296
```

- Repräsentation der unteren Bits des Resultats entspricht der unsigned Multiplikation

$\Rightarrow$ kein separater Algorithmus erforderlich
Beweis: siehe Bryant, O'Hallaron: Abschnitt 2.3.5 [BO15]

Java Puzzlers No. 3

J. Bloch, N. Gafter: *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, 2005


```
public static void main( String args[] ) {
  final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
  final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
  System.out.println(  MICROS_PER_DAY / MILLIS_PER_DAY );
}
```

- druckt den Wert 5, nicht 1000!
- `MICROS_PER_DAY` mit 32-bit berechnet, dabei Überlauf
- Konvertierung nach 64-bit `long` erst bei Zuweisung
- `long`-Konstante schreiben: `24L * 60 * 60 * 1000 * 1000`

Division im Dualsystem

- $d = a/b$ mit Dividend a und Divisor b

- funktioniert genau wie im Dezimalsystem
- schrittweise Subtraktion des Divisors
- Berücksichtigen des „Stellenversetzens“

- in vielen Prozessoren nicht (oder nur teilweise) durch Hardware unterstützt
- daher deutlich langsamer als Multiplikation

Division im Dualsystem (cont.)

- Beispiele

$100_{10}/3_{10} = 110\,0100_2/11_2 = 10\,0001_2$

1100100	/	11	=	0100001					
1				0					
11				1					
−11									
0				0					
0				0					
1				0					
10				0					
100				1					
−11									
1				1	(Rest)				

Division im Dualsystem (cont.)

$$91_{10}/13_{10} = 101\,1011_2/1101_2 = 111_2$$

$$\begin{array}{rcl|l}
1011011 & / & 1101 = & 0111 \\
1011 & & & 0 \\
10110 & & & 1 \\
-1101 & & & \\
\hline
10011 & & & 1 \\
-1101 & & & \\
\hline
01101 & & & 1 \\
-1101 & & & \\
\hline
0 & & &
\end{array}$$

Höhere mathematische Funktionen

Berechnung von $\sqrt{x}$, $\log x$, $\exp x$, $\sin x$ ...?

- Approximation über Polynom (Taylor-Reihe) bzw.
  Approximation über rationale Funktionen
  - vorberechnete Koeffizienten für höchste Genauigkeit
  - Ausnutzen mathematischer Identitäten für Skalierung
- Sukzessive Approximation über iterative Berechnungen
  - Beispiele: Quadratwurzel und Reziprok-Berechnung
  - häufig schnelle (quadratische) Konvergenz
- Berechnungen erfordern nur die Grundrechenarten

Reziprokwert: Iterative Berechnung von $1/x$

- Berechnung des Reziprokwerts $y = 1/x$ über

$$y_{i+1} = y_i \cdot (2 - x \cdot y_i)$$

- geeigneter Startwert y_0 als Schätzung erforderlich

- Beispiel $x = 3$, $y_0 = 0,5$:

$$\begin{aligned}
y_1 &= 0,5 \cdot (2 - 3 \cdot 0,5) && = 0,25 \\
y_2 &= 0,25 \cdot (2 - 3 \cdot 0,25) && = 0,3125 \\
y_3 &= 0,3125 \cdot (2 - 3 \cdot 0,3125) && = 0,33203125 \\
y_4 &= 0,3332824 \\
y_5 &= 0,333333332557231 \\
y_6 &= 0,3333333333333333
\end{aligned}$$

Quadratwurzel: Heron-Verfahren für $\sqrt{x}$

Babylonisches Wurzelziehen

- Sukzessive Approximation von $y = \sqrt{x}$ gemäß

$$y_{n+1} = \frac{y_n + x/y_n}{2}$$

- quadratische Konvergenz in der Nähe der Lösung
- Anzahl der gültigen Stellen verdoppelt sich mit jedem Schritt

- aber langsame Konvergenz fernab der Lösung
- Lookup-Tabelle und Tricks für brauchbare Startwerte y_0

Informationstreue

Welche mathematischen Eigenschaften gelten bei der Informationsverarbeitung, in der gewählten Repräsentation?

Beispiele:

- Gilt $x^2 \geq 0$?
  - float: ja
  - signed integer: nein
- Gilt $(x + y) + z = x + (y + z)$?
  - integer: ja
  - float: nein

  1.0E20 + (-1.0E20 + 3.14) = 0

Festkomma Addition

unsigned Arithmetik

- Wortbreite auf w begrenzt
- kommutative Gruppe / Abel'sche Gruppe
  - Abgeschlossenheit $0 \leq a \oplus_w^u b \leq 2^w - 1$
  - Kommutativgesetz $a \oplus_w^u b = b \oplus_w^u a$
  - Assoziativgesetz $a \oplus_w^u (b \oplus_w^u c) = (a \oplus_w^u b) \oplus_w^u c$
  - neutrales Element $a \oplus_w^u 0 = a$
  - Inverses $a \oplus_w^u \overline{a} = 0; \overline{a} = 2^w - a$

Festkomma Addition (cont.)

signed Arithmetik 2-Komplement

- Wortbreite auf w begrenzt
- signed und unsigned Addition sind auf Bit-Ebene identisch

  $a \oplus_w^s b = U2S(S2U(a) \oplus_w^u S2U(b))$

$\Rightarrow$ isomorphe Algebra zu $\oplus_w^u$

- kommutative Gruppe / Abel'sche Gruppe
  - Abgeschlossenheit $-2^{w-1} \leq a \oplus_w^s b \leq 2^{w-1} - 1$
  - Kommutativgesetz $a \oplus_w^s b = b \oplus_w^s a$
  - Assoziativgesetz $a \oplus_w^s (b \oplus_w^s c) = (a \oplus_w^s b) \oplus_w^s c$
  - neutrales Element $a \oplus_w^s 0 = a$
  - Inverses $a \oplus_w^s \overline{a} = 0; \quad \overline{a} = -a,\ a \neq -2^{w-1}$
    $a,\ a = -2^{w-1}$

Festkomma Multiplikation

unsigned Arithmetik

- Wortbreite auf w begrenzt
- Modulo-Arithmetik $a \otimes_w^u b = (a \cdot b) \, mod \, 2^w$
- $\otimes_w^u$ und $\oplus_w^u$ bilden einen kommutativen Ring
  - $\oplus_w^u$ ist eine kommutative Gruppe
  - Abgeschlossenheit $0 \leq a \otimes_w^u b \leq 2^w - 1$
  - Kommutativgesetz $a \otimes_w^u b = b \otimes_w^u a$
  - Assoziativgesetz $a \otimes_w^u (b \otimes_w^u c) = (a \otimes_w^u b) \otimes_w^u c$
  - neutrales Element $a \otimes_w^u 1 = a$
  - Distributivgesetz $a \otimes_w^u (b \oplus_w^u c) = (a \otimes_w^u b) \oplus_w^u (a \otimes_w^u c)$

Festkomma Multiplikation (cont.)

signed Arithmetik

- signed und unsigned Multiplikation sind auf Bit-Ebene identisch
- ...

isomorphe Algebren

- unsigned Addition und Multiplikation; Wortbreite w
- signed Addition und Multiplikation; Wortbreite w 2-Kompl.
- isomorph zum Ring der ganzen Zahlen *modulo* 2^w

- Ordnungsrelation im Ring der ganzen Zahlen
  - a > 0 $\longrightarrow a + b > b$
  - $a > 0, b > 0 \longrightarrow a \cdot b > 0$
  - diese Relationen gelten nicht bei Rechnerarithmetik! Überlauf

Gleitkomma Addition

Vergleich mit kommutativer Gruppe

- Abgeschlossen? Ja
- Kommutativ? Ja
- Assoziativ? Nein
  (Überlauf, Rundungsfehler)
- Null ist neutrales Element? Ja
- Inverses Element existiert? Fast
  (außer für NaN und Infinity)

- Monotonie? $a \geq b \longrightarrow (a + c) \geq (b + c)$ Fast
  (außer für NaN und Infinity)

Gleitkomma Multiplikation

Vergleich mit kommutativem Ring

- Abgeschlossen? Ja
  (aber Infinity oder NaN möglich)
- Kommutativ? Ja
- Assozativ? Nein
  (Überlauf, Rundungsfehler)
- Eins ist neutrales Element? Ja
- Distributivgesetz? Nein

- Monotonie? $a \geq b; c \geq 0 \longrightarrow (a \cdot c) \geq (b \cdot c)$ Fast
  (außer für NaN und Infinity)

Literatur

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

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–8689–4238–5

Literatur (cont.)

[Omo94] A.R. Omondi: *Computer Arithmetic Systems – Algorithms, Architecture and Implementations*. Prentice-Hall International, 1994. ISBN 0–13–334301–4

[Kor01] I. Koren: *Computer Arithmetic Algorithms*. 2nd edition, CRC Press, 2001. ISBN 978–1–568–81160–4. `www.ecs.umass.edu/ece/koren/arith`

[Spa76] O. Spaniol: *Arithmetik in Rechenanlagen*. B. G. Teubner, 1976. ISBN 3–519–02332–6

Gliederung

Gliederung (cont.)

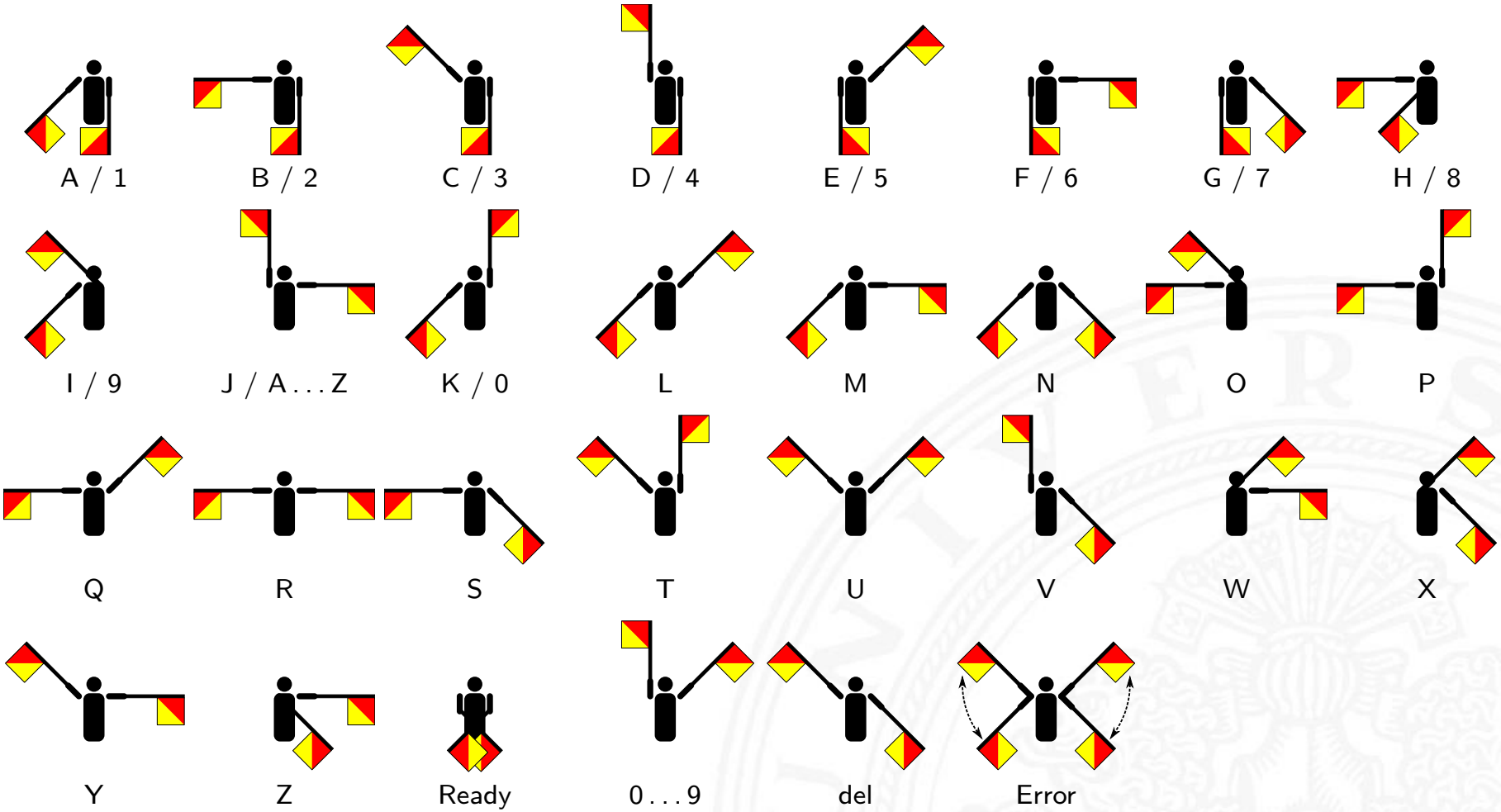
Wiederholung: Zeichen

- **Zeichen**: engl. *character*
  Element z aus einer zur Darstellung von Information vereinbarten, einer Abmachung unterliegenden, endlichen Menge Z von Elementen
- Die Menge Z heißt **Zeichensatz** oder **Zeichenvorrat**
  engl. *character set*
- **Binärzeichen**: engl. *binary element, binary digit, bit*
  Jedes der Zeichen aus einem Vorrat / aus einer Menge von zwei Symbolen
- **Numerischer Zeichensatz**: Zeichenvorrat aus Ziffern und/oder Sonderzeichen zur Darstellung von Zahlen
- **Alphanumerischer Zeichensatz**: Zeichensatz aus (mindestens) den Dezimalziffern und den Buchstaben des gewöhnlichen Alphabets, meistens auch mit Sonderzeichen (Leerzeichen, Punkt, Komma usw.)

Wiederholung: Zeichen (cont.)

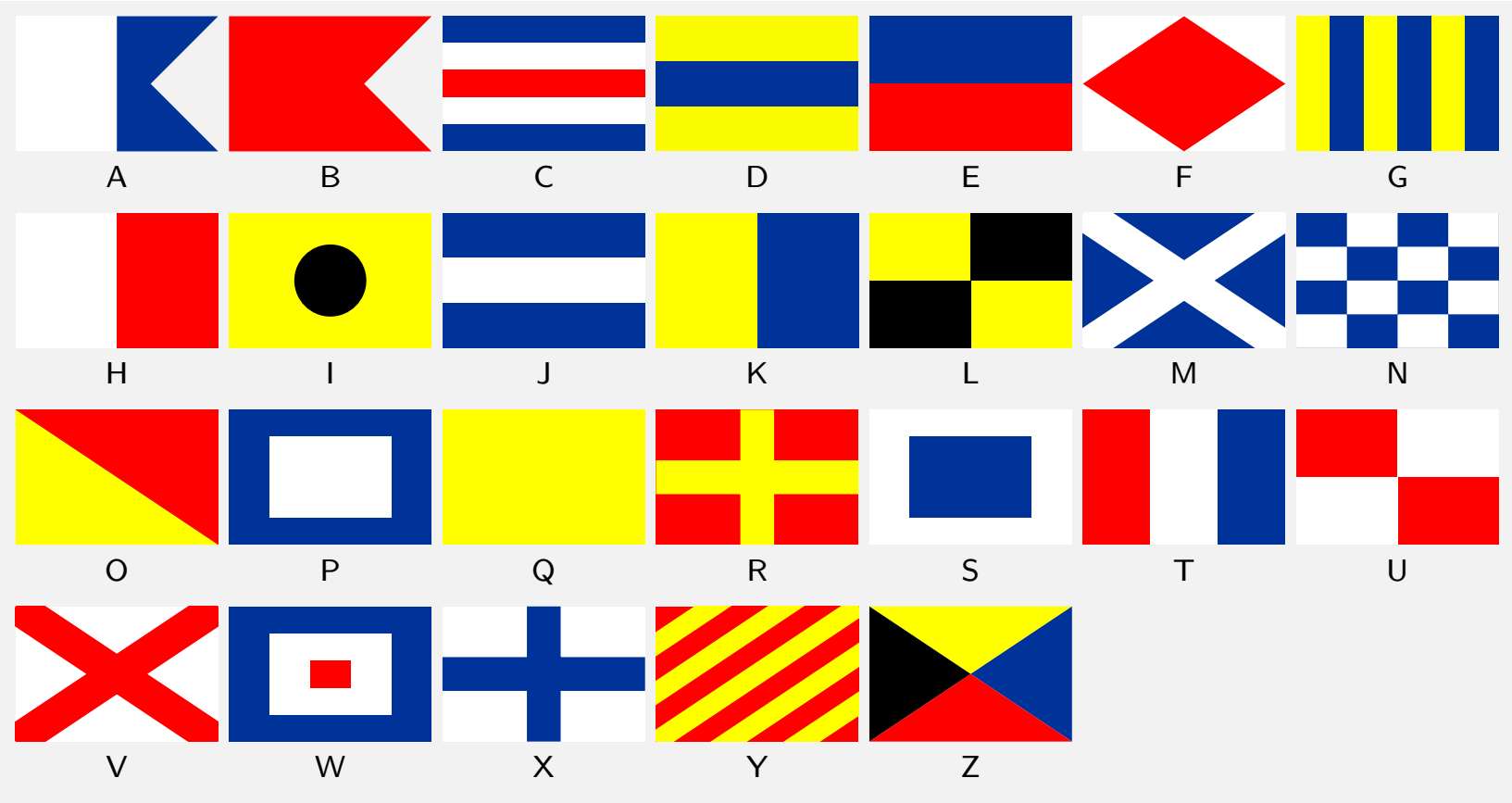
- **Alphabet**: engl. *alphabet*
  Ein in vereinbarter Reihenfolge geordneter Zeichenvorrat
- **Zeichenkette**: engl. *string*
  Eine Folge von Zeichen
- **Wort**: engl. *word*
  Eine Folge von Zeichen, die in einem gegebenen Zusammenhang als Einheit bezeichnet wird
- Worte mit 8 bit werden als **Byte** bezeichnet
- **Stelle**: engl. *position*
  Die Lage/Position eines Zeichens innerhalb einer Zeichenkette

Flaggen-Signale



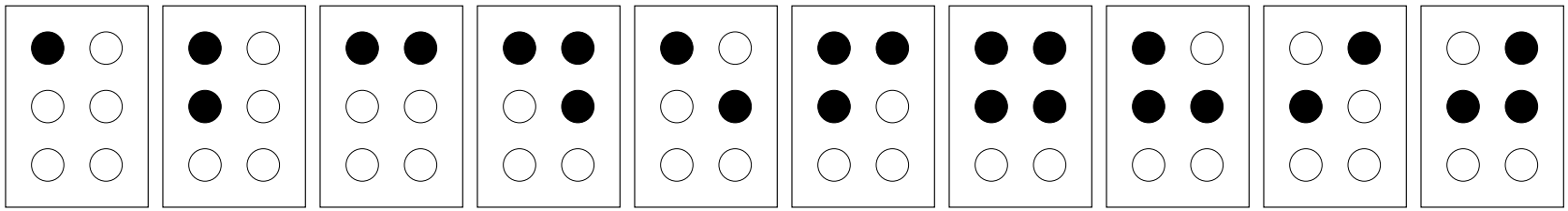
wikipedia.org/wiki/Winkeralphabet

Flaggen-Alphabet

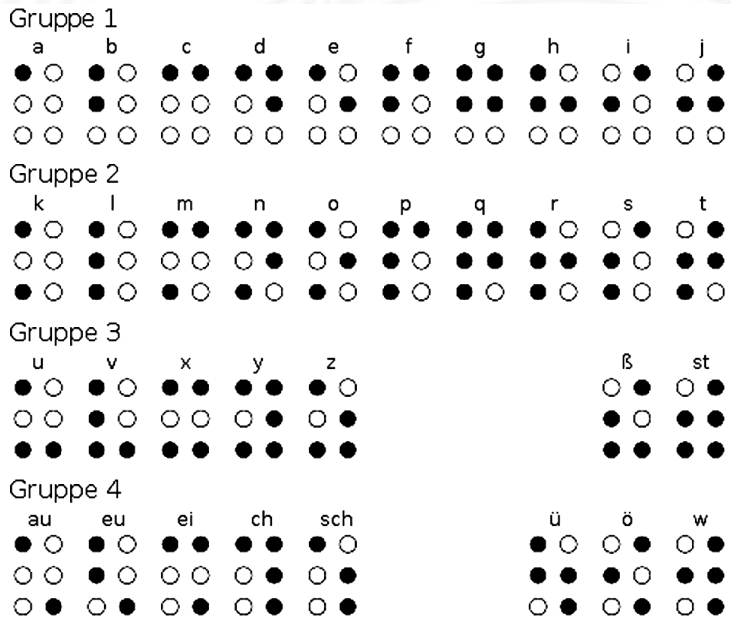


de.wikipedia.org/wiki/Flaggenalphabet

Braille: Blindenschrift



- Symbole als 2x3 Matrix (geprägte Punkte)
- Erweiterung auf 2x4 Matrix (für Computer)
- bis zu 64 (256) mögliche Symbole
- diverse Varianten
  - ein Symbol pro Buchstabe
  - ein Symbol pro Silbe
  - Kurzschrift/Steno



Morse-Code

Codetabelle

● kurzer Ton — langer Ton

A	●—	S	●●●	.	●—●—●—	S-Start	—●—●—
B	—●●●	T	—	,	——●●——	Verst.	●●●—●
C	—●—●	U	●●—	?	●●——●●	S-Ende	●—●—●
D	—●●	V	●●●—	´	●—————●	V-Ende	●●●—●—
E	●	W	●——	!	—●—●——	Error	●●●●●●●●
F	●●—●	X	—●●—	/	—●●—●		
G	——●	Y	—●——	(	—●——●	Ä	●—●—
H	●●●●	Z	——●●	)	—●——●—	À	●——●—
I	●●	0	—————	&	●—●●●	É	●●—●●
J	●———	1	●————	:	———●●●	È	●—●●—
K	—●—	2	●●———	;	—●—●—●	Ö	———●
L	●—●●	3	●●●——	=	—●●●—	Ü	●●——
M	——	4	●●●●—	+	●—●—●	ß	●●●——●●
N	—●	5	●●●●●	-	—●●●●—	CH	————
O	———	6	—●●●●	_	●●——●—	Ñ	——●——
P	●——●	7	——●●●	"	●—●●—●	...	
Q	——●—	8	———●●	$	●●●—●●—		
R	●—●	9	————●	@	●——●—●	SOS	●●● ——— ●●●

Morse-Code (cont.)

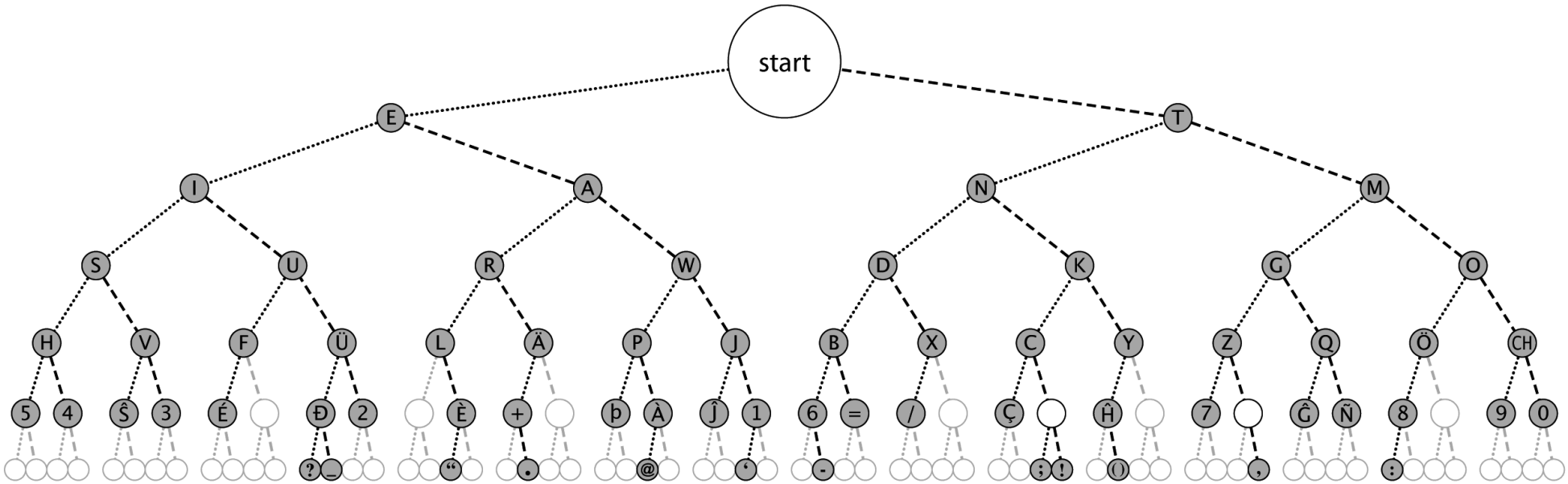
- Eindeutigkeit

Codewort:	● ● ● ● ● – ●
E	●
I	● ●
N	– ●
R	● – ●
S	● ● ●

  - bestimmte Morse-Sequenzen sind mehrdeutig
  - Pause zwischen den Symbolen notwendig

- Codierung
  - Häufigkeit der Buchstaben = 1 / Länge des Codewortes
  - Effizienz: kürzere Codeworte
  - Darstellung als Codebaum

Morse-Code: Baumdarstellung (Ausschnitt)



- Anordnung der Symbole entsprechend ihrer Codierung

ASCII

American Standard Code for Information Interchange

- eingeführt 1967, aktualisiert 1986: ANSI X3.4-1986
- viele Jahre der dominierende Code für Textdateien
- alle Zeichen einer typischen Schreibmaschine
- Erweiterung des früheren 5-bit Fernschreiber-Codes (Murray-Code)

- 7-bit pro Zeichen, 128 Zeichen insgesamt
- 95 druckbare Zeichen: Buchstaben, Ziffern, Sonderzeichen (Codierung im Bereich 21 . . . 7E)
- 33 Steuerzeichen (engl: *control characters*) (0 . . . 1F, 7F)

ASCII: Codetabelle

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	*NUL*	*SOH*	*STX*	*ETX*	*EOT*	*ENQ*	*ACK*	*BEL*	*BS*	*HT*	*LF*	*VT*	*FF*	*CR*	*SO*	*SI*
1...	*DLE*	*DC1*	*DC2*	*DC3*	*DC4*	*NAK*	*SYN*	*ETB*	*CAN*	*EM*	*SUB*	*ESC*	*FS*	*GS*	*RS*	*US*
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

- SP = Leerzeichen, CR = carriage-return, LF = line-feed
- ESC = escape, DEL = delete, BEL = bell usw.

https://de.wikipedia.org/wiki/ASCII

ISO-8859 Familie

- Erweiterung von ASCII um Sonderzeichen und Umlaute
- 8-bit Codierung: bis max. 256 Zeichen darstellbar

- Latin-1: Westeuropäisch
- Latin-2: Mitteleuropäisch
- Latin-3: Südeuropäisch
- Latin-4: Baltisch
- Latin-5: Kyrillisch
- Latin-6: Arabisch
- Latin-7: Griechisch
- usw.

- immer noch nicht für mehrsprachige Dokumente geeignet

ISO-8859-1: Codetabelle (1)

Erweiterung von ASCII für westeuropäische Sprachen

Code	…0	…1	…2	…3	…4	…5	…6	…7	…8	…9	…A	…B	…C	…D	…E	…F
0…	*nicht belegt*															
1…																
2…	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3…	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4…	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5…	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6…	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7…	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8…	*nicht belegt*															
9…																
A…	*NBSP*	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	*SHY*	®	¯
B…	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C…	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D…	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E…	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F…	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

ISO-8859-1: Codetabelle (2)

Sonderzeichen gemeinsam für alle 8859 Varianten

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	*NUL*	*SOH*	*STX*	*ETX*	*EOT*	*ENQ*	*ACK*	*BEL*	*BS*	*HT*	*LF*	*VT*	*FF*	*CR*	*SO*	*SI*
1...	*DLE*	*DC1*	*DC2*	*DC3*	*DC4*	*NAK*	*SYN*	*ETB*	*CAN*	*EM*	*SUB*	*ESC*	*FS*	*GS*	*RS*	*US*
2...	wie ISO/IEC 8859, Windows-125X und US-ASCII															
3...																
4...																
5...																
6...																
7...																DEL
8...	*PAD*	*HOP*	*BPH*	*NBH*	*IND*	*NEL*	*SSA*	*ESA*	*HTS*	*HTJ*	*VTS*	*PLD*	*PLU*	*RI*	*SS2*	*SS3*
9...	*DCS*	*PU1*	*PU2*	*STS*	*CCH*	*MW*	*SPA*	*EPA*	*SOS*	*SGCI*	*SCI*	*CSI*	*ST*	*OSC*	*PM*	*APC*
A...	wie ISO/IEC 8859-1 und Windows-1252															
B...																
C...																
D...																
E...																
F...																

ISO-8859-2

Erweiterung von ASCII für slawische Sprachen

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	*NUL*	*SOH*	*STX*	*ETX*	*EOT*	*ENQ*	*ACK*	*BEL*	*BS*	*HT*	*LF*	*VT*	*FF*	*CR*	*SO*	*SI*
1...	*DLE*	*DC1*	*DC2*	*DC3*	*DC4*	*NAK*	*SYN*	*ETB*	*CAN*	*EM*	*SUB*	*ESC*	*FS*	*GS*	*RS*	*US*
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8...	*PAD*	*HOP*	*BPH*	*NBH*	*IND*	*NEL*	*SSA*	*ESA*	*HTS*	*HTJ*	*VTS*	*PLD*	*PLU*	*RI*	*SS2*	*SS3*
9...	*DCS*	*PU1*	*PU2*	*STS*	*CCH*	*MW*	*SPA*	*EPA*	*SOS*	*SGCI*	*SCI*	*CSI*	*ST*	*OSC*	*PM*	*APC*
A...	*NBSP*	Ą	˘	Ł	¤	Ľ	Ś	§	¨	Š	Ş	Ť	Ź	*SHY*	Ž	Ż
B...	°	ą	˛	ł	´	ľ	ś	ˇ	¸	š	ş	ť	ź	˝	ž	ż
C...	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
D...	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
E...	ŕ	á	â	ă	ä	Í	ć	ç	č	é	ę	ë	ě	í	î	ď
F...	đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	ţ	˙

ISO-8859-15

Modifizierte ISO-8859-1 mit € (0xA4)

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	*NUL*	*SOH*	*STX*	*ETX*	*EOT*	*ENQ*	*ACK*	*BEL*	*BS*	*HT*	*LF*	*VT*	*FF*	*CR*	*SO*	*SI*
1...	*DLE*	*DC1*	*DC2*	*DC3*	*DC4*	*NAK*	*SYN*	*ETB*	*CAN*	*EM*	*SUB*	*ESC*	*FS*	*GS*	*RS*	*US*
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8...	*PAD*	*HOP*	*BPH*	*NBH*	*IND*	*NEL*	*SSA*	*ESA*	*HTS*	*HTJ*	*VTS*	*PLD*	*PLU*	*RI*	*SS2*	*SS3*
9...	*DCS*	*PU1*	*PU2*	*STS*	*CCH*	*MW*	*SPA*	*EPA*	*SOS*	*SGCI*	*SCI*	*CSI*	*ST*	*OSC*	*PM*	*APC*
A...	*NBSP*	¡	¢	£	€	¥	Š	§	š	©	ª	«	¬	*SHY*	®	¯
B...	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
C...	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D...	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E...	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F...	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Microsoft: Codepages 437, 850, 1252

- Zeichensatz des IBM-PC ab 1981
- Erweiterung von ASCII auf einen 8-bit Code
- einige Umlaute (westeuropäisch)
- Grafiksymbole

- `https://de.wikipedia.org/wiki/Codepage_437`
- verbesserte Version: Codepage 850, 858 (€-Symbol an `0xD5`)
- Codepage 1252 entspricht (weitgehend) ISO-8859-1
- Sonderzeichen liegen an anderen Positionen als bei ISO-8859

Microsoft: Codepage 850

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫	☼
1...	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
2...		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂
8...	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9...	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	×	ƒ
A...	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	½	¼	¡	«	»
B...	░	▒	▓	│	┤	Á	Â	À	©	╣	║	╗	╝	¢	¥	┐
C...	└	┴	┬	├	─	┼	ã	Ã	╚	╔	╩	╦	╠	═	╬	¤
D...	ð	Ð	Ê	Ë	È	ı	Í	Î	Ï	┘	┌	█	▄	¦	Ì	▀
E...	Ó	ß	Ô	Ò	õ	Õ	µ	þ	Þ	Ú	Û	Ù	ý	Ý	¯	´
F...		±	‗	¾	¶	§	÷	¸	°	¨	·	¹	³	²	■	

Austausch von Texten?

- die meisten gängigen Codes (abwärts-) kompatibel mit ASCII
- unterschiedliche Codierung für Umlaute (soweit vorhanden)
- unterschiedliche Codierung der Sonderzeichen

- Systemspezifische Konventionen für Zeilenende
  - abhängig von Rechner- und Betriebssystem
  - Konverter-Tools: `dos2unix`, `unix2dos`, `iconv`

Betriebssystem	Zeichensatz	Abkürzung	Hex-Code	Escape
Unix, Linux, Mac OS X, AmigaOS, BSD	ASCII	*LF*	`0A`	\n
Windows, DOS, OS/2, CP/M, TOS (Atari)	ASCII	*CR LF*	`0D 0A`	\r\n
Mac OS bis Version 9, Apple II	ASCII	*CR*	`0D`	\r
AIX OS, OS 390	EBCDIC	*NEL*	15	

Unicode: Motivation

- zunehmende Vernetzung und Globalisierung
- internationaler Datenaustausch?
- Erstellung mehrsprachiger Dokumente?
- Unterstützung orientalischer oder asiatischer Sprachen?

- ASCII oder ISO-8859-1 reicht nicht aus
- temporäre Lösungen konnten sich nicht durchsetzen, z.B:
  ISO-2022: Umschaltung zwischen mehreren Zeichensätzen durch Spezialbefehle (*Escapesequenzen*).

$\Rightarrow$ **Unicode** als System zur Codierung aller Zeichen aller bekannten (lebenden oder toten) Schriftsysteme

Unicode: Versionen und History

- auch abgekürzt als UCS: **Universal Character Set**
- zunehmende Verbreitung (Betriebssysteme, Applikationen)
- Darstellung erfordert auch entsprechende Schriftarten
- `http://www.unicode.org`
  `http://www.unicode.org/charts`

- 1991 1.0.0: europäisch, nahöstlich, indisch
- 1992 1.0.1: ostasiatisch (Han)
- 1993 akzeptiert als ISO/IEC-10646 Standard
- ...
- 2019 12.1.0: inzwischen 137 929 Zeichen
  - Sprachzeichen, Hieroglyphen etc.
  - Symbole: Satzzeichen, Währungen ($ ... ₿), Pfeile, mathematisch, technisch, Braille, Noten etc.
  - Emojis (3 187 aktuell) / Kombinationen

`home.unicode.org`, `unicode.org/main.html`, `de.wikipedia.org/wiki/Unicode`, `unicode-table.com/de`

Unicode: Schreibweise

- ursprüngliche Version nutzt 16-bit pro Zeichen
- die sogenannte „*Basic Multilingual Plane*“
- Schreibweise hexadezimal als `U+xxxx`
- Bereich von `U+0000` . . . `U+FFFF`
- Schreibweise in Java-Strings: `\uxxxx`
  z.B. `\u03A9` für Ω, `\u20AC` für das €-Symbol

- mittlerweile mehr als 2^{16} Zeichen
- Erweiterung um „*Extended Planes*“
- `U+10000` . . . `U+10FFFF`

Unicode: in Webseiten (HTML)

- HTML-Header informiert über verwendeten Zeichensatz
- Unterstützung und Darstellung abhängig vom Browser
- Demo `http://kermitproject.org/utf8.html`

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html;
      charset=utf-8">
<title>UTF-8 Sampler</title>

<META ...
</head>
...
```

1. **English:** The quick brown fox jumps over the lazy dog.
2. **Jamaican:** Chruu, a kwik di kwik brong fox a jomp huova di liezi daag de, yu no siit?
3. **Irish:** "An bfuil do chroí ag bualadh ó fhaitíos an ghrá a mheall lena phóg éada ó shlí do leasa thú?" "D'fhuascail Íosa Úrmhac na hÓighe Beannaithe pór Éava agus Ádhaimh."
4. **Dutch:** Pa's wijze lynx bezag vroom het fikse aquaduct.
5. **German:** Falsches Üben von Xylophonmusik quält jeden größeren Zwerg. (1)
6. **German:** Im finſteren Jagdſchloß am offenen Felsquellwaſſer patzte der affig-flatterhafte kauzig-höfliche Bäcker über ſeinem verſifften kniffligen C-Xylophon. (2)
7. **Norwegian:** Blåbærsyltetøy ("blueberry jam", includes every extra letter used in Norwegian).
8. **Swedish:** Flygande bäckasiner söka strax hwila på mjuka tuvor.
9. **Icelandic:** Sævör grét áðan því úlpan var ónýt.
10. **Finnish:** (5) Törkylempijävongahdus (This is a perfect pangram, every letter appears only once. Translating it is an art on its own, but I'll say "rude lover's yelp". :-D)
11. **Finnish:** (5) Albert osti fagotin ja töräytti puhkuvan melodian. (Albert bought a bassoon and hooted an impressive melody.)
12. **Finnish:** (5) On sangen hauskaa, että polkupyörä on maanteiden jokapäiväinen ilmiö. (It's pleasantly amusing, that the bicycle is an everyday sight on the roads.)
13. **Polish:** Pchnąć w tę łódź jeża lub osiem skrzyń fig.
14. **Czech:** Příliš žluťoučký kůň úpěl ďábelské ódy.
15. **Slovak:** Starý kôň na hŕbe kníh žuje tíško povädnuté ruže, na stĺpe sa ďateľ učí kvákať novú ódu o živote.
16. **Slovenian:** Šerif bo za domačo vajo spet kuhal žgance.
17. **Greek** (monotonic): ξεσκεπάζω την ψυχοφθόρα βδελυγμία
18. **Greek** (polytonic): ξεσκεπάζω τὴν ψυχοφθόρα βδελυγμία
19. **Russian:** Съешь же ещё этих мягких французских булок да выпей чаю.
20. **Russian:** В чащах юга жил-был цитрус? Да, но фальшивый экземпляр! ёъ.
21. **Bulgarian:** Жълтата дюля беше щастлива, че пухът, който цъфна, замръзна като гьон.
22. **Sami (Northern):** Vuol Ruoŧa geđggiid leat máŋga luosa ja čuovžža.
23. **Hungarian:** Árvíztűrő tükörfúrógép.
24. **Spanish:** El pingüino Wenceslao hizo kilómetros bajo exhaustiva lluvia y frío, añoraba a su querido cachorro.
25. **Spanish:** Volé cigüeña que jamás cruzó París, exhibe flor de kiwi y atún.
26. **Portuguese:** O próximo vôo à noite sobre o Atlântico, põe freqüentemente o único médico. (3)
27. **French:** Les naïfs ægithales hâtifs pondant à Noël où il gèle sont sûrs d'être déçus en voyant leurs drôles d'œufs abîmés.
28. **Esperanto:** Eĥoŝanĝo ĉiuĵaŭde.
29. **Hebrew:** .זה כיף סתם לשמוע איך תנצח קרפד עץ טוב בגן
30. **Japanese** (Hiragana):

    いろはにほへど　ちりぬるを
    わがよたれぞ　つねならむ
    うゐのおくやま　けふこえて
    あさきゆめみじ　ゑひもせず (4)

Šota Rustaveli's Veṗxis Ṭq̇aosani, Ϸh, *The Knight in the Tiger's Skin* (Georgian):

> ვეპხის ტყაოსანი შოთა რუსთაველი
>
> ღმერთსი შემვედრე, ნუთუ კვლა დამხსნას სოფლისა შრომასა, ცეცხლს, წყალსა და მიწასა, ჰაერთა თანა მრომასა; მომცნეს ფრთენი და აღვფრინდე, მივჰხვდე მას ჩემსა ნდომასა, დღისით და ღამით ვჰხედვიდე მზისა ელვათა კრთომაასა.

Tamil poetry of Subramaniya Bharathiyar: **சுப்ரமணிய பாரதியார்** (1882-1921):

> **யாமறிந்த மொழிகளிலே தமிழ்மொழி போல் இனிதாவது எங்கும் காணோம்,**
> **பாமரராய் விலங்குகளாய், உலகனைத்தும் இகழ்ச்சிசொலப் பான்மை கெட்டு,**
> **நாமமது தமிழரெனக் கொண்டு இங்கு வாழ்ந்திடுதல் நன்றோ? சொல்லீர்!**
> **தேமதுரத் தமிழோசை உலகமெலாம் பரவும்வகை செய்தல் வேண்டும்.**

Unicode: Latin-Zeichen

- Zeichen im Bereich U+0000 bis U+007F wie ASCII
  www.unicode.org/charts/PDF/U0000.pdf

- Bereich von U+0100 bis U+017F für Latin-A
  Europäische Umlaute und Sonderzeichen
  www.unicode.org/charts/PDF/U0100.pdf

- viele weitere Sonderzeichen ab U+0180
  Latin-B, Latin-C usw.

Unicode: Mathematische Symbole und Operatoren

Vielfältige Auswahl von Symbolen und Operatoren

- griechisch www.unicode.org/charts/PDF/U0370.pdf
- letterlike Symbols www.unicode.org/charts/PDF/U2100.pdf

- Pfeile www.unicode.org/charts/PDF/U2190.pdf
- Operatoren www.unicode.org/charts/PDF/U2A00.pdf
- ...

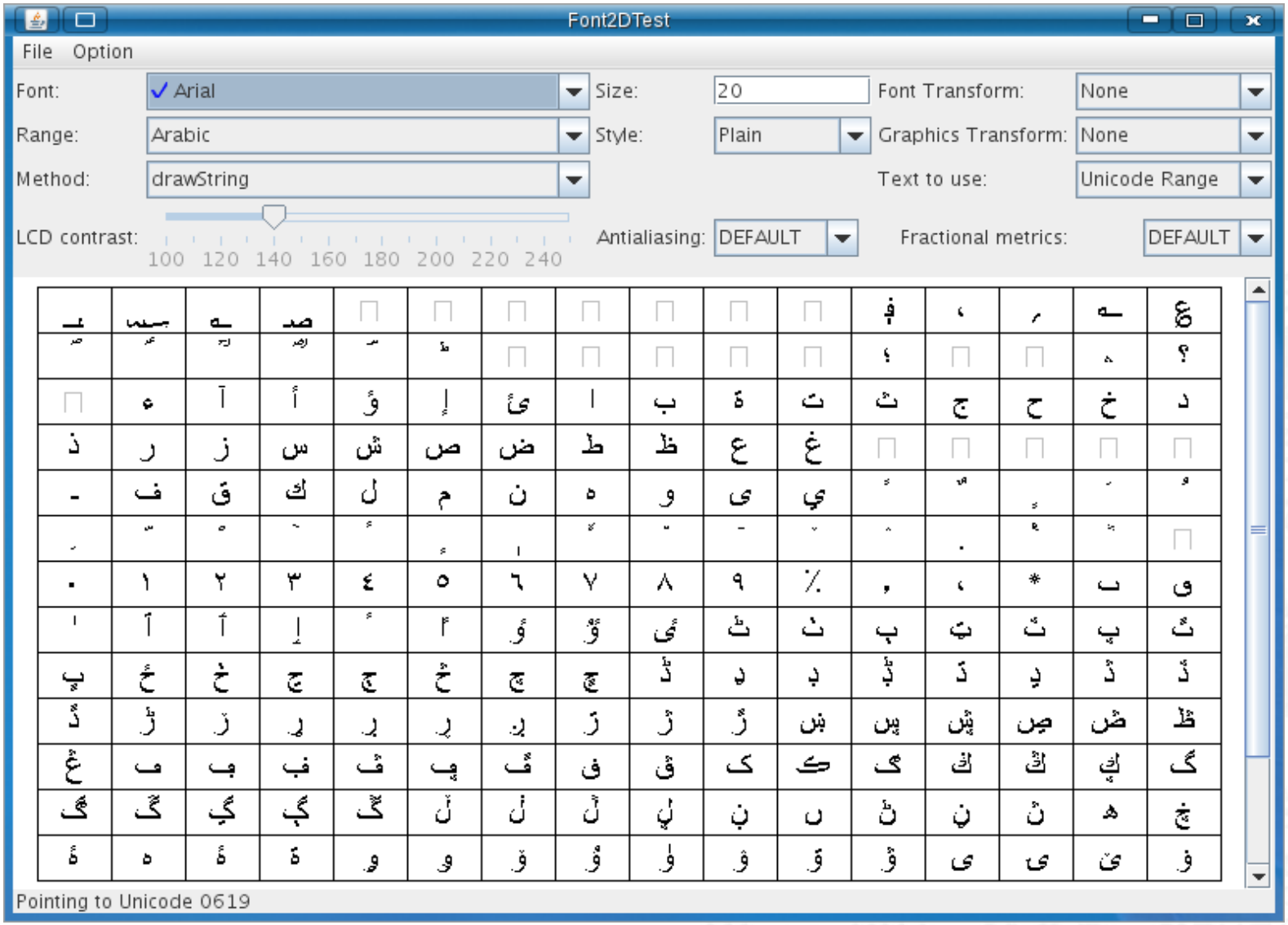
- Dingbats www.unicode.org/charts/PDF/U2700.pdf

Unicode: Asiatische Sprachen

Chinesisch (traditional/simplified), Japanisch, Koreanisch

- U+3400 bis U+4DBF
  www.unicode.org/charts/PDF/U3400.pdf
- U+4E00 bis U+9FCF
  www.unicode.org/charts/PDF/U4E00.pdf

Unicode: Java2D Fontviewer



Oracle [JavaD]: JDK demos and samples . . . /demo/jfc/Font2DTest

Unicode: Repräsentation?

- 16-bit für jedes Zeichen, bis zu 65 536 Zeichen
- schneller Zugriff auf einzelne Zeichen über Arrayzugriffe (Index)
- aber: doppelter Speicherbedarf gegenüber ASCII/ISO-8859-1
- Verwendung u.a. in Java: Datentyp `char`

- ab Unicode 3.0 mehrere *Planes* zu je 65 536 Zeichen
- direkte Repräsentation aller Zeichen erfordert 32-bit/Zeichen
- vierfacher Speicherbedarf gegenüber ISO-8859-1

- bei Dateien ist möglichst kleine Dateigröße wichtig

$\Rightarrow$ Codierung als UTF-8 oder UTF-16

UTF-8

Zeichen	Unicode	Unicode binär	UTF-8 binär	UTF-8 hexadezimal
Buchstabe *y*	U+0079	00000000 01111001	01111001	79
Buchstabe *ä*	U+00E4	00000000 11100100	11000011 10100100	C3 A4
Zeichen für eingetragene Marke ®	U+00AE	00000000 10101110	11000010 10101110	C2 AE
Eurozeichen €	U+20AC	00100000 10101100	11100010 10000010 10101100	E2 82 AC
Violinschlüssel 𝄞	U+1D11E	00000001 11010001 00011110	11110000 10011101 10000100 10011110	F0 9D 84 9E

https://de.wikipedia.org/wiki/UTF-8

- effiziente Codierung von „westlichen“ Unicode-Texten
- Zeichen werden mit variabler Länge codiert, 1 . . . 4-Bytes
- volle Kompatibilität mit ASCII

UTF-8: Algorithmus

Unicode-Bereich (hexadezimal)	UTF-Codierung (binär)	Anzahl (benutzt)
0000 0000 - 0000 007F	0*** ****	128
0000 0080 - 0000 07FF	110* **** 10** ****	1 920
0000 0800 - 0000 FFFF	1110 **** 10** **** 10** ****	63 488
0001 0000 - 0010 FFFF	1111 0*** 10** **** 10** **** 10** ****	bis 2^{21}

- untere 128 Zeichen kompatibel mit ASCII
- Sonderzeichen westlicher Sprachen je zwei Bytes
- führende Eins markiert Multi-Byte Zeichen
- Anzahl der führenden Einsen gibt Anz. Bytegruppen an
- Zeichen ergibt sich als Bitstring aus den $***\ldots*$
- theoretisch bis zu sieben Folgebytes a 6-bit: max. 2^{42} Zeichen

Sprach-Einstellungen: Locale

Locale: die Sprach-Einstellungen und Parameter

- auch: `i18n` („internationalization“)
- Sprache der Benutzeroberfläche
- Tastaturlayout/-belegung
- Zahlen-, Währungs-, Datums-, Zeitformate

- Linux/POSIX: Einstellung über die Locale-Funktionen der Standard C-Library (Befehl `locale`)
  Java: `java.util.Locale`
  Windows: Einstellung über System/Registry-Schlüssel

dos2unix, unix2dos

- Umwandeln von ASCII-Texten (z.B. Programm-Quelltexte) zwischen DOS/Windows und Unix/Linux Maschinen

- Umwandeln von `a.txt` in Ausgabedatei `b.txt`:

```
dos2unix -c ascii -n a.txt b.txt
dos2unix -c iso   -n a.txt b.txt
dos2unix -c mac   -n a.txt b.txt
```

- Umwandeln von Unix nach DOS/Windows, Codepage 850:

```
unix2dos -850     -n a.txt b.txt
```

iconv

Das „Schweizer-Messer“ zur Umwandlung von Textcodierungen

- Optionen
  - `-l` Liste der unterstützten Codierungen ausgeben
  - `-f` ⟨*encoding*⟩ Codierung der Eingabedatei
  - `-t` ⟨*encoding*⟩ Codierung der Ausgabedatei
  - `-o` ⟨*filename*⟩ Name der Ausgabedatei
- Beispiel

```
iconv -f iso-8859-1 -t utf-8 -o foo.utf8.txt foo.txt
```

Base64-Codierung

Übertragung von (Binär-) Dateien zwischen verschiedenen Rechnern?

- SMTP (Internet Mail-Protokoll) verwendet 7-bit ASCII
- bei Netzwerk-Übertragung müssen alle Rechner/Router den verwendeten Zeichensatz unterstützen

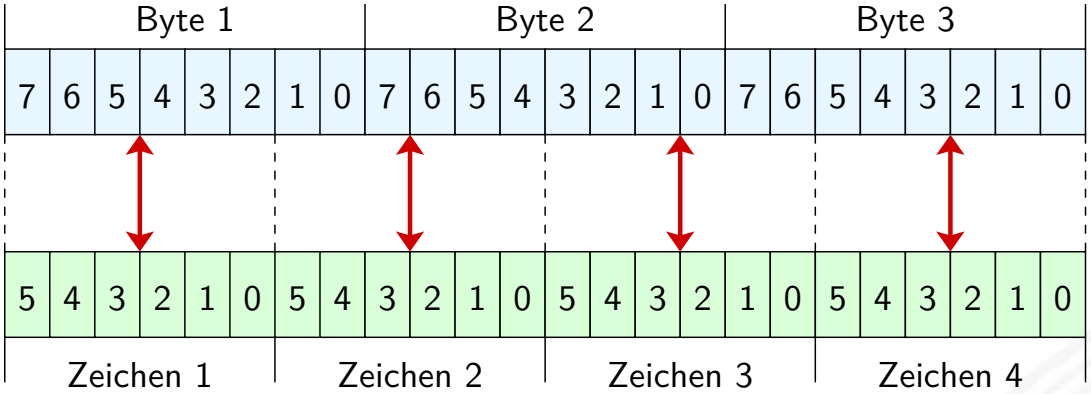
$\Rightarrow$ Verfahren zur Umcodierung der Datei in 7-bit ASCII notwendig

$\Rightarrow$ etabliert ist das **Base64** Verfahren (RFC 2045)

- alle E-Mail Dateianhänge und 8-bit Textdateien
- Umcodierung benutzt nur Buchstaben, Ziffern und drei Sonderzeichen
- Daten werden byteweise in ASCII Symbole umgesetzt

Base64-Codierung: Prinzip

1. Codierung von drei Bytes als vier 6-bit Zeichen



- 3×8-bit $\Leftrightarrow$ 4×6-bit
- 6-bit Binärwerte: $0 \ldots 63$
- nutzen 64 (von 128) 7-bit ASCII Symbolen

2.

Zahl	ASCII	Zuordnung der ASCII-Zeichen
$0 \ldots 25$	A...Z	
$26 \ldots 51$	a...z	
$52 \ldots 61$	0...9	
62	+	
63	/	
	=	Füllzeichen, falls Anz. Bytes nicht durch 3 teilbar
	CR	Zeilenumbruch (opt.), meistens nach 76 Zeichen

Base64-Codierung: Prinzip (cont.)

Text content	**M**								**a**								**n**							
ASCII	77								97								110							
Bit pattern	0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Index	19								22				5				46							
Base64-encoded	**T**								**W**				**F**				**u**							

- drei 8-bit Zeichen, neu gruppiert als vier 6-bit Blöcke
- Zuordnung des jeweiligen Buchstabens/Ziffer
- ggf. =, == am Ende zum Auffüllen
- Übertragung dieser Zeichenfolge ist 7-bit kompatibel
- resultierende Datei ca. 33% größer als das Original

Base64-Codierung: Java

- in neueren Java Versionen (> 1.8) im JDK enthalten

  Module `java.base`, Package `java.util`:
  `Base64Encoder`, bzw. `Base64Decoder`

- diverse andere Packages
  - Apache Commons Codec `http://commons.apache.org/proper/commons-codec`

    `org.apache.commons.codec.binary.Base64InputStream`
    `org.apache.commons.codec.binary.Base64OutputStream`
  - JAXB (Java Architecture for XML Binding)
    in `javax.xml.bind.DatatypeConverter`
    `parseBase64Binary`, `printBase64Binary`

    Beispiel in *Java ist auch eine Insel* [Ull18]

    `openbook.rheinwerk-verlag.de/javainsel/04_008.html#u4.7.4`
  - ...

Literatur

[Uni] The Unicode Consortium; Mountain View, CA.
`home.unicode.org`, `unicode.org/main.html`

[JavaI] Oracle Corporation; Redwood Shores, CA.
The Java Tutorials – Trail: Internationalization.
`docs.oracle.com/javase/tutorial/i18n`

[JavaD] Oracle Corporation: *Java SE Downloads*.
`www.oracle.com/technetwork/java/javase/downloads`

[Ull18] C. Ullenboom: *Java ist auch eine Insel – Einführung, Ausbildung, Praxis*. 14. Auflage, Rheinwerk Verlag GmbH, 2018. ISBN 978–3–8362–6721–2

12. Auflage (Java 8) unter
`openbook.rheinwerk-verlag.de/javainsel`, bzw.
`www.tutego.de/javabuch`

Gliederung

Gliederung (cont.)

Nutzen einer (abstrakten) Algebra?!

Analyse und Beschreibung von

- gemeinsamen, wichtigen Eigenschaften
- mathematischer Operationen
- mit vielfältigen Anwendungen

Spezifiziert durch

- die Art der Elemente (z.B. ganze Zahlen, Aussagen usw.)
- die Verknüpfungen (z.B. Addition, Multiplikation)
- zentrale Elemente (z.B. Null-, Eins-, inverse Elemente)

Anwendungen:	Computerarithmetik	$\rightarrow$ Datenverarbeitung
	Fehlererkennung/-korrektur	$\rightarrow$ Datenübertragung
	Codierung	$\rightarrow$ Repräsentation
	...	

Boole'sche Algebra

- George Boole, 1850: Untersuchung von logischen Aussagen mit den Werten *true* (wahr) und *false* (falsch)
- Definition einer Algebra mit diesen Werten
- drei grundlegende Funktionen:
  - NEGATION (NOT) Schreibweisen: $\neg a$, $\overline{a}$, ~a
  - UND –"– $a \wedge b$, $a \,\&\, b$
  - ODER –"– $a \vee b$, $a \mid b$
  - XOR –"– $a \oplus b$, $a \hat{} b$
- Claude Shannon, 1937: Realisierung der Boole'schen Algebra mit Schaltfunktionen (binäre digitale Logik)

Grundverknüpfungen

- zwei Werte: *wahr* (*true*, 1) und *falsch* (*false*, 0)
- drei grundlegende Verknüpfungen:

NOT(x)

x	
0	1
1	0

AND(x, y)

x \ y	0	1
0	0	0
1	0	1

OR(x, y)

x \ y	0	1
0	0	1
1	1	1

XOR(x, y)

x \ y	0	1
0	0	1
1	1	0

- alle logischen Operationen lassen sich mit diesen Funktionen darstellen

$\Rightarrow$ *vollständige Basismenge*

Anzahl der binären Funktionen

- insgesamt 4 Funktionen mit einer Variable

  $f_0(x) = 0,\ f_1(x) = 1,\ f_2(x) = x,\ f_3(x) = \neg x$

- insgesamt 16 Funktionen zweier Variablen (s. Beispiel)
- allgemein 2^{2^n} Funktionen von n Variablen

- später noch viele Beispiele

Anzahl der binären Funktionen (cont.)

x =	0	1	0	1				
y =	0	0	1	1	**Bezeichnung**	**Notation**	**alternativ**	**Java / C**
	0	0	0	0	Nullfunktion	0		`0`
	0	0	0	1	AND	$x \cap y$	$x \wedge y$	`x&&y`
	0	0	1	0	Inhibition	$x < y$		`x<y`
	0	0	1	1	Identität y	y		`y`
	0	1	0	0	Inhibition	$x > y$		`x>y`
	0	1	0	1	Identität x	x		`x`
	0	1	1	0	XOR	$x \oplus y$	$x \neq y$	`x!=y`
	0	1	1	1	OR	$x \cup y$	$x \vee y$	`x||y`
	1	0	0	0	NOR	$\neg(x \cup y)$	$\overline{x \vee y}$	`!(x||y)`
	1	0	0	1	Äquivalenz	$\neg(x \oplus y)$	$x = y$	`x==y`
	1	0	1	0	NICHT x	$\neg x$	$\overline{x}$	`!x`
	1	0	1	1	Implikation	$x \leq y$	$x \rightarrow y$	`x<=y`
	1	1	0	0	NICHT y	$\neg y$	$\overline{y}$	`!y`
	1	1	0	1	Implikation	$x \geq y$	$x \leftarrow y$	`x>=y`
	1	1	1	0	NAND	$\neg(x \cap y)$	$\overline{x \wedge y}$	`!(x&&y)`
	1	1	1	1	Einsfunktion	1		`1`

Boole'sche Algebra - formale Definition

- 6-Tupel $\langle \{0,1\}, \vee, \wedge, \neg, \ 0,1 \rangle$ bildet eine Algebra

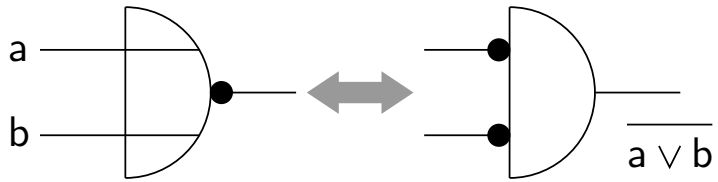
- $\{0,1\}$ Menge mit zwei Elementen
- $\vee$ ist die „Addition“
- $\wedge$ ist die „Multiplikation“
- $\neg$ ist das „Komplement“ (nicht das Inverse!)
- 0 (false) ist das Nullelement der Addition
- 1 (true) ist das Einselement der Multiplikation

Rechenregeln: Ring / Algebra

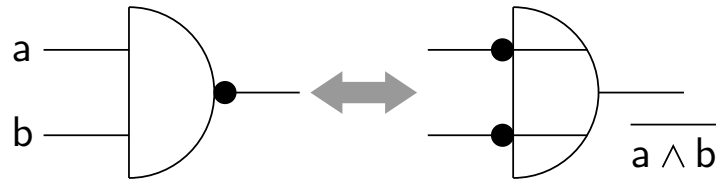
Eigenschaft	Ring der ganzen Zahlen	Boole'sche Algebra
Kommutativgesetz	$a+b=b+a$	$a \vee b = b \vee a$
	$a \cdot b = b \cdot a$	$a \wedge b = b \wedge a$
Assoziativgesetz	$(a+b)+c=a+(b+c)$	$(a \vee b) \vee c = a \vee (b \vee c)$
	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	$(a \wedge b) \wedge c = a \wedge (b \wedge c)$
Distributivgesetz	$a \cdot (b+c) = (a \cdot b)+(a \cdot c)$	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
Identitäten	$a+0=a$	$a \vee 0 = a$
	$a \cdot 1 = a$	$a \wedge 1 = a$
Vernichtung	$a \cdot 0 = 0$	$a \wedge 0 = 0$
Auslöschung	$-(-a) = a$	$\neg(\neg a) = a$
Inverses	$a+(-a)=0$	—
Distributivgesetz	—	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
Komplement	—	$a \vee \neg a = 1$
	—	$a \wedge \neg a = 0$
Idempotenz	—	$a \vee a = a$
	—	$a \wedge a = a$
Absorption	—	$a \vee (a \wedge b) = a$
	—	$a \wedge (a \vee b) = a$
De Morgan Regeln	—	$\neg(a \vee b) = \neg a \wedge \neg b$
	—	$\neg(a \wedge b) = \neg a \vee \neg b$

De Morgan Regeln

$$\neg(a \vee b) = \neg a \wedge \neg b$$



$$\neg(a \wedge b) = \neg a \vee \neg b$$



1. Ersetzen von *UND* durch *ODER* und umgekehrt
   $\Rightarrow$ Austausch der Funktion
2. Invertieren aller Ein- und Ausgänge

Verwendung

- bei der Minimierung logischer Ausdrücke
- beim Entwurf von Schaltungen
- siehe Kapitel *8 Schaltfunktionen* und *9 Schaltnetze*

XOR: Exklusiv-Oder / Antivalenz

$\Rightarrow$ entweder a oder b (ausschließlich)
a ungleich b ($\Rightarrow$ Antivalenz)

- $a \oplus b = (\neg a \wedge b) \vee (a \wedge \neg b)$
  genau einer von den Termen a und b ist wahr
- $a \oplus b = (a \vee b) \wedge \neg(a \wedge b)$
  entweder a ist wahr, oder b ist wahr, aber nicht beide gleichzeitig
- $a \oplus a = 0$

Logische Operationen in Java und C

- Datentyp für Boole'sche Logik
  - Java: Datentyp `boolean`
  - C: implizit für alle Integertypen
- Vergleichsoperationen
- Logische Grundoperationen
- Bitweise logische Operationen
  = parallele Berechnung auf Integer-Datentypen
- Auswertungsreihenfolge
  - Operatorprioritäten
  - Auswertung von links nach rechts
  - (optionale) Klammerung

Vergleichsoperationen

- a == b wahr, wenn *a* gleich *b*
  a != b wahr, wenn *a* ungleich *b*
  a >= b wahr, wenn *a* größer oder gleich *b*
  a > b wahr, wenn *a* größer *b*
  a < b wahr, wenn *a* kleiner *b*
  a <= b wahr, wenn *a* kleiner oder gleich *b*

- Vergleich zweier Zahlen, Ergebnis ist logischer Wert
- Java: Integerwerte alle im Zweierkomplement
  C: Auswertung berücksichtigt signed/unsigned-Typen

Logische Operationen in C

- zusätzlich zu den Vergleichsoperatoren <, <=, ==, !=, >, >=
- drei **logische** Operatoren:

!	logische Negation
&&	logisches UND
||	logisches ODER

- Interpretation der Integerwerte:
  der Zahlenwert *0* $\Leftrightarrow$ logische 0 (false)
  alle anderen Werte $\Leftrightarrow$ logische 1 (true)

$\Rightarrow$ völlig andere Semantik als in der Mathematik Achtung!

$\Rightarrow$ völlig andere Funktion als die bitweisen Operationen

Logische Operationen in C (cont.)

- verkürzte Auswertung von links nach rechts (*shortcut*)
  - Abbruch, wenn Ergebnis feststeht
  - \+ kann zum Schutz von Ausdrücken benutzt werden
  - – kann aber auch Seiteneffekte haben, z.B. Funktionsaufrufe

- Beispiele
  - `(a > b) || ((b != c) && (b <= d))`
  -

Ausdruck	Wert
`!0x41`	`0x00`
`!0x00`	`0x01`
`!!0x00`	`0x00`
`0x69 && 0x55`	`0x01`
`0x69 || 0x55`	`0x01`

Logische Operationen in C: Logisch vs. Bitweise

- der Zahlenwert *0* $\Leftrightarrow$ logische 0 (false)
  alle anderen Werte $\Leftrightarrow$ logische 1 (true)

- Beispiel: `x = 0x66` und `y = 0x93`

bitweise Operation		logische Operation	
Ausdruck	Wert	Ausdruck	Wert
`x`	`0110 0110`	`x`	`0000 0001`
`y`	`1001 0011`	`y`	`0000 0001`
`x & y`	`0000 0010`	`x && y`	`0000 0001`
`x | y`	`1111 0111`	`x || y`	`0000 0001`
`~x | ~y`	`1111 1101`	`!x || !y`	`0000 0000`
`x & ~y`	`0110 0100`	`x && !y`	`0000 0000`

Logische Operationen in C: verkürzte Auswertung

- logische Ausdrücke werden von links nach rechts ausgewertet
- Klammern werden natürlich berücksichtigt
- Abbruch, sobald der Wert eindeutig feststeht (*shortcut*)
- Vor- oder Nachteile möglich (codeabhängig)
  - \+ `(a && 5/a)` niemals Division durch Null. Der Quotient wird nur berechnet, wenn der linke Term ungleich Null ist.
  - \+ `(p && *p++)` niemals Nullpointer-Zugriff. Der Pointer wird nur verwendet, wenn `p` nicht Null ist.

Ternärer Operator

- ⟨*condition*⟩ `?` ⟨*true-expression*⟩ `:` ⟨*false-expression*⟩
- Beispiel: `(x < 0) ? -x : x` Absolutwert von x

Logische Operationen in Java

- Java definiert eigenen Datentyp `boolean`
- elementare Werte `false` und `true`
- alternativ `Boolean.FALSE` und `Boolean.TRUE`
- **keine** Mischung mit Integer-Werten wie in C

- Vergleichsoperatoren <, <=, ==, !=, >, >=
- verkürzte Auswertung von links nach rechts (*shortcut*)

Ternärer Operator

- ⟨*condition*⟩ `?` ⟨*true-expression*⟩ `:` ⟨*false-expression*⟩
- Beispiel: `(x < 0) ? -x : x` Absolutwert von x

Bitweise logische Operationen

Integer-Datentypen doppelt genutzt:

1. Zahlenwerte (Ganzzahl, Zweierkomplement, Gleitkomma)

   arithmetische Operationen: Addition, Subtraktion usw.

2. Binärwerte mit w einzelnen Bits (Wortbreite w)

   Boole'sche Verknüpfungen, bitweise auf allen w Bits

   - Grundoperationen: Negation, UND, ODER, XOR
   - Schiebe-Operationen: shift-left, rotate-right usw.

Bitweise logische Operationen (cont.)

- Integer-Datentypen interpretiert als Menge von Bits

$\Rightarrow$ bitweise logische Operationen möglich

- in Java und C sind vier Operationen definiert:

Negation	`~x`	Invertieren aller einzelnen Bits		
UND	`x&y`	Logisches UND	aller einzelnen Bits	
OR	`x|y`	–"–	ODER	–"–
XOR	`x^y`	–"–	XOR	–"–

- alle anderen Funktionen können damit dargestellt werden

  es gibt insgesamt 2^{2^n} Operationen mit n Operanden

Bitweise logische Operationen: Beispiel


```
x  = 0010 1110
y  = 1011 0011

~x = 1101 0001    alle Bits invertiert
~y = 0100 1100    alle Bits invertiert

x & y = 0010 0010    bitweises UND
x | y = 1011 1111    bitweises ODER
x ^ y = 1001 1101    bitweises XOR
```

Schiebeoperationen

- Ergänzung der bitweisen logischen Operationen
- für alle Integer-Datentypen verfügbar

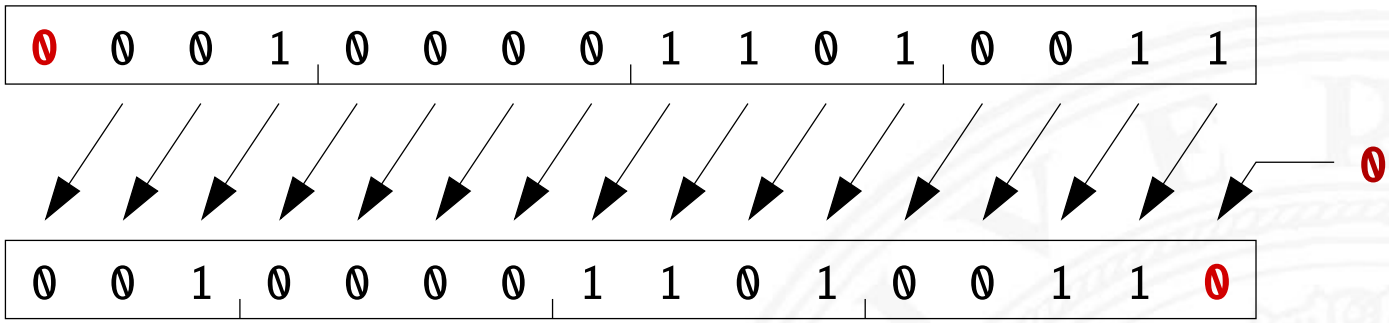
- fünf Varianten

Shift-Left	`shl`
Logical Shift-Right	`srl`
Arithmetic Shift-Right	`sra`
Rotate-Left	`rol`
Rotate-Right	`ror`

- Schiebeoperationen in Hardware leicht zu realisieren
- auf fast allen Prozessoren im Befehlssatz

Shift-Left (`shl`)

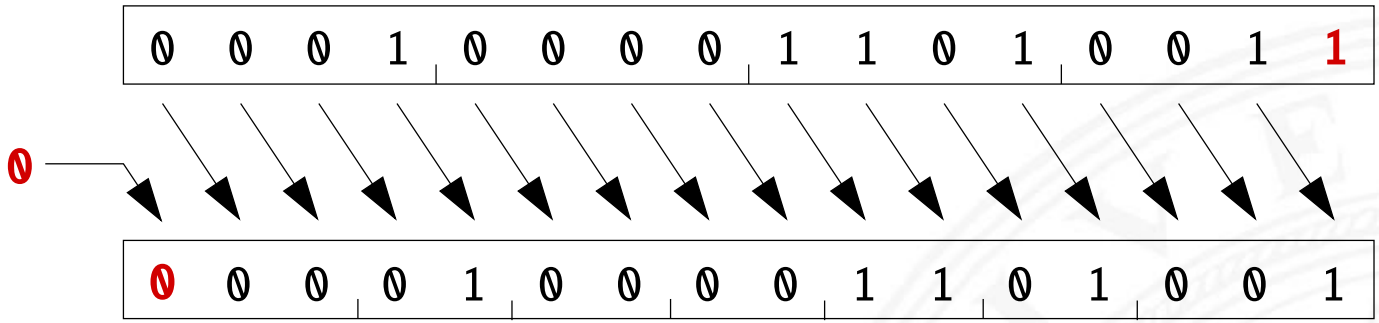
- Verschieben der Binärdarstellung von x um n bits nach links
- links herausgeschobene n bits gehen verloren
- von rechts werden n Nullen eingefügt



- in Java und C direkt als Operator verfügbar: `x << n`
- `shl` um n bits entspricht der Multiplikation mit 2^n

Logical Shift-Right (srl)

- Verschieben der Binärdarstellung von x um n bits nach rechts
- rechts herausgeschobene n bits gehen verloren
- von links werden n Nullen eingefügt



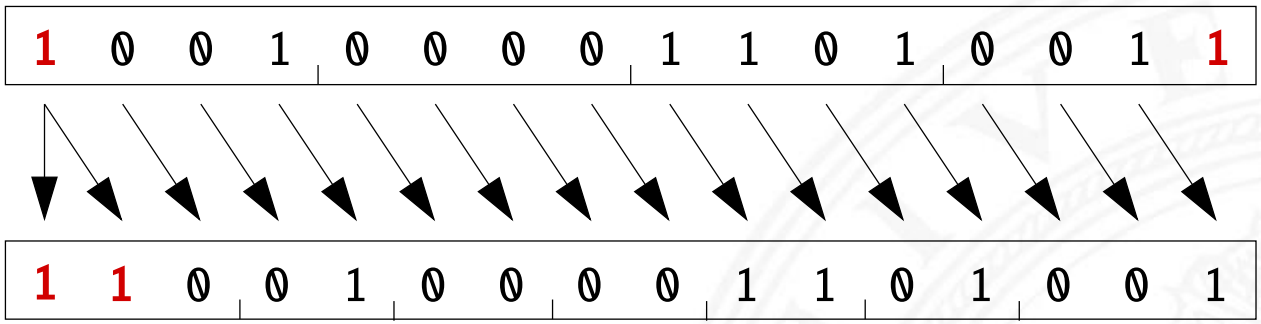
- in Java direkt als Operator verfügbar: `x >>> n`

  in C nur für `unsigned`-Typen definiert: `x >> n`

  für `signed`-Typen nicht vorhanden

Arithmetic Shift-Right (sra)

- Verschieben der Binärdarstellung von x um n bits nach rechts
- rechts herausgeschobene n bits gehen verloren
- von links wird n-mal das MSB (Vorzeichenbit) eingefügt
- Vorzeichen bleibt dabei erhalten (gemäß Zweierkomplement)

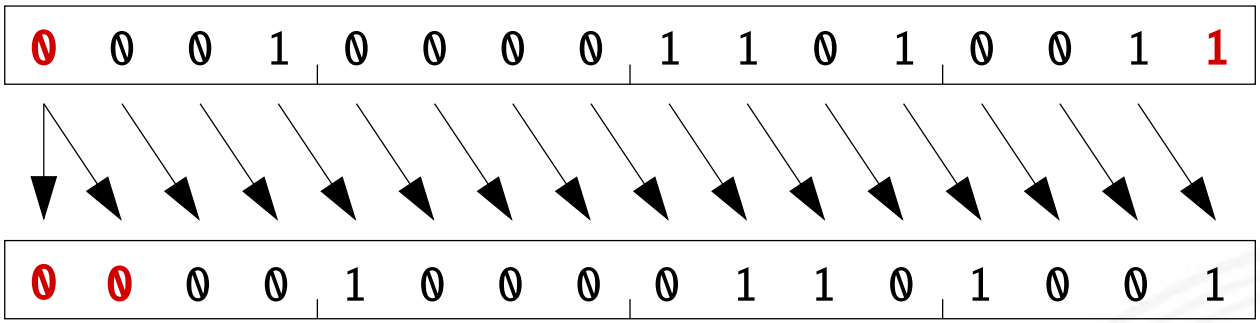


- in Java direkt als Operator verfügbar: `x >> n`

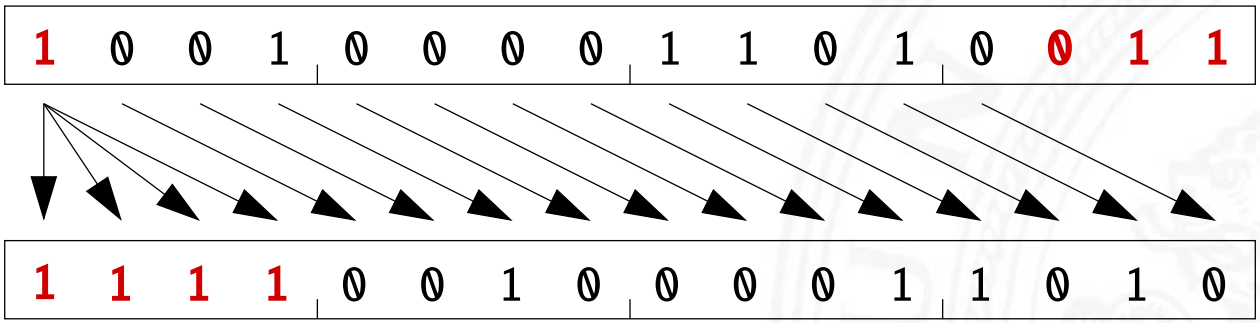
  in C nur für `signed`-Typen definiert: `x >> n`
- `sra` um n bits ist ähnlich der Division durch 2^n

Arithmetic Shift-Right: Beispiel

- `x >> 1` aus `0x10D3` (4 307) wird `0x0869` (2 153)



- `x >> 3` aus `0x90D3` (-28 460) wird `0xF21A` (-3 558)



Arithmetic Shift-Right: Division durch Zweierpotenzen?

- positive Werte: `x >> n` entspricht Division durch 2^n
- negative Werte: `x >> n` ähnlich Division durch 2^n
  aber Ergebnis ist zu klein!

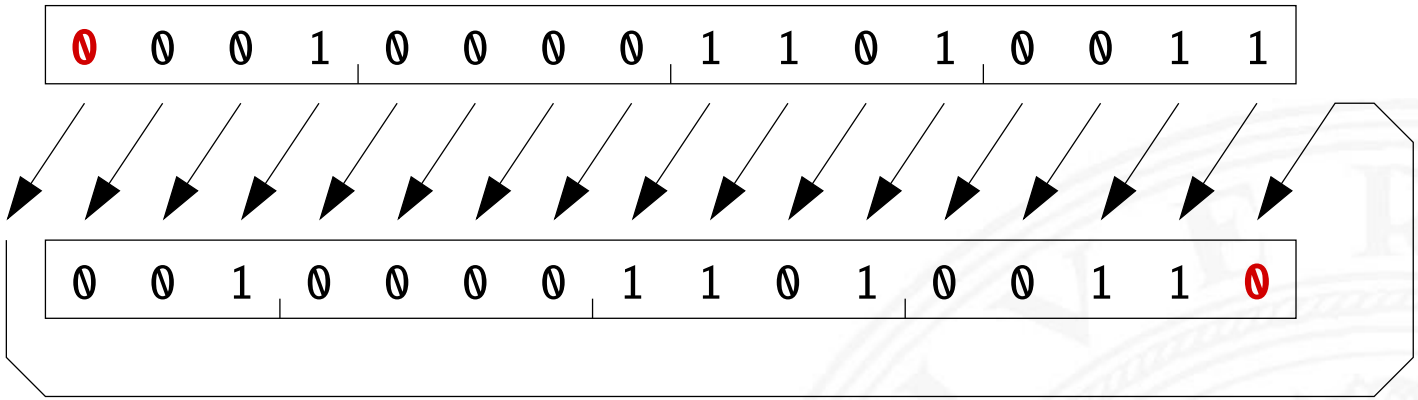
- gerundet in Richtung negativer Werte statt in Richtung Null:

  1111 1011 (-5)
  1111 1101 (-3)
  1111 1110 (-2)
  1111 1111 (-1)

- in C: Kompensation durch Berechnung von `(x + (1<<k)-1) >> k`
  Details: Bryant, O'Hallaron [BO15]

Rotate-Left (rol)

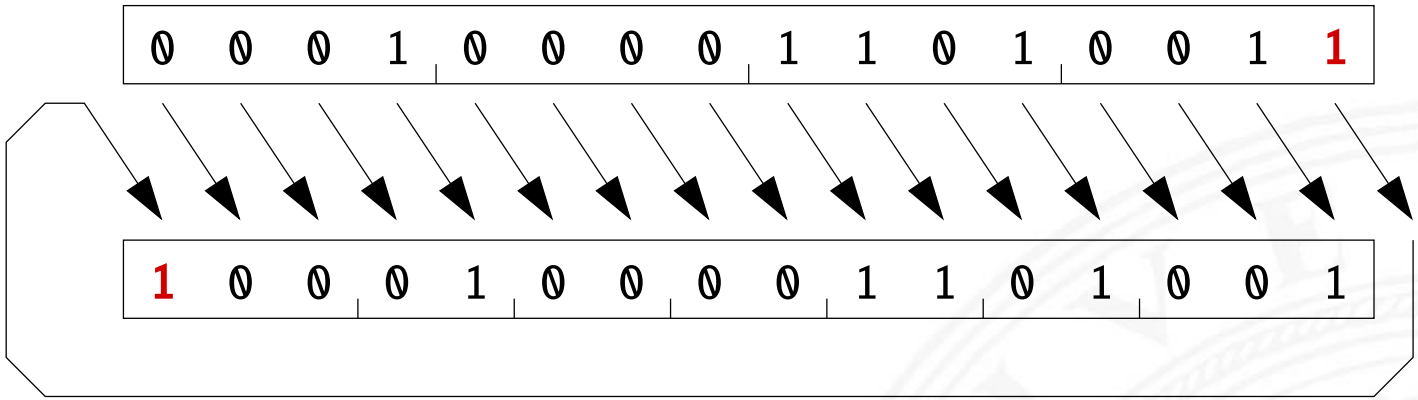
- Rotation der Binärdarstellung von x um n bits nach links
- herausgeschobene Bits werden von rechts wieder eingefügt



- in Java und C nicht als Operator verfügbar
- Java: `Integer.rotateLeft( int x, int distance )`

Rotate Right (ror)

- Rotation der Binärdarstellung von x um n bits nach rechts
- herausgeschobene Bits werden von links wieder eingefügt



- in Java und C nicht als Operator verfügbar
- Java: `Integer.rotateRight( int x, int distance )`

Shifts statt Integer-Multiplikation

- Integer-Multiplikation ist auf vielen Prozessoren langsam oder evtl. gar nicht als Befehl verfügbar

- Add./Subtraktion und logische Operationen: typisch 1 Takt
  Shift-Operationen: meistens 1 Takt

$\Rightarrow$ eventuell günstig, Multiplikation mit Konstanten durch entsprechende Kombination aus shifts+add zu ersetzen

- Beispiel: $9 \cdot x = (8+1) \cdot x$ ersetzt durch `(x << 3) + x`
- viele Compiler erkennen solche Situationen

Beispiel: bit-set, bit-clear

Bits an Position p in einem Integer setzen oder löschen?

- Maske erstellen, die genau eine 1 gesetzt hat
- dies leistet `(1 << p)`, mit $0 \leq p \leq w$ bei Wortbreite w

```
public int bit_set( int x, int pos ) {
  return x |  (1 << pos);       // mask = 0...010...0
}

public int bit_clear( int x, int pos ) {
  return x & ~(1 << pos);       // mask = 1...101...1
}
```

Beispiel: Byte-Swapping *network to/from host*

Linux: /usr/include/bits/byteswap.h (distributionsabhängig)

```
...
/* Swap bytes in 32 bit value.  */
#define __bswap_32(x) \
  ((((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >>  8) |\
   (((x) & 0x0000ff00) <<  8) | (((x) & 0x000000ff) << 24))
...
```

Linux: /usr/include/netinet/in.h

```
...
#  if __BYTE_ORDER == __LITTLE_ENDIAN
#   define ntohl(x) __bswap_32 (x)
#   define ntohs(x) __bswap_16 (x)
#   define htonl(x) __bswap_32 (x)
#   define htons(x) __bswap_16 (x)
#  endif
...
```

Beispiel: RGB-Format für Farbbilder

Farbdarstellung am Monitor / Bildverarbeitung?

- Matrix aus $w \times h$ Bildpunkten
- additive Farbmischung aus Rot, Grün, Blau
- pro Farbkanal typischerweise 8-bit, Wertebereich $0 \ldots 255$
- Abstufungen ausreichend für (untrainiertes) Auge

- je ein 32-bit Integer pro Bildpunkt
- typisch: `0x00RRGGBB` oder `0xAARRGGBB`
- je 8-bit für Alpha/Transparenz, rot, grün, blau

- `java.awt.image.BufferedImage(TYPE_INT_ARGB)`

Beispiel: RGB-Rotfilter


```
public BufferedImage redFilter( BufferedImage src ) {
  int    w = src.getWidth();
  int    h = src.getHeight();
  int type = BufferedImage.TYPE_INT_ARGB;
  BufferedImage dest = new BufferedImage( w, h, type );

  for( int y=0; y < h; y++ ) {          // alle Zeilen
    for( int x=0; x < w; x++ ) {        // von links nach rechts
      int  rgb = src.getRGB( x, y ); // Pixelwert bei (x,y)
                                     // rgb = 0xAARRGGBB
      int  red = (rgb & 0x00FF0000); // Rotanteil maskiert
      dest.setRGB( x, y, red );
    }
  }
  return dest;
}
```

Beispiel: RGB-Graufilter


```
public BufferedImage grayFilter( BufferedImage src ) {
  ...
  for( int y=0; y < h; y++ ) {   // alle Zeilen
    for( int x=0; x < w; x++ ) { // von links nach rechts
      int   rgb = src.getRGB( x, y );        // Pixelwert
      int   red = (rgb & 0x00FF0000) >>>16; // Rotanteil
      int green = (rgb & 0x0000FF00) >>> 8; // Grünanteil
      int  blue = (rgb & 0x000000FF);        // Blauanteil

      int  gray = (red + green + blue) / 3; // Mittelung

      dest.setRGB( x, y, (gray<<16)|(gray<<8)|gray );
    }
  }
  ...
}
```

Beispiel: Bitcount – while-Schleife

Anzahl der gesetzten Bits in einem Wort?

- Anwendung z.B. für Kryptalgorithmen (Hamming-Abstand)
- Anwendung für Medienverarbeitung

```
public static int bitcount( int x ) {
  int count = 0;

  while( x != 0 ) {
    count += (x & 0x00000001);  // unterstes bit addieren
    x = x >>> 1;                // 1-bit rechts-schieben
  }

  return count;
}
```

Beispiel: Bitcount – parallel, tree

- Algorithmus mit Schleife ist einfach aber langsam
- schnelle parallele Berechnung ist möglich

```
int BitCount(unsigned int u)
{ unsigned int uCount;
  uCount = u - ((u >> 1) & 03333333333)
             - ((u >> 2) & 011111111111);
  return ((uCount + (uCount >> 3)) & 030707070707) % 63;
}
```

- java.lang.Integer.bitCount()

```
public static int bitCount(int i) {
    // HD, Figure 5-2
    i = i - ((i >>> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
    i = (i + (i >>> 4)) & 0x0f0f0f0f;
    i = i + (i >>> 8);
    i = i + (i >>> 16);
    return i & 0x3f;
}
```

Beispiel: Bitcount – parallel, tree (cont.)

- viele Algorithmen: bit-Maskierung und Schieben
  - `http://gurmeet.net/puzzles/fast-bit-counting-routines`
  - `http://graphics.stanford.edu/~seander/bithacks.html`
  - `https://tekpool.wordpress.com/category/bit-count/`
  - D. E. Knuth: *The Art of Computer Programming*: Volume 4A, Combinational Algorithms: Part1, Abschnitt 7.1.3 [Knu09]
- viele neuere Prozessoren/DSPs: eigener bitcount-Befehl

Tipps & Tricks: Rightmost bits

D. E. Knuth: *The Art of Computer Programming*, Vol 4.1 [Knu09]

Grundidee: am weitesten rechts stehenden 1-Bits / 1-Bit Folgen erzeugen Überträge in arithmetischen Operationen

- Integer x, mit $x = (\alpha\, 0\, [1]^a\, 1\, [0]^b)_2$

  beliebiger Bitstring α, eine Null, dann $a+1$ Einsen und b Nullen, mit $a \geq 0$ und $b \geq 0$.
- Ausnahmen: $x = -2^b$ und $x = 0$

$$
\begin{aligned}
\Rightarrow \qquad x &= (\alpha\, 0\, [1]^a\, 1\, [0]^b)_2 \\
\overline{x} &= (\overline{\alpha}\, 1\, [0]^a\, 0\, [1]^b)_2 \\
x-1 &= (\alpha\, 0\, [1]^a\, 0\, [1]^b)_2 \\
-x &= (\overline{\alpha}\, 1\, [0]^a\, 1\, [0]^b)_2
\end{aligned}
$$

$$
\Rightarrow \quad \overline{x} + 1 = -x = \overline{x-1}
$$

Literatur

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

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–8689–4238–5

Literatur (cont.)

[Knu09] D.E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009. ISBN 978–0–321–58050–4

[Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript*. Universität Hamburg, FB Informatik, 2005. `tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1`

Gliederung

Gliederung (cont.)

Definition: Codierung

Unter **Codierung** versteht man das Umsetzen einer vorliegenden Repräsentation A in eine andere Repräsentation B

- häufig liegen beide Repräsentationen A und B in derselben Abstraktionsebene
- die Interpretation von B nach A muss eindeutig sein
- eine **Umcodierung** liegt vor, wenn die Interpretation umkehrbar eindeutig ist

Code, Codewörter

- **Codewörter**: die Wörter der Repräsentation B aus einem Zeichenvorrat Z
- **Code**: die Menge aller Codewörter
- **Blockcode**: alle Codewörter haben dieselbe Länge

- **Binärzeichen**: der Zeichenvorrat z enthält genau zwei Zeichen
- **Binärwörter**: Codewörter aus Binärzeichen
- **Binärcode**: alle Codewörter sind Binärwörter

Gründe für den Einsatz von Codes

- effiziente Darstellung und Verarbeitung von Information
- Datenkompression, -reduktion
- Sicherheitsaspekte

- Übertragung von Information
  - Verkleinerung der zu übertragenden Datenmenge
  - Anpassung an die Technik des Übertragungskanals
  - Fehlererkennende und -korrigierende Codes

- Sicherheit von Information
  - Geheimhaltung, z.B. Chiffrierung in der Kryptologie
  - Identifikation, Authentifikation

Wichtige Aspekte

Unterteilung gemäß der Aufgabenstellung

- **Quellencodierung**: Anpassung an Sender/Quelle
- **Kanalcodierung**: Anpassung an Übertragungsstrecke
- **Verarbeitungscodierung**: im Rechner

- sehr unterschiedliche Randbedingungen und Kriterien für diese Teilbereiche: zum Beispiel sind fehlerkorrigierende Codes bei der Nachrichtenübertragung essenziell, im Rechner wegen der hohen Zuverlässigkeit weniger wichtig

Darstellung von Codes

- **Wertetabellen**
  - jede Zeile enthält das Urbild (zu codierende Symbol) und das zugehörige Codewort
  - sortiert, um das Auffinden eines Codeworts zu erleichtern
  - technische Realisierung durch Ablegen der Wertetabelle im Speicher, Zugriff über Adressierung anhand des Urbilds
- **Codebäume**
  - Anordnung der Symbole als Baum
  - die zu codierenden Symbole als Blätter
  - die Zeichen an den Kanten auf dem Weg von der Wurzel zum Blatt bilden das Codewort
- **Logische Gleichungen**
- **Algebraische Ausdrücke**

Codierung von Text

- siehe letzte Woche
- Text selbst als Reihenfolge von Zeichen
- ASCII, ISO-8859 und Varianten, Unicode, UTF-8

Für geschriebenen (formatierten) Text:

- Trennung des reinen Textes von seiner Formatierung
- Formatierung: Schriftart, Größe, Farbe usw.
- diverse applikationsspezifische Binärformate
- Markup-Sprachen (SGML, HTML)

Codierungen für Dezimalziffern

	BCD	Gray	Exzess-3	Aiken	biquinär	1-aus-10	2-aus-5
0	0000	0000	0011	0000	000001	0000000001	11000
1	0001	0001	0100	0001	000010	0000000010	00011
2	0010	0011	0101	0010	000100	0000000100	00101
3	0011	0010	0110	0011	001000	0000001000	00110
4	0100	0110	0111	0100	010000	0000010000	01001
5	0101	0111	1000	1011	100001	0000100000	01010
6	0110	0101	1001	1100	100010	0001000000	01100
7	0111	0100	1010	1101	100100	0010000000	10001
8	1000	1100	1011	1110	101000	0100000000	10010
9	1001	1101	1100	1111	110000	1000000000	10100

- alle Codes der Tabelle sind Binärcodes
- alle Codes der Tabelle sind Blockcodes
- jede Spalte der Tabelle listet alle Codewörter eines Codes

Codierungen für Dezimalziffern (cont.)

- jede Wandlung von einem Code der Tabelle in einen anderen Code ist eine Umcodierung
- aus den Codewörtern geht nicht hervor, welcher Code vorliegt

- Dezimaldarstellung in Rechnern unüblich, die obigen Codes werden also kaum noch verwendet

Begriffe für Binärcodes

- **Minimalcode**: alle $N = 2^n$ Codewörter bei Wortlänge n werden benutzt
- **Redundanter Code**: nicht alle möglichen Codewörter werden benutzt
- **Gewicht**: Anzahl der Einsen in einem Codewort
- **komplementär**: zu jedem Codewort c existiert ein gültiges Codewort $\overline{c}$
- **einschrittig**: aufeinanderfolgende Codewörter unterscheiden sich nur an einer Stelle
- **zyklisch (einschr.)**: bei n geordneten Codewörtern ist $c_0 = c_n$

Dualcode

- der Name für Codierung der Integerzahlen im Stellenwertsystem
- Codewort

$$c = \sum_{i=0}^{n-1} a_i \cdot 2^i, \qquad a_i \in \{0, 1\}$$

- alle Codewörter werden genutzt: Minimalcode
- zu jedem Codewort existiert ein komplementäres Codewort
- bei fester Wortbreite ist c_0 gleich $c_n \Rightarrow$ zyklisch
- nicht einschrittig

Einschrittige Codes

- möglich für Mengen mit Ordnungsrelation
- Elemente der Menge werden durch Binärwörter codiert
- **einschrittiger Code**: die Codewörter für benachbarte Elemente der Menge unterscheiden sich in genau einer Stelle
- **zyklisch einschrittig**: das erste und letzte Wort des Codes unterscheiden sich ebenfalls genau in einer Stelle

- Einschrittige Codes werden benutzt, wenn ein Ablesen der Bits auch beim Wechsel zwischen zwei Codeworten möglich ist (bzw. nicht verhindert werden kann)

  z.B.: Winkelcodierscheiben oder digitale Schieblehre
- viele interessante Varianten möglich (s. Knuth: *AoCP* [Knu11])

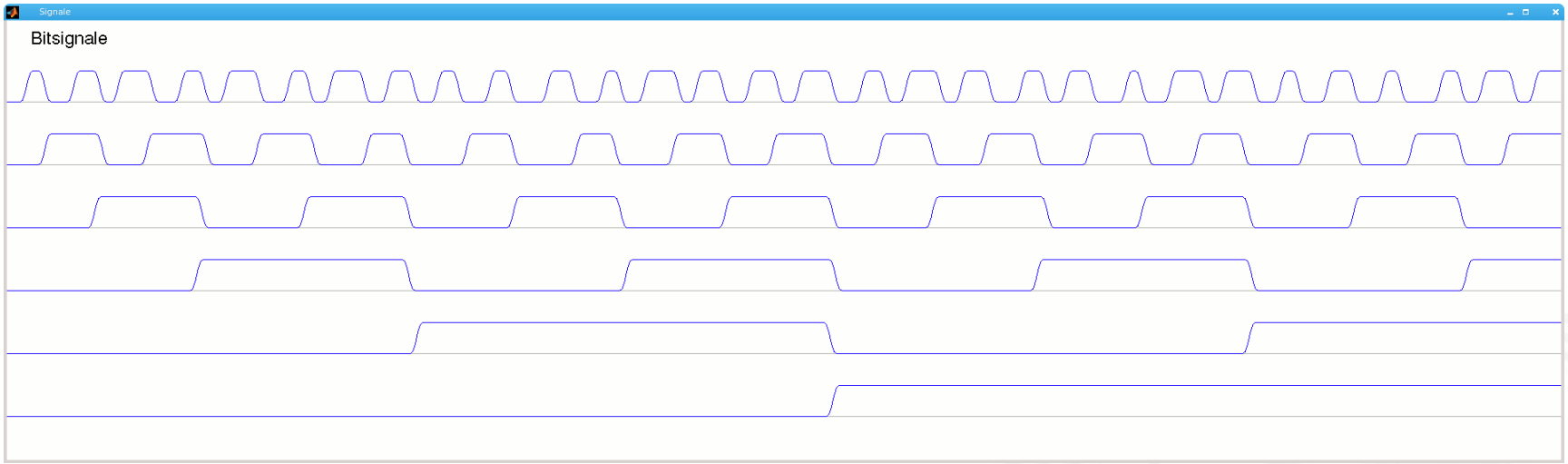
Einschrittige Codes: Matlab-Demo

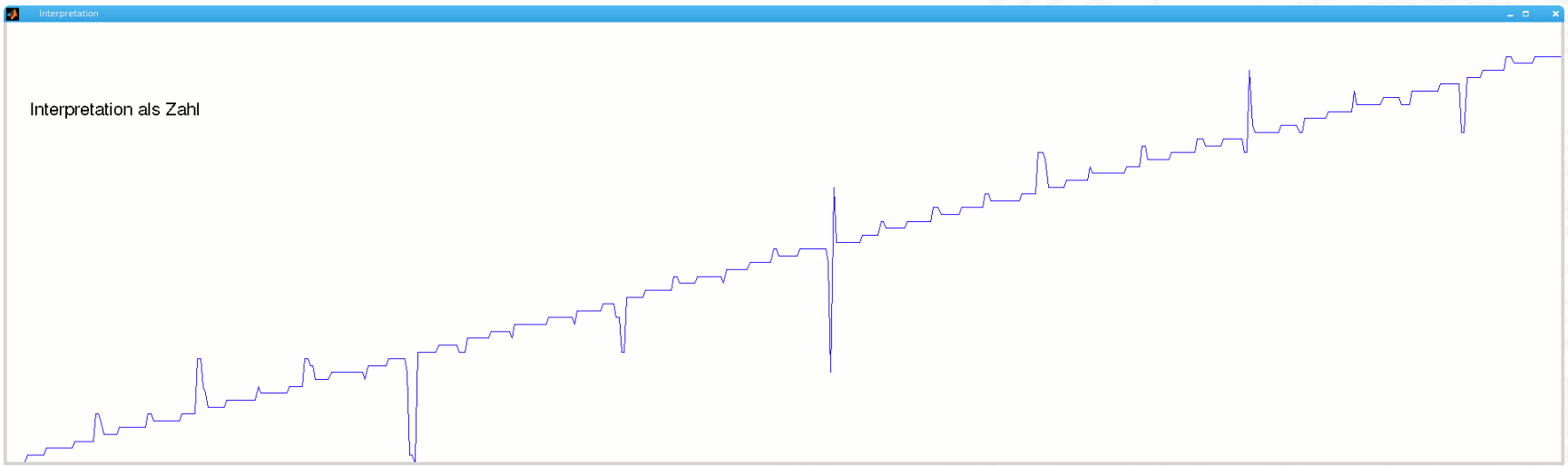
- Ablesen eines Wertes mit leicht gegeneinander verschobenen Übergängen der Bits [Hei05], Kapitel 1.4
  - `demoeinschritt(0:59)` normaler Dualcode
  - `demoeinschritt(einschritt(60))` einschrittiger Code

- maximaler Ablesefehler
  - 2^{n-1} beim Dualcode
  - 1 beim einschrittigen Code

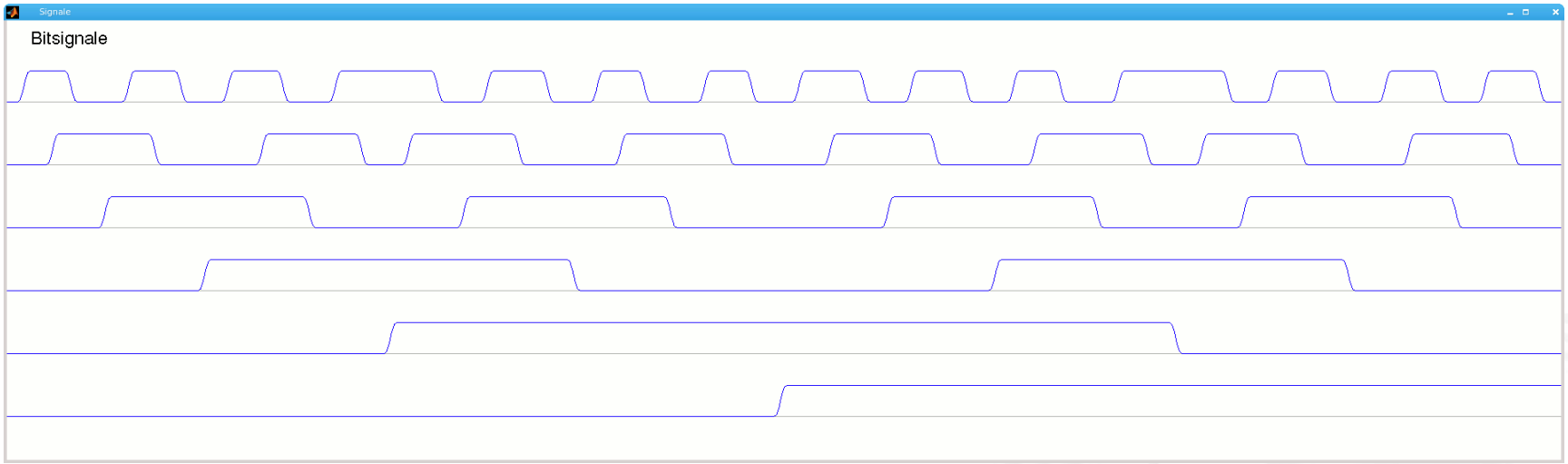
- Konstruktion eines einschrittigen Codes
  - rekursiv
  - als ununterbrochenen Pfad im KV-Diagramm (s.u.)

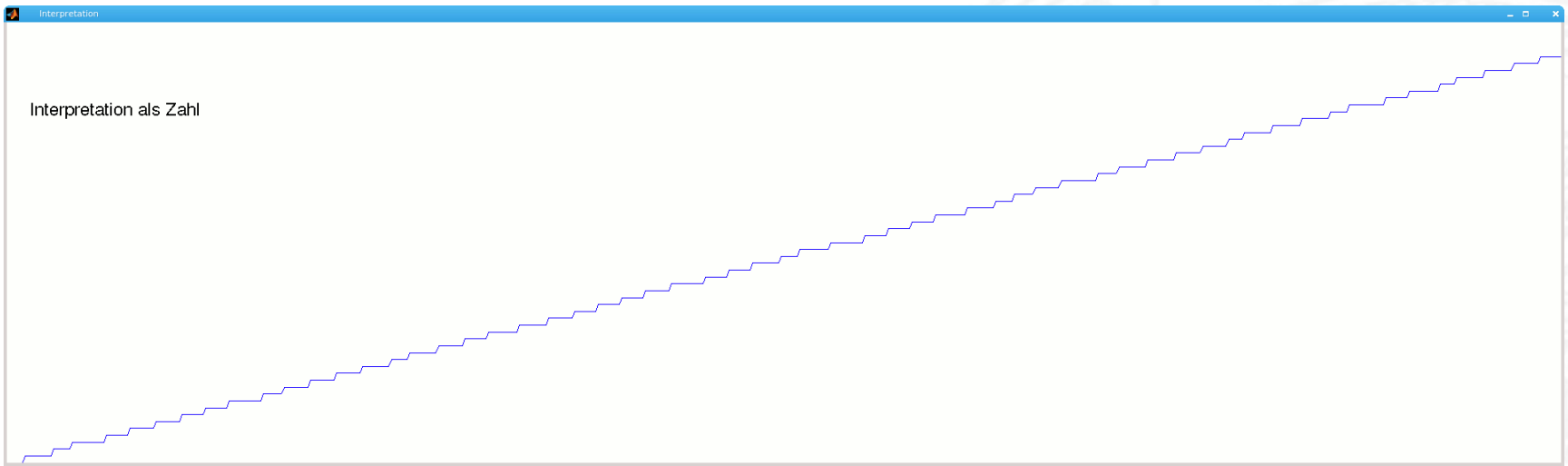
Ablesen des Wertes aus Dualcode



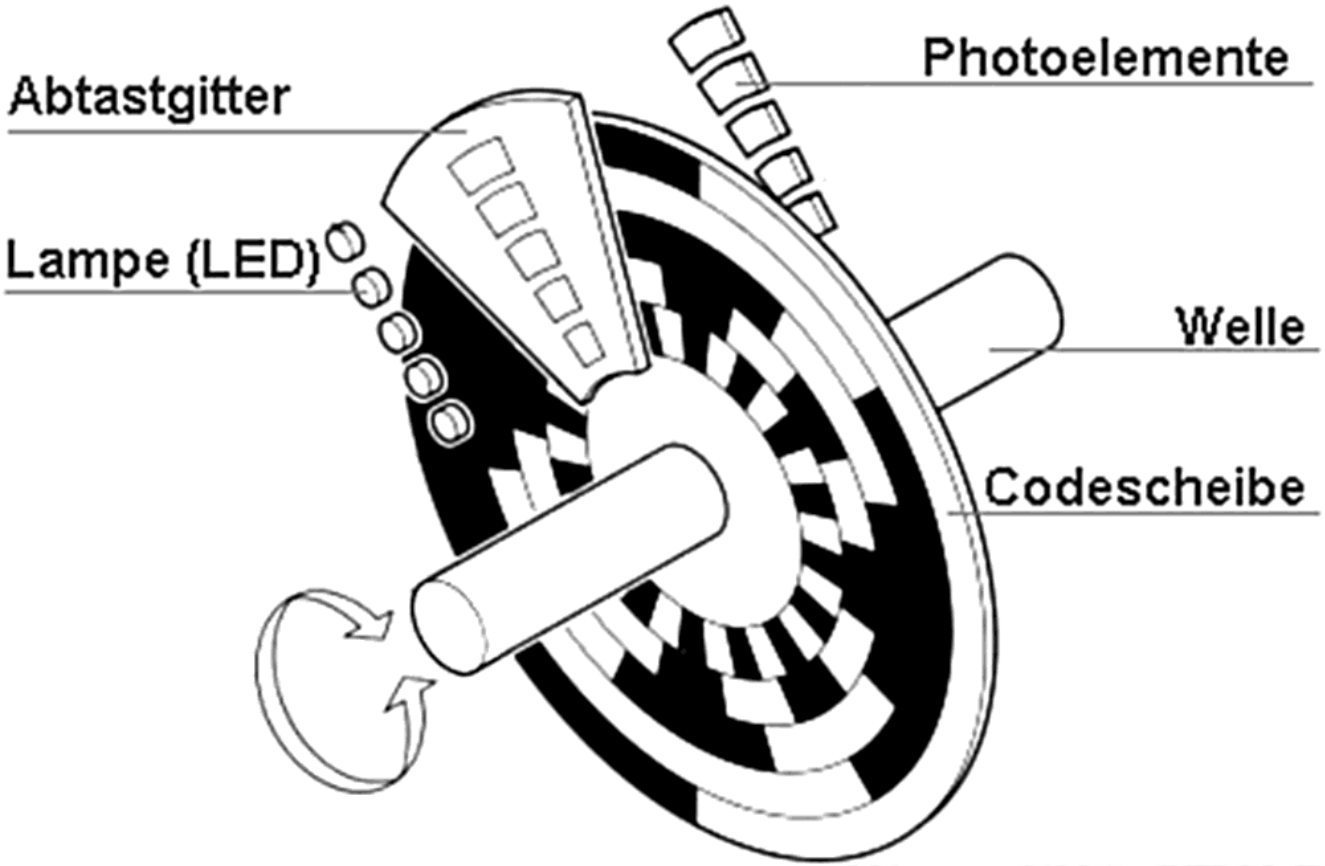


Ablesen des Wertes aus einschrittigem Code

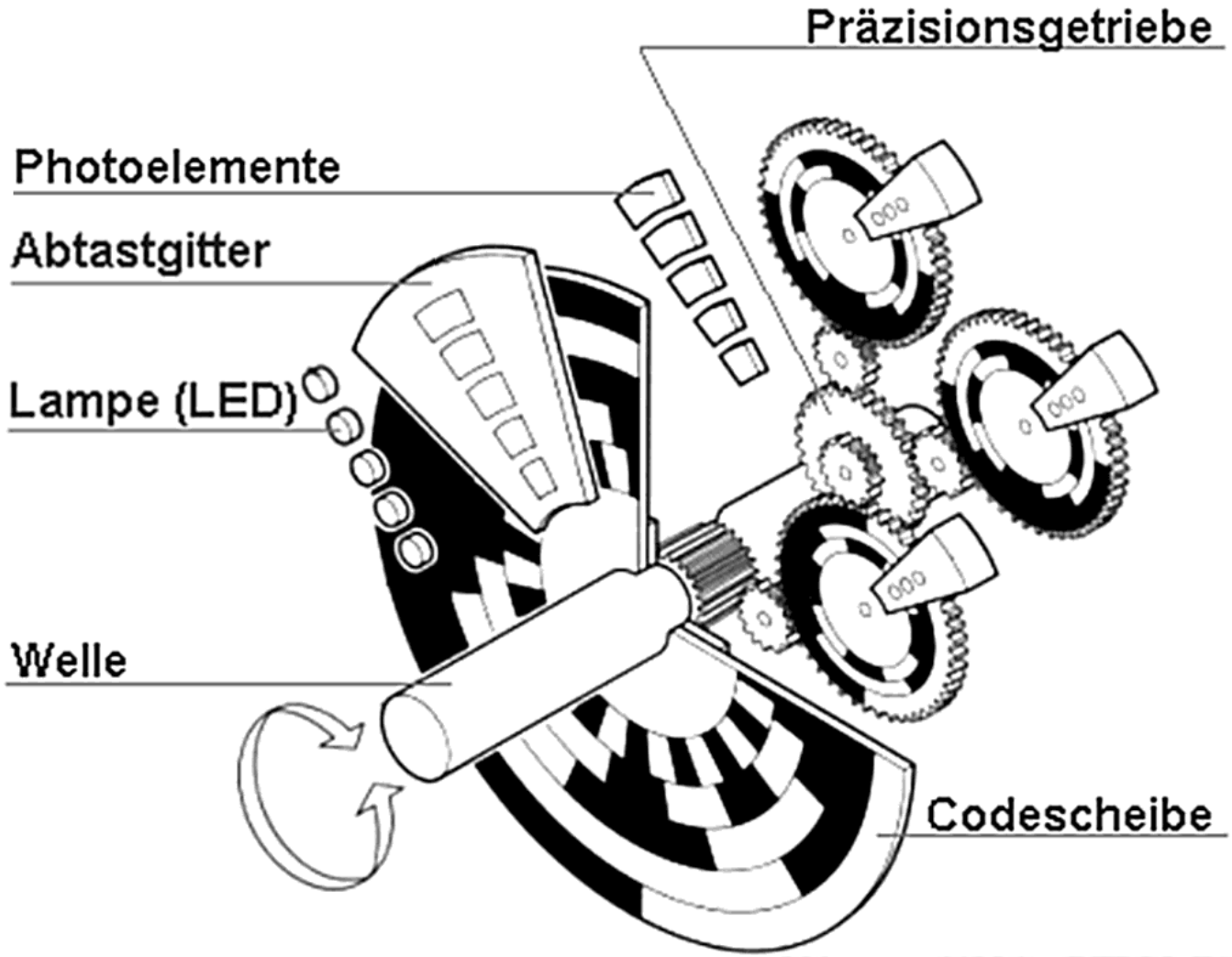




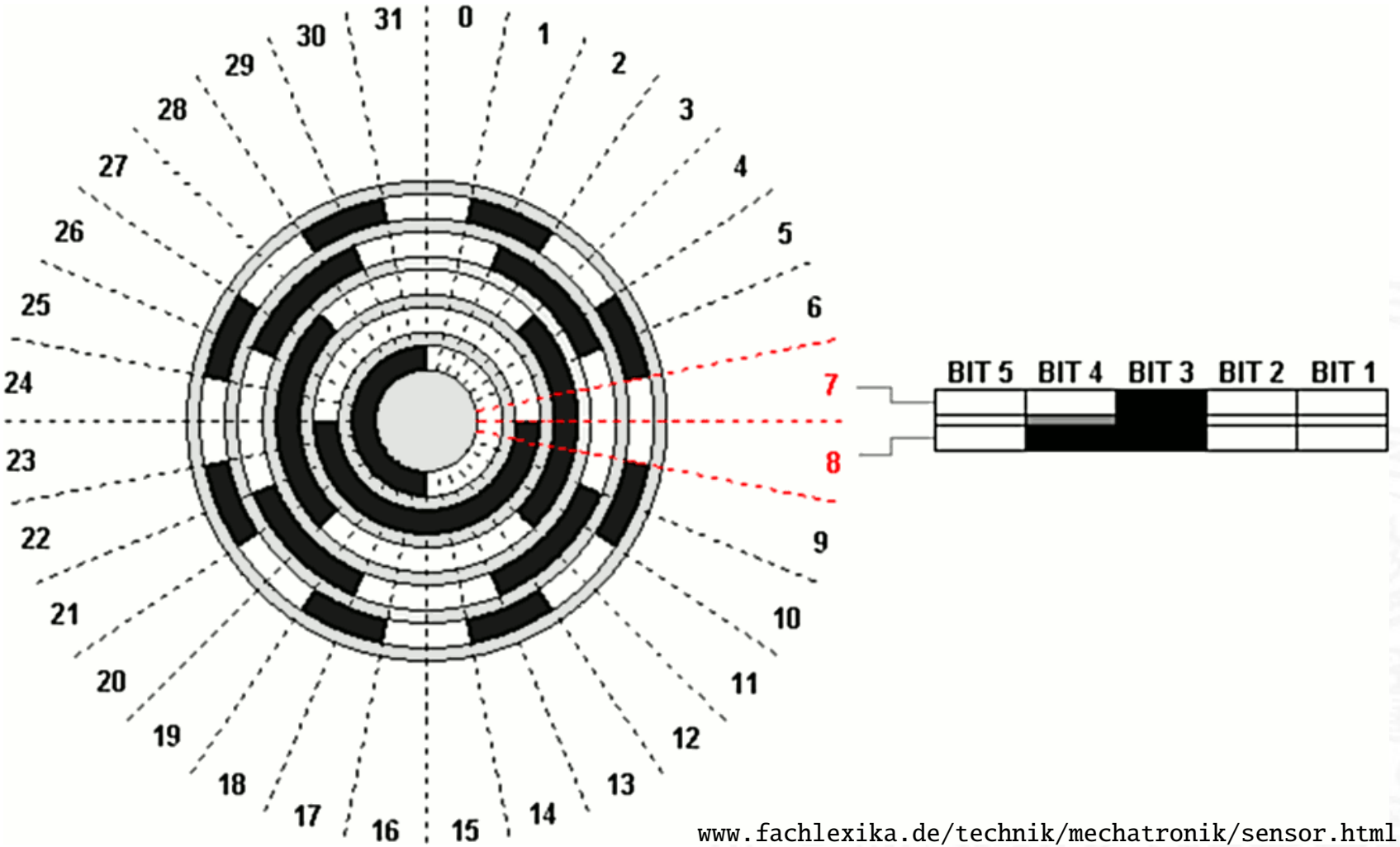
Gray-Code: Prinzip eines Winkeldrehgebers



Gray-Code: mehrstufiger Drehgeber

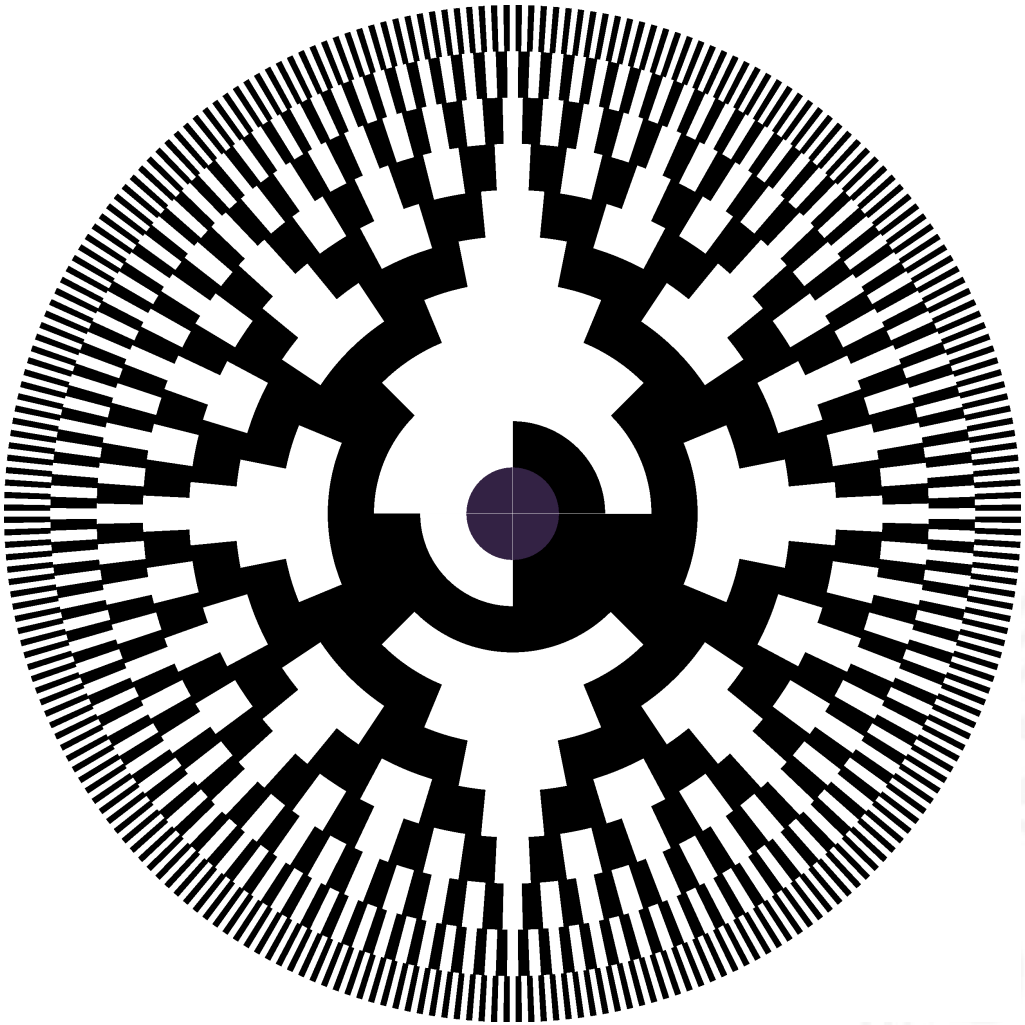


Gray-Code: 5-bit Codierscheibe



www.fachlexika.de/technik/mechatronik/sensor.html

Gray-Code: 10-bit Codierscheibe



Einschrittiger Code: rekursive Konstruktion

- Starte mit zwei Codewörtern: 0 und 1

- Gegeben: Einschrittiger Code C mit n Codewörtern
- Rekursion: Erzeuge Code C_2 mit (bis zu) $2n$ Codewörtern
  1. hänge eine führende 0 vor alle vorhandenen n Codewörter
  2. hänge eine führende 1 vor die in umgekehrter Reihenfolge notierten Codewörter

$\{ 0, 1 \}$

$\{ 00, 01, \quad 11, 10 \}$

$\{ 000, 001, 011, 010, \quad 110, 111, 101, 100 \}$

...

$\Rightarrow$ Gray-Code

Karnaugh-Veitch Diagramm

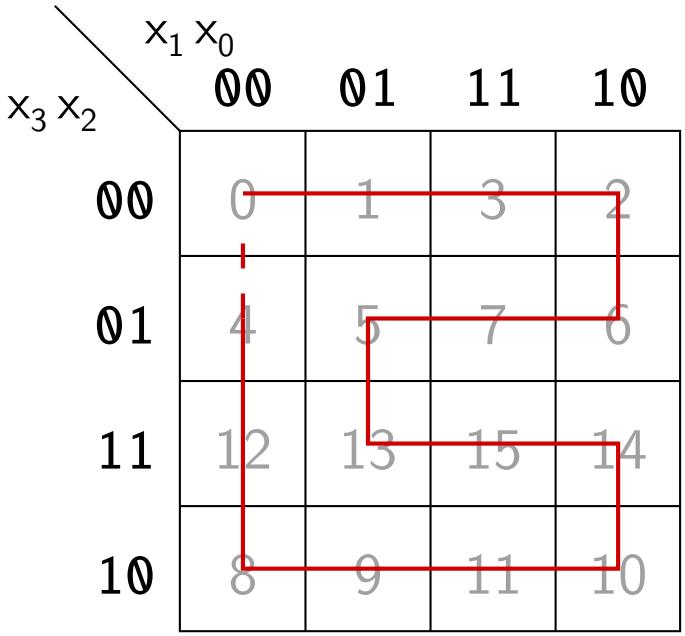
$x_3\,x_2$ \ $x_1\,x_0$	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

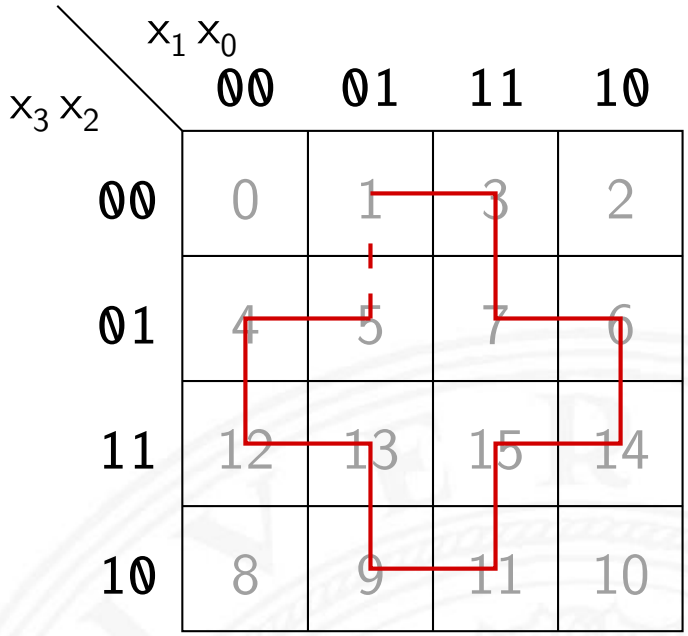
$x_3\,x_2$ \ $x_1\,x_0$	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

- 2D-Diagramm mit $2^n = 2^{n_y} \times 2^{n_x}$ Feldern
- gängige Größen sind: 2×2, 2×4, 4×4
  darüber hinaus: mehrere Diagramme der Größe 4×4
- Anordnung der Indizes ist im einschrittigen-Code / Gray-Code

$\Rightarrow$ benachbarte Felder unterscheiden sich gerade um 1 Bit

Einschrittiger Code: KV-Diagramm



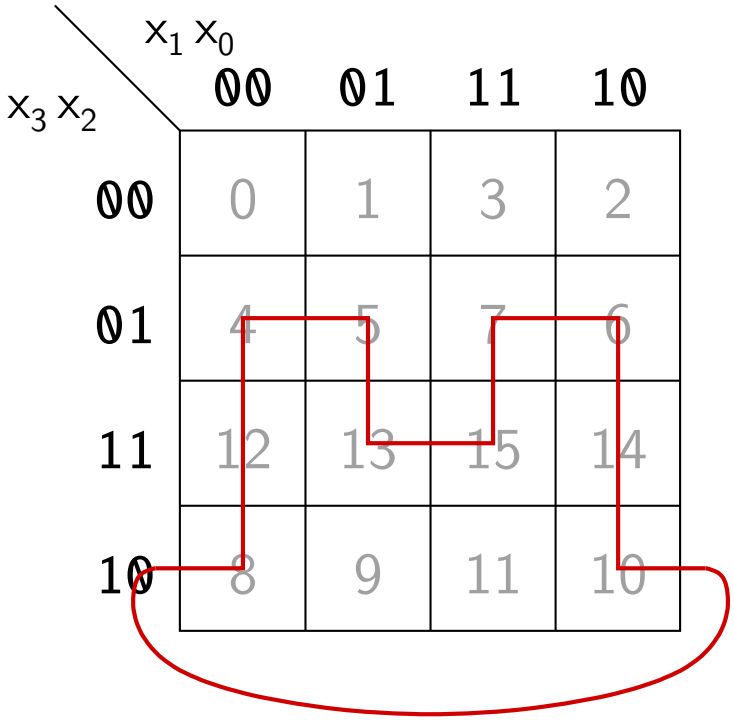


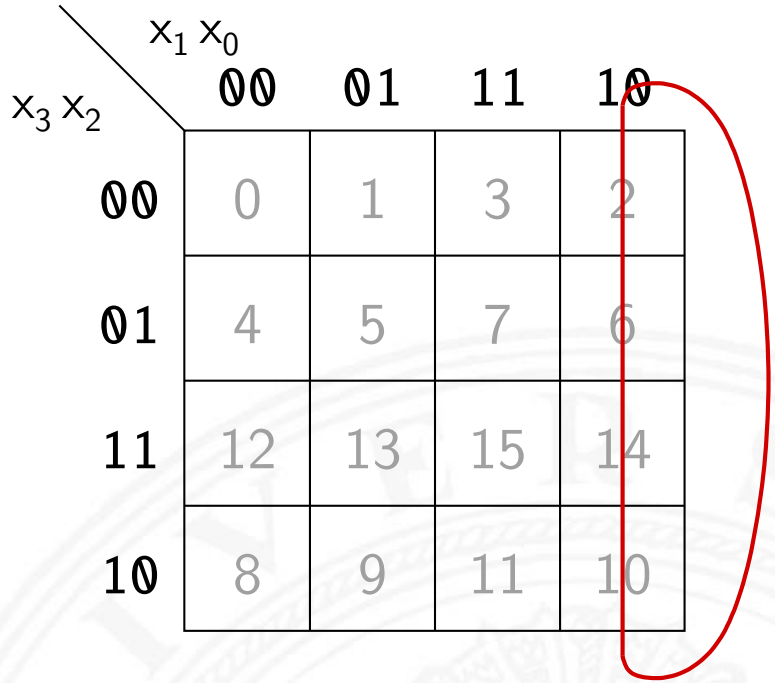
- Pfade

  0,1,3,2,6,7,5,13,15,14,10,11,9,8,12,4 1,3,7,6,14,15,11,9,13,12,4,5

- jeder Pfad entspricht einem einschrittigen Code
- geschlossener Pfad: zyklisch einschrittiger Code

Einschrittiger Code: KV-Diagramm (cont.)





- Pfade

  4,5,13,15,7,6,14,10,8,12 2,6,14,10

- linke und rechte Spalte unterscheiden sich um 1 Bit
  obere und untere Zeile unterscheiden sich um 1 Bit

$\Rightarrow$ KV-Diagramm als „außen zusammengeklebt“ denken

$\Rightarrow$ Pfade können auch „außen herum“ geführt werden

Gray-Code: Umwandlung in/von Dualcode

Umwandlung: Dual- in Graywort

1. MSB des Dualworts wird MSB des Grayworts
2. von links nach rechts: bei jedem Koeffizientenwechsel im Dualwort wird das entsprechende Bit im Graywort 1, sonst 0

- Beispiele 0011 $\rightarrow$ 0010, 1110 $\rightarrow$ 1001, 0110 $\rightarrow$ 0101 usw.

- `gray(x) = x ^ (x >>> 1)`

- in Hardware einfach durch paarweise XOR-Operationen

  [HenHA] Hades Demo: `10-gates/15-graycode/dual2gray`

Gray-Code: Umwandlung in/von Dualcode (cont.)

Umwandlung: Gray- in Dualwort

1. MSB wird übernommen
2. von links nach rechts: wenn das Graywort eine Eins aufweist, wird das vorhergehende Bit des Dualworts invertiert in die entsprechende Stelle geschrieben, sonst wird das Zeichen der vorhergehenden Stelle direkt übernommen

- Beispiele $0010 \rightarrow 0011$, $1001 \rightarrow 1110$, $0101 \rightarrow 0110$ usw.

- in Hardware einfach durch Kette von XOR-Operationen

Optimalcodes: Codes variabler Länge

- Einsatz zur Quellencodierung
- Minimierung der Datenmenge durch Anpassung an die Symbolhäufigkeiten
- häufige Symbole bekommen kurze Codewörter,
  seltene Symbole längere Codewörter

- anders als bei Blockcodes ist die Trennung zwischen Codewörtern nicht durch Abzählen möglich

$\Rightarrow$ Einhalten der **Fano-Bedingung** notwendig

oder Einführen von **Markern** zwischen den Codewörtern

Fano-Bedingung

Eindeutige Decodierung eines Codes mit variabler Wortlänge?

Fano-Bedingung

Kein Wort aus einem Code bildet den Anfang eines anderen Codeworts

- die sogenannte **Präfix-Eigenschaft**
- nach R. M. Fano (1961)

- ein **Präfix-Code** ist eindeutig decodierbar
- Blockcodes sind Präfix-Codes

Fano-Bedingung: Beispiele

- Telefonnummern: das Vorwahlsystem gewährleistet die Fano-Bedingung

110, 112	: Notrufnummern
42883 2502	: Ortsnetz (keine führende Null)
040 42883 2502	: nationales Netz
0049 40 42883 2502	: internationale Rufnummer

- Morse-Code: Fano-Bedingung verletzt

Morse-Code

Codetabelle

● kurzer Ton — langer Ton

A	●—	S	●●●	.	●—●—●—	S-Start	—●—●—
B	—●●●	T	—	,	——●●——	Verst.	●●●—●
C	—●—●	U	●●—	?	●●——●●	S-Ende	●—●—●
D	—●●	V	●●●—	´	●—————●	V-Ende	●●●—●—
E	●	W	●——	!	—●—●——	Error	●●●●●●●●
F	●●—●	X	—●●—	/	—●●—●		
G	——●	Y	—●——	(	—●——●	Ä	●—●—
H	●●●●	Z	——●●	)	—●——●—	À	●——●—
I	●●	0	—————	&	●—●●●	É	●●—●●
J	●———	1	●————	:	———●●●	È	●—●●—
K	—●—	2	●●———	;	—●—●—●	Ö	———●
L	●—●●	3	●●●——	=	—●●●—	Ü	●●——
M	——	4	●●●●—	+	●—●—●	ß	●●●——●●
N	—●	5	●●●●●	-	—●●●●—	CH	————
O	———	6	—●●●●	_	●●——●—	Ñ	——●——
P	●——●	7	——●●●	"	●—●●—●	...	
Q	——●—	8	———●●	$	●●●—●●—		
R	●—●	9	————●	@	●——●—●	SOS	●●● ——— ●●●

Morse-Code (cont.)

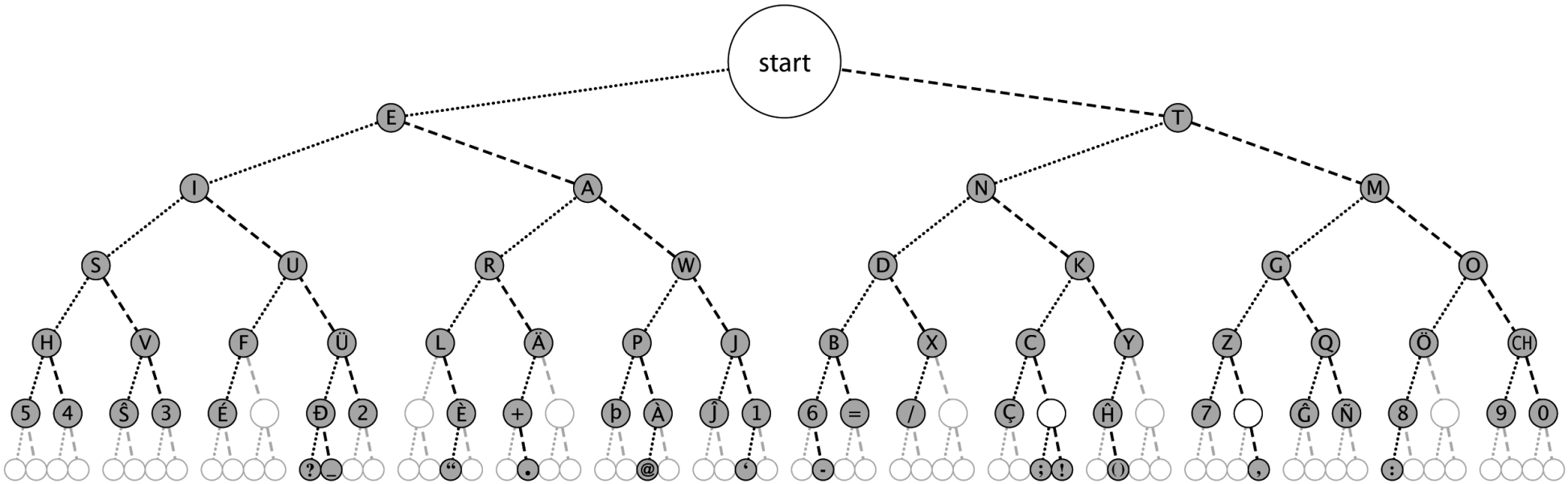
- Eindeutigkeit

Codewort:	● ● ● ● ● – ●
E	●
I	● ●
N	– ●
R	● – ●
S	● ● ●

  - bestimmte Morse-Sequenzen sind mehrdeutig
  - Pause zwischen den Symbolen notwendig

- Codierung
  - Häufigkeit der Buchstaben = 1 / Länge des Codewortes
  - Effizienz: kürzere Codeworte
  - Darstellung als Codebaum

Morse-Code: Codebaum (Ausschnitt)



- Symbole als Knoten oder Blätter
- Knoten: Fano-Bedingung verletzt
- Codewort am Pfad von Wurzel zum Knoten/Blatt ablesen

Morse-Code: Umschlüsselung

Umschlüsselung des Codes für binäre Nachrichtenübertragung

- 110 als Umschlüsselung des langen Tons –
  10 als Umschlüsselung des kurzen Tons •
  0 als Trennzeichen zwischen Morse-Codewörtern

- der neue Code erfüllt die Fano-Bedingung

  jetzt eindeutig decodierbar: 101010011011011001010100 (SOS)

- viele andere Umschlüsselungen möglich, z.B.:
  1 als Umschlüsselung des langen Tons –
  01 als Umschlüsselung des kurzen Tons •
  00 als Trennzeichen zwischen Morse-Codewörtern

Codierung nach Fano (Shannon-Fano Codierung)

Gegeben: die zu codierenden Urwörter a_i
und die zugehörigen Wahrscheinlichkeiten $p(a_i)$

- Ordnung der Urwörter anhand ihrer Wahrscheinlichkeiten $p(a_1) \geq p(a_2) \geq \ldots \geq p(a_n)$
- Einteilung der geordneten Urwörter in zwei Gruppen mit möglichst gleicher Gesamtwahrscheinlichkeit: $a_1 \ldots a_i$ und $a_{i+1} \ldots a_n$. Eine Gruppe bekommt als erste Codewortstelle eine 0, die andere eine 1
- Diese Teilgruppen werden erneut geteilt und den Hälften wieder eine 0, bzw. eine 1, als nächste Codewortstelle zugeordnet
- Das Verfahren wird wiederholt, bis jede Teilgruppe nur noch ein Element enthält

- bessere Codierung, je größer die Anzahl der Urwörter
- nicht eindeutig

Codierung nach Fano: Beispiel

Urbildmenge $\{A, B, C, D\}$ und zugehörige Wahrscheinlichkeiten $\{0.45, 0.1, 0.15, 0.3\}$

0. Sortierung nach Wahrscheinlichkeiten ergibt $\{A, D, C, B\}$
1. Gruppenaufteilung ergibt $\{A\}$ und $\{D, C, B\}$
   Codierung von A mit 0 und den anderen Symbolen als $1*$
2. weitere Teilung ergibt $\{D\}$ und $\{C, B\}$
3. letzte Teilung ergibt $\{C\}$ und $\{B\}$

$\Rightarrow$ Codewörter sind $A = 0$, $D = 10$, $C = 110$ und $B = 111$

mittlere Codewortlänge L

- $L = 0.45 \cdot 1 + 0.3 \cdot 2 + 0.15 \cdot 3 + 0.1 \cdot 3 = 1.8$
- zum Vergleich: Blockcode mit 2 Bits benötigt $L = 2$

Codierung nach Fano: Deutsche Großbuchstaben

Buchstabe a_i	Wahrscheinlichkeit $p(a_i)$	Code (Fano)	Bits
Leerzeichen	0.15149	000	3
E	0.14700	001	3
N	0.08835	010	3
R	0.06858	0110	4
I	0.06377	0111	4
S	0.05388	1000	4
...	...	...	...
Ö	0.00255	111111110	9
J	0.00165	1111111110	10
Y	0.00017	11111111110	11
Q	0.00015	111111111110	12
X	0.00013	111111111111	12

Ameling: *Fano-Code der Buchstaben der deutschen Sprache*, 1992

Codierung nach Huffman

Gegeben: die zu codierenden Urwörter a_i
und die zugehörigen Wahrscheinlichkeiten $p(a_i)$

- Ordnung der Urwörter anhand ihrer Wahrscheinlichkeiten $p(a_1) \leq p(a_2) \leq \ldots \leq p(a_n)$
- in jedem Schritt werden die zwei Wörter mit der geringsten Wahrscheinlichkeit zusammengefasst und durch ein neues ersetzt
- das Verfahren wird wiederholt, bis eine Menge mit nur noch zwei Wörtern resultiert
- rekursive Codierung als Baum (z.B.: links 0, rechts 1)
- ergibt die kleinstmöglichen mittleren Codewortlängen
- Abweichungen zum Verfahren nach Fano sind aber gering
- vielfältiger Einsatz (u.a. bei JPEG, MPEG ...)

Codierung nach Huffman: Beispiel

Urbildmenge $\{A, B, C, D\}$ und zugehörige Wahrscheinlichkeiten $\{0.45, 0.1, 0.15, 0.3\}$

0. Sortierung nach Wahrscheinlichkeiten ergibt $\{B, C, D, A\}$
1. Zusammenfassen von B und C als neues Wort E, Wahrscheinlichkeit von E ist dann $p(E) = 0.1 + 0.15 = 0.25$
2. Zusammenfassen von E und D als neues Wort F mit $p(F) = 0.55$
3. Zuordnung der Bits entsprechend der Wahrscheinlichkeiten
   - $F = 0$ und $A = 1$
   - Split von F in $D = 00$ und $E = 01$
   - Split von E in $C = 010$ und $B = 011$

$\Rightarrow$ Codewörter sind $A = 1$, $D = 00$, $C = 010$ und $B = 011$

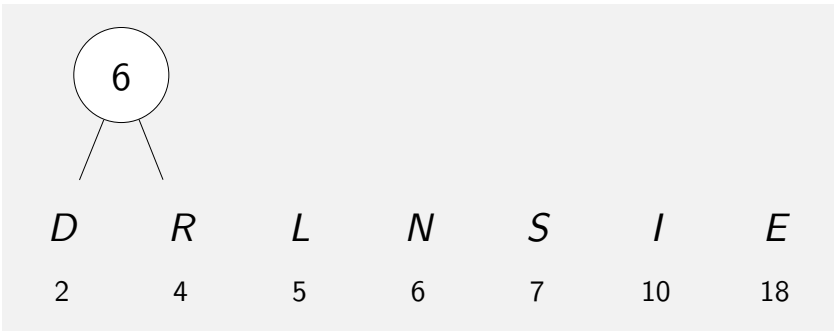
Bildung eines Huffman-Baums

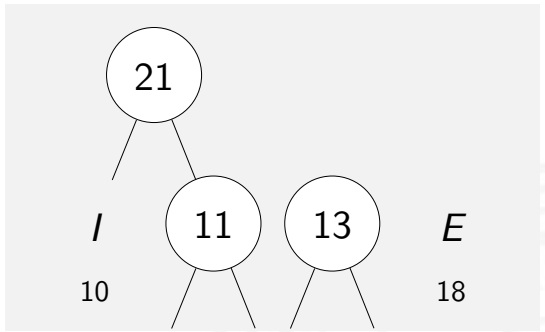
- Alphabet $= \{E, I, N, S, D, L, R\}$
- relative Häufigkeiten
  $E = 18, I = 10, N = 6, S = 7, D = 2, L = 5, R = 4$

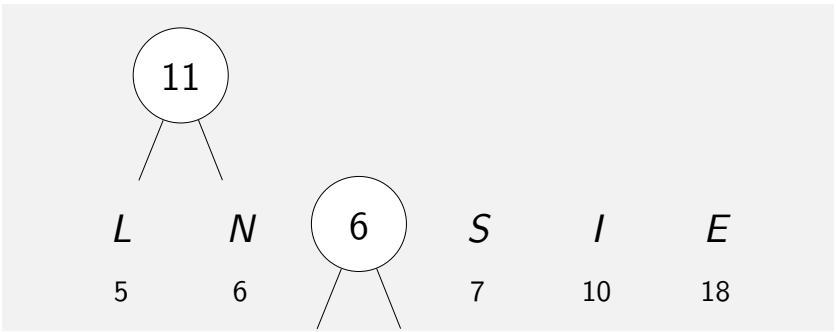
- Sortieren anhand der Häufigkeiten
- Gruppierung (rekursiv)
- Aufbau des Codebaums
- Ablesen der Codebits

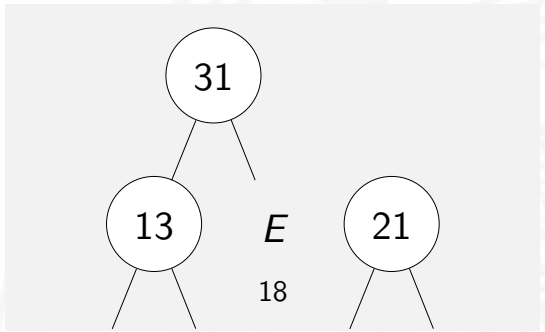
Bildung eines Huffman-Baums (cont.)

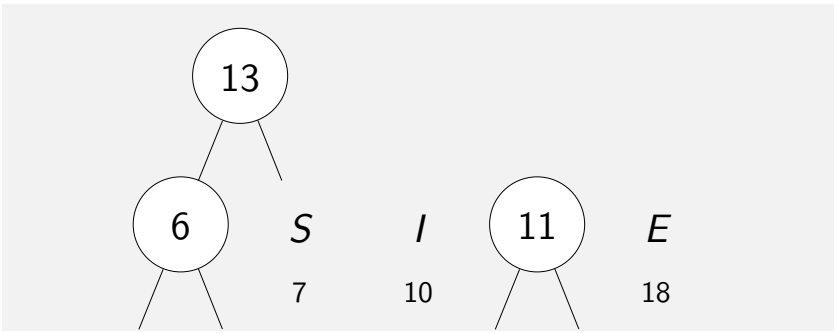
D	*R*	*L*	*N*	*S*	*I*	*E*
2	4	5	6	7	10	18

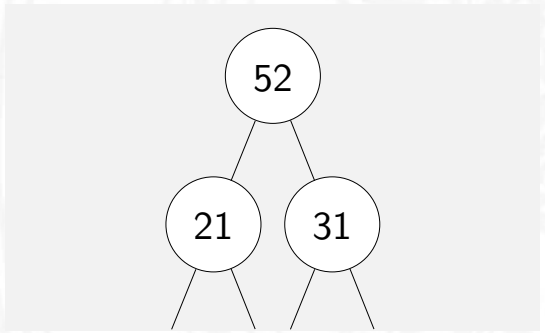




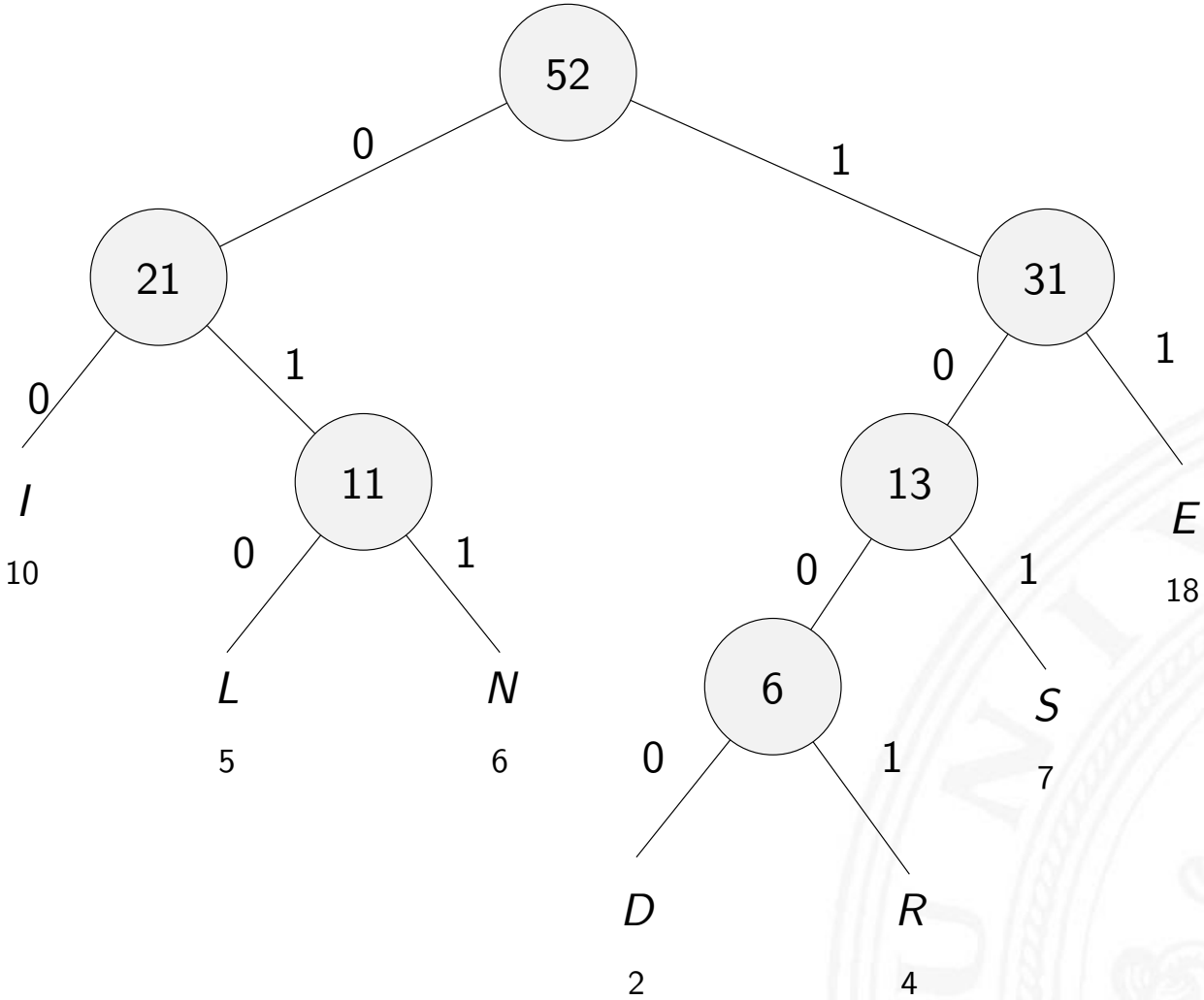








Bildung eines Huffman-Baums (cont.)



I	00
L	010
N	011
D	1000
R	1001
S	101
E	11

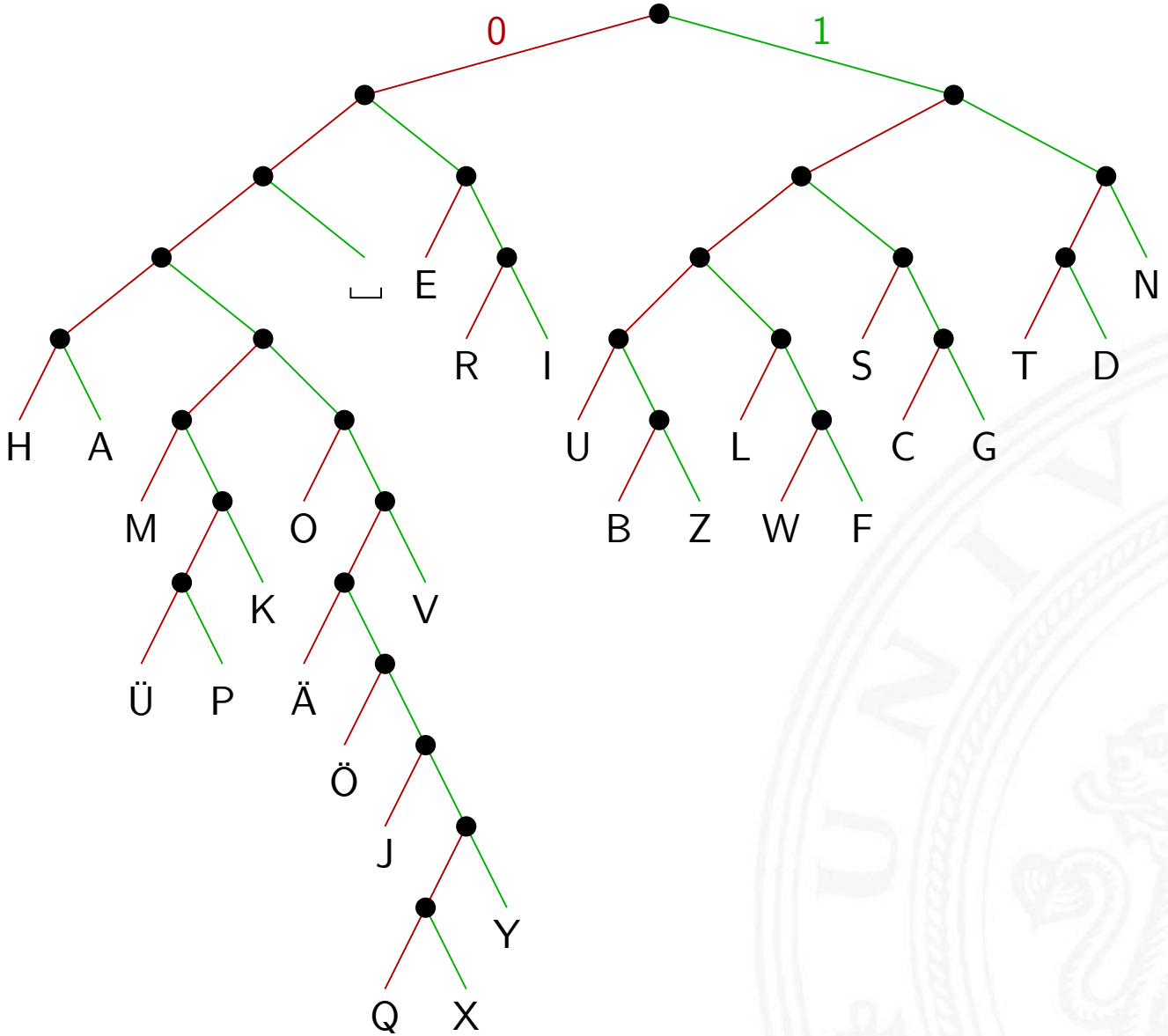
1001 00 11 101 11

R *I* *E* *S* *E*

Codierung nach Huffman: Deutsche Großbuchstaben

Zeichen	Code	Zeichen	Code
Leerzeichen	001	O	000110
E	010	B	100010
N	111	Z	100011
R	0110	W	100110
I	0111	F	100111
S	1010	K	0001011
T	1100	V	0001111
D	1101	Ü	00010100
H	00000	P	00010101
A	00001	Ä	00011100
U	10000	Ö	000111010
L	10010	J	0001110110
C	10110	Y	00011101111
G	10111	Q	000111011100
M	000100	X	000111011101

Codierung nach Huffman: Codebaum



ca. 4.5 Bits/Zeichen,
1.7-Mal besser als ASCII

Codierung nach Huffman: Minimale Codelänge

Beweis der Minimalität

- Sei C ein Huffman-Code mit durchschnittlicher Codelänge L
- Sei D ein weiterer Präfix-Code mit durchschnittlicher Codelänge M, mit $M < L$ und M minimal
- Berechne die C und D zugeordneten Decodierbäume A und B
- Betrachte die beiden Endknoten für Symbole kleinster Wahrscheinlichkeit:
  - Weise dem Vorgängerknoten das Gewicht $p_{s-1} + p_s$ zu
  - streiche die Endknoten
  - mittlere Codelänge reduziert sich um $p_{s-1} + p_s$
- Fortsetzung führt dazu, dass Baum C sich auf Baum mit durchschnittlicher Länge 1 reduziert und D auf Länge < 1. Dies ist aber nicht möglich $\square$

Codierung nach Huffman: Symbole mit $p \geq 0.5$

Was passiert, wenn ein Symbol eine Häufigkeit $p_0 \geq 0.5$ aufweist?

- die Huffman-Codierung müsste weniger als ein Bit zuordnen, dies ist jedoch nicht möglich

$\Rightarrow$ Huffman- (und Fano-) Codierung ist in diesem Fall ineffizient

- Beispiel: Bild mit einheitlicher Hintergrundfarbe codieren
- andere Ideen notwendig
  - Lauflängencodierung (Fax, GIF, PNG)
  - Cosinustransformation (JPEG) usw.

Dynamic Huffman Coding

was tun, wenn

- die Symbolhäufigkeiten nicht vorab bekannt sind?
- die Symbolhäufigkeiten sich ändern können?

Dynamic Huffman Coding (Knuth 1985)

- Encoder protokolliert die (bisherigen) Symbolhäufigkeiten
- Codebaum wird dynamisch aufgebaut und ggf. umgebaut

- Decoder arbeitet entsprechend:
  Codebaum wird mit jedem decodierten Zeichen angepasst
- Symbolhäufigkeiten werden nicht explizit übertragen

D. E. Knuth: *Dynamic Huffman Coding*, 1985 [Knu85]

Kraft-Ungleichung

- Leon G. Kraft, 1949

  `https://de.wikipedia.org/wiki/Kraft-Ungleichung`

- Eine notwendige und hinreichende Bedingung für die Existenz eines eindeutig decodierbaren s-elementigen Codes C mit Codelängen $l_1 \leq l_2 \leq l_3 \leq \ldots \leq l_s$ über einem q-nären Zeichenvorrat F ist:

$$\sum_{i=1}^{s} \frac{1}{q^{l_i}} \leq 1$$

- Beispiel

  $\{1, 00, 01, 11\}$ ist nicht eindeutig decodierbar,
  denn $\frac{1}{2} + 3 \cdot \frac{1}{4} = 1.25 > 1$

Kraft-Ungleichung: Beispiel

- Sei $F = \{0, 1, 2\}$ (ternäres Alphabet)
- Seien die geforderten Längen der Codewörter: 1,2,2,2,2,2,3,3,3

- Einsetzen in die Ungleichung: $\frac{1}{3} + 5 \cdot \frac{1}{3^2} + 3 \cdot \frac{1}{3^3} = 1$

$\Rightarrow$ Also existiert ein passender Präfixcode.

- Konstruktion entsprechend des Beweises

  0 10 11 12 20 21 220 221 222

Kraft-Ungleichung: Beweisidee

Sei $l_s = m$ und seien u_i die Zahl der Codewörter der Länge i

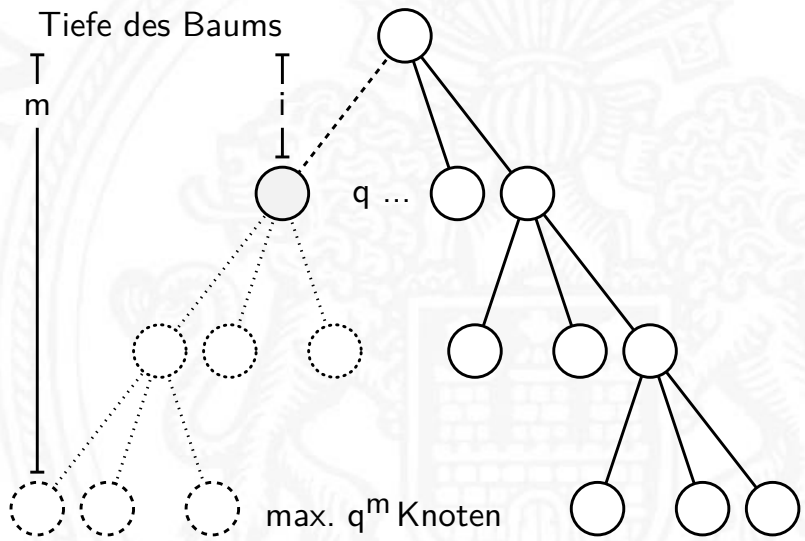
- Wir schreiben

$$\sum_{i=1}^{s} \frac{1}{q^{l_i}} = \sum_{j=1}^{m} \frac{u_j}{q^j} = \frac{1}{q^m} \sum_{j=1}^{m} u_j \cdot q^{m-j} \leq 1$$

$$u_m + \sum_{j=1}^{m-1} u_j \cdot q^{m-j} \leq q^m \qquad (*)$$

- Jedes Codewort der Länge i „verbraucht“ q^{m-i} Wörter aus F^m
- Summe auf der linken Seite von $(*)$ ist die Zahl der durch den Code C benutzten Wörter von F^m

$\Rightarrow$ erfüllt C die Präfix-Bedingung, dann gilt $(*)$



Informationsbegriff

- n mögliche sich gegenseitig ausschließende Ereignisse A_i
  die zufällig nacheinander mit Wahrscheinlichkeiten p_i eintreten
- stochastisches Modell $W\{A_i\} = p_i$

- angewendet auf Informationsübertragung:
  das Symbol a_i wird mit Wahrscheinlichkeit p_i empfangen

- Beispiel
  - $p_i = 1$ und $p_j = 0 \quad \forall j \neq i$
  - dann wird mit Sicherheit das Symbol A_i empfangen
  - der Empfang bringt keinen Informationsgewinn

$\Rightarrow$ Informationsgewinn („Überraschung“) wird größer, je kleiner p_i

Geeignetes Maß für die Information?

- Wir erhalten die Nachricht A mit der Wahrscheinlichkeit p_A und anschließend die unabhängige Nachricht B mit der Wahrscheinlichkeit p_B
- Wegen der Unabhängigkeit ist die Wahrscheinlichkeit beider Ereignisse gegeben durch das Produkt $p_A \cdot p_B$
- Informationsgewinn („Überraschung“) größer, je kleiner p_i
- Wahl von $1/p$ als Maß für den Informationsgewinn?
- möglich, aber der Gesamtinformationsgehalt zweier (mehrerer) Ereignisse wäre das Produkt der einzelnen Informationsgehalte
- additive Größe wäre besser $\Rightarrow$ Logarithmus von $1/p$ bilden

Erinnerung: Logarithmus

- Umkehrfunktion zur Exponentialfunktion
- formal: für gegebenes a und b ist der Logarithmus die Lösung der Gleichung $a = b^x$
- falls die Lösung existiert, gilt: $x = \log_b(a)$

- Beispiel $3 = \log_2(8)$, denn $2^3 = 8$

- Rechenregeln
  - $log(x \cdot y) = log(x) + log(y)$ (Addition statt Multiplikation)
  - $b^{\log_b(x)} = x$ und $\log_b(b^x) = x$
  - $\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$
  - $\log_2(x) = \ln(x)/\ln(2) = \ln(x)/0,693141718$

Definition: Informationsgehalt

Informationsgehalt eines Ereignisses A_i mit Wahrscheinlichkeit p_i?

- als messbare und daher additive Größe
- durch Logarithmierung (Basis 2) der Wahrscheinlichkeit:

$$I(A_i) = \log_2(\frac{1}{p_i}) = -\log_2(p_i)$$

- **Informationsgehalt** I (oder Information) von A_i
  auch **Entscheidungsgehalt** genannt

- Beispiel: zwei Nachrichten A und B

$$I(A) + I(B) = \log_2(\frac{1}{p_A \cdot p_B}) = \log_2(\frac{1}{p_A}) + \log_2(\frac{1}{p_B})$$

Einheit Bit

$$I(A_i) = \log_2(\frac{1}{p_i}) = -\log_2(p_i)$$

- Wert von I ist eine reelle Größe
- gemessen in der Einheit **1 Bit**

- Beispiel: nur zwei mögliche Symbole 0 und 1 mit gleichen Wahrscheinlichkeiten $p_0 = p_1 = \frac{1}{2}$

  Der Informationsgehalt des Empfangs einer 0 oder 1 ist dann $I(0) = I(1) = \log_2(1/\frac{1}{2}) = 1\,\text{Bit}$

„Bit“ Verwechslungsgefahr

Bit:	als Maß für den Informationsgehalt	Maßeinheit
bit:	Anzahl der Binärstellen	–"–
Bit:	Binärzeichen, Symbol 0 oder 1	(Kap. „5 Zeichen und Text“)

Ungewissheit, Überraschung, Information

- Vor dem Empfang einer Nachricht gibt es Ungewissheit über das Kommende

  Beim Empfang gibt es die Überraschung

  Und danach hat man den Gewinn an Information

- Alle drei Begriffe in der oben definierten Einheit **Bit** messen
- Diese Quantifizierung der Information ist zugeschnitten auf die Nachrichtentechnik

- umfasst nur einen Aspekt des umgangssprachlichen Begriffs Information

Informationsgehalt: Beispiele

Meteorit

- die Wahrscheinlichkeit, an einem Tag von einem Meteor getroffen zu werden, sei $p_M = 10^{-16}$
- Kein Grund zur Sorge, weil die Ungewissheit von $I = \log_2(1/(1-p_M)) \approx 3,2 \cdot 10^{-16}$ sehr klein ist

  Ebenso klein ist die Überraschung, wenn das Unglück nicht passiert $\Rightarrow$ Informationsgehalt der Nachricht „Ich wurde nicht vom Meteor erschlagen“ ist sehr klein
- Umgekehrt wäre die Überraschung groß: $\log_2(1/p_M) = 53,15$

Informationsgehalt: Beispiele (cont.)

Würfeln

- bei vielen Spielen hat die 6 eine besondere Bedeutung
- hier betrachten wir aber zunächst nur die Wahrscheinlichkeit von Ereignissen, nicht deren Semantik

- die Wahrscheinlichkeit, eine 6 zu würfeln, ist $\frac{1}{6}$
- $I(6) = \log_2(1/\frac{1}{6}) = 2,585$

Informationsgehalt: Beispiele (cont.)

Information eines Buchs

- Gegeben seien zwei Bücher
  1. deutscher Text
  2. mit Zufallsgenerator mit Gleichverteilung aus Alphabet mit 80-Zeichen erzeugt
- Informationsgehalt in beiden Fällen?
  1. Im deutschen Text abhängig vom Kontext!
     Beispiel: Empfangen wir als deutschen Text „*Der Begrif*“, so ist „*f*“ als nächstes Symbol sehr wahrscheinlich
  2. beim Zufallstext liefert jedes neue Symbol die zusätzliche Information $I = \log_2(1/\frac{1}{80})$

$\Rightarrow$ der Zufallstext enthält die größtmögliche Information

Informationsgehalt: Beispiele (cont.)

Einzelner Buchstabe

- die Wahrscheinlichkeit, in einem Text an einer gegebenen Stelle das Zeichen „A“ anzutreffen sei $W\{A\} = p = 0,01$

- Informationsgehalt $I(A) = \log_2(1/0,01) = 6,6439$
- wenn der Text in ISO-8859-1 codiert vorliegt, werden 8 Binärstellen zur Repräsentation des „A“ benutzt
- der Informationsgehalt ist jedoch geringer

Entropie

Obige Definition der Information lässt sich nur jeweils auf den Empfang eines speziellen Zeichens anwenden

- Was ist die **durchschnittliche Information** bei Empfang eines Symbols?
- diesen Erwartungswert bezeichnet man als **Entropie** des Systems (auch **mittlerer Informationsgehalt**)

- Wahrscheinlichkeiten aller möglichen Ereignisse A_i seien $W\{A_i\} = p_i$
- da jeweils eines der möglichen Symbole eintrifft, gilt $\sum_i p_i = 1$

Entropie (cont.)

- dann berechnet sich die Entropie H als Erwartungswert

$$
\begin{aligned}
H &= E\{I(A_i)\} \\
&= \sum_i p_i \cdot I(A_i) \\
&= \sum_i p_i \cdot \log_2(\frac{1}{p_i}) \\
&= -\sum_i p_i \cdot \log_2(p_i)
\end{aligned}
$$

- als Funktion der Symbol-Wahrscheinlichkeiten nur abhängig vom stochastischen Modell

Entropie: Beispiele

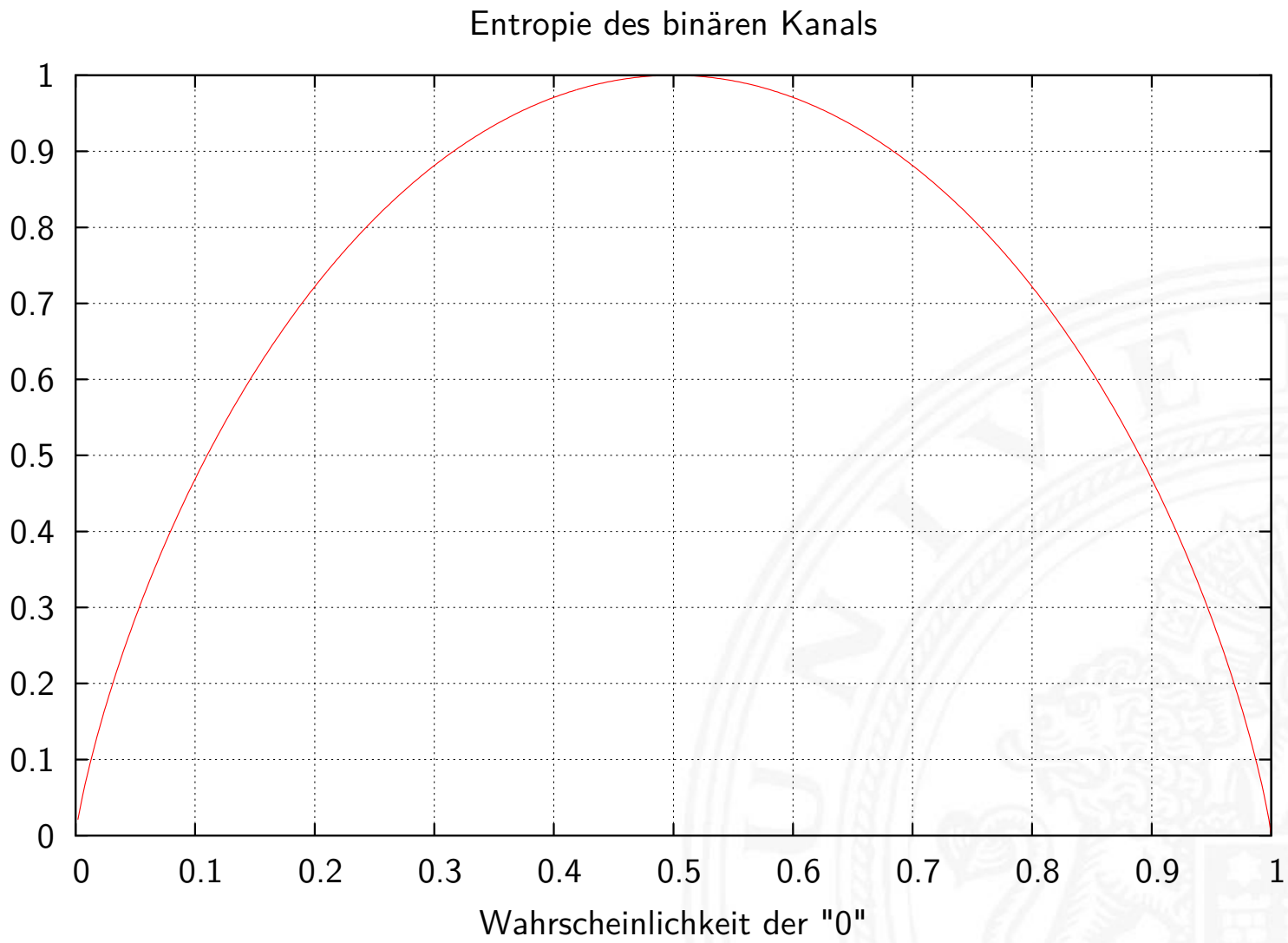
1. drei mögliche Ereignisse mit Wahrscheinlichkeiten $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\}$

- dann berechnet sich die Entropie zu
  $H = -(\frac{1}{2}\log_2(\frac{1}{2}) + \frac{1}{3}\log_2(\frac{1}{3}) + \frac{1}{6}\log_2(\frac{1}{6})) = 1,4591$

2. Empfang einer Binärstelle mit den Wahrscheinlichkeiten
   $p_0 = q$ und $p_1 = (1 - q)$.

- für $q = \frac{1}{2}$ erhält man
  $H = -(\frac{1}{2}\log_2(\frac{1}{2}) + (1 - \frac{1}{2})\log_2(1 - \frac{1}{2})) = 1.0$
- mittlerer Informationsgehalt beim Empfang einer Binärstelle mit gleicher Wahrscheinlichkeit für beide Symbole ist genau 1 Bit

Entropie: Diagramm



Entropie bei Empfang einer Binärstelle mit den Wahrscheinlichkeiten $p_0 = q$ und $p_1 = (1 - q)$

Entropie: Symbolverteilung

- mittlerer Informationsgehalt einer Binärstelle nur dann 1 Bit, wenn beide möglichen Symbole gleich wahrscheinlich

- entsprechendes gilt auch für größere Symbolmengen
- Beispiel: 256 Symbole (8-bit), gleich wahrscheinlich

  $H = \sum_i p_i \log_2(1/p_i) = 256 \cdot \frac{1}{256} \cdot \log_2(1/\frac{1}{256}) = 8\,\text{Bit}$

Entropie: einige Eigenschaften

1. $H(p_1, p_2, \ldots, p_n)$ ist maximal, falls $p_i = 1/n \quad (1 \leq i \leq n)$
2. H ist symmetrisch, für jede Permutation π von $1, 2, \ldots, n$ gilt:
   $H(p_1, p_2, \ldots, p_n) = H(p_{\pi(1)}, p_{\pi(2)}, \ldots, p_{\pi(n)})$
3. $H(p_1, p_2, \ldots, p_n) \geq 0 \quad$ mit $H(0, 0 \ldots 0, 1, 0 \ldots 0, 0) = 0$
4. $H(p_1, p_2, \ldots, p_n, 0) = H(p_1, p_2, \ldots, p_n)$
5. $H(1/n, 1/n, \ldots, 1/n) \leq H(1/(n+1), 1/(n+1), \ldots, 1/(n+1))$
6. H ist stetig in seinen Argumenten
7. Additivität: seien $n, m \in N^+$
   $H(\frac{1}{n \cdot m}, \frac{1}{n \cdot m}, \ldots, \frac{1}{n \cdot m}) = H(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}) + H(\frac{1}{m}, \frac{1}{m}, \ldots, \frac{1}{m})$

Möglicher Informationsgehalt

- **möglicher Informationsgehalt** H_0 ist durch Symbolcodierung festgelegt (entspricht **mittlerer Codewortlänge** $\bar{l}$)

$$H_0 = \sum_i p_i \cdot \log_2(q^{l_i})$$

  - stochastisches Modell $W\{A_i\} = p_i$ (Wahrscheinlichkeiten von Ereignissen A_i)
  - Codierung der Ereignisse (der Symbole) $C(A_i)$ durch Code der Länge l_i über einem q-nären Alphabet

- für Binärcodes gilt $H_0 = \sum_i p_i \cdot l_i$

- binäre Blockcodes mit Wortlänge N bits: $H_0 = N$

Redundanz

- **Redundanz** (engl. *code redundancy*):
  die Differenz zwischen dem möglichen und dem tatsächlich genutzten Informationsgehalt $R = H_0 - H$
  - möglicher Informationsgehalt H_0 ist durch Symbolcodierung festgelegt = mittlere Codewortlänge
  - tatsächliche Informationsgehalt ist die Entropie H

- **relative Redundanz**: $r = \frac{H_0 - H}{H_0}$

- binäre Blockcodes mit Wortlänge N bits: $H_0 = N$
  gegebener Code mit m Wörtern a_i und $p(a_i)$:

$$R = H_0 - H = H_0 - \left(- \sum_{i=1}^{m} p(a_i) \cdot \log_2(p(a_i)) \right)$$

$$= N + \sum_{i=1}^{m} p(a_i) \cdot \log_2(p(a_i))$$

Kanalkapazität

Informationstheorie ursprünglich entwickelt zur

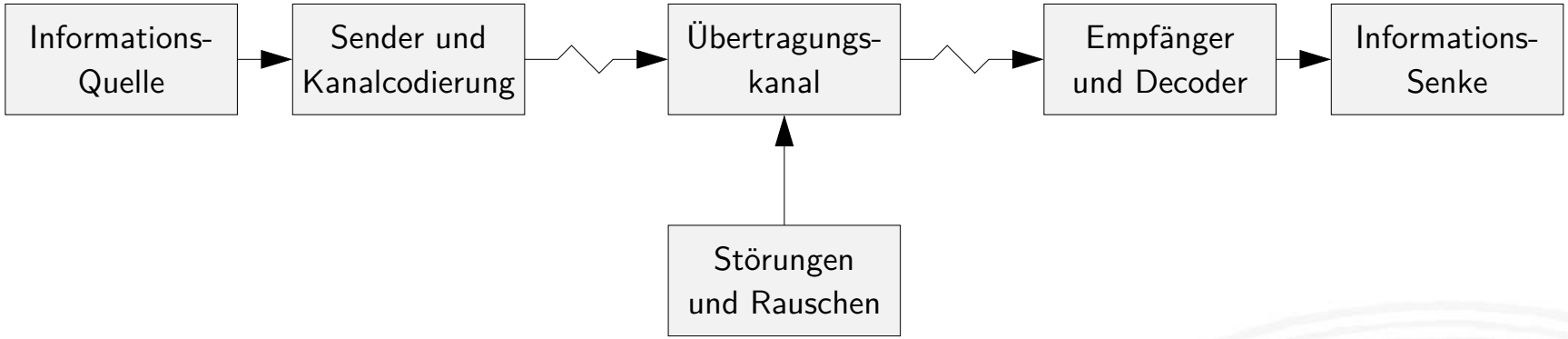
- formalen Behandlung der Übertragung von Information
- über reale, nicht fehlerfreie Kanäle
- deren Verhalten als stochastisches Modell formuliert werden kann

- zentrales Resultat ist die **Kanalkapazität** C des **binären symmetrischen Kanals**
- der maximal pro Binärstelle übertragbare Informationsgehalt

$$C = 1 - H(F)$$

mit $H(F)$ der Entropie des Fehlerverhaltens

Erinnerung: Modell der Informationsübertragung



- Informationsquelle
- Sender mit möglichst effizienter Kanalcodierung
- gestörter und verrauschter Übertragungskanal
- Empfänger mit Decodierer und Fehlererkennung/-korrektur
- Informationssenke und -verarbeitung

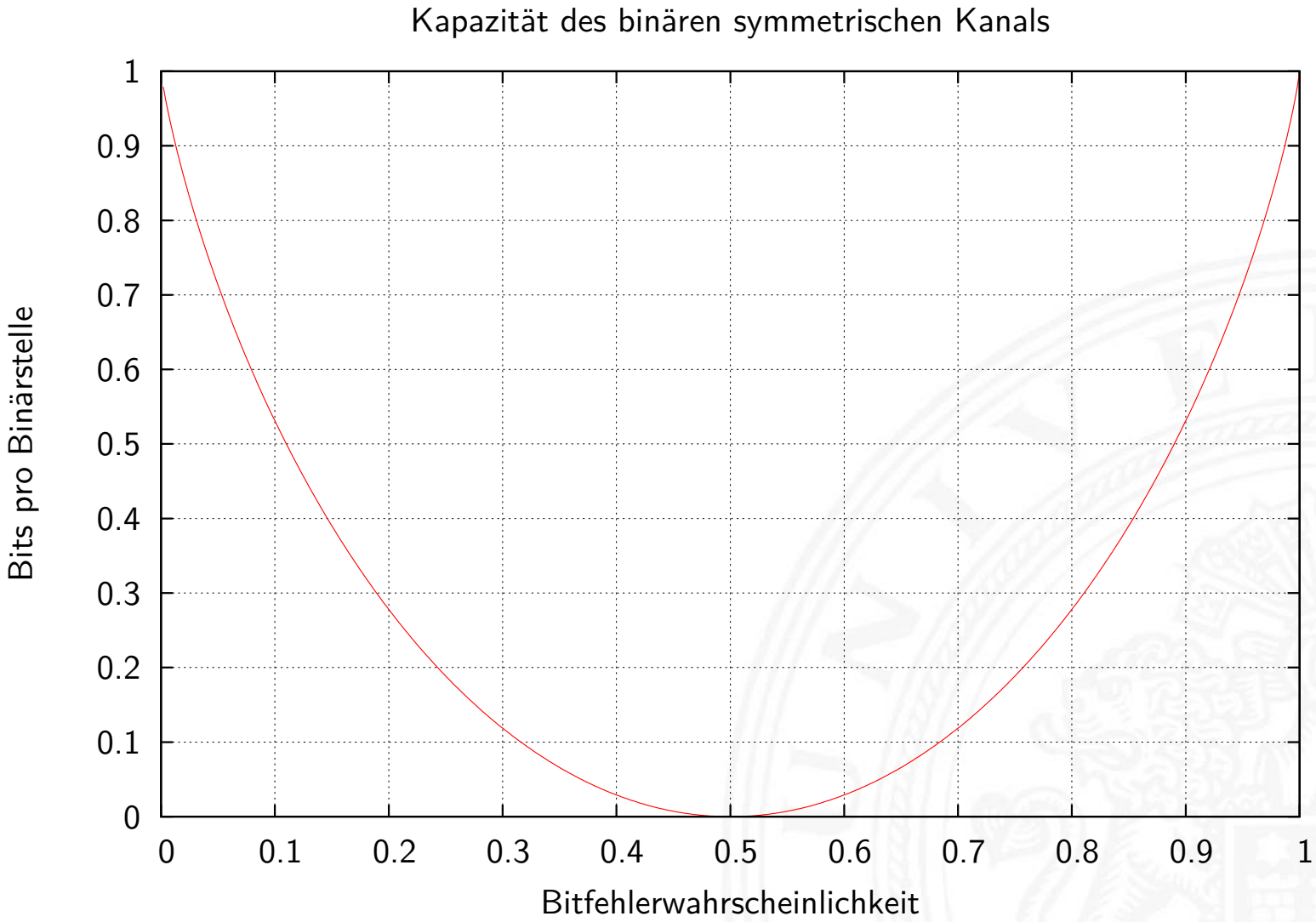
Binärer symmetrischer Kanal

- Wahrscheinlichkeit der beiden Symbole 0 und 1 ist gleich $(\frac{1}{2})$
- Wahrscheinlichkeit P, dass bei Übertragungsfehlern aus einer 0 eine 1 wird = Wahrscheinlichkeit, dass aus einer 1 eine 0 wird
- Wahrscheinlichkeit eines Fehlers an Binärstelle i ist unabhängig vom Auftreten eines Fehlers an anderen Stellen
- Entropie des Fehlerverhaltens

$$H(F) = P \cdot \log_2(1/P) + (1-P) \cdot \log_2(1/(1-P))$$

- Kanalkapazität ist $C = 1 - H(F)$

Kanalkapazität: Diagramm



Kanalkapazität: Konsequenzen

- bei $P = 0,5$ ist die Kanalkapazität $C = 0$

$\Rightarrow$ der Empfänger kann die empfangenen Daten nicht von einer zufälligen Sequenz unterscheiden

- bei $P > 0,5$ steigt die Kapazität wieder an

  (rein akademischer Fall: Invertieren aller Bits)

Die Kanalkapazität ist eine obere Schranke

- wird in der Praxis nicht erreicht (Fehler)
- Theorie liefert keine Hinweise, wie die fehlerfreie Übertragung praktisch durchgeführt werden kann

Shannon-Theorem

C. E. Shannon: *Communication in the Presence of Noise*; Proc. IRE, Vol.37, No.1, 1949

Gegeben:
binärer symmetrischer Kanal mit der Störwahrscheinlichkeit P
und der Kapazität $C(P)$

Shannon-Theorem

Falls die Übertragungsrate R kleiner als $C(P)$ ist,
findet man zu jedem $\epsilon > 0$ einen Code $\mathcal{C}$ mit
der Übertragungsrate $R(\mathcal{C})$ und $C(P) \geq R(\mathcal{C}) \geq R$ und
der Fehlerdecodierwahrscheinlichkeit $< \epsilon$

auch: C. E. Shannon: *A Mathematical Theory of Communication* `math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf`

Shannon-Theorem (cont.)

C. E. Shannon: *Communication in the Presence of Noise*; Proc. IRE, Vol.37, No.1, 1949

$\Rightarrow$ Wenn die Übertragungsrate kleiner als die Kanalkapazität ist, existieren Codes, die beliebig zuverlässig sind

$\ldots$ und deren Signalübertragungsraten beliebig nahe der Kanalkapazität liegen

- leider liefert die Theorie keine Ideen zur Realisierung
- die Nachrichten müssen sehr lang sein
- der Code muss im Mittel sehr viele Fehler in jeder Nachricht korrigieren
- mittlerweile sehr nah am Limit: Turbo-Codes, LDPC-Codes usw.

Fehlererkennende / -korrigierende Codes

Motivation

- Informationstheorie
- Kanalkapazität
- Shannon-Theorem

- zuverlässige Datenübertragung ist möglich
- aber (bisher) keine Ideen für die Realisierung

$\Rightarrow$ fehlererkennende Codes

$\Rightarrow$ fehlerkorrigierende Codes

Fehlertypen

diverse mögliche Fehler bei der Datenübertragung

- Verwechslung eines Zeichens $a \rightarrow b$
- Vertauschen benachbarter Zeichen $ab \rightarrow ba$
- Vertauschen entfernter Zeichen $abc \rightarrow cba$
- Zwillings-/Bündelfehler $aa \rightarrow bb$
- usw.

- abhängig von der Technologie / der Art der Übertragung
  - Bündelfehler durch Kratzer auf einer CD
  - Bündelfehler bei Funk durch längere Störimpulse
  - Buchstabendreher beim „Eintippen“ eines Textes

Begriffe zur Fehlerbehandlung

- **Block-Code**: k-Informationsbits werden in n-Bits codiert
- **Faltungscodes**: ein Bitstrom wird in einen Codebitstrom höherer Bitrate codiert
  - Bitstrom erzeugt Folge von Automatenzuständen
  - Decodierung über bedingte Wahrscheinlichkeiten bei Zustandsübergängen
  - im Prinzip linear, Faltungscodes passen aber nicht in Beschreibung unten

- **linearer** (n, k)**-Code**: ein k-dimensionaler Unterraum des $GF(2)^n$
- **modifizierter Code**: eine oder mehrere Stellen eines linearen Codes werden systematisch verändert (d.h. im $GF(2)$ invertiert) Null- und Einsvektor gehören nicht mehr zum Code
- **nichtlinearer Code**: weder linear noch modifiziert

Einschub: $GF(2)$, $GF(2)^n$

de.wikipedia.org/wiki/Endlicher_Körper

en.wikipedia.org/wiki/GF(2)

Boole'sche Algebra

Details: Mathe-Skript, Wikipedia, v.d. Heide [Hei05]

- basiert auf: UND, ODER, Negation
- UND $\approx$ Multiplikation
  ODER $\approx$ Addition
- aber: kein inverses Element für die ODER-Operation
  $\Rightarrow$ kein Körper

Galois-Feld mit zwei Elementen: GF(2)

- Körper, zwei Verknüpfungen: UND und XOR
- UND als Multiplikation
  XOR als Addition *mod* 2
- additives Inverses existiert: $x \oplus x = 0$

Begriffe zur Fehlerbehandlung (cont.)

- **systematischer Code**: wenn die zu codierende Information direkt (als Substring) im Codewort enthalten ist

- **zyklischer Code**
  - ein Block-Code (identische Wortlänge aller Codewörter)
  - für jedes Codewort gilt: auch alle zyklischen Verschiebungen (Rotationen, z.B. rotate-left) sind Codeworte

  $\Rightarrow$ bei serieller Übertragung erlaubt dies die Erkennung/Korrektur von Bündelfehlern

Verfahren zur Fehlerbehandlung

- **Automatic Repeat Request** (ARQ): der Empfänger erkennt ein fehlerhaftes Symbol und fordert dies vom Sender erneut an
  - bidirektionale Kommunikation erforderlich
  - unpraktisch bei großer Entfernung / Echtzeitanforderungen

- **Vorwärtsfehlerkorrektur** (*Forward Error Correction*, FEC): die übertragene Information wird durch zusätzliche Redundanz (z.B. Prüfziffern) gesichert
  - der Empfänger erkennt fehlerhafte Codewörter und kann diese selbständig korrigieren

- je nach Einsatzzweck sind beide Verfahren üblich
- auch kombiniert

Hamming-Abstand

- **Hamming-Abstand**: die Anzahl der Stellen, an denen sich zwei Binärcodewörter der Länge w unterscheiden
- **Hamming-Gewicht**: Hamming-Abstand eines Codeworts vom Null-Wort

- Beispiel $\quad a = 0110\,0011$
  $\quad\quad\quad\quad\quad\quad b = 1010\,0111$

$\Rightarrow$ Hamming-Abstand von a und b ist 3

Hamming-Gewicht von b ist 5

- Java: `Integer.bitcount( a ^ b )`

Fehlererkennende und -korrigierende Codes

- Zur *Fehlererkennung* und *Fehlerkorrektur* ist eine Codierung mit Redundanz erforderlich
- Repräsentation enthält mehr Bits, als zur reinen Speicherung nötig wären

- Codewörter so wählen, dass sie alle paarweise mindestens den Hamming-Abstand d haben

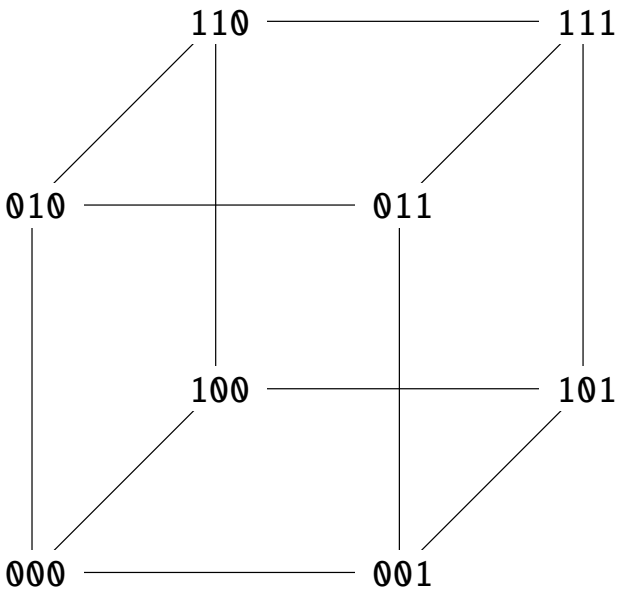
  dieser Abstand heißt dann **Minimalabstand** d

$\Rightarrow$ Fehlererkennung bis zu $(d-1)$ fehlerhaften Stellen

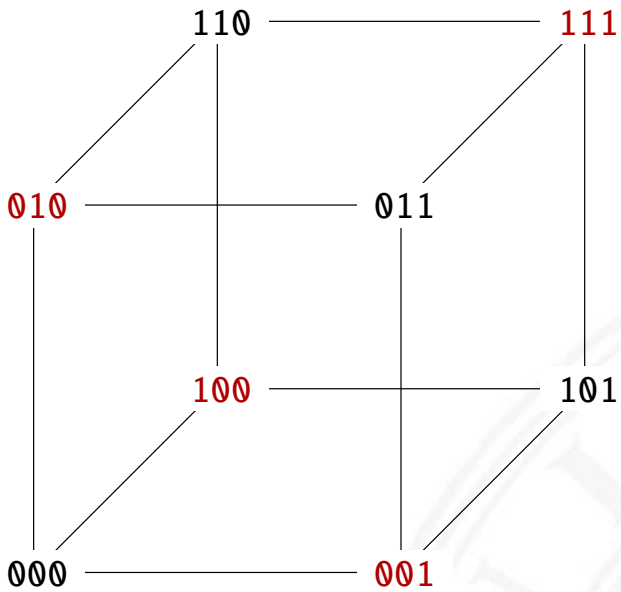
Fehlerkorrektur bis zu $((d-1)/2)$ –"–

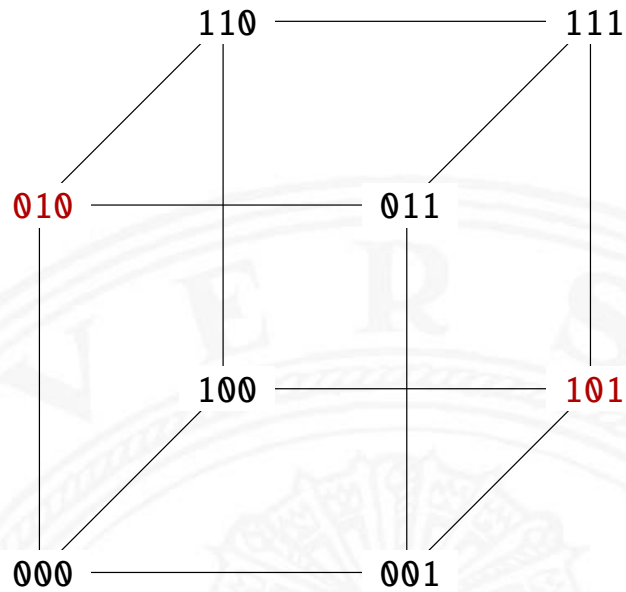
Fehlererkennende und -korrigierende Codes (cont.)

- Hamming-Abstand 2

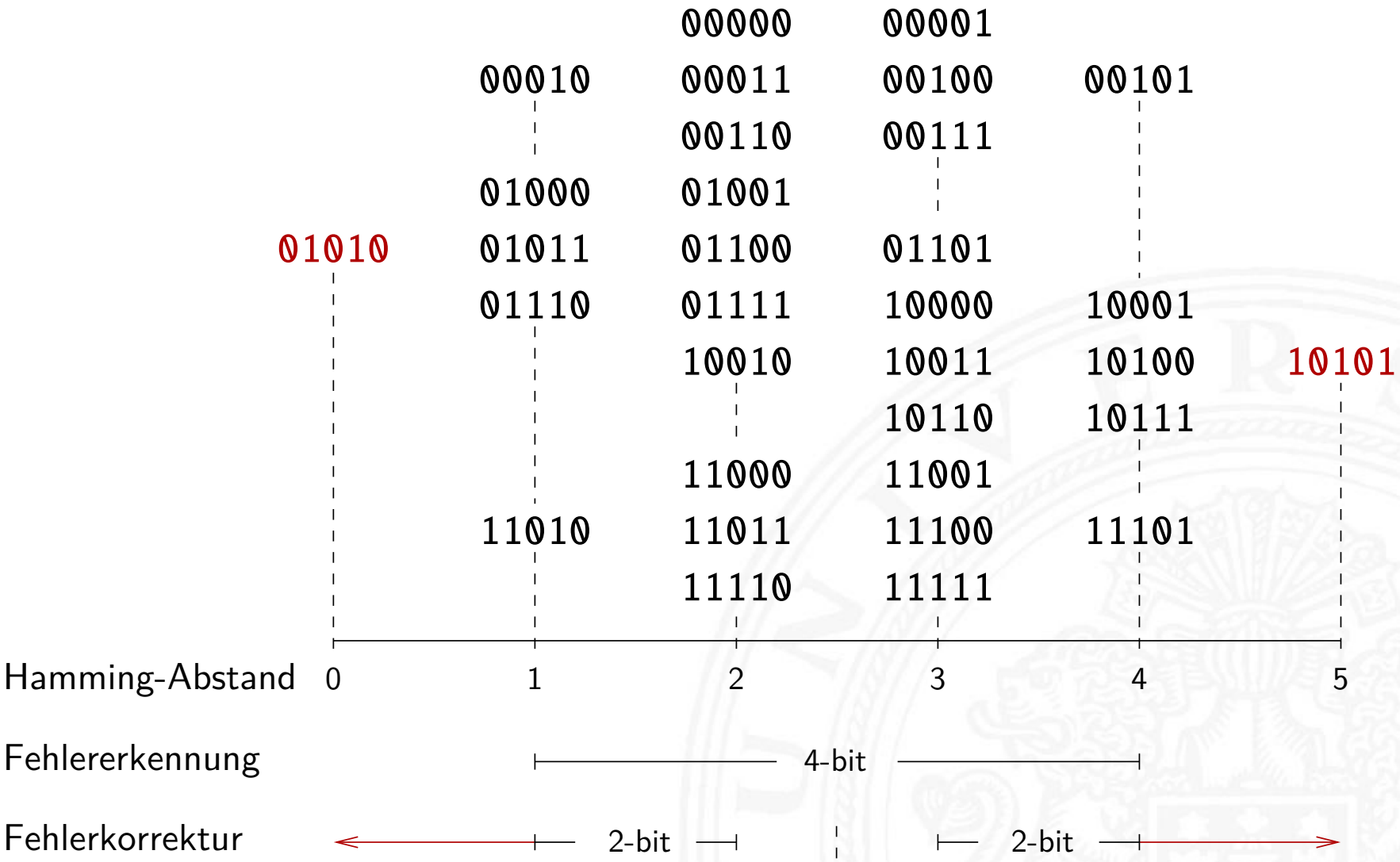


3





Fehlererkennende und -korrigierende Codes (cont.)



Prüfinformation

Man fügt den Daten **Prüfinformation** hinzu, oft **Prüfsumme** genannt

- zur Fehlererkennung
- zur Fehlerkorrektur
- zur Korrektur einfacher Fehler, Entdeckung schwerer Fehler

verschiedene Verfahren

- Prüfziffer, Parität
- Summenbildung
- CRC-Verfahren (*cyclic-redundancy check*)
- BCH-Codes (Bose, Ray-Chauduri, Hocquengham)
- RS-Codes (Reed-Solomon)

Paritätscode

- das Anfügen eines **Paritätsbits** an ein Binärcodewort $z = (z_1, \ldots, z_n)$ ist die einfachste Methode zur Erkennung von Einbitfehlern
- die Parität wird berechnet als

$$p = \left(\sum_{i=1}^{n} z_i \right) \mod 2$$

- **gerade Parität** (*even parity*): $y_{even} = (z_1, \ldots, z_n, p)$
  $p(y_{even}) = (\sum_i y_i) \mod 2 = 0$

  ungerade Parität (*odd parity*): $y_{odd} = (z_1, \ldots, z_n, \overline{p})$
  $p(y_{odd}) = (\sum_i y_i) \mod 2 = 1$

Paritätscode: Eigenschaften

- in der Praxis meistens Einsatz der ungeraden Parität:

  pro Codewort y_{odd} mindestens eine Eins $\Rightarrow$ elektr. Verbindung

- Hamming-Abstand zweier Codewörter im Paritätscode ist mindestens 2, weil sich bei Ändern eines Nutzbits jeweils auch die Parität ändert: $d = 2$

- Erkennung von Einbitfehlern möglich:

  Berechnung der Parität im Empfänger und Vergleich mit der erwarteten Parität
- Erkennung von (ungeraden) Mehrbitfehlern

Zweidimensionale Parität

- Anordnung der Daten / Informations-Bits als Matrix
- Berechnung der Parität für alle Zeilen und Spalten
- optional auch für Zeile/Spalte der Paritäten

- entdeckt 1-bit Fehler in allen Zeilen und Spalten
- erlaubt Korrektur von allen 1-bit und vielen n-bit Fehlern

- natürlich auch weitere Dimensionen möglich
  n-dimensionale Anordnung und Berechnung von n Paritätsbits

Zweidimensionale Parität: Beispiel

H	100 1000	0
A	100 0001	0
M	100 1101	0
M	100 1101	0
I	100 1001	1
N	100 1110	0
G	100 0111	0
	100 1001	1

Fehlerfall	100 1000	0
	100 0101	0
	110 1101	0
	100 1101	0
	000 1001	1
	100 1110	0
	100 0111	0
	100 1000	1

- Symbol: 7 ASCII-Zeichen, gerade Parität (*even*)
  64 bits pro Symbol (49 für Nutzdaten und 15 für Parität)
- links: Beispiel für ein Codewort und Paritätsbits
- rechts: empfangenes Codewort mit vier Fehlern,
  davon ein Fehler in den Paritätsbits

Zweidimensionale Parität: Einzelfehler

H	100 1000	0
A	100 0001	0
M	100 1101	0
M	100 1101	0
I	100 1001	1
N	100 1110	0
G	100 0111	0
	100 1001	1

Fehlerfall

100 1000	0
100 0101	0 1
100 1101	0
100 1101	0
100 1001	1
100 1110	0
100 0111	0
100 1001	1
1	

- Empfänger: berechnet Parität und vergleicht mit gesendeter P.
- Einzelfehler: Abweichung in je einer Zeile und Spalte

$\Rightarrow$ Fehler kann daher zugeordnet und korrigiert werden

- Mehrfachfehler: nicht alle, aber viele erkennbar (korrigierbar)

Zweidimensionale Parität: Dezimalsystem

- Parität als Zeilen/Spaltensumme mod 10 hinzufügen

- Daten

3	7	4
5	4	8
1	3	5

Parität

3	7	4	4
5	4	8	7
1	3	5	9
9	4	7	0

Fehlerfall

3	7	4	4	
5	4	3	7	2
1	3	5	9	
9	4	7	0	
		2		

International Standard Book Number

ISBN-10 (1970), ISBN-13

- an EAN (*European Article Number*) gekoppelt
- Codierung eines Buches als Tupel

1. Präfix (nur ISBN-13)
2. Gruppennummer für den Sprachraum als Fano-Code:
   $0-7$, $80-94$, $950-995$, $9960-9989$, $99900-99999$
   - 0, 1: englisch – AUS, UK, USA ...
   - 2: französisch – F ...
   - 3: deutsch – A, DE, CH
   - ...
3. Verlag, Nummer als Fano-Code:
   $00-19$ (1 Mio Titel), $20-699$ (100 000 Titel) usw.
4. verlagsinterne Nummer
5. Prüfziffer

ISBN-10: Prüfverfahren

- ISBN-10 Zahl: $z_1, z_2, \ldots, z_{10}$
- Prüfsumme berechnen, Symbol X steht für Ziffer 10

$$\sum_{i=1}^{9} (i \cdot z_i) \mod 11 = z_{10}$$

- ISBN-Zahl zulässig, genau dann wenn

$$\sum_{i=1}^{10} (i \cdot z_i) \mod 11 = 0$$

- Beispiel: 1-292-10176-8 Bryant, O'Hallaron [BO15]

$$1 \cdot 1 + 2 \cdot 2 + 3 \cdot 9 + 4 \cdot 2 + 5 \cdot 1 + 6 \cdot 0 + 7 \cdot 1 + 8 \cdot 7 + 9 \cdot 6 = 162$$

$$162 \mod 11 = 8$$

$$162 + 10 \cdot 8 = 242 \qquad 242 \mod 11 = 0$$

ISBN: Fehlertypen

- Prüfziffer schützt gegen Verfälschung einer Ziffer
  –"– Vertauschung zweier Ziffern
  –"– „Falschdopplung“ einer Ziffer

- Beispiel: vertausche i-te und j-te Ziffer (mit $i \neq j$)
  Prüfsumme: $\langle korrekt \rangle$ - $\langle falsch \rangle$
  $= i \cdot z_i + j \cdot z_j - j \cdot z_i - i \cdot z_j = (i - j) \cdot (z_i - z_j)$ mit $z_i \neq z_j$.

3-fach Wiederholungscode / (3,1)-Hamming-Code

- dreifache Wiederholung jedes Datenworts
- (3,1)-Hamming-Code: Generatormatrix ist $G = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$
- Codewörter ergeben sich als Multiplikation von G mit dem Informationsvektor u (jeweils ein Bit)

  $u = 0: \quad x = (111)^T \cdot (0) = (000)$

  $u = 1: \quad x = (111)^T \cdot (1) = (111)$

- Verallgemeinerung als n-fach Wiederholungscode
- systematischer Code mit Minimalabstand $D = n$
- Decodierung durch Mehrheitsentscheid: 1-bit Fehlerkorrektur

– Nachteil: geringe Datenrate

Hamming-Code

- Hamming-Abstand 3
- korrigiert 1-bit Fehler, erkennt (viele) 2-bit und 3-bit Fehler

(N, n)-Hamming-Code

- Datenwort n-bit $(d_1, d_2, \ldots, d_n)$

  um k-Prüfbits ergänzen $(p_1, p_2, \ldots, p_k)$

$\Rightarrow$ Codewort mit $N = n + k$ bit

- Fehlerkorrektur gewährleisten: $2^k \geq N + 1$
  - 2^k Kombinationen mit k-Prüfbits
  - 1 fehlerfreier Fall
  - N zu markierende Bitfehler

Hamming-Code (cont.)

1. bestimme kleinstes k mit $n \leq 2^k - k - 1$
2. Prüfbits an Bitpositionen: $2^0, 2^1, \ldots, 2^{k-1}$
   Originalbits an den übrigen Positionen

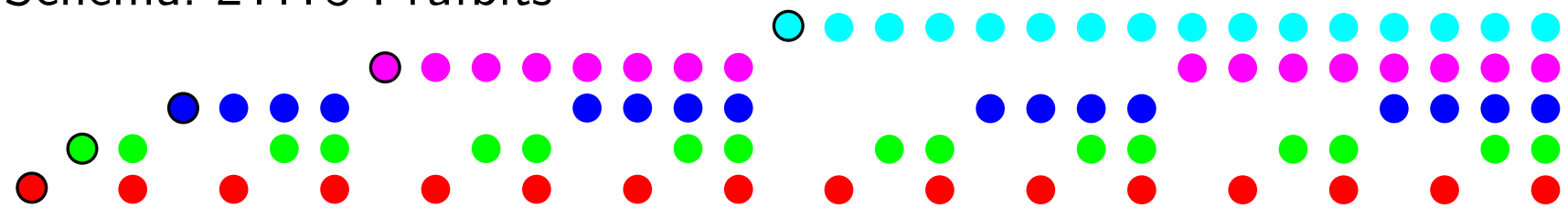
Position	1	2	3	4	5	6	7	8	9	...
	001	010	011	100	101	110	111	000	001	
Bit	p_1	p_2	d_1	p_3	d_2	d_3	d_4	p_4	d_5	...

3. berechne Prüfbit i als mod 2-Summe der Bits (XOR),
   deren Positionsnummer ein gesetztes i-bit enthält

$$p_1 = d_1 \oplus d_2 \oplus d_4 \oplus d_5 \oplus \ldots$$
$$p_2 = d_1 \oplus d_3 \oplus d_4 \oplus d_6 \oplus \ldots$$
$$p_3 = d_2 \oplus d_3 \oplus d_4 \oplus d_8 \oplus \ldots$$
$$p_4 = d_5 \oplus d_6 \oplus d_7 \oplus d_8 \oplus \ldots$$
$$\ldots$$

Hamming-Code (cont.)

Schema: 2 . . . 5 Prüfbits



2^0	2^1		2^2				2^3								2^4															
c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}	c_{18}	c_{19}	c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}	c_{26}	c_{27}	c_{28}	c_{29}	c_{30}	c_{31}
p_1	p_2	d_1	p_3	d_2	d_3	d_4	p_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	p_5	d_{12}	d_{13}	d_{14}	d_{15}	d_{16}	d_{17}	d_{18}	d_{19}	d_{20}	d_{21}	d_{22}	d_{23}	d_{24}	d_{25}	d_{26}

(7,4)-Hamming-Code

- $p_1 = d_1 \oplus d_2 \oplus d_4$
  $p_2 = d_1 \oplus d_3 \oplus d_4$
  $p_3 = d_2 \oplus d_3 \oplus d_4$

(15,11)-Hamming-Code

- $p_1 = d_1 \oplus d_2 \oplus d_4 \oplus d_5 \oplus d_7 \oplus d_9 \oplus d_{11}$
  $p_2 = d_1 \oplus d_3 \oplus d_4 \oplus d_6 \oplus d_7 \oplus d_{10} \oplus d_{11}$
  $p_3 = d_2 \oplus d_3 \oplus d_4 \oplus d_8 \oplus d_9 \oplus d_{10} \oplus d_{11}$
  $p_4 = d_5 \oplus d_6 \oplus d_7 \oplus d_8 \oplus d_9 \oplus d_{10} \oplus d_{11}$

(7,4)-Hamming-Code

- sieben Codebits für je vier Datenbits
- linearer (7,4)-Block-Code
- Generatormatrix ist

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Codewort $c = G \cdot d$

(7,4)-Hamming-Code (cont.)

- Prüfmatrix H orthogonal zu gültigen Codewörtern: $H \cdot c = 0$

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

  für ungültige Codewörter $H \cdot c \neq 0$

$\Rightarrow$ „Fehlersyndrom“ liefert Information über Fehlerposition / -art

Fazit: Hamming-Codes

$+$ größere Wortlangen: besseres Verhältnis von Nutz- zu Prüfbits

$+$ einfaches Prinzip, einfach decodierbar

$-$ es existieren weit bessere Codes

(7,4)-Hamming-Code: Beispiel

- Codieren von $d = (0, 1, 1, 0)$

$$c = G \cdot d = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

(7,4)-Hamming-Code: Beispiel (cont.)

- Prüfung von Codewort $c = (1, 1, 0, 0, 1, 1, 0)$

$$H \cdot c = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

(7,4)-Hamming-Code: Beispiel (cont.)

- im Fehlerfall $c = (1, 1, 1, 0, 1, 1, 0)$

$$H \cdot c = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

$\Rightarrow$ Fehlerstelle:

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Index: 1 2 3 4 5 6 7

Fehlerrate

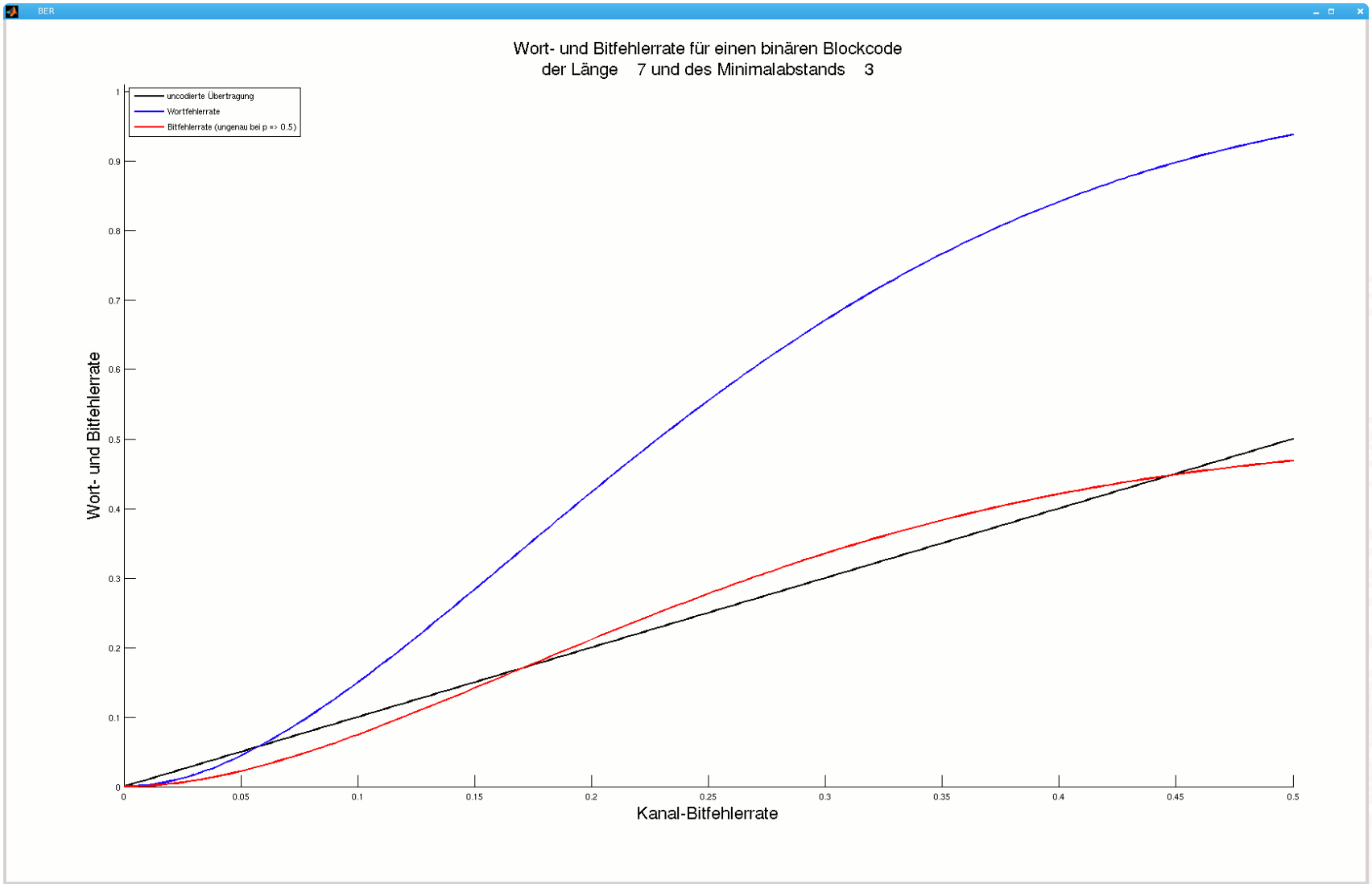
- (n, k)-Code: k-Informationsbits werden in n-Bits codiert
- Minimalabstand d der Codewörter voneinander
- ermöglicht Korrektur von r Bitfehlern $\quad r \leq (d-1)/2$

$\Rightarrow$ nicht korrigierbar sind: $r+1, r+2, \ldots, n$ Bitfehler

- Übertragungskanal hat Bitfehlerwahrscheinlichkeit

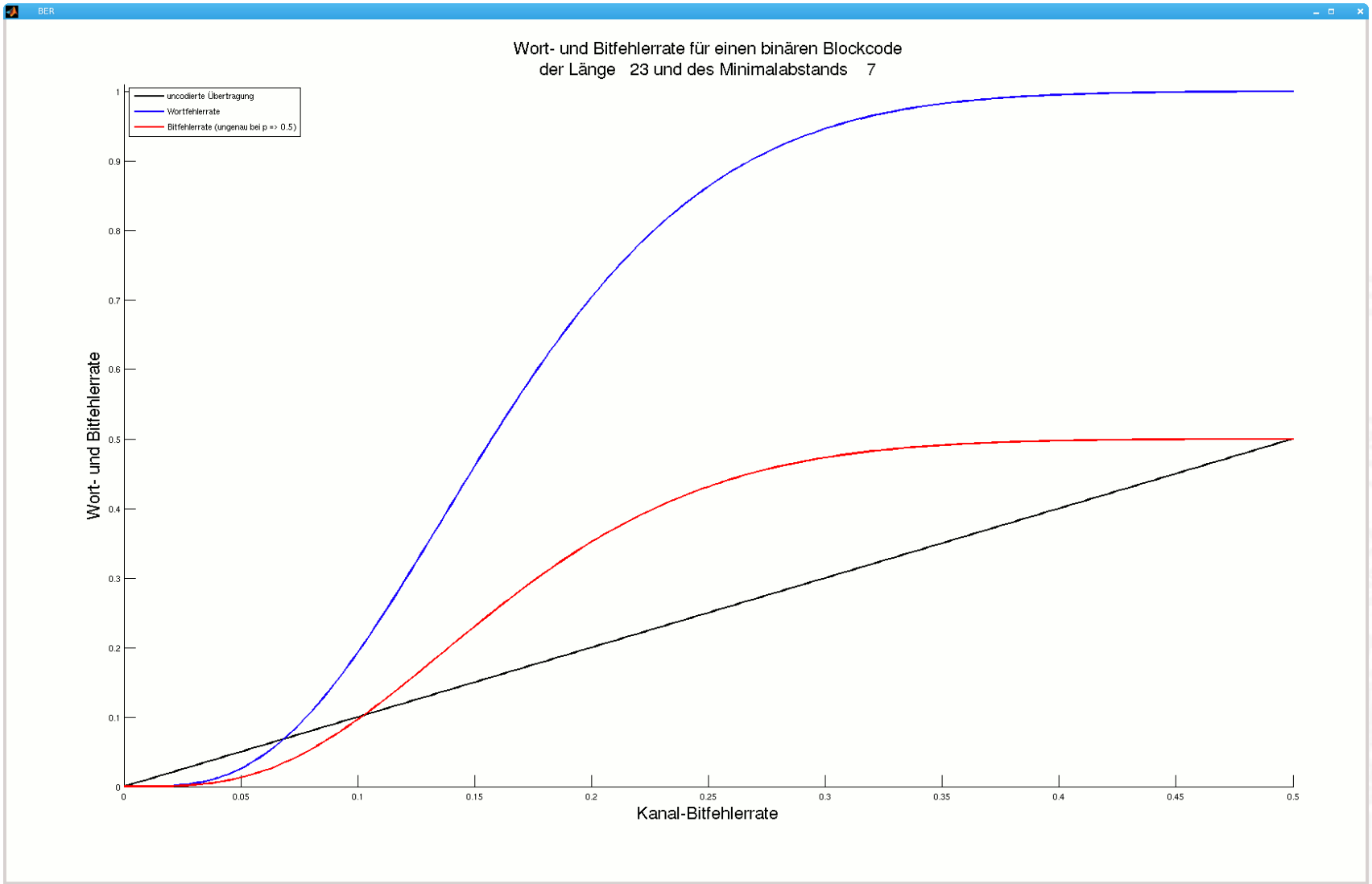
$\Rightarrow$ Wortfehlerwahrscheinlichkeit: Summe der Wahrscheinlichkeiten nicht korrigierbarer Bitfehler

Fehlerrate: (7,4)-Hamming-Code



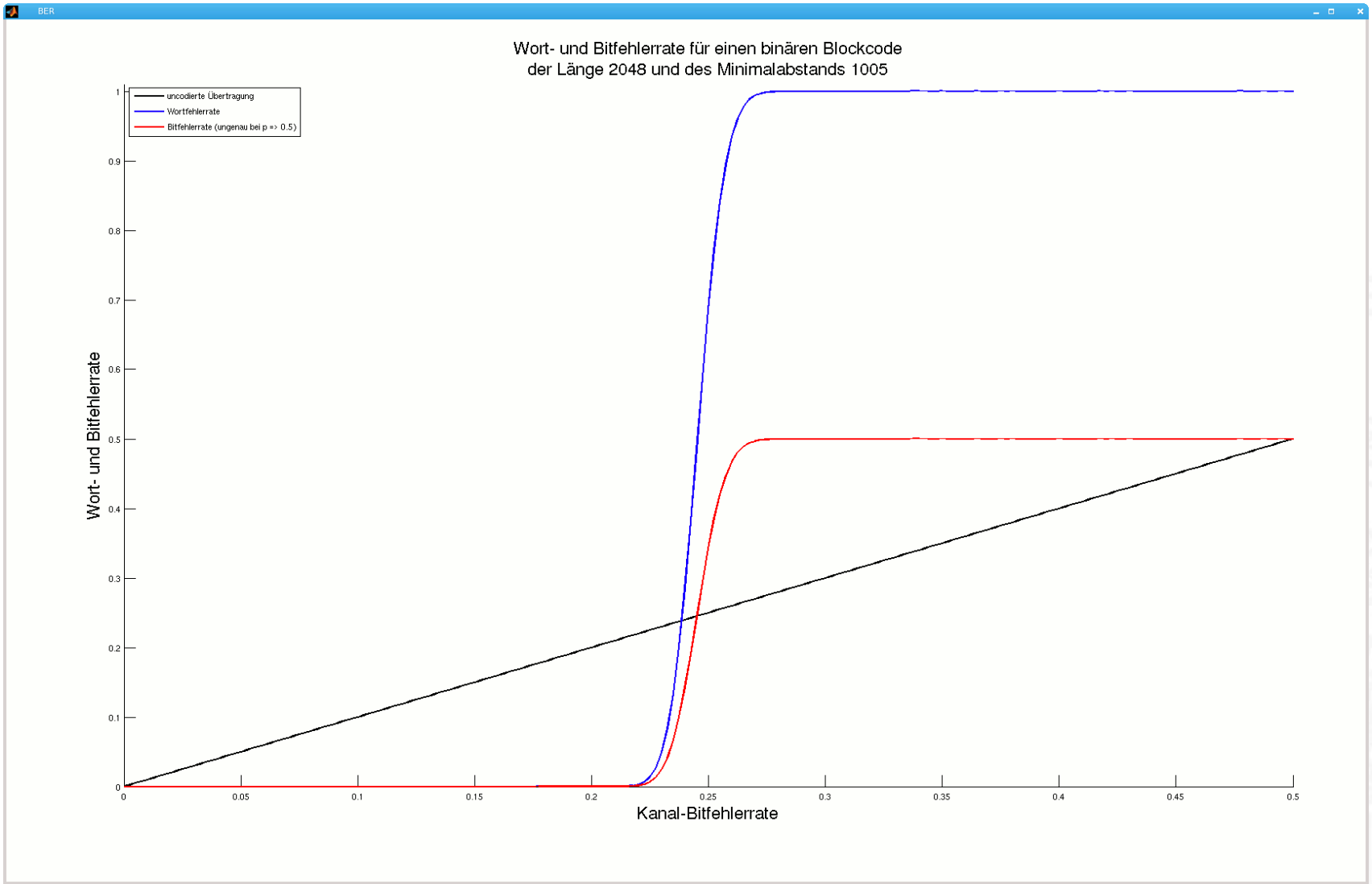
[Hei05]

Fehlerrate: (23,12)-Golay-Code



[Hei05]

Fehlerrate: (2048,8)-Randomcode



[Hei05]

Binärpolynome

- jedem n-bit Wort $(d_1, d_2, \ldots, d_n)$ lässt sich ein Polynom über dem Körper $\{0, 1\}$ zuordnen
- Beispiel, mehrere mögliche Zuordnungen

$$\begin{aligned} 100\,1101 &= 1 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 1 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0 \\ &= x^6 + x^3 + x^2 + x^0 \\ &= x^0 + x^3 + x^4 + x^6 \\ &= x^0 + x^{-3} + x^{-4} + x^{-6} \\ &\ldots \end{aligned}$$

- mit diesen Polynomen kann „gerechnet“ werden: Addition, Subtraktion, Multiplikation, Division
- Theorie: Galois-Felder

Zyklische Codes (CRC)

CRC (*Cyclic Redundancy Check*)

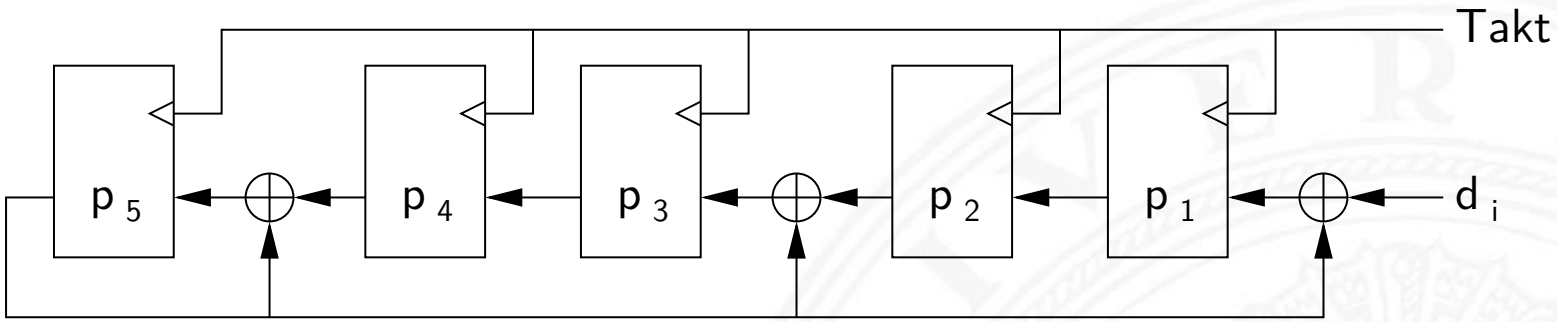
- Polynomdivision als Basis für CRC-Codes erzeugt Prüfbits
- zyklisch: Codewörter werden durch Schieben und Modifikation (mod 2 Summe) ineinander überführt

- Familie von Codes zur Fehlererkennung
  insbesondere auch zur Erkennung von Bündelfehlern

- in sehr vielen Codes benutzt
  - Polynom 0x04C11DB7 (CRC-32) in Ethernet, ZIP, PNG ...
  - weitere CRC-Codes in USB, ISDN, GSM, openPGP ...

Zyklische Codes (CRC) (cont.)

- Sehr effiziente Software- oder Hardwarerealisierung
  - rückgekoppelte Schieberegister und XOR
    LFSR (*Linear Feedback Shift Register*)
  - Beispiel $x^5 + x^4 + x^2 + 1$



- Codewort erstellen
  - Datenwort d_i um k 0-bits verlängern, Grad des Polynoms: k
  - bitweise in CRC-Check schieben
  - Divisionsrest bildet Registerinhalt p_i
  - Prüfbits p_i an ursprüngliches Datenwort anhängen

Zyklische Codes (CRC) (cont.)

- Test bei Empfänger
  - übertragenes Wort bitweise in CRC-Check schieben
    gleiches Polynom / Hardware wie bei Codierung
  - fehlerfrei, wenn Divisionsrest/Registerinhalt $= 0$

- je nach Polynom (# Prüfbits) unterschiedliche Güte
- Galois-Felder als mathematische Grundlage

- `en.wikipedia.org/wiki/Cyclic_redundancy_check`
  `en.wikipedia.org/wiki/Computation_of_CRC`
  `de.wikipedia.org/wiki/Zyklische_Redundanzprüfung`
  `de.wikipedia.org/wiki/LFSR`

Praxisbeispiel: EAN-13 Produktcode

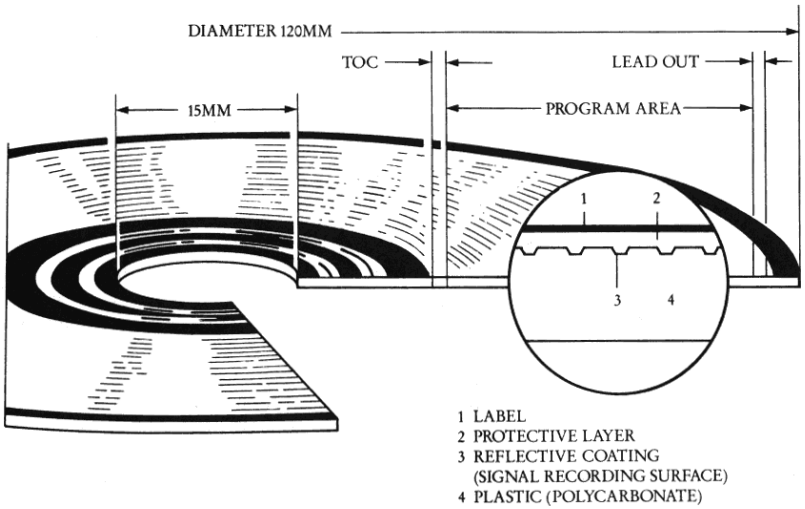
de.wikipedia.org/wiki/European_Article_Number

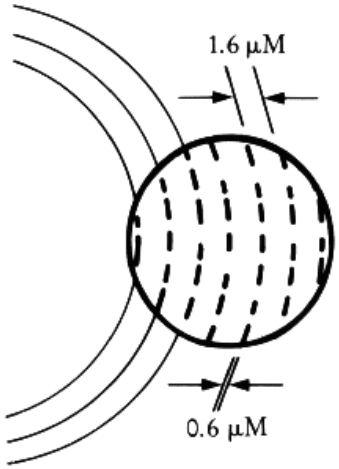
Kombination diverser Codierungen:

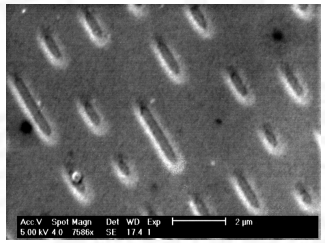
- Land, Unternehmen, Artikelnummer, Prüfsumme
- 95-stelliges Bitmuster
  - schwarz $\widehat{=}$ 1, weiss $\widehat{=}$ 0
  - max. vier aufeinanderfolgende weisse/schwarze Bereiche
  - Randzeichen: 101
    Trennzeichen in der Mitte: 01010
- 13 Ziffern: 7 links, 6 rechts
  - jede Ziffer mit 7 bit codiert, je zwei Linien und Freiräume
  - 3 Varianten pro Ziffer: links ungerade/gerade, rechts
  - 12 Ziffern Code, 11 Ziffern direkt codiert
  - 1. Ziffer über Abfolge von u/g Varianten
  - 13. Ziffer als Prüfsumme

Compact Disc
Audio-CD und CD-ROM

- Polycarbonatscheibe, spiralförmige geprägte Datenspur







  - spiralförmige Spur, ca. 16000 Windungen, Start innen
  - geprägte Vertiefungen *pits*, dazwischen *lands*
  - Wechsel pit/land oder land/pit codiert 1, dazwischen 0
- Auslesen durch Intensität von reflektiertem Laserstrahl
- 650 MiB Kapazität, Datenrate $\approx$ 150 KiB/sec (1x speed)

Compact Disc (cont.)
Audio-CD und CD-ROM

- von Anfang an auf billigste Fertigung ausgelegt
- mehrstufige Fehlerkorrekturcodierung fest vorgesehen
- Kompensation von Fertigungsmängeln und -toleranzen
- Korrektur von Staub und Kratzern etc.

- Audio-CD: Interpolation nicht korrigierbarer Fehler
- Daten-CD: geschachtelte weitere Codierung
- Bitfehlerrate $\leq 10^{11}$

Compact Disc: Mehrstufige Codierung

- Daten in *Frames* à 24 Bytes aufteilen
- 75 *Sektoren* mit je 98 Frames pro Sekunde
- Sektor enthält 2 352 Bytes Nutzdaten (und 98 Bytes *Subcode*)

- pro Sektor 784 Byte Fehlerkorrektur hinzufügen
- Interleaving gegen Burst-Fehler (z.B. Kratzer)
- Code kann bis 7 000 fehlende Bits korrigieren

- *eight-to-fourteen* Modulation: 8-Datenbits in 14-Codebits
  2 . . . 10 Nullen zwischen zwei Einsen (pit/land Übergang)

- Daten-CD zusätzlich mit äußerem 2D *Reed-Solomon Code*
- pro Sektor 2 048 Bytes Nutzdaten, 276 Bytes RS-Fehlerschutz

Farbbilder: JPEG

Joint Picture Experts Group Bildformat (1992)

- für die Speicherung von Fotos / Bildern
- verlustbehaftet

mehrere Codierungsschritte

1.	Farbraumkonvertierung: RGB nach YUV	verlustbehaftet
2.	Aufteilung in Blöcke zu je 8x8 Pixeln	verlustfrei
3.	DCT (*discrete cosinus transformation*)	verlustfrei
4.	Quantisierung (einstellbar)	verlustbehaftet
5.	Huffman-Codierung	verlustfrei

Video: MPEG

Motion Picture Experts Group: Sammelname der Organisation und diverser aufeinander aufbauender Standards

Codierungsschritte für Video

1. Einzelbilder wie JPEG (YUV, DCT, Huffman)
2. Differenzbildung mehrerer Bilder (Bewegungskompensation)
3. *Group of Pictures* (*I*-Frames, *P*-Frames, *B*-Frames)
4. Zusammenfassung von Audio, Video, Metadaten im sogenannten PES (*Packetized Elementary Stream*)
5. *Transport-Stream* Format für robuste Datenübertragung

Literatur

[Ham87] R.W. Hamming: *Information und Codierung*. VCH, 1987. ISBN 3–527–26611–9

[Hei05a] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript*. Universität Hamburg, FB Informatik, 2005. `tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1`

[Hei05b] K. von der Heide: *Vorlesung: Digitale Datenübertragung*. Universität Hamburg, FB Informatik, 2005, Vorlesungsskript. `tams.informatik.uni-hamburg.de/lectures/2005ss/vorlesung/Digit`

[HenHA] N. Hendrich: *HADES — HAmburg DEsign System*. Universität Hamburg, FB Informatik, Lehrmaterial. `tams.informatik.uni-hamburg.de/applets/hades/webdemos`

Literatur (cont.)

[RL09] W.E. Ryan, S. Lin: *Channel codes: classical and modern*. Cambridge University Press, 2009. ISBN 0–521–84868–7

[Knu85] D.E. Knuth: *Dynamic Huffman Coding*. in: *J. of Algorithms* 6 (1985), Nr. 2, S. 163–180

[Knu11] D.E. Knuth: *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 2011. ISBN 978–0–201–03804–0

Gliederung

Gliederung (cont.)

Schaltfunktionen

- **Schaltfunktion**: eine eindeutige Zuordnungsvorschrift f, die jeder Wertekombination $(b_1, b_2, \ldots, b_n)$ von Schaltvariablen einen Wert zuweist:

$$y = f(b_1, b_2, \ldots, b_n) \in \{0, 1\}$$

- **Schaltvariable**: eine Variable, die nur endlich viele Werte annehmen kann – typisch sind binäre Schaltvariablen
- **Ausgangsvariable**: die Schaltvariable am Ausgang der Funktion, die den Wert y annimmt
- bereits bekannt: *elementare Schaltfunktionen* (AND, OR usw.) wir betrachten jetzt Funktionen von n Variablen

Beschreibung von Schaltfunktionen

- textuelle Beschreibungen

  formale Notation, Schaltalgebra, Beschreibungssprachen

- tabellarische Beschreibungen

  Funktionstabelle, KV-Diagramme ...

- graphische Beschreibungen

  Kantorovic-Baum (Datenflussgraph), Schaltbild ...

- Verhaltensbeschreibungen $\Rightarrow$ „was“
- Strukturbeschreibungen $\Rightarrow$ „wie“

Funktionstabelle

- Tabelle mit Eingängen x_i und Ausgangswert $y = f(x)$
- Zeilen im Binärcode sortiert
- zugehöriger Ausgangswert eingetragen

x_3	x_2	x_1	$f(x)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Funktionstabelle (cont.)

- Kurzschreibweise: nur die Funktionswerte notiert

  $f(x_2, x_1, x_0) = \{0, 0, 1, 1, 0, 0, 1, 0\}$

- n Eingänge: Funktionstabelle umfasst 2^n Einträge
- Speicherbedarf wächst exponentiell mit n

  z.B.: 2^{33} Bit für 16-bit Addierer (16+16+1 Eingänge)

$\Rightarrow$ daher nur für kleine Funktionen geeignet

- Erweiterung auf *don't-care* Terme, s.u.

Verhaltensbeschreibung

- Beschreibung einer Funktion als Text über ihr Verhalten
- Problem: umgangssprachliche Formulierungen oft mehrdeutig
- logische Ausdrücke in Programmiersprachen
- Einsatz spezieller (Hardware-) Beschreibungssprachen
  z.B.: Verilog, VHDL, SystemC

umgangssprachlich: Mehrdeutigkeit

„Das Schiebedach ist ok (y), wenn der Öffnungskontakt (x_0) **oder** der Schließkontakt (x_1) funktionieren **oder beide nicht** aktiv sind (Mittelstellung des Daches)“

K. Henke, H.-D. Wuttke: *Schaltsysteme* [WH03]

zwei mögliche Missverständnisse

- *oder*: als OR oder XOR?
- *beide nicht*: x_1 und x_0 nicht, oder x_1 nicht und x_0 nicht?

$\Rightarrow$ je nach Interpretation völlig unterschiedliche Schaltung

Strukturbeschreibung

- **Strukturbeschreibung**: eine Spezifikation der konkreten Realisierung einer Schaltfunktion

- vollständig geklammerte algebraische Ausdrücke

  $f = x_1 \oplus (x_2 \vee x_3)$
- Datenflussgraphen
- Schaltpläne mit Gattern (s.u.)
- PLA-Format für zweistufige AND-OR Schaltungen (s.u.)
- ...

Funktional vollständige Basismenge

- Menge M von Verknüpfungen über $GF(2)$ heißt **funktional vollständig**, wenn die Funktionen $f, g \in T_2$:

$$f(x_1, x_2) = x_1 \oplus x_2$$
$$g(x_1, x_2) = x_1 \wedge x_2$$

  allein mit den in M enthaltenen Verknüpfungen geschrieben werden können

- Boole'sche Algebra: { AND, OR, NOT }
- Reed-Muller Form: { AND, XOR, 1 }
- technisch relevant: { NAND }, { NOR }

Normalformen

- Jede Funktion kann auf beliebig viele Arten beschrieben werden

Suche nach Standardformen

- in denen man alle Funktionen darstellen kann
- Darstellung mit universellen Eigenschaften
- eindeutige Repräsentation $\Rightarrow$ einfache Überprüfung, ob (mehrere) gegebene Funktionen übereinstimmen

- Beispiel: Darstellung ganzrationaler Funktionen

$$f(x) = \sum_{i=0}^{n} a_i x^i$$

a_i: Koeffizienten
x^i: Basisfunktionen

Normalformen (cont.)

Normalform einer Boole'schen Funktion

- analog zur Potenzreihe
- als Summe über Koeffizienten $\{0, 1\}$ und Basisfunktionen

$$f = \sum_{i=1}^{2^n} \hat{f}_i \hat{B}_i, \quad \hat{f}_i \in \mathrm{GF}(2)$$

mit $\hat{B}_1, \ldots, \hat{B}_{2^n}$ einer Basis des T^n

Definition: Normalform

- funktional vollständige Menge V der Verknüpfungen von $\{0, 1\}$
- Seien $\oplus, \otimes \in V$ und assoziativ

- Wenn sich alle $f \in T^n$ in der Form

$$f = (\hat{f}_1 \otimes \hat{B}_1) \oplus \ldots \oplus (\hat{f}_{2^n} \otimes \hat{B}_{2^n})$$

schreiben lassen, so wird die Form als **Normalform** und die Menge der $\hat{B}_i$ als **Basis** bezeichnet.

- Menge von 2^n Basisfunktionen $\hat{B}_i$
  Menge von 2^{2^n} möglichen Funktionen f

Disjunktive Normalform (DNF)

- **Minterm**: die UND-Verknüpfung *aller* Schaltvariablen einer Schaltfunktion, die Variablen dürfen dabei negiert oder nicht negiert auftreten

- **Disjunktive Normalform**: die disjunktive Verknüpfung aller Minterme m mit dem Funktionswert 1

$$f = \bigvee_{i=1}^{2^n} \hat{f}_i \cdot m(i), \quad \text{mit} \quad m(i): \text{Minterm}(i)$$

auch: *kanonische disjunktive Normalform*
sum-of-products (SOP)

Disjunktive Normalform: Minterme

- Beispiel: alle 2^3 Minterme für drei Variablen
- jeder Minterm nimmt nur für eine Belegung der Eingangsvariablen den Wert 1 an

x_3	x_2	x_1	Minterme
0	0	0	$\overline{x_3} \wedge \overline{x_2} \wedge \overline{x_1}$
0	0	1	$\overline{x_3} \wedge \overline{x_2} \wedge x_1$
0	1	0	$\overline{x_3} \wedge x_2 \wedge \overline{x_1}$
0	1	1	$\overline{x_3} \wedge x_2 \wedge x_1$
1	0	0	$x_3 \wedge \overline{x_2} \wedge \overline{x_1}$
1	0	1	$x_3 \wedge \overline{x_2} \wedge x_1$
1	1	0	$x_3 \wedge x_2 \wedge \overline{x_1}$
1	1	1	$x_3 \wedge x_2 \wedge x_1$

Disjunktive Normalform: Beispiel

x_3	x_2	x_1	$f(x)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

- Funktionstabelle: Minterm $0 \equiv \overline{x_i} \quad 1 \equiv x_i$
- für f sind nur drei Koeffizienten der DNF gleich 1

$\Rightarrow$ DNF: $f(x) = (\overline{x_3} \wedge x_2 \wedge \overline{x_1}) \vee (\overline{x_3} \wedge x_2 \wedge x_1) \vee (x_3 \wedge x_2 \wedge \overline{x_1})$

Allgemeine disjunktive Form

- **disjunktive Form** (sum-of-products): die disjunktive Verknüpfung (ODER) von Termen. Jeder Term besteht aus der UND-Verknüpfung von Schaltvariablen, die entweder direkt oder negiert auftreten können
- entspricht dem Zusammenfassen („Minimierung“) von Termen aus der disjunktiven Normalform
- disjunktive Form ist nicht eindeutig (keine Normalform)

- Beispiel

  DNF $f(x) = (\overline{x_3} \wedge x_2 \wedge \overline{x_1}) \vee (\overline{x_3} \wedge x_2 \wedge x_1) \vee (x_3 \wedge x_2 \wedge \overline{x_1})$

  minimierte disjunktive Form $f(x) = (\overline{x_3} \wedge x_2) \vee (x_3 \wedge x_2 \wedge \overline{x_1})$

Allgemeine disjunktive Form

- **disjunktive Form** (sum-of-products): die disjunktive Verknüpfung (ODER) von Termen. Jeder Term besteht aus der UND-Verknüpfung von Schaltvariablen, die entweder direkt oder negiert auftreten können
- entspricht dem Zusammenfassen („Minimierung“) von Termen aus der disjunktiven Normalform
- disjunktive Form ist nicht eindeutig (keine Normalform)

- Beispiel

  DNF $f(x) = (\overline{x_3} \wedge x_2 \wedge \overline{x_1}) \vee (\overline{x_3} \wedge x_2 \wedge x_1) \vee (x_3 \wedge x_2 \wedge \overline{x_1})$

  minimierte disjunktive Form $f(x) = (\overline{x_3} \wedge x_2) \vee (x_3 \wedge x_2 \wedge \overline{x_1})$

  $f(x) = (x_2 \wedge \overline{x_1}) \vee (\overline{x_3} \wedge x_2 \wedge x_1)$

Konjunktive Normalform (KNF)

- **Maxterm**: die ODER-Verknüpfung *aller* Schaltvariablen einer Schaltfunktion, die Variablen dürfen dabei negiert oder nicht negiert auftreten

- **Konjunktive Normalform**: die konjunktive Verknüpfung aller Maxterme μ mit dem Funktionswert 0

$$f = \bigwedge_{i=1}^{2^n} \hat{f}_i \cdot \mu(i), \quad \text{mit} \quad \mu(i): \ \text{Maxterm}(i)$$

auch: *kanonische konjunktive Normalform*
product-of-sums (POS)

Konjunktive Normalform: Maxterme

- Beispiel: alle 2^3 Maxterme für drei Variablen
- jeder Maxterm nimmt nur für eine Belegung der Eingangsvariablen den Wert 0 an

x_3	x_2	x_1	Maxterme
0	0	0	$x_3 \vee x_2 \vee x_1$
0	0	1	$x_3 \vee x_2 \vee \overline{x_1}$
0	1	0	$x_3 \vee \overline{x_2} \vee x_1$
0	1	1	$x_3 \vee \overline{x_2} \vee \overline{x_1}$
1	0	0	$\overline{x_3} \vee x_2 \vee x_1$
1	0	1	$\overline{x_3} \vee x_2 \vee \overline{x_1}$
1	1	0	$\overline{x_3} \vee \overline{x_2} \vee x_1$
1	1	1	$\overline{x_3} \vee \overline{x_2} \vee \overline{x_1}$

Konjunktive Normalform: Beispiel

x_3	x_2	x_1	f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

- Funktionstabelle: Maxterm $0 \equiv x_i \quad 1 \equiv \overline{x_i}$
- für f sind fünf Koeffizienten der KNF gleich 0

$\Rightarrow$ KNF: $f(x) = (x_3 \vee x_2 \vee x_1) \wedge (x_3 \vee x_2 \vee \overline{x_1}) \wedge (\overline{x_3} \vee x_2 \vee x_1) \wedge (\overline{x_3} \vee x_2 \vee \overline{x_1}) \wedge (\overline{x_3} \vee \overline{x_2} \vee \overline{x_1})$

Allgemeine konjunktive Form

- **konjunktive Form** (product-of-sums): die konjunktive Verknüpfung (UND) von Termen. Jeder Term besteht aus der ODER-Verknüpfung von Schaltvariablen, die entweder direkt oder negiert auftreten können
- entspricht dem Zusammenfassen („Minimierung“) von Termen aus der konjunktiven Normalform
- konjunktive Form ist nicht eindeutig (keine Normalform)

- Beispiel

KNF $f(x) = (x_3 \vee x_2 \vee x_1) \wedge (x_3 \vee x_2 \vee \overline{x_1}) \wedge (\overline{x_3} \vee x_2 \vee x_1) \wedge (\overline{x_3} \vee x_2 \vee \overline{x_1}) \wedge (\overline{x_3} \vee \overline{x_2} \vee \overline{x_1})$

minimierte konjunktive Form

$f(x) = (x_3 \vee x_2) \wedge (x_2 \vee x_1) \wedge (\overline{x_3} \vee \overline{x_1})$

Reed-Muller Form

- **Reed-Muller Form**: die additive Verknüpfung aller Reed-Muller-Terme mit dem Funktionswert 1

$$f = \bigoplus_{i=1}^{2^n} \hat{f}_i \cdot RM(i)$$

- mit den Reed-Muller Basisfunktionen $RM(i)$

- Erinnerung: Addition im GF(2) ist die XOR-Operation

Reed-Muller Form: Basisfunktionen

- Basisfunktionen sind:

$\{1\}$, (0 Variablen)

$\{1, \quad x_1\}$, (1 Variable)

$\{1, \quad x_1, \quad x_2, x_2x_1\}$, (2 Variablen)

$\{1, \quad x_1, \quad x_2, x_2x_1, \quad x_3, x_3x_1, x_3x_2, x_3x_2x_1\}$, (3 Variablen)

...

$\{RM(n-1), \quad x_n \cdot RM(n-1)\}$ (n Variablen)

- rekursive Bildung: bei n bit alle Basisfunktionen von $(n-1)$-bit und zusätzlich das Produkt von x_n mit den Basisfunktionen von $(n-1)$-bit

Reed-Muller Form: Umrechnung

Umrechnung von gegebenem Ausdruck in Reed-Muller Form?

- Ersetzen der Negation: $\overline{a} = a \oplus 1$

  Ersetzen der Disjunktion: $a \vee b = a \oplus b \oplus ab$

  Ausnutzen von: $a \oplus a = 0$

- Beispiel

$$
\begin{aligned}
f(x_1, x_2, x_3) &= (\overline{x_1} \vee x_2)x_3 \\
&= (\overline{x_1} \oplus x_2 \oplus \overline{x_1}x_2)x_3 \\
&= ((1 \oplus x_1) \oplus x_2 \oplus (1 \oplus x_1)x_2)x_3 \\
&= (1 \oplus x_1 \oplus x_2 \oplus x_2 \oplus x_1x_2)x_3 \\
&= x_3 \oplus x_1x_3 \oplus x_1x_2x_3
\end{aligned}
$$

Reed-Muller Form: Transformationsmatrix

- lineare Umrechnung zwischen Funktion f, bzw. der Funktionstabelle (disjunktive Normalform), und RMF
- Transformationsmatrix A kann rekursiv definiert werden (wie die RMF-Basisfunktionen)
- Multiplikation von A mit f ergibt Koeffizientenvektor r der RMF

$$r = A \cdot f \quad \text{und} \quad f = A \cdot r$$

gilt wegen: $r = A \cdot f$ und $A \cdot A = I$, also $f = A \cdot r$!

$$A_0 = (1)$$

$$A_1 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

Reed-Muller Form: Transformationsmatrix (cont.)

$$A_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$A_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$\ldots$

$$A_n = \begin{pmatrix} A_{n-1} & 0 \\ A_{n-1} & A_{n-1} \end{pmatrix}$$

Reed-Muller Form: Beispiel

x_3	x_2	x_1	$f(x)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

- Berechnung durch Rechenregeln der Boole'schen Algebra oder Aufstellen von A_3 und Ausmultiplizieren: $f(x) = x_2 \oplus x_3 x_2 x_1$
- häufig kompaktere Darstellung als DNF oder KNF

Reed-Muller Form: Beispiel (cont.)

- $f(x_3, x_2, x_1) = \{0, 0, 1, 1, 0, 0, 1, 0\}$ (Funktionstabelle)
- Aufstellen von A_3 und Ausmultiplizieren

$$r = A_3 \cdot f = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Basisfunktionen: $\{1, x_1, x_2, x_2x_1, x_3, x_3x_1, x_3x_2, x_3x_2x_1\}$

führt zur gesuchten RMF:

$f(x_3, x_2, x_1) = r \cdot RM(3) = x_2 \oplus x_3x_2x_1$

Grafische Darstellung: Entscheidungsbäume

- Darstellung einer Schaltfunktion als Baum/Graph
- jeder Knoten ist einer Variablen zugeordnet

  jede Verzweigung entspricht einer `if-then-else`-Entscheidung

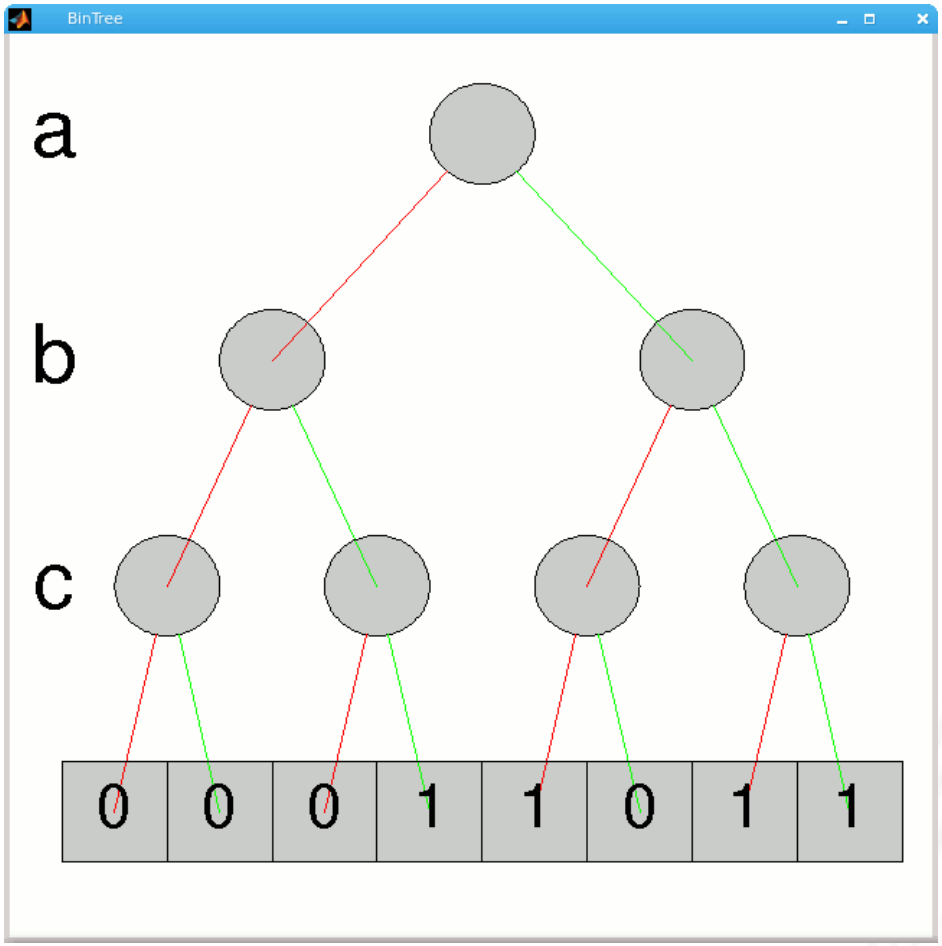
- vollständige Baum realisiert Funktionstabelle

\+ einfaches Entfernen/Zusammenfassen redundanter Knoten

- Beispiel: Multiplexer

  $f(a, b, c) = (a \wedge \overline{c}) \vee (b \wedge c)$

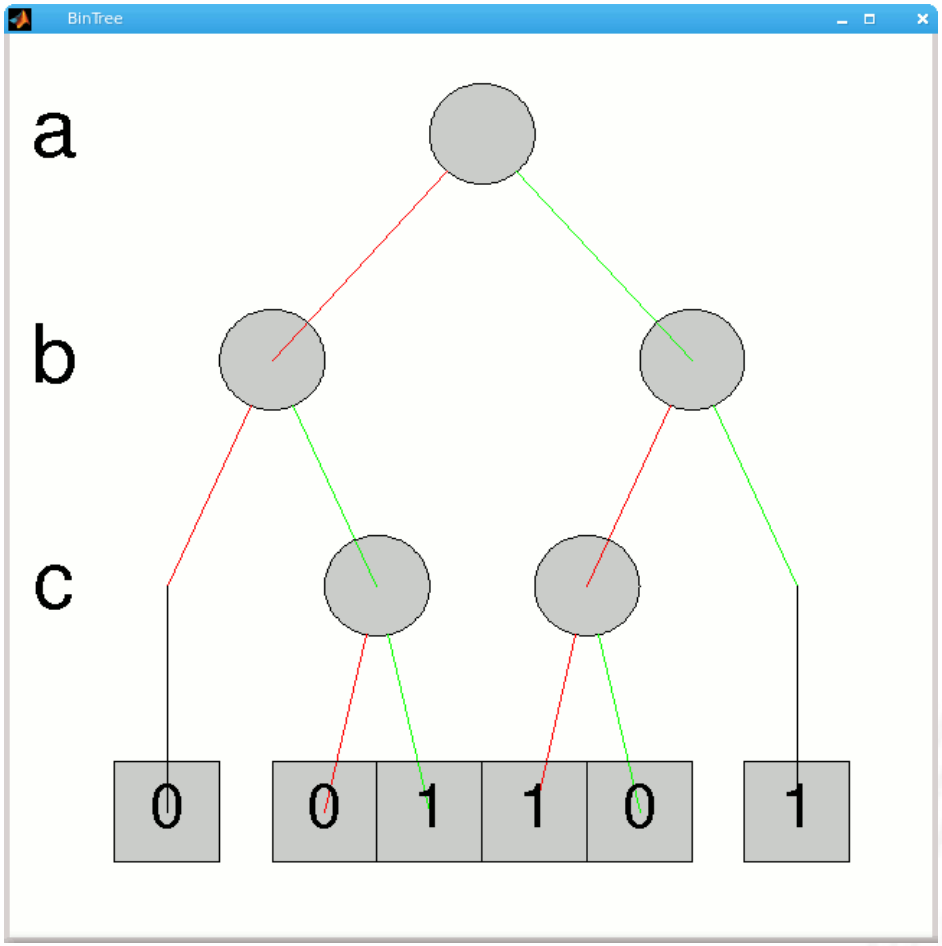
Entscheidungsbaum: Beispiel



- $f(a, b, c) = (a \wedge \overline{c}) \vee (b \wedge c)$

- rot: 0-Zweig
  grün: 1-Zweig

Entscheidungsbaum: Beispiel (cont.)



- $f(a, b, c) = (a \wedge \overline{c}) \vee (b \wedge c)$

$\Rightarrow$ Knoten entfernt

- rot: 0-Zweig
  grün: 1-Zweig

Reduced Ordered Binary-Decision Diagrams (ROBDD)

Binäres Entscheidungsdiagramm

- Variante des Entscheidungsbaums
- vorab gewählte Variablenordnung (*ordered*)
- redundante Knoten werden entfernt (*reduced*)
- ein ROBDD ist eine Normalform für eine Funktion

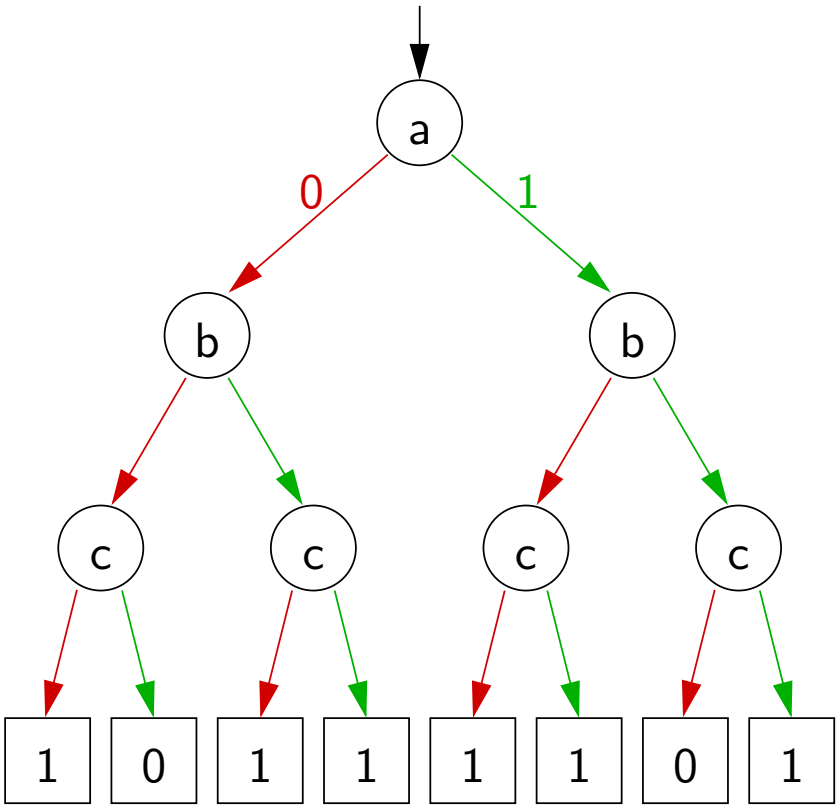
- viele praxisrelevante Funktionen sehr kompakt darstellbar
  $\mathcal{O}(n) \ldots \mathcal{O}(n^2)$ Knoten bei n Variablen
- wichtige Ausnahme: n-bit Multiplizierer ist $\mathcal{O}(2^n)$
- derzeit das Standardverfahren zur Manipulation von (großen) Schaltfunktionen

R. E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*, [Bry86]

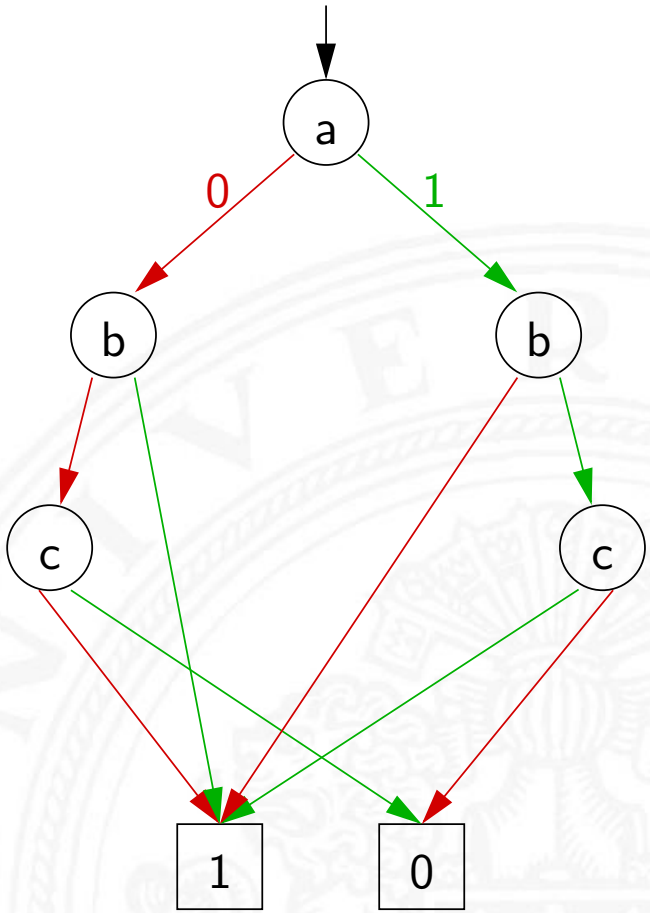
ROBDD vs. Entscheidungsbaum

Entscheidungsbaum

$f = (a\,b\,c) \vee (a\,\overline{b}) \vee (\overline{a}\,b) \vee (\overline{a}\,\overline{b}\,\overline{c})$

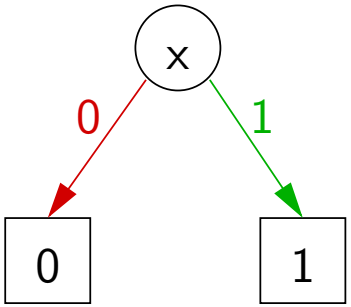


ROBDD

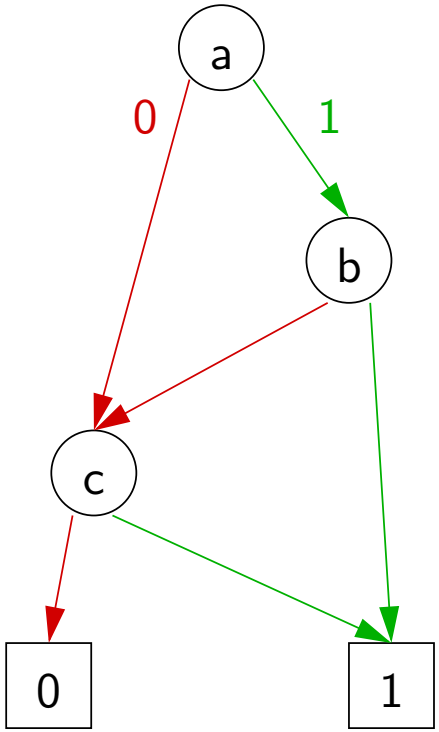


ROBDD: Beispiele

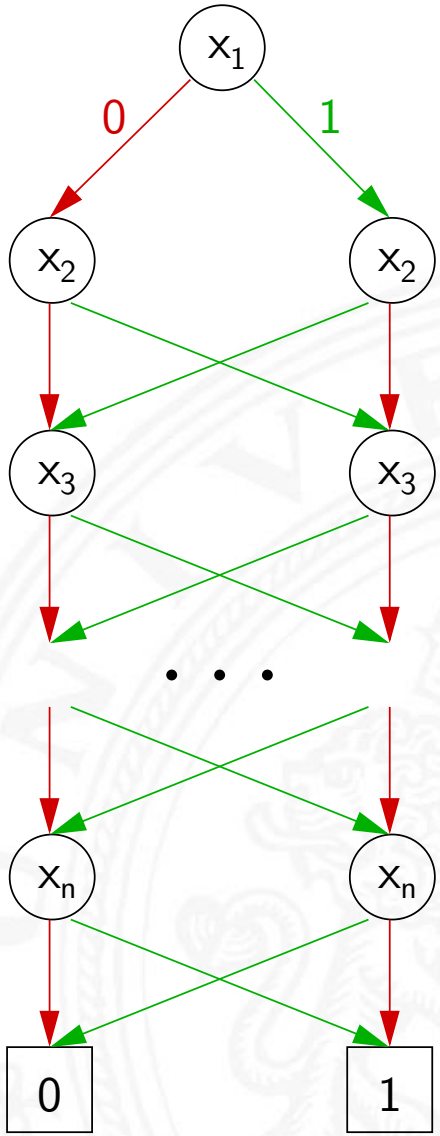
$f(x) = x$



$g = (a\,b) \vee c$



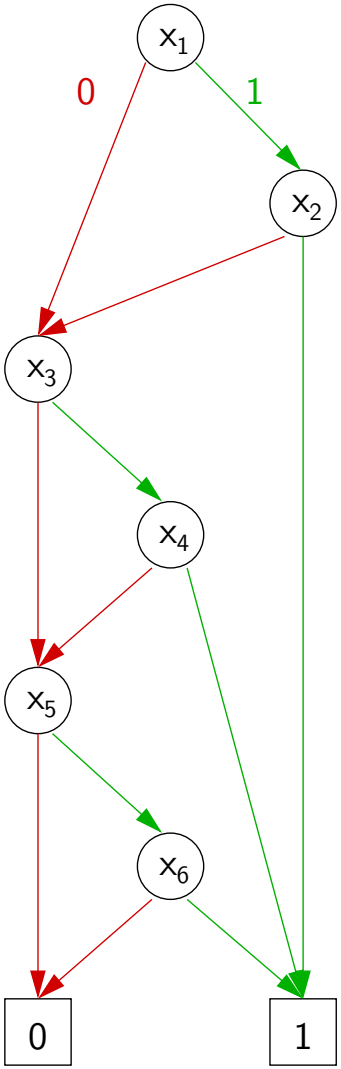
Parität $p = x_1 \oplus x_2 \oplus \ldots x_n$



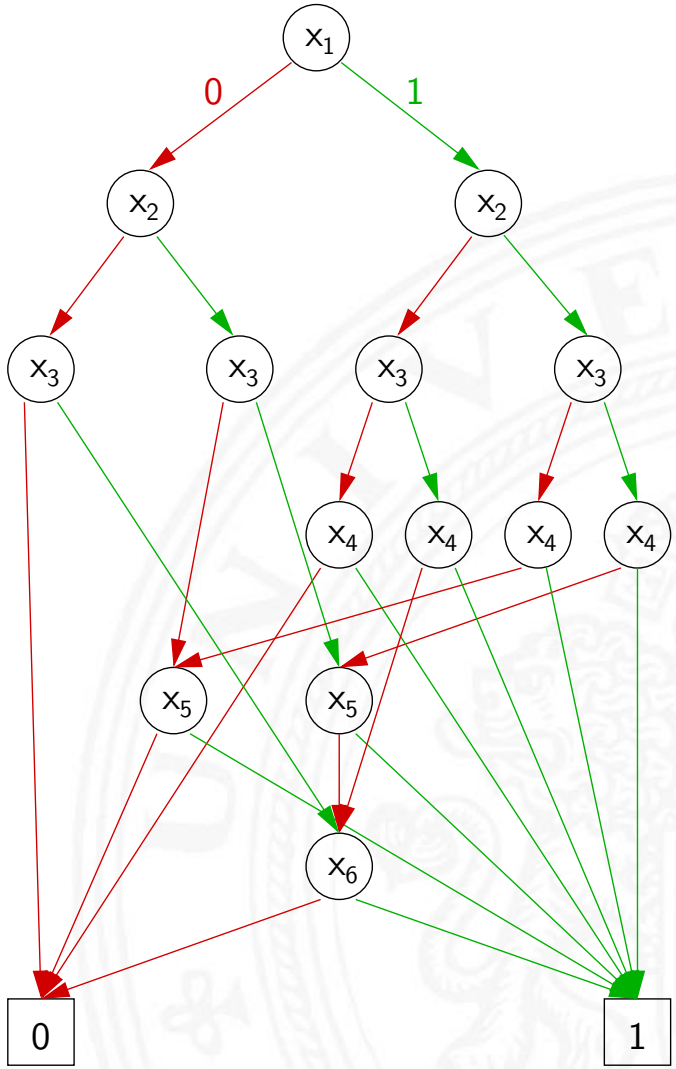
ROBDD: Problem der Variablenordnung

- Anzahl der Knoten oft stark abhängig von der Variablenordnung

$f = x_1\,x_2 \vee x_3\,x_4 \vee x_5\,x_6$



$g = x_1\,x_4 \vee x_2\,x_5 \vee x_3\,x_6$



Minimierung von Schaltfunktionen

- mehrere (beliebig viele) Varianten zur Realisierung einer gegebenen Schaltfunktion bzw. eines Schaltnetzes

Minimierung des Realisierungsaufwandes:

- diverse Kriterien, technologieabhängig

Hardwarekosten	Anzahl der Gatter
Hardwareeffizienz	z.B. NAND statt XOR
Geschwindigkeit	Anzahl der Stufen, Laufzeiten
Testbarkeit	Erkennung von Produktionsfehlern
Robustheit	z.B. ionisierende Strahlung

Algebraische Minimierungsverfahren

- Vereinfachung der gegebenen Schaltfunktionen durch Anwendung der Gesetze der Boole'schen Algebra

- im Allgemeinen nur durch Ausprobieren
- ohne Rechner sehr mühsam
- keine allgemeingültigen Algorithmen bekannt

- Heuristische Verfahren
  - Suche nach *Primimplikanten* (= kürzeste Konjunktionsterme)
  - Quine-McCluskey-Verfahren und Erweiterungen

Algebraische Minimierung: Beispiel

- Ausgangsfunktion in DNF

$$y(x) = \overline{x_3}\, x_2\, x_1\, \overline{x_0} \vee \overline{x_3}\, x_2\, x_1\, x_0 \vee x_3\, \overline{x_2}\, \overline{x_1}\, x_0 \vee x_3\, \overline{x_2}\, x_1\, \overline{x_0} \vee x_3\, \overline{x_2}\, x_1\, x_0 \vee x_3\, x_2\, \overline{x_1}\, x_0 \vee x_3\, x_2\, x_1\, \overline{x_0} \vee x_3\, x_2\, x_1\, x_0$$

- Zusammenfassen benachbarter Terme liefert

$$y(x) = \overline{x_3}\, x_2\, x_1 \vee x_3\, \overline{x_2}\, x_0 \vee x_3\, \overline{x_2}\, x_1 \vee x_3\, x_2\, x_0 \vee x_3\, x_2\, x_1$$

- aber bessere Lösung ist möglich (weiter Umformen)

$$y(x) = x_2\, x_1 \vee x_3\, x_0 \vee x_3\, x_1$$

Grafische Minimierungsverfahren

- Darstellung einer Schaltfunktion im KV-Diagramm
- Interpretation als disjunktive Normalform (konjunktive NF)

- Zusammenfassen benachbarter Terme durch **Schleifen**
- alle 1-Terme mit möglichst wenigen Schleifen abdecken
  (alle 0-Terme –"– $\equiv$ konjunktive Normalform)
- Ablesen der minimierten Funktion, wenn keine weiteren Schleifen gebildet werden können

- beruht auf der menschlichen Fähigkeit, benachbarte Flächen auf einen Blick zu „sehen“
- bei mehr als 6 Variablen nicht mehr praktikabel

Erinnerung: Karnaugh-Veitch Diagramm

$x_3\,x_2$ \ $x_1\,x_0$	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

$x_3\,x_2$ \ $x_1\,x_0$	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

- 2D-Diagramm mit $2^n = 2^{n_y} \times 2^{n_x}$ Feldern
- gängige Größen sind: 2×2, 2×4, 4×4
  darüber hinaus: mehrere Diagramme der Größe 4×4
- Anordnung der Indizes ist im einschrittigen-Code / Gray-Code

$\Rightarrow$ benachbarte Felder unterscheiden sich gerade um 1 Bit

KV-Diagramme: 2...4 Variable (2×2, 2×4, 4×4)

x_1 \ x_0	0	1
0	00	01
1	10	11

x_2 \ $x_1 x_0$	00	01	11	10
0	000	001	011	010
1	100	101	111	110

$x_3 x_2$ \ $x_1 x_0$	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

KV-Diagramm für Schaltfunktionen

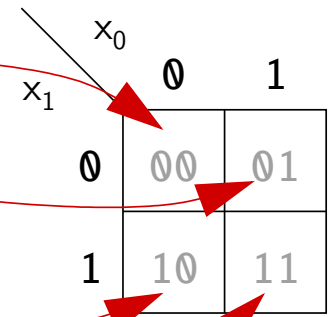
- Funktionswerte in zugehöriges Feld im KV-Diagramm eintragen
- Werte `0` und `1`
  Don't-Care „$*$“ für nicht spezifizierte Werte wichtig!
- 2D-Äquivalent zur Funktionstabelle

- praktikabel für 3 . . . 6 Eingänge
- fünf Eingänge: zwei Diagramme à 4×4 Felder
  sechs Eingänge: vier Diagramme à 4×4 Felder

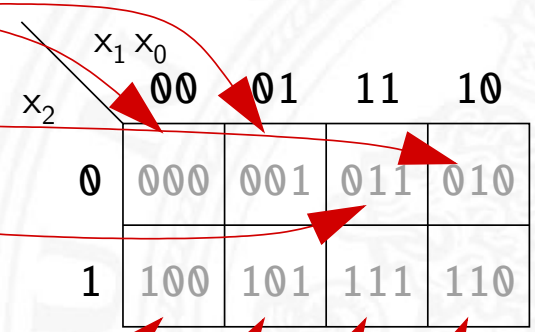
- viele Strukturen „auf einen Blick“ erkennbar

KV-Diagramm: Zuordnung zur Funktionstabelle

x_1	x_0	y=f(x)
0	0	f(0 0)
0	1	f(0 1)
1	0	f(1 0)
1	1	f(1 1)

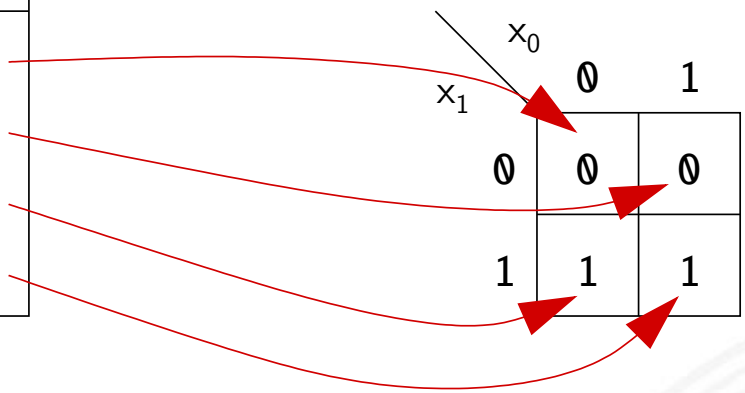


x_2	x_1	x_0	y=f(x)
0	0	0	f(0 0 0)
0	0	1	f(0 0 1)
0	1	0	f(0 1 0)
0	1	1	f(0 1 1)
1	0	0	f(1 0 0)
1	0	1	f(1 0 1)
1	1	0	f(1 1 0)
1	1	1	f(1 1 1)

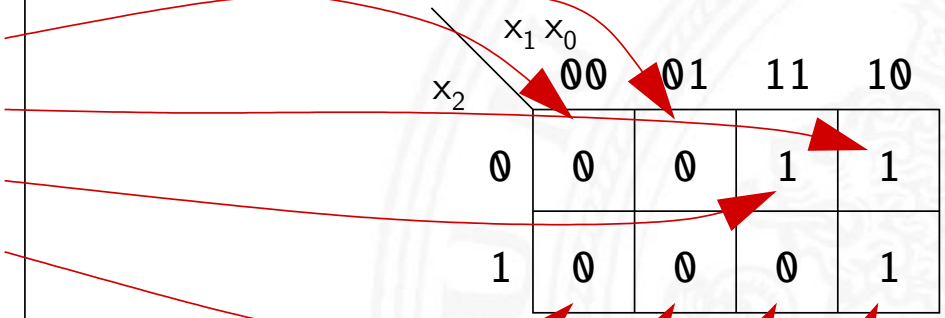


KV-Diagramm: Eintragen aus Funktionstabelle

x_1	x_0	y=f(x)
0	0	0
0	1	0
1	0	1
1	1	1



x_2	x_1	x_0	y=f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0



KV-Diagramm: Beispiel

$x_3\,x_2$ \ $x_1\,x_0$	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

$x_3\,x_2$ \ $x_1\,x_0$	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	0

- Beispielfunktion in DNF mit vier Termen:
  $f(x) = (\overline{x_3}\,\overline{x_2}\,\overline{x_1}\,\overline{x_0}) \vee (\overline{x_3}\,\overline{x_2}\,x_1\,\overline{x_0}) \vee (x_3\,\overline{x_2}\,x_1\,x_0) \vee (x_3\,x_2\,x_1\,x_0)$
- Werte aus Funktionstabelle an entsprechender Stelle ins Diagramm eintragen

Schleifen: Zusammenfassen benachbarter Terme

- benachbarte Felder unterscheiden sich um 1-Bit
- falls benachbarte Terme beide 1 sind $\Rightarrow$ Funktion hängt an dieser Stelle nicht von der betroffenen Variable ab
- zugehörige (Min-) Terme können zusammengefasst werden

- Erweiterung auf vier benachbarte Felder (4x1 1x4 2x2)
  –"– auf acht –"– (4x2 2x4) usw.
- aber keine Dreier- Fünfergruppen usw. (Gruppengröße 2^i)

- Nachbarschaft auch „außen herum“
- mehrere Schleifen dürfen sich überlappen

Schleifen: Ablesen der Schleifen

x_1	x_0	y=f(x)
0	0	0
0	1	0
1	0	1
1	1	1

$$f(x_1, x_0) = x_1$$

x_1 \ x_0	0	1
0	0	0
1	1	1

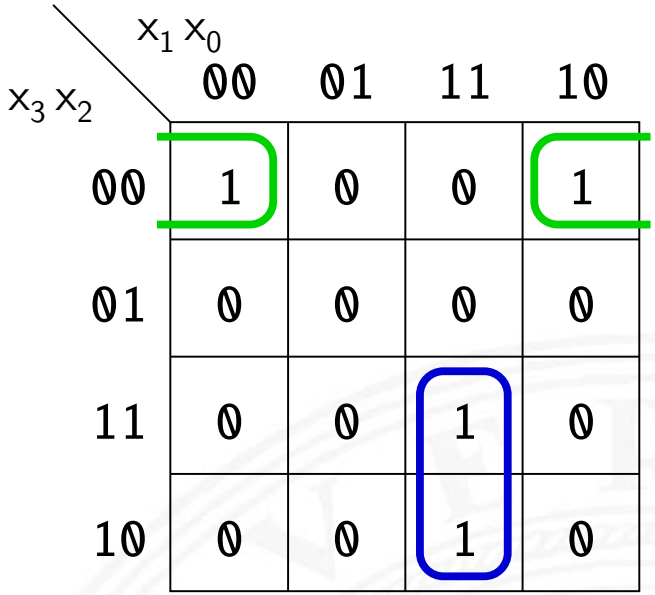
x_2	x_1	x_0	y=f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

$$f(x_2, x_1, x_0) = \overline{x_2}\, x_1 \vee x_1\, \overline{x_0}$$

x_2 \ $x_1 x_0$	00	01	11	10
0	0	0	1	1
1	0	0	0	1

Schleifen: Ablesen der Schleifen (cont.)

$x_3\,x_2$ \ $x_1\,x_0$	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	0



- insgesamt zwei Schleifen möglich
- grün entspricht $(\overline{x_3}\,\overline{x_2}\,\overline{x_0}) = (\overline{x_3}\,\overline{x_2}\,\overline{x_1}\,\overline{x_0}) \vee (\overline{x_3}\,\overline{x_2}\,x_1\,\overline{x_0})$

  blau entspricht $(x_3\,x_1\,x_0) = (x_3\,x_2\,x_1\,x_0) \vee (x_3\,\overline{x_2}\,x_1\,x_0)$
- minimierte disjunktive Form $f(x) = (\overline{x_3}\,\overline{x_2}\,\overline{x_0}) \vee (x_3\,x_1\,x_0)$

Schleifen: interaktive Demonstration

- Minimierung mit KV-Diagrammen [Kor16]

  `tams.informatik.uni-hamburg.de/research/software/tams-tools/kvd-editor.html`

  - Auswahl der Funktionalität: *Edit function*, *Edit loops*
  - Explizite Eingabe: *Open Diagram - From Expressions* Tipp!

  1 Funktion: Maustaste ändert Werte

  2 Schleifen: Auswahl und Aufziehen mit Maustaste

  - Anzeige des zugehörigen Hardwareaufwands und der Schaltung

- Applet zur Minimierung mit KV-Diagrammen [HenKV]

  `tams.informatik.uni-hamburg.de/applets/kvd`

  - Auswahl der Funktionalität: *Edit function*, *Add loop* ...
  - Ändern der Ein-/Ausgänge: *File - Examples - User define dialog*

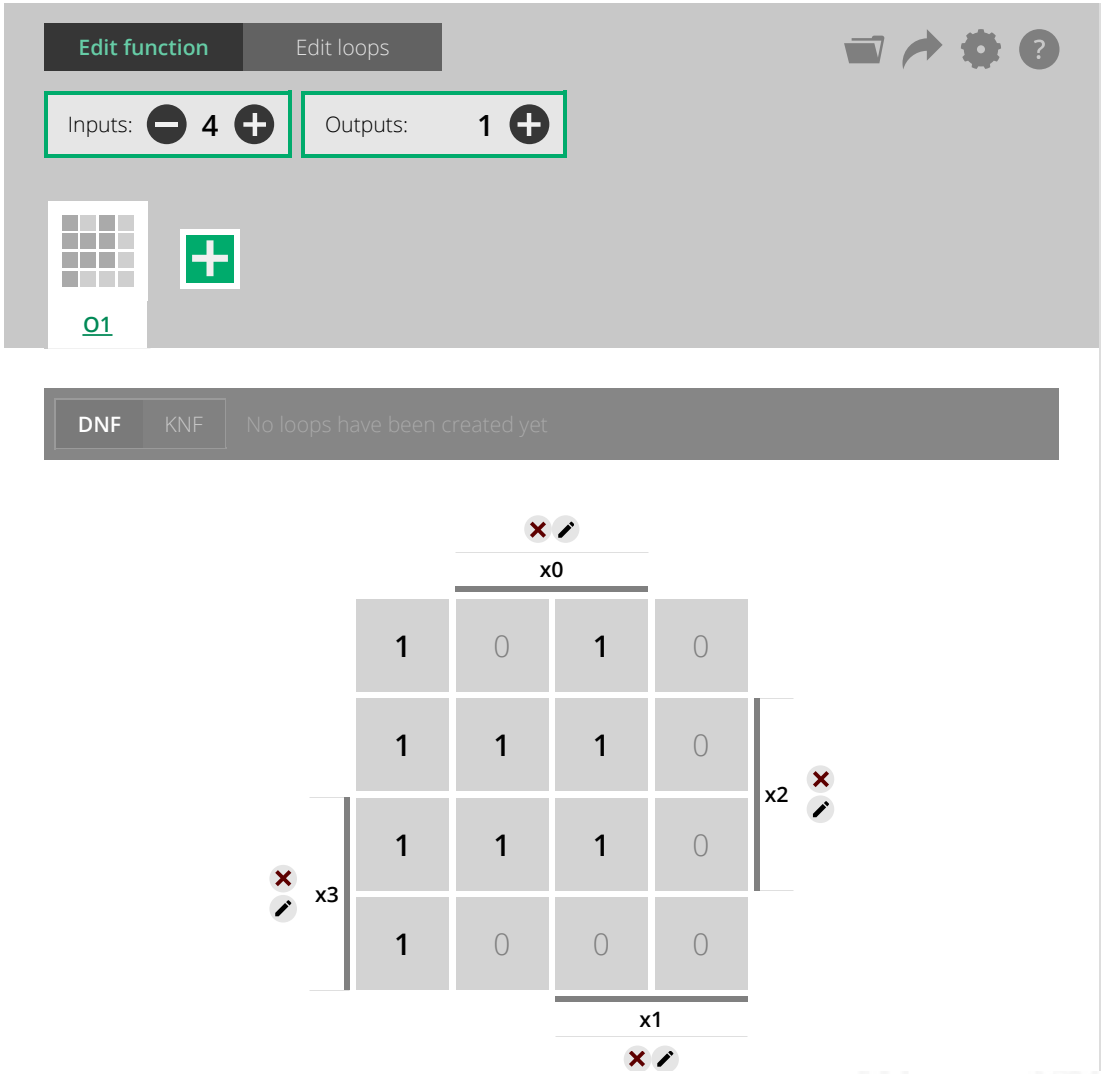
  1 Funktion: Maustaste ändert Werte

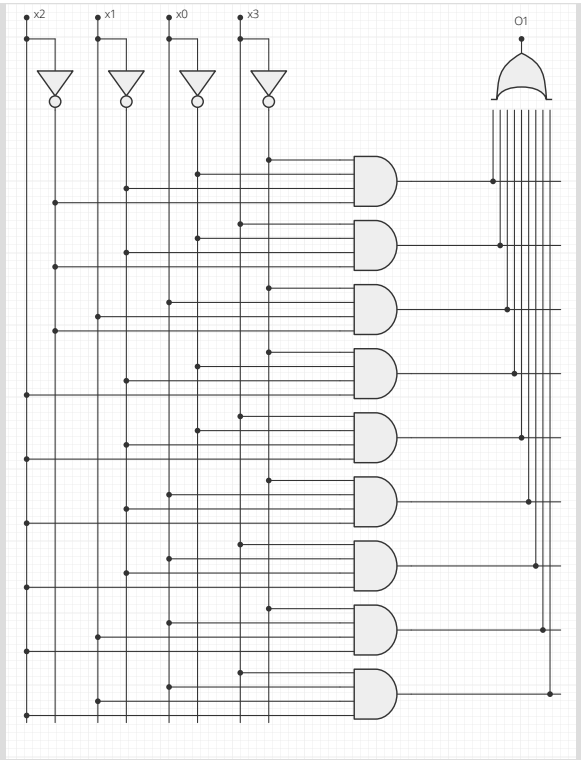
  2 Schleifen: Maustaste, *shift*+Maus, *ctrl*+Maus

  - Anzeige des zugehörigen Hardwareaufwands und der Schaltung
  - Achtung: andere Anordnung der Eingangsvariablen als im Skript

    $\Rightarrow$ andere Anordnung der Terme im KV-Diagramm

KV-Diagramm Editor: Screenshots

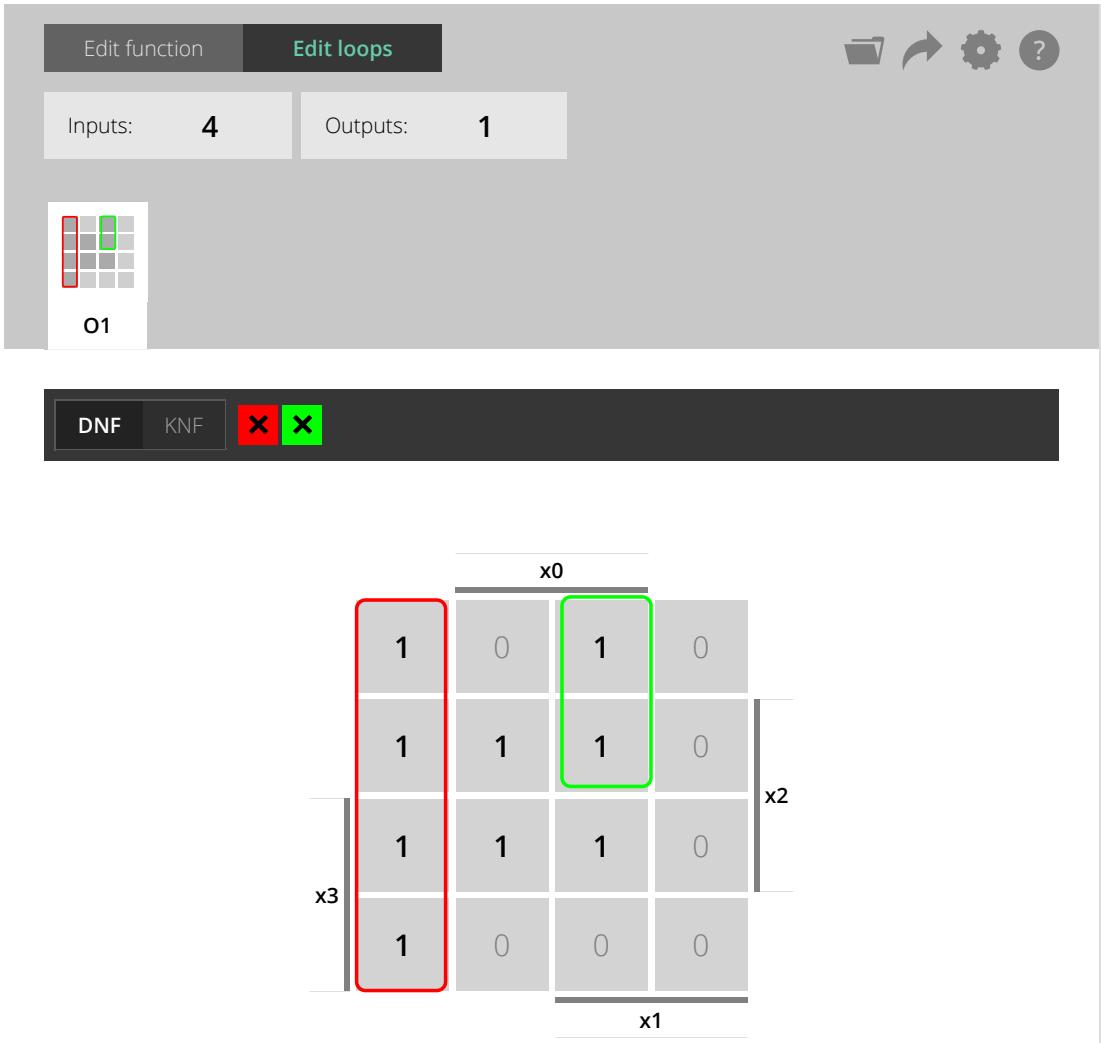


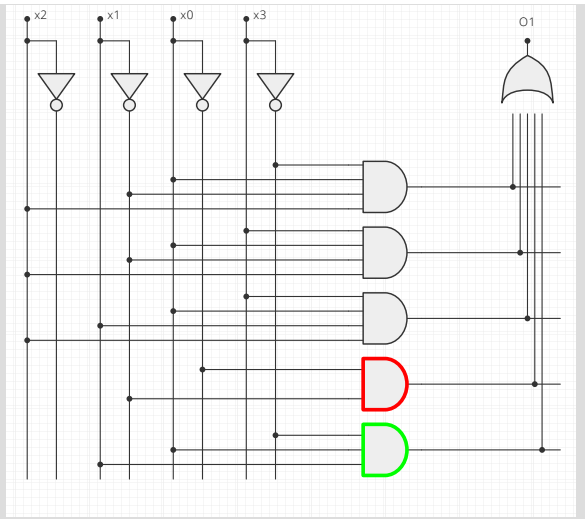


Costs: 10 gates with 45 inputs

Eingabe der Schaltfunktion

KV-Diagramm Editor: Screenshots (cont.)

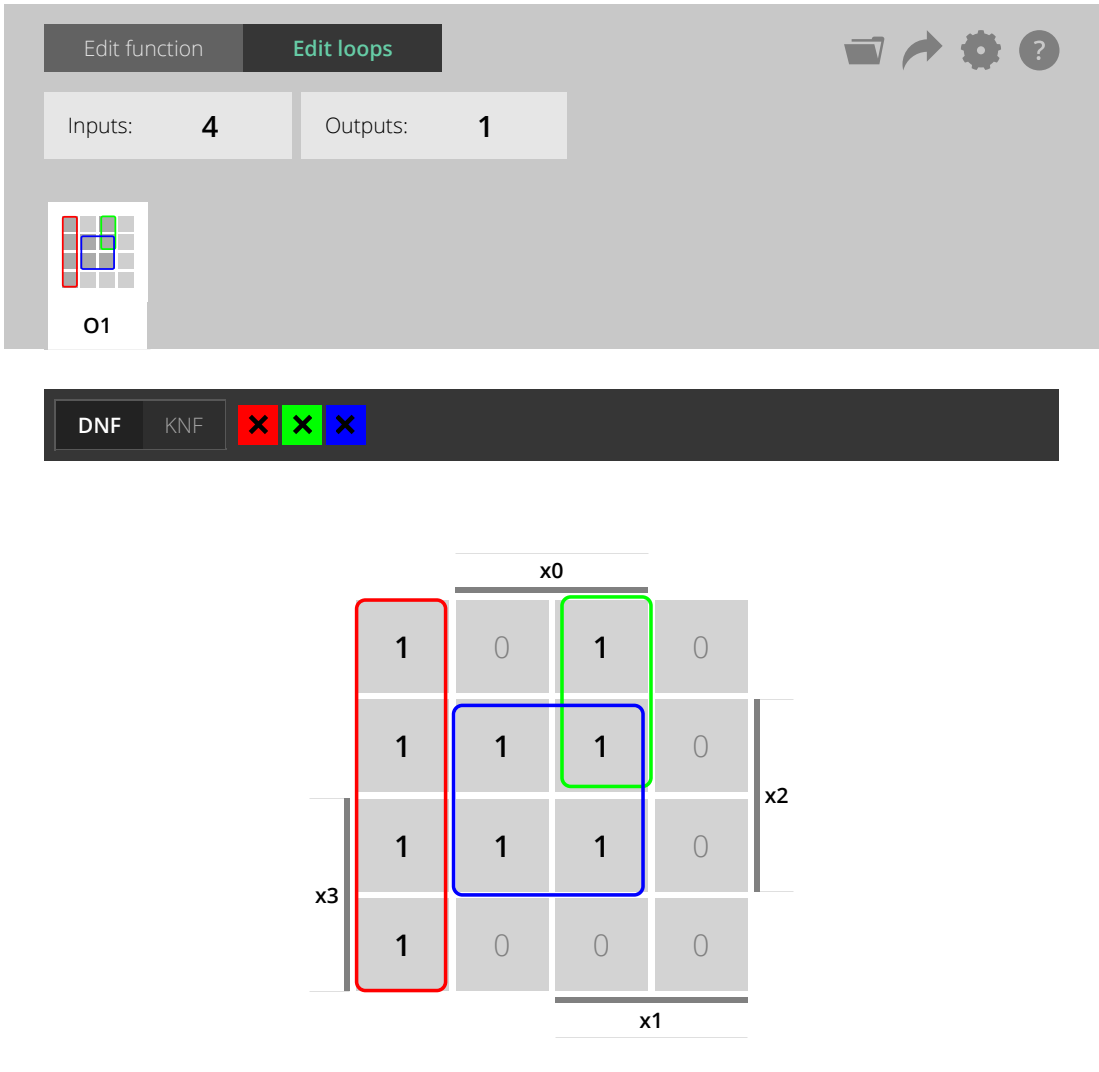


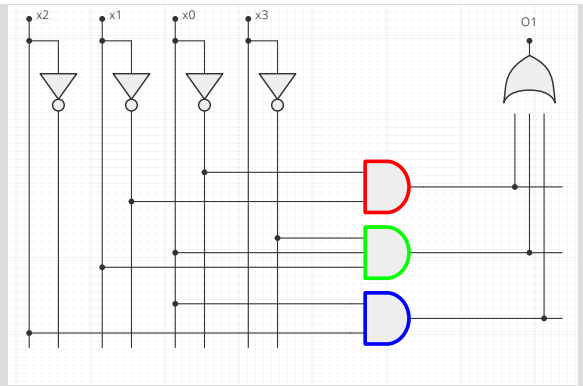


Costs: 6 gates with 22 inputs

Minimierung durch Schleifenbildung

KV-Diagramm Editor: Screenshots (cont.)

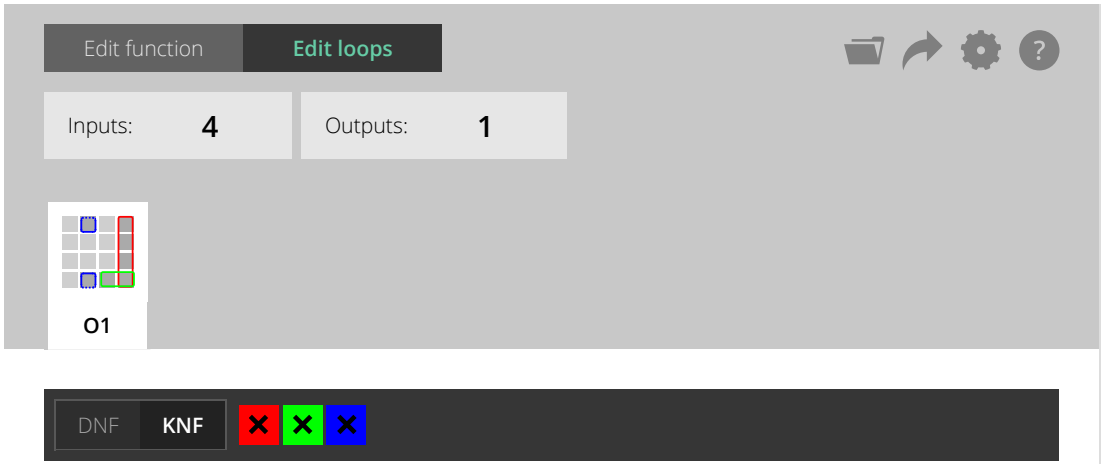


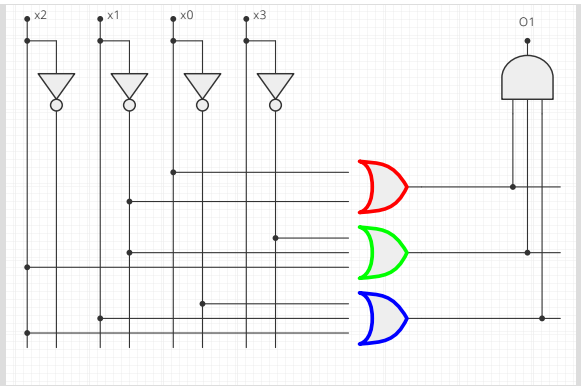


Costs: 4 gates with 10 inputs

- Hardware-Kosten: # Gatter, Eingänge

KV-Diagramm Editor: Screenshots (cont.)





Costs: 4 gates with 11 inputs

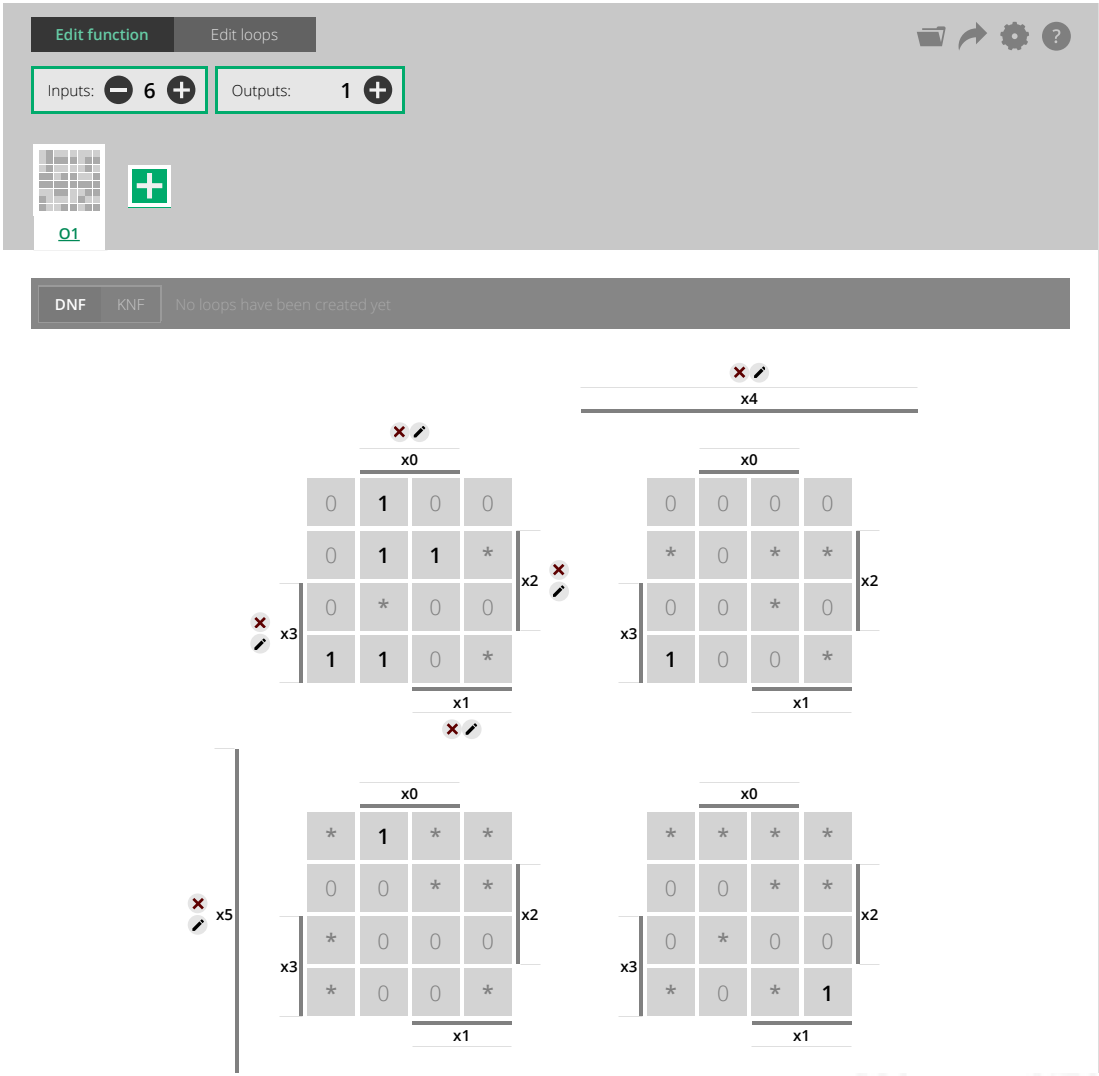
Konjunktive Form

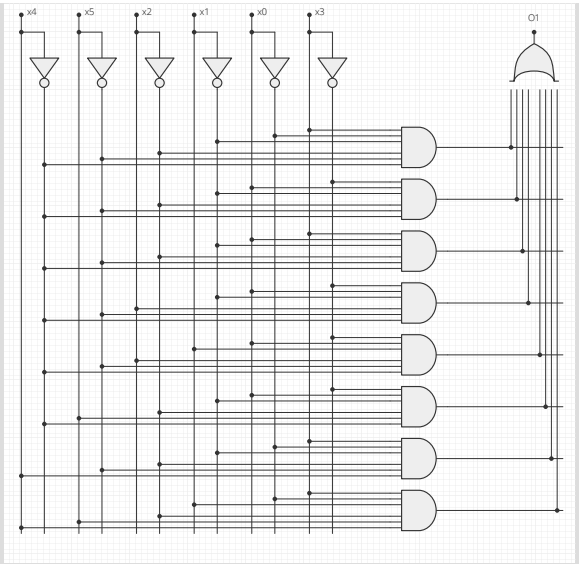
Don't-Care Terme

- in der Praxis: viele Schaltfunktionen unvollständig definiert
  weil bestimmte Eingangskombinationen nicht vorkommen
- zugehörige Terme als ***Don't-Care*** markieren
  typisch: Sternchen „$*$“ in Funktionstabelle/KV-Diagramm

- solche Terme bei Minimierung nach Wunsch auf 0/1 setzen
- Schleifen dürfen *Don't-Cares* enthalten
- Schleifen möglichst groß

KV-Diagramm Editor: 6 Variablen, *Don't-Cares*





Costs: 9 gates with 56 inputs

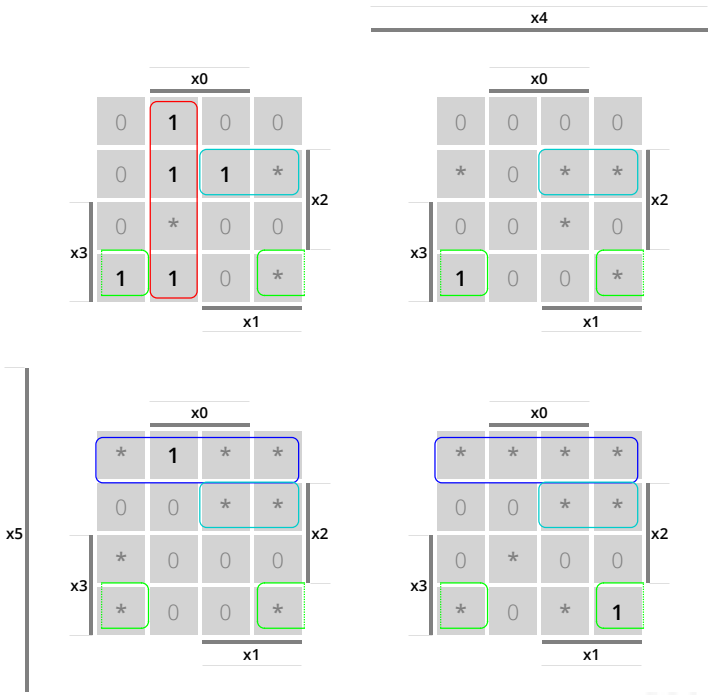
KV-Diagramm Editor: 6 Variablen, *Don't-Cares* (cont.)

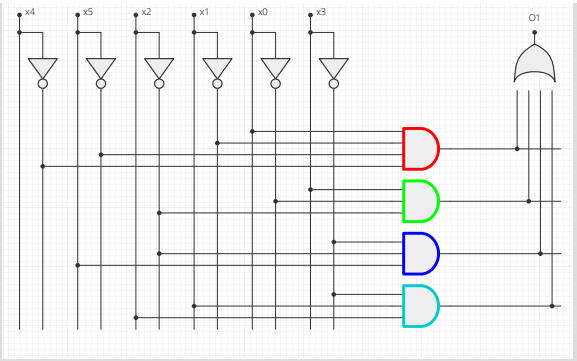
Edit function | Edit loops

Inputs: 6 Outputs: 1

O1

DNF KNF





Costs: 5 gates with 17 inputs

Quine-McCluskey-Algorithmus

- Algorithmus zur Minimierung einer Schaltfunktion
- Notation der Terme in Tabellen, n Variablen
- Prinzip entspricht der Minimierung im KV-Diagramm
  aber auch geeignet für mehr als sechs Variablen
- Grundlage gängiger Minimierungsprogramme

- Sortieren der Terme nach Hamming-Abstand
- Erkennen der unverzichtbaren Terme („Primimplikanten“)
- Aufstellen von Gruppen benachbarter Terme (mit Distanz 1)
- Zusammenfassen geeigneter benachbarter Terme

Becker, Molitor: *Technische Informatik – eine einführende Darstellung* [BM08]

Schiffmann, Schmitz: *Technische Informatik I* [SS04]

Literatur

[BM08] B. Becker, P. Molitor: *Technische Informatik – eine einführende Darstellung*. 2. Auflage, Oldenbourg, 2008. ISBN 978–3–486–58650–3

[SS04] W. Schiffmann, R. Schmitz: *Technische Informatik 1 – Grundlagen der digitalen Elektronik*. 5. Auflage, Springer-Verlag, 2004. ISBN 978–3–540–40418–7

[WH03] H.D. Wuttke, K. Henke: *Schaltsysteme – Eine automatenorientierte Einführung*. Pearson Studium, 2003. ISBN 978–3–8273–7035–8

[Bry86] R.E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*. in: *IEEE Trans. Computers* 35 (1986), Nr. 8, S. 677–691

Literatur (cont.)

[Kor16] Laszlo Korte: *TAMS Tools for eLearning*.
Universität Hamburg, FB Informatik, 2016, BSc Thesis. `tams.informatik.uni-hamburg.de/research/software/tams-tools`

[HenKV] N. Hendrich: *KV-Diagram Simulation*.
Universität Hamburg, FB Informatik, Lehrmaterial.
`tams.informatik.uni-hamburg.de/applets/kvd`

[Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript*. Universität Hamburg, FB Informatik, 2005.
`tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1`

Gliederung

Gliederung (cont.)

Gliederung (cont.)

13. Assembler-Programmierung
14. Rechnerarchitektur II
15. Betriebssysteme

Schaltnetze: Definition

- **Schaltnetz** oder auch

  kombinatorische Schaltung (*combinational logic circuit*):

  ein digitales System mit n Eingängen $(b_1, b_2, \ldots, b_n)$ und m-Ausgängen $(y_1, y_2, \ldots, y_m)$, dessen Ausgangsvariablen zu jedem Zeitpunkt nur von den aktuellen Werten der Eingangsvariablen abhängen

  Beschreibung als Vektorfunktion $\vec{y} = F(\vec{b})$

- Bündel von Schaltfunktionen (mehrere SF)
- ein Schaltnetz darf keine Rückkopplungen enthalten

Schaltnetze: Definition (cont.)

- Begriff: „Schaltnetz“
  - technische Realisierung von Schaltfunktionen / Funktionsbündeln
  - Struktur aus einfachen Gatterfunktionen:
    triviale Funktionen mit wenigen (2 . . . 4) Eingängen
- in der Praxis können Schaltnetze nicht statisch betrachtet werden: Gatterlaufzeiten spielen eine Rolle

Elementare digitale Schaltungen

- Schaltsymbole
- Grundgatter (Inverter, AND, OR usw.)
- Kombinationen aus mehreren Gattern

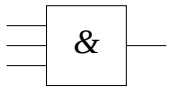
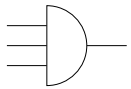
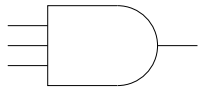
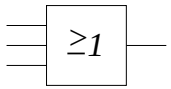
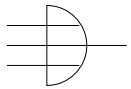
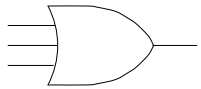
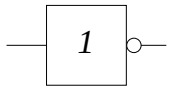
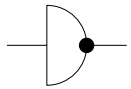
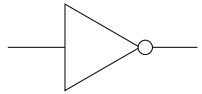
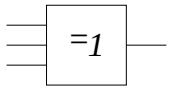
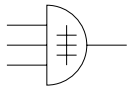
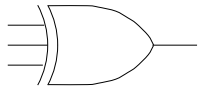
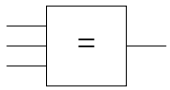
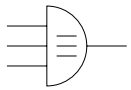
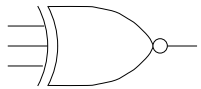
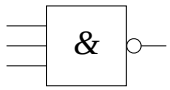
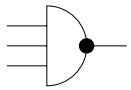
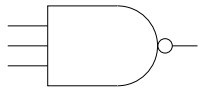
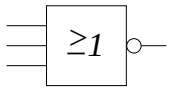
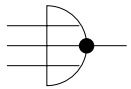
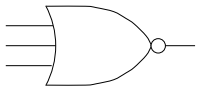
- Schaltnetze (mehrere Ausgänge)
- Beispiele

- Arithmetisch/Logische Operationen

Schaltpläne (*schematics*)

- standardisierte Methode zur Darstellung von Schaltungen
- genormte Symbole für Komponenten
  - Spannungs- und Stromquellen, Messgeräte
  - Schalter und Relais
  - Widerstände, Kondensatoren, Spulen
  - Dioden, Transistoren (bipolar, MOS)
  - **Gatter**: logische Grundoperationen (UND, ODER usw.)
  - **Flipflops**: Speicherglieder
- Verbindungen
  - Linien für Drähte (Verbindungen)
  - Anschlusspunkte für Drahtverbindungen
  - dicke Linien für *n*-bit Busse, Anzapfungen usw.
- komplexe Bausteine, hierarchisch zusammengesetzt

Schaltsymbole

Schaltzeichen DIN 40700 (ab 1976)	Früher	in USA	Benennung
			UND - Glied (AND)
			ODER - Glied (OR)
			NICHT - Glied (NOT)
			Exklusiv-Oder - Glied (Exclusive-OR, XOR)
			Aquivalenz - Glied (Logic identity)
			UND - Glied mit negiertem Ausgang (NAND)
			ODER - Glied mit negiertem Ausgang (NOR)
			Negation eines Eingangs
			Negation eines Ausgangs

Schiffmann, Schmitz:
Technische Informatik 1 [SS04]

Logische Gatter

- **Logisches Gatter** (*logic gate*): die Bezeichnung für die Realisierung einer logischen Grundfunktion als gekapselte Komponente (in einer gegebenen Technologie)

- 1 Eingang: Treiberstufe/Verstärker und Inverter (Negation)
- 2 Eingänge: AND/OR, NAND/NOR, XOR, XNOR
- 3 und mehr Eingänge: AND/OR, NAND/NOR, Parität
- Multiplexer

- vollständige Basismenge erforderlich (mindestens 1 Gatter)
- in Halbleitertechnologie sind NAND/NOR besonders effizient

Schaltplan-Editor und -Simulator

Spielerischer Zugang zu digitalen Schaltungen:

- mit Experimentierkasten oder im Logiksimulator
- interaktive Simulation erlaubt direktes Ausprobieren
- Animation und Visualisierung der logischen Werte
- „entdeckendes Lernen“

- Diglog: `john-lazzaro.github.io/chipmunk` [Laz]
- Hades: [HenHA]
  `tams.informatik.uni-hamburg.de/applets/hades/webdemos`
  `tams.informatik.uni-hamburg.de/applets/hades/webdemos/toc.html`
  - Demos laufen im Browser (Java erforderlich)
  - Grundschaltungen, Gate-Level Circuits ...
    einfache Prozessoren ...

Hades: Grundkomponenten

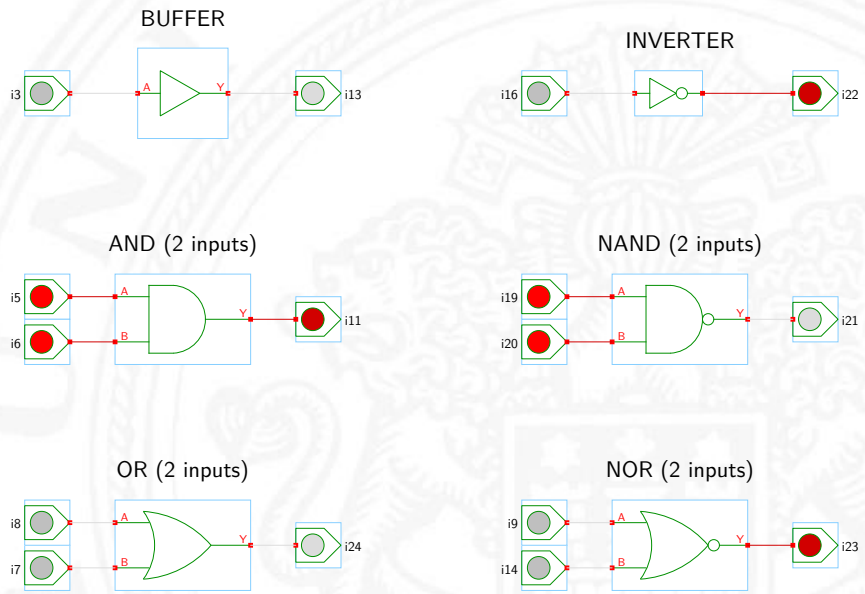
- Vorführung des Simulators

  Hades Demo: `00-intro/00-welcome/chapter`

- Eingang: Schalter + Anzeige („*Ipin*“)
- Ausgang: Anzeige („*Opin*“)
- Taktgenerator
- PowerOnReset
- Anzeige / Leuchtdiode
- Siebensegmentanzeige

...

[HenHA] Hades Demo: `10-gates/00-gates/basic`



Hades: *glow-mode* Visualisierung

- Farbe einer Leitung codiert den logischen Wert
- Einstellungen sind vom Benutzer konfigurierbar

- Defaultwerte

blau	glow-mode ausgeschaltet	
hellgrau	logisch	0
rot	logisch	1
orange	tri-state	Z $\Rightarrow$ kein Treiber (bidirektionale Busse)
magenta	undefined	X $\Rightarrow$ Kurzschluss, ungültiger Wert
cyan	unknown	U $\Rightarrow$ nicht initialisiert

Hades: Bedienung

- Menü: Anzeigeoptionen, Edit-Befehle usw.

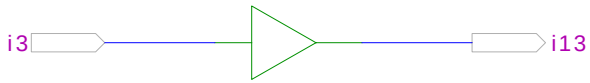
- Editorfenster mit Popup-Menü für häufige Aktionen
- Rechtsklick auf Komponenten öffnet Eigenschaften/Parameter (*property-sheets*)
- optional „tooltips“ (enable im Layer-Menü)

- Simulationssteuerung: *run*, *pause*, *rewind*
- Anzeige der aktuellen Simulationszeit

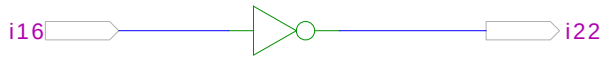
- Details siehe Hades-Webseite: Kurzreferenz, Tutorial
  `tams.informatik.uni-hamburg.de/applets/hades/webdemos/docs.html`

Gatter: Verstärker, Inverter, AND, OR

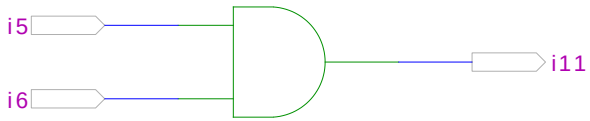
BUFFER



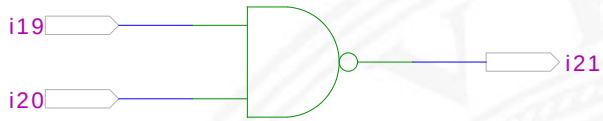
INVERTER



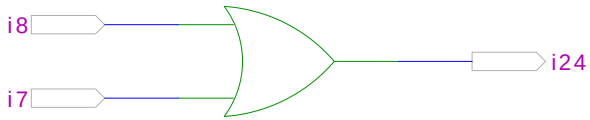
AND (2 inputs)



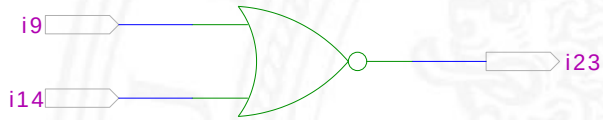
NAND (2 inputs)



OR (2 inputs)

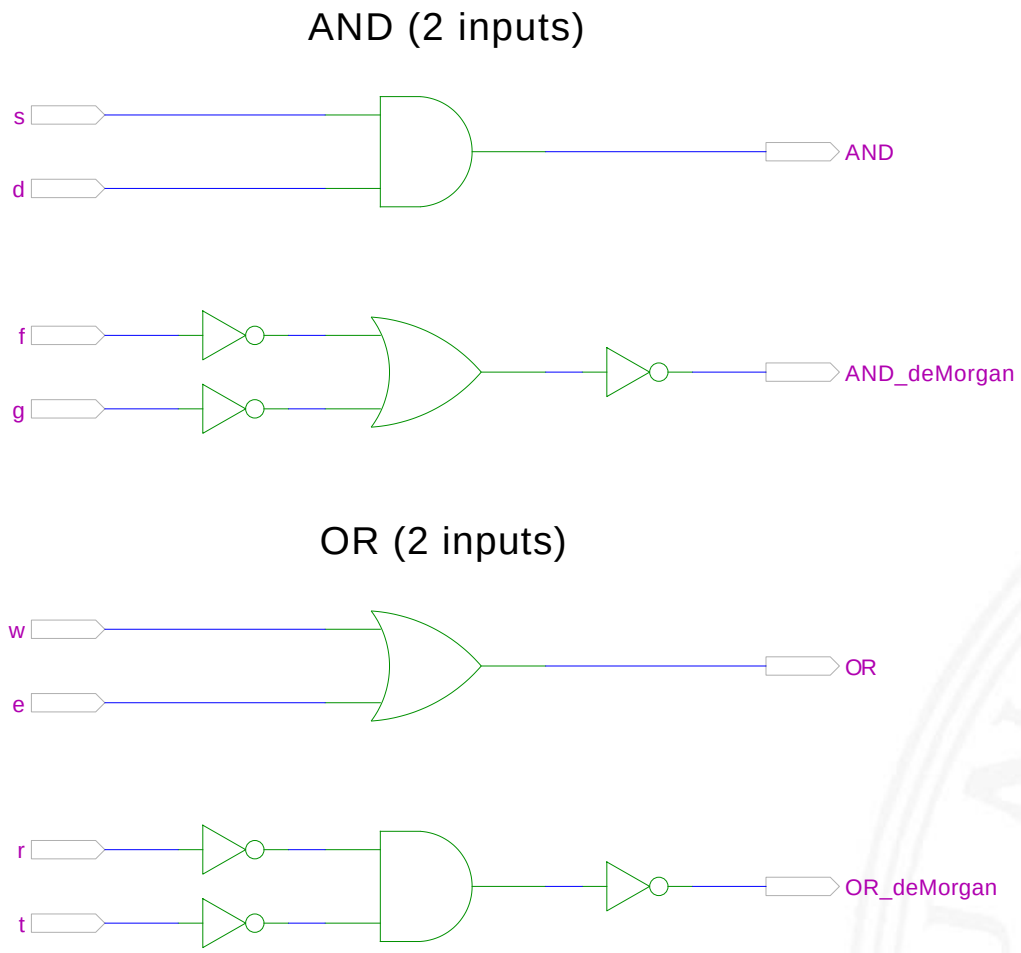


NOR (2 inputs)



[HenHA] Hades Demo: `10-gates/00-gates/basic`

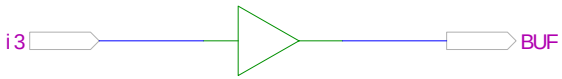
Grundschaltungen: De Morgan Regel



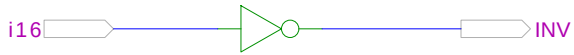
[HenHA] Hades Demo: `10-gates/00-gates/de-morgan`

Gatter: AND/NAND mit zwei, drei, vier Eingängen

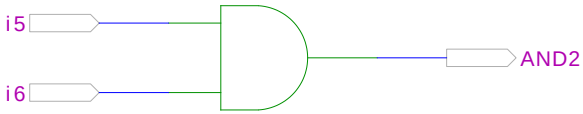
BUFFER



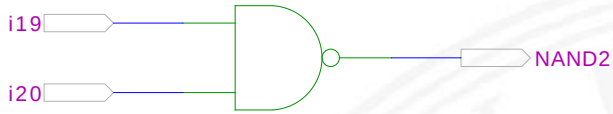
INVERTER



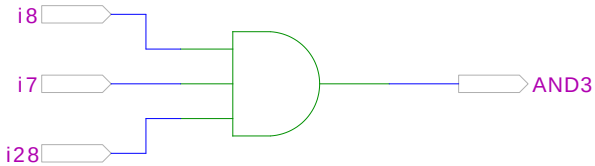
AND (2 inputs)



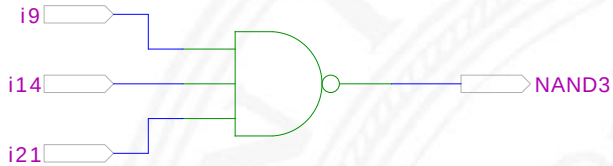
NAND (2 inputs)



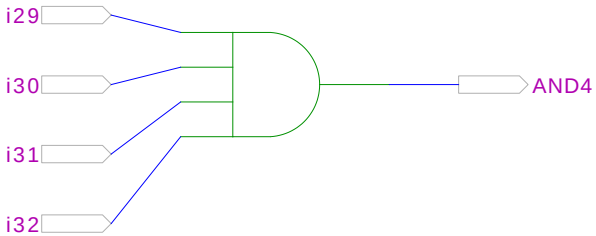
AND (3 inputs)



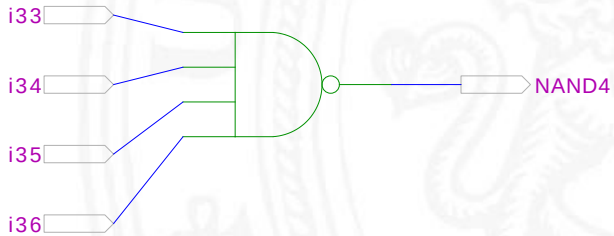
NAND (3 inputs)



AND (4 inputs)

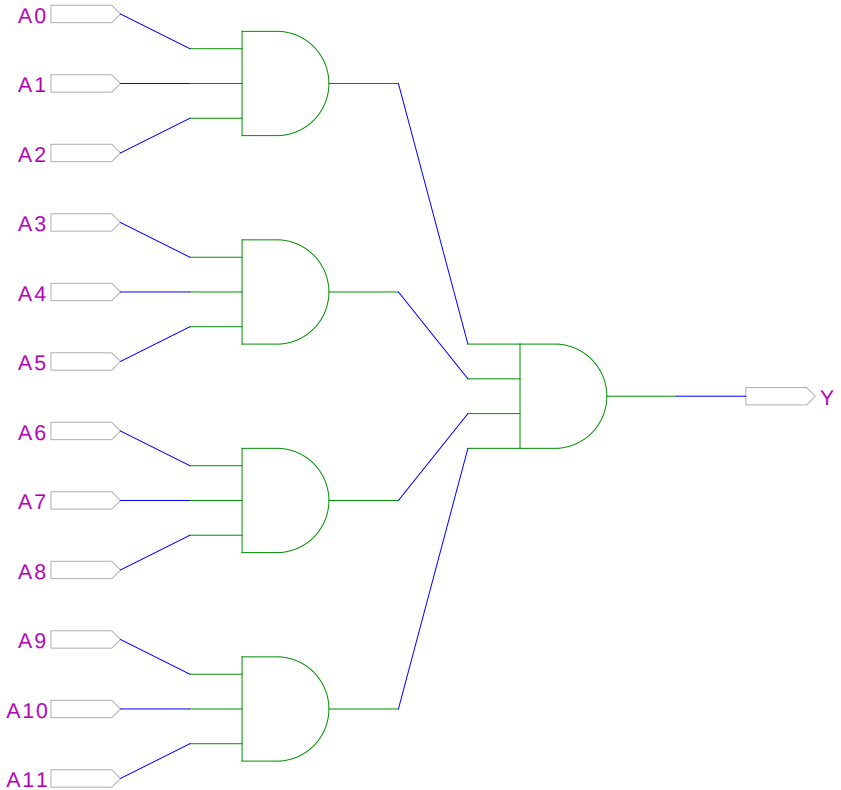


NAND (4 inputs)

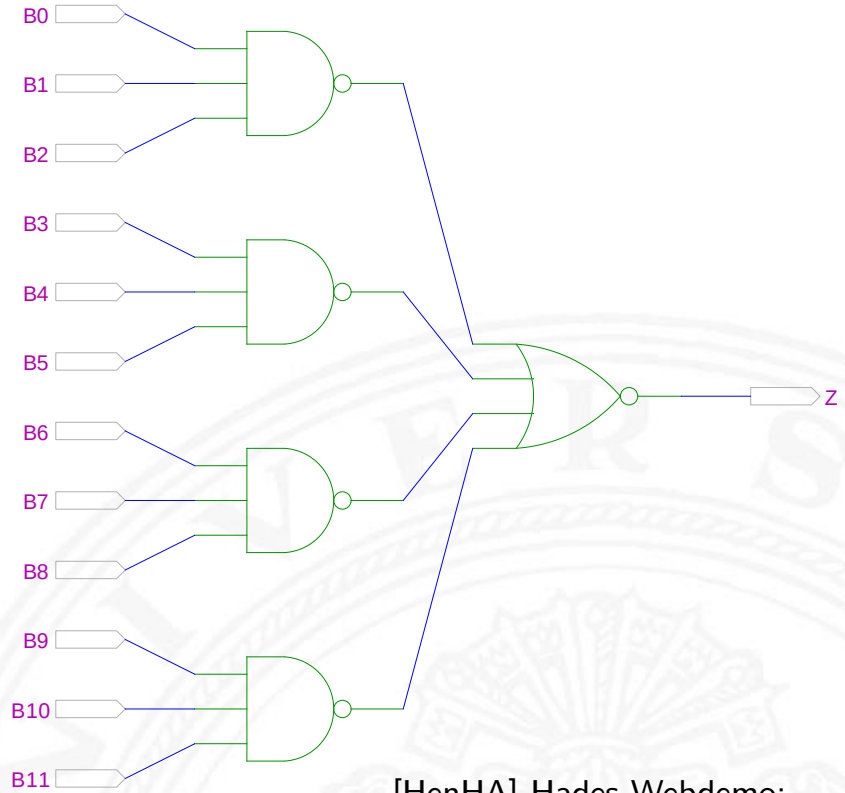


[HenHA] Hades Demo: `10-gates/00-gates/and`

Gatter: AND mit zwölf Eingängen



AND3-AND4



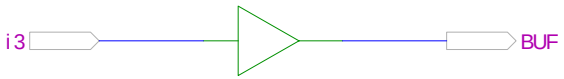
[HenHA] Hades Webdemo:
`10-gates/00-gates/andbig`

NAND3-NOR4 (De Morgan)

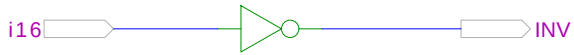
- in der Regel max. 4 Eingänge pro Gatter
  Grund: elektrotechnische Nachteile

Gatter: OR/NOR mit zwei, drei, vier Eingängen

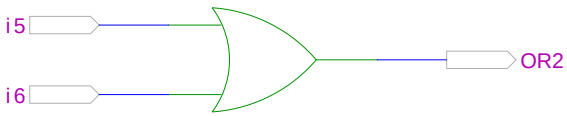
BUFFER



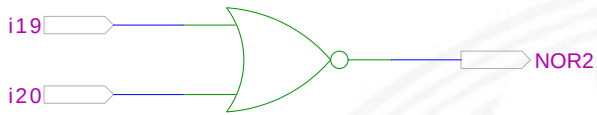
INVERTER



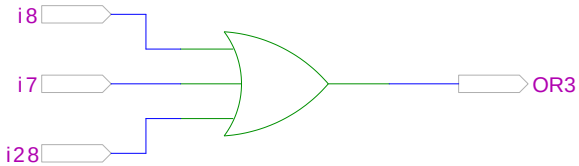
OR (2 inputs)



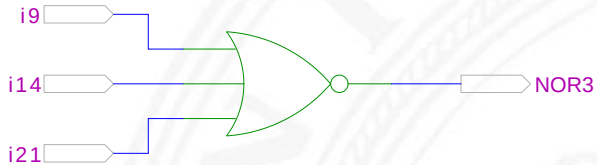
NOR (2 inputs)



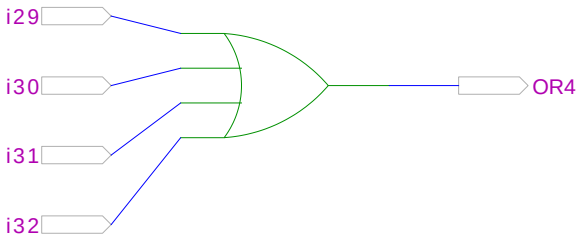
OR (3 inputs)



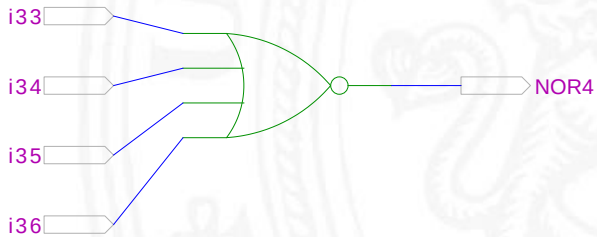
NOR (3 inputs)



OR (4 inputs)



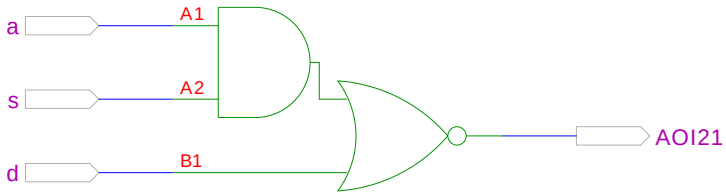
NOR (4 inputs)



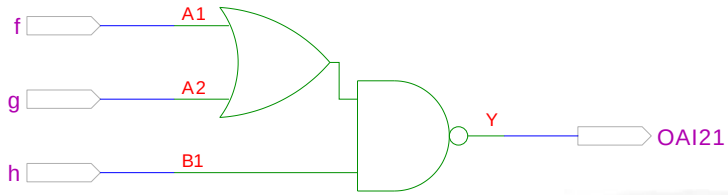
[HenHA] Hades Demo: `10-gates/00-gates/or`

Komplexgatter

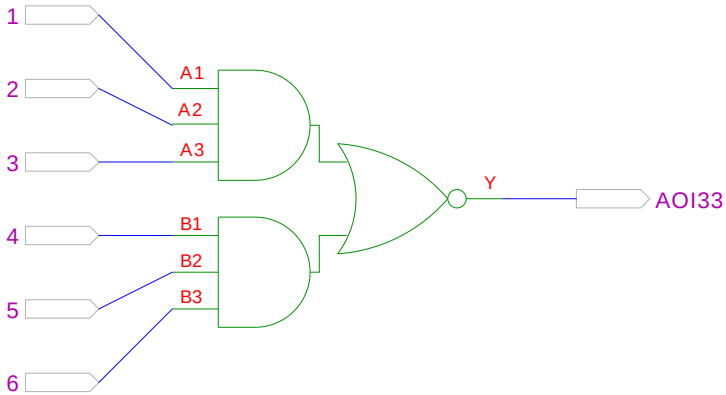
AOI21 (And-Or-Invert)



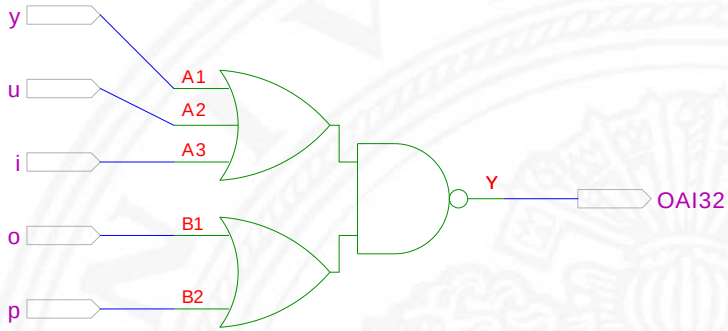
OAI21 (Or-And-Invert)



AOI33 (And-Or-Invert)



OAI32 (Or-And-Invert)

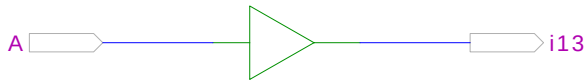


[HenHA] Hades Demo: `10-gates/00-gates/complex`

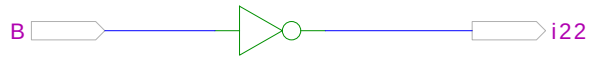
- in CMOS-Technologie besonders günstig realisierbar
  entsprechen vom Aufwand nur einem Gatter

Gatter: XOR und XNOR

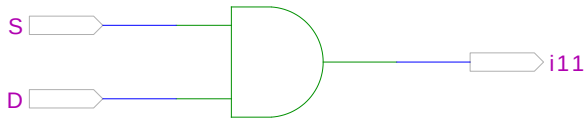
BUFFER



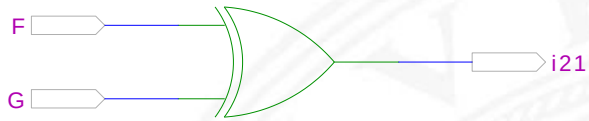
INVERTER



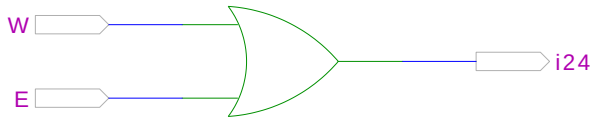
AND (2 inputs)



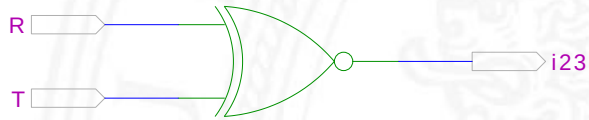
XOR (2 inputs)



OR (2 inputs)



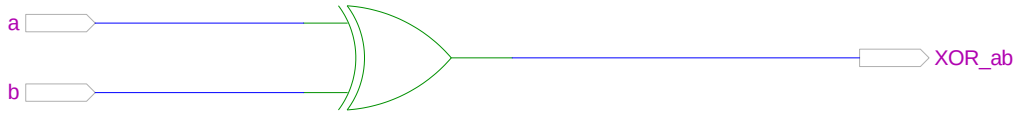
XNOR (2 inputs)



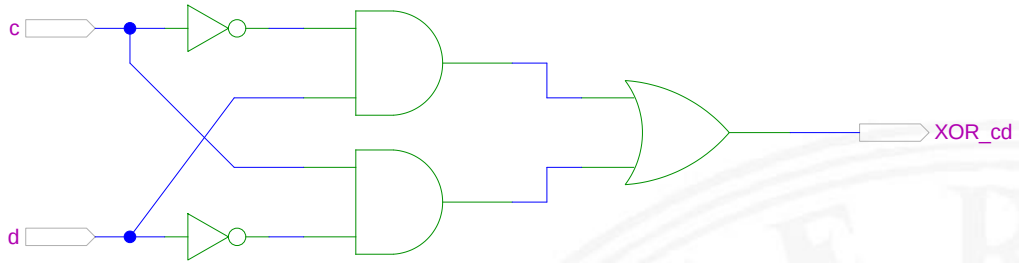
[HenHA] Hades Demo: `10-gates/00-gates/xor`

XOR und drei Varianten der Realisierung

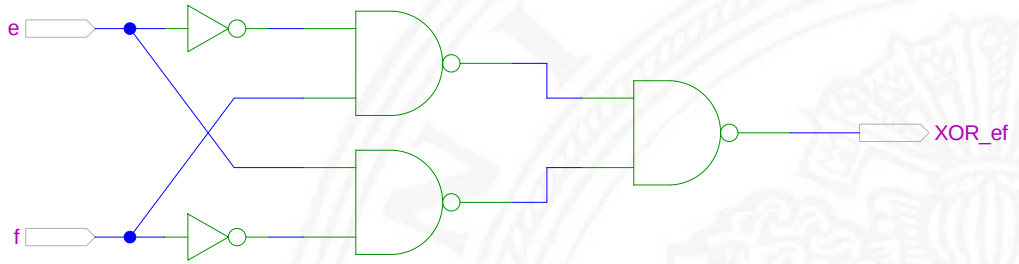
- Symbol



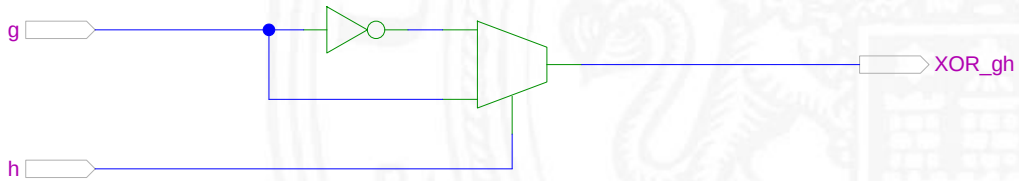
- AND-OR



- NAND-NAND



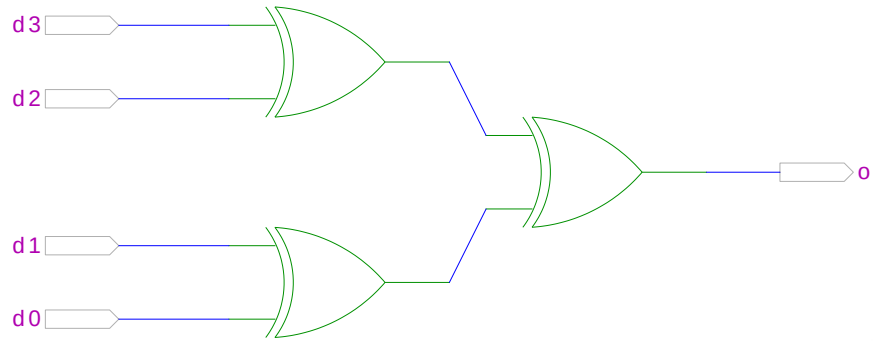
- mit Multiplexer



[HenHA] Hades Demo: `10-gates/11-xor/xor-variants`

XOR zur Berechnung der Parität

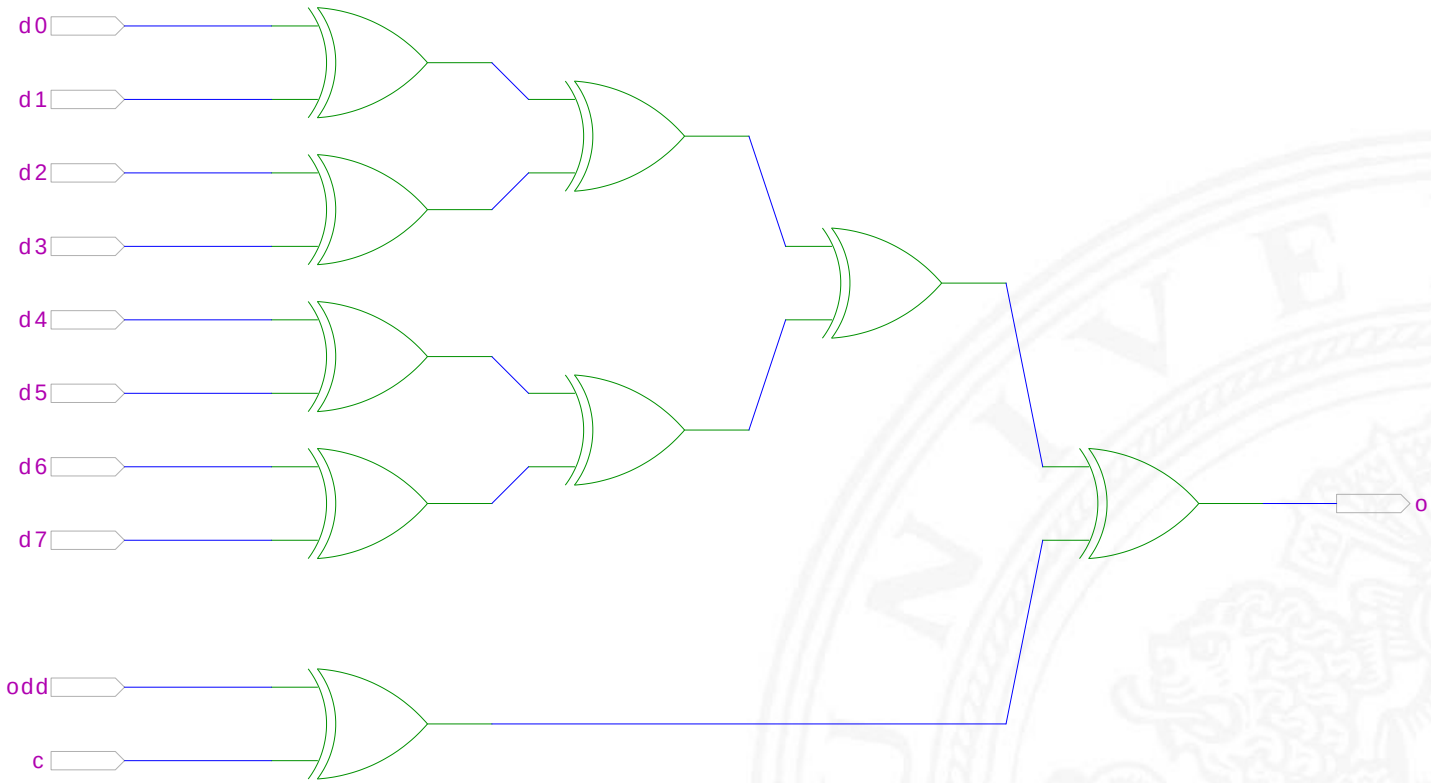
- Parität, siehe „Codierung – Fehlererkennende Codes“
- 4-bit Parität: $d_3 \oplus d_2 \oplus d_1 \oplus d_0$



[HenHA] Hades Demo: `10-gates/12-parity/parity4`

XOR zur Berechnung der Parität (cont.)

- 8-bit, bzw. 10-bit: Umschaltung odd/even
  Kaskadierung über c-Eingang



[HenHA] Hades Demo: `10-gates/12-parity/parity8`

2:1-Multiplexer

Umschalter zwischen zwei Dateneingängen („Wechselschalter“)

- ein Steuereingang: s
  zwei Dateneingänge: a_1 und a_0
  ein Datenausgang: y
- wenn $s = 1$ wird a_1 zum Ausgang y durchgeschaltet
  wenn $s = 0$ wird a_0 –"–

s	a_1	a_0	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

2:1-Multiplexer (cont.)

- kompaktere Darstellung der Funktionstabelle durch Verwendung von $*$ (don't care) Termen

s	a_1	a_0	y
0	*	0	0
0	*	1	1
1	0	*	0
1	1	*	1

s	a_1	a_0	y
0	*	a_0	a_0
1	a_1	*	a_1

- wenn $s = 0$ hängt der Ausgangswert nur von a_0 ab
  wenn $s = 1$ –"– a_1 ab

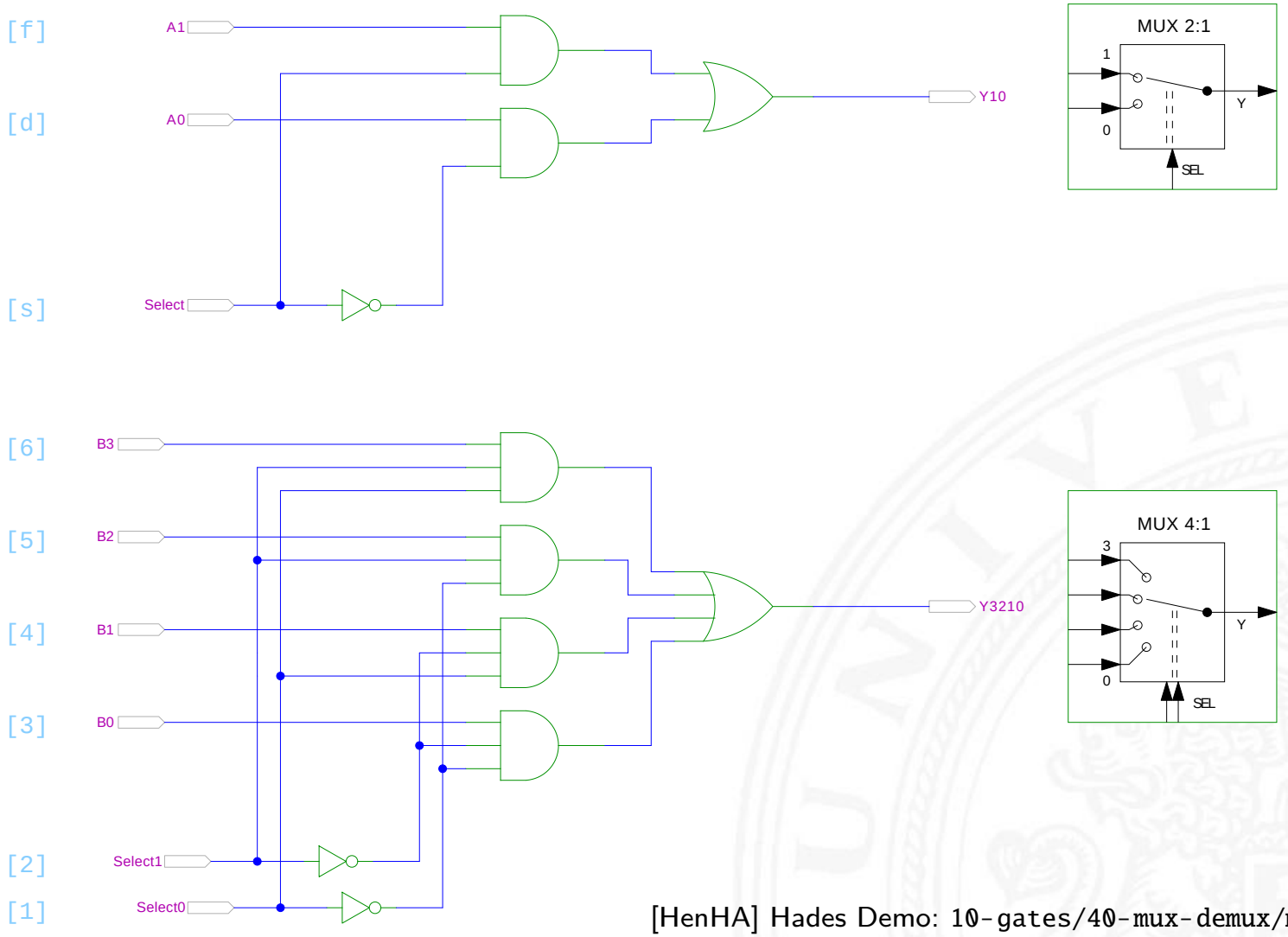
n:1-Multiplexer

Umschalten zwischen mehreren Dateneingängen

- $\lceil \log_2(n) \rceil$ Steuereingänge: $s_m, \ldots, s_0$
  n Dateneingänge: $a_{n-1}, \ldots, a_0$
  ein Datenausgang: y

s_1	s_0	a_3	a_2	a_1	a_0	y
0	0	*	*	*	0	0
0	0	*	*	*	1	1
0	1	*	*	0	*	0
0	1	*	*	1	*	1
1	0	*	0	*	*	0
1	0	*	1	*	*	1
1	1	0	*	*	*	0
1	1	1	*	*	*	1

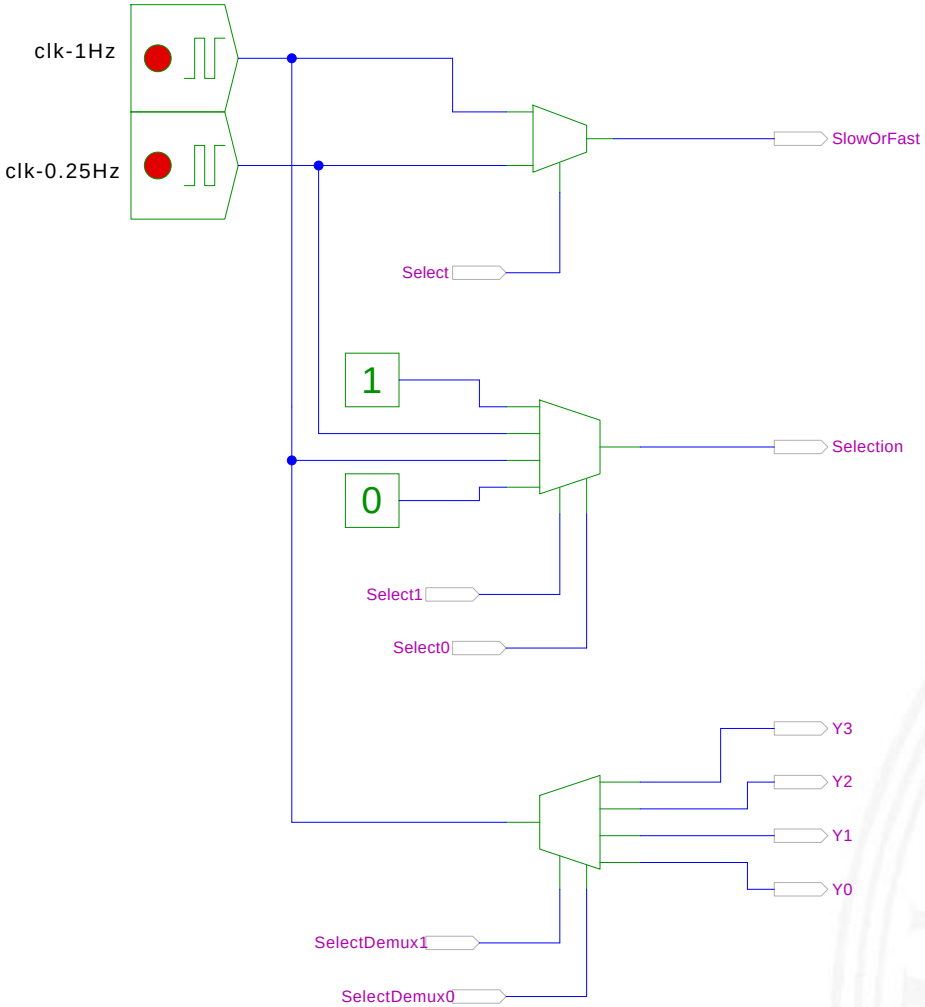
2:1 und 4:1 Multiplexer

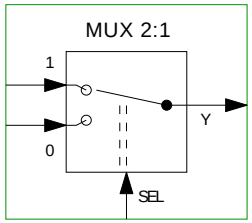


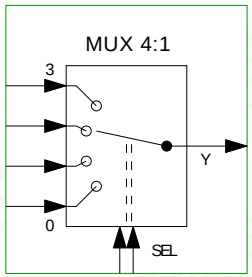
[HenHA] Hades Demo: `10-gates/40-mux-demux/mux21-mux41`

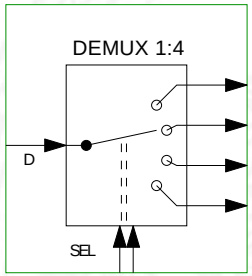
- keine einheitliche Anordnung der Dateneingänge in Schaltplänen: höchstwertiger Eingang manchmal oben, manchmal unten

Multiplexer und Demultiplexer



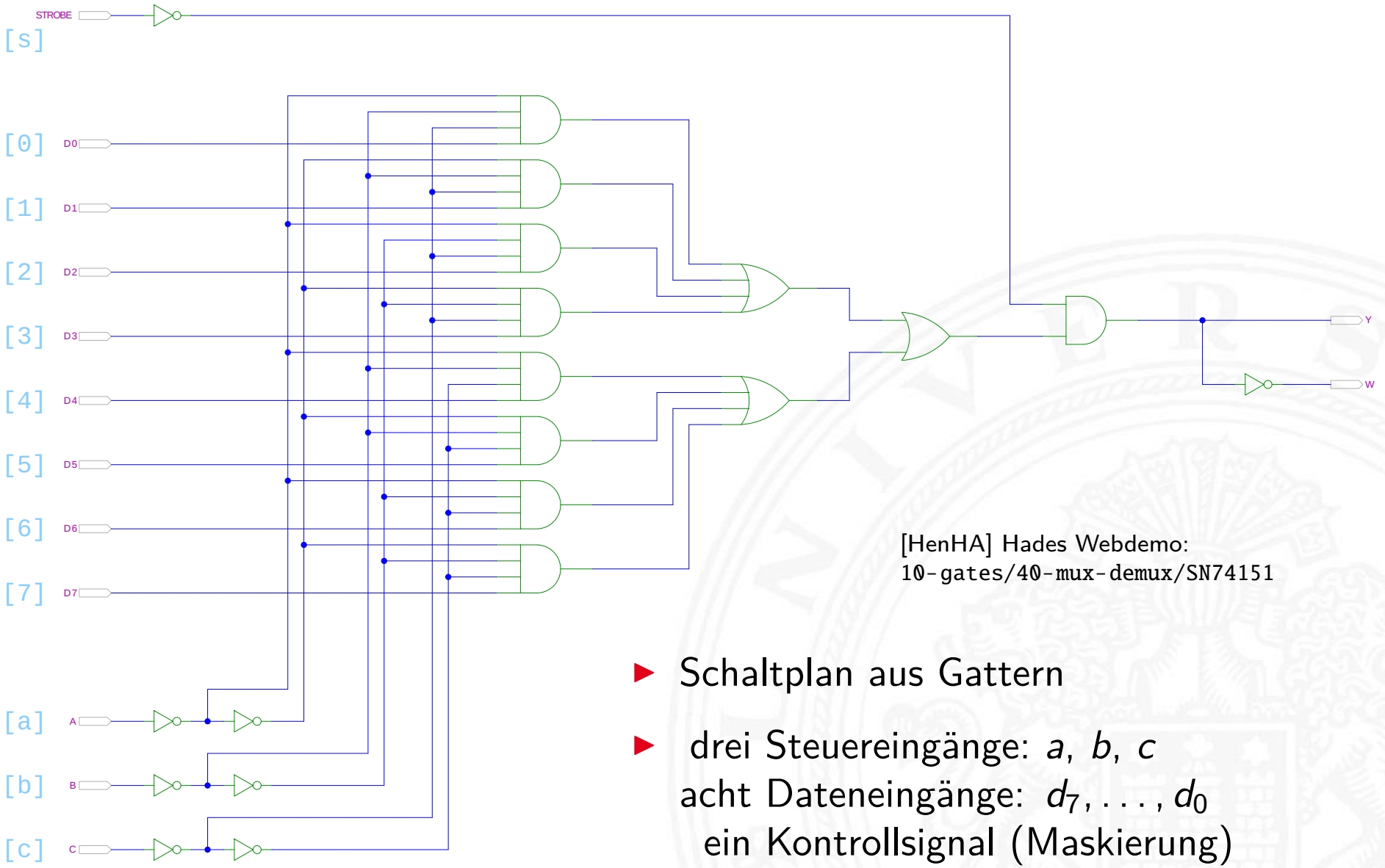






[HenHA] Hades Demo: `10-gates/40-mux-demux/mux-demux`

8-bit Multiplexer: Integrierte Schaltung 74151

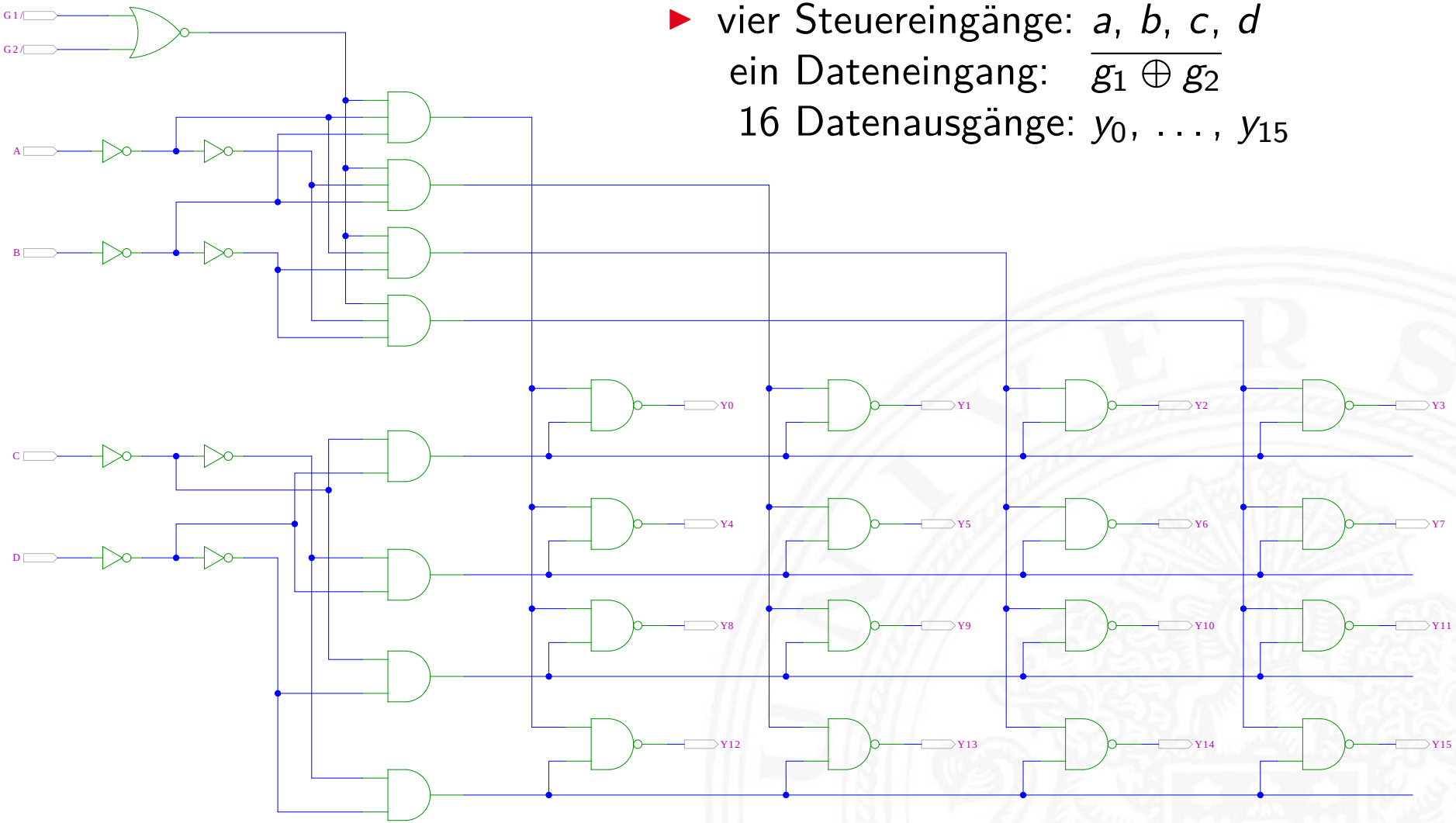


[HenHA] Hades Webdemo:
`10-gates/40-mux-demux/SN74151`

- Schaltplan aus Gattern
- drei Steuereingänge: a, b, c
  acht Dateneingänge: $d_7, \ldots, d_0$
  ein Kontrollsignal (Maskierung)

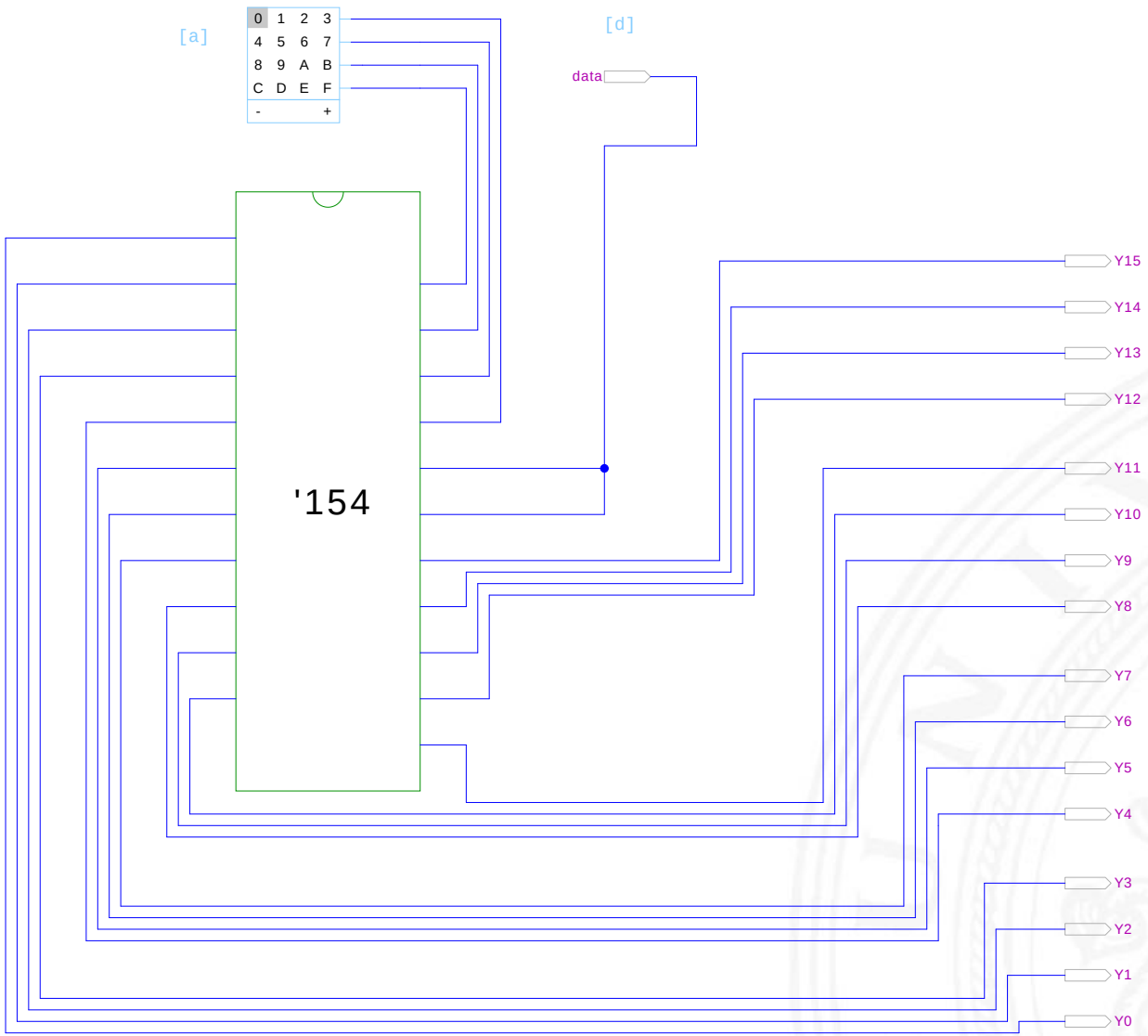
16-bit Demultiplexer: Integrierte Schaltung 74154

- vier Steuereingänge: a, b, c, d
  ein Dateneingang: $\overline{g_1 \oplus g_2}$
  16 Datenausgänge: $y_0, \ldots, y_{15}$



[HenHA] Hades Demo: `10-gates/40-mux-demux/SN74154`

16-bit Demultiplexer: 74154 als Adressdecoder



[HenHA] Hades Demo: `10-gates/40-mux-demux/demo74154`

Beispiele für Schaltnetze

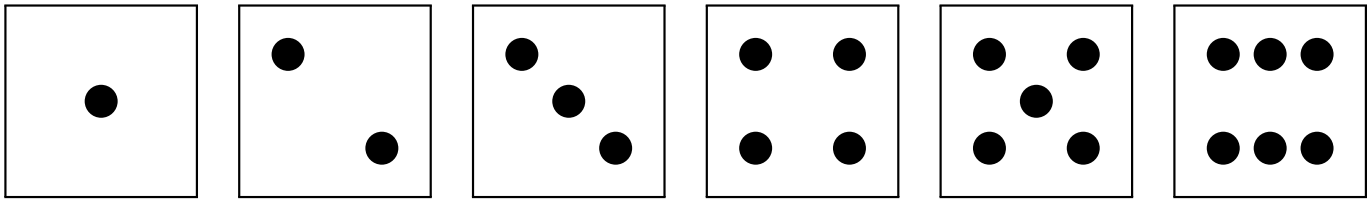
- Schaltungen mit mehreren Ausgängen
- Bündelminimierung der einzelnen Funktionen

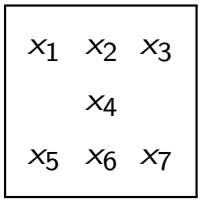
ausgewählte typische Beispiele

- „Würfel“-Decoder
- Umwandlung vom Dual-Code in den Gray-Code
- (7,4)-Hamming-Code: Encoder und Decoder
- Siebensegmentanzeige

Beispiel: „Würfel“-Decoder

Visualisierung eines Würfels mit sieben LEDs

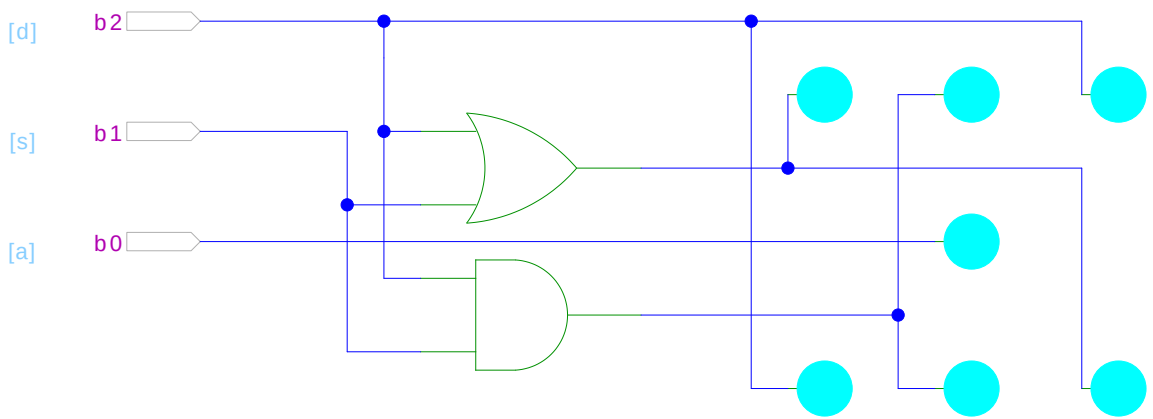




- Eingabewert von $0 \ldots 6$
- Anzeige ein bis sechs Augen: eingeschaltet

Wert	b_2	b_1	b_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1	0	0	0
2	0	1	0	1	0	0	0	0	0	1
3	0	1	1	1	0	0	1	0	0	1
4	1	0	0	1	0	1	0	1	0	1
5	1	0	1	1	0	1	1	1	0	1
6	1	1	0	1	1	1	0	1	1	1

Beispiel: „Würfel“-Decoder (cont.)



[HenHA] Hades Demo: `10-gates/10-wuerfel/wuerfel`

- Anzeige wie beim Würfel: ein bis sechs Augen
- Minimierung ergibt:

$x_1 = x_7 = b_2 \vee b_1$ links oben, rechts unten

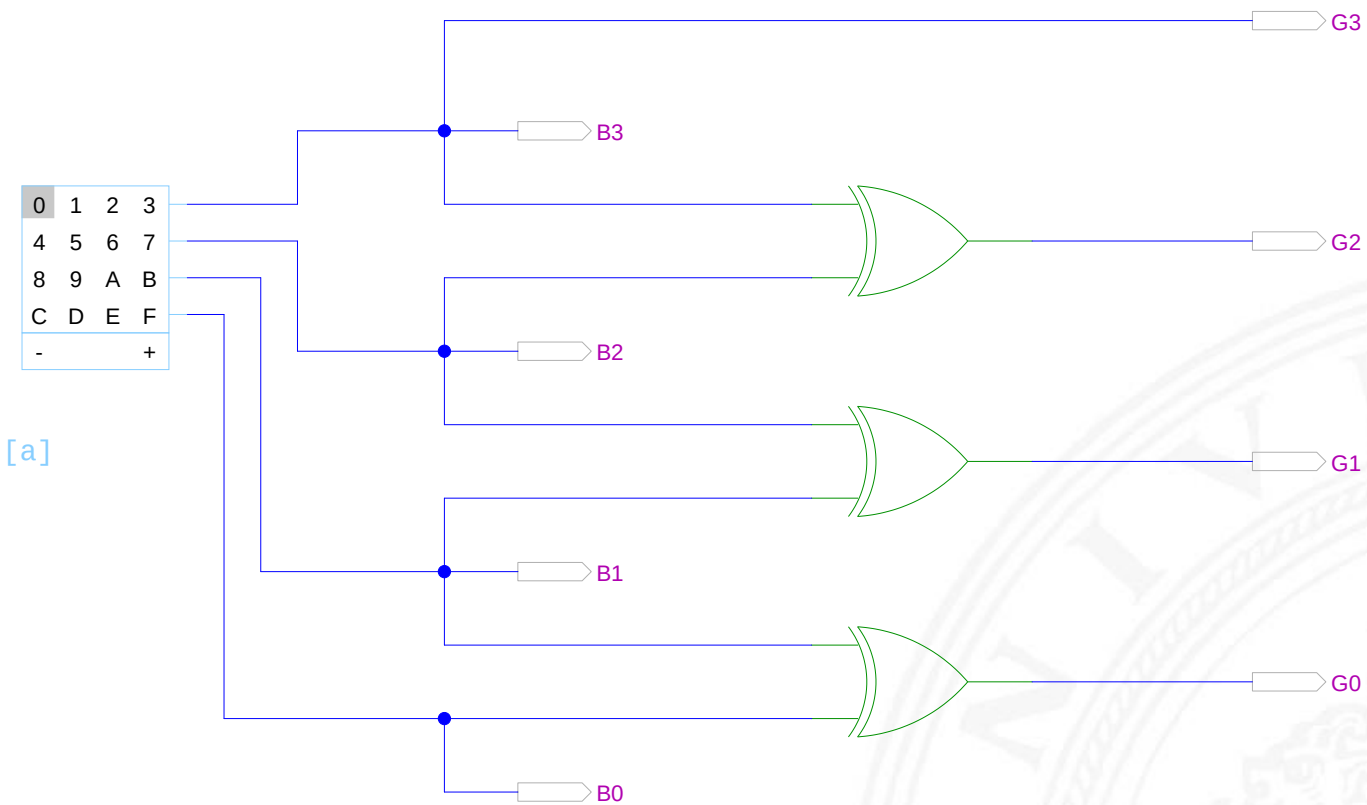
$x_2 = x_6 = b_2 \wedge b_1$ mitte oben, mitte unten

$x_3 = x_5 = b_2$ rechts oben, links unten

$x_4 = b_0$ Zentrum

Beispiel: Umwandlung vom Dualcode in den Graycode

XOR benachbarter Bits



binary code

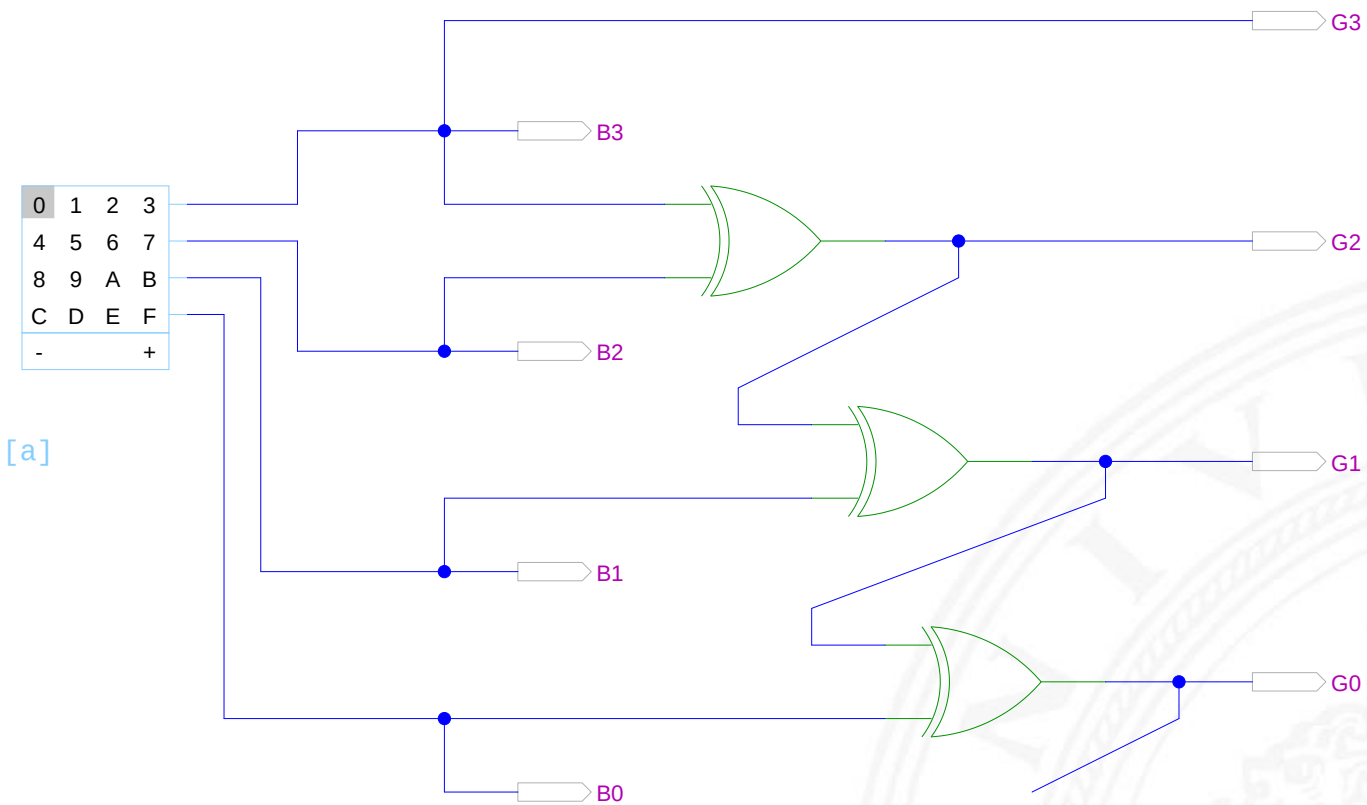
4-bit dual to Gray code converter

Gray code

[HenHA] Hades Demo: `10-gates/15-graycode/dual2gray`

Beispiel: Umwandlung vom Graycode in den Dualcode

XOR-Kette



Gray-Code

4-bit Gray- to dual-code converter

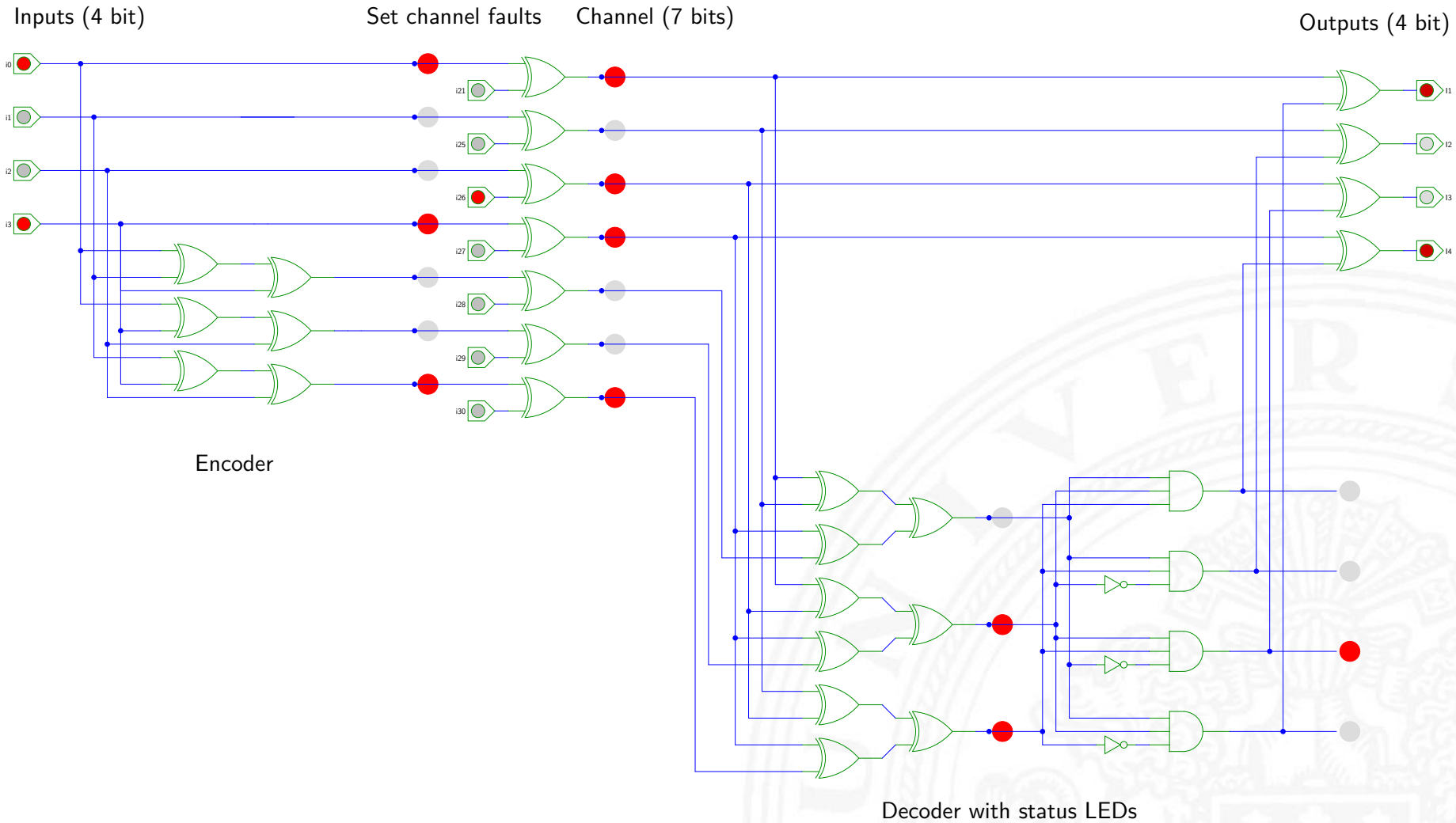
Dual-Code

[HenHA] Hades Demo: `10-gates/15-graycode/gray2dual`

(7,4)-Hamming-Code: Encoder und Decoder

- Encoder — linke Seite
  - vier Eingabebits
  - Hamming-Encoder erzeugt drei Paritätsbits
- Übertragungskanal — Mitte
  - Übertragung von sieben Codebits
  - Einfügen von Übertragungsfehlern durch Invertieren von Codebits mit XOR-Gattern
- Decoder und Fehlerkorrektur — rechte Seite
  - Decoder liest die empfangenen sieben Bits
  - Syndrom-Berechnung mit XOR-Gattern und Anzeige erkannter Fehler
  - Korrektur gekippter Bits — rechts oben

(7,4)-Hamming-Code: Encoder und Decoder (cont.)



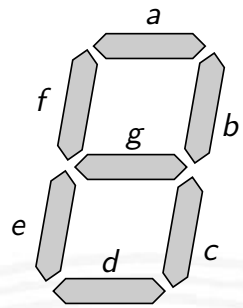
Encoder

Decoder with status LEDs

[HenHA] Hades Demo: `10-gates/50-hamming/hamming`

Siebensegmentanzeige

- sieben einzelne Leuchtsegmente (z.B. Leuchtdioden)
- Anzeige stilisierter Ziffern von `0` bis 9
- auch für Hex-Ziffern: `A`, `b`, `C`, `d`, `E`, `F`



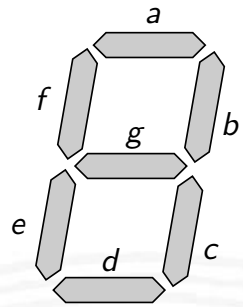
- sieben Schaltfunktionen, je eine pro Ausgang
- Umcodierung von 4-bit Dualwerten in geeignete Ausgangswerte
- Segmente im Uhrzeigersinn: *a* (oben) bis *f*, *g* innen

- eingeschränkt auch als alphanumerische Anzeige für Ziffern und (einige) Buchstaben
  - gemischt Groß- und Kleinbuchstaben
  - Probleme mit `M`, `N` usw.

Siebensegmentanzeige: Funktionen

- Funktionen für Hex-Anzeige, 0 . . . F

	0	1	2	3	4	5	6	7	8	9	A	b	C	d	E	F
a =	1	0	1	1	0	1	1	1	1	1	1	0	0	0	1	1
b =	1	1	1	1	1	0	0	1	1	1	1	1	0	0	1	0
c =	1	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0
d =	1	0	1	1	0	1	1	0	1	1	0	1	1	1	1	0
e =	1	0	1	0	0	0	1	0	1	0	1	1	1	1	1	1
f =	1	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1
g =	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1



- für Ziffernanzeige mit *Don't Care*-Termen

  a = 1 0 1 1 0 1 1 1 1 1 ∗ ∗ ∗ ∗ ∗ ∗

  b = usw.

Siebensegmentanzeige: Bündelminimierung

- zum Beispiel mit sieben KV-Diagrammen ...
- dabei versuchen, gemeinsame Terme zu finden und zu nutzen

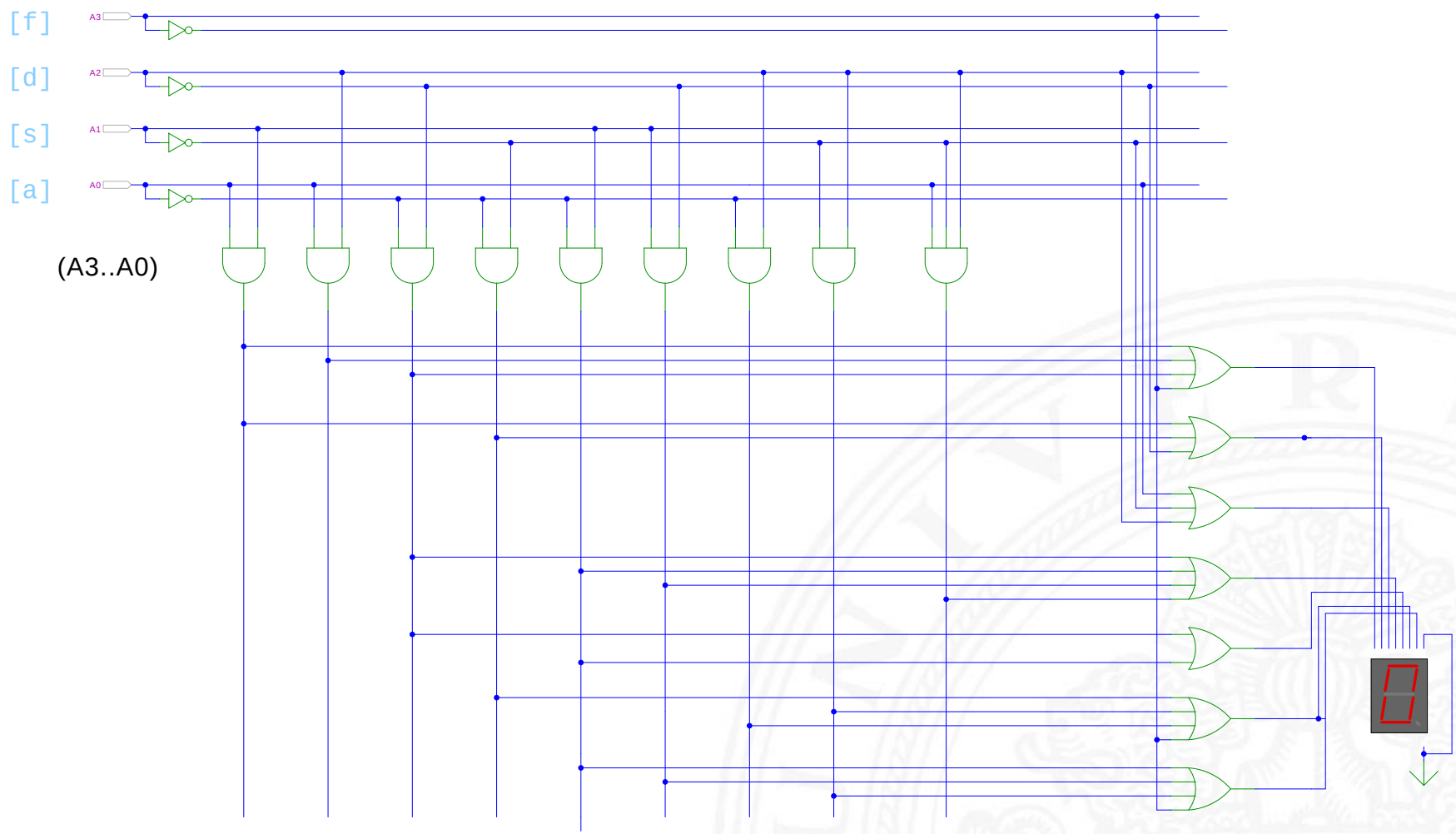
Minimierung als Übungsaufgabe?

- nächste Folie zeigt Lösung aus Schiffmann, Schmitz [SS04]

- als mehrstufige Schaltung ist günstigere Lösung möglich

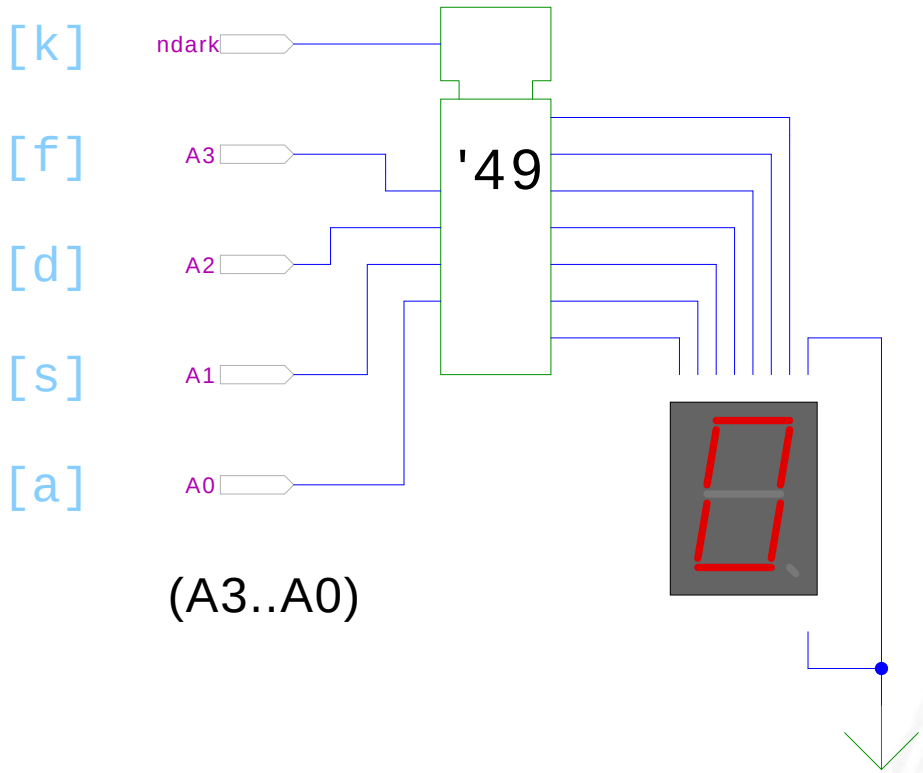
  Knuth: *AoCP, Volume 4, Fascicle 0*, 7.1.2, Seite 112ff [Knu08]

Siebensegmentdecoder: Ziffern 0 . . . 9



[HenHA] Hades Demo: `10-gates/20-sevensegment/sevensegment`

Siebensegmentdecoder: Integrierte Schaltung 7449



[HenHA] Hades Demo: `10-gates/20-sevensegment/SN7449-demo`

- Beispiel für eine integrierte Schaltung (IC)
- Anzeige von 0 . . . 9, Zufallsmuster für A . . . F, „Dunkeltastung“

Siebensegmentanzeige: mehrstufige Realisierung

Minimale Anzahl der Gatter für die Schaltung?

- Problem vermutlich nicht optimal lösbar (nicht *tractable*)
- Heuristik basierend auf „häufig“ verwendeten Teilfunktionen
- Eingänge x_1, x_2, x_3, x_4, Ausgänge $a, \ldots, g$

$$x_5 = x_2 \oplus x_3$$
$$x_6 = \overline{x_1} \wedge x_4$$
$$x_7 = x_3 \wedge \overline{x_6}$$
$$x_8 = x_1 \oplus x_2$$
$$x_9 = x_4 \oplus x_5$$
$$x_{10} = \overline{x_7} \wedge x_8$$
$$x_{11} = x_9 \oplus x_{10}$$
$$x_{12} = x_5 \wedge x_{11}$$

$$x_{13} = x_1 \oplus x_7$$
$$x_{14} = x_5 \oplus x_6$$
$$x_{15} = x_7 \vee x_{12}$$
$$x_{16} = x_1 \vee x_5$$
$$x_{17} = x_5 \vee x_6$$
$$x_{18} = x_9 \wedge x_{10}$$
$$x_{19} = x_3 \wedge x_9$$

$$\overline{a} = x_{20} = x_{14} \wedge \overline{x_{19}}$$
$$\overline{b} = x_{21} = x_7 \oplus x_{12}$$
$$\overline{c} = x_{22} = \overline{x_8} \wedge x_{15}$$
$$\overline{d} = x_{23} = x_9 \wedge \overline{x_{13}}$$
$$\overline{e} = x_{24} = x_6 \vee x_{18}$$
$$\overline{f} = x_{25} = \overline{x_8} \wedge x_{17}$$
$$g = x_{26} = x_7 \vee x_{16}$$

D. E. Knuth: *AoCP, Volume 4, Fascicle 0*, Kap 7.1.2, Seite 113 [Knu08]

Logische und arithmetische Operationen

- Halb- und Volladdierer
- Addierertypen
  - Ripple-Carry
  - Carry-Lookahead

- Multiplizierer
- Quadratwurzel

- Barrel-Shifter
- ALU

Halbaddierer

- **Halbaddierer**: berechnet 1-bit Summe s und Übertrag c_o (*carry-out*) von zwei Eingangsbits a und b

a	b	c_o	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$c_o = a \wedge b$$

$$s = a \oplus b$$

Volladdierer

- **Volladdierer**: berechnet 1-bit Summe s und Übertrag c_o (*carry-out*) von zwei Eingangsbits a, b sowie Eingangsübertrag c_i (*carry-in*)

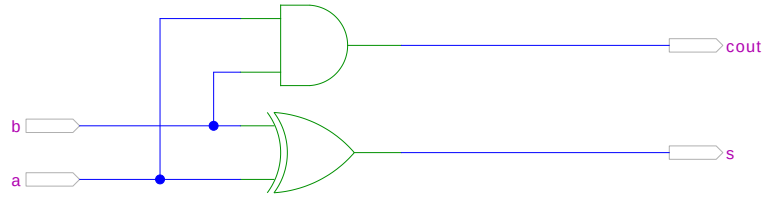
a	b	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$c_o = ab \vee ac_i \vee bc_i = (ab) \vee (a \vee b)c_i$$

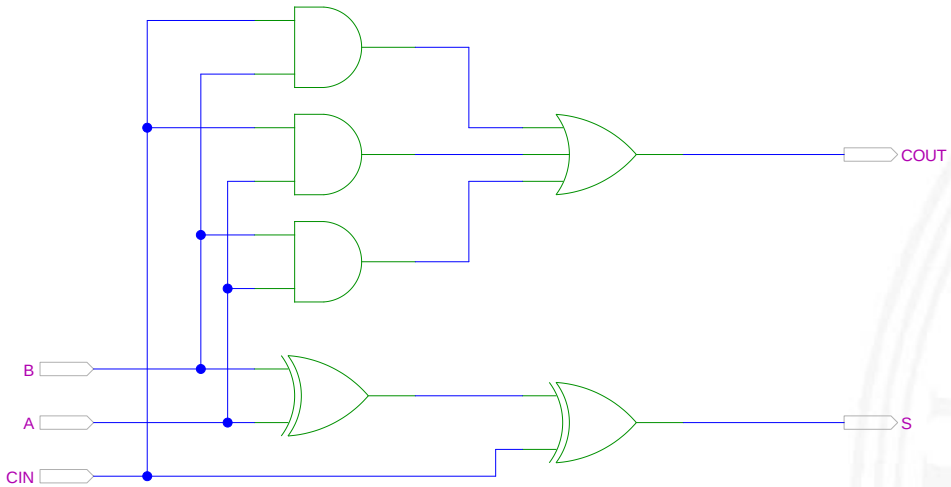
$$s = a \oplus b \oplus c_i$$

Schaltbilder für Halb- und Volladdierer

1-bit half-adder: (COUT,S) = (A+B)



1-bit full-adder: (COUT,S) = (A+B+Cin)



[HenHA] Hades Demo: `20-arithmetic/10-adders/halfadd-fulladd`

n-bit Addierer

- Summe: $s_n = a_n \oplus b_n \oplus c_n$

$$
\begin{aligned}
s_0 &= a_0 \oplus b_0 \\
s_1 &= a_1 \oplus b_1 \oplus c_1 \\
s_2 &= a_2 \oplus b_2 \oplus c_2 \\
&\dots \\
s_n &= a_n \oplus b_n \oplus c_n
\end{aligned}
$$

- Übertrag: $c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$

$$
\begin{aligned}
c_1 &= (a_0 b_0) \\
c_2 &= (a_1 b_1) \vee (a_1 \vee b_1) c_1 \\
c_3 &= (a_2 b_2) \vee (a_2 \vee b_2) c_2 \\
&\dots \\
c_{n+1} &= (a_n b_n) \vee (a_n \vee b_n) c_n
\end{aligned}
$$

n-bit Addierer (cont.)

- n-bit Addierer theoretisch als zweistufige Schaltung realisierbar
- direkte und negierte Eingänge, dann AND-OR Netzwerk
- Aufwand steigt exponentiell mit n an,
  für Ausgang n sind $2^{(2n-1)}$ Minterme erforderlich

$\Rightarrow$ nicht praktikabel

- Problem: Übertrag (*carry*)
  $c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$
  rekursiv definiert

n-bit Addierer (cont.)

Diverse gängige Alternativen für Addierer

- Ripple-Carry
  - lineare Struktur
  - \+ klein, einfach zu implementieren
  - – langsam, Laufzeit $\mathcal{O}(n)$
- Carry-Lookahead (CLA)
  - Baumstruktur
  - \+ schnell
  - – teuer (Flächenbedarf der Hardware)
- Mischformen: Ripple-block CLA, Block CLA, Parallel Prefix
- andere Ideen: Carry-Select, Conditional Sum, Carry-Skip

...

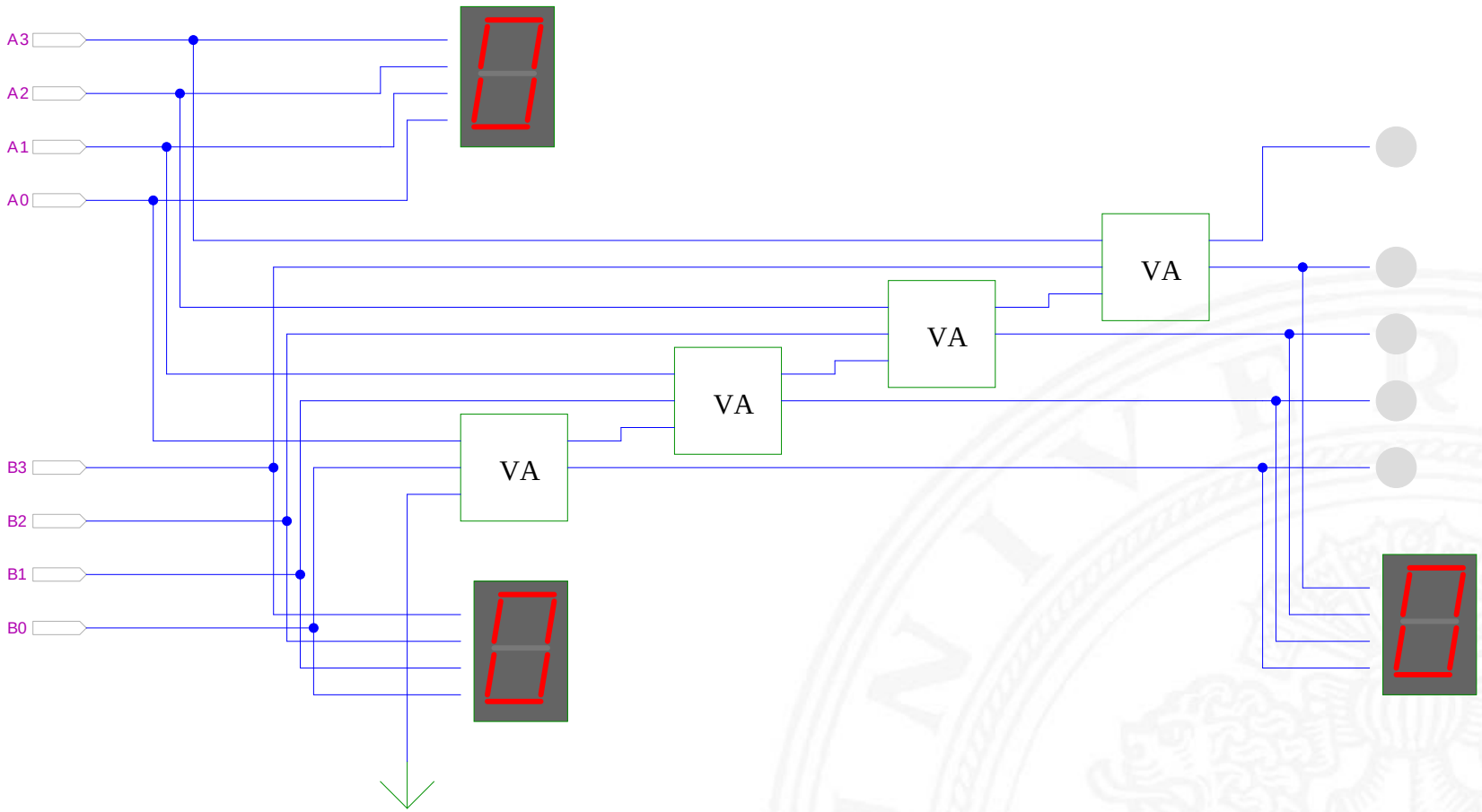
Ripple-Carry Adder

- Kaskade aus n einzelnen Volladdierern
- Carry-out von Stufe i treibt Carry-in von Stufe $i+1$
- Gesamtverzögerung wächst mit der Anzahl der Stufen als $\mathcal{O}(n)$

- Addierer in Prozessoren häufig im *kritischen Pfad*
- möglichst hohe Performanz ist essenziell
- Ripple-Carry in CMOS-Technologie bis ca. 10-bit geeignet
- bei größerer Wortbreite gibt es effizientere Schaltungen

- Überlauf-Erkennung: $c_o(n) \neq c_o(n-1)$

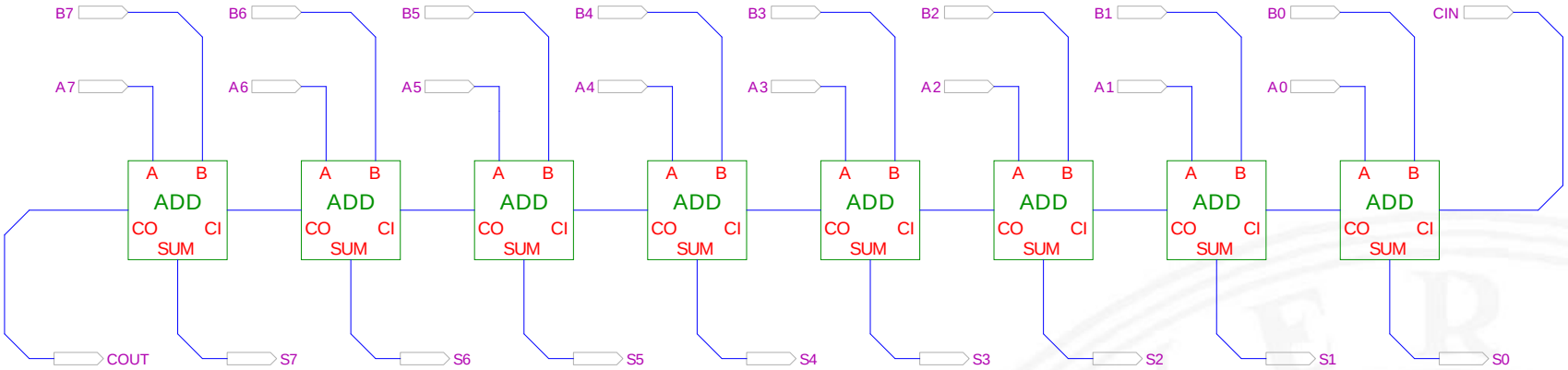
Ripple-Carry Adder: 4-bit



Schiffmann, Schmitz: *Technische Informatik I* [SS04]

Ripple-Carry Adder: Hades-Beispiel mit Verzögerungen

- Kaskade aus acht einzelnen Volladdierern



[HenHA] Hades Demo: `20-arithmetic/10-adders/ripple`

- Gatterlaufzeiten in der Simulation bewusst groß gewählt
- Ablauf der Berechnung kann interaktiv beobachtet werden
- alle Addierer arbeiten parallel
- aber Summe erst fertig, wenn alle Stufen durchlaufen sind

Subtrahierer

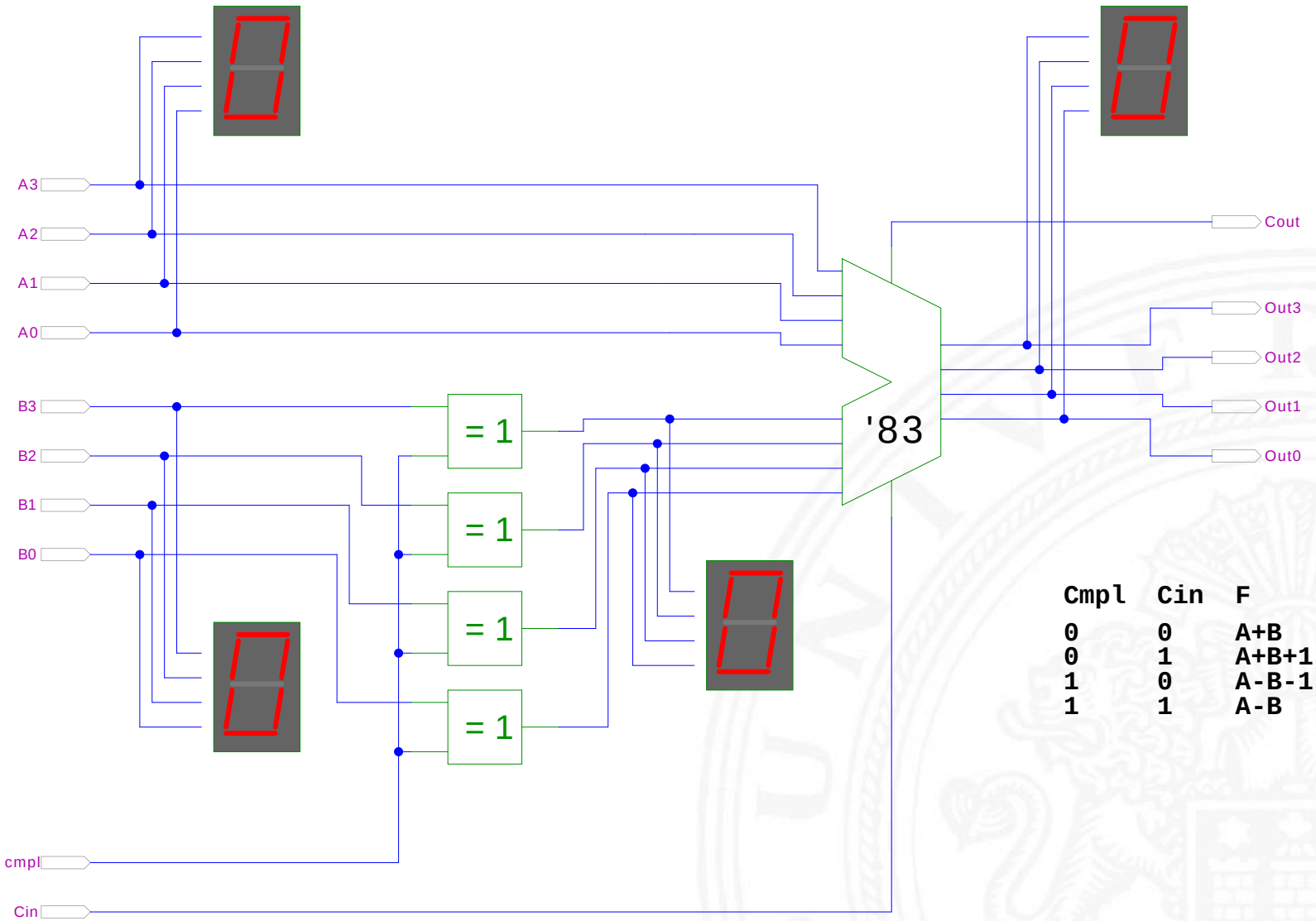
Zweierkomplement

- $(A - B)$ ersetzt durch Addition des 2-Komplements von B
- 2-Komplement: Invertieren aller Bits und Addition von Eins
- Carry-in Eingang des Addierers bisher nicht benutzt

Subtraktion quasi „gratis“ realisierbar

- normalen Addierer verwenden
- Invertieren der Bits von B (1-Komplement)
- Carry-in Eingang auf 1 setzen (Addition von 1)
- Resultat ist $A + \overline{B} + 1 = A - B$

Subtrahierer: Beispiel 7483 – 4-bit Addierer



Cmpl	Cin	F	
0	0	A+B	
0	1	A+B+1	(A++)
1	0	A-B-1	(A--)
1	1	A-B	

Schiffmann, Schmitz: *Technische Informatik I* [SS04]

Schnelle Addierer

- Addierer in Prozessoren häufig im *kritischen Pfad*
- möglichst hohe Performanz ist essenziell

$\Rightarrow$ bestimmt Taktfrequenz

- Carry-Select Adder: Gruppen von Ripple-Carry
- Carry-Lookahead Adder: Baumstruktur zur Carry-Berechnung
- ...

- über 10 Addierer „Typen“ (für 2 Operanden)
- Addition mehrerer Operanden
- Typen teilweise technologieabhängig
- Übersicht beispielsweise auf
  `www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html`

Carry-Select Adder: Prinzip

Ripple-Carry Addierer muss auf die Überträge warten ($\mathcal{O}(n)$)

- Aufteilen des n-bit Addierers in mehrere Gruppen mit je m_i-bits
- für jede Gruppe
  - jeweils zwei m_i-bit Addierer
  - einer rechnet mit $c_i = 0$ $(a + b)$, der andere mit $c_i = 1$ $(a + b + 1)$
  - 2:1-Multiplexer mit m_i-bit wählt die korrekte Summe aus

- Sobald der Wert von c_i bekannt ist (Ripple-Carry), wird über den Multiplexer die benötigte Zwischensumme ausgewählt
- Das berechnete Carry-out c_o der Gruppe ist das Carry-in c_i der folgenden Gruppe

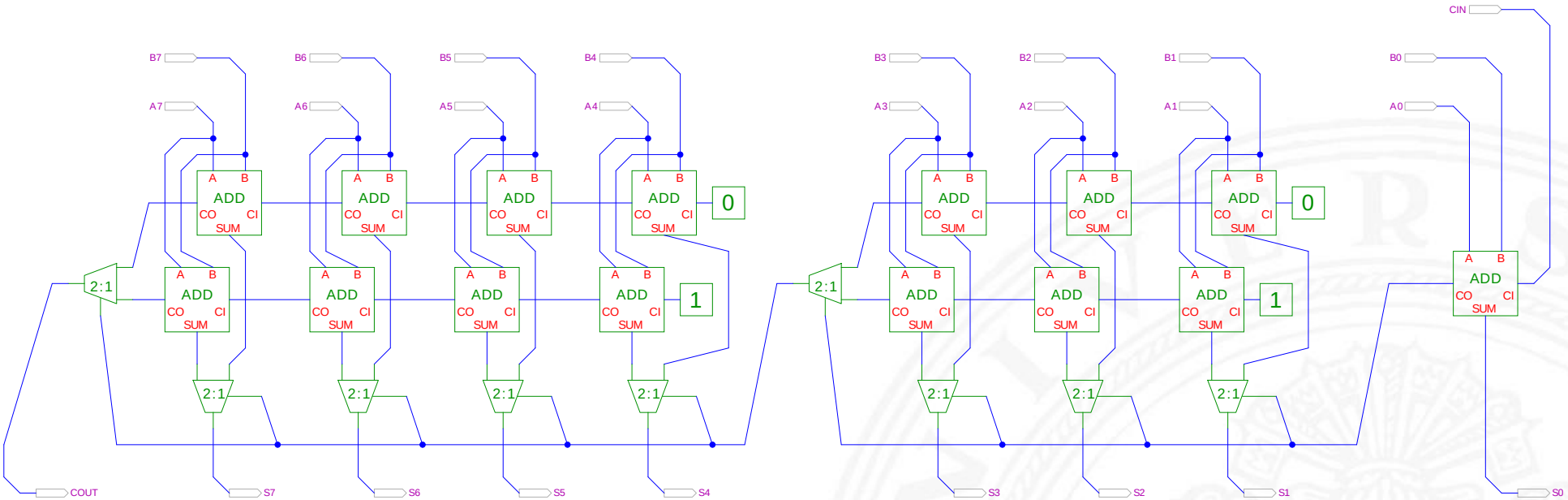
$\Rightarrow$ Verzögerung reduziert sich auf die Verzögerung eines m-bit Addierers plus die Verzögerungen der Multiplexer

Carry-Select Adder: Beispiel

8-Bit Carry-Select Adder (4 + 3 + 1 bit blocks)

4-bit Carry-Select Adder block

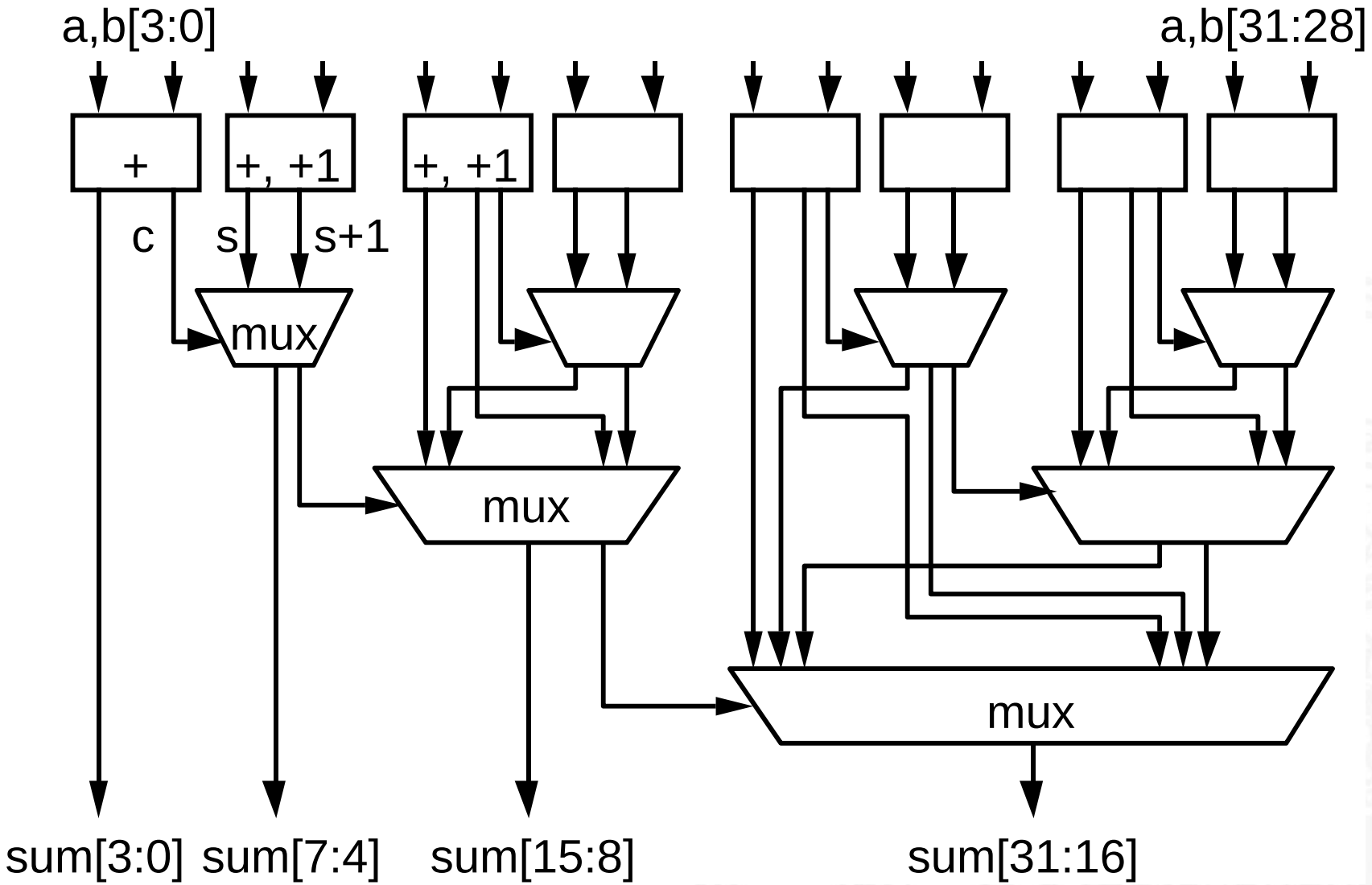
3-bit Carry-Select Adder block



[HenHA] Hades Demo: `20-arithmetic/20-carryselect/adder_carryselect`

- drei Gruppen: 1-bit, 3-bit, 4-bit
- Gruppengrößen so wählen, dass Gesamtverzögerung minimal

Carry-Select Adder: Beispiel ARM v6



S. Furber: *ARM System-on-Chip Architecture* [Fur00]

Carry-Lookahead Adder: Prinzip

- $c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$

- Einführung von Hilfsfunktionen

  $g_n = (a_n b_n)$ „generate carry“

  $p_n = (a_n \vee b_n)$ „propagate carry“

  $c_{n+1} = g_n \vee p_n c_n$

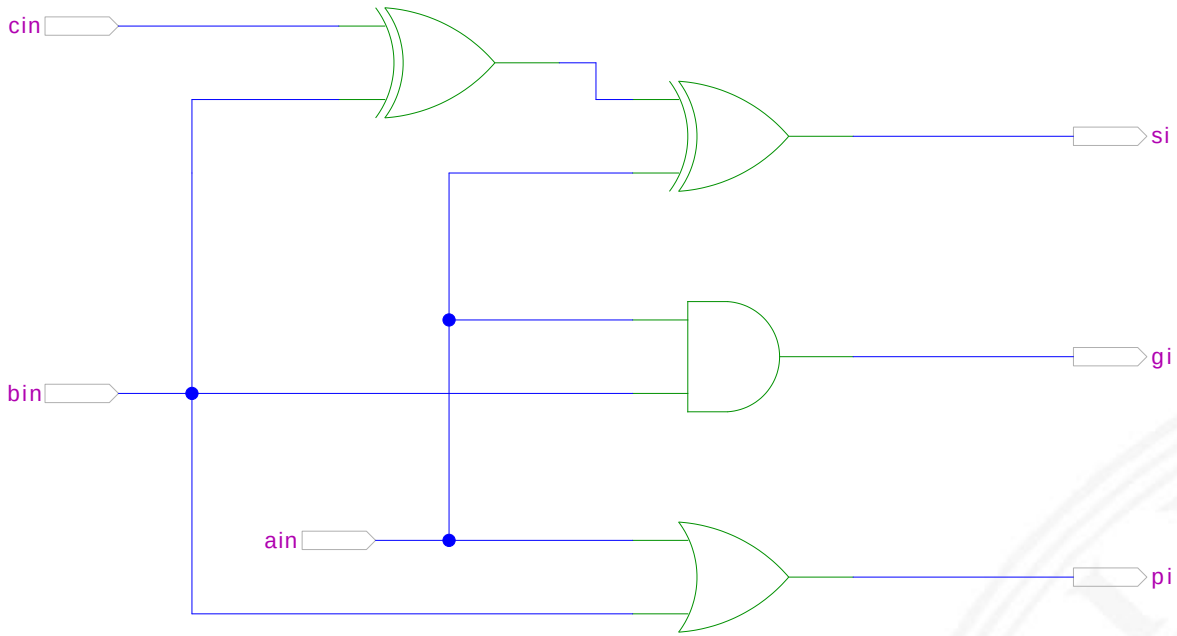
- *generate*: Carry-out erzeugen, unabhängig von Carry-in

  propagate: Carry-out weiterleiten / Carry-in maskieren

- Berechnung der g_n und p_n in einer Baumstruktur

  Tiefe des Baums ist $\log_2 N \Rightarrow$ entsprechend schnell

Carry-Lookahead Adder: SUM-Funktionsblock

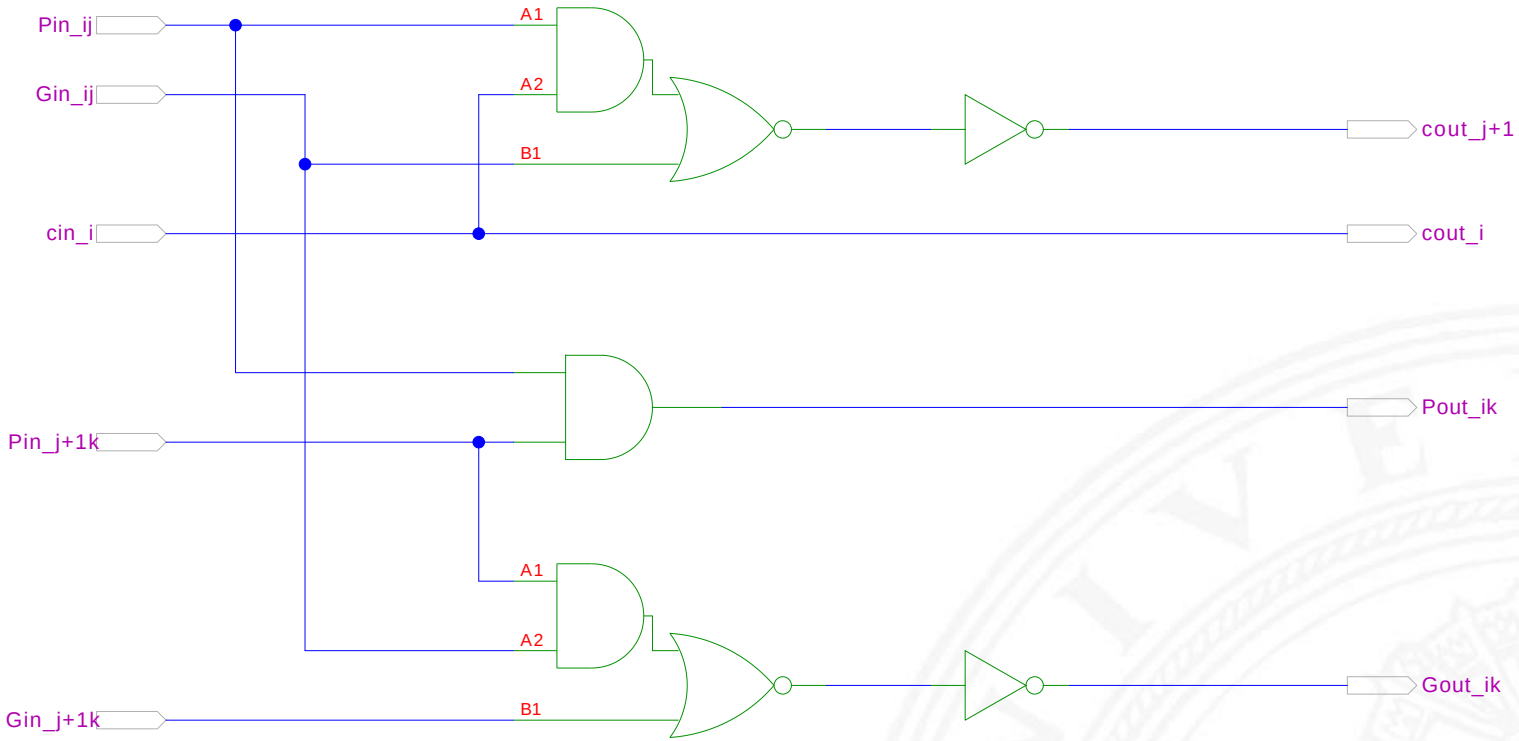


[HenHA] Hades Demo: `20-arithmetic/30-cla/sum`

- 1-bit Addierer, $s = a_i \oplus b_i \oplus c_i$
- keine Berechnung des Carry-out
- Ausgang $g_i = a_i \wedge b_i$ liefert *generate carry*

  $p_i = a_i \vee b_i$ –"– *propagate carry*

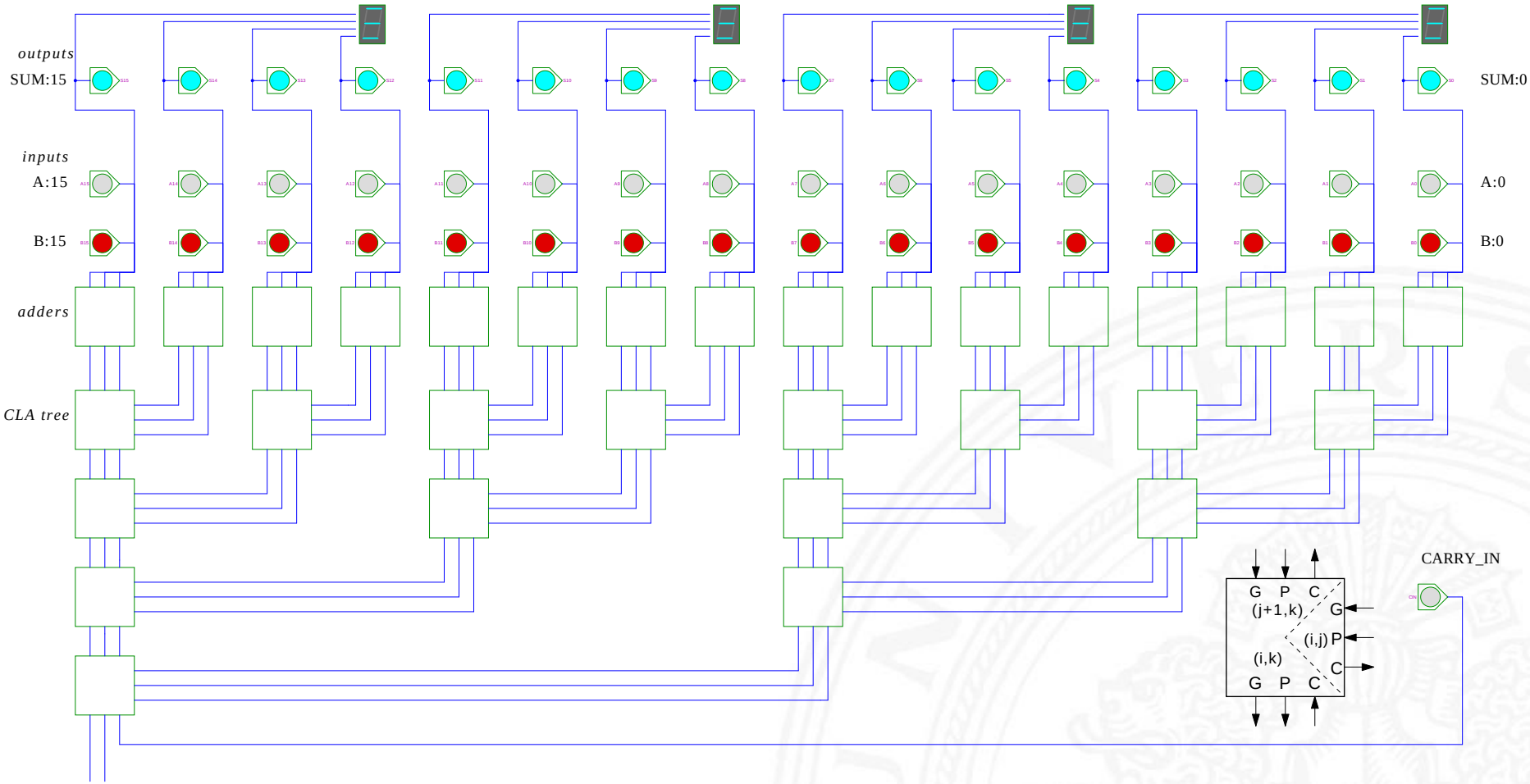
Carry-Lookahead Adder: CLA-Funktionsblock



[HenHA] Hades Demo: `20-arithmetic/30-cla/cla`

- Eingänge
  - propagate/generate Signale von zwei Stufen
  - carry-in Signal
- Ausgänge
  - propagate/generate Signale zur nächsthöheren Stufe
  - carry-out Signale: Durchleiten und zur nächsthöheren Stufe

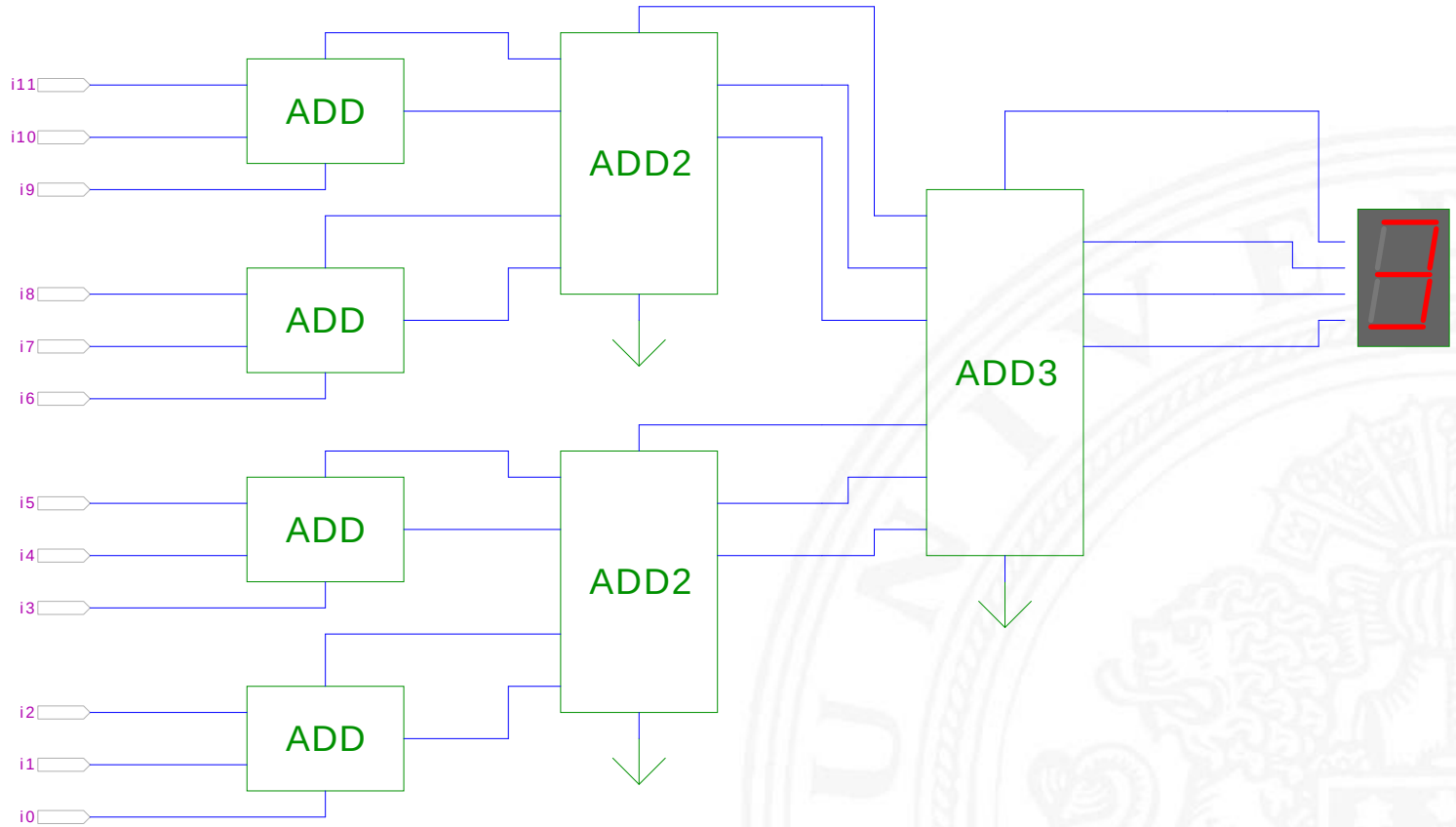
Carry-Lookahead Adder: 16-bit Addierer



[HenHA] Hades Demo: `20-arithmetic/30-cla/adder16`

Addition mehrerer Operanden

- Addierer-Bäume
- Beispiel: Bitcount



[HenHA] Hades Demo: `20-arithmetic/80-bitcount/bitcount`

Addierer: Zusammenfassung

- Halbaddierer $(a \oplus b)$
- Volladdierer $(a \oplus b \oplus c_i)$

- Ripple-Carry
  - Kaskade aus Volladdierern, einfach und billig
  - aber manchmal zu langsam, Verzögerung: $\mathcal{O}(n)$
- Carry-Select Prinzip
  - Verzögerung $\mathcal{O}(\sqrt{n})$
- Carry-Lookahead Prinzip
  - Verzögerung $\mathcal{O}(\ln n)$

- Subtraktion durch Zweierkomplementbildung

  erlaubt auch Inkrement (`A++`) und Dekrement (`A--`)

Multiplizierer

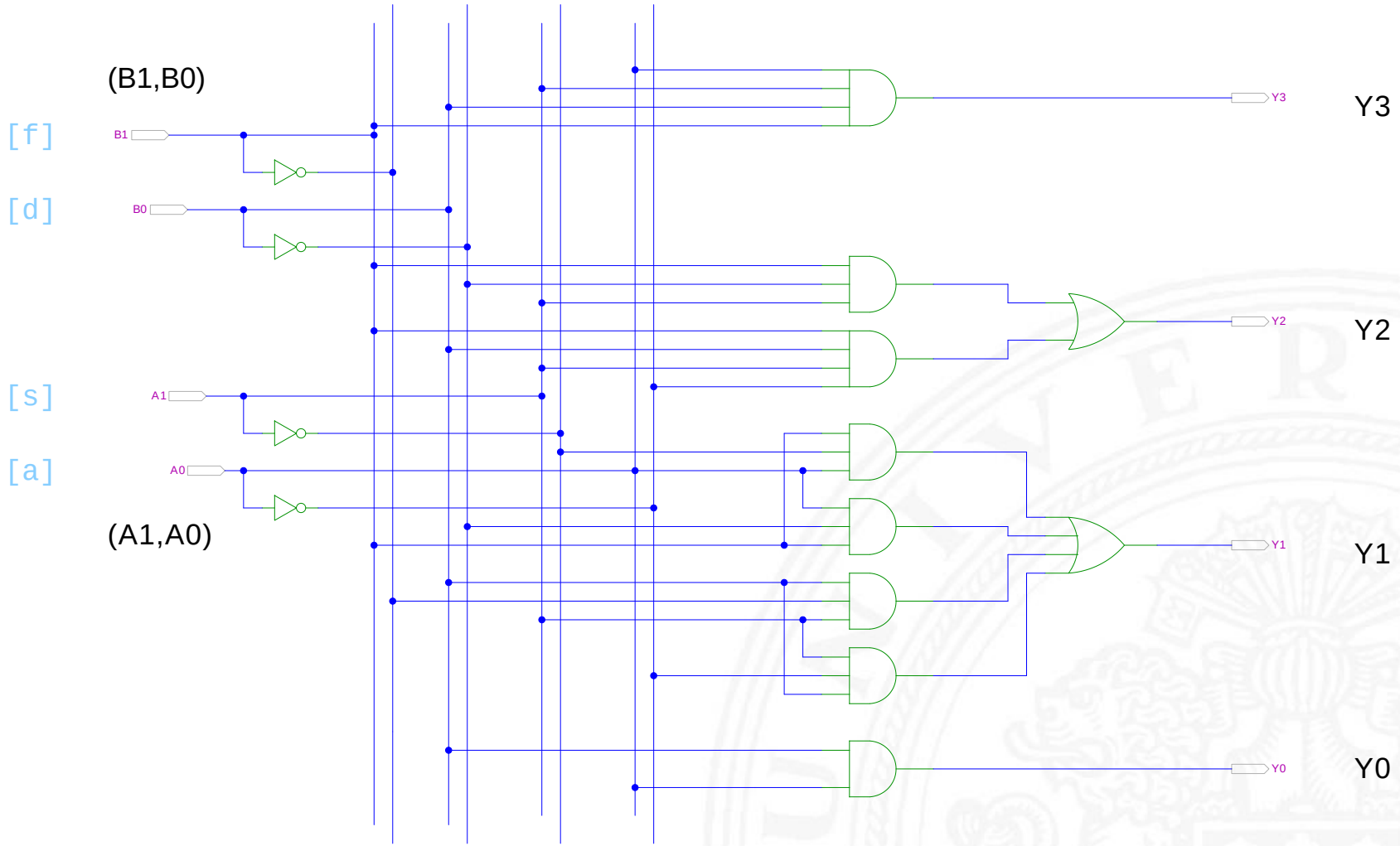
- Teilprodukte als UND-Verknüpfung des Multiplikators mit je einem Bit des Multiplikanden
- Aufaddieren der Teilprodukte mit Addierern
- Realisierung als Schaltnetz erfordert:

  n^2 UND-Gatter (bitweise eigentliche Multiplikation)

  n^2 Volladdierer (Aufaddieren der Teilprodukte)
- abschließend ein n-bit Addierer für die Überträge
- in heutiger CMOS-Technologie kein Problem

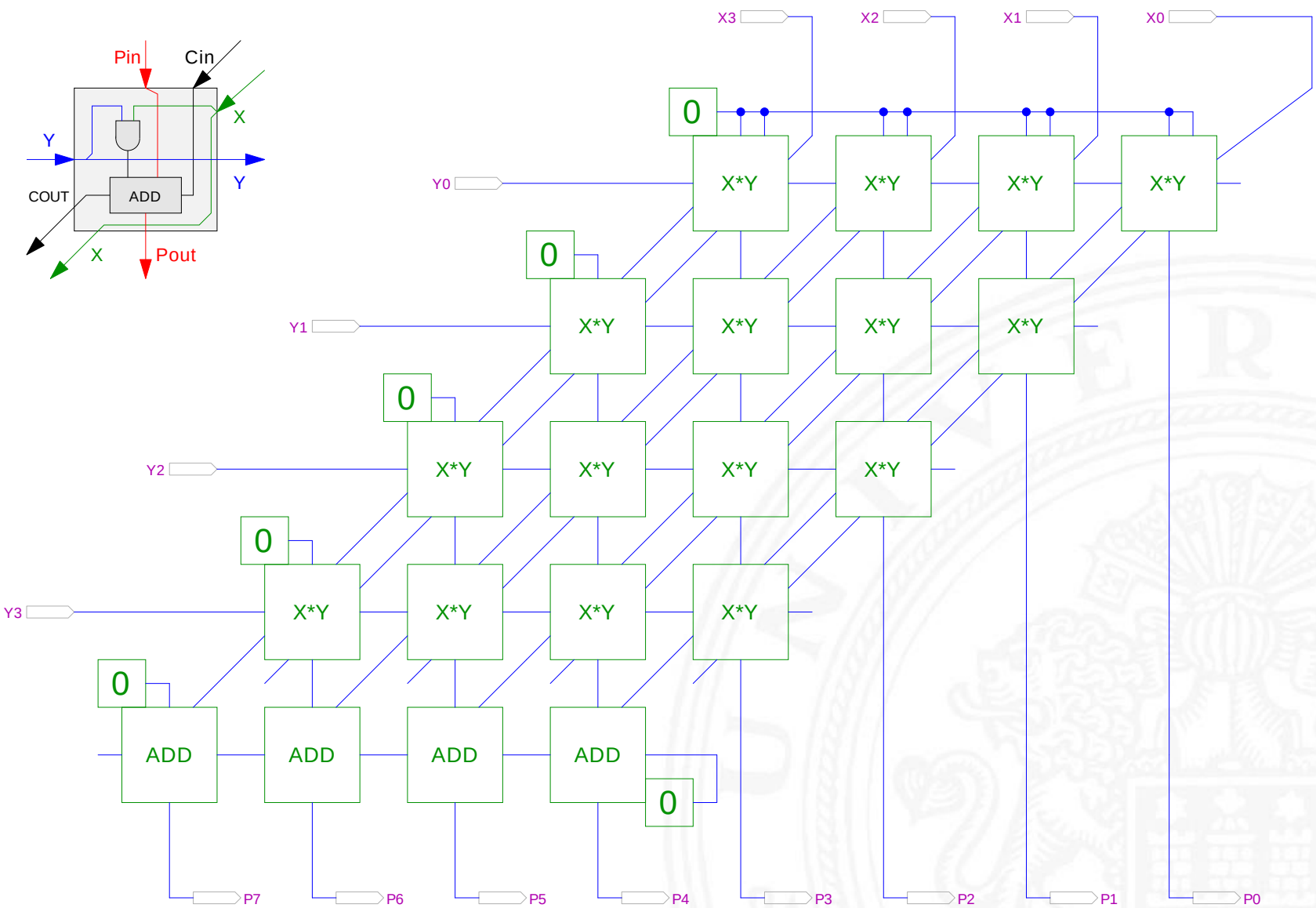
- alternativ: Schaltwerke (Automaten) mit sukzessiver Berechnung des Produkts in mehreren Takten durch Addition und Schieben

2x2-bit Multiplizierer – als zweistufiges Schaltnetz



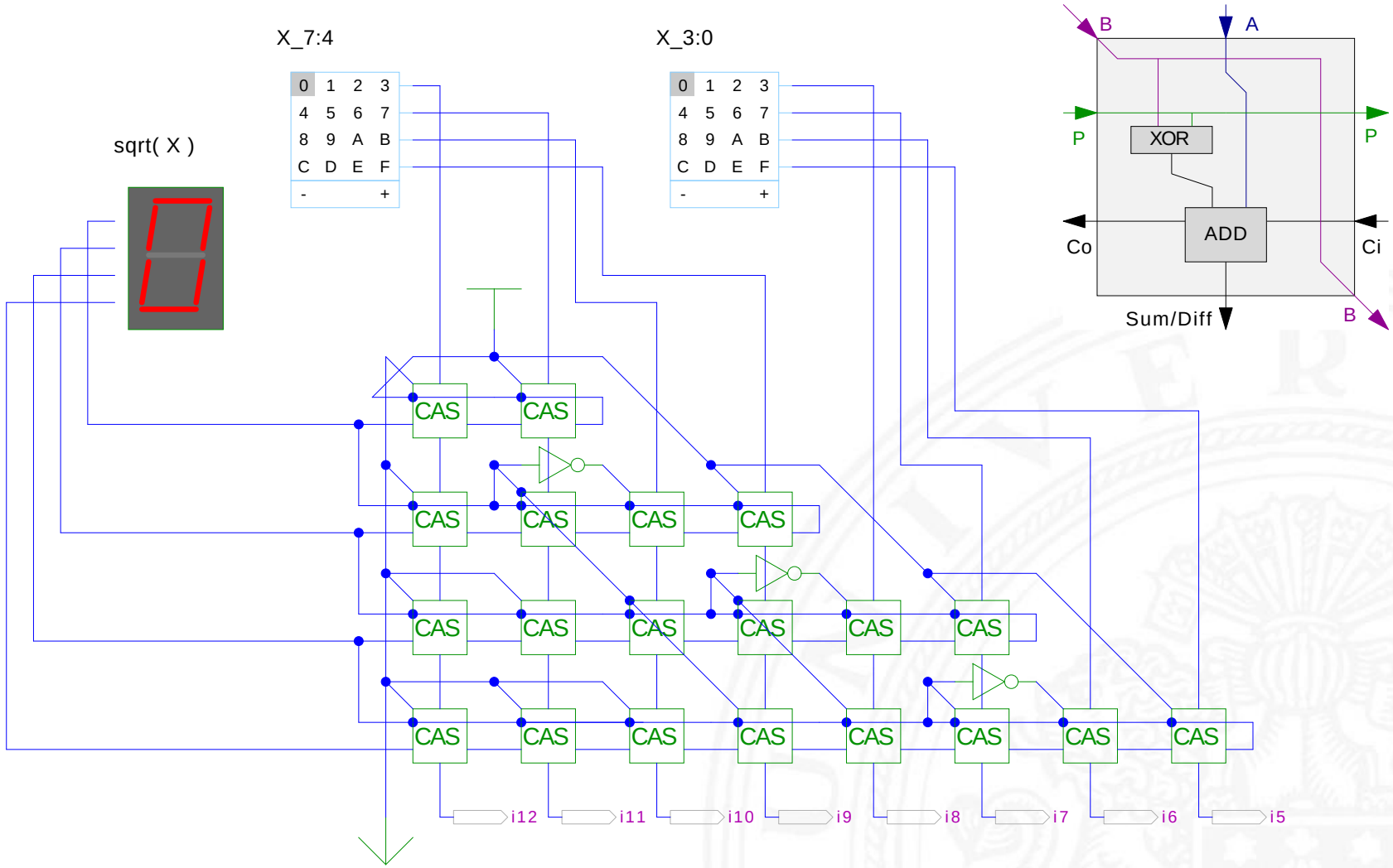
[HenHA] Hades Demo: `10-gates/13-mult2x2/mult2x2`

4x4-bit Multiplizierer – Array



[HenHA] Hades Demo: `20-arithmetic/60-mult/mult4x4`

4x4-bit Quadratwurzel



[HenHA] Hades Demo: `20-arithmetic/90-sqrt/sqrt4`

Multiplizierer

weitere wichtige Themen aus Zeitgründen nicht behandelt

- *Carry-Save Adder* zur Summation der Teilprodukte
- *Booth-Codierung* (effiziente Multiplikation)
- Multiplikation von Zweierkomplementzahlen
- Multiplikation von Gleitkommazahlen

- *CORDIC-Algorithmen*
  - Multiplikation, Division
  - iterative Berechnung höherer Funktionen: Exponentialfunktion, Logarithmus, trigonometrische Funktionen

- bei Interesse: Literatur anschauen [Omo94, Kor01, Spa76]

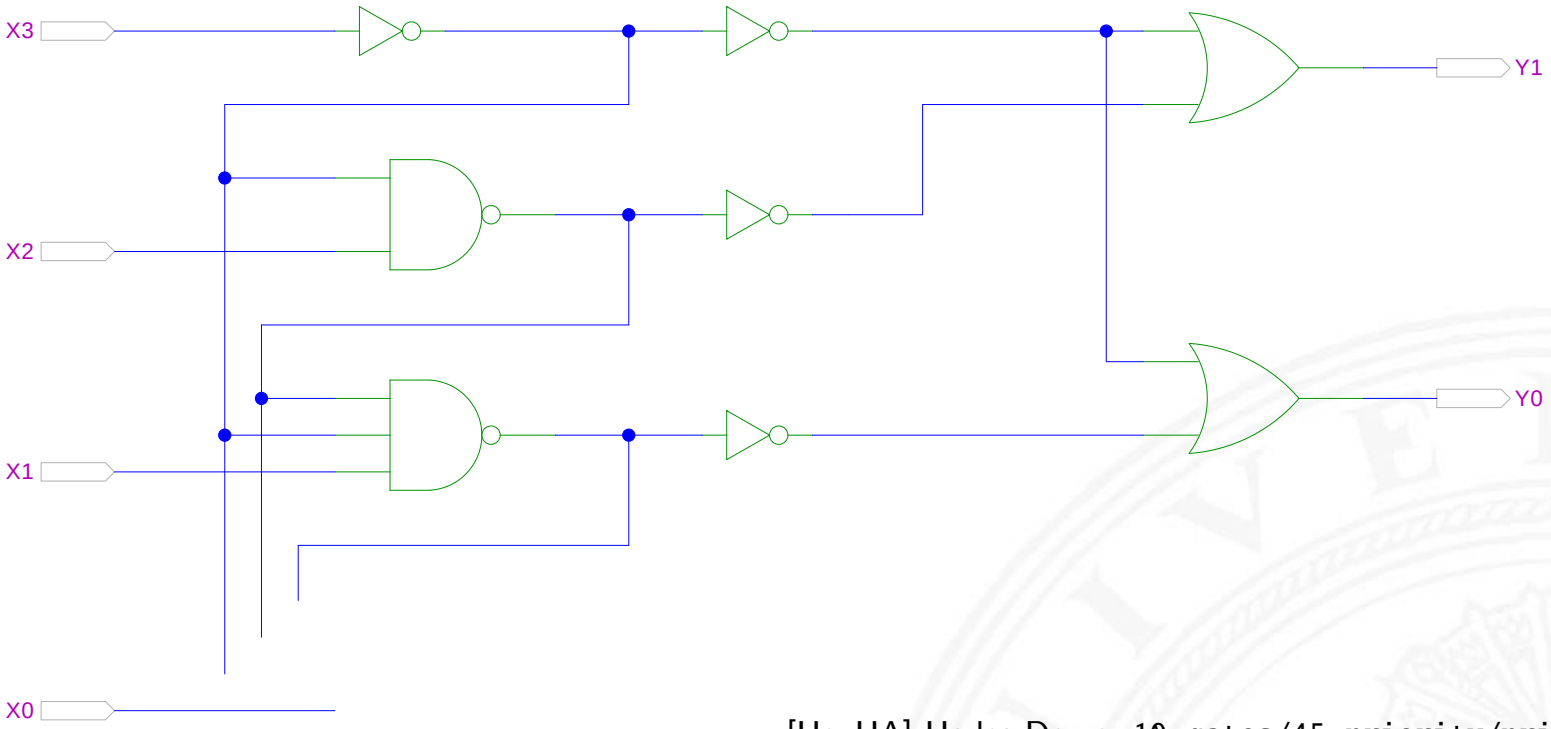
Priority Encoder

- Anwendung u.a. für Interrupt-Priorisierung
- Schaltung konvertiert n-bit Eingabe in eine Dualcodierung
- Wenn Bit n aktiv ist, werden alle niedrigeren Bits $(n-1), \ldots, 0$ ignoriert

x_3	x_2	x_1	x_0	y_1	y_0
1	*	*	*	1	1
0	1	*	*	1	0
0	0	1	*	0	1
0	0	0	*	0	0

- unabhängig von niederwertigstem Bit $\Rightarrow x_0$ kann entfallen

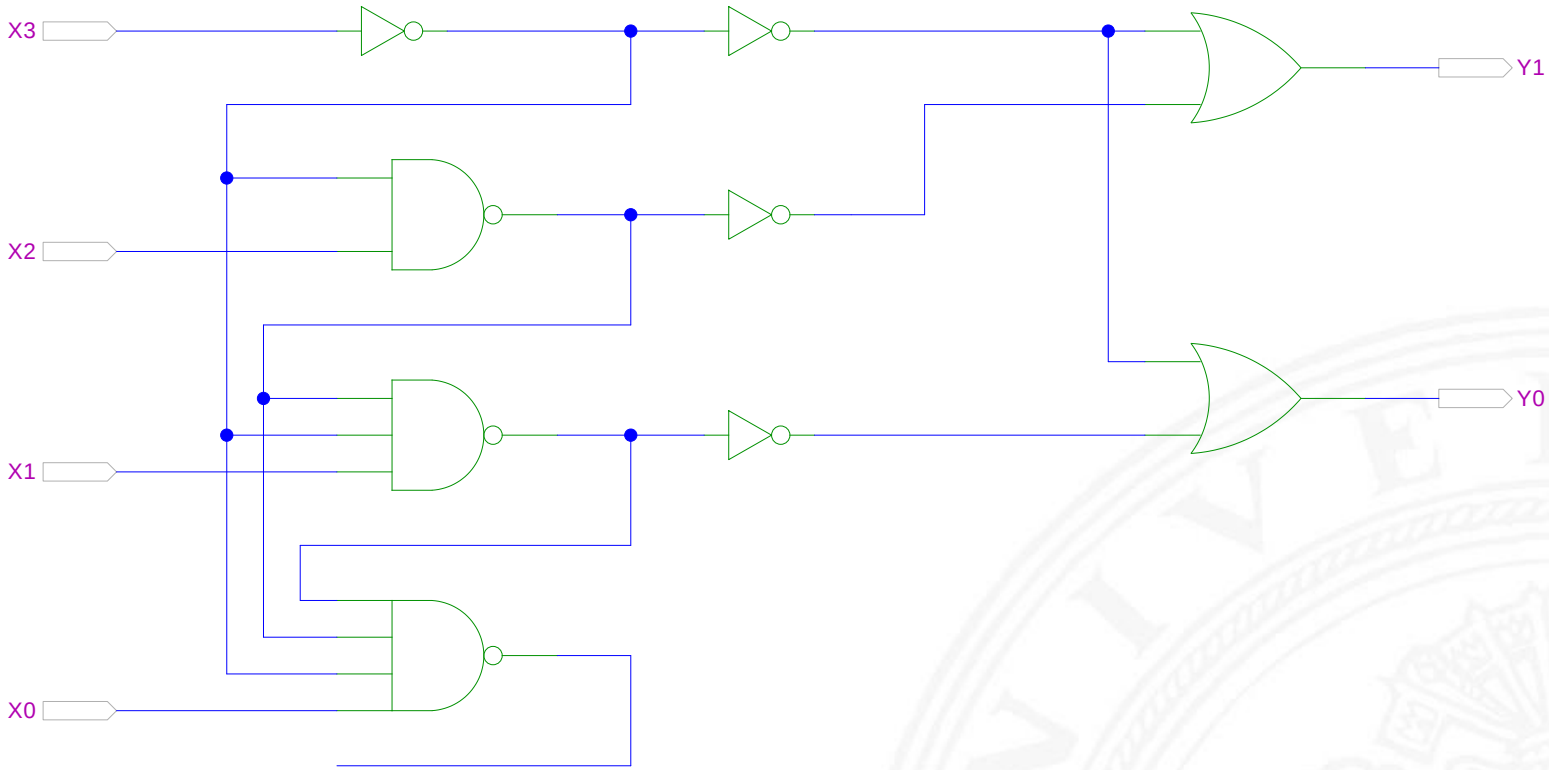
4:2 Prioritätsencoder



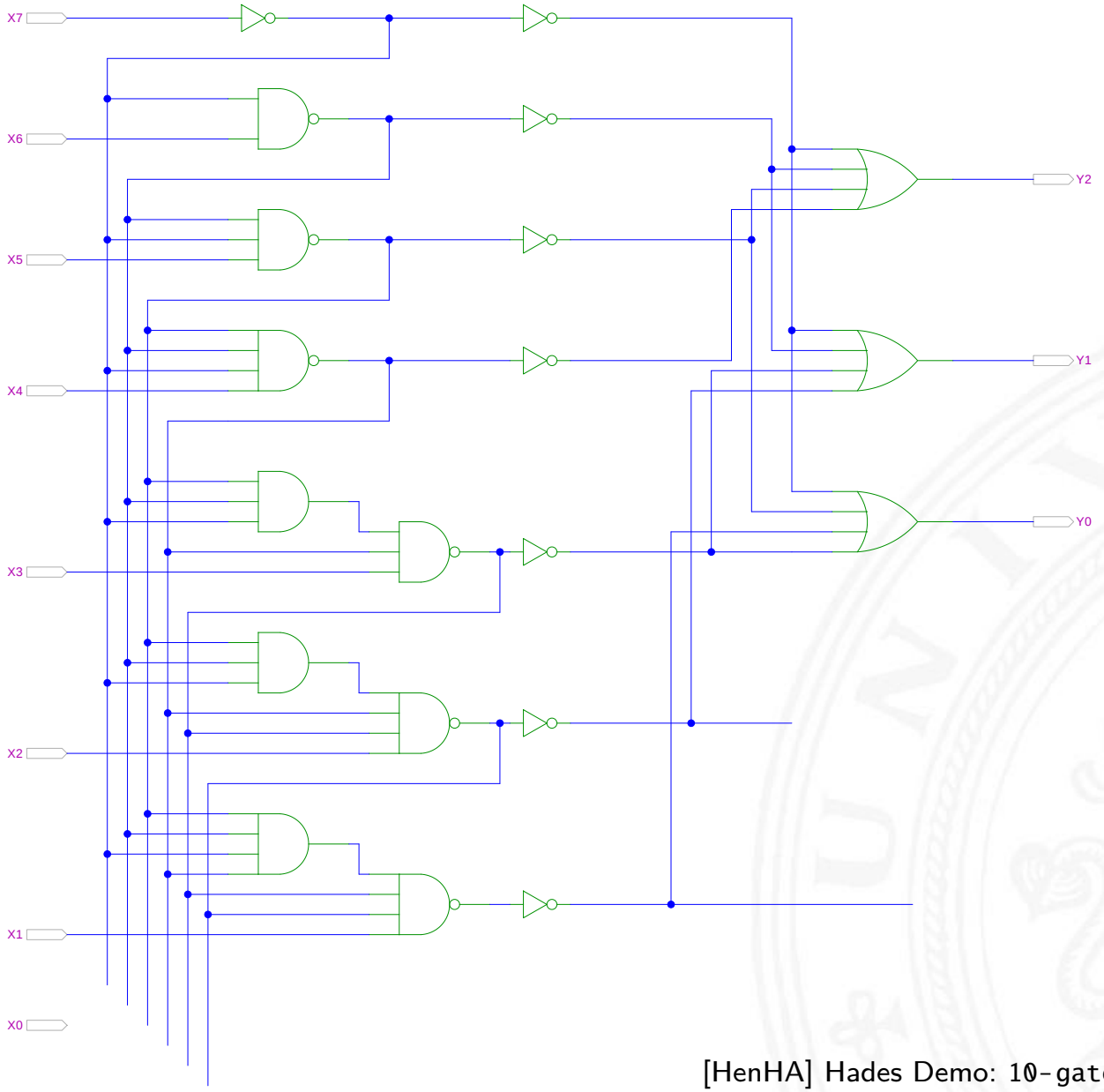
[HenHA] Hades Demo: `10-gates/45-priority/priority42`

- zweistufige Realisierung (Inverter ignoriert)
- aktive höhere Stufe blockiert alle niedrigeren Stufen

4:2 Prioritätsencoder: Kaskadierung



8:3 Prioritätsencoder



[HenHA] Hades Demo: `10-gates/45-priority/priority83`

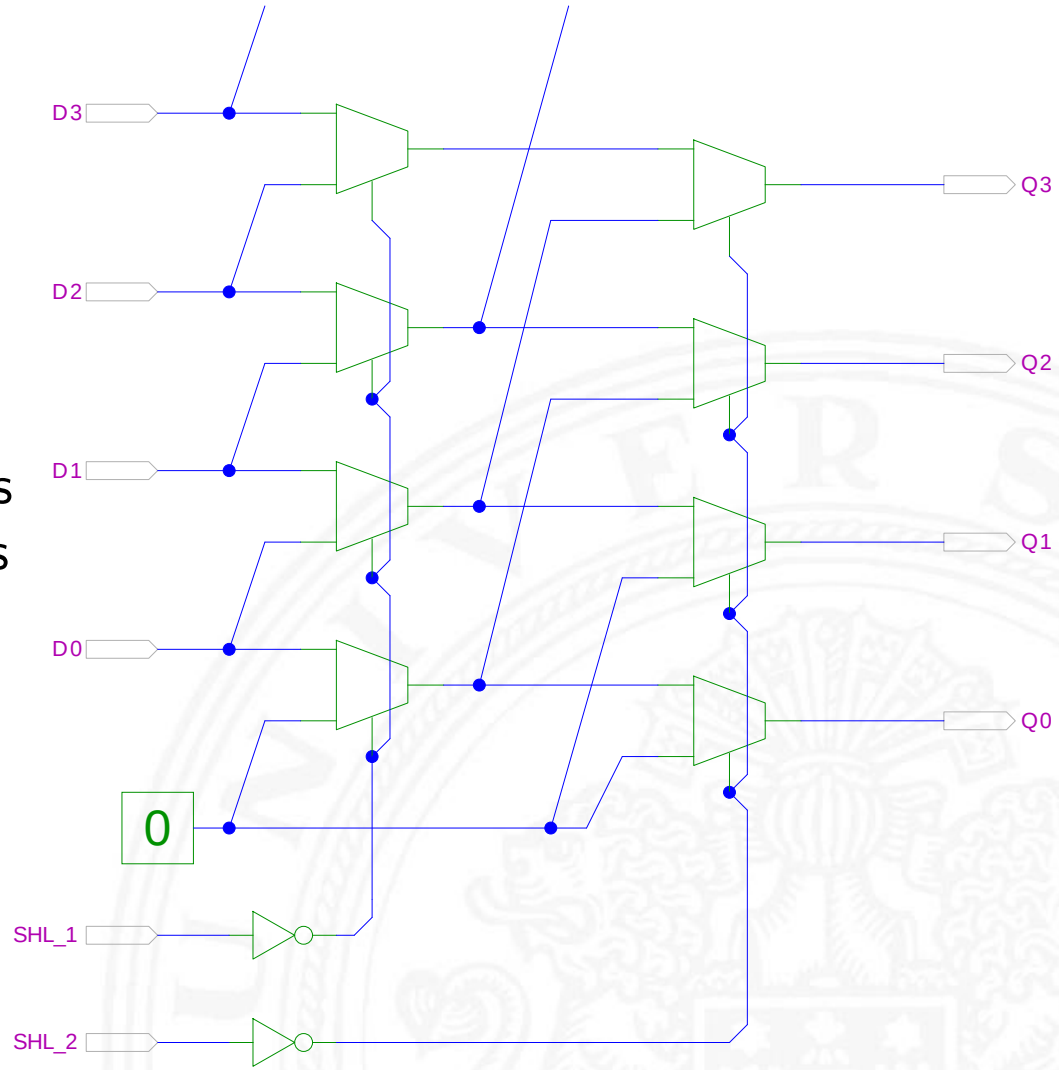
Shifter: zweistufig, shift-left um 0...3 Bits

- n-Dateneingänge D_i

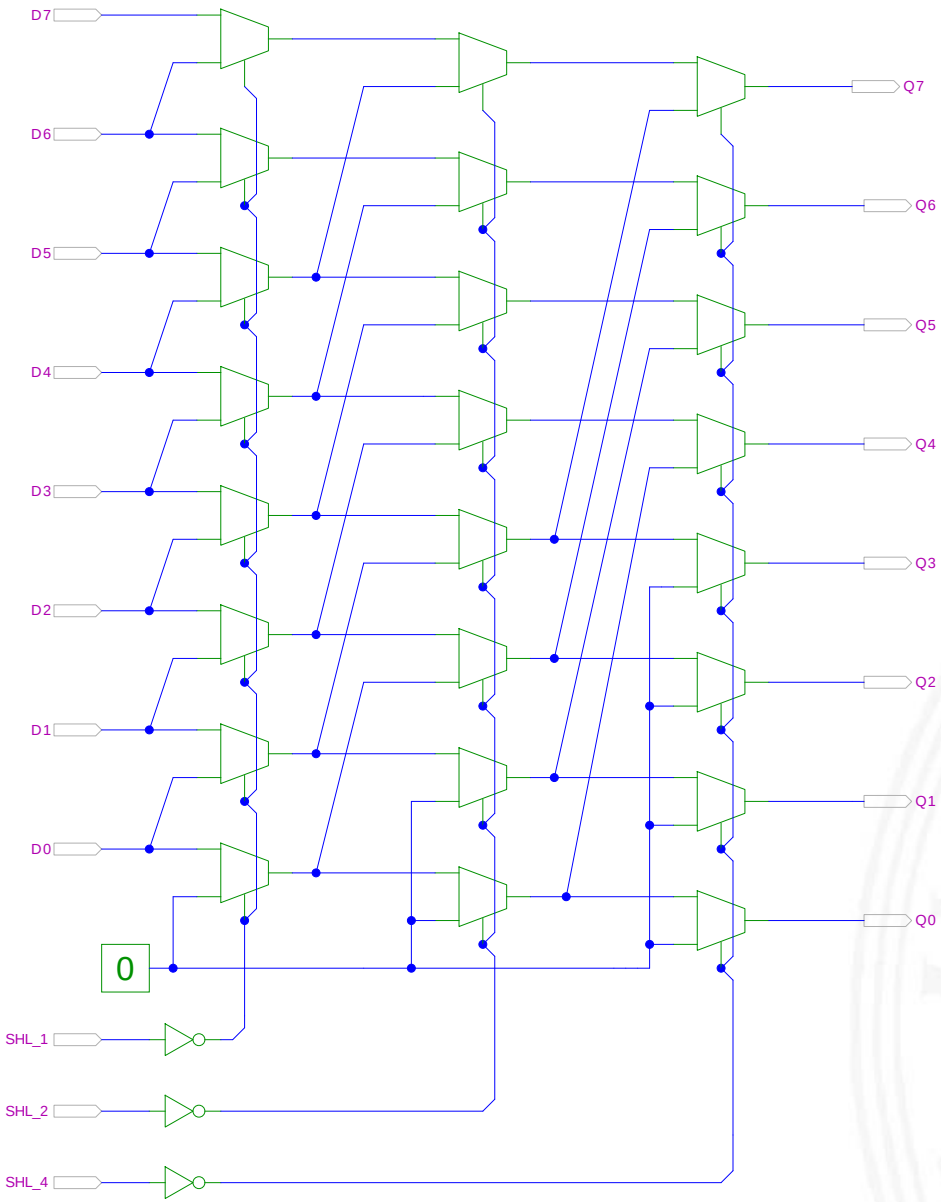
  n-Datenausgänge Q_i

- 2:1 Multiplexer Kaskade
  - Stufe 0: benachbarte Bits
  - Stufe 1: übernächste Bits
  - usw.

- von rechts 0 nachschieben



8-bit Barrel-Shifter



[HenHA] Hades Webdemo:
`10-gates/60-barrel/shifter8`

Shift-Right, Rotate etc.

- Prinzip der oben vorgestellten Schaltungen gilt auch für alle übrigen Shift- und Rotate-Operationen

- Logic shift right: von links Nullen nachschieben

  Arithmetic shift right: oberstes Bit nachschieben

- Rotate left / right: außen herausgeschobene Bits auf der anderen Seite wieder hineinschieben

\+ alle Operationen typischerweise in einem Takt realisierbar

– Problem: Hardwareaufwand bei großen Wortbreiten und beliebigem Schiebe-/Rotate-Argument

Arithmetisch-Logische Einheit (ALU)

Arithmetisch-logische Einheit ALU (*Arithmetic Logic Unit*)

- kombiniertes Schaltnetz für arithmetische und logische Operationen
- das zentrale Rechenwerk in Prozessoren

Funktionsumfang variiert von Typ zu Typ

- Addition und Subtraktion — 2-Komplement
- bitweise logische Operationen — Negation, UND, ODER, XOR
- Schiebeoperationen — shift, rotate
- evtl. Multiplikation

- Integer-Division selten verfügbar (separates Rechenwerk)

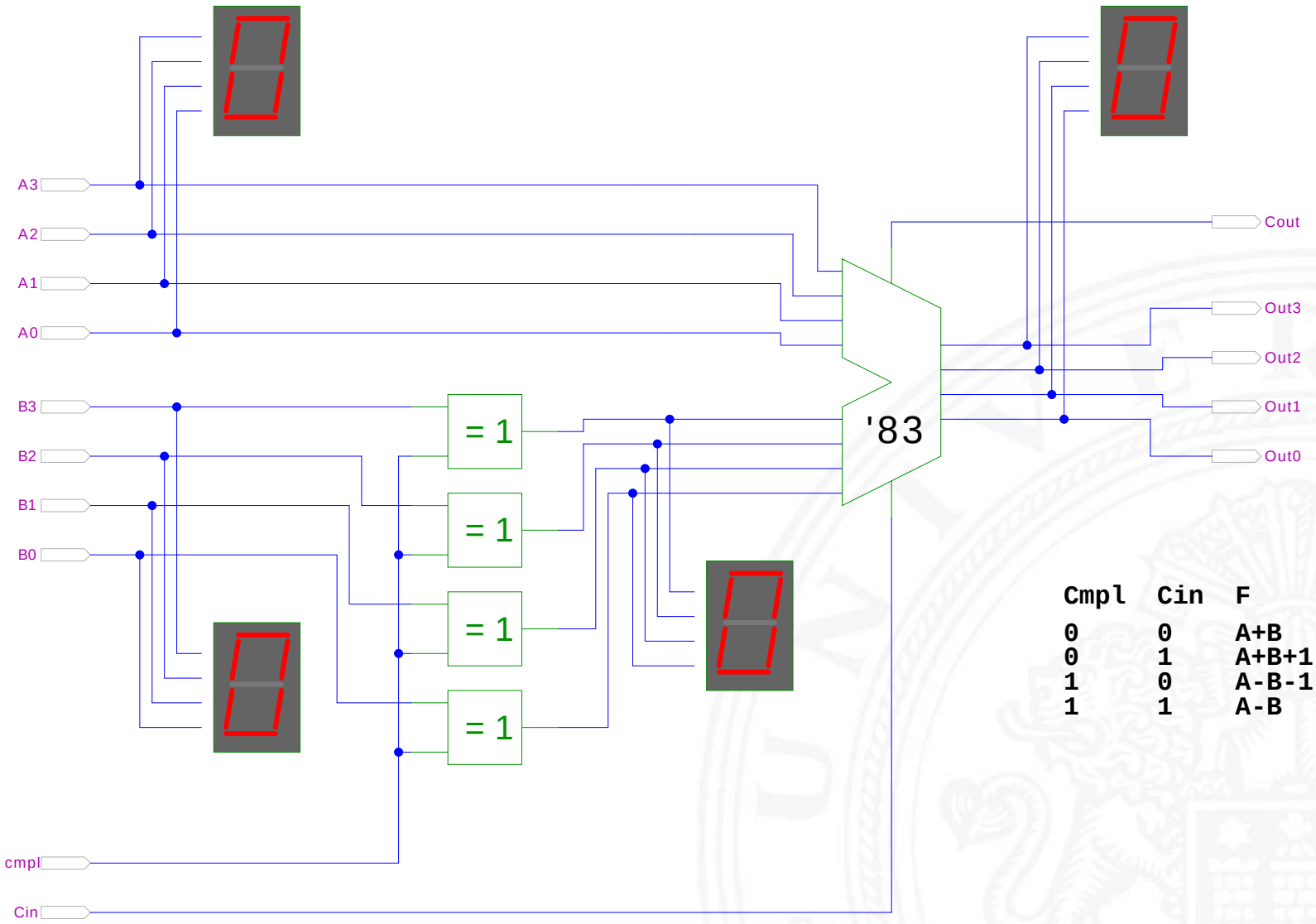
ALU: Addierer und Subtrahierer

- Addition $(A + B)$ mit normalem Addierer

- XOR-Gatter zum Invertieren von Operand B
- Steuerleitung sub aktiviert das Invertieren und den Carry-in c_i
- wenn aktiv, Subtraktion als $(A - B) = A + \overline{B} + 1$
- ggf. auch Inkrement $(A + 1)$ und Dekrement $(A - 1)$

- folgende Folien: 7483 ist IC mit 4-bit Addierer

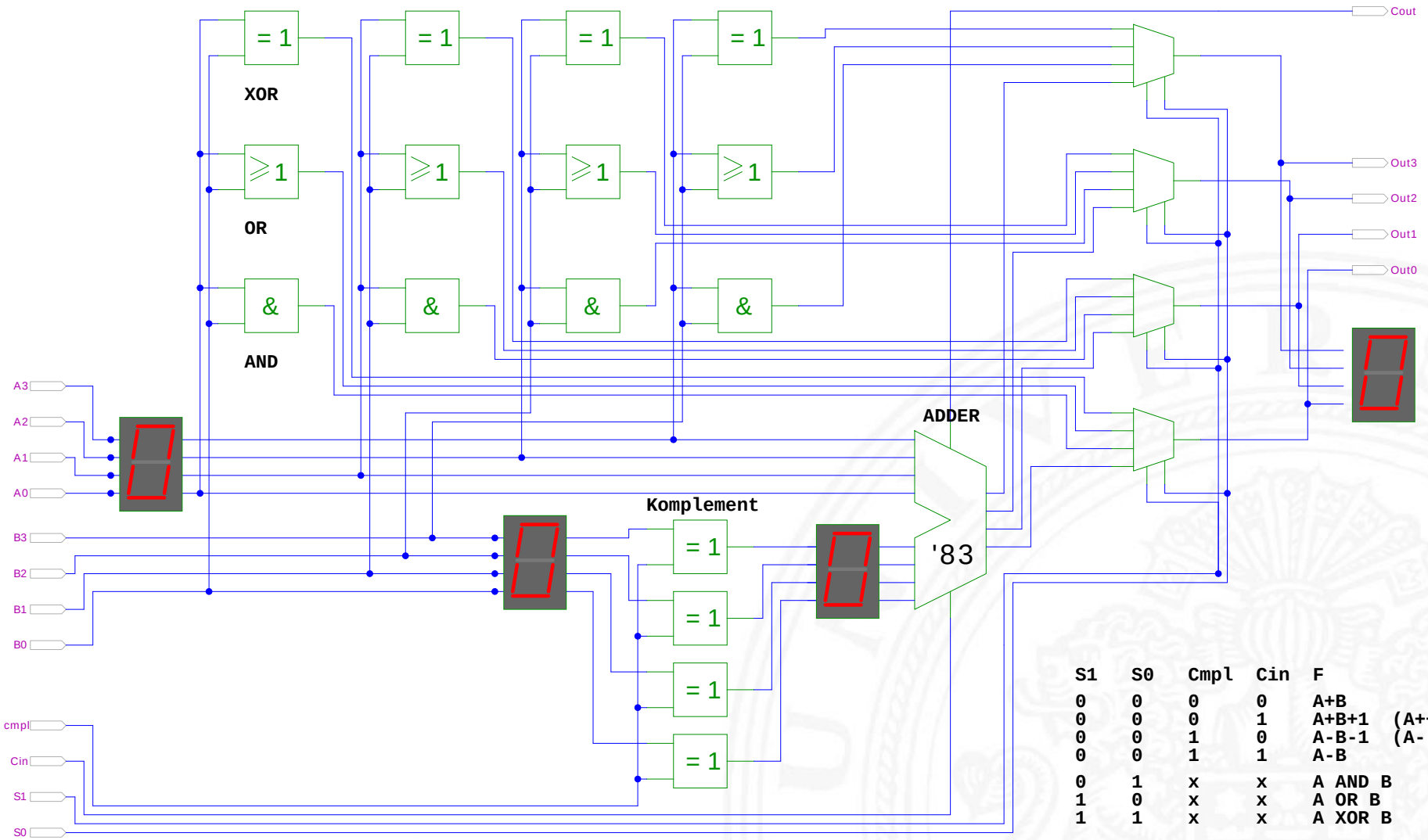
ALU: Addierer und Subtrahierer



Cmpl	Cin	F	
0	0	A+B	
0	1	A+B+1	(A++)
1	0	A-B-1	(A--)
1	1	A-B	

Schiffmann, Schmitz: *Technische Informatik I* [SS04]

ALU: Addierer und bitweise Operationen



S1	S0	Cmpl	Cin	F	
0	0	0	0	A+B	
0	0	0	1	A+B+1	(A++)
0	0	1	0	A-B-1	(A--)
0	0	1	1	A-B	
0	1	x	x	A AND B	
1	0	x	x	A OR B	
1	1	x	x	A XOR B	

Schiffmann, Schmitz: *Technische Informatik I* [SS04]

ALU: Prinzip

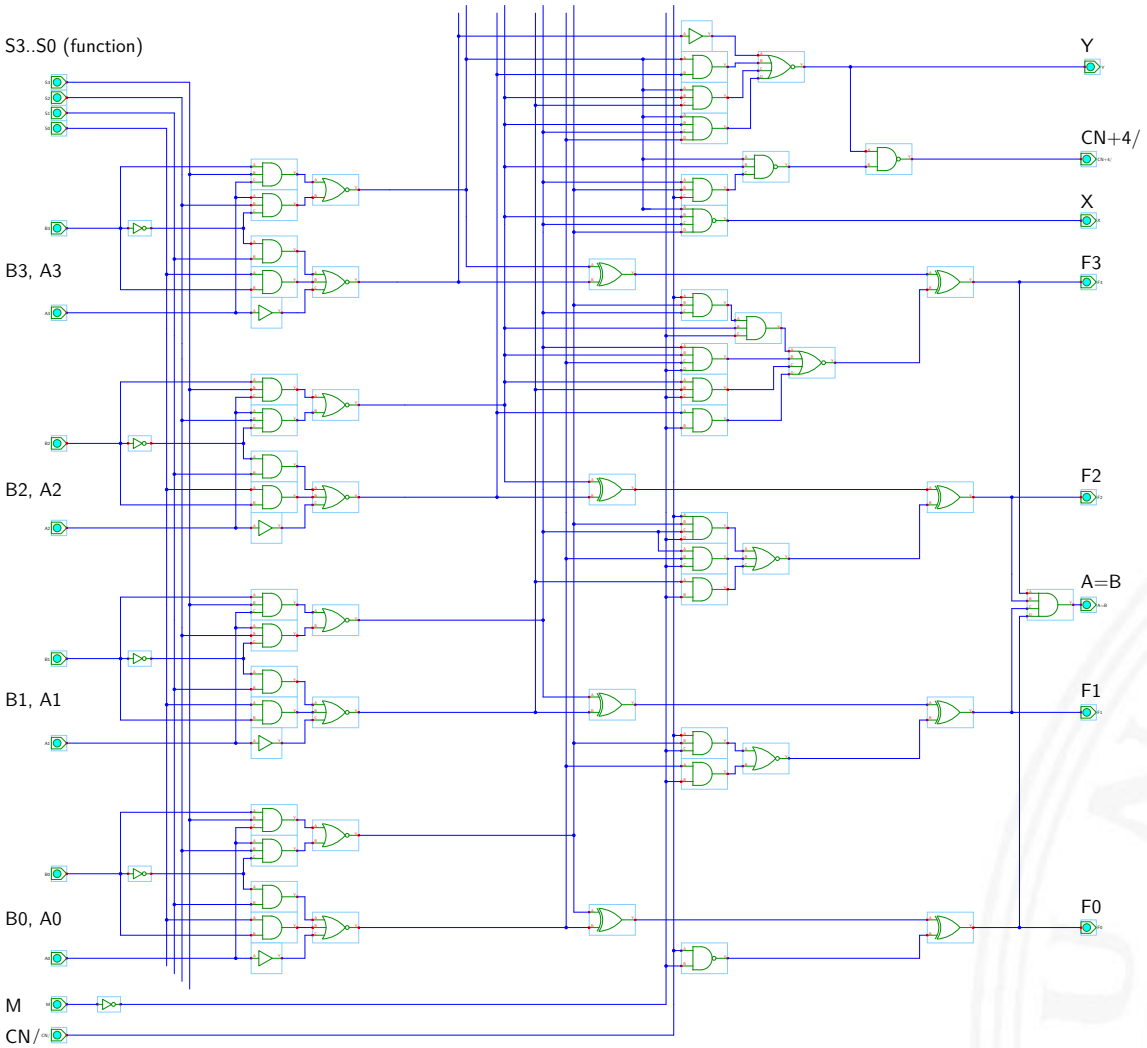
vorige Folie zeigt die „triviale“ Realisierung einer ALU

- mehrere parallele Rechenwerke für die m einzelnen Operationen
  n-bit Addierer, n-bit Komplement, n-bit OR usw.
- Auswahl des Resultats über n-bit m:1-Multiplexer

nächste Folie: Realisierung in der Praxis (IC 74181)

- erste Stufe für bitweise logische Operationen und Komplement
- zweite Stufe als Carry-Lookahead Addierer
- weniger Gatter und schneller

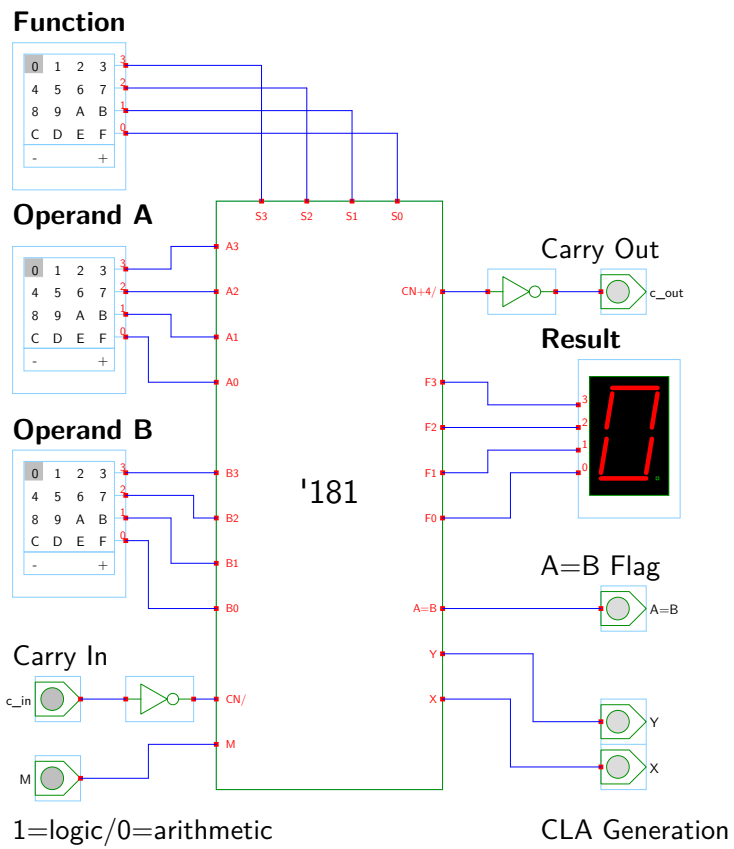
ALU: 74181 – Aufbau



selection				logic functions	arithmetic functions
S3	S2	S1	S0	M = 1	M = 0 Cn = 1 (no carry)
0	0	0	0	F = !A	F = A
0	0	0	1	F = !(A or B)	F = A or B
0	0	1	0	F = !A and B	F = A or !B
0	0	1	1	F = !A and B	F = -1
0	1	0	0	F = 0	F = A + (A and !B)
0	1	0	1	F = !B	F = (A or B) + (A and !B)
0	1	1	0	F = A xor B	F = A - B - 1
0	1	1	1	F = A and !B	F = (A and !B) - 1
1	0	0	0	F = !A or B	F = A + (A and B)
1	0	0	1	F = A xnor B	F = A + B
1	0	1	0	F = B	F = (A or !B) + (A and B)
1	0	1	1	F = A and B	F = (A and B) - 1
1	1	0	0	F = 1	F = A + A
1	1	0	1	F = A or !B	F = (A or B) + A
1	1	1	0	F = A or B	F = (A or !B) + A
1	1	1	1	F = A	F = A - 1
					F + 1 Cn = 0 (carry in)

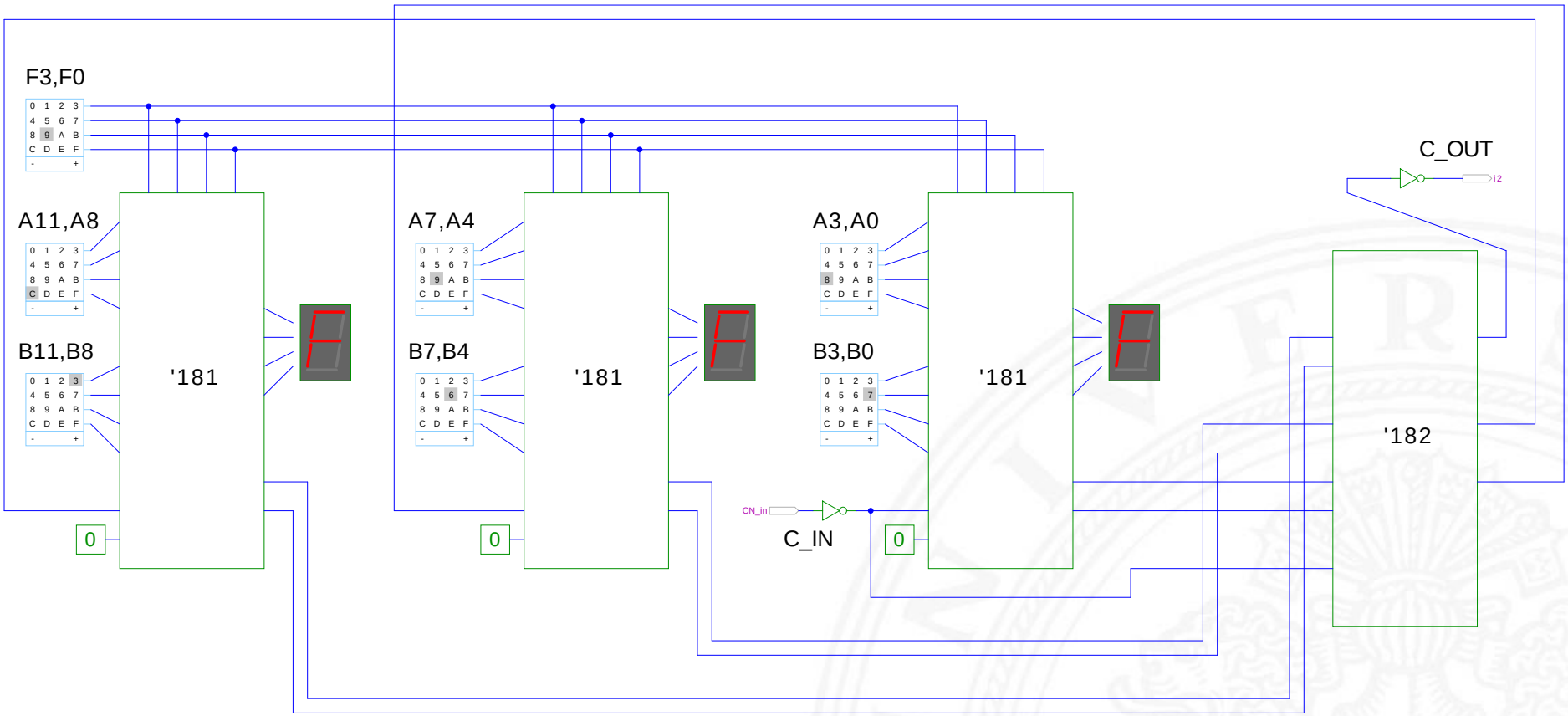
[HenHA] Hades Demo: `20-arithmetic/50-74181/SN74181`

ALU: 74181 – Funktionstabelle



selection				logic functions	arithmetic functions
S3	S2	S1	S0	M = 1	M = 0 Cn = 1 (no carry)
0	0	0	0	F = !A	F = A
0	0	0	1	F = !(A or B)	F = A or B
0	0	1	0	F = !A and B	F = A or !B
0	0	1	1	F = !A and B	F = -1
0	1	0	0	F = 0	F = A + (A and !B)
0	1	0	1	F = !B	F = (A or B) + (A and !B)
0	1	1	0	F = A xor B	F = A - B - 1
0	1	1	1	F = A and !B	F = (A and !B) - 1
1	0	0	0	F = !A or B	F = A + (A and B)
1	0	0	1	F = A xnor B	F = A + B
1	0	1	0	F = B	F = (A or !B) + (A and B)
1	0	1	1	F = A and B	F = (A and B) - 1
1	1	0	0	F = 1	F = A + A
1	1	0	1	F = A or !B	F = (A or B) + A
1	1	1	0	F = A or B	F = (A or !B) + A
1	1	1	1	F = A	F = A - 1
					F + 1 Cn = 0 (carry in)

[HenHA] Hades Demo: `20-arithmetic/50-74181/demo-74181-ALU`



[HenHA] Hades Demo: `20-arithmetic/50-74181/demo-74182-ALU-CLA`

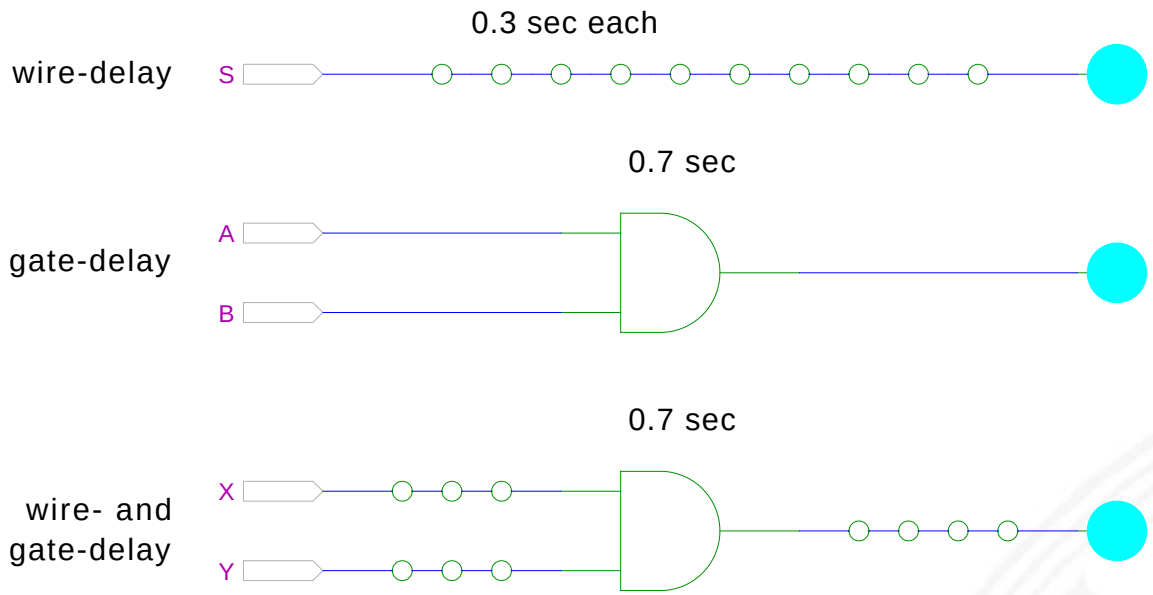
Zeitverhalten einer Schaltung: Modellierung

Wie wird das Zeitverhalten eines Schaltnetzes modelliert?

Gängige Abstraktionsebenen mit zunehmendem Detaillierungsgrad

1. algebraische Ausdrücke: keine zeitliche Abhängigkeit
2. „fundamentales Modell“: Einheitsverzögerung des algebraischen Ausdrucks um eine Zeit τ
3. individuelle Gatterverzögerungen
   - mehrere Modelle, unterschiedlich detailliert
   - Abstraktion elektrischer Eigenschaften
4. Gatterverzögerungen + Leitungslaufzeiten (geschätzt, berechnet)
5. Differentialgleichungen für Spannungen und Ströme (verschiedene „Ersatzmodelle“)

Gatterverzögerung vs. Leitungslaufzeiten

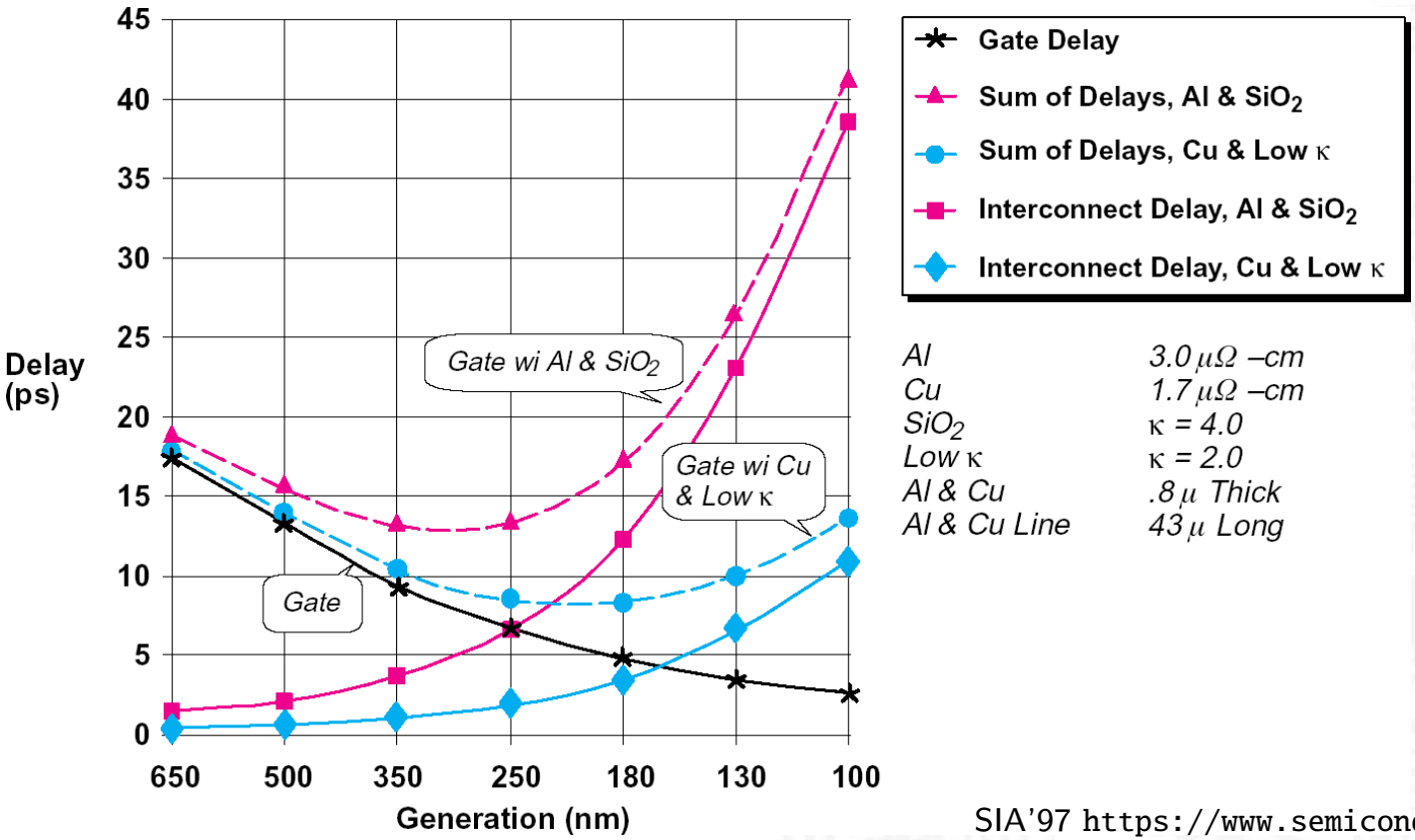


[HenHA] Hades Demo: `12-gatedelay/10-delaydemo/gate-vs-wire-delay`

- früher: Gatterverzögerungen $\gg$ Leitungslaufzeiten
- Schaltungen modelliert durch Gatterlaufzeiten
- aktuelle „Submicron“-Halbleitertechnologie:
  Leitungslaufzeiten $\gg$ Gatterverzögerungen

Gatterverzögerung vs. Leitungslaufzeiten (cont.)

- Leitungslaufzeiten
  - lokale Leitungen: schneller (weil Strukturen kleiner)
  - globale Leitungen: langsamer
  - nicht mehr alle Punkte des Chips in einem Taktzyklus erreichbar



SIA'97 https://www.semiconductors.org

Zeitverhalten

- alle folgenden Schaltungsbeispiele werden mit Gatterverzögerungen modelliert (einfacher Handhabbar)
- Gatterlaufzeiten als Vielfache einer Grundverzögerung (τ)
- aber Leitungslaufzeiten ignoriert

- mögliche Verfeinerungen
  - gatterabhängige Schaltzeiten für INV, NAND, NOR, XOR etc.
  - unterschiedliche Schaltzeiten für Wechsel: $0 \rightarrow 1$ und $1 \rightarrow 0$
  - unterschiedliche Schaltzeiten für 2-, 3-, 4-Input Gatter
  - Schaltzeiten sind abhängig von der Anzahl nachfolgender Eingänge (engl. *fanout*)

Exkurs: Lichtgeschwindigkeit und Taktraten

- Lichtgeschwindigkeit im Vakuum: $c \approx 300\,000\,\text{km/sec}$
  $\approx 30\,\text{cm/ns}$
- in Metallen und Halbleitern langsamer: $c \approx 20\,\text{cm/ns}$

$\Rightarrow$ bei 1 Gigahertz Takt: Ausbreitung um ca. 20 Zentimeter

Abschätzungen:

- Prozessor: ca. 3 cm Diagonale $<$ 10 GHz Taktrate
- Platine: ca. 20 cm Kantenlänge $<$ 1 GHz Takt

$\Rightarrow$ prinzipiell kann (schon heute) ein Signal innerhalb eines Takts nicht von einer Ecke des ICs zur Anderen gelangen

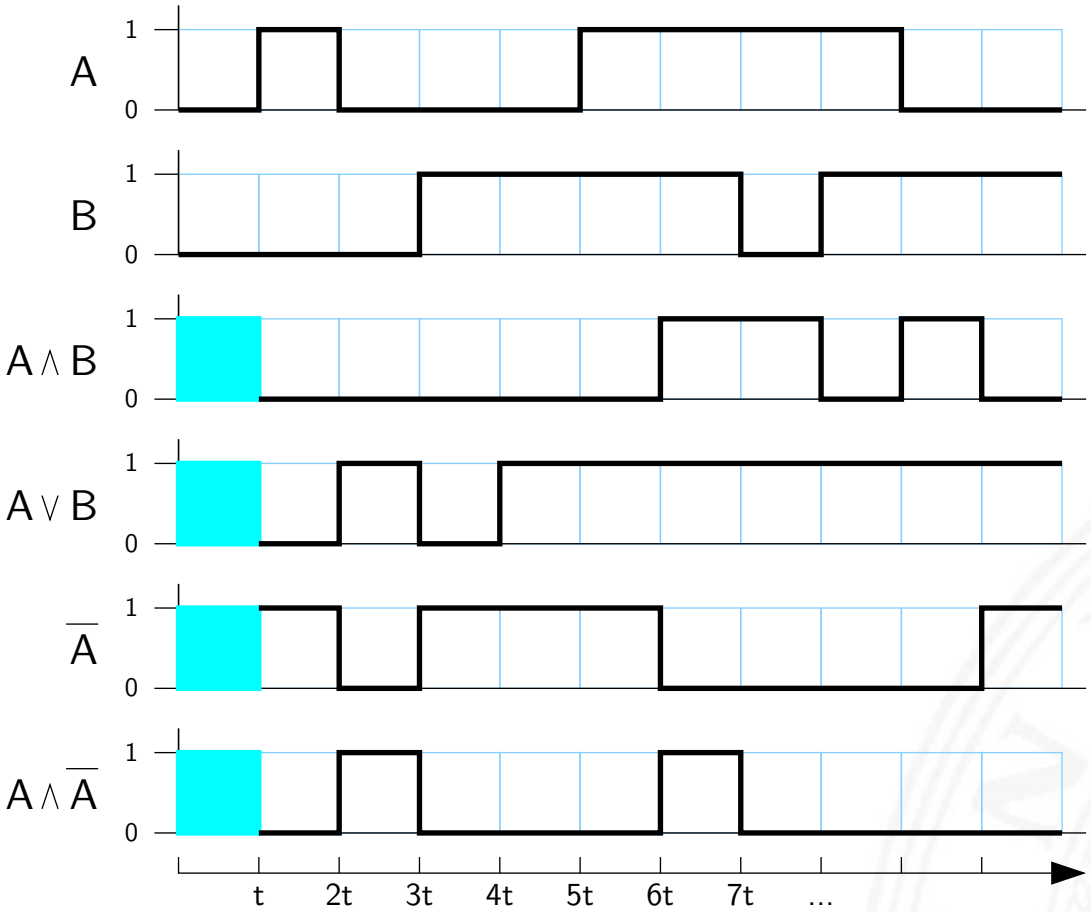
Impulsdiagramme

- **Impulsdiagramm** (engl. *waveform*): Darstellung der logischen Werte einer Schaltfunktion als Funktion der Zeit

- als Abstraktion des tatsächlichen Verlaufs
- Zeit läuft von links nach rechts
- Schaltfunktion(en): von oben nach unten aufgelistet

- Vergleichbar den Messwerten am Oszilloskop (analoge Werte) bzw. den Messwerten am Logic-State-Analyzer (digitale Werte)
- ggf. Darstellung mehrerer logischer Werte (z.B. 0,1,Z,U,X)

Impulsdiagramm: Beispiel



- im Beispiel jeweils eine „Zeiteinheit“ Verzögerung für jede einzelne logische Operation
- Ergebnis einer Operation nur, wenn die Eingaben definiert sind
- im ersten Zeitschritt noch undefinierte Werte

Hazards

- **Hazard**: die Eigenschaft einer Schaltfunktion, bei bestimmten Kombinationen der individuellen Verzögerungen ihrer Verknüpfungsglieder ein Fehlverhalten zu zeigen
- engl. auch *Glitch*

- **Hazardfehler**: das aktuelle Fehlverhalten einer realisierten Schaltfunktion aufgrund eines Hazards

Hazards: Klassifikation

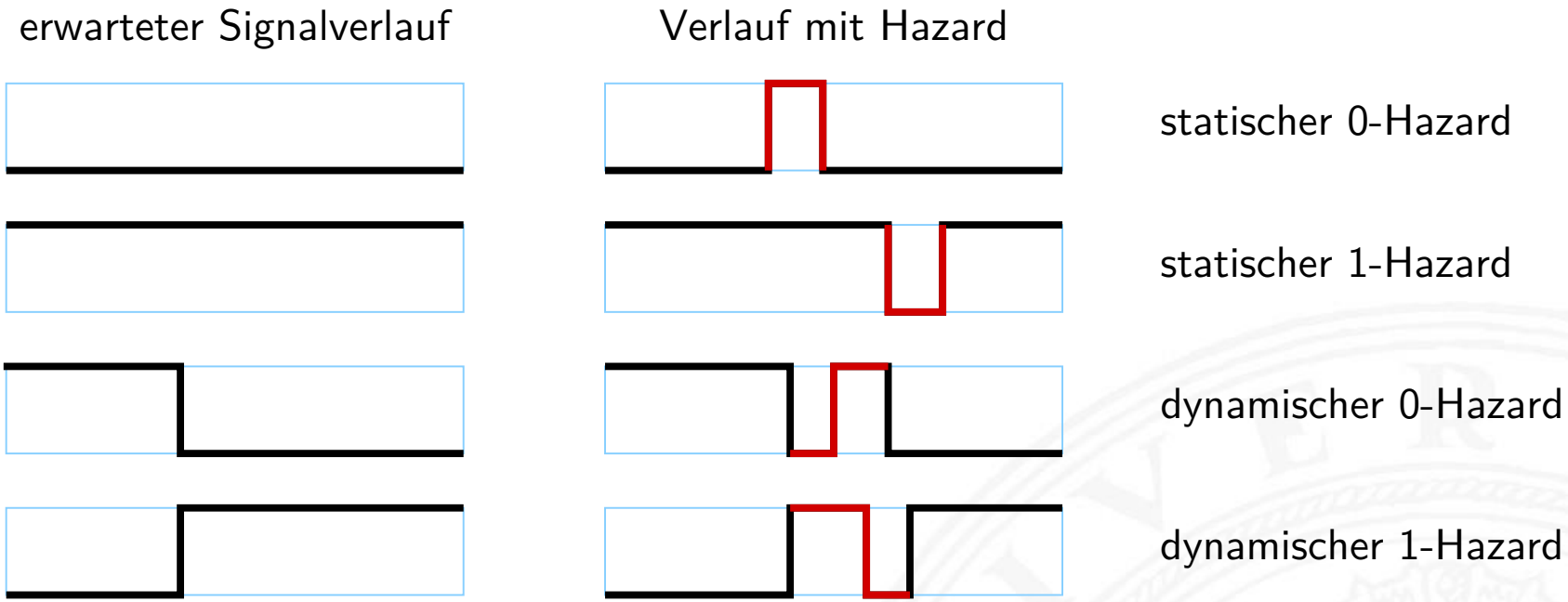
nach der Erscheinungsform am Ausgang

- **statisch**: der Ausgangswert soll unverändert sein, es tritt aber ein Wechsel auf
- **dynamisch**: der Ausgangswert soll (einmal) wechseln, es tritt aber ein mehrfacher Wechsel auf

nach den Eingangsbedingungen, unter denen der Hazard auftritt

- **Strukturhazard**: bedingt durch die Struktur der Schaltung, auch bei Umschalten eines einzigen Eingangswertes
- **Funktionshazard**: bedingt durch die Funktion der Schaltung

Hazards: statisch vs. dynamisch

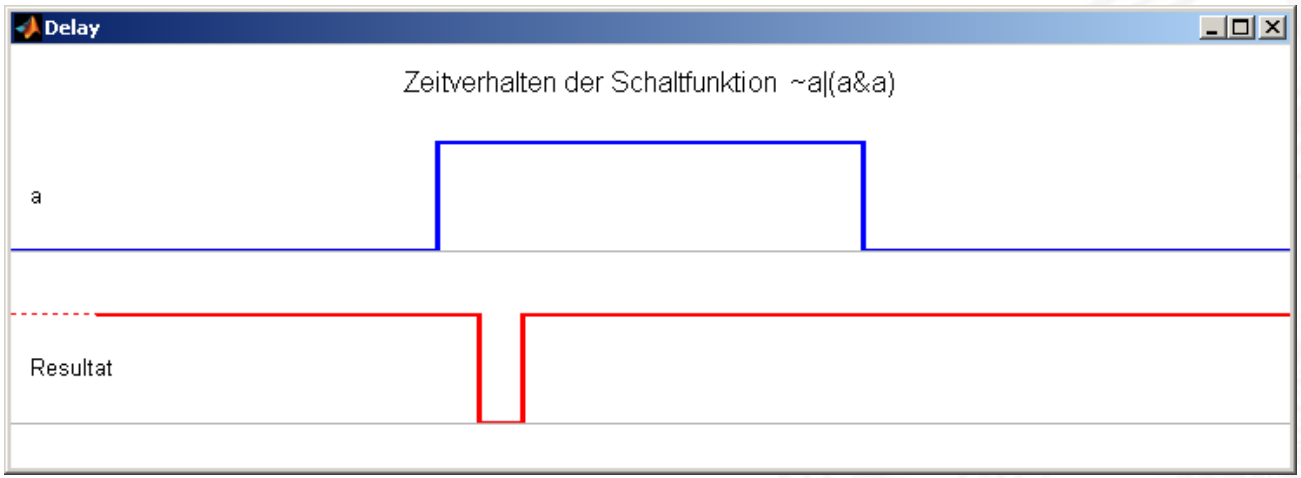


! Begriffsbildung in der Literatur nicht einheitlich:
0 / 1 als „richtiger Wert“
–"– „fehlerhafter Wert“

- 0-Hazard wenn der Wert 0 ausgegeben werden soll, zwichenzeitlich aber 1 erscheint (und umgekehrt)
- statisch oder dynamisch (dann auch 1-0, bzw. 0-1 Hazard)
- es können natürlich auch mehrfache Hazards auftreten

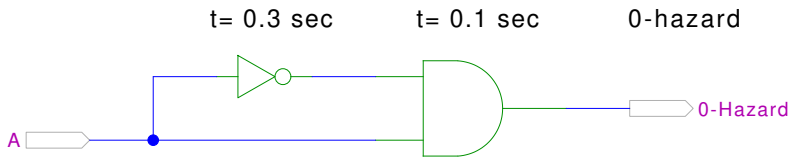
Hazards: Strukturhazard

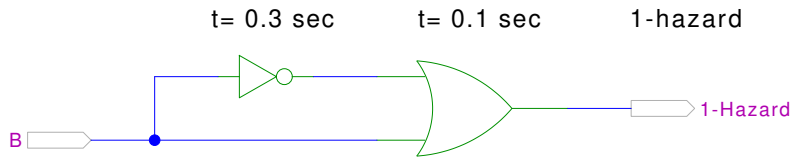
- **Strukturhazard** wird durch die gewählte Struktur der Schaltung verursacht
- auch, wenn sich nur eine Variable ändert
- Beispiel: $f(a) = \overline{a} \vee (a \wedge a)$
  $\overline{a}$ schaltet schneller ab, als $(a \wedge a)$ einschaltet

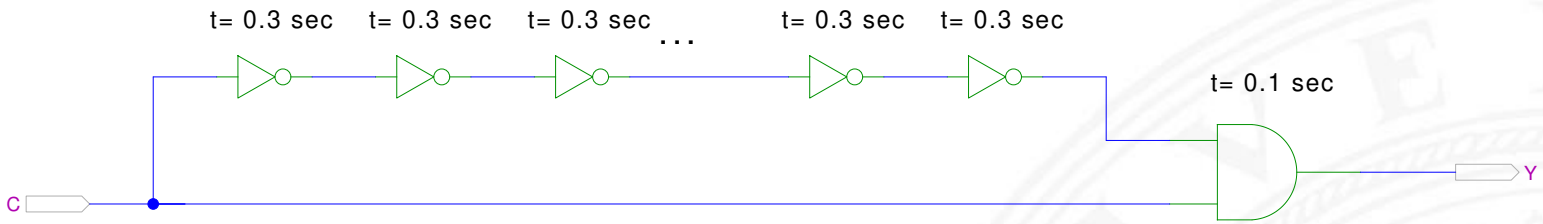


- Hazard kann durch Modifikation der Schaltung beseitigt werden
  im Beispiel mit: $f(a) = 1$

Strukturhazards: Beispiele





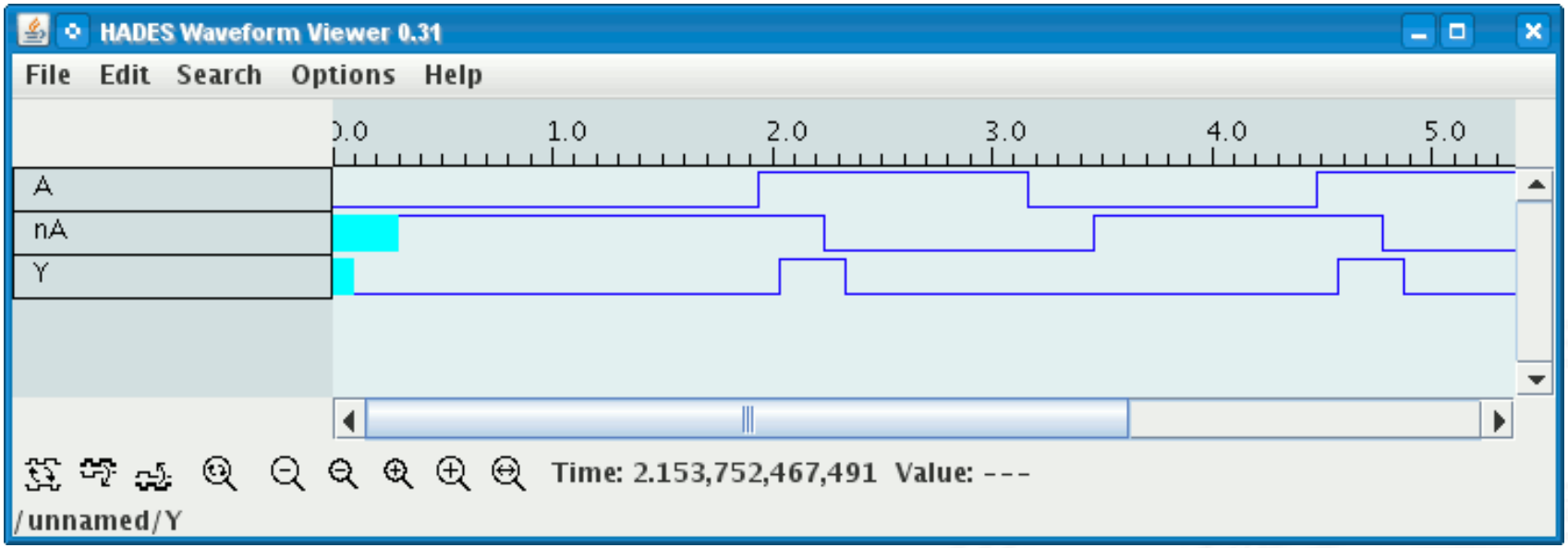


[HenHA] Hades Demo: `12-gatedelay/30-hazards/padding`

- logische Funktion ist $(a \wedge \overline{a}) = 0$ bzw. $(a \vee \overline{a}) = 1$
- aber ein Eingang jeweils durch Inverter verzögert

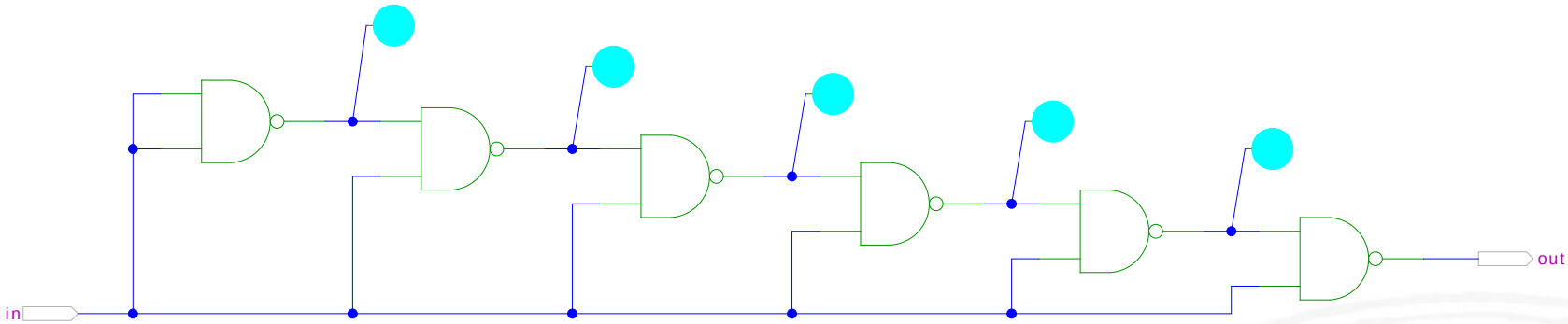
$\Rightarrow$ kurzer Impuls beim Umschalten von $0 \rightarrow 1$ bzw. $1 \rightarrow 0$

Strukturhazards: Beispiele (cont.)



- Schaltung $(a \wedge \overline{a}) = 0$ erzeugt (statischen-0) Hazard
- Länge des Impulses abhängig von Verzögerung im Inverter
- Kette von Invertern erlaubt Einstellung der Pulslänge

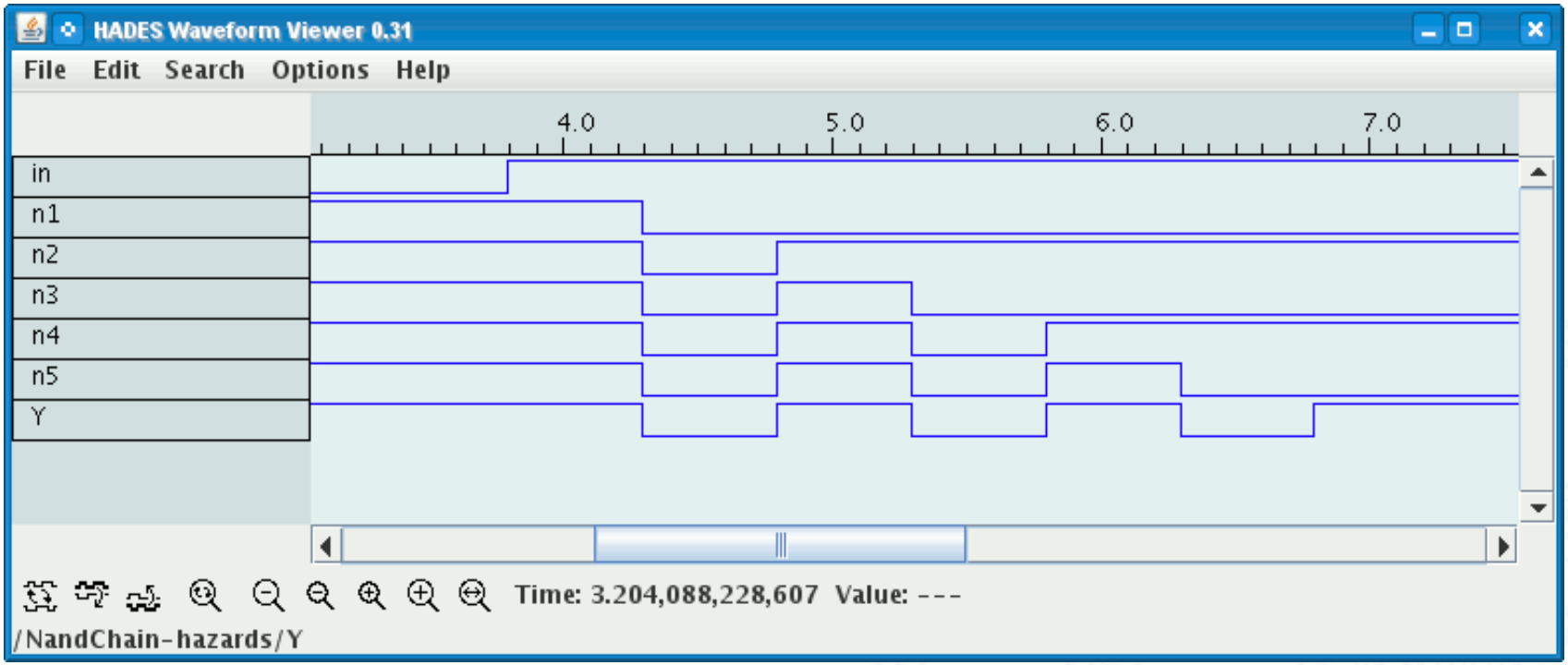
Strukturhazards extrem: NAND-Kette



[HenHA] Hades Demo: `12-gatedelay/30-hazards/nandchain`

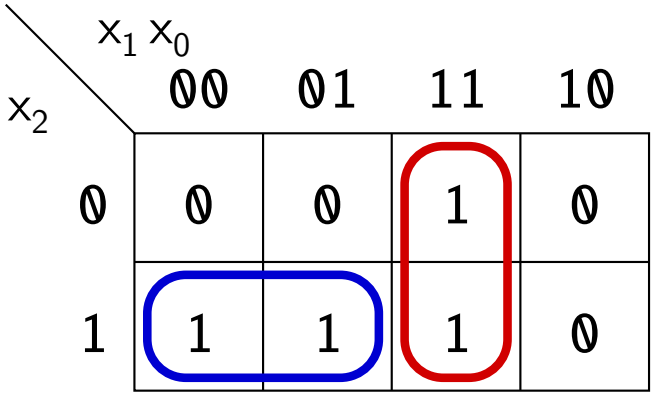
- alle NAND-Gatter an Eingang in angeschlossen
- $in = 0$ erzwingt $y_i = 1$
- Übergang in von 0 auf 1 startet Folge von Hazards

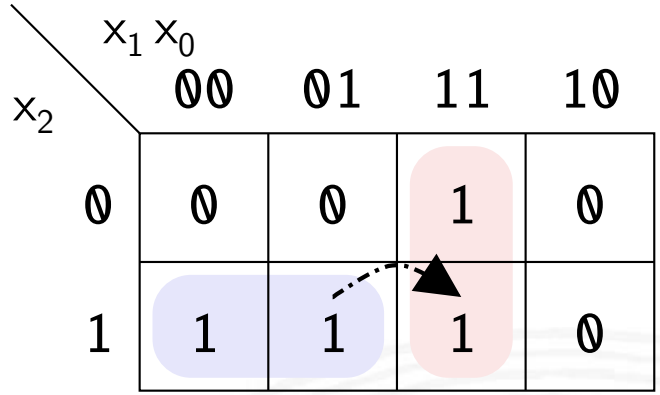
Strukturhazards extrem: NAND-Kette (cont.)



- Schaltung erzeugt Folge von (statischen-1) Hazards
- Anzahl der Impulse abhängig von Anzahl der Gatter

Strukturhazards im KV-Diagramm



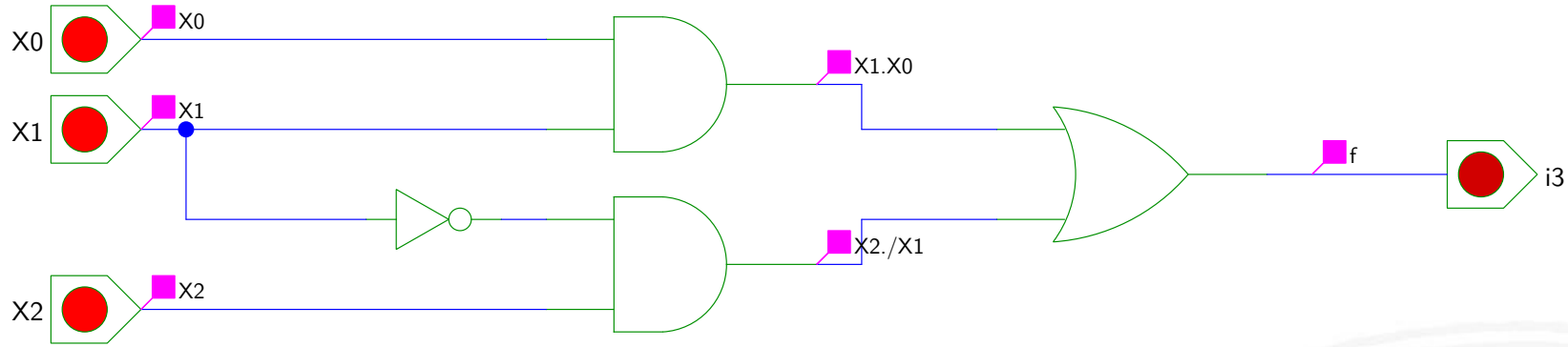


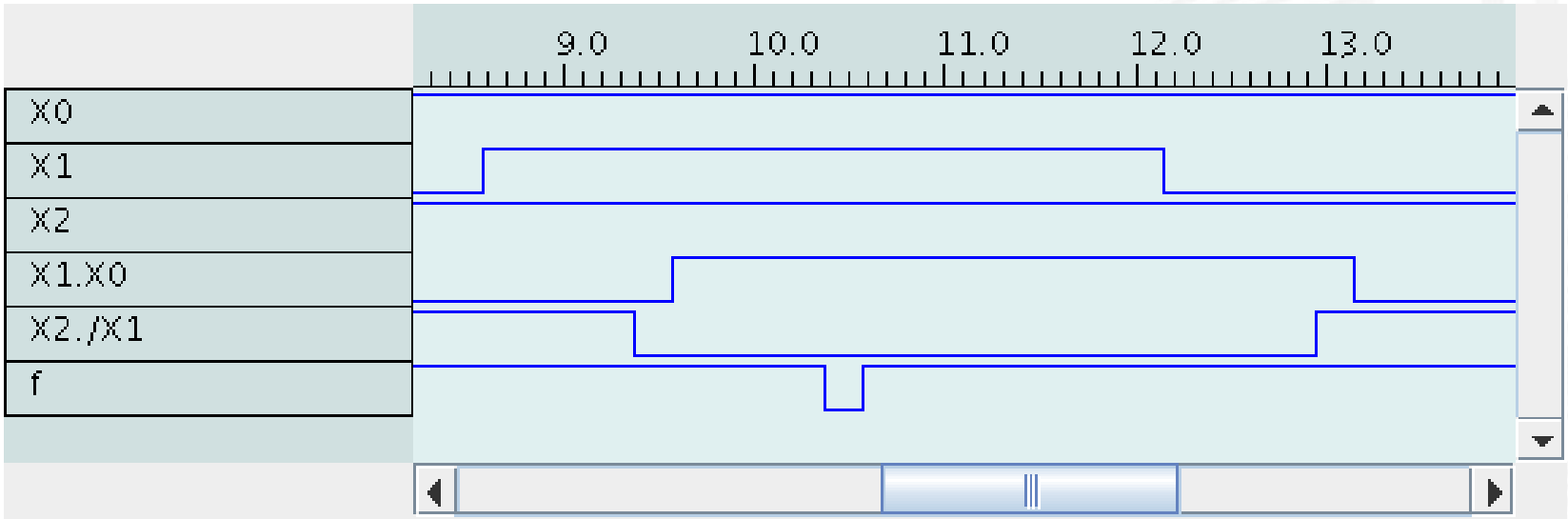
- Funktion $f = (x_2\,\overline{x_1}) \vee (x_1\,x_0)$
- realisiert in disjunktiver Form mit 2 Schleifen

Strukturhazard beim Übergang von $(x_2\,\overline{x_1}\,x_0)$ nach $(x_2\,x_1\,x_0)$

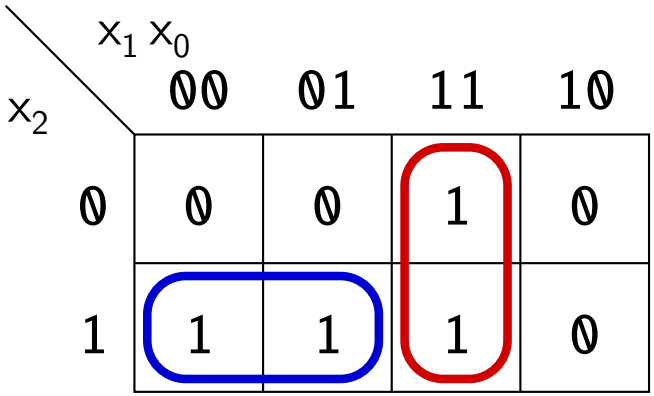
- Gatter $(x_2\,\overline{x_1})$ schaltet ab, Gatter $(x_1\,x_0)$ schaltet ein
- Ausgang evtl. kurz 0, abhängig von Verzögerungen

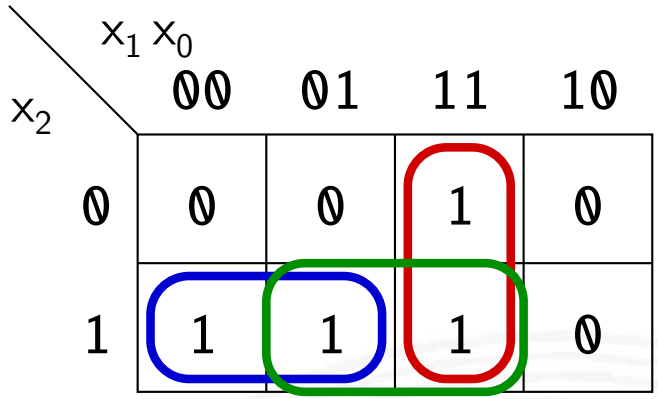
Strukturhazards im KV-Diagramm (cont.)





Strukturhazards beseitigen





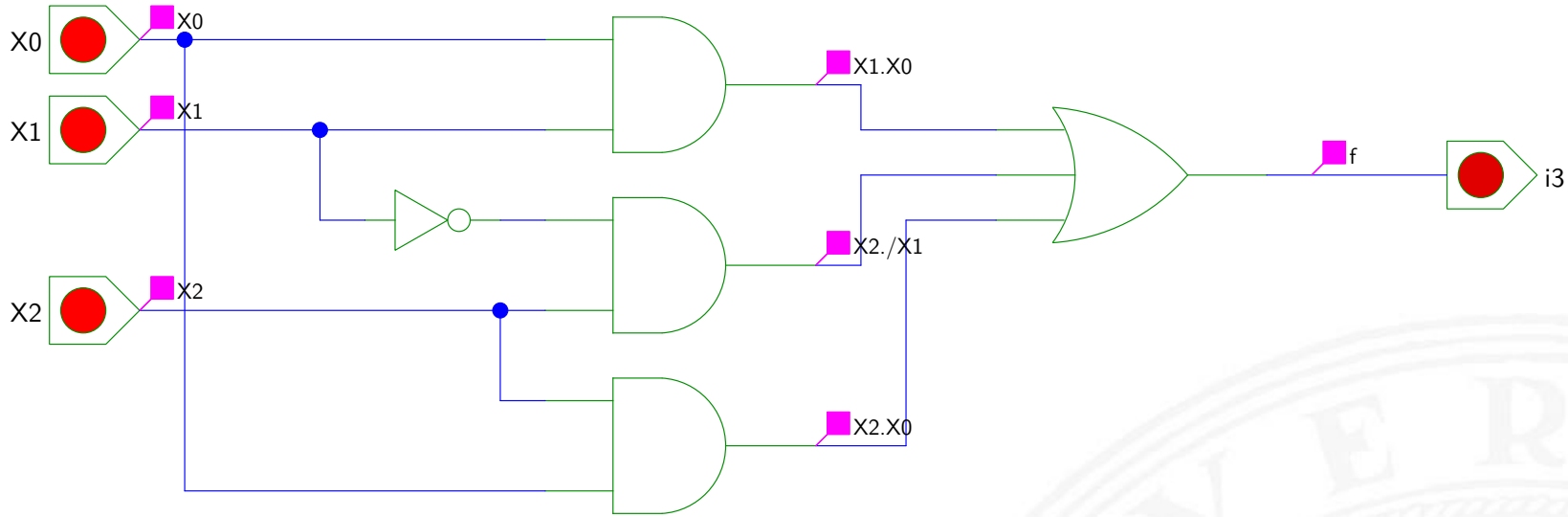
- Funktion $f = (x_2\,\overline{x_1}) \vee (x_1\,x_0)$
- realisiert in disjunktiver Form mit **3 Schleifen**

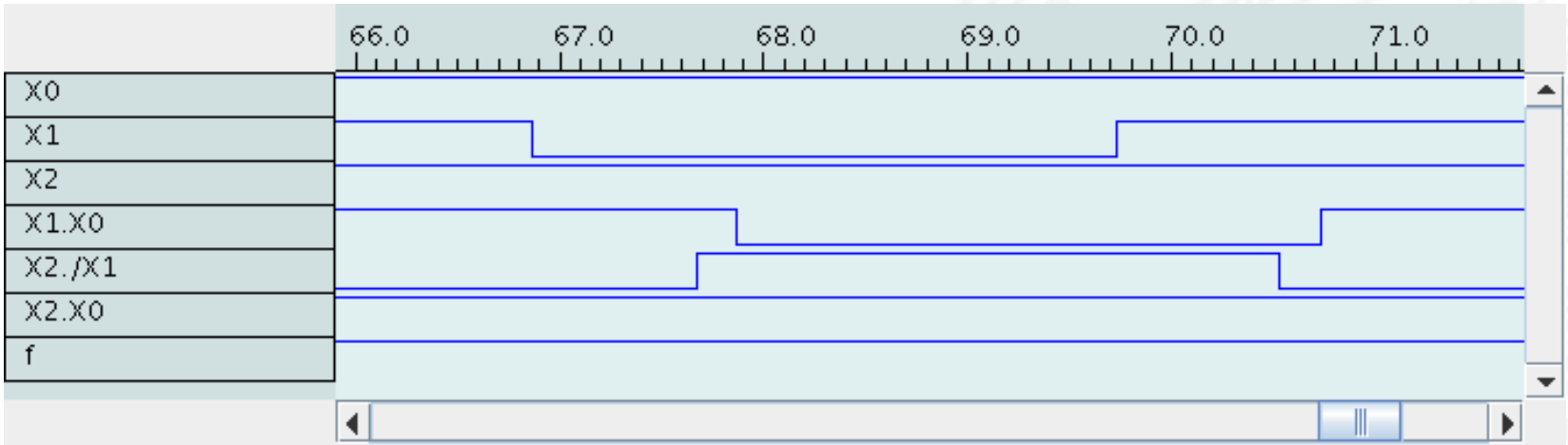
  $f = (x_2\,\overline{x_1}) \vee (x_1\,x_0) \vee (x_2\,x_0)$

\+ Strukturhazard durch zusätzliche Schleife beseitigt

\− aber höhere Hardwarekosten als bei minimierter Realisierung

Strukturhazards beseitigen (cont.)



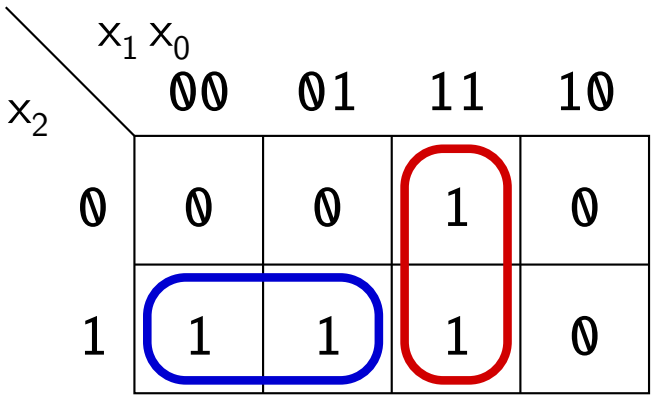


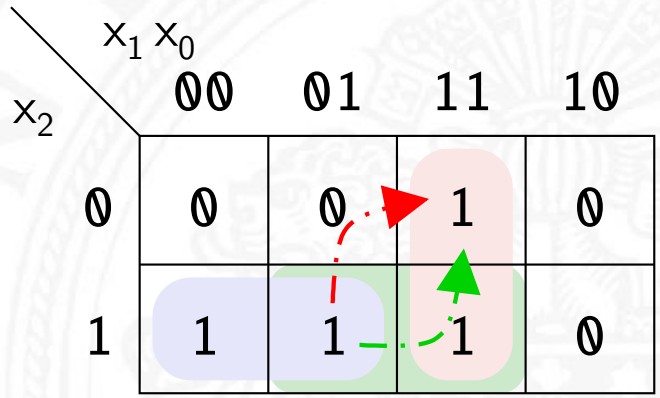
Hazards: Funktionshazard

- **Funktionshazard** kann bei gleichzeitigem Wechsel mehrerer Eingangswerte als Eigenschaft der Schaltfunktion entstehen
- Problem: Gleichzeitigkeit an Eingängen

$\Rightarrow$ Funktionshazard kann nicht durch strukturelle Maßnahmen verhindert werden

- Beispiel: Übergang von $(x_2\,\overline{x_1}\,x_0)$ nach $(\overline{x_2}\,x_1\,x_0)$





Literatur

[Knu08] D.E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 0, Introduction to Combinatorial Algorithms and Boolean Functions*.
Addison-Wesley Professional, 2008. ISBN 978–0–321–53496–5

[Knu09] D.E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams*.
Addison-Wesley Professional, 2009. ISBN 978–0–321–58050–4

[SS04] W. Schiffmann, R. Schmitz: *Technische Informatik 1 – Grundlagen der digitalen Elektronik*.
5. Auflage, Springer-Verlag, 2004. ISBN 978–3–540–40418–7

[Weg87] I. Wegener: *The Complexity of Boolean Functions*.
John Wiley & Sons, 1987. ISBN 3–519–02107–2.
`ls2-www.cs.uni-dortmund.de/monographs/bluebook`

Literatur (cont.)

[BM08] B. Becker, P. Molitor: *Technische Informatik – eine einführende Darstellung*. 2. Auflage, Oldenbourg, 2008. ISBN 978–3–486–58650–3

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

[Omo94] A.R. Omondi: *Computer Arithmetic Systems – Algorithms, Architecture and Implementations*. Prentice-Hall International, 1994. ISBN 0–13–334301–4

[Kor01] I. Koren: *Computer Arithmetic Algorithms*. 2nd edition, CRC Press, 2001. ISBN 978–1–568–81160–4. `www.ecs.umass.edu/ece/koren/arith`

[Spa76] O. Spaniol: *Arithmetik in Rechenanlagen*. B. G. Teubner, 1976. ISBN 3–519–02332–6

Interaktives Lehrmaterial

[Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript*. Uni. Hamburg, FB Informatik, 2005. tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1

[HenHA] N. Hendrich: *HADES — HAmburg DEsign System*. Universität Hamburg, FB Informatik, Lehrmaterial. tams.informatik.uni-hamburg.de/applets/hades/webdemos

[HenKV] N. Hendrich: *KV-Diagram Simulation*. Universität Hamburg, FB Informatik, Lehrmaterial. tams.informatik.uni-hamburg.de/applets/kvd

[Kor16] Laszlo Korte: *TAMS Tools for eLearning*. Universität Hamburg, FB Informatik, 2016, BSc Thesis. tams.informatik.uni-hamburg.de/research/software/tams-tools

[Laz] J. Lazarro: *Chipmunk design tools (AnaLog, DigLog)*. UC Berkeley, Berkeley, CA. john-lazzaro.github.io/chipmunk

Gliederung

Gliederung (cont.)

Gliederung (cont.)

Schaltwerke

- **Schaltwerk**: Schaltung mit Rückkopplungen und Verzögerungen

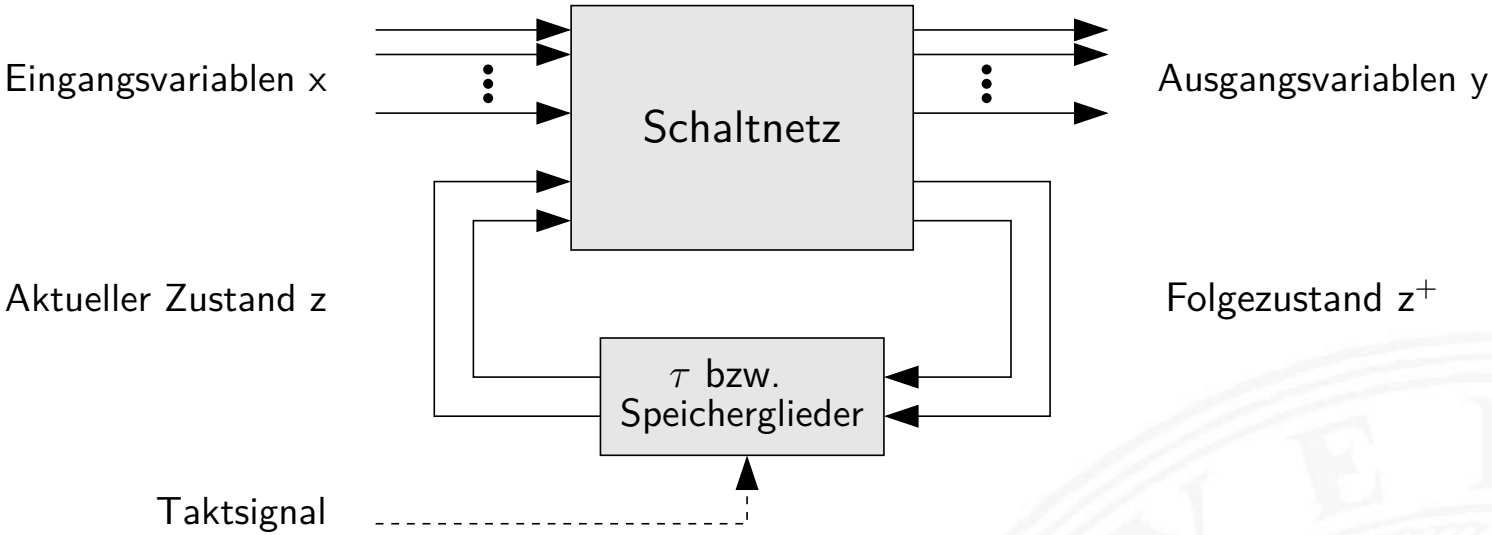
- fundamental andere Eigenschaften als Schaltnetze
- Ausgangswerte nicht nur von Eingangswerten abhängig sondern auch von der Vorgeschichte

$\Rightarrow$ interner Zustand repräsentiert „Vorgeschichte“

- ggf. stabile Zustände $\Rightarrow$ Speicherung von Information
- bei unvorsichtigem Entwurf: chaotisches Verhalten

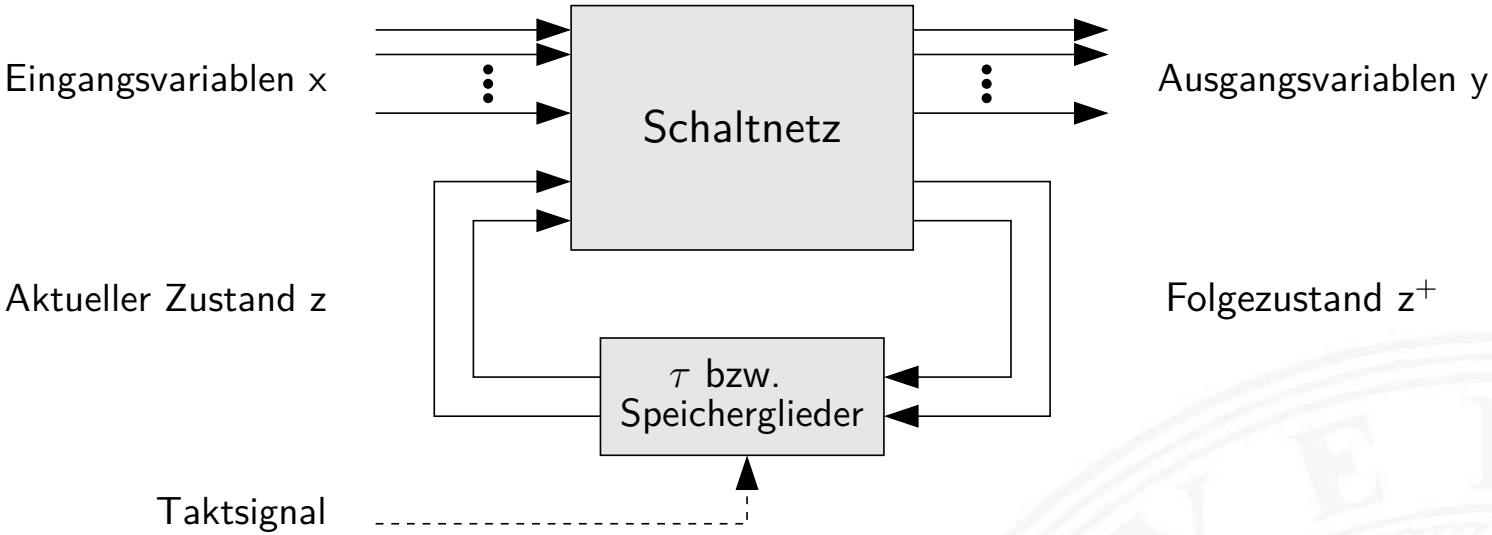
- Definition mit Rückkopplungen
  - Widerspruch: $x = \overline{x}$
  - Mehrdeutigkeit: $x = \overline{(\overline{x})}$
  - Beispiel mit zwei Variablen: $x = \overline{(a \wedge y)} \quad y = \overline{(b \wedge x)}$

Schaltwerke: Blockschaltbild



- Eingangsvariablen x und Ausgangsvariablen y
- Aktueller Zustand z
- Folgezustand z^+
- Rückkopplung läuft über Verzögerungen τ / Speicherglieder

Schaltwerke: Blockschaltbild (cont.)



zwei prinzipielle Varianten für die Zeitglieder

1. nur (Gatter-) Verzögerungen: **asynchrone** oder **nicht getaktete Schaltwerke**
2. getaktete Zeitglieder: **synchrone** oder **getaktete Schaltwerke**

Synchrone und Asynchrone Schaltwerke

- **synchrone Schaltwerke**: die Zeitpunkte, an denen das Schaltwerk von einem stabilen Zustand in einen stabilen Folgezustand übergeht, werden explizit durch ein Taktsignal (*clock*) vorgegeben

- **asynchrone Schaltwerke**: hier fehlt ein Taktgeber, Änderungen der Eingangssignale wirken sich unmittelbar aus (entsprechend der Gatterverzögerungen τ)
- potenziell höhere Arbeitsgeschwindigkeit
- aber sehr aufwändiger Entwurf
- fehleranfälliger (z.B. leicht veränderte Gatterverzögerungen durch Bauteil-Toleranzen, Spannungsschwankungen usw.)

FSM – **F**inite **S**tate **M**achine

- Deterministischer Endlicher Automat mit Ausgabe
- 2 äquivalente Modelle
  - Mealy: Ausgabe hängt *von Zustand und Eingabe* ab
  - Moore: –"– *nur vom Zustand* ab
- 6-Tupel $\langle Z, \Sigma, \Delta, \delta, \lambda, z_0 \rangle$
  - Z Menge von Zuständen
  - Σ Eingabealphabet
  - Δ Ausgabealphabet
  - δ Übergangsfunktion $\delta : Z \times \Sigma \rightarrow Z$
  - λ Ausgabefunktion $\lambda : Z \times \Sigma \rightarrow \Delta$ Mealy-Modell
    $\lambda : Z \rightarrow \Delta$ Moore- –"–
  - z_0 Startzustand

Mealy-Modell und Moore-Modell

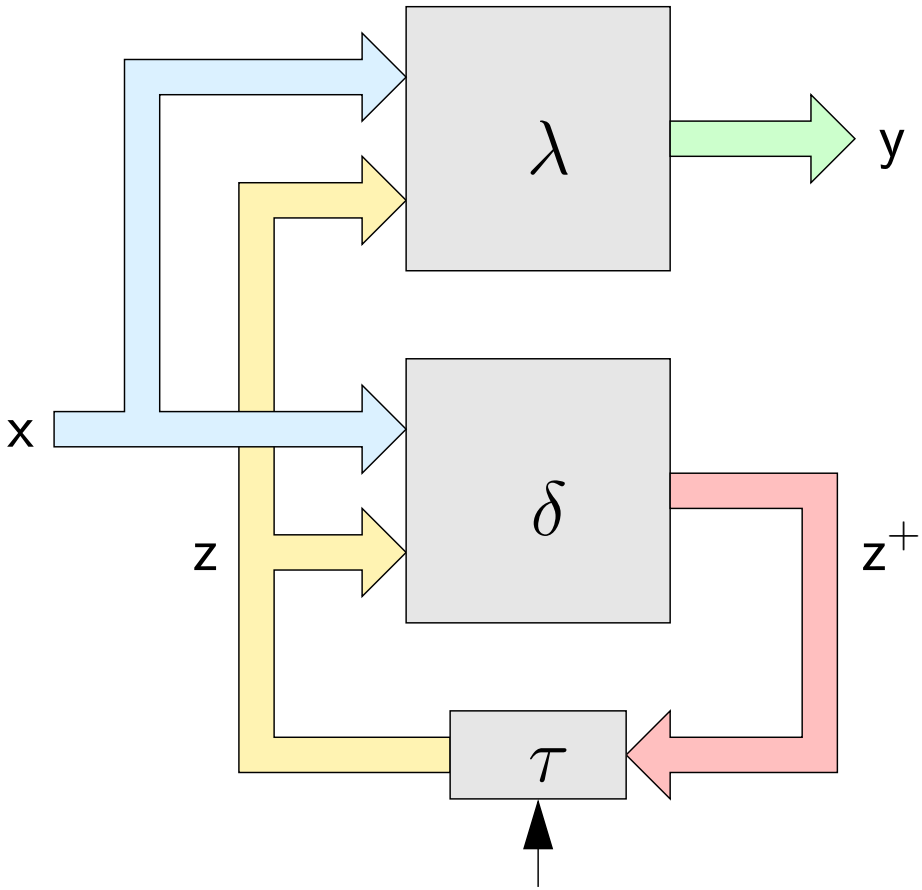
- **Mealy-Modell**: die Ausgabe hängt vom Zustand z und vom momentanen Input x ab
- **Moore-Modell**: die Ausgabe des Schaltwerks hängt nur vom aktuellen Zustand z ab

Ausgabefunktion:	$y = \lambda(z, x)$	Mealy
	$y = \lambda(z)$	Moore
Überführungsfunktion:	$z^+ = \delta(z, x)$	Moore und Mealy

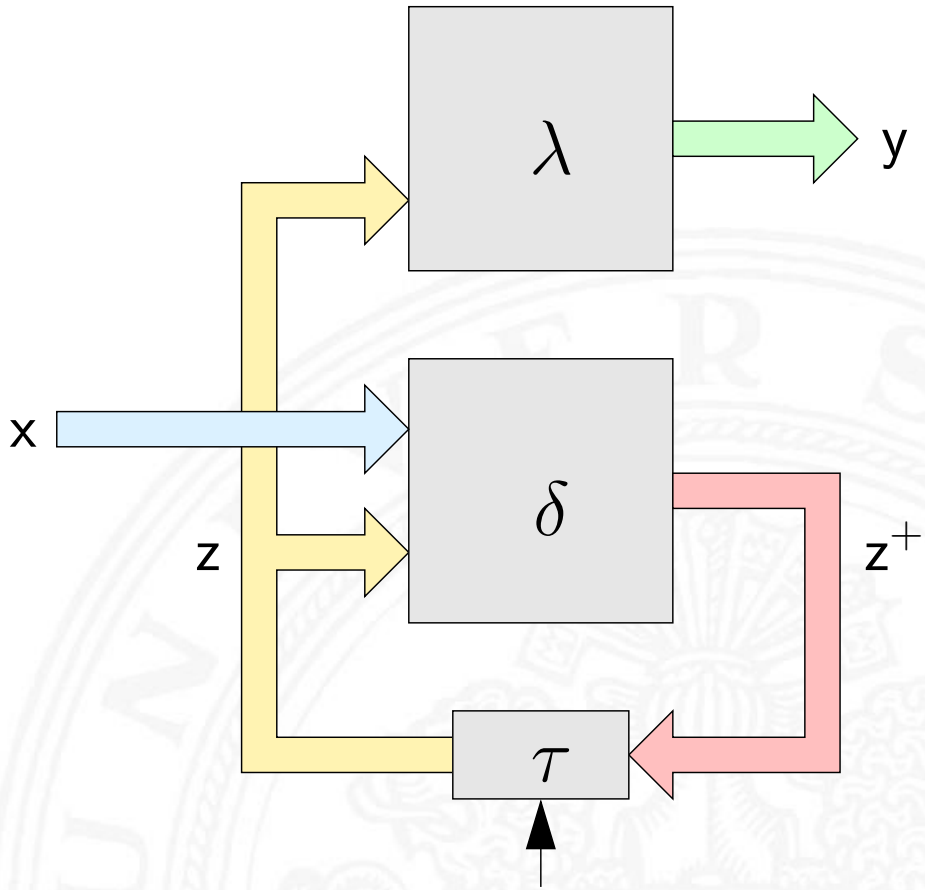
- **Speicherglieder** oder Verzögerung τ im Rückkopplungspfad

Mealy-Modell und Moore-Modell (cont.)

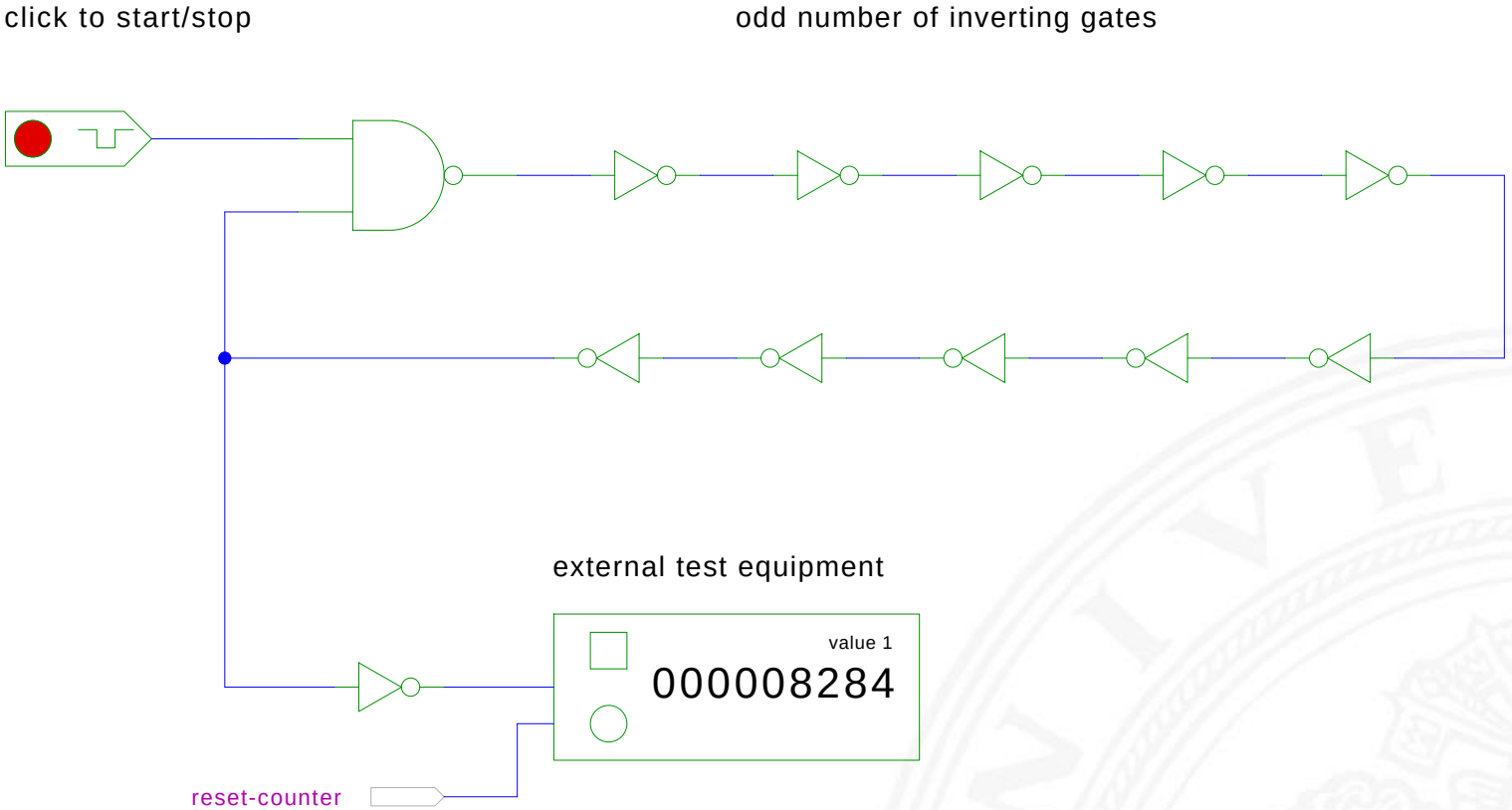
- Mealy-Automat



Moore-Automat



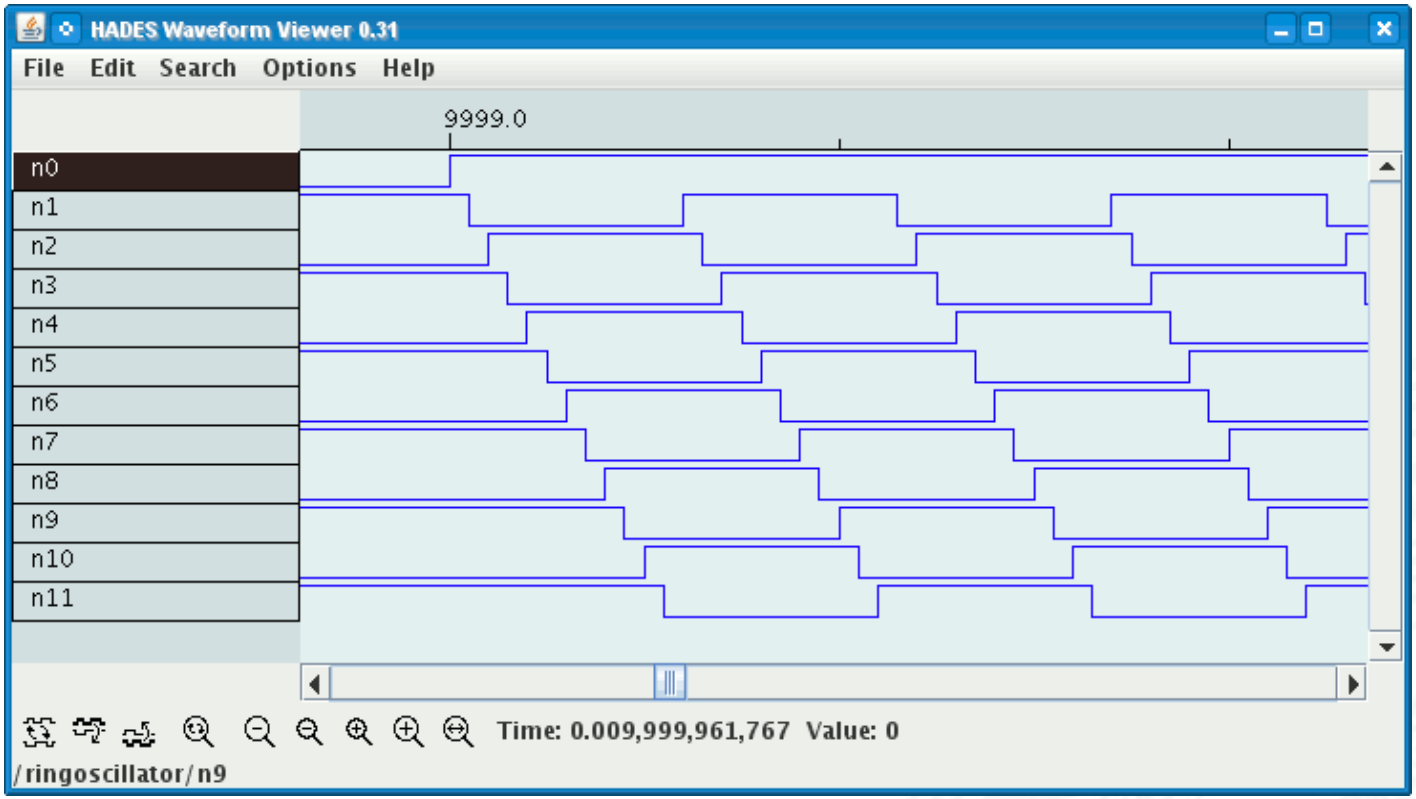
Asynchrone Schaltungen: Beispiel Ringoszillator



[HenHA] Hades Demo: `12-gatedelay/20-ringoscillator/ringoscillator`

- stabiler Zustand, solange der Eingang auf 0 liegt
- instabil sobald der Eingang auf 1 wechselt (Oszillation)

Asynchrone Schaltungen: Beispiel Ringoszillator (cont.)



- Rückkopplung: ungerade Anzahl n invertierender Gatter ($n \geq 3$)
- Start/Stop über steuerndes NAND-Gatter
- Oszillation mit maximaler Schaltfrequenz
  z.B.: als Testschaltung für neue (Halbleiter-) Technologien

Asynchrone Schaltungen: Probleme

- das Schaltwerk kann stabile und nicht-stabile Zustände enthalten
- die Verzögerungen der Bauelemente sind nicht genau bekannt und können sich im Betrieb ändern
- Variation durch Umweltparameter
  z.B. Temperatur, Versorgungsspannung, Alterung

$\Rightarrow$ sehr schwierig, die korrekte Funktion zu garantieren
z.B. mehrstufige Handshake-Protokolle

- in der Praxis überwiegen synchrone Schaltwerke
- Realisierung mit **Flipflops** als Zeitgliedern

Synchrone Schaltungen

- alle Rückkopplungen der Schaltung laufen über spezielle Zeitglieder: „Flipflops“
- diese definieren / speichern einen stabilen Zustand, unabhängig von den Eingabewerten und Vorgängen im δ-Schaltnetz

- Hinzufügen eines zusätzlichen Eingangssignals: „Takt“
- die Zeitglieder werden über das Taktsignal gesteuert verschiedene Möglichkeiten: Pegel- und Flankensteuerung, Mehrphasentakte (s.u.)

$\Rightarrow$ synchrone Schaltwerke sind wesentlich einfacher zu entwerfen und zu analysieren als asynchrone Schaltungen

Zeitglieder / Flipflops

- **Zeitglieder**: Bezeichnung für die Bauelemente, die den Zustand des Schaltwerks speichern können
- **bistabile Bauelemente** (Kippglieder) oder **Flipflops**

- zwei stabile Zustände $\Rightarrow$ speichert 1 Bit

  1 – Setzzustand

  0 – Rücksetzzustand

- Übergang zwischen Zuständen durch geeignete Ansteuerung

Flipflops

- Name für die **elementaren** Schaltwerke
- mit genau zwei Zuständen Z_0 und Z_1
- Zustandsdiagramm hat zwei Knoten und vier Übergänge (s.u.)

- Ausgang als Q bezeichnet und dem Zustand gleichgesetzt
- meistens auch invertierter Ausgang $\overline{Q}$ verfügbar

- Flipflops sind selbst nicht getaktet
- sondern „sauber entworfene“ asynchrone Schaltwerke
- Anwendung als Verzögerungs-/Speicherelemente in getakteten Schaltwerken

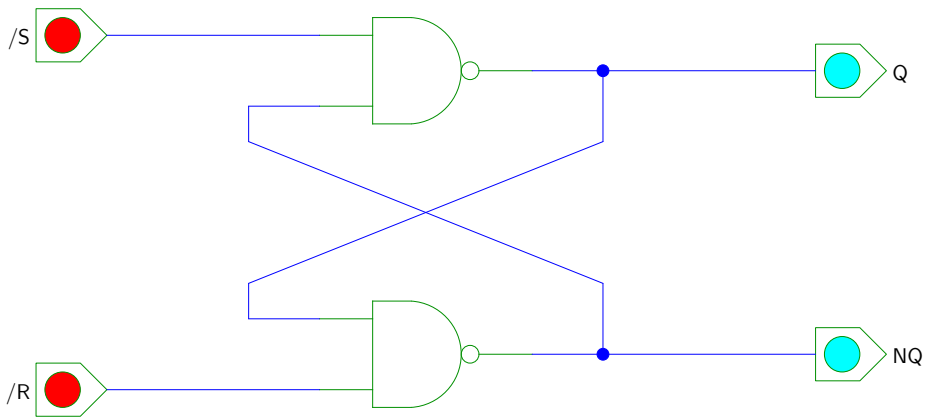
Flipflops: Typen

- Basis-Flipflop „Reset-Set-Flipflop“
- getaktetes RS-Flipflop

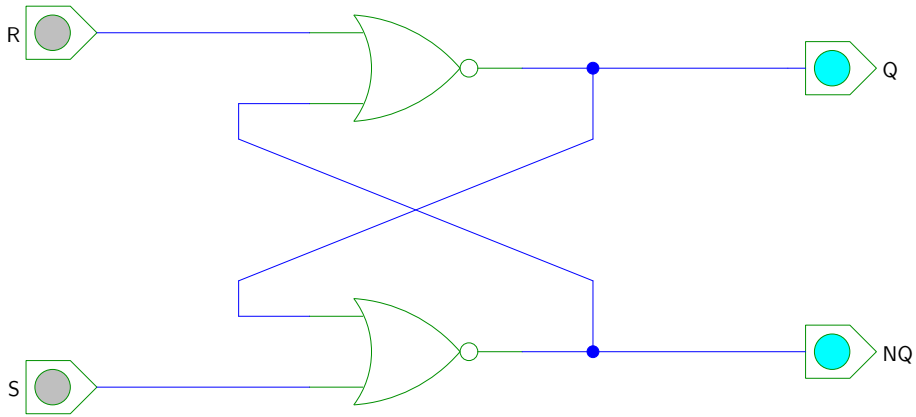
- pegelgesteuertes D-Flipflop „D-Latch“
- flankengesteuertes D-Flipflop „D-Flipflop“

- JK-Flipflop
- weitere. . .

RS-Flipflop: NAND- und NOR-Realisierung



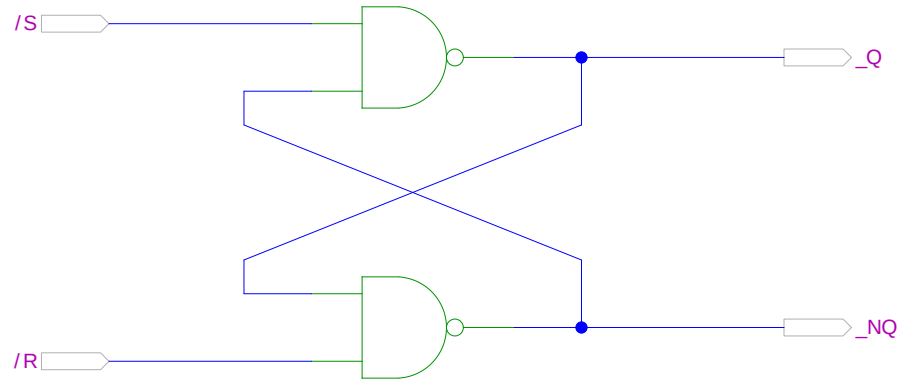
/S	/R	Q	NQ	NAND
0	0	1	1	forbidden
0	1	1	0	
1	0	0	1	
1	1	Q*	NQ*	store

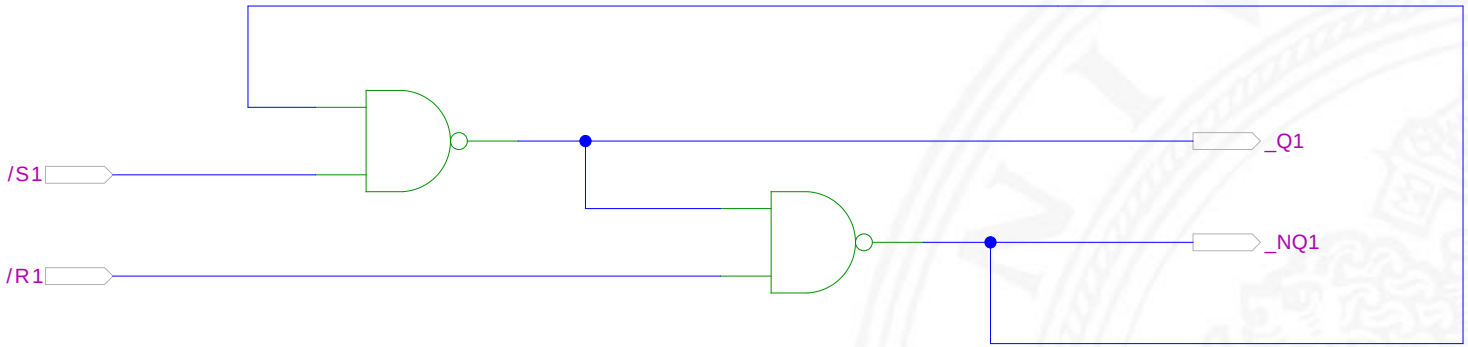


S	R	Q	NQ	NOR
0	0	Q*	NQ*	store
0	1	0	1	
1	0	1	0	
1	1	0	0	forbidden

[HenHA] Hades Demo: `16-flipflops/10-srff/srff`

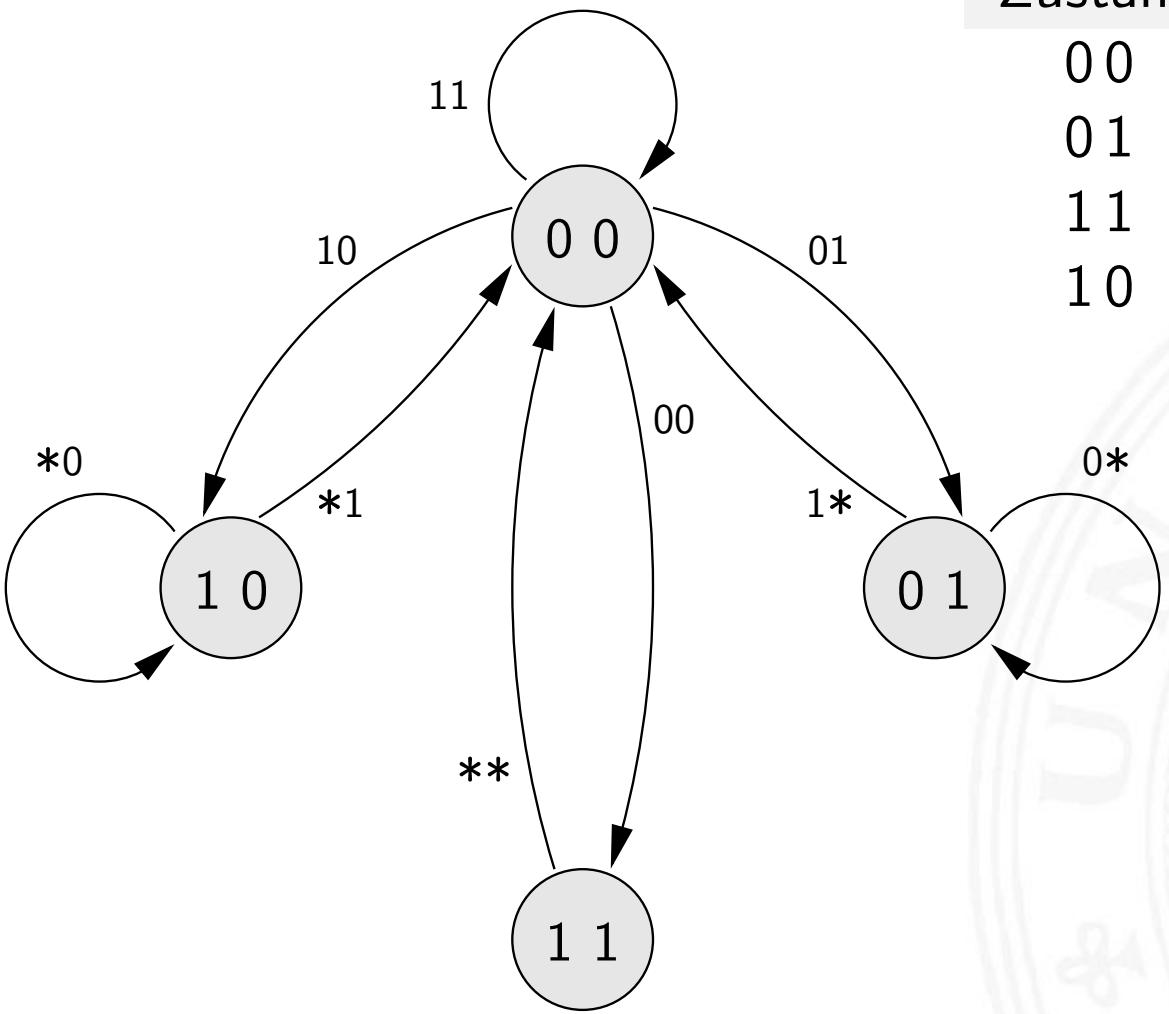
RS-Flipflop: Varianten des Schaltbilds





[HenHA] Hades Demo: `16-flipflops/10-srff/srff2`

NOR RS-Flipflop: Zustandsdiagramm und Flusstafel



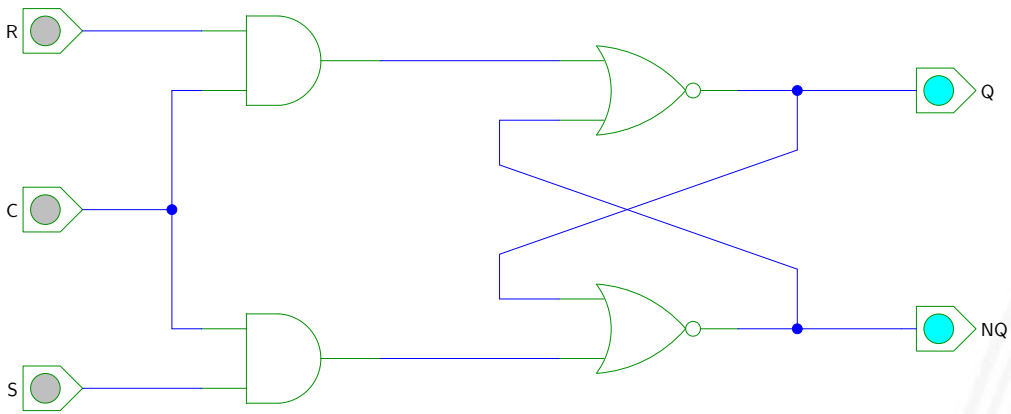
	Eingabe $[S\,R]$			
	0 0	0 1	1 1	1 0
Zustand	Folgezustand			$[Q\,\overline{Q}]$
0 0	1 1	0 1	0 0	1 0
0 1	0 1	0 1	0 0	0 0
1 1	0 0	0 0	0 0	0 0
1 0	1 0	0 0	0 0	1 0

stabiler Zustand

RS-Flipflop mit Takt

- RS-Basisflipflop mit zusätzlichem Takteingang C
- Änderungen nur wirksam, während C aktiv ist

- Struktur



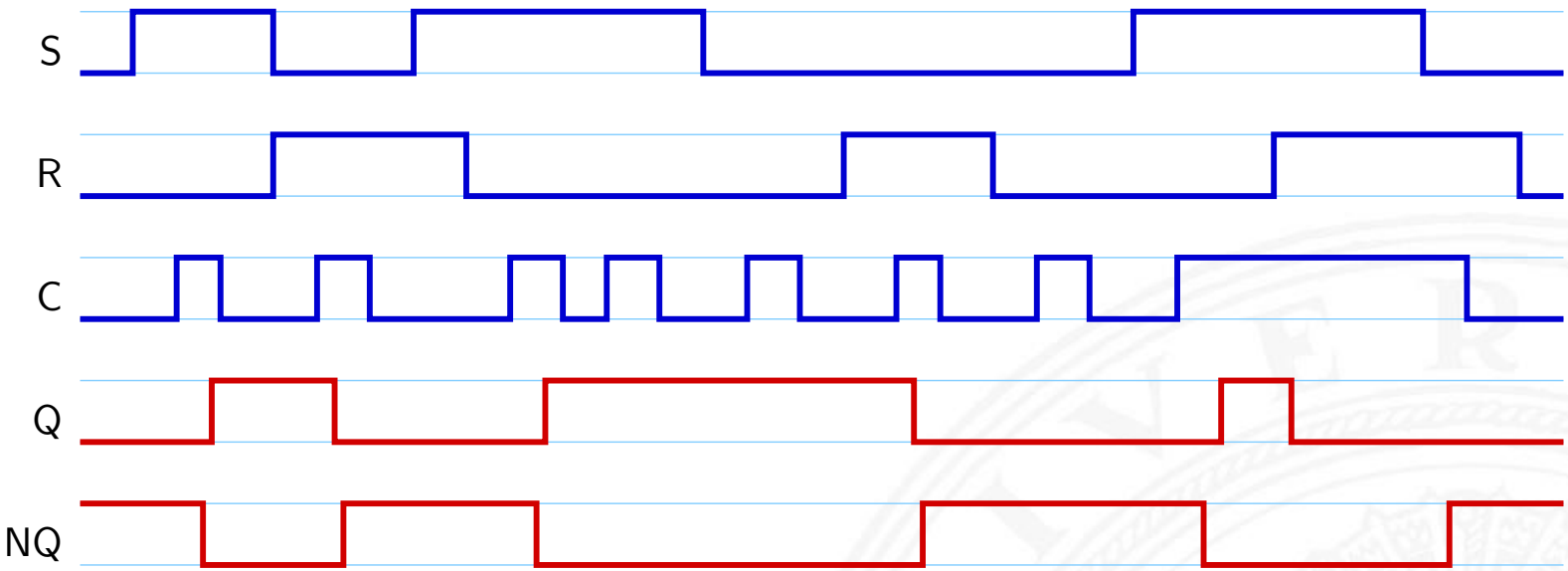
C	S	R	Q	NQ	NOR
0	X	X	Q*	NQ*	store
1	0	0	Q*	NQ*	store
1	0	1	0	1	
1	1	0	1	0	
1	1	1	0	0	forbidden

[HenHA] Hades Demo: `16-flipflops/10-srff/clocked-srff`

- $Q = \overline{(NQ \vee (R \wedge C))}$
  $NQ = \overline{(Q \vee (S \wedge C))}$

RS-Flipflop mit Takt (cont.)

- Impulsdiagramm



- $Q = \overline{(NQ \vee (R \wedge C))}$
  $NQ = \overline{(Q \vee (S \wedge C))}$

Pegelgesteuertes D-Flipflop (D-Latch)

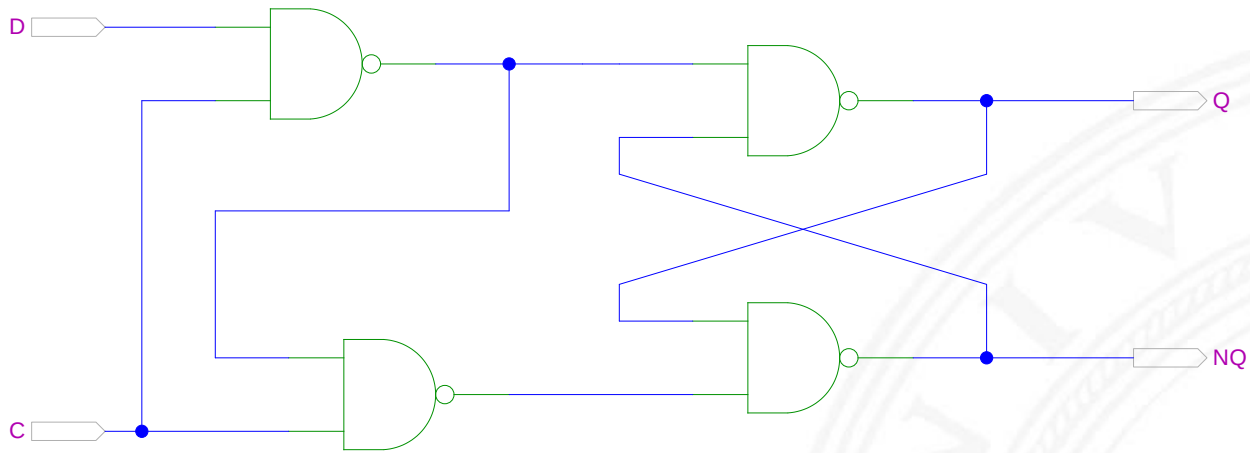
- Takteingang C
- Dateneingang D
- aktueller Zustand Q, Folgezustand Q^+

C	D	Q^+
0	0	Q
0	1	Q
1	0	0
1	1	1

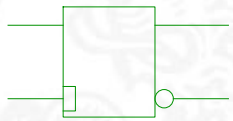
- Wert am Dateneingang wird durchgeleitet, wenn das Taktsignal
  1 ist $\Rightarrow$ *high*-aktiv
  0 ist $\Rightarrow$ *low*-aktiv

Pegelgesteuertes D-Flipflop (D-Latch) (cont.)

- Realisierung mit getaktetem RS-Flipflop und einem Inverter
  $S = D$, $R = \overline{D}$
- minimierte NAND-Struktur

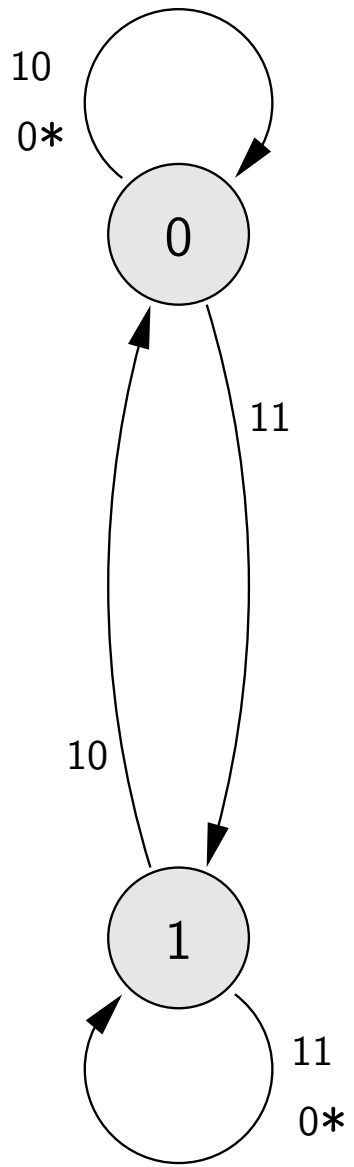


- Symbol



[HenHA] Hades Demo: `16-flipflops/20-dlatch/dlatch`

D-Latch: Zustandsdiagramm und Flusstafel



	Eingabe $[C\ D]$			
	0 0	0 1	1 1	1 0
Zustand $[Q]$	Folgezustand			$[Q^+]$
0	0	0	1	0
1	1	1	1	0

stabiler Zustand

Flankengesteuertes D-Flipflop

- Takteingang C
- Dateneingang D
- aktueller Zustand Q, Folgezustand Q^+

C	D	Q^+
0	$*$	Q
1	$*$	Q
$\uparrow$	0	0
$\uparrow$	1	1

- Wert am Dateneingang wird gespeichert, wenn das Taktsignal sich von 0 auf 1 ändert $\Rightarrow$ Vorderflankensteuerung
  –"– 1 auf 0 ändert $\Rightarrow$ Rückflankensteuerung
- Realisierung als Master-Slave Flipflop oder direkt

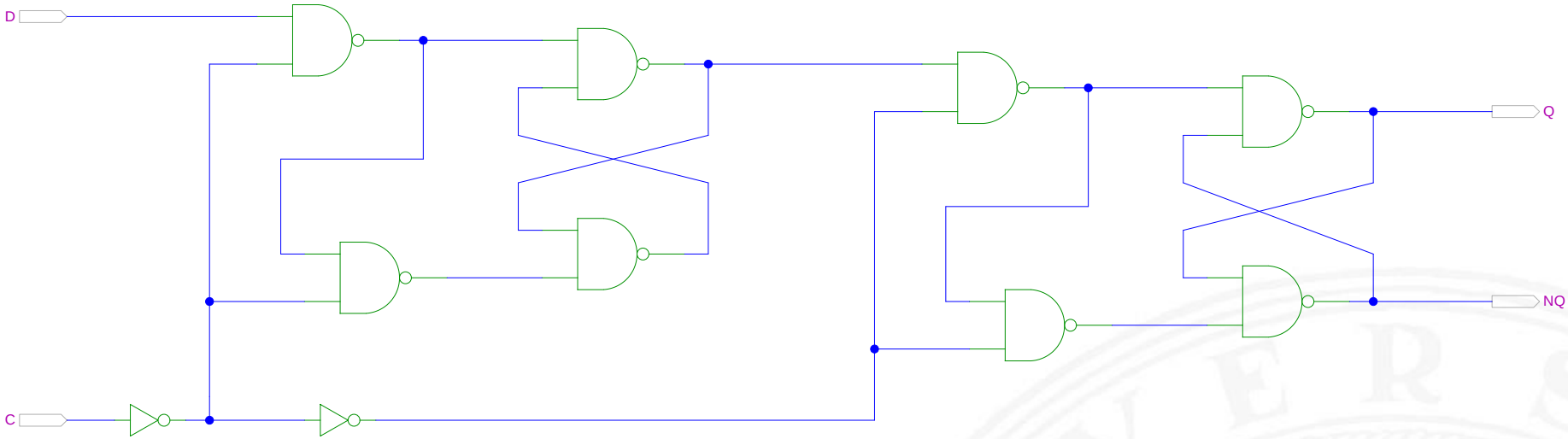
Master-Slave D-Flipflop

- zwei kaskadierte D-Latches
- hinteres Latch erhält invertierten Takt

- vorderes „Master“-Latch: low-aktiv (transparent bei $C = 0$)
  hinteres „Slave“-Latch: high-aktiv (transparent bei $C = 1$)
- vorderes Latch speichert bei Wechsel auf $C = 1$
- wenig später (Gatterverzögerung im Inverter der Taktleitung) übernimmt das hintere Slave-Latch diesen Wert
- anschließend Input für das Slave-Latch stabil
- Slave-Latch speichert, sobald Takt auf $C = 0$ wechselt

$\Rightarrow$ dies entspricht effektiv einer **Flankensteuerung**:
Wert an D nur relevant, kurz bevor Takt auf $C = 1$ wechselt

Master-Slave D-Flipflop (cont.)



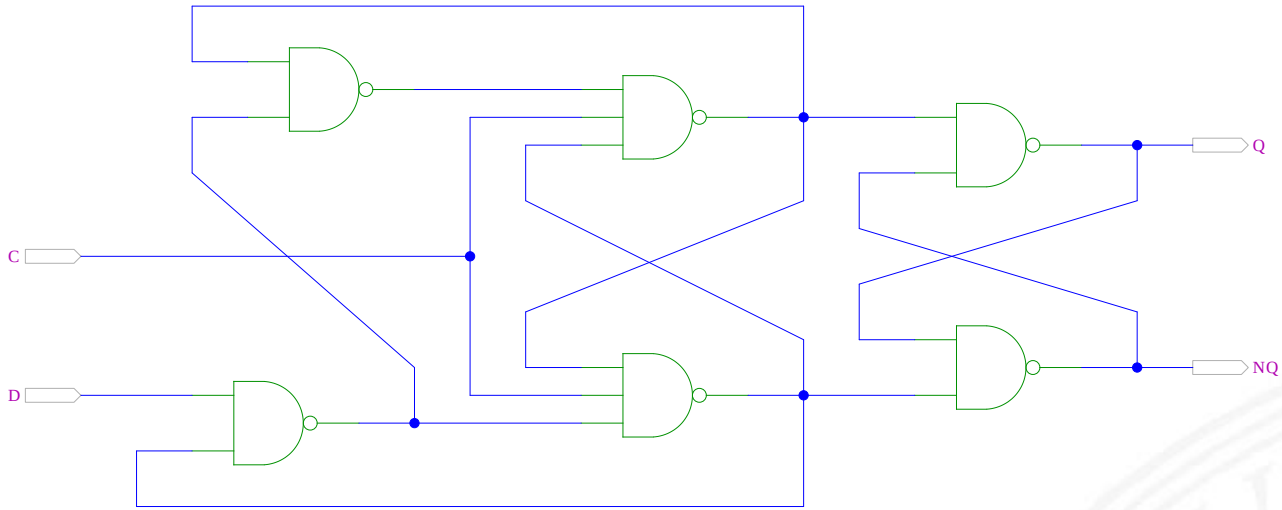
[HenHA] Hades Demo: `16-flipflops/20-dlatch/dff`

- zwei kaskadierte pegel-gesteuerte D-Latches

C=0 Master aktiv (transparent)
Slave hat (vorherigen) Wert gespeichert

C=1 Master speichert Wert
Slave transparent, leitet Wert von Master weiter

Vorderflanken-gesteuertes D-Flipflop



- Dateneingang *D* wird nur durch Takt-Vorderflanke ausgewertet
- Gatterlaufzeiten für Funktion essenziell
- Einhalten der Vorlauf- und Haltezeiten vor/nach der Taktflanke (s.u. *Zeitbedingungen*)

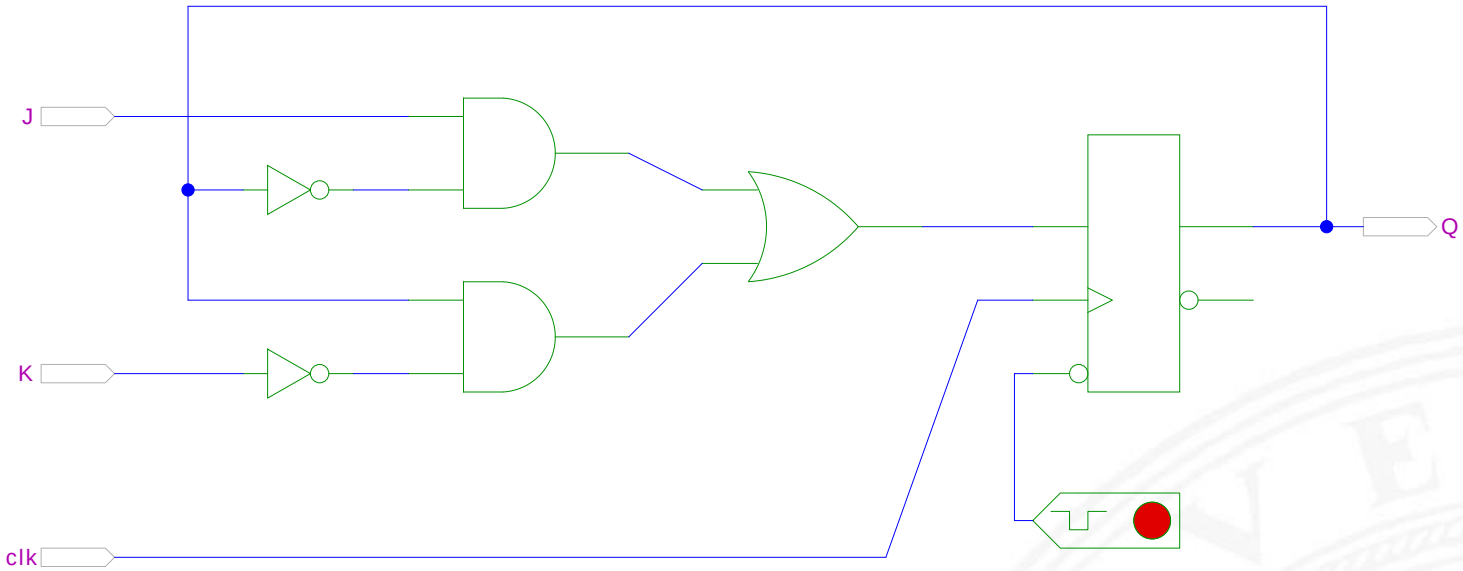
JK-Flipflop

- Takteingang C
- Steuereingänge J („jump“) und K („kill“)
- aktueller Zustand Q, Folgezustand Q^+

C	J	K	Q^+	Funktion
$*$	$*$	$*$	Q	Wert gespeichert
$\uparrow$	0	0	Q	Wert gespeichert
$\uparrow$	0	1	0	Rücksetzen
$\uparrow$	1	0	1	Setzen
$\uparrow$	1	1	$\overline{Q}$	Invertieren

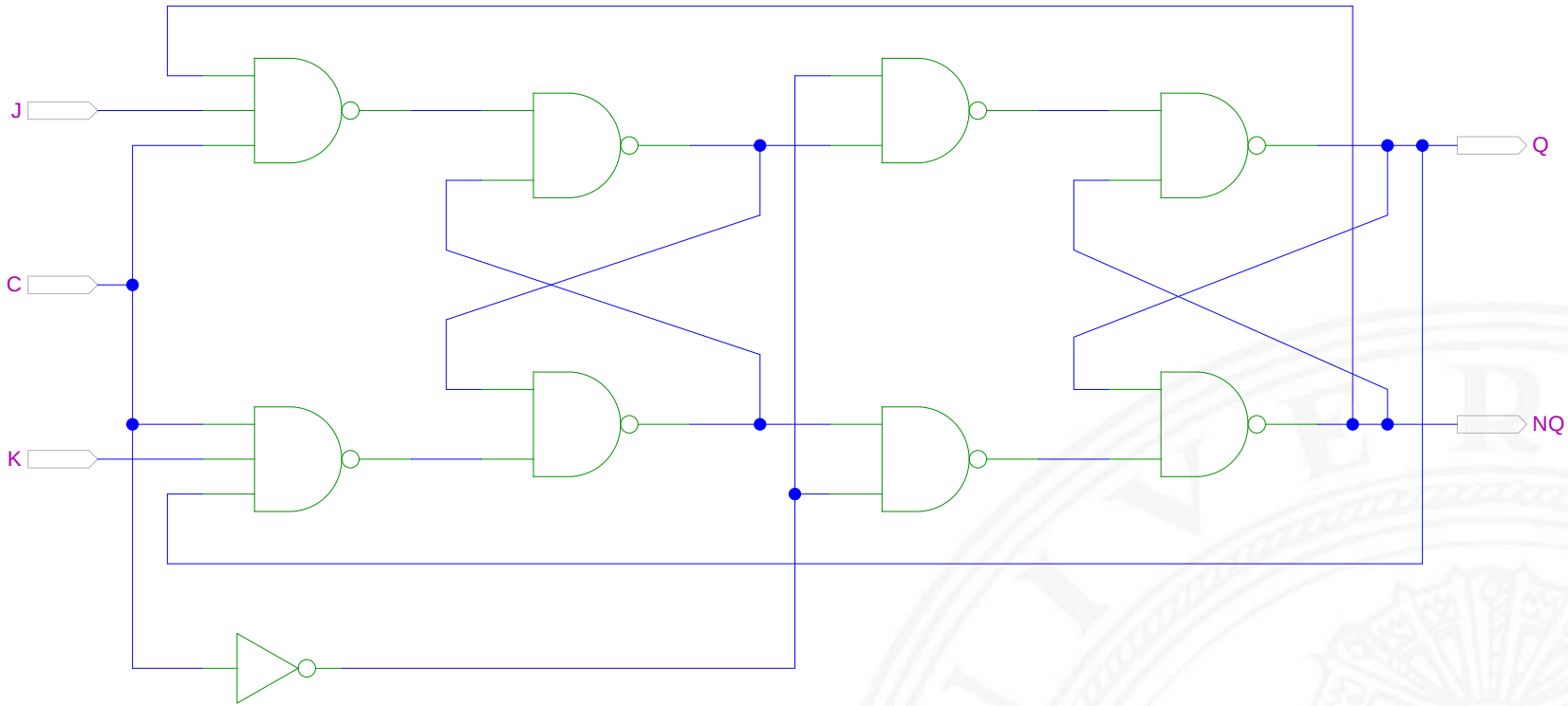
- universelles Flipflop, sehr flexibel einsetzbar
- in integrierten Schaltungen nur noch selten verwendet (höherer Hardware-Aufwand als Latch/D-Flipflop)

JK-Flipflop: Realisierung mit D-Flipflop



[HenHA] Hades Demo: `16-flipflops/40-jkff/jkff-prinzip`

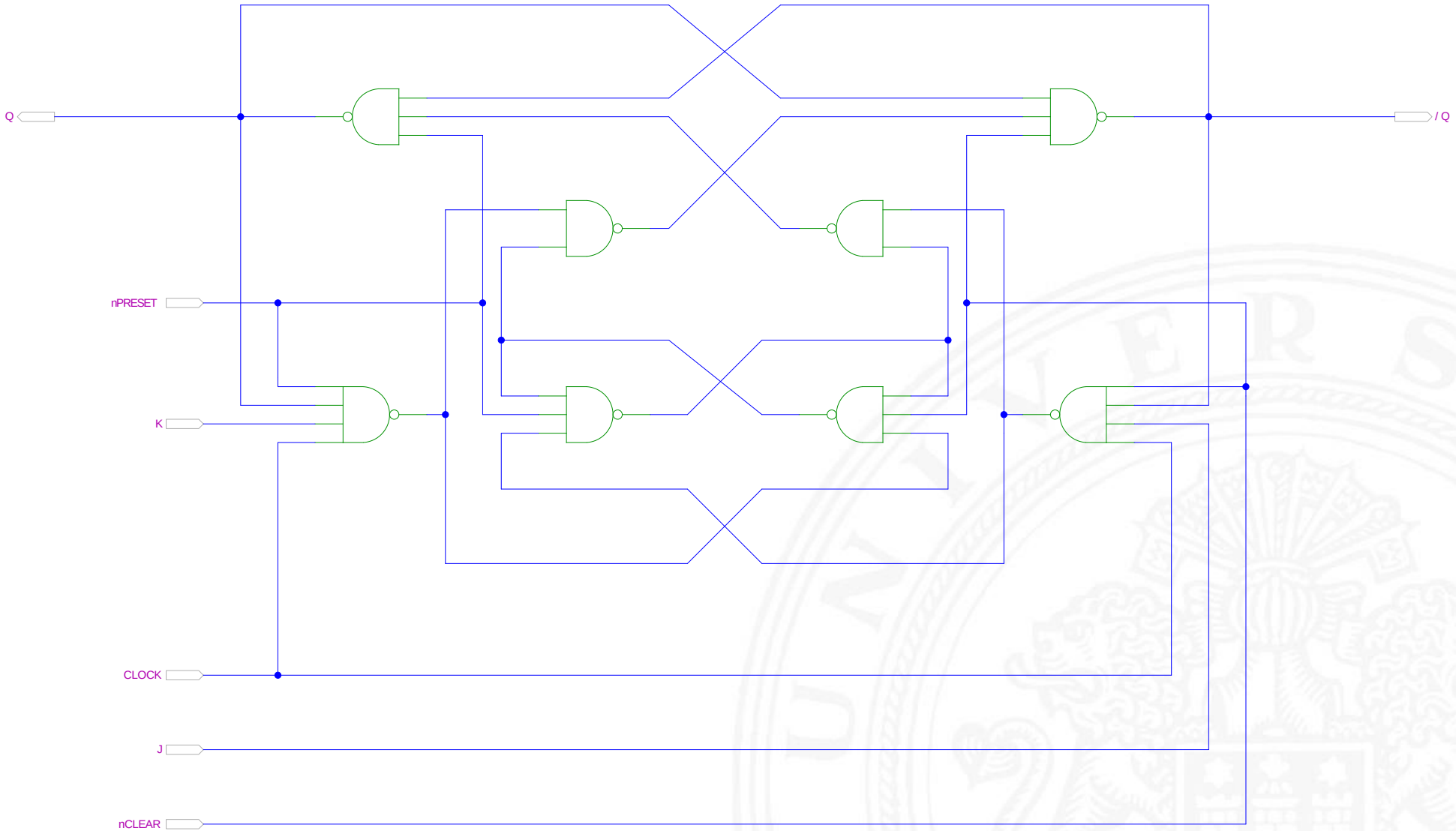
JK-Flipflop: Realisierung als Master-Slave Schaltung



[HenHA] Hades Demo: `16-flipflops/40-jkff/jkff`

- Achtung: Schaltung wegen Rückkopplungen schwer zu initialisieren

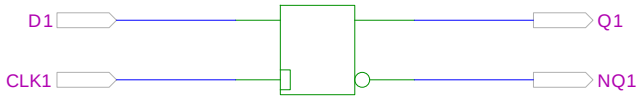
JK-Flipflop: tatsächliche Schaltung im IC 7476

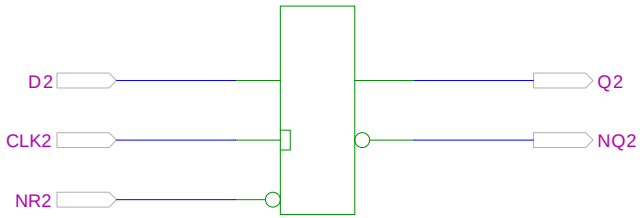


[HenHA] Hades Demo: `16-flipflops/40-jkff/SN7476-single`

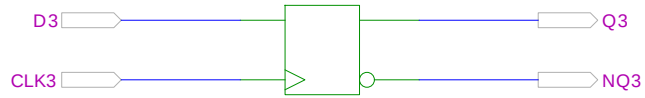
Flipflop-Typen: Komponenten/Symbole in Hades

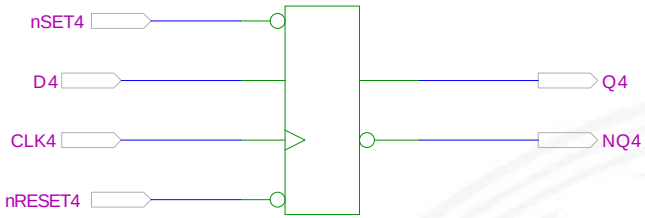
D-type latches



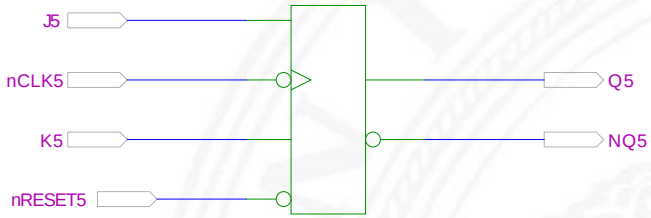


D-type flipflops

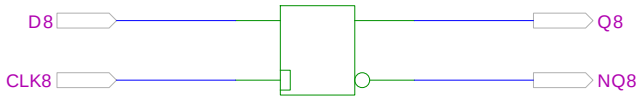




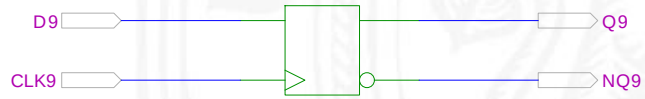
JK flipflop



metastable D-Latch (don't use!)

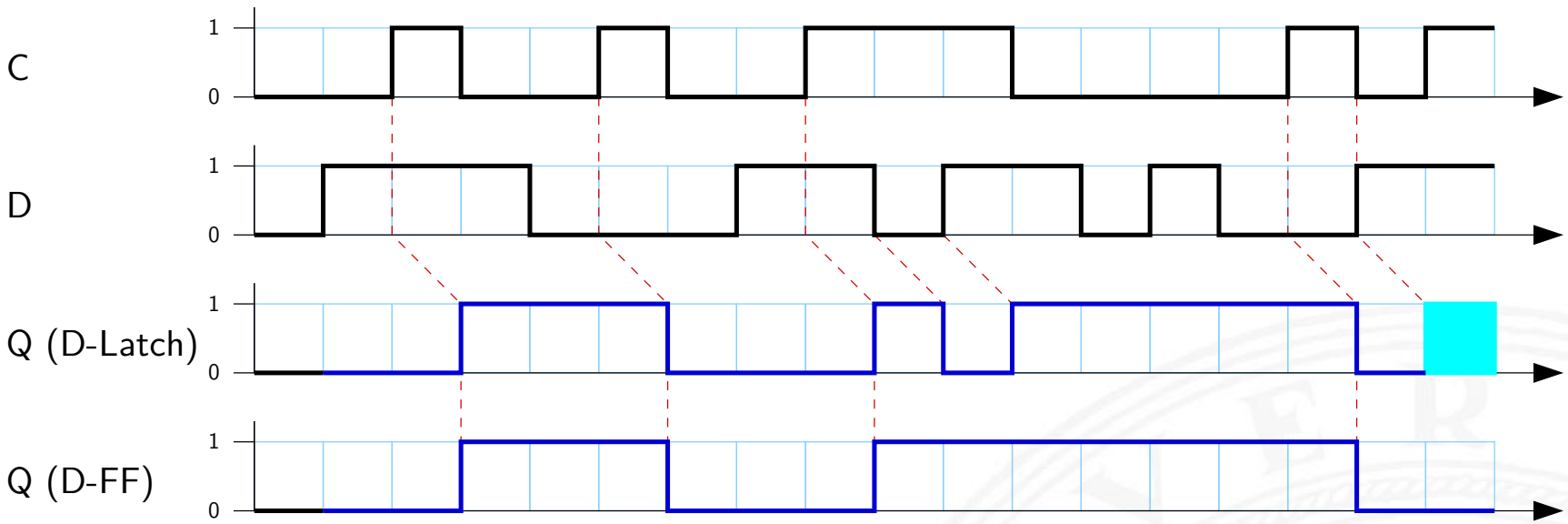


metastable D-flipflop (don't use!)



[HenHA] Hades Demo: `16-flipflops/50-ffdemo/flipflopdemo`

Flipflop-Typen: Impulsdiagramme



- pegel- und vorderflankengesteuertes Flipflop
- beide Flipflops hier mit jeweils einer Zeiteinheit Verzögerung
- am Ende undefinierte Werte im Latch
  - gleichzeitiger Wechsel von C und D
  - Verletzung der Zeitbedingungen
  - in der Realität wird natürlich ein Wert 0 oder 1 gespeichert, abhängig von externen Parametern (Temperatur, Versorgungsspannung etc.) kann er sich aber ändern

Flipflops: Zeitbedingungen

- Flipflops werden entwickelt, um Schaltwerke einfacher entwerfen und betreiben zu können
- Umschalten des Zustandes durch das Taktsignal gesteuert

- aber: jedes Flipflop selbst ist ein asynchrones Schaltwerk mit kompliziertem internem Zeitverhalten
- Funktion kann nur garantiert werden, wenn (typ-spezifische) Zeitbedingungen eingehalten werden

$\Rightarrow$ Daten- und Takteingänge dürfen sich nicht gleichzeitig ändern

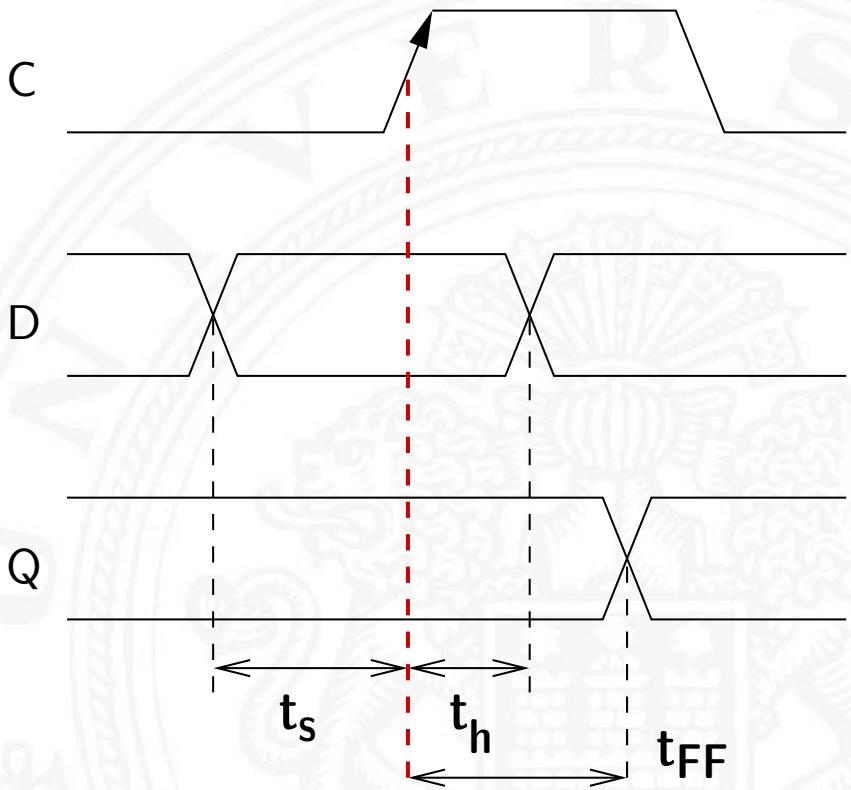
Welcher Wert wird gespeichert?

$\Rightarrow$ „Vorlauf- und Haltezeiten“ (*setup- / hold-time*)

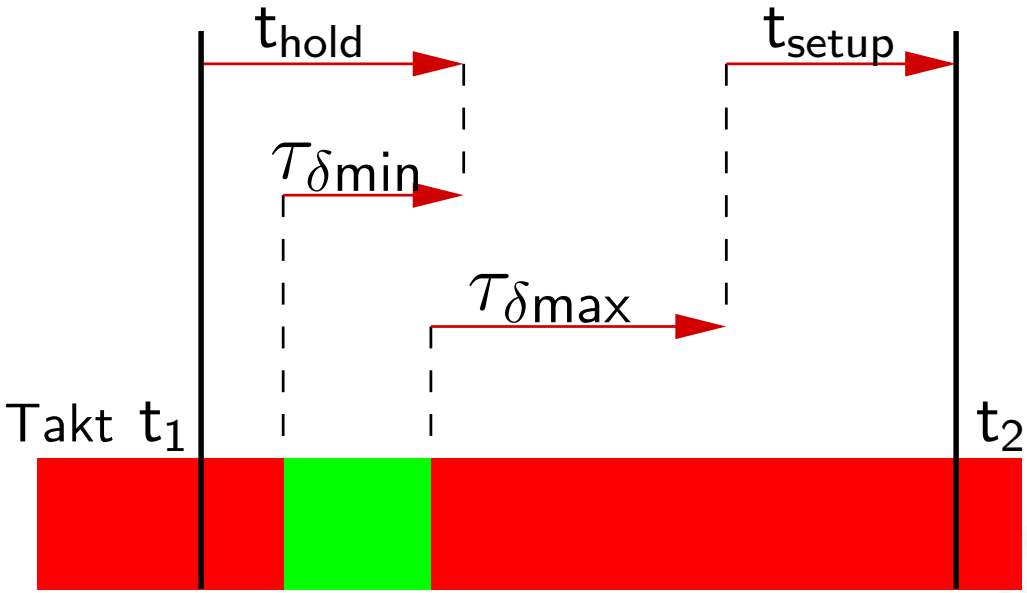
Flipflops: Vorlauf- und Haltezeit

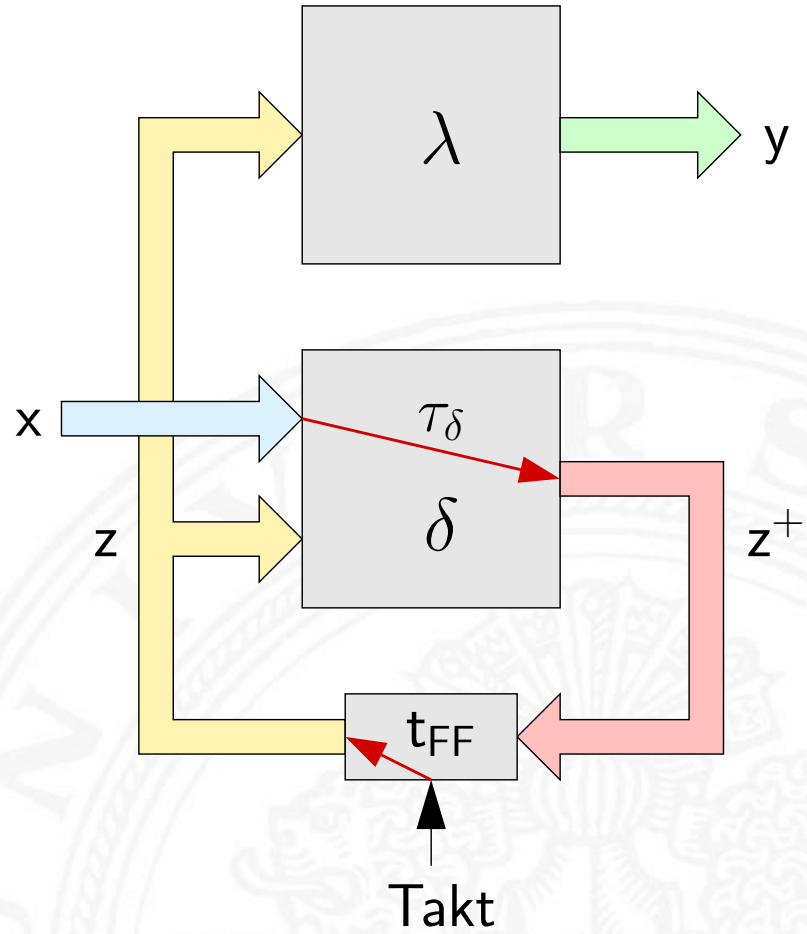
- t_s Vorlaufzeit (engl. *setup-time*): Zeitintervall, innerhalb dessen das Datensignal *vor* dem nächsten Takt stabil anliegen muss
- t_h Haltezeit (engl. *hold-time*): Zeitintervall, innerhalb dessen das Datensignal *nach* einem Takt noch stabil anliegen muss
- t_{FF} Ausgangsverzögerung

$\Rightarrow$ Verletzung der Zeitbedingungen „falscher“ Wert an Q



Zeitbedingungen: Eingangsvektor

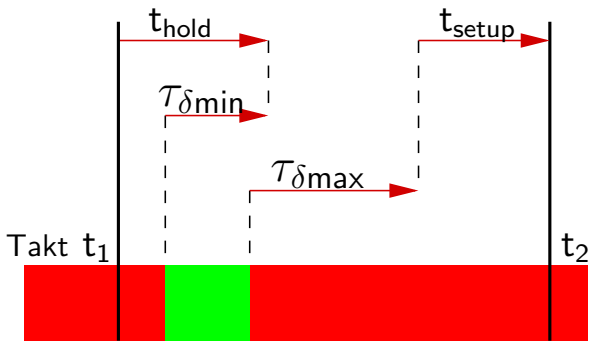




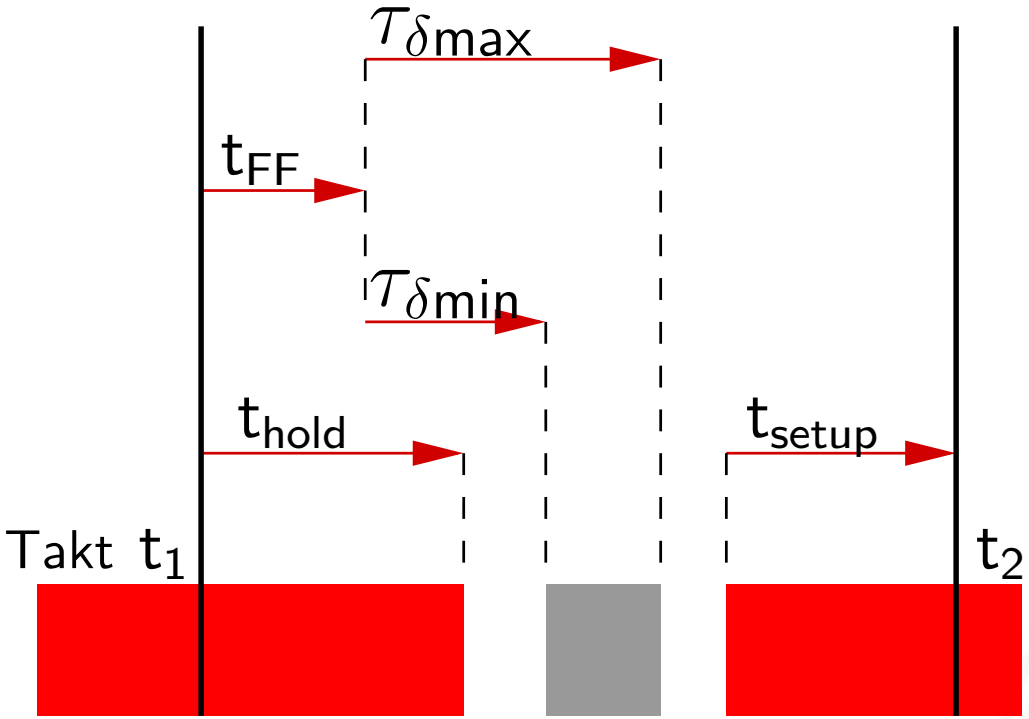
Zeitbedingungen: Eingangsvektor (cont.)

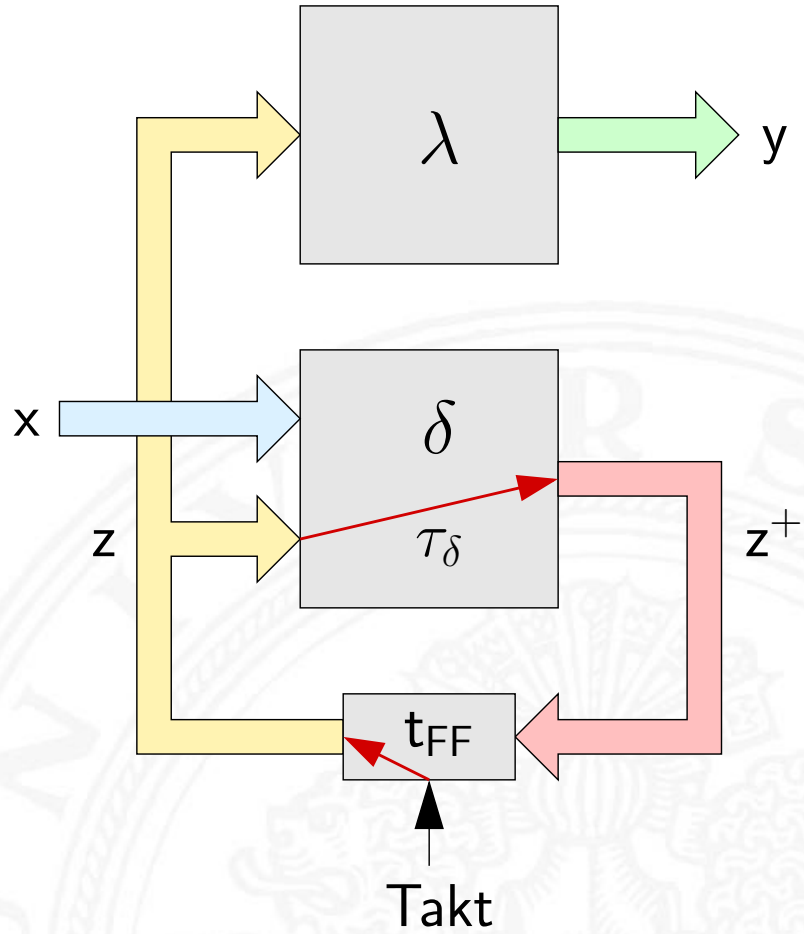
- Änderungen der Eingangswerte x werden beim Durchlaufen von δ mindestens um $\tau_{\delta_{\text{min}}}$, bzw. maximal um $\tau_{\delta_{\text{max}}}$ verzögert
- um die Haltezeit der Zeitglieder einzuhalten, darf x sich nach einem Taktimpuls frühestens zum Zeitpunkt $(t_1 + t_{\text{hold}} - \tau_{\delta_{\text{min}}})$ wieder ändern
- um die Vorlaufzeit vor dem nächsten Takt einzuhalten, muss x spätestens zum Zeitpunkt $(t_2 - t_{\text{setup}} - \tau_{\delta_{\text{max}}})$ wieder stabil sein

$\Rightarrow$ Änderungen dürfen nur im grün markierten Zeitintervall erfolgen



Zeitbedingungen: interner Zustand

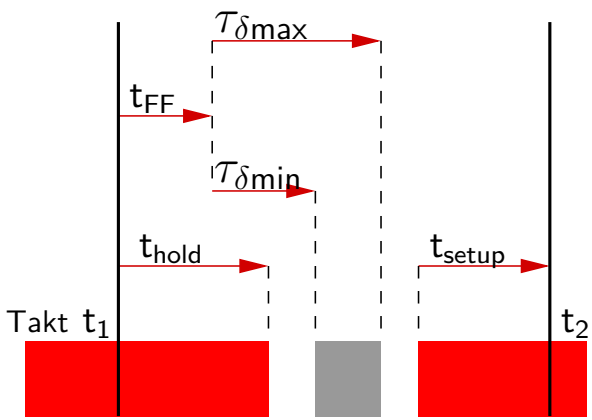




Zeitbedingungen: interner Zustand (cont.)

- zum Zeitpunkt t_1 wird ein Taktimpuls ausgelöst
- nach dem Taktimpuls vergeht die Zeit t_{FF}, bis die Zeitglieder (Flipflops) ihren aktuellen Eingangswert z^+ übernommen haben und als neuen Zustand z am Ausgang bereitstellen
- die neuen Werte von z laufen durch das δ-Schaltnetz, der schnellste Pfad ist dabei $\tau_{\delta_{min}}$ und der langsamste ist $\tau_{\delta_{max}}$

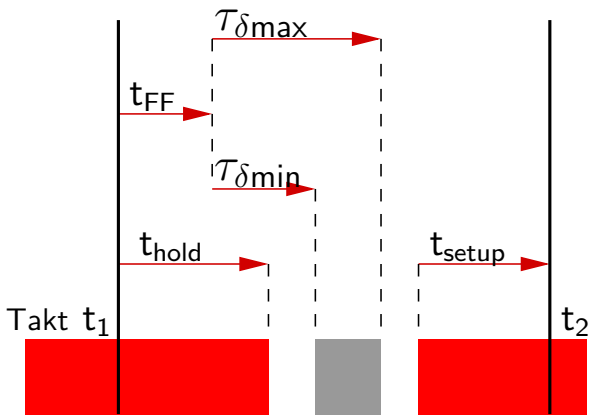
$\Rightarrow$ innerhalb der Zeitintervalls $(t_{FF} + \tau_{\delta_{min}})$ bis $(t_{FF} + \tau_{\delta_{max}})$ ändern sich die Werte des Folgezustands z^+ = grauer Bereich



Zeitbedingungen: interner Zustand (cont.)

- die Änderungen dürfen frühestens zum Zeitpunkt $(t_1 + t_{hold})$ beginnen, ansonsten würde Haltezeit verletzt

  ggf. muss $\tau_{\delta_{min}}$ vergrößert werden, um diese Bedingung einhalten zu können (zusätzliche Gatterverzögerungen)
- die Änderungen müssen sich spätestens bis zum Zeitpunkt $(t_2 - t_{setup})$ stabilisiert haben (der Vorlaufzeit der Flipflops vor dem nächsten Takt)



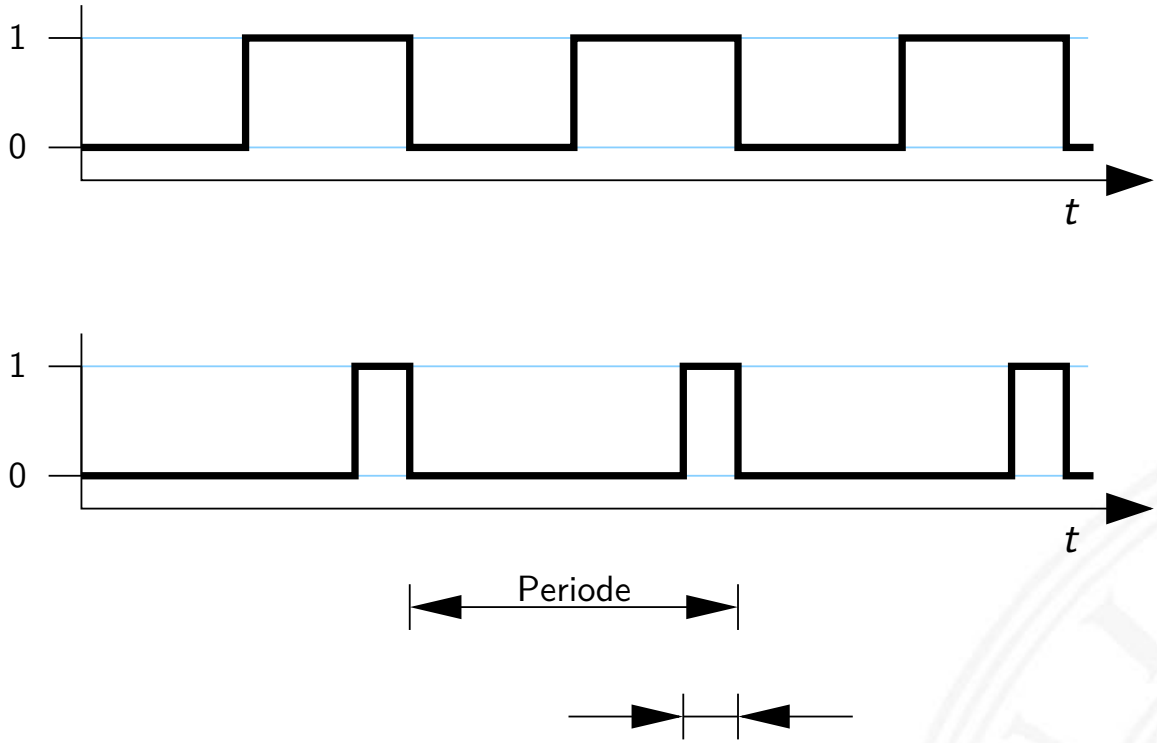
Maximale Taktfrequenz einer Schaltung

- aus obigen Bedingungen ergibt sich sofort die maximal zulässige Taktfrequenz einer Schaltung
- Umformen und Auflösen nach dem Zeitpunkt des nächsten Takts ergibt zwei Bedingungen

$$\Delta t \geq (t_{FF} + \tau_{\delta_{max}} + t_{setup}) \quad \text{und}$$
$$\Delta t \geq (t_{hold} + t_{setup})$$

- falls diese Bedingung verletzt wird („Übertakten“), kann es (datenabhängig) zu Fehlfunktionen kommen

Taktsignal: Prinzip

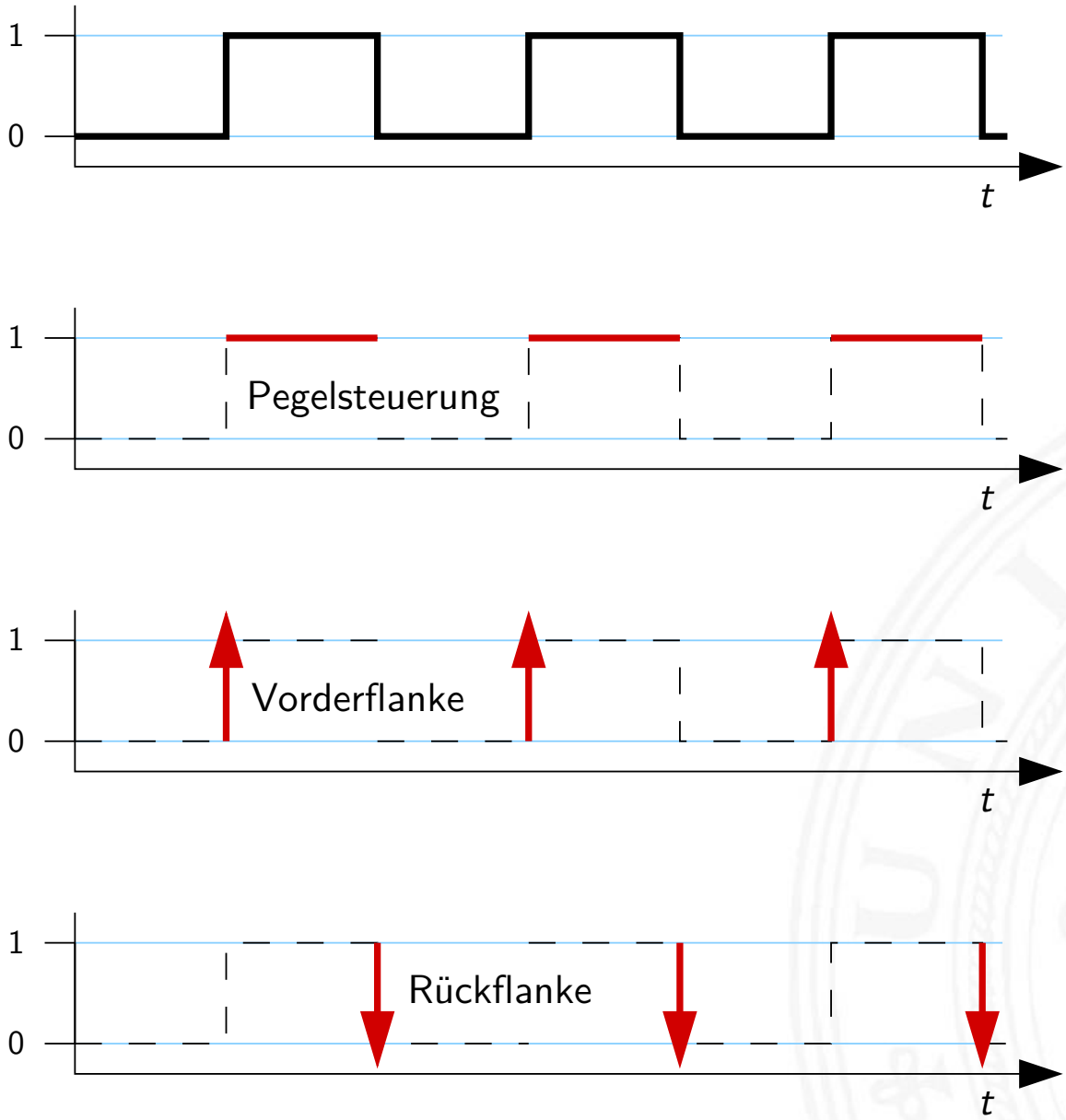


- periodisches digitales Signal, Frequenz f bzw. Periode τ
- oft symmetrisch
- asymmetrisch für Zweiphasentakt (s.u.)

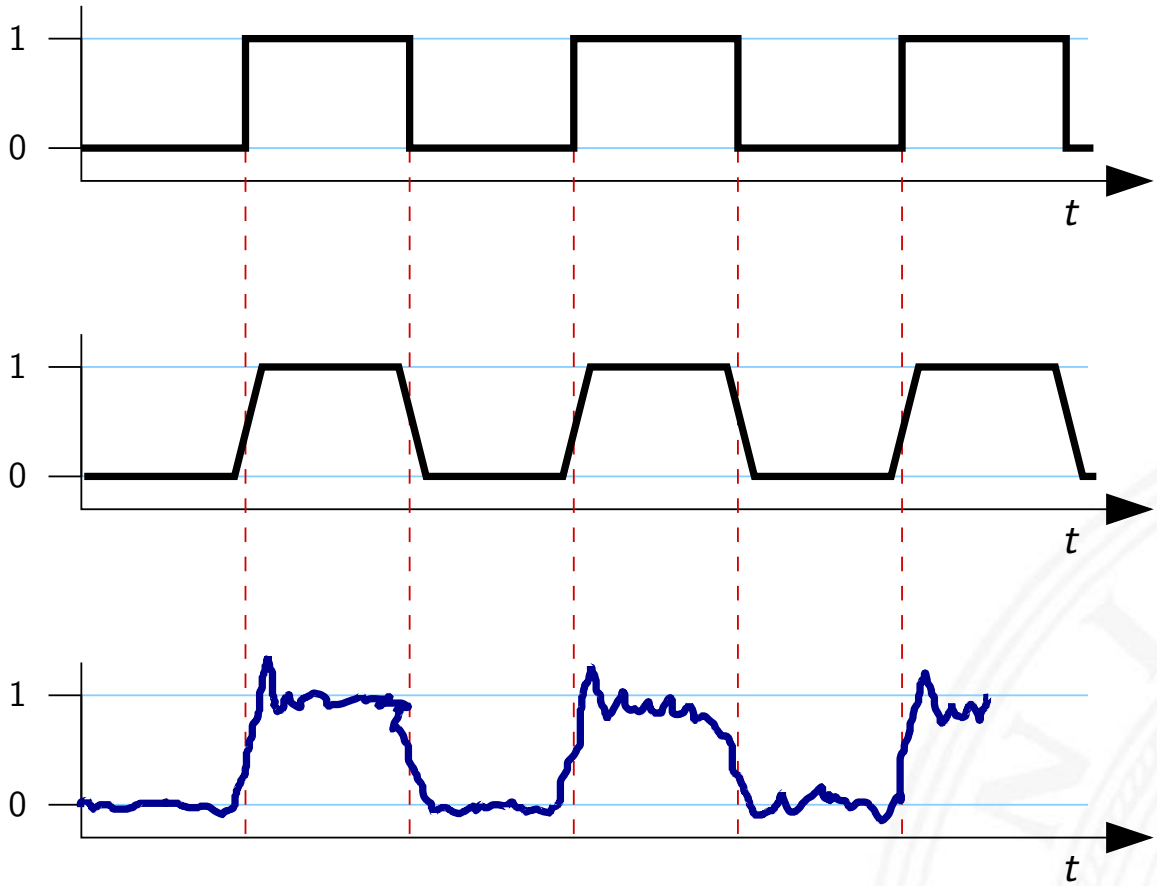
Taktsignal: Varianten

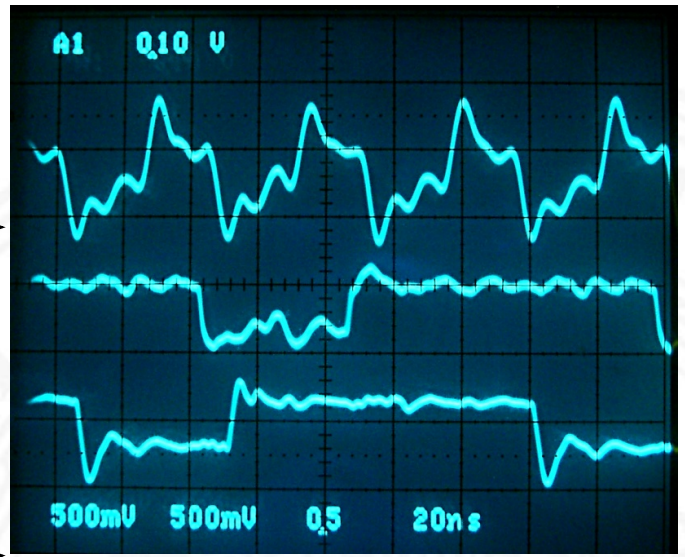
- **Pegelsteuerung**: Schaltung reagiert, während das Taktsignal den Wert 1 (bzw. 0) aufweist
- **Flankensteuerung**: Schaltung reagiert nur, während das Taktsignal seinen Wert wechselt
  - Vorderflankensteuerung: Wechsel von 0 nach 1
  - Rückflankensteuerung: –"– von 1 nach 0
- Zwei- und Mehrphasentakte

Taktsignal: Varianten (cont.)



Taktsignal: Prinzip und Realität

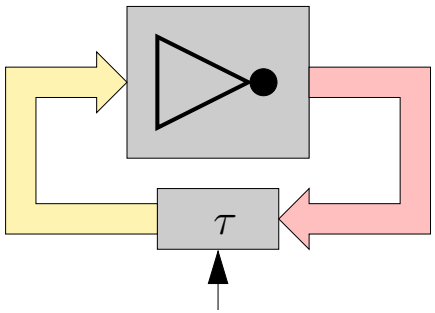




- Werteverläufe in realen Schaltungen stark gestört
- Überschwingen/Übersprechen benachbarter Signale
- Flankensteilheit nicht garantiert (bei starker Belastung)
  ggf. besondere Gatter („Schmitt-Trigger“)

Problem mit Pegelsteuerung

- während des aktiven Taktpegels werden Eingangswerte direkt übernommen
- falls invertierende Rückkopplungspfade in δ vorliegen, kommt es dann zu instabilen Zuständen (Oszillationen)



- einzelne pegelgesteuerte Zeitglieder (D-Latches) garantieren keine stabilen Zustände

$\Rightarrow$ Verwendung von je zwei pegelgesteuerten Zeitgliedern und Einsatz von Zweiphasentakt oder

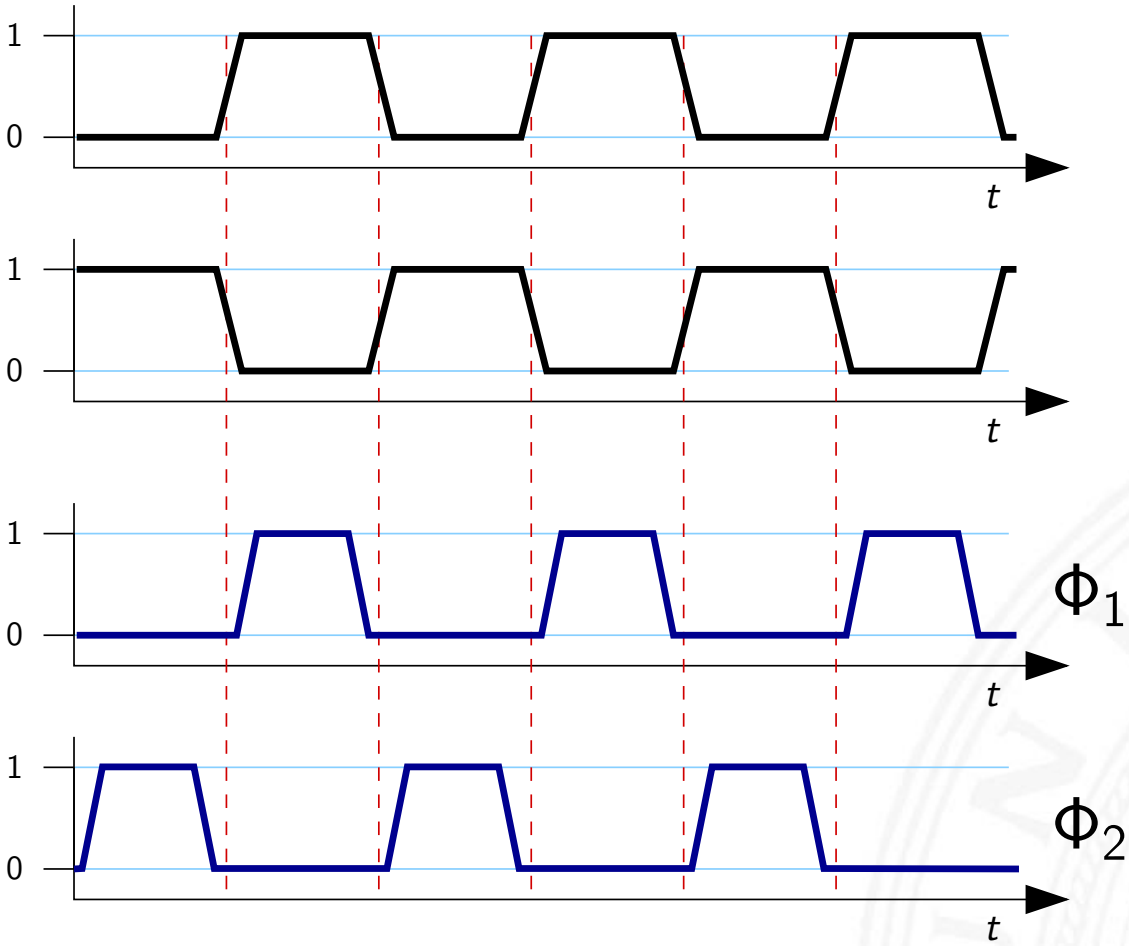
$\Rightarrow$ Verwendung flankengesteuerter D-Flipflops

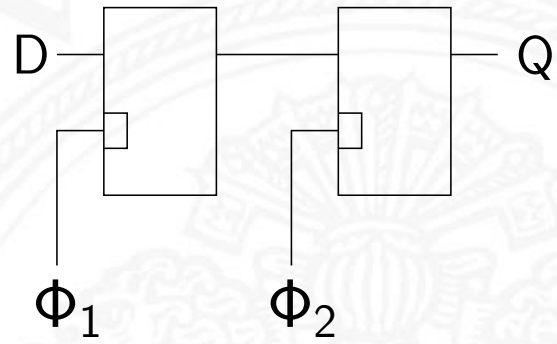
Zweiphasentakt

- pegelgesteuertes D-Latch ist bei aktivem Takt *transparent*
- rück-gekoppelte Werte werden sofort wieder durchgelassen
- Oszillation bei invertierten Rückkopplungen

- Reihenschaltung aus jeweils zwei D-Latches
- zwei separate Takte Φ_1 und Φ_2
  - bei Takt Φ_1 übernimmt vorderes Flipflop den Wert
    erst bei Takt Φ_2 übernimmt hinteres Flipflop
  - vergleichbar Master-Slave Prinzip bei D-FF aus Latches

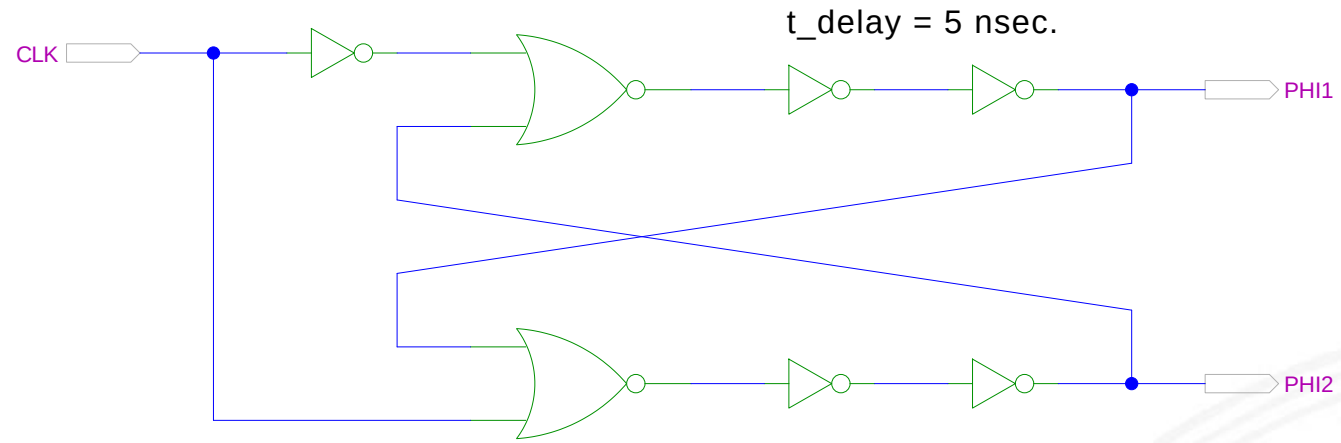
Zweiphasentakt (cont.)

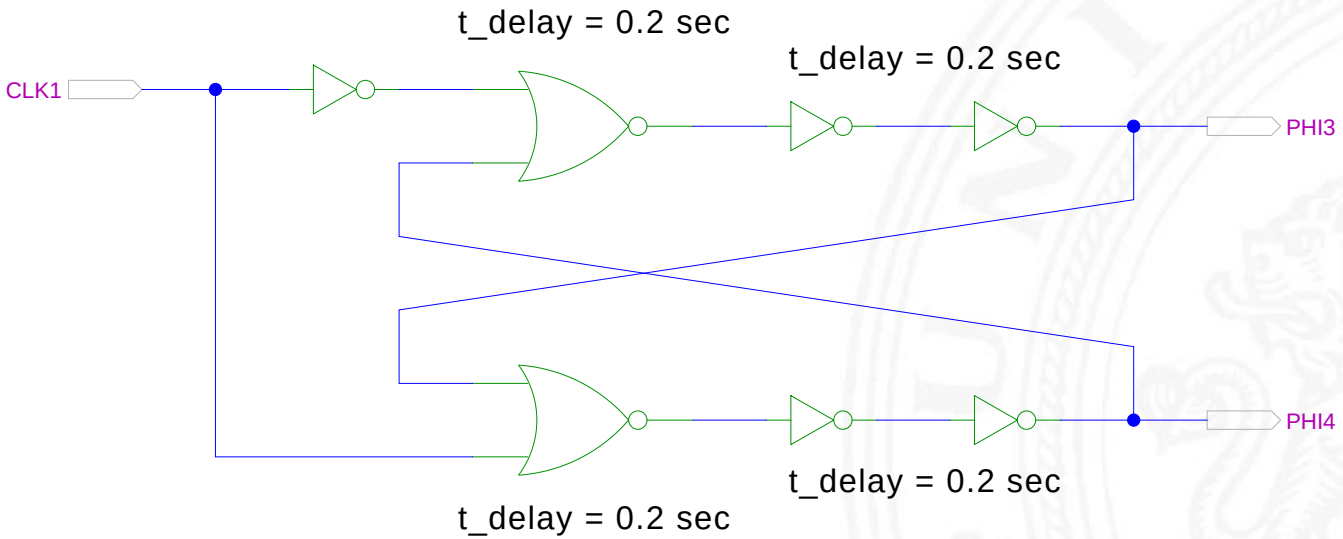




- nicht überlappender Takt mit Phasen Φ_1 und Φ_2
- vorderes D-Latch übernimmt Eingangswert D während Φ_1 bei Φ_2 übernimmt das hintere D-Latch und liefert Q

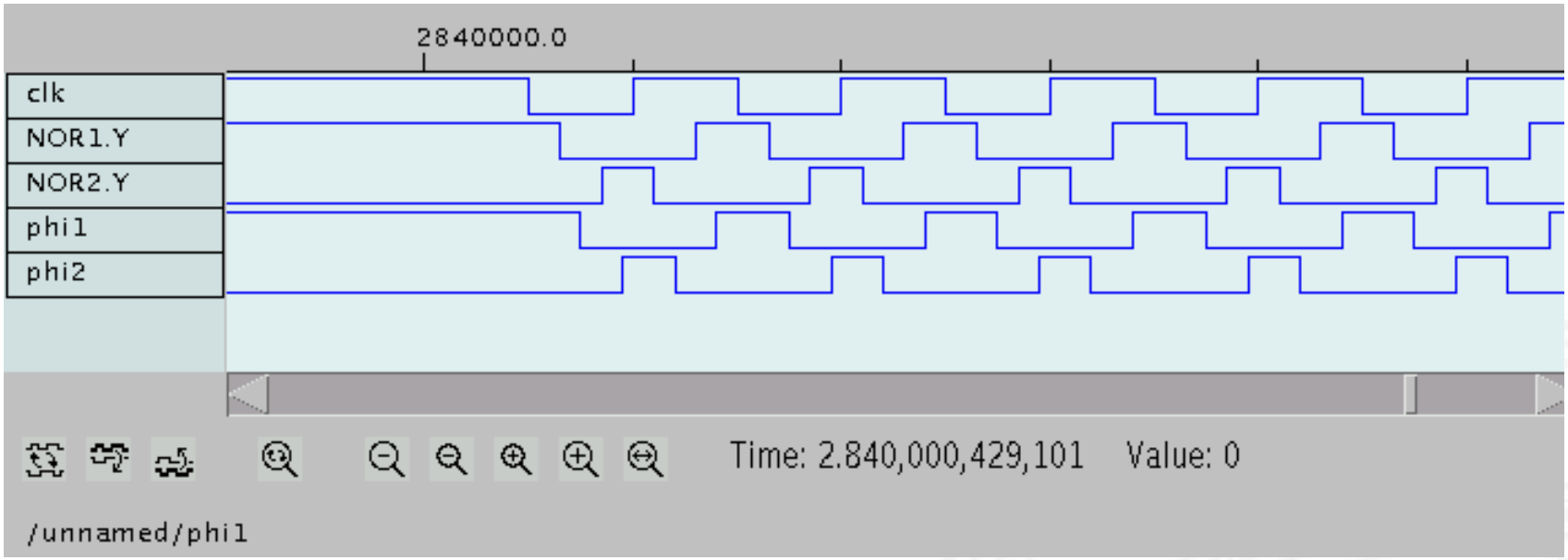
Zweiphasentakt: Erzeugung





[HenHA] Hades Demo: `12-gatedelay/40-tpcg/two-phase-clock-gen`

Zweiphasentakt: Erzeugung (cont.)



- Verzögerungen geeignet wählen
- Eins-Phasen der beiden Takte c_1 und c_2 sauber getrennt

$\Rightarrow$ nicht-überlappende Taktimpulse zur Ansteuerung von Schaltungen mit 2-Phasen-Taktung

Beschreibung von Schaltwerken

- viele verschiedene Möglichkeiten
- graphisch oder textuell

- algebraische Formeln/Gleichungen
- Flusstafel und Ausgangstafel

- Zustandsdiagramm
- State-Charts (hierarchische Zustandsdiagramme)

- Programme (Hardwarebeschreibungssprachen)

Flusstafel und Ausgangstafel

- entspricht der Funktionstabelle von Schaltnetzen

- **Flusstafel**: Tabelle für die Folgezustände als Funktion des aktuellen Zustands und der Eingabewerte

= beschreibt das δ-Schaltnetz

- **Ausgangstafel**: Tabelle für die Ausgabewerte als Funktion des aktuellen Zustands (und der Eingabewerte [Mealy-Modell])

= beschreibt das λ-Schaltnetz

Beispiel: Ampel

- vier Zustände: {rot, rot-gelb, grün, gelb}
- Codierung beispielsweise als 2-bit Vektor (z_1, z_0)

- Flusstafel

Zustand	Codierung		Folgezustand	
	z_1	z_0	z_1^+	z_0^+
rot	0	0	0	1
rot-gelb	0	1	1	0
grün	1	0	1	1
gelb	1	1	0	0

Beispiel: Ampel (cont.)

- Ausgangstafel

Zustand	Codierung		Ausgänge		
	z_1	z_0	rt	ge	gr
rot	0	0	1	0	0
rot-gelb	0	1	1	1	0
grün	1	0	0	0	1
gelb	1	1	0	1	0

- Funktionstabelle für drei Schaltfunktionen
- Minimierung z.B. mit KV-Diagrammen

Zustandsdiagramm

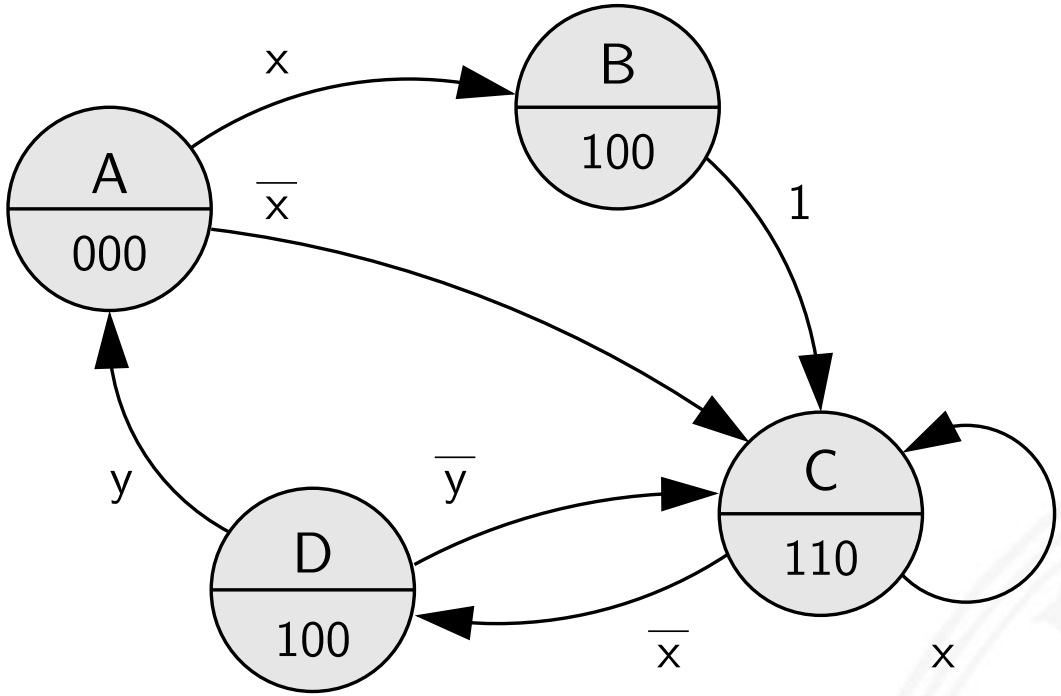
- **Zustandsdiagramm**: Grafische Darstellung eines Schaltwerks
- je ein Knoten für jeden Zustand
- je eine Kante für jeden möglichen Übergang

- Knoten werden passend benannt
- Kanten werden mit den Eingabemustern gekennzeichnet, bei denen der betreffende Übergang auftritt

- Moore-Schaltwerke: Ausgabe wird zusammen mit dem Namen im Knoten notiert
- Mealy-Schaltwerke: Ausgabe hängt vom Input ab und wird an den Kanten notiert

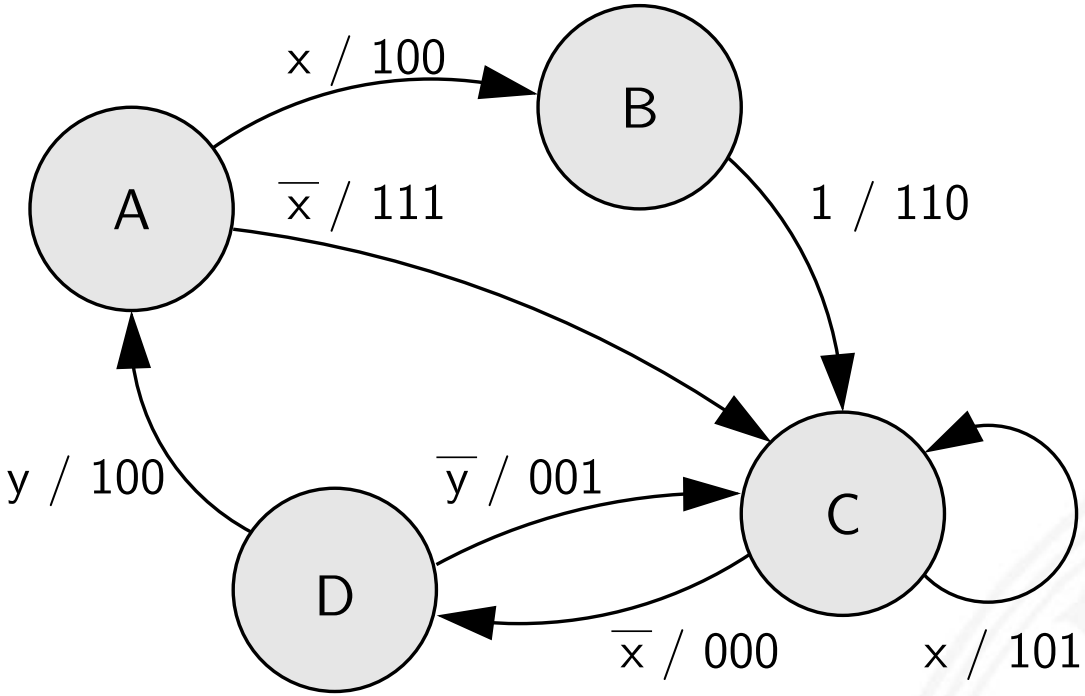
siehe auch en.wikipedia.org/wiki/State_diagram

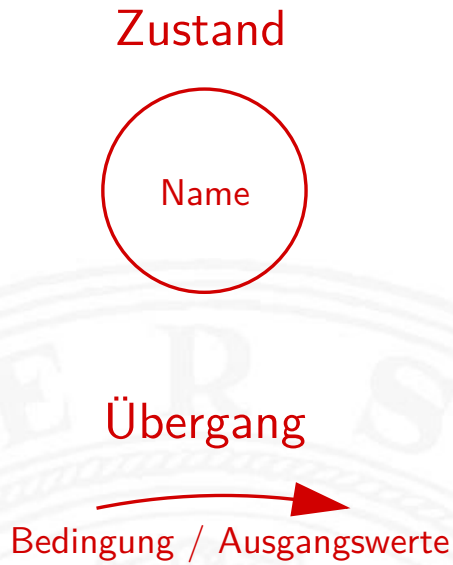
Zustandsdiagramm: Moore-Automat



- Ausgangswerte hängen nur vom Zustand ab
- können also im jeweiligen Knoten notiert werden
- Übergänge werden als Pfeile mit der Eingangsbelegung notiert, die den Übergang aktiviert
- ggf. Startzustand markieren (z.B. Segment, doppelter Kreis)

Zustandsdiagramm: Mealy-Automat





- Ausgangswerte hängen nicht nur vom Zustand sondern auch von den Eingabewerten ab
- Ausgangswerte an den zugehörigen Kanten notieren
- übliche Notation: *Eingangsbelegung / Ausgangswerte*

„State-Charts“

- erweiterte Zustandsdiagramme

1. Hierarchien, erlauben Abstraktion
   - Knoten repräsentieren entweder einen Zustand
   - oder einen eigenen (Unter-) Automaten
   - *History*-, *Default*-Mechanismen
2. Nebenläufigkeit, parallel arbeitende FSMs
3. Timer, Zustände nach max. Zeit verlassen

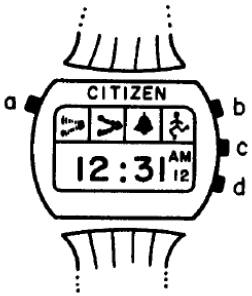
- beliebte Spezifikation für komplexe Automaten, eingebettete Systeme, Kommunikationssysteme, Protokolle etc.

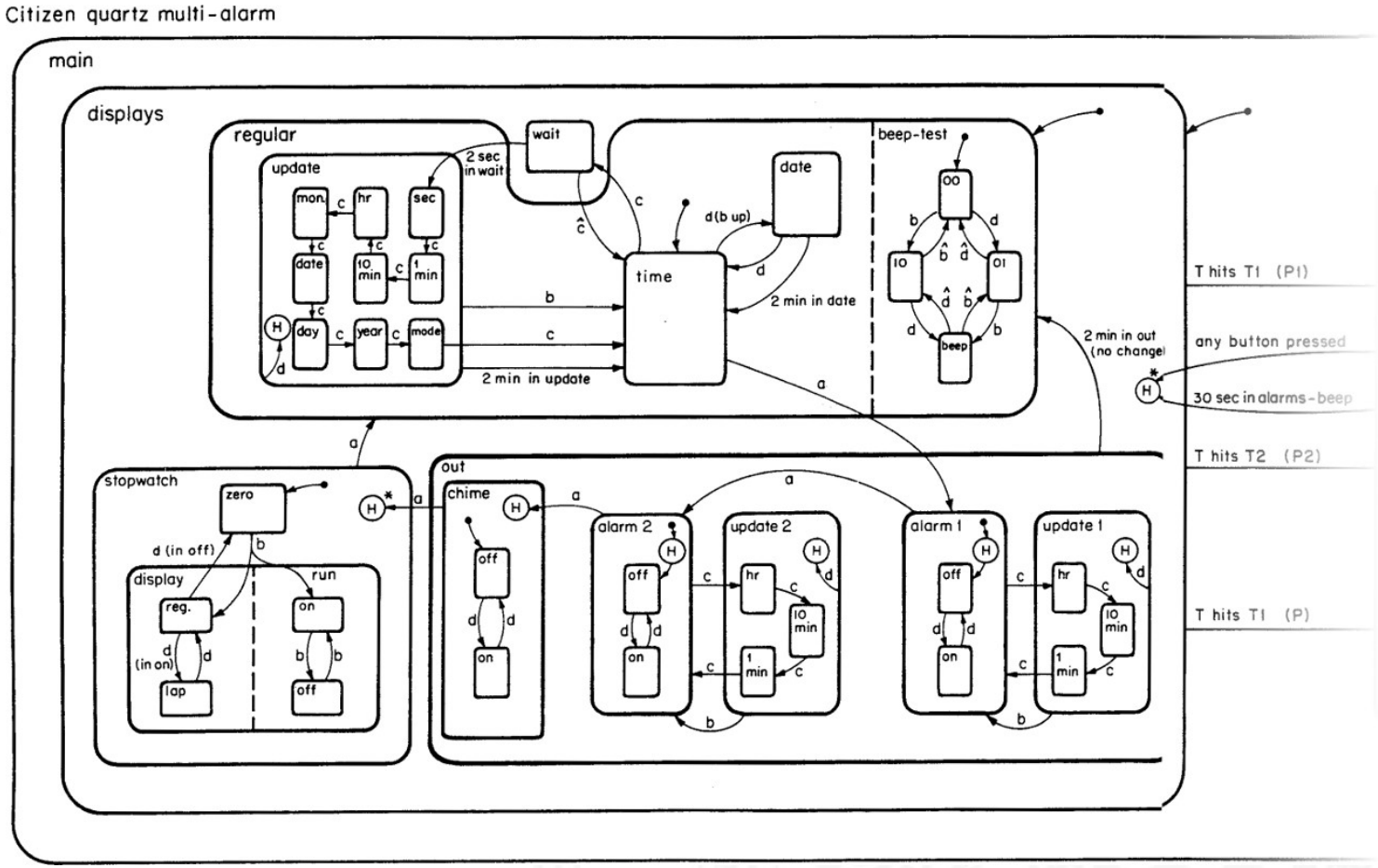
- David Harel, *Statecharts – A visual formalism for complex systems*, CS84-05, Department of Applied Mathematics, The Weizmann Institute of Science, 1984 [Har87]

www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf

„State-Charts“ (cont.)

- Beispiel Digitaluhr





Endliche Automaten

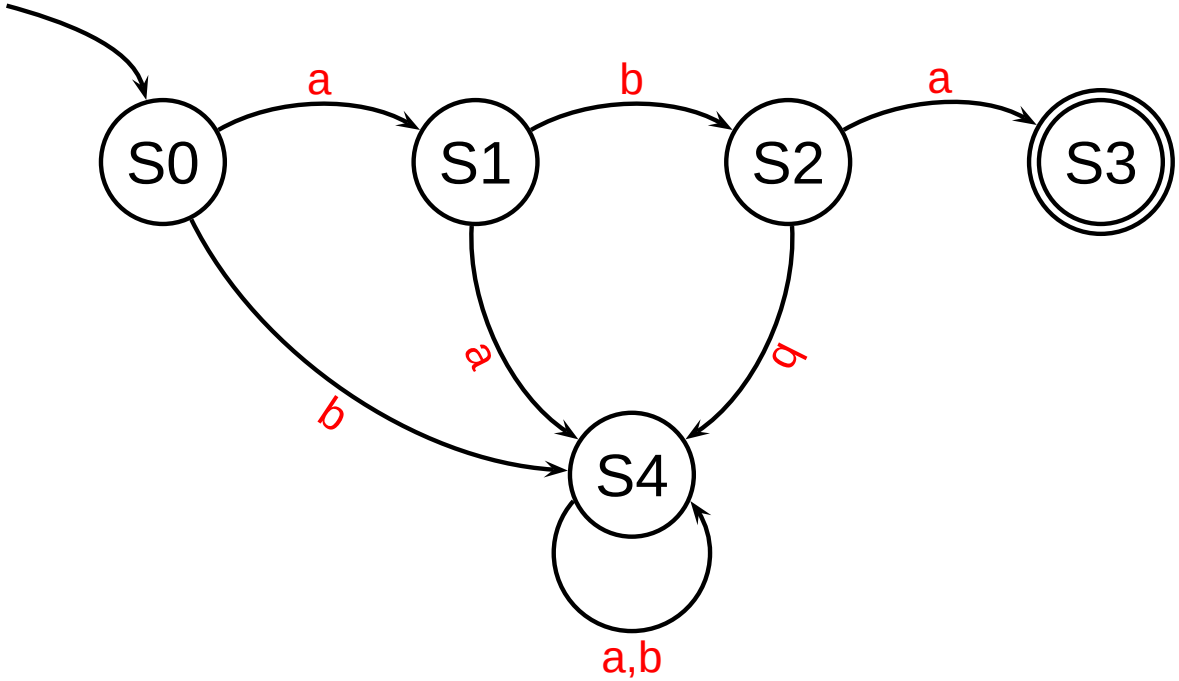
- eines der gängigen Konzepte der Informatik
- Modellierung, Entwurf und Simulation
  - zeitliche Abfolgen interner Systemzustände
  - bedingte Zustandswechsel
  - Reaktionen des Systems auf „Ereignisse“
  - Folgen von Aktionen
  - ...
- weitere „*spezielle*“ Anwendungsszenarien
  - verteilte Systeme (Client-Server etc.)
  - Echtzeitsysteme, ggf. mit Erweiterungen
  - eingebettete Systeme
  - ...

zahlreiche Beispiele

Endliche Automaten (cont.)

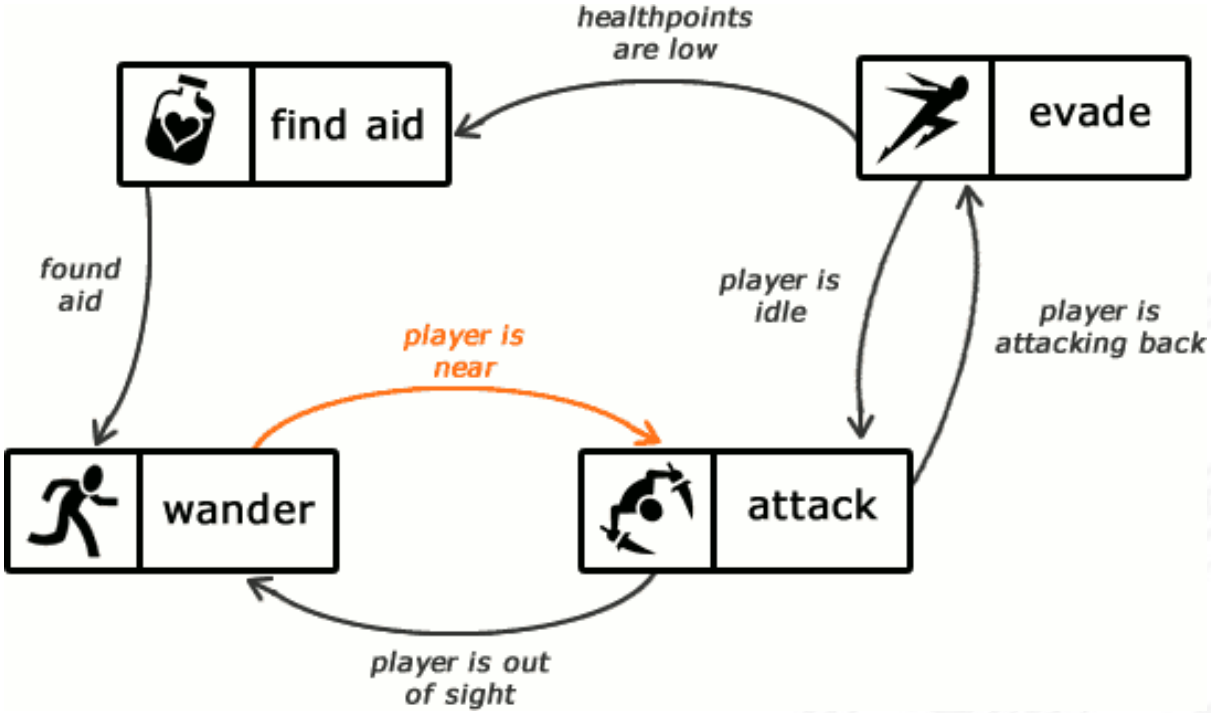
- in der Programmierung ...

  Erkennung des Worts: „a b a“



Endliche Automaten (cont.)

Game-Design: Verhalten eines Bots

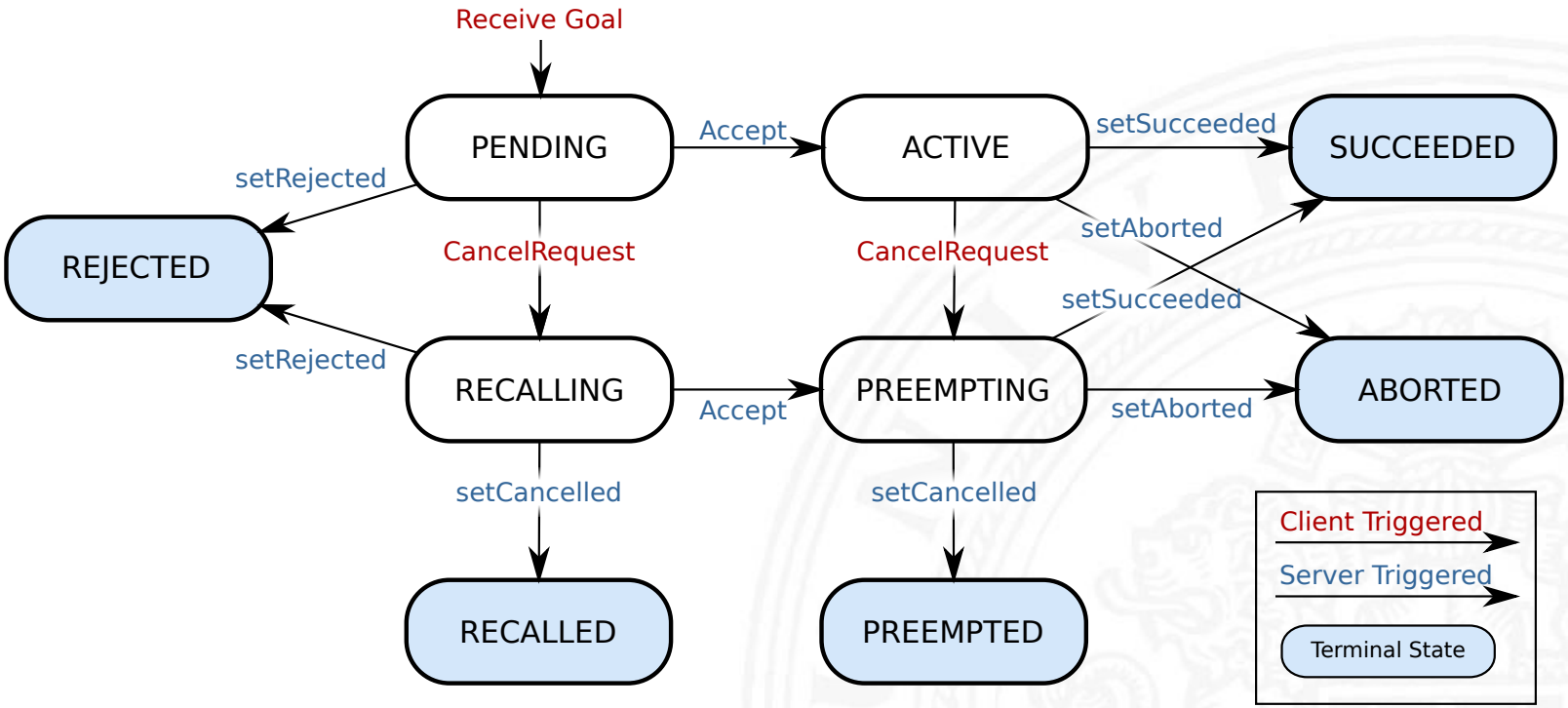


gamedevelopment.tutsplus.com/tutorials/
finite-state-machines-theory-and-implementation--gamedev-11867

Endliche Automaten (cont.)

- Beschreibung von Protokollen
- Verhalten verteilter Systeme: Client-Server Architektur

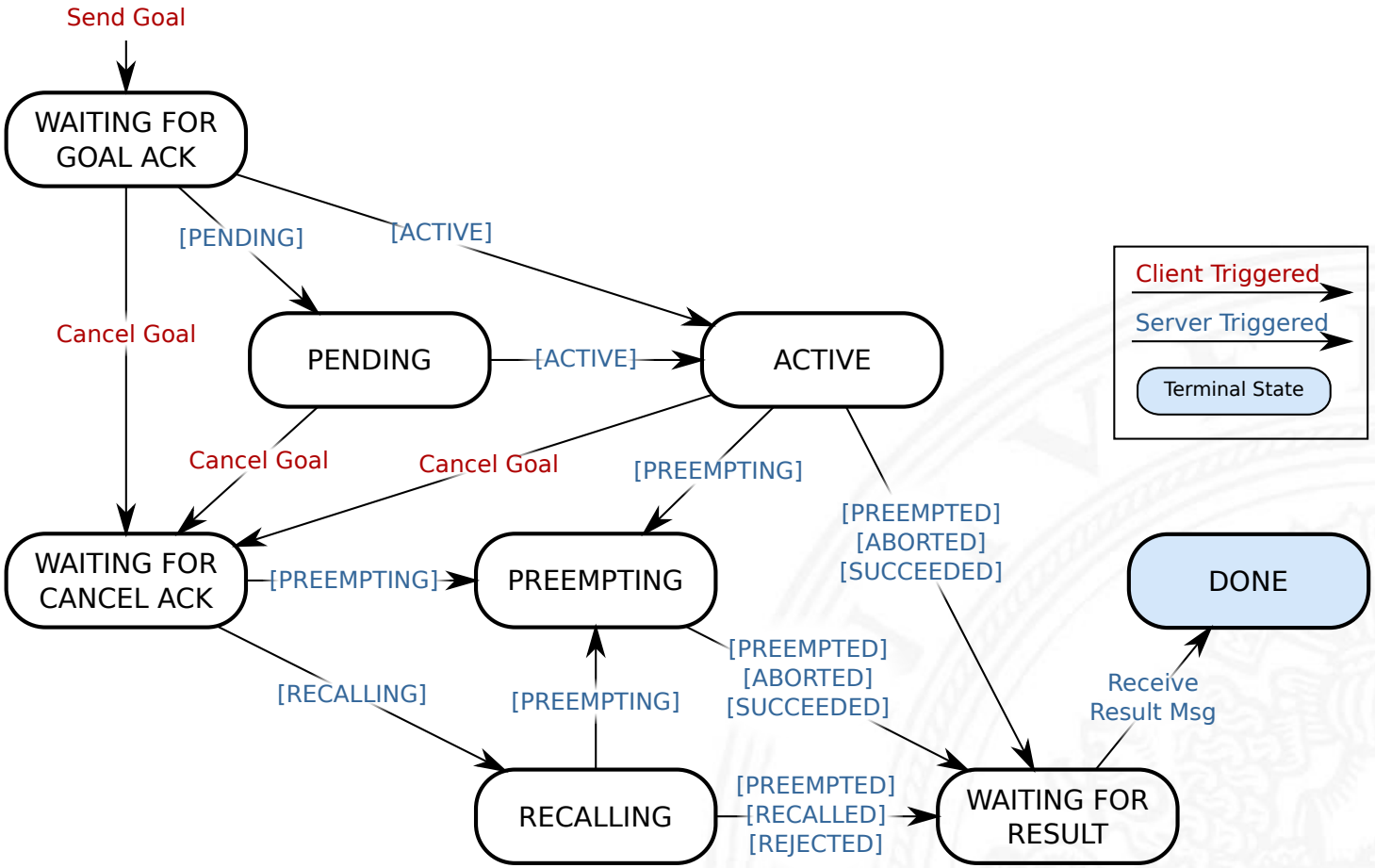
Server State Transitions



wiki.ros.org/actionlib/DetailedDescription

Endliche Automaten (cont.)

Client State Transitions

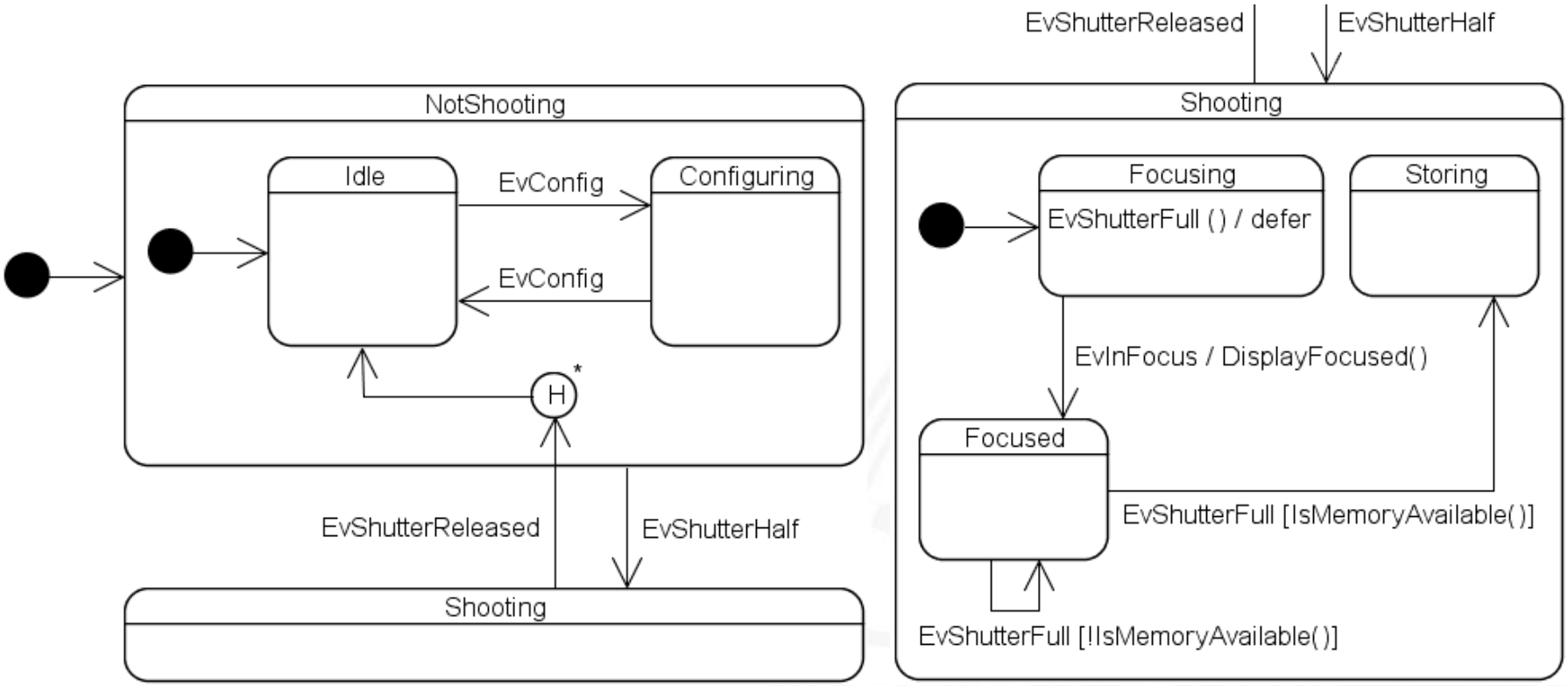


wiki.ros.org/actionlib/DetailedDescription

Endliche Automaten (cont.)

- Unterstützung durch Bibliotheken und Werkzeuge

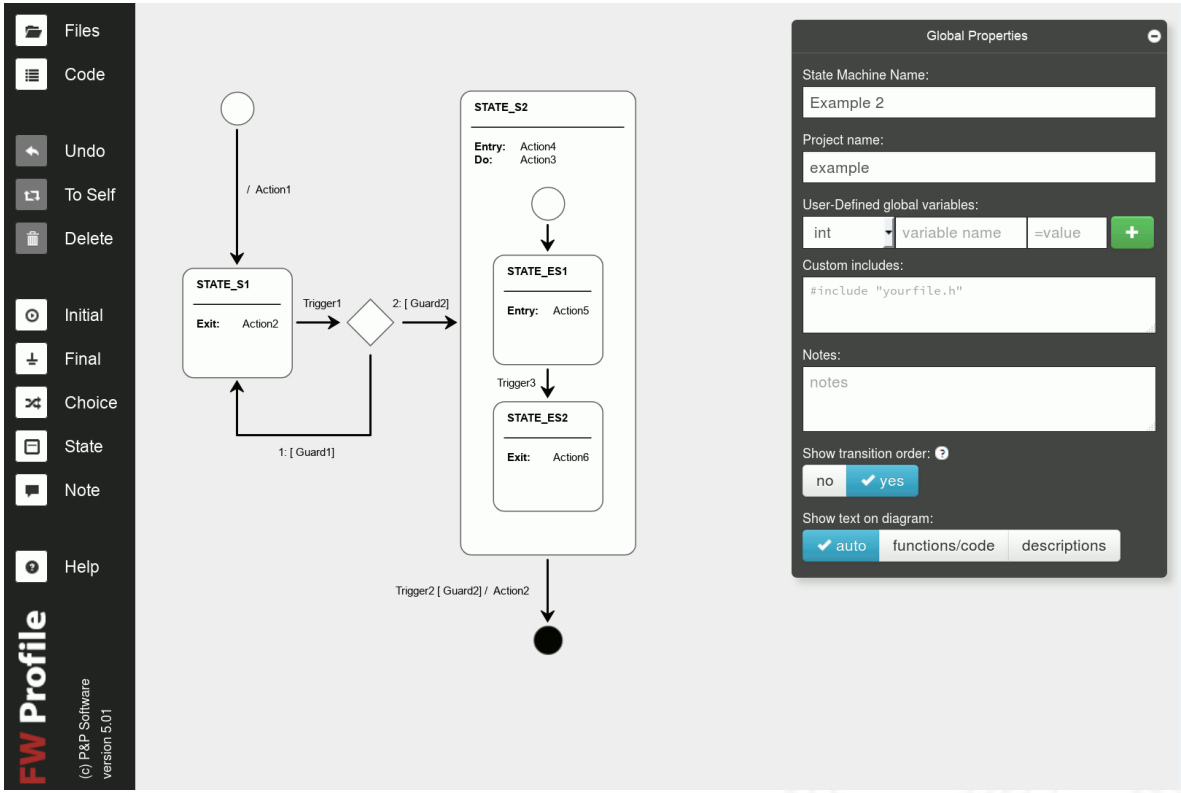
  State-Chart Bibliothek: Beispiel Digitalkamera



`www.boost.org/doc/libs/1_71_0/libs/statechart/doc`

Endliche Automaten (cont.)

FSM Editor / Code-Generator



`github.com/pnp-software/fwprofile`, `pnp-software.com/fwprofile`

⇒ beliebig viele weitere Beispiele . . .
„Endliche Automaten“ werden in RSB nur hardwarenah genutzt

Hardwarebeschreibungssprachen

- Beschreibung eines Schaltwerks als Programm:

- normale Hochsprachen — C, Java
- spezielle Bibliotheken für normale Sprachen — SystemC, Hades
- spezielle Hardwarebeschreibungssprachen — Verilog, VHDL

- Hardwarebeschreibungssprachen unterstützen Modellierung paralleler Abläufe und des Zeitverhaltens einer Schaltung
- wird hier nicht vertieft
- lediglich zwei Beispiele: D-Flipflop in Verilog und VHDL

D-Flipflop in Verilog


```
module dff (clock, reset, din, dout);  // Black-Box Beschreibung
input clock, reset, din;               // Ein- und Ausgänge
output dout;                           //

reg dout;                              // speicherndes Verhalten

  always @(posedge clock or reset)     // Trigger für Code
  begin                                //
    if (reset)                         // async. Reset
      dout = 1'b0;                     //
    else                               // implizite Taktvorderflanke
      dout = din;                      //
    end                                //
endmodule
```

- Deklaration eines Moduls mit seinen Ein- und Ausgängen
- Deklaration der speichernden Elemente („reg“)
- Aktivierung des Codes bei Signalwechseln („posedge clock“)

D-Flipflop in VHDL

Very High Speed Integrated Circuit Hardware Description Language


```
library ieee;
use ieee.std_logic_1164.all;

entity dff is                           -- Black-Box Beschreibung
port (  clock   : in  std_logic;        -- Ein- und Ausgänge
        reset   : in  std_logic;        --
        din     : in  std_logic;        --
        dout    : out std_logic);       --
end entity dff;

architecture behav of dff is            -- Verhaltensmodell
begin                                   --
  dff_p: process (reset, clock) is      -- Trigger für Prozess
  begin                                 --
    if reset = '1' then                 -- async. Reset
        dout <= '0';                    --
    elsif rising_edge(clock) then       -- Taktvorderflanke
        dout <= din;                    --
    end if;                             --
  end process dff_p;                    --
end architecture behav;
```

Entwurf von Schaltwerken: sechs Schritte

1. Spezifikation (textuell oder graphisch, z.B. Zustandsdiagramm)
2. Aufstellen einer formalen Übergangstabelle
3. Reduktion der Zahl der Zustände
4. Wahl der Zustandscodierung und Aufstellen der Übergangstabelle
5. Minimierung der Schaltnetze
6. Überprüfung des realisierten Schaltwerks

ggf. mehrere Iterationen

Entwurf von Schaltwerken: Zustandscodierung

Vielfalt möglicher Codierungen

- binäre Codierung: minimale Anzahl der Zustände
- einschrittige Codes
- one-hot Codierung: ein aktives Flipflop pro Zustand
- applikationsspezifische Zwischenformen

- es gibt Entwurfsprogramme zur Automatisierung
- gemeinsame Minimierung des Realisierungsaufwands von Ausgangsfunktion, Übergangsfunktion und Speichergliedern

Entwurf von Schaltwerken: Probleme

Entwurf ausgehend von Funktionstabellen problemlos

- alle Eingangsbelegungen und Zustände werden berücksichtigt
- don't-care Terme können berücksichtigt werden

zwei typische Fehler bei Entwurf ausgehend vom Zustandsdiagramm

- mehrere aktive Übergänge bei bestimmten Eingangsbelegungen
  $\Rightarrow$ Widerspruch
- keine Übergänge bei bestimmten Eingangsbelegungen
  $\Rightarrow$ Vollständigkeit

Überprüfung der Vollständigkeit

p Zustände, Zustandsdiagramm mit Kanten $h_{ij}(x)$:
Übergang von Zustand i nach Zustand j unter Belegung x

- für jeden Zustand überprüfen:

  kommen alle (spezifizierten) Eingangsbelegungen auch tatsächlich in Kanten vor?

$$\forall i : \quad \bigvee_{j=0}^{2^p-1} h_{ij}(x) = 1$$

Überprüfung der Widerspruchsfreiheit

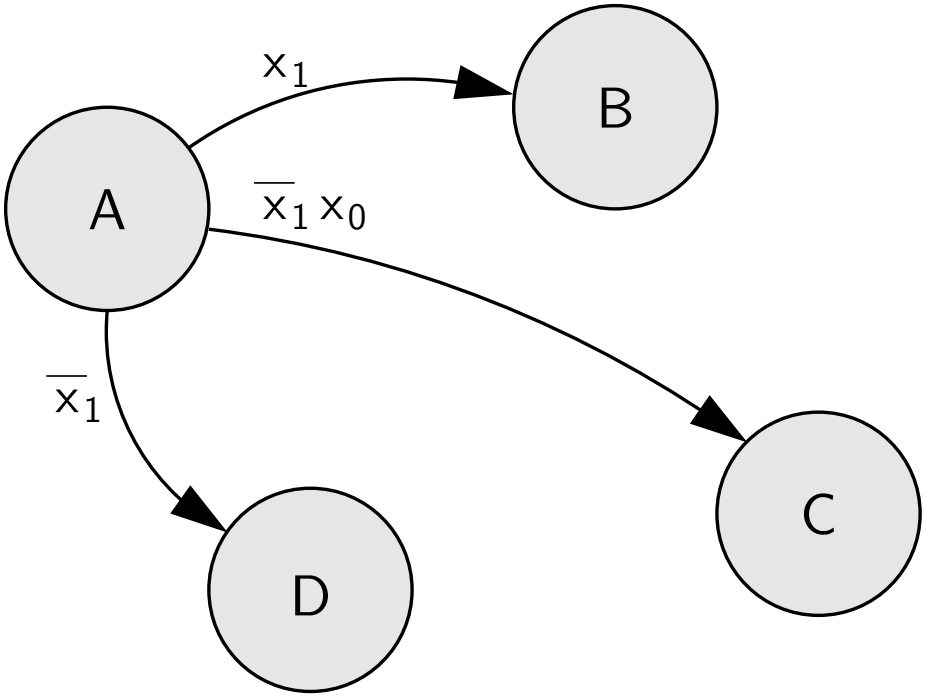
p Zustände, Zustandsdiagramm mit Kanten $h_{ij}(x)$:
Übergang von Zustand i nach Zustand j unter Belegung x

- für jeden Zustand überprüfen:

  kommen alle (spezifizierten) Eingangsbelegungen nur einmal vor?

$$\forall i : \bigvee_{j,k=0,\ j\neq k}^{2^{p-1}} \left(h_{ij}(x) \wedge h_{ik}(x)\right) = 0$$

Vollständigkeit und Widerspruchsfreiheit: Beispiel



- Zustand A, Vollständigkeit: $x_1 \vee \overline{x_1}\,x_0 \vee \overline{x_1} = 1$ vollständig
- Zustand A, Widerspruchsfreiheit: alle Paare testen

  $x_1 \wedge \overline{x_1}\,x_0 = 0$ ok

  $x_1 \wedge \overline{x_1} = 0$ ok

  $\overline{x_1}\,x_0 \wedge \overline{x_1} \neq 0$ für $x_1 = 0$ und $x_0 = 1$ beide Übergänge aktiv

Schaltwerke: Beispiele

- Verkehrsampel
  - drei Varianten mit unterschiedlicher Zustandscodierung
- Zählschaltungen
  - einfacher Zähler, Zähler mit Enable (bzw. Stop),
  - Vorwärts-Rückwärts Zähler, Realisierung mit JK-Flipflops und D-Flipflops
- Digitaluhr
  - BCD Zähler
  - DCF77 Protokoll
  - Siebensegment-Anzeige

Schaltwerksentwurf: Ampel

Beispiel Verkehrsampel:

- drei Ausgänge: {rot, gelb, grün}
- vier Zustände: {rot, rot-gelb, grün, gelb}
- zunächst kein Eingang, feste Zustandsfolge wie oben

- Aufstellen des Zustandsdiagramms
- Wahl der Zustandscodierung
- Aufstellen der Tafeln für δ- und λ-Schaltnetz
- anschließend Minimierung der Schaltnetze
- Realisierung (je 1 D-Flipflop pro Zustandsbit) und Test

Schaltwerksentwurf: Ampel – Variante 1

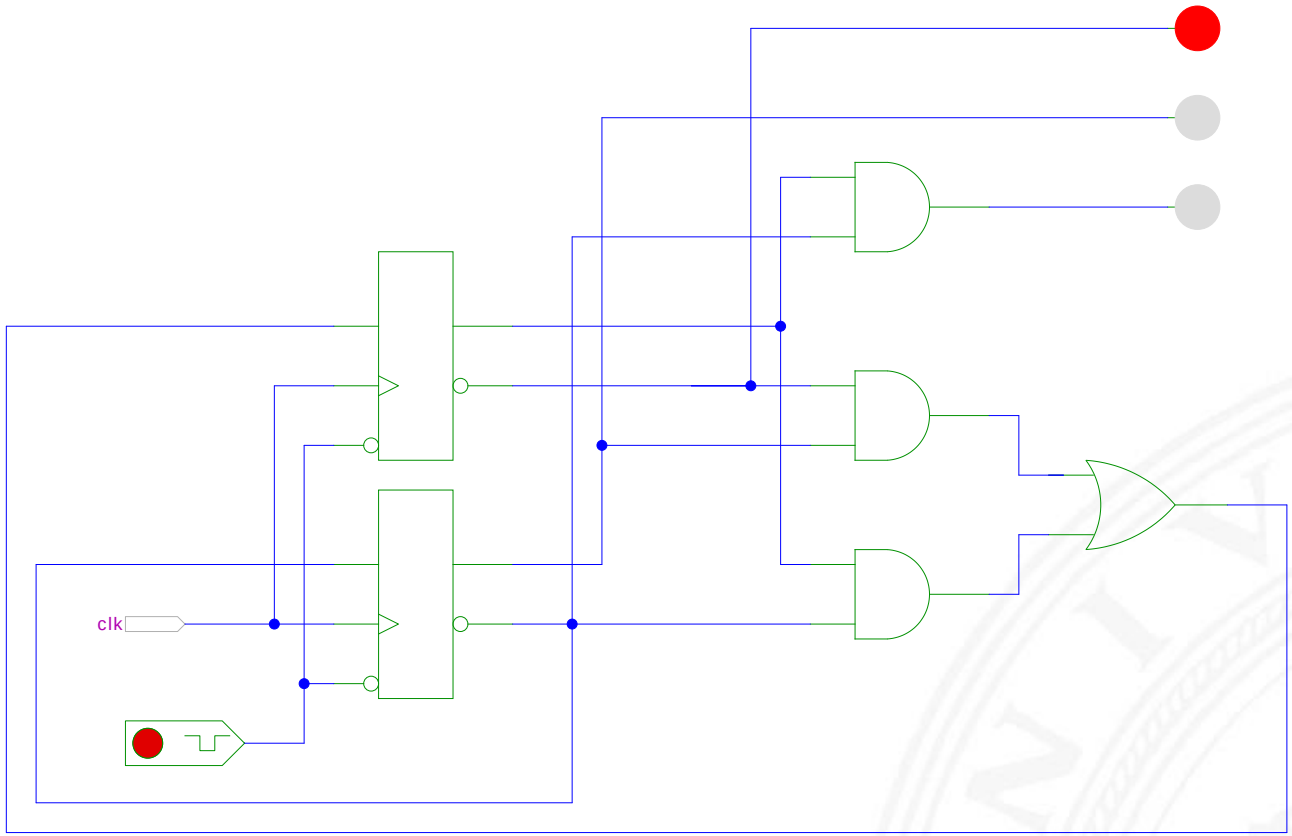
- vier Zustände, Codierung als 2-bit Vektor (z_1, z_0)
- Fluss- und Ausgangstafel für binäre Zustandscodierung

Zustand	Codierung		Folgezustand		Ausgänge		
	z_1	z_0	z_1^+	z_0^+	rt	ge	gr
rot	0	0	0	1	1	0	0
rot-gelb	0	1	1	0	1	1	0
grün	1	0	1	1	0	0	1
gelb	1	1	0	0	0	1	0

- resultierende Schaltnetze

$$z_1^+ = (z_1 \wedge \overline{z_0}) \vee (\overline{z_1} \wedge z_0) = z_1 \oplus z_0$$
$$z_0^+ = \overline{z_0}$$
$$rt = \overline{z_1}$$
$$ge = z_0$$
$$gr = (z_1 \wedge \overline{z_0})$$

Schaltwerksentwurf: Ampel – Variante 1 (cont.)



[HenHA] Hades Demo: `18-fsm/10-trafficlight/ampel_41`

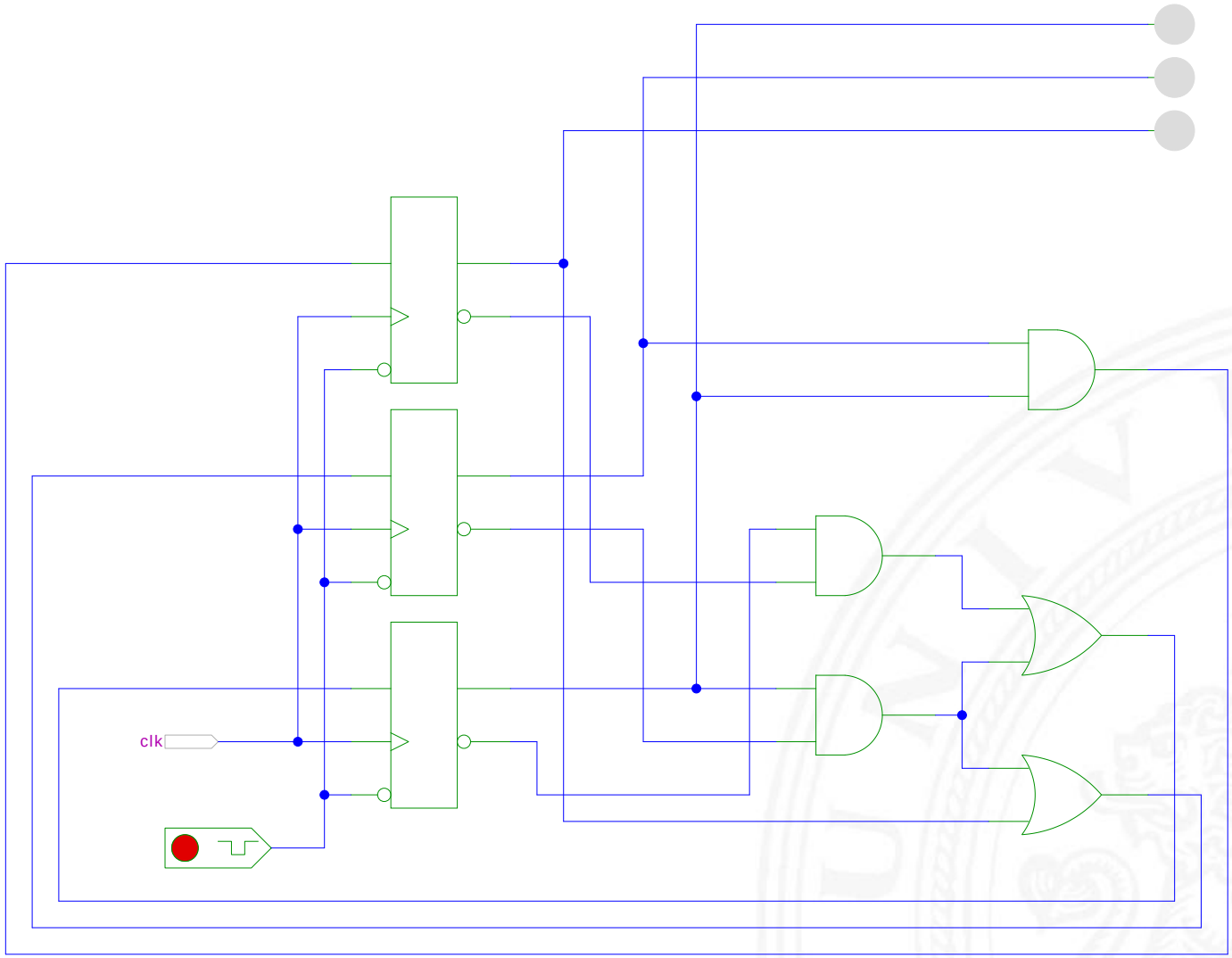
Schaltwerksentwurf: Ampel – Variante 2

- 4+1 Zustände, Codierung als 3-bit Vektor (z_2, z_1, z_0)
  Reset-Zustand: alle Lampen aus
- Zustandsbits korrespondieren mit den aktiven Lampen:
  $gr = z_2$, $ge = z_1$ und $rt = z_0$

Zustand	Codierung			Folgezustand		
	z_2	z_1	z_0	z_2^+	z_1^+	z_0^+
reset	0	0	0	0	0	1
rot	0	0	1	0	1	1
rot-gelb	0	1	1	1	0	0
grün	1	0	0	0	1	0
gelb	0	1	0	0	0	1

- benutzt 1-bit zusätzlich für die Zustände
- Ausgangsfunktion λ minimal: entfällt
- Übergangsfunktion δ: $z_2^+ = (z_1 \wedge z_0)$ $z_1^+ = z_2 \vee (\overline{z_1} \wedge z_0)$
  $z_0^+ = (\overline{z_2} \wedge \overline{z_0}) \vee (\overline{z_1} \wedge z_0)$

Schaltwerksentwurf: Ampel – Variante 2 (cont.)



[HenHA] Hades Demo: `18-fsm/10-trafficlight/ampel_42`

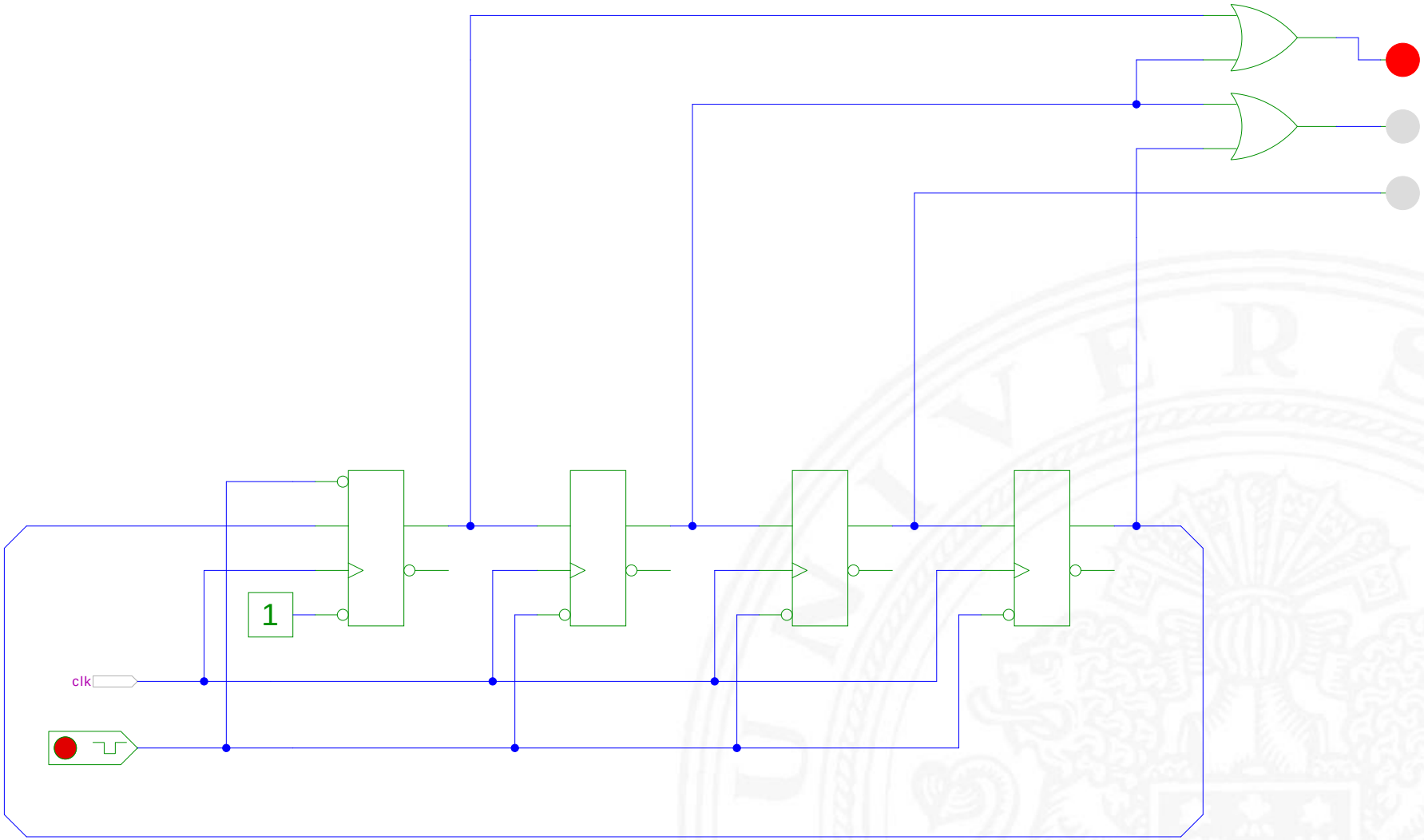
Schaltwerksentwurf: Ampel – Variante 3

- vier Zustände, Codierung als 4-bit *one-hot* Vektor (z_3, z_2, z_1, z_0)
- Beispiel für die Zustandscodierung

Zustand	Codierung				Folgezustand			
	z_3	z_2	z_1	z_0	z_3^+	z_2^+	z_1^+	z_0^+
rot	0	0	0	1	0	0	1	0
rot-gelb	0	0	1	0	0	1	0	0
grün	0	1	0	0	1	0	0	0
gelb	1	0	0	0	0	0	0	1

- 4-bit statt minimal 2-bit für die Zustände
- Übergangsfunktion δ minimal: Rotate-Left um 1
  $\Rightarrow$ Automat sehr schnell, hohe Taktrate möglich
- Ausgangsfunktion λ sehr einfach:
  $gr = z_2 \quad ge = z_3 \vee z_1 \quad rt = z_1 \vee z_0$

Schaltwerksentwurf: Ampel – Variante 3 (cont.)



[HenHA] Hades Demo: `18-fsm/10-trafficlight/ampel_44`

Schaltwerksentwurf: Ampel – Zusammenfassung

- viele Möglichkeiten der Zustandscodierung

- Dualcode: minimale Anzahl der Zustände
- applikations-spezifische Codierungen
- One-Hot Encoding: viele Zustände, einfache Schaltnetze
- ...

- Kosten/Performanz des Schaltwerks abhängig von Codierung
- Heuristiken zur Suche nach (relativem) Optimum

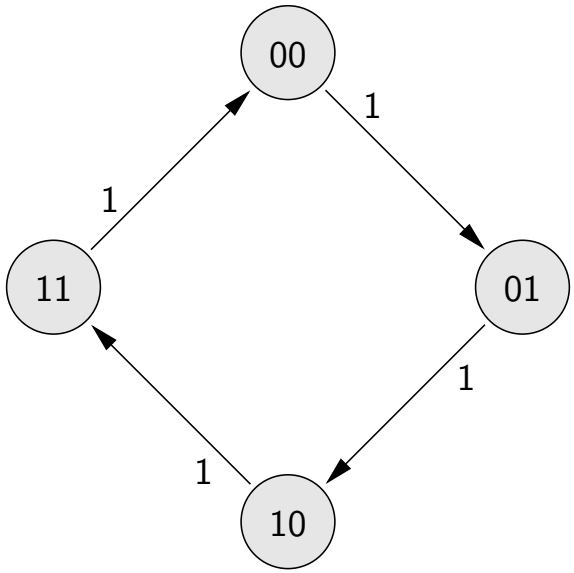
Zählschaltungen

- diverse Beispiele für Zählschaltungen

- Zustandsdiagramme und Flusstafeln
- Schaltbilder

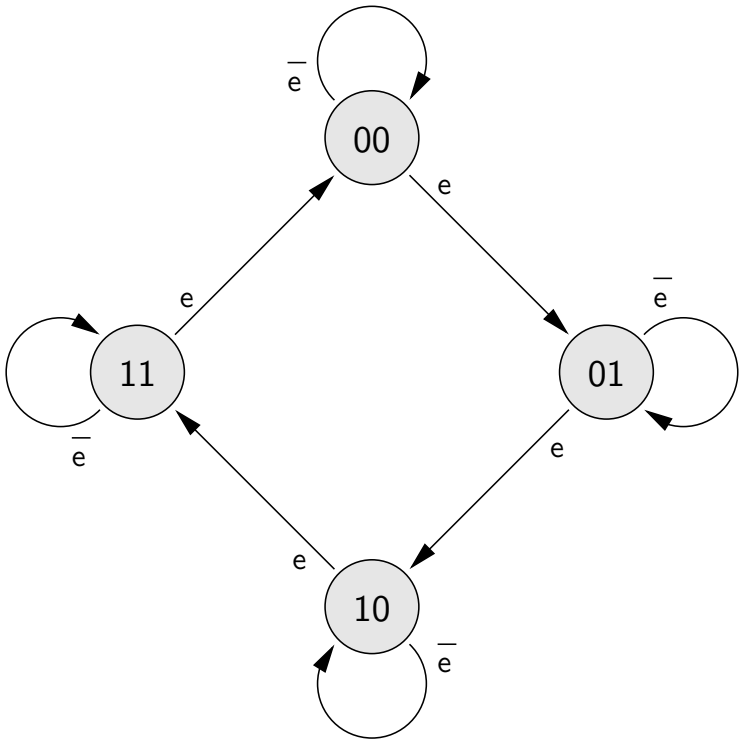
- *n*-bit Vorwärtszähler
- *n*-bit Zähler mit Stop und/oder Reset
- Vorwärts-/Rückwärtszähler
- synchrone und asynchrone Zähler
- Beispiel: Digitaluhr (BCD Zähler)

2-bit Zähler: Zustandsdiagramm



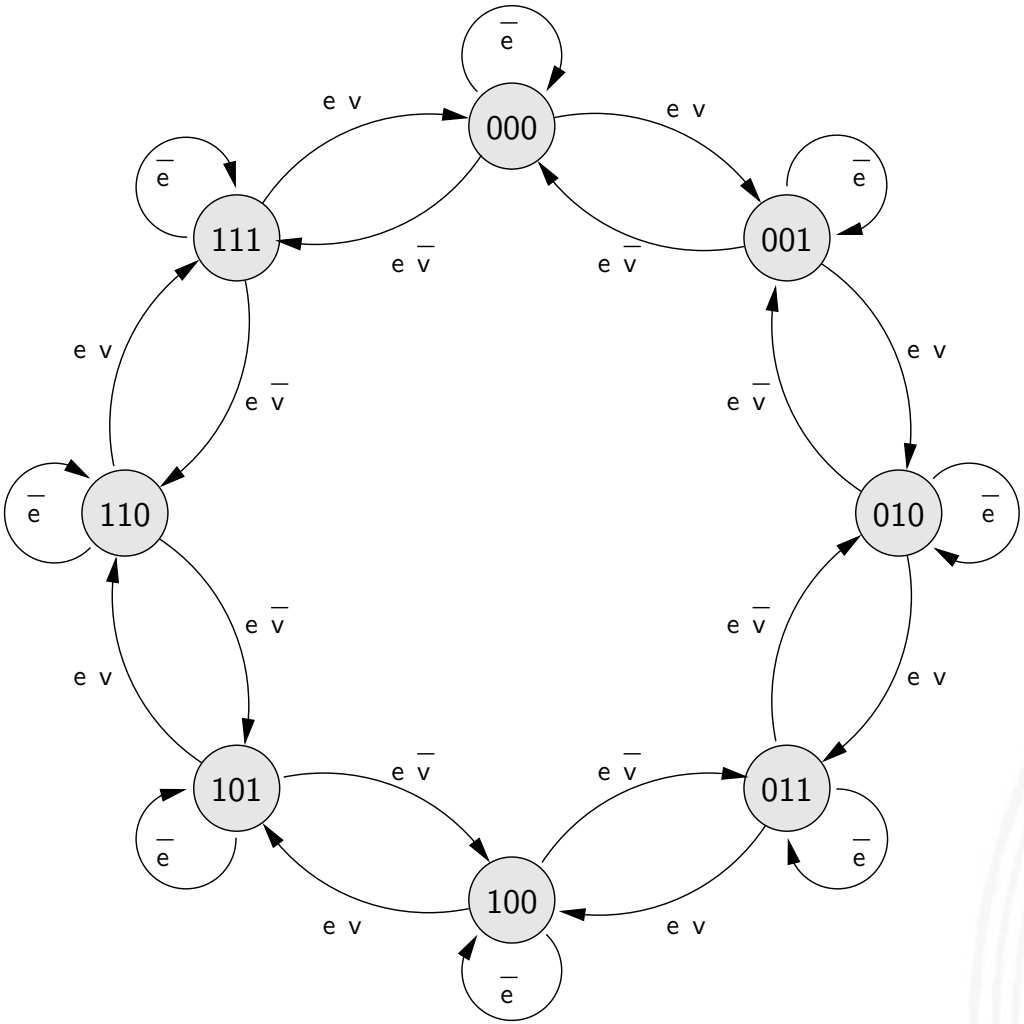
- Zähler als „trivialer“ endlicher Automat

2-bit Zähler mit Enable: Zustandsdiagramm, Flusstafel



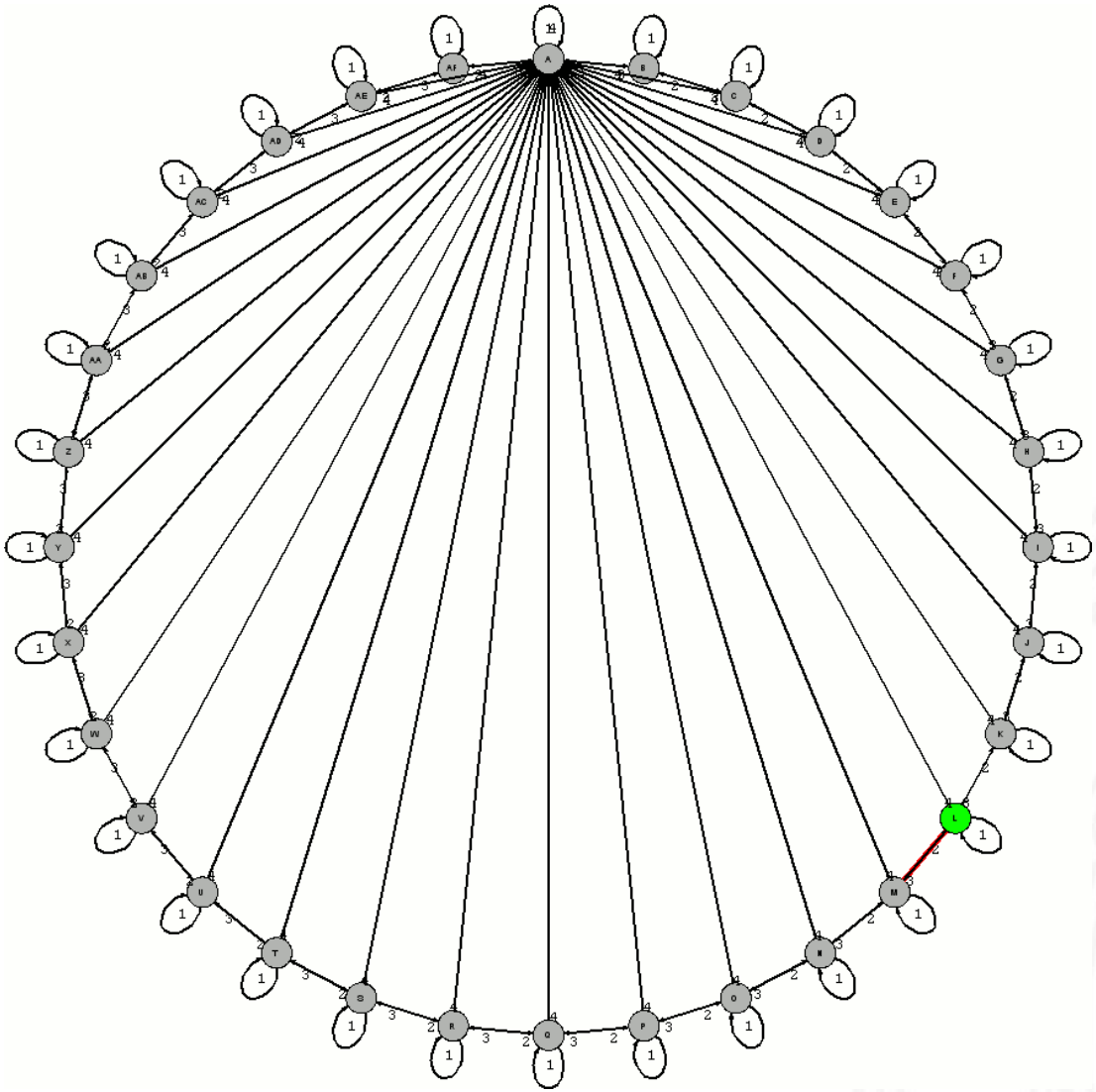
Eingabe	e	$\overline{e}$
Zustand	Folgezustand	
00	01	00
01	10	01
10	11	10
11	00	11

3-bit Zähler mit Enable, Vor-/Rückwärts



Eingabe	$e\,v$	$e\,\bar{v}$	$\bar{e}\,*$
Zustand	Folgezustand		
000	001	111	000
001	010	000	001
010	011	001	010
011	100	010	011
100	101	011	100
101	110	100	101
110	111	101	110
111	000	110	111

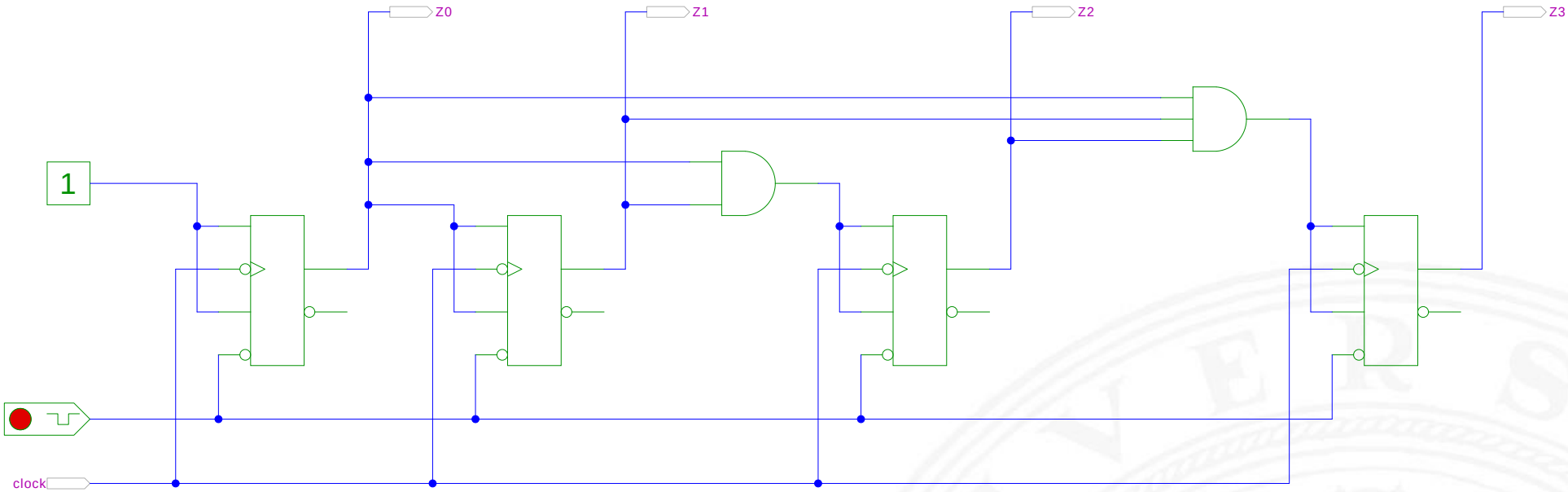
5-bit Zähler mit Reset: Zustandsdiagramm und Flusstafel



Zustand	Index der Eingabe 1	2	3	4
A	A	B	AF	A
B	B	C	A	A
C	C	D	B	A
D	D	E	C	A
E	E	F	D	A
F	F	G	E	A
G	G	H	F	A
H	H	I	G	A
I	I	J	H	A
J	J	K	I	A
K	K	L	J	A
L	L	M	K	A
M	M	N	L	A
N	N	O	M	A
O	O	P	N	A
P	P	Q	O	A
Q	Q	R	P	A
R	R	S	Q	A
S	S	T	R	A
T	T	U	S	A
U	U	V	T	A
V	V	W	U	A
W	W	X	V	A
X	X	Y	W	A
Y	Y	Z	X	A
Z	Z	AA	Y	A
AA	AA	AB	Z	A
AB	AB	AC	AA	A
AC	AC	AD	AB	A
AD	AD	AE	AC	A
AE	AE	AF	AD	A
AF	AF	A	AE	A

Eingabe 1: stop, 2: zählen, 3: rückwärts zählen, 4: Reset nach A

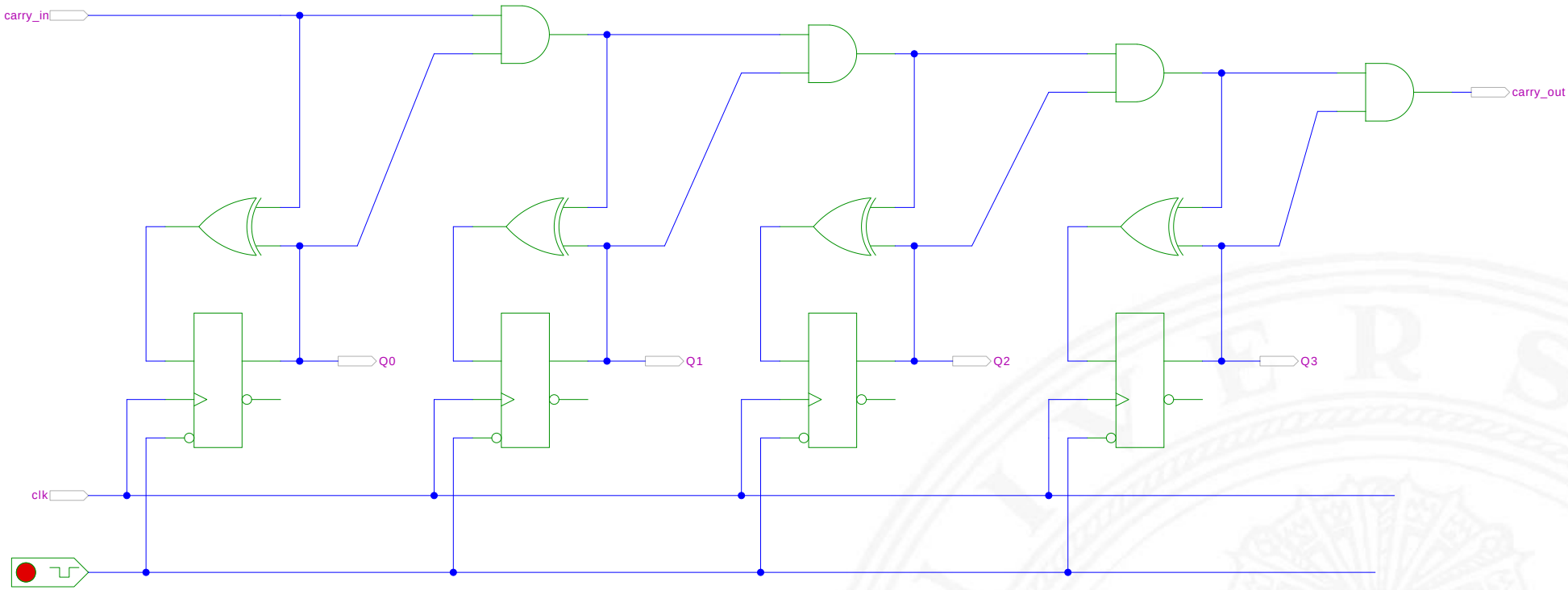
4-bit Binärzähler mit JK-Flipflops



[HenHA] Hades Demo: `30-counters/30-sync/sync`

- $J_0 = K_0 = 1$: Ausgang z_0 wechselt bei jedem Takt
- $J_i = K_i = (z_0 z_1 \ldots z_{i-1})$: Ausgang z_i wechselt, wenn alle niedrigeren Stufen 1 sind

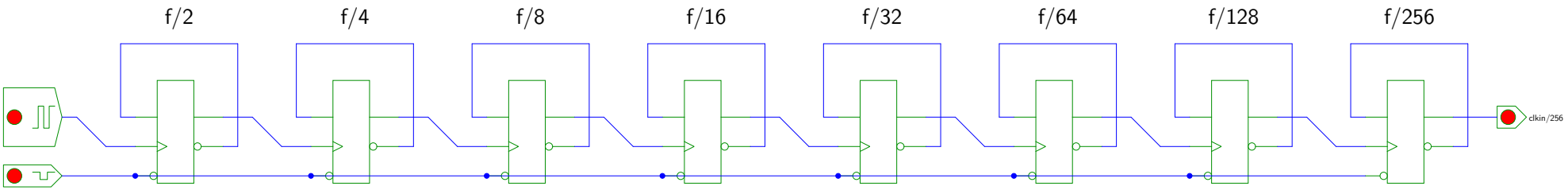
4-bit Binärzähler mit D-Flipflops (kaskadierbar)



[HenHA] Hades Demo: `30-counters/30-sync/sync-dff`

- $D_0 = Q_0 \oplus c_{in}$ wechselt bei Takt, wenn c_{in} aktiv ist
- $D_i = Q_i \oplus (c_{in} Q_0 Q_1 \ldots Q_{i-1})$ wechselt, wenn alle niedrigeren Stufen und Carry-in c_{in} 1 sind

Asynchroner n-bit Zähler/Teiler mit D-Flipflops



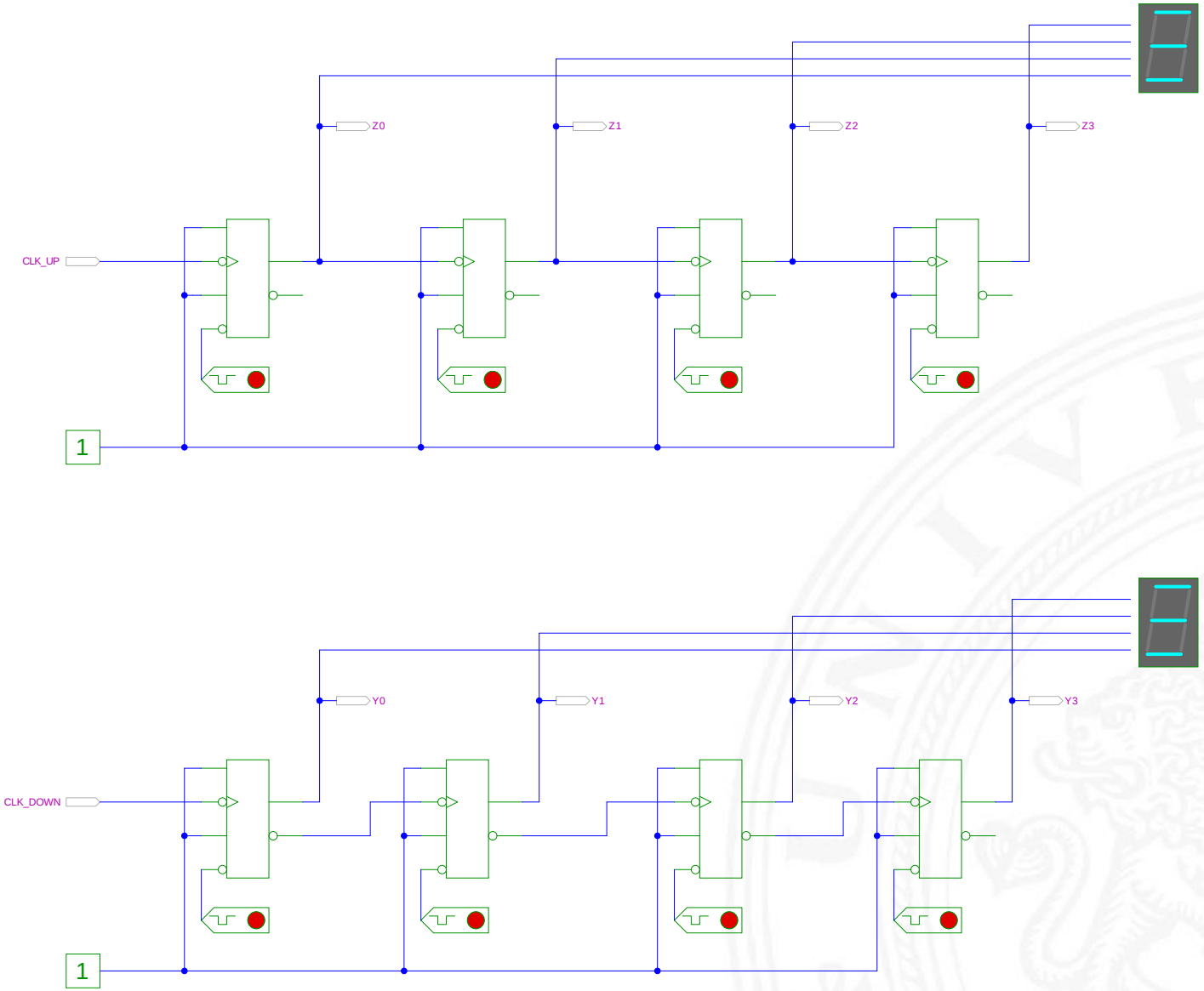
[HenHA] Hades Demo: `30-counters/20-async/counter-dff`

- $D_i = \overline{Q_i}$: jedes Flipflop wechselt bei seinem Taktimpuls
- Takteingang C_0 treibt nur das vorderste Flipflop
- $C_i = Q_{i-1}$: Ausgang der Vorgängerstufe als Takt von Stufe i

- erstes Flipflop wechselt bei jedem Takt $\Rightarrow$ Zählrate $C_0/2$
  zweites Flipflop bei jedem zweiten Takt $\Rightarrow$ Zählrate $C_0/4$
  n-tes Flipflop bei jedem n-ten Takt $\Rightarrow$ Zählrate $C_0/2^n$
- sehr hohe maximale Taktrate

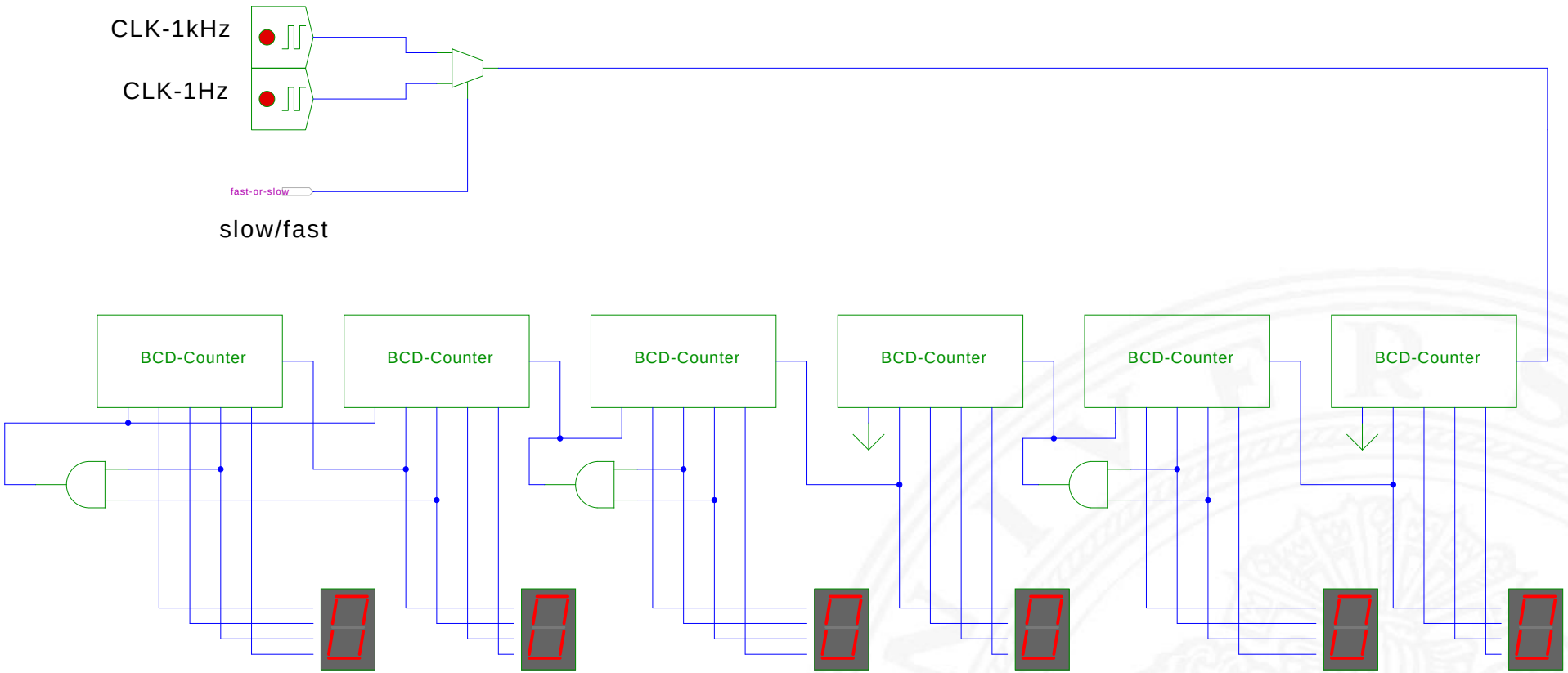
– Achtung: Flipflops schalten nacheinander, nicht gleichzeitig

Asynchrone 4-bit Vorwärts- und Rückwärtszähler



[HenHA] Hades Demo: `30-counters/20-async/counter`

Digitaluhr mit BCD Zählern



[HenHA] Hades Demo: `30-counters/80-digiclock/digiclock`

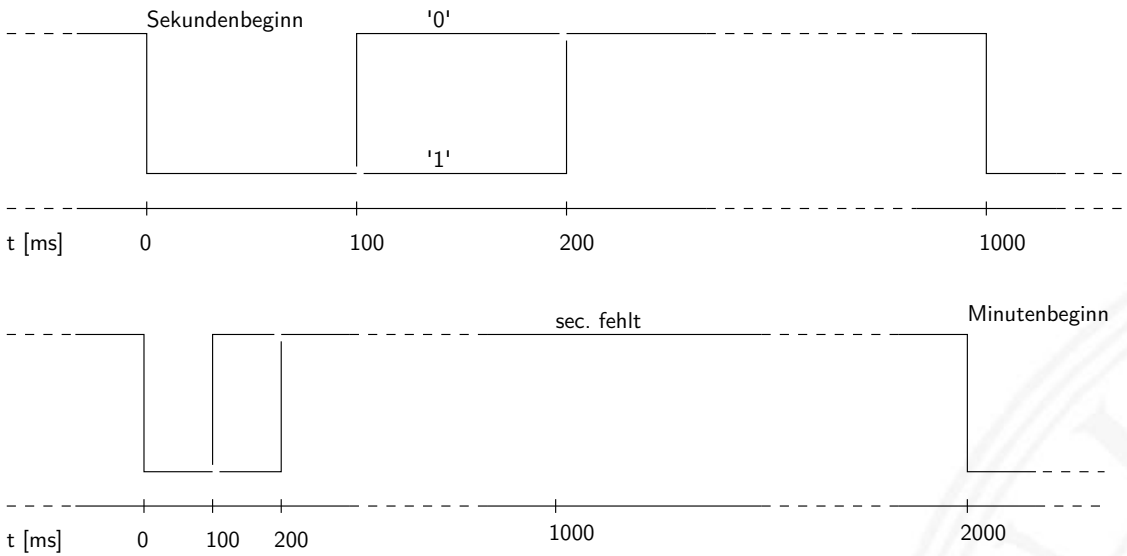
- Stunden Minuten Sekunden (hh:mm:ss)
- async. BCD Zähler mit Takt (rechts) und Reset (links unten)
- Übertrag 1er- auf 10er-Stelle jeweils beim Übergang $9 \rightarrow 0$
- Übertrag und Reset der Zehner beim Auftreten des Wertes 6

Funkgesteuerte DCF 77 Uhr

- Beispiel für komplexe Schaltung
  - mehrere einfache Komponenten
  - gekoppelte Automaten, Zähler etc.
- DCF 77 Zeitsignal
  - Langwelle 77,5 KHz
  - Sender nahe Frankfurt
  - ganz Deutschland abgedeckt
- pro Sekunde wird ein Bit übertragen
  - Puls mit abgesenktem Signalpegel: „Amplitudenmodulation“
  - Pulslänge: 100 ms entspricht Null, 200 ms entspricht Eins
  - Pulsbeginn ist Sekundenbeginn
- pro Minute werden 59 Bits übertragen
  - Uhrzeit hh:mm (implizit Sekunden), MEZ/MESZ
  - Datum dd:mm:yy, Wochentag
  - Parität
  - fehlender 60ster Puls markiert Ende einer Minute

Funkgesteuerte DCF 77 Uhr (cont.)

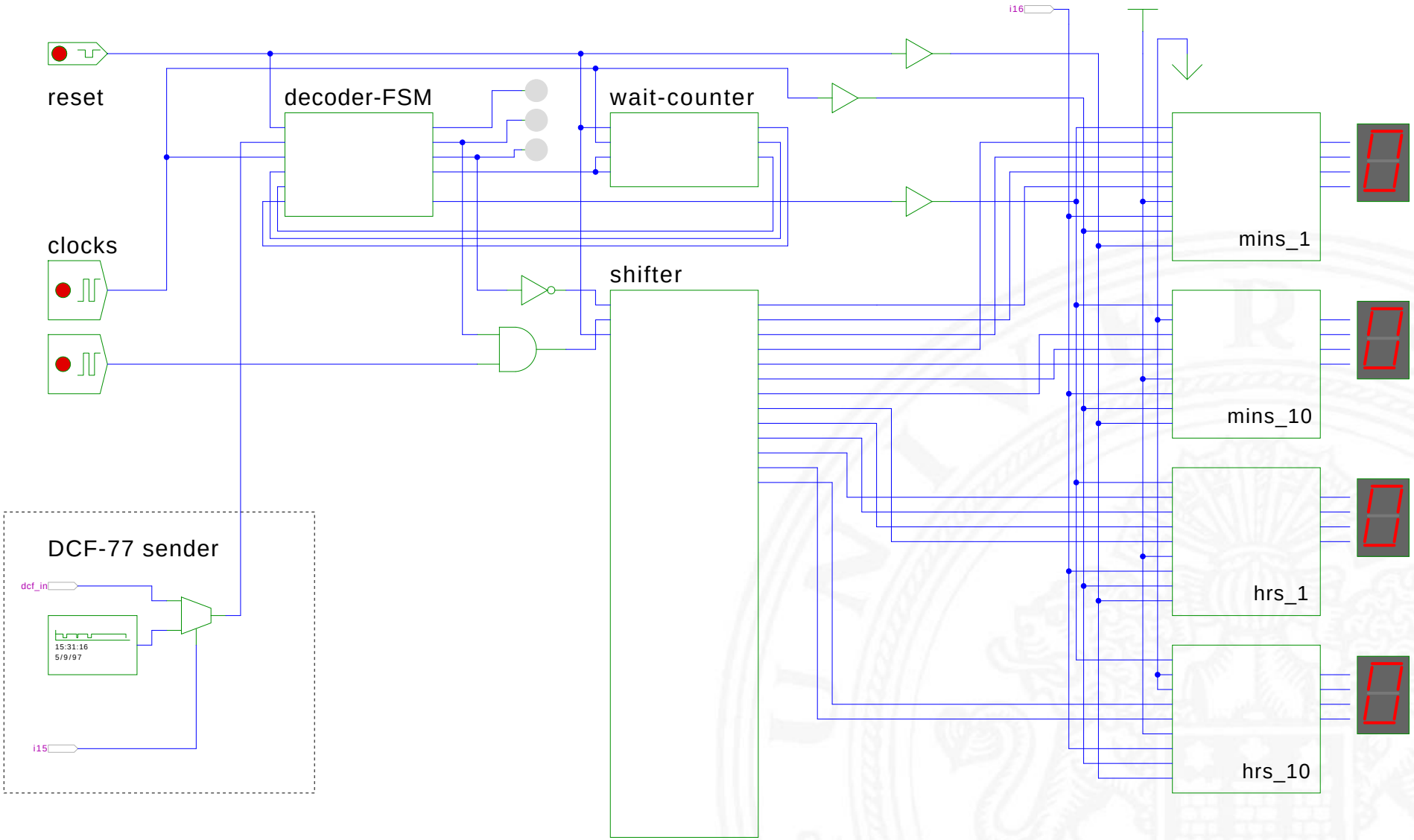
- Decodierung der Bits aus DCF 77 Protokoll mit entsprechend entworfenem Schaltwerk



- siehe z.B.: `de.wikipedia.org/wiki/DCF77`

Sek.	Bit	Wert	Bedeutung	Beispiel
0	0		Minutenbeginn	0
1	*		nicht belegt, i.A. wird 0 gesendet	0
⋮				
14	*		– s.o. –	0
15	0 / 1		1: Reserveantenne an	0
16	0 / 1		1: Stundensprung folgt (z.B. Sommerzeit)	0
17	0 / 1	2	Zeitzonenbits: geben die Abweichung von der	1
18	0 / 1	1	Weltzeit in Stunden an (MEZ = 1, MESZ = 2)	0
19	0 / 1		1: Schaltsekunde folgt	0
20	1		Start der Zeitcodierung	1
21	0 / 1	1	Minuten – Stelle 1	1
22	0 / 1	2	– s.o. –	0
23	0 / 1	4	– s.o. –	1
24	0 / 1	8	– s.o. –	0 = 5
25	0 / 1	10	Minuten – Stelle 10	1
26	0 / 1	20	– s.o. –	1
27	0 / 1	40	– s.o. –	0 = 3
28	0 / 1		Prüfbit zu 21 . . . 27 (even)	0
29	0 / 1	1	Stunden – Stelle 1	1
30	0 / 1	2	– s.o. –	0
31	0 / 1	4	– s.o. –	0
32	0 / 1	8	– s.o. –	1 = 9
33	0 / 1	10	Stunden – Stelle 10	1
34	0 / 1	20	– s.o. –	0 = 1
35	0 / 1		Prüfbit zu 29 . . . 34 (even)	1
36	0 / 1	1	Kalendertag – Stelle 1	0
37	0 / 1	2	– s.o. –	1
38	0 / 1	4	– s.o. –	0
39	0 / 1	8	– s.o. –	0 = 2
40	0 / 1	10	Kalendertag – Stelle 10	0
41	0 / 1	20	– s.o. –	0 = 0
42	0 / 1	1	Wochentag	1
43	0 / 1	2	– s.o. –	0
44	0 / 1	4	– s.o. –	0 = 1 (Mo)
45	0 / 1	1	Kalendermonat – Stelle 1	0
46	0 / 1	2	– s.o. –	1
47	0 / 1	4	– s.o. –	0
48	0 / 1	8	– s.o. –	0 = 2
49	0 / 1	10	Kalendermonat – Stelle 10	1 = 1
50	0 / 1	1	Kalenderjahr – Stelle 1	1
51	0 / 1	2	– s.o. –	0
52	0 / 1	4	– s.o. –	0
53	0 / 1	8	– s.o. –	1 = 9
54	0 / 1	10	Kalenderjahr – Stelle 10	1
55	0 / 1	20	– s.o. –	0
56	0 / 1	40	– s.o. –	0
57	0 / 1	80	– s.o. –	0 = 1
58	0 / 1		Prüfbit zu 36 . . . 57 (even)	1
59	–		Marke fehlt, weder 0 noch 1 wird gesendet	
	Beispiel:		Mo. 02.12.19: 19.35	

Funkgesteuerte DCF 77 Uhr: Gesamtsystem



[HenHA] Hades Demo: `45-misc/80-dcf77/dcf77`

Multiplex-Siebensegment-Anzeige

Ansteuerung mehrstelliger Siebensegment-Anzeigen?

- direkte Ansteuerung erfordert $7 \cdot n$ Leitungen für n Ziffern
- und je einen Siebensegment-Decoder pro Ziffer

Zeit-Multiplex-Verfahren benötigt nur $7 + n$ Leitungen

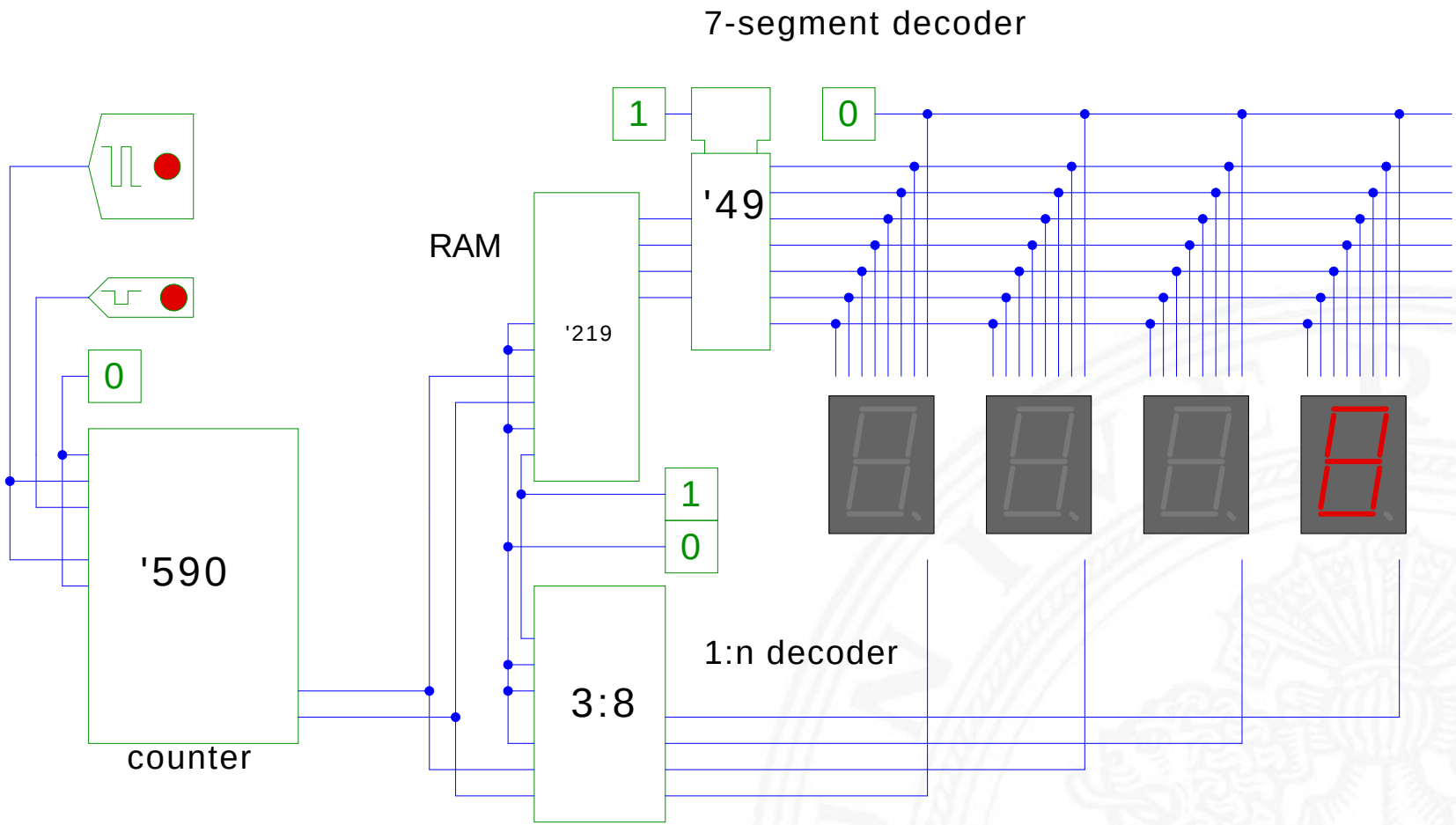
- die Anzeigen werden nacheinander nur ganz kurz eingeschaltet
- ein gemeinsamer Siebensegment-Decoder
  Eingabe wird entsprechend der aktiven Ziffer umgeschaltet
- das Auge sieht die leuchtenden Segmente und „mittelt“
- ab ca. 100 Hz Frequenz erscheint die Anzeige ruhig

Multiplex-Siebensegment-Anzeige (cont.)

Hades-Beispiel: Kombination mehrerer bekannter einzelner Schaltungen zu einem komplexen Gesamtsystem

- vierstellige Anzeige
- darzustellende Werte sind im RAM (74219) gespeichert
- Zähler-IC (74590) erzeugt 2-bit Folge $\{00, 01, 10, 11\}$
- 3:8-Decoder-IC (74138) erzeugt daraus die Folge $\{1110, 1101, 1011, 0111\}$ um nacheinander je eine Anzeige zu aktivieren (low-active)
- Siebensegment-Decoder-IC (7449) treibt die sieben Segmentleitungen

Multiplex-Siebensegment-Anzeige (cont.)



[HenHA] Hades Demo: `45-misc/50-displays/multiplexed-display`

Literatur

[SS04] W. Schiffmann, R. Schmitz: *Technische Informatik 1 – Grundlagen der digitalen Elektronik*.
5. Auflage, Springer-Verlag, 2004. ISBN 978–3–540–40418–7

[Rei98] N. Reifschneider: *CAE-gestützte IC-Entwurfsmethoden*.
Prentice Hall, 1998. ISBN 3–8272–9550–5

[WE94] N.H.E. Weste, K. Eshraghian:
Principles of CMOS VLSI design – A systems perspective.
2nd edition, Addison-Wesley, 1994. ISBN 0–201–53376–6

[Har87] D. Harel: *Statecharts: A visual formalism for complex systems*. in: *Sci. Comput. Program.* 8 (1987), Juni, Nr. 3,
S. 231–274. ISSN 0167–6423

Interaktives Lehrmaterial

[HenHA] N. Hendrich: *HADES — HAmburg DEsign System*. Universität Hamburg, FB Informatik, Lehrmaterial.
`tams.informatik.uni-hamburg.de/applets/hades/webdemos`

[Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript*. Universität Hamburg, FB Informatik, 2005.
`tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1`

Gliederung

Gliederung (cont.)

Was ist Rechnerarchitektur?

Definitionen

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

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

Was ist Rechnerarchitektur? (cont.)

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

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

Rechnerarchitektur: zwei Aspekte

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

   → folgt ab Kapitel *12 Instruction Set Architecture*

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

Wdh. von-Neumann Konzept

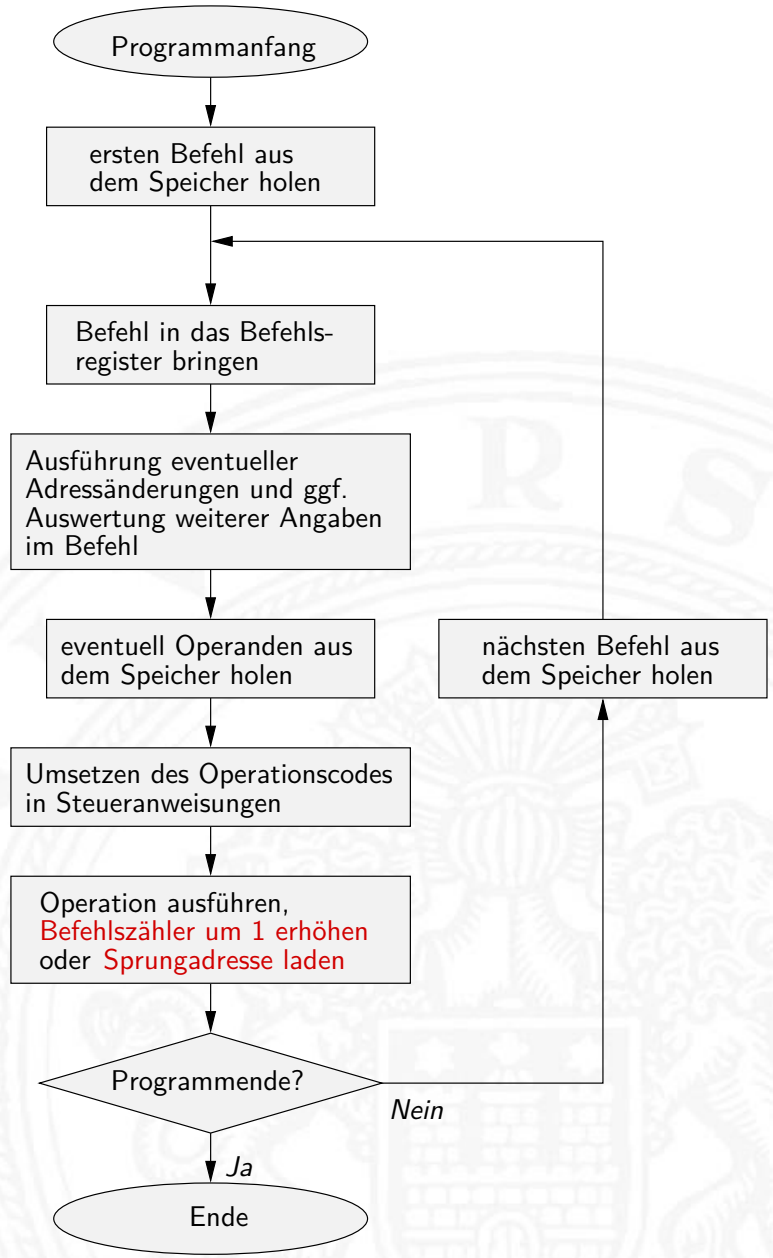
- J. Mauchly, J.P. Eckert, J. von-Neumann 1945
- Abstrakte Maschine mit minimalem Hardwareaufwand
  - System mit Prozessor, Speicher, Peripheriegeräten
  - die Struktur ist unabhängig von dem Problem, das Problem wird durch austauschbaren Speicherinhalt (Programm) beschrieben
- gemeinsamer Speicher für Programme und Daten
  - fortlaufend adressiert
  - Programme können wie Daten manipuliert werden
  - Daten können als Programm ausgeführt werden
- Befehlszyklus: Befehl holen, decodieren, ausführen

$\Rightarrow$ enorm flexibel

- **alle** aktuellen Rechner basieren auf diesem Prinzip
- aber vielfältige Architekturvarianten, Befehlssätze usw.

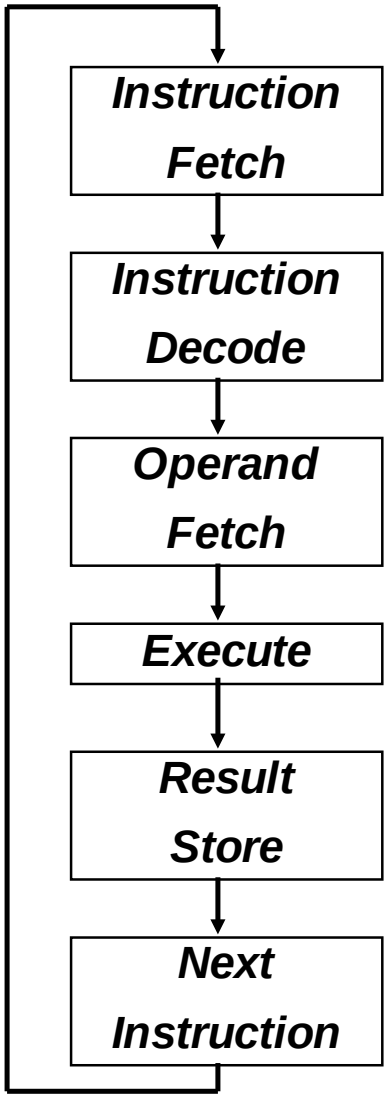
Wdh. Programmverarbeitung

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



Wdh. Programmverarbeitung (cont.)

- Ausführungszyklus



Befehl aus Programmspeicher holen

auszuführende Aktionen und Länge der Instruktion bestimmen, ggf. Worte nachladen

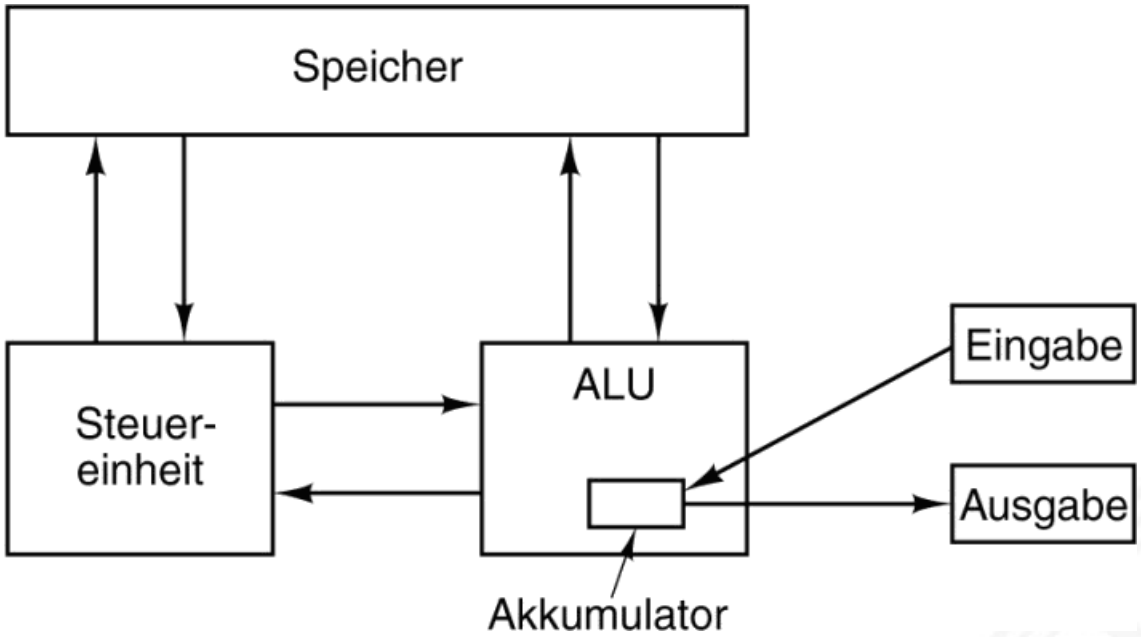
Operanden ermitteln und laden

Ergebnis der Operation berechnen bzw. Status ermitteln

Ergebnisse für später abspeichern

Folgeoperation ermitteln

Wdh. von-Neumann Rechner



[TA14]

Fünf zentrale Komponenten:

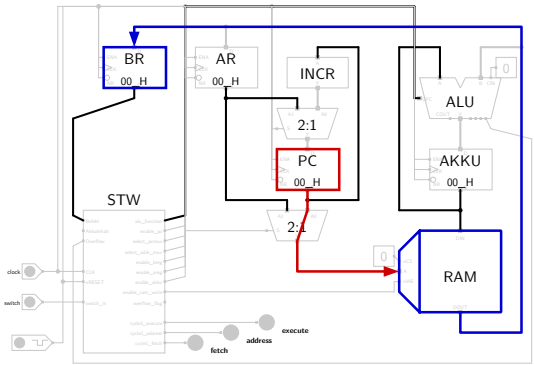
- Prozessor mit **Steuerwerk** und **Rechenwerk** (ALU, Register)
- **Speicher**, gemeinsam genutzt für Programme und Daten
- **Eingabe-** und **Ausgabewerke**
- verbunden durch Bussystem

Wdh. von-Neumann Rechner (cont.)

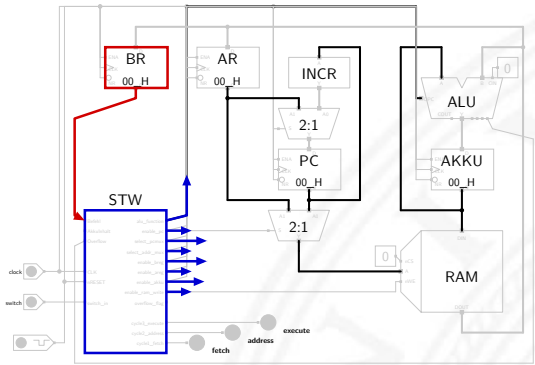
- Prozessor (CPU) = Steuerwerk + Operationswerk
- Steuerwerk: zwei zentrale Register
  - Befehlszähler (*program counter PC*)
  - Befehlsregister (*instruction register IR*)
- Operationswerk (Datenpfad, *data-path*)
  - Rechenwerk (*arithmetic-logic unit ALU*)
  - Universalregister (mind. 1 *Akkumulator*,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

- Verschaltung der Hardwarekomponenten für alle mögl. Datentransfers
- abhängig vom Befehl werden nur bestimmte Pfade aktiv
- Ausführungszyklus

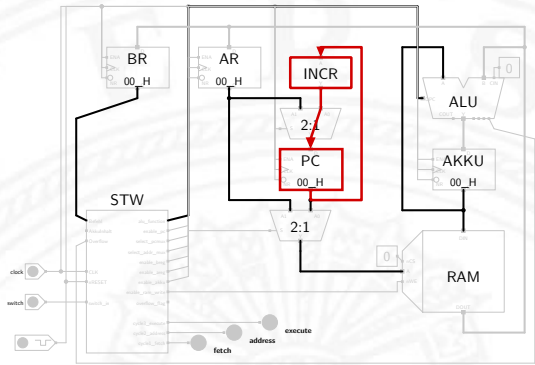
Befehl holen

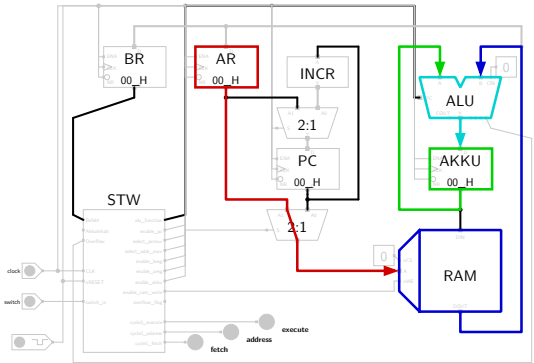


decodieren

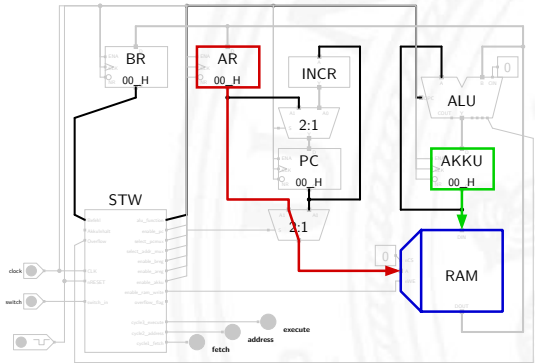


PC inkrementieren

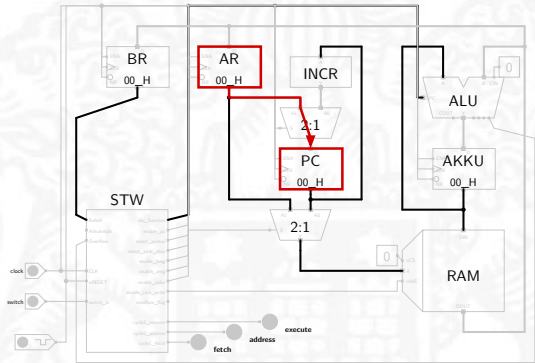




rechnen



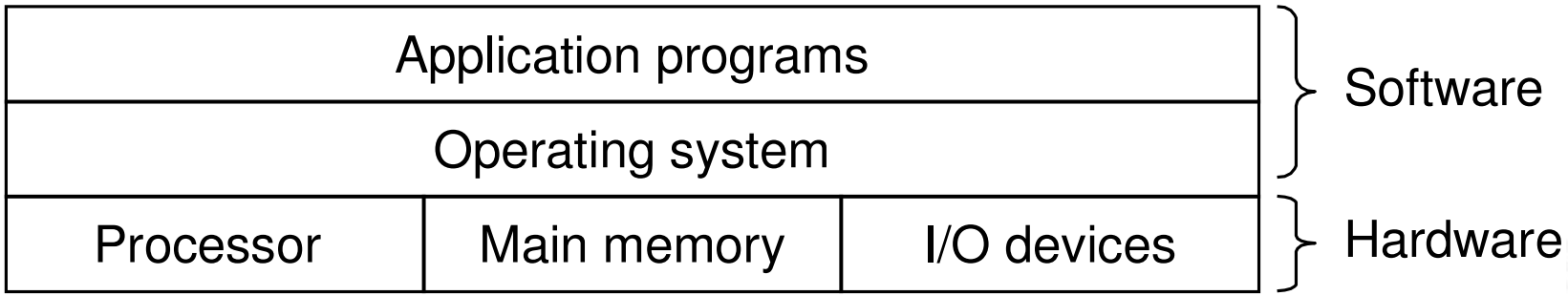
speichern



springen

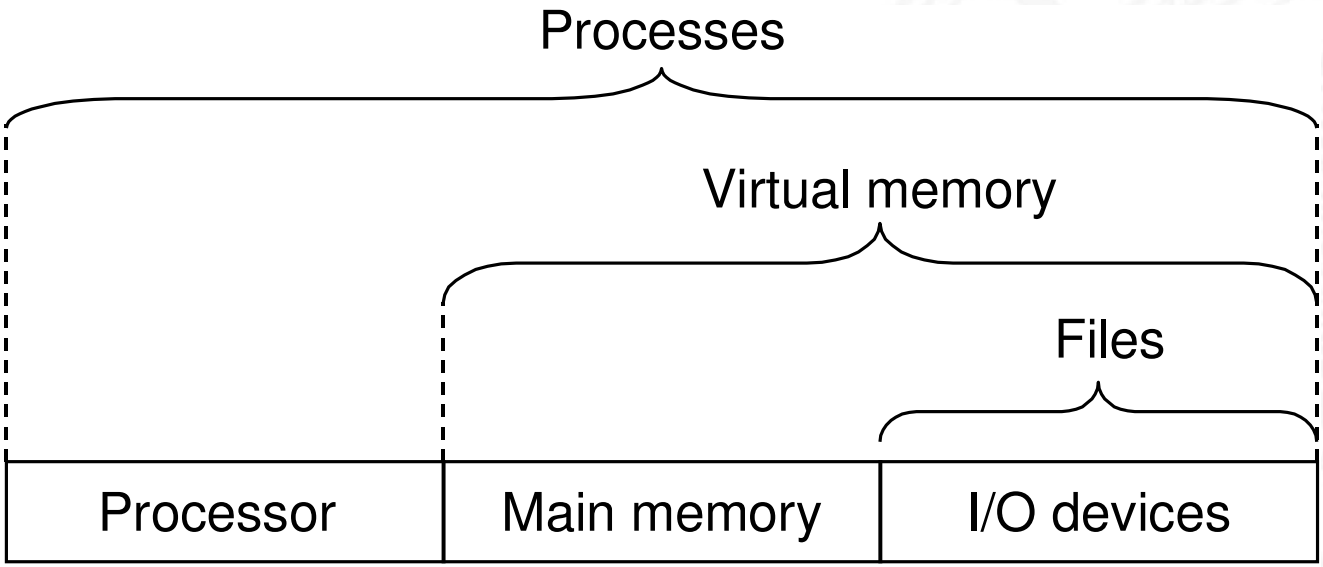
Beschreibungsebenen

- Schichten-Ansicht: Software – Hardware



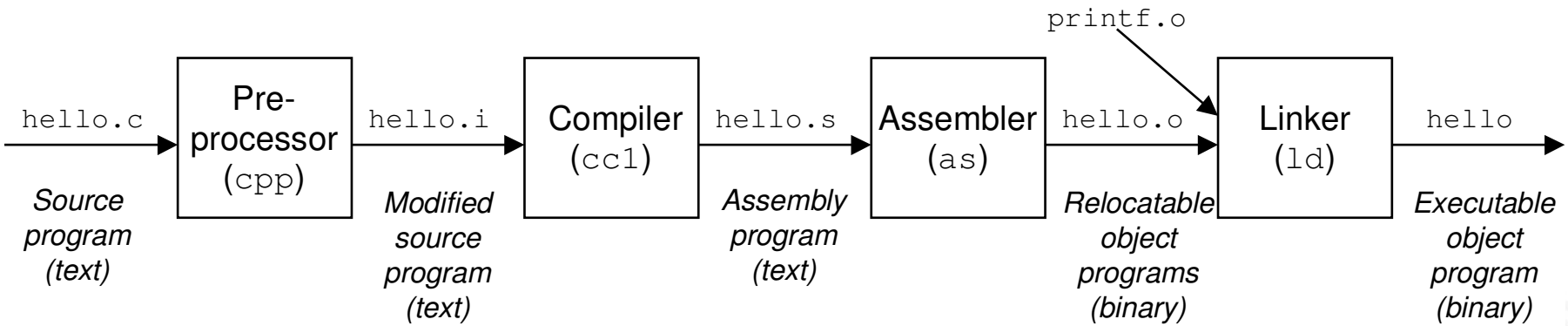
[BO15]

- Abstraktionen durch Betriebssystem



[BO15]

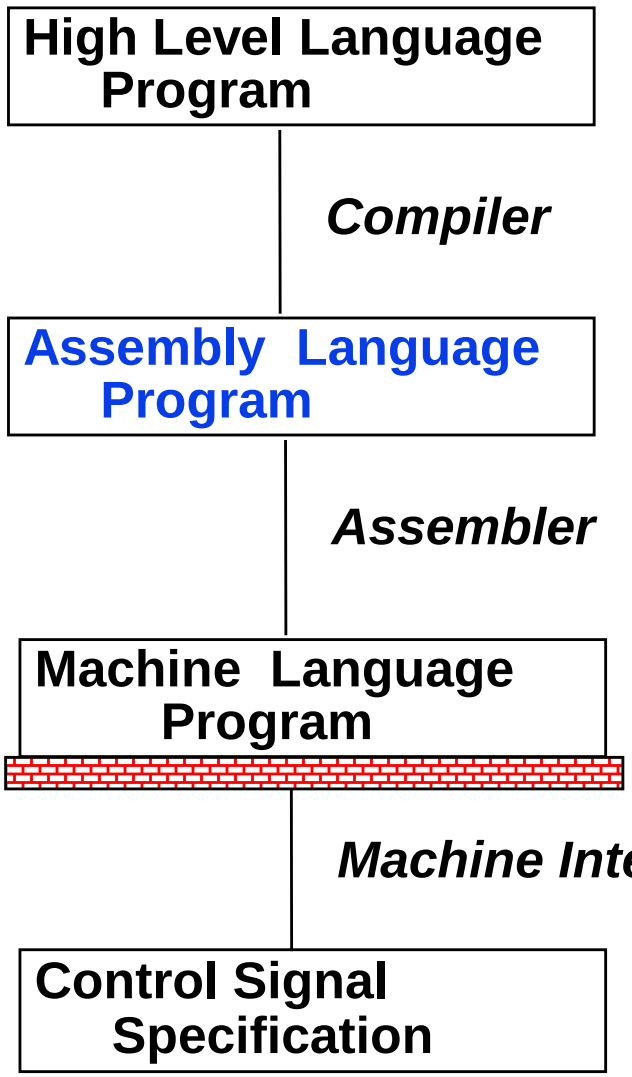
Das Compilierungssystem



[BO15]

- verschiedene Repräsentationen des Programms
  - Hochsprache
  - Assembler
  - Maschinensprache
- Ausführung der Maschinensprache
  - von-Neumann Zyklus: Befehl holen, decodieren, ausführen
  - reale oder virtuelle Maschine

Das Compilierungssystem (cont.)




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $15,   0($2)
lw  $16,   4($2)
sw $16,   0($2)
sw $15,   4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

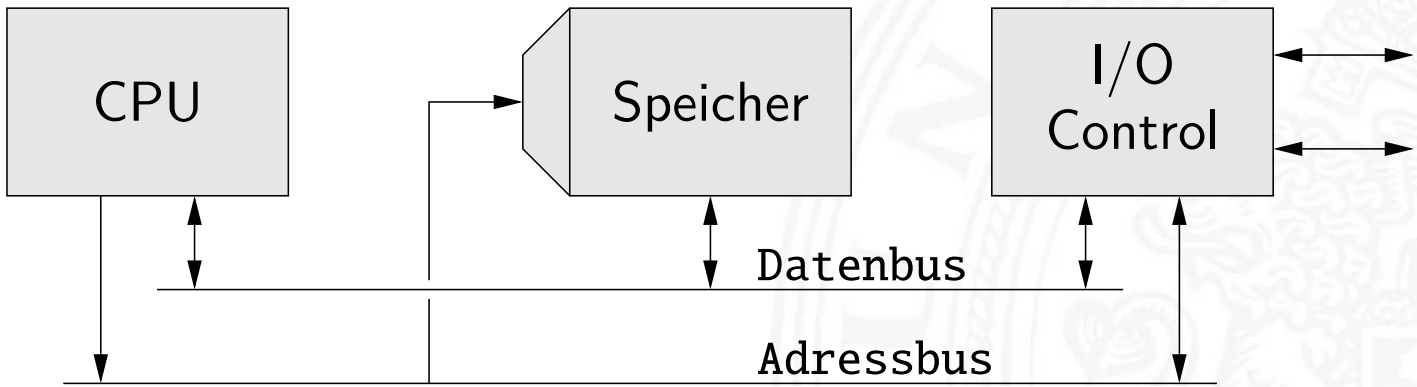
ALUOP[0:3] <= InstReg[9:11] & MASK

[PH16b]

Abstraktion im VLSI-Entwurf

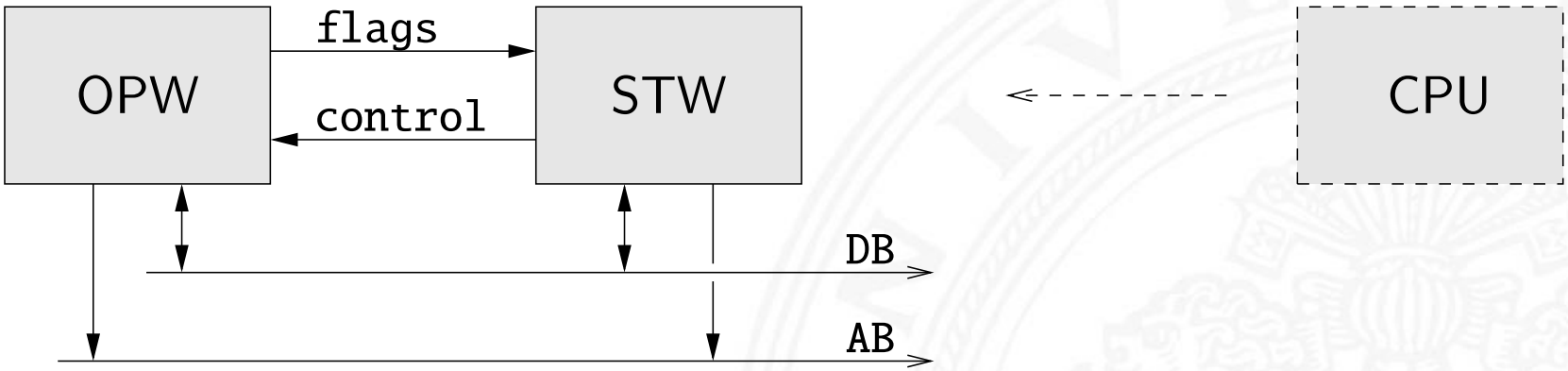
Hardware Abstraktionsebenen

- keine einheitliche Bezeichnung in der Literatur
- Architekturebene
  - Funktion/Verhalten Leistungsanforderungen
  - Struktur Netzwerk
    aus Prozessoren, Speicher, Busse, Controller ...
  - Nachrichten Programme, Prokolle
  - Geometrie Systempartitionierung



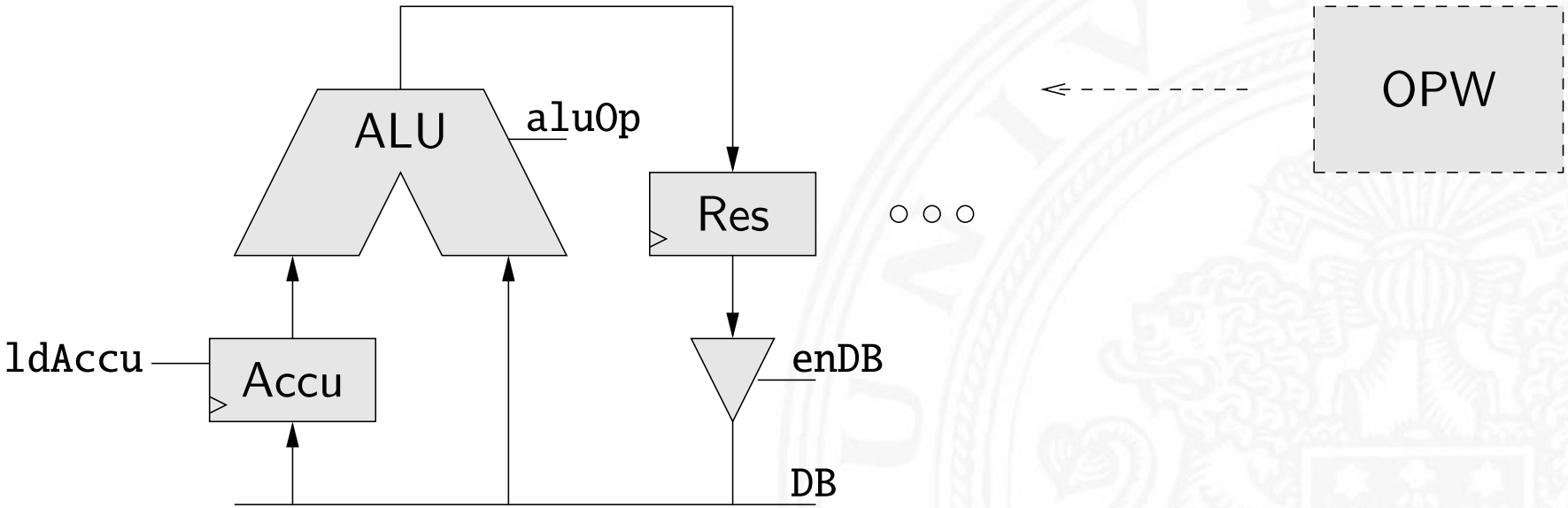
Abstraktion im VLSI-Entwurf (cont.)

- Hauptblockebene (Algorithmenebene, funktionale Ebene)
  - Funktion/Verhalten Algorithmen, formale Funktionsmodelle
  - Struktur Blockschaltbild
    aus Hardwaremodule, Busse ...
  - Nachrichten Prokolle
  - Geometrie Cluster



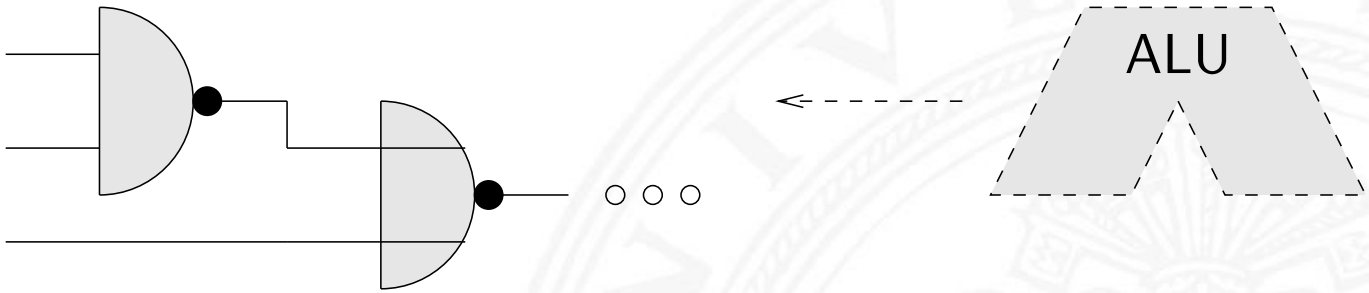
Abstraktion im VLSI-Entwurf (cont.)

- Register-Transfer Ebene
  - Funktion/Verhalten Daten- und Kontrollfluss, Automaten ...
  - Struktur RT-Diagramm
    aus Register, Multiplexer, ALUs ...
  - Nachrichten Zahlencodierungen, Binärworte ...
  - Geometrie Floorplan



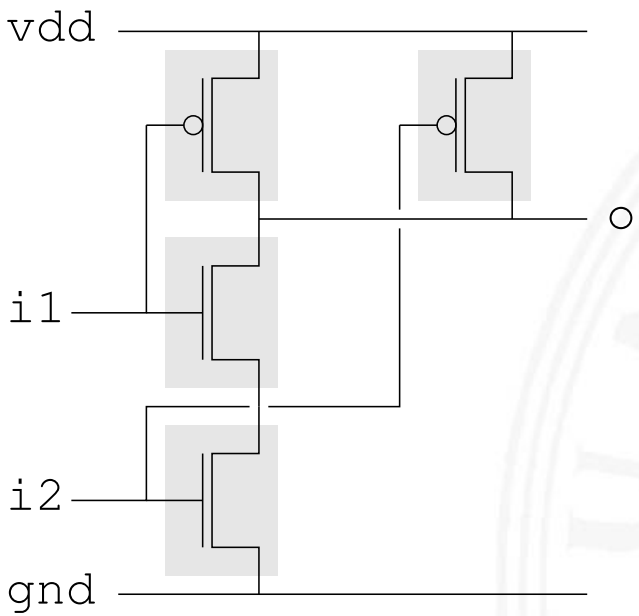
Abstraktion im VLSI-Entwurf (cont.)

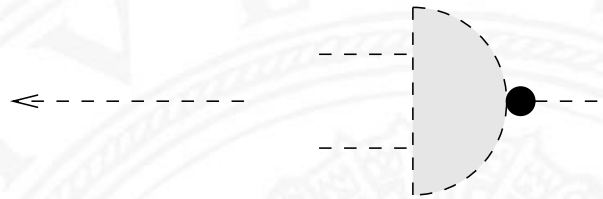
- Logikebene (Schaltwerkebene)
  - Funktion/Verhalten Boole'sche Gleichungen
  - Struktur Gatternetzliste, Schematic
    aus Gatter, Flipflops, Latches ...
  - Nachrichten Bit
  - Geometrie Moduln



Abstraktion im VLSI-Entwurf (cont.)

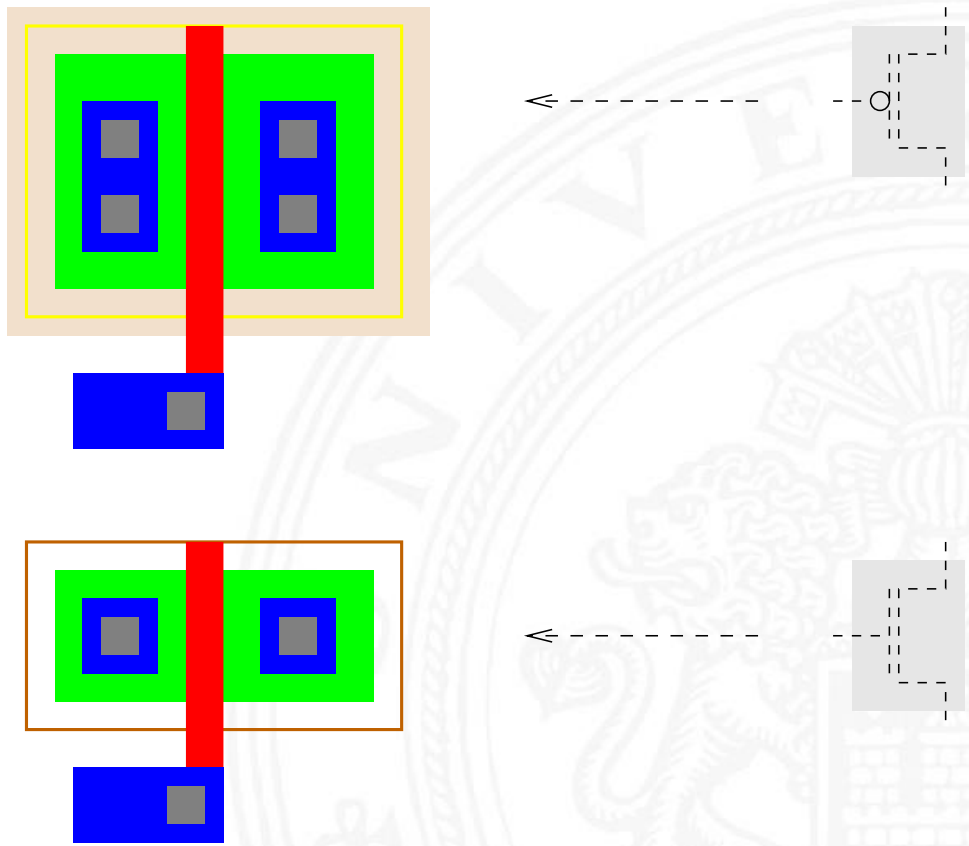
- elektrische Ebene (Schaltkreisebene)
  - Funktion/Verhalten Differentialgleichungen
  - Struktur elektrisches Schaltbild
    aus Transistoren, Kondensatoren ...
  - Nachrichten Ströme, Spannungen
  - Geometrie Polygone, Layout → physikalische Ebene



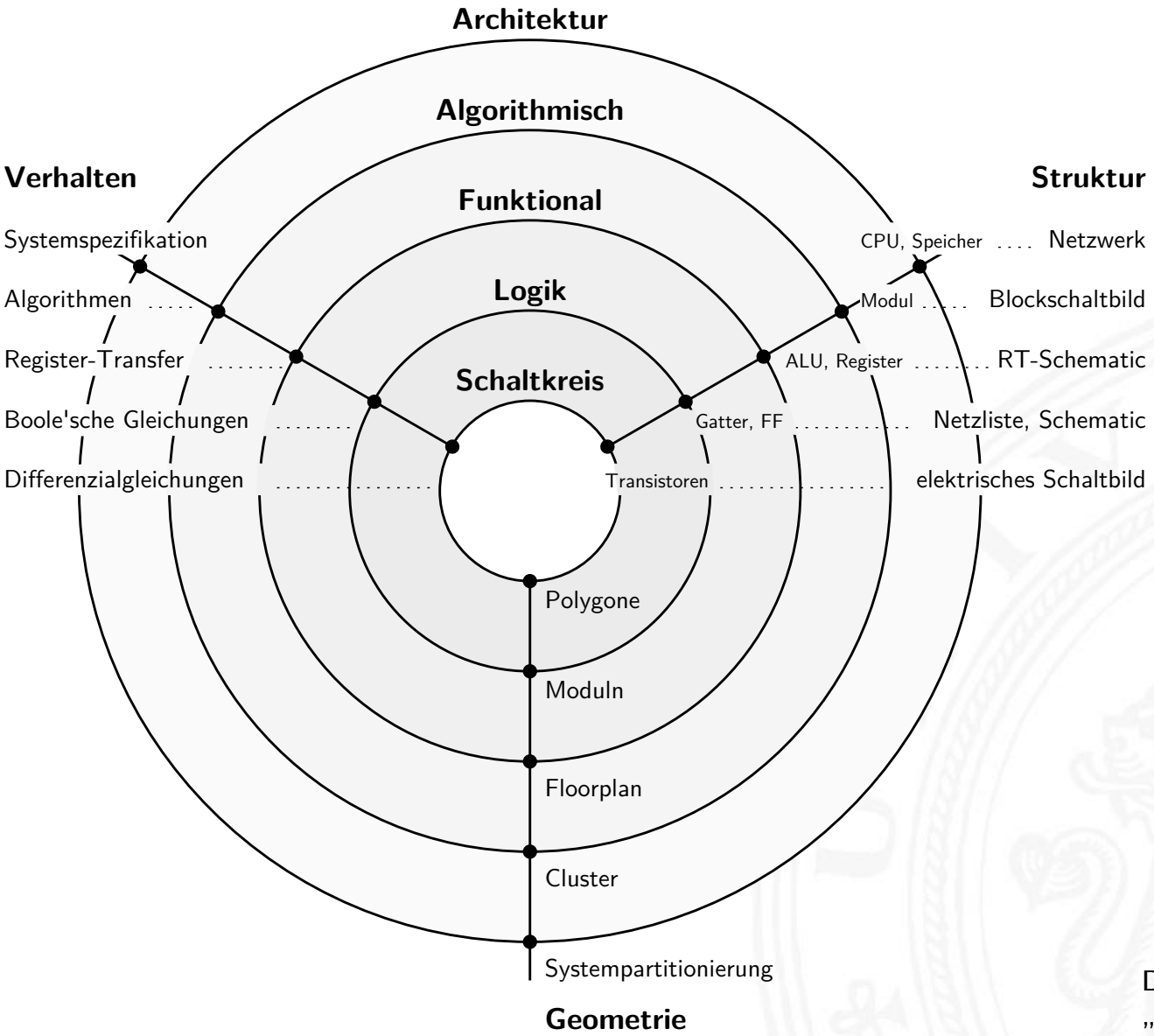


Abstraktion im VLSI-Entwurf (cont.)

- physikalische Ebene (geometrische Ebene)
  - Funktion/Verhalten partielle DGL
  - Struktur Dotierungsprofile



Y-Diagramm



D. Gajski, R. Kuhn 1983:
„New VLSI Tools“ [GK83]

Y-Diagramm (cont.)

drei unterschiedliche Aspekte/Dimensionen:

1. Verhalten
2. Struktur (logisch)
3. Geometrie (physikalisch)

- Start möglichst abstrakt, z.B. als Verhaltensbeschreibung
- Ende des Entwurfsprozesses ist vollständige IC Geometrie für die Halbleiterfertigung (Planarprozess)

- Entwurfsprogramme („EDA“, *Electronic Design Automation*) unterstützen den Entwerfer: setzen Verhalten in Struktur und Struktur in Geometrien um

Systemmodellierung

Modellierung eines digitalen Systems als Schaltung aus

- speichernden Komponenten
  - Registern — Flipflops, Register, Registerbank
  - Speichern — SRAM, DRAM, ROM, PLA
- funktionalen Schaltnetzen
  - Addierer, arithmetische Schaltungen
  - logische Operationen
  - „random-logic“ Schaltnetzen
- Verbindungsleitungen
  - Busse / Leitungsbündel
  - Multiplexer und Tri-state Treiber

Rechnerarchitektur: Hardwarestruktur

- bisher:
  - Gatter und Schaltnetze
  - Flipflops als einzelne Speicherglieder
  - Schaltwerke zur Ablaufsteuerung

- weitere Komponenten: Register-Transfer- und Hauptblockebene
  - Speicher
  - Busse, Bustiming
  - Mikroprogrammierung zur Ablaufsteuerung

Speicher

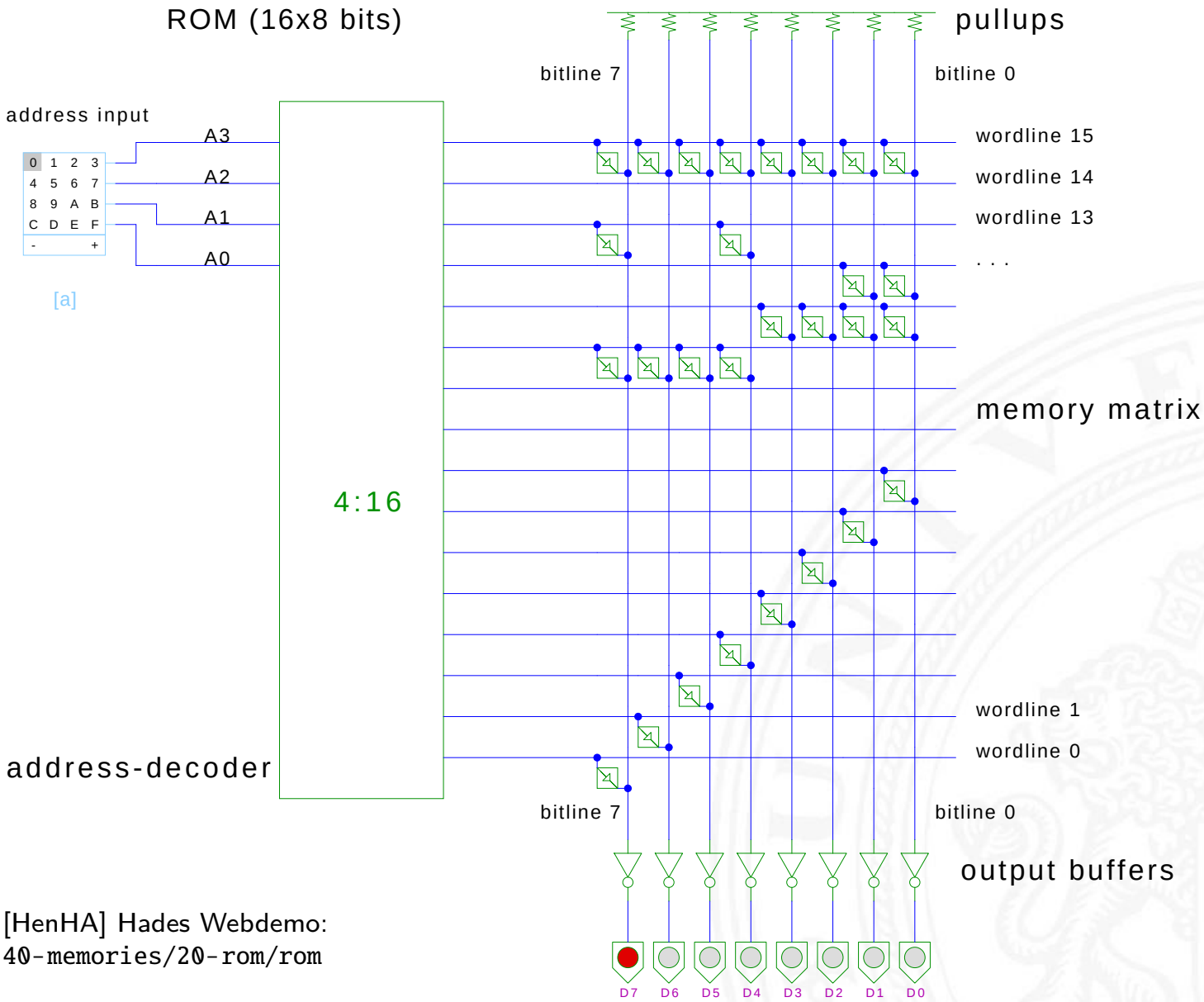
- System zur Speicherung von Information
- als Feld von N Adressen mit je m-bit Speicherworten
- typischerweise mit n-bit Adressen und $N = 2^n$
- Kapazität also $2^n \cdot m$ Bits

- Klassifikation
  - Speicherkapazität?
  - Schreibzugriffe möglich?
  - Schreibzugriffe auf einzelne Bits/Bytes oder nur Blöcke?
  - Information flüchtig oder dauerhaft gespeichert?
  - Zugriffszeiten beim Lesen und Schreiben
  - Technologie

Speicherbausteine: Varianten

Typ	Kategorie	Löschen	byte-adressierbar	flüchtig	Typische Anwendung
SRAM	Lesen/Schreiben	elektrisch	ja	ja	Level-2 Cache
DRAM	Lesen/Schreiben	elektrisch	ja	ja	Hauptspeicher (alt)
SDRAM	Lesen/Schreiben	elektrisch	ja	ja	Hauptspeicher
ROM	nur Lesen	—	nein	nein	Geräte in großen Stückzahlen
PROM	nur Lesen	—	nein	nein	Geräte in kleinen Stückzahlen
EPROM	vorw. Lesen	UV-Licht	nein	nein	Prototypen
EEPROM	vorw. Lesen	elektrisch	ja	nein	Prototypen
Flash	Lesen/Schreiben	elektrisch	nein	nein	Speicherkarten, Mobile Geräte, SSDs

ROM: Read-Only Memory



[HenHA] Hades Webdemo:
`40-memories/20-rom/rom`

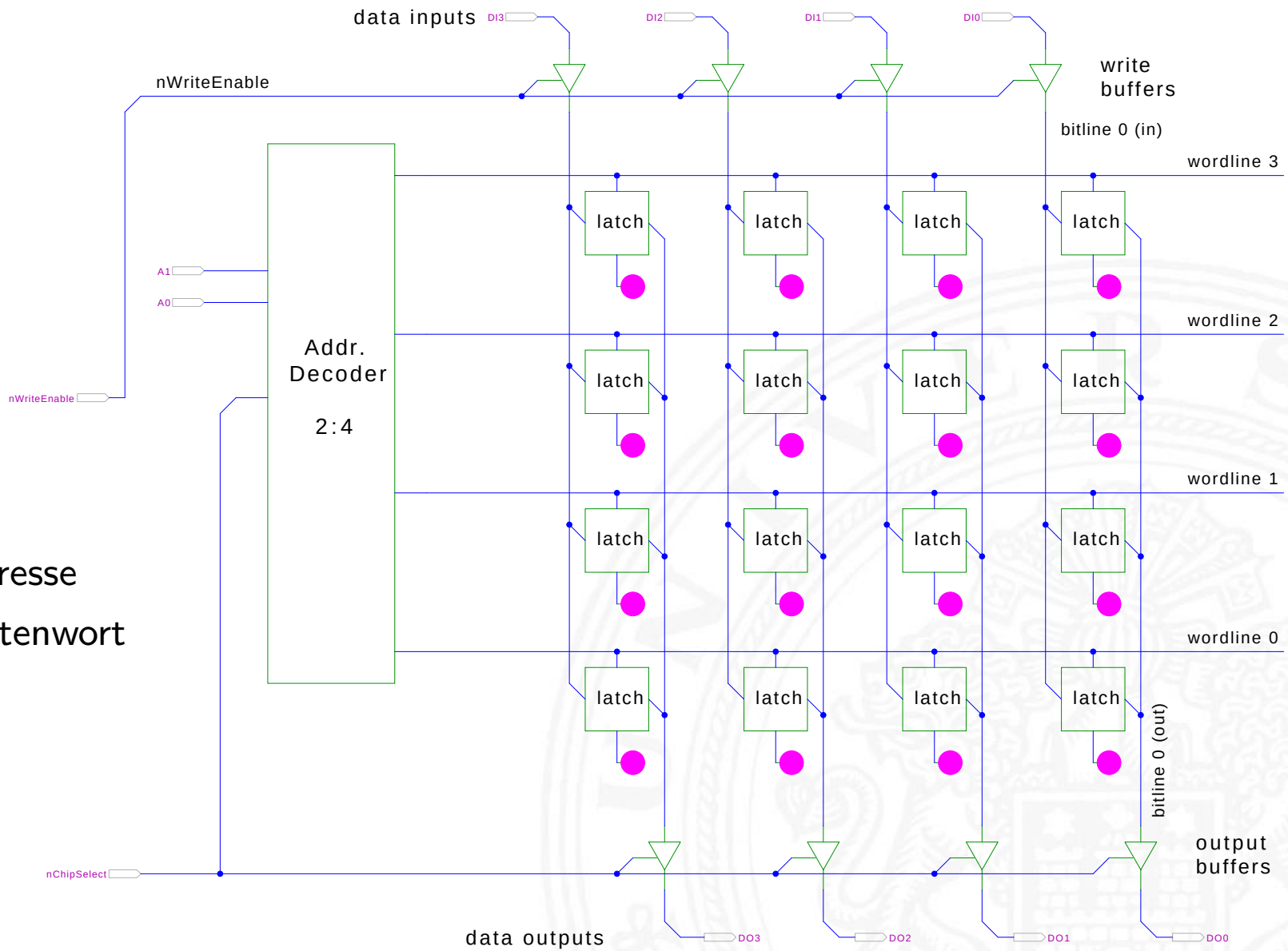
RAM: Random-Access Memory

Speicher, der im Betrieb gelesen und geschrieben werden kann

- Arbeitsspeicher des Rechners
- für Programme und Daten
- keine Abnutzungseffekte
- benötigt Spannungsversorgung zum Speichern

- Aufbau als Matrixstruktur
- n Adressbits, konzeptionell 2^n Wortleitungen
- m Bits pro Wort
- Realisierung der einzelnen Speicherstellen?
  - statisches RAM: 6-Transistor Zelle SRAM
  - dynamisches RAM: 1-Transistor Zelle DRAM

RAM: Blockschaltbild



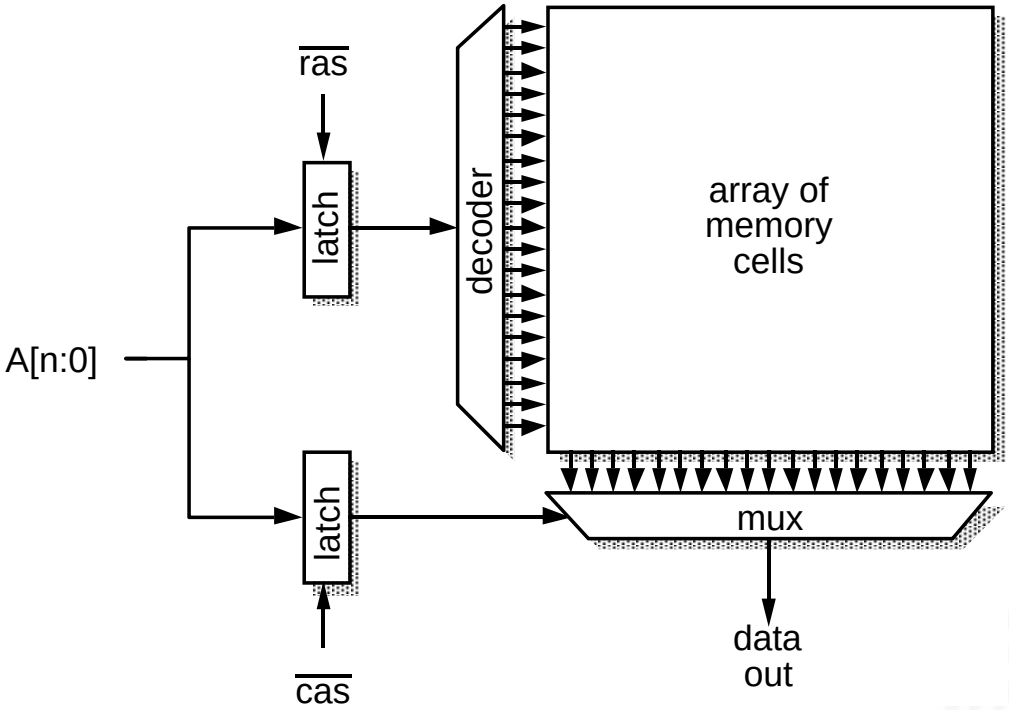
4×4 bit

2-bit Adresse

4-bit Datenwort

[HenHA] Hades Demo: `40-memories/40-ram/ram`

RAM: RAS/CAS-Adressdecodierung



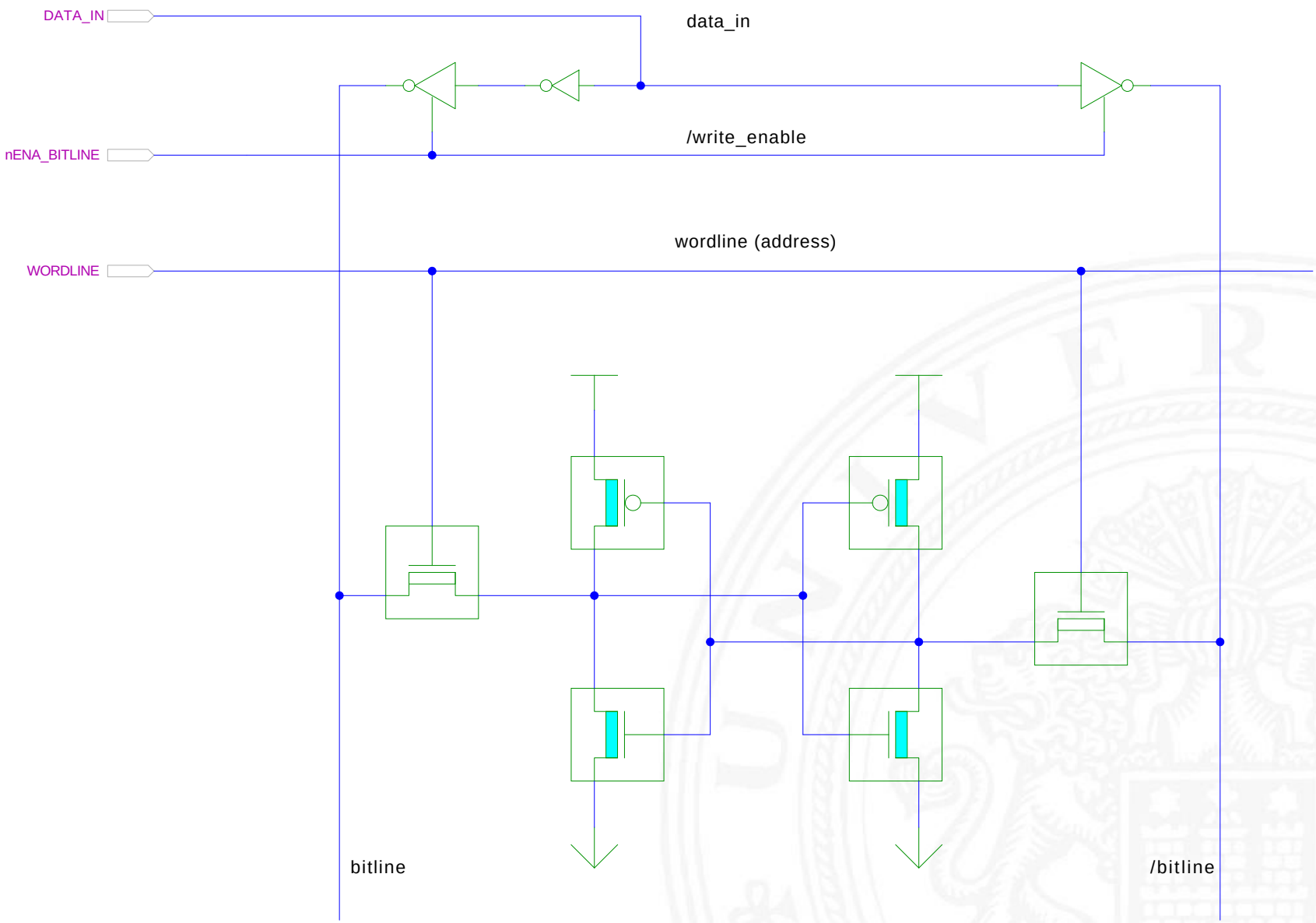
Furber: *ARM SoC Architecture* [Fur00]

- Aufteilen der Adresse in zwei Hälften
- $\overline{ras}$ „row address strobe“ wählt „Wordline“
  $\overline{cas}$ „column address strobe“ –"– „Bitline“
- je ein $2^{(n/2)}$-bit Decoder/Mux statt ein 2^n-bit Decoder

SRAM: statisches RAM

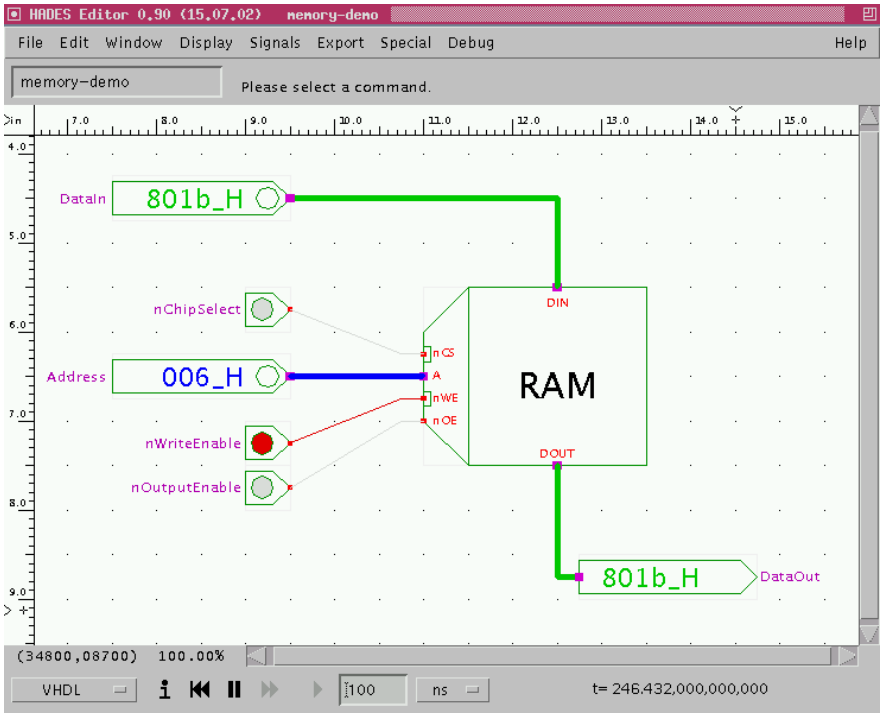
- Inhalt bleibt gespeichert solange Betriebsspannung anliegt
- *sechs-Transistor* Zelle zur Speicherung
  - weniger Platzverbrauch als Latches/Flipflops
  - kompakte Realisierung in CMOS-Technologie (s.u.)
  - zwei rückgekoppelte Inverter zur Speicherung
  - zwei n-Kanal Transistoren zur Anbindung an die Bitlines
- schneller Zugriff: Einsatz für Caches
- deutlich höherer Platzbedarf als DRAMs

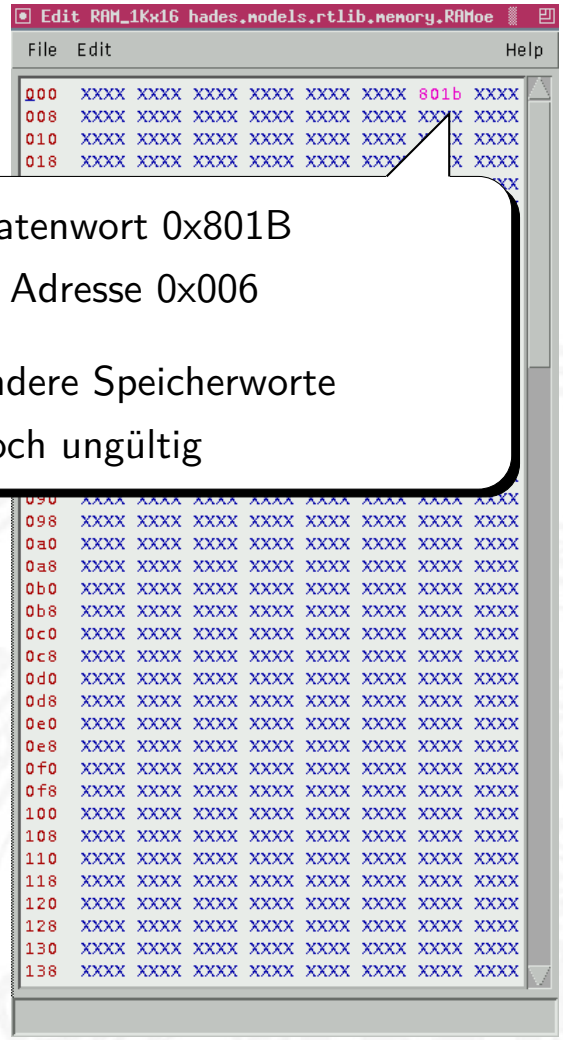
SRAM: Sechs-Transistor Speicherstelle („6T“)



[HenHA] Hades Demo: `05-switched/40-cmos/sramcell`

SRAM: Hades Demo





Datenwort 0x801B
in Adresse 0x006

andere Speicherworte
noch ungültig

- nur aktiv wenn $nCS = 0$ (*chip select*)
- Schreiben wenn $nWE = 0$ (*write enable*)
- Ausgabe wenn $nOE = 0$ (*output enable*)

[HenHA] Hades Demo: `50-rtlib/40-memory/ram`

SRAM: Beispiel IC 6116

- integrierte Schaltung, 16 Ki bit Kapazität
- Organisation als 2 Ki Worte mit je 8-bit

- 11 Adresseingänge (A10 . . . A0)
- 8 Anschlüsse für gemeinsamen Daten-Eingang/-Ausgang
- 3 Steuersignale
  - $\overline{CS}$ chip-select: Speicher nur aktiv wenn $\overline{CS} = 0$
  - $\overline{WE}$ write-enable: Daten an gewählte Adresse schreiben
  - $\overline{OE}$ output-enable: Inhalt des Speichers ausgeben

- interaktive Hades-Demo zum Ausprobieren [HenHA]
  - Hades Demo: `40-memories/40-ram/demo-6116`
  - Hades Demo: `40-memories/40-ram/two-6116`

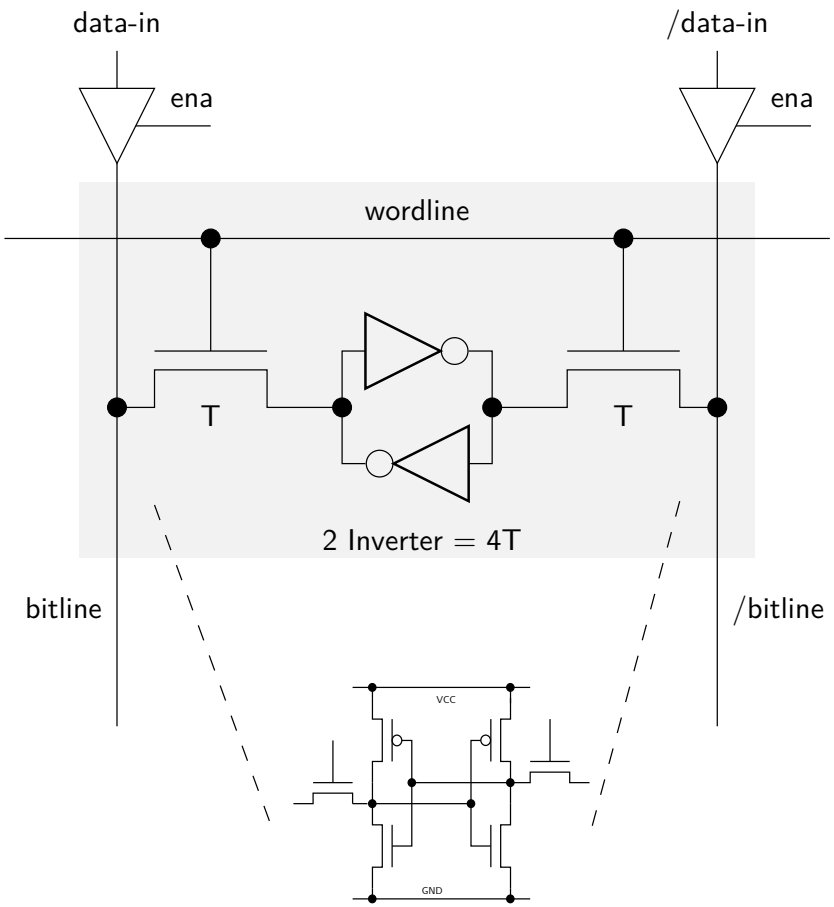
DRAM: dynamisches RAM

- Information wird in winzigen Kondensatoren gespeichert
- pro Bit je ein Transistor und Kondensator

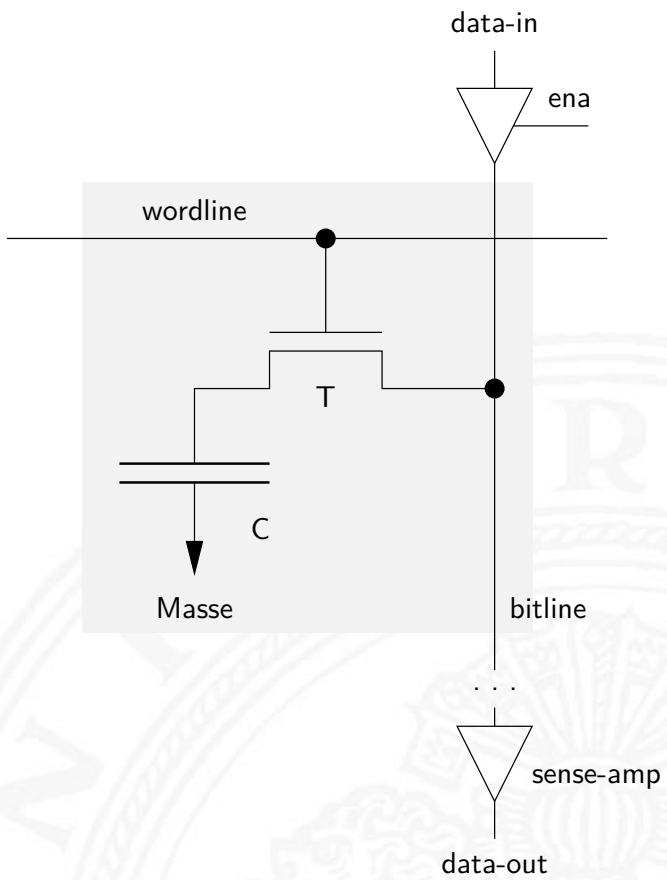
- jeder Lesezugriff entlädt den Kondensator
- *Leseverstärker* zur Messung der Spannung auf der Bitline Schwellwertvergleich zur Entscheidung logisch 0/1

– Information muss anschließend neu geschrieben werden
– auch ohne Lese- oder Schreibzugriff ist regelmäßiger *Refresh* notwendig, wegen Selbstentladung (Millisekunden)
– $10\times$ langsamer als SRAM
\+ DRAM für hohe Kapazität optimiert, minimaler Platzbedarf

SRAM vs. DRAM



- 6 Transistoren/bit
- statisch (kein refresh)
- schnell
- $10 \ldots 50 \times$ DRAM Fläche



- 1 Transistor/bit
- $C = 5\,\text{fF} \approx 47\,000$ Elektronen
- langsam (sense-amp)
- minimale Fläche

DRAM: Stacked- und Trench-Zelle

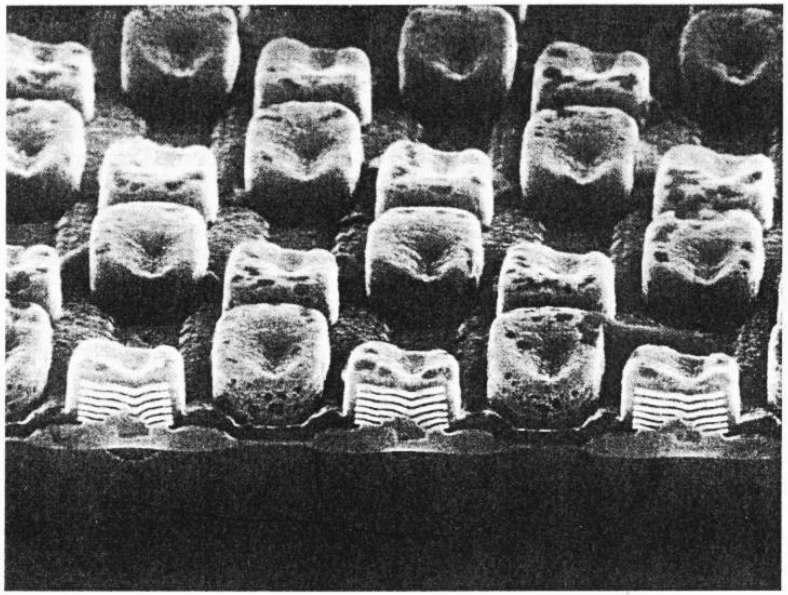
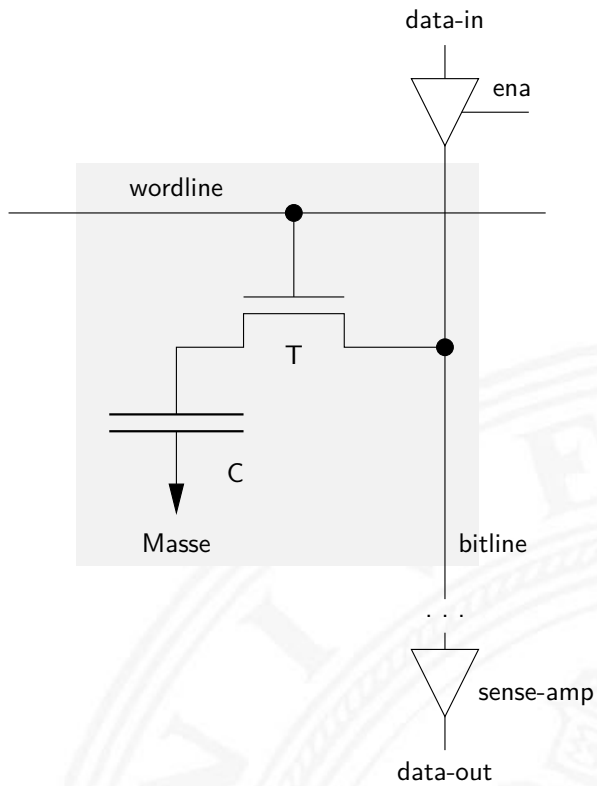


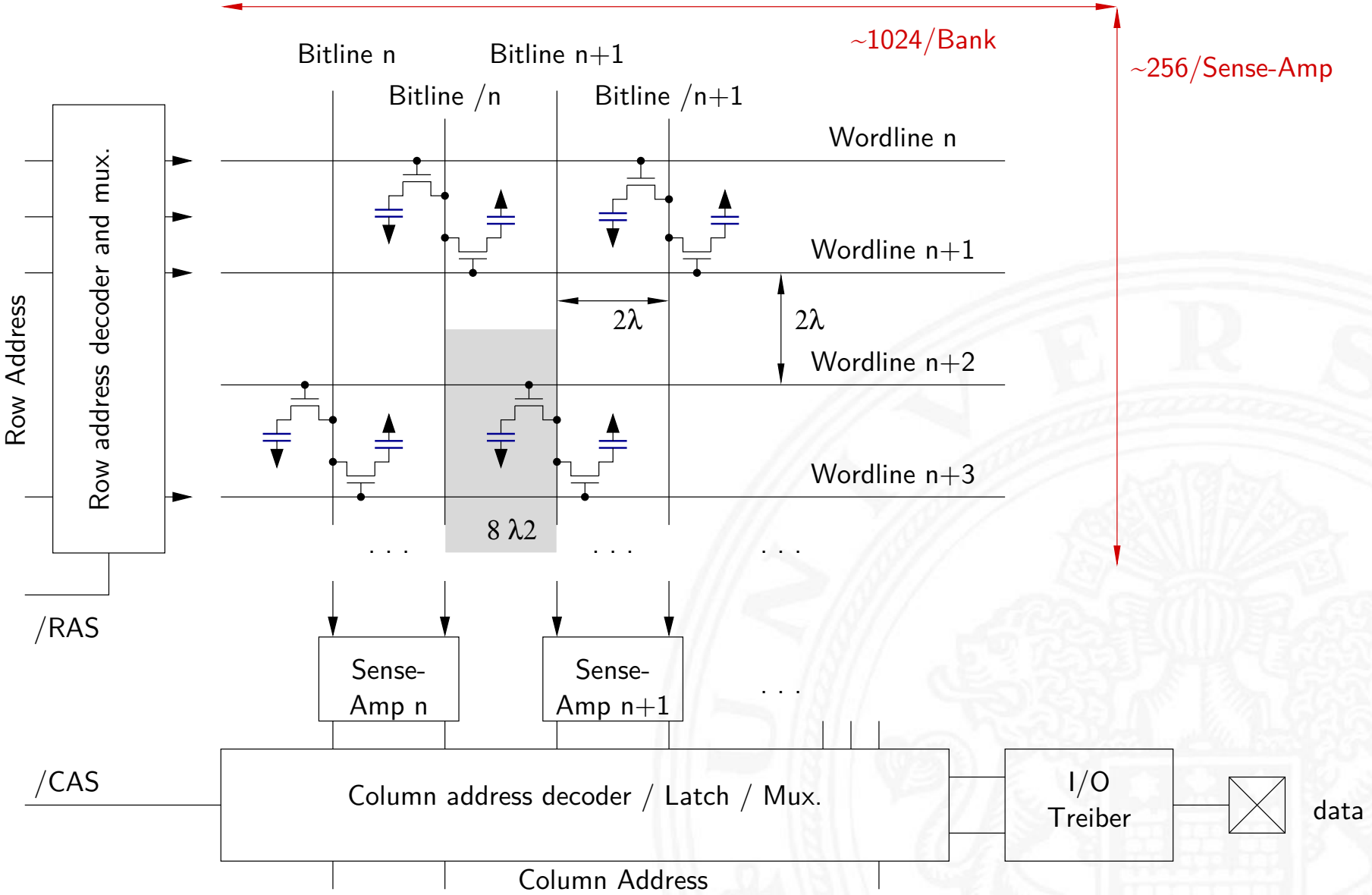
Abb. 7: Prototyp von Speicherzellen (Stapelkondensatoren) für zukünftige Speicherchips wie den Ein-Gigabit-Chip. Da für DRAM-Chips eine minimale Speicherkapazität von 25 fF notwendig ist, bringt es erhebliche Platzvorteile, die Kondensatorelemente vertikal übereinander zu stapeln. Die Dicke der Schichten beträgt etwa 50 nm. (Foto: Siemens)



Siemens 1 Gbit DRAM IBM CMOS-6X embedded DRAM

- zwei Bauformen: „stacked“ und „trench“
- Kondensatoren
  - möglichst kleine Fläche
  - Kapazität gerade ausreichend

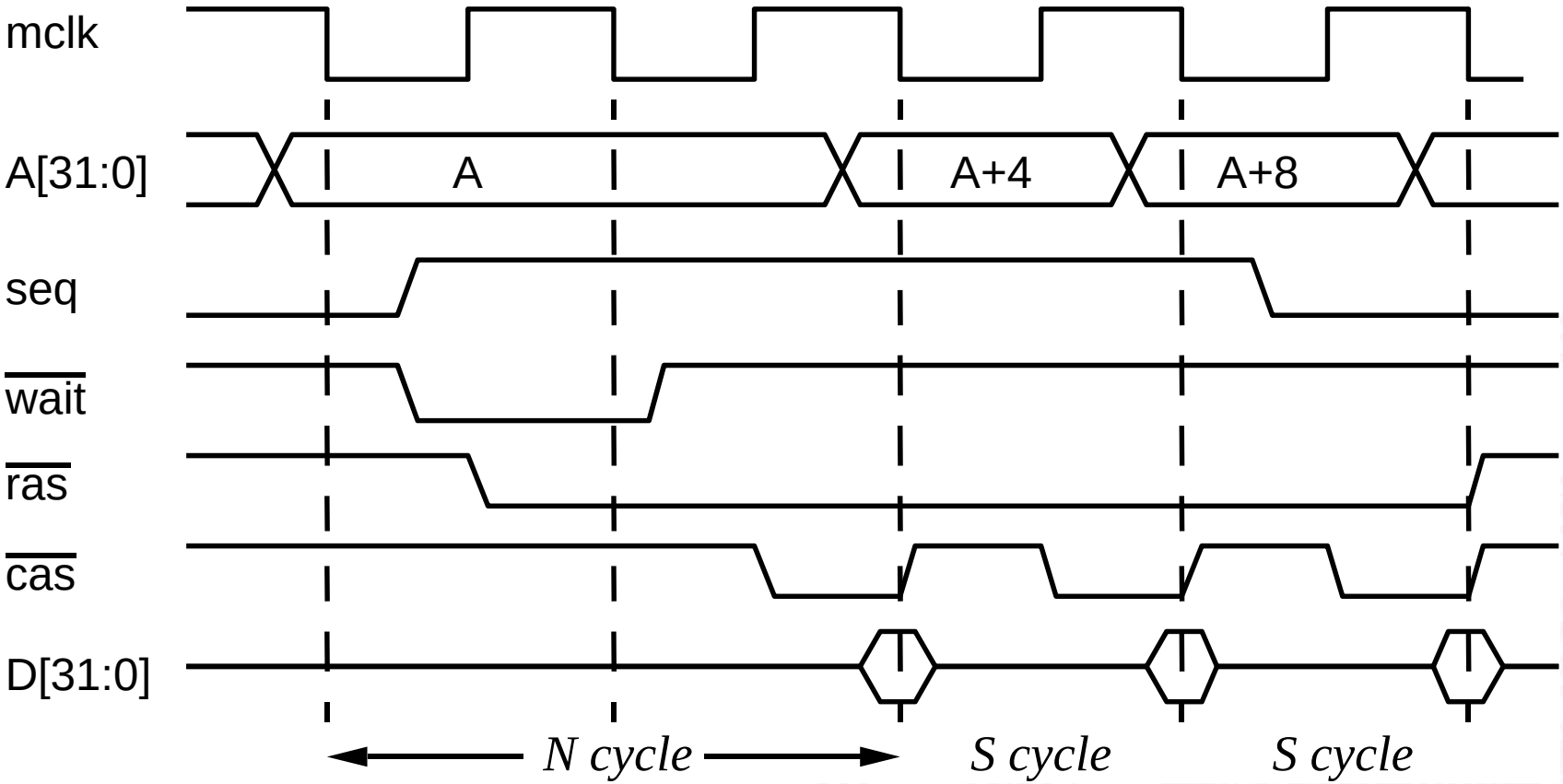
DRAM: Layout


~1024/Bank
~256/Sense-Amp
Bitline n
Bitline n+1
Bitline /n
Bitline /n+1
Wordline n
Wordline n+1
Wordline n+2
Wordline n+3
2λ
2λ
8 λ2
Row Address
Row address decoder and mux.
/RAS
Sense-
Amp n
Sense-
Amp n+1
/CAS
Column address decoder / Latch / Mux.
I/O
Treiber
data
Column Address

DRAM: Varianten

- veraltete Varianten
  - FPM: *fast-page mode*
  - EDO: *extended data-out*
  - ...
- heute gebräuchlich
  - SDRAM: Ansteuerung synchron zu Taktsignal
  - DDR-SDRAM: *double-data rate* Ansteuerung wie SDRAM
    Daten werden mit steigender und fallender Taktflanke übertragen
  - DDR2 . . . DDR5: Varianten mit höherer Taktrate
    aktuelle Übertragungsraten bis 51,2 GByte/sec pro Speicherkanal
  - GDDR3 . . . GDDR6 (*Graphics Double Data Rate*)
    derzeit bis 128 GByte/sec
  - HBM . . . HBM3 (*High Bandwidth Memory*)
    derzeit bis 512 GByte/sec
  - HMC . . . HMC2 (*Hybrid Memory Cube*)
    derzeit bis 480 GByte/sec

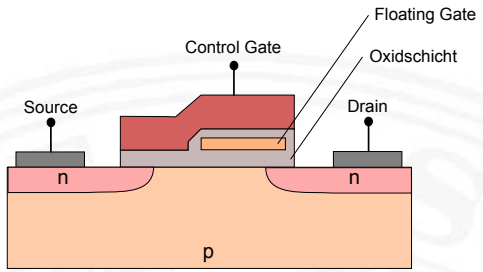
SDRAM: Lesezugriff auf sequenzielle Adressen



[Fur00]

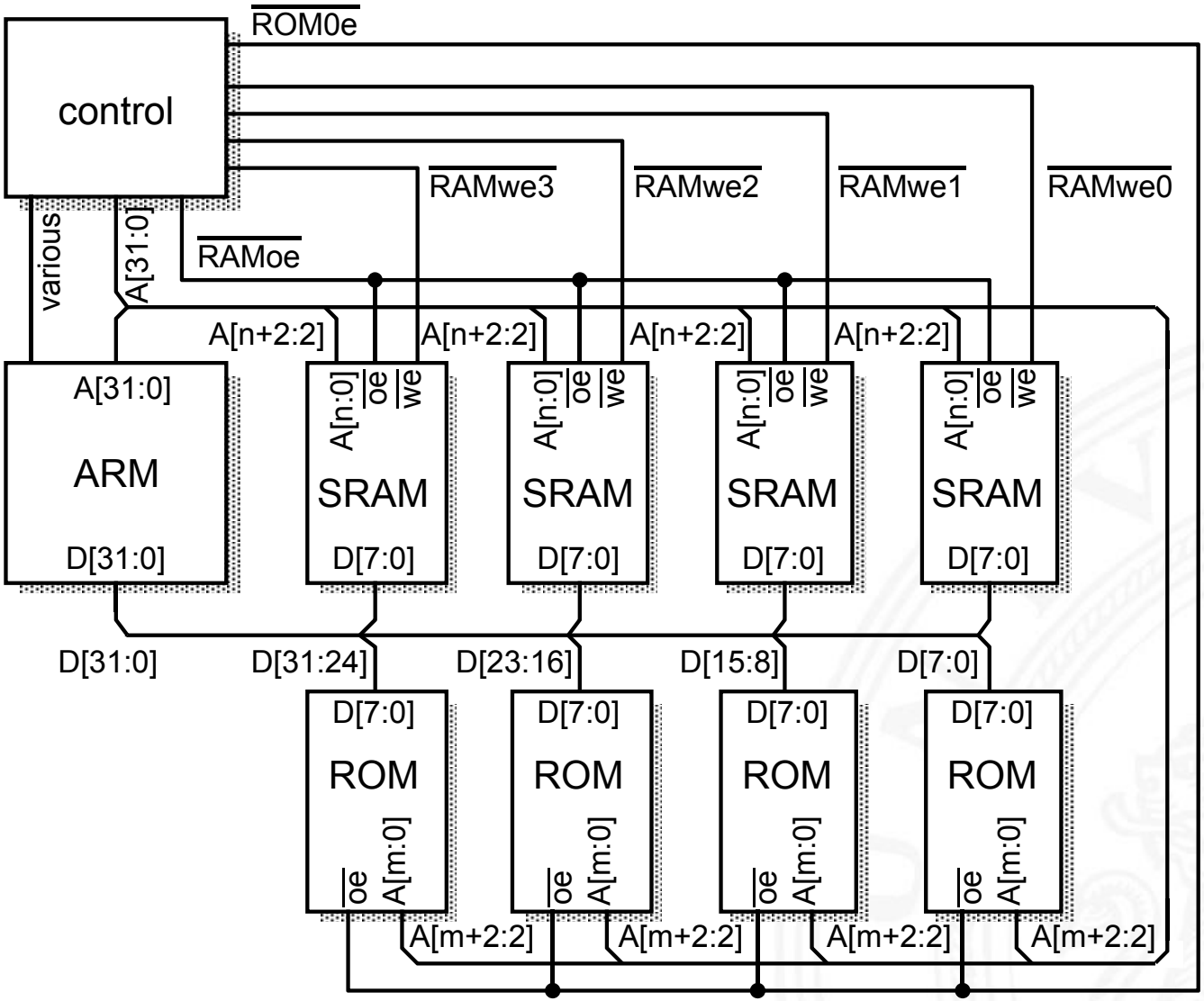
Flash

- ähnlich kompakt und kostengünstig wie DRAM
- nichtflüchtig (*non-volatile*): Information bleibt beim Ausschalten erhalten



- spezielle *floating-gate* Transistoren
  - das *floating-gate* ist komplett nach außen isoliert
  - einmal gespeicherte Elektronen sitzen dort fest
- Auslesen beliebig oft möglich, schnell
- Schreibzugriffe problematisch
  - intern hohe Spannung erforderlich (Gate-Isolierung überwinden)
  - Schreibzugriffe einer „0“ nur blockweise
  - pro Zelle nur einige 10 000 . . . 100 M Schreibzugriffe möglich

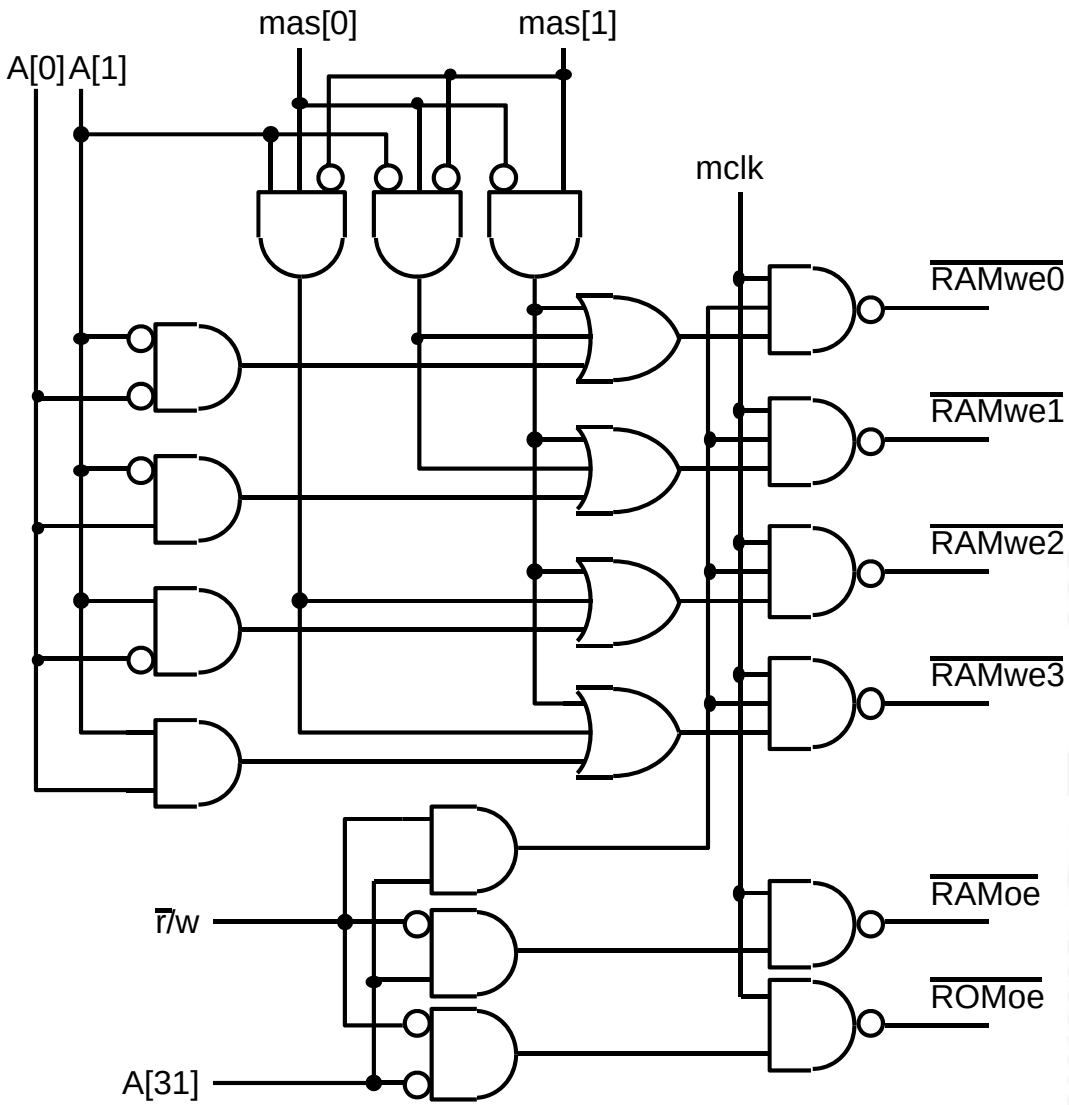
Typisches Speichersystem



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

[Fur00]

Typisches Speichersystem: Adressdecodierung



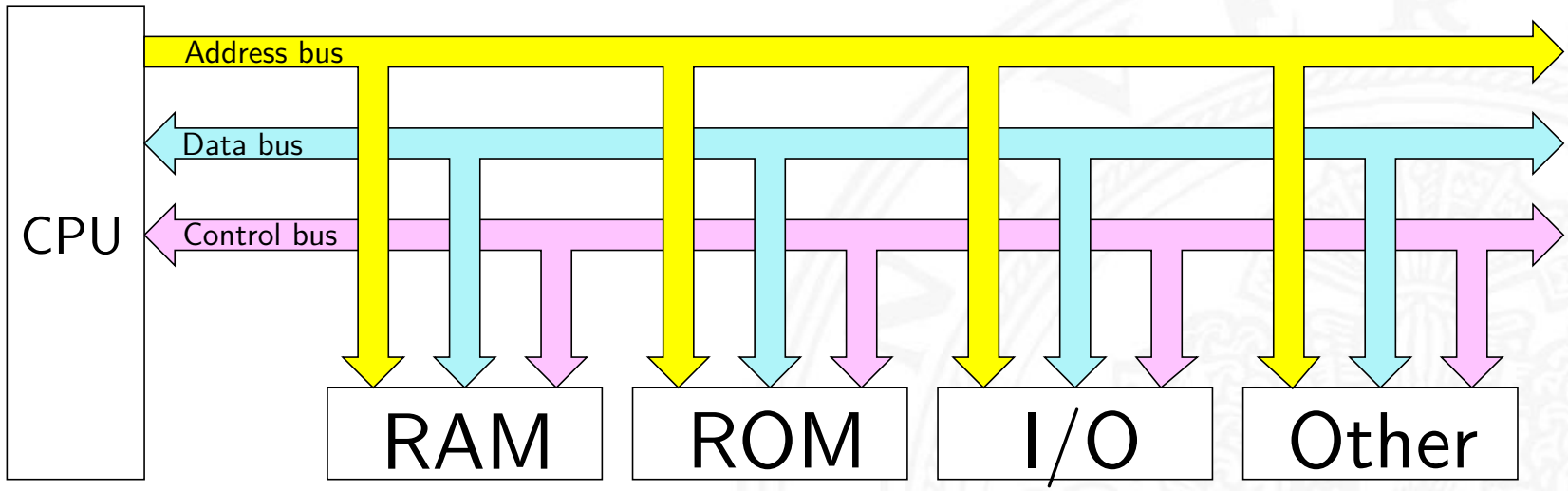
[Fur00]

Bussysteme

- **Bus:** elektrische (und logische) Verbindung
  - mehrere Geräte
  - mehrere Blöcke innerhalb einer Schaltung
- Bündel aus Daten- und Steuersignalen
- mehrere Quellen (und mehrere Senken [lesende Zugriffe])
  - spezielle elektrische Realisierung:
    Tri-State-Treiber oder Open-Drain
- Bus-Arbitrierung: wer darf, wann, wie lange senden?
  - Master-Slave
  - gleichberechtigte Knoten, Arbitrierungsprotokolle
- synchron: mit globalem Taktsignal vom „Master“-Knoten
  asynchron: Wechsel von Steuersignalen löst Ereignisse aus

Bussysteme (cont.)

- typische Aufgaben
  - Kernkomponenten (CPU, Speicher . . .) miteinander verbinden
  - Verbindungen zu den Peripherie-Bausteinen
  - Verbindungen zu Systemmonitor-Komponenten
  - Verbindungen zwischen I/O-Controllern und -Geräten
  - . . .



Bussysteme (cont.)

- viele unterschiedliche Typen, standardisiert
  mit sehr unterschiedlichen Anforderungen
  - High-Performance
  - einfaches Protokoll, billige Komponenten
  - Multi-Master-Fähigkeit, zentrale oder dezentrale Arbitrierung
  - Echtzeitfähigkeit, Daten-Streaming
  - wenig Leitungen bis zu Zweidraht-Bussen:
    I^2C, SPI, System-Management-Bus ...
  - lange Leitungen: EIA-485, RS-232, Ethernet ...
  - Funkmedium: WLAN, Bluetooth (logische Verbindung)

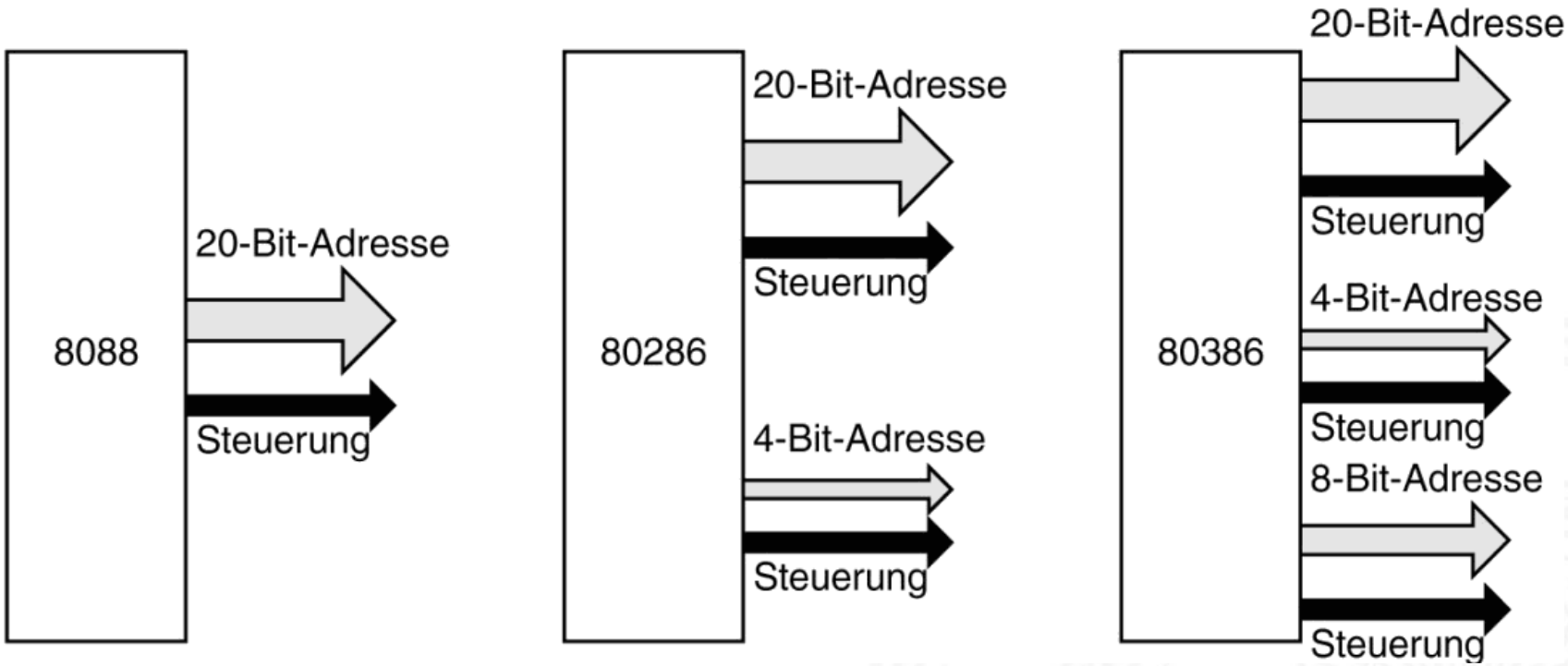
Bus: Mikroprozessorsysteme

typisches n-bit Mikroprozessor-System:

- n Adress-Leitungen, also Adressraum 2^n Bytes — Adressbus
- n Daten-Leitungen — Datenbus

- Steuersignale — Control
  - clock: Taktsignal
  - read/write: Lese-/Schreibzugriff (aus Sicht des Prozessors)
  - wait: Wartezeit/-zyklen für langsame Geräte
  - ...
- um Leitungen zu sparen, teilweise gemeinsam genutzte Leitungen sowohl für Adressen als auch Daten.
  Zusätzliches Steuersignal zur Auswahl Adressen/Daten

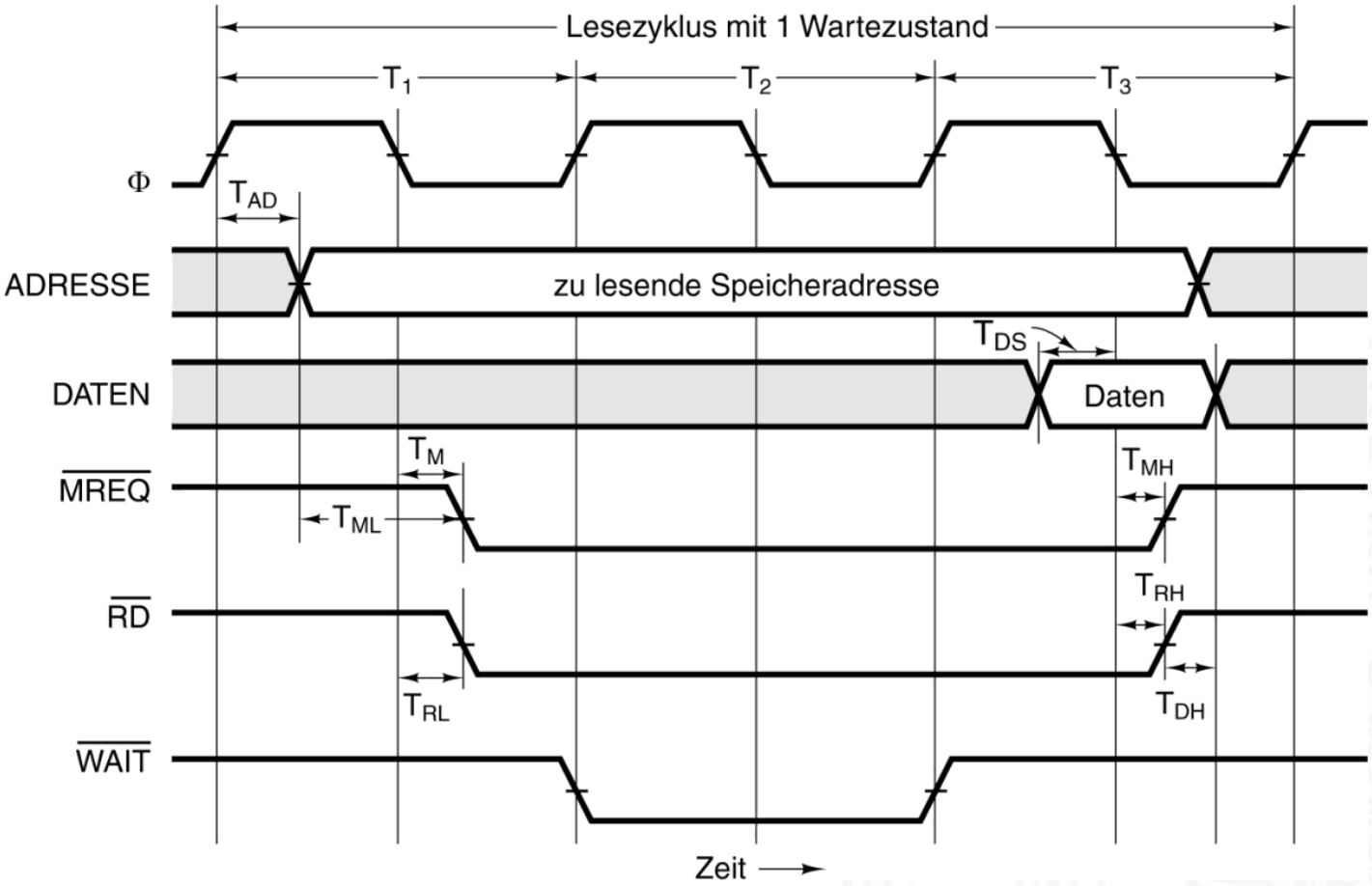
Adressbus: Evolution beim Intel x86



[TA14]

- 20-bit: 1 MiByte Adressraum
  24-bit: 16 MiByte
  32-bit: 4 GiByte
- alle Erweiterungen abwärtskompatibel
- 64-bit Architekturen: 48-, 56-, 64-bit Adressraum

Synchroner Bus: Timing



[TA14]

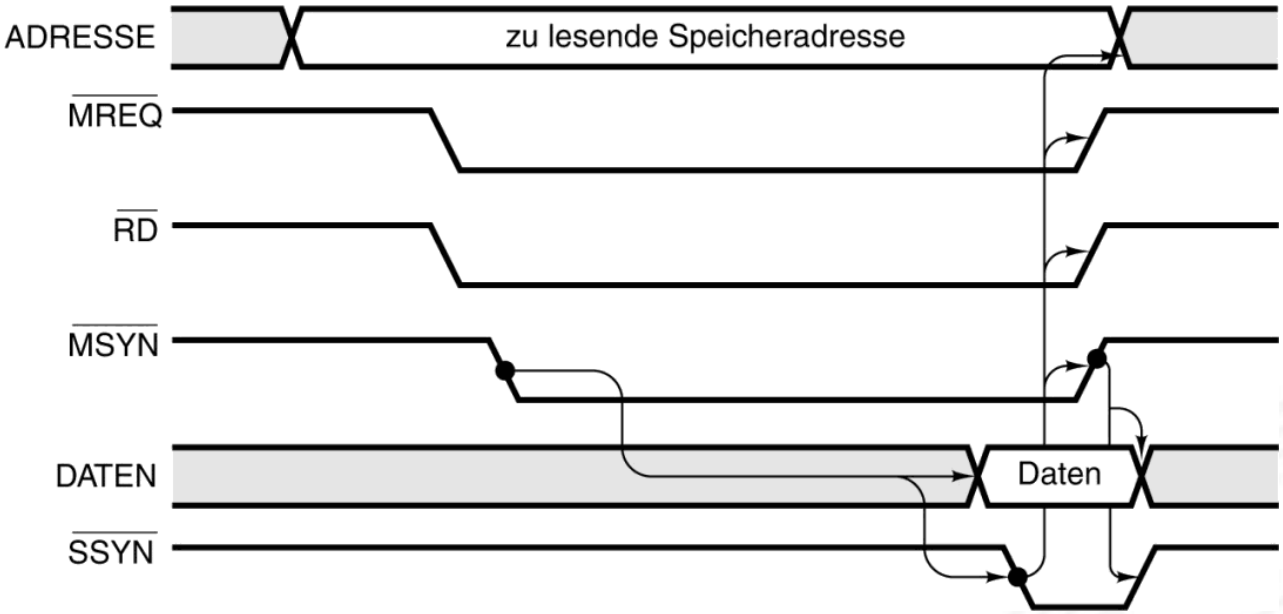
- alle Zeiten über Taktsignal Φ gesteuert
- $\overline{MREQ}$-Signal zur Auswahl Speicher oder I/O-Geräte
- $\overline{RD}$ signalisiert Lesezugriff
- Wartezyklen, solange der Speicher $\overline{WAIT}$ aktiviert

Synchroner Bus: Timing (cont.)

- typische Parameter

Symbol		[ns] Min	Max
T_{AD}	Adressausgabeverzögerung		4
T_{ML}	Adresse ist vor $\overline{MREQ}$ stabil	2	
T_M	$\overline{MREQ}$-Verzögerung nach fallender Flanke von Φ in T_1		3
T_{RL}	RD-Verzögerung nach fallender Flanke von Φ in T_1		3
T_{DS}	*Setup*-Zeit vor fallender Flanke von Φ	2	
T_{MH}	$\overline{MREQ}$-Verzögerung nach fallender Flanke von Φ in T_3		3
T_{RH}	$\overline{RD}$-Verzögerung nach fallender Flanke von Φ in T_3		3
T_{DH}	*Hold*-Zeit nach der Deaktivierung von $\overline{RD}$	0	

Asynchroner Bus: Lesezugriff



[TA14]

- Steuersignale $\overline{MSYN}$: Master fertig
  $\overline{SSYN}$: Slave fertig
- flexibler für Geräte mit stark unterschiedlichen Zugriffszeiten

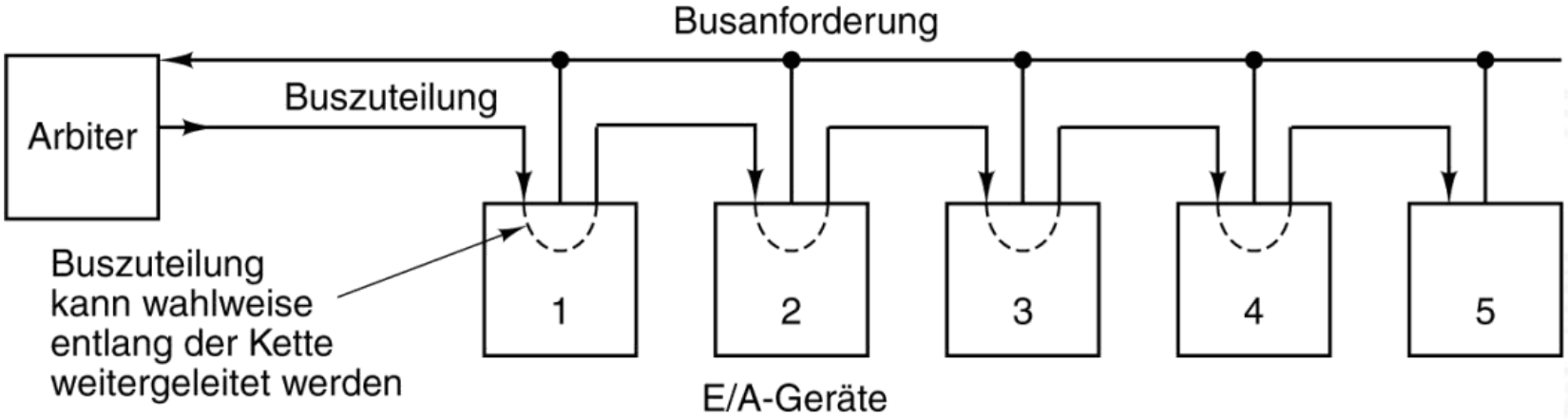
Bus Arbitrierung

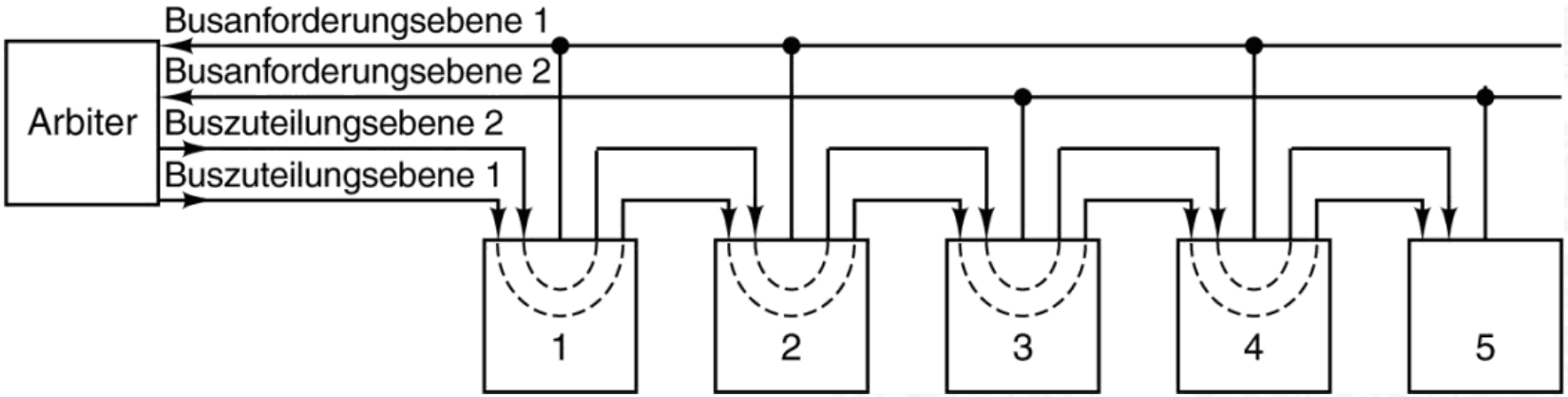
- mehrere Komponenten wollen Übertragung initiieren
  immer nur ein Transfer zur Zeit möglich
- der Zugriff muss serialisiert werden

1. zentrale Arbitrierung
   - Arbiter gewährt Bus-Requests
   - Strategien
     - Prioritäten für verschiedene Geräte
     - „round-robin“ Verfahren
     - „Token“-basierte Verfahren
     - usw.
2. dezentrale Arbitrierung
   - protokollbasiert
   - Beispiel
     - Komponenten sehen ob Bus frei ist
     - beginnen zu senden
     - Kollisionserkennung: gesendete Daten lesen
     - ggf. Übertragung abbrechen
     - „später“ erneut versuchen

Bus Arbitrierung (cont.)

- I/O-Geräte oft höher priorisiert als die CPU
  - I/O-Zugriffe müssen schnell/sofort behandelt werden
  - Benutzerprogramm kann warten





[TA14]

Bus Bandbreite

- Menge an (Nutz-) Daten, die pro Zeiteinheit übertragen werden kann
- zusätzlicher Protokolloverhead ⇒ Brutto- / Netto-Datenrate
RS-232	50	bit/sec	...	460	Kbit/sec
I^2C	100	Kbit/sec (Std.)	...	3,4	Mbit/sec (High Speed)
USB	1,5	Mbit/sec (1.x)	...	40	Gbit/sec (4.0)
ISA	128	Mbit/sec	...		
PCI	1	Gbit/sec (2.0)	...	4,3	Gbit/sec (3.0)
AGP	2,1	Gbit/sec (1x)	...	34,1	Gbit/sec (8x 64-bit)
PCI Express	250	MByte/sec (1.x)	...	63,0	GByte/sec (5.0) x16
HyperTransport	3,2	GByte/sec (1.0)	...	51,2	GByte/sec (3.1)
NVLink	80,0	GByte/sec (1.0)	...	150,0	GByte/sec (2.0)
- `en.wikipedia.org/wiki/List_of_interface_bit_rates`

Beispiel: PCI-Bus

Peripheral Component Interconnect (Intel 1991)

- 33 MHz Takt — optional 66 MHz Takt
- 32-bit Bus-System — optional auch 64-bit
- gemeinsame Adress-/Datenleitungen
- Arbitrierung durch Bus-Master — CPU

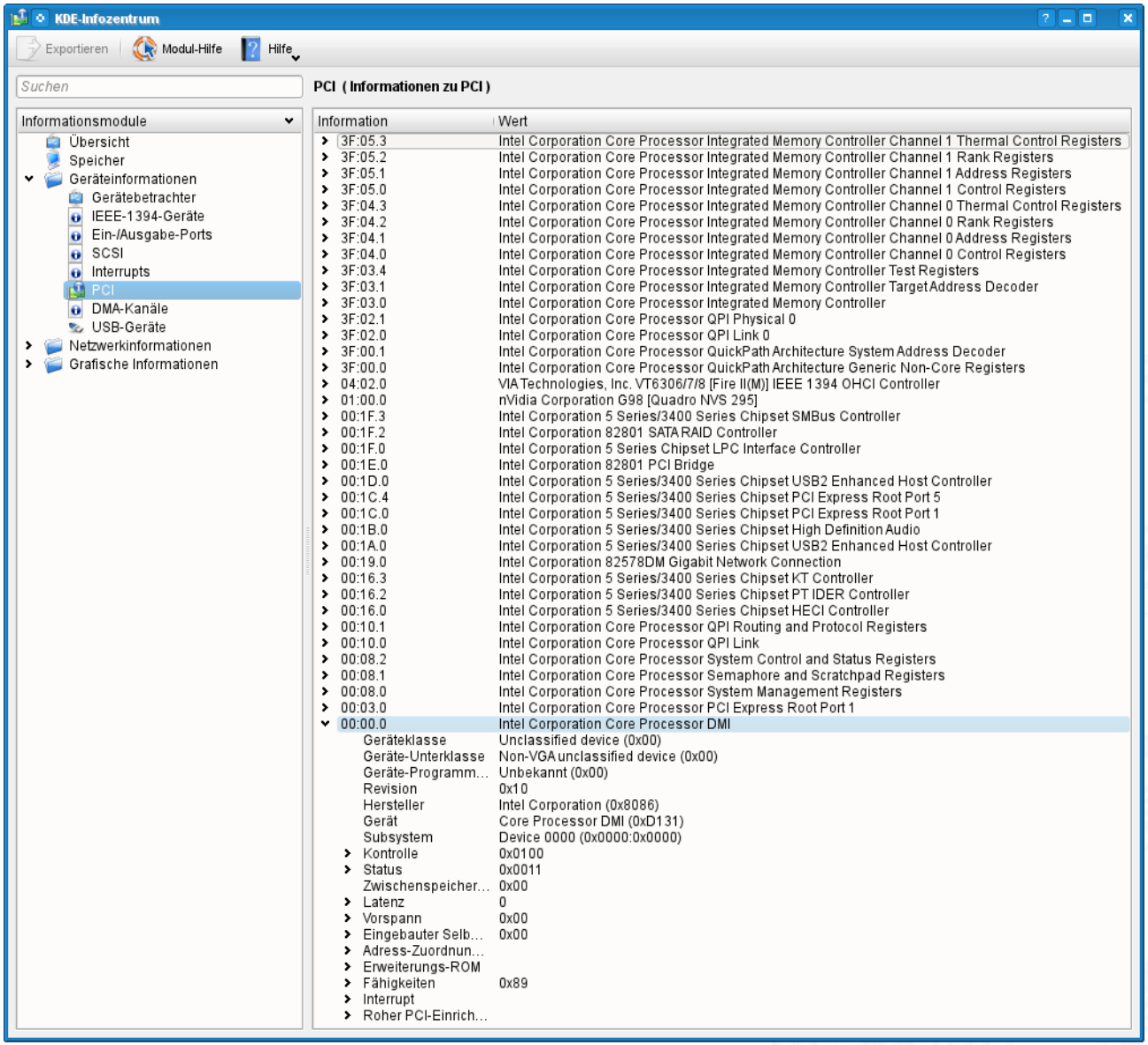
- Abwärtskompatibilität
  - PCI-Bus als logische Verbindung (SW-Layer) zu Komponenten
  - technisch: PCIe

- Auto-Konfiguration
  - angeschlossene Geräte werden automatisch erkannt
  - eindeutige Hersteller- und Geräte-Nummern
  - Betriebssystem kann zugehörigen Treiber laden
  - automatische Zuweisung von Adressbereichen und IRQs

PCI-Bus: Peripheriegeräte


```
[maeder@tams165]~>lspci -v
00:00.0 Host bridge: Intel Corporation Sky Lake Host Bridge/DRAM Registers (rev 08)
        Subsystem: Dell Device 06dc
        Flags: bus master, fast devsel, latency 0
        Capabilities: <access denied>

00:02.0 VGA compatible controller: Intel Corporation Sky Lake Integrated Graphics (rev 07)
        Subsystem: Dell Device 06dc
        Flags: bus master, fast devsel, latency 0, IRQ 134
        Memory at e0000000 (64-bit, non-prefetchable) [size=16M]
        Memory at d0000000 (64-bit, prefetchable) [size=256M]
        I/O ports at f000 [size=64]
        Expansion ROM at <unassigned> [disabled]
        Capabilities: <access denied>
        Kernel driver in use: i915_bpo

00:04.0 Signal processing controller: Intel Corporation Device 1903 (rev 08)
        Subsystem: Dell Device 06dc
        Flags: bus master, fast devsel, latency 0, IRQ 16
        Memory at e1340000 (64-bit, non-prefetchable) [size=32K]
        Capabilities: <access denied>
        Kernel driver in use: proc_thermal

00:14.0 USB controller: Intel Corporation Device 9d2f (rev 21) (prog-if 30 [XHCI])
        Subsystem: Dell Device 06dc
        Flags: bus master, medium devsel, latency 0, IRQ 125
        Memory at e1330000 (64-bit, non-prefetchable) [size=64K]
        ...
```

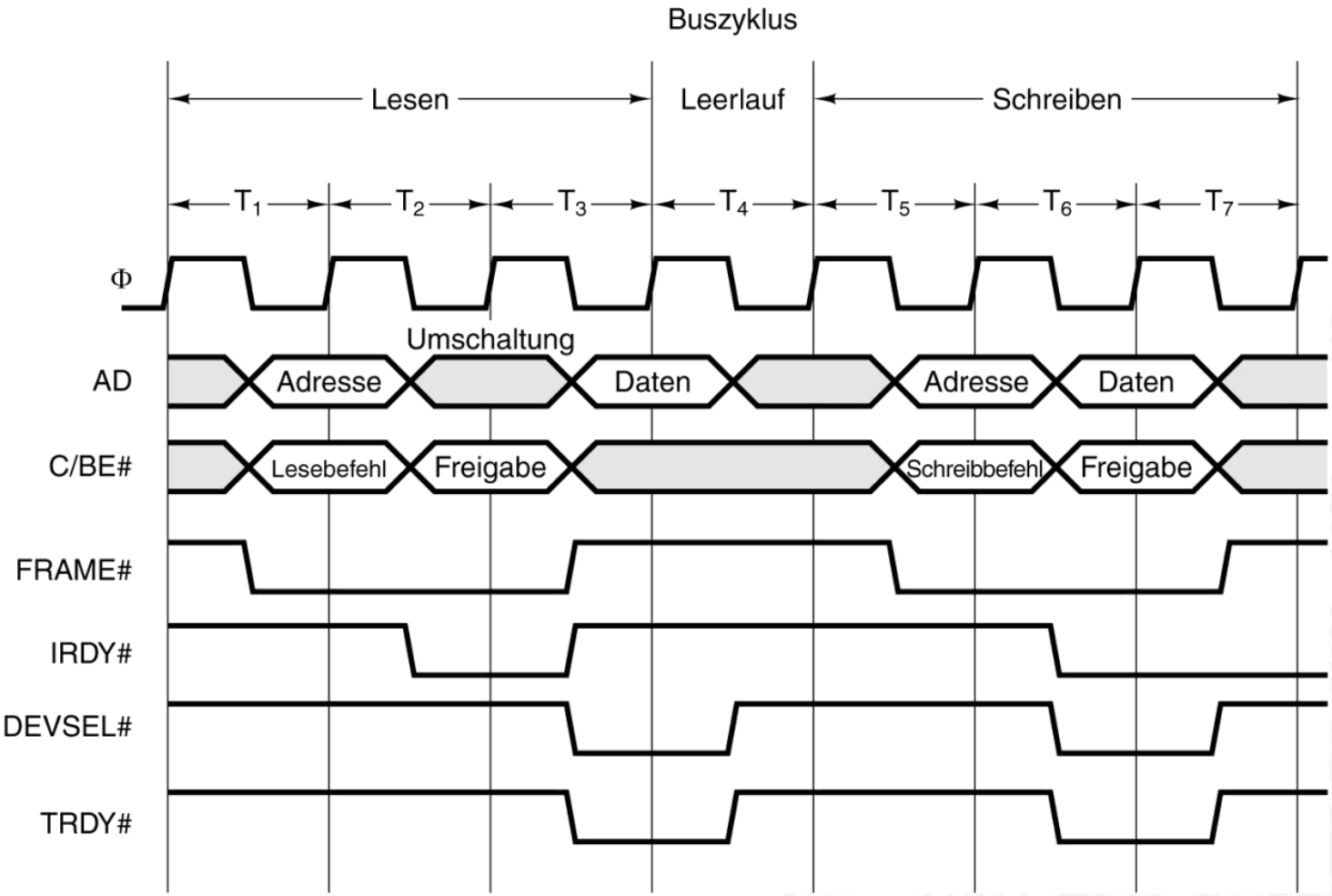
PCI-Bus: Peripheriegeräte (cont.)



PCI-Bus: Leitungen („mindestens“)

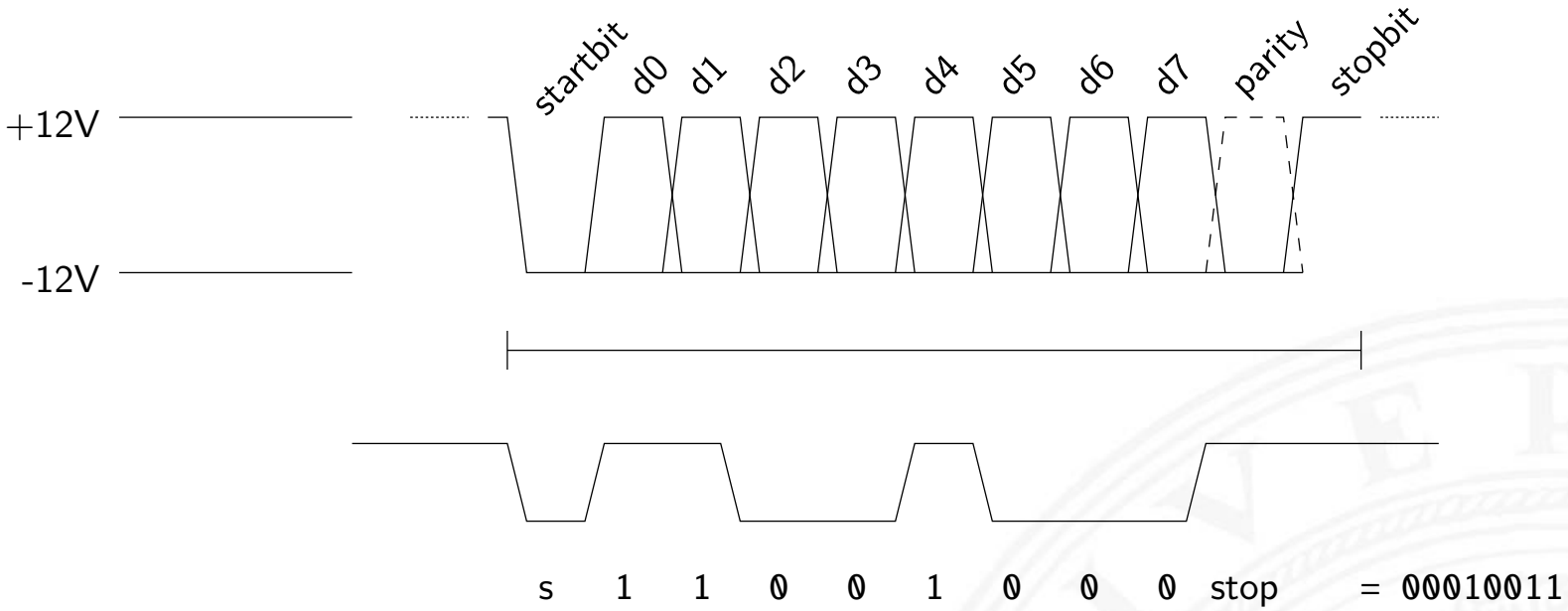
Signal	Leitungen	Master	Slave	Beschreibung
CLK	1			Takt (33 oder 66 MHz)
AD	32	×	×	Gemultiplexte Adress- und Datenleitungen
PAR	1	×		Adress- oder Datenparitätsbit
C/BE	4	×		Busbefehl/Bitmap für Byte Enable (zeigt gültige Datenbytes an)
FRAME#	1	×		Kennzeichnet, dass AD und C/BE aktiviert sind
IRDY#	1	×		Lesen: Master wird akzeptieren Schreiben: Daten liegen an
IDSEL	1	×		Wählt Konfigurationsraum statt Speicher
DEVSEL#	1		×	Slave hat seine Adresse decodiert und ist in Bereitschaft
TRDY#	1		×	Lesen: Daten liegen an Schreiben: Slave wird akzeptieren
STOP#	1		×	Slave möchte Transaktion sofort abbrechen
PERR#	1			Empfänger hat Datenparitätsfehler erkannt
SERR#	1			Adressparätätsfehler oder Systemfehler erkannt
REQ#	1			Bus-Arbitration: Anforderung des Busses
GNT#	1			–"– Zuteilung des Busses
RST#	1			Setzt das System und alle Geräte zurück

PCI-Bus: Transaktionen



[TA14]

RS-232: Serielle Schnittstelle



- Baudrate 300, 600, . . . , 19 200, 38 400, 115 200 bits/sec
  Anzahl Datenbits 5, 6, 7, 8
  Anzahl Stopbits 1, 2
  Parität none, odd, even
- minimal drei Leitungen: GND, TX, RX (Masse, Transmit, Receive)
- oft weitere Leitungen für erweitertes Handshake

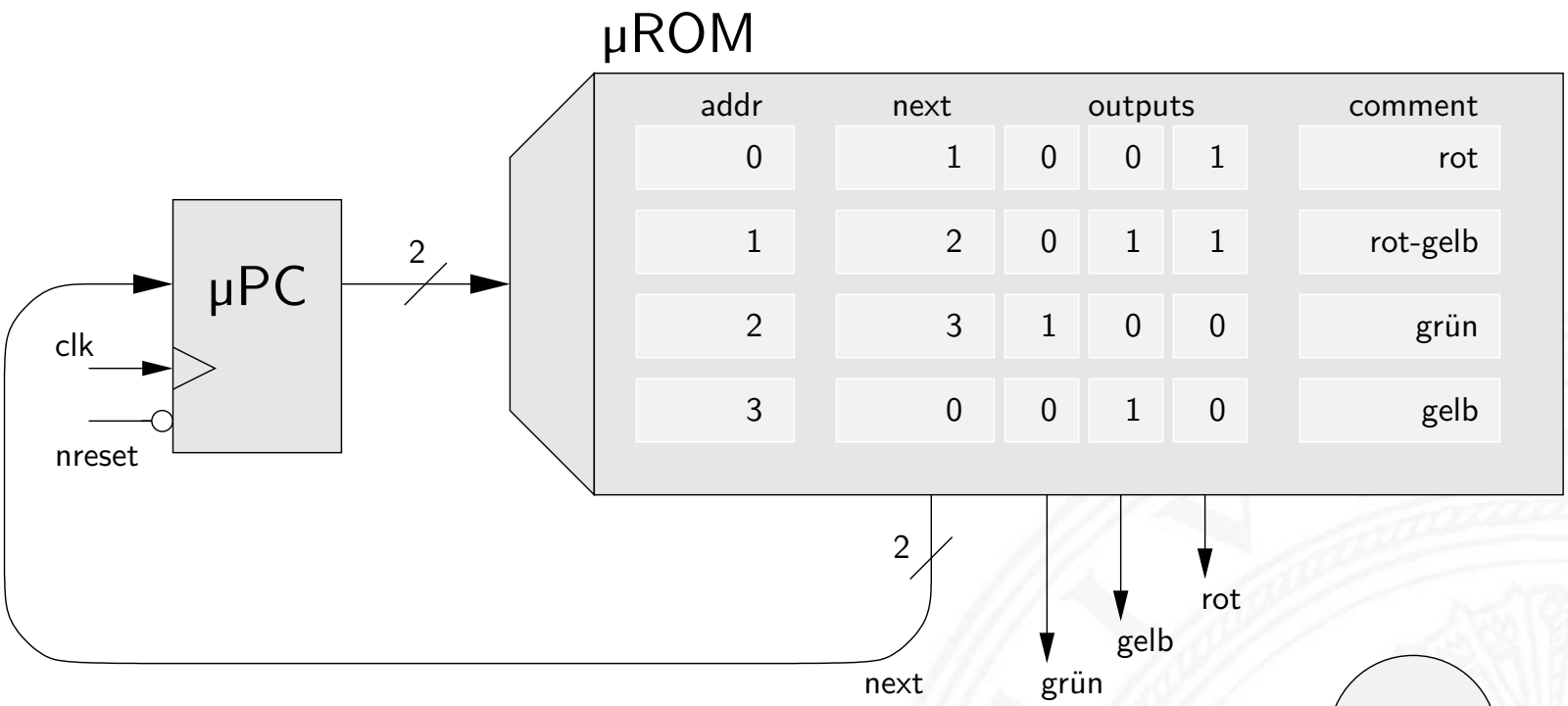
Ablaufsteuerung mit Mikroprogramm

- als Alternative zu direkt entworfenen Schaltwerken

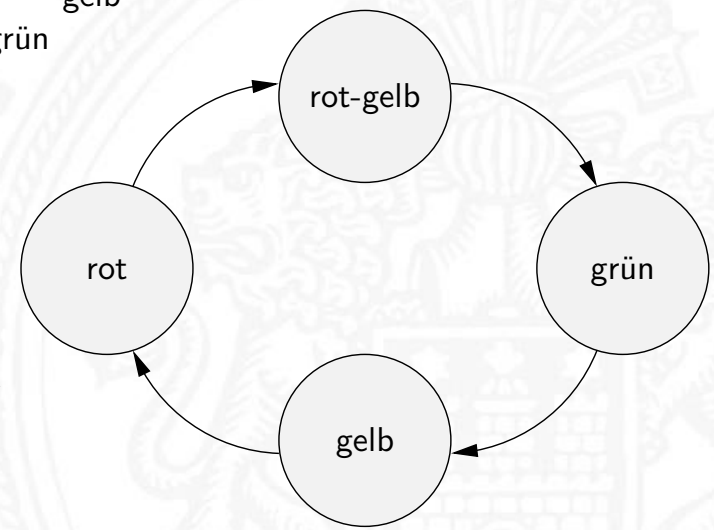
- *Mikroprogrammzähler* μPC: Register für aktuellen Zustand
- μPC adressiert den Mikroprogrammspeicher μROM
- μROM konzeptionell in mehrere Felder eingeteilt
  - die verschiedenen Steuerleitungen
  - ein oder mehrere Felder für Folgezustand
  - ggf. zusätzliche Logik und Multiplexer zur Auswahl unter mehreren Folgezuständen
  - ggf. Verschachtelung und Aufruf von Unterprogrammen: „nanoProgramm“

- siehe „Praktikum Rechnerstrukturen und Betriebssysteme“

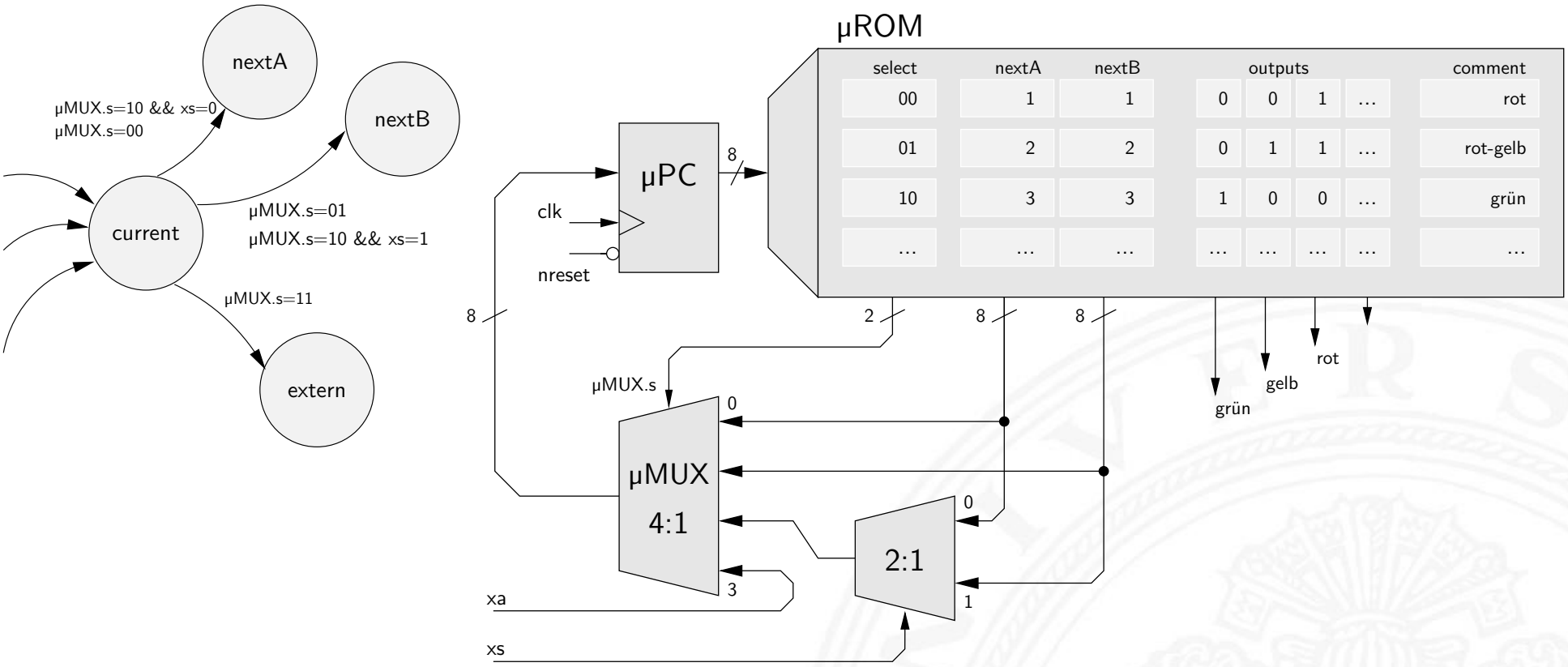
Mikroprogramm: Beispiel Ampel



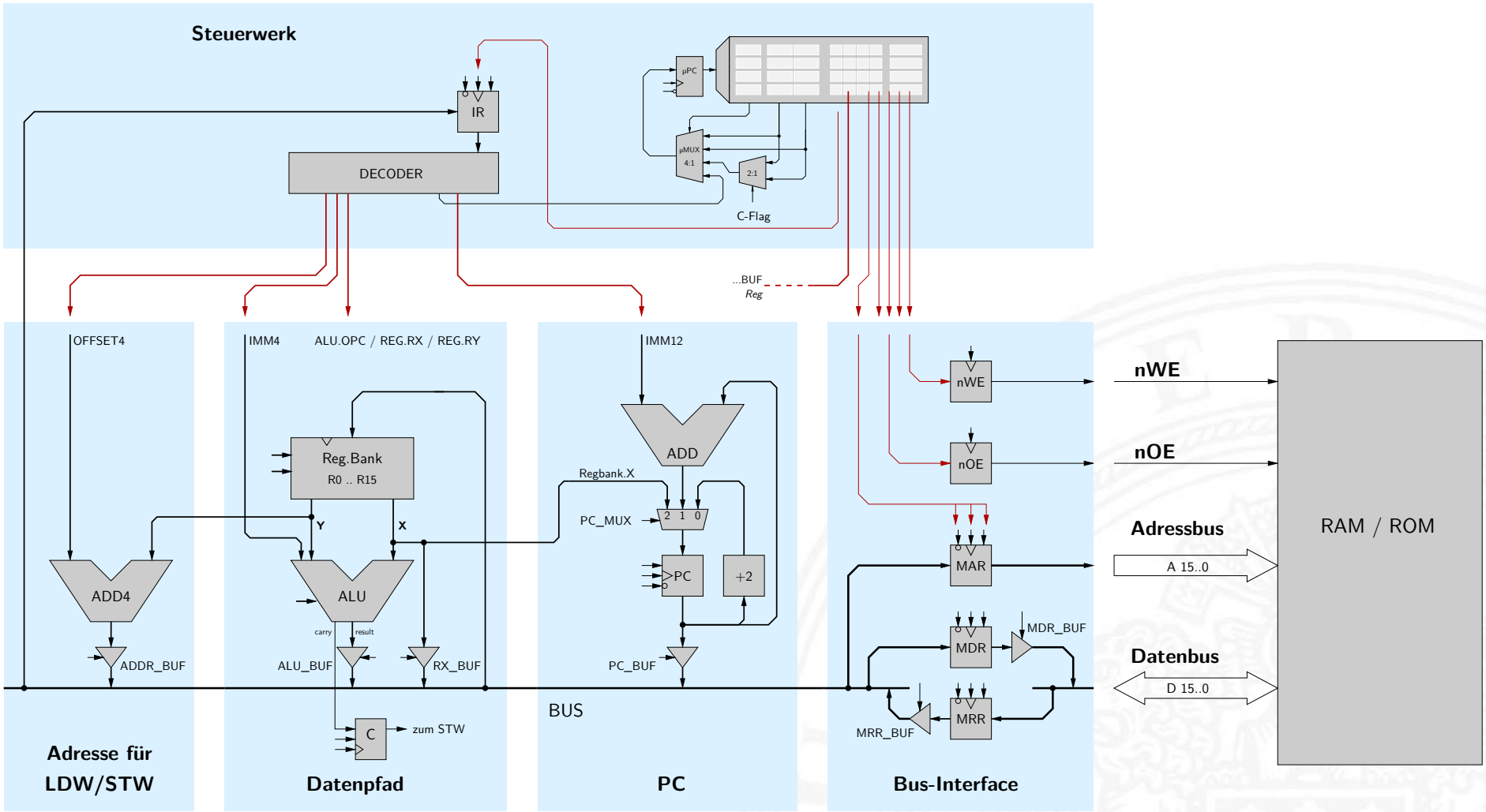
- *µPC* adressiert das *µROM*
- *next*-Ausgang liefert Folgezustand
- andere Ausgänge steuern die Schaltung = die Lampen der Ampel



Mikroprogramm: Beispiel zur Auswahl des Folgezustands



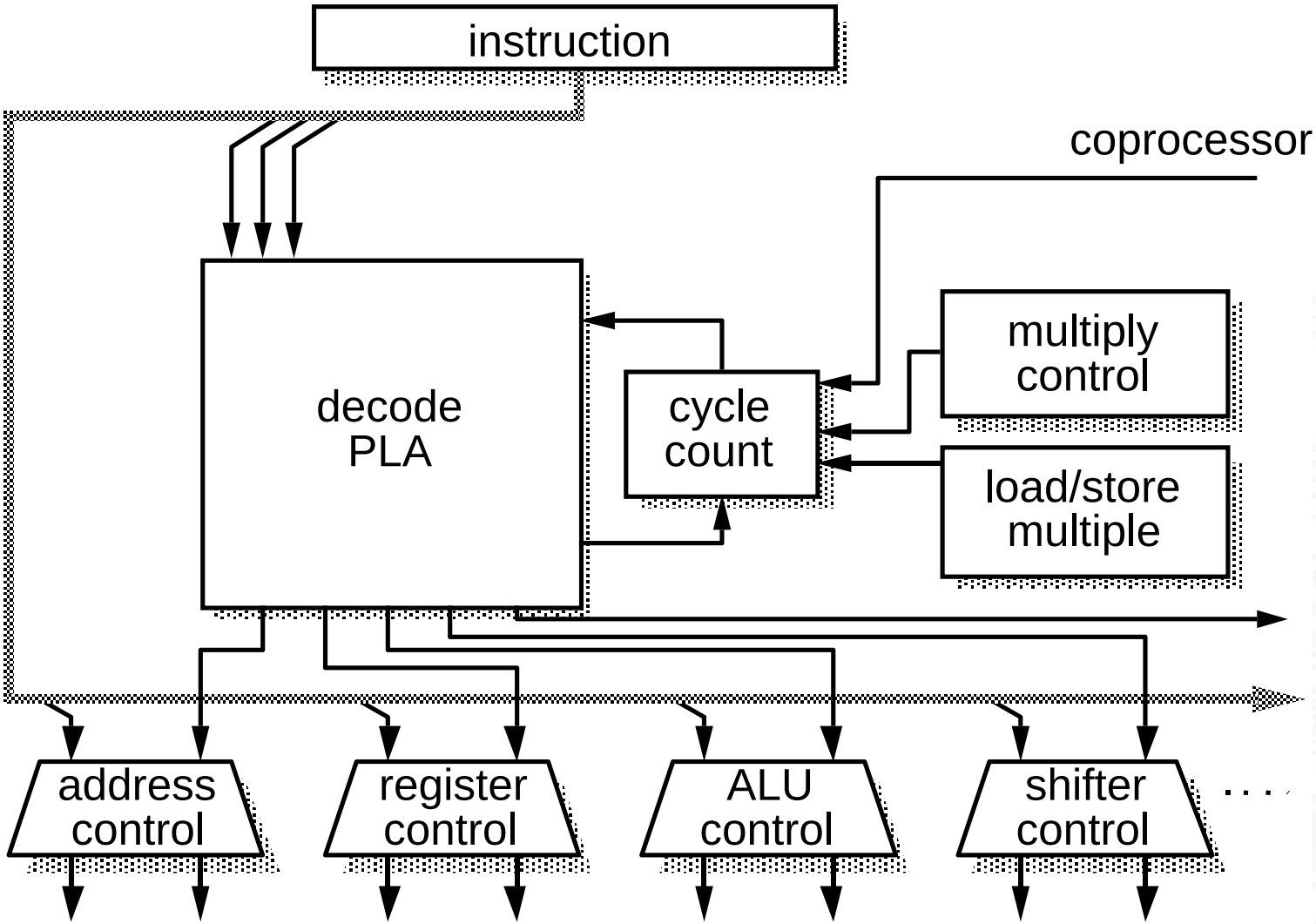
- Multiplexer erlaubt Auswahl des μPC Werts
- $nextA$, $nextB$ aus dem μROM, externer xa Wert
- xs Eingang für bedingte Sprünge



- aktuelle Architekturen: weniger Mikroprogrammierung

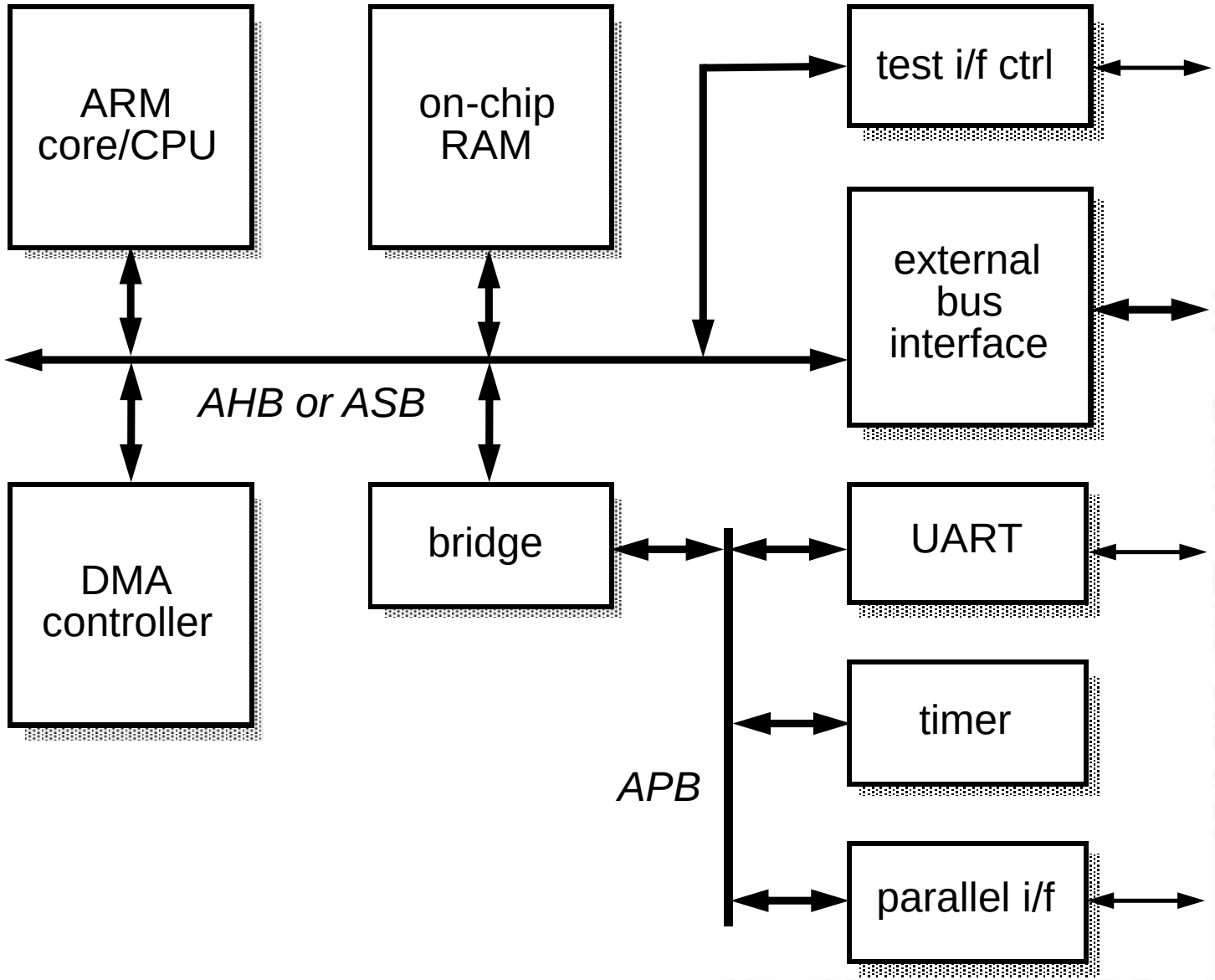
$\Rightarrow$ Pipelining (folgt in *14 Rechnerarchitektur II – Pipelining*)

Mikroprogramm: Befehlsdecoder des ARM 7 Prozessors



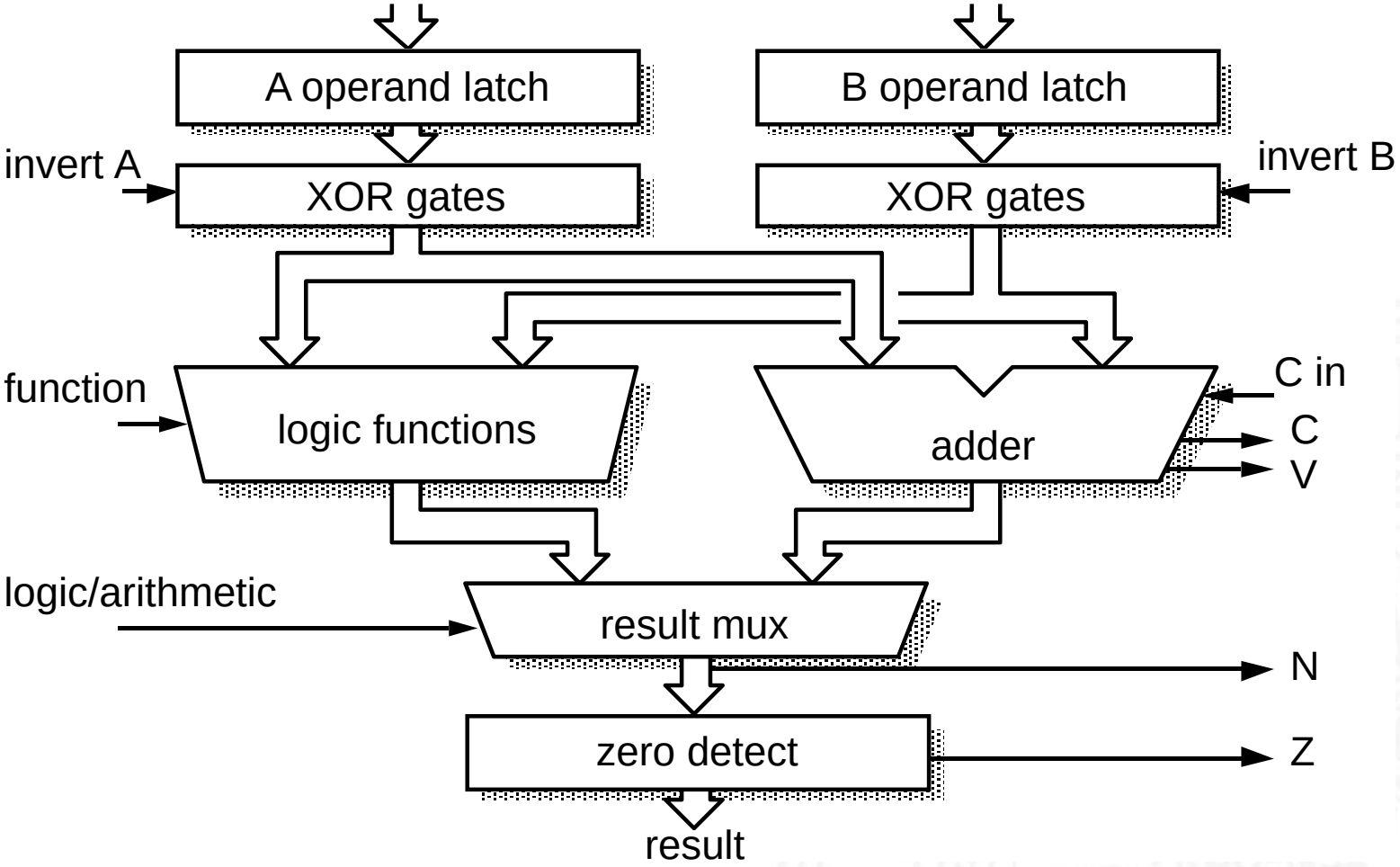
[Fur00]

typisches ARM SoC System



S. Furber: *ARM System-on-Chip Architecture* [Fur00]

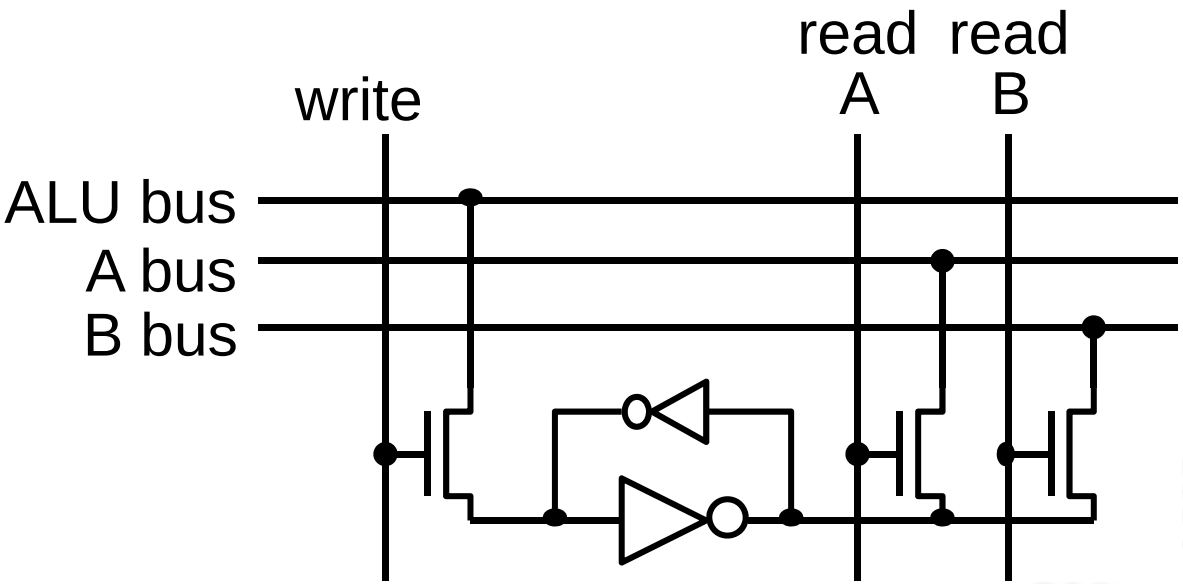
RT-Ebene: ALU des ARM 6 Prozessors



[Fur00]

- Register für die Operanden A und B
- Addierer und separater Block für logische Operationen

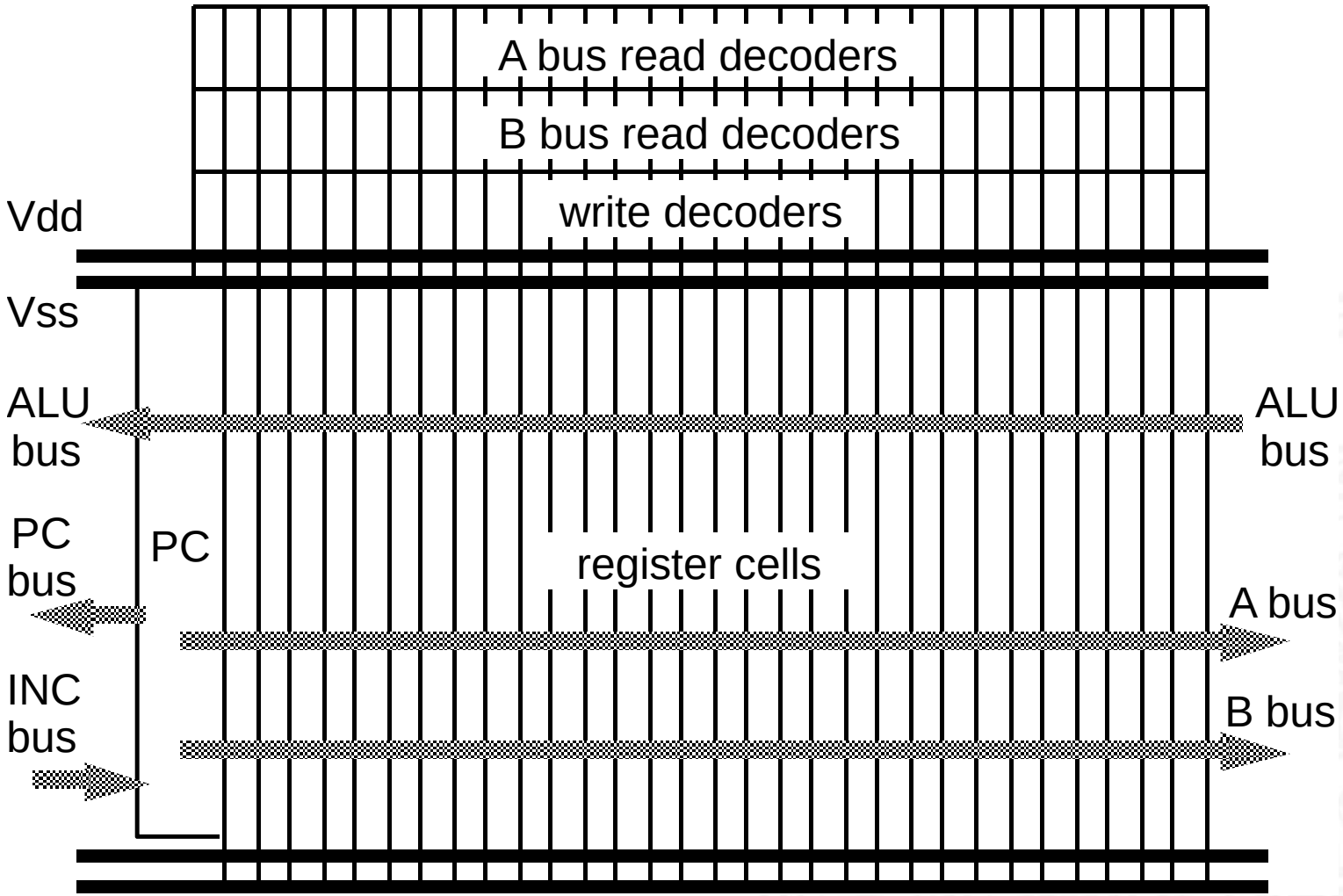
Multi-Port-Registerbank: Zelle



[Fur00]

- Prinzip wie 6T-SRAM: rückgekoppelte Inverter
- mehrere (hier zwei) parallele Lese-Ports
- mehrere Schreib-Ports möglich, aber kompliziert

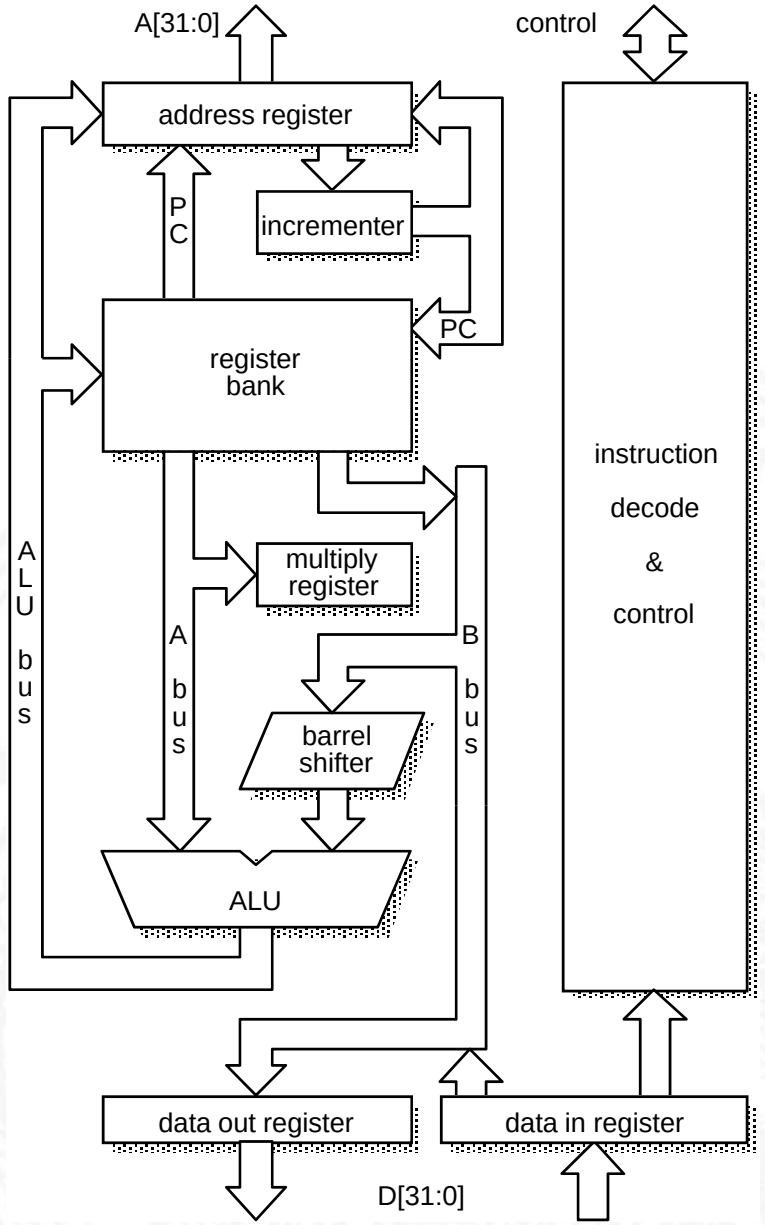
Multi-Port Registerbank: Floorplan/Chiplayout



[Fur00]

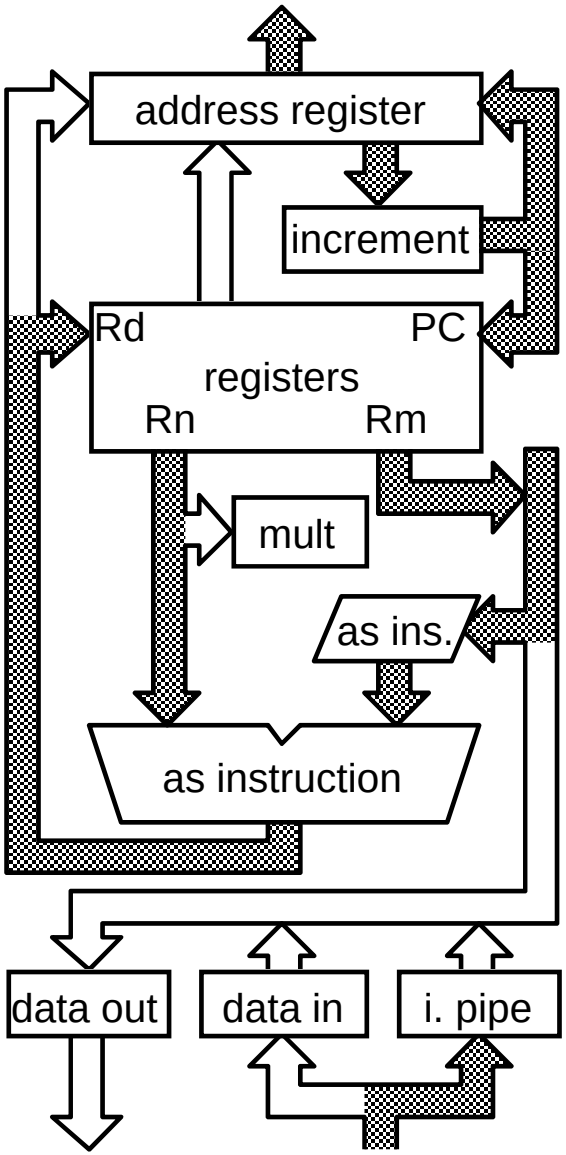
Kompletter Prozessor: ARM 3

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

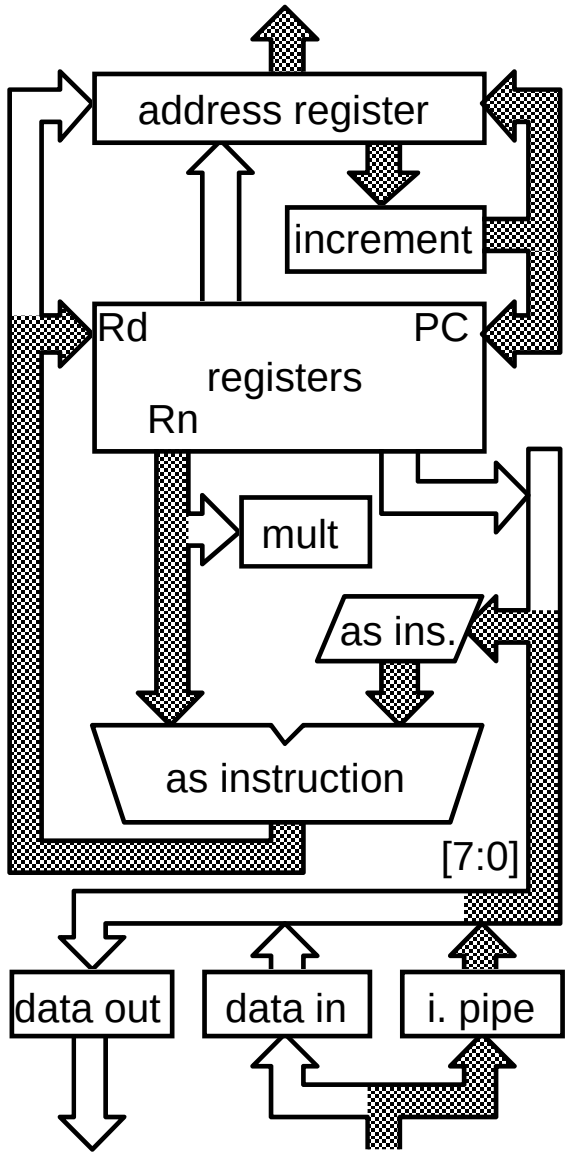


[Fur00]

ARM Datentransfer: Register-Operationen



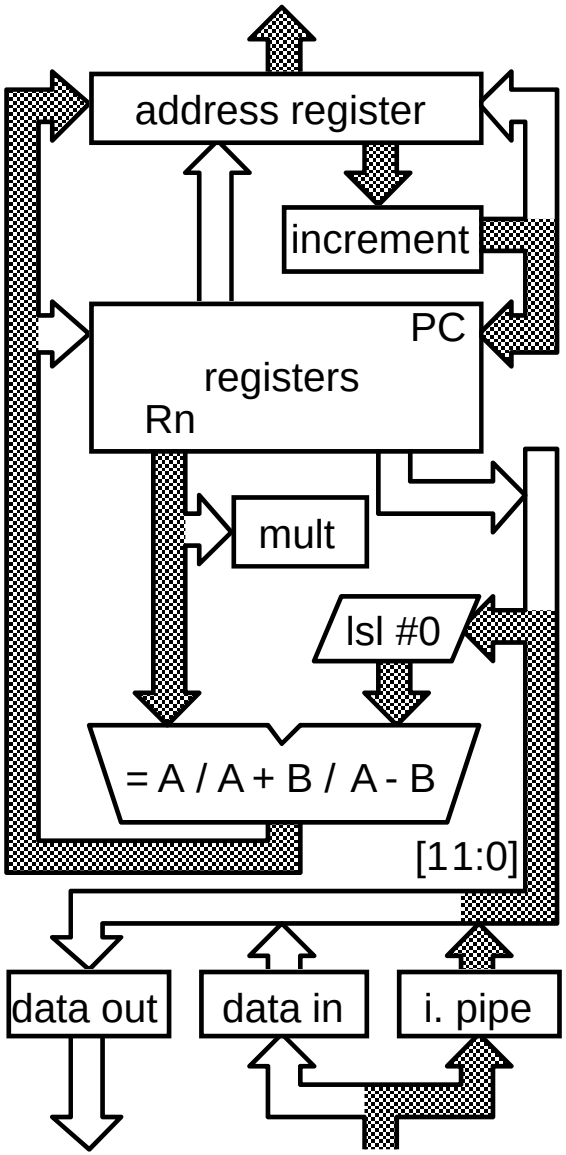
(a) register – register operations



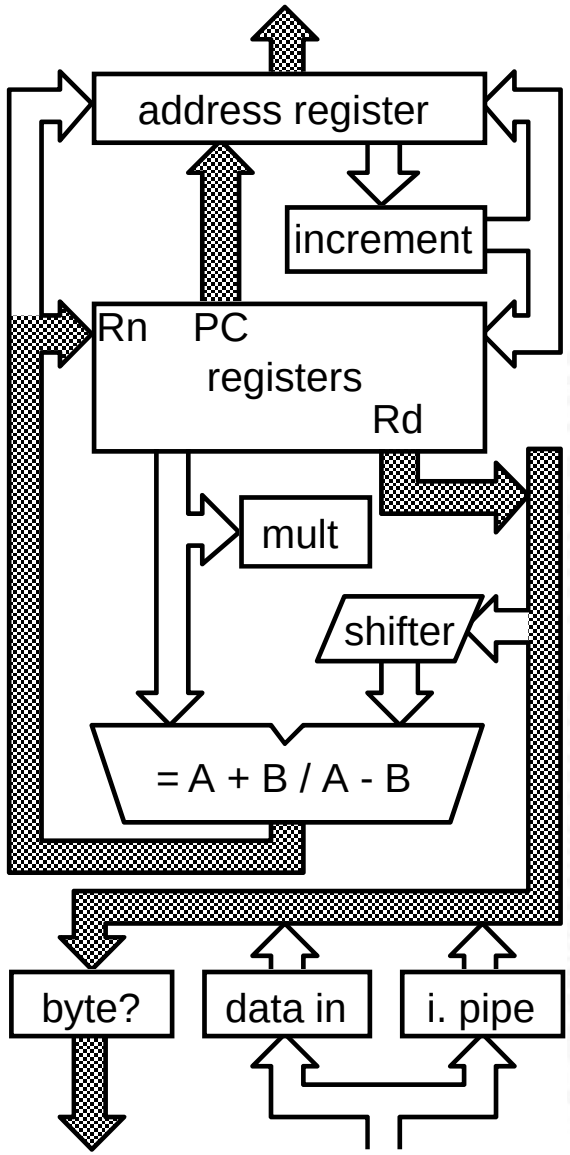
(b) register – immediate operations

[Fur00]

ARM Datentransfer: Store-Befehl



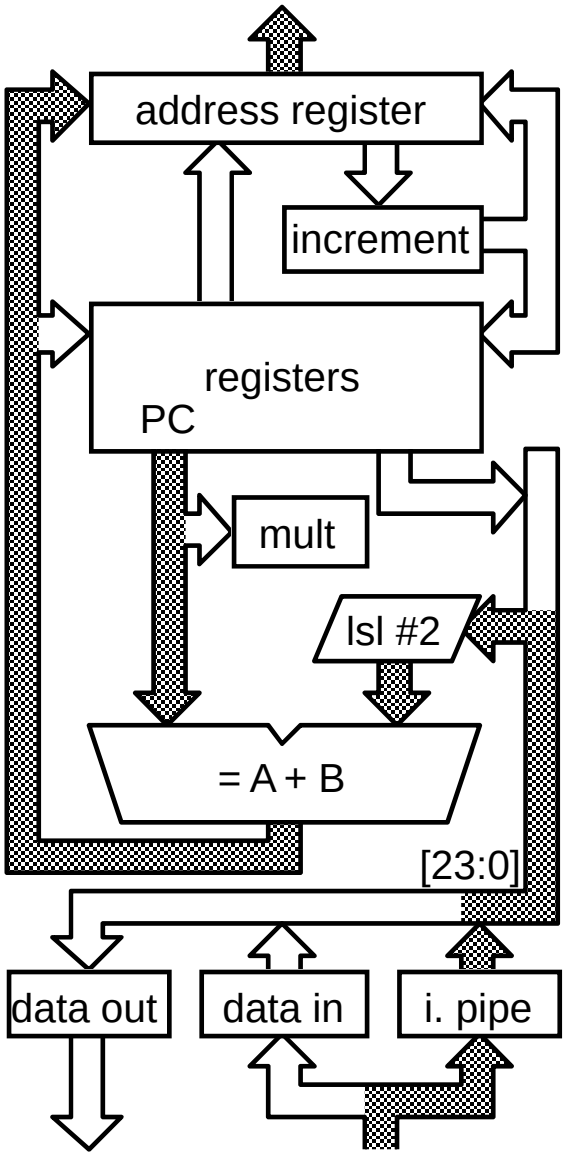
(a) 1st cycle - compute address



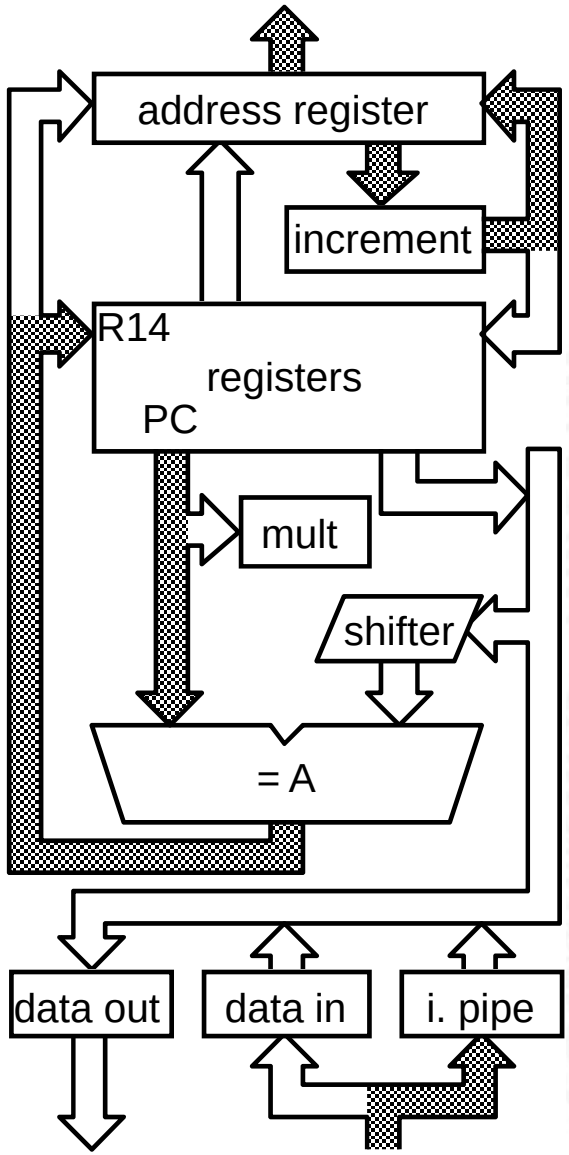
(b) 2nd cycle – store & auto-index

[Fur00]

ARM Datentransfer: Funktionsaufruf/Sprungbefehl



(a) 1st cycle - compute branch target



(b) 2nd cycle - save return address

[Fur00]

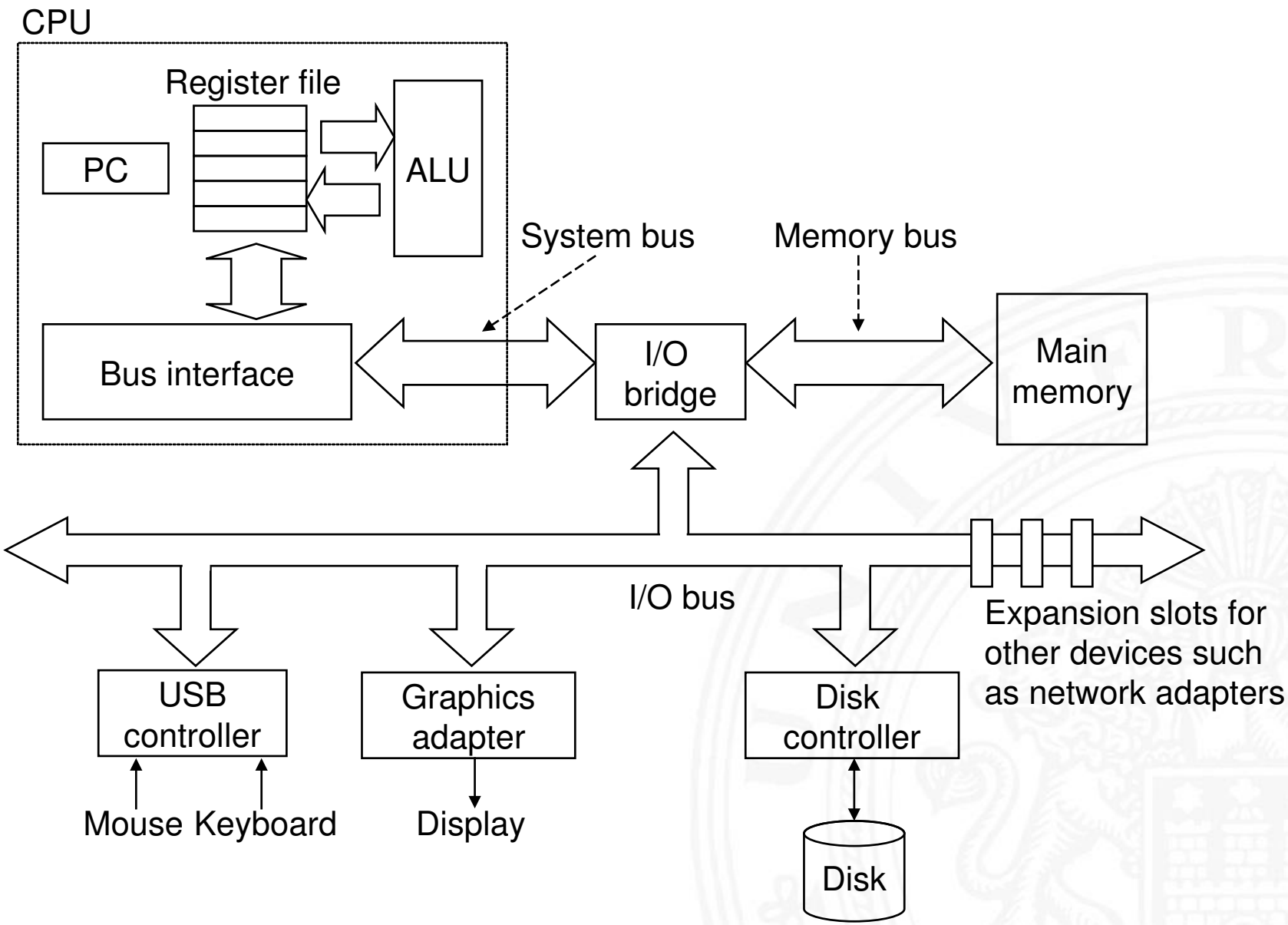
Wie rechnet ein Rechner?

- „Choreografie“ der Funktionseinheiten?
- Wie kommuniziert man mit Rechnern?
- Was passiert beim Einschalten des Rechners?

- Erweiterungen des von-Neumann Konzepts
  - parallele, statt sequenzieller Befehlsabarbeitung
    $\Rightarrow$ *Pipelining*
  - mehrere Ausführungseinheiten
    $\Rightarrow$ *superskalare Prozessoren, Mehrkern-Architekturen*
  - dynamisch veränderte Abarbeitungsreihenfolge
    $\Rightarrow$ *„out-of-order execution“*
  - getrennte Daten- und Instruktionsspeicher
    $\Rightarrow$ *Harvard-Architektur*
  - *Speicherhierarchie*, Caches etc.

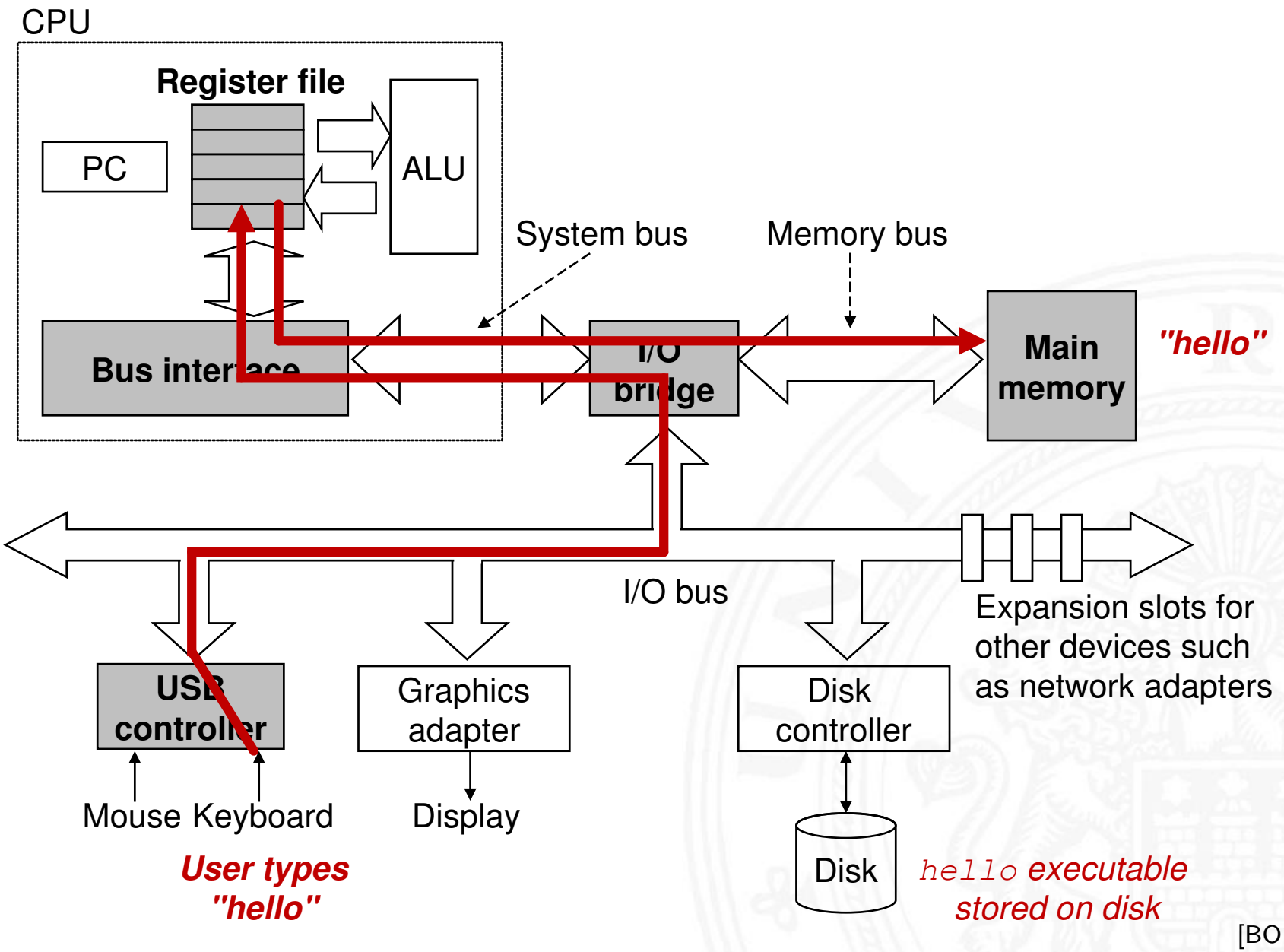
$\rightarrow$ siehe Kapitel *14 Rechnerarchitektur II*

Hardwareorganisation eines typischen Systems



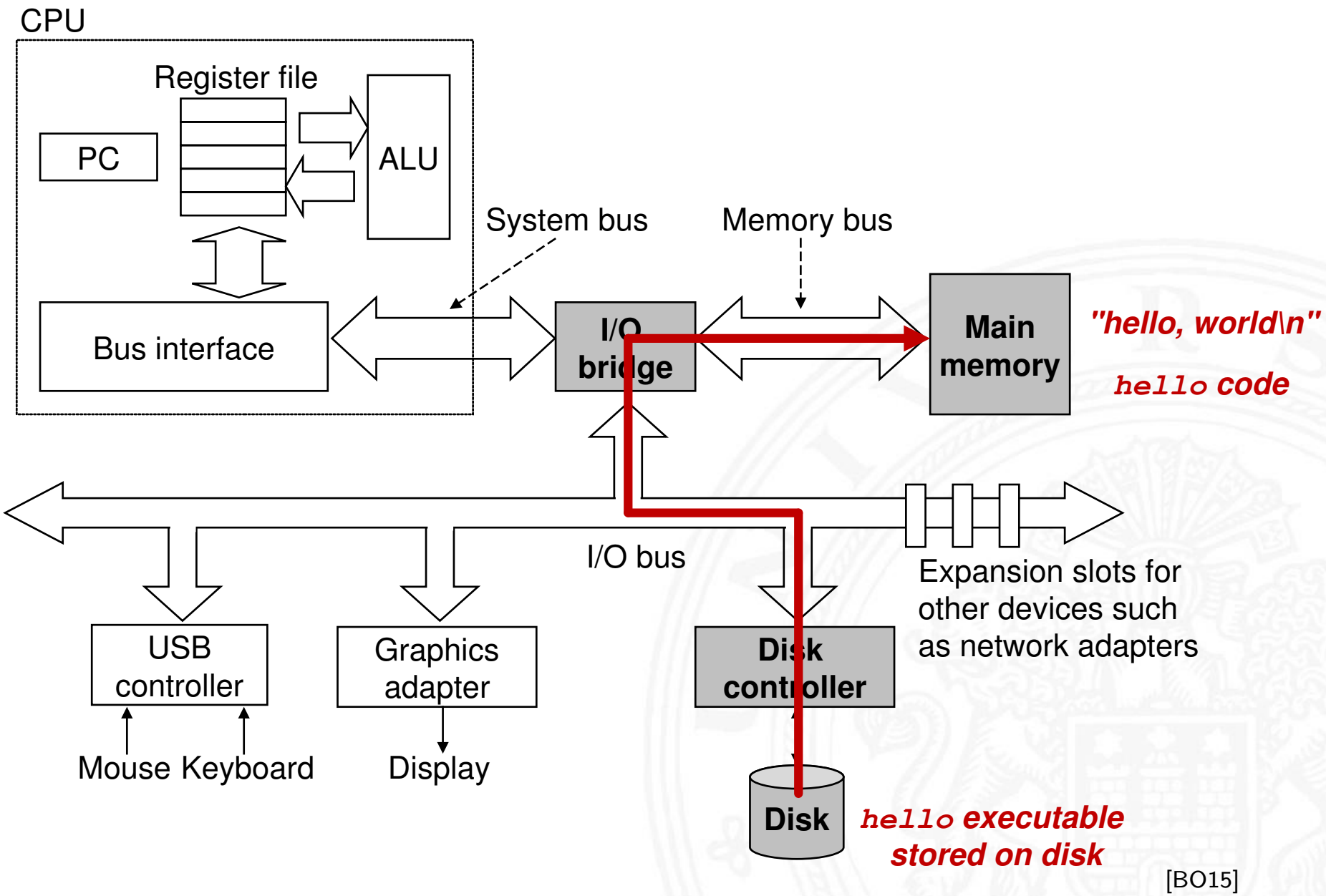
[BO15]

Programmausführung: 1. Benutzereingabe



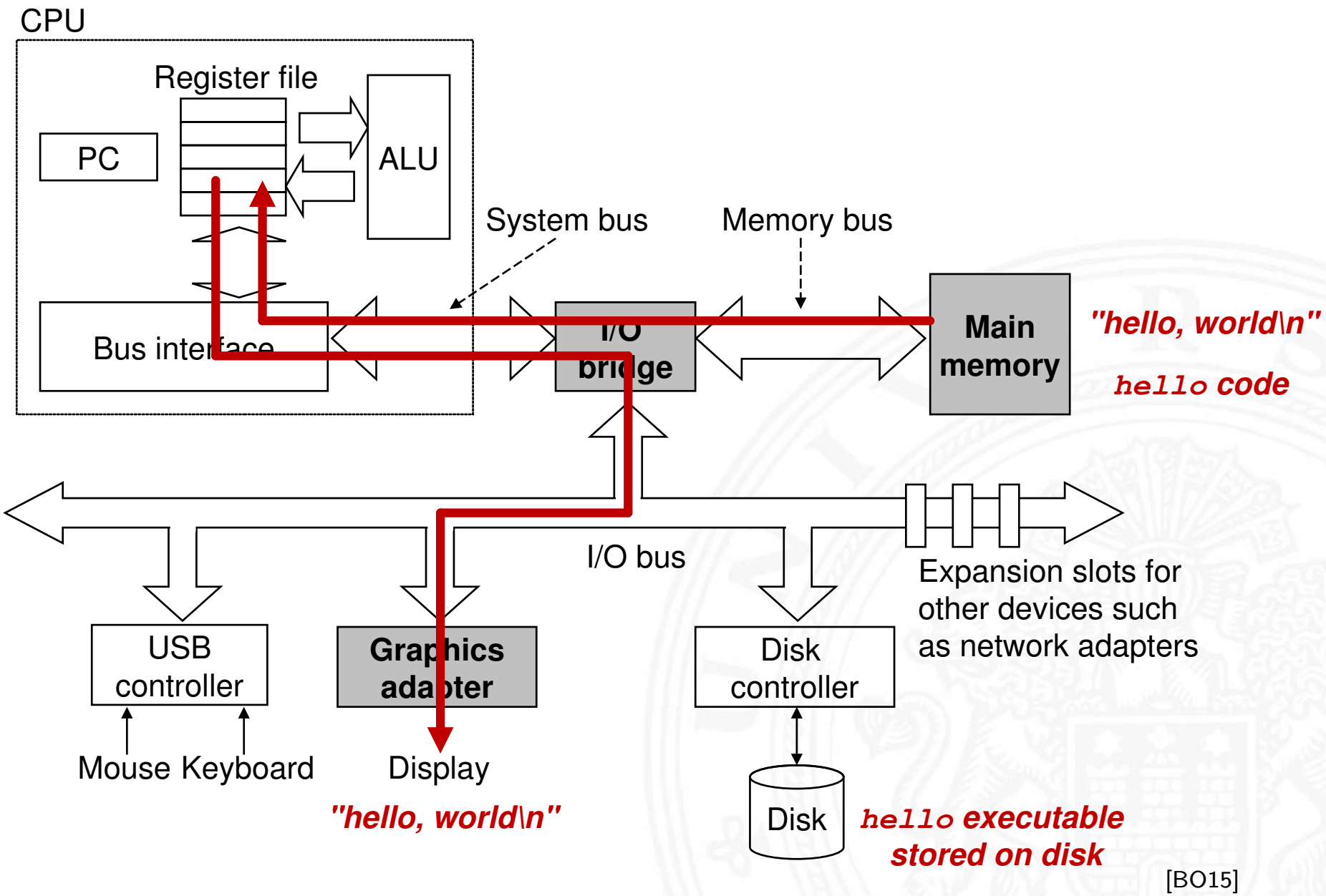
[BO15]

Programmausführung: 2. Programm laden



[BO15]

Programmausführung: 3. Programmlauf



[BO15]

Boot-Prozess

Was passiert beim Einschalten des Rechners?

- Chipsatz erzeugt Reset-Signale für alle ICs
- Reset für die zentralen Prozessor-Register (PC ...)
- PC wird auf Startwert initialisiert (z.B. 0xFFFF FFEF)
- Befehlszyklus wird gestartet
- Prozessor greift auf die Startadresse zu
  dort liegt ein ROM mit dem Boot-Programm
- Initialisierung und Selbsttest des Prozessors
- Löschen und Initialisieren der Caches
- Konfiguration des Chipsatzes
- Erkennung und Initialisierung von I/O-Komponenten
- Laden des Betriebssystems

Literatur

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

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–8689–4238–5

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

[GK83] D.D. Gajski, R.H. Kuhn: *Guest Editors' Introduction: New VLSI Tools*. in: *IEEE Computer* 16 (1983), December, Nr. 12, S. 11–14. ISSN 0018–9162

Literatur (cont.)

[PH17] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – RISC-V Edition*.
Morgan Kaufmann Publishers Inc., 2017.
ISBN 978–0–12–812275–4

[PH16b] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle*.
5. Auflage, Oldenbourg, 2016. ISBN 978–3–11–044605–0

[Mäd11] A. Mäder: *Vorlesung: Rechnerarchitektur und Mikrosystemtechnik*. Universität Hamburg,
FB Informatik, 2011, Vorlesungsfolien. `tams.informatik.uni-hamburg.de/lectures/2011ws/vorlesung/ram`

Literatur (cont.)

[HenHA] N. Hendrich: *HADES — HAmburg DEsign System*. Universität Hamburg, FB Informatik, Lehrmaterial. `tams.informatik.uni-hamburg.de/applets/hades/webdemos`

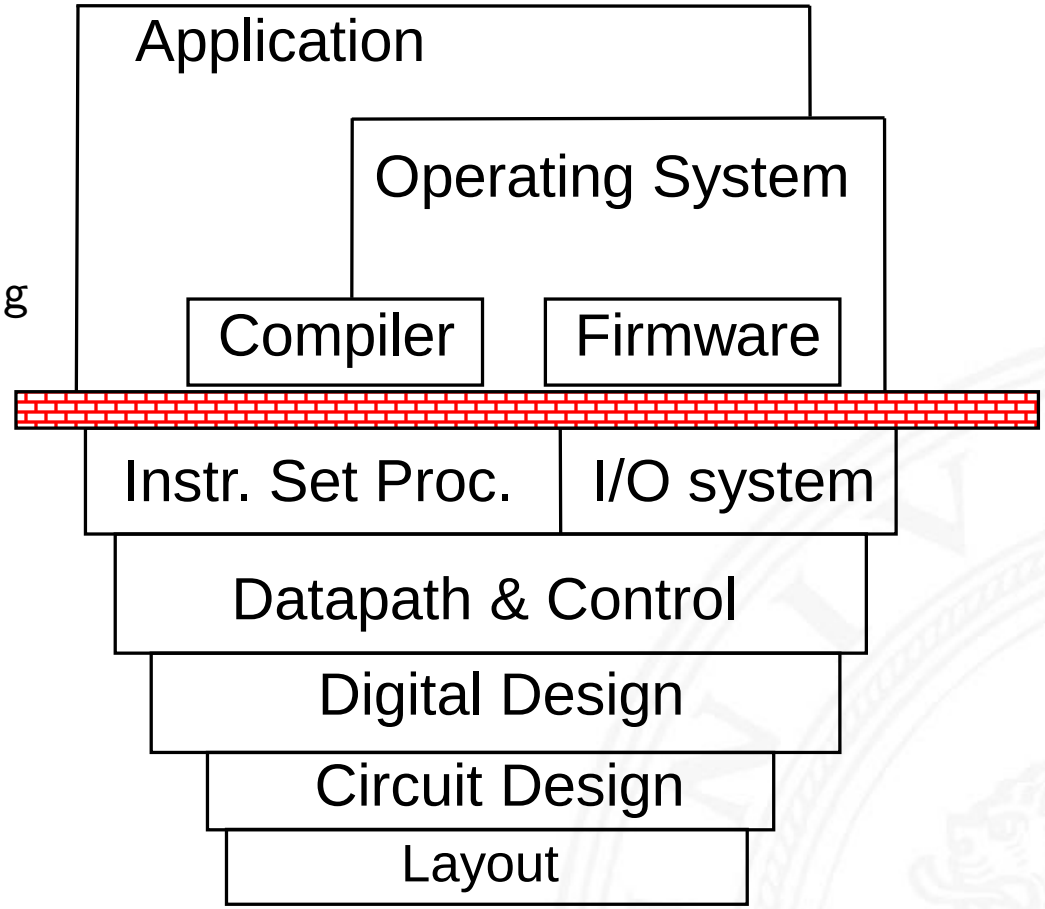
Gliederung

Gliederung (cont.)

Rechnerstrukturen und Betriebssysteme – Einordnung

Softwareentwicklung

Vorlesung RSB



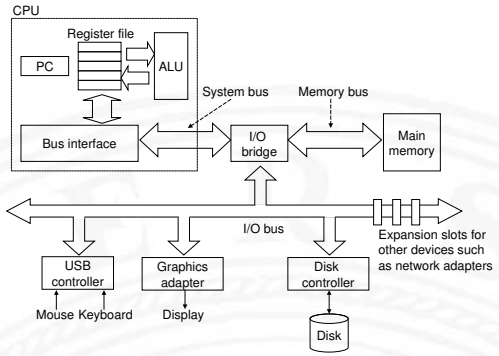
Instruction Set Architecture

[PH16b]

Befehlssatzarchitektur – ISA

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

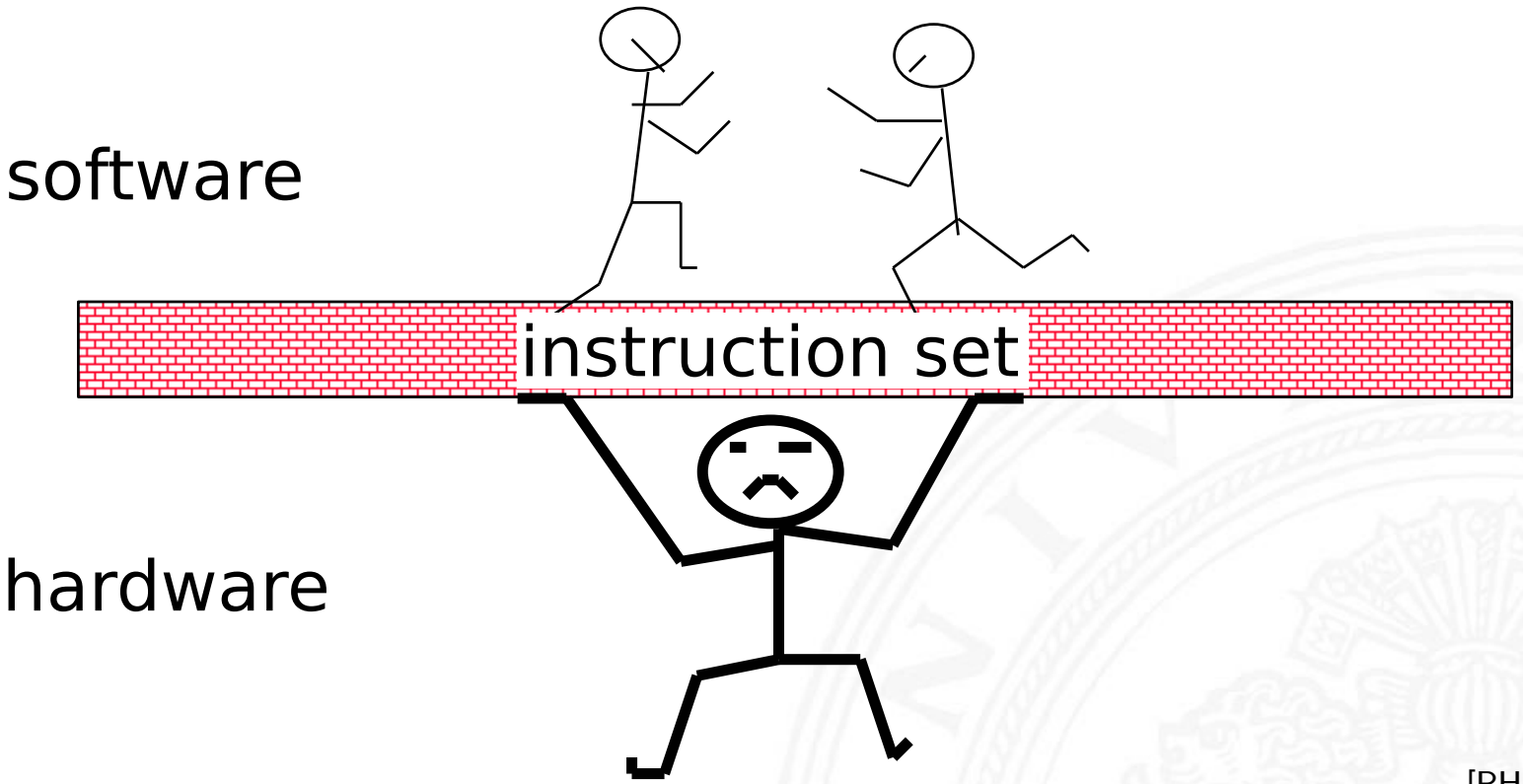
$\Rightarrow$ alle für den Programmierer sichtbaren Attribute eines Rechners



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

Befehlssatzarchitektur – ISA (cont.)

- Befehlssatz: die zentrale Schnittstelle



[PH16b]

Merkmale der Instruction Set Architecture

- Speichermodell — Wortbreite, Adressierung ...
- Rechnerklasse — Stack-/Akku-/Registermaschine
- Registersatz — Anzahl und Art der Rechenregister
- Befehlssatz — Definition aller Befehle
- Art, Zahl der Operanden — Anzahl/Wortbreite/Reg./Speicher
- Ausrichtung der Daten — Alignment/Endianness
- Ein- und Ausgabe, Unterbrechungsstruktur (Interrupts)
- Systemsoftware — Loader, Assembler, Compiler, Debugger

Beispiele für charakteristische ISA

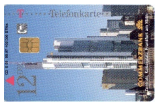
in dieser Vorlesung bzw. im Praktikum angesprochen

- MIPS — klassischer 32-bit RISC
- D·CORE — „Demo Rechner“, 16-bit
- x86 / x86-64 (x64) / amd64 — CISC, Verwendung in PCs

- Assemblerprogrammierung, Kontrollstrukturen und Datenstrukturen werden am Beispiel der x86-Architektur vorgestellt

- viele weitere Architekturen (z.B. Mikrocontroller) werden aus Zeitgründen nicht weiter behandelt

Artenvielfalt vom „Embedded Architekturen“

									
Prozessor	1 μC	1 μC	1 ASIC	1 μP, ASIP	DSPs	1 μP, 3 DSP	1 μP, DSP	$\approx$100 μC, μP, DSP	1 μP, ASIP
[bit]	4 . . . 32	8	—	16 . . . 32	32	32	32	8 . . . 64	16 . . . 32
Speicher	1 K . . . 1 M	$<$ 8 K	$<$ 1 K	1 . . . 64 M	1 . . . 64 M	$<$ 512 M	8 . . . 64 M	1 K . . . 10 M	$<$ 64 M
Netzwerk	cardIO	—	RS-232	diverse	GSM	MIDI	V.90	CAN . . .	I^2C . . .
Echtzeit	—	—	soft	soft	hard	soft	hard	hard	hard
Sicherheit	keine	mittel	keine	gering	gering	gering	gering	hoch	hoch

- riesiges Spektrum: 4 . . . 64 bit Prozessoren, DSPs, digitale/analoge ASICs . . .
- Sensoren/Aktoren: Tasten, Displays, Druck, Temperatur, Antennen, CCD . . .
- sehr unterschiedliche Anforderungen:
  Echtzeit, Sicherheit, Zuverlässigkeit, Leistungsaufnahme, Abwärme, Temperaturbereich, Störstrahlung, Größe, Kosten etc.

Speicherorganisation

- Adressierung
- Wortbreite, Speicherkapazität
- „Big Endian“ / „Little Endian“
- „Alignment“
- „Memory-Map“
- Beispiel: PC mit Windows

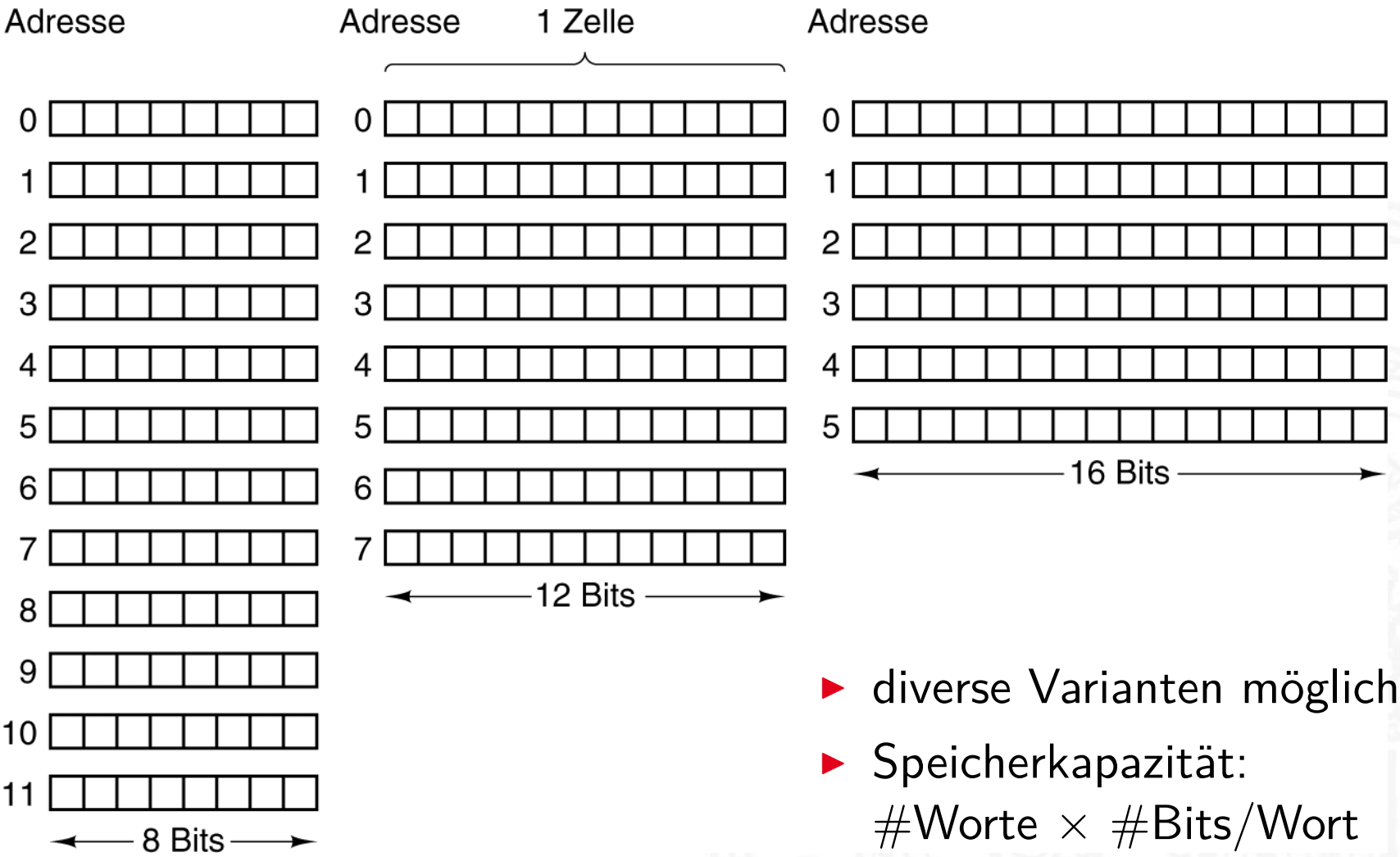
- spätere Themen
  - Cache-Organisation für schnelleren Zugriff
  - Virtueller Speicher für Multitasking
  - Synchronisation in Multiprozessorsystemen (z.B. MESI-Protokoll)

Aufbau und Adressierung des Speichers

- Abspeichern von Zahlen, Zeichen, Strings?
  - kleinster Datentyp üblicherweise ein Byte (8-bit)
  - andere Daten als Vielfache: 16-bit, 32-bit, 64-bit ...
- Organisation und Adressierung des Speichers?
  - Adressen typisch in Bytes angegeben
  - erlaubt Adressierung einzelner ASCII-Zeichen usw.
- aber Maschine/Prozessor arbeitet wortweise
- Speicher daher ebenfalls wortweise aufgebaut
- typischerweise 32-bit oder 64-bit

Hauptspeicherorganisation

3 Organisationsformen eines 96-bit Speichers: 12×8, 8×12, 6×16 Bits



[TA14]

- diverse Varianten möglich
- Speicherkapazität: #Worte $\times$ #Bits/Wort
- meist Byte-adressiert

Wortbreite

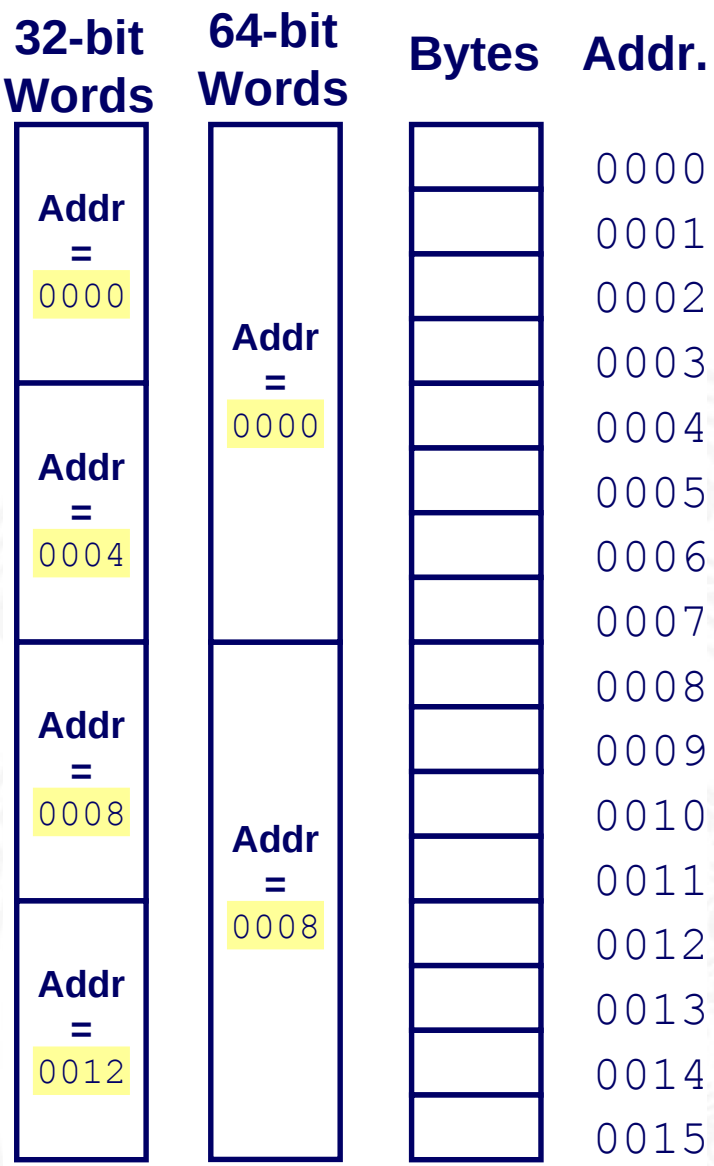
- Speicherwortbreiten historisch wichtiger Computer

Computer	Bits/Speicherzelle
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

- heute dominieren 8/16/32/64-bit Systeme
- erlaubt 8-bit ASCII, 16-bit Unicode, 32-/64-bit Floating-Point
- Beispiel x86: „byte“, „word“, „double word“, „quad word“

Wort-basierte Organisation des Speichers

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



[BO15]

Wdh. Datentypen auf Maschinenebene

- gängige Prozessoren unterstützen mehrere Datentypen
- entsprechend der elementaren Datentypen in C, Java ...
- `void*` ist ein **Pointer** (Referenz, Speicheradresse)
- Beispiel für die Anzahl der Bytes:

C Datentyp	DEC Alpha	typ. 32-bit	Intel IA-32 (x86)
int	4	4	4
long int	8	4	4
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	8	10/12
void *	8	4	4

Wdh. Datentypen auf Maschinenebene (cont.)

Abhängigkeiten (!)

- Prozessor
- Betriebssystem
- Compiler

www.agner.org/optimize/calling_conventions.pdf

segment word size	16 bit			32 bit						64 bit			
compiler	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu v.3.x	Intel Linux	Microsoft	Intel Windows	Gnu	Intel Linux
bool	2	1	1	1	1	1	1	1	1	1	1	1	1
char	1	1	1	1	1	1	1	1	1	1	1	1	1
wchar_t		2		2	2	2	2	2	2	2	2	4	4
short int	2	2	2	2	2	2	2	2	2	2	2	2	2
int	2	2	2	4	4	4	4	4	4	4	4	4	4
long int	4	4	4	4	4	4	4	4	4	4	4	8	8
__int64				8	8			8	8	8	8	8	8
enum	2	2	1	4	4	4	4	4	4	4	4	4	4
float	4	4	4	4	4	4	4	4	4	4	4	4	4
double	8	8	8	8	8	8	8	8	8	8	8	8	8
long double	10	10	8	8	16	10	8	12	12	8	16	16	16
__m64				8	8				8		8	8	8
__m128				16	16				16	16	16	16	16
__m256					32				32		32		32
pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
far pointer	4	4	4										
function pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
data member pointer (min)	2	4	6	4	4	8	4	4	4	4	4	8	8
data member pointer (max)		4	6	12	12	8	12	4	4	12	12	8	8
member function pointer (min)	2	12	6	4	4	12	4	8	8	8	8	16	16
member function pointer (max)		12	6	16	16	12	16	8	8	24	24	16	16

Table 1 shows how many bytes of storage various objects use for different compilers.

Byte-Order

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

Zwei Möglichkeiten / Konventionen:

- **Big Endian**: Sun, Mac usw.

  das MSB (*most significant byte*) hat die kleinste Adresse

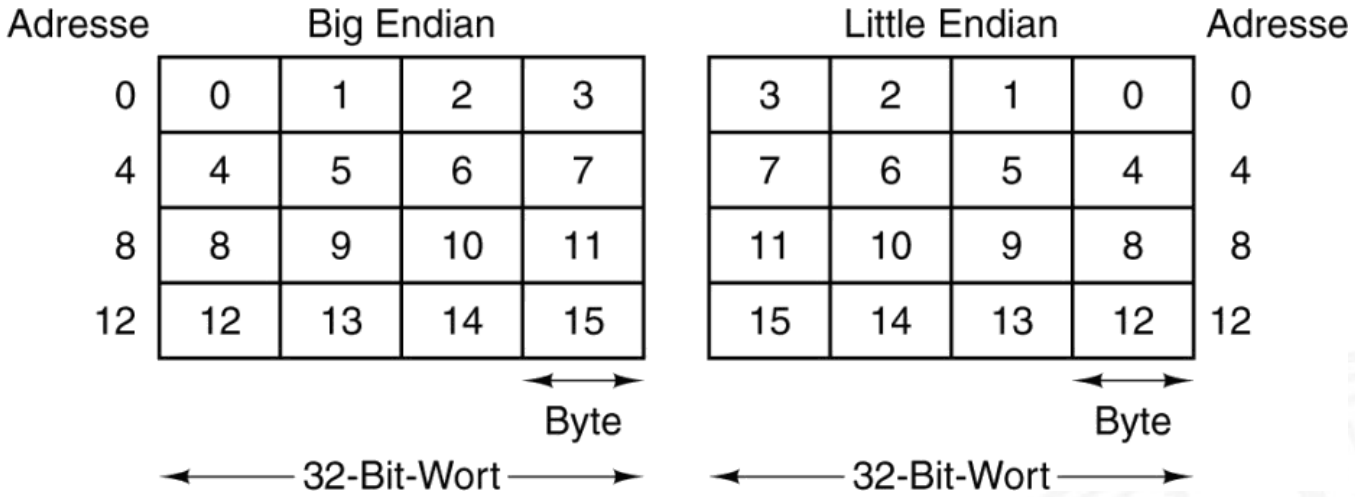
  das LSB (*least significant byte*) hat die höchste –"–

- **Little Endian**: Alpha, x86

  das MSB hat die höchste, das LSB die kleinste Adresse

satirische Referenz auf Gulliver's Reisen (Jonathan Swift)

Big- vs. Little Endian



[TA14]

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

Byte-Order: Beispiel

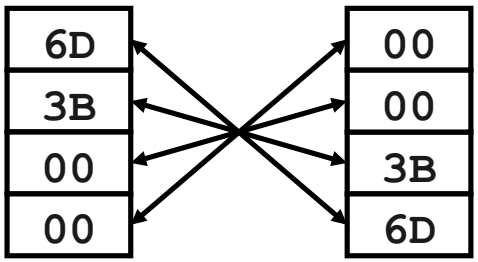
```
      int A =  15213;
      int B = -15213;
long int C =  15213;
```

Dezimal: 15213

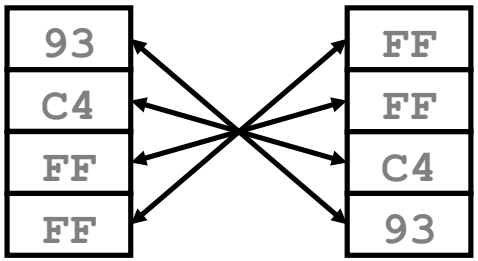
Binär: 0011 1011 0110 1101

Hex: 3 B 6 D

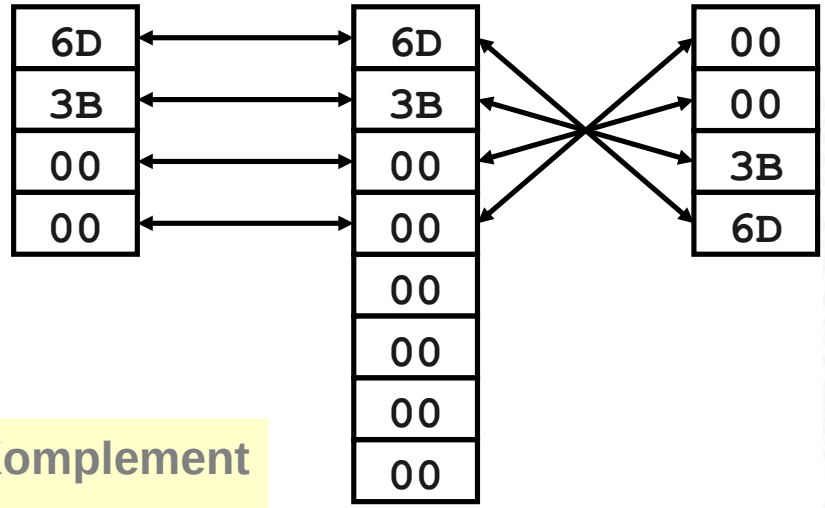
Linux/Alpha A **Sun A**



Linux/Alpha B **Sun B**



Linux C **Alpha C** **Sun C**



2-Komplement

Big Endian

Little Endian

[BO15]

Byte-Order: Beispiel Datenstruktur


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

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

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

Byte-Order: Beispiel x86 und SPARC


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

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

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

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

Netzwerk Byte-Order

- Byte-Order muss bei Datenübertragung zwischen Rechnern berücksichtigt und eingehalten werden

- Internet-Protokoll (IP) nutzt ein Big Endian Format

$\Rightarrow$ auf x86-Rechnern müssen alle ausgehenden und ankommenden Datenpakete umgewandelt werden

- zugehörige Hilfsfunktionen / Makros in `netinet/in.h`
  - inaktiv auf Big Endian, **byte-swapping** auf Little Endian
  - `ntohl(x)`: network-to-host-long
  - `htons(x)`: host-to-network-short
  - ...

Beispiel: Byte-Swapping *network to/from host*

Linux: /usr/include/bits/byteswap.h (distributionsabhängig)

```
...
/* Swap bytes in 32 bit value.  */
#define __bswap_32(x) \
  ((((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >>  8) |\
   (((x) & 0x0000ff00) <<  8) | (((x) & 0x000000ff) << 24))
...
```

Linux: /usr/include/netinet/in.h

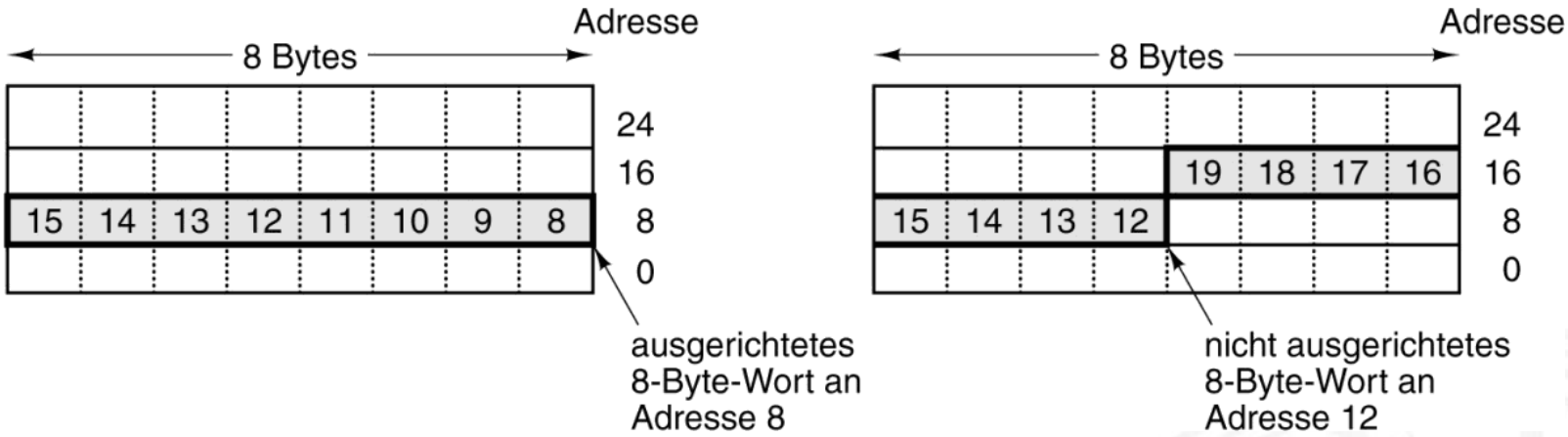
```
...
#  if __BYTE_ORDER == __LITTLE_ENDIAN
#   define ntohl(x) __bswap_32 (x)
#   define ntohs(x) __bswap_16 (x)
#   define htonl(x) __bswap_32 (x)
#   define htons(x) __bswap_16 (x)
#  endif
...
```

Programm zum Erkennen der Byte-Order

- Programm gibt Daten byteweise aus
- C-spezifische Typ- (Pointer-) Konvertierung
- Details: Bryant, O'Hallaron: 2.1.4 (Abb. 2.3, 2.4) [BO15]

```
void show_bytes( byte_pointer start, int len ) {
  int i;
  for( i=0; i < len; i++ ) {
    printf( " %.2x", start[i] );
  }
  printf( "\n" );
}

void show_double( double x ) {
  show_bytes( (byte_pointer) &x, sizeof( double ));
}
  ...
```

„Misaligned“ Zugriff



[TA14]

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

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

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

Memory Map

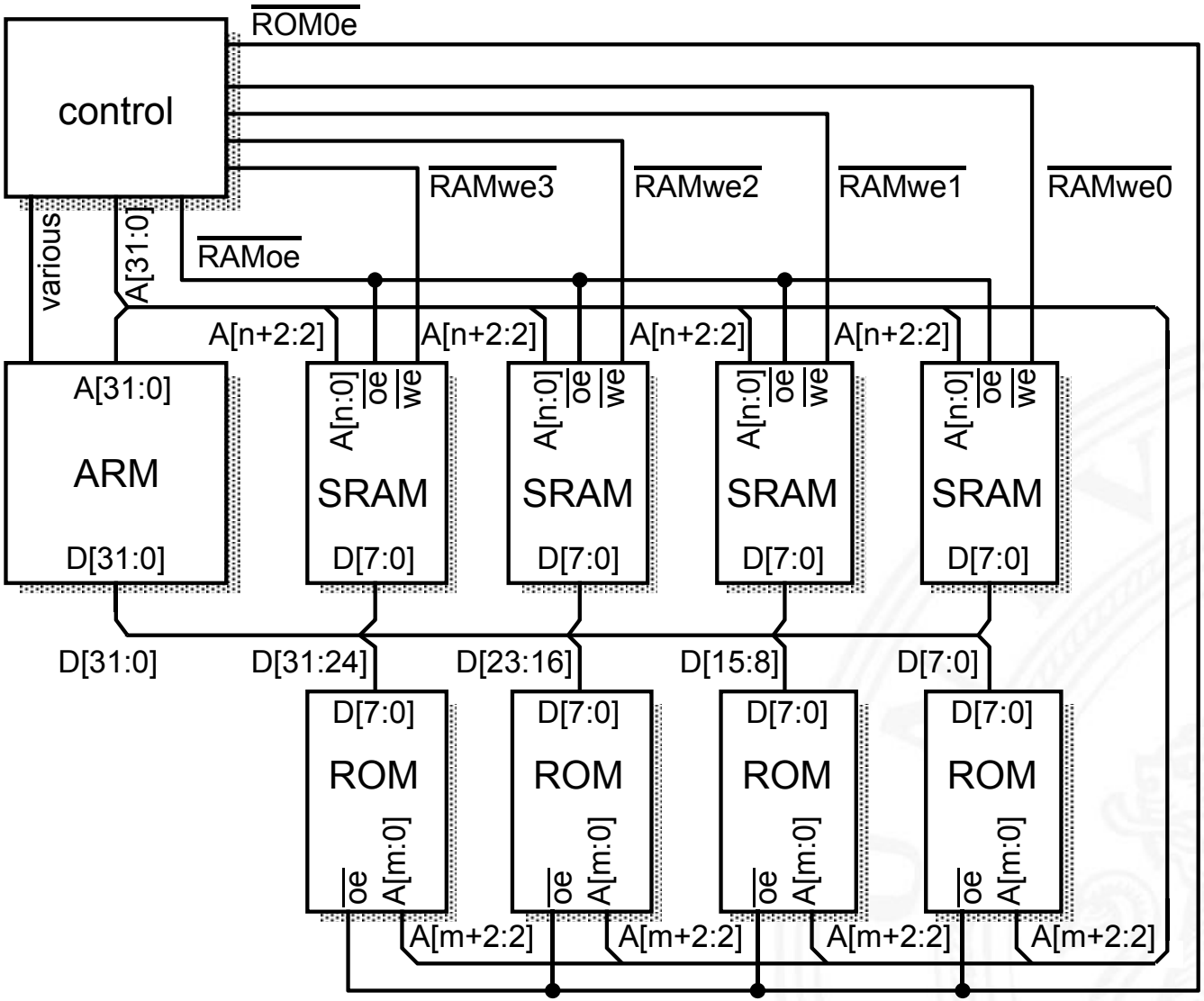
- CPU kann im Prinzip alle möglichen Adressen ansprechen
- in der Regel: kein voll ausgebauter Speicher
  32 bit Adresse entsprechen 4 GiB Hauptspeicher, 64 bit ...

$\Rightarrow$ „Memory Map“

- Adressdecoder als Hardwareeinheit
  - Aufteilung in *read-write*- und *read-only*-Bereiche
  - ROM zum Booten notwendig
  - Read-only in *eingebetteten Systemen*: Firmware, OS, Programme
  - zusätzliche Speicherbereiche für „memory mapped“ I/O

- Adressabbildung in Betriebssystemen (Windows, Linux etc.)
  - Zuordnung von Adressen zu „realem“ Speicher
  - alle Hardwarekomponenten (+ Erweiterungskarten)
    Ein-/Ausgabekanäle
    Interrupts
  - Verwaltung über *Treiber*

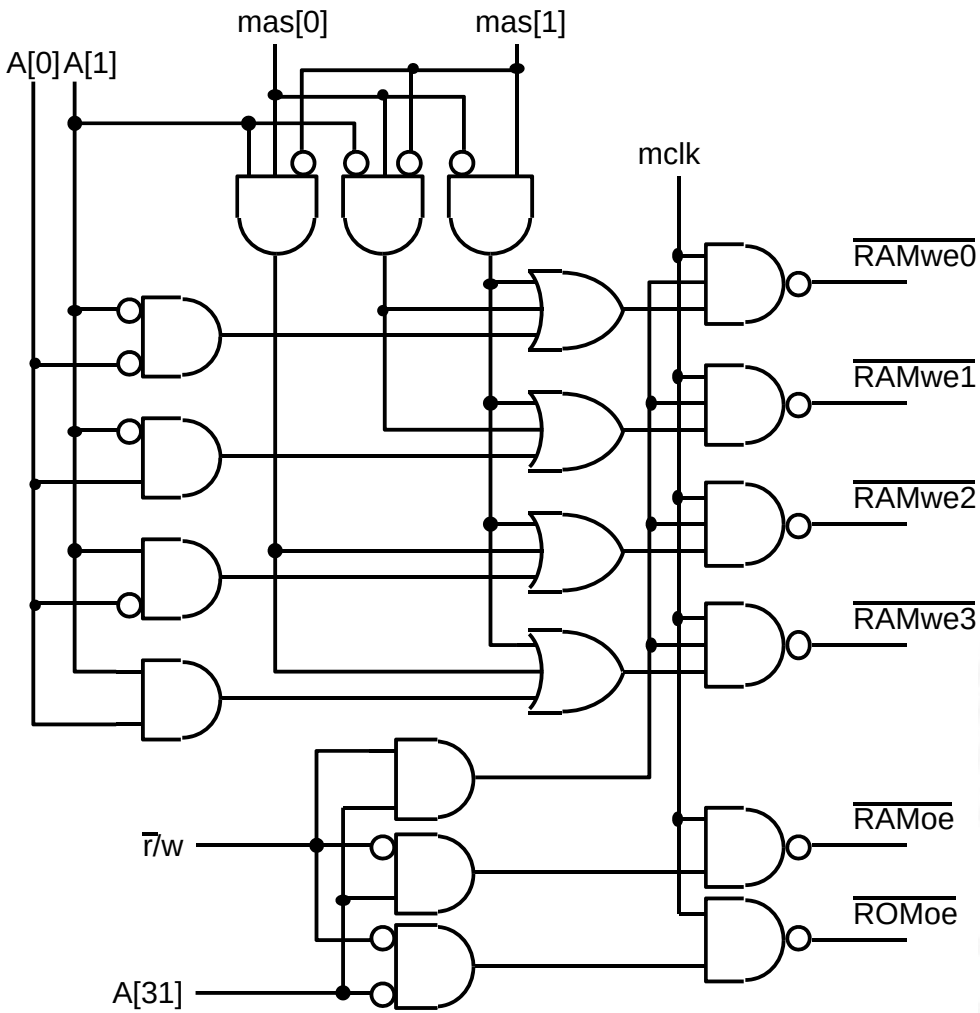
Adressabbildung Hardware: ARM



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

[Fur00]

Adressabbildung Hardware: ARM (cont.)

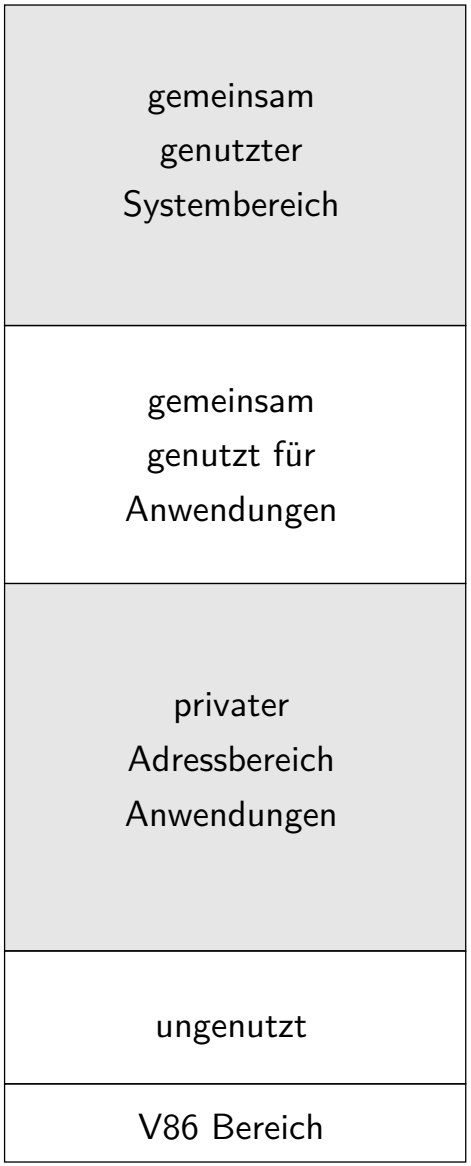


Adressdecoder Hardware

[Fur00]

Adressabbildung in Betriebssystemen

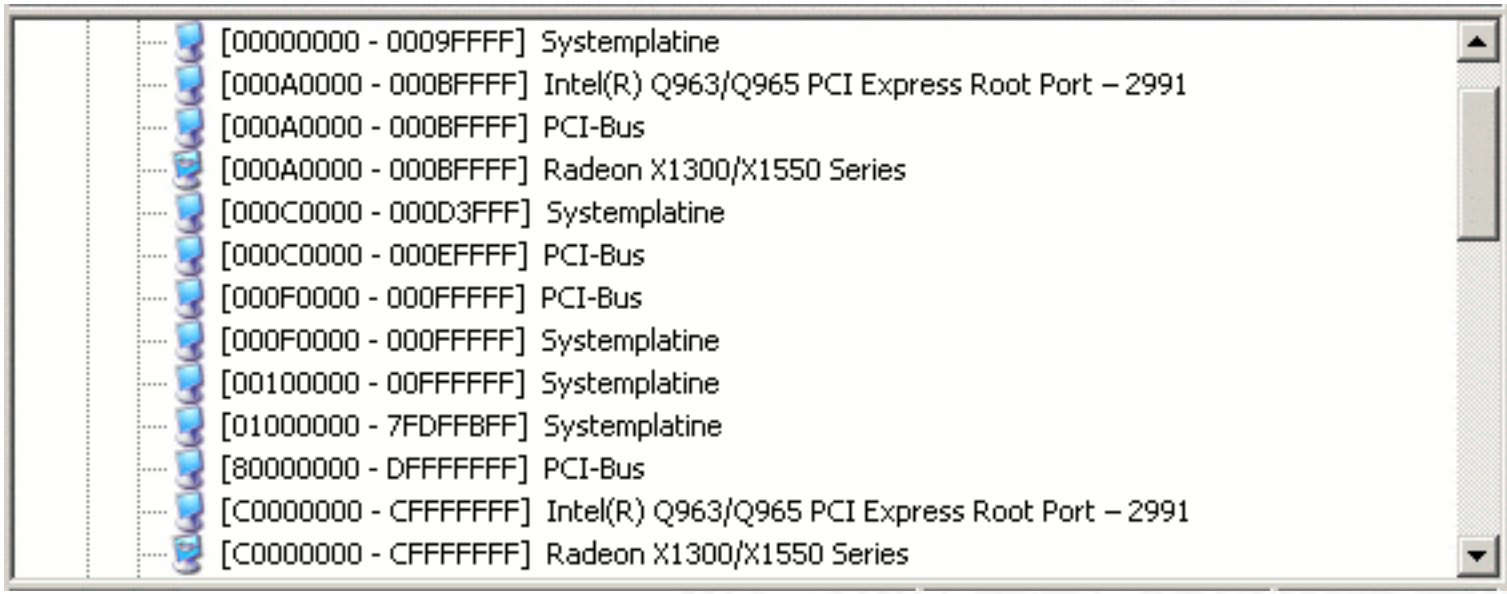
- Windows 95 (32-bit System)



- Kernel, Treiber usw. im oberen 1 GiB-Bereich
- Systembibliotheken (DLLs)
- 2 GiB für Anwendungen
- DOS-Bereich (real-mode) immer noch für Boot + Geräte „untere“ 20-bit

Adressabbildung in Betriebssystemen (cont.)

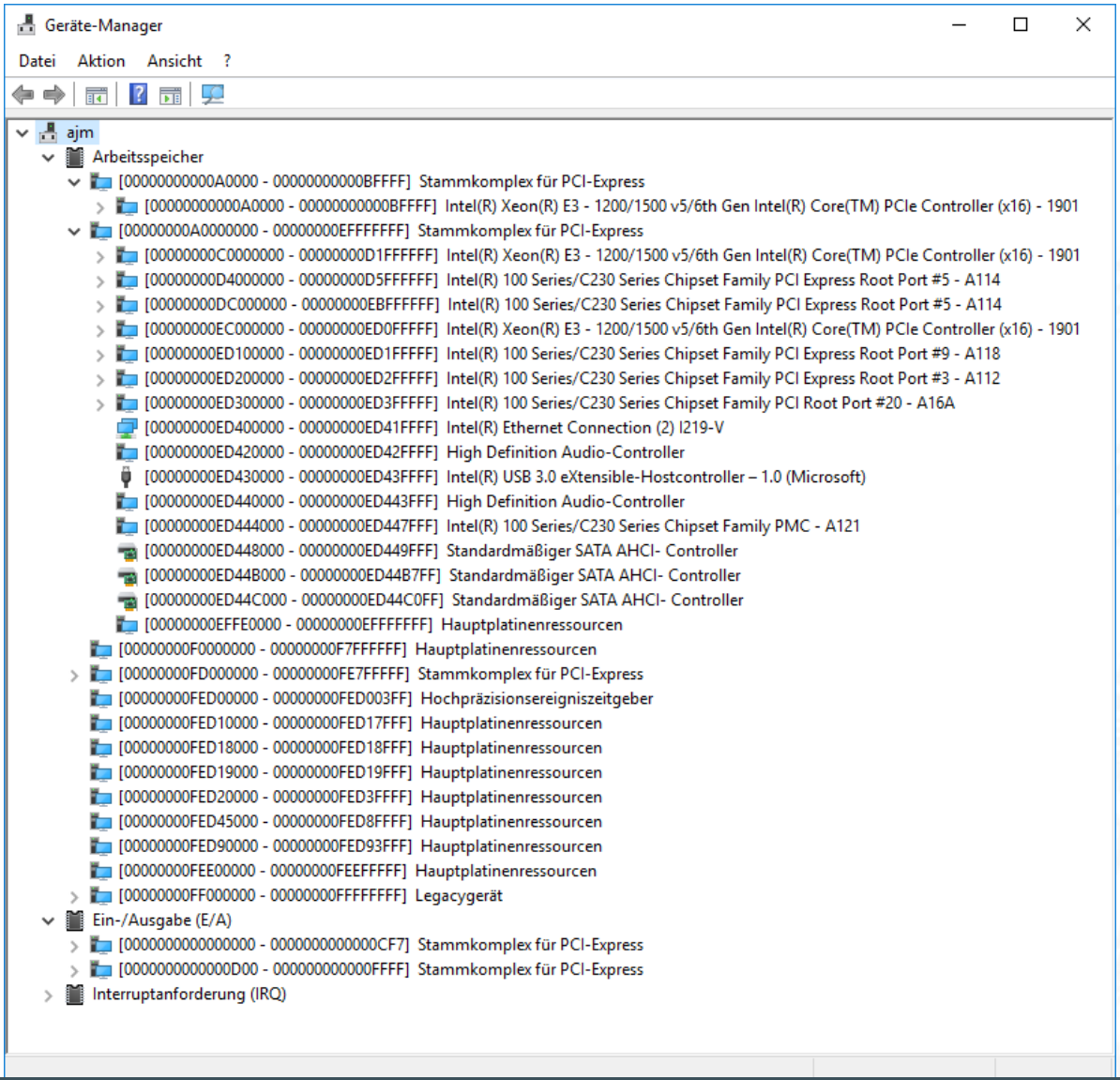
- Windows 95 (32-bit System)
  - 32-bit Adressen, 4 GiByte Adressraum
  - Aufteilung 2 GiB für Programme, obere 1+1 GiB für Windows
  - unabhängig von physikalischem Speicher
  - Beispiel der Zuordnung, diverse Bereiche für I/O reserviert



  - x86 I/O-Adressraum gesamt nur 64 KiByte
  - je nach Zahl der I/O-Geräte evtl. fast voll ausgenutzt
  - Adressen vom BIOS zugeteilt

Adressabbildung in Betriebssystemen (cont.)

- Windows 10 (64-bit System)

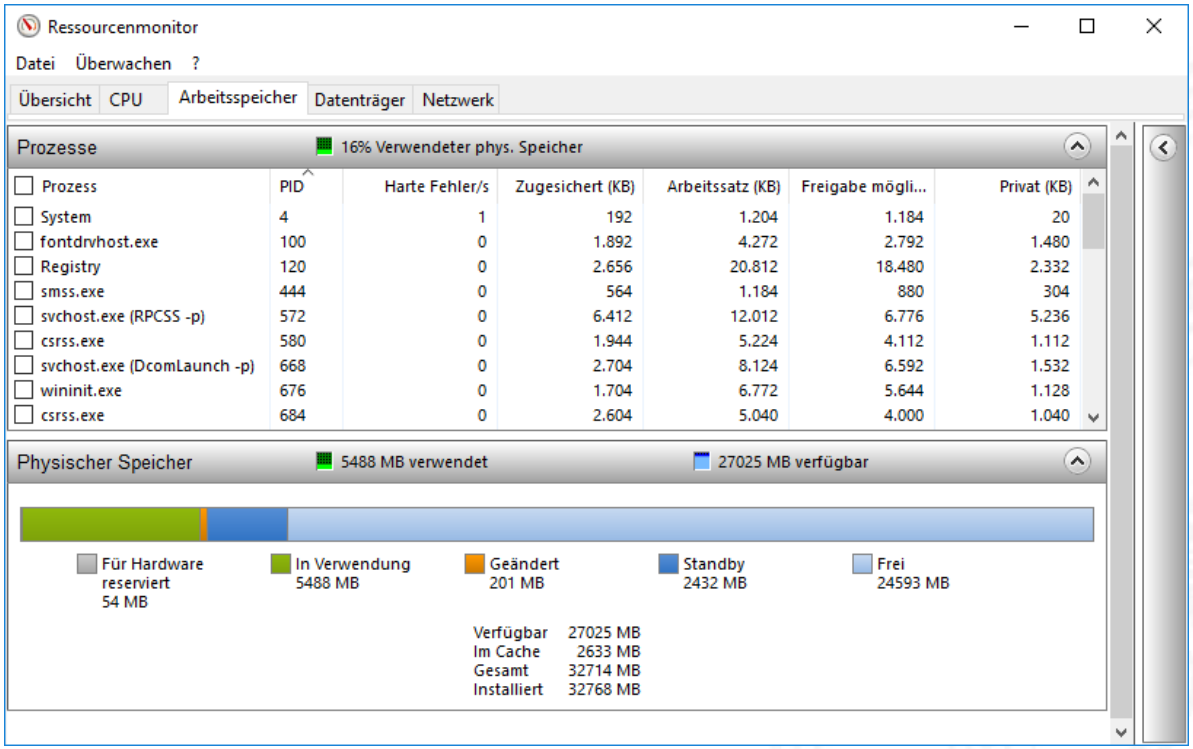


⇒ Adressbereich

Adressabbildung in Betriebssystemen (cont.)

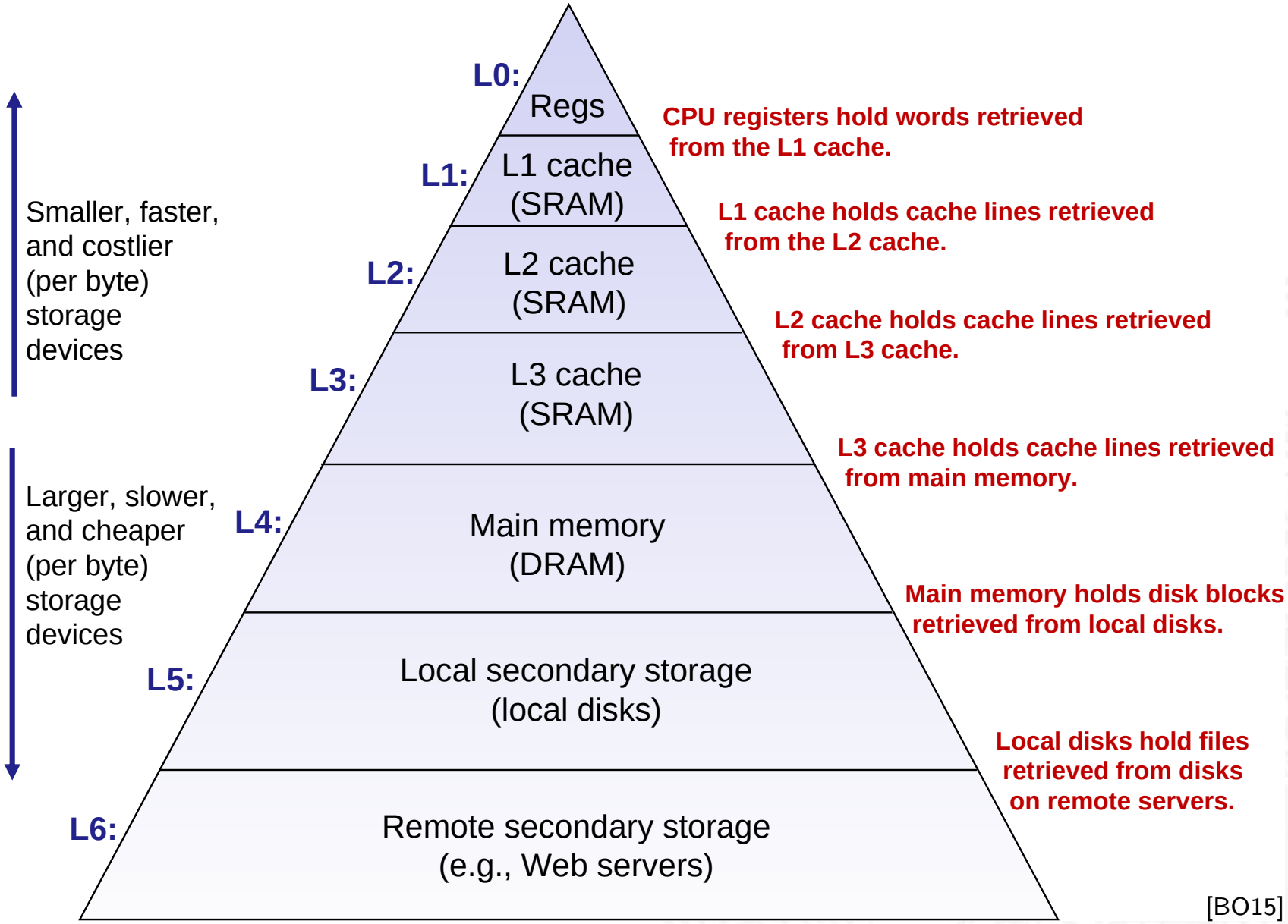
mehrstufige Abbildung:

1. alle Hardwarekomponenten, Ein-/Ausgabeeinheiten und Interrupts $\Rightarrow$ Adressbereich
2. Adressbereich $\Rightarrow$ physikalischer Speicher



- Linux ausprobieren: `lshw`, `kinfocenter`, `sysinfo` etc.

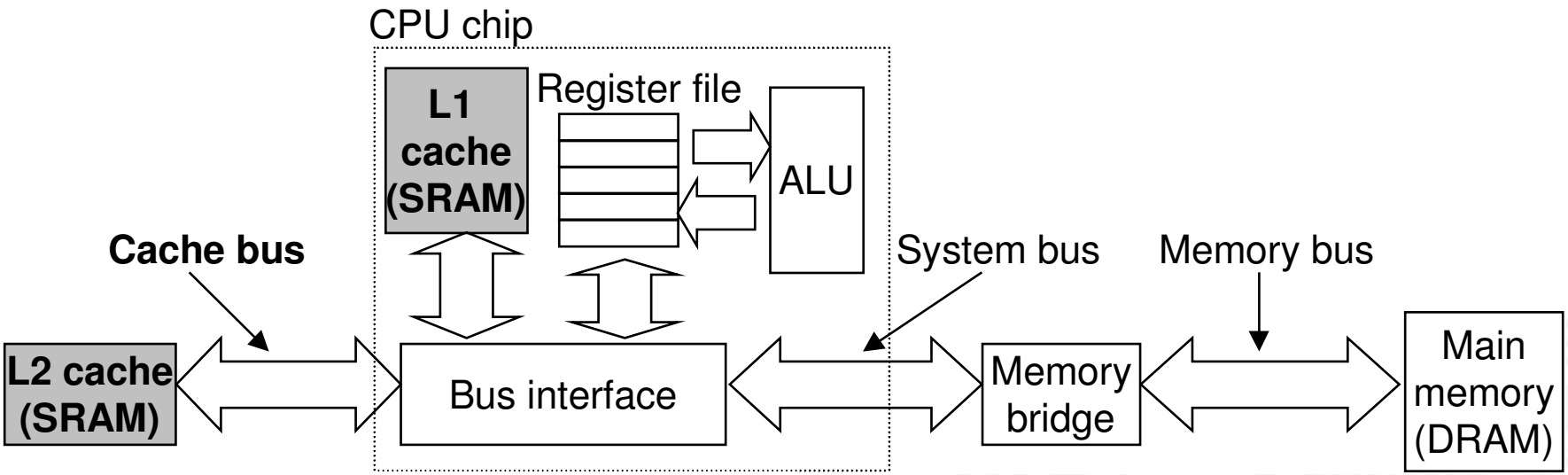
Speicherhierarchie



[BO15]

später mehr . . .

Speicherhierarchie: Cache



[BO15]

- schneller Zwischenspeicher: überbrückt Geschwindigkeits-unterschied zwischen CPU und Hauptspeicher

- Cache Strategien
  - Welche Daten sollen in Cache?
  - Welche werden aus (vollem) Cache entfernt?
- Cache Abbildung: direct-mapped, n-fach assoz., voll assoziativ
- Cache Organisation: Größe, Wortbreite etc.

Speicher ist wichtig!

- Speicher ist nicht unbegrenzt
  - muss zugeteilt und verwaltet werden
  - viele Anwendungen werden vom Speicher dominiert
- besondere Sorgfalt beim Umgang mit Speicher
  - Fehler sind besonders gefährlich und schwer zu Debuggen
  - Auswirkungen sind sowohl zeitlich als auch räumlich entfernt
- Speicherleistung ist nicht gleichbleibend

  Wechselwirkungen: Speichersystem $\Leftrightarrow$ Programme
  - „Cache“- und „Virtual“-Memory Auswirkungen können Performanz/Programmleistung stark beeinflussen
  - Anpassung des Programms an das Speichersystem kann Geschwindigkeit bedeutend verbessern

$\rightarrow$ siehe *14 Rechnerarchitektur II – Speicherhierarchie*

ISA-Merkmale des Prozessors

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

Befehlszyklus

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

Steuerwerk		
PC	Program Counter	Adresse des Befehls
IR	Instruction Register	aktueller Befehl

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

Instruction Fetch

„Befehl holen“ Phase im Befehlszyklus

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

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

Instruction Decode

„Befehl decodieren“ Phase im Befehlszyklus

- ▷ Befehl steht im Befehlsregister IR

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

Operand Fetch

- wird meist zu anderen Phasen hinzugezählt

  RISC: Teil von *Instruction Decode*

  CISC: –"– *Instruction Execute*

1. Operanden holen

Instruction Execute

„Befehl ausführen“ Phase im Befehlszyklus

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

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

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

Welche Befehle braucht man?

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

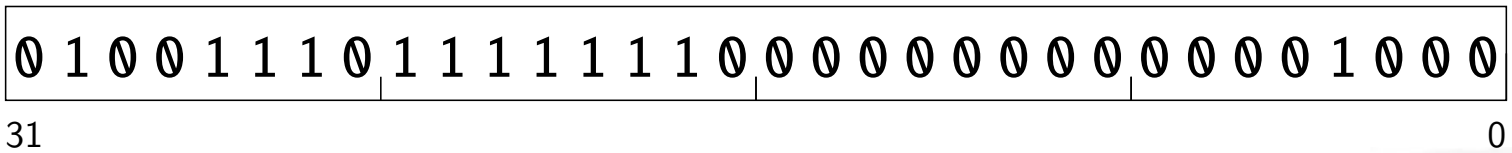
⇒ Befehlssätze und Computerarchitekturen (Details später)

CISC – Complex Instruction Set Computer

RISC – Reduced Instruction Set Computer

Befehls-Decodierung

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



Wie soll die Hardware diesen Wert interpretieren?

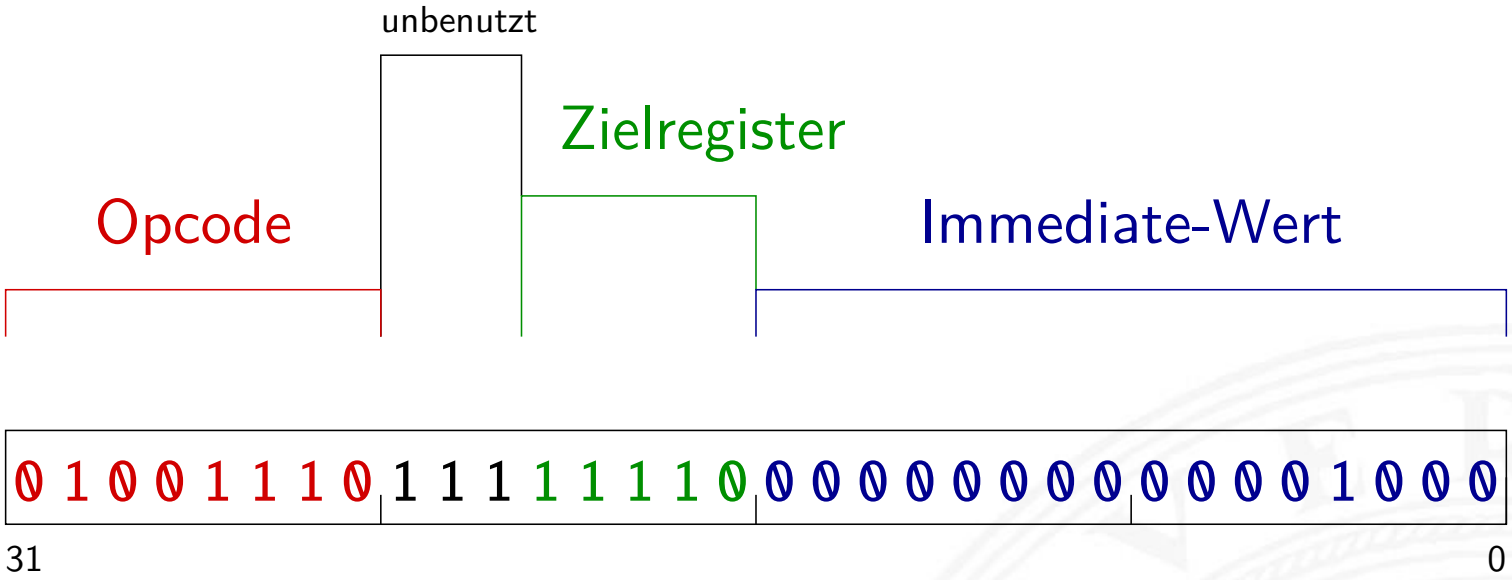
- direkt in einer Tabelle nachschauen (Mikrocode-ROM)
- Problem: Tabelle müsste 2^{32} Einträge haben

$\Rightarrow$ Aufteilung in Felder: Opcode und Operanden

$\Rightarrow$ Decodierung über mehrere, kleine Tabellen

$\Rightarrow$ unterschiedliche Aufteilung für unterschiedliche Befehle: Befehlsformate

Befehlsformate



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

Befehlsformat: drei Beispielarchitekturen

- MIPS: Beispiel für 32-bit RISC Architekturen
  - alle Befehle mit 32-bit codiert
  - nur 3 Befehlsformate (R, I, J)
- D·CORE: Beispiel für 16-bit Architektur
  - siehe Praktikum RSB (64-042) für Details
- Intel x86: Beispiel für CISC-Architekturen
  - irreguläre Struktur, viele Formate
  - mehrere Codierungen für einen Befehl
  - 1-Byte . . . 36-Bytes pro Befehl

Befehlsformat: Beispiel MIPS

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

wenige Befehlsformate

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

MIPS: Übersicht

„Microprocessor without Interlocked Pipeline Stages“

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

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

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

MIPS: Registermodell

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

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

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

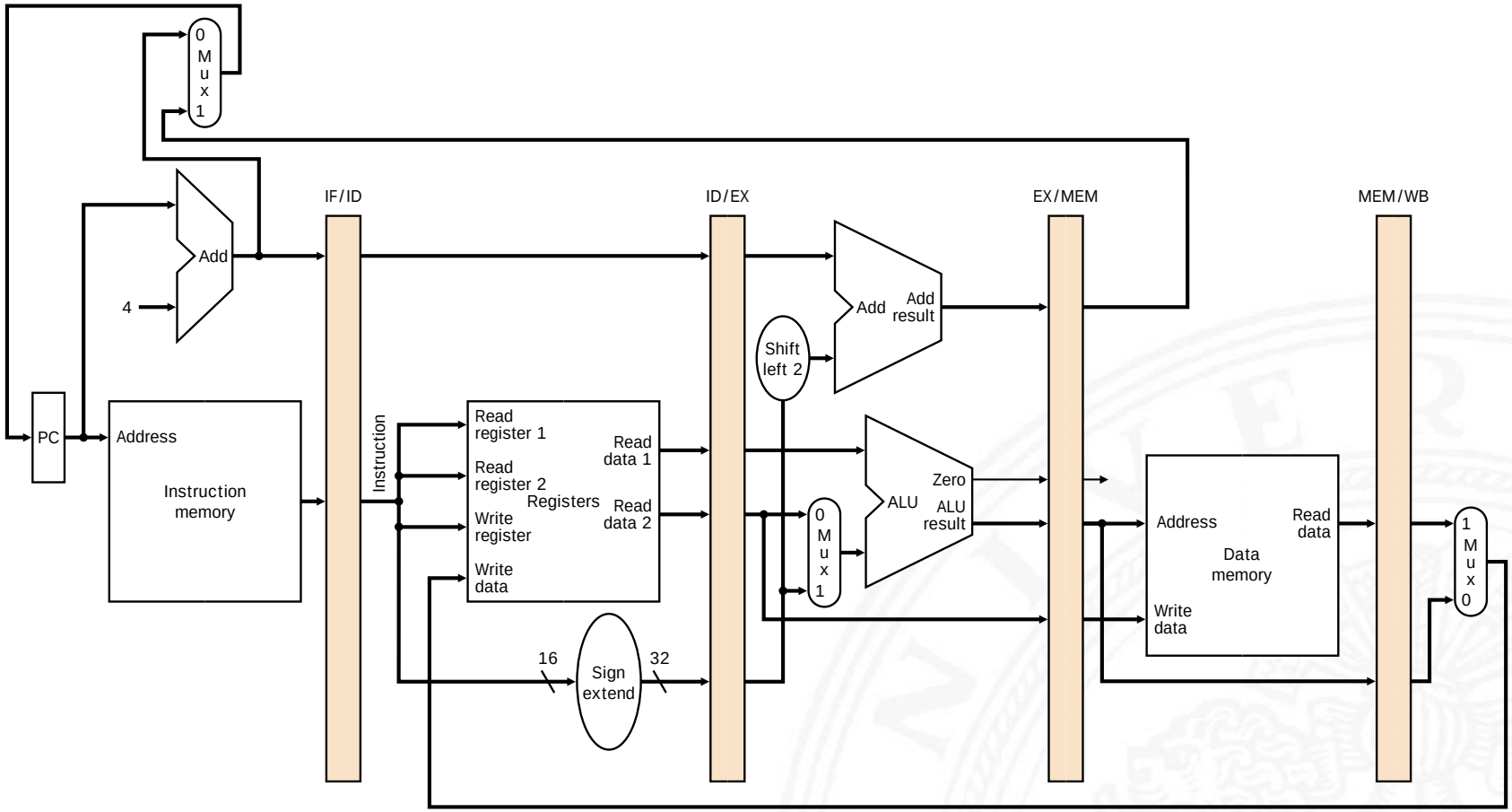
R1 = (R2 < R3)	`slt R1, R2, R3`

MIPS: Befehlssatz

- Übersicht und Details: [PH17, PH16b]
  David A. Patterson, John L. Hennessy: *Computer Organization and Design – The Hardware/Software Interface*
- dort auch hervorragende Erläuterung der Hardwarestruktur

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

MIPS: Hardwarestruktur



[PH16b]

PC
I-Cache

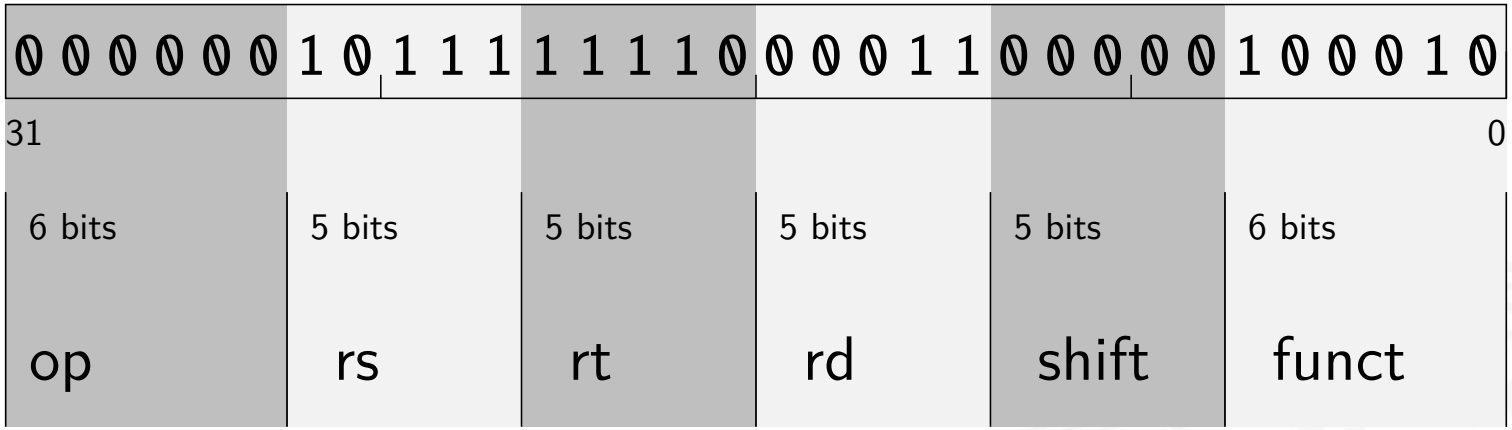
Register
(R0 . . . R31)

ALUs

Speicher
D-Cache

MIPS: Befehlsformate

Befehl im R-Format

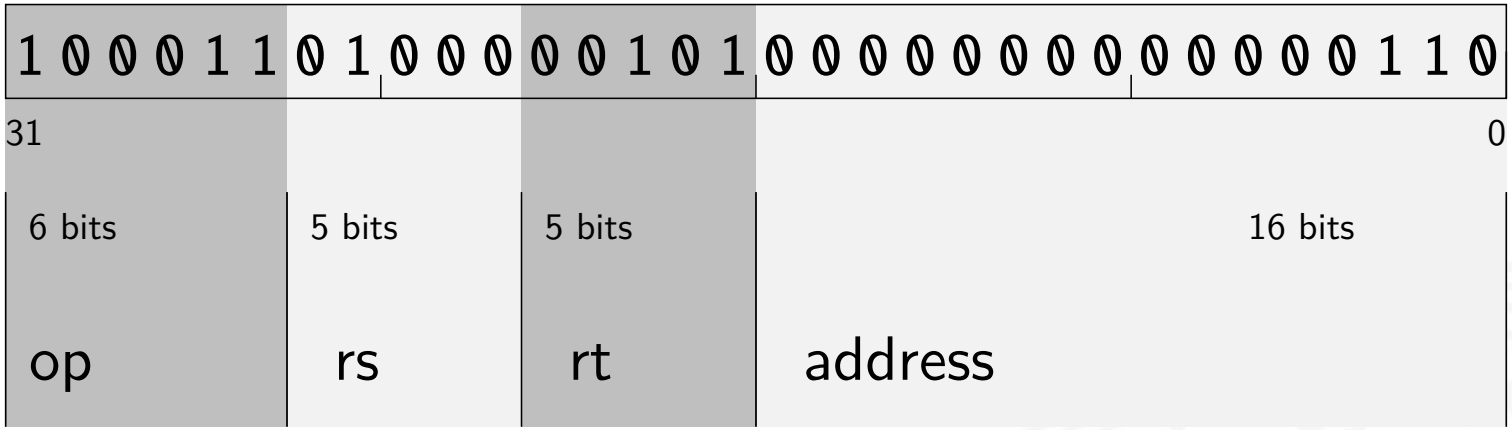


▶ op:	Opcode	Typ des Befehls	0	„alu-op“
rs:	source register 1	erster Operand	23	„r23“
rt:	source register 2	zweiter Operand	30	„r30“
rd:	destination register	Zielregister	3	„r3“
shift:	shift amount	(optionales Shiften)	0	„0“
funct:	ALU function	Rechenoperation	34	„sub“

⇒ r3 = r23 - r30 `sub r3, r23, r30`

MIPS: Befehlsformate

Befehl im I-Format



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

⇒ r5 = MEM[r8+6]

```
lw r5, 6(r8)
```

Befehlsformat: Beispiel M·CORE

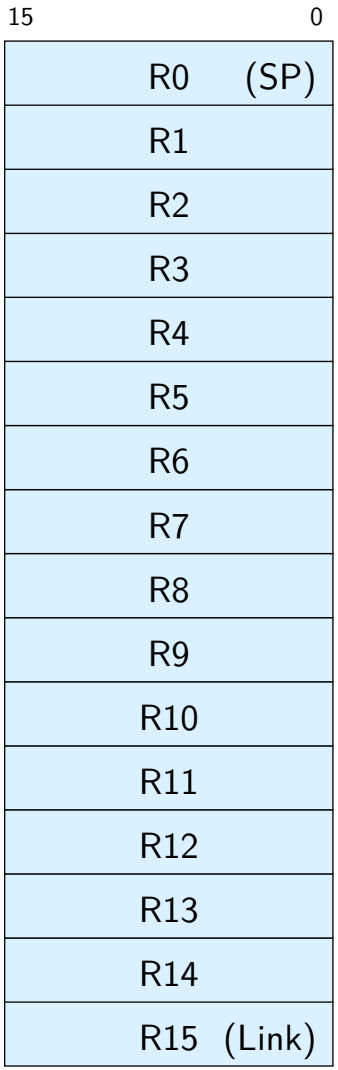
- 32-bit RISC Architektur, Motorola 1998
- besonders einfaches Programmiermodell
  - Program Counter PC
  - 16 Universalregister R0 . . . R15
  - Statusregister C („carry flag“)
  - 16-bit Befehle (um Programmspeicher zu sparen)
- Verwendung
  - Mikrocontroller für eingebettete Systeme
    z.B. „*Smart Cards*“
  - siehe `en.wikipedia.org/wiki/M.CORE`

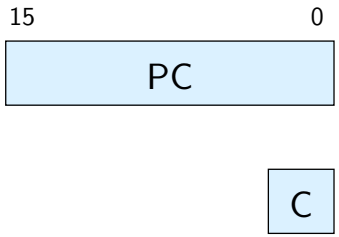
- ähnlich M·CORE
- gleiches Registermodell, aber nur 16-bit Wortbreite
  - Program Counter PC
  - 16 Universalregister R0 . . . R15
  - Statusregister C („carry flag“)

- Subset der Befehle, einfachere Codierung
- vollständiger Hardwareaufbau in Hades verfügbar
  - [HenHA] Hades Demo: `60-dcore/t3/chapter`

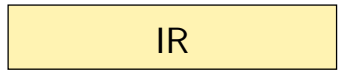
  oder Simulator mit Assembler
  - `tams.informatik.uni-hamburg.de/publications/onlineDoc`
    (`t3asm.jar` / `winT3asm.exe`)

D·CORE: Registermodell





- 16 Universalregister
- Programmzähler
- 1 Carry-Flag



- Befehlsregister

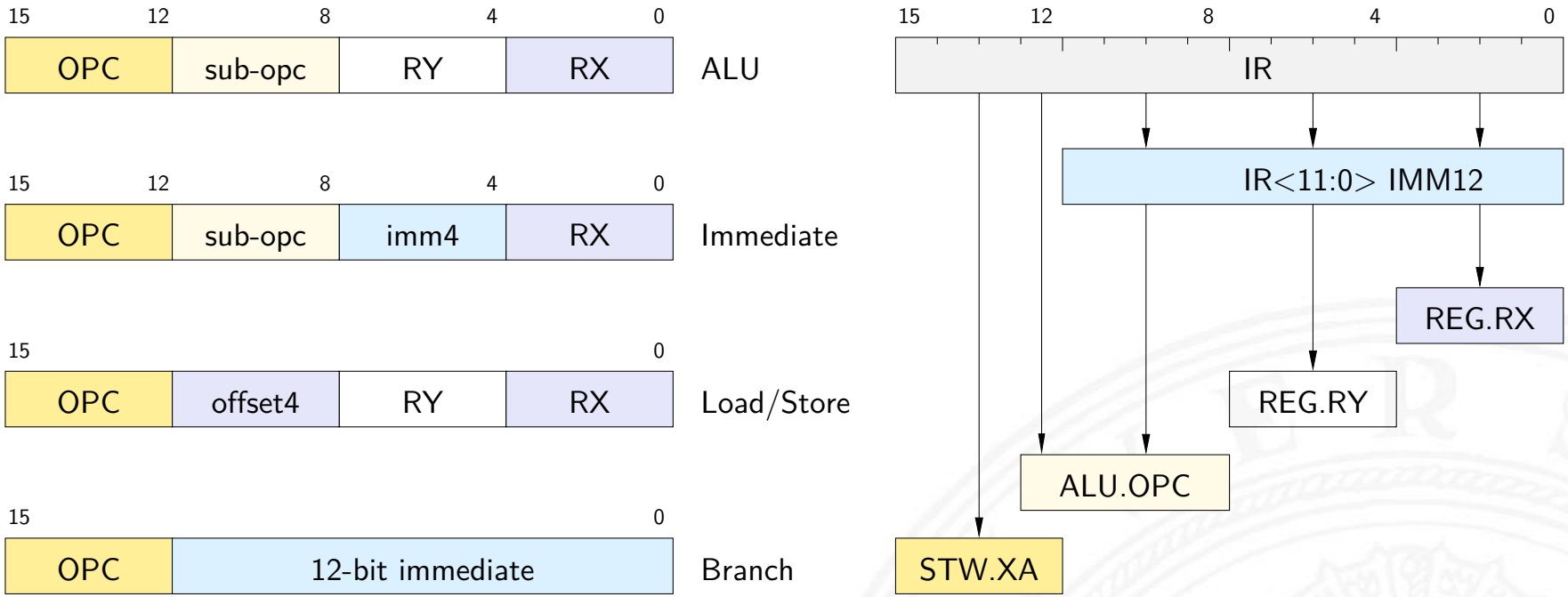
- Bus-Interface

- sichtbar für Programmierer: R0 . . . R15, PC und C

D·CORE: Befehlssatz

Befehl	Funktion
`mov`	move register
`addu, addc`	Addition (ohne, mit Carry)
`subu`	Subtraktion
`and, or, xor`	logische Operationen
`lsl, lsr, asr`	logische, arithmetische Shifts
`cmpe, cmpne ...`	Vergleichsoperationen
`movi, addi ...`	Operationen mit Immediate-Operanden
`ldw, stw`	Speicherzugriffe, load/store
`br, jmp`	unbedingte Sprünge
`bt, bf`	bedingte Sprünge
`jsr`	Unterprogrammaufruf
`trap`	Software interrupt
`rfi`	return from interrupt

D·CORE: Befehlsformate



- 4-bit Opcode, 4-bit Registeradressen
- einfaches Zerlegen des Befehls in die einzelnen Felder

Adressierungsarten

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

$\Rightarrow$ wichtige Unterscheidungsmerkmale für Rechnerarchitekturen

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

Beispiel: Add-Befehl

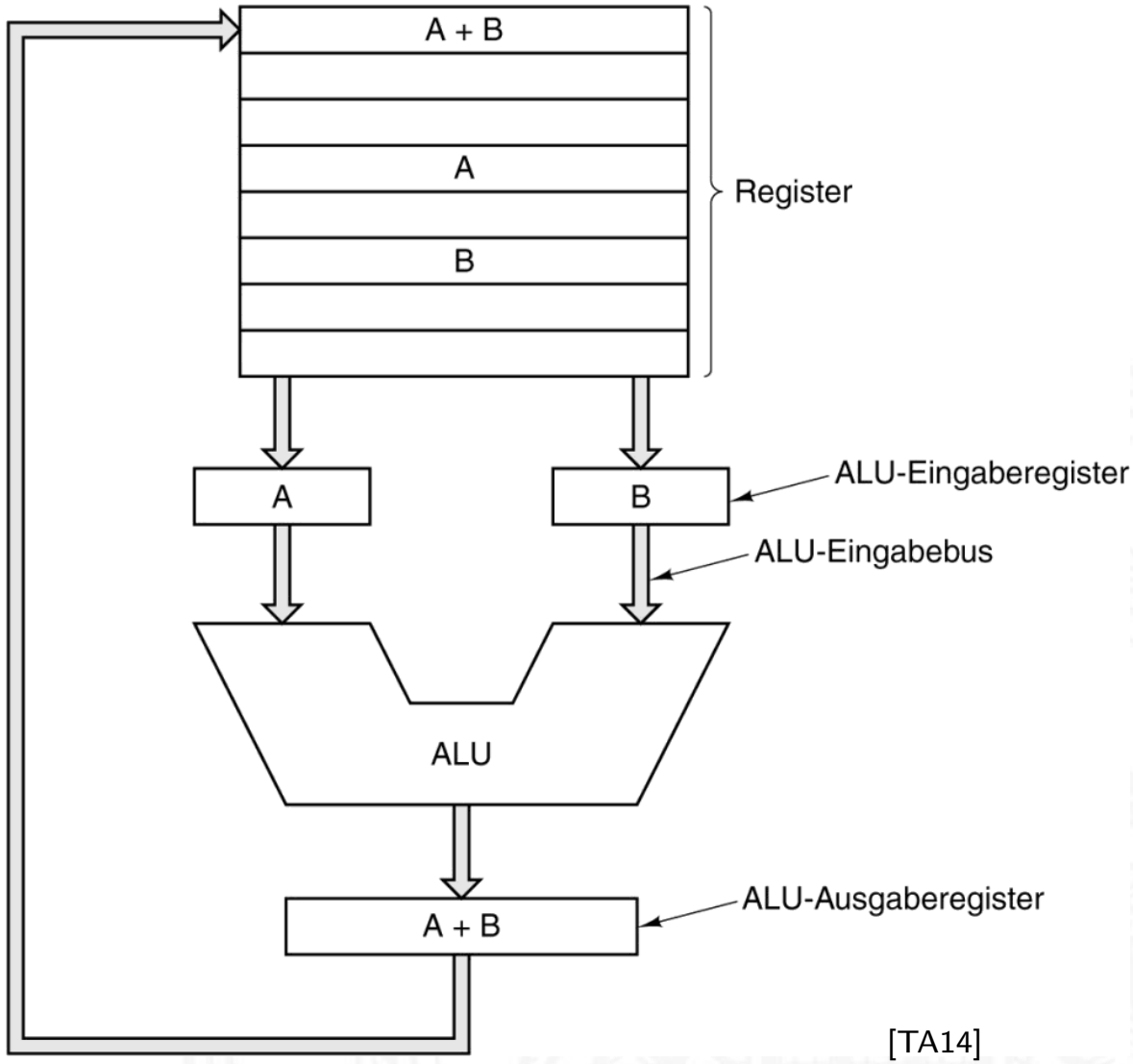
- Rechner soll „rechnen“ können
- typische arithmetische Operation nutzt 3 Variablen
  Resultat, zwei Operanden: $X = Y + Z$

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

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

Beispiel: Datenpfad

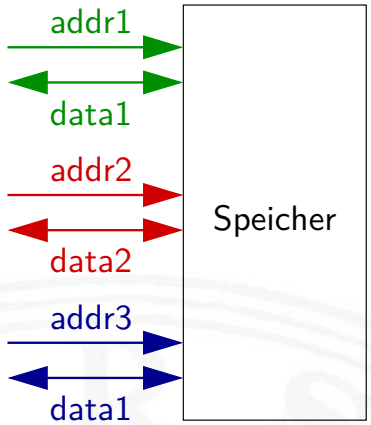
- Register (-bank)
  - liefern Operanden
  - speichern Resultate
- interne Hilfsregister
- ALU, typ. Funktionen:
  - add, add-carry, sub
  - and, or, xor
  - shift, rotate
  - compare
  - (floating point ops.)



[TA14]

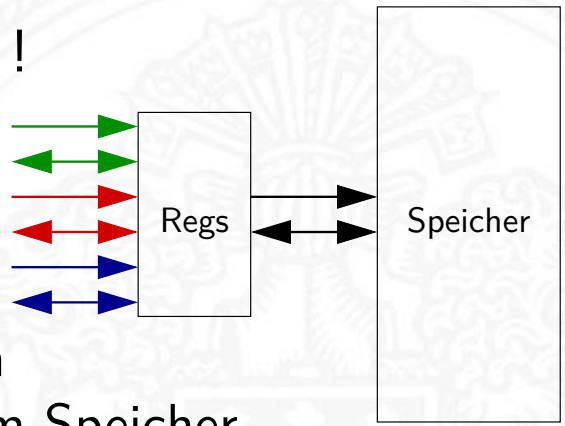
Woher kommen die Operanden?

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



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

$\Rightarrow$ Register im Prozessor zur Zwischenspeicherung !
  - Datentransfer zwischen Speicher und Registern
    Load reg = MEM[addr]
    Store MEM[addr] = reg



  - RISC: Rechenbefehle arbeiten *nur* mit Registern
  - CISC: gemischt, Operanden in Registern oder im Speicher

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

3-Adress Format

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

2-Adress Format

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

1-Adress Format

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

0-Adress Format

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

Beispiel: n-Adress Maschine

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

3-Adress Maschine

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

2-Adress Maschine

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

1-Adress Maschine

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

0-Adress Maschine

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

Beispiel: Stack-Maschine / 0-Adress Maschine

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

0-Adress Maschine	TOS	NOS	Stack
`push E`	E		
`push D`	D	E	
`mul`	D*E		
`push C`	C	D*E	
`add`	C+D*E		
`push B`	B	C+D*E	
`push A`	A	B	C+D*E
`sub`	A-B	C+D*E	
`div`	(A-B)/(C+D*E)		
`pop Z`			

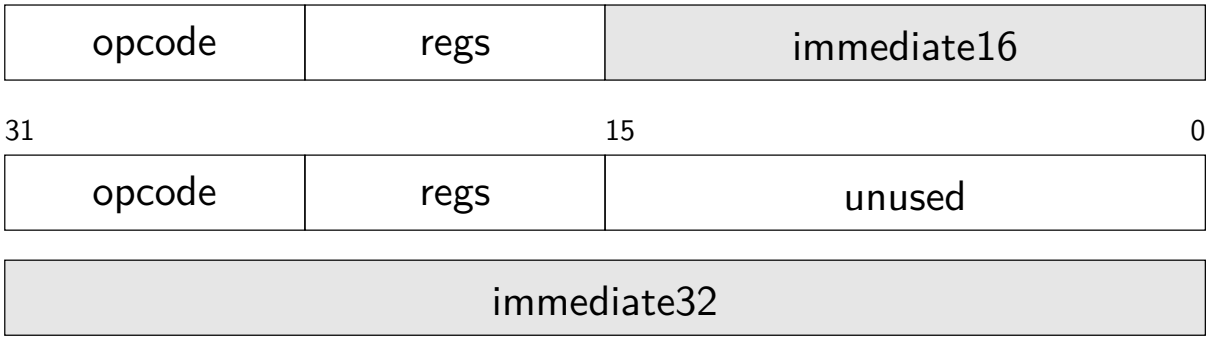
Adressierungsarten

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

Adressierungsarten (cont.)

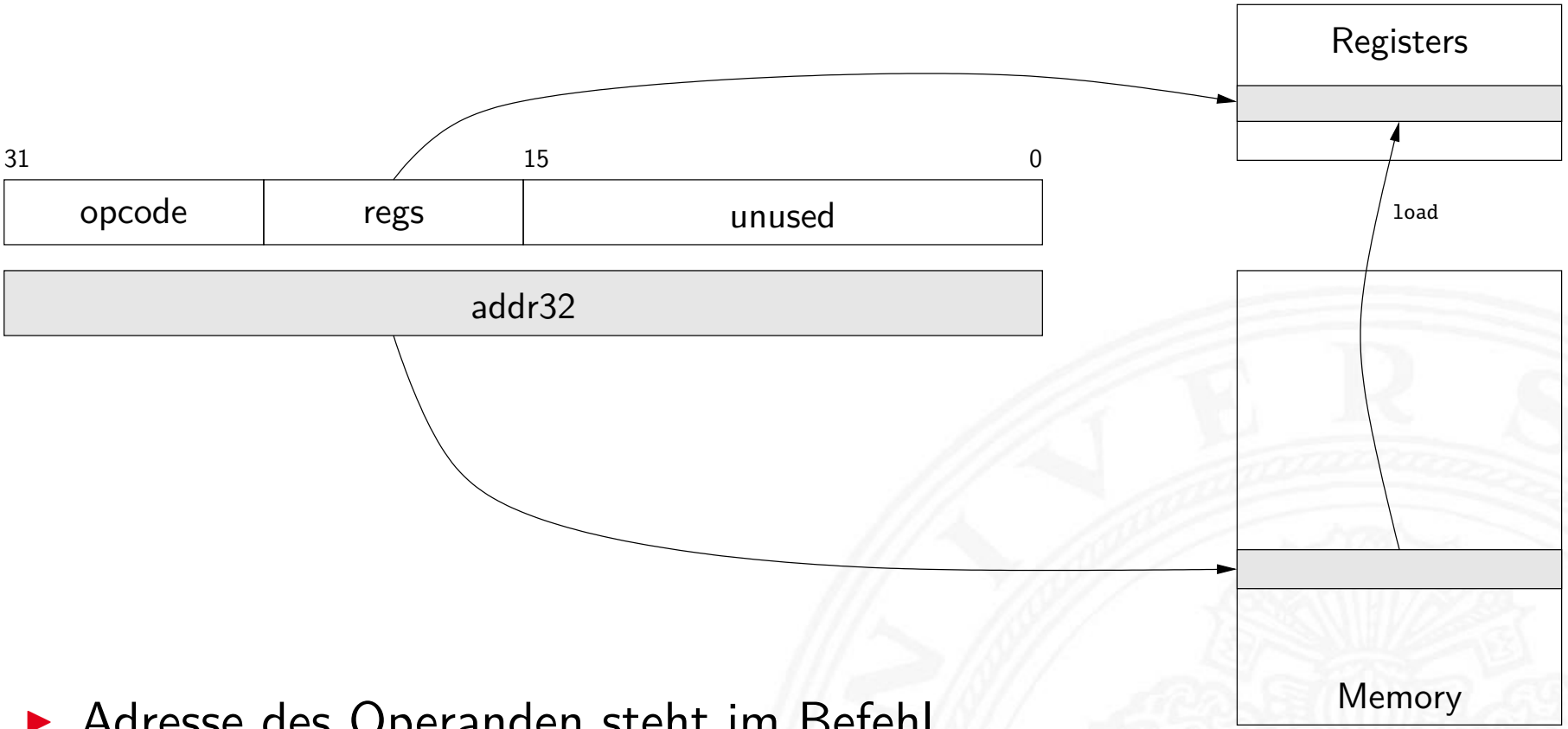
- „register“
  - wie Direktmodus, aber Register statt Speicher
  - 32 Register: benötigen 5 bit im Befehl
  - genug Platz für 2- oder 3-Adress Formate
- „register-indirekt“
  - Befehl spezifiziert ein Register
  - mit der Speicheradresse des Operanden
  - ein zusätzlicher Speicherzugriff
- „indiziert“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

Immediate Adressierung



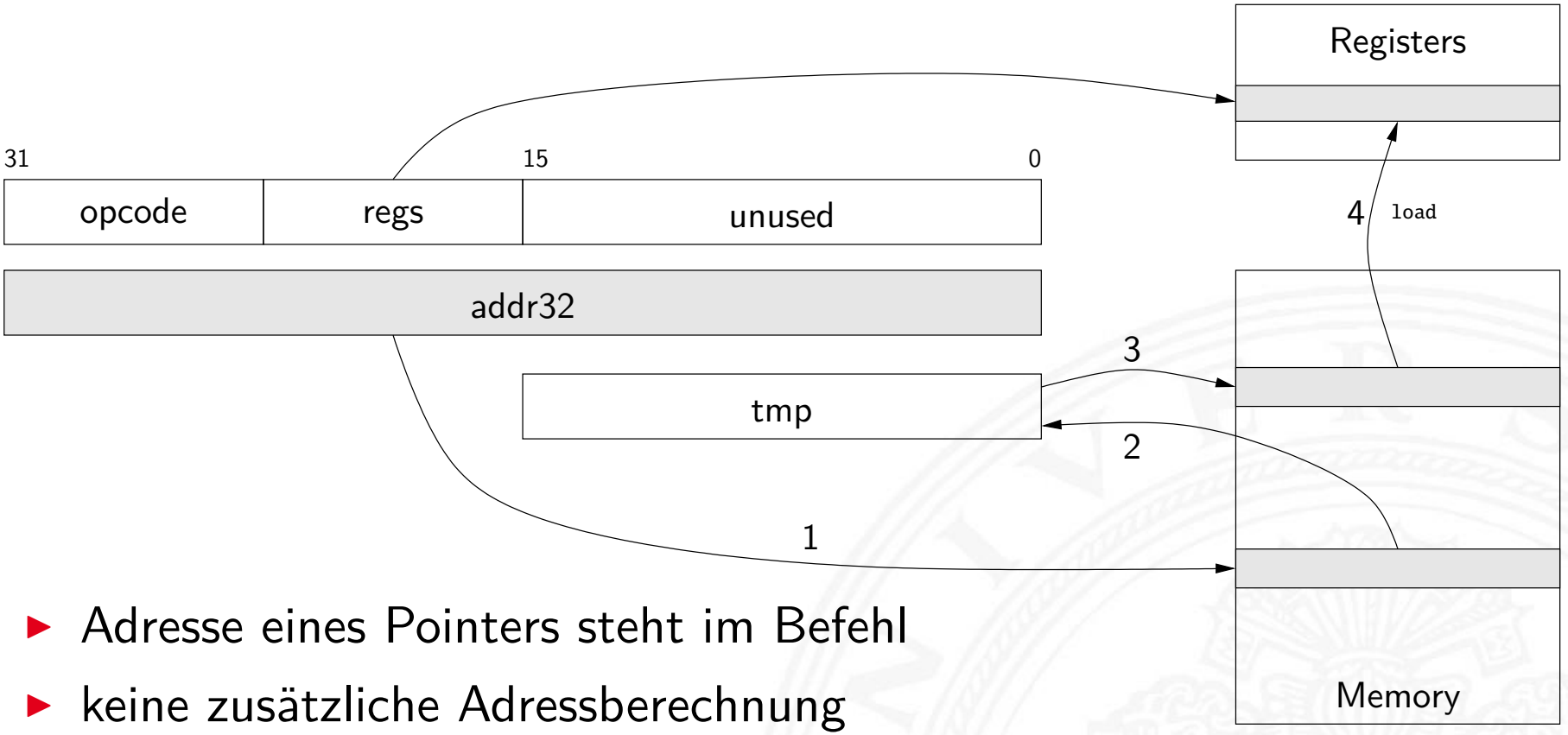
- Operand steht direkt im Befehl, kein zusätzlicher Speicherzugriff
- Länge des Operanden $<$ (Wortbreite - Opcodebreite)
- Darstellung größerer Zahlenwerte
  - 2-Wort Befehle (x86)
    zweites Wort für Immediate-Wert
  - mehrere Befehle (MIPS, SPARC)
    z.B. obere/untere Hälfte eines Wortes
  - Immediate-Werte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

Direkte Adressierung



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

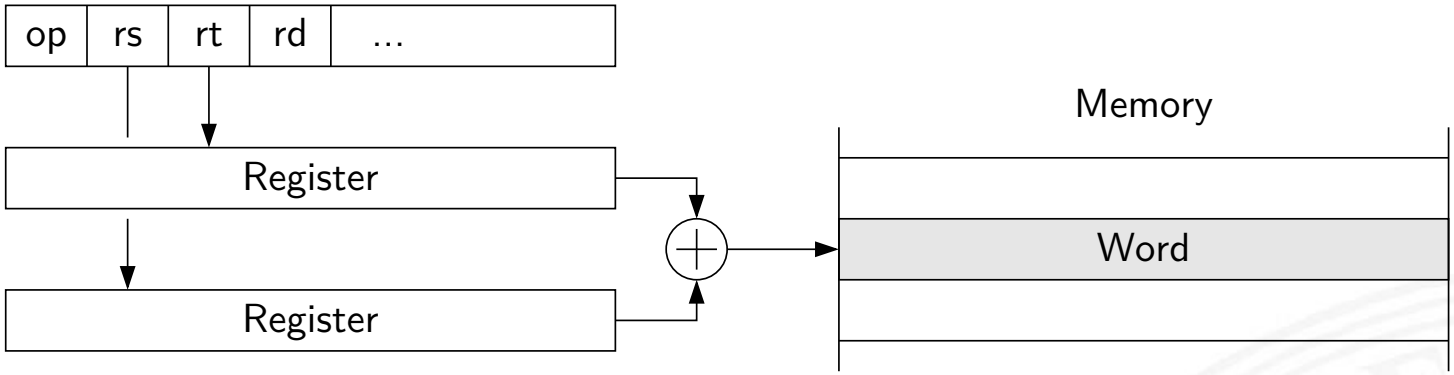
Indirekte Adressierung



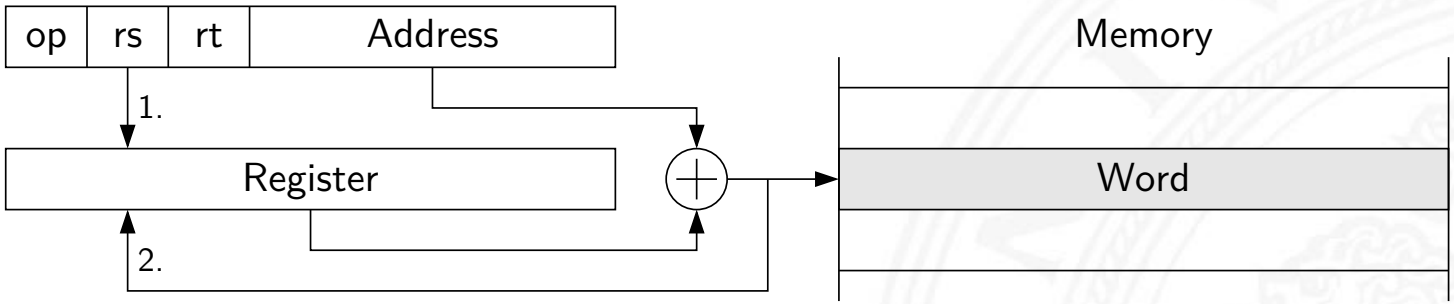
- Adresse eines Pointers steht im Befehl
- keine zusätzliche Adressberechnung
- zwei zusätzliche Speicherzugriffe:
  z.B. tmp = MEM[addr32] R3 = MEM[tmp]
- typische CISC-Adressierungsart, viele Taktzyklen
- kommt bei RISC-Rechnern nicht vor

Indizierte Adressierung

Indexaddressing



Updateaddressing



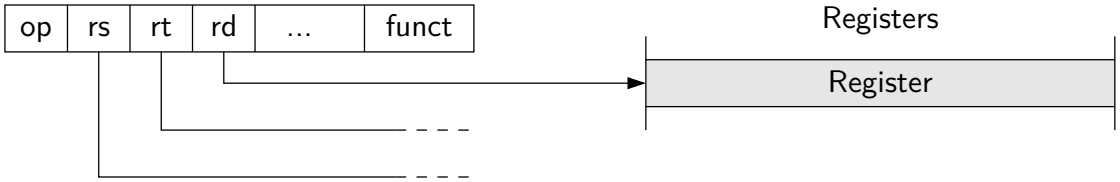
- indizierte Adressierung, z.B. für Arrayzugriffe
  - addr = $\langle Sourceregister \rangle + \langle Basisregister \rangle$
  - addr = $\langle Sourceregister \rangle$ + offset;
    Sourceregister = addr

Beispiel: MIPS Adressierungsarten

1. Immediate addressing

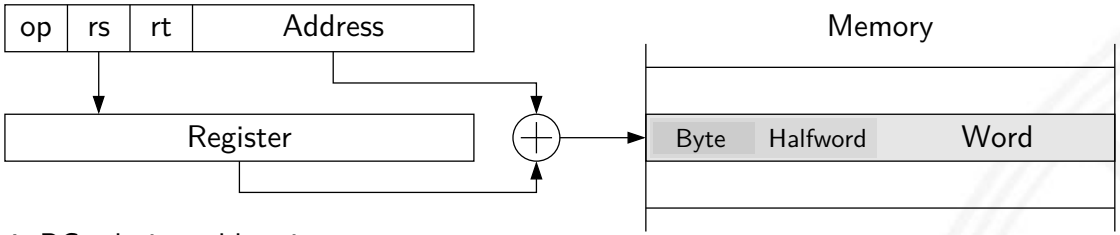
immediate

2. Register addressing



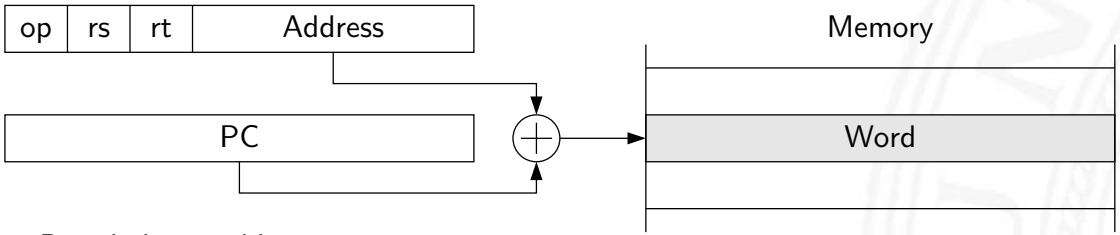
register

3. Base addressing



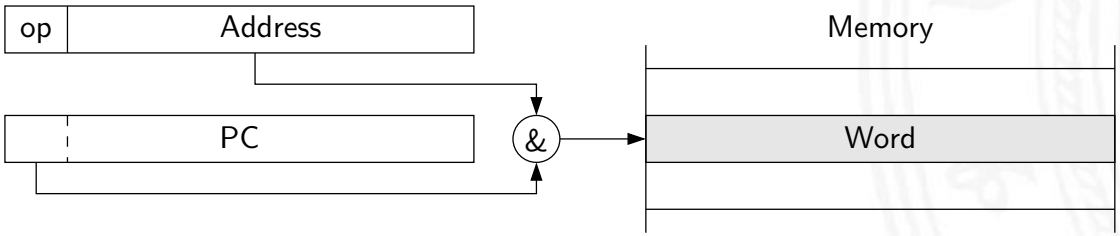
index + offset

4. PC-relative addressing



PC + offset

5. Pseudodirect addressing



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

typische Adressierungsarten

welche Adressierungsarten / -Varianten sind üblich?

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

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

Intel x86-Architektur

- übliche Bezeichnung für die Intel-Prozessorfamilie
- von 8086, 80286, 80386, 80486, Pentium . . . Pentium 4, Core 2, Core-i . . .
- eigentlich „IA-32“ (Intel architecture, 32-bit) . . . „IA-64“

- irreguläre Struktur: CISC
- historisch gewachsen: diverse Erweiterungen (MMX, SSE . . .)
- Abwärtskompatibilität: IA-64 mit IA-32 Emulation
- ab 386 auch wie reguläre 8-Register Maschine verwendbar

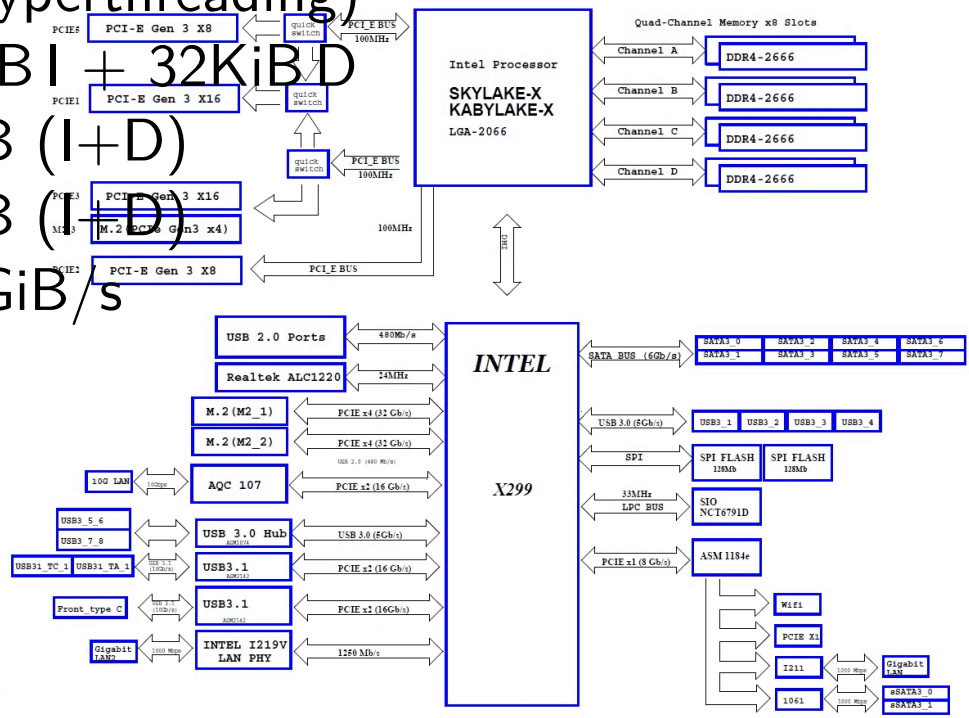
Hinweis: niemand erwartet, dass Sie sich alle Details merken

Intel x86: Evolution

Chip	Datum	MHz	Transistoren	Speicher	Anmerkungen
4004	4/1971	0,108	2 300	640 B	erster Mikroprozessor auf einem Chip
8008	4/1972	0,108	3 500	16 KiB	erster 8-bit Mikroprozessor
8080	4/1974	2	6 000	64 KiB	„general-purpose“ CPU auf einem Chip
8086	6/1978	5–10	29 000	1 MiB	erste 16-bit CPU auf einem Chip
8088	6/1979	5–8	29 000	1 MiB	Einsatz im IBM-PC
80286	2/1982	8–12	134 000	16 MiB	„Protected-Mode“
80386	10/1985	16–33	275 000	4 GiB	erste 32-bit CPU
80486	4/1989	25-100	1,2M	4 GiB	integrierter 8K Cache
Pentium	3/1993	60–233	3,1M	4 GiB	zwei Pipelines, später MMX
Pentium Pro	3/1995	150–200	5,5M	4 GiB	integrierter first und second-level Cache
Pentium II	5/1997	233–400	7,5M	4 GiB	Pentium Pro plus MMX
Pentium III	2/1999	450–1 400	9,5–44M	4 GiB	SSE-Einheit
Pentium 4	11/2000	1 300–3 600	42–188M	4 GiB	Hyperthreading
Core-2	5/2007	1 600–3 200	143–410M	4 GiB	64-bit Architektur, Mehrkernprozessoren
Core-i...	11/2008	2,500–3,600	$>$ 700M	64 GiB	Speichercontroller, Taktanpassung
...					GPU, I/O-Contr., Spannungsregelung ... Befehlssatz: AVX ...

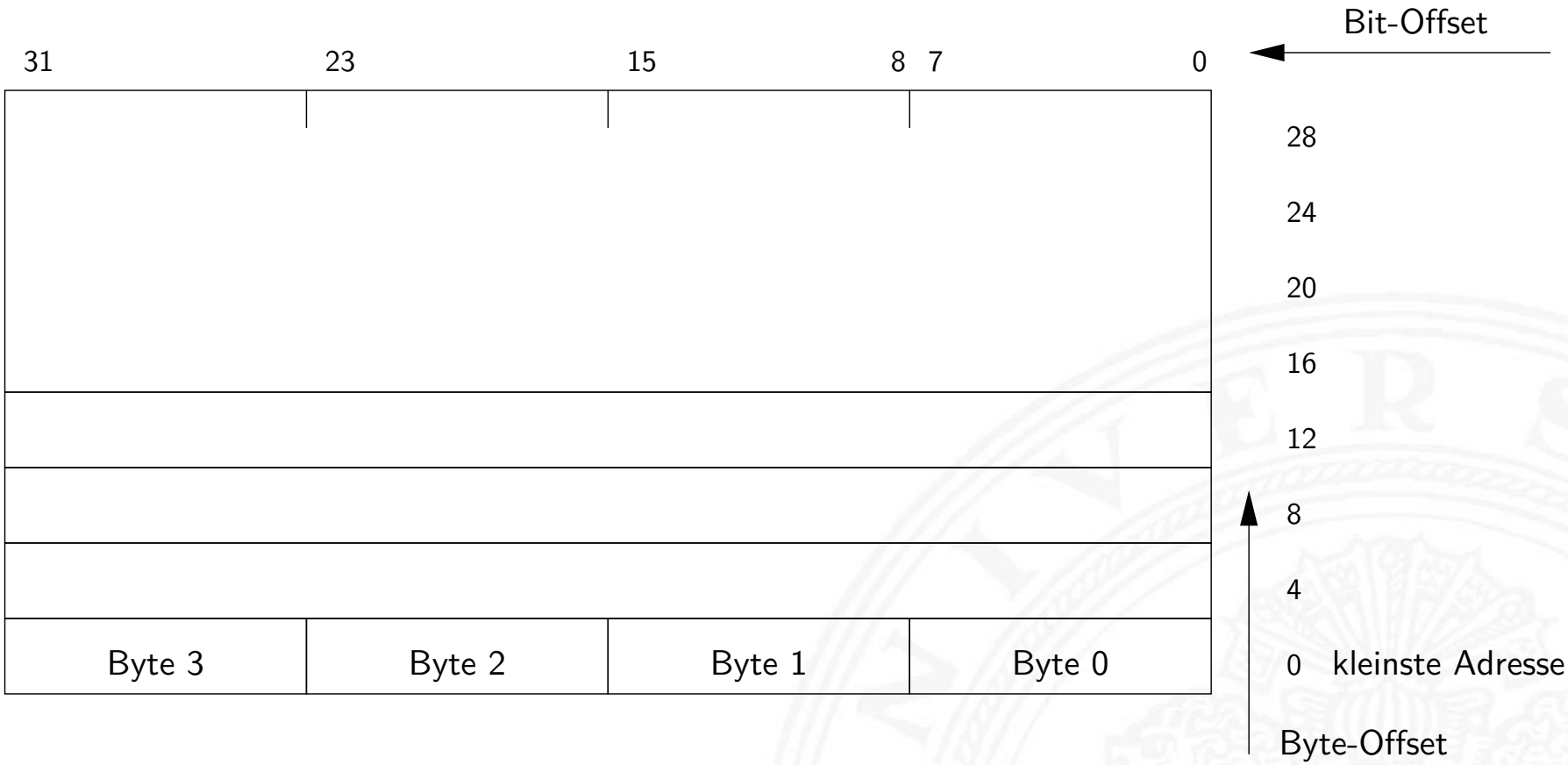
Beispiel: Core i9-9980XE Prozessor

Taktfrequenz	3,0 GHz (max. 4,4 GHz)
Anzahl der Cores	18 ($\times$ 2 Hyperthreading)
L1 Cache	18 $\times$ 32 KiB I + 32KiB D
L2 Cache	18 $\times$ 1 MiB (I+D)
L3 Cache	24,75 MiB (I+D)
Memory Controller	4 $\times$ 19,89 GiB/s
DMI Durchsatz	8 GT/s
Verbindung zu Chipsatz	
Bus Interface	64 Bits
Prozess	14 nm
Versorgungsspannung	0,7 - 1,2 V
Wärmeabgabe	$\sim$ 165 W



Quellen: `ark.intel.com`
`www.intel.de`
`en.wikichip.org`

x86: Speichermodell

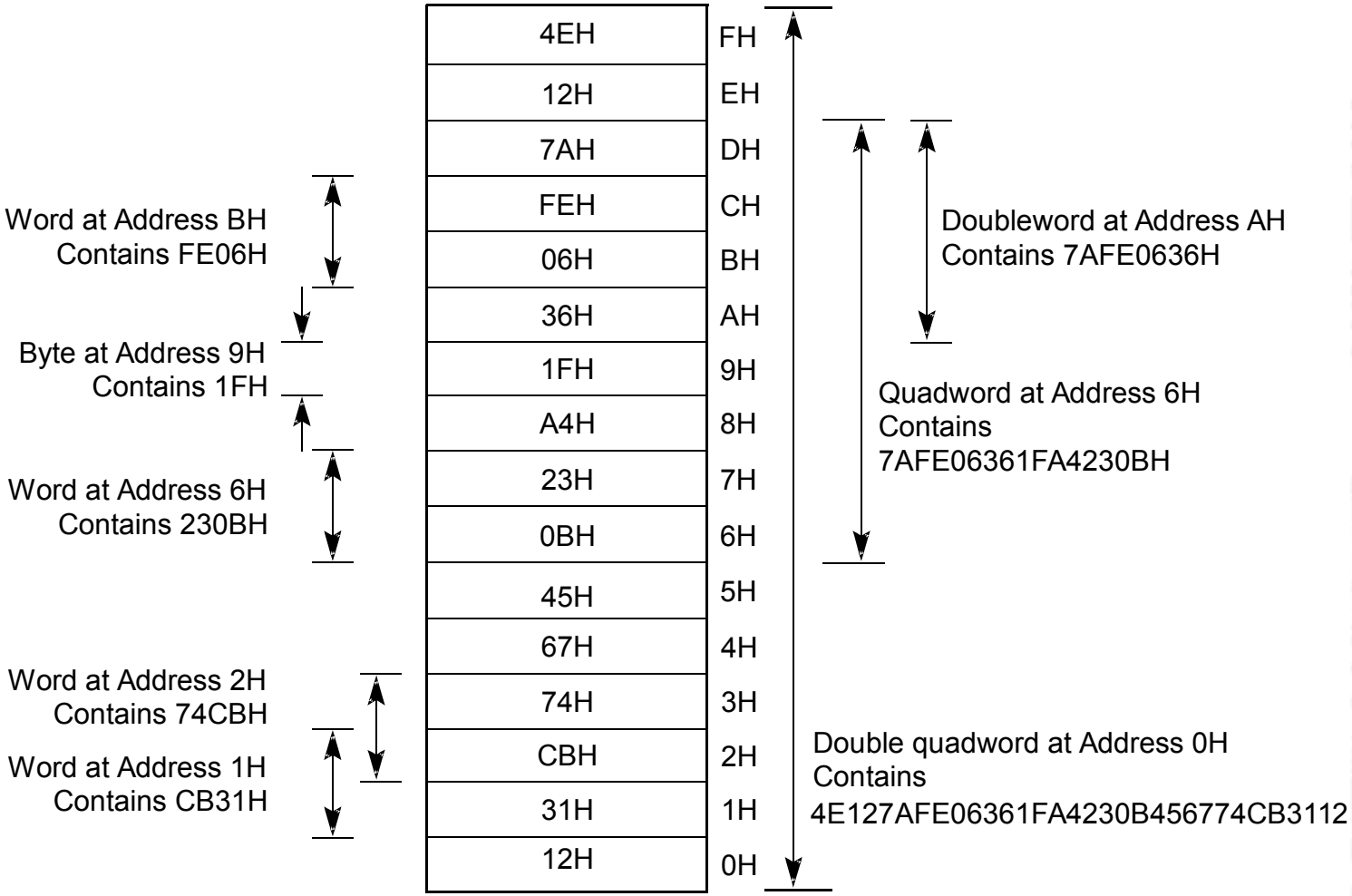


- „Little Endian“: LSB eines Wortes bei der kleinsten Adresse

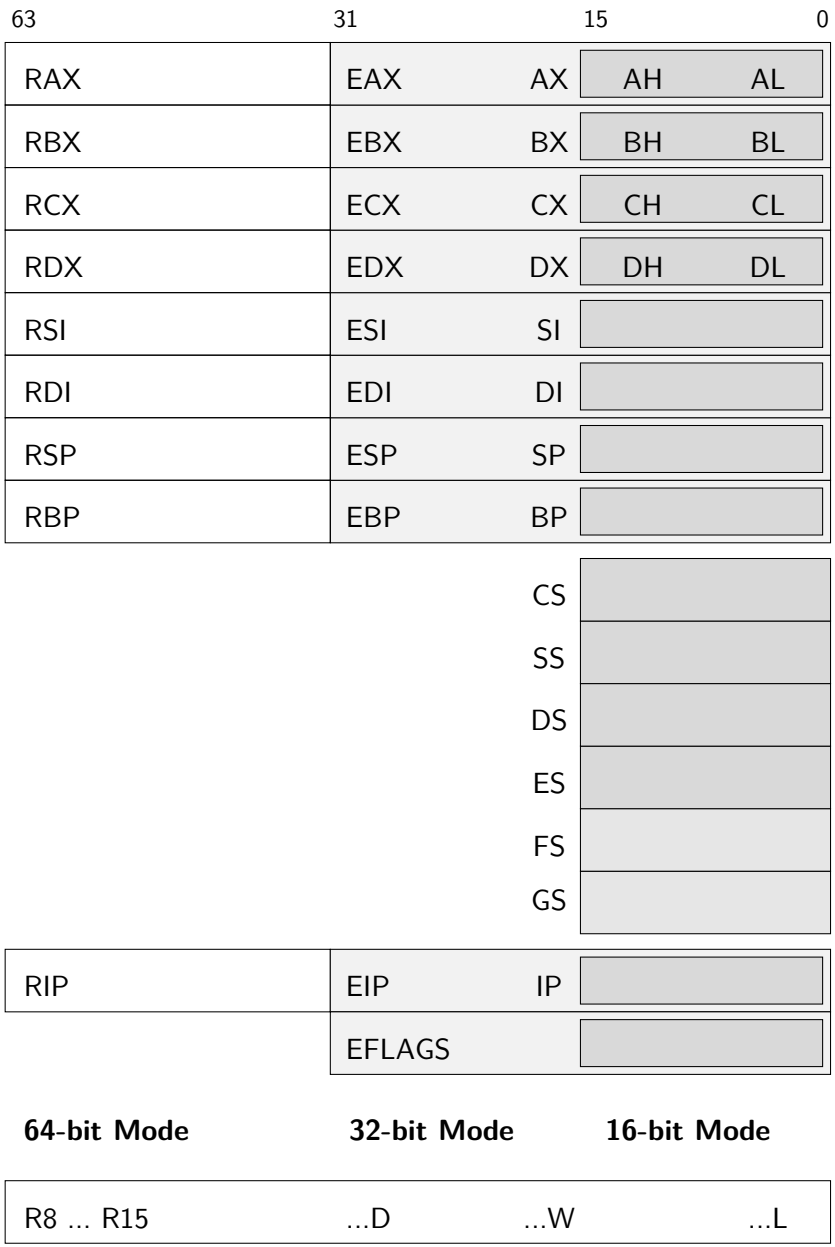
x86: Speichermodell (cont.)

- Speicher voll byte-adressierbar
- misaligned Zugriffe langsam
- Beispiel

[IA64]



x86: Register



accumulator

base addr

count: String, Loop

data, multiply/divide

index, string src

index, string dst

stackptr

base of stack segment

code segment

stack segment

data segment

extra data segment

PC

status

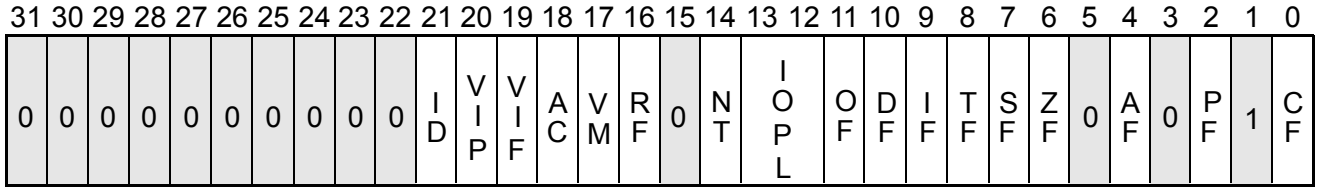
8086

E... ab 386

R... x86-64

FP Status

x86: EFLAGS Register



X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.

[IA64]

x86: Datentypen

bytes

word

doubleword

quadword

integer (2-complement b/w/dw/qw)

ordinal (unsigned b/w/dw/qw)

BCD (one digit per byte, multiple bytes)

packed BCD (two digits per byte, multiple bytes)

near pointer (32 bit offset)

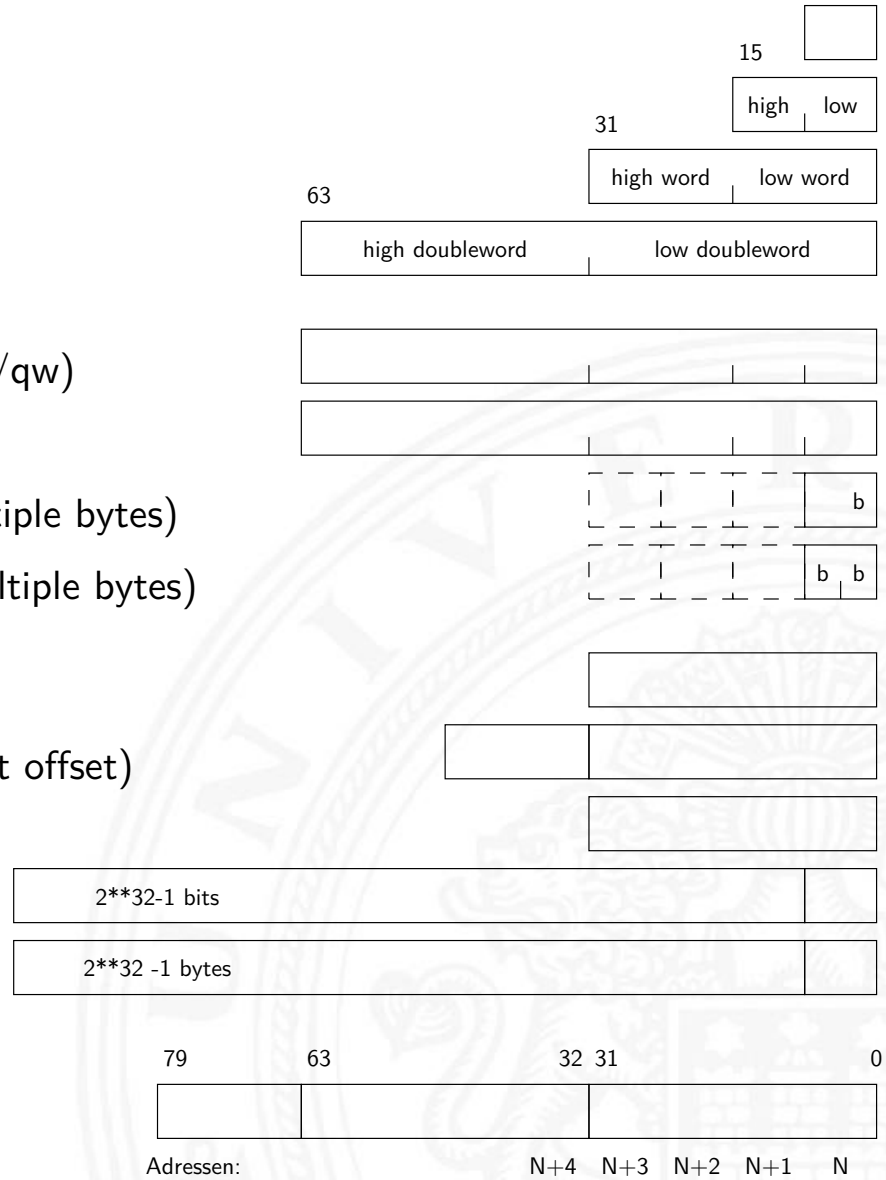
far pointer (16 bit segment + 32 bit offset)

bit field

bit string

byte string

float / double / extended



x86: Befehlssatz

Funktionalität	
Datenzugriff	`mov, xchg`
Stack-Befehle	`push, pusha, pop, popa`
Typumwandlung	`cwd, cdq, cbw` (byte→word), `movsx ...`
Binärarithmetik	`add, adc, inc, sub, sbb, dec, cmp, neg ...` `mul, imul, div, idiv ...`
Dezimalarithmetik	(packed/unpacked BCD) `daa, das, aaa ...`
Logikoperationen	`and, or, xor, not, sal, shr, shr ...`
Sprungbefehle	`jmp, call, ret, int, iret, loop, loopne ...`
String-Operationen	`ovs, cmps, scas, load, stos ...`
„high-level“	`enter` (create stack frame) ...
diverses	`lahf` (load AH from flags) ...
Segment-Register	`far call, far ret, lds` (load data pointer)

- CISC: zusätzlich diverse Ausnahmen/Spezialfälle

x86: Befehlsformate

- außergewöhnlich komplexes Befehlsformat

1.	prefix	repeat / segment override / etc.
2.	opcode	eigentlicher Befehl
3.	register specifier	Ziel / Quellregister
4.	address mode specifier	diverse Varianten
5.	scale-index-base	Speicheradressierung
6.	displacement	Offset
7.	immediate operand	

- außer dem Opcode alle Bestandteile optional
- unterschiedliche Länge der Befehle, von 1 . . . 36 Bytes

$\Rightarrow$ extrem aufwändige Decodierung

$\Rightarrow$ CISC – **C**omplex **I**nstruction **S**et **C**omputer

x86: Befehlsformat-Modifier („prefix“)

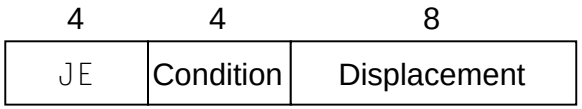
- alle Befehle können mit Modifiern ergänzt werden

segment override	Adresse aus angewähltem Segmentregister
address size	Umschaltung 16/32/64-bit Adresse
operand size	Umschaltung 16/32/64-bit Operanden
repeat	Stringoperationen: für alle Elemente
lock	Speicherschutz bei Multiprozessorsystemen

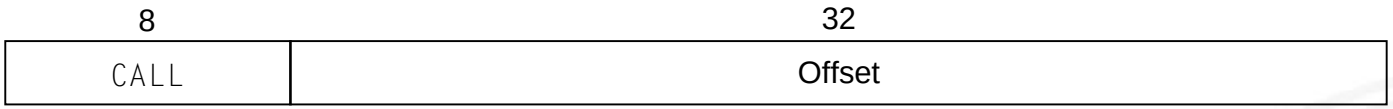
x86 Befehlscodierung: Beispiele

a. JE EIP + displacement

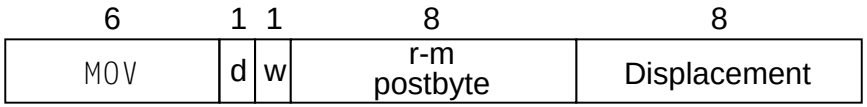
[PH16b]



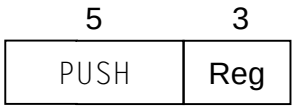
b. CALL



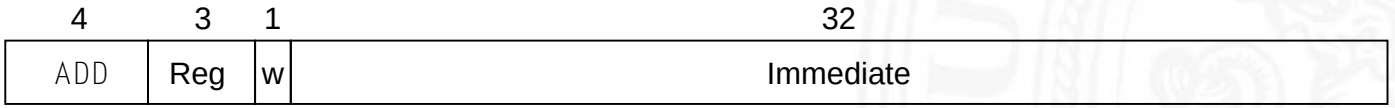
c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42

- 1 Byte . . . 36 Bytes
- vollkommen irregulär
- w: Auswahl 16/32 bit

x86 Befehlscodierung: Beispiele (cont.)

Instruction	Function
JE name	If equal (CC) {EIP=name}; EIP−128 ≤ name < EIP+128
JMP name	{EIP=name};
CALL name	SP=SP−4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI + 45]	EBX=M[EDI+45]
PUSH ESI	SP=SP−4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX=EAX+6765
TEST EDX,#42	Set condition codes (flags) with EDX & 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

[PH16b]

x86: Assembler-Beispiel print(. . .)


```
Addr  Opcode     Assembler                    C Quellcode
-------------------------------------------------------------------------------
                 .file    "hello.c"
                 .text
0000 48656C6C    .string  "Hello x86!\\n"
     6F207838
     36210A00
                 .text
                 print:
0000 55             pushl %ebp                | void print( char* s ) {
0001 89E5           movl %esp,%ebp
0003 53             pushl %ebx
0004 8B5D08         movl 8(%ebp),%ebx
0007 803B00         cmpb $0,(%ebx)            |   while( *s != 0 ) {
000a 7418           je .L18
                    .align 4
                 .L19:
000c A100000000     movl stdout,%eax          |     putc( *s, stdout );
0011 50             pushl %eax
0012 0FBE03         movsbl (%ebx),%eax
0015 50             pushl %eax
0016 E8FCFFFFFF     call _IO_putc
001b 43             incl %ebx                 |     s++;
001c 83C408         addl $8,%esp              |   }
001f 803B00         cmpb $0,(%ebx)
0022 75E8           jne .L19
                 .L18:
0024 8B5DFC         movl -4(%ebp),%ebx        | }
0027 89EC           movl %ebp,%esp
0029 5D             popl %ebp
002a C3             ret
```

x86: Assembler-Beispiel main(. . .)

```
Addr  Opcode     Assembler                    C Quellcode
-------------------------------------------------------------------------------
                 .Lfe1:
                 .Lscope0:
002b 908D7426       .align 16
     00
                 main:
0030 55             pushl %ebp               | int main( int argc, char** argv ) {
0031 89E5           movl %esp,%ebp
0033 53             pushl %ebx

0034 BB00000000     movl $.LC0,%ebx          |   print( "Hello x86!\\n" );
0039 803D0000       cmpb $0,.LC0
     000000
0040 741A           je .L26
0042 89F6           .align 4
                 .L24:
0044 A100000000     movl stdout,%eax
0049 50             pushl %eax
004a 0FBE03         movsbl (%ebx),%eax
004d 50             pushl %eax
004e E8FCFFFFFF     call _IO_putc
0053 43             incl %ebx
0054 83C408         addl $8,%esp
0057 803B00         cmpb $0,(%ebx)
005a 75E8           jne .L24
                 .L26:
005c 31C0           xorl %eax,%eax           |   return 0;
005e 8B5DFC         movl -4(%ebp),%ebx       | }
0061 89EC           movl %ebp,%esp
0063 5D             popl %ebp
0064 C3             ret
```

Bewertung der ISA

Kriterien für einen *guten* Befehlssatz

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

Bewertung der ISA (cont.)

Statistiken zeigen: Dominanz der einfachen Instruktionen

- x86-Prozessor

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

Bewertung der ISA (cont.)

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

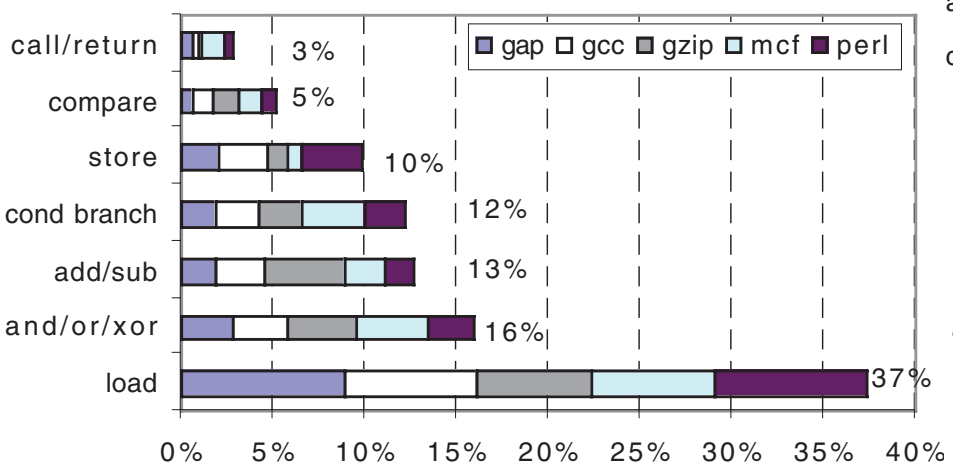
Figure D.15 80x86 instruction mix for five SPECint92 programs.

[HP17]

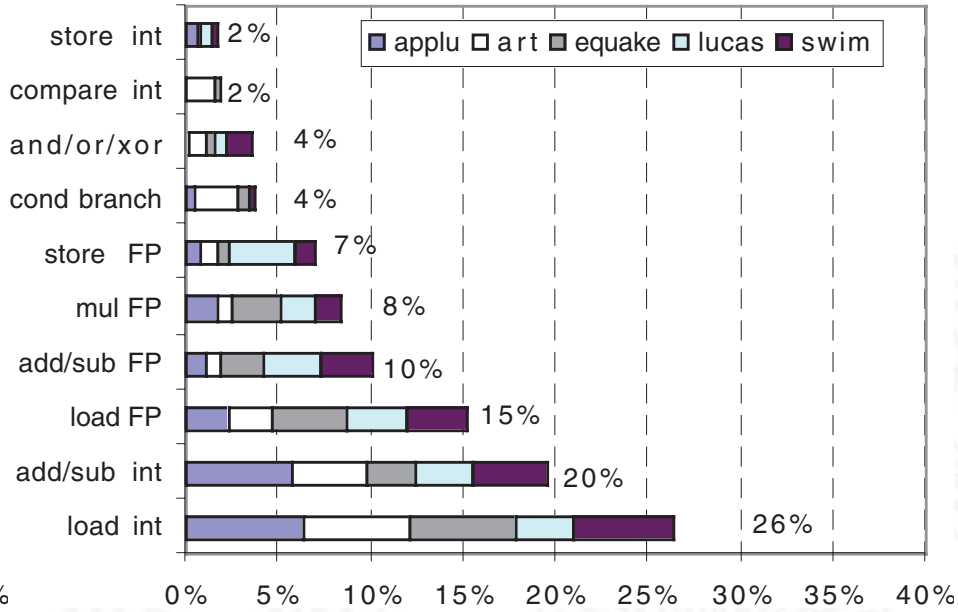
Bewertung der ISA (cont.)

- MIPS-Prozessor

[HP17]



SPECint2000 (96%)



SPECfp2000 (97%)

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

$\Rightarrow$ Motivation für RISC

CISC – Complex Instruction Set Computer

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

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

typische Merkmale

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

CISC – Complex Instruction Set Computer (cont.)

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

CISC – Complex Instruction Set Computer (cont.)

Vor- / Nachteile

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

Beispiele

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

CISC – Mikroprogrammierung

- ein Befehl kann nicht in einem Takt abgearbeitet werden

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

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

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

▸ horizontale Mikroprog.

CISC – Mikroprogrammierung (cont.)

2. vertikale Mikroprogrammierung

▸ vertikale Mikroprog.

   - kurze Mikroprogrammworte
   - Spalten enthalten Mikrooperationscode
   - mehrstufige Decodierung für Kontrolllleitungen

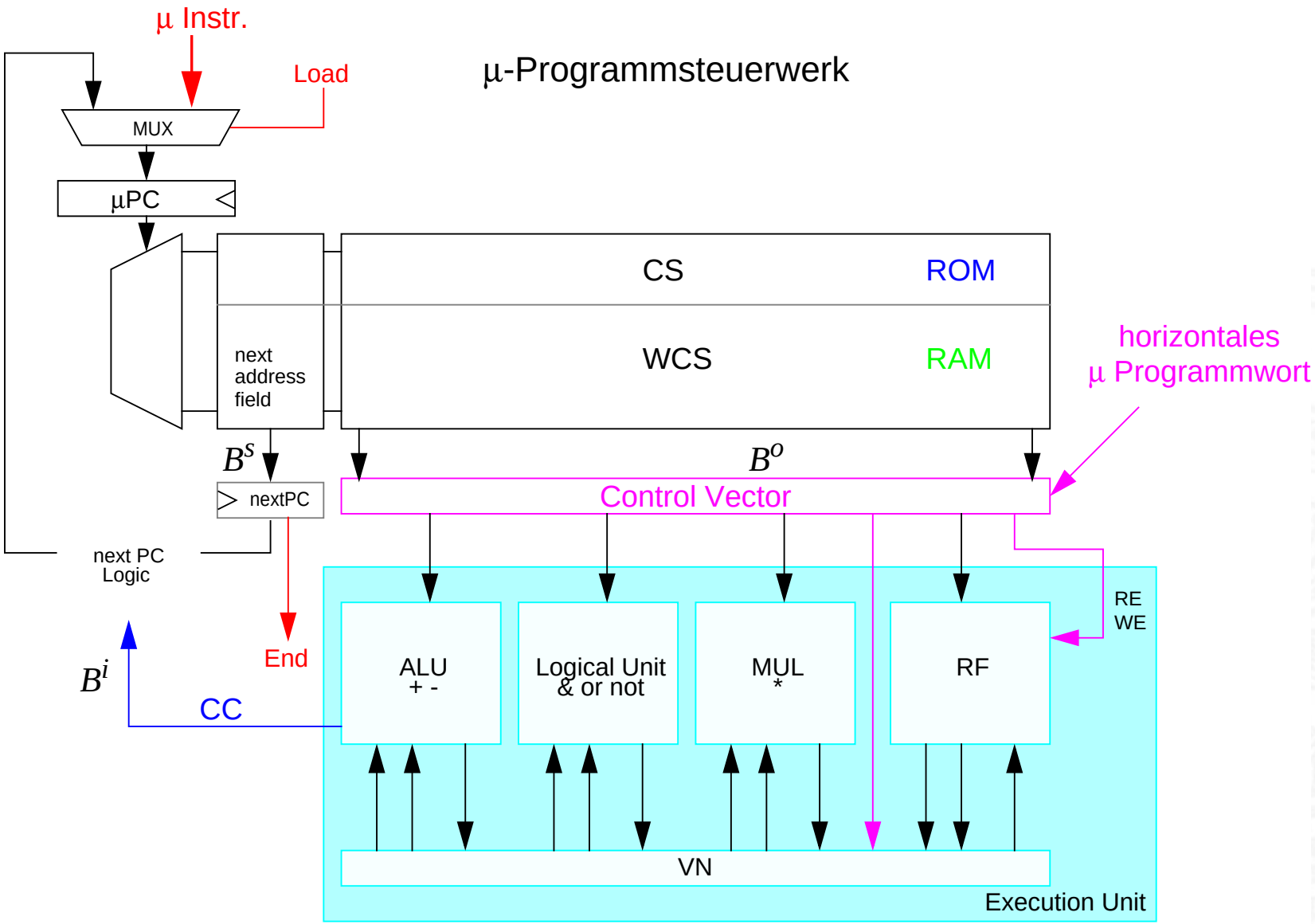
\+ CISC-Befehlssatz mit wenigen Mikrobefehlen realisieren

\+ μ-Programm im RAM: Mikrobefehlssatz austauschbar

\− (mehrstufige) ROM/RAM Zugriffe: zeitaufwändig

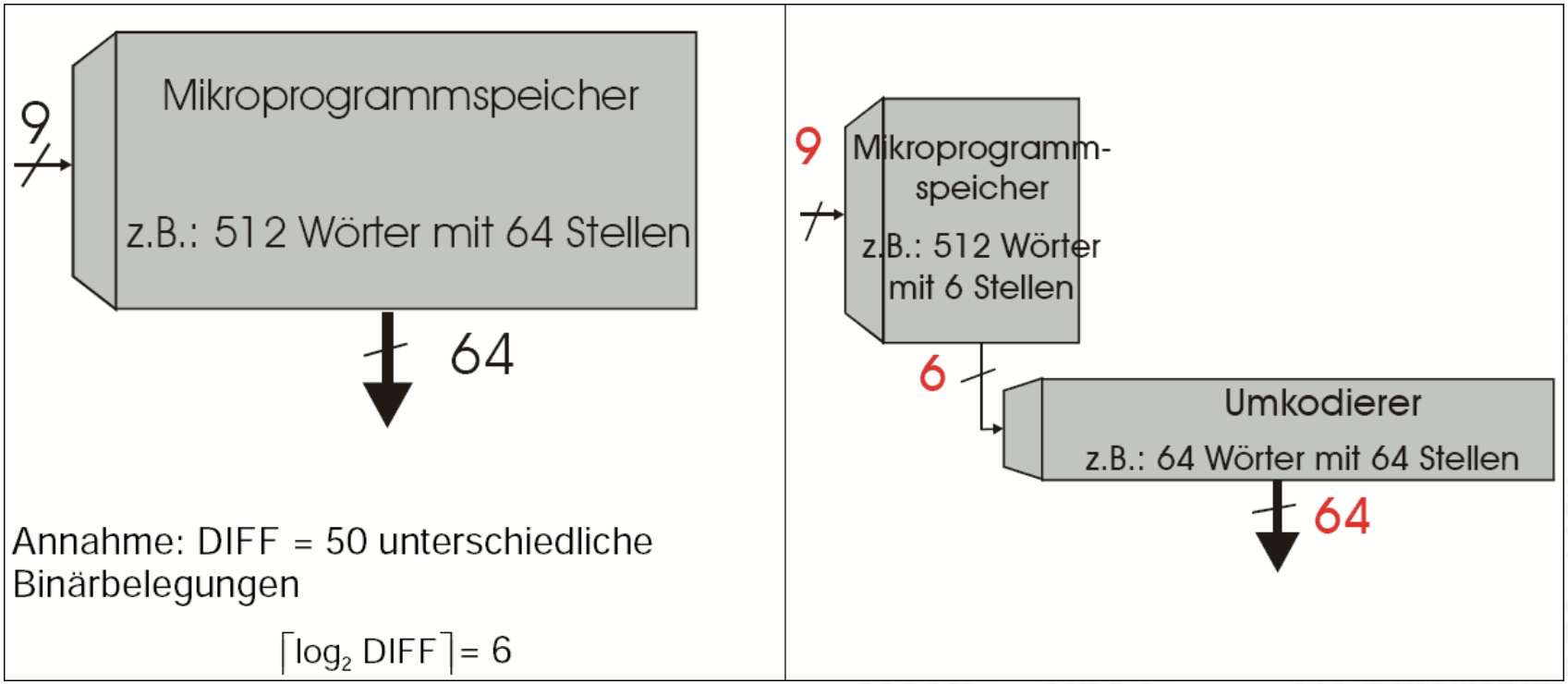
$\Rightarrow$ wird inzwischen nur noch benutzt, um CISC Befehle in RISC-artige Sequenzen umzusetzen (x86)

horizontale Mikroprogrammierung



◂ Mikroprogrammierung

vertikale Mikroprogrammierung



◂ Mikroprogrammierung

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

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

- Grundidee: Komplexitätsreduktion in der CPU
- seit den 80er Jahren: „RISC-Boom“
  - internes Projekt bei IBM
  - von Hennessy (Stanford) und Patterson (Berkeley) publiziert
- Hochsprachen und optimierende Compiler

  $\Rightarrow$ kein Bedarf mehr für mächtige Assemblerbefehle

  $\Rightarrow$ pro Assemblerbefehl muss nicht mehr „möglichst viel“ lokal in der CPU gerechnet werden (CISC Mikroprogramm)

Beispiele

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

typische Merkmale

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

RISC – **R**educed **I**nstruction **S**et **C**omputer (cont.)

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

RISC – **R**educed **I**nstruction **S**et **C**omputer (cont.)

Vor- / Nachteile

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

CISC vs. RISC

ursprüngliche Debatte

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

aktueller Stand

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

ISA Design heute

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

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

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

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

  $\Rightarrow$ benötigt hoch entwickelte Compiler

Literatur

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

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–8689–4238–5

Literatur (cont.)

[PH17] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – RISC-V Edition*.
Morgan Kaufmann Publishers Inc., 2017.
ISBN 978–0–12–812275–4

[PH16b] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle*.
5. Auflage, Oldenbourg, 2016. ISBN 978–3–11–044605–0

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

Literatur (cont.)

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

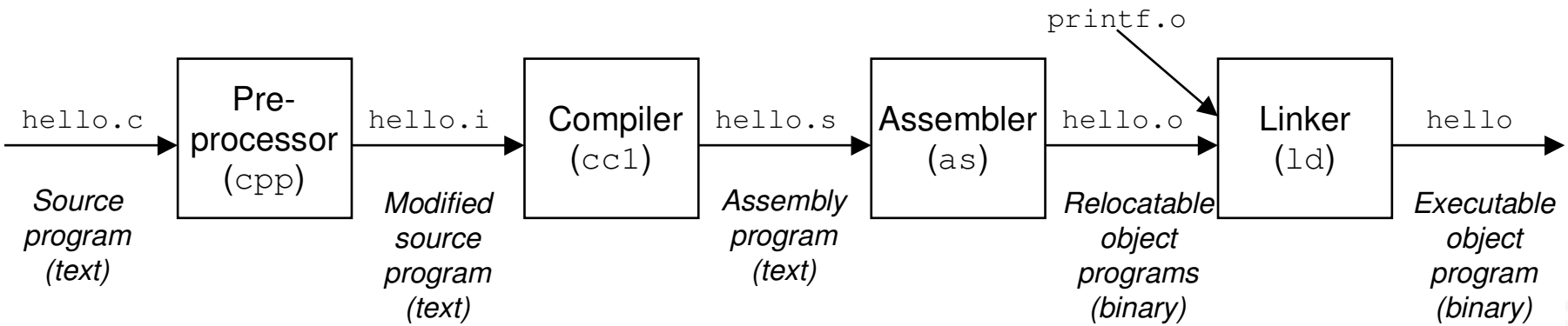
[HenHA] N. Hendrich: *HADES — HAmburg DEsign System*.
Universität Hamburg, FB Informatik, Lehrmaterial.
`tams.informatik.uni-hamburg.de/applets/hades/webdemos`

[IA64] *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture*.
Intel Corp.; Santa Clara, CA.
`software.intel.com/en-us/articles/intel-sdm`

Gliederung

Gliederung (cont.)

Wdh. Compilierungssystem



[BO15]

- verschiedene Repräsentationen des Programms
  - Hochsprache
  - Assembler
  - Maschinensprache
- Ausführung der Maschinensprache
  - von-Neumann Zyklus: Befehl holen, decodieren, ausführen
  - reale oder virtuelle Maschine

Warum Assembler?

Programme werden nur noch selten in Assembler geschrieben

- Programmentwicklung in Hochsprachen weit produktiver
- Compiler/Tools oft besser als handcodierter Assembler

aber Grundwissen bleibt trotzdem unverzichtbar

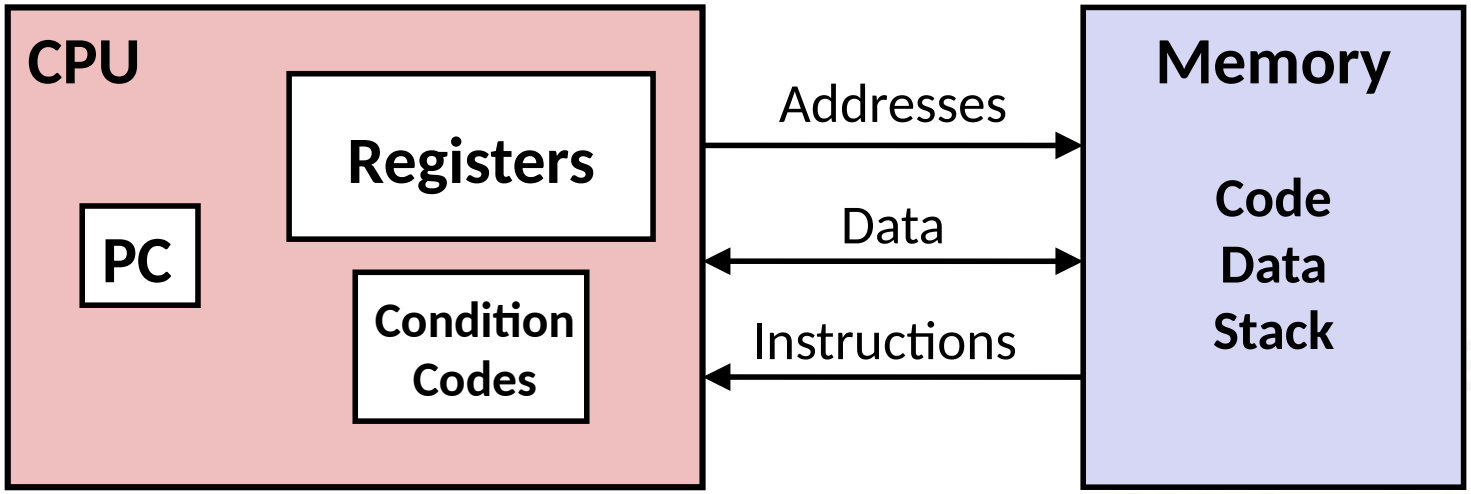
- Verständnis des Ausführungsmodells auf der Maschinenebene
- Programmverhalten bei Fehlern / Debugging
  - das High-Level Sprachmodell ist dort nicht anwendbar
- Programmleistung verstärken
  - Ursachen für Programm-Ineffizienz verstehen
  - effiziente „maschinengerechte“ Datenstrukturen / Algorithmen
- Systemsoftware implementieren
  - Compilerbau: Maschinencode als Ziel
  - Betriebssysteme implementieren (Prozesszustände verwalten)
  - Gerätetreiber schreiben

Assembler in RSB

- Beschränkung auf wesentliche Konzepte
  - GNU Assembler für x86-64 (Linux, 64-bit)
  - nur ein Datentyp: 64-bit Integer (`long`)
  - nur kleiner Subset des gesamten Befehlssatzes

- diverse nicht behandelte Themen
  - Makros
  - Implementierung eines Assemblers (2-pass)
  - Tipps für effizientes Programmieren
  - Befehle für die Systemprogrammierung (supervisor mode)
  - x86 Gleitkommabefehle
  - ...

Assemblersicht des Programmierers



[BO15]

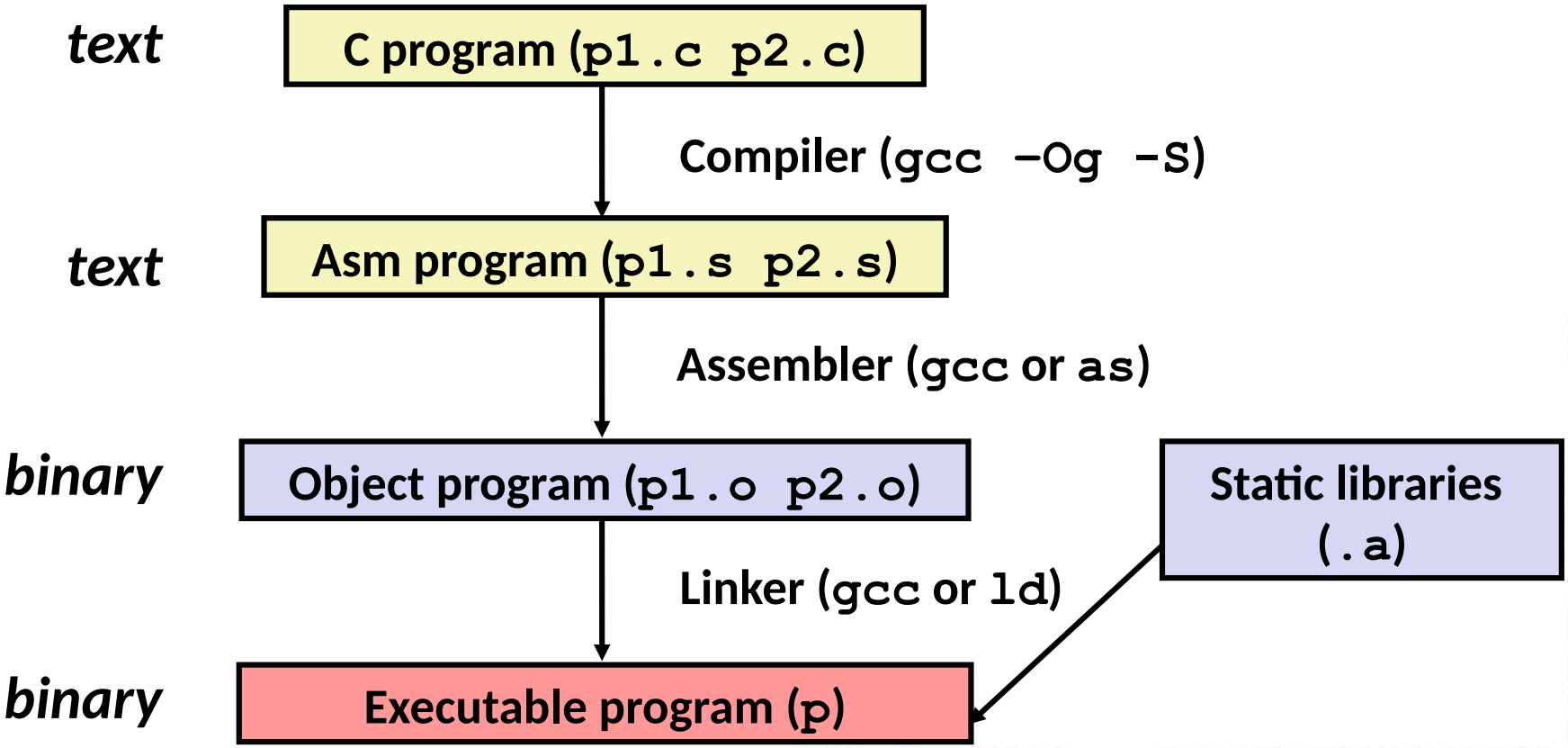
beobachtbare Zustände

- Programmzähler (*Instruction Pointer*) — x86-64 `rip` Register
  - Adresse der nächsten Anweisung
- Registerbank — `rax`...`rbp` Register
  - häufig benutzte Programmdaten — `r8`...`r15` Register
- Zustandscodes — EFLAGS Register
  - gespeicherte Statusinformationen über die letzte arithmetische Operation
  - für bedingte Sprünge benötigt (*Conditional Branch*)

Assemblersicht des Programmierers (cont.)

- Speicher
  - byteweise adressierbares Array
  - Code, Nutzerdaten, (einige) OS Daten
  - beinhaltet Kellerspeicher für Unterprogrammaufrufe „Stack“
  - –"– dynamischen Adressraum „Heap“

Umwandlung von C in Objektcode



[BO15]

Compilieren zu Assemblercode: Funktion sum()

sum.c

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

sum.s

```
sumstore:
    pushq   %rbx
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

[BO15]

- Befehl `gcc -Og -S sum.c`
- Erzeugt `sum.s`

Assemblercode: sum.s


```
    .globl  sumstore
    .type   sumstore, @function
sumstore:
.LFB35:
    .cfi_startproc
    pushq   %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
.LFE35:
    .size   sumstore, .-sumstore
```

Assemblercode: sum.s


```
    .globl  sumstore
    .type   sumstore, @function
sumstore:
.LFB35:
    .cfi_startproc
    pushq   %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
.LFE35:
    .size   sumstore, .-sumstore
```

```
sumstore:
    pushq   %rbx
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

- alles was mit „.“ beginnt: Label, Anweisungen für Linker

Assembler: Charakteristika

- hardwarenahe Programmierung: Zugriff auf kompletten Befehlssatz und alle Register einer Maschine

- je ein Befehl pro Zeile
  - Mnemonics für die einzelnen Maschinenbefehle
  - Konstanten als Dezimalwerte oder Hex-Werte
  - eingängige Namen für alle Register
  - Adressen für alle verfügbaren Adressierungsarten
  - Konvention bei gcc/as x86: Ziel einer Operation steht rechts

- symbolische Label für Sprungadressen
  - Verwendung in Sprungbefehlen
  - globale Label definieren Einsprungpunkte für den Linker/Loader

Assembler: Datentypen

- nur die von der Maschine unterstützten „primitiven“ Daten
- keine Aggregattypen wie Arrays, Strukturen, oder Objekte
  - nur fortlaufend adressierbare Bytes im Speicher

- Ganzzahl-Daten, z.B. 1, 2, 4, oder 8 Bytes — 8 . . . 64 bits
  - Datenwerte für Variablen — int/long/long long
  - positiv oder vorzeichenbehaftet — signed/unsigned
  - Textzeichen (ASCII, Unicode) — char
- Gleitkomma-Daten mit 4 oder 8 Bytes — float/double
- Adressen bzw. „Pointer“ — untypisierte Adressverweise

Assembler: Befehle/Operationen

- arithmetische/logische Funktionen auf Registern und Speicher
  - Addition/Subtraktion, Multiplikation usw.
  - bitweise logische und Schiebe-Operationen
- Datentransfer zwischen Speicher und Registern
  - Daten aus Speicher in Register laden
  - Registerdaten im Speicher ablegen
  - ggf. auch Zugriff auf Spezial-/OS-register
- Kontrolltransfer
  - unbedingte / bedingte Sprünge
  - Unterprogrammaufrufe: Sprünge zu/von Prozeduren
  - Interrupts, Exceptions, System-Calls

- Makros: Folge von Assemblerbefehlen

Objektcode: Funktion `sumstore()`

- 14 Bytes Programmcode
- x86-Instruktionen mit 1-, 3- oder 5 Bytes
  Erklärung s.u.

- Startadresse: `0x400595`
- vom Compiler/Assembler gewählt

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Assembler und Linker

Assembler

- übersetzt `.s` zu `.o`
- binäre Codierung jeder Anweisung
- (fast) vollständiges Bild des ausführbaren Codes
- keine Verknüpfungen zu Code aus anderen Dateien / zu Bibliotheksfunktionen

Linker / Binder

- löst Referenzen zwischen Dateien auf
- kombiniert mit statischen Laufzeit-Bibliotheken
  - z.B. Code für `malloc, printf`
- manche Bibliotheken sind *dynamisch* verknüpft
  - Verknüpfung wird beim Laden in Speicher bzw. zur Laufzeit erstellt

Beispiel: Maschinenbefehl für Speichern

- C-Code

```
*dest = t;
```

  - speichert Wert `t` nach Adresse aus `dest`

- Assembler

```
movq %rax, (%rbx)
```

  - Kopiere einen 8-Byte Wert in den Hauptspeicher
    - *Quad*-Worte in x86-64 Terminologie
  - Operanden

`t:`	Register	`%rax`
`dest:`	Register	`%rbx`
`*dest:`	Speicher	`M[%rbx]`

- Objektcode (x86-Befehlssatz)

```
0x40059e:  48 89 03
```

  - 3-Byte Befehl
  - an Speicheradresse `0x40059e`

Objektcode Disassembler: objdump

```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

[BO15]

- `objdump -d ...`
  - Werkzeug zur Untersuchung des Objektcodes
  - rekonstruiert aus Binärcode den Assemblercode
  - kann auf vollständigem, ausführbaren Programm (`a.out`) oder einer `.o` Datei ausgeführt werden

Was kann „disassembliert“ werden?


```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by Microsoft End User License Agreement

[BO15]

- alles, was als ausführbarer Code interpretiert werden kann
- Disassembler untersucht Bytes und rekonstruiert Assemblerquelle (soweit wie möglich)

x86 Assemblerprogrammierung

- Adressierungsarten
- arithmetische Operationen
- Statusregister
- Umsetzung von Programmstrukturen

Einschränkungen

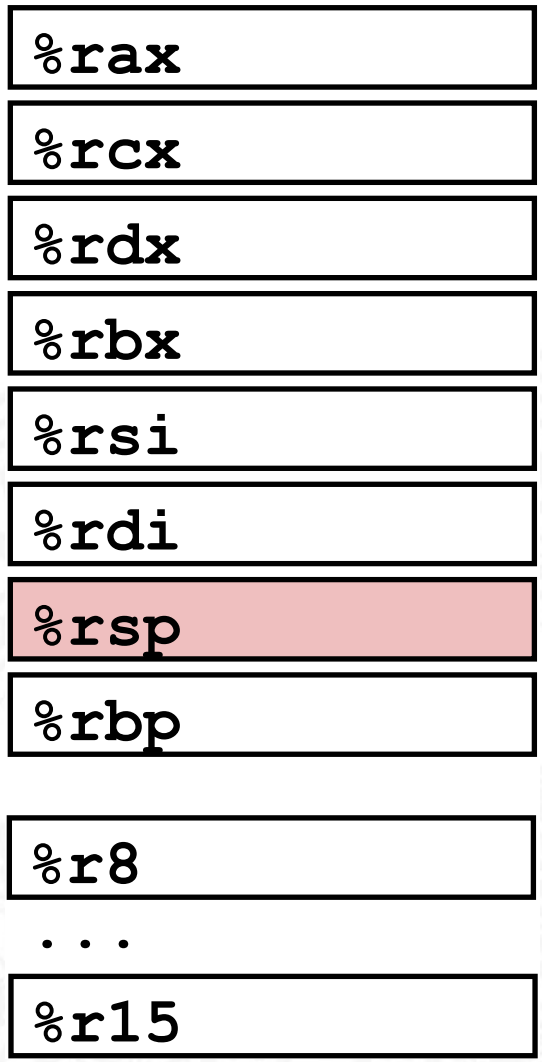
- Beispiele nutzen nur die 64-bit Datentypen `long` bei Linux (unter Windows nur 4-Byte!)
  - x86-64 wird wie 16-Register 64-bit Maschine benutzt (=RISC)
  - CISC Komplexität und Tricks bewusst vermieden
- Beispiele nutzen gcc/as Syntax (vs. Microsoft, Intel)

Grafiken und Beispiele dieses Abschnitts sind aus R.E. Bryant, D.R. O'Hallaron: *Computer systems – A programmers perspective* [BO15], bzw. dem zugehörigen Foliensatz

Datentransfer „move“

- Format: `movq` ⟨*src*⟩`,` ⟨*dst*⟩
- transferiert ein 8-Byte „long“ Wort
- sehr häufige Instruktion

- Typ der Operanden
  - Immediate: Konstante, ganzzahlig
    - wie C-Konstante, aber mit dem Präfix `$`
    - z.B.: `$0x400`, `$-533`
    - codiert mit 1, 2 oder 4 Bytes
  - Register: 16 Ganzzahl-Register
    - `%rsp` (ggf. auch `%rbp`) für spezielle Aufgaben reserviert
    - z.T. Spezialregister für andere Anweisungen
  - Speicher: 8 konsekutive Speicherbytes
    - zahlreiche Adressmodi



movq Operanden-Kombinationen

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

- *Mem-Mem* Kombination nicht möglich

movq: Operanden/Adressierungsarten

- Immediate: `$x` $\rightarrow$ `x`
  - positiver (oder negativer) Integerwert
- Register: `%R` $\rightarrow$ `Reg[R]`
  - Inhalt eines der 16 Universalregister `rax`...`r15`
- Normal: `(R)` $\rightarrow$ `Mem[Reg[R]]`
  - Register `R` spezifiziert die Speicheradresse
  - Beispiel: `movq (%rcx), %rax`
- Displacement: `D(R)` $\rightarrow$ `Mem[Reg[R]+D]`
  - Register `R`
  - Konstantes „Displacement“ D spezifiziert den „offset“
  - Beispiel: `movl 8(%rbp), %rdx`

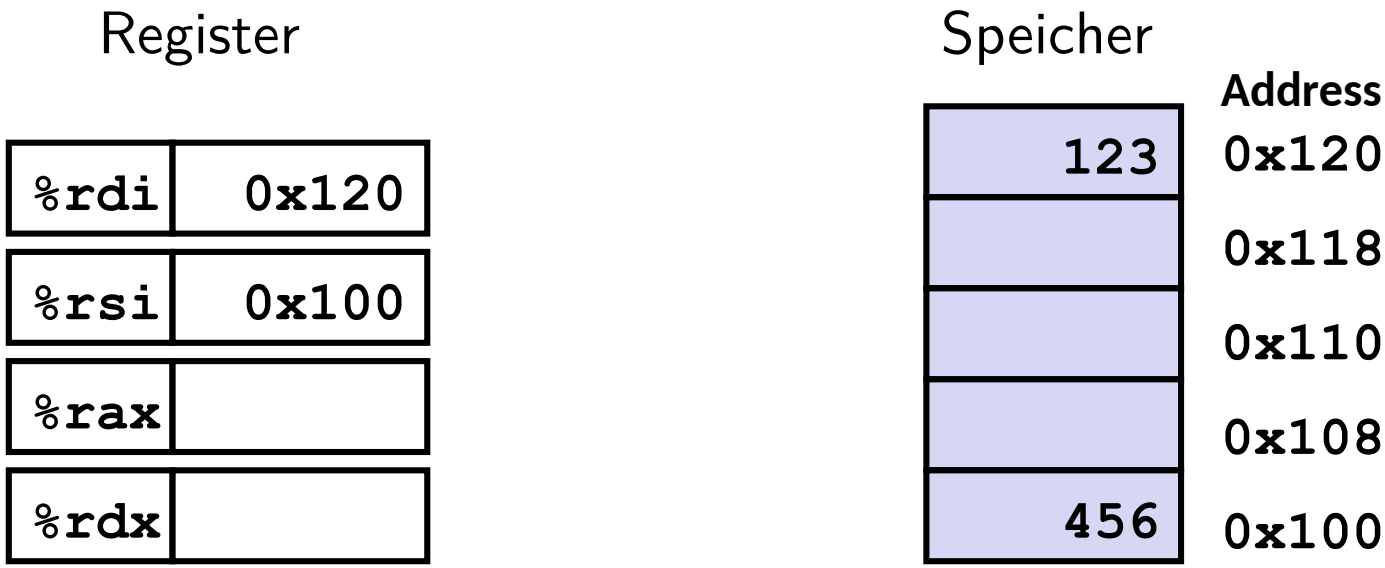
Beispiel: Funktion swap()


```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

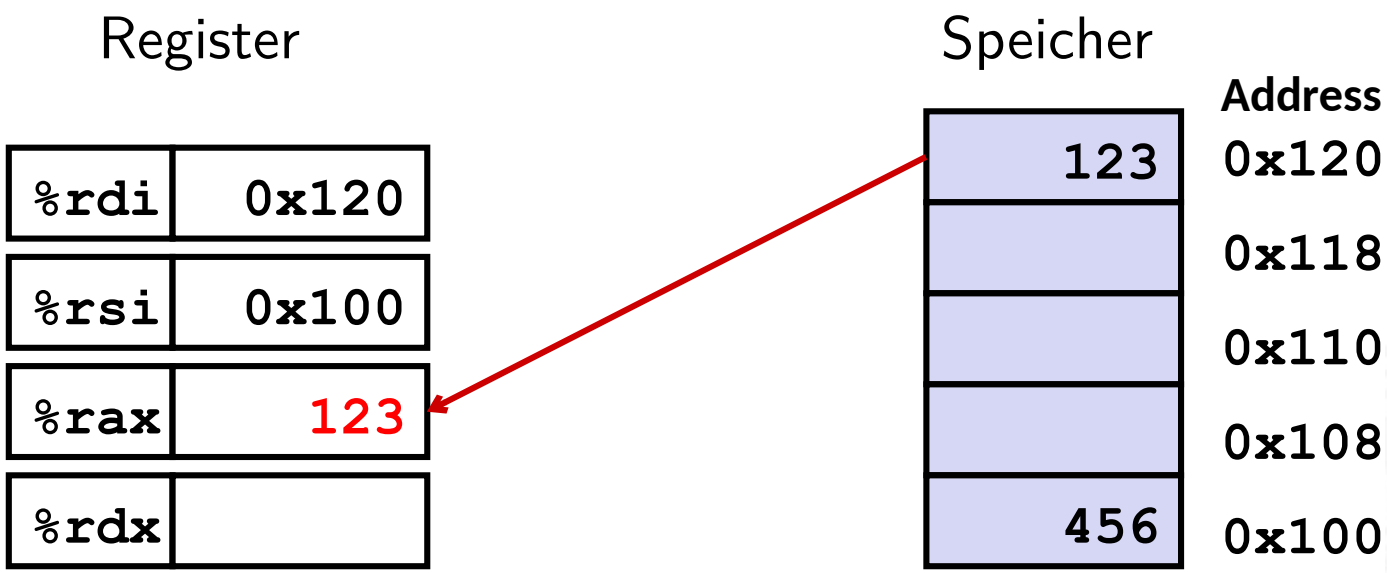
Register	Funktion
%rdi	Argument xp
%rsi	Argument yp
%rax	t0
%rdx	t1

Funktionsweise von swap()



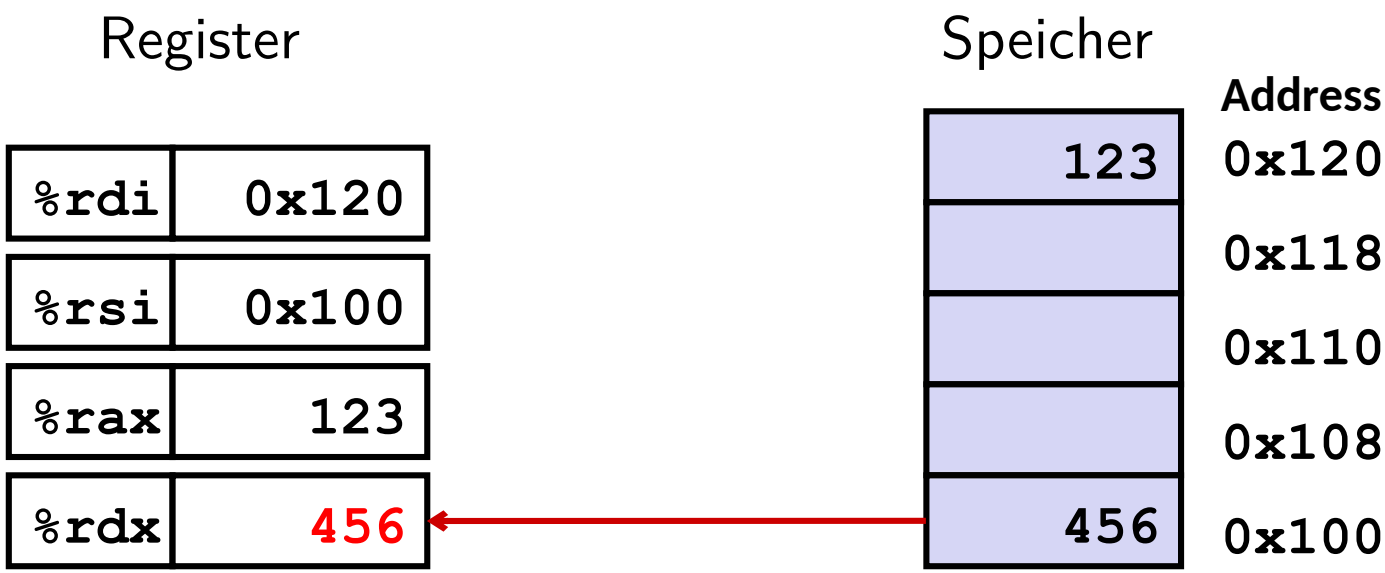

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Funktionsweise von swap()



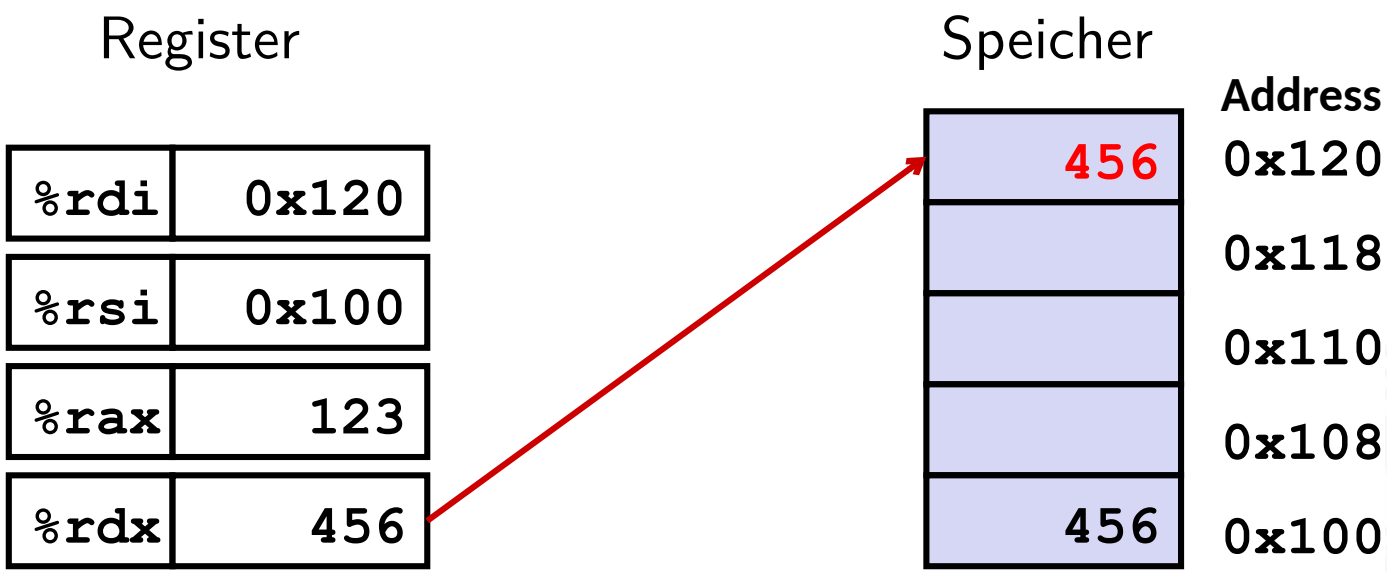

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Funktionsweise von swap()



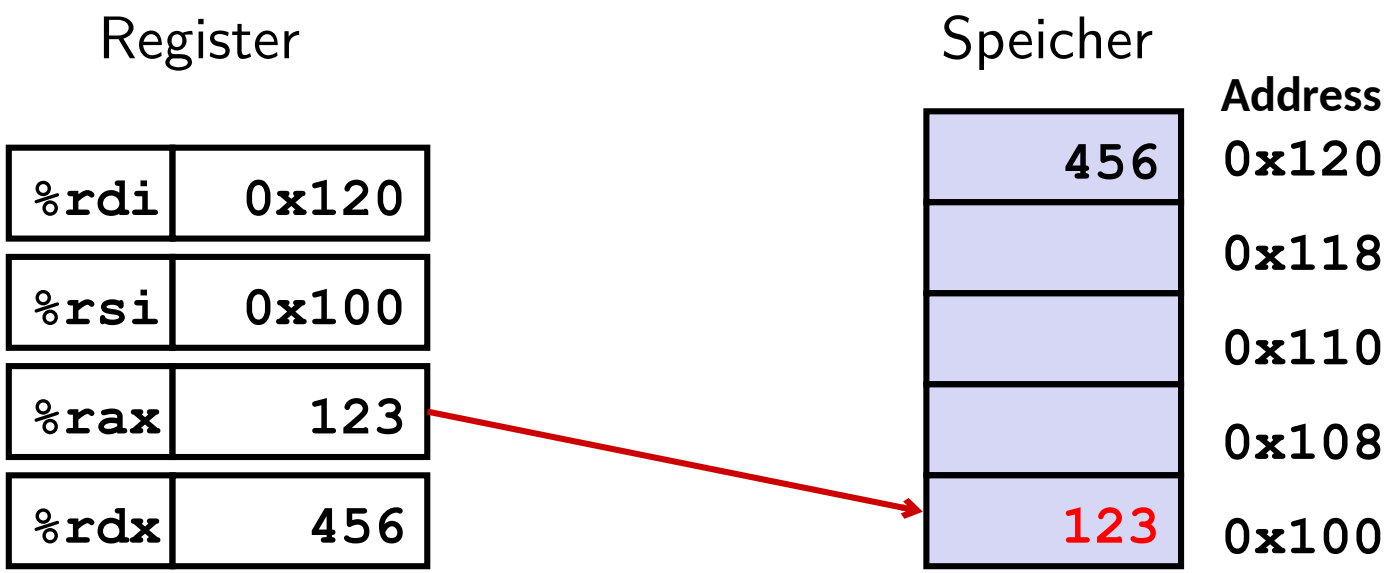

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Funktionsweise von swap()




```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Funktionsweise von swap()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Indizierte Adressierung

- allgemeine Form
  - `Imm(Rb,Ri,S)` → `Mem[Reg[Rb]+S*Reg[Ri]+Imm]`
    - ⟨*Imm*⟩ Offset
    - ⟨*Rb*⟩ Basisregister: eines der 16 Integer-Register
    - ⟨*Ri*⟩ Indexregister: jedes außer `%rsp`
      `%rbp` grundsätzlich möglich, jedoch unwahrscheinlich
    - ⟨*S*⟩ Skalierungsfaktor 1, 2, 4 oder 8

- gebräuchlichste Fälle
  - `(Rb)` → `Mem[Reg[Rb]]`
  - `Imm(Rb)` → `Mem[Reg[Rb] + Imm]`
  - `(Rb,Ri)` → `Mem[Reg[Rb] + Reg[Ri]]`
  - `Imm(Rb,Ri)` → `Mem[Reg[Rb] + Reg[Ri] + Imm]`
  - `(Rb,Ri,S)` → `Mem[Reg[Rb] + S*Reg[Ri]]`

Beispiel: Adressberechnung

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Arithmetische Operationen

- binäre Operatoren

Format		Berechnung	
`addq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩+⟨*src*⟩	
`subq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩-⟨*src*⟩	
`imulq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩*⟨*src*⟩	
`salq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩<<⟨*src*⟩	auch `shlq`
`sarq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩>>⟨*src*⟩	arithmetisch
`shrq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩>>⟨*src*⟩	logisch
`xorq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩^⟨*src*⟩	
`andq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩&⟨*src*⟩	
`orq`	⟨*src*⟩ , ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩|⟨*src*⟩	

Arithmetische Operationen (cont.)

- unäre Operatoren

Format	Berechnung
`incq` ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩+1
`decq` ⟨*dst*⟩	⟨*dst*⟩ = ⟨*dst*⟩-1
`negq` ⟨*dst*⟩	⟨*dst*⟩ = -⟨*dst*⟩
`notq` ⟨*dst*⟩	⟨*dst*⟩ = ~⟨*dst*⟩

- `leaq`-Befehl: *load effective address*

  `leaq` ⟨*src*⟩ , ⟨*dst*⟩

  - Adressberechnung für (späteren) Ladebefehl
  - Speichert die Adressausdruck ⟨*src*⟩ in Register ⟨*dst*⟩
    `Imm(Rb,Ri,S)` → `Reg[Rb]+S*Reg[Ri]+Imm`
  - wird oft von Compilern für arithmetische Berechnung genutzt
    s. Beispiele

Beispiel: arithmetische Operationen


```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

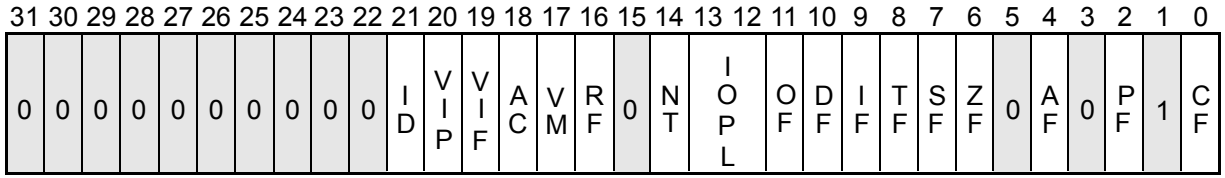
```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

Register	Funktion
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z, t4
%rax	t1, t2, rval
%rcx	t5

Kontrollfluss / Programmstrukturen

- Zustandscodes
  - Setzen
  - Testen
- Ablaufsteuerung
  - Verzweigungen: „If-then-else“
  - Schleifen: „Loop“-Varianten
  - Mehrfachverzweigungen: „Switch“

x86: EFLAGS Register



X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

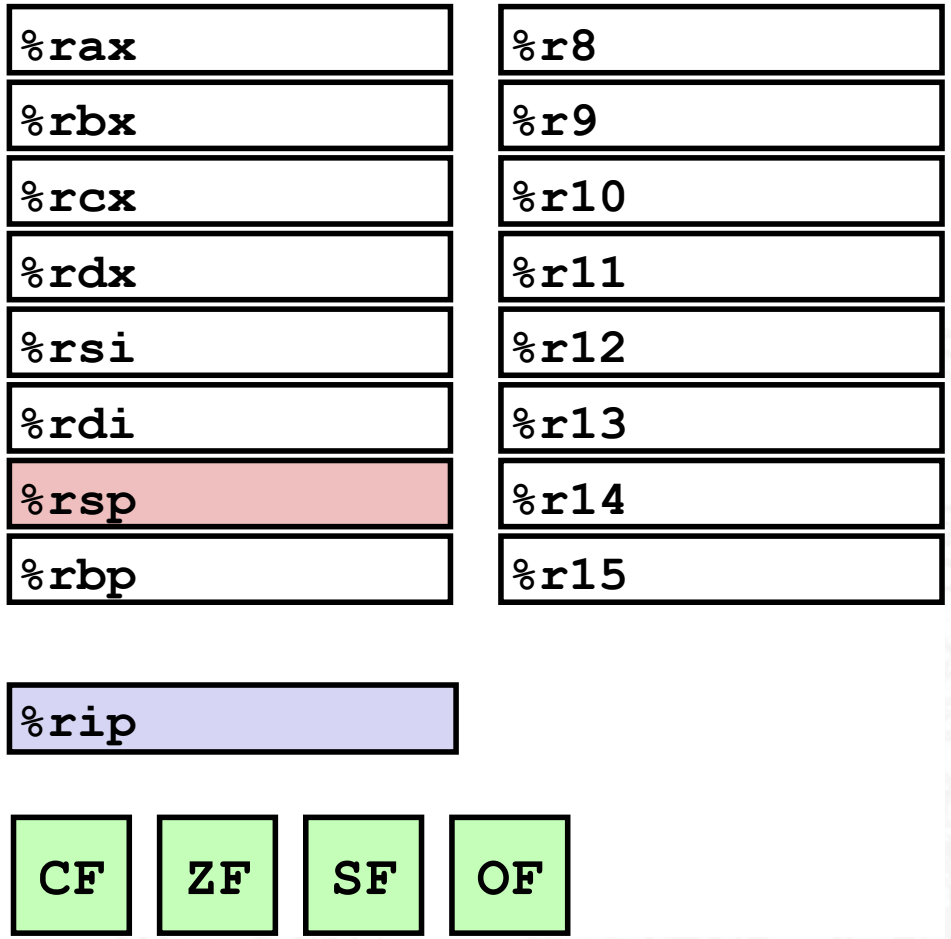
Reserved bit positions. DO NOT USE.
Always set to values previously read.

[IA64]

- x86-64: RFLAGS $\widehat{=}$ EFLAGS, mit „0“ erweitert

Prozessor aus Sicht des Programmierers

- temporäre Daten
  %rax, ...
- Top of Stack
  %rsp
- Programmzähler
  %rip
- Flag-Bits
  CF, ZF, SF, OF



Zustandscodes

- vier relevante „Flags“ im Statusregister EFLAGS/RFLAGS
  - `CF` Carry Flag
  - `ZF` Zero Flag
  - `SF` Sign Flag
  - `OF` Overflow Flag

1. implizite Aktualisierung durch arithmetische Operationen
   - Beispiel: `addq` $\langle src \rangle$, $\langle dst \rangle$ in C: `t=a+b`
   - `CF` höchstwertiges Bit generiert Übertrag: Unsigned-Überlauf
   - `ZF` wenn $t = 0$
   - `SF` wenn $t < 0$
   - `OF` wenn das Zweierkomplement überläuft
     $(a > 0 \;\&\&\; b > 0 \;\&\&\; t < 0) \;||\; (a < 0 \;\&\&\; b < 0 \;\&\&\; t \geq 0)$

Zustandscodes (cont.)

2. explizites Setzen durch Vergleichsoperation
   - Beispiel: `cmpq` $\langle src2 \rangle$, $\langle src1 \rangle$
     wie Berechnung von $\langle src1 \rangle$ - $\langle src2 \rangle$ (`subq` $\langle src2 \rangle$, $\langle src1 \rangle$)
     jedoch ohne Abspeichern des Resultats

   - `CF` höchstwertiges Bit generiert Übertrag
   - `ZF` setzen wenn $src1 = src2$
   - `SF` setzen wenn $(src1 - src2) < 0$
   - `OF` setzen wenn das Zweierkomplement überläuft
     $(a > 0 \;\&\&\; b < 0 \;\&\&\; (a - b) < 0) \;||$
     $(a < 0 \;\&\&\; b > 0 \;\&\&\; (a - b) \geq 0)$

Zustandscodes (cont.)

3. explizites Setzen durch Testanweisung
   - Beispiel: `testq` $\langle src2 \rangle, \langle src1 \rangle$
     wie Berechnung von $\langle src1 \rangle \& \langle src2 \rangle$ (`andq` $\langle src2 \rangle, \langle src1 \rangle$)
     jedoch ohne Abspeichern des Resultats

   $\Rightarrow$ hilfreich, wenn einer der Operanden eine Bitmaske ist

   - ZF setzen wenn $src1 \& src2 = 0$
   - SF setzen wenn $src1 \& src2 < 0$

Zustandscodes lesen: `set...`-Befehle

- Befehle setzen ein einzelnes Byte (LSB) in Universalregister
- die anderen 7-Bytes werden nicht verändert

SetX	Condition	Description
sete	**ZF**	Equal / Zero
setne	**~ZF**	Not Equal / Not Zero
sets	**SF**	Negative
setns	**~SF**	Nonnegative
setg	**~(SF^OF)&~ZF**	Greater (Signed)
setge	**~(SF^OF)**	Greater or Equal (Signed)
setl	**(SF^OF)**	Less (Signed)
setle	**(SF^OF)|ZF**	Less or Equal (Signed)
seta	**~CF&~ZF**	Above (unsigned)
setb	**CF**	Below (unsigned)

Beispiel: Zustandscodes lesen

- ein-Byte Zieloperand (Register, Speicher)
- meist kombiniert mit `movzbl`
  move with zero-extend byte to long
  also Löschen der Bits 31 . . . 8

```
int gt (long x, long y)
{
  return x > y;
}
```

```
cmpq    %rsi, %rdi    # Compare x:y
    setg    %al           # Set when >
    movzbl %al, %eax      # Zero rest of %rax
    ret
```

Sprünge („Jump“): j...-Befehle

- unbedingter- / bedingter Sprung (abhängig von Zustandscode)

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Assembler: Label

- Assemblercode enthält je einen Maschinenbefehl pro Zeile
- normale Programmausführung ist sequenziell
- Befehle beginnen an eindeutig bestimmten Speicheradressen

- Label: symbolische Namen für bestimmte Adressen
  - am Beginn einer Zeile oder vor einem Befehl
  - vom Programmierer / Compiler vergeben
  - als symbolische Adressen für Sprünge verwendet

  - `_max`: global, Beginn der Funktion `max()`
  - `L9:` lokal, nur vom Assembler verwendete interne Adresse

  - Label müssen in einem Programm eindeutig sein

if-Verzweigung / bedingter Sprung


```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Funktion
%rdi	Argument x
%rsi	Argument y
%rax	Rückgabewert

if Übersetzung – goto

- entspricht C Code mit goto

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

- Compilerabhängigkeit

`-fno-if-conversion`

`if` Übersetzung – `goto` (cont.)

- getrennte Code Abschnitte: `then`, `else`
- „passenden“ ausführen
- Codeäquivalent

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

`if` Übersetzung – *conditional move*

- `cmov..`-Befehl `-fif-conversion`
- kein Sprung

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Funktion
%rdi	Argument x
%rsi	Argument y
%rax	Rückgabewert

```
absdiff:
    movq    %rdi, %rax  # x
    subq    %rsi, %rax  # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx  # eval = y-x
    cmpq    %rsi, %rdi  # x:y
    cmovle  %rdx, %rax  # if <=, result = eval
    ret
```

`if` Übersetzung – *conditional move* (cont.)

- \+ keine Sprünge (gut für Pipelining)
- – *beide Ausdrücke werden berechnet*
  - – Performanz, wenn komplizierte Berechnung
  - – Unsicher
  - – Seiteneffekte!
- ▶ Codeäquivalent

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

do ... while Übersetzung

- C Code

```
do
   Body
   while (Test);
```

goto-Version

```
loop:
   Body
   if (Test)
      goto loop
```

- beliebige Folge von C Anweisungen als Schleifenkörper
- Abbruchbedingung ist zurückgelieferter Integer Wert
  - $= 0$ entspricht Falsch: Schleife verlassen
  - $\neq 0$ –"– Wahr: nächste Iteration
- Rückwärtssprung setzt Schleife fort

do ... while Übersetzung (cont.)

- C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

do ... while Übersetzung (cont.)


```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

Register	Funktion
%rdi	Argument x
%rax	Rückgabewert

```
    movl    $0, %eax    #   result = 0
.L2:                    # loop:
    movq    %rdi, %rdx
    andl    $1, %edx    #   t = x & 0x1
    addq    %rdx, %rax  #   result += t
    shrq    %rdi        #   x >>= 1
    jne     .L2         #   if(x) goto loop
    rep; ret
```

while Übersetzung – Sprung zu Test

- C Code

```
while (Test)
  Body
```

Sprung zu *Test* -Og

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

while Übersetzung – Sprung zu Test (cont.)

- C Code

```
long pcount_while
   (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Sprung zu *Test*

```
long pcount_goto_jtm
   (unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

while Übersetzung – do ... while

- C Code -01

```
while (Test)
  Body
```

do ... while Äquivalent

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

goto-Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

while Übersetzung – do ... while (cont.)

- C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

do ... while

```
long pcount_goto_dw
   (unsigned long x) {
   long result = 0;
   if (!x) goto done;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
 done:
   return result;
}
```

for Übersetzung

- mehrstufige Übersetzung: `for ...` → `while` → ...

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update ;
}
```

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

for Übersetzung (cont.)

- C Code

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  if (!(i < WSIZE))
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;
  if (i < WSIZE)
    goto loop;
 done:
  return result;
}
```

Init

!Test

Body

Update

Test

switch Mehrfachverzweigungen

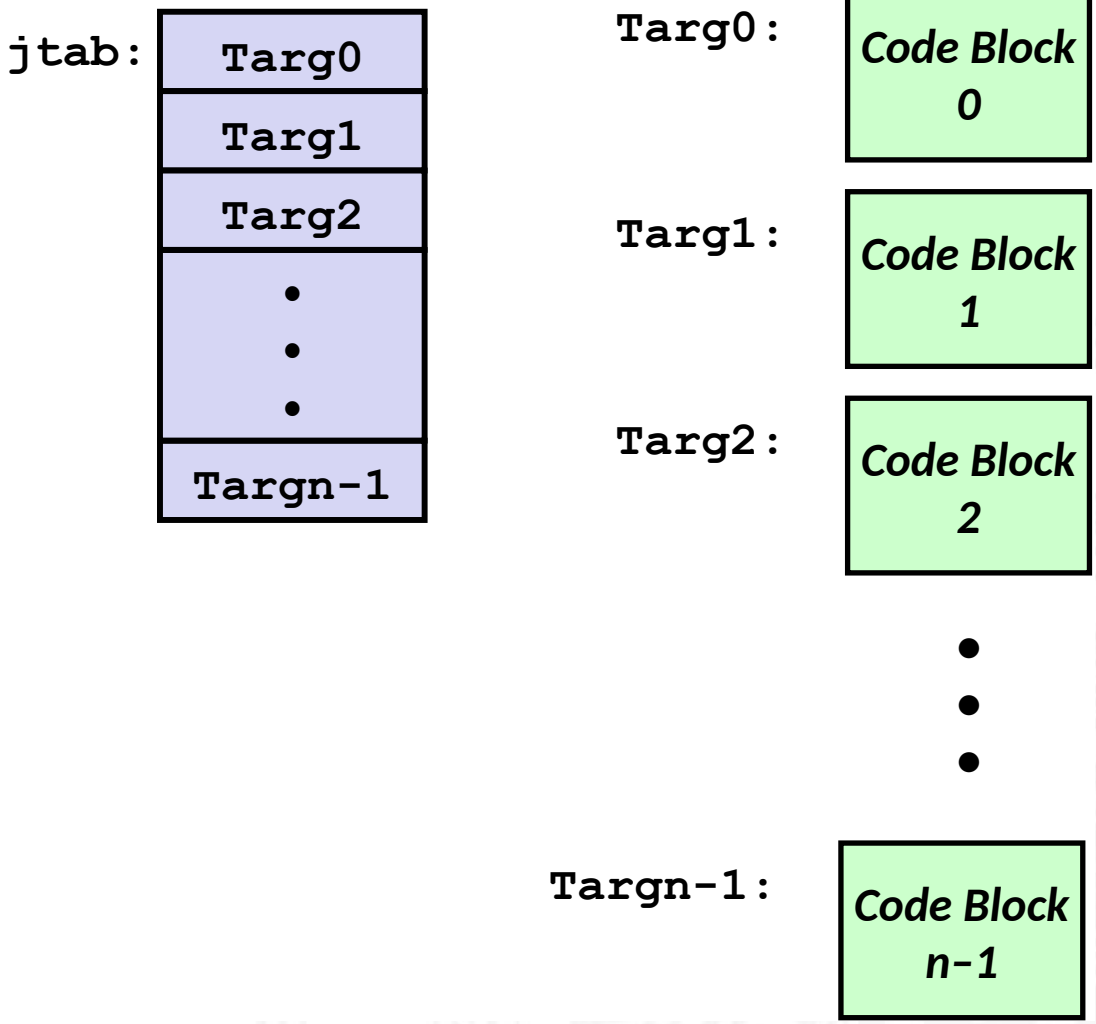
- Implementierungsoptionen
  1. Folge von „If-then-else“
     - \+ gut bei wenigen Alternativen
     - – langsam bei vielen Fällen
  2. Sprungtabelle „Jump Table“
     - Vermeidet einzelne Abfragen
     - möglich falls Alternativen kleine ganzzahlige Konstanten sind
  - Compiler (gcc) wählt eine der beiden Varianten entsprechend der Fallstruktur

```
long my_switch
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Sprungtabelle


```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
     • • •
  case val_n-1:
    Block n-1
}
```

```
goto *JTab[x];
```



- Vorteil: k-fach Verzweigung in $\mathcal{O}(1)$ Operationen

Sprungtabelle (cont.)

C Code

```
long my_switch(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Sprungtabelle

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
my_switch:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # use default
    jmp     *.L4(,%rdi,8)   # goto *Jtab[x]
```

Register	Funktion
%rdi	Arg. x
%rsi	Arg. y
%rdx	Arg. z

Sprungtabelle (cont.)

- Compiler erzeugt Code für jeden `case` Zweig
  - je ein Label am Start der Zweige: `.L8`, `.L3`, `.L5`...
  - werden dann vom Assembler/Linker in Adressen umgesetzt
- Tabellenstruktur
  - jedes Ziel benötigt 4 Bytes
  - Basisadresse bei `.L4`
- Sprünge
  - Direkt: `jmp .L8`
  - Indirekt: `jmp *.L4(,%rdi,8)`
    - Start der Sprungtabelle: `.L4`
    - Register `%rdi` speichert x
    - Skalierungsfaktor 8 für Tabellenoffset
    - Sprungziel: effektive Adresse `.L4` $+ x \times 8$

Sprungtabelle (cont.)


```
.section    .rodata
  .align 8
.L4:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L5
    w = y/z;
    /* Fall Through */
case 3:       // .L9
    w += z;
    break;
case 5:
case 6:       // .L7
    w -= z;
    break;
default:      // .L8
    w = 2;
}
```

Mechanismen für Prozeduraufrufe

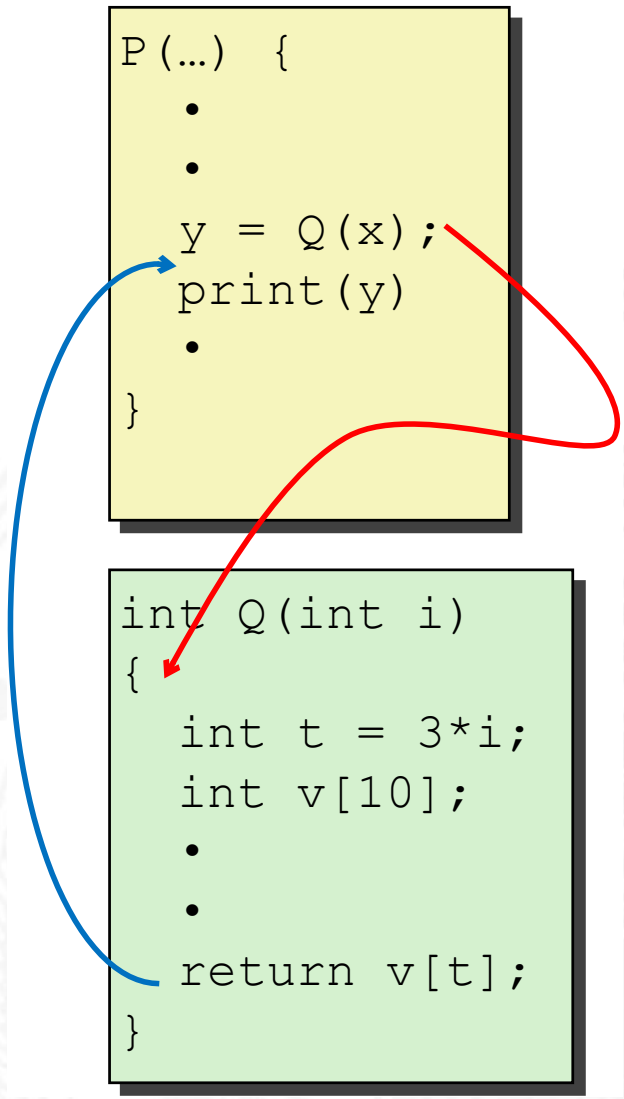
- Kontrollübergabe
  - zu Unterprogrammcode
  - zurück zu Aufruf
- Datenübergabe
  - Argumente
  - Rückgabewert
- Speicherverwaltung
  - Allokation während der Ausführung
  - Freigabe nach `return`

```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

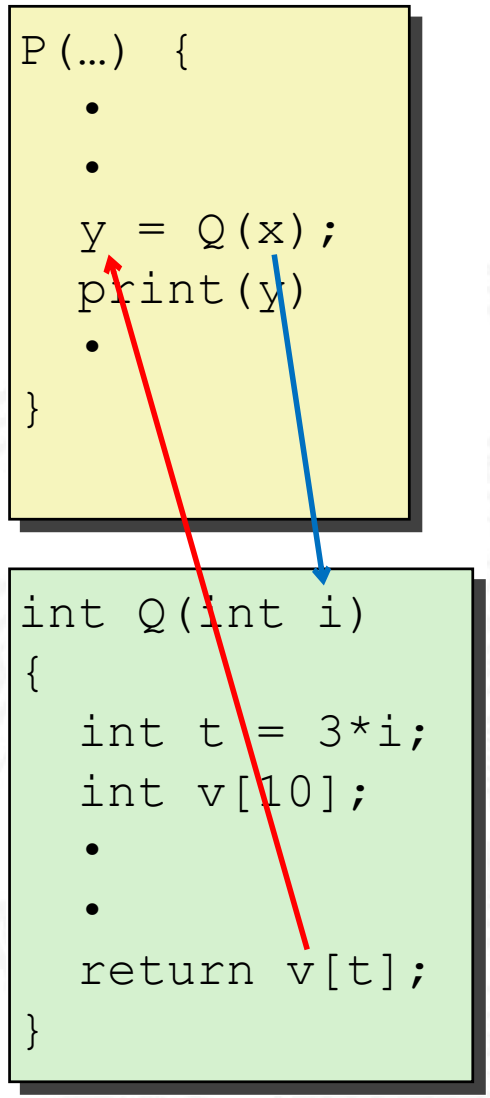
Mechanismen für Prozeduraufrufe

- Kontrollübergabe
  - zu Unterprogrammcode
  - zurück zu Aufruf
- Datenübergabe
  - Argumente
  - Rückgabewert
- Speicherverwaltung
  - Allokation während der Ausführung
  - Freigabe nach `return`



Mechanismen für Prozeduraufrufe

- Kontrollübergabe
  - zu Unterprogrammcode
  - zurück zu Aufruf
- Datenübergabe
  - Argumente
  - Rückgabewert
- Speicherverwaltung
  - Allokation während der Ausführung
  - Freigabe nach `return`



Mechanismen für Prozeduraufrufe

- Kontrollübergabe
  - zu Unterprogrammcode
  - zurück zu Aufruf
- Datenübergabe
  - Argumente
  - Rückgabewert
- Speicherverwaltung
  - Allokation während der Ausführung
  - Freigabe nach `return`

```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

Mechanismen für Prozeduraufrufe

- Kontrollübergabe
  - zu Unterprogrammcode
  - zurück zu Aufruf
- Datenübergabe
  - Argumente
  - Rückgabewert
- Speicherverwaltung
  - Allokation während der Ausführung
  - Freigabe nach `return`

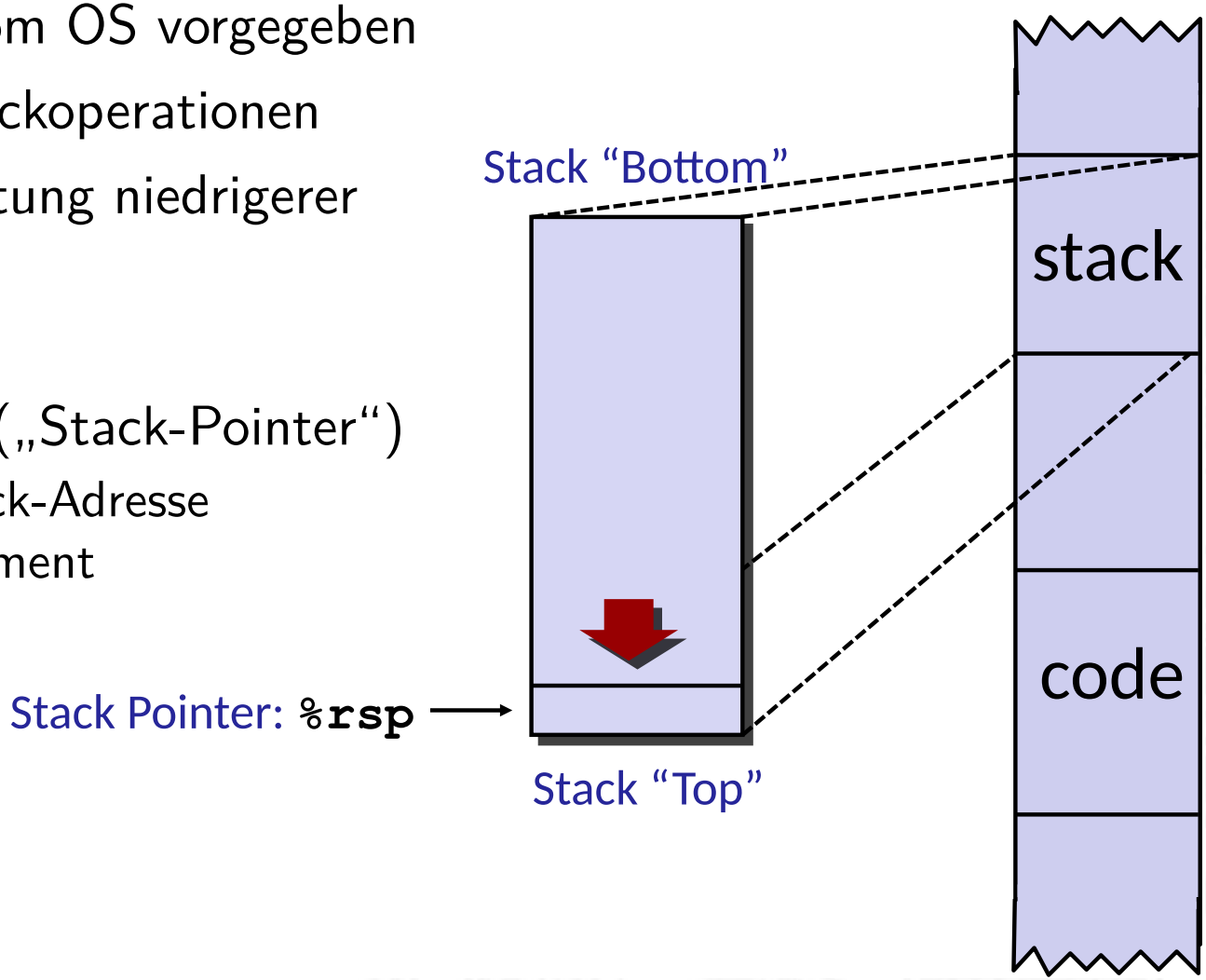
```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

Stack (Kellerspeicher)

- Speicherregion
- Startadresse vom OS vorgegeben
- Zugriff mit Stackoperationen
- wächst in Richtung niedrigerer Adressen

- Register `%rsp` („Stack-Pointer“)
  - aktuelle Stack-Adresse
  - oberstes Element



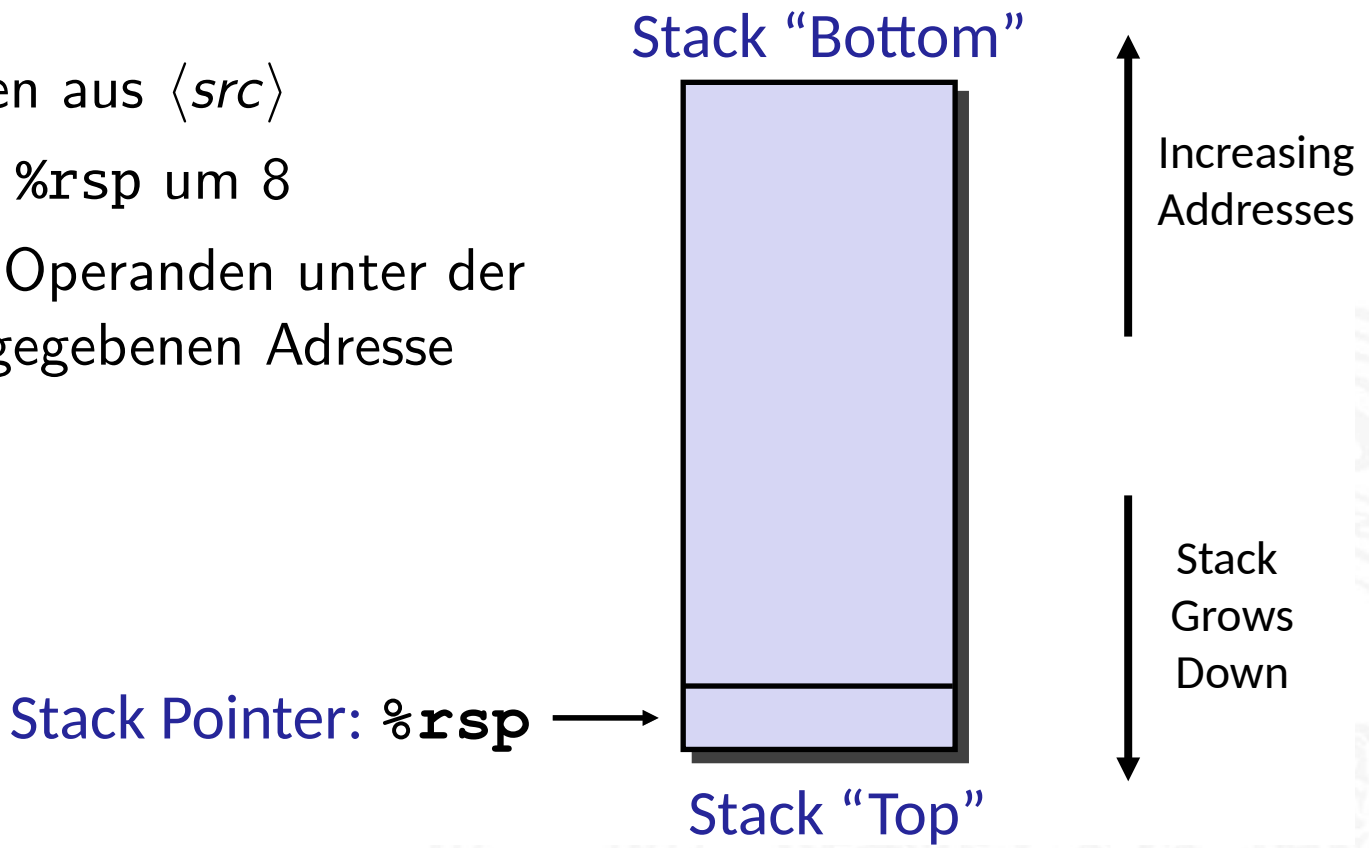
Stack (Kellerspeicher) (cont.)

- Implementierung von Funktionen/Prozeduren
  - Speicherplatz für Aufruf-Parameter
  - Speicherplatz für lokale Variablen
  - Rückgabe der Funktionswerte
  - auch für rekursive Funktionen (!)
- mehrere Varianten/Konventionen
  - Parameterübergabe in Registern
  - „Caller-Save“
  - „Callee-Save“
  - Kombinationen davon
  - Aufruf einer Funktion muss deren Konvention berücksichtigen

Stack: Push

`pushq` ⟨*src*⟩

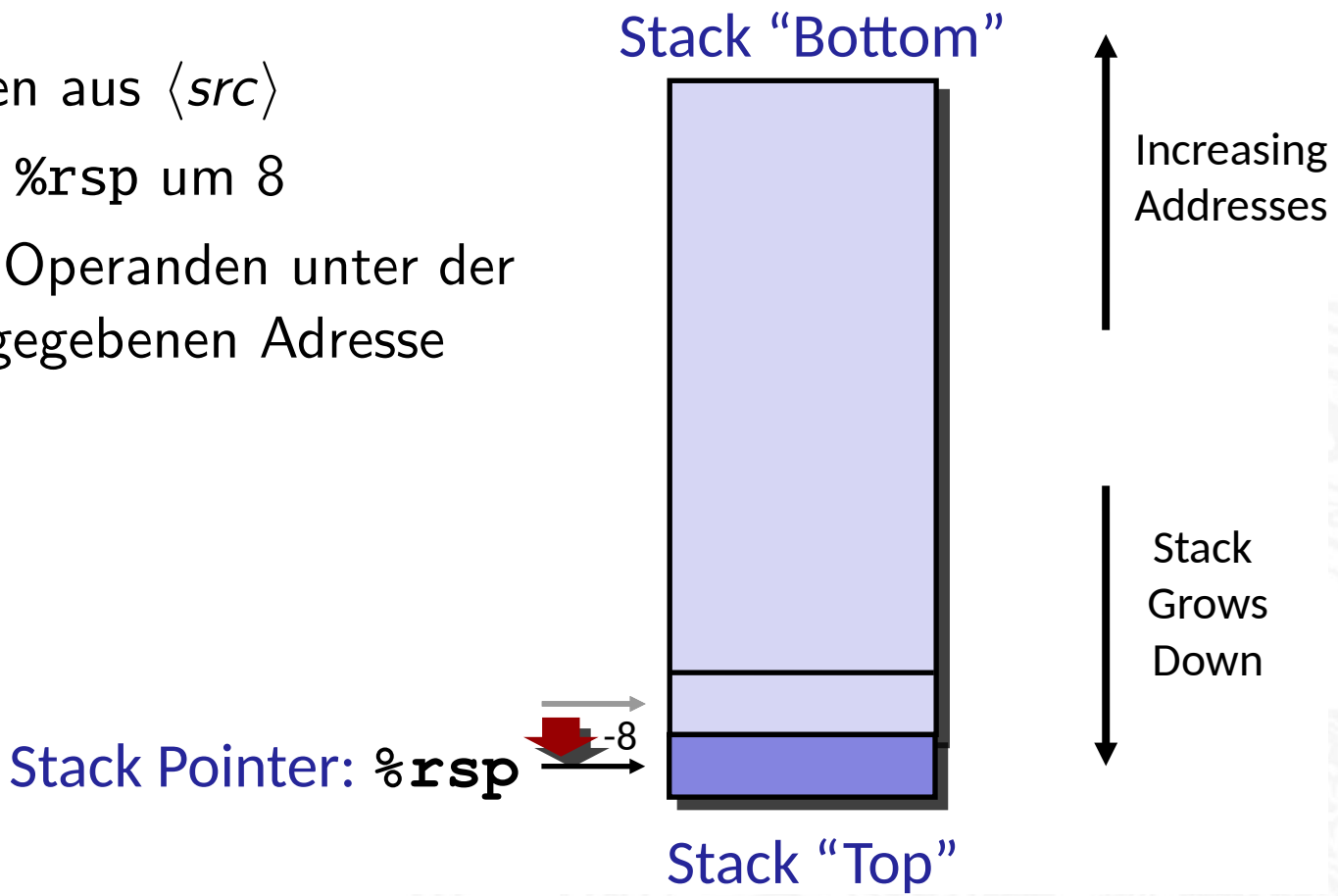
- holt Operanden aus ⟨*src*⟩
- dekrementiert `%rsp` um 8
- speichert den Operanden unter der von `%rsp` vorgegebenen Adresse



Stack: Push

`pushq` ⟨*src*⟩

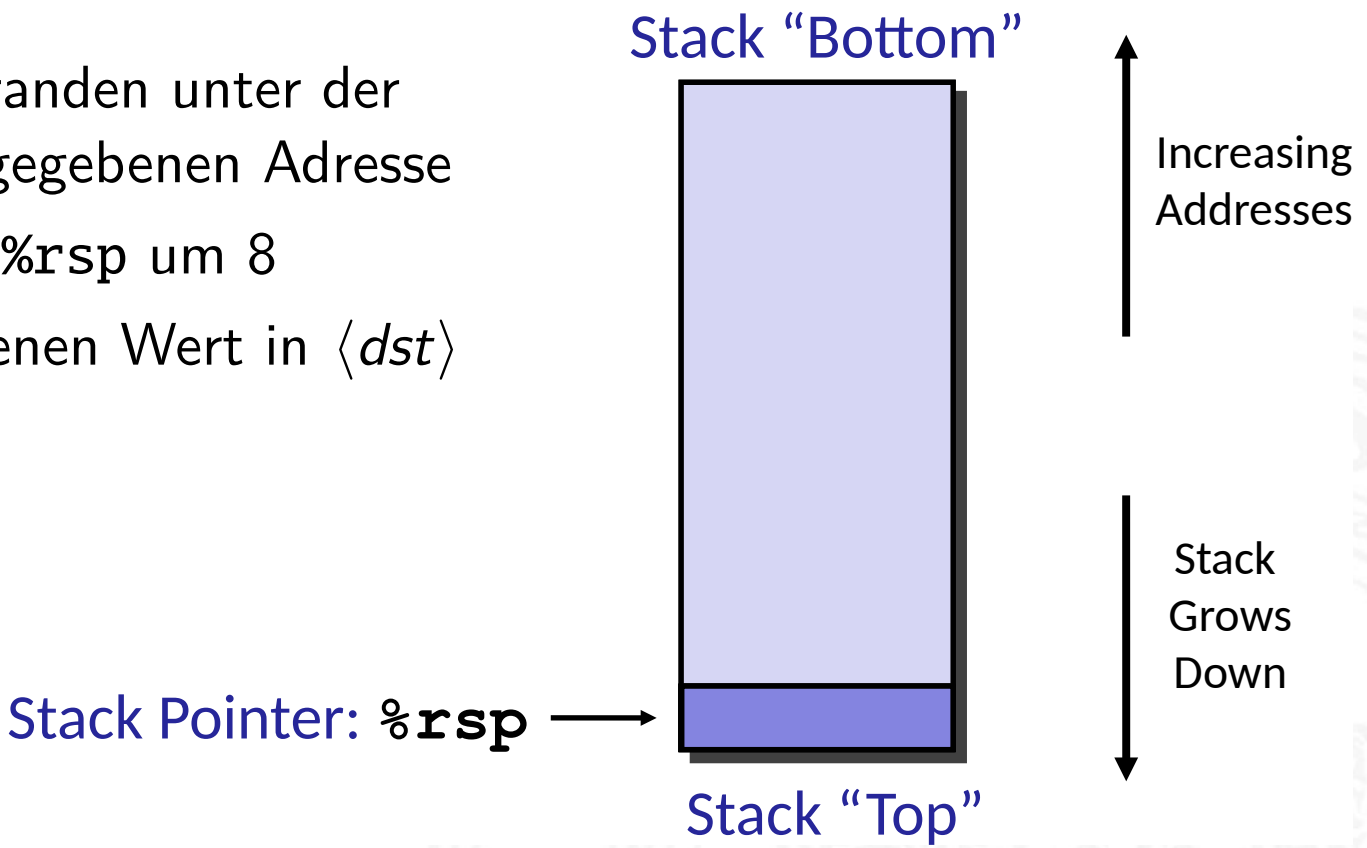
- holt Operanden aus ⟨*src*⟩
- dekrementiert `%rsp` um 8
- speichert den Operanden unter der von `%rsp` vorgegebenen Adresse



Stack: Pop

`popq` ⟨*dst*⟩

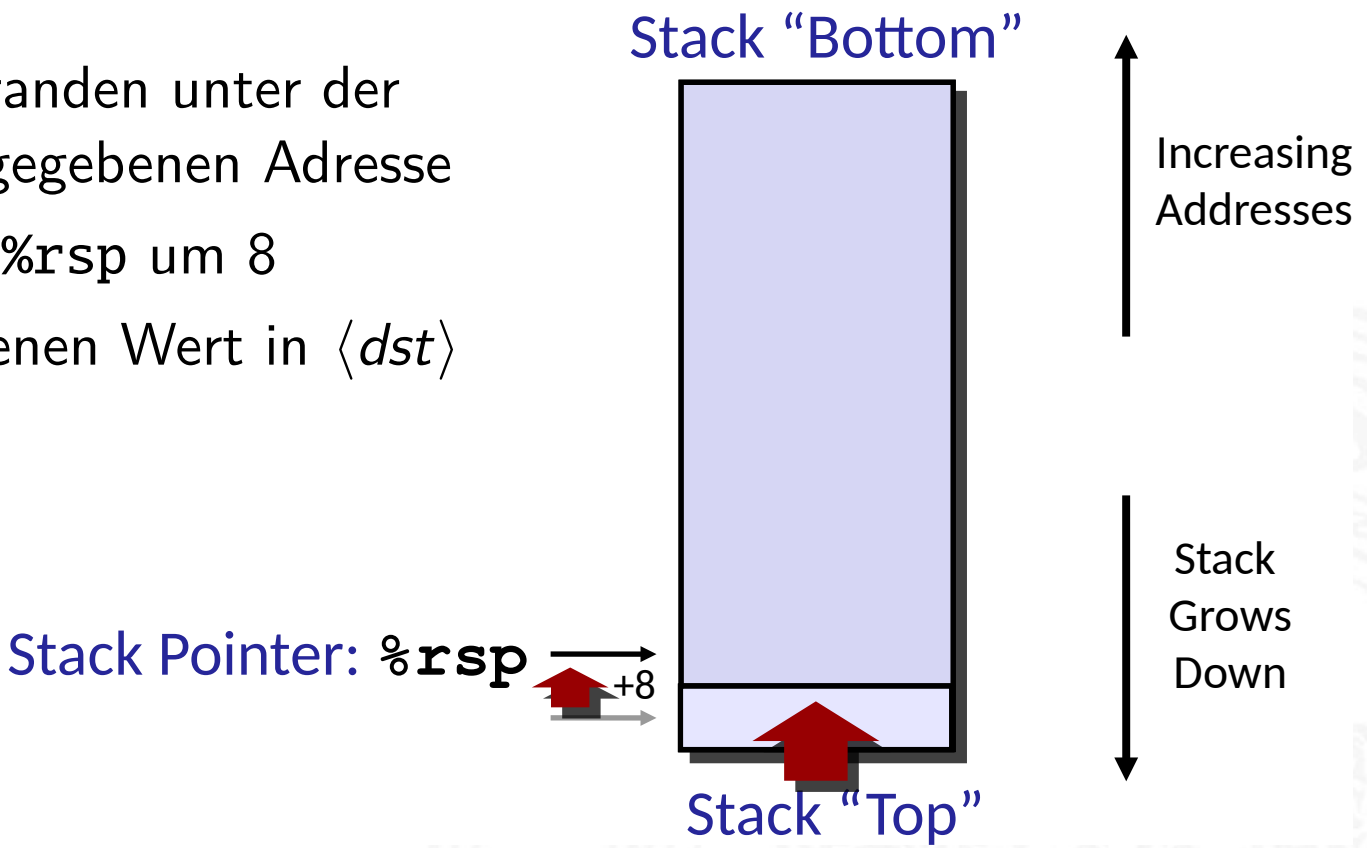
- liest den Operanden unter der von `%rsp` vorgegebenen Adresse
- inkrementiert `%rsp` um 8
- schreibt gelesenen Wert in ⟨*dst*⟩



Stack: Pop

`popq` ⟨*dst*⟩

- liest den Operanden unter der von `%rsp` vorgegebenen Adresse
- inkrementiert `%rsp` um 8
- schreibt gelesenen Wert in ⟨*dst*⟩



Funktions-/Prozeduraufruf

- x86 ist CISC: spezielle Maschinenbefehle für Funktionsaufruf
  - `call` zum Aufruf einer Funktion
  - `ret` zum Rücksprung aus der Funktion
  - beide Funktionen ähnlich `jmp`: `rip` wird modifiziert
  - Parameterübergabe über Register und/oder Stack
- Stack zur Unterstützung von `call` und `ret`
- Register mit Spezialaufgaben
  - `%rsp` „stack-pointer“: Speicheradresse des top-of-stack
  - `%rbp` „base-pointer“: Speicheradresse des aktuellen Frame (s.u.)
- Prozeduraufruf: `call` ⟨*label*⟩
  - Rücksprungadresse auf Stack („Push“)
  - Sprung zu ⟨*label*⟩
- Wert der Rücksprungadresse
  - Adresse der auf den `call` folgenden Anweisung
- Rücksprung `ret`
  - Rücksprungadresse vom Stack („Pop“)
  - Sprung zu dieser Adresse

Codebeispiel Unterprogrammaufruf

```
void multstore(long x, long y, long
*dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

```
long mult2(long a, long b)
{
  long s = a * b;
  return s;
}
```

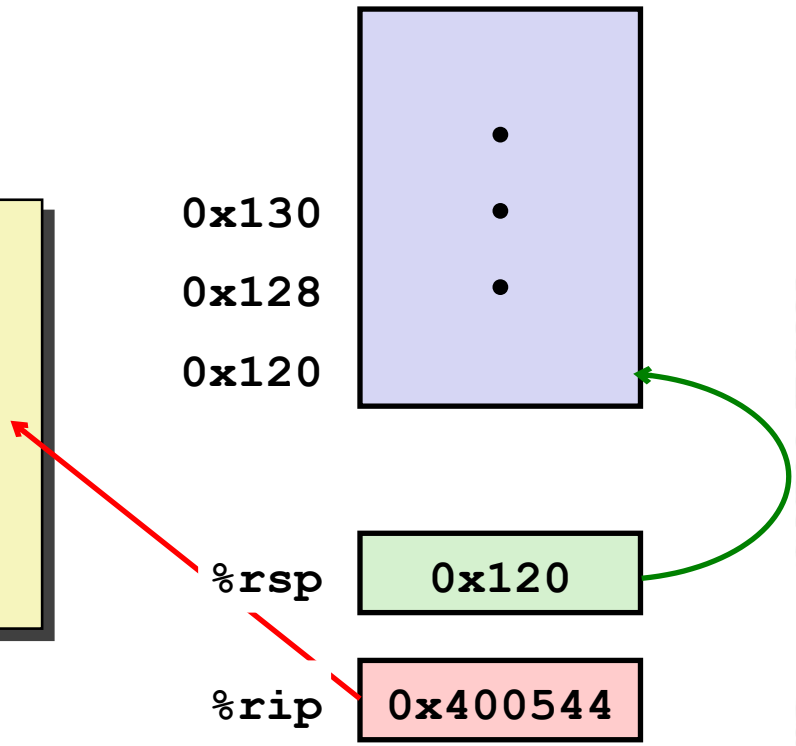
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax   # a
  400553:  imul   %rsi,%rax   # a * b
  400557:  retq               # Return
```

Kontrollübergabe

- Prozeduraufruf `callq`

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

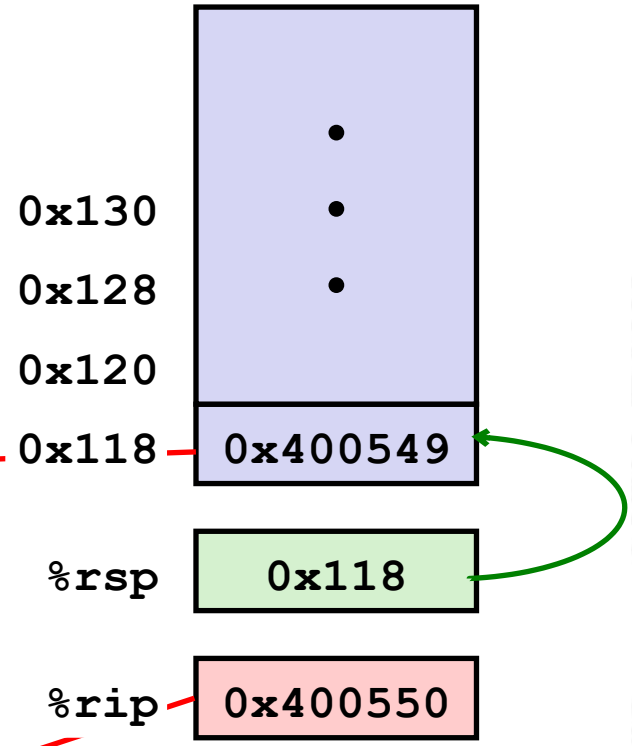


Kontrollübergabe (cont.)

- Rücksprungadresse auf Stack
- Programmzähler setzen `%rip`

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

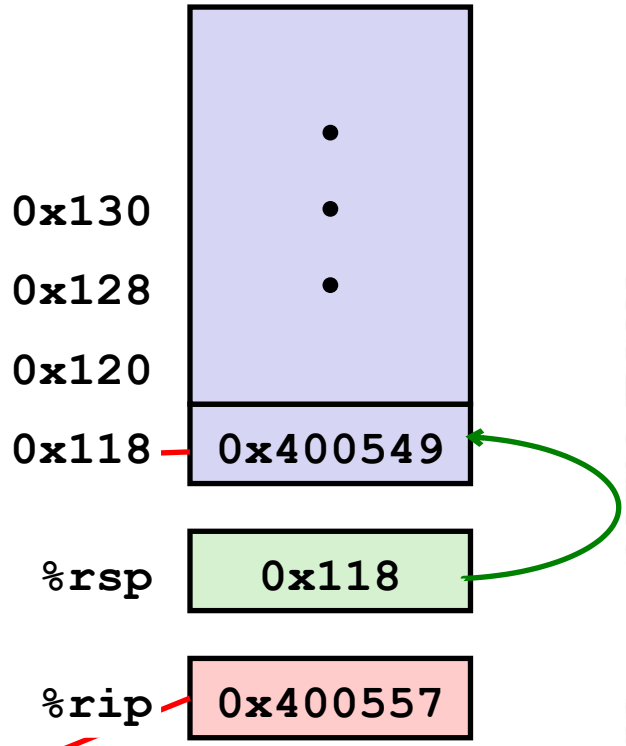


Kontrollübergabe (cont.)

- Rücksprung `retq`

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

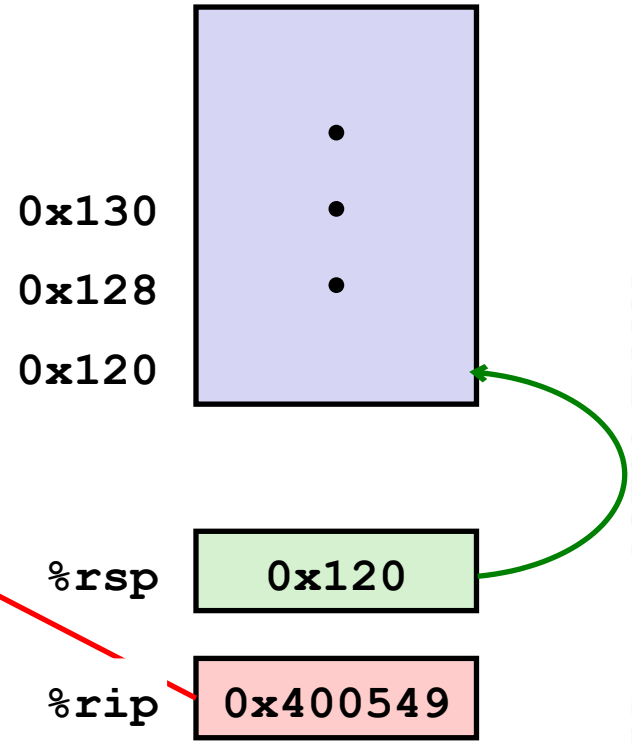


Kontrollübergabe (cont.)

- Rücksprungadresse vom Stack
- Programmzähler setzen `%rip`

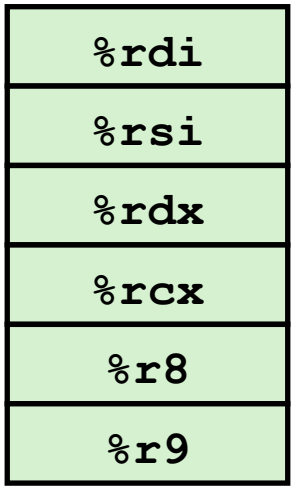
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

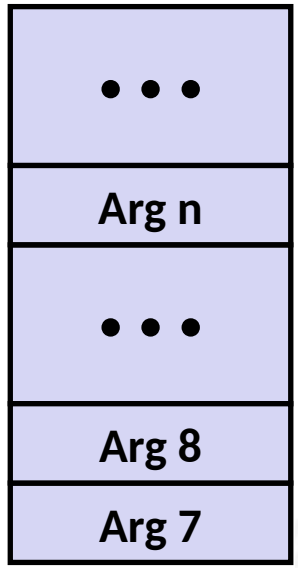


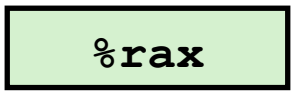
Datenübergabe

- Register



Stack





- Konvention
  - die ersten 6 Argumente: Register `%rdi`, `%rsi`...
  - Stack wenn notwendig
  - Rückgabewert: Register `%rax`

Datenübergabe (cont.)

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: mov    %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)      # Save at dest
  • • •
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

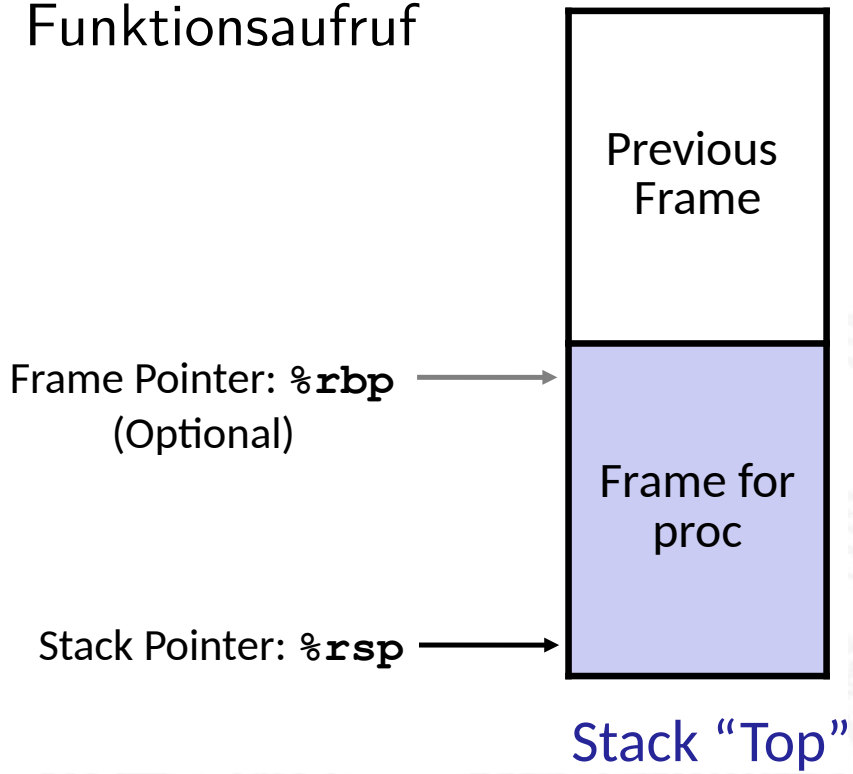
```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax   # a
  400553:  imul   %rsi,%rax   # a * b
  # s in %rax
  400557:  retq               # Return
```

Stack-basierende Programmierung

- für alle Programmiersprachen, die Rekursion unterstützen
  - C, Pascal, Java, Lisp usw.
    - Code muss „reentrant“ sein
    - erlaubt mehrfache, simultane Instanziierungen einer Prozedur
  - benötigt Platz, um den Zustand jeder Instanziierung zu speichern
    - ggf. Argumente
    - lokale Variable(n)
    - Rücksprungadresse
- Stack-„Prinzip“
  - dynamischer Zustandsspeicher für Aufrufe
  - zeitlich limitiert: vom Aufruf (`call`) bis zum Rücksprung (`ret`)
  - aufgerufenes Unterprogramm („Callee“) wird vor dem aufrufendem Programm („Caller“) beendet
- Stack-„Frame“
  - der Bereich/Zustand einer einzelnen Prozedur-Instanziierung

Stack-Frame

- „Closure“: alle Daten für einen Funktionsaufruf
- Daten
  - Rücksprungadresse
  - ggf. lokale Variablen
  - ggf. temporäre Daten



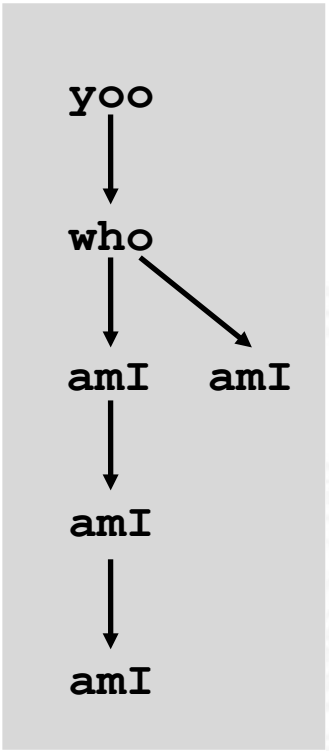
- Speicherverwaltung durch Funktion
  - bei Aufruf wird Stack gefüllt: „Set-up“ Code
  - bei Return wieder freigegeben: „Finish“ Code

Beispiel: Prozeduraufrufe

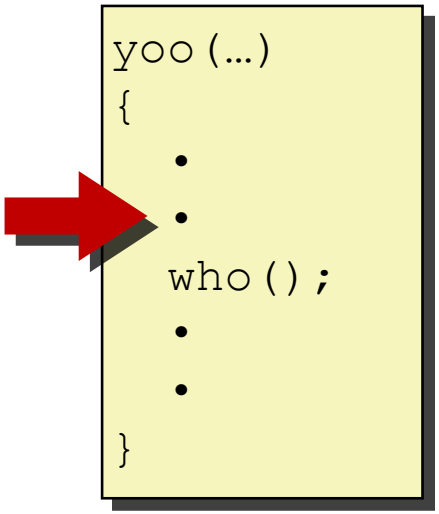

```
yoo(...)
{
   •
   •
   who();
   •
   •
}
```

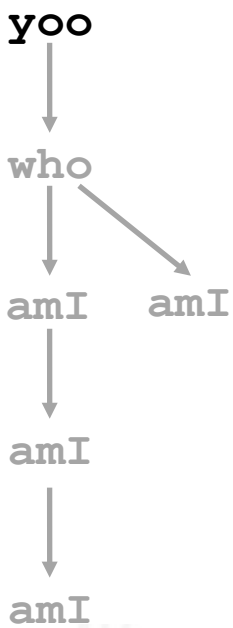
```
who(...)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

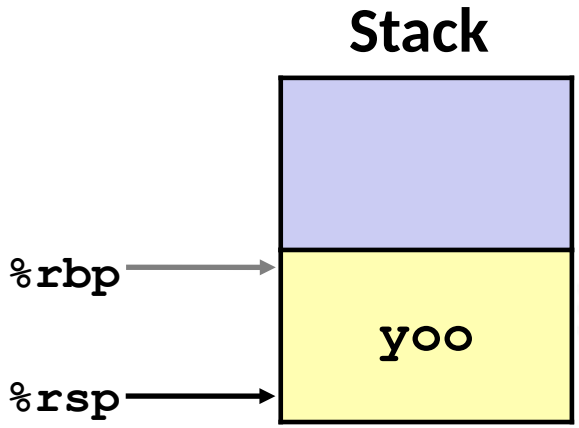
```
amI(...)
{
   •
   •
   amI();
   •
   •
}
```



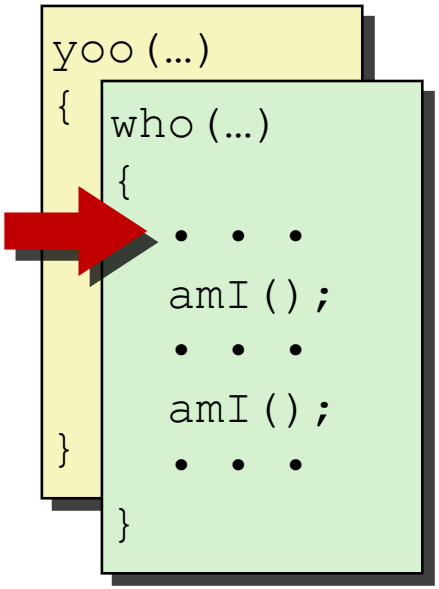
Beispiel: Prozeduraufrufe (cont.)

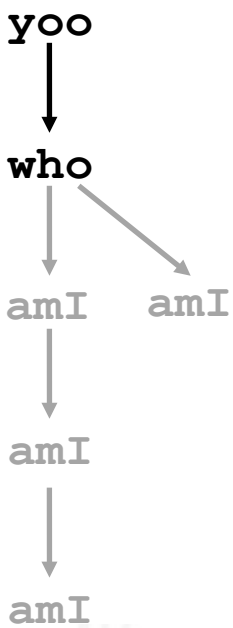


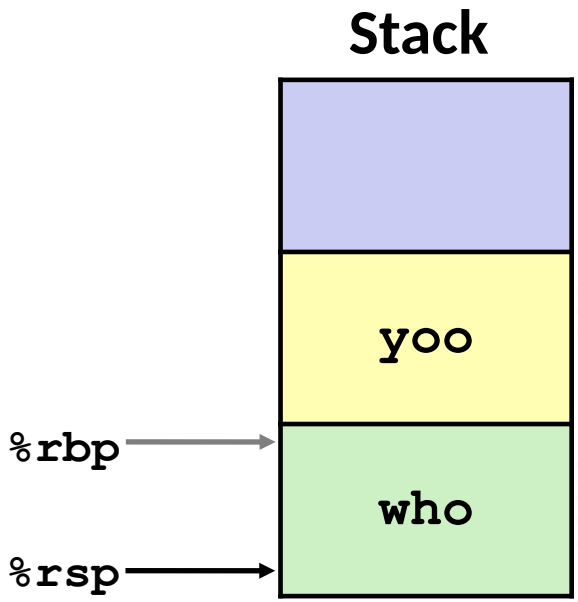




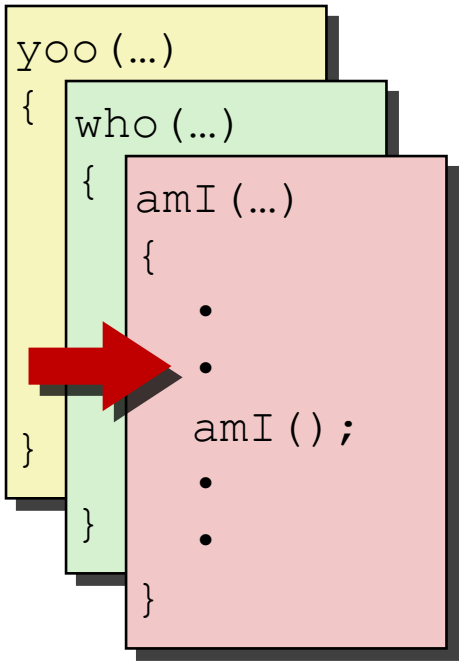
Beispiel: Prozeduraufrufe (cont.)

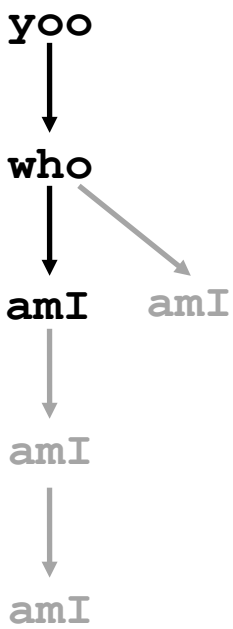


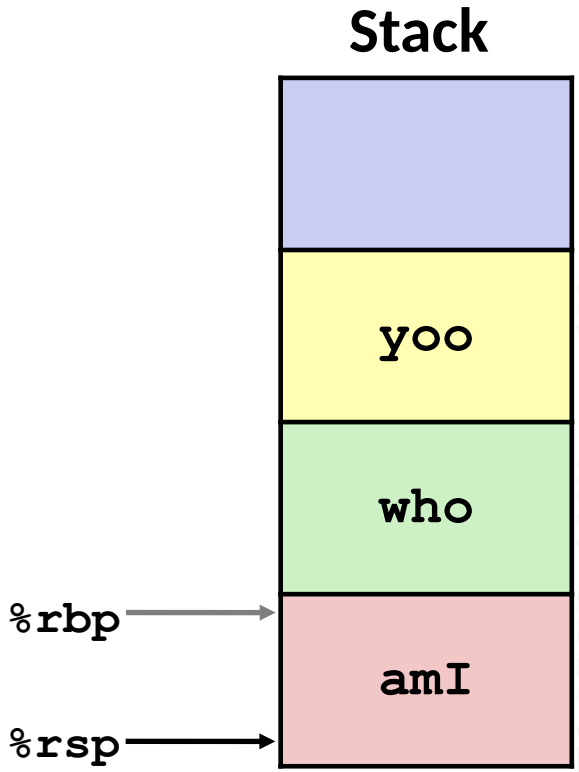




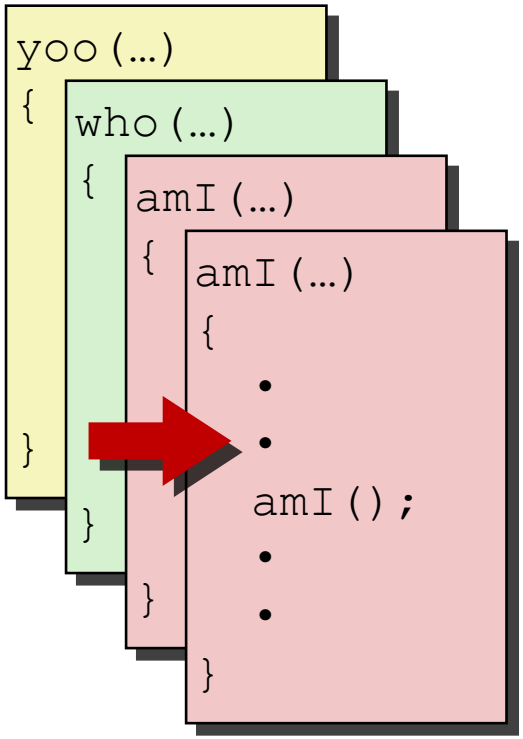
Beispiel: Prozeduraufrufe (cont.)

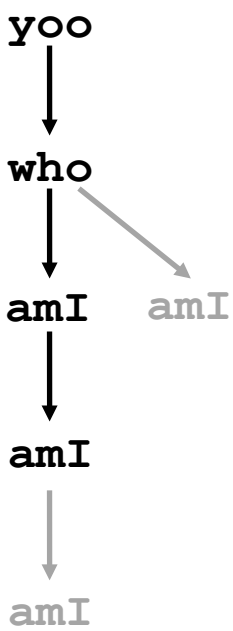


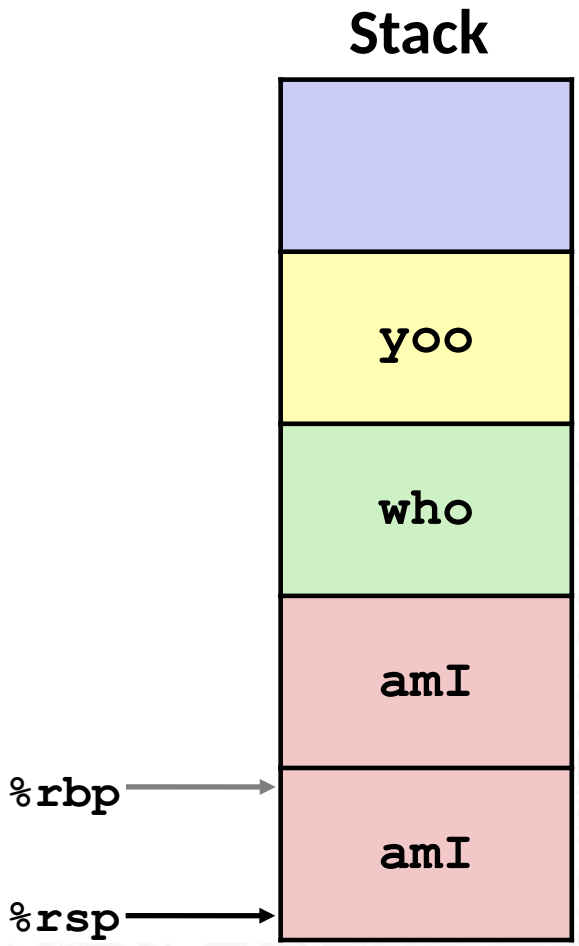




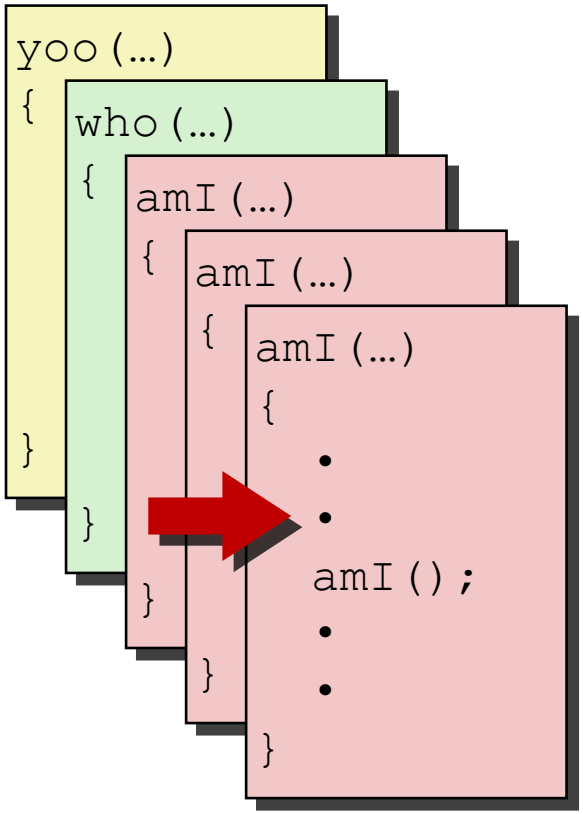
Beispiel: Prozeduraufrufe (cont.)

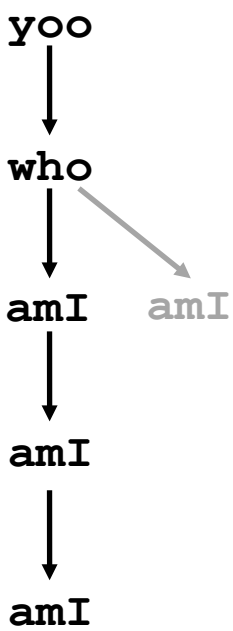


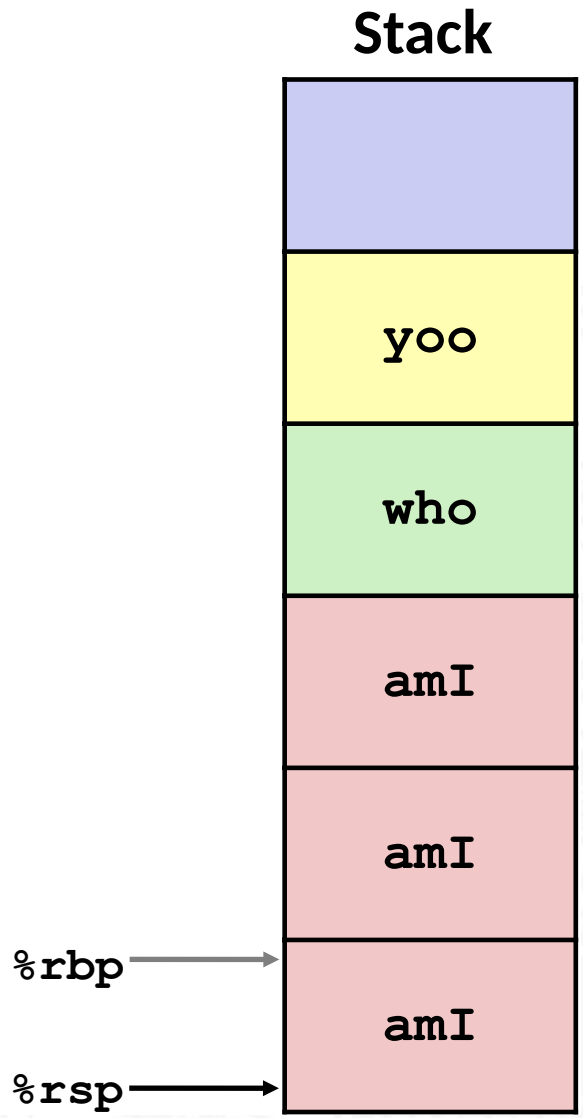




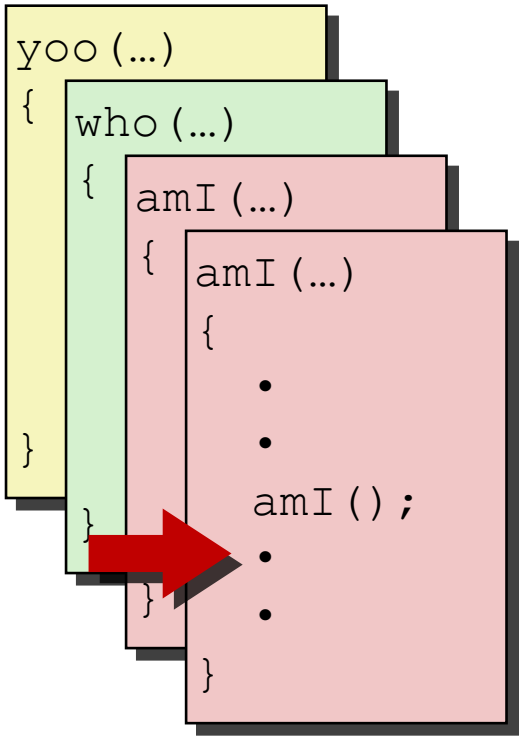
Beispiel: Prozeduraufrufe (cont.)

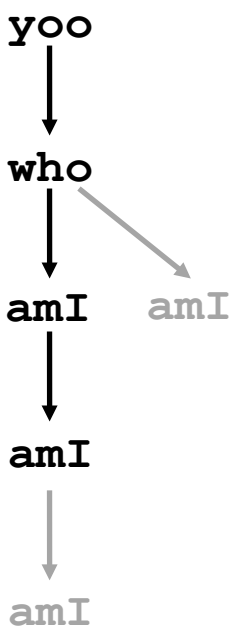


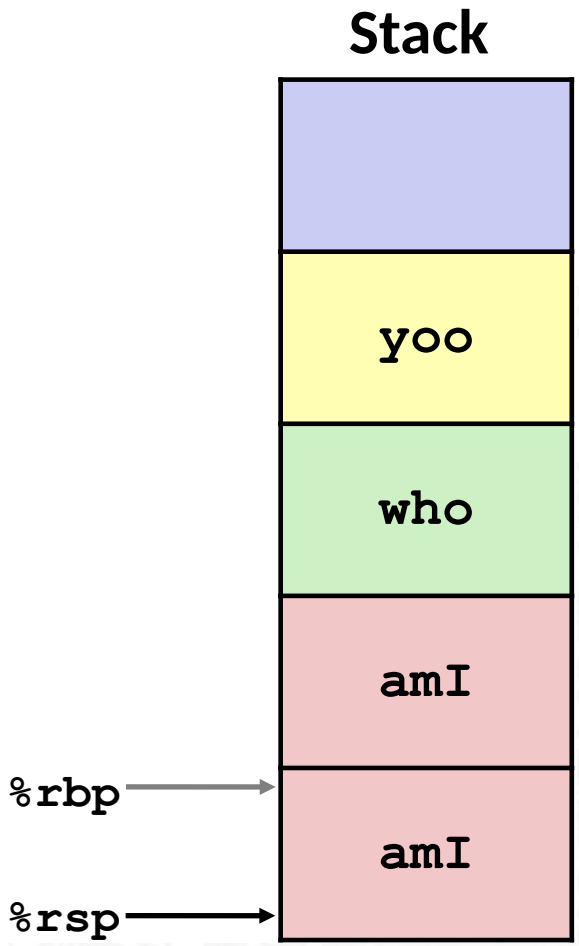




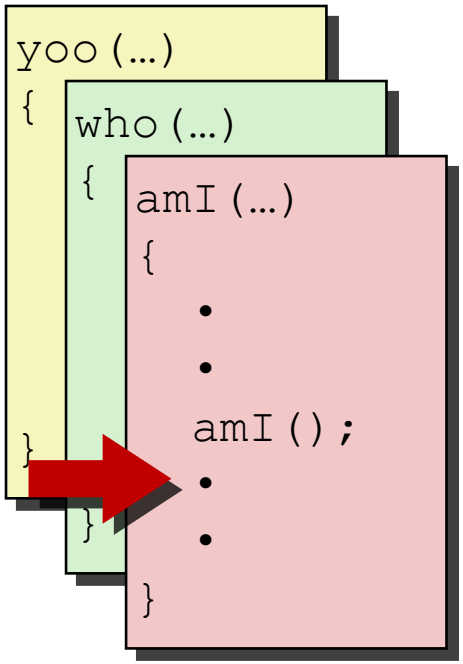
Beispiel: Prozeduraufrufe (cont.)

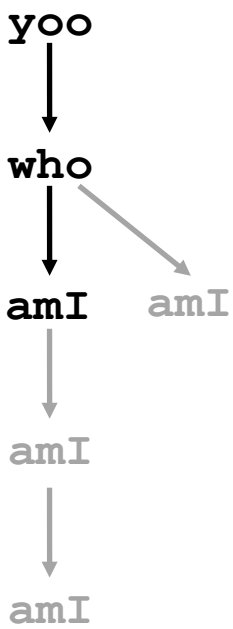


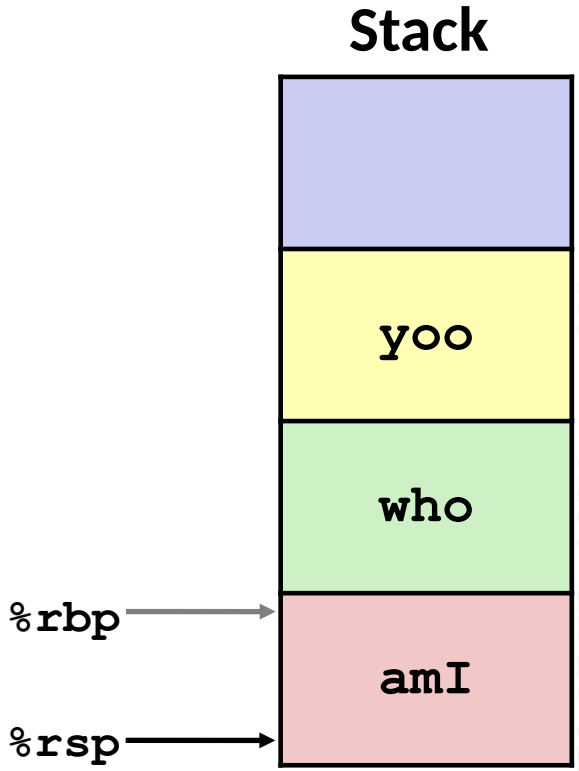




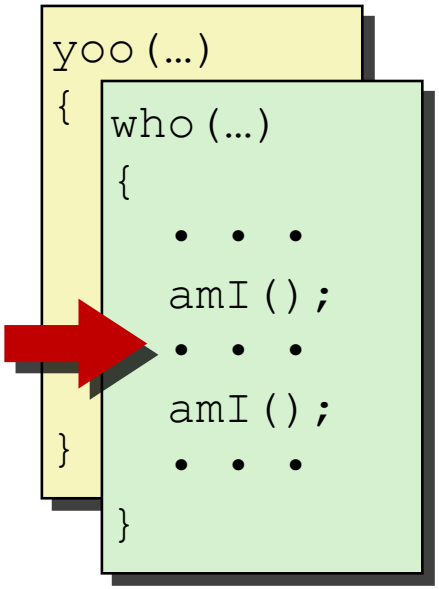
Beispiel: Prozeduraufrufe (cont.)

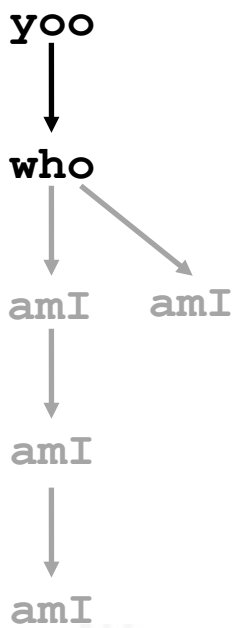


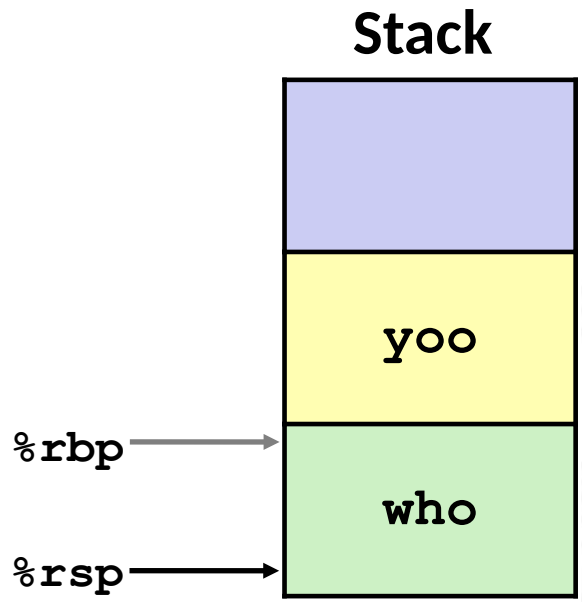




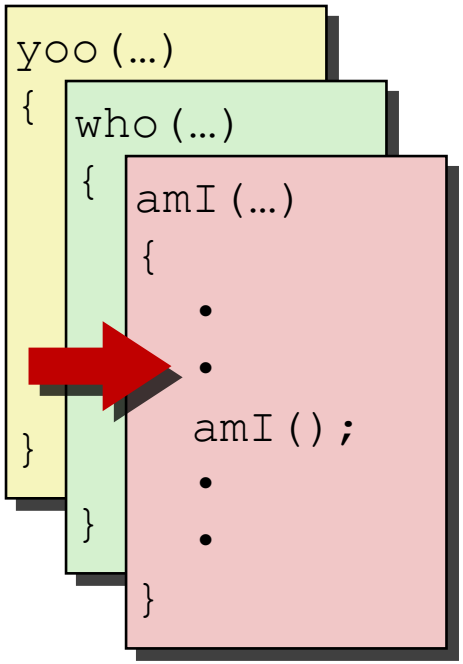
Beispiel: Prozeduraufrufe (cont.)

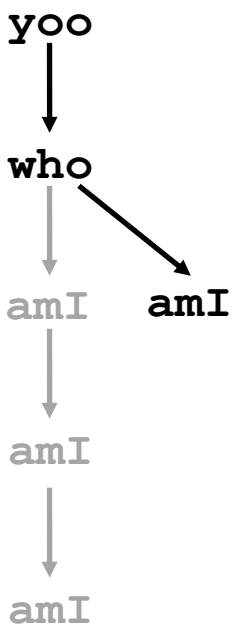


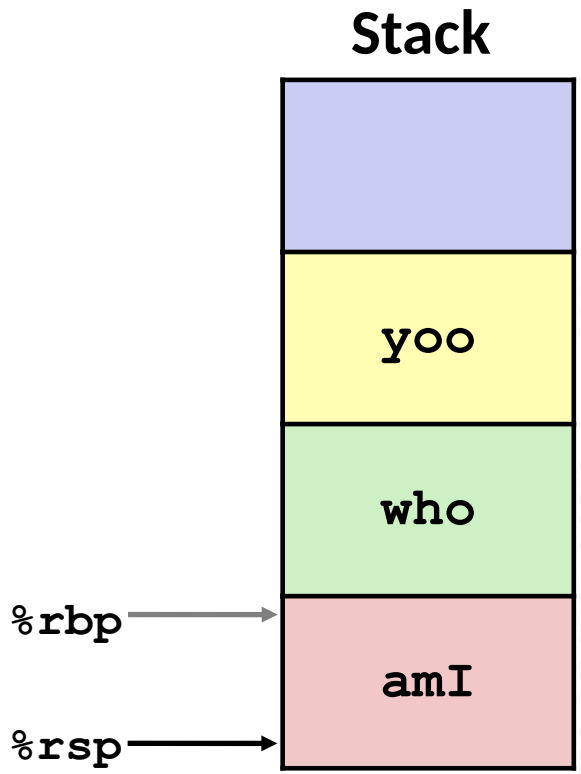




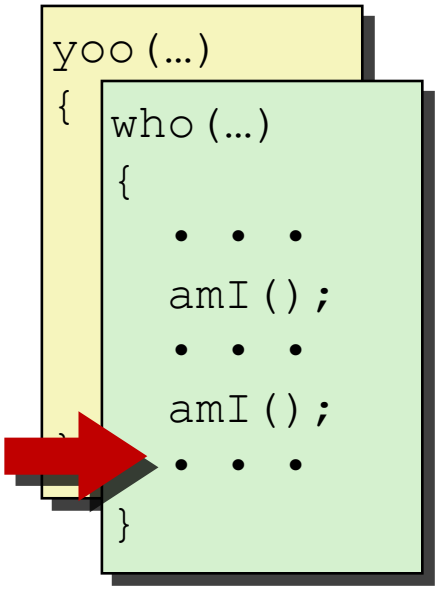
Beispiel: Prozeduraufrufe (cont.)

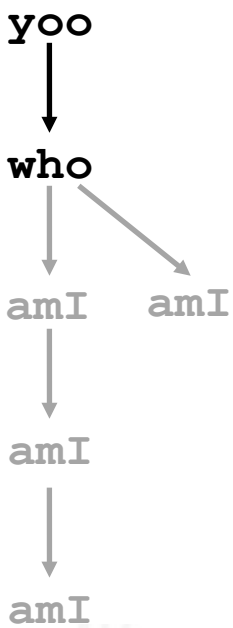


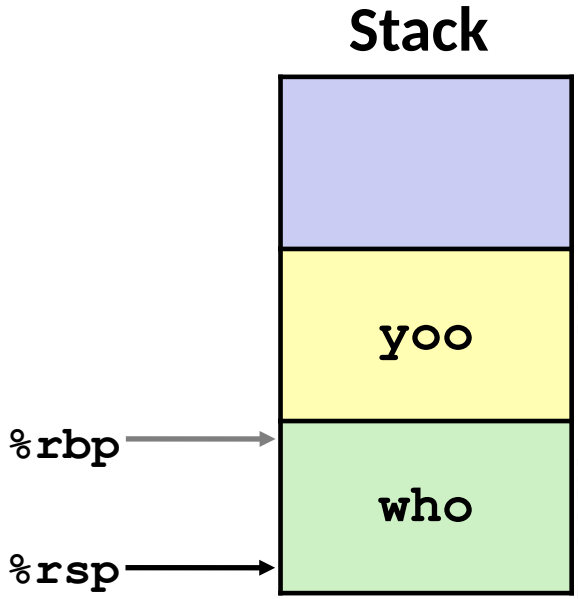




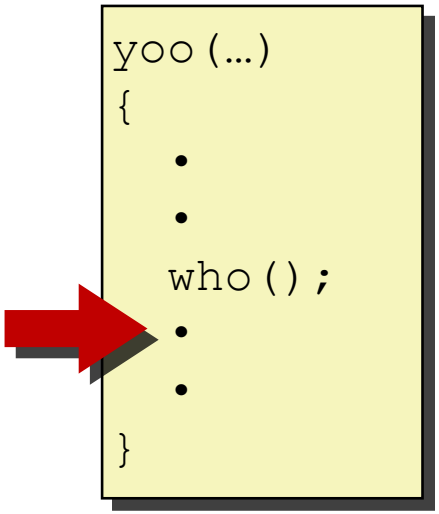
Beispiel: Prozeduraufrufe (cont.)

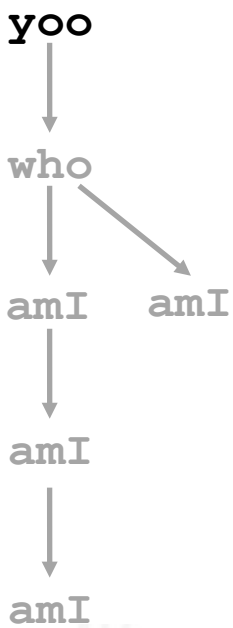


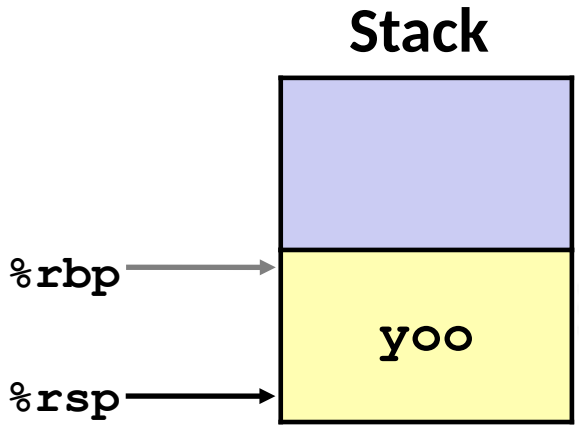




Beispiel: Prozeduraufrufe (cont.)







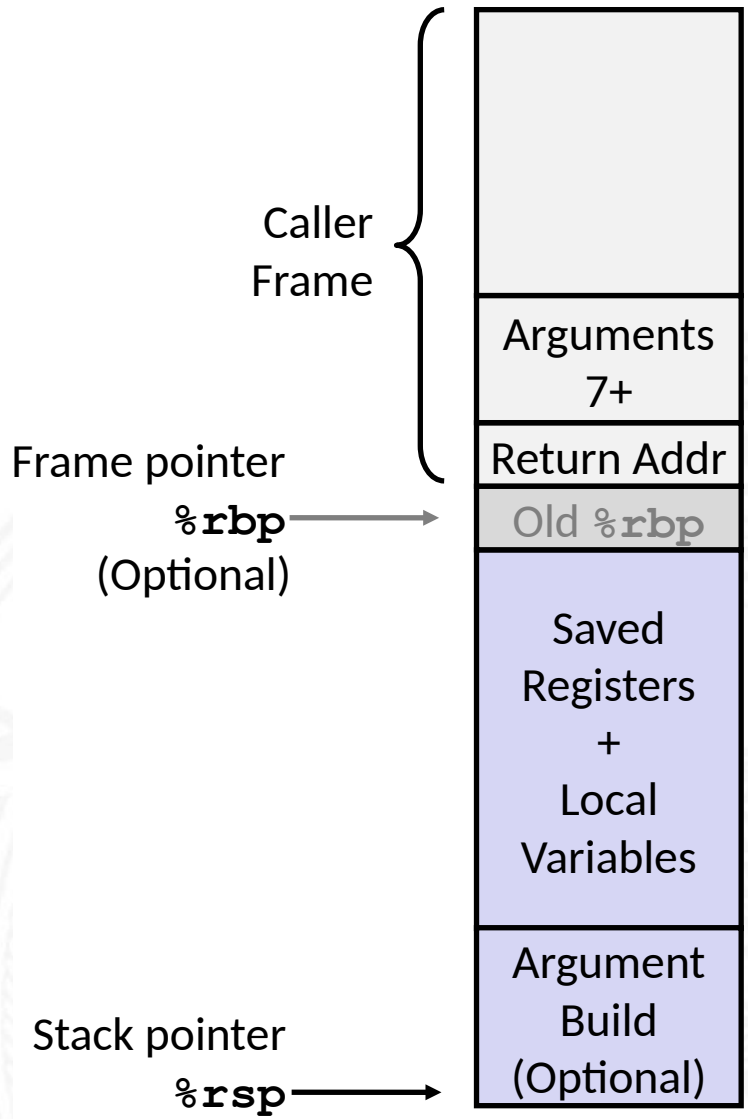
x86-64/Linux Stack-Frame

aktueller Stack-Frame / „Callee“

- Argumente: Parameter für Funktionsaufruf
- lokale Variablen
  - wenn sie nicht in Registern gehalten werden können
- gespeicherter Registerkontext
- Zeiger auf vorherigen Frame

„Caller“ Stack-Frame

- Rücksprungadresse
  - von `call`-Anweisung erzeugt
- Argumente für aktuellen Aufruf



Register Sicherungskonventionen

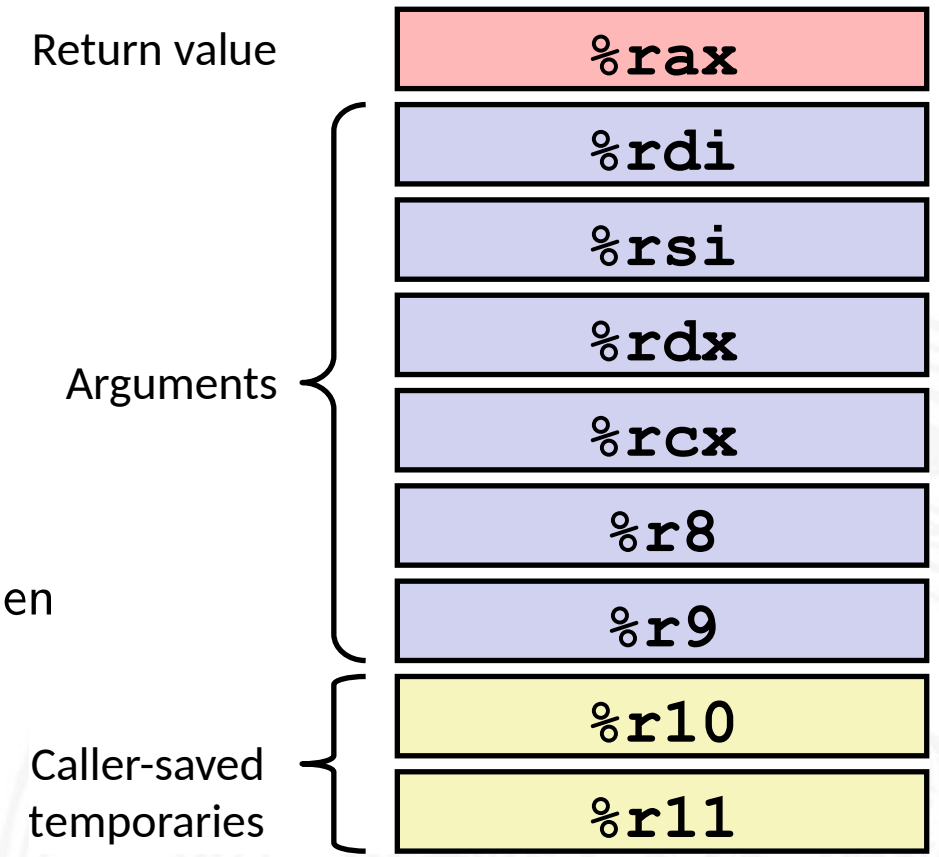
- yoo („Caller“) ruft Prozedur `who` („Callee“) auf

⇒ *Welche Register können temporär von `who` genutzt werden?*

- zwei mögliche Konventionen
  - „Caller-Saved“
    yoo speichert in seinen Frame vor Prozeduraufruf
  - „Callee-Saved“
    `who` speichert in seinen Frame vor Benutzung

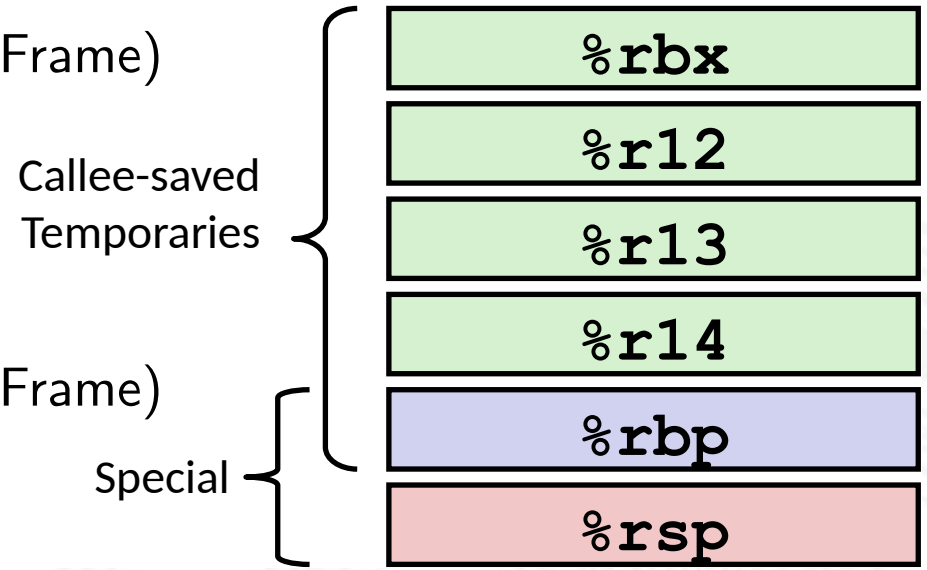
Register Sicherungskonventionen (cont.)

- `%rax`
  - Rückgabewert
  - Caller-Saved
  - kann lokal geschrieben werden
- `%rdi...%r9`
  - Argumente
  - Caller-Saved
  - können lokal geschrieben werden
- `%r10`, `%r11`
  - Caller-Saved
  - können lokal geschrieben werden



Register Sicherungskonventionen (cont.)

- `%rbx`, `%r12`...`%r14`
  - Callee-Saved
  - Prozedur muss sichern (Stack-Frame) und zurückschreiben
- `%rbp`
  - Callee-Saved
  - Prozedur muss sichern (Stack-Frame) und zurückschreiben
  - ▷ Frame-Pointer $\widehat{=}$ Beginn des eigenen Frames
- `%rsp`
  - Behandlung durch `call`/`return`
  - Sonderfall, quasi Callee-Save



Elementare Datentypen

- Ganzzahl (Integer)
  - wird in allgemeinen Registern gespeichert
  - abhängig von den Anweisungen: *signed*/*unsigned*
  - Architektur-, Compiler-, OS-abhängig

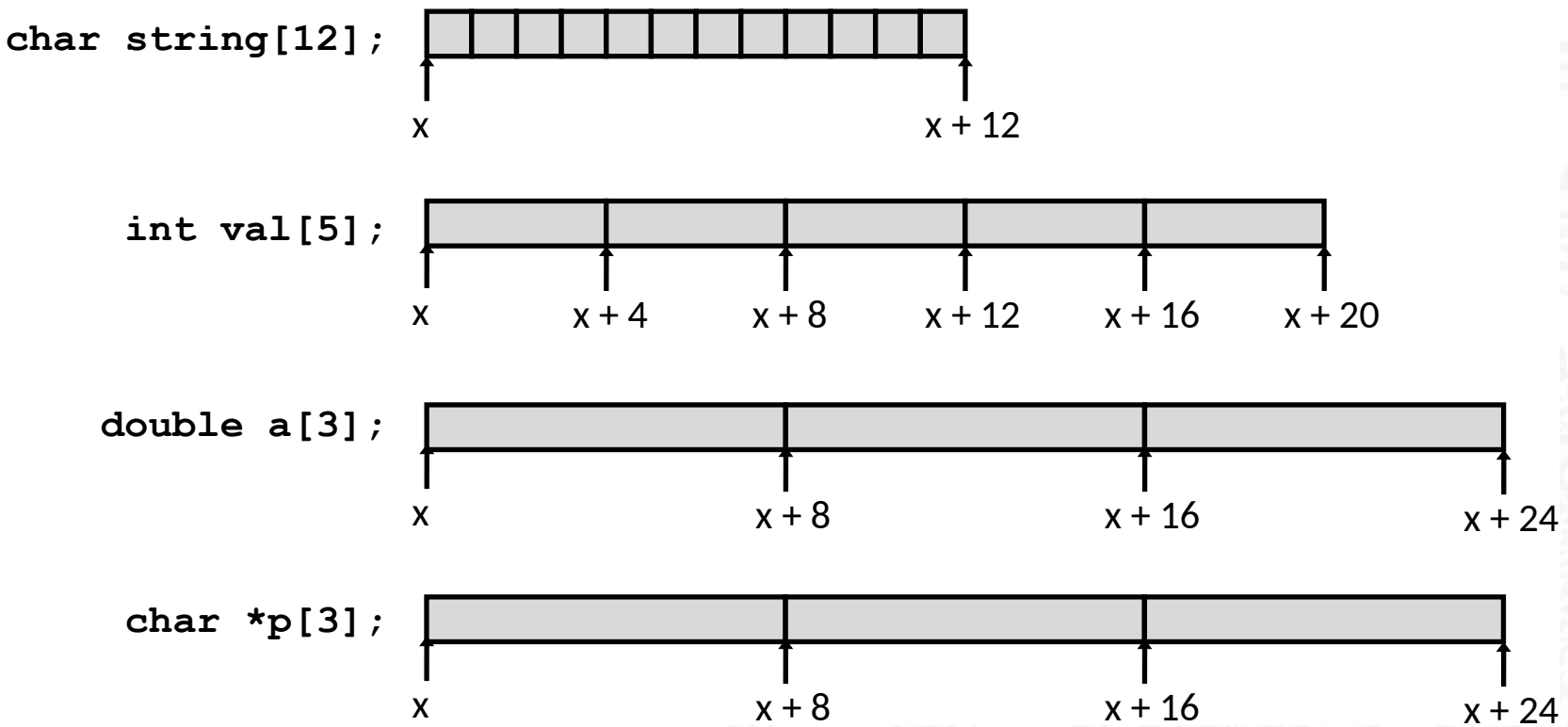
Intel x86	as	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
doubleword	l	4	[unsigned] int / long
quadword	q	8	[unsigned] long / long long

- Gleitkomma (Floating Point)
  - wird in Gleitkomma-Registern gespeichert
  - as: *Gnu ASsembler*

Intel x86	as	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12	long double

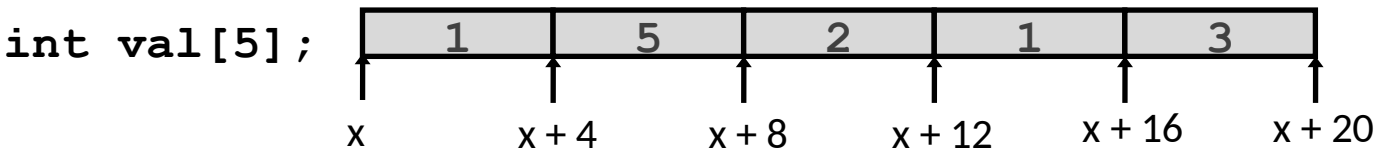
Array: Allokation / Speicherung

- `T A[N];`
  - Array `A` mit Daten von Typ `T` und `N` Elementen
  - fortlaufender Speicherbereich von `N` $\times$ sizeof(`T`) Bytes



Array: Zugriffskonvention

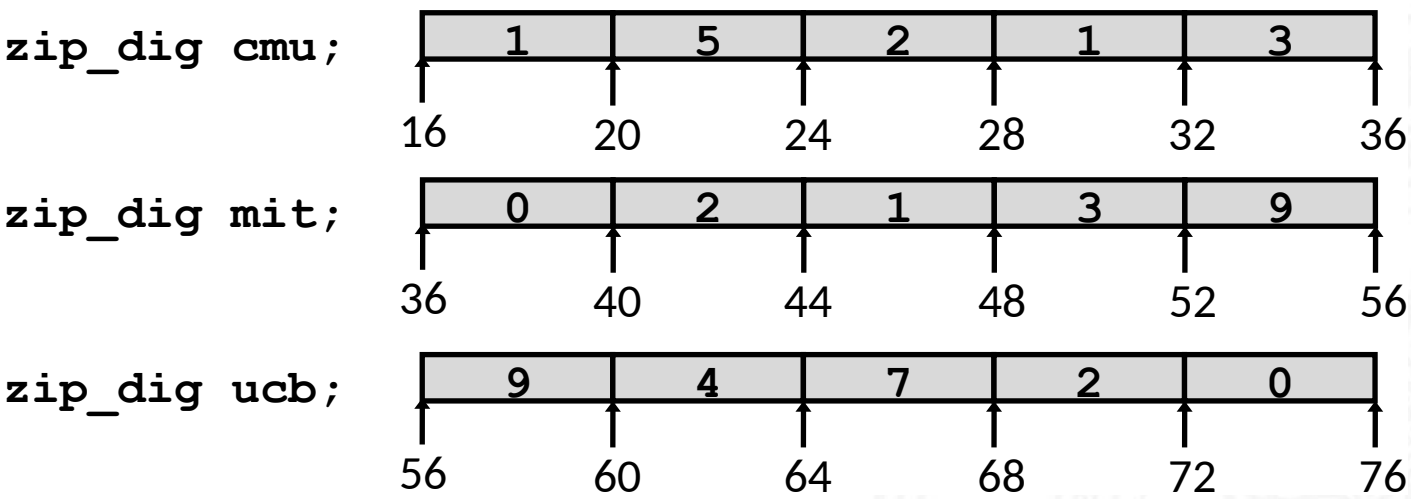
- `T A[N];`
  - Array `A` mit Daten von Typ `T` und `N` Elementen
  - Bezeichner `A` zeigt auf erstes Element des Arrays: Element 0



Reference	Type	Value	
`val[4]`	`int`	`3`	
`val`	`int *`	`x`	
`val+1`	`int *`	`x + 4`	
`&val[2]`	`int *`	`x + 8`	
`val[5]`	`int`	`??`	
`*(val+1)`	`int`	`5`	`//val[1]`
`val + i`	`int *`	`x + 4 * i`	`//&val[i]`

Beispiel: einfacher Arrayzugriff


```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

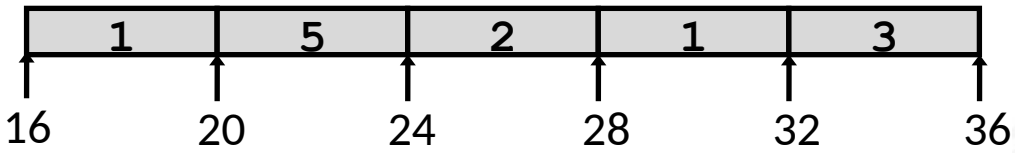


Beispiel: einfacher Arrayzugriff (cont.)

- Adressieren von `%rdi` $+ 4 \times$ `%rsi`

$\Rightarrow$ Speicheradresse `(%rdi,%rsi,4)`

`zip_dig cmu;`




```
int get_digit
  (zip_dig z, int digit)
{
  return z[digit];
}
```

Register	Funktion
%rdi	Array Startadresse
%rsi	Array Index

x86-64

```
  # %rdi = z
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

Beispiel: einfacher Arrayzugriff (cont.)

Achtung bei Arrayzugriffen:

- keine Bereichsüberprüfung („*bounds checking*“)
- Verhalten außerhalb des Indexbereichs ist Implementierungsabhängig

Zwei- und mehrdimensionale Arrays

$(N \times M)$ Matrizen? drei grundsätzliche Möglichkeiten

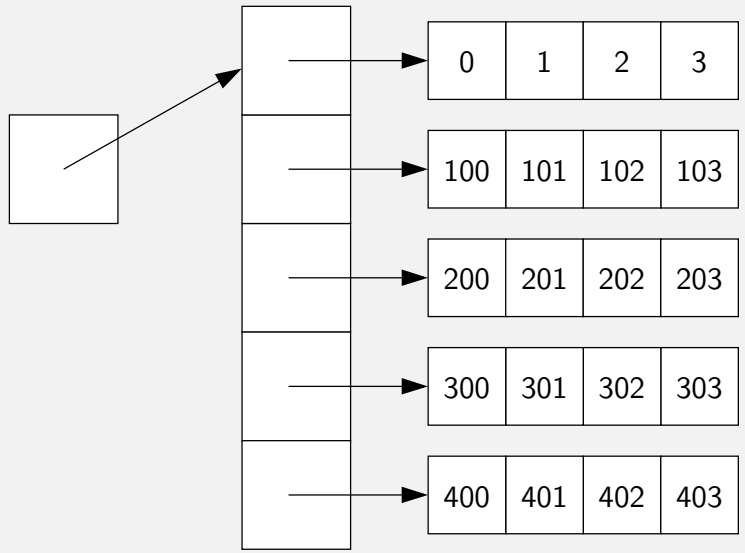
1. Array von Pointern auf Zeilen-Arrays von Elementen — Java
   - sehr flexibel, auch für nicht-rechteckige Layouts
   - Sharing/Aliasing von Zeilen möglich

- Array von $N \times M$ Elementen und passende Adressierung
  2. row-major Anordnung — C, C++
  3. column-major Anordnung — Matlab, FORTRAN

- bei Verwendung/Mischung von Bibliotheksfunktionen aus anderen Sprachen unbedingt berücksichtigen

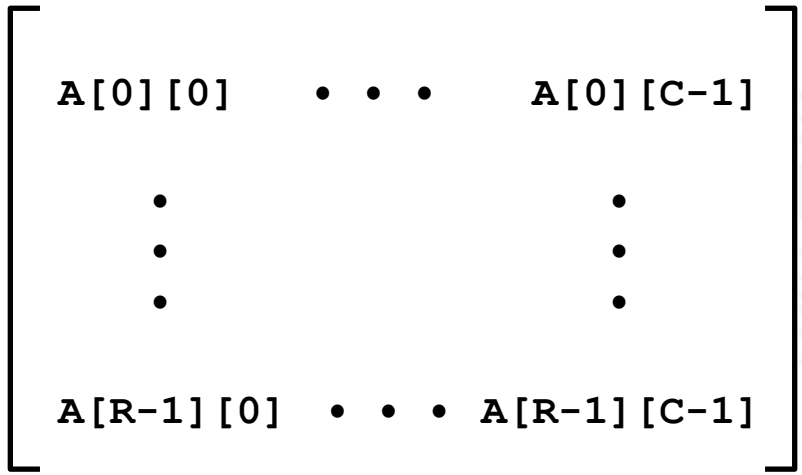
Java: Array von Pointern auf Arrays von Elementen

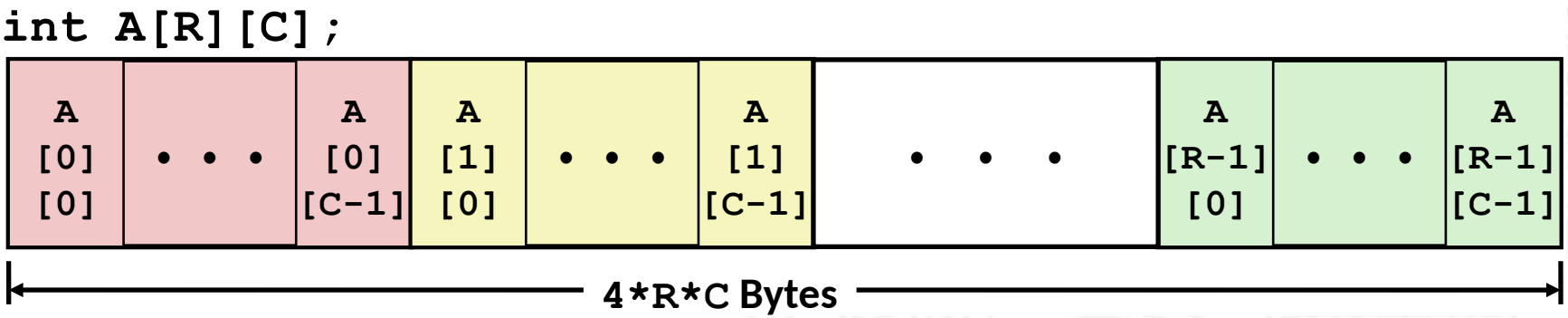

```
class MatrixDemo {
  int matrix[][]; // matrix[i]->

  public MatrixDemo( int NROWS, int NCOLS ) {
    matrix = new int[NROWS][NCOLS];
    for( int r=0; r < matrix.length; r++ ) {
      for( int c =0; c < matrix[r].length; c++ ) {
        matrix[r][c] = 100*r + c;
      }
    }
    // int[] row0 = matrix[0];
    // int    m23 = matrix[2][3];
  }
  public int get( int r, int c ) {
    return matrix[r][c];
  }
}
```



Zweidimensionale Arrays in C

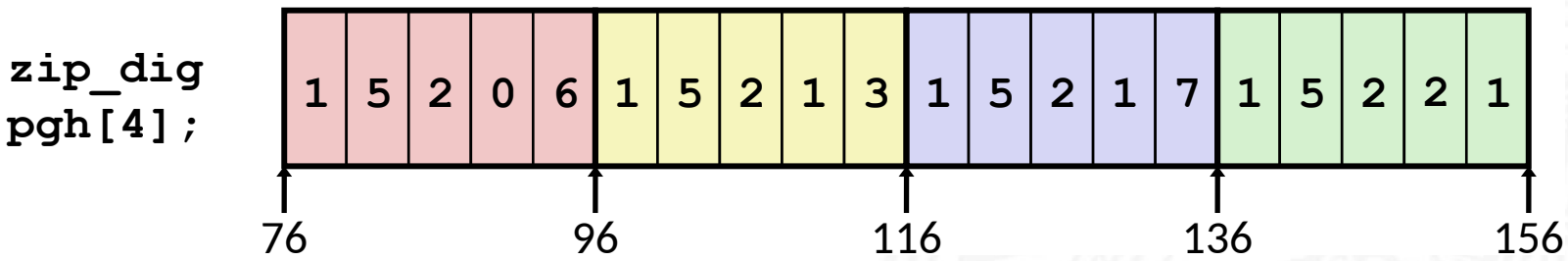
- Deklaration ⟨*T*⟩ ⟨*A*⟩ [⟨*R*⟩][⟨*C*⟩];
- Größe: ⟨*R*⟩*⟨*C*⟩*sizeof(⟨*T*⟩) Bytes
- „*row-major*“ Anordnung





Zweidimensionale Arrays in C (cont.)


```
#define PCOUNT 4
typedef int zip_dig[5];

zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



Mehrdimensionale Arrays: entsprechend

- d-dimensionales $N_1 \times N_2 \times \cdots \times N_d$ Array
  - Element adressiert mit Tupel $(n_1, n_2, \ldots, n_d)$, mit d (zero-offset) Indizes $n_k \in [0, N - K - 1]$

- row-major Anordnung: letzte Dimension ist fortlaufend

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

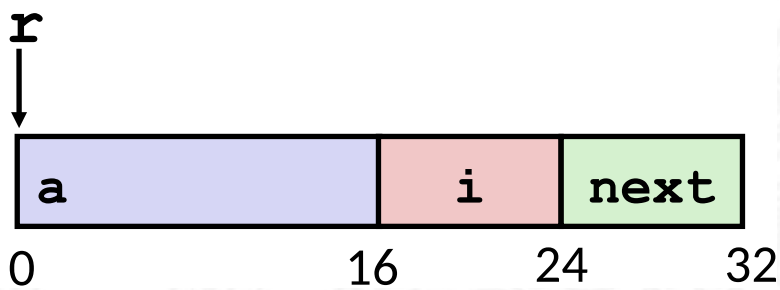
- column-major Anordnung: erste Dimension ist fortlaufend

$$n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\cdots + N_{d-1} n_d) \cdots)) = \sum_{k=1}^{d} \left(\prod_{\ell=1}^{k-1} N_\ell \right) n_k$$

Strukturen (*Records*)

- Allokation eines zusammenhängenden Speicherbereichs
- Elemente der Struktur über Bezeichner referenziert
- verschiedene Typen der Elemente sind möglich
- Zeiger `r` auf Byte-Array
  - für Zugriff auf Struktur(element)
  - Compiler bestimmt Offset für jedes Element

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

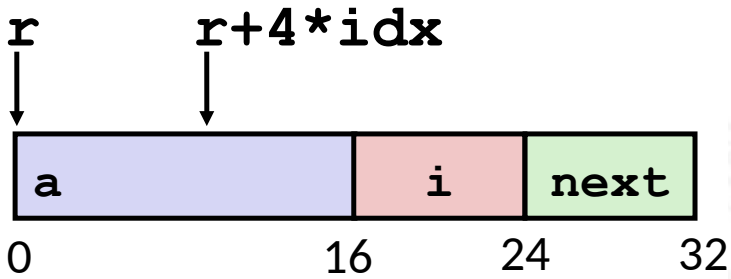


Beispiel: Strukturzugriffe

- Arrayzugriff in Struktur

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

```
int *get_ap
  (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```



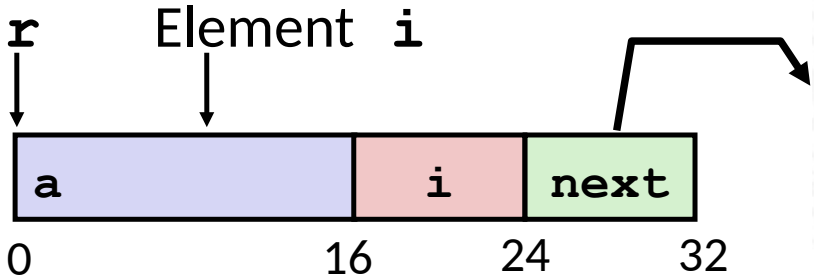

```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Beispiel: Strukturzugriffe (cont.)

- Verlinkte Liste durchlaufen

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```




```
.L11:                                  # loop:
  movslq  16(%rdi), %rax               #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)          #   M[r+4*i] = val
  movq    24(%rdi), %rdi               #   r = M[r+24]
  testq   %rdi, %rdi                   #   Test r
  jne     .L11                         #   if !=0 goto loop
```

Ausrichtung der Datenstrukturen (*Alignment*)

- Datenstrukturen an Wortgrenzen ausrichten
  double- / quad-word
- sonst Problem
  - ineffizienter Zugriff über Wortgrenzen hinweg
  - virtueller Speicher und Caching

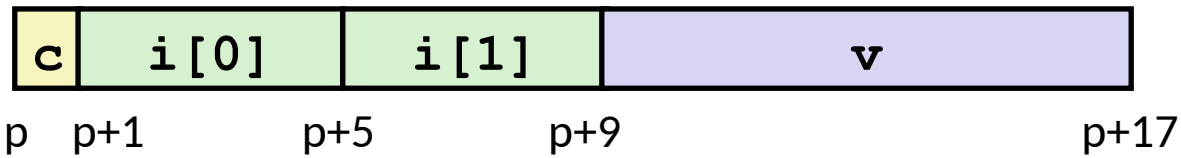
⇒ Compiler erzeugt „Lücken“ zur richtigen Ausrichtung

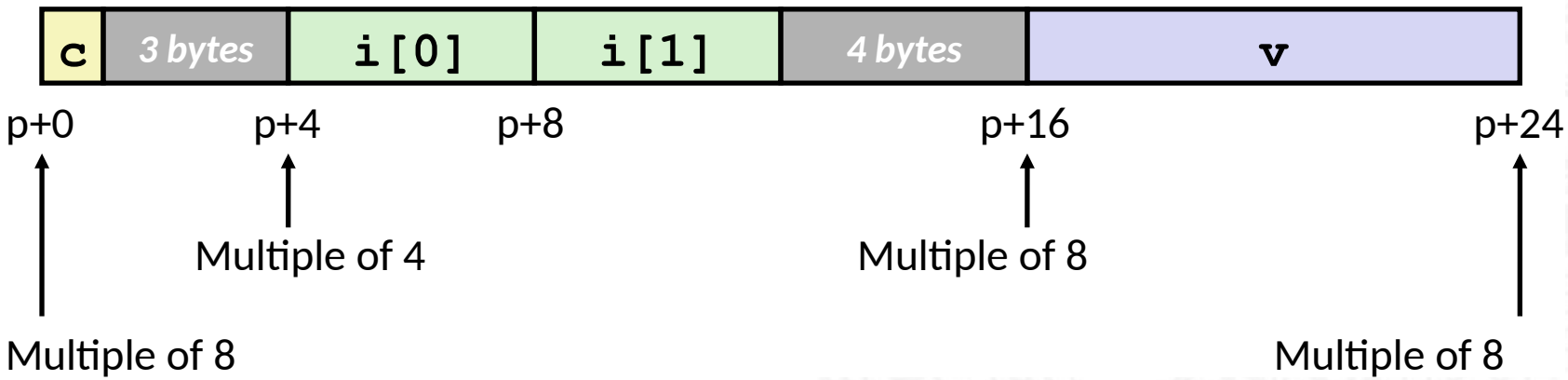
- typisches Alignment (x86-64)

Länge	Typ	Alignment
1 Byte	`char`	keine speziellen Verfahren
2 Byte	`short`	Adressbits: ...0
4 Byte	`int`, `float`	–"– ...00
8 Byte	`double`, `long`, `char *`	–"– ...000

Beispiel: Structure Alignment


```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```





Linker und Loader

- Programm in mehrere Quelldateien aufgeteilt

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

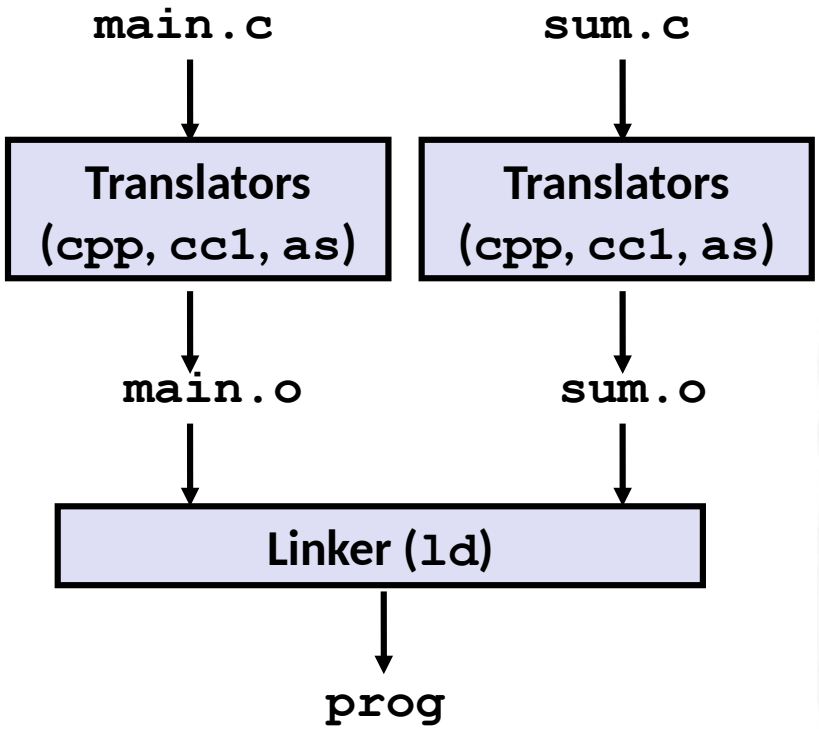
main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Linker und Loader (cont.)

- Compiler(-driver) startet einzelne Programme Linux `gcc`
  - Präprozessor (`cpp`), Compiler (`cc`), Assembler (`as`) und Linker (`ld`)
  - „Feintuning“ und Steuerung über Kommandozeilen-Parameter 'zig Parameter für jedes Teilprogramm `man gcc`



Linker und Loader (cont.)

\+ Modularität
- Programm in übersichtlichen kleinen Dateien
- Funktionen können wiederverwendet werden

$\Rightarrow$ vorgefertigte Programmbibliotheken

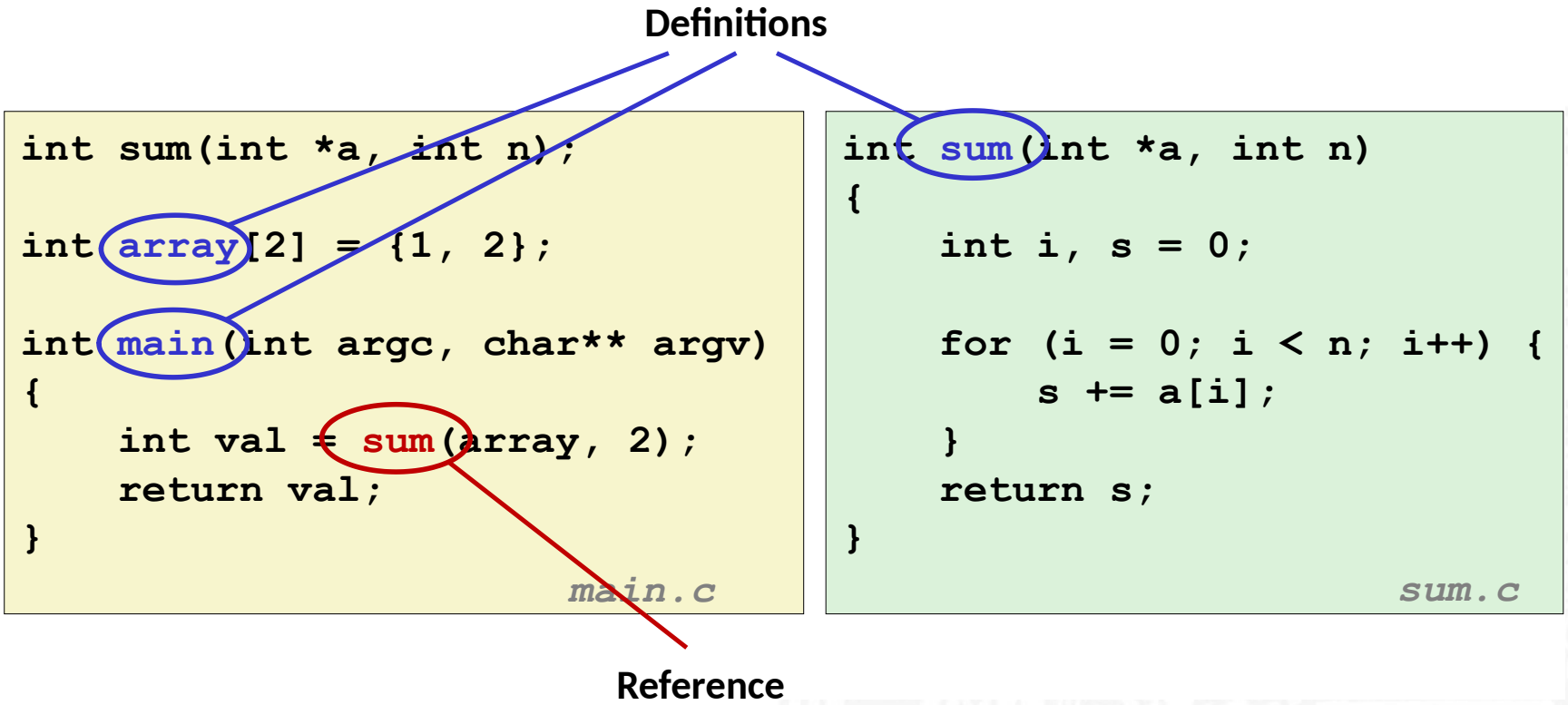
\+ Effizienz

$\Rightarrow$ Zeitvorteil
- nach Änderung müssen nur kleine Teile neu übersetzt werden
- ermöglicht paralleles Compilieren

$\Rightarrow$ (Speicher-) Platzvorteil
- wichtige Funktionen in Datei aggregiert (z.B. `malloc`, `printf`)
- ermöglicht gemeinsame Nutzung

Aufgaben des Linkers

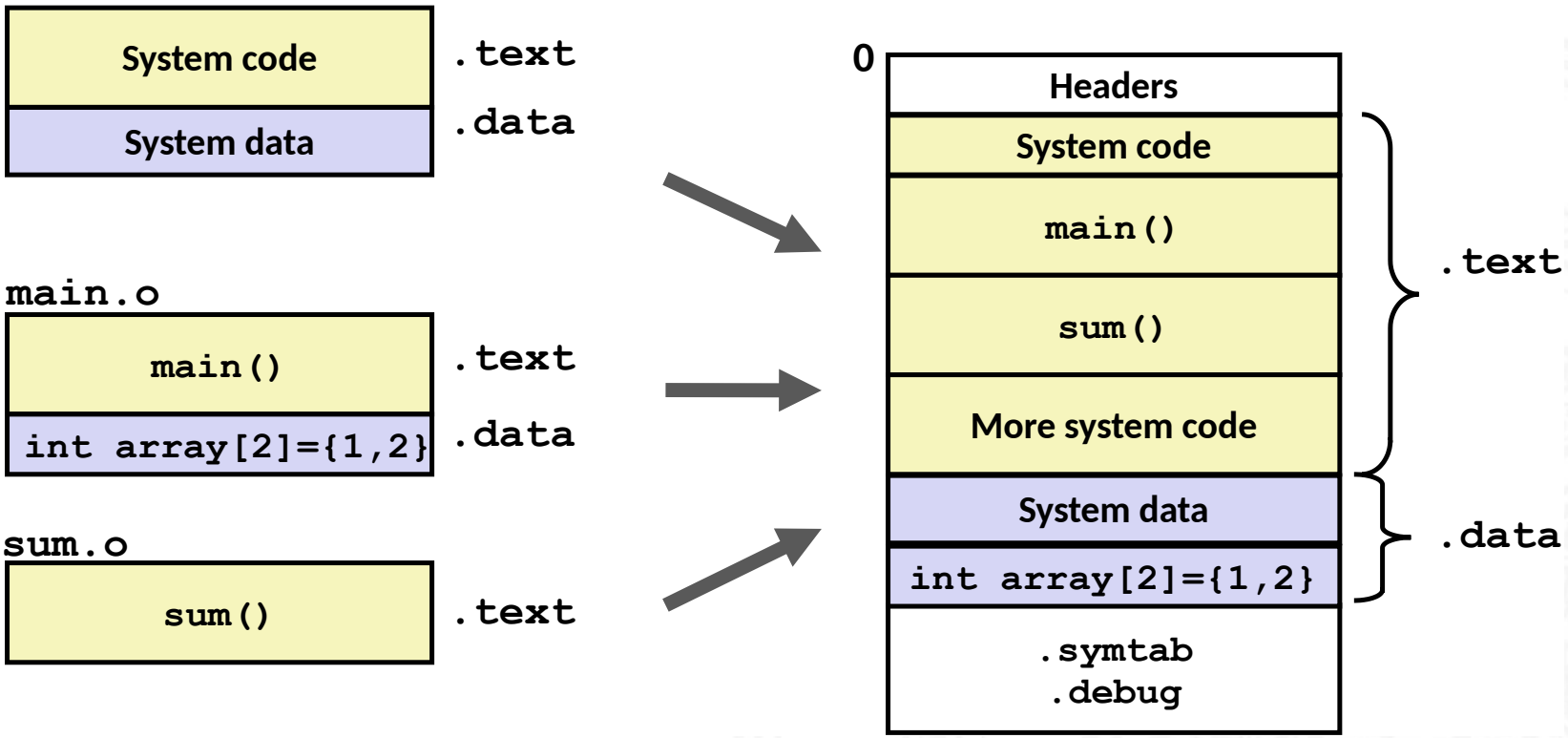
1. Symbole identifizieren (globale Variablen, Funktionen)
   Symbole auflösen (= eindeutig machen) ⇒ Symboltabelle



Aufgaben des Linkers (cont.)

2. „Relocation“
   - Programmcode und -daten der Quelldateien zusammenfassen
   - Symboltabellen zusammenfassen

   ⇒ Speicheradressen eindeutig machen: Sprünge+Symbole



⇒ erzeugt ein ausführbares Programm für *Loader*

Statisches / dynamisches Linken

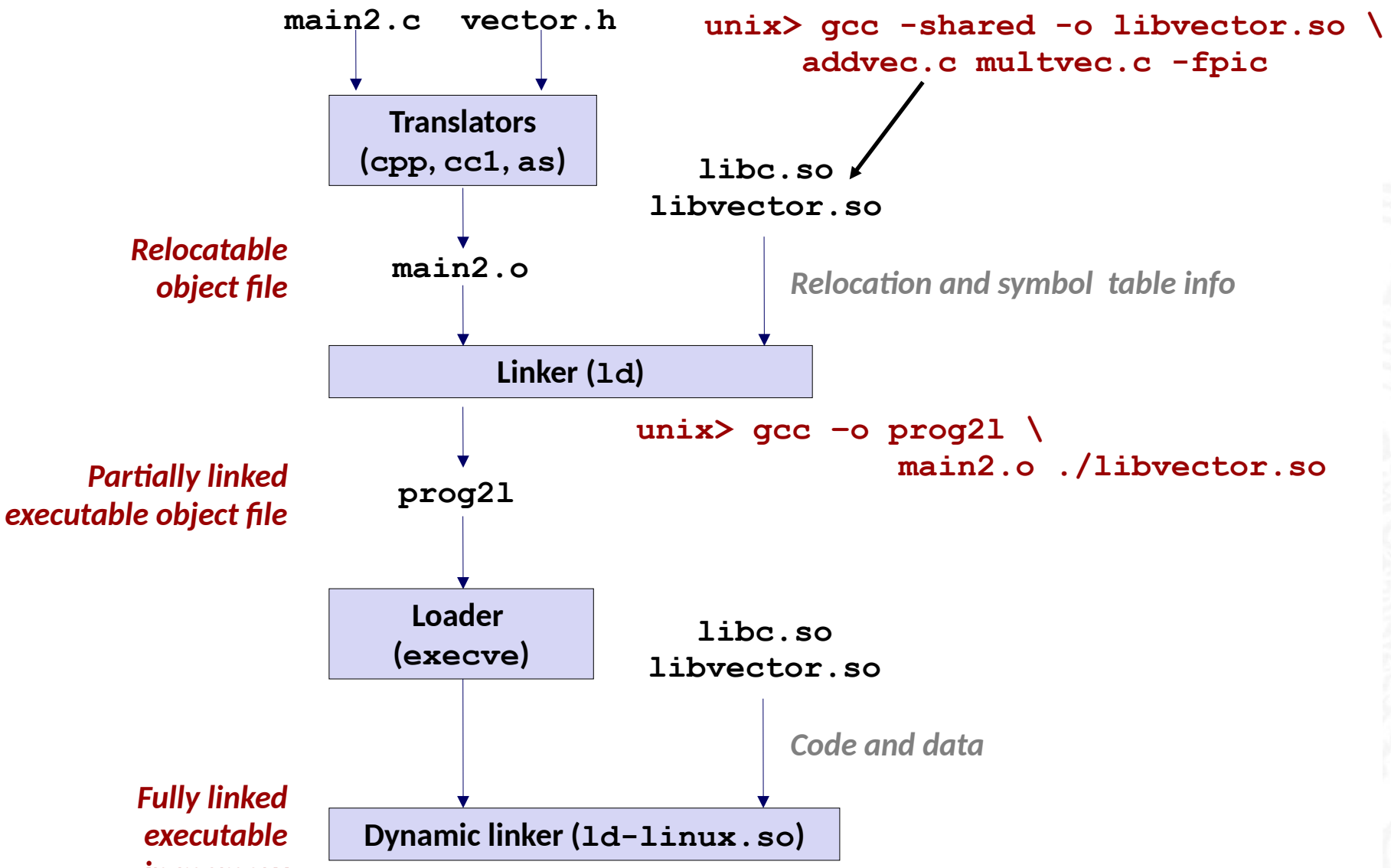
- Statisches Binden („static linking“)
  - Funktionen aus Bibliotheksarchiven (`.a`-Dateien) werden in ausführbares Programm eingebaut
  - nicht genutzte Funktionen werden entfernt
  - Linken während Compilierung

- Dynamisches Binden („dynamic linking“)
  - Bibliotheken werden erst beim Laden in Speicher oder sogar erst zur Laufzeit dazugelinkt
  - gemeinsame Nutzung von Funktionen durch mehrere Prozesse (incl. Betriebssystem); die zugehörigen Bibliotheken liegen aber (maximal) einmal im Speicher

  - signifikant effizienter als separat statische gelinkte Programme
  - Linux: `.so`-Dateien – „Shared Object“
    Windows: `.dll`-Dateien – „Dynamic Link Libraries“

Statisches / dynamisches Linken (cont.)

dynamisches Linken beim Laden



Assembler-Programmierung

viele Themen aus Zeitgründen nicht behandelt

- Linker und Loader
  - genauere Funktionsweise von Linker und Loader
  - programmiertechnische Realisierung
  - Probleme bei der Symbolauflösung
- Speicherverwaltung
  - dynamische Speicherverwaltung / der „Heap“
- Objektorientierte Konzepte
  - Daten mit zugehörigen Methoden kombinieren

Assembler-Programmierung (cont.)

- *Was kann zur Laufzeit alles schief gehen?*
  - Pufferüberläufe
  - Sicherheitsaspekte

- *Wie ist die Verbindung zum Betriebssystem?*
- ...

weitere Informationen unter:

→ R.E. Bryant, D.R. O'Hallaron:
Computer systems – A programmers perspective [BO15]

→ die „passende" Vorlesung der *Carnegie Mellon Uni.*
`www.cs.cmu.edu/~213` – Foliensätze unter „Schedule"

Literatur

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

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–8689–4238–5

[IA64] *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture*.
Intel Corp.; Santa Clara, CA.
`software.intel.com/en-us/articles/intel-sdm`

Literatur (cont.)

[PH17] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – RISC-V Edition*.
Morgan Kaufmann Publishers Inc., 2017.
ISBN 978–0–12–812275–4

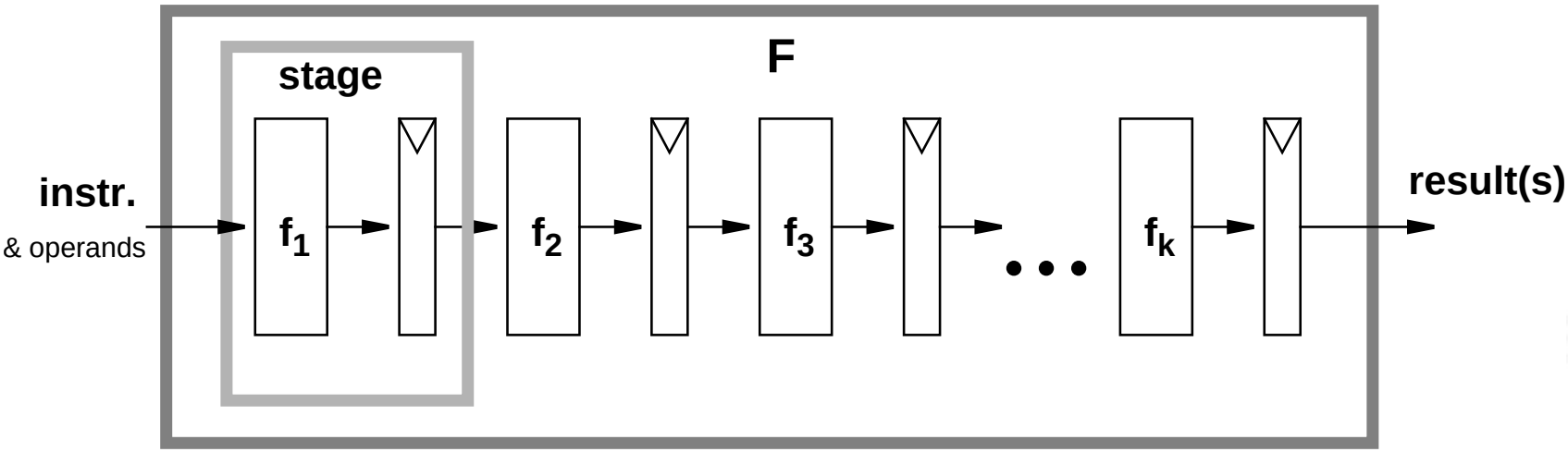
[PH16b] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle*.
5. Auflage, Oldenbourg, 2016. ISBN 978–3–11–044605–0

[Hyd10] R. Hyde: *The Art of Assembly Language Programming*.
2nd edition, No Starch Press, 2010. ISBN 978–1–59327–207–4.
`www.plantation-productions.com/Webster/www.artofasm.com`

Gliederung

Gliederung (cont.)

Pipelining / Fließbandverarbeitung



Grundidee

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

Pipelining / Fließbandverarbeitung (cont.)

Pipelining-Konzept

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

Konsequenz

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

Pipelining / Fließbandverarbeitung (cont.)

Arithmetische Pipelines

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

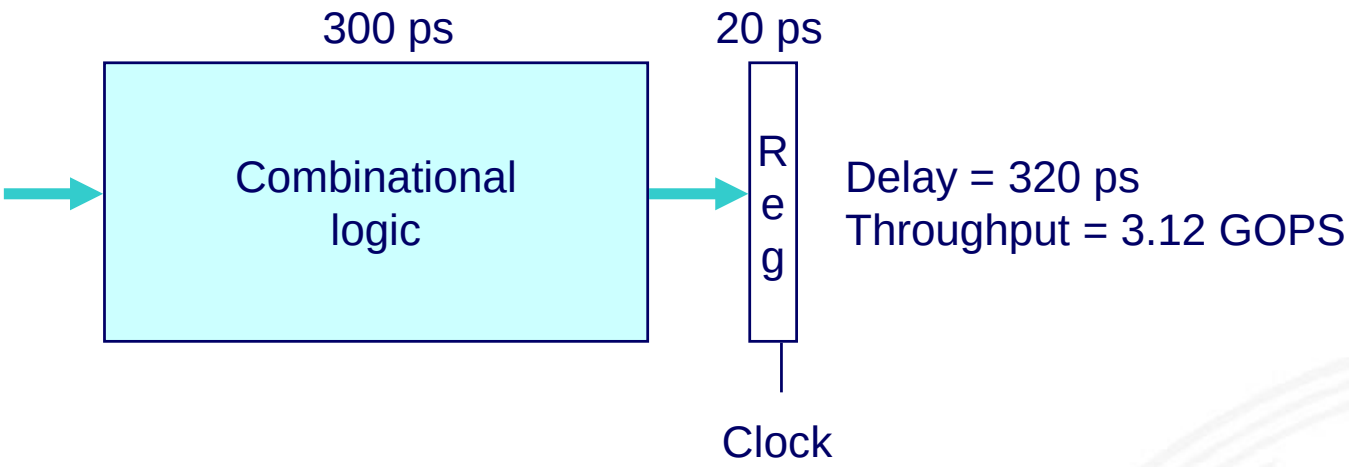
\+ Erhöhung des Durchsatzes, wenn Berechnung mehrfach hintereinander ausgeführt wird

Befehlspipeline im Prozessor

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

→ folgt in *Befehlspipeline*, ab Folie 1038

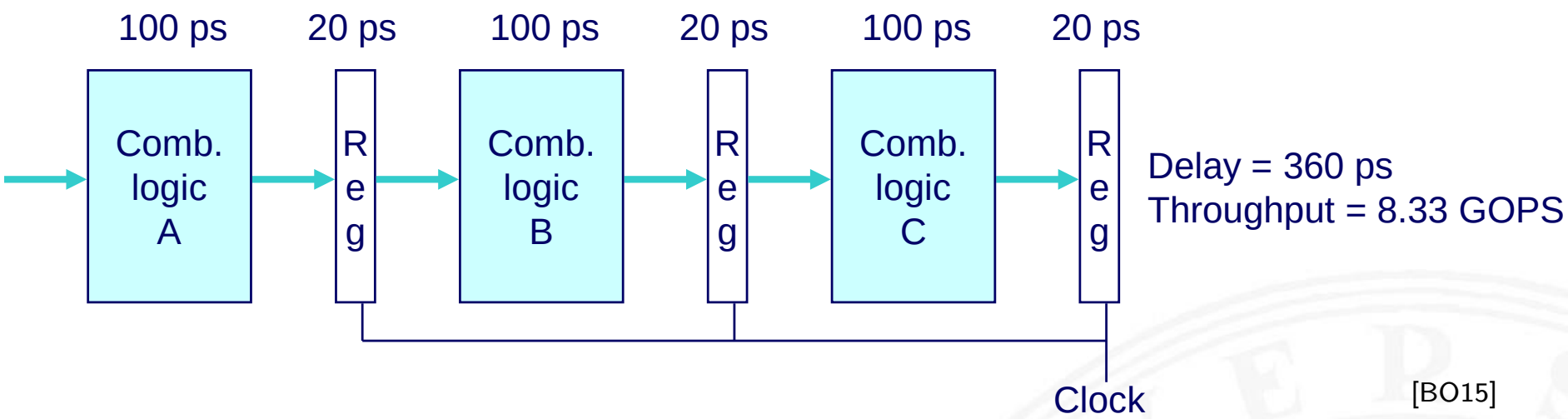
Beispiel: Schaltnetz ohne Pipeline



[BO15]

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

Beispiel: Version mit 3-stufiger Pipeline

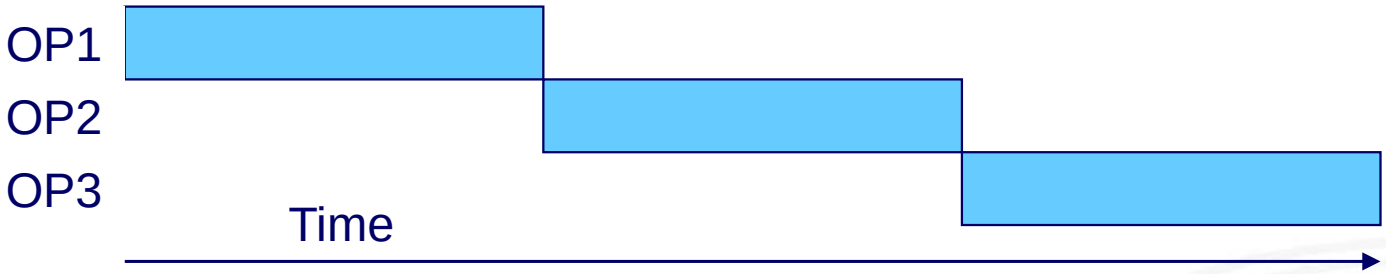


[BO15]

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

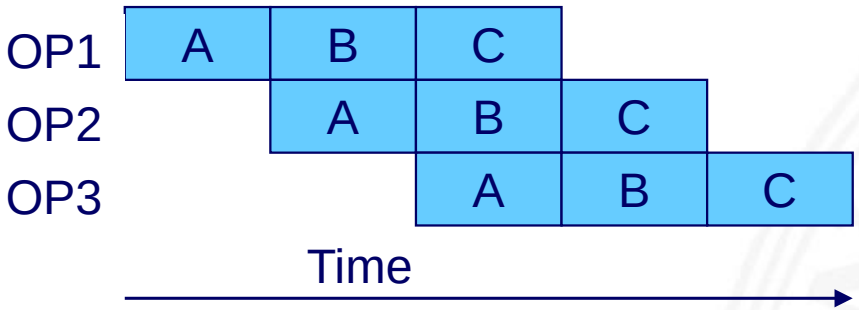
Prinzip: 3-stufige Pipeline

- ohne Pipeline

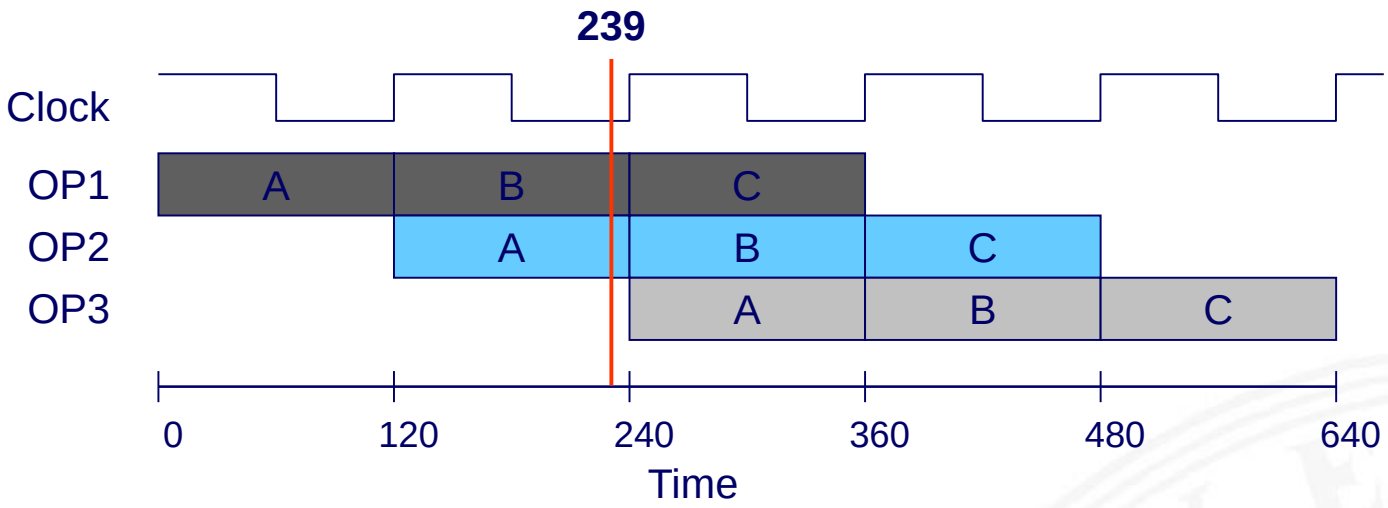


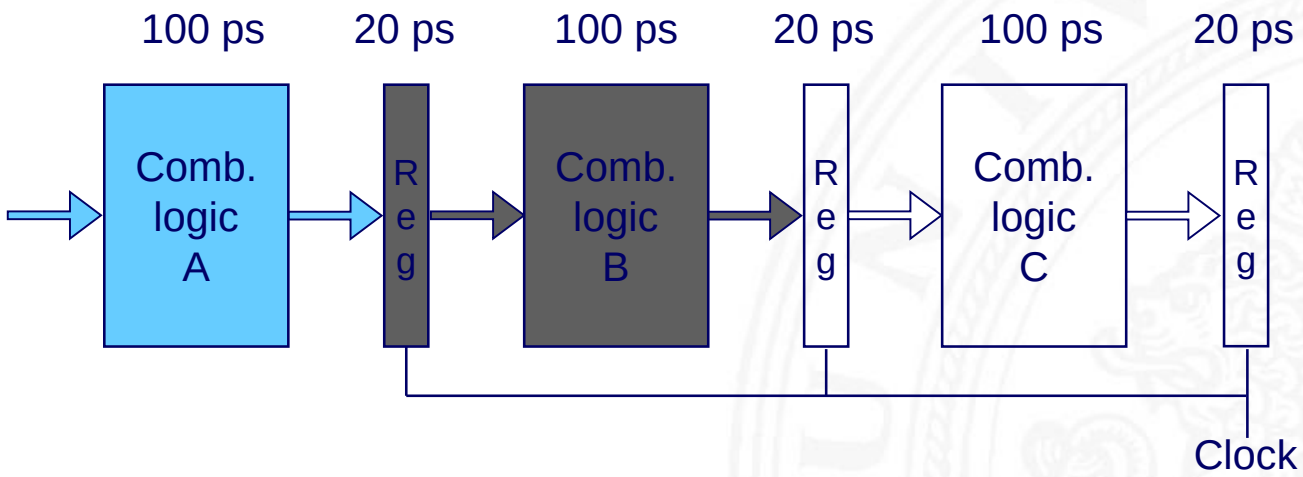
- 3-stufige Pipeline

[BO15]



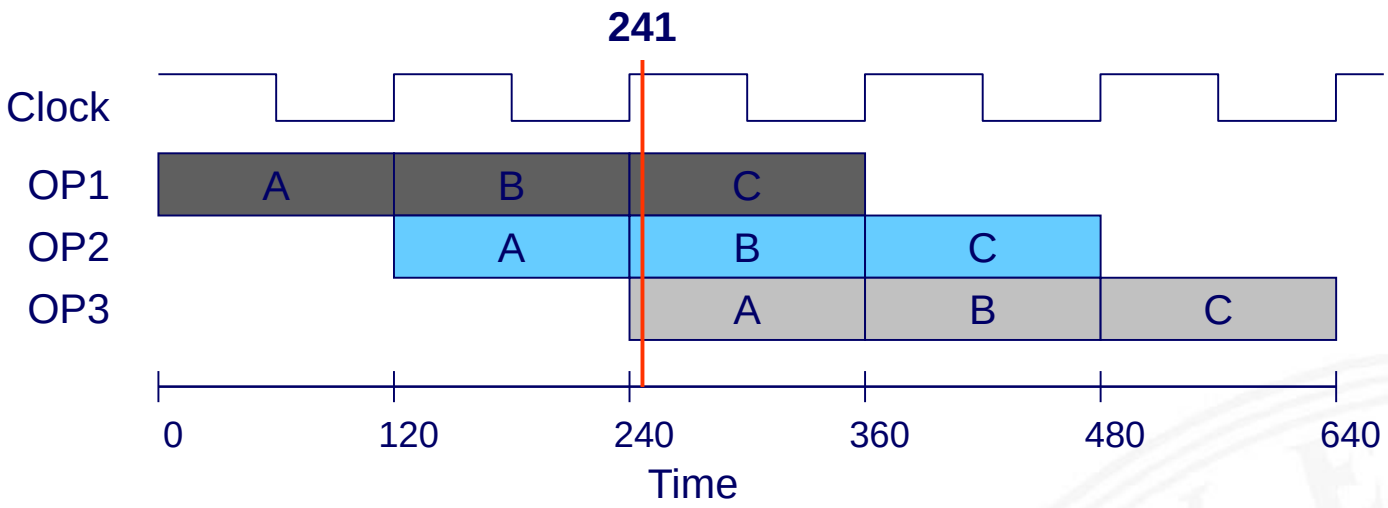
Timing: 3-stufige Pipeline

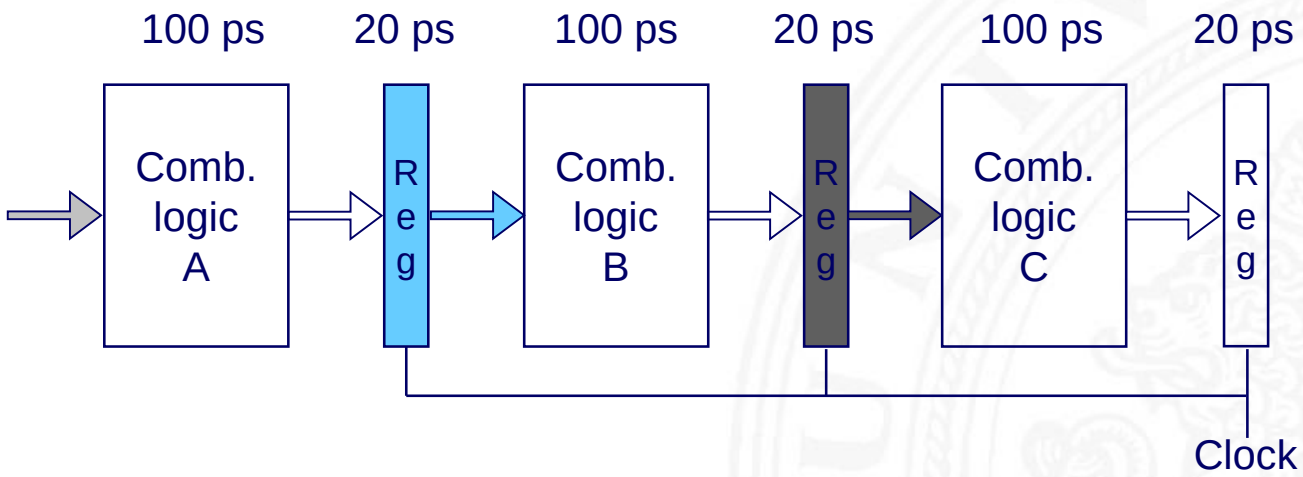




[BO15]

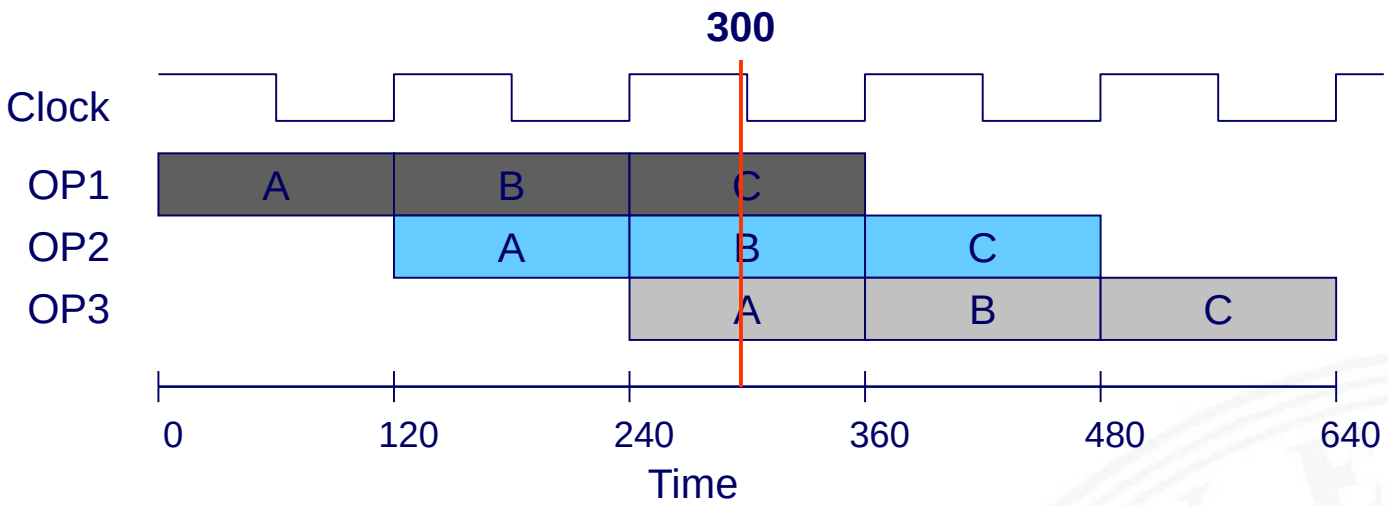
Timing: 3-stufige Pipeline

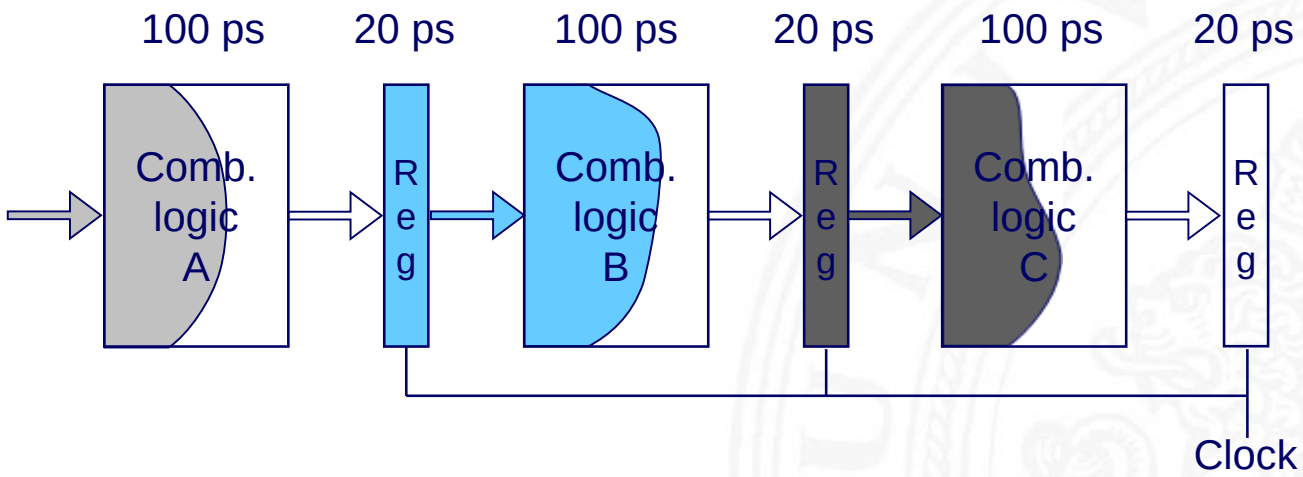




[BO15]

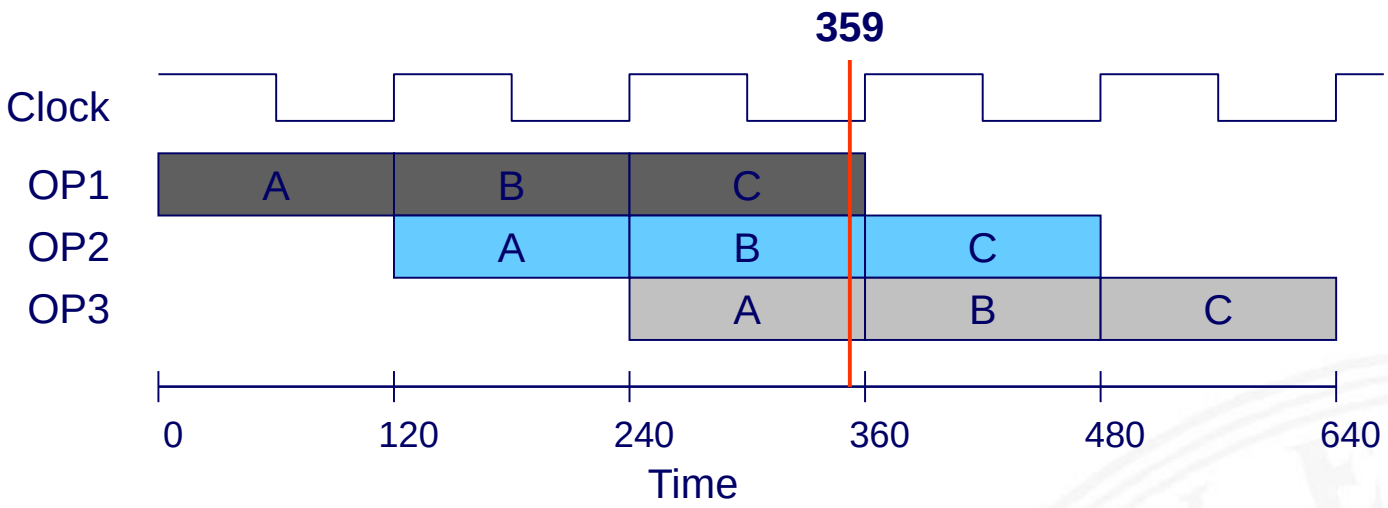
Timing: 3-stufige Pipeline

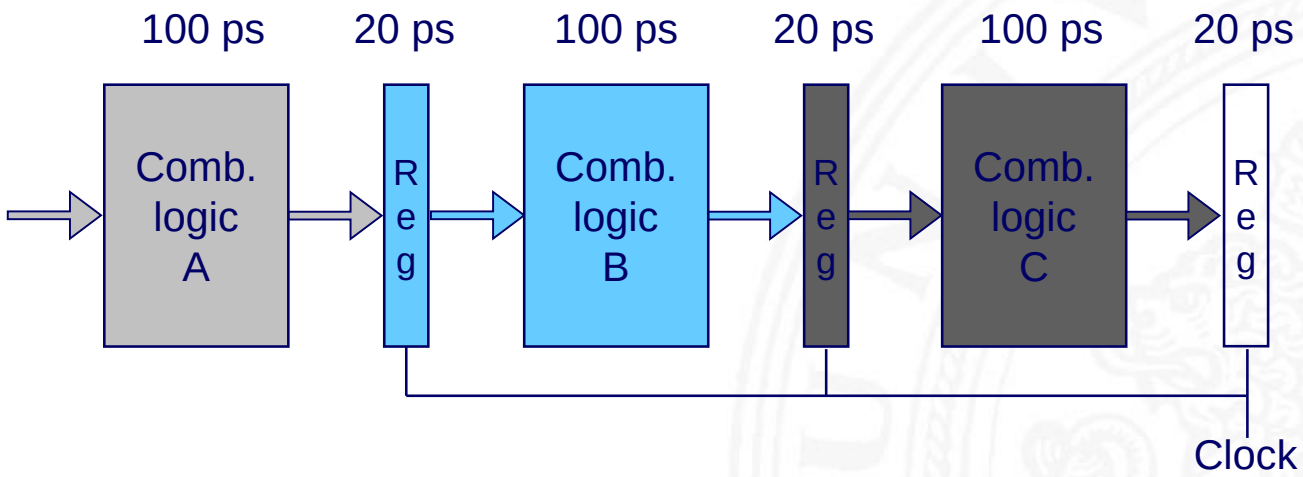




[BO15]

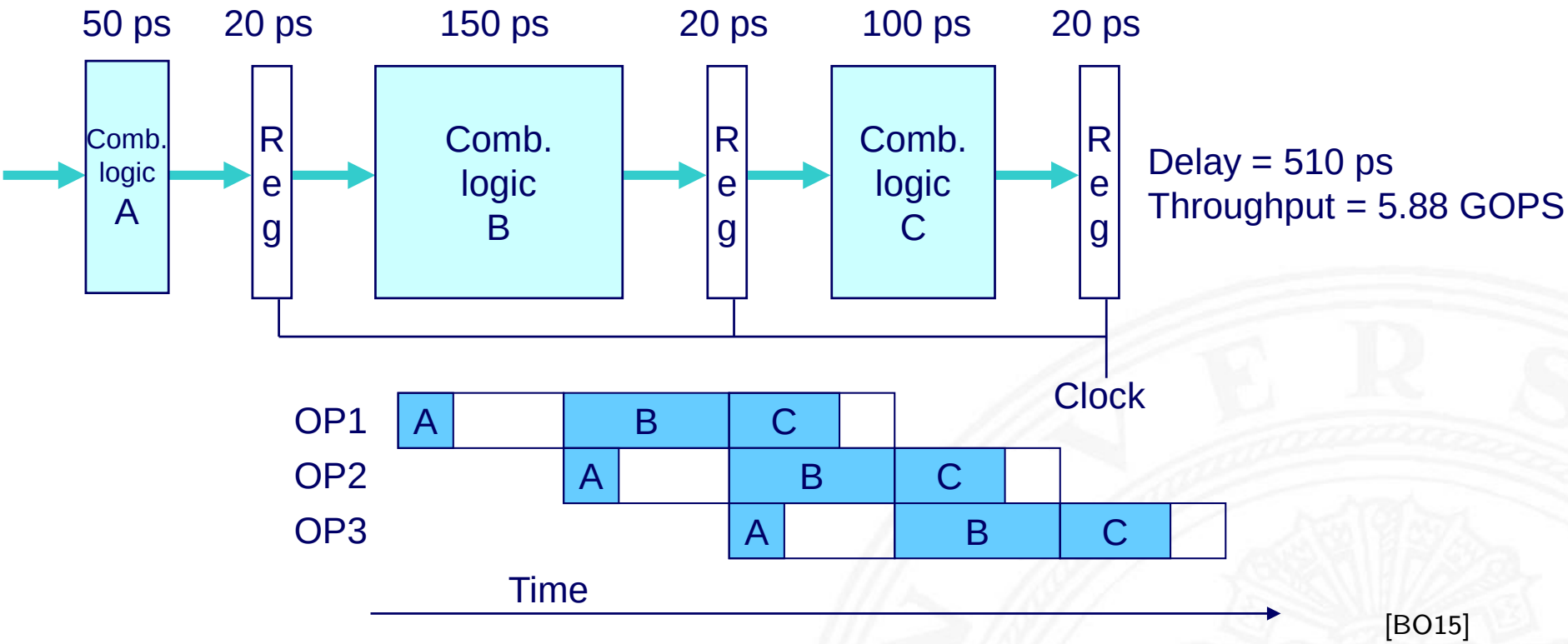
Timing: 3-stufige Pipeline





[BO15]

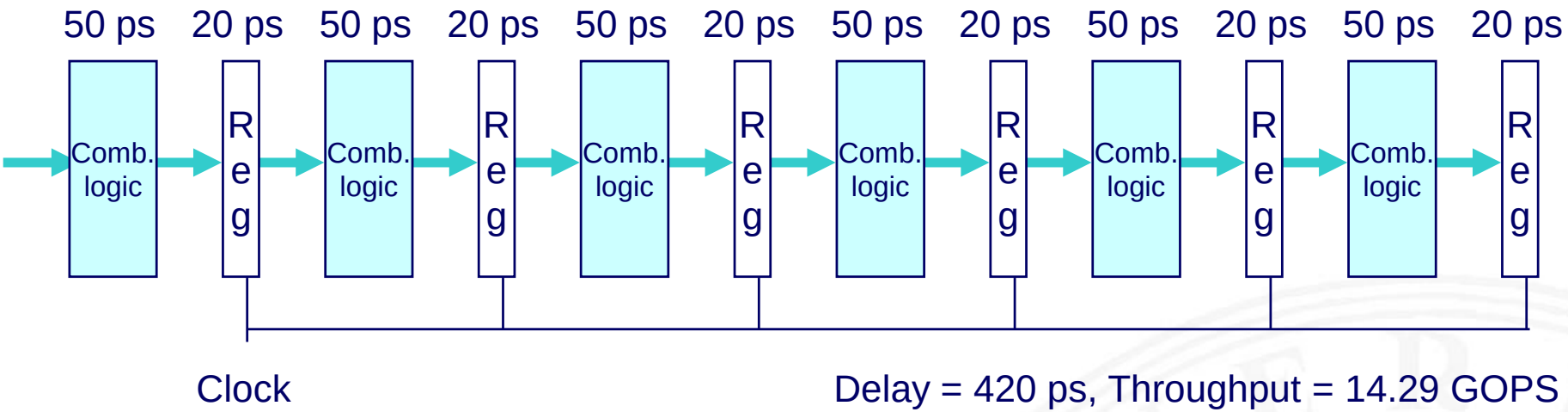
Limitierungen: nicht uniforme Verzögerungen



[BO15]

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

Limitierungen: Register „Overhead“

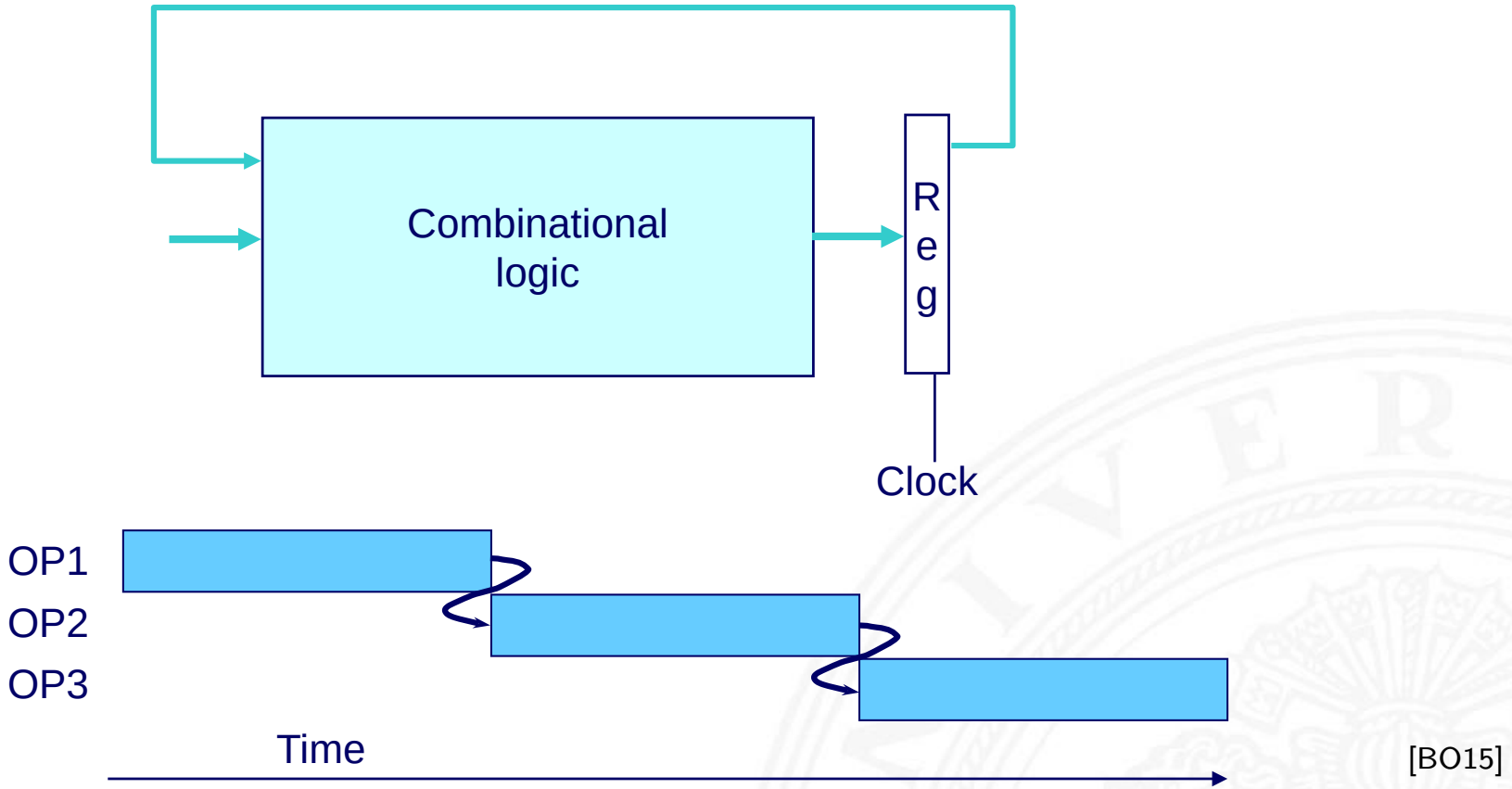


[BO15]

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

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

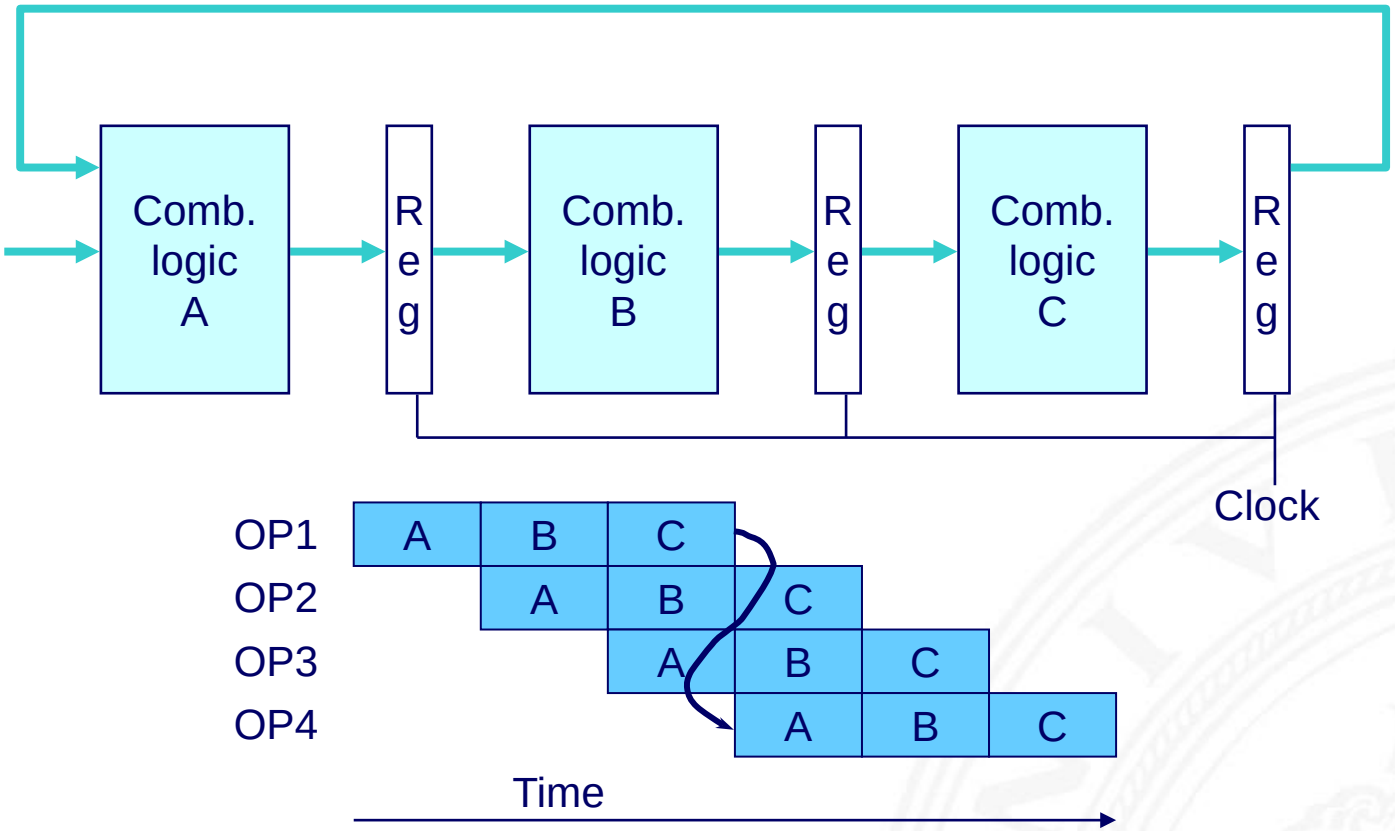
Limitierungen: Datenabhängigkeiten



[BO15]

- jede Operation hängt vom Ergebnis der Vorhergehenden ab

Limitierungen: Datenabhängigkeiten (cont.)



[BO15]

⇒ Resultat-Feedback kommt zu spät für die nächste Operation

⇒ Pipelining ändert Verhalten des gesamten Systems

von-Neumann Befehlszyklus

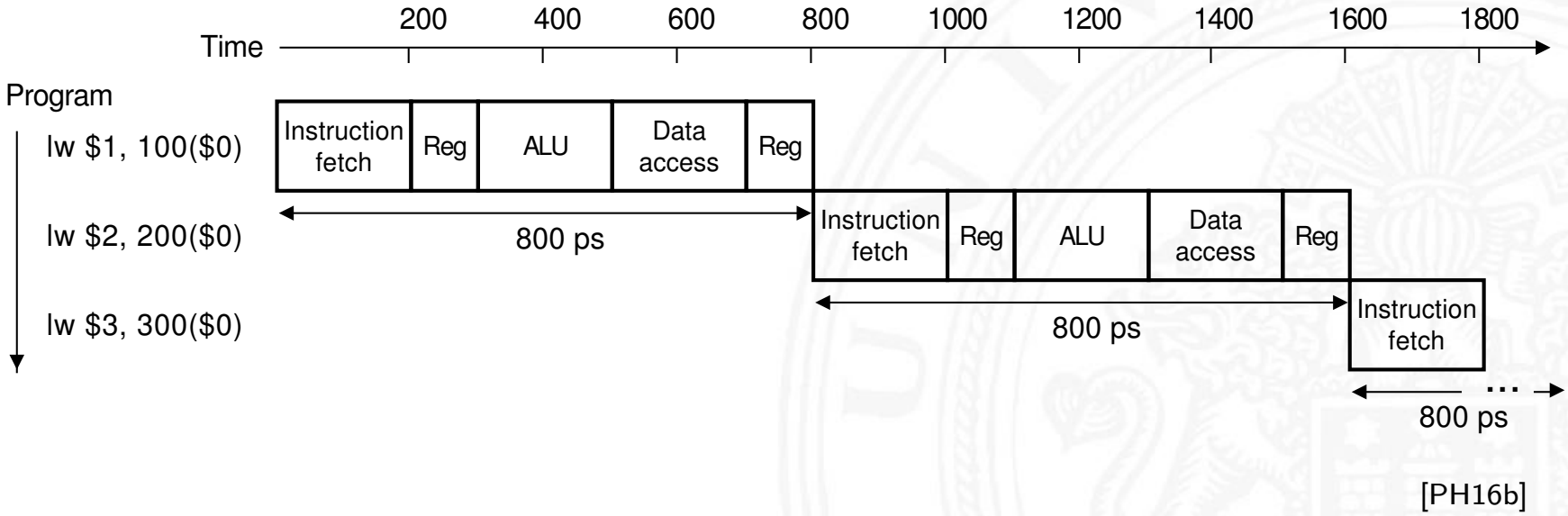
typische Schritte der Befehlsabarbeitung (von ISA abhängig)

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

von-Neumann Befehlszyklus (cont.)

- je nach Instruktion sind nicht alle Schritte notwendig
  - *nop*: nur Instruction-Fetch
  - *jump*: kein Speicher- und Registerzugriff
  - *aluOp*: kein Speicherzugriff
- Pipeline kann auch feiner unterteilt werden (meist mehr Stufen)

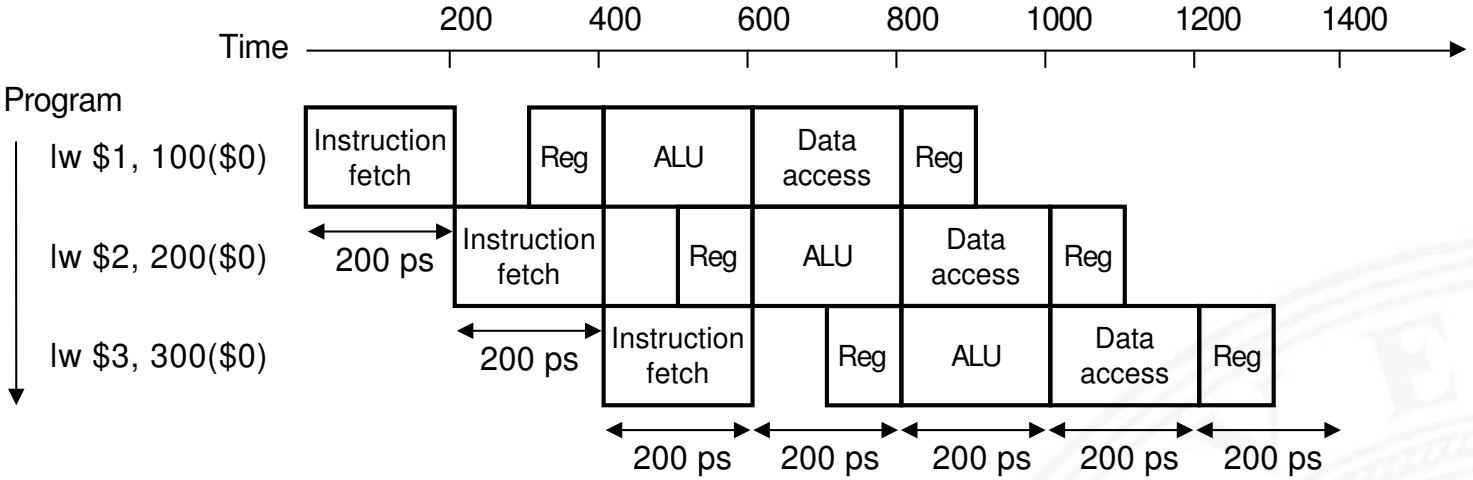
serielle Bearbeitung ohne Pipelining



[PH16b]

von-Neumann Befehlszyklus (cont.)

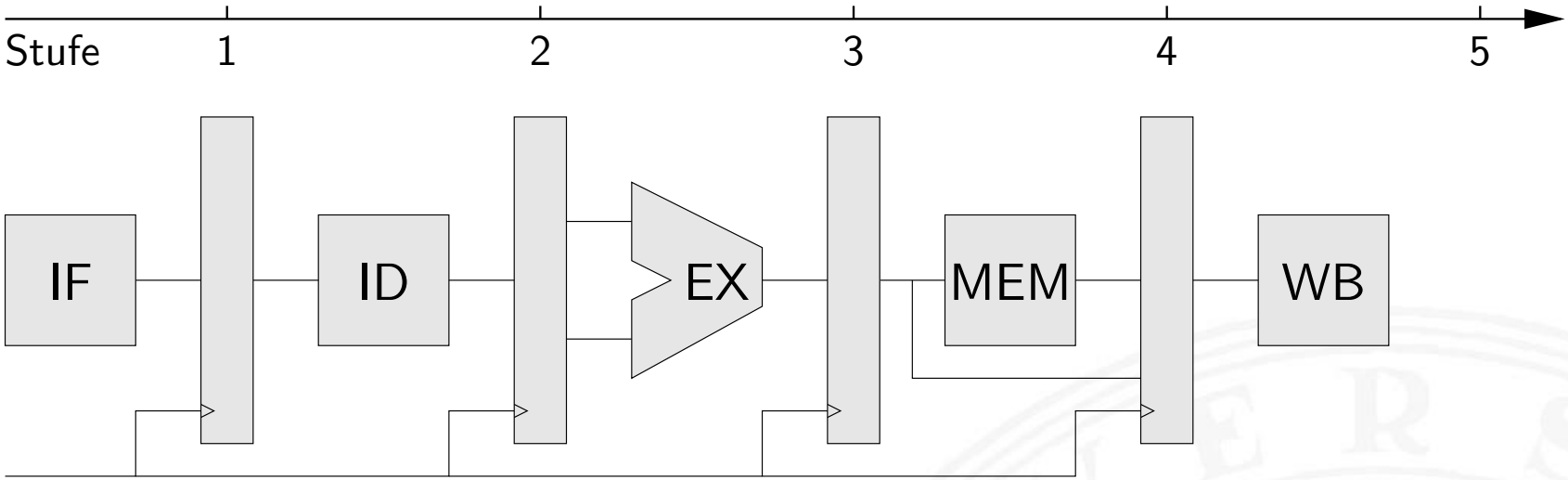
Pipelining für die einzelnen Schritte der Befehlsausführung



[PH16b]

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

Klassische 5-stufige Pipeline

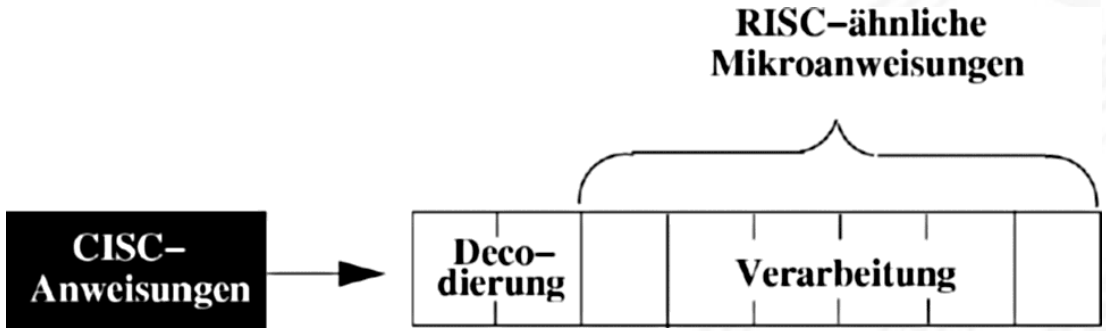


- Grundidee der ursprünglichen RISC-Architekturen
- \+ Durchsatz ca. 3 . . . 5 × besser als serielle Ausführung
- \+ guter Kompromiss aus Leistung und Hardwareaufwand

- MIPS-Architektur (aus Patterson, Hennessy [PH16b])
  - ▸ MIPS ohne Pipeline
  - ▸ MIPS Pipeline
  - ▸ Pipeline Schema

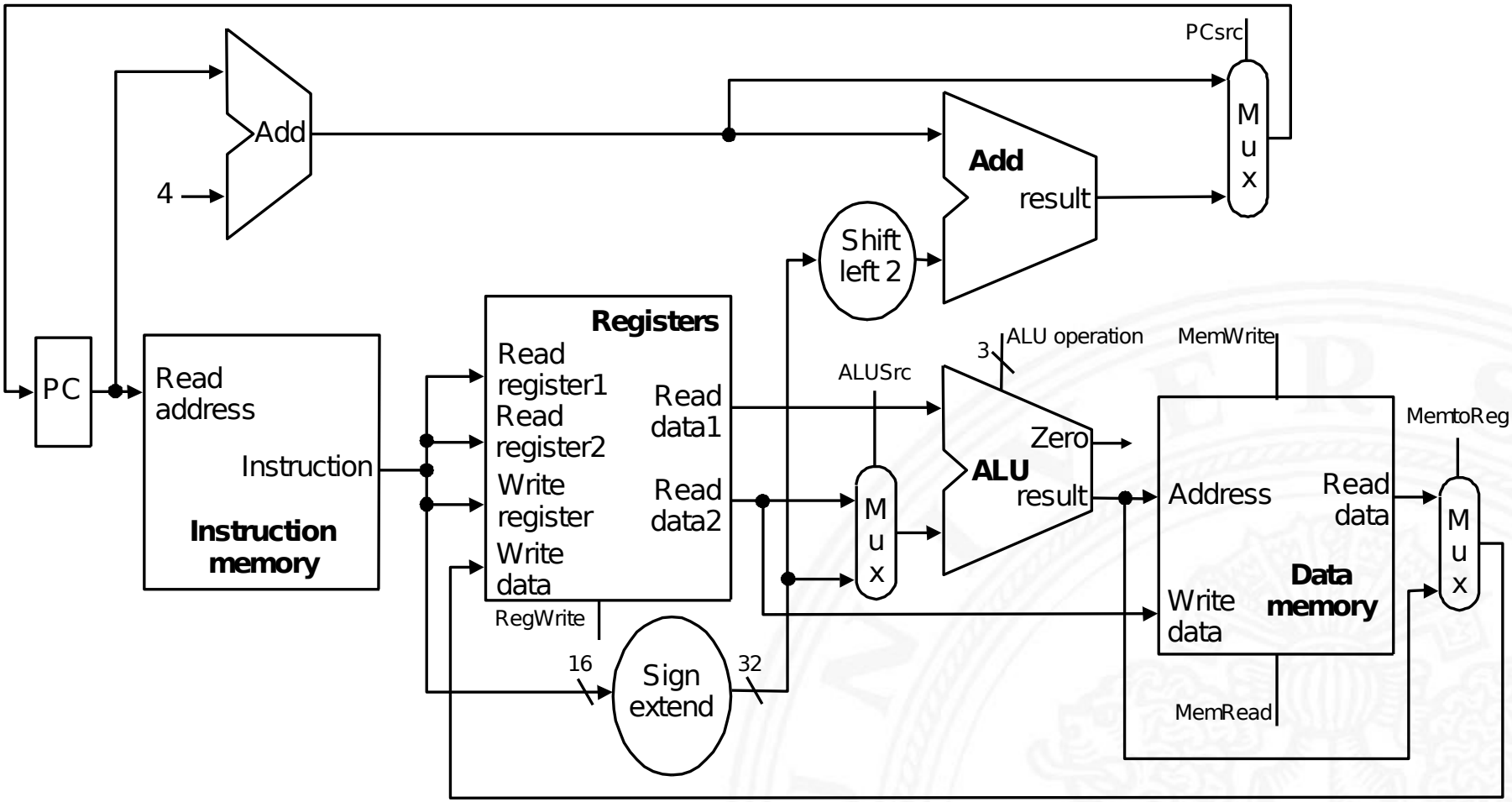
Klassische 5-stufige Pipeline (cont.)

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



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

MIPS: serielle Realisierung ohne Pipeline

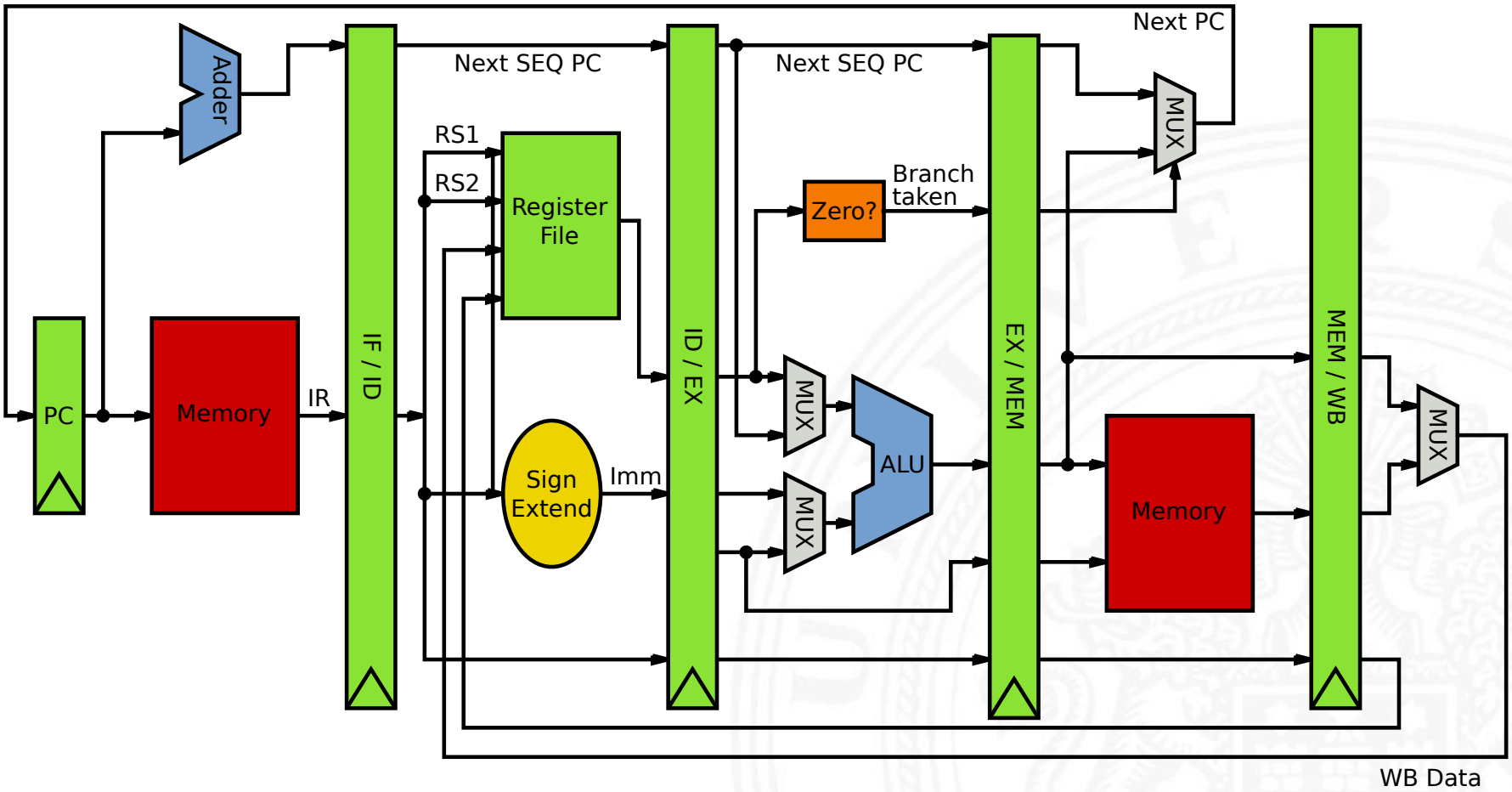


längster Pfad: PC - IM - REG - MUX - ALU - DM - MUX - PC/REG [PH16b]

◂ RISC Pipelining

MIPS: mit 5-stufiger Pipeline

Instruction Fetch	Instruction Decode Register Fetch	Execute Address Calc.	Memory Access	Write Back
IF	ID	EX	MEM	WB



◂ RISC Pipelining

MIPS: mit 5-stufiger Pipeline (cont.)

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

Prozessorpipeline – Begriffe

Begriffe

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

Prozessorpipeline – Bewertung

Vor- und Nachteile

+ Schaltnetze in kleinere Blöcke aufgeteilt $\Rightarrow$ höherer Takt
+ im Idealfall ein neuer Befehl pro Takt gestartet $\Rightarrow$ höherer Durchsatz, bessere Performanz
+ geringer Zusatzaufwand an Hardware
+ Pipelining ist für den Programmierer nicht direkt sichtbar!
  - Achtung: Daten-/Kontrollabhängigkeiten (s.u.)
- Latenz wird nicht verbessert, bleibt bestenfalls gleich
- Pipeline Takt limitiert durch langsamste Pipelinestufe unausgewogene Pipelinestufen reduzieren den Takt und damit die Performanz
- zusätzliche Zeiten, um Pipeline zu füllen bzw. zu leeren

Prozessorpipeline – Bewertung (cont.)

Analyse

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

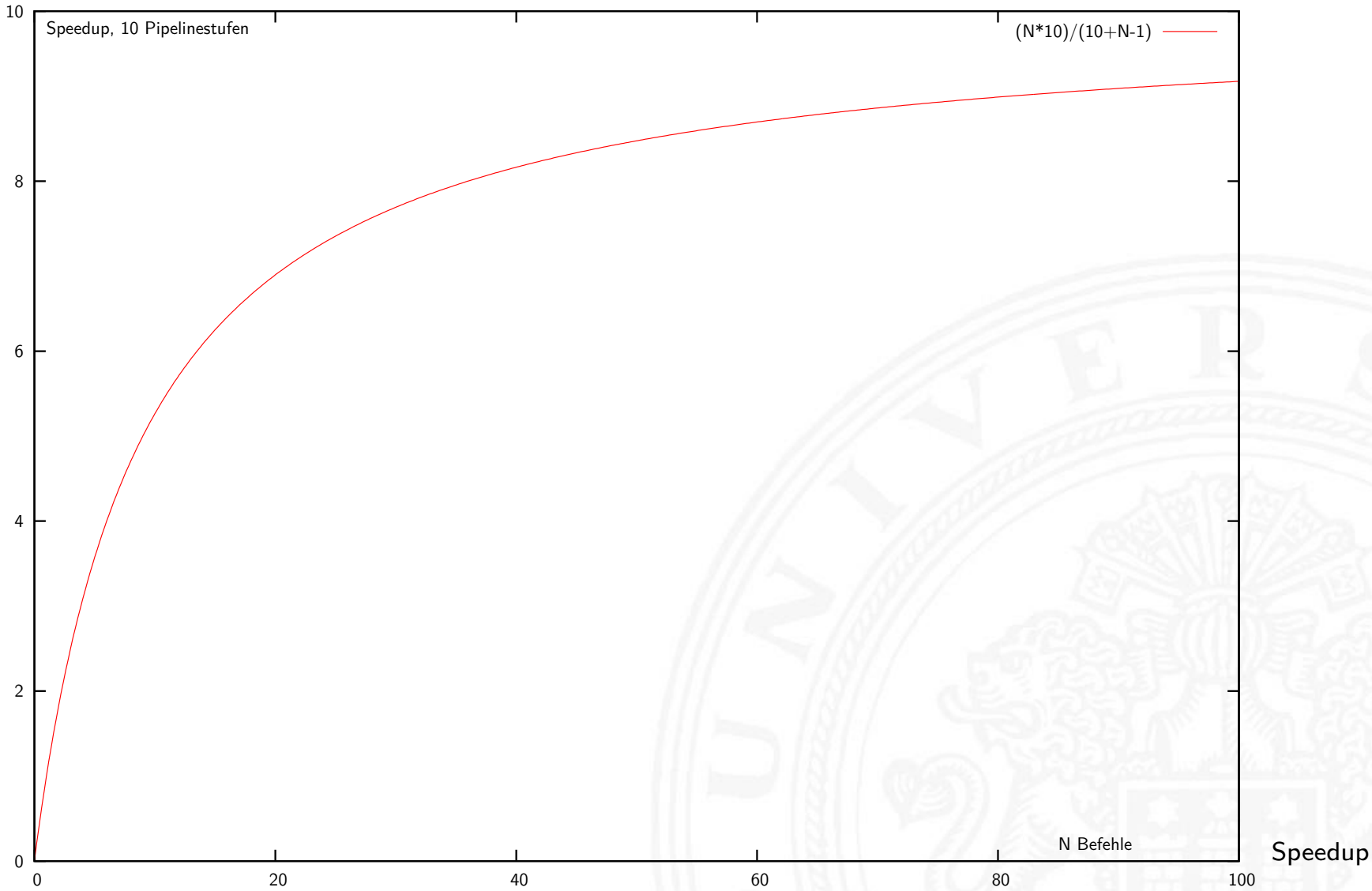
- „Speedup“ $S = \frac{N \cdot K}{K+N-1}$, $\lim_{N \to \infty} S = K$

$\Rightarrow$ ein großer Speedup wird erreicht durch

- große Pipelinetiefe: K
- lange Instruktionssequenzen: N

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

Prozessorpipeline – Bewertung (cont.)



Prozessorpipeline – Dimensionierung

- größeres K wirkt sich direkt auf den Durchsatz aus
- weniger Logik zwischen den Registern, höhere Taktfrequenzen
- zusätzlich: technologischer Fortschritt (1985 . . . 2017)
- Beispiele

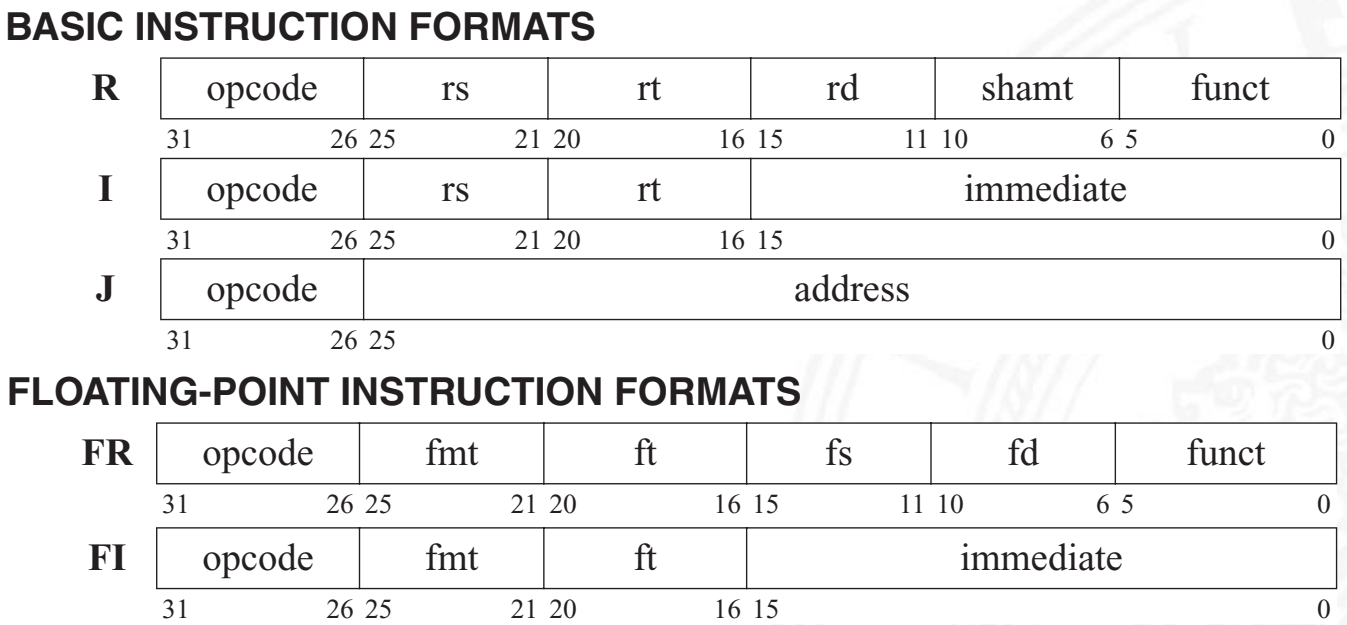
CPU	Pipelinestufen	Taktfrequenz $[MHz]$
80 386	1	33
Pentium	5	300
Motorola G4	4	500
Motorola G4e	7	1 000
Pentium II/III	12	1 400
Athlon XP	10/15	2 500
Athlon 64, Opteron	12/17	$\leq$ 3 000
Pentium 4	20	$\leq$ 3 800
Core i-..	14/19	$\leq$ 4 200
Ryzen ..	19	$\leq$ 4 000

Prozessorpipeline – Auswirkungen

Architekturentscheidungen, die sich auf das Pipelining auswirken

gut für Pipelining

- gleiche Instruktionslänge
- wenige Instruktionsformate
- Load/Store Architektur



MIPS-Befehlsformate [PH16b]

Prozessorpipeline – Auswirkungen (cont.)

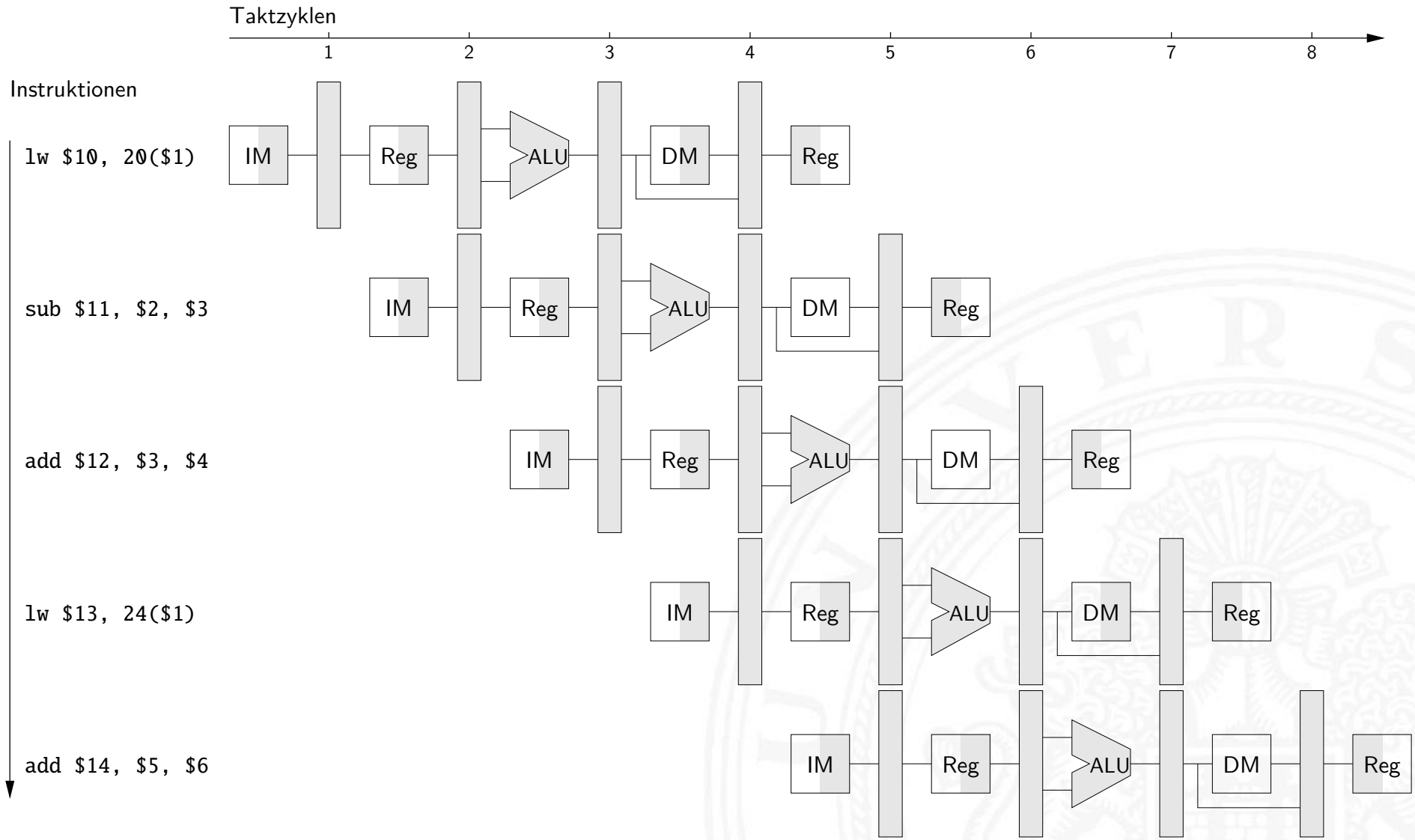
schlecht für Pipelining: *Pipelinekonflikte / -Hazards*

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

sehr schlecht für Pipelining

- Unterbrechung des Programmkontexts („*Context Switch*“): Interrupt, System-Call, Exception, Prozesswechsel . . .

Pipeline Schema



RISC Pipelining

Motivation: ständig steigende Anforderungen

- Simulationen, Wettervorhersage, Gentechnologie ...
- Datenbanken, Transaktionssysteme, Suchmaschinen ...
- Softwareentwicklung, Schaltungsentwurf ...

- Performanz eines einzelnen Prozessors ist begrenzt

$\Rightarrow$ Hardware: Verteilen eines Programms auf mehrere Prozessoren

$\Rightarrow$ Software: kommunizierende Prozesse und Multithreading

Vielfältige Möglichkeiten

- wie viele und welche Prozessoren?
- Kommunikation zwischen den Prozessoren?
- Programmierung und Software/Tools?

Begriffe

- **Antwortzeit**: die Gesamtzeit zwischen Programmstart und -ende, inklusive I/O-Operationen, Unterbrechungen etc. („*wall clock time*“, „*response time*“, „*execution time*“)

  $$\text{performance} = \frac{1}{\text{execution time}}$$

- **Ausführungszeit**: reine CPU-Zeit

  Unix `time`-Befehl:

  ```
  597.07u 0.15s 9:57.61 99.9%
  ```

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

- **Durchsatz**: Anzahl der bearbeiteten Programme / Zeit

- **Speedup**: $s = \frac{\text{performance x}}{\text{performance y}} = \frac{\text{execution time y}}{\text{execution time x}}$

Wie kann man Performanz verbessern?

- Ausführungszeit = $\langle \textit{Anzahl der Befehle} \rangle \cdot \langle \textit{Zeit pro Befehl} \rangle$

- weniger Befehle
  - *gute* Algorithmen
  - bessere Compiler
  - mächtigere Befehle (CISC)
- weniger Zeit pro Befehl
  - bessere Technologie
  - Architektur: Pipelining, Caches ...
  - einfachere Befehle (RISC)
- parallele Ausführung
  - superskalare Architekturen, SIMD, MIMD

Amdahl's Gesetz

Möglicher Speedup durch Beschleunigung einer Teilfunktion?

1. System berechnet Programm P,
   darin Funktion X mit Anteil $0 < f < 1$ der Gesamtzeit
2. System berechnet Programm P,
   Funktion X' ist schneller als X mit Speedup S_X

Amdahl's Gesetz

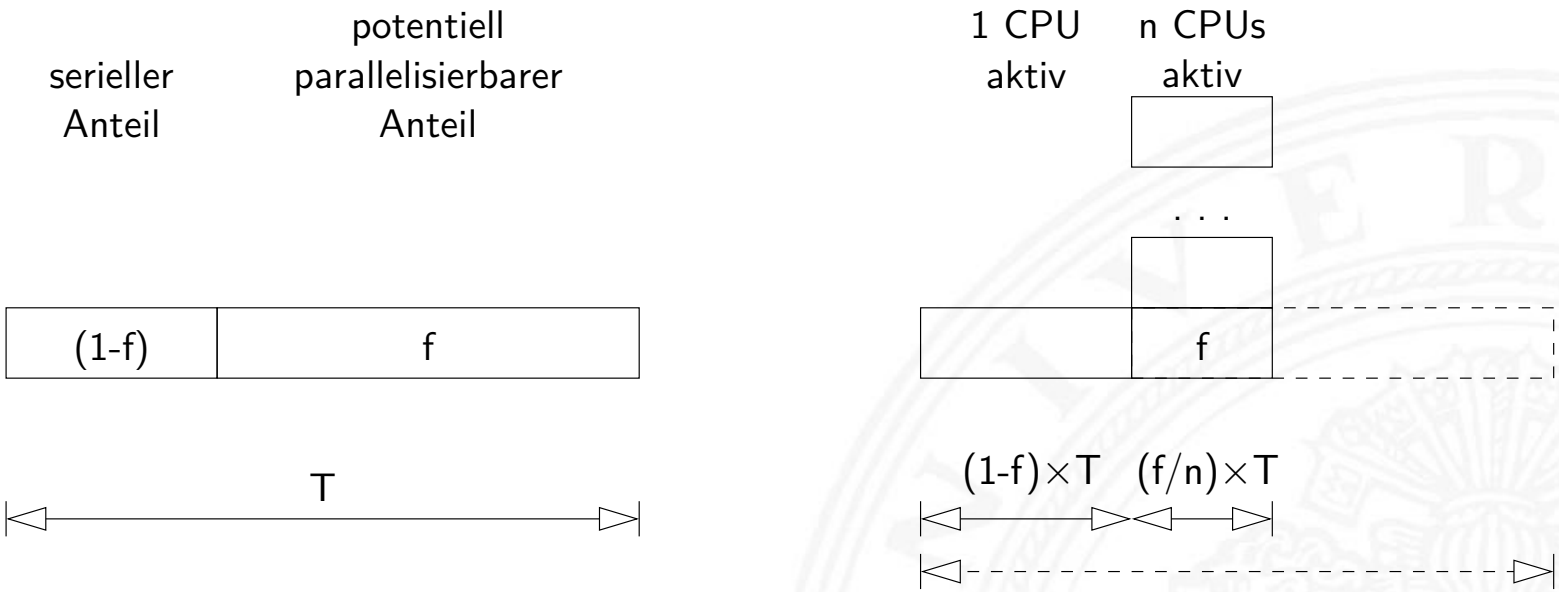
Gene Amdahl, Architekt der IBM S/360, 1967

- Speedup $$S_{gesamt} = \frac{1}{(1-f) + f/S_X}$$

Amdahl's Gesetz (cont.)

Speedup $$S_{gesamt} = \frac{1}{(1-f) + f/S_X}$$

- nur ein Teil f des Gesamtproblems wird beschleunigt



$\Rightarrow$ möglichst großer Anteil f

$\Rightarrow$ Optimierung lohnt nur für relevante Operationen

allgemeingültig: entsprechend auch für Projektplanung, Verkehr ...

Amdahl's Gesetz (cont.)

- ursprüngliche Idee: Parallelrechner mit n-Prozessoren

Speedup $$S_{gesamt} = \frac{1}{(1-f) + k(n) + f/n}$$

n	# Prozessoren als Verbesserungsfaktor
f	Anteil parallelisierbarer Berechnung
$1-f$	Anteil nicht parallelisierbarer Berechnung
$k()$	Kommunikationsoverhead zwischen den Prozessoren

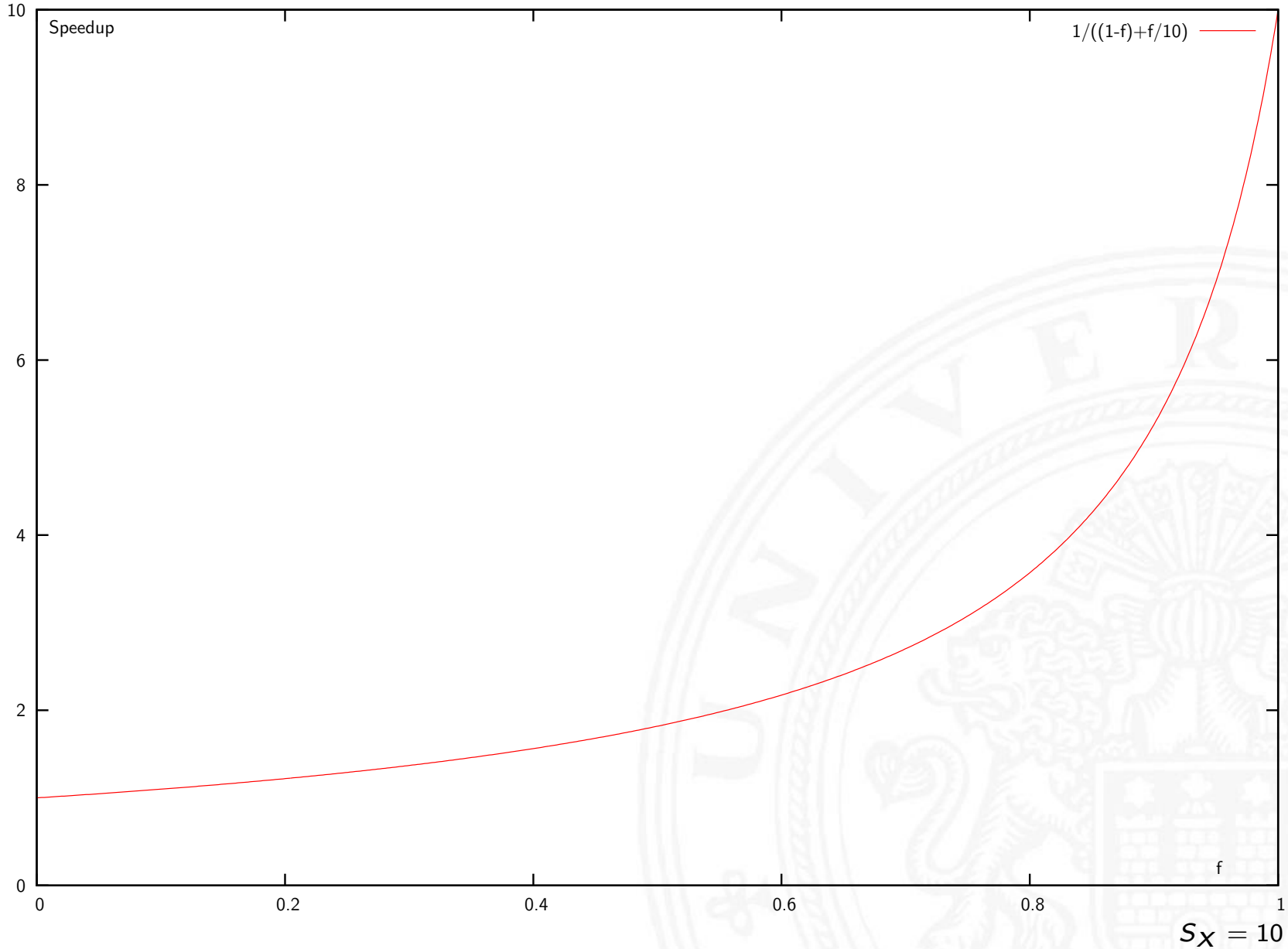
- Aufgaben verteilen
- Arbeit koordinieren
- Ergebnisse zusammensammeln

Amdahl's Gesetz: Beispiele

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

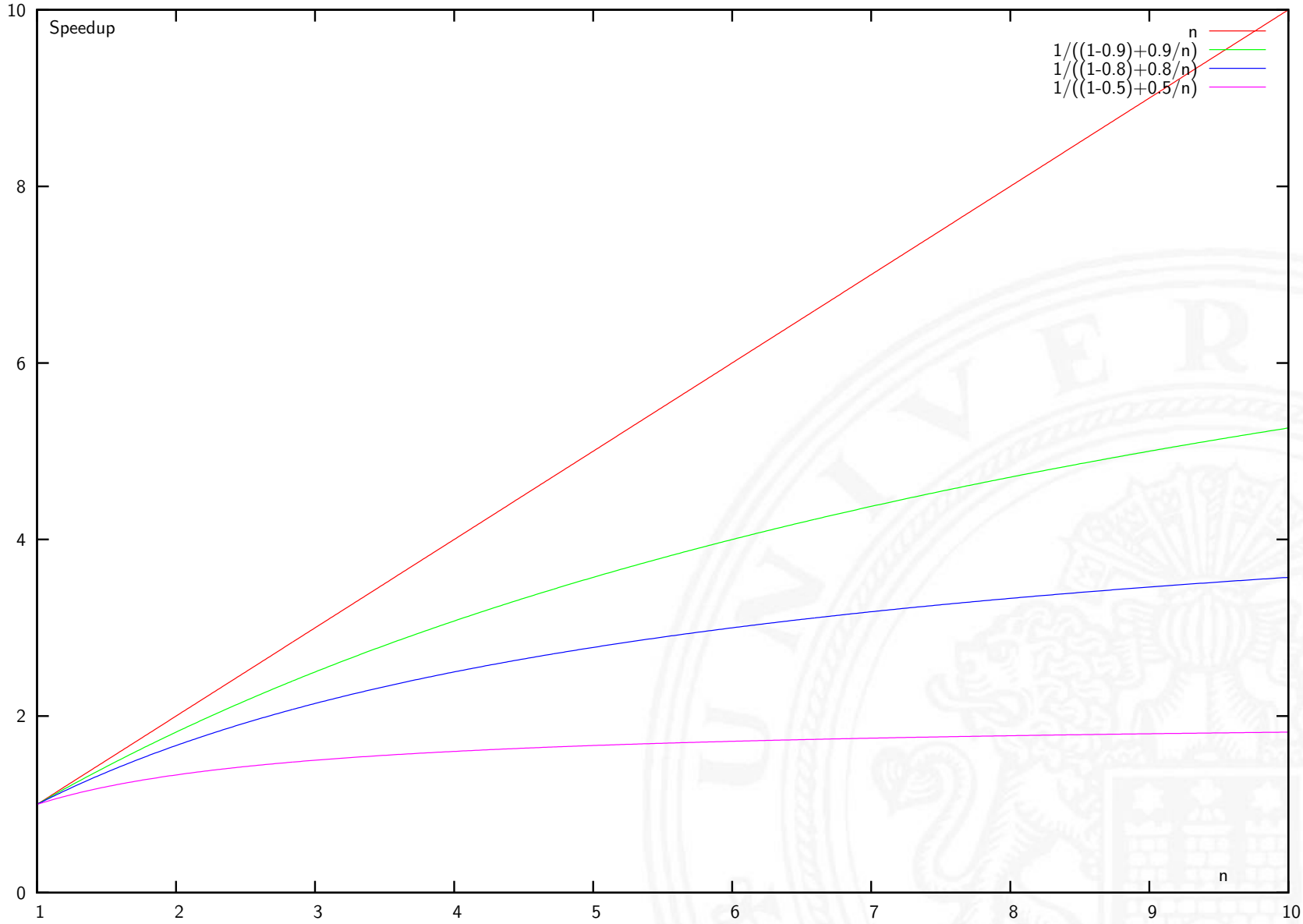
- Optimierung bringt nichts, wenn der nicht beschleunigte „serielle“ Anteil $(1 - f)$ eines Programms überwiegt
- – n-Prozessoren (große S_X) wirken *nicht linear*
- – die erreichbare Parallelität in Hochsprachen-Programmen ist gering, typisch $S_{gesamt} \leq 4$
- \+ viele Prozesse/Tasks, unabhängig voneinander: Serveranwendungen, virtuelle Maschinen, Container ...

Amdahl's Gesetz: Beispiele (cont.)

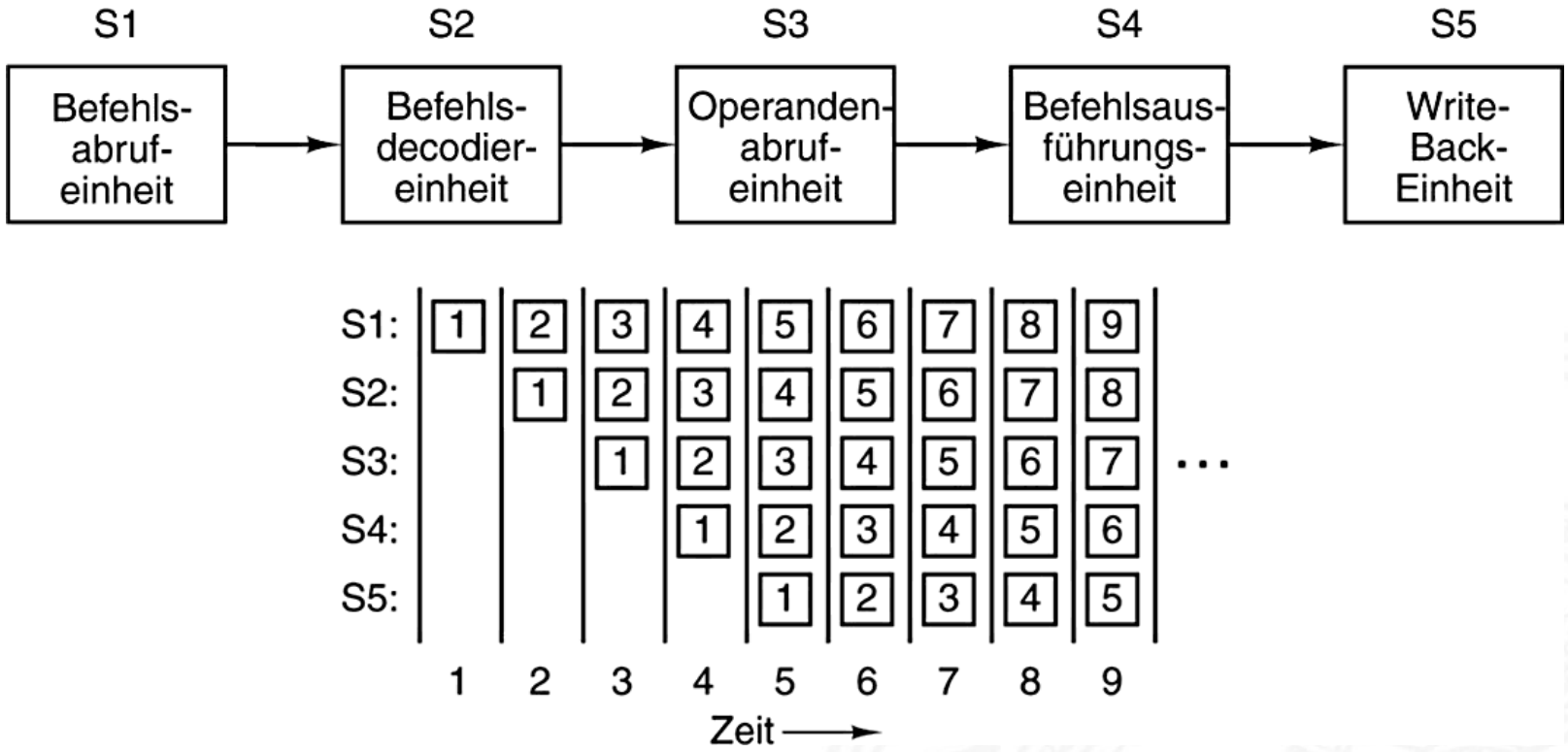


$S_X = 10$

Amdahl's Gesetz: Beispiele (cont.)



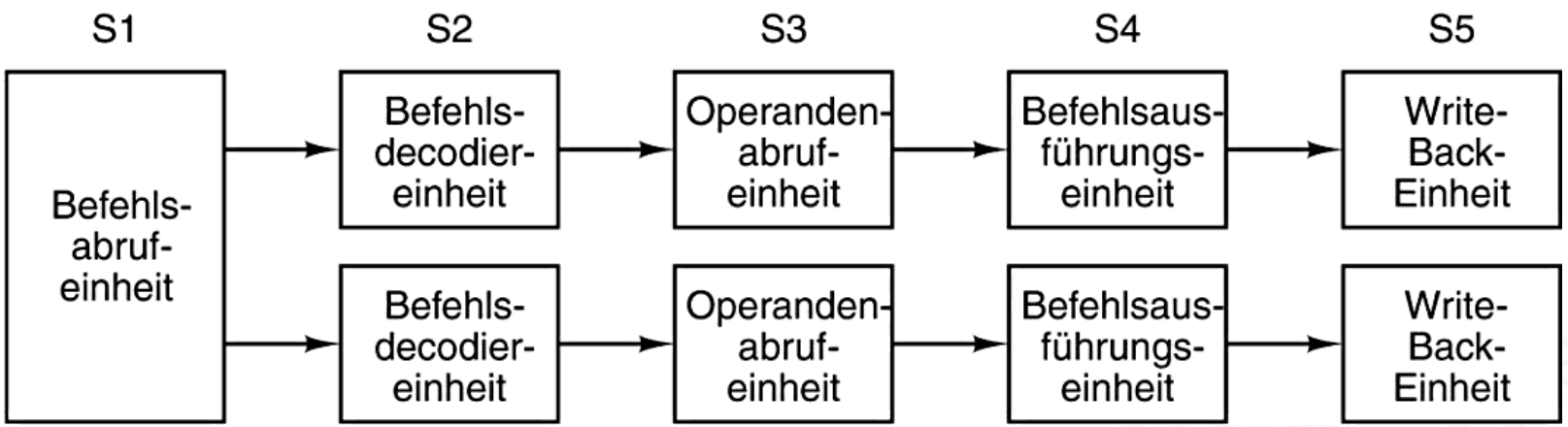
Befehlspipeline



[TA14]

- Befehl in kleinere, schnellere Schritte aufteilen ⇒ höherer Takt
- mehrere Instruktionen überlappt ausführen ⇒ höherer Durchsatz

Parallele Pipelines



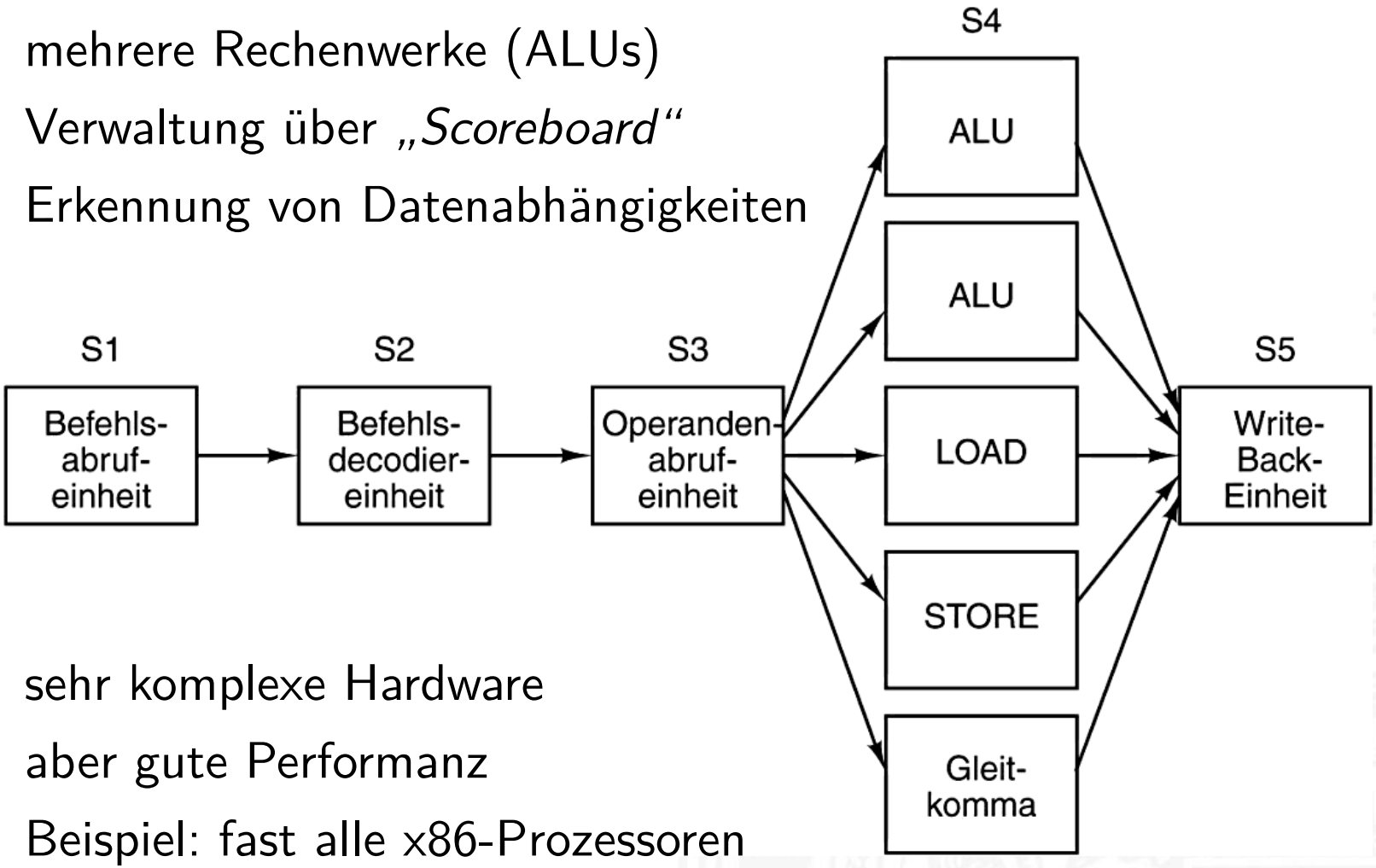
[TA14]

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

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

Superskalarer Prozessor

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



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

[TA14]

Superskalar

- Superskalare CPUs besitzen mehrere Recheneinheiten: 4 . . . 12
- in jedem Takt werden (dynamisch) mehrere Instruktionen eines konventionell linearen Instruktionsstroms abgearbeitet

$\Rightarrow$ ILP (**I**nstruction **L**evel **P**arallelism)

- Hardware verteilt initiierte Instruktionen auf Recheneinheiten
- pro Takt kann *mehr als eine* Instruktion initiiert werden
  Die Anzahl wird dynamisch von der Hardware bestimmt:
  0 . . . „*Instruction Issue Bandwidth*“

\+ sehr effizient, alle modernen CPUs sind superskalar

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

Superskalar – Datenabhängigkeiten

Datenabhängigkeiten

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

Datenabhängigkeiten superskalarer Prozessoren

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

Superskalar – Datenabhängigkeiten (cont.)

„Register Renaming“

- Hardware löst (einige) Datenabhängigkeiten der Pipeline auf
- zwei Arten von Registersätzen
  1. Architektur-Register: „logische Register“ der ISA
  2. viele Hardware-Register: „Rename Register“ (180 Int, 168 FP)
     - dynamische Abbildung von ISA- auf Hardware-Register
- Kontextwechsel aufwändig: „Rename Register“ speichern
- Beispiel
  - Originalcode

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

  nach Renaming

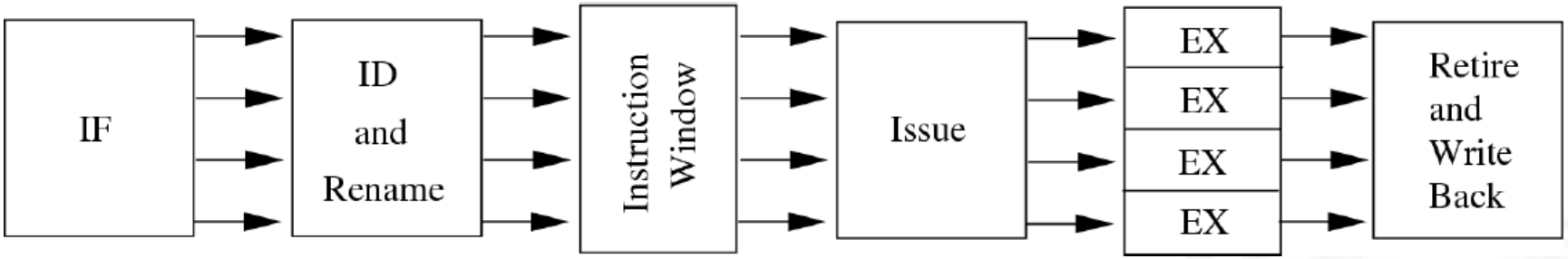
```
tmp1 = a + b;
res1 = c + tmp1;
tmp2 = d + e;
res2 = tmp2 - f;
tmp  = tmp2;
```

  - Parallelisierung des modifizierten Codes

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

Superskalar – Pipeline

Aufbau der superskalaren Pipeline

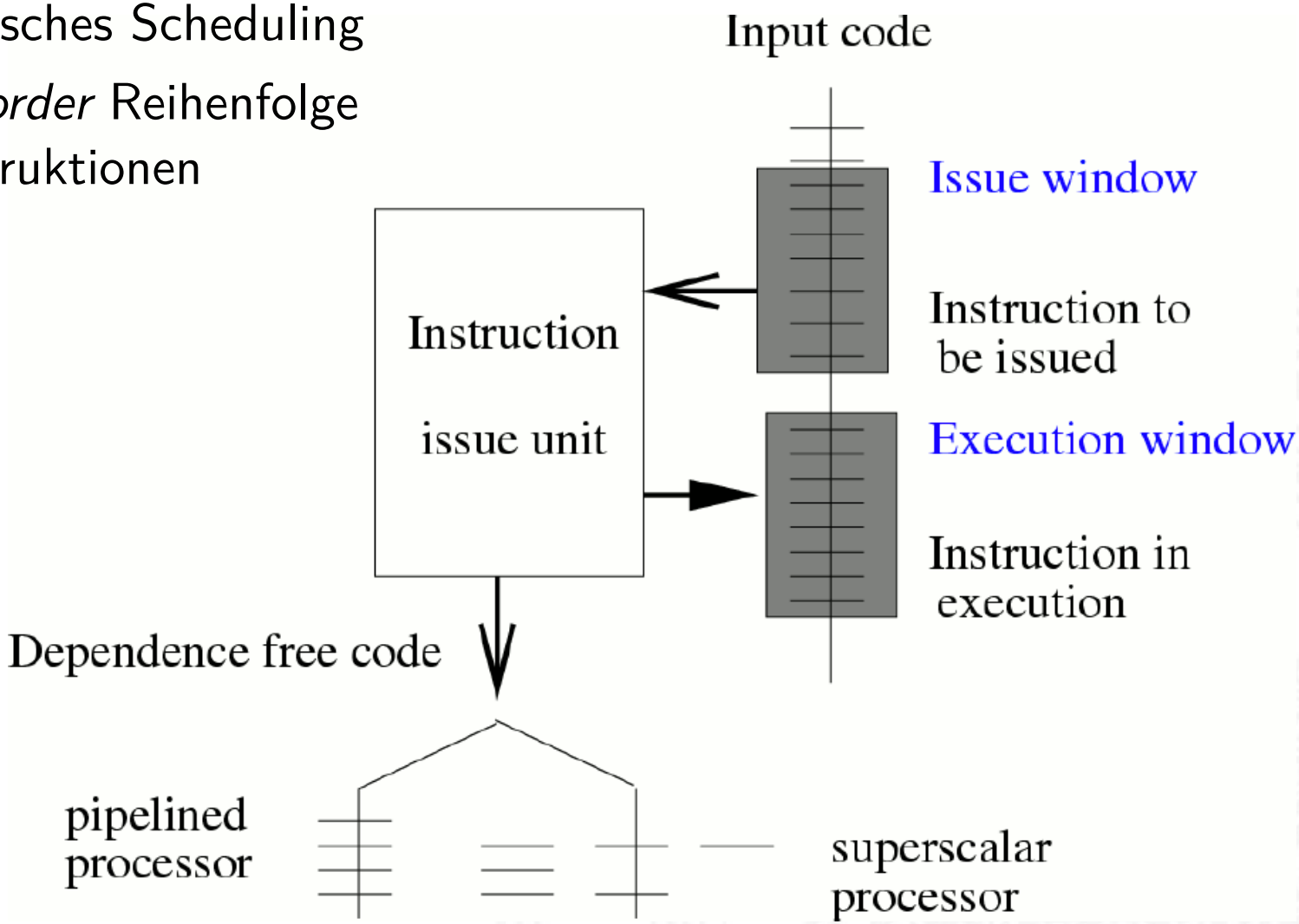


- lange Pipelines mit vielen Phasen: Fetch (Prefetch, Predecode), Decode / Register-Renaming, Issue, Dispatch, Execute, Retire (Commit, Complete / Reorder), Write-Back
- je nach Implementation unterschiedlich aufgeteilt
- entscheidend für superskalare Architektur sind die Schritte
  vor den ALUs: Issue,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

Superskalar – Pipeline (cont.)

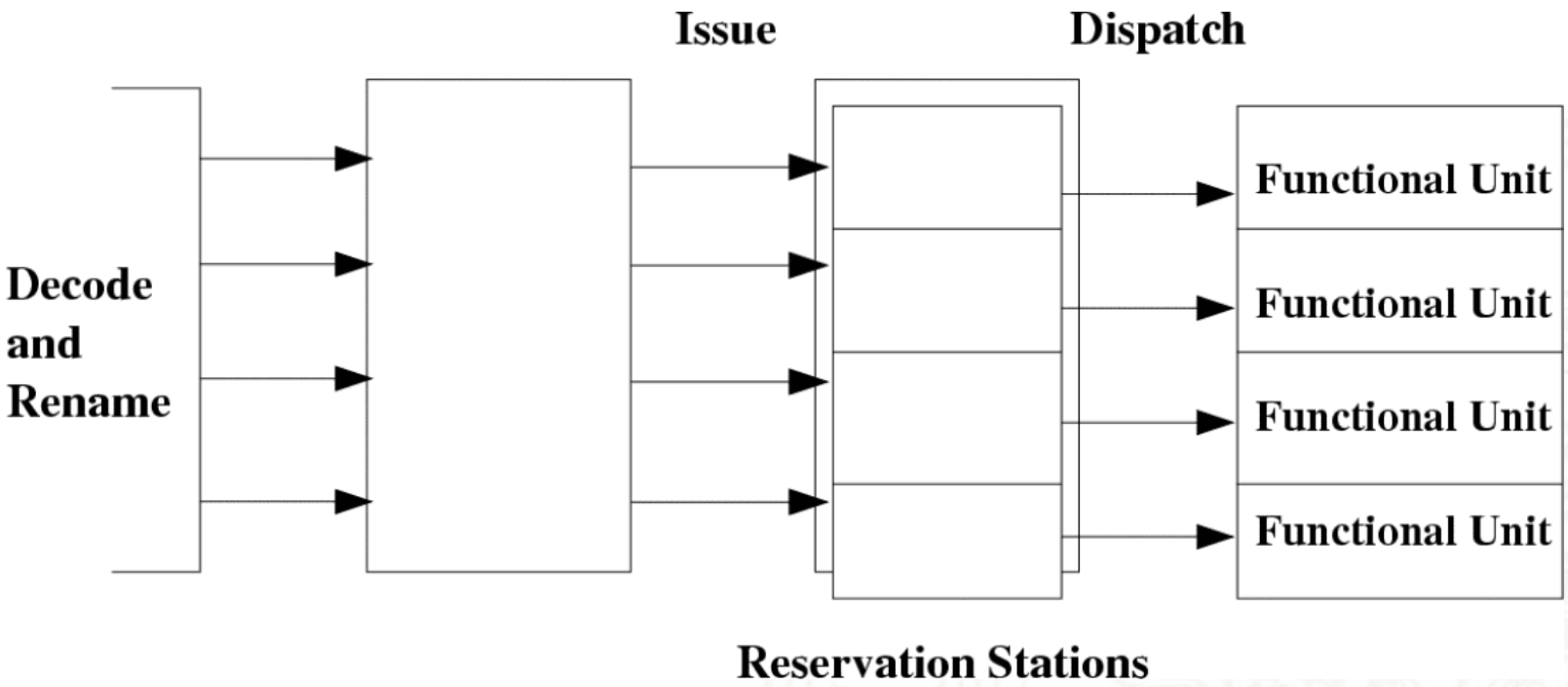
- Dynamisches Scheduling

⇒ *out-of-order* Reihenfolge der Instruktionen



Superskalar – Pipeline (cont.)

- Issue: globale Sicht
  Dispatch: getrennte Ausschnitte in „*Reservation Stations*“



Superskalar – Pipeline (cont.)

- Reservation Station für jede Funktionseinheit
  - speichert: initiierte Instruktionen die auf Recheneinheit warten
  - –"– zugehörige Operanden
  - –"– ggf. Zusatzinformation
  - Instruktion bleibt blockiert, bis alle Parameter bekannt sind und wird dann an die zugehörige ALU weitergeleitet
- ggf. „*Retire*“-Stufe
  - Reorder-Buffer: erzeugt wieder *in-order* Reihenfolge
  - commit: „richtig ausgeführte“ Instruktionen gültig machen
    abort: Instruktionen verwerfen, z.B. Sprungvorhersage falsch
- Dynamisches Scheduling: zuerst '67 in IBM 360 (R. Tomasulo)
  - Forwarding
  - Registerumbenennung und Reservation Stations

Superskalar – Probleme

Spezielle Probleme superskalarer Pipelines

- komplexe Datenabhängigkeiten
  - die verschiedenen ALUs haben unterschiedliche Latenzzeiten
  - Befehle „warten“ in den Reservation Stations

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

$\Rightarrow$ Kontextwechsel noch aufwändiger, muss ggf. warten

$\Rightarrow$ Optimierungstechniken wichtig

- optimiertes (dynamisches) Scheduling: Scoreboard, Tomasulo-Algorithmus
- *„Loop Unrolling“* (längere Codesequenzen ohne Sprünge)

Superskalar – Probleme (cont.)

Softwareunterstützung für Pipelining superskalarer Prozessoren
„*Software Pipelining*“

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

$\Rightarrow$ zusätzliche Optimierungsmöglichkeiten

Beispiel: Pentium 4 / NetBurst Architektur

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

Beispiel: Pentium 4 / NetBurst Architektur (cont.)

- große Pipelinelänge $\Rightarrow$ sehr hohe Taktfrequenzen

Basic Pentium III Processor Misprediction Pipeline

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

Basic Pentium 4 Processor Misprediction Pipeline

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

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

Beispiel: Pentium 4 / NetBurst Architektur (cont.)

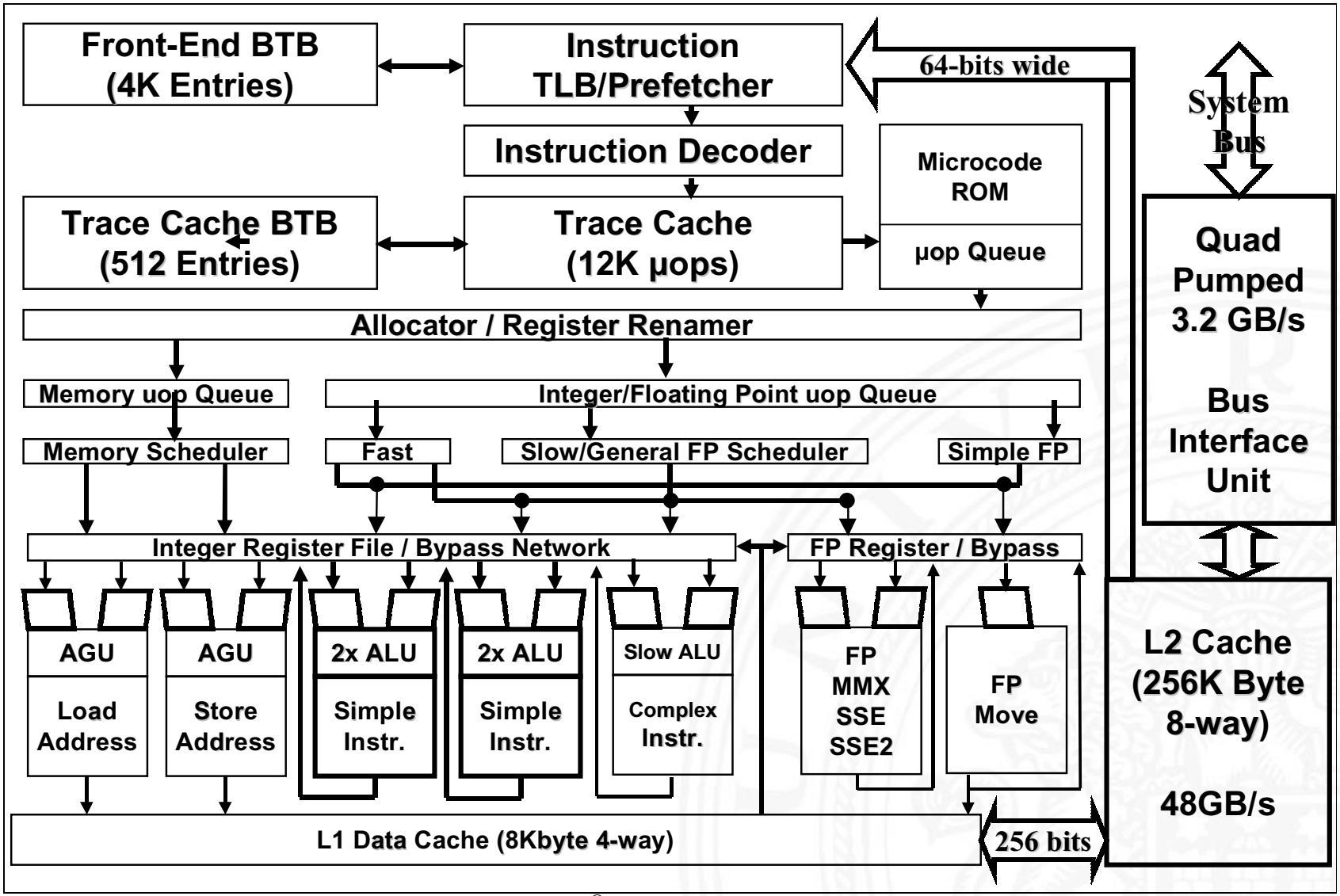
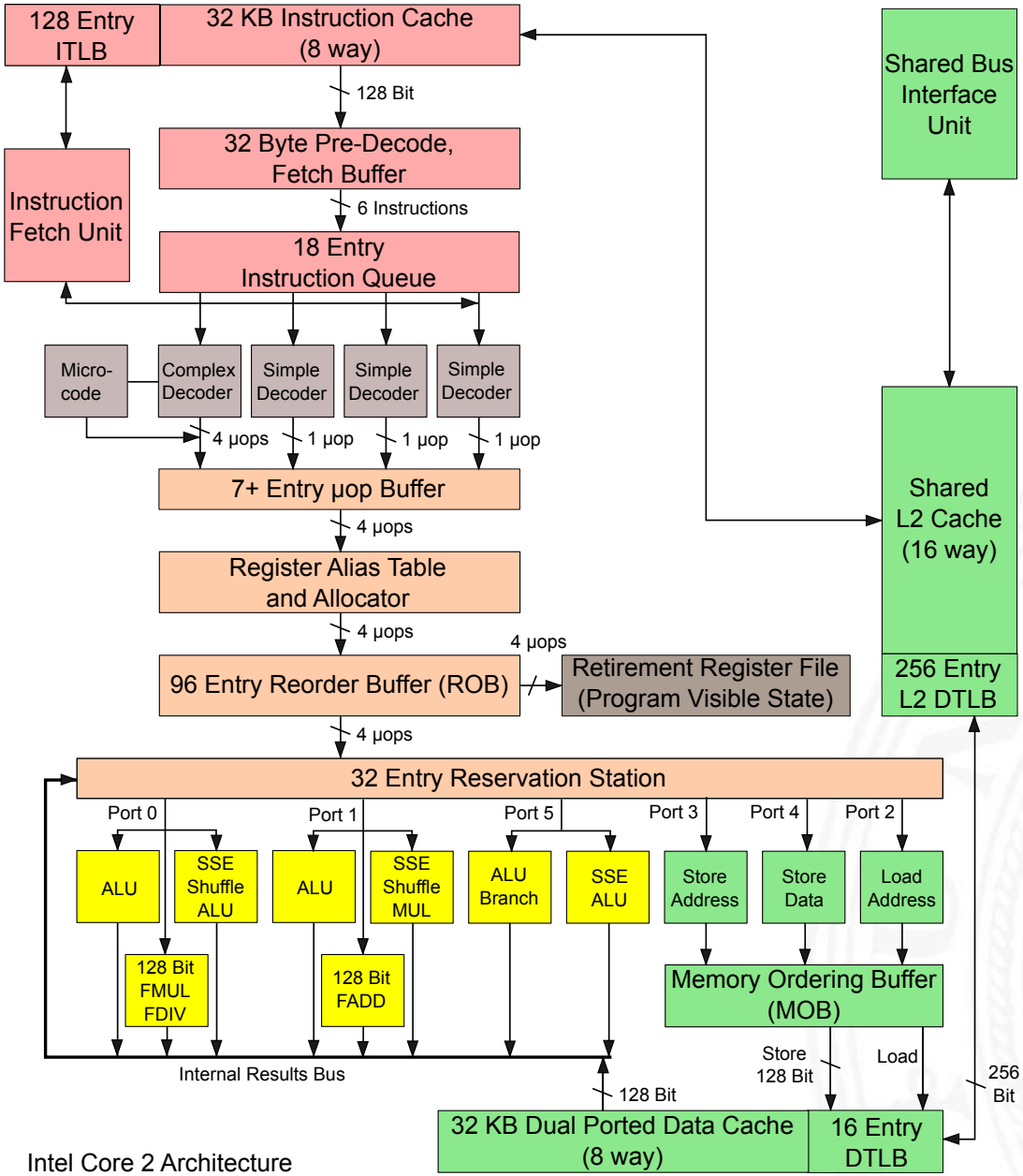


Figure 4: Pentium® 4 processor microarchitecture

Intel: Q1, 2001 [Intel]

Beispiel: Core 2 Architektur



Intel Core 2 Architecture

Parallelrechner mit mehreren Prozessoren

- *Moore's Law* – technischer Fortschritt
  - immer schnellere Schaltungen
  - immer mehr Transistoren pro IC möglich
- Taktfrequenzen $> 10\,\text{GHz}$ nicht sinnvoll realisierbar
  - hoher Takt nur bei einfacher Hardware möglich
  - Stromverbrauch bei CMOS proportional zum Takt

$\Rightarrow$ höhere Rechenleistung durch Mehrprozessorsysteme

- Datenaustausch
  1. gemeinsamer Speicher („*Shared-memory*“) oder
  2. Verbindungsnetzwerk („*Message-passing*“)
- Probleme
  - Overhead durch Kommunikation
  - Parallelität in den Algorithmen
  - Komplexität bei der Programmierung

Flynn-Klassifikation

SISD *„Single Instruction, Single Data“*

- jeder klassische von-Neumann Rechner (z.B. PC)

SIMD *„Single Instruction, Multiple Data“*

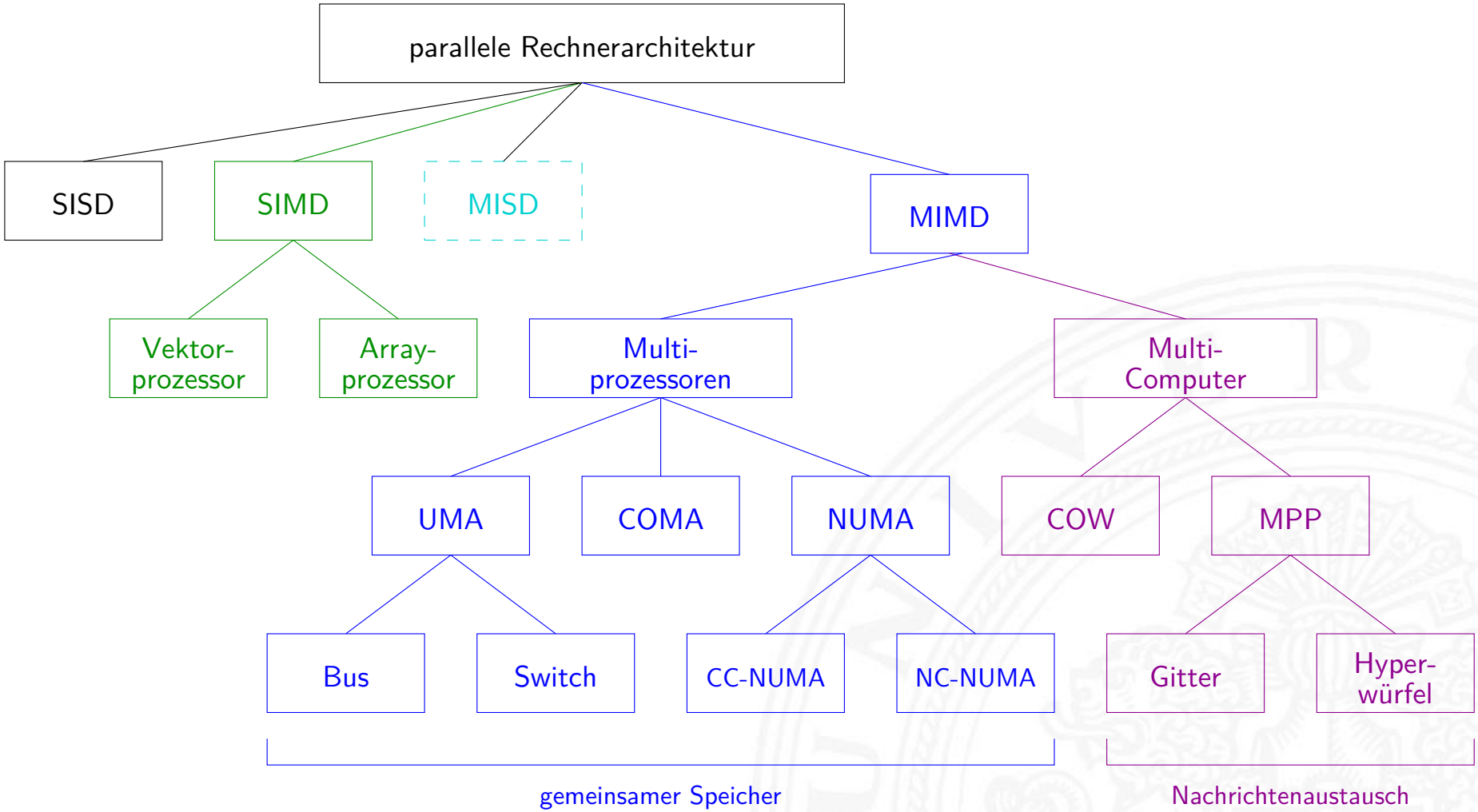
- Vektorrecher/Feldrechner
  z.B. Connection-Machine 2: 65 536 Prozessoren
- Erweiterungen in Befehlssätzen: superskalare Recheneinheiten werden direkt angesprochen
  z.B. x86 MMX, SSE, VLIW-Befehle: 2 . . . 8 fach parallel

MIMD *„Multiple Instruction, Multiple Data“*

- Multiprozessormaschinen
  z.B. Compute-Cluster, aber auch Multi-Core CPU

MISD *„Multiple Instruction, Single Data“* :-)

Detaillierte Klassifikation



[TA14]

Anmerkungen

- Programmierung: ein ungelöstes Problem
  - Aufteilung eines Programms auf die CPUs/Rechenknoten?
  - insbesondere bei komplexen Kommunikationsnetzwerken
- Programme sind nur teilweise parallelisierbar
  - Parallelität einzelner Programme: kleiner 8
    gilt für Desktop-, Server-, Datenbankanwendungen etc.
  - $\Rightarrow$ hochgradig parallele Rechner sind dann Verschwendung
- *Wohin mit den Transistoren aus „Moore's Law“?*
  - $\Rightarrow$ SMP-/Mehrkern-CPUs (2 . . . 64 Proz.) sind technisch attraktiv
- Grafikprozessoren (GPUs)
  - neben 3D-Grafik zunehmender Computing-Einsatz (OpenCL)
  - Anwendungen: Numerik, Simulation, „Machine Learning“ . . .
  - hohe Fließkomma-Rechenleistung

SMP: *Symmetric Multiprocessing*

- mehrere (gleichartige) Prozessoren
- gemeinsamer Hauptspeicher und I/O-Einheiten
- Zugriff über Verbindungsnetzwerk oder Bus
- geringer Kommunikationsoverhead

\+ Bus-basierte Systeme sind sehr kostengünstig

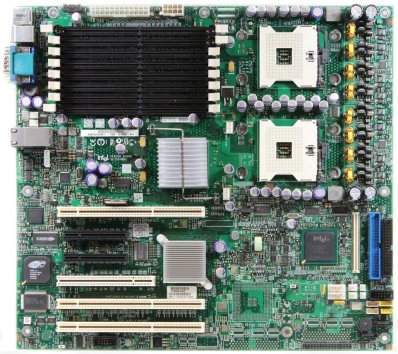
– aber schlecht skalierbar: Bus als Flaschenhals!

– Konsistenz der Daten
  - lokale Caches für gute Performanz notwendig
  - Hauptspeicher und Cache(s): Cache-Kohärenz MESI-Protokoll und „*Snooping*“

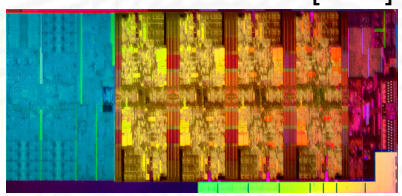
→ siehe *14.3 Speicherhierarchie – Cache Speicher*
  - Registerinhalte: ? problematisch

– Prozesse wechseln CPUs: „*Hopping*“

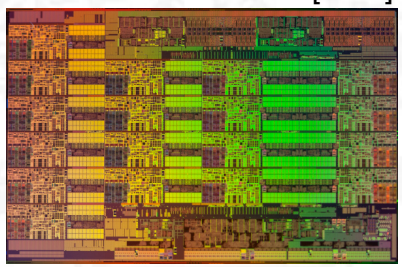
- Multi-Core Prozessoren sind „SMP on-a-chip“



8-Kern Core-i9 9xxx [Intel]



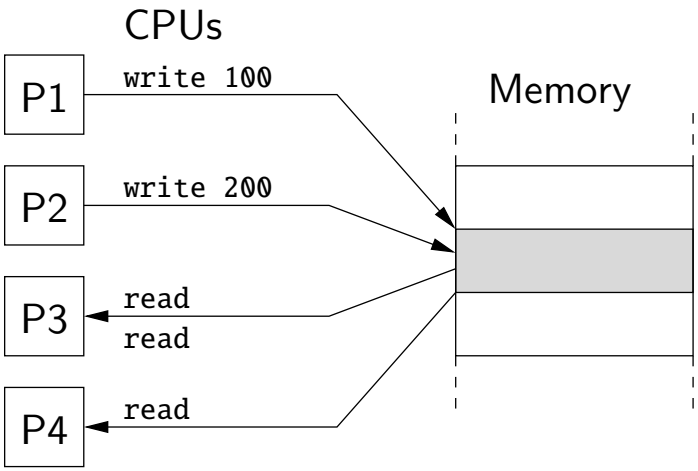
18-Kern Xeon E7 v3 [Intel]



SMP: Eigenschaften

Symmetric Multiprocessing

- alle CPUs gleichrangig, Zugriff auf Speicher und I/O
- Konsistenz: *Gleichzeitiger Zugriff auf eine Speicheradresse?*



$W1 \rightarrow 100$	$W1 \rightarrow 100$	$W2 \rightarrow 200$
$W2 \rightarrow 200$	$R3 \leftarrow 100$	$R4 \leftarrow 200$
$R3 \leftarrow 200$	$W2 \rightarrow 200$	$R3 \leftarrow 200$
$R3 \leftarrow 200$	$R3 \leftarrow 200$	$W1 \rightarrow 100$
$R4 \leftarrow 200$	$R4 \leftarrow 200$	$R3 \leftarrow 100$

$\Rightarrow$ „*Locking*“ Mechanismen und Mutexe

- spez. Befehle, atomare Operationen, Semaphore etc.
- explizit im Code zu programmieren

$\rightarrow$ siehe *15.3 Betriebssysteme – Synchronisation und Kommunikation*

SMP: Cache-Kohärenz

Cache für schnelle Prozessoren notwendig

- jede CPU hat eigene Cache (L1, L2 ...)
- aber gemeinsamer Hauptspeicher

Problem der *Cache-Kohärenz*

- Prozessor P_2 greift auf Daten zu, die im Cache von P_1 liegen
  - P_2 Lesezugriff: P_1 muss seinen Wert P_2 liefern
  - P_2 Schreibzugriff: P_1 muss Wert von P_2 übernehmen oder seinen Cache ungültig machen
  - *Was ist mit gleichzeitigen Zugriffen von P_1, P_2?*
- diverse Protokolle zur Cache-Kohärenz
  - z.B. MESI-Protokoll mit „*Snooping*“ *Modified*, *Exclusive*, *Shared*, *Invalid*
  - Caches enthalten Wert, Tag und 2 bit MESI-Zustand

→ siehe *14.3 Speicherhierarchie – Cache Speicher*, ab Folie 1100

SMP: `volatile`

- MESI-Verfahren garantiert Cache-Kohärenz für Werte im Cache und im Hauptspeicher

Vorsicht: Was ist mit den Registern?

- Variablen in Registern werden von MESI nicht erkannt
- Compiler versucht, häufig benutzte Variablen soweit wie möglich in Registern zu halten
- globale/*shared*-Variablen niemals in Registern halten
- Java, C: Deklaration als `volatile`

SMP: Erreichbarer Speedup (bis 32 Threads)

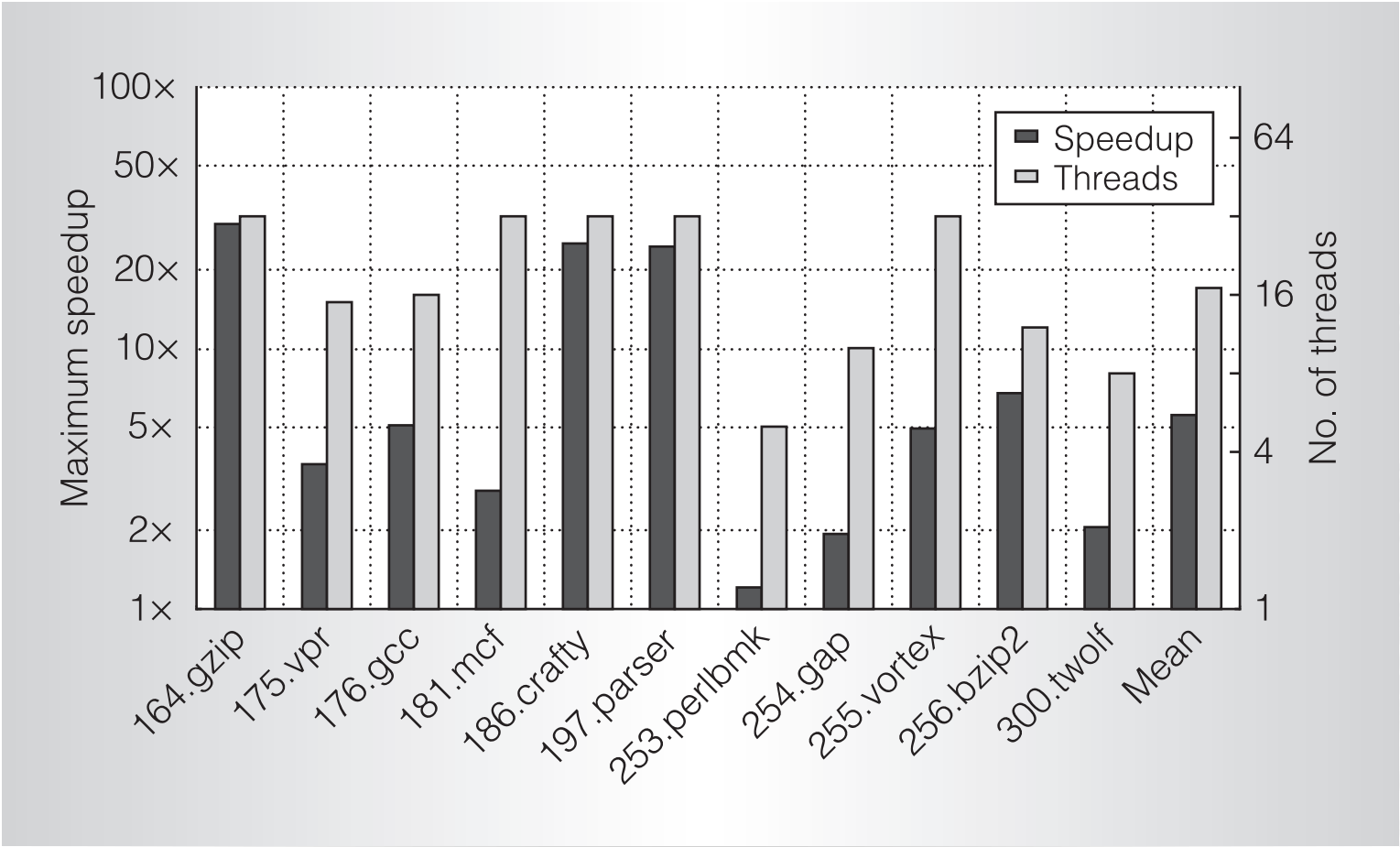
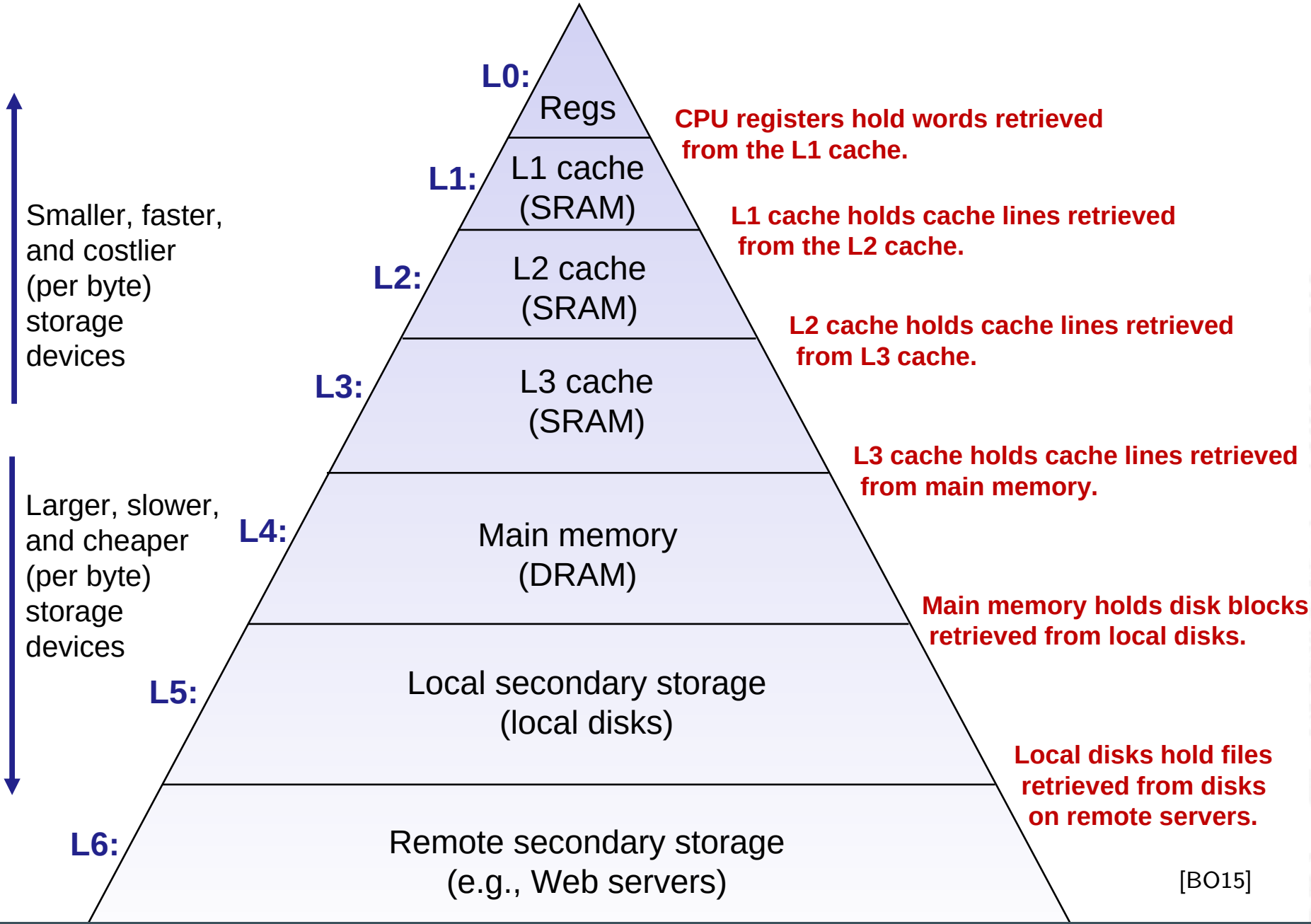


Figure 4. Maximum speedup achieved on up to 32 threads over single-threaded execution (black bars) and minimum number of threads at which the maximum speedup occurred (gray bars).

M.J .Bridges et.al.: *Revisiting the Sequential Programming Model for the Multicore Era*, IEEE Micro 2008 [Br+08]

Speicherhierarchie



[BO15]

Speicherhierarchie (cont.)

Gesamtsystem kombiniert verschiedene Speicher

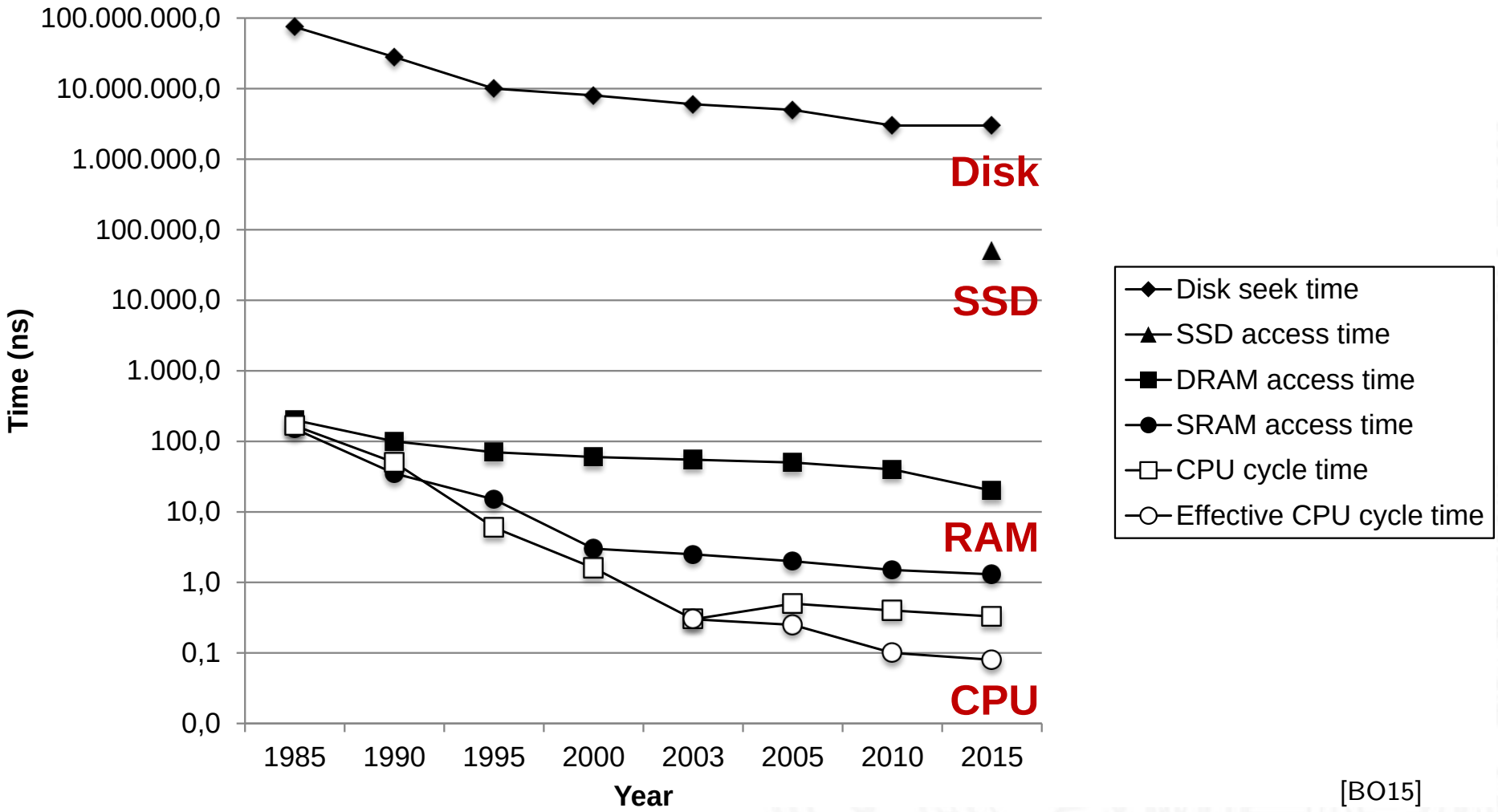
- wenige KByte Register (-bank) im Prozessor
- einige MByte SRAM als schneller Zwischenspeicher
- einige GByte DRAM als Hauptspeicher
- einige TByte Festplatte als nichtflüchtiger Speicher
- Hintergrundspeicher (CD/DVD/BR, Magnetbänder)
- das WWW und Cloud-Services

Kompromiss aus Kosten, Kapazität, Zugriffszeit

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

Prozessor-Memory Gap

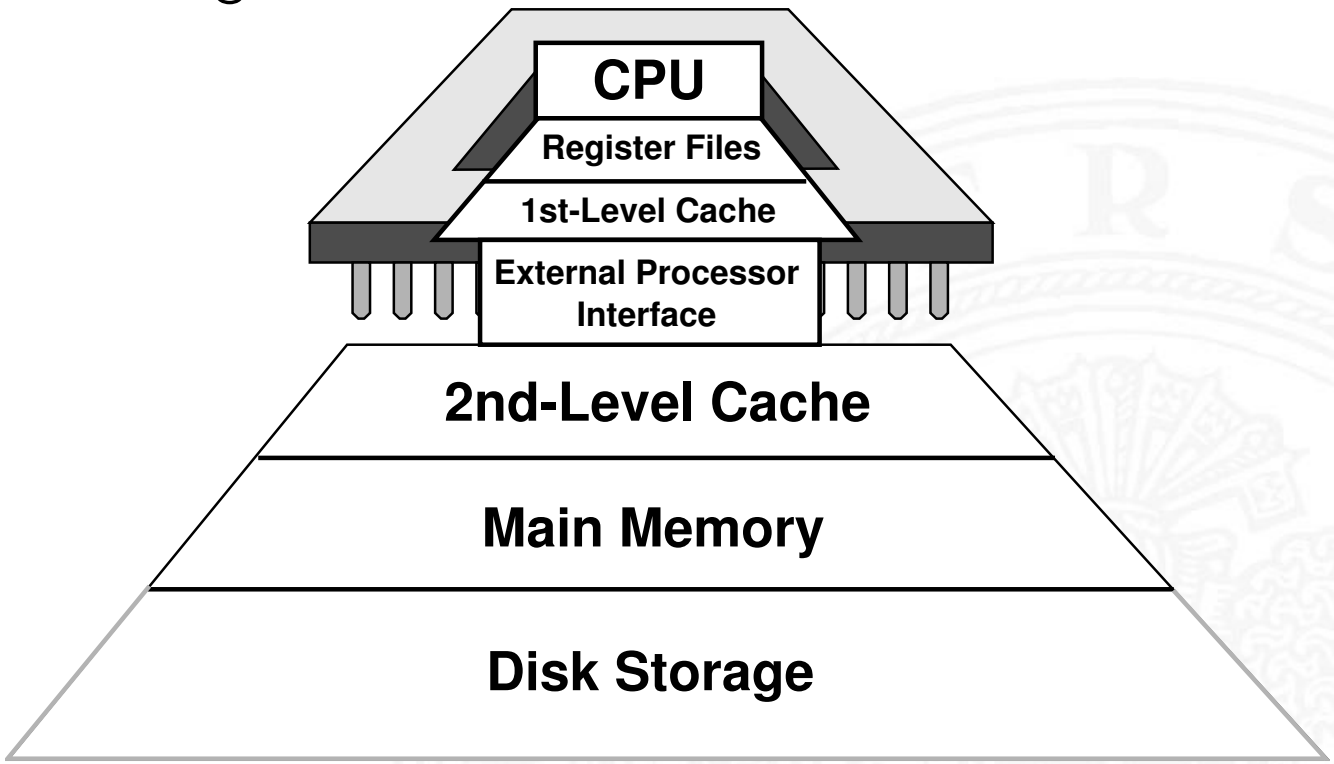
- stetig wachsende Lücke zwischen CPU-, Memory- und Disk-Geschwindigkeiten



[BO15]

Motivation

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



Motivation (cont.)

- schnelle vs. langsame Speichertechnologie

  schnell : hohe Kosten/Byte geringe Kapazität

  langsam : geringe –"– hohe –"–

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

⇒ Motivation für spezielle Organisation von Speichersystemen

Speicherhierarchie

Verwaltung der Speicherhierarchie

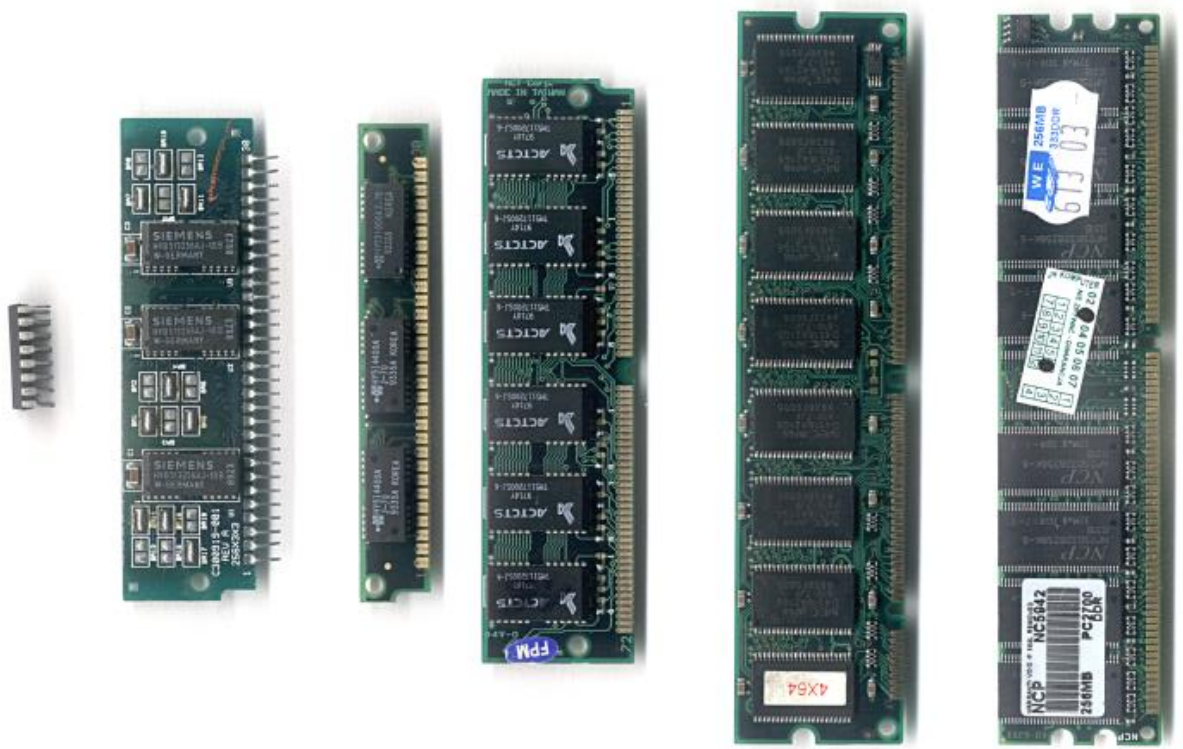
- Register $\leftrightarrow$ Memory
  - Compiler
  - Assembler-Programmierer
- Cache $\leftrightarrow$ Memory
  - Hardware
- Memory $\leftrightarrow$ Disk
  - Hardware und Betriebssystem: Paging, Segmentation
  - Programmierer und –"– : Dateien

L0: Register

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

L1-L4: Halbleiterspeicher RAM

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



L5: Festplatten

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

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

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

L6: Hintergrundspeicher

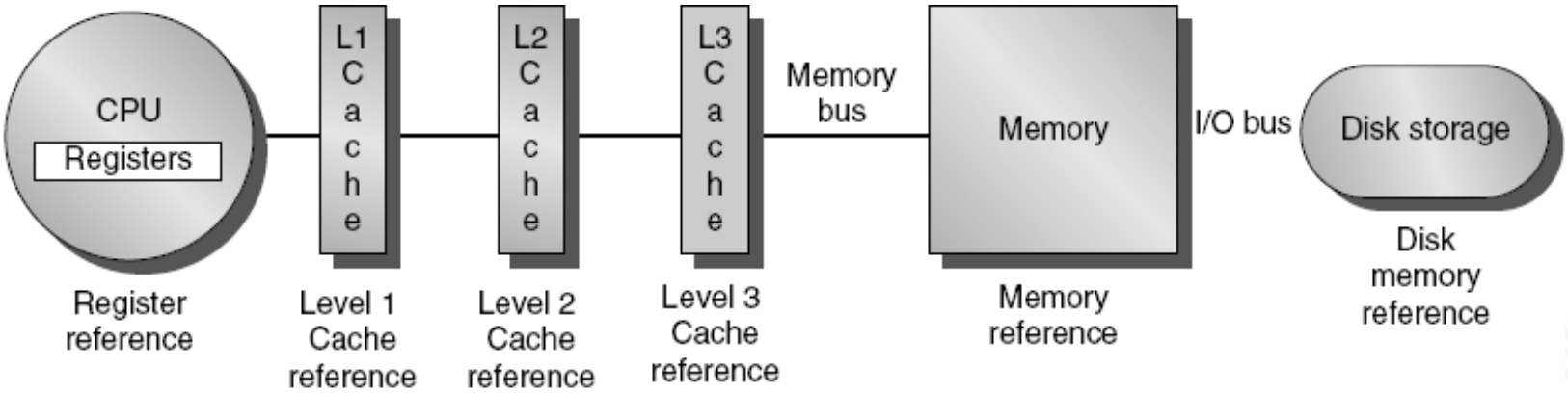
- enorme Speicherkapazität
- langsame Zugriffszeiten

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

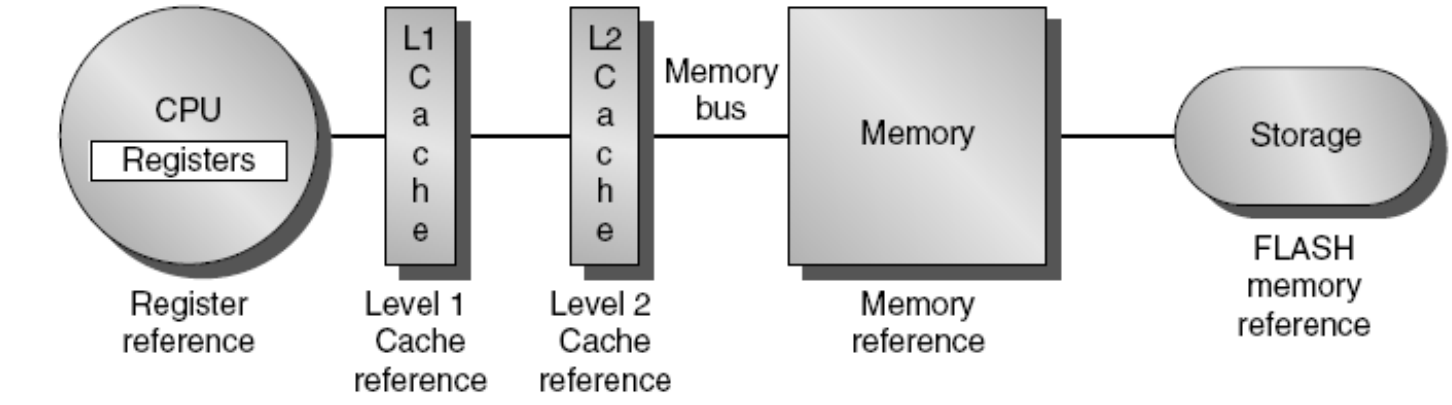
- WWW und Internet-Services, Cloud-Services
  - Cloud-Farms ggf. ähnlich schnell wie L5 Festplatten, da Netzwerk schneller als der Zugriff auf eine lokale Festplatte

- in dieser Vorlesung nicht behandelt

Speicherhierarchie: zwei Beispiele



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

[HP17]

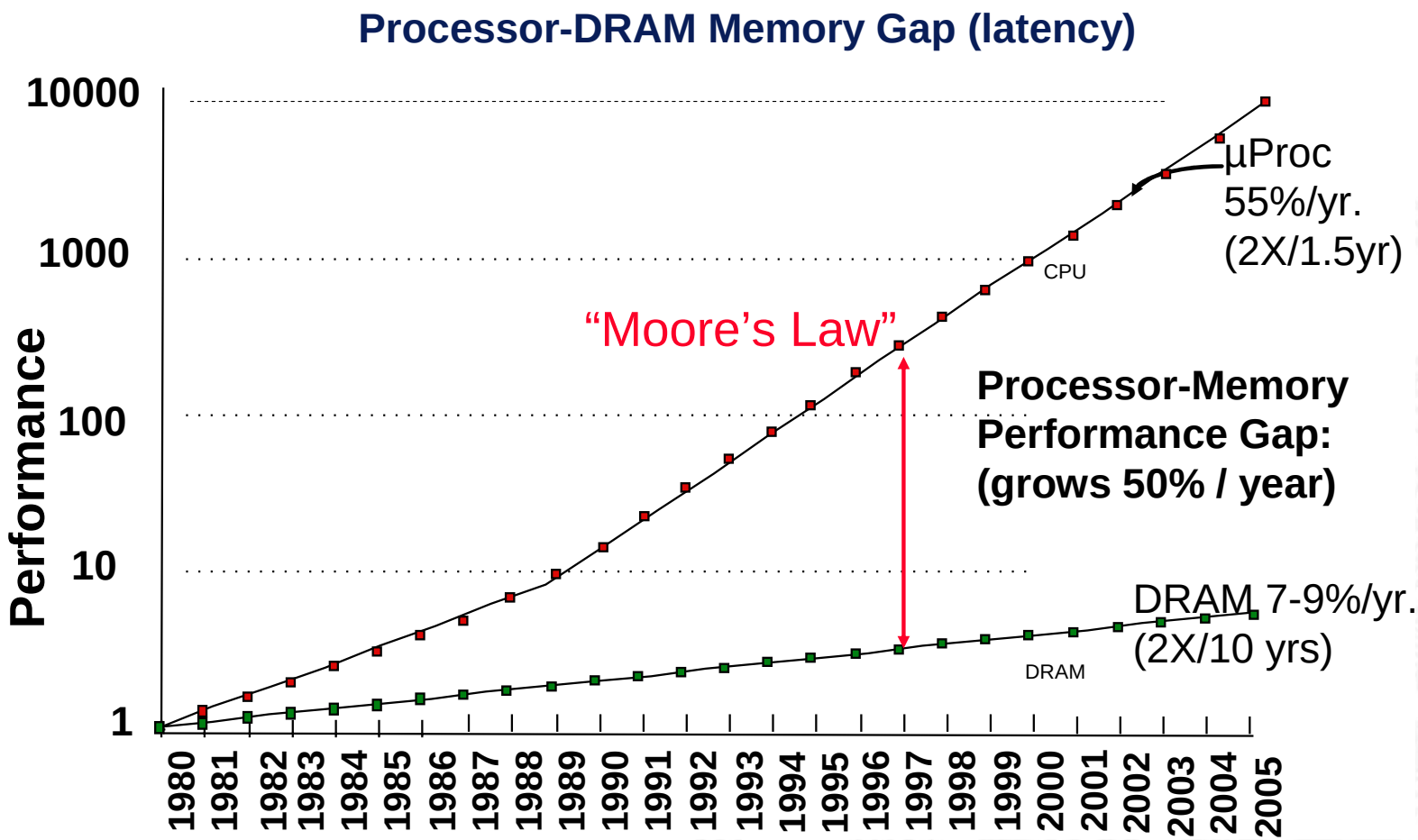
Eigenschaften der Speichertypen

Speicher	Vorteile	Nachteile
Register	sehr schnell	sehr teuer
SRAM	schnell	teuer, große Chips
DRAM	hohe Integration	Refresh nötig, langsam
Platten	billig, Kapazität	sehr langsam, mechanisch

Beispiel	Hauptspeicher	Festplatte	SSD
Latenz	8 ns	4 ms	0,2/0,4 ms
Bandbreite	25,6 GB/sec (pro Kanal, bis 4)	1,5 GB/sec	3/2 GB/sec (r/w)
Kosten/GB	6 €	2,5 ct. 1 TB: 25 €	25 ct.

Cache

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



[PH16b]

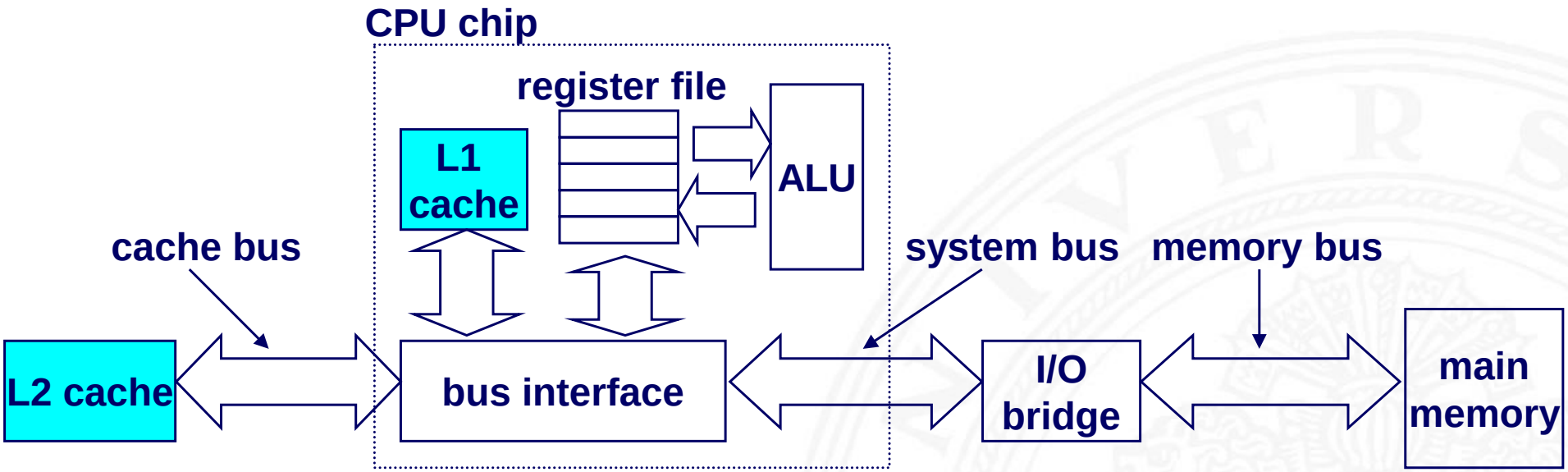
⇒ Cache als schneller Zwischenspeicher zum Hauptspeicher

Cache (cont.)

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

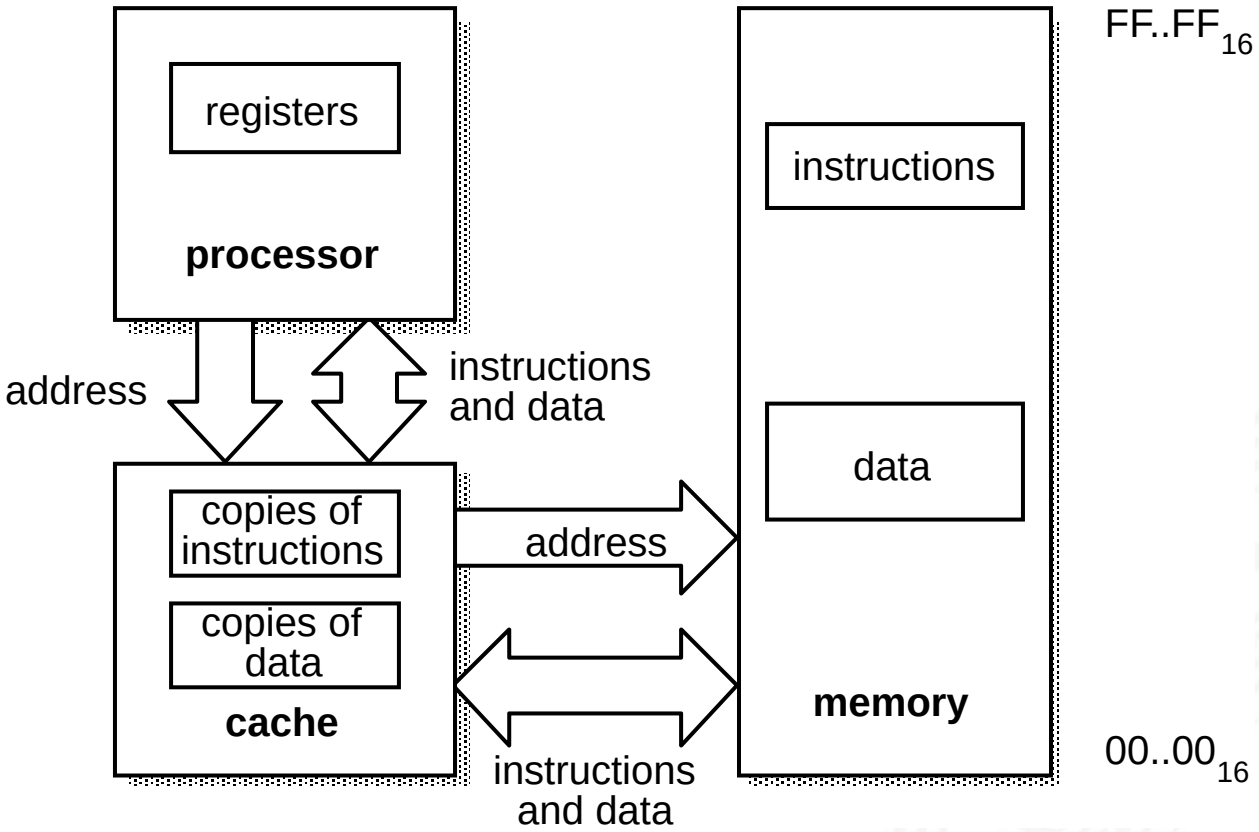
Cache (cont.)

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



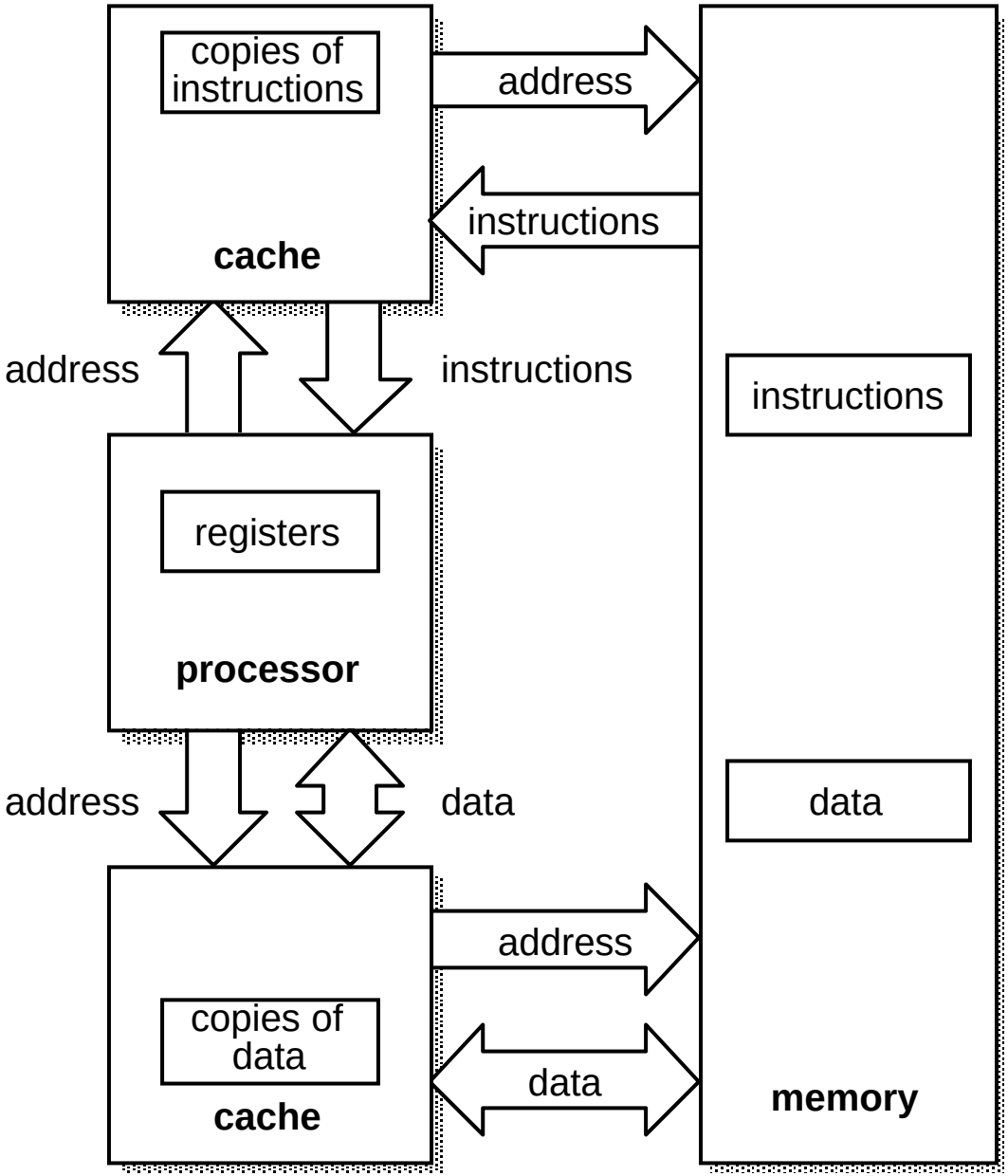
[BO15]

gemeinsamer Cache / „unified Cache“



[Fur00]

separate Instruction-/Data Caches



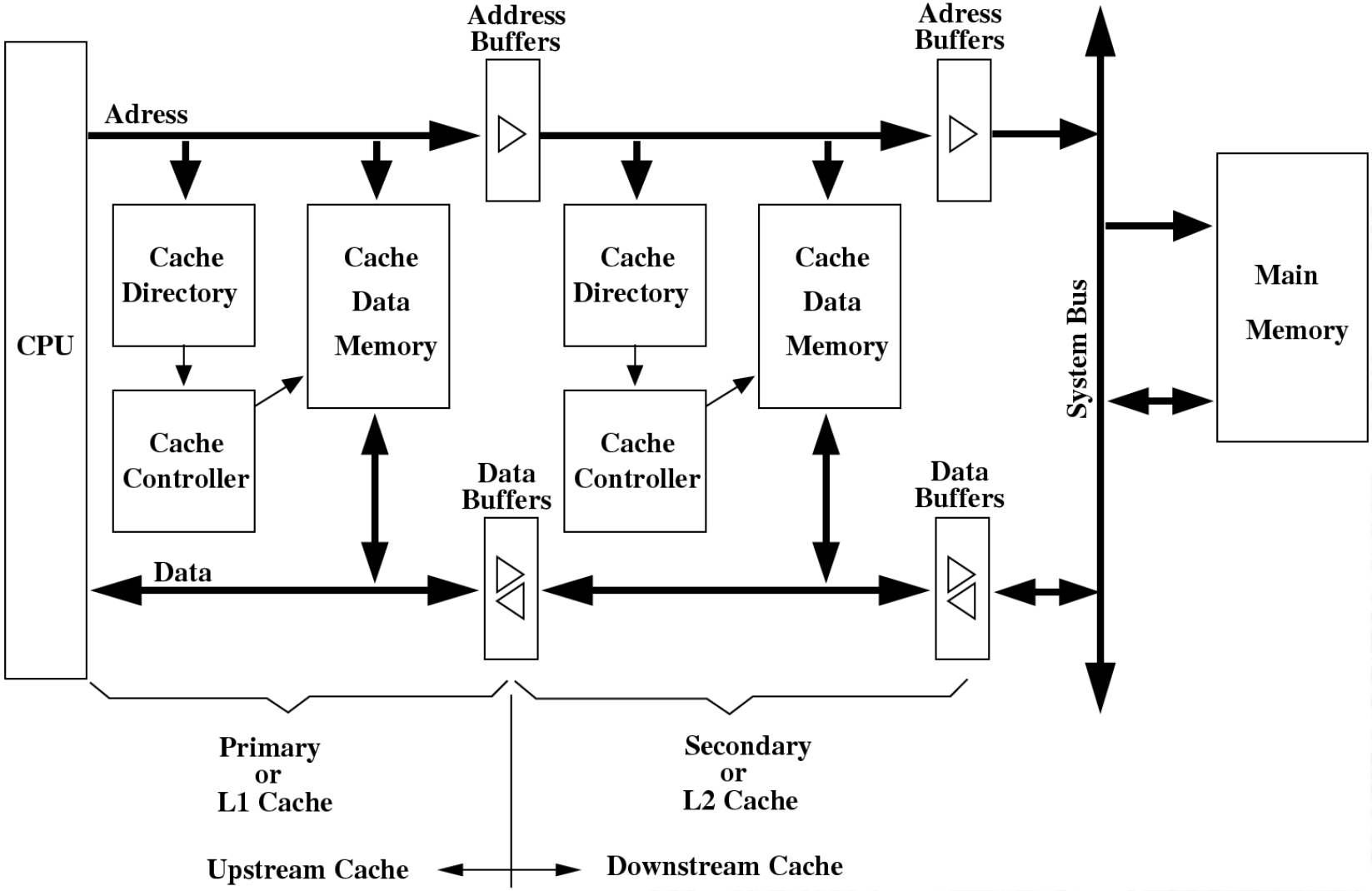
$FF..FF_{16}$

$00..00_{16}$

[Fur00]

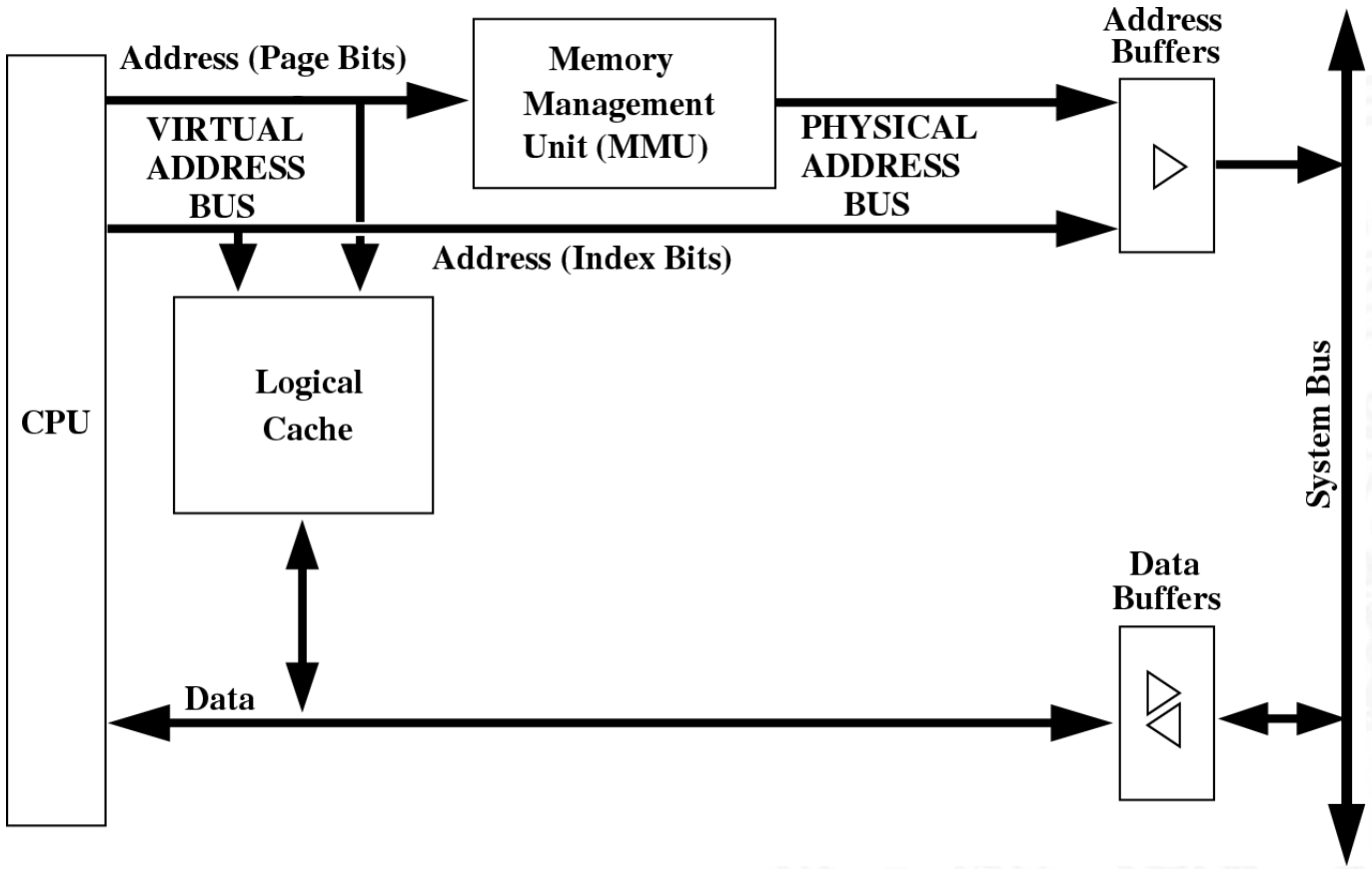
Cache – Position

- First- und Second-Level Cache



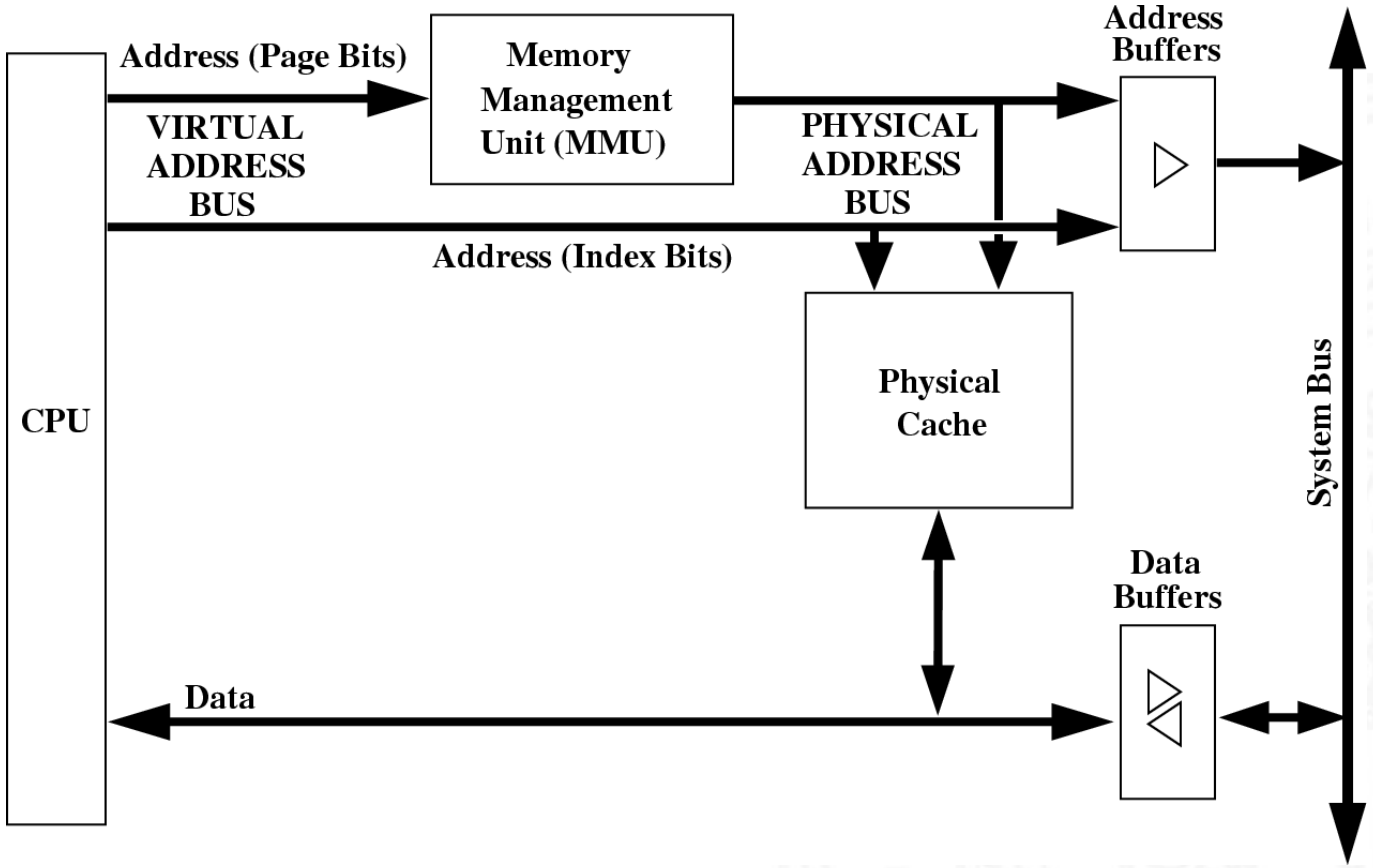
Cache – Position (cont.)

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



Cache – Position (cont.)

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



Cache – Position (cont.)

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

  $\Rightarrow$ Cache-Kohärenz wichtig
  - gemeinsam genutzte Daten konsistent halten (s.u.)

Cache – Strategie

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

- *temporale Lokalität*:
  die Daten, die zuletzt häufig gebraucht wurden
- *räumliche Lokalität*:
  die Daten, die nahe den zuletzt gebrauchten liegen

- verschiedene Platzierungs-, Ersetzungs- und Rückschreibestategien für den Cache

Cache – Performanz

Cacheperformanz

- Begriffe

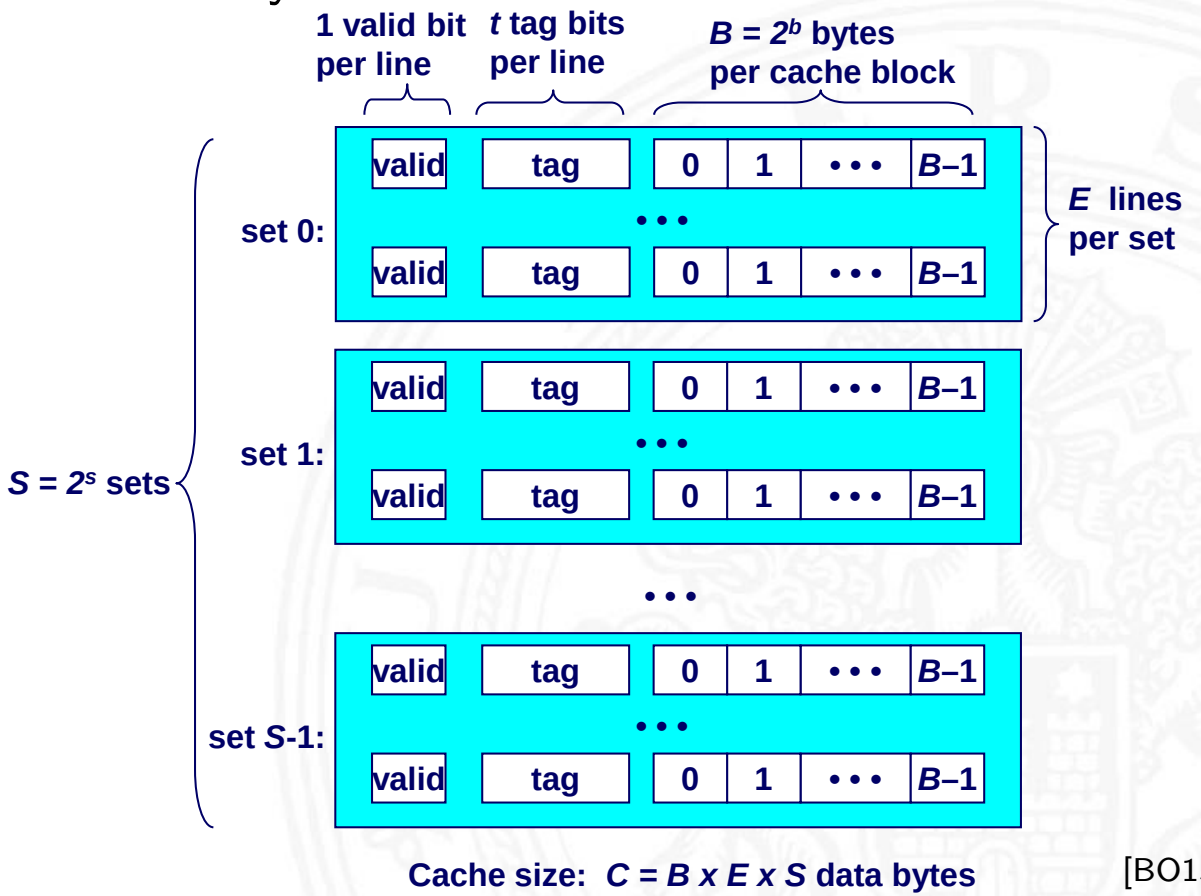
Treffer (Hit)		Zugriff auf Datum, ist bereits im Cache
Fehler (Miss)		–"– ist nicht –"–
Treffer-Rate	R_{Hit}	Wahrscheinlichkeit, Datum ist im Cache
Fehler-Rate	R_{Miss}	$1 - R_{Hit}$
Hit-Time	T_{Hit}	Zeit, bis Datum bei Treffer geliefert wird
Miss-Penalty	T_{Miss}	zusätzlich benötigte Zeit bei Fehler

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

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

Cache Organisation

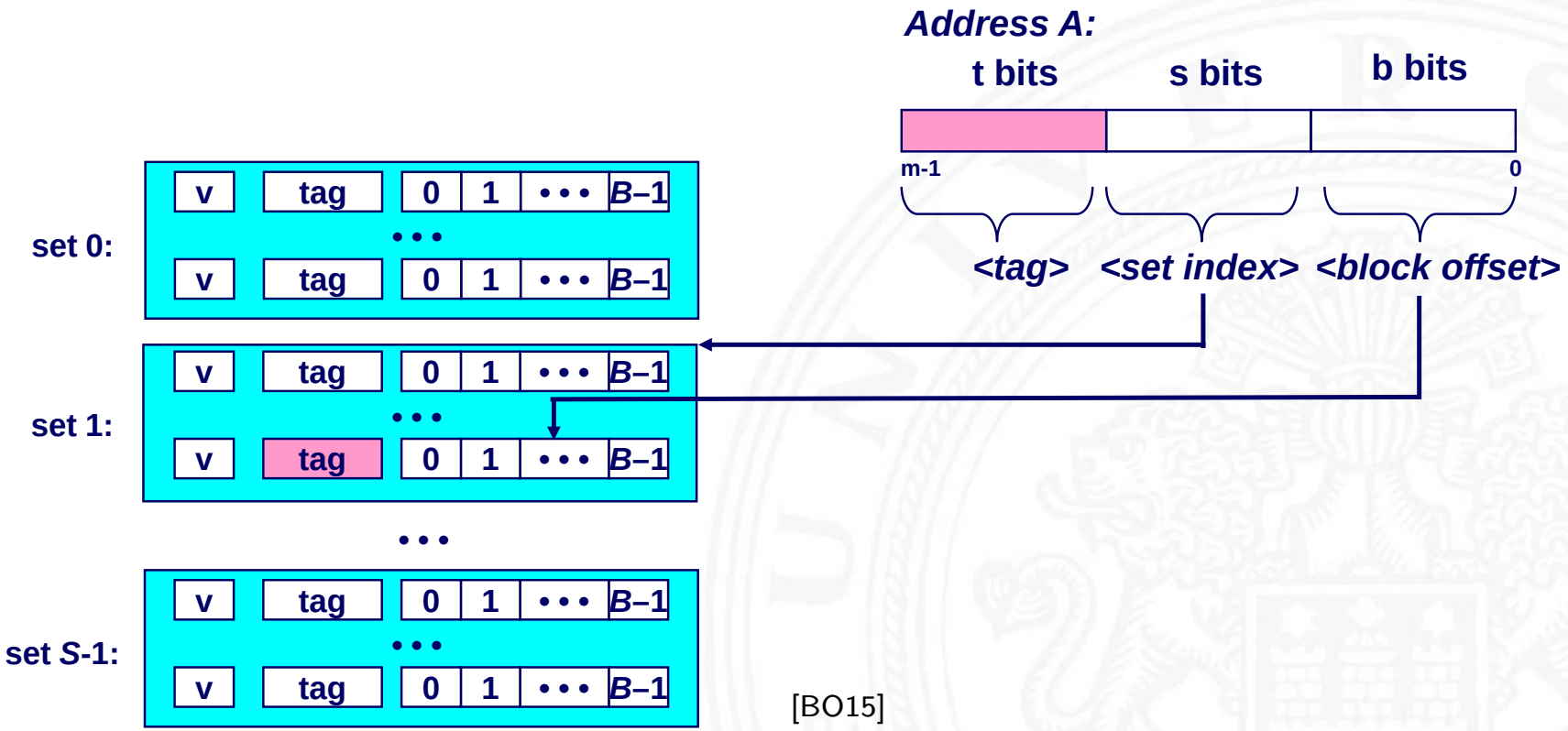
- Cache ist ein Array von Speicher-Bereichen („sets“)
- jeder Bereich enthält eine oder mehrere Zeilen
- jede Zeile enthält einen Datenblock
- jeder Block enthält mehrere Byte



[BO15]

Adressierung von Caches

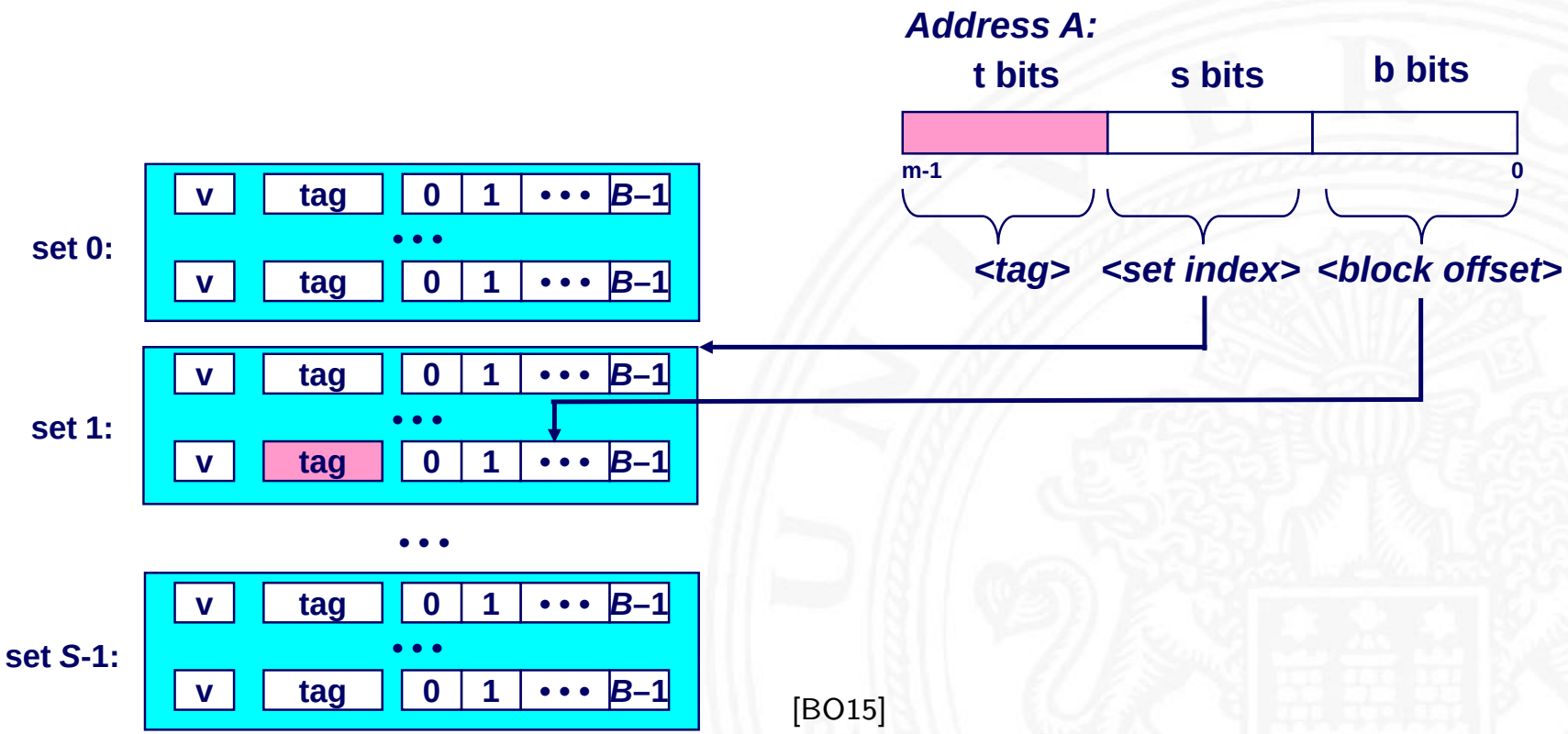
- Adressteil $\langle set\ index \rangle$ von A bestimmt Bereich („set“)
- Adresse A ist im Cache, wenn
  1. Cache-Zeile ist als gültig markiert („valid“)
  2. Adressteil $\langle tag \rangle$ von A = „tag“ Bits des Bereichs



[BO15]

Adressierung von Caches (cont.)

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



[BO15]

Cache – Organisation

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

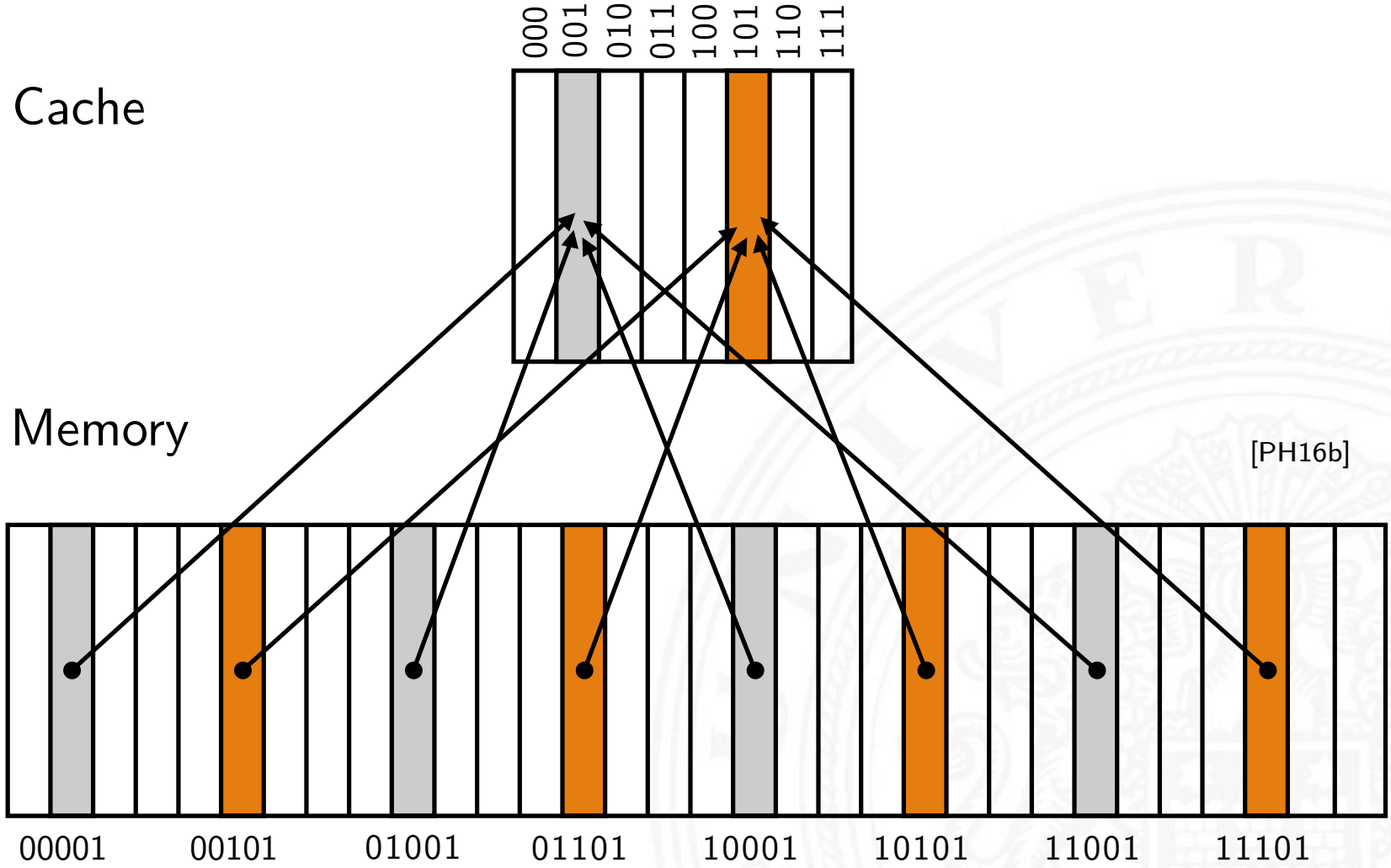
direkt abgebildet / direct mapped jeder Speicheradresse ist genau eine Cache-Speicherzelle zugeordnet

n-fach bereichsassoziativ / set associative
jeder Speicheradresse ist eine von n möglichen Cache-Speicherzellen zugeordnet (E in [BO15])

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

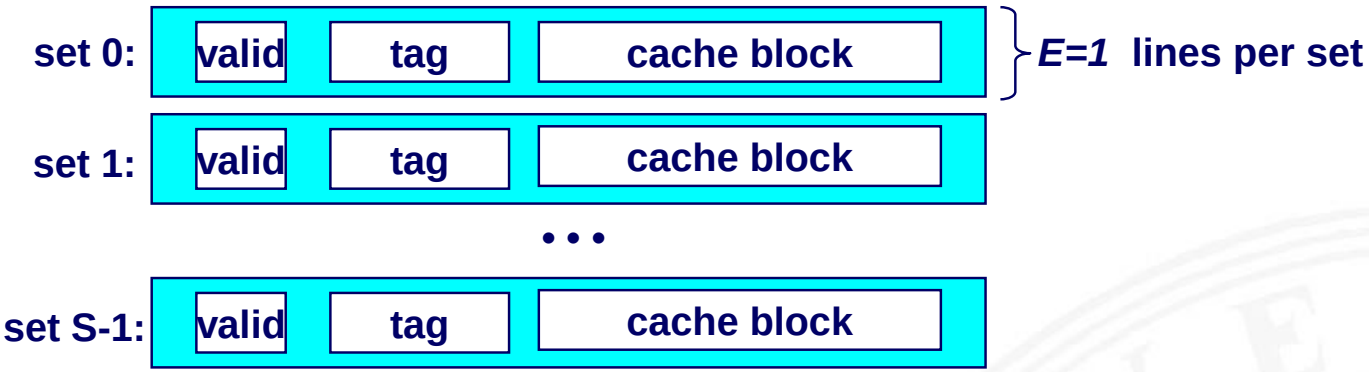
Cache: direkt abgebildet / „direct mapped“

- jeder Adresse ist genau eine Speicherzelle im Cache zugeordnet



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

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



[BO15]

\+ einfachste Cache-Art

\+ große Caches möglich

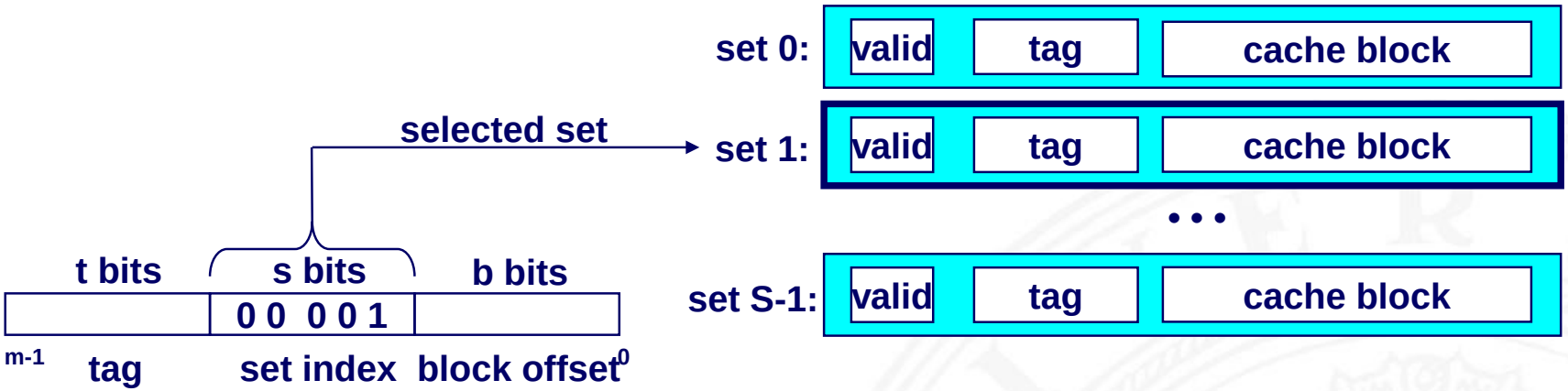
– Effizienz, z.B. Zugriffe auf A, $A + n \cdot S \ldots$
$\Rightarrow$ „*Cache Thrashing*“

Beispiel (s.o.): Zugriff auf „00101“, „01101“, „10101“, „11101“

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

Zugriff auf direkt abgebildete Caches

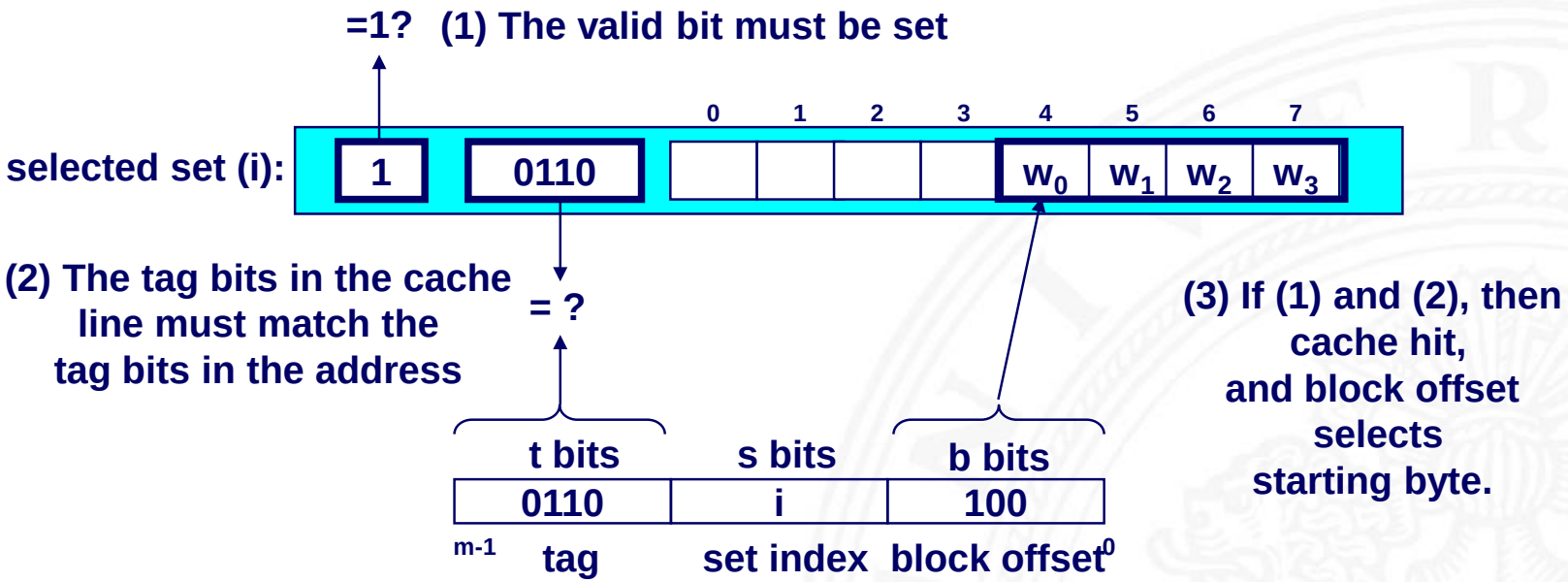
1. Bereichsauswahl durch Bits ⟨*set index*⟩



[BO15]

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

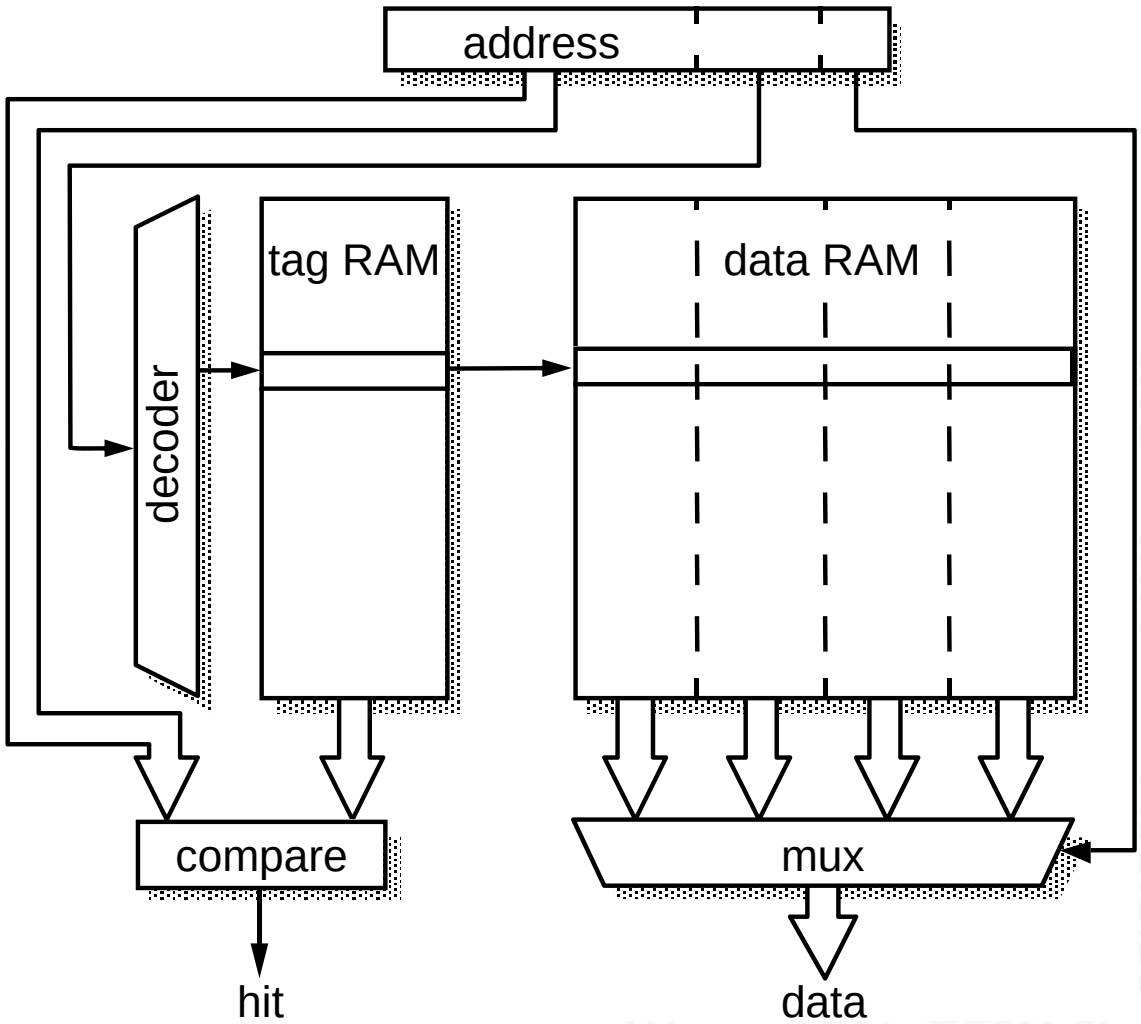
2. ⟨*valid*⟩: sind die Daten gültig?
3. „Line matching“: stimmt ⟨*tag*⟩ überein?
4. Wortselektion extrahiert Wort unter Offset ⟨*block offset*⟩



[BO15]

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

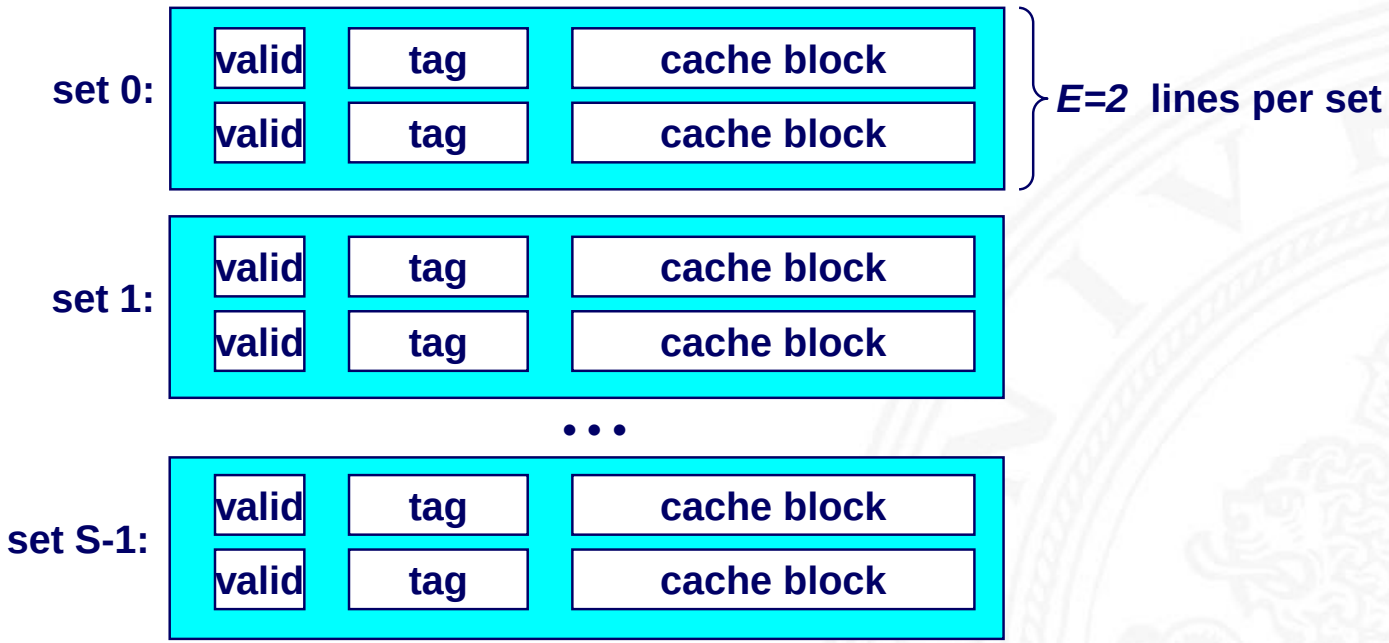
Prinzip



[Fur00]

Cache: bereichsassoziativ / „set associative“

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

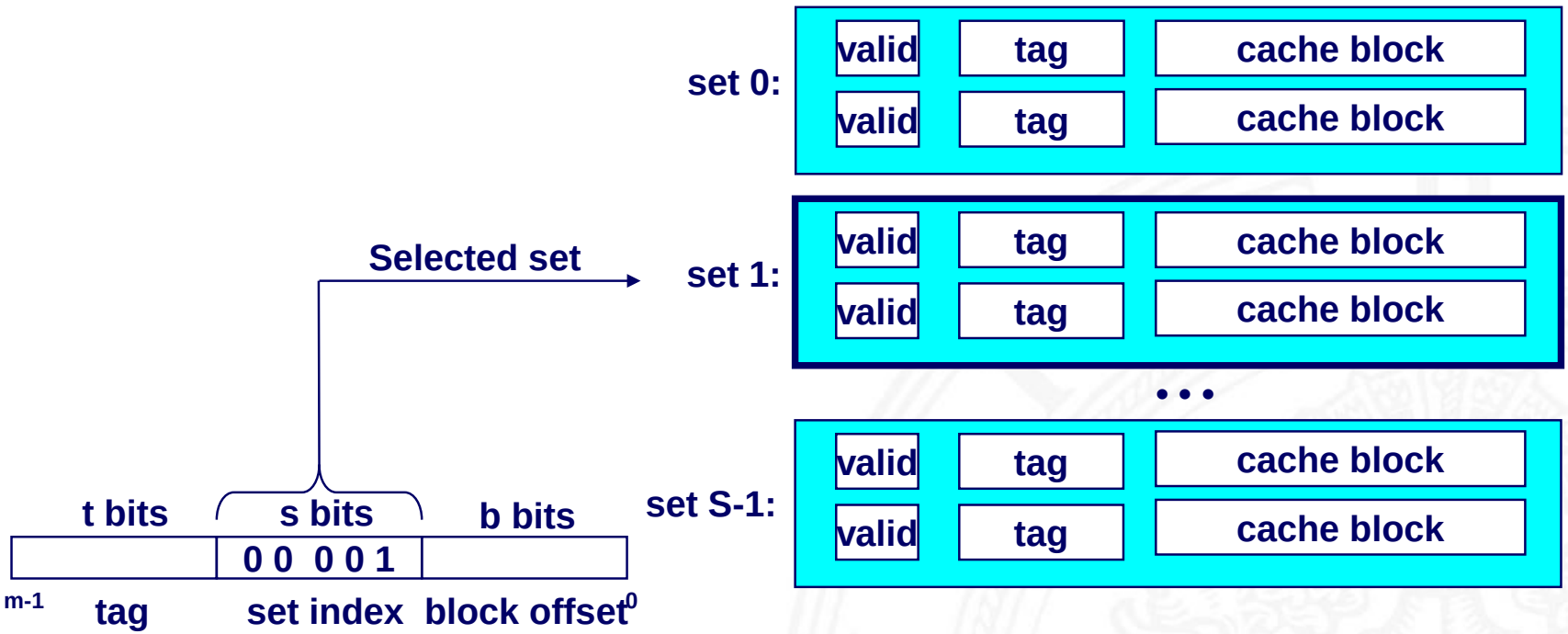


[BO15]

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

Zugriff auf n-fach assoziative Caches

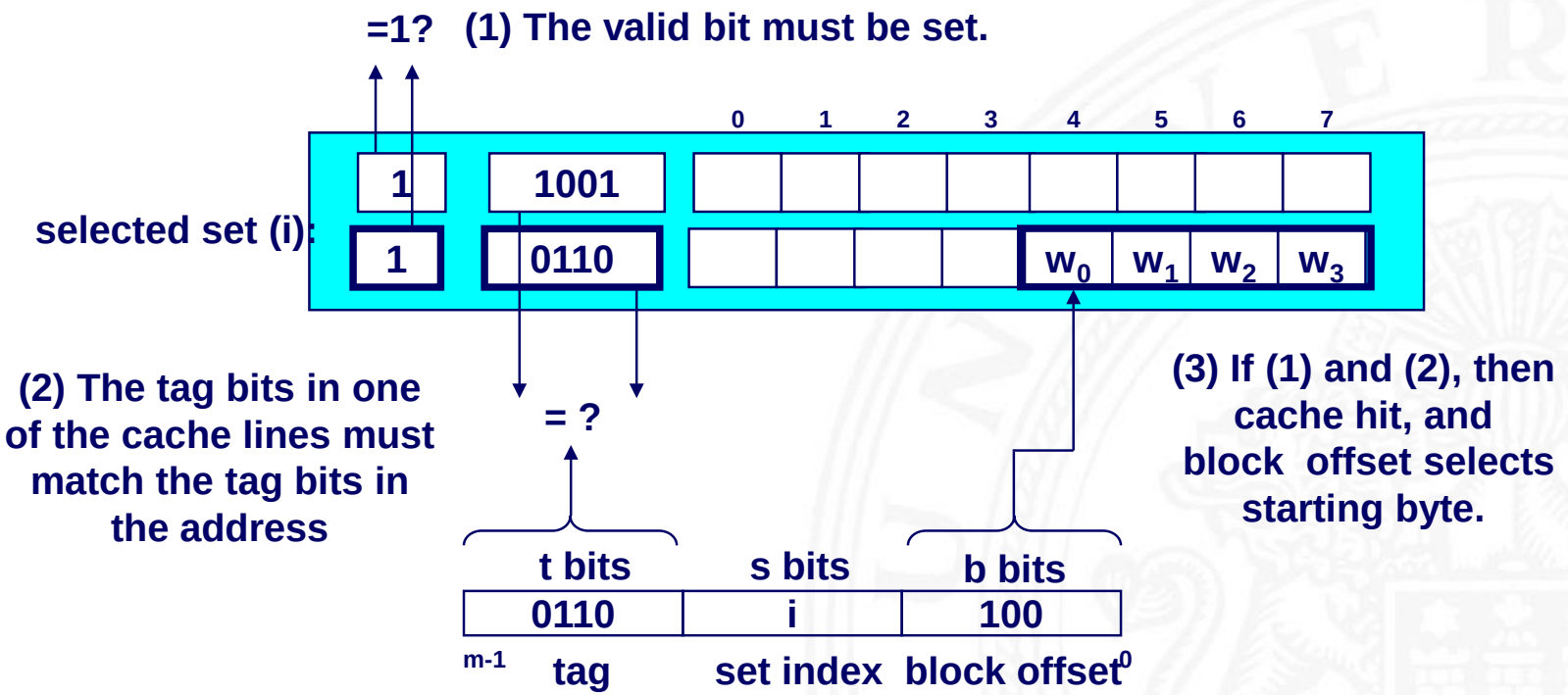
1. Bereichsauswahl durch Bits ⟨*set index*⟩



[BO15]

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

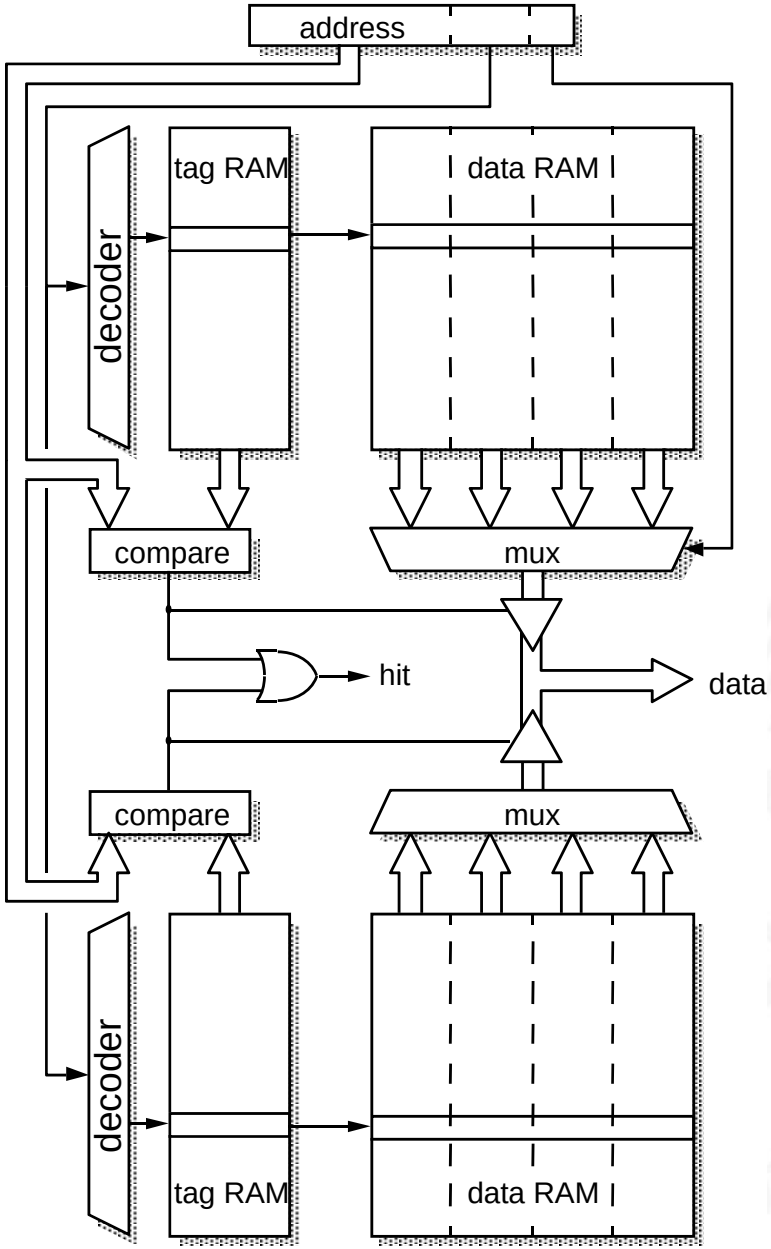
2. ⟨*valid*⟩: sind die Daten gültig?
3. „Line matching“: Cache-Zeile mit passendem ⟨*tag*⟩ finden? dazu Vergleich aller „tags“ des Bereichs ⟨*set index*⟩
4. Wortselektion extrahiert Wort unter Offset ⟨*block offset*⟩



[BO15]

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

Prinzip



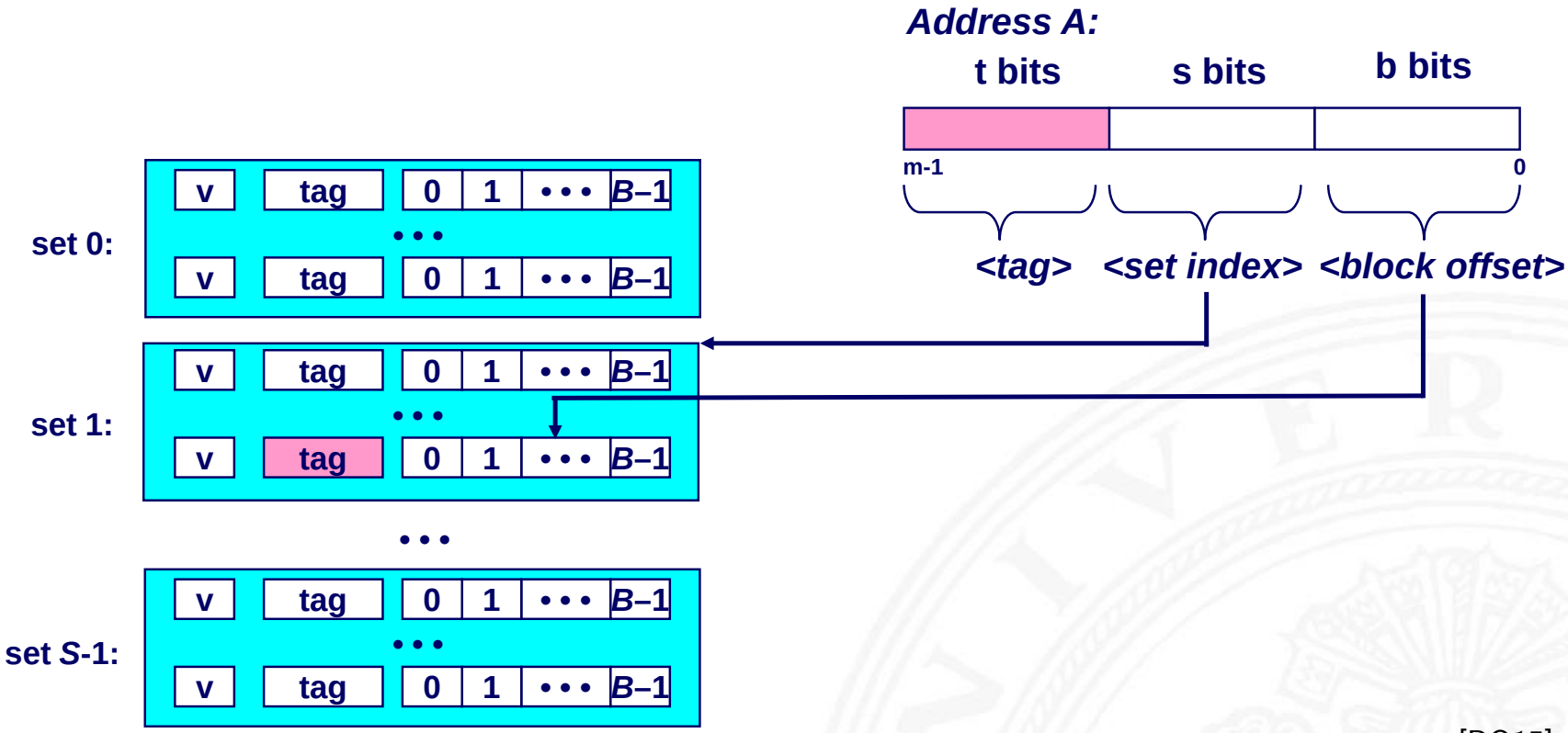
[Fur00]

Cache: voll-assoziativ

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

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

Cache – Dimensionierung



[BO15]

- Parameter: S, B, E
- Cache speichert immer größere Blöcke / „*Cache-Line*“
- Wortauswahl durch $\langle$*block offset*$\rangle$ in Adresse

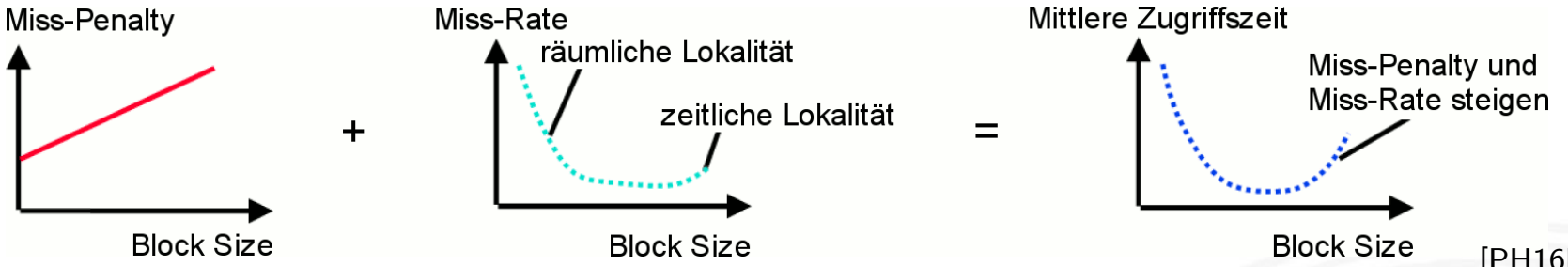
Cache – Dimensionierung (cont.)

Vor- und Nachteile des Cache

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

Cache – Dimensionierung (cont.)

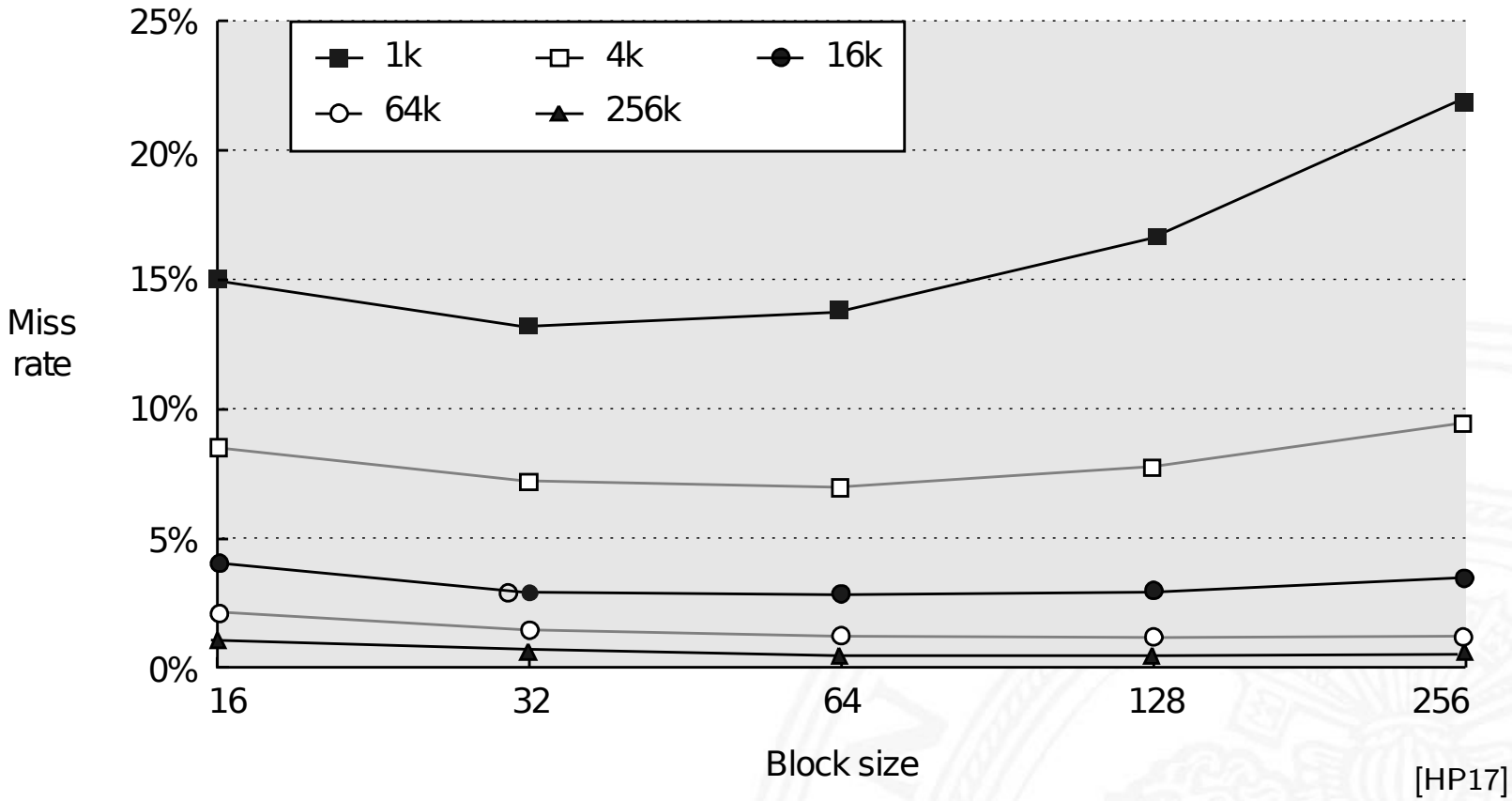
Cache- und Block-Dimensionierung



[PH16b]

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

Cache – Dimensionierung (cont.)



[HP17]

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

Cache-Misses

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

Cache Ersetzungsstrategie

Wenn der Cache gefüllt ist, welches Datum wird entfernt?

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

Cache Schreibstrategie

Wann werden modifizierte Daten des Cache zurückgeschrieben?

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

Cache-Kohärenz

- Daten zwischen Cache und Speicher konsistent halten
- notwendig wenn mehrere Einheiten (Bus-Master: Prozessor, DMA-Controller) auf Speicher zugreifen können:
  wichtig für „*Symmetric Multiprocessing*“
- Harvard-Architektur hat getrennte Daten- und Instruktions-Speicher
  - Instruktionen sind read-only

  $\Rightarrow$ einfacherer Instruktions-Cache

  $\Rightarrow$ Cache-Kohärenz Problem betrifft D-Cache
- Cache-Kohärenz Protokolle und „*Snooping*“
  - alle Prozessoren(P_1, P_2 ...) überwachen alle Bus-Transaktionen
    Cache „schnüffelt“ am Speicherbus
  - Prozessor P_2 greift auf Daten zu, die im Cache von P_1 liegen
    P_2 Schreibzugriff $\Rightarrow$ P_1 Cache aktualisieren / ungültig machen
    P_2 Lesezugriff $\Rightarrow$ P_1 Cache liefert Daten
  - *Was ist mit gleichzeitige Zugriffen von P_1, P_2?*

Cache-Kohärenz (cont.)

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

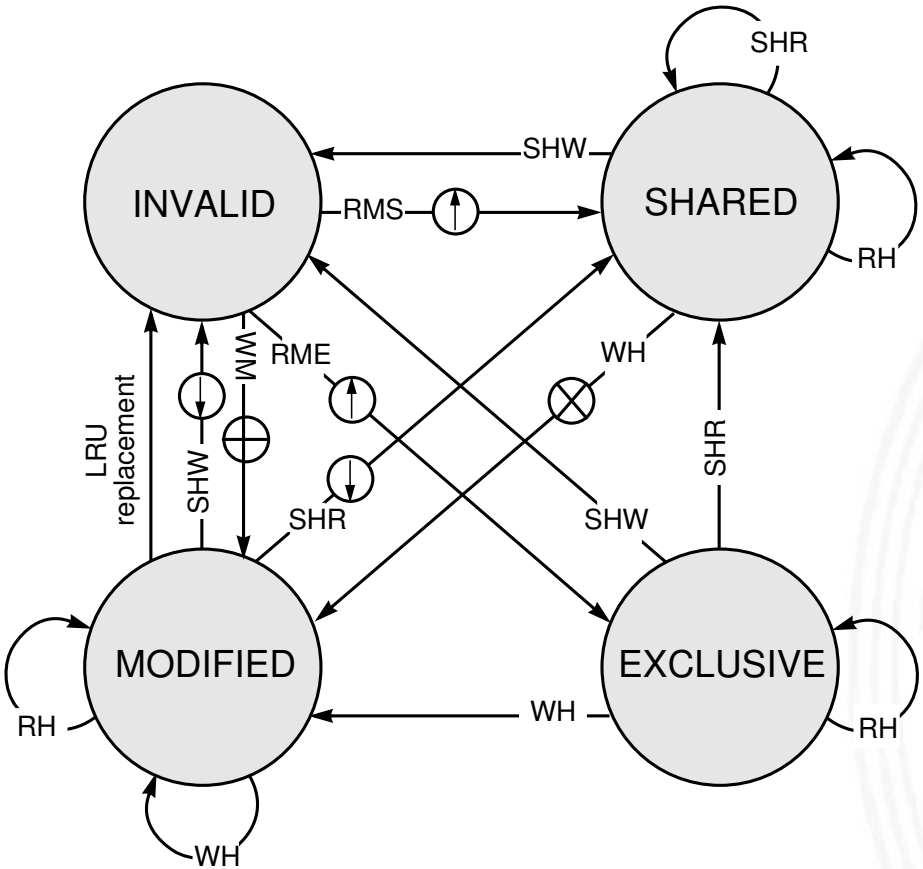
siehe z.B.: `en.wikipedia.org/wiki/Cache_coherence`

MESI Protokoll

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

MESI Protokoll (cont.)

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



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

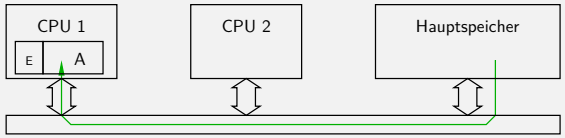
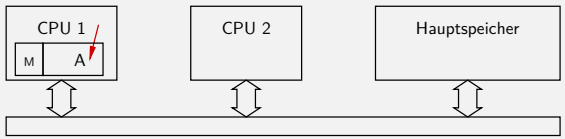
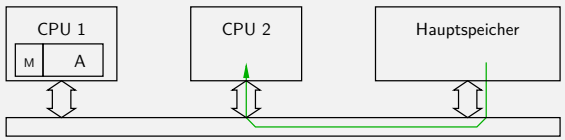
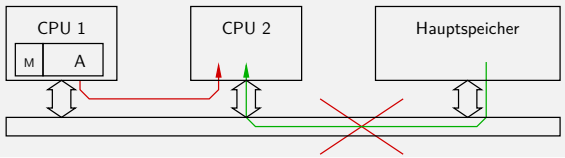
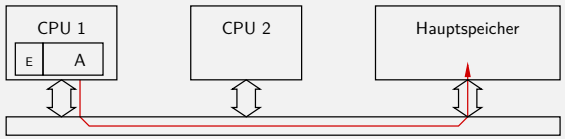
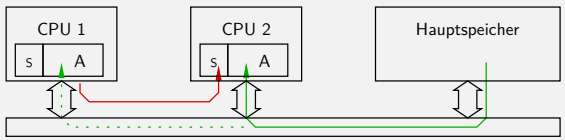
RH = Read hit
RMS = Read miss, shared
RME = Read miss, exclusive
WH = Write hit
WM = Write miss
SHR = Snoop hit on a read
SHW = Snoop hit on a write or read-with-intent-to-modify

Bus Transactions

= Snoop push
= Invalidate transaction
= Read-with-intent-to-modify
= Read

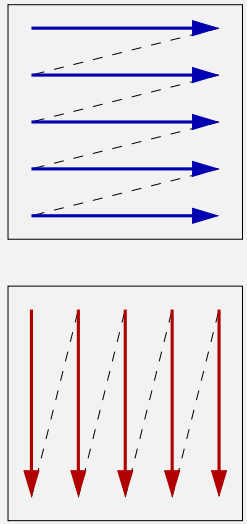
MESI Protokoll (cont.)

- „*Snooping*“ Beispiel

	MESI-Status Wert A:	CPU1	CPU2
	CPU 1 lädt Wert A	E	I
	CPU 1 modifiziert A	M	I
	CPU 2 lädt A (aber Wert modified)	- -	- -
	CPU 1 SNOOP! (CPU2 read gestoppt)	- -	- -
	CPU 1 schreibt A (CPU2 read gestoppt)	E	I
	CPU 2 lädt A (CPU2 read restart, A shared)	S	S

Cache Effekte bei Matrixzugriffen


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



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

Cache vs. Programmcode

Programmierer kann für maximale Cacheleistung optimieren

- ▷ Datenstrukturen werden fortlaufend alloziert

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

Systeme bevorzugen einen *Cache-freundlichen* Code

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

Literatur

[PH17] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – RISC-V Edition*.
Morgan Kaufmann Publishers Inc., 2017.
ISBN 978–0–12–812275–4

[PH16b] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle*.
5. Auflage, Oldenbourg, 2016. ISBN 978–3–11–044605–0

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

Literatur (cont.)

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

[TA14] A.S. Tanenbaum, T. Austin: *Rechnerarchitektur – Von der digitalen Logik zum Parallelrechner*.
6. Auflage, Pearson Deutschland GmbH, 2014.
ISBN 978–3–8689–4238–5

[Tan06] A.S. Tanenbaum:
Computerarchitektur – Strukturen, Konzepte, Grundlagen.
5. Auflage, Pearson Studium, 2006. ISBN 3–8273–7151–1

Literatur (cont.)

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

[Br+08] M.J. Bridges [u. a.]: *Revisiting the Sequential Programming Model for the Multicore Era*.
in: *IEEE Micro* 1 Vol. 28 (2008), S. 12–20.

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

Gliederung

Gliederung (cont.)

Betriebssysteme in RSB

- genug Stoff für eigene Vorlesungen
- Themen
  - Prozesse und Threads
  - Synchronisation und Kommunikation; Deadlocks
  - Scheduling
  - Speicherverwaltung; *Virtual Memory*
  - Dateiverwaltung und I/O
- nicht behandelt
  - Praxisbeispiele: Windows, Unix, Linux, Android ...
  - Dateisysteme
  - Virtualisierung; Container

  → *VSS (Verteilte Systeme und Systemsicherheit)*
  Sicherheit, RAID

  → *ES (Eingebettete Systeme)*
  Eingebettete Betriebssysteme, Echtzeitverhalten

→ Grafiken, wenn nicht anders angegeben, aus: W. Stallings: *Operating Systems – Internals and Design Principles* [Sta17]

Betriebssysteme

Was sind Betriebssysteme?

Im Prinzip Software, wie jedes andere Programm auch!

Was machen Betriebssysteme?

- Verwalten der „teuren“ Hardware für optimale Nutzung
  - Prozessor(en)
  - Systembus(se)
  - Hauptspeicher
  - Festplatten / SSDs
  - Ein-/Ausgabeeinheiten (I/O)

  $\Rightarrow$ Anpassen der Geschwindigkeiten
- Koordination aller Programme, Dienste und Benutzer
  Wer darf wann worauf zugreifen?
- Bereitstellen von Systemdiensten („*Service*“) und Schnittstellen („*System-Call*“) für (andere) Programme, bzw. die Benutzer
  Wie ist der Zugriff geregelt?

$\Rightarrow$ BS sind meist die komplexeste Software auf dem Computer!

1. erste Computer / serielle Verarbeitung

- kein Betriebssystem
- Programmierer arbeitet direkt mit Computer an Konsole
- Benutzer können nur nacheinander am Computer arbeiten

– Reservierung des Systems

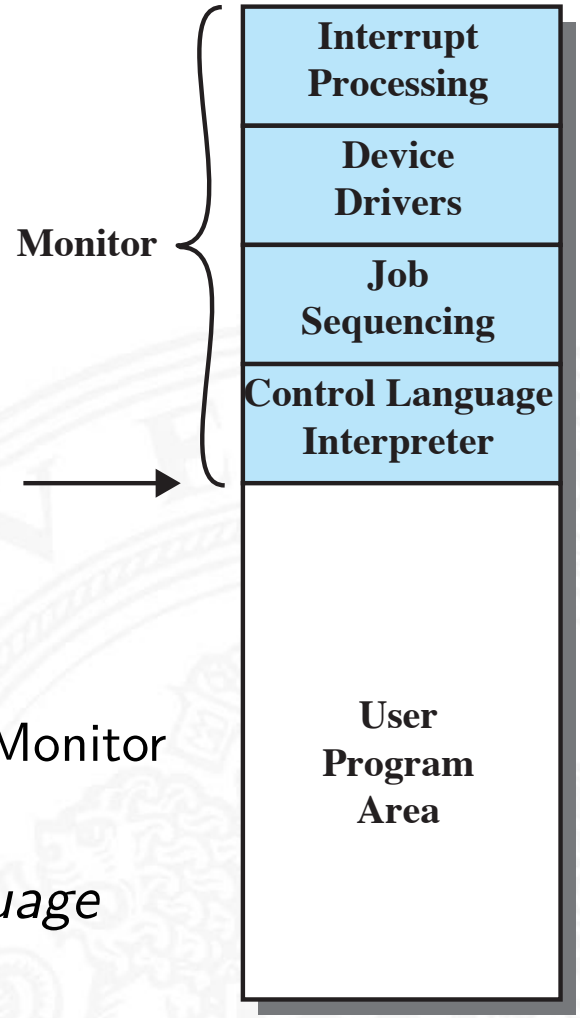
  längerer Job: wird nicht fertig oder
  Reservierungen verschieben sich

  kürzerer Job: System bleibt ungenutzt

– „Rüstzeit“: Vorbereitung auf Programmlauf

2. einfache Batch-Systeme

- Benutzer hat keinen direkten Zugriff
- Operator bündelt Jobs als „*Batch*“
- *Monitor* als zentrales Programm arbeitet Job-Queue ab
- immer im Speicher (*Resident Monitor*)
- Funktionsweise
  - Monitor liest Job ein
  - übergibt Kontrolle an Benutzerprogramm ($\widehat{=}$ Prozeduraufruf)
  - Programm übergibt nach Ende Kontrolle an Monitor ($\widehat{=}$ Rücksprung)
- Instruktionen für Monitor: *Job Control Language*



2. einfache Batch-Systeme (cont.)

- wichtige Eigenschaften
  - **Memory protection**: Jobs haben keinen Zugriff auf Monitor-Speicherbereich
  - **Timer** begrenzt Laufzeit von Jobs
  - **privilegierte Instruktionen** nur durch Monitor ausführbar
  - **Interrupts** bessere, flexiblere Kontrolle der Jobs

$\Rightarrow$ zwei Modi

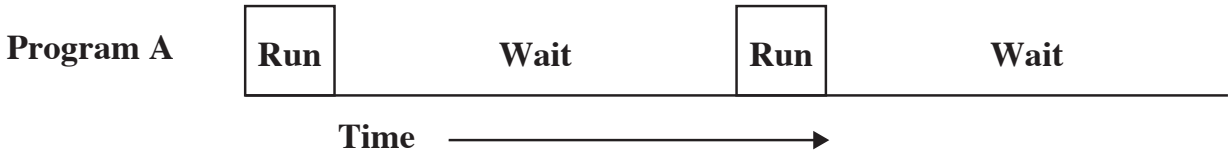
1. **User Mode** für Batch-Job
   - einige Speicherbereiche sind gesperrt
   - einige Befehle sind nicht ausführbar
2. **Kernel Mode** für Monitor
   - Zugriff auf geschützte Speicheradressen
   - privilegierte Befehle sind ausführbar

- Overhead, verglichen mit serieller Abarbeitung
  - $-$ Prozessor muss zusätzlich Monitor bearbeiten
  - $-$ zusätzlicher Speicherbedarf für Monitor
  - $+$ insgesamt aber bessere Auslastung des Computers

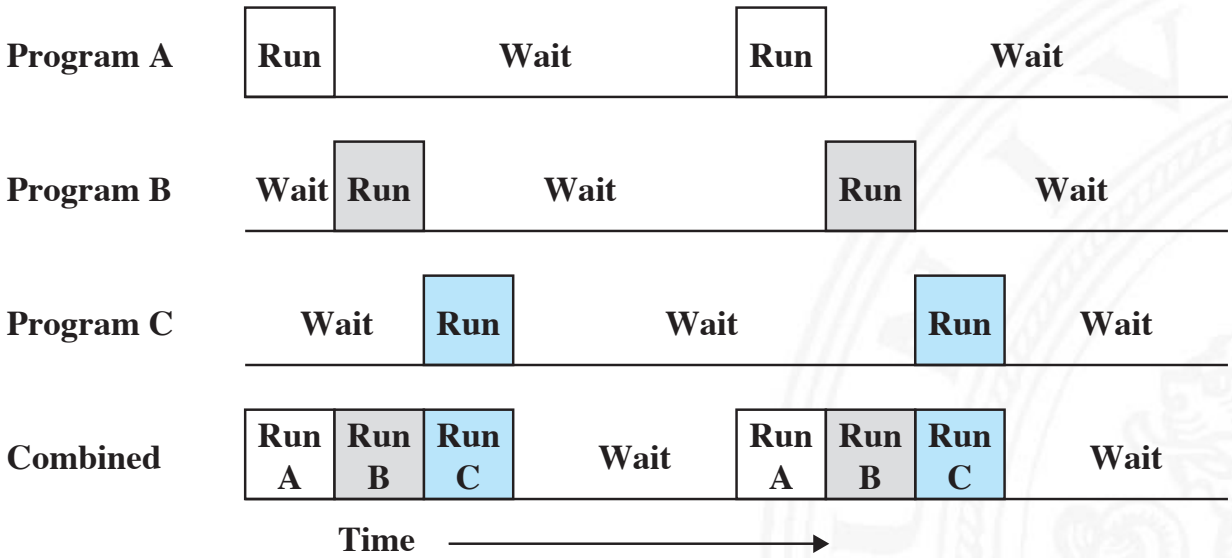
3. Multiprogramm Batch-Systeme

unterschiedliche Geschwindigkeiten ⇒ Prozessor wartet meist

- Uniprogramming



- Multiprogramming, Multitasking



\+ Job wartet auf I/O ⇒ Monitor wechselt zu anderem Job

– Speicherbedarf für Monitor und alle Jobs

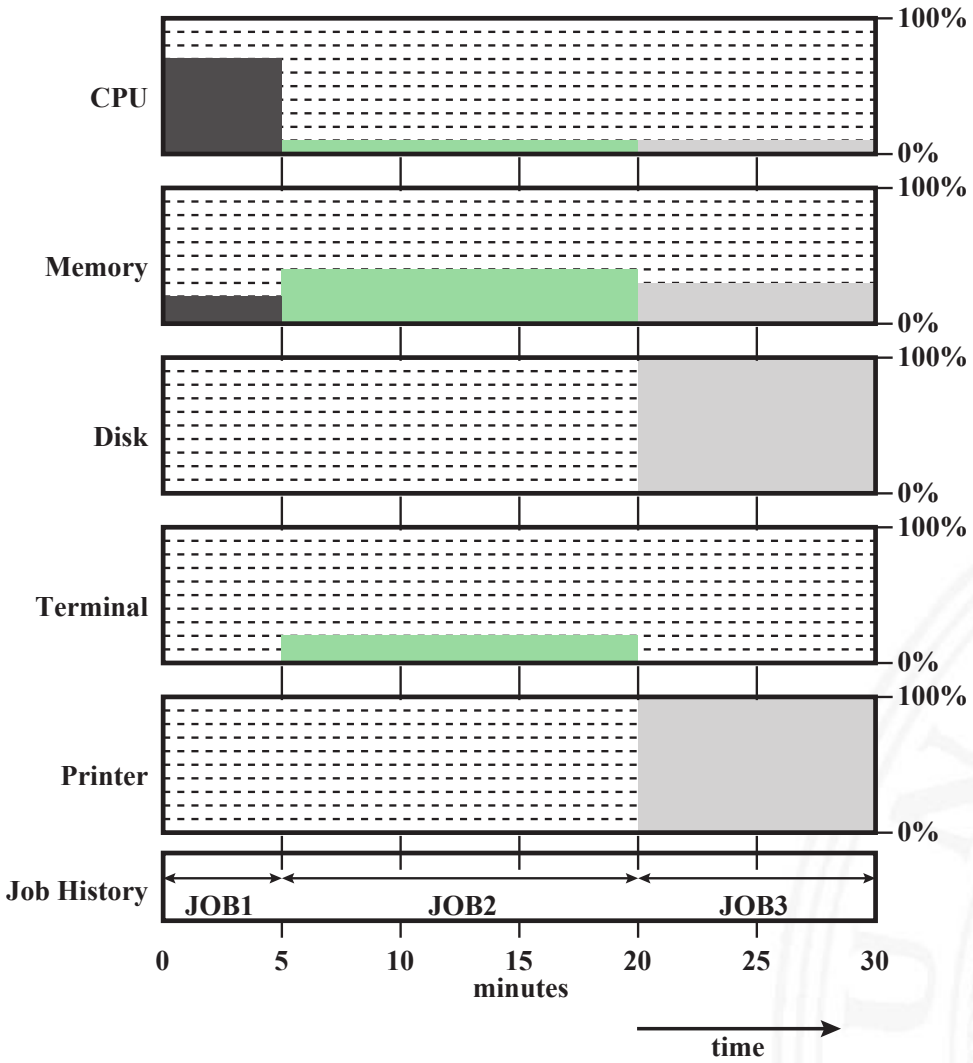
3. Multiprogramm Batch-Systeme (cont.)

- Beispiel

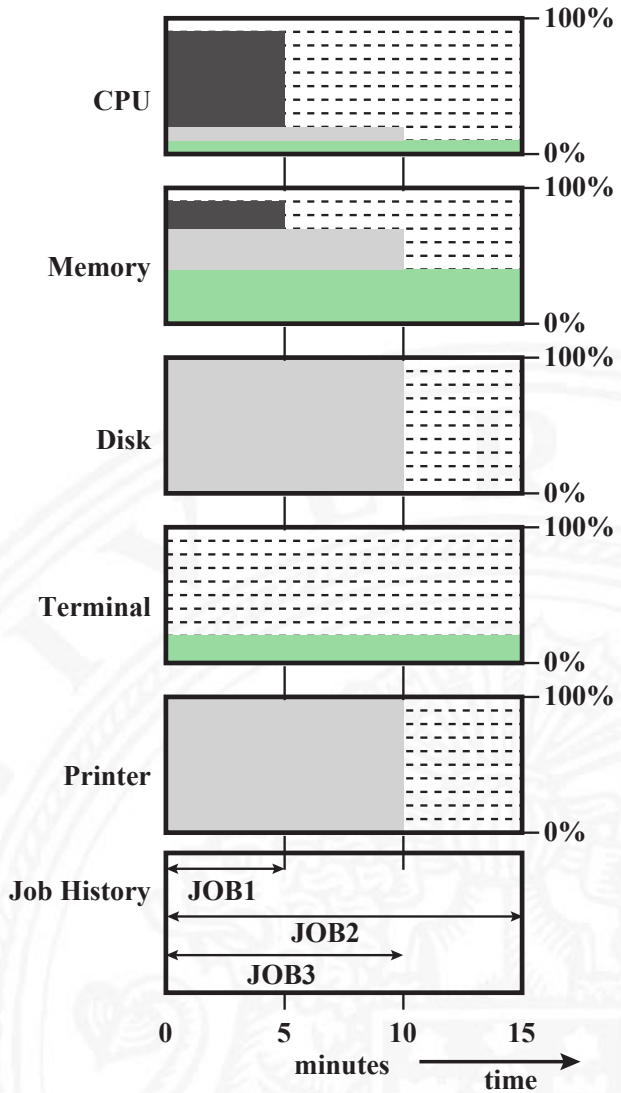
	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

3. Multiprogramm Batch-Systeme (cont.)



(a) Uniprogramming



(b) Multiprogramming

4. *Time-Sharing* Betrieb

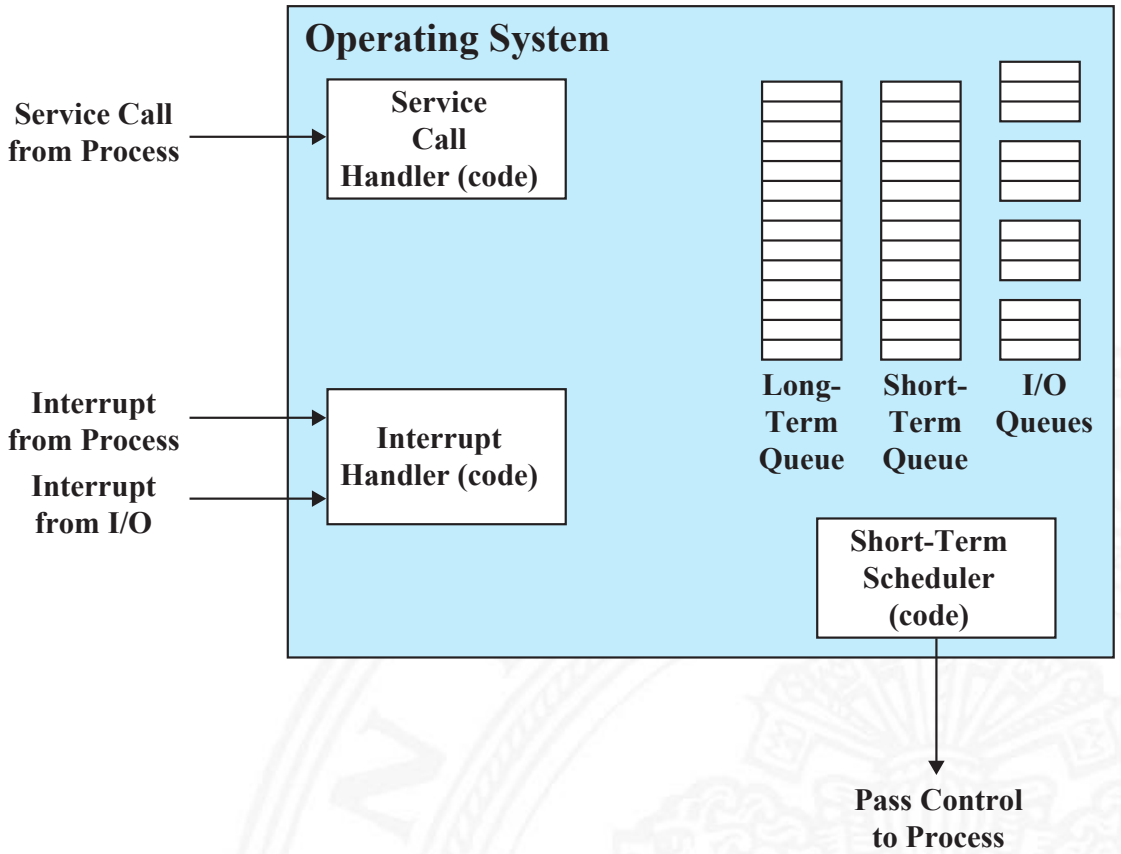
- Erweiterung von Multitasking für interaktive Jobs
- Prozessor/Ressourcen werden zwischen Benutzern geteilt
- Zugriff über Terminals (Kommandozeile)
  später grafische Oberflächen
- Optimierungsziel

	Batch Multiprogramm	Time-Sharing
Optimierung	maximale Prozessornutzung	minimale Antwortzeit
BS Kontrolle	Job Control Language	Benutzereingabe

- typisch: Zeitscheiben Verfahren (*Time Slicing*)
  - periodische Interrupts durch Systemclock
  - Betriebssystem übernimmt Kontrolle
  - prüft ob anderer Prozess laufen soll
  - Benutzerprozess ist „unterbrochen“ („*Preemption*“)
  - sein Status wird gesichert
  - Daten für neuen / fortzusetzenden Prozess werden geladen

aktuelle Betriebssysteme

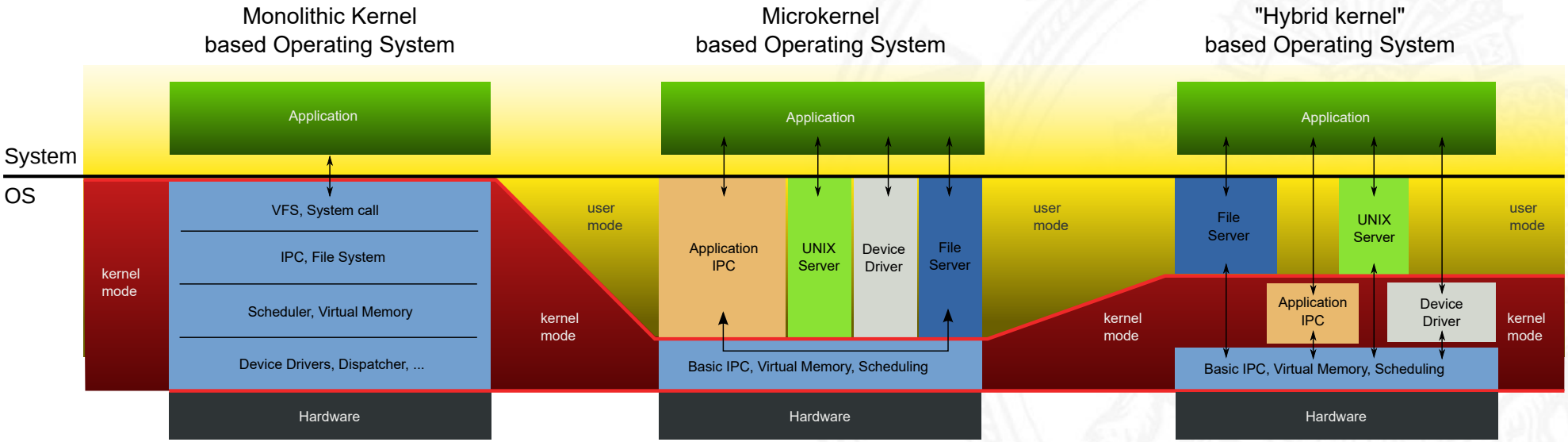
- zentrale Elemente



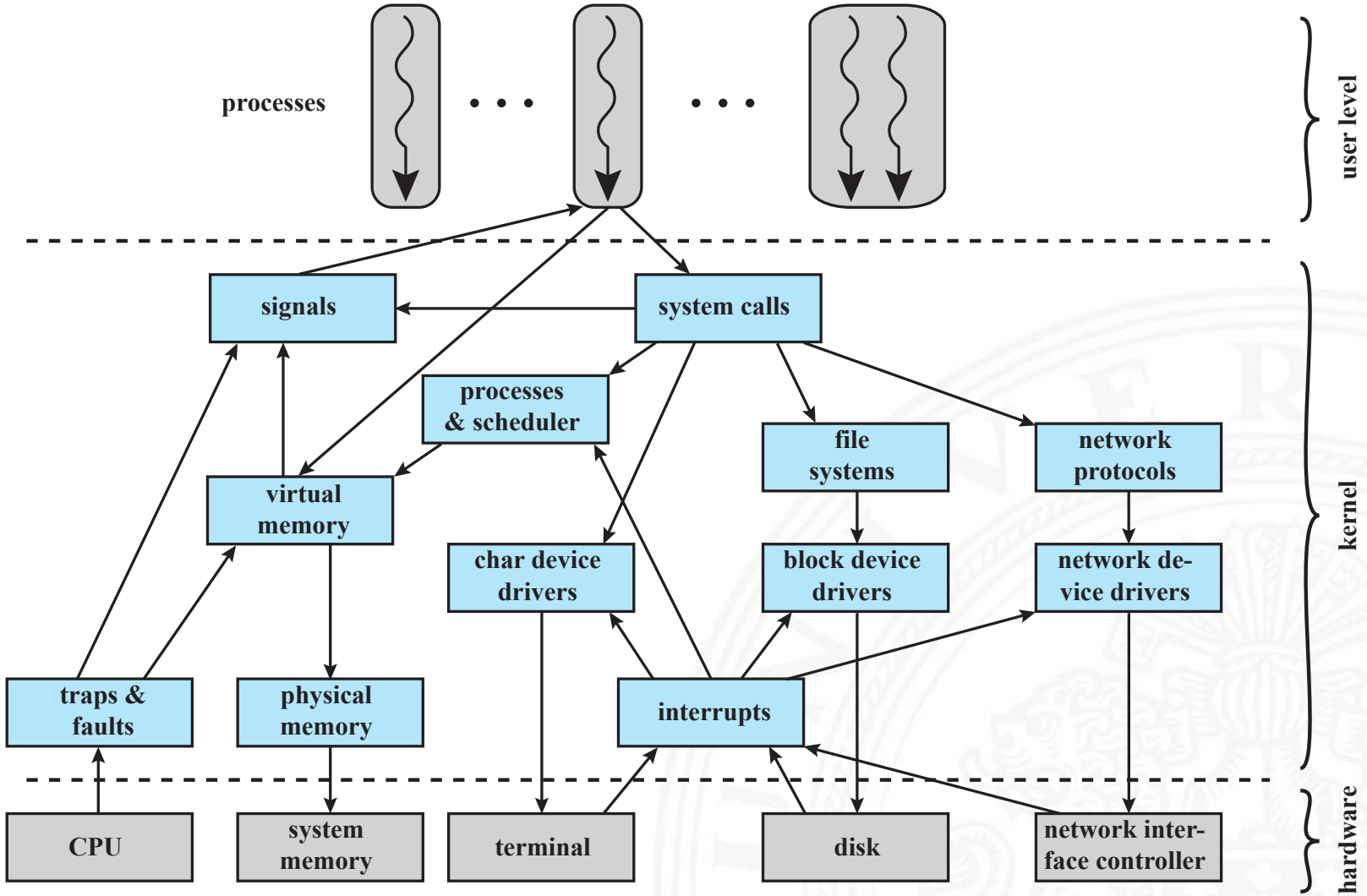
- **Service Call / System Call**:
  Programm ruft BS-Funktion auf $\Rightarrow$ Ressourcenzugriff
- **Interrupt**: besonderes Ereignis
- **FIFO Queues**: Warteschlangen
- **Scheduler**: CPU / Kontrolle wird an Prozess übergeben

aktuelle Betriebssysteme (cont.)

- Architekturansätze
  - **Monolithischer Kernel** alle Funktionalitäten, Treiber etc. in gemeinsamen Kernel; das Programm „Betriebssystem“
  - **Microkernel** enthält nur
    - Scheduling
    - Interprozess-Kommunikation
    - Adressverwaltung
    - restliche Funktionalität als getrennte Prozesse
  - **hybride Kernel** Mischformen



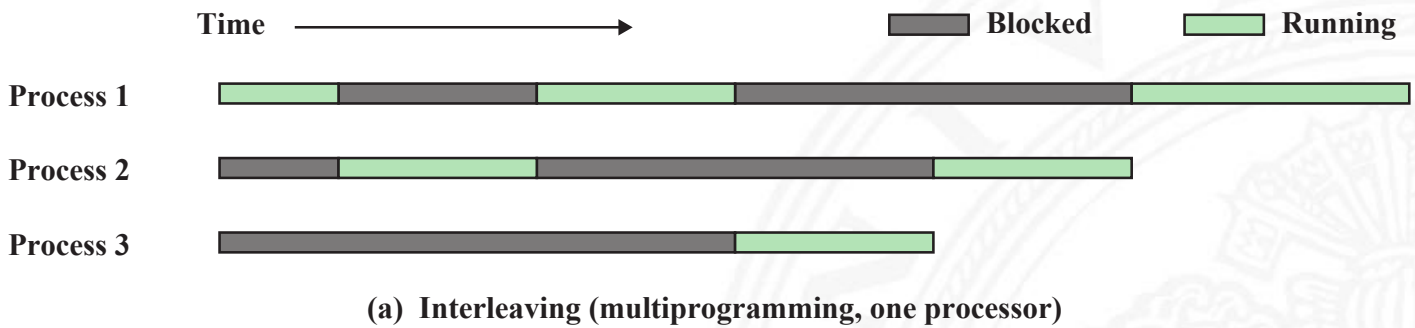
aktuelle Betriebssysteme (cont.)



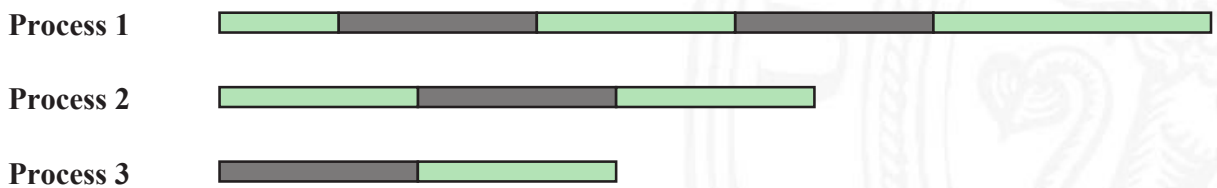
Monolithischer Kernel (Linux): Teilkomponenten

aktuelle Betriebssysteme (cont.)

- weitere Konzepte
  - Multithreading $\Rightarrow$ bessere Granularität
  - Multiprocessing (SMP)
    - Verwaltung mehrerer Prozessoren
    - für Benutzer transparent
    + Verfügbarkeit, Performanz, Skalierbarkeit etc.
    − schwierig zu implementieren ...



(a) Interleaving (multiprogramming, one processor)



(b) Interleaving and overlapping (multiprocessing; two processors)

aktuelle Betriebssysteme (cont.)

- verteilte Betriebssysteme, einheitliche Sicht auf Cluster
- spezielle Anforderungen
  - Echtzeit Betriebssysteme
  - Fehlertoleranz

Aufgaben eines Betriebssystems

- Prozessverwaltung
  - Prozesse starten und beenden
  - Scheduling: Prozesse CPUs zuordnen
  - Prozesswechsel
  - Prozesssynchronisation und Interprozesskommunikation
  - Verwaltung der dazu notwendigen Datenstrukturen (Prozesskontrollblock)
- Speicherverwaltung
  - Zuordnung des (virtuellen) Adressraums zu Prozessen
  - *Swapping*: Hauptspeicher $\Leftrightarrow$ sekundärer Speicher
  - Seitenadressierung (*Paging*) und Segmentierung
- Ein-/Ausgabeverwaltung
  - Verwaltung von FIFOs
  - Zuordnung von I/O-Geräten und -Kanälen zu Prozessen
- weitere Funktionen
  - Interruptverarbeitung
  - Abrechnung der Ressourcen (*Accounting*)
  - Protokollierung (*Monitoring*)

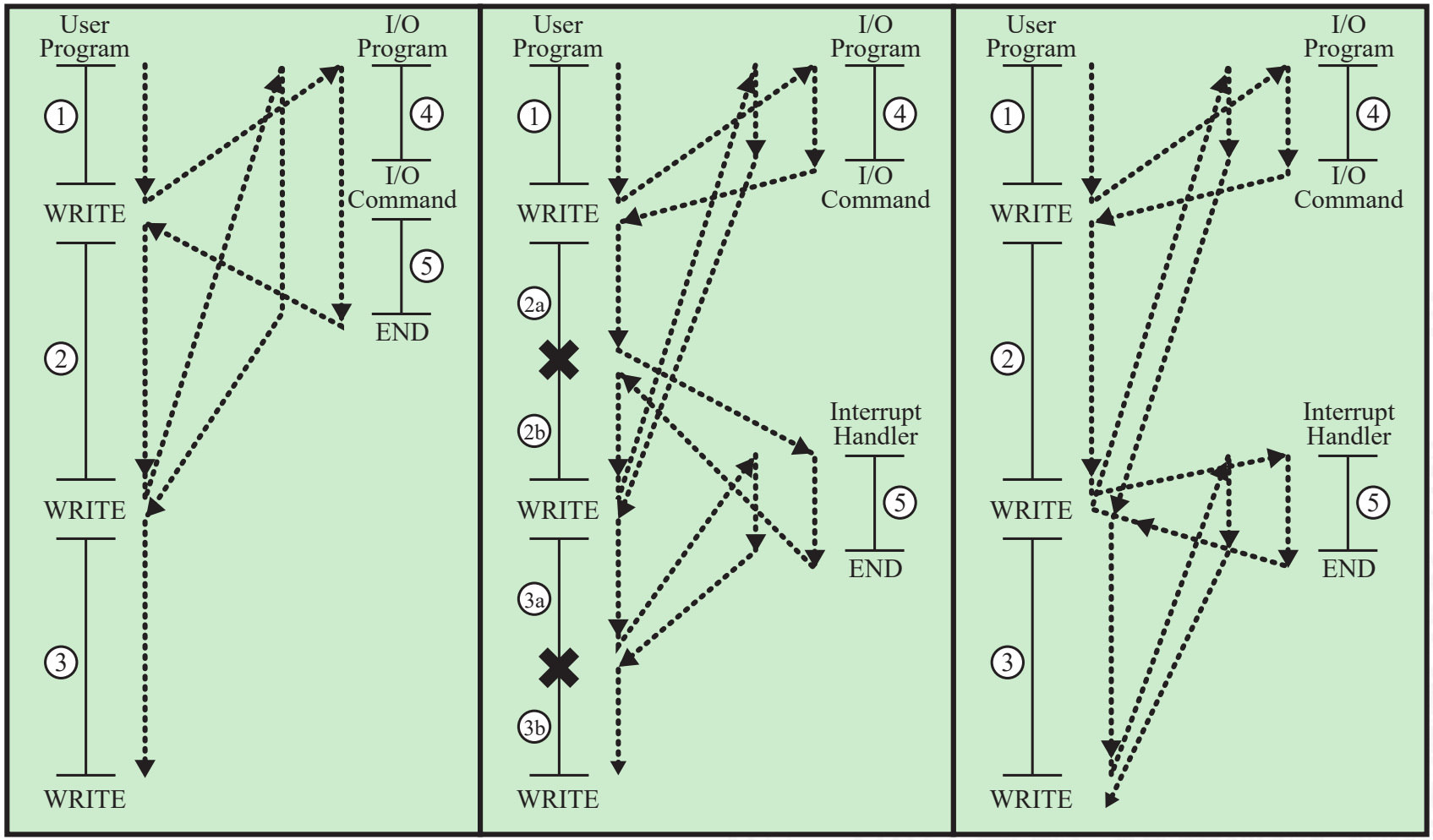
Interrupts

- sequenzieller Ablauf der Programmabarbeitung wird unterbrochen
- Bessere Ausnutzung des Prozessors
  - I/O, Platten, Hauptspeicher langsamer als CPU
  - CPU muss „warten“ $\Rightarrow$ schlechte Nutzung
- Interrupts durch
  - **Programm**: Ausnahmebehandlung („*Exception*“) z.B. Überlauf, Division durch 0, illegale Anweisung, ungültige Speicheradresse
  - **Timer**: regelmäßige Ausführung von Aufgaben
  - **I/O**: externe Hardware meldet: Ende einer Operation, Fehler
  - **Hardwarefehler**: Speicherparität, Spannungsversorgung, Temperatur ...

Interrupt: Beispiel

- Benutzerprogramm schreibt auf Festplatte, rechnet (1,2,3)
- I/O-Programm für Plattenzugriff (4,5)
  - Teil des Betriebssystems
  - Schnittstelle durch *System-Call*
- Zeit zwischen *I/O Command* und *Interrupt* wird durch langsames Gerät bestimmt

Interrupt: Beispiel (cont.)

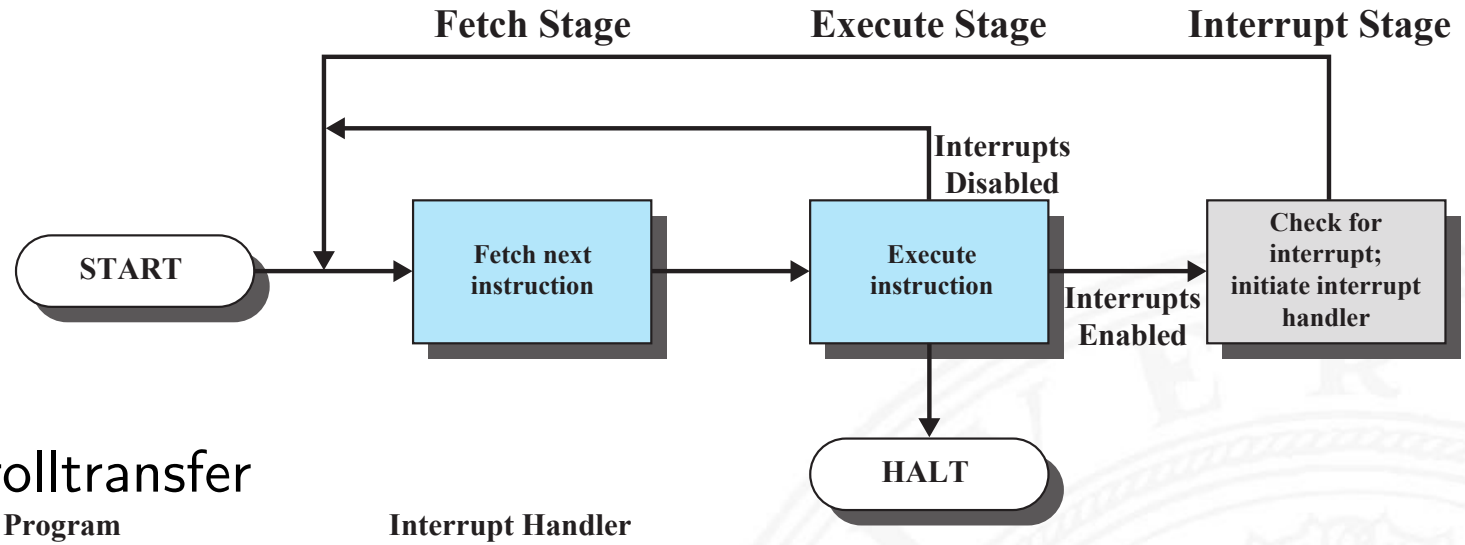


(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

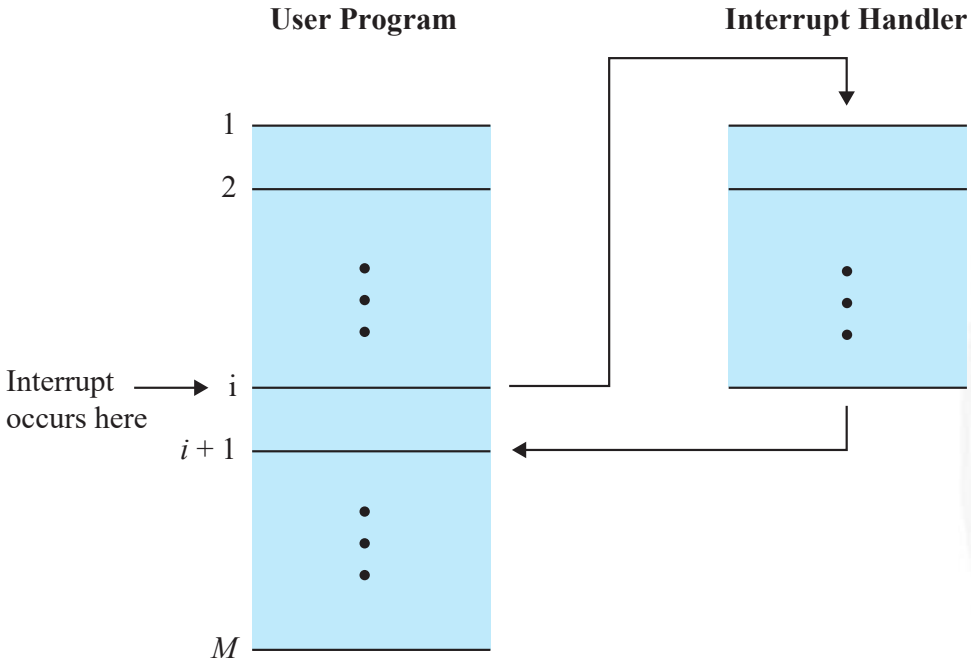
✖ = interrupt occurs during course of execution of user program

Interrupt: Programmablauf

- Ausführungszyklus der Befehle ergänzt

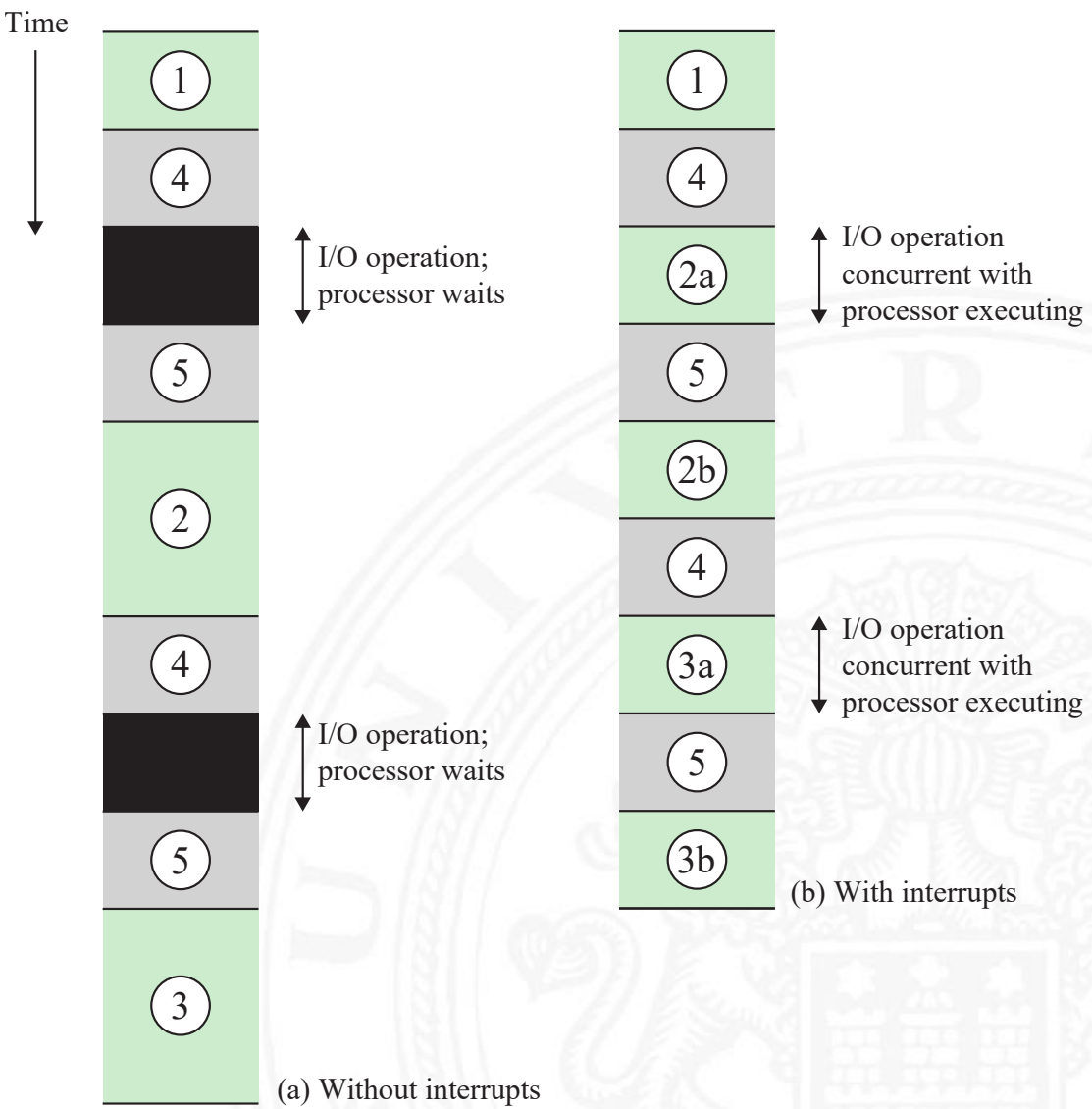


- Kontrolltransfer



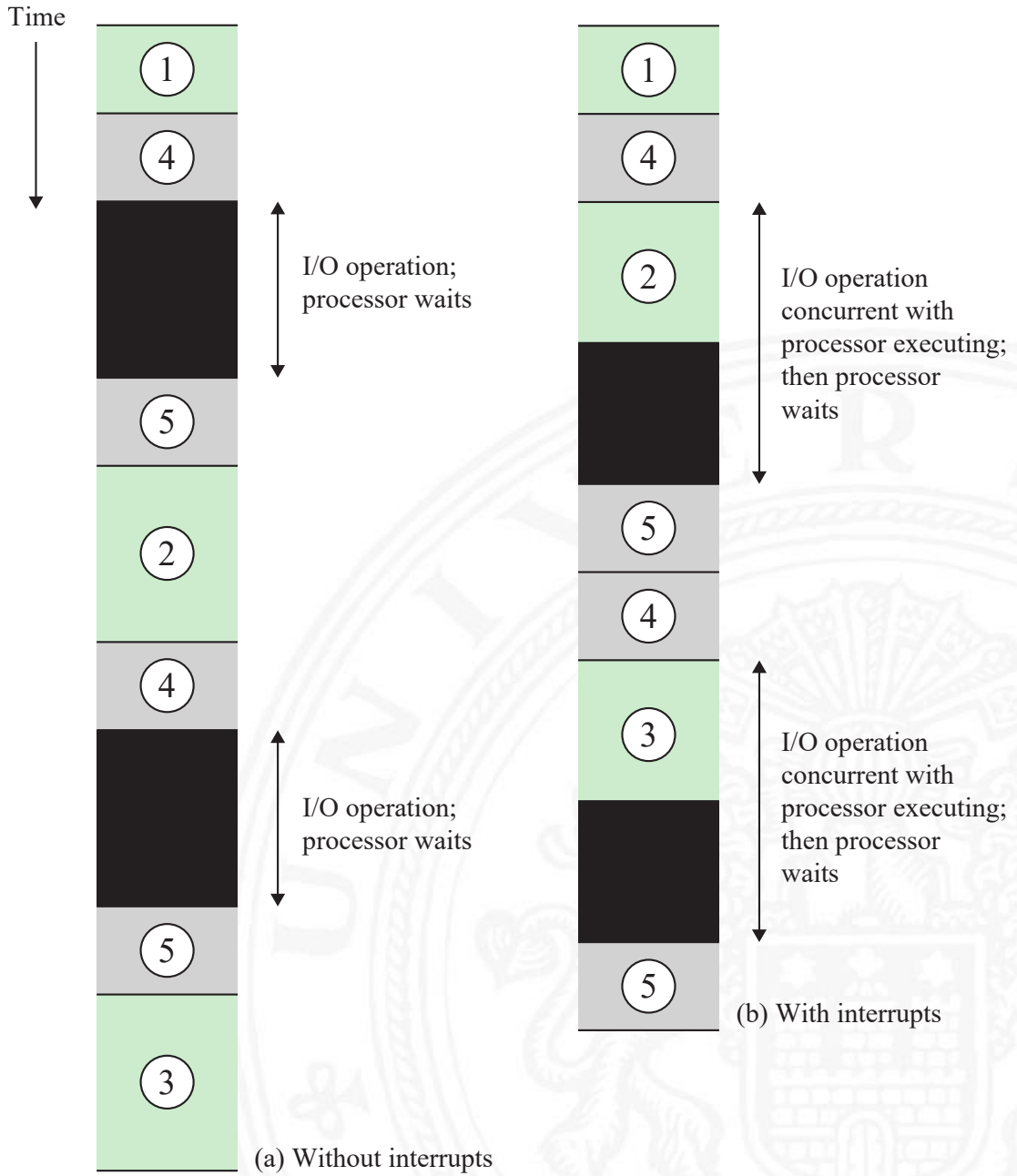
Interrupt: Programmablauf (cont.)

- kurze I/O Wartezeit



Interrupt: Programmablauf (cont.)

- lange I/O Wartezeit

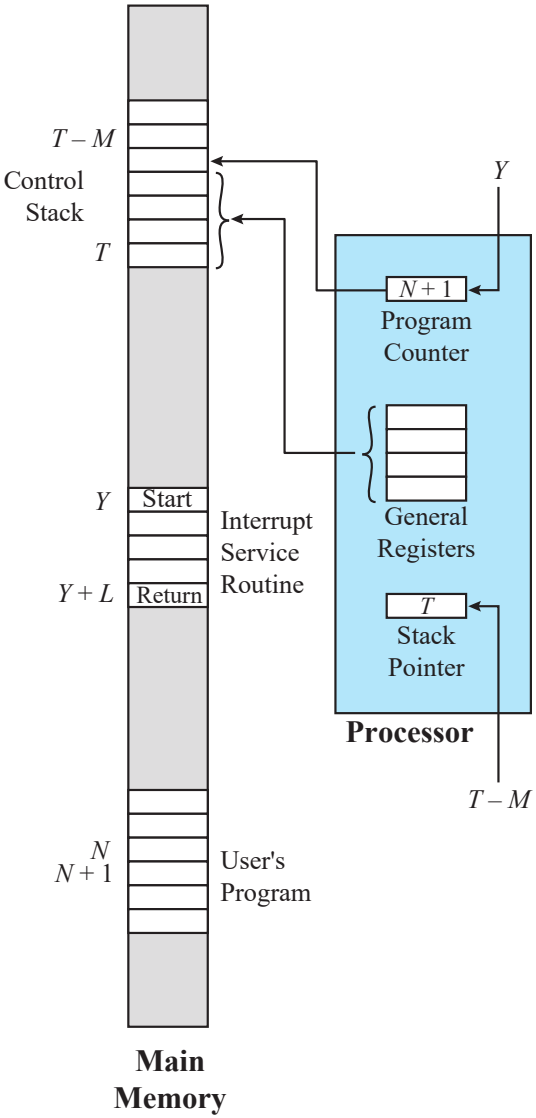


Interruptverarbeitung

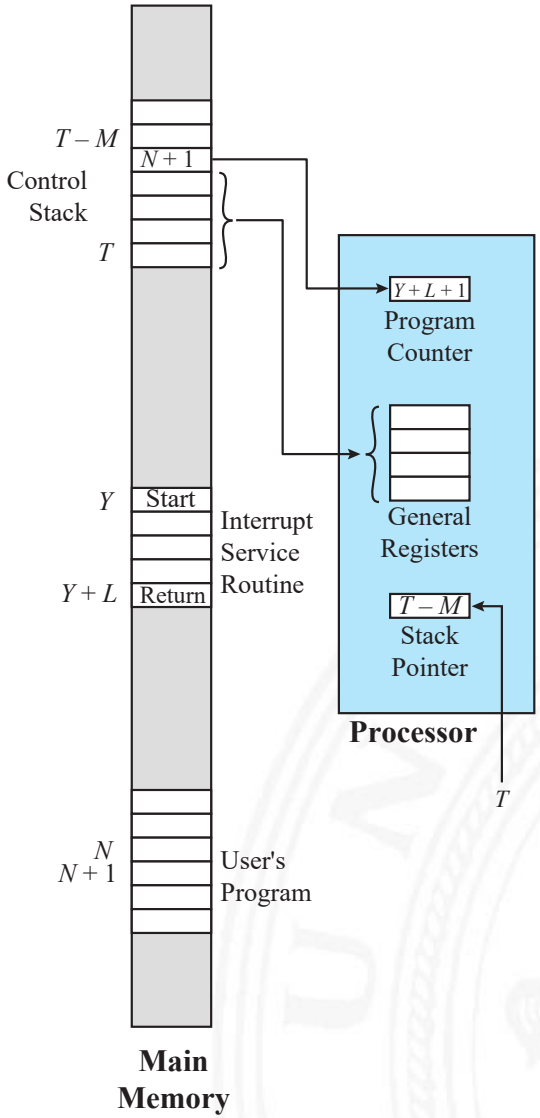
1. HW Gerät, Systemhardware liefert Interrupt
2. HW Prozessor beendet aktiven Befehl Pipelining!
3. HW Prozessor bestätigt Interrupt
4. HW Programmstatus (PC, Register, Speicherinformation etc.) auf *Control Stack* sichern
5. HW PC mit Interrupt (-startadresse) initialisieren

   Wechsel in privilegierten Modus (*Kernel Mode*)
6. SW ggf. weitere Informationen auf *Control Stack* sichern
7. SW Interruptverarbeitung / *Interrupt Handler* (Programm)
8. SW Programmstatus aus 6. wiederherstellen
9. SW Programmstatus aus 4. wiederherstellen;
   PC mit Programmfortsetzung initialisieren

Interruptverarbeitung (cont.)



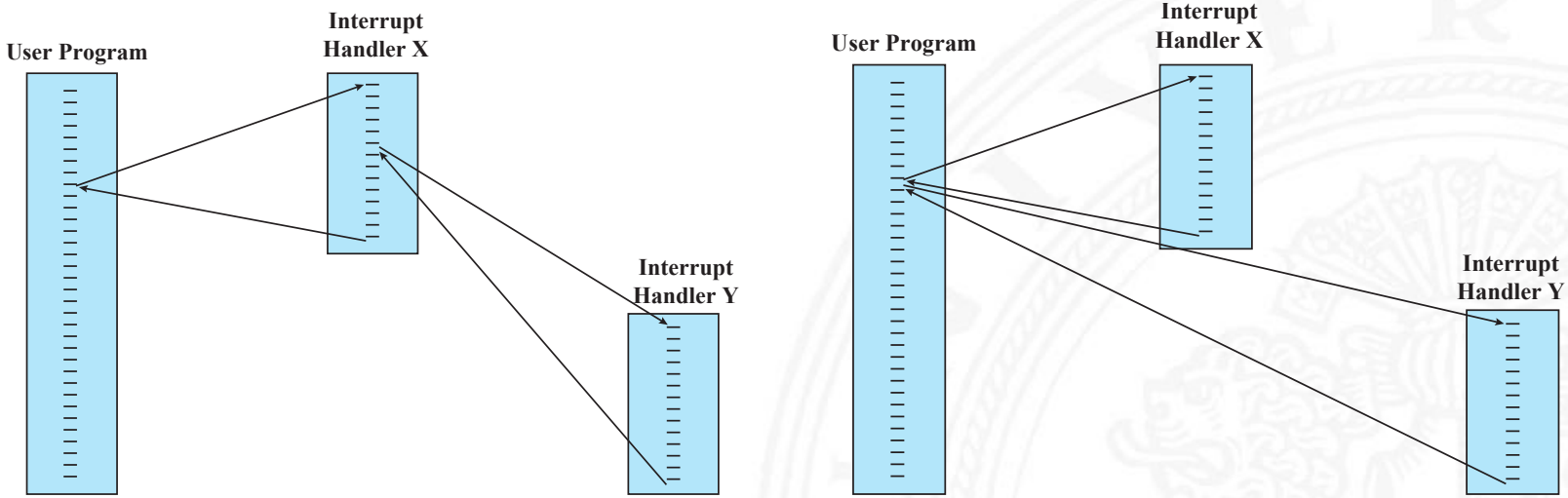
(a) Interrupt occurs after instruction at location N



(b) Return from interrupt

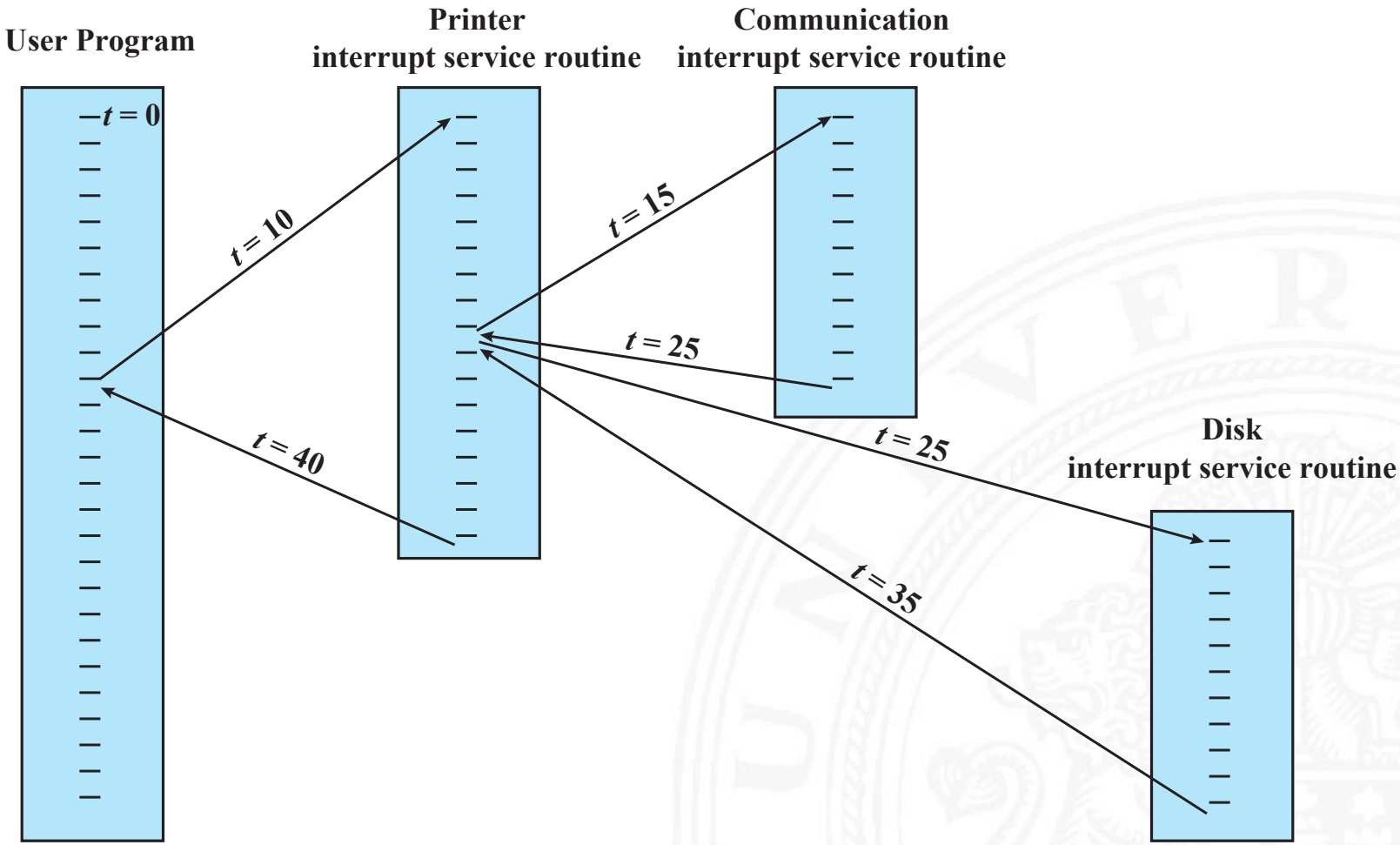
Mehrfache Interrupts

- während Interruptverarbeitung kommt Interrupt
  1. weitere Interrupts deaktivieren
  2. verschiedene Interruptprioritäten
- Interrupts können „verloren gehen“, ggf. Zwischenspeichern
- Schachtelung und/oder sequenzielle Abarbeitung



Mehrfache Interrupts (cont.)

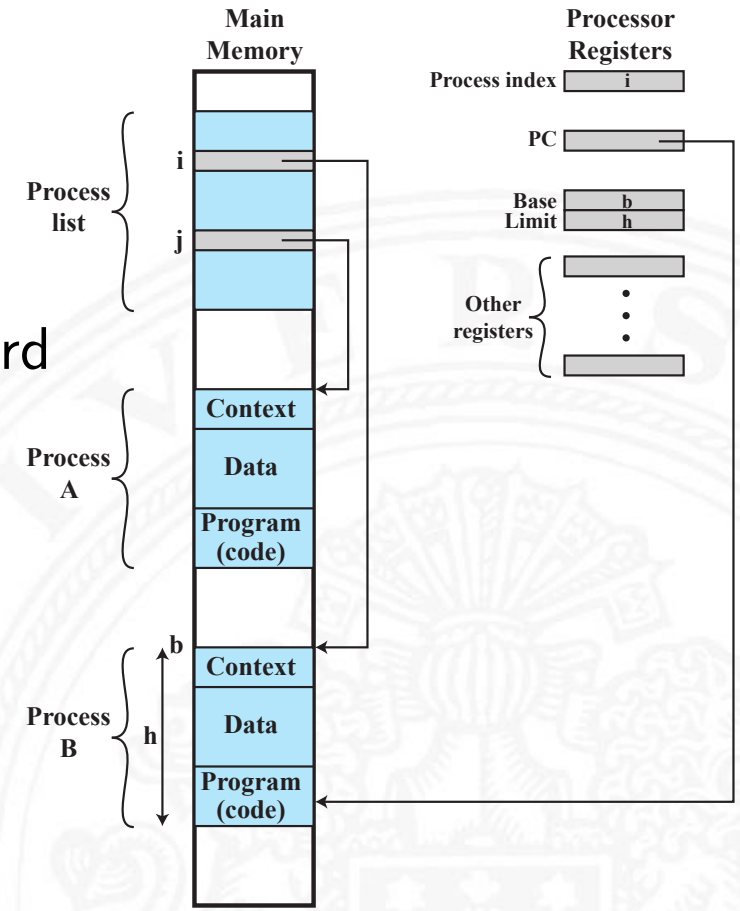
- Beispiel: zeitlicher Verlauf



Prozesse

Prozess: zentral verwaltete Einheit in Betriebssystemen

- Programm während der Ausführung
- Instanz eines laufenden Programms
- Einheit, die Prozessor zugewiesen wird
  –"– die von Prozessor ausgeführt wird
- Aktivität bestehend aus
  - einem einzelnem sequenziellen Ablauf
  - einem Zustand
  - zugehörigen Systemressourcen



Fehler bei Prozessabarbeitung

- Synchronisationsfehler
  - Prozess muss warten
  - Reaktivierung durch externes Ereignis

  $\Rightarrow$ Ereignis wird nicht, wird mehrfach ausgelöst
- gegenseitiger Ausschluss (*Mutual Exclusion*)
  - mehrere Prozesse mit gleichzeitigen Zugriff auf Ressource, z.B. Speicher, Datei

  $\Rightarrow$ Sperrmechanismen: Semaphore, Mutex, Monitor
- nichtdeterministisches Verhalten
  - mehrere Prozesse/Threads kommunizieren über *Shared Memory*

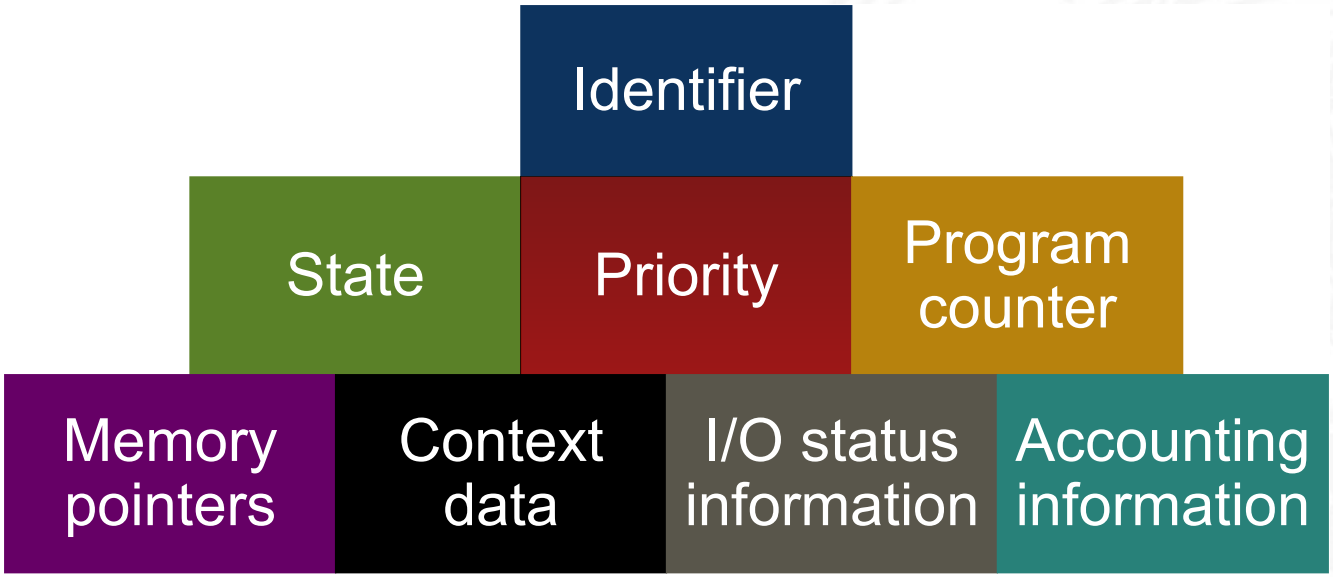
  $\Rightarrow$ transiente Effekte: Programme überschreiben sich Werte abhängig vom Scheduling durch Betriebssystem
- Deadlocks

  $\Rightarrow$ Prozesse warten (zyklisch) aufeinander

$\rightarrow$ siehe Abschnitt *15.4 Synchronisation und Kommunikation*

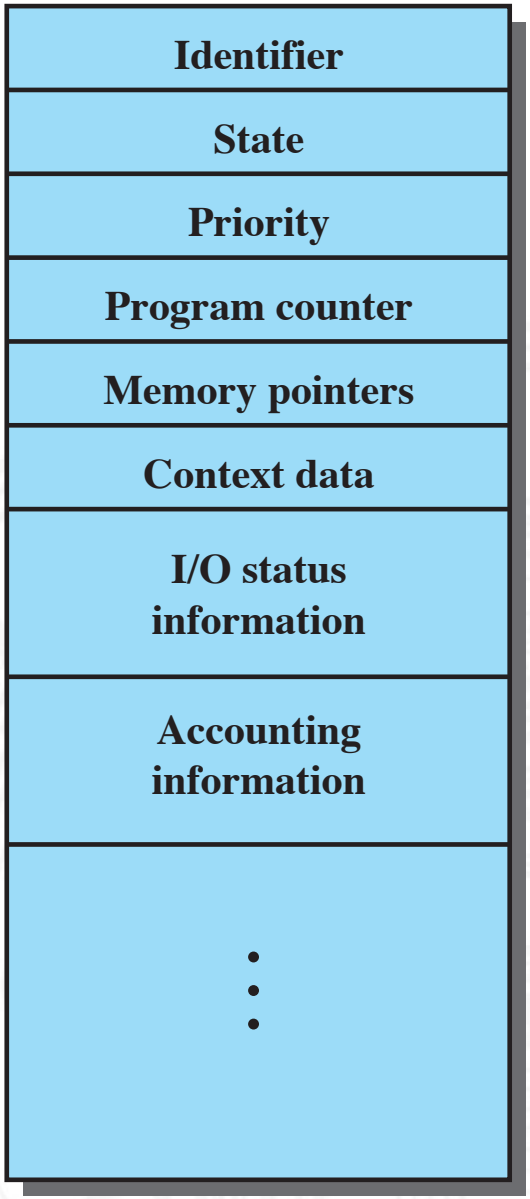
Komponenten eines Prozesses

1. das ausführbare Programm (*Text*-Segment)
2. die zugehörigen Daten (*Data*-Segment)
3. der Programmkontext
   - prozessspezifische Daten des Betriebssystems
   - Inhalt der Prozessorregister
   - Warten auf Ereignisse?
   - Prioritäten, Rechte, Abrechnungsinformationen etc.



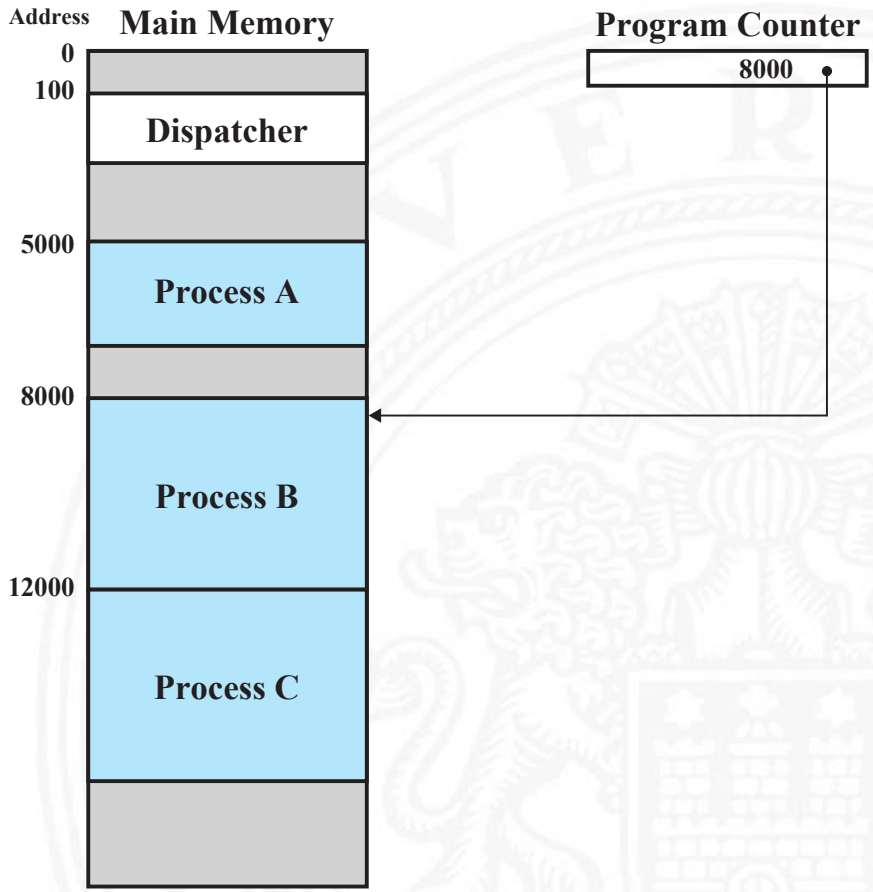
Komponenten eines Prozesses (cont.)

- Prozesskontrollblock
  speichert Verwaltungsinformation



Prozessablauf

- *Trace*: Folge von Instruktionen
  - für einzelnen Prozess ⇒ Laufzeitverhalten
  - für Prozessor ⇒ zeigt Prozesswechsel
- *Dispatcher*: BS-Komponente, die Prozessor Prozessen zuordnet
- Beispiel



Prozessablauf (cont.)

- 3 Prozesse: A, B, C
- Dispatcher (blau)

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A **(b) Trace of Process B** **(c) Trace of Process C**

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
----------Timeout	
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
---------- I/O Request	
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
----------Timeout	
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
----------Timeout	
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
----------Timeout	

laufende Prozesse

- Prozesse starten
  - neuer Batch-Job
  - interaktiver Login (Kommandozeile / *Shell*)
  - durch Betriebssystem: neuer Dienst, z.B.: nach Booten
  - durch laufenden Prozess: explizite Parallelität

- *Parent*: laufender Prozess, der Neuen startet

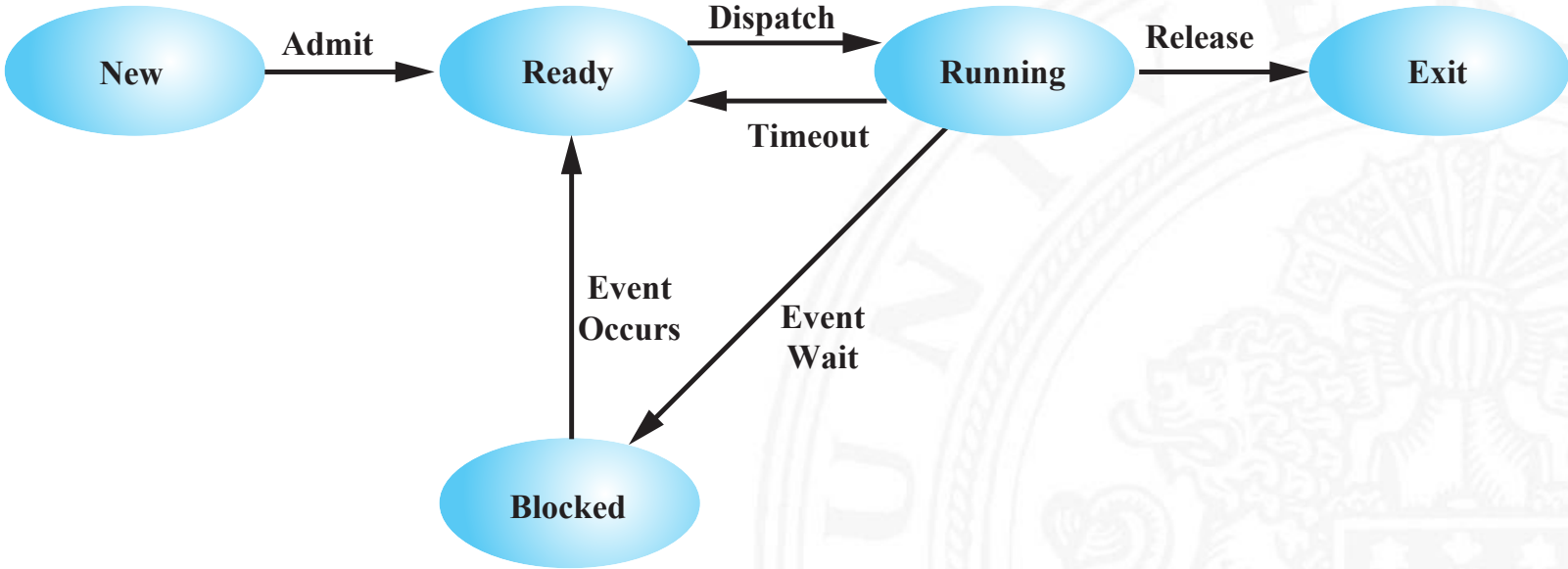
  Child: neu gestarteter Prozess
- Prozesshierarchie (Baumstruktur) (`htop`)

laufende Prozesse (cont.)

- Prozessende
  - normales Programmende, Berechnung/Aufgabe fertig
  - Zeitlimit überschritten:
    maximale Laufzeit, Warten auf Benutzereingabe ...
  - Timeout: Warten auf Event/Systemsignal
  - Speicherlimit: kein (virtueller) Speicher mehr verfügbar
  - Adressverletzung: versuchter Zugriff auf ungültige Speicheradresse
  - Zugriffsfehler: ungültiger Zugriff auf Ressource,
    z.B.: Schreiben in read-only Datei
  - Ein-/Ausgabefehler:
    Lesefehler in Datei, Datei nicht vorhanden ...
  - Arithmetischer Fehler: Teilen durch 0 ...
  - Datenfehler: „falscher“ Typ in Datenstrukturen ...
  - ungültiger Befehl: kein Assemblerbefehl (in Datensegment?)
  - unerlaubter Befehl: privilegierter Befehl im User Mode
  - Parent Anfrage an Child
  - Parent terminiert $\Rightarrow$ Child-Prozess beenden
  - Abbruch durch: Betriebssystem, Operator, Benutzer

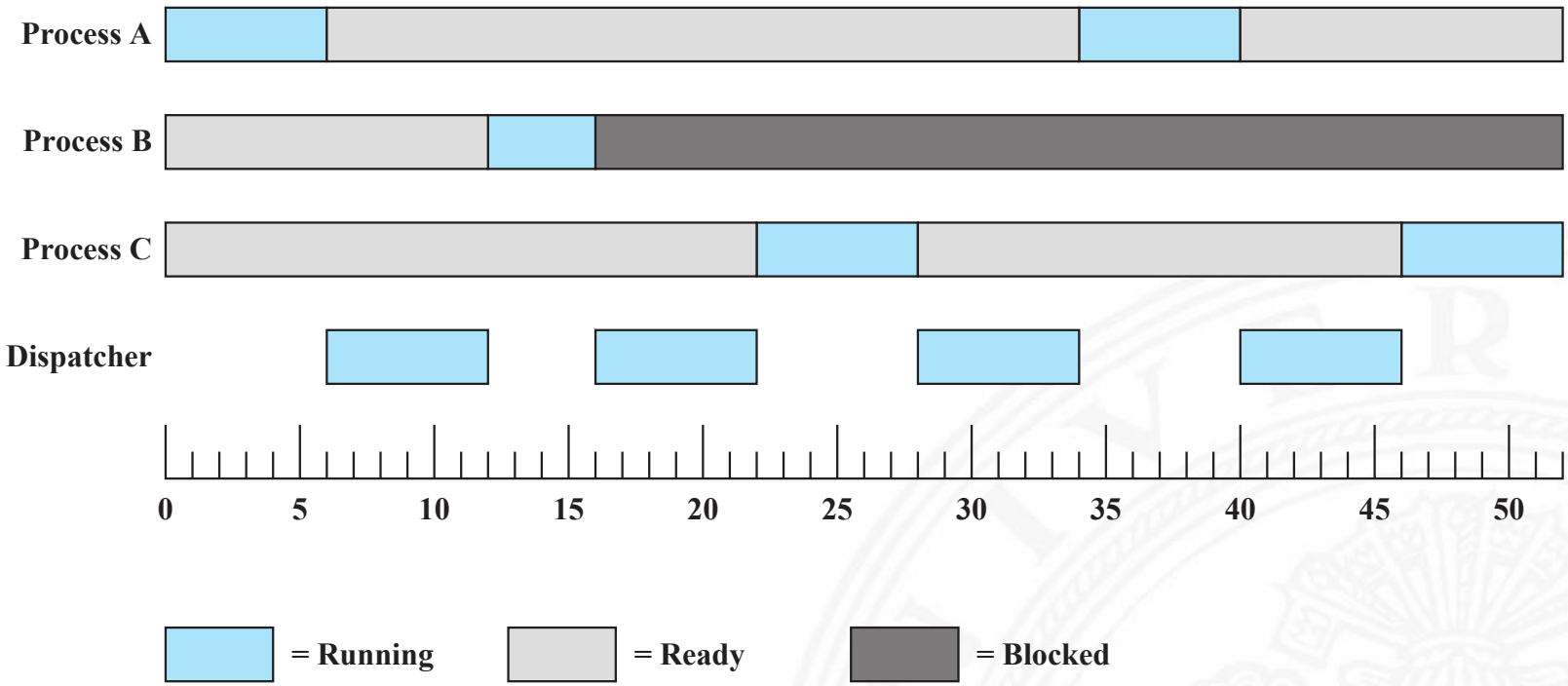
Prozessmodell

- Prozesse werden dynamisch gestartet und beendet
- –"– warten auf I/O / Systemereignisse
- –"– werden unterbrochen: Time-Sharing Betrieb
- Verwaltung durch Dispatcher
- Zustandsautomat



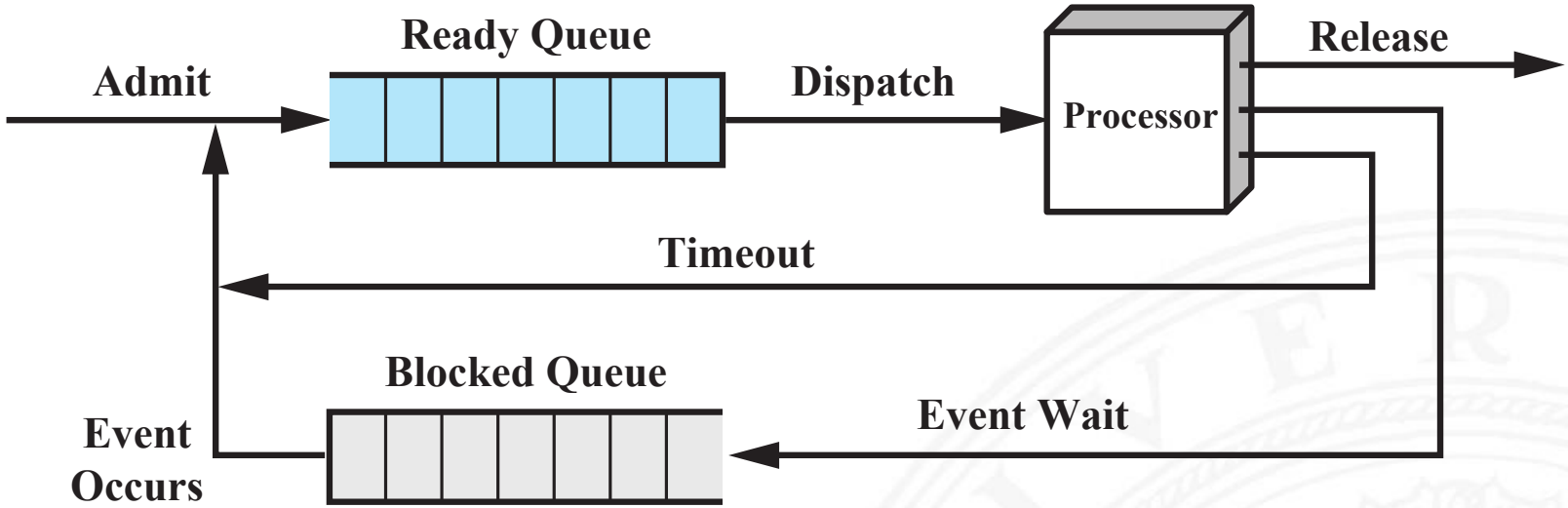
Prozessmodell (cont.)

- Trace

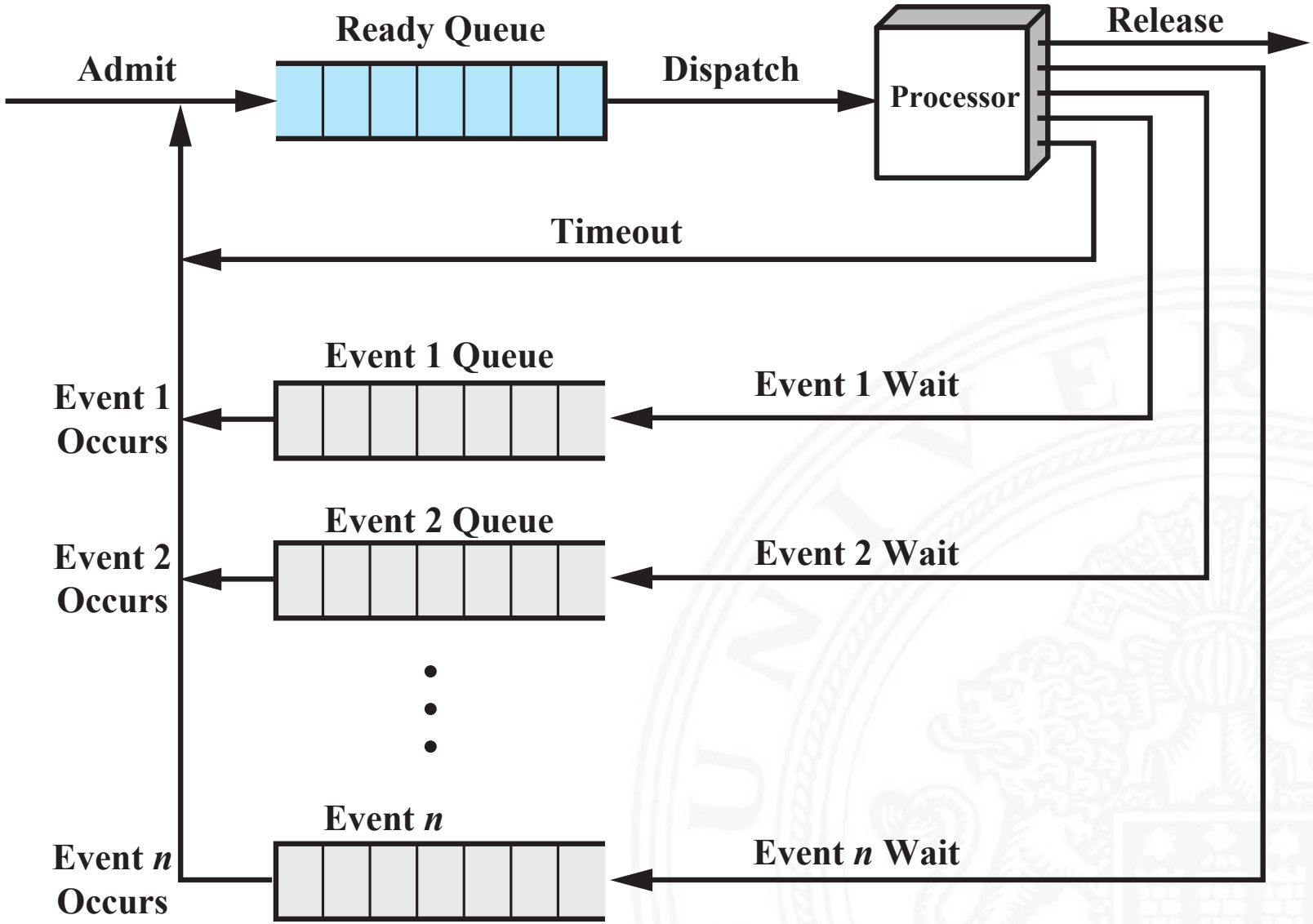


Prozessmodell (cont.)

- Warteschlangen, ggf. mit Trennung nach Signal

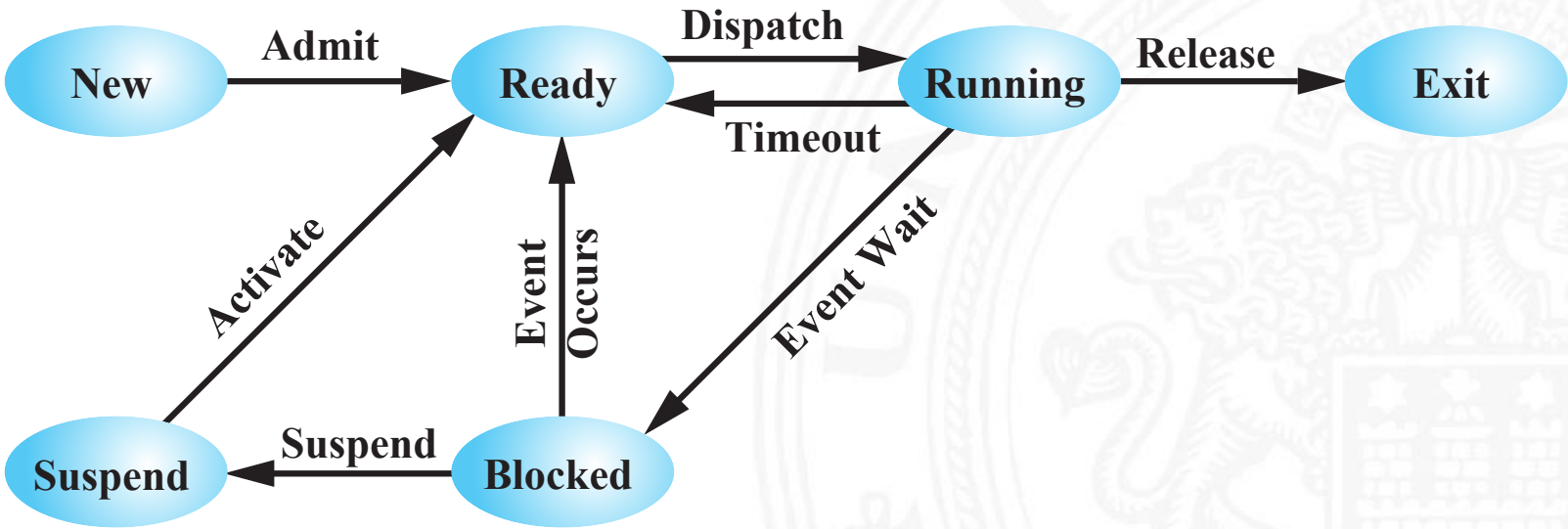


Prozessmodell (cont.)

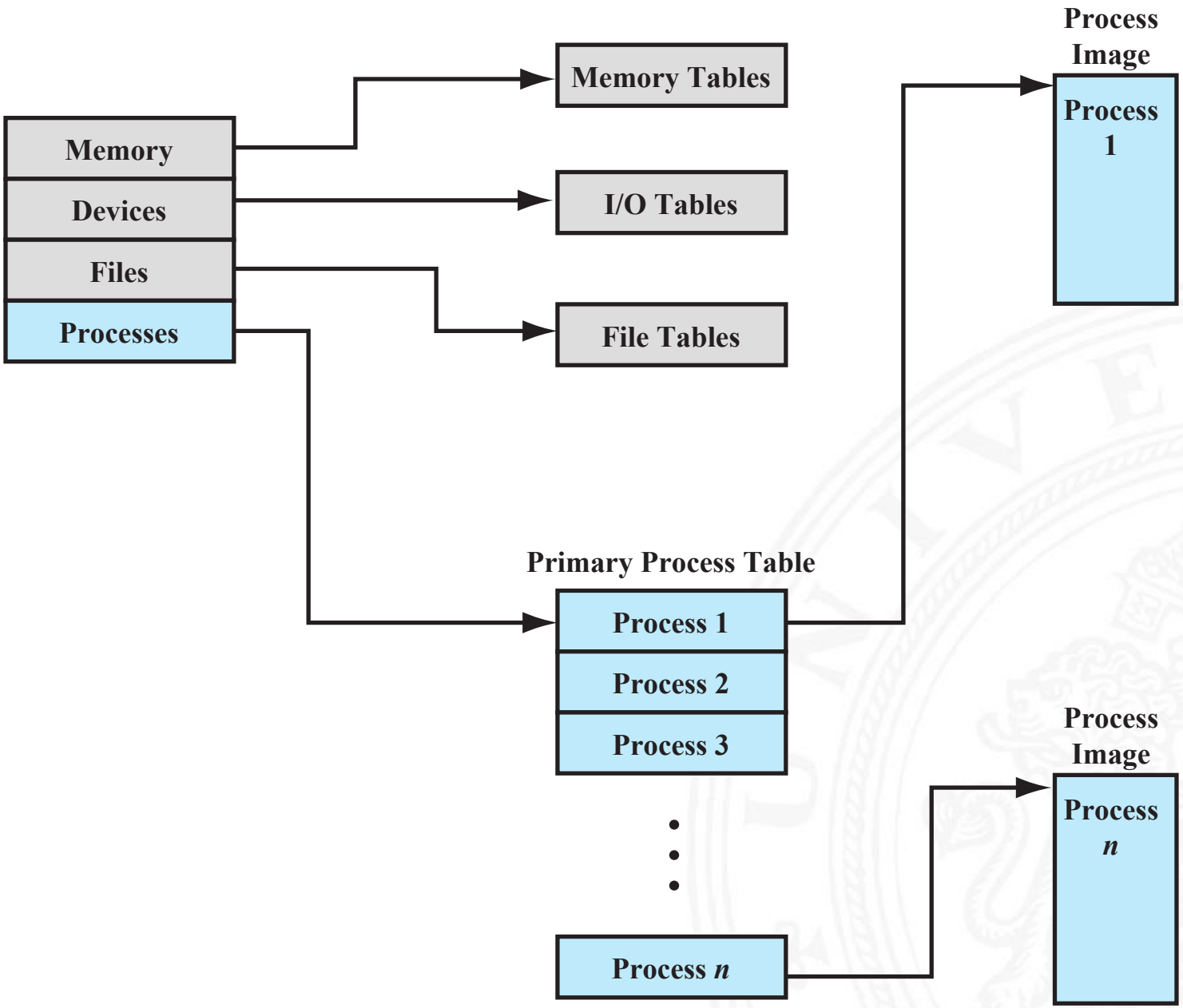


Prozessunterbrechung

- *Swapping*: Auslagerung von Prozessen, bzw. Teilen von Prozessen auf sekundären Speicher (HDD, SSD)
- Prozessunterbrechung (*suspend*) durch
  - Swapping: Betriebssystem benötigt Hauptspeicher
  - Timing: periodische Ausführung ...
  - Parent Anfrage an Child, z.B.: zur Synchronisation
  - Unterbrechung durch: Betriebssystem, Operator, Benutzer
- Erweiterung des Prozessmodells



Kontrollstrukturen



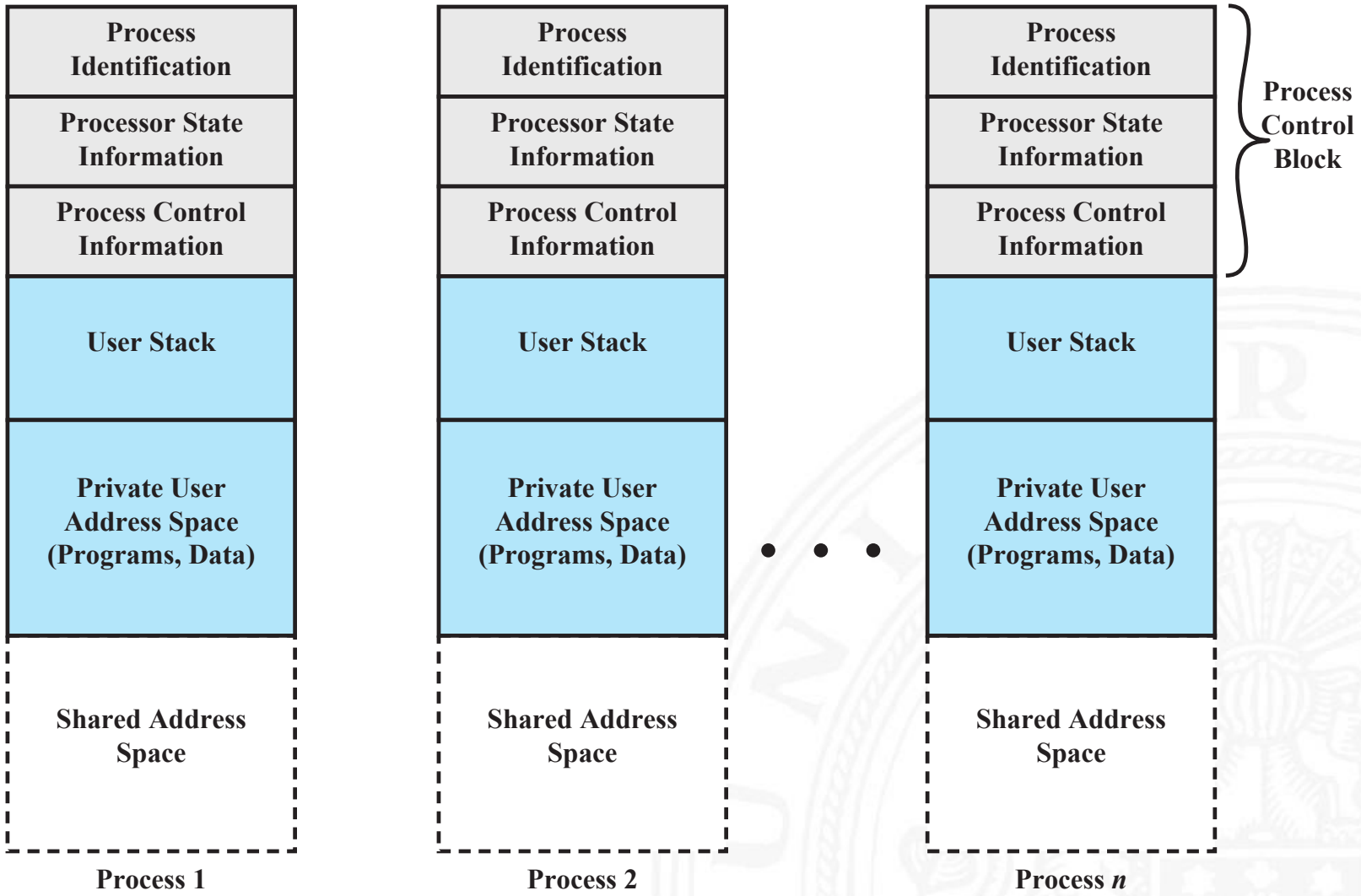
Kontrollstrukturen (cont.)

- Prozesstabellen
- Speicherverwaltung: *Memory Tables*
  - Verwaltung von virtuellem Speicher
  - Zuordnung für Hauptspeicher (RAM)
  - Zuordnung für sekundären Speicher (HDD, SSD)
  - Attribute für Speicherblöcke, z.B.: Speicherschutz
- Ein-/Ausgabeverwaltung: *I/O Tables*
  - Zuordnung zu Prozessen
  - Status von I/O-Befehlen
  - Informationen zu Befehlen: Startadresse in Hauptspeicher, Datengröße
- Datei-Verwaltung: *File Tables*
  - Existenz von Dateien / Dateinamen
  - Ort auf Sekundärspeicher
  - Status, z.B.: geöffnet (rw, ro)
  - weitere Attribute: Zugriffsrechte, Zeitstempel etc.

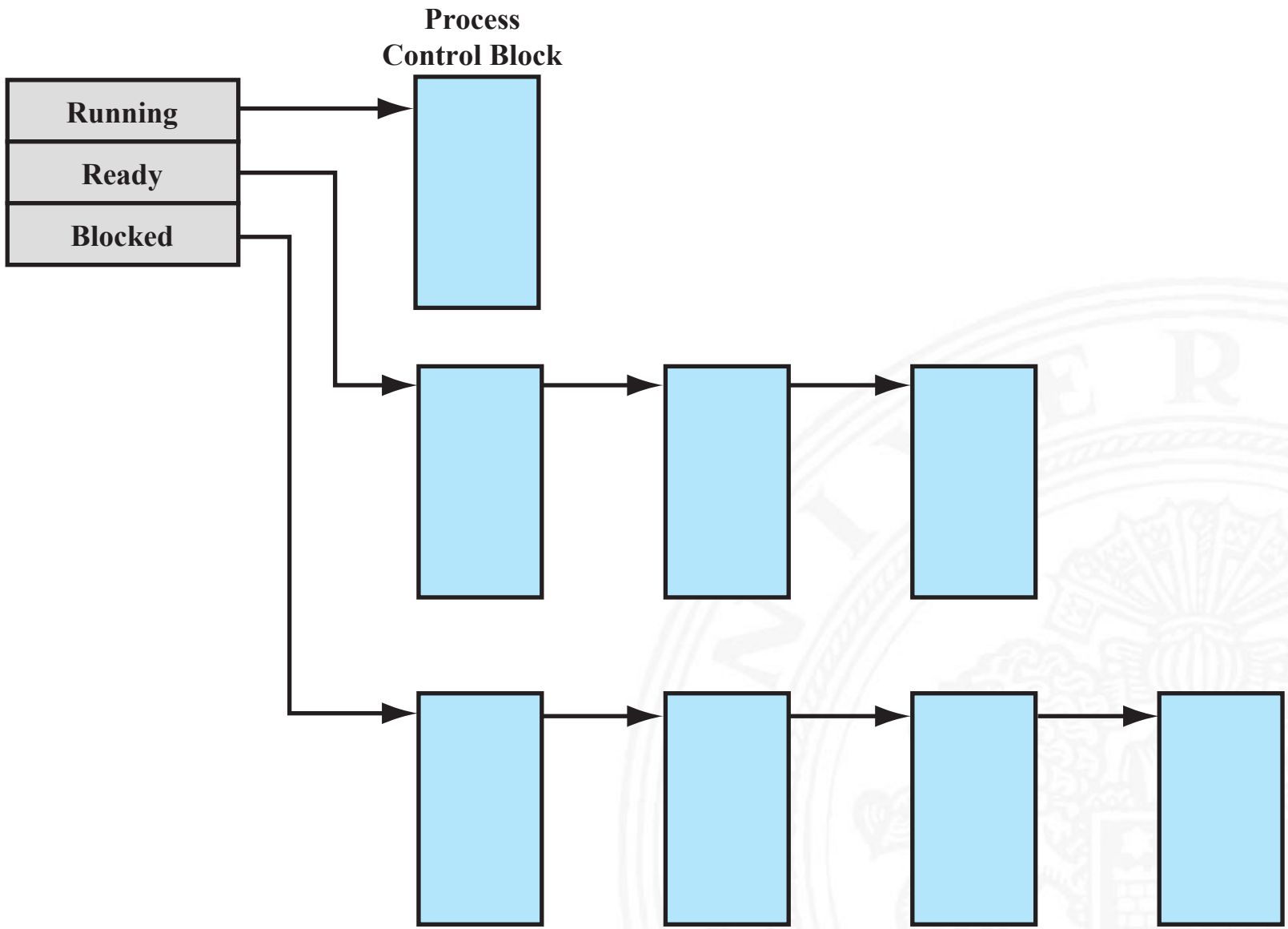
Prozessabbild / *Process image*

- Programmcode
- Datenstrukturen des Programms: statische Datenstrukturen und dynamischer Speicher, z.B.: *Heap*
- *Stack*: Unterprogrammaufrufe und -Datenstrukturen
  → siehe Abschnitt *13.3 Funktionsaufrufe und Stack*
- Prozesskontrollblock, siehe Folie 1171
  - Identifier, Parent, Child-Liste
  - Register: für Benutzer sichtbar + „*Rename-Register*“
  - Status-Register: Programmzähler, Flags, Modus Interrupts Enabled ...
  - Stack-Pointer
  - Scheduling Information: Zustand des Prozessmodells, Priorität ...
  - Informationen für Interprozesskommunikation
  - Privilegien: Zugriffsrechte auf Speicherbereiche, I/O ...
  - Speicherverwaltung: Tabellen für *Virtual Memory*
  - aktuelle Ressourcen, z.B.: geöffnete Dateien
  - ...

Prozessabbild / *Process image* (cont.)



Prozessabbild / *Process image* (cont.)

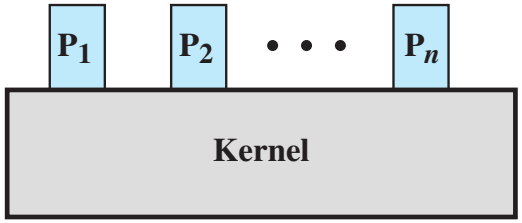


Kontextwechsel

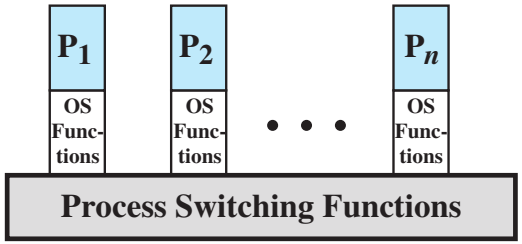
- Interrupt-Mechanismen
  - Interrupt: asynchrones, externes Ereignis
  - Trap: Fehler während der Programmabarbeitung
  - System-Call: Aufruf einer Betriebssystemfunktion
- kein „wartender“ Interrupt $\Rightarrow$ nächsten Befehl holen
- Interrupt löst Kontextwechsel aus
  - Programmzähler mit Interrupt Handler initialisieren
  - Wechsel *User Mode* $\Rightarrow$ *Kernel Mode* für privilegierte Instruktionen
- Kontextwechsel
  - Kontext des Prozesses sichern
  - Prozesskontrollblock aktualisieren
  - –"– in „passende“ Warteschlange einfügen
  - anderen Prozess zur Ausführung wählen
  - dessen Prozesskontrollblock aktualisieren
  - Speicherstrukturen für neuen Prozess aktualisieren (Seitentabelle)
  - Kontext des neuen Prozesses einrichten

Kontextwechsel (cont.)

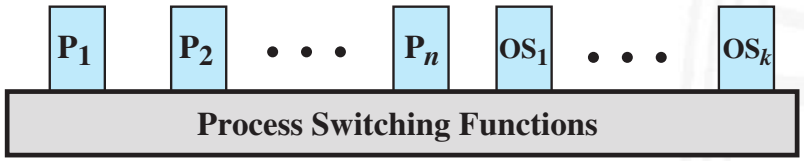
- Realisierungen
  - getrennter Betriebssystemkernel

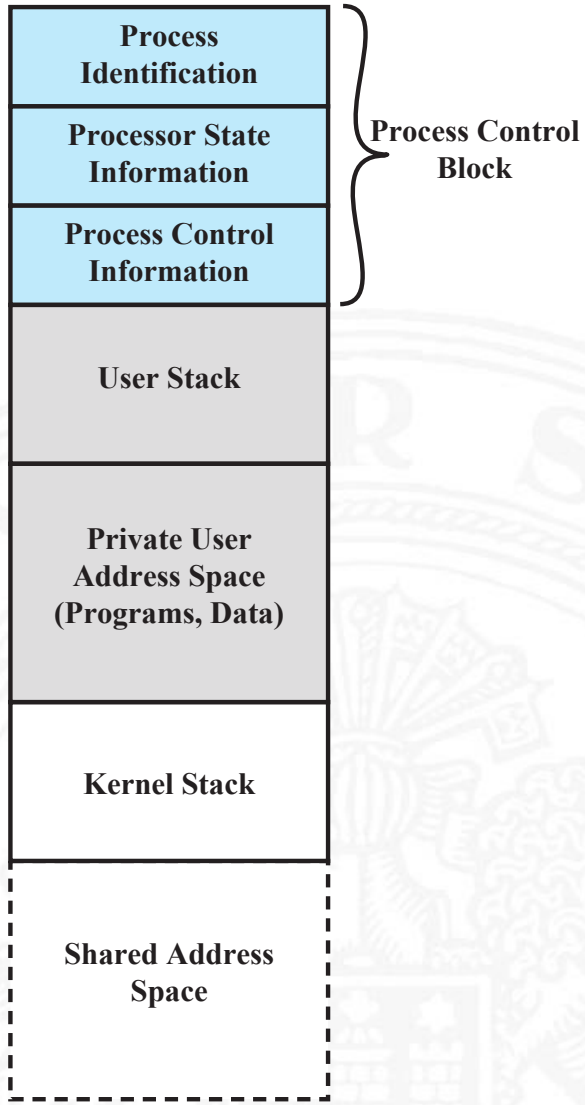


  - innerhalb des Benutzerprogramms



  - eigene Services/Prozesse (*Microkernel*)





Prozesse und Threads

- *Thread / Lightweight Process*
  - Betriebssystem: Zuordnung zu Prozessor (CPU)

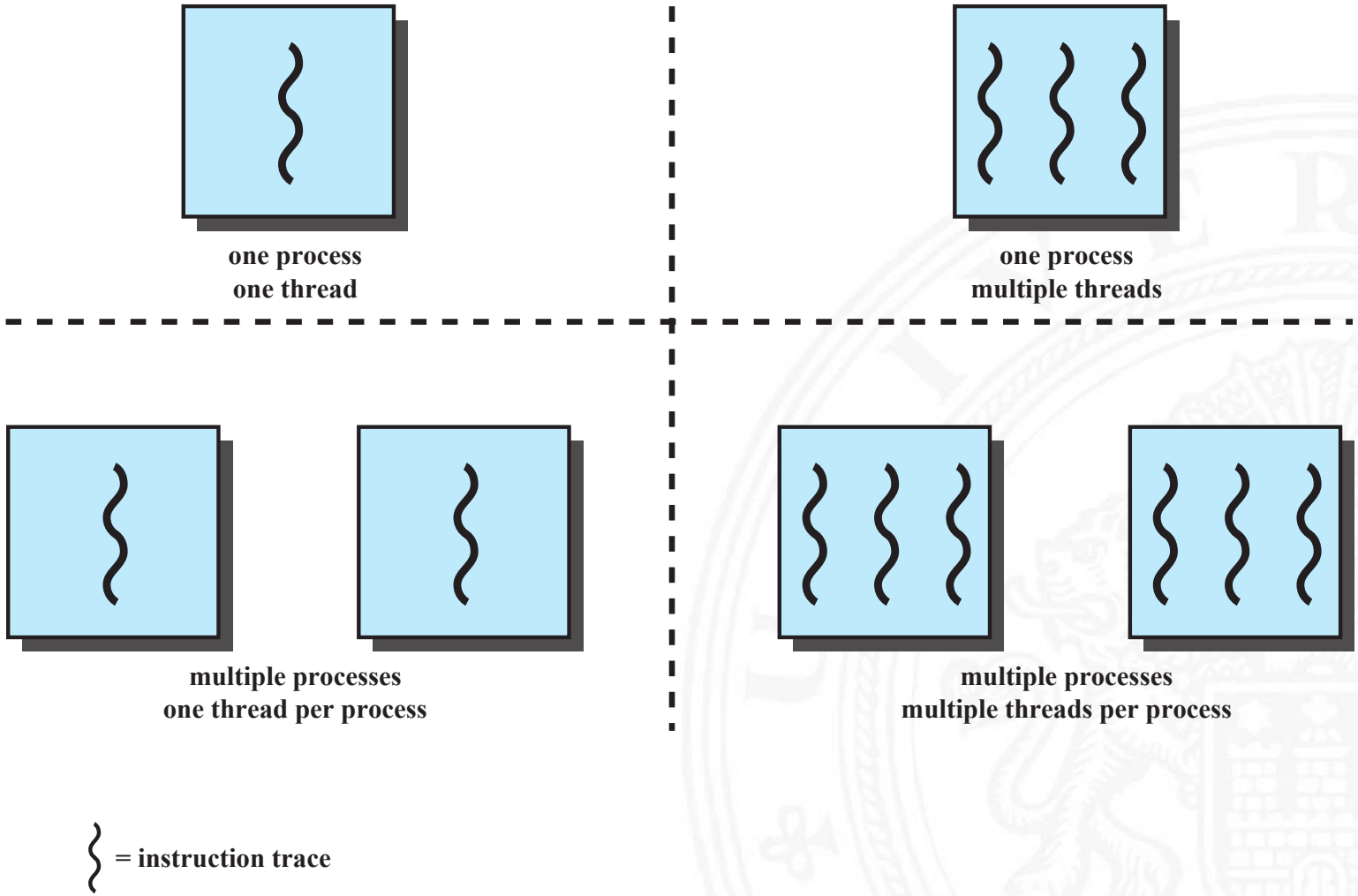
  $\Rightarrow$ Programmablauf (*Scheduling*, *Dispatching*)

- Prozess
  - Betriebssystem: Zuordnung zu Ressourcen (Speicher, Dateien, I/O ...)

  $\Rightarrow$ gesamter Kontrollblock

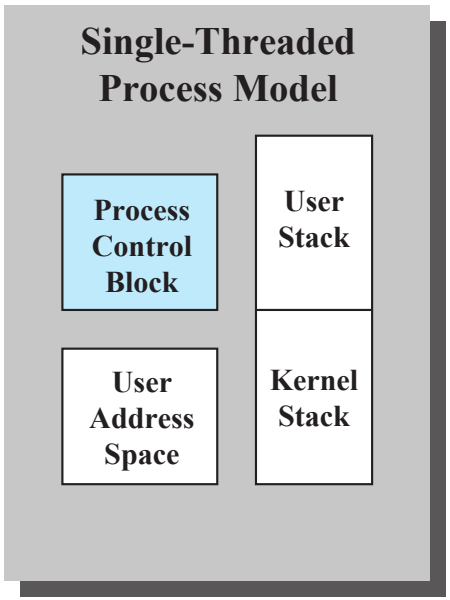
Prozesse und Threads (cont.)

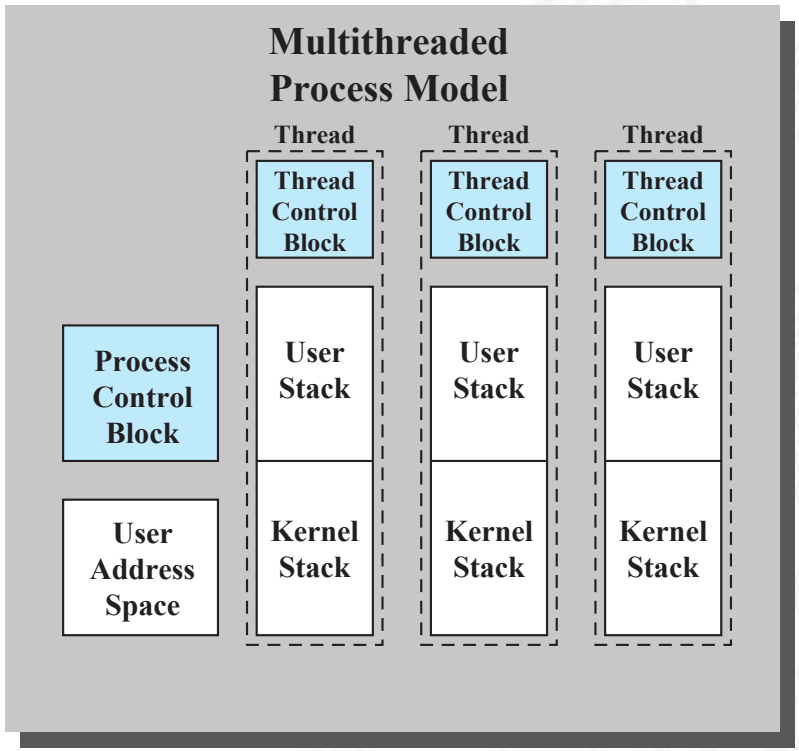
- *Multithreading*
  - mehrere parallele Ausführungen innerhalb eines Prozesses
  - von Programmiersprache und Betriebssystem abhängig



Threads

- eigener Zustand ($\widehat{=}$ Prozesszustand)
- eigener Kontext
- eigener Stack
- ggf. eigenen, statischen (Variablen-) Speicher
- Zugriff aller Datenstrukturen und Ressourcen des Prozesses
- geteilt mit allen anderen Threads



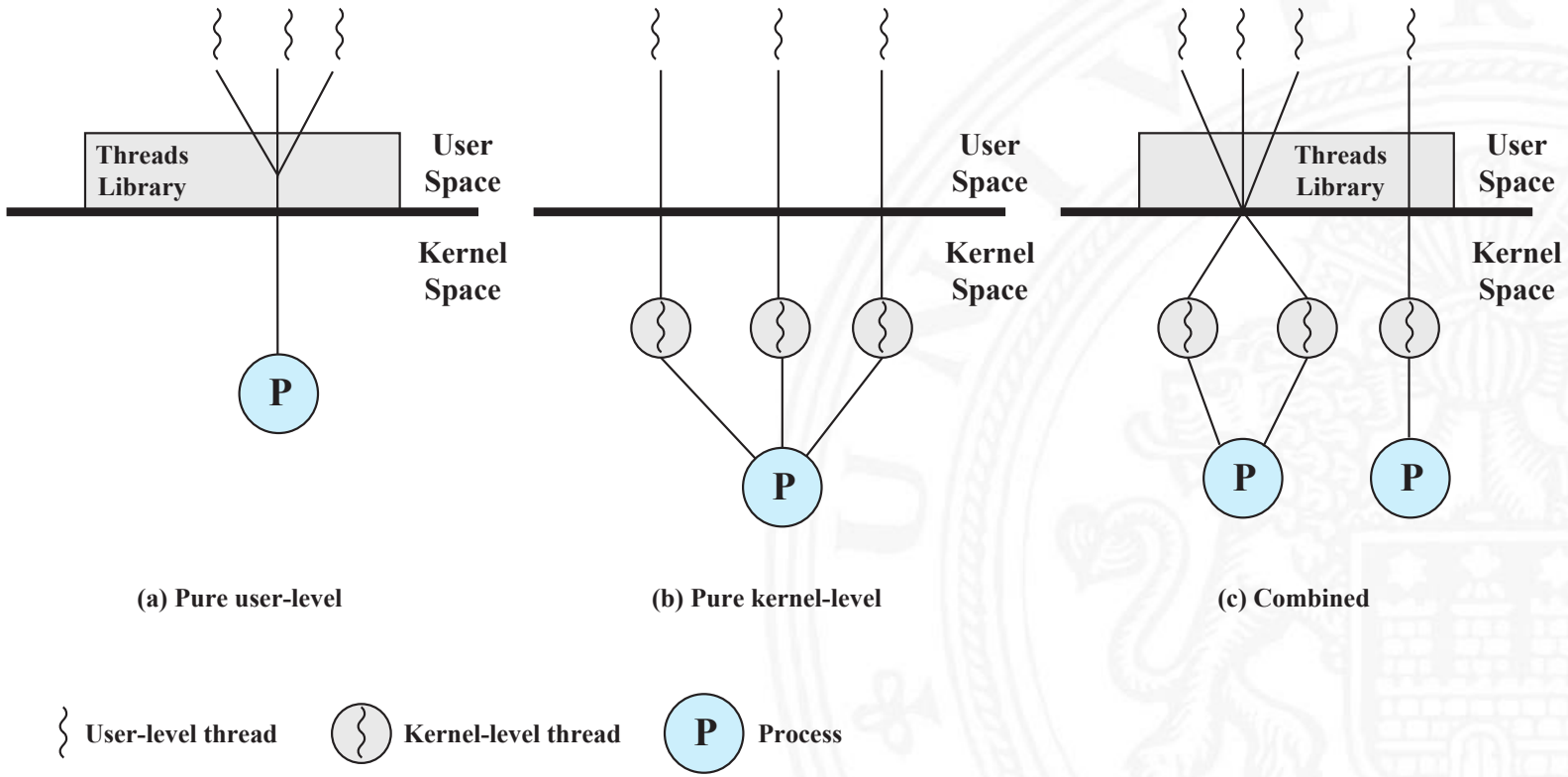


Threads (cont.)

- Vor- und Nachteile
  - \+ einfacher zu verwalten / erzeugen
  - \+ schneller zu beenden
  - \+ Wechsel zwischen Threads schneller als Prozess-Kontextwechsel
  - \+ effizientere Kommunikation
  - \+ effektiv, wenn auf I/O gewartet wird, z.B.: Serverdienste, RPC (Remote procedure calls), Browser-Tabs ...
  - \+ Parallelität ausnutzen
  - – Synchronisation wichtig! alle Threads arbeiten im gleichen Adressraum
- Arten von Threads
  - User Level Thread (ULT), eigene Bibliotheken
    - \+ Thread-Wechsel ohne Kernel Privilegien
    - \+ spezifisches (eigenes!) Scheduling
    - \+ läuft auf allen Betriebssystemen
    - – für Kernel nur ein Ablauf → keine Parallelität, System-Call blockiert alles

Threads (cont.)

- Kernel Level Thread (KLT)
  - \+ mehrere Threads in Multiprozessorumgebung
  - \+ Prozess (andere Threads) kann trotz blockiertem Thread weiterlaufen
  - \+ Betriebssystem selbst kann Multithreaded sein
  - – Wechsel zwischen Threads eines Prozesses bedingt Moduswechsel
- Mischformen



Nebenläufigkeit

- nebenläufige Prozesse und Threads
  - Multiprogramming: viele Prozesse, ein Prozessor
  - Multiprocessing: viele Prozesse, mehrere Prozessoren
  - verteiltes Rechnen

$\Rightarrow$ abwechselndes und überlapptes Rechnen

$\Rightarrow$ Timing / Abarbeitungsgeschwindigkeit nicht vorhersehbar

  - Aktivitäten anderer Prozesse oder der Benutzer
  - Interrupts
  - Scheduling durch Betriebssystem

- Begriffe
  - **atomare Operation**: Funktion oder Aktion; kann nicht unterteilt/unterbrochen werden, auch wenn sie aus mehreren Schritten besteht. Wird komplett oder gar nicht wirksam. Zentraler Mechanismus, zur Trennung nebenläufiger Prozesse.
  - **Critical Section / kritische Sektion**: Codebereiche mehrerer Prozesse, in denen auf gemeinsame Ressourcen (z.B. Speicher) zugegriffen wird.

Nebenläufigkeit (cont.)

- **Deadlock**: zwei oder mehr Prozesse können nicht weiterarbeiten, da sie gegenseitig aufeinander warten.
- **Livelock**: zwei oder mehr Prozesse wechseln ständig ihre Zustände durch Aktivitäten jeweils anderer Prozesse, ohne Fortschritte in der Bearbeitung.
- **Mutual Exclusion / gegenseitiger Ausschluss**: wenn ein Prozess in seiner Critical Section ist, kann kein zweiter Prozess in einer Critical Section sein, die die gleichen Ressourcen nutzt.
- **Race Condition**: mehrere Threads/Prozesse lesen und schreiben Daten, wobei das Ergebnis von deren zeitlicher Reihenfolge abhängig ist.
- **Starvation / „verhungern“**: ein lauffähiger Prozess könnte (weiter-) arbeiten, wird aber nie bedient.

- Kommunikationsmechanismen zwischen Prozessen/Threads
  - gemeinsamer Speicher (*Shared Memory*) $\Rightarrow$ Mutual Exclusion
  - Nachrichtenaustausch

Mutual Exclusion / gegenseitiger Ausschluss

- notwendig, um Race Conditions zu vermeiden
- durch Sicherung von Critical Sections
- mögliche Probleme: Deadlock, Starvation

- Uniprozessor: keine Interrupts in Critical Section
- atomare Hardwareoperationen („*compare & swap*“)
  - \+ gilt für: Uni-/Multiprozessor, beliebige Anz. Prozesse
  - \+ einfach zu verifizieren
  - \+ für beliebige Anzahl kritischer Sektionen
  - – *Busy-waiting*: Prozessor arbeitet immer
  - – Starvation möglich (wenn mehrere Prozesse warten)
  - – Deadlock möglich

Mutual Exclusion / gegenseitiger Ausschluss (cont.)

- Software Mechanismen
  - Implementierung in Software nicht trivial!!!
    (Dekker-Algorithmus; E. W. Dijkstra; Peterson-Algorithmus)
  - **Semaphor**: Integer Variable, für die drei atomare Operationen möglich sind: initialisieren, increment, decrement
  - **Mutex / binärer Semaphor**: Werte 0 und 1
  - **Monitor**: Programmiersprachen Konzept, das Variablen und Zugriffsprozeduren als Datentyp kapselt. Immer nur ein Prozess hat Zugriff darauf.

Semaphor

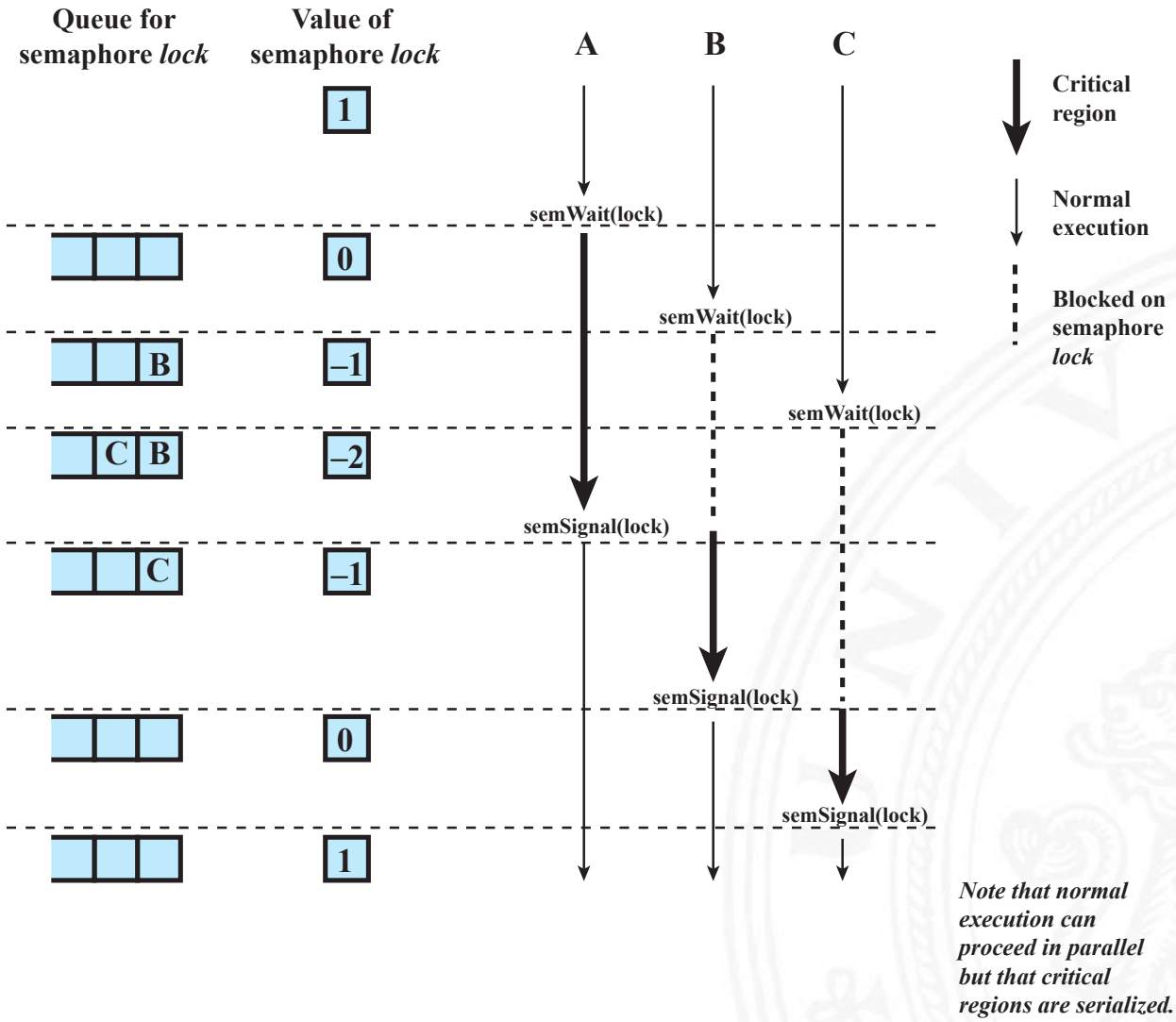
- Integer Variable, initialisiert mit Anz. gleichzeitiger Zugriffe
- atomare Operationen

  semWait decrement + aufrufender Prozess muss ggf. warten
  $\Rightarrow$ Beginn der Critical Section

  semSignal increment + ein wartender Prozess kann starten
  $\Rightarrow$ Ende der Critical Section
- starke Semaphor: am längsten wartender Prozess wird gestartet
  $\Rightarrow$ Queue für wartende Prozesse
- schwache Semaphor: beliebiger, wartender Prozess wird gewählt

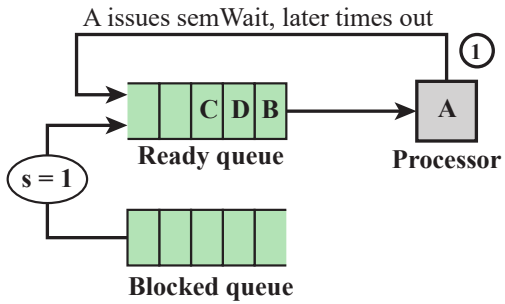
Semaphor (cont.)

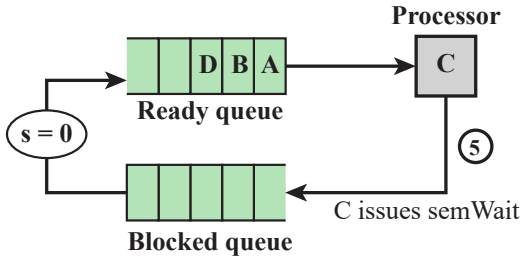
- Beispiele

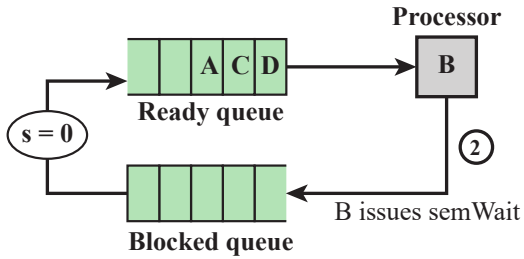


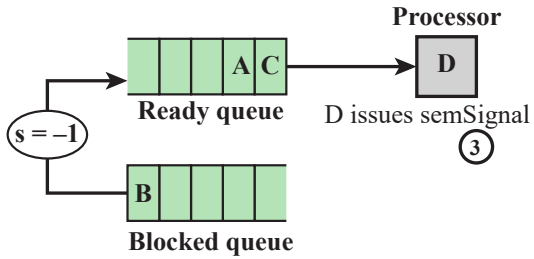
Note that normal execution can proceed in parallel but that critical regions are serialized.

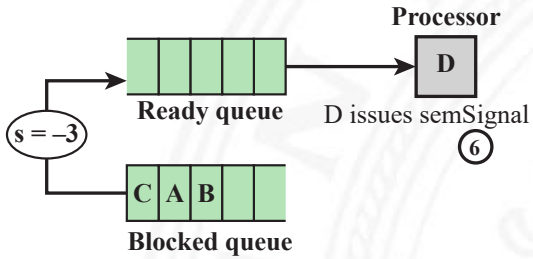
Semaphor (cont.)

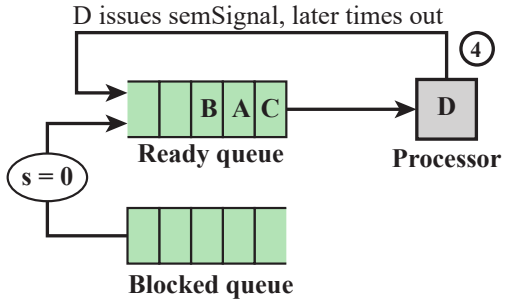


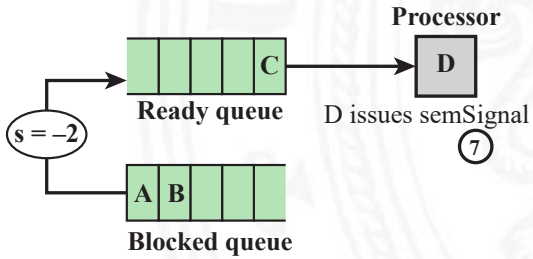








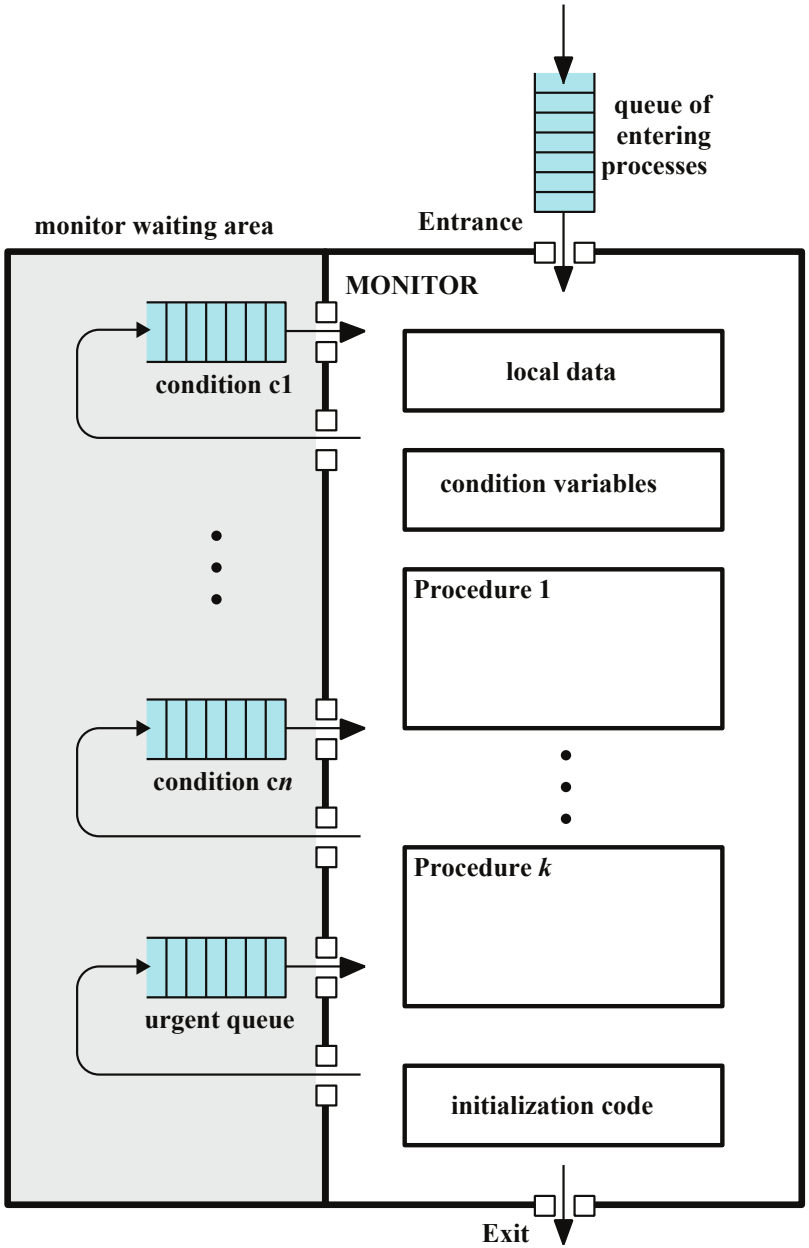




Monitor

- in Programmiersprachen: Concurrent Pascal, Ada, Modula ...
- in Java (`synchronized`), aber keine Bedingungsvariablen
- einfacher zu handhaben als Semaphor, gleiche Funktionalität
- nur ein Prozess darf im Monitor sein
- Synchronisation: Bedingungsvariablen (*condition variables*) und Funktionen `wait`, `signal`
- mehrere Warteschlangen: extern (Eintritt in Monitor), für jede Bedingungsvariable

Monitor (cont.)



Nachrichtenaustausch (*Message Passing*)

- geht auch für (räumlich) verteilte Systeme
- Kommunikationsfunktionen

  send (⟨*dst*⟩ , ⟨*data*⟩) sendet Daten
  blockierend / nicht blockierend

  receive (⟨*src*⟩ , ⟨*data*⟩) empfängt Daten
  blockierend / nicht blockierend / testend
- Varianten
  - block. `send` + block. `receive` ⇒ *Rendezvous*
    nicht block. `send` + block. `receive`
    nicht block. `send` + nicht block. `receive`
  - direkte Adressierung (s.o.) / indirekte Adressierung
    ⇒ 1:1, 1:n, m:1, m:n Beziehungen

```
process P is
  ...
  send(Q, A)
  ...
```

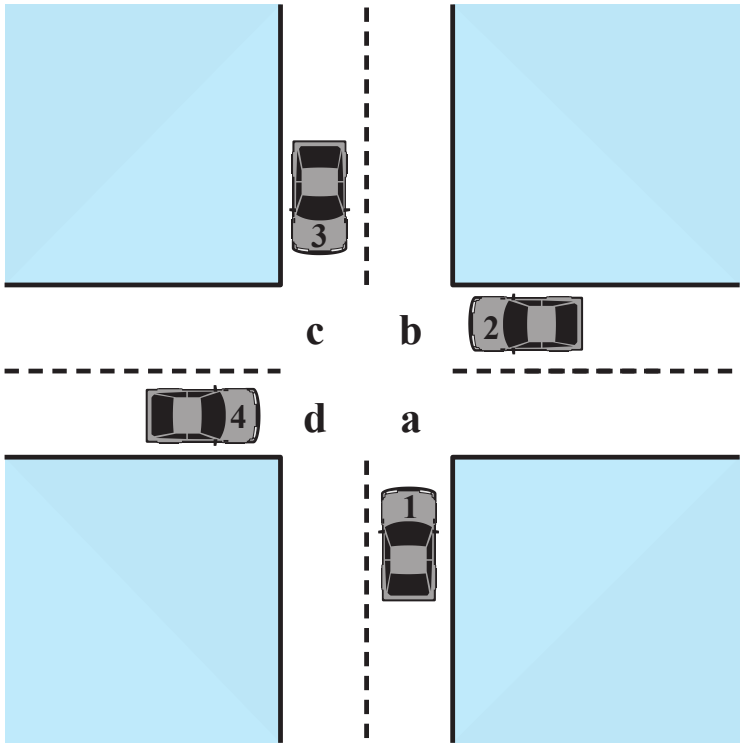
```
process Q is
  ...
  receive(P, A)
  ...
```

Deadlock

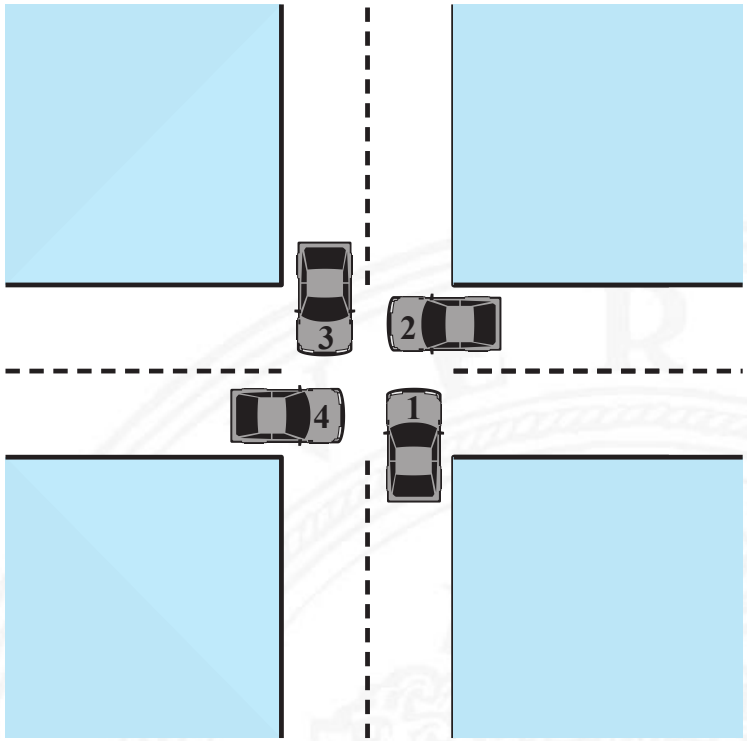
- Dauerhaftes Blockieren mehrerer Prozesse, die um Ressourcen konkurrieren, bzw. miteinander kommunizieren
- Deadlock, wenn jeder Prozess blockiert auf etwas wartet, was nur ein anderer blockierter Prozess anstoßen kann
- im Allgemeinen: keine effiziente Lösung

Deadlock (cont.)

- Beispiel: „rechts vor links“



(a) Deadlock possible



(b) Deadlock

- jedes Fahrzeug braucht 2 Ressourcen
  1: a,b 2: b,c 3: c,d 4: d,a

Deadlock (cont.)

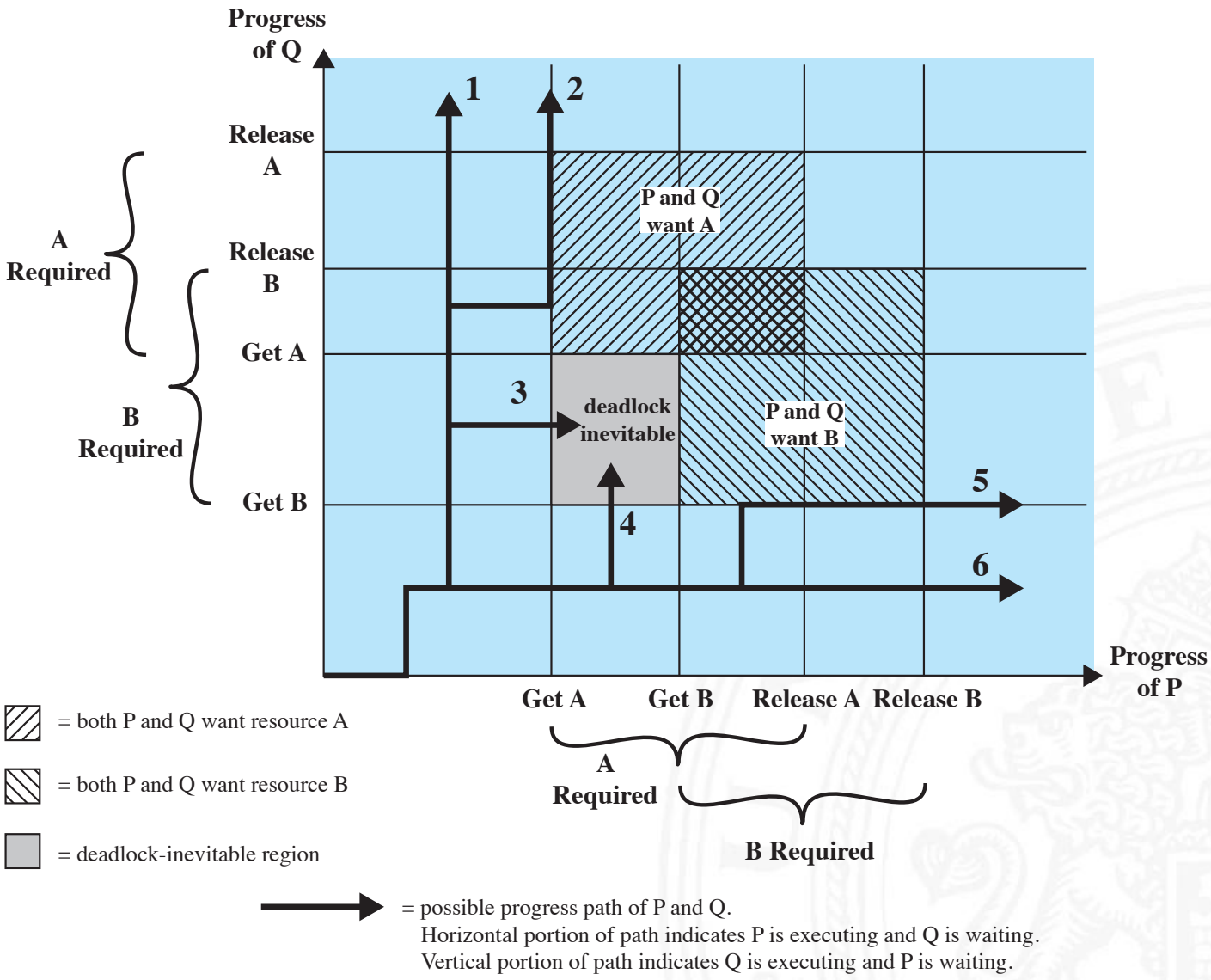
- Beispiel: zwei Programme, zwei Mutexe

```
process P is
  ...
  get(A)
  ...
  get(B)
  ...
  release(A)
  ...
  release(B)
  ...
```

```
process Q is
  ...
  get(B)
  ...
  get(A)
  ...
  release(B)
  ...
  release(A)
  ...
```

  - alternierender Ablauf der Prozesse
  - zweidimensional dargestellt

Deadlock (cont.)



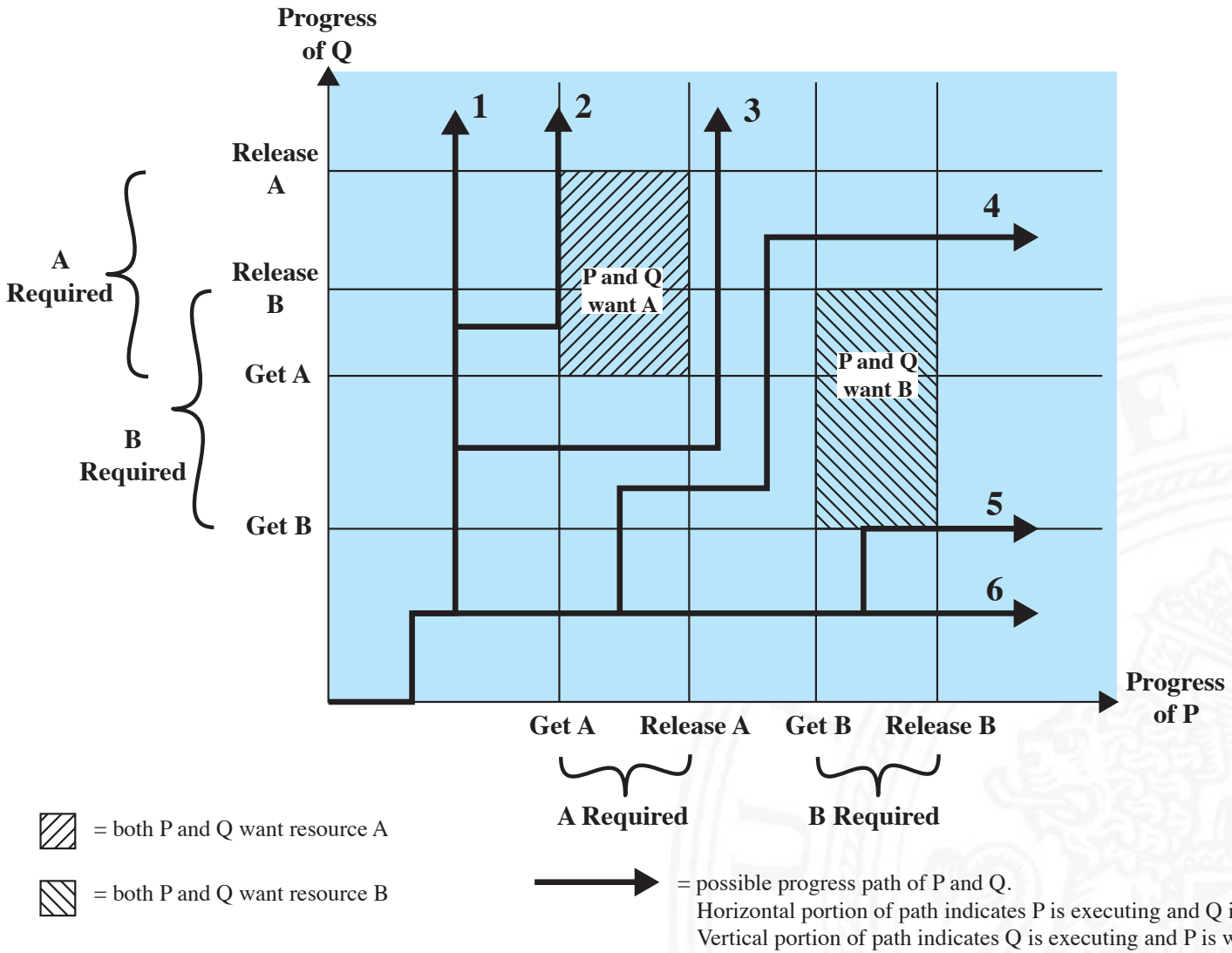
Deadlock (cont.)

- vorheriges Beispiel ohne Deadlock

```
process P is
  ...
  get(A)
  ...
  release(A)
  ...
  get(B)
  ...
  release(B)
  ...
```

```
process Q is
  ...
  get(B)
  ...
  get(A)
  ...
  release(B)
  ...
  release(A)
  ...
```

Deadlock (cont.)



Ressourcen

- wiederverwendbare (*reusable*) Ressource
  - wird bei Benutzung durch Prozess/Task nicht verbraucht
  - Prozessor, I/O-Kanal, Hauptspeicher, sekundärer Speicher, Datenstrukturen
- verbrauchbare (*consumable*) Ressource
  - wird erzeugt und bei Nutzung verbraucht
  - Interrupts, Signale, Nachrichten etc. (in FIFOs)
- Beispiel: wiederverwendbare Ressource = 200 KiB Speicher

```
process P is
  ...
  malloc(70 KiB)
  ...
  malloc(80 KiB)
  ...
```

```
process Q is
  ...
  malloc(80 KiB)
  ...
  malloc(60 KiB)
  ...
```

Ressourcen (cont.)

- Beispiel: verbrauchbare Ressource = Nachrichten,
  `receive` blockierend

```
process Q is
  ...
  receive(P, M1)
  ...
  send(M2, P)
  ...
```

```
process P is
  ...
  receive(Q, M3)
  ...
  send(M4, Q)
  ...
```

Bedingungen für Deadlock

1. Mutual Exclusion
   - ohne Mutual Exclusion kein Deadlock
     $\Rightarrow$ aber u.U. inkonsistente Daten
2. Hold-and-Wait
   - Prozess hat exklusiven Zugriff auf Ressource und fragt weitere an
3. No Preemption: Ressourcen können nicht entzogen werden
   - *Preemption* hier als zwangsweiser Entzug der Ressource
- Circular Wait: mehrere Prozesse/Tasks warten zyklisch aufeinander

$\Rightarrow$ 1. bis 3. notwendige Bedingungen
$+$ Circular Wait (zur Laufzeit) $=$ Deadlock

Maßnahmen gegen Deadlock

1. Deadlock verhindern
   - indirekt: drei notwendige Bedingungen für Deadlock
     - zu Mutual Exclusion: meist unverzichtbar
     - zu Hold-and-Wait: Prozess fordert gleichzeitig (atomar) alle Ressourcen/Locks an
     - zu No-Preemption: Test auf Ressource, wenn nicht verfügbar: kein Warten, sondern Rückgabe; Betriebssystem „entzieht“ Ressource
   - direkt: *Circular Wait* nicht zulassen
     - Einführen einer Ordnung/Reihenfolge für alle Ressourcen
     - muss in allen Prozessen eingehalten werden
2. Deadlock vermeiden
   - Ressource nicht zuteilen, wenn Deadlock möglich
     $\Rightarrow$ algorithmisch lösbar (*Banker's algorithm*)
   - Prozess nicht starten, der zu Deadlock führen kann

   \+ weniger Restriktiv als „Deadlock verhindern“

   \+ kein Rollback nötig, wie in „Deadlock Erkennung“

Maßnahmen gegen Deadlock (cont.)

3. Deadlock Erkennung
   - ▸ Periodischer Test auf Deadlock und ggf. (partielles) Rücksetzen
   - \+ 1. und 2. schränken Prozesse ein;
     gegenteiliger Ansatz: Zugriff erlauben
   - \+ einfacher Algorithmus
   - – Overhead durch periodische Checks
   - – „Zurücksetzen“ der Prozesse nicht trivial; Checkpoints

⇒ Einteilung der Ressourcen in „Klassen“ mit verschiedenen „Arten/Typen“ von Deadlocks und Einsatz unterschiedlicher Deadlock Strategien

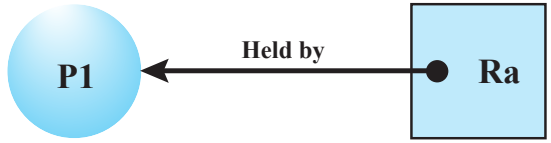
Maßnahmen gegen Deadlock (cont.)

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

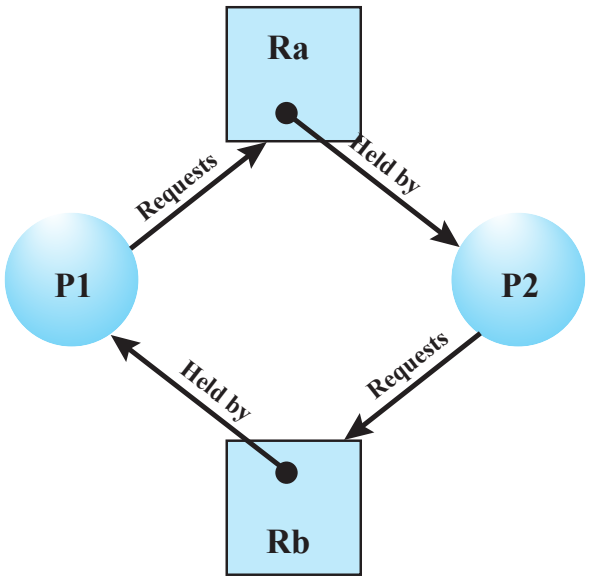
Maßnahmen gegen Deadlock (cont.)

- Graph zum Ressourcenbesitz

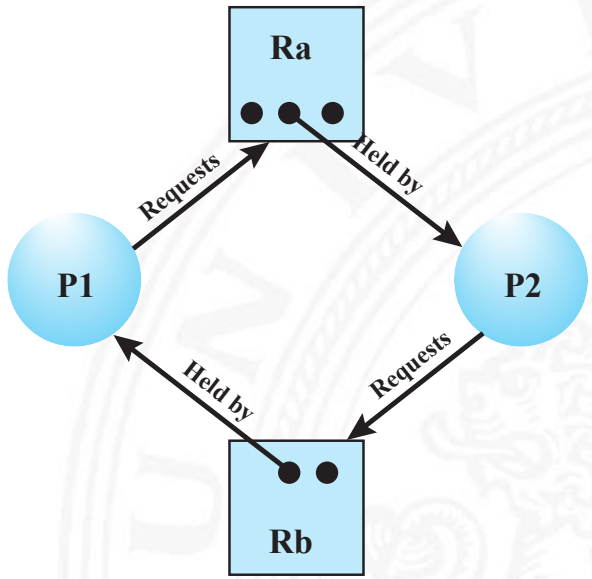
(a) Resouce is requested



(b) Resource is held

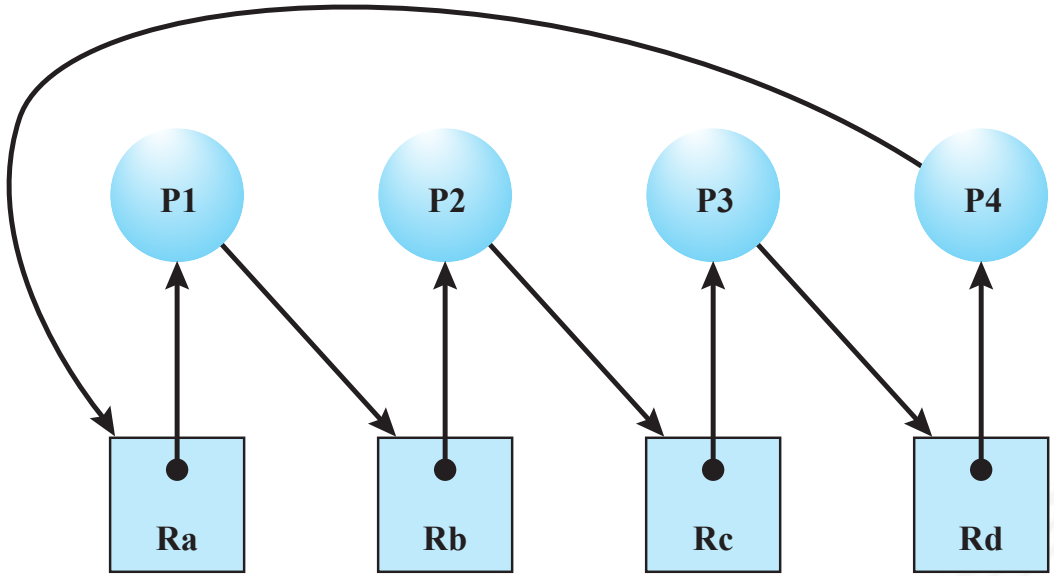


(c) Circular wait



(d) No deadlock

Maßnahmen gegen Deadlock (cont.)



Kreuzung: „rechts vor links“

- Algorithmen zu Deadlock Vermeidung oder Erkennung nutzen daraus abgeleitete Matrizen zu: Ressourcenanfragen und -besitz

Scheduling und Ressourcenmanagement

- Hauptfunktionalität von Betriebssystemen: Ressourcenmanagement
- wichtig dabei Effizienz
  Antwortverhalten (*Responsiveness*)
  Fairness

$\Rightarrow$ Scheduling / Ablaufplanung

- betrifft mehrere Ressourcen: Prozessor, Speicher, I/O Geräte

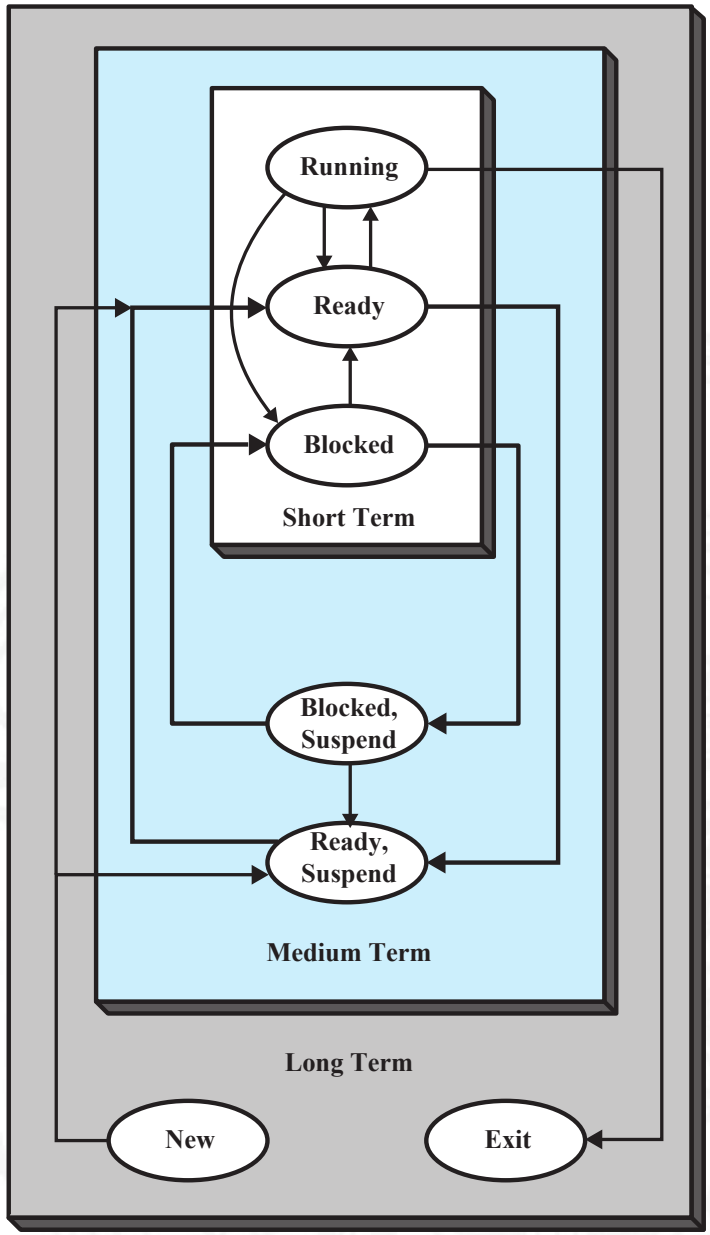
- **Long-term**: Welche Prozesse sollen in Menge der Jobs?
  - beeinflusst Multiprogramming: Anzahl der Jobs auf Computer
  - Strategien: First-come, First-served; nach Prioritäten; Ressourcen
- **Medium-term**: Welche Prozesse sollen in Hauptspeicher?
  - Teil der Speicherverwaltung $\rightarrow$ Abschnitt *15.6 Speicherverwaltung*
  - Auswirkungen auf Multiprogramming: Prozesse nicht lauffähig, wenn nicht im Speicher

Scheduling und Ressourcenmanagement (cont.)

- **Short-term**: Welcher Prozess wird durch CPU ausgeführt?
  - Dispatcher: wird häufig aufgerufen
  - Start durch: Interrupts, System-Calls, Signale (Semaphor, Mutex)
  - quantitative Kriterien
    Benutzer: Antwortverhalten (*Responsiveness*)
    System: Prozessornutzung, Job-Durchsatz, Ressourcenauslastung
  - qualitative Kriterien Fairness, Deadlockfrei, keine Starvation, Vorhersagbarkeit, Echtzeitfähigkeit etc.
- **I/O Scheduler**: Welche I/O-Anfrage geht an Gerät?
  - mehrere
  - gerätespezifisch

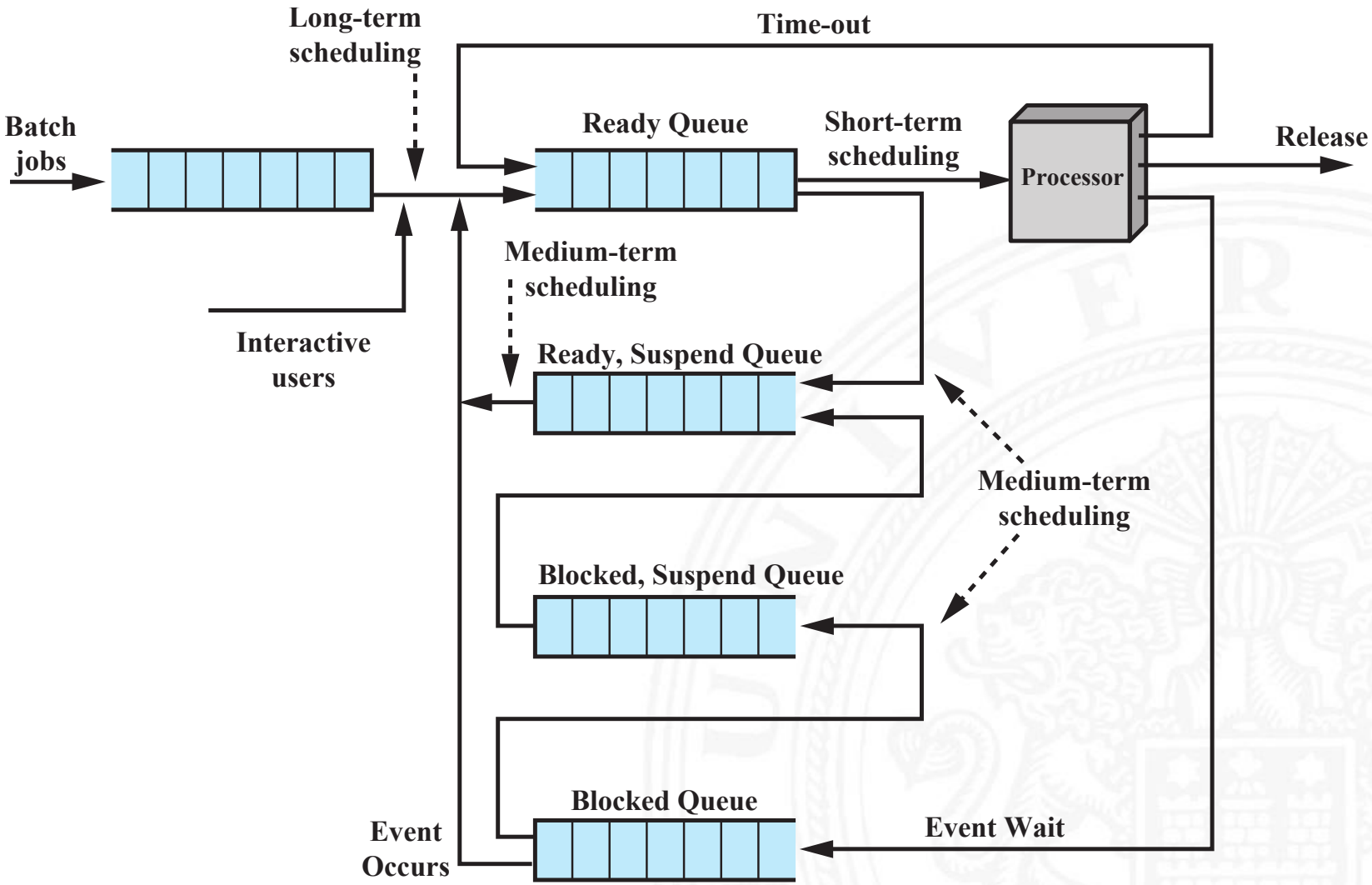
Scheduling und Ressourcenmanagement (cont.)

- verschiedene Zustände im Prozessmodell (vergl. Folie 1177)



Scheduling und Ressourcenmanagement (cont.)

- Scheduling Queues



Short-term Scheduling

- Aktivierung
  - neuer Prozess in *Ready* Queue
  - Interrupt (bringt Prozess in *Ready*)
  - periodisch
- Funktion zur Auswahl der Prozesse abhängig von
  - Prioritäten
  - Ressourcenbedarf
  - Prozessabarbeitung

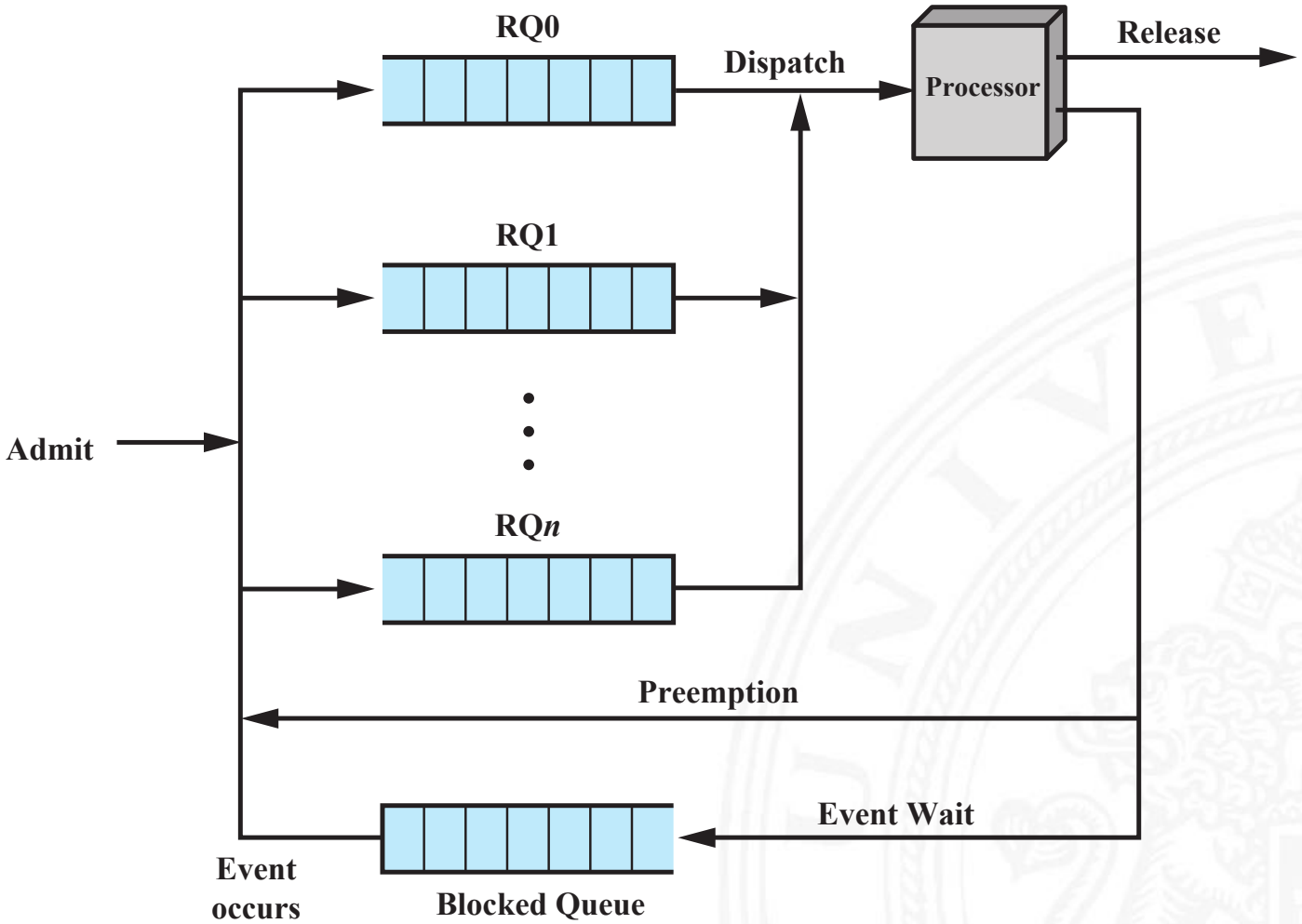
    w : bisherige Wartezeit

    e : bisherige Ausführungszeit (*Execution time*)

    s : gesamte Ausführungszeit (*Service time*)
- Preemption: Unterbrechung von Jobs?
  - **ohne**: gestarteter Prozess läuft bis Ende oder I/O waiting
  - **mit**: Prozess wird unterbrochen und in *Ready*-Queue eingereiht

Short-term Scheduling (cont.)

- Short-term Queues mit Prioritäten



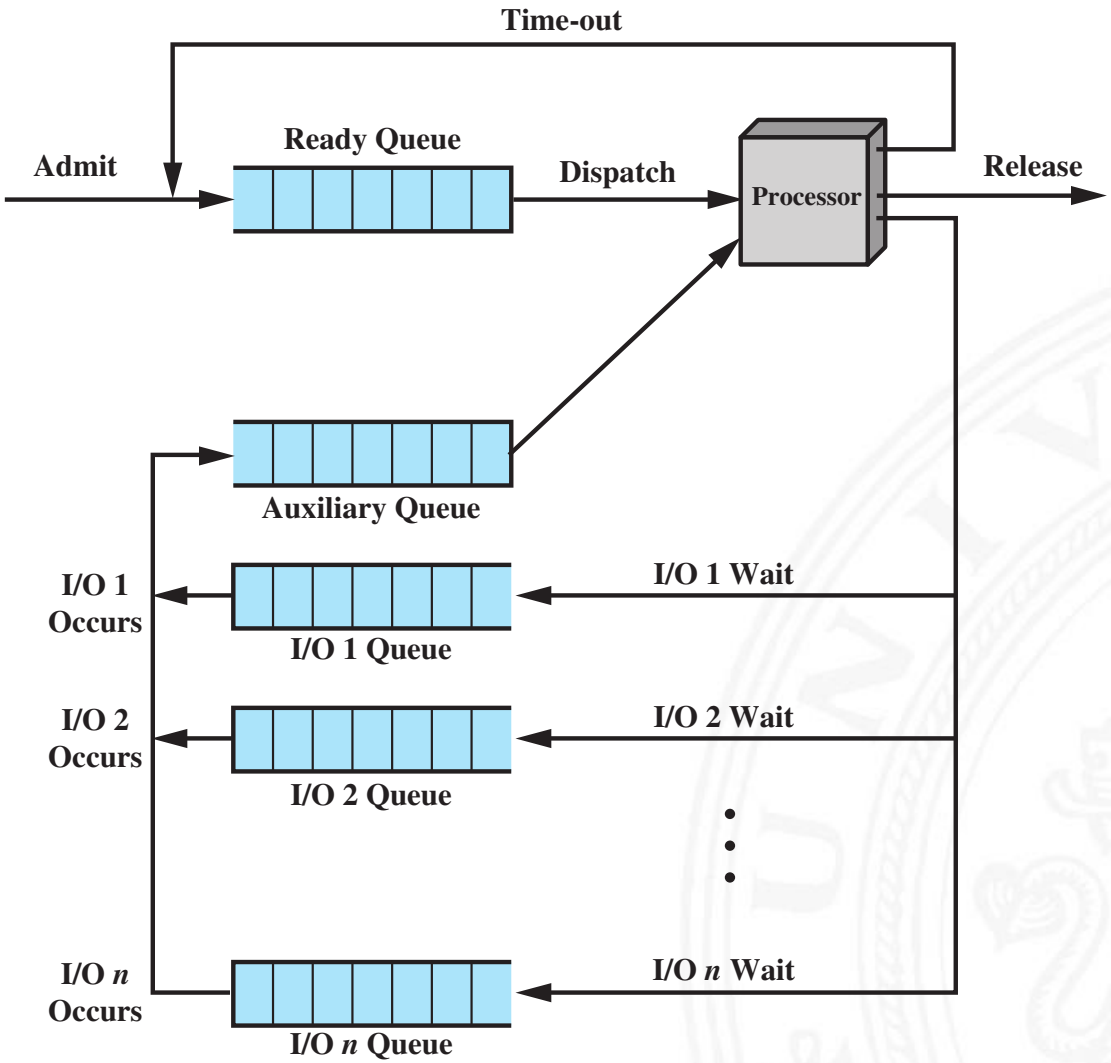
Scheduling Algorithmen

- **First-come, First-served** (FCFS)
  - Non-Preemptive
  - einfache Implementation: FIFO
  - bevorzugt länger laufende Prozesse
  - bevorzugt rechenlastige Prozesse gegenüber I/O-lastigen

- **Round-Robin** (RR) – Zeitscheiben Verfahren
  - Preemptive
  - Länge des Zeitslots?
  - Gut für Transaction Processing (Durchsatz, Fairness)

Scheduling Algorithmen (cont.)

- Round Robin Queues

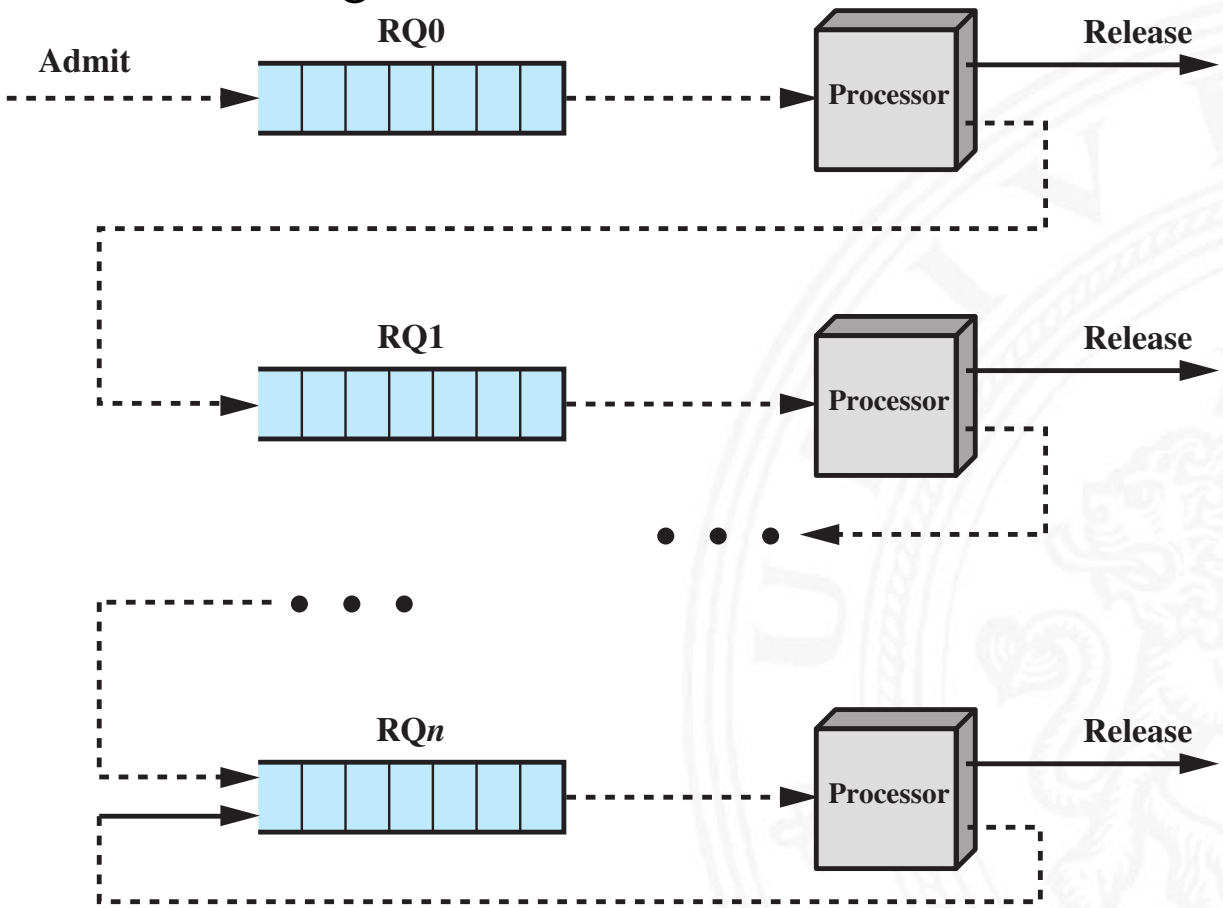


Scheduling Algorithmen (cont.)

- **Shortest Process Next** (SPN)
  - Non-Preemptive
  - Prozess mit kürzester Ausführungszeit startet
  - Ausführungszeit s schätzen?
  - Starvation für lang laufende Prozesse möglich
  - Interaktive Prozesse: Durchschnittsbildung der letzten Aktivitäten, ggf. „exponentielles Altern“ (= Wichtung älterer Werte nimmt ab)
- **Shortest Remaining Time** (SRT)
  - Preemptive Version von SPN
  - Prozess mit kürzester Restzeit startet
  - Ausführungszeit s schätzen?
  - Starvation für lang laufende Prozesse möglich
- **Highest Response Ratio Next** (HRRN)
  - Non-Preemptive
  - Response Ratio: $r = \frac{(w+s)}{s}$
  - Prozess mit größtem r startet
  - Fair, auch für lang laufende Prozesse wegen w

Scheduling Algorithmen (cont.)

- **Feedback Scheduling**
  - Preemptive
  - Round-Robin mit mehreren Queues: Start in Hochpriorisierter, sukzessiver Abstieg in weniger priorisierte Queues
  - relativer Vorzug kurz laufender Prozesse



Scheduling Algorithmen (cont.)

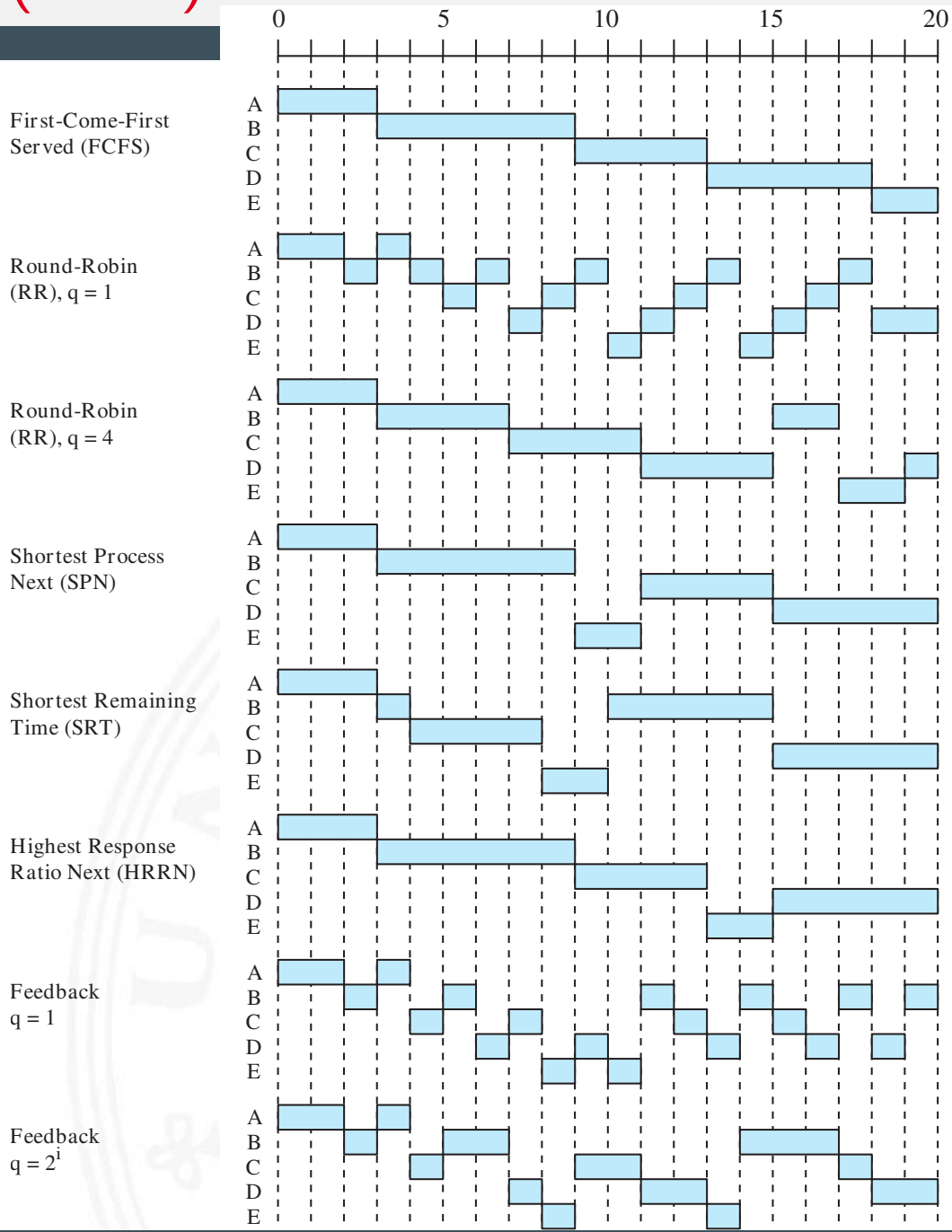
- ... viele weitere Algorithmen, es fehlen
  - Gruppierung von Prozessen (*Process Groups*)
  - periodische Tasks
  - Echtzeitsysteme: Prozesse haben eine Deadline!

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s – e]		(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Through-Put	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

Scheduling Algorithmen (cont.)

- Beispiel

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



Scheduling Algorithmen (cont.)

- Performanz des Scheduling? – Bewertung?
  - Rechnersystem: Desktop, Kontroll-/Steuerungsrechner, Server (DB, Web-Dienste . . .), HPC (Großrechner, Supercomputer)
  - Anwendungsszenarien: welche, wie viele Prozesse?
  - I/O: welche Geräte, wie schnell?
  - Aufwand und Effizienz des Scheduling
  - Aufwand für Kontextwechsel

$\Rightarrow$ Modellierung über Warteschlangentheorie, stochastische Prozesse

Multiprozessor Scheduling

- unterschiedliche Kopplungen:
  Cluster (schwach) ... Symmetrical Multiprocessing (stark)
- Scheduling auch für Spezial- (Co-) Prozessoren
- Granularität, wichtig für Synchronisation
  - Parallelität inhärent in Thread
  - Parallelität innerhalb eines Prozesses / Multithreading
  - kommunizierende Prozesse (Shared-Memory)
  - verteilte Prozesse in Netzwerk (Message-Passing)
  - Menge unabhängiger Prozesse

Multiprozessor Scheduling (cont.)

- Zuordnung von Prozessen zu Prozessoren
  - dynamisch: Menge von Prozessen $\rightarrow$ Pool von Prozessoren
  - statisch: Prozess wird Prozessor zugeordnet
    - \+ Scheduling einfacher
    - \+ Group-Scheduling
    - – ggf. Prozessorleerlauf (dann Load-Balancing)
- Architekturen (Wo läuft der Scheduler?)
  - Peer Systeme / verteiltes Scheduling
  - Master-Slave
    - \+ einfach zu implementieren
    - \+ weniger Overhead
    - – Point of Failure
    - – Bottleneck

Aufgaben des Betriebssystems

- Trennung der Prozesse voneinander
- Verwaltung von dynamischem Speicher
- Unterstützung modularer Programme
- Schutz: Integrität der Daten
- Schutz: Durchsetzung von Zugriffsrechten
- Realisierung von Langzeitspeicher (Dateisysteme)

Virtueller Speicher

- sekundärer Speicher (HDD, SSD) ist Teil des Speichers
- logische Adressen in Programmen sind unabhängig von
  - dem physikalisch vorhandenem Speicher
  - physikalischen Adressen (Adressen zur Laufzeit)
- mehrere Prozesse, Benutzerjobs. . . sind gleichzeitig im Speicher

$\Rightarrow$ Adressen im Code werden zu **virtuellen Adressen**:
logische Adresse + Adressübersetzung

- Adressübersetzung entspricht Funktion
- meist als Tabelle realisiert

Virtueller Speicher (cont.)

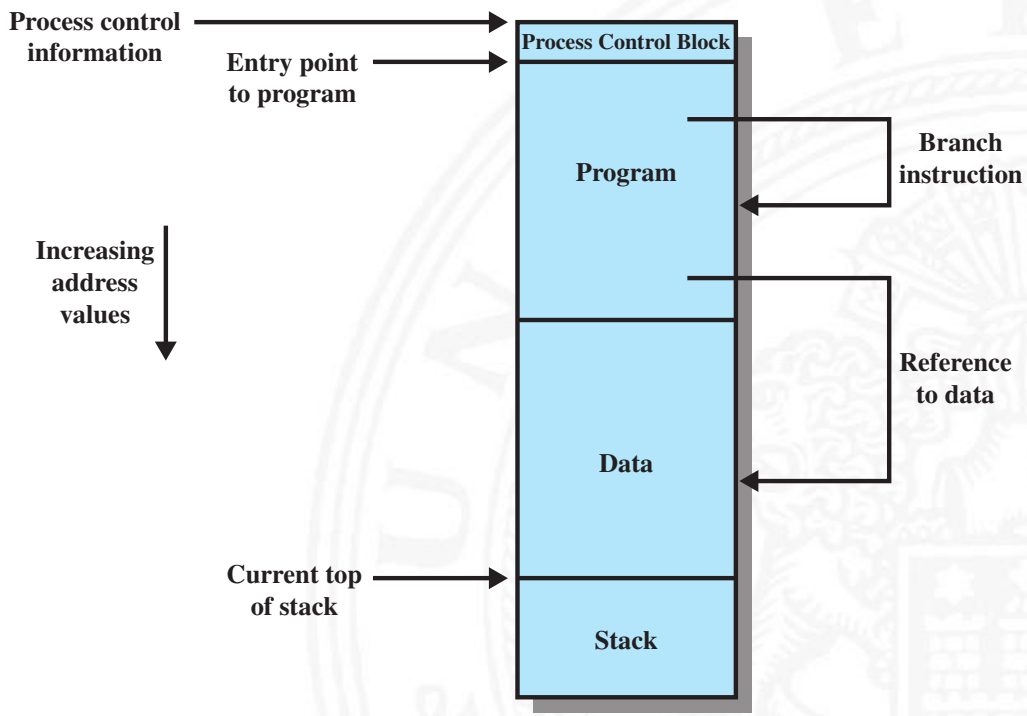
- **Frame / Kachel**
  - Block fester Größe im Hauptspeicher
- **Page / Seite**
  - Block fester Größe im sekundären Speicher (HDD, SSD),
  - kann temporär in Frame (im Hauptspeicher) kopiert werden

  $\Rightarrow$ Paging
- **Segment**
  - Block variabler Größe im sekundären Speicher, kann temporär
  - in passenden Bereich im Hauptspeicher kopiert werden

  $\Rightarrow$ Segmentierung
  - in Seiten unterteilt werden, die jeweils kopiert werden

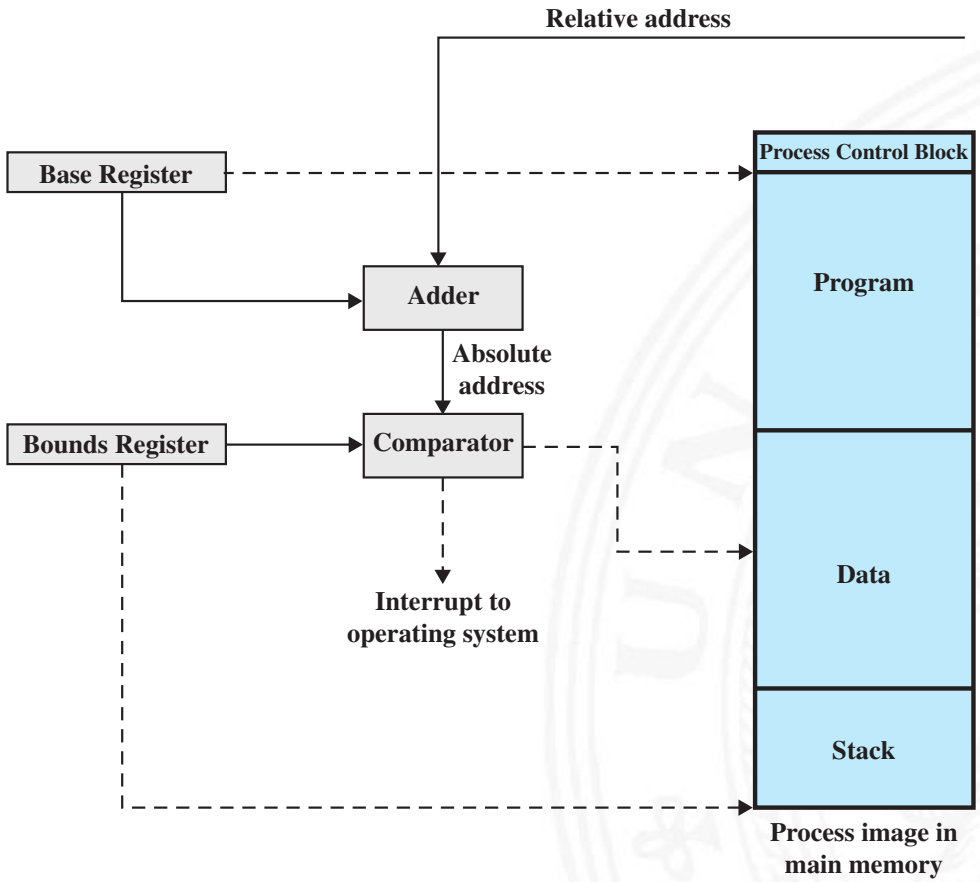
  $\Rightarrow$ Segmentierung + Paging

- Adressumsetzung / *Relocation*
  - **logische Adressen**: Adressen in (Assembler-) Programm
  - **relative Adressen**:
    relativ zu Bezug (Basisadresse), i.d.R. logische Adressen
  - **physikalische / absolute Adressen**:
    Adressen des Hauptspeichers



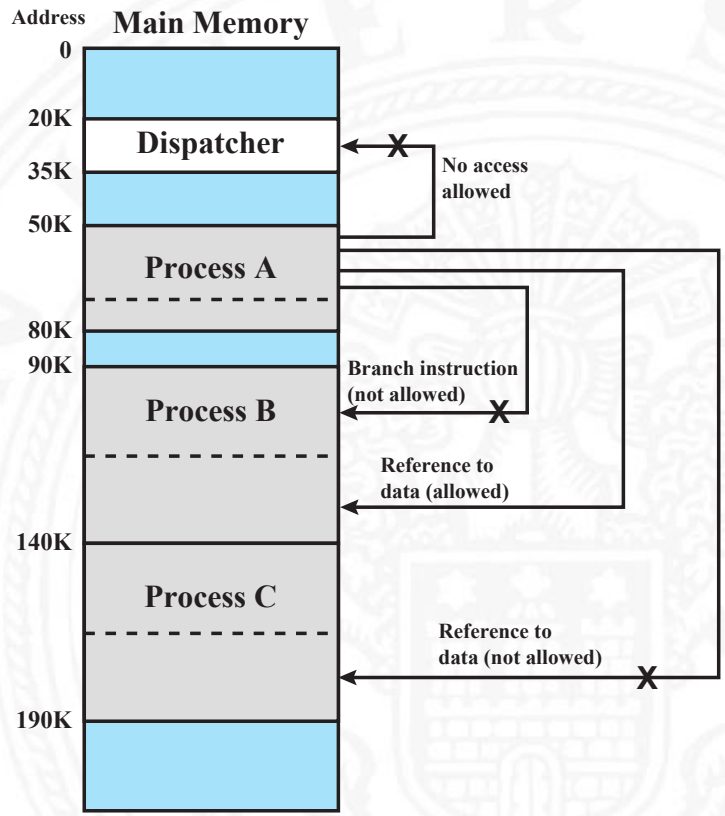
Memory Management (cont.)

- Swapping: Prozess auslagern, u.U. an anderer Stelle in Hauptspeicher fortsetzen
- Abbildung: logische $\rightarrow$ physikalische Adressen



Memory Management (cont.)

- Schutz, hier Zugriff
  - Relocation verwaltet auch Zugriffsrechte auf Speicherbereiche
  - bei Adressumrechnung in Segment- und Seiten-Tabellen
- gemeinsam genutzte Code- und Datenbereiche
  - Relocation ermöglicht auch Einbindung von Speicherbereichen in Adressraum mehrerer Prozesse
  - gemeinsam genutzte Segmente



Memory Management (cont.)

- Trennung von logischer und physischer Organisation
  1. Segmentierung
  2. Paging / Seitenadressierung
  3. Kombination von: Segmentierung und Paging
- bei Programmlauf: nicht alle Segmente/Seiten des Prozesses müssen gleichzeitig in Hauptspeicher sein
  - **Resident Set**: Adressbereiche (Text, Data) in Hauptspeicher
  - **Working Set**: im Programm gerade genutzt (Lokalität)
- während Programmlauf
  1. Interrupt, wenn Adresse des Prozesses nicht in Hauptspeicher
  2. Prozess wechselt in „blocked“
  3. Datentransfer: sekundärer Speicher $\rightarrow$ Hauptspeicher durch DMA (im Hintergrund)
  4. Dispatcher lässt anderer Prozess rechnen
  5. Interrupt, wenn Datentransfer fertig
  6. Prozess wechselt in „ready“

Memory Management (cont.)

+ kleinerer Adressraum je Prozess $\Rightarrow$ mehr Prozesse im System
  $\Rightarrow$ bessere CPU-Auslastung
+ Prozesse können größer sein als gesamter Hauptspeicher
+ für Programmierer transparent,
  von Betriebssystem und HW verwaltet

Memory Management (cont.)

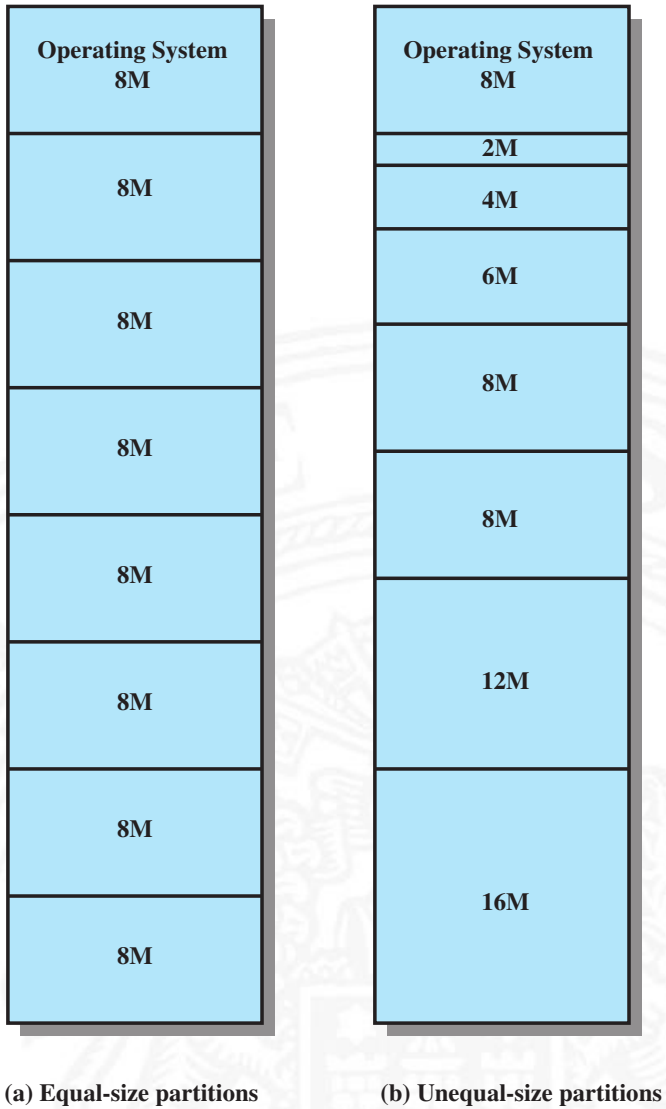
- Memory Management

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

Partitionierung

- Speicher in feste Bereiche unterteilt
  - Anzahl, Größe der Speicherbereiche?
  - Programme zu groß für Partition
    ⇒ Overlay-Techniken
  - interne Fragmentierung:
    ungenutzter Speicher in Partition

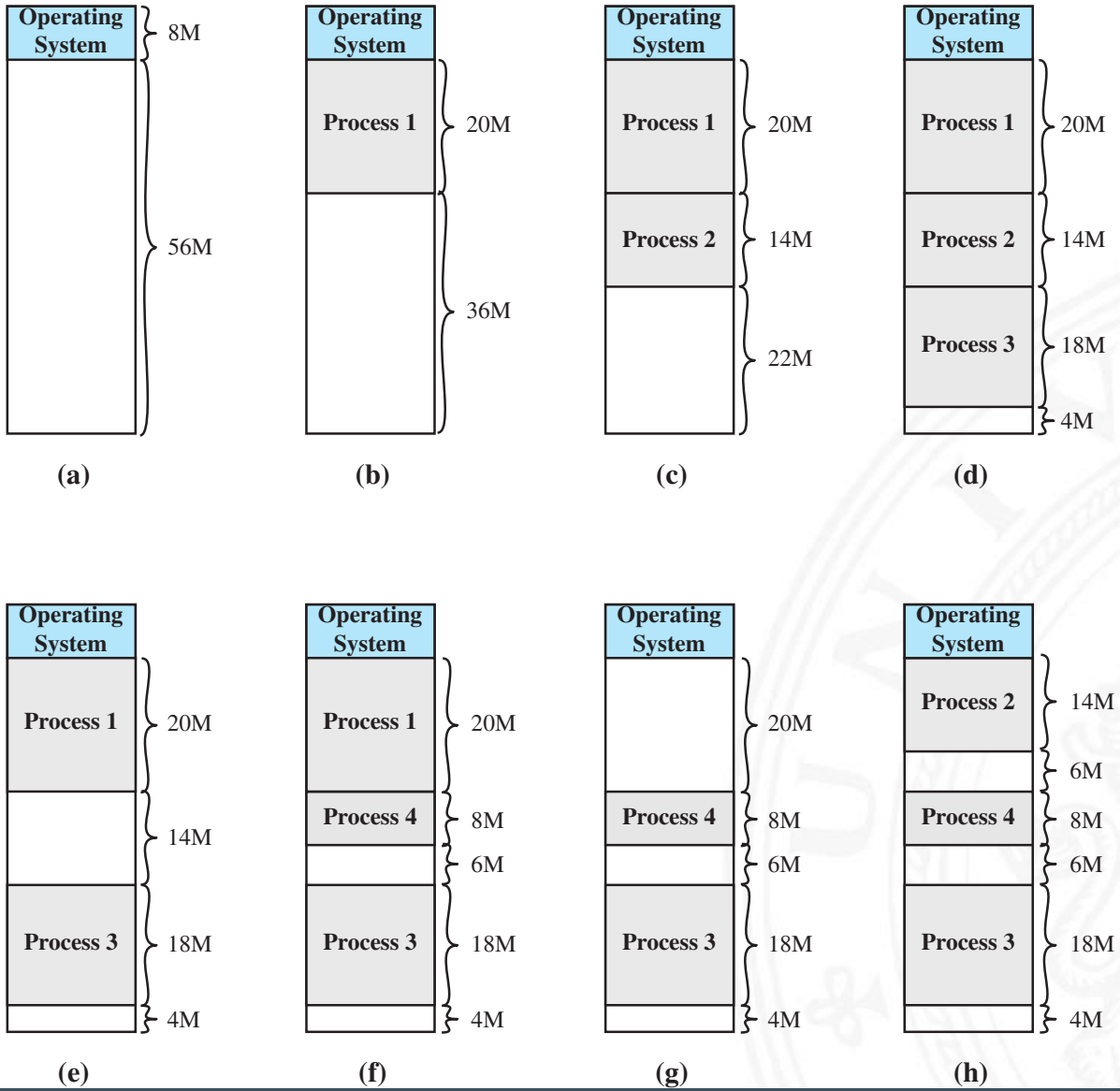
⇒ schlechte Speicherausnutzung
⇒ obsolet



(a) Equal-size partitions (b) Unequal-size partitions

Partitionierung (cont.)

- dynamische Partitionierung

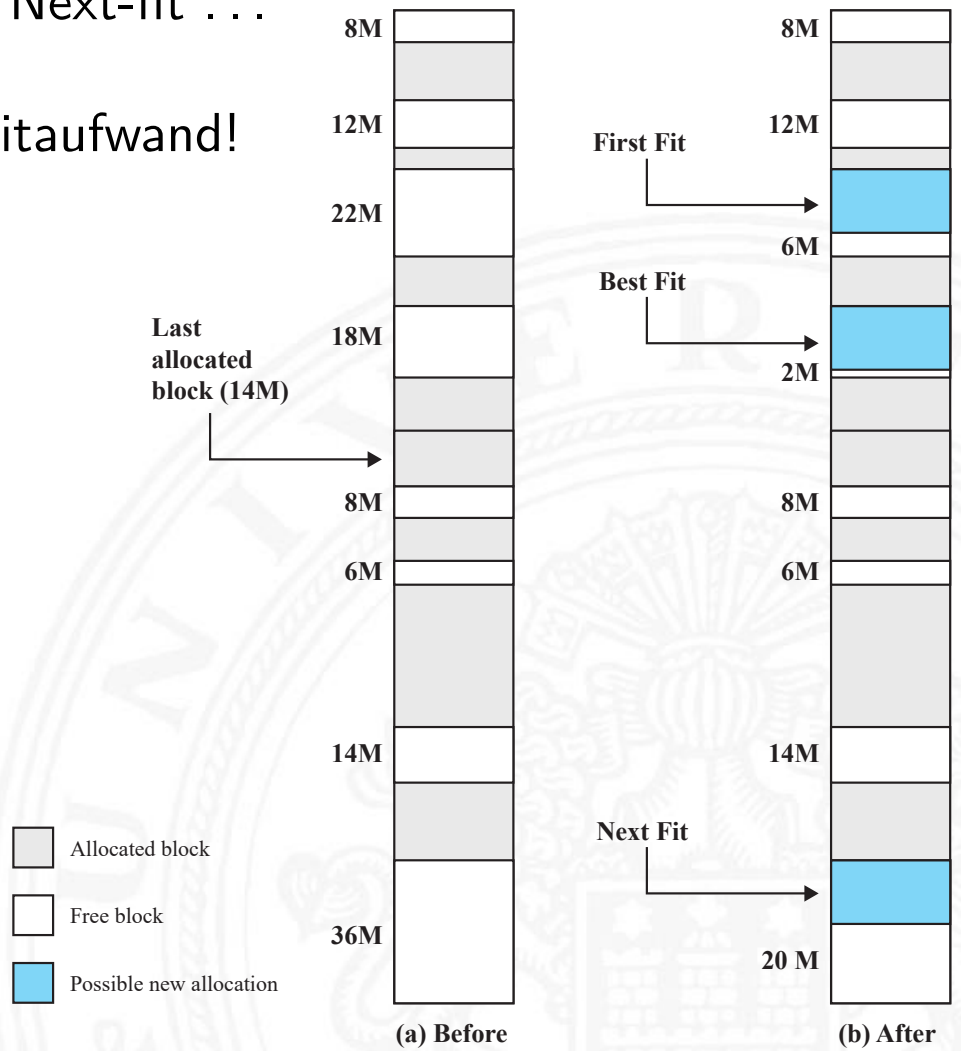


Partitionierung (cont.)

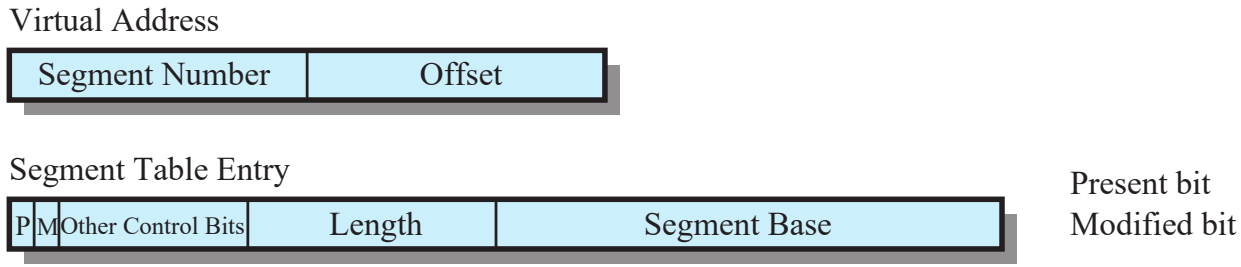
- externe Fragmentierung durch Ein- und Auslagern von Prozessen
- Platzierung: Best-fit, First-fit, Next-fit ...

– Kompaktierung notwendig, Zeitaufwand!

⇒ obsolet



Segmentierung



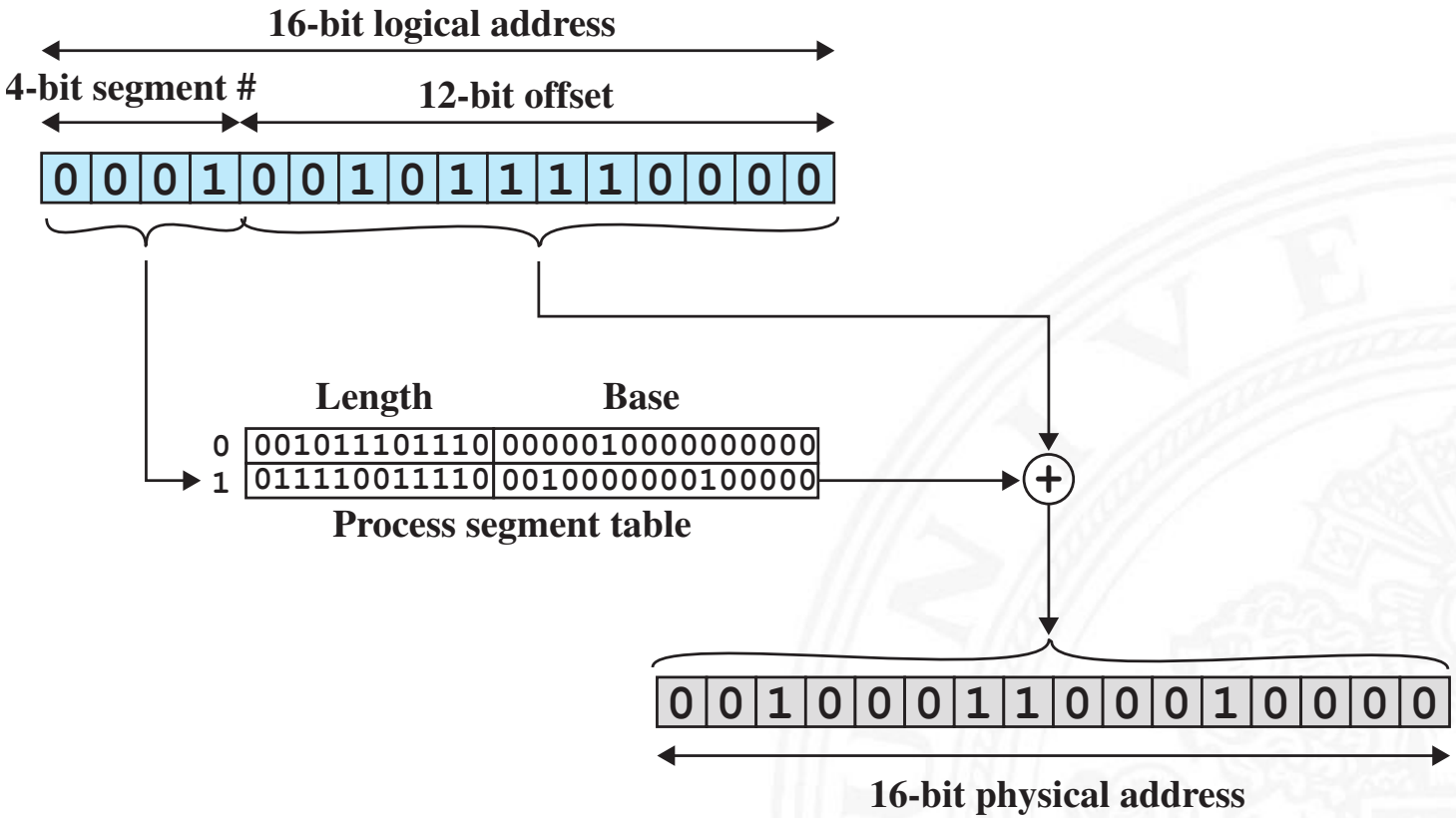
- „logische“ Unterteilung des Programms (Programmiersicht)
  - *Text*: Binärcode, read only
  - *Data*: statische Daten + dynamischer Speicher (*Heap*), read write
- variable Größe der Segmente
- ähnlich dynamischer Partitionierung
- für Programmierer sichtbar

\+ keine interne Fragmentierung

– externe Fragmentierung

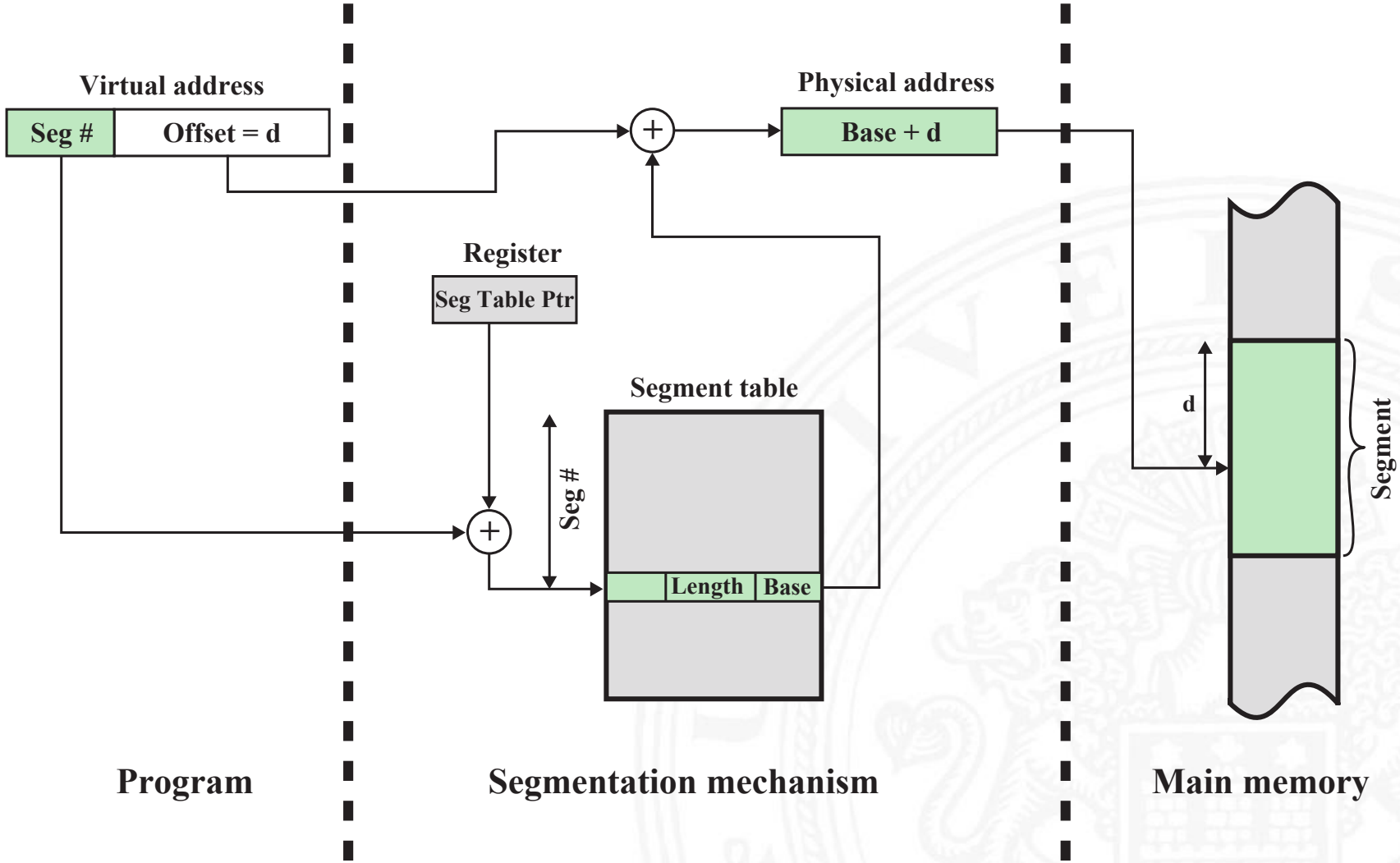
Segmentierung (cont.)

- Segmenttabelle $\langle segNr \rangle \rightarrow \langle Basisadresse \rangle + \langle Länge \rangle$
  - wird für jeden Prozess angelegt



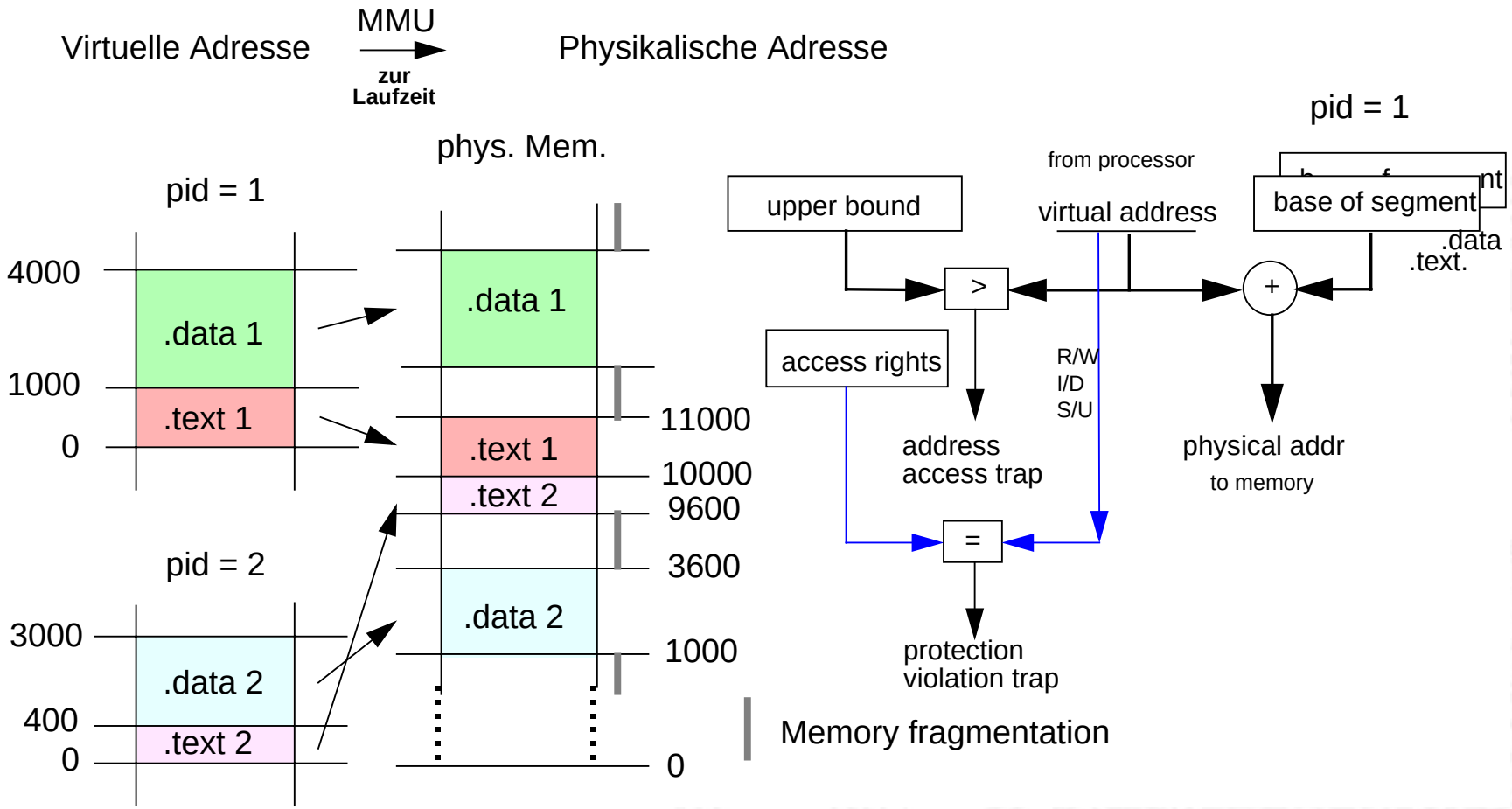
Segmentierung (cont.)

- Adressierung

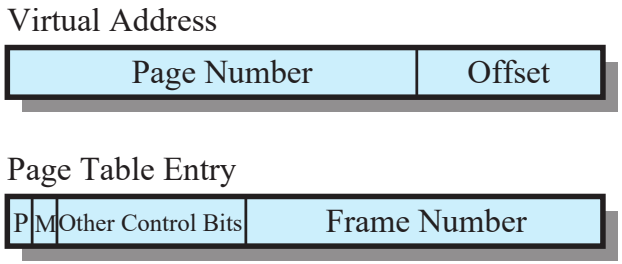


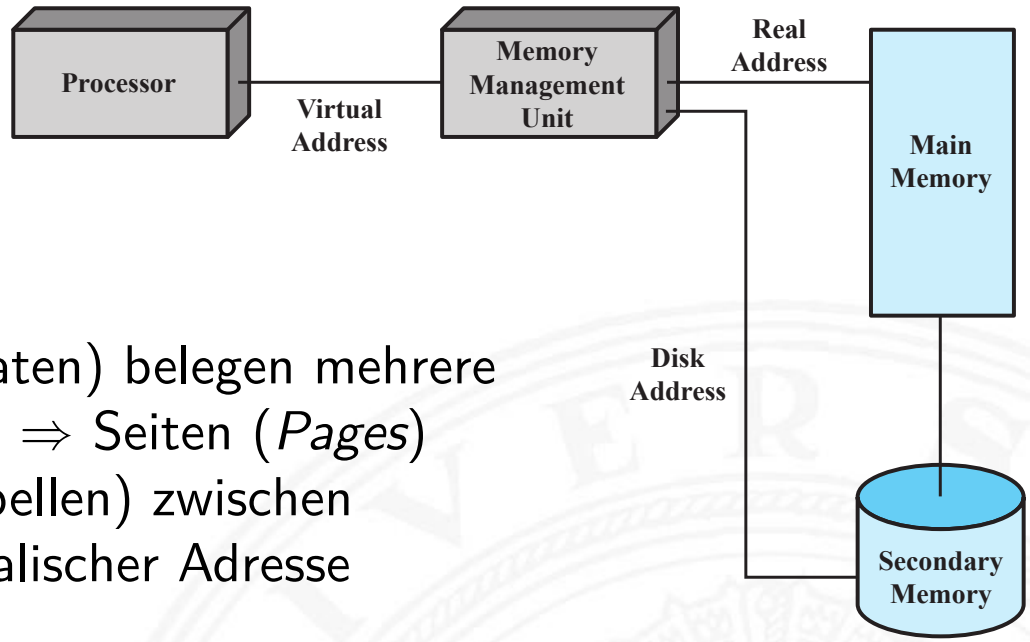
Segmentierung (cont.)

- Beispiel



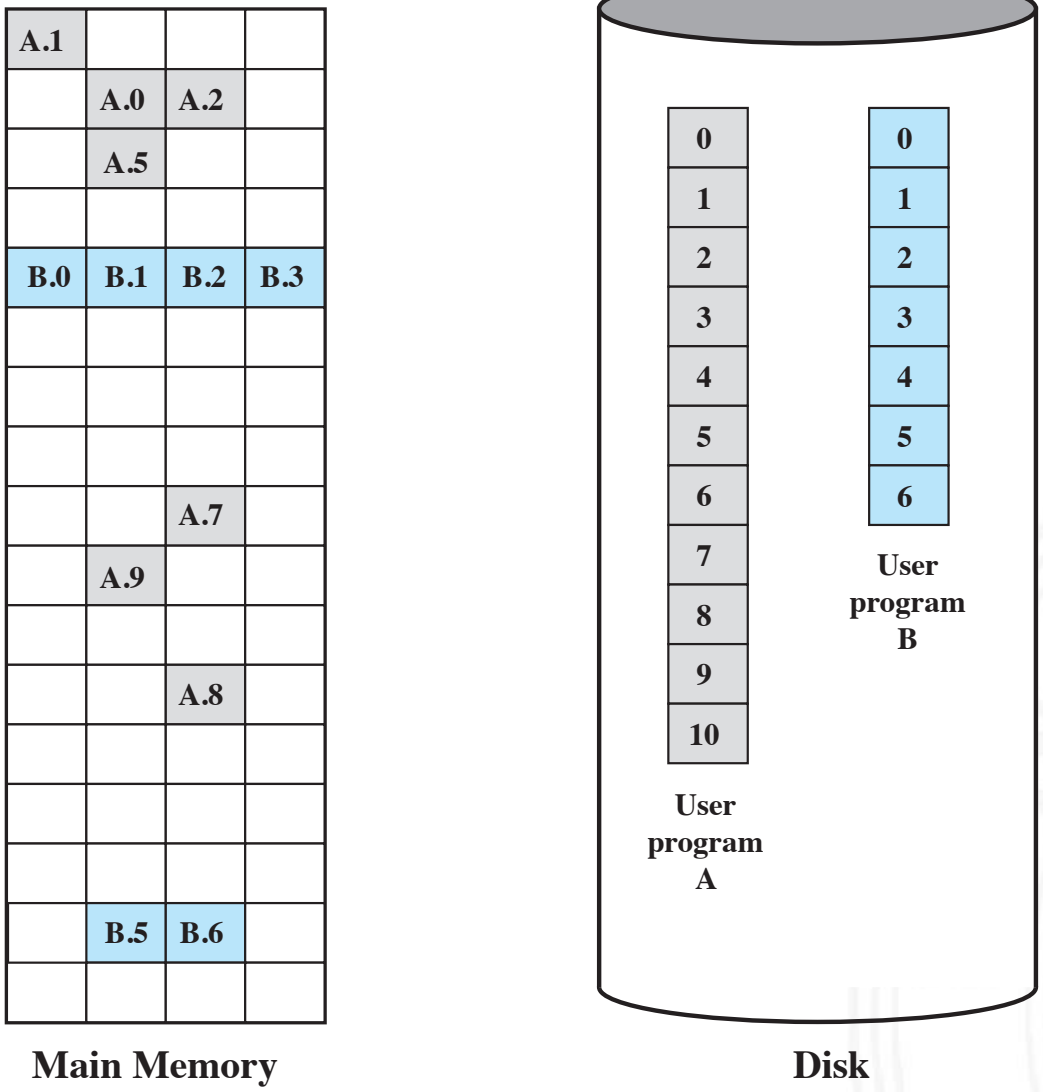
Paging





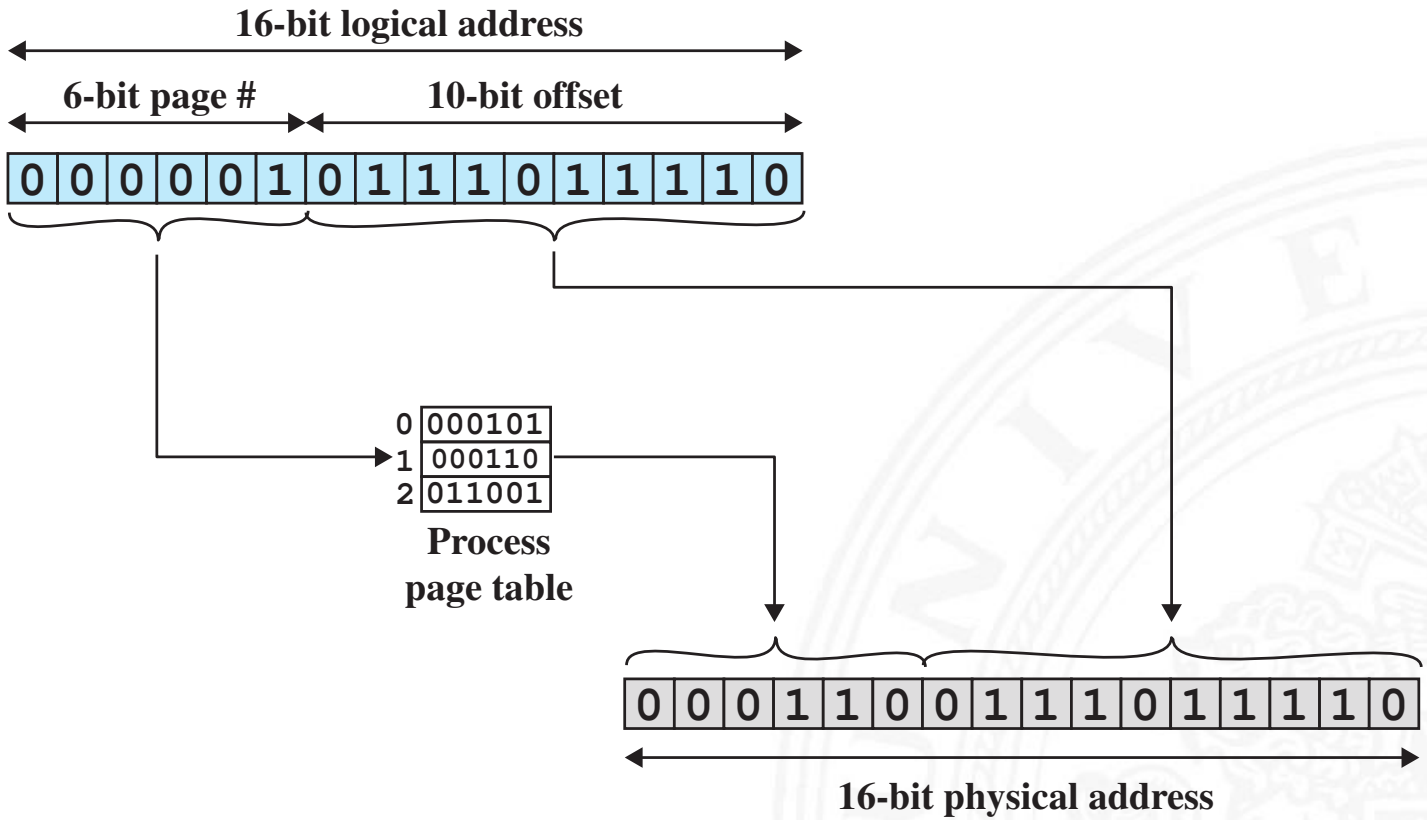
- Unterteilung des Programms
  - Prozesse (Instruktionen+Daten) belegen mehrere Speicherblöcke fester Größe ⇒ Seiten (*Pages*)
  - dynamische Abbildung (Tabellen) zwischen virtueller und realer, physikalischer Adresse
  - Speicherzugriff:
    $\langle \mathit{virt.\ Adresse} \rangle = \langle \mathit{Seitennr.} \rangle + \langle \mathit{offset} \rangle$
- Seite kann
  1. an beliebiger Stelle Hauptspeicher stehen
  2. auf sekundären Speicher (HDD, SSD) ausgelagert sein
     - Verwaltung durch *Memory Management Unit* (MMU)
- für Programmierer transparent

Paging (cont.)


A.1
A.0
A.2
A.5
B.0
B.1
B.2
B.3
A.7
A.9
A.8
B.5
B.6
Main Memory
0
1
2
3
4
5
6
7
8
9
10
User program A
0
1
2
3
4
5
6
User program B
Disk

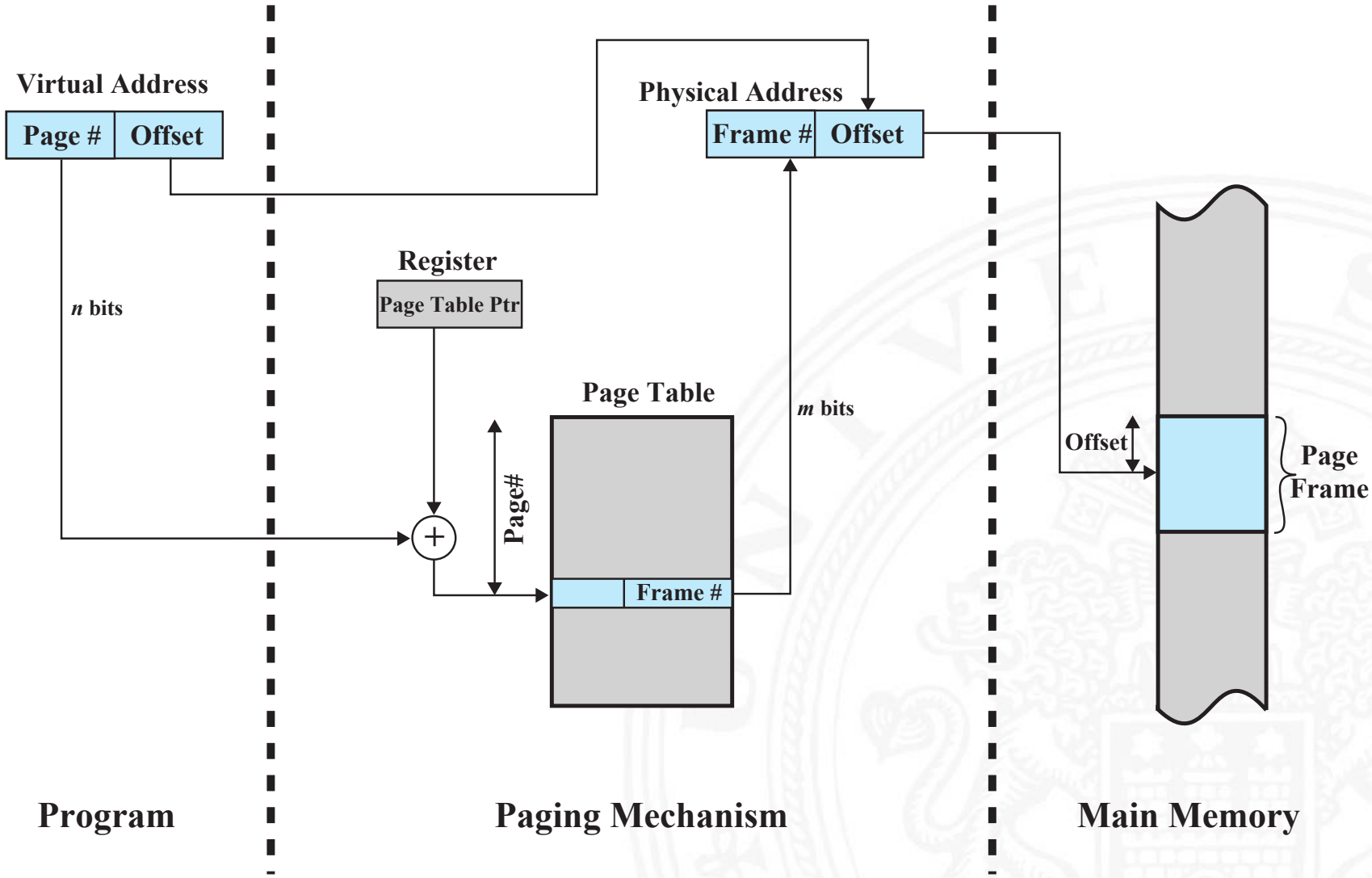
Paging (cont.)

- Seitentabelle / Page Table $\langle pageNr \rangle \rightarrow \langle frameAddr \rangle$
  - wird für jeden Prozess angelegt



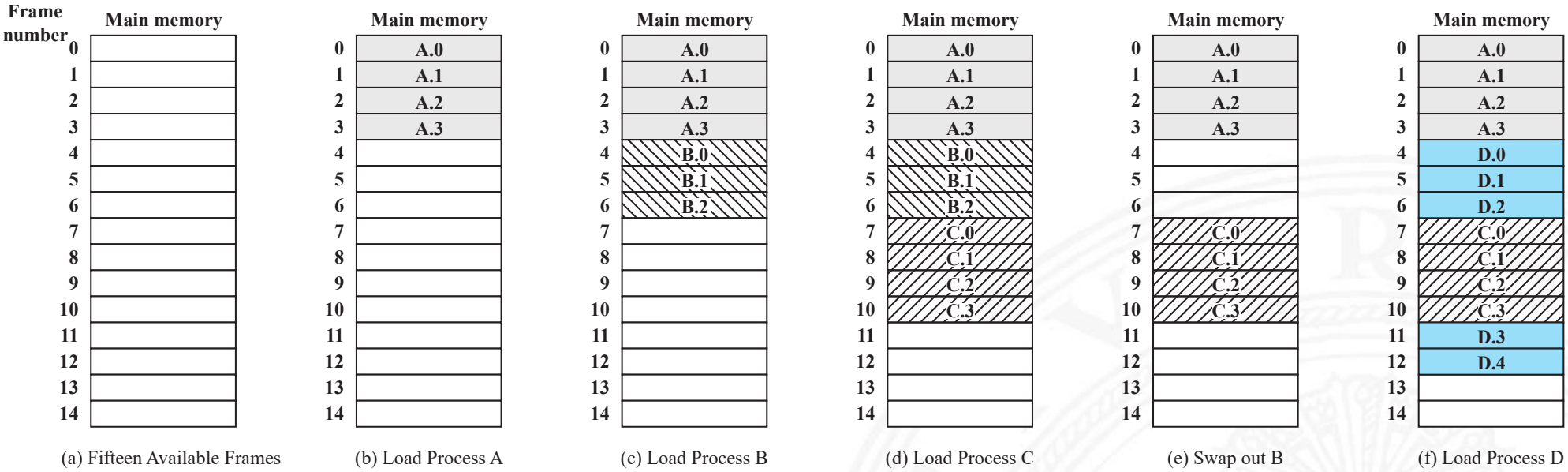
Paging (cont.)

- Adressierung



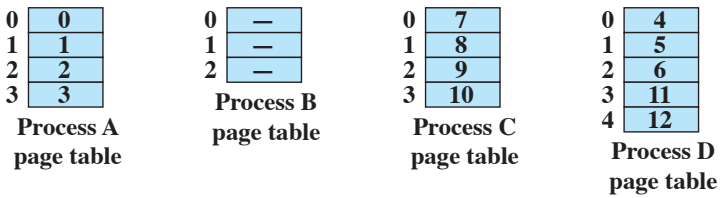
Paging (cont.)

- Beispiel



(a) Fifteen Available Frames (b) Load Process A (c) Load Process B (d) Load Process C (e) Swap out B (f) Load Process D

nach (f)

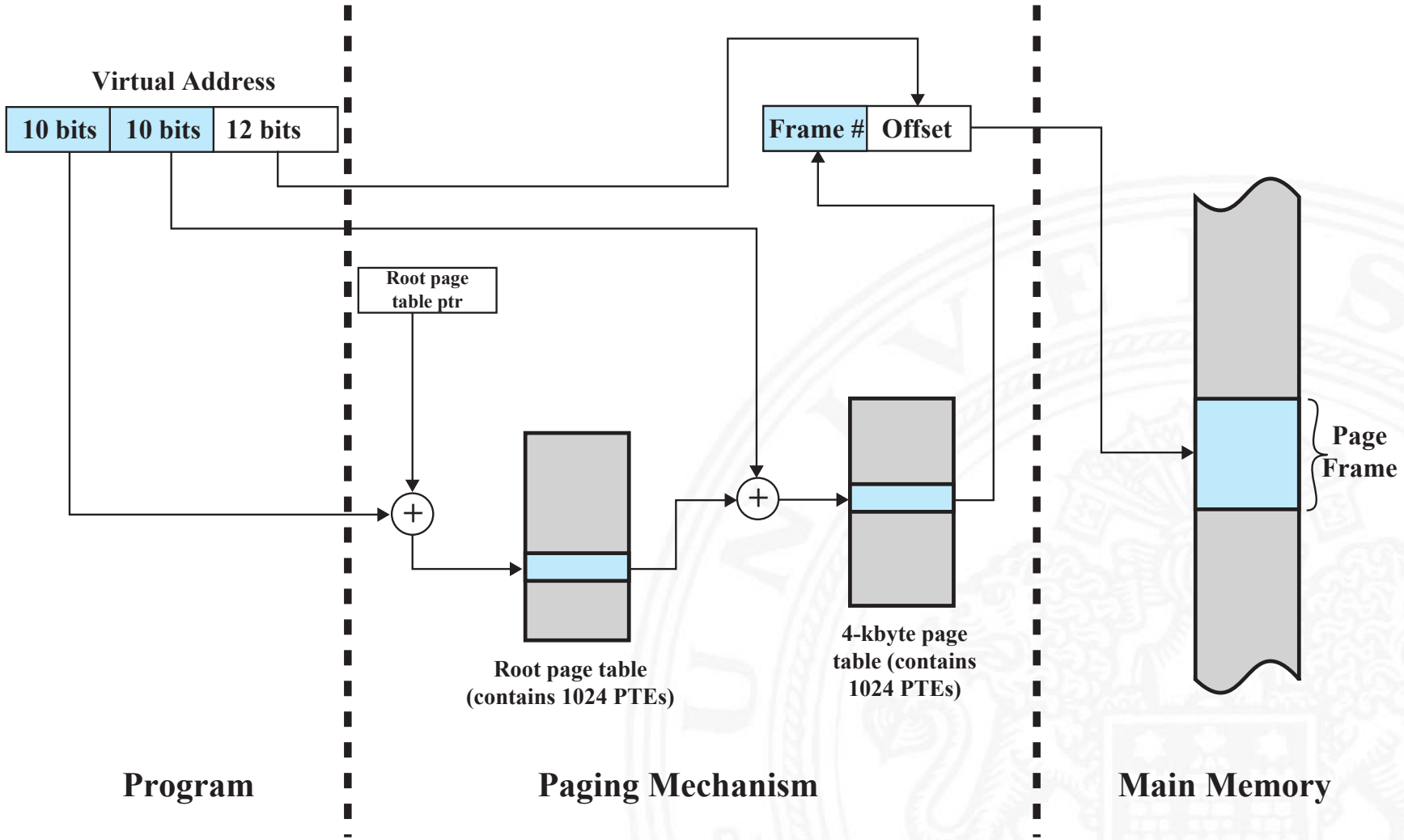


Paging (cont.)

- Seitengröße
  - 1 KiB, 4 KiB, 8 KiB ... 16 GiB (Hardware-, BS-abhängig!)
  - klein vs. groß
    - \+ weniger interne Fragmentierung
    - – aber sehr viele Seiten
    - – Prozesstabelle größer
    - – Datentransfer zu sekundärem Speicher für große Seiten effizienter
- mehrstufige Übersetzung
  - Tabelle wird zu groß $\Rightarrow$ mehrstufige Tabellen
  - – mehrfacher Zugriff langsam

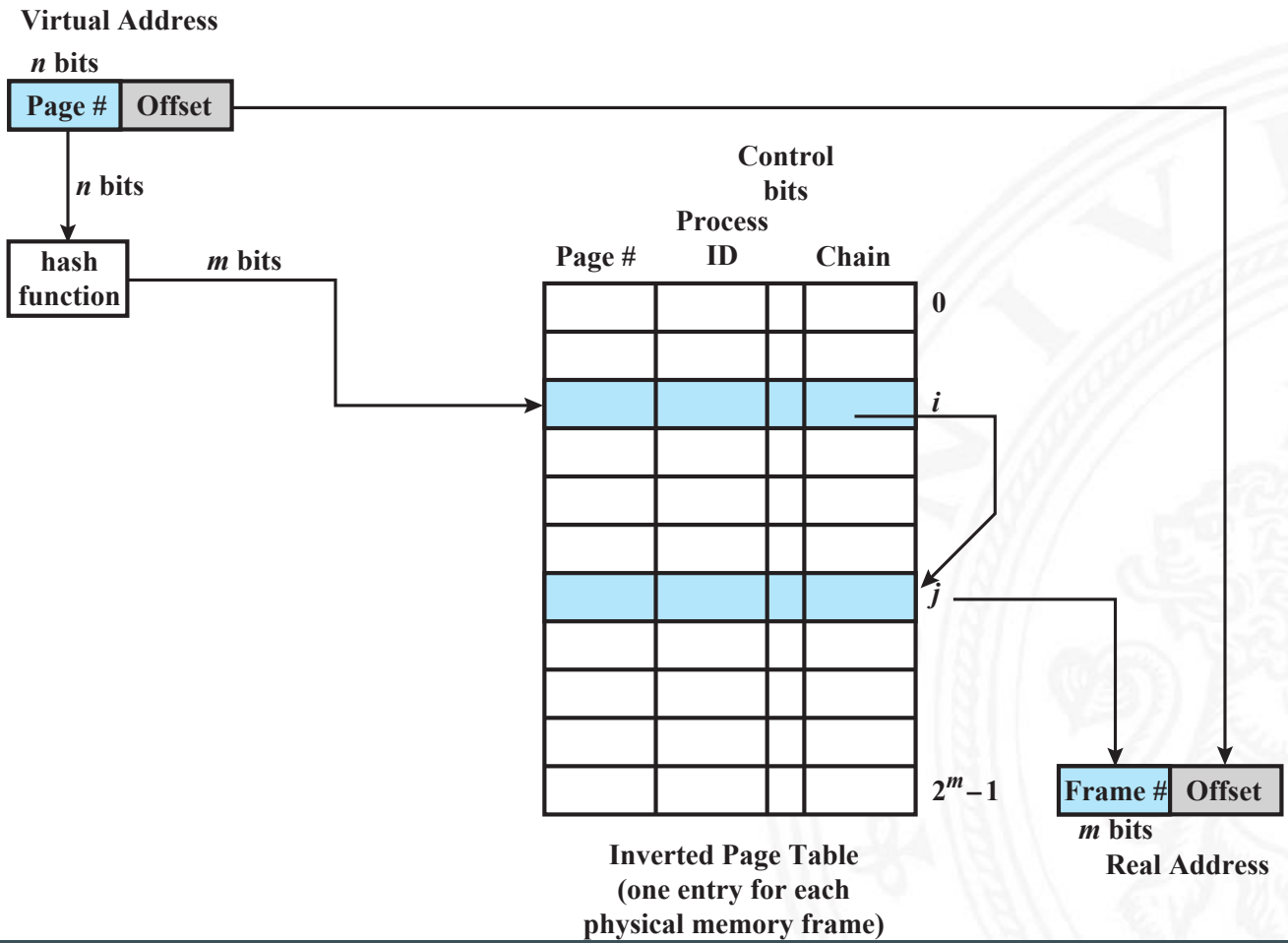
Paging (cont.)

mehrstufige Übersetzung



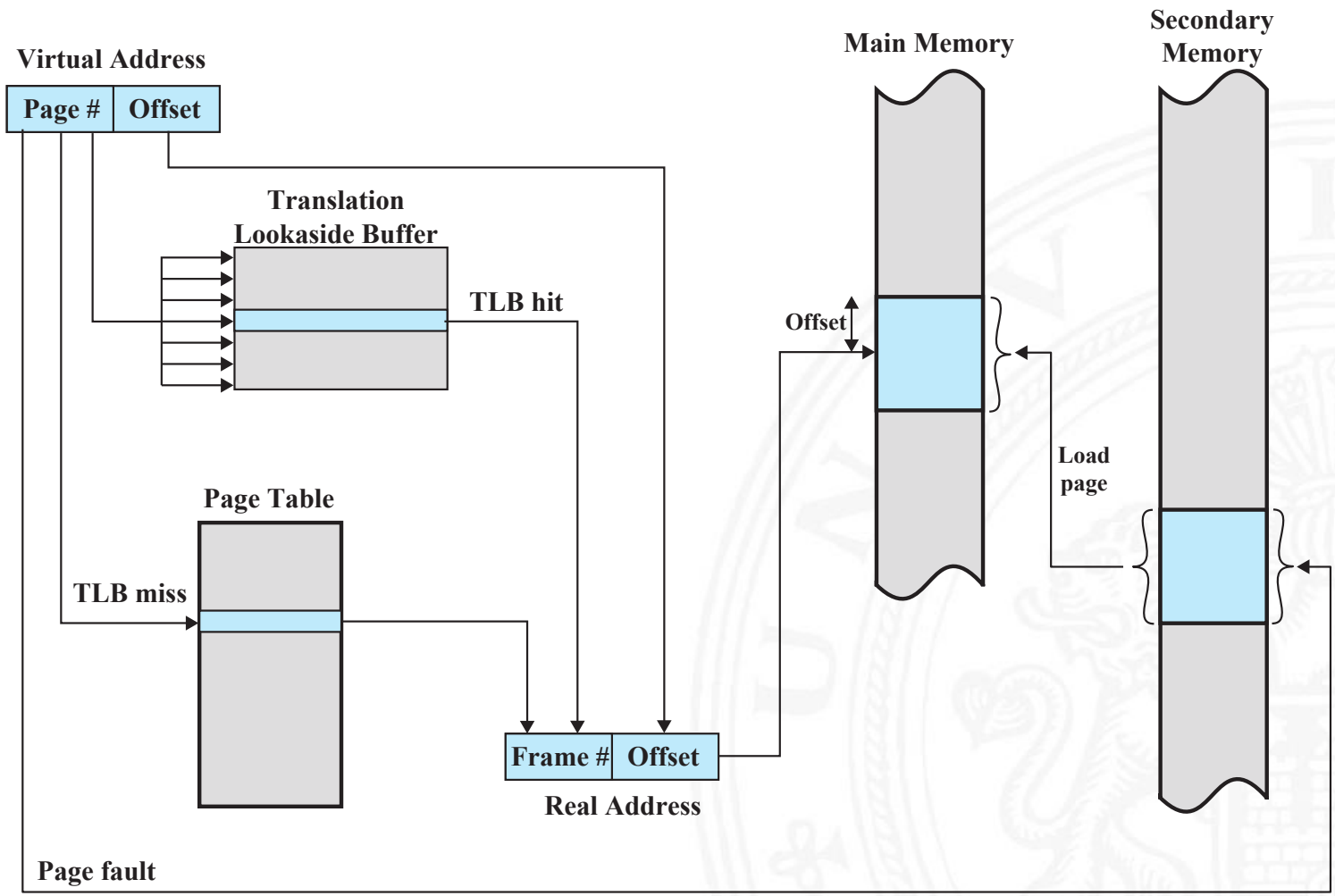
Paging (cont.)

- *Inverted Page Table*
  - statt (mehrerer) großer Seitentabellen (für alle virtuellen Adressen)
  - eine Tabelle (für alle physikalischen Adressen)
  - Hash-Funktion + verkettete Liste (Suche!)



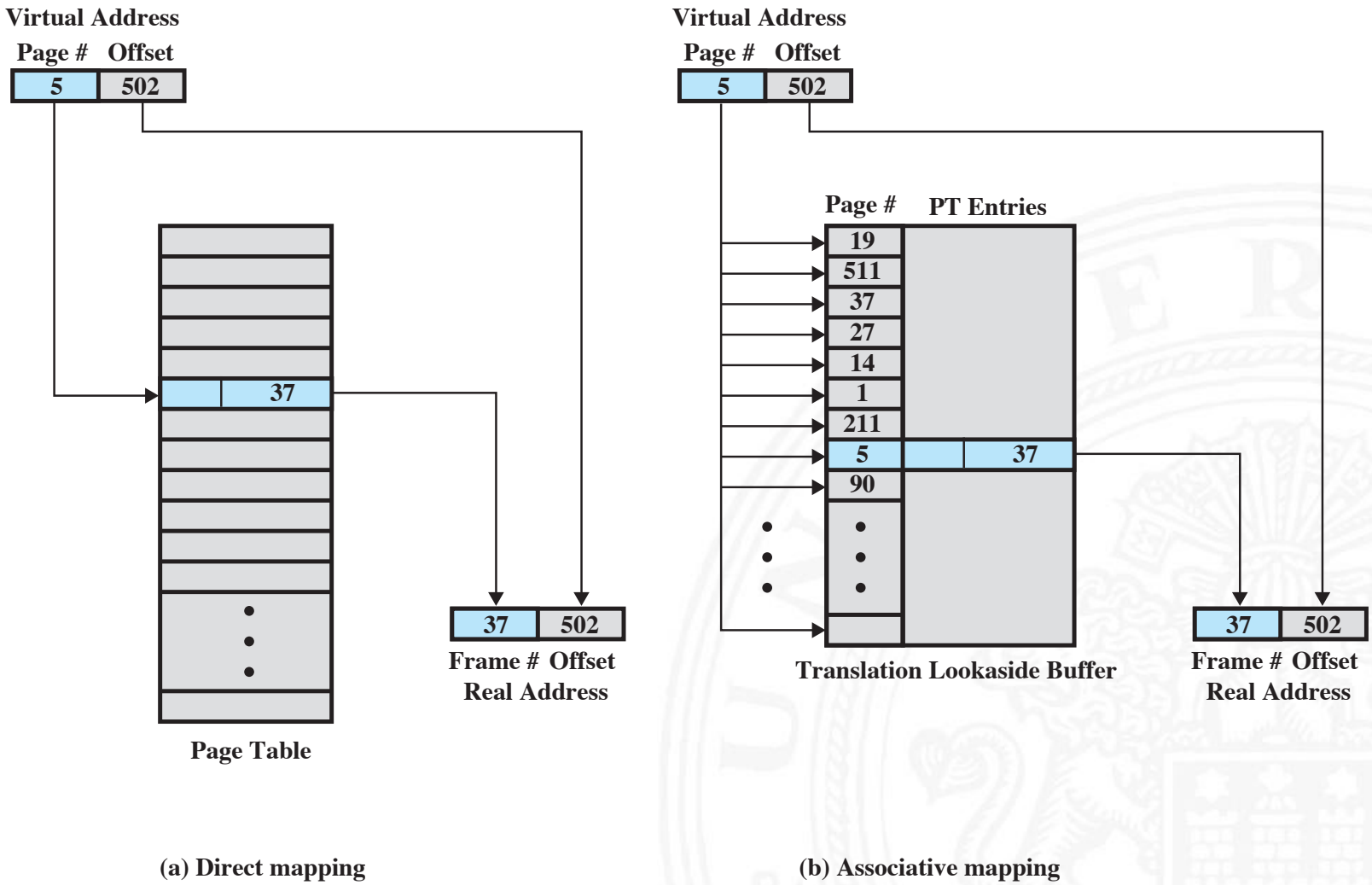
Paging (cont.)

- *Translation Lookaside Buffer (TLB)*
  - schneller Zugriff
  - voll-assoziativer Cache

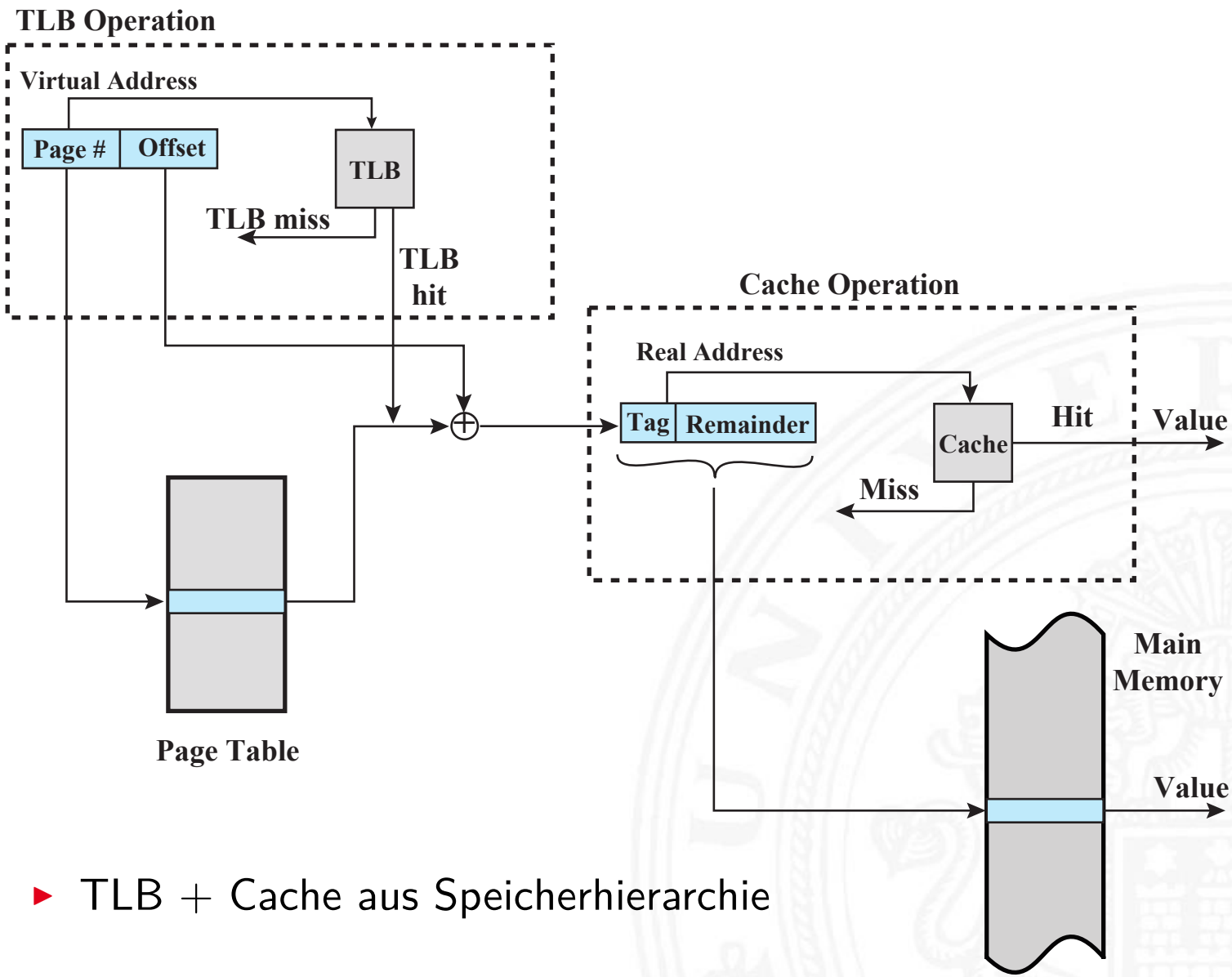


Paging (cont.)

- voll-assoziativer Cache

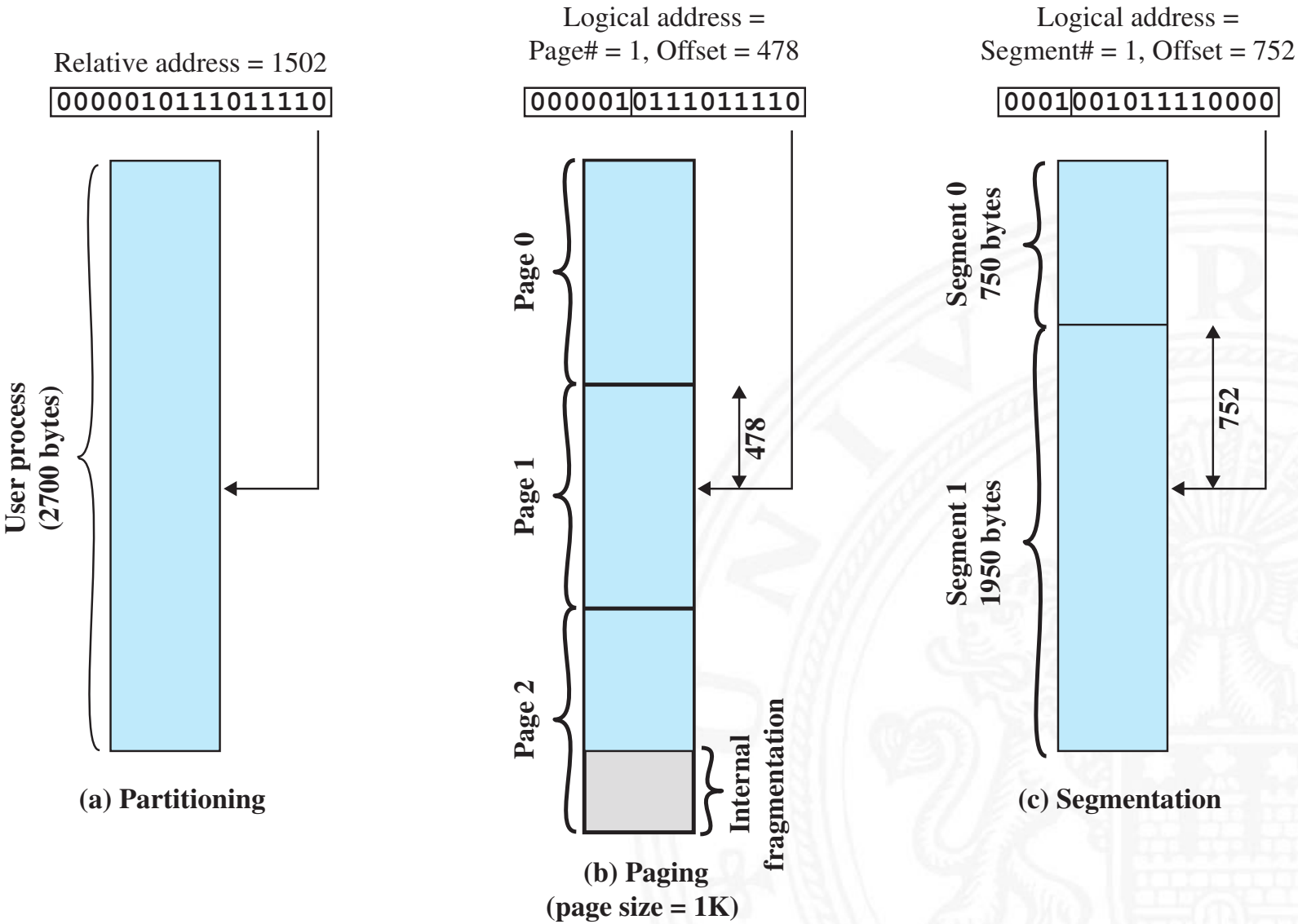


Paging (cont.)

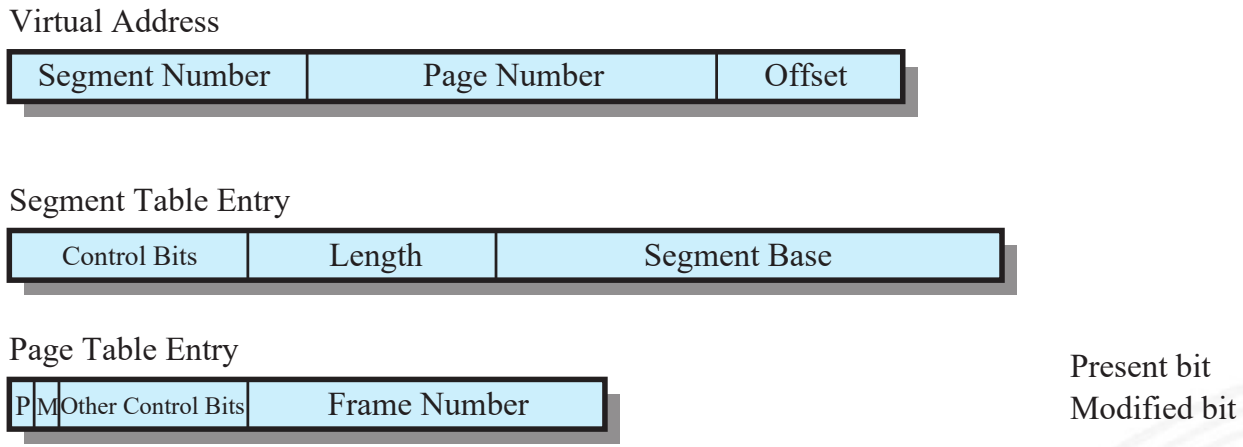


- TLB + Cache aus Speicherhierarchie

- Übersicht Adressierung



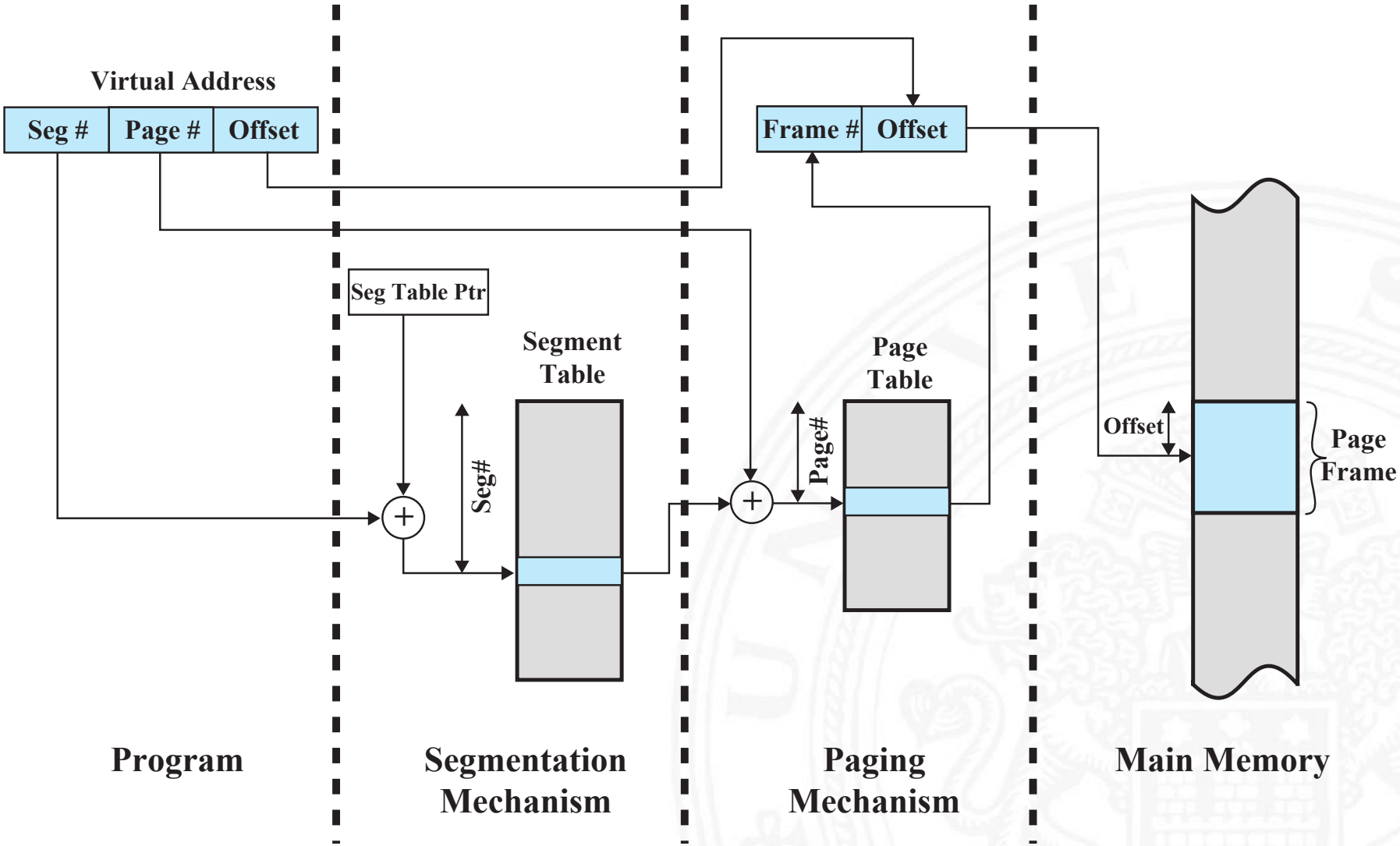
aktuelle Betriebssysteme (cont.)



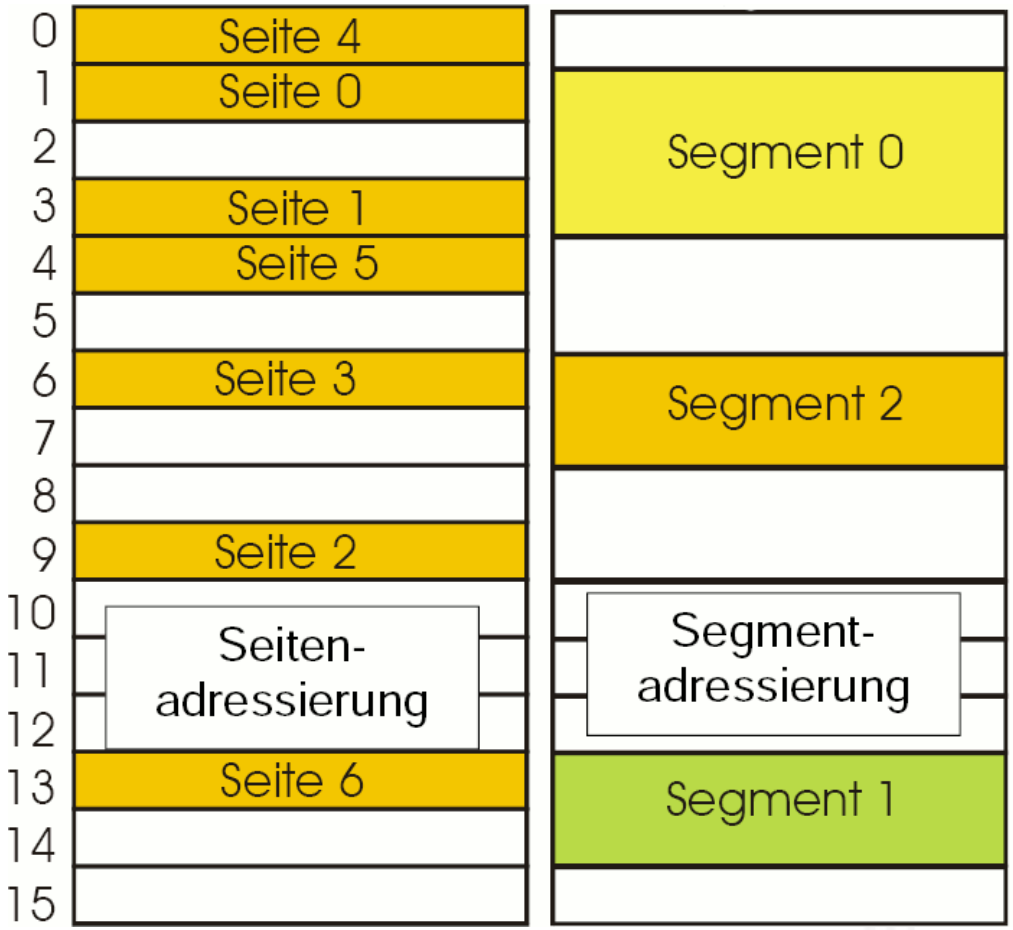
- Segmentierung + Paging
  - Segmentierung ⇒ logische Trennung (Text, Data)
  - + Paging ⇒ effiziente Verwaltung
- im Betriebssystem festlegen
  - wann Seiten Laden? (auf Anfrage, Prepaging)
  - wo in Hauptspeicher?, welche Seiten werden ausgelagert?
    = Platzierungs- und Ersetzungsstrategie
  - wieviel Multiprogramming? (Anzahl der Prozesse)
  - wieviel Hauptspeicher pro Prozess (fest, dynamisch . . .)
  - vielfältige Wechselwirkungen: Anwendungsszenario, Caching etc.

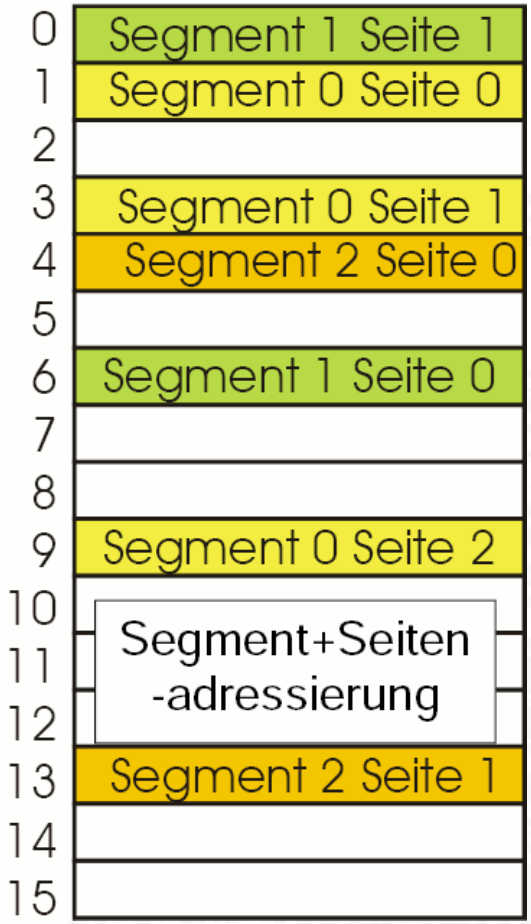
aktuelle Betriebssysteme (cont.)

- Adressierung



aktuelle Betriebssysteme (cont.)



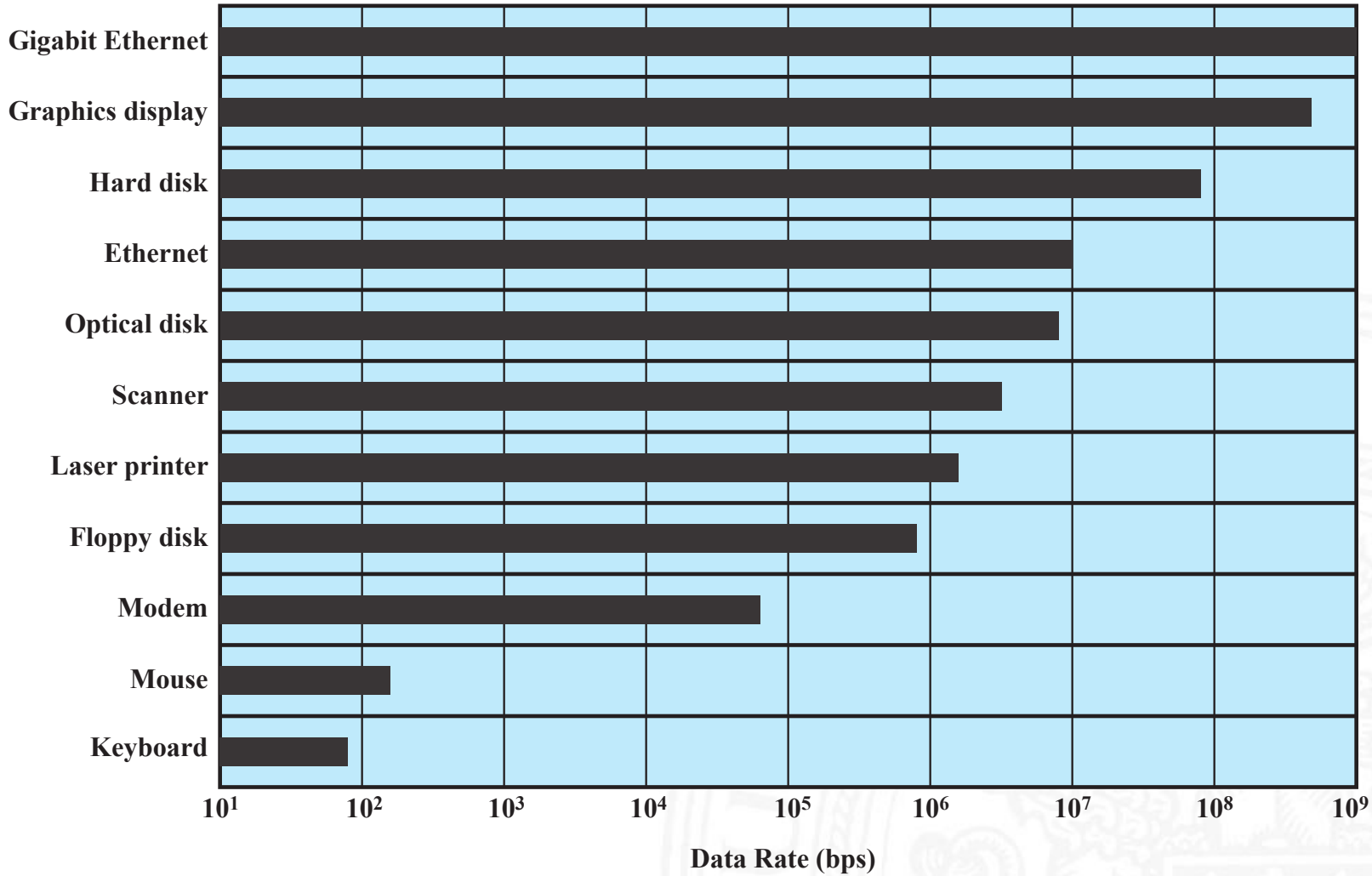


Ein-/Ausgabegeräte

- Mensch-Maschine Schnittstellen
  - Displays, Tastatur, Maus, Terminals, Drucker ...
- Schnittstellen zu Computerperipherie
  - Festplatten, Speichermedien, Sensoren, Controller ...
- Kommunikation zwischen Computersystemen
  - Netzwerk, Modems ...

- Charakteristika
  - Datenrate
  - Anwendung
  - Schnittstellen
  - Protokoll / Fehlerbedingungen
  - Daten (Zeichen, Datenstrukturen, Blöcke von Bytes ...)
  - Repräsentation der Daten
  - ...

Ein-/Ausgabegeräte (cont.)



I/O-Optimierung

- Effizienz
  - I/O oft als „*Bottleneck*“ im System
  - extrem langsam, verglichen mit Hauptspeicher oder CPU
- Generalität
  - einheitliche Schnittstelle für (möglichst viele) Geräte (-klassen)
  - sowohl als Programmierschnittstelle
  - als auch für das Betriebssystem selber
  - hierarchische, modulare Konzepte

  $\Rightarrow$ wegen der völlig unterschiedlichen Geräteeigenschaften und der Dynamik der technischen Entwicklung schlecht realisierbar
- stufenweise Entwicklung
  1. Prozessor kontrolliert Gerät direkt
  2. Prozessor kontrolliert Gerät über I/O-Modul (Controller)
  3. + Interruptsteuerung
  4. I/O-Controller hat Hauptspeicherzugriff (DMA)
  5. I/O-Prozessor: eigener Befehlssatz (programmierbar), statt „Automat“
  6. I/O-System mit eigenem Speicher = Computer, realisiert I/O-Dienst

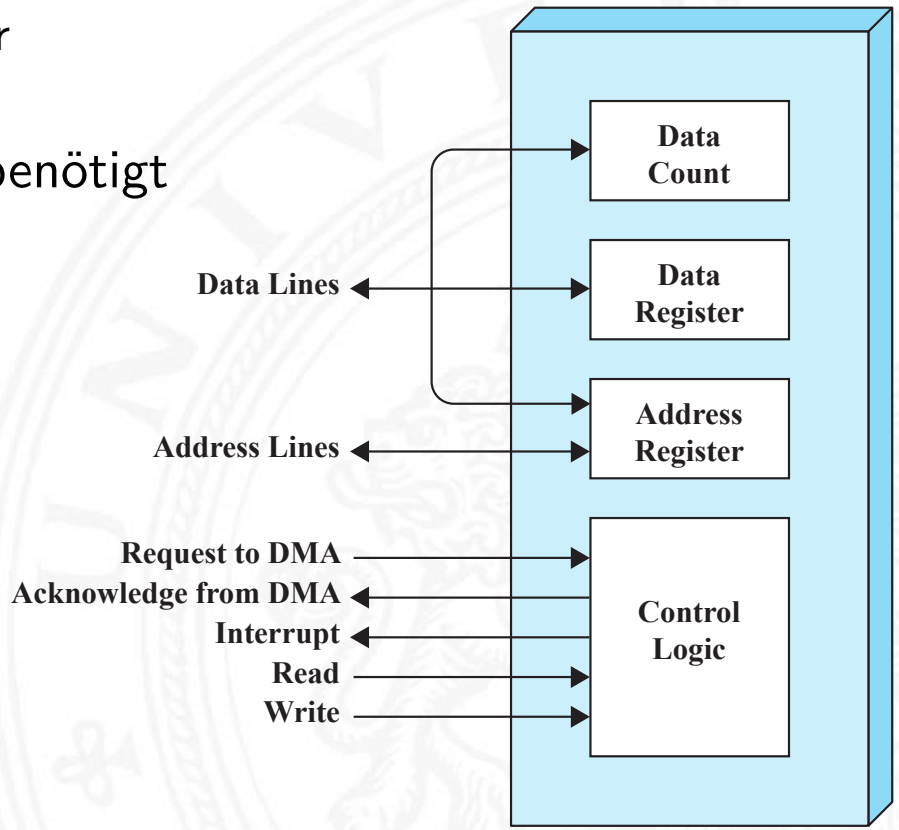
Ein-/Ausgabe Behandlung

1. Programmed I/O
   - I/O-Modul führt Operation aus
   - setzt Bits in Status-Register (Kommunikation)
   - Prozessor fragt periodisch Status ab
   - – „*Busy-waiting*“, schlechte Performanz

2. Interrupt gesteuert
   - Befehl von Prozessor an I/O-Modul
   - Interrupt, wenn I/O bereit
   - Datentransfer durch Prozessor (in ISR)
   - – Beteiligung des Prozessors
     Performanz (mehrere Instruktionen pro Datentransfer)
   - → siehe *15.2 Betriebssysteme – Interrupts*

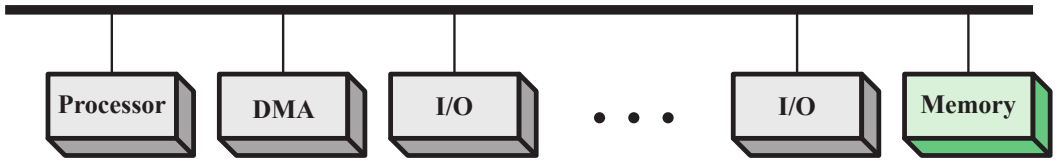
Ein-/Ausgabe Behandlung (cont.)

3. DMA (*Direct Memory Access*)
   - separate Hardwareeinheit mit Zugriff auf Systembus
   - DMA-Controller übertragt selbständig Daten
   - DMA-Kommando mit
     - Lesen / Schreiben
     - Adresse des I/O Geräts
     - Startadresse in Hauptspeicher
     - Anzahl Datenwörter
   - Konflikt, wenn Prozessor Bus benötigt

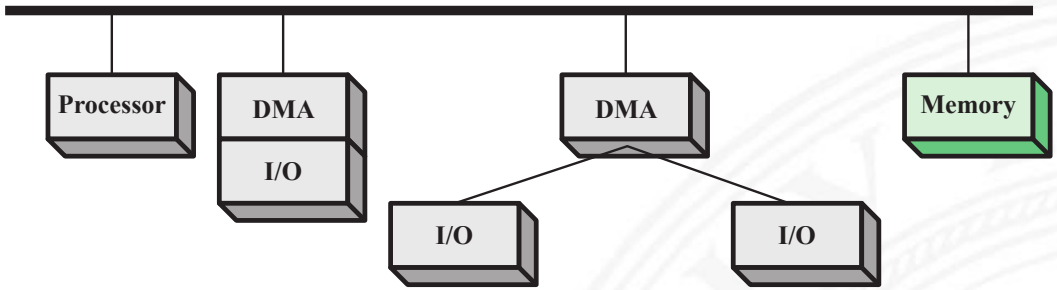


Ein-/Ausgabe Behandlung (cont.)

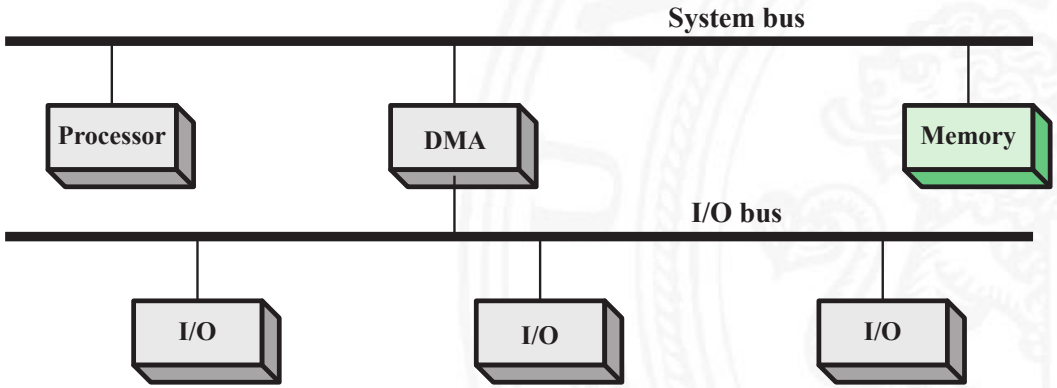
DMA



(a) Single-bus, detached DMA



(b) Single-bus, Integrated DMA-I/O



(c) I/O bus

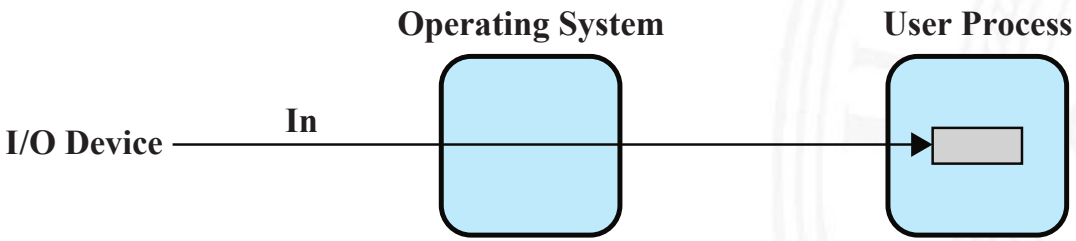
Ein-/Ausgabe Behandlung (cont.)

- I/O Betriebsarten

	No Interrupts	**Use of Interrupts**
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

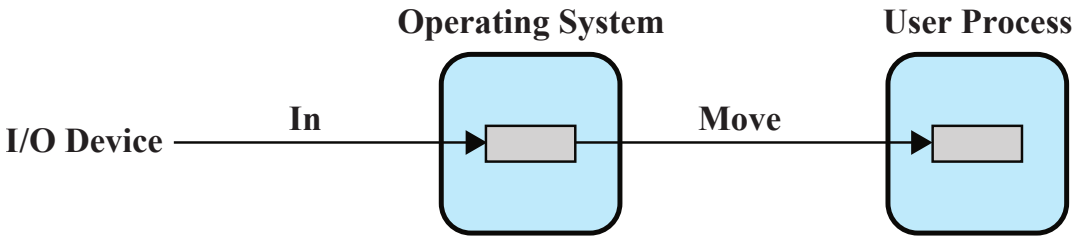
- Schnittstelle zum Benutzerprogramm / *Buffering*
  - lesend: „read-ahead“, Daten schon bereitstellen
  - schreibend: verzögerte (autonome) Ausführung

  $\Rightarrow$ Zwischenspeicher, in der Regel FIFO

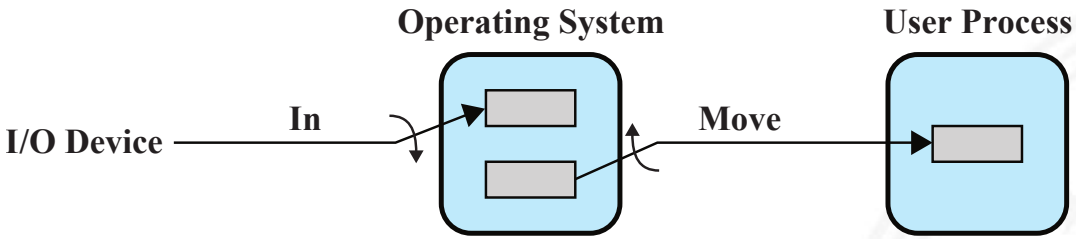


(a) No buffering

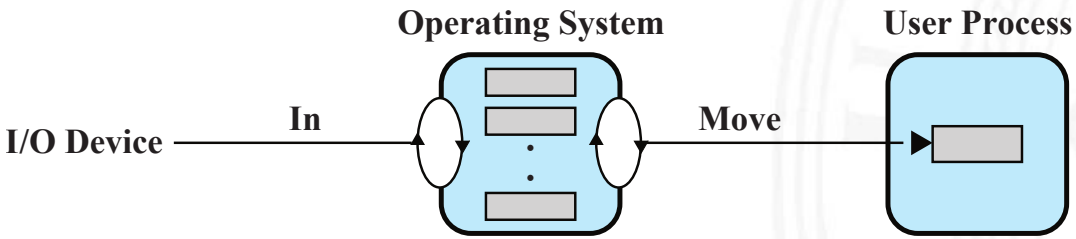
Ein-/Ausgabe Behandlung (cont.)



(b) Single buffering



(c) Double buffering

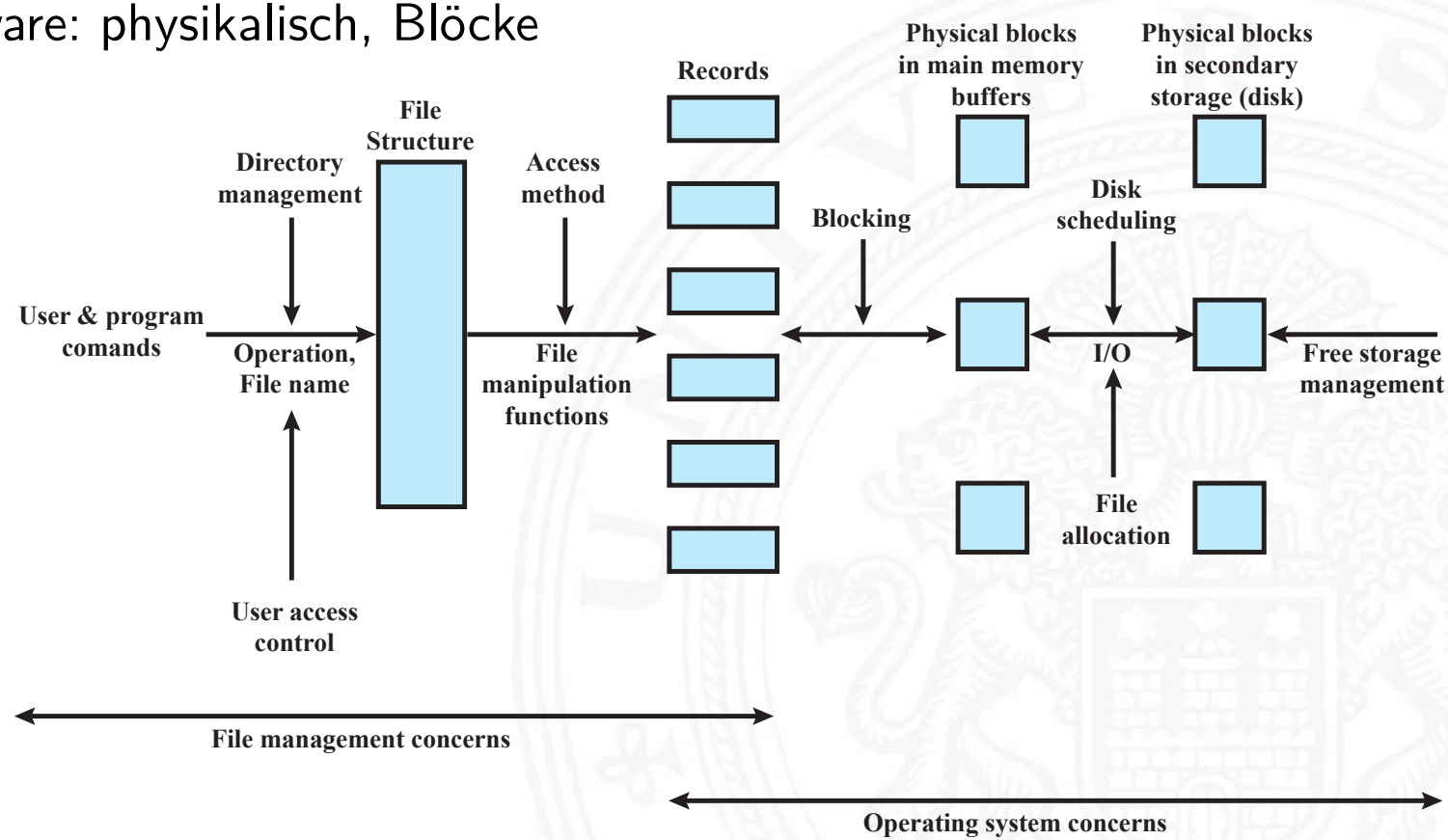


(d) Circular buffering

Weitere Themen

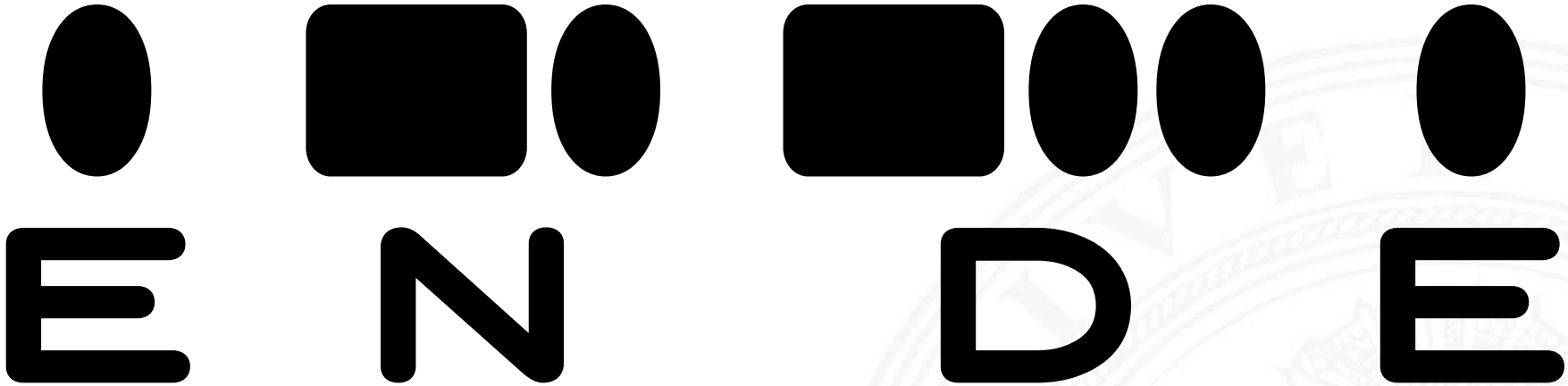
- Festplattenzugriffe: das zentrale Thema bei I/O-Optimierung
  - viele Algorithmen zum „Umsortieren“ von Schreib-/Lesebefehlen

  $\Rightarrow$ Optimierung des Durchsatzes
- Daten-Organisation
  - Benutzer: logisch, Dateisysteme und Verzeichnisse
  - Hardware: physikalisch, Blöcke



Weitere Themen

Weitere Themen


E N D E

Literatur

[Sta17] W. Stallings: *Operating Systems – Internals and Design Principles*.
9th, global ed., Pearson Education, 2017.
ISBN 978–1–292–21429–0

[Bau17] C. Baun: *Betriebssysteme kompakt*.
Springer-Verlag GmbH, 2017.
ISBN 978–3–662–53142–6

[Bra17] R. Brause: *Betriebssysteme – Grundlagen und Konzepte*.
Springer-Verlag GmbH, 2017.
ISBN 978–3–662–54099–2

[TB16] A.S. Tanenbaum, H. Bos: *Moderne Betriebssysteme*.
4. Auflage, Pearson Deutschland GmbH, 2016.
ISBN 978–3–8689–4270–5

Literatur (cont.)

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