

64-040 Modul InfB-RSB

Rechnerstrukturen und Betriebssysteme

[https://tams.informatik.uni-hamburg.de/
lectures/2023ws/vorlesung/rsb](https://tams.informatik.uni-hamburg.de/lectures/2023ws/vorlesung/rsb)
<https://lernen.min.uni-hamburg.de>

Andreas Mäder



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

Wintersemester 2023/2024



Gliederung

64-040 Rechnerstrukturen und Betriebssysteme

1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. Logische Operationen
7. Codierung
8. Schaltfunktionen
9. Schaltnetze
10. Schaltwerke
11. Rechnerarchitektur I
12. Instruction Set Architecture



- 13. Assembler-Programmierung
- 14. Rechnerarchitektur II
- 15. Betriebssysteme



1. Einführung

- Exkurs: Geschichte
- Personal Computer
- von-Neumann Konzept
- Moore's Law
- System on a chip
- Roadmap und Grenzen des Wachstums
- Literatur

- 2. Informationsverarbeitung
- 3. Ziffern und Zahlen
- 4. Arithmetik
- 5. Zeichen und Text
- 6. Logische Operationen





Gliederung (cont.)

1 Einführung

64-040 Rechnerstrukturen und Betriebssysteme

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



A. Mäder

5



Informatik

1 Einführung

64-040 Rechnerstrukturen und Betriebssysteme

Brockhaus-Enzyklopädie: „Informatik“

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



A. Mäder

6



Brockhaus-Enzyklopädie: „Informatik“

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

system. Verarbeitung: von-Neumann Paradigma

- ▶ 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 Paradigma

- ▶ 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 arbeiten technische Schaltungen (Hardware) Befehle ab?
- ⇒ Hardwareentwicklung



Trennung von Software und Hardware?

1 Einführung

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ seit 80er Jahren: unterschiedliche, getrennte Paradigmen
 - SW** ▶ Hardware ist vorgegeben
 - ▶ Abstraktion von der Hardware möglich
 - ▶ Programmierung in Hochsprachen (Produktivität)
 - HW** ▶ technische Werte als Optimierungsziel (Taktfrequenz, Latenz, Durchsatz, Leistungsaufnahme etc.)
 - ▶ getrieben von technischer Entwicklung (*Moore's Law*)
 - ▶ Maschinenbefehl wird auf Hardwarearchitektur ausgeführt
 - ▶ Kontext aus SW und Betriebssystem wird nicht beachtet
- ▶ Trend: technischer Fortschritt langsamer
Leistungssteigerungen durch neue Architekturkonzepte

A. Mäder

7



Trennung von Software und Hardware?

1 Einführung

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ seit 80er Jahren: unterschiedliche, getrennte Paradigmen
 - SW** ▶ Hardware ist vorgegeben
 - ▶ Abstraktion von der Hardware möglich
 - ▶ Programmierung in Hochsprachen (Produktivität)
 - HW** ▶ technische Werte als Optimierungsziel (Taktfrequenz, Latenz, Durchsatz, Leistungsaufnahme etc.)
 - ▶ getrieben von technischer Entwicklung (*Moore's Law*)
 - ▶ Maschinenbefehl wird auf Hardwarearchitektur ausgeführt
 - ▶ Kontext aus SW und Betriebssystem wird nicht beachtet
 - ▶ Trend: technischer Fortschritt langsamer
Leistungssteigerungen durch neue Architekturkonzepte
- ⇒ dies funktioniert seit Jahren!

A. Mäder

7



Trennung von Software und Hardware?

1 Einführung

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ seit 80er Jahren: unterschiedliche, getrennte Paradigmen
 - SW** ▶ Hardware ist vorgegeben
 - ▶ Abstraktion von der Hardware möglich
 - ▶ Programmierung in Hochsprachen (Produktivität)
 - HW** ▶ technische Werte als Optimierungsziel (Taktfrequenz, Latenz, Durchsatz, Leistungsaufnahme etc.)
 - ▶ getrieben von technischer Entwicklung (*Moore's Law*)
 - ▶ Maschinenbefehl wird auf Hardwarearchitektur ausgeführt
 - ▶ Kontext aus SW und Betriebssystem wird nicht beachtet
- ▶ Trend: technischer Fortschritt langsamer
Leistungssteigerungen durch neue Architekturkonzepte
- ▶ dies funktioniert seit Jahren ... bis Ende 2017



...

Wechselwirkungen zwischen SW und HW
als „Angriffsszenario“

A. Mäder

7



Trennung von Software und Hardware? (cont.)

1 Einführung

64-040 Rechnerstrukturen und Betriebssysteme

Konsequenz

verschiedene Sichtweisen funktionieren nicht mehr!

Programmierer: Grundverständnis techn. Funktionsweise und Rechnerarchitektur

Hardwaredesigner: Programmabläufe und Betriebssysteme wichtig!

A. Mäder

8



Trennung von Software und Hardware? (cont.)

Konsequenz

verschiedene Sichtweisen funktionieren nicht mehr!

Programmierer: Grundverständnis techn. Funktionsweise und Rechnerarchitektur

Hardwaredesigner: Programmabläufe und Betriebssysteme wichtig!

Motivation für Rechnerstrukturen und Betriebssysteme



Trennung von Software und Hardware? (cont.)

Konsequenz

verschiedene Sichtweisen funktionieren nicht mehr!

Programmierer: Grundverständnis techn. Funktionsweise und Rechnerarchitektur

Hardwaredesigner: Programmabläufe und Betriebssysteme wichtig!

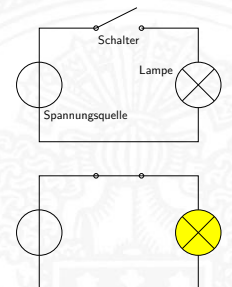
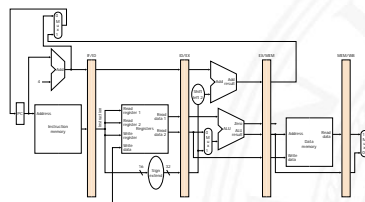
Motivation für Rechnerstrukturen und Betriebssysteme

⇒ Interaktion von SW und HW

```
#include <stdio.h>

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

- ▶ „performante“, „sichere“ Software programmieren
- ▶ Systemsicht / Variantenvielfalt von Mikroprozessorsystemen
- ▶ Bewertung von Trends und Perspektiven





Konsequenz

verschiedene Sichtweisen funktionieren nicht mehr!

Programmierer: Grundverständnis techn. Funktionsweise und Rechnerarchitektur

Hardwaredesigner: Programmabläufe und Betriebssysteme wichtig!

Motivation für Rechnerstrukturen und Betriebssysteme

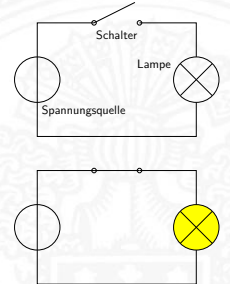
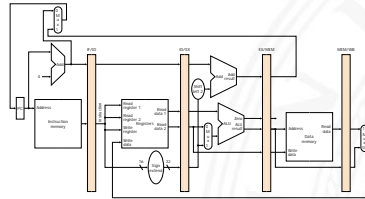
⇒ Interaktion von SW und HW

```
#include <stdio.h>

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

- ▶ „performante“, „sichere“ Software programmieren
- ▶ Systemsicht / Variantenvielfalt von Mikroprozessorsystemen
- ▶ Bewertung von Trends und Perspektiven

⇒ Wie funktioniert ein Computer?



▶ Formel $n! = \prod_{n=1}^n n$ $0! = 1$

▶ C Code */* Fakultät... - nicht rekursiv */*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}

int main(int argc, char **argv)
{ int arg;

  if ((argc != 2) || ((arg = atoi(argv[1])) < 0))
  { printf("usage: \"fak <n>\", n >= 0\n");
    exit(1);
  }

  printf("%d! = %d\n", arg, fak(arg));
  exit(0);
}
```



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4! $\frac{n = \quad}{res = \quad}$



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4! $\frac{n = 4}{res = \quad}$



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4! $\frac{n = 4}{res = 1}$



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4! $\frac{n = 4}{res = 1}$



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4! $\frac{n = 4}{res = 1 \quad 4}$



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4! $\frac{n = 4 \quad 3}{res = 1 \quad 4}$



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4! $\frac{n = 4 \quad 3}{res = 1 \quad 4}$



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4! $\frac{n = 4 \quad 3}{res = 1 \quad 4 \quad 12}$



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4! $\frac{n = 4 \quad 3 \quad 2}{res = 1 \quad 4 \quad 12}$



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4! $\frac{n = 4 \quad 3 \quad 2}{res = 1 \quad 4 \quad 12}$



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4!

n =	4	3	2	
res =	1	4	12	24



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4!

n =	4	3	2	1
res =	1	4	12	24



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

Ablauf 4!

n = 4	3	2	1
res = 1	4	12	24



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

- *Wie werden Programmanweisungen schrittweise abgearbeitet?*



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n  = n - 1;
  }
  return res;
}
```

- ▶ *Wie werden Programmanweisungen schrittweise abgearbeitet?*
- ▶ *Wie wird Information (hier Zahlen) technisch dargestellt?*



Wie funktioniert ein Computer?

```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n  = n - 1;
  }
  return res;
}
```

- ▶ *Wie werden Programmanweisungen schrittweise abgearbeitet?*
- ▶ *Wie wird Information (hier Zahlen) technisch dargestellt?*
- ▶ *Welche technischen Komponenten werden benötigt?
Wie funktionieren sie? Wie kann damit gerechnet werden?*



```
unsigned int fak(unsigned int n)
{ unsigned int res = 1;
  while (n > 1)
  { res = n * res;
    n = n - 1;
  }
  return res;
}
```

- ▶ *Wie werden Programmanweisungen schrittweise abgearbeitet?*
- ▶ *Wie wird Information (hier Zahlen) technisch dargestellt?*
- ▶ *Welche technischen Komponenten werden benötigt?
Wie funktionieren sie? Wie kann damit gerechnet werden?*
- ▶ *Welche Mechanismen sorgen dafür, dass mehrere Anwendungen
(und mehrere Benutzer) „gleichzeitig“ arbeiten?*



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)
 - ▶ Übertragungsraten / Bandbreite (Netzwerke)
 2. neue Entwurfparadigmen und -werkzeuge
- ⇒ Möglichkeiten und Anwendungsfelder
- ⇒ Produkte und Techniken

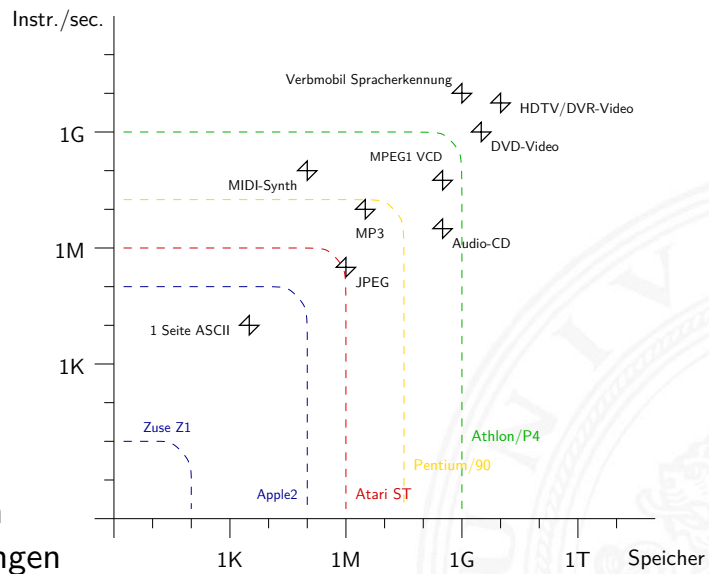


Fortschritt (cont.)

Kriterien / Maßgrößen

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

⇒ jede Rechnergeneration erlaubt neue Anwendungen



Beispiel: technischer Fortschritt

Hardware für den Mondflug





Hardware für den Mondflug



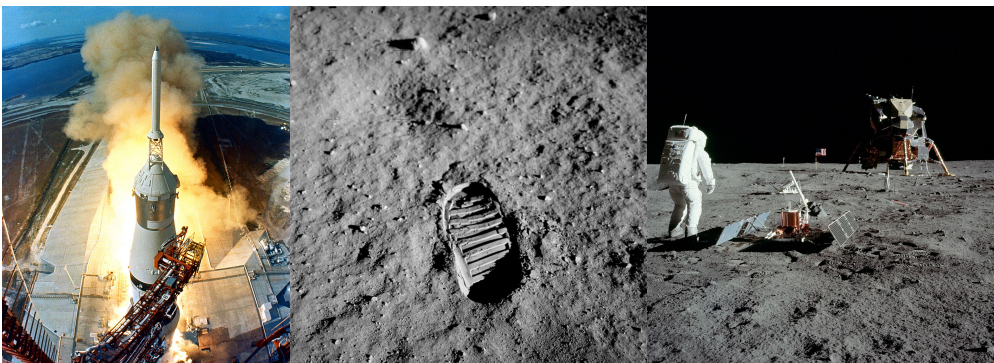
... aber nicht 2023

A. Mäder

13



Beispiel: Apollo 11 (1969)



- ▶ bernd-leitenberger.de/computer-raumfahrt1.shtml
- ▶ history.nasa.gov/computers/contents.html
- ▶ en.wikipedia.org/wiki/Apollo_Guidance_Computer
- ▶ en.wikipedia.org/wiki/IBM_System/360
www.computerhistory.org/revolution/mainframe-computers/7

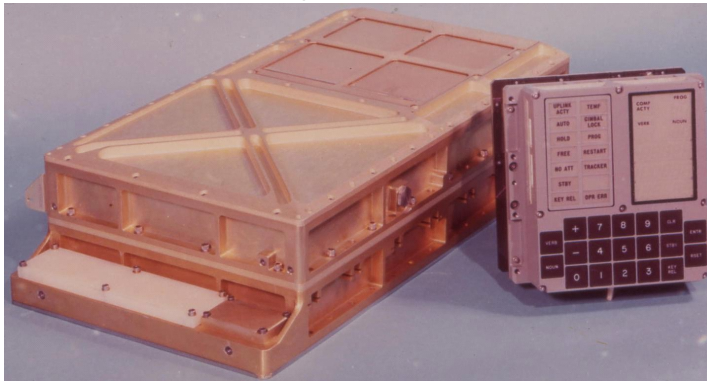
A. Mäder

14



vor 54 Jahren (cont.)

1. Bordrechner: AGC (Apollo Guidance Computer)



Dryden Flight Research Center EC96-45408-1 Photographed 1996
Apollo computer interface box used in the F-8 Digital Fly-By-Wire program. NASA photo by Dennis Taylor

- ▶ Dimension 61 × 32 × 15,0 cm 31,7 kg
20 × 20 × 17,5 cm 8,0 kg
- ▶ Taktfrequenz: 1,024 MHz
- ▶ 16-bit Worte, nur Festkomma
- ▶ Speicher ROM 36 KWorte 72 KByte
RAM 2 KWorte 4 KByte

Addition: $\approx 20 \mu s$, mehrere Takte

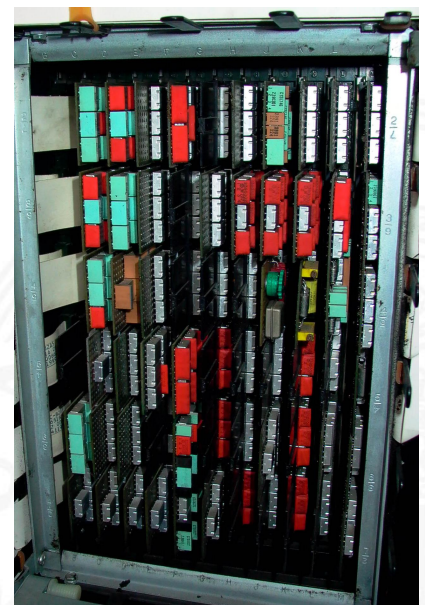
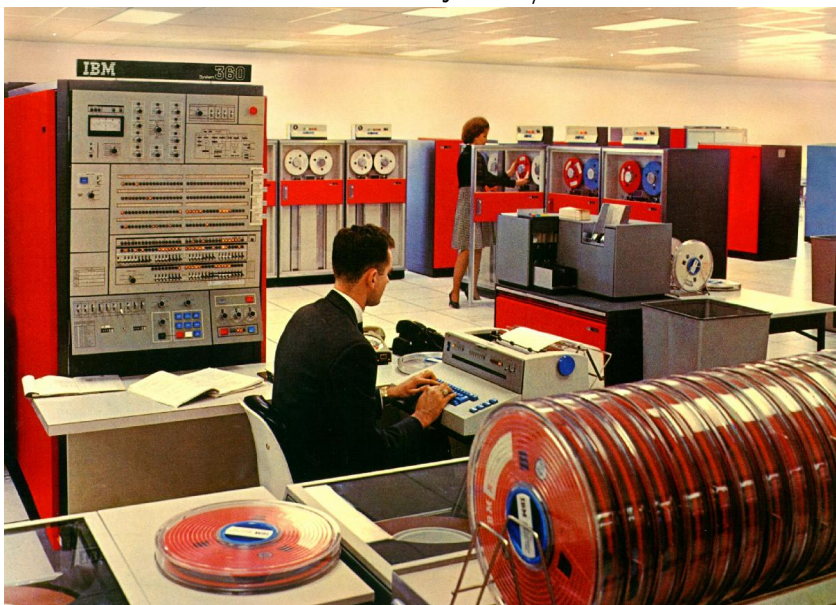
Zykluszeit: 11,7 μs , 12 Takte, 85,3 KHz



vor 54 Jahren (cont.)

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

▶ YouTube Video





vor 54 Jahren (cont.)

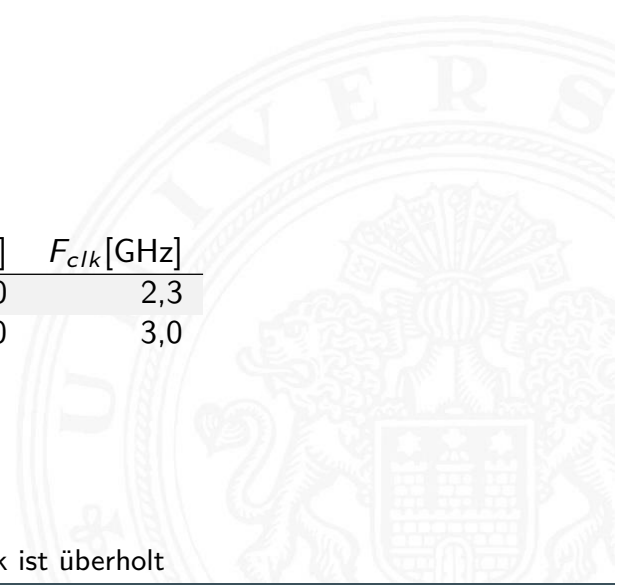
- ▶ je nach Ausstattung: Anzahl der „Schränke“
- ▶ Taktfrequenz: bis 5 MHz
- ▶ 32-bit Worte, 24-bit Adressraum (16 MByte)
- ▶ Speicherhierarchie: bis 1 MByte Hauptspeicher (1,3 MHz Zykluszeit)
- ▶ (eigene) Fließkomma Formate
- ▶ Rechenleistung: 0,7 Dhrystone MIPS



vor 54 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 ¹

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



¹Daten aktueller Systeme sind nicht verfügbar – Benchmark ist überholt



vor 54 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 ¹

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

⇒ Moore's Law

¹Daten aktueller Systeme sind nicht verfügbar – Benchmark ist überholt



Timeline: Rechengeschichte

- ???? 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)

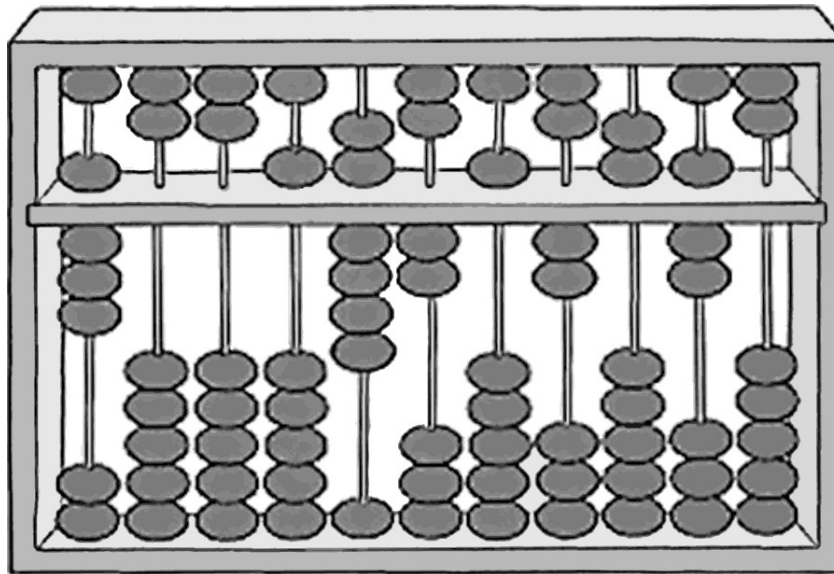


Abakus

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“

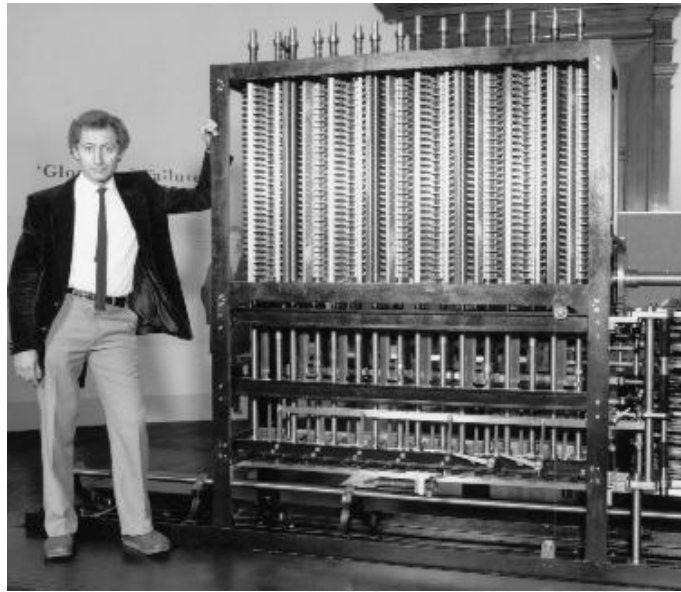
1673 Leibniz: Staffelwalze, Multiplikation/Division

1774 Philipp Matthäus Hahn: erste gebrauchsfähige '4-Spezies'-Maschine



Difference Engine

Charles Babbage 1822: Berechnung nautischer Tabellen

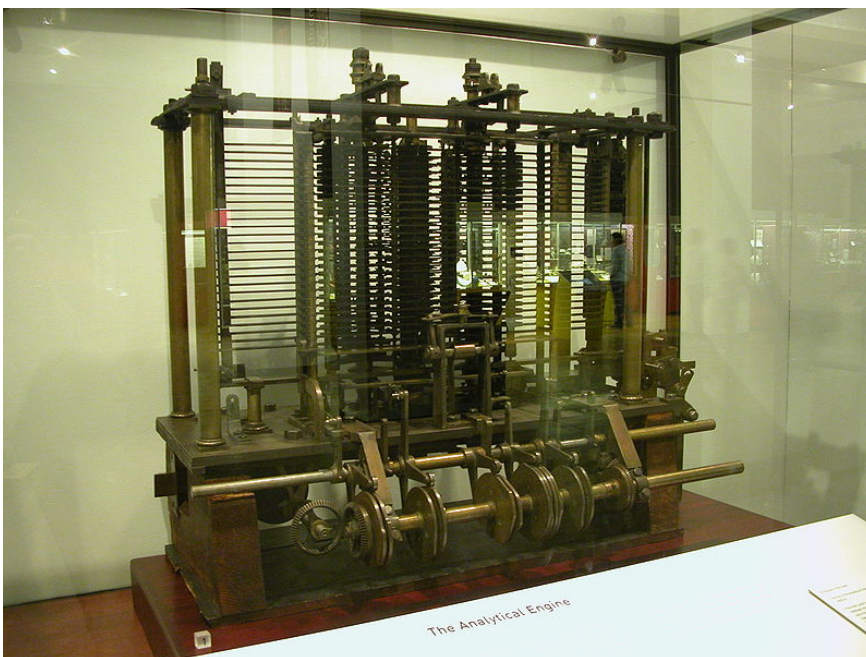


Original von 1832 und Nachbau von 1989, London Science Museum



Analytical Engine

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





Programmierbarkeit

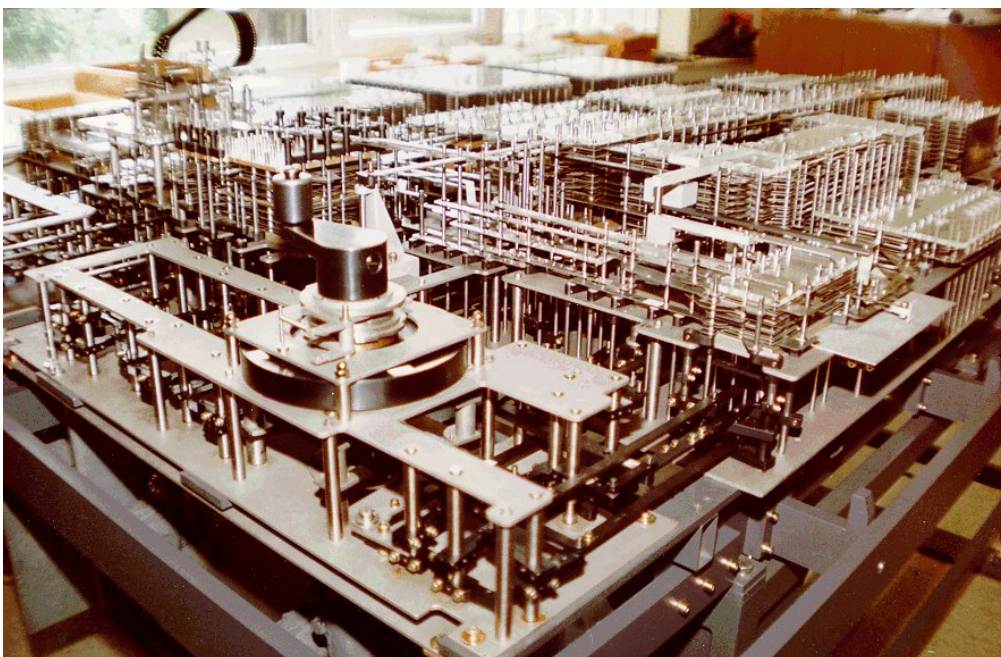
- ▶ nicht mehr eine Maschine für eine Aufgabe / ein Problem sondern **feste Hardware**
 - + problemspezifisch wird ein **flexibles Programm** entwickelt

 - ▶ schrittweises Problemlösen ⇒ Algorithmen
 - ▶ zentrale Paradigmen der Informatik
- ⇒ **von-Neumann Konzept**



Zuse Z1

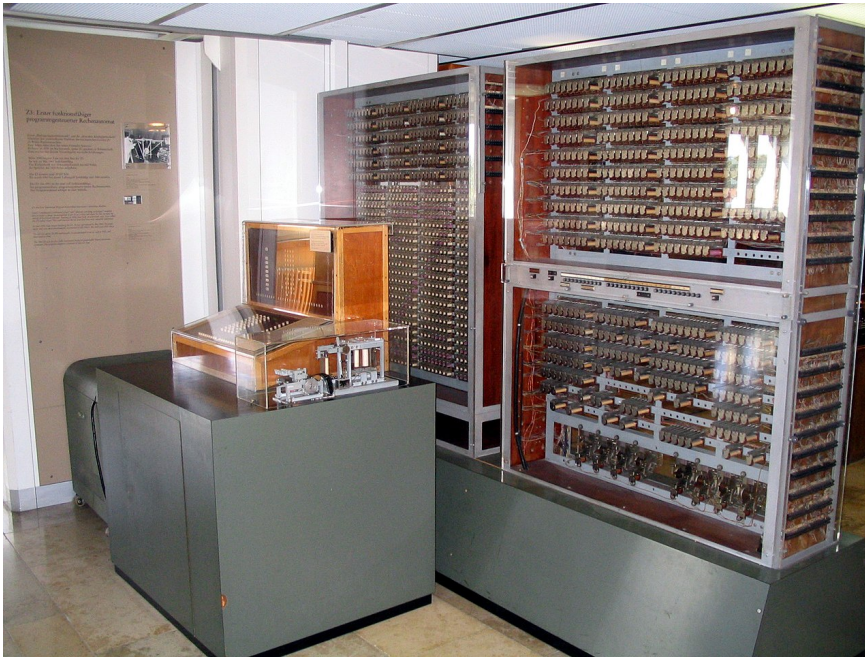
Konrad Zuse 1937: 64 Register, 22-bit, mechanisch, Lochfilm





Zuse Z3

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

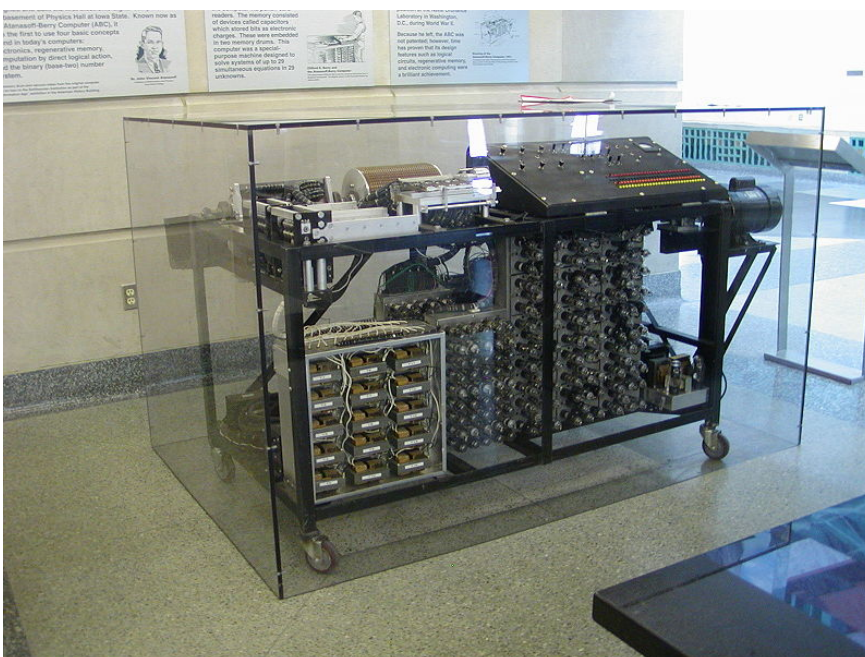


A. Mäder



Atanasoff-Berry Computer (ABC)

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



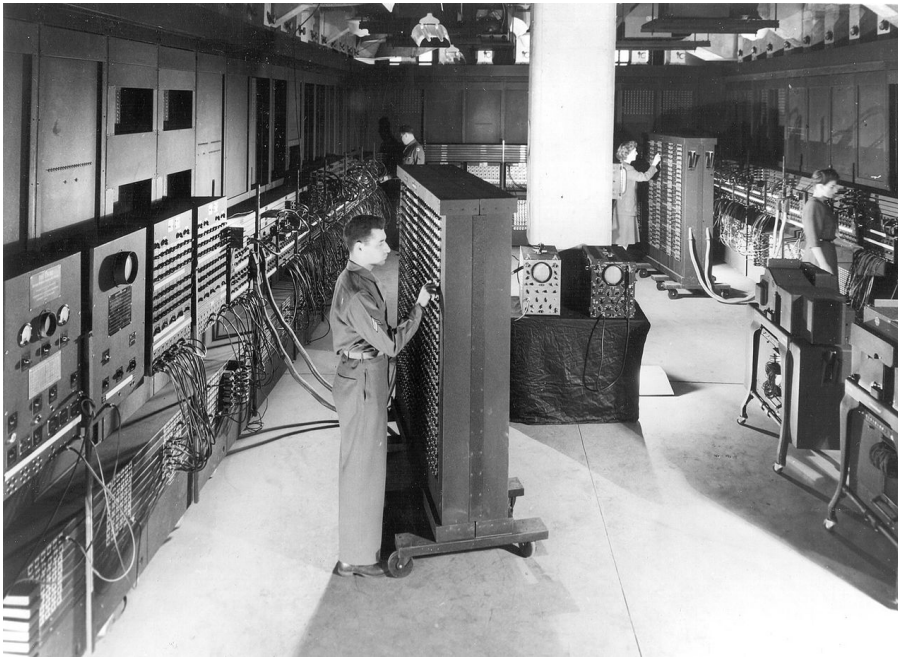
A. Mäder





ENIAC – Electronic Numerical Integrator and Computer

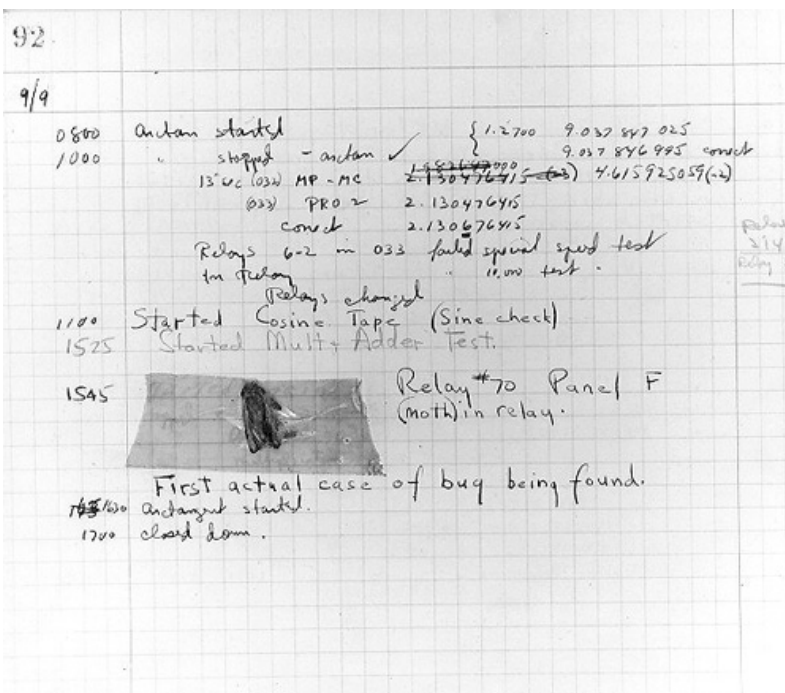
J. Mauchly & J.P. Eckert, 1946: Röhren, Steckbrett-Programm



A. Mäder



First computer bug



A. Mäder





EDVAC

J. Mauchly, J.P. Eckert & J. von Neumann, 1949: Röhren, speicherprogrammiert

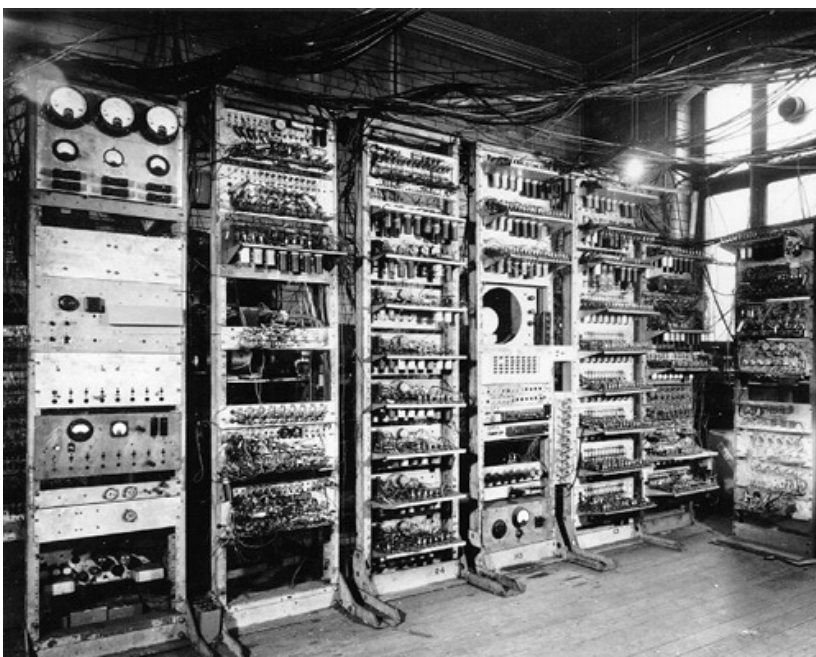


A. Mäder



Manchester Mark-1

F.C. Williams & T. Kilburn, 1949: Trommelspeicher, Indexregister



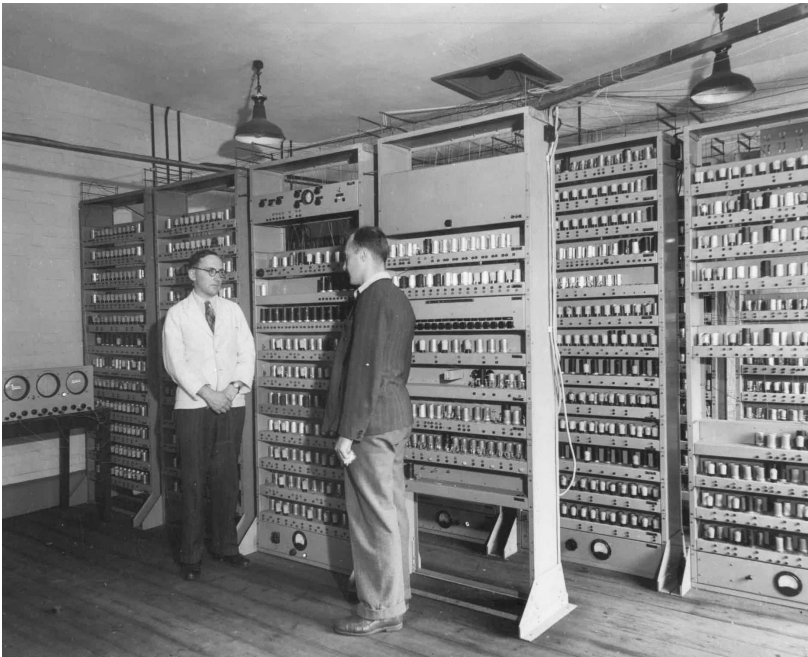
A. Mäder





Manchester EDSAC

M. Wilkes 1951: Mikroprogrammierung, Unterprogramme, speicherprogrammiert



A. Mäder



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 **CISC**-Rechner: VAX, 68000, 8086 etc. (**C**omplex **I**nstruction **S**et **C**omputer)



erste Betriebssysteme

1.1 Einführung - Exkurs: Geschichte

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ erste Rechner jeweils nur von einer Person benutzt
 - ▶ Anwender = Programmierer = Operator
 - ▶ Programm laden, ausführen, Fehler suchen usw.
- ⇒ Maschine wird nicht gut ausgelastet
- ⇒ Anwender mit lästigen Details überfordert

Einführung von Betriebssystemen

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

A. Mäder

33



zweite Generation: Transistoren

1.1 Einführung - Exkurs: Geschichte

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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 à 18-bit)
 - ▶ 200 KHz Taktfrequenz
 - ▶ 120 000 \$
 - ▶ Grafikdisplay: erste Computerspiele
 - ▶ Nachfolger PDP-8: 16 000 \$
 - ▶ erstes Bussystem
 - ▶ 50 000 Stück verkauft



A. Mäder

34

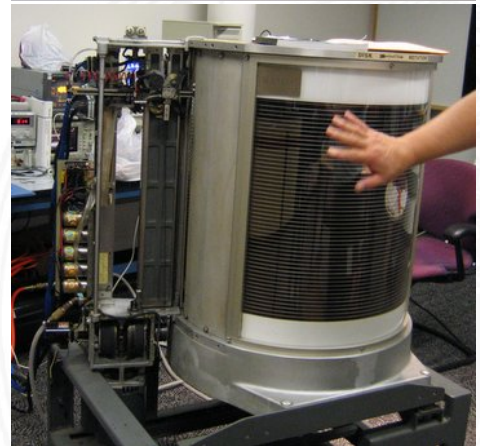


Massenspeicher bei frühen Computern

- ▶ Lochkarten
- ▶ Lochstreifen
- ▶ Magnetband

- ▶ Magnettrommel
- ▶ Festplatte

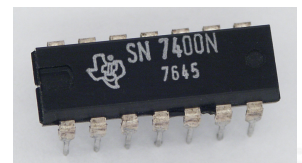
IBM 350 RAMAC (1956)
5 MByte, 600 ms Zugriffszeit



de.wikibooks.org/wiki/Computerhardware_für_Anfänger



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



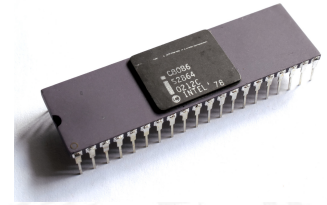
- ▶ IBM Serie-360: viele Maschinen, ein einheitlicher Befehlssatz
- ▶ volle Softwarekompatibilität

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

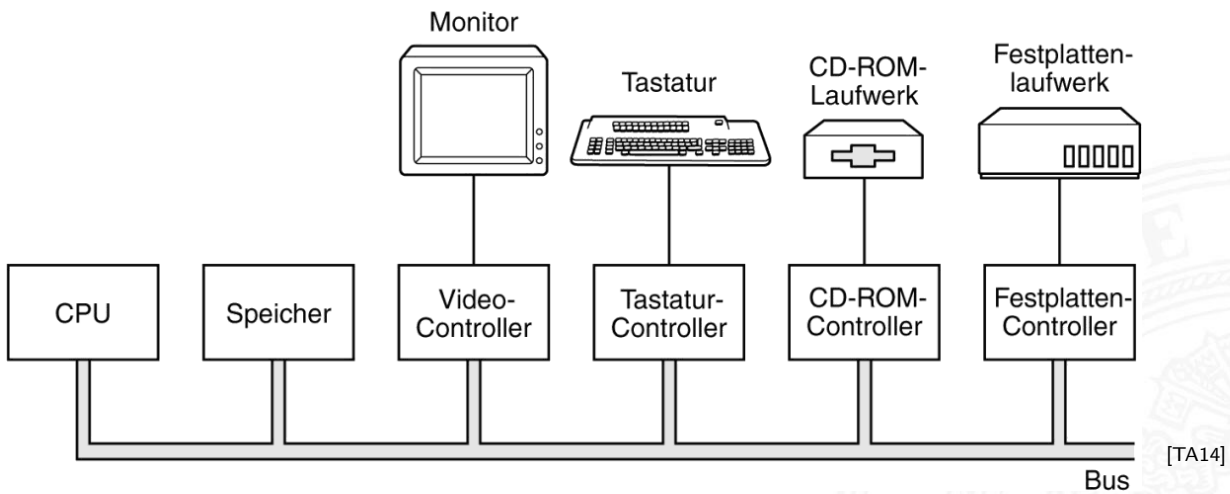


Xerox Alto: first workstation (1973)





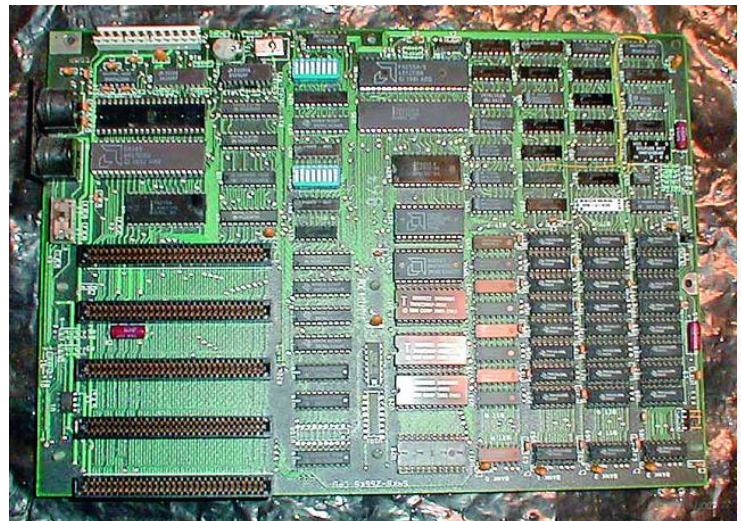
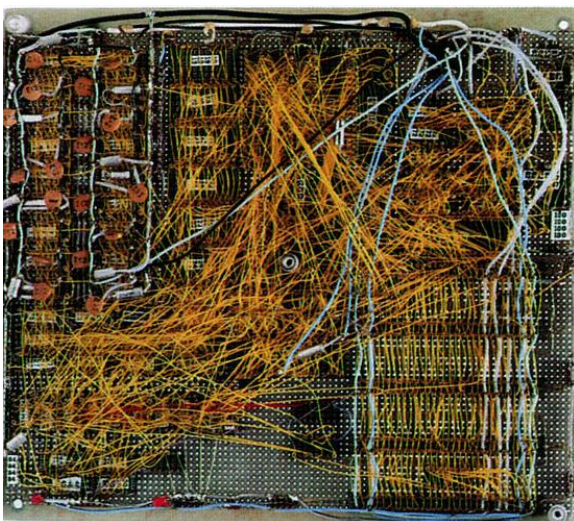
Personal Computer: Aufbau des IBM PC (1981)



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

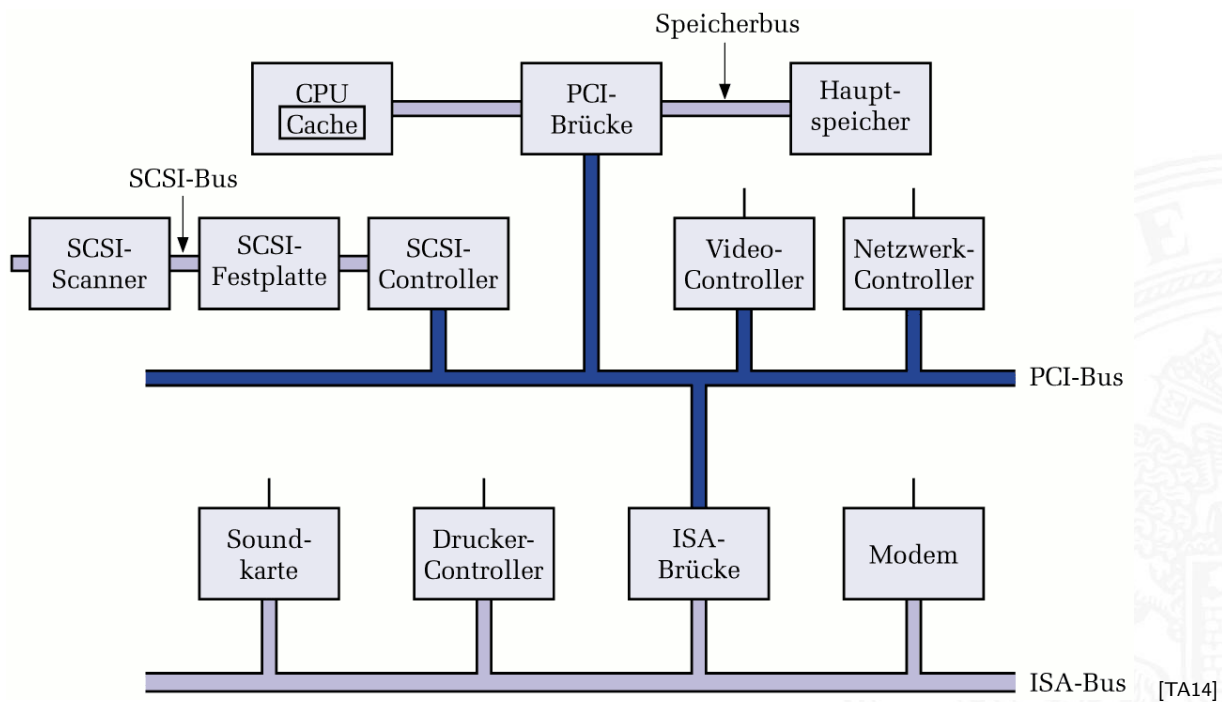


PC Prototyp (1981) und Hauptplatine

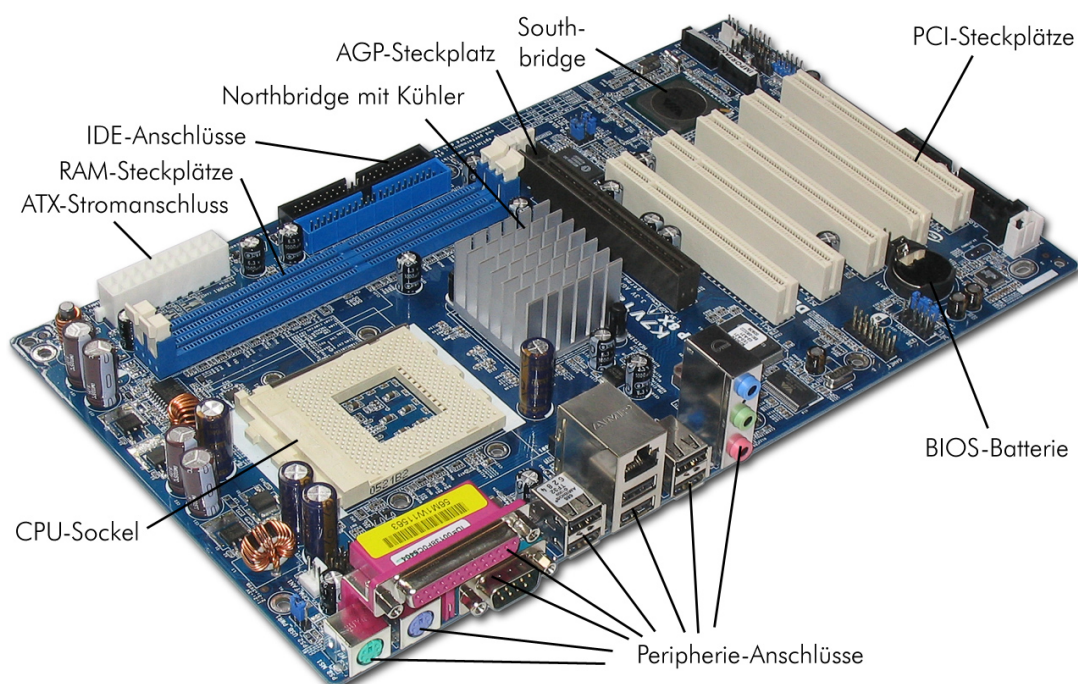




Aufbau mit PCI-Bus (2000)

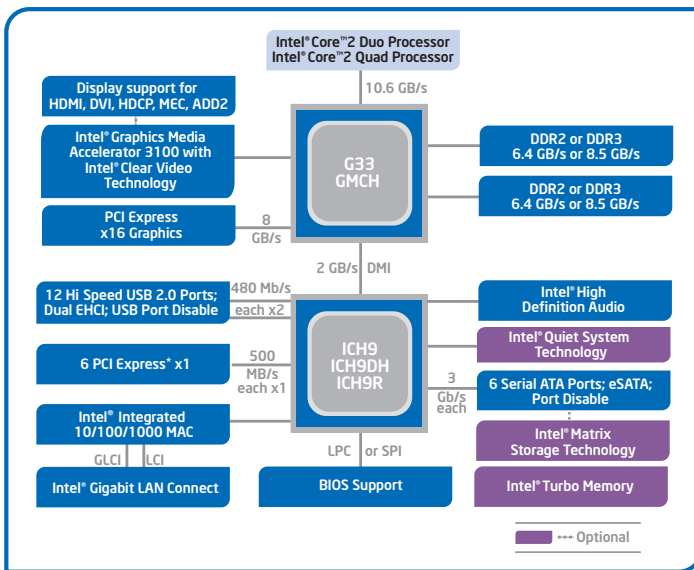


Hauptplatine (2005)





Aufbau (2010)

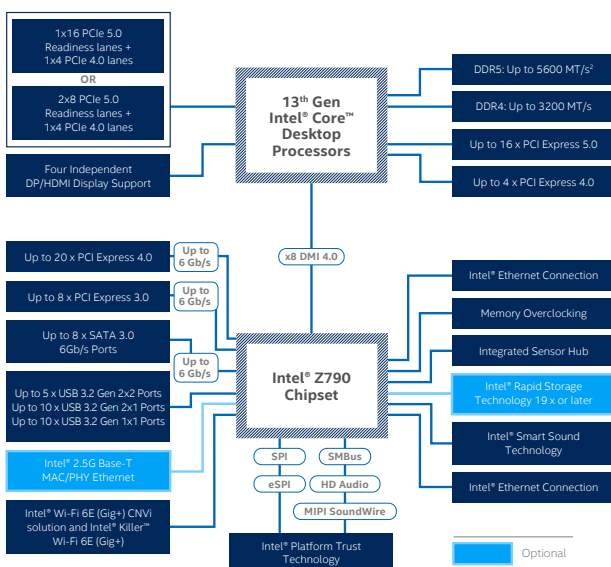


Intel ark.intel.com

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



Aufbau (2022)



Intel ark.intel.com

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



▶ 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

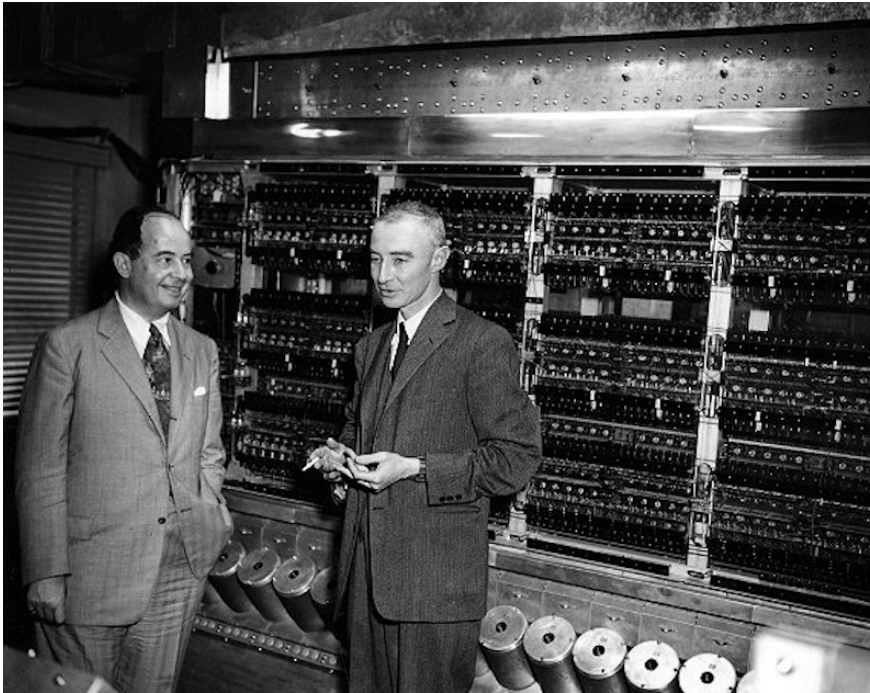
Typ	Preis [\$]	Beispielanwendung
Wegwerfcomputer	0,5	Glückwunschkarten
Mikrocontroller	5	Uhren, Geräte, Autos
Mobile Computer und Spielkonsolen	50	Smartphones, Tablets, Heimvideospiele
Personalcomputer	500	Desktop- oder Notebook-Computer
Server	5 000	Netzwerkserver
Workstation Verbund	50 000 – 500 000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



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



von-Neumann Rechner: IAS Computer

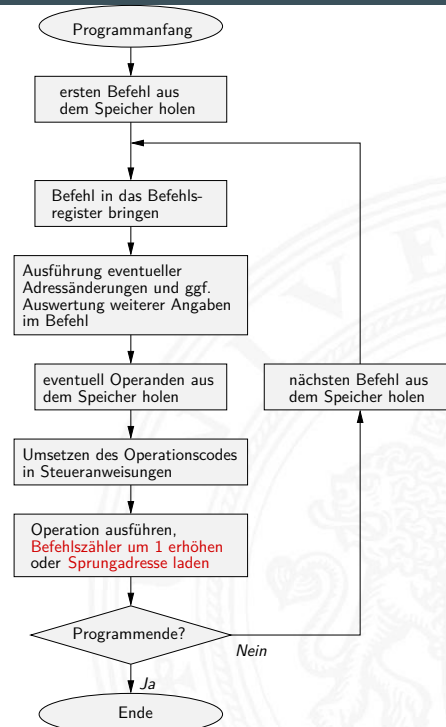


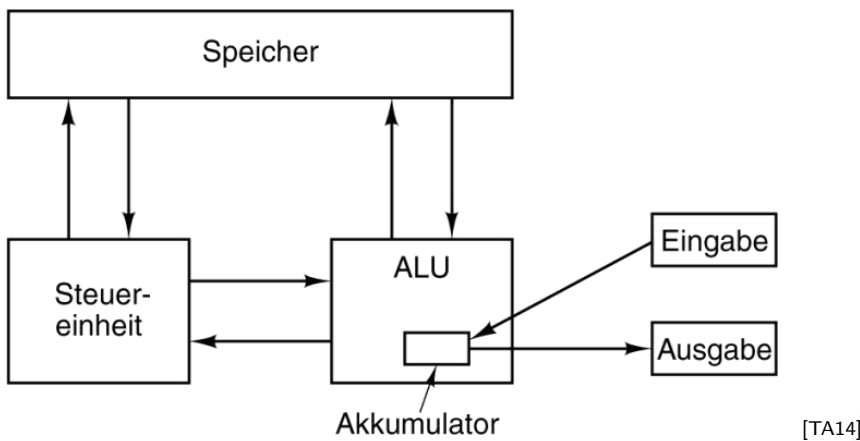
John von Neumann, R. J. Oppenheimer, IAS Computer Princeton computerhistory.org



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



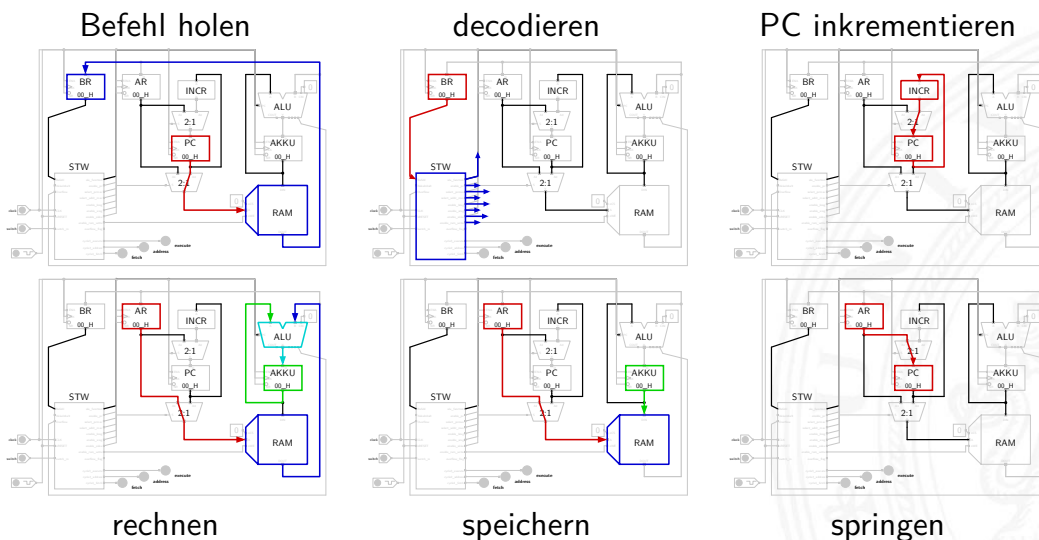


Fünf zentrale Komponenten:

- ▶ 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

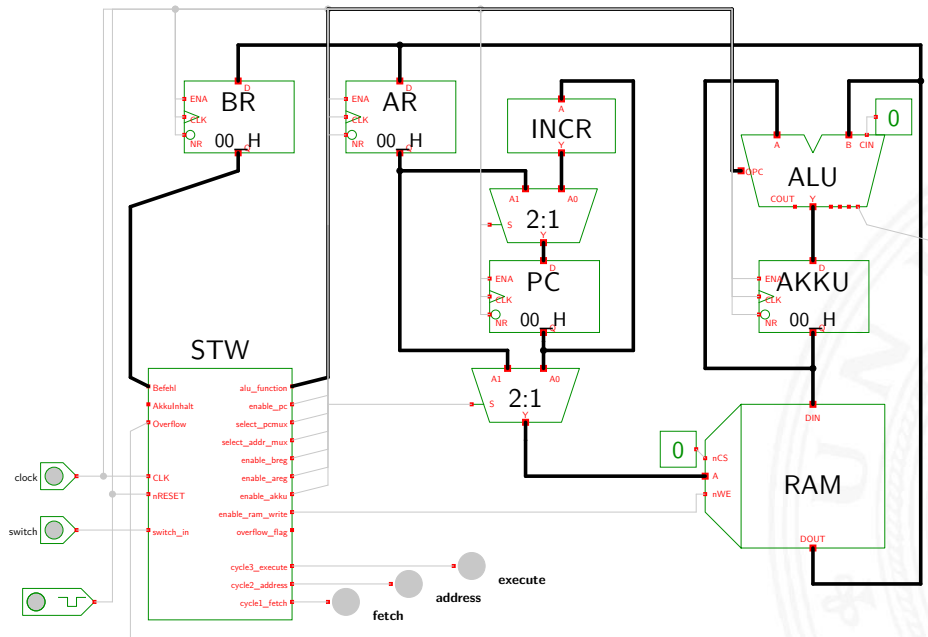




Beispiel: PRIMA (die primitive Maschine)

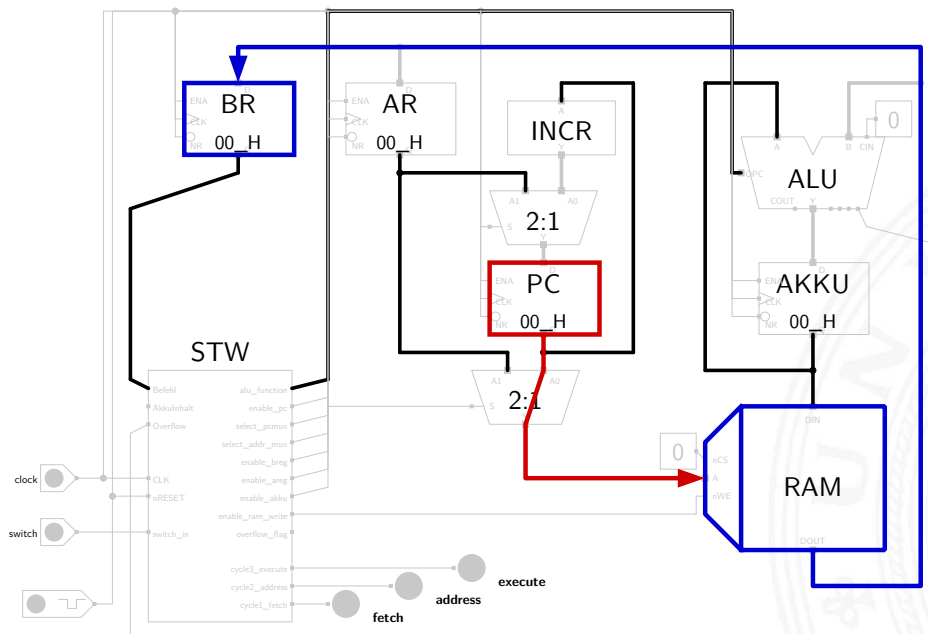
► ein (minimaler) 8-bit von-Neumann Rechner

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



PRIMA: Befehl holen

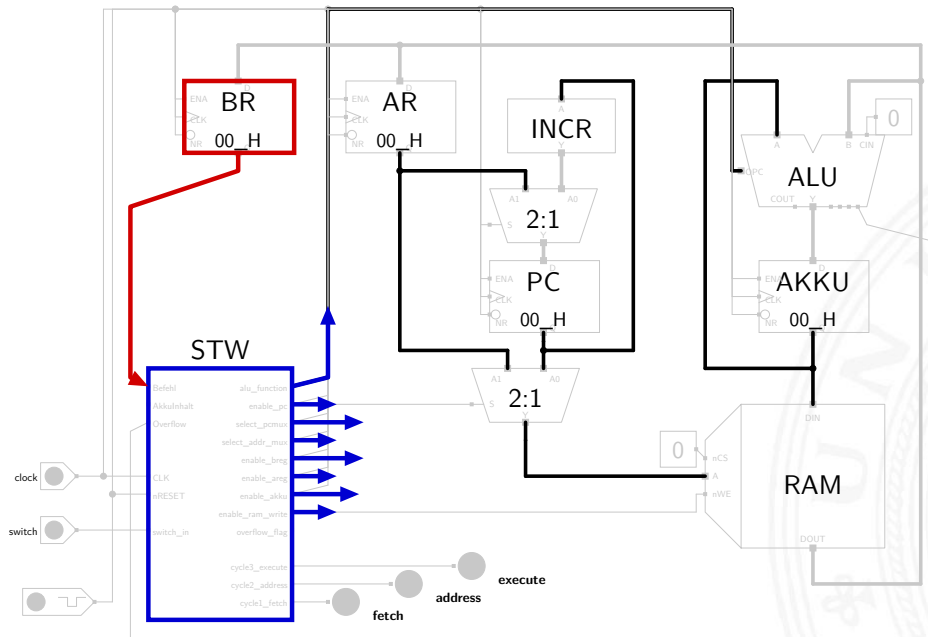
► $BR = RAM[PC]$





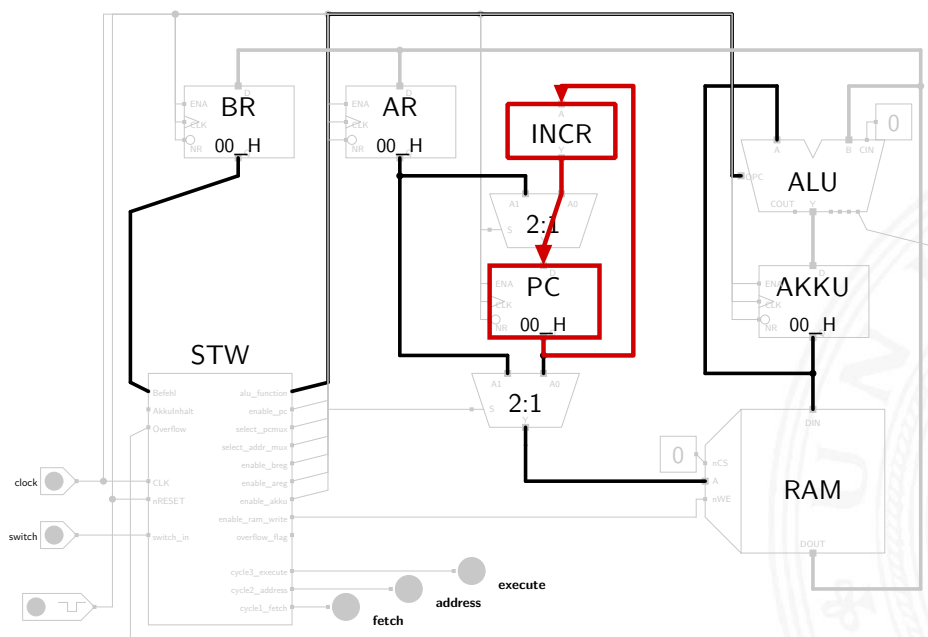
PRIMA: decodieren

- Steuersignale = decode(BR)



PRIMA: PC inkrementieren

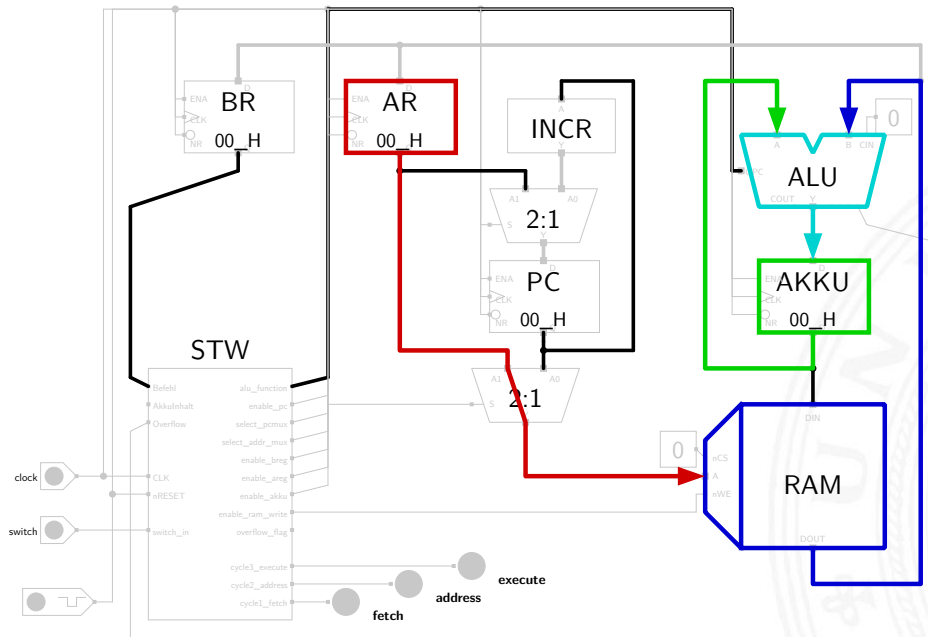
- $PC = PC + 1$





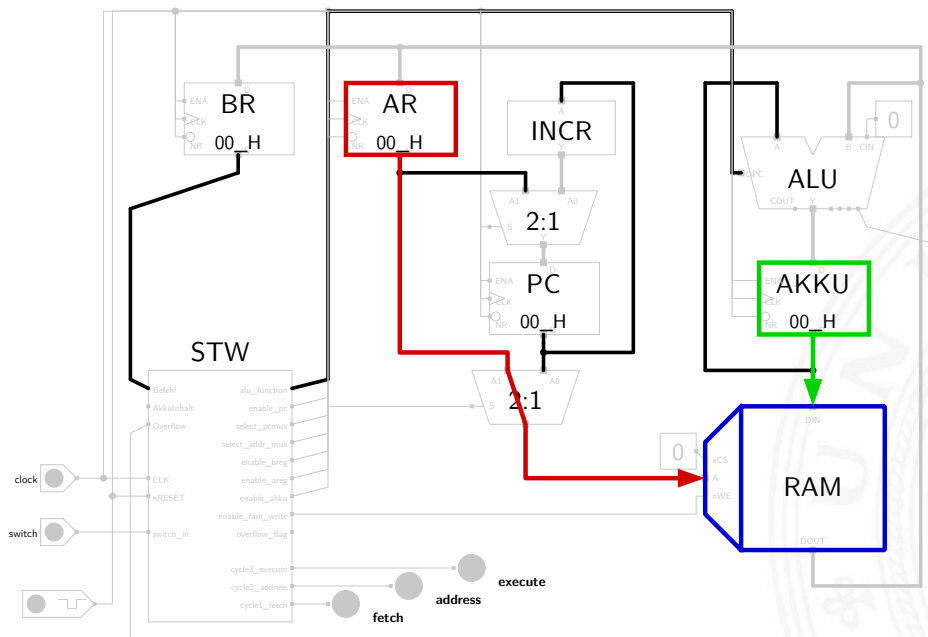
PRIMA: rechnen

► Akku = Akku + RAM[AR]



PRIMA: speichern

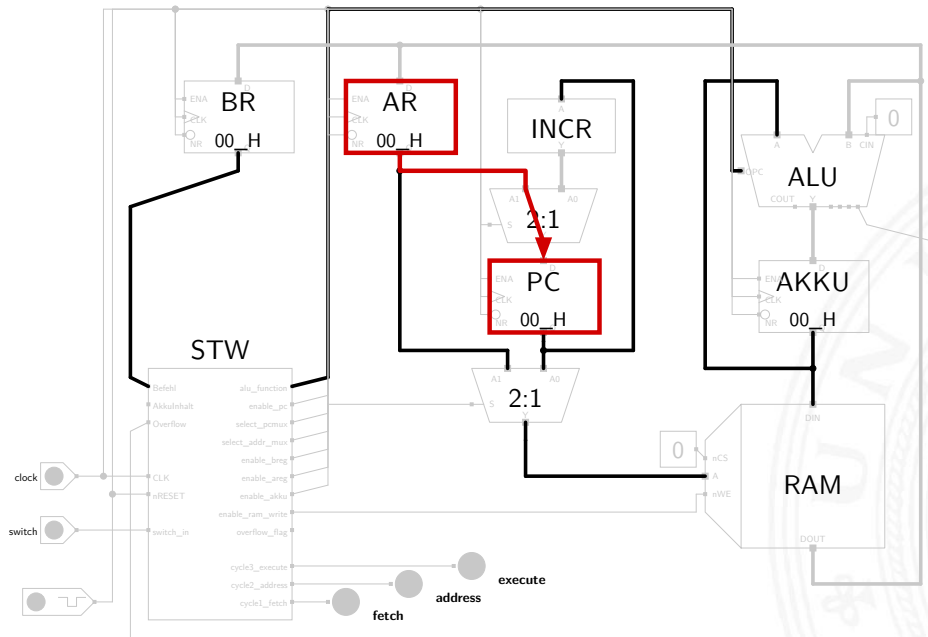
► RAM[AR] = Akku





PRIMA: springen

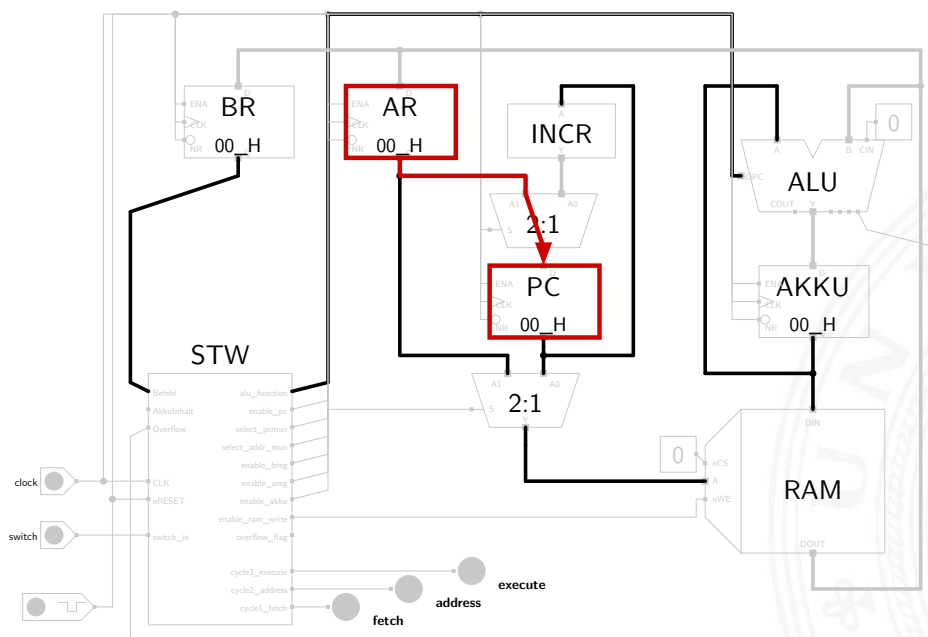
► PC = AR



PRIMA: springen

► PC = AR

später dazu mehr...





Moore's Law

- ▶ bessere Technologie ermöglicht immer kleinere Transistoren
- ▶ Kosten (Material/Entwurf/Fertigung) sind proportional zur Chipfläche
- ⇒ bei gleicher Funktion kleinere und billigere Chips
- ⇒ bei gleicher Größe leistungsfähigere Chips

Moore's Law

Gordon Moore, Mitgründer von Intel, 1965

Speicherkapazität von ICs vervierfacht sich alle drei Jahre

- ⇒ schnelles **exponentielles Wachstum**
 - ▶ klares Kostenoptimum bei hoher Integrationsdichte
 - ▶ trifft auch auf Prozessoren zu



Moore's Law (cont.)

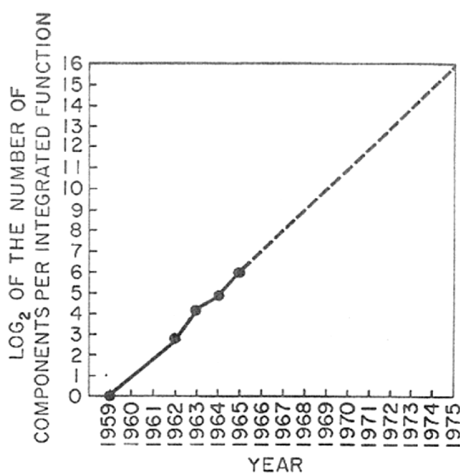


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

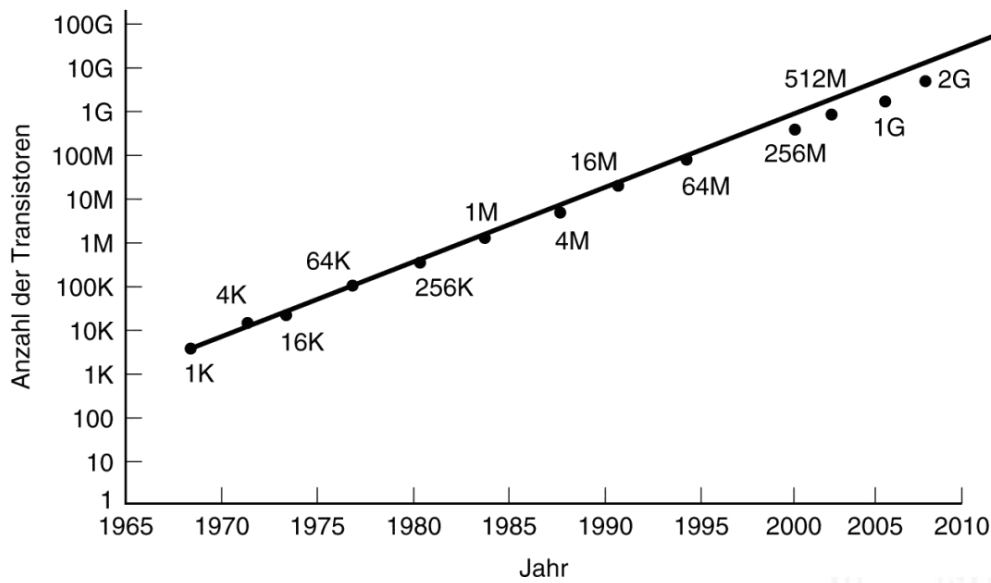
Gordon Moore, 1965, [Moo65]:
Cramming more components onto integrated circuits

Wird das so weitergehen?

- ▶ Vorhersage gilt immer noch
- ▶ „IRDS“ Prognosen bis zum Jahr 2037 [IRDS22]



Moore's Law: Transistoren pro Speicherchip

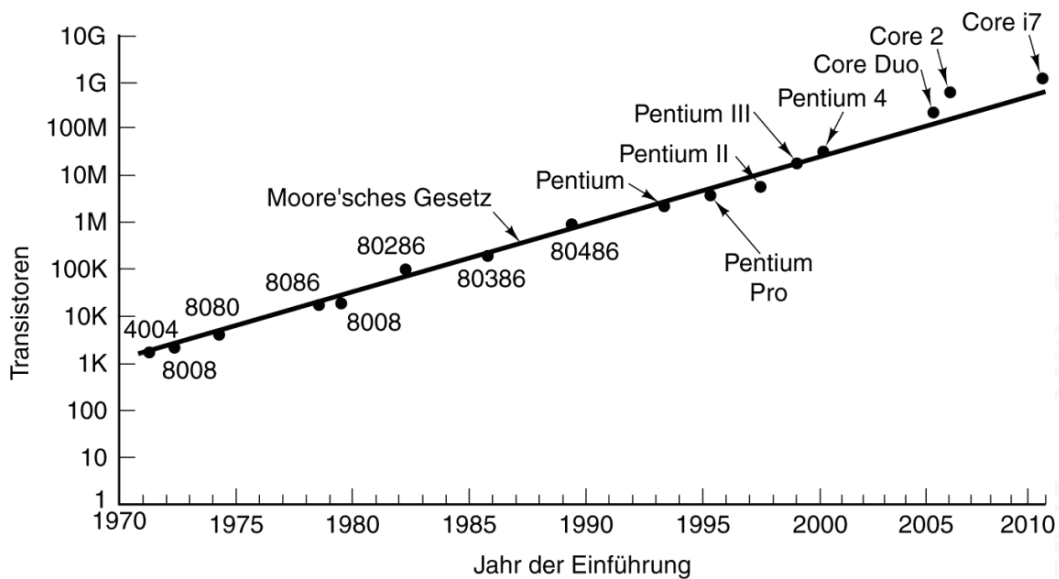


[TA14]

- ▶ Vorhersage: 60% jährliches Wachstum der Transistoranzahl pro IC
Verdopplung alle 18 Monate (12...24 Monate)



Moore's Law: Evolution der Prozessoren



[TA14]



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



[Intel] Intel Innovation 2023



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

Transistoren pro IC (monolithisch)

Modell		Typ	Jahr	Trans. [Mrd.]
M2 Max	Apple	CPU	2023	67,0



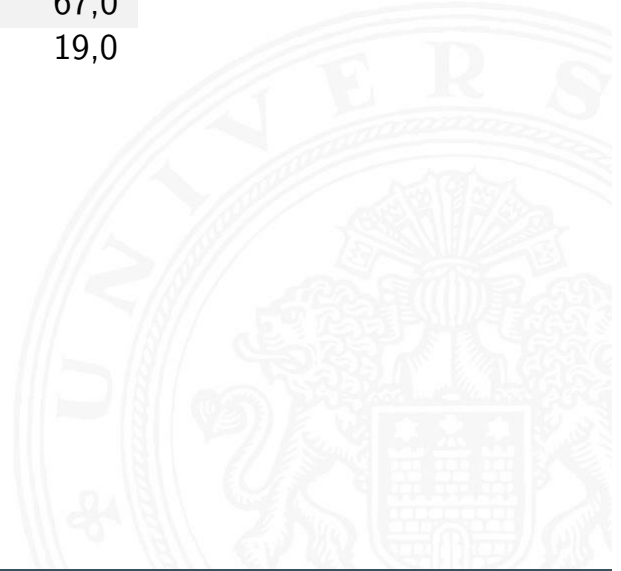
Moore's Law: Evolution der Prozessoren (cont.)

1.4 Einführung - Moore's Law

64-040 Rechnerstrukturen und Betriebssysteme

Transistoren pro IC (monolithisch)

Modell		Typ	Jahr	Trans. [Mrd.]
M2 Max	Apple	CPU	2023	67,0
A17	Apple	SOC	2023	19,0



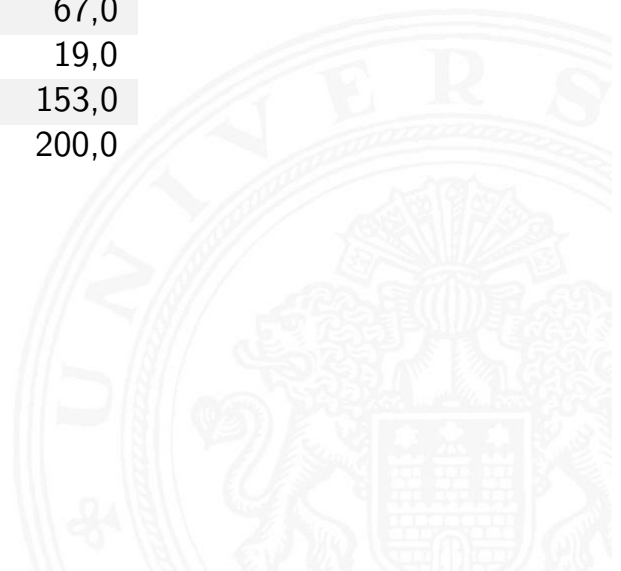
Moore's Law: Evolution der Prozessoren (cont.)

1.4 Einführung - Moore's Law

64-040 Rechnerstrukturen und Betriebssysteme

Transistoren pro IC (monolithisch)

Modell		Typ	Jahr	Trans. [Mrd.]
M2 Max	Apple	CPU	2023	67,0
A17	Apple	SOC	2023	19,0
MI300X	AMD	GPU	2023	153,0
GH200	Nvidia	GPU	2023	200,0

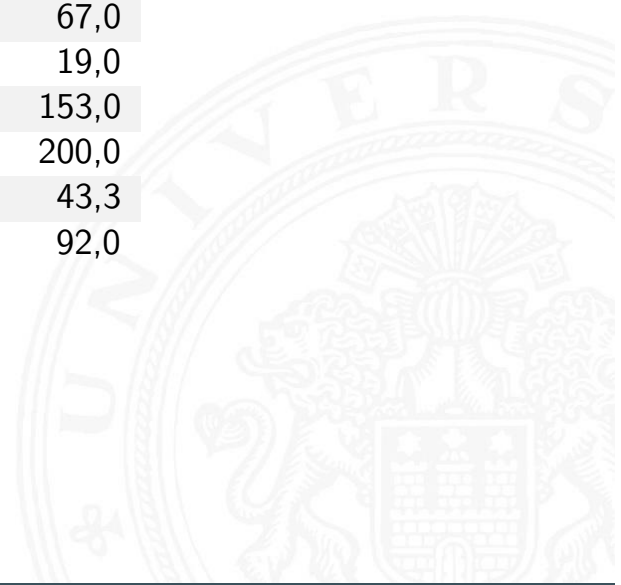




Moore's Law: Evolution der Prozessoren (cont.)

Transistoren pro IC (monolithisch)

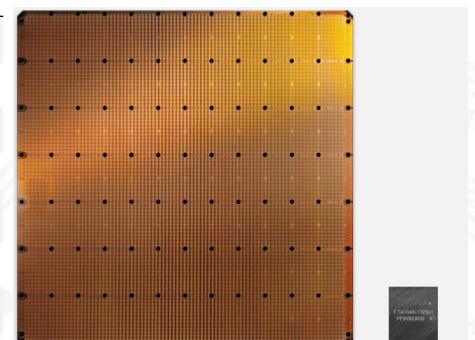
Modell		Typ	Jahr	Trans. [Mrd.]
M2 Max	Apple	CPU	2023	67,0
A17	Apple	SOC	2023	19,0
MI300X	AMD	GPU	2023	153,0
GH200	Nvidia	GPU	2023	200,0
Stratix 10	Intel (Altera)	FPGA	2019	43,3
VP 1802	AMD (Xilinx)	FPGA	2021	92,0



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

Transistoren pro IC (monolithisch)

Modell		Typ	Jahr	Trans. [Mrd.]
M2 Max	Apple	CPU	2023	67,0
A17	Apple	SOC	2023	19,0
MI300X	AMD	GPU	2023	153,0
GH200	Nvidia	GPU	2023	200,0
Stratix 10	Intel (Altera)	FPGA	2019	43,3
VP 1802	AMD (Xilinx)	FPGA	2021	92,0
WSE-2	Cerebras	WSc	2021	2 600,0



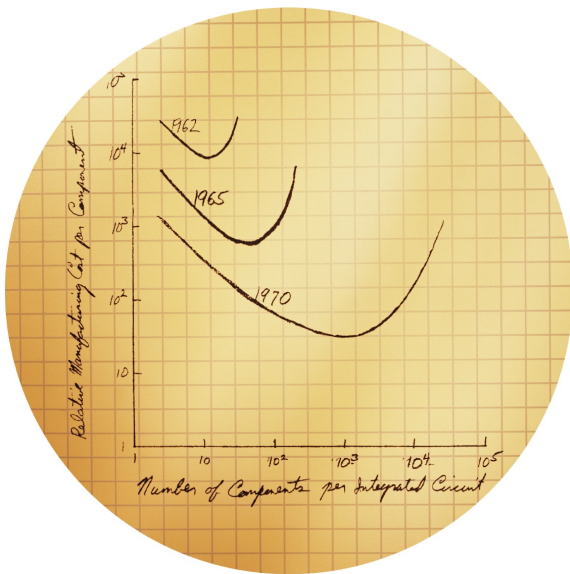
Cerebras WSE-2
46,225mm² Silicon
2.6 Trillion transistors

Largest GPU
826mm² Silicon
54.2 Billion transistors

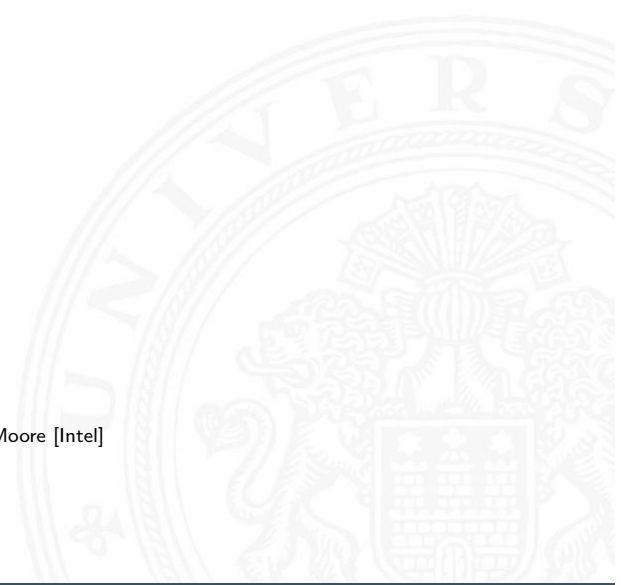
[www.cerebras.net]



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 BlueGene/Q Sequoia	(Power BQC 16C 1,6GHz)		16 324,8	1 572 864
2013	NUDT Tianhe-2	(Xeon E5-2692 12C 2,2 GHz + Phi 31S1P)		33 862,7	3 120 000
2016	Sunway TaihuLight	(Sunway SW26010 260C 1,45 GHz)		93 014,6	10 649 600
2018	Summit	(IBM Power9 22C 3,07 GHz + NVIDIA GV100)		143 500,0	2 397 824
2021	Fugaku	(Fujitsu A64FX 48C 2,2 GHz)		442 010,0	7 630 848
2023	JUWELS [D #13]	(Epyc 7402 24C 2,8 GHz + NVIDIA A100)		44 120,0	449 280
2023	Frontier	(AMD Epyc 64C 2,0 GHz + Instinct)		1 194 000,0	8 699 904



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 BlueGene/Q Sequoia	(Power BQC 16C 1,6GHz)		16 324,8	1 572 864	7 890
2013	NUDT Tianhe-2	(Xeon E5-2692 12C 2,2 GHz + 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	Summit	(IBM Power9 22C 3,07 GHz + NVIDIA GV100)		143 500,0	2 397 824	9 783
2021	Fugaku	(Fujitsu A64FX 48C 2,2 GHz)		442 010,0	7 630 848	29 899
2023	JUWELS [D #13]	(Epyc 7402 24C 2,8 GHz + NVIDIA A100)		44 120,0	449 280	1 764
2023	Frontier	(AMD Epyc 64C 2,0 GHz + Instinct)		1 194 000,0	8 699 904	22 703

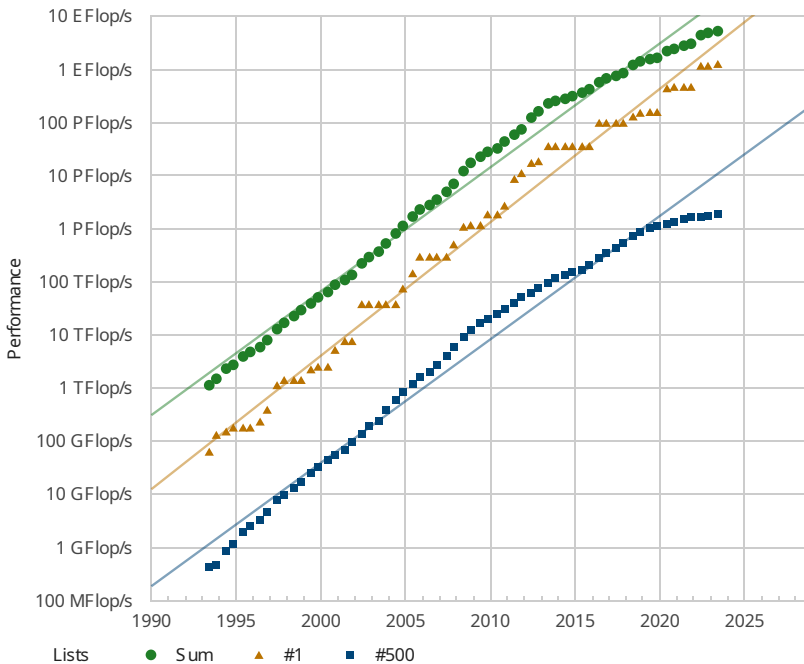


Leistungssteigerung der Spitzenrechner seit 1993 (cont.)

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

1.4 Einführung - Moore's Law

64-040 Rechnerstrukturen und Betriebssysteme



A. Mäder

67



Moore's Law: Aktuelle Trends

1.4 Einführung - Moore's Law

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Miniaturisierung schreitet weiter fort
- ▶ Taktraten physikalisch limitiert
 - ▶ Technologie / Strukturgrößen
 - ▶ Leistungsaufnahme \Rightarrow Spannungsversorgung + Kühlung

Entwicklungen

- ▶ seit 2011: CPU plus Grafikeinheit
- ▶ Integration mehrerer CPUs auf einem Chip (2-...64-Cores)
- ▶ Cache Speicher (SRAM als schneller Zwischenspeicher) auf dem Die
- ▶ Integration von Peripheriegeräten (Speicherinterface, PCIe, ...)
- ▶ heterogene Architekturen: „performance“ / „efficiency“
- ▶ multi-Chip Module (CPU, IO, Cache) + 3D Stapel
- ▶ **SoC**: „System on a chip“

A. Mäder

68



SoC: System on a chip

1.5 Einführung - System on a chip

64-040 Rechnerstrukturen und Betriebssysteme

Gesamtes System auf einem Chip integriert:

- ▶ ein oder mehrere Prozessoren, z.T. verschiedene Typen
 - ▶ hohe Rechenleistung
 - ▶ energieeffizient
- ⇒ z.B. ARM mit *big.LITTLE* Konzept; aktuell Laptop, PC: Intel, AMD
- ▶ Cache Hierarchie: 1st Level (D+I pro CPU) / 2nd (pro CPU) / 3rd (alle Kerne)
- ▶ dedizierte Prozessoren: Grafik, Video(de)codierung, DSP, AI ...
- ▶ Peripherieschnittstellen: Bussysteme, Speichercontroller
- ▶ Hauptspeicher (speziell "Embedded"), Grafikspeicher
- ▶ weitere Speicher für Medien/Netzwerkoperationen



SoC: System on a chip (cont.)

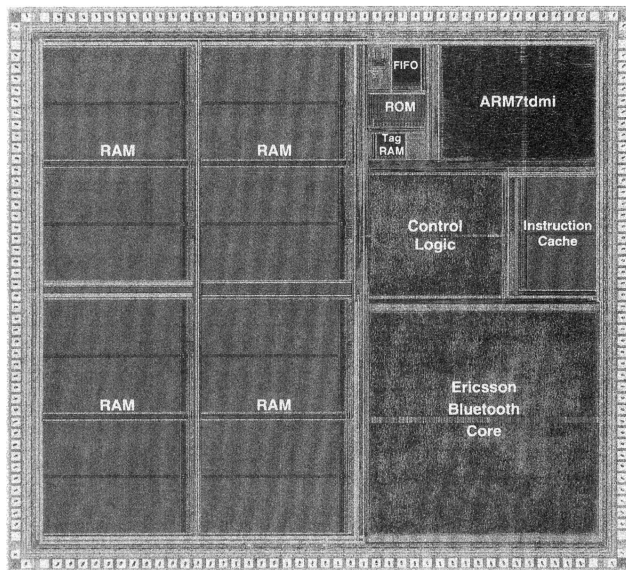
1.5 Einführung - System on a chip

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Peripherieblöcke nach Kundenwunsch konfiguriert:
 - ▶ Displaysteuerung: DP, HDMI ...
 - ▶ A/V-Schnittstellen: Kamera, Mikrofone, Audio ...
 - ▶ serielle und parallele Schnittstellen, SPI, I/O-Pins ...
 - ▶ Feldbusse: I²C, CAN ...
 - ▶ PC-like: USB, Firewire, SATA ...
 - ▶ Netzwerk kabelgebunden (Ethernet)
 - ▶ Funkschnittstellen: WLAN, Bluetooth, 5G ...
- ▶ Smartphones, Tablet-Computer, Medien-/DVD-Player, WLAN-Router, NAS-/Home-Server ...



▶ Bluetooth-Controller (2000)



[Fur00]

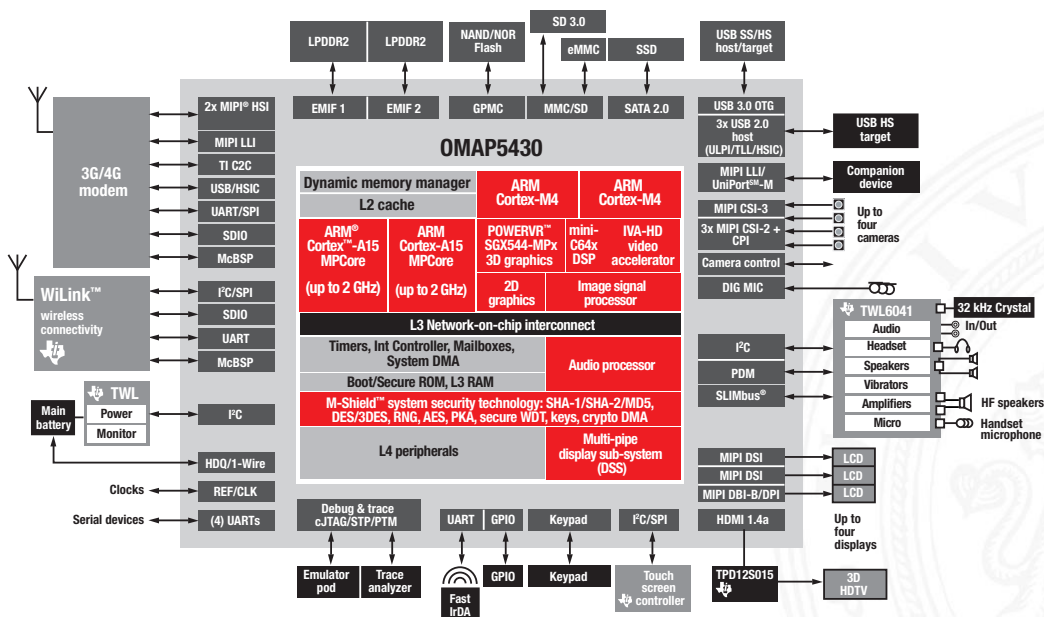
© VLSI Technology, Inc.

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



▶ Texas Instruments OMAP 5430 (2011)

[TI]

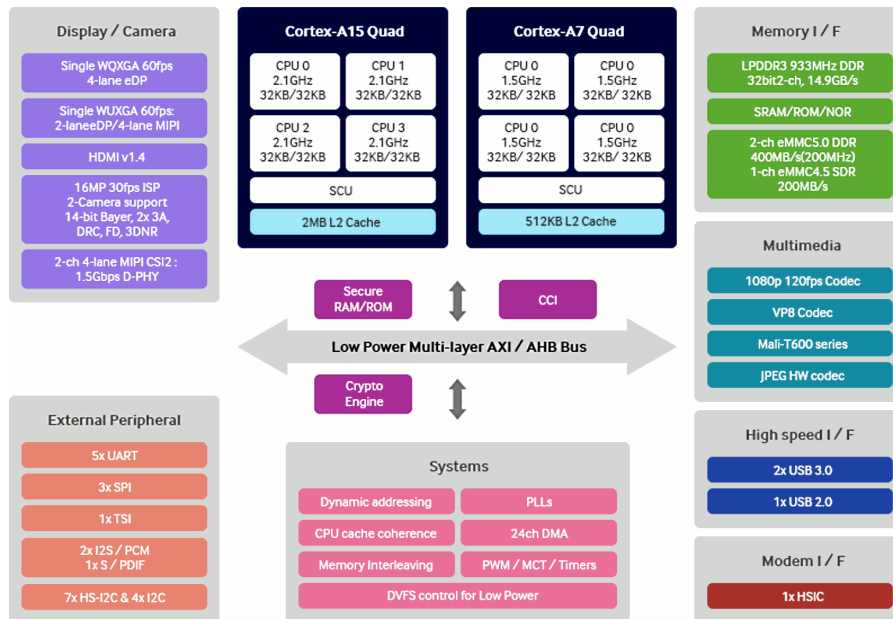




SoC Beispiele (cont.)

▶ Samsung Exynos-5422 (2014)

[Samsung]

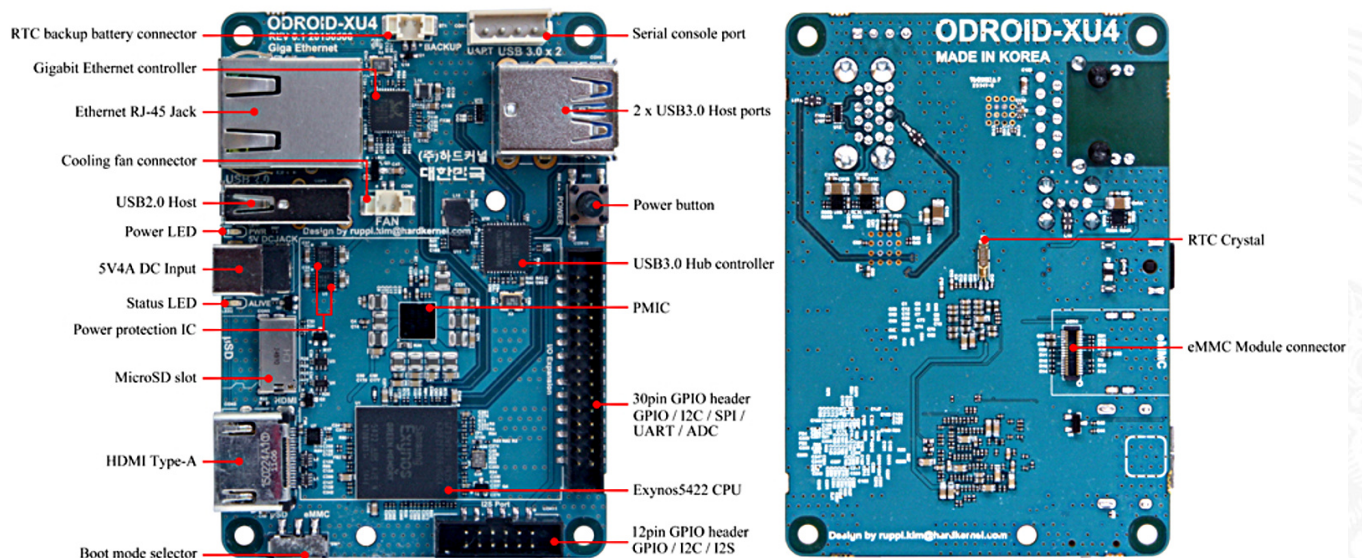


SoC Beispiele (cont.)

▶ Beispiel: Odroid XU4

[Hardkernel]

- ▶ vollständiger 8-Kern Mikrocomputer
- ▶ Betriebssystem: Android oder Linux





- ▶ 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 für 1-bit gespeichert wird
 - ▶ Ab diesem Zeitpunkt gibt es bessere Performanz nur noch durch bessere Algorithmen / Architekturen!
- ⇒ Annahme: 50 % Skalierung pro Jahr, 200 Elektronen/Speicherzelle
gesucht: $x \hat{=}$ Jahre Fortschritt
- ⇒ $200 / (1,5^x) \geq 1$
 $x = \ln(200) / \ln(1,5) \approx 13$ Jahre

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



IEEE International Roadmap for Devices and Systems
irds.ieee.org/editions/2022

- ▶ 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
- ▶ neue Technologien: Quantencomputing etc.



Roadmap: IRDS (cont.)

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

YEAR OF PRODUCTION	2017	2019	2021	2024	2027	2030	2033
Logic Industry "Node Range" Labeling (nm)	P54M36	P48M28	P42M24	P36M21	P32M14	P32M14T2	P32M14T4
IDM-Foundry node labeling	"10"	"7"	"5"	"3"	"2.1"	"1.5"	"1.0"
Logic device structure options	finFET FDSON	finFET LGAA	finFET LGAA	LGAA VGAA	LGAA VGAA	VGAA, LGAA 3D VLSI	VGAA, LGAA 3D VLSI
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 technology 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 (1T1) 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

1.6 Einführung - Roadmap und Grenzen des Wachstums

64-040 Rechnerstrukturen und Betriebssysteme

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

1.6 Einführung - Roadmap und Grenzen des Wachstums

64-040 Rechnerstrukturen und Betriebssysteme

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.
- ⇒ 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?

1.6 Einführung - Roadmap und Grenzen des Wachstums

64-040 Rechnerstrukturen und Betriebssysteme

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
- ▶ Prozesskontrolle, Locking, Interrupts, Scheduling
- ▶ virtueller Speicher, Dateisystem, Ein- / Ausgabe
- ▶ ...

A. Mäder

79



Literatur

1.7 Einführung - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

[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

A. Mäder

80



[IRDS22] *International Roadmap for Devices and Systems (IRDS) 2022 Edition.*

IEEE International Roadmap for Devices and Systems, 2022.

irds.ieee.org/editions/2022

[Intel] Intel Corp.; Santa Clara, CA.

www.intel.com ark.intel.com

www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html

www.intel.de/content/www/de/de/silicon-innovations

[TI] Texas Instruments Inc.; Dallas, TX.

www.ti.com

[Samsung] Samsung Electronics Co., Ltd.; Suwon, Südkorea.

www.samsung.com

[Hardkernel] Hardkernel co., Ltd.; AnYang, Südkorea.

www.hardkernel.com



1. Einführung

2. Informationsverarbeitung

Semantic Gap

Abstraktionsebenen

Beispiel: HelloWorld

Definitionen und Begriffe

Informationsübertragung

Zeichen

Literatur

3. Ziffern und Zahlen

4. Arithmetik

5. Zeichen und Text

6. Logische Operationen



Gliederung (cont.)

2 Informationsverarbeitung

64-040 Rechnerstrukturen und Betriebssysteme

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



A. Mäder

83



Definition: Digitalrechner

2 Informationsverarbeitung

64-040 Rechnerstrukturen und Betriebssysteme

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**

A. Mäder

84



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
- ⇒ extrem niedriges Abstraktionsniveau
- ▶ natürliche Sprache immer mit Kontextwissen
Beispiel: „Vereinbaren Sie einen Besprechungstermin“ ⇒ Wo? Wer? Thema?
 - ▶ **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, Entwicklung ...
 - ▶ Zuverlässigkeit, Wartungsfreundlichkeit, Sicherheit ...

Achtung / Vorsicht:

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



Rückwirkung von unteren Ebenen: Arithmetik

2.1 Informationsverarbeitung - Semantic Gap

64-040 Rechnerstrukturen und Betriebssysteme

```

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
    }
}

```

A. Mäder

87



Rückwirkung von unteren Ebenen: Performanz

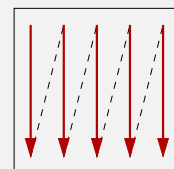
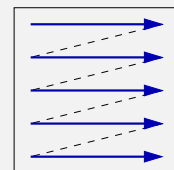
2.1 Informationsverarbeitung - Semantic Gap

64-040 Rechnerstrukturen und Betriebssysteme

```

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;
}

```



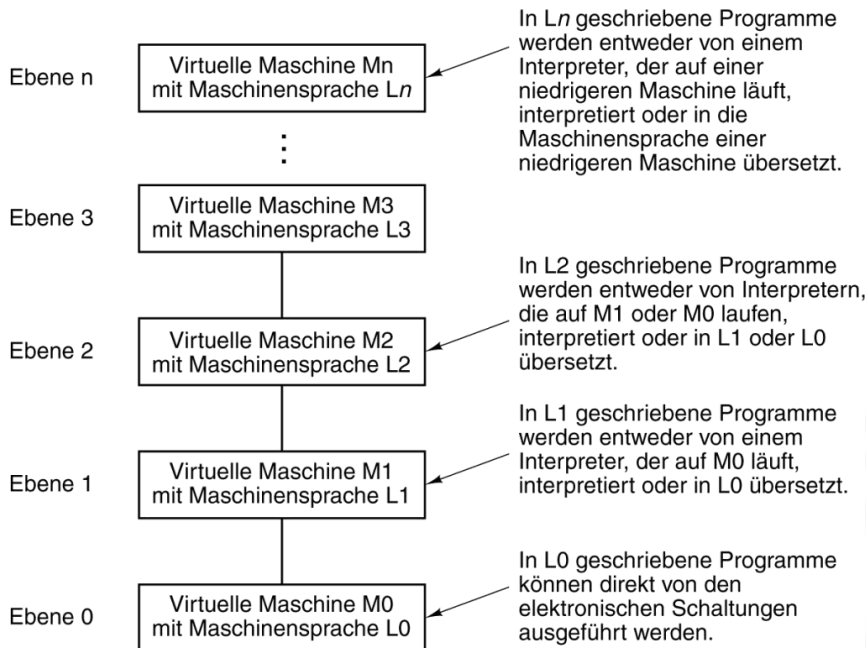
Matrix creation (5000×5000)	2105 ms	
Matrix row-col summation	75 ms	
Matrix col-row summation	383 ms	⇒ 5 × langsamer
Sum = 600,8473695346258 / 600,8473695342268		⇒ andere Werte

A. Mäder

88



Maschine mit mehreren Ebenen



Tanenbaum: *Structured Computer Organization* [TA14]



Abstraktionsebenen und Sprachen

- ▶ jede Ebene definiert eine neue (mächtigere) Sprache
- ▶ Abstraktionsebene \iff Sprache
- ▶ $L_0 < L_1 < L_2 < L_3 < \dots$

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



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

⇒ 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



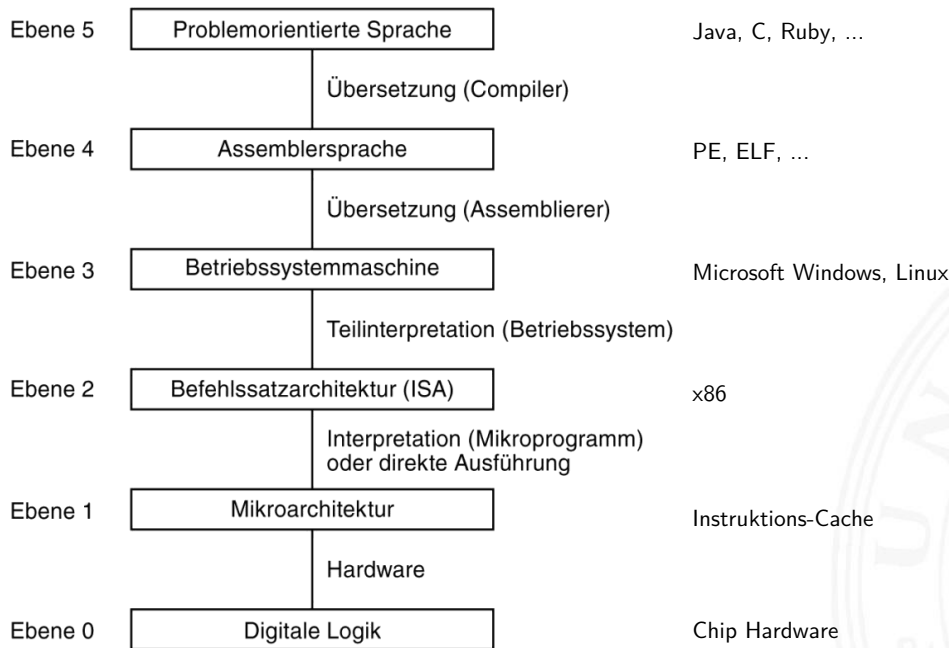
- | | |
|---------------------|--|
| Anwendungsebene | Hochsprachen (Java, C, Ruby, ...) |
| Assemblerebene | low-level Anwendungsprogrammierung |
| Betriebssystemebene | Betriebssystem, Systemprogrammierung |
| Rechnerarchitektur | Schnittstelle zwischen SW und HW: Befehlssatz, Datentypen |
| Mikroarchitektur | Steuerwerk, Operationswerk: Register, ALU, Speicher, Busse ... |
| Logikebene | Grundsaltungen: Gatter, Flipflops ... |
| Transistorebene | Elektrotechnik: Transistoren, Widerstände, Kapazitäten ... |
| Physikalische Ebene | Chip-Layout: Geometrien für die IC-Fertigung |



Beispiel: Sechs Ebenen

2.2 Informationsverarbeitung - Abstraktionsebenen

64-040 Rechnerstrukturen und Betriebssysteme



A. Mäder

93



Hinweis: Ebenen vs. Vorlesungen im BSc-Studiengang

2.2 Informationsverarbeitung - Abstraktionsebenen

64-040 Rechnerstrukturen und Betriebssysteme

Anwendungsebene: SE1+SE2, AD ...

Assemblerebene: RSB

Betriebssystemebene: RSB, MB

Rechnerarchitektur: RSB

Mikroarchitektur: RSB

Logikebene: RSB

Device-Level: -

A. Mäder

94



HelloWorld: Anwendungsebene Quellcode

2.3 Informationsverarbeitung - Beispiel: HelloWorld

64-040 Rechnerstrukturen und Betriebssysteme

```
/* 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
```

A. Mäder

95



HelloWorld: Assemblerebene cat HelloWorld.s

2.3 Informationsverarbeitung - Beispiel: HelloWorld

64-040 Rechnerstrukturen und Betriebssysteme

```
.file "HelloWorld.c"  
.text  
.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)  
leaq .LC0(%rip), %rax  
movq %rax, %rdi  
call puts@PLT  
movl $0, %eax  
leave  
.cfi_def_cfa 7, 8  
ret  
.cfi_endproc  
.LFE0:  
.size main, .-main  
.ident "GCC: (Debian 12.2.0-14) 12.2.0"  
.section .note.GNU-stack,"",@progbits
```

A. Mäder

96



```

00000000 457f 464c 0102 0001 0000 0000 0000 0000
00000020 0001 003e 0001 0000 0000 0000 0000 0000
00000040 0000 0000 0000 0000 0228 0000 0000 0000
00000060 0000 0000 0040 0000 0000 0040 000d 000c
00001000 4855 e589 8348 10ec 7d89 48fc 7589 48f0
00001200 058d 0000 0000 8948 e8c7 0000 0000 00b8
00001400 0000 c900 48c3 6c65 6f6c 202c 6f77 6c72
00001600 2164 0000 4347 3a43 2820 6544 6962 6e61
00002000 3120 2e32 2e32 2d30 3431 2029 3231 322e
00002200 302e 0000 0000 0000 0014 0000 0000 0000
00002400 7a01 0052 7801 0110 0c1b 0807 0190 0000
00002600 001c 0000 001c 0000 0000 0000 0025 0000
00003000 4100 100e 0286 0d43 6006 070c 0008 0000
00003200 0000 0000 0000 0000 0000 0000 0000 0000
00003400 0000 0000 0000 0000 0001 0000 0004 fff1
00003600 0000 0000 0000 0000 0000 0000 0000 0000
00004000 0000 0000 0003 0001 0000 0000 0000 0000
...

```



```

HelloWorld.o:          file format 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:   48 8d 05 00 00 00 lea   0x0(%rip),%rax    # 16 <main+0x16>
16:   48 89 c7          mov    %rax,%rdi
19:   e8 00 00 00 00   call  1e <main+0x1e>
1e:   b8 00 00 00 00   mov    $0x0,%eax
23:   c9              leave
24:   c3              ret

```



HelloWorld: Maschinencode `od -x HelloWorld.exe`

```

00000000 457f 464c 0102 0001 0000 0000 0000 0000
00000020 0003 003e 0001 0000 1050 0000 0000 0000
00000040 0040 0000 0000 0000 3698 0000 0000 0000
00000060 0000 0000 0040 0038 000d 0040 001f 001e
00001000 0006 0000 0004 0000 0040 0000 0000 0000
00001200 0040 0000 0000 0000 0040 0000 0000 0000
00001400 02d8 0000 0000 0000 02d8 0000 0000 0000
00001600 0008 0000 0000 0000 0003 0000 0004 0000
00002000 0318 0000 0000 0000 0318 0000 0000 0000
00002200 0318 0000 0000 0000 001c 0000 0000 0000
00002400 001c 0000 0000 0000 0001 0000 0000 0000
00002600 0001 0000 0004 0000 0000 0000 0000 0000
00003000 0000 0000 0000 0000 0000 0000 0000 0000
00003200 0618 0000 0000 0000 0618 0000 0000 0000
00003400 1000 0000 0000 0000 0001 0000 0005 0000
00003600 1000 0000 0000 0000 1000 0000 0000 0000
00004000 1000 0000 0000 0000 0169 0000 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“
... jedenfalls in Bezug auf L1 Programmausführung

Karen Panetta Lentz

Entscheidung für Software- oder Hardwarerealisierung?

- ▶ abhängig von vielen Faktoren, u.a.
- ▶ Kosten, Performanz, Zuverlässigkeit
- ▶ Anzahl der (vermuteten) Änderungen und Updates
- ▶ Sicherheit gegen Kopieren ...
- ▶ Beispiele: Virtualisierung im RZ ⇒ Software
(High-Level Synthese+IP+) FPGAs ⇒ Hardware



- ▶ **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**
- ▶ **Interpretation** ist das Ermitteln der Information aus einer Repräsentation
- ▶ **Verstehen** ist das Verbinden einer Information mit ihrer Bedeutung in der realen Welt



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: ● ● — — — ● ● ● ● ●

$$2 \cdot 10^1 + 5 \cdot 10^0 = 25$$

$$1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^0 = 25$$

$$2 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 = 25$$

siehe: 3.2 Stellenwertsystem



Interpretation: Information vs. Repräsentation

- ▶ Wo auch immer Repräsentationen auftreten, ist eigentlich die Information gemeint, beispielsweise:

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

- ▶ Die Information selbst kann man überhaupt nicht notieren!
- ▶ Es muss immer Absprachen über die verwendete Repräsentation geben.

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:	BCD, 4-bit binär	0011 0100 0111
Bit:	elektrische Spannung	0,1V 0,1V 2,5V 2,5V ...



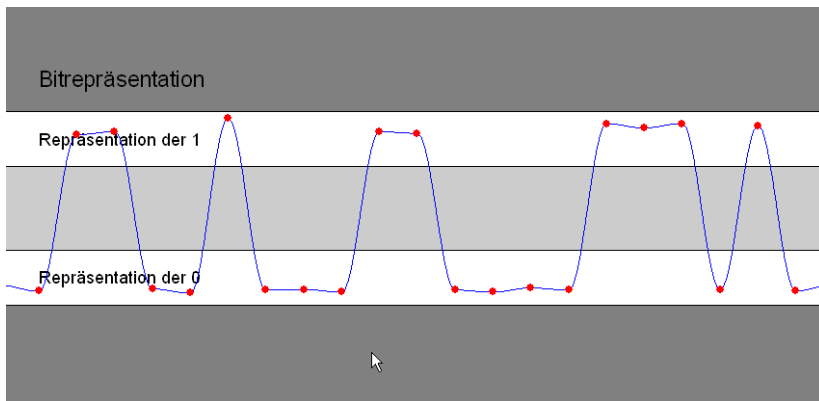
Repräsentation: Ebenen

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 $\pi = 3,1415\dots$ 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...“
- ▶ ...



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:
 - ▶ $0,0V \leq V(t) \leq 0,7V$: Interpretation als 0
 - ▶ $1,7V \leq V(t) \leq 2,5V$: Interpretation als 1
 - ▶ außerhalb und innerhalb: ungültige Werte



- ▶ Aussagen
 - N1 Er besucht General Motors
 - N2 Unwetter am Alpenstrand
 - 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?
- ⇒ **Interpretation:** Es handelt sich um drei **Nachrichten**, die jeweils zwei verschiedene **Informationen** enthalten

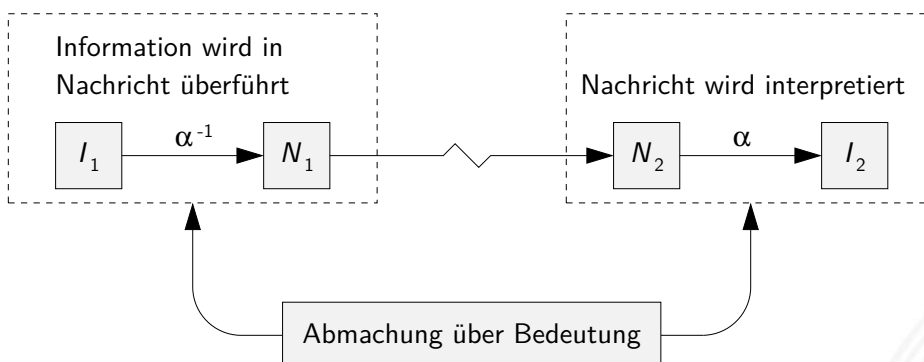


Information vs. Nachricht (cont.)

- ▶ **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: 21° Temperaturangabe in Grad Celsius oder Fahrenheit
- ▶ Die Nachricht ist also eine Darstellung von Informationen und nicht der Übermittlungsvorgang



Modell der Informationsübertragung

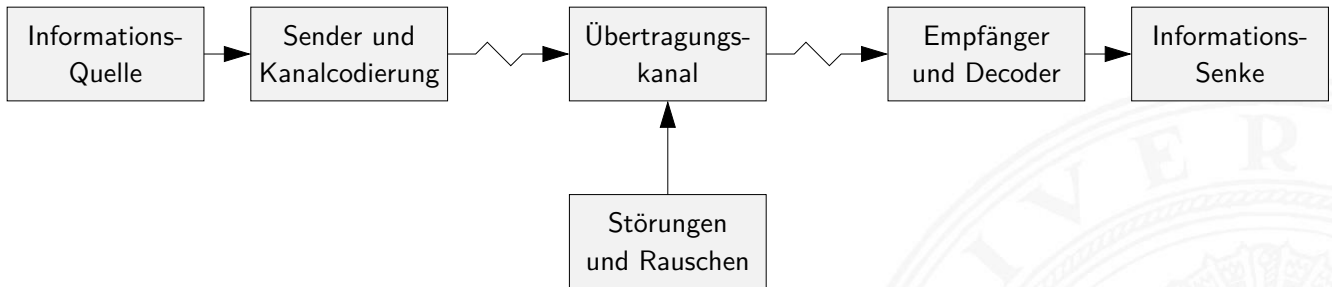


Beschreibung der **Informationsübermittlung:**

- ▶ 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



Nachrichtentechnisches Modell: **Störungen** bei der Übertragung



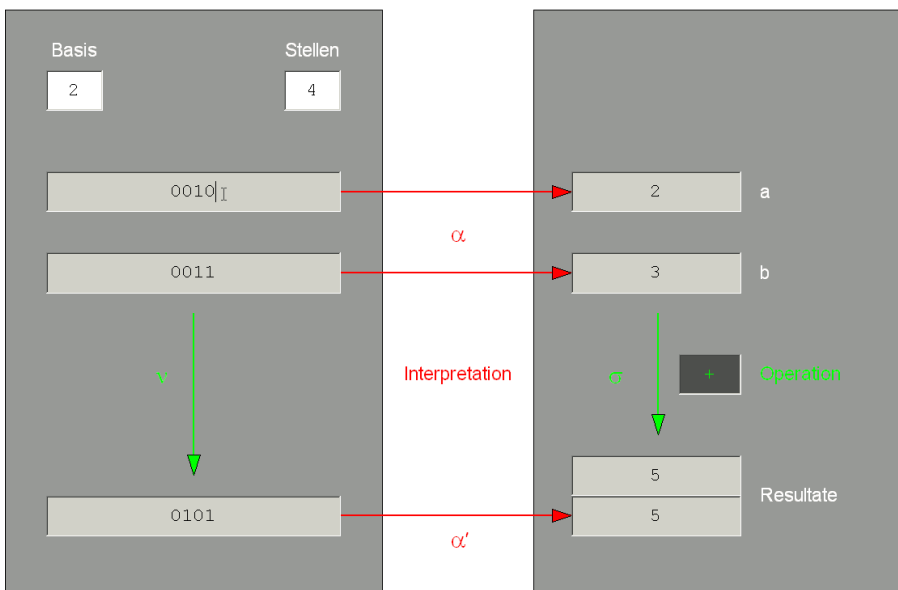
Beispiele

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



Repräsentation

Information



Repräsentation natürlicher Zahlen durch Stellenwertsysteme

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



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}, σ) und (\mathcal{R}, ν)
- ▶ ist σ bijektiv, liegt ein **Isomorphismus** vor



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



- ▶ **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
 - ▶ $Z_1 = \{0, 1\}$
 - ▶ $Z_2 = \{0, 1, 2, \dots, 9, A, B, C, D, E, F\}$
 - ▶ $Z_3 = \{\alpha, \beta, \gamma, \dots, \omega\}$
 - ▶ $Z_4 = \{CR, LF\}$
- ▶ **Numerischer Zeichensatz:** Zeichenvorrat aus Ziffern und/oder Sonderzeichen zur Darstellung von Zahlen
- ▶ **Alphanumerischer Zeichensatz:** Zeichensatz aus (mindestens) den Dezimalziffern und den Buchstaben des Alphabets, meistens auch mit Sonderzeichen (Leerzeichen, Punkt, Komma usw.)



- ▶ **Binärzeichen:** engl. *binary element, binary digit, bit*
Jedes der Zeichen aus einem Vorrat / aus einer Menge von zwei Symbolen
Beispiele
 - ▶ $Z_1 = \{0, 1\}$
 - ▶ $Z_2 = \{\text{high, low}\}$
 - ▶ $Z_3 = \{\text{rot, grün}\}$
 - ▶ $Z_4 = \{+, -\}$
- ▶ **Alphabet:** engl. *alphabet*
Ein in vereinbarter Reihenfolge geordneter Zeichenvorrat $\mathcal{A} = Z$
Beispiele
 - ▶ $\mathcal{A}_1 = \{0, 1, 2, \dots, 9\}$
 - ▶ $\mathcal{A}_2 = \{\text{Mo, Di, Mi, Do, Fr, Sa, So}\}$
 - ▶ $\mathcal{A}_3 = \{A, B, C, \dots, Z\}$



- ▶ **Zeichenkette:** Eine Folge von Zeichen engl. *string*
- ▶ **Wort:** engl. *word*
Zeichenkette, die in einem gegebenen Zusammenhang als Einheit bezeichnet wird
- ▶ Worte aus 8 Binärzeichen (8 bit) werden als **Byte** bezeichnet
- ▶ **Stelle:** Die Position eines Zeichens in einer Zeichenkette engl. *position*

Beispiele

- ▶ $s_1 = \text{Hello, world!}$
- ▶ $s_2 = \text{das sind vier Worte}$
- ▶ $s_3 = \text{die Zeichenkette hat 32 Stellen!}$



- 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)



[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, Vorlesungsskript.

tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1



1. Einführung

2. Informationsverarbeitung

3. Ziffern und Zahlen

Konzept der Zahl

Stellenwertsystem

Umrechnung zwischen verschiedenen Basen

Zahlenbereich und Präfixe

Festkommazahlen

Darstellung negativer Zahlen

Gleitkomma und IEEE 754

Maschinenworte

Literatur

4. Arithmetik



Gliederung (cont.)

3 Ziffern und Zahlen

64-040 Rechnerstrukturen und Betriebssysteme

5. Zeichen und Text
6. Logische Operationen
7. Codierung
8. Schaltfunktionen
9. Schaltnetze
10. Schaltwerke
11. Rechnerarchitektur I
12. Instruction Set Architecture
13. Assembler-Programmierung
14. Rechnerarchitektur II
15. Betriebssysteme



A. Mäder

119



Konzept der Zahl

3.1 Ziffern und Zahlen - Konzept der Zahl

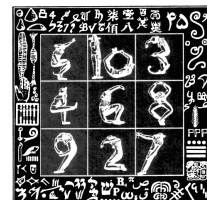
64-040 Rechnerstrukturen und Betriebssysteme

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

Abstraktion zum:

- ▶ Zählen
- ▶ Speichern
- ▶ Rechnen

Georges Ifrah
Universal-
geschichte der
Zahlen



A. Mäder

120



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

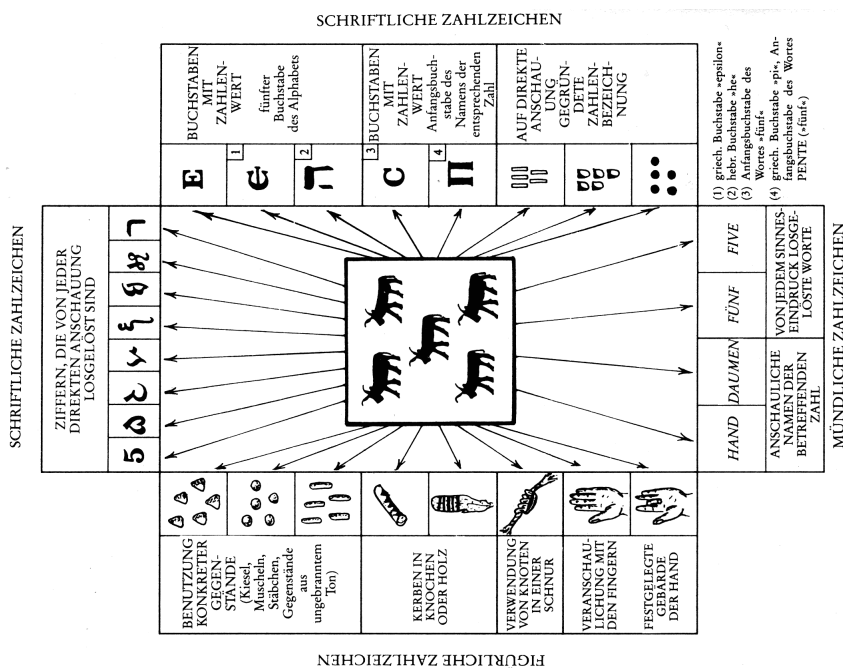


Abb. 11: Verschiedene, einer ganzen Zahl (hier der Zahl 5) zugeordnete Symbole.

[lfr10]



Zählen mit den Fingern („digits“)

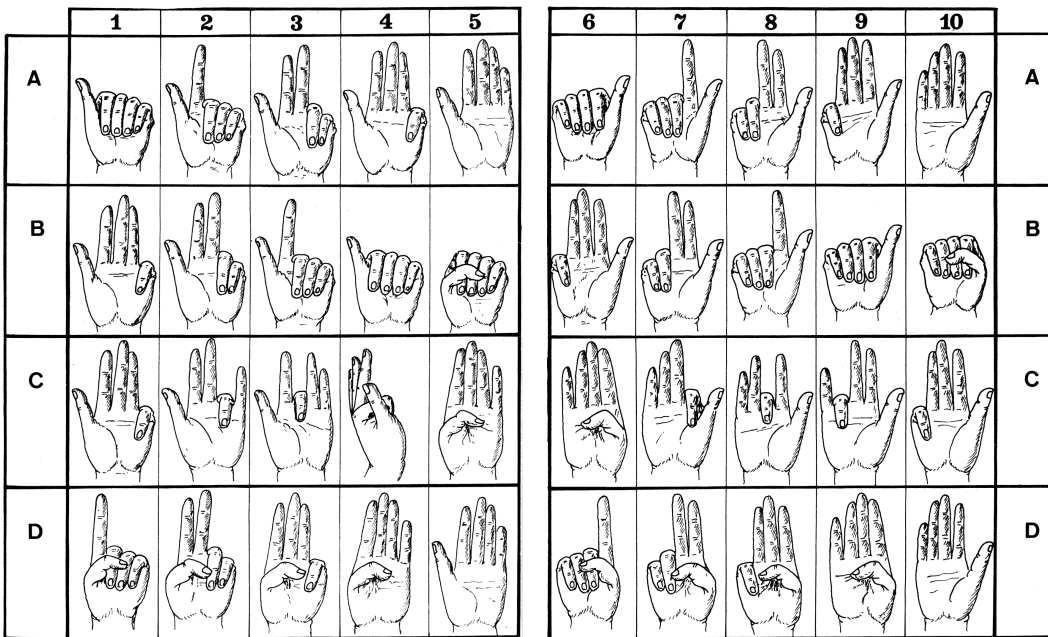


Abb. 12: Verschiedene Möglichkeiten des Zählens mit den Fingern.

[lfr10]



Speicherung

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.

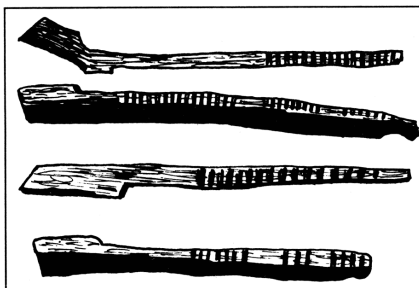
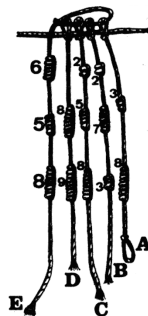


Abb. 59: Englische Kerbhölzer aus dem 13. Jahrhundert. (Sammlung Society of Antiquaries, London; Zeichnung nach Menninger 1957/58, II, 42)

Knotenschnüre

[lfr10]



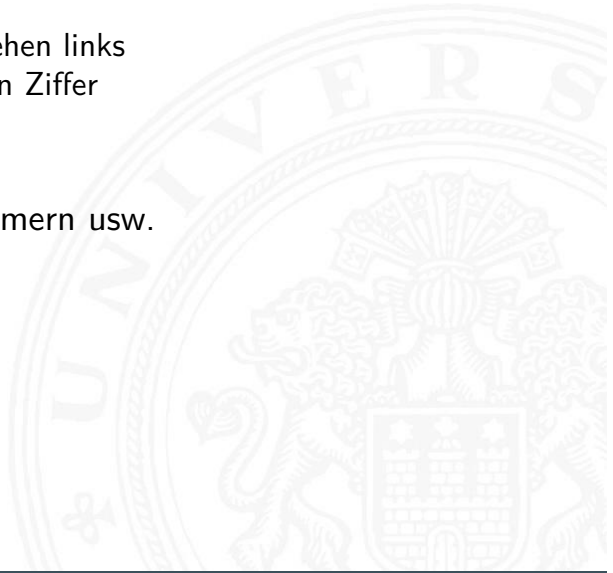
658	89	258	273	38
E	D	C	B	A

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. Le-land Locke 1923)



Rechnen: Römische Ziffern

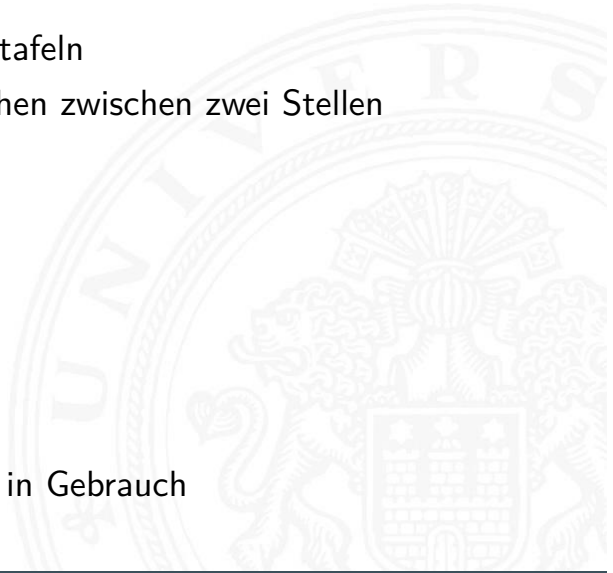
- ▶ Ziffern: I=1, V=5, X=10, L=50, C=100, D=500, M=1000
- ▶ Werte eins bis zehn: I, II, III, IV, V, VI, VII, VIII, IX, X
- ▶ Position der Ziffern ist signifikant:
 - ▶ nach Größe der Ziffernsymbole sortiert, größere stehen links
 - ▶ andernfalls Abziehen der kleineren von der größeren Ziffer
 - ▶ IV=4, VI=6, XL=40, LXX=70, CM=900
- ▶ heute noch in Gebrauch: Jahreszahlen, Seitennummern usw.
Beispiele: MDCCCXIII=1813, MMXXIII=2023
- keine Symbole zur Darstellung großer Zahlen
- Rechenoperationen so gut wie unmöglich

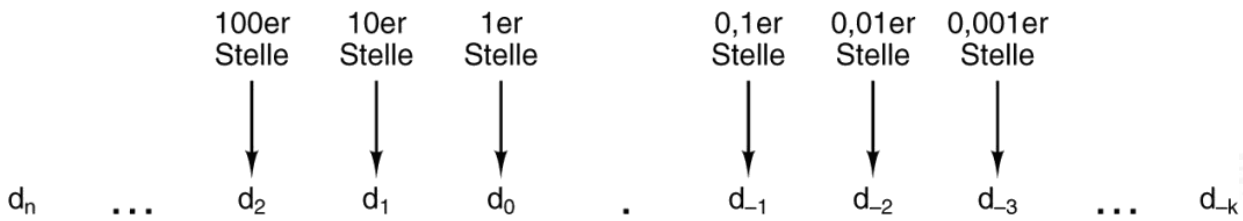


Babylon: Sexagesimalsystem

- ▶ vor ungefähr 4 000 Jahren, erstes **Stellenwertsystem**
- ▶ Basis 60
- ▶ zwei Symbole: | = 1 und < = 10
Einritzen gerader und gewinkelter Striche auf Tontafeln
- ▶ Null bekannt: wird nicht mitgeschrieben, Leerzeichen zwischen zwei Stellen
- ▶ Beispiele

▶	5
▶ <<	23
▶ <<<	$90 = 1 \cdot 60 + 3 \cdot 10$
▶ <<	$3621 = 1 \cdot 3600 + 0 \cdot 60 + 2 \cdot 10 + 1$
- ▶ für Zeitangaben und Winkелеinteilung heute noch in Gebrauch





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

[TA14]

- ▶ das im Alltag gebräuchliche Zahlensystem
- ▶ Vorkommastellen: Einer, Zehner, Hunderter, Tausender ...
- Nachkommastellen: Zehntel, Hundertstel, Tausendstel ...



- ▶ Wahl einer geeigneten Zahlenbasis b („Radix“)
 - ▶ 10: Dezimalsystem
 - ▶ 16: Hexadezimalsystem (Sedezimalsystem)
 - ▶ 2: Dualsystem
- ▶ Menge der entsprechenden Ziffern $\{0, 1, \dots, b - 1\}$
- ▶ 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 114

- + 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

$$\begin{array}{l}
 0_2 = 0_{10} \\
 1_2 = 1_{10} \\
 11_2 = 3_{10} \quad 2^1 + 2^0 \\
 110100_2 = 52_{10} \quad 2^5 + 2^4 + 2^2 \\
 11111110_2 = 254_{10} \quad 2^8 + 2^7 + \dots + 2^2 + 2^1
 \end{array}$$



Addition im Dualsystem

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

+	0	1
0	0	1
1	1	10

- ▶ Beispiel

$$\begin{array}{r}
 10110011 \\
 + 00111001 \\
 \hline
 \ddot{U} \ 11 \ 11 \\
 \hline
 11101100
 \end{array}
 \quad
 \begin{array}{r}
 = 179 \\
 = 57 \\
 \hline
 11 \\
 \hline
 = 236
 \end{array}$$

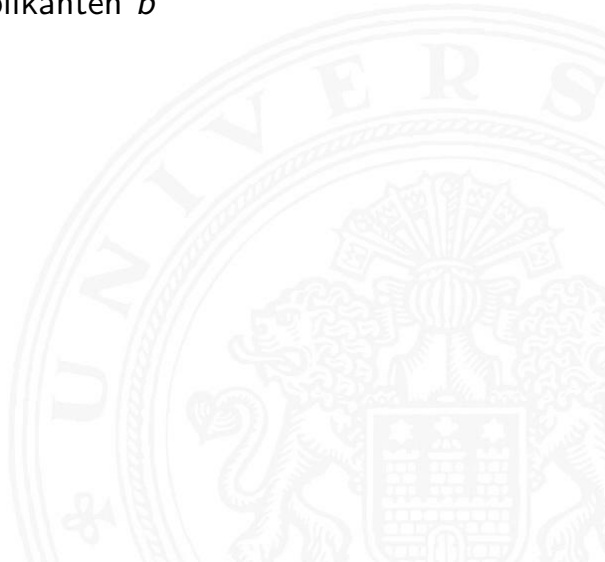




Multiplikation im Dualsystem

- ▶ 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 – sehr einfach: $\cdot 0 / \cdot 1$

$$\begin{array}{c|cc}
 \cdot & 0 & 1 \\
 \hline
 0 & 0 & 0 \\
 1 & 0 & 1
 \end{array}$$



Multiplikation im Dualsystem (cont.)

- ▶ Beispiel

$$\begin{array}{r}
 10110011 \cdot 1101 = 179 \cdot 13 = 2327 \\
 \hline
 10110011 \quad 1 \\
 10110011 \quad 1 \\
 00000000 \quad 0 \\
 10110011 \quad 1 \\
 \hline
 \ddot{U} \quad 11101111 \\
 \hline
 100100010111
 \end{array}$$

$$\begin{array}{l}
 = 1001\ 0001\ 0111 \\
 = 0x917
 \end{array}$$





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.
- ⇒ Fehler: Dezimalzahl in C mit 0 beginnen!
- ▶ 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 $\{0, 1, 2, \dots, 9, A, B, C, D, E, F\}$
- ▶ C-Schreibweise mit Präfix 0x – Klein- oder Großbuchstaben
 - ▶ $0x00000001 = 1_{10}$
 - $0x000000fe = 254_{10} = 15 \cdot 16 + 14$
 - $0x0000ffff = 65\,535_{10} = 15 \cdot 4\,096 + 15 \cdot 256 + 15 \cdot 16 + 15$
 - $0xcafebabe = \dots$ 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

$0x0 = 0000$	$0x1 = 0001$	$0x2 = 0010$	$0x3 = 0011$
$0x4 = 0100$	$0x5 = 0101$	$0x6 = 0110$	$0x7 = 0111$
$0x8 = 1000$	$0x9 = 1001$	$0xA = 1010$	$0xB = 1011$
$0xC = 1100$	$0xD = 1101$	$0xE = 1110$	$0xF = 1111$



Beispiel: Darstellungen der Zahl 2023

Binär

$$\begin{aligned}
& 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \\
& 1 \cdot 2^{10} + 1 \cdot 2^9 + 1 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
& 1024 + 512 + 256 + 128 + 64 + 32 + 0 + 0 + 4 + 2 + 1
\end{aligned}$$

Oktal

$$\begin{aligned}
& 3 \quad 7 \quad 4 \quad 7 \\
& 3 \cdot 8^3 + 7 \cdot 8^2 + 4 \cdot 8^1 + 7 \cdot 8^0 \\
& 1536 + 448 + 32 + 7
\end{aligned}$$

Dezimal

$$\begin{aligned}
& 2 \quad 0 \quad 2 \quad 3 \\
& 2 \cdot 10^3 + 0 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 \\
& 2000 + 0 + 20 + 3
\end{aligned}$$

Hexadezimal

$$\begin{aligned}
& 7 \quad E \quad 7 \\
& 7 \cdot 16^2 + E \cdot 16^1 + 7 \cdot 16^0 \\
& 1792 + 224 + 7
\end{aligned}$$



Umrechnung Dual-/Oktal-/Hexadezimalsystem

► Beispiele

Hexadezimal

$$\begin{array}{ccccccc}
1 & 9 & 4 & 8 & . & B & 6 \\
\hline
0001 & 1001 & 0100 & 1000 & . & 1011 & 1011 & 00 \\
\hline
1 & 4 & 5 & 1 & 0 & . & 5 & 5 & 4
\end{array}$$

Binär

Oktal

Hexadezimal

$$\begin{array}{ccccccc}
7 & B & A & 3 & . & B & C & 4 \\
\hline
0111 & 1011 & 1101 & 0011 & . & 1011 & 1110 & 0010 & 00 \\
\hline
7 & 5 & 6 & 4 & 3 & . & 5 & 7 & 0 & 4
\end{array}$$

Binär

Oktal

- Gruppieren von jeweils 3 bzw. 4 Bits
- bei Festkomma vom Dezimalpunkt aus nach links (2^n) für Vorkommastellen
rechts (2^{-m}) für Nachkommastellen



Umrechnung zwischen verschiedenen Basen

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

- ▶ Algorithmen zur Umrechnung notwendig
- ▶ Exemplarisch Vorstellung von drei Varianten:
 1. vorberechnete Potenztabellen
 2. Divisionsrestverfahren
 3. Horner-Schema

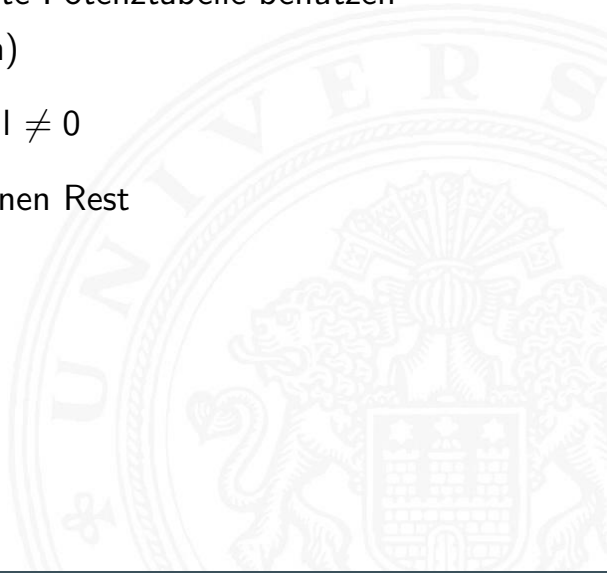
Basis: 60



Umwandlung über Potenztabellen

Vorgehensweise für Integerzahlen

- 1.a Subtraktion des größten Vielfachen einer Potenz des Zielsystems von der umzuwandelnden Zahl, dabei vorberechnete Potenztabelle benutzen
- 1.b Notation dieses größten Vielfachen (im Zielsystem)
 - ▶ Schritte wiederholen solange der Rest der Zahl $\neq 0$
- 2.a Subtraktion des größten Vielfachen vom verbliebenen Rest
- 2.b Addition dieses Vielfachen (im Zielsystem)
- ... usw.





Potenztabellen Dual/Dezimal

Stelle ₂	Wert ₁₀	Stelle ₁₀	Wert ₂
2 ⁰	1	10 ⁰	1
2 ¹	2	10 ¹	1010
2 ²	4	10 ²	110 0100
2 ³	8	10 ³	11 1110 1000
2 ⁴	16	10 ⁴	10 0111 0001 0000
2 ⁵	32	10 ⁵	0x1 86 A0
2 ⁶	64	10 ⁶	0xF 42 40
2 ⁷	128	10 ⁷	0x98 96 80
2 ⁸	256	10 ⁸	0x5 F5 E1 00
2 ⁹	512	10 ⁹	0x3B 9A CA 00
2 ¹⁰	1 024	10 ¹⁰	0x2 54 0B E4 00
2 ¹¹	2 048	10 ¹¹	0x17 48 76 E8 00
2 ¹²	4 096	10 ¹²	0xE8 D4 A5 10 00



Potenztabellen: Beispiel

- Umwandlung Dezimal- in Dualzahl

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

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	



Potenztabellen: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

$$Z = (10100011)_2 \leftrightarrow (163)_{10}$$

10100011			
– 1100100	$1 \cdot 10^2$	100	
00111111			
– 111100	$6 \cdot 10^1$	+ 60	
11			
– 11	$3 \cdot 10^0$	+ 3	
0		<u>163</u>	



Potenztabellen: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

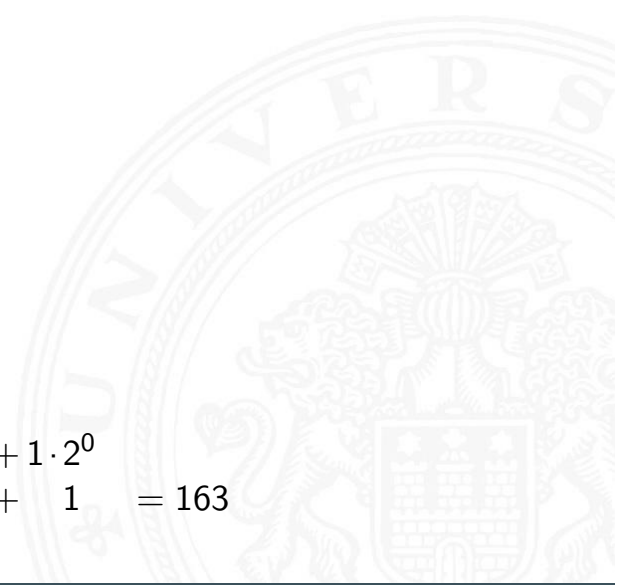
$$Z = (10100011)_2 \leftrightarrow (163)_{10}$$

10100011			
– 1100100	$1 \cdot 10^2$	100	
00111111			
– 111100	$6 \cdot 10^1$	+ 60	
11			
– 11	$3 \cdot 10^0$	+ 3	
0		<u>163</u>	

einfacher: Aufsummieren der Potenzen

$$1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$128 + 0 + 32 + 0 + 0 + 0 + 2 + 1 = 163$$





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

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

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



Divisionsrestverfahren: Beispiel

- ▶ Umwandlung Dual- in Dezimalzahl

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

$$\begin{array}{rcl}
 (1010\ 0011)_2 : (1010)_2 = 1\ 0000 & \text{Rest } (11)_2 \hat{=} 3 & 10^0 \\
 (1\ 0000)_2 : (1010)_2 = & 1 & \text{Rest } (110)_2 \hat{=} 6 & 10^1 \\
 (1)_2 : (1010)_2 = & 0 & \text{Rest } (1)_2 \hat{=} 1 & 10^2
 \end{array}$$

Hinweis: Division in Basis b folgt



Divisionsrestverfahren: Beispiel (cont.)

► Umwandlung Dezimal- in Dualzahl

$$Z = (1789)_{10} \leftrightarrow (110\ 1111\ 1101)_2$$

1789 : 2 = 894	Rest 1	2 ⁰	
894 : 2 = 447	Rest 0	:	
447 : 2 = 223	Rest 1		
223 : 2 = 111	Rest 1		
111 : 2 = 55	Rest 1		
55 : 2 = 27	Rest 1		
27 : 2 = 13	Rest 1		
13 : 2 = 6	Rest 1		
6 : 2 = 3	Rest 0	↑	Leserichtung
3 : 2 = 1	Rest 1	:	
1 : 2 = 0	Rest 1	2 ¹⁰	



Divisionsrestverfahren: Algorithmus

Algorithmus

rechentechnisch

darzustellende Zahl x

123

Basis q

2

```

a := x
while a > 0
  yn := a mod q
  a := a div q
end

```



n = 1
 a = 123
 (a > 0) = 1
 a mod 2 = 1
 a div 2 = 61

000000000000000001
Resultat

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





Horner-Schema

- ▶ Darstellung einer Potenzsumme durch ineinander verschachtelte Faktoren

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

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

$$1 \cdot 1010 = 1010$$

$$+ \quad 110$$

$$\hline 1\ 0000 \cdot 1010 = 1010\ 0000$$

$$+ \quad \quad 11$$

$$\hline 1010\ 0011$$



Horner-Schema: Beispiel (cont.)

► Umwandlung Dual- in Dezimalzahl

1. Darstellung als Potenzsumme

$$Z = (1010\ 0011)_2 =$$

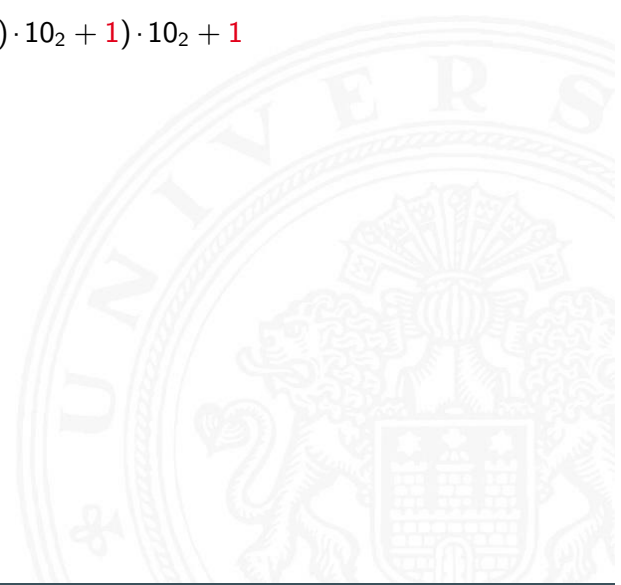
$$((((((1 \cdot 10_2 + 0) \cdot 10_2 + 1) \cdot 10_2 + 0) \cdot 10_2 + 0) \cdot 10_2 + 0) \cdot 10_2 + 1) \cdot 10_2 + 1$$

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

$$1 \cdot 2 = 2$$

$$\begin{array}{r} + 0 \\ \hline 2 \cdot 2 = 4 \end{array}$$

$$\begin{array}{r} + 1 \\ \hline 5 \cdot 2 = 10 \end{array}$$

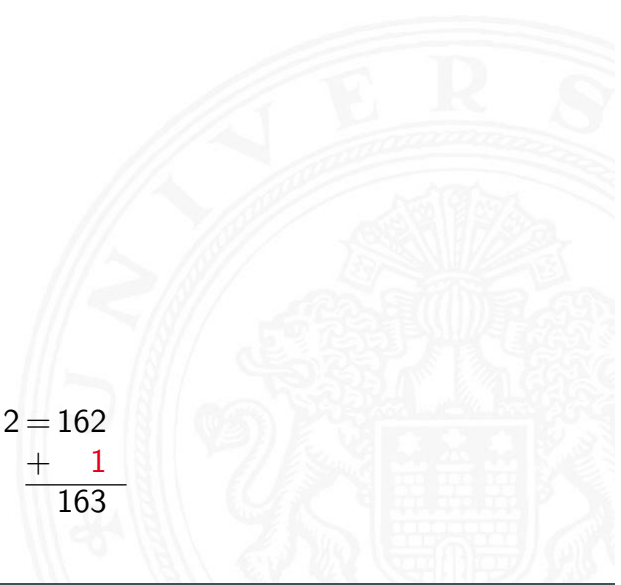
$$\begin{array}{r} + 0 \\ \hline 10 \cdot 2 = 20 \end{array}$$

$$\begin{array}{r} + 0 \\ \hline 20 \cdot 2 = 40 \end{array}$$

$$\begin{array}{r} + 0 \\ \hline 40 \cdot 2 = 80 \end{array}$$

$$\begin{array}{r} + 1 \\ \hline 81 \cdot 2 = 162 \end{array}$$

$$\begin{array}{r} + 1 \\ \hline 163 \end{array}$$





Horner-Schema: Beispiel (cont.)

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



Horner-Schema: Beispiel (cont.)

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

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

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

101110110111

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = 93$$

$$1 + 2 \cdot 93 = 187$$

$$0 + 2 \cdot 187 = 374$$

$$1 + 2 \cdot 374 = 749$$

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

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



Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

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

101110110111

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = 93$$

$$1 + 2 \cdot 93 = 187$$

$$0 + 2 \cdot 187 = 374$$

$$1 + 2 \cdot 374 = 749$$

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

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





Horner-Schema: Beispiel (cont.)

3.3 Ziffern und Zahlen - Umrechnung zwischen verschiedenen Basen

64-040 Rechnerstrukturen und Betriebssysteme

- Umwandlung Dual- in Dezimalzahl

$$Z = (1011\ 1011\ 0111)_2 \leftrightarrow (2999)_{10}$$

101110110111

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = 93$$

$$1 + 2 \cdot 93 = 187$$

$$0 + 2 \cdot 187 = 374$$

$$1 + 2 \cdot 374 = 749$$

$$1 + 2 \cdot 749 = 1499$$

$$1 + 2 \cdot 1499 = 2999$$



Horner-Schema: Beispiel (cont.)

3.3 Ziffern und Zahlen - Umrechnung zwischen verschiedenen Basen

64-040 Rechnerstrukturen und Betriebssysteme

- Umwandlung Dual- in Dezimalzahl

$$Z = (1011\ 1011\ 0111)_2 \leftrightarrow (2999)_{10}$$

101110110111

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = 93$$

$$1 + 2 \cdot 93 = 187$$

$$0 + 2 \cdot 187 = 374$$

$$1 + 2 \cdot 374 = 749$$

$$1 + 2 \cdot 749 = 1499$$

$$1 + 2 \cdot 1499 = 2999$$





Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

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

$$101110110111$$

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = 93$$

$$1 + 2 \cdot 93 = 187$$

$$0 + 2 \cdot 187 = 374$$

$$1 + 2 \cdot 374 = 749$$

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

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



Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

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

$$101110110111$$

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = 93$$

$$1 + 2 \cdot 93 = 187$$

$$0 + 2 \cdot 187 = 374$$

$$1 + 2 \cdot 374 = 749$$

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

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





Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

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

$$10111011\mathbf{1}0111$$

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = \mathbf{93}$$

$$\mathbf{1} + 2 \cdot \mathbf{93} = 187$$

$$0 + 2 \cdot 187 = 374$$

$$1 + 2 \cdot 374 = 749$$

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

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



Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

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

$$10111011\mathbf{0}111$$

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = 93$$

$$1 + 2 \cdot 93 = \mathbf{187}$$

$$\mathbf{0} + 2 \cdot \mathbf{187} = 374$$

$$1 + 2 \cdot 374 = 749$$

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

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





Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

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

$$101110110111$$

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = 93$$

$$1 + 2 \cdot 93 = 187$$

$$0 + 2 \cdot 187 = 374$$

$$1 + 2 \cdot 374 = 749$$

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

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



Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

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

$$101110110111$$

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = 93$$

$$1 + 2 \cdot 93 = 187$$

$$0 + 2 \cdot 187 = 374$$

$$1 + 2 \cdot 374 = 749$$

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

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





Horner-Schema: Beispiel (cont.)

- Umwandlung Dual- in Dezimalzahl

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

101110110111

$$1 + 2 \cdot 0 = 1$$

$$0 + 2 \cdot 1 = 2$$

$$1 + 2 \cdot 2 = 5$$

$$1 + 2 \cdot 5 = 11$$

$$1 + 2 \cdot 11 = 23$$

$$0 + 2 \cdot 23 = 46$$

$$1 + 2 \cdot 46 = 93$$

$$1 + 2 \cdot 93 = 187$$

$$0 + 2 \cdot 187 = 374$$

$$1 + 2 \cdot 374 = 749$$

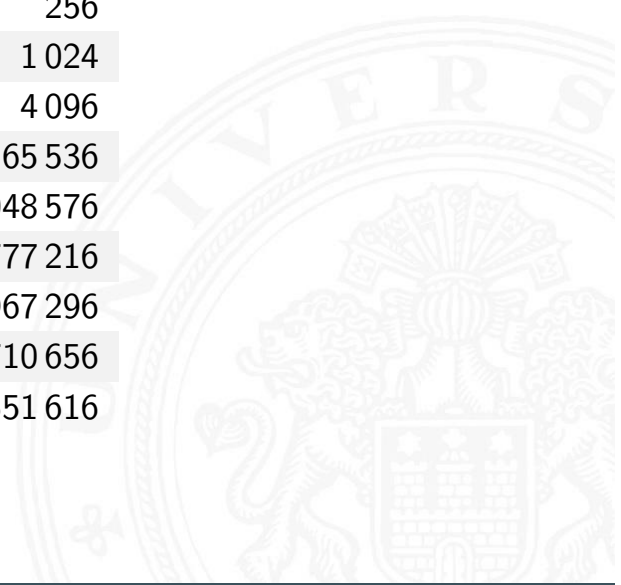
$$1 + 2 \cdot 749 = 1\ 499$$

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



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

- ▶ Präfixangabe als Abkürzung von Zehnerpotenzen für die vereinfachte Schreibweise von großen bzw. sehr kleinen Zahlen
- ▶ Beispiele
 - ▶ Lichtgeschwindigkeit: $300\,000\text{ Km/s} = 30\text{ cm/ns}$
 - ▶ Ruheenergie des Elektrons: $0,51\text{ MeV}$
 - ▶ Strukturbreite heutiger Mikrochips: 4 nm
 - ▶ usw.
- ▶ auch für das Dualsystem gibt es entsprechende Präfixe
- ▶ Vielfache von $2^{10} = 1024 \approx 1000$



Präfixe für Einheiten im Dezimalsystem

Faktor	Name	Symbol
10^{30}	quetta	Q
10^{27}	ronna	R
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^{-30}	quecto	q
10^{-27}	ronto	r
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}	mikro	μ
10^{-3}	milli	m
10^{-2}	centi	c
10^{-1}	dezi	d





Präfixe für Einheiten im Dualsystem

3.4 Ziffern und Zahlen - Zahlenbereich und Präfixe

64-040 Rechnerstrukturen und Betriebssysteme

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 mebibit = 1 048 576 bit
1 gibibit = 1 073 741 824 bit

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

A. Mäder

156



Präfixe für Einheiten im Dualsystem (cont.)

3.4 Ziffern und Zahlen - Zahlenbereich und Präfixe

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ in der Praxis nicht immer sauber verwendet
- ▶ meistens ergibt sich die Bedeutung aber aus dem Kontext
- ▶ bei Speicherbausteinen sind Zweierpotenzen üblich, es werden aber dezimale Präfixe verwendet
 - ▶ DRAM-Modul mit 4 GB Kapazität: gemeint sind 2^{32} Bytes
 - ▶ Flash-Speicherkarte 64 GB Kapazität: gemeint sind 2^{36} Bytes
- ▶ bei Festplatten wird Kapazität dezimal angegeben
 - ▶ Festplatte mit 4 TB Kapazität: typisch $4 \cdot 10^{12}$ Bytes
 - ▶ die tatsächliche angezeigte verfügbare Kapazität ist geringer, weil das jeweilige Dateisystem Platz für seine eigenen Verwaltungsinformationen belegt

A. Mäder

157



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

$$\begin{aligned} |z| &= \sum_{i=0}^{n-1} a_i \cdot b^i + \sum_{i=-m}^{i=-1} a_i \cdot b^i \\ &= \sum_{i=-m}^{n-1} a_i \cdot b^i \end{aligned}$$

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

- ▶ Der erste Summand ist der ganzzahlige Anteil, während der zweite Summand für den gebrochenen Anteil steht: n Vorkomma- und m Nachkommastellen



- ▶ $2^{-1} = 0,5$
- ▶ $2^{-2} = 0,25$
- ▶ $2^{-3} = 0,125$
- ▶ $2^{-4} = 0,0625$
- ▶ $2^{-5} = 0,03125$
- ▶ $2^{-6} = 0,015625$
- ▶ $2^{-7} = 0,0078125$
- ▶ $2^{-8} = 0,00390625$
- ▶ ...
- ▶ 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	B=2	B=10
0,1	0,00011	0,001	0,125
0,2	0,0011	0,010	0,25
0,3	0,01001	0,011	0,375
0,4	0,0110	0,100	0,5
0,5	0,1	0,101	0,625
0,6	0,1001	0,110	0,75
0,7	0,10110	0,111	0,875
0,8	0,1100		
0,9	0,11100		



Umrechnung: Dezimalbruch nach Dual

Potenztafel zur Umrechnung

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

- ▶ Beispiel: Dezimal 0,3

Berechnung durch Subtraktion der Werte

$$\begin{aligned}
 (0,3)_{10} &= 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + \dots \\
 &= 2^{-2} + 2^{-5} + 2^{-6} + 2^{-9} + \dots \\
 &= (0,01001)_2
 \end{aligned}$$



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
 $\text{---} \geq 1$: 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

$$\begin{array}{rcll}
2 \cdot 0,59375 & = & 1,1875 & \rightarrow 1 \quad 2^{-1} \\
2 \cdot 0,1875 & = & 0,375 & \rightarrow 0 \quad \vdots \\
2 \cdot 0,375 & = & 0,75 & \rightarrow 0 \quad \downarrow \text{ Leserichtung} \\
2 \cdot 0,75 & = & 1,5 & \rightarrow 1 \quad \vdots \\
2 \cdot 0,5 & = & 1,0 & \rightarrow 1 \quad 2^{-5} \\
(0,59375)_{10} & \leftrightarrow & (0,10011)_2 &
\end{array}$$



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

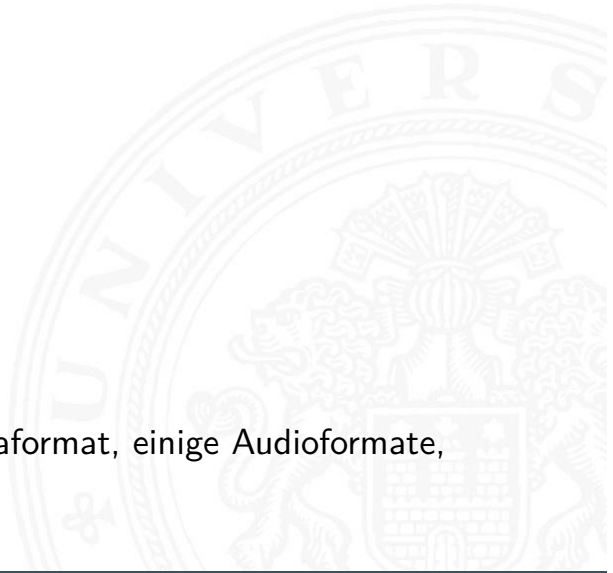


Exzess-Codierung (Offset-basiert)

- ▶ einfache Um-Interpretation der Binärcodierung

$$z = c - offset$$

- ▶ z vorzeichenbehafteter Wert (Zahlenwert)
 - ▶ c binäre Ganzzahl (Code)
 - ▶ beliebig gewählter Offset
- Null wird nicht mehr durch 000...0 dargestellt
- + Größenvergleich zweier Zahlen bleibt einfach
- ▶ Anwendung: Exponenten im IEEE 754 Gleitkommaformat, einige Audioformate, Ausgaben von A/D-Wandlern ...





Exzess-Codierung: Beispiele

3.6 Ziffern und Zahlen - Darstellung negativer Zahlen

64-040 Rechnerstrukturen und Betriebssysteme

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

$$z = c - \text{offset}$$



b-Komplement

3.6 Ziffern und Zahlen - Darstellung negativer Zahlen

64-040 Rechnerstrukturen und Betriebssysteme

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

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

$$= 0, \quad \text{für } z = 0$$

- ▶ 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

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



$(b - 1)$ -Komplement

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

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

- ▶ 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

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

- ▶ im Fall $m = 0$ gilt offenbar $K_b(z) = K_{b-1}(z) + 1$
- ⇒ 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↔9 1↔8 2↔7 3↔6 4↔5
	Summe:	99 999 = 100 000 - 1



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

- ⇒ das b -Komplement kann sehr einfach berechnet werden:
es werden einfach die einzelnen Bits/Ziffern invertiert
und 1, bzw. b^{-m} an der niedrigsten Stelle aufaddiert

▶ 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↔9 1↔8 2↔7 3↔6 4↔5
	Summe:	100 000



Subtraktion mit b -Komplement

- ▶ bei Rechnung mit fester Stellenzahl n gilt:

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

weil b^n gerade nicht mehr in n Stellen hineinpasst

- ▶ also gilt für die Subtraktion auch:

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

⇒ Subtraktion kann durch Addition des b -Komplements ersetzt werden!
 Voraussetzung: begrenzte Stellenanzahl

- ▶ und für Integerzahlen gilt außerdem

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



Subtraktion mit Einer- und Zweierkomplement

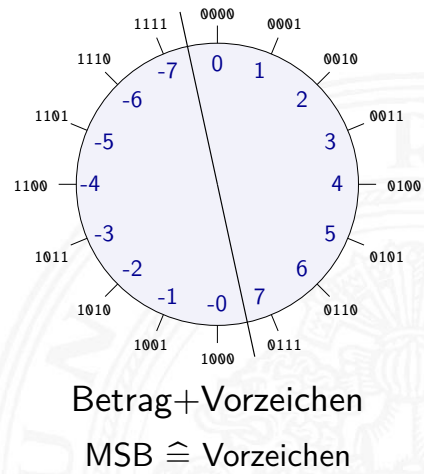
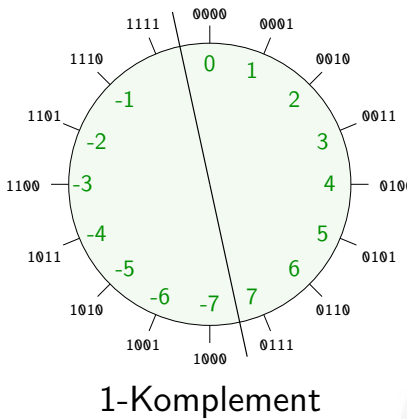
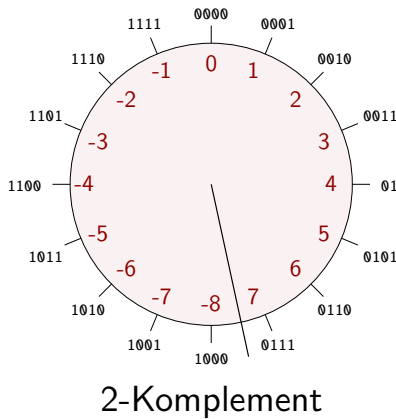
- ▶ Subtraktion ersetzt durch Addition des Komplements

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



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



Darstellung negativer Zahlen: Beispiele (8-bit)

N	+N	-N	-N	-N	-N
Dezimal	Binär	VZ+Betrag	1-Komplement	2-Komplement	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



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



Grundidee: **halblogarithmische Darstellung einer Zahl**

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

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

- ▶ 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 und Subtraktion sind dann sehr aufwändig



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

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

 - ▶ Schreibweise in C/Java: $\langle \text{Vorzeichen} \rangle \langle \text{Mantisse} \rangle E \langle \text{Exponent} \rangle$
- | | | |
|----------|-----------------------|-------------------------------|
| 6.023E23 | $6,023 \cdot 10^{23}$ | Avogadro-Zahl |
| 1.6E-19 | $1,6 \cdot 10^{-19}$ | Elementarladung des Elektrons |



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}

normalisierte Zahlen
 $\pm 0, \langle \text{mantisse} \rangle \cdot 10^{\pm \langle \text{exponent} \rangle}$



Gleitkomma: Historie

3.7 Ziffern und Zahlen - Gleitkomma und IEEE 754

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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, $\approx 50K$ FLOPS
- ▶ 1985 IEEE 754 Standard für Gleitkomma
- ▶ 1989 Intel 486 mit integriertem Koprozessor
- ▶ 1995 Java-Spezifikation fordert IEEE 754
- ▶ 1997 ASCI-RED: 1,1 TFLOPS (7 264 Pentium Pro)
- ▶ 2008 Roadrunner: 1,0 PFLOPS (12 240 Cell, 6 120 Opteron)
- ▶ 2022 Frontier: 1,1 EFLOPS (37 888 Instinct, 9 472 Epyc)

...

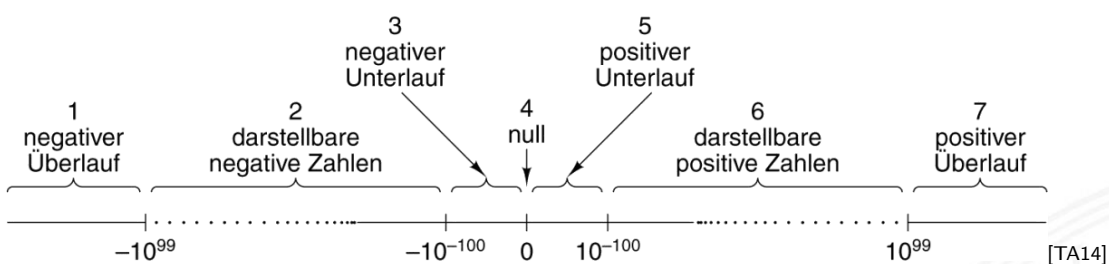
FLOPS := Floating-Point Operations Per Second



Gleitkomma: Zahlenbereiche

3.7 Ziffern und Zahlen - Gleitkomma und IEEE 754

64-040 Rechnerstrukturen und Betriebssysteme



[TA14]

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



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

- ▶ diese Darstellung ist bisher nicht eindeutig:

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

normalisierte Darstellung

- ▶ Exponent anpassen, bis Mantisse im Bereich $1 \leq m < b$ liegt
- ⇒ Darstellung ist dann eindeutig
- ⇒ im Dualsystem: erstes Vorkommabit ist dann 1 und muss nicht explizit gespeichert werden
- ▶ evtl. zusätzlich sehr kleine Zahlen nicht-normalisiert



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*)

V	Exponent	Mantisse	
1	8	23	bit

Exzess-127 Codierung für Exponent

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

V	Exponent	Mantisse	
1	11	52	bit

Exzess-1023 Codierung für Exponent

IEEE 754 Zahl	Exponent	Mantisse
normalisiert	00 ... 001 bis 11 ... 110	$1 \leq m < 2$ 1,m
denormalisiert	00 ... 000 ($2^{1-\text{exzess}}$)	$0 < m < 1$ 0,m
Null (+0, -0)	00 ... 000	$m = 0$
NaN, Infinity	11 ... 111	vergl. Folie 193



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 ... + 127	-1022 ... + 1023
kleinste normalisierte Zahl	2^{-126}	2^{-1022}
größte -"-	$\approx 2^{128}$	$\approx 2^{1024}$
kleinste nicht normalisierte Zahl	$\approx 10^{-45}$	$\approx 10^{-324}$
\cong Dezimalbereich	$\approx 10^{-38} \dots 10^{38}$	$\approx 10^{-308} \dots 10^{308}$
dezimale Genauigkeit [Stellen]	≈ 7	≈ 16

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



kürzere Zahlenformate

- ▶ großer Zahlenbereich gefordert, Genauigkeit weniger wichtig
- ▶ Bildverarbeitung (HDR), ML (maschinelles Lernen), ...
- + weniger Speicherbedarf und Rechenleistung, schnellere Datenübertragung

- ▶ 16-bit Format: halbe Genauigkeit (*half precision, binary16*)

V	Exponent	Mantisse	Exzess-15 Codierung für Exponent
1	5	10	bit

- ▶ *bfloat16*

V	Exponent	Mantisse	Exzess-127 Codierung für Exponent
1	8	7	bit

- ▶ ... viele weitere Minifloat-Formate, sogar 8-bit



Beispiele: float

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

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

- ▶ 1 1000 0000 1110 0000 0000 0000 0000 0000

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

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

- ▶ 0 1111 1110 0001 0011 0000 0000 0000 0000

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

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



Beispiele: float (cont.)

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

- ▶ 1 0000 0001 0000 0000 0000 0000 0000 0000

$$\begin{aligned} z &= -1 \cdot 2^{(1-127)} \cdot (1 + 0 + 0 + \dots + 0) \\ &= -1 \cdot 2^{-126} \cdot 1,0 = -1,17549435 \cdot 10^{-38} \end{aligned}$$

- ▶ 0 0111 1111 0000 0000 0000 0000 0000 001

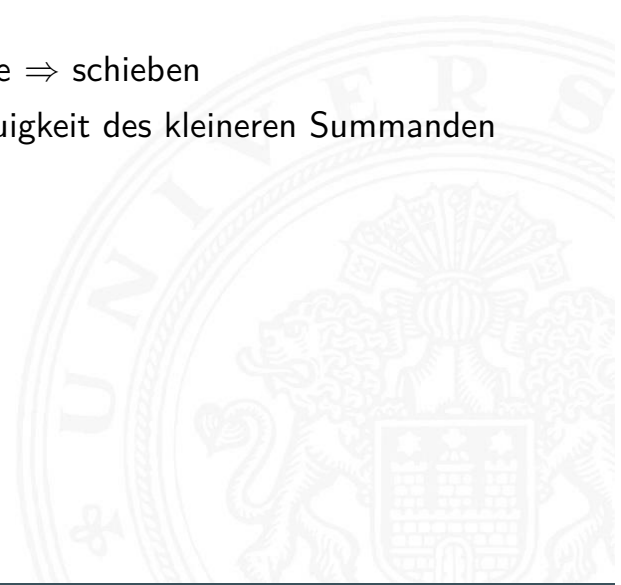
$$\begin{aligned} z &= +1 \cdot 2^{(127-127)} \cdot (1 + 2^{-23}) \\ &= 1 \cdot (1 + 0,00000012) = 1,00000012 \end{aligned}$$



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





Gleitkoma-Addition: Beispiel

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

$$\begin{aligned} y &= (a + b) \\ &= (9,725 \cdot 10^7 + 0,3016 \cdot 10^7) \\ &= (9,725 + 0,3016) \cdot 10^7 \\ &= (10,0266) \cdot 10^7 \\ &= 1,00266 \cdot 10^8 \\ &= 1,003 \cdot 10^8 \end{aligned}$$

Angleichung der Exponenten

Distributivgesetz

Addition der Mantissen

Normalisierung

Runden bei fester Stellenzahl

- ▶ 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 unterscheiden sich nur in einigen, wenig signifikanten Stellen

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



Gleitkomma: Multiplikation, Division

3.7 Ziffern und Zahlen - Gleitkomma und IEEE 754

64-040 Rechnerstrukturen und Betriebssysteme

Multiplikation von Gleitkommazahlen $y = a_1 \cdot a_2$

- ▶ Multiplikation der Mantissen und Vorzeichen

Vorzeichen s_i ist hier -1^{sBit}

Anmerkung: Berechnung $sBit = sBit_1 \text{ XOR } sBit_2$

XOR (\oplus): Folie 292

- ▶ Addition der Exponenten
- ▶ ggf. Normalisierung des Resultats

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

Division entsprechend:

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

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



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

3.7 Ziffern und Zahlen - Gleitkomma und IEEE 754

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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: $2/0$, $-3/0$ 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 $V \mid 0 < Exp < Max \mid$ jedes Bitmuster

denormalisiert $V \mid 00...0 \mid$ jedes Bitmuster $\neq 00...0$

0 $V \mid 00...0 \mid 00...0$

Inf $V \mid 11...1 \mid 00...0$

NaN $V \mid 11...1 \mid$ jedes Bitmuster $\neq 00...0$

- ▶ Rechnen mit *Inf* funktioniert normal: $0/Inf = 0$
- ▶ *NaN* für undefinierte Werte: $\text{sqrt}(-1)$, $\text{arcsin}(2.0)$...
- ▶ 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)         = 1.4142135623730951
sqrt(-1)        = NaN
0 + NaN         = NaN
NaN == NaN      = false
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
- ⇒ Wertigkeit des LSB
- ⇒ 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.
- ▶ Programmiersprachenunterstützung, 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
- ▶ mathematisch komplexes Thema

- ▶ 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

segment word size	16 bit			32 bit				64 bit					
	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu, Clang	Intel Linux	Microsoft	Intel Windows	Gnu, Clang	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_t				8	8			8	8	8	8	8	8
enum (typical)	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	8
m128				16	16			16	16	16	16	16	16
m256				32	32			32	32	32	32	32	32
m512				64	64			64	64	64	64	64	64
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

www.agner.org/optimize/calling_conventions.pdf



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

[If10] 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.)

3.9 Ziffern und Zahlen - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [Gol91] D. Goldberg: *What every computer scientist should know about floating-point.*
in: *ACM Computing Surveys* 23 (1991), March, Nr. 1, S. 5–48.
docs.oracle.com/cd/E19957-01/800-7895/800-7895.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
www-cs-faculty.stanford.edu/~knuth/taocp.html
- [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.)

3.9 Ziffern und Zahlen - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript.*
Universität Hamburg, FB Informatik, 2005, Vorlesungsskript.
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 978-0-13-334301-4
- [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 978-3-519-02332-6



Gliederung

4 Arithmetik

64-040 Rechnerstrukturen und Betriebssysteme

1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. **Arithmetik**
 - Addition und Subtraktion
 - Multiplikation
 - Division
 - Höhere Funktionen
 - Mathematische Eigenschaften
 - Literatur
5. Zeichen und Text
6. Logische Operationen
7. Codierung



A. Mäder

204



Gliederung (cont.)

4 Arithmetik

64-040 Rechnerstrukturen und Betriebssysteme

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



A. Mäder

205



Wiederholung: Stellenwertsystem („Radixdarstellung“)

- ▶ Wahl einer geeigneten Zahlenbasis b („Radix“)
 - ▶ 10: Dezimalsystem
 - ▶ 16: Hexadezimalsystem (Sedezimalsystem)
 - ▶ 2: Dualsystem
- ▶ Menge der entsprechenden Ziffern $\{0, 1, \dots, b - 1\}$
- ▶ 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:

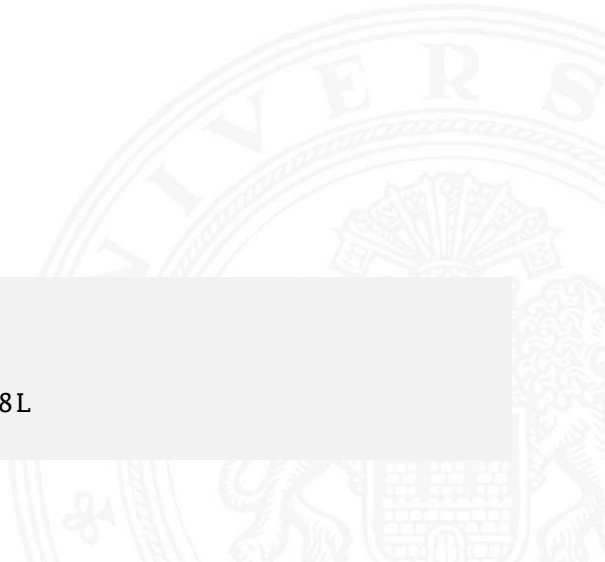
- ▶ Zahlenbereiche definiert in Headerdatei `/usr/include/limits.h`
LONG_MIN, LONG_MAX, ULONG_MAX etc.
- ▶ Zweierkomplement (signed), Ganzzahl (unsigned)
- ▶ die Werte sind plattformabhängig (!)

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

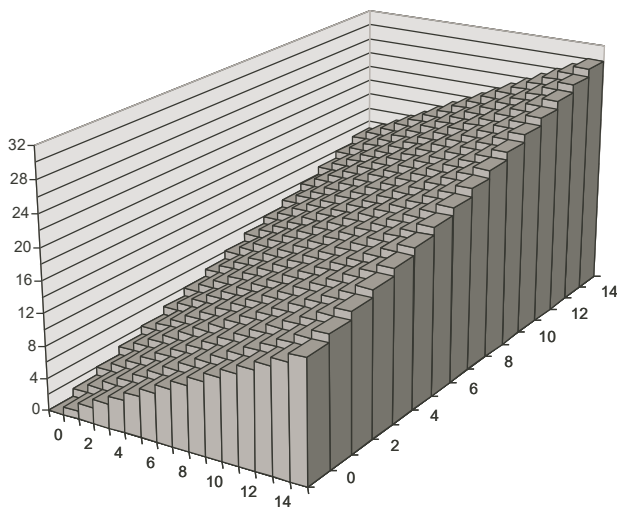
- ▶ Beispiel

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



unsigned Addition: Visualisierung

Integer addition



[BO15]

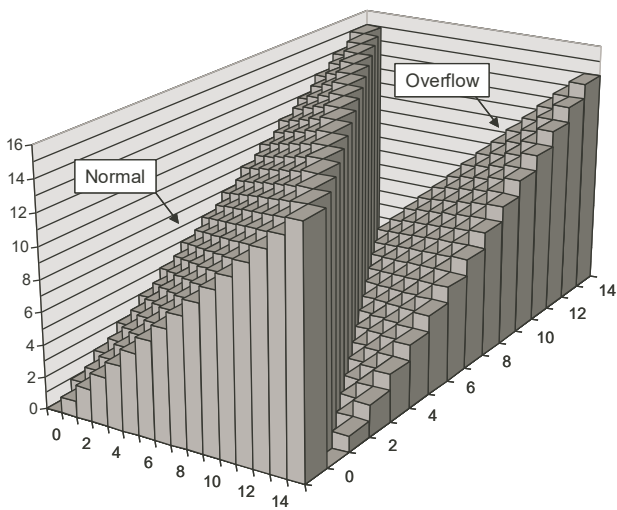
- ▶ Wortbreite der Operanden ist w , hier 4-bit
- ▶ Zahlenbereich der Operanden x, y ist $0 \dots (2^w - 1)$
- ▶ Zahlenbereich des Resultats s ist $0 \dots (2^{w+1} - 2)$





unsigned Addition: Visualisierung (cont.)

Unsigned addition (4-bit word)

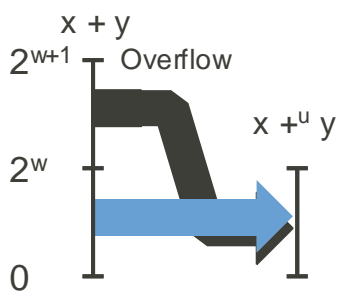


[BO15]

- ▶ Wortbreite der Operanden **und des Resultats** ist w
- ⇒ Überlauf, sobald das Resultat größer als $(2^w - 1)$
- ⇒ oberstes Bit geht verloren



unsigned Addition: Überlauf



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



Subtraktion im Dualsystem

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

$$\begin{array}{r}
 1011\ 0011 \\
 - 0011\ 1001 \\
 \hline
 \text{Ü } 1111 \\
 \hline
 111\ 1010
 \end{array}
 \qquad
 \begin{array}{r}
 = 179 \\
 = 57 \\
 \hline
 = 122
 \end{array}$$

- ▶ Alternative: Subtraktion durch Addition des b -Komplements ersetzen



Subtraktion mit b -Komplement

- ▶ bei Rechnung mit fester Stellenzahl n gilt:

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

weil b^n gerade nicht mehr in n Stellen hineinpasst

- ▶ also gilt für die Subtraktion auch:

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

⇒ Subtraktion kann durch Addition des b -Komplements ersetzt werden!
 Voraussetzung: begrenzte Stellenanzahl

- ▶ und für Integerzahlen gilt außerdem

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



- ▶ 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 ersetzt durch Addition des Komplements

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

- ▶ **(b-1)-Komplement** der Zahl z

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

$$= 0 \quad \text{für } z = 0$$
- b-Komplement** der Zahl z

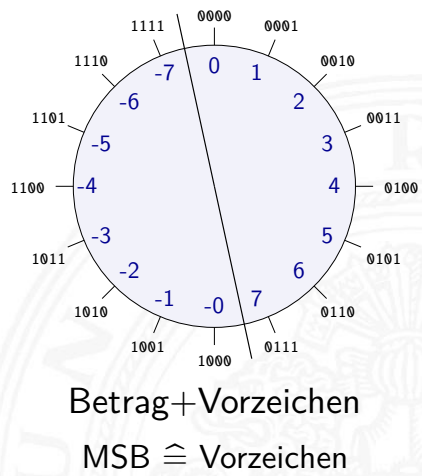
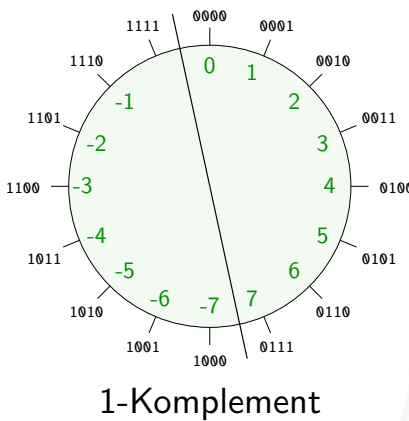
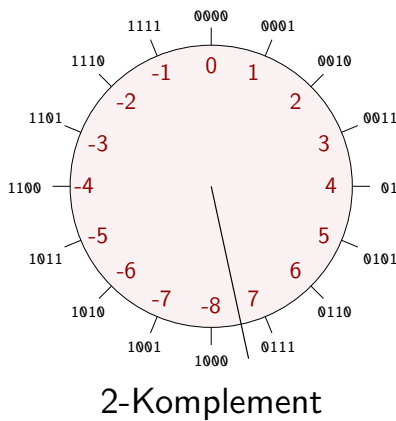
$$K_b(z) = b^n - z \quad \text{für } z \neq 0$$

$$= 0 \quad \text{für } z = 0$$



Veranschaulichung: Zahlenkreis

Beispiel für 4-bit Zahlen

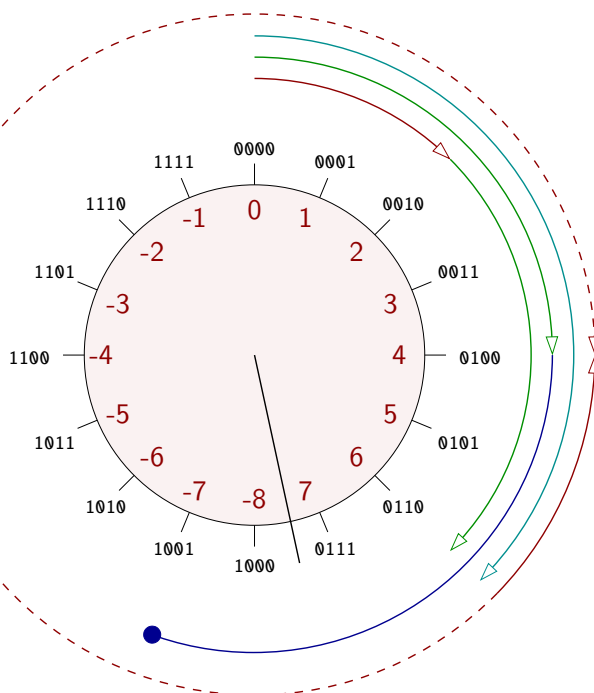


- ▶ Komplement-Arithmetik als Winkeladdition
- ▶ Web-Anwendung: *Visualisierung im Zahlenkreis* (JavaScript, aus [Kor16])



Zahlenkreis: Addition, Subtraktion

2-Kompl.



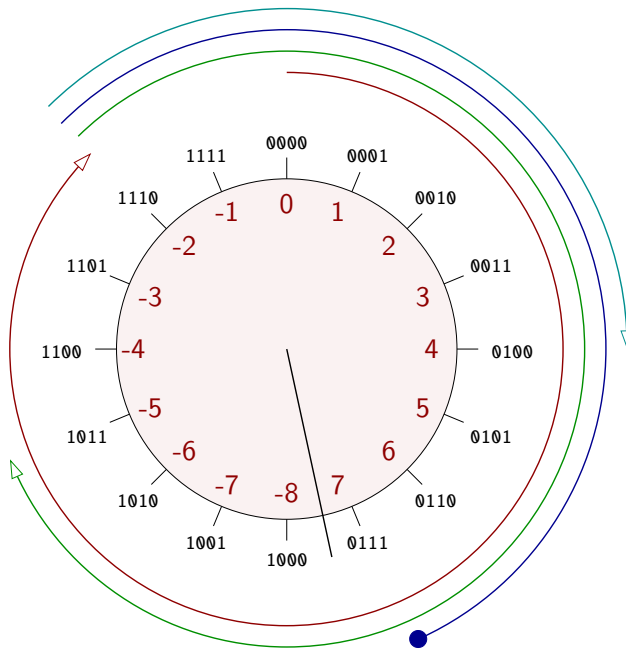
$$\begin{aligned} 0010+0100 &= 0110 \\ 0100+0101 &= 1001 \\ 0110-0010 &= 0100 \end{aligned}$$

$$\begin{aligned} 0010 & 1110 \\ 0100 & \\ 0101 & \\ 0110 & \end{aligned}$$



Zahlenkreis: Addition, Subtraktion (cont.)

2-Kompl.



$$1110 + 1101 = 1011$$

$$1110 + 1001 = \mathbf{0111}$$

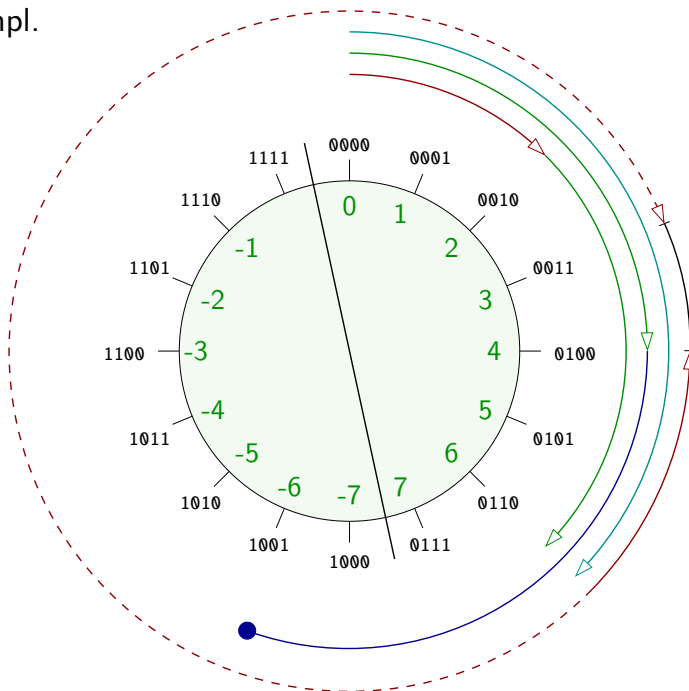
$$1110 + 0110 = \mathbf{0100}$$

1110
1101
1001
0110



Zahlenkreis: Addition, Subtraktion (cont.)

1-Kompl.



$$0010 + 0100 = 0110$$

$$0100 + 0101 = \mathbf{1001}$$

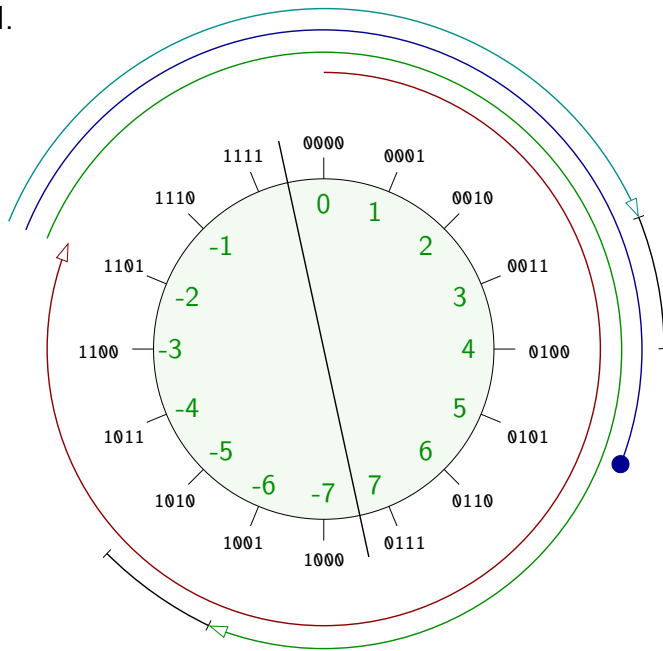
$$0110 + 1101 + 1 = \mathbf{0100}$$

0010 1101
0100
0101
0110



Zahlenkreis: Addition, Subtraktion (cont.)

1-Kompl.



$$1101 + 1100 + 1 = 1010$$

$$1101 + 1000 = \mathbf{0101}$$

$$1101 + 0110 + 1 = \mathbf{0100}$$

1101
 1100
 1000
 0110



in C: unsigned Zahlen

- ▶ 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

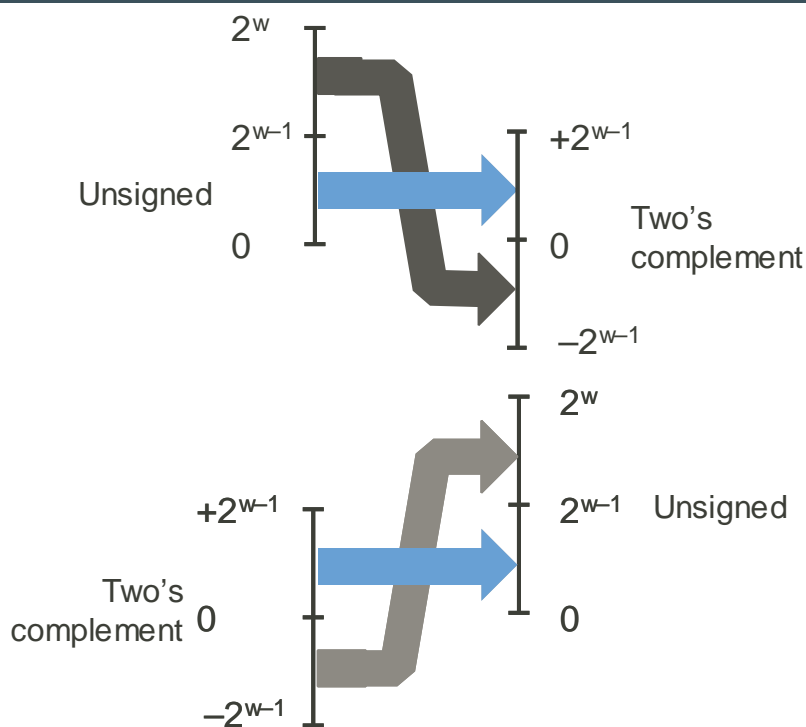
```
short int      x = 15213;
unsigned short int ux = (unsigned short) x; // 15213

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 Fehler
2147483647	>	-2147483648	signed	1
2147483647U	>	-2147483648	unsigned	0
2147483647	>	(int) 2147483648U	signed	1
-1	>	-2	signed	1
(unsigned) -1	>	-2	unsigned	1



Sign-Extension

- ▶ Gegeben: w -bit Integer x
- ▶ Umwandeln in $w + k$ -bit Integer x' mit gleichem Wert?
- ▶ **Sign-Extension:** Vorzeichenbit kopieren

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

- ▶ 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



Java Puzzlers No.5

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

```
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?



Java Puzzlers No.5

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

```
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?

$$\begin{array}{r}
 0xffffffffcafebabe \quad \text{sign-extension!} \\
 0x0000000100000000 \\
 \hline
 \ddot{U} 11111110 \\
 \hline
 00000000cafebabe
 \end{array}$$



Ariane-5 Absturz

- ▶ 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
- ▶ 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 – sehr einfach: $\cdot 0 / \cdot 1$

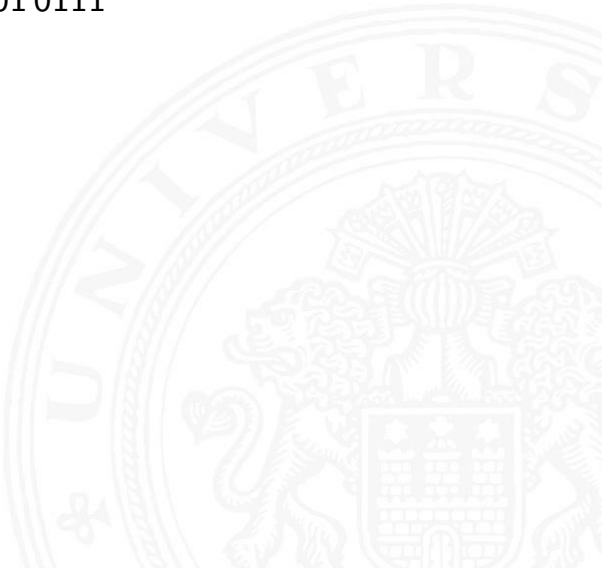
·	0	1
0	0	0
1	0	1



Multiplikation im Dualsystem (cont.)

▶ Beispiel

$$\begin{array}{r}
 10110011 \cdot 1101 \\
 \hline
 10110011 \quad 1 \\
 10110011 \quad 1 \\
 00000000 \quad 0 \\
 10110011 \quad 1 \\
 \hline
 \text{Ü } 11101111 \\
 \hline
 100100010111
 \end{array}
 = 179 \cdot 13 = 2327 = 1001\ 0001\ 0111 = 0x917$$

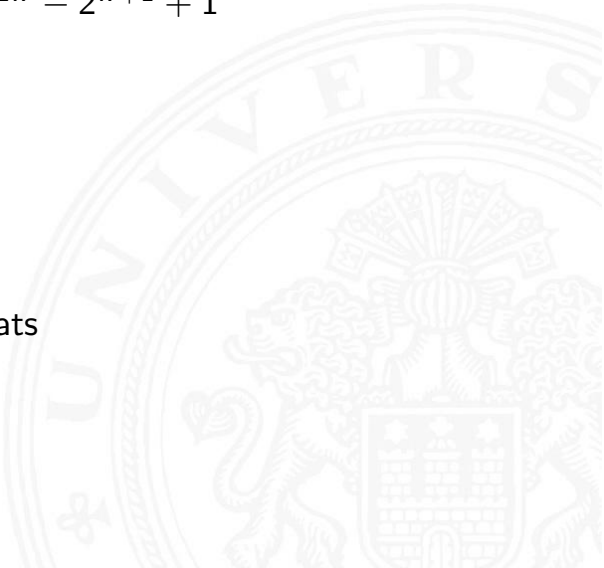


unsigned Multiplikation

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

⇒ 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





- ▶ Zahlenbereich der Operanden: $-2^{w-1} \dots (2^{w-1} - 1)$
 - ▶ Zahlenbereich des Resultats: $-2^{w-1} \cdot (2^{w-1} - 1) \dots (2^{2w-2})$
- ⇒ 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
- ⇒ 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 );  
}
```



Java Puzzlers No. 3

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

4.2 Arithmetik - Multiplikation

64-040 Rechnerstrukturen und Betriebssysteme

```
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 wird mit 32-bit berechnet, dabei Überlauf
 - ▶ Konvertierung nach 64-bit long erst bei der Zuweisung
- ⇒ long-Konstante schreiben: `24L * 60 * 60 * 1000 * 1000`

A. Mäder

234



Division im Dualsystem

4.3 Arithmetik - Division

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ $d = a/b$ 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

A. Mäder

235



Division im Dualsystem (cont.)

► Beispiele

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

$$\begin{array}{r}
 1100100 \ / \ 11 = 0100001 \\
 \begin{array}{r}
 1 \qquad \qquad \qquad 0 \\
 11 \qquad \qquad \qquad 1 \\
 -11 \\
 \hline
 0 \qquad \qquad \qquad 0 \\
 0 \qquad \qquad \qquad 0 \\
 1 \qquad \qquad \qquad 0 \\
 10 \qquad \qquad \qquad 0 \\
 100 \qquad \qquad \qquad 1 \\
 -11 \\
 \hline
 1 \qquad \qquad \qquad 1 \text{ (Rest)}
 \end{array}
 \end{array}$$



Division im Dualsystem (cont.)

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

$$\begin{array}{r}
 1011011 \ / \ 1101 = 0111 \\
 \begin{array}{r}
 1011 \qquad \qquad \qquad 0 \\
 10110 \qquad \qquad \qquad 1 \\
 -1101 \\
 \hline
 10011 \qquad \qquad \qquad 1 \\
 -1101 \\
 \hline
 01101 \qquad \qquad \qquad 1 \\
 -1101 \\
 \hline
 0
 \end{array}
 \end{array}$$



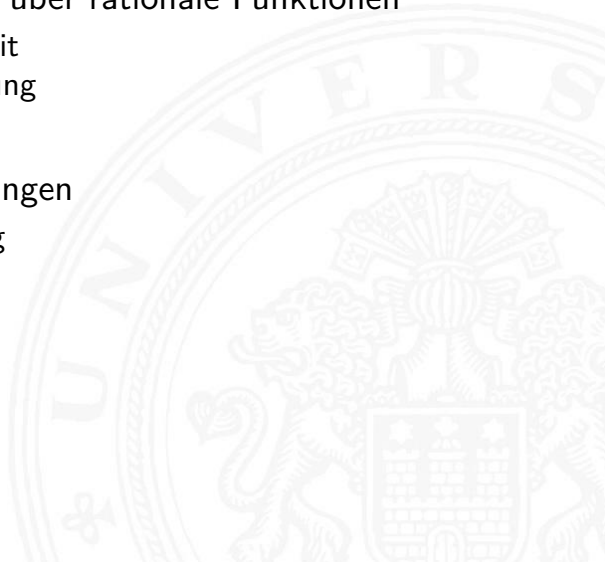


Berechnung von \sqrt{x} , $\log x$, $\exp x$, $\sin x \dots ?$

- ▶ Approximation über Polynom (Taylor-Reihe) bzw. 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



- ▶ 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 $x = 3$, $y_0 = 0,5$:

$$\begin{aligned}
 y_1 &= 0,5 \cdot (2 - 3 \cdot 0,5) &&= 0,25 \\
 y_2 &= 0,25 \cdot (2 - 3 \cdot 0,25) &&= 0,3125 \\
 y_3 &= 0,3125 \cdot (2 - 3 \cdot 0,3125) &&= 0,33203125 \\
 y_4 &= 0,3332824 \\
 y_5 &= 0,333333332557231 \\
 y_6 &= 0,333333333333333
 \end{aligned}$$





Quadratwurzel: Heron-Verfahren für \sqrt{x} Babylonisches Wurzelziehen

4.4 Arithmetik - Höhere Funktionen

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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

A. Mäder

240



Informationstreue

4.5 Arithmetik - Mathematische Eigenschaften

64-040 Rechnerstrukturen und Betriebssysteme

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

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

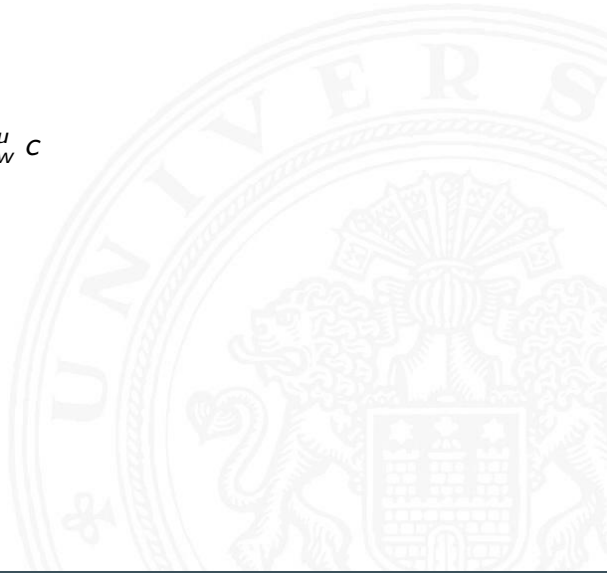
A. Mäder

241



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 $a \oplus_w^u \bar{a} = 0; \bar{a} = 2^w - a$



signed Arithmetik

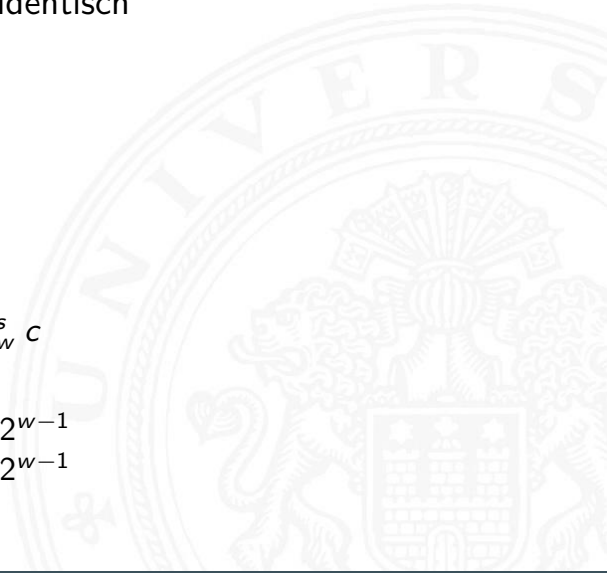
2-Komplement

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

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

⇒ 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 $a \oplus_w^s \bar{a} = 0; \bar{a} = -a, a \neq -2^{w-1}$
 $a, a = -2^{w-1}$





unsigned Arithmetik

- ▶ Wortbreite auf w begrenzt
- ▶ Modulo-Arithmetik $a \otimes_w^u b = (a \cdot b) \bmod 2^w$
- ▶ \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 $a \otimes_w^u (b \oplus_w^u c) = (a \otimes_w^u b) \oplus_w^u (a \otimes_w^u c)$



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
- ▶ isomorph zum Ring der ganzen Zahlen *modulo* 2^w
- ▶ Ordnungsrelation im Ring der ganzen Zahlen
 - ▶ $a > 0 \implies a + b > b$
 - ▶ $a > 0, b > 0 \implies a \cdot b > 0$
 - ▶ diese Relationen **gelten nicht** bei Rechnerarithmetik!

2-Komplement

Überlauf



Gleitkomma Addition

Vergleich mit kommutativer Gruppe

4.5 Arithmetik - Mathematische Eigenschaften

64-040 Rechnerstrukturen und Betriebssysteme

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

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



Gleitkomma Multiplikation

Vergleich mit kommutativem Ring

4.5 Arithmetik - Mathematische Eigenschaften

64-040 Rechnerstrukturen und Betriebssysteme

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

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



Literatur

4.6 Arithmetik - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [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
- [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.)

4.6 Arithmetik - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [Omo94] A.R. Omondi: *Computer Arithmetic Systems – Algorithms, Architecture and Implementations.*
Prentice-Hall International, 1994. ISBN 978-0-13-334301-4
- [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 978-3-519-02332-6



Gliederung

5 Zeichen und Text

64-040 Rechnerstrukturen und Betriebssysteme

1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
 - Ad-Hoc Codierungen
 - ASCII und ISO-8859
 - Unicode
 - Tipps und Tricks
 - Base64-Codierung
 - Literatur
6. Logische Operationen
7. Codierung



Gliederung (cont.)

5 Zeichen und Text

64-040 Rechnerstrukturen und Betriebssysteme

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





Wiederholung: Zeichen

5 Zeichen und Text

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ **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 Alphabets, meistens auch mit Sonderzeichen (Leerzeichen, Punkt, Komma usw.)

A. Mäder

252



Wiederholung: Zeichen (cont.)

5 Zeichen und Text

64-040 Rechnerstrukturen und Betriebssysteme

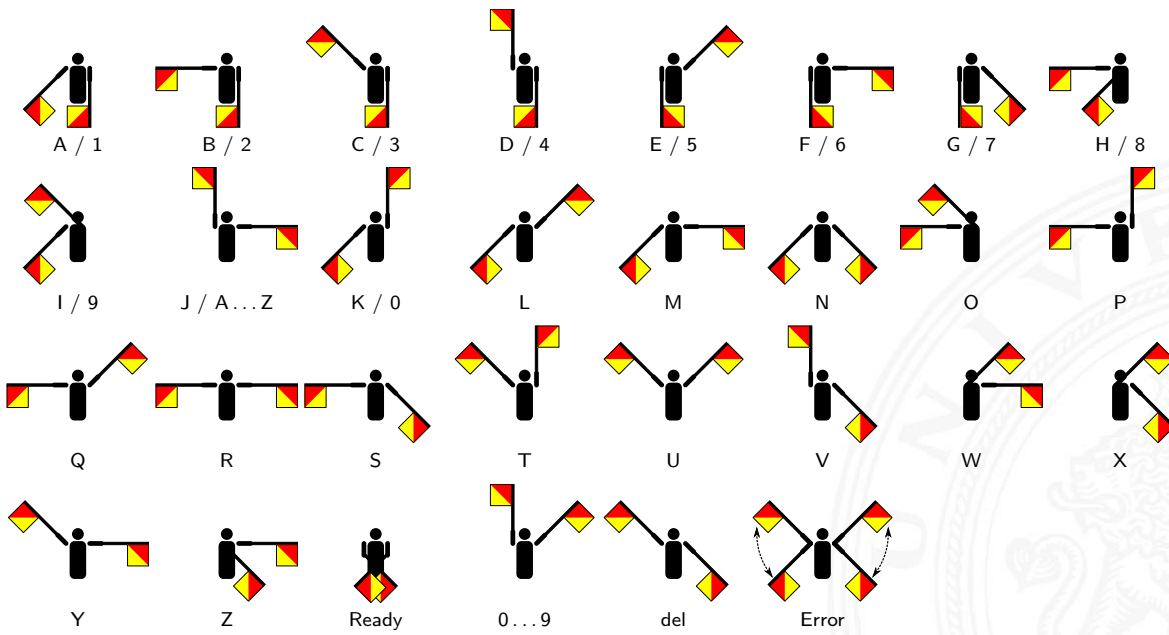
- ▶ **Alphabet:** engl. *alphabet*
Ein in vereinbarter Reihenfolge geordneter Zeichenvorrat
- ▶ **Zeichenkette:** Eine Folge von Zeichen engl. *string*
- ▶ **Wort:** engl. *word*
Zeichenkette, die in einem gegebenen Zusammenhang als Einheit bezeichnet wird
- ▶ Worte aus 8 Binärzeichen (8 bit) werden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

A. Mäder

253



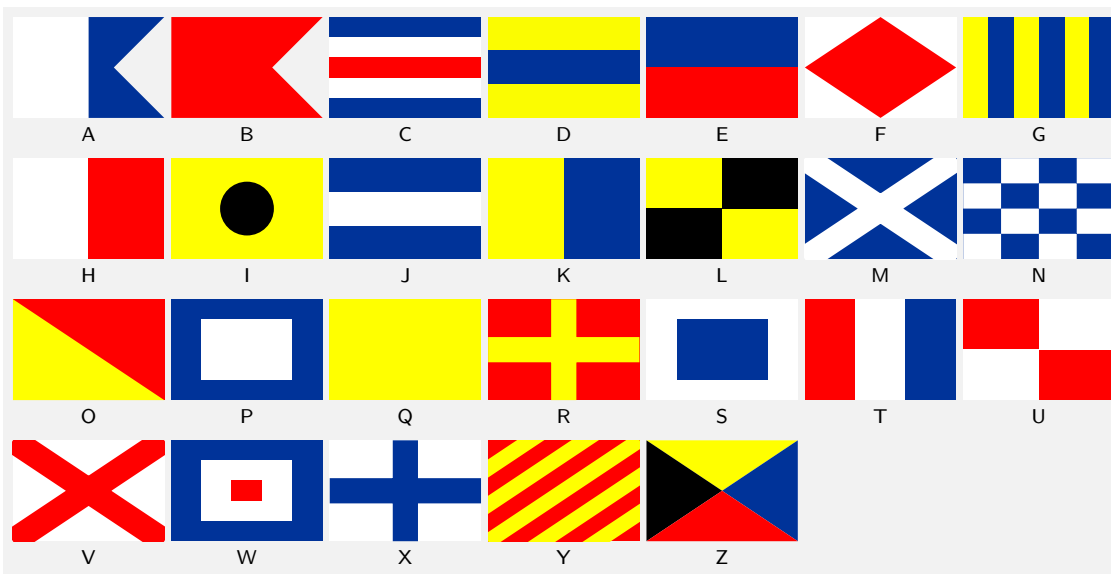
Flaggen-Signale



de.wikipedia.org/wiki/Winkeralphabet



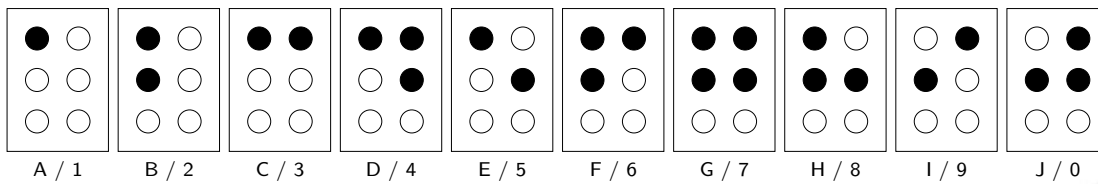
Flaggen-Alphabet



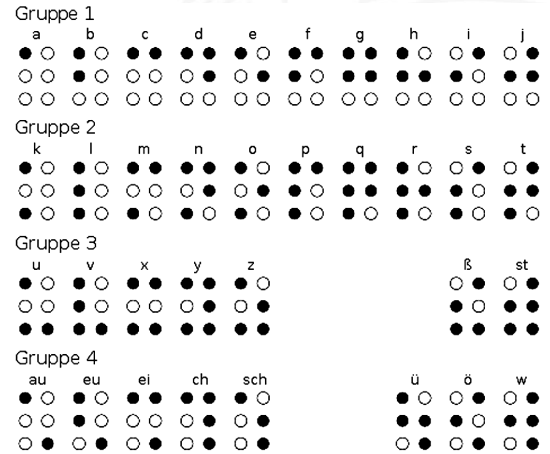
de.wikipedia.org/wiki/Flaggenalphabet



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 •••	. •–•–•–
B –•••	T –	, –•••–
C –•–•	U ••–	? ••–•••
D –••	V •••–	' •–•–••
E •	W •–•	! –•–•–
F ••–•	X –••–	/ –••–•
G –•–•	Y –•–•–	(–•–••
H ••••	Z –•••) –•–•–
I ••	0 –•–•–	& •–•••
J •–•–	1 •–•–•–	: –•–•••
K –•–	2 ••–•–	; –•–•••
L •–••	3 •••–•–	= –•••–
M –•–	4 ••••–	+ •–•••
N –•	5 •••••	- –••••–
O –•–•	6 –••••	– ••–•–
P •–••	7 –••••	" •–•••
Q –•–•	8 –•–••	\$ ••••–•–
R •–•	9 –•–••	@ •–•••
		S-Start –•–•–
		Verst. •••–•
		S-Ende •–•–•
		V-Ende •••–•–
		Error ••••••••
		Ä •–•–
		À •–•–
		É ••–••
		È •–••–
		Ö –•–•
		Ü ••–•
		ß •••–•••
		CH –•–•–
		Ñ –•–•–
		...
		SOS ••• –•–•••



Morse-Code (cont.)

► Eindeutigkeit Codewort: ● ● ● ● ● - ●

E	●
I	● ●
N	- ●
R	● - ●
S	● ● ●

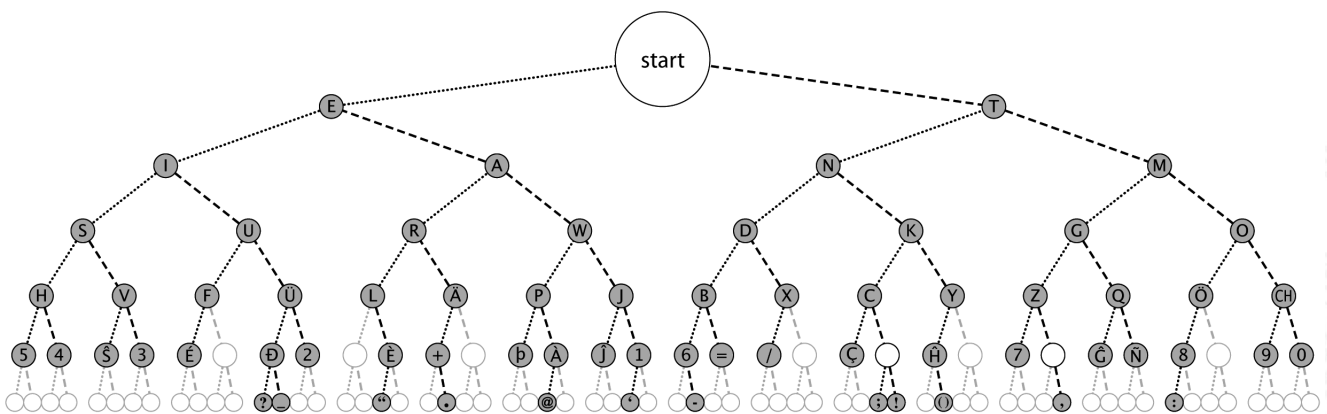
- bestimmte Morse-Sequenzen sind mehrdeutig
- Pause zwischen den Symbolen notwendig

► 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 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

de.wikipedia.org/wiki/ASCII

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



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...	<i>nicht belegt</i>															
1...	<i>nicht belegt</i>															
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...	<i>nicht belegt</i>															
9...	<i>nicht belegt</i>															
A...	<i>NBSP</i>	ı	ø	£	¤	¥	¦	§	¨	©	ª	«	¬	<i>SHY</i>	®	–
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	Ꞥ	Ł	Ś	ş	ˆ	Š	Ş	Ť	Ž	SHY	Ž	Ž
B...	°	ą	˙	ł	´	ł	ś	˘	š	ş	ť	ž	˝	ž	ž	
C...	Ř	Á	Â	Ă	Ä	Á	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
D...	Đ	Ń	Ň	Ó	Ô	Õ	Ö	×	Ř	Ú	Ú	Û	Ü	Ý	Ť	ß
E...	ř	á	â	ă	ä	á	ć	ç	č	é	ę	ë	ě	í	î	ď
F...	đ	ń	ň	ó	ô	õ	ö	÷	ř	ú	ú	û	ü	ý	ť	·



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	ı	ø	£	€	¥	Š	š	©	ª	«	¬	SHY	®	¯	
B...	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	ÿ	ı
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
- ▶ 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	â	ç	ê	ë	è	ï	î	ì	Ä	Å
9...	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	×	f
A...	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	½	¼	¡	«	»
B...	⌘	⌘	⌘		†	Á	Â	À	©	¶	¶	¶	¶	¢	¥	₹
C...	Ł	ł	ł	ł	—	†	ã	Ã	ℓ	ℓ	ℓ	ℓ	ℓ	=	†	¤
D...	ð	Ð	É	È	È	ı	í	î	ï	ı	ı	ı	ı	ı	ı	ı
E...	Ó	ß	Ô	Ò	ö	Ö	µ	þ	þ	Ú	Û	Ù	ý	Ý	—	·
F...		±	=	¾	¶	§	÷	ˆ	°	ˆ	·	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

5.3 Zeichen und Text - Unicode

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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**
Spezialbefehle zur Umschaltung zwischen mehreren Zeichensätzen,
sog. *Escapesequenzen*
- ⇒ **Unicode** als System zur Codierung aller Zeichen aller bekannten Schriftsysteme
auch für tote Schriften/Sprachen

A. Mäder

270



Unicode: Versionen und History

5.3 Zeichen und Text - Unicode

64-040 Rechnerstrukturen und Betriebssysteme

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

- ▶ 1991 1.0.0: europäisch, nahöstlich, indisch
- ▶ 1992 1.0.1: ostasiatisch (Han)
- ▶ 1993 akzeptiert als ISO/IEC-10646 Standard
...
- ▶ 2023 15.1.0: inzwischen 149 813 Zeichen
 - ▶ Sprachzeichen, Hieroglyphen etc.
 - ▶ Symbole: Satzzeichen, Währungen (\$... €), Pfeile, mathematisch, technisch, Braille, Noten etc.
 - ▶ Emojis (3 782 aktuell) / Kombinationen

home.unicode.org unicode.org/main.html de.wikipedia.org/wiki/Unicode symb1.cc/de/unicode/table

A. Mäder

271



Unicode: Schreibweise

5.3 Zeichen und Text - Unicode

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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 2¹⁶ Zeichen
- ▶ Erweiterung um „*Extended Planes*“
- ▶ U+10000 ... U+10FFFF

A. Mäder

272



Unicode: in Webseiten (HTML)

5.3 Zeichen und Text - Unicode

64-040 Rechnerstrukturen und Betriebssysteme

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

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

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

A. Mäder

273



Unicode: Beispieltabellen

Latin-Zeichen

- ▶ U+0000 bis U+007F: ASCII www.unicode.org/charts/PDF/U0000.pdf
- ▶ U+0100 bis U+017F: Latin-A www.unicode.org/charts/PDF/U0100.pdf
Europäische Umlaute und Sonderzeichen
- ▶ ab U+0180 weitere Sonderzeichen: Latin-B, Latin-C usw.

Symbole 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

Asiatische Sprachen: Chinesisch (traditional/simplified), Japanisch, Koreanisch

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



Unicode: Java2D Fontviewer

Displaying Unicode 0600 to 06CF

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
- ⇒ 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
Violenschlüssel	U+1D11E	00000001 11010001 00011110	11110000 10011101 10000100 10011110	F0 9D 84 9E

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: Codierung

5.3 Zeichen und Text - Unicode

64-040 Rechnerstrukturen und Betriebssysteme

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 ***...*
- ▶ theoretisch bis zu sieben Folgebytes a 6-bit: max. 2^{42} Zeichen

A. Mäder

280



Sprach-Einstellungen: Locale

5.4 Zeichen und Text - Tipps und Tricks

64-040 Rechnerstrukturen und Betriebssysteme

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

A. Mäder

281



Textkonvertierungen

- ▶ `dos2unix`, `unix2dos`: Umwandeln von Dateien (z.B. Programm-Quelltexte) zwischen DOS/Windows und Unix/Linux: Codierung und Zeilenenden

```
dos2unix -h           Optionen anzeigen / Hilfe
dos2unix -ascii -n a.txt b.txt  nur Umbrüche (von a.txt nach b.txt)
dos2unix -iso -n a.txt b.txt    Umbrüche und ISO-8851-1
unix2dos -850 -n a.txt b.txt    Umbrüche und Codepage 850
```

- ▶ `iconv`: „Universalwerkzeug“ zur Umwandlung von Textcodierungen

```
iconv -l             Liste der unterstützten Codierungen
iconv -f <encoding> ... Codierung der Eingabedatei
iconv -t <encoding> ... Codierung der Ausgabedatei
iconv -o <filename> ... Name der Ausgabedatei
```

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

- ▶ Konvertierungsfunktionen in den meisten Texteditoren enthalten!



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

⇒ Verfahren zur Umcodierung der Datei in 7-bit ASCII notwendig

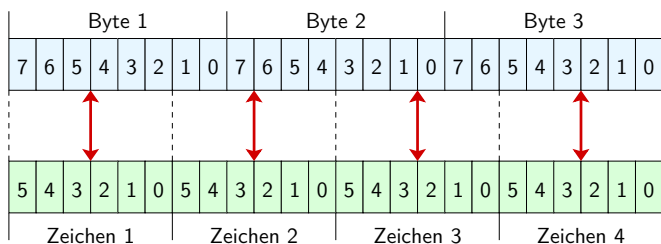
⇒ etabliert ist das **Base64** Verfahren (RFC 2045)

- ▶ 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



- ▶ $3 \times 8\text{-bit} \Leftrightarrow 4 \times 6\text{-bit}$
- ▶ 6-bit Binärwerte: 0...63
- ▶ nutzen 64 (von 128) 7-bit ASCII Symbolen

2. Zahl ASCII Zuordnung der ASCII-Zeichen

Zahl	ASCII	Zuordnung der ASCII-Zeichen
0...25	A...Z	
26...51	a...z	
52...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
Base64-encoded	T	W	F

- ▶ 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



- ▶ in neueren Java Versionen direkt im JDK enthalten
Module `java.base`, Package `java.util`: `Base64Encoder`, bzw. `Base64Decoder`

- ▶ diverse andere Packages
 - ▶ Apache Commons Codec (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* [Ull21]
openbook.rheinwerk-verlag.de/javainssel/05_008.html#u5.8.4
 - ▶ ...



- [Uni] The Unicode Consortium; Mountain View, CA.
home.unicode.org, unicode.org/main.html
- [JavaI] Oracle Corporation: *The Java Tutorials – Trail: Internationalization*.
docs.oracle.com/javase/tutorial/i18n
- [JavaD] Oracle Corporation: *Java SE Downloads*.
www.oracle.com/java/technologies/downloads
- [Ull21] C. Ullenboom: *Java ist auch eine Insel – Einführung, Ausbildung, Praxis*.
16. Auflage, Rheinwerk Verlag GmbH, 2021. ISBN 978-3-8362-8745-6
15. Auflage Online verfügbar: openbook.rheinwerk-verlag.de/javainssel,
bzw. www.tutego.de/javabuch



Gliederung

6 Logische Operationen

64-040 Rechnerstrukturen und Betriebssysteme

1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. **Logische Operationen**
 - Boole'sche Algebra
 - Boole'sche Operationen
 - Bitweise logische Operationen
 - Schiebeoperationen
 - Anwendungsbeispiele
 - Literatur
7. Codierung



Gliederung (cont.)

6 Logische Operationen

64-040 Rechnerstrukturen und Betriebssysteme

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





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 → Datenverarbeitung
 Fehlererkennung/-korrektur → Datenübertragung
 Codierung → 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: $\neg a, \bar{a}, \sim a$
▶ UND	–"– $a \wedge b, a \& b$
▶ ODER	–"– $a \vee b, a b$
▶ 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	0
1	0	0	1

OR(x,y)

x	y	0	1
0	0	0	1
1	1	1	1

XOR(x,y)

x	y	0	1
0	0	0	1
1	1	1	0

- ▶ alle logischen Operationen lassen sich mit diesen Funktionen darstellen
- ⇒ *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



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	$x \cap y$	$x \wedge y$	<code>x && y</code>
0 0 1 0		Inhibition	$x < y$		<code>x < y</code>
0 0 1 1		Identität y	y		y
0 1 0 0		Inhibition	$x > y$		<code>x > y</code>
0 1 0 1		Identität x	x		x
0 1 1 0		Antivalenz/XOR	$x \neq y$	$x \oplus y$	<code>x != y</code>
0 1 1 1		OR	$x \cup y$	$x \vee y$	<code>x y</code>
1 0 0 0		NOR	$\neg(x \cup y)$	$\overline{x \vee y}$	<code>!(x y)</code>
1 0 0 1		Äquivalenz/XNOR	$x = y$	$\overline{(x \oplus y)}$	<code>x == y</code>
1 0 1 0		NICHT x	$\neg x$	\bar{x}	<code>!x</code>
1 0 1 1		Implikation	$x \leq y$	$x \rightarrow y$	<code>x <= y</code>
1 1 0 0		NICHT y	$\neg y$	\bar{y}	<code>!y</code>
1 1 0 1		Implikation	$x \geq y$	$x \leftarrow y$	<code>x >= y</code>
1 1 1 0		NAND	$\neg(x \cap y)$	$\overline{x \wedge y}$	<code>!(x && y)</code>
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

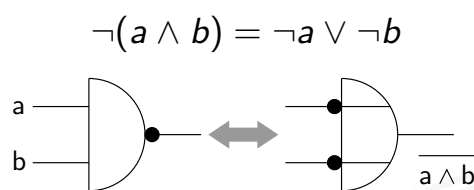
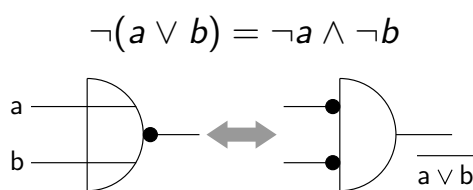


Rechenregeln: Ring / Algebra

Eigenschaft	Ring der ganzen Zahlen	Boole'sche Algebra
Kommutativgesetz	$a + b = b + a$ $a \cdot b = b \cdot a$	$a \vee b = b \vee a$ $a \wedge b = b \wedge a$
Assoziativgesetz	$(a + b) + c = a + (b + c)$ $(a \cdot b) \cdot c = a \cdot (b \cdot c)$	$(a \vee b) \vee c = a \vee (b \vee c)$ $(a \wedge b) \wedge c = a \wedge (b \wedge c)$
Distributivgesetz	$a \cdot (b+c) = (a \cdot b) + (a \cdot c)$	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
Identitäten	$a + 0 = a$ $a \cdot 1 = a$	$a \vee 0 = a$ $a \wedge 1 = a$
Vernichtung	$a \cdot 0 = 0$	$a \wedge 0 = 0$
Auslöschung	$\neg(\neg a) = a$	$\neg(\neg a) = a$
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



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

- ▶ „entweder a oder b “ (ausschließlich), bzw. „ a ungleich b “ \Rightarrow Antivalenz

XOR(x,y)

x	y	0	1
0	0	0	1
1	1	1	0

- ▶ $a \oplus b = (\neg a \wedge b) \vee (a \wedge \neg b)$
genau einer von den Termen a und b ist wahr
- ▶ $a \oplus b = (a \vee b) \wedge \neg(a \wedge b) = (a \vee b) \wedge (\neg a \vee \neg b)$
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
- \Rightarrow völlig andere Funktion als die bitweisen Operationen

Achtung!



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
<code>!0x41</code>	<code>0x00</code>
<code>!0x00</code>	<code>0x01</code>
<code>!!0x00</code>	<code>0x00</code>
<code>0x69 && 0x55</code>	<code>0x01</code>
<code>0x69 0x55</code>	<code>0x01</code>



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
<code>x</code>	<code>0110 0110</code>	<code>x</code>	<code>0000 0001</code>
<code>y</code>	<code>1001 0011</code>	<code>y</code>	<code>0000 0001</code>
<code>x & y</code>	<code>0000 0010</code>	<code>x && y</code>	<code>0000 0001</code>
<code>x y</code>	<code>1111 0111</code>	<code>x y</code>	<code>0000 0001</code>
<code>~x ~y</code>	<code>1111 1101</code>	<code>!x !y</code>	<code>0000 0000</code>
<code>x & ~y</code>	<code>0110 0100</code>	<code>x && !y</code>	<code>0000 0000</code>



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

- ▶ $\langle \text{condition} \rangle ? \langle \text{true-expression} \rangle : \langle \text{false-expression} \rangle$
- ▶ 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

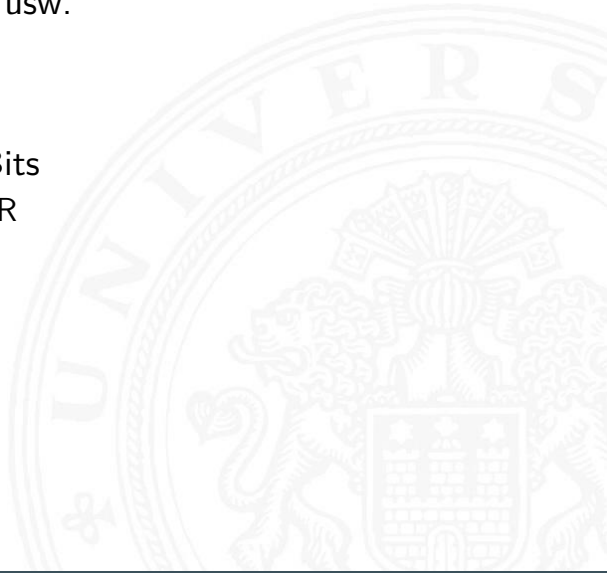
- ▶ $\langle \text{condition} \rangle ? \langle \text{true-expression} \rangle : \langle \text{false-expression} \rangle$
- ▶ Beispiel: `(x < 0) ? -x : x` Absolutwert von x



Bitweise logische Operationen

Integer-Datentypen doppelt genutzt:

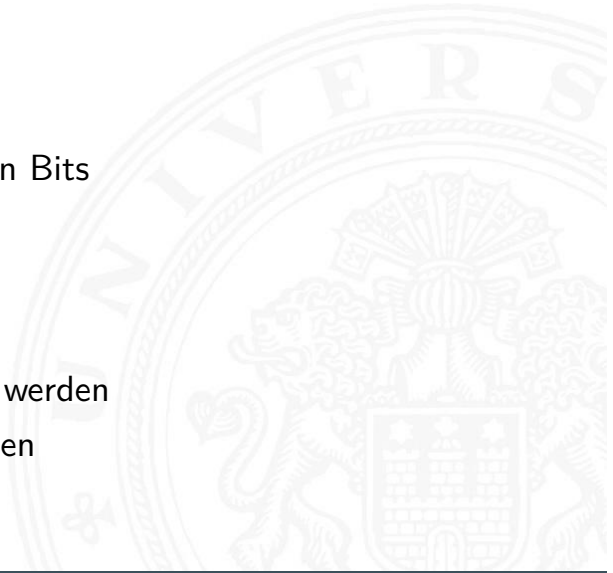
1. Zahlenwerte (Ganzzahl, Zweierkomplement, Gleitkomma)
arithmetische Operationen: Addition, Subtraktion usw.
2. Binärwerte mit w einzelnen Bits (Wortbreite w)
Boole'sche Verknüpfungen, bitweise auf allen w Bits
 - ▶ Grundoperationen: Negation, UND, ODER, XOR
 - ▶ Schiebe-Operationen: shift-left, rotate-right usw.



Bitweise logische Operationen (cont.)

- ▶ Integer-Datentypen interpretiert als Menge von Bits
- ⇒ bitweise logische Operationen möglich
- ▶ in Java und C sind vier Operationen definiert:

Negation	$\sim x$	Invertieren aller einzelnen Bits
UND	$x \& y$	Logisches UND aller einzelnen Bits
OR	$x y$	–"– ODER –"–
XOR	$x \wedge y$	–"– XOR –"–
- ▶ alle anderen Funktionen können damit dargestellt werden
es gibt insgesamt 2^{2^n} Operationen mit n Operanden





Bitweise logische Operationen: Beispiel

$x = 0010\ 1110$

$y = 1011\ 0011$

$\sim x = 1101\ 0001$ alle Bits invertiert

$\sim y = 0100\ 1100$ alle Bits invertiert

$x \& y = 0010\ 0010$ bitweises UND

$x | y = 1011\ 1111$ bitweises ODER

$x \wedge y = 1001\ 1101$ bitweises XOR

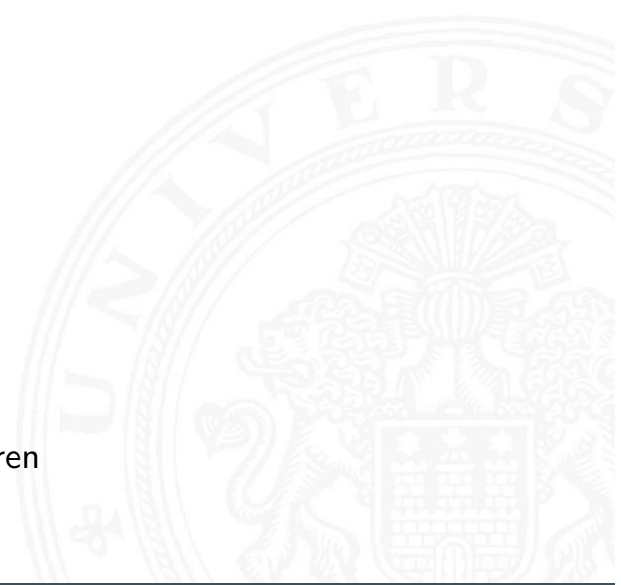


Schiebeoperationen

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

- ▶ fünf Varianten
 - Shift-Left shl
 - Logical Shift-Right sr1
 - Arithmetic Shift-Right sra
 - Rotate-Left rol
 - Rotate-Right ror

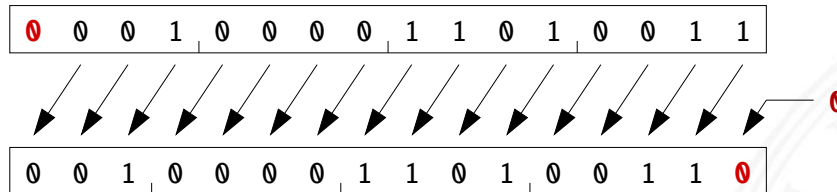
- ▶ Schiebeoperationen in Hardware leicht zu realisieren
- ▶ auf fast allen Prozessoren im Befehlssatz





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

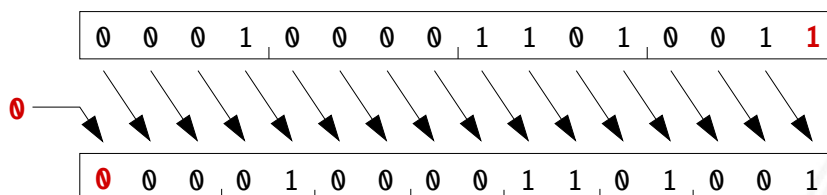


- ▶ in Java und C direkt als Operator verfügbar: $x \ll n$
- ▶ `shl` um n bits entspricht der Multiplikation mit 2^n



Logical Shift-Right (shr)

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

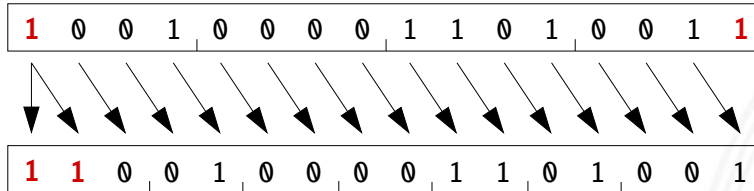


- ▶ in Java direkt als Operator verfügbar: $x \ggg n$
- ▶ in C nur für `unsigned`-Typen definiert: $x \gg n$
- ▶ für `signed`-Typen nicht vorhanden



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)

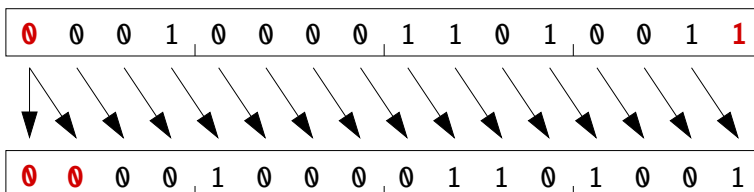


- ▶ in Java direkt als Operator verfügbar: $x \gg n$
- ▶ in C nur für signed-Typen definiert: $x \gg n$
- ▶ sra um n bits ist ähnlich der Division durch 2^n

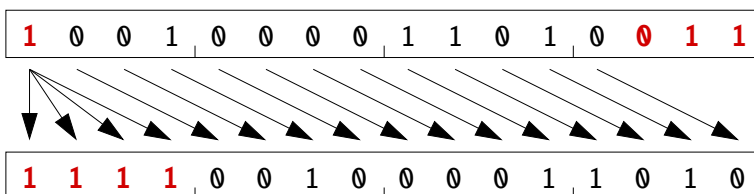


Arithmetic Shift-Right: Beispiel

- ▶ $x \gg 1$ aus $0x10D3$ (4307) wird $0x0869$ (2153)



- ▶ $x \gg 3$ aus $0x90D3$ (-28460) wird $0xF21A$ (-3558)





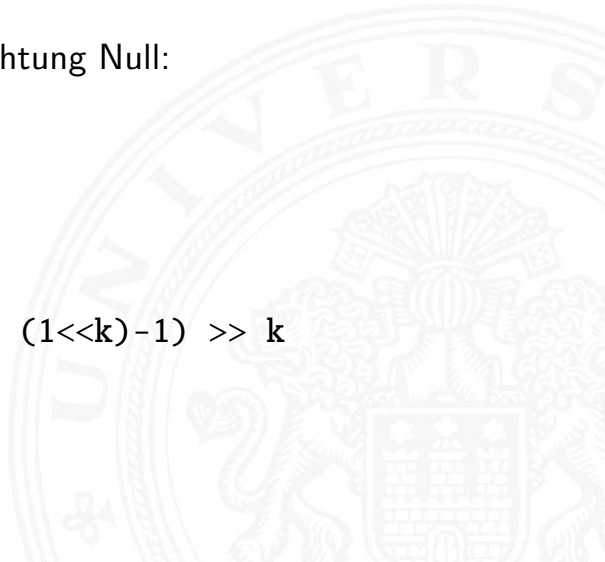
Arithmetic Shift-Right: Division durch Zweierpotenzen?

- ▶ positive Werte: $x \gg n$ entspricht Division durch 2^n
- ▶ negative Werte: $x \gg n$ ähnlich Division durch 2^n , aber Ergebnis ist zu klein!

- ▶ gerundet in Richtung negativer Werte statt in Richtung Null:

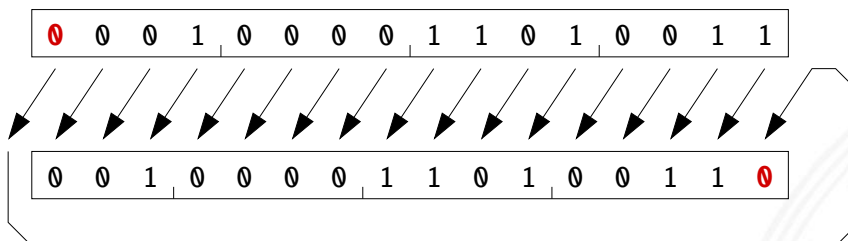
1111 1011 (-5)
 1111 1101 (-3)
 1111 1110 (-2)
 1111 1111 (-1)

- ▶ in C: Kompensation durch Berechnung von $(x + (1 \ll k) - 1) \gg k$
 Details: Bryant, O'Hallaron [BO15]

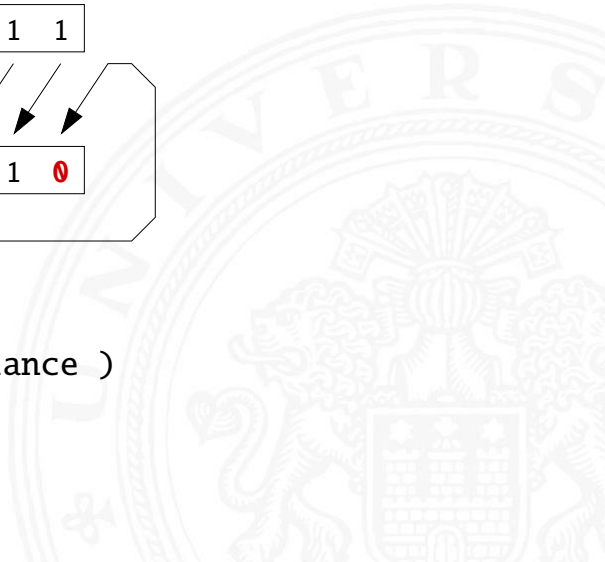


Rotate-Left (rol)

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



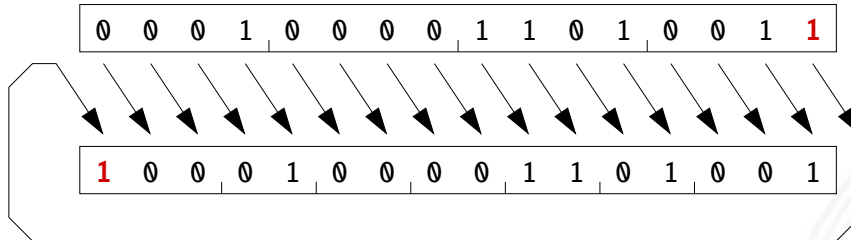
- ▶ in Java und C nicht als Operator verfügbar
- ▶ Java: `Integer.rotateLeft(int x, int distance)`





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
 - ▶ Addition/Subtraktion und logische Operationen: typisch 1 Takt
 - ▶ Shift-Operationen: meistens 1 Takt
- ⇒ Trick: Multiplikation mit Konstanten ersetzen durch Kombination aus shifts+add
- ▶ Beispiel: $9 \cdot x = (8 + 1) \cdot x$ ersetzt durch $(x \ll 3) + x$
 - ▶ 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 $(1 \ll p)$, 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 $0 \dots 255$
- ▶ 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) & 033333333333)
           - ((u >> 2) & 011111111111);
  return ((uCount + (uCount >> 3)) & 030707070707) % 63;
}
```

- ▶ `java.lang.Integer.bitCount()`



Beispiel: Bitcount – parallel, tree (cont.)

```
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;
}
```

- ▶ viele Algorithmen: bit-Maskierung und Schieben
 - ▶ gurmeet.net/puzzles/fast-bit-counting-routines
 - ▶ graphics.stanford.edu/~seander/bithacks.html
 - ▶ 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]

6.5 Logische Operationen - Anwendungsbeispiele

64-040 Rechnerstrukturen und Betriebssysteme

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 $a, b \geq 0$

▶ Ausnahmen: $x = -2^b$ und $x = 0$

$$\Rightarrow x = (\alpha 0 [1]^a 1 [0]^b)_2$$

$$\bar{x} = (\bar{\alpha} 1 [0]^a 0 [1]^b)_2$$

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

$$-x = (\bar{\alpha} 1 [0]^a 1 [0]^b)_2$$

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



Literatur

6.6 Logische Operationen - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

[BO15] R.E. Bryant, D.R. O'Hallaron:

Computer systems – A programmers perspective.

3rd global ed., Pearson Education Ltd., 2015.

csapp.cs.cmu.edu

ISBN 978-1-292-10176-7

[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



- [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
www-cs-faculty.stanford.edu/~knuth/taocp.html
- [Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript.* Universität Hamburg, FB Informatik, 2005, Vorlesungsskript.
tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1



1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. Logische Operationen
7. **Codierung**
 - Grundbegriffe
 - Ad-Hoc Codierungen
 - Einschrittige Codes
 - Quellencodierung
 - Symbolhäufigkeiten
 - Informationstheorie



Gliederung (cont.)

7 Codierung

64-040 Rechnerstrukturen und Betriebssysteme

Entropie
Kanalcodierung
Fehlererkennende Codes
Zyklische Codes
Praxisbeispiele
Literatur

8. Schaltfunktionen

9. Schaltnetze

10. Schaltwerke

11. Rechnerarchitektur I

12. Instruction Set Architecture

13. Assembler-Programmierung

14. Rechnerarchitektur II



A. Mäder

330



Gliederung (cont.)

7 Codierung

64-040 Rechnerstrukturen und Betriebssysteme

15. Betriebssysteme



A. Mäder

331



Definition: Codierung

7.1 Codierung - Grundbegriffe

64-040 Rechnerstrukturen und Betriebssysteme

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

7.1 Codierung - Grundbegriffe

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ **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
- ▶ Beispiel: fehlerkorrigierende Codes sind bei der Nachrichtenübertragung essenziell, im Rechner wegen der hohen Zuverlässigkeit weniger wichtig



Darstellung von Codes

7.1 Codierung - Grundbegriffe

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ **Wertetabellen**
 - ▶ jede Zeile enthält das Urbild (zu codierendes 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 ergeben das Codewort
- ▶ **Logische Gleichungen**
- ▶ **Algebraische Ausdrücke**

A. Mäder

336



Codierung von Text

7.2 Codierung - Ad-Hoc Codierungen

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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)

A. Mäder

337



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 eingesetzt



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 \bar{c}
- ▶ **einschrittig:** jeweils aufeinanderfolgende Codewörter unterscheiden sich immer nur an einer Stelle
- ▶ **zyklisch:** bei n geordneten Codewörtern ist $c_0 = c_n$
- ▶ **zyklisch einschrittig**



Dualcode

- ▶ der Name für Codierung der Integerzahlen im Stellenwertsystem
- ▶ Codewort

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

- ▶ 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



- ▶ möglich für Mengen mit Ordnungsrelation
- ▶ Elemente der Menge werden durch Binärwörter codiert
- ▶ **einschrittiger Code**: der Ordnung folgend, unterscheiden sich benachbarte Codewörter der Menge 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])



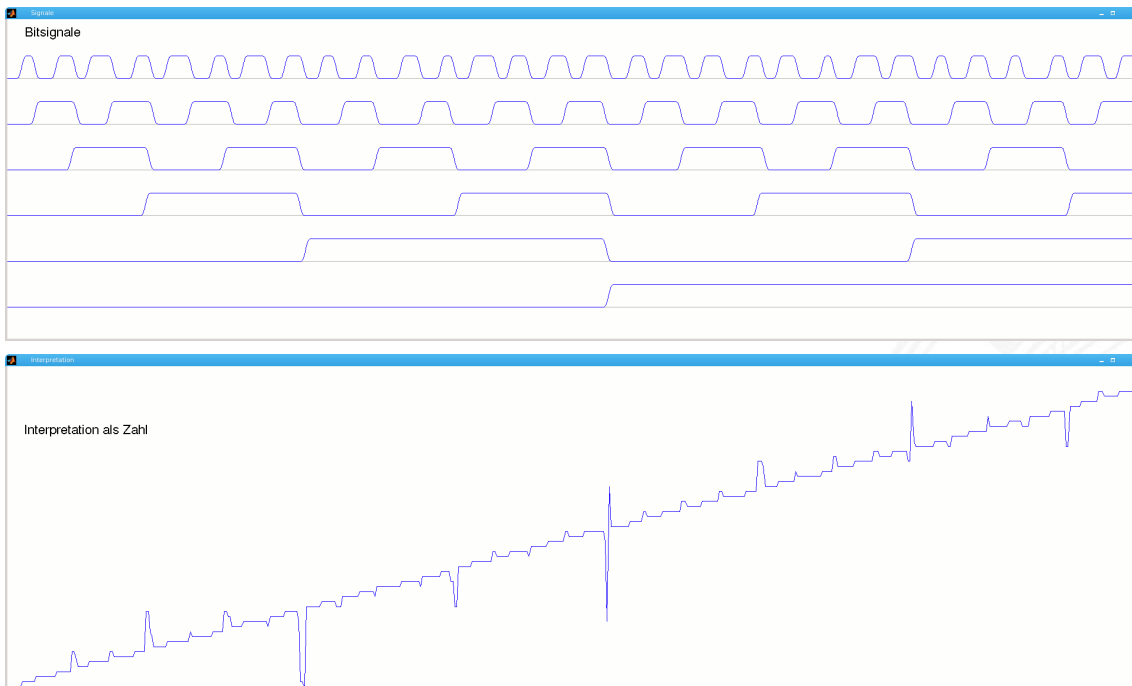
- ▶ Ablesen eines Wertes mit leicht gegeneinander verschobenen Übergängen der Bits
z.B.: [Hei05], Kapitel 1.4
 - ▶ `demoeinschritt(0:59)` normaler Dualcode
 - ▶ `demoeinschritt(einschritt(60))` einschrittiger Code

- ▶ maximaler Ablesefehler
 - ▶ 2^{n-1} beim Dualcode z.B.: 0111 → 1111 → 1000 statt 0111 → 1000
 - ▶ 1 beim einschrittigen Code

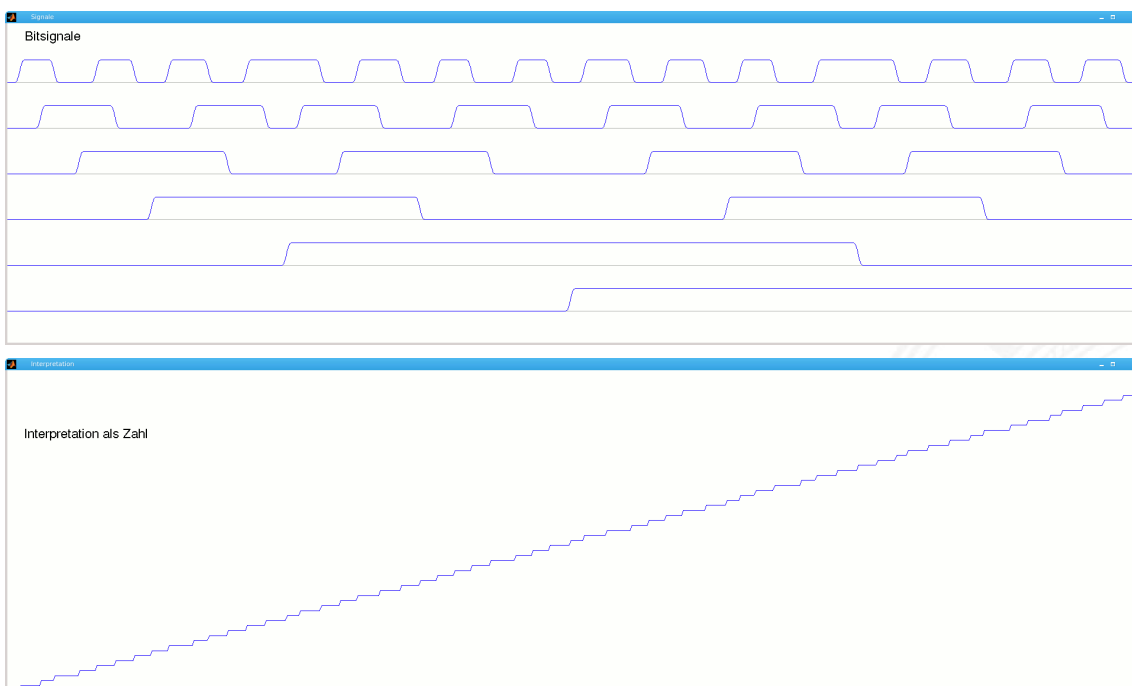
- ▶ Konstruktion eines einschrittigen Codes
 - ▶ rekursiv
 - ▶ als ununterbrochenen Pfad im KV-Diagramm (s.u.)



Ablezen des Wertes aus Dualcode

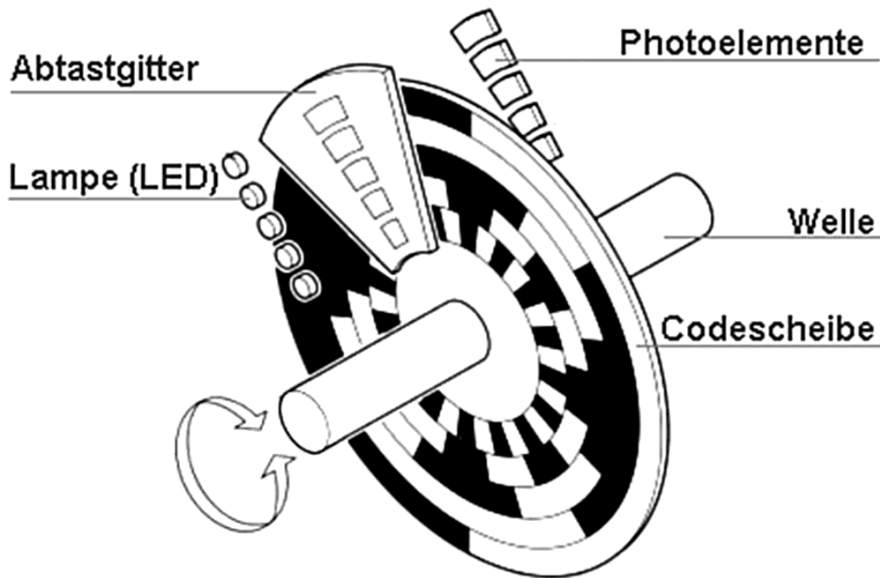


Ablezen des Wertes aus einschrittigem Code

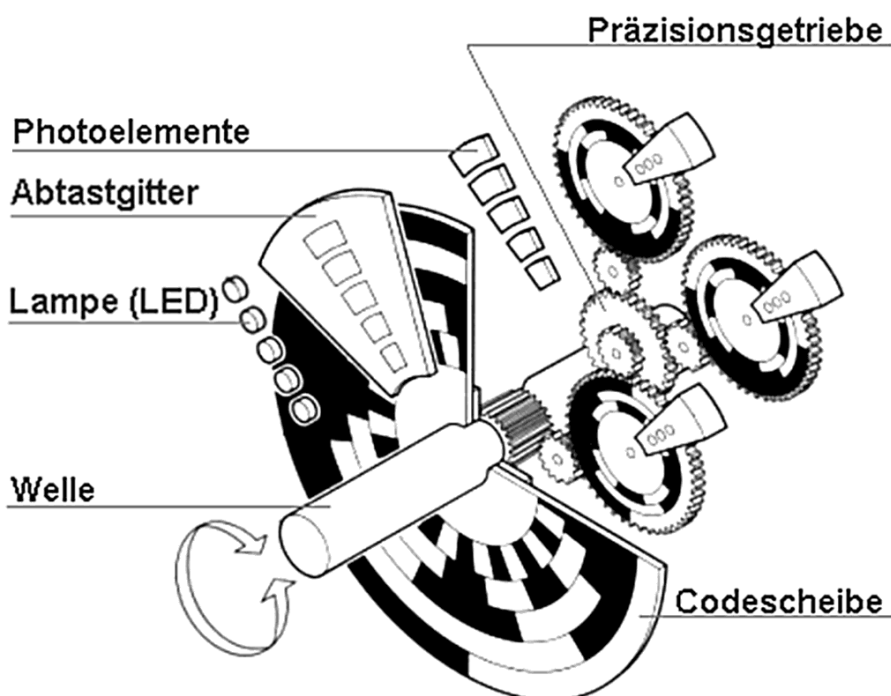




Gray-Code: Prinzip eines Winkeldrehgebers

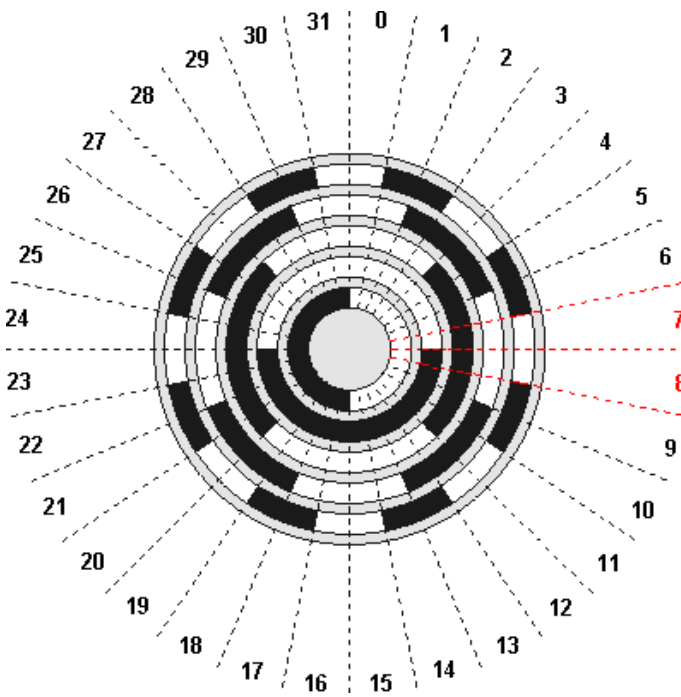


Gray-Code: mehrstufiger Drehgeber





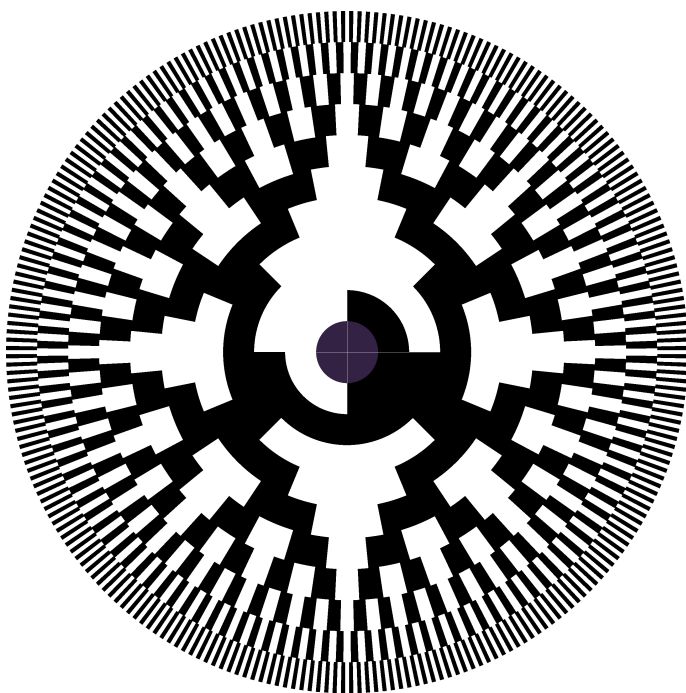
Gray-Code: 5-bit Codierscheibe



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. schreibe eine führende 0 vor alle vorhandenen n Codewörter
 2. schreibe eine führende 1 vor die in umgekehrter Reihenfolge notierten Codewörter

$\{ 0, 1 \}$
 $\{ 00, 01, 11, 10 \}$
 $\{ 000, 001, 011, 010, 110, 111, 101, 100 \}$
 ...

⇒ Gray-Code



Karnaugh-Veitch Diagramm

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

	$x_1 x_0$			
	00	01	11	10
$x_3 x_2$				
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
- ⇒ benachbarte Felder unterscheiden sich gerade um 1 Bit



Einschrittiger Code: KV-Diagramm

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

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

- ▶ 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.)

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

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

- ▶ 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
- ⇒ KV-Diagramm als „außen zusammengeklebt“ denken
- ⇒ Pfade können auch „außen herum“ geführt werden



Gray-Code: Umwandlung in/von Dualcode

7.3 Codierung - Einschrittige Codes

64-040 Rechnerstrukturen und Betriebssysteme

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.
 - ▶ $\text{gray}(x) = x \oplus (x \gg 1)$
 - ▶ in Hardware einfach durch paarweise XOR-Operationen
[HenHA] Hades Demo: 10-gates/15-graycode/dual2gray



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

7.3 Codierung - Einschrittige Codes

64-040 Rechnerstrukturen und Betriebssysteme

Umwandlung: Gray- in Dualwort

1. MSB wird übernommen
 2. Stellenweise 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

7.4 Codierung - Quellencodierung

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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 direkt durch Abzählen möglich
- ⇒ Einhalten der **Fano-Bedingung** notwendig
oder Einführen von **Markern** zwischen den Codewörtern

A. Mäder

356



Fano-Bedingung

7.4 Codierung - Quellencodierung

64-040 Rechnerstrukturen und Betriebssysteme

Eindeutige Decodierung eines Codes mit variabler Wortlänge?

Fano-Bedingung

Kein Wort aus einem Code bildet den Anfang eines anderen Codeworts

- ▶ die sogenannte **Präfix-Eigenschaft** (eigentlich: präfix-frei)
- ▶ nach R. M. Fano (1961)
- ▶ ein **Präfixcode** ist eindeutig decodierbar
- ▶ Blockcodes sind Präfixcodes

A. Mäder

357



- ▶ 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



Codetabelle

	•kurzer Ton	–langer Ton
A •–	S •••	. •–•–•–
B –•••	T –	, –•••–
C –•–•	U ••–	? ••–•••
D –••	V •••–	’ •–•–••
E •	W •–	! –•–•–
F ••–•	X –••–	/ –••–•
G –••	Y –•–•	(–•–••
H ••••	Z –•••) –•–•–
I ••	0 –•–•–	& •–•••
J •–•–	1 •–•–•	: –•–•••
K –•–	2 ••–•–	; –•–•••
L •–••	3 •••–	= –•••–
M –•–	4 ••••–	+ •–•••
N –•	5 •••••	- –••••–
O –•–•	6 –••••	_ ••–••–
P •–••	7 –••••	" •–••••
Q –•–•	8 –•–••	\$ •••••–
R •–•	9 –•–••	@ •–••••
		S-Start –•–•–
		Verst. •••••
		S-Ende •–•••
		V-Ende ••••–
		Error ••••••••
		Ä •–•–
		À •–•–
		É ••–••
		È •–••–
		Ö –•–•
		Ü ••–
		ß •••••••
		CH –•–•–
		Ñ –•–•–
		...
		SOS ••• –•–•••



Morse-Code (cont.)

► Eindeutigkeit Codewort: ● ● ● ● ● - ●

E	●
I	● ●
N	- ●
R	● - ●
S	● ● ●

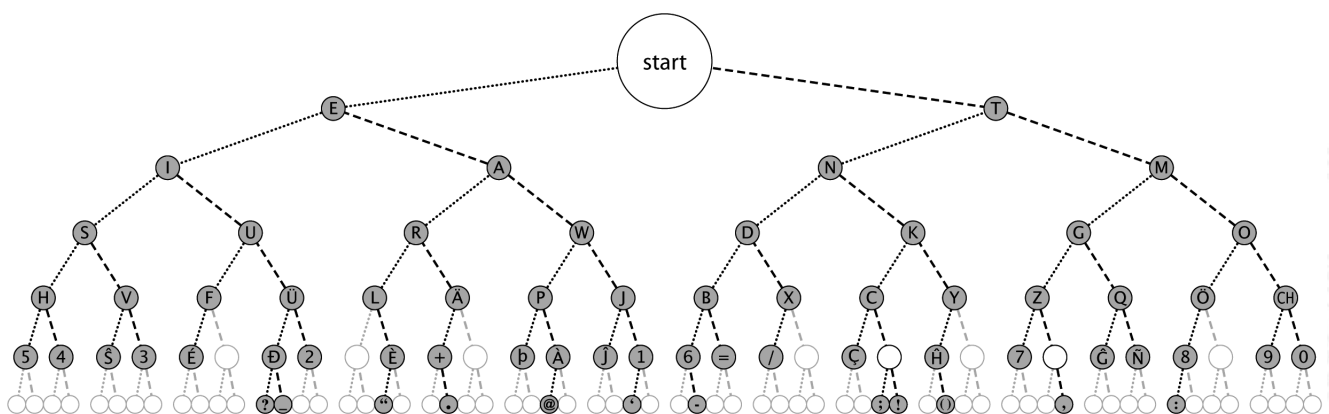
- bestimmte Morse-Sequenzen sind mehrdeutig
- Pause zwischen den Symbolen notwendig

► 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

7.4 Codierung - Quellencodierung

64-040 Rechnerstrukturen und Betriebssysteme

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

A. Mäder

362



Codierung nach Fano (Shannon-Fano Codierung)

7.5 Codierung - Symbolhäufigkeiten

64-040 Rechnerstrukturen und Betriebssysteme

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

- ▶ Ordnung der Urwörter anhand ihrer Wahrscheinlichkeiten
 $p(a_1) \geq p(a_2) \geq \dots \geq p(a_n)$
- ▶ Einteilung der geordneten Urwörter in zwei Gruppen mit möglichst gleicher Gesamtwahrscheinlichkeit: $a_1 \dots a_i$ und $a_{i+1} \dots a_n$. 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

A. Mäder

363



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 $\{B\}$
- ⇒ 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

7.5 Codierung - Symbolhäufigkeiten

64-040 Rechnerstrukturen und Betriebssysteme

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

- ▶ Ordnung der Urwörter anhand ihrer Wahrscheinlichkeiten
 $p(a_1) \leq p(a_2) \leq \dots \leq p(a_n)$
- ▶ 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 ...)

A. Mäder

366



Codierung nach Huffman: Beispiel

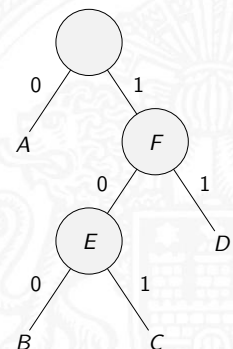
7.5 Codierung - Symbolhäufigkeiten

64-040 Rechnerstrukturen und Betriebssysteme

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 mit Wahrscheinlichkeit
 $p(E) = 0,1 + 0,15 = 0,25$ Liste: $\{E, D, A\}$
2. Zusammenfassen von E und D als neues Wort F mit Wahrscheinlichkeit
 $p(F) = 0,55$ Liste: $\{A, F\}$
3. Zuordnung der Bits entsprechend der Wahrscheinlichkeiten
 - ▶ $A = 0$ und $F = 1$
 - ▶ Split von F in $E = 10$ und $D = 11$
 - ▶ Split von E in $B = 100$ und $C = 101$

⇒ Codewörter sind $A = 0$, $D = 11$, $C = 101$ und $B = 100$



A. Mäder

367



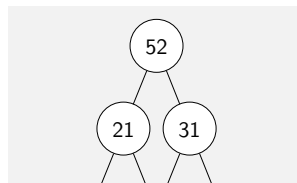
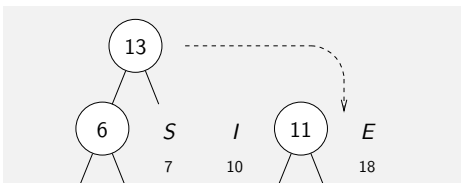
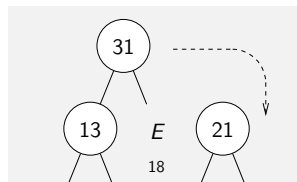
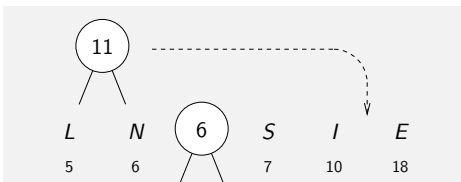
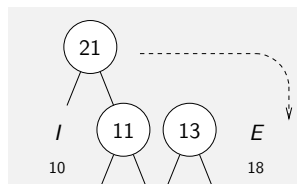
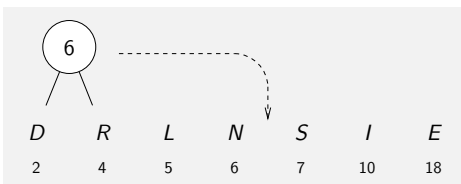
Bildung eines Huffman-Baums

- ▶ 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



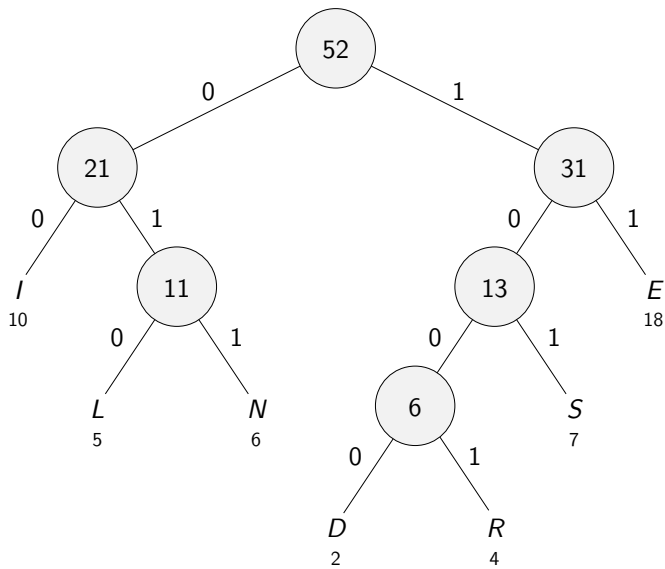
Bildung eines Huffman-Baums (cont.)

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

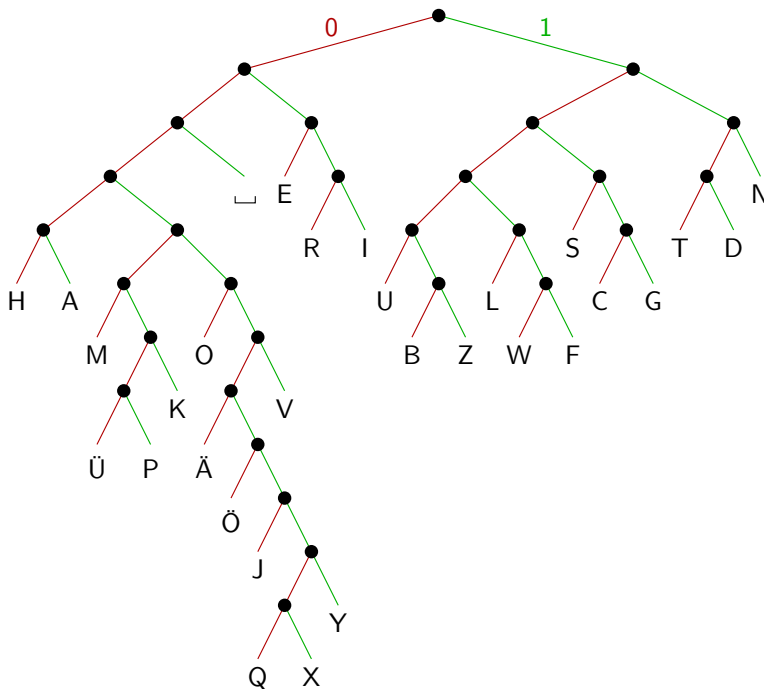
Zeichen	Code	Zeichen	Code
Leerzeichen	001	O	000110
E	010	B	100010
N	111	Z	100011
R	0110	W	100110
I	0111	F	100111
S	1010	K	0001011
T	1100	V	0001111
D	1101	Ü	00010100
H	00000	P	00010101
A	00001	Ä	00011100
U	10000	Ö	000111010
L	10010	J	0001110110
C	10110	Y	00011101111
G	10111	Q	000111011100
M	000100	X	000111011101



Codierung nach Huffman: Codebaum

7.5 Codierung - Symbolhäufigkeiten

64-040 Rechnerstrukturen und Betriebssysteme



ca. 4,5 Bits/Zeichen,
1,7-Mal besser als ASCII



Codierung nach Huffman: Minimale Codelänge

7.5 Codierung - Symbolhäufigkeiten

64-040 Rechnerstrukturen und Betriebssysteme

Beweis der Minimalität

- ▶ Sei C ein Huffman-Code mit durchschnittlicher Codelänge L
- ▶ Sei D ein weiterer Präfixcode 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 $p_{s-1} + p_s$
- ▶ der letzte Schritt wird wiederholt: dadurch wird C auf einen Baum mit durchschnittlicher Länge 1 reduziert und D auf einen Baum mit Länge < 1
Dies ist aber nicht möglich \square



Codierung nach Huffman: Symbole mit $p \geq 0,5$

7.5 Codierung - Symbolhäufigkeiten

64-040 Rechnerstrukturen und Betriebssysteme

Was passiert, wenn ein Symbol eine Häufigkeit $p \geq 0,5$ aufweist?

- ▶ die Huffman-Codierung müsste weniger als ein Bit zuordnen, was nicht geht
- ⇒ Huffman- (und Fano-) Codierungen sind in diesem Fall ineffizient
- ▶ Beispiel: Bild mit einheitlicher Hintergrundfarbe codieren
- ▶ andere Ideen notwendig
 - ▶ Lauflängencodierung (Fax, GIF, PNG)
 - ▶ Cosinustransformation (JPEG) usw.



Dynamic Huffman Coding

7.5 Codierung - Symbolhäufigkeiten

64-040 Rechnerstrukturen und Betriebssysteme

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, dabei wird der Codebaum mit jedem decodierten Zeichen angepasst
- ▶ Symbolhäufigkeiten werden nicht explizit übertragen

D. E. Knuth: *Dynamic Huffman Coding*, 1985 [Knu85]



Kraft-Ungleichung

- ▶ Leon G. Kraft, 1949 de.wikipedia.org/wiki/Kraft-Ungleichung
- ▶ Eine notwendige und hinreichende Bedingung für die Existenz eines eindeutig decodierbaren s -elementigen Codes C mit Codelängen $l_1 \leq l_2 \leq l_3 \leq \dots \leq l_s$ 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

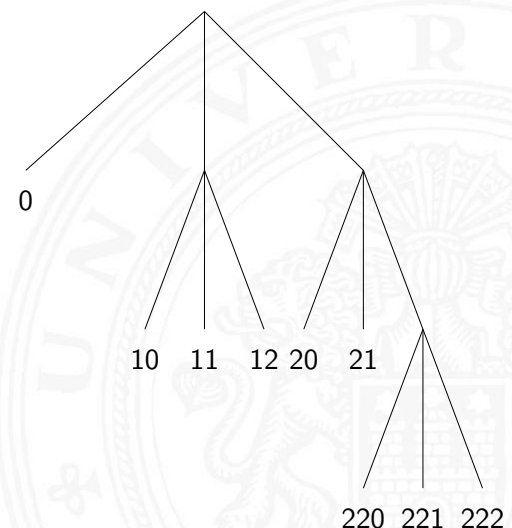
- ▶ Sei $F = \{0, 1, 2\}$ (ternäres Alphabet)
Seien die geforderten Längen der Codewörter: 1,2,2,2,2,2,3,3,3

▶ Berechnung: $\frac{1}{3} + 5 \cdot \frac{1}{3^2} + 3 \cdot \frac{1}{3^3} = 1$

⇒ ein passender Präfixcode existiert!

- ▶ Konstruktion entsprechend des Beweises (s.u.)

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

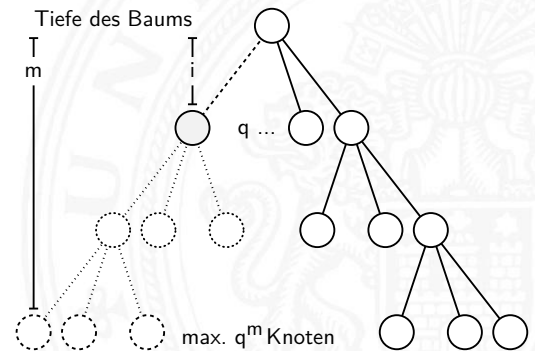
► Wir schreiben

$$\sum_{i=1}^s \frac{1}{q^i} = \sum_{j=1}^m \frac{u_j}{q^j} = \frac{1}{q^m} \sum_{j=1}^m u_j \cdot q^{m-j} \leq 1$$

$$(*) \quad u_m + \sum_{j=1}^{m-1} u_j \cdot q^{m-j} \leq q^m$$

- 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

⇒ 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 $W\{A_i\} = p_i$
- im Kontext von 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

⇒ Informationsgewinn („Überraschung“) wird größer, je kleiner p_i



Geeignetes Maß für die Information?

7.6 Codierung - Informationstheorie

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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: 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: Logarithmus von $1/p$ bilden

A. Mäder

380



Erinnerung: Logarithmus

7.6 Codierung - Informationstheorie

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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 $\log_b(b^x) = x$
 - ▶ $\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$
 - ▶ $\log_2(x) = \ln(x)/\ln(2) = \ln(x)/0,69314718$

A. Mäder

381



Definition: Informationsgehalt

Informationsgehalt eines Ereignisses A_i mit Wahrscheinlichkeit p_i ?

- ▶ als messbare und daher additive Größe
- ▶ durch Logarithmierung (Basis 2) der Wahrscheinlichkeit:

$$I(A_i) = \log_2\left(\frac{1}{p_i}\right) = -\log_2(p_i)$$

- ▶ **Informationsgehalt** I (oder Information) von A_i
auch **Entscheidungsgehalt** genannt
- ▶ Beispiel: zwei Nachrichten A und B

$$I(A) + I(B) = \log_2\left(\frac{1}{p_A \cdot p_B}\right) = \log_2\left(\frac{1}{p_A}\right) + \log_2\left(\frac{1}{p_B}\right)$$



Einheit Bit

$$I(A_i) = \log_2\left(\frac{1}{p_i}\right) = -\log_2(p_i)$$

- ▶ 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

7.6 Codierung - Informationstheorie

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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

7.6 Codierung - Informationstheorie

64-040 Rechnerstrukturen und Betriebssysteme

Meteorit

- ▶ die Wahrscheinlichkeit, von einem Meteor getroffen zu werden, sei $p_M = 10^{-16}$
- ▶ Kein Grund zur Sorge, weil die Ungewissheit von sehr klein ist
 $I = \log_2(1/(1 - p_M)) \approx 3,2 \cdot 10^{-16}$
Ebenso klein ist die Überraschung, wenn das Unglück nicht passiert
- ⇒ Informationsgehalt der Nachricht „Ich wurde nicht vom Meteor erschlagen“
ist sehr klein
- ⇒ Umgekehrt wäre die Überraschung groß: $\log_2(1/p_M) = 53,15$



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 $\frac{1}{6}$
- ▶ $I(6) = \log_2(1/\frac{1}{6}) = 2,585$



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 Begriff“, so ist „f“ als nächstes Symbol sehr wahrscheinlich
 2. beim Zufallstext liefert jedes neue Symbol die zusätzliche Information $I = \log_2(1/\frac{1}{80})$
- ⇒ der Zufallstext enthält die größtmögliche Information



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



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 $\sum_i p_i = 1$



- ▶ Die Entropie H als Erwartungswert berechnet sich als Funktion der Symbol-Wahrscheinlichkeiten nur abhängig vom stochastischen Modell

$$\begin{aligned} H &= E\{I(A_i)\} \\ &= \sum_i p_i \cdot I(A_i) \\ &= \sum_i p_i \cdot \log_2\left(\frac{1}{p_i}\right) \\ &= - \sum_i p_i \cdot \log_2(p_i) \end{aligned}$$



1. drei mögliche Ereignisse mit Wahrscheinlichkeiten $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\}$

- ▶ dann berechnet sich die Entropie zu

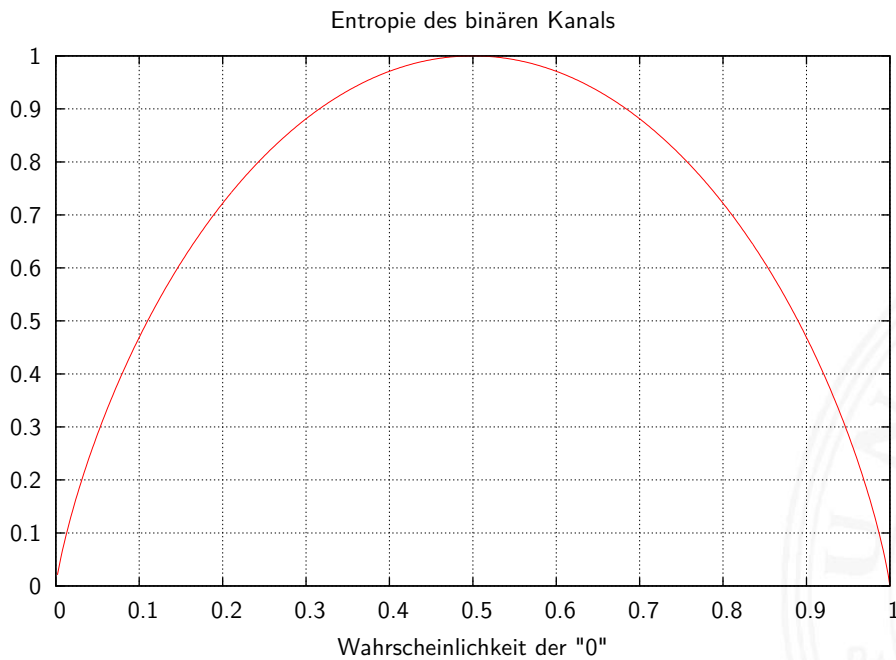
$$H = -\left(\frac{1}{2} \log_2\left(\frac{1}{2}\right) + \frac{1}{3} \log_2\left(\frac{1}{3}\right) + \frac{1}{6} \log_2\left(\frac{1}{6}\right)\right) = 1,4591$$

2. Empfang einer Binärstelle mit den Wahrscheinlichkeiten $p_0 = q$ und $p_1 = (1 - q)$.

- ▶ für $q = \frac{1}{2}$ erhält man

$$H = -\left(\frac{1}{2} \log_2\left(\frac{1}{2}\right) + \left(1 - \frac{1}{2}\right) \log_2\left(1 - \frac{1}{2}\right)\right) = 1,0$$

- ▶ mittlerer Informationsgehalt beim Empfang einer Binärstelle mit gleicher Wahrscheinlichkeit für beide Symbole ist genau 1 Bit



Entropie bei Empfang einer Binärstelle mit den Wahrscheinlichkeiten $p_0 = q$ und $p_1 = (1 - q)$



- ▶ 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

7.7 Codierung - Entropie

64-040 Rechnerstrukturen und Betriebssysteme

1. $H(p_1, p_2, \dots, p_n)$ ist maximal, falls $p_i = 1/n$ ($1 \leq i \leq n$)
2. H ist symmetrisch, für jede Permutation π von $1, 2, \dots, n$ gilt:
 $H(p_1, p_2, \dots, p_n) = H(p_{\pi(1)}, p_{\pi(2)}, \dots, p_{\pi(n)})$
3. $H(p_1, p_2, \dots, p_n) \geq 0$ mit $H(0, 0 \dots 0, 1, 0 \dots 0, 0) = 0$
4. $H(p_1, p_2, \dots, p_n, 0) = H(p_1, p_2, \dots, p_n)$
5. $H(1/n, 1/n, \dots, 1/n) \leq H(1/(n+1), 1/(n+1), \dots, 1/(n+1))$
6. H ist stetig in seinen Argumenten
7. Additivität: seien $n, m \in \mathbb{N}^+$
 $H(\frac{1}{n \cdot m}, \frac{1}{n \cdot m}, \dots, \frac{1}{n \cdot m}) = H(\frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n}) + H(\frac{1}{m}, \frac{1}{m}, \dots, \frac{1}{m})$

A. Mäder

394



Möglicher Informationsgehalt

7.7 Codierung - Entropie

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ **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: $H_0 = N$

A. Mäder

395



- ▶ **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 $p(a_i)$:

$$\begin{aligned} R &= H_0 - H = H_0 - \left(- \sum_{i=1}^m p(a_i) \cdot \log_2(p(a_i)) \right) \\ &= N + \sum_{i=1}^m p(a_i) \cdot \log_2(p(a_i)) \end{aligned}$$



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

$$C = 1 - H(F)$$

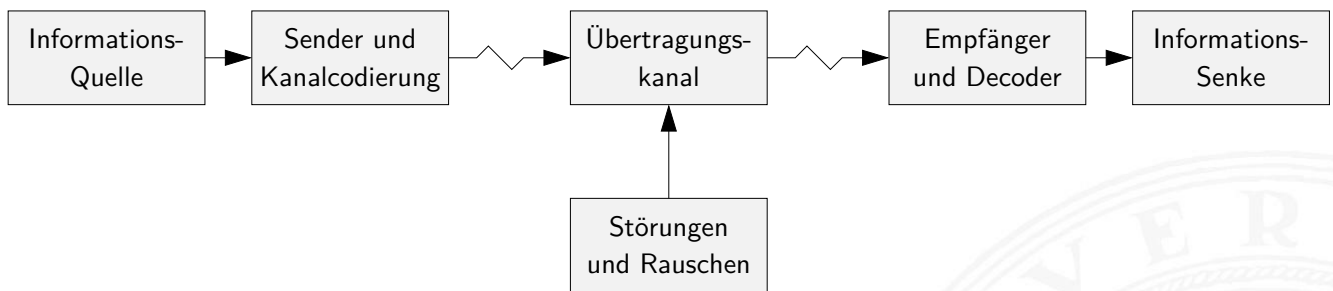
mit $H(F)$ der Entropie des Fehlerverhaltens



Erinnerung: Modell der Informationsübertragung

7.8 Codierung - Kanalcodierung

64-040 Rechnerstrukturen und Betriebssysteme



- ▶ Informationsquelle
- ▶ Sender mit möglichst effizienter Kanalcodierung
- ▶ gestörter und verrauschter Übertragungskanal
- ▶ Empfänger mit Decodierer und Fehlererkennung/-korrektur
- ▶ Informationssenke und -verarbeitung



Binärer symmetrischer Kanal

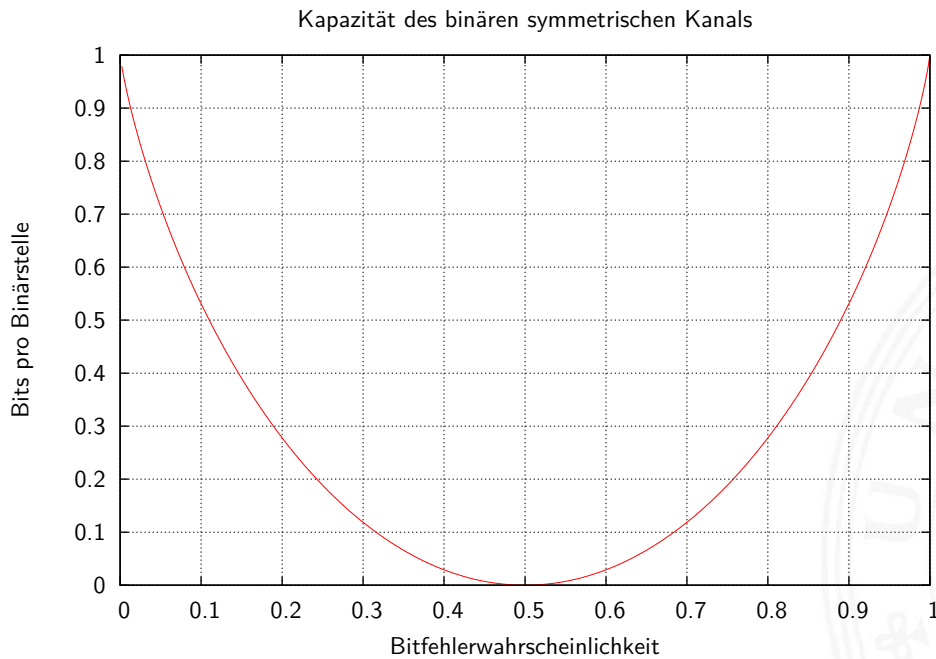
7.8 Codierung - Kanalcodierung

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Wahrscheinlichkeit der beiden Symbole 0 und 1 ist gleich $\left(\frac{1}{2}\right)$
- ▶ 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
- ▶ Fehlerverhalten binär: Fehler tritt entweder auf oder nicht
⇒ Entropie des Fehlerverhaltens

$$H(F) = P \cdot \log_2(1/P) + (1 - P) \cdot \log_2(1/(1 - P))$$

- ▶ Kanalkapazität ist $C = 1 - H(F)$



- ▶ bei $P = 0,5$ ist die Kanalkapazität $C = 0$
- ⇒ 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

7.8 Codierung - Kanalcodierung

64-040 Rechnerstrukturen und Betriebssysteme

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

7.8 Codierung - Kanalcodierung

64-040 Rechnerstrukturen und Betriebssysteme

⇒ Wenn die Übertragungsrate kleiner als die Kanalkapazität ist, existieren Codes, die beliebig zuverlässig sind

... 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.



Motivation

- ▶ Informationstheorie
- ▶ Kanalkapazität
- ▶ Shannon-Theorem

- ▶ zuverlässige Datenübertragung ist möglich
- ▶ aber (bisher) keine Ideen für die Realisierung

- ⇒ fehlererkennende Codes
- ⇒ fehlerkorrigierende Codes



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



- ▶ **Blockcode**
 - ▶ identische Wortlänge aller Codewörter
 - ▶ k -Informationsbits werden in n -Bits codiert
 - ▶ Blöcke hängen nicht voneinander ab
- ▶ **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



spezielle Blockcodes

- ▶ **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
- ▶ **systematischer Code**
 - ▶ die zu codierende Information ist direkt (als Substring) im Codewort enthalten
 - ▶ z.B.: durch Anhängen von Paritätsbits
- ▶ **zyklischer Code**
 - ▶ für jedes Codewort gilt: zyklische Verschiebungen (Rotationen) sind auch Codeworte
 - ⇒ bei serieller Übertragung erlaubt dies die Erkennung/Korrektur von Bündelfehlern



Einschub: $GF(2)$, $GF(2)^n$

de.wikipedia.org/wiki/Endlicher_Körper [en.wikipedia.org/wiki/GF\(2\)](https://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: $x \oplus x = 0$



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 Sender codiert mit Redundanz
 - ▶ der Empfänger erkennt fehlerhafte Codewörter und kann diese 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 $a = 0110\ 0011$
 $b = 1010\ 0111$
- ⇒ Hamming-Abstand von a und b ist 3
Hamming-Gewicht von b ist 5
- ▶ Java: `Integer.bitcount(a ^ b)`



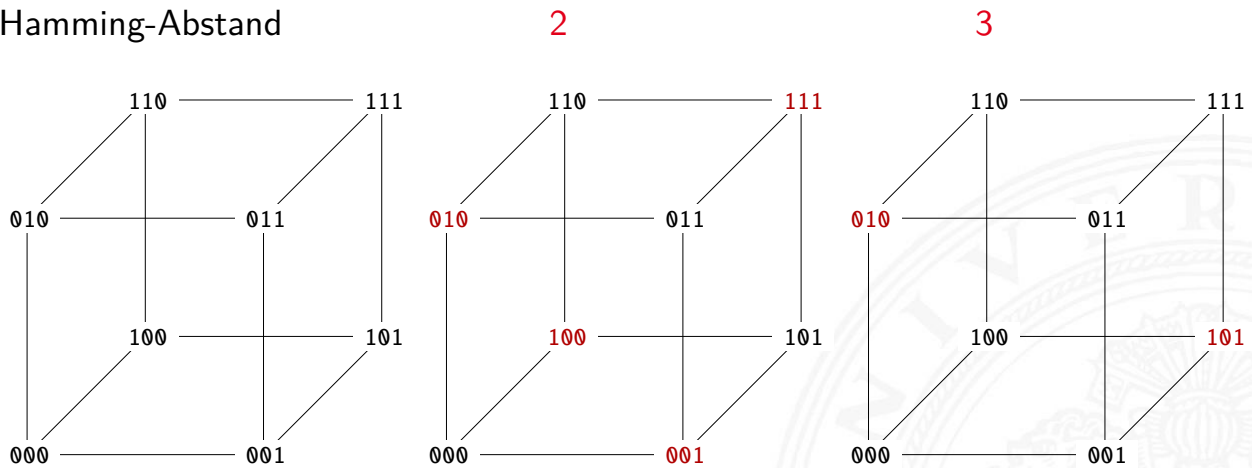
Fehlererkennende und -korrigierende Codes

- ▶ Zur *Fehlererkennung* und *Fehlerkorrektur* ist eine Codierung mit zusätzlicher 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
- ⇒ Fehlererkennung bis zu $(d - 1)$ fehlerhaften Stellen
Fehlerkorrektur bis zu $((d - 1)/2)$ —

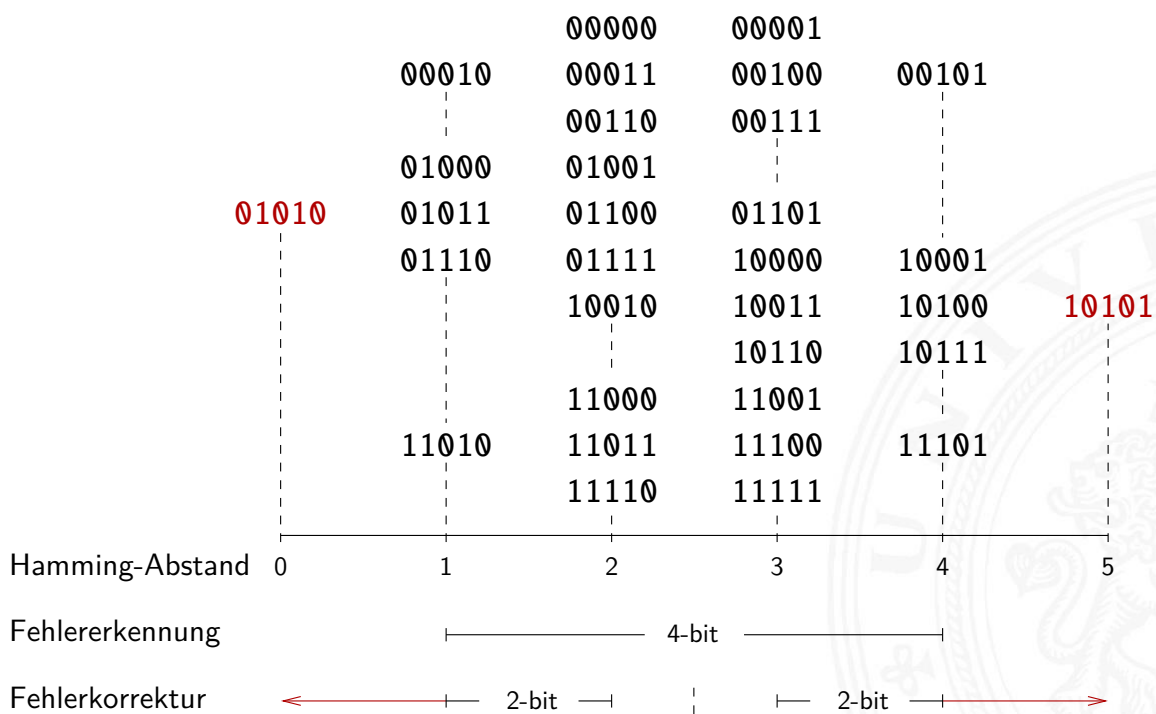


Fehlererkennende und -korrigierende Codes (cont.)

► Hamming-Abstand



Fehlererkennende und -korrigierende Codes (cont.)





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-Chaudhuri, Hocquengham)
- ▶ RS-Codes (Reed-Solomon)



- ▶ das Anfügen eines **Paritätsbits** an ein Binärcodewort $z = (z_1, \dots, z_n)$ ist die einfachste Methode zur Erkennung von Einbitfehlern
- ▶ die Parität wird berechnet als

$$p = \left(\sum_{i=1}^n z_i \right) \bmod 2$$

- ▶ **gerade Parität** (*even parity*): $y_{\text{even}} = (z_1, \dots, z_n, p)$
 $p(y_{\text{even}}) = (\sum_i y_i) \bmod 2 = 0$
- ▶ **ungerade Parität** (*odd parity*): $y_{\text{odd}} = (z_1, \dots, z_n, \bar{p})$
 $p(y_{\text{odd}}) = (\sum_i y_i) \bmod 2 = 1$



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	Fehlerfall	100 1000		0
A	100 0001		0		100 0101		0
M	100 1101		0		110 1101		0
M	100 1101		0		100 1101		0
I	100 1001		1		000 1001		1
N	100 1110		0		100 1110		0
G	100 0111		0		100 0111		0
	<hr/>				<hr/>		
	100 1001		1		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: Beispiel (cont.)

H	100 1000		0	Fehlerfall	100 1000		0
A	100 0001		0		100 0101		0 1
M	100 1101		0		100 1101		0
M	100 1101		0		100 1101		0
I	100 1001		1		100 1001		1
N	100 1110		0		100 1110		0
G	100 0111		0		100 0111		0
	<hr/>				<hr/>		
	100 1001		1		100 1001		1
					1		

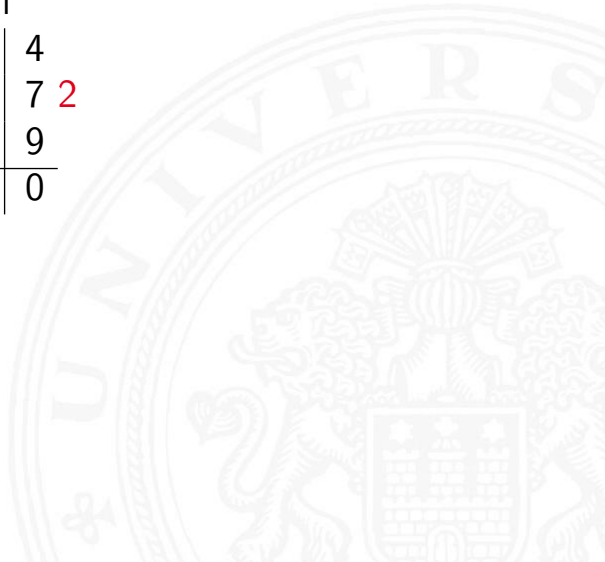
- ▶ Empfänger: berechnet Parität und vergleicht mit gesendeter Parität
- ▶ Einzelfehler: Abweichung in je einer Zeile und Spalte
⇒ Fehler kann daher zugeordnet und korrigiert werden
- ▶ Mehrfachfehler: nicht alle, aber viele erkennbar (korrigierbar)



Zweidimensionale Parität: Dezimalsystem

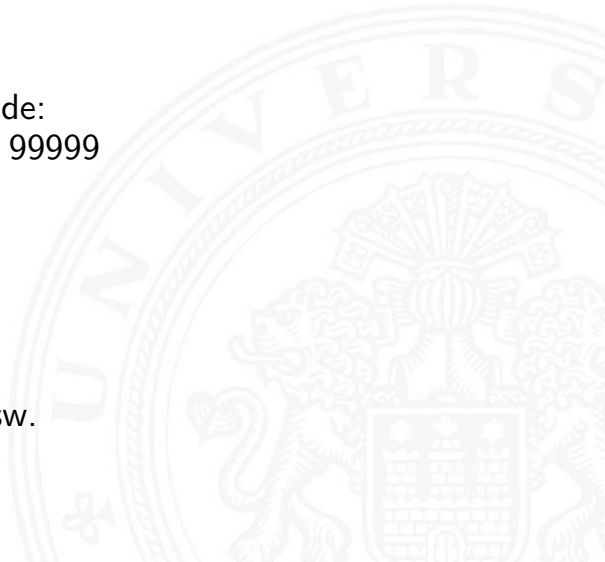
- ▶ Parität als Zeilen/Spaltensumme mod 10 hinzufügen

▶ Daten	Parität	Fehlerfall
3 7 4	3 7 4 4	3 7 4 4
5 4 8	5 4 8 7	5 4 3 7 2
1 3 5	1 3 5 9	1 3 5 9
	<hr/> 9 4 7 0	<hr/> 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, \dots, z_{10}
- ▶ Prüfsumme berechnen, Symbol X steht für Ziffer 10

$$\sum_{i=1}^9 (i \cdot z_i) \bmod 11 = z_{10}$$

- ▶ ISBN-Zahl zulässig, genau dann wenn

$$\sum_{i=1}^{10} (i \cdot z_i) \bmod 11 = 0$$

- ▶ Beispiel: 1-292-10176-8

Bryant, O'Hallaron [BO15]

$$1 \cdot 1 + 2 \cdot 2 + 3 \cdot 9 + 4 \cdot 2 + 5 \cdot 1 + 6 \cdot 0 + 7 \cdot 1 + 8 \cdot 7 + 9 \cdot 6 = 162 \quad 162 \bmod 11 = 8$$

$$\text{Prüfung: } 162 + 10 \cdot 8 = 242 \quad 242 \bmod 11 = 0$$



ISBN: Fehlertypen

- ▶ Prüfziffer schützt gegen Verfälschung einer Ziffer
 - "- Vertauschung zweier Ziffern
 - "- „Falschdopplung“ einer Ziffer

- ▶ Beispiel: vertausche i -te und j -te Ziffer (mit $i \neq j$)

$$\begin{aligned} \text{Prüfsumme: } \langle \text{korrekt} \rangle - \langle \text{falsch} \rangle &= i \cdot z_i + j \cdot z_j - j \cdot z_i - i \cdot z_j \quad \text{mit } z_i \neq z_j \\ &= (i - j) \cdot (z_i - z_j) \end{aligned}$$



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)
 - $u = 0 : x = (111)^T \times (0) = (000)$
 - $u = 1 : x = (111)^T \times (1) = (111)$
- ▶ 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, \dots, d_n)
um k -Prüfbits ergänzen (p_1, p_2, \dots, p_k)
- ⇒ 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



(7,4)-Hamming-Code

- ▶ sieben Codebits für je vier Datenbits
- ▶ linearer (7,4)-Blockcode
- ▶ 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 $c = G \times d$



(7,4)-Hamming-Code (cont.)

- ▶ Prüfmatrix H orthogonal zu gültigen Codewörtern: $H \times c = 0$

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

für ungültige Codewörter $H \times c \neq 0$

⇒ „Fehlersyndrom“ liefert Information über Fehlerposition / -art

Fazit: Hamming-Codes

- + größere Wortlängen: besseres Verhältnis von Nutz- zu Prüfbits
- + einfaches Prinzip, einfach decodierbar
- es existieren weit bessere Codes



(7,4)-Hamming-Code: Beispiel

- ▶ Codieren von $d = (0, 1, 1, 0)$

$$c = G \times d = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$



(7,4)-Hamming-Code: Beispiel (cont.)

- ▶ Prüfung von Codewort $c = (1, 1, 0, 0, 1, 1, 0)$

$$H \times c = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$



(7,4)-Hamming-Code: Beispiel (cont.)

▶ im Fehlerfall $c = (1, 1, 1, 0, 1, 1, 0)$

$$H \times c = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

⇒ Fehlerstelle: $(1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0)$

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Index: $1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7$



Fehlerrate

- ▶ Übertragungskanal hat Bitfehlerwahrscheinlichkeit
- ▶ (n,k) -Code: k -Informationsbits werden in n -Bits codiert
- ▶ Minimalabstand d der Codewörter voneinander
- ▶ ermöglicht Korrektur von r Bitfehlern $r \leq (d - 1)/2$

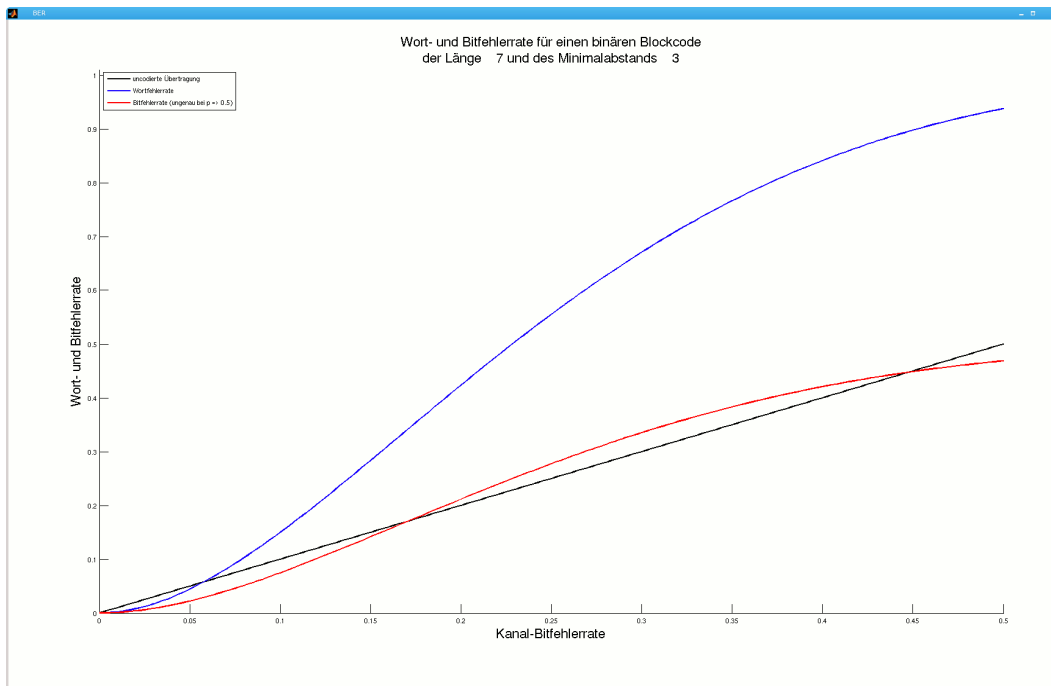
⇒ nicht korrigierbar sind: $r + 1, r + 2, \dots, n$ Bitfehler

⇒ Wortfehlerwahrscheinlichkeit

- ▶ Summe der Wahrscheinlichkeiten nicht korrigierbarer Bitfehler
- ▶ von Wortlänge abhängig



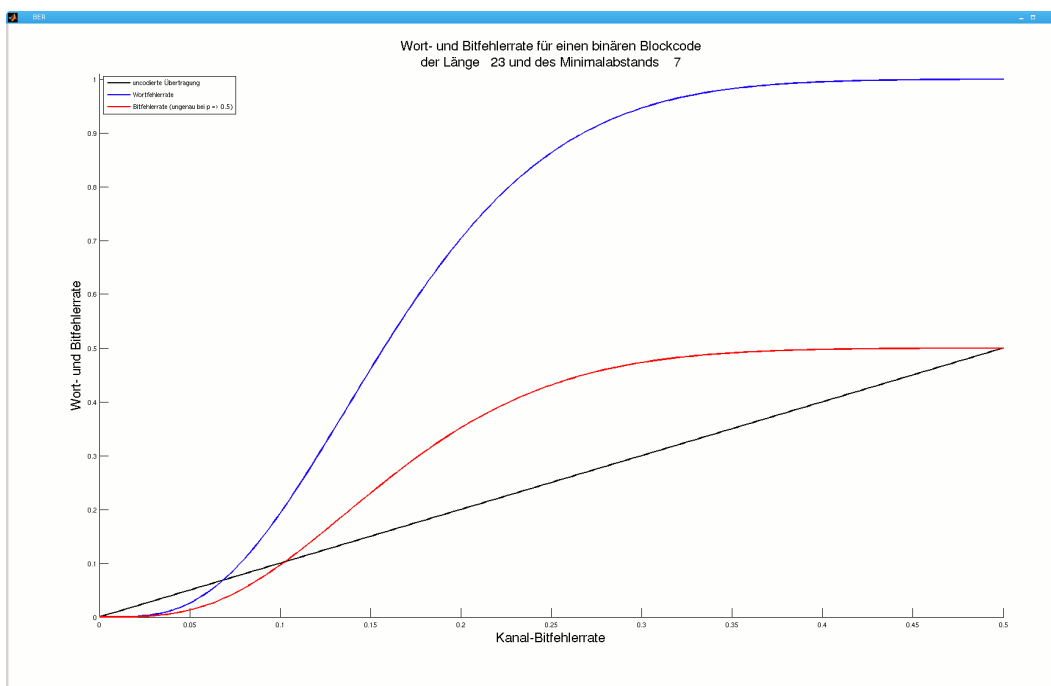
Fehlerrate: (7,4)-Hamming-Code



[Hei05]



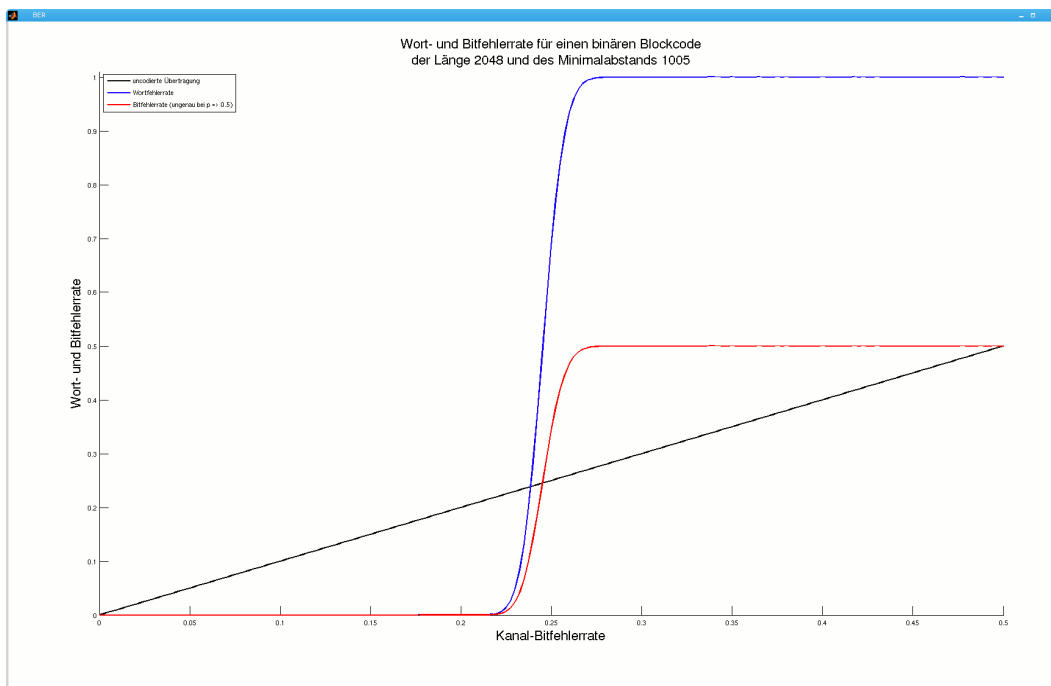
Fehlerrate: (23,12)-Golay-Code



[Hei05]



Fehlerrate: (2048,8)-Randomcode



[Hei05]



Binärpolynome

- ▶ jedem n -bit Wort (d_1, d_2, \dots, d_n) lässt sich ein Polynom über dem Körper mit den Elementen $\{0, 1\}$ zuordnen
- ▶ Beispiel, mehrere mögliche Zuordnungen

$$\begin{aligned}
 100\ 1101 &= 1 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 1 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0 \\
 &= x^6 + x^3 + x^2 + x^0 \\
 &= x^0 + x^3 + x^4 + x^6 \\
 &= x^0 + x^{-3} + x^{-4} + x^{-6} \\
 &\dots
 \end{aligned}$$

- ▶ mit diesen Polynomen kann „gerechnet“ werden: Addition, Subtraktion, Multiplikation, Division
- ▶ Theorie: Galois-Felder $GF(2)^n$



Zyklische Codes (CRC)

7.10 Codierung - Zyklische Codes

64-040 Rechnerstrukturen und Betriebssysteme

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 ...

A. Mäder

438

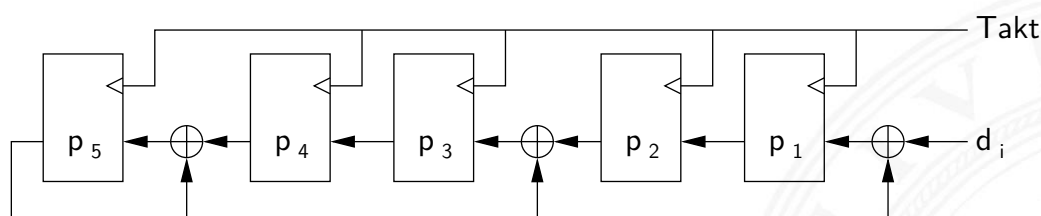


Zyklische Codes (CRC) (cont.)

7.10 Codierung - Zyklische Codes

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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

A. Mäder

439



Zyklische Codes (CRC) (cont.)

7.10 Codierung - Zyklische Codes

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Test bei Empfänger
 - ▶ übertragenes Wort bitweise in CRC-Check schieben
gleiches Polynom / Hardware wie bei Codierung
 - ▶ fehlerfrei, wenn Divisionsrest/Registereinhalt = 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

A. Mäder

440



EAN-13 Produktcode

de.wikipedia.org/wiki/European_Article_Number

7.11 Codierung - Praxisbeispiele

64-040 Rechnerstrukturen und Betriebssysteme

Kombination diverser Codierungen:

- ▶ Land, Unternehmen, Artikelnummer, Prüfsumme
- ▶ 95-stelliges Bitmuster
 - ▶ schwarz $\hat{=}$ 1, weiss $\hat{=}$ 0
 - ▶ max. vier aufeinanderfolgende weisse/schwarze Bereiche
 - ▶ 101 als Randzeichen
01010 als Trennzeichen in der Mitte
- ▶ 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



A. Mäder

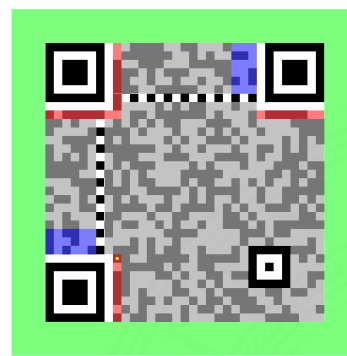
441



QR-Code

de.wikipedia.org/wiki/QR-Code en.wikipedia.org/wiki/QR_code

- ▶ verschiedene internationale Standards
- ▶ mehrere Fehlerkorrekturlevel bis zu 30 % der Daten
- ▶ Größen: 21×21... 177×177 Pixel
- ▶ Zeichensätze: Ziffern, alphanum., byte (ISO-8859-1), Kanji...
- ▶ Format: BCH-Codierung
Daten: Reed-Solomon Code (8-bit Blöcke)
- ▶ 8 Maskierungsarten (XOR) für max. hell/dunkel Wechsel
- ▶ viele Varianten / Erweiterungen
 - ▶ Grafiken, Logos (Fehlerkorrektur ausnutzen)
 - ▶ Frame-QR-Code mit Zeichenfläche
 - ▶ mehrere Farben für extra Daten: HCC2D (High Capacity Colored 2-Dimensional Code), JAB (Just Another Barcode)...

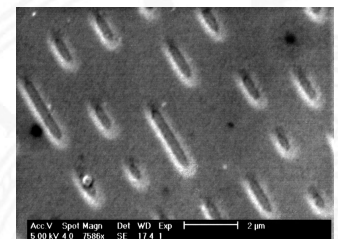
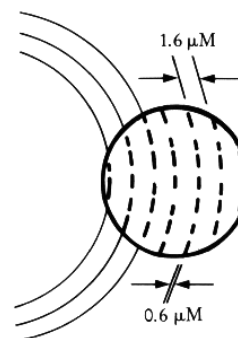
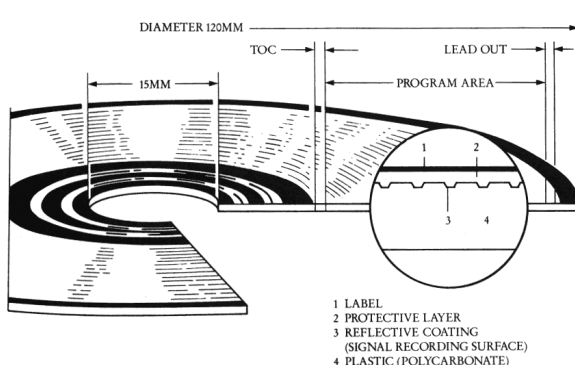


- 1. Version
- 2. Format
- 3. Fehlerkorrigierbare Daten
- 4. Erforderliche Muster:
 - 4.1. Position
 - 4.2. Ausrichtung
 - 4.3. Synchronisation
 - 4.4. dark module
- 5. Ruhezone



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



Compact Disc (cont.)

Audio-CD und CD-ROM

- ▶ Auslesen durch Intensität von reflektiertem Laserstrahl
- ▶ 650 MiB Kapazität, Datenrate ≈ 150 KiB/sec (1x speed)
- ▶ 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

7.11 Codierung - Praxisbeispiele

64-040 Rechnerstrukturen und Betriebssysteme

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 (<i>discrete cosinus transformation</i>) | verlustfrei |
| 4. Quantisierung (einstellbar) | verlustbehaftet |
| 5. Huffman-Codierung | verlustfrei |

A. Mäder

446



Video: MPEG

7.11 Codierung - Praxisbeispiele

64-040 Rechnerstrukturen und Betriebssysteme

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

A. Mäder

447



- [Ham87] R.W. Hamming: *Information und Codierung*.
VCH, 1987. ISBN 978-3-527-26611-1
- [Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript*.
Universität Hamburg, FB Informatik, 2005, Vorlesungsskript.
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



- [RL09] W.E. Ryan, S. Lin: *Channel codes: classical and modern*.
Cambridge University Press, 2009. ISBN 978-0-521-84868-8
- [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
www-cs-faculty.stanford.edu/~knuth/taocp.html



Gliederung

8 Schaltfunktionen

64-040 Rechnerstrukturen und Betriebssysteme

1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. Logische Operationen
7. Codierung
8. **Schaltfunktionen**
 - Definition
 - Darstellung
 - Normalformen
 - Entscheidungsbäume und OBDDs
 - Realisierungsaufwand und Minimierung



Gliederung (cont.)

8 Schaltfunktionen

64-040 Rechnerstrukturen und Betriebssysteme

Minimierung mit KV-Diagrammen
Literatur

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





- ▶ **Schaltfunktion:** eine eindeutige Zuordnungsvorschrift f , die jeder Wertekombination (b_1, b_2, \dots, b_n) von Schaltvariablen einen Wert zuweist:

$$y = f(b_1, b_2, \dots, b_n) \in \{0, 1\}$$

- ▶ **Schaltvariable:** eine Variable, die nur endlich viele Werte annehmen kann, typisch sind binäre Schaltvariablen
- ▶ **Ausgangvariable:** die Schaltvariable am Ausgang der Funktion, für den Wert y
- ▶ bereits bekannt: *elementare Schaltfunktionen* (AND, OR usw.)
wir betrachten jetzt Funktionen von n Variablen



- ▶ 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)
- ⇒ daher nur für kleine Funktionen geeignet
- ▶ Erweiterung auf *don't-care* Terme, s.u.



- ▶ Beschreibung einer Funktion als Text über ihr Verhalten
- ▶ Problem: umgangssprachliche Formulierungen oft mehrdeutig
- ▶ logische Ausdrücke in Programmiersprachen
- ▶ Einsatz spezieller (Hardware-) Beschreibungssprachen: Verilog, VHDL, SystemC



„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?

⇒ je nach Interpretation völlig unterschiedliche Schaltung



- ▶ **Strukturbeschreibung:** eine Spezifikation der konkreten Realisierung einer Schaltfunktion
- ▶ vollständig geklammerte algebraische Ausdrücke
$$f = x_1 \oplus (x_2 \vee x_3)$$
- ▶ Datenflussgraphen
- ▶ Blockschaltbilder
- ▶ Schaltpläne mit Gattern (Kapitel 9)
- ▶ PLA-Format für zweistufige AND-OR Schaltungen
- ▶ ...



- ▶ 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 }



- ▶ 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



Normalform einer Boole'schen Funktion

- ▶ analog zur Potenzdarstellung von Polynomen
- ▶ als Summe über Koeffizienten $\{0,1\}$ und Basisfunktionen

$$f = \sum_{i=1}^{2^n} \hat{f}_i \hat{B}_i, \quad \hat{f}_i \in \text{GF}(2)$$

mit $\hat{B}_1, \dots, \hat{B}_{2^n}$ 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

$$f = (\hat{f}_1 \otimes \hat{B}_1) \oplus \dots \oplus (\hat{f}_{2^n} \otimes \hat{B}_{2^n})$$

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

$$f = \bigvee_{i=1}^{2^n} \hat{f}_i \cdot m(i), \quad \text{mit } m(i) : \text{Minterm}(i)$$

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 $0 \equiv \overline{x_i}$ $1 \equiv x_i$
 - ▶ für f sind nur drei Koeffizienten der DNF gleich 1
- ⇒ 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 von Termen (ODER)
 - ▶ jeder Term besteht aus der UND-Verknüpfung von Schaltvariablen, direkt oder negiert
 - ▶ in Termen müssen nicht alle Schaltvariablen vorkommen (anders als bei Mintermen)
- ▶ entspricht dem Zusammenfassen von Termen aus der disjunktiven Normalform
schaltalgebraische Minimierung
- ▶ disjunktive Form ist nicht eindeutig \Rightarrow ist keine Normalform

▶ Beispiel

DNF

$$f(x) = (\bar{x}_3 \wedge x_2 \wedge \bar{x}_1) \vee (\bar{x}_3 \wedge x_2 \wedge x_1) \vee (x_3 \wedge x_2 \wedge \bar{x}_1)$$

minimierte disjunktive Form

$$f(x) = (\bar{x}_3 \wedge x_2) \vee (x_3 \wedge x_2 \wedge \bar{x}_1)$$



Allgemeine disjunktive Form

- ▶ **Disjunktive Form** (sum-of-products)
 - ▶ die disjunktive Verknüpfung von Termen (ODER)
 - ▶ jeder Term besteht aus der UND-Verknüpfung von Schaltvariablen, direkt oder negiert
 - ▶ in Termen müssen nicht alle Schaltvariablen vorkommen (anders als bei Mintermen)
- ▶ entspricht dem Zusammenfassen von Termen aus der disjunktiven Normalform
schaltalgebraische Minimierung
- ▶ disjunktive Form ist nicht eindeutig \Rightarrow ist keine Normalform

▶ Beispiel

DNF

$$f(x) = (\bar{x}_3 \wedge x_2 \wedge \bar{x}_1) \vee (\bar{x}_3 \wedge x_2 \wedge x_1) \vee (x_3 \wedge x_2 \wedge \bar{x}_1)$$

minimierte disjunktive Form

$$f(x) = (\bar{x}_3 \wedge x_2) \vee (x_3 \wedge x_2 \wedge \bar{x}_1)$$

$$f(x) = (x_2 \wedge \bar{x}_1) \vee (\bar{x}_3 \wedge x_2 \wedge x_1)$$



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

$$f = \bigwedge_{i=1}^{2^n} \hat{f}_i \cdot \mu(i), \quad \text{mit } \mu(i) : \text{Maxterm}(i)$$

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 $0 \equiv x_i$ $1 \equiv \bar{x}_i$
 - ▶ für f sind fünf Koeffizienten der KNF gleich 0
- ⇒ KNF: $f(x) = (x_3 \vee x_2 \vee x_1) \wedge (x_3 \vee x_2 \vee \bar{x}_1) \wedge (\bar{x}_3 \vee x_2 \vee x_1) \wedge (\bar{x}_3 \vee x_2 \vee \bar{x}_1) \wedge (\bar{x}_3 \vee \bar{x}_2 \vee \bar{x}_1)$



Allgemeine konjunktive Form

- ▶ **Konjunktive Form** (product-of-sums)
 - ▶ die konjunktive Verknüpfung von Termen (UND)
 - ▶ jeder Term besteht aus der ODER-Verknüpfung von Schaltvariablen, direkt oder negiert
 - ▶ in Termen müssen nicht alle Schaltvariablen vorkommen (anders als bei Maxtermen)
- ▶ entspricht dem Zusammenfassen von Termen aus der konjunktiven Normalform
schaltalgebraische Minimierung
- ▶ konjunktive Form ist nicht eindeutig ⇒ ist keine Normalform
- ▶ Beispiel

KNF $f(x) = (x_3 \vee x_2 \vee x_1) \wedge (x_3 \vee x_2 \vee \bar{x}_1) \wedge (\bar{x}_3 \vee x_2 \vee x_1) \wedge (\bar{x}_3 \vee x_2 \vee \bar{x}_1) \wedge (\bar{x}_3 \vee \bar{x}_2 \vee \bar{x}_1)$

minimierte konjunktive Form
 $f(x) = (x_3 \vee x_2) \wedge (x_2 \vee x_1) \wedge (\bar{x}_3 \vee \bar{x}_1)$



Anmerkung

- ▶ nach der Boole'schen Algebra sind **Konjunktion** \wedge und **Disjunktion** \vee gleichwertig!
⇒ es gibt also keine vorrangige Operation
- ▶ häufig wird folgende verkürzte **multiplikative Schreibweise** in schaltalgebraischen Ausdrücken benutzt
- ▶ ab statt $a \wedge b$
 $ab \vee cd$ statt $(a \wedge b) \vee (c \wedge d)$
- ⇒ der Operator \wedge wird weggelassen oder durch \cdot ersetzt
- ⇒ die Ausdrücke sind (implizit) geklammert ⇒ \wedge bindet stärker als \vee (falsch)



- ▶ **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



- ▶ Basisfunktionen sind:

$\{1\},$	(0 Variablen)
$\{1, x_1\},$	(1 Variable)
$\{1, x_1, x_2, x_2x_1\},$	(2 Variablen)
$\{1, x_1, x_2, x_2x_1, x_3, x_3x_1, x_3x_2, x_3x_2x_1\},$	(3 Variablen)
...	
$\{RM(n-1), x_n \cdot RM(n-1)\}$	(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



Umrechnung von gegebenem Ausdruck in Reed-Muller Form?

- ▶ Ersetzen der Negation: $\bar{a} = a \oplus 1$
- Ersetzen der Disjunktion: $a \vee b = a \oplus b \oplus ab$
- Ausnutzen von: $a \oplus a = 0$

- ▶ Beispiel

$$\begin{aligned}
 f(x_1, x_2, x_3) &= (\bar{x}_1 \vee x_2)x_3 \\
 &= (\bar{x}_1 \oplus x_2 \oplus \bar{x}_1x_2)x_3 \\
 &= ((1 \oplus x_1) \oplus x_2 \oplus (1 \oplus x_1)x_2)x_3 \\
 &= (1 \oplus x_1 \oplus x_2 \oplus x_2 \oplus x_1x_2)x_3 \\
 &= x_3 \oplus x_1x_3 \oplus x_1x_2x_3
 \end{aligned}$$



Reed-Muller Form: Transformationsmatrix

- ▶ lineare Umrechnung zwischen Funktion f , bzw. deren Funktionstabelle, und RMF
- ▶ Transformationsmatrix A kann rekursiv definiert werden (ähnlich den RMF-Basisfunktionen)
- ▶ Multiplikation von A mit f ergibt Koeffizientenvektor r der RMF

$$r = A \times f \quad \text{und} \quad f = A \times r$$

gilt wegen: $r = A \times f$ und $A \times A = I$, also $f = A \times r$!

$$A_0 = (1)$$

$$A_1 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$



Reed-Muller Form: Transformationsmatrix (cont.)

$$A_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$A_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

...

$$A_n = \begin{pmatrix} A_{n-1} & 0 \\ A_{n-1} & A_{n-1} \end{pmatrix}$$



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: $f(x) = x_2 \oplus x_3x_2x_1$
- ▶ 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

$$r = A_3 \times f = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

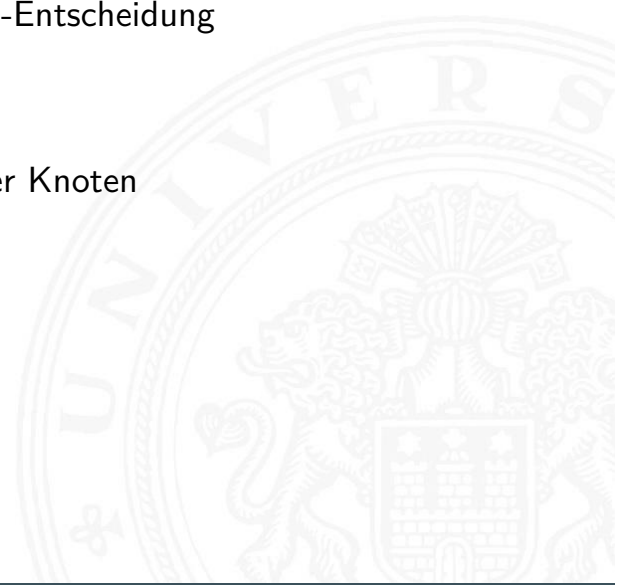
Basisfunktionen: $\{1, x_1, x_2, x_2x_1, x_3, x_3x_1, x_3x_2, x_3x_2x_1\}$

führt zur gesuchten RMF: $f(x_3, x_2, x_1) = r \times RM(3) = x_2 \oplus x_3x_2x_1$

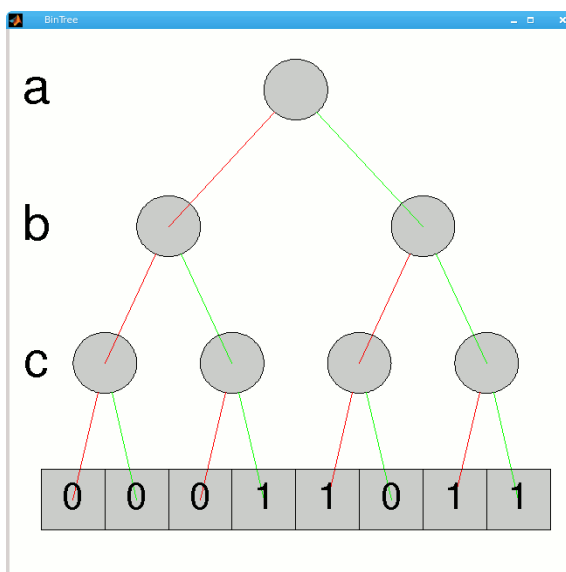


Grafische Darstellung: Entscheidungsbäume

- ▶ 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
 $f(a,b,c) = (a \wedge \bar{c}) \vee (b \wedge c)$



Entscheidungsbaum: Beispiel



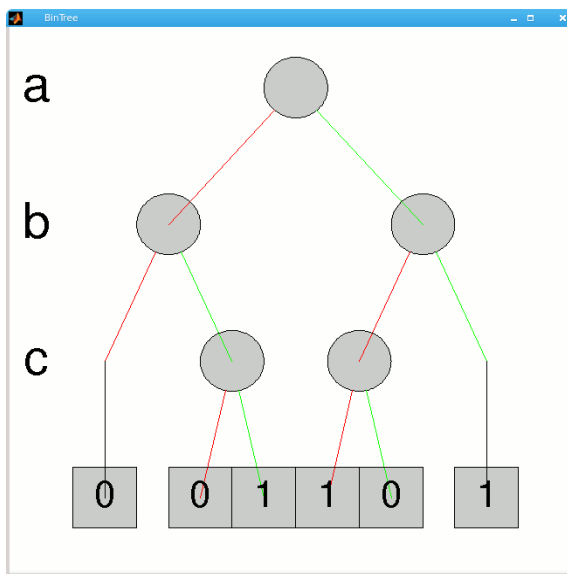
▶ $f(a,b,c) = (a \wedge \bar{c}) \vee (b \wedge c)$

- ▶ rot: 0-Zweig
- grün: 1-Zweig





Entscheidungsbaum: Beispiel (cont.)



$$f(a,b,c) = (a \wedge \bar{c}) \vee (b \wedge c)$$

⇒ 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
 $\mathcal{O}(n) \dots \mathcal{O}(n^2)$ Knoten bei n Variablen
- ▶ wichtige Ausnahme: n -bit Multiplizierer ist $\mathcal{O}(2^n)$
- ▶ derzeit das Standardverfahren zur Manipulation von (großen) Schaltfunktionen

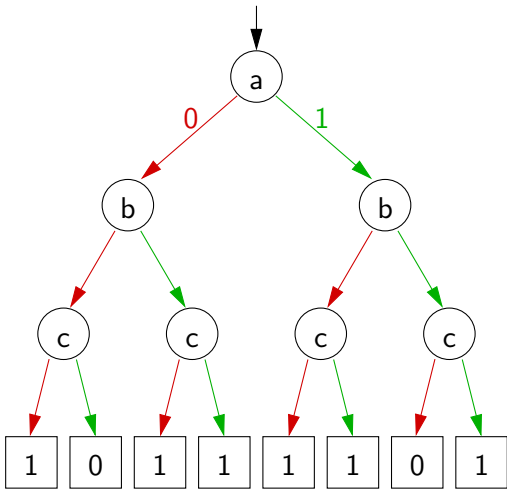
R. E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*, [Bry86]



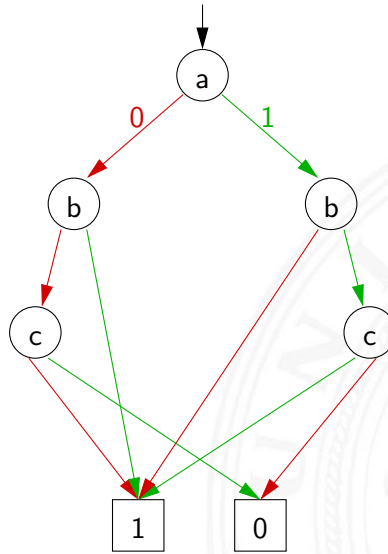
ROBDD vs. Entscheidungsbaum

Entscheidungsbaum

$$f = (abc) \vee (a\bar{b}) \vee (\bar{a}b) \vee (\bar{a}\bar{b}\bar{c})$$

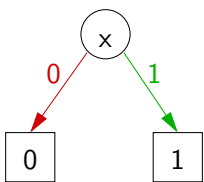


ROBDD

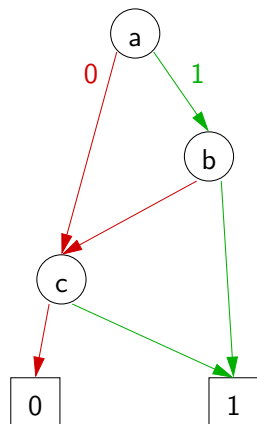


ROBDD: Beispiele

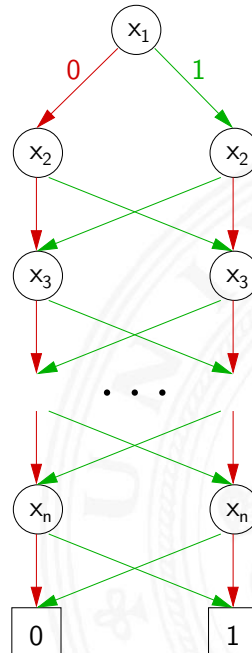
$$f(x) = x$$



$$g = (ab) \vee c$$



$$\text{Parität } p = x_1 \oplus x_2 \oplus \dots \oplus x_n$$



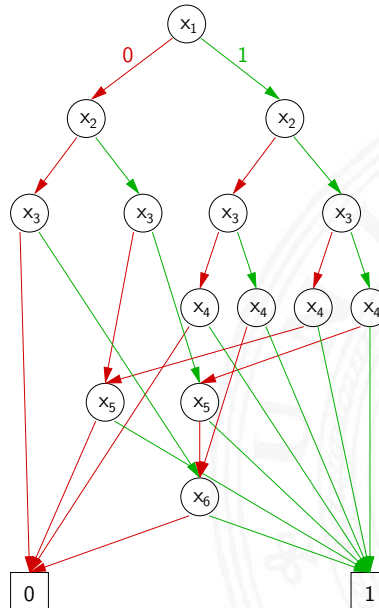
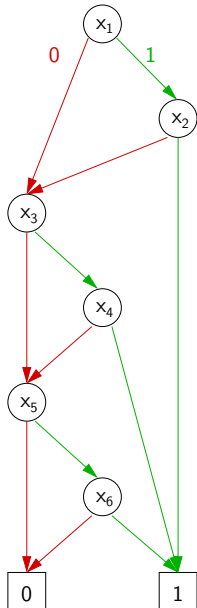


ROBDD: Problem der Variablenordnung

- 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
- Hardwareeffizienz
- Geschwindigkeit
- Testbarkeit
- Robustheit

Anzahl der Gatter / Chipfläche

z.B. NAND statt XOR

Anzahl der Stufen, Laufzeiten

Erkennung von Produktionsfehlern

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

$$\begin{aligned}
 y(x) = & \bar{x}_3 x_2 x_1 \bar{x}_0 \vee \bar{x}_3 x_2 x_1 x_0 \\
 & \vee x_3 \bar{x}_2 \bar{x}_1 x_0 \vee x_3 \bar{x}_2 x_1 \bar{x}_0 \\
 & \vee x_3 \bar{x}_2 x_1 x_0 \vee x_3 x_2 \bar{x}_1 x_0 \\
 & \vee x_3 x_2 x_1 \bar{x}_0 \vee x_3 x_2 x_1 x_0
 \end{aligned}$$

- ▶ Zusammenfassen von Termen liefert

$$y(x) = \bar{x}_3 x_2 x_1 \vee x_3 \bar{x}_2 x_0 \vee x_3 \bar{x}_2 x_1 \vee x_3 x_2 x_0 \vee x_3 x_2 x_1$$



Algebraische Minimierung: Beispiel

- ▶ Ausgangsfunktion in DNF

$$\begin{aligned}
 y(x) = & \overline{x_3} x_2 x_1 \overline{x_0} \vee \overline{x_3} x_2 x_1 x_0 \\
 & \vee x_3 \overline{x_2} \overline{x_1} x_0 \vee x_3 \overline{x_2} x_1 \overline{x_0} \\
 & \vee x_3 \overline{x_2} x_1 x_0 \vee x_3 x_2 \overline{x_1} x_0 \\
 & \vee x_3 x_2 x_1 \overline{x_0} \vee x_3 x_2 x_1 x_0
 \end{aligned}$$

- ▶ Zusammenfassen von Termen 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 / als konjunktive Normalform
- ▶ Zusammenfassen benachbarter Terme durch **Schleifen**
- ▶ alle 1-Terme mit möglichst wenigen Schleifen abdecken
(alle 0-Terme $\overline{\text{---}}$ \equiv konjunktive Normalform)
- ▶ minimierte Funktion ablesen, wenn keine weiteren Schleifen gebildet werden können
- ▶ nutzt Fähigkeit des Menschen, benachbarte Flächen auf einen Blick zu „sehen“
- ▶ bei mehr als 6 Variablen nicht mehr praktikabel



Erinnerung: Karnaugh-Veitch Diagramm

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

	$x_1 x_0$			
	00	01	11	10
$x_3 x_2$				
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
- ⇒ benachbarte Felder unterscheiden sich gerade um 1 Bit



KV-Diagramme: 2...4 Variable (2×2 , 2×4 , 4×4)

	x_0	
	0	1
x_1		
0	00	01
1	10	11

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

	$x_1 x_0$			
	00	01	11	10
x_2				
0	000	001	011	010
1	100	101	111	110



KV-Diagramm für Schaltfunktionen

- ▶ Funktionswerte 0 und 1 in zugehöriges Feld im KV-Diagramm eintragen
zusätzlich: *Don't-Care* „*“ für nicht spezifizierte Werte

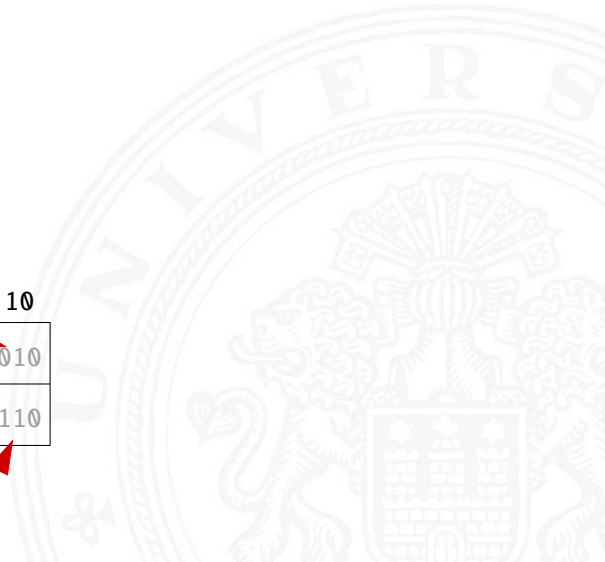
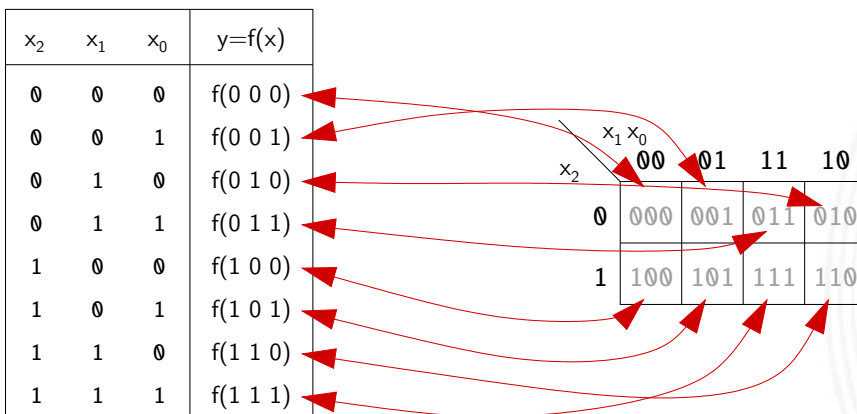
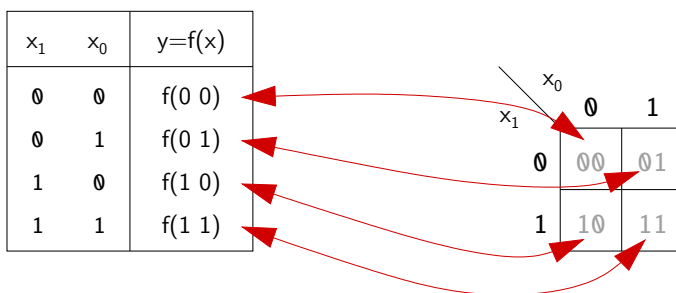
wichtig!

- ▶ 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

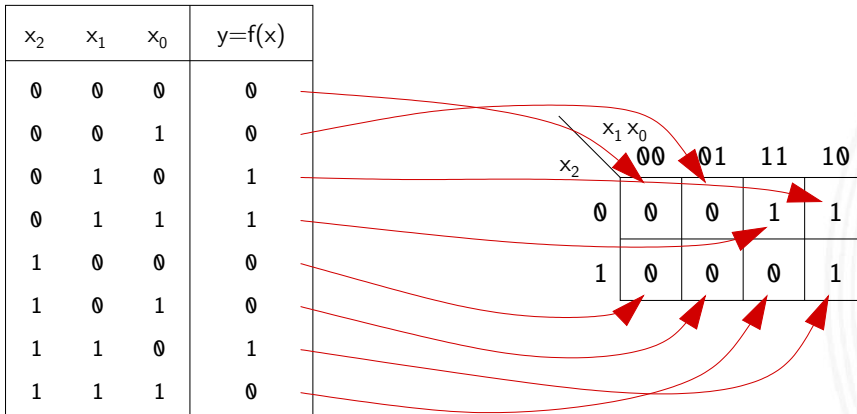
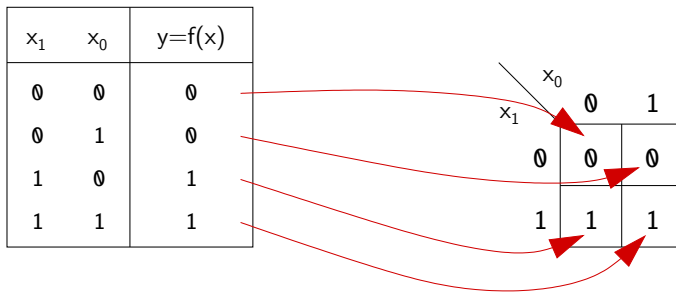


KV-Diagramm: Zuordnung zur Funktionstabelle

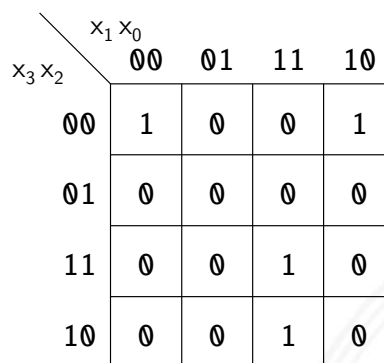
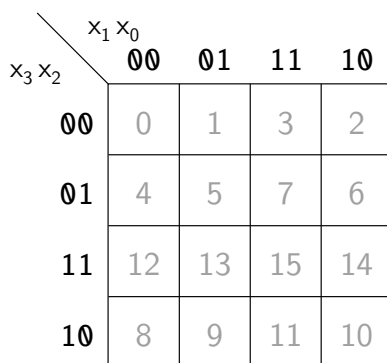




KV-Diagramm: Eintragen aus Funktionstabelle



KV-Diagramm: Beispiel



- ▶ 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

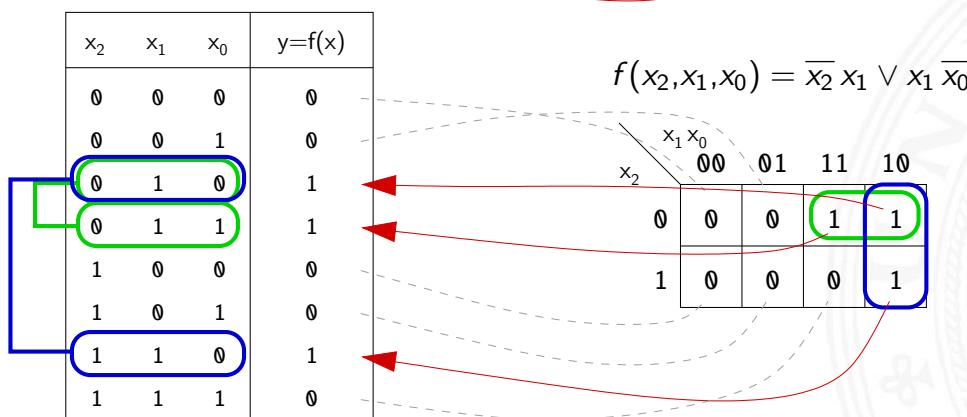
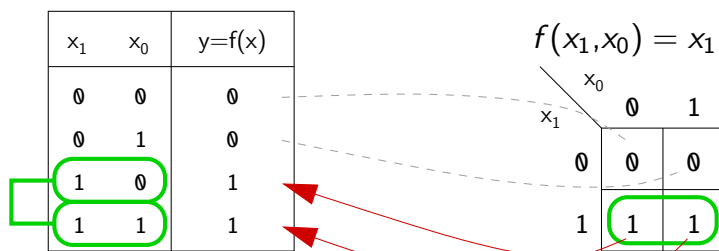


Schleifen: Zusammenfassen benachbarter Terme

- ▶ 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
- ▶ die zugehörigen (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 im KV-Diagramm





Schleifen: Ablesen im KV-Diagramm (cont.)

8.6 Schaltfunktionen - Minimierung mit KV-Diagrammen

64-040 Rechnerstrukturen und Betriebssysteme

$x_3 x_2$	$x_1 x_0$ 00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	0

$x_3 x_2$	$x_1 x_0$ 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

8.6 Schaltfunktionen - Minimierung mit KV-Diagrammen

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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*
 - 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
⇒ andere Anordnung der Terme im KV-Diagramm

Tipp!



KV-Diagramm Editor: Screenshots

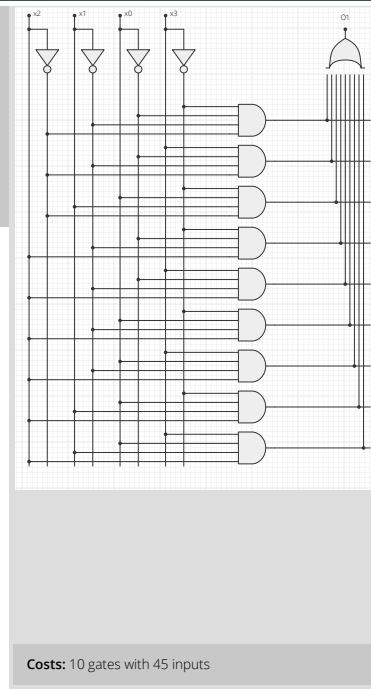
Edit function Edit loops

Inputs: 4 Outputs: 1

01

DNF KNF No loops have been created yet

	x0				
	1	0	1	0	
	1	1	1	0	
x2	1	1	1	0	
x3	1	0	0	0	
					x1



Eingabe der Schaltfunktion



KV-Diagramm Editor: Screenshots (cont.)

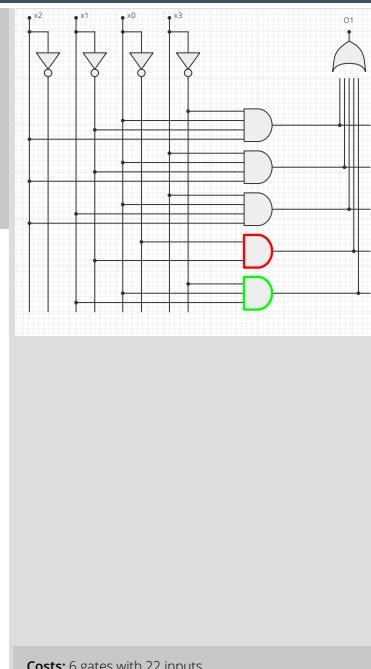
Edit function Edit loops

Inputs: 4 Outputs: 1

01

DNF KNF

	x0				
	1	0	1	0	
	1	1	1	0	
x2	1	1	1	0	
x3	1	0	0	0	
					x1



Minimierung durch Schleifenbildung



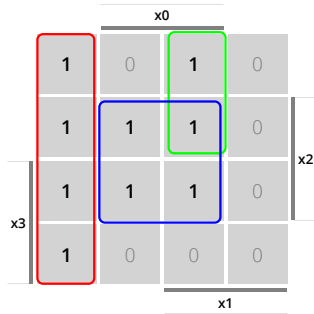
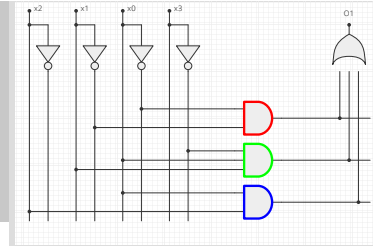
KV-Diagramm Editor: Screenshots (cont.)

Edit function Edit loops

Inputs: 4 Outputs: 1

O1

DNF KNF x x x



Costs: 4 gates with 10 inputs

Hardware-Kosten: # Gatter, Eingänge



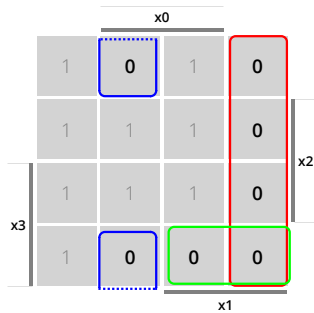
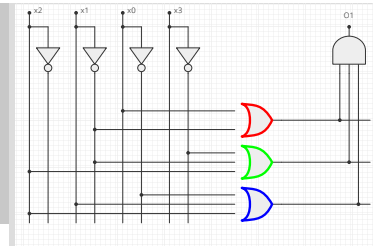
KV-Diagramm Editor: Screenshots (cont.)

Edit function Edit loops

Inputs: 4 Outputs: 1

O1

DNF KNF x x x



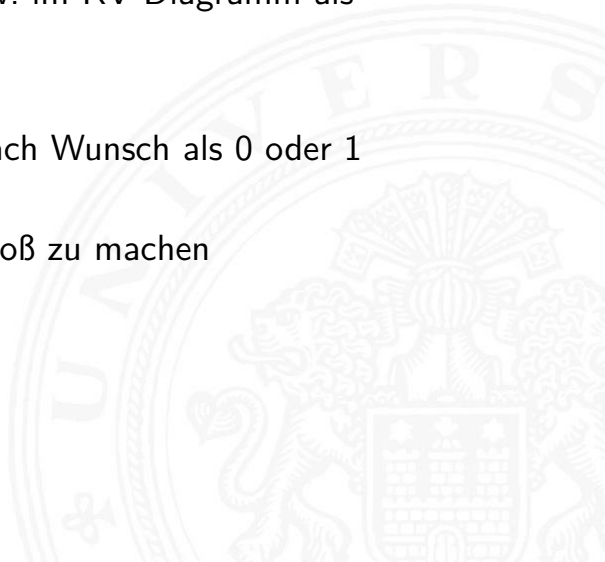
Costs: 4 gates with 11 inputs

Konjunktive Form



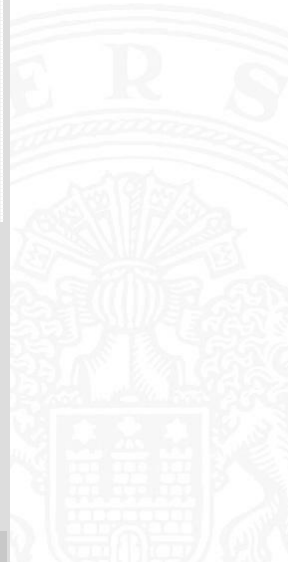
Don't-Care Terme

- ▶ in der Praxis sind viele Schaltfunktionen unvollständig definiert weil bestimmte Eingangskombinationen nicht vorkommen
 - ▶ zugehörige Terme werden in Funktionstabelle, bzw. im KV-Diagramm als **Don't-Care** markiert: „*“
- ⇒ bei der Minimierung können *Don't-Care* Terme nach Wunsch als 0 oder 1 angenommen werden
- ⇒ *Don't-Cares* verwenden um Schleifen möglichst groß zu machen



KV-Diagramm Editor: 6 Variablen, Don't-Cares

Costs: 9 gates with 56 inputs





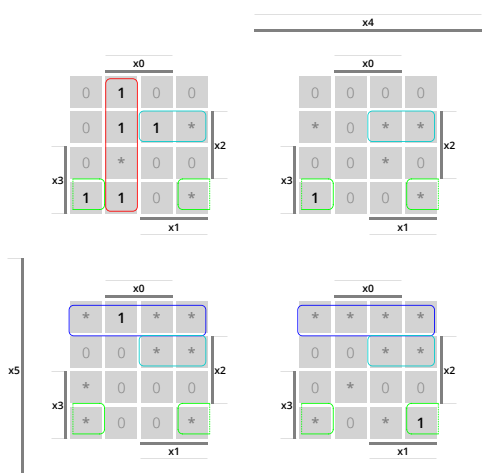
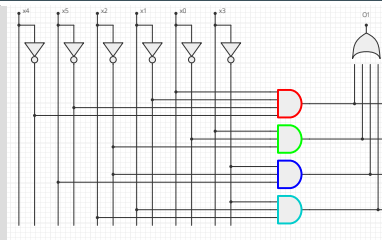
KV-Diagramm Editor: 6 Variablen, *Don't-Cares* (cont.)

Edit function Edit loops

Inputs: 6 Outputs: 1

O1

DNF KNF x x x x



Costs: 5 gates with 17 inputs



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



[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, Vorlesungsskript.

tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1



1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. Logische Operationen
7. Codierung
8. Schaltfunktionen
9. **Schaltnetze**
 - Definition
 - Schaltpläne und Symbole
 - Hades: Editor und Simulator



Gliederung (cont.)

9 Schaltnetze

64-040 Rechnerstrukturen und Betriebssysteme

Logische Gatter

Inverter, AND, OR

XOR und Parität

Einfache Schaltnetze

Multiplexer, Demultiplexer

Prioritätsencoder

Beispiele

ALU Schaltnetze

Addierer

Multiplizierer

Barrel-Shifter

ALU (Arithmetisch-Logische Einheit)

Zeitverhalten von Schaltungen

Hazards

Literatur

10. Schaltwerke



Gliederung (cont.)

9 Schaltnetze

64-040 Rechnerstrukturen und Betriebssysteme

11. Rechnerarchitektur I

12. Instruction Set Architecture

13. Assembler-Programmierung

14. Rechnerarchitektur II

15. Betriebssysteme





- ▶ **Schaltnetz** oder auch **kombinatorische Schaltung** (*combinational logic circuit*) ein digitales System mit n Eingängen (b_1, b_2, \dots, b_n) und m -Ausgängen (y_1, y_2, \dots, 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



- ▶ Begriff: „Schaltnetz“
 - ▶ technische Realisierung von Schaltfunktionen / Funktionsbündeln
 - ▶ Struktur aus einfachen Gatterfunktionen
Gatter: triviale Funktion mit wenigen (2...4) Eingängen
- ▶ Schaltnetze werden nicht nur statisch betrachtet
 - ▶ Gatterlaufzeiten spielen eine Rolle
 - ▶ max. Taktfrequenz durch Schaltnetz-Verzögerungen definiert



Elementare digitale Schaltungen

- ▶ Schaltsymbole
- ▶ Grundgatter (Inverter, AND, OR usw.)
- ▶ Kombinationen aus mehreren Gattern

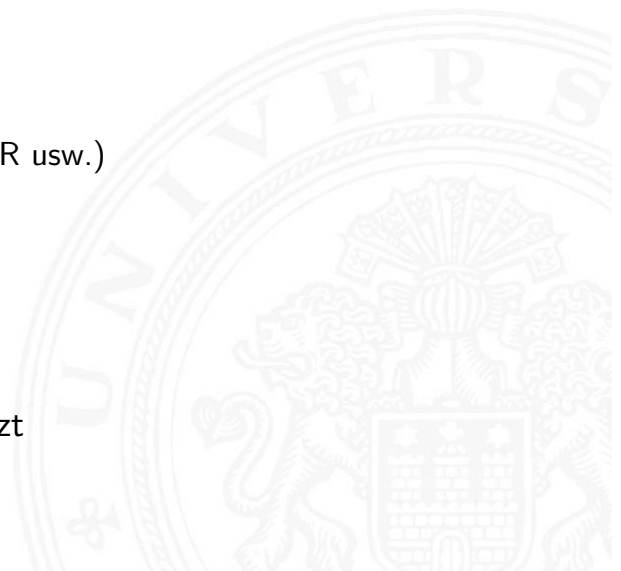
- ▶ Schaltnetze (mehrere Ausgänge)
- ▶ Beispiele

- ▶ 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





DIN 40700 (ab 1976)	Schaltzeichen		Benennung
	Früher	in USA	
			UND - Glied (AND)
			ODER - Glied (OR)
			NICHT - Glied (NOT)
			Exklusiv-Oder - Glied (Exclusive-OR, XOR)
			Aquivalenz - Glied (Logic identity)
			UND - Glied mit negier- tem Ausgang (NAND)
			ODER - Glied mit negier- tem Ausgang (NOR)
			Negation eines Eingangs
			Negation eines Ausgangs

Schiffmann, Schmitz: Technische Informatik 1 [SS04]



- ▶ **Logisches Gatter** (*logic gate*): realisiert eine logische Grundfunktion als einzelne (elektro-) technische Komponente
- ▶ 1 Eingang: Treiberstufe (Verstärker) und Inverter (Negation)
- ▶ 2 Eingänge: AND, OR, XOR, NAND, NOR, XNOR
- ▶ 3 und mehr Eingänge: AND, OR, NAND, NOR, Volladdierer, Parität
- ▶ Multiplexer
- ▶ vollständige Basismenge erforderlich (mindestens 1 Gatter)
- ▶ in Halbleitertechnologie sind NAND/NOR besonders effizient



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-lazaro.github.io/chipmunk

[Laz]

▶ Hades: tams.informatik.uni-hamburg.de/applets/hades/webdemos
tams.informatik.uni-hamburg.de/applets/hades/webdemos/toc.html

[HenHA]

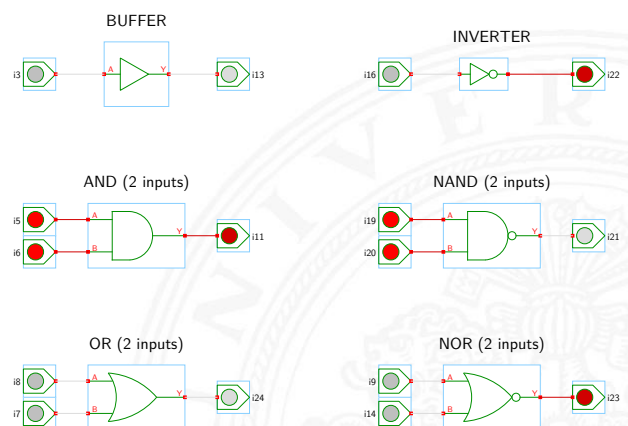
- ▶ Funktion: Schaltplaneingabe, Simulation, Impulsdiagramme ... als Java-Archiv
Webstart Demos im Browser
- ▶ Grundsaltungen, Gate-Level Circuits ...
einfache Prozessoren ...



▶ 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



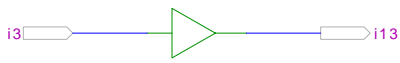
Hades: Bedienung

- ▶ Menü: Anzeigoptionen, 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

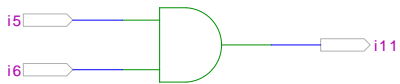
BUFFER



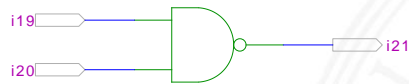
INVERTER



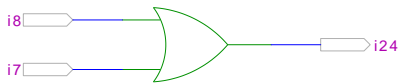
AND (2 inputs)



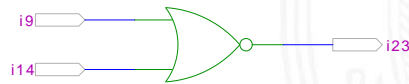
NAND (2 inputs)



OR (2 inputs)

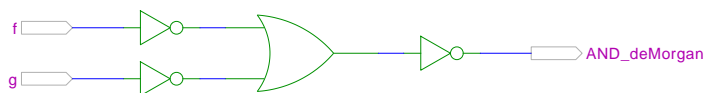


NOR (2 inputs)

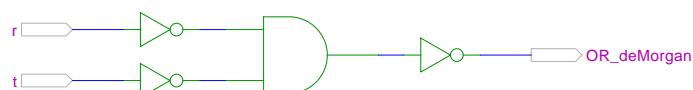


Grundschaltungen: De Morgan Regel

AND (2 inputs)



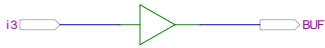
OR (2 inputs)



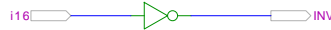


Gatter: AND/NAND mit zwei, drei, vier Eingängen

BUFFER



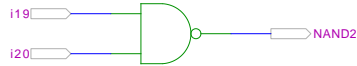
INVERTER



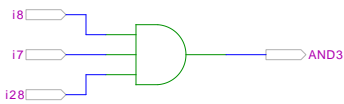
AND (2 inputs)



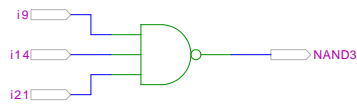
NAND (2 inputs)



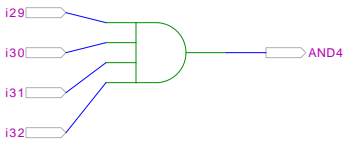
AND (3 inputs)



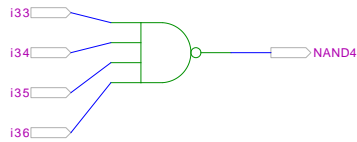
NAND (3 inputs)



AND (4 inputs)



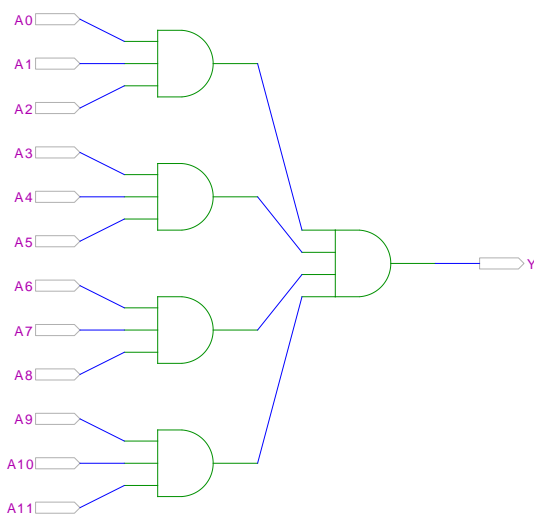
NAND (4 inputs)



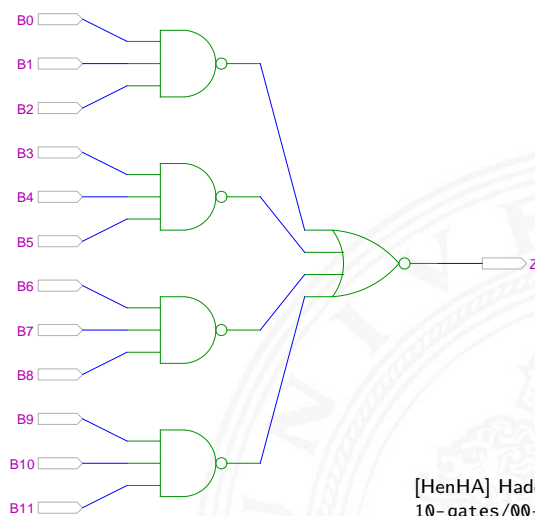
[HenHA] Hades Demo: 10-gates/00-gates/and



Gatter: AND mit zwölf Eingängen



AND3-AND4



NAND3-NOR4 (De Morgan)

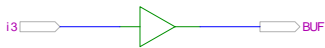
[HenHA] Hades Demo: 10-gates/00-gates/andbig

- ▶ in der Regel max. 4 Eingänge pro Gatter
- Grund: elektrotechnische Nachteile

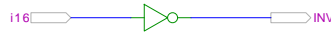


Gatter: OR/NOR mit zwei, drei, vier Eingängen

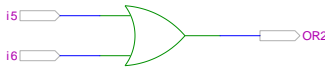
BUFFER



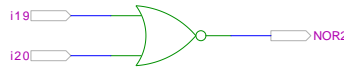
INVERTER



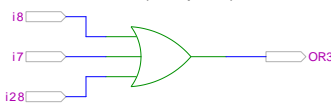
OR (2 inputs)



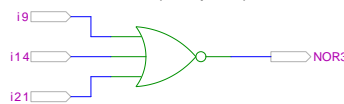
NOR (2 inputs)



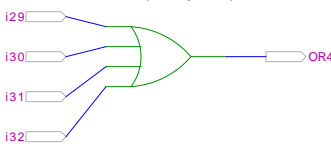
OR (3 inputs)



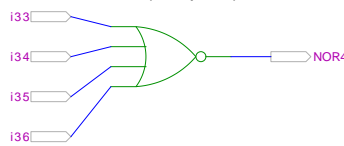
NOR (3 inputs)



OR (4 inputs)



NOR (4 inputs)

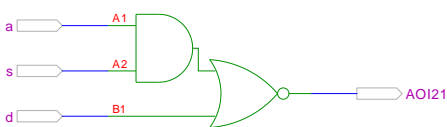


[HenHA] Hades Demo: 10-gates/00-gates/or

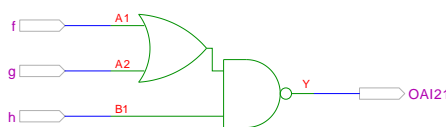


Komplexgatter

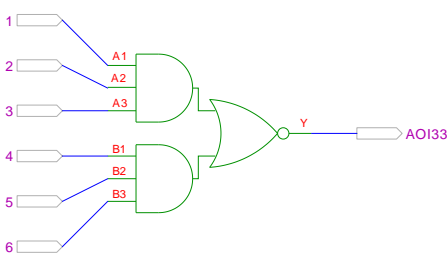
AOI21 (And-Or-Invert)



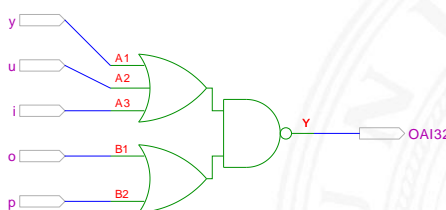
OAI21 (Or-And-Invert)



AOI33 (And-Or-Invert)



OAI32 (Or-And-Invert)



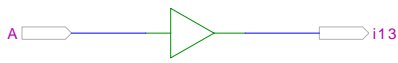
[HenHA] Hades Demo: 10-gates/00-gates/complex

- ▶ in CMOS-Technologie besonders günstig realisierbar
- ⇒ elektrotechnisch nur ein einziges Gatter

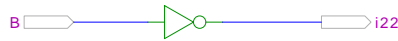


Gatter: XOR und XNOR

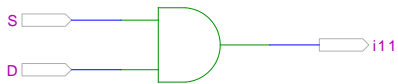
BUFFER



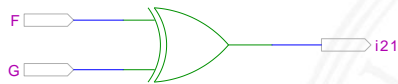
INVERTER



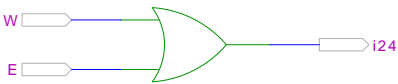
AND (2 inputs)



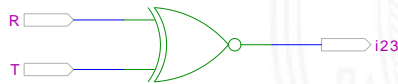
XOR (2 inputs)



OR (2 inputs)



XNOR (2 inputs)

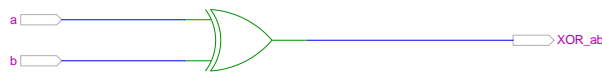


[HenHA] Hades Demo: 10-gates/00-gates/xor

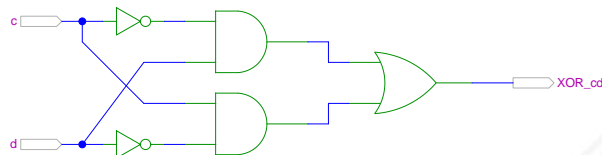


XOR und drei Varianten der Realisierung

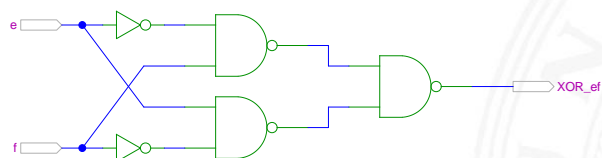
► Symbol



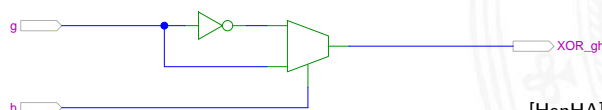
► AND-OR



► NAND-NAND



► mit Multiplexer



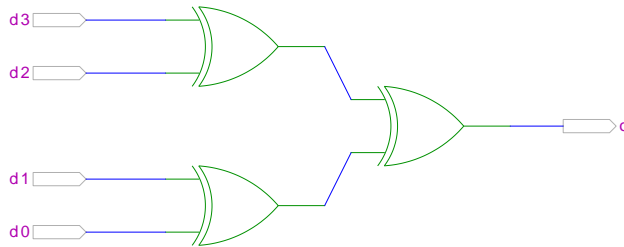
[HenHA] Hades Demo: 10-gates/11-xor/xor-variants



XOR zur Berechnung der Parität

► Parität, siehe „Codierung – Fehlererkennende Codes“

► 4-bit Parität: $d_3 \oplus d_2 \oplus d_1 \oplus d_0$

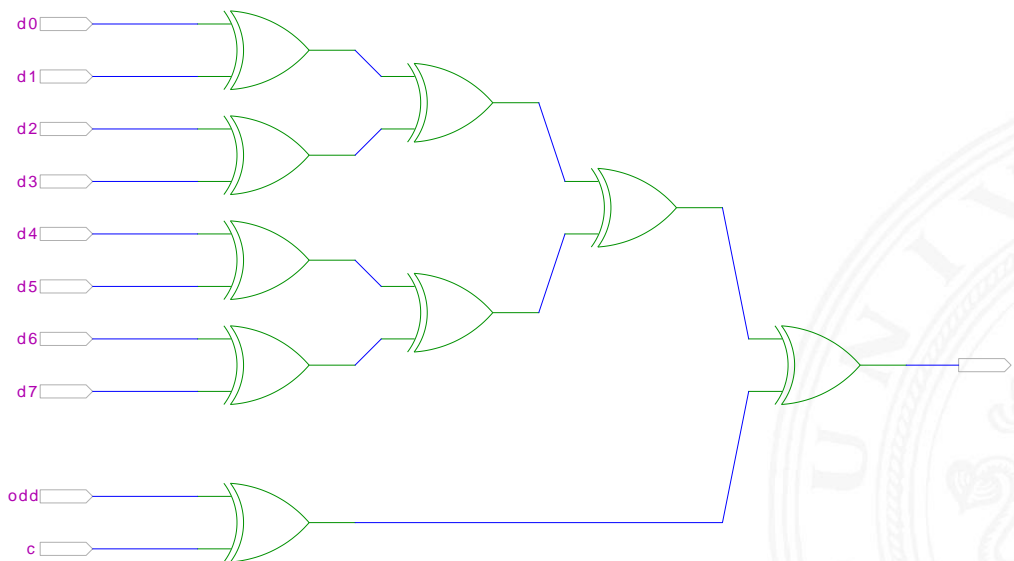


[HenHA] Hades Demo: 10-gates/12-parity/parity4



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



- ▶ Schaltungen mit mehreren Ausgängen
- ▶ Bündelminimierung der einzelnen Funktionen

Standard-Schaltnetze

- ▶ Multiplexer, Demultiplexer
- ▶ Prioritätsencoder

ausgewählte typische Beispiele

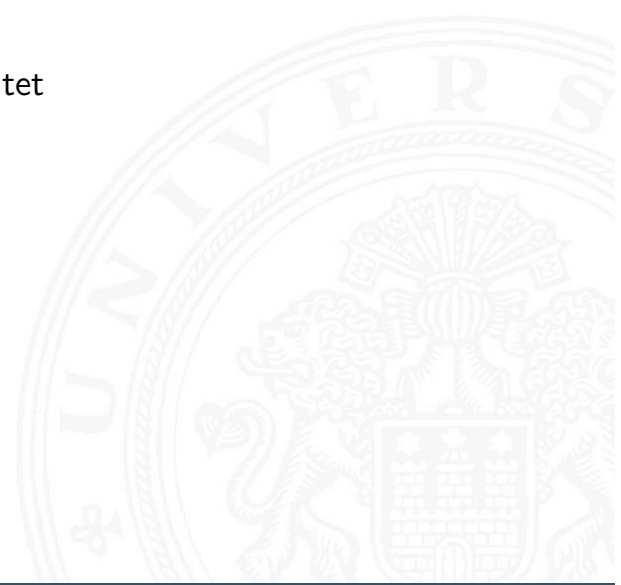
- ▶ „Würfel“-Decoder
- ▶ Umwandlung vom Dual-Code in den Gray-Code
- ▶ (7,4)-Hamming-Code: Encoder und Decoder
- ▶ Siebensegmentanzeige



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

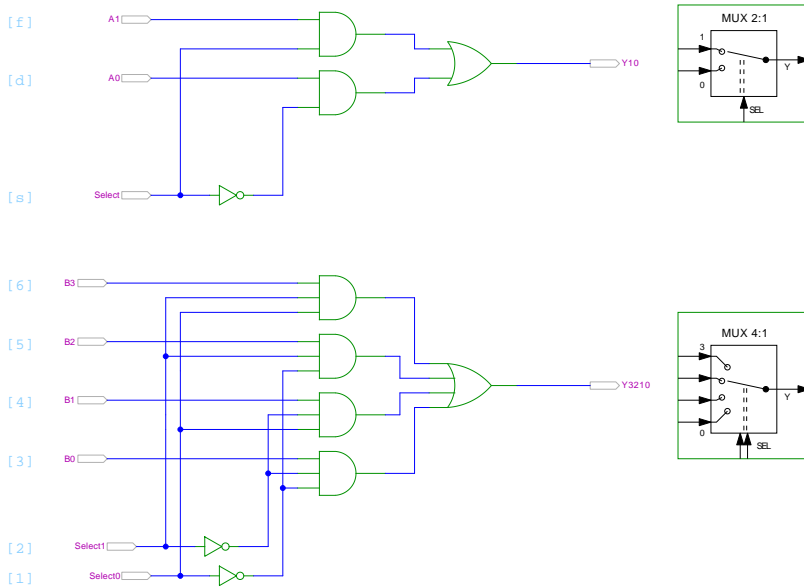
- ▶ 4:1-Multiplexer

s ₁	s ₀	a ₃	a ₂	a ₁	a ₀	y
0	0	*	*	*	a ₀	a ₀
0	1	*	*	a ₁	*	a ₁
1	0	*	a ₂	*	*	a ₂
1	1	a ₃	*	*	*	a ₃

- ▶ $\lceil \log_2(n) \rceil$ Steuereingänge: s_m, \dots, s_0
 n Dateneingänge: a_{n-1}, \dots, a_0
ein Datenausgang: y



2:1 und 4:1 Multiplexer

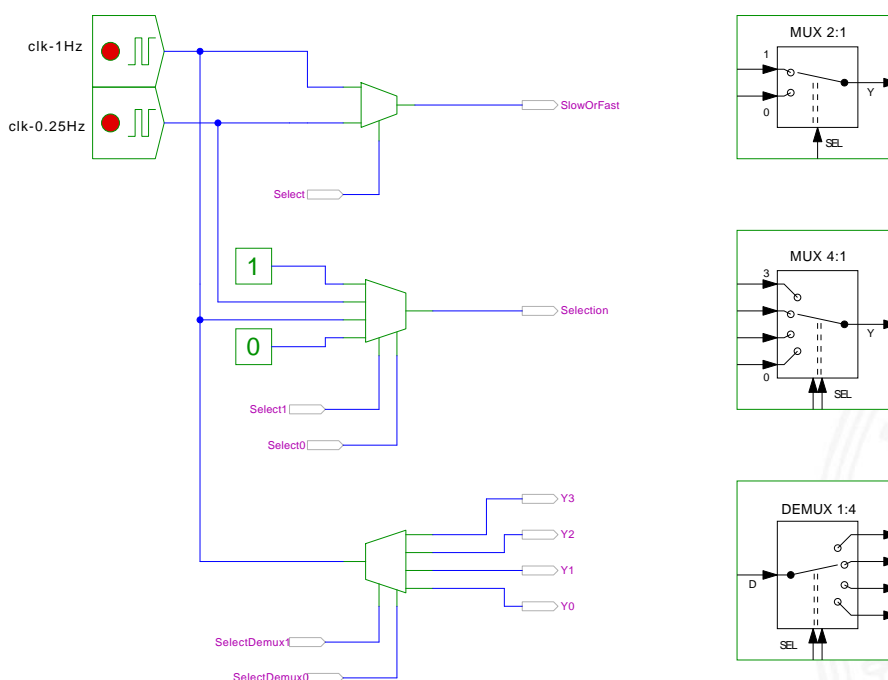


[HenHA] Hades Demo: 10-gates/40-mux-demux/mux21-mux41

- ▶ keine einheitliche Anordnung der Dateneingänge in Schaltplänen: höchstwertiger Eingang manchmal oben, manchmal unten



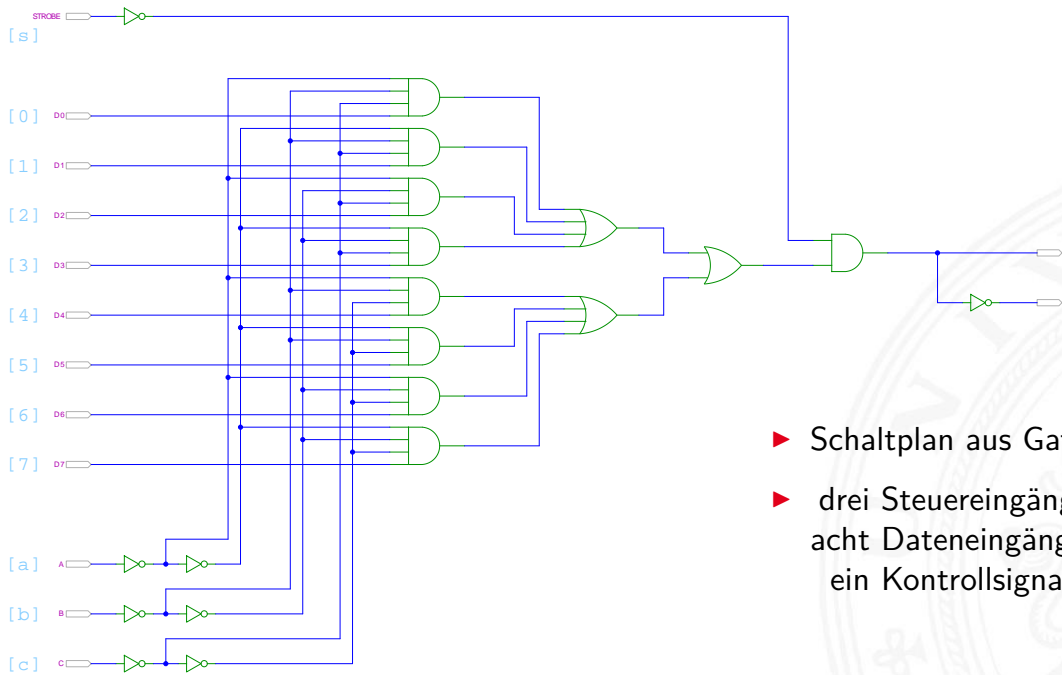
Multiplexer und Demultiplexer



[HenHA] Hades Demo: 10-gates/40-mux-demux/mux-demux



8-bit Multiplexer: Integrierte Schaltung 74151

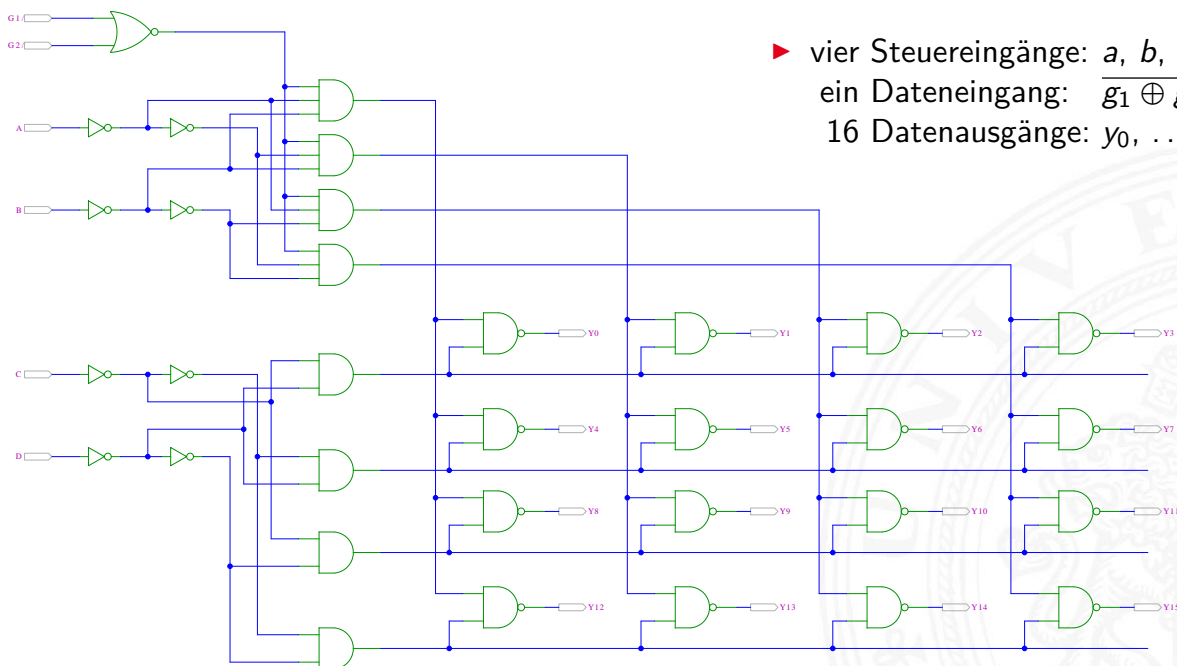


- ▶ Schaltplan aus Gattern
- ▶ drei Steuereingänge: a, b, c
acht Dateneingänge: d_7, \dots, d_0
ein Kontrollsignal (Maskierung)

[HenHA] Hades Demo: 10-gates/40-mux-demux/SN74151



16-bit Demultiplexer: Integrierte Schaltung 74154

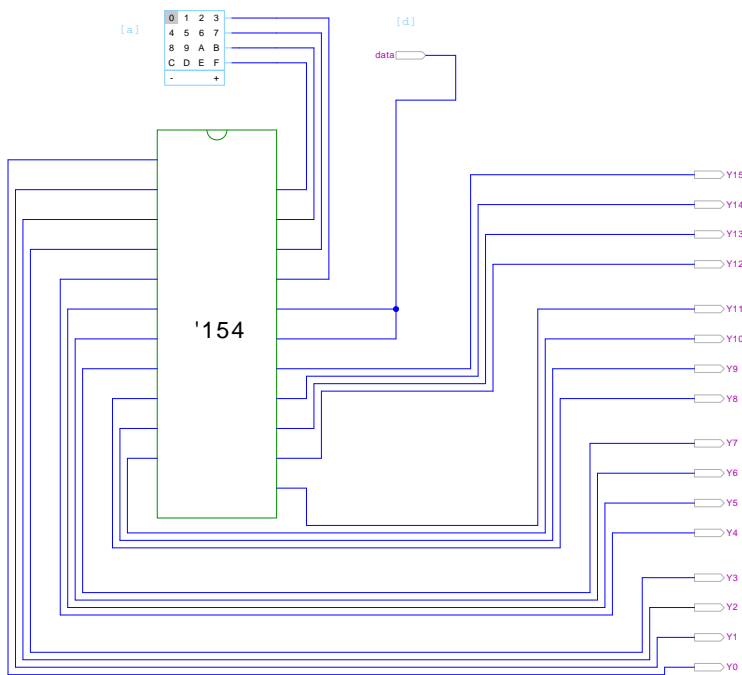


- ▶ vier Steuereingänge: a, b, c, d
ein Dateneingang: $g_1 \oplus g_2$
16 Datenausgänge: y_0, \dots, y_{15}

[HenHA] Hades Demo: 10-gates/40-mux-demux/SN74154



16-bit Demultiplexer: 74154 als Adresdecoder



[HenHA] Hades Demo: 10-gates/40-mux-demux/demo74154



Priority Encoder

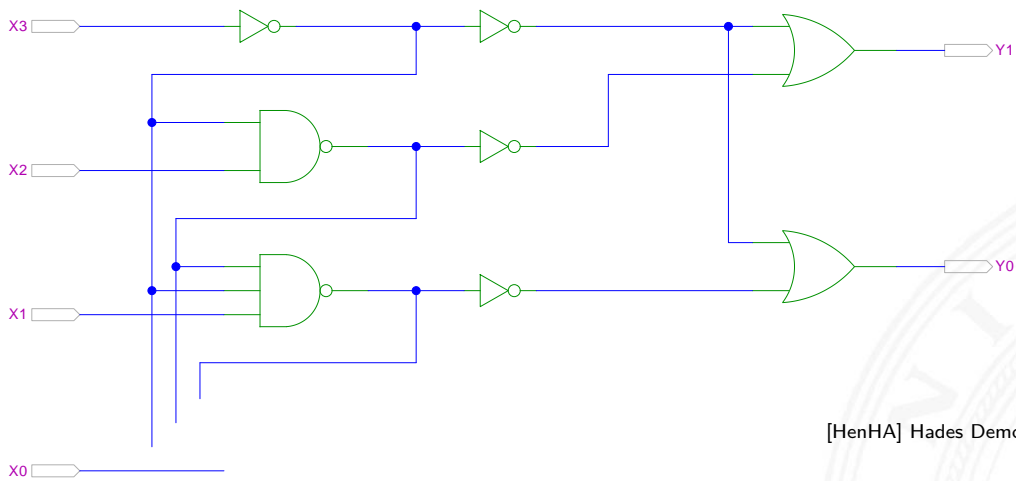
- ▶ Anwendung u.a. für Interrupt-Priorisierung
- ▶ Schaltung konvertiert n -bit Eingabe in eine Dualcodierung
- ▶ Wenn Bit n aktiv ist, werden alle niedrigeren Bits ($n - 1$), \dots , 0 ignoriert

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

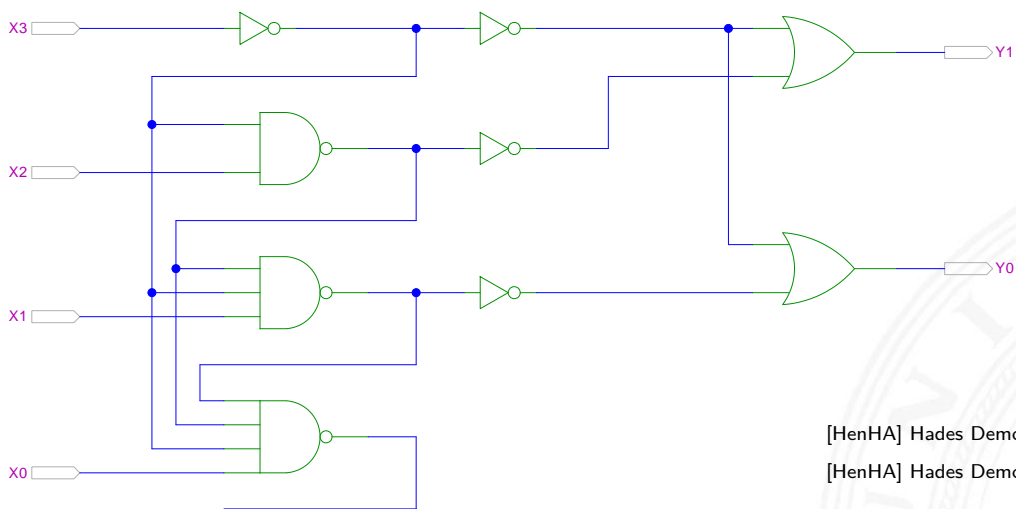


[HenHA] Hades Demo: 10-gates/45-priority/priority42

- ▶ zweistufige Realisierung (Inverter ignoriert)
- ▶ aktive höhere Stufe blockiert alle niedrigeren Stufen



4:2 Prioritätsencoder



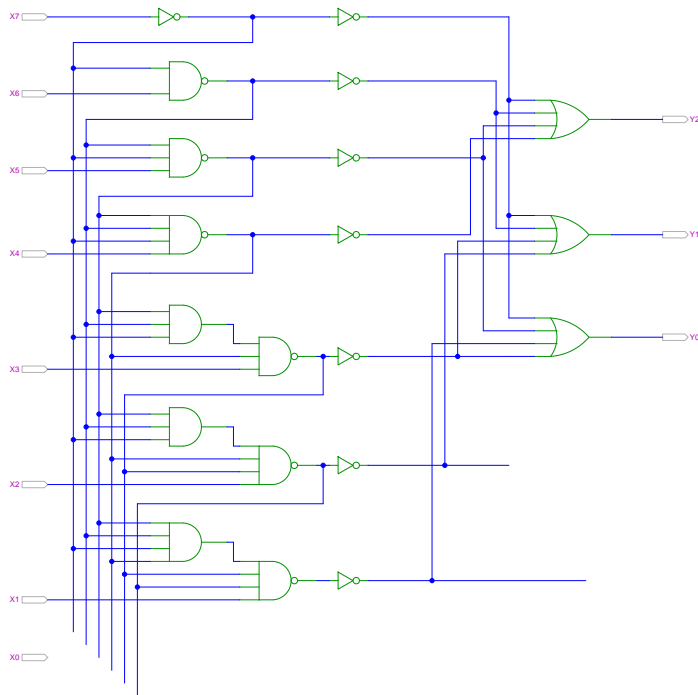
[HenHA] Hades Demo: 10-gates/45-priority/priority42

[HenHA] Hades Demo: 10-gates/45-priority/priority53

- ▶ zweistufige Realisierung (Inverter ignoriert)
- ▶ aktive höhere Stufe blockiert alle niedrigeren Stufen
- ▶ Kaskadierung



8:3 Prioritätsencoder

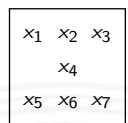
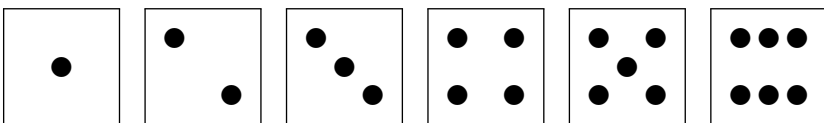


[HenHA] Hades Demo: 10-gates/45-priority/priority83



Beispiel: „Würfel“-Decoder

Visualisierung eines Würfels mit sieben LEDs

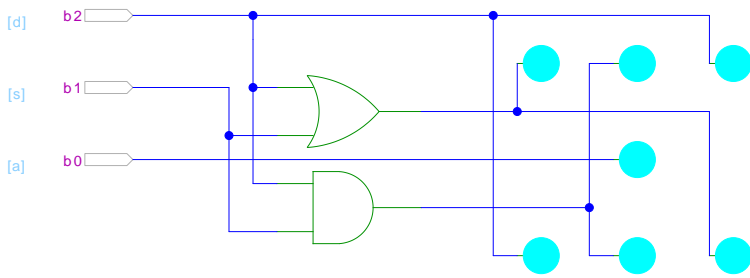


- ▶ Eingabewert von 0...6
- ▶ Anzeige ein bis sechs Augen: eingeschaltet

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.)

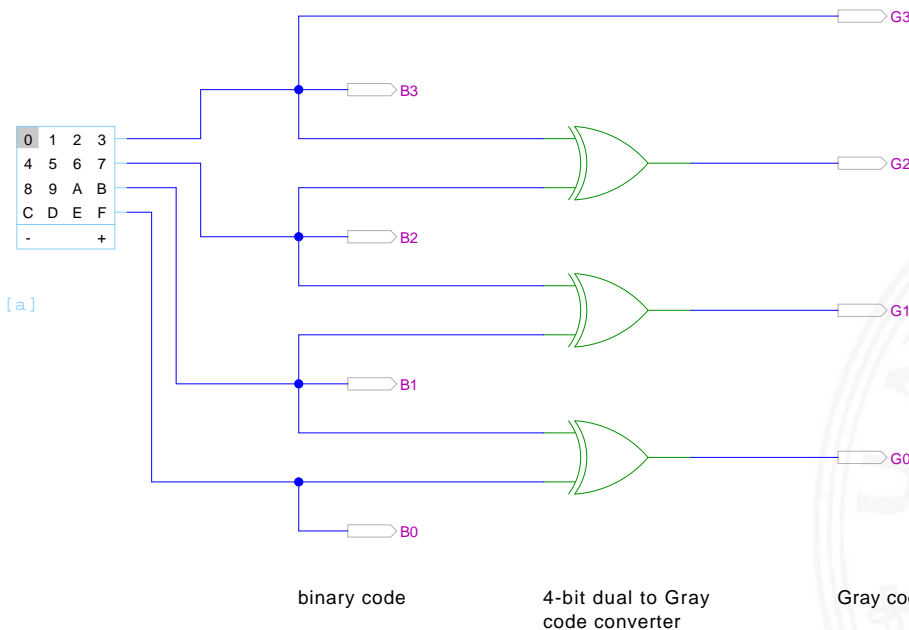


[HenHA] Hades Demo: 10-gates/10-wuerfel/wuerfel

- ▶ 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



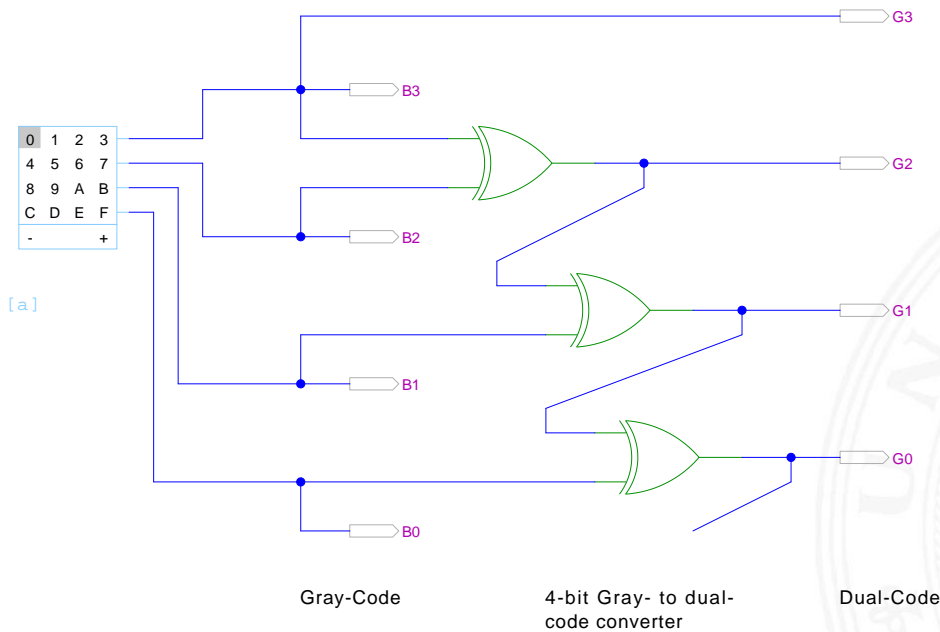
[HenHA] Hades Demo: 10-gates/15-graycode/dual2gray



Beispiel: Umwandlung vom Graycode in den Dualcode XOR-Kette

9.5.3 Schaltnetze - Einfache Schaltnetze - Beispiele

64-040 Rechnerstrukturen und Betriebssysteme



[HenHA] Hades Demo: 10-gates/15-graycode/gray2dual

A. Mäder

547



(7,4)-Hamming-Code: Encoder und Decoder

9.5.3 Schaltnetze - Einfache Schaltnetze - Beispiele

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Encoder
 - ▶ vier Eingabebits
 - ▶ Hamming-Encoder erzeugt drei Paritätsbits
- ▶ Übertragungskanal
 - ▶ Übertragung von sieben Codebits
 - ▶ Einfügen von Übertragungsfehlern durch Invertieren von Codebits mit XOR-Gattern
- ▶ Decoder und Fehlerkorrektur
 - ▶ Decoder liest die empfangenen sieben Bits
 - ▶ Syndrom-Berechnung mit XOR-Gattern und Anzeige erkannter Fehler
 - ▶ Korrektur gekippter Bits

linke Seite

Mitte

rechte Seite

rechts oben

A. Mäder

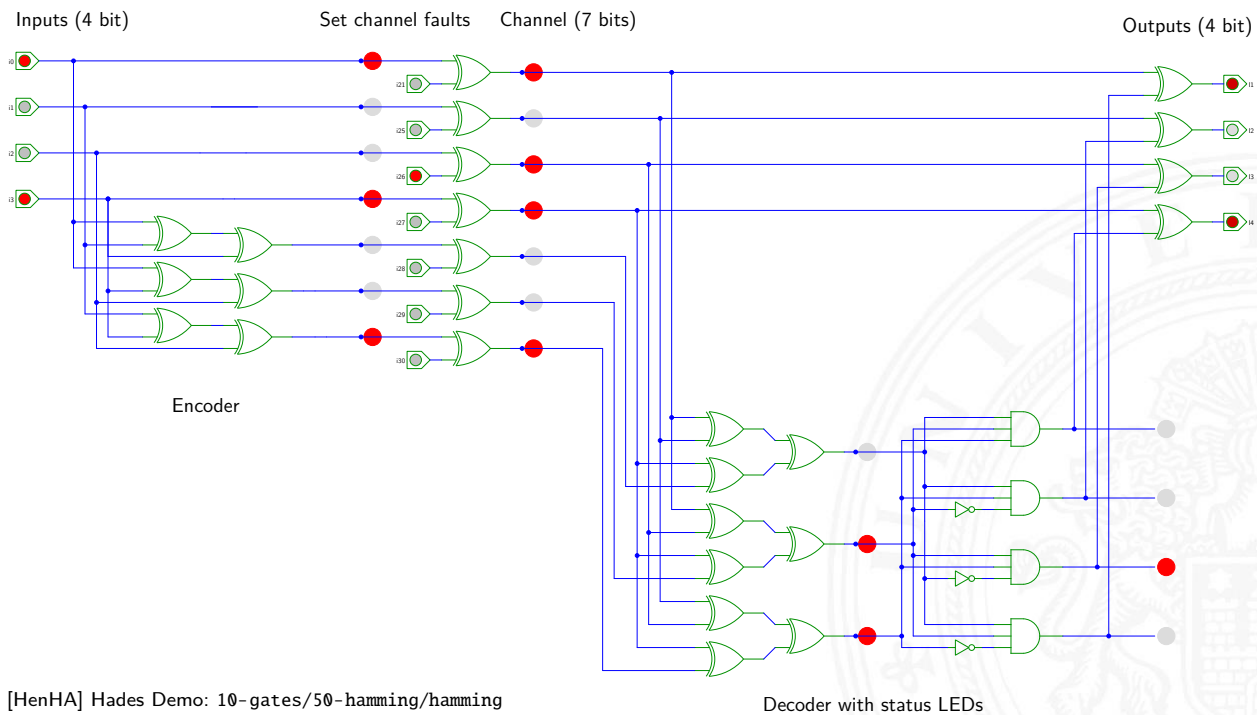
548



(7,4)-Hamming-Code: Encoder und Decoder (cont.)

9.5.3 Schaltnetze - Einfache Schaltnetze - Beispiele

64-040 Rechnerstrukturen und Betriebssysteme



[HenHA] Hades Demo: 10-gates/50-hamming/hamming

A. Mäder

549

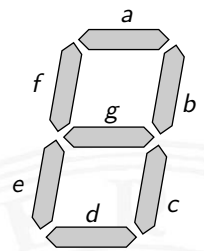


mehrere Ausgänge: Siebensegmentanzeige

9.5.3 Schaltnetze - Einfache Schaltnetze - Beispiele

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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.



A. Mäder

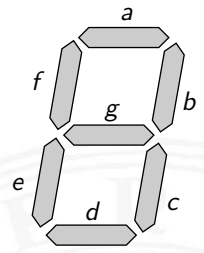
550



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	0	0	1	1	1	1	0	0	1	0	0	
c	1	1	0	1	1	1	1	1	1	1	1	0	1	0	0	
d	1	0	1	1	0	1	1	0	1	1	1	1	1	1	0	
e	1	0	1	0	0	0	1	0	1	0	1	1	1	1	1	
f	1	0	0	0	1	1	1	1	1	1	1	1	0	0	1	
g	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1	



- ▶ für Ziffernanzeige mit *Don't Care*-Termen

$a = 1011011111*****$
 $b = \text{usw.}$



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]

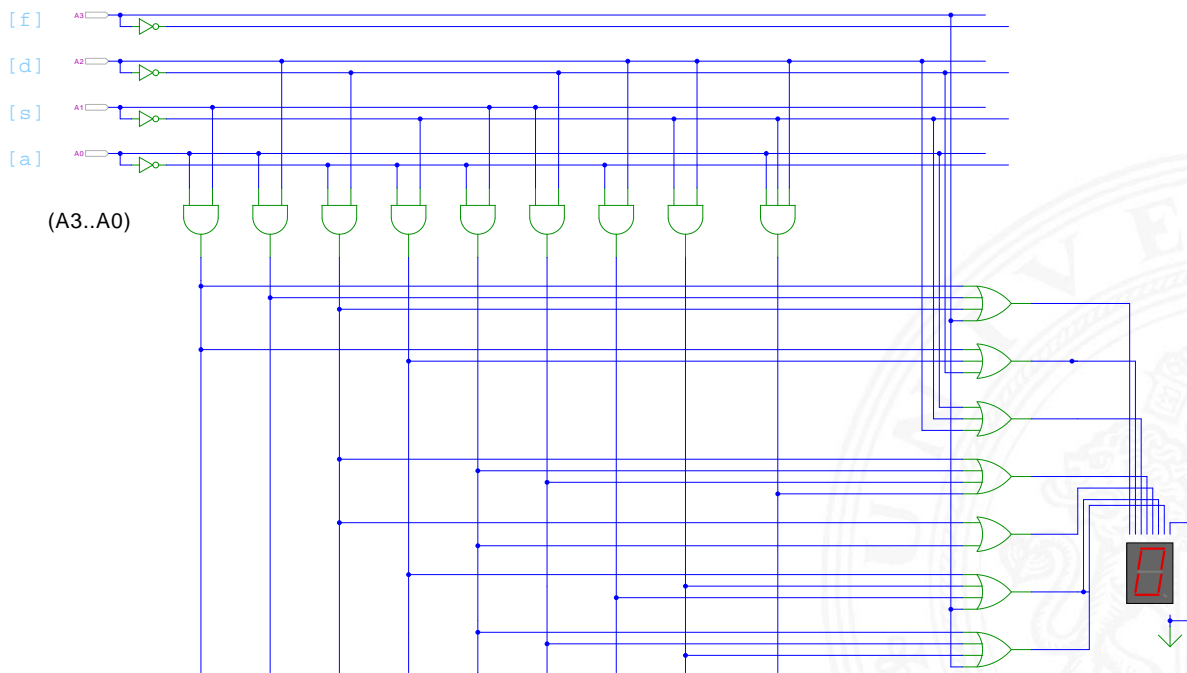
- ▶ als mehrstufige Schaltung ist günstigere Lösung möglich

Knuth: *AoCP, Volume 4, Fascicle 0, 7.1.2, Seite 112ff*

[Knu08]



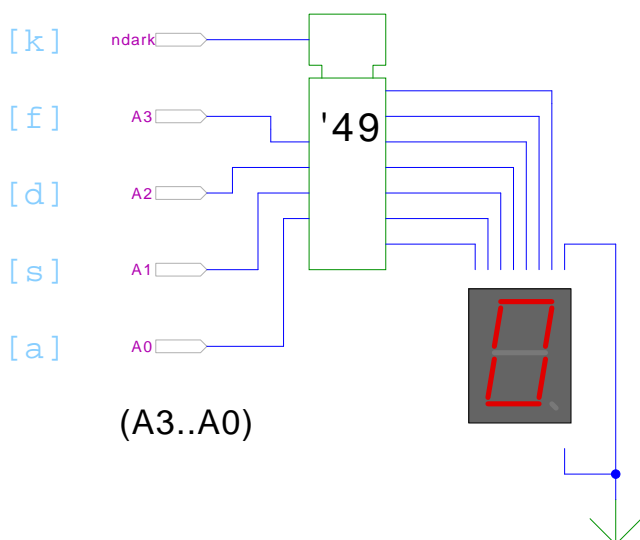
Siebensegmentdecoder: Ziffern 0...9



[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, \dots, g

$x_5 = x_2 \oplus x_3$	$x_{13} = x_1 \oplus x_7$	$\bar{a} = x_{20} = x_{14} \wedge \overline{x_{19}}$
$x_6 = \overline{x_1} \wedge x_4$	$x_{14} = x_5 \oplus x_6$	$\bar{b} = x_{21} = x_7 \oplus x_{12}$
$x_7 = x_3 \wedge \overline{x_6}$	$x_{15} = x_7 \vee x_{12}$	$\bar{c} = x_{22} = \overline{x_8} \wedge x_{15}$
$x_8 = x_1 \oplus x_2$	$x_{16} = x_1 \vee x_5$	$\bar{d} = x_{23} = x_9 \wedge \overline{x_{13}}$
$x_9 = x_4 \oplus x_5$	$x_{17} = x_5 \vee x_6$	$\bar{e} = x_{24} = x_6 \vee x_{18}$
$x_{10} = \overline{x_7} \wedge x_8$	$x_{18} = x_9 \wedge x_{10}$	$\bar{f} = x_{25} = \overline{x_8} \wedge x_{17}$
$x_{11} = x_9 \oplus x_{10}$	$x_{19} = x_3 \wedge x_9$	$g = x_{26} = x_7 \vee x_{16}$
$x_{12} = x_5 \wedge x_{11}$		

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*) aus 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*) aus 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

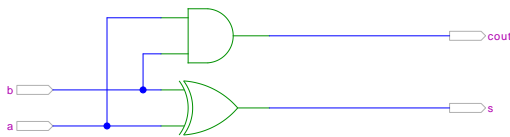
$$c_o = ab \vee ac_i \vee bc_i = (ab) \vee (a \vee b)c_i$$

$$s = a \oplus b \oplus c_i$$

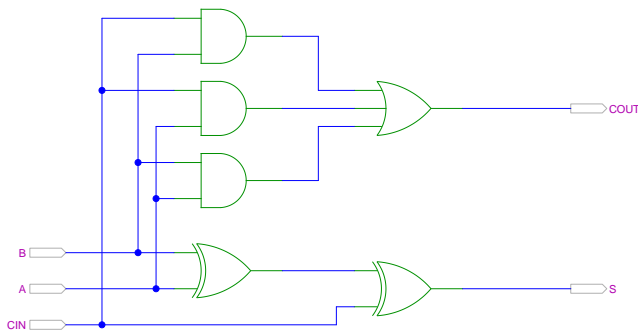


Schaltbilder für Halb- und Volladdierer

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: $s_n = a_n \oplus b_n \oplus c_n$

$$s_0 = a_0 \oplus b_0$$

$$s_1 = a_1 \oplus b_1 \oplus c_1$$

$$s_2 = a_2 \oplus b_2 \oplus c_2$$

...

$$s_n = a_n \oplus b_n \oplus c_n$$

► Übertrag: $c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$

$$c_1 = (a_0 b_0)$$

$$c_2 = (a_1 b_1) \vee (a_1 \vee b_1) c_1$$

$$c_3 = (a_2 b_2) \vee (a_2 \vee b_2) c_2$$

...

$$c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$$



n-bit Addierer (cont.)

9.6.1 Schaltnetze - ALU Schaltnetze - Addierer

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ *n*-bit Addierer theoretisch als zweistufige Schaltung realisierbar
 - ▶ direkte und negierte Eingänge, dann AND-OR Netzwerk
 - ▶ Aufwand steigt exponentiell mit *n* an,
für Ausgangsbit *n* sind $2^{(2n-1)}$ Minterme erforderlich
- ⇒ nicht praktikabel
-
- ▶ Problem: Übertrag (*carry*) ist rekursiv definiert als
$$c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$$



n-bit Addierer (cont.)

9.6.1 Schaltnetze - ALU Schaltnetze - Addierer

64-040 Rechnerstrukturen und Betriebssysteme

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

9.6.1 Schaltnetze - ALU Schaltnetze - Addierer

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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: $c_o(n) \neq c_o(n - 1)$

A. Mäder

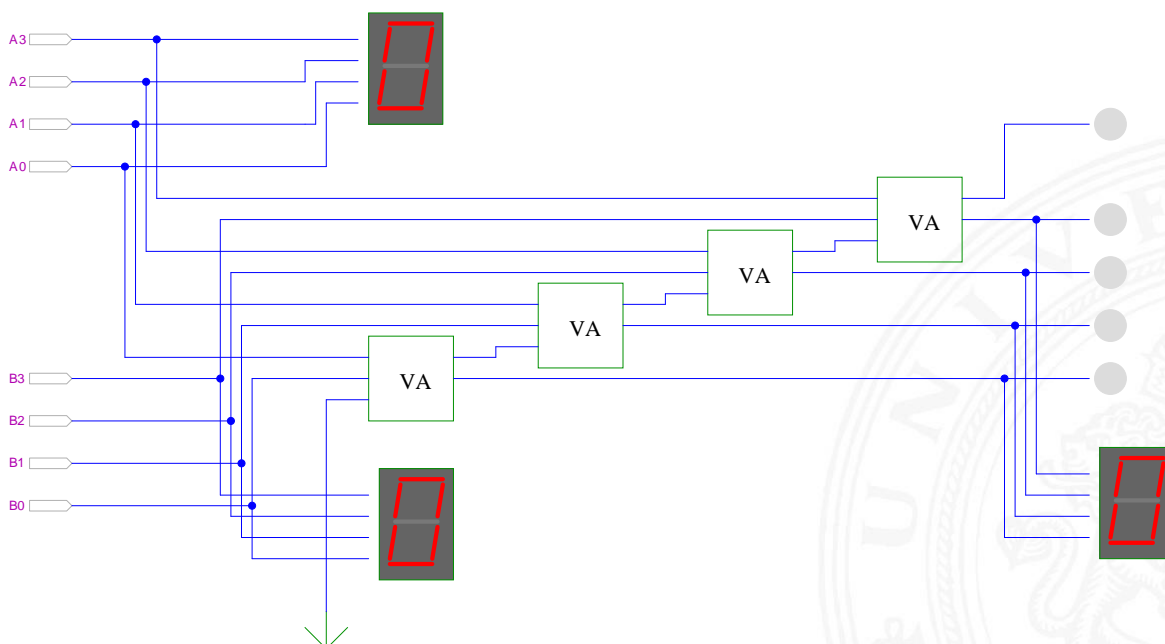
563



Ripple-Carry Adder: 4-bit

9.6.1 Schaltnetze - ALU Schaltnetze - Addierer

64-040 Rechnerstrukturen und Betriebssysteme



Schiffmann, Schmitz: *Technische Informatik I* [SS04]

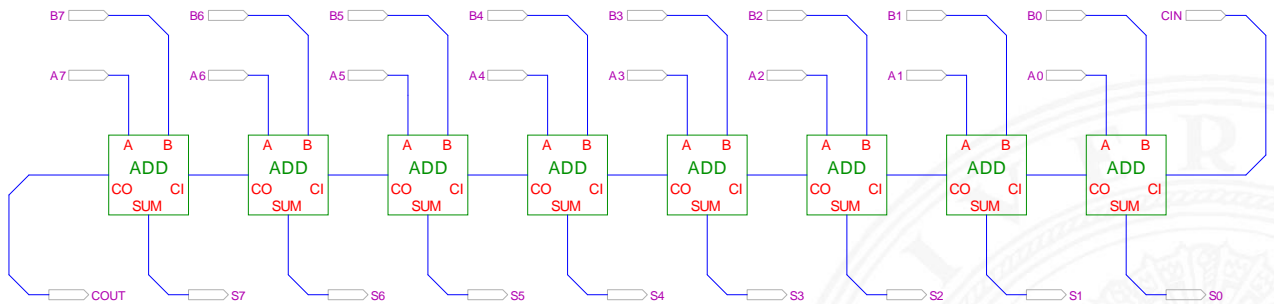
A. Mäder

564



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

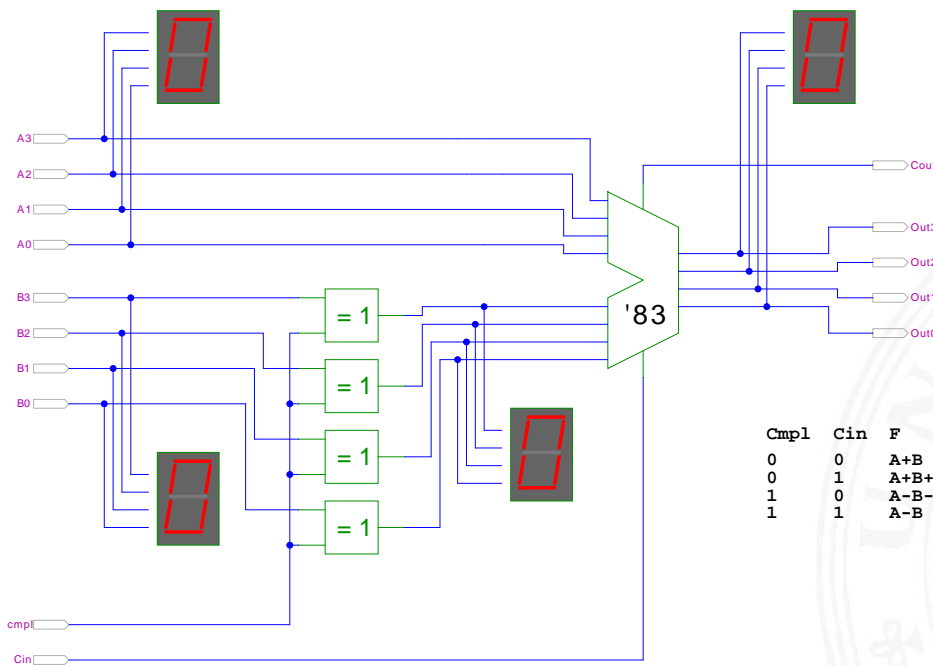
- ▶ $(A - B)$ ersetzt durch Addition des 2-Komplements von B
- ▶ 2-Komplement: Invertieren aller Bits und Addition von Eins
- ▶ Carry-in Eingang des niederwertigsten Bits bisher nicht benutzt

Subtraktion quasi „gratis“ realisierbar

- ▶ LSB: Volladdierer (mit c_i) statt Halbaddierer verwenden
- ▶ Invertieren der Bits von B (1-Komplement)
- ▶ Carry-in Eingang auf 1 setzen (Addition von 1)
- ▶ Resultat ist $A + \bar{B} + 1 = A - B$



Subtrahierer: Beispiel 7483 – 4-bit Addierer



Schiffmann, Schmitz: Technische Informatik I [SS04]



Schnelle Addierer

- ▶ Addierer in Prozessoren häufig im *kritischen Pfad*
- ▶ möglichst hohe Performanz ist essenziell
- ⇒ 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
- ⇒ Verzögerung reduziert sich auf die Verzögerung eines m -bit Addierers plus die Verzögerungen der Multiplexer

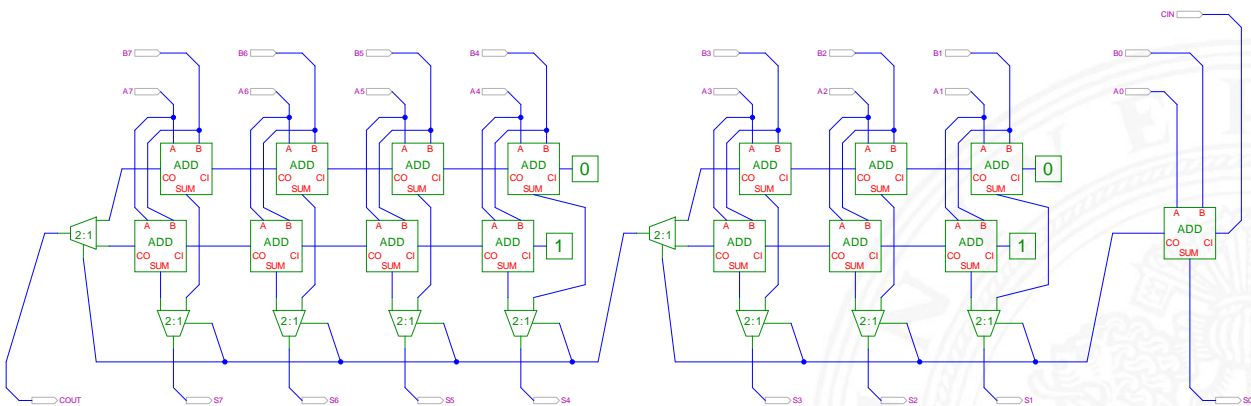


Carry-Select Adder: Beispiel

8-Bit Carry-Select Adder (4 + 3 + 1 bit blocks)

4-bit Carry-Select Adder block

3-bit Carry-Select Adder block

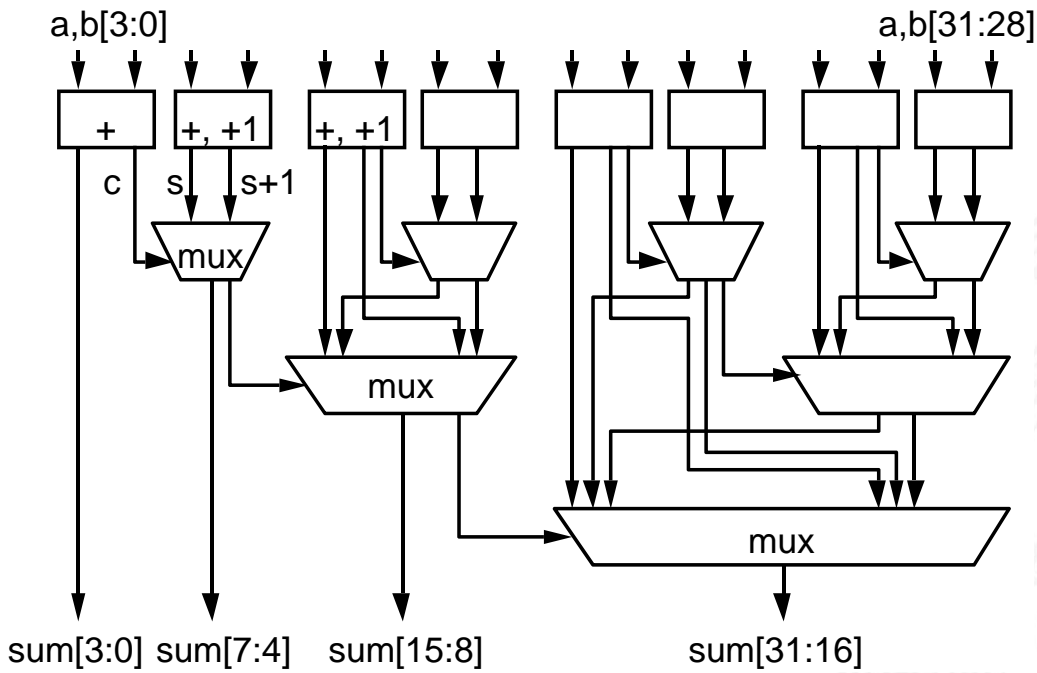


[HenHA] Hades Demo: 20-arithmetic/20-carryselect/adder_carryselect

- ▶ drei Gruppen: 1-bit, 3-bit, 4-bit
- ▶ Gruppengrößen so wählen, dass Gesamtverzögerung minimal wird



Carry-Select Adder: Beispiel ARM v6



Carry-Lookahead Adder: Prinzip

▶ $c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$

- ▶ Einführung von Hilfsfunktionen

$g_n = (a_n b_n)$

$p_n = (a_n \vee b_n)$

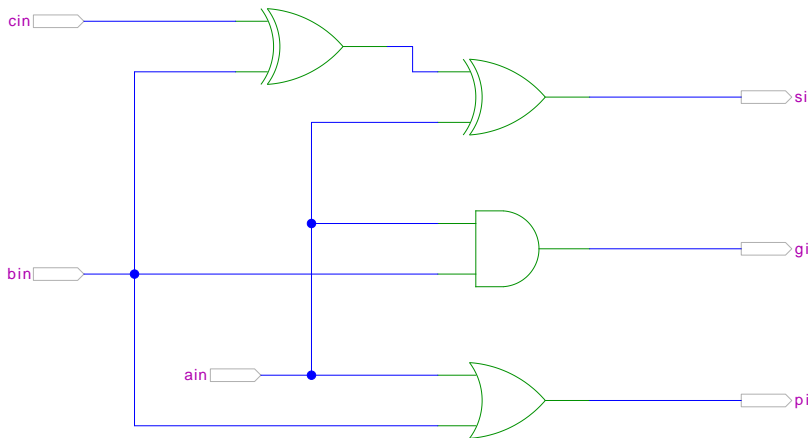
$c_{n+1} = g_n \vee p_n c_n$

„generate carry“
„propagate carry“

- ▶ *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

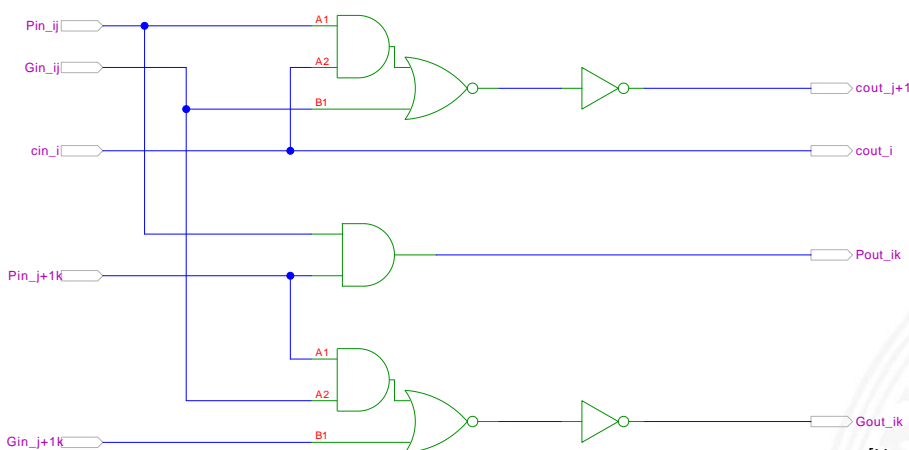


[HenHA] Hades Demo: 20-arithmetic/30-cla/sum

- ▶ 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

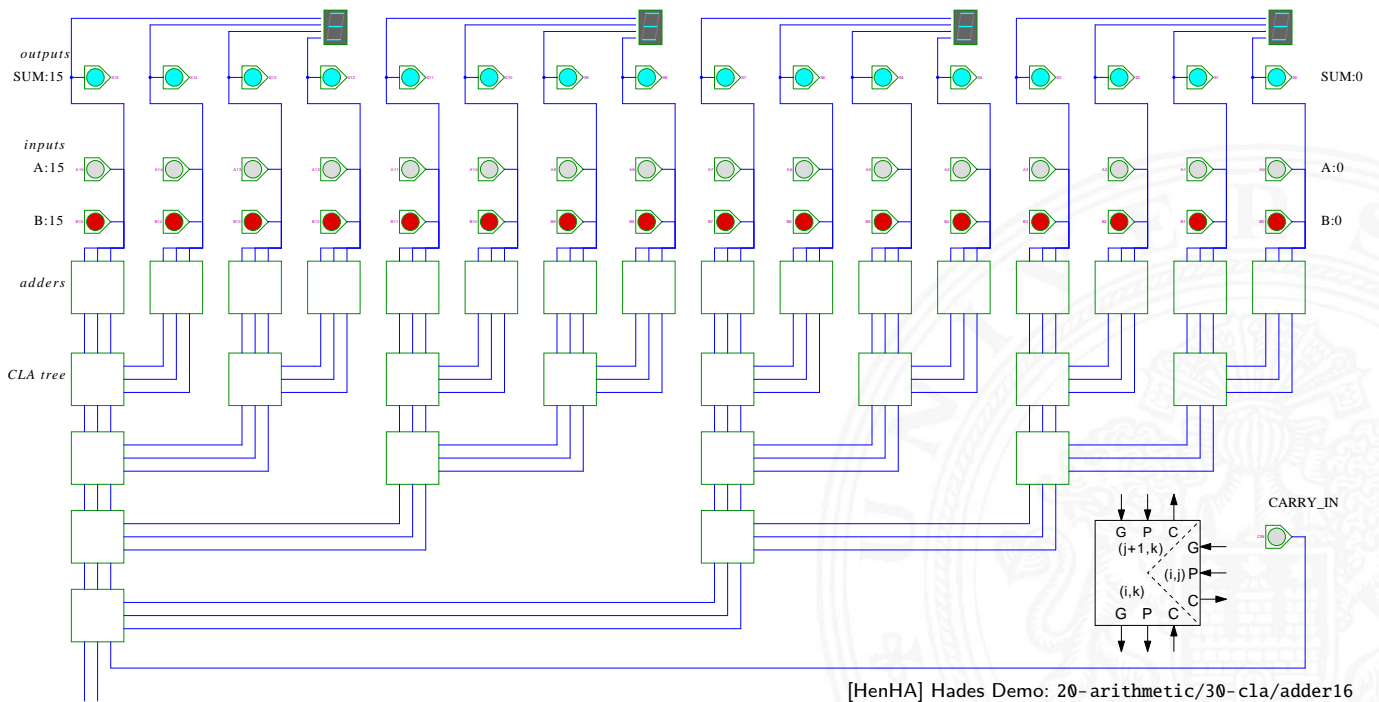


[HenHA] Hades Demo: 20-arithmetic/30-cla/cla

- ▶ 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

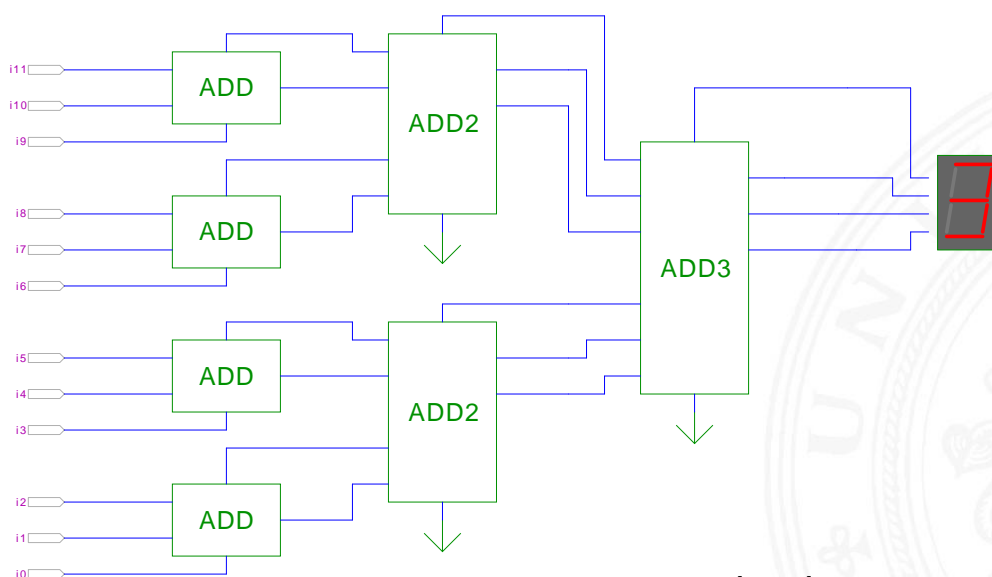


[HenHA] Hades Demo: 20-arithmetic/30-cla/adder16



Addition mehrerer Operanden

- ▶ Addierer-Bäume
- ▶ Beispiel: Bitcount



[HenHA] Hades Demo: 20-arithmetic/80-bitcount/bitcount



Addierer: Zusammenfassung

- ▶ Halbaddierer: $s = a \oplus b$ $c_o = a b$
- ▶ Volladdierer: $s = a \oplus b \oplus c_i$ $c_o = a b \vee a c_i \vee b c_i$

- ▶ 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--)



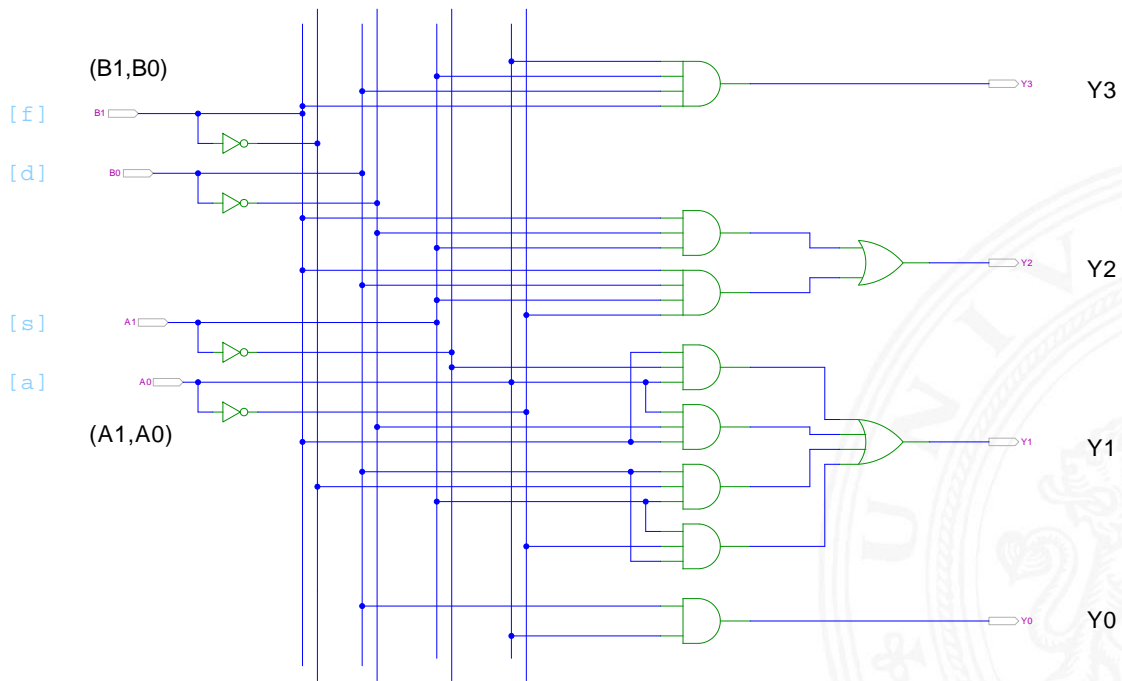
Multiplizierer

- ▶ 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



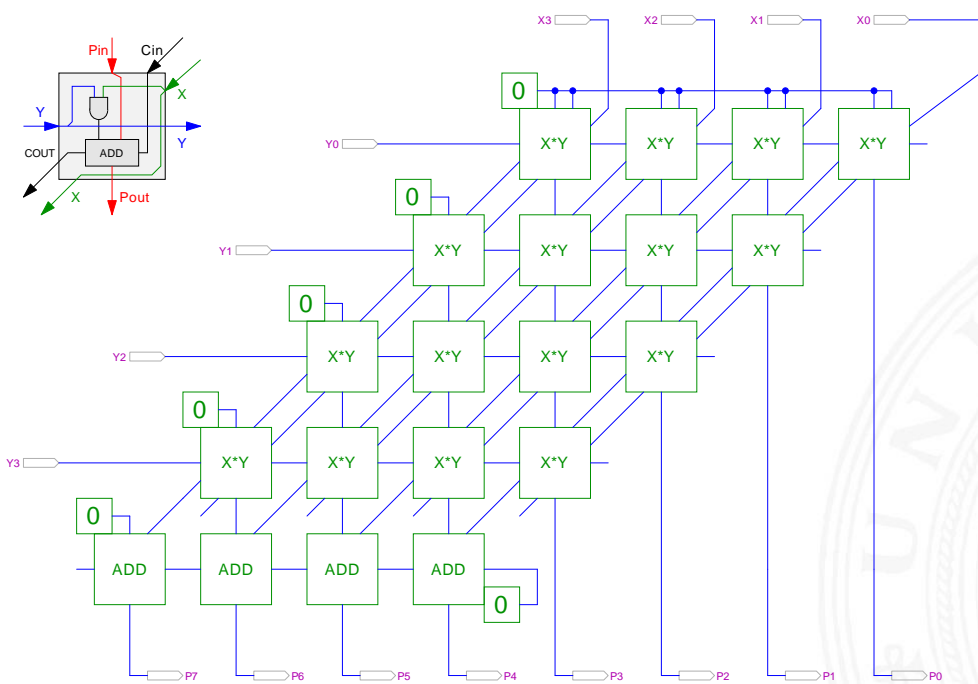
2x2-bit Multiplizierer – als zweistufiges Schaltnetz



[HenHA] Hades Demo: 10-gates/13-mult2x2/mult2x2



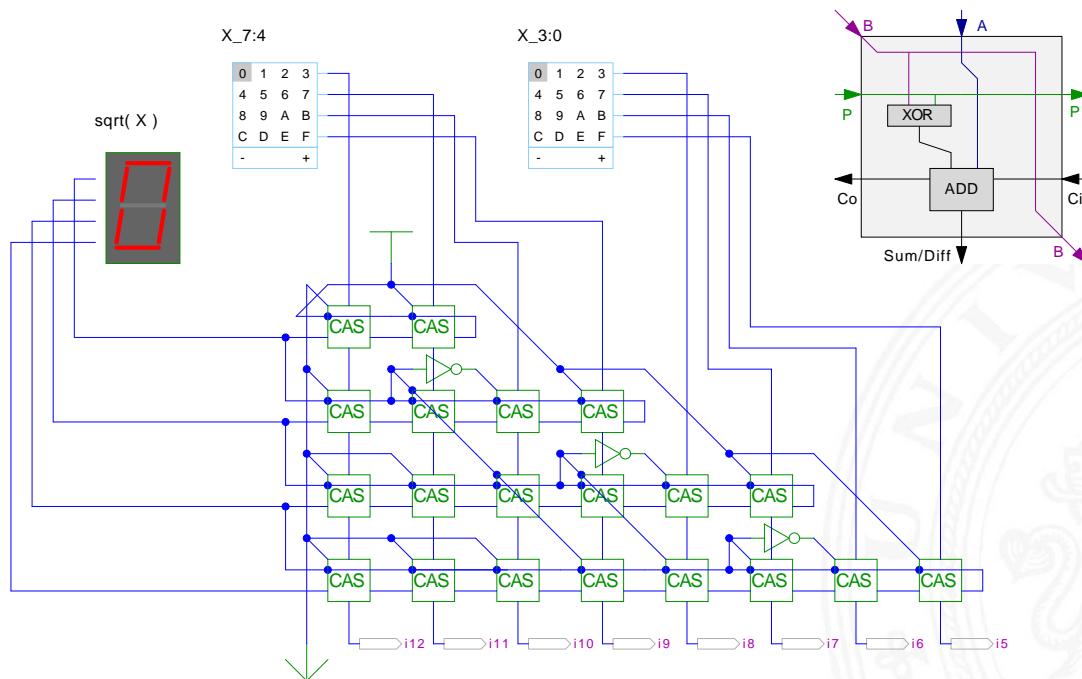
4x4-bit Multiplizierer – Array



[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

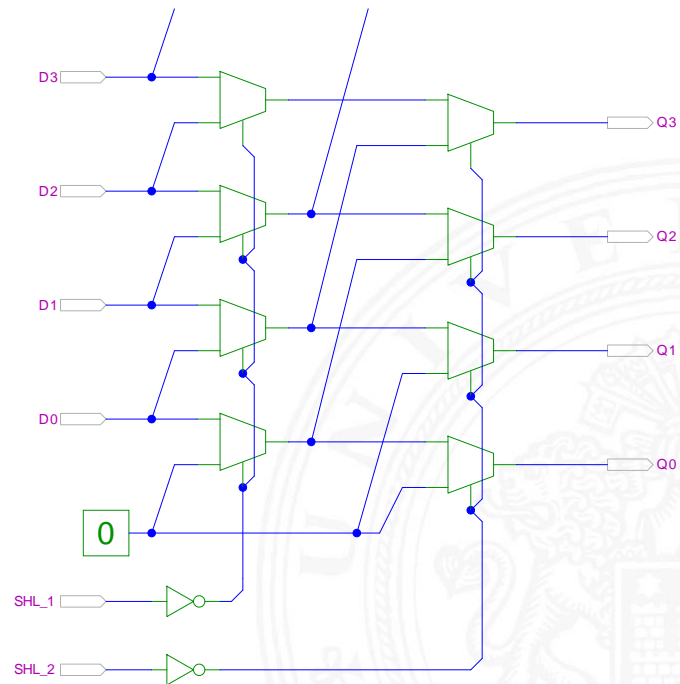
- ▶ *CORDIC-Algorithmen*
 - ▶ Multiplikation, Division
 - ▶ iterative Berechnung höherer Funktionen: Exponentialfunktion, Logarithmus, trigonometrische Funktionen

- ▶ bei Interesse: Literatur anschauen [Omo94, Kor01, Spa76]



Shifter: zweistufig, shift-left um 0...3 Bits

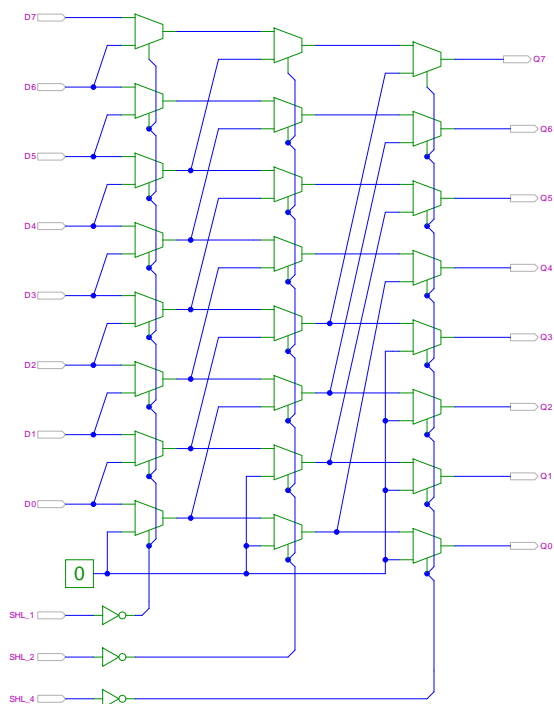
- ▶ n -Dateneingänge D_i
 n -Datenausgänge Q_i
- ▶ 2:1 Multiplexer Kaskade
 - ▶ Stufe 0: benachbarte Bits
 - ▶ Stufe 1: übernächste Bits
 - ▶ usw.
- ▶ von rechts 0 nachschieben



[HenHA] Hades Demo: 10-gates/60-barrel/barrel-shifter



8-bit Barrel-Shifter



[HenHA] Hades Demo: 10-gates/60-barrel/shifter8



Shift-Right, Rotate etc.

9.6.3 Schaltnetze - ALU Schaltnetze - Barrel-Shifter

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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- bzw. Rotate-Argument



Arithmetisch-Logische Einheit (ALU)

9.6.4 Schaltnetze - ALU Schaltnetze - ALU (Arithmetisch-Logische Einheit)

64-040 Rechnerstrukturen und Betriebssysteme

Arithmetisch-logische Einheit ALU (*Arithmetic Logic Unit*)

- ▶ kombiniertes Schaltnetz für arithmetische und logische Operationen
- ▶ das zentrale Rechenwerk in Prozessoren

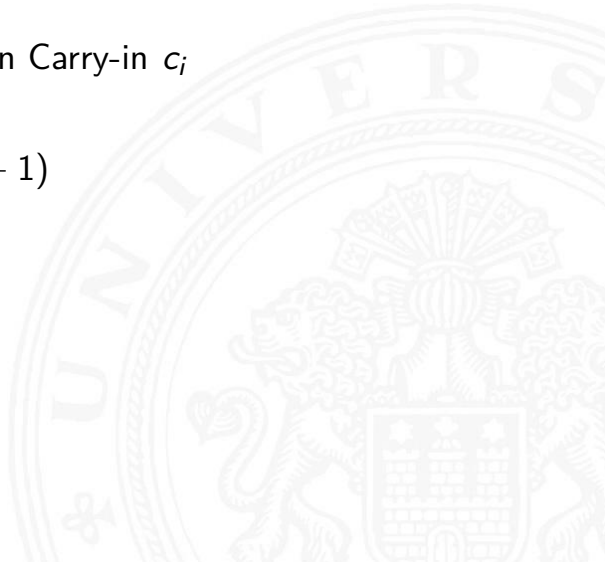
Funktionsumfang variiert von Typ zu Typ

- ▶ Addition und Subtraktion
 - ▶ bitweise logische Operationen
 - ▶ Schiebeoperationen
 - ▶ evtl. Multiplikation
 - ▶ Integer-Division selten verfügbar (separates Rechenwerk)
- 2-Komplement
Negation, UND, ODER, XOR
shift, rotate

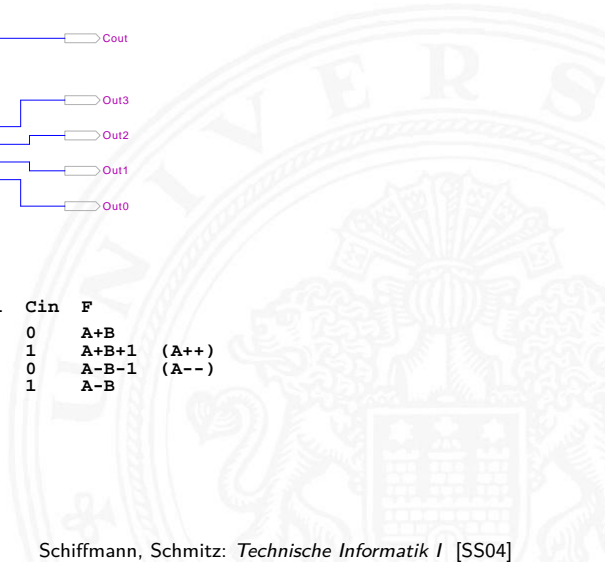
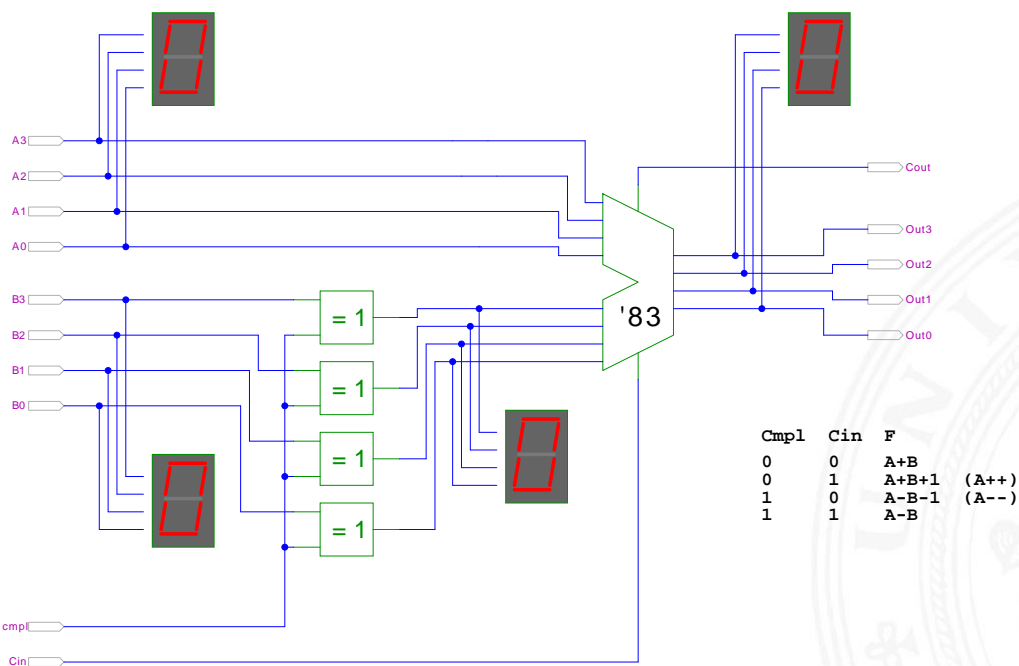


ALU: Addierer und Subtrahierer

- ▶ 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

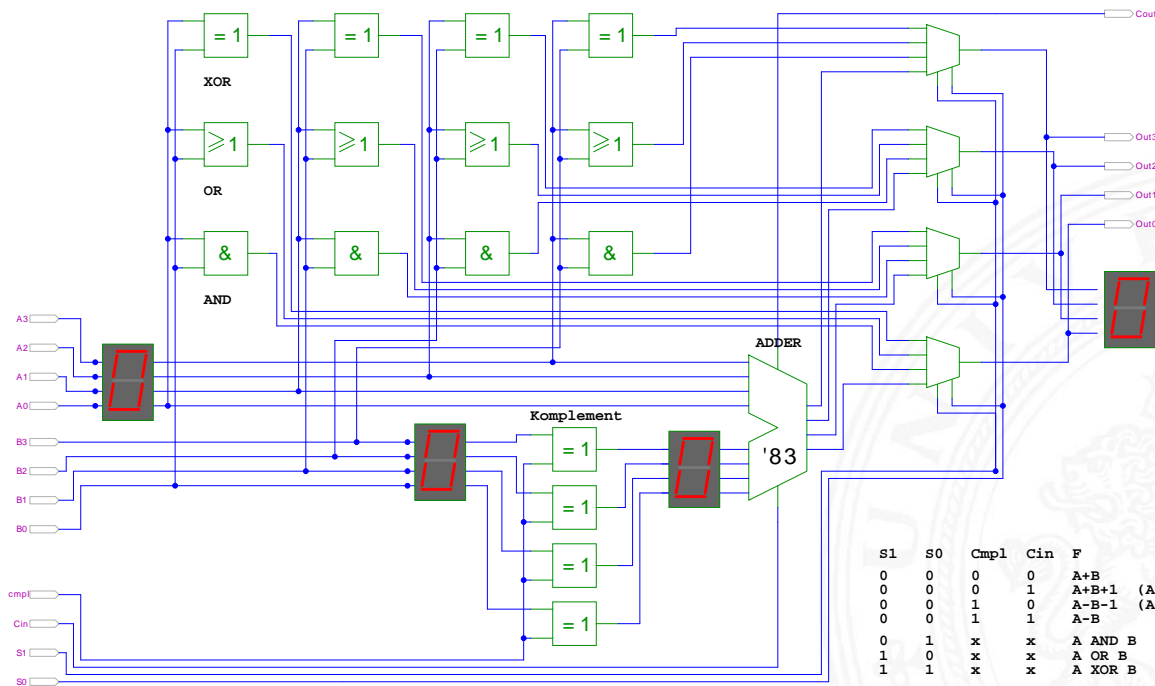


ALU: Addierer und Subtrahierer





ALU: Addierer und bitweise Operationen



Schiffmann, Schmitz: Technische Informatik I [SS04]



ALU: Prinzip

vorige Folie zeigt die „triviale“ Realisierung einer ALU

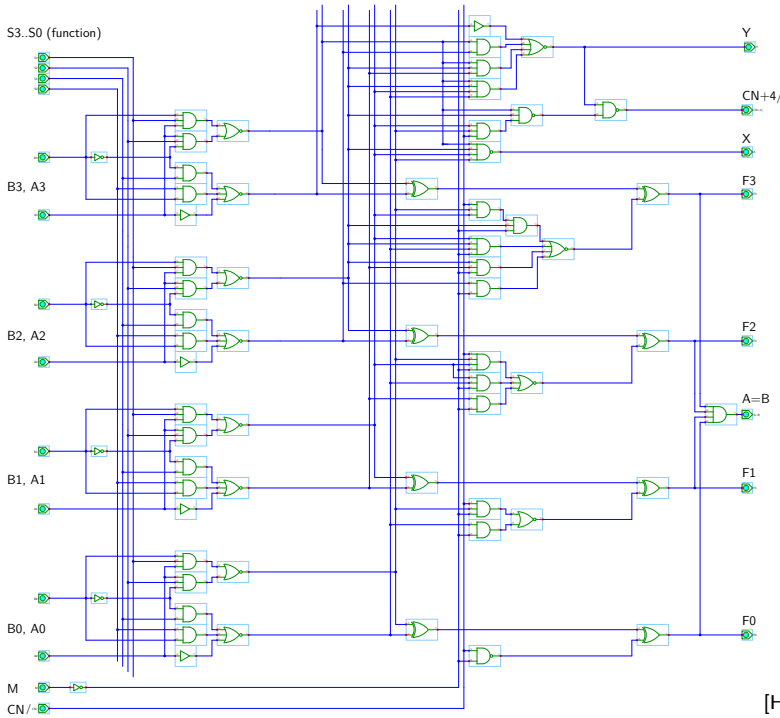
- ▶ mehrere parallele Rechenwerke für die m einzelnen Operationen
 n -bit Addierer, n -bit Komplement, n -bit OR usw.
- ▶ Auswahl des Resultats über n -bit $m:1$ -Multiplexer

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

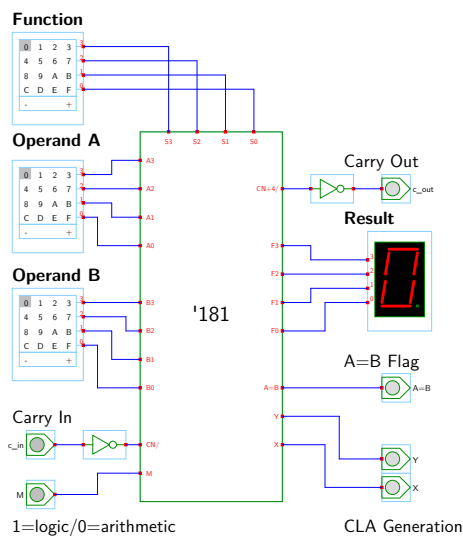


selection S3 S2 S1 S0	logic functions M = 1	arithmetic functions
		M = 0 Cn = 1 (no carry)
0 0 0 0	F = !A	F = A
0 0 0 1	F = !(A or B)	F = A or B
0 0 1 0	F = !A and B	F = A or !B
0 0 1 1	F = !A and B	F = -1
0 1 0 0	F = 0	F = A + (A and !B)
0 1 0 1	F = !B	F = (A or B) + (A and !B)
0 1 1 0	F = A xor B	F = 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



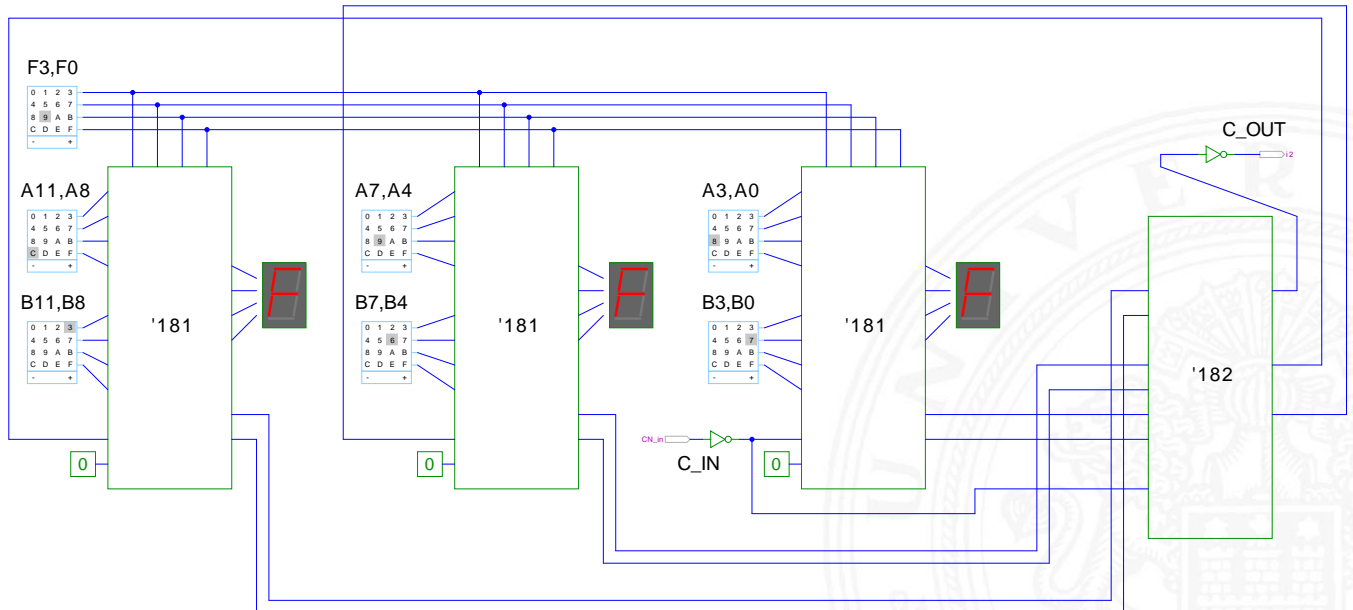
selection S3 S2 S1 S0	logic functions M = 1	arithmetic functions
		M = 0 Cn = 1 (no carry)
0 0 0 0	F = !A	F = A
0 0 0 1	F = !(A or B)	F = A or B
0 0 1 0	F = !A and B	F = A or !B
0 0 1 1	F = !A and B	F = -1
0 1 0 0	F = 0	F = A + (A and !B)
0 1 0 1	F = !B	F = (A or B) + (A and !B)
0 1 1 0	F = A xor B	F = 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



ALU: 74181 und 74182 CLA

12-bit ALU mit Carry-Lookahead Generator 74182



[HenHA] Hades Demo: 20-arithmetic/50-74181/demo-74182-ALU-CLA



Zeitverhalten einer Schaltung: Modellierung

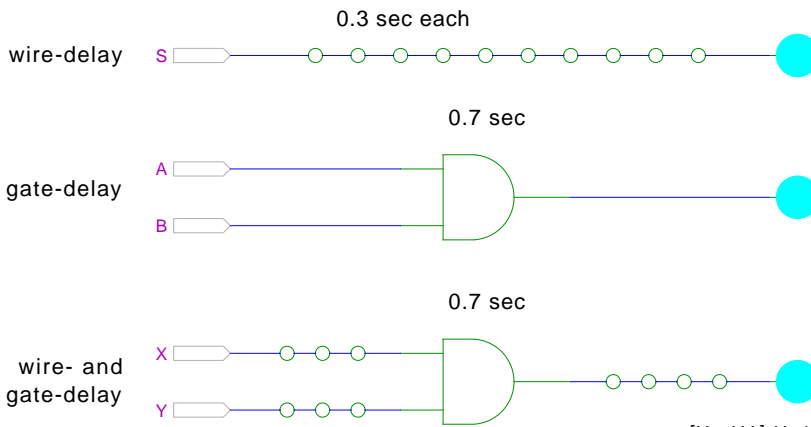
Wie wird das Zeitverhalten eines Schaltnetzes modelliert?

Gängige Abstraktionsebenen mit zunehmendem Detaillierungsgrad

1. Ausdrücke der Boole'schen Algebra: 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



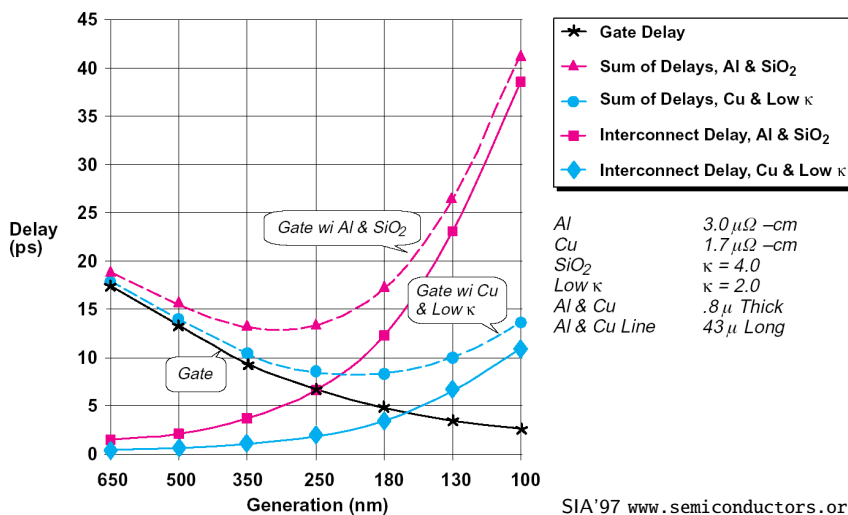
[HenHA] Hades Demo: 12-gatedelay/10-delaydemo/gate-vs-wire-delay

- ▶ 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 www.semiconductors.org



- ▶ 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*)



- ▶ Lichtgeschwindigkeit im Vakuum: $c \approx 300\,000 \text{ km/sec}$
 $\approx 30 \text{ cm/ns}$
 - ▶ in Metallen und Halbleitern langsamer: $c \approx 20 \text{ cm/ns}$
- ⇒ bei 1 Gigahertz Takt: Ausbreitung um ca. 20 Zentimeter

Abschätzungen:

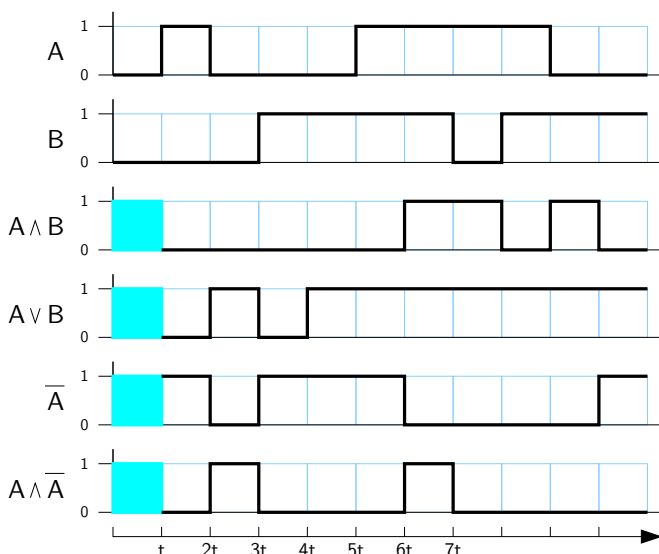
- ▶ Prozessor: ca. 3 cm Diagonale < 10 GHz Taktrate
 - ▶ Platine: ca. 20 cm Kantenlänge < 1 GHz Takt
- ⇒ prinzipiell kann (schon heute) ein Signal innerhalb eines Takts nicht von einer Ecke des ICs zur Anderen gelangen



- ▶ **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 Operation
- ▶ Ergebnis einer Operation nur, wenn die Eingaben definiert sind
- ▶ im ersten Zeitschritt noch undefinierte Werte



- ▶ **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



nach der Erscheinungsform am Ausgang

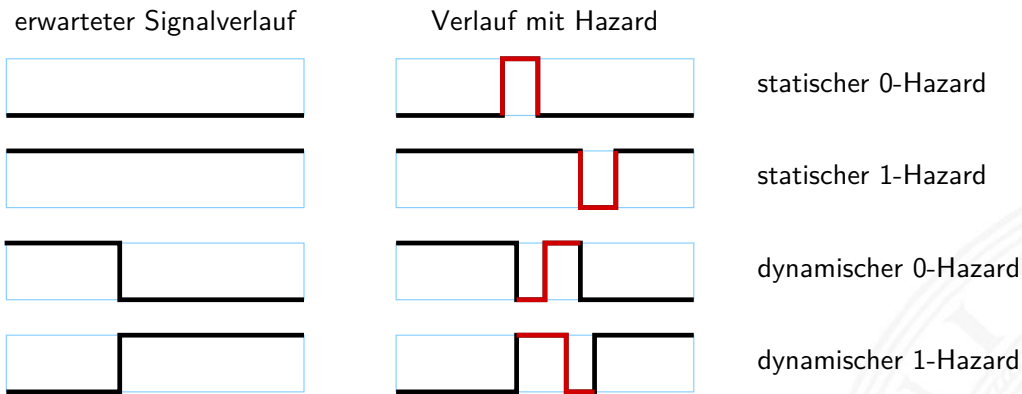
- ▶ **statisch:** der Ausgangswert soll unverändert sein, es tritt aber ein Wechsel auf
- ▶ **dynamisch:** der Ausgangswert soll (einmal) wechseln, es tritt aber ein mehrfacher Wechsel auf

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



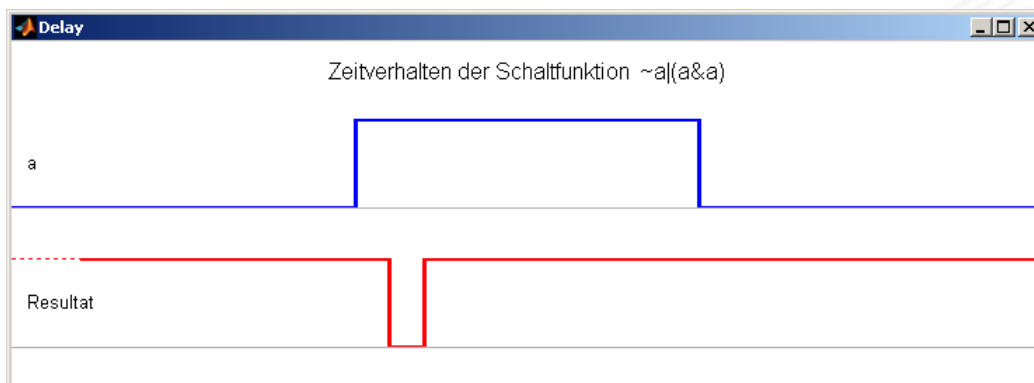
! Begriffsbildung in der Literatur nicht einheitlich: 0 / 1 als „richtiger Wert“
 –"– „fehlerhafter Wert“

- ▶ 0-Hazard wenn der Wert 0 ausgegeben werden soll, zwischenzeitlich aber 1 erscheint (1-Hazard umgekehrt)
- ▶ statisch oder dynamisch (dann auch 1-0, bzw. 0-1 Hazard)
- ▶ es können natürlich auch mehrfache Hazards auftreten



Hazards: Strukturhazard

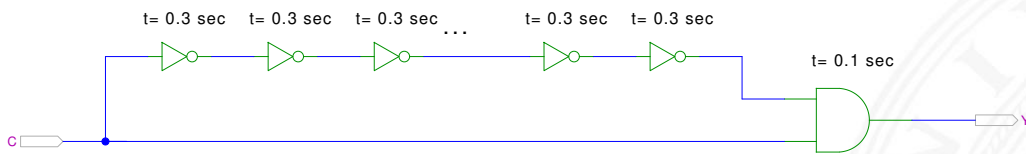
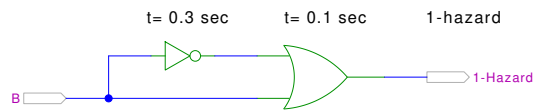
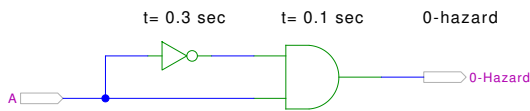
- ▶ **Strukturhazard** wird durch die gewählte Struktur der Schaltung verursacht
- ▶ auch, wenn sich nur eine Variable ändert
- ▶ Beispiel: $f(a) = \bar{a} \vee (a \wedge a)$
 a wechselt von $a = 0 \rightarrow 1$: \bar{a} schaltet schneller ab, als $(a \wedge a)$ einschaltet



- ▶ Hazard kann durch Modifikation der Schaltung beseitigt werden, hier: $f(a) = 1$



Strukturhazards: Beispiele

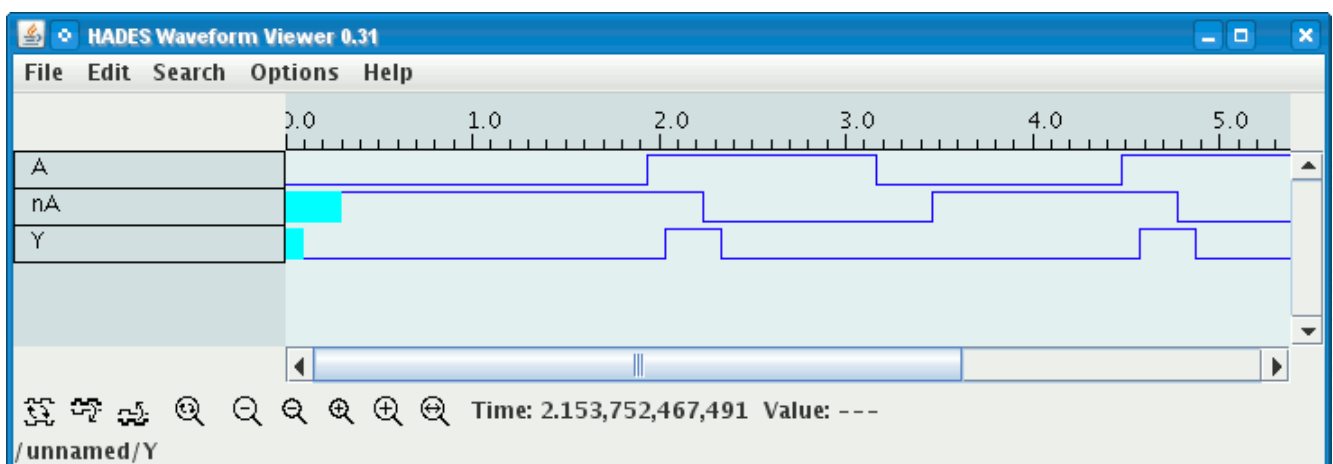


[HenHA] Hades Demo: 12-gatedelay/30-hazards/padding

- ▶ logische Funktion ist $(a \wedge \bar{a}) = 0$ bzw. $(a \vee \bar{a}) = 1$
- ▶ aber ein Eingang jeweils durch Inverter verzögert
- ⇒ kurzer Impuls beim Umschalten von $0 \rightarrow 1$ bzw. $1 \rightarrow 0$



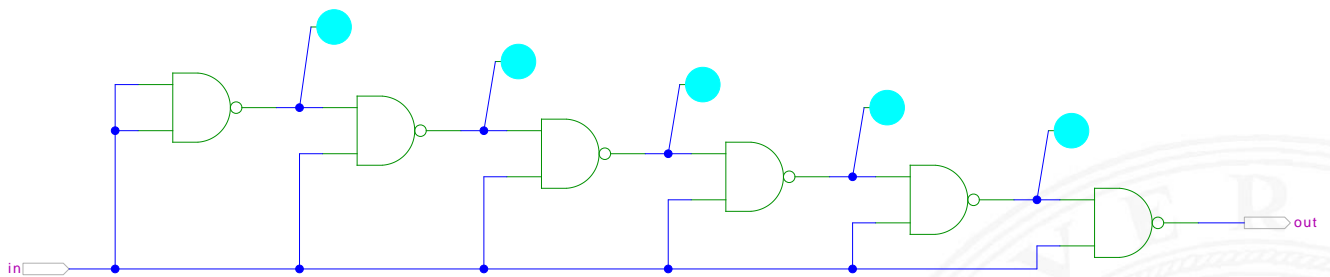
Strukturhazards: Beispiele (cont.)



- ▶ Schaltung $(a \wedge \bar{a}) = 0$ erzeugt (statischen-0) Hazard
- ▶ Länge des Impulses abhängig von Verzögerung im Inverter
- ▶ Kette von Invertern erlaubt Einstellung der Pulslänge



Strukturhazards extrem: NAND-Kette

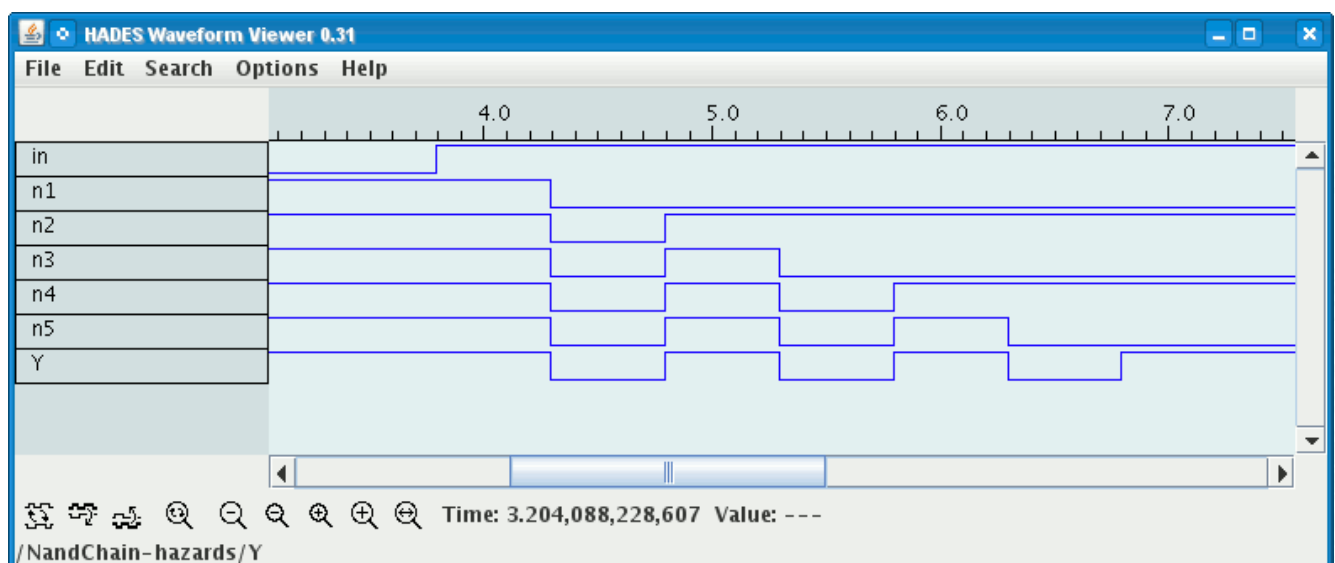


[HenHA] Hades Demo: 12-gatedelay/30-hazards/nandchain

- ▶ alle NAND-Gatter an Eingang in angeschlossen
- ▶ $in = 0$ erzwingt $y_i = 1$
- ▶ Übergang in von 0 auf 1 startet Folge von Hazards



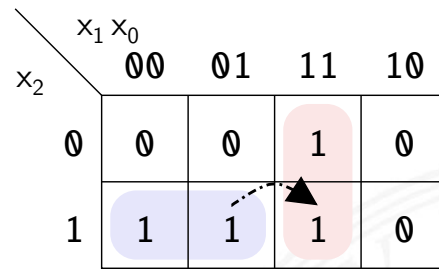
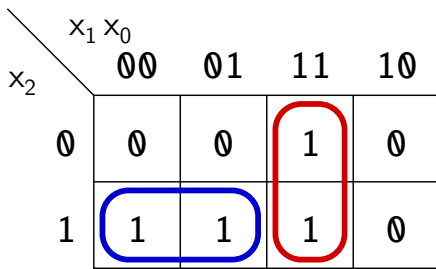
Strukturhazards extrem: NAND-Kette (cont.)



- ▶ Schaltung erzeugt Folge von (statischen-1) Hazards
- ▶ Anzahl der Impulse abhängig von Anzahl der Gatter



Strukturhazards im KV-Diagramm



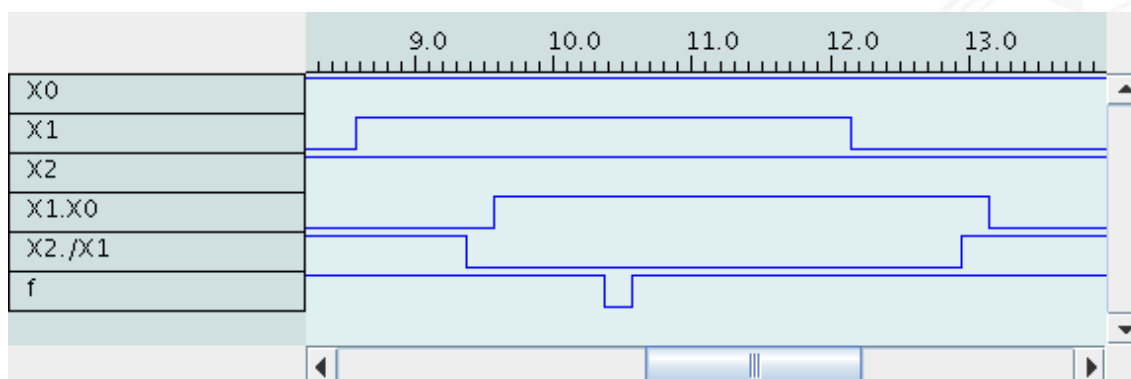
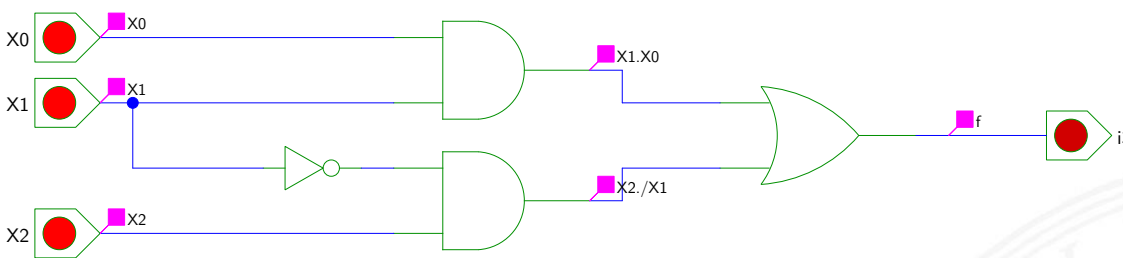
- ▶ Funktion $f = (x_2 \bar{x}_1) \vee (x_1 x_0)$
- ▶ realisiert in disjunktiver Form mit 2 Schleifen

Strukturhazard beim Übergang von $(x_2 \bar{x}_1 x_0)$ nach $(x_2 x_1 x_0)$

- ▶ Gatter $(x_2 \bar{x}_1)$ schaltet ab, Gatter $(x_1 x_0)$ schaltet ein
- ▶ Ausgang evtl. kurz 0, abhängig von Verzögerungen

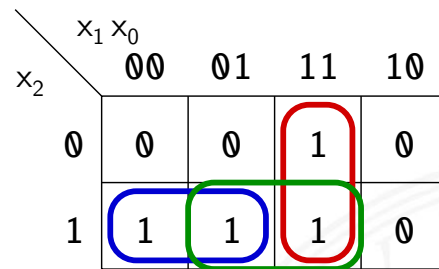
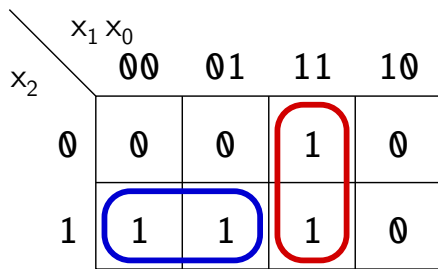


Strukturhazards im KV-Diagramm (cont.)





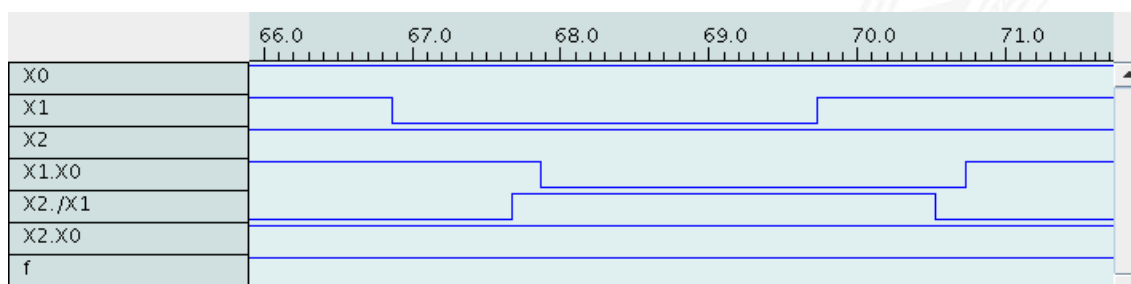
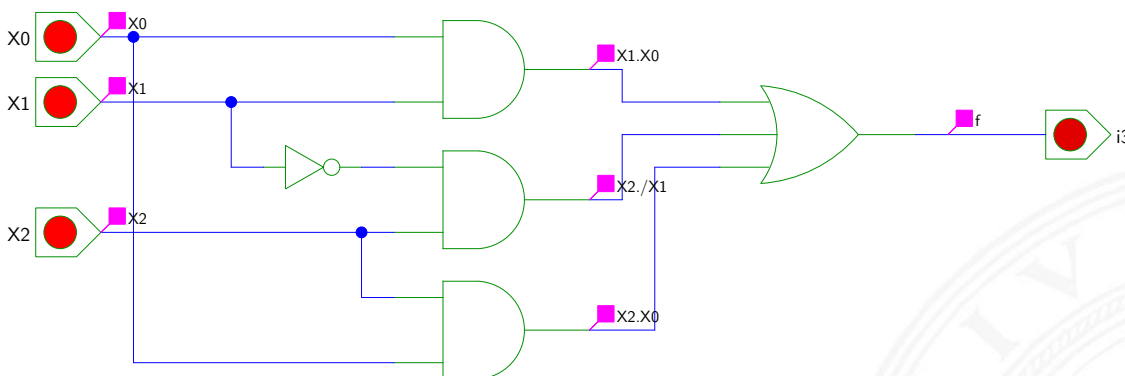
Strukturhazards beseitigen



- ▶ Funktion $f = (x_2 \bar{x}_1) \vee (x_1 x_0)$
- ▶ realisiert in disjunktiver Form mit **3 Schleifen**
 $f = (x_2 \bar{x}_1) \vee (x_1 x_0) \vee (x_2 x_0)$
- + Strukturhazard durch zusätzliche Schleife beseitigt
- aber höhere Hardwarekosten als bei minimierter Realisierung



Strukturhazards beseitigen (cont.)





Hazards: Funktionshazard

9.8 Schaltnetze - Hazards

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ **Funktionshazard** kann bei gleichzeitigem Wechsel mehrerer Eingangswerte als **Eigenschaft der Schaltfunktion** entstehen
- ▶ Problem: Gleichzeitigkeit an Eingängen
- ⇒ Funktionshazard kann nicht durch strukturelle Maßnahmen verhindert werden
- ▶ Beispiel: Übergang von $(x_2 \bar{x}_1 x_0)$ nach $(\bar{x}_2 x_1 x_0)$

		$x_1 x_0$			
		00	01	11	10
x_2	0	0	0	1	0
	1	1	1	1	0

		$x_1 x_0$			
		00	01	11	10
x_2	0	0	0	1	0
	1	1	1	1	0

A. Mäder

613



Literatur

9.9 Schaltnetze - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [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
www-cs-faculty.stanford.edu/~knuth/taocp.html
- [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
- [Weg87] I. Wegener: *The Complexity of Boolean Functions.* John Wiley & Sons, 1987. ISBN 978-0-471-91555-3
eccc.weizmann.ac.il/static/books/The_Complexity_of_Boolean_Functions
- [SS04] W. Schiffmann, R. Schmitz: *Technische Informatik 1 – Grundlagen der digitalen Elektronik.* 5. Auflage, Springer-Verlag, 2004. ISBN 978-3-540-40418-7

A. Mäder

614



Literatur (cont.)

9.9 Schaltnetze - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [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 978-0-13-334301-4
- [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 978-3-519-02332-6

A. Mäder

615



Interaktives Lehrmaterial

9.9 Schaltnetze - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript*.
Universität Hamburg, FB Informatik, 2005, Vorlesungsskript.
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

A. Mäder

616



[Laz] J. Lazzaro: *Chipmunk design tools (AnaLog, DigLog)*.

UC Berkeley, Berkeley, CA. [john-lazzaro.github.io/chipmunk](https://github.com/jlazzaro/chipmunk)



1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. Logische Operationen
7. Codierung
8. Schaltfunktionen
9. Schaltnetze
- 10. Schaltwerke**
 - Definition und Modelle
 - Asynchrone (ungetaktete) Schaltungen





Gliederung (cont.)

10 Schaltwerke

64-040 Rechnerstrukturen und Betriebssysteme

Synchrone (getaktete) Schaltungen

Flipflops

RS-Flipflop

D-Latch

D-Flipflop

JK-Flipflop

Hades

Zeitbedingungen

Taktschemata

Beschreibung von Schaltwerken

Entwurf von Schaltwerken

Beispiele

Ampelsteuerung

Zählschaltungen

Literatur



Gliederung (cont.)

10 Schaltwerke

64-040 Rechnerstrukturen und Betriebssysteme

11. Rechnerarchitektur I

12. Instruction Set Architecture

13. Assembler-Programmierung

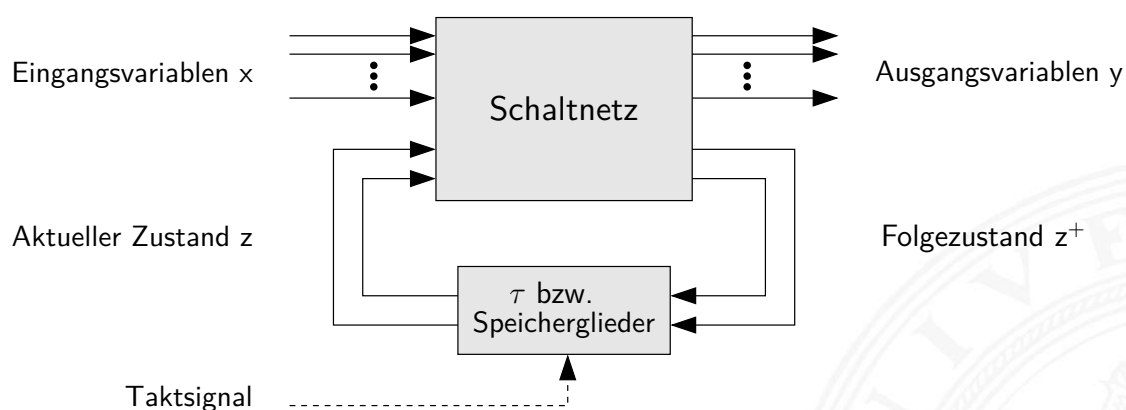
14. Rechnerarchitektur II

15. Betriebssysteme

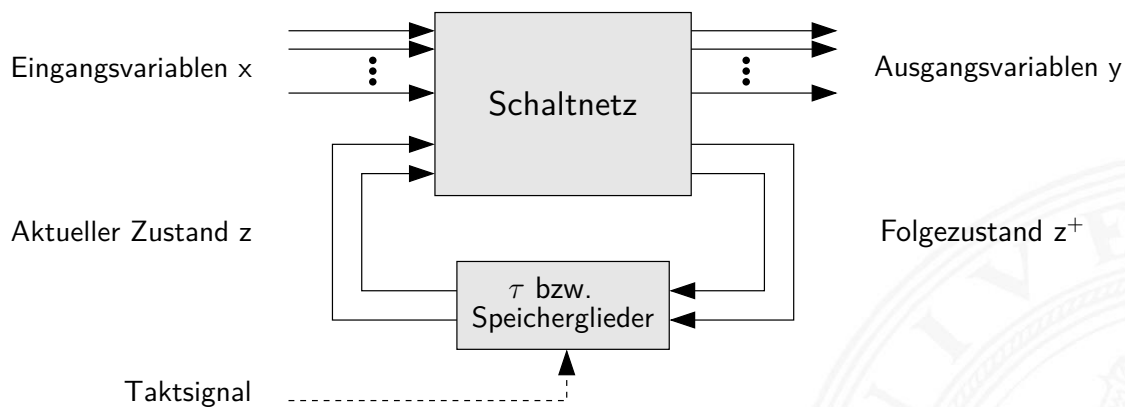




- ▶ **Schaltwerk:** Schaltung mit Rückkopplungen und Verzögerungen
- ▶ fundamental andere Eigenschaften als Schaltnetze
- ▶ Ausgangswerte nicht nur von Eingangswerten abhängig, sondern auch von der zeitlichen Abfolge der Eingangswerte
- ⇒ interner Zustand repräsentiert „Vorgeschichte“
- ▶ stabile Zustände ⇒ Speicherung von Information
- ▶ bei schlechtem Entwurf: chaotisches Verhalten und Instabilitäten
- ▶ Definition von Schaltwerken enthält Rückkopplungen
 - ▶ Widerspruch: $x = \bar{x}$
 - ▶ Mehrdeutigkeit: $x = \overline{(\bar{x})}$
 - ▶ Beispiel mit zwei Variablen: $x = \overline{(a \wedge y)}$ $y = \overline{(b \wedge x)}$



- ▶ Eingangsvariablen x und Ausgangsvariablen y
- ▶ Aktueller Zustand z
- ▶ Folgezustand z^+
- ▶ Rückkopplung läuft über Verzögerungen τ / Speicherglieder



zwei prinzipielle Varianten für die Zeitglieder

1. nur (Gatter-) Verzögerungen: **asynchrone** oder **nicht getaktete Schaltwerke**
2. getaktete Zeitglieder: **synchrone** oder **getaktete Schaltwerke**

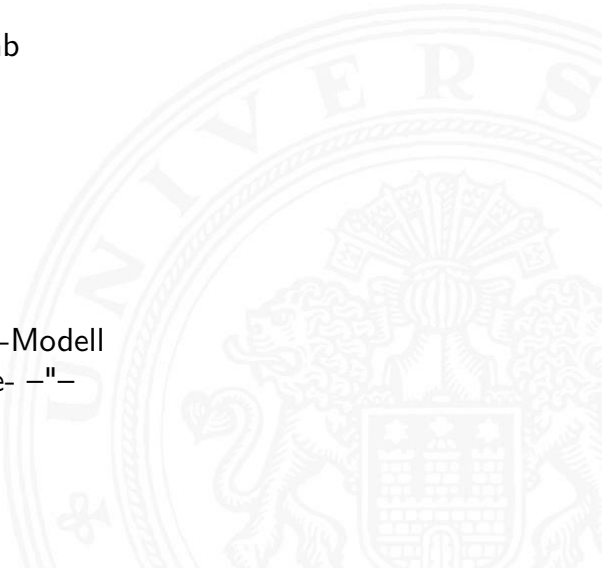


- ▶ **synchrone Schaltwerke:** die Zeitpunkte, an denen das Schaltwerk von einem stabilen Zustand in einen stabilen Folgezustand übergeht, werden explizit durch ein Taktsignal (*clock*) vorgegeben
- ▶ **asynchrone Schaltwerke:** hier fehlt ein Taktgeber und Änderungen der Eingangssignale wirken sich unmittelbar aus (nach Gatterverzögerungen τ)
- ▶ potenziell höhere Arbeitsgeschwindigkeit
- ▶ aber sehr aufwändiger Entwurf
- ▶ deutlich fehleranfälliger (z.B. leicht veränderte Gatterverzögerungen durch Bauteil-Toleranzen, Spannungsschwankungen usw.)



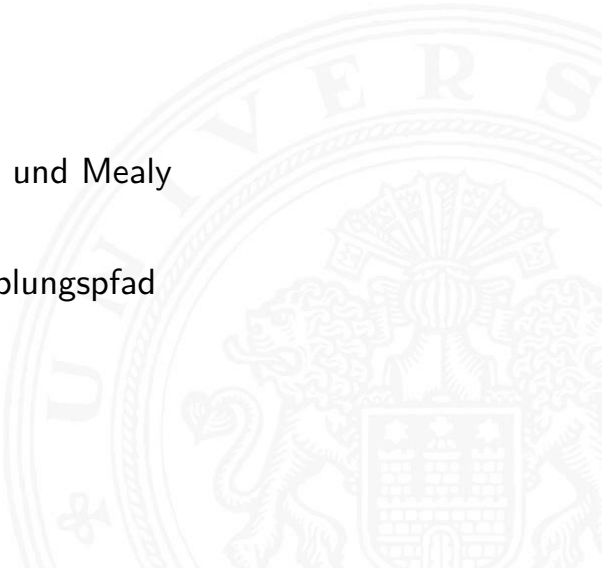
FSM – Finite State Machine

- ▶ 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:** Ausgabe hängt vom Zustand z und vom momentanen Input x ab
- ▶ **Moore-Modell:** Ausgabe des Schaltwerks hängt nur vom aktuellen Zustand z ab
- ▶ **Ausgabefunktion:**

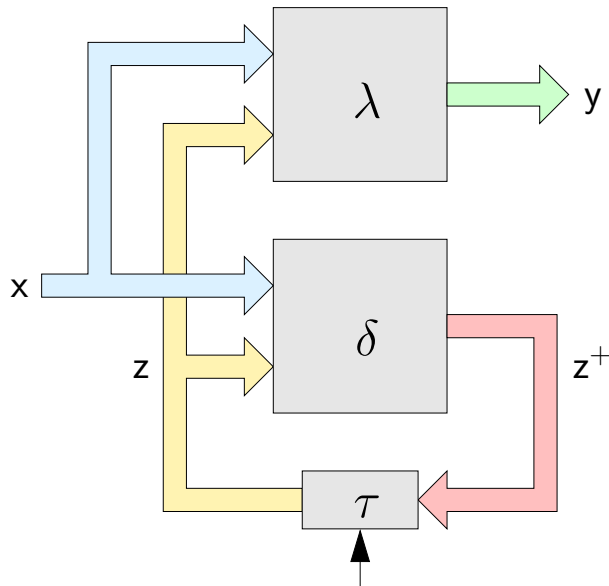
$y = \lambda(z,x)$	Mealy
$y = \lambda(z)$	Moore
- ▶ **Überföhrungsfunktion:** $z^+ = \delta(z,x)$ Moore und Mealy
- ▶ **Speicherglieder** oder Verzögerung τ im Rückkopplungspfad



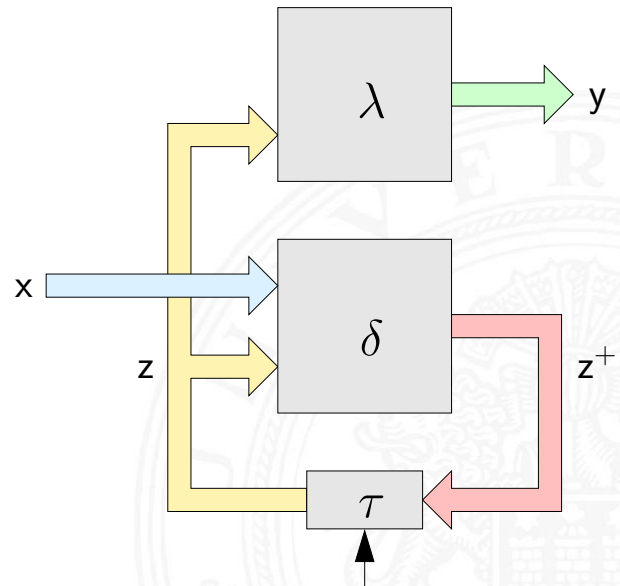


Mealy-Modell und Moore-Modell (cont.)

Mealy-Automat



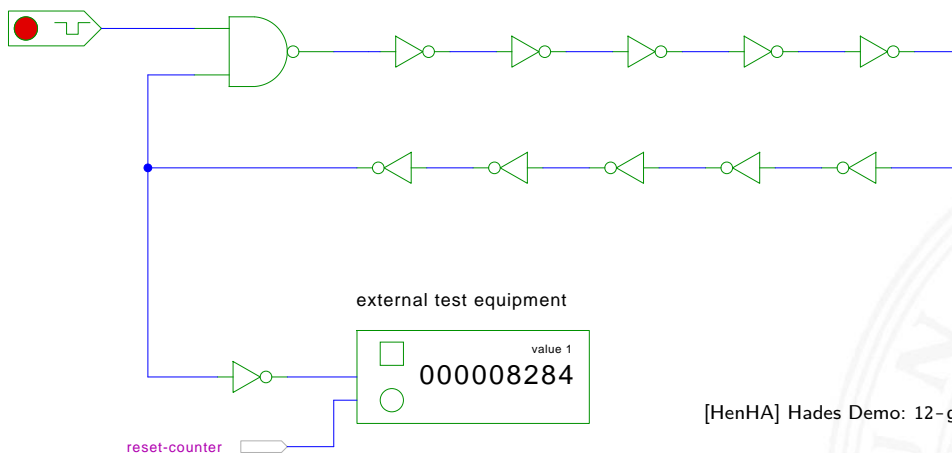
Moore-Automat



Asynchrone Schaltungen: Beispiel Ringoszillator

click to start/stop

odd number of inverting gates

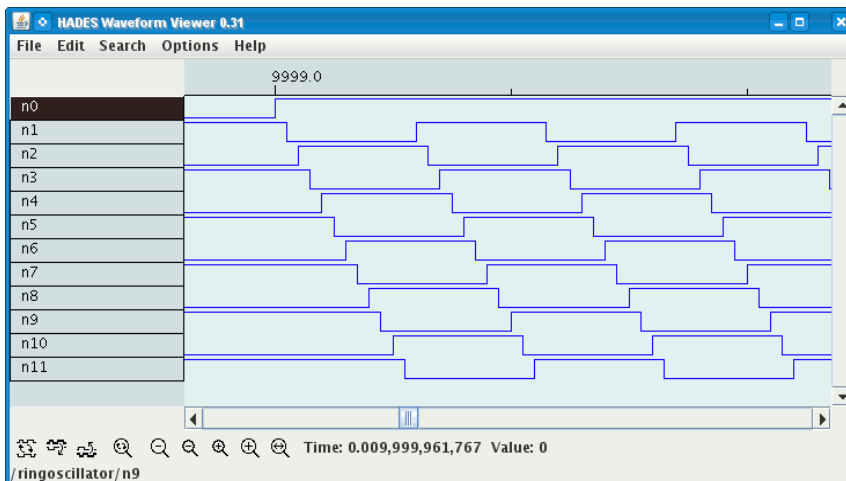


[HenHA] Hades Demo: 12-gatedelay/20-ringoscillator/ringoscillator

- ▶ 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
 - ▶ Verzögerungen der elektrischen Bauelemente sind wegen Fertigungstoleranz nicht genau bekannt und können sich im Betrieb ändern
 - ▶ Variation durch Umweltparameter
z.B. Temperatur, Versorgungsspannung, Alterung
- ⇒ sehr schwierig, die korrekte Funktion zu garantieren
z.B. mehrstufige Handshake-Protokolle
- ▶ in der Praxis deshalb **synchrone Schaltwerke**
 - ▶ Realisierung mit **Flipflops** als Zeitgliedern



Synchrone Schaltungen

10.3 Schaltwerke - Synchrone (getaktete) Schaltungen

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ alle Rückkopplungen der Schaltung laufen über spezielle Zeitglieder: „Flipflops“
 - ▶ diese definieren/speichern einen stabilen Zustand des Schaltnetzes, 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 ...
- ⇒ synchrone Schaltwerke sind wesentlich einfacher zu entwerfen und zu analysieren als asynchrone Schaltungen
- ▶ **Flipflops**, bzw. **bistabile Bauelemente** (Kippglieder)
 - ▶ je Flipflop: zwei stabile Zustände \Rightarrow speichert 1 Bit
 - ▶ Übergang zwischen diesen Zuständen durch geeignete Ansteuerung

A. Mäder

631



Flipflops

10.4 Schaltwerke - Flipflops

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Bezeichnung für **elementare** Schaltwerke
- ▶ mit genau zwei Zuständen Z_0 und Z_1
- ▶ Ausgang als Q bezeichnet und dem Zustand gleichgesetzt
- ▶ meistens auch invertierter Ausgang \bar{Q} verfügbar
- ▶ Flipflops sind selbst nicht getaktet
- ▶ sondern „sauber entworfene“ asynchrone Schaltwerke
- ▶ Anwendung als Verzögerungs-/Speicherelemente in getakteten Schaltwerken

A. Mäder

632



Flipflops: Typen

- ▶ Basis-Flipflop
- ▶ getaktetes RS-Flipflop

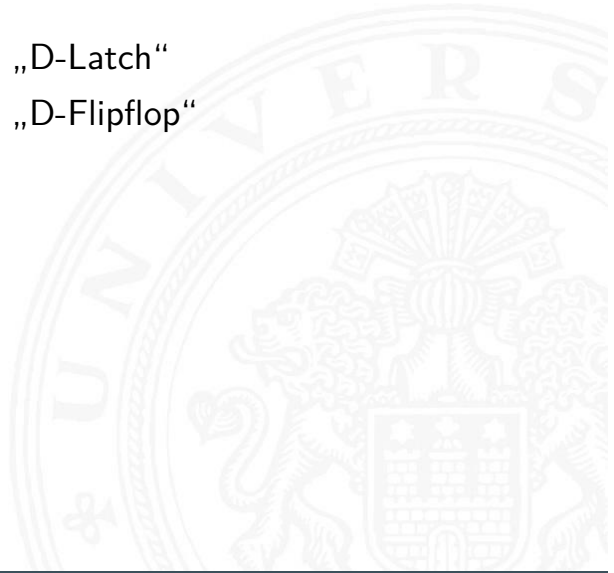
- ▶ pegelgesteuertes D-Flipflop
- ▶ flankengesteuertes D-Flipflop

- ▶ JK-Flipflop
- ▶ weitere...

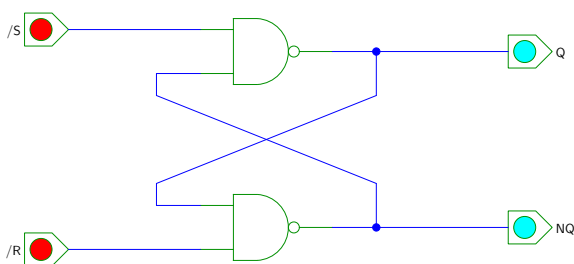
„Reset-Set-Flipflop“

„D-Latch“

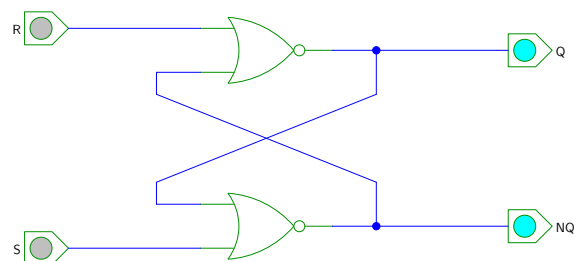
„D-Flipflop“



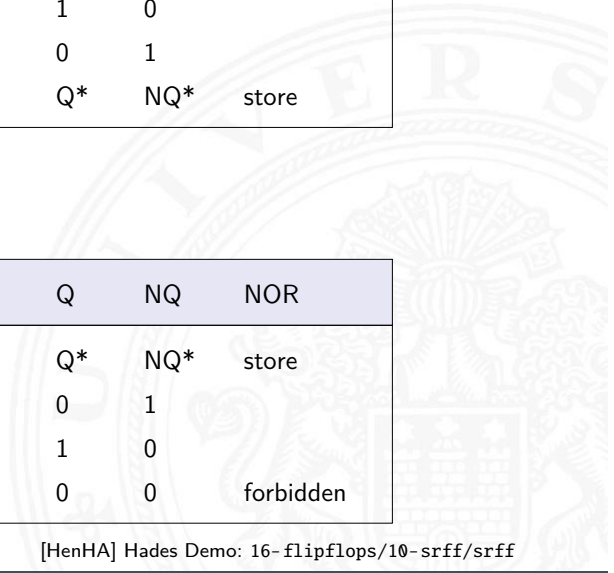
RS-Flipflop: NAND- und NOR-Realisierung



/S	/R	Q	NQ	NAND
0	0	1	1	forbidden
0	1	1	0	
1	0	0	1	
1	1	Q*	NQ*	store

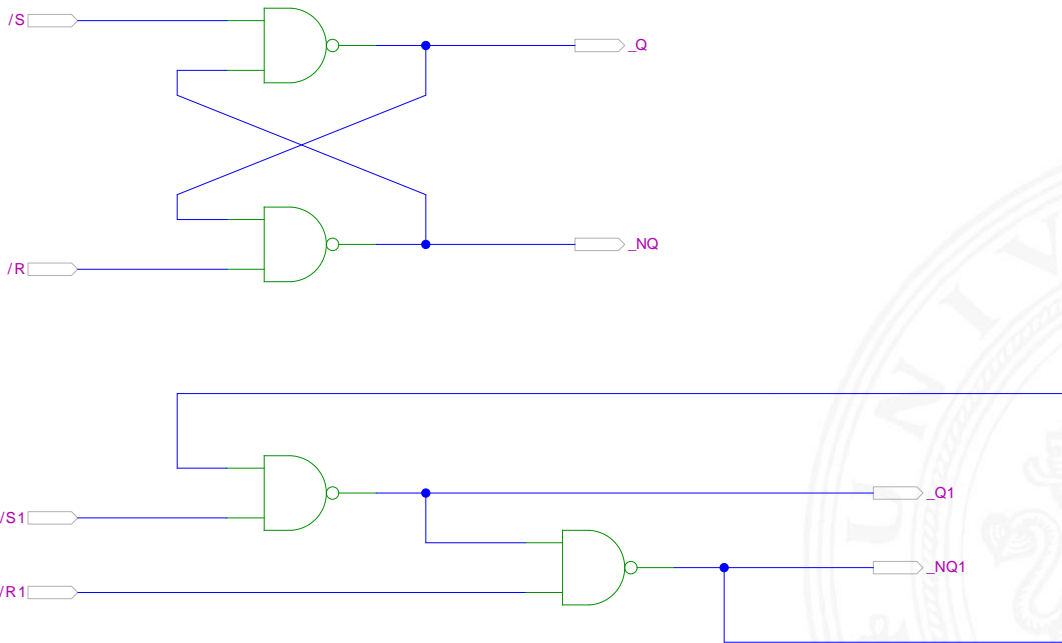


S	R	Q	NQ	NOR
0	0	Q*	NQ*	store
0	1	0	1	
1	0	1	0	
1	1	0	0	forbidden





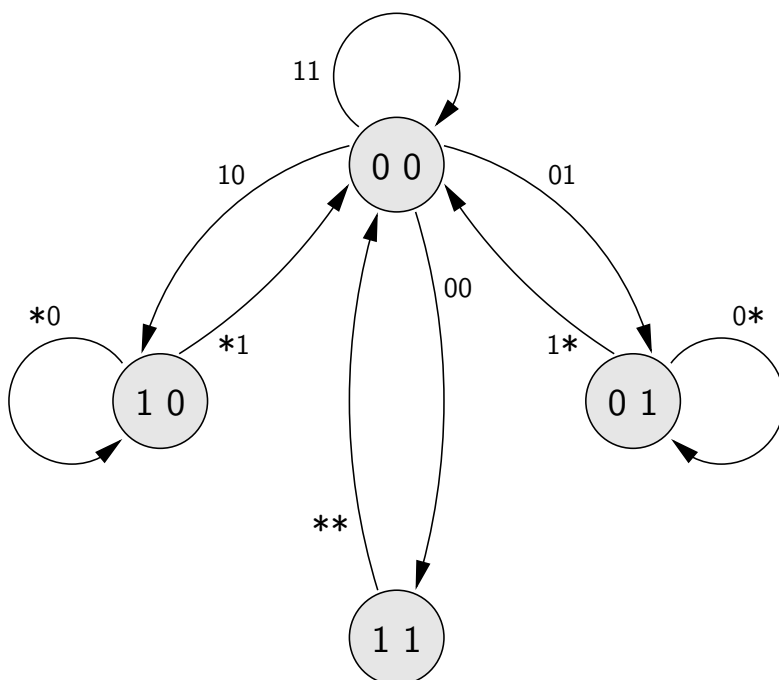
RS-Flipflop: Varianten des Schaltbilds



[HenHA] Hades Demo: 16-flipflops/10-srff/srff2



NOR RS-Flipflop: Zustandsdiagramm und Flusstafel

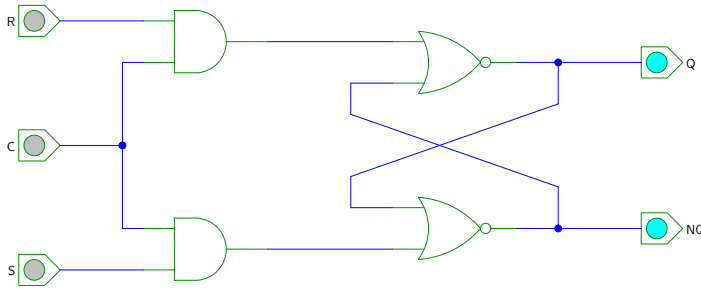


Zustand	Eingabe [S R]			
	00	01	11	10
00	11	01	00	10
01	01	01	00	00
11	00	00	00	00
10	10	00	00	10

stabiler Zustand



RS-Flipflop mit Takt



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

- ▶ RS-Basisflipflop mit zusätzlichem Takteingang C
- ▶ Änderungen nur wirksam, während C aktiv ist

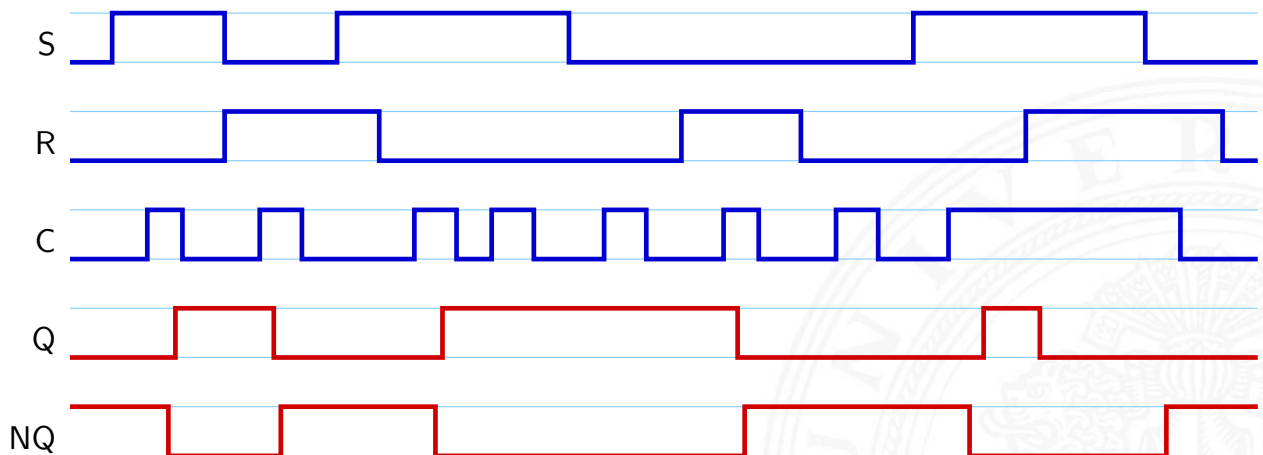
$$Q = \overline{NQ \vee (R \wedge C)}$$

$$NQ = \overline{Q \vee (S \wedge C)}$$



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^+

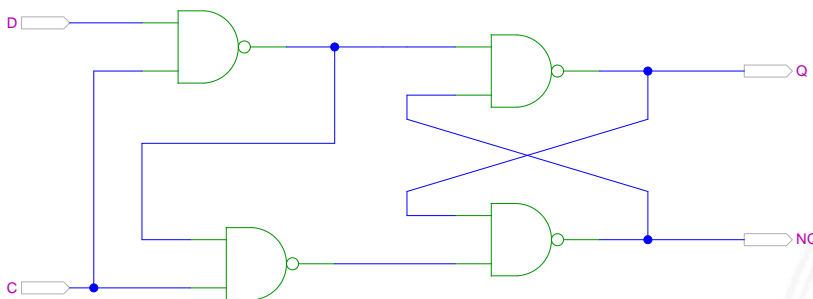
C	D	Q^+
0	0	Q
0	1	Q
1	0	0
1	1	1

- ▶ Wert am Dateneingang wird durchgeleitet, wenn das Taktsignal 1 ist \Rightarrow *high*-aktiv
0 ist \Rightarrow *low*-aktiv

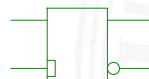


Pegelgesteuertes D-Flipflop (D-Latch) (cont.)

- ▶ Realisierung mit getaktetem RS-Flipflop und einem Inverter: $S = D, R = \bar{D}$
- ▶ minimierte NAND-Struktur

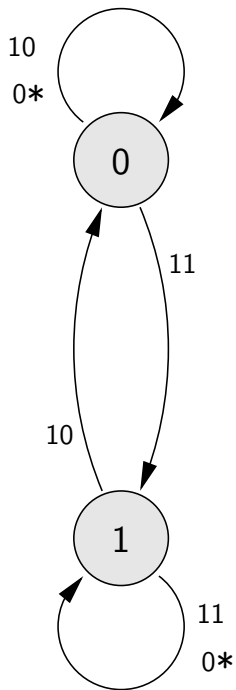


- ▶ Symbol





D-Latch: Zustandsdiagramm und Flusstafel



Zustand [Q]	Eingabe [C D]			
	00	01	11	10
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
↑	0	0
↑	1	1

- ▶ Wert am Dateneingang wird gespeichert, wenn das Taktsignal sich von 0 auf 1 ändert ⇒ Vorderflankensteuerung
- "– 1 auf 0 ändert ⇒ Rückflankensteuerung
- ▶ Realisierung als Master-Slave Flipflop oder direkt



Master-Slave D-Flipflop

10.4.3 Schaltwerke - Flipflops - D-Flipflop

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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
- ⇒ dies entspricht effektiv einer **Flankensteuerung**:
Wert an D nur relevant, kurz bevor Takt auf $C = 1$ wechselt

A. Mäder

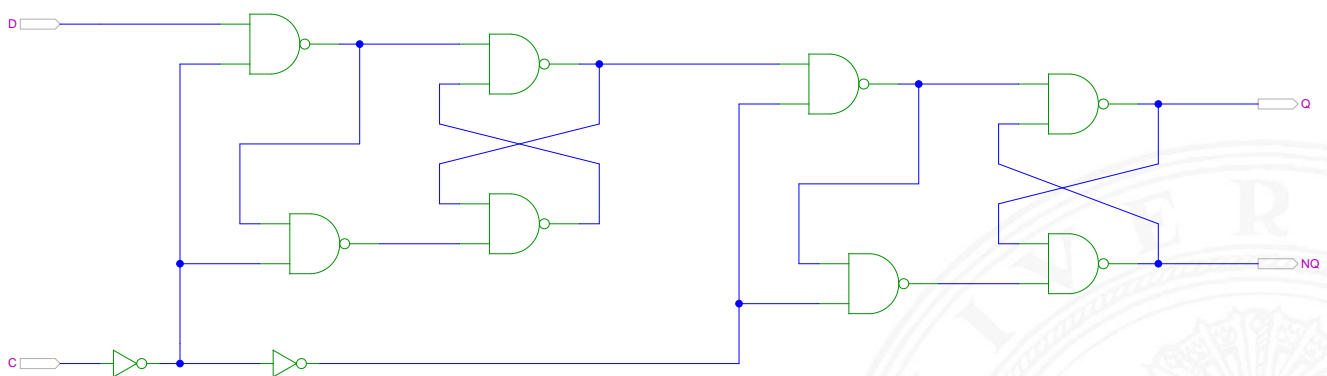
643



Master-Slave D-Flipflop (cont.)

10.4.3 Schaltwerke - Flipflops - D-Flipflop

64-040 Rechnerstrukturen und Betriebssysteme



[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

A. Mäder

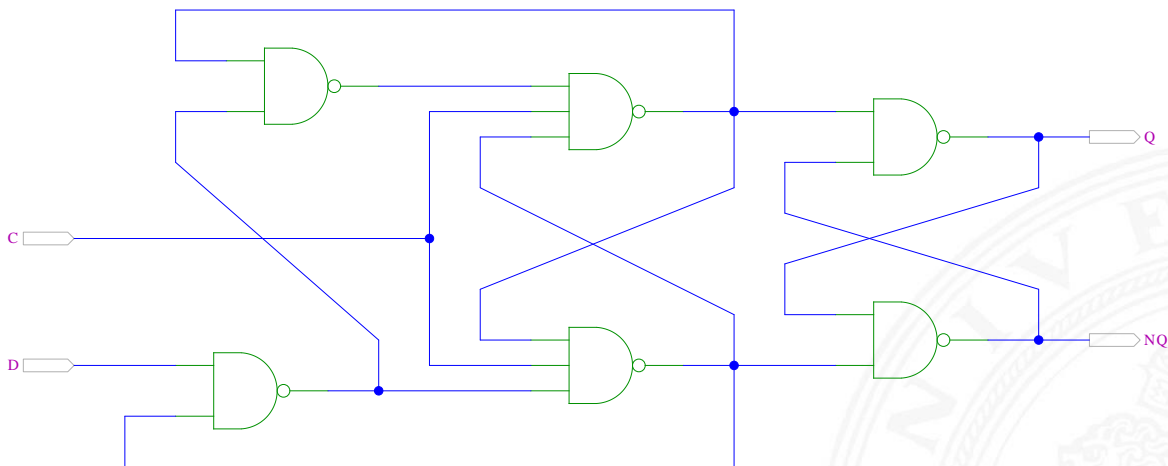
644



Vorderflanken-gesteuertes D-Flipflop

10.4.3 Schaltwerke - Flipflops - D-Flipflop

64-040 Rechnerstrukturen und Betriebssysteme



- ▶ Dateneingang D wird nur durch Takt-Vorderflanke ausgewertet
- ▶ Gatterlaufzeiten für Funktion essenziell
- ▶ Einhalten der Vorlauf- und Haltezeiten vor/nach der Taktflanke (s.u. *Zeitbedingungen*)

A. Mäder

645



JK-Flipflop

10.4.4 Schaltwerke - Flipflops - JK-Flipflop

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Takteingang C
- ▶ Steuereingänge J („jump“) und K („kill“)
- ▶ aktueller Zustand Q , Folgezustand Q^+

C	J	K	Q^+	Funktion
*	*	*	Q	Wert gespeichert
↑	0	0	Q	Wert gespeichert
↑	0	1	0	Rücksetzen
↑	1	0	1	Setzen
↑	1	1	\bar{Q}	Invertieren

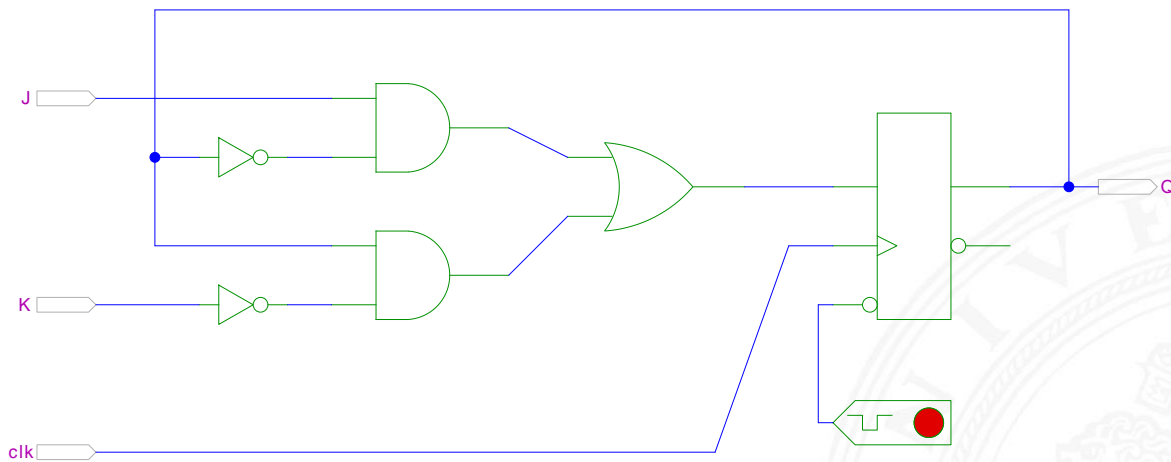
- ▶ universelles Flipflop, sehr flexibel einsetzbar
- ▶ in integrierten Schaltungen nur noch selten verwendet (höherer Hardware-Aufwand als Latch/D-Flipflop)

A. Mäder

646



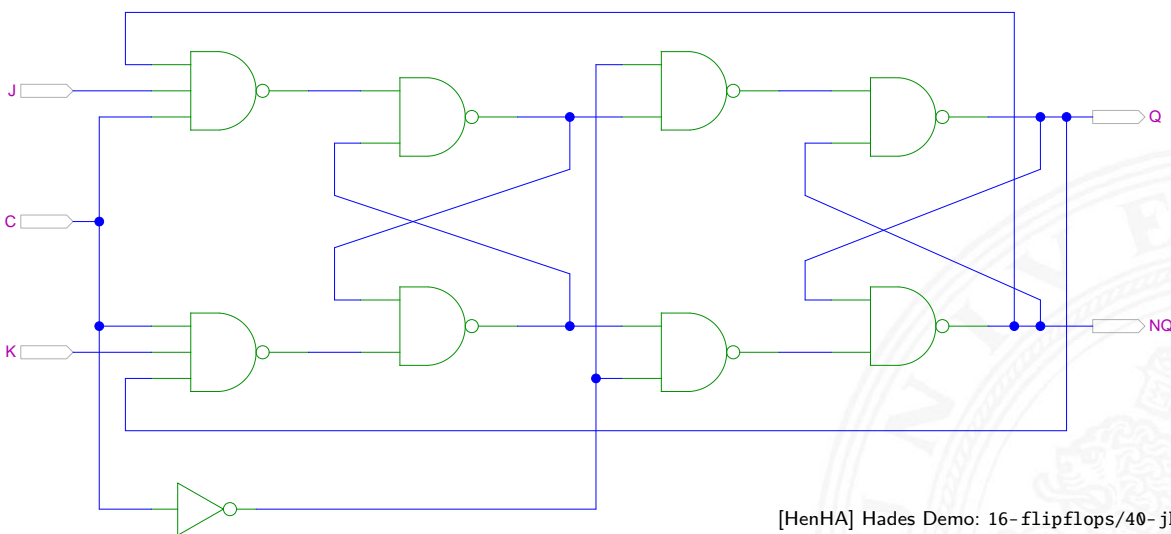
JK-Flipflop: Realisierung mit D-Flipflop



[HenHA] Hades Demo: 16-flipflops/40-jkff/jkff-prinzip



JK-Flipflop: Realisierung als Master-Slave Schaltung

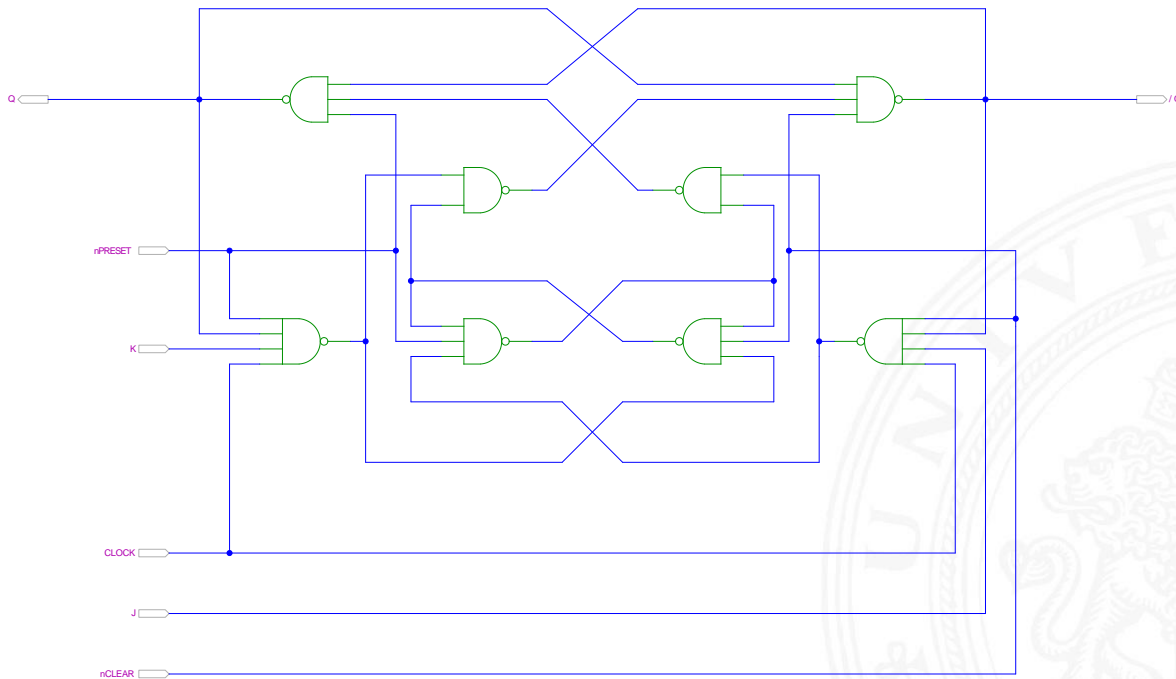


[HenHA] Hades Demo: 16-flipflops/40-jkff/jkff

► Achtung: Schaltung wegen Rückkopplungen schwer zu initialisieren



JK-Flipflop: tatsächliche Schaltung im IC 7476

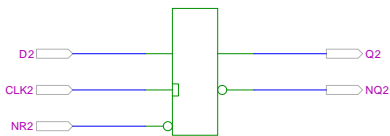
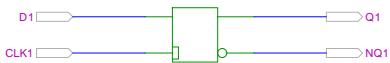


[HenHA] Hades Demo: 16-flipflops/40-jkff/SN7476-single

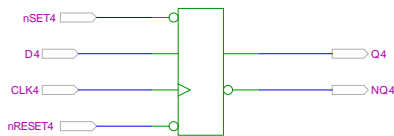
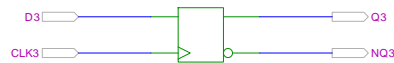


Flipflop-Typen: Komponenten/Symbole in Hades

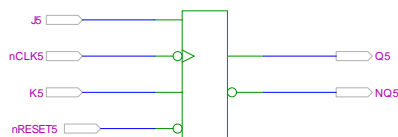
D-type latches



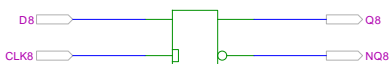
D-type flipflops



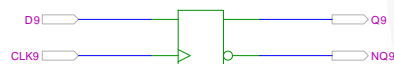
JK flipflop



metastable D-Latch (don't use!)



metastable D-flipflop (don't use!)



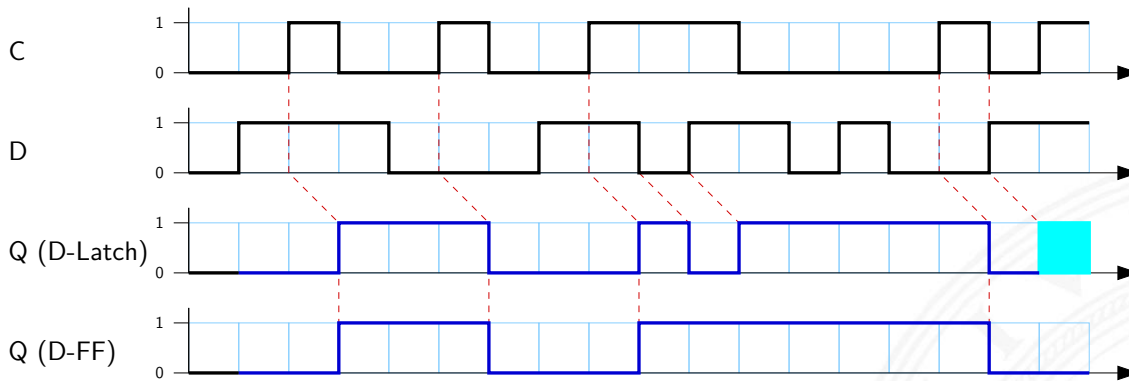
[HenHA] Hades Demo: 16-flipflops/50-ffdemo/flipflopdemo



Flipflop-Typen: Impulsdiagramme

10.4.5 Schaltwerke - Flipflops - Hades

64-040 Rechnerstrukturen und Betriebssysteme



- ▶ pegel- und vorderflankengesteuertes Flipflop im Vergleich
- ▶ beide Flipflops hier mit jeweils einer Zeiteinheit Verzögerung
- ▶ undefinierte Werte im Latch (cyan dargestellt)
 - ▶ Verletzung der Zeitbedingungen!
 - ▶ gleichzeitiger Wechsel von C und D
 - ▶ in der Realität wird natürlich ein Wert 0 oder 1 gespeichert, er ist aber von externen Parametern abhängig: Temperatur, Versorgungsspannung etc.

A. Mäder

651



Flipflops: Zeitbedingungen

10.5 Schaltwerke - Zeitbedingungen

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Flipflops werden entwickelt, um Schaltwerke einfacher zu entwerfen und betreiben
 - ▶ Umschalten des Zustandes wird synchron 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
- ⇒ Daten- und Takteingänge dürfen sich nicht gleichzeitig ändern
Welcher Wert wird gespeichert?
- ⇒ „Vorlauf- und Haltezeiten“ (*setup- / hold-time*)

A. Mäder

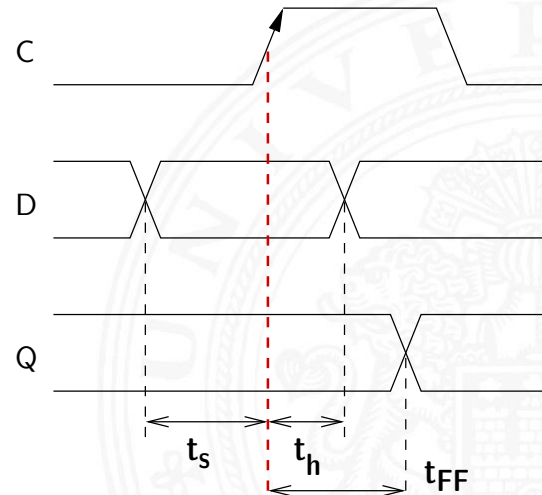
652



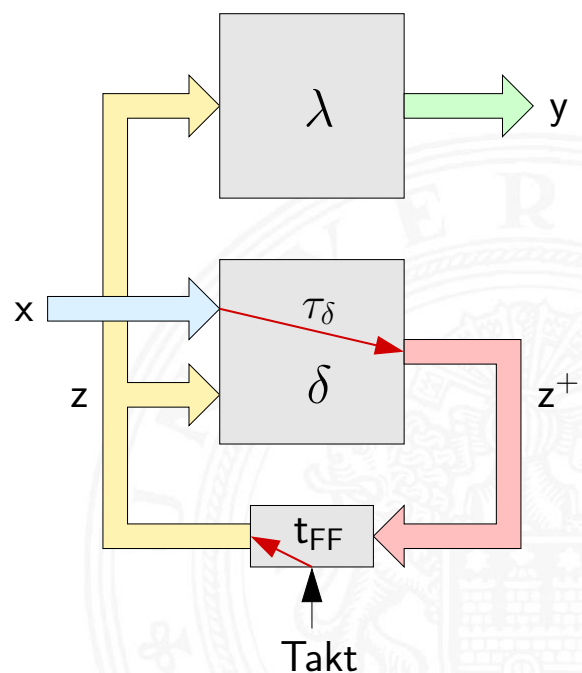
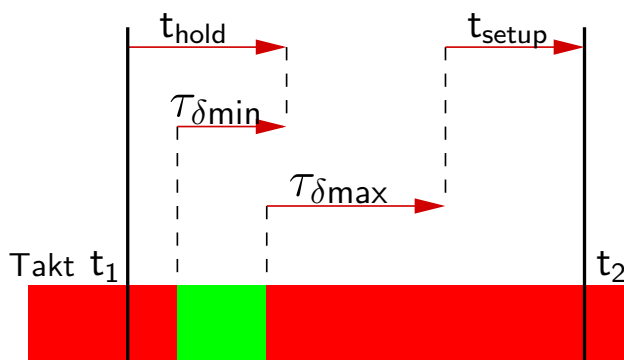
Flipflops: Vorlauf- und Haltezeit

- ▶ 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

⇒ Verletzung der Zeitbedingungen „falscher“ Wert an Q



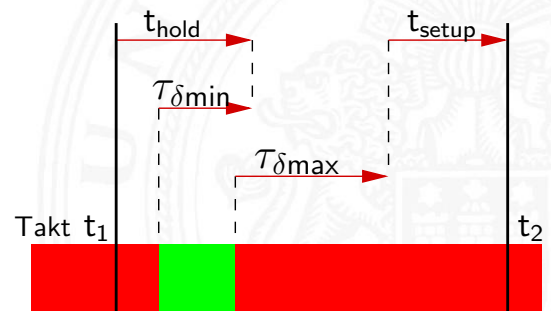
Zeitbedingungen: Eingangsvektor



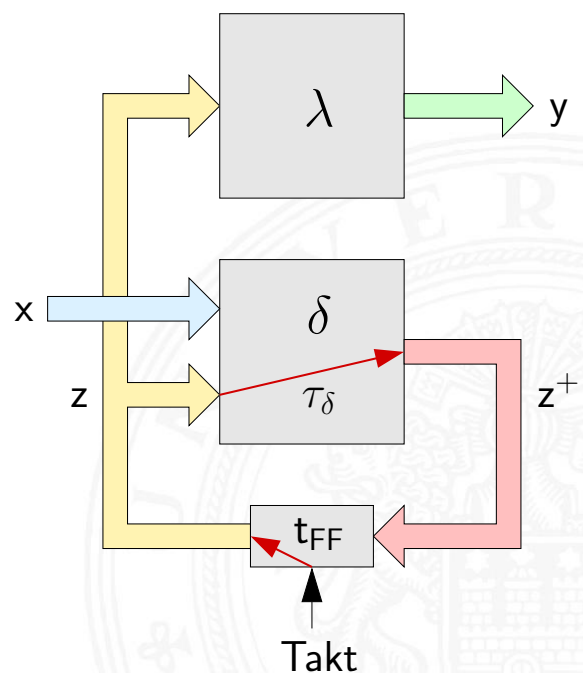
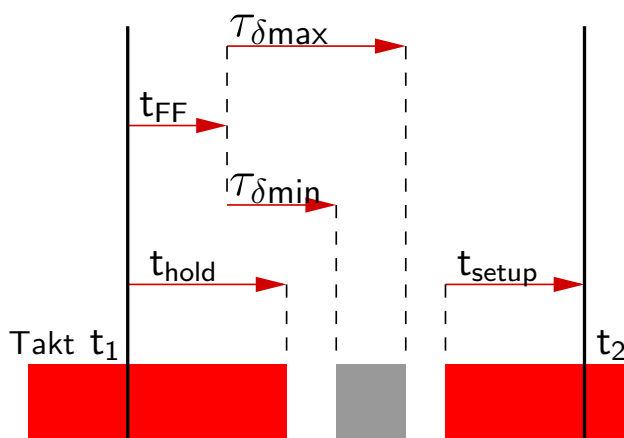


Zeitbedingungen: Eingangsvektor (cont.)

- ▶ Änderungen der Eingangswerte x werden beim Durchlaufen von δ mindestens um $\tau_{\delta\min}$, bzw. maximal um $\tau_{\delta\max}$ verzögert
 - ▶ um die Haltezeit der Zeitglieder einzuhalten, darf x sich nach einem Taktimpuls frühestens zum Zeitpunkt $(t_1 + t_{\text{hold}} - \tau_{\delta\min})$ wieder ändern
 - ▶ um die Vorlaufzeit vor dem nächsten Takt einzuhalten, muss x spätestens zum Zeitpunkt $(t_2 - t_{\text{setup}} - \tau_{\delta\max})$ wieder stabil sein
- ⇒ Änderungen dürfen nur innerhalb des grün markierten Zeitintervall erfolgen



Zeitbedingungen: interner Zustand

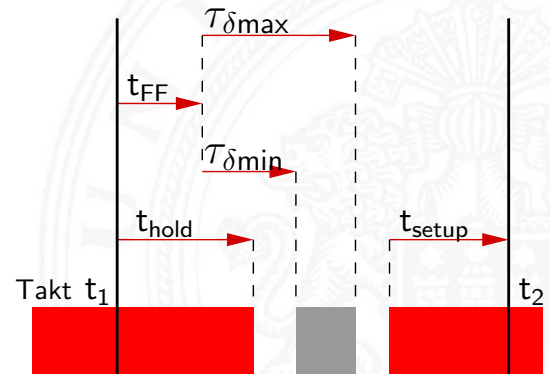




Zeitbedingungen: interner Zustand (cont.)

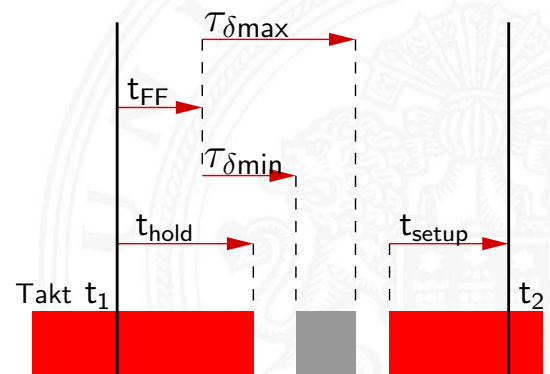
- ▶ zum Zeitpunkt t_1 wird ein Taktimpuls ausgelöst
- ▶ nach t_{FF} haben die Zeitglieder (Flipflops) den aktuellen Eingangswert z^+ übernommen und geben ihn am Ausgang als neuen Zustand z aus
- ▶ diese neuen Werte von z laufen durch das δ -Schaltnetz, dabei ist der schnellste Pfad $\tau_{\delta_{min}}$ und der langsamste $\tau_{\delta_{max}}$

⇒ der Folgezustand ändert sich innerhalb des grau markierten Zeitintervalls:
 $[(t_{FF} + \tau_{\delta_{min}}) \dots (t_{FF} + \tau_{\delta_{max}})]$



Zeitbedingungen: interner Zustand (cont.)

- ▶ Änderungen am FF-Eingang z^+ dürfen frühestens zum Zeitpunkt $(t_1 + t_{hold})$ beginnen, sonst wird die Haltezeit t_{hold} verletzt
dazu muss ggf. $\tau_{\delta_{min}}$ vergrößert werden (zusätzliche Gatterverzögerungen)
- ▶ Änderungen am FF-Eingang z^+ müssen sich spätestens bis zum Zeitpunkt $(t_2 - t_{setup})$ stabilisiert haben, wegen der Vorlaufzeit t_{setup} vor dem nächsten Takt





Maximale Taktfrequenz einer Schaltung

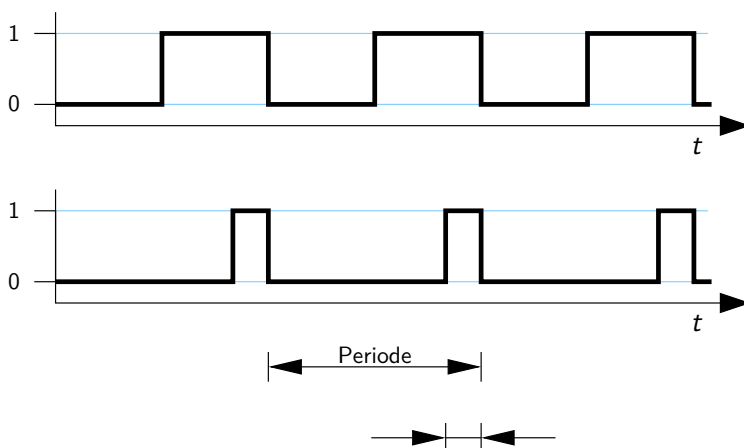
- ▶ aus den beiden vorigen Bedingungen ergibt sich sofort die maximal zulässige Taktfrequenz einer Schaltung
- ▶ Umformen und Auflösen nach dem Zeitpunkt des nächsten Takts ergibt zwei notwendige Zeitbedingungen

$$\Delta t \geq (t_{FF} + \tau_{\delta_{\max}} + t_{\text{setup}}) \quad \text{und}$$
$$\Delta t \geq (t_{\text{hold}} + t_{\text{setup}})$$

- ▶ falls dieses Timing verletzt wird (z.B. durch „Übertakten“), kann es (datenabhängig) zu Fehlfunktionen kommen



Taktsignal: Prinzip



- ▶ periodisches digitales Signal, Frequenz f bzw. Periode τ
- ▶ oft symmetrisch
- ▶ asymmetrisch für Zweiphasentakt (s.u.)



▶ Pegelsteuerung

Schaltung reagiert auf Eingang, solange das Taktsignal den Wert 1 (bzw. 0) hat

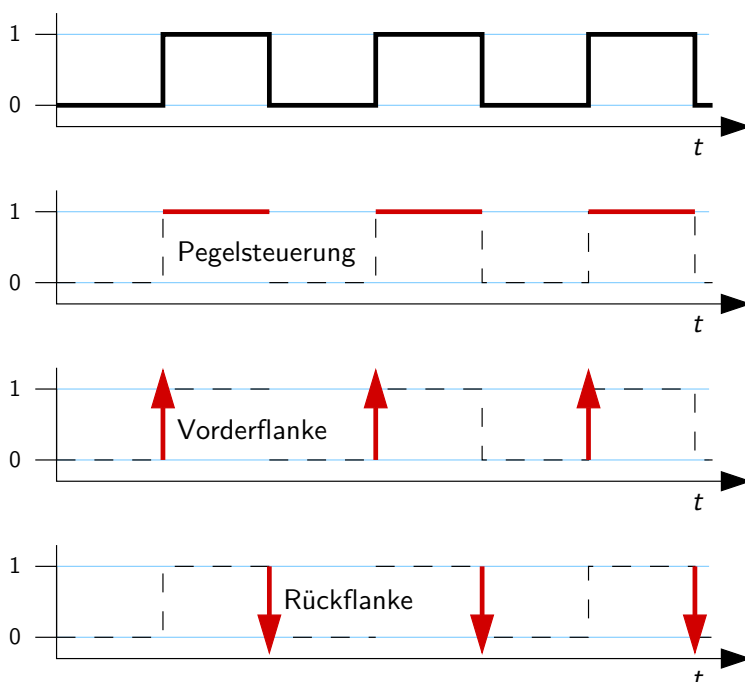
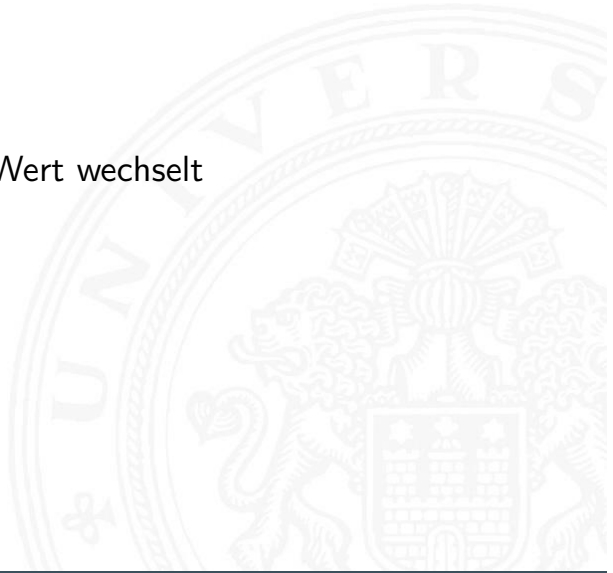
- ▶ high-aktiv: Takt = 1
- ▶ low-aktiv: Takt = 0

▶ Flankensteuerung

Schaltung reagiert nur, wenn das Taktsignal den Wert wechselt

- ▶ Vorderflankensteuerung: Wechsel von 0 nach 1
- ▶ Rückflankensteuerung: —" von 1 nach 0

▶ Zwei- und Mehrphasentakte

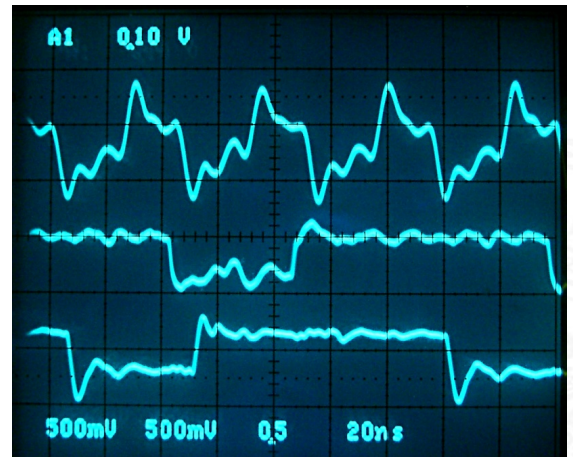
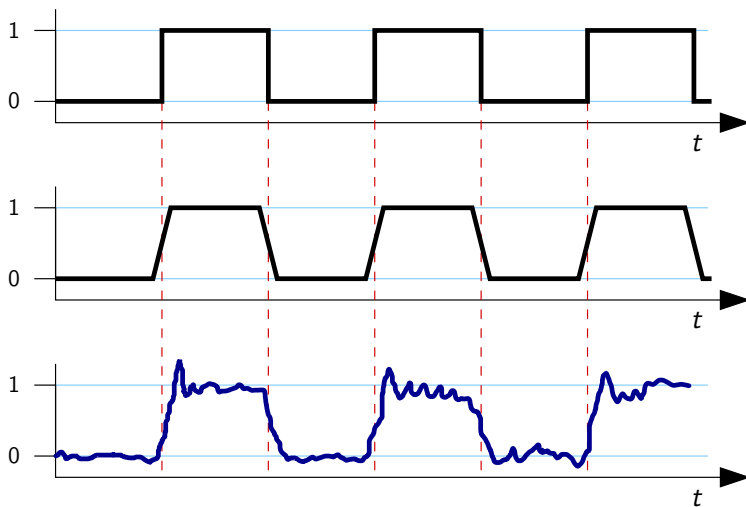




Taktsignal: Prinzip und Realität

10.6 Schaltwerke - Taktschemata

64-040 Rechnerstrukturen und Betriebssysteme



- ▶ Werteverläufe in realen Schaltungen stark gestört
- ▶ Überschwingen/Übersprechen benachbarter Signale
- ▶ Flankensteilheit nicht garantiert (bei starker Belastung)
ggf. besondere Gatter („Schmitt-Trigger“)

A. Mäder

663

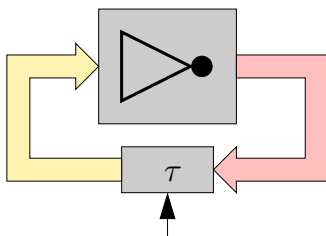


Problem bei Pegelsteuerung

10.6 Schaltwerke - Taktschemata

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ während des aktiven Taktpegels werden Eingangswerte minimal verzögert an den Ausgang weiter gegeben
- ▶ durch Invertierungen in den Rückkopplungspfaden von δ , kommt es zu instabilen Zuständen (Oszillationen: $0 \rightarrow 1 \rightarrow 0 \rightarrow \dots$)



- ▶ einzelne pegelgesteuerte Zeitglieder (D-Latches) funktionieren nicht in Rückkopplungspfaden
- ⇒ Verwendung von je zwei pegelgesteuerten Zeitgliedern mit Zweiphasentakt oder
- ⇒ Verwendung flankengesteuerter D-Flipflops

A. Mäder

664



Zweiphasentakt

10.6 Schaltwerke - Taktschemata

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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, disjunkte 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

A. Mäder

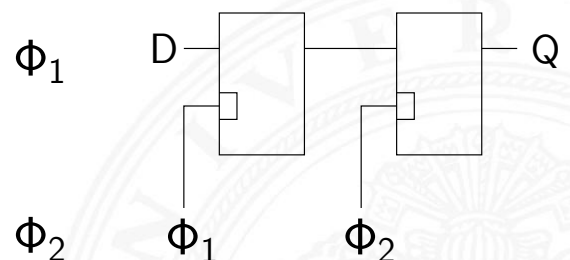
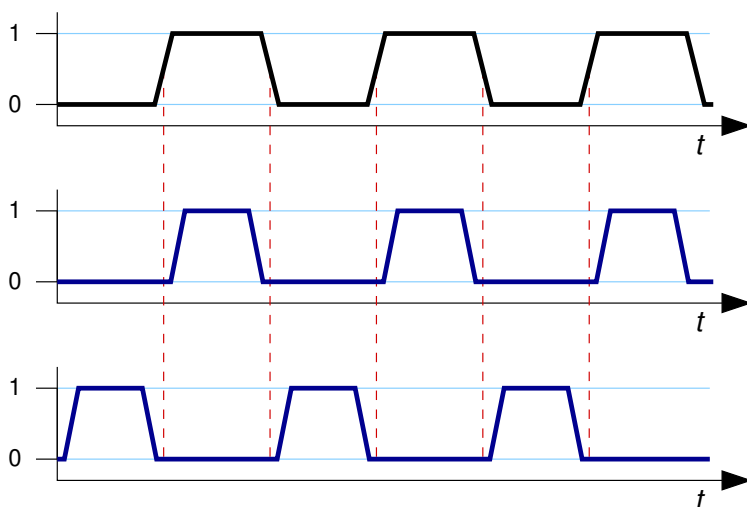
665



Zweiphasentakt (cont.)

10.6 Schaltwerke - Taktschemata

64-040 Rechnerstrukturen und Betriebssysteme



- ▶ 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

A. Mäder

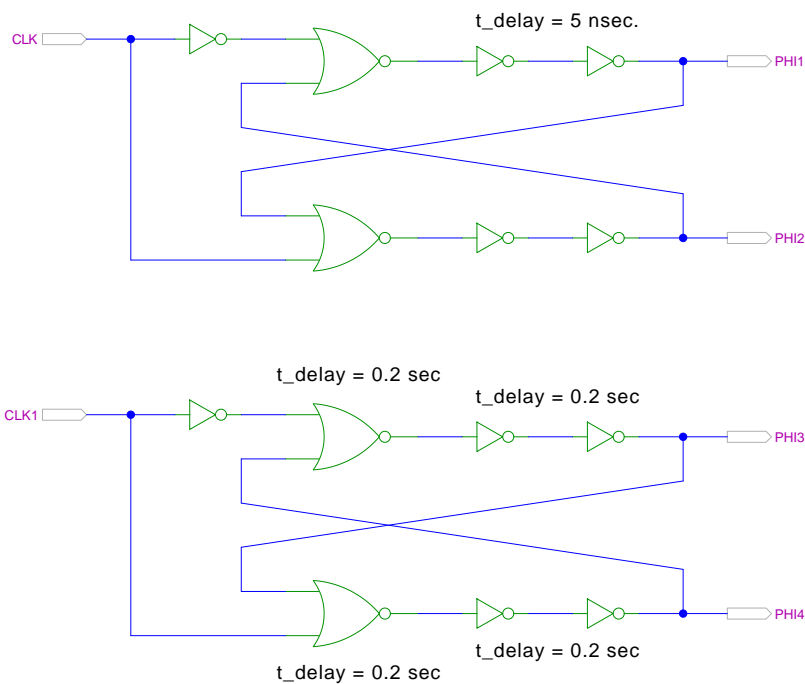
666



Zweiphasentakt: Erzeugung

10.6 Schaltwerke - Taktschemata

64-040 Rechnerstrukturen und Betriebssysteme



[HenHA] Hades Demo: 12-gatedelay/40-tpcg/two-phase-clock-gen

A. Mäder

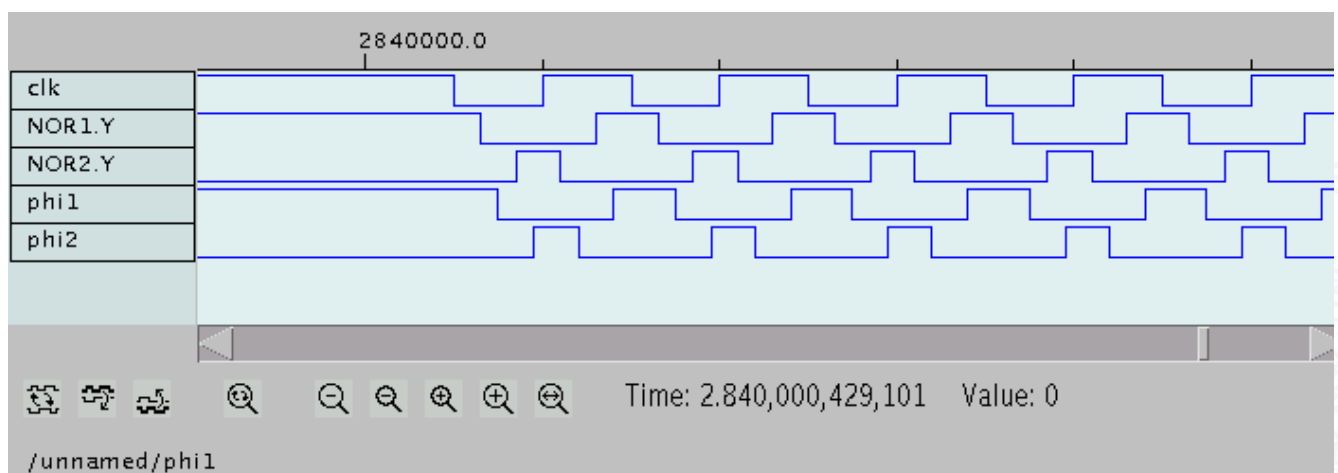
667



Zweiphasentakt: Erzeugung (cont.)

10.6 Schaltwerke - Taktschemata

64-040 Rechnerstrukturen und Betriebssysteme



- ▶ Verzögerungen geeignet wählen
 - ▶ Eins-Phasen der beiden Takte Φ_1 und Φ_2 sauber getrennt
- ⇒ nicht-überlappende Taktimpulse zur Ansteuerung von Schaltungen mit 2-Phasen-Taktung

A. Mäder

668



Beschreibung von Schaltwerken

10.7 Schaltwerke - Beschreibung von Schaltwerken

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ viele verschiedene Möglichkeiten
- ▶ graphisch oder textuell

- ▶ algebraische Formeln/Gleichungen
- ▶ Flusstafel und Ausgangstafel

- ▶ Zustandsdiagramm
- ▶ State-Charts (hierarchische Zustandsdiagramme)

- ▶ Programme (Hardwarebeschreibungssprachen)



Flusstafel und Ausgangstafel

10.7 Schaltwerke - Beschreibung von Schaltwerken

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ **Flusstafel**
Tabelle für die Folgezustände als Funktion des aktuellen Zustands und der Eingabe
= beschreibt das δ -Schaltnetz

- ▶ **Ausgangstafel**
Tabelle für die Ausgabewerte als Funktion des aktuellen Zustands
(und der Eingabe [Mealy-Modell])
= beschreibt das λ -Schaltnetz

- ▶ entsprechen Funktionstabellen von Schaltnetzen
- ▶ meistens in einer gemeinsamen Tabelle zusammengefasst



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

▶ Ausgangstafel

Zustand	Codierung		Ausgänge		
	z_1	z_0	<i>rt</i>	<i>ge</i>	<i>gr</i>
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

- ▶ Funktionstabellen für 2+3 Schaltfunktionen
- ▶ Minimierung z.B. mit KV-Diagrammen



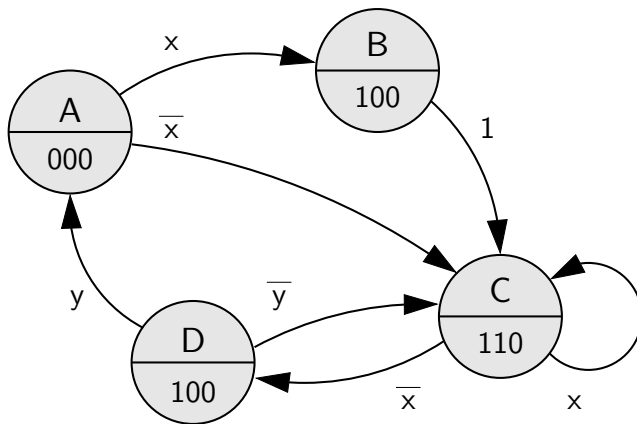
Zustandsdiagramm

- ▶ **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 von Zustand (Knoten) und Input ab, sie wird deshalb an den Kanten notiert

siehe auch en.wikipedia.org/wiki/State_diagram



Zustandsdiagramm: Moore-Automat



Zustand



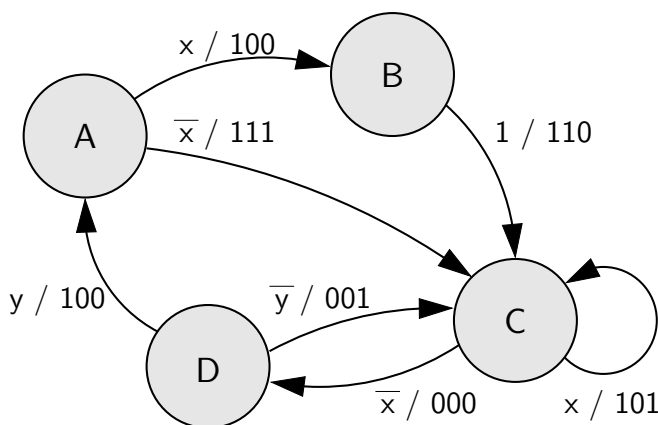
Übergang



- ▶ Ausgangswerte hängen ausschließlich vom Zustand ab
- ▶ können deshalb 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



Zustand



Übergang



- ▶ Ausgangswerte hängen nicht nur vom Zustand sondern auch von der Eingabe ab
- ▶ Ausgangswerte an den zugehörigen Kanten notieren
- ▶ übliche Notation: *Eingangsbelegung / Ausgangswerte*



„State-Charts“

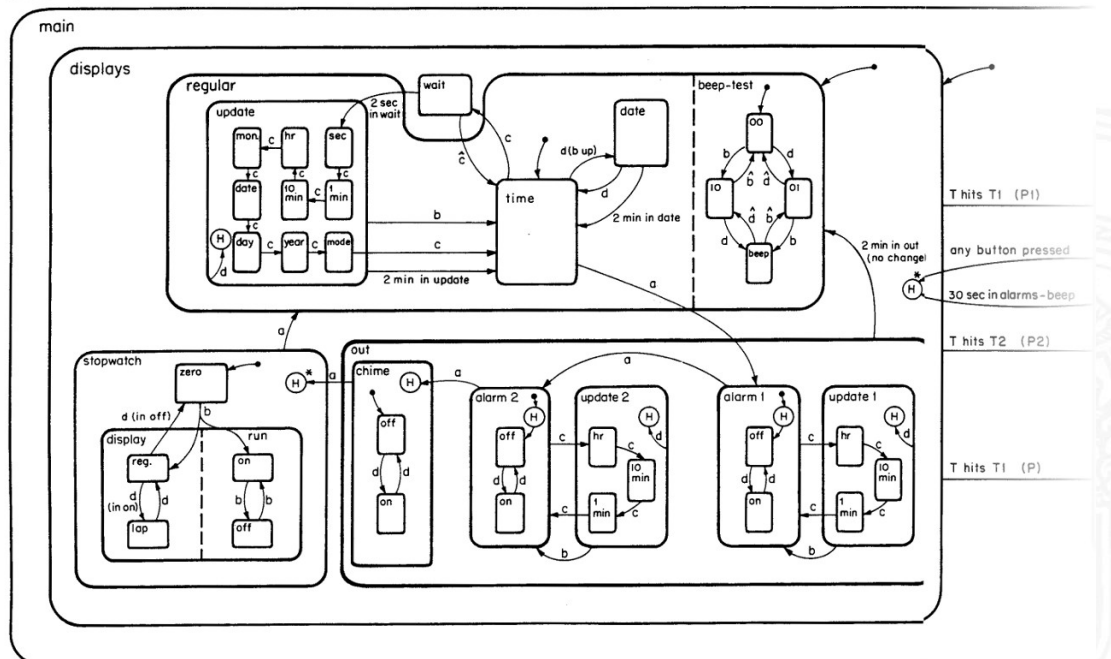
- ▶ erweiterte Zustandsdiagramme
 - 1. Hierarchien von Zuständen/Automaten erlauben Abstraktion
 - ▶ Knoten repräsentieren entweder einen Zustand
 - ▶ oder einen eigenen (Unter-) Automaten
 - ▶ *History-, Default-Mechanismen*
 - 2. Nebenläufigkeit durch mehrere parallel arbeitende FSMs
 - 3. Timer: Zustände können nach max. Zeit verlassen werden
-
- ▶ 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 Citizen quartz multi-alarm



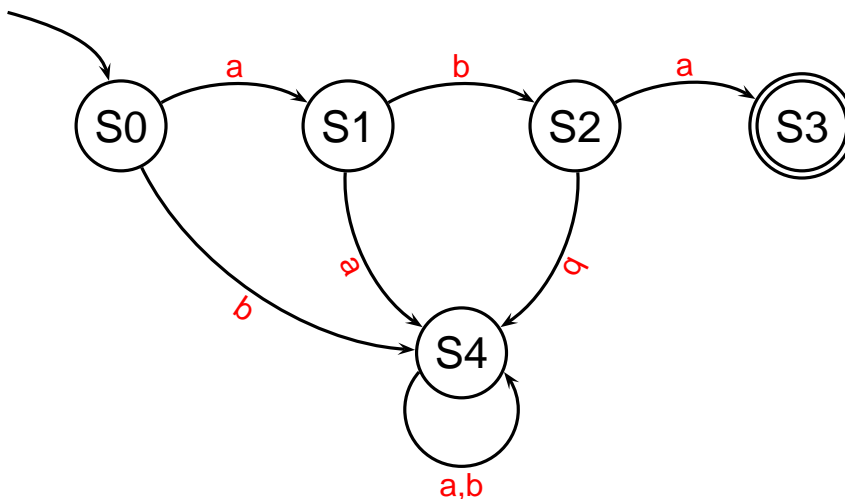


- ▶ eines der **grundlegenden Konzepte der Informatik**
- ▶ Modellierung, Entwurf und Simulation
 - ▶ zeitliche Abfolgen interner Systemzustände
 - ▶ bedingte Zustandswechsel
 - ▶ Reaktionen des Systems auf „Ereignisse“
 - ▶ Folgen von Aktionen
 - ▶ ...
- ▶ weitere „*spezielle*“ Anwendungsszenarien
 - ▶ verteilte Systeme (Client-Server etc.)
 - ▶ Echtzeitsysteme, ggf. mit Erweiterungen
 - ▶ eingebettete Systeme
 - ▶ ...

zahlreiche Beispiele

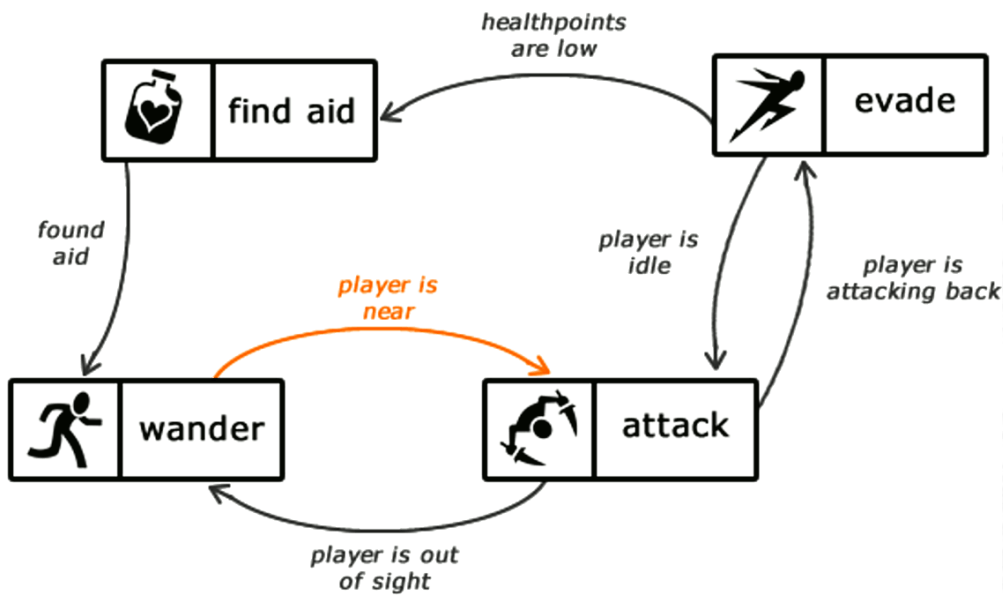


- ▶ in der Programmierung ...
Erkennung des Worts: „a b a“





Game-Design: Verhalten eines Bots

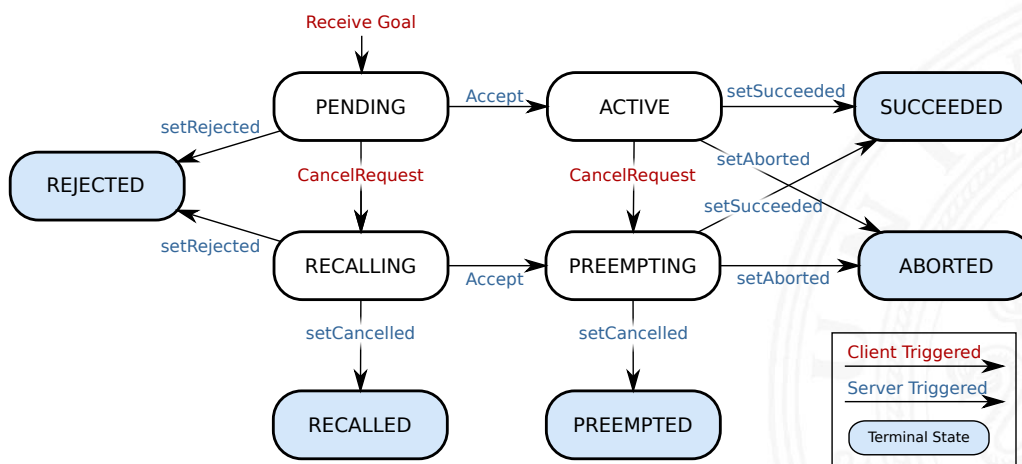


code.tutsplus.com/finite-state-machines-theory-and-implementation--gamedev-11867t



- ▶ Beschreibung von Protokollen
- ▶ Verhalten verteilter Systeme: Client-Server Architektur

Server State Transitions

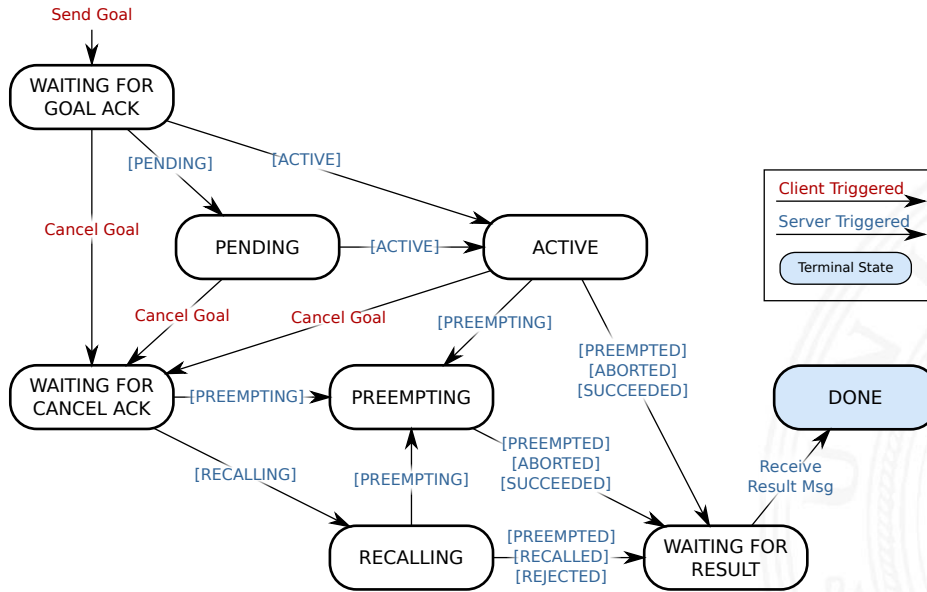


wiki.ros.org/actionlib/DetailedDescription



Endliche Automaten (cont.)

Client State Transitions

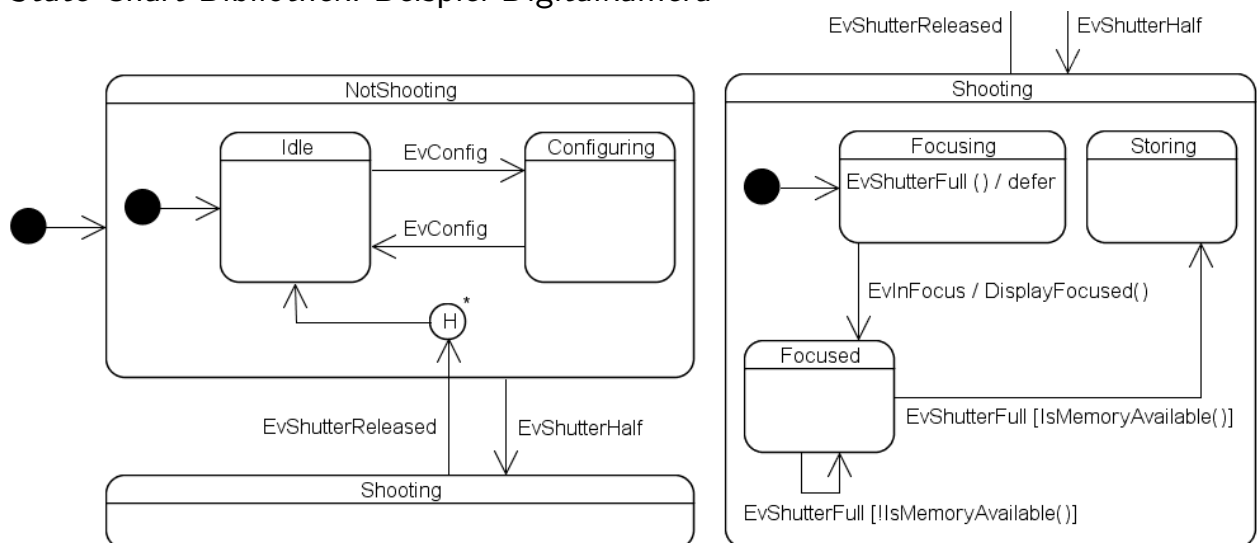


wiki.ros.org/actionlib/DetailedDescription



Endliche Automaten (cont.)

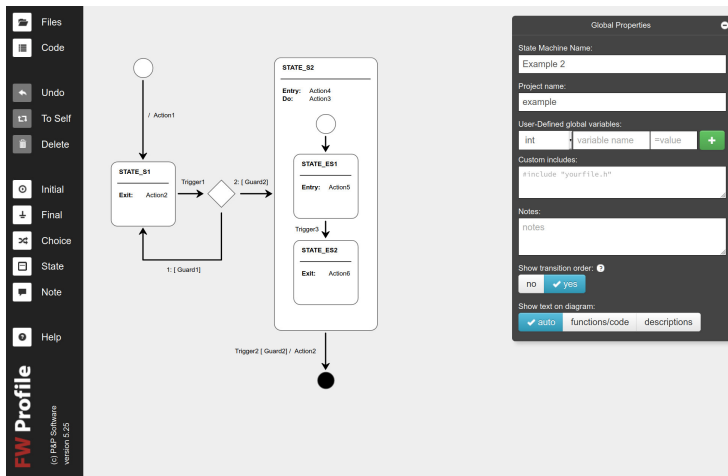
- ▶ Unterstützung durch Bibliotheken und Werkzeuge
- State-Chart Bibliothek: Beispiel Digitalkamera



www.boost.org/doc/libs/1_83_0/libs/statechart/doc



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



- ▶ Beschreibung eines Schaltwerks als Programm
 - ▶ normale Hochsprachen C, Java
 - ▶ spezielle Bibliotheken für normale Sprachen SystemC, Hades
 - ▶ Hardwarebeschreibungssprachen Verilog, VHDL

- ▶ Eigenschaften von Hardwarebeschreibungssprachen
 - ▶ Abstraktion und Hierarchie
 - ▶ Modellierung paralleler Abläufe
 - ▶ detailliertes Zeitverhalten von Schaltungen / Leitungen
 - ▶ Systembeschreibung von Hardware und Software

- ▶ Vergleichbar mit parallelen Programmiersprachen, z.B.: Ada
- ▶ hier nicht weiter vertieft. ...
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 behavior 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 behavior;

```



Entwurf von Schaltwerken: sechs Schritte

10.8 Schaltwerke - Entwurf von Schaltwerken

64-040 Rechnerstrukturen und Betriebssysteme

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

10.8 Schaltwerke - Entwurf von Schaltwerken

64-040 Rechnerstrukturen und Betriebssysteme

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
⇒ Widerspruch
- ▶ keine Übergänge bei bestimmten Eingangsbelegungen
⇒ 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?

$$\forall i : \bigvee_{j=0}^{2^p-1} h_{ij}(x) = 1$$



Überprüfung der Widerspruchsfreiheit

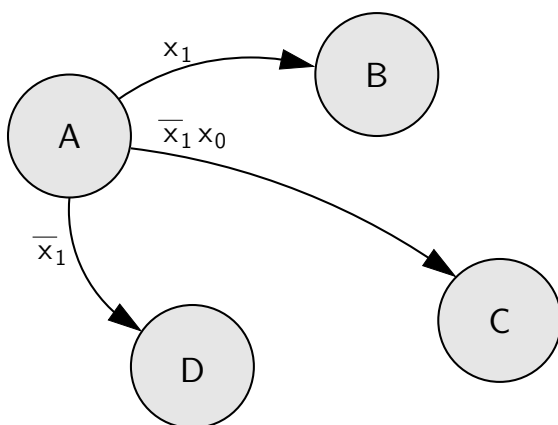
p Zustände, Zustandsdiagramm mit Kanten $h_{ij}(x)$:
Übergang von Zustand i nach Zustand j unter Belegung x

- für jeden Zustand überprüfen:
kommen alle (spezifizierten) Eingangsbelegungen nur einmal vor?

$$\forall i : \bigvee_{j,k=0, j \neq k}^{2^p-1} (h_{ij}(x) \wedge h_{ik}(x)) = 0$$



Vollständigkeit und Widerspruchsfreiheit: Beispiel



- für Zustand A

Vollständigkeit

$$x_1 \vee \overline{x_1} x_0 \vee \overline{x_1} = 1 \quad \text{ok}$$

Widerspruchsfreiheit

$$x_1 \wedge \overline{x_1} x_0 = 0 \quad \text{ok}$$

alle Paare testen

$$x_1 \wedge \overline{x_1} = 0 \quad \text{ok}$$

$$\overline{x_1} x_0 \wedge \overline{x_1} \neq 0 \quad \text{zwei Übergänge aktiv! } x_1 = 0, x_0 = 1$$



- ▶ 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



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



Schaltwerkentwurf: Ampel – Variante 1

- ▶ vier Zustände, Codierung als 2-bit Vektor (z_1, z_0)
- ▶ Fluss- und Ausgangstafel für binäre Zustandskodierung

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

$$z_1^+ = (z_1 \wedge \overline{z_0}) \vee (\overline{z_1} \wedge z_0) = z_1 \oplus z_0$$

$$z_0^+ = \overline{z_0}$$

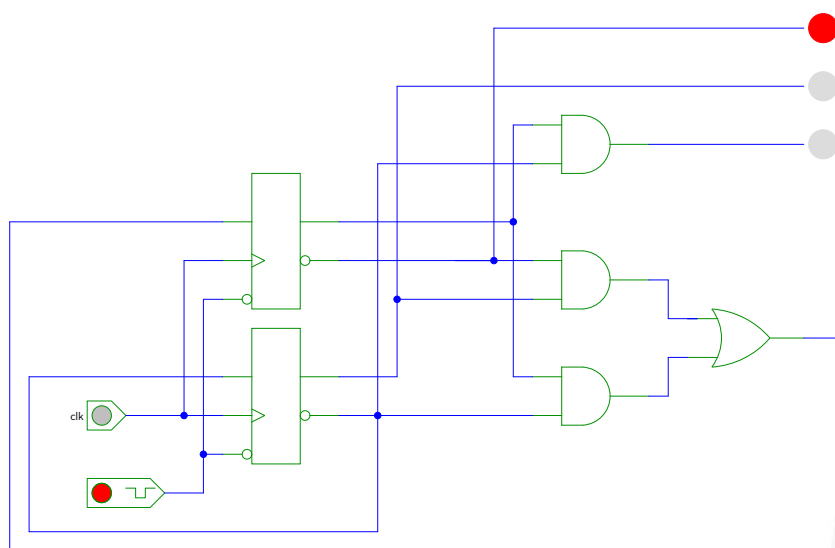
$$rt = \overline{z_1}$$

$$ge = z_0$$

$$gr = (z_1 \wedge \overline{z_0})$$



Schaltwerkentwurf: 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 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

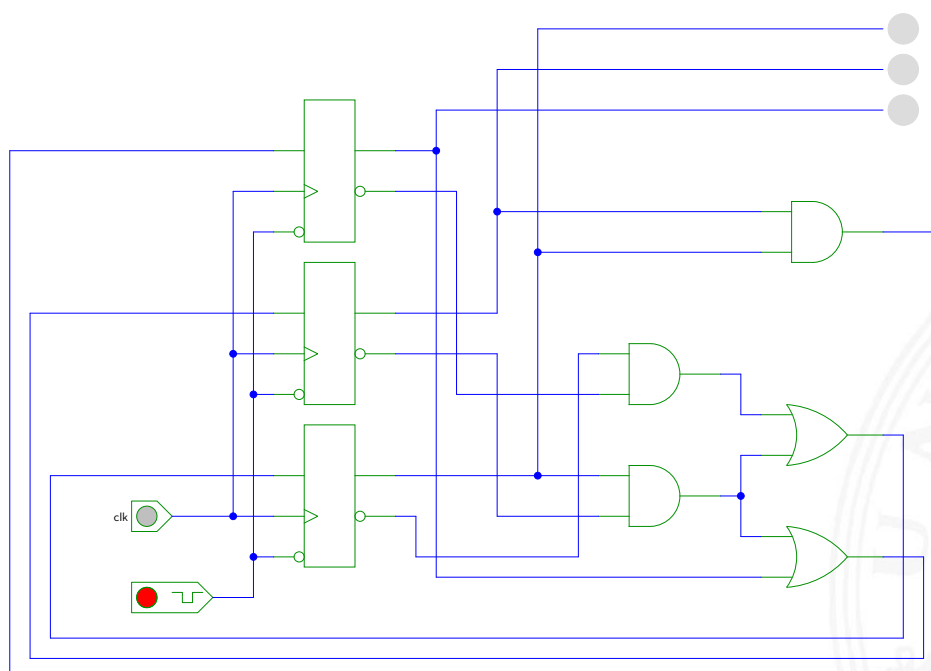
- ▶ benutzt 1-bit zusätzlich für die Zustände
- ▶ Ausgangsfunktion λ minimal: entfällt
- ▶ Übergangsfunktion δ : $z_2^+ = (z_1 \wedge z_0)$

$$z_1^+ = z_2 \vee (\bar{z}_1 \wedge z_0)$$

$$z_0^+ = (\bar{z}_2 \wedge \bar{z}_0) \vee (\bar{z}_1 \wedge z_0)$$



Schaltwerksentwurf: Ampel – Variante 2 (cont.)





Schaltwerksentwurf: Ampel – Variante 3

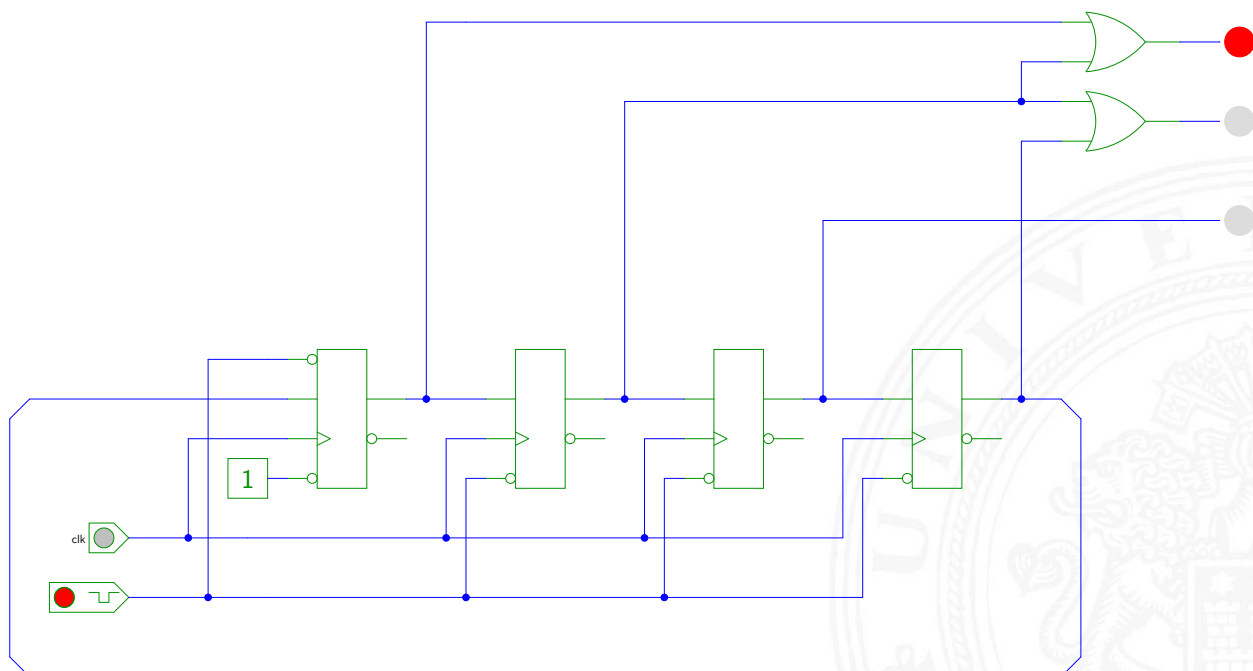
- ▶ vier Zustände, Codierung als 4-bit *one-hot* Vektor (z_3, z_2, z_1, z_0)
- ▶ Beispiel für die Zustandscodierung

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.)





Schaltwerksentwurf: Ampel – Zusammenfassung

10.9.1 Schaltwerke – Beispiele – Ampelsteuerung

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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

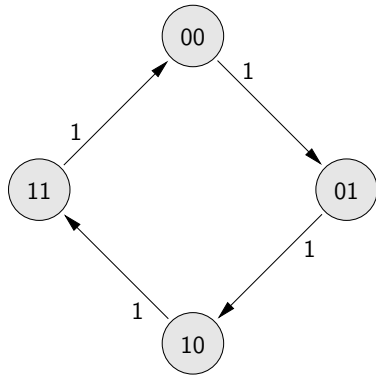
10.9.2 Schaltwerke – Beispiele – Zählschaltungen

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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)



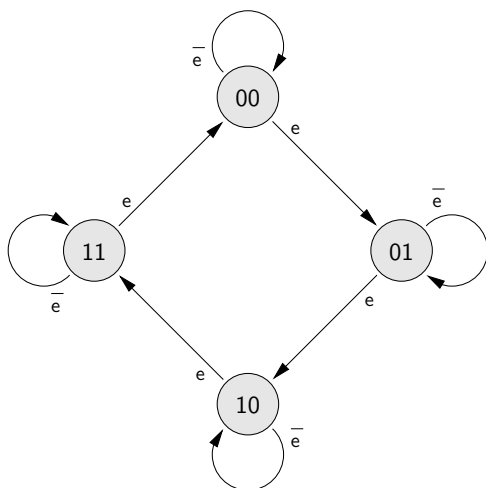
2-bit Zähler: Zustandsdiagramm



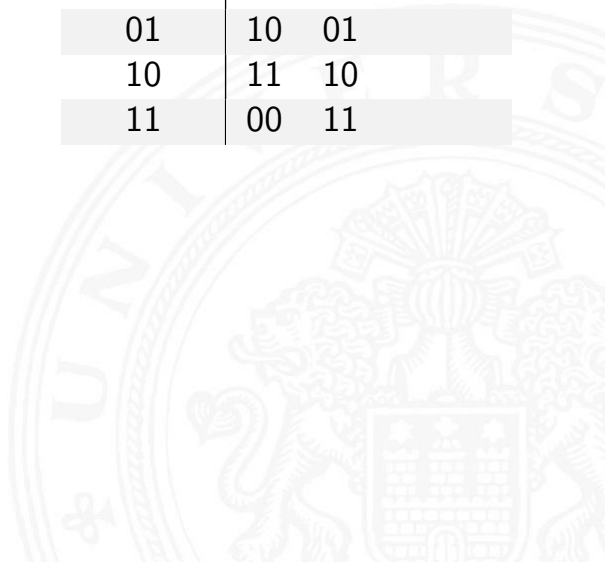
► Zähler als „trivialer“ endlicher Automat



2-bit Zähler mit Enable: Zustandsdiagramm, Flusstafel

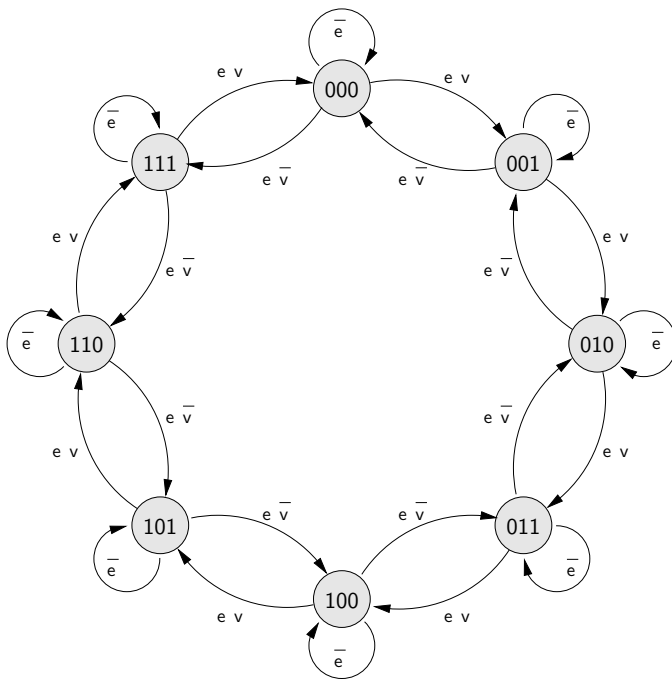


Eingabe	e	\bar{e}
Zustand	Folgezustand	
00	01	00
01	10	01
10	11	10
11	00	11





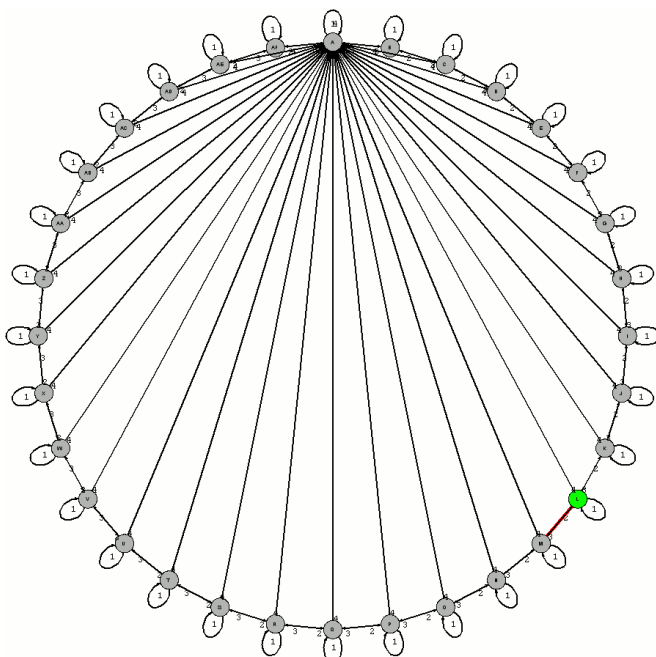
3-bit Zähler mit Enable, Vor-/Rückwärts



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

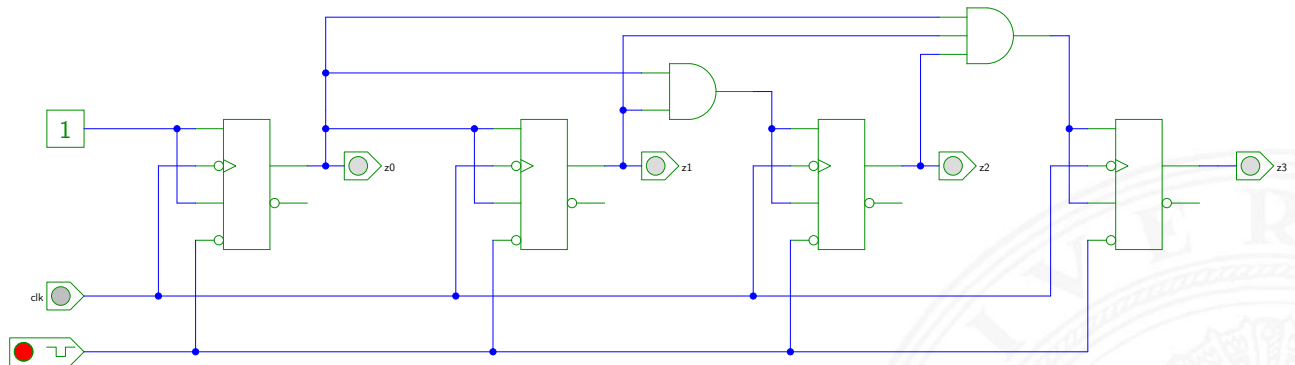


Zustand	Index der Eingabe			
	1	2	3	4
A	A	B	AF	A
B	B	C	A	A
C	C	D	B	A
D	D	E	C	A
E	E	F	D	A
F	F	G	E	A
G	G	H	F	A
H	H	I	G	A
I	I	J	H	A
J	J	K	I	A
K	K	L	J	A
L	L	M	K	A
M	M	N	L	A
N	N	O	M	A
O	O	P	N	A
P	P	Q	O	A
Q	Q	R	P	A
R	R	S	Q	A
S	S	T	R	A
T	T	U	S	A
U	U	V	T	A
V	V	W	U	A
W	W	X	V	A
X	X	Y	W	A
Y	Y	Z	X	A
Z	Z	AA	Y	A
AA	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

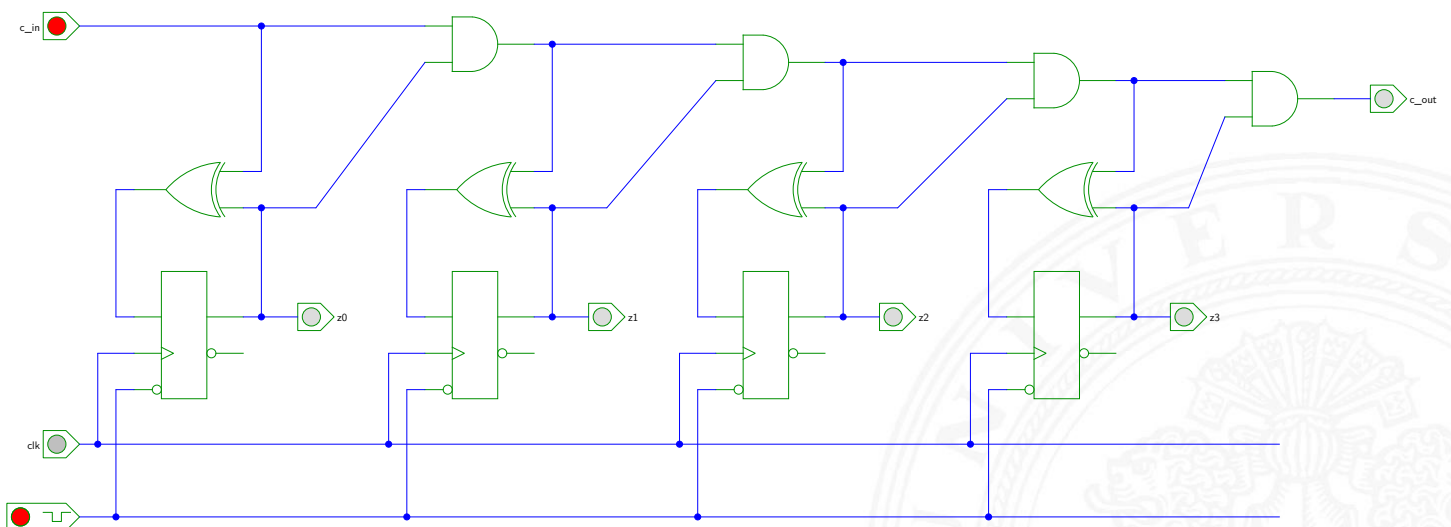


[HenHA] Hades Demo: 30-counters/30-sync/sync

- ▶ $J_0 = K_0 = 1$: Ausgang z_0 wechselt bei jedem Takt
- ▶ $J_i = K_i = (z_0 z_1 \dots z_{i-1})$: Ausgang z_i wechselt, wenn alle niedrigeren Stufen 1 sind



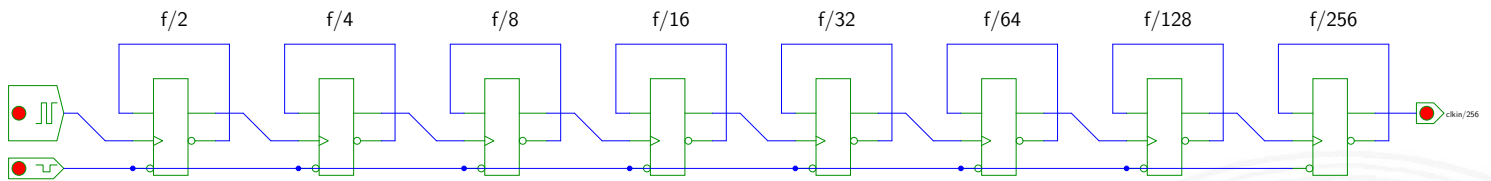
4-bit Binärzähler mit D-Flipflops (kaskadierbar)



[HenHA] Hades Demo: 30-counters/30-sync/sync-dff

- ▶ $D_0 = z_0 \oplus c_{in}$ wechselt bei Takt, wenn Carry-in $c_{in} = 1$
- ▶ $D_i = z_i \oplus (c_{in} z_0 z_1 \dots z_{i-1})$ wechselt, wenn alle niedrigeren Stufen und $c_{in} = 1$

Asynchroner n -bit Zähler/Teiler mit D-Flipflops

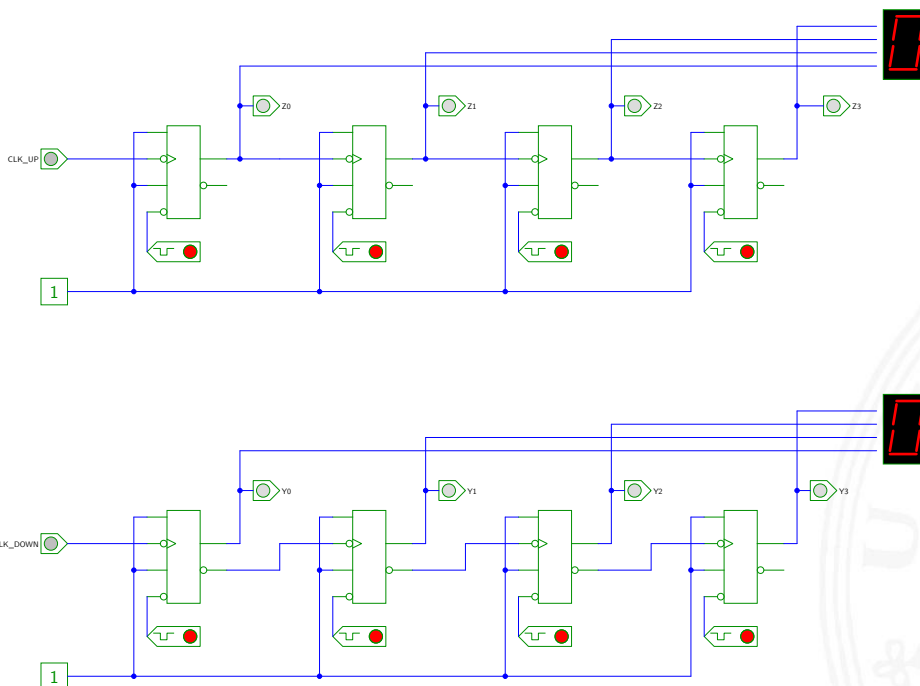


[HenHA] Hades Demo: 30-counters/20-async/counter-dff

- ▶ $D_i = \bar{z}_i$: jedes Flipflop wechselt bei seinem Taktimpuls
- ▶ Takteingang C_0 treibt nur das vorderste Flipflop
- ▶ $C_i = z_{i-1}$: 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



- [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 978-3-8272-9550-7
www.ibdr.de/Private/caemain.htm
- [WE93] N.H.E. Weste, K. Eshraghian:
Principles of CMOS VLSI design – A systems perspective.
2nd edition, Addison-Wesley, 1993. ISBN 978-0-201-53376-7
- [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



- [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, Vorlesungsskript.
tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1



Gliederung

11 Rechnerarchitektur I

64-040 Rechnerstrukturen und Betriebssysteme

1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. Logische Operationen
7. Codierung
8. Schaltfunktionen
9. Schaltnetze
10. Schaltwerke
- 11. Rechnerarchitektur I**
Motivation



Gliederung (cont.)

11 Rechnerarchitektur I

64-040 Rechnerstrukturen und Betriebssysteme

von-Neumann Rechner
Beschreibungsebenen
 Software
 HW Abstraktionsebenen
Hardwarestruktur
 Speicherbausteine
 Busse
 Mikroprogrammierung
 Beispielsystem: ARM
Wie rechnet ein Rechner?
Literatur

12. Instruction Set Architecture
13. Assembler-Programmierung
14. Rechnerarchitektur II





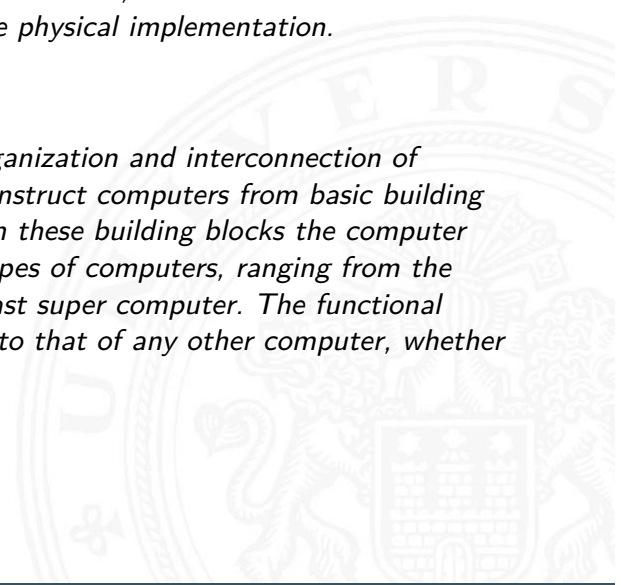
15. Betriebssysteme



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. 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.*





Was ist Rechnerarchitektur? (cont.)

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

11.2 Rechnerarchitektur I - von-Neumann Rechner

64-040 Rechnerstrukturen und Betriebssysteme

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

A. Mäder

719

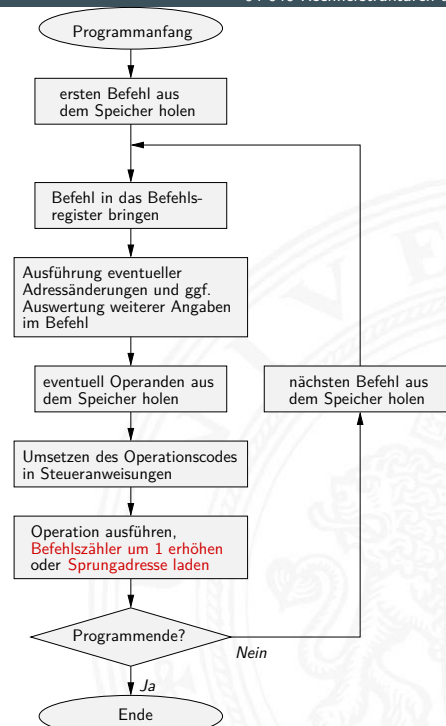


Wdh. Programmverarbeitung

11.2 Rechnerarchitektur I - von-Neumann Rechner

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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



A. Mäder

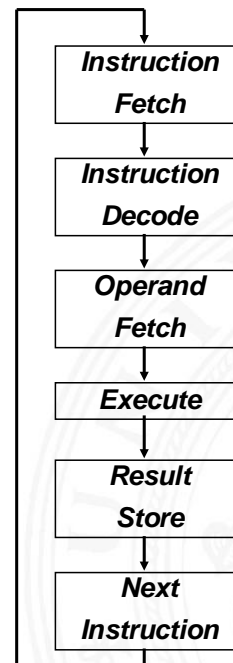
720



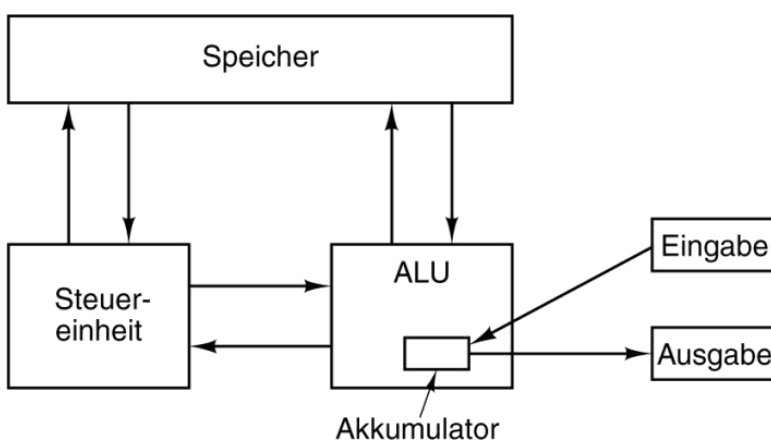
Wdh. Programmverarbeitung (cont.)

► Ausführungszyklus

1. Befehl aus Programmspeicher holen
2. auszuführende Aktionen und Länge der Instruktion bestimmen, ggf. Worte nachladen
3. Operanden ermitteln und laden
4. Ergebnis der Operation berechnen bzw. Status ermitteln
5. Ergebnisse für später abspeichern
6. 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.)

11.2 Rechnerarchitektur I - von-Neumann Rechner

64-040 Rechnerstrukturen und Betriebssysteme

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

A. Mäder

723

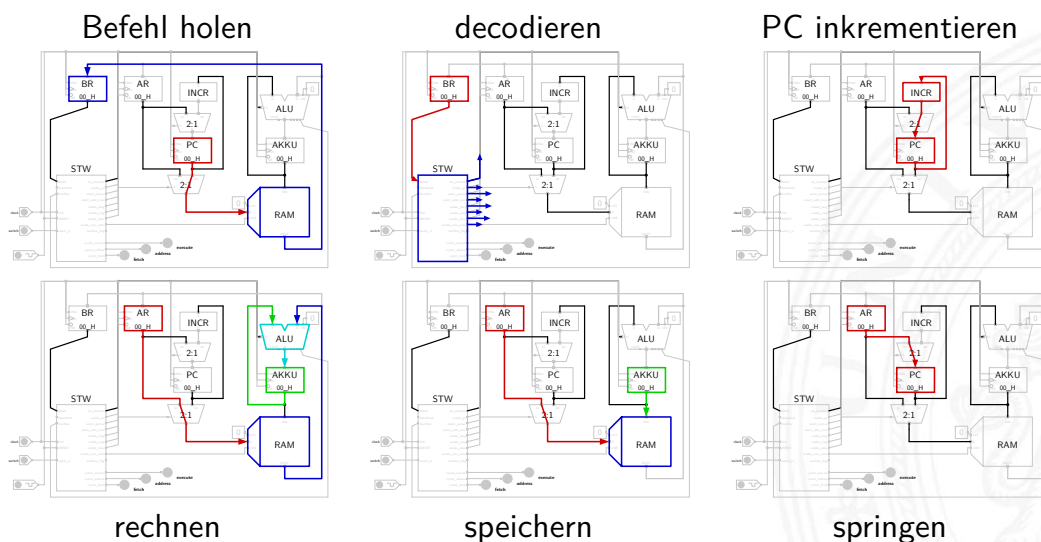


Wdh. Hardwareaufbau / Rechnerarchitektur

11.2 Rechnerarchitektur I - von-Neumann Rechner

64-040 Rechnerstrukturen und Betriebssysteme

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



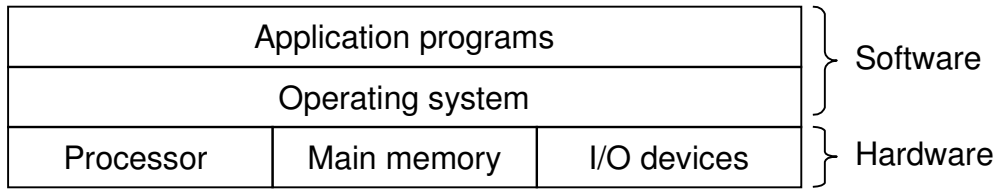
A. Mäder

724



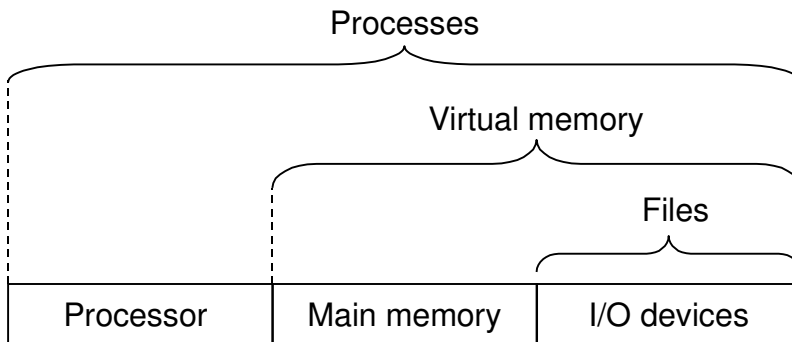
Beschreibungsebenen

- ▶ Schichten-Ansicht: Software – Hardware



[BO15]

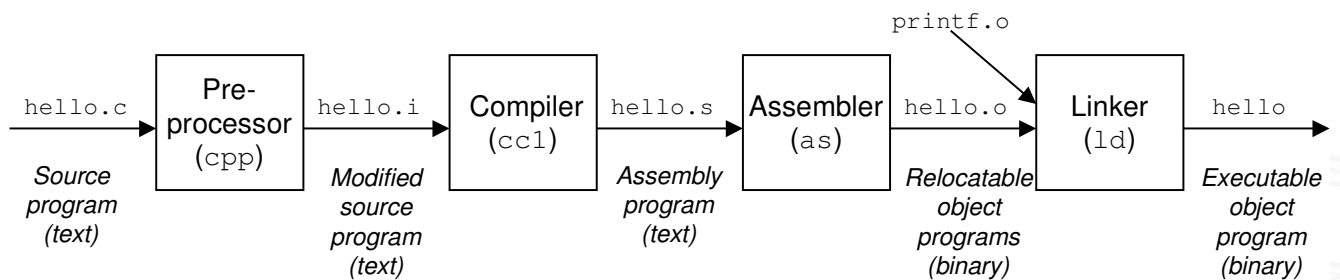
- ▶ Abstraktionen durch Betriebssystem



[BO15]



Das Compilierungssystem

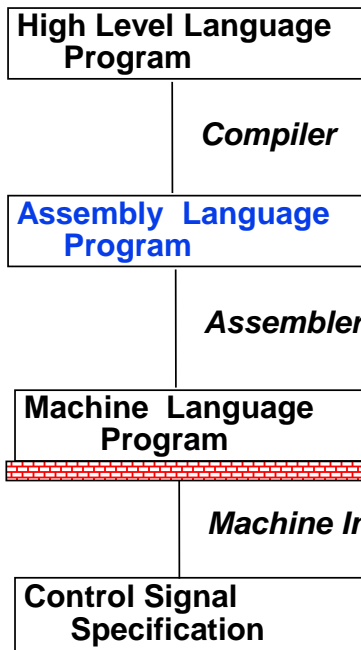


[BO15]

- ▶ verschiedene Repräsentationen des Programms
 - ▶ Hochsprache
 - ▶ Assembler
 - ▶ Maschinsprache
- ▶ Ausführung der Maschinsprache
 - ▶ von-Neumann Zyklus: Befehl holen, decodieren, ausführen
 - ▶ reale oder virtuelle Maschine



Das Compilingssystem (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
  
```

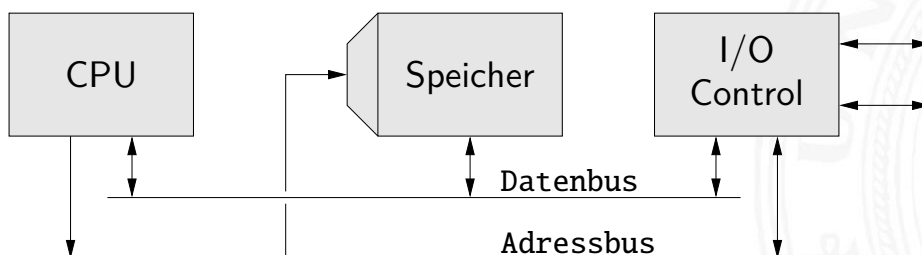
[PH22]



Abstraktion im VLSI-Entwurf

Hardware Abstraktionsebenen

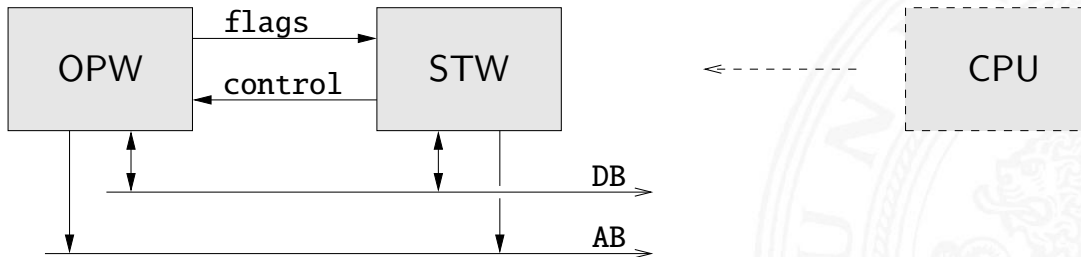
- keine einheitliche Bezeichnung in der Literatur
- ▶ Architekturebene
 - ▶ Funktion/Verhalten Leistungsanforderungen
 - ▶ Struktur Netzwerk
aus Prozessoren, Speicher, Busse, Controller ...
 - ▶ Nachrichten Programme, Protokolle
 - ▶ Geometrie Systempartitionierung





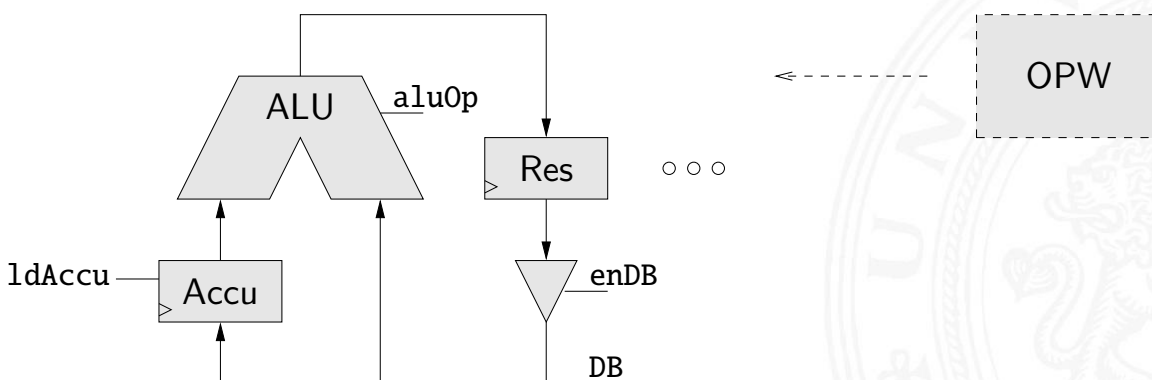
Abstraktion im VLSI-Entwurf (cont.)

- ▶ Hauptblockebene (Algorithmebene, funktionale Ebene)
 - ▶ Funktion/Verhalten Algorithmen, formale Funktionsmodelle
 - ▶ Struktur Blockschaltbild
aus Hardwaremodule, Busse ...
 - ▶ Nachrichten Prokoll
 - ▶ Geometrie Cluster



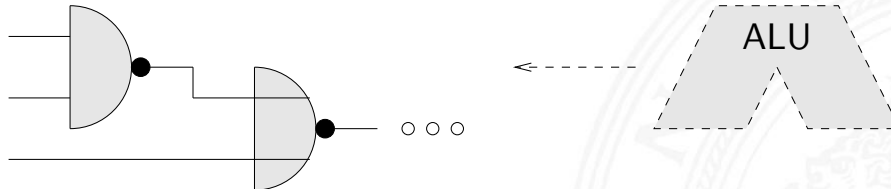
Abstraktion im VLSI-Entwurf (cont.)

- ▶ Register-Transfer Ebene
 - ▶ Funktion/Verhalten Daten- und Kontrollfluss, Automaten ...
 - ▶ Struktur RT-Diagramm
aus Register, Multiplexer, ALUs ...
 - ▶ Nachrichten Zahlencodierungen, Binärworte ...
 - ▶ Geometrie Floorplan

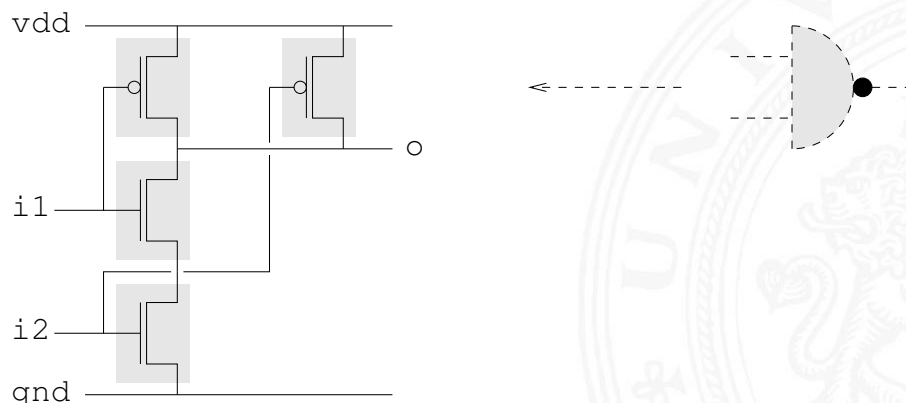




- ▶ Logikebene (Schaltwerkebene)
 - ▶ Funktion/Verhalten Boole'sche Gleichungen
 - ▶ Struktur Gatternetzliste, Schematic
aus Gatter, Flipflops, Latches ...
 - ▶ Nachrichten Bit
 - ▶ Geometrie Moduln



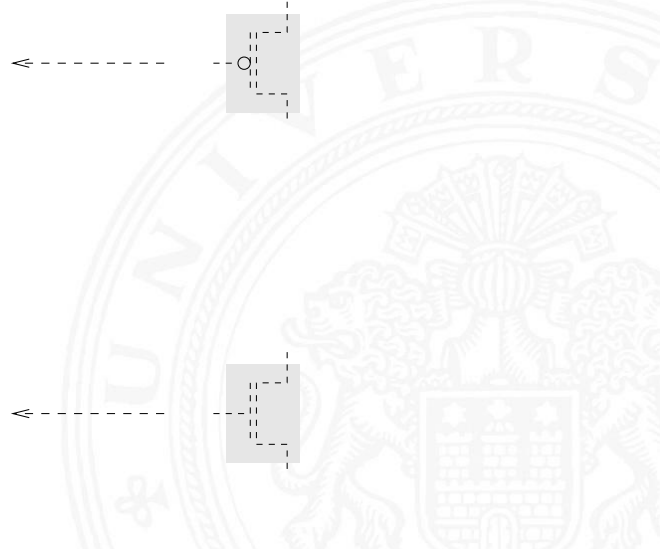
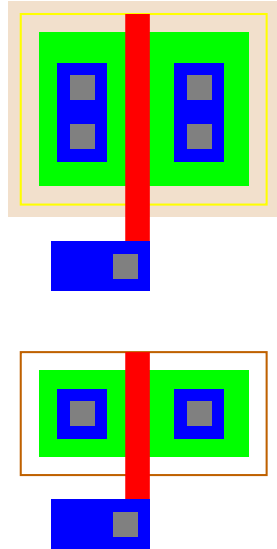
- ▶ elektrische Ebene (Schaltkreisebene)
 - ▶ Funktion/Verhalten Differentialgleichungen
 - ▶ Struktur elektrisches Schaltbild
aus Transistoren, Kondensatoren ...
 - ▶ Nachrichten Ströme, Spannungen
 - ▶ Geometrie Polygone, Layout → physikalische Ebene



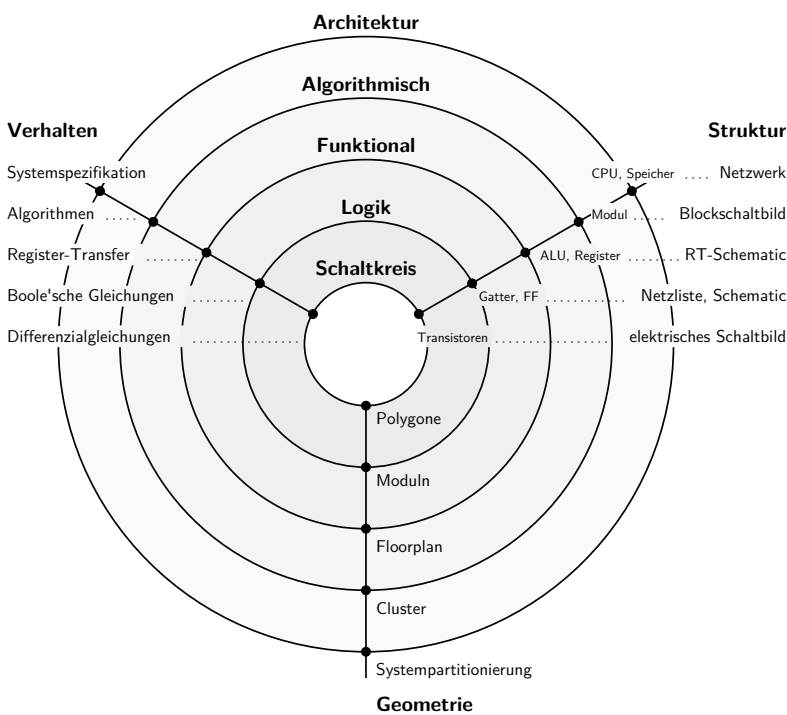


Abstraktion im VLSI-Entwurf (cont.)

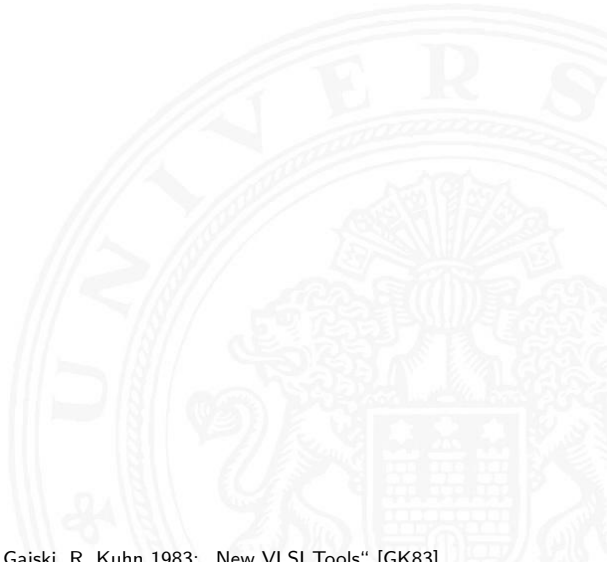
- ▶ physikalische Ebene (geometrische Ebene)
 - ▶ Funktion/Verhalten partielle DGL
 - ▶ Struktur Dotierungsprofile



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, als Verhaltensbeschreibung
 - ▶ Ende des Entwurfsprozesses ist das vollständige IC Layout für die Chipfertigung, Prüfmuster für Tests auf Fertigungsfehler oder Selbsttestmechanismen im IC
 - ... und die (erfolgreich) simulierte Netzliste mit Gatter- und Leitungsverzögerungen
 - ▶ Entwurfsprogramme („EDA“, *Electronic Design Automation*) dabei notwendig: 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



- ▶ bis jetzt
 - ▶ Gatter und Schaltnetze
 - ▶ Flipflops als einzelne Speicherglieder
 - ▶ Schaltwerke zur Ablaufsteuerung

- ▶ weitere Komponenten: Register-Transfer- und Hauptblockebene
 - ▶ Speicher
 - ▶ Busse, Bustiming
 - ▶ Mikroprogrammierung zur Ablaufsteuerung



- ▶ 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

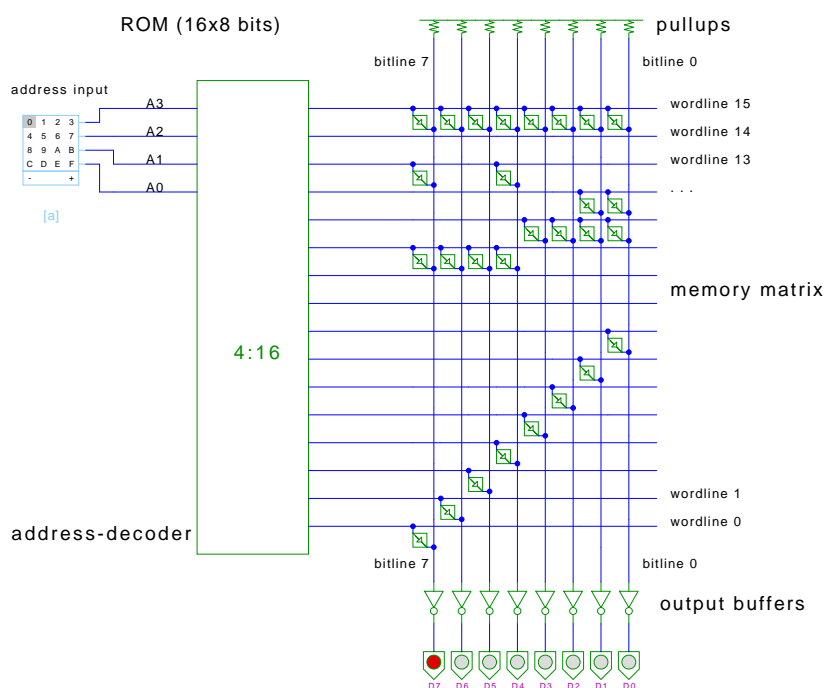


Speicherbausteine: Varianten

Typ	Kategorie	Löschen	byte-adressierbar	flüchtig	Typische Anwendung
SRAM	Lesen/Schreiben	elektrisch	ja	ja	Cache Speicher
DRAM	Lesen/Schreiben	elektrisch	ja	ja	Hauptspeicher (alt)
SDRAM	Lesen/Schreiben	elektrisch	ja	ja	Hauptspeicher
ROM	nur Lesen	—	nein	nein	Embedded (große Stückzahlen)
PROM	nur Lesen	—	nein	nein	Embedded (kleine Stückzahlen)
EPROM	vorw. Lesen	UV-Licht	nein	nein	Prototypen
EEPROM	vorw. Lesen	elektrisch	ja	nein	Prototypen
Flash	Lesen/Schreiben	elektrisch	nein	nein	Speicherkarten, SSDs, Mobile Geräte



ROM: Read-Only Memory



16 × 8 bit
 4-bit Adresse
 8-bit Datenwort



RAM: Random-Access Memory

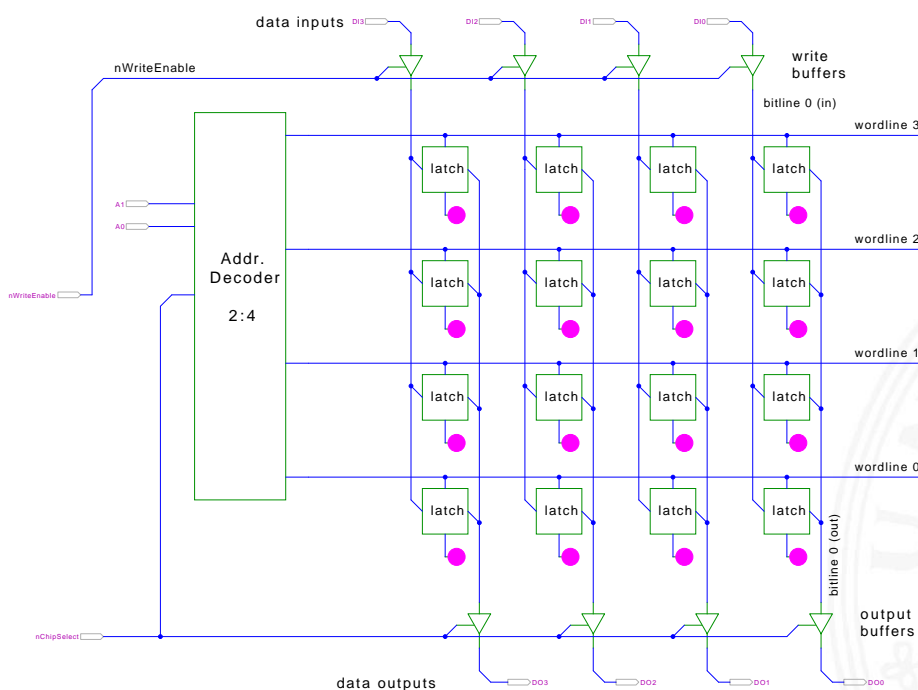
Speicher, der im Betrieb gelesen und geschrieben werden kann

- ▶ 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 \Rightarrow SRAM
 - ▶ dynamisches RAM: 1-Transistor Zelle \Rightarrow DRAM



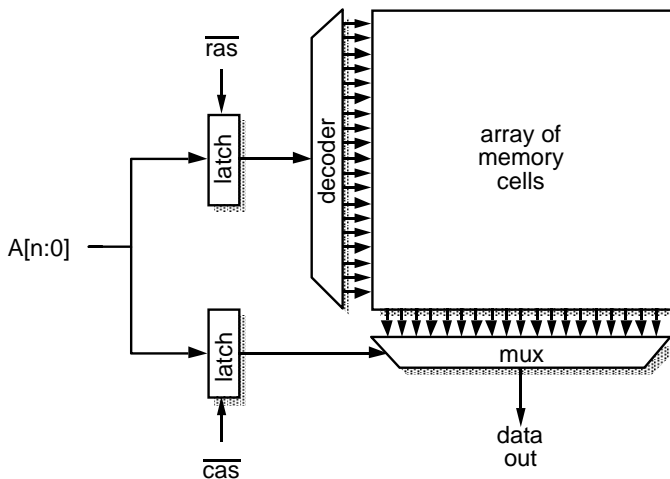
RAM: Blockschaltbild



4 × 4 bit
 2-bit Adresse
 4-bit Datenwort



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

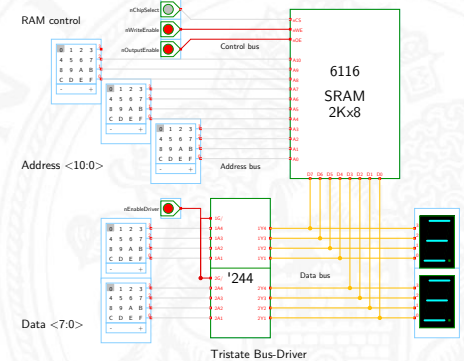


SRAM: Beispiel IC 6116

- ▶ integrierte Schaltung, 16 Ki bit Kapazität
- ▶ Organisation als 2 Ki Worte mit je 8-bit

- ▶ 11 Adresseingänge (A10...A0)
- ▶ 8 Anschlüsse für Tristate 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

- ▶ Hades-Demo zum Ausprobieren [HenHA]
 - ▶ Hades Demo: 40-memories/40-ram/demo-6116
 - ▶ Hades Demo: 40-memories/40-ram/two-6116



DRAM: dynamisches RAM

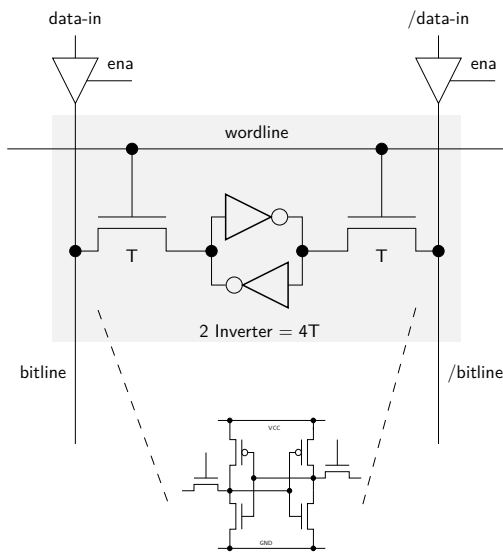
- ▶ Information wird in winzigen Kondensatoren gespeichert
- ▶ pro Bit je ein Transistor und Kondensator

- ▶ jeder Lesezugriff entlädt den Kondensator
- ▶ *Leseverstärker* zur Messung der Spannung auf der Bitline
Schwellwertvergleich zur Entscheidung logisch 0/1

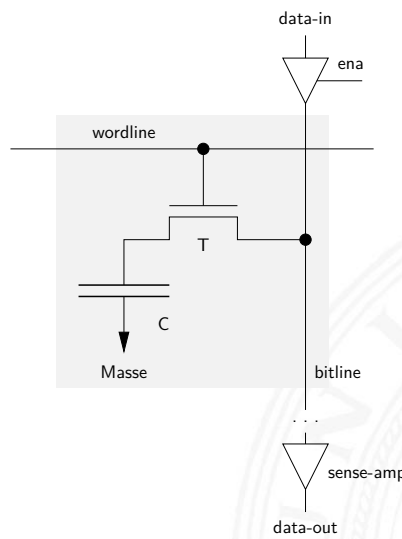
- Information muss anschließend neu geschrieben werden
- auch ohne Lese- oder Schreibzugriff ist regelmäßiger *Refresh* notwendig,
wegen Selbstentladung (Millisekunden)
- 10 × langsamer als SRAM
- + DRAM für hohe Kapazität optimiert, minimaler Platzbedarf



SRAM vs. DRAM



- ▶ 6 Transistoren/bit
- ▶ statisch (kein refresh)
- ▶ schnell



- ▶ 1 Transistor/bit
- ▶ $C = 5 \text{ fF} \approx 47 \text{ 000 Elektronen}$
- ▶ langsam (sense-amp)



DRAM: Stacked- und Trench-Zelle

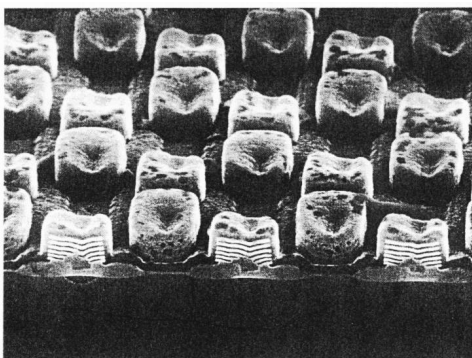
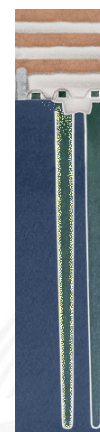
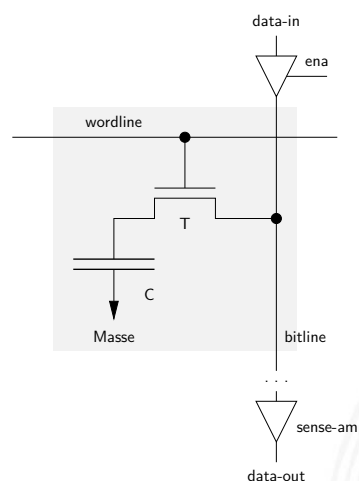


Abb. 7: Prototyp von Speicherzellen (Stapelkondensatoren) für zukünftige Speicherchips wie den Ein-Gigabit-Chip. Da für DRAM-Chips eine minimale Speicherkapazität von 25 fF notwendig ist, bringt es erhebliche Platzvorteile, die Kondensatorelemente vertikal übereinander zu stapeln. Die Dicke der Schichten beträgt etwa 50 nm. (Foto: Siemens)

Siemens 1 Gbit DRAM

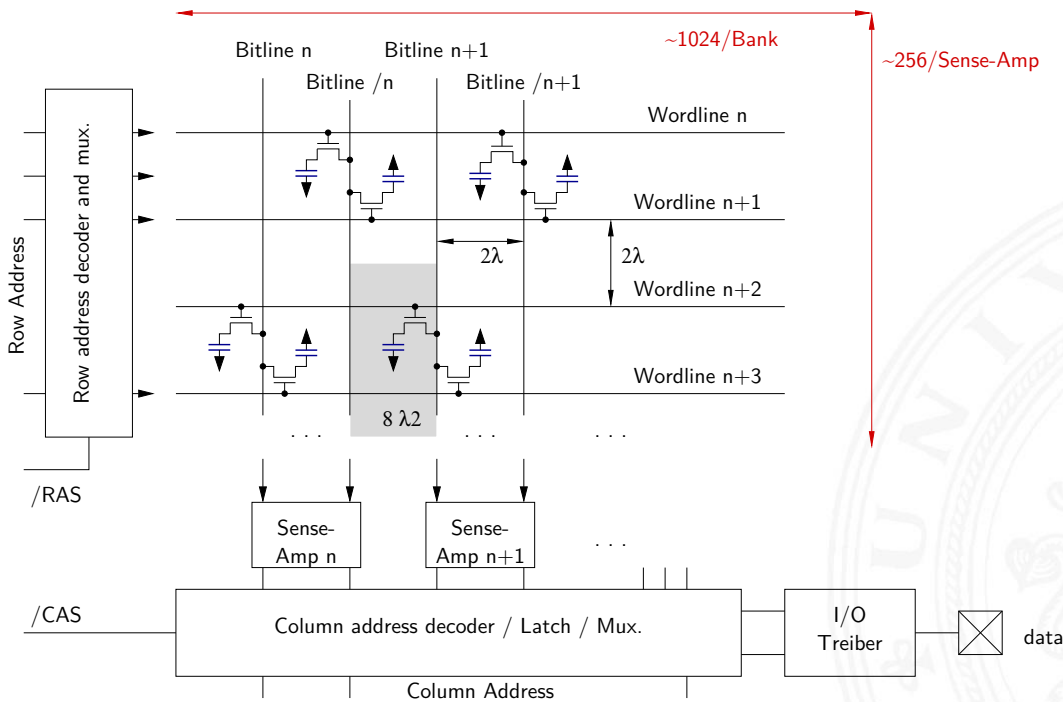
- ▶ zwei Bauformen: „stacked“ und „trench“
- ▶ Kondensatoren
 - ▶ möglichst kleine Fläche
 - ▶ Kapazität gerade ausreichend



IBM CMOS-6X embedded DRAM



DRAM: Layout



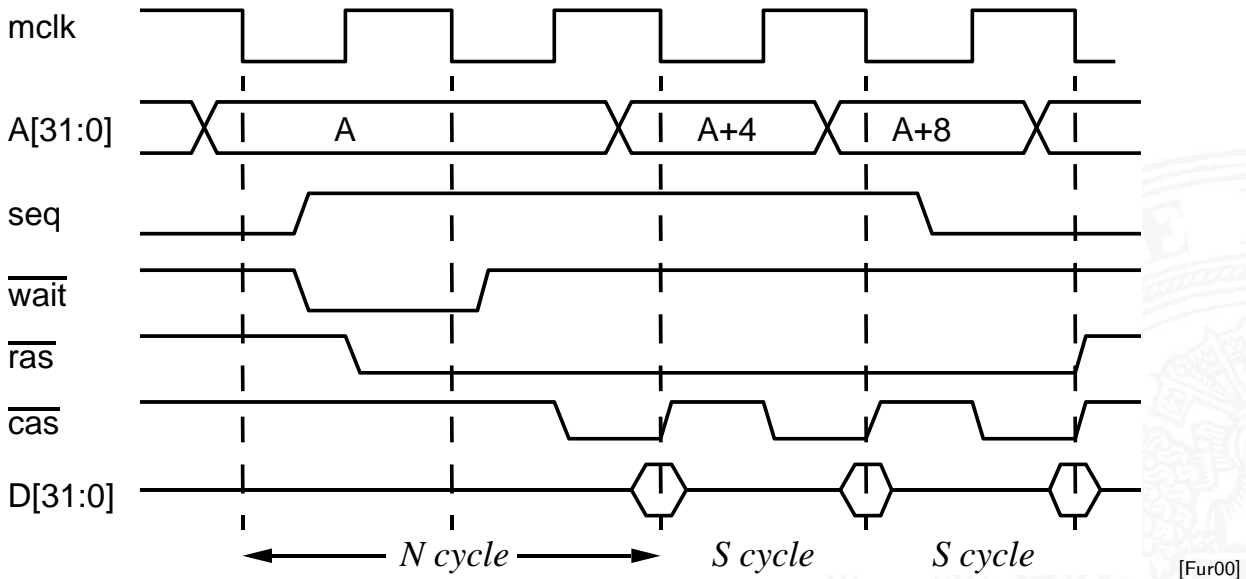
DRAM: Varianten

- ▶ 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... GDDR6X (*Graphics DDR*)
derzeit bis 168 GByte/sec
 - ▶ HBM... HBM3 (*High Bandwidth Memory*)
derzeit bis 819,2 GByte/sec
 - ▶ HMC... HMC2 (*Hybrid Memory Cube*)
derzeit bis 480 GByte/sec

... DDR6 × 2
... GDDR7 × 1,5

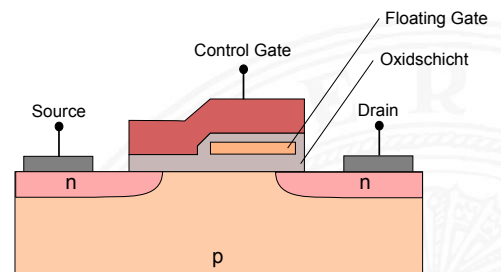


SDRAM: Lesezugriff auf sequenzielle Adressen



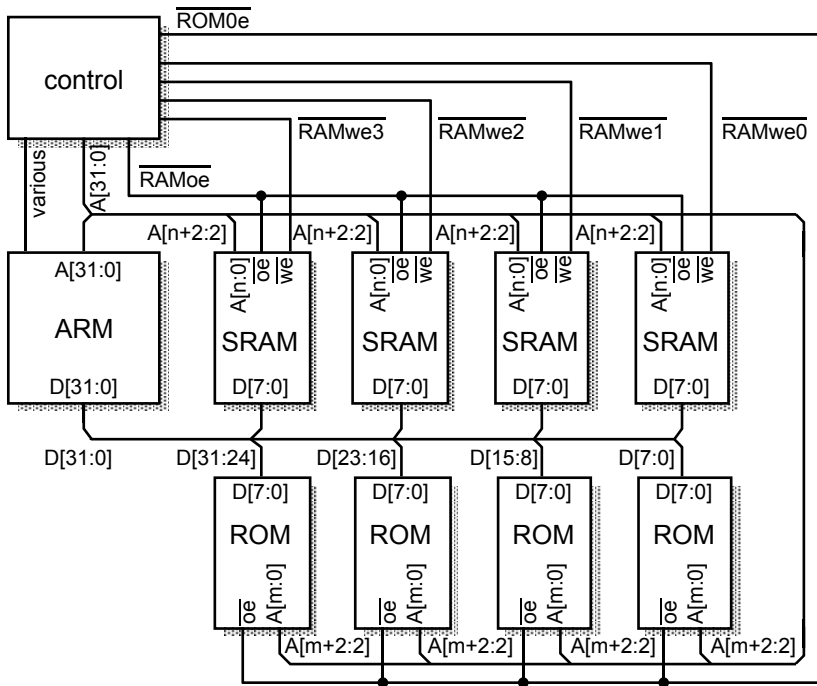
Flash

- ▶ 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

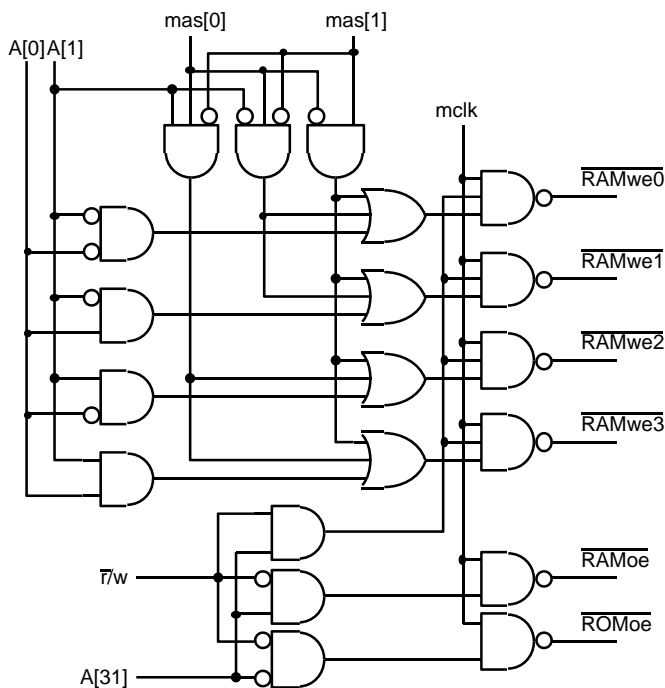


32-bit ARM Proz.
 4 × 8-bit SRAMs
 4 × 8-bit ROMs

[Fur00]



Typisches Speichersystem: Adressdecodierung



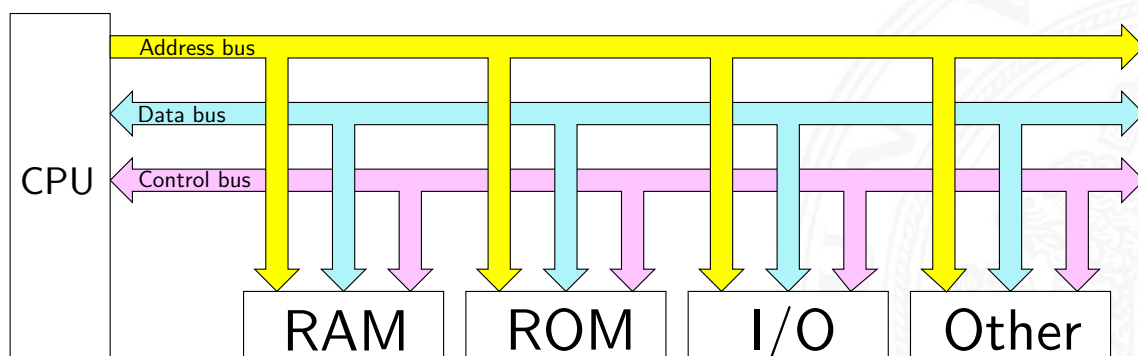
[Fur00]



- ▶ **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



- ▶ typische Aufgaben
 - ▶ Kernkomponenten (CPU, Speicher ...) miteinander verbinden
 - ▶ Verbindungen zu den Peripherie-Bausteinen
 - ▶ Verbindungen zu Systemmonitor-Komponenten
 - ▶ Verbindungen zwischen I/O-Controllern und -Geräten
 - ▶ ...





- ▶ viele Typen, standardisiert mit sehr unterschiedlichen Anforderungen
 - ▶ High-Performance (= Datendurchsatz)
 - ▶ einfaches Protokoll, billige Komponenten
 - ▶ Multi-Master-Fähigkeit, zentrale oder dezentrale Arbitrierung
 - ▶ Echtzeitfähigkeit, Daten-Streaming
 - ▶ wenig Leitungen bis zu Zweidraht-Bussen: I²C, SPI, System-Management-Bus ...
 - ▶ lange Leitungen: EIA-485, RS-232, Ethernet ...
 - ▶ Funkmedium: WLAN, Bluetooth (logische Verbindung)

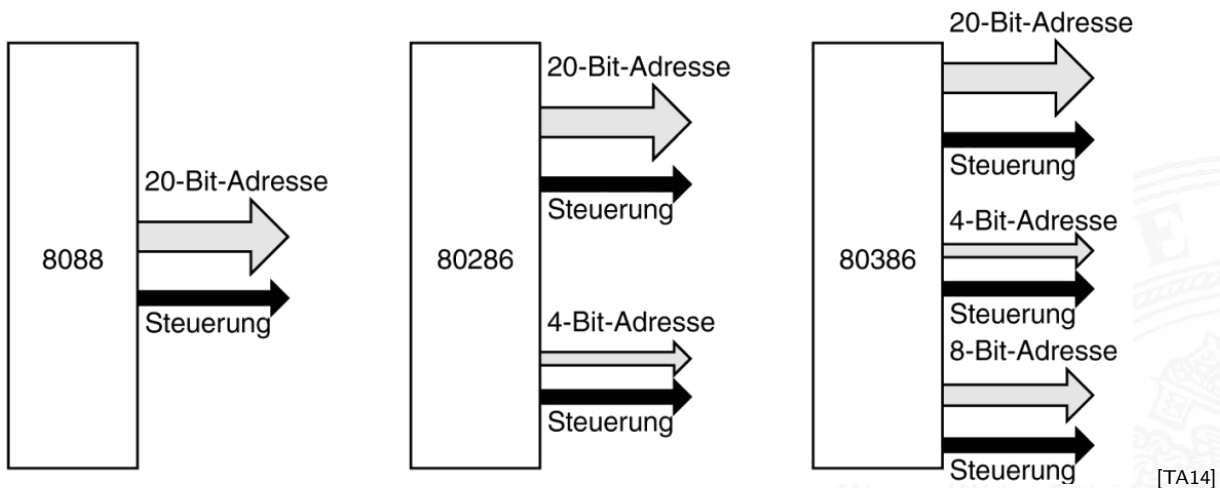


typisches n -bit Mikroprozessor-System:

- ▶ 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 Busse sowohl für Adressen als auch Daten ⇒ zusätzliches Steuersignal zur Auswahl Adressen/Daten



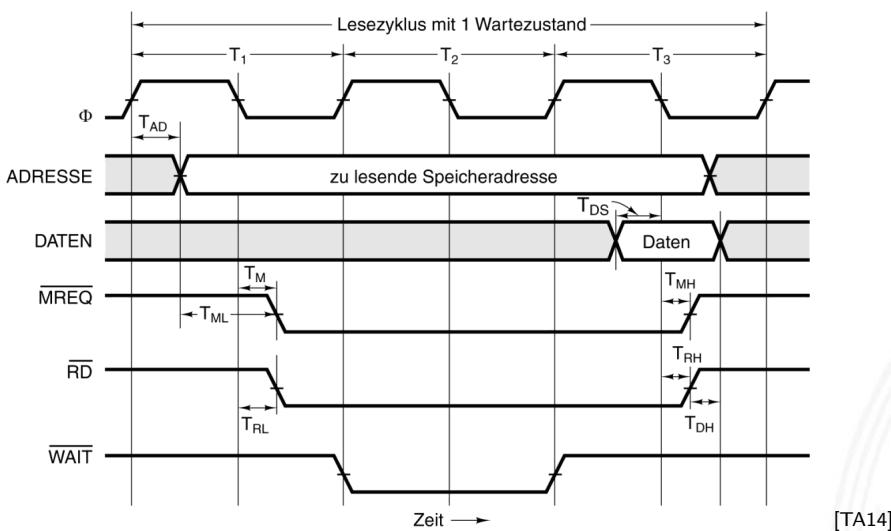
Adressbus: Evolution beim Intel x86



- ▶ 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



- ▶ 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



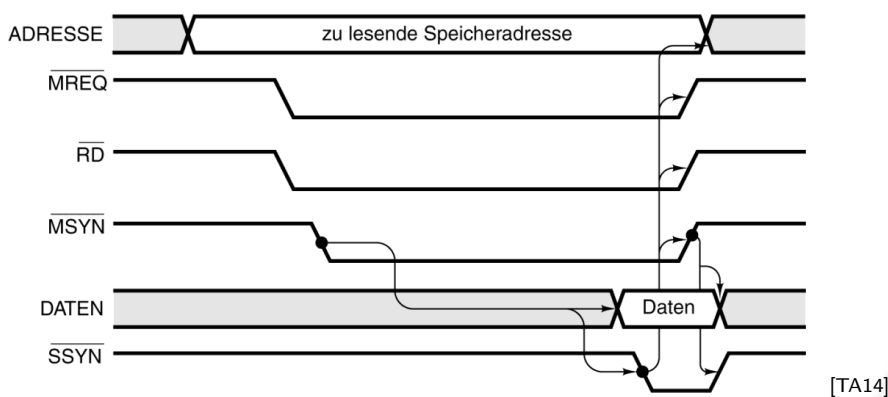
Synchroner Bus: Timing (cont.)

- ▶ 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} \overline{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



- ▶ Steuersignale \overline{MSYN} : Master fertig
 \overline{SSYN} : Slave fertig
- ▶ flexibler für Geräte mit stark unterschiedlichen Zugriffszeiten



- ▶ mehrere Komponenten wollen Übertragung initiieren, aber nur ein Transfer zur Zeit
- ▶ der Bus Zugriff muss serialisiert werden

1. zentrale Arbitrierung

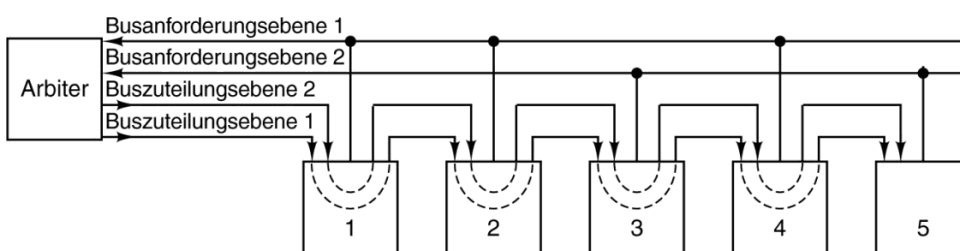
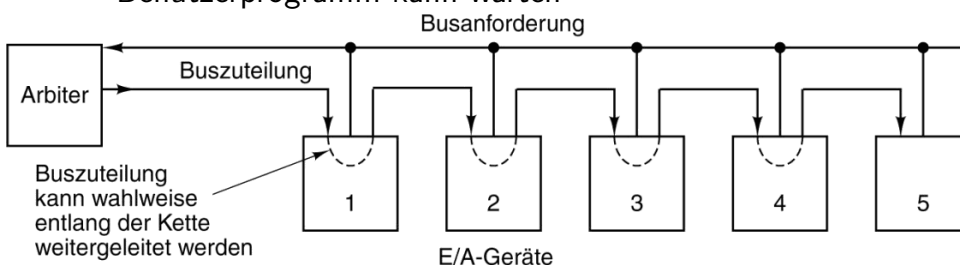
- ▶ Arbiter gewährt Bus-Requests
- ▶ Strategien
 - ▶ Prioritäten für verschiedene Geräte
 - ▶ „round-robin“ Verfahren
 - ▶ „Token“-basierte Verfahren
 - ▶ usw.

2. dezentrale Arbitrierung

- ▶ protokollbasiert
- ▶ Beispiel
 - ▶ Komponenten sehen ob Bus frei ist
 - ▶ wenn ja: Start der Übertragung; wenn nein: warten
 - ▶ zur Kollisionserkennung: Daten vom Bus lesen
 - ▶ bei Inkonsistenzen: Übertragung abbrechen und „später“ erneut versuchen



- ▶ 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 ² C	100 Kbit/sec (Std.)	...	3,4 Mbit/sec (High Speed)
USB	1,5 Mbit/sec (1.x)	...	80 Gbit/sec (4.0v2)
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	2,0 Gbit/sec (1.x)	...	121 Gbit/sec (7.0) ×16
HyperTransport	25,6 Gbit/sec (1.0)	...	409,6 Gbit/sec (3.1)
NVLink	640,0 Gbit/sec (1.0)	...	1,8 Tbit/sec (4.0)
Infinity Fabric		...	4,1 Tbit/sec

- ▶ en.wikipedia.org/wiki/List_of_interface_bit_rates



Beispiel: PCI-Bus

Peripheral Component Interconnect (Intel 1991)

- ▶ 33 MHz Takt
 - ▶ 32-bit Bus-System
 - ▶ gemeinsame Adress-/Datenleitungen
 - ▶ Arbitrierung durch Bus-Master (die 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
- optional 66 MHz Takt
optional auch 64-bit



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.)

The screenshot shows the Windows Device Manager window with the 'PCI' category selected. The 'Intel Corporation Core Processor DMI' device is highlighted, and its properties are displayed in the right pane. The properties include:

- Geräteklasse: Unclassified device (0x00)
- Geräte-Unterklasse: Non-VGA unclassified device (0x00)
- Geräte-Programm...: Unbekannt (0x00)
- Revision: 0x10
- Hersteller: Intel Corporation (0x8086)
- Gerät: Core Processor DMI (0x0131)
- Subsystem: Device 0000 (0x0000.0x0000)
- Kontrolle: 0x0100
- Status: 0x0011
- Zwischenspeicher...: 0x00
- Latenz: 0
- Vorspann: 0x00
- Eingebauter Selb...: 0x00
- Adress-Zuordnun...: 0x00
- Erweiterungs-ROM: 0x00
- Fähigkeiten: 0x09
- Interrupt: 0x09
- Rohrer PCI-Eintrich...: 0x00

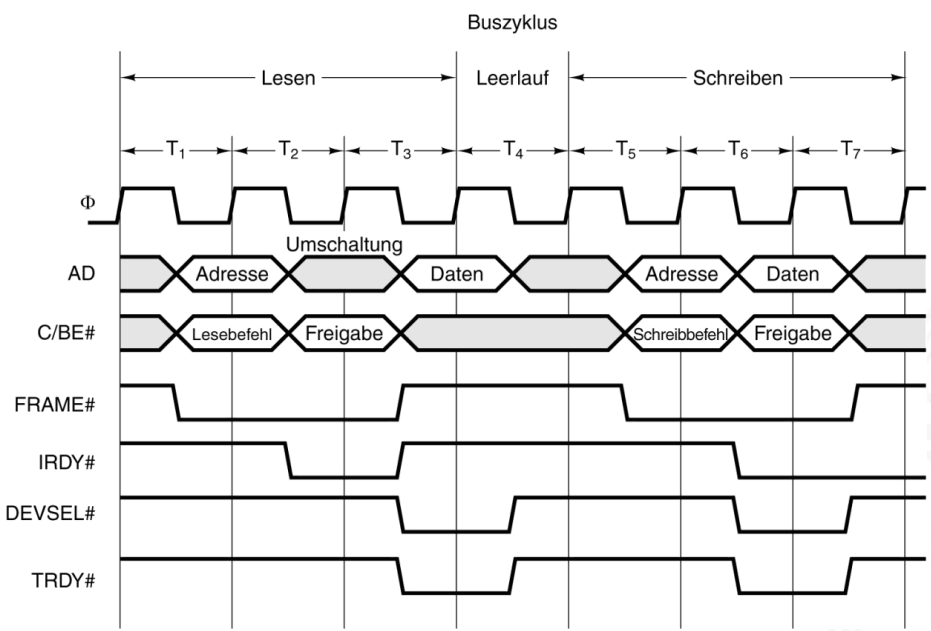


PCI-Bus: Leitungen („mindestens“)

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			Adressparitätsfehler 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



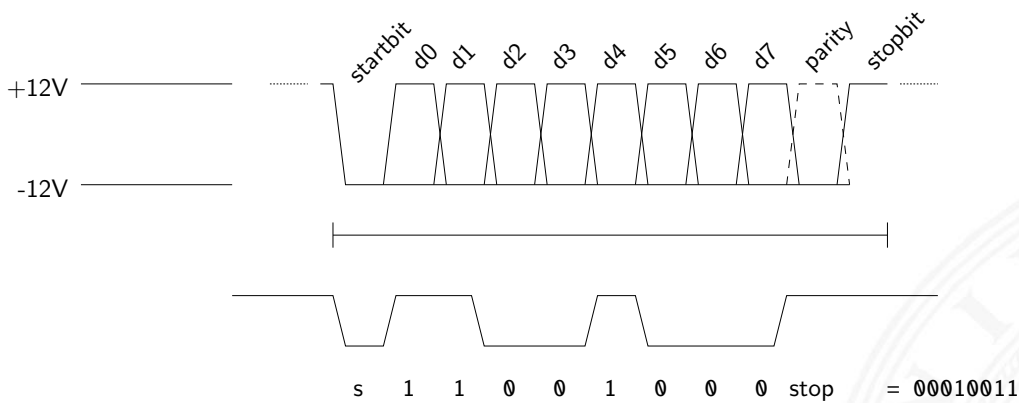
[TA14]



RS-232: Serielle Schnittstelle

11.4.2 Rechnerarchitektur I - Hardwarestruktur - Busse

64-040 Rechnerstrukturen und Betriebssysteme



- ▶ 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

A. Mäder

773



Ablaufsteuerung mit Mikroprogramm

11.4.3 Rechnerarchitektur I - Hardwarestruktur - Mikroprogrammierung

64-040 Rechnerstrukturen und Betriebssysteme

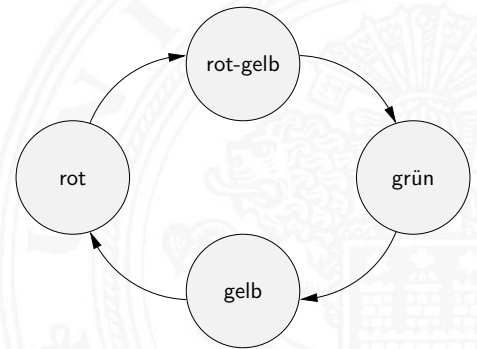
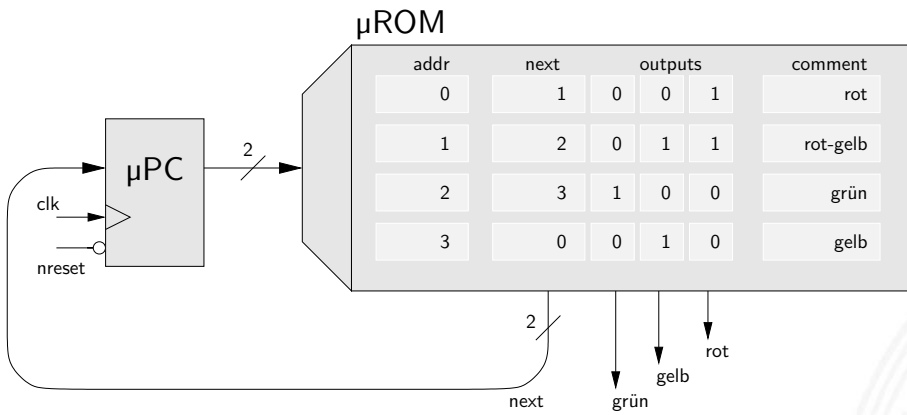
- ▶ als Alternative zu direkt entworfenen Schaltwerken
- ▶ *Mikroprogrammzähler* μPC : Register für aktuellen Zustand
- ▶ μPC adressiert den Mikroprogrammspeicher μROM
- ▶ μROM konzeptionell in mehrere Felder eingeteilt
 - ▶ die verschiedenen Steuerleitungen
 - ▶ ein oder mehrere Felder für Folgezustand
 - ▶ ggf. zusätzliche Logik und Multiplexer zur Auswahl unter mehreren Folgezuständen
 - ▶ ggf. Verschachtelung und Aufruf von Unterprogrammen: „nanoProgramm“
- ▶ siehe „Praktikum Rechnerstrukturen und Betriebssysteme“

A. Mäder

774



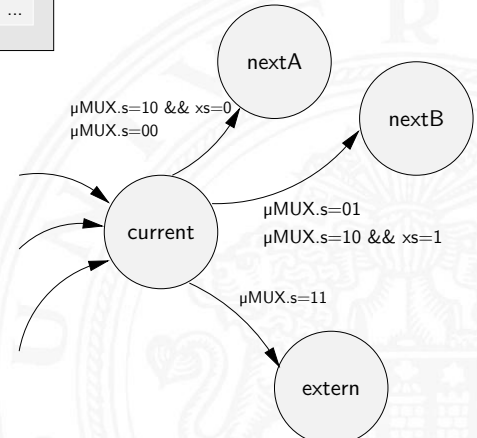
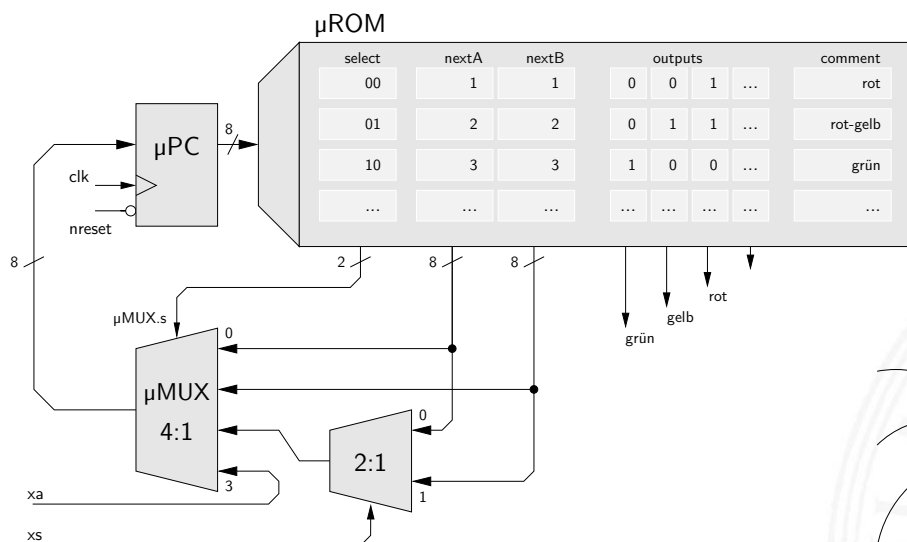
Mikroprogramm: Beispiel Ampel



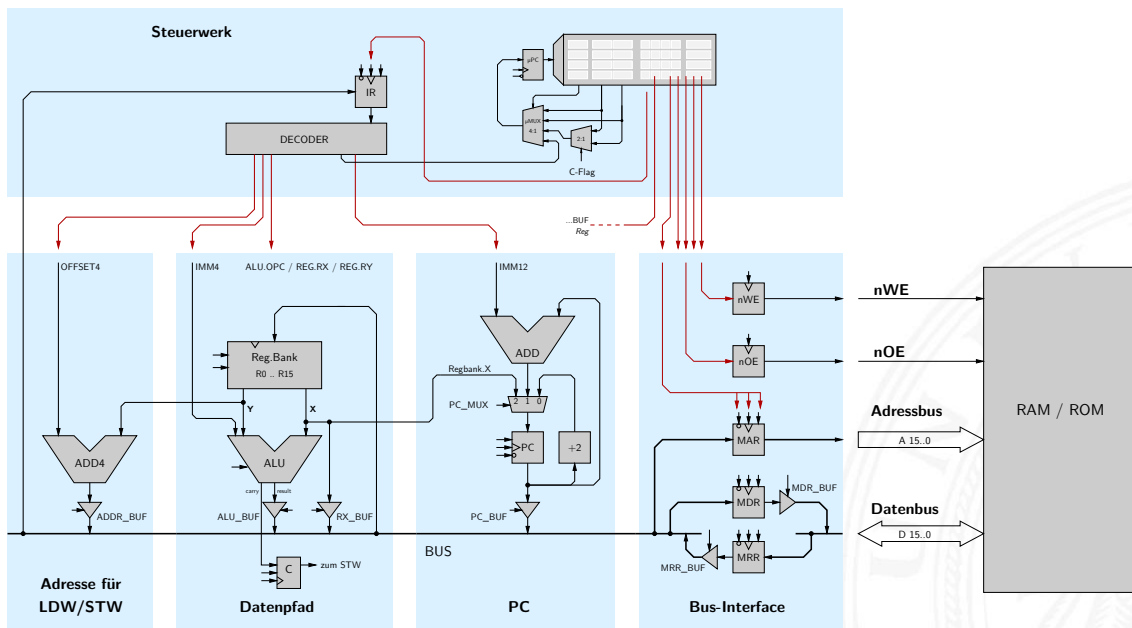
- ▶ μPC adressiert das μROM
- ▶ $next$ -Ausgang liefert Folgezustand
- ▶ andere Ausgänge steuern die Schaltung = die Lampen der Ampel



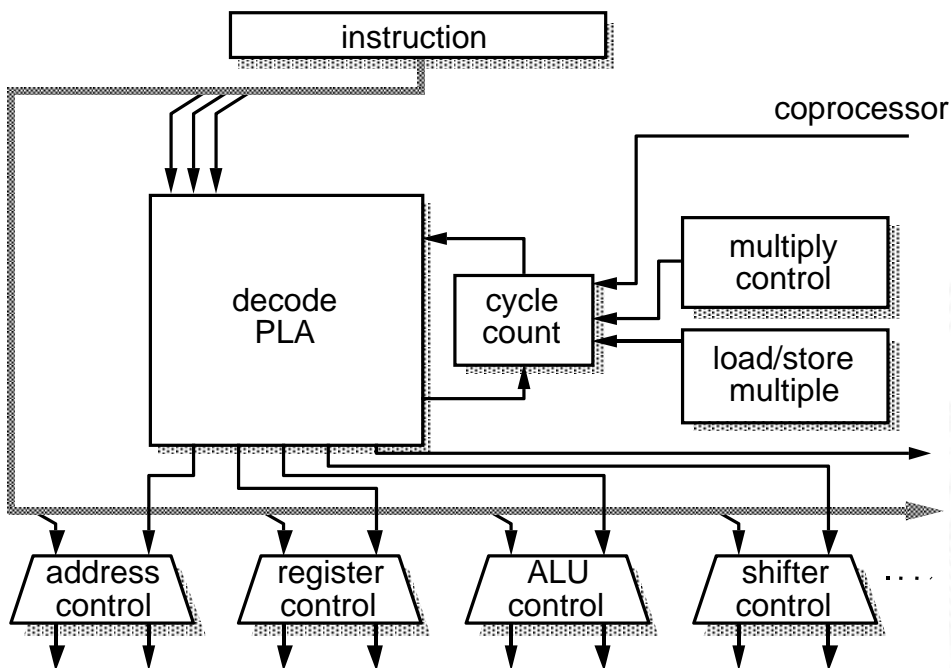
Mikroprogramm: Beispiel zur Auswahl des Folgezustands



- ▶ Multiplexer erlaubt Auswahl des μPC Werts
- ▶ $nextA$, $nextB$ aus dem μROM , externer xa Wert
- ▶ xs Eingang für bedingte Sprünge



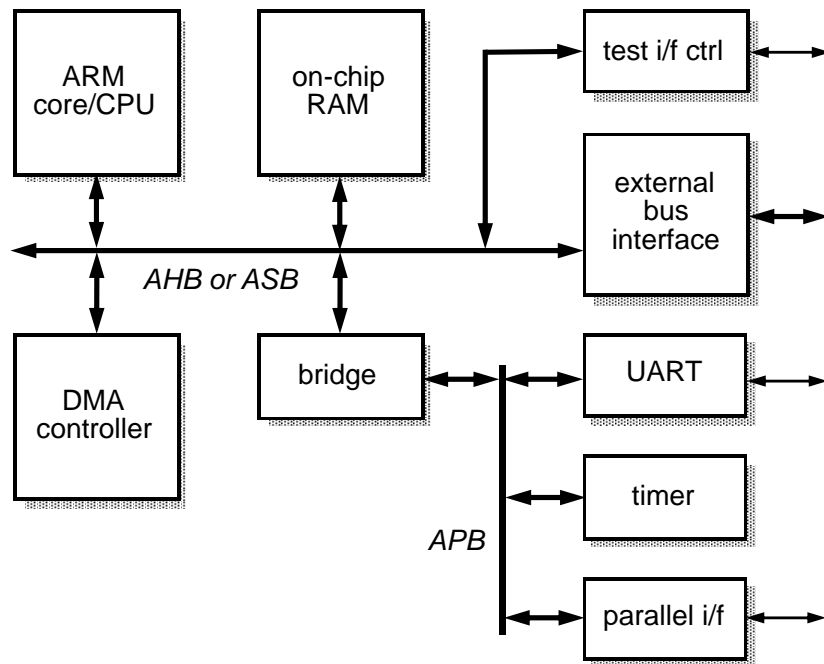
- ▶ aktuelle Architekturen: weniger Mikroprogrammierung
- ⇒ Pipelining (folgt in 14 Rechnerarchitektur II – Pipelining)



[Fur00]



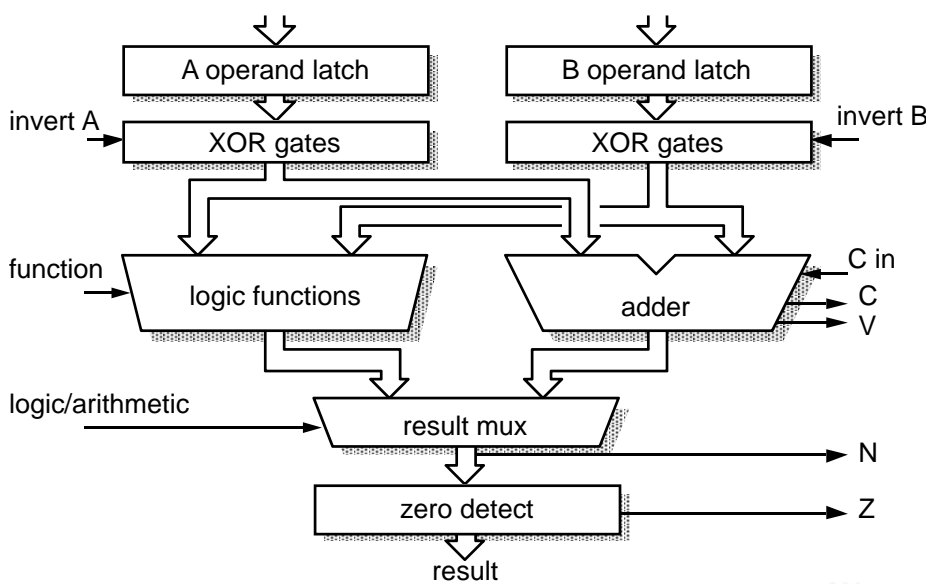
typisches ARM SoC System



S. Furber: ARM System-on-Chip Architecture [Fur00]



RT-Ebene: ALU des ARM 6 Prozessors

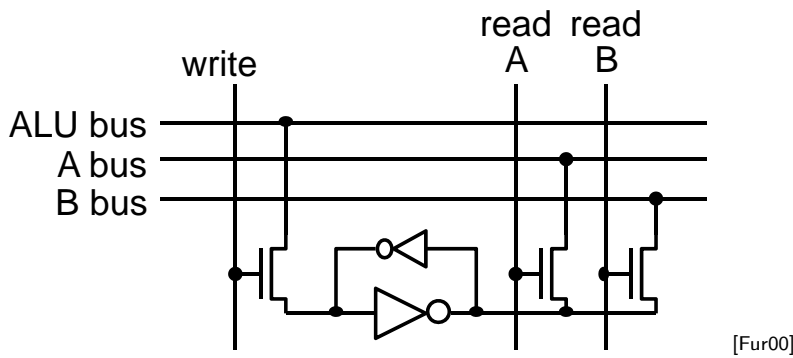


[Fur00]

- ▶ Register für die Operanden A und B
- ▶ Addierer und separater Block für logische Operationen



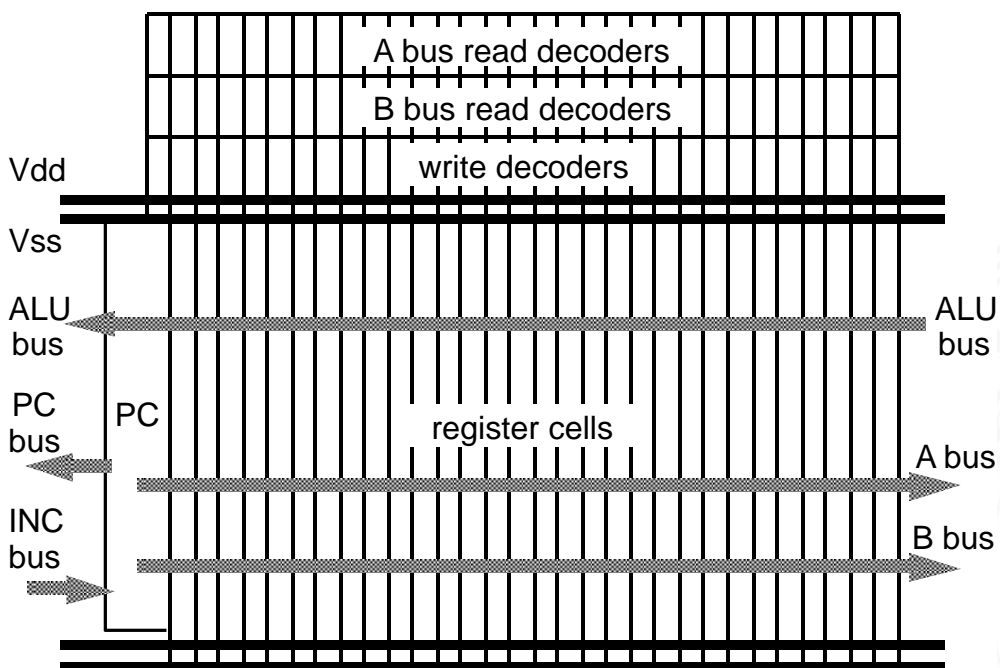
Multi-Port-Registerbank: Zelle



- ▶ Prinzip wie 6T-SRAM: rückgekoppelte Inverter
- ▶ mehrere (hier zwei) parallele Lese-Ports
- ▶ mehrere Schreib-Ports möglich, aber kompliziert



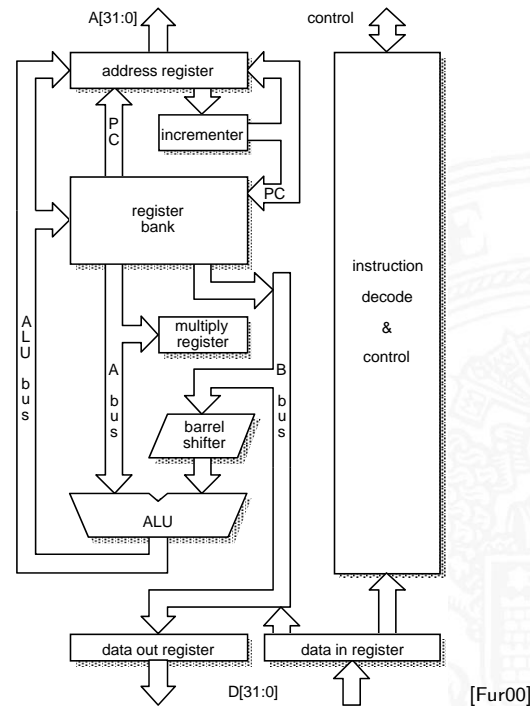
Multi-Port Registerbank: Floorplan/Chiplayout





Kompletter Prozessor: ARM 3

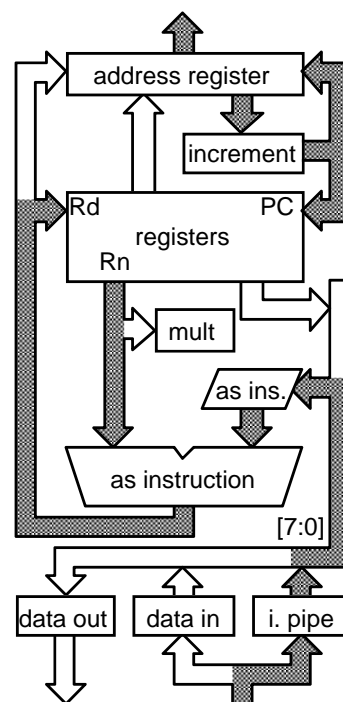
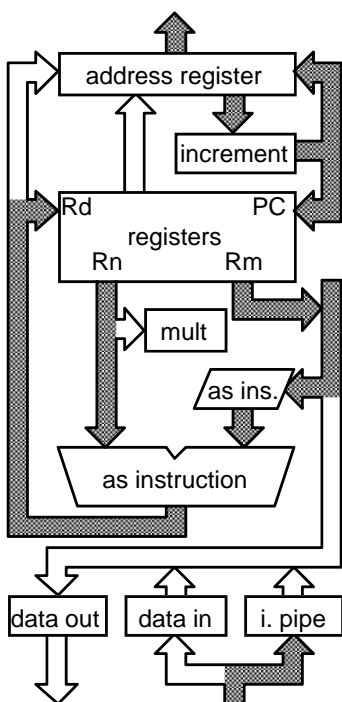
- ▶ Registerbank (inkl. Program Counter)
- ▶ Inkrementer
- ▶ Adress-Register
- ▶ ALU, Multiplizierer, Shifter
- ▶ Speicherinterface (Data-In / -Out)
- ▶ Steuerwerk
- ▶ bis ARM 7: 3-stufige Pipeline *fetch, decode, execute*



[Fur00]

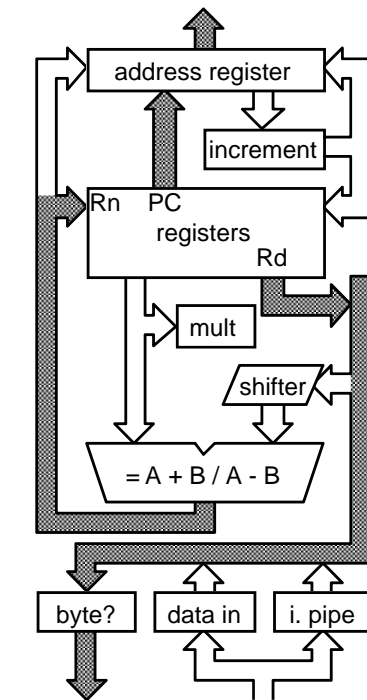
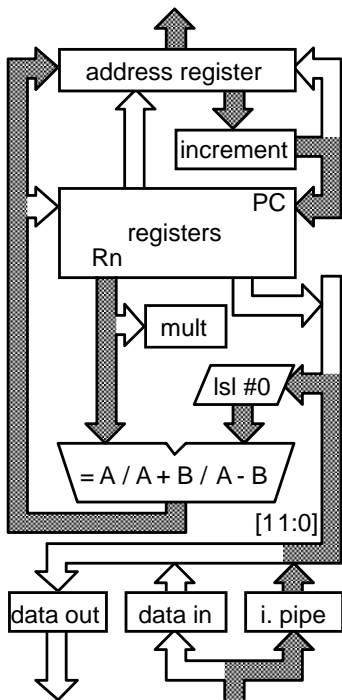


ARM Datentransfer: Register-Operationen

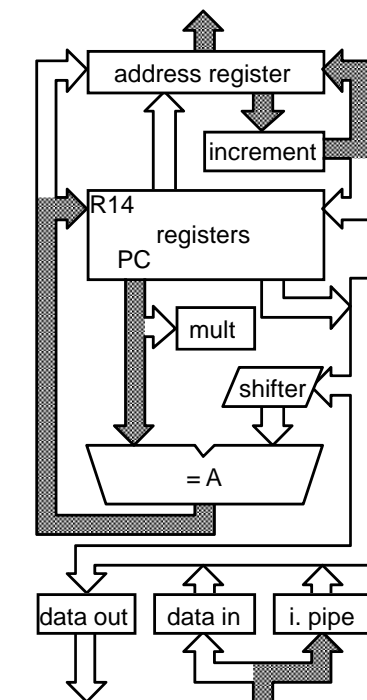
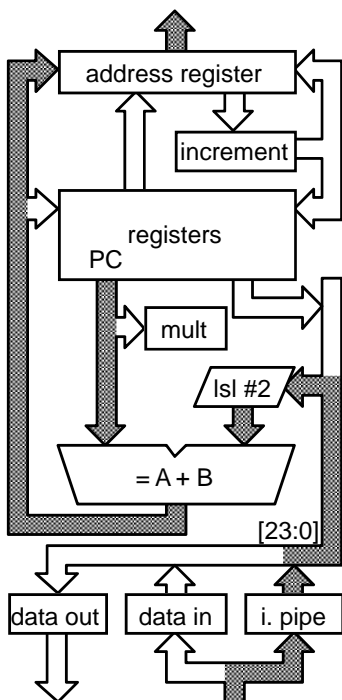




ARM Datentransfer: Store-Befehl



ARM Datentransfer: Funktionsaufruf/Sprungbefehl





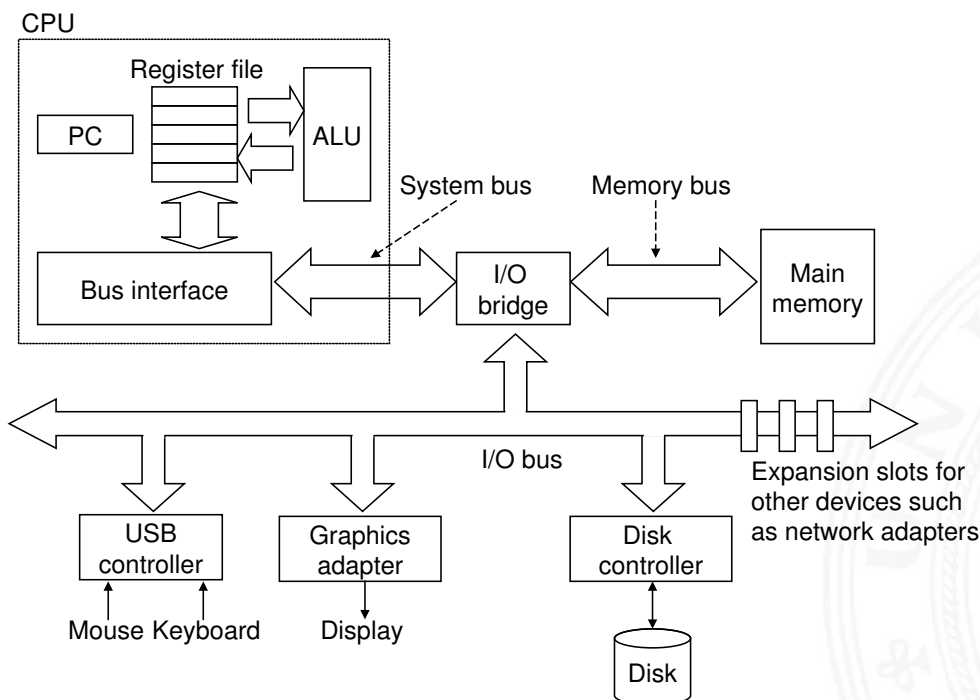
Wie rechnet ein Rechner?

- ▶ „Choreografie“ der Funktionseinheiten
 - ▶ Wie wird ein Programm gestartet?
 - ▶ Was passiert beim Einschalten des Rechners?

 - ▶ Erweiterungen des von-Neumann Konzepts
 - ▶ parallele, statt sequenzieller Befehlsabarbeitung
⇒ *Pipelining*
 - ▶ mehrere Ausführungseinheiten
⇒ *superskalare Prozessoren, Mehrkern-Architekturen*
 - ▶ dynamisch veränderte Abarbeitungsreihenfolge
⇒ „*out-of-order execution*“
 - ▶ getrennte Daten- und Instruktionsspeicher
⇒ *Harvard-Architektur*
 - ▶ *Speicherhierarchie, Caches* etc.
- siehe Kapitel 14 *Rechnerarchitektur II*

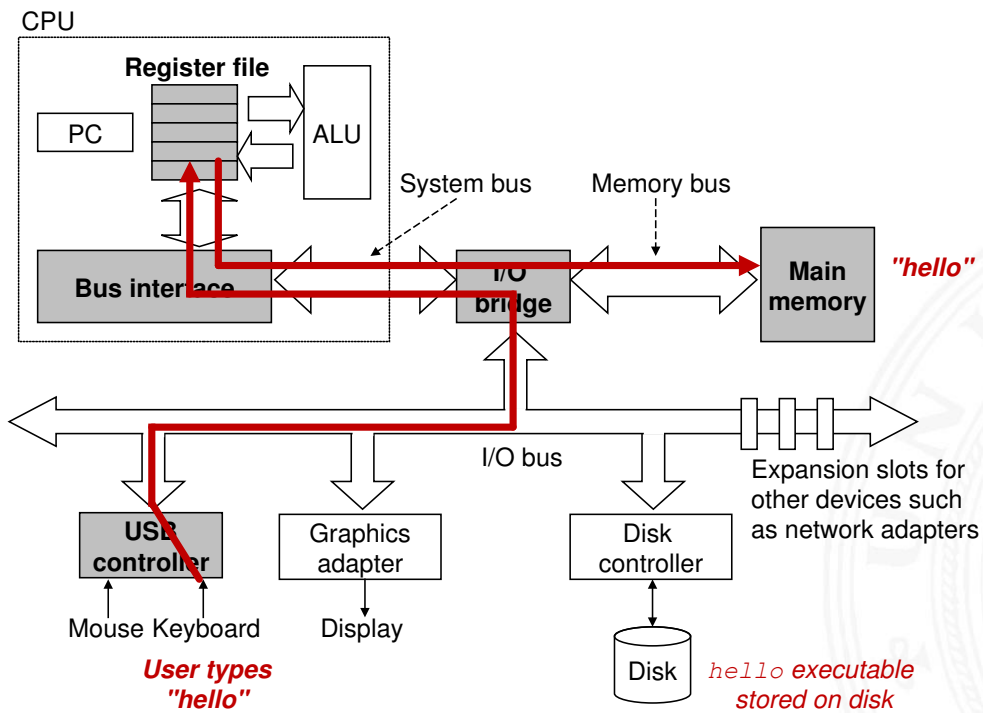


Hardwareorganisation eines typischen Systems

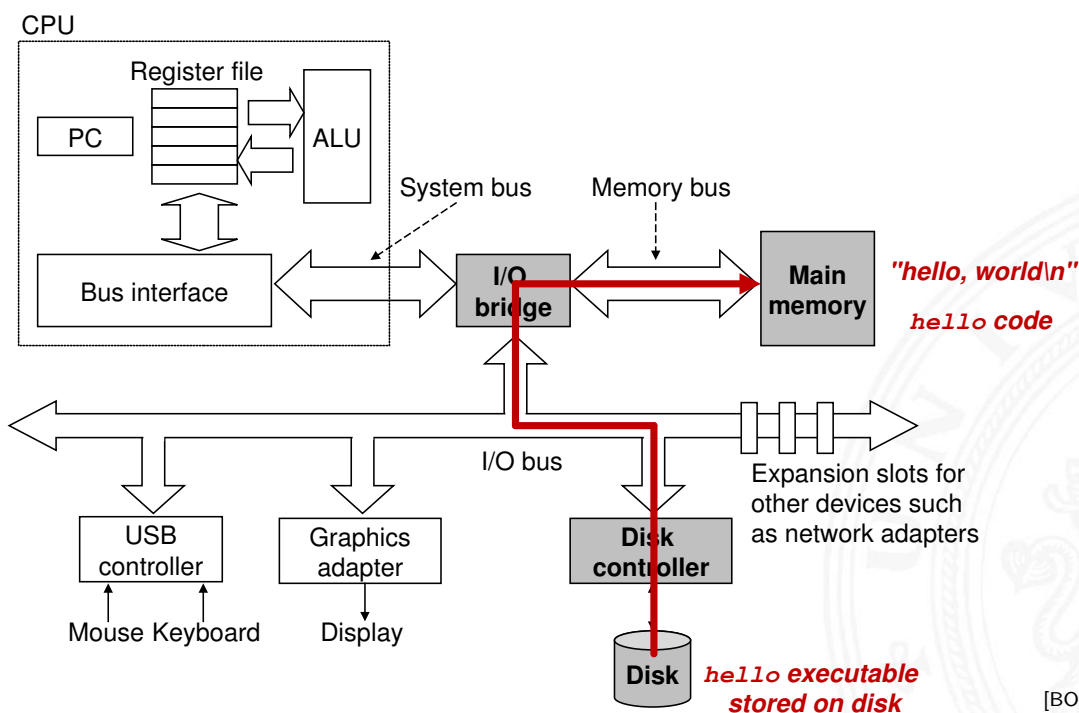




Programmausführung: 1. Benutzereingabe



Programmausführung: 2. Programm laden

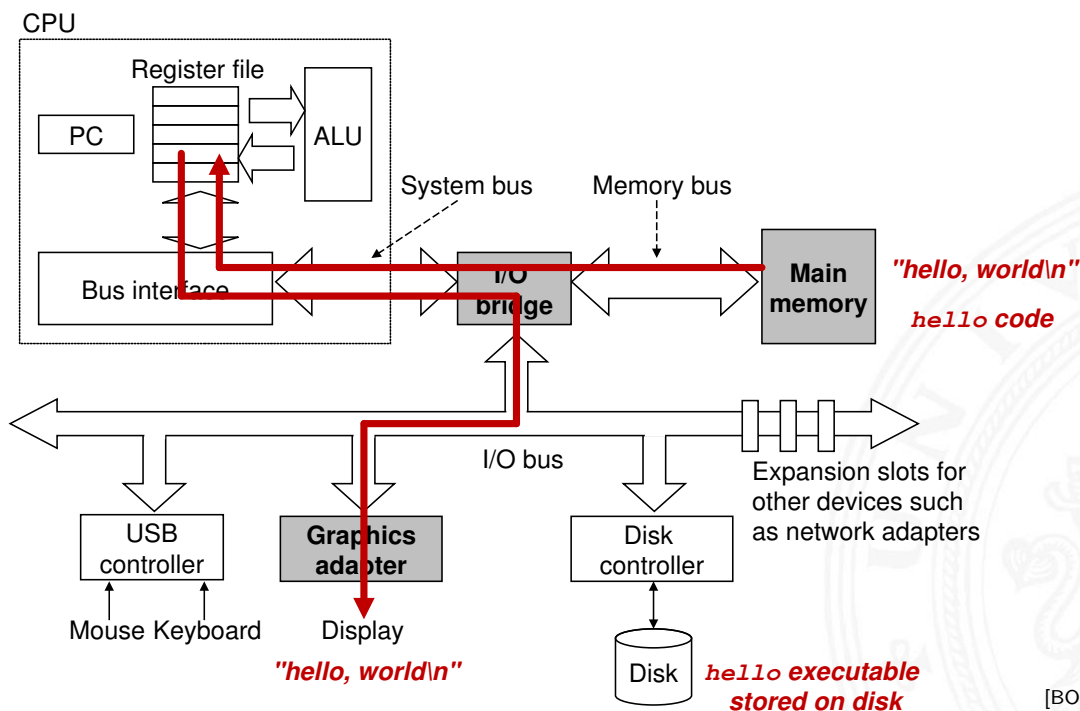




Programmausführung: 3. Programmlauf

11.5 Rechnerarchitektur I - Wie rechnet ein Rechner?

64-040 Rechnerstrukturen und Betriebssysteme



[BO15]

A. Mäder

791



Boot-Prozess

Was passiert beim Einschalten des Rechners?

11.5 Rechnerarchitektur I - Wie rechnet ein Rechner?

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Chipsatz erzeugt Reset-Signale für alle ICs
- ▶ Reset für die zentralen Prozessor-Register (PC ...)
- ▶ PC wird auf Startwert initialisiert
- ▶ Befehlszyklus wird gestartet
- ▶ Initialisierung und Selbsttest des Prozessors
- ▶ Interrupt (APIC) / Protokoll getrieben: Auswahl des Bootstrap Kerns
- ▶ Prozessor greift auf Startadresse zu, wo Boot-Programm beginnt (ROM, Flash)
- ▶ Löschen und Initialisieren der Caches
- ▶ Konfiguration des Chipsatzes
- ▶ Erkennung und Initialisierung von I/O-Komponenten
- ▶ Laden des Betriebssystems, dabei Start aller CPU-Kerne

A. Mäder

792



- [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



- [PH22] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle – MIPS Edition.*
6. Auflage, De Gruyter Oldenbourg, 2022. ISBN 978-3-11-075598-5
- [PH20] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – RISC-V Edition.*
2nd edition, Morgan Kaufmann Publishers Inc., 2020. ISBN 978-0-12-820331-6
- [Mäd11] A. Mäder: *Vorlesung: Rechnerarchitektur und Mikrosystemtechnik.*
Universität Hamburg, FB Informatik, 2011, Vorlesungsfolien.
tams.informatik.uni-hamburg.de/lectures/2011ws/vorlesung/ram
- [HenHA] N. Hendrich: *HADES — HAMBURG DESign System.*
Universität Hamburg, FB Informatik, Lehrmaterial.
tams.informatik.uni-hamburg.de/applets/hades/webdemos



Gliederung

12 Instruction Set Architecture

64-040 Rechnerstrukturen und Betriebssysteme

1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. Logische Operationen
7. Codierung
8. Schaltfunktionen
9. Schaltnetze
10. Schaltwerke
11. Rechnerarchitektur I
12. Instruction Set Architecture



Gliederung (cont.)

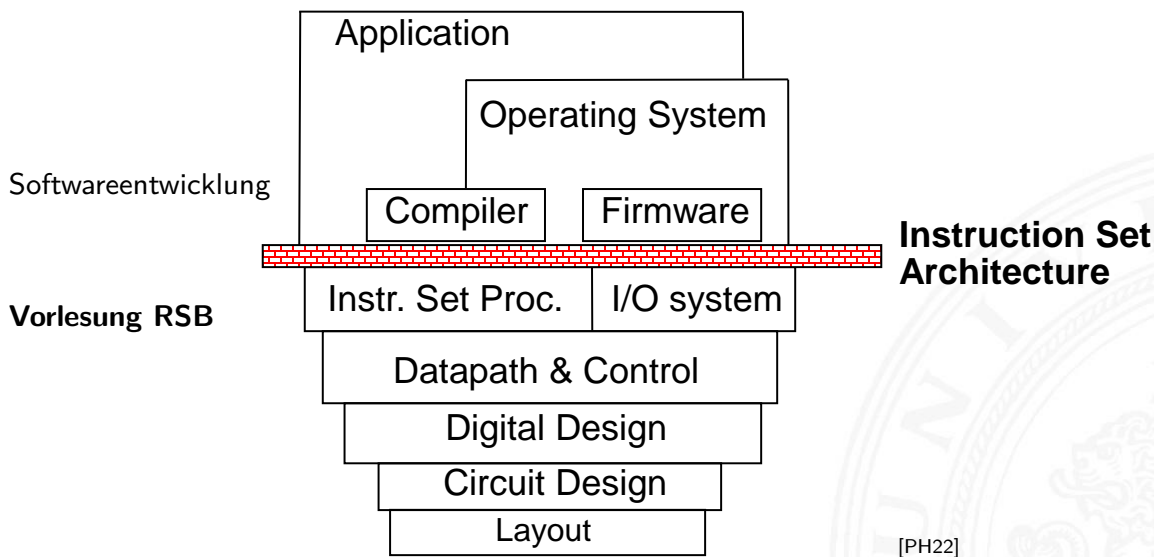
12 Instruction Set Architecture

64-040 Rechnerstrukturen und Betriebssysteme

Speicherorganisation
Befehlssatz
Befehlsformate
Adressierungsarten
Intel x86-Architektur
Befehlssätze
Literatur

13. Assembler-Programmierung
14. Rechnerarchitektur II
15. Betriebssysteme

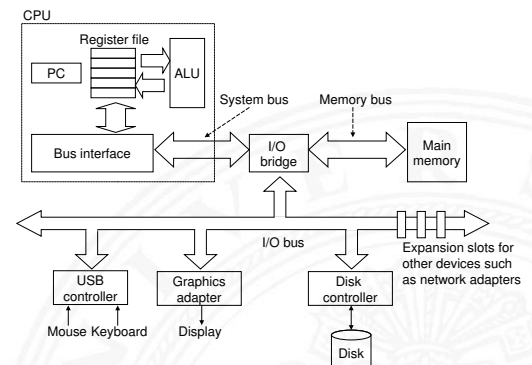




ISA – Instruction Set Architecture

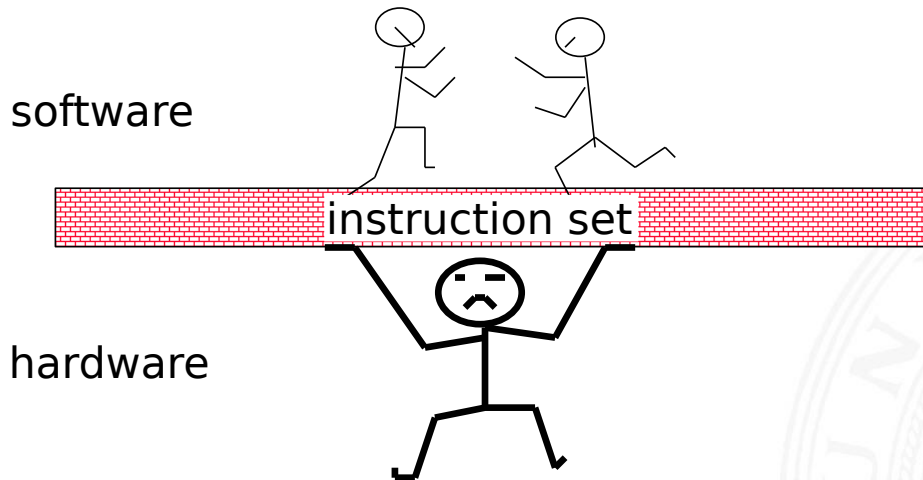
⇒ alle für den Programmierer sichtbaren Attribute einer Rechnerarchitektur

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





- ▶ Befehlssatz: die zentrale Schnittstelle



- ▶ 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



- ▶ 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)



- ▶ Abspeichern von Zahlen, Zeichen, Strings?
 - ▶ kleinster Datentyp üblicherweise ein Byte (8-bit)
 - ▶ andere Daten als Vielfache: 16-bit, 32-bit, 64-bit ...

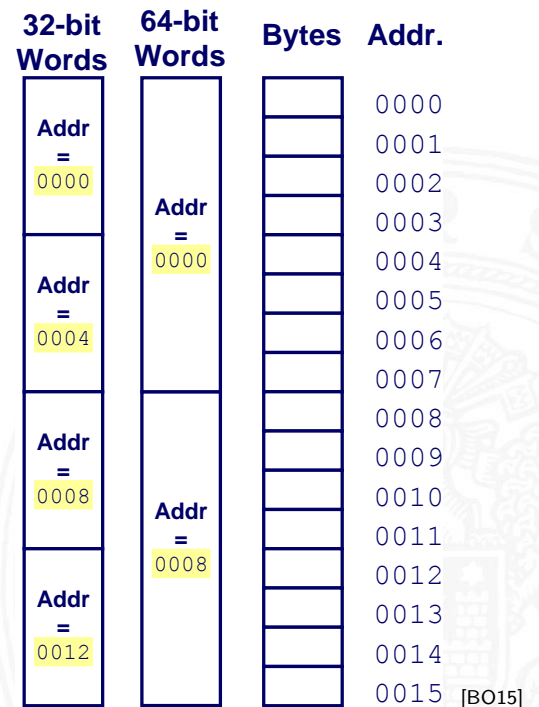
- ▶ Organisation und Adressierung des Speichers?
 - ▶ Adressen typisch in Bytes angegeben
 - ▶ erlaubt Adressierung einzelner ASCII-Zeichen usw.

- ▶ aber Maschine/Prozessor arbeitet wortweise
- ▶ Speicher daher ebenfalls wortweise aufgebaut
- ▶ typischerweise 32-bit oder 64-bit



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



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

segment word size	16 bit			32 bit				64 bit					
	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu, Clang	Intel Linux	Microsoft	Intel Windows	Gnu, Clang	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_t				8	8			8	8	8	8	8	8
enum (typical)	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	8
m128				16	16			16	16	16	16	16	16
m256				32	32			32	32	32	32	32	32
m512				64	64			64	64	64	64	64	64
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

www.agner.org/optimize/calling_conventions.pdf

Byte-Order

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

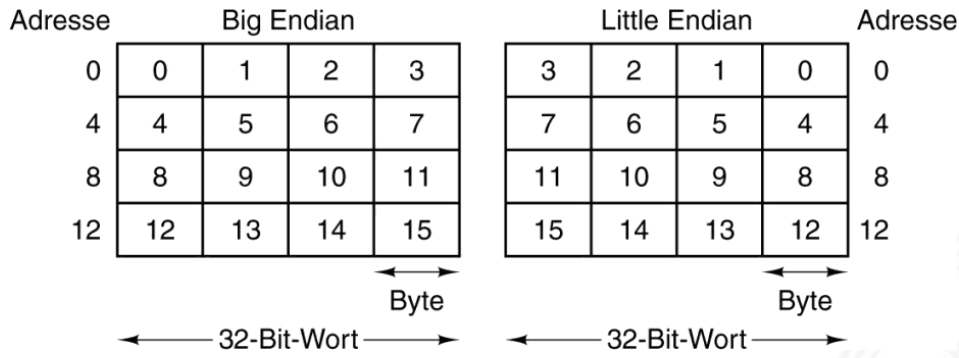
Zwei Möglichkeiten / Konventionen:

- ▶ **Big Endian:** Sun, Mac usw.
das MSB (*most significant byte*) hat die kleinste Adresse
das LSB (*least significant byte*) hat die höchste –"
- ▶ **Little Endian:** Alpha, x86
das MSB hat die höchste, das LSB die kleinste Adresse

satirische Referenz auf Gulliver's Reisen (Jonathan Swift)



Big- vs. Little Endian



[TA14]

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

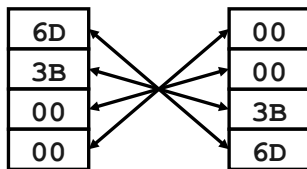


Byte-Order: Beispiel

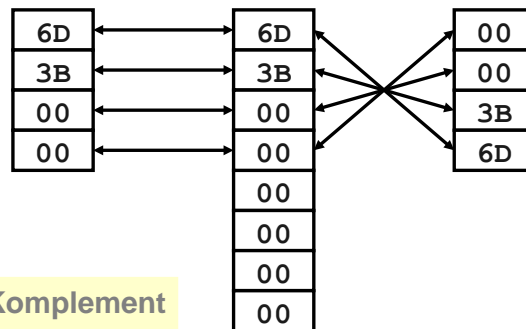
```
int A = 15213;
int B = -15213;
long int C = 15213;
```

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

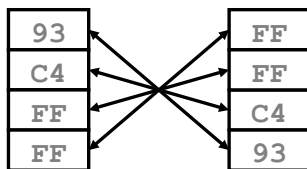
Linux/Alpha A Sun A



Linux c Alpha c Sun c



Linux/Alpha B Sun B



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...
```



- ▶ Byte-Order muss bei Datenübertragung zwischen Rechnern berücksichtigt und eingehalten werden

- ▶ Internet-Protokoll (IP) nutzt ein Big Endian Format
- ⇒ 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
 - ▶ ...



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/inet/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 [BO15], 2.1.4 (Abb. 2.3, 2.4)

```

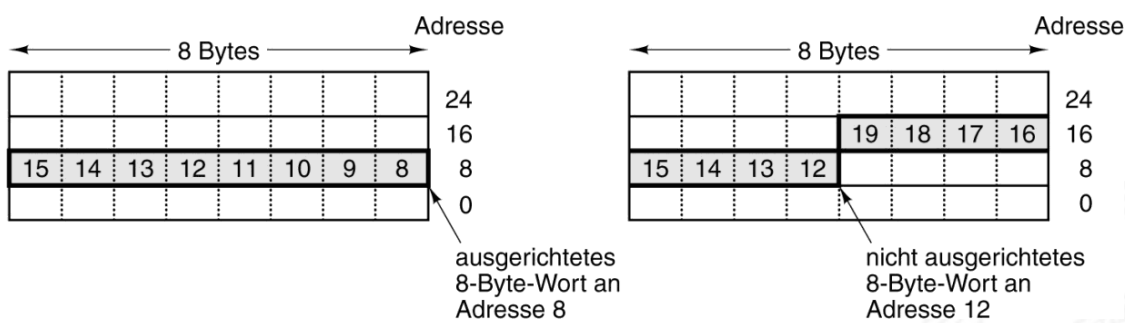
void show_bytes( byte_pointer start, int len ) {
    int i;
    for( i=0; i < len; i++ ) {
        printf( " %.2x", start[i] );
    }
    printf( "\n" );
}

void show_double( double x ) {
    show_bytes( (byte_pointer) &x, sizeof( double ) );
}
...

```



„Misaligned“ Zugriff



- ▶ 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 Speicherwort
- ⇒ was passiert bei „krummen“ (*misaligned*) Adressen?
 - ▶ automatische Umsetzung auf mehrere Zugriffe (x86)
 - ▶ Programmabbruch (SPARC)

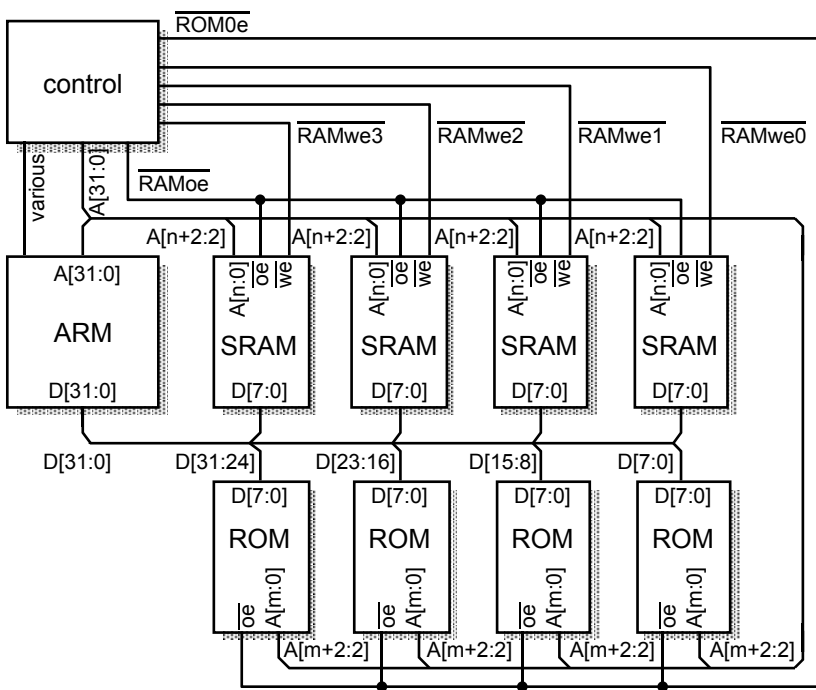


Memory Map

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

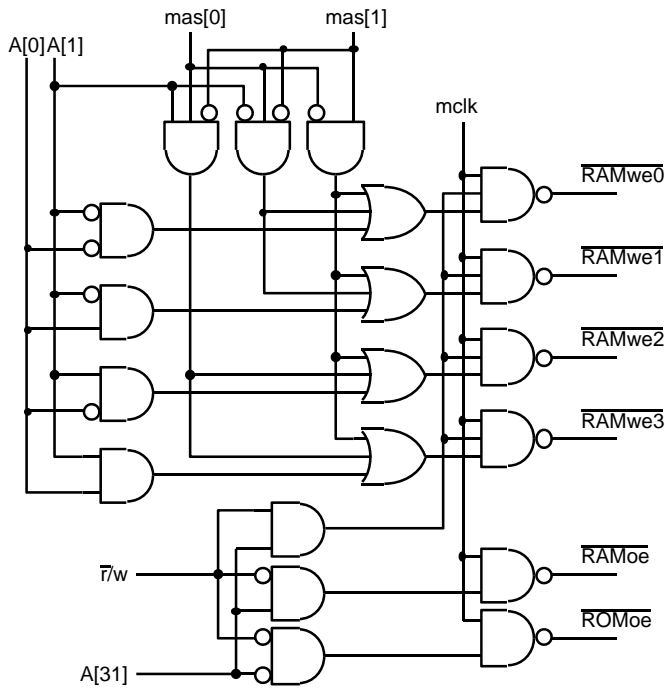


32-bit ARM Proz.
 4 × 8-bit SRAMs
 4 × 8-bit ROMs

[Fur00]



Adressabbildung Hardware: ARM (cont.)



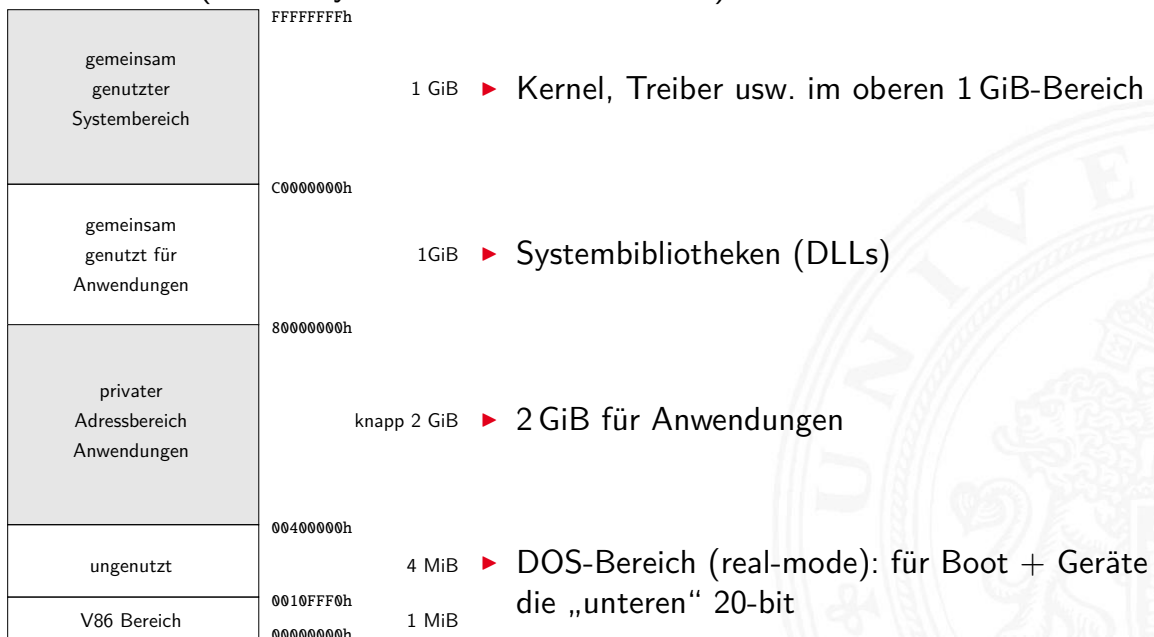
Adressdecoder Hardware

[Fur00]



Adressabbildung in Betriebssystemen

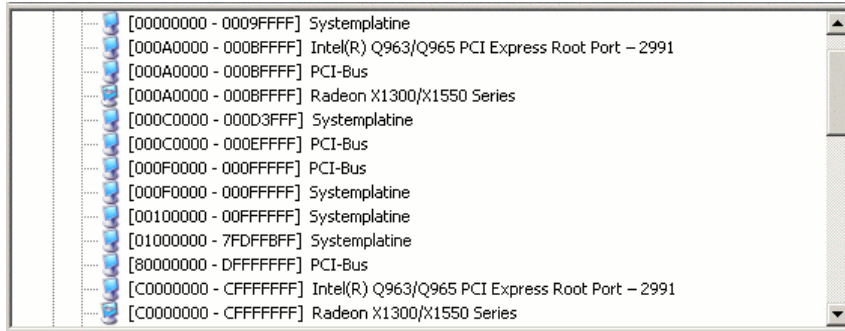
► Windows 95 (32-bit System: 4 GiB Adressraum)





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

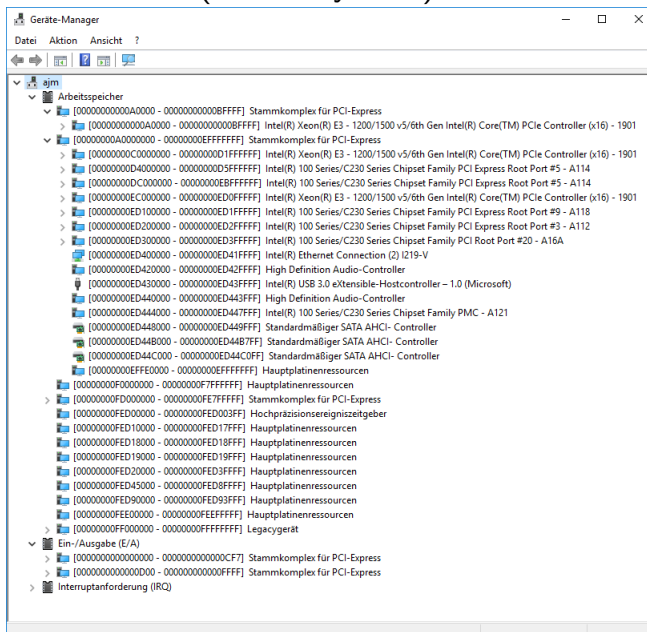


- ▶ x86 I/O-Adressraum gesamt nur 64 KiByte
- ▶ je nach Zahl der I/O-Geräte evtl. fast voll ausgenutzt
- ▶ Adressen vom BIOS zugeteilt



Adressabbildung in Betriebssystemen (cont.)

- ▶ Windows 10 (64-bit System)

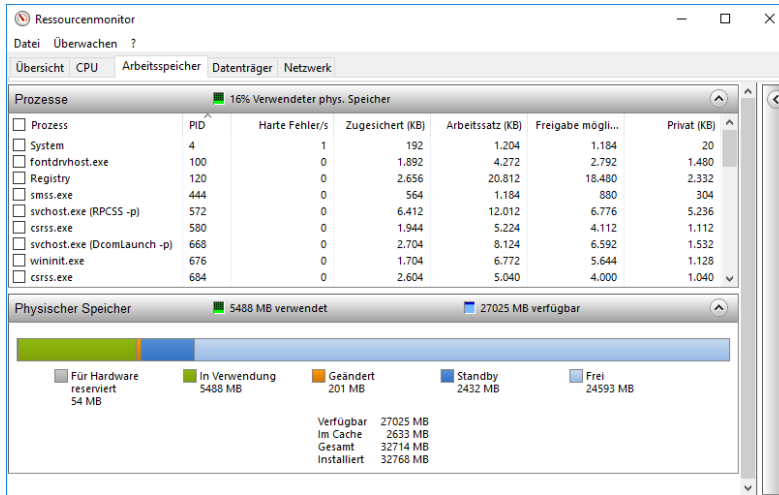


⇒ Adressbereiche

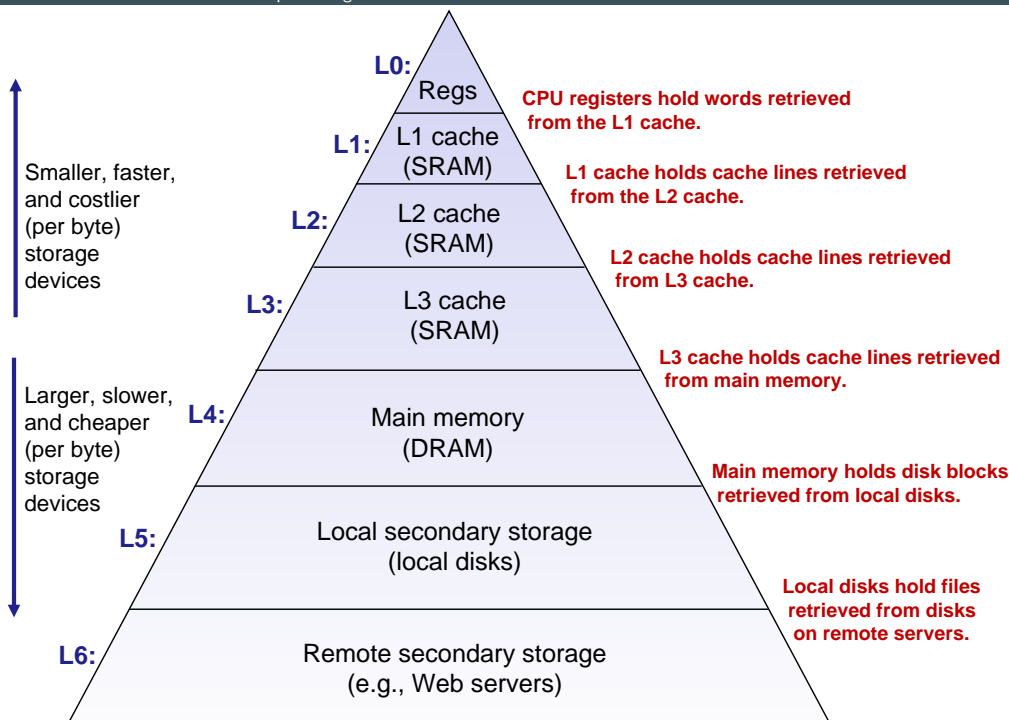


mehrstufige Abbildung:

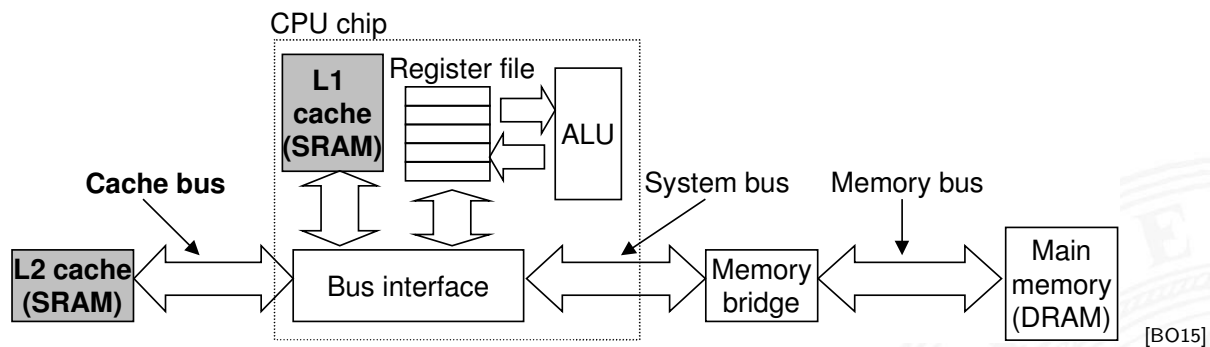
1. alle Hardwarekomponenten, Ein-/Ausgabeeinheiten und Interrupts ⇒ Adressbereiche
2. Adressbereiche ⇒ physikalischer Speicher



► Linux siehe: free, vmstat, dmidecode, hwinfo, top, lshw, ...



[BO15] später mehr ...



- ▶ schneller, mehrstufiger Zwischenspeicher (Level 1, 2, 3 ...)
überbrückt Geschwindigkeitsunterschied 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 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 konstant
von Größe des Speichers, Datenstrukturen und -zugriffen abhängig
- ▶ Wechselwirkungen: Speichersystem \Leftrightarrow Programme
 - ▶ Cacheorganisation und Virtual-Memory beeinflussen Performanz/Programmleistung
 - ▶ Anpassung des Programms an das Speichersystem kann Laufzeitverhalten verbessern

→ siehe 14 Rechnerarchitektur II – Speicherhierarchie



ISA-Merkmale des Prozessors

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



Befehlszyklus

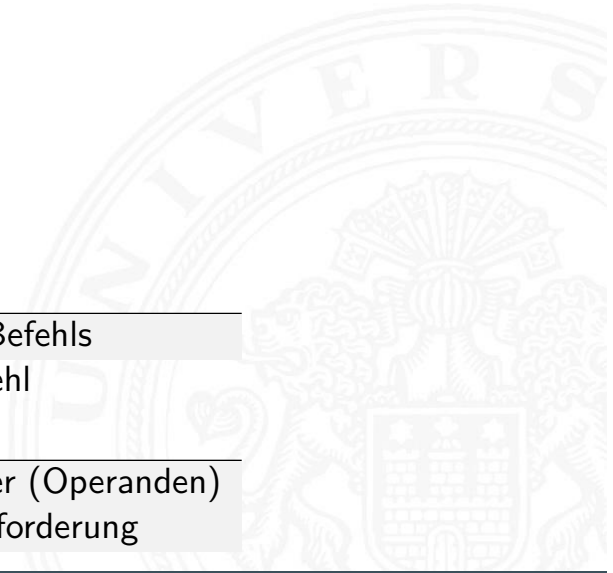
- ▶ Prämisse: von-Neumann Prinzip
 - ▶ Daten und Befehle im gemeinsamen Hauptspeicher
- ▶ Abarbeitung des Befehlszyklus in Endlosschleife
 - ▶ Programmzähler PC adressiert den Speicher
 - ▶ gelesener Wert kommt in das Befehlsregister IR
 - ▶ Befehl decodieren
 - ▶ Befehl ausführen
 - ▶ nächsten Befehl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

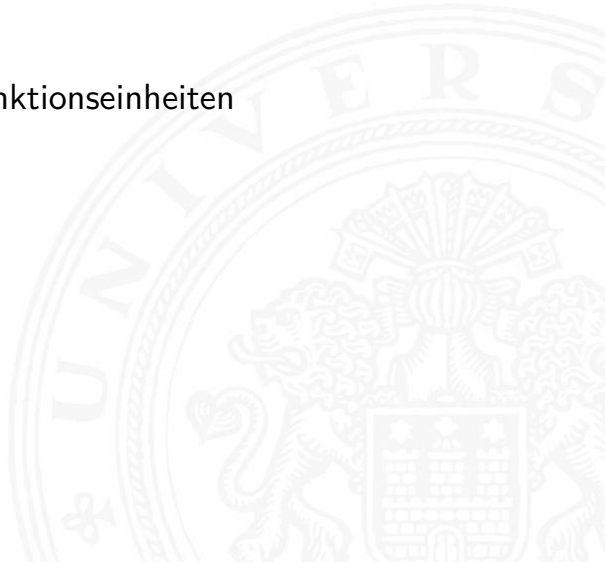


Instruction Execute

„Befehl ausführen“ Phase im Befehlszyklus

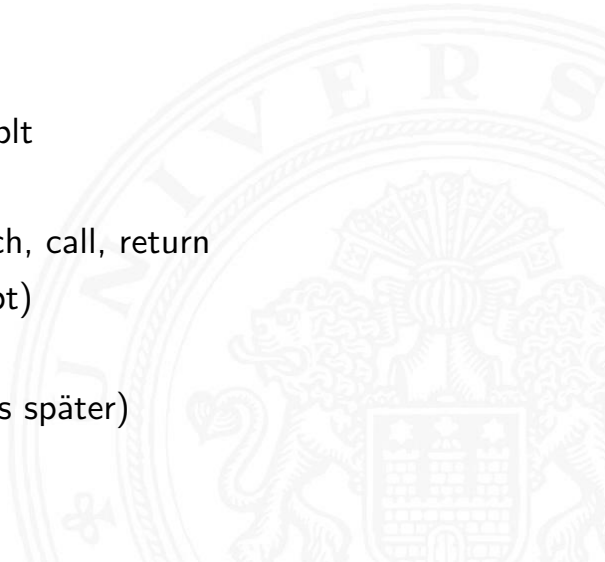
- ▷ Befehl steht im Befehlsregister IR
- ▷ Decoder hat Opcode und Operanden entschlüsselt
- ▷ Steuersignale liegen an Funktionseinheiten
- 1. Ausführung des Befehls durch Aktivierung der Funktionseinheiten
- 2. ggf. Programmzähler setzen/inkrementieren

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



Welche Befehle braucht man?

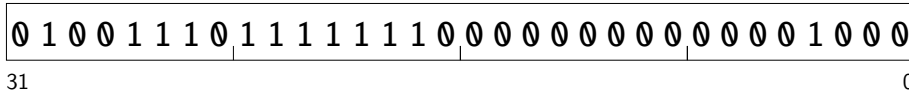
Befehlsklassen	Beispiele
▶ arithmetische Operationen	add, sub, inc, dec, mult, div
logische Operationen	and, or, xor
schiebe Operationen	shl, sra, srl, ror
▶ Vergleichsoperationen	cmpeq, cmpgt, cmplt
▶ Datentransfers	load, store, I/O
▶ Programm-Kontrollfluss	jump, jmq, branch, call, return
▶ Maschinensteuerung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



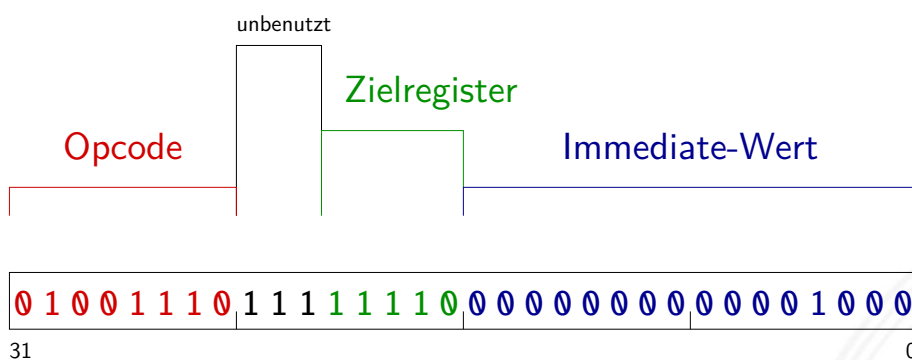
Wie soll die Hardware diesen Wert interpretieren?

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

- ⇒ Aufteilung in Felder: Opcode und Operanden
- ⇒ Decodierung über mehrere, kleine Tabellen
- ⇒ unterschiedliche Aufteilung für unterschiedliche Befehle: **Befehlsformate**



Befehlsformate



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



Befehlsformat: drei Beispielarchitekturen

12.3 Instruction Set Architecture - Befehlsformate

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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

12.3 Instruction Set Architecture - Befehlsformate

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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 (Schreiboperationen werden ignoriert)
 - ▶ R0 Tricks

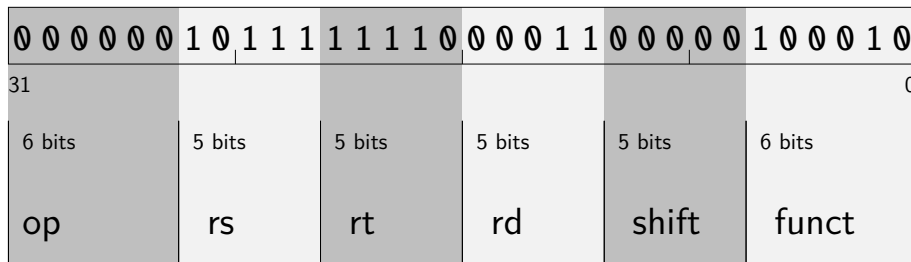
R5 = -R5	<code>sub R5, R0, R5</code>
R4 = 0	<code>add R4, R0, R0</code>
R3 = 17	<code>addi R3, R0, 17</code>
if (R2 != 0)	<code>bne R2, R0, label</code>
- ▶ keine separaten Statusflags
- ▶ Vergleichsoperationen setzen Zielregister auf 0 bzw. 1

R1 = (R2 < R3)	<code>slt R1, R2, R3</code>
----------------	-----------------------------



MIPS: Befehlsformate

Befehl im R-Format

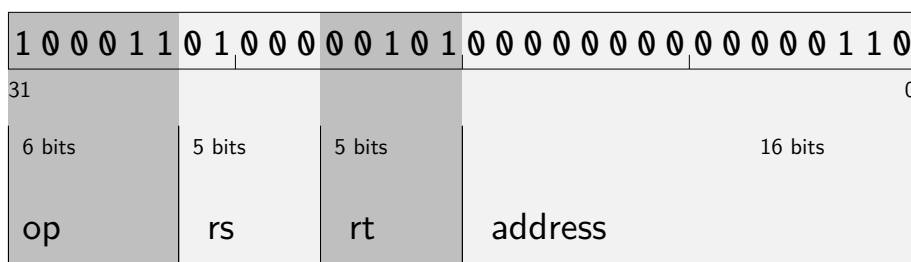


- ▶ op: Opcode Typ des Befehls 0 = „alu-op“
 - rs: source register 1 erster Operand 23 = „r23“
 - rt: source register 2 zweiter Operand 30 = „r30“
 - rd: destination register Zielregister 3 = „r3“
 - shift: shift amount (optionales Shiften) 0 = „0“
 - funct: ALU function Rechenoperation 34 = „sub“
-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

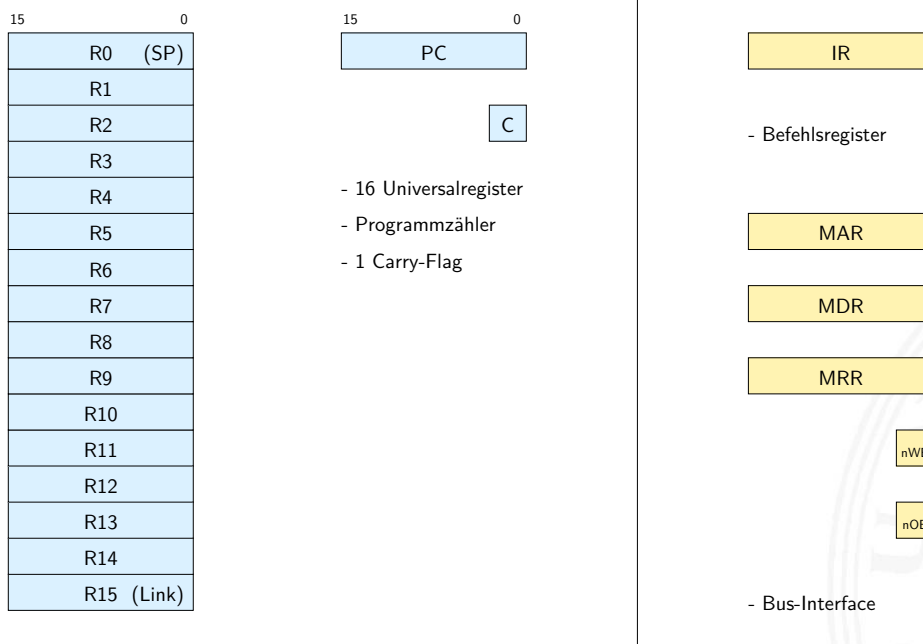


D-CORE

- ▶ 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 aus den Praktikumsunterlagen
 - ▶ 64-042: Rechnerstrukturen und Betriebssysteme (t3asm)



D-CORE: Registermodell



► sichtbar für Programmierer: R0 ... R15, PC und C



D-CORE: Befehlssatz

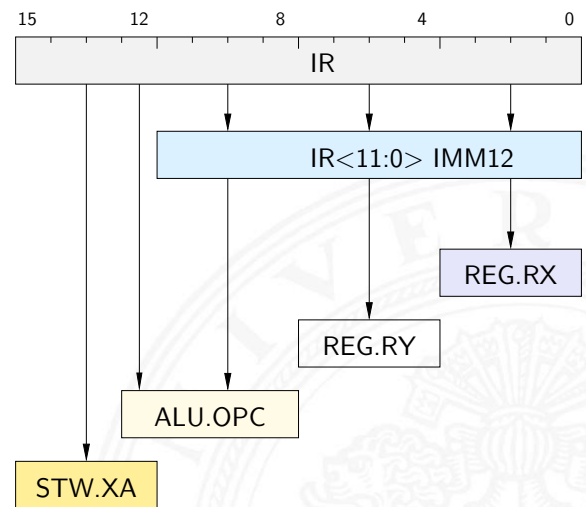
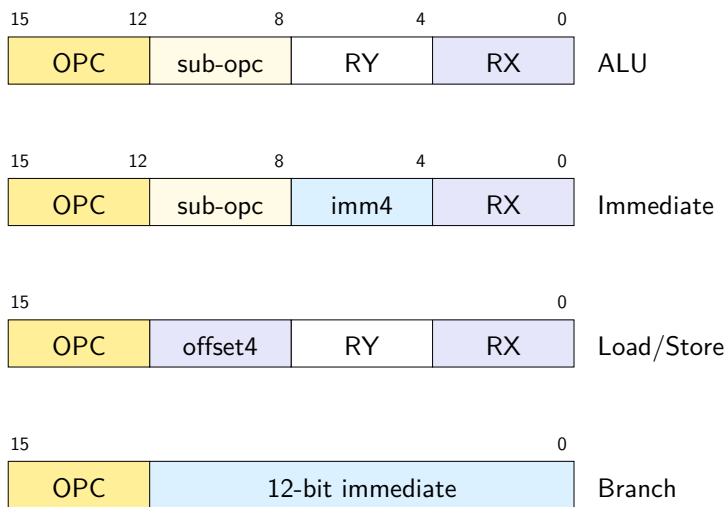
Befehl	Funktion
mov	move register
addu, addc	Addition (ohne, mit Carry)
subu	Subtraktion
and, or, xor	logische Operationen
lsl, lsr, asr	logische, arithmetische Shifts
cmpe, cmpne ...	Vergleichsoperationen
movi, addi ...	Operationen mit Immediate-Operanden
ldw, stw	Speicherzugriffe: load und store
br, jmp	unbedingte Sprünge
bt, bf	bedingte Sprünge
jsr	Unterprogrammaufruf
trap	Software interrupt
rfi	return from interrupt



D-CORE: Befehlsformate

12.3 Instruction Set Architecture - Befehlsformate

64-040 Rechnerstrukturen und Betriebssysteme



- ▶ 4-bit Opcode, 4-bit Registeradressen
- ▶ einfaches Zerlegen des Befehls in die einzelnen Felder



Adressierungsarten

12.4 Instruction Set Architecture - Adressierungsarten

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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.

⇒ wichtige Unterscheidungsmerkmale für Rechnerarchitekturen

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



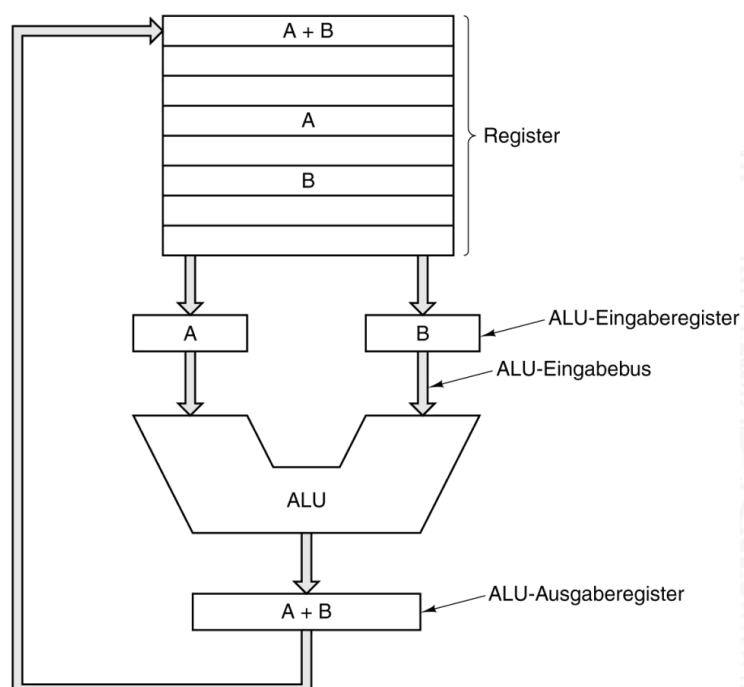
Beispiel: Add-Befehl

- ▶ Rechner soll „rechnen“ können
- ▶ typische arithmetische Operation nutzt 3 Variablen
Resultat, zwei Operanden: $X = Y + Z$
add r2, r4, r5 Inhalt von r4 und r5 addieren, Resultat in r2 speichern
- ▶ woher kommen die Operanden?
- ▶ wo soll das Resultat hin?
 - ▶ Speicher
 - ▶ Register
- ▶ entsprechende Klassifikation der Architektur



Beispiel: Datenpfad

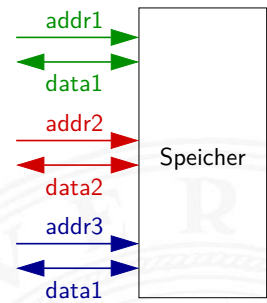
- ▶ Register (-bank)
 - ▶ liefern Operanden
 - ▶ speichern Resultate
- ▶ interne Hilfsregister
- ▶ ALU, typ. Funktionen:
 - ▶ add, add-carry, sub
 - ▶ and, or, xor
 - ▶ shift, rotate
 - ▶ compare
 - ▶ (floating point ops.)





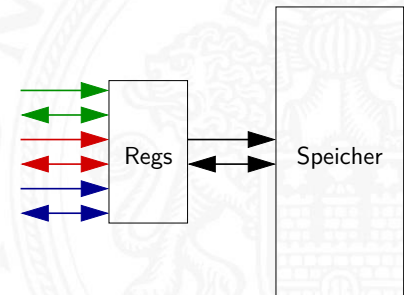
Woher kommen die Operanden?

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



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

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



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

- 3-Adress Format
 - ▶ $X = Y + Z$
 - ▶ sehr flexibel, leicht zu programmieren
 - ▶ Befehl muss 3 Adressen codieren
- 2-Adress Format
 - ▶ $X = X + Z$
 - ▶ eine Adresse doppelt verwendet: Resultat und ein Operand
 - ▶ Format wird häufig verwendet
- 1-Adress Format
 - ▶ $ACC = ACC + Z$
 - ▶ alle Befehle nutzen das Akkumulator-Register
 - ▶ häufig in älteren / 8-bit Rechnern
- 0-Adress Format
 - ▶ $TOS = TOS + NOS$
 - ▶ Stapelspeicher: *top of stack, next of stack*
 - ▶ Adressverwaltung entfällt
 - ▶ im Compilerbau beliebt



Beispiel: n-Adress Maschine

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

Hilfsregister: T

3-Adress Maschine

```

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

```

2-Adress Maschine

```

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

```

1-Adress Maschine

```

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

```

0-Adress Maschine

```

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

```



Beispiel: Stack-Maschine / 0-Adress Maschine

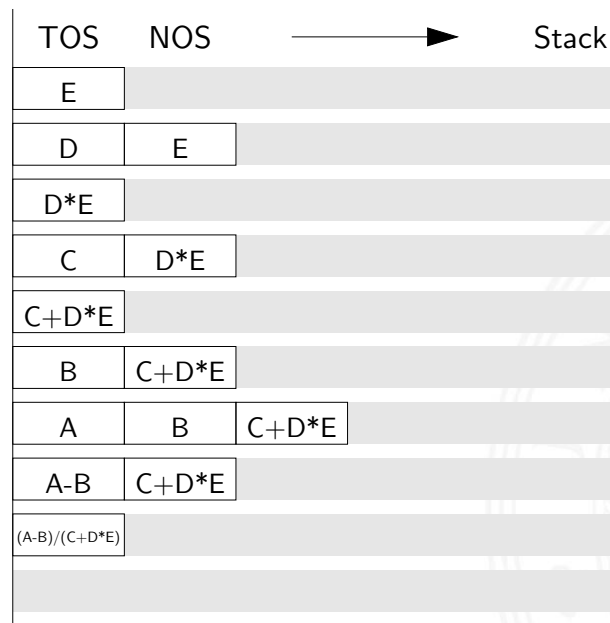
Beispiel: $Z = (A-B) / (C + D * E)$

0-Adress Maschine

```

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

```





Adressierungsarten

12.4 Instruction Set Architecture - Adressierungsarten

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ „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)

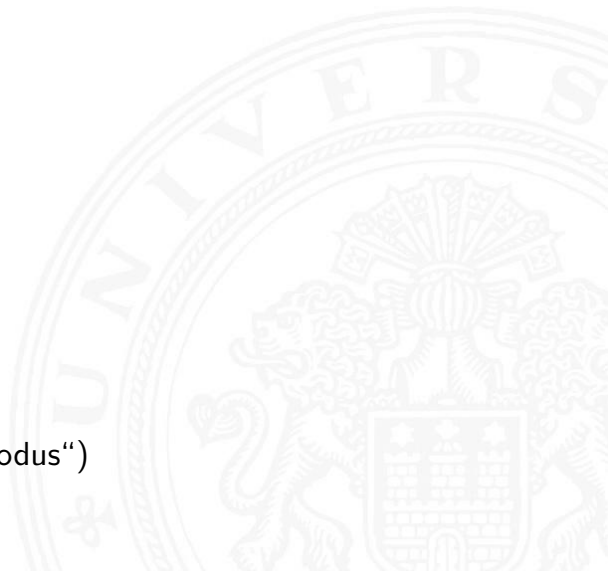


Adressierungsarten (cont.)

12.4 Instruction Set Architecture - Adressierungsarten

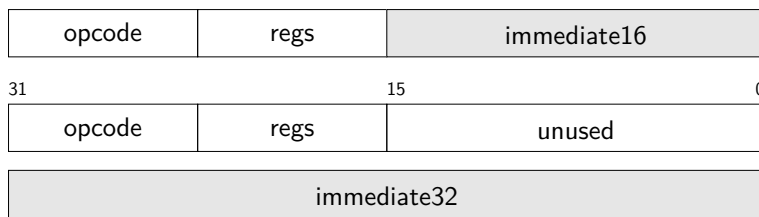
64-040 Rechnerstrukturen und Betriebssysteme

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





Immediate Adressierung



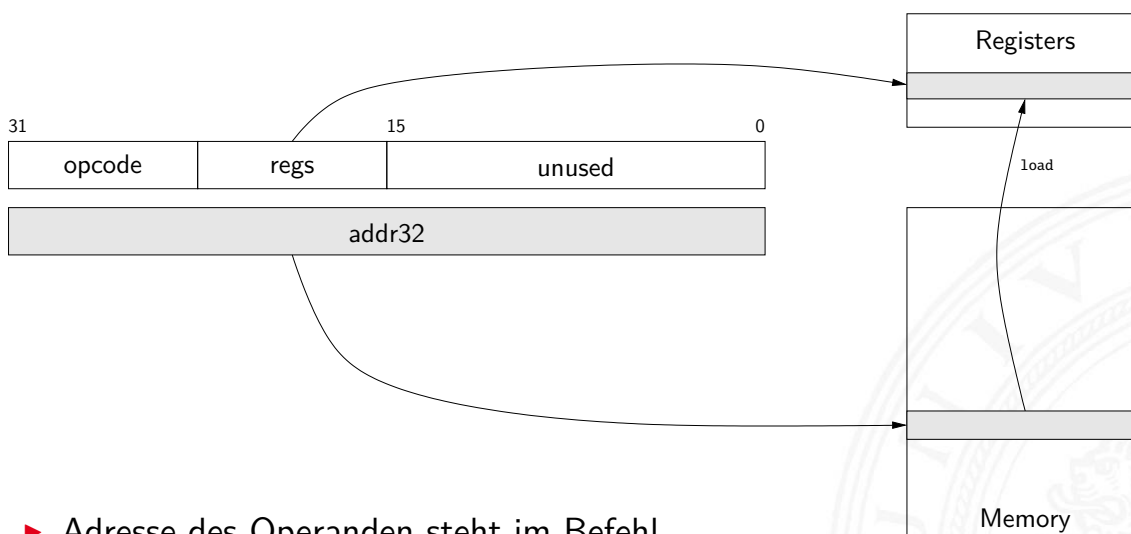
1-Wort Befehl

2-Wort Befehl

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



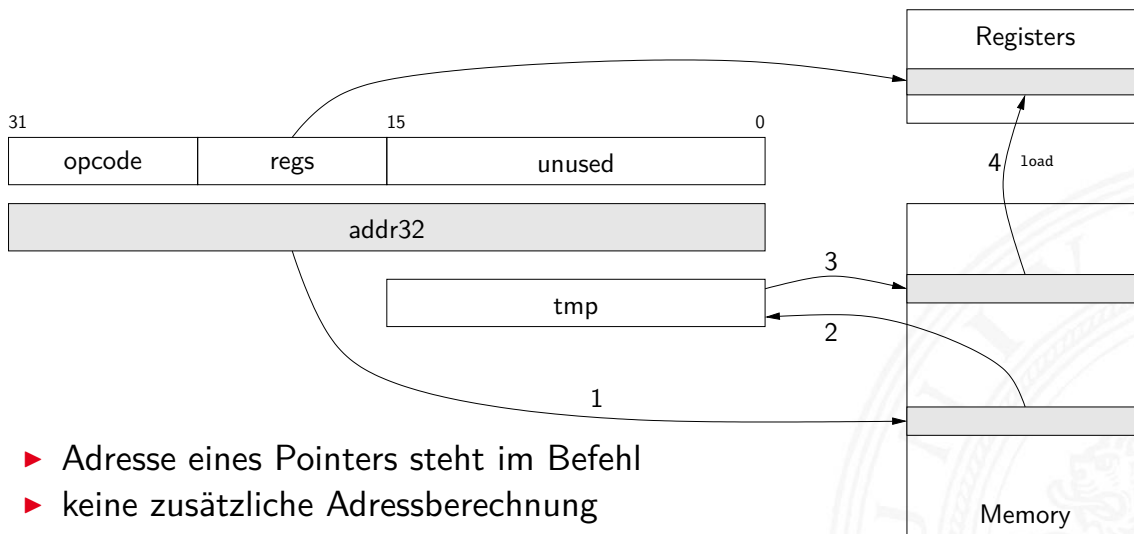
Direkte Adressierung



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



Indirekte Adressierung

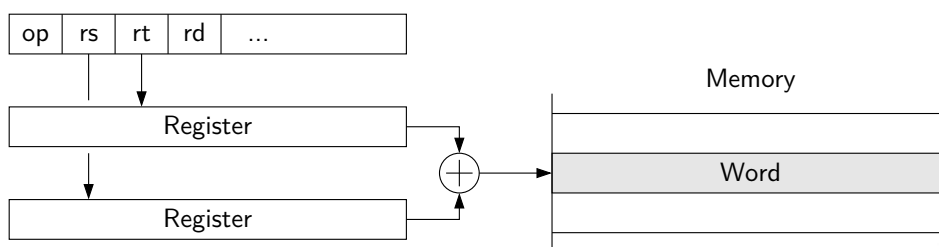


- ▶ Adresse eines Pointers steht im Befehl
- ▶ keine zusätzliche Adressberechnung
- ▶ zwei zusätzliche Speicherzugriffe:
z.B. $tmp = MEM[addr32]$ $R3 = MEM[tmp]$
- ▶ typische CISC-Adressierungsart, viele Taktzyklen
- ▶ kommt bei RISC-Rechnern nicht vor

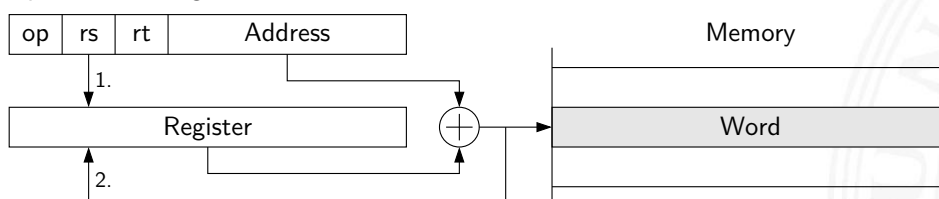


Indizierte Adressierung

Indexaddressing



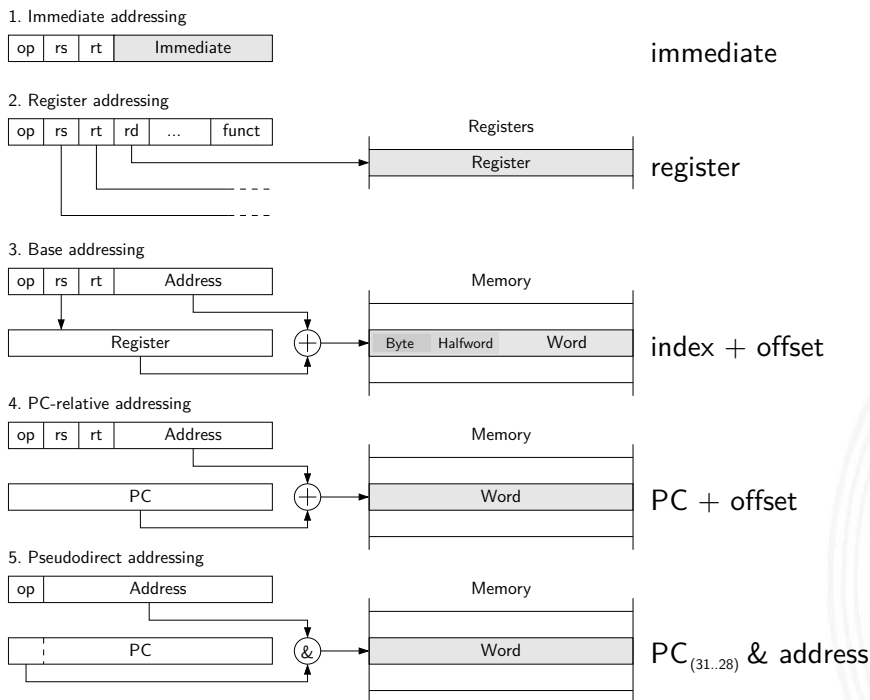
Updateaddressing



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



Beispiel: MIPS Adressierungsarten



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

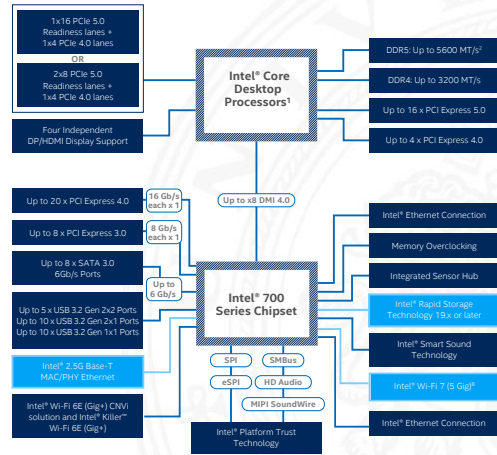
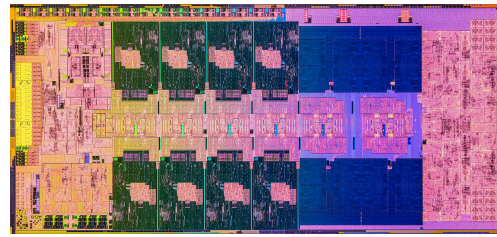
Chip	Datum	MHz	Transistoren	Speicher	Anmerkungen
4004	4/1971	0,108	2 300	640 B	erster Mikroprozessor auf einem Chip
8008	4/1972	0,108	3 500	16 KiB	erster 8-bit Mikroprozessor
8080	4/1974	2	6 000	64 KiB	„general-purpose“ CPU auf einem Chip
8086	6/1978	5–10	29 000	1 MiB	erste 16-bit CPU auf einem Chip
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 ...
...	11/2021				Performance- / Efficiency-Core ...



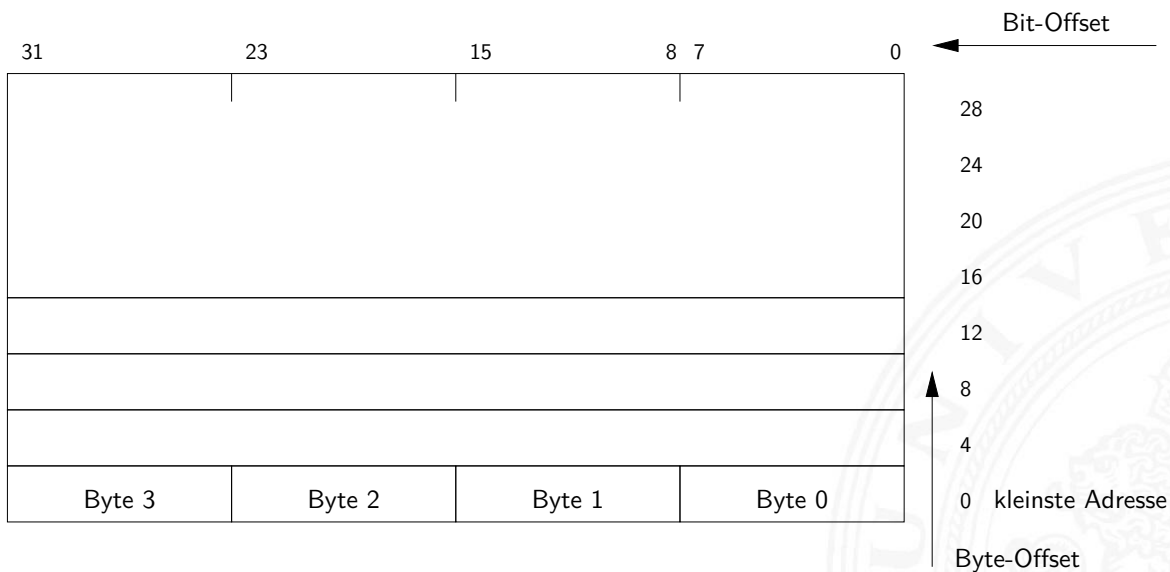
Beispiel: Core i9-14900K Prozessor

- Performance Cores 8 (× 2 Hyperthreading)
- Taktfrequenz 3,2 GHz (max. 6,0 GHz)
- L1 Cache 8 × 32 KiB I + 48 KiB D
- L2 Cache 8 × 2,0 MiB (I+D)
- Efficiency Cores 16
- Taktfrequenz 2,4 GHz (max. 4,4 GHz)
- L1 Cache 16 × 32 KiB I + 64 KiB D
- L2 Cache 4 × 2 MiB (I+D)
- L3 Cache 36 MiB (I+D)
- Memory Controller 2 × 44,8 GiB/s
- DMI Durchsatz 16 GT/s ↔ Chipsatz
- Prozess 7 nm
- Größe 257 mm²
- Leistungsaufnahme 125 W (< 253 W)

Quellen: ark.intel.com, www.intel.de
en.wikichip.org



x86: Speichermodell

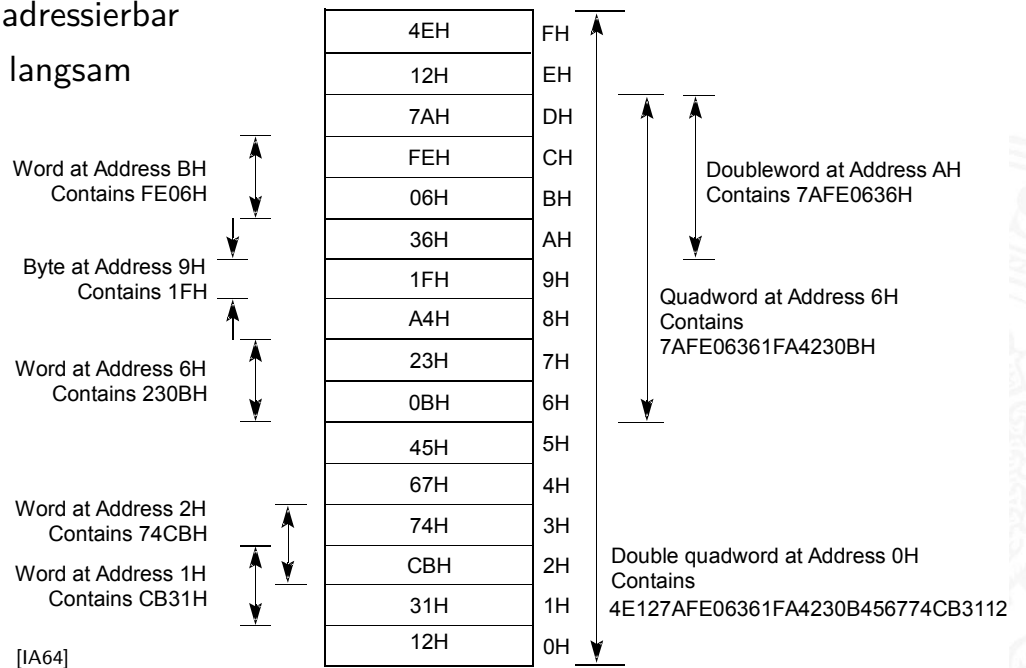


► „Little Endian“: LSB eines Wortes bei der kleinsten Adresse



x86: Speichermodell (cont.)

- ▶ Speicher voll byte-adressierbar
- ▶ misaligned Zugriffe langsam
- ▶ Beispiel

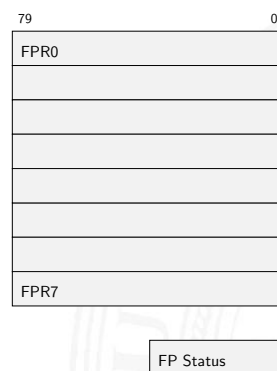


x86: Register

63	31	15	0		
RAX	EAX	AX	AH AL	accumulator	
RBX	EBX	BX	BH BL	base addr	
RCX	ECX	CX	CH CL	count: String, Loop	
RDX	EDX	DX	DH DL	data, multiply/divide	
RSI	ESI	SI		index, string src	
RDI	EDI	DI		index, string dst	
RSP	ESP	SP		stackptr	
RBP	EBP	BP		base of stack segment	
		CS		code segment	
		SS		stack segment	
		DS		data segment	
		ES		extra data segment	
		FS			
		GS			
RIP	EIP	IP		PC	
	EFLAGS			status	
64-bit Mode		32-bit Mode		16-bit Mode	
R8 ... R15	...D	...W	...L		

Legend for register shading:

- Dark grey: 8086
- Light grey: E... ab 386
- White: R... x86-64





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
- ⇒ extrem aufwändige Decodierung
- ⇒ CISC – **C**omplex **I**nstruction **S**et **C**omputer

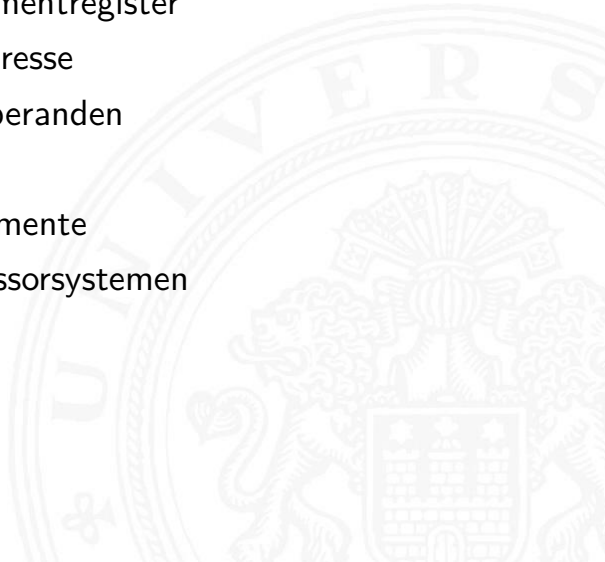


x86: Befehlsformat-Modifizier („prefix“)

- ▶ alle Befehle können mit Modifiern ergänzt werden

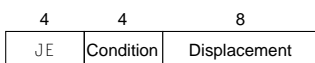
segment override Adresse aus angewähltem Segmentregister
 address size Umschaltung 16/32/64-bit Adresse
 operand size Umschaltung 16/32/64-bit Operanden

repeat Stringoperationen: für alle Elemente
 lock Speicherschutz bei Multiprozessorsystemen



x86 Befehlskodierung: Beispiele

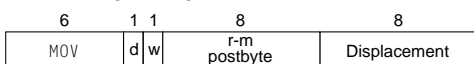
a. JE EIP + displacement



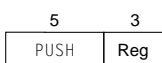
b. CALL



c. MOV EBX, [EDI + 45]



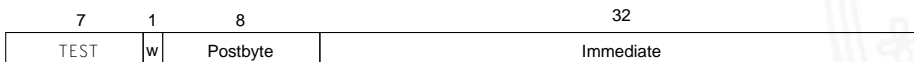
d. PUSH ESI



e. ADD EAX, #6765

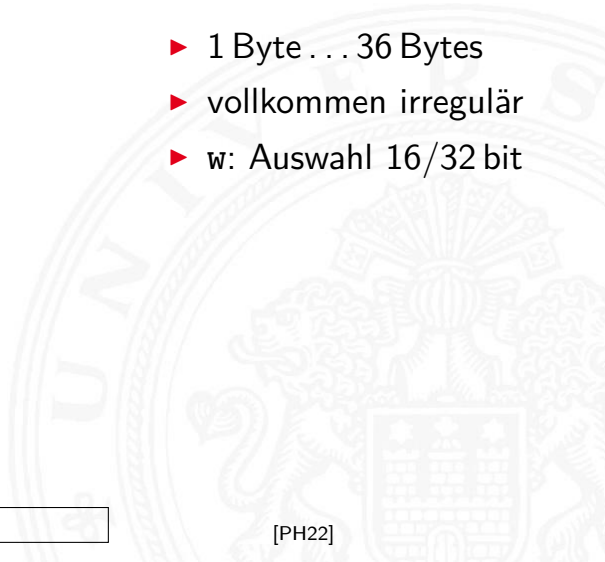


f. TEST EDX, #42



- ▶ 1 Byte . . . 36 Bytes
- ▶ vollkommen irregulär
- ▶ w: Auswahl 16/32 bit

[PH22]





x86 Befehlskodierung: 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

[PH22]



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 0FB003   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	0FB03	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 ⇔ 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.)

► MIPS-Prozessor

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%

[HP17]

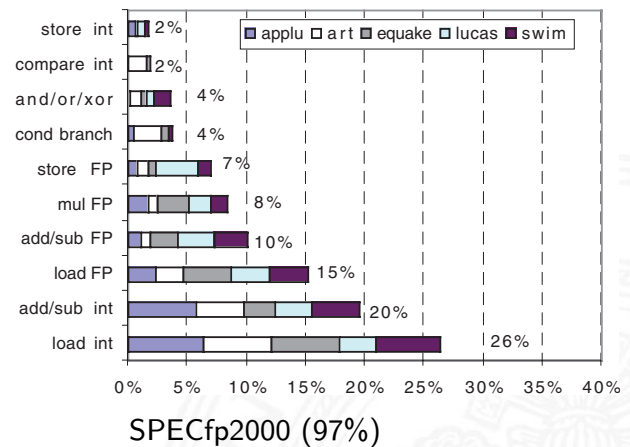
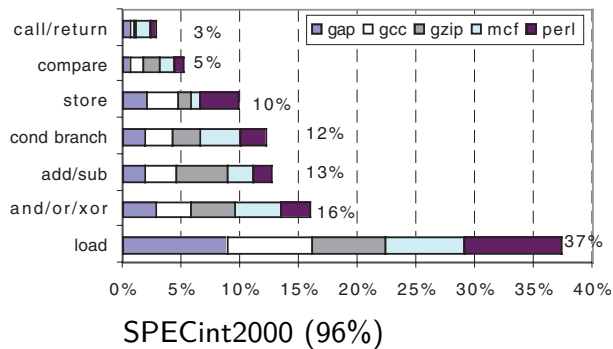
Figure D.15 80x86 instruction mix for five SPECint92 programs.



Bewertung der ISA (cont.)

12.6 Instruction Set Architecture - Befehlssätze

64-040 Rechnerstrukturen und Betriebssysteme



- ▶ über 80 % der Berechnungen eines typischen Programms verwenden nur ca. 20 % der Instruktionen einer CPU
 - ▶ sehr einfache Instruktionen werden am häufigsten gebraucht: load, store, add ...
- ⇒ Motivation für RISC



CISC – Complex Instruction Set Computer

12.6 Instruction Set Architecture - Befehlssätze

64-040 Rechnerstrukturen und Betriebssysteme

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 Befehlskodierung
 - ▶ mehrere Schreib- und Lesezugriffe pro Befehl
- ▶ viele verschiedene Datentypen



CISC – Complex Instruction Set Computer (cont.)

12.6 Instruction Set Architecture - Befehlssätze

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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.)

12.6 Instruction Set Architecture - Befehlssätze

64-040 Rechnerstrukturen und Betriebssysteme

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



- ▶ ein Befehl kann nicht in einem Takt abgearbeitet werden
- ⇒ Unterteilung in Mikroinstruktionen (\varnothing 5...7)
- ▶ Ablaufsteuerung durch endlichen Automaten
- ▶ meist als ROM (RAM) implementiert, das *Mikroprogrammwort*e beinhaltet

1. horizontale Mikroprogrammierung

- ▶ langes Mikroprogrammwort (ROM-Zeile)
- ▶ steuert direkt alle Operationen
- ▶ Spalten entsprechen: Kontrollleitungen und Folgeadressen

▶ horizontale Mikroprog.



2. vertikale Mikroprogrammierung

- ▶ kurze Mikroprogrammwort
- ▶ Spalten enthalten Mikrooperationscode
- ▶ mehrstufige Decodierung für Kontrollleitungen

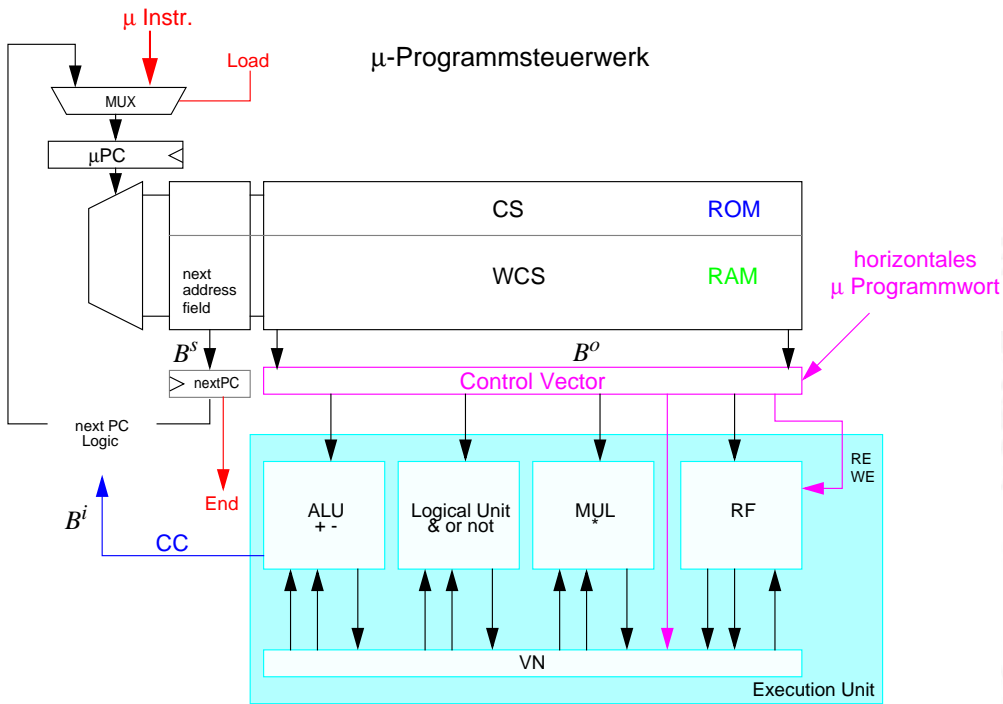
▶ vertikale Mikroprog.

- + CISC-Befehlssatz mit wenigen Mikrobefehlen realisieren
- + μ -Programm im RAM: Mikrobefehlssatz austauschbar
- (mehrstufige) ROM/RAM Zugriffe: zeitaufwändig

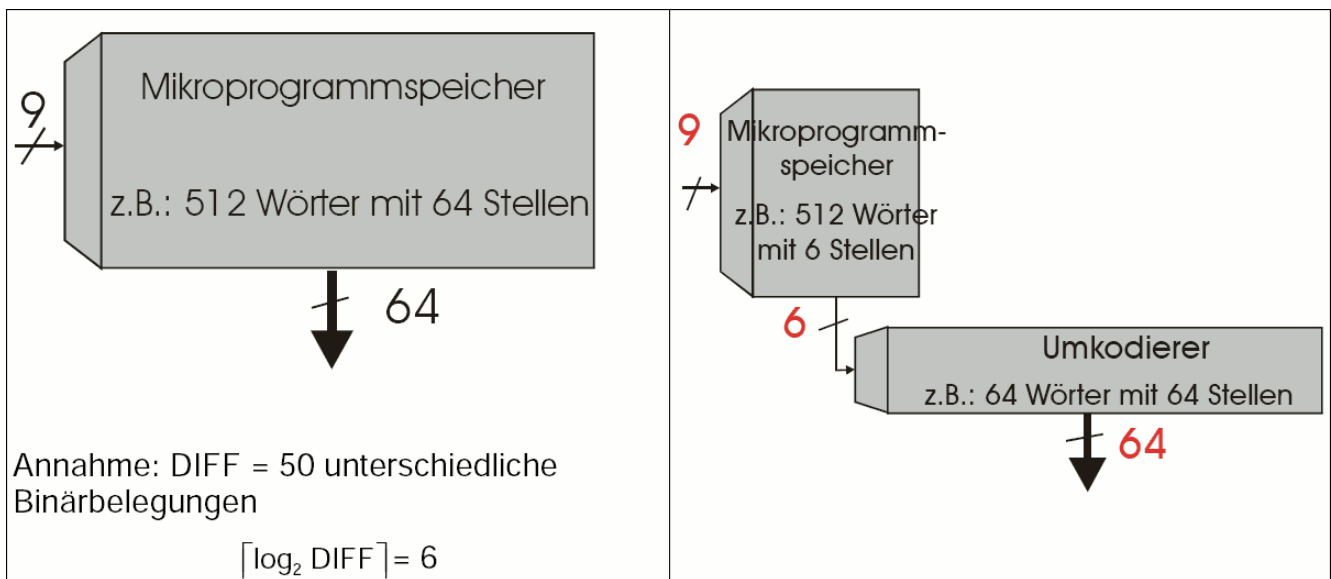
⇒ wird inzwischen nur noch benutzt, um CISC Befehle in RISC-artige Sequenzen umzusetzen (x86)



horizontale Mikroprogrammierung



vertikale Mikroprogrammierung





RISC – Reduced Instruction Set Computer

oft auch: „Regular Instruction Set Computer“

- ▶ seit den 80er Jahren: „RISC-Boom“
 - ▶ internes Projekt bei IBM
 - ▶ von Hennessy (Stanford) und Patterson (Berkeley) publiziert
- ▶ Grundidee: Komplexitätsreduktion in der CPU
- ▶ optimierende Compiler statt Assemblerprogrammierung
 - ⇒ statt Mikroprogramm in CISC: mehrere, einfache Assemblerbefehle
 - ⇒ CISC Komplexität der Befehle wird durch Compiler abgelöst

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, sehr schneller Hardware ausgeführt



RISC – Reduced Instruction Set Computer (cont.)

- ▶ reguläre Struktur, z.B. 32-bit Wortbreite, 32-bit Befehle
- ▶ nur ein-Wort Befehle
- ▶ alle Befehle in gleicher Zeit ausführbar ⇒ 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



RISC – Reduced Instruction Set Computer (cont.)

Vor- / Nachteile

- + fest-verdrahtete Logik, kein Mikroprogramm
- + einfache Instruktionen, wenige Adressierungsarten
- + Pipelining gut möglich
- + Cycles per Instruction = 1, bei Superskalarität > 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



- ▶ 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“)
- ⇒ neuer Befehlssatz mit expliziter Parallelität (EPIC)
 - ⇒ 64-bit Wortgrößen (überwinden Adressraumlimits)
 - ⇒ benötigt hoch entwickelte Compiler



- [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
- [PH22] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle – MIPS Edition.*
6. Auflage, De Gruyter Oldenbourg, 2022. ISBN 978-3-11-075598-5
- [PH20] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – RISC-V Edition.*
2nd edition, Morgan Kaufmann Publishers Inc., 2020. ISBN 978-0-12-820331-6



- [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
- [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



1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. Logische Operationen
7. Codierung
8. Schaltfunktionen
9. Schaltnetze
10. Schaltwerke
11. Rechnerarchitektur I
12. Instruction Set Architecture



13. Assembler-Programmierung

Motivation

Grundlagen der Assemblerebene

x86 Assembler

Elementare Befehle + Adressierung

Operationen

Kontrollfluss

Sprungbefehle und Schleifen

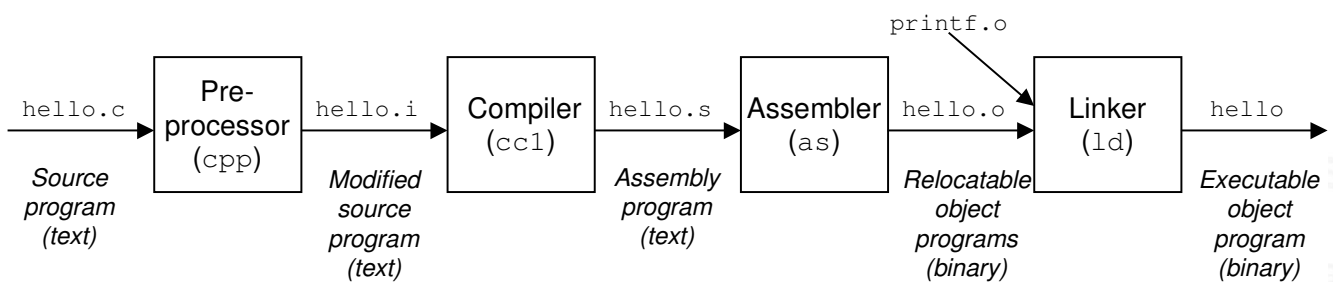
Funktionsaufrufe und Stack

Linker und Loader

Literatur

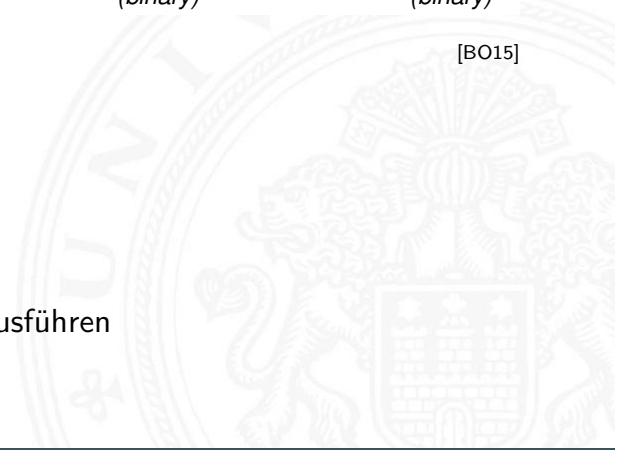
14. Rechnerarchitektur II

15. Betriebssysteme



[B015]

- ▶ verschiedene Repräsentationen des Programms
 - ▶ Hochsprache
 - ▶ Assembler
 - ▶ Maschinsprache
- ▶ Ausführung der Maschinsprache
 - ▶ 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

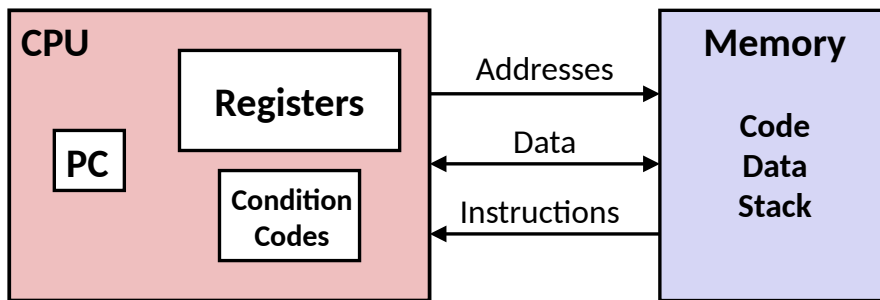
aber **Grundwissen** bleibt trotzdem **unverzichtbar**

- ▶ 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
 - ▶ Speicherverwaltung der Datentypen
 - ▶ Behandlung von Makros
 - ▶ Implementierung eines Assemblers (2-pass)
 - ▶ Tipps für effizientes Programmieren
 - ▶ Befehle für die Systemprogrammierung (supervisor mode)
 - ▶ x86 Gleitkommabefehle
 - ▶ ...



[BO15]

beobachtbare Zustände

- ▶ Programmzähler (*Instruction Pointer*)
 - ▶ Adresse der nächsten Anweisung
- ▶ Registerbank
 - ▶ häufig benutzte Programmdate
- ▶ Zustandscodes
 - ▶ Statusinformationen über die letzte ALU Operation
 - ▶ für bedingte Sprünge benötigt (*Conditional Branch*)

x86-64 rip Register

rax ... rbp Register

r8 ... r15 Register

EFLAGS Register



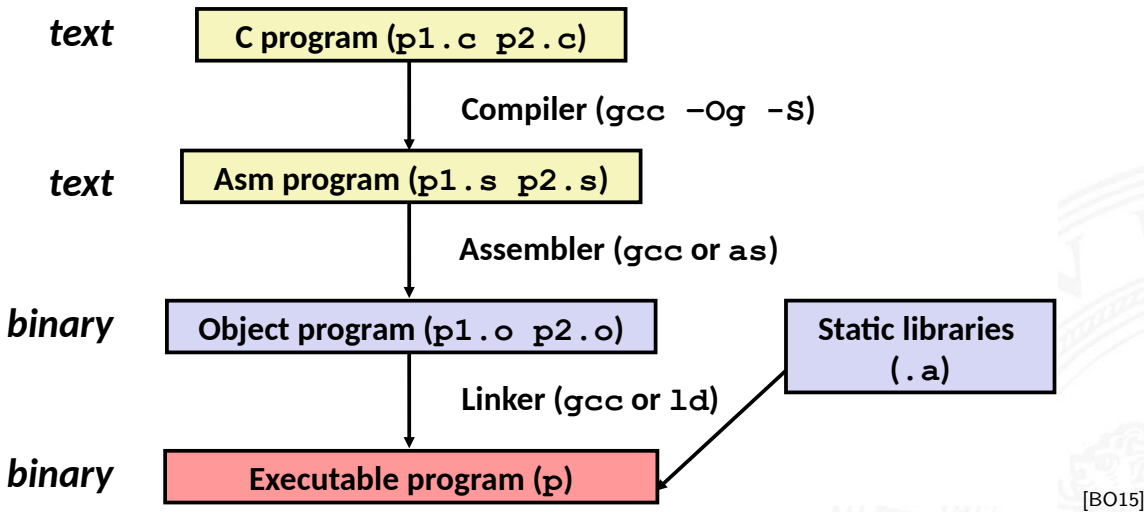
- ▶ Speicher
 - ▶ byteweise adressierbares Array
 - ▶ pro Task: Code, Nutzerdaten, (einige) OS Daten
 - ▶ -- Kellerspeicher für Unterprogrammaufrufe
 - ▶ -- Adressraum dynamisch allozierter Datenstrukturen

„Stack“

„Heap“



Umwandlung von C in Objektcode



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
 - ▶ Datenwerte für Variablen
 - ▶ positiv oder vorzeichenbehaftet
 - ▶ Textzeichen (ASCII, Unicode)
 - ▶ Gleitkomma-Daten mit 4 oder 8 Bytes
 - ▶ Adressen bzw. „Pointer“
- 8...64 bits
int/long/long long
signed/unsigned
char
float/double
untypisierte Speicheradressen



- ▶ 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



- ▶ 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

- ▶ 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 den Speicher, bzw. zur Laufzeit erstellt



Beispiel: Maschinenbefehl für Speichern

- ▶ C-Code `*dest = t;`
 - ▶ speichert Wert `t` nach Adresse aus `dest`

```
*dest = t;
```

- ▶ 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]

```
movq %rax, (%rbx)
```

- ▶ Objektcode (x86-Befehlssatz)
 - ▶ 3-Byte Befehl
 - ▶ an Speicheradresse `0x40059e`

```
0x40059e: 48 89 03
```




Objektcode Disassembler: objdump

[BO15]

```
000000000400595 <sumstore>:
400595: 53          push   %rbx
400596: 48 89 d3    mov    %rdx,%rbx
400599: e8 f2 ff ff callq  400590 <plus>
40059e: 48 89 03    mov    %rax,(%rbx)
4005a1: 5b          pop    %rbx
4005a2: c3          retq
```

▶ objdump -d ...

- ▶ Werkzeug zur Untersuchung des Objektcodes
- ▶ rekonstruiert aus Binärcode den Assemblercode
- ▶ kann auf vollständigem, ausführbarem Programm (*executable*) oder einer .o Datei ausgeführt werden



Was kann „disassembliert“ werden?

[BO15]

```
% 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
```

- ▶ alles, was als ausführbarer Code interpretiert werden kann
- ▶ Disassembler untersucht Bytes und rekonstruiert Assemblerquelle (soweit möglich)



- ▶ 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)

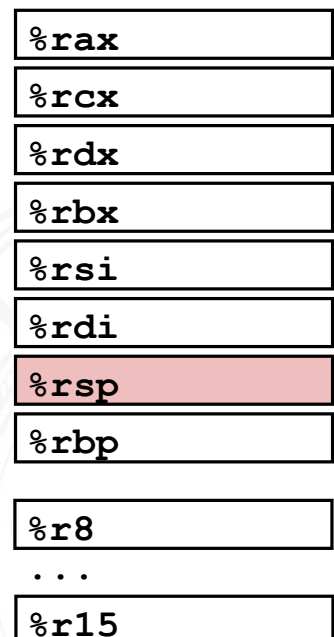
Grafiken und Beispiele dieses Abschnitts sind aus

R.E. Bryant, D.R. O'Hallaron: *Computer systems – A programmers perspective* [BO15]

bzw. dem zugehörigen Foliensatz



- ▶ 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: $R \rightarrow \text{Reg}[R]$
 - ▶ Inhalt eines der 16 Universalregister %rax ... %r15
 - ▶ Registername R beginnt immer mit %
- ▶ Normal: $(R) \rightarrow \text{Mem}[\text{Reg}[R]]$
 - ▶ Register R spezifiziert die Speicheradresse
 - ▶ Beispiel: movq (%rcx), %rax
- ▶ Displacement: $D(R) \rightarrow \text{Mem}[\text{Reg}[R]+D]$
 - ▶ Register R als Basis-Speicheradresse
 - ▶ Konstantes „Displacement“ D spezifiziert den „offset“
 - ▶ Beispiel: movq 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()

Register

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Speicher

	Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

```

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()

Register

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	

Speicher

Address	
0x120	123
0x118	
0x110	
0x108	
0x100	456

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()

Register

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Speicher

Address	
0x120	123
0x118	
0x110	
0x108	
0x100	456

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()

Register

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Speicher

Address	
0x120	456
0x118	
0x110	
0x108	
0x100	456

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()

Register

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Speicher

Address	
0x120	456
0x118	
0x110	
0x108	
0x100	123

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
 - ▶ $\text{Imm}(\text{Rb}, \text{Ri}, \text{S}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}] + \text{S} * \text{Reg}[\text{Ri}] + \text{Imm}]$
 - ▶ $\langle \text{Imm} \rangle$ Offset
 - ▶ $\langle \text{Rb} \rangle$ Basisregister: eines der 16 Integer-Register
 - ▶ $\langle \text{Ri} \rangle$ Indexregister: jedes außer %rsp
%rbp grundsätzlich möglich, jedoch unwahrscheinlich
 - ▶ $\langle \text{S} \rangle$ Skalierungsfaktor 1, 2, 4 oder 8

- ▶ gebräuchlichste Fälle
 - ▶ $(\text{Rb}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}]]$
 - ▶ $\text{Imm}(\text{Rb}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}] + \text{Imm}]$
 - ▶ $(\text{Rb}, \text{Ri}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}] + \text{Reg}[\text{Ri}]]$
 - ▶ $\text{Imm}(\text{Rb}, \text{Ri}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}] + \text{Reg}[\text{Ri}] + \text{Imm}]$
 - ▶ $(\text{Rb}, \text{Ri}, \text{S}) \rightarrow \text{Mem}[\text{Reg}[\text{Rb}] + \text{S} * \text{Reg}[\text{Ri}]]$



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



► binäre Operatoren

Format	Berechnung
addq $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle + \langle src \rangle$
subq $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle - \langle src \rangle$
imulq $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle * \langle src \rangle$
salq $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \ll \langle src \rangle$ auch shlq
sarq $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \gg \langle src \rangle$ arithmetisch
shrq $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \gg \langle src \rangle$ logisch
xorq $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \wedge \langle src \rangle$
andq $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \& \langle src \rangle$
orq $\langle src \rangle, \langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle \mid \langle src \rangle$



► unäre Operatoren

Format	Berechnung
incq $\langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle + 1$
decq $\langle dst \rangle$	$\langle dst \rangle = \langle dst \rangle - 1$
negq $\langle dst \rangle$	$\langle dst \rangle = -\langle dst \rangle$
notq $\langle dst \rangle$	$\langle dst \rangle = \sim \langle dst \rangle$

► leaq-Befehl: *load effective address*

leaq $\langle src \rangle, \langle dst \rangle$

- Adressberechnung für (späteren) Ladebefehl
- Speichert die Adressausdruck $\langle src \rangle$ in Register $\langle dst \rangle$
 $\text{Imm}(\text{Rb}, \text{Ri}, \text{S}) \rightarrow \text{Reg}[\text{Rb}] + \text{S} * \text{Reg}[\text{Ri}] + \text{Imm}$
- wird oft von Compilern für arithmetische Berechnung genutzt
s. Beispiele



Beispiel: arithmetische Operationen

13.3.2 Assembler-Programmierung - x86 Assembler - Operationen

64-040 Rechnerstrukturen und Betriebssysteme

```

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

13.3.3 Assembler-Programmierung - x86 Assembler - Kontrollfluss

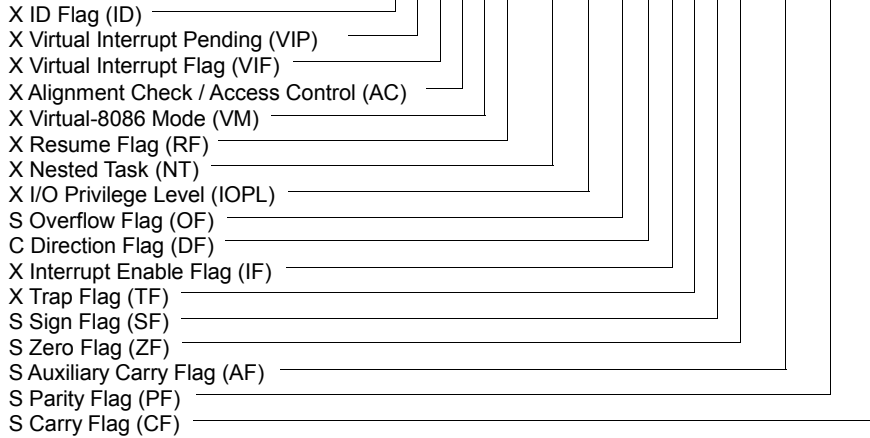
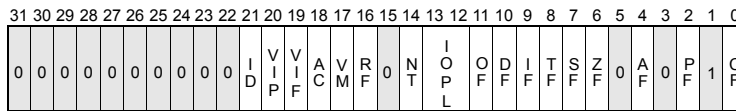
64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Zustandscodes
 - ▶ Setzen
 - ▶ Testen

- ▶ Ablaufsteuerung
 - ▶ Verzweigungen: „If-then-else“
 - ▶ Schleifen: „Loop“-Varianten
 - ▶ Mehrfachverzweigungen: „Switch“



x86: EFLAGS Register



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.

► x86-64: RFLAGS \cong EFLAGS, mit „0“ erweitert

[IA64]



Prozessor aus Sicht des Programmierers

- temporäre Daten, Standardregister
%rax, ...
- Top of Stack
%rsp
- Programmzähler
%rip
- Flag-Bits
CF, ZF, SF, OF

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

%rip

CF	ZF	SF	OF
----	----	----	----



- ▶ 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 <src>, <dst>` 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
 $(a > 0 \ \&\& \ b > 0 \ \&\& \ t < 0) \ || \ (a < 0 \ \&\& \ b < 0 \ \&\& \ t \geq 0)$



2. explizites Setzen durch Vergleichsoperation

- ▶ Beispiel: `cmpq <src2>, <src1>`
wie Berechnung von `<src1> - <src2>` (subq <src2>, <src1>)
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
 $(a > 0 \ \&\& \ b < 0 \ \&\& \ (a - b) < 0) \ || \ (a < 0 \ \&\& \ b > 0 \ \&\& \ (a - b) \geq 0)$



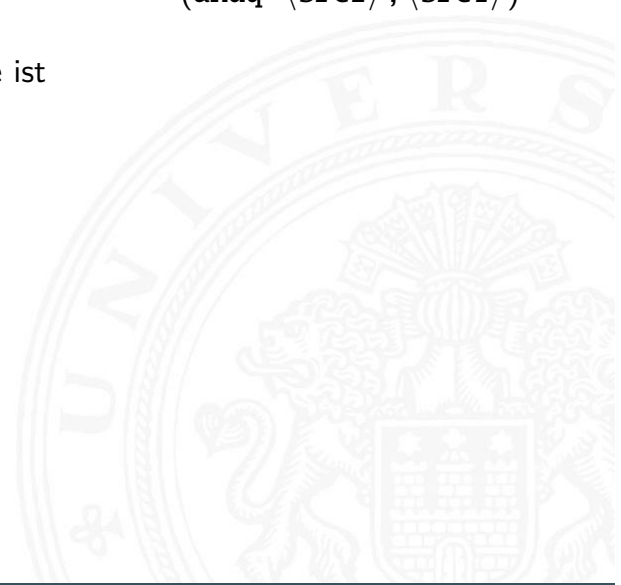
3. explizites Setzen durch Testanweisung

- ▶ Beispiel: `testq <src2>, <src1>`
wie Berechnung von `<src1> & <src2>`
jedoch ohne Abspeichern des Resultats

`(andq <src2>, <src1>)`

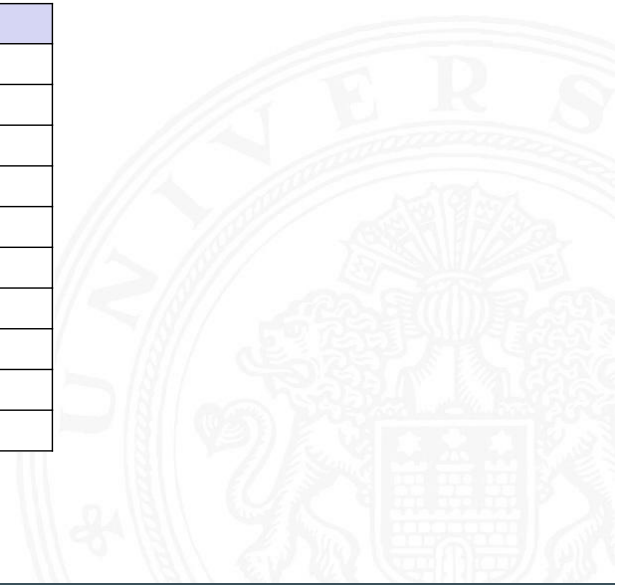
⇒ hilfreich, wenn einer der Operanden eine Bitmaske ist

- ▶ ZF setzen wenn $src1 \& src2 = 0$
- ▶ SF setzen wenn $src1 \& src2 < 0$



- ▶ Befehle setzen ein einzelnes Byte (LSB) in Universalregister
- ▶ die anderen 7-Bytes werden nicht verändert

SetX	Condition	Description
<code>sete</code>	ZF	Equal / Zero
<code>setne</code>	~ZF	Not Equal / Not Zero
<code>sets</code>	SF	Negative
<code>setns</code>	~SF	Nonnegative
<code>setg</code>	~(SF^OF) & ~ZF	Greater (Signed)
<code>setge</code>	~(SF^OF)	Greater or Equal (Signed)
<code>setl</code>	(SF^OF)	Less (Signed)
<code>setle</code>	(SF^OF) ZF	Less or Equal (Signed)
<code>seta</code>	~CF & ~ZF	Above (unsigned)
<code>setb</code>	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
<code>jmp</code>	<code>1</code>	Unconditional
<code>jbe</code>	<code>ZF</code>	Equal / Zero
<code>jne</code>	<code>~ZF</code>	Not Equal / Not Zero
<code>js</code>	<code>SF</code>	Negative
<code>jns</code>	<code>~SF</code>	Nonnegative
<code>jg</code>	<code>~(SF^OF) & ~ZF</code>	Greater (Signed)
<code>jge</code>	<code>~(SF^OF)</code>	Greater or Equal (Signed)
<code>jl</code>	<code>(SF^OF)</code>	Less (Signed)
<code>jle</code>	<code>(SF^OF) ZF</code>	Less or Equal (Signed)
<code>ja</code>	<code>~CF & ~ZF</code>	Above (unsigned)
<code>jb</code>	<code>CF</code>	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
- ▶ keine Sprünge mehr \Rightarrow gut für Pipelining

-fif-conversion

```

long absdiff
(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}

```

```

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

```

Register	Funktion
%rdi	Argument x
%rsi	Argument y
%rax	Rückgabewert



if Übersetzung – conditional move (cont.)

- ▶ beide Ausdrücke werden berechnet
 - + Parallelisierung durch Hardware möglich (*Superskalarität*)
 - Performanz bei komplizierter Berechnung
 - Unsicher, da Seiteneffekte!
- ▶ Codeäquivalent

```

val = Test
? Then_Expr
: Else_Expr;

```

```

result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;

```



andere Kontrollstrukturen

- ▶ `while...`, `until...`, `for...` Schleifen
 - ▶ können durch Kombinationen von Sprüngen ersetzt werden
 - ▶ bedingte und unbedingte Vorwärts-/Rückwärtssprünge
- ▶ `case...` Mehrfachverzweigungen
 - ▶ Bedingung wird in Zahl umgesetzt
 - ▶ entspricht Offset in Tabelle mit Sprungzielen
 - ▶ dann unbedingter Sprung
- ▶ jetzt weiter mit Unterprogrammen...



Mechanismen für Prozeduraufrufe

- ▶ Kontrollübergabe
 - ▶ zum Unterprogrammcode
 - ▶ zurück zum Aufruf
- ▶ Datenübergabe
 - ▶ der Argumente
 - ▶ für 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

13.3.5 Assembler-Programmierung - x86 Assembler - Funktionsaufrufe und Stack

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ **Kontrollübergabe**
 - ▶ zum Unterprogrammcode
 - ▶ zurück zum Aufruf
- ▶ **Datenübergabe**
 - ▶ der Argumente
 - ▶ für 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

13.3.5 Assembler-Programmierung - x86 Assembler - Funktionsaufrufe und Stack

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ **Kontrollübergabe**
 - ▶ zum Unterprogrammcode
 - ▶ zurück zum Aufruf
- ▶ **Datenübergabe**
 - ▶ der Argumente
 - ▶ für 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 Prozeduraufufe

13.3.5 Assembler-Programmierung - x86 Assembler - Funktionsaufrufe und Stack

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Kontrollübergabe
 - ▶ zum Unterprogrammcode
 - ▶ zurück zum Aufruf
- ▶ Datenübergabe
 - ▶ der Argumente
 - ▶ für 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 Prozeduraufufe

13.3.5 Assembler-Programmierung - x86 Assembler - Funktionsaufrufe und Stack

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Kontrollübergabe
 - ▶ zum Unterprogrammcode
 - ▶ zurück zum Aufruf
- ▶ Datenübergabe
 - ▶ der Argumente
 - ▶ für Rückgabewert
- ▶ Speicherverwaltung
 - ▶ Allokation während der Ausführung
 - ▶ Freigabe nach `return`

```
P(...) {  
  .  
  .  
  y = Q(x);  
  print(y)  
  .  
}
```

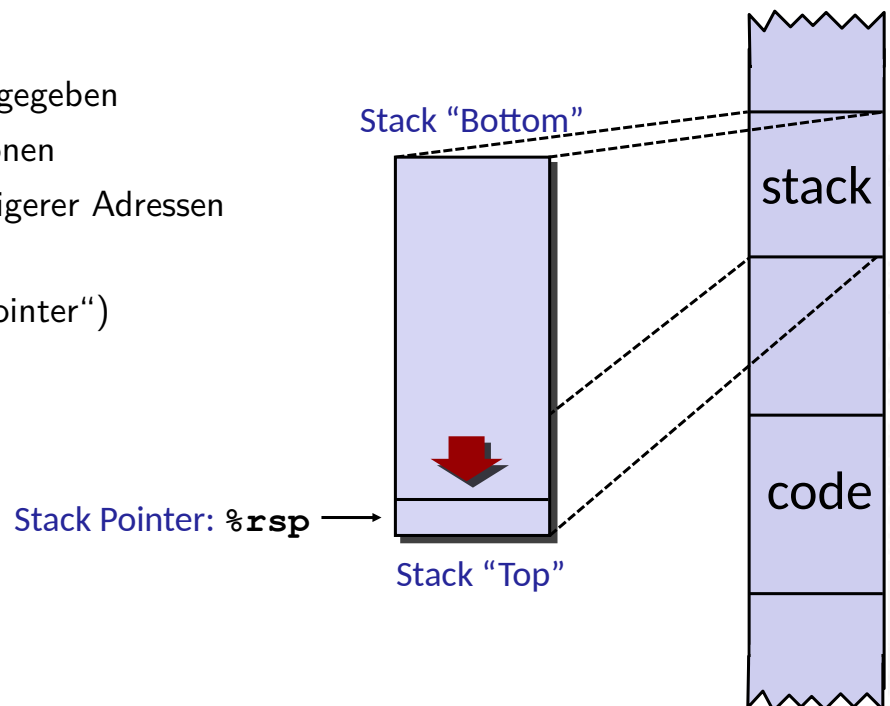
```
int Q(int i)  
{  
  int t = 3*i;  
  int v[10];  
  .  
  .  
  return v[t];  
}
```



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.)

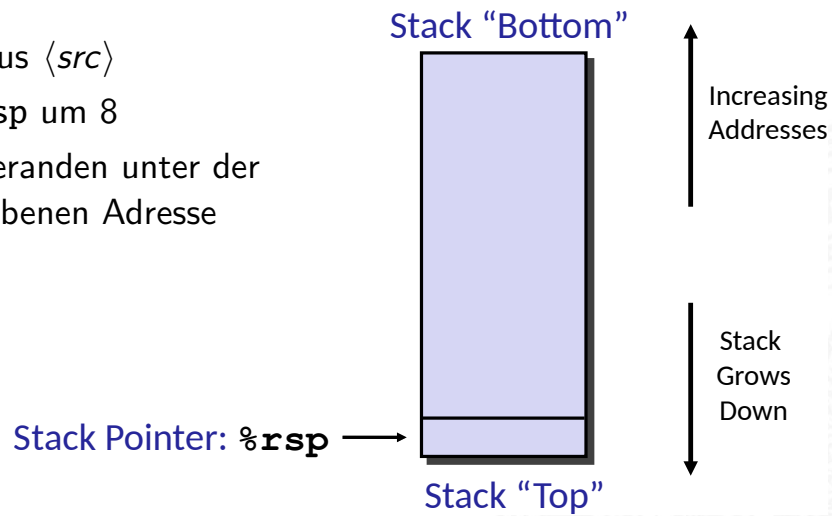
- ▶ Implementierung von Funktionen/Prozeduren
 - ▶ Speicherplatz für Aufruf-Parameter
 - ▶ Speicherplatz für lokale Variablen
 - ▶ Rückgabe der Funktionswerte
 - ▶ für rekursive Funktionen benötigt!
- ▶ mehrere Varianten/Konventionen
 - ▶ Parameterübergabe in Registern
 - ▶ „Caller-Save“
 - ▶ „Callee-Save“
 - ▶ Kombinationen davon
 - ▶ Aufruf einer Funktion muss deren Konvention berücksichtigen



Stack: Push

`pushq <src>`

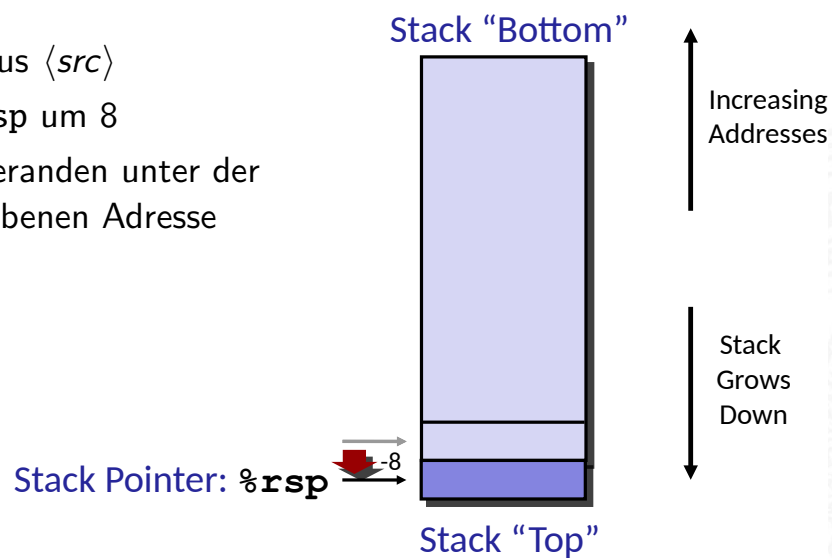
- ▶ holt Operanden aus `<src>`
- ▶ dekrementiert `%rsp` um 8
- ▶ speichert den Operanden unter der von `%rsp` vorgegebenen Adresse



Stack: Push

`pushq <src>`

- ▶ holt Operanden aus `<src>`
- ▶ dekrementiert `%rsp` um 8
- ▶ speichert den Operanden unter der von `%rsp` vorgegebenen Adresse

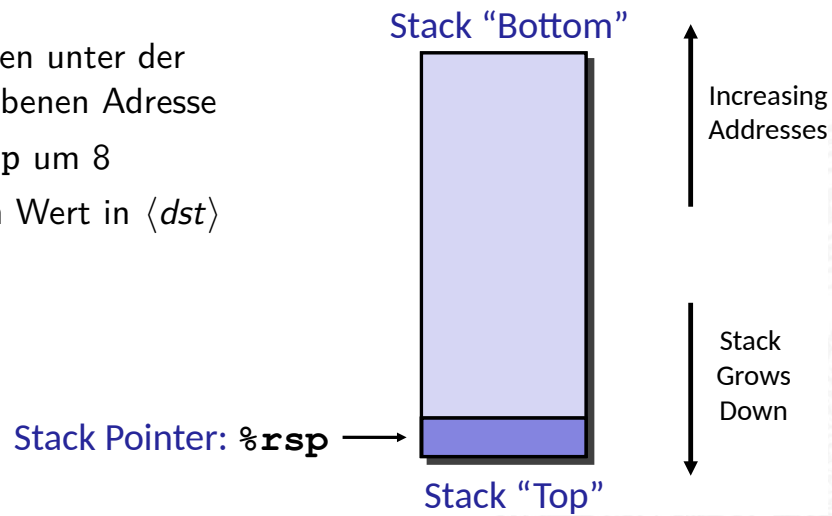




Stack: Pop

`popq <dst>`

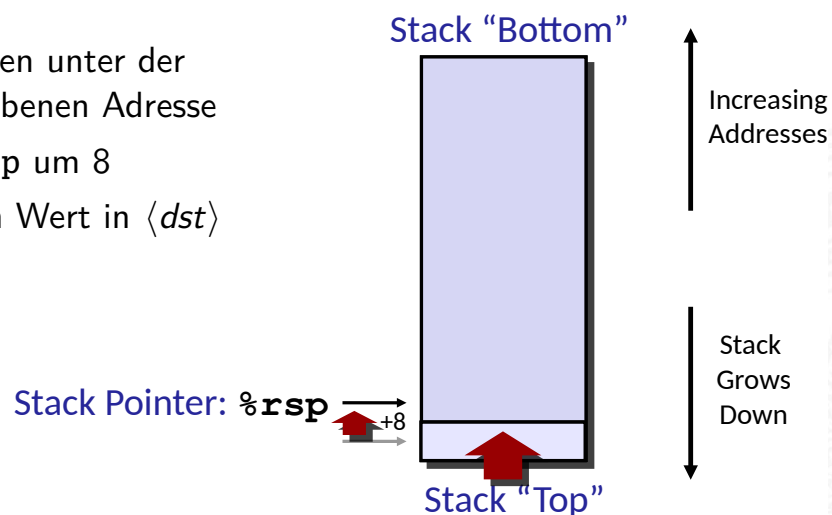
- ▶ liest den Operanden unter der von `%rsp` vorgegebenen Adresse
- ▶ inkrementiert `%rsp` um 8
- ▶ schreibt gelesenen Wert in `<dst>`



Stack: Pop

`popq <dst>`

- ▶ liest den Operanden unter der von `%rsp` vorgegebenen Adresse
- ▶ inkrementiert `%rsp` um 8
- ▶ schreibt gelesenen Wert in `<dst>`





- ▶ 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
- ▶ Register mit Spezialaufgaben
 - ▶ `%rsp` „stack-pointer“: Speicheradresse des top-of-stack
 - ▶ `%rbp` „base-pointer“: Speicheradresse des aktuellen Frame
- ▶ Prozeduraufruf: `call <label>`
 - ▶ Rücksprungadresse auf Stack („Push“)
 - ▶ Sprung zu `<label>`
- ▶ Rücksprung: `ret`
 - ▶ Rücksprungadresse vom Stack („Pop“)
 - ▶ Sprung zu dieser Adresse



- ▶ Sprungadressen
 - ▶ Unterprogramm: Adresse der ersten Programmanweisung `<label>`
 - ▶ Rücksprung: Adresse der auf den `call` folgenden Anweisung
- ▶ Stack zur Unterstützung von `call` und `ret`
 - ▶ ggf. Parameter
 - ▶ Rücksprungadresse
 - ▶ lokale Variablen
 - ▶ ggf. Rückgabewerte



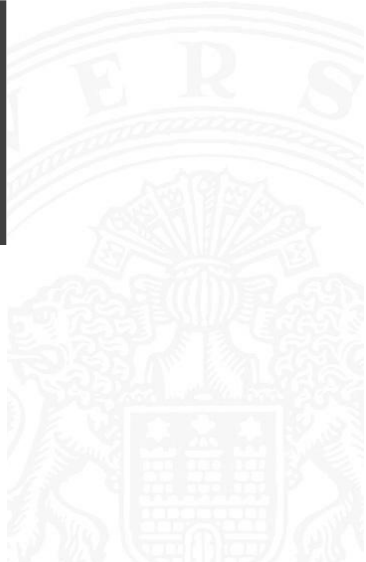
Codebeispiel Unterprogrammaufruf

```
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
```

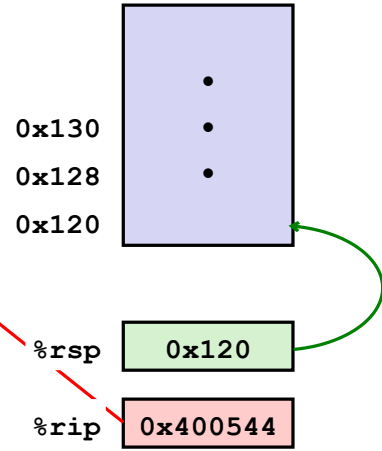


Kontrollübergabe

► Prozeduraufruf callq

```
0000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov   %rax,(%rbx)
.
.
```

```
0000000000400550 <mult2>:
400550: mov   %rdi,%rax
.
.
400557: retq
```



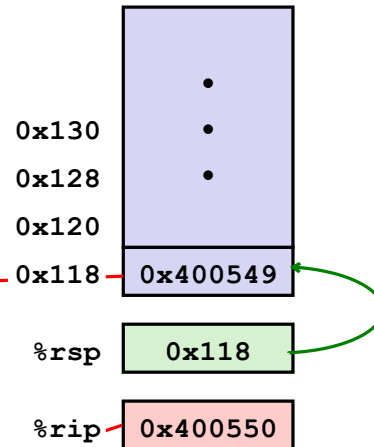


Kontrollübergabe (cont.)

- ▶ Rücksprungadresse auf Stack
- ▶ Programmzähler setzen %rip

```
0000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov  %rax, (%rbx)
.
.
```

```
0000000000400550 <mult2>:
400550: mov  %rdi,%rax
.
.
400557: retq
```

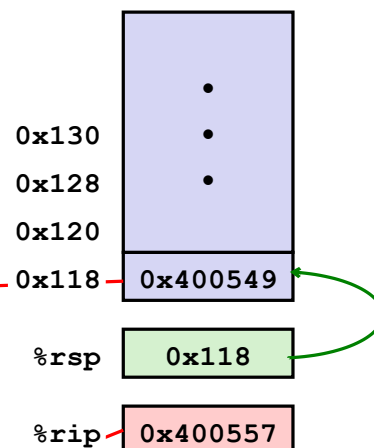


Kontrollübergabe (cont.)

- ▶ Rücksprung retq

```
0000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov  %rax, (%rbx)
.
.
```

```
0000000000400550 <mult2>:
400550: mov  %rdi,%rax
.
.
400557: retq
```



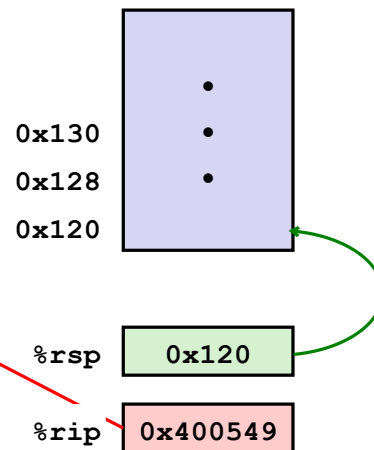


Kontrollübergabe (cont.)

- ▶ Rücksprungadresse vom Stack
- ▶ Programmzähler setzen %rip

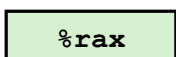
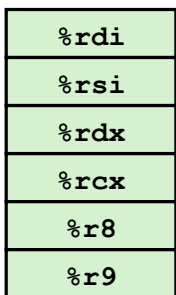
```
0000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov  %rax, (%rbx)
.
.
```

```
0000000000400550 <mult2>:
400550: mov  %rdi,%rax
.
.
400557: retq
```

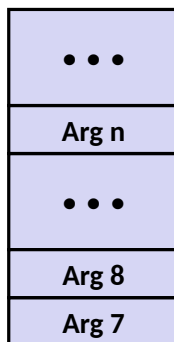


Datenübergabe

- ▶ Register



- ▶ Stack



- ▶ Konvention

- ▶ die ersten 6 Argumente: Register %rdi, %rsi...
- ▶ Stack wenn notwendig
- ▶ Rückgabewert: Register %rax





Datenübergabe (cont.)

13.3.5 Assembler-Programmierung - x86 Assembler - Funktionsaufrufe und Stack

64-040 Rechnerstrukturen und Betriebssysteme

```

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

```

A. Mäder

956



Stack-basierende Programmierung

13.3.5 Assembler-Programmierung - x86 Assembler - Funktionsaufrufe und Stack

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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 aufrufenden Programm („Caller“) beendet
- ▶ Stack-„Frame“
 - ▶ der Bereich/Zustand einer einzelnen Prozedur-Instanziierung

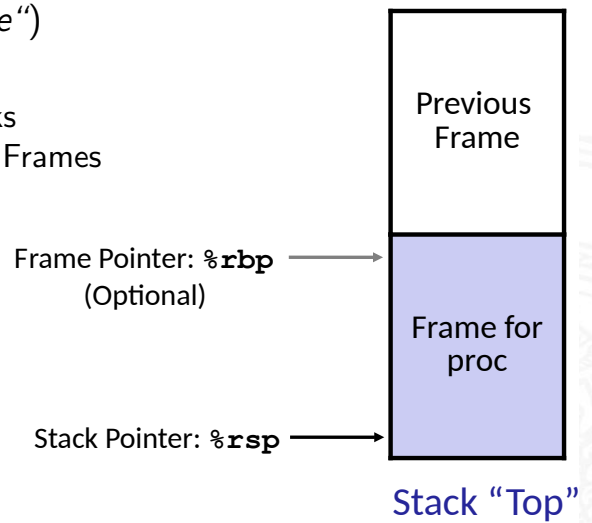
A. Mäder

957



Stack-Frame

- ▶ alle Daten für einen Funktionsaufruf („Closure“)
- ▶ Adressverweise („Pointer“)
 - ▶ Stackpointer `%rsp`: das obere Ende des Stacks
 - ▶ Framepointer `%rbp`: der Anfang des aktuellen Frames
- ▶ Daten
 - ▶ ggf. Aufruf-Parameter
 - ▶ 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 ();
  .
  .
}

```

```

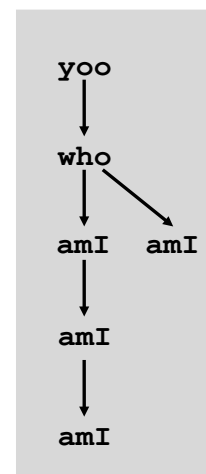
who (...)
{
  . . .
  amI ();
  . . .
  amI ();
  . . .
}

```

```

amI (...)
{
  .
  .
  amI ();
  .
  .
}

```

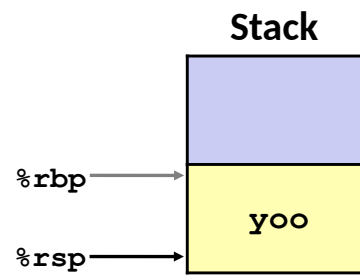
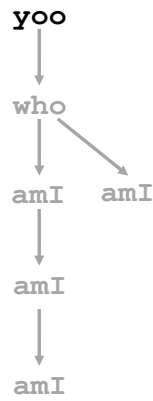
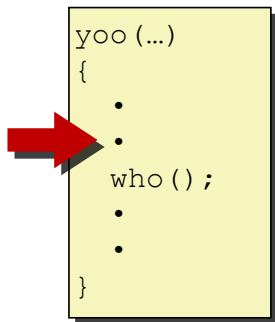




Beispiel: Prozeduraufrufe (cont.)

13.3.5 Assembler-Programmierung - x86 Assembler - Funktionsaufrufe und Stack

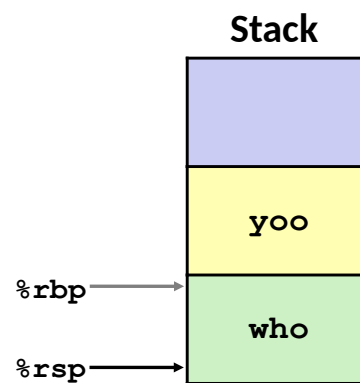
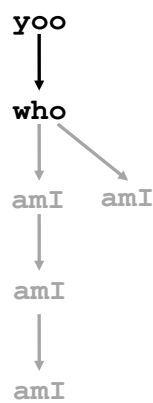
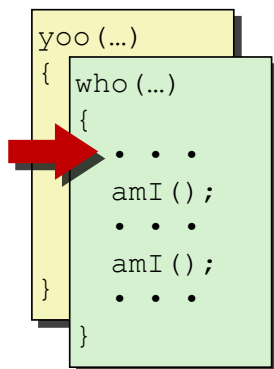
64-040 Rechnerstrukturen und Betriebssysteme



Beispiel: Prozeduraufrufe (cont.)

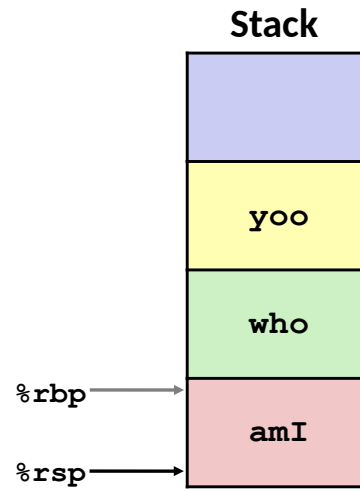
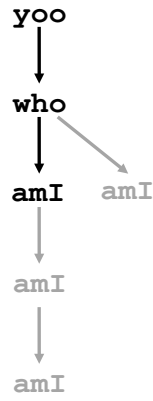
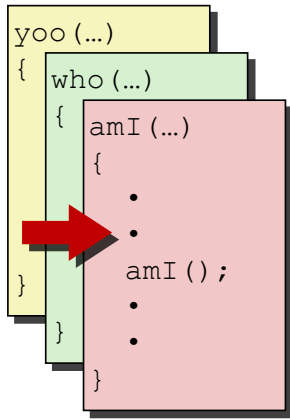
13.3.5 Assembler-Programmierung - x86 Assembler - Funktionsaufrufe und Stack

64-040 Rechnerstrukturen und Betriebssysteme

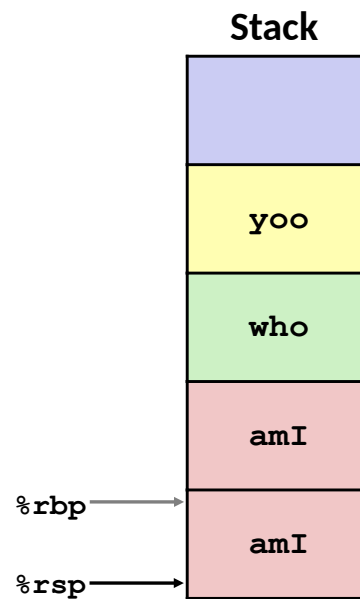
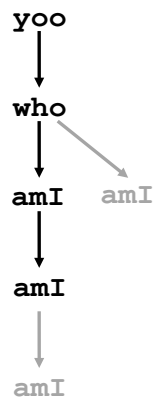
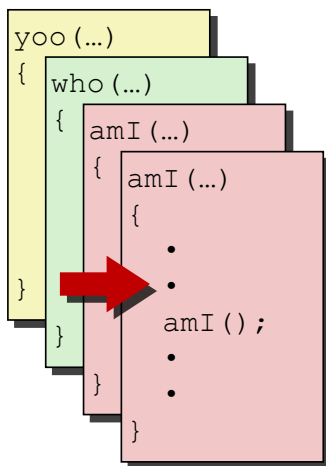




Beispiel: Prozeduraufrufe (cont.)

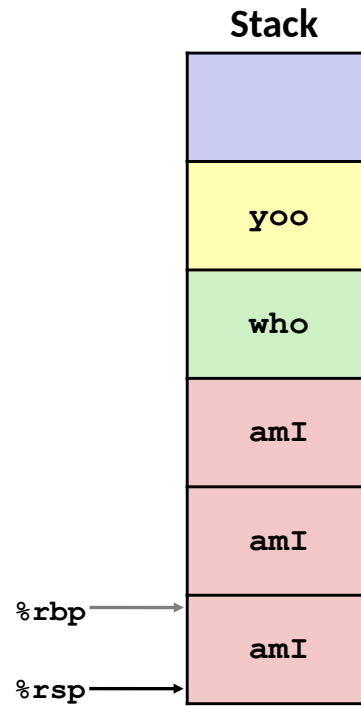
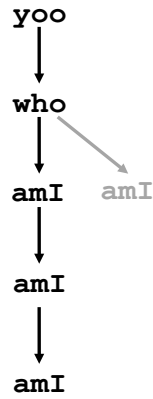
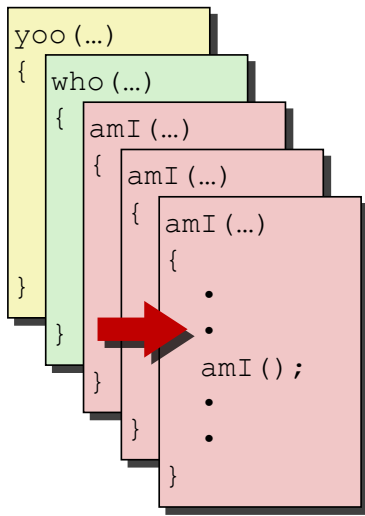


Beispiel: Prozeduraufrufe (cont.)

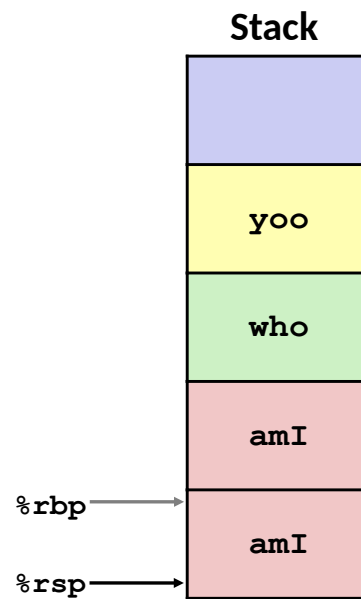
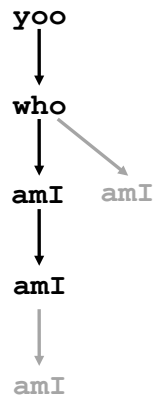
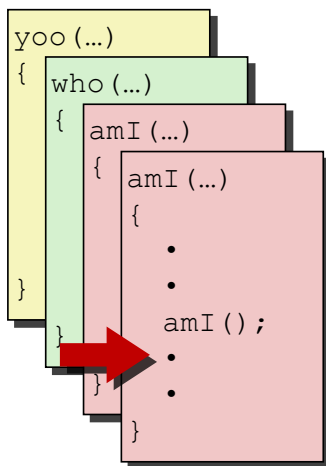




Beispiel: Prozeduraufrufe (cont.)



Beispiel: Prozeduraufrufe (cont.)

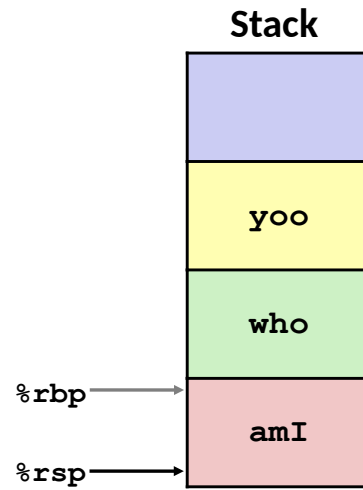
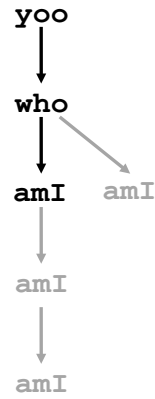
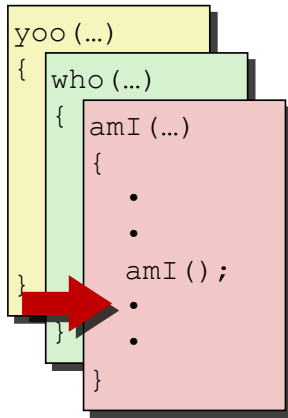




Beispiel: Prozeduraufrufe (cont.)

13.3.5 Assembler-Programmierung - x86 Assembler - Funktionsaufrufe und Stack

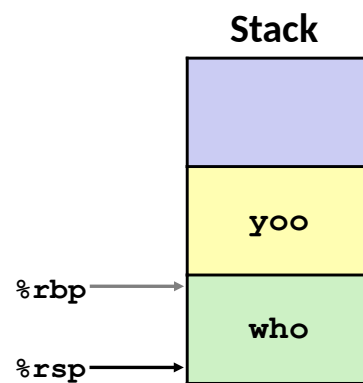
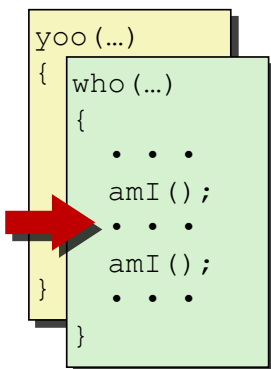
64-040 Rechnerstrukturen und Betriebssysteme



Beispiel: Prozeduraufrufe (cont.)

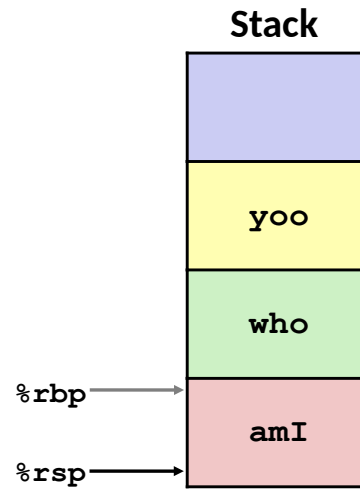
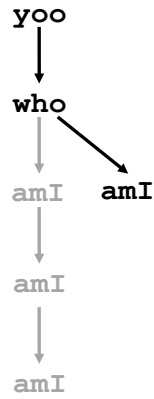
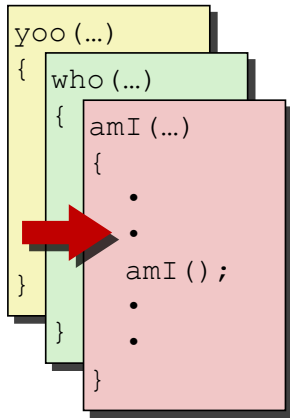
13.3.5 Assembler-Programmierung - x86 Assembler - Funktionsaufrufe und Stack

64-040 Rechnerstrukturen und Betriebssysteme

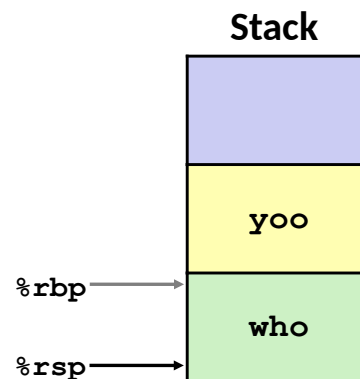
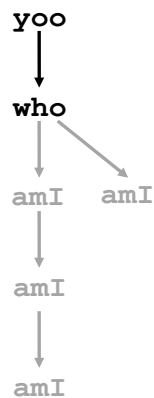
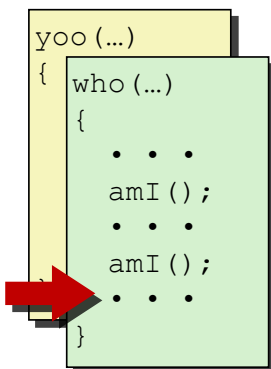




Beispiel: Prozeduraufrufe (cont.)



Beispiel: Prozeduraufrufe (cont.)



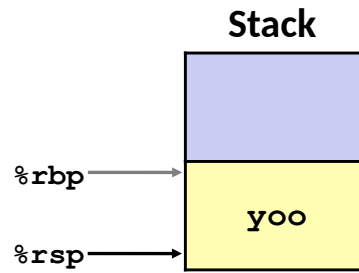
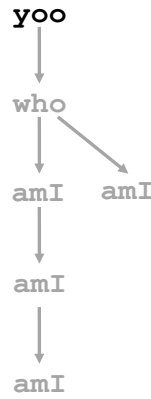


Beispiel: Prozeduraufufe (cont.)

```

yoo (...)
{
  .
  .
  who ();
  .
  .
}

```



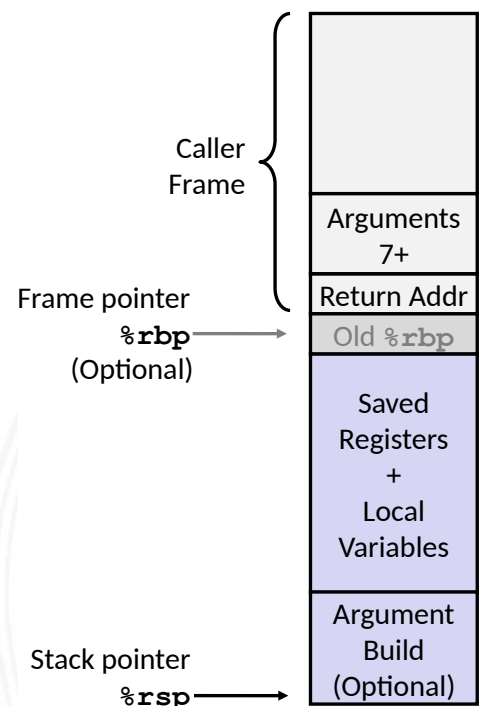
x86-64/Linux Stack-Frame

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





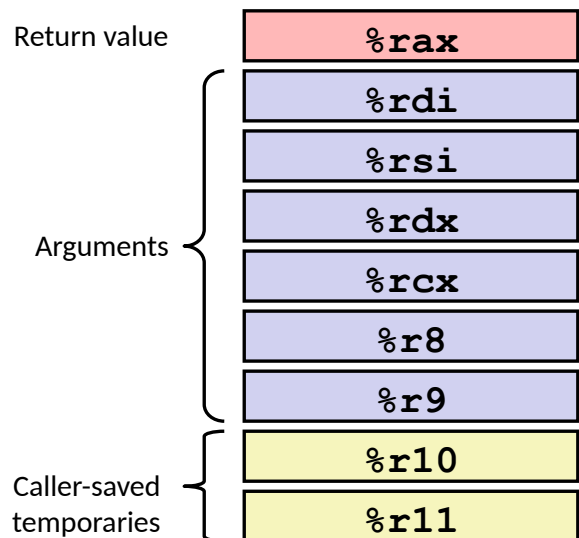
Register Sicherungskonventionen

- ▶ yoo („Caller“) ruft Prozedur who („Callee“) auf
- ⇒ Welche Register können temporär von who genutzt werden?
- ▶ zwei mögliche Konventionen
 - ▶ „Caller-Save“: yoo speichert in seinen Frame vor Prozeduraufruf
 - ▶ „Callee-Save“: who speichert in seinen Frame vor Benutzung



Register Sicherungskonventionen (cont.)

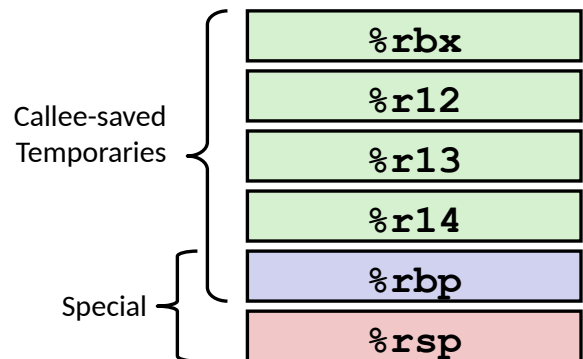
- ▶ **%rax**
 - ▶ Rückgabewert
 - ▶ Caller-Save
 - ▶ kann lokal geschrieben werden
- ▶ **%rdi...%r9**
 - ▶ Argumente
 - ▶ Caller-Save
 - ▶ können lokal geschrieben werden
- ▶ **%r10, %r11**
 - ▶ Caller-Save
 - ▶ können lokal geschrieben werden





Register Sicherungskonventionen (cont.)

- ▶ **%rbx, %r12...%r14**
 - ▶ Callee-Save
 - ▶ Prozedur muss sichern (Stack-Frame) und zurückschreiben
- ▶ **%rbp**
 - ▶ Callee-Save
 - ▶ Prozedur muss sichern (Stack-Frame) und zurückschreiben
 - ▶ Frame-Pointer $\hat{=}$ Beginn des eigenen Frames
- ▶ **%rsp**
 - ▶ Behandlung durch call/return
 - ▶ Sonderfall, quasi Callee-Save



Linker und Loader

- ▶ Programm in mehrere Quelldateien aufgeteilt

```
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

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

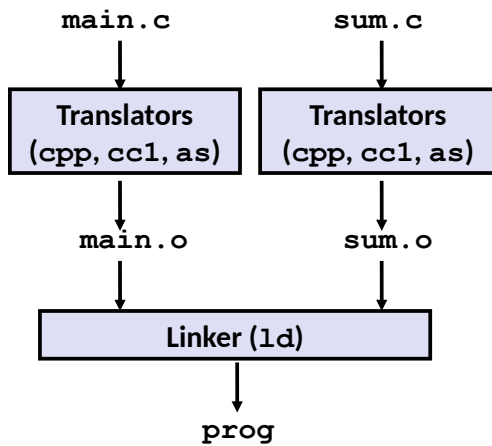
sum.c



Linker und Loader (cont.)

- ▶ Compiler(-driver) startet einzelne Programme
 - ▶ Präprozessor (cpp), Compiler (cc), Assembler (as) und Linker (ld)
 - ▶ „Feintuning“ und Steuerung über Kommandozeilen-Parameter
'zig Optionen für jedes Teilprogramm (s. man gcc)

Linux gcc

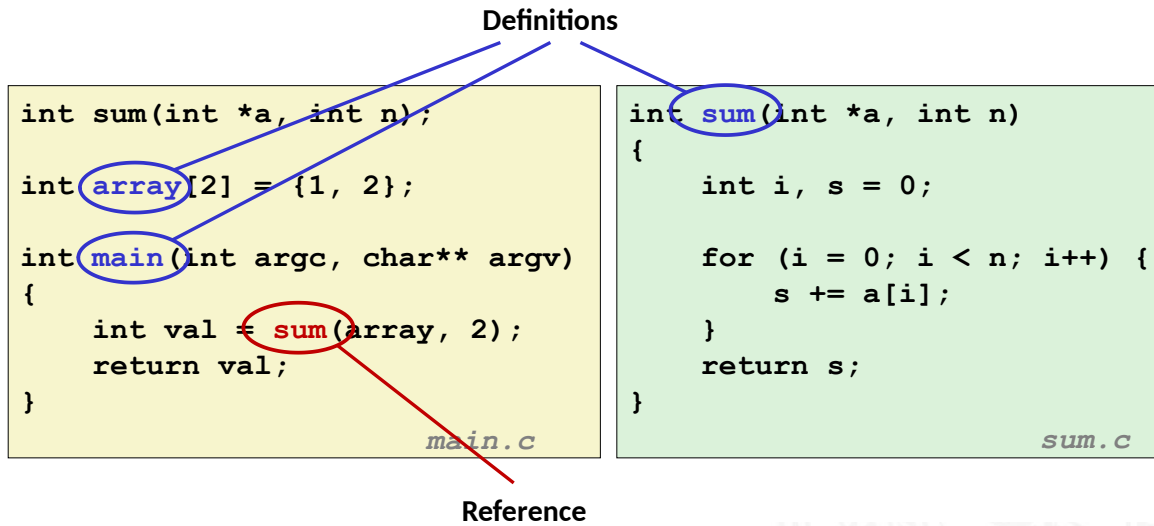


Linker und Loader (cont.)

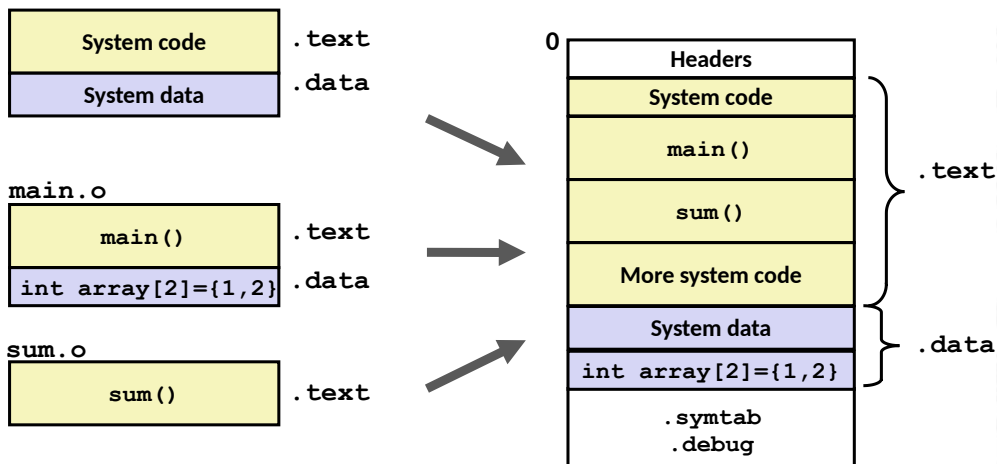
- + Modularität
 - ▶ Programm in übersichtlichen kleinen Dateien
 - ▶ Funktionen können wiederverwendet werden
 - ⇒ vorgefertigte Programmbibliotheken
- + Effizienz
 - ⇒ Zeitvorteil
 - ▶ nach Änderung müssen nur kleine Teile neu übersetzt werden
 - ▶ ermöglicht paralleles Compilieren
 - ⇒ (Speicher-) Platzvorteil
 - ▶ wichtige Funktionen in Programmbibliotheken aggregiert
 - ▶ ermöglicht gemeinsame Nutzung ⇒ *Shared Libraries*
 - ▶ effizienteres, vereinfachtes Programmieren



1. Symbole identifizieren (globale Variablen, Funktionen)
Symbole auflösen (eindeutig machen) ⇒ Symboltabelle



2. „Relocation“
 - ▶ Programmcode und -daten der Quelldateien zusammenführen
 - ▶ alle Symboltabellen zusammenfassen
 ⇒ gemeinsamer, eindeutiger Adressraum: Sprünge+Symbole



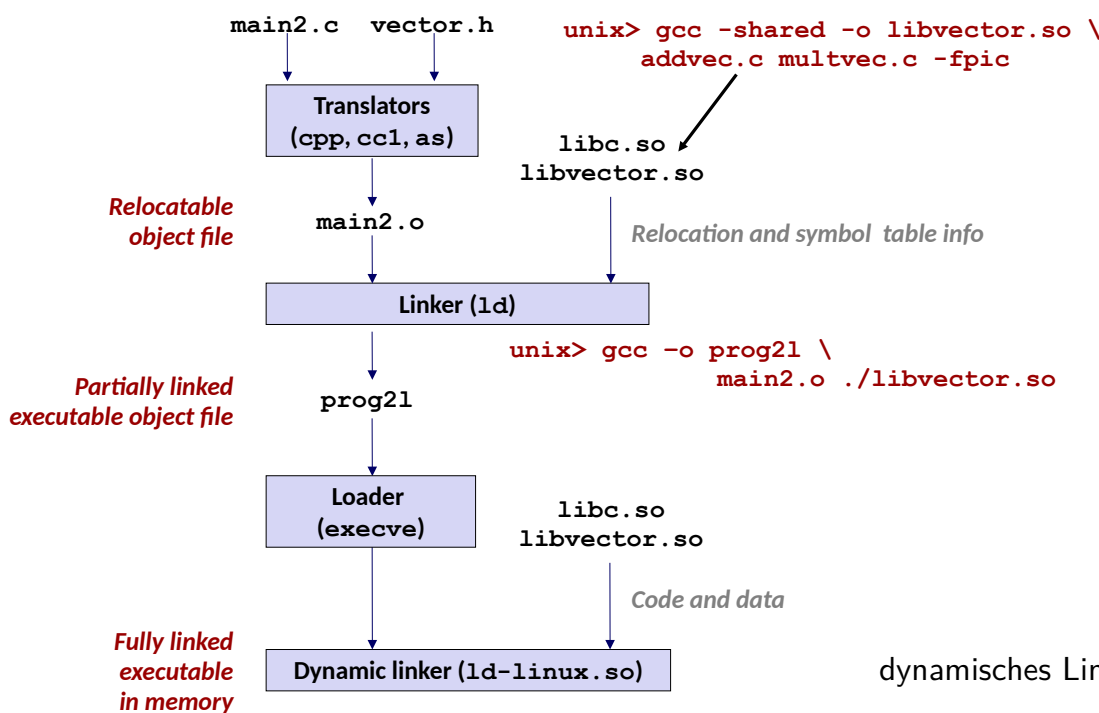
⇒ erzeugt ein ausführbares Programm für *Loader*



- ▶ **Statisches Binden** („static linking“)
 - ▶ Funktionen aus Bibliotheksarchiven (.a-Dateien) werden in das Programm integriert
 - ▶ nicht genutzte Funktionen werden entfernt
 - ▶ Linken während Compilierung

- ▶ **Dynamisches Binden** („dynamic linking“)
 - ▶ Bibliotheken werden erst beim Laden in Speicher oder 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“



dynamisches Linken beim Laden



viele Themen aus Zeitgründen nicht behandelt

- ▶ Linker und Loader
 - ▶ genauere Funktionsweise von Linker und Loader
 - ▶ programmiertechnische Realisierung
 - ▶ Probleme bei der Symbolauflösung

- ▶ Speicherverwaltung
 - ▶ Abbildung der Datenstrukturen auf Bytes im Speicher
 - ▶ Adressberechnung für Arrays, Records
 - ▶ Alignment
 - ▶ dynamische Speicherverwaltung / der „Heap“

- ▶ Objektorientierte Konzepte
 - ▶ Daten mit zugehörigen Methoden kombinieren



- ▶ *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

13.5 Assembler-Programmierung - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [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.)

13.5 Assembler-Programmierung - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [PH22] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle – MIPS Edition.*
6. Auflage, De Gruyter Oldenbourg, 2022. ISBN 978-3-11-075598-5
- [PH20] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – RISC-V Edition.*
2nd edition, Morgan Kaufmann Publishers Inc., 2020. ISBN 978-0-12-820331-6
- [Hyd10] R. Hyde: *The Art of Assembly Language.*
2nd edition, No Starch Press, 2010. ISBN 978-1-5932-7207-4
www.randallhyde.com/AssemblyLanguage/www.artofasm.com
- [Hyd21] R. Hyde: *The Art of 64-bit Assembly, Volume 1.*
No Starch Press, 2021. ISBN 978-1-7185-0108-9
artofasm.randallhyde.com



1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. Logische Operationen
7. Codierung
8. Schaltfunktionen
9. Schaltnetze
10. Schaltwerke
11. Rechnerarchitektur I
12. Instruction Set Architecture



13. Assembler-Programmierung

14. Rechnerarchitektur II

Pipelining

- Befehlspipeline

- MIPS

- Bewertung

Parallelität

- Amdahl's Gesetz

- Superskalare Rechner

- Parallelrechner

- Symmetric Multiprocessing

Speicherhierarchie

- Speichertypen

- Cache Speicher

Literatur

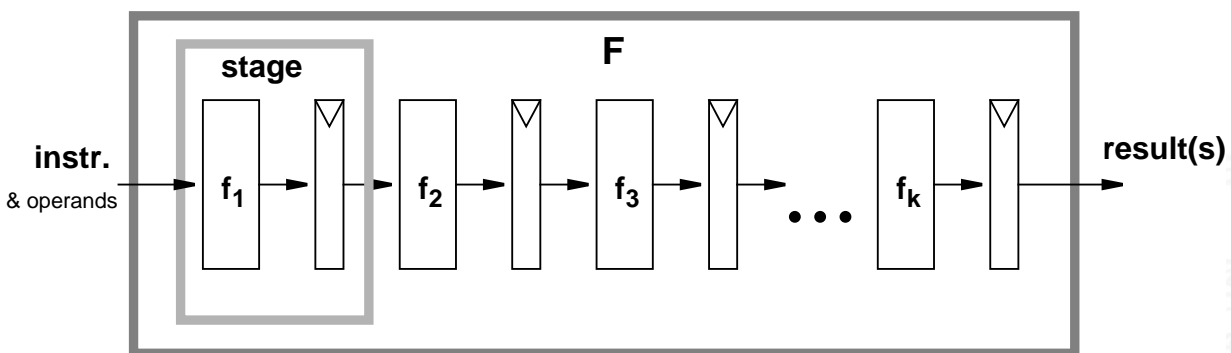




15. Betriebssysteme



Pipelining / Fließbandverarbeitung



Grundidee

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



Pipelining-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



Arithmetische Pipelines

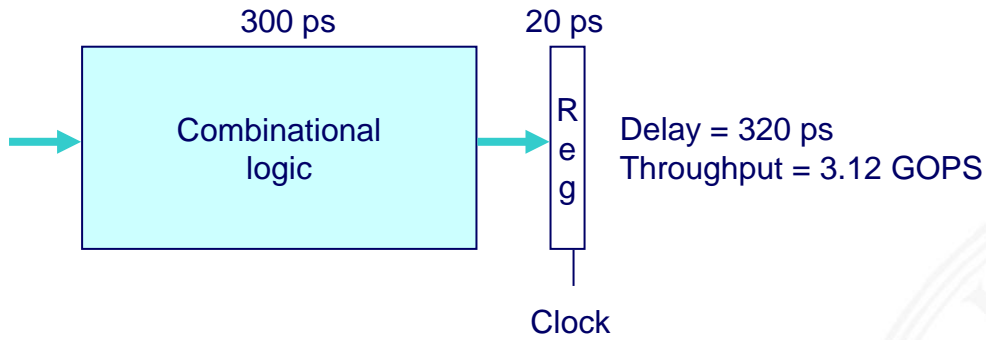
- ▶ Idee: lange Berechnung in Teilschritte zerlegen
- ▶ wichtig bei komplizierteren arithmetischen Operationen
 - ▶ die sonst sehr lange dauern (weil ein großes Schaltnetz)
 - ▶ die als Schaltnetz extrem viel Hardwareaufwand erfordern
 - ▶ Beispiele: Multiplikation, Division, Fließkommaoperationen ...
- + Erhöhung des Durchsatzes, wenn die Berechnung mehrfach hintereinander ausgeführt wird / die Pipeline ausgelastet ist

Befehlspipeline im Prozessor

- ▶ Idee: die Phasen der von-Neumann Befehlsabarbeitung (Befehl holen, Befehl decodieren ...) als Pipeline implementieren
- folgt in *Befehlspipeline*, ab Folie 1000



Beispiel: Schaltnetz ohne Pipeline

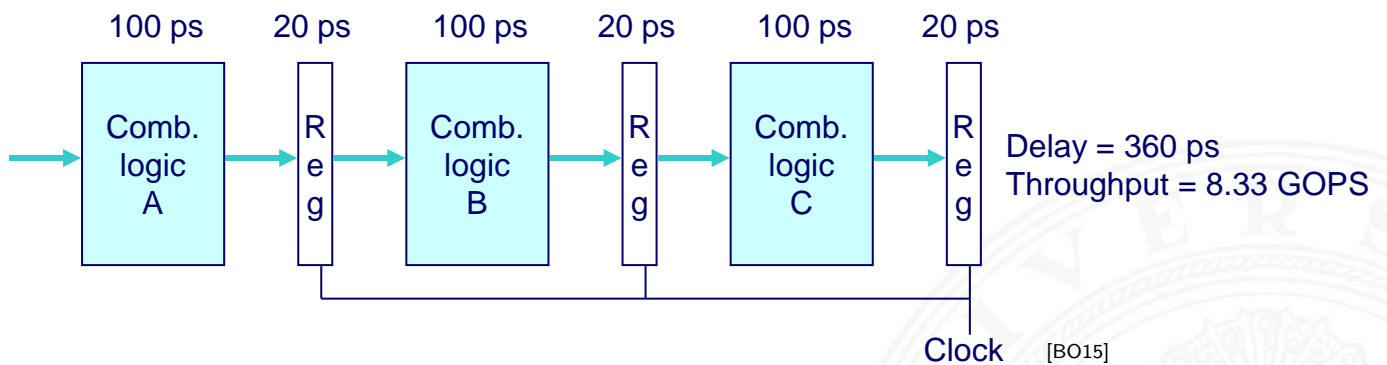


[B015]

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



Beispiel: Version mit 3-stufiger Pipeline



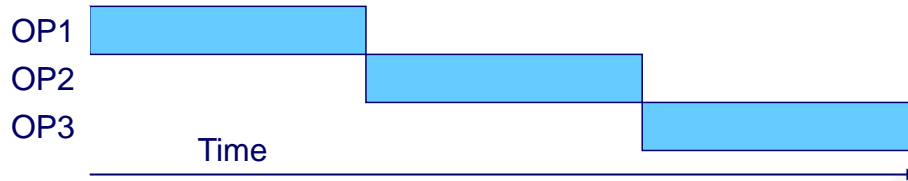
[B015]

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



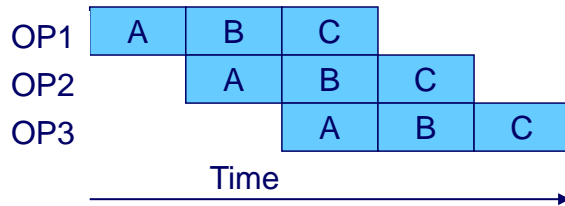
Prinzip: 3-stufige Pipeline

▶ ohne Pipeline

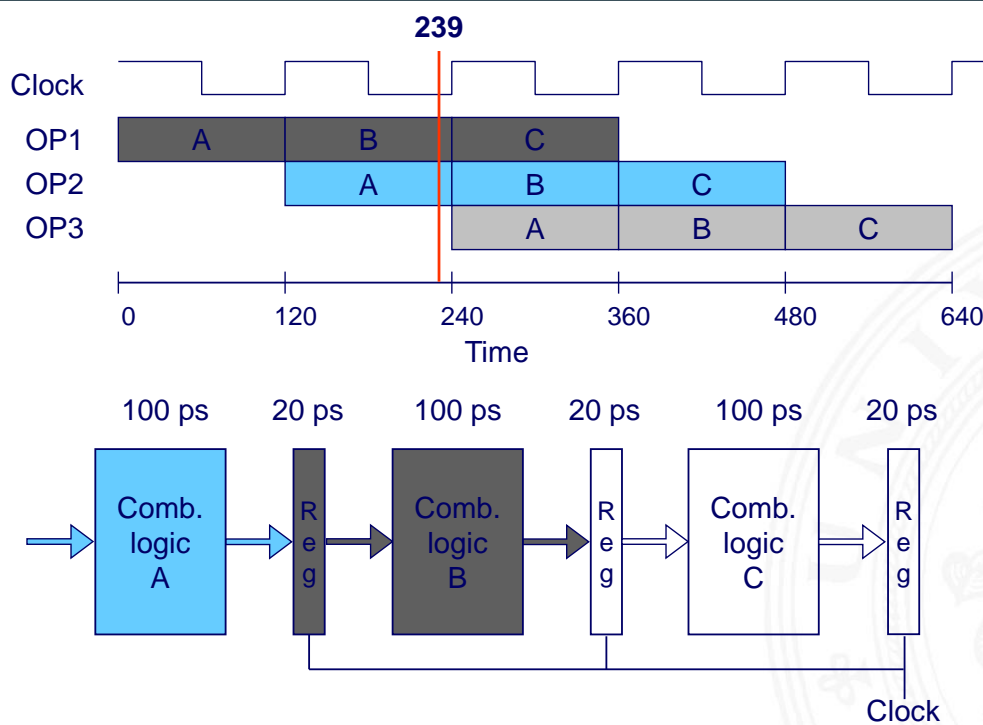


[BO15]

▶ 3-stufige Pipeline



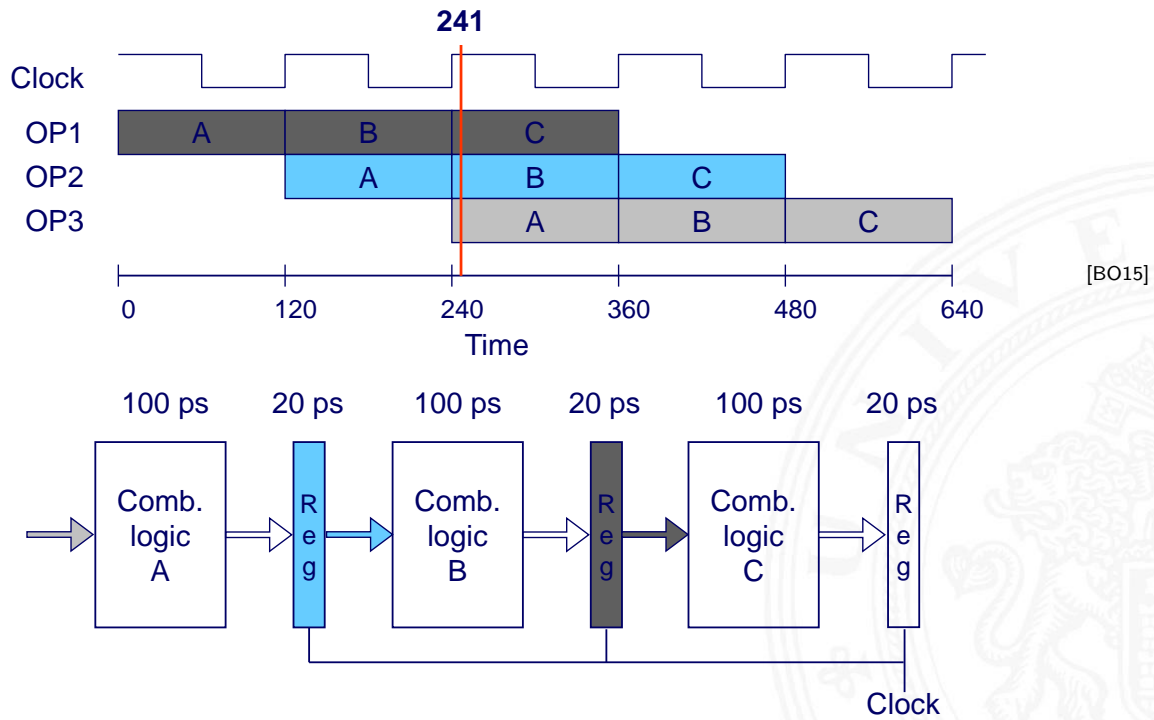
Timing: 3-stufige Pipeline



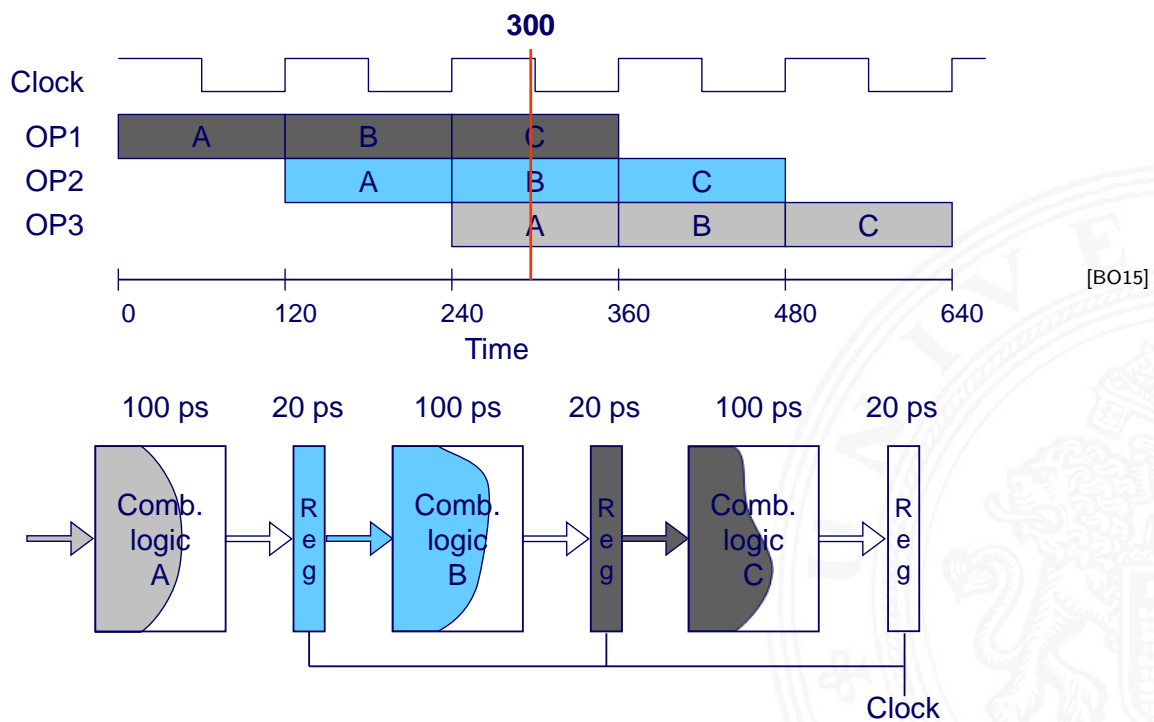
[BO15]



Timing: 3-stufige Pipeline

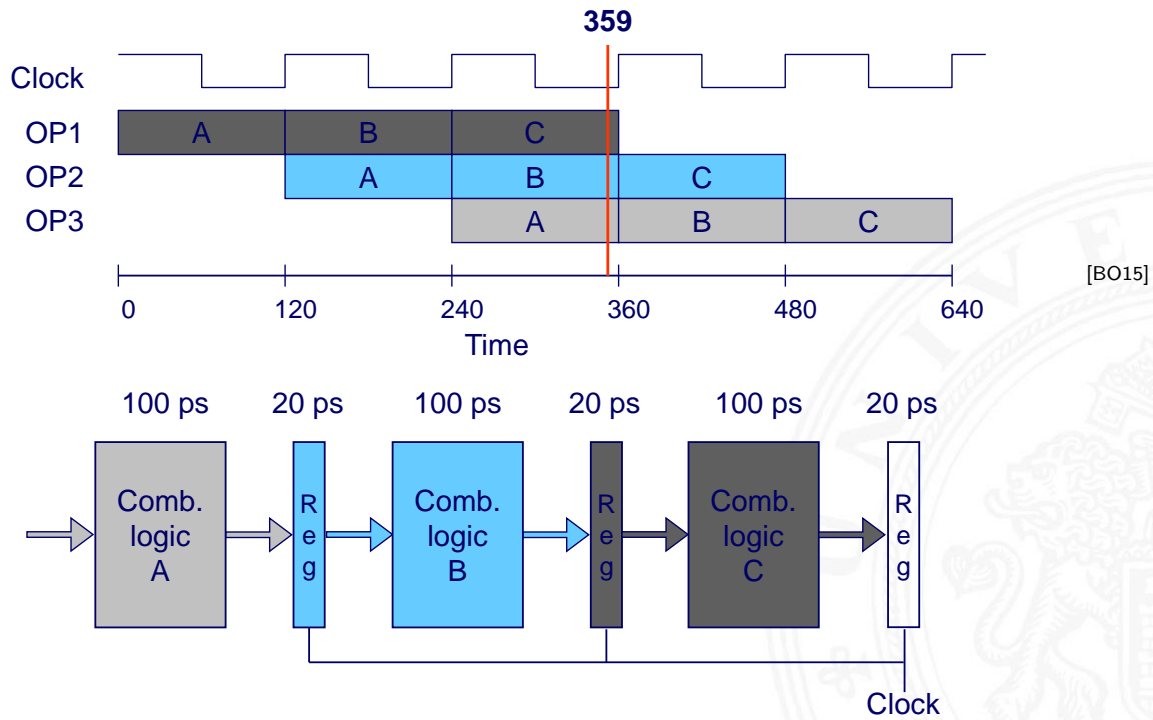


Timing: 3-stufige Pipeline

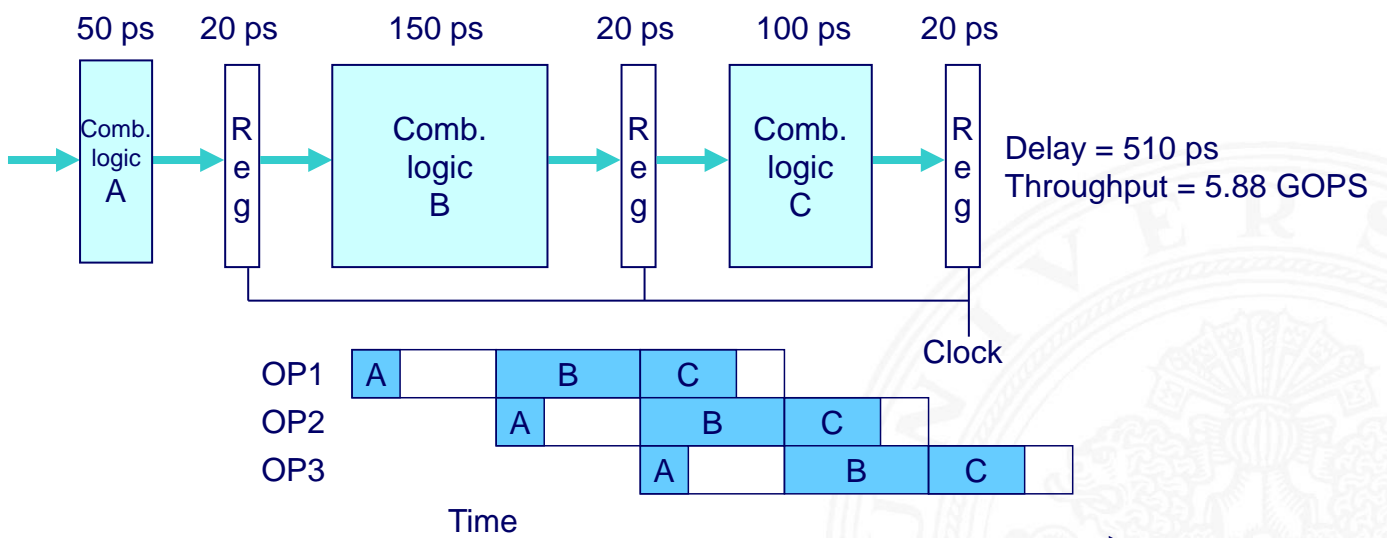




Timing: 3-stufige Pipeline



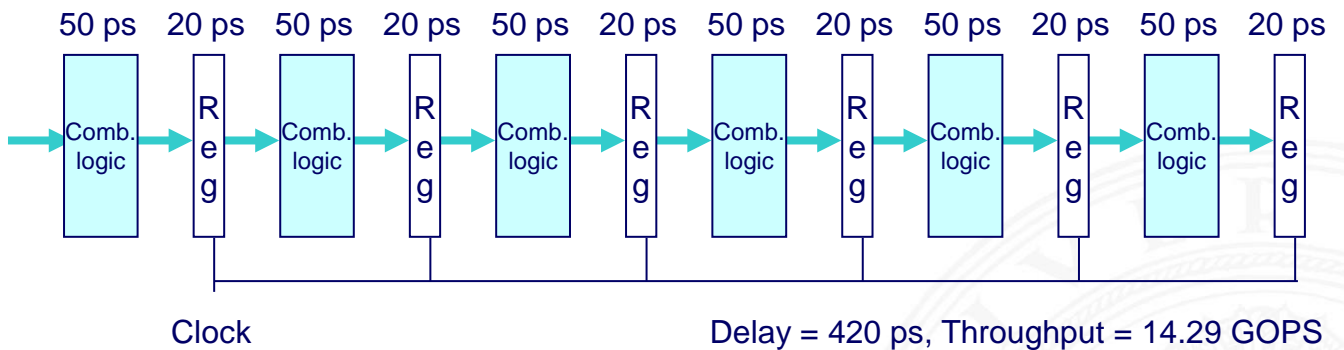
Limitierungen: nicht uniforme Verzögerungen



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



Limitierungen: Register „Overhead“



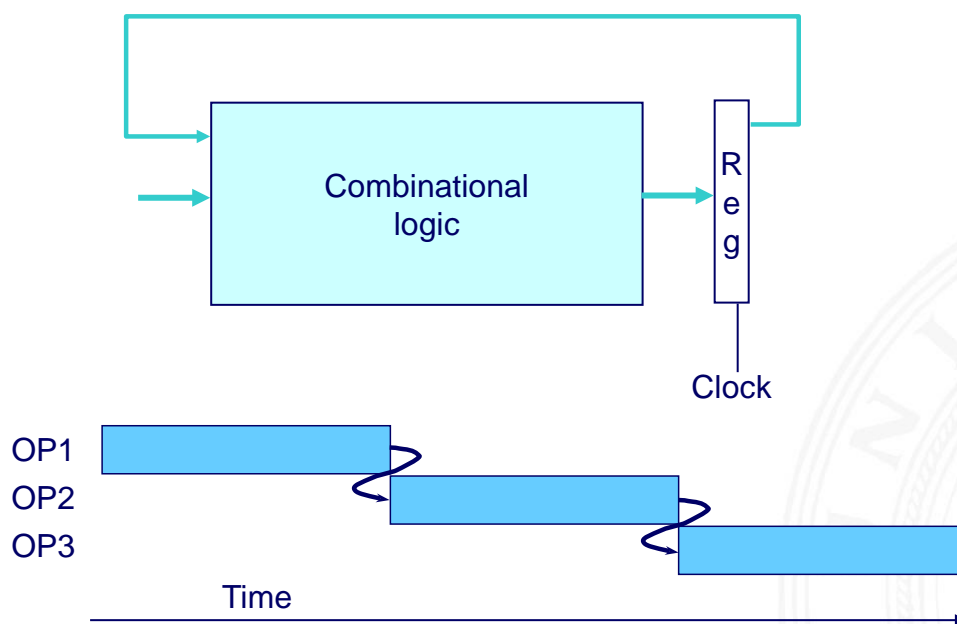
[BO15]

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

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



Limitierungen: Datenabhängigkeiten

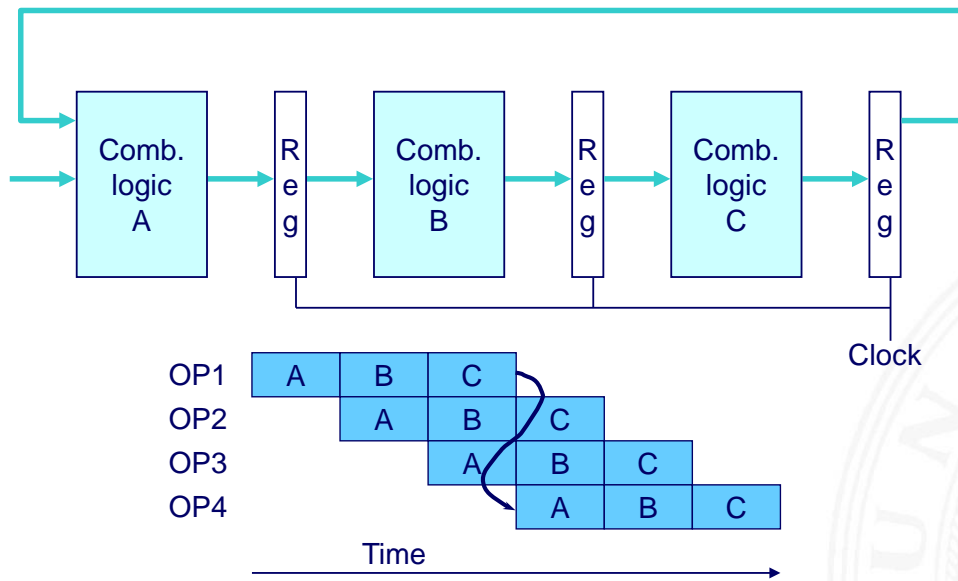


[BO15]

- ▶ 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** **E**xecute
ALU führt Befehl aus

- MEM** **M**emory access
Speicherzugriff: Daten laden/abspeichern

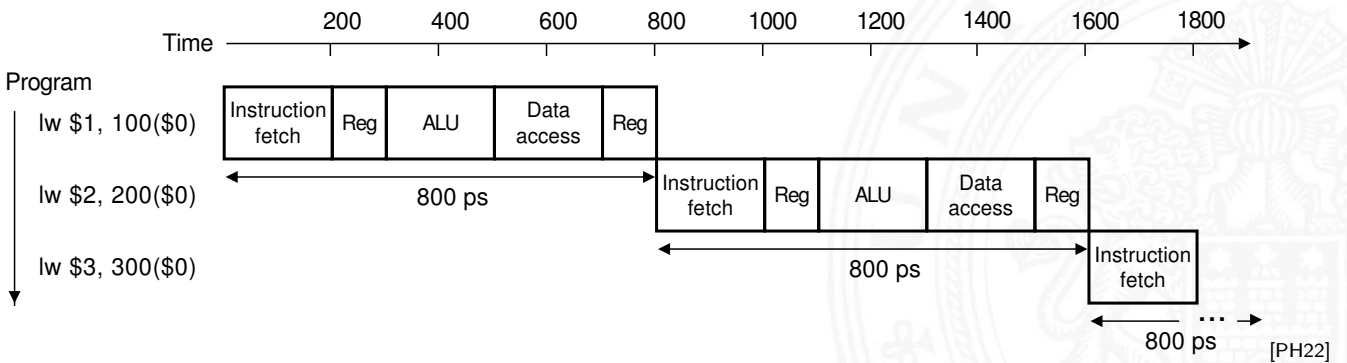
- WB** **W**rite **B**ack
Ergebnis in Register zurückschreiben



von-Neumann Befehlszyklus (cont.)

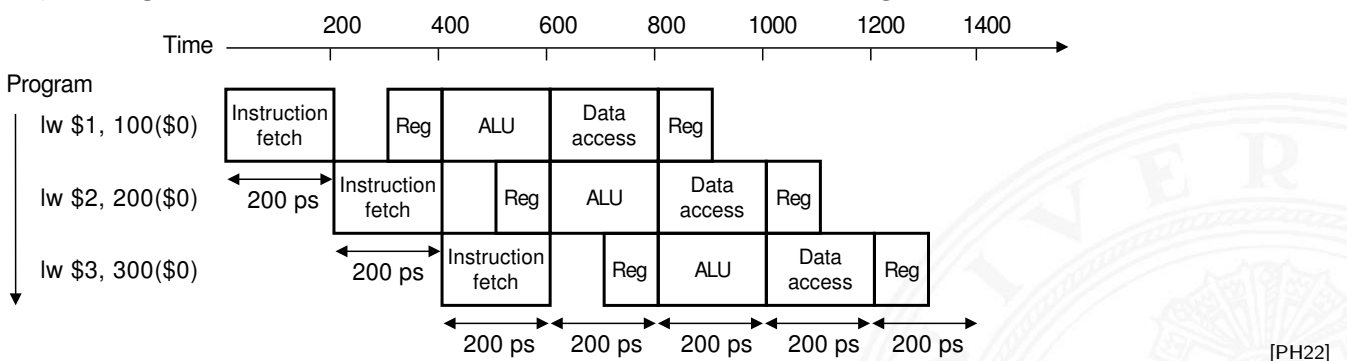
- ▶ je nach Instruktion sind nicht alle Schritte notwendig
 - ▶ *nop*: nur Instruction-Fetch
 - ▶ *jump*: kein Speicher- und Registerzugriff
 - ▶ *aluOp*: kein Speicherzugriff
- ▶ Pipeline kann auch feiner unterteilt werden (meist mehr Stufen)

serielle Bearbeitung ohne Pipelining



von-Neumann Befehlszyklus (cont.)

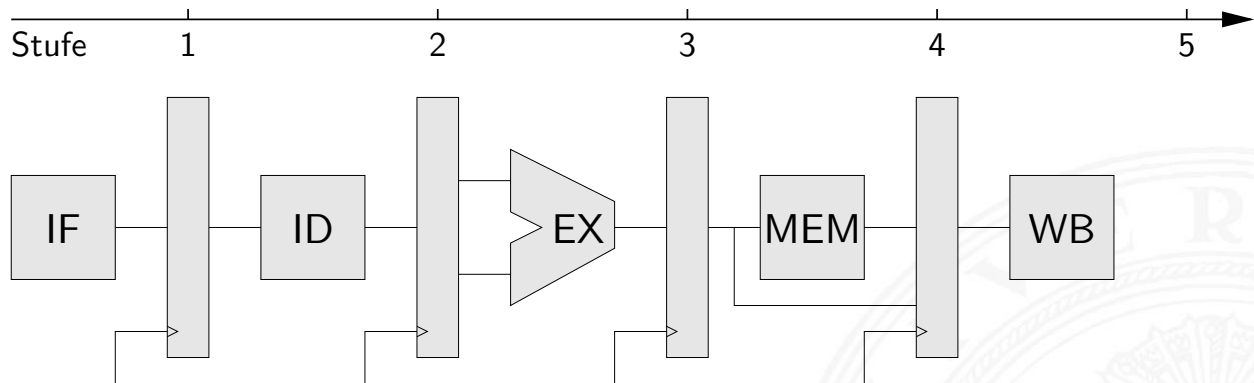
Pipelining für die einzelnen Schritte der Befehlsausführung



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



Klassische 5-stufige Pipeline



- ▶ Grundidee der ursprünglichen RISC-Architekturen
- + Durchsatz ca. $3 \dots 5 \times$ besser als serielle Ausführung
- + guter Kompromiss aus Leistung und Hardwareaufwand
- ▶ MIPS-Architektur (aus Patterson, Hennessy [PH22])

▶ MIPS ohne Pipeline

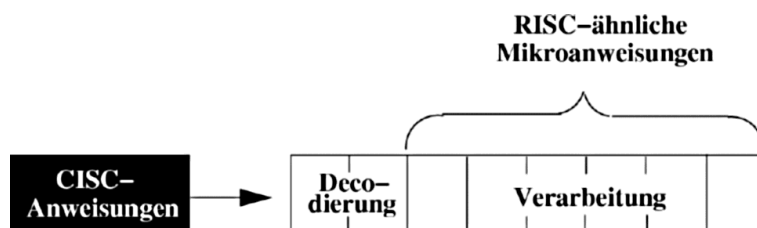
▶ MIPS Pipeline

▶ Pipeline Schema



Klassische 5-stufige Pipeline (cont.)

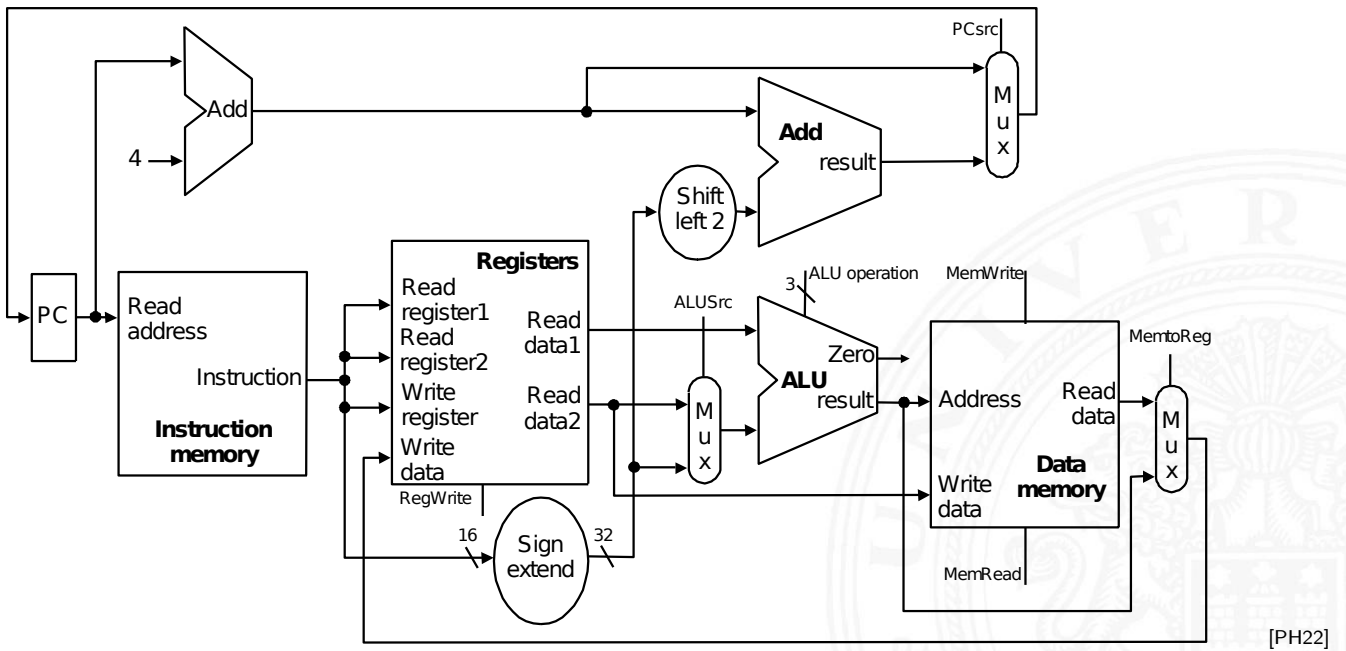
- ▶ RISC ISA: Pipelining wird direkt umgesetzt
 - ▶ Befehlssätze auf diese Pipeline hin optimiert
 - ▶ IBM-801, MIPS R-2000/R-3000 (1985), SPARC (1987)
- ▶ CISC-Architekturen heute ebenfalls mit Pipeline
 - ▶ Motorola 68020 (zweistufige Pipeline, 1984), Intel 486 (1989), Pentium (1993) ...
 - ▶ Befehle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



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



MIPS: serielle Realisierung ohne Pipeline



[PH22]

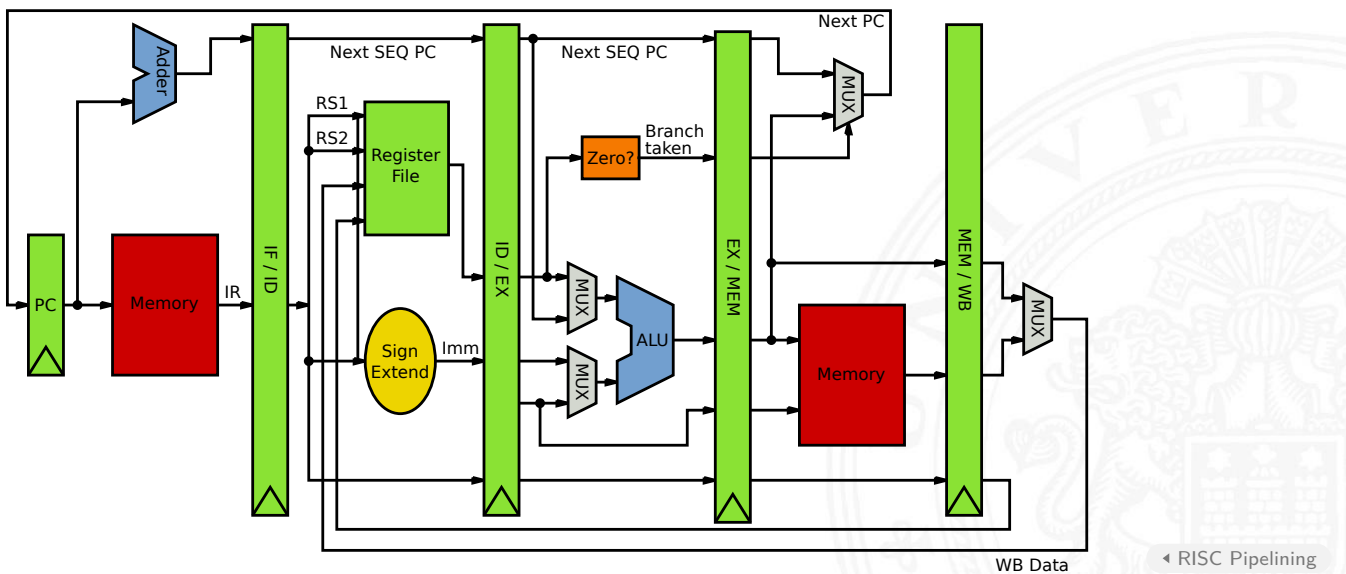
längster Pfad: PC - IM - REG - MUX - ALU - DM - MUX - PC/REG

◀ RISC Pipelining



MIPS: mit 5-stufiger Pipeline

Instruction Fetch IF	Instruction Decode Register Fetch ID	Execute Address Calc. EX	Memory Access MEM	Write Back WB
-------------------------	--	--------------------------------	----------------------	------------------



◀ RISC Pipelining



MIPS: mit 5-stufiger Pipeline (cont.)

14.1.2 Rechnerarchitektur II - Pipelining - MIPS

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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

14.1.3 Rechnerarchitektur II - Pipelining - Bewertung

64-040 Rechnerstrukturen und Betriebssysteme

Begriffe

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



Vor- und Nachteile

- + Schaltnetze sind in kleinere Blöcke aufgeteilt
⇒ höhere Taktfrequenzen
- + im Idealfall wird ein neuer Befehl pro Takt gestartet
⇒ höherer Durchsatz, bessere Performanz
- + geringer Zusatzaufwand an Hardware
- + Pipelining ist für den Programmierer nicht direkt sichtbar!
 - Achtung: Daten-/Kontrollabhängigkeiten (s.u.)
- Latenz wird nicht verbessert, bleibt bestenfalls gleich
- Pipeline Takt limitiert durch langsamste Pipelinestufe
unausgewogene Pipelinestufen reduzieren den Takt und damit den Durchsatz
- zusätzliche Zeiten, um Pipeline zu füllen bzw. zu leeren



Analyse

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

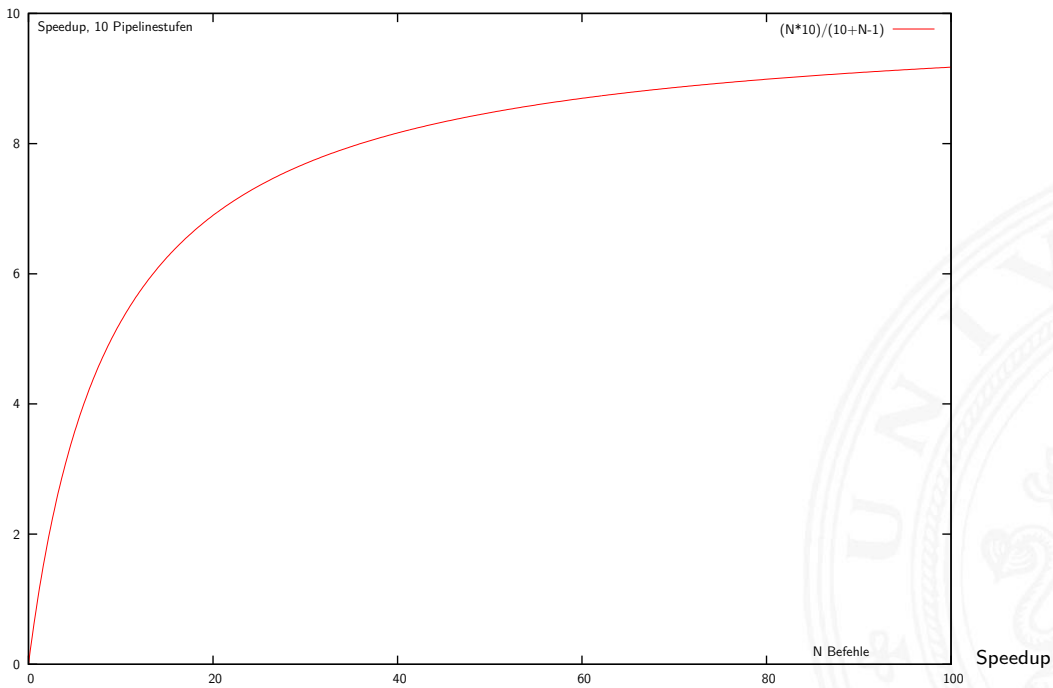
- ▶ „Speedup“ $S = \frac{N \cdot K}{K + N - 1}$, $\lim_{N \rightarrow \infty} S = K$

- ⇒ 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	Pipelinstufen	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	≤ 3 000
	Pentium 4	20...31	≤ 3 800
	Core i-..	14/19	≤ 4 200
	Ryzen ..	19	≤ 4 000



Architekturentscheidungen, die sich auf das Pipelining auswirken

gut für Pipelining

- ▶ gleiche Instruktionslänge
- ▶ wenige Instruktionsformate
- ▶ Load/Store Architektur

BASIC INSTRUCTION FORMATS

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

FLOATING-POINT INSTRUCTION FORMATS

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

MIPS-Befehlsformate [PH22]



schlecht für Pipelining: *Pipelinekonflikte / -Hazards*

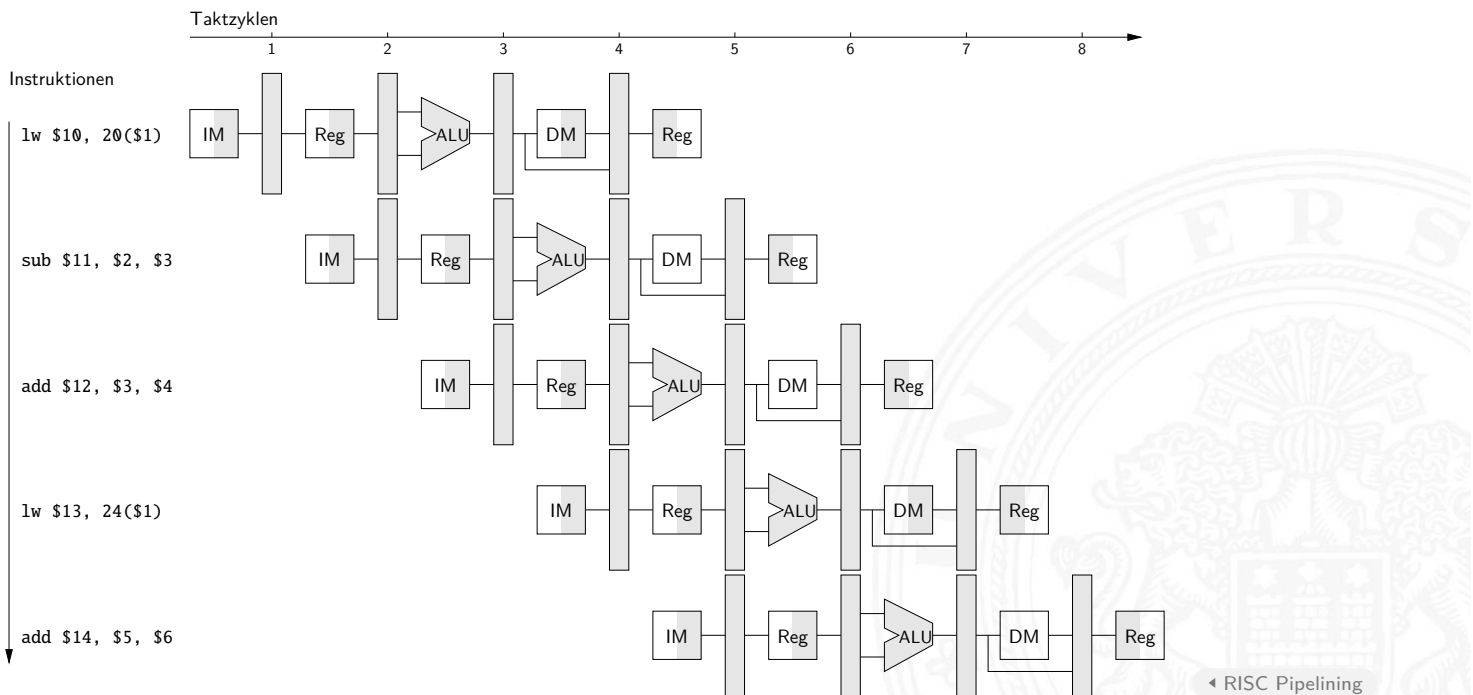
- ▶ **Strukturkonflikt:** gleichzeitiger Zugriff auf eine Ressource durch mehrere Pipelinestufen
- ▶ **Datenkonflikt:** Ergebnisse von Instruktionen werden vor dem Write-Back bereits in der Pipeline benötigt
- ▶ **Steuerkonflikt:** (bedingte) Sprungbefehle verändern die Befehlssequenz innerhalb der Pipeline

sehr schlecht für Pipelining

- ▶ Unterbrechung des Programmkontexts („Context Switch“): Interrupt, System-Call, Exception, Prozesswechsel ...



Pipeline Schema



Motivation: ständig steigende Anforderungen

- ▶ Simulationen, Wettervorhersage, Gentechnologie ...
- ▶ Datenbanken, Transaktionssysteme, Suchmaschinen ...
- ▶ Softwareentwicklung, Schaltungsentwurf ...
- ▶ Performanz eines einzelnen Prozessors ist begrenzt
- ⇒ Hardware: Verteilen eines Programms auf mehrere Prozessoren
- ⇒ Software: kommunizierende Prozesse und Multithreading

Vielfältige Möglichkeiten

- ▶ wie viele und welche Prozessoren?
- ▶ Kommunikation zwischen den Prozessoren?
- ▶ Programmierung und Software/Tools?



- ▶ **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:** $s = \frac{\text{performance } x}{\text{performance } y} = \frac{\text{execution time } y}{\text{execution time } x}$



- ▶ Ausführungszeit = $\langle \text{Anzahl der Befehle} \rangle \cdot \langle \text{Zeit pro Befehl} \rangle$

- ▶ 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** Programm P wird ausgeführt, darin Funktion X mit Anteil $0 < f < 1$ der Gesamtzeit
- 2. System** Programm P wird ausgeführt, Funktion X' ist schneller als X mit Speedup S_X

Amdahl's Gesetz

Gene Amdahl, Architekt der IBM S/360, 1967

► Speedup

$$S_{gesamt} = \frac{1}{(1 - f) + f/S_X}$$

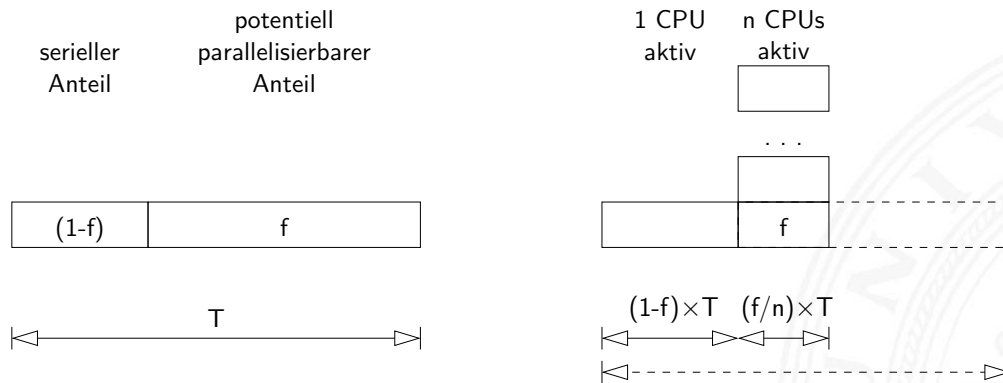


Amdahl's Gesetz (cont.)

Speedup

$$S_{gesamt} = \frac{1}{(1 - f) + f/S_X}$$

► nur ein Teil f des Gesamtproblems wird beschleunigt



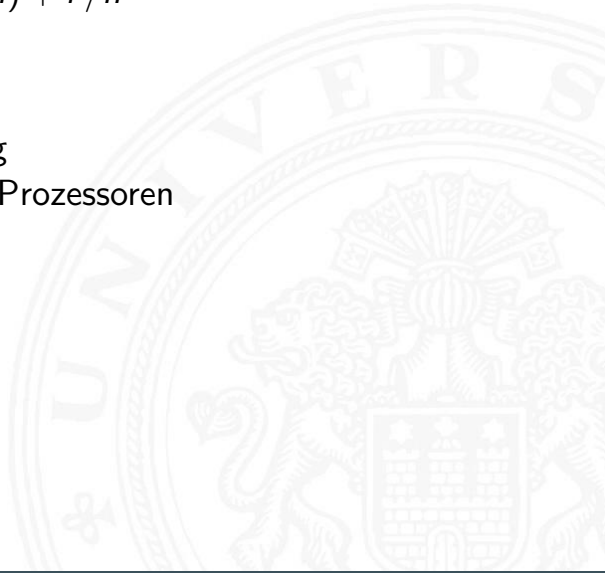
- ⇒ möglichst großer Anteil f
- ⇒ Optimierung lohnt nur für relevante Operationen
allgemeingültig: entsprechend auch für Projektplanung, Verkehr ...



- ▶ ursprüngliche Idee: Parallelrechner mit n -Prozessoren

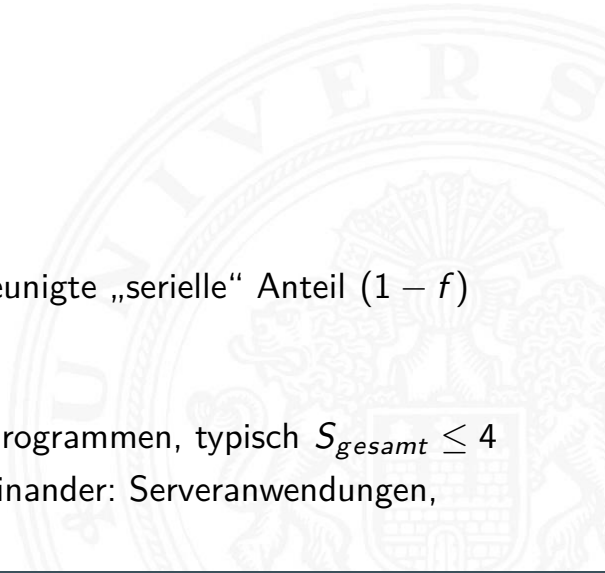
Speedup $S_{gesamt} = \frac{1}{(1 - f) + k(n) + f/n}$

- 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



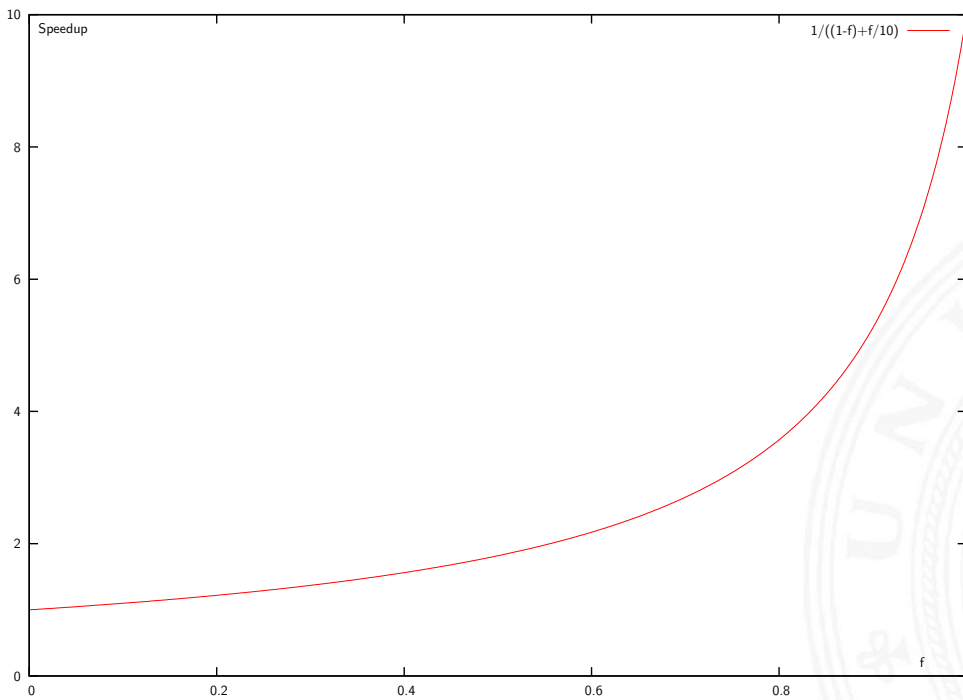
S_X	f	S_{gesamt}
10	0,1	$1/(0,9 + 0,01) = 1,1$
2	0,5	$1/(0,5 + 0,25) = 1,33$
2	0,9	$1/(0,1 + 0,45) = 1,82$
1,1	0,98	$1/(0,02 + 0,89) = 1,1$
4	0,5	$1/(0,5 + 0,125) = 1,6$
4 536	0,8	$1/(0,2 + 0,0 \dots) = 5,0$
9 072	0,99	$1/(0,01 + 0,0 \dots) = 98,92$

- ▶ Optimierung bringt nichts, wenn der nicht beschleunigte „serielle“ Anteil $(1 - f)$ eines Programms überwiegt
- n -Prozessoren (große S_X) wirken *nicht linear*
- geringe erreichbare Parallelität in Hochsprachen-Programmen, typisch $S_{gesamt} \leq 4$
- + oft viele Prozesse/Tasks, jeweils unabhängig voneinander: Serveranwendungen, virtuelle Maschinen, Container ...

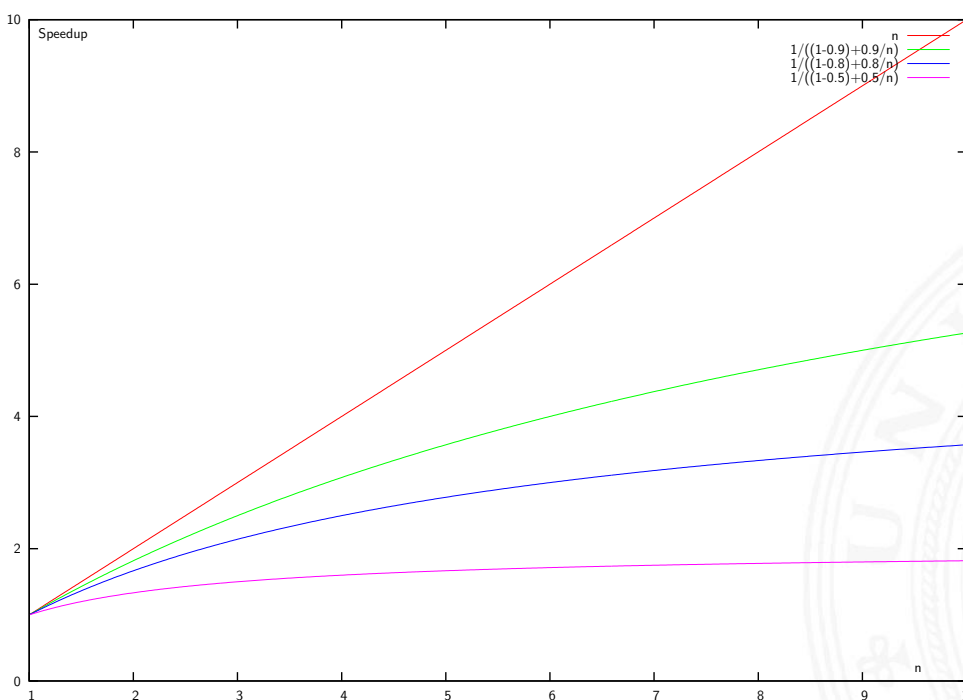




Amdahl's Gesetz: Beispiele (cont.)

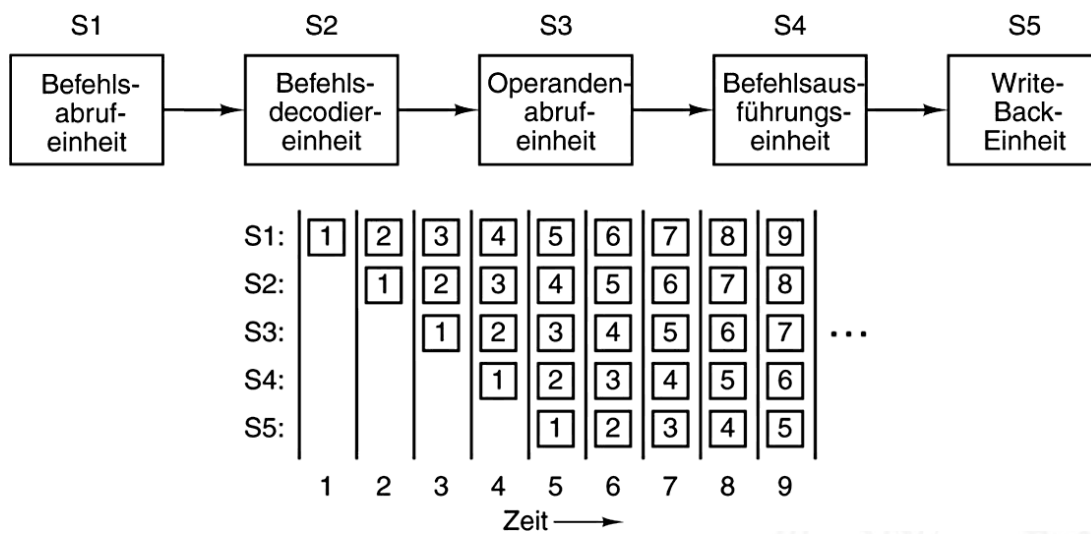


Amdahl's Gesetz: Beispiele (cont.)





Befehlspipeline

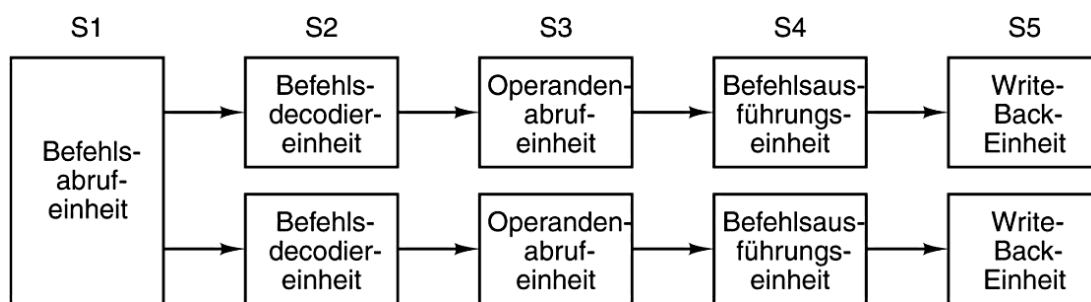


[TA14]

- ▶ Befehl in kleinere, schnellere Schritte aufteilen ⇒ höherer Takt
- ▶ mehrere Instruktionen überlappt ausführen ⇒ höherer Durchsatz



Parallele Pipelines

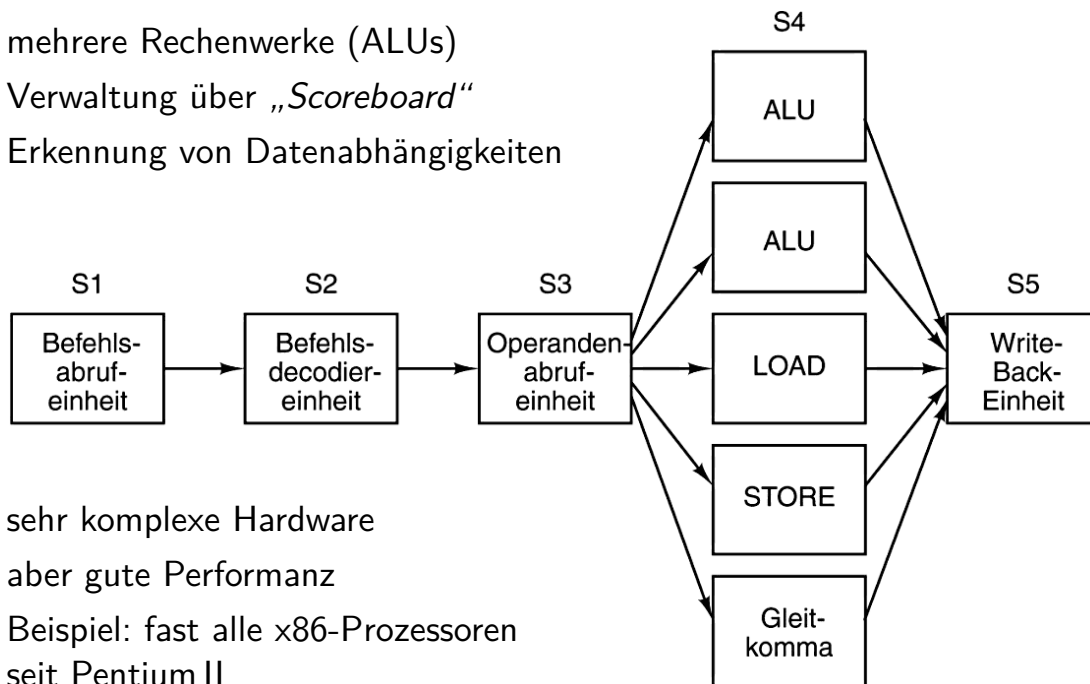


[TA14]

- ▶ im Bild jeweils zwei Operationen pro Pipelinestufe
- ▶ parallele („superskalare“) Ausführung
- ▶ komplexe Hardware (Daten- und Kontrollabhängigkeiten)
- ▶ Beispiel: Pentium



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



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

[TA14]



- ▶ Superskalare CPUs besitzen mehrere Recheneinheiten: 4 ... 12
 - ▶ in jedem Takt werden (dynamisch) mehrere Instruktionen eines konventionell linearen Instruktionsstroms abgearbeitet
- ⇒ ILP (Instruction **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 von Hazards wird verschärft



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



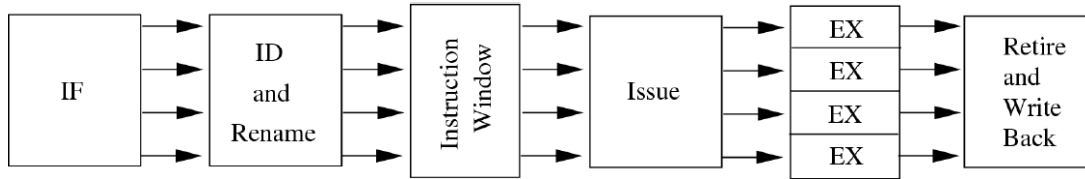
- ▶ Hardware löst (einige) Datenabhängigkeiten der Pipeline auf
- ▶ zwei Arten von Registersätzen
 1. Architektur-Register: „logische Register“ der ISA
 2. viele Hardware-Register: „Rename Register“ (180 Int, 168 FP)
 - ▶ dynamische Abbildung von ISA- auf Hardware-Register
- Kontextwechsel aufwändig: „Rename Register“ speichern
- ▶ Beispiel

Originalcode	nach Renaming
<code>tmp = a + b;</code>	<code>tmp1 = a + b;</code>
<code>res1 = c + tmp;</code>	<code>res1 = c + tmp1;</code>
<code>tmp = d + e;</code>	<code>tmp2 = d + e;</code>
<code>res2 = tmp - f;</code>	<code>res2 = tmp2 - f;</code>
	<code>tmp = tmp2;</code>

- ▶ Parallelisierung des modifizierten Codes
- | | | |
|-------------------------------|-------------------------------|--------------------------|
| <code>tmp1 = a + b;</code> | <code>tmp2 = d + e;</code> | |
| <code>res1 = c + tmp1;</code> | <code>res2 = tmp2 - f;</code> | <code>tmp = tmp2;</code> |



Aufbau der superskalaren Pipeline

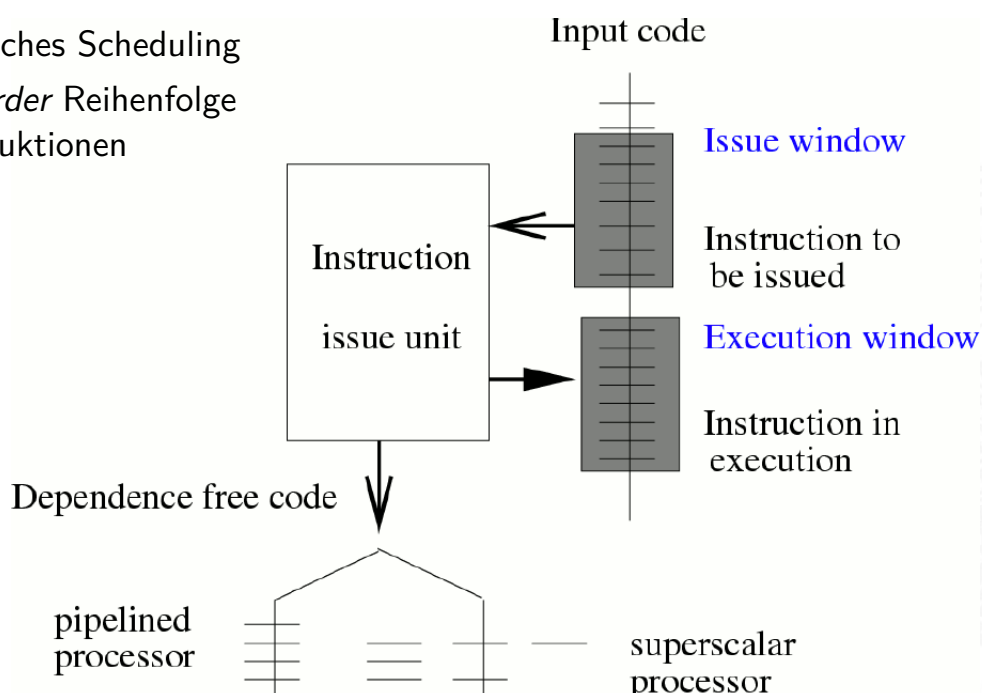


- ▶ lange Pipelines mit vielen Phasen: Fetch (Prefetch, Predecode), Decode / Register-Renaming, Issue, Dispatch, Execute, Retire (Commit, Complete / Reorder), Write-Back
- ▶ je nach Implementation unterschiedlich aufgeteilt
- ▶ entscheidend für superskalare Architektur sind die Schritte vor den ALUs: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



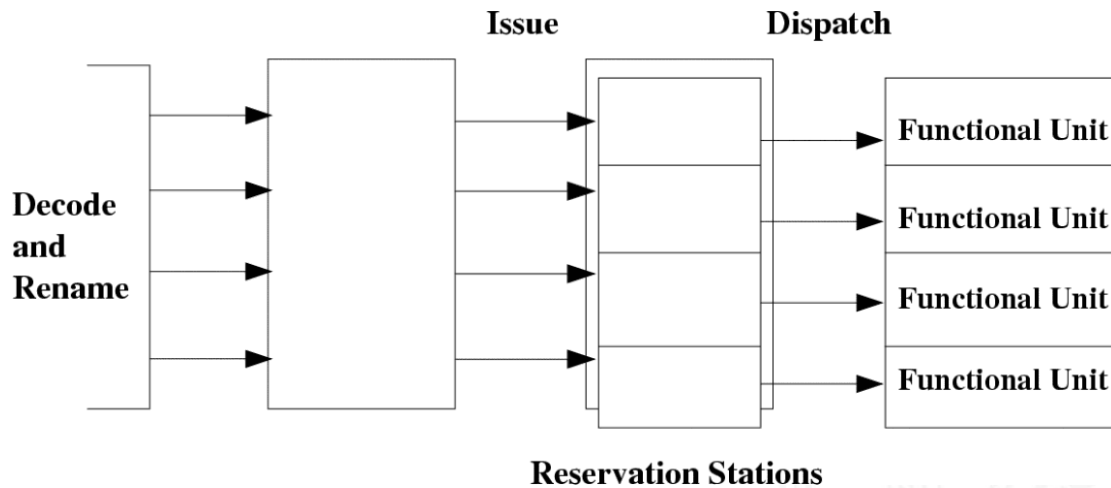
Superskalar – Pipeline (cont.)

- ▶ Dynamisches Scheduling
- \Rightarrow *out-of-order* Reihenfolge der Instruktionen





- ▶ Issue: globale Sicht
Dispatch: getrennte Ausschnitte in „Reservation Stations“



- ▶ Reservation Station für jede Funktionseinheit
 - ▶ speichert: initiierte Instruktionen die auf Recheneinheit warten
 - ▶ –"– zugehörige Operanden
 - ▶ –"– ggf. Zusatzinformation
 - ▶ Instruktion bleibt blockiert, bis alle Parameter bekannt sind
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



Spezielle Probleme superskalarer Pipelines: weitere Hazard-Möglichkeiten

- komplexe Datenabhängigkeiten
 - ▶ die verschiedenen ALUs haben unterschiedliche Latenzzeiten
 - ▶ Befehle „warten“ in den Reservation Stations
- ⇒ Datenabhängigkeiten können sich mit jedem Takt ändern
- Kontrollflussabhängigkeiten
 - ▶ Anzahl der Instruktionen zwischen bedingten Sprüngen limitiert Anzahl parallelisierbarer Instruktionen
- ⇒ Kontextwechsel noch aufwändiger, muss ggf. warten
- ⇒ Optimierungstechniken wichtig
 - ▶ optimiertes (dynamisches) Scheduling: Scoreboard, Tomasulo-Algorithmus
 - ▶ „Loop Unrolling“ (längere Codesequenzen ohne Sprünge)



Softwareunterstützung für Pipelining superskalarer Prozessoren: „Software Pipelining“

- ▶ Codeoptimierungen beim Compilieren als notwendiger Ersatz/Ergänzung der Pipelineunterstützung durch Hardware
 - ▶ Compiler hat „globalen“ Überblick
- ⇒ zusätzliche Optimierungsmöglichkeiten



Beispiel: Pentium 4 / NetBurst Architektur

14.2.2 Rechnerarchitektur II - Parallelität - Superskalare Rechner

64-040 Rechnerstrukturen und Betriebssysteme

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

A. Mäder

1037



Beispiel: Pentium 4 / NetBurst Architektur (cont.)

14.2.2 Rechnerarchitektur II - Parallelität - Superskalare Rechner

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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

A. Mäder

1038



Beispiel: Pentium 4 / NetBurst Architektur (cont.)

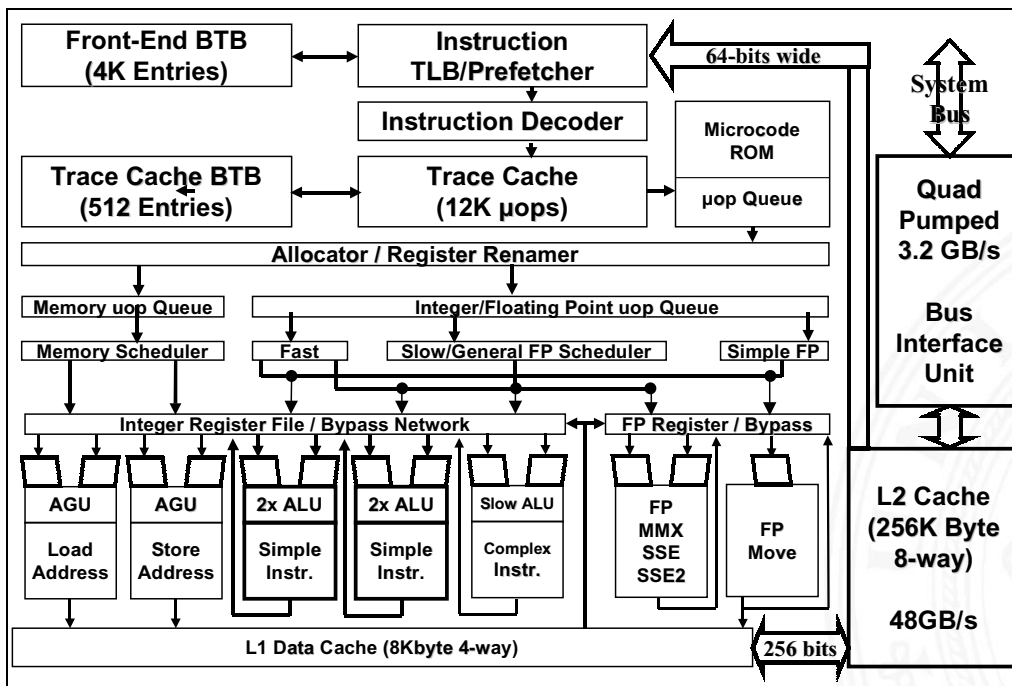
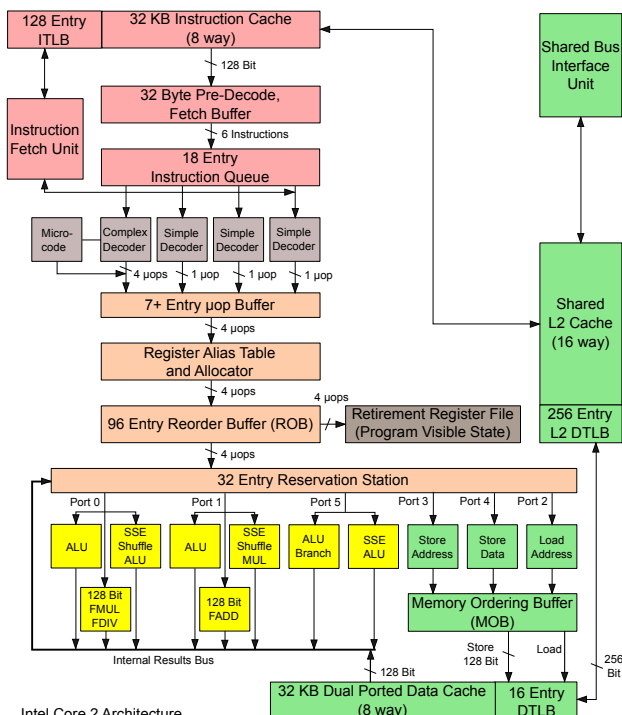


Figure 4: Pentium® 4 processor microarchitecture

Intel: Q1, 2001 [Intel]



Beispiel: Core 2 Architektur



Intel Core 2 Architecture



Parallelrechner mit mehreren Prozessoren

14.2.3 Rechnerarchitektur II - Parallelität - Parallelrechner

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ *Moore's Law* – technischer Fortschritt
 - ▶ immer schnellere Schaltungen
 - ▶ immer mehr Transistoren pro IC möglich
 - ▶ Taktfrequenzen > 10 GHz nicht sinnvoll realisierbar
 - ▶ hoher Takt nur bei einfacher Hardware möglich
 - ▶ Stromverbrauch bei CMOS proportional zum Takt
- ⇒ 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

A. Mäder

1041



Flynn-Klassifikation

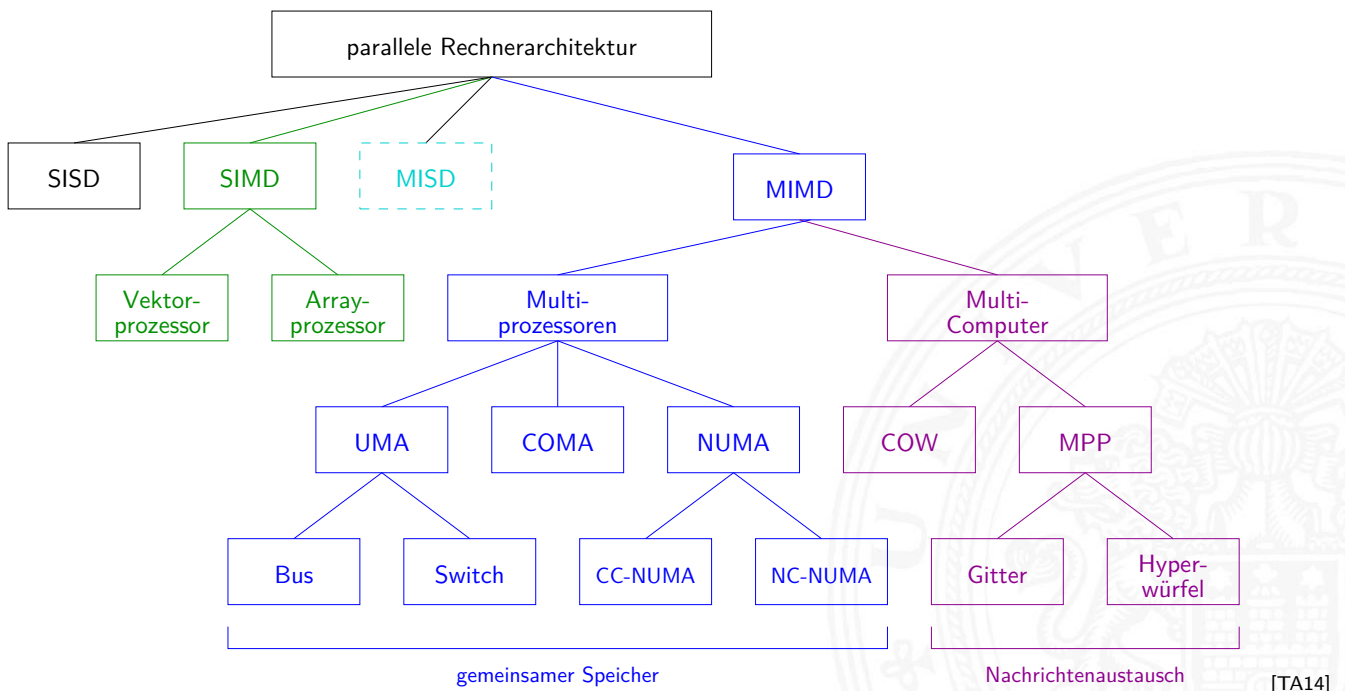
14.2.3 Rechnerarchitektur II - Parallelität - Parallelrechner

64-040 Rechnerstrukturen und Betriebssysteme

- SISD** „*Single Instruction, Single Data*“
 - ▶ jeder klassische von-Neumann Rechner, z.B. PC
- SIMD** „*Single Instruction, Multiple Data*“
 - ▶ Vektorrechner/Feldrechner
 - z.B. Connection-Machine 2: 65 536 Prozessoren
 - ▶ Befehlssatzerweiterungen um superskalare Recheneinheiten direkt anzusprechen
 - 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*“ :-)

A. Mäder

1042



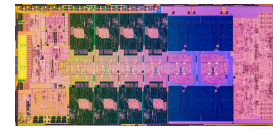
- ▶ Programmierung: ein ungelöstes Problem
 - ▶ Aufteilung eines Programms auf die CPUs/Rechenknoten?
 - ▶ insbesondere bei komplexen Kommunikationsnetzwerken
- ▶ Programme sind nur teilweise parallelisierbar
 - ▶ Parallelität einzelner Programme meist < 8
 - ▶ gilt für Desktop-, Server-, Datenbankanwendungen etc.
 - ⇒ hochgradig parallele Rechner sind dann Verschwendung
- ▶ *Wohin mit den Transistoren aus „Moore’s Law“?*
 - ⇒ SMP-/Mehrkern-CPU (2... 64 Proz.) sind technisch attraktiv
- ▶ Grafikprozessoren (GPUs)
 - ▶ neben 3D-Grafik zunehmender Computing-Einsatz (OpenCL)
 - ▶ Anwendungen: Numerik, Simulation, „Machine Learning“ ...
 - ▶ hohe Fließkomma-Rechenleistung



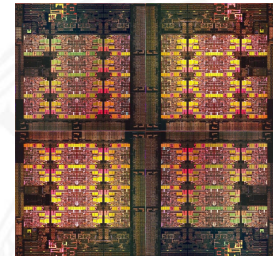
SMP: Symmetric Multiprocessing

- ▶ 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“

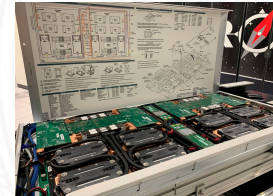
8+16-Kern Core-i9 14900K



60-Kern 4th Gen. Xeon



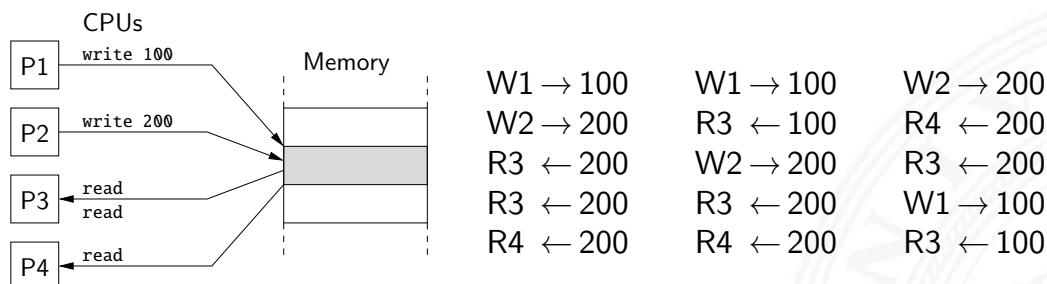
Frontier Blade 2 CPU, 8 GPU



SMP: Eigenschaften

Symmetric Multiprocessing

- ▶ alle CPUs gleichrangig, Zugriff auf Speicher und I/O
- ▶ Konsistenz: *Gleichzeitiger Zugriff auf eine Speicheradresse?*



⇒ „Locking“ Mechanismen und Mutexe

- ▶ spez. Befehle, atomare Operationen, Semaphore etc.
- ▶ explizit im Code zu programmieren

→ siehe 15.4 Betriebssysteme – Synchronisation und Kommunikation



SMP: Cache-Kohärenz

Cache für schnelle Prozessoren notwendig

- ▶ jede CPU hat eigene Caches: 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 seine Caches 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 1062



SMP: volatile

- ▶ MESI-Verfahren garantiert Cache-Kohärenz für Werte in Cache und Hauptspeicher

Vorsicht: Was ist mit den Registern?

- ▶ Variablen in Registern werden von MESI nicht erkannt
- ▶ Compiler versucht, häufig benutzte Variablen soweit möglich in Registern zu halten
- ▶ globale/*shared*-Variablen niemals in Registern halten
- ▶ Java, C: Deklaration als *volatile*



SMP: Erreichbarer Speedup (bis 32 Threads)

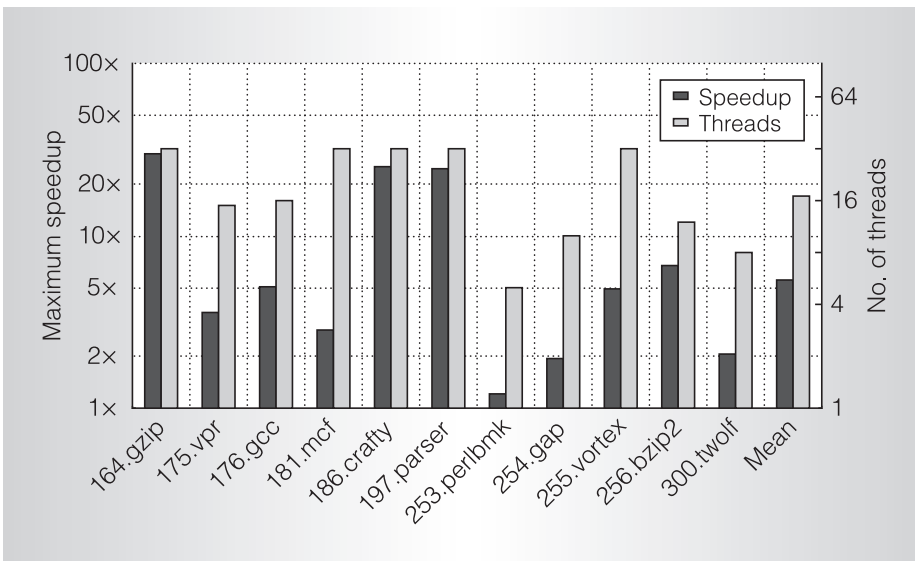
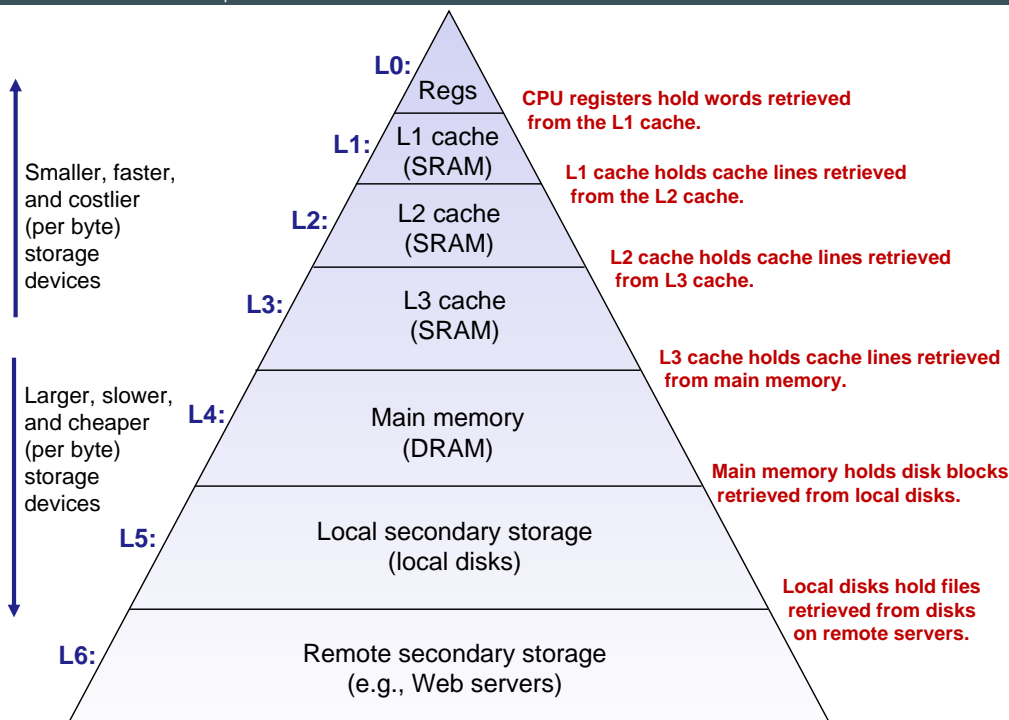


Figure 4. Maximum speedup achieved on up to 32 threads over single-threaded execution (black bars) and minimum number of threads at which the maximum speedup occurred (gray bars).



Speicherhierarchie





Gesamtsystem kombiniert verschiedene Speicher

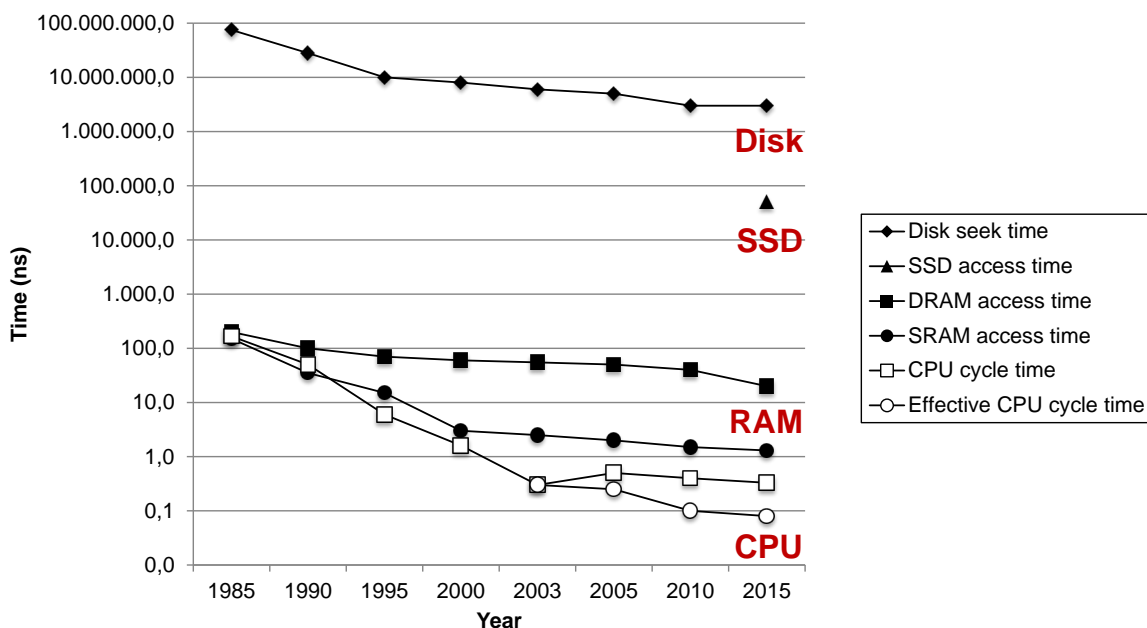
- ▶ wenige KByte Register (-bank) im Prozessor
- ▶ einige MByte SRAM als schneller Zwischenspeicher
- ▶ einige GByte DRAM als Hauptspeicher
- ▶ einige TByte Festplatte als nichtflüchtiger Speicher
- ▶ Hintergrundspeicher (CD/DVD/BR, Magnetbänder)
- ▶ das WWW und Cloud-Services

Kompromiss aus Kosten, Kapazität, Zugriffszeit

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



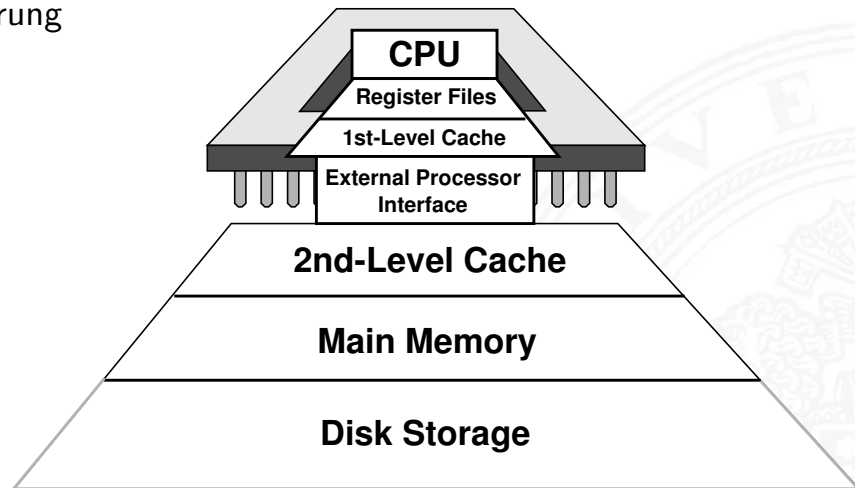
- ▶ stetig wachsende Lücke zwischen CPU-, Memory- und Disk-Geschwindigkeiten





Motivation

- ▶ Geschwindigkeit der Prozessoren vs. Speicher
- ▶ Kosten für den Speicherplatz
- ▶ permanente Speicherung
 - ▶ magnetisch
 - ▶ optisch
 - ▶ mechanisch



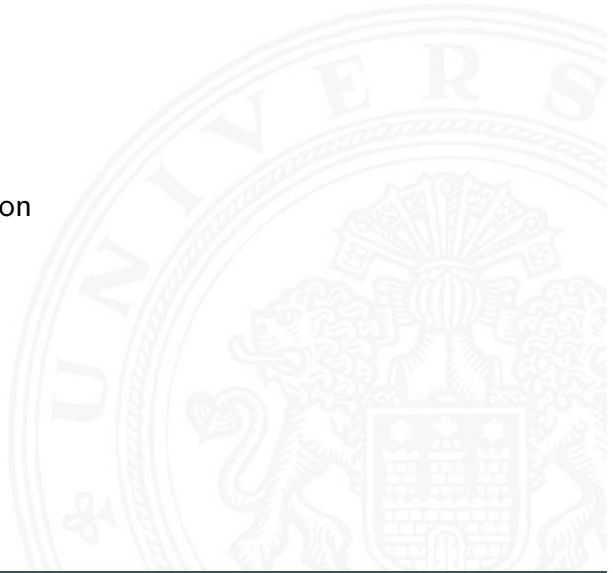
Motivation (cont.)

- ▶ schnelle vs. langsame Speichertechnologie
 - schnell : hohe Kosten/Byte geringe Kapazität
 - langsam : geringe Kosten/Byte hohe Kapazität
- ▶ wachsender Abstand zwischen CPU und Speichergeschwindigkeit
 - ▶ Prozessor läuft mit mehreren GHz Takt
 - ▶ Register können mithalten, aber nur einige KByte Kapazität
 - ▶ DRAM braucht 50...100 ns für Zugriff: 100 × langsamer
 - ▶ Festplatte braucht 2,5...10 ms für Zugriff: 1 000 000 × langsamer
- ▶ Lokalität der Programme wichtig
 - ▶ kleiner Adressraum im Programmkontext
 - ▶ aufeinanderfolgende Speicherzugriffe sind meistens „lokal“
 - ▶ gut geschriebene Programme haben meist eine gute Lokalität

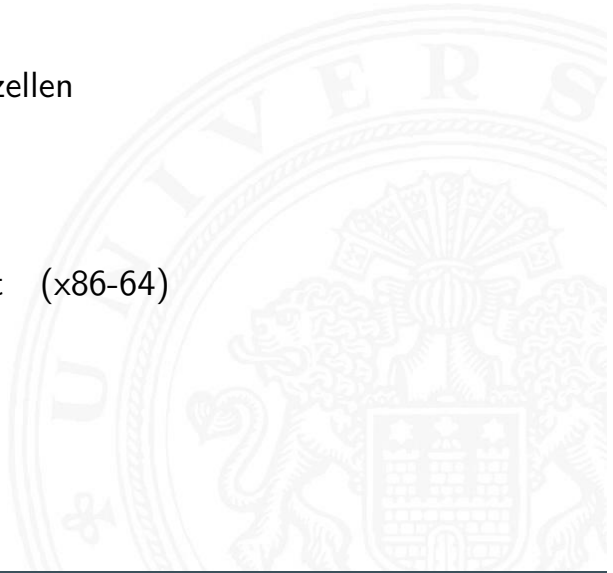
⇒ Motivation für spezielle Organisation von Speichersystemen = **Speicherhierarchie**



- ▶ Register ↔ Memory
 - ▶ Compiler
 - ▶ Assembler-Programmierer
- ▶ Cache ↔ Memory
 - ▶ Hardware
- ▶ Memory ↔ Disk
 - ▶ Hardware und Betriebssystem: Paging, Segmentation
 - ▶ Programmierer und –"– : Dateien



- ▶ 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 à 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 / SSD

- ▶ 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

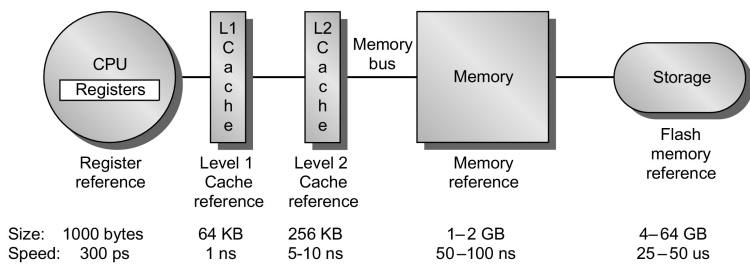


L6: Hintergrundspeicher

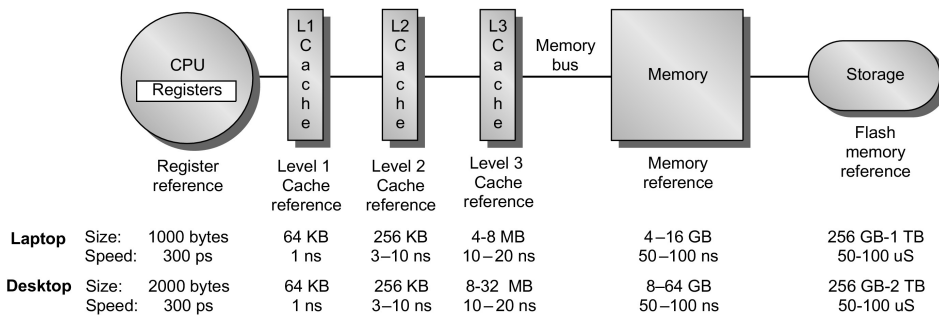
- ▶ enorme Speicherkapazität
- ▶ langsame Zugriffszeiten
- ▶ Archivspeicher und Backup für (viele) Festplatten
 - ▶ Magnetbänder
 - ▶ RAID-Verbund aus mehreren Festplatten
 - ▶ optische Datenspeicher: CD-ROM, DVD-ROM, BlueRay
- ▶ WWW und Internet-Services, Cloud-Services
 - ▶ Cloud-Farms ggf. ähnlich schnell wie L5 Festplatten
 - ▶ Netzwerk schneller als der Zugriff auf eine lokale (Magnet-) Festplatte
- ▶ in dieser Vorlesung nicht behandelt



Speicherhierarchie: zwei Beispiele



(A) Memory hierarchy for a personal mobile device



(B) Memory hierarchy for a laptop or a desktop

[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

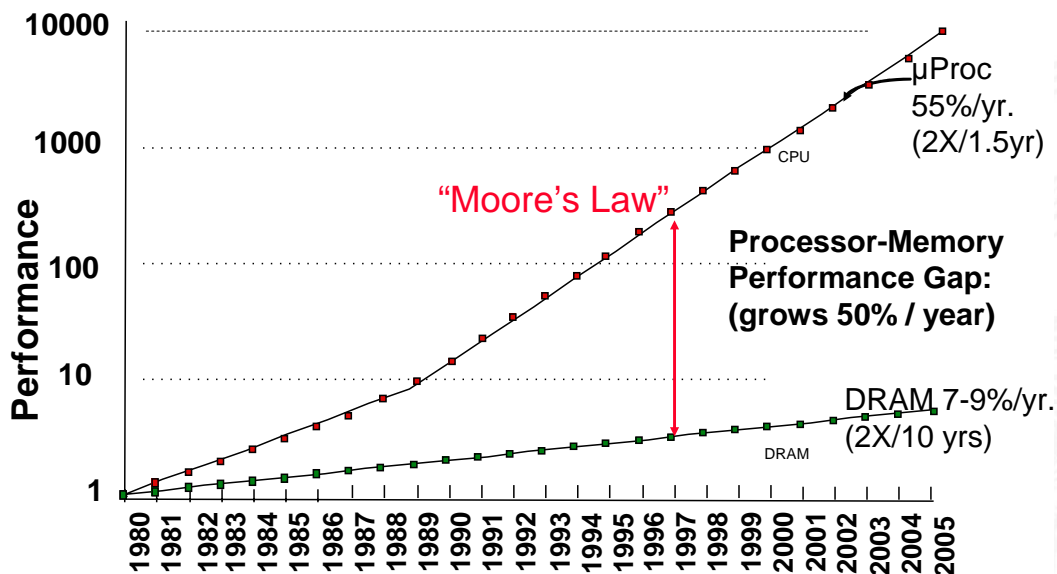
► Beispiel	Hauptspeicher	Festplatte	SSD
Latenz	6 ns	2 ms	10/40 μ s
Bandbreite	64 GB/sec (pro Kanal, bis 8)	1,5 GB/sec	7/4 GB/sec (r/w)
Kosten/GB	2,5 €	1,5 ct. 1 TB: 15 €	4,1 ct. 43 €



Cache

► „Memory Wall“: DRAM zu langsam für CPU

Processor-DRAM Memory Gap (latency)



[PH22]



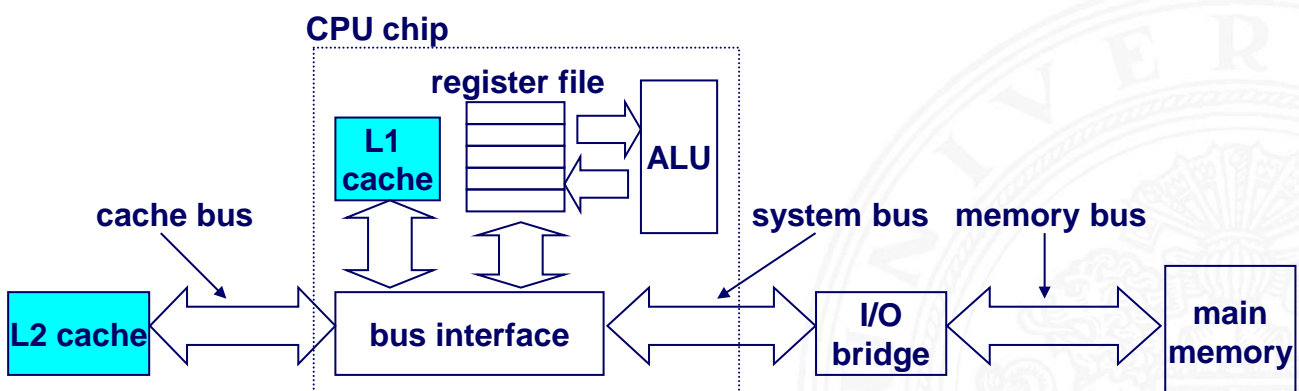
Cache (cont.)

- ⇒ Cache als schneller Zwischenspeicher zum Hauptspeicher
 - ▶ 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
 - ▶ nutzt die Lokalität von Speicherzugriffen durch ein laufendes Programm
 - ▶ ca. 80% der Zugriffe greifen auf 20% der Adressen zu
 - ▶ manchmal auch 90% / 10% oder noch besser
 - ▶ de.wikipedia.org/wiki/Cache
en.wikipedia.org/wiki/CPU_cache
[en.wikipedia.org/wiki/Cache_\(computing\)](http://en.wikipedia.org/wiki/Cache_(computing))



Cache (cont.)

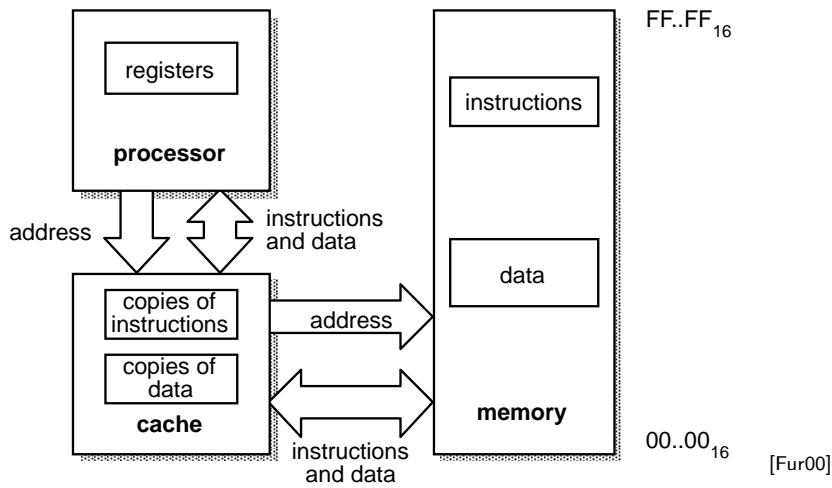
- ▶ CPU referenziert Adresse
 - ▶ parallele Suche in L1 (level 1), L2 ... und Hauptspeicher
 - ▶ erfolgreiche Suche liefert Datum, Abbruch laufender Suchen



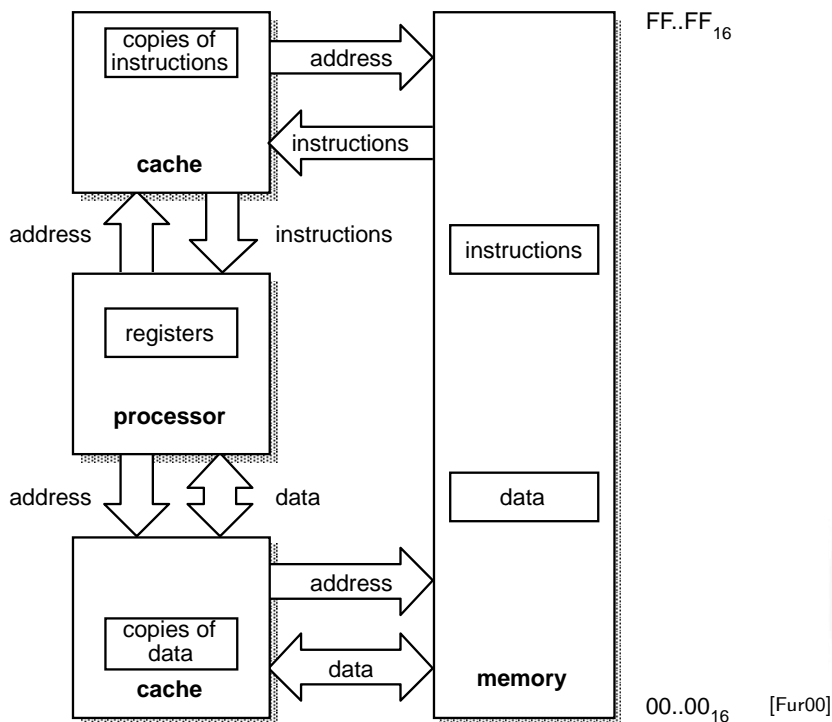
[BO15]



gemeinsamer Cache / „unified Cache“



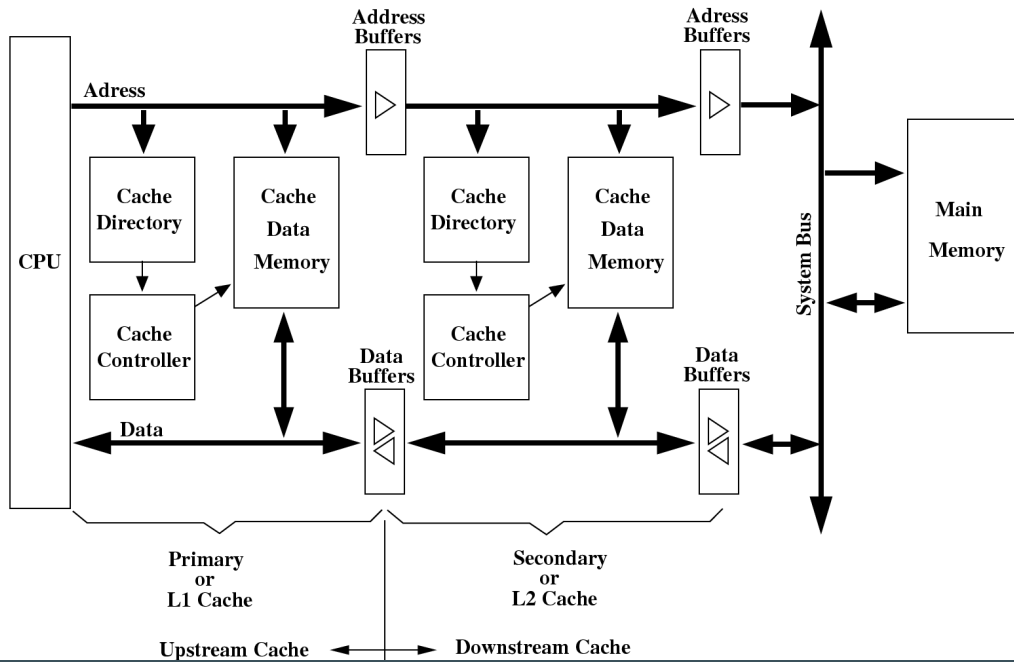
separate Instruction-/Data Caches





Cache – Position

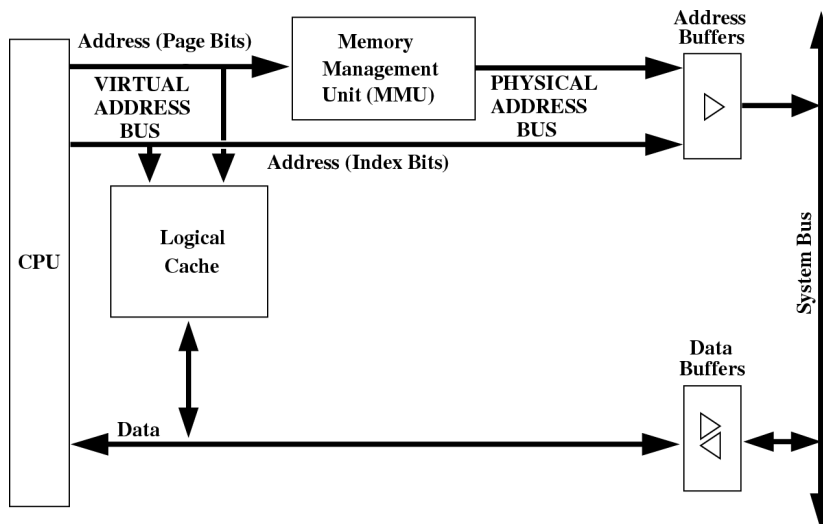
► First- und Second-Level Cache



Cache – Position (cont.)

► Virtueller Cache

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





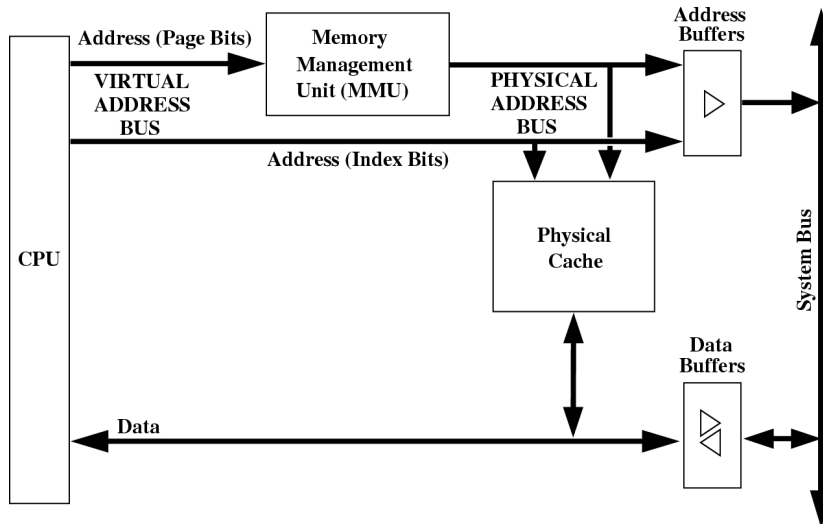
Cache – Position (cont.)

14.3.2 Rechnerarchitektur II - Speicherhierarchie - Cache Speicher

64-040 Rechnerstrukturen und Betriebssysteme

► Physikalischer Cache

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



A. Mäder

1069



Cache – Position (cont.)

14.3.2 Rechnerarchitektur II - Speicherhierarchie - Cache Speicher

64-040 Rechnerstrukturen und Betriebssysteme

► typische Cache Organisation

- First-Level Cache: getrennte Instruktionen- und Daten-Caches
- Second-Level Cache: gemeinsamer Cache je Prozessorkern
- Third-Level Cache: gemeinsamer Cache für alle Prozessorkerne

► bei mehreren Prozessoren / Prozessorkernen

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

A. Mäder

1070



Cachestrategie: *Welche Daten sollen in den Cache?*

Diejenigen, die bald wieder benötigt werden!

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



Cacheperformanz

- ▶ Begriffe

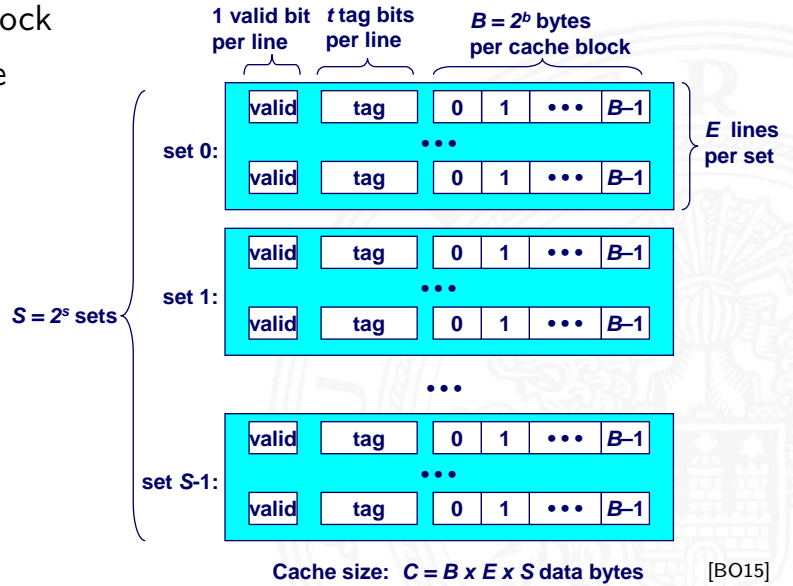
Treffer (Hit)		Zugriff auf Datum, ist bereits im Cache
Fehler (Miss)		–"– ist nicht –"–
Treffer-Rate	R_{Hit}	Wahrscheinlichkeit, Datum ist im Cache
Fehler-Rate	R_{Miss}	$1 - R_{Hit}$
Hit-Time	T_{Hit}	Zeit, bis Datum bei Treffer geliefert wird
Miss-Penalty	T_{Miss}	zusätzlich benötigte Zeit bei Fehler

- ▶ 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



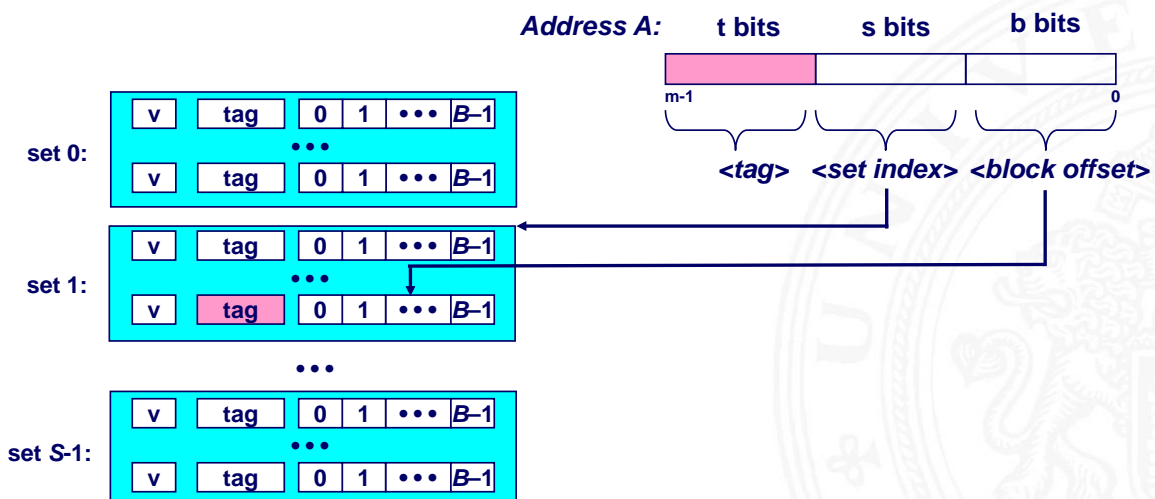
Cache – Organisation

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



Cache – Adressierung

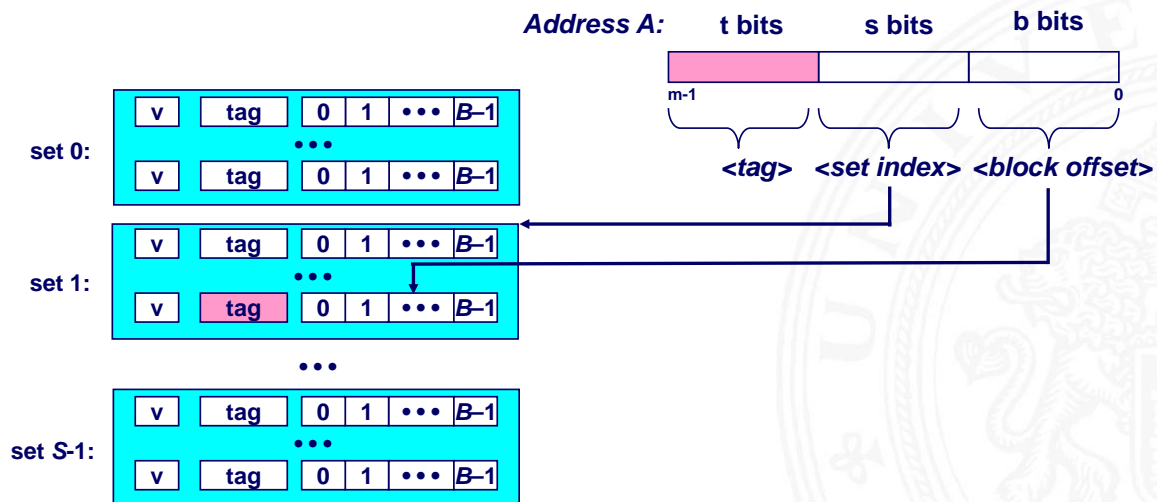
- ▶ Adressteil $\langle set\ index \rangle$ von A bestimmt Bereich („set“)
- ▶ Adresse A ist im Cache, wenn
 1. Cache-Zeile ist als gültig markiert („valid“)
 2. Adressteil $\langle tag \rangle$ von $A =$ „tag“ Bits des Bereichs





Cache – Adressierung (cont.)

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



[BO15]



Cache – Varianten

- ▶ 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

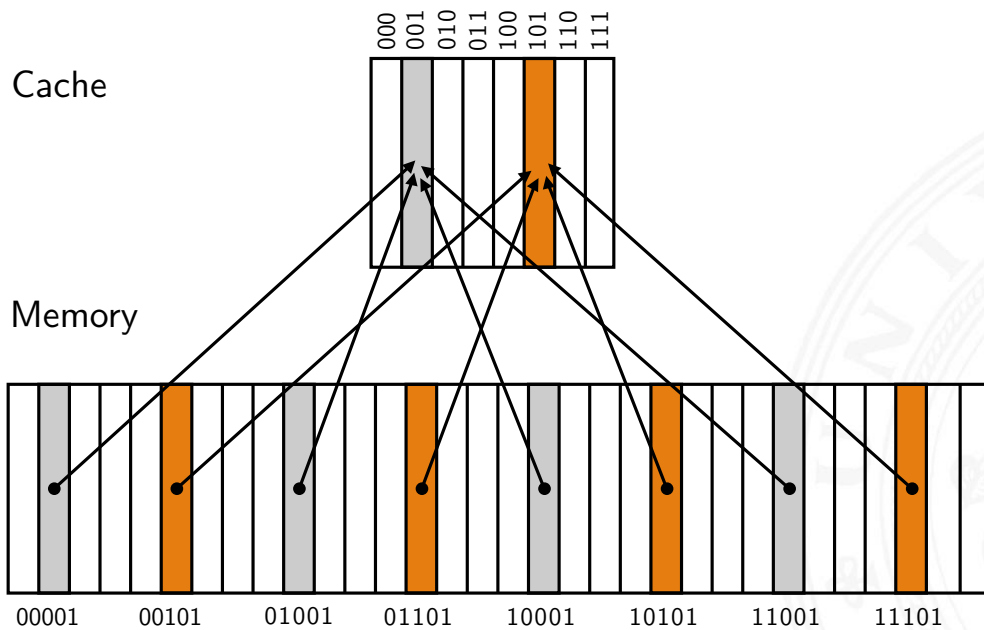
jeder Speicheradresse ist eine von n möglichen Cache-Speicherzellen zugeordnet, Cache-Line E in den Grafiken [BO15]

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

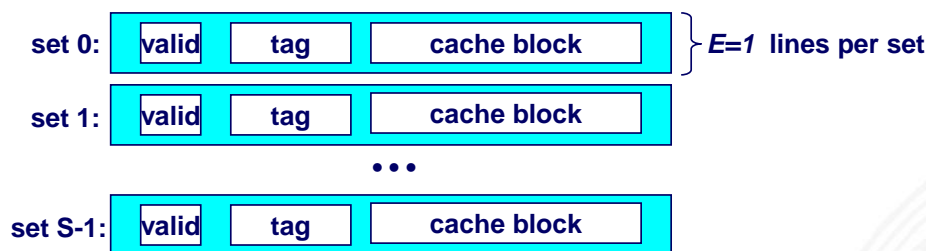


[PH22]



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

- ▶ genau 1 Zeile („cache line“) pro Bereich, S Bereiche (Sets)



[BO15]

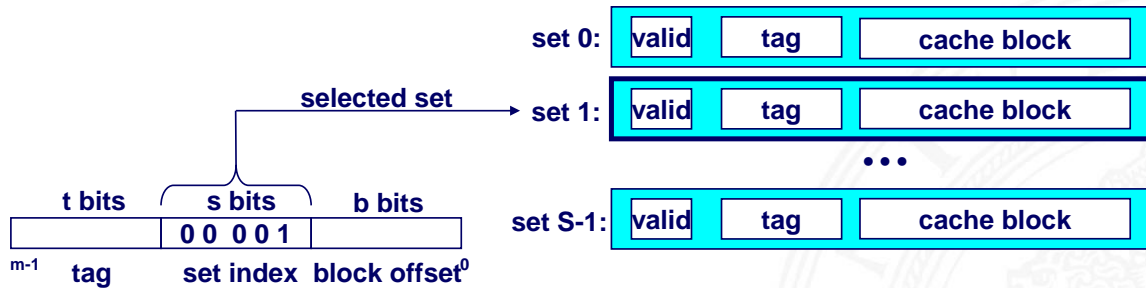
- + einfachste Cache-Art
- + große Caches möglich
- Effizienz, z.B. Zugriffe auf $A, A + n \cdot S \dots \Rightarrow$ „Cache Thrashing“
 Beispiel (s.o.): Zugriff auf Adressen „00101“, „01101“, „10101“, „11101“



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

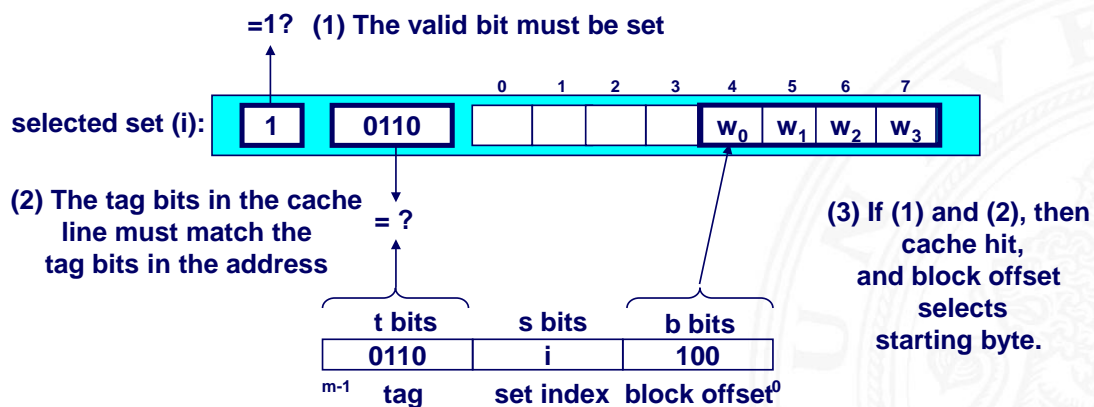
Zugriff auf direkt abgebildete Caches

1. Bereichsauswahl durch Bits $\langle set\ index \rangle$



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

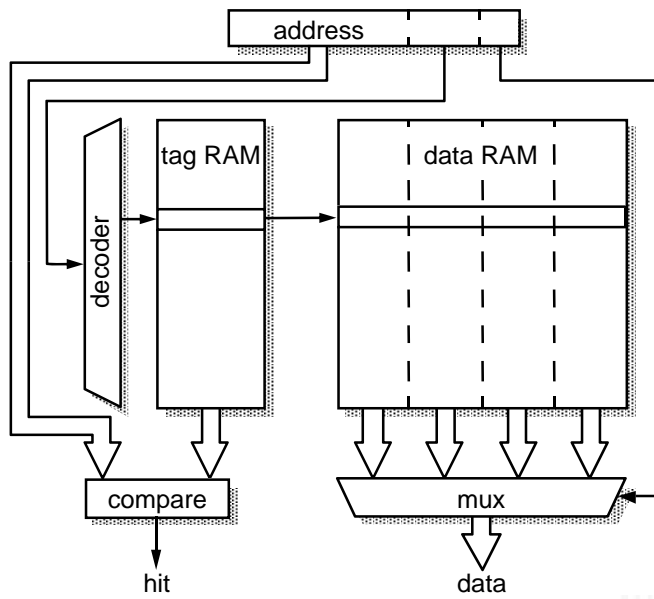
2. $\langle valid \rangle$: sind die Daten gültig?
3. „Line matching“: stimmt $\langle tag \rangle$ überein?
4. Wortselektion extrahiert Wort unter Offset $\langle block\ offset \rangle$





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

Prinzip

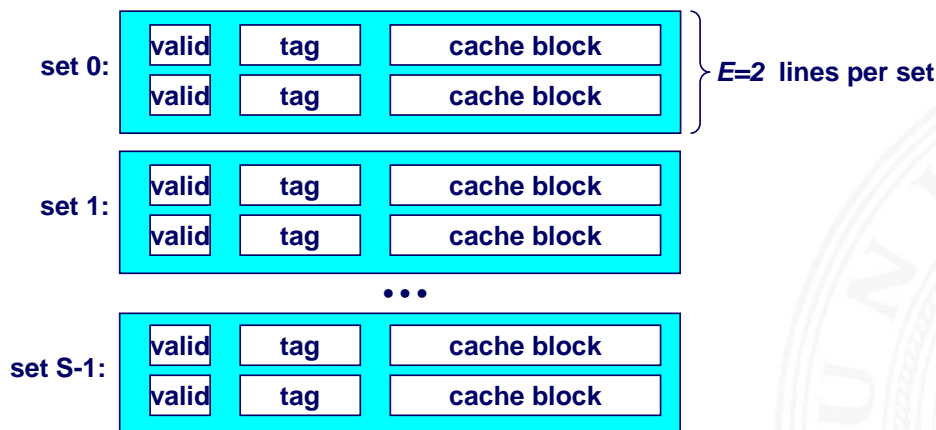


[Fur00]



Cache: bereichsassoziativ / „set associative“

- ▶ jeder Speicheradresse ist ein Bereich S mit mehreren Cachezeilen E zugeordnet
- ▶ n -fach assoziative Caches: $E=2, 4 \dots$ „2-way set associative cache“, „4-way ...“



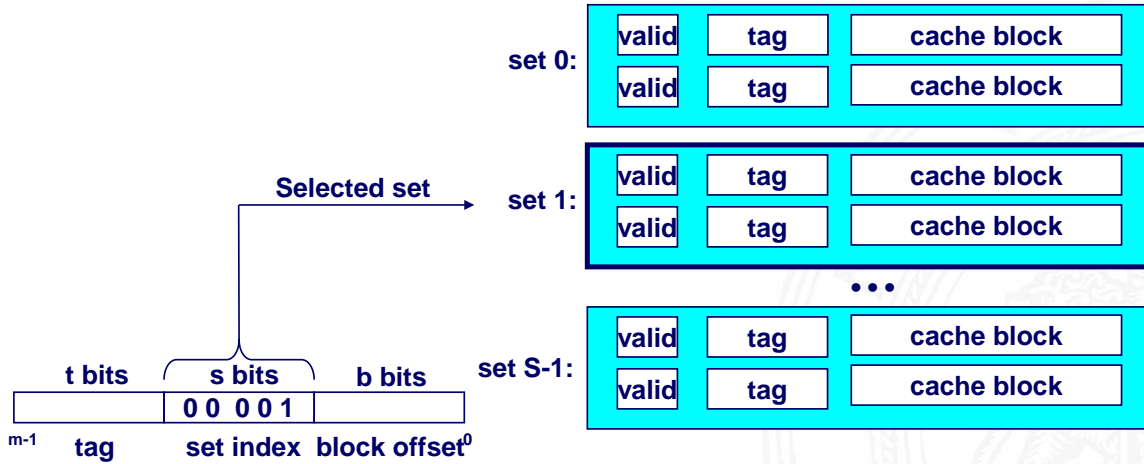
[BO15]



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

Zugriff auf n-fach assoziative Caches

1. Bereichsauswahl durch Bits $\langle set\ index \rangle$

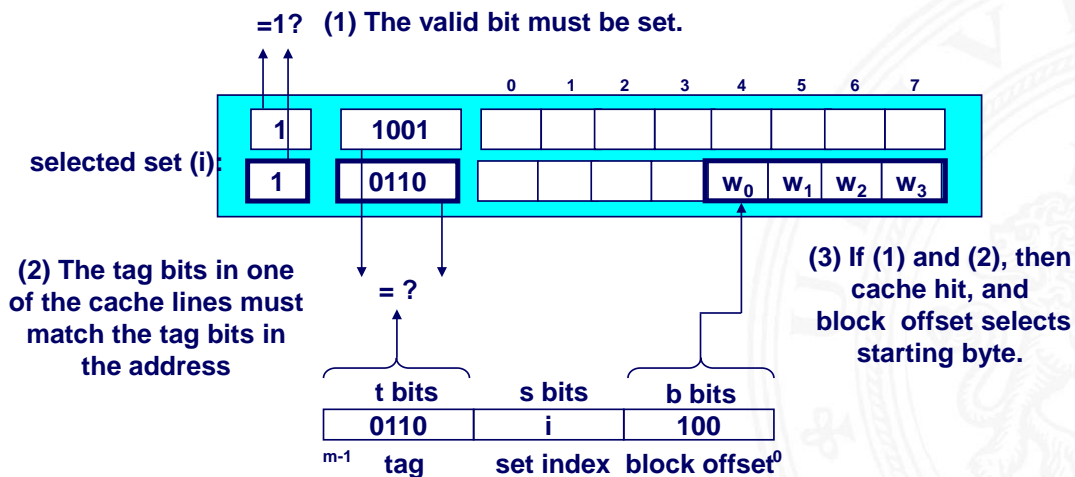


[BO15]



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

- $\langle valid \rangle$: sind die Daten gültig?
- „Line matching“: Cache-Zeile mit passendem $\langle tag \rangle$ finden?
dazu Vergleich aller „tags“ des Bereichs $\langle set\ index \rangle$
- Wortselektion extrahiert Wort unter Offset $\langle block\ offset \rangle$



[BO15]

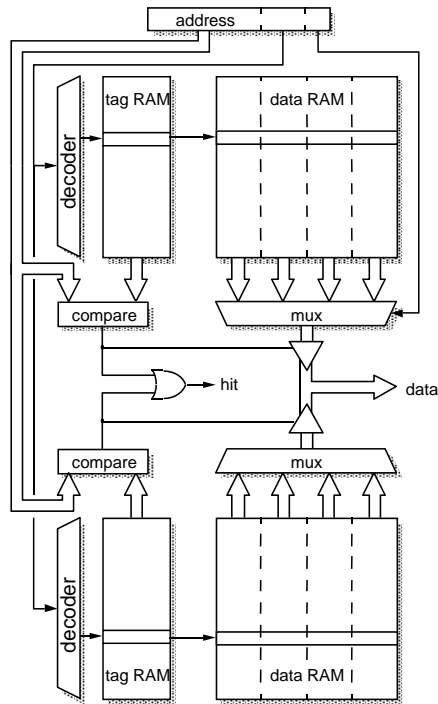


Cache: bereichsassoziativ / „set associative“ (cont.)

14.3.2 Rechnerarchitektur II - Speicherhierarchie - Cache Speicher

64-040 Rechnerstrukturen und Betriebssysteme

Prinzip



[Fur00]

A. Mäder

1085



Cache: voll-assoziativ

14.3.2 Rechnerarchitektur II - Speicherhierarchie - Cache Speicher

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ jeder Adresse des Speichers kann jede beliebige Cachezeile zugeordnet werden
- ▶ Spezialfall: nur ein Cachebereich S
- benötigt E -Vergleicher
- nur für sehr kleine Caches realisierbar

A. Mäder

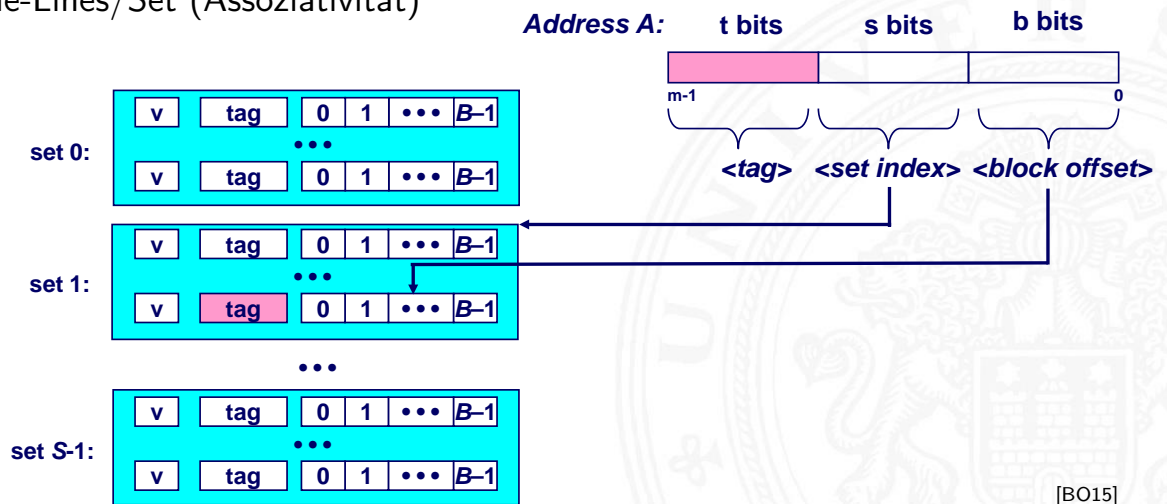
1086



Cache – Dimensionierung

Parameter

- ▶ $S = \#$ Cache Bereiche Teil der Adresse
- ▶ $B = \text{Byte/Block}$ Teil der Adresse
- ▶ $E = \text{Cache-Lines/Set (Assoziativität)}$



Cache – Dimensionierung (cont.)

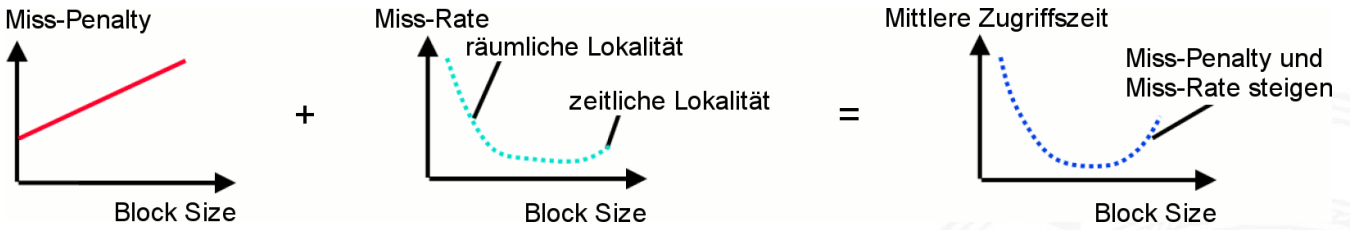
Vor- und Nachteile des Cache

- + 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

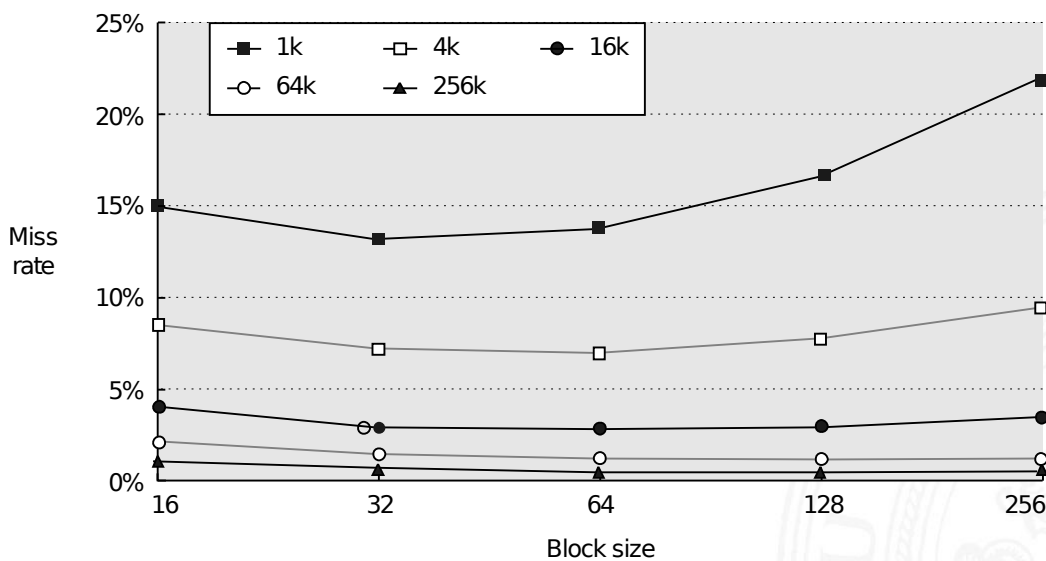


[PH22]

- ▶ 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 Cache Kapazität



Cache Ersetzungsstrategie

Wenn der Cache gefüllt ist, welches Datum wird entfernt?

- ▶ zufällige Auswahl
- ▶ **LRU (Least Recently Used)**: 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 (Least Frequently Used)**: der am seltensten benutzte Cache Eintrag
 - ▶ durch Zugriffszähler implementiert



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 dann in den Speicher schreiben, wenn das Datum des Cache ersetzt werden würde
 - + häufig genutzte Werte (z.B. lokale Variablen) werden nur im Cache modifiziert
 - Cache-Kohärenz ist nicht gegeben
 - ⇒ spezielle Befehle für „Cache-Flush“
 - ⇒ „non-cacheable“ Speicherbereiche



- ▶ Daten zwischen Cache und Speicher konsistent halten
- ▶ notwendig wenn mehrere Einheiten (Bus-Master: Prozessor, DMA-Controller) auf den Speicher zugreifen können: wichtig für „*Symmetric Multiprocessing*“
- ▶ Harvard-Architektur hat getrennte Daten- und Instruktions-Speicher
 - ▶ Instruktionen sind read-only
 - ⇒ einfacherer Instruktions-Cache
 - ⇒ Cache-Kohärenz Problem betrifft D-Cache
- ▶ Cache-Kohärenz Protokolle und „*Snooping*“
 - ▶ alle Prozessoren ($P_1, P_2 \dots$) 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 ?



- ▶ 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

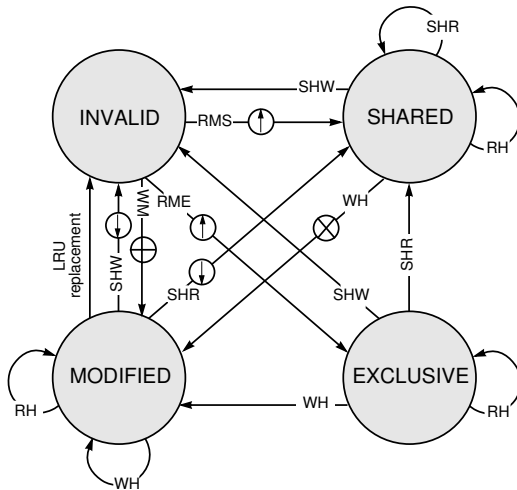


- ▶ Caches enthalten Wert, Tag und zwei Statusbits für die vier Protokollzustände
 - ▶ **Modified**: gültiger Wert, nur in diesem Cache, gegenüber Hauptspeicher-Wert verändert
 - ▶ **Exclusive**: gültiger Wert, nur in diesem Cache nicht verändert (unmodified)
 - ▶ **Shared**: gültiger Wert, in mehreren Caches vorhanden nicht verändert (unmodified)
 - ▶ **Invalid**: 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

Diagramm	Event	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

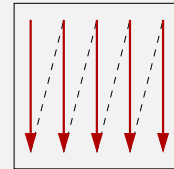
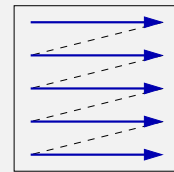


Cache Effekte bei Matrixzugriffen

```

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

Cache-freundlicher Code

- ▶ Erreichen der optimalen Leistung ist plattformspezifisch
 - ▶ Cachegrößen, Zeilengrößen, Assoziativität etc.
 - ▶ Optimierung durch Compiler / manuell fehleranfällig
- ▶ generelle Empfehlungen
 - ▶ „working set“ klein ⇒ zeitliche Lokalität
 - ▶ kleine Adressfortschaltungen („strides“) ⇒ räumliche Lokalität



Literatur

14.4 Rechnerarchitektur II - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [PH22] D.A. Patterson, J.L. Hennessy: *Rechnerorganisation und Rechnerentwurf – Die Hardware/Software-Schnittstelle – MIPS Edition*.
6. Auflage, De Gruyter Oldenbourg, 2022. ISBN 978-3-11-075598-5
- [PH20] D.A. Patterson, J.L. Hennessy: *Computer Organization and Design – The Hardware Software Interface – RISC-V Edition*.
2nd edition, Morgan Kaufmann Publishers Inc., 2020. ISBN 978-0-12-820331-6
- [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
- [Fur00] S. Furber: *ARM System-on-Chip Architecture*.
2nd edition, Pearson Education Limited, 2000. ISBN 978-0-201-67519-1



Literatur (cont.)

14.4 Rechnerarchitektur II - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [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
- [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.



Gliederung

15 Betriebssysteme

64-040 Rechnerstrukturen und Betriebssysteme

1. Einführung
2. Informationsverarbeitung
3. Ziffern und Zahlen
4. Arithmetik
5. Zeichen und Text
6. Logische Operationen
7. Codierung
8. Schaltfunktionen
9. Schaltnetze
10. Schaltwerke
11. Rechnerarchitektur I
12. Instruction Set Architecture



A. Mäder

1103



Gliederung (cont.)

15 Betriebssysteme

64-040 Rechnerstrukturen und Betriebssysteme

13. Assembler-Programmierung
14. Rechnerarchitektur II
15. **Betriebssysteme**
 - Historische Entwicklung
 - Interrupts
 - Prozesse und Threads
 - Synchronisation und Kommunikation
 - Scheduling
 - Speicherverwaltung
 - I/O und Dateiverwaltung



A. Mäder

1104



- ▶ genug Stoff für eigene Vorlesungen – Übersicht in RSB
- ▶ 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
 - *Moderne Betriebssysteme – VL + Übung*
 - *VSS (Verteilte Systeme und Systemsicherheit)*
 - *ES (Eingebettete Systeme)*

Grafiken, wenn nicht anders angegeben, aus:

W. Stallings: *Operating Systems – Internals and Design Principles* [Sta17]

Sicherheit, RAID
Embedded OS, Echtzeit



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)
- ⇒ 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?*
- ⇒ BS sind meist die komplexeste Software auf dem Computer!



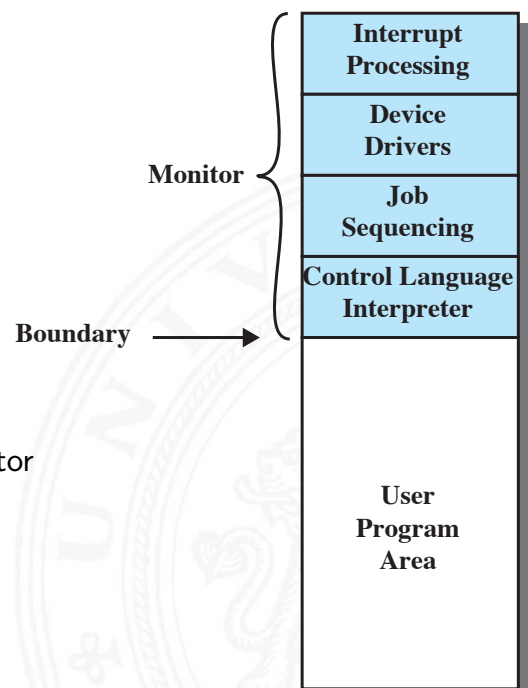
1. erste Computer / serielle Verarbeitung

- ▶ 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 (≙ Prozeduraufruf)
 - ▶ Programm übergibt nach Ende Kontrolle an Monitor (≙ 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

⇒ 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

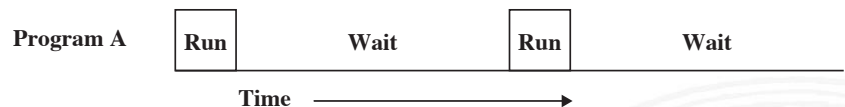
- ▶ Overhead, verglichen mit serieller Abarbeitung
 - Prozessor muss zusätzlich Monitor bearbeiten
 - zusätzlicher Speicherbedarf für Monitor
 - + insgesamt aber bessere Auslastung des Computers



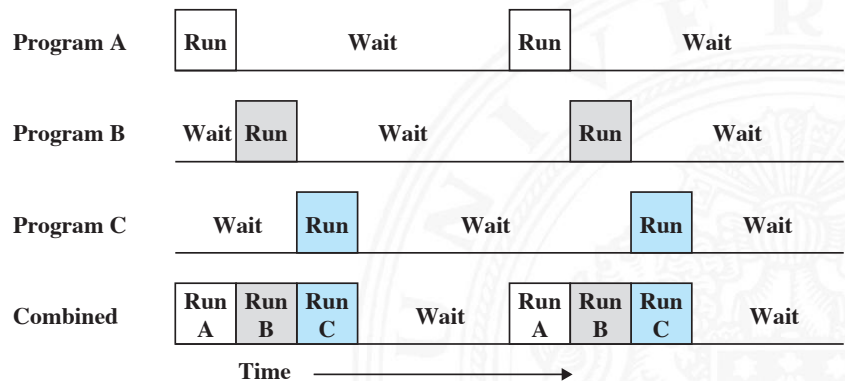
3. Multiprogramm Batch-Systeme

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.)

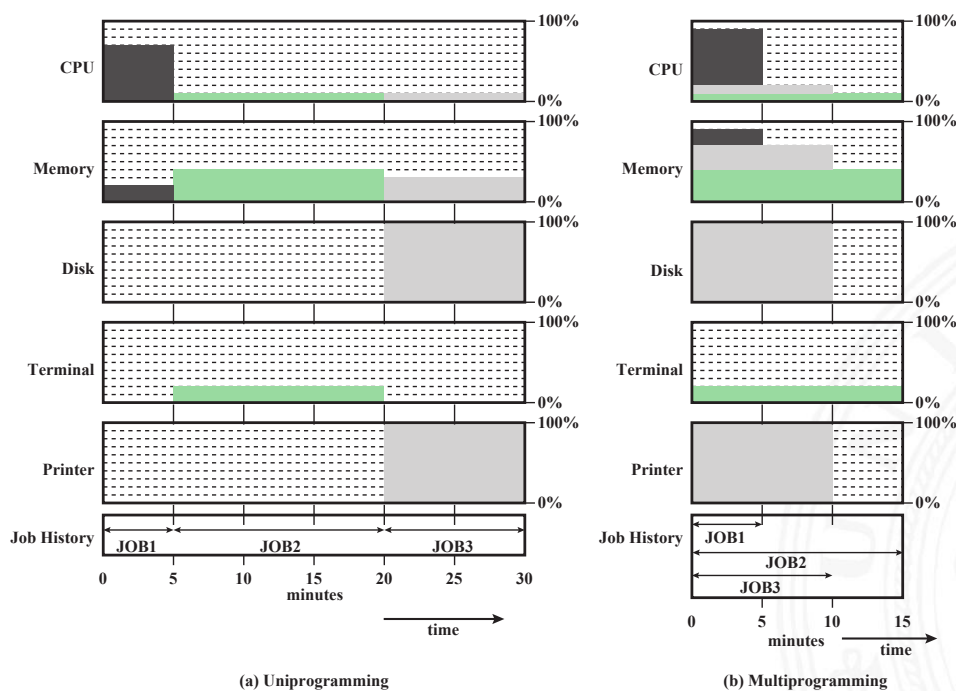
► Beispiel

	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.)





4. Time-Sharing Betrieb

- ▶ Erweiterung von Multitasking für interaktive Jobs
- ▶ Prozessor/Ressourcen werden zwischen Benutzern geteilt
- ▶ Zugriff über Terminals (Kommandozeile), später grafische Oberflächen
- ▶ Optimierungsziel

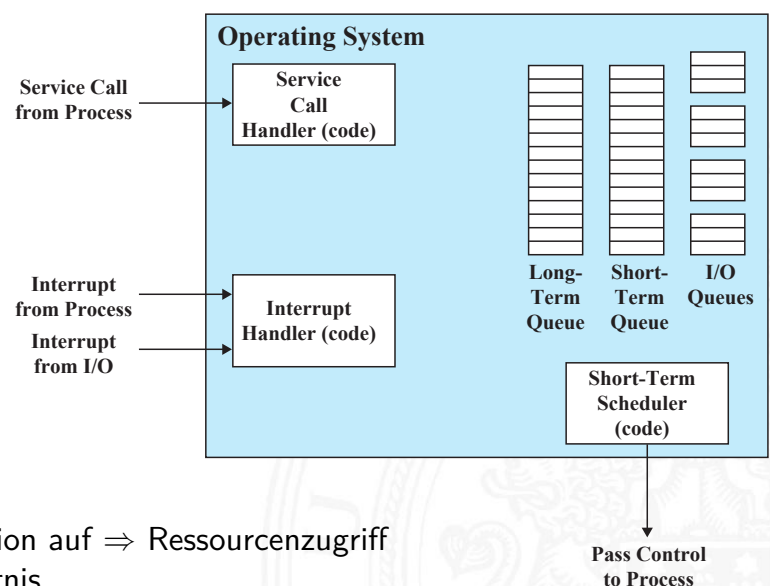
	Batch Multiprogramm	Time-Sharing
Optimierung	maximale Prozessornutzung	minimale Antwortzeit
BS Kontrolle	Job Control Language	Benutzereingabe

- ▶ typisch: Zeitscheiben Verfahren (*Time Slicing*)
 - ▶ periodische Interrupts durch Systemclock
 - ▶ Betriebssystem übernimmt Kontrolle
 - ▶ prüft ob anderer Prozess laufen soll
 - ▶ Benutzerprozess wird „unterbrochen“ („*Preemption*“)
 - ▶ sein Status wird gesichert
 - ▶ Daten für neuen / fortzusetzenden Prozess werden geladen



aktuelle Betriebssysteme

- ▶ zentrale Elemente

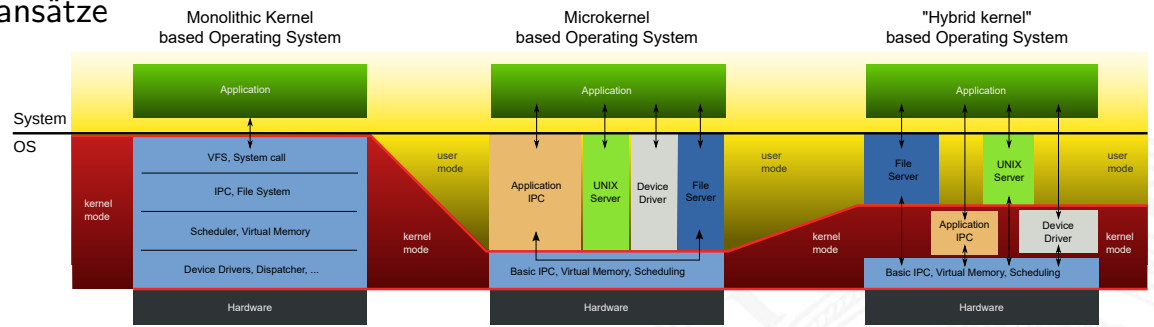


- ▶ **Service Call / System Call:** Programm ruft BS-Funktion auf ⇒ Ressourcenzugriff
- ▶ **Interrupt:** besonderes Ereignis
- ▶ **FIFO Queues:** Warteschlangen
- ▶ **Scheduler:** CPU / Kontrolle wird an Prozess übergeben



aktuelle Betriebssysteme (cont.)

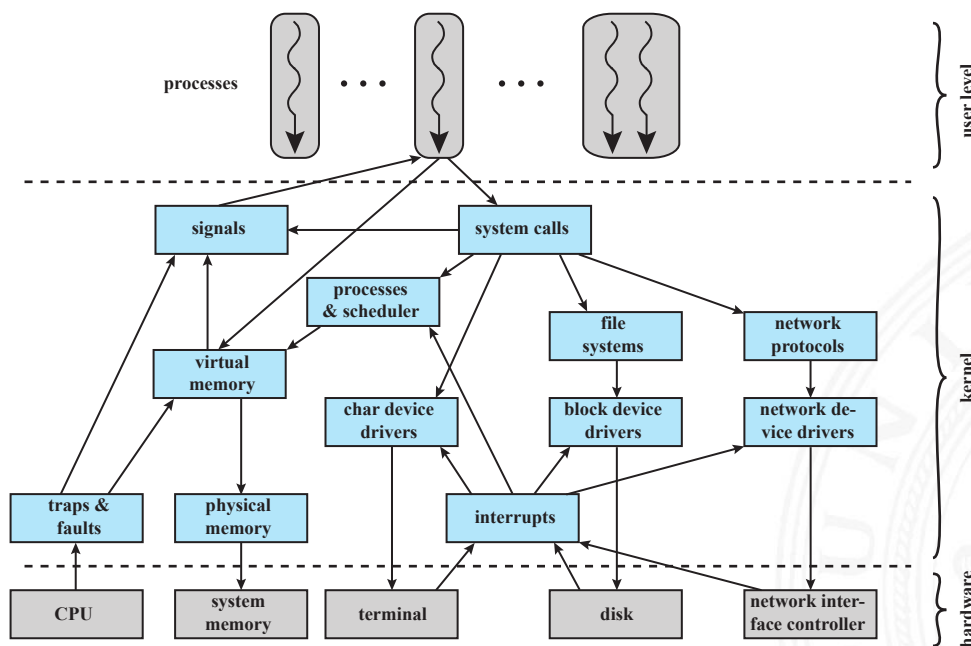
Architekturansätze



- ▶ **Monolithischer Kernel** alle Funktionalitäten, Treiber etc. zusammengefasst das Programm „Betriebssystem“
- ▶ **Mikrokernel** enthält nur
 - ▶ Scheduling
 - ▶ Interprozess-Kommunikation
 - ▶ Adressverwaltung
 - ▶ restliche Funktionalität als getrennte Prozesse
- ▶ **hybride Kernel** Mischformen



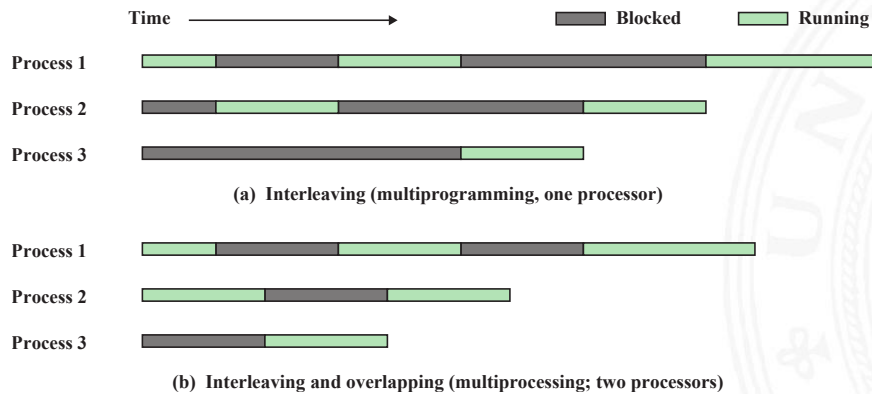
aktuelle Betriebssysteme (cont.)



Monolithischer Kernel (Linux): Teilkomponenten



- ▶ weitere Konzepte
 - ▶ Multithreading \Rightarrow bessere Granularität
 - ▶ Multiprocessing (SMP)
 - ▶ Verwaltung mehrerer Prozessoren
 - ▶ für Benutzer transparent
 - + Verfügbarkeit, Performanz, Skalierbarkeit etc.
 - schwierig zu implementieren ...



- ▶ verteilte Betriebssysteme, einheitliche Sicht auf Cluster
- ▶ spezielle Anforderungen
 - ▶ Echtzeit Betriebssysteme
 - ▶ Fehlertoleranz



- ▶ 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*)



- ▶ 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 ...

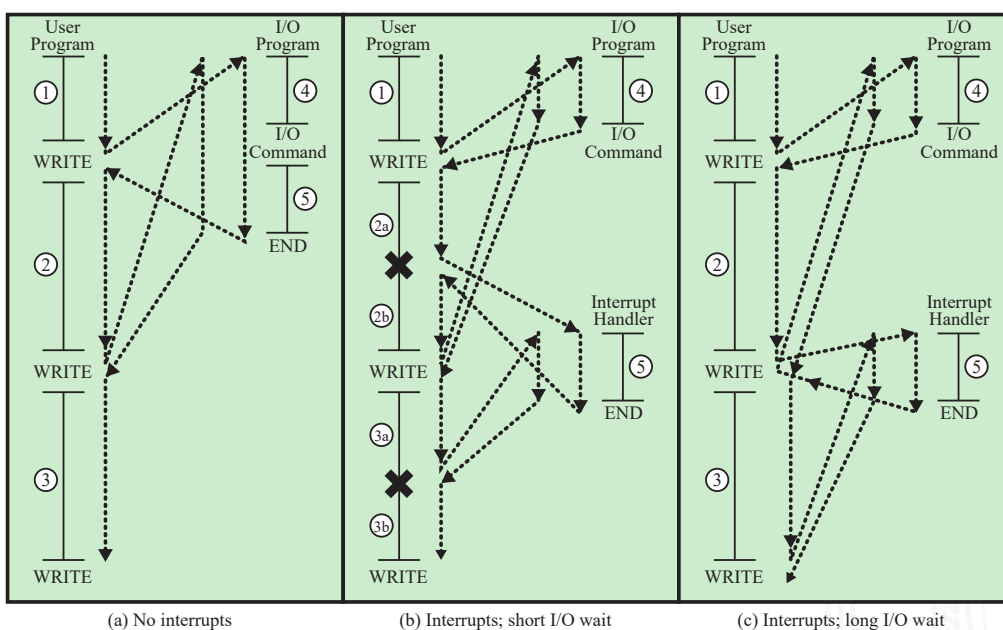


Interrupt: Beispiel

- ▶ 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.)

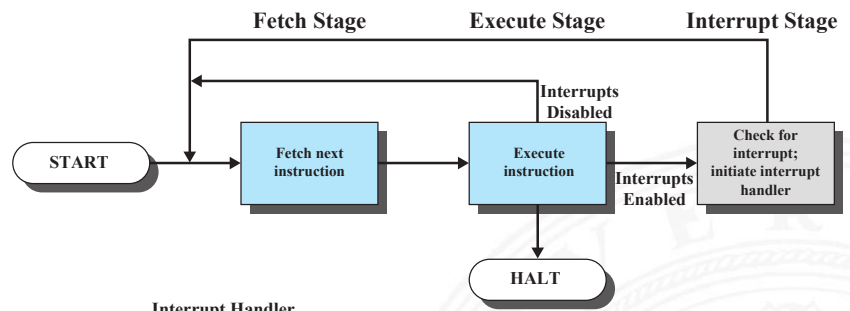


✗ = interrupt occurs during course of execution of user program

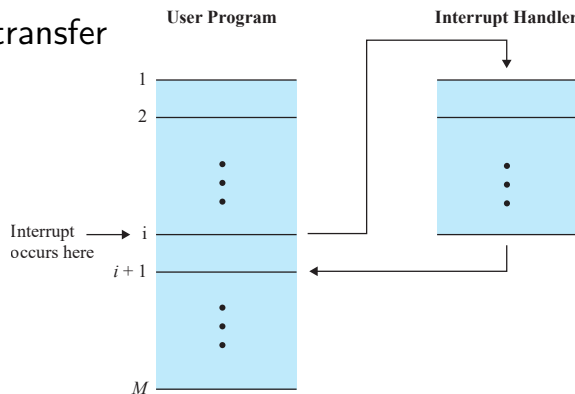


Interrupt: Programmablauf

- ▶ Ausführungszyklus ergänzt

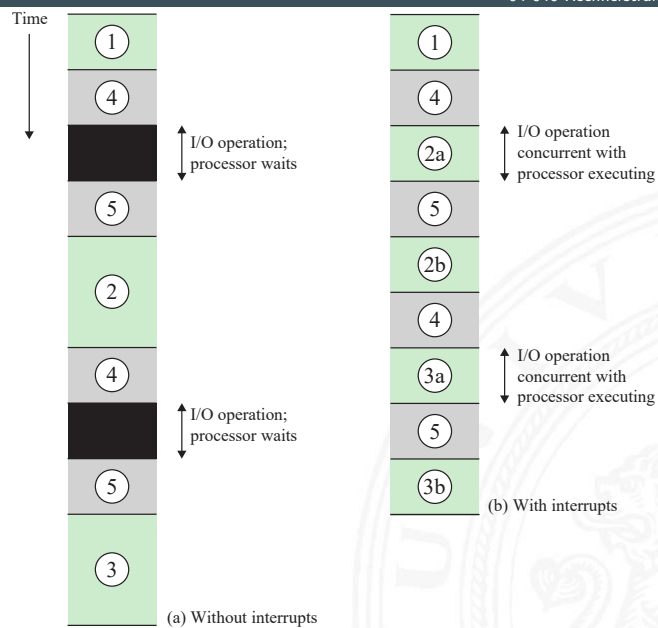


- ▶ Kontrolltransfer



Interrupt: Programmablauf (cont.)

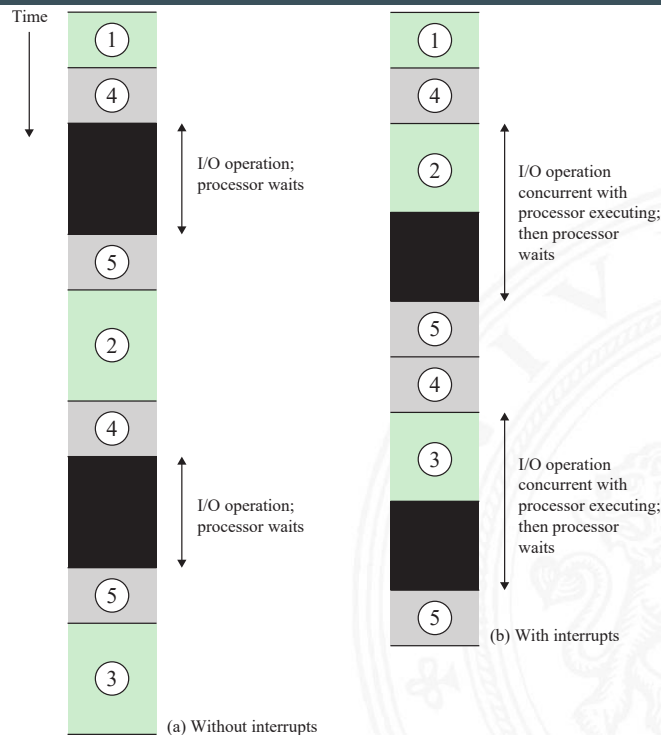
- ▶ kurze I/O Wartezeit





Interrupt: Programmablauf (cont.)

► lange I/O Wartezeit

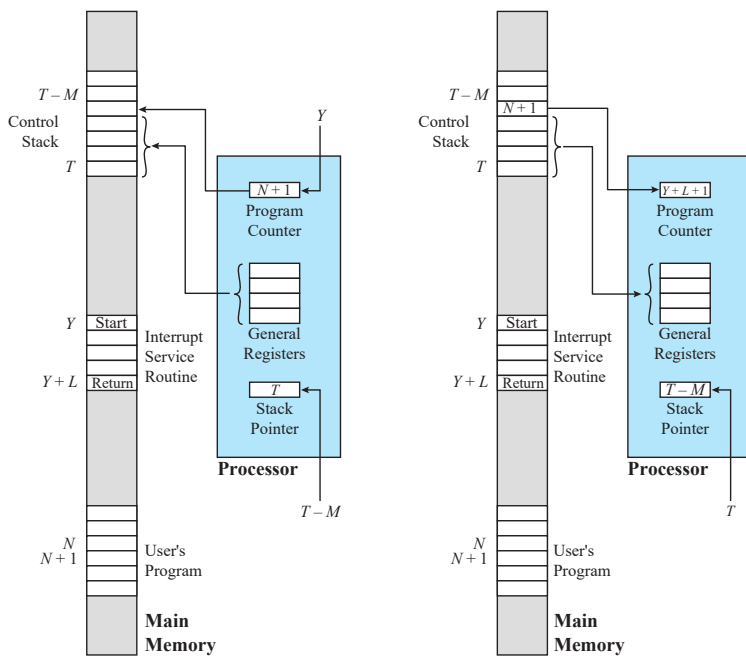


Interruptverarbeitung

1. HW Gerät / Systemhardware liefert Interrupt
2. HW Prozessor beendet aktiven Befehl, wegen Pipelining ggf. mehrere Schritte
3. HW Prozessor bestätigt Interrupt
4. HW Programmstatus (PC, Register, Speicherlimits 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 Status aus 6. wiederherstellen
9. SW Status aus 4. wiederherstellen; PC für Programmfortsetzung initialisieren
+ Wechsel in Standardmodus (*User Mode*)



Interruptverarbeitung (cont.)



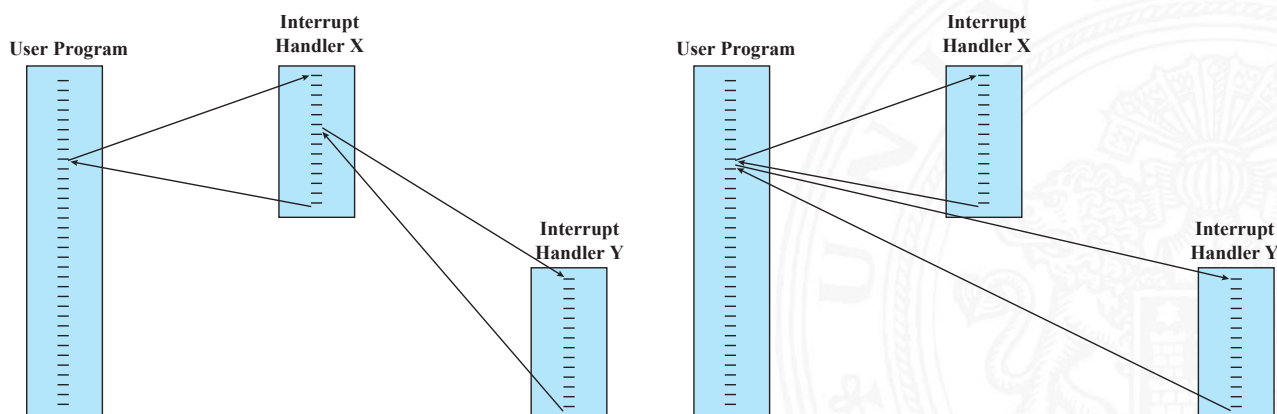
(a) Interrupt occurs after instruction at location N

(b) Return from interrupt



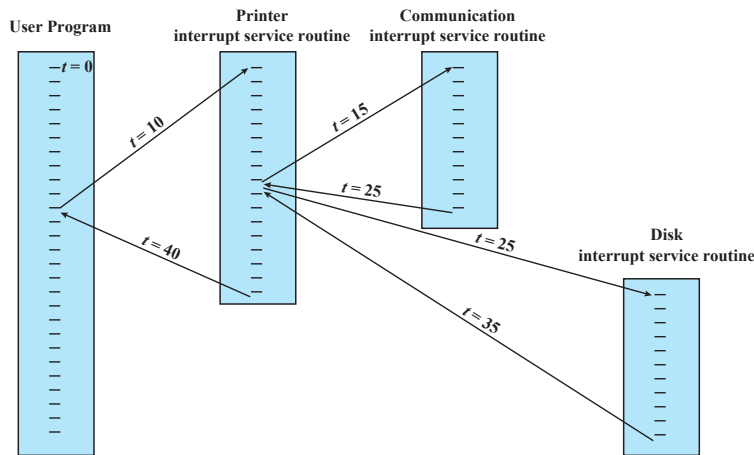
Mehrfache Interrupts

- ▶ während Interruptverarbeitung kommt Interrupt
 1. weitere Interrupts deaktivieren
 2. verschiedene Interruptprioritäten
- ▶ Interrupts können „verloren gehen“, ggf. Zwischenspeichern
- ▶ Schachtelung und/oder sequenzielle Abarbeitung



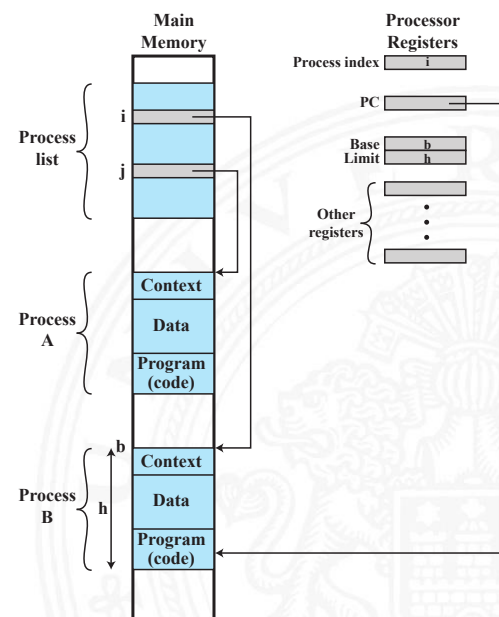


▶ Beispiel: zeitlicher Verlauf



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 einzelnen sequenziellen Ablauf
 - ▶ einem Zustand
 - ▶ zugehörigen Systemressourcen





Fehler bei Prozessabarbeitung

15.3 Betriebssysteme - Prozesse und Threads

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Synchronisationsfehler
 - ▶ Prozess muss warten
 - ▶ Reaktivierung durch externes Ereignis
 - ⇒ Ereignis wird nicht, wird mehrfach ausgelöst
 - ▶ gegenseitiger Ausschluss (*Mutual Exclusion*)
 - ▶ mehrere Prozesse mit gleichzeitigen Zugriff auf Ressource, z.B. Speicher, Datei
 - ⇒ Sperrmechanismen: Semaphore, Mutex, Monitor
 - ▶ nichtdeterministisches Verhalten
 - ▶ mehrere Prozesse/Threads kommunizieren über *Shared Memory*
 - ⇒ transiente Effekte: Programme überschreiben sich Werte
 - ⇒ vom Scheduling durch Betriebssystem abhängig
 - ▶ Deadlocks
 - ⇒ Prozesse warten (zyklisch) aufeinander
- siehe Abschnitt 15.4 *Synchronisation und Kommunikation*

A. Mäder

1131

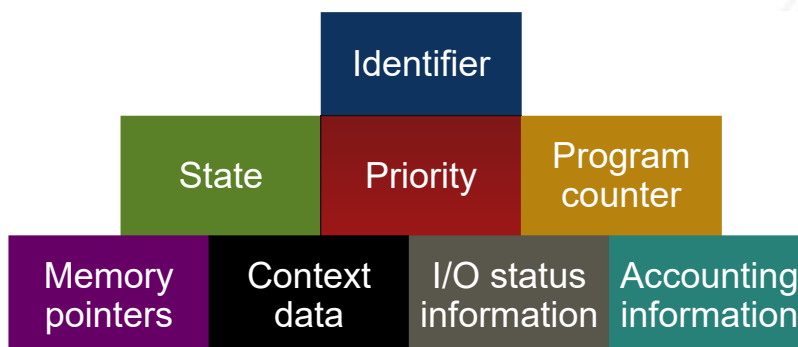


Komponenten eines Prozesses

15.3 Betriebssysteme - Prozesse und Threads

64-040 Rechnerstrukturen und Betriebssysteme

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.



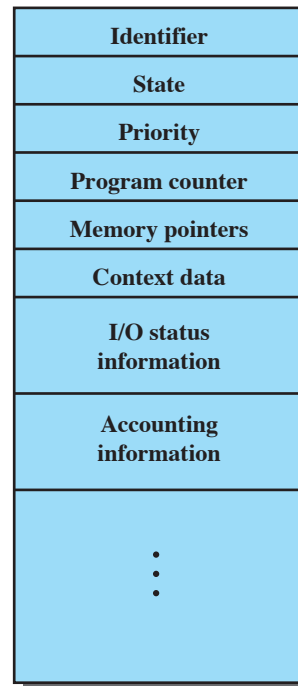
A. Mäder

1132



Komponenten eines Prozesses (cont.)

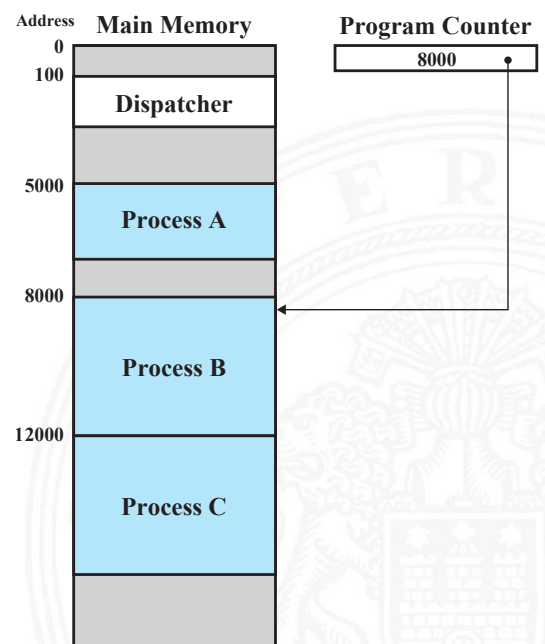
- ▶ Prozesskontrollblock speichert Verwaltungsinformation



Prozessablauf

- ▶ *Trace*: Folge von Instruktionen
 - ▶ für einzelnen Prozess \Rightarrow Laufzeitverhalten
 - ▶ für Prozessor \Rightarrow zeigt Prozesswechsel
- ▶ *Dispatcher*: Betriebssystemkomponente, die Prozessor Prozessen zuordnet

- ▶ Beispiel
 - ▶ Dispatcher
 - ▶ 3 Prozesse: A, B, C





Prozessablauf (cont.)

Trace

- ▶ Dispatcher (im Trace: blau)
- ▶ 3 Prozesse: A, B, C

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	27	12004
2	5001	28	12005
3	5002	-----Timeout	
4	5003	29	100
5	5004	30	101
6	5005	31	102
-----Timeout		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	-----Timeout	
16	8003	41	100
----- I/O Request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	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, z.B.: *htop*)

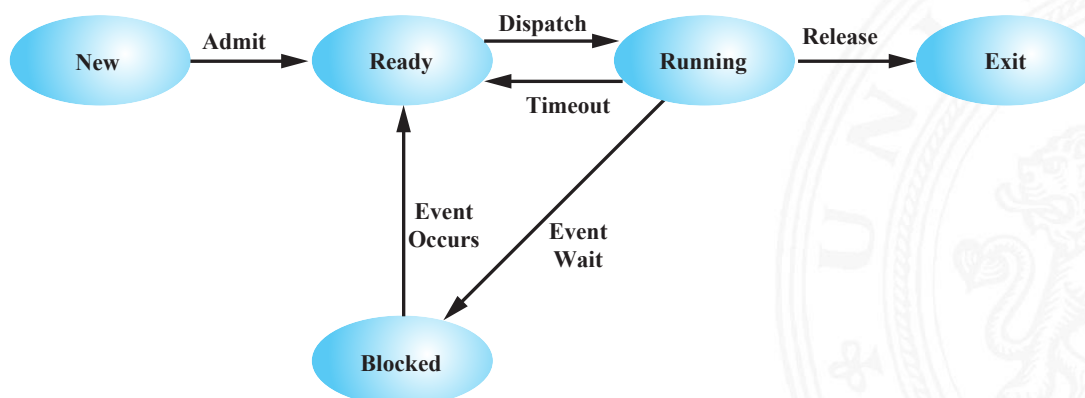


► Prozessende

- 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 Ressourcenzugriff, 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 ⇒ Child-Prozess beenden
- Abbruch durch: Betriebssystem, Operator, Benutzer



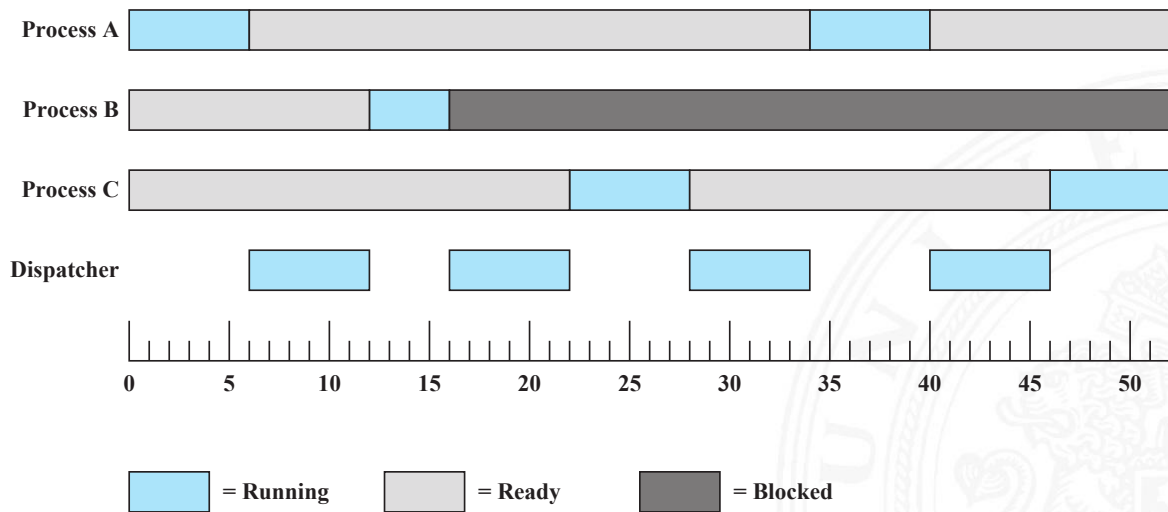
- Prozesse werden dynamisch gestartet und beendet
- –"– warten auf I/O / Systemereignisse
- –"– werden unterbrochen: Time-Sharing Betrieb
- Verwaltung durch Dispatcher
- Zustandsautomat





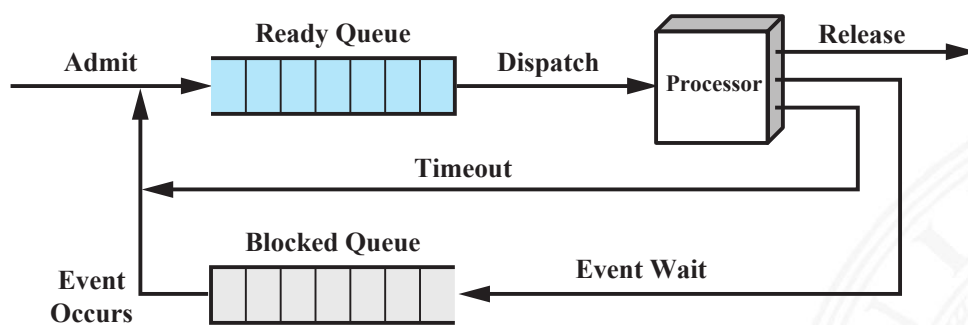
Prozessmodell (cont.)

► Trace



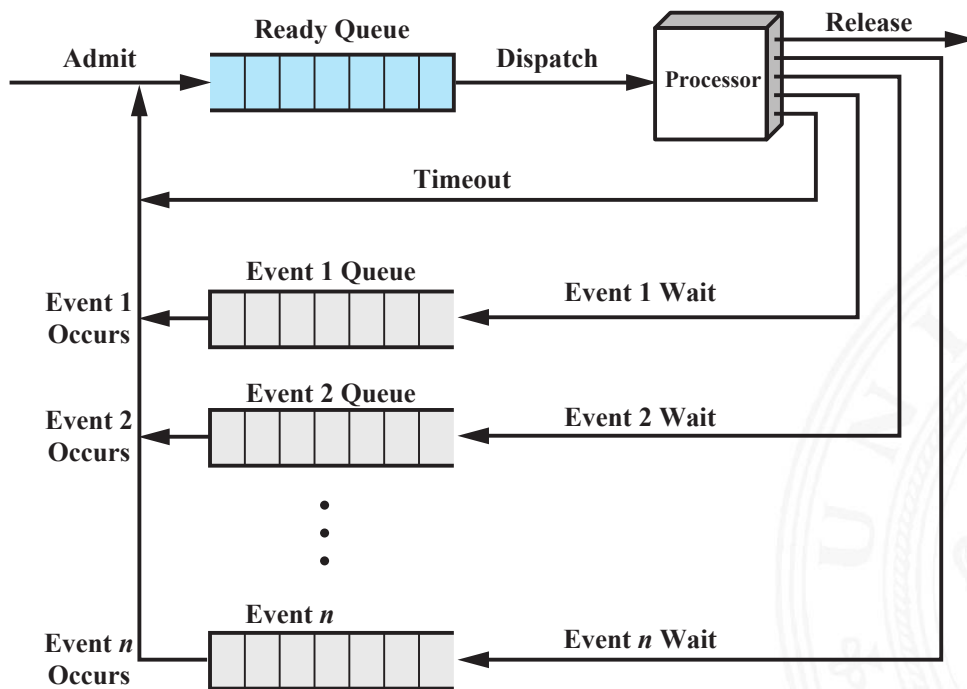
Prozessmodell (cont.)

► Warteschlangen, ggf. mit Trennung nach Signal



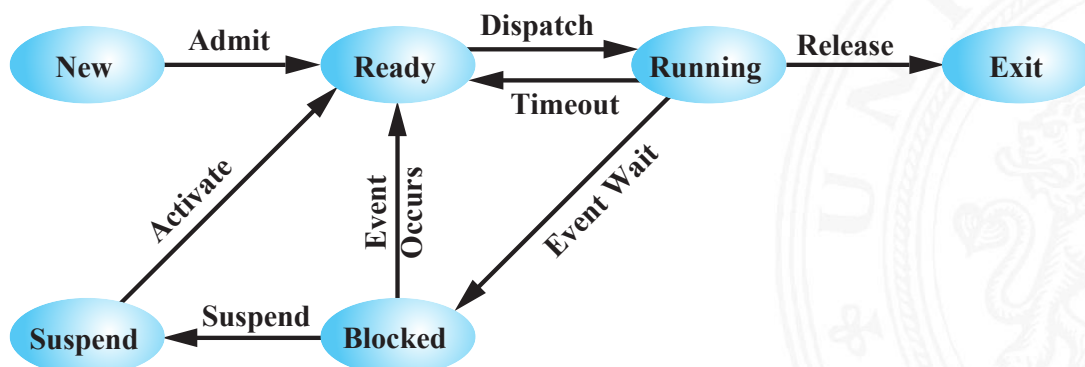


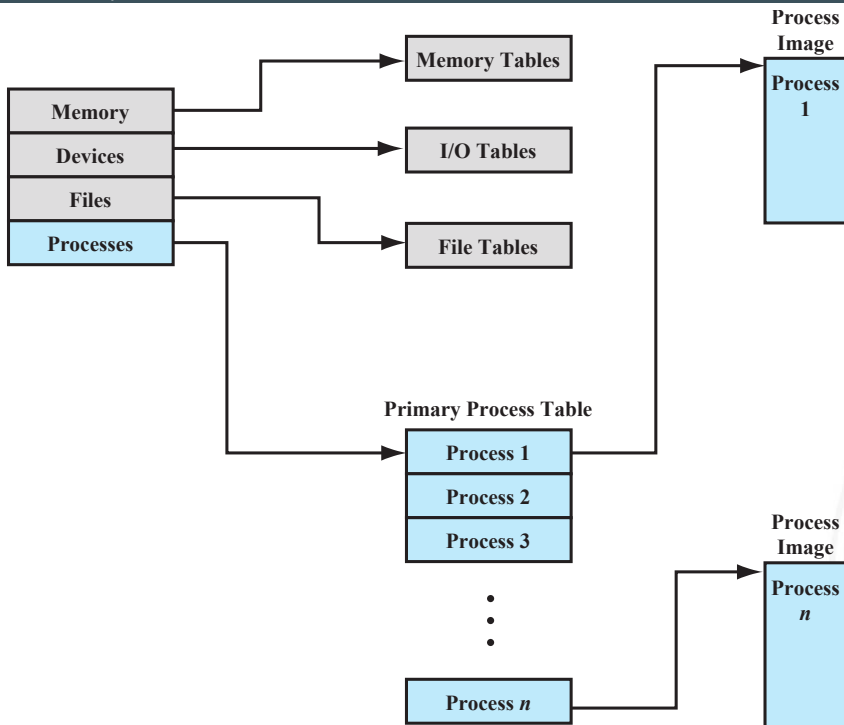
Prozessmodell (cont.)



Prozessunterbrechung

- ▶ *Swapping*: Prozesse (auch teilweise) von RAM auf virtuellen Speicher auslagern
- ▶ Prozessunterbrechung (*suspend*) durch
 - ▶ Swapping: Betriebssystem benötigt Hauptspeicher
 - ▶ Timing: bei periodischer Ausführung ...
 - ▶ Parent Anfrage an Child, z.B.: zur Synchronisation
 - ▶ Unterbrechung durch: Betriebssystem, Operator, Benutzer
- ▶ Erweiterung des Prozessmodells

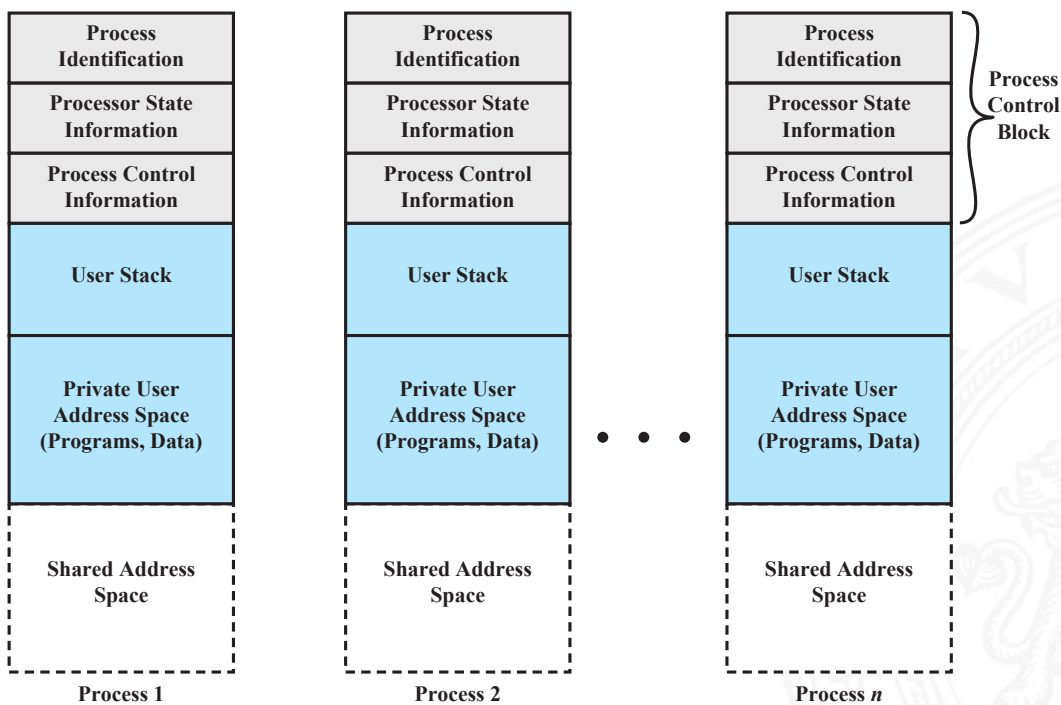


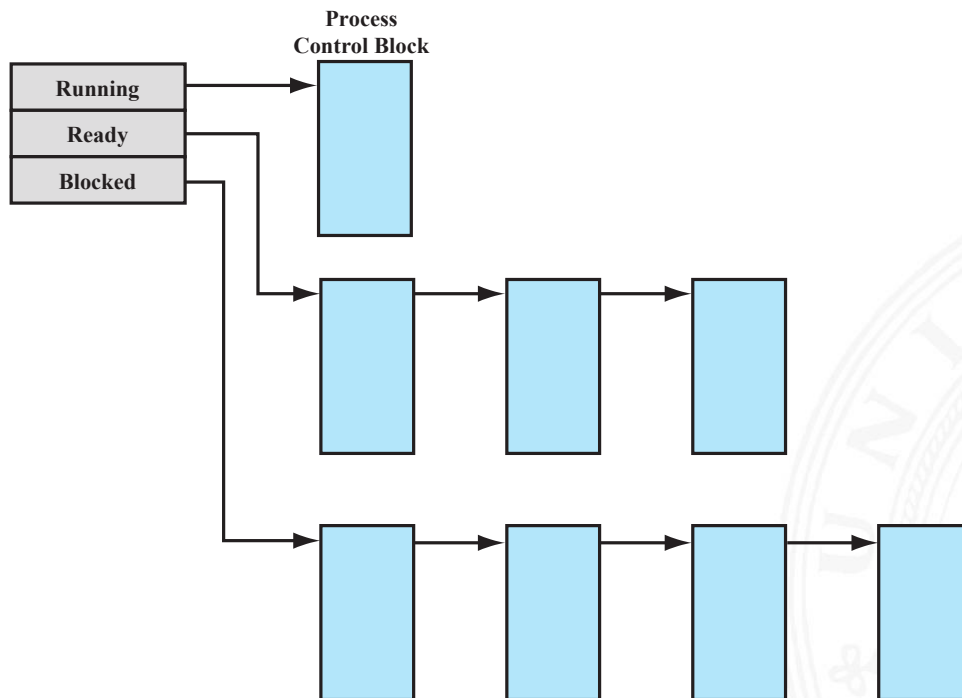


- ▶ Prozesstabellen
- ▶ Speicherverwaltung: *Memory Tables*
 - ▶ Verwaltung von virtuellem Speicher
 - ▶ Zuordnung für Hauptspeicher (RAM)
 - ▶ Zuordnung für sekundären Speicher (HDD, SSD)
 - ▶ Attribute für Speicherblöcke, z.B.: Speicherschutz
- ▶ Ein-/Ausgabeverwaltung: *I/O Tables*
 - ▶ Zuordnung 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.



- ▶ Programmcode
- ▶ Datenstrukturen des Programms: statische Daten und dynamischer Speicher, z.B.: *Heap*
- ▶ *Stack*: Unterprogrammaufrufe und -Datenstrukturen
→ siehe Abschnitt 13.3 *Funktionsaufrufe und Stack*
- ▶ Prozesskontrollblock, siehe Folie 1132 f
 - ▶ 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: geöffnete Dateien ...
 - ▶ ...





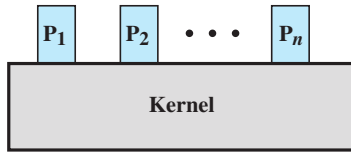
- ▶ Interrupt-Mechanismen
 - ▶ Interrupt: asynchrones, externes Ereignis
 - ▶ Trap: Fehler während der Programmabarbeitung
 - ▶ System-Call: Aufruf einer Betriebssystemfunktion
- ▶ kein „wartender“ Interrupt ⇒ nächsten Befehl holen
- ▶ Interrupt löst Kontextwechsel aus
 - ▶ Programmzähler mit Interrupt Handler initialisieren
 - ▶ Wechsel *User Mode* ⇒ *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



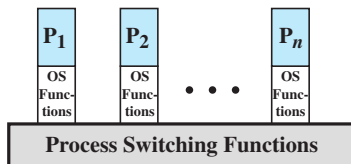
Kontextwechsel (cont.)

► Realisierungen

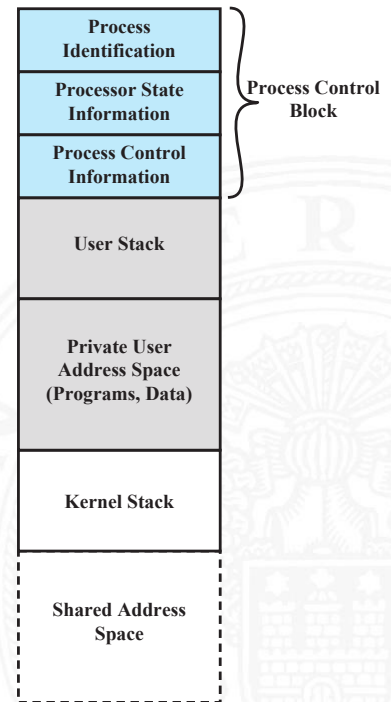
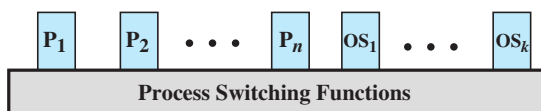
- getrennter Betriebssystemkernel



- innerhalb des Benutzerprogramms



- eigene Services/Prozesse (*Mikrokernel*)



Prozesse und Threads

► Thread / Lightweight Process

- Betriebssystem: Zuordnung zu Prozessor (CPU)
- ⇒ Programmablauf (*Scheduling, Dispatching*)

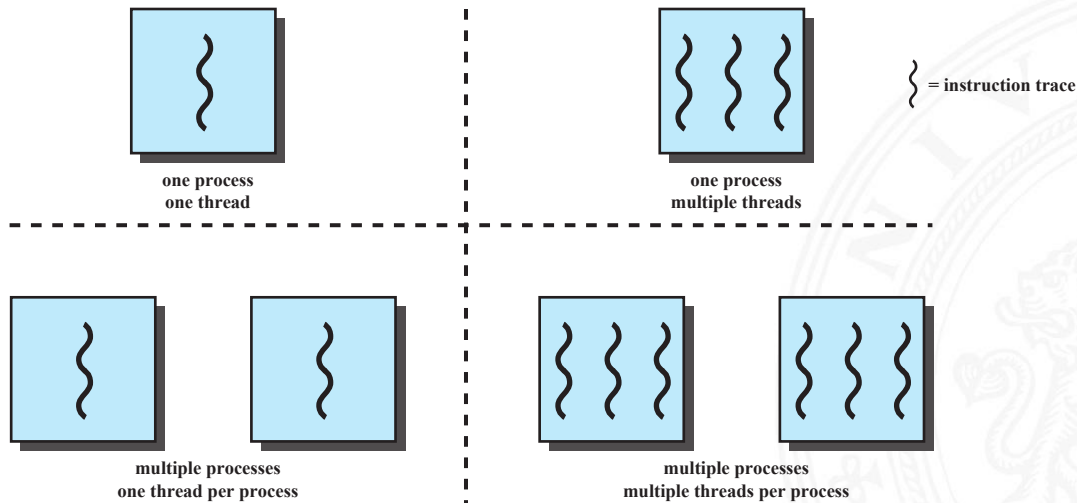
► Prozess

- Betriebssystem: Zuordnung zu Ressourcen (Speicher, Dateien, I/O ...)
- ⇒ gesamter Kontrollblock



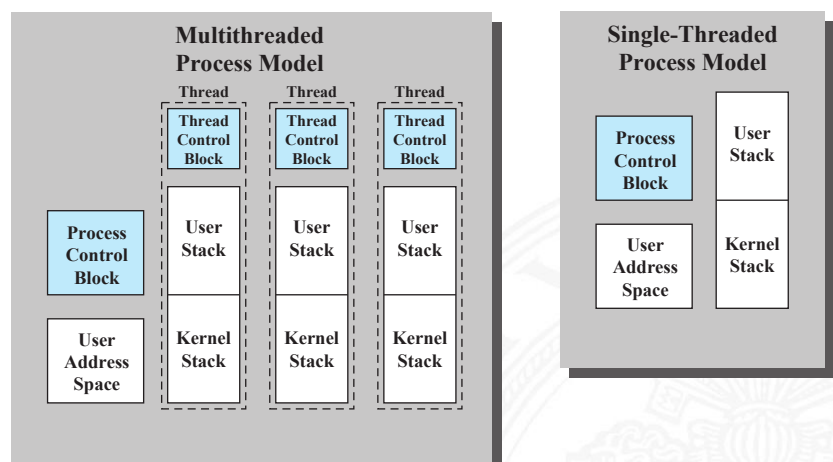
▶ Multithreading

- ▶ mehrere parallele Ausführungen innerhalb eines Prozesses
- ▶ von Programmiersprache und Betriebssystem abhängig



▶ je Thread

- ▶ eigener Zustand ($\hat{=}$ Prozesszustand)
- ▶ eigener Kontext
- ▶ eigener Stack
- ▶ ggf. statischen Speicher



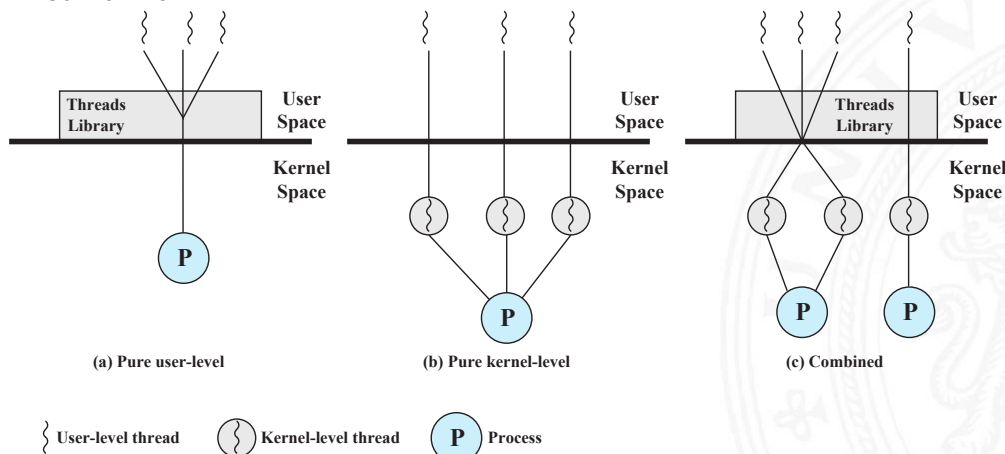
- ▶ Zugriff auf alle Datenstrukturen und Ressourcen des Prozesses
- ▶ Speicher und Ressourcen mit allen anderen Threads geteilt



- ▶ 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



- ▶ 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
- ⇒ abwechselndes und überlapptes Rechnen
- ⇒ Timing / Abarbeitungsgeschwindigkeit nicht vorhersehbar
 - ▶ Aktivitäten anderer Prozesse oder der Benutzer
 - ▶ Interrupts
 - ▶ Scheduling durch Betriebssystem
- ▶ Begriffe
 - ▶ **atomare Operation**: 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**: ist ein Prozess in seiner Critical Section, dann kann kein zweiter Prozess in eine Critical Section eintreten, die die gleichen Ressourcen nutzt
- ▶ **Race Condition**: mehrere Threads/Prozesse lesen und schreiben Daten; das Ergebnis ist von deren zeitlicher Reihenfolge abhängig
- ▶ **Starvation / „verhungern“**: ein lauffähiger Prozess könnte (weiter-) arbeiten, wird aber nie bedient
- ▶ Kommunikationsmechanismen zwischen Prozessen/Threads
 - ▶ gemeinsamer Speicher (*Shared Memory*) ⇒ Mutual Exclusion
 - ▶ Nachrichtenaustausch



Mutual Exclusion / gegenseitiger Ausschluss

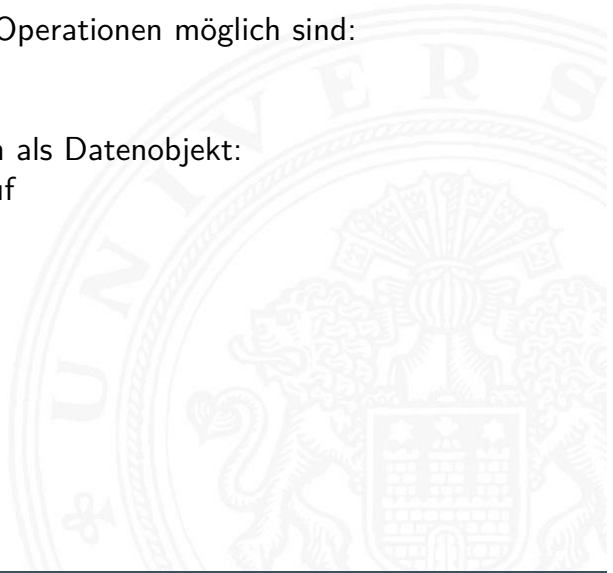
- ▶ notwendig, um Race Conditions zu vermeiden
- ▶ durch Sicherung von Critical Sections
- ▶ mögliche Probleme: Deadlock, Starvation

- ▶ Uniprozessor: keine Interrupts in Critical Section
- ▶ atomare Hardwareoperationen („*compare & swap*“)
 - + gilt für: Uni-/Multiprozessor, beliebige Anzahl 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 Schnittstellen
 - ▶ verschiedene Varianten, in Bibliotheken implementiert (Dekker-Algorithmus; E. W. Dijkstra; Peterson-Algorithmus) nicht trivial !!!
 - ▶ **Semaphor**: Integer Variable, für die drei atomare Operationen möglich sind: initialisieren, increment, decrement
 - ▶ **Mutex / binärer Semaphor**: Werte 0 und 1
 - ▶ **Monitor**: kapselt Variablen und Zugriffsprozeduren als Datenobjekt: immer nur ein Prozess hat exklusiven Zugriff darauf
einfacher handhabbar als Mutex oder Semaphor

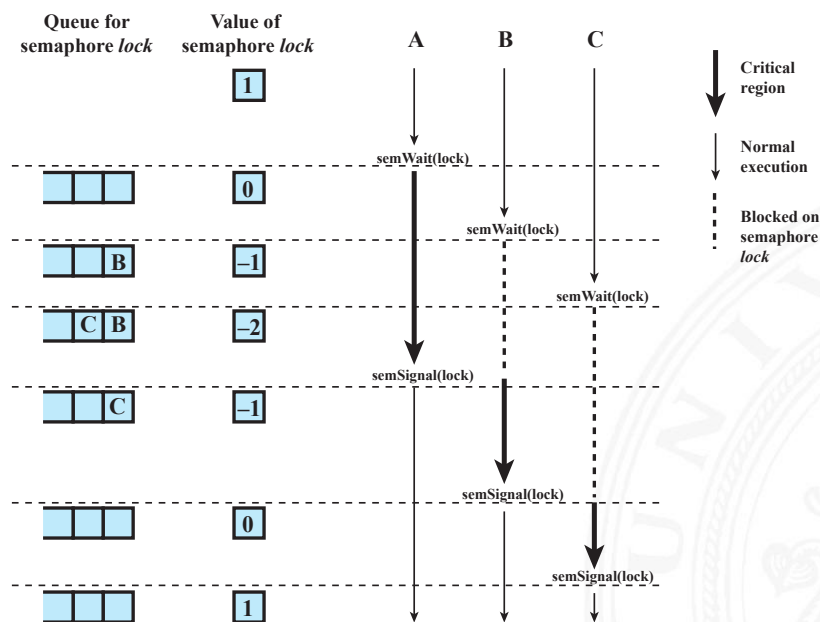




- ▶ Integer Variable, initialisiert mit Anz. gleichzeitiger Zugriffe
- ▶ atomare Operationen: init., decr., incr.
 - semWait** $\hat{=}$ Beginn der Critical Section
decrement + aufrufender Prozess muss ggf. warten
 - semSignal** $\hat{=}$ Ende der Critical Section
increment + ein wartender Prozess kann starten
- ▶ starke Semaphor: am längsten wartender Prozess wird gestartet
⇒ Queue für wartende Prozesse
- ▶ schwache Semaphor: beliebiger, wartender Prozess wird gewählt



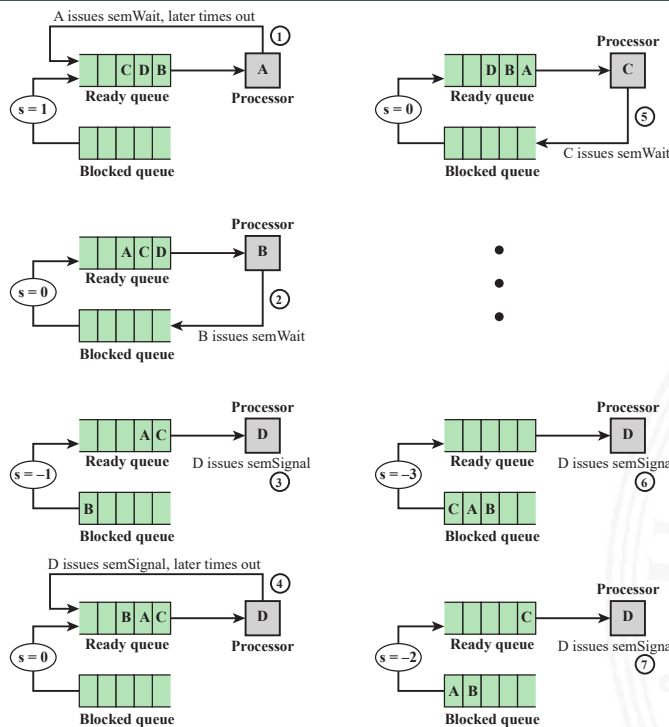
▶ Beispiele



Note that normal execution can proceed in parallel but that critical regions are serialized.

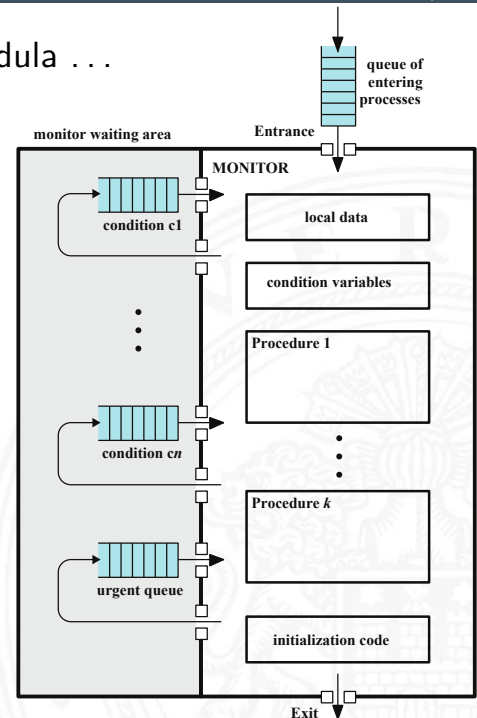


Semaphor (cont.)



Monitor

- ▶ Programmiersprachen: Concurrent Pascal, Ada, Modula ...
- ▶ Java: synchronized, keine Bedingungsvariablen
- ▶ nur ein Prozess darf im Monitor sein
- ▶ einfacher zu handhaben als Semaphor, gleiche Funktionalität
- ▶ Synchronisation
 - ▶ Bedingungsvariablen (*condition variables*)
 - ▶ Funktionen: wait, signal
- ▶ Warteschlangen
 - ▶ extern für Eintritt in Monitor
 - ▶ für jede Bedingungsvariable





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` \Rightarrow *Rendezvous*
 - `nicht block. send + block. receive`
 - `nicht block. send + nicht block. receive`
 - ▶ direkte Adressierung (s.o.) / indirekte Adressierung
 \Rightarrow 1:1, 1:n, m:1, m:n Beziehungen

```
process P is
...
send(Q, A)
...
```

```
process Q is
...
receive(P, A)
...
```



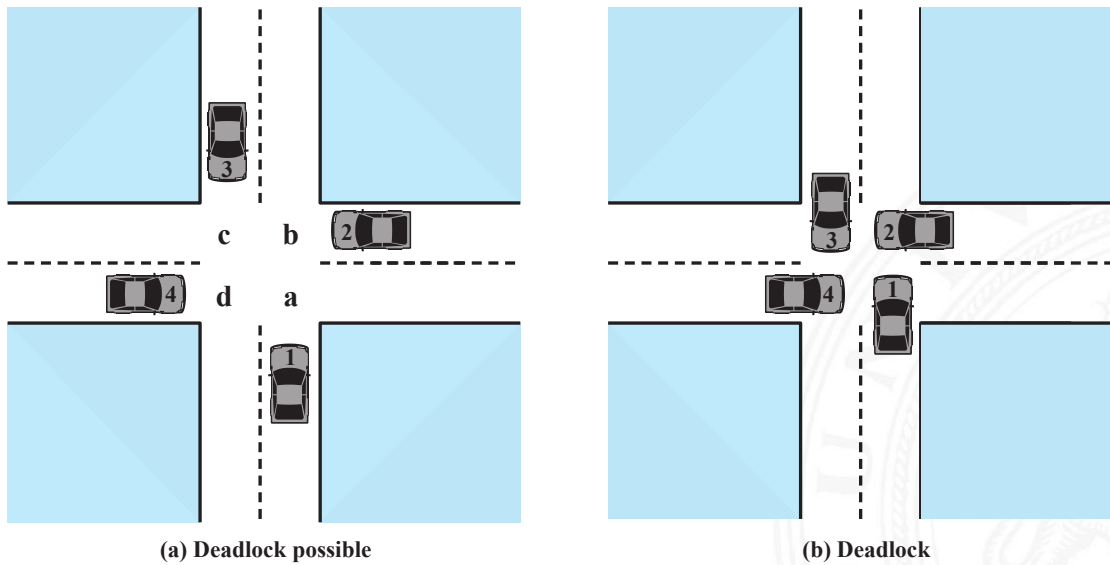
Deadlock

- ▶ Dauerhaftes Blockieren mehrerer Prozesse, die
 - ▶ um Ressourcen konkurrieren
 - ▶ miteinander kommunizieren
- ▶ Deadlock, wenn jeder Prozess blockiert auf etwas wartet, was immer nur einer der anderen blockierten Prozess anstoßen kann
- ▶ **im Allgemeinen: keine effiziente Lösung**



Deadlock (cont.)

► Beispiel: „rechts vor links“



► jedes Fahrzeug braucht 2 Ressourcen 1: a,b 2: b,c 3: c,d 4: d,a



Deadlock (cont.)

► Beispiel: zwei Programme, zwei Mutexe

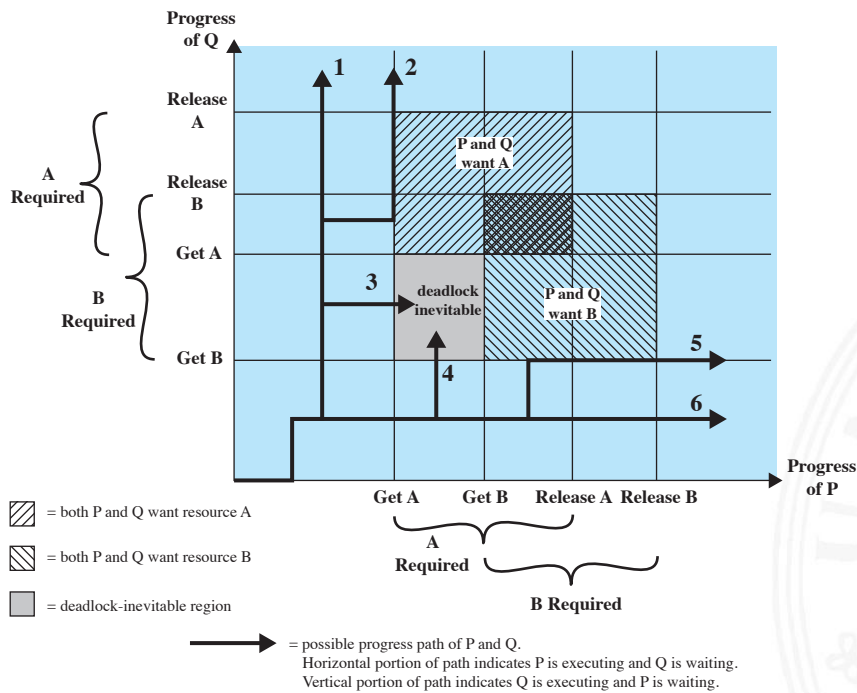
```
process P is
  ...
  get(A)
  ...
  get(B)
  ...
  release(A)
  ...
  release(B)
  ...
```

```
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

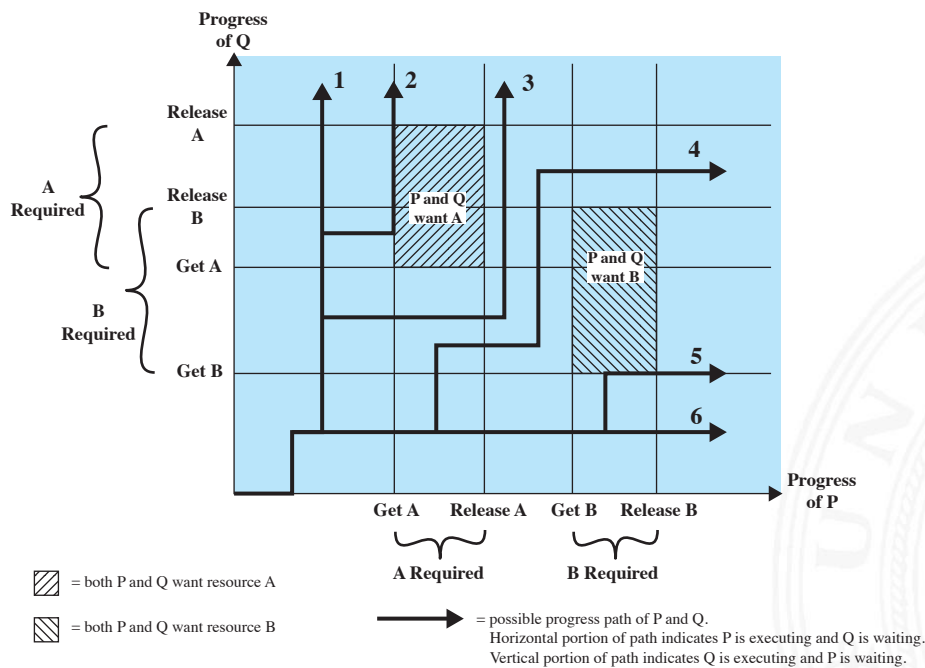
```

...
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, Codebereiche (Critical Sections)
- ▶ verbrauchbare (*consumable*) Ressource
 - ▶ wird erzeugt und bei Nutzung verbraucht
 - ▶ Interrupts, Signale
Nachrichten etc. (in FIFOs)

Betriebssystem
Programmierer

Betriebssystem
Programmierer

- ▶ Beispiel: wiederverwendbare Ressource = 200 KiB Speicher

process P is

```

...
malloc(70 KiB)
...
malloc(80 KiB)
...

```

process Q is

```

...
malloc(80 KiB)
...
malloc(60 KiB)
...

```



- ▶ Beispiel: verbrauchbare Ressource = Nachrichten, receive blockierend

```
process Q is
  ...
  receive(P, M1)
  ...
  send(P, M2)
  ...
```

```
process P is
  ...
  receive(Q, M3)
  ...
  send(Q, M4)
  ...
```



1. Mutual Exclusion
 - ▶ ohne Mutual Exclusion kein Deadlock
⇒ 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
- ⇒ 1. bis 3. notwendige Bedingungen
+ Circular Wait (zur Laufzeit) = Deadlock



Maßnahmen gegen Deadlock

15.4 Betriebssysteme - Synchronisation und Kommunikation

64-040 Rechnerstrukturen und Betriebssysteme

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, wenn Ressource 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
⇒ 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“

A. Mäder

1173



Maßnahmen gegen Deadlock (cont.)

15.4 Betriebssysteme - Synchronisation und Kommunikation

64-040 Rechnerstrukturen und Betriebssysteme

3. Deadlock Erkennung

- ▶ Periodischer Test auf Deadlock und ggf. (partielles) Rücksetzen
 - + 1. und 2. schränken Prozesse ein; gegenteiliger Ansatz: alle Ressourcenzugriffe 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

A. Mäder

1174



Maßnahmen gegen Deadlock (cont.)

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none"> •Works well for processes that perform a single burst of activity •No preemption necessary 	<ul style="list-style-type: none"> •Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	<ul style="list-style-type: none"> •Convenient when applied to resources whose state can be saved and restored easily 	<ul style="list-style-type: none"> •Preempts more often than necessary
		Resource ordering	<ul style="list-style-type: none"> •Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design 	<ul style="list-style-type: none"> •Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none"> •No preemption necessary 	<ul style="list-style-type: none"> •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	<ul style="list-style-type: none"> •Never delays process initiation •Facilitates online handling 	<ul style="list-style-type: none"> •Inherent preemption losses

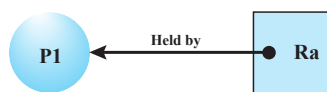


Maßnahmen gegen Deadlock (cont.)

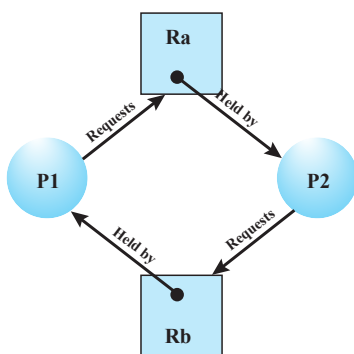
► Graph zum Ressourcenbesitz



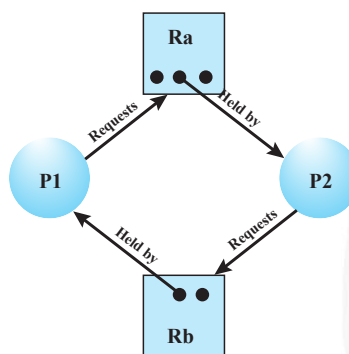
(a) Resource is requested



(b) Resource is held



(c) Circular wait



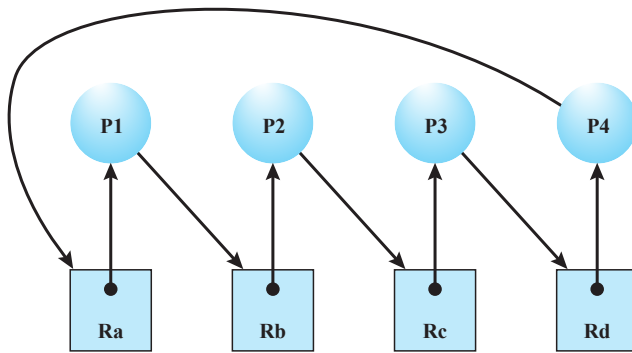
(d) No deadlock



Maßnahmen gegen Deadlock (cont.)

15.4 Betriebssysteme - Synchronisation und Kommunikation

64-040 Rechnerstrukturen und Betriebssysteme



Kreuzung: „rechts vor links“

- ▶ effiziente Darstellung als Matrizen: Ressourcenanfragen und -besitz
- ▶ Standardverfahren in Algorithmen zu Deadlock Vermeidung oder Erkennung



Scheduling und Ressourcenmanagement

15.5 Betriebssysteme - Scheduling

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Hauptfunktionalität von Betriebssystemen: Ressourcenmanagement
- ▶ wichtig dabei
Effizienz
Antwortverhalten (*Responsiveness*)
Fairness
- ⇒ 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 → Abschnitt 15.6
 - ▶ Auswirkungen auf Multiprogramming: Prozesse nicht lauffähig, wenn nicht im Speicher



Scheduling und Ressourcenmanagement (cont.)

15.5 Betriebssysteme - Scheduling

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ **Short-term:** Welcher Prozess wird durch CPU ausgeführt?
 - ▶ Dispatcher: wird häufig aufgerufen
 - ▶ Start durch: Interrupts, System-Calls, Signale (Semaphor, Mutex)
 - ▶ quantitative Kriterien
 - für Benutzer: Antwortverhalten (*Responsiveness*)
 - für System: Prozessornutzung, Ressourcenauslastung, Job-Durchsatz
 - ▶ qualitative Kriterien: Fairness, Deadlockfrei, keine Starvation, Vorhersagbarkeit, Echtzeitfähigkeit etc.
- ▶ **I/O Scheduler:** Welche I/O-Anfrage geht an Gerät?
 - ▶ getrennte Geräteklassen
 - ▶ gerätespezifische Algorithmen

A. Mäder

1179

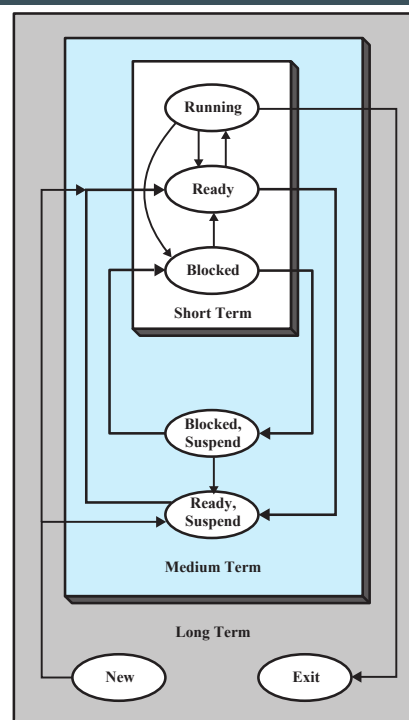


Scheduling und Ressourcenmanagement (cont.)

15.5 Betriebssysteme - Scheduling

64-040 Rechnerstrukturen und Betriebssysteme

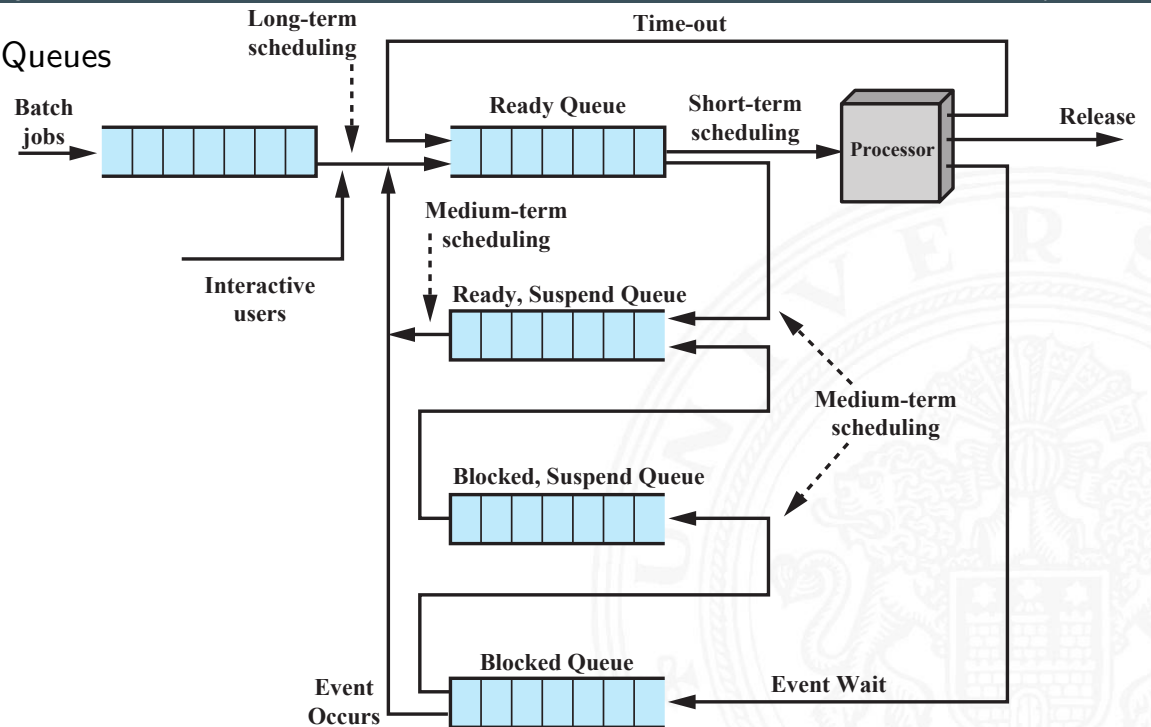
- ▶ verschiedene Zustände im Prozessmodell (vergl. Folie 1138)



A. Mäder

1180

► Scheduling Queues

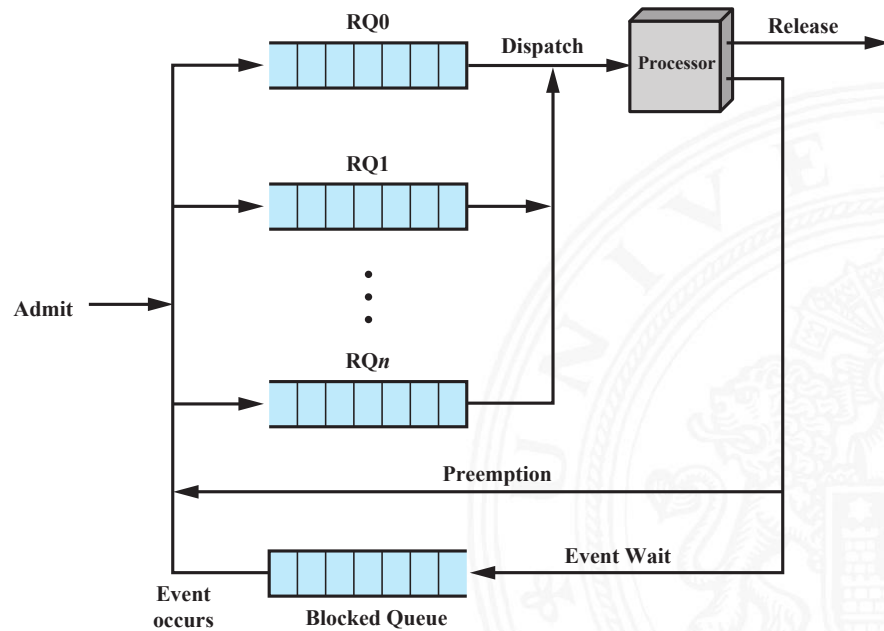


Short-term Scheduling

- 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



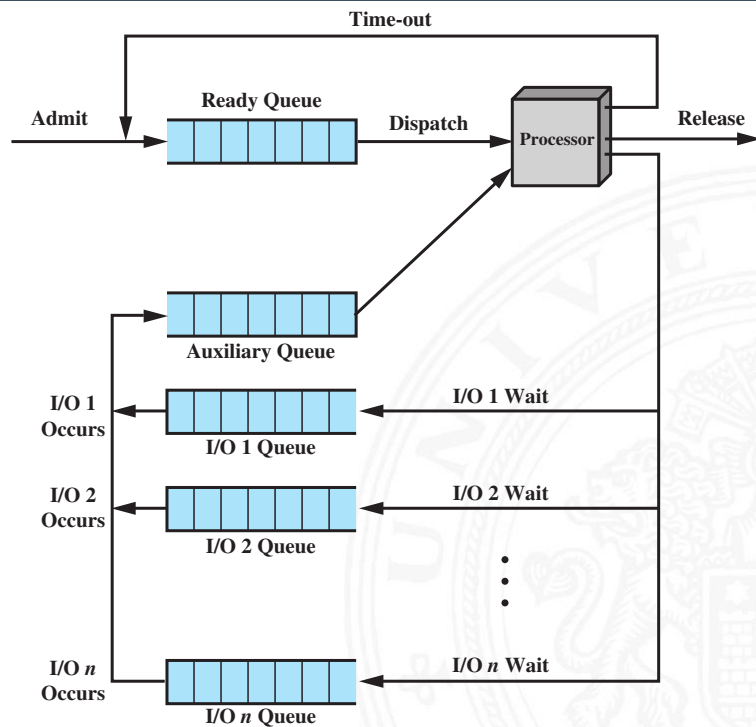
▶ Short-term Queues mit Prioritäten



- ▶ **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)



▶ Round Robin Queues



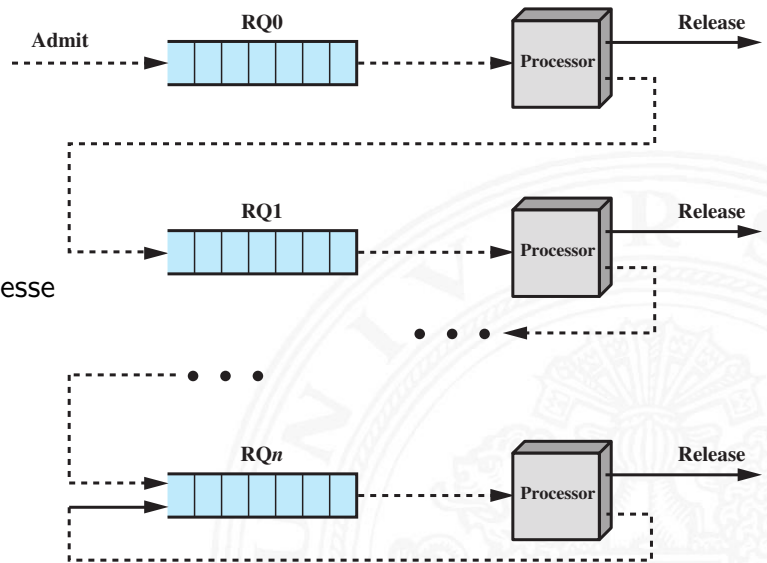
- ▶ **Shortest Process Next (SPN)**
 - ▶ Non-Preemptive
 - ▶ Prozess mit kürzester Ausführungszeit startet
 - ▶ Ausführungszeit s schätzen?
 - ▶ Starvation für lang laufende Prozesse möglich
 - ▶ Interaktive Prozesse: Durchschnittsbildung der letzten Aktivitäten, ggf. „exponentielles Altern“ (= Wichtung 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
 - ▶ in hochpriorisierter Queue starten
 - ▶ sukzessiver Abstieg in weniger priorisierte Queues
- ▶ relativer Vorzug kurz laufender Prozesse



Scheduling Algorithmen (cont.)

Scheduling Vergleichstabelle

- ▶ ... viele weitere Algorithmen
 - ▶ Gruppierung von Prozessen (*Process Groups*)
 - ▶ periodische Tasks
 - ▶ Echtzeitsysteme
- ▶ Prozesse haben eine Deadline!

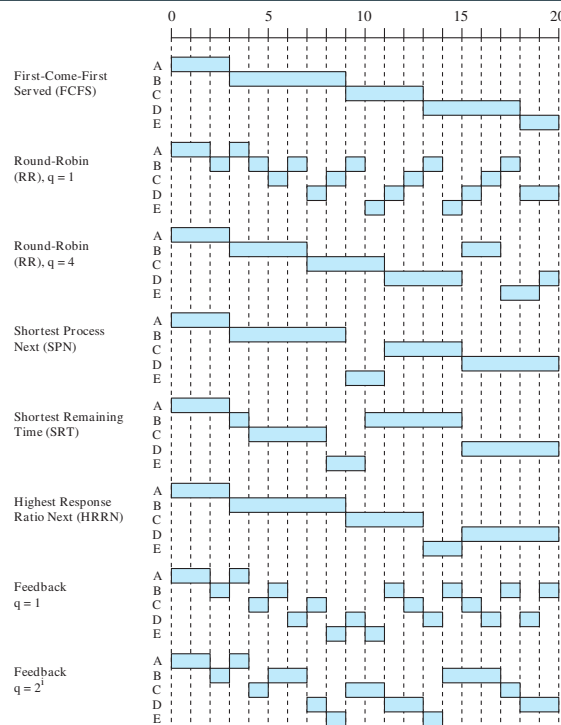
	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s - e]		(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Through-Put	Not emphasized	May be low if quantum is too 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?

- Rechnerystem: Smartphone, Tablet, 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

⇒ Modellierung über Warteschlangentheorie, stochastische Prozesse

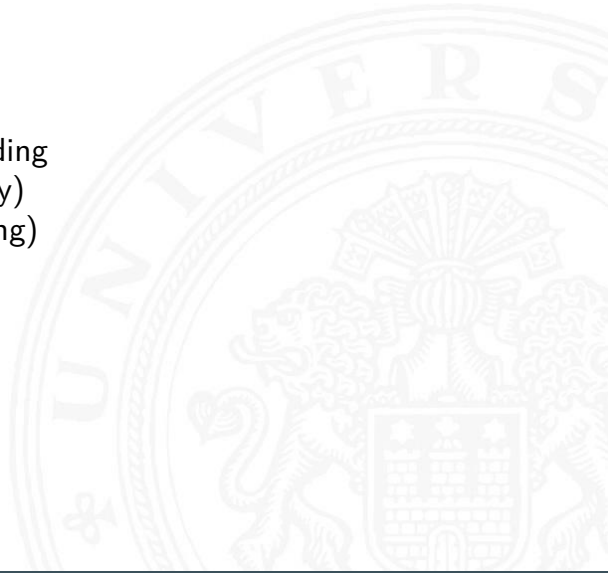


Multiprozessor Scheduling

15.5 Betriebssysteme - Scheduling

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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



A. Mäder

1191



Multiprozessor Scheduling (cont.)

15.5 Betriebssysteme - Scheduling

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Zuordnung von Prozessen zu Prozessoren
 - ▶ dynamisch: Menge von Prozessen → 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



A. Mäder

1192



Aufgaben des Betriebssystems

15.6 Betriebssysteme - Speicherverwaltung

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ Trennung der Prozesse voneinander
- ▶ Verwaltung von dynamischem Speicher
- ▶ Unterstützung modularer Programme
- ▶ Schutz: Integrität der Daten
- ▶ Schutz: Durchsetzung von Zugriffsrechten
- ▶ Realisierung von Langzeitspeicher (Dateisysteme)



A. Mäder

1193



Virtueller Speicher

15.6 Betriebssysteme - Speicherverwaltung

64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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
- ⇒ Adressen im Code werden zu **virtuellen Adressen**:
logische Adresse + Adressübersetzung
- ▶ Adressübersetzung entspricht Funktion
 - ▶ meist als Tabelle realisiert



A. Mäder

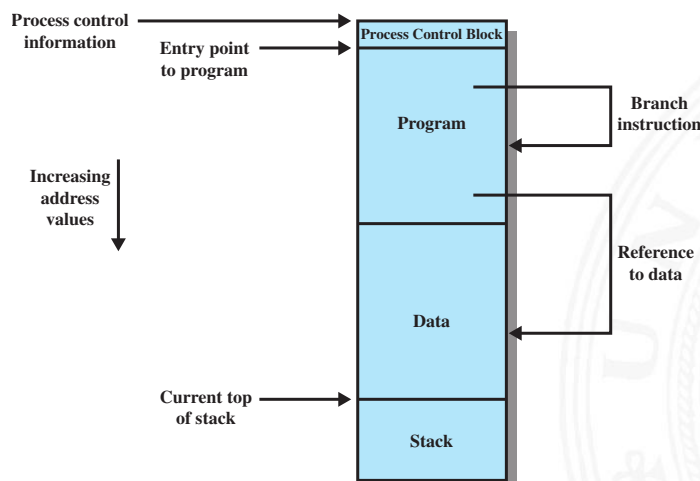
1194



- ▶ **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
 - ⇒ Paging
- ▶ **Segment**
 - ▶ Block variabler Größe im sekundären Speicher, kann temporär
 - ▶ in passenden Bereich im Hauptspeicher kopiert werden
 - ⇒ Segmentierung
 - ▶ in Seiten unterteilt werden, die jeweils kopiert werden
 - ⇒ Segmentierung + Paging



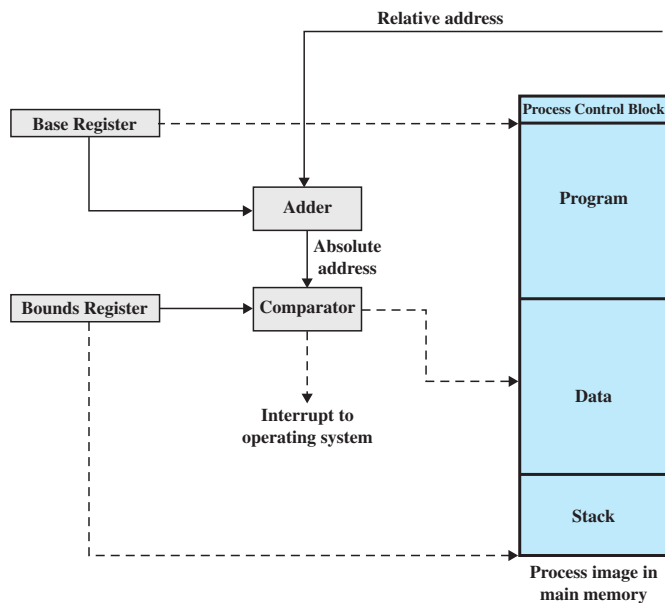
- ▶ **Adressumsetzung / Relocation**
 - ▶ **logische Adressen:** Adressen in (Assembler-) Programm
 - ▶ **relative Adressen:** relativ zu Bezug (Basisadresse), i.d.R. logische Adressen
 - ▶ **physikalische / absolute Adressen:** Adressen des Hauptspeichers





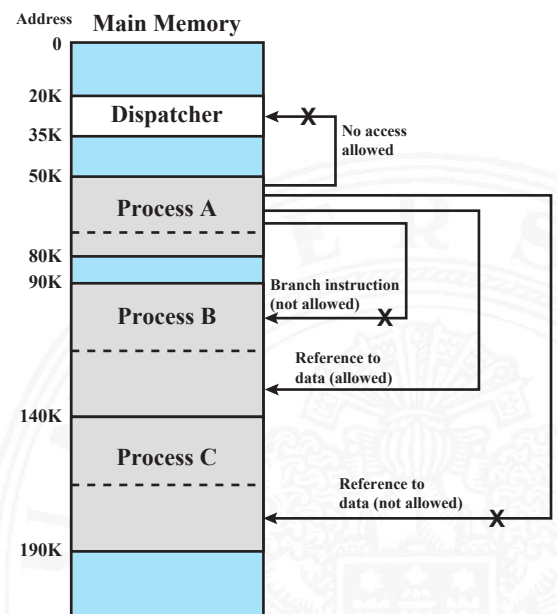
Memory Management (cont.)

- ▶ Swapping: Prozess auslagern, u.U. an anderer Stelle in Hauptspeicher fortsetzen
- ▶ Abbildung: logische → physikalische Adressen



Memory Management (cont.)

- ▶ Zugriffsschutz
 - ▶ Relocation verwaltet auch Zugriffsrechte auf Speicherbereiche
 - ▶ bei Adressumrechnung über Segment- und Seiten-Tabellen
- ▶ gemeinsam genutzte Code- und Datenbereiche
 - ▶ Relocation ermöglicht die Einblendung von Speicher 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
- ▶ nicht alle Segmente/Seiten des Prozesses müssen während der Ausführung gleichzeitig in Hauptspeicher sein
 - ▶ **Resident Set:** Adressbereiche (Text, Data) in Hauptspeicher
 - ▶ **Working Set:** im Programm gerade genutzt (Lokalität)
- ▶ beim Programmlauf
 1. Interrupt, wenn Adresse des Prozesses nicht in Hauptspeicher
 2. Prozess wechselt in „blocked“
 3. Datentransfer von sekundärem Speicher (HDD, SSD) in Hauptspeicher durch DMA
Direct Memory Access: Hardware (z.B.: MMU) arbeitet im Hintergrund
 4. Dispatcher lässt anderer Prozess rechnen
 5. Interrupt, wenn DMA-Datentransfer fertig
 6. Prozess wechselt in „ready“



Memory Management (cont.)

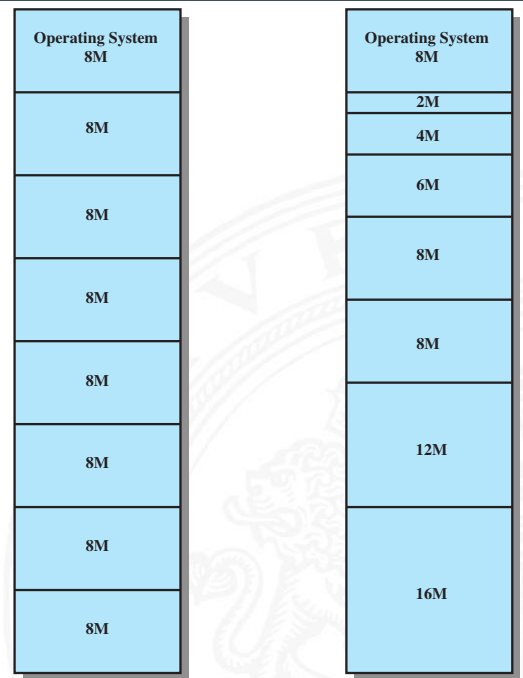
- + kleinerer Adressraum je Prozess
 - ⇒ mehr Prozesse im System
 - ⇒ bessere CPU-Auslastung
- + Prozesse können größer sein als gesamter Hauptspeicher
- + für Programmierer transparent, von Betriebssystem und HW verwaltet

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 den Partitionen
- ⇒ schlechte Speicherausnutzung
- ⇒ obsolet



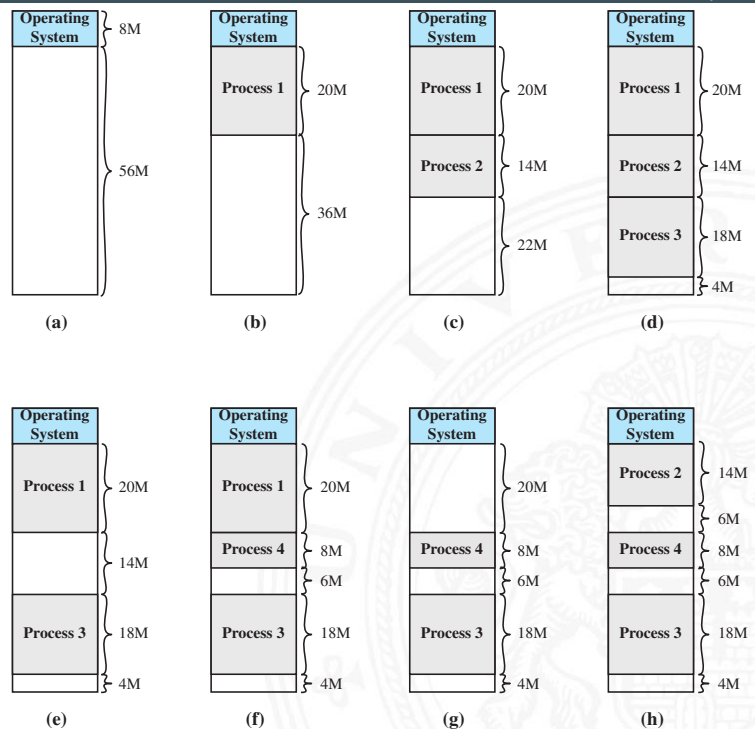
(a) Equal-size partitions

(b) Unequal-size partitions



Partitionierung (cont.)

- ▶ dynamische Partitionierung



(a)

(b)

(c)

(d)

(e)

(f)

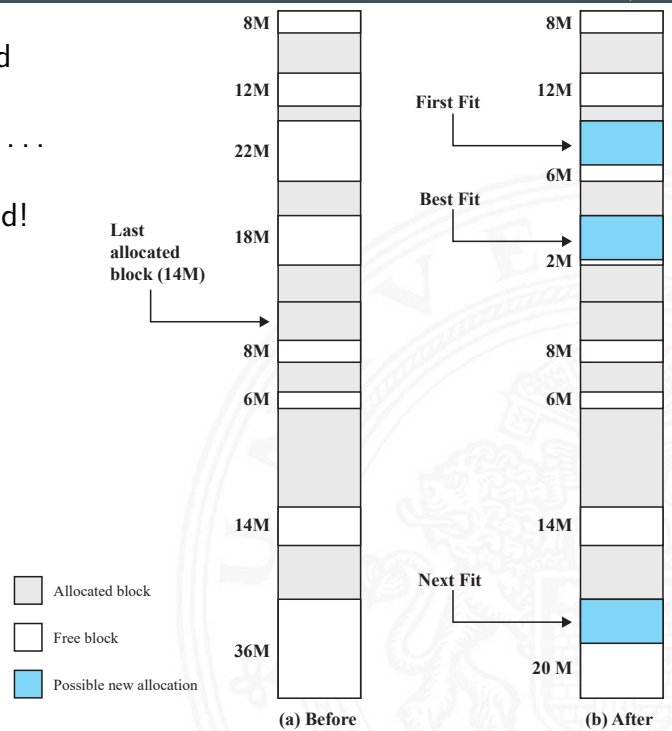
(g)

(h)



Partitionierung (cont.)

- ▶ externe Fragmentierung durch Ein- und Auslagern von Prozessen
- ▶ Platzierung: Best-fit, First-fit, Next-fit ...
- Kompaktierung notwendig, Zeitaufwand!
- ⇒ obsolet



Segmentierung

Virtual Address



Segment Table Entry



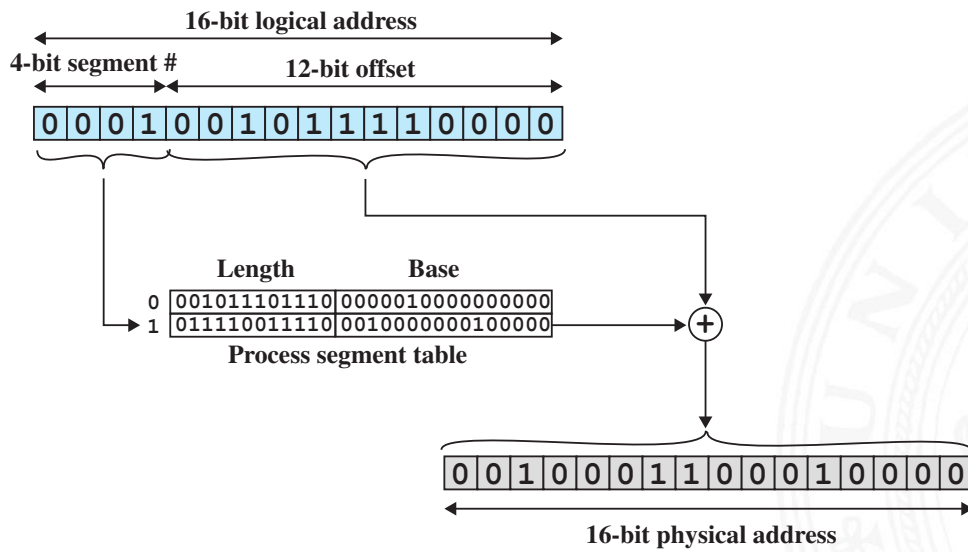
Present bit
Modified bit

- ▶ „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



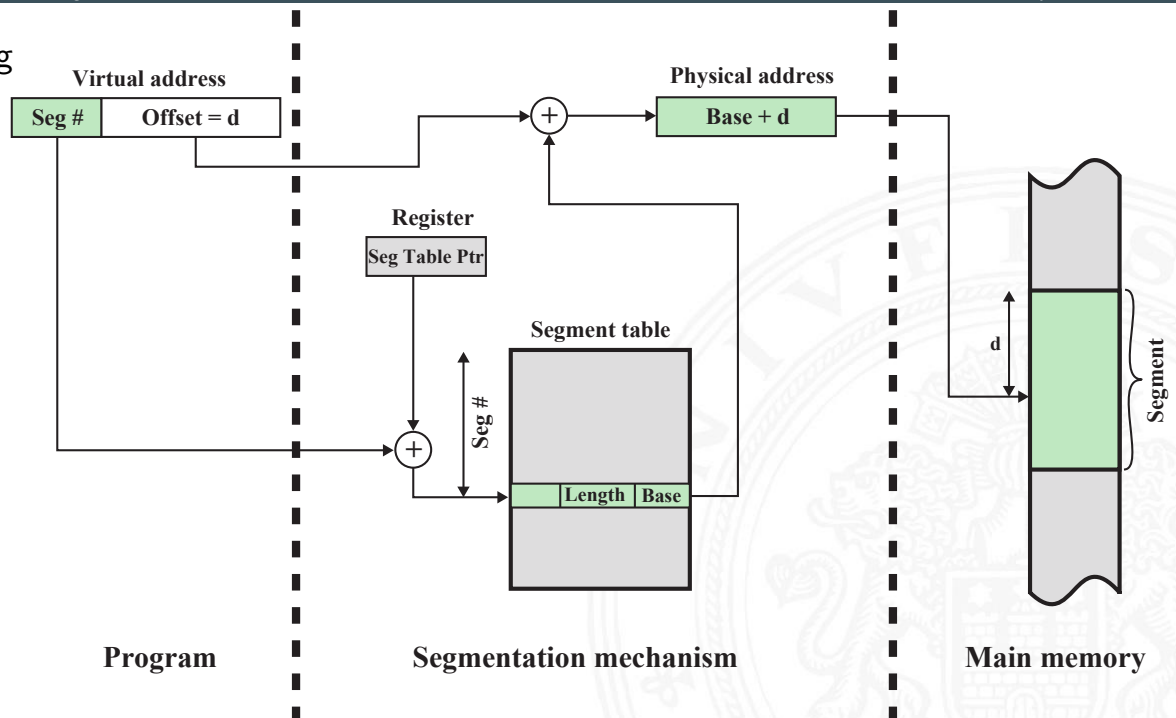
Segmentierung (cont.)

- ▶ Segmenttabelle $\langle segNr \rangle \rightarrow \langle Basisadresse \rangle + \langle L\ddot{a}nge \rangle$
 - ▶ wird f \ddot{u} r jeden Prozess angelegt



Segmentierung (cont.)

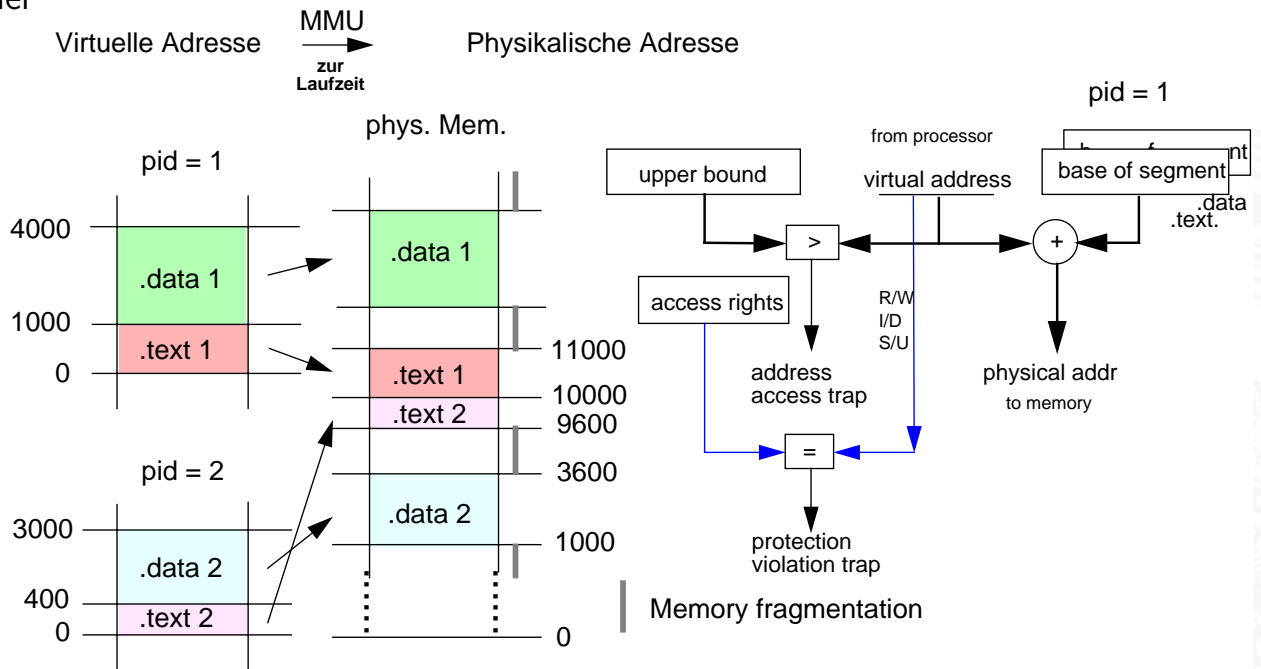
- ▶ Adressierung





Segmentierung (cont.)

▶ Beispiel

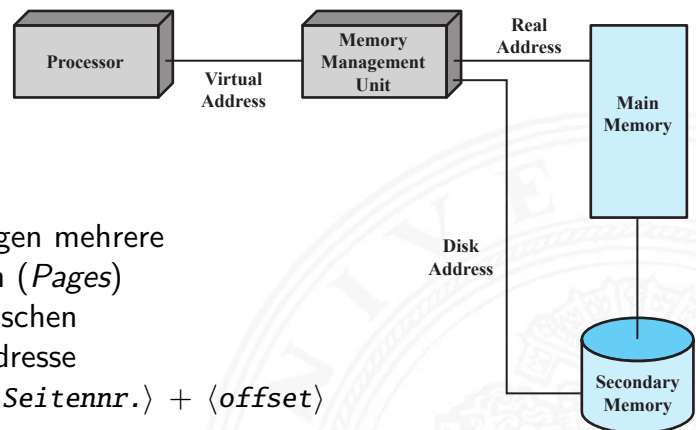


Paging

Virtual Address



Page Table Entry



▶ Unterteilung des Programms

- ▶ Prozesse (Instruktionen+Daten) belegen mehrere Speicherblöcke fester Größe ⇒ Seiten (*Pages*)
- ▶ dynamische Abbildung (Tabellen) zwischen virtueller und realer, physikalischer Adresse
- ▶ Speicherzugriff: $\langle virt. Adresse \rangle = \langle Seitennr. \rangle + \langle offset \rangle$

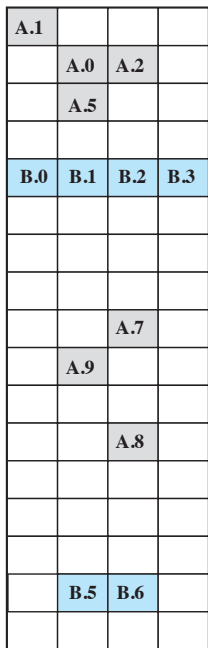
▶ Seite kann

1. an beliebiger Stelle Hauptspeicher stehen
2. auf sekundären Speicher (HDD, SSD) ausgelagert sein
 - ▶ Verwaltung durch *Memory Management Unit* (MMU)

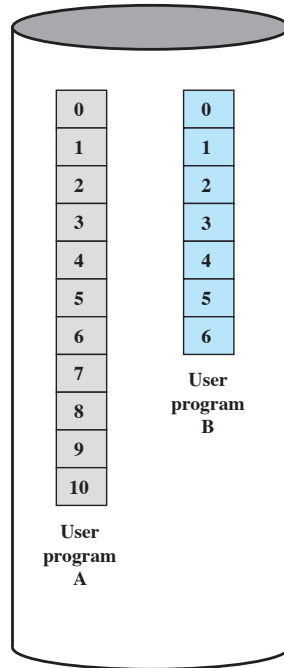
▶ für Programmierer transparent



Paging (cont.)



Main Memory

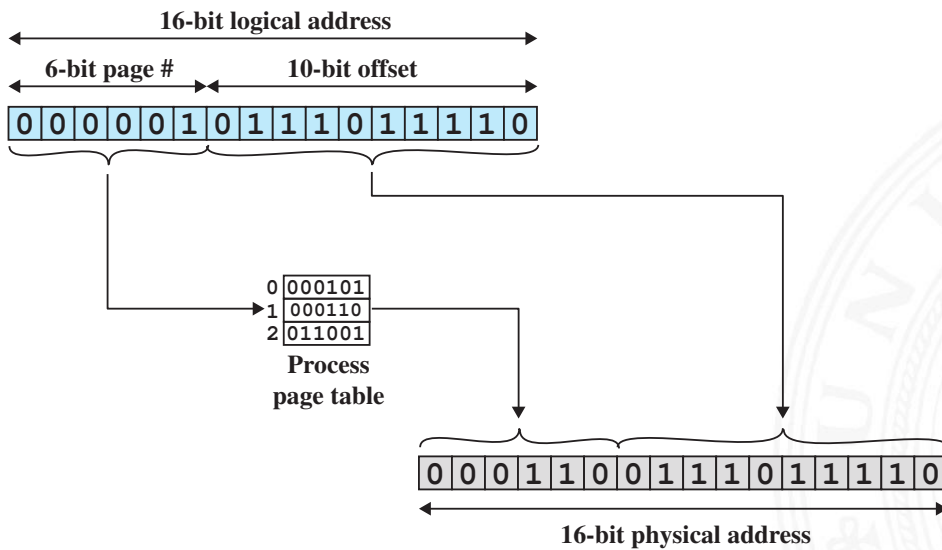


Disk



Paging (cont.)

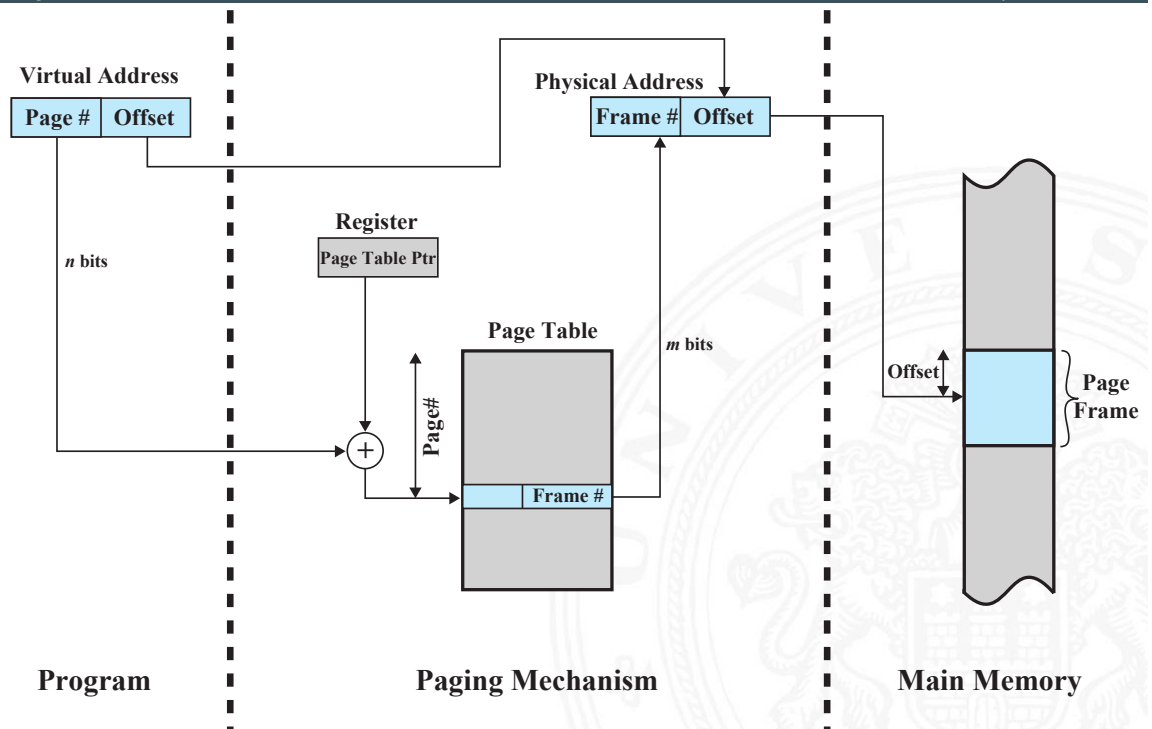
- ▶ Seitentabelle / Page Table $\langle pageNr \rangle \rightarrow \langle frameAddr \rangle$
 - ▶ wird für jeden Prozess angelegt





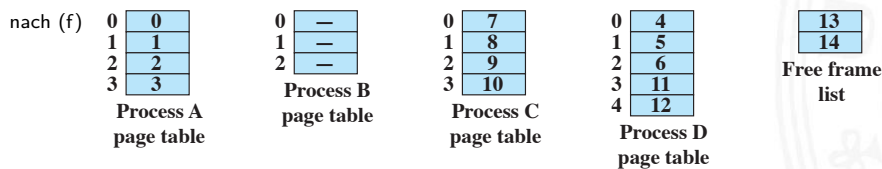
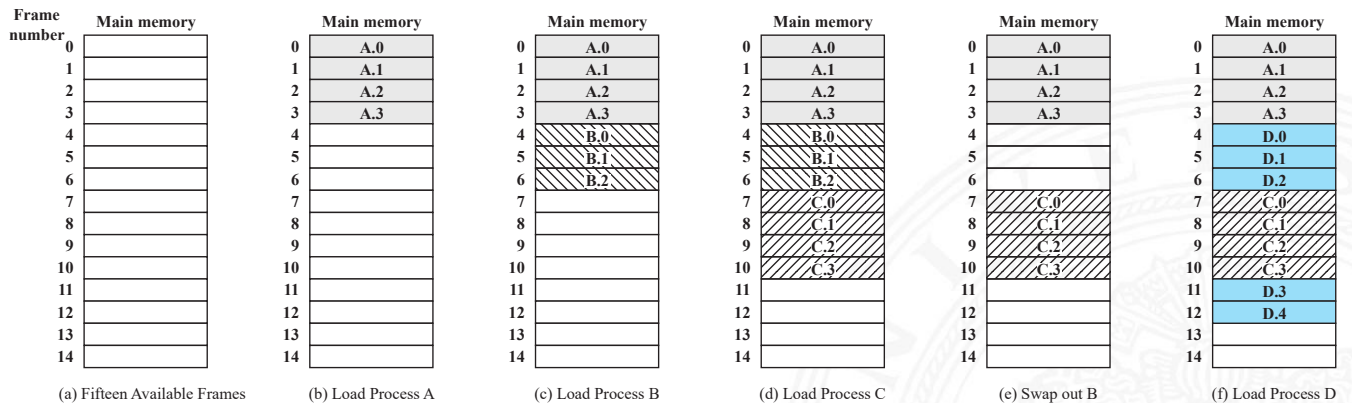
Paging (cont.)

► Adressierung



Paging (cont.)

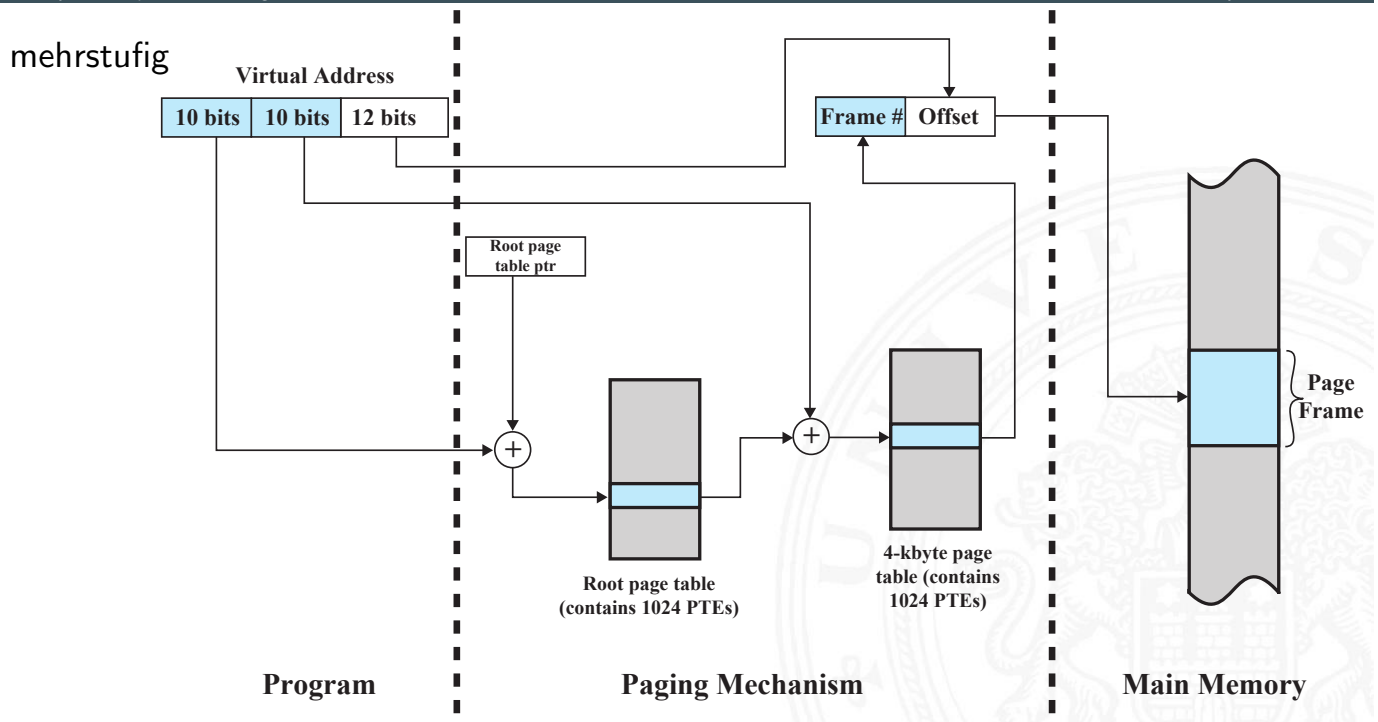
► Beispiel





- ▶ 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

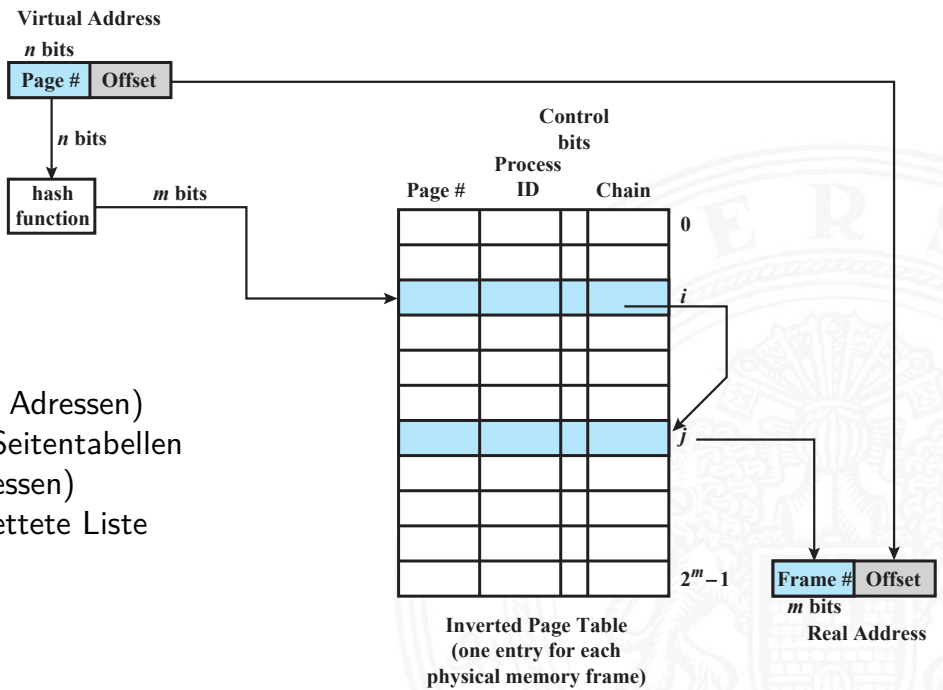
- ▶ mehrstufige Übersetzung
 - ▶ Tabelle wird zu groß \Rightarrow mehrstufige Seitentabellen
 - mehrfacher Zugriff langsam





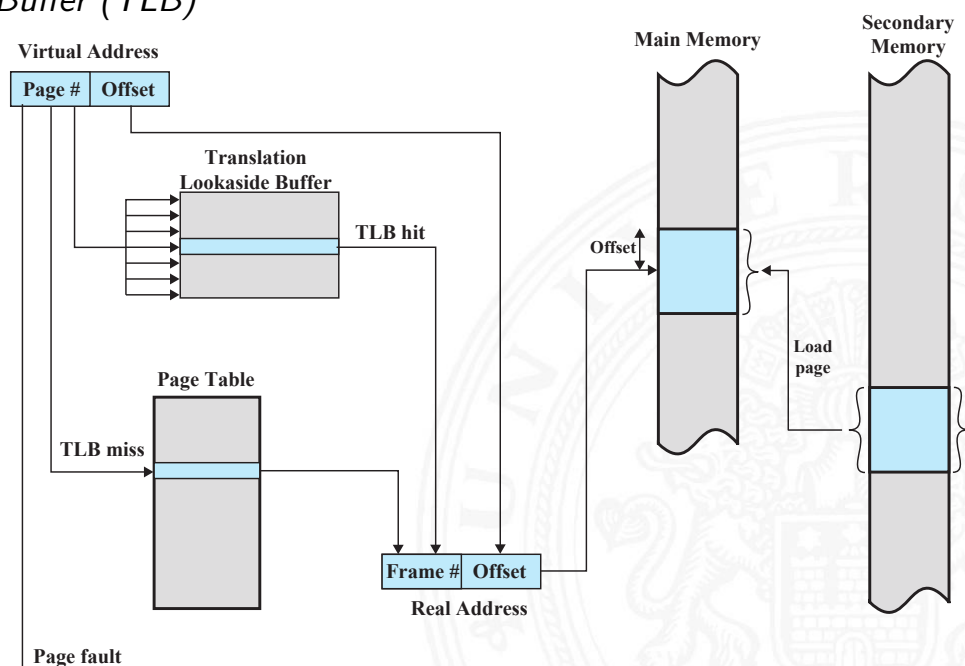
▶ Inverted Page Table

- ▶ eine Tabelle (für alle physikalischen Adressen)
- ▶ statt mehrerer großer Seitentabellen (für alle virtuellen Adressen)
- ▶ Hash-Funktion + verkettete Liste effiziente Suche!



▶ Translation Lookaside Buffer (TLB)

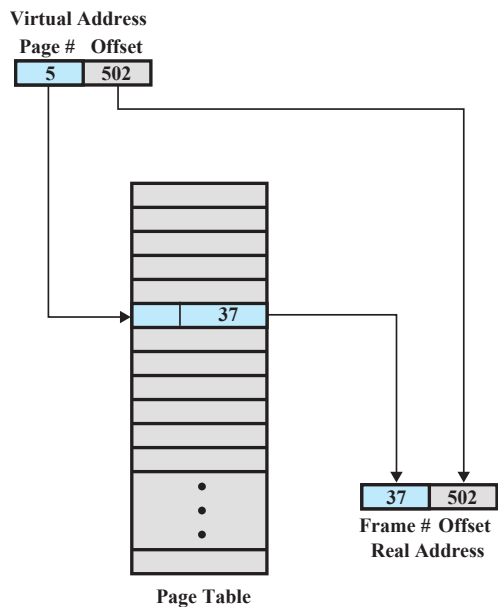
- ▶ voll-assozi. Cache
- ▶ schneller Zugriff





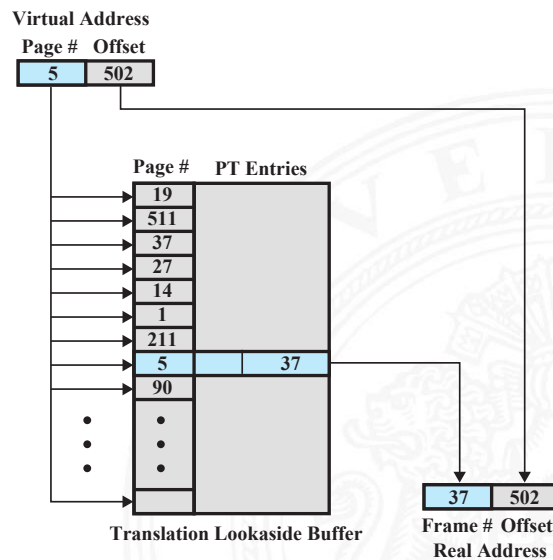
Paging (cont.)

▶ Seitentabelle



(a) Direct mapping

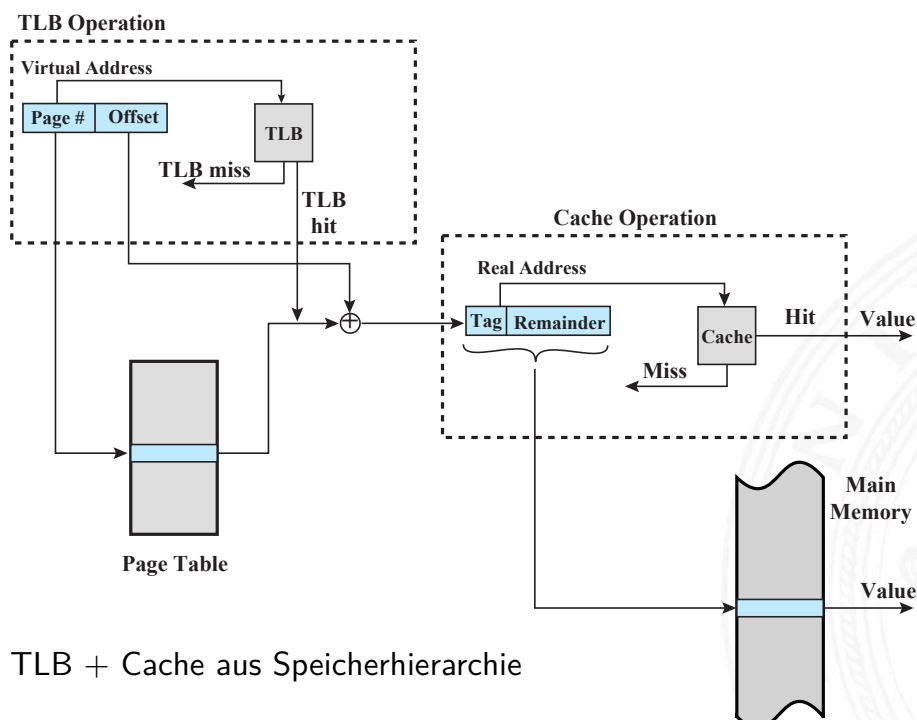
voll-assoziativer TLB Cache



(b) Associative mapping



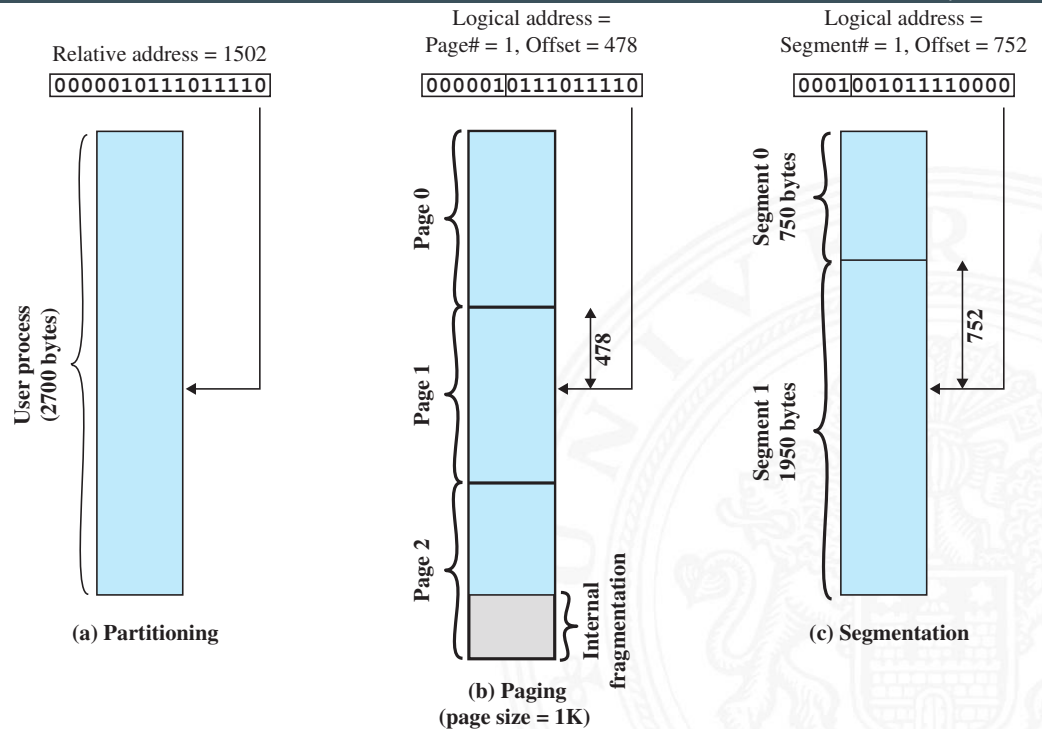
Paging (cont.)



▶ TLB + Cache aus Speicherhierarchie



► Übersicht Adressierung



Virtual Address

Segment Number	Page Number	Offset
----------------	-------------	--------

Segment Table Entry

Control Bits	Length	Segment Base
--------------	--------	--------------

Page Table Entry

P	Other Control Bits	Frame Number
---	--------------------	--------------

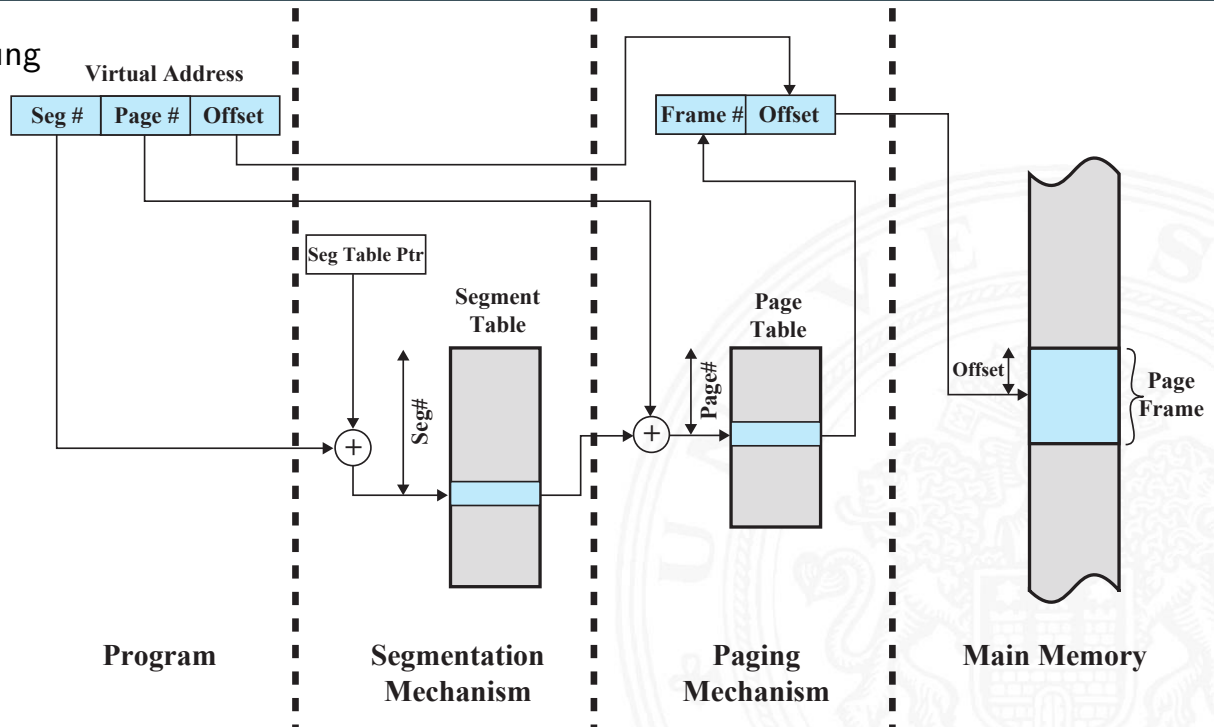
Present bit
Modified bit

- Segmentierung + Paging
 - Segmentierung ⇒ logische Trennung (Text, Data)
 - + Paging ⇒ effiziente Verwaltung
- durch Betriebssystem festgelegt
 - 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.)

Seiten-Adressierung

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

Segment-Adressierung

Segment 0	
Segment 2	
Segment 1	

Segment+Seiten-Adressierung

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



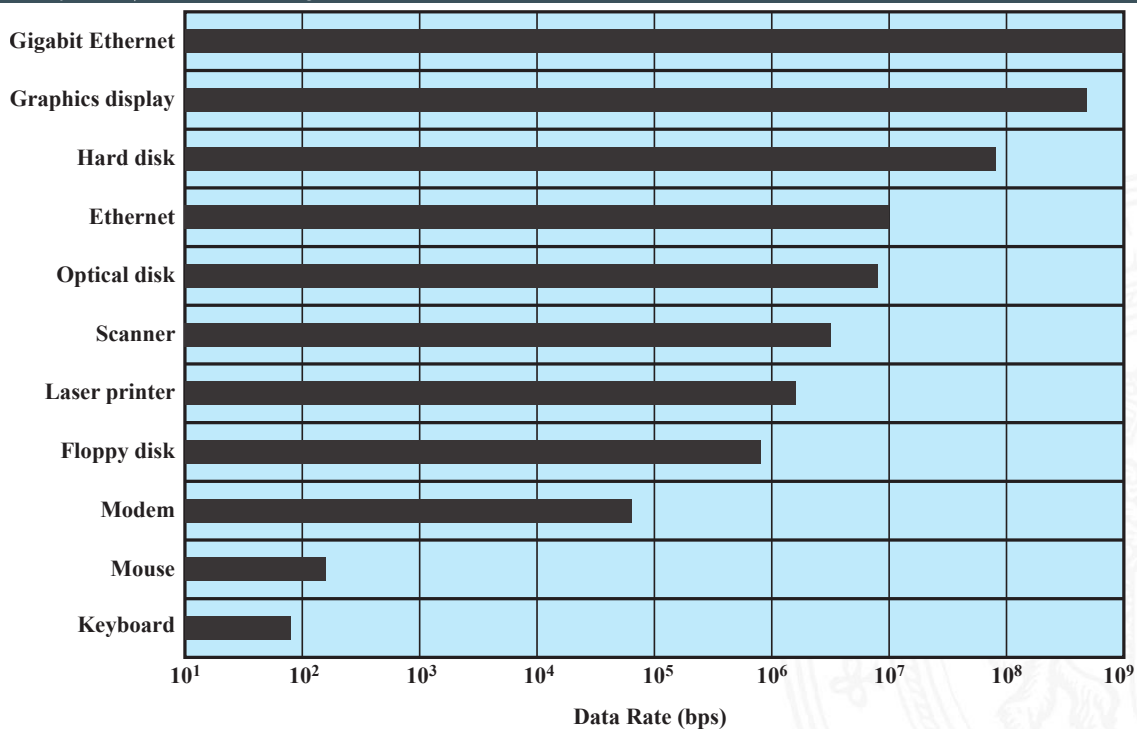
Ein-/Ausgabegeräte

- ▶ 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.)





- ▶ 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
- ⇒ wegen völlig unterschiedlicher Geräteeigenschaften und der technischen Innovation nicht (oder nur 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

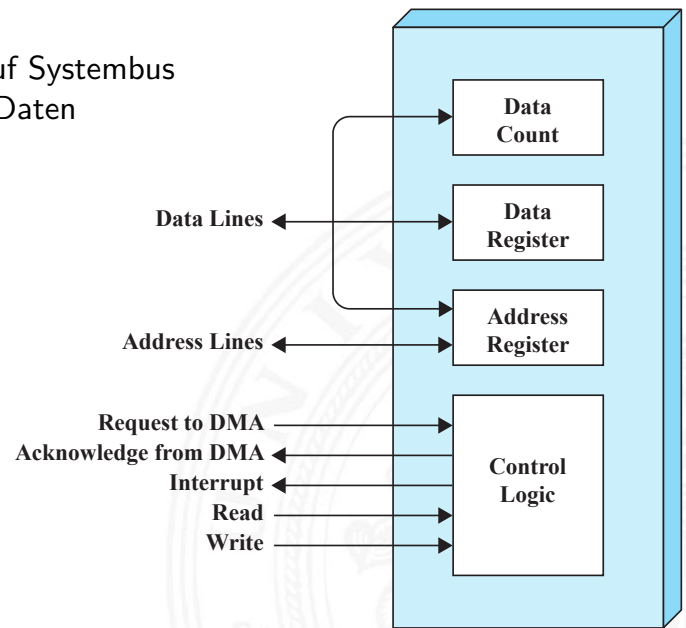


1. Programmed I/O
 - ▶ I/O-Modul führt Operation aus
 - ▶ setzt Bits in Status-Register (Kommunikation)
 - ▶ Prozessor fragt periodisch Status ab
 - „*Busy-waiting*“, schlechte Performanz
2. Interrupt gesteuert
 - ▶ Befehl von Prozessor an I/O-Modul
 - ▶ Interrupt, wenn I/O bereit
 - ▶ Datentransfer durch Prozessor (in ISR)
 - Beteiligung des Prozessors
 - Perfomanz (mehrere Instruktionen pro Datentransfer)
 - siehe 15.2 Betriebssysteme – Interrupts

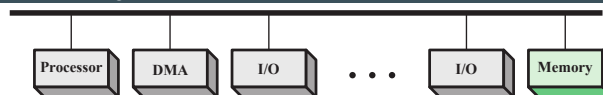


3. DMA (Direct Memory Access)

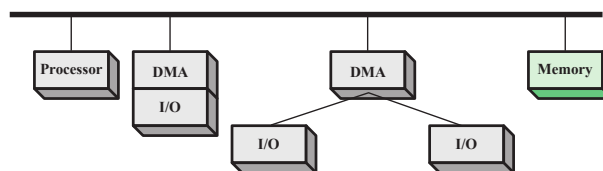
- ▶ separate Hardwareeinheit mit Zugriff auf Systembus
- ▶ DMA-Controller überträgt selbständig Daten
- ▶ DMA-Kommando mit
 - ▶ Lesen / Schreiben
 - ▶ Adresse des I/O Geräts
 - ▶ Startadresse in Hauptspeicher
 - ▶ Anzahl Datenwörter
- ▶ Konflikt, wenn Prozessor Bus benötigt



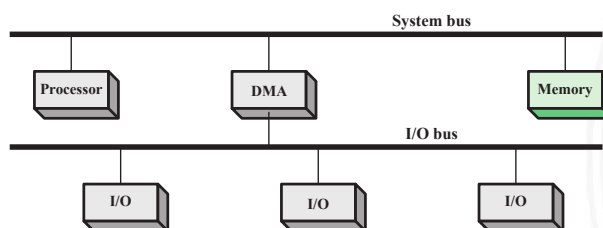
DMA



(a) Single-bus, detached DMA



(b) Single-bus, Integrated DMA-I/O



(c) I/O bus



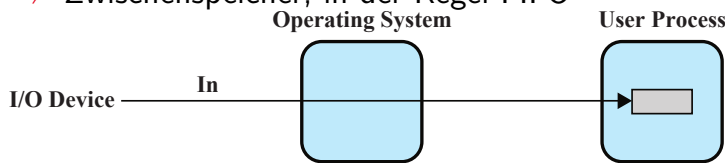
Ein-/Ausgabe Behandlung (cont.)

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

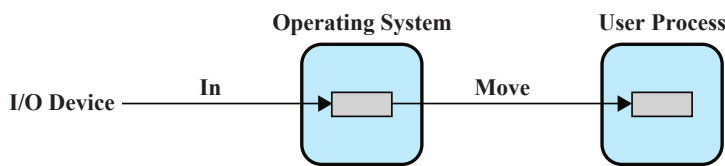
- ▶ lesend: „read-ahead“, Daten schon bereitstellen
 - ▶ schreibend: verzögerte (autonome) Ausführung
- ⇒ Zwischenspeicher, in der Regel FIFO



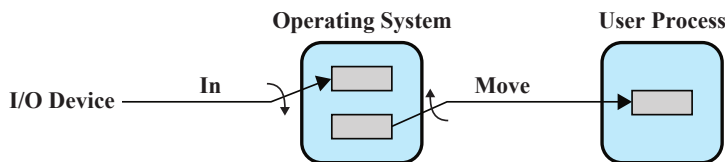
(a) No buffering



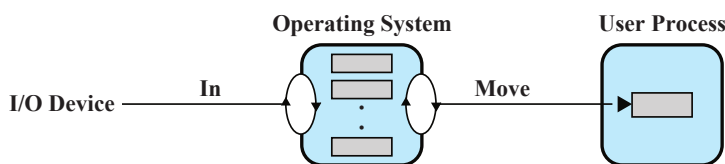
Ein-/Ausgabe Behandlung (cont.)



(b) Single buffering



(c) Double buffering



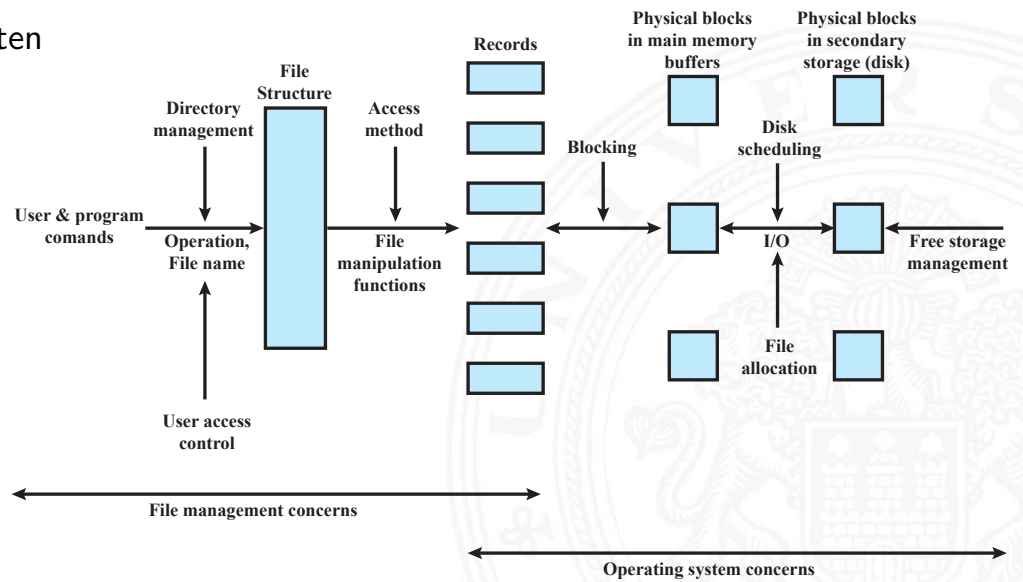
(d) Circular buffering



- ▶ Festplattenzugriffe: das zentrale Thema bei I/O-Optimierung
 - ▶ viele Algorithmen zum „Umsortieren“ von Schreib-/Lesebefehlen
 - ⇒ Optimierung des Durchsatzes

- ▶ Organisation von Daten

- ▶ für Benutzer
 - ▶ logisch
 - ▶ Dateisysteme
- ▶ für Hardware
 - ▶ physikalisch
 - ▶ Blöcke

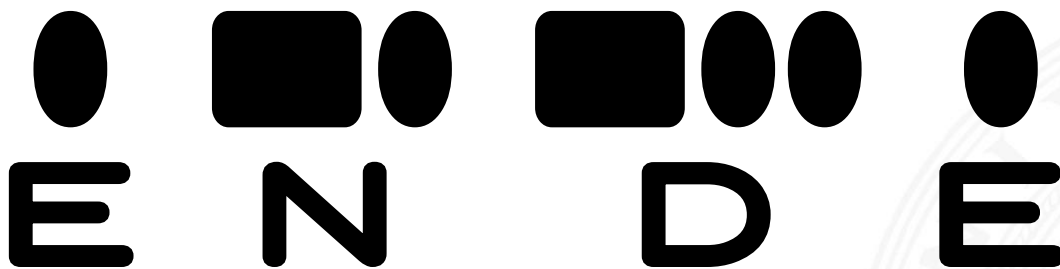




Weitere Themen



Weitere Themen





Literatur

15.8 Betriebssysteme - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [Sta17] W. Stallings:
Operating Systems – Internals and Design Principles.
9th global ed., Pearson Education, 2017. ISBN 978-1-292-21429-0
- [Bau20] C. Baun: *Betriebssysteme kompakt –
Grundlagen, Daten, Speicher, Dateien, Prozesse und Kommunikation.*
2. Auflage, Springer-Verlag GmbH, 2020. ISBN 978-3-662-61410-5
- [SGG18] A. Silberschatz, P.B. Galvin, G. Gagne:
Operating System Concepts.
10th edition, John Wiley & Sons, 2018. ISBN 978-1-119-32091-3



Literatur (cont.)

15.8 Betriebssysteme - Literatur

64-040 Rechnerstrukturen und Betriebssysteme

- [Bra17] R. Brause:
Betriebssysteme – Grundlagen und Konzepte.
4. Auflage, Springer-Verlag GmbH, 2017. ISBN 978-3-662-54099-2
- [TB23] A.S. Tanenbaum, H. Bos:
Modern Operating Systems.
5th Global ed., Pearson, 2023. ISBN 978-1-292-72789-9
- [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`