64-040 Modul IP7: Rechnerstrukturen

http://tams.informatik.uni-hamburg.de/
lectures/2011ws/vorlesung/rs

Andreas Mäder

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

Wintersemester 2011/2012

Gliederung

Gliederung (cont.)

Gliederung

Gliederung (cont.)

Inhalt und Lernziele

- Wie funktioniert ein Digitalrechner?
- Warum Mikroprozessoren?

Kennenlernen der Themen:

- Prinzip des von-Neumann-Rechners
- Abstraktionsebenen, Hardware/Software-Schnittstelle
- Rechnerarithmetik, Zahldarstellung, Codierung
- Prozessor mit Steuerwerk und Operationswerk
- Speicher und -ansteuerung, Adressierungsarten
- Befehlssätze, Maschinenprogrammierung
- Assemblerprogrammierung, Speicherverwaltung

- Fähigkeit zum Einschätzen zukünftiger Entwicklungen
- Chancen und Grenzen der Miniaturisierung

Motivation

- Wie funktioniert ein Digitalrechner?
- Mikroprozessoren?

Warum ist das überhaupt wichtig?

- Informatik ohne Digitalrechner undenkbar
- Grundverständnis der Interaktion von SW und HW
- zum Beispiel für „performante“ Software
- Variantenvielfalt von Mikroprozessorsystemen
  - Supercomputer, Server, Workstations, PCs, ...
  - Medienverarbeitung, Mobile Geräte, ...
  - RFID-Tags, Wegwerfcomputer, ...
- Bewertung von Trends und Perspektiven

Motivation

- ständige Fortschritte in Mikroelektronik und Optoelektronik
- und zwar weiterhin *exponentielles* Wachstum (50 %...100 % pro Jahr)
  - Rechenleistung von Prozessoren („Performance“)
  - Speicherkapazität (DRAM, SRAM, FLASH)
  - Speicherkapazität (Festplatten)
  - Bandbreite (Netzwerke)
- ständig neue Möglichkeiten und Anwendungen
- ständig neue Produkte und Techniken
- und ganz gewiss kein „stationärer Zustand“
- Roadmaps derzeit bis über 2020 hinaus...

Technologie-Fortschritt

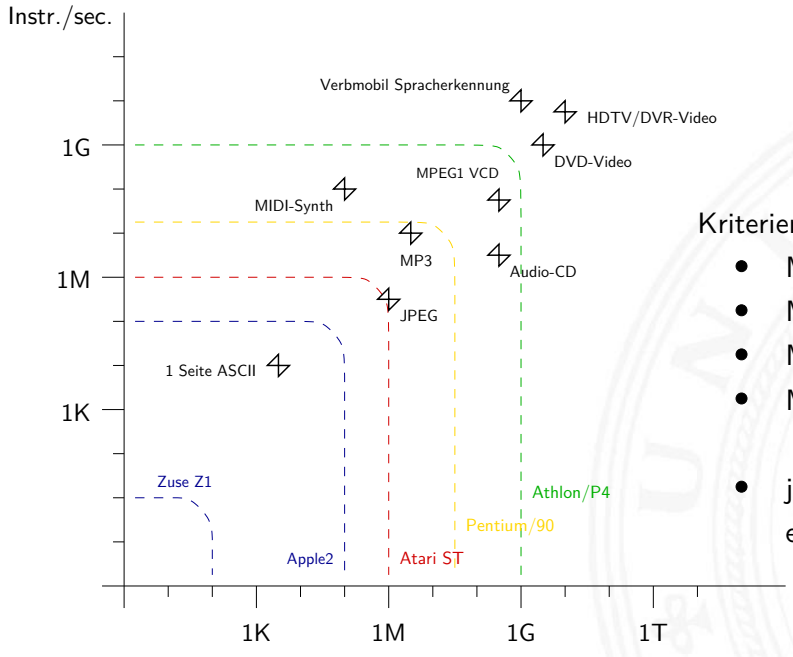
- exponentielles Wachstum, typisch 50 % pro Jahr
- ständig neue Möglichkeiten und Anwendungsfelder
- ständig neue Produkte und Techniken

- Details zu Rechnerorganisation veralten schnell
- aber die Konzepte bleiben gültig (!)

- Schwerpunkt der Vorlesung auf dem „Warum“
- bitte ein Gefühl für Größenordnungen entwickeln

- Software entwickelt sich teilweise viel langsamer
- LISP seit 1958, Prolog 1972, Smalltalk/OO 1972, usw.

Technologie-Fortschritt: neue Anwendungsfelder



Kriterien:

- MIPS
- MBytes (RAM, Platte)
- Mbps
- MPixel

- jede Rechnergeneration erlaubt neue Anwendungen

Neue Anwendungsfelder: Beispiel ReBirth

Propellerheads ReBirth 1996, www.rebirthmuseum.com

- Techno per Software: Echtzeit-Software-Emulation der legendären Roland Synthesizer TB-303 TR-808 TR-909 auf einem PC

Neue Anwendungsfelder: Beispiel Autotune

Sie sehen gut aus, aber Ihr Gesang ist lausig?

Antares Autotune 1999

Themen heute

- Geschichte der Datenverarbeitung
- Wichtige Beispiele

- Technologie-Fortschritt, Skalierung
- Moore's Gesetz, ITRS-Roadmap
- Grenzen der Miniaturisierung: Smart-Dust

- Grundprinzip des von-Neumann-Rechners
- Aufbau, Befehlszyklus, Befehlssatz

Gliederung

Gliederung (cont.)

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
2 Digitalrechner
64-040 Rechnerstrukturen

Gliederung (cont.)

Definition: Digitalrechner

Tanenbaum: Structured Computer Organization

A digital computer is a machine that can solve problems for people by carrying out instructions given to it. A sequence of instructions describing how to perform a certain task is called a program.
The electronic circuits of each computer can recognize and directly execute a limited set of simple instructions into which all its programs must be converted before they can be executed.

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

Befehlssatz und Semantic Gap

- *. . . directly execute a limited set of simple instructions. . .*

Typische Beispiele für solche Befehle:

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

$\Rightarrow$ extrem niedriges Abstraktionsniveau

- natürliche Sprache mit Kontextwissen
  Beispiel: „vereinbaren Sie einen Termin mit dem Steuerberater“
- **Semantic gap**: Diskrepanz zu einfachen/elementaren Anweisungen

- Vermittlung zwischen Mensch und Computer erfordert zusätzliche Abstraktionsebenen und Software

Rechnerarchitektur bzw. -organisation

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

- diverse Optimierungs-Kriterien/Möglichkeiten:
  - Performance, Hardwarekosten, Softwarekosten, ...
  - Wartungsfreundlichkeit, Stromverbrauch, ...

Achtung / Vorsicht:

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

Rückwirkung von unteren Ebenen: Arithmetik

```
public class Overflow {
  ...
  public static void main( String[] args ) {
    printInt( 0 );                    //  0
    printInt( 1 );                    //  1
    printInt( -1 );                   // -1
    printInt( 2+(3*4) );              // 14
    printInt( 100*200*300 );          // 6000000
    printInt( 100*200*300*400 );      // -1894967296          (!)
    printDouble( 1.0 );               // 1.0
    printDouble( 0.3 );               // 0.3
    printDouble( 0.1 + 0.1 + 0.1 );   // 0.3000000000000004 (!)
    printDouble( (0.3) - (0.1+0.1+0.1) ); // -5.5E-17       (!)
  }
}
```

Rückwirkung von unteren Ebenen: Performance

```
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.	⇒ 5x langsamer
Sum = 600.8473695346258 / 600.8473695342268		⇒ andere Werte

Maschine mit mehreren Ebenen

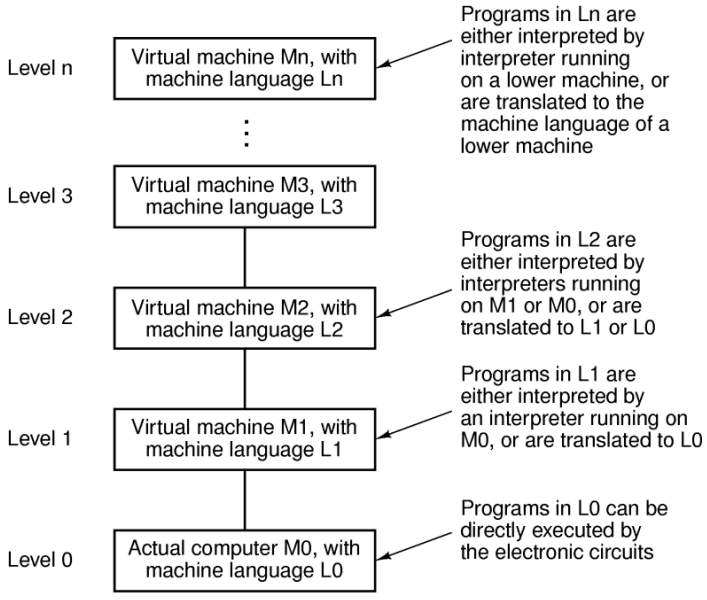


Figure 1-1. A multilevel machine.

Tanenbaum: Structured Computer Organization

Abstraktionsebenen und Sprachen

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

Software zur Übersetzung zwischen den Ebenen

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

Virtuelle Maschine

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

$\Rightarrow$ dies ist gleichwertig mit einer:

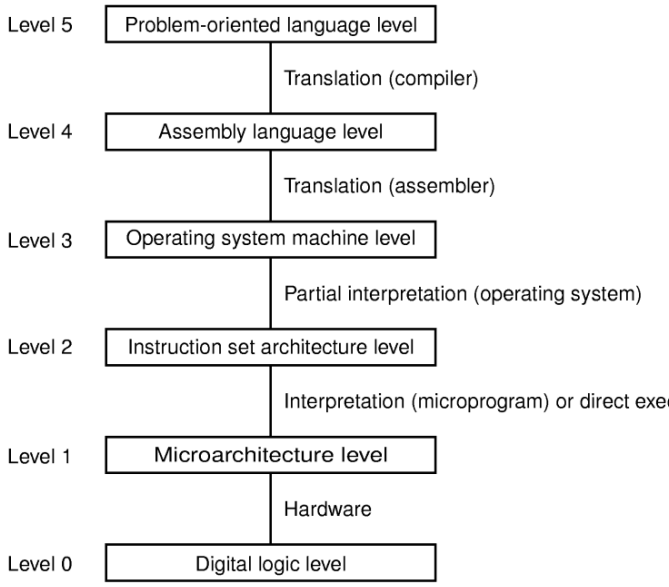
Virtuellen Maschine M1 für die Sprache L1

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

Übliche Einteilung der Ebenen

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

Beispiel: Sechs Ebenen



z.B. Smalltalk, Visual Basic

COFF, ELF, ...

Microsoft Windows XP

Intel x86+MMX+SSE2

Pentium-IV 512KB Cache

konkreter Pentium-IV Chip

Figure 1-2. A six-level computer. The support method for each level is supported is indicated below it (along with the name of the supporting program).

Hinweis: Ebenen vs. Vorlesungen im BSc-Studiengang

Anwendungsebene:	SE1..SE3, AD, ...
Assemblerebene:	RS
Betriebssystemebene:	GSS
Rechnerarchitektur:	RS, RAM
Mikroarchitektur:	RS, RAM
Logikebene:	RS, RAM
Device-Level:	RAM

HelloWorld: Anwendungsebene: Quellcode

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

#include <stdio.h>

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

Übersetzung

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

HelloWorld: Assemblerebene: cat HelloWorld.s

```
main:
  leal   4(%esp), %ecx
  andl   $-16, %esp
  pushl  -4(%ecx)
  pushl  %ebp
  movl   %esp, %ebp
  pushl  %ecx
  subl   $4, %esp
  movl   $.LC0, (%esp)
  call   puts
  movl   $0, %eax
  addl   $4, %esp
  popl   %ecx
  popl   %ebp
  leal   -4(%ecx), %esp
  ret
```


HelloWorld: Objectcode: od -x HelloWorld.o

```
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000020 0001 0003 0001 0000 0000 0000 0000 0000
0000040 00f4 0000 0000 0000 0034 0000 0000 0028
0000060 000b 0008 4c8d 0424 e483 fff0 fc71 8955
0000100 51e5 ec83 c704 2404 0000 0000 fce8 ffff
0000120 b8ff 0000 0000 c483 5904 8d5d fc61 00c3
0000140 6548 6c6c 2c6f 7720 726f 646c 0021 4700
0000160 4343 203a 4728 554e 2029 2e34 2e31 2032
0000200 3032 3630 3131 3531 2820 7270 7265 6c65
0000220 6165 6573 2029 5328 5355 2045 694c 756e
0000240 2978 0000 732e 6d79 6174 0062 732e 7274
0000260 6174 0062 732e 7368 7274 6174 0062 722e
0000300 6c65 742e 7865 0074 642e 7461 0061 622e
0000320 7373 2e00 6f72 6164 6174 2e00 6f63 6d6d
0000340 6e65 0074 6e2e 746f 2e65 4e47 2d55 7473
...
```

HelloWorld: Disassemblieren: objdump -d HelloWorld.o

```
HelloWorld.o:     file format elf32-i386
Disassembly of section .text:
00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  0xfffffffc(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   51                      push   %ecx
   e:   83 ec 04                sub    $0x4,%esp
  11:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  18:   e8 fc ff ff ff          call   19 <main+0x19>
  1d:   b8 00 00 00 00          mov    $0x0,%eax
  22:   83 c4 04                add    $0x4,%esp
...
```


HelloWorld: Maschinencode: od -x HelloWorld.exe

```
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000020 0002 0003 0001 0000 8310 0804 0034 0000
0000040 126c 0000 0000 0000 0034 0020 0009 0028
0000060 001c 001b 0006 0000 0034 0000 8034 0804
0000100 8034 0804 0120 0000 0120 0000 0005 0000
0000120 0004 0000 0003 0000 0154 0000 8154 0804
0000140 8154 0804 0013 0000 0013 0000 0004 0000
0000160 0001 0000 0001 0000 0000 0000 8000 0804
0000200 8000 0804 04c4 0000 04c4 0000 0005 0000
0000220 1000 0000 0001 0000 0f14 0000 9f14 0804
0000240 9f14 0804 0104 0000 0108 0000 0006 0000
0000260 1000 0000 0002 0000 0f28 0000 9f28 0804
...
```

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“ (K.P.Lentz)
— jedenfalls in Bezug auf L1 Programmausführung

Entscheidung für Software- oder Hardwarerealisierung?

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

von-Neumann-Konzept

- J. Mauchly, J.P. Eckert, J. von-Neumann 1945
- System mit Prozessor, Speicher, Peripheriegeräten

- gemeinsamer Speicher für Programme und Daten
- 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

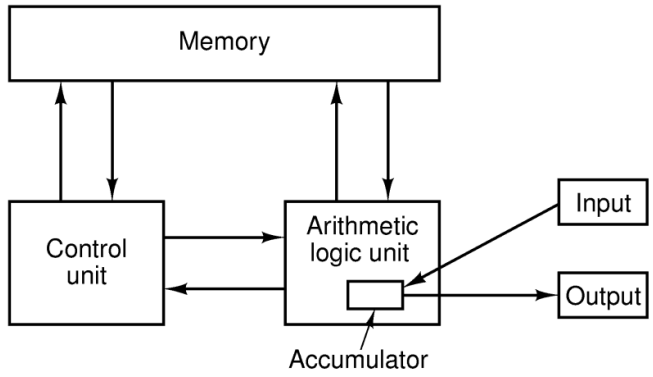


Figure 1-5. The original von Neumann machine.

Fünf zentrale Komponenten:

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

von-Neumann Rechner (cont.)

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

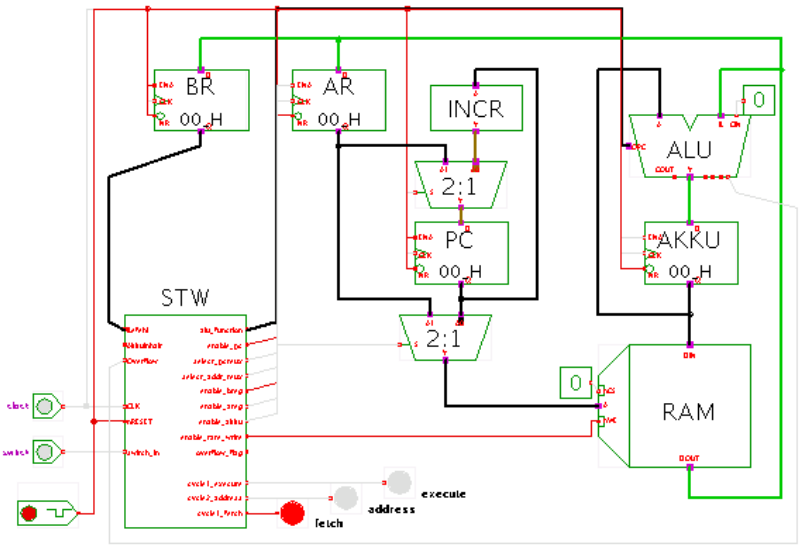
PRIMA: die Primitive Maschine

ein (minimaler) 8-bit von-Neumann Rechner

- RAM: Hauptspeicher 256 Worte à 8-bit
- vier 8-bit Register:
  - PC: program-counter
  - BR: instruction register („Befehlsregister“)
  - AR: address register (Speicheradressen und Sprungbefehle)
  - AKKU: accumulator (arithmetische Operationen)
- eine ALU für Addition, Inkrement, Shift-Operationen
- ein Schalter als Eingabegerät
- sehr einfacher Befehlssatz
- Demo: `http://tams.informatik.uni-hamburg.de/applets/hades/webdemos/50-rtlib/90-prima/chapter.html`

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
2.5 Digitalrechner - von-Neumann-Konzept
64-040 Rechnerstrukturen

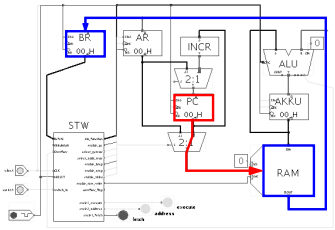
PRIMA: die Primitive Maschine



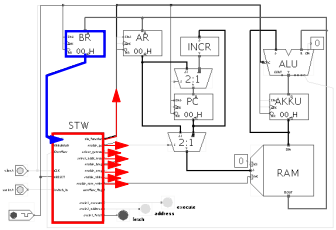
`http://tams.informatik.uni-hamburg.de/applets/hades/webdemos/50-rtlib/90-prima/chapter.html`

PRIMA: die Zyklen

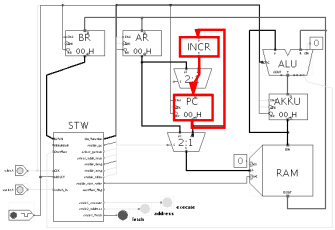
Befehl holen

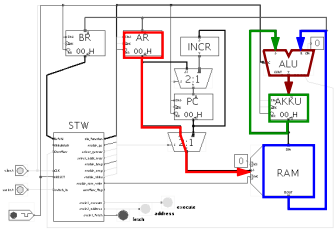


decodieren

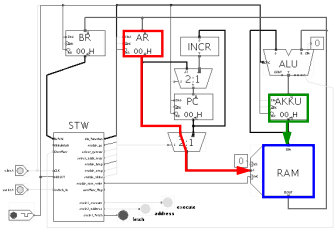


PC inkrementieren

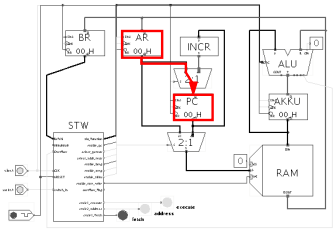




rechnen



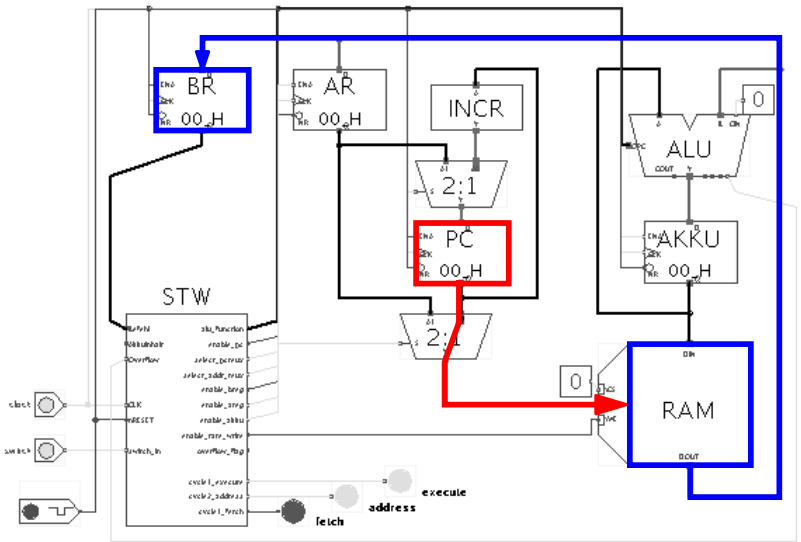
speichern



springen

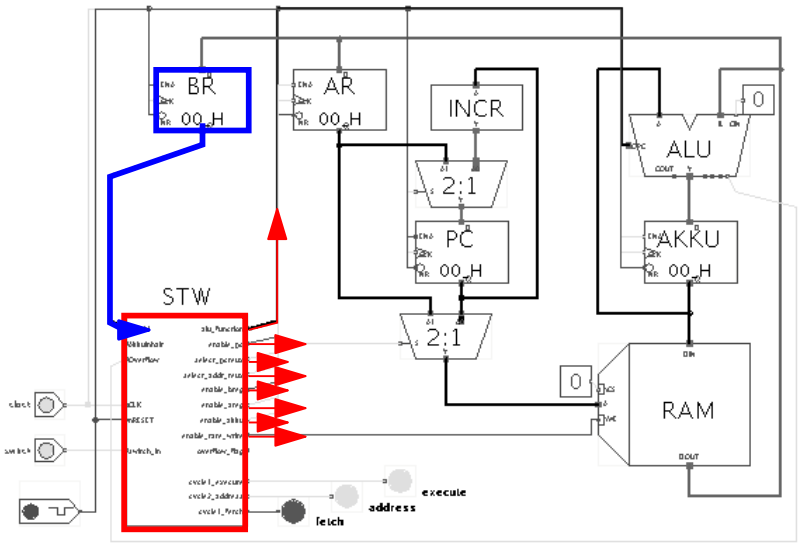
PRIMA: Befehl holen

BR = RAM[PC]



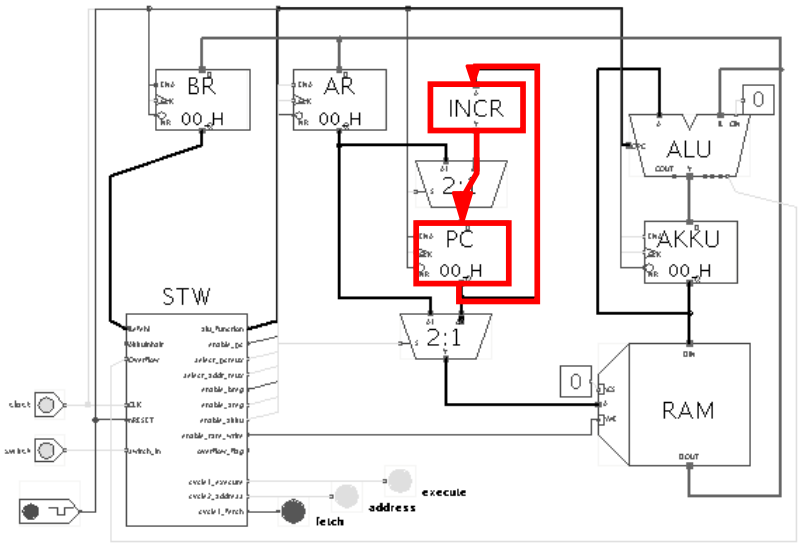
PRIMA: decodieren

Steuersignale = decode(BR)



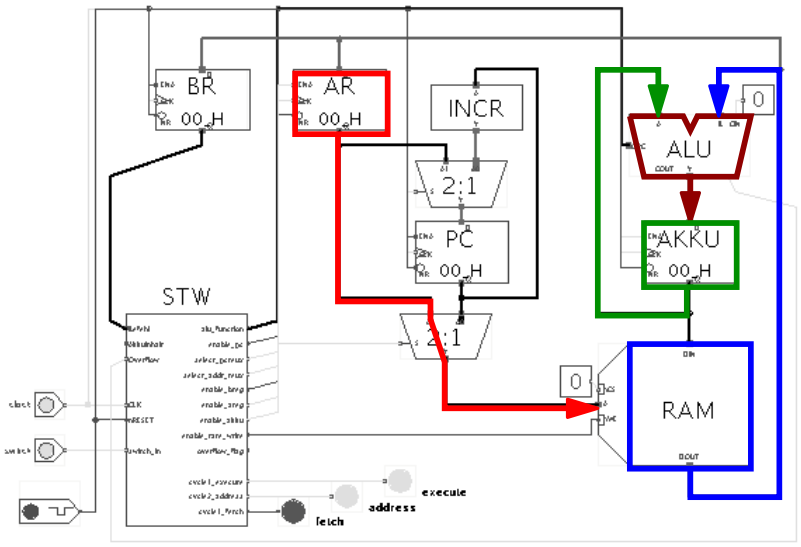
PRIMA: PC inkrementieren

PC = PC+1



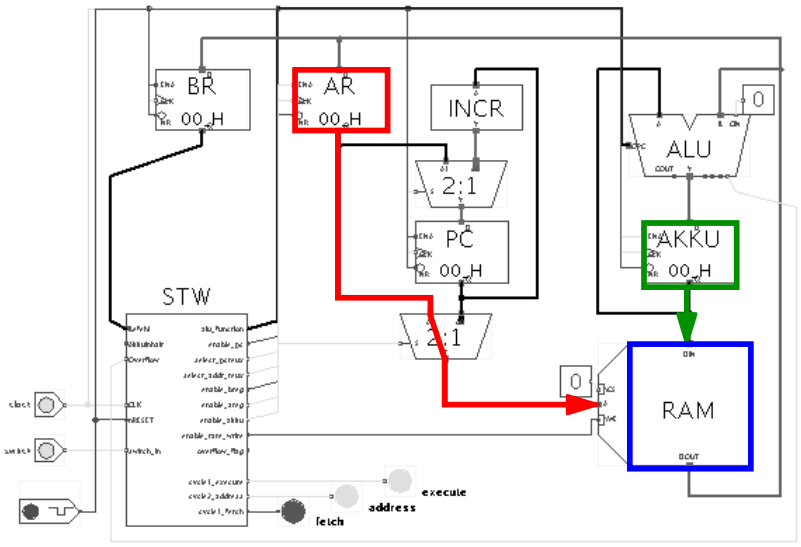
PRIMA: rechnen

Akku = Akku + RAM[AR]



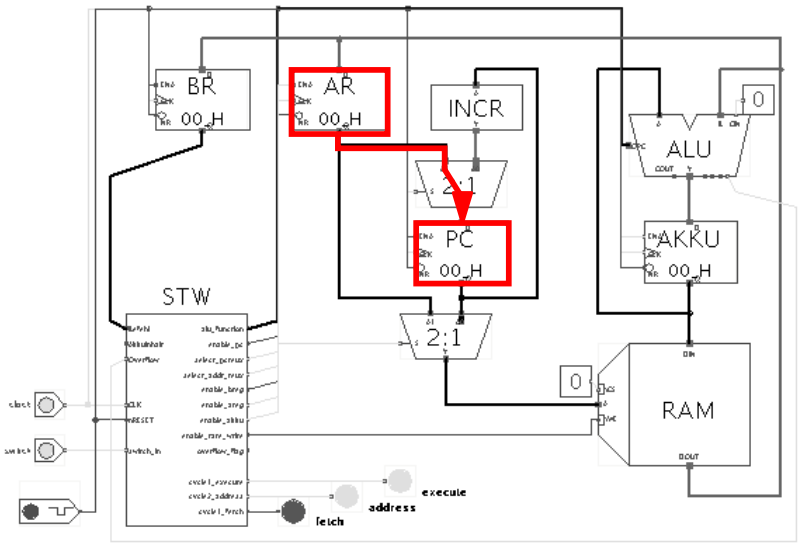
PRIMA: speichern

RAM[AR] = Akku



PRIMA: springen

PC = AR



PRIMA: Simulator

PC: 238
AR: 251
BR: 0
AKKU: 0
OV: 0
state: 0
SW:
trace:
hex:
disassemble:

	0	20	40	60	80	100	120	140	160	180	200	220	240
0	0	72	128	12	72	0	128	128	1	72	14	131	9
1	0	4	200	0	2	199	108	92	191	191	0	42	252
2	0	72	9	72	14	0	0	72	137	128	72	72	6
3	1	5	250	5	0	198	0	193	92	158	250	253	0
4	10	14	72	131	72	72	0	14	9	11	9	9	72
5	9	0	101	68	3	197	0	0	189	44	248	252	252
6	0	72	9	128	9	127	0	1	0	33	72	193	128
7	42	3	45	28	8	92	0	193	191	22	251	234	216
8	10	9	10	9	72	8	0	128	72	11	9	9	0
9	100	2	0	4	5	0	0	138	171	183	249	250	1
10	14	72	72	12	128	8	0	14	9	5	72	0	8
11	0	248	45	0	28	0	0	0	0	0	252	251	0
12	72	9	9	72	128	8	9	72	0	0	9	72	0
13	2	3	3	4	92	0	195	192	192	0	254	250	1
14	9	72	10	131	0	9	1	15	72	7	72	9	9
15	9	249	0	92	0	121	194	0	192	5	253	251	0
16	72	9	72	9	0	0	137	72	9	2	9	0	
17	45	7	3	2	0	196	142	191	191	0	253	251	
18	9	72	9	10	0	72	72	9	10	22	12	72	
19	8	221	5	0	0	121	193	190	0	77	0	251	

ADD 251

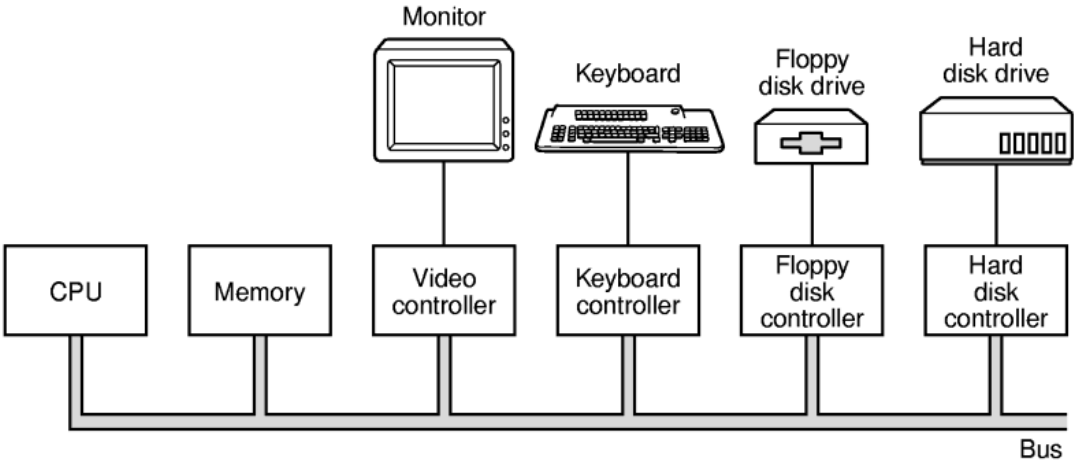
Ablaufprotokoll, '?<enter>' für Hilfe...

Cmd>

Takt | Befehl | 5 Befehle | Reset | RAM löschen | Laden... | Sichern...

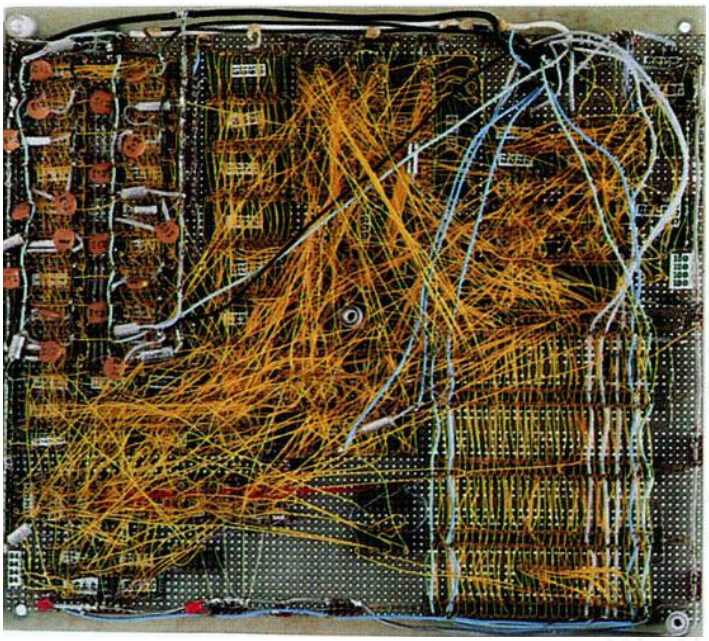
http://tams.informatik.uni-hamburg.de/applets/jython/prima.html

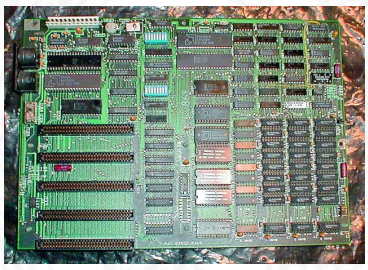
Personal Computer: Aufbau des IBM PC (1981)



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

Personal Computer: Prototyp (1981) und Hauptplatine





Personal Computer: Aufbau mit PCI-Bus (2000)

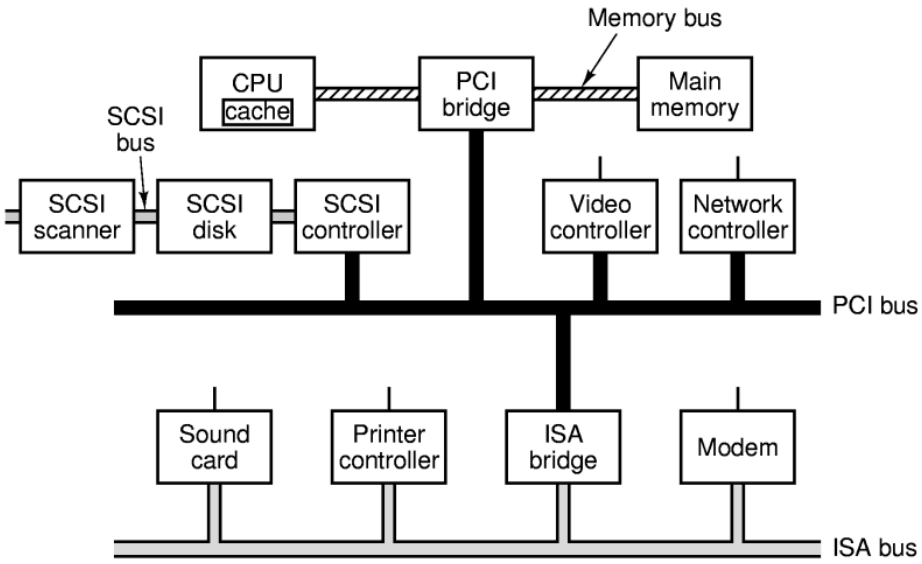
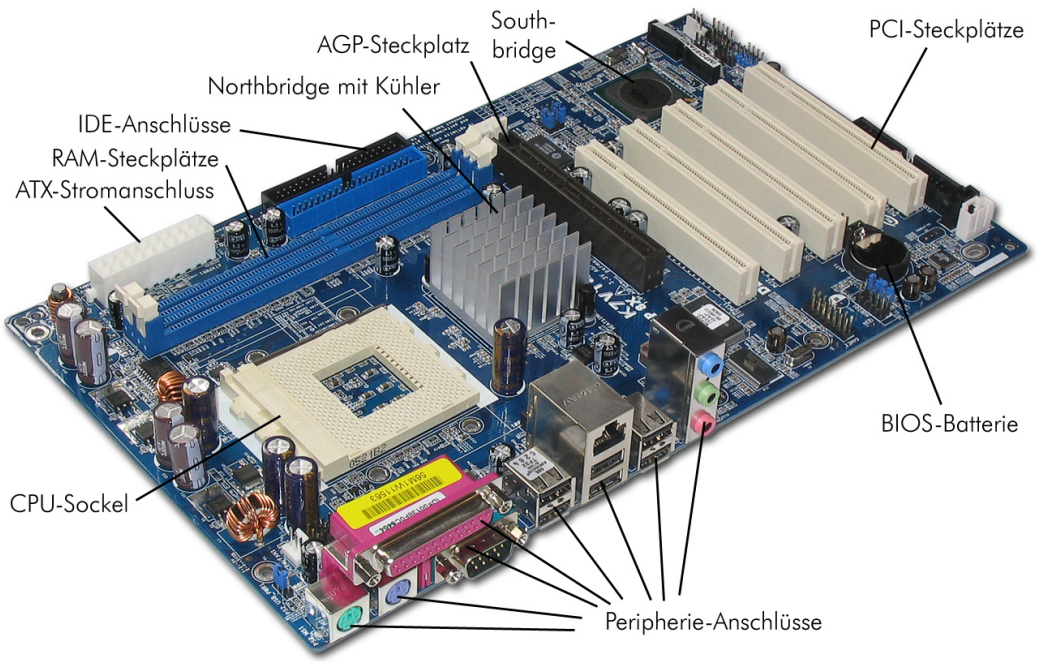


Figure 2-30. A typical modern PC with a PCI bus and an ISA bus. The modem and sound card are ISA devices; the SCSI controller is a PCI device.

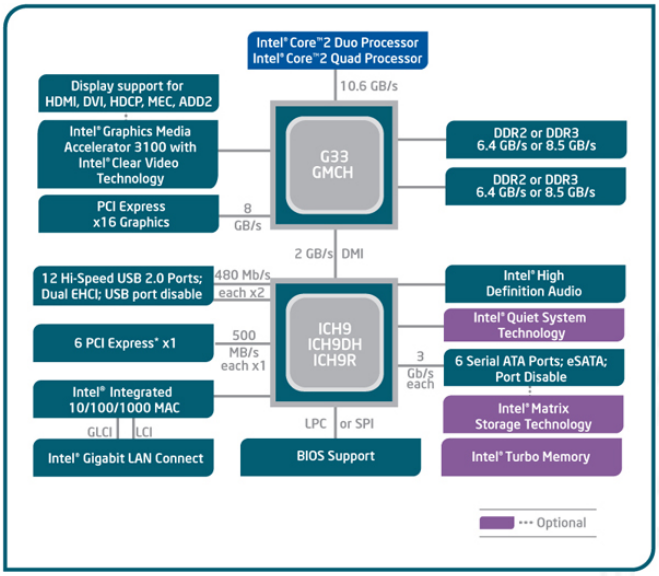
Tanenbaum

Personal Computer: Hauptplatine (2005)



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

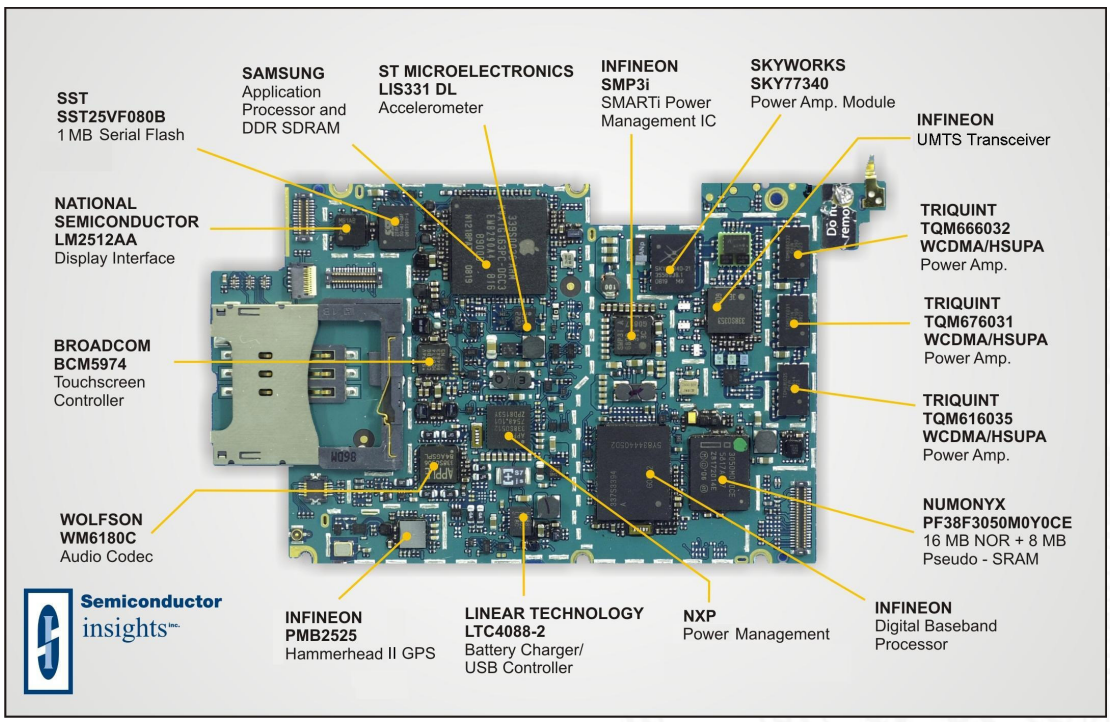
Personal Computer: Aufbau (2010)



Intel

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

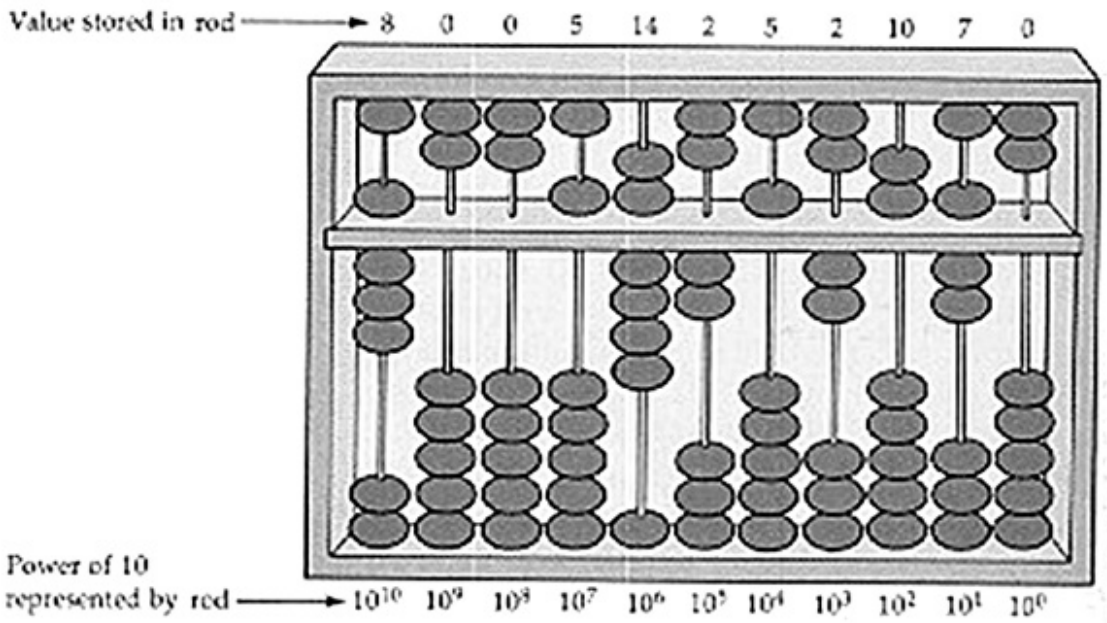
Mobilgeräte: Smartphone (2010)



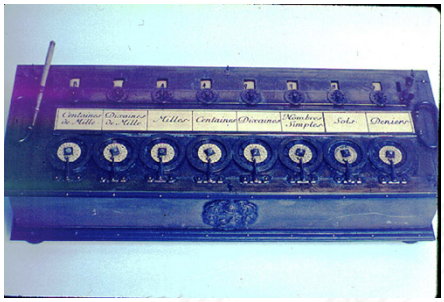
Timeline: Vorgeschichte

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

Abakus



Mechanische Rechenmaschinen



1623 Schickard: Sprossenrad, Addierer/Subtrahierer

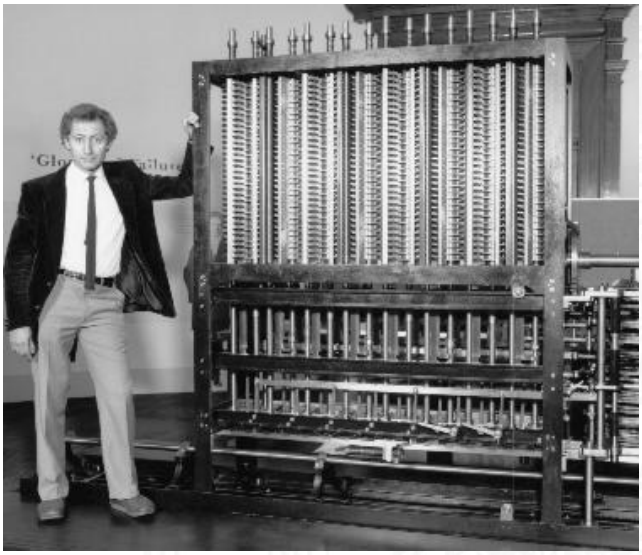
1642 Pascal: „Pascalene“

1673 Leibniz: Staffelwalze, Multiplikation/Division

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

Difference Engine

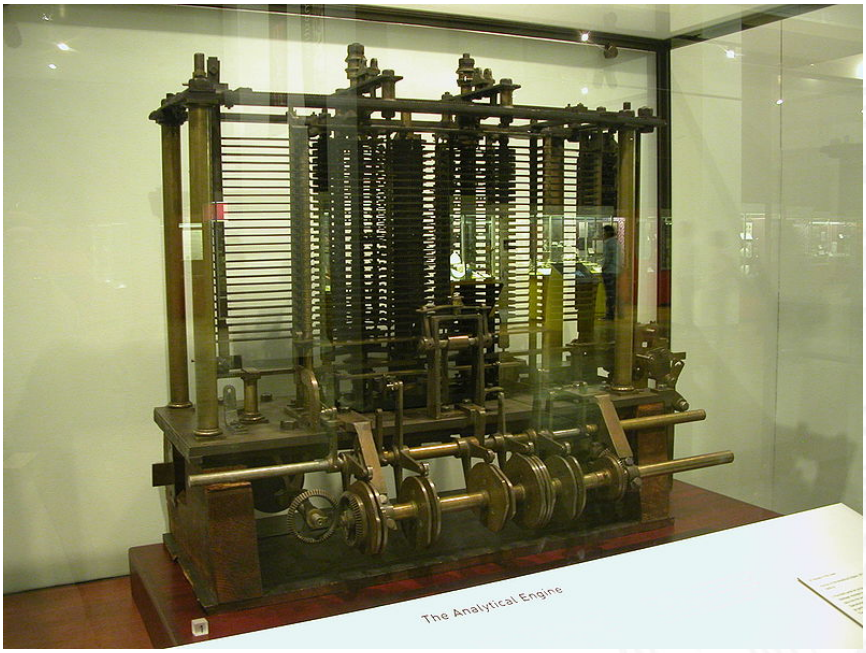
Charles Babbage 1822: Berechnung nautischer Tabellen



Original von 1832 und Nachbau von 1989, London Science Museum

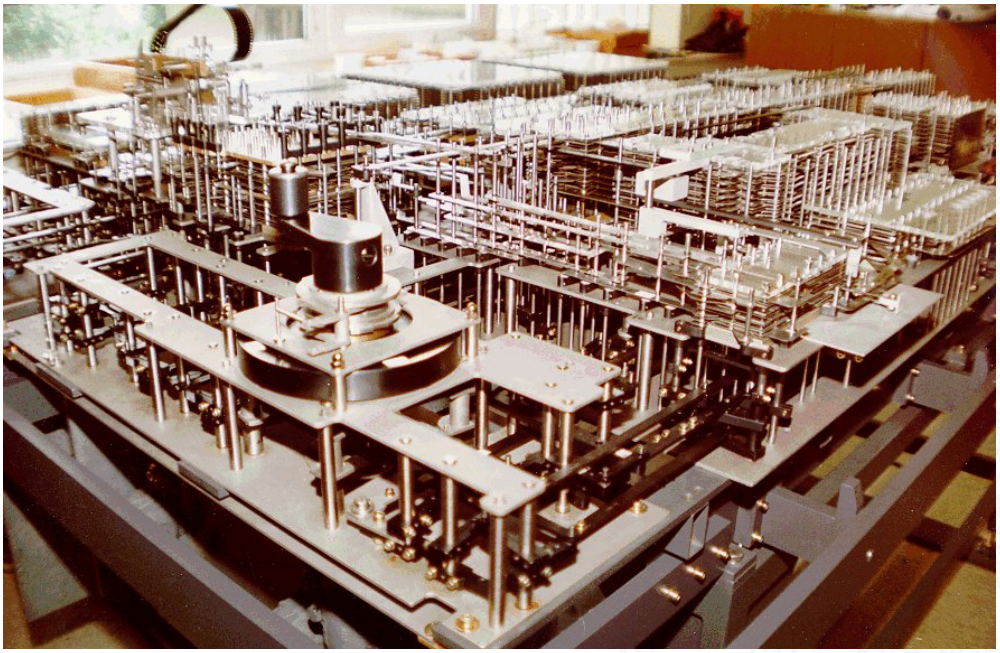
Analytical Engine

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



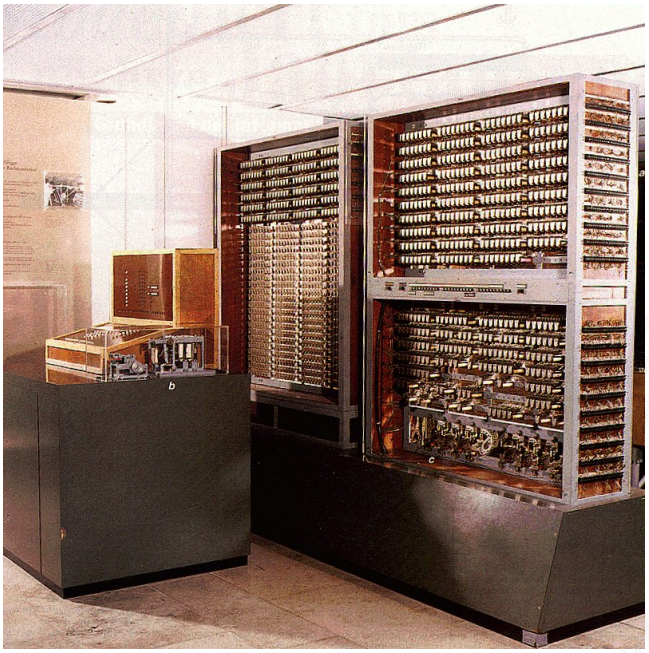
Zuse Z1

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



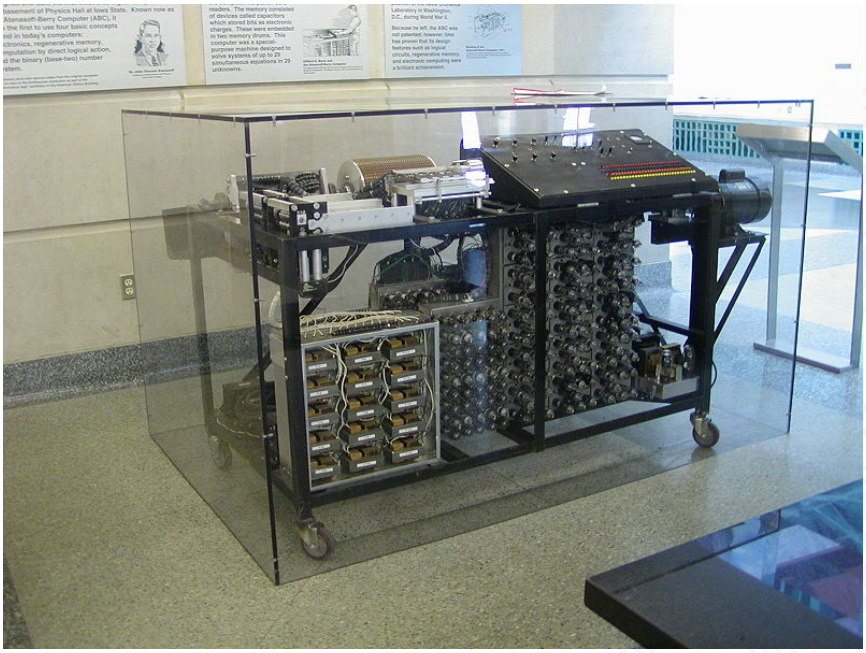
Zuse Z3

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



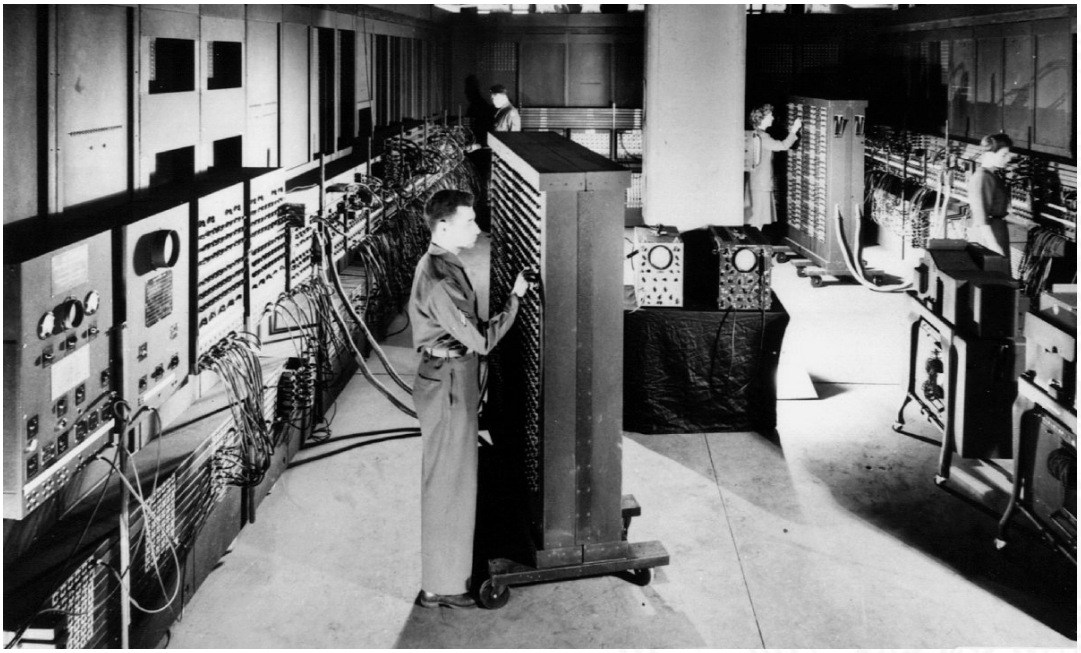
Atanasoff-Berry Computer (ABC)

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

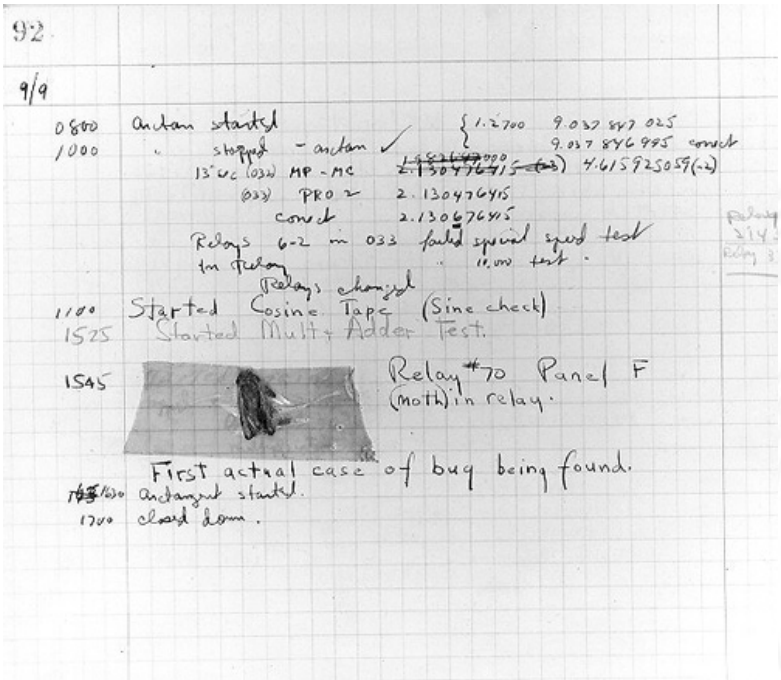


ENIAC — Electronic Numerical Integrator and Computer

Mauchly & Eckert, 1946: Röhren, Steckbrett-Programm

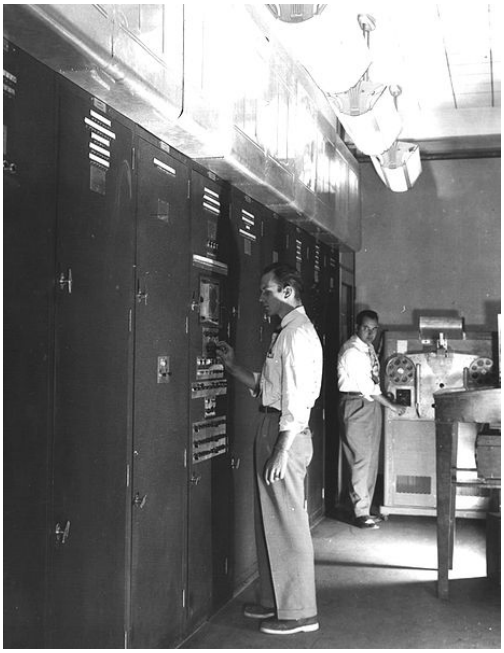


First computer bug



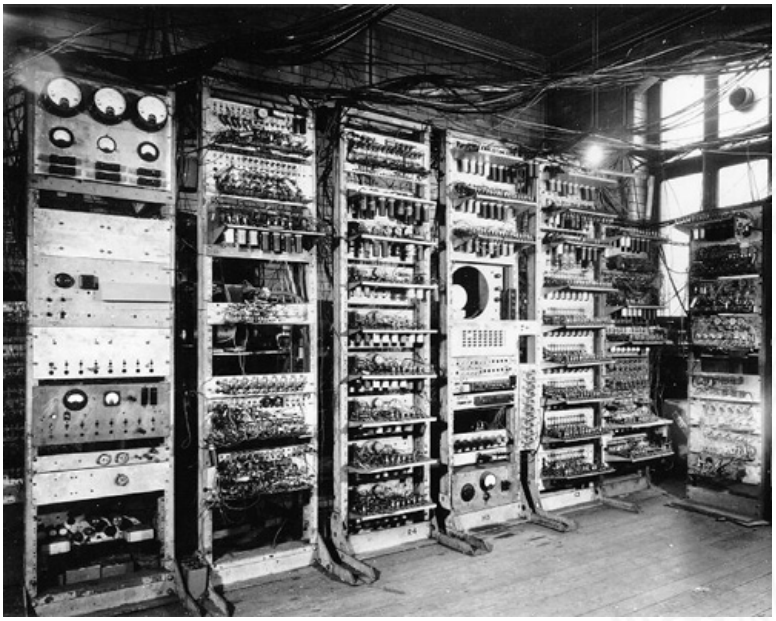
EDVAC

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



Manchester Mark-1

Williams & Kilburn, 1949: Trommelspeicher, Indexregister



Manchester EDSAC

Wilkes 1951: Mikroprogrammierung, Unterprogramme, speicherprogrammiert

Timeline: Verbesserungen

1952: IBM 701	(Pipeline)
1964: IBM S/360	(Rechnerfamilie, software-kompatibel)
1971: Intel 4004	(4-bit Mikroprozessor)
1972: Intel 8008	(8-bit Mikrocomputer-System)
1979: Motorola 68000	(16/32-bit Mikroprozessor)
1980: Intel 8087	(Gleitkomma-Koprozessor)
1981: Intel 8088	(8/16-bit für IBM PC)
1984: Motorola 68020	(32-bit, Pipeline, on-chip Cache)
1992: DEC Alpha AXP	(64-bit RISC-Mikroprozessor)
1997: Intel MMX	(MultiMedia eXtension Befehlssatz)
2006: Sony Playstation 3	(1+8 Kern-Multiprozessor)
2006: Intel-VT / AMD-V	(Virtualisierung)
...	

erste Computer, ca. 1950:

- zunächst noch kaum Softwareunterstützung
- nur zwei Schichten:

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

Mikroprogrammierung (Maurice Wilkes, Cambridge, 1951):

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

erste Betriebssysteme

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

⇒ Maschine wird nicht gut ausgelastet

⇒ Anwender mit lästigen Details überfordert

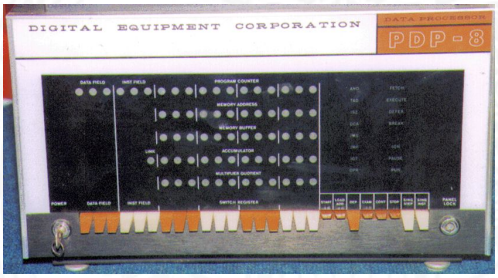
Einführung von **Betriebssystemen**

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

zweite Generation: Transistoren

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

- Beispiel Digial Equipment Corporation PDP-1 (1961)
  - 4K Speicher (4096 Worte á 18-bit)
  - 200 kHz Taktfrequenz
  - 120.000 $
  - Grafikdisplay: erste Computerspiele
- Nachfolger PDP-8: 16.000 $
  - erstes Bussystem
  - 50.000 Stück verkauft



Festplatten

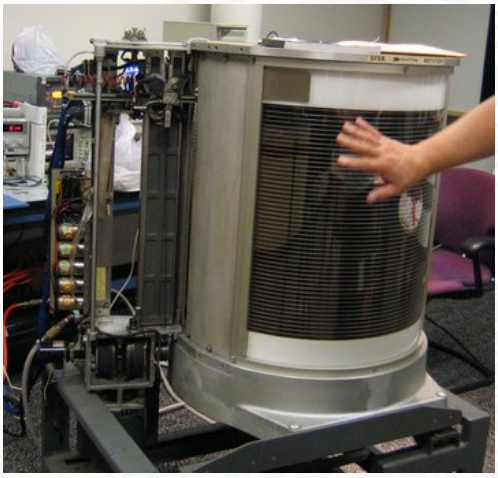
Massenspeicher bei frühen Computern:

- Lochkarten
- Lochstreifen
- Magnetband

- Magnettrommel
- Festplatte

  IBM 350 RAMAC (1956)

  5 MByte, 600 ms Zugriffszeit



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

dritte Generation: ICs

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

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

Property	Model 30	Model 40	Model 50	Model 65
Relative performance	1	3.5	10	21
Cycle time (nsec)	1000	625	500	250
Maximum memory (KB)	64	256	256	512
Bytes fetched per cycle	1	2	4	16
Maximum number of data channels	3	3	4	6

Figure 1-7. The initial offering of the IBM 360 product line.

vierte Generation: VLSI

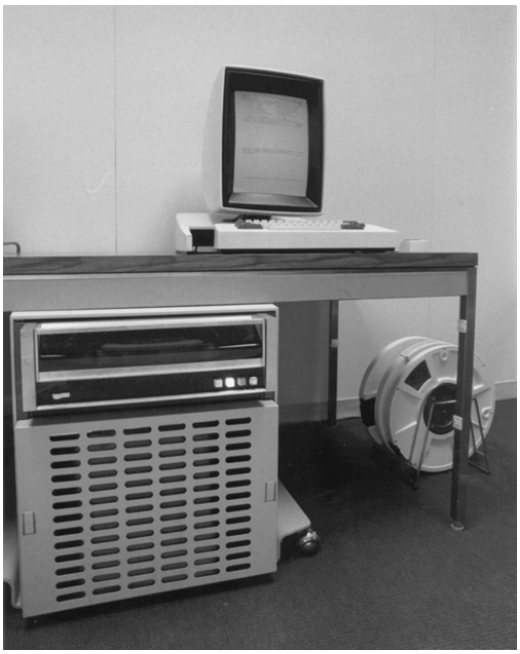
- VLSI = *Very Large Scale Integration*
- ab 10.000+ Transistoren pro Chip

- 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



Rechner-Spektrum

Type	Price ($)	Example application
Disposable computer	1	Greeting cards
Embedded computer	10	Watches, cars, appliances
Game computer	100	Home video games
Personal computer	1K	Desktop or portable computer
Server	10K	Network server
Collection of Workstations	100K	Departmental minisupercomputer
Mainframe	1M	Batch data processing in a bank
Supercomputer	10M	Long range weather prediction

Figure 1-9. The current spectrum of computers available. The prices should be taken with a grain (or better yet, a metric ton) of salt.

Gliederung

Gliederung (cont.)

Moore's Law

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

⇒ bei gleicher Funktion kleinere und billigere Chips

⇒ bei gleicher Größe leistungsfähigere Chips

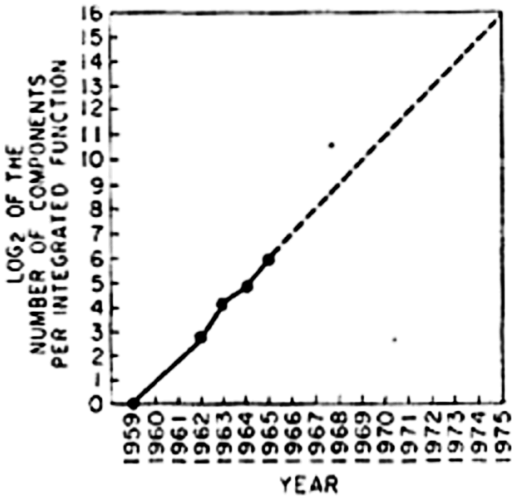
Moore's Law — Gordon Moore, Mitgründer von Intel, 1965

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.)



Gordon Moore 1965:
„Cramming more components onto integrated circuits"

Wird das so weitergehen?

- Vorhersage gilt immer noch
- „*ITRS*" Prognose bis über Jahr 2020 hinaus

Moore's Law: Transistoren pro Speicherchip

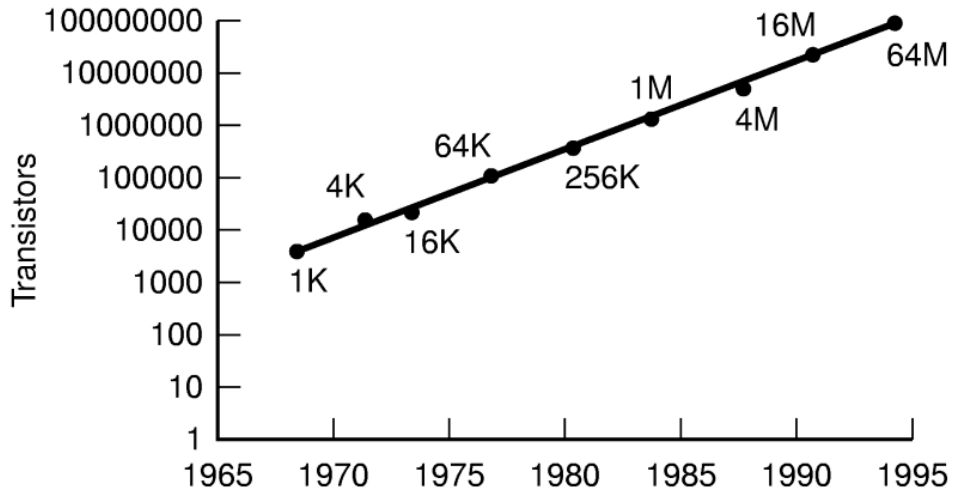
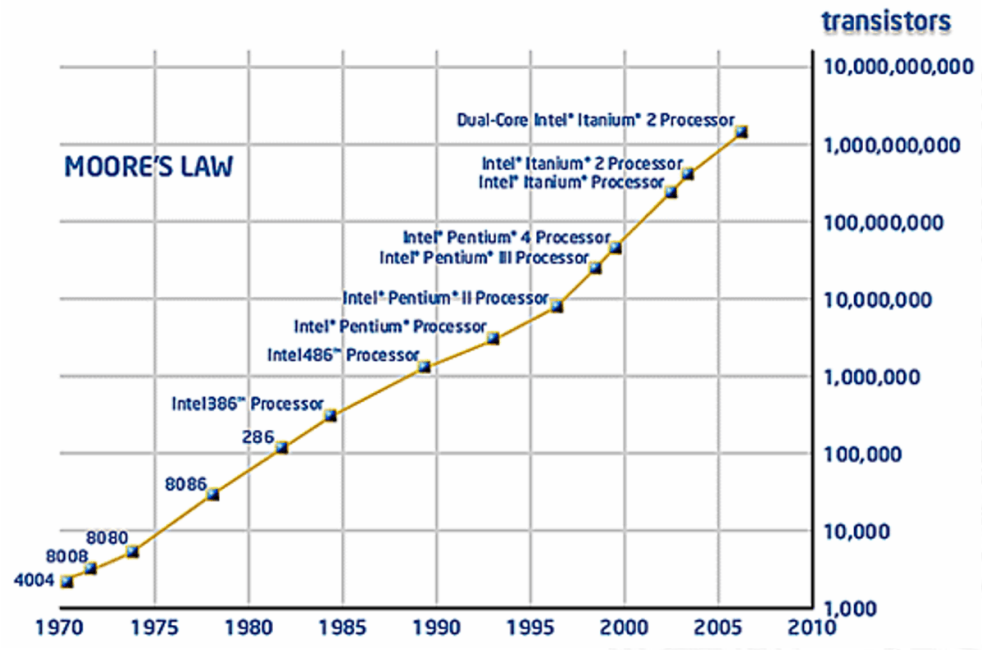


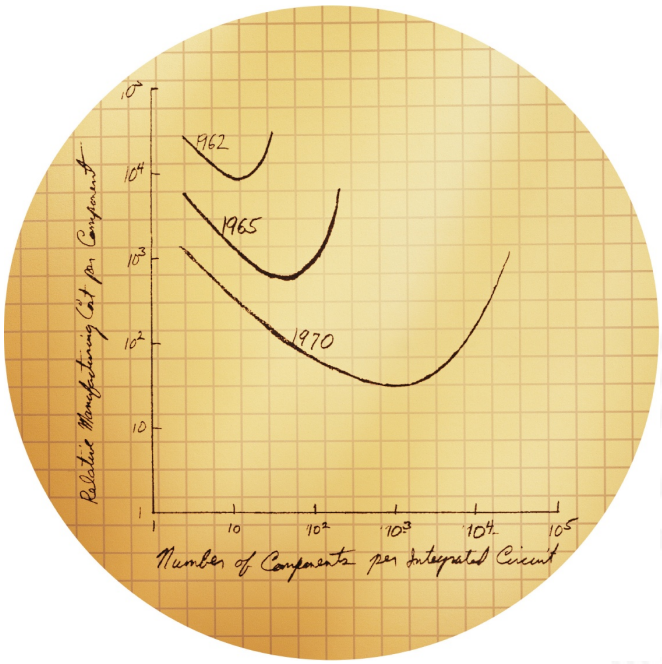
Figure 1-8. Moore's law predicts a 60 percent annual increase in the number of transistors that can be put on a chip. The data points given in this figure are memory sizes, in bits.

Moore's Law: Evolution des Intel x86 (bis 2010)



Intel

Moore's Law: Kosten pro Komponente



Originalskizze von G. Moore www.intel.com

Moore's Law: Formel und Beispiele

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

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

Jahr	Rechner	Linpack in Gflop/s	Zahl der Prozessoren
1993	Fujitsu NWT	124	140
1994	Intel Paragon XP/S MP	281	6.768
1996	Hitachi CP-PACS	368	2.048
1997	Intel ASCI Red (200 MHz Pentium Pro)	1.338	9.152
1998	ASCI Blue-Pacific (IBM SP 640E)	2.144	5.808
1999	ASCI Intel Red (Pentium II Xeon)	2.379	9.632
2000	ASCI White, IBM (SP Power 3)	4.903	7.424
2002	Earth Simulator, NEC	35.610	5.104
2006	JUBL	45.600	16.384
2008	IBM Roadrunner	1.105.000	124.400[1]
2009	Jaguar am ORNL, Cray	1.759.000	224.162[2]

[1] Anzahl der Kerne (6.480 Opteron, 12.960 Cell)
[2] Anzahl der Kerne (Basis: Opteron)

Moore's Law: Aktuelle Trends

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

Entwicklungen

- 4 GByte Hauptspeicher (und mehr) wird bezahlbar
- Übergang von 32-bit auf 64-bit Adressierung

⇒ Integration mehrerer CPUs auf einem Chip (Dual-/Quad-Core)
⇒ zunehmende Integration von Peripheriegeräten
⇒ ab 2011: CPU plus leistungsfähiger Grafikchip
⇒ **SoC**: „System on a chip“

SoC: System on a chip

Gesamtes System auf einem Chip integriert:

- ein oder mehrere Prozessoren
- Befehls- und Daten-Caches für die Prozessoren
- Hauptspeicher (dieser evtl. auch extern)
- weitere Speicher für Medien/Netzwerkoperationen
- Peripherieblöcke nach Kundenwunsch konfiguriert:
  - serielle und parallele Schnittstellen, I/O-Pins
  - Displayansteuerung
  - USB, Firewire, SATA
  - Netzwerk kabelgebunden (Ethernet)
  - Funkschnittstellen: WLAN, Bluetooth, GSM/UMTS
  - Feldbusse: I^2C, CAN, ...
- Handy, Medien-/DVD-Player, WLAN-Router, usw.

SoC Beispiel: Bluetooth-Controller – Chiplayout

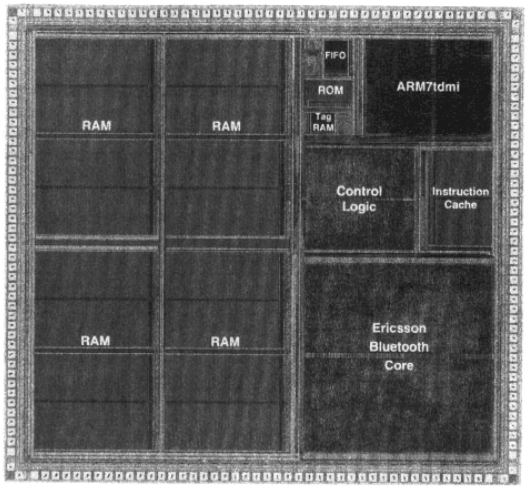


Figure 13.9 Bluetooth Baseband Controller die photograph.

Table 13.1 Bluetooth characteristics.

Process	0.25 µm	Transistors	4,300,000	MIPS	12
Metal layers	3	Die area	20 mm^2	Power	75 mW
Vdd (typical)	2.5 V	Clock	0–13 MHz	MIPS/W	160

S. Furber, *ARM System-on-Chip Architecture*, 2000

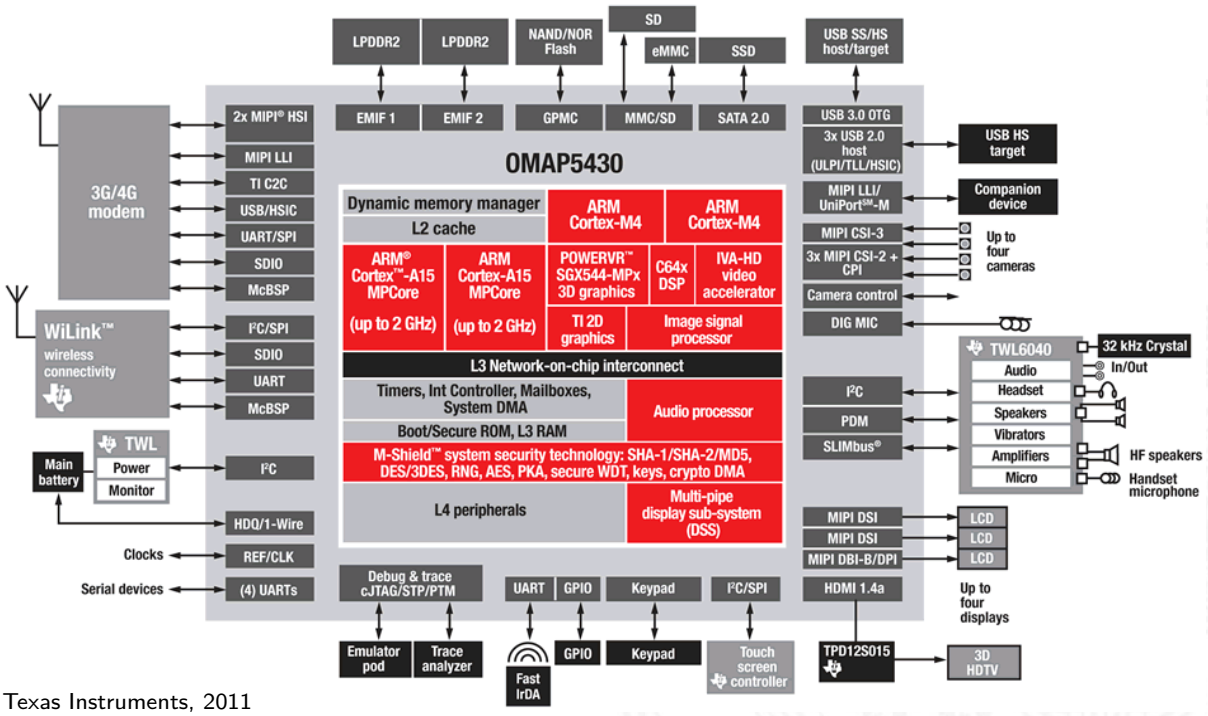
SoC Beispiel: OMAP 5430

- mehrere (verschiedene) CPUs
- Grafikbeschleuniger
- Chipsatz (Speichercontroller, Interconnect, ...)
- Schnittstellen (WiFi, 4G, USB, Audio, I/O, ...)

OMAP5430 Key Benefits

- Designed to drive Smartphones, Tablets and other multimedia-rich mobile devices
- Multi-core ARM® Cortex™ processors
  - Two ARM Cortex-A15 MPCore processors capable of speeds up to 2 GHz each
  - Two ARM Cortex-M4 processors for low-power offload and real-time responsiveness
- Multi-core POWERVR™ SGX544-MPx graphics accelerators drive 3D gaming and 3D user interfaces
- Dedicated TI 2D BitBlt graphics accelerator
- IVA-HD hardware accelerators enable full HD 1080p60, multi-standard video encode/decode as well as 1080p30 stereoscopic 3D (S3D)
- Faster, higher-quality image and video capture with up to 24 megapixels (or 12 megapixels S3D) imaging and 1080p60 (or 1080p30S3D) video
- Supports four cameras and four displays simultaneously
- Packaging and memory: 14mm x 14mm, 0.4mm pitch PoP dual-channel LPDDR2 memory

SoC Beispiel: OMAP 5430 (cont.)



Texas Instruments, 2011

Smart Dust

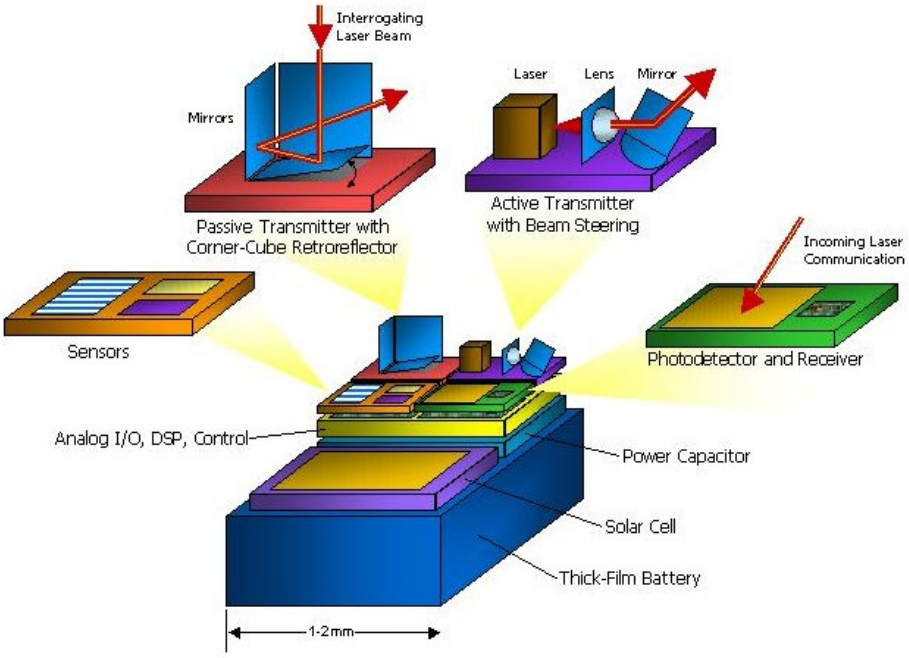
Wie klein kann man Computer bauen?

- Berkeley Projekt: **Smart Dust** 2002-2006
- Integration kompletter Rechensysteme auf 1 mm^3
  - vollständiger Digitalrechner CPU, Speicher, I/O
  - Sensoren Photodioden, Kompass, Gyro
  - Kommunikation Funk, optisch
  - Stromversorgung Photozellen, Batterie, Vibration, Mikroturbine
  - Echtzeit-Betriebssystem Tiny OS
  - inklusive autonome Vernetzung

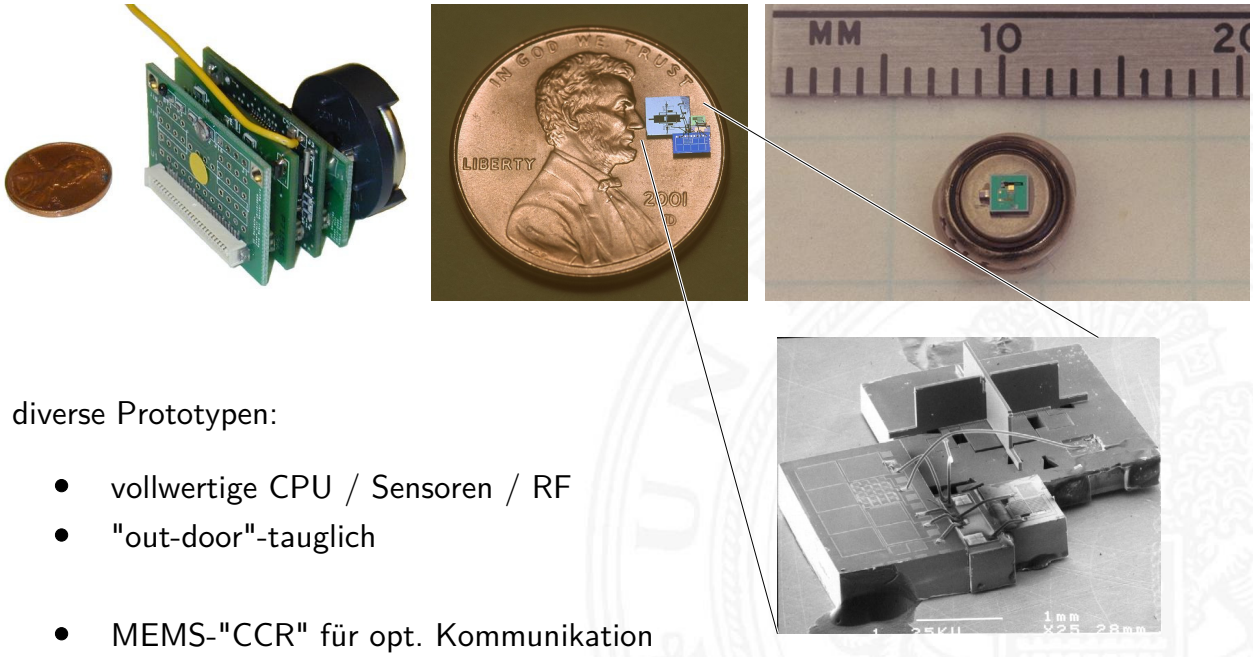
- Massenfertigung? Tausende autonome Mikrorechner
- „Ausstreuen“ in der Umgebung
- vielfältige Anwendungen

Berkeley Sensor & Actuator Center, `eecs.berkeley.edu`

Smart Dust: Konzept



Smart Dust: Prototypen

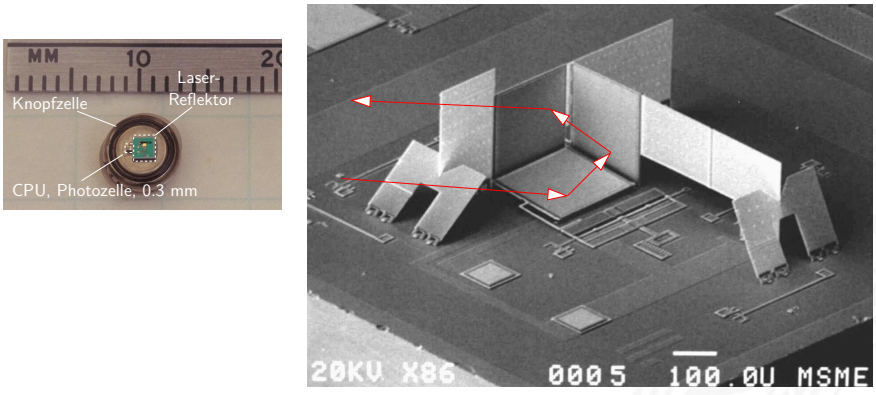


diverse Prototypen:

- vollwertige CPU / Sensoren / RF
- "out-door"-tauglich

- MEMS-"CCR" für opt. Kommunikation

Smart Dust: Corner-cube reflector („Katzenauge“)

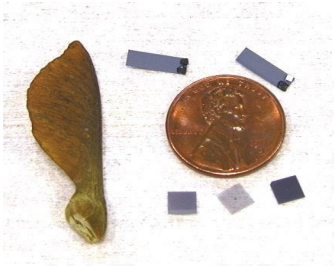


- CCR: seitlich zwei starre Spiegel, Gold auf Silizium
- untere Spiegelfläche beweglich (elektrostatisch, ca. 30 V)
- gezielte Modulation von eingestrahltem Laserlicht
- Reichweiten > 100 m demonstriert

robotics.eecs.berkeley.edu/~pister/SmartDust

Smart Dust: Energieverbrauch

Miniatur-Solarzellen
Wirkungsgrad ca. 3%
26 µW/mm in vollem Sonnenlicht



Batterien:	~ 1J/mm³		
Kondensatoren:	~10 mJ/mm³		
Solarzellen:	~ 0.1 mW/mm	~ 1J/mm /day	(außen,Sonne)
	~ 10 µW/mm	~ 10mJ/mm /day	(innen)
Digitalschaltung	1 nJ/instruction	(StrongArm SA1100)	
Analoger Sensor	1 nJ/sample		
Kommunikation	1 nJ/bit	(passive transmitter, s.u.)	
opt. digitale ASICs:	~ 5 pJ/bit	(LFSR Demonstrator, 1.4V)	

Grenzen des Wachstums

- Jeder exponentielle Verlauf stößt irgendwann an natürliche oder wirtschaftliche Grenzen.
- Beispiel: eine DRAM-Speicherzelle speichert derzeit etwa 100.000 Elektronen. Durch die Verkleinerung werden es mit jeder neuen Technologiestufe weniger.
- Offensichtlich ist die Grenze spätestens erreicht, wenn nur noch ein einziges Elektron gespeichert würde.
- Ab diesem Zeitpunkt gibt es bessere Performance nur noch durch bessere Algorithmen / Architekturen

- Annahme: 50 % Wachstum pro Jahr, $a^b = \exp(b \cdot \ln a)$
- Elektronen pro Speicherzelle: $100000/(1.5^{x/\text{Jahre}}) \geq 1$.
- $x = \ln(100.000)/\ln(1.5) \approx 28$ Jahre

Roadmap: ITRS

International **T**echnology **R**oadmap for **S**emiconductors

`http://www.itrs.net/reports.html`

- non-profit Organisation
- diverse Fördermitglieder
  - Halbleiterhersteller
  - Geräte-Hersteller
  - Unis, Forschungsinstitute
  - Fachverbände aus USA, Europa, Asien
- Jährliche Publikation einer langjährigen Vorhersage
- Zukünftige Entwicklung der Halbleitertechnologie
- Komplexität typischer Chips (Speicher, Prozessoren, SoC, ...)
- Modellierung, Simulation, Entwurfssoftware

Moore's Law: Schöpferische Pause

Beispiel für die Auswirkung von Moore's Law.

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

Wie lösen wir das Problem ?

Moore's Law: Schöpferische Pause

Beispiel für die Auswirkung von Moore's Law.

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

Ein mögliches Vorgehen ist dann das folgende:

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

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
3.3 Moore's Law - Roadmap und Grenzen des Wachstums
64-040 Rechnerstrukturen

Moore's Law: Schöpferische Pause

Beispiel für die Auswirkung von Moore's Law.

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

Ein mögliches Vorgehen ist dann das folgende:

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

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

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

Wie geht es jetzt weiter?

Ab jetzt erst mal ein *bottom-up* Vorgehen: Start mit grundlegenden Aspekten, dann Kennenlernen aller Komponenten des Digitalrechners und Konstruktion eines vollwertigen Rechners.

- Grundlagen der Repräsentation von Information
- Darstellung von Zahlen und Zeichen
- arithmetische und logische Operationen
- ...

- Vorkenntnisse nicht nötig (aber hilfreich)

Gliederung

Gliederung (cont.)

Information

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

Repräsentation (Beispiele)

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

- Text deutsch: fünfundzwanzig
- Text englisch: twentyfive

...

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

...

- Morse-Code: ..---- .....

Information vs. Interpretation

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

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

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

- Repräsentation ist häufig mehrstufig, z.B.

Zahl:	Dezimalzahl	347
Ziffer:	4-bit binär	0011 0100 0111
Bit:	elektrische Spannung	0.1V 0.1V 3.3V 3.3V ...

Repräsentation vs. Ebenen

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

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

- Beispiel: Repräsentation der Zahl $\pi = 3.1415\ldots$ im
  - x86 Prozessor — 80-bit Binärdaten, Spannungen
  - Hauptspeicher — 64-bit Binärdaten, Spannungen
  - Festplatte — codierte Zahl, magnetische Bereiche
  - CD-ROM — codierte Zahl, Land/Pits-Bereiche
  - Papier — Text, „3.14159265...“
  - ...

Information vs. Nachricht

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

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

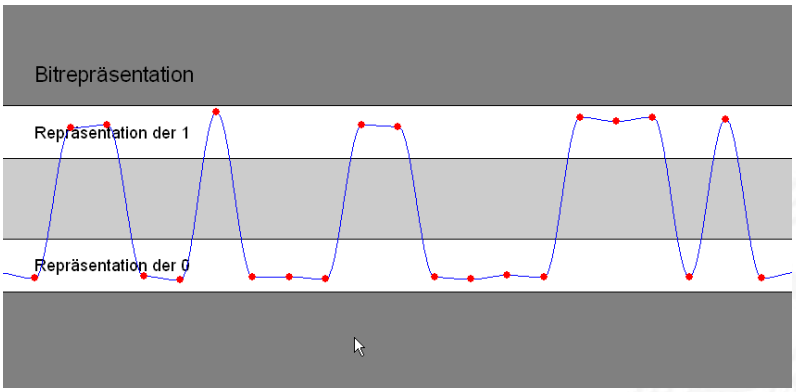
⇒ **Interpretation**: Es handelt sich um drei **Nachrichten**, die jeweils zwei verschiedene **Informationen** enthalten

Information vs. Repräsentation

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

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

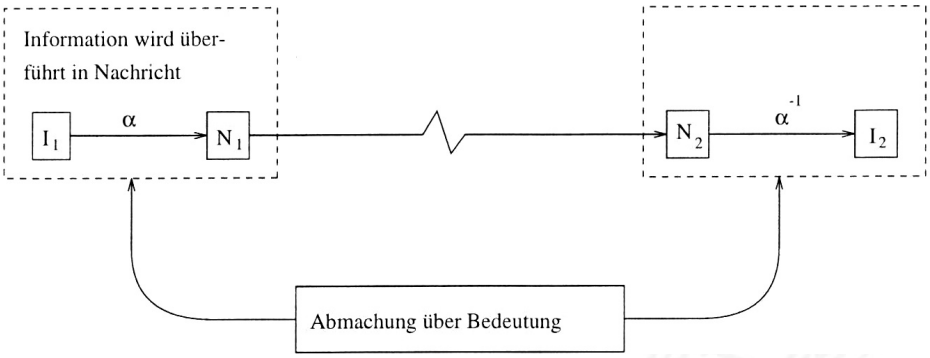
Beispiel: Binärwerte in 5 V-CMOS-Technologie



Klaus von der Heide,
Interaktives Skript T1, demobitrep

- Spannungsverlauf des Signals ist kontinuierlich
- Abtastung zu bestimmten Zeitpunkten
- Quantisierung über abgegrenzte Wertebereiche:
  - $0.0\ V \leq a(t) \leq 1.2\ V$: Interpretation als 0
  - $3.3\ V \leq a(t) \leq 5.0\ V$: Interpretation als 1
  - außerhalb und innerhalb: ungültige Werte

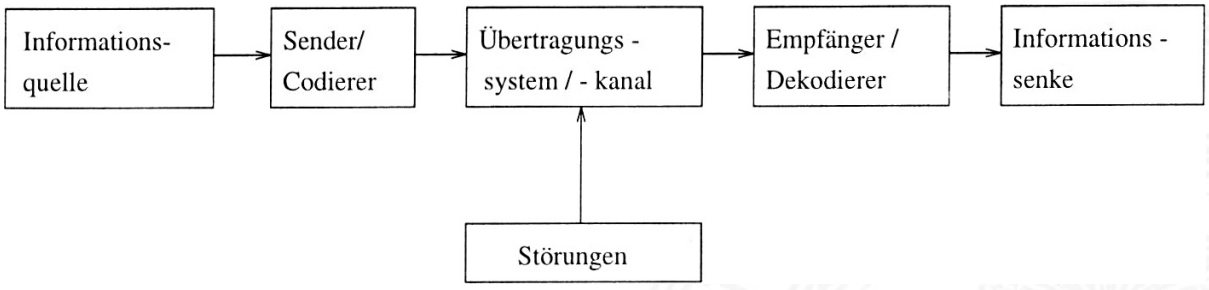
Modell der Informationsübertragung



Beschreibung der **Informationsübermittlung**:

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

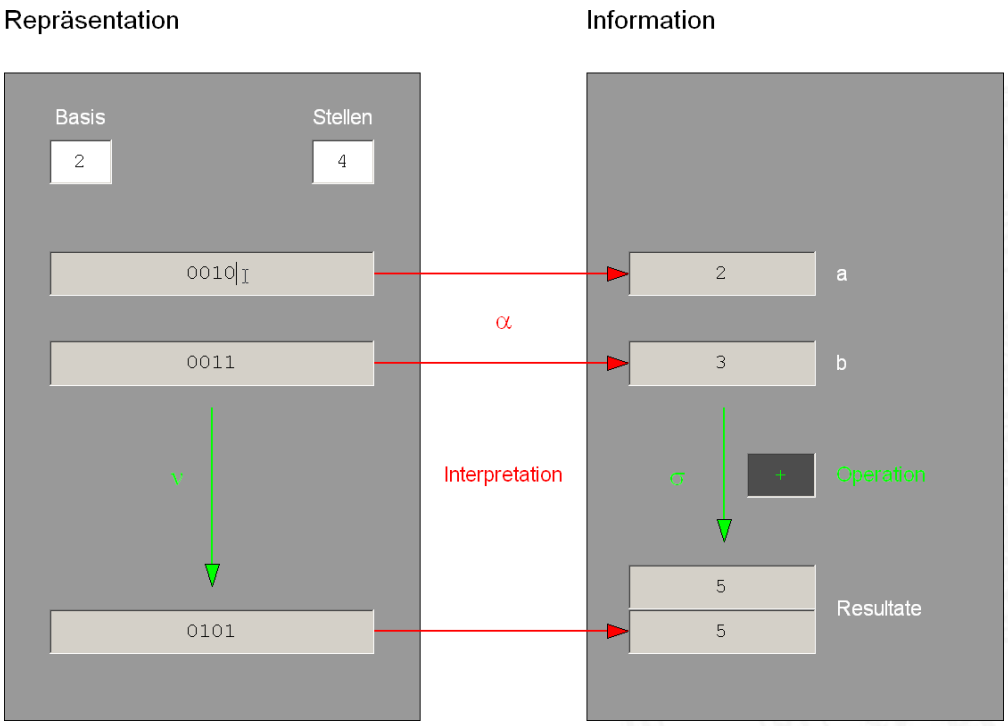
Nachrichtentechnisches Modell der Informationsübertragung



Beispiele für **Störungen**:

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

Demo: Information vs. Repräsentation



Klaus von der Heide, Interaktives Skript T1, inforepres

Informationstreue

Ergibt α gefolgt von σ dasselbe wie ν gefolgt von α', dann heißt ν **informationstreu**.

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

- σ injektiv: **Umschlüsselung**
- ν injektiv: **Umcodierung**
- σ innere Verknüpfung der Menge $\mathcal{J}$ und ν innere Verknüpfung der Menge $\mathcal{R}$: dann ist α ein Homomorphismus der algebraischen Strukturen $(\mathcal{J}, \sigma)$ und $(\mathcal{R}, \nu)$.
- σ bijektiv: Isomorphismus

Informationstreue (cont.)

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

Beispiele:

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

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

Beschreibung von Information durch Zeichen

- **Zeichen** (engl. *character*): Element z aus einer zur Darstellung von Information vereinbarten, einer Abmachung unterliegenden, endlichen Menge Z von Elementen.
- Die Menge heißt **Zeichensatz** oder **Zeichenvorrat** (engl. *character set*).
- Beispiel:
  - $\mathcal{Z}_1 = \{0, 1\}$
  - $\mathcal{Z}_2 = \{0, 1, 2, \ldots, 9, A, B, C, D, E, F\}$
  - $\mathcal{Z}_3 = \{\alpha, \beta, \gamma, \ldots, \omega\}$
  - $\mathcal{Z}_4 = \{\mathrm{CR}, \mathrm{LF}\}$

Binärzeichen

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

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

Alphabet

- **Alphabet** (engl. *alphabet*): Ein in vereinbarter Reihenfolge geordneter Zeichenvorrat $\mathcal{A} = \mathcal{Z}$
- Beispiel:
  - $\mathcal{A}_1 = \{0,1,2,\ldots, 9\}$
  - $\mathcal{A}_2 = \{\text{So,Mo,Di,Mi,Do,Fr,Sa}\}$
  - $\mathcal{A}_3 = \{\text{'A', 'B', } \ldots, \text{'Z'}\}$
- **Numerischer Zeichensatz**: Zeichenvorrat aus Ziffern und/oder Sonderzeichen zur Darstellung von Zahlen
- **Alphanumerischer Zeichensatz**: Zeichensatz aus (mindestens) den Dezimalziffern und den Buchstaben des gewöhnlichen Alphabets, meistens auch mit Sonderzeichen (Leerzeichen, Punkt, Komma usw.)

Zeichenkette

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

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

Gliederung

Gliederung (cont.)

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
5 Zahldarstellung
64-040 Rechnerstrukturen

Gliederung (cont.)

Darstellung von Zahlen und Zeichen: Übersicht

- Natürliche Zahlen (engl. *integer numbers*)
- Festkommazahlen (engl. *fixed point numbers*)
- Gleitkommazahlen (engl. *floating point numbers*)

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

- Pointer (Referenzen, Maschinenadressen)

Konzept der Zahl

- das Messen ist der Ursprung der Zahl
- als Abstraktion der Anzahl von Objekten
- die man abzählen kann
- Anwendung des Distributivgesetzes:

  2 Äpfel + 5 Äpfel = 7 Äpfel
  2 Birnen + 5 Birnen = 7 Birnen
  ...
  $\Rightarrow 2 + 5 = 7$

Eigenschaften eines Zahlensystems

- Zahlenbereich: kleinste und größte darstellbare Zahl?
- Darstellung negativer Werte?
- Darstellung gebrochener Werte?
- Darstellung 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?

Literaturtipp

Georges Ifrah, *Universalgeschichte der Zahlen*, 1998

Klassifikation verschiedener Zahlensysteme

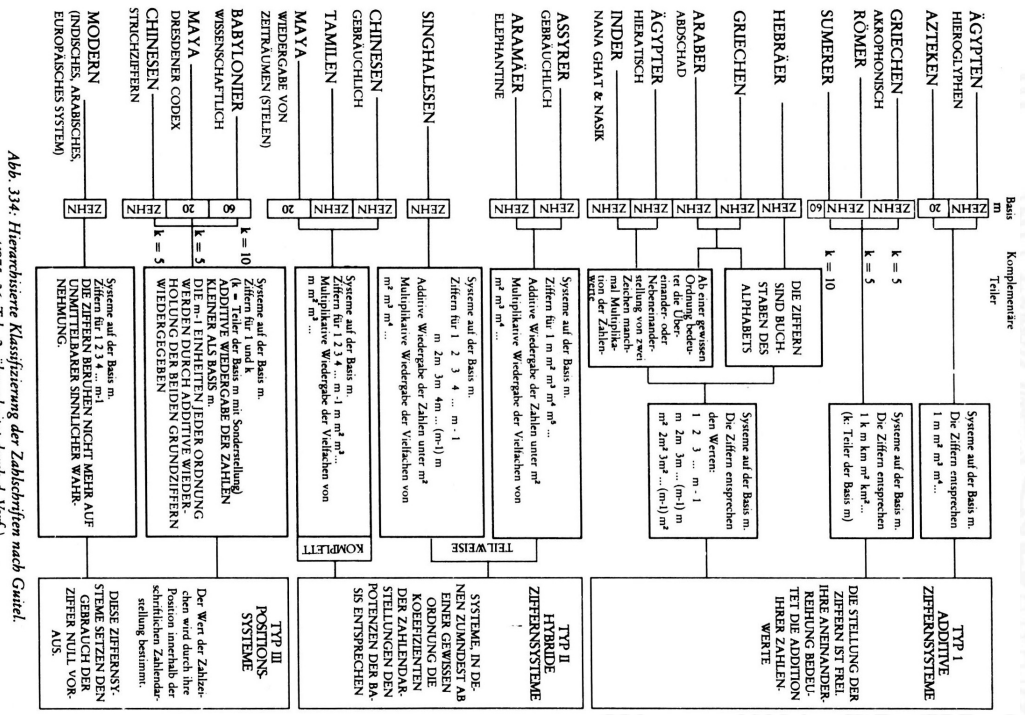


Abb. 334: Hierarchisierte Klassifizierung der Zahlschriften nach Guitel. (1975, 36, Tab. 2; überarbeitet durch d. Verf.)

Direkte Wahrnehmung vs. Zählen

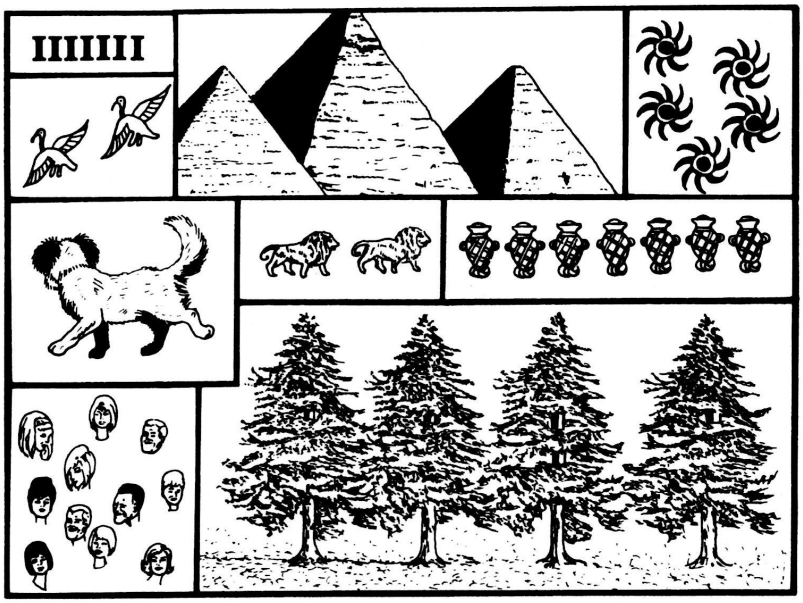


Abb. 1: Auf einen Blick können wir mit unserer direkten Zahlenwahrnehmung feststellen, ob eine Gesamtheit ein, zwei, drei oder vier Elemente umfaßt; Mengen, die größer sind, müssen wir meistens »zählen« – oder mit Hilfe des Vergleichs oder der gedanklichen Aufteilung in Teilmengen erfassen –, da unsere direkte Wahrnehmung nicht mehr ausreicht, exakte Angaben zu machen.

Abstraktion: Verschiedene Symbole für eine Zahl

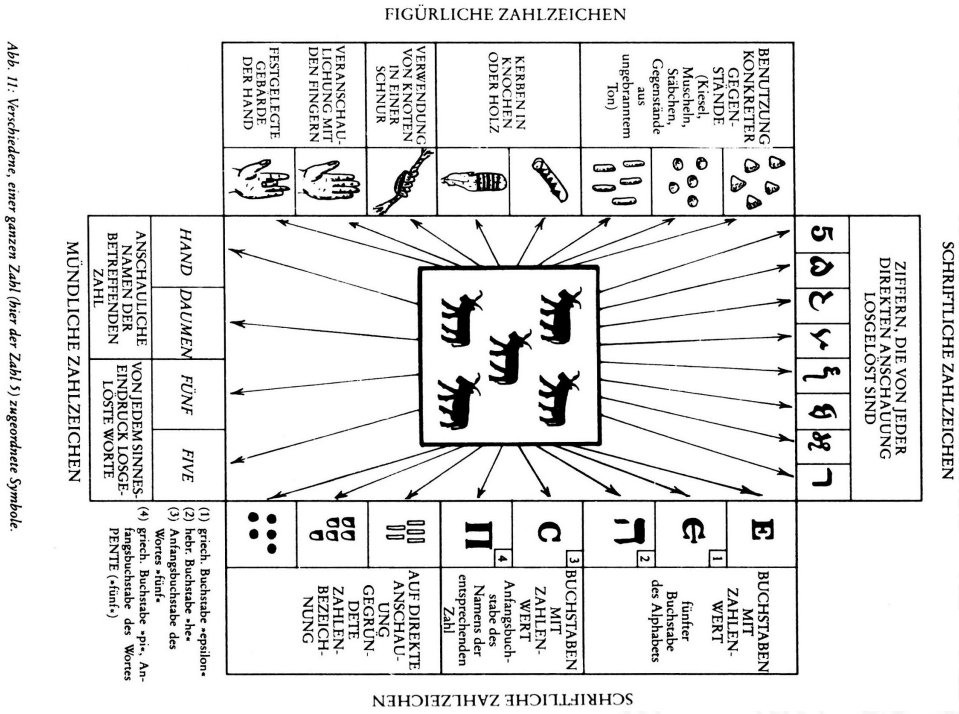
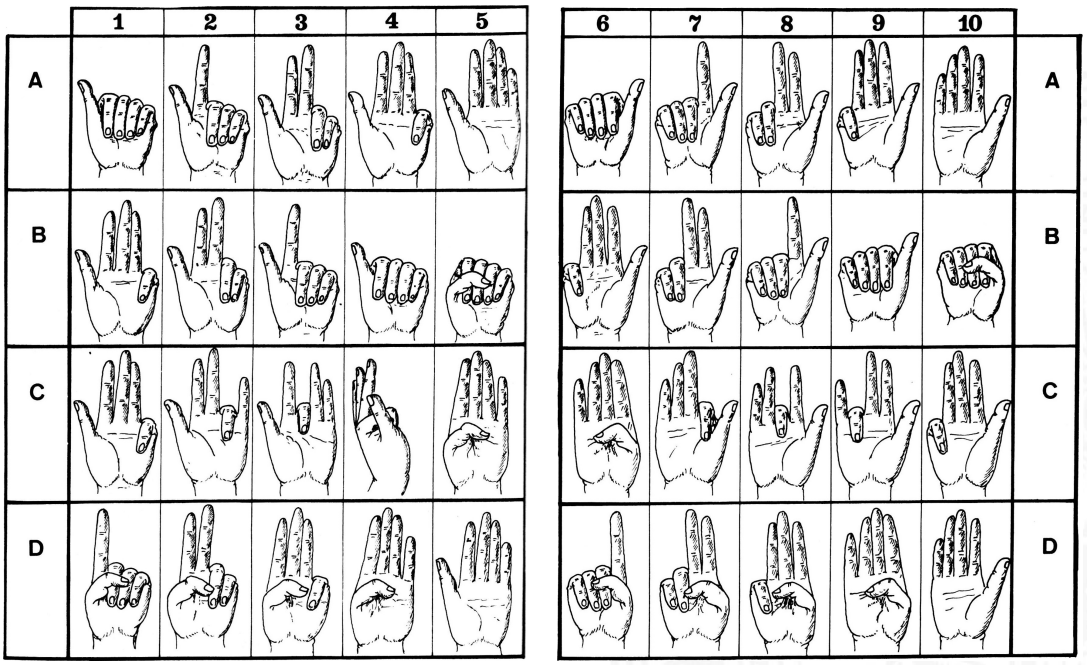


Abb. 11: Verschiedene, einer ganzen Zahl (hier der Zahl 5) zugeordnete Symbole.

Zählen mit den Fingern („digits“)



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)

48 Tonkügelchen im Inneren: tamper-proof

Speicherung: Kerbhölzer

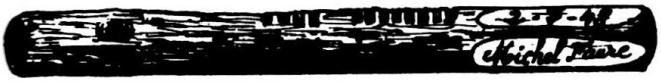


Abb. 58: Kerbhölzer aus Bäckereien in Frankreich, wie sie in kleinen Ortschaften auf dem Lande üblich waren.

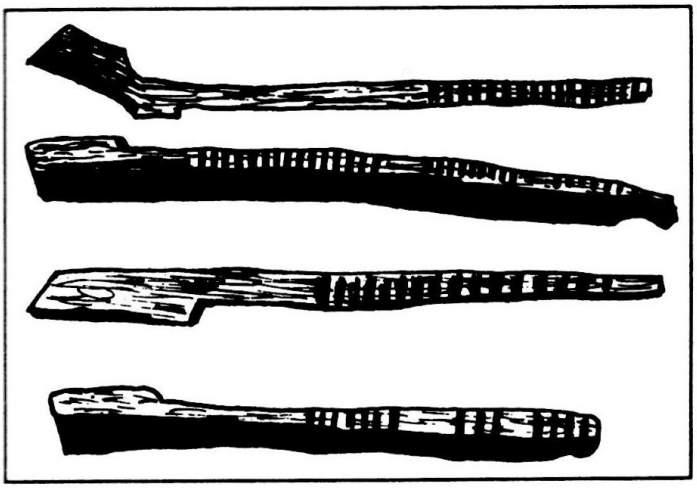


Abb. 59: Englische Kerbhölzer aus dem 13. Jahrhundert. (Sammlung Society of Antiquaries, London; Zeichnung nach Menninger 1957/58, II, 42)

Speicherung: Knotenschnüre

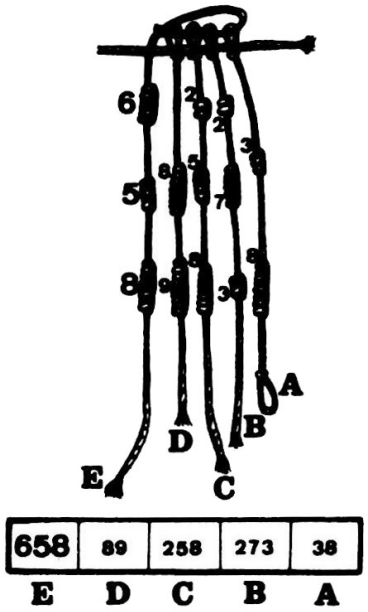


Abb. 66: Interpretation eines quipu: Die Zahl 658 auf der Schnur E ist gleich der Summe der Zahlen auf den Schnüren A, B, C und D. Dieses Bündel ist das erste an einem peruanischen quipu. (American Museum of Natural History, New York, B 8713; vgl. Leland Locke 1923)

Rechnen: Römische Ziffern

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

– keine Symbole zur Darstellung großer Zahlen
– Rechenoperationen so gut wie unmöglich

Römischer Abakus

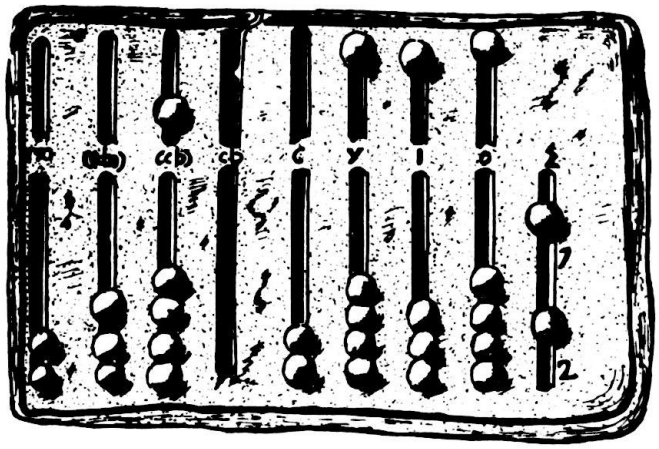
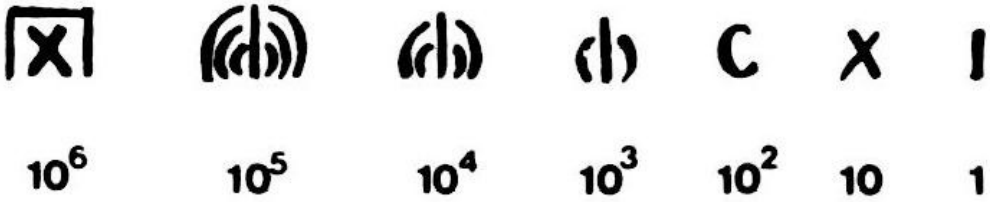


Abb. 87: Römischer Handabakus. (Cabinet des Médailles, Bibliothèque Nationale Paris, br. 1925)



Römischer Abakus

Dagegen können im Rahmen einer entwickelten Stellenwertschrift nicht nur alle beliebigen Zahlen jeder Größenordnung mit einer beschränkten Anzahl von Ziffern dargestellt werden, sondern mit ihr kann auch sehr einfach gerechnet werden. Und eben deshalb ist unser Ziffernsystem eine der Grundlagen der geistigen Fähigkeiten der modernen Menschen.

Als Beweis dafür führen wir mit römischen Ziffern eine einfache Addition durch:

CCLXVI	266
MDCCCVII	1 807
DCL	650
MLXXX	1 080
MMMDCCCIII	3 803

Ohne Übertragung auf unsere Zahlschrift wäre das sehr schwierig, wenn nicht unmöglich – und dabei handelt es sich doch bloß um eine Addition! Wie verhielte sich das erst bei einer Multiplikation oder gar bei einer Division? Mit diesen Ziffernsystemen kann nicht gerechnet werden, da ihre Grundziffern einen festgelegten Zahlenwert haben. Diese Ziffern sind keine Recheneinheiten, sondern Abkürzungen, mit denen Ergebnisse von Rechnungen festgehalten werden können, die mit Gegenständen auf der Rechentafel, dem Abakus oder dem Kugelbrett bereits gelöst worden waren.

Das Stellenwertsystem („Radixdarstellung“)

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

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

b Basis a_i Koeffizient an Stelle i

- universell verwendbar, für beliebig große Zahlen

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

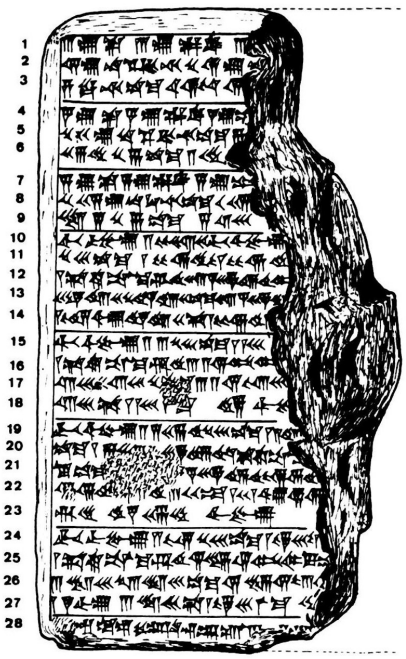
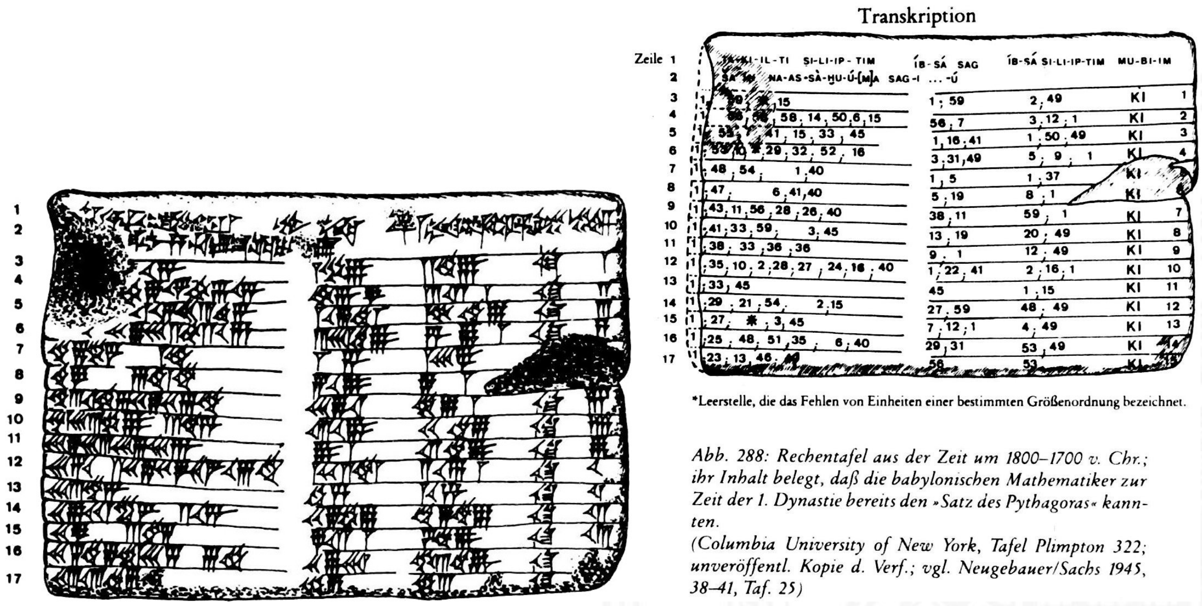


Abb. 289: Mathematische Tafel aus Uruk; sie wurde bei Schwarzgrabungen gefunden und stammt aus dem 2. oder 3. Jh. v. Chr. Es handelt sich um eines der ältesten bekannten Zeugnisse für die Verwendung der babylonischen Null. (Musée du Louvre, Taf. AO 6484, Rückseite; Thureau-Dangin 1922, Nr. 33, Taf. 62;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

Babylon: Beispiel mit Satz des Pythagoras



*Leerstelle, die das Fehlen von Einheiten einer bestimmten Größenordnung bezeichnet.

Abb. 288: Rechentafel aus der Zeit um 1800–1700 v. Chr.; ihr Inhalt belegt, daß die babylonischen Mathematiker zur Zeit der 1. Dynastie bereits den »Satz des Pythagoras« kannten.
(Columbia University of New York, Tafel Plimpton 322; unveröffentl. Kopie d. Verf.; vgl. Neugebauer/Sachs 1945, 38–41, Taf. 25)

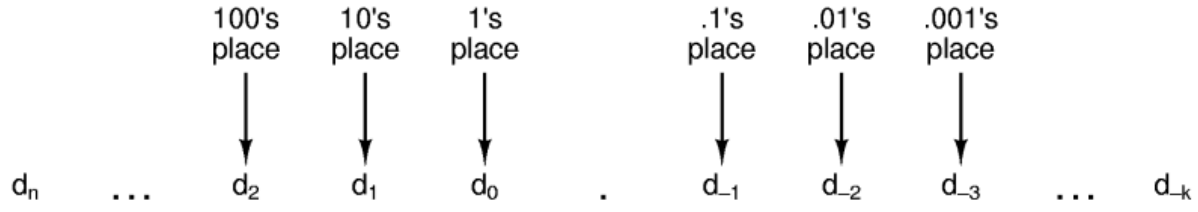
Babylon: Sexagesimalsystem

- Einführung vor ungefähr 4000 Jahren, erstes Stellenwertsystem
- Basis 60
- zwei Symbole: | = 1 und < = 10
- Einritzen gerader und gewinkelter Striche auf Tontafeln
- Null bekannt, aber nicht mitgeschrieben
- Leerzeichen zwischen zwei Stellen
- Beispiele für Zahlen:
  - ||||| 5
  - <<||| 23
  - | <<< $90 = 1 \cdot 60 + 3 \cdot 10$
  - | <<| $3621 = 1 \cdot 3600 + 0 \cdot 60 + 2 \cdot 10 + 1$
- für Zeitangaben und Winkeleinteilung heute noch in Gebrauch

Babylon: Beispiel Potenztabelle 100^i

Klaus von der Heide
Interaktives Skript T1
powersbabylon

Dezimalsystem



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

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

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
    s.o.: *Kapitel 4: Information – Binärwerte...*
  - 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	1024
...	...	...

Addition im Dualsystem

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

+	0	1
0	0	1
1	1	10

- Beispiel

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

Multiplikation im Dualsystem

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

- Multiplikationsmatrix ist sehr einfach:

×	0	1
0	0	0
1	0	1

Multiplikation im Dualsystem (cont.)

- Beispiel

$$
\begin{array}{rl}
10110011 & \times\ 1101 \\
\hline
10110011 & 1 \\
10110011 & 1 \\
00000000 & 0 \\
10110011 & 1 \\
\hline
\text{Ü}\ 11101111 & \\
\hline
100100010111 &
\end{array}
\quad = 179 \cdot 13 \quad = 2327
$$

$$= 1001\,0001\,0111$$

$$= 0x917$$

Oktalsystem

- Basis 8
- Zeichensatz ist $\{0, 1, 2, 3, 4, 5, 6, 7\}$
- C-Schreibweise mit führender Null als Präfix:
  - $0001 = 1_{10}$
    $0013 = 11_{10} = 1 \cdot 8 + 3$
    $0375 = 253_{10} = 3 \cdot 64 + 7 \cdot 8 + 5$
    usw.
  - $\Rightarrow$ Hinweis: also führende Null in C für Dezimalzahlen unmöglich
- für Menschen leichter lesbar als Dualzahlen
- Umwandlung aus/vom Dualsystem durch Zusammenfassen bzw. Ausschreiben von je drei Bits:
  $00 = 000, 01 = 001, 02 = 010, 03 = 011,$
  $04 = 100, 05 = 101, 06 = 110, 07 = 111$

Hexadezimalsystem

- Basis 16
- Zeichensatz ist $\{0, 1, 2, \ldots, 9, A, B, C, D, E, F\}$
- C-Schreibweise mit Präfix $0x$ – Klein- oder Großbuchstaben
  - $0x00000001 = 1_{10}$
    $0x000000fe = 254_{10} = 15 \cdot 16 + 14$
    $0x0000ffff = 65535_{10} = 15 \cdot 4096 + 15 \cdot 256 + 15 \cdot 16 + 15$
    $0xcafebabe = \ldots$ erstes Wort in Java Class-Dateien
    usw.
- viel leichter lesbar als entsprechende Dualzahl
- Umwandlung aus/vom Dualsystem: Zusammenfassen bzw. Ausschreiben von je vier Bits:
  $0x0 = 0000, 0x1 = 0001, 0x2 = 0010, \ldots, 0x9 = 1001,$
  $0xA = 1010, 0xB = 1011, 0xC = 1100, 0xD = 1101, 0xE = 1110, 0xF = 1111$

Beispiel: Darstellungen der Zahl 2001

Binary 1 1 1 1 1 0 1 0 0 0 1

$1\times2^{10} + 1\times2^{9} + 1\times2^{8} + 1\times2^{7} + 1\times2^{6} + 0\times2^{5} + 1\times2^{4} + 0\times2^{3} + 0\times2^{2} + 0\times2^{1} + 1\times2^{0}$

1024 + 512 + 256 + 128 + 64 + 0 + 16 + 0 + 0 + 0 + 1

Octal 3 7 2 1

$3\times8^{3} + 7\times8^{2} + 2\times8^{1} + 1\times8^{0}$

1536 + 448 + 16 + 1

Decimal 2 0 0 1

$2\times10^{3} + 0\times10^{2} + 0\times10^{1} + 1\times10^{0}$

2000 + 0 + 0 + 1

Hexadecimal 7 D 1

$7\times16^{2} + 13\times16^{1} + 1\times16^{0}$

1792 + 208 + 1

Umrechnung Dual-/Oktal-/Hexadezimalsystem

Example 1

Hexadecimal		1		9		4		8	.	B		6
Binary	0	0	0	1	1	0	0	1	0	1	0	0	0	.	1	0	1	1	0	0
Octal		1		4		5		1	0	.	5		5		4					

Example 2

Hexadecimal		7		B		A		3	.	B		C		4
Binary	0	1	1	1	1	0	1	1	1	0	1	0	0	0	1	1	.	1	0	1	1	1	0	0	0	1	0	0
Octal		7		5		6		4		3	.	5		7		0		4										

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

Umrechnung zwischen verschiedenen Basen

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

Basis: 60

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

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

Umwandlung über Potenztabellen

Vorgehensweise für Integerzahlen:

- Subtraktion des größten Vielfachen einer Potenz des Zielsystems (gemäß der vorberechneten Potenztabelle) von der umzuwandelnden Zahl
- Notation dieses größten Vielfachen (im Zielsystem)
- Subtraktion wiederum des größten Vielfachen vom verbliebenen Rest
- Addition des zweiten Vielfachen zum ersten
- Wiederholen, bis Rest $= 0$

Potenztabellen Dual/Dezimal

Stelle	Wert
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024
...	

Stelle	Wert im Dualsystem
10^0	1
10^1	1010
10^2	110 0100
10^3	111 1101 0000
10^4	10 0111 0001 0000
10^5	0x186A0
10^6	0xF4240
10^7	0x989680
10^8	0x5F5E100
10^9	0x369ACA00
10^{10}	

Potenztabellen: Beispiel

Beispiel: Umwandlung Dezimal- in Dualzahl

(Verwendung von Potenztabelle 1.3.1.1/1)

Annahme: $z = (163)_{10}$

163			
-128	(2^7)		1000.0000
35			
-32	(2^5)	+	10.0000
3			
-2	(2^1)	+	10
1			
-1	(2^0)	+	1
0			1010.0011

$(163)_{10} \longleftrightarrow (1010.0011)_2$

Potenztabellen: Beispiel (cont.)

Beispiel: Umformung Dual- in Dezimalzahl

gegeben: $z = (1010.0011)_2$

	1010.0011				
-	110.0100	$(100)_{10}$	$= 1 \cdot 100$	$=$	$1 \cdot 10^2$
	0011.1111				
-	11.1100	$(60)_{10}$	$= 6 \cdot 10$	$=$	$6 \cdot 10^1$
	11				
-	11	$(3)_{10}$	$= 3 \cdot 1$	$=$	$3 \cdot 10^0$
	0	$(163)_{10}$			

Divisionsrestverfahren

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

$$\begin{array}{rcrl} 163 & : & 2 = 81 & Rest\ 1 \\ 81 & : & 2 = 40 & Rest\ 1 \\ 40 & : & 2 = 20 & Rest\ 0 \\ 20 & : & 2 = 10 & Rest\ 0 \\ 10 & : & 2 = 5 & Rest\ 0 \\ 5 & : & 2 = 2 & Rest\ 1 \\ 2 & : & 2 = 1 & Rest\ 0 \\ 1 & : & 2 = 0 & Rest\ 1 \end{array} \uparrow$$

$(163)_{10} \longleftrightarrow (1010.0011)_2$

Divisionsrestverfahren: Beispiel

Beispiel: Umwandlung Dual- in Dezimalzahl:

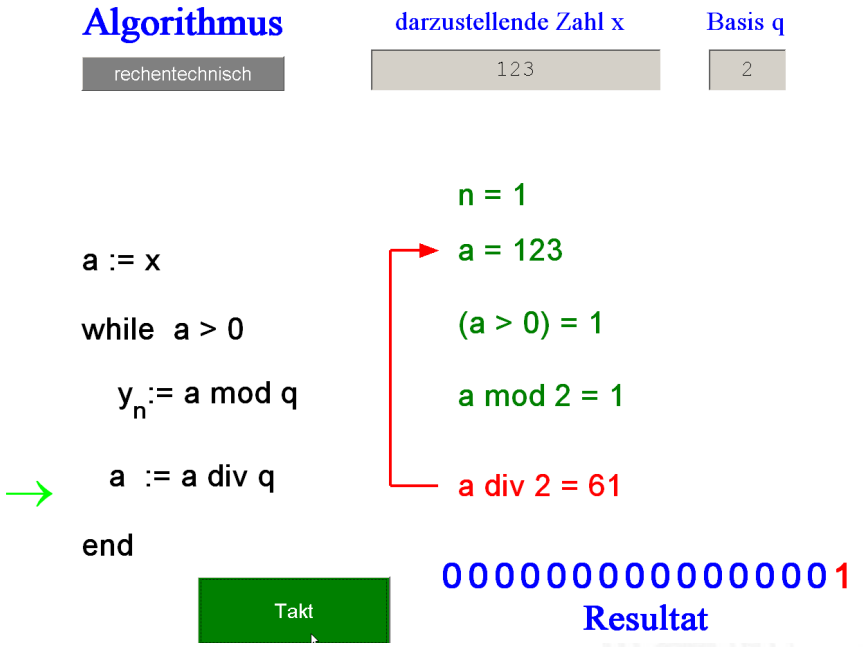
(wie oben)

$$\begin{array}{rcrl} (1010.0011)_2 & : & (1010)_2 = 1.0000 & Rest\ (11)_2 \mathrel{\widehat{=}} (3)_{10} \\ (1.0000)_2 & : & (1010)_2 = 1 & Rest\ (110)_2 \mathrel{\widehat{=}} (6)_{10} \\ (1)_2 & : & (1010)_2 = 0 & Rest\ (1)_2 \mathrel{\widehat{=}} (1)_{10} \end{array} \uparrow$$

$(1010.0011)_2 \longleftrightarrow (163)_{10}$

Hinweis: Division in Basis *b* folgt

Demo: Interaktives Skript, stellen2stellen



Klaus von der Heide
Interaktives Skript T1
stellen2stellen

Horner-Schema

- Darstellung einer Potenzsumme durch ineinander verschachtelte Faktoren

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

Vorgehensweise:

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

Horner-Schema: Beispiel

gegeben: $(163)_{10}$
$163 = (1 \cdot 10 + 6) \cdot 10 + 3$

Umsetzung von Faktoren und Summanden ins Zielzahlensystem:

$$\begin{aligned}(10)_{10} &\longleftrightarrow (1010)_2\\ (1)_{10} &\longleftrightarrow (0001)_2\\ (6)_{10} &\longleftrightarrow (0110)_2\\ (3)_{10} &\longleftrightarrow (0011)_2\end{aligned}$$

Durchführung der arithmetischen Operation

$$\begin{array}{rcl} 0001 \cdot 1010 &=& \ 1010\\ &+& \underline{\ 0110}\\ \multicolumn{3}{r}{1.0000 \cdot 1010 = 1010.0000}\\ &&+\ \ 0011\\ &&\overline{1010.0011}\end{array}$$

Horner-Schema: Beispiel (cont.)

Rückumwandlung von Dual- in Dezimalzahl

$$\begin{aligned}(1010.0011)_2 = ((((((1 \cdot 10_2) + 0) \cdot 10_2 + 1) \cdot 10_2 + 0) \cdot 10_2 + 0) \cdot 10_2\\ +0) \cdot 10_2 + 1) \cdot 10_2 + 1\end{aligned}$$

Umsetzung von Faktoren und Summanden

$$\begin{aligned}(0)_2 &\longleftrightarrow (0)_{10}\\ (1)_2 &\longleftrightarrow (1)_{10}\\ (10)_2 &\longleftrightarrow (2)_{10}\end{aligned}$$

Horner-Schema: Beispiel (cont.)

Berechnung:

$$
\begin{array}{rrrrrrrrrrrrrrrrrrrrr}
1 & \cdot 2 & = & 2 & & & & & & & & & & & & & & & & & \\
 & + & & \underline{0} & & & & & & & & & & & & & & & & & \\
 & & 2 & \cdot 2 & = & 4 & & & & & & & & & & & & & & & \\
 & & & + & & \underline{1} & & & & & & & & & & & & & & & \\
 & & & & 5 & \cdot 2 & = & 10 & & & & & & & & & & & & & \\
 & & & & & + & & \underline{0} & & & & & & & & & & & & & \\
 & & & & & & 10 & \cdot 2 & = & 20 & & & & & & & & & & & \\
 & & & & & & & + & & \underline{0} & & & & & & & & & & & \\
 & & & & & & & & 20 & \cdot 2 & = & 40 & & & & & & & & & \\
 & & & & & & & & & + & & \underline{0} & & & & & & & & & \\
 & & & & & & & & & & 40 & \cdot 2 & = & 80 & & & & & & & \\
 & & & & & & & & & & & + & & \underline{1} & & & & & & & \\
 & & & & & & & & & & & & 81 & \cdot 2 & = & 162 & & & & & \\
 & & & & & & & & & & & & & + & & \underline{1} & & & & & \\
 & & & & & & & & & & & & & & & \underline{\underline{163}} & & & & &
\end{array}
$$

Zahlenbereich bei fester Wortlänge

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

Präfixe

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

- Lichtgeschwindigkeit: 300 000 km/s = 30 cm/ns
- Ruheenergie des Elektrons: 0,51 MeV
- Strukturbreite heutiger Mikrochips: 32 nm
- usw.

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

Präfixe für Einheiten im Dezimalsystem

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

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

Präfixe für Einheiten im Dualsystem

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

Beispiele:

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

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

Präfixe für Einheiten im Dualsystem

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

- DRAM-Modul mit 1 GB Kapazität: gemeint sind 2^{30} Bytes
- Flash-Speicherkarte 4 GB Kapazität: gemeint sind 2^{32} Bytes

- Festplatte mit Angabe 1 TB Kapazität: typisch 10^{12} Bytes
- die tatsächliche angezeigte verfügbare Kapazität ist oft geringer, weil das jeweilige Dateisystem Platz für seine eigenen Verwaltungsinformationen belegt.

Festkommadarstellung

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

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

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

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

Nachkommastellen im Dualsystem

- $2^{-1} = 0.5$
  $2^{-2} = 0.25$
  $2^{-3} = 0.125$
  $2^{-4} = 0.0625$
  $2^{-5} = 0.03125$
  $2^{-6} = 0.015625$
  $2^{-7} = 0.0078125$
  ...
- alle Dualbrüche sind im Dezimalsystem exakt darstellbar (d.h. mit endlicher Wortlänge)

Beispiel: Umrechnung dezimal 0.3 nach dual

Betrachtung von gebrochenen Zahlen im Dualsystem:

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:

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

Beispiel: Dezimalbrüche, eine Nachkommastelle

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

B=10	B=2	B=2	B=10
0,1	$0, 0\overline{0011}$	0,001	0,125
0,2	$0, \overline{0011}$	0,010	0,250
0,3	$0, \overline{01001}$	0,011	0,375
0,4	$0, \overline{0110}$	0,100	0,5
0,5	0,1	0,101	0,625
0,6	$0, \overline{1001}$	0,110	0,750
0,7	$0, \overline{10110}$	0,111	0,875
0,8	$0, \overline{1100}$		
0,9	$0, \overline{11100}$		

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

Darstellung negativer Zahlen: Beispiele

N decimal	N binary	–N signed mag.	–N 1's compl.	–N 2's compl.	–N excess 128
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11011010	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Nonexistent	Nonexistent	Nonexistent	10000000	00000000

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
...			
0111	$+7$	1111	-7

– doppelte Codierung der Null: $+0$, -0
– Rechenwerke für Addition/Subtraktion aufwendig

Exzess-Codierung (Offset-basiert)

- einfache Um-Interpretation der Binärcodierung

$$z = Z - \mathit{offset}$$

- mit z vorzeichenbehafteter Wert, Z binäre Ganzzahl,
- beliebig gewählter Offset

– Null wird also nicht mehr durch 000...0 dargestellt
\+ Größenvergleich zweier Zahlen bleibt einfach

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

Exzess-Codierung: Beispiel

Bitmuster	Binärcode	Exzess-8	Exzess-6	$(z = Z - \text{offset})$
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	

b-Komplement: Definition

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

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

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

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

b-Komplement: Beispiele

$$
\begin{array}{llll}
b = 10 & n = 4 & K_{10}(3763)_{10} & = 10^4 - 3763 \quad = 6237_{10} \\
 & n = 2 & K_{10}(0,3763)_{10} & = 10^2 - 0,3763 \quad = 99,6237_{10} \\
 & n = 0 & K_{10}(0,3763)_{10} & = 10^0 - 0,3763 \quad = 0,6237_{10} \\
b = 2 & n = 2 & K_2(10,01)_2 & = 2^2 - 10,01_2 \quad = 01,11_2 \\
 & n = 8 & K_2(10,01)_2 & = 2^8 - 10,01_2 \quad = 11111101,11_2
\end{array}
$$

$(b-1)$-Komplement: Definition

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

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

- b: die Basis des Stellenwertsystems
- n: Anzahl der zu berücksichtigenden Vorkommastellen
- m: Anzahl der Nachkommastellen

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

$(b-1)$-Komplement: Trick

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

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

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

▶ Dualsystem:	1-Komplement	1100 1001
	alle Bits invertieren	0011 0110
▶ Dezimalsystem:	9-Komplement	24453
	alle Ziffern invertieren	75546
	$0\leftrightarrow9$ $1\leftrightarrow8$ $2\leftrightarrow7$ $3\leftrightarrow6$ $4\leftrightarrow5$	
	Summe:	$99999 = 100000 - 1$

Subtraktion mit b-Komplement

- bei Rechnung mit fester Stellenzahl n gilt:

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

weil b^n gerade nicht mehr in n Stellen hineinpasst**!**

- also gilt für die Subtraktion auch:

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

- Subtraktion kann also durch Addition des b-Komplements ersetzt werden
- und für Integerzahlen gilt außerdem

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

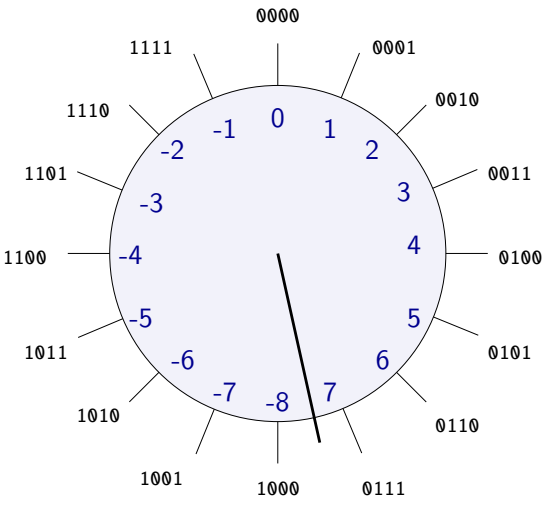
Subtraktion mit Einer- und Zweierkomplement

- Subtraktion ersetzt durch Addition des Komplements

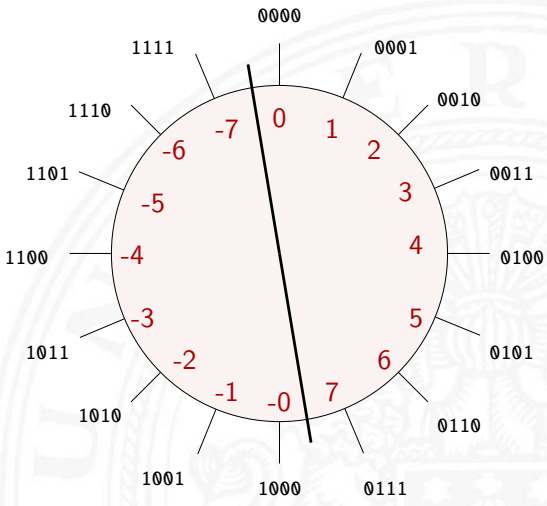
Decimal	1's complement	2's complement
10	00001010	00001010
+ (–3)	11111100	11111101
+7	1 00000110	1 00000111
	carry 1	discarded
	00000111	

Veranschaulichung: Zahlenkreis

Beispiel für w-bit



Zweierkomplement



Betrag und Vorzeichen

Gleitkommaformat

Wie kann man „wissenschaftliche“ Zahlen darstellen?

- Masse der Sonne $1,989 \cdot 10^{30}$ kg
- Masse eines Elektrons 0,00000 00000 00000 00016 g
- Anzahl der Atome pro Mol 6023 00000 00000 00000 00000

  ...

Darstellung im Stellenwertsystem?

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

Gleitkomma: Motivation

Grundidee: **halblogarithmische Darstellung einer Zahl**:

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

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

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

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

Gleitkomma: Dezimalsystem

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

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

- Schreibweise in C/Java: Vorzeichen Mantisse E Exponent

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

Gleitkomma: Beispiel für Zahlenbereiche

Digits in fraction	Digits in exponent	Lower bound	Upper bound
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

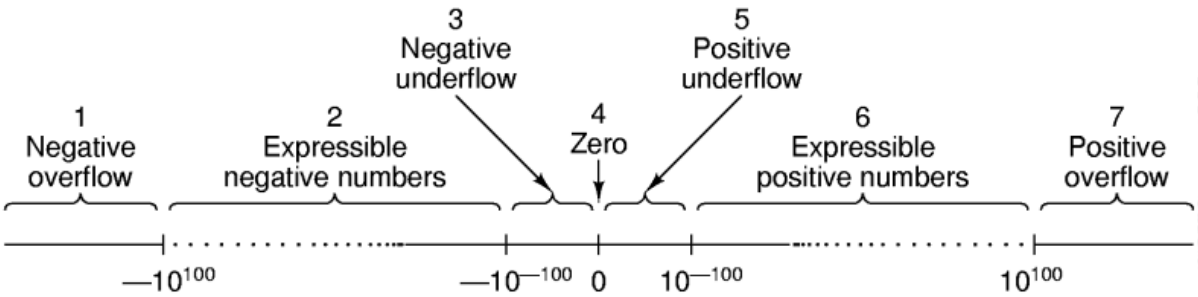
Gleitkomma: Historie

- 1937 Zuse: Z1 mit 22-bit Gleitkomma-Datenformat
- 195x Verbreitung von Gleitkomma-Darstellung für numerische Berechnungen
- 1980 Intel 8087: erster Koprozessor-Chip, ca. 45.000 Transistoren, ca. 50k FLOPS/s
- 1985 IEEE-754 Standard für Gleitkomma
- 1989 Intel 486 mit integriertem Koprozessor
- 1995 Java-Spezifikation fordert IEEE-754
- 1996 ASCI-RED: 1 TFLOPS (9152x PentiumPro)
- 2008 Roadrunner: 1 PFLOPS (12960x Cell)

...

FLOPS := Floating-Point Operations Per Second

Gleitkomma: Zahlenbereiche



- Darstellung üblicherweise als Betrag+Vorzeichen
- negative und positive Zahlen gleichberechtigt (symmetrisch)
- separate Darstellung für den Wert Null
- sieben Zahlenbereiche: siehe Bild
- relativer Abstand benachbarter Zahlen bleibt ähnlich (vgl. dagegen Integer: 0/1, 1/2, 2/3, ..., 65535/65536, ...)

Gleitkomma: Normalisierte Darstellung

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

- diese Darstellung ist bisher nicht eindeutig:

  $123 \cdot 10^0 = 12.3 \cdot 10^1 = 1.23 \cdot 10^2 = 0.123 \cdot 10^3 = \quad \ldots$

normalisierte Darstellung

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

$\Rightarrow$ Darstellung ist dann eindeutig

$\Rightarrow$ im Dualsystem: erstes Vorkommabit ist dann also 1, muss also nicht gespeichert werden

- evtl. zusätzlich sehr kleine Zahlen nicht-normalisiert

IEEE-754 Standard

bis 1985 ein Wildwuchs von Gleitkomma-Formaten:

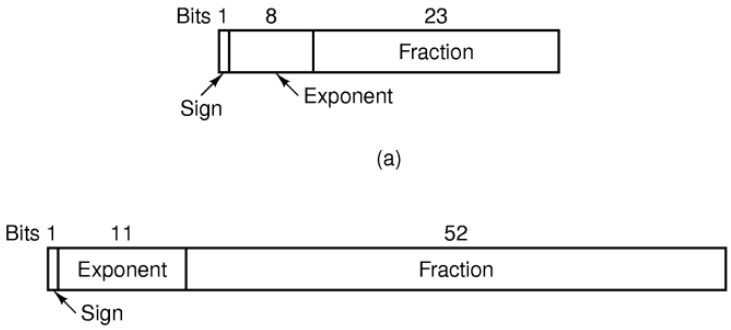
- unterschiedliche Anzahl Bits in Mantisse und Exponent
- Exponent mit Basis 2, 10, oder 16
- diverse Algorithmen zur Rundung
- jeder Hersteller mit eigener Variante

– Numerische Algorithmen nicht portabel

- 1985: Publikation des Standards IEEE-754 zur Vereinheitlichung
- klare Regeln, auch für Rundungsoperationen
- große Akzeptanz, mittlerweile der universale Standard

Details: unter anderem in `en.wikipedia.org/wiki/IEEE_754`

IEEE-754: *float* und *double*



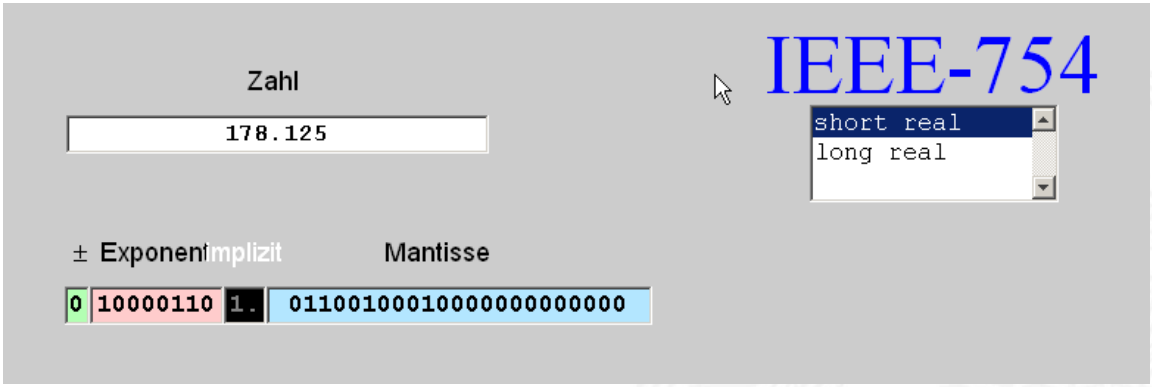
- 32-bit-Format: einfache Genauigkeit (*single precision*, *float*)
- 64-bit-Format: doppelte Genauigkeit (*double precision*, *double*)

- Mantisse als normalisierte Dualzahl: $1 \leq m < 2$
- Exponent in Exzess-127 bzw. Exzess-1023 Codierung
- einige Sonderwerte: Null ($+0$, -0), NaN, Infinity

IEEE-754: Zahlenbereiche

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	–126 to +127	–1022 to +1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

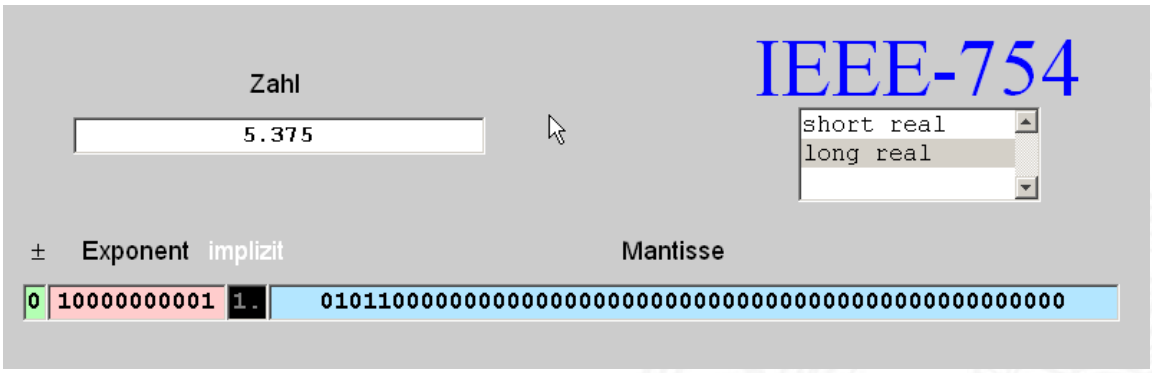
Matlab-Demo: demoieee754



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

Klaus von der Heide, Interaktives Skript T1, demoieee754

Matlab-Demo: demoieee754



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

  Erinnerung: $\ln_2(10) = \ln(10)/\ln(2) \approx 3.321$

Beispiele: float

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

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

- 1 1000 0000 1110 0000 0000 0000 0000 000

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

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

- 0 1111 1110 0001 0011 0000 0000 0000 000

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

  $= 2^{127} \cdot 1,07421875 = 1,953965 \cdot 10^{38}$

Beispiele: float (cont.)

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

- 1 0000 0001 0000 0000 0000 0000 0000 000

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

  $= -1 \cdot 2^{-126} \cdot 1.0 = -1,1755 \cdot 10^{-38}$

- 0 0111 1111 0000 0000 0000 0000 0000 001

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

  $= 1 \cdot (1 + 0,0000001) = 1,0000001$

BitsToFloat.java

```
public static void main( String[] args ) {
  p( "1", "01111111", "00000000000000000000000" );
}
public void p( String s, String e, String m ) {
  int     sign = (Integer.parseInt( s, 2 ) & 0x1) << 31;
  int exponent = (Integer.parseInt( e, 2 ) & 0xFF) << 23;
  int mantisse = (Integer.parseInt( m, 2 ) & 0x007FFFFF);
  int bits = sign | exponent | mantisse;
  float  f = Float.intBitsToFloat( bits );
  System.out.println( dumpIntBits(bits) + " " + f );
}
public String dumpIntBits( int i ) {
  StringBuffer sb = new StringBuffer();
  for( int mask=0x80000000; mask != 0; mask = mask >>> 1 ) {
    sb.append( ((i & mask) != 0) ? "1" : "0" );
  }
  return sb.toString();
}
```


Beispiele: BitsToFloat

```
10111111100000000000000000000000 -1.0
00111111100000000000000000000000 1.0
00111111100000000000000000000001 1.0000001
01000000010000000000000000000000 3.0
01000000011000000000000000000000 3.5
01000000011100000000000000000000 3.75
01000000011111111111111111111111 3.9999998
01000000110000000000000000000000 6.0
00000000100000000000000000000000 1.17549435E-38
11000000011100000000000000000000 -3.75
01111111010000000000000000000000 2.5521178E38
01111111000010011000000000000000 1.8276885E38
01111111011111111111111111111111 3.4028235E38
01111111100000000000000000000000 Infinity
11111111100000000000000000000000 -Infinity
01111111110000000000000000000000 NaN
01111111100001111000000000000000 NaN
```

Gleitkomma: Addition, Subtraktion

Addition von Gleitkommazahlen $y = a_1 + a_2$

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

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

- Beispiele in den Übungen

Gleitkomma-Addition: Beispiel

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

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

- normalerweise nicht informationstreu !

Achtung: Auslöschung

Probleme bei Subtraktion 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 stark ungenau
- Beispiel: (1.0E20 + 3.14159) = 1.0E20

- Fall 2: Exponenten und Mantisse fast gleich
- fast alle Bits der Mantisse löschen sich aus
- Resultat hat nur noch wenige Bits effektiver Genauigkeit

Gleitkomma: Multiplikation, Division

Multiplikation von Gleitkommazahlen $y = a_1 \cdot a_2$

- Multiplikation der Mantissen und Vorzeichen
- Addition der Exponenten
- ggf. Normalisierung des Resultats

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

Division entsprechend:

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

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

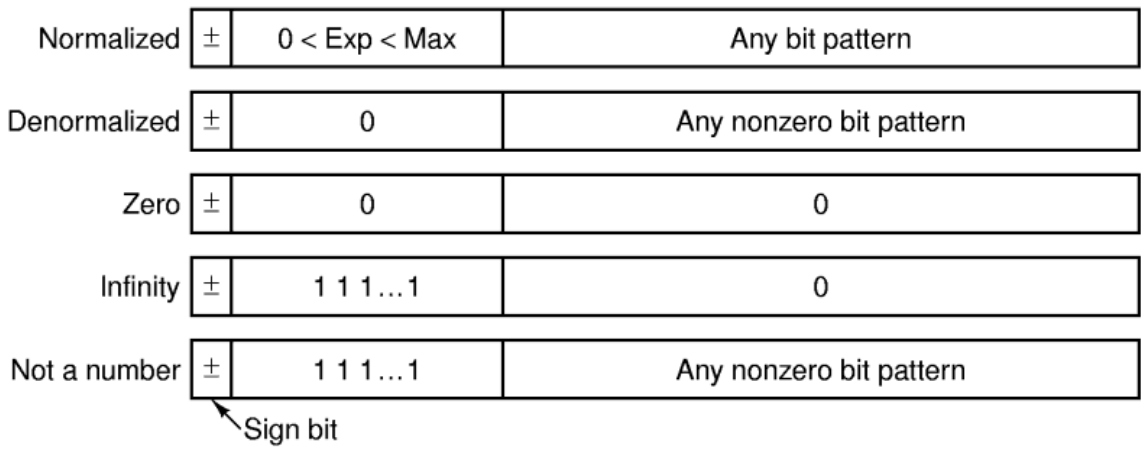
IEEE-754: Infinity, Not-a-Number

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

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

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

IEEE-754: *Inf*, *NaN*, ± 0



- Rechnen mit *Inf* funktioniert normal: $0/Inf = 0$
- jede Operation mit *NaN* liefert wieder *NaN*

IEEE-754: FloatInfNaNDemo.java

```
java FloatInfNaNDemo
 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
sqrt(2) = 1.4142135623730951
sqrt(-1) = NaN
0 + NaN = NaN
NaN == NaN? false              Achtung (!)
Infinity > NaN? false          Achtung (!)
```


ULP: Unit in the last place

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

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

Rundungsfehler

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

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

Datentypen in der Praxis: Maschinenworte

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

- 32-bit Rechner:
  - Wortlänge für Integerwerte ist 32-bit
  - adressierbarer Speicher ist 2^{32} Bytes (4 GiB)
  - bereits zu knapp für speicherhungrige Applikationen
- sukzessive Übergang zu 64-bit Rechnern

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

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

`www.agner.org/optimize/calling_conventions.pdf`

Literatur: Vertiefung

- Klaus von der Heide, *Vorlesung: Technische Informatik 1 — interaktives Skript*, Universität Hamburg, FB Informatik, 2005
  `tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1`
  Float/Double-Demonstration: demoieee754
- Donald E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 0: Introduction to Combinatorial Algorithms and Boolean Functions*, Addison-Wesley, 2008
- Donald E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques, Binary Decision Diagrams*, Addison-Wesley, 2009

Literatur: Vertiefung (cont.)

- David Goldberg, *What every computer scientist should know about floating-point*, 1991
  `www.validlab.com/goldberg/paper.pdf`
- Georges Ifrah, *Universalgeschichte der Zahlen*, div. Verlage, 1998

Gliederung

Gliederung (cont.)

Rechner-Arithmetik

- Wiederholung: Stellenwertsystem
- Addition: Ganzzahlen, Zweierkomplementzahlen
- Überlauf

- Multiplikation
- Division

- Schiebe-Operationen

Wiederholung: Stellenwertsystem

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

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

- b: Basis, a_i Koeffizient an Stelle i
- universell verwendbar, für beliebig große Zahlen

Integer-Datentypen in C und Java

C:

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

Java:

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

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

  etc.
- Werte sind für die Sprache fest definiert

Addition im Dualsystem

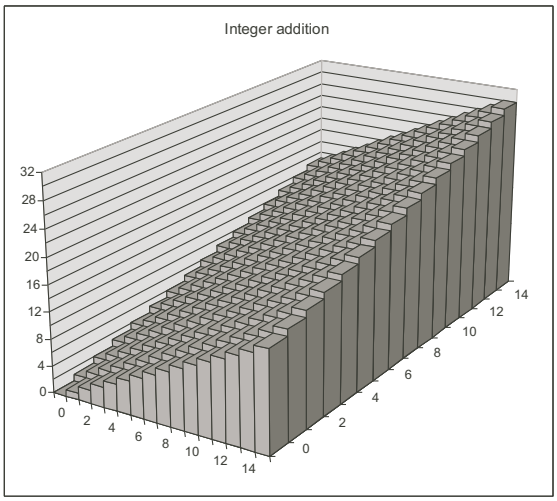
- funktioniert genau wie im Dezimalsystem
- Addition mehrstelliger Zahlen erfolgt stellenweise
- Additionsmatrix:

+	0	1
0	0	1
1	1	10

- Beispiel

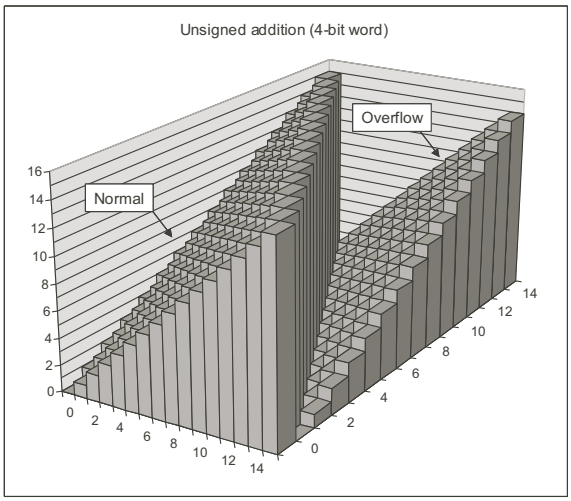
	1011 0011	= 179
+	0011 1001	= 57
Ü	11 11	11
	1110 1100	= 236

Visualisierung: 4-bit Addition



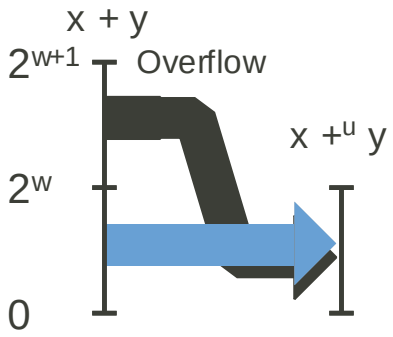
- Wortbreite w, hier 4-bit
- Zahlenbereich der Operanden x, y ist $0 \,..\, (2^w - 1)$
- Zahlenbereich des Resultats s ist $0 \,..\, (2^{w+1} - 2)$

Visualisierung: 4-bit unsigned Addition



- Operanden und Resultat jeweils 4-bit
- Überlauf, sobald das Resultat größer als $(2^w - 1)$
- oberstes Bit geht verloren

Überlauf: unsigned Addition



- Wortbreite w, hier 4-bit
- Zahlenbereich der Operanden x, y ist $0 .. (2^w - 1)$
- Zahlenbereich des Resultats s ist $0 .. (2^{w+1} - 2)$
- Werte $s \geq 2^w$ werden in den Bereich $0 .. 2^w - 1$ abgebildet

Subtraktion im Dualsystem

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

- Beispiel

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

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

Subtraktion mit b-Komplement

- bei Rechnung mit fester Stellenzahl n gilt:

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

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

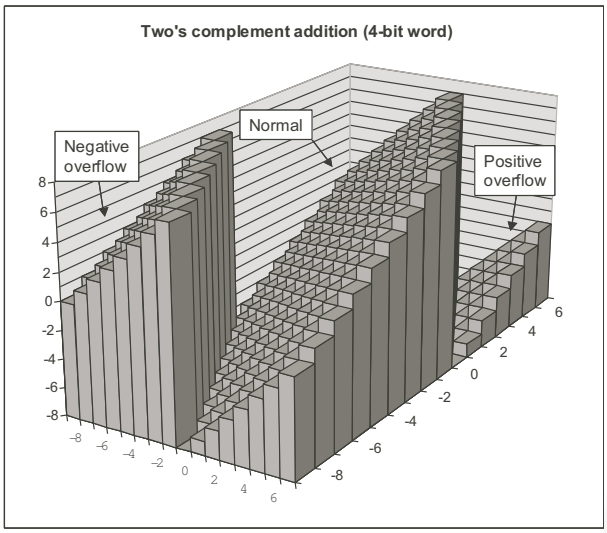
$$x - y = x + K_b(y)$$

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

- und für Integerzahlen gilt außerdem

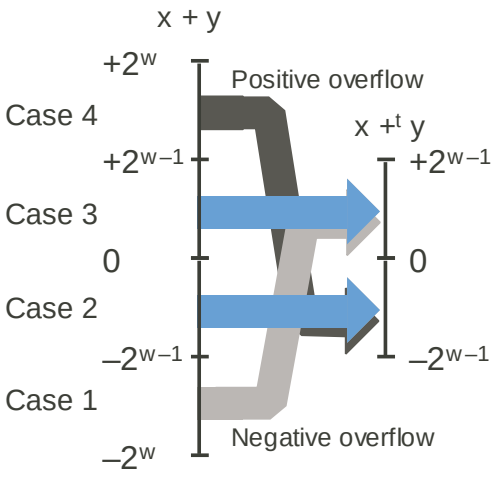
$$x - y = x + K_{b-1}(y) + 1$$

Visualisierung: 4-bit signed Addition (Zweierkomplement)



- Zahlenbereich der Operanden: -2^{w-1} .. $(2^{w-1} - 1)$
- Zahlenbereich des Resultats: -2^w .. $(2^w - 2)$
- Überlauf in beide Richtungen möglich

Überlauf: signed Addition



- Zahlenbereich der Operanden: -2^{w-1} .. $(2^{w-1}-1)$
- Zahlenbereich des Resultats: -2^{w} .. $(2^{w}-2)$
- Überlauf in beide Richtungen möglich

Überlauf: Erkennung

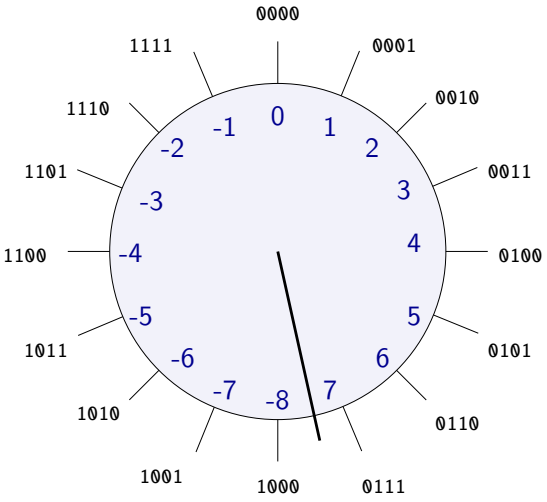
- Erkennung eines Überlaufs bei der Addition?
- wenn beide Operanden das gleiche Vorzeichen haben
- und Vorzeichen des Resultats sich unterscheidet

- Java-Codebeispiel

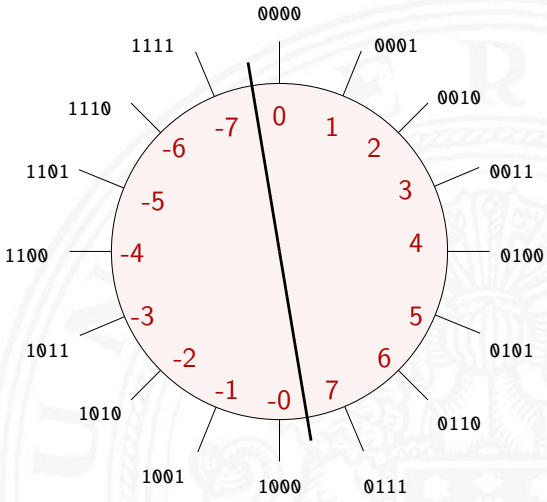
```
int a, b, sum;          // operands and sum
boolean ovf;            // ovf flag indicates overflow

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

Veranschaulichung: Zahlenkreis

Beispiel für w-bit

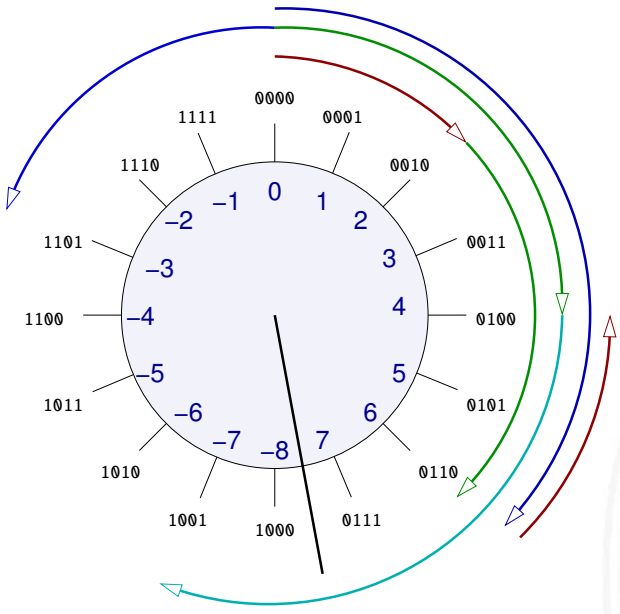


Zweierkomplement



Betrag und Vorzeichen

Zahlenkreis: Addition, Subtraktion



0010
0100
0101
0110

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

Unsigned-Zahlen in C

- 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];
}
```

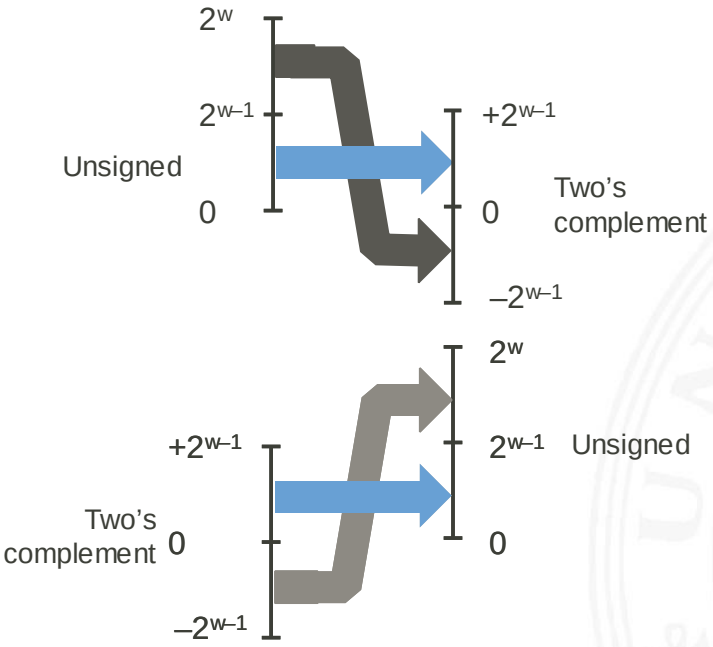

Unsigned-Typen in C: Casting-Regeln

- 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`

Interpretation: unsigned/signed



Typumwandlung in C: Vorsicht

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

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

Fehler

Sign-Extension

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

- **Sign-Extension**: Vorzeichenbit kopieren

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

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?

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

Ariane-5 Absturz

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. 370 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

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

Multiplikation im Dualsystem

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

- Multiplikationsmatrix ist sehr einfach:

$\times$	0	1
0	0	0
1	0	1

Multiplikation im Dualsystem (cont.)

- Beispiel

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

Multiplikation: Wertebereich unsigned

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

Multiplikation: Zweierkomplement

- Zahlenbereich der Operanden: $-2^{w-1} \,..\, (2^{w-1} - 1)$
- Zahlenbereich des Resultats: $-2^w \cdot (2^{w-1} - 1) \,..\, (2^{2w-2})$
- bis zu $2w$ bits erforderlich

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

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

- Wichtig: Bit-Repräsentation der unteren Bits des Resultats entspricht der unsigned Multiplikation
- kein separater Algorithmus erforderlich
- Beweis: siehe Bryant/O'Hallaron, 2.3.5

Java Puzzlers No. 3

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

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

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

Division: Dualsystem

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

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

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

Division: Beispiel im Dualsystem

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

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


Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
6.3 Arithmetik - Division
64-040 Rechnerstrukturen

Division: Beispiel im Dualsystem (cont.)

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

```
 1011011 / 1101 = 0111
 1011               0
 10110               1
-1101
----------------------
  10011               1
 -1101
----------------------
   01101               1
  -1101
----------------------
       0
```

Höhere mathematische Funktionen

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

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

Reziprokwert: Iterative Berechnung von $1/x$

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

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

- geeigneter Startwert y_0 als Schätzung erforderlich
- Beispiel $x = 3$, $y_0 = 0.5$:

$$\begin{aligned}
y_1 &= 0.5 \cdot (2 - 3 \cdot 0.5) && = 0.25 \\
y_2 &= 0.25 \cdot (2 - 3 \cdot 0.25) && = 0.3125 \\
y_3 &= 0.3125 \cdot (2 - 3 \cdot 0.3125) && = 0.33203125 \\
y_4 &= 0.3332824 \\
y_5 &= 0.3333333332557231 \\
y_6 &= 0.3333333333333333
\end{aligned}$$

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

Babylonisches Wurzelziehen

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

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

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

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

Informationstreue

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

Beispiele:

- Gilt $x^2 \geq 0$?
  - float: ja
  - signed integer: nein

- Gilt $(x + y) + z = x + (y + z)$?
  - integer: ja
  - float: nein

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

Eigenschaften: Rechnerarithmetik

- Addition ist kommutative Gruppe / Abel'sche Gruppe
  - Gruppe: Abgeschlossenheit, Assoziativgesetz, neutrales Element, Inverses
  - Kommutativgesetz
- Multiplikation
  - Abgeschlossenheit: $0 \leq \mathrm{UMult}_v(u, v) \leq 2^{w-1}$
  - Mult. ist kommutativ: $\mathrm{UMult}_v(u, v) = \mathrm{UMult}_v(v, u)$
  - Mult. ist assoziativ:
    $\mathrm{UMult}_v(t, \mathrm{UMult}_v(u, v)) = \mathrm{UMult}_v(\mathrm{UMult}_v(t, u), v)$
  - Eins ist neutrales Element: $\mathrm{UMult}_v(u, 1) = u$
  - Distributivgesetz:
    $\mathrm{UMult}_v(t,\ \mathrm{UAdd}_v(u, v)) = \mathrm{UAdd}_v(\mathrm{UMult}_v(t, u), \mathrm{UMult}_v(t, v))$

Eigenschaften: Rechnerarithmetik (cont.)

Isomorphe Algebren

- Unsigned Addition und Multiplikation (Wortbreite w Bits)
- Zweierkomplement-Addition und Multiplikation (w Bits)
- Isomorph zum Ring der ganzen Zahlen modulo 2^w

- Ring der ganzen Zahlen: Ordnungsrelationen
  - u $> 0 \longrightarrow u + v > v$
  - $u > 0, v > 0 \longrightarrow u \cdot v > 0$
  - diese Relationen gelten nicht bei Rechnerarithmetik (Überlauf)

Eigenschaften: Gleitkomma-Addition

verglichen mit Abel'scher Gruppe

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

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

Eigenschaften: Gleitkomma-Multiplikation

verglichen mit kommutativem Ring

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

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

Gliederung

Gliederung (cont.)

Gliederung (cont.)

Darstellung von Texten

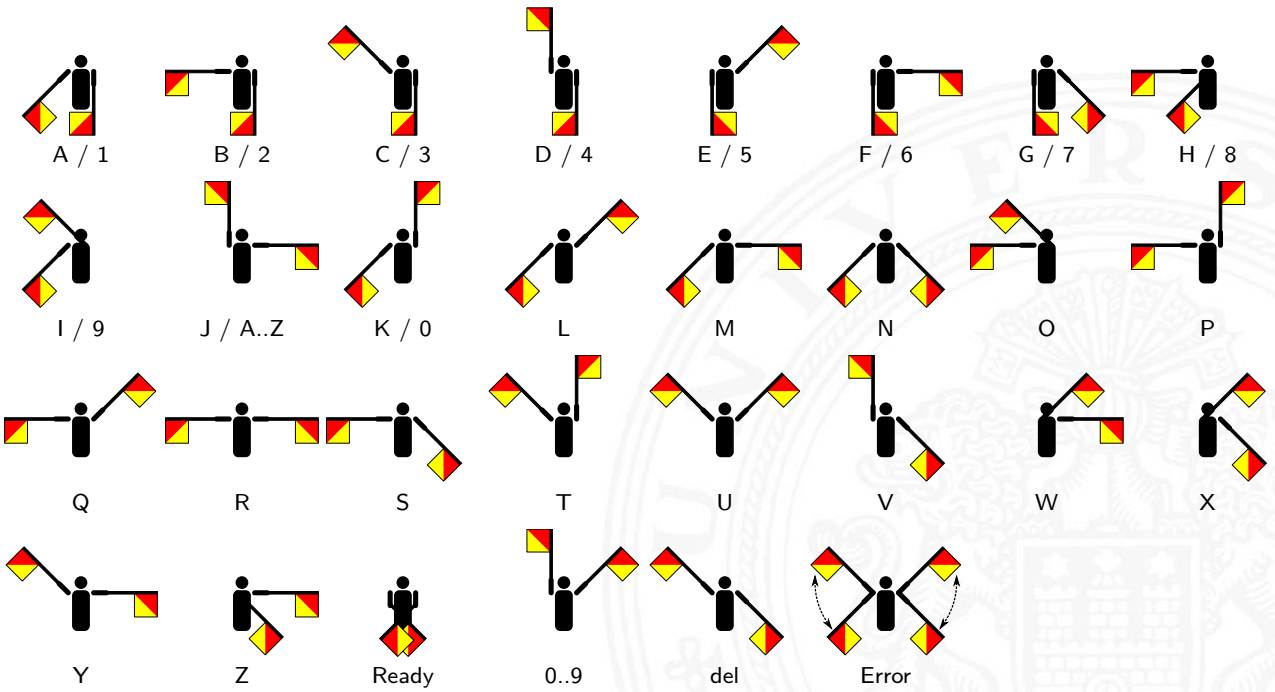
- Ad-Hoc Codierungen
  - Flaggen-Alphabet
  - Braille-Code
  - Morse-Code
- ASCII und ISO-8859-1
- Unicode

Wiederholung: Zeichenkette

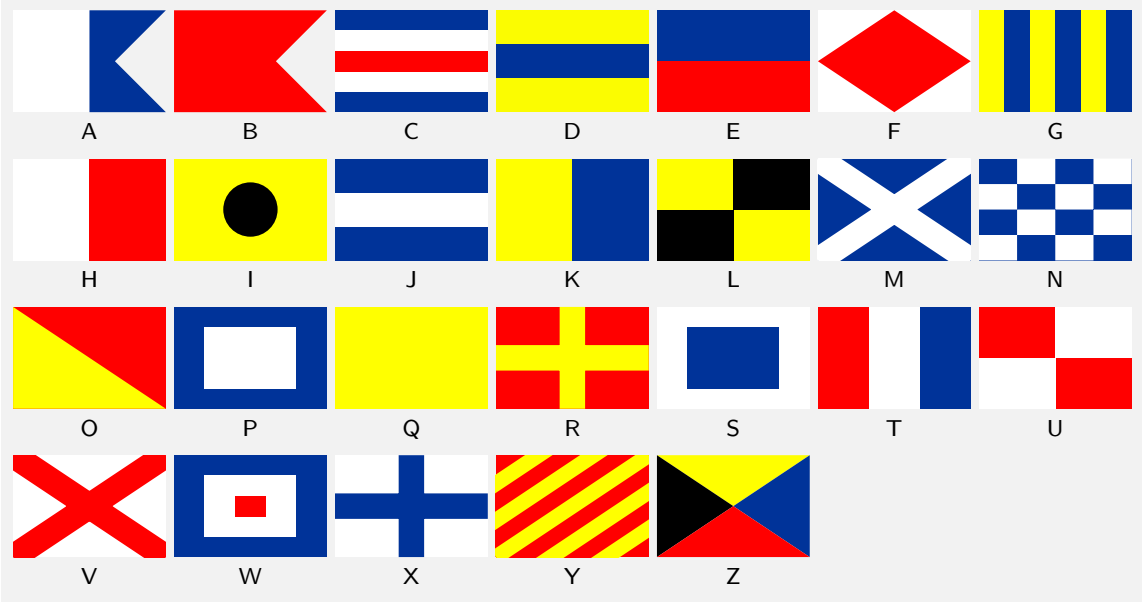
- **Zeichenkette** (engl. *string*): Eine Folge von Zeichen
- **Wort** (engl. *word*): Eine Folge von Zeichen, die in einem gegebenen Zusammenhang als Einheit bezeichnet wird. Worte mit 8 Bit werden als **Byte** bezeichnet.
- **Stelle** (engl. *position*): Die Lage/Position eines Zeichens innerhalb einer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

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

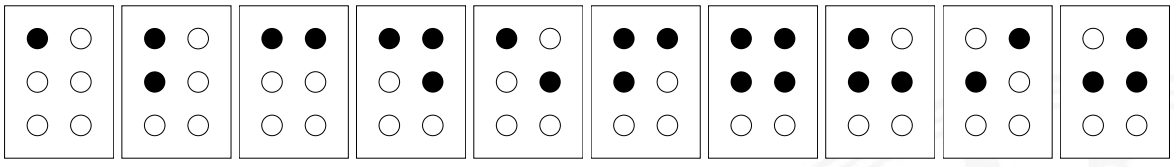
Flaggen-Signale



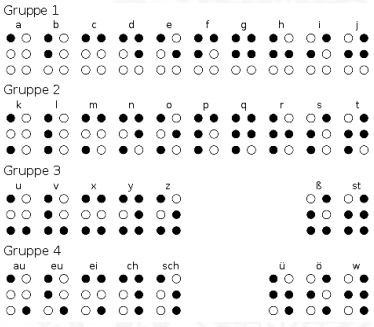
Flaggen-Alphabet



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

Punkt: kurzer Ton
Strich: langer Ton

Codetabelle des Morse-Codes

a	● –	o	– – –	4	● ● ●● –
ä	● – ● –	ö	– – – ●	5	● ● ● ● ●
å	● – – ● –	p	● – – ●	6	– ● ● ●●
b	– ● ● ●	q	– – ● –	7	– – ● ● ●
c	– ● – ●	r	● – ●	8	– – – ●●
ch	– – – –	s	● ● ●	9	– – – – ●
d	– ●●	t	–		
e	●	u	●● –	.	● – ● – ● –
é	●● – ●●	ü	●● – –	,	– – ●● – –
f	●● – ●	v	● ● ● –	:	– – – ● ● ●
g	– – ●	w	● – –	-	– ● ● ●● –
h	● ● ●●	x	– ●● –	'	● – – – – ●
i	●●	y	– ● – –	(	– ● – – ● –
j	● – – –	z	– – ●●	?	●● – – ●●
k	– ● –			“	● – ●● – ●
l	● – ●●	0	– – – – –	Notruf	● ● ● – – – ● ● ●
m	– –	1	● – – – –	SP	●●
n	– ●	2	●● – – –	Anfang	– ● – ● –
ñ	– – ● – –	3	● ● ● – –	Ende	● ● ● – ● –

Morse-Code (cont.)

- Eindeutigkeit

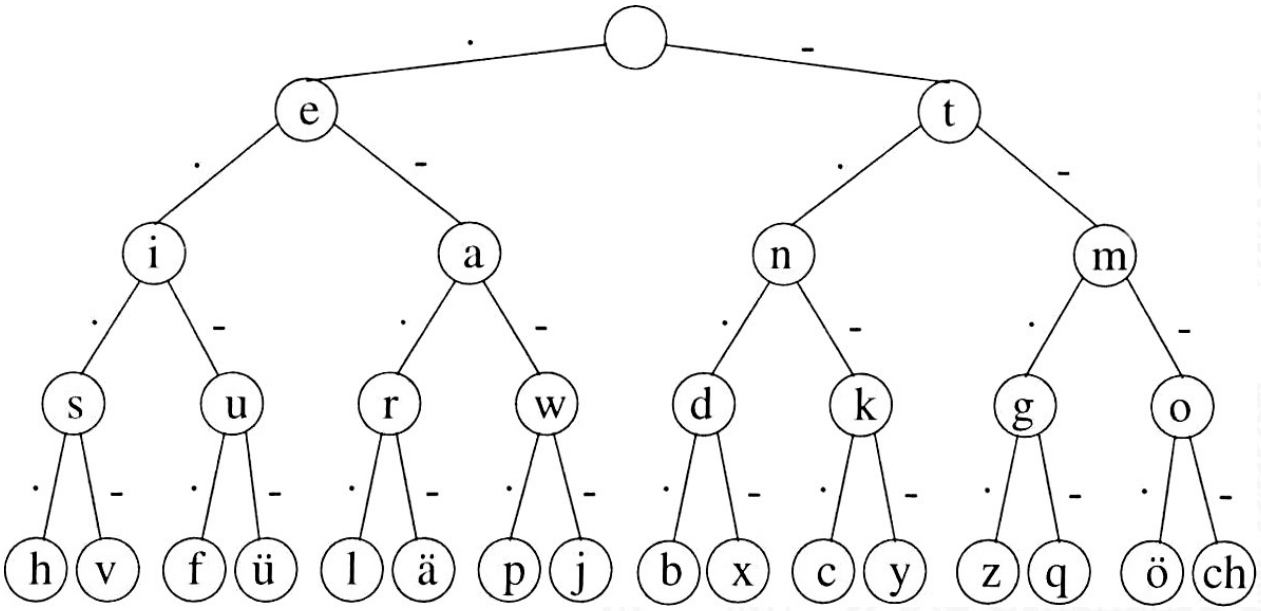
  Codewort: ● ● ● ● ● – ●

  e ⟶ ●
  i ⟶ ● ●
  n ⟶ – ●
  r ⟶ ● – ●
  s ⟶ ● ● ●

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

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

Morse-Code: Baumdarstellung (Ausschnitt)



- Anordnung der Symbole entsprechend ihrer Codierung

ASCII

American Standard Code for Information Interchange

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

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

ASCII: Codetabelle

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

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

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

ISO-8859 Familie

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

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

- immer noch nicht für mehrsprachige Dokumente geeignet

ISO-8859-1: Codetabelle (1)

Erweiterung von ASCII für westeuropäische Sprachen

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

ISO-8859-1: Codetabelle (2)

Sonderzeichen gemeinsam für alle 8859 Varianten

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

ISO-8859-2

Erweiterung von ASCII für slawische Sprachen

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

ISO-8859-15

Modifizierte ISO-8859-1 mit €

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8...	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9...	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
A...	NBSP	¡	¢	£	€	¥	Š	§	š	©	ª	«	¬	SHY	®	¯
B...	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
C...	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D...	Đ	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: Codepage 437, 850, 1252

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

- `http://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

Windows: Codepage 850

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

Austausch von Texten?

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

- Unterschiedliche Konvention für Zeilenende
  - DOS/Windows: CR/LF (`0D 0A`)
  - Unix/Linux: LF
  - Mac OS 9: CR

  - Konverter-Tools: dos2unix, unix2dos, iconv

Unicode: Motivation

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

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

⇒ **Unicode** als System zur Codierung aller Zeichen aller bekannten (lebenden oder toten) Schriftsysteme

Unicode: Versionen und History

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

- 1991 1.0.0: europäisch, nahöstlich, indisch
- 1992 1.0.1: ostasiatisch (Han)
- 1993 akzeptiert als ISO/IEC-10646 Standard
- ...
- 2010 6.0: knapp 110.000 Zeichen enthalten

`http://www.unicode.org`, `http://de.wikipedia.org/wiki/Unicode`

Unicode: Schreibweise

- ursprüngliche Version nutzt 16-bit pro Zeichen
- die sogenannte „*Basic Multilingual Plane*“
- Schreibweise hexadezimal als `U+xxxx`
- Bereich von `U+0000` .. `U+FFFF`
- Schreibweise in Java-Strings: `\uxxxx`
  z.B. `\u03A9` für Ω, `\u20A0` für das €-Symbol

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

Unicode: in Webseiten (HTML)

- HTML-Header informiert über verwendeten Zeichensatz
- Unterstützung und Darstellung abhängig vom Browser
- Demo `http://www.columbia.edu/~fdc/utf8`

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


Unicode: Demo

http://www.columbia.edu/~fdc/utf8

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

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

Unicode: Demo (cont.)

http://www.columbia.edu/~fdc/utf8

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

ვეპხის ტყაოსანი შოთა რუსთაველი

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

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

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

Unicode: Latin-Zeichen

- Zeichen im Bereich U+0000 bis U+007F wie ASCII
  www.unicode.org/charts/PDF/U0000.pdf
- Bereich von U+0100 bis U+017F für Latin-A
  Europäische Umlaute und Sonderzeichen
  www.unicode.org/charts/PDF/U0100.pdf
- viele weitere Sonderzeichen ab U+0180
  Latin-B, Latin-C, usw.

Unicode: Mathematische Symbole und Operatoren

Vielfältige Auswahl von Symbolen und Operatoren

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

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

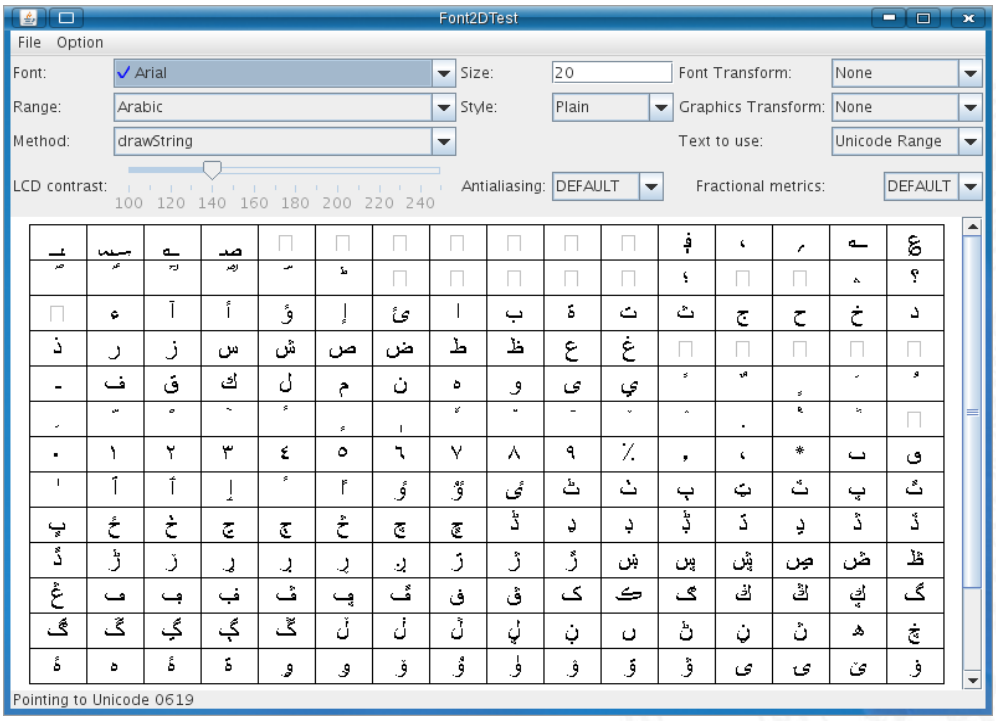
- Dingbats `www.unicode.org/charts/PDF/U2700.pdf`

Unicode: Asiatische Sprachen

Chinesisch (traditional/simplified), Japanisch, Koreanisch

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

Unicode: Java2D Fontviewer



Oracle
Java JDK examples
`../demo/jfc/Font2DTest`

Unicode: Repräsentation?

- 16-bit für jedes Zeichen, bis zu 65536 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: mehrere *Planes* zu je 65536 Zeichen
- direkte Repräsentation aller Zeichen erfordert 32-bit/Zeichen
- vierfacher Speicherbedarf gegenüber ISO-8859-1

- bei Dateien ist möglichst kleine Dateigröße wichtig
- effizientere Codierung üblich: UTF-16 und UTF-8

UTF-8

Zeichen	Unicode	Unicode binär	UTF-8 binär	UTF-8 hexadezimal
Buchstabe *y*	U+0079	00000000 01111001	01111001	0x79
Buchstabe *ä*	U+00E4	00000000 11100100	11000011 10100100	0xC3 0xA4
Zeichen für eingetragene Marke ®	U+00AE	00000000 10101110	11000010 10101110	0xC2 0xAE
Eurozeichen €	U+20AC	00100000 10101100	11100010 10000010 10101100	0xE2 0x82 0xAC
Violinschlüssel 𝄞	U+1D11E	00000001 11010001 00011110	11110000 10011101 10000100 10011110	0xF0 0x9D 0x84 0x9E

`http://de.wikipedia.org/wiki/UTF-8`

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

UTF-8: Algorithmus

Unicode-Bereich (hexadezimal)	UTF-Codierung (binär)	Anzahl (benutzt)
0000 0000 - 0000 007F	0*** ****	128
0000 0080 - 0000 07FF	110* **** 10** ****	1920
0000 0800 - 0000 FFFF	1110 **** 10** **** 10** ****	63488
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

Sprach-Einstellungen: Locale

Locale: die Sprach-Einstellungen und Parameter

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

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

`dos2unix`, `unix2dos`

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

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

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

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

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

iconv

Das Schweizer-Messer zur Umwandlung von Textcodierungen.

- Optionen
  - `-f, --from-code=<encoding>` Codierung der Eingabedatei
  - `-t, --to-code=<encoding>` Codierung der Ausgabedatei
  - `-l, --list` Liste der unterstützten Codierungen ausgeben
  - `-o, --output=<filename>` Name der Ausgabedatei

- Beispiel

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


base64-Codierung

Übertragung von (Binär-) Dateien zwischen verschiedenen Rechnern?

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

⇒ Verfahren zur Umcodierung der Datei in 7-bit ASCII notwendig
⇒ etabliert ist das **base-64** Verfahren (RFC 2045)

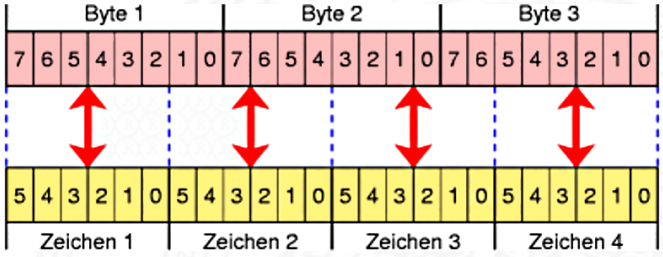
- alle e-mail Dateianhänge und 8-bit Textdateien
- Umcodierung benutzt nur Buchstaben, Ziffern und drei Sonderzeichen

base64-Codierung: Prinzip

- Codierung von drei 8-bit Bytes als vier 6-bit Zeichen
- erfordert 64 der verfügbaren 128 7-bit ASCII Symbole

- Codierung

`A..Z`	Codes:	0..25
`a..z`	Codes:	26..51
`0..9`	Codes:	52..61
+	Code:	62
/	Code:	63



=	Füllzeichen, falls Anzahl der Bytes nicht durch 3 teilbar
CR	Zeilenumbruch (optional), meistens nach 76 Zeichen

base64-Codierung: Prinzip (cont.)

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

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

base64-Codierung: Tools

- im Java JDK enthalten
  aber im inoffiziellen internen Teil
  `sun.misc.BASE64Encoder`, bzw. `sun.misc.BASE64Decoder`

- aber diverse (open-source) Implementierungen verfügbar
  Beispiel: Apache Commons http://commons.apache.org/codec
  `org.apache.commons.codec.binary.Base64`
  `org.apache.commons.codec.binary.Base64InputStream`
  `org.apache.commons.codec.binary.Base64OutputStream`

base64-Codierung: Beispiel

openbook.galileodesign.de/javainsel7/javainsel_04_008.htm

```
import java.io.IOException;
import java.util.*;
import sun.misc.*;

public class Base64Demo
{
  public static void main( String[] args ) throws IOException
  {
    byte[] bytes1 = new byte[ 112 ];
    new Random().nextBytes( bytes1 );

    // buf in String
    String s = new BASE64Encoder().encode( bytes1 );
    System.out.println( s );

    // Zum Beispiel:
    // QFgwDyiQ28/4GsF75fqLMj/bAIWNwOuBmE/SCl3H2XQFpSsSz0jtyR0LU+kLiwWsnSUZljJr97Hy
    // LA3YUbf96Ym2zx9F9Y1N7P5lsOCb/vr2crTQ/gXs757qaJF9E3szMN+E0CSSslDrrzcNBrlcQg==
    // String in byte[]
    byte[] bytes2 = new BASE64Decoder().decodeBuffer( s );
    System.out.println( Arrays.equals(bytes1, bytes2) );    // true
  }
}
```

Literatur: Vertiefung

- `http://www.unicode.org`
- The Java Tutorials, *Trail: Internationalization*
  `http://download.oracle.com/javase/tutorial/i18n`

Gliederung

Gliederung (cont.)

Wiederholung: Grundbegriffe der Algebra

- Mengen
- Relationen, Verknüpfungen
- Gruppe, Abel'sche Gruppe
- Körper, Ring
- Vektorraum
- usw.

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: z.B. fehlerkorrigierende Codes auf CD/DVD

Boole'sche Algebra

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

Grundverknüpfungen

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

NOT(x)

x	
0	1
1	0

AND(x, y)

y \ x	0	1
0	0	0
1	0	1

OR(x, y)

y \ x	0	1
0	0	1
1	1	1

XOR(x,y)

y \ x	0	1
0	0	1
1	1	0

- alle logischen Operationen lassen sich mit diesen Funktionen darstellen (*vollständige Basismenge*)

Grundverknüpfungen

- zwei Werte, $\{0, 1\}$
- 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
- allgemein 2^{2^n} Funktionen von n Variablen
- später noch viele Beispiele

Alle Funktionen von zwei Variablen

x = 0 1 0 1 y = 0 0 1 1	Bezeichnung	Notation	Alternativnotation	Java/C-Notation
0 0 0 0	Nullfunktion	0		`0`
0 0 0 1	AND	$x \cap y$		`x&&y`
0 0 1 0	Inhibition	$y > x$		`y>x`
0 0 1 1	Identität y	y		`y`
0 1 0 0	Inhibition	$x > y$		`x>y`
0 1 0 1	Identität x	x		`x`
0 1 1 0	XOR	$x \oplus y$	$x \neq y$	`x!=y`
0 1 1 1	OR	$x \cup y$		`x||y`
1 0 0 0	NOR	$\neg(x \cup y)$		`!(x||y)`
1 0 0 1	Äquivalenz	$\neg(x \oplus y)$	$x = y$	`x==y`
1 0 1 0	NICHT x	$\neg x$	x'	`!x`
1 0 1 1	Implikation	$x \leq y$	$x \rightarrow y$	`y>=x`
1 1 0 0	NICHT y	$\neg y$	y'	`!y`
1 1 0 1	Implikation	$x \geq y$	$x \leftarrow y$	`x>=y`
1 1 1 0	NAND	$\neg(x \cap y)$		`!(x&&y)`
1 1 1 1	Einsfunktion	1		`1`

Boole'sche Algebra

- 6-Tupel $< \{0,1\}, \vee, \wedge, \neg, \ 0,1 >$ 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

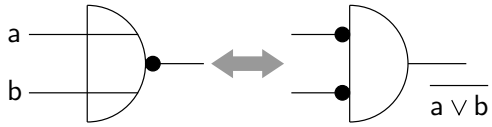
Eigenschaft	Ring der ganzen Zahlen	Boole'sche Algebra
Kommutativ-gesetz	$a + b = b + a$ $a \times b = b \times a$	$a \vee b = b \vee a$ $a \wedge b = b \wedge a$
Assoziativ-gesetz	$(a + b) + c = a + (b + c)$ $(a \times b) \times c = a \times (b \times c)$	$(a \vee b) \vee c = a \vee (b \vee c)$ $(a \wedge b) \wedge c = a \wedge (b \wedge c)$
Distributiv-gesetz	$a \times (b + c) = (a \times b) + (a \times c)$	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
Identitäten	$a + 0 = a$ $a \times 1 = a$	$a \vee 0 = a$ $a \wedge 1 = a$
Vernichtung	$a \times 0 = 0$	$a \wedge 0 = 0$
Auslöschung	$-(-a) = a$	$\neg(\neg a) = a$
Inverses	$a + (-a) = 0$	—

Rechenregeln: Ring / Algebra (cont.)

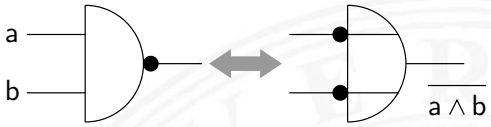
Eigenschaft	Ring der ganzen Zahlen	Boole'sche Algebra
Distributivgesetz	—	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
Komplement	— —	$a \vee \neg a = 1$ $a \wedge \neg a = 0$
Idempotenz	— —	$a \vee a = a$ $a \wedge a = a$
Absorption	— —	$a \vee (a \wedge b) = a$ $a \wedge (a \vee b) = a$
De-Morgan Regeln	— —	$\neg(a \vee b) = \neg a \wedge \neg b$ $\neg(a \wedge b) = \neg a \vee \neg b$

De-Morgan Regeln

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



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



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

Verwendung

- bei der Minimierung logischer Ausdrücke
- beim Entwurf von Schaltungen
- siehe Abschnitte: „Schaltfunktionen“ und „Schaltnetze“

XOR: Exklusiv-Oder / Antivalenz

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

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

Gliederung

Gliederung (cont.)

Gliederung (cont.)

Logische Operationen in Java und C

- eigener Datentyp?
  - Java: Datentyp `boolean`
  - C: implizit für alle Integertypen
- Vergleichsoperationen
- logische Grundoperationen
- Auswertungsreihenfolge / -prioritäten
- logische Operationen auch bitweise parallel möglich (s.u.)

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
- Auswertung von links nach rechts,
  mit (optionaler) Klammerung

Logische Operationen in C

- zusätzlich zu den Vergleichsoperatoren <, <=, ==, !=, >, >=
- drei **logische** Operatoren:
  ! logische Negation
  && logisches UND
  || logisches ODER

- Interpretation der Integerwerte:
  der Zahlenwert *0* ⇔ logische 0 (false)
  alle anderen Werte ⇔ logische 1 (true)
- völlig andere Semantik als in der Mathematik (!)
- völlig andere Funktion als die bitweisen Operationen (s.u.)

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))`
Beispiel	Wert
`!0x41`	`0x00`
`!0x00`	`0x01`
`!!0x00`	`0x00`
`0x69 && 0x55`	`0x01`
`0x69 || 0x55`	`0x01`

Logische Operationen in C: Logisch vs. Bitweise

- der Zahlenwert *0* ⇔ logische 0 (false)
  alle anderen Werte ⇔ logische 1 (true)

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

Ausdruck (bitweise)	Wert	Ausdruck (logisch)	Wert
`x`	0110 0110	`x`	0000 0001
`y`	1001 0011	`y`	0000 0001
`x & y`	0000 0010	`x && y`	0000 0001
`x | y`	1111 0111	`x || y`	0000 0001
`~x | ~y`	1111 1101	`!x || !y`	0000 0000

Logische Operationen in C: verkürzte Auswertung

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

Ternärer Operator

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

Logische Operationen in Java

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

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

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

Bitweise logische Operationen

Integer-Datentypen doppelt genutzt:

1. Zahlenwerte (Ganzzahl, Zweierkomplement, Gleitkomma)
   arithmetische Operationen: Addition, Subtraktion, usw.

2. Binärwerte mit w einzelnen Bits (Wortbreite w)
   Boole'sche Verknüpfungen, bitweise auf allen w Bits
   - Grundoperationen: Negation, UND, ODER, XOR
   - Schiebe-Operationen: shift-left, rotate-right, usw.

Bitweise logische Operationen (cont.)

- Integer-Datentypen interpretiert als Menge von Bits
- bitweise logische Operationen möglich
- es gibt insgesamt 2^{2^n} Operationen mit n Operanden

- in Java und C sind vier Operationen definiert:

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

- alle anderen Funktionen können damit dargestellt werden

Bitweise logische Operationen: Beispiel

```
    x = 0010 1110
    y = 1011 0011

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

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


Schiebeoperationen

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

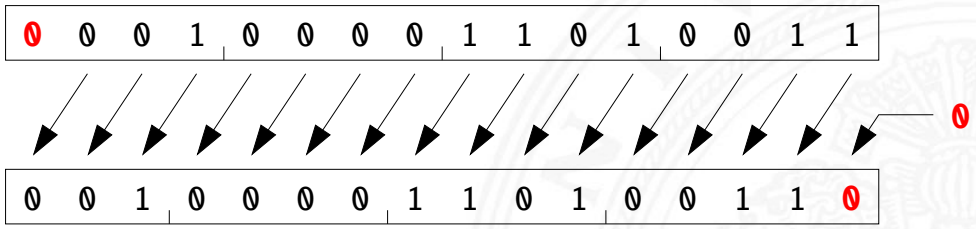
- fünf Varianten:

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

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

Shift-Left (shl)

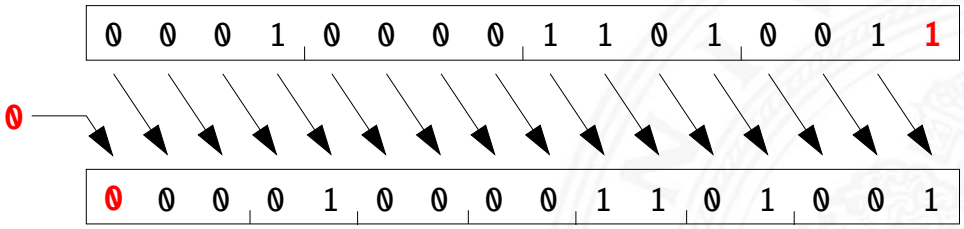
- Verschieben der Binärdarstellung von x um n bits nach links
- links herausgeschobene n bits gehen verloren
- von rechts werden n Nullen eingefügt



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

Logical Shift-Right (srl)

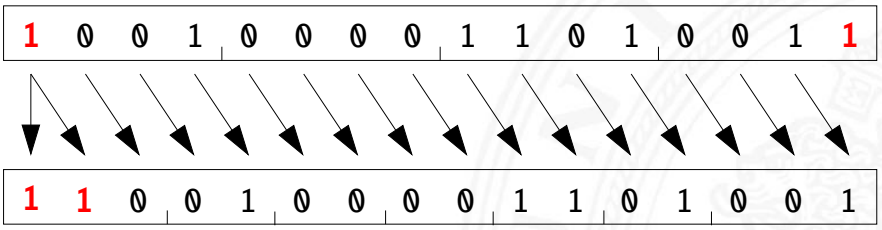
- Verschieben der Binärdarstellung von x um n bits nach rechts
- rechts herausgeschobene n bits gehen verloren
- von links werden n Nullen eingefügt



- in Java direkt als Operator verfügbar: `x >>> n`
  in C nur für `unsigned`-Typen definiert: `x >> n`
  für `signed`-Typen nicht vorhanden

Arithmetic Shift-Right (sra)

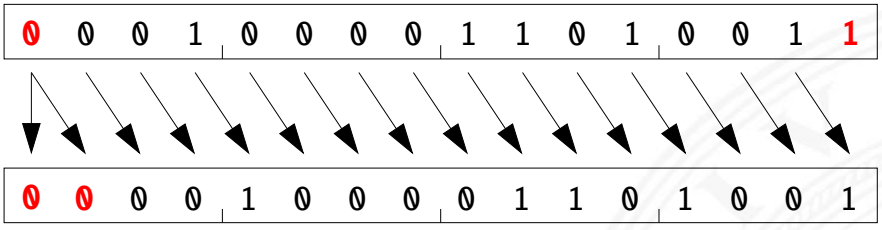
- Verschieben der Binärdarstellung von x um n bits nach rechts
- rechts herausgeschobene n bits gehen verloren
- von links wird n-mal das MSB (Vorzeichenbit) eingefügt
- Vorzeichen bleibt dabei erhalten (gemäß Zweierkomplement)



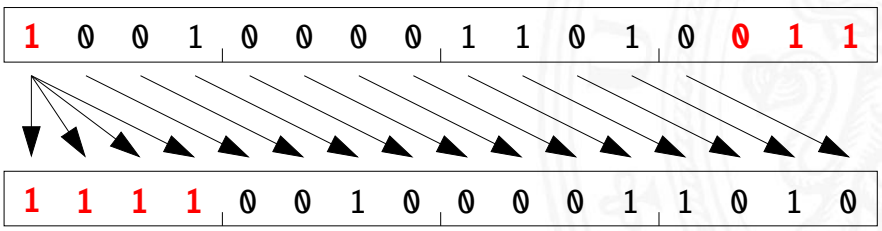
- in Java direkt als Operator verfügbar: `x >> n`
  in C nur für `signed`-Typen definiert: `x >> n`
- `sra` um n bits ist ähnlich der Division durch 2^n

Arithmetic Shift-Right: Beispiel

- `x >> 1` aus `0x10D3` (4307) wird `0x0869` (2153)



- `x >> 3` aus `0x90D3` (-28460) wird `0xF21A` (-3558)



Arithmetic Shift-Right: Division durch Zweierpotenzen?

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

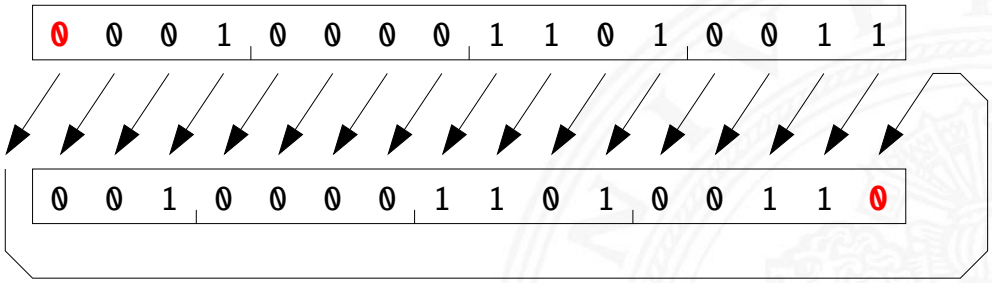
- gerundet in Richtung negativer Werte statt in Richtung Null:

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

- C: Kompensation durch Berechnung von `(x + (1<<k)-1) >> k`
  Details: Bryant & O'Hallaron

Rotate-Left (`rol`)

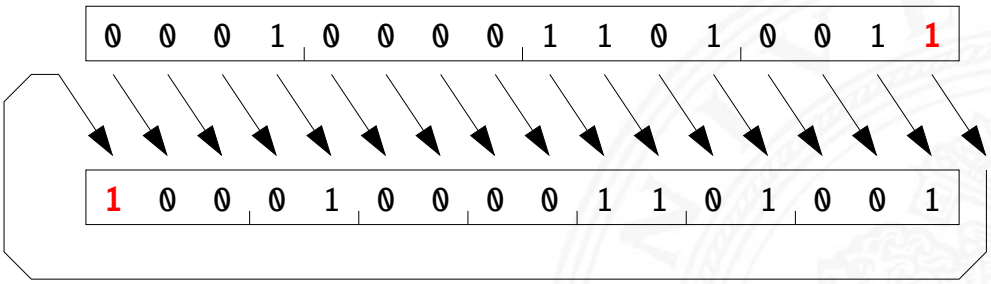
- Rotation der Binärdarstellung von x um n bits nach links
- herausgeschobene Bits werden von rechts wieder eingefügt



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

Rotate Right (ror)

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



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

Shifts statt Integer-Multiplikation

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

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

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

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

Beispiel: bit-set, bit-clear

Bits an Position p in einem Integer setzen oder löschen?

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

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

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


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

Linux: `/usr/include/bits/byteswap.h`

```
...
/* 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..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 (mit while-Schleife)

Anzahl der gesetzten Bits in einem Wort?

- Anwendung z.B. für Kryptalgorithmen (Hamming-Distanz)
- 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;
}
```

- viele Algorithmen: bit-Maskierung und Schieben Übungsaufgabe 4.4
  - `http://gurmeet.net/puzzles/fast-bit-counting-routines`
  - `http://graphics.stanford.edu/~seander/bithacks.html`
  - Donald E. Knuth, *The Art of Computer Programming*: Volume 4A, Combinational Algorithms: Part1, Abschnitt 7.1.3
  - `java.lang.Integer.bitCount()`
- viele neuere Prozessoren/DSPs: eigener bitcount-Befehl

Tipps & Tricks: Rightmost bits

Donald E. Knuth, *The Art of Computer Programming*, Vol 4.1

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

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

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

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

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

Tipps & Tricks: Rightmost bits (cont.)

Donald E. Knuth, *The Art of Computer Programming*, Vol 4.1

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

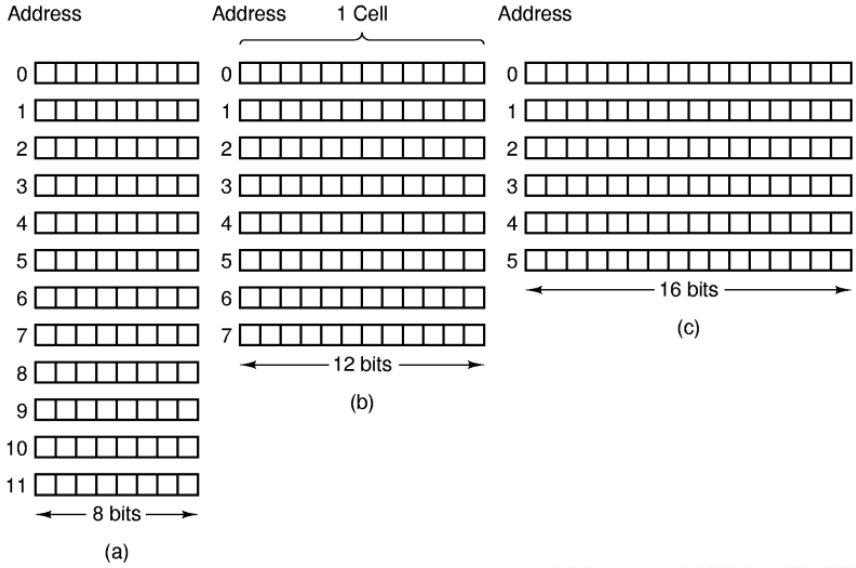
$x\&(x-1) = (\alpha\ \ 0[1]^a 0[0]^b)_2$	letzte 1 entfernt
$x\&-x = (0^\infty 0[0]^a 1[0]^b)_2$	letzte 1 extrahiert
$x \mid -x = (1^\infty 1[1]^a 1[0]^b)_2$	letzte 1 nach links verschmiert
$x \oplus -x = (1^\infty 1[1]^a 0[0]^b)_2$	letzte 1 entfernt und verschmiert
$x \mid (x-1) = (\alpha\ \ 0[1]^a 1[1]^b)_2$	letzte 1 nach rechts verschmiert
$\overline{x}\&(x-1) = (0^\infty 0[0]^a 0[1]^b)_2$	letzte 1 nach rechts verschmiert
$((x \mid (x-1)) + 1)\&x = (\alpha\ \ 0[0]^a 0[0]^b)_2$	letzte 1-Bit Folge entfernt

Aufbau und Adressierung des Speichers

- Abspeichern von Zahlen, Zeichen, Strings?
  - kleinster Datentyp üblicherweise ein Byte (8-bit)
  - andere Daten als Vielfache: 16-bit, 32-bit, 64-bit, ...
- Organisation und Adressierung des Speichers?
  - Adressen typisch in Bytes angegeben
  - erlaubt Adressierung einzelner ASCII-Zeichen, usw.
- aber Maschine/Prozessor arbeitet wortweise
- Speicher daher ebenfalls wortweise aufgebaut
- typischerweise 32-bit oder 64-bit

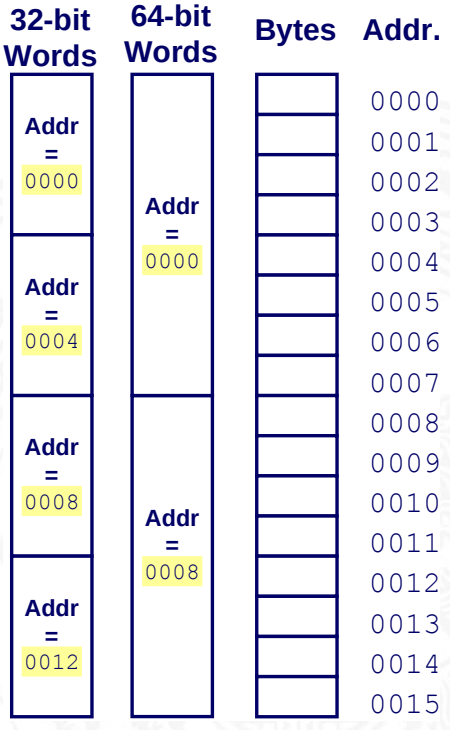
Speicher-Organisation

- Speicherkapazität: Anzahl der Worte · Bits/Wort
- Beispiele: $12 \cdot 8$ $8 \cdot 12$ $6 \cdot 16$ Bits



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



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

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

`www.agner.org/optimize/calling_conventions.pdf`

Byte-Order

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

Zwei Möglichkeiten / Konventionen:

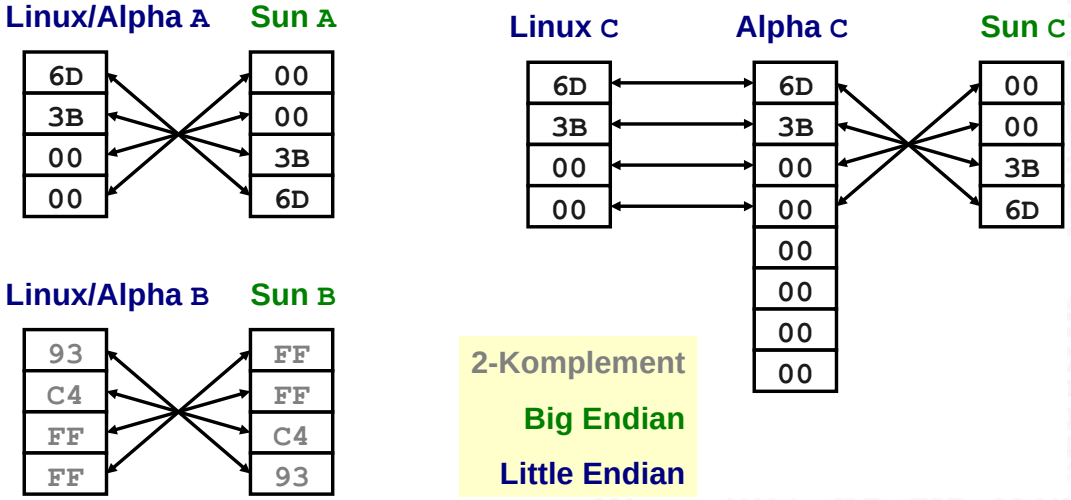
- **Big Endian**: Sun, Mac, usw.
  das MSB (*most significant byte*) hat die kleinste Adresse
  das LSB (*least significant byte*) hat die höchste –"–
- **Little Endian**: Alpha, x86
  das MSB hat die höchste, das LSB die kleinste Adresse

satirische Referenz auf Gulliver's Reisen (Jonathan Swift)

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



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: x86 und SPARC

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

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

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

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

Netzwerk-Byteorder

- Byteorder 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
  - ...

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

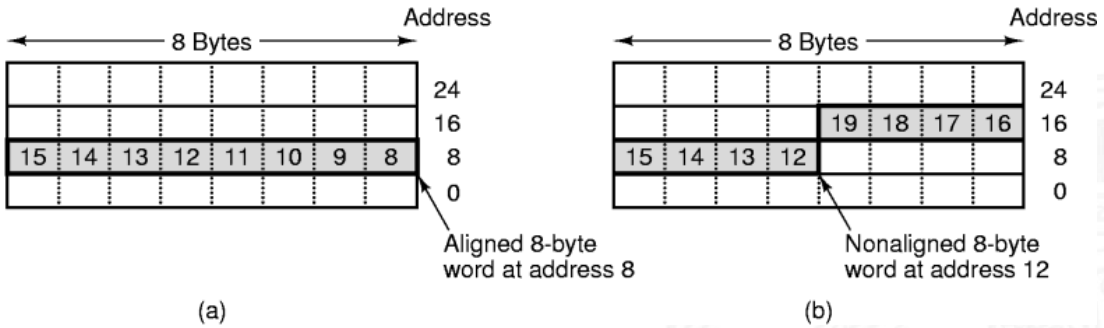
Linux: `/usr/include/bits/byteswap.h`

```
...
/* 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
...
```

Misaligned Memory Access



- Speicher Byte-weise adressiert
- aber Zugriffe lesen/schreiben jeweils ein ganzes Wort

Was passiert bei „krummen“ (*misaligned*) Adressen?

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

Programm zum Erkennen der Byteorder

- Programm gibt Daten byteweise aus
- C-spezifische Typ- (Pointer-) Konvertierung
- Details: siehe Bryant 2.1.4 (und Abbildungen 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 ));
}
  ...
```

Literatur: Vertiefung

- Donald E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques, Binary Decision Diagrams*, Addison-Wesley, 2009

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

Gliederung

Gliederung (cont.)

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
10 Codierung
64-040 Rechnerstrukturen

Gliederung (cont.)

Definition: Codierung

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

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

Code, Codewörter

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

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

Gründe für den Einsatz von Codes

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

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

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

Wichtige Aspekte

Unterteilung gemäß der Aufgabenstellung

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

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

Darstellung von Codes

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

Codierung von Text

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

Für geschriebenen (formatierten) Text:

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

Codierungen für Dezimalziffern

	BCD	Gray	Exzess3	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

Codierungen für Dezimalziffern

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

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

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

Begriffe für Binärcodes

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

Dualcode

- der Name für Codierung der Integerzahlen im Stellenwertsystem
- Codewort

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

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

Einschrittige Codes

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

- Einschrittige Codes werden benutzt, wenn ein Ablesen der Bits auch beim Wechsel zwischen zwei Codeworten möglich ist (bzw. nicht verhindert werden kann)
- z.B.: Winkelcodierscheiben oder digitale Schieblehre
- viele interessante Varianten möglich: siehe Knuth, AoCP

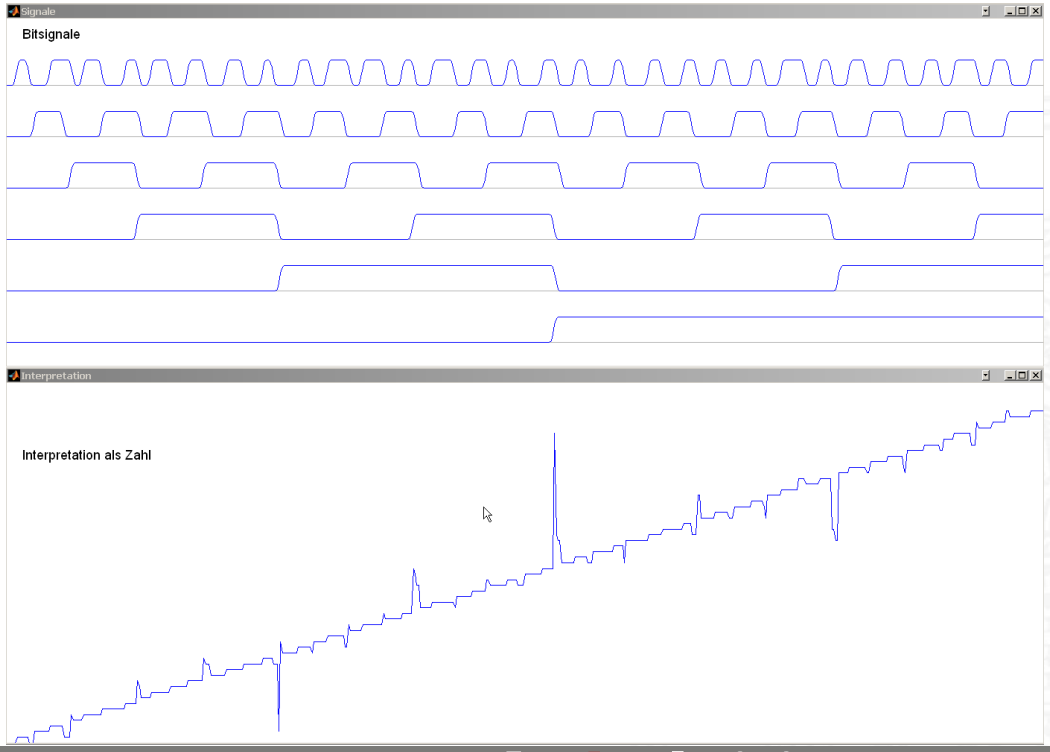
Einschrittige Codes: Matlab-Demo

- T1-Skript, Kapitel 1.4: Ablesen eines Wertes mit leicht gegeneinander verschobenen Übergängen der Bits
  - `demoeinschritt(0:59)` normaler Dualcode
  - `demoeinschritt(einschritt(60))` einschrittiger Code

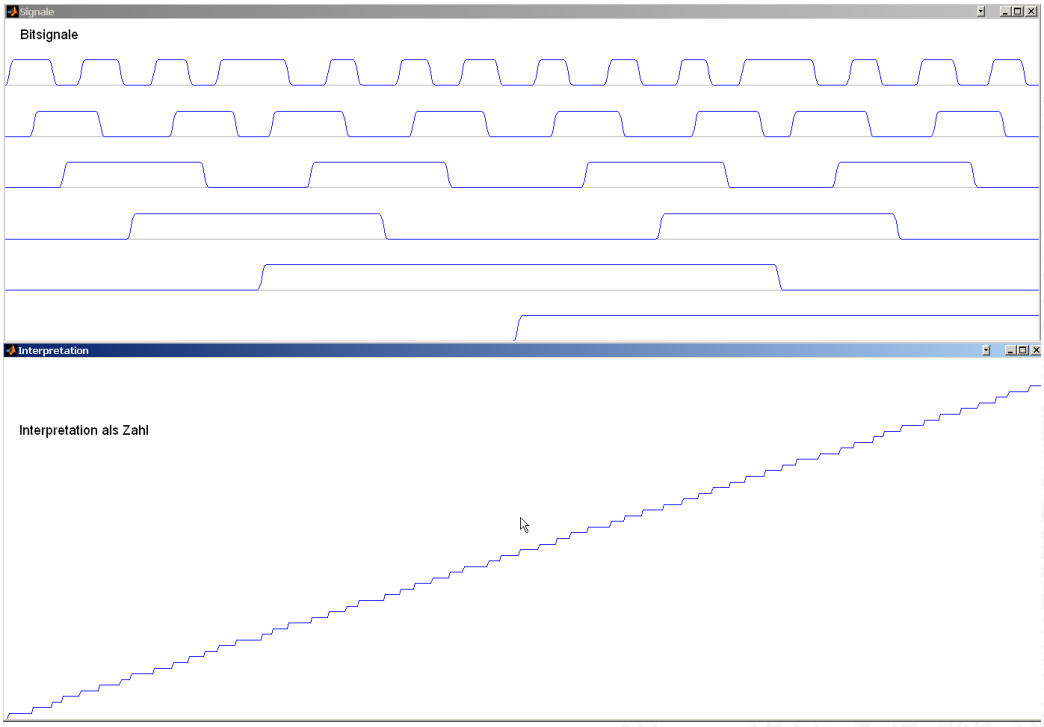
- maximaler Ablesefehler
  - 2^{n-1} beim Dualcode
  - 1 beim einschrittigen Code

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

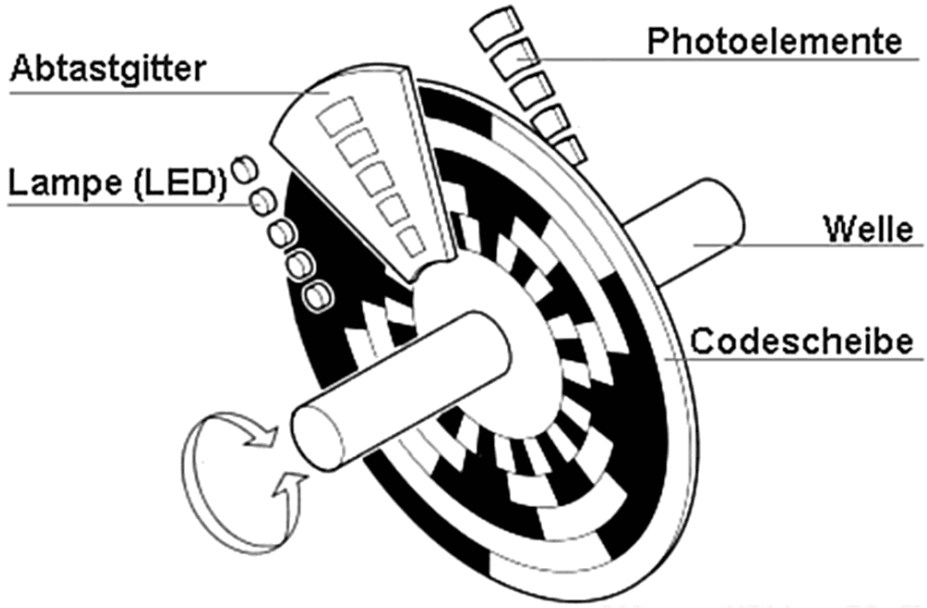
Ablesen des Wertes aus Dualcode



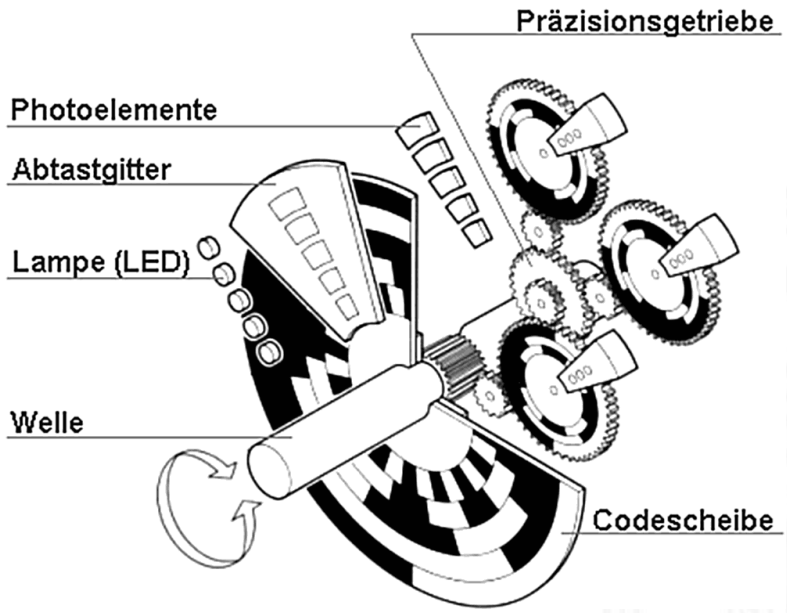
Ablesen des Wertes aus einschrittigem Code



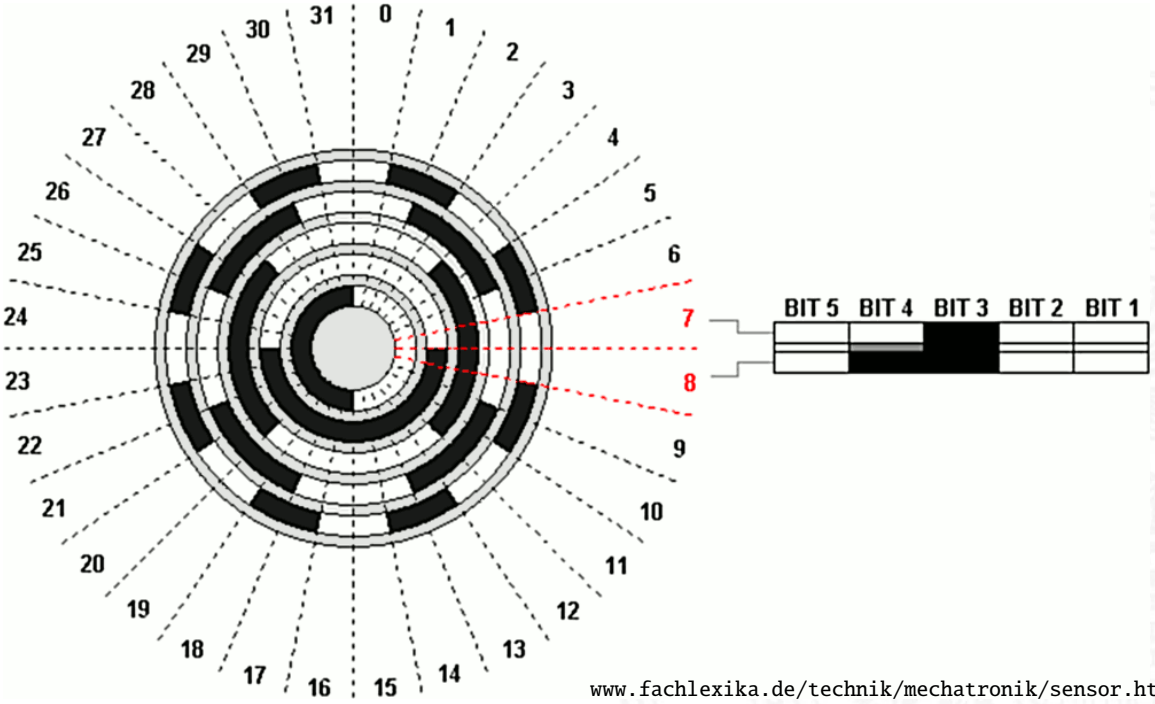
Gray-Code: Prinzip eines Winkeldrehgebers



Gray-Code: mehrstufiger Drehgeber

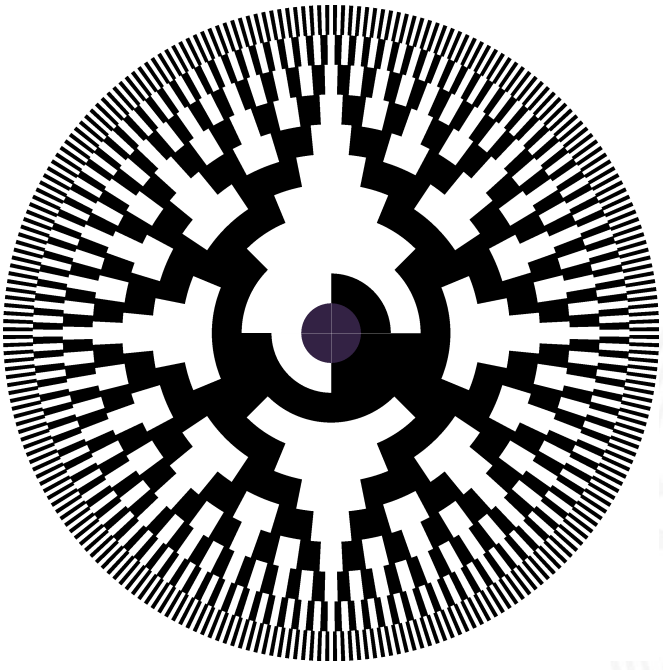


Gray-Code: 5-bit Codierscheibe



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

Gray-Code: 10-bit Codierscheibe



java GreycodeWheel 600 5 5 10

Einschrittiger Code: rekursive Konstruktion

- Starte mit zwei Codewörtern: 0 und 1

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

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

Karnaugh-Veitch Diagramm

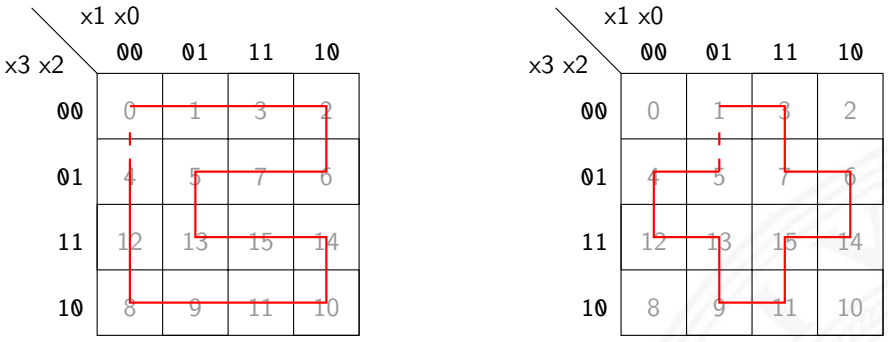
x3 x2 \ x1 x0	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

x3 x2 \ x1 x0	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

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

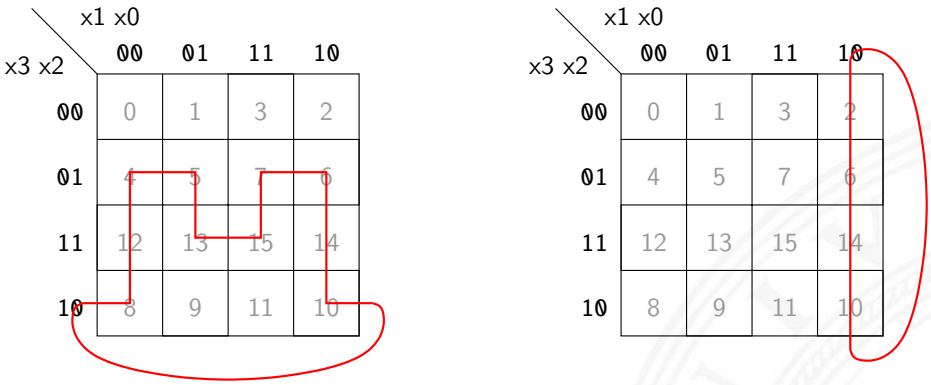
$\Rightarrow$ benachbarte Felder unterscheiden sich gerade um 1 Bit

Einschrittiger Code: KV-Diagramm



- jeder Pfad entspricht einem einschrittigen Code
- geschlossener Pfad: zyklisch einschrittiger Code
  - links: 0,1,3,2,6,7,5,13,15,14,10,11,9,8,12,4
  - rechts: 1,3,7,6,14,15,11,9,13,12,4,5

Einschrittiger Code: KV-Diagramm (cont.)



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

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

- Pfade können auch „außen herum“ geführt werden
  - links: 4,5,13,15,7,6,14,10,8,12 rechts: 2,6,14,10

Gray-Code: Umwandlung in/von Dualcode

Umwandlung: Dual- in Graywort

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

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

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

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

Umwandlung: Gray- in Dualwort

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

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

- in Hardware einfach durch Kette von XOR-Operationen

  `http://tams.informatik.uni-hamburg.de/applets/hades/webdemos/10-gates/15-graycode/dual2gray.html`

Optimalcodes: Codes variabler Länge

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

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

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

oder Einführen von **Markern** zwischen den Codewörtern

Fano-Bedingung

Eindeutige Decodierung eines Codes mit variabler Wortlänge?

Fano-Bedingung

Kein Wort aus einem Code bildet den Anfang eines anderen Codewortes

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

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

Fano-Bedingung: Beispiele

- Telefonnummern: das Vorwahlsystem gewährleistet die Fano-Bedingung

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

- Morse-Code: Fano-Bedingung verletzt

Morse-Code

Punkt: kurzer Ton
Strich: langer Ton

Codetabelle des Morse-Codes

a	● –	o	– – –	4	● ● ●● –
ä	● – ● –	ö	– – – ●	5	● ● ● ● ●
å	● – – ● –	p	● – – ●	6	– ● ● ●●
b	– ● ● ●	q	– – ● –	7	– – ● ● ●
c	– ● – ●	r	● – ●	8	– – – ●●
ch	– – – –	s	● ● ●	9	– – – – ●
d	– ●●	t	–		
e	●	u	●● –	.	● – ● – ● –
é	●● – ●●	ü	●● – –	,	– – ●● – –
f	●● – ●	v	● ● ● –	:	– – – ● ● ●
g	– – ●	w	● – –	-	– ● ● ●● –
h	● ● ●●	x	– ●● –	'	● – – – – ●
i	●●	y	– ● – –	(	– ● – – ● –
j	● – – –	z	– – ●●	?	●● – – ●●
k	– ● –			“	● – ●● – ●
l	● – ●●	0	– – – – –	Notruf	● ● ● – – – ● ● ●
m	– –	1	● – – – –	SP	●●
n	– ●	2	●● – – –	Anfang	– ● – ● –
ñ	– – ● – –	3	● ● ● – –	Ende	● ● ● – ● –

Morse-Code (cont.)

- Eindeutigkeit

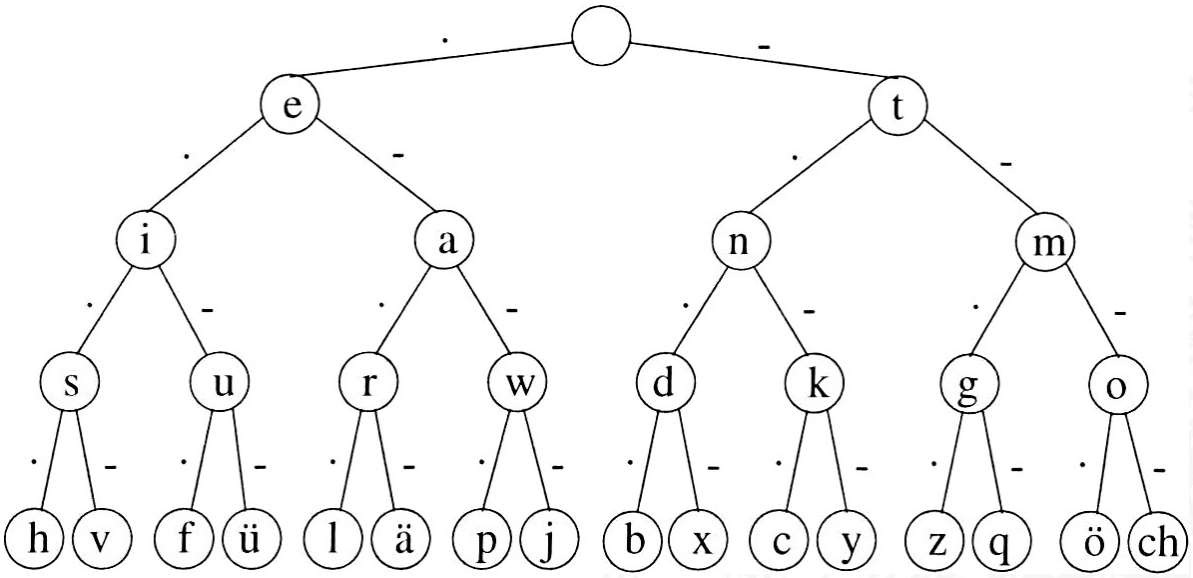
  Codewort: ●●●●●−●

  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
- Codewort am Pfad von Wurzel zum Blatt ablesen

Morse-Code: Umschlüsselung

Umschlüsselung des Codes für binäre Nachrichtenübertragung

- 110 als Umschlüsselung des langen Tons –
  10 als Umschlüsselung des kurzen Tons .
  0 als Trennzeichen zwischen Morse-Codewörtern
- der neue Code erfüllt die Fano-Bedingung
  jetzt eindeutig decodierbar: 10101001101101100101010 0 (SOS)
- viele andere Umschlüsselungen möglich, z.B.:
  1 als Umschlüsselung des langen Tons –
  01 als Umschlüsselung des kurzen Tons .
  00 als Trennzeichen zwischen Morse-Codewörtern

Codierung nach Fano

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

- Ordnung der Urwörter anhand ihrer Wahrscheinlichkeiten
  $p(a_1) \geq p(a_2) \geq \cdots \geq p(a_n)$
- Einteilung der geordneten Urwörter in zwei Gruppen mit möglichst gleicher Gesamtwahrscheinlichkeit. Eine Gruppe bekommt als erste Codewortstelle eine 0, die andere eine 1
- Diese Teilgruppen werden wiederum entsprechend 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
- vorteilhafter, je größer die Anzahl der Urwörter (!)

Codierung nach Fano: Beispiel

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

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

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

mittlere Codewortlänge L

- $L = 0.45 \cdot 1 + 0.3 \cdot 2 + 0.15 \cdot 3 + 0.1 \cdot 3 = 1.8$
- zum Vergleich: Blockcode mit 2 Bits benötigt $L = 2$

Codierung nach Fano: Deutsche Großbuchstaben

Buchstabe a_i	Wahrscheinlichkeit $p(a_i)$	Code (Fano)	Bits
Leerzeichen	0.15149	000	3
E	0.14700	001	3
N	0.08835	010	3
R	0.06858	0110	4
I	0.06377	0111	4
S	0.05388	1000	4
...	...	...	...
Ö	0.00255	111111110	9
J	0.00165	1111111110	10
Y	0.00017	11111111110	11
Q	0.00015	111111111110	12
X	0.00013	111111111111	12

Fano-Code der Buchstaben der deutschen Sprache, Ameling 1992

Codierung nach Huffman

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

- Ordnung der Urwörter anhand ihrer Wahrscheinlichkeiten $p(a_1) \leq p(a_2) \leq \cdots \leq p(a_n)$
- in jedem Schritt werden die zwei Wörter mit der geringsten Wahrscheinlichkeit zusammengefasst und durch ein neues ersetzt
- das Verfahren wird wiederholt, bis eine Menge mit nur noch zwei Wörtern resultiert
- rekursive Codierung als Baum (z.B.: links 0, rechts 1)
- ergibt die kleinstmöglichen mittleren Codewortlängen
- Abweichungen zum Verfahren nach Fano sind aber gering
- vielfältiger Einsatz (u.a. bei JPEG, MPEG, ...)

Codierung nach Huffman: Beispiel

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

0. Sortierung nach Wahrscheinlichkeiten ergibt $\{B, C, D, A\}$
1. Zusammenfassen von B und C als neues Wort E, Wahrscheinlichkeit von E ist dann $p(E) = 0.1 + 01.5 = 0.25$
2. Zusammenfassen von D und E als neues Wort F mit $p(F) = 0.55$
3. Zuordnung der Bits entsprechend der Wahrscheinlichkeiten
   - $F = 0$ und $A = 1$
   - Split von F in $D = 00$ und $E = 01$
   - Split von E in $C = 010$ und $B = 011$

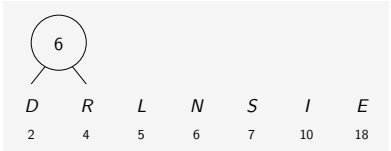
$\Rightarrow$ Codewörter sind $A = 1$, $D = 00$, $C = 010$ und $B = 011$

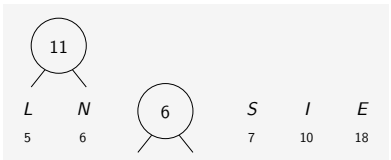
Bildung eines Huffman-Baums

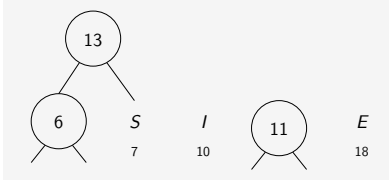
- Alphabet $= \{E, I, N, S, D, L, R\}$
- relative Häufigkeiten
  $E = 18, I = 10, N = 6, S = 7, D = 2, L = 5, R = 4$

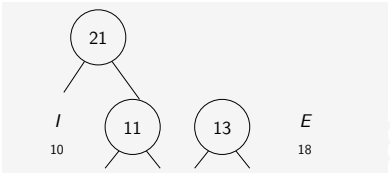
- Sortieren anhand der Häufigkeiten
- Gruppierung (rekursiv)
- Aufbau des Codebaums
- Ablesen der Codebits

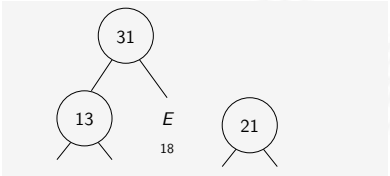
Bildung eines Huffman-Baums (cont.)

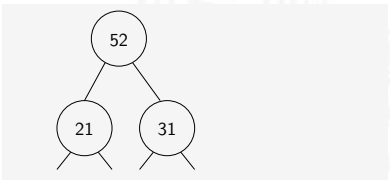




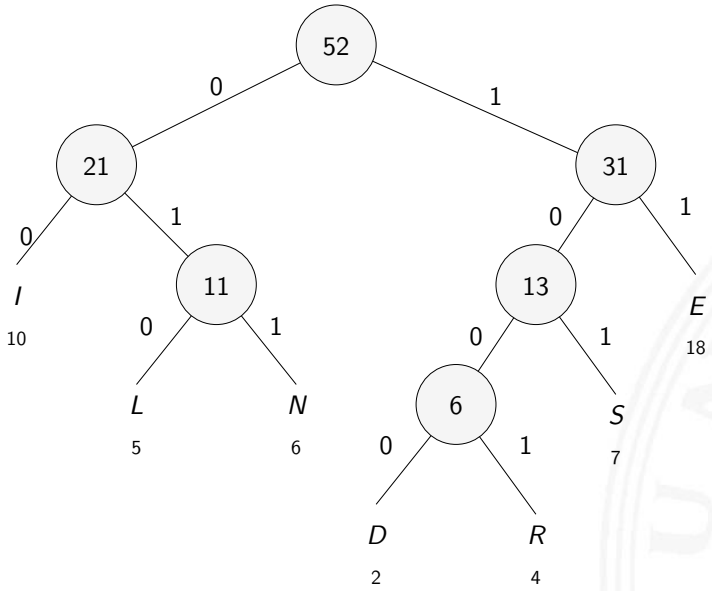








Bildung eines Huffman-Baums (cont.)



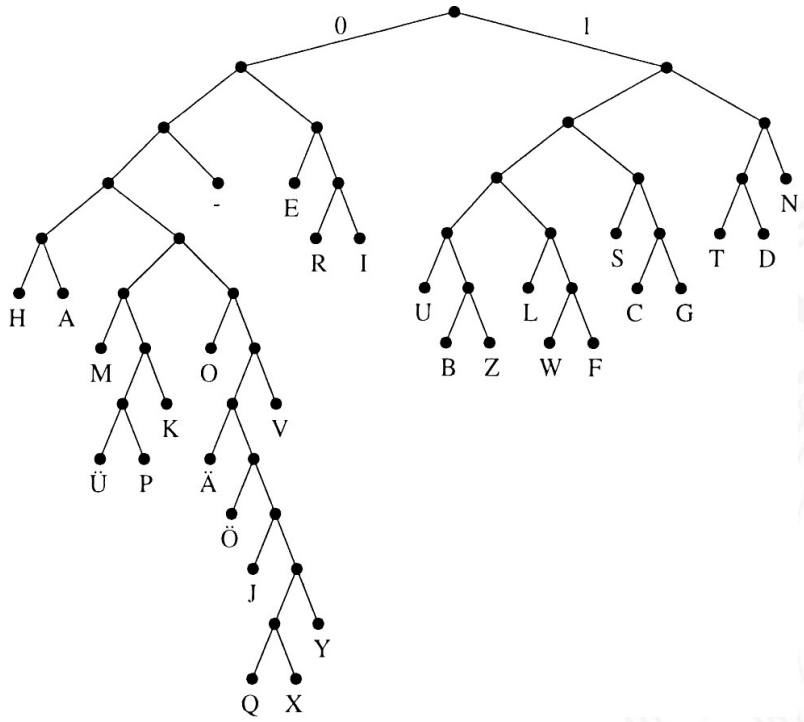
I	00
L	010
N	011
D	1000
R	1001
S	101
E	11

1001 00 11 101 11
R I E S E

Codierung nach Huffman: Deutsche Großbuchstaben

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

Codierung nach Huffman: Codebaum



ca. 4.5 Bits/Zeichen,
1.7-Mal besser als ASCII

Codierung nach Huffman: Minimale Codelänge

- Sei C ein Huffman-Code mit durchschnittlicher Codelänge L
- Sei D ein weiterer Präfix-Code mit durchschnittlicher Codelänge M, mit $M < L$ und M minimal
- Berechne die C und D zugeordneten Decodierbäume A und B
- Betrachte die beiden Endknoten für Symbole kleinster Wahrscheinlichkeit:
  - Weise dem Vorgängerknoten das Gewicht $p_{s-1} + p_s$ zu
  - streiche die Endknoten
  - Codelänge reduziert sich um $p_{s-1} + p_s$
- Fortsetzung führt dazu, dass Baum C sich auf Baum mit durchschnittlicher Länge 1 reduziert, und D auf Länge < 1. Dies ist aber nicht möglich.

Codierung nach Huffman: Symbole mit $p \geq 0.5$

Was passiert, wenn ein Symbol eine Häufigkeit $p_0 \geq 0.5$ aufweist?

- die Huffman-Codierung müsste weniger als ein Bit zuordnen, dies ist jedoch nicht möglich

$\Rightarrow$ Huffman- (und Fano-) Codierung ist in diesem Fall ineffizient

- Beispiel: Codierung eines Bildes mit einheitlicher Hintergrundfarbe
- andere Ideen notwendig
  - Lauflängencodierung (Fax, GIF, PNG)
  - Cosinustransformation (JPEG), usw.

Dynamic Huffman Coding

was tun, wenn

- die Symbolhäufigkeiten nicht vorab bekannt sind?
- die Symbolhäufigkeiten sich ändern können?

Dynamic Huffman Coding (Knuth 1985)

- Encoder protokolliert die (bisherigen) Symbolhäufigkeiten
- Codebaum wird dynamisch aufgebaut und ggf. umgebaut

- Decoder arbeitet entsprechend:
  Codebaum wird mit jedem decodierten Zeichen angepasst
- Symbolhäufigkeiten werden nicht explizit übertragen

D.E. Knuth, Dynamic Huffman Coding, J.of Algorithms 6, 163–180, 1985

Kraft-Ungleichung

- Leon G. Kraft, 1949
  `http://de.wikipedia.org/wiki/Kraft-Ungleichung`

- Eine notwendige und hinreichende Bedingung für die Existenz eines eindeutig decodierbaren s-elementigen Codes C mit Codelängen $l_1 \leq l_2 \leq l_3 \leq \cdots \leq l_s$ über einem q-nären Zeichenvorrat F ist:

$$\sum_{i=1}^{s} \frac{1}{q^{l_i}} \leq 1$$

- Beispiel
  $\{1, 00, 01, 11\}$ ist nicht eindeutig decodierbar,
  denn $\frac{1}{2} + 3 \cdot \frac{1}{4} = 1.25 > 1$

Kraft-Ungleichung: Beispiel

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

- Einsetzen in die Ungleichung: $\frac{1}{3} + 5 \cdot \frac{1}{3^2} + 3 \cdot \frac{1}{3^3} = 1$

$\Rightarrow$ Also existiert ein passender Präfixcode.

- Konstruktion entsprechend des Beweises

  0 10 11 12 20 21 220 221 222

Kraft-Ungleichung: Beweis

Sei $l_s = m$ und seien u_i die Zahl der Codewörter der Länge i

- Wir schreiben

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

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

- Jedes Codewort der Länge i „verbraucht“ q^{m-i} Wörter aus F^m
- Summe auf der linken Seite von $(*)$ ist die Zahl der durch den Code C benutzten Wörter von F^m

$\Rightarrow$ Wenn C die Präfix-Bedingung erfüllt, gilt $(*)$

Informationstheorie

- Informationsbegriff
- Maß für die Information?
- Entropie
- Kanalkapazität

Informationsbegriff

- n mögliche sich gegenseitig ausschließende Ereignisse A_i
- die zufällig nacheinander mit Wahrscheinlichkeiten p_i eintreten
- stochastisches Modell $W\{A_i\} = p_i$

- angewendet auf Informationsübertragung:
  das Symbol a_i wird mit Wahrscheinlichkeit p_i empfangen

- Beispiel
  - $p_i = 1$ und $p_j = 0 \quad \forall j \neq i$
  - dann wird mit Sicherheit das Symbol A_i empfangen
  - der Empfang bringt keinen Informationsgewinn

$\Rightarrow$ Informationsgewinn („Überraschung“) wird größer, je kleiner p_i

Geeignetes Maß für die Information?

- Wir erhalten die Nachricht A mit der Wahrscheinlichkeit p_A und anschließend die unabhängige Nachricht B mit der Wahrscheinlichkeit p_B

- Wegen der Unabhängigkeit ist die Wahrscheinlichkeit beider Ereignisse gegeben durch das Produkt $p_A \cdot p_B$.

- Informationsgewinn („Überraschung“) größer, je kleiner p_i
- Wahl von $1/p$ als Maß für den Informationsgewinn?
- möglich, aber der Gesamtinformationsgehalt zweier (mehrerer) Ereignisse wäre das Produkt der einzelnen Informationsgehalte

- additive Größe wäre besser $\Rightarrow$ Logarithmus von $1/p$ bilden

Erinnerung: Logarithmus

- Umkehrfunktion zur Exponentialfunktion
- formal: für gegebenes a und b ist der Logarithmus die Lösung der Gleichung $a = b^x$
- falls die Lösung existiert, gilt: $x = \log_b(a)$

- Beispiel $3 = \log_2(8)$, denn $2^3 = 8$

- Rechenregeln
  - $log(x \cdot y) = log(x) + log(y)$
  - $b^{\log_b(x)} = x$ und $\log_b(b^x) = x$
  - $\log_b(x) = \frac{\log_a(x)}{log_a(b)}$
  - $\log_2(x) = \log(x)/\log(2) = \log(x)/0,693141718$

Erinnerung: Binärer Logarithmus

- $\log_2(x) = 0.b_1b_2b_3\ldots = \sum_{k>0} b_k 2^{-k}$ mit $b_k \in \{0,1\}$

  $\log_2(x^2) = b_1.b_2b_3\ldots$ wegen $\log(x^2) = 2\log(x)$

- Berechnung

  Input: $1 < x < 2$ (ggf. vorher skalieren)
  Output: Nachkommastellen b_i der Binärdarstellung von $ld(x)$

```
i = 0
LOOP
  i = i+1
  x = x*x
  IF (x >= 2)
  THEN   x  = x/2
         bi = 1
  ELSE   bi = 0
  END IF
END LOOP
```

Definition: Informationsgehalt

Informationsgehalt eines Ereignisses A_i mit Wahrscheinlichkeit p_i?

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

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

- **Informationsgehalt** I (oder Information) von A_i
- auch **Entscheidungsgehalt** genannt

- Beispiel: zwei Nachrichten A und B

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

Informationsgehalt: Einheit Bit

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

- Wert von I ist eine reelle Größe
- gemessen in der Einheit **1 Bit**

- Beispiel: nur zwei mögliche Symbole 0 und 1 mit gleichen Wahrscheinlichkeiten $p_0 = p_1 = \frac{1}{2}$

  Der Informationsgehalt des Empfangs einer 0 oder 1 ist dann $I(0) = I(1) = \log_2(1/\frac{1}{2}) = 1\,\text{Bit}$

- Achtung: die Einheit Bit nicht verwechseln mit Binärstellen oder den Symbolen 0 und 1

Ungewissheit, Überraschung, Information

Vor dem Empfang einer Nachricht gibt es Ungewissheit über das Kommende

Beim Empfang gibt es die Überraschung

Und danach hat man den Gewinn an Information

- Alle drei Begriffe in der oben definierten Einheit **Bit** messen
- Diese Quantifizierung der Information ist zugeschnitten auf die Nachrichtentechnik

- umfasst nur einen Aspekt des umgangssprachlichen Begriffs Information

Informationsgehalt: Beispiele

Meteorit

- die Wahrscheinlichkeit, an einem Tag von einem Meteor getroffen zu werden, sei $p_M = 10^{-16}$

- Kein Grund zur Sorge, weil die Ungewissheit von $I = \log_2(1/(1-p_M)) \approx 3,2 \cdot 10^{-16}$ sehr klein ist

  Ebenso klein ist die Überraschung, wenn das Unglück nicht passiert $\Rightarrow$ Informationsgehalt der Nachricht „Ich wurde nicht vom Meteor erschlagen“ ist sehr klein

- Umgekehrt wäre die Überraschung groß: $\log_2(1/p_M) = 53,15$

Informationsgehalt: Beispiele (cont.)

Würfeln

- bei vielen Spielen hat die 6 eine besondere Bedeutung
- hier betrachten wir aber zunächst nur die Wahrscheinlichkeit von Ereignissen, nicht deren Semantik

- die Wahrscheinlichkeit, eine 6 zu würfeln, ist $1/6$
- $I(6) = \log_2(\frac{1}{6}) = 2,585$

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
10.6 Codierung - Informationstheorie
64-040 Rechnerstrukturen

Informationsgehalt: Beispiele (cont.)

Information eines Buchs

- Gegeben seien zwei Bücher
  1. deutscher Text
  2. mit Zufallsgenerator mit Gleichverteilung aus Alphabet mit 80-Zeichen erzeugt
- Informationsgehalt in beiden Fällen?
  1. Im deutschen Text abhängig vom Kontext!
     Beispiel: Empfangen wir als deutschen Text „*Der Begrif*“, so ist „*f*“ als nächstes Symbol sehr wahrscheinlich
  2. beim Zufallstext liefert jedes neue Symbol die zusätzliche Information $I = \log_2(1/(1/80))$

$\Rightarrow$ der Zufallstext enthält die größtmögliche Information

Informationsgehalt: Beispiele (cont.)

Einzelner Buchstabe

- die Wahrscheinlichkeit, in einem Text an einer gegebenen Stelle das Zeichen „A“ anzutreffen sei $W\{A\} = p = 0,01$

- Informationsgehalt $I(A) = \log_2(1/0,01) = 6,6439$
- wenn der Text in ISO-8859-1 codiert vorliegt, werden 8 Binärstellen zur Repräsentation des „A“ benutzt
- der Informationsgehalt ist jedoch geringer

Bit : als Maß für den Informationsgehalt
bit : Anzahl der Binärstellen 0 und 1

Entropie

Obige Definition der Information lässt sich nur jeweils auf den Empfang eines speziellen Zeichens anwenden

- Was ist die **durchschnittliche Information** bei Empfang eines Symbols?
- diesen Erwartungswert bezeichnet man als **Entropie** des Systems

- Wahrscheinlichkeiten aller möglichen Ereignisse A_i seien $W\{A_i\} = p_i$
- da jeweils eines der möglichen Symbole eintrifft, gilt $\sum_i p_i = 1$

Entropie (cont.)

- dann berechnet sich die Entropie H als Erwartungswert

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

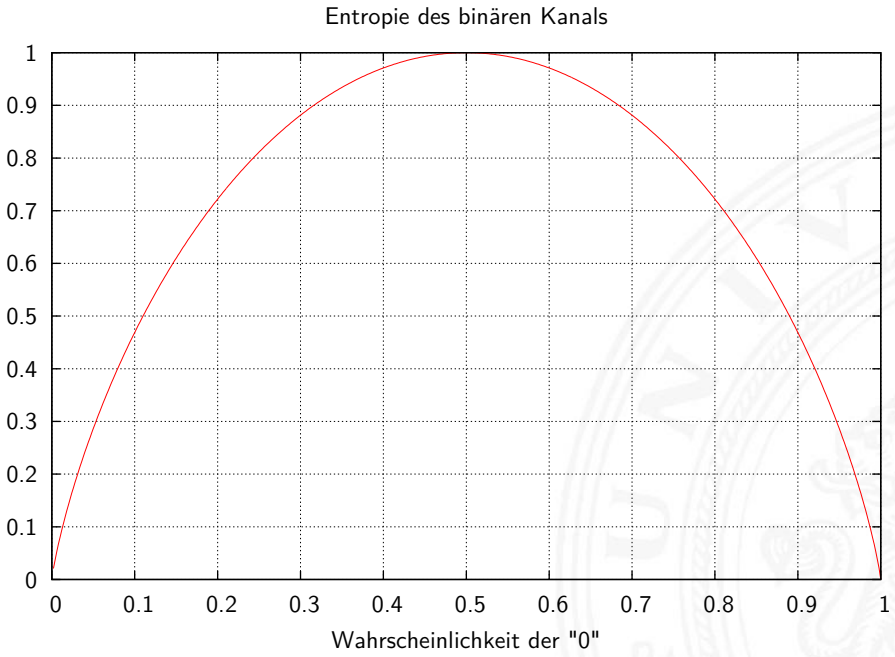
- als Funktion der Symbol-Wahrscheinlichkeiten nur abhängig vom stochastischen Modell

Entropie: Beispiele

1. drei mögliche Ereignisse mit Wahrscheinlichkeiten $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\}$
   - dann berechnet sich die Entropie zu $H = -(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{3}\log_2\frac{1}{3} + \frac{1}{6}\log_2\frac{1}{6}) = 1,4591$

2. Empfang einer Binärstelle mit den Wahrscheinlichkeiten $p_0 = q$ und $p_1 = (1 - q)$.
   - für $q = \frac{1}{2}$ erhält man $H = -(\frac{1}{2}\log_2\frac{1}{2} + (1 - \frac{1}{2})\log_2(1 - \frac{1}{2})) = 1.0$
   - mittlerer Informationsgehalt beim Empfang einer Binärstelle mit gleicher Wahrscheinlichkeit für beide Symbole ist genau 1 Bit

Entropie: Diagramm



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

Entropie: Gleichverteilte Symbole

- 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 Bytes), gleich wahrscheinlich
  $H = \sum_i p_i \log_2(1/p_i) = 256 \cdot (1/256) \cdot \log_2(1/(1/256)) = 8\,\text{Bit}$
- **Redundanz**: die Differenz zwischen dem aufgrund der Symbole möglichen (z.B. Wortlängen) und dem tatsächlich genutzten Informationsinhalt

Entropie: einige Eigenschaften

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

Redundanz

- **Redundanz** (engl. *code redundancy*): die Differenz zwischen dem aufgrund der Symbole möglichen (z.B. Wortlängen) und dem tatsächlich genutzten Informationsinhalt $R = H_0 - H$
- relative Redundanz: $r = \frac{H_0 - H}{H_0}$
- binäre Blockcodes mit Wortlänge N bits: $H_0 = N$
  gegebener Code mit m Wörtern a_i und $p(a_i)$:

$$R = H_0 - H = H_0 - \left(-\sum_{i=1}^{m} p(a_i) \log_2 p(a_i) \right)$$

$$= N + \sum_{i=1}^{m} p(a_i) \log_2 p(a_i)$$

Kanalkapazität

Informationstheorie ursprünglich entwickelt zur:

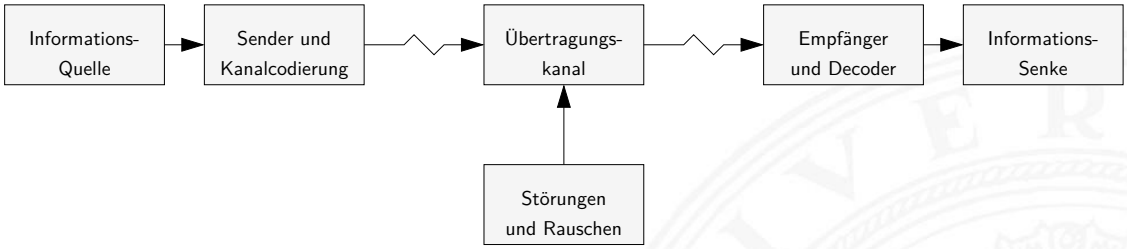
- formalen Behandlung der Übertragung von Information
- über reale nicht fehlerfreie Kanäle
- deren Verhalten als stochastisches Modell formuliert werden kann

- zentrales Resultat ist die **Kanalkapazität** C des **binären symmetrischen Kanals**
- der maximal pro Binärstelle übertragbare Informationsgehalt

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

mit $H(F)$ der Entropie des Fehlerverhaltens

Erinnerung: Modell der Informationsübertragung

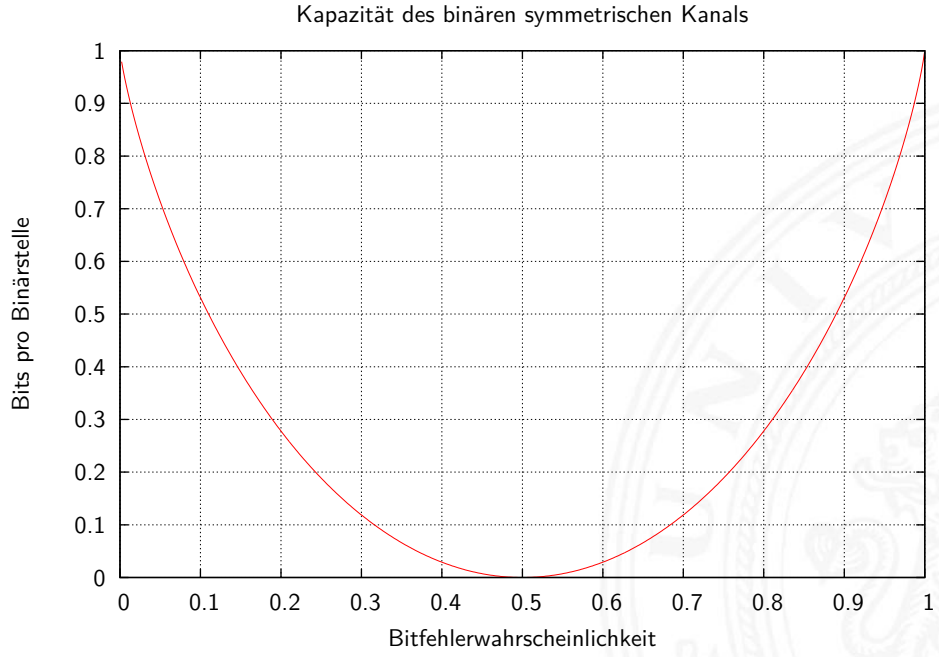


- Informationsquelle
- Sender mit möglichst effizienter Kanalcodierung
- gestörter und verrauschter Übertragungskanal
- Empfänger mit Decodierer und Fehlererkennung/-korrektur
- Informationssenke und -verarbeitung

Binärer symmetrischer Kanal

- Wahrscheinlichkeit der beiden Symbole 0 und 1 ist gleich $(\frac{1}{2})$
- Wahrscheinlichkeit P, dass bei Übertragungsfehlern aus einer 0 eine 1 wird = Wahrscheinlichkeit, dass aus einer 1 eine 0 wird
- Wahrscheinlichkeit eines Fehlers an Binärstelle i ist unabhängig vom Auftreten eines Fehlers an anderen Stellen
- Entropie des Fehlerverhaltens
  $H(F) = P \cdot \log_2(1/P) + (1-P) \cdot \log_2(1/(1-P))$
- Kanalkapazität ist $C = 1 - H(F)$

Kanalkapazität: Diagramm



Kanalkapazität: Konsequenzen

- bei $P = 0.5$ ist die Kanalkapazität $C = 0$

$\Rightarrow$ der Empfänger kann die empfangenen Daten nicht von einer zufälligen Sequenz unterscheiden

- bei $P > 0.5$ steigt die Kapazität wieder an
  (rein akademischer Fall: Invertieren aller Bits)

Die Kanalkapazität ist eine obere Schranke

- wird in der Praxis nicht erreicht (Fehler)
- Theorie liefert keine Hinweise, wie die fehlerfreie Übertragung praktisch durchgeführt werden kann

Shannon-Theorem

C. E. Shannon: *Communication in the Presence of Noise*; Proc. IRE, Vol.37, No.1, 1949

Gegeben:
binärer symmetrischer Kanal mit Störwahrscheinlichkeit P
und 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 Übertragungsrate $R(\mathcal{C})$ und $C(P) \geq R(\mathcal{C}) \geq R$ und
Fehlerdecodierwahrscheinlichkeit $< \epsilon$

Shannon-Theorem (cont.)

C. E. Shannon: *Communication in the Presence of Noise*; Proc. IRE, Vol.37, No.1, 1949

- leider liefert die Theorie keine Ideen zur Realisierung
- die Nachrichten müssen sehr lang sein
- der Code muss im Mittel sehr viele Fehler in jeder Nachricht korrigieren
- mittlerweile sehr nah am Limit: Turbo-Codes, LDPC Codes, usw.

Fehlererkennende / -korrigierende Codes

Motivation

- Informationstheorie
- Kanalkapazität
- Shannon-Theorem

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

$\Rightarrow$ fehlererkennende Codes

$\Rightarrow$ fehlerkorrigierende Codes

Fehlertypen

diverse mögliche Fehler bei der Datenübertragung

- Verwechslung eines Zeichens $a \rightarrow b$
- Vertauschen benachbarter Zeichen $ab \rightarrow ba$
- Vertauschen entfernter Zeichen $abc \rightarrow cba$
- Zwillings-/Bündelfehler $aa \rightarrow bb$
- usw.

- abhängig von der Technologie / der Art der Übertragung
  - Bündelfehler durch Kratzer auf einer CD
  - Bündelfehler bei Funk durch längere Störimpulse
  - Buchstabendreher beim „Eintippen“ eines Textes

Begriffe zur Fehlerbehandlung

- **Block-Code**: k-Informationsbits werden in n-Bits codiert
- **Faltungscodes**: ein Bitstrom wird in einen Codebitstrom höherer Bitrate codiert

- **linearer** (n, k)**-Code**: ein k-dimensionaler Unterraum des $GF(2)^n$
- **modifizierter Code**: eine oder mehrere Stellen eines linearen Codes werden systematisch verändert (d.h. im $GF(2)$ invertiert) Null- und Einsvektor gehören nicht mehr zum Code
- **nichtlinearer Code**: weder linear noch modifiziert

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

Boole'sche Algebra:

- basiert auf: UND, ODER, Negation
- UND $\approx$ Multiplikation
  ODER $\approx$ Addition
- aber: kein inverses Element für die ODER-Operation
  $\Rightarrow$ kein Körper

Galois-Field mit zwei Elementen: GF(2)

- Körper, zwei Verknüpfungen: UND und XOR
- UND als Multiplikation
  XOR als Addition *mod* 2
- additives Inverses existiert: $x \oplus x = 0$

Details: Mathe-Skript, Wikipedia, vdHeide: Technische Informatik 1

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
10.9 Codierung - Fehlererkennende Codes
64-040 Rechnerstrukturen

Begriffe zur Fehlerbehandlung (cont.)

- **systematischer Code**: wenn die zu codierende Information direkt (als Substring) im Codewort enthalten ist

- **zyklischer Code**
  - ein Block-Code (identische Wortlänge aller Codewörter)
  - zu jedem Codewort gehören auch sämtliche zyklischen Verschiebungen (Rotationen, z.b. rotate-left) des Wortes auch zum Code
  - bei serieller Übertragung erlaubt dies die Erkennung/Korrektur von Bündelfehlern

ARQ und FEC

- **Automatic Repeat Request** (ARQ): der Empfänger erkennt ein fehlerhaftes Symbol und fordert dies vom Sender erneut an
  - bidirektionale Kommunikation erforderlich
  - unpraktisch bei großer Entfernung / Echtzeitanforderungen
- **Vorwärtsfehlerkorrektur** (*Forward Error Correction*, FEC): die übertragene Information wird durch zusätzliche Redundanz (z.B. Prüfziffern) gesichert
  - der Empfänger erkennt fehlerhafte Codewörter und kann diese selbständig korrigieren
- je nach Einsatzzweck sind beide Verfahren üblich
- auch kombiniert

Hamming-Abstand

- **Hamming-Abstand**: die Anzahl der Stellen, an denen sich zwei Binärcodewörter der Länge w unterscheiden
- **Hamming-Gewicht**: Hamming-Abstand eines Codewortes vom Null-Wort
- Beispiel $\quad a = 0110\,0011$
  $\quad\quad\quad\quad b = 1010\,0111$

$\Rightarrow$ Hamming-Abstand von a und b ist 3

Hamming-Gewicht von b ist 5

- Java: `Integer.bitcount( a ^ b )`

Fehlererkennende und -korrigierende Codes

- Zur *Fehlererkennung* und *Fehlerkorrektur* ist eine Codierung mit Redundanz erforderlich
- Repräsentation enthält mehr Bits, als zur reinen Speicherung nötig wären

- Codewörter so wählen, dass sie paarweise mindestens den Hamming-Abstand d haben
- dieser Abstand heißt dann **Minimalabstand** d

- Fehlererkennung bis zu $(d-1)$ fehlerhaften Stellen
- Fehlerkorrektur bis zu $((d-1)/2)$ fehlerhaften Stellen

Prüfinformation

Man fügt den Daten **Prüfinformation** hinzu, oft **Prüfsumme** genannt

- zur Fehlerkennung
- zur Fehlerkorrektur
- zur Korrektur einfacher Fehler, Entdeckung schwerer Fehler

verschiedene Verfahren

- Prüfziffer, Parität
- Summenbildung
- CRC-Verfahren (*cyclic-redundancy check*)
- BCH-Codes (Bose, Ray-Chauduri, Hocquengham)
- RS-Codes (Reed-Solomon)

Paritätscode

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

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

- **gerade Parität** (*even parity*): $y_{even} = (z_1, \ldots, z_n, p)$
  $p(y_{even}) = (\sum_i y_i) \mod 2 = 0$
- **ungerade Parität** (*odd parity*): $y_{odd} = (z_1, \ldots, z_n, \overline{p})$
  $p(y_{odd}) = (\sum_i y_i) \mod 2 = 1$

Paritätscode: Eigenschaften

- in der Praxis meistens Einsatz der ungeraden Parität:
  pro Codewort y_{odd} mindestens je eine Null und Eins
- Hamming-Abstand zweier Codewörter im Paritätscode ist mindestens 2, weil sich bei Ändern eines Nutzbits jeweils auch die Parität ändert: $d = 2$
- Erkennung von Einbitfehlern möglich:
  Berechnung der Parität im Empfänger und Vergleich mit der erwarteten Parität
- Erkennung von (ungeraden) Mehrbitfehlern

Zweidimensionale Parität

- Anordnung der Daten / Informations-Bits als Matrix
- Berechnung der Parität für alle Zeilen und Spalten
- optional auch für Zeile/Spalte der Paritäten

- entdeckt 1-bit Fehler in allen Zeilen und Spalten
- erlaubt Korrektur von allen 1-bit und vielen n-bit Fehlern

- natürlich auch weitere Dimensionen möglich
  n-dimensionale Anordnung und Berechnung von n Paritätsbits

Zweidimensionale Parität: Beispiel

H	100 1000	0
A	100 0001	0
M	100 1101	0
M	100 1101	0
I	100 1001	1
N	100 1110	0
G	100 0111	0
I	100 1001	1

Fehlerfall	100 1000	0
	100 0101	0
	110 1101	0
	100 1101	0
	000 1001	1
	100 1110	0
	100 0111	0
	100 1000	1

- 64-Bits pro Symbol, davon 49 für Nutzdaten und 15 für Parität
- links: Beispiel für ein Codewort und Paritätsbits
- rechts: empfangenes Codewort mit vier Einzelfehlern,
  davon ein Fehler in den Paritätsbits

Zweidimensionale Parität: Einzelfehler

H	100 1000	0
A	100 0001	0
M	100 1101	0
M	100 1101	0
I	100 1001	1
N	100 1110	0
G	100 0111	0
I	100 1001	1

Fehlerfall	100 1000	0
	100 0101	0 1
	100 1101	0
	100 1101	0
	100 1001	1
	100 1110	0
	100 0111	0
	100 1001	1
	1	

- Empfänger: berechnet Parität und vergleicht mit gesendeter P.
- Einzelfehler: Abweichung in je einer Zeile und Spalte

$\Rightarrow$ Fehler kann daher zugeordnet und korrigiert werden

- Mehrfachfehler: nicht alle, aber viele erkennbar (korrigierbar)

Zweidimensionale Parität: Dezimalsystem

- Parität als Zeilen/Spaltensumme mod 10 hinzufügen

- Daten

3	7	4
5	4	8
1	3	5

Parität

3	7	4	4
5	4	8	7
1	3	5	9
9	4	7	

Fehlerfall

3	7	4	4
5	4	3	2
1	3	5	9
9	4	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

Prüfverfahren für ISBN-10

- ISBN-10 Zahl: $z_1, z_2, \ldots, z_{10}$
- Prüfsumme berechnen, Symbol X steht für Ziffer 10

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

- ISBN-Zahl zulässig, genau dann wenn

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

- Beispiel: 0-13-713336-7

$1 \cdot 0 + 2 \cdot 1 + 3 \cdot 3 + 4 \cdot 7 + 5 \cdot 1 + 6 \cdot 3 + 7 \cdot 3 + 8 \cdot 3 + 9 \cdot 6 = 161$

$161 \mod 11 = 7$

$161 + 10 \cdot 7 = 231$ $\qquad$ $231 \mod 11 = 0$

Prüfverfahren für ISBN: Fehlertypen

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

- Beispiel: vertausche i-te und j-te Ziffer (mit $i \neq j$)
  Prüfsumme: $\langle korrekt \rangle$ - $\langle falsch \rangle$
  $= i \cdot z_i + j \cdot z_j - j \cdot z_i - i \cdot z_j = (i - j) \cdot (z_i - z_k)$ mit $z_i \neq z_j$.

(3,1)-Wiederholungscode

- dreifache Wiederholung jedes Datenworts
- Generatormatrix ist
$$G = (111)$$
- Codewörter ergeben sich als Multiplikation von G mit dem Informationsvektor u (jeweils ein Bit)
  $u = 0: \quad x = (111) \cdot (0) = (000)$
  $u = 1: \quad x = (111) \cdot (1) = (111)$

- Verallgemeinerung als (n, k)-Wiederholungscode
- systematischer Code mit Minimalabstand $D = n$
- Decodierung durch Mehrheitsentscheid: 1-bit Fehlerkorrektur
- – Nachteil: geringe Datenrate

Hamming-Code

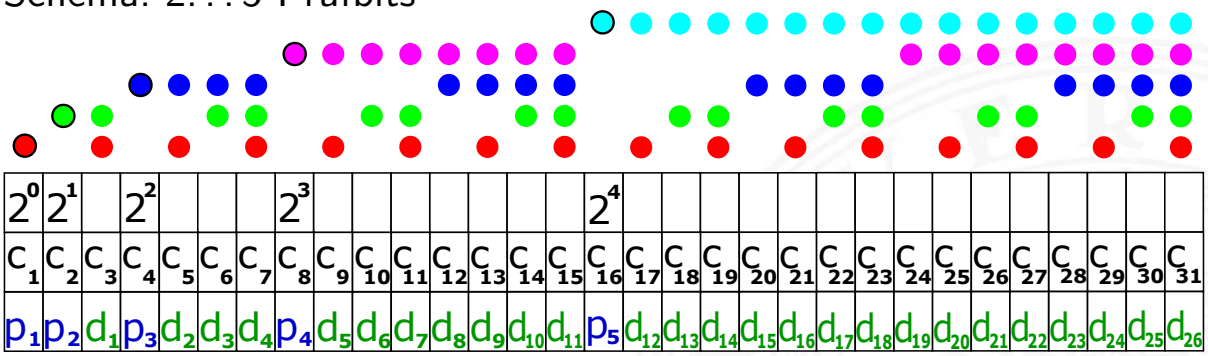
- Hamming-Abstand 3
- korrigiert 1-bit Fehler, erkennt (viele) 2-bit und 3-bit Fehler

Verfahren: Datenwort n-bit $(d_1, d_2, \ldots, d_n)$

1. bestimme kleinstes k mit $n \leq s^k - k - 1$
2. Prüfbits an Bitpositionen: $2^0, 2^1, \ldots, 2^{k-1}$
3. Originalbits an den übrigen Positionen
4. berechne Prüfbit i als mod 2-Summe der Bits deren Positionsnummer ein gesetztes i-bit enthält
5. dabei werden auch die Prüfbits berücksichtigt

Hamming-Code (cont.)

Schema: 2...5 Prüfbits



(7,4)-Hamming-Code

- $p_1 = d_1 \oplus d_2 \oplus d_4$
  $p_2 = d_1 \oplus d_3 \oplus d_4$
  $p_3 = d_2 \oplus d_3 \oplus d_4$

(7,4)-Hamming-Code

- sieben Codebits für je vier Datenbits
- linearer (7,4)-Block-Code
- Generatormatrix ist

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

- Codewort $c = G \cdot d$

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

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

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

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

$\Rightarrow$ „Fehlersyndrom“ liefert Information über Fehlerposition / -art

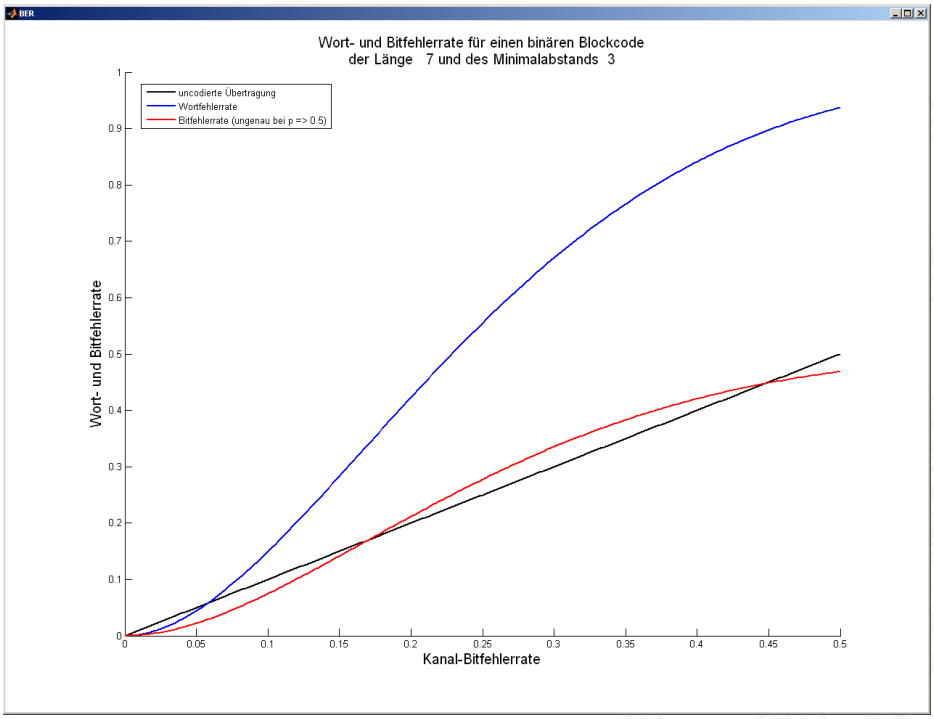
Fazit

- Hamming-Codes für diverse Wortlängen konstruierbar

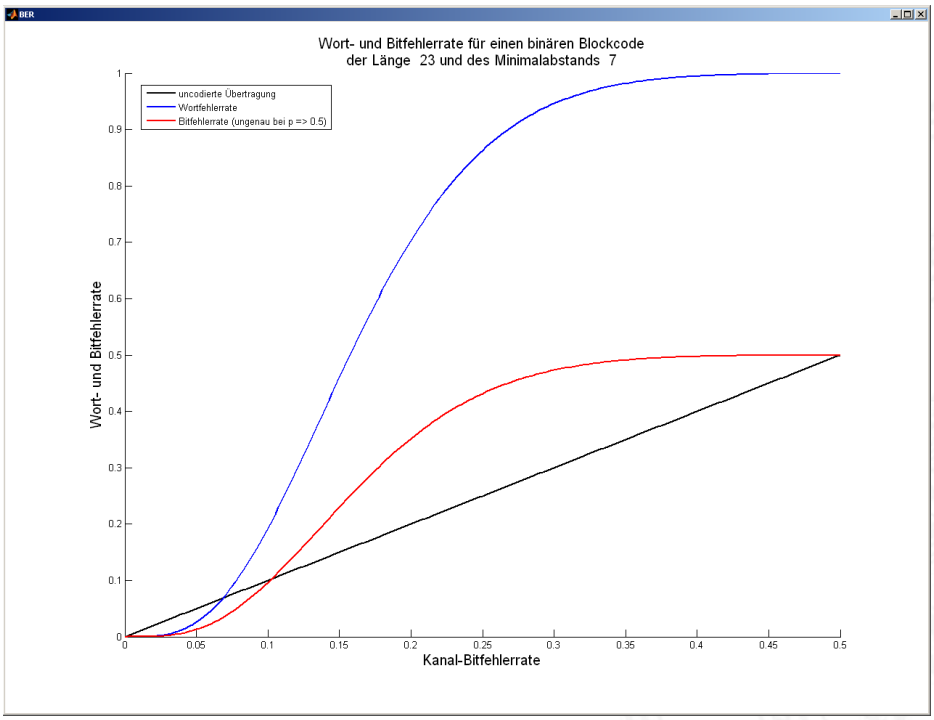
\+ einfaches Prinzip, einfach decodierbar

\− es existieren weit bessere Codes

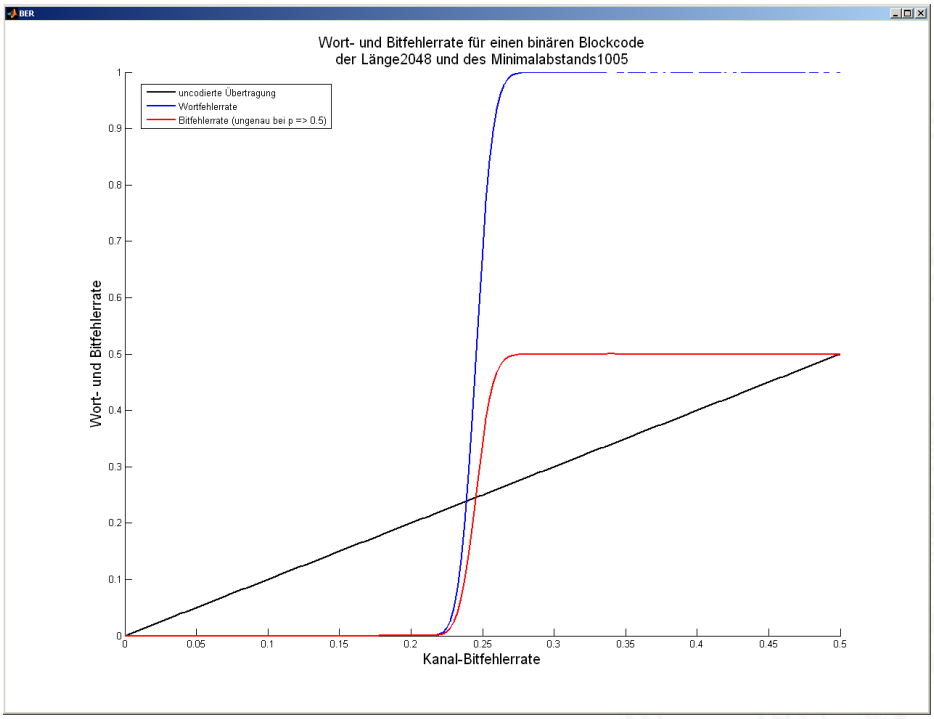
(7,4)-Hamming-Code: Fehlerrate



(23,7)-Hamming-Code: Fehlerrate



(2048,1005)-Zufalls-Code: Fehlerrate



Binärpolynome

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

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

- mit diesen Polynomen kann „gerechnet“ werden: Addition, Subtraktion, Multiplikation, Division
- Theorie: Galois-Felder

Zyklische Codes (CRC)

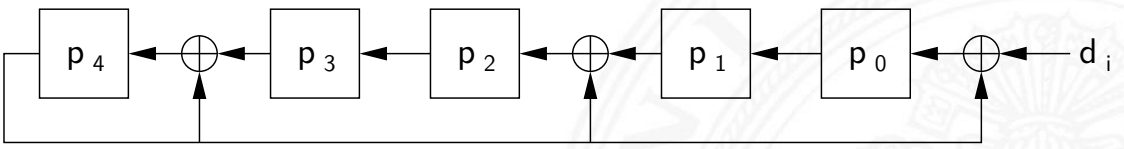
CRC (*Cyclic Redundancy Check*)

- Polynomdivision als Basis für CRC-Codes erzeugt Prüfbits
- zyklisch: Codewörter werden durch Schieben und Modifikation (mod 2 Summe) ineinander überführt
- Familie von Codes zur Fehlererkennung
  insbesondere auch zur Erkennung von Bündelfehlern
- in sehr vielen Codes benutzt
  - Polynom 0x04C11DB7 (CRC-32) in Ethernet, ZIP, PNG ...
  - weitere CRC-Codes in USB, ISDN, GSM, openPGP ...

Zyklische Codes (CRC) (cont.)

Sehr effiziente Software- oder Hardwarerealisierung

- rückgekoppelte Schieberegister und XOR
  LFSR (*Linear Feedback Shift Register*)
- Beispiel $x^5 + x^4 + x^2 + 1$



- Datenwort d_i wird bitweise in CRC-Check geschoben,
  Divisionsrest bildet Registerinhalt p_i
- Prüfbits p_i an Datenwort anhängen
  Test bei Empfänger: fehlerfrei, wenn $= 0$

Zyklische Codes (CRC) (cont.)

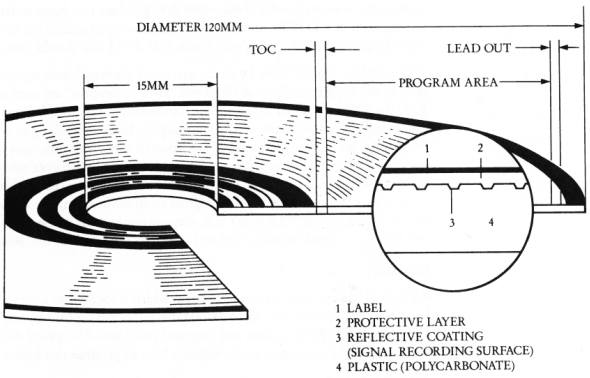
Je nach Polynom (# Prüfbits) unterschiedliche Güte

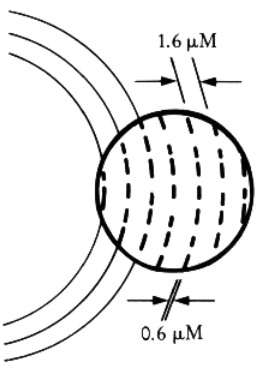
- Galois-Felder als mathematische Grundlage
- en.wikipedia.org/wiki/Cyclic_redundancy_check
  en.wikipedia.org/wiki/Computation_of_CRC
  de.wikipedia.org/wiki/Zyklische_Redundanzprüfung
  de.wikipedia.org/wiki/LFSR

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 $\approx$ 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 2352 Bytes Nutzdaten (und 98 Bytes *Subcode*)

- pro Sektor 784 Byte Fehlerkorrektur hinzufügen
- Interleaving gegen Burst-Fehler (z.B. Kratzer)
- Code kann bis 7000 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 2048 Bytes Nutzdaten, 276 Bytes RS-Fehlerschutz

Farbbilder: JPEG

Joint Picture Experts Group Bildformat (1992)

- für die Speicherung von Fotos / Bildern
- verlustbehaftet

mehrere Codierungsschritte

1.	Farbraumkonvertierung: RGB nach YUV	verlustbehaftet
2.	Aufteilung in Blöcke zu je 8x8 Pixeln	verlustfrei
3.	DCT (*discrete cosinus transformation*)	verlustfrei
4.	Quantisierung (einstellbar)	verlustbehaftet
5.	Huffman-Codierung	verlustfrei

Video: MPEG

Motion Picture Experts Group: Sammelname der Organisation und diverser aufeinander aufbauender Standards

Codierungsschritte für Video

1. Einzelbilder wie JPEG (YUV, DCT, Huffman)
2. Differenzbildung mehrerer Bilder (Bewegungskompensation)
3. *Group of Pictures* (*I*-Frames, *P*-Frames, *B*-Frames)
4. Zusammenfassung von Audio, Video, Metadaten im sogenannten PES (*Packetized Elementary Stream*)
5. *Transport-Stream* Format für robuste Datenübertragung

Digitales Fernsehen: DVB

Digital Video Broadcast: Sammelname für die europäischen Standards für digitales Fernsehen

Codierungsschritte

1. Videocodierung nach MPEG-2 (geplant: MPEG-4)
2. Multiplexing mehrerer Programme nach MPEG-TS
3. optional: Metadaten (Electronic Program Guide)
4. vier Varianten für die eigentliche Kanalcodierung
   - DVB-S: Satellite
   - DVB-C: Cable
   - DVB-T: Terrestrial
   - DVB-H: Handheld/Mobile

Literatur: Vertiefung

- Richard W. Hamming, *Information und Codierung*, VCH, 1987
- Klaus von der Heide, *Vorlesung: Technische Informatik 1 — interaktives Skript*, Universität Hamburg, FB Informatik, 2005
  `tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1`
- Klaus von der Heide, *Vorlesung: Digitale Datenübertragung*, Universität Hamburg, FB Informatik, 2005
  `tams.informatik.uni-hamburg.de/lectures/2005ss/vorlesung/Digit`
- William E. Ryan, Shu Lin, *Channel codes: classical and modern*, Cambridge University Press, 2009

Gliederung

Gliederung (cont.)

Gliederung (cont.)

Schaltfunktionen

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

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

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

Beschreibung von Schaltfunktionen

- textuelle Beschreibungen
  formale Notation, Schaltalgebra, Beschreibungssprachen
- tabellarische Beschreibungen
  Funktionstabelle, KV-Diagramme, ...
- graphische Beschreibungen
  Kantorovic-Baum (Datenflussgraph), Schaltbild, ...
- Verhaltensbeschreibungen $\Rightarrow$ „was“
- Strukturbeschreibungen $\Rightarrow$ „wie“

Funktionstabelle

- Tabelle mit Eingängen x_i und Ausgangswert $y = f(x)$
- Zeilen im Binärcode sortiert
- zugehöriger Ausgangswert eingetragen

x_3	x_2	x_1	$f(x)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Funktionstabelle (cont.)

- Kurzschreibweise: nur die Funktionswerte notiert
  $f(x_2, x_1, x_0) = \{0, 0, 1, 1, 0, 0, 1, 0\}$

- n Eingänge: Funktionstabelle umfasst 2^n Einträge
- Speicherbedarf wächst exponentiell mit n
  z.B.: 2^{33} Bit für 16-bit Addierer (16+16+1 Eingänge)

$\Rightarrow$ daher nur für kleine Funktionen geeignet

- Erweiterung auf don't-care Terme, s.u.

Verhaltensbeschreibung

- Beschreibung einer Funktion als Text über ihr Verhalten

- Problem: umgangssprachliche Formulierungen oft mehrdeutig

- logische Ausdrücke in Programmiersprachen
- Einsatz spezieller (Hardware-) Beschreibungssprachen
  z.B.: Verilog, VHDL, SystemC

umgangssprachlich: Mehrdeutigkeit

„Das Schiebedach ist ok (y), wenn der Öffnungskontakt (x_0) **oder** der Schließkontakt (x_1) funktionieren **oder beide nicht** aktiv sind (Mittelstellung des Daches)“

K. Henke, H.-D. Wuttke, *Schaltsysteme*

zwei mögliche Missverständnisse

- *oder*: als OR oder XOR?
- *beide nicht*: x_1 und x_0 nicht, oder x_1 nicht und x_2 nicht?

$\Rightarrow$ je nach Interpretation völlig unterschiedliche Schaltung

Strukturbeschreibung

- **Strukturbeschreibung**: eine Spezifikation der konkreten Realisierung einer Schaltfunktion

- vollständig geklammerte algebraische Ausdrücke
  $f = x_1 \oplus (x_2 \oplus x_3)$
- Datenflussgraphen
- Schaltpläne mit Gattern (s.u.)
- PLA-Format für zweistufige AND-OR Schaltungen (s.u.)
- ...

Funktional vollständige Basismenge

- Menge M von Verknüpfungen über $GF(2)$ heißt **funktional vollständig**, wenn die Funktionen $f, g \in T_2$:

$$f(x_1, x_2) = x_1 \oplus x_2$$
$$g(x_1, x_2) = x_1 \wedge x_2$$

  allein mit den in M enthaltenen Verknüpfungen geschrieben werden können

- Boole'sche Algebra: { AND, OR, NOT }
- Reed-Muller-Form: { AND, XOR, 1 }
- technisch relevant: { NAND }, { NOR }

Normalformen

- Jede Funktion kann auf beliebig viele Arten beschrieben werden

Suche nach Standardformen:

- in denen man alle Funktionen darstellen kann
- Darstellung mit universellen Eigenschaften
- eindeutige Repräsentation (einfache Überprüfung, ob gegebene Funktionen übereinstimmen)

- Beispiel: Darstellung von reellen Funktionen als Potenzreihe $f(x) = \sum_{i=0}^{\infty} a_i x^i$

Normalformen (cont.)

- Darstellung von reellen Funktionen als Potenzreihe
  $f(x) = \sum_{i=0}^{\infty} a_i x^i$

Normalform einer Boole'schen Funktion:

- analog zur Potenzreihe
- als Summe über Koeffizienten $\{0, 1\}$ und Basisfunktionen

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

mit $\hat{B}_1, \ldots, \hat{B}_{2^n}$ einer Basis des T^n

Definition: Normalform

- funktional vollständige Menge V der Verknüpfungen von $\{0, 1\}$
- Seien $\oplus, \otimes \in V$ und assoziativ

- Wenn sich alle $f \in T^n$ in der Form

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

schreiben lassen, so wird die Form als **Normalform** und die Menge der $\hat{B}_i$ als **Basis** bezeichnet.

- Menge von 2^n Basisfunktionen $\hat{B}_i$
  Menge von 2^{2^n} möglichen Funktionen f

Disjunktive Normalform (DNF)

- **Minterm**: die UND-Verknüpfung *aller* Schaltvariablen einer Schaltfunktion, die Variablen dürfen dabei negiert oder nicht negiert auftreten

- **Disjunktive Normalform**: die disjunktive Verknüpfung aller Minterme m mit dem Funktionswert 1

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

auch: *kanonische disjunktive Normalform*
sum-of-products (SOP)

Disjunktive Normalform: Minterme

- Beispiel: alle 2^3 Minterme für drei Variablen
- jeder Minterm nimmt nur für eine Belegung der Eingangsvariablen den Wert 1 an

x_3	x_2	x_1	Minterme
0	0	0	$\overline{x_3} \wedge \overline{x_2} \wedge \overline{x_1}$
0	0	1	$\overline{x_3} \wedge \overline{x_2} \wedge x_1$
0	1	0	$\overline{x_3} \wedge x_2 \wedge \overline{x_1}$
0	1	1	$\overline{x_3} \wedge x_2 \wedge x_1$
1	0	0	$x_3 \wedge \overline{x_2} \wedge \overline{x_1}$
1	0	1	$x_3 \wedge \overline{x_2} \wedge x_1$
1	1	0	$x_3 \wedge x_2 \wedge \overline{x_1}$
1	1	1	$x_3 \wedge x_2 \wedge x_1$

Disjunktive Normalform: Beispiel

x_3	x_2	x_1	$f(x)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

- Zeilen der Funktionstabelle entsprechen jeweiligem Minterm
- für f sind nur drei Koeffizienten der DNF gleich 1

$\Rightarrow$ DNF: $f(x) = (\overline{x_3} \wedge x_2 \wedge \overline{x_1}) \vee (\overline{x_3} \wedge x_2 \wedge x_1) \vee (x_3 \wedge x_2 \wedge \overline{x_1})$

Allgemeine disjunktive Form

- **disjunktive Form** (sum-of-products): die disjunktive Verknüpfung (ODER) von Termen. Jeder Term besteht aus der UND-Verknüpfung von Schaltvariablen, die entweder direkt oder negiert auftreten können
- entspricht dem Zusammenfassen („Minimierung“) von Termen aus der disjunktiven Normalform
- disjunktive Form ist nicht eindeutig (keine Normalform)

- Beispiel

  DNF $f(x) = (\overline{x_3} \wedge x_2 \wedge \overline{x_1}) \vee (\overline{x_3} \wedge x_2 \wedge x_1) \vee (x_3 \wedge x_2 \wedge \overline{x_1})$

  minimierte disjunktive Form $f(x) = (\overline{x_3} \wedge x_2) \vee (x_3 \wedge x_2 \wedge \overline{x_1})$

Allgemeine disjunktive Form

- **disjunktive Form** (sum-of-products): die disjunktive Verknüpfung (ODER) von Termen. Jeder Term besteht aus der UND-Verknüpfung von Schaltvariablen, die entweder direkt oder negiert auftreten können
- entspricht dem Zusammenfassen („Minimierung“) von Termen aus der disjunktiven Normalform
- disjunktive Form ist nicht eindeutig (keine Normalform)

- Beispiel

  DNF $f(x) = (\overline{x_3} \wedge x_2 \wedge \overline{x_1}) \vee (\overline{x_3} \wedge x_2 \wedge x_1) \vee (x_3 \wedge x_2 \wedge \overline{x_1})$

  minimierte disjunktive Form $f(x) = (\overline{x_3} \wedge x_2) \vee (x_3 \wedge x_2 \wedge \overline{x_1})$

  $f(x) = (x_2 \wedge \overline{x_1}) \vee (\overline{x_3} \wedge x_2 \wedge x_1)$

Konjunktive Normalform (KNF)

- **Maxterm**: die ODER-Verknüpfung *aller* Schaltvariablen einer Schaltfunktion, die Variablen dürfen dabei negiert oder nicht negiert auftreten

- **Konjunktive Normalform**: die konjunktive Verknüpfung aller Maxterme μ mit dem Funktionswert 0

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

auch: *kanonische konjunktive Normalform*
product-of-sums (POS)

Konjunktive Normalform: Maxterme

- Beispiel: alle 2^3 Maxterme für drei Variablen
- jeder Maxterm nimmt nur für eine Belegung der Eingangsvariablen den Wert 0 an

x_3	x_2	x_1	Maxterme
0	0	0	$x_3 \vee x_2 \vee x_1$
0	0	1	$x_3 \vee x_2 \vee \overline{x_1}$
0	1	0	$x_3 \vee \overline{x_2} \vee x_1$
0	1	1	$x_3 \vee \overline{x_2} \vee \overline{x_1}$
1	0	0	$\overline{x_3} \vee x_2 \vee x_1$
1	0	1	$\overline{x_3} \vee x_2 \vee \overline{x_1}$
1	1	0	$\overline{x_3} \vee \overline{x_2} \vee x_1$
1	1	1	$\overline{x_3} \vee \overline{x_2} \vee \overline{x_1}$

Konjunktive Normalform: Beispiel

x_3	x_2	x_1	f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

- Zeilen der Funktionstabelle $\approx$ „invertierter“ Maxterm
- für f sind fünf Koeffizienten der KNF gleich 0

$\Rightarrow$ KNF: $f(x) = (x_3 \vee x_2 \vee x_1) \wedge (x_3 \vee x_2 \vee \overline{x_1}) \wedge (\overline{x_3} \vee x_2 \vee x_1) \wedge (\overline{x_3} \vee x_2 \vee \overline{x_1}) \wedge (\overline{x_3} \vee \overline{x_2} \vee \overline{x_1})$

Allgemeine konjunktive Form

- **konjunktive Form** (product-of-sums): die konjunktive Verknüpfung (UND) von Termen. Jeder Term besteht aus der ODER-Verknüpfung von Schaltvariablen, die entweder direkt oder negiert auftreten können
- entspricht dem Zusammenfassen („Minimierung“) von Termen aus der konjunktiven Normalform
- konjunktive Form ist nicht eindeutig (keine Normalform)

- Beispiel
  KNF $f(x) = (x_3 \vee x_2 \vee x_1) \wedge (x_3 \vee x_2 \vee \overline{x_1}) \wedge (\overline{x_3} \vee x_2 \vee x_1) \wedge (\overline{x_3} \vee x_2 \vee \overline{x_1}) \wedge (\overline{x_3} \vee \overline{x_2} \vee \overline{x_1})$
  minimierte konjunktive Form
  $f(x) = (x_3 \vee x_2) \wedge (x_2 \vee x_1) \wedge (\overline{x_3} \vee \overline{x_1})$

Reed-Muller-Form

- **Reed-Muller-Form**: die additive Verknüpfung aller Reed-Muller-Terme mit dem Funktionswert 1

$$f = \bigoplus_{i=1}^{2^n} \hat{f}_i \cdot RM(i)$$

- mit den Reed-Muller Basisfunktionen $RM(i)$

- Erinnerung: Addition im GF(2) ist die XOR-Operation

Reed-Muller-Form: Basisfunktionen

- Basisfunktionen sind:

$$\begin{aligned}
&\{1\}, && \text{(0 Variablen)}\\
&\{1, \quad x_1\}, && \text{(1 Variable)}\\
&\{1, \quad x_1, \quad x_2, x_2x_1\}, && \text{(2 Variablen)}\\
&\{1, \quad x_1, \quad x_2, x_2x_1, \quad x_3, x_3x_1, x_3x_2, x_3x_2x_1\}, && \text{(3 Variablen)}\\
&\ldots\\
&\{RM(n-1), \quad x_n \cdot RM(n-1)\} && \text{(n Variablen)}
\end{aligned}$$

- rekursive Bildung: bei n bit alle Basisfunktionen von $(n-1)$-bit und zusätzlich das Produkt von x_n mit den Basisfunktionen von $(n-1)$-bit

Reed-Muller-Form: Umrechnung

Umrechnung von gegebenem Ausdruck in Reed-Muller Form?

- Ersetzen der Negation: $\overline{a} = a \oplus 1$

  Ersetzen der Disjunktion: $a \vee b = a \oplus b \oplus ab$

  Ausnutzen von: $a \oplus a = 0$

- Beispiel

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

Reed-Muller-Form: Transformationsmatrix

- lineare Umrechnung zwischen Funktion f, bzw. der Funktionstabelle (distributive Normalform), und RMF
- Transformationsmatrix A kann rekursiv definiert werden (wie die RMF-Basisfunktionen)
- Multiplikation von A mit f ergibt Koeffizientenvektor r der RMF

$$r = A \cdot f, \quad \text{und} \quad f = A \cdot r$$

- weitere Details in:
  Klaus von der Heide, *Vorlesung: Technische Informatik T1*,
  `tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1`
- Hinweis: Beziehung zu Fraktalen (Sirpinski-Dreieck)

Reed-Muller-Form: Transformationsmatrix (cont.)

- $r = A \cdot f$ (und $A \cdot A = I$, also $f = A \cdot r$ (!))

$$A_0 = (1)$$

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

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

Reed-Muller-Form: Transformationsmatrix (cont.)

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

...

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

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: $f(x) = x_2 \oplus x_3 x_2 x_1$
- häufig kompaktere Darstellung als DNF oder KNF

Reed-Muller-Form: Beispiel (cont.)

- $f(x_3, x_2, x_1) = \{0, 0, 1, 1, 0, 0, 1, 0\}$ (Funktionstabelle)
- Aufstellen von A_3 und Ausmultiplizieren

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

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

Grafische Darstellung: Entscheidungsbäume

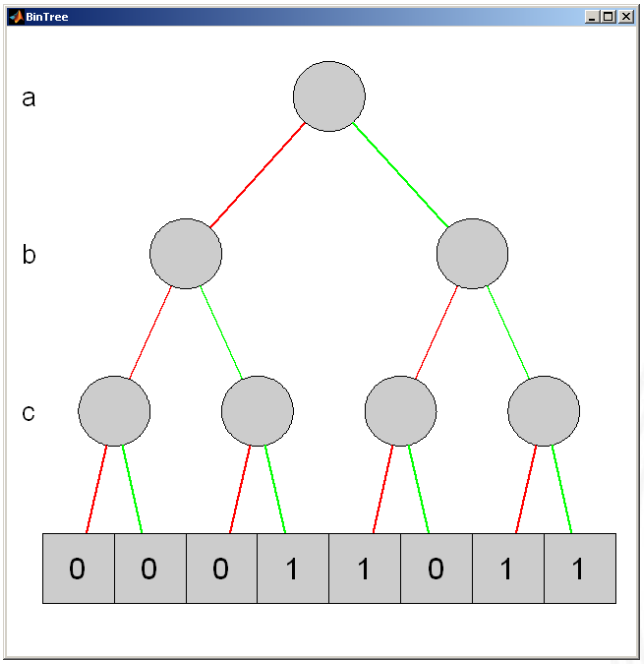
- Darstellung einer Schaltfunktion als Baum/Graph
- jeder Knoten ist einer Variablen zugeordnet
  jede Verzweigung entspricht einer `if-then-else`-Entscheidung

- vollständige Baum realisiert Funktionstabelle

\+ einfaches Entfernen/Zusammenfassen redundanter Knoten

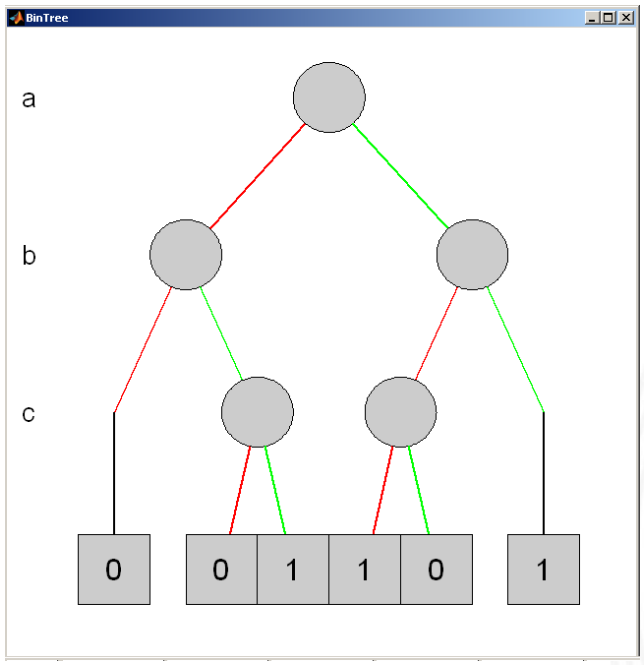
- Beispiel: Multiplexer
  $f(a, b, c) = (a \wedge \overline{c}) \vee (b \wedge c)$

Entscheidungsbaum: Beispiel



- $f(a,b,c) = (a \wedge \overline{c}) \vee (b \wedge c)$
- rot: 0-Zweig
  grün: 1-Zweig

Entscheidungsbaum: Beispiel (cont.)



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

$\Rightarrow$ Knoten entfernt

- rot: 0-Zweig
  grün: 1-Zweig

Reduced Ordered Binary-Decision Diagrams (ROBDD)

Binäres Entscheidungsdiagramm

- Variante des Entscheidungsbaums
- vorab gewählte Variablenordnung (*ordered*)
- redundante Knoten werden entfernt (*reduced*)
- ein ROBDD ist eine Normalform für eine Funktion

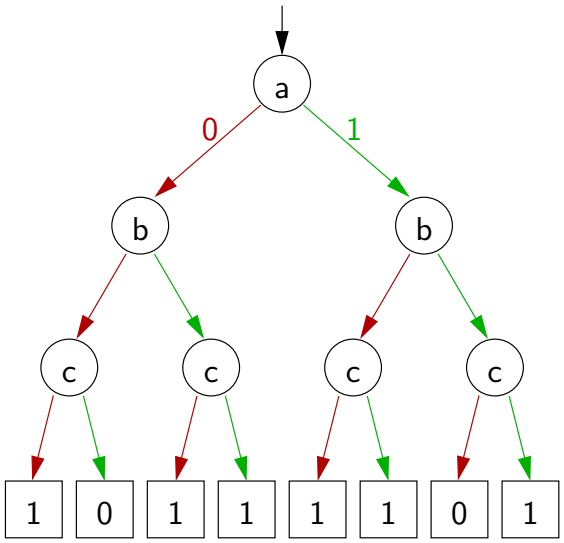
- viele praxisrelevante Funktionen sehr kompakt darstellbar $O(n)..O(n^2)$ Knoten bei n Variablen
- wichtige Ausnahme: n-bit Multiplizierer ist $O(2^n)$
- derzeit das Standardverfahren zur Manipulation von (großen) Schaltfunktionen

R. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Trans. Computers (1986)

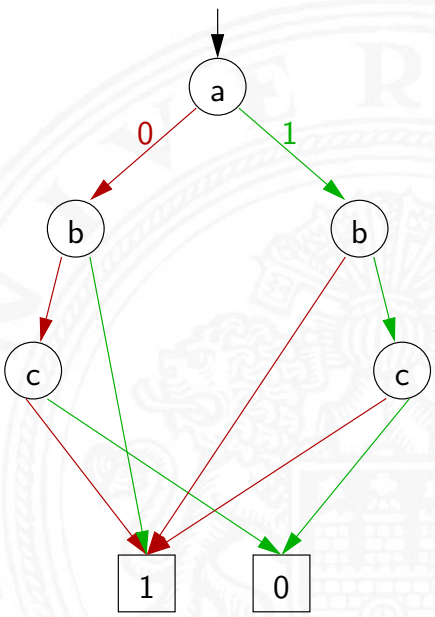
ROBDD vs. Entscheidungsbaum

Entscheidungsbaum

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

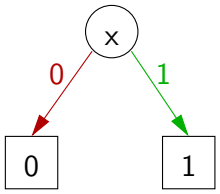


ROBDD

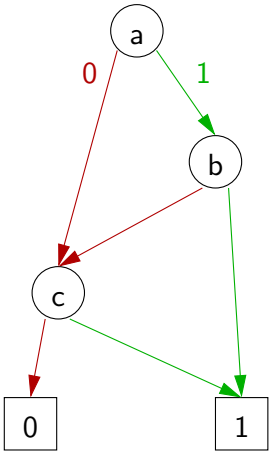


ROBDD: Beispiele

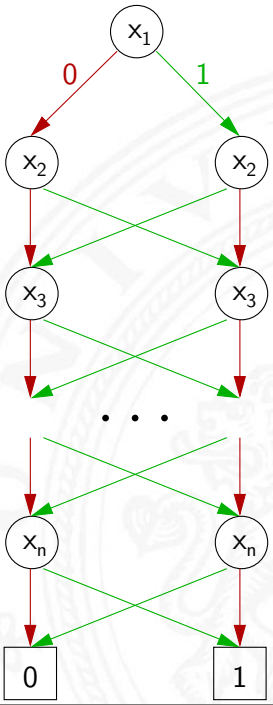
$f(x) = x$



$g = (a\ b) \vee c$



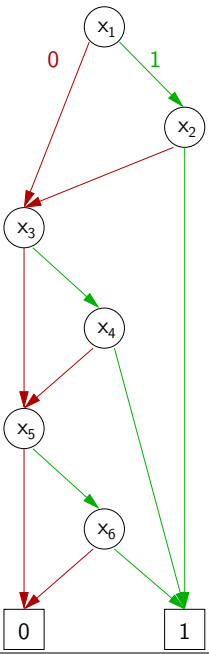
Parität $p = x_1 \oplus x_2 \oplus \ldots x_n$



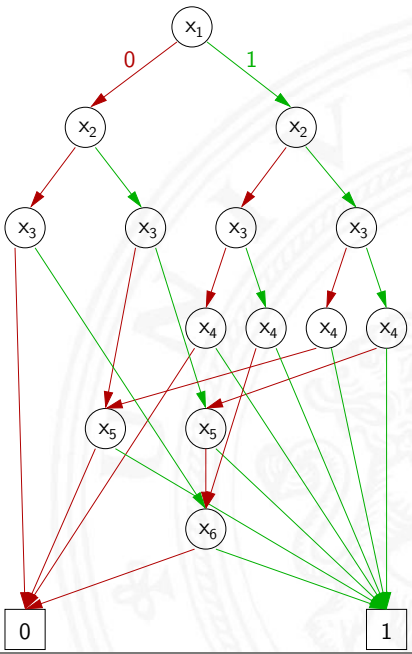
ROBDD: Problem der Variablenordnung

- Anzahl der Knoten oft stark abhängig von der Variablenordnung

$f = x_1\ x_2 \vee x_3\ x_4 \vee x_5\ x_6$



$g = x_1\ x_4 \vee x_2\ x_5 \vee x_3\ x_6$



Minimierung von Schaltfunktionen

- mehrere (beliebig viele) Varianten zur Realisierung einer gegebenen Schaltfunktion bzw. eines Schaltnetzes

Minimierung des Realisierungsaufwandes:

- diverse Kriterien, technologieabhängig

- Hardwarekosten (Anzahl der Gatter)
- Hardwareeffizienz (z.B. NAND statt XOR)
- Geschwindigkeit (Anzahl der Stufen, Laufzeiten)
- Testbarkeit (Erkennung von Produktionsfehlern)
- Robustheit (z.B. ionisierende Strahlung)

Algebraische Minimierungsverfahren

- Vereinfachung der gegebenen Schaltfunktionen durch Anwendung der Gesetze der Boole'schen Algebra

- im Allgemeinen nur durch Ausprobieren
- ohne Rechner sehr mühsam
- keine allgemeingültigen Algorithmen bekannt

- Heuristische Verfahren
  - Suche nach *Primimplikanten* (= kürzeste Konjunktionsterme)
  - Quine-McCluskey-Verfahren und Erweiterungen

Algebraische Minimierung: Beispiel

- Ausgangsfunktion in DNF

$$y(x) = \overline{x_3}x_2x_1\overline{x_0} \vee \overline{x_3}x_2x_1x_0 \vee x_3\overline{x_2}\,\overline{x_1}x_0 \vee x_3\overline{x_2}x_1\overline{x_0} \vee x_3\overline{x_2}x_1x_0 \vee x_3x_2\overline{x_1}x_0 \vee x_3x_2x_1\overline{x_0} \vee x_3x_2x_1x_0$$

- Zusammenfassen benachbarter Terme liefert

$$y(x) = \overline{x_3}x_2x_1 \vee x_3\overline{x_2}x_0 \vee x_3\overline{x_2}x_1 \vee x_3x_2x_0 \vee x_3x_2x_1$$

- aber bessere Lösung ist möglich (weiter Umformen)

$$y(x) = x_2x_1 \vee x_3x_0 \vee x_3x_1$$

Grafische Minimierungsverfahren

- Darstellung einer Schaltfunktion im KV-Diagramm
- Interpretation als disjunktive Normalform

- Zusammenfassen benachbarter Terme durch **Schleifen**
- alle 1-Terme mit möglichst wenigen Schleifen abdecken
- Ablesen der minimierten Funktion, wenn keine weiteren Schleifen gebildet werden können

- beruht auf der menschlichen Fähigkeit, benachbarte Flächen auf einen Blick zu „sehen“
- bei mehr als 6 Variablen nicht mehr praktikabel

Erinnerung: Karnaugh-Veitch-Diagramm

x3 x2 \ x1 x0	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

x3 x2 \ x1 x0	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

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

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

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

x1 \ x0	0	1
0	00	01
1	10	11

x2 \ x1 x0	00	01	11	10
0	000	001	011	010
1	100	101	111	110

x3 x2 \ x1 x0	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

KV-Diagramm für Schaltfunktionen

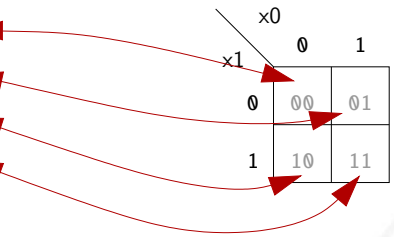
- Funktionswerte in zugehöriges Feld im KV-Diagramm eintragen
- Werte 0 und 1
  don't-care „∗“ für nicht spezifizierte Werte (!)
- 2D-Äquivalent zur Funktionstabelle

- praktikabel für 3..6 Eingänge
- fünf Eingänge: zwei Diagramme a 4×4 Felder
  sechs Eingänge: vier Diagramme a 4×4 Felder

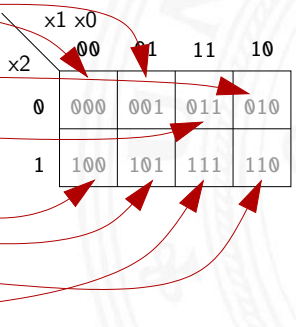
- viele Strukturen „auf einen Blick“ erkennbar

KV-Diagramm: Zuordnung zur Funktionstabelle

x1	x0	y=f(x)
0	0	f(0 0)
0	1	f(0 1)
1	0	f(1 0)
1	1	f(1 1)



x2	x1	x0	y=f(x)
0	0	0	f(0 0 0)
0	0	1	f(0 0 1)
0	1	0	f(0 1 0)
0	1	1	f(0 1 1)
1	0	0	f(1 0 0)
1	0	1	f(1 0 1)
1	1	0	f(1 1 0)
1	1	1	f(1 1 1)



KV-Diagramm: Eintragen aus Funktionstabelle

x1	x0	y=f(x)
0	0	0
0	1	0
1	0	1
1	1	1

x1 \ x0	0	1
0	0	0
1	1	1

x2	x1	x0	y=f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

x2 \ x1 x0	00	01	11	10
0	0	0	1	1
1	0	0	0	1

KV-Diagramm: Beispiel

x3 x2 \ x1 x0	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

x3 x2 \ x1 x0	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	0	0	1	0
10	0	0	1	0

- Beispielfunktion in DNF mit vier Termen:
  $f(x) = (\overline{x_3}x_2\overline{x_1}\overline{x_0}) \vee (\overline{x_3}\overline{x_2}x_1\overline{x_0}) \vee (x_3\overline{x_2}x_1x_0) \vee (x_3x_2x_1x_0)$
- Werte aus Funktionstabelle an entsprechender Stelle ins Diagramm eintragen

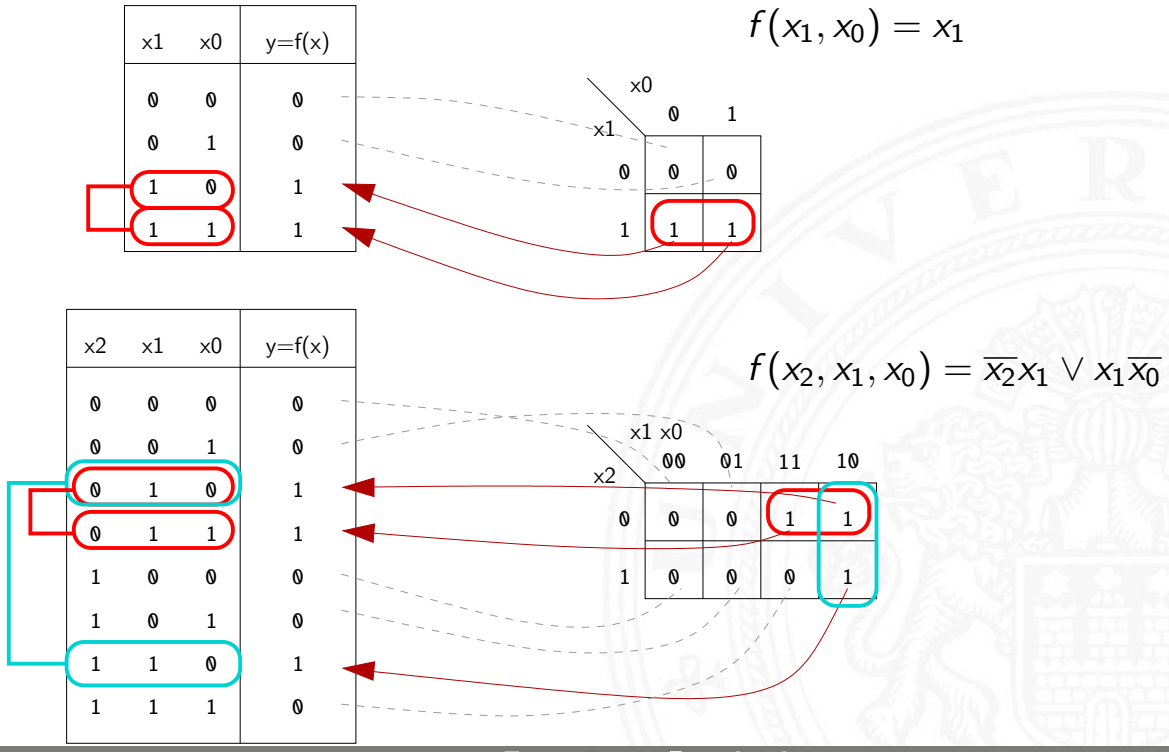
Schleifen: Zusammenfassen benachbarter Terme

- benachbarte Felder unterscheiden sich um 1-Bit
- falls benachbarte Terme beide 1 sind $\Rightarrow$ Funktion hängt an dieser Stelle nicht von der betroffenen Variable ab
- zugehörige (Min-) Terme können zusammengefasst werden

- Erweiterung auf vier benachbarte Felder (4x1 1x4 2x2)
  Erweiterung auf acht benachbarte Felder (4x2 2x4) usw.
- aber keine Dreier- Fünfergruppen, usw. (Gruppengröße 2^i)

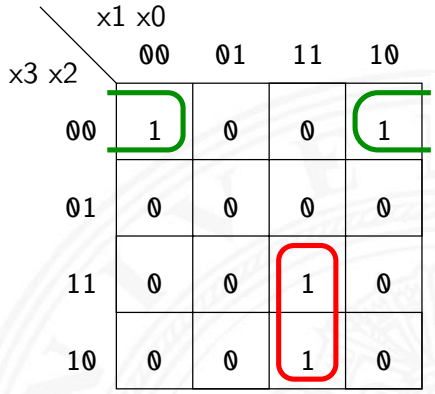
- Nachbarschaft auch „außen herum“
- mehrere Schleifen dürfen sich überlappen

Schleifen: Ablesen der Schleifen



Schleifen: Ablesen der Schleifen (cont.)

x3 x2 \ x1 x0	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 $(\overline{x_3 x_2 x_0}) = (\overline{x_3 x_2 x_1 x_0}) \vee (\overline{x_3 x_2} x_1 \overline{x_0})$
  rot 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 $f(x) = (\overline{x_3 x_2 x_0}) \vee (x_3 x_1 x_0)$

Schleifen: interaktive Demonstration

- Applet zur Minimierung mit KV-Diagrammen
  `tams.informatik.uni-hamburg.de/applets/kvd`

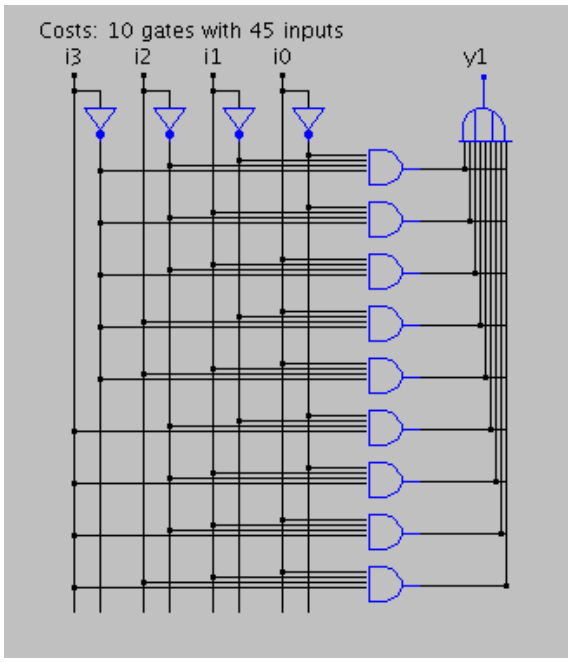
1. Auswahl oder Eingabe einer Funktion (2..6 Variablen)
2. Interaktives Setzen und Erweitern von Schleifen
   (click, shift+click, control+click)
3. Anzeige der zugehörigen Hardwarekosten und Schaltung

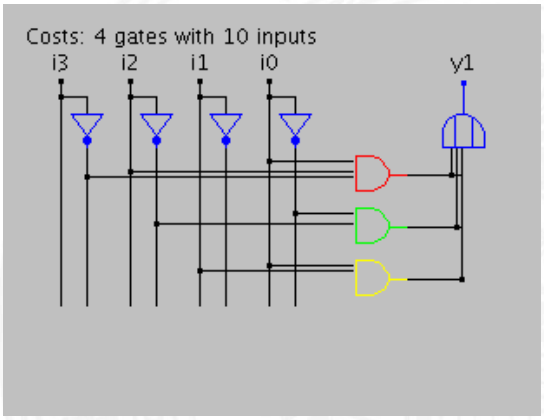
- Achtung: andere Anordnung der Eingangsvariablen als im Skript

$\Rightarrow$ entsprechend andere Anordnung der Terme im KV-Diagramm
Prinzip bleibt aber gleich

KV-Diagramm Applet: Screenshots

KV-Diagramm Applet: zugehörige Hardwarekosten



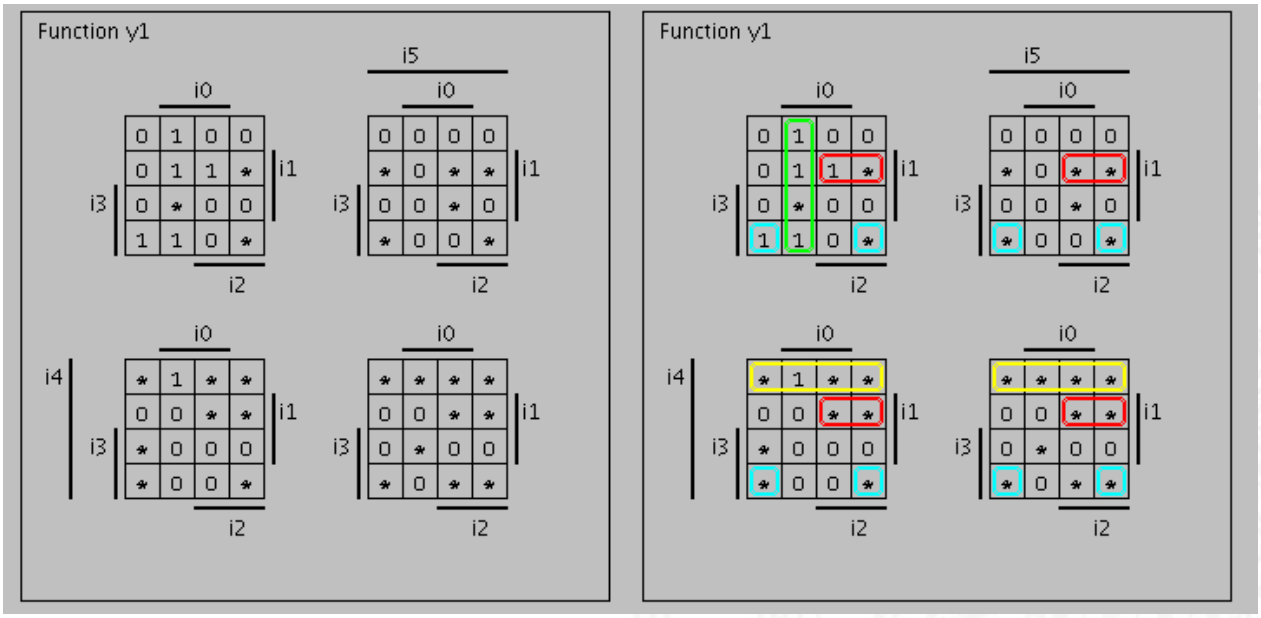


Don't-Care Terme

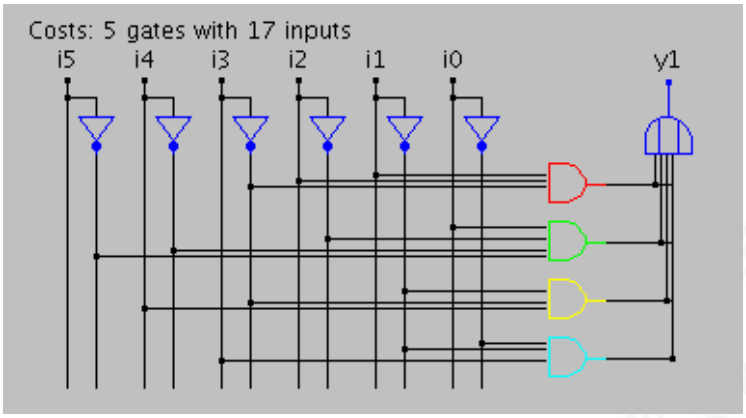
- in der Praxis: viele Schaltfunktionen unvollständig definiert
  weil bestimmte Eingangskombinationen nicht vorkommen
- zugehörige Terme als **Don't Care** markieren
  typisch: Sternchen „$*$“ in Funktionstabelle/KV-Diagramm

- solche Terme bei Minimierung nach Wunsch auf 0/1 setzen
- Schleifen dürfen Don't Cares enthalten
- Schleifen möglichst groß

KV-Diagramm Applet: 6 Variablen, *Don't Cares*



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



- Schaltung und Realisierungsaufwand (# Gatter, Eingänge) nach der Minimierung

Quine-McCluskey-Algorithmus

- Algorithmus zur Minimierung einer Schaltfunktion
- Notation der Terme in Tabellen, n Variablen
- Prinzip entspricht der Minimierung im KV-Diagramm aber auch geeignet für mehr als sechs Variablen
- Grundlage gängiger Minimierungsprogramme

- Sortieren der Terme nach Hamming-Distanz
- Erkennen der unverzichtbaren Terme („Primimplikanten“)
- Aufstellen von Gruppen benachbarter Terme (mit Distanz 1)
- Zusammenfassen geeigneter benachbarter Terme

Becker, Drechsler, Molitor, *Technische Informatik: Eine Einführung*

Schiffmann, Schmitz, *Technische Informatik*

Gliederung

Gliederung (cont.)

Gliederung (cont.)

Schaltnetze: Definition

- **Schaltnetz** oder auch
  kombinatorische Schaltung (*combinational logic circuit*):
  ein digitales System mit n-Eingängen $(b_1, b_2, \ldots, b_n)$ und m-Ausgängen $(y_1, y_2, \ldots, y_m)$, dessen Ausgangsvariablen zu jedem Zeitpunkt nur von den aktuellen Werten der Eingangsvariablen abhängen

  Beschreibung als Vektorfunktion $\vec{y} = F(\vec{b})$

- Hinweis: ein Schaltnetz darf keine Rückkopplungen enthalten
- in der Praxis können Schaltnetze nicht statisch betrachtet werden: Gatterlaufzeiten spielen eine Rolle

Elementare digitale Schaltungen

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

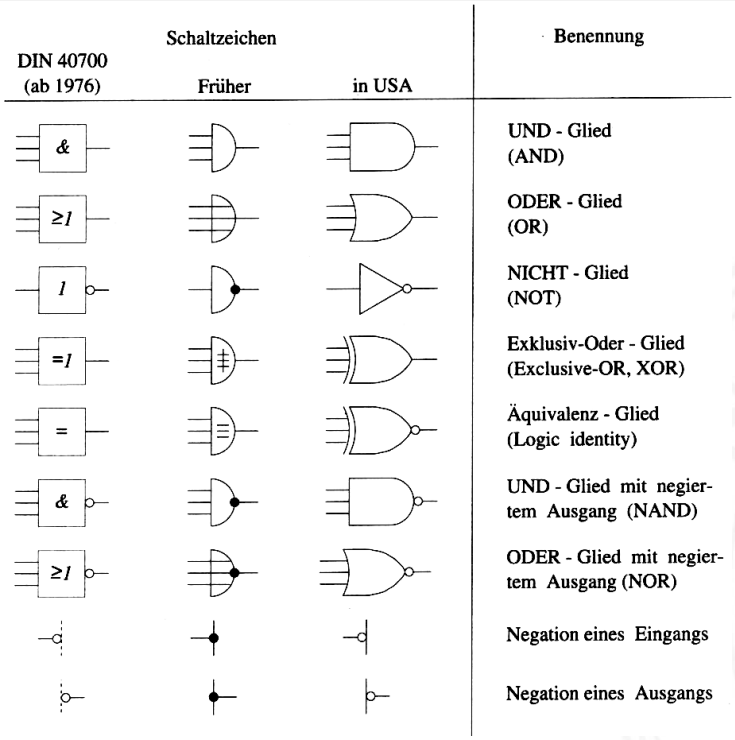
- Schaltnetze (mehrere Ausgänge)
- Beispiele

- Arithmetisch/Logische Operationen

Schaltpläne (*schematics*)

- standardisierte Methode zur Darstellung von Schaltungen
- genormte Symbole für Komponenten
  - Spannungs- und Stromquellen, Messgeräte
  - Schalter und Relais
  - Widerstände, Kondensatoren, Spulen
  - Dioden, Transistoren (bipolar, MOS)
  - **Gatter**: logische Grundoperationen (UND, ODER, usw.)
  - **Flipflops**: Speicherglieder
- Verbindungen
  - Linien für Drähte (Verbindungen)
  - Lötpunkte für Drahtverbindungen
  - dicke Linien für *n*-bit Busse, Anzapfungen, usw.
- komplexe Bausteine ggf. hierarchisch

Schaltsymbole



Schiffmann, Schmitz,
Technische Informatik 1

Logische Gatter

- **Logisches Gatter** (*logic gate*): die Bezeichnung für die Realisierung einer logischen Grundfunktion als gekapselte Komponente (in einer gegebenen Technologie)

- 1 Eingang: Treiberstufe/Verstärker und Inverter (Negation)
- 2 Eingänge: AND/OR, NAND/NOR, XOR, XNOR
- 3- und mehr Eingänge: AND/OR, NAND/NOR, Parität
- Multiplexer

- mindestens Gatter für eine vollständige Basismenge erforderlich
- in Halbleitertechnologie sind NAND/NOR besonders effizient

Schaltplan-Editor und -Simulator

Spielerischer Zugang zu digitalen Schaltungen:

- mit Experimentierkasten oder im Logiksimulator
- interaktive Simulation erlaubt direktes Ausprobieren
- Animation und Visualisierung der logischen Werte
- „entdeckendes Lernen“

- Diglog: `www.cs.berkeley.edu/~lazzaro/chipmunk`
- Hades: `tams.informatik.uni-hamburg.de/applets/hades/webdemos`
  - Demos laufen im Browser (Java erforderlich)
  - Grundschaltungen, Gate-Level Circuits...

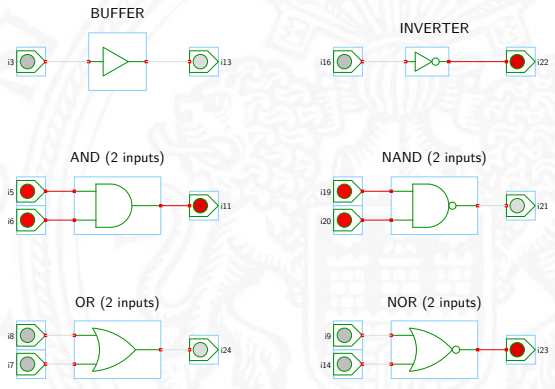
`tams.informatik.uni-hamburg.de/applets/hades/webdemos/toc.html`

Hades: Grundkomponenten

- Vorführung des Simulators

- Eingang: Schalter + Anzeige („*Ipin*“)
- Ausgang: Anzeige („*Opin*“)
- Taktgenerator
- PowerOnReset
- Anzeige / Leuchtdiode
- Siebensegmentanzeige

...



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	$\Rightarrow$ keine Treiber
magenta	undefined-X	$\Rightarrow$ Kurzschluss, ungültiger Wert
cyan	unknown-U	$\Rightarrow$ nicht initialisiert

Hades: Bedienung

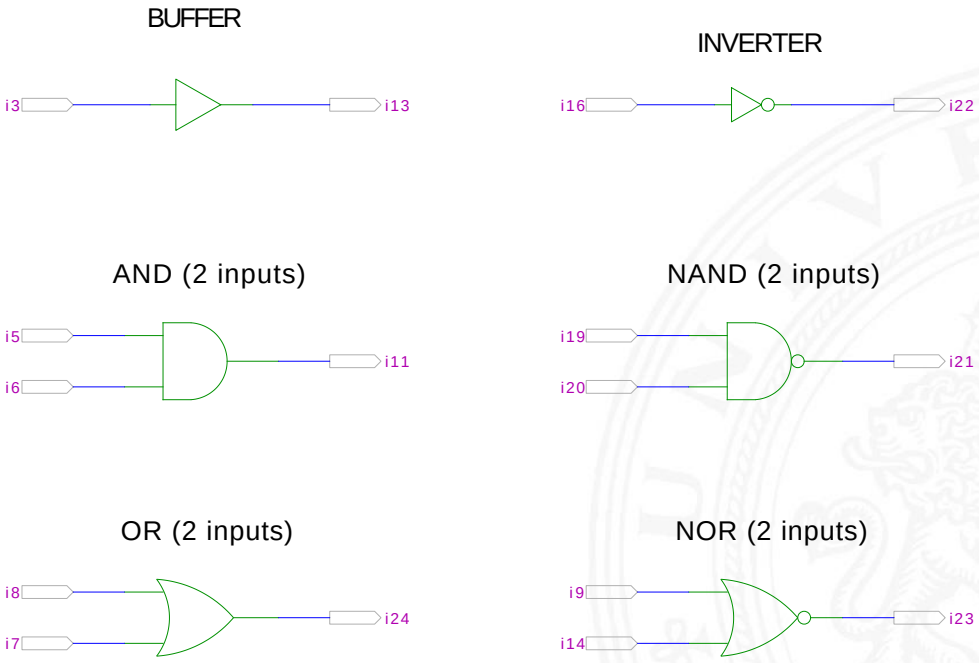
- Menü: Anzeigeoptionen, Edit-Befehle, usw.

- Editorfenster mit Popup-Menü für häufige Aktionen
- Rechtsklick auf Komponenten öffnet Eigenschaften/Parameter (*property-sheets*)
- optional „tooltips“ (enable im Layer-Menü)

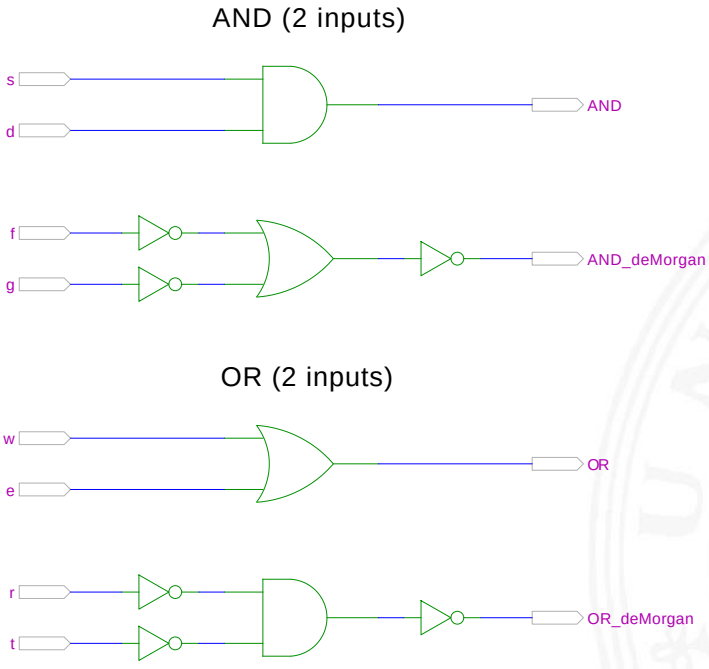
- Simulationssteuerung: *run*, *pause*, *rewind*
- Anzeige der aktuellen Simulationszeit

- Details siehe Hades-Webseite: Kurzreferenz, Tutorial
  `tams.informatik.uni-hamburg.de/applets/hades/webdemos/docs.html`

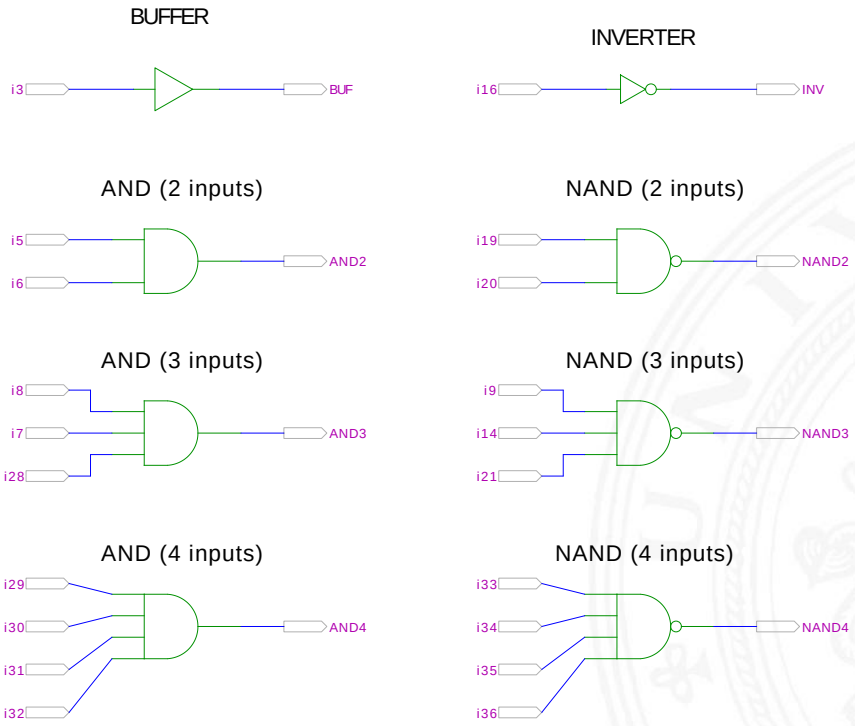
Gatter: Verstärker, Inverter, AND, OR



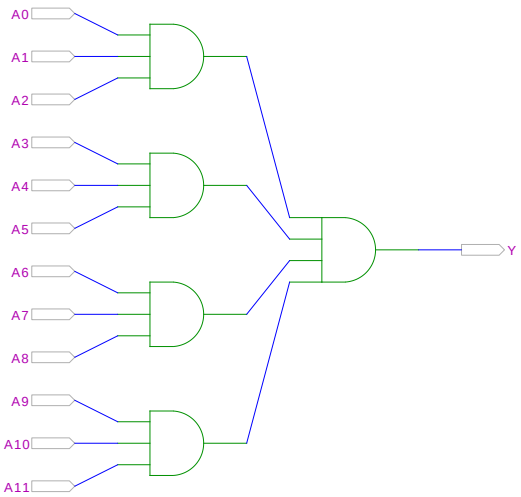
Grundschaltungen: De'Morgan Regel



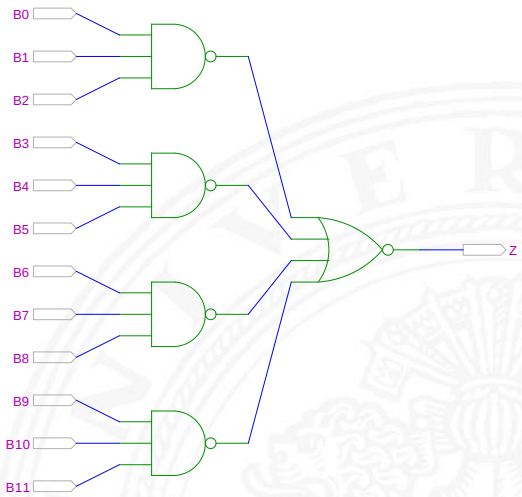
Gatter: AND/NAND mit zwei, drei, vier Eingängen



Gatter: AND mit zwölf Eingängen



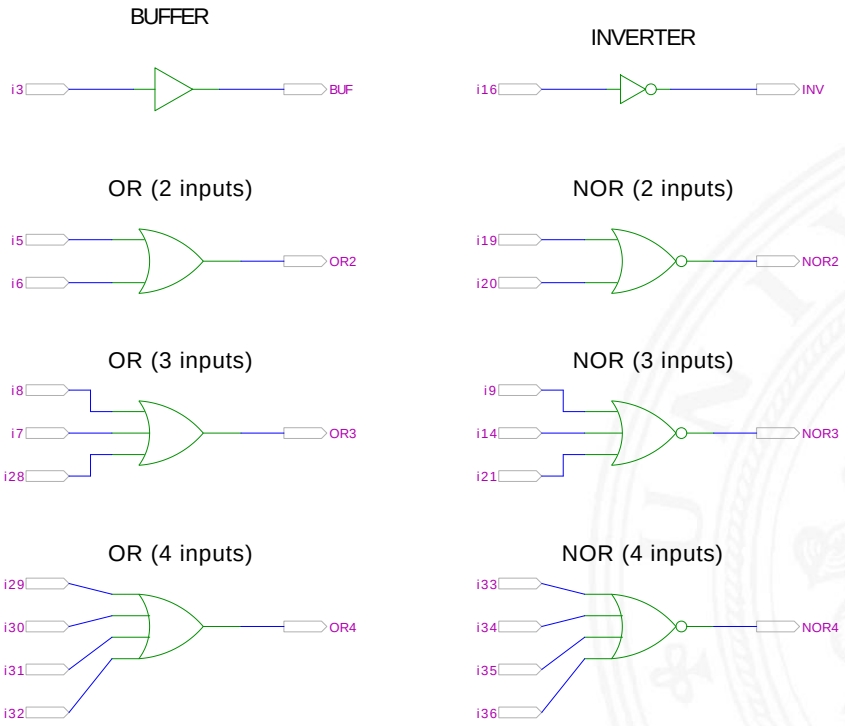
AND3-AND4



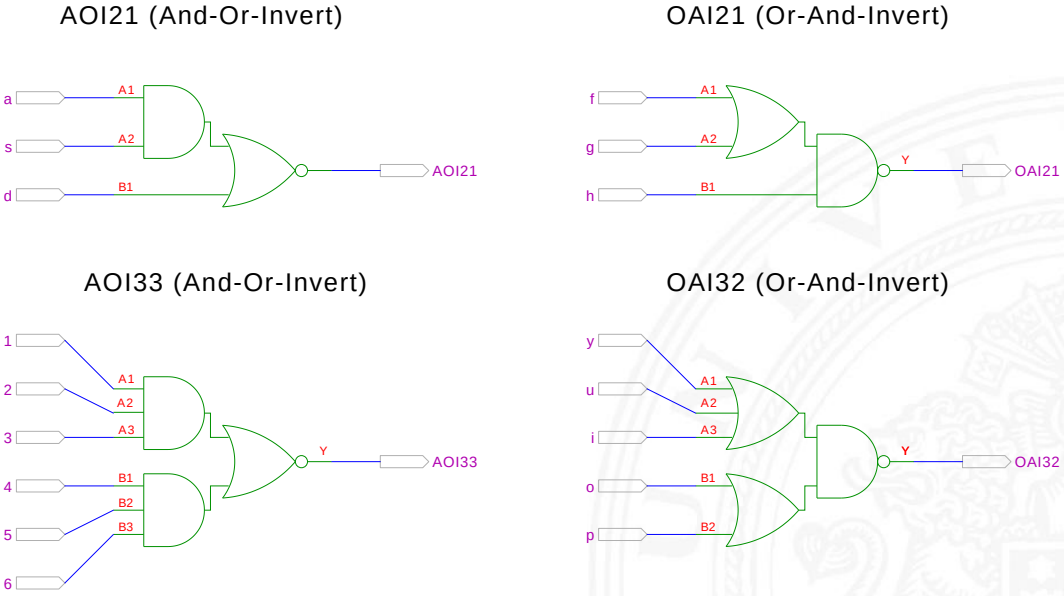
NAND3-NOR4 (de-Morgan)

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

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

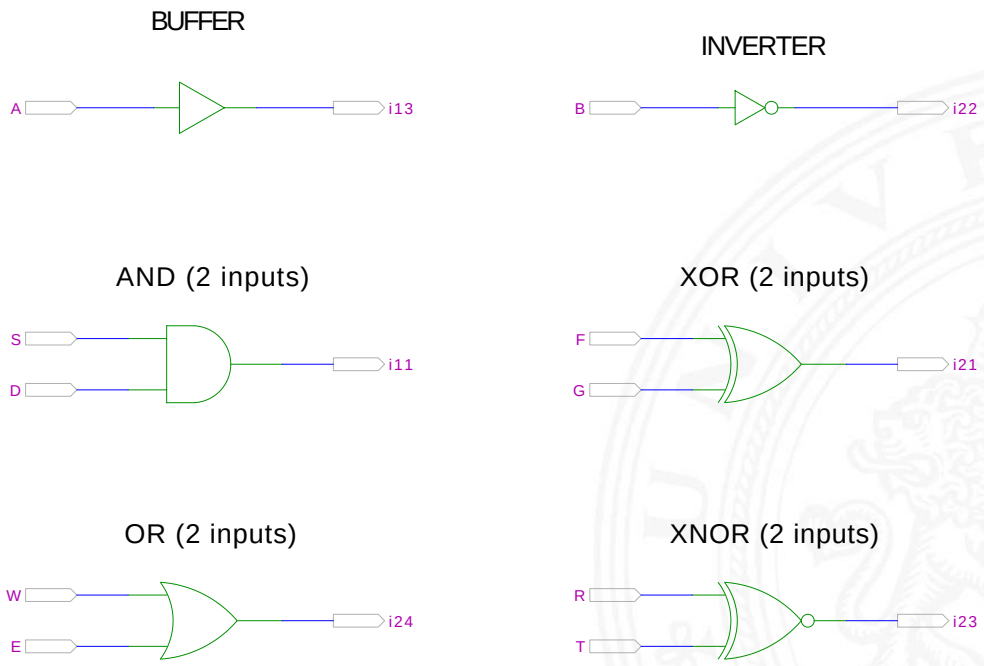


Komplexgatter



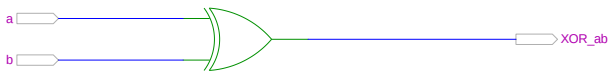
- in CMOS-Technologie besonders günstig realisierbar
  entsprechen vom Aufwand *einem* Gatter

Gatter: XOR und XNOR

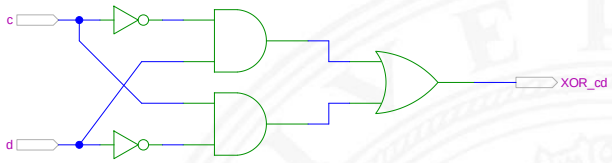


XOR und drei Varianten der Realisierung

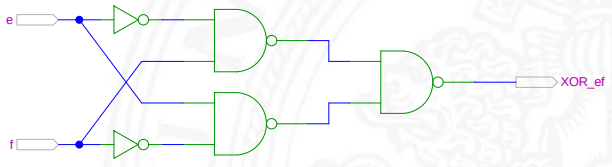
- Symbol



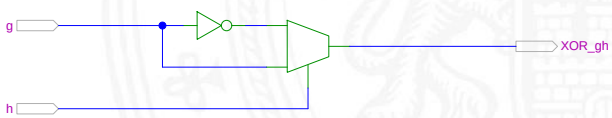
- AND-OR



- NAND-NAND

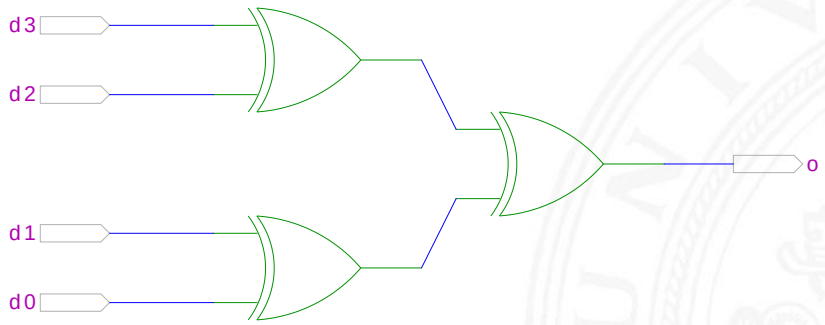


- mit Multiplexer



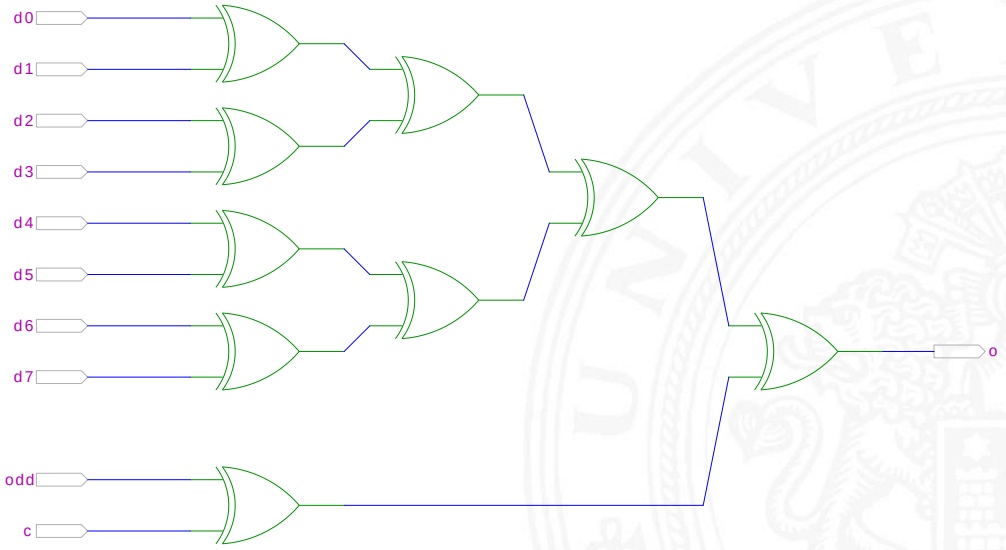
XOR zur Berechnung der Parität

- Parität, siehe „Codierung – Fehlererkennende Codes“
- 4-bit Parität



XOR zur Berechnung der Parität (cont.)

- 8-bit, bzw. 10-bit: Umschaltung odd/even
  Kaskadierung über c-Eingang



2:1-Multiplexer

Umschalter zwischen zwei Dateneingängen („Wechselschalter“)

- ein Steuereingang: s
  zwei Dateneingänge: a_1 und a_0
  ein Datenausgang: y
- wenn $s = 1$ wird a_1 zum Ausgang y durchgeschaltet
  wenn $s = 0$ wird a_0 –"–

s	a_1	a_0	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

2:1-Multiplexer (cont.)

- kompaktere Darstellung der Funktionstabelle durch Verwendung von $*$ (don't care) Termen

s	a_1	a_0	y
0	*	0	0
0	*	1	1
1	0	*	0
1	1	*	1

s	a_1	a_0	y
0	*	a_0	a_0
1	a_1	*	a_1

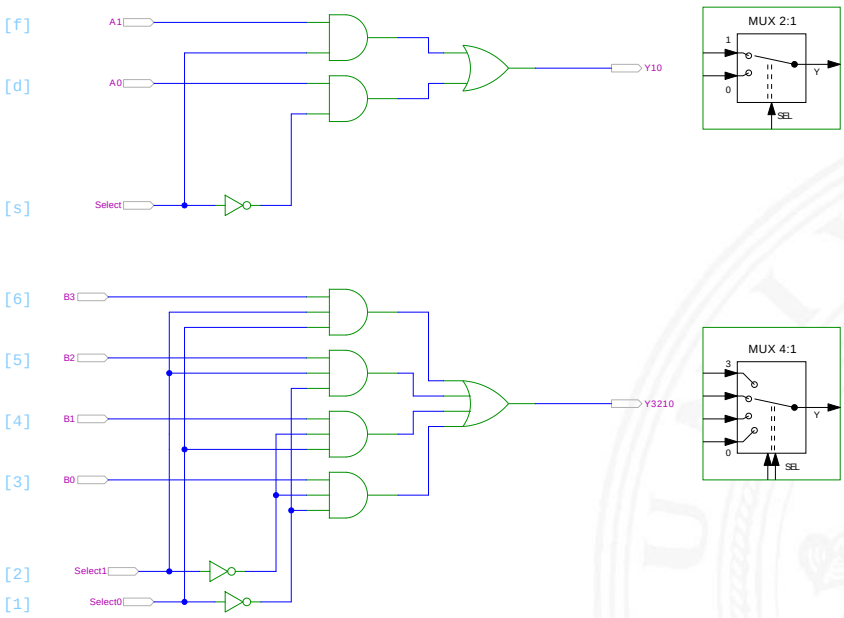
n:1-Multiplexer

Umschalten zwischen mehreren Dateneingängen

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

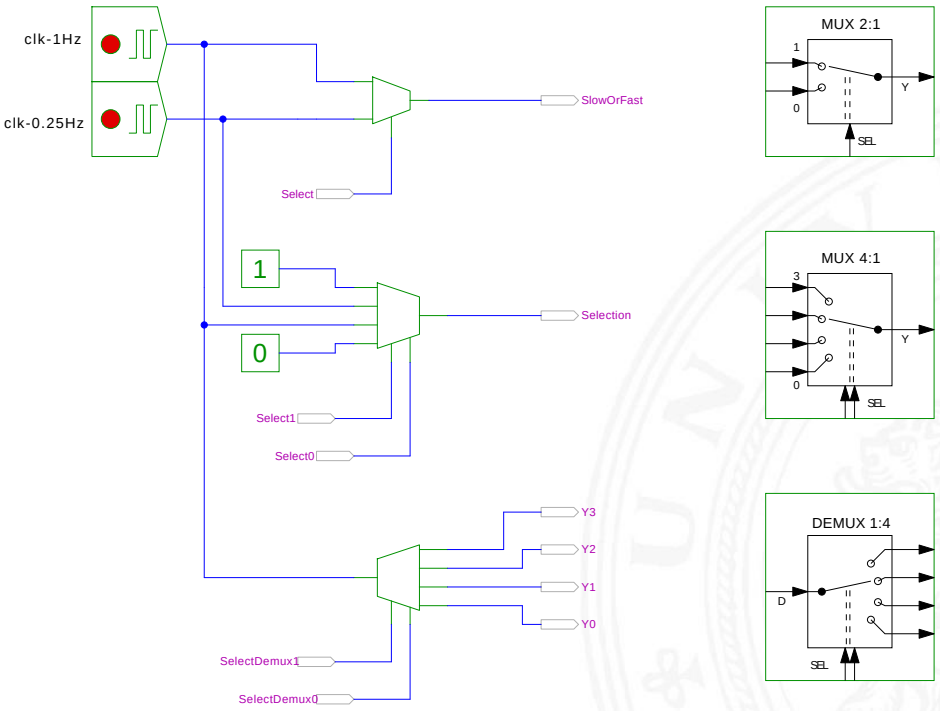
s_1	s_0	a_3	a_2	a_1	a_0	y
0	0	*	*	*	0	0
0	0	*	*	*	1	1
0	1	*	*	0	*	0
0	1	*	*	1	*	1
1	0	*	0	*	*	0
1	0	*	1	*	*	1
1	1	0	*	*	*	0
1	1	1	*	*	*	1

2:1 und 4:1 Multiplexer

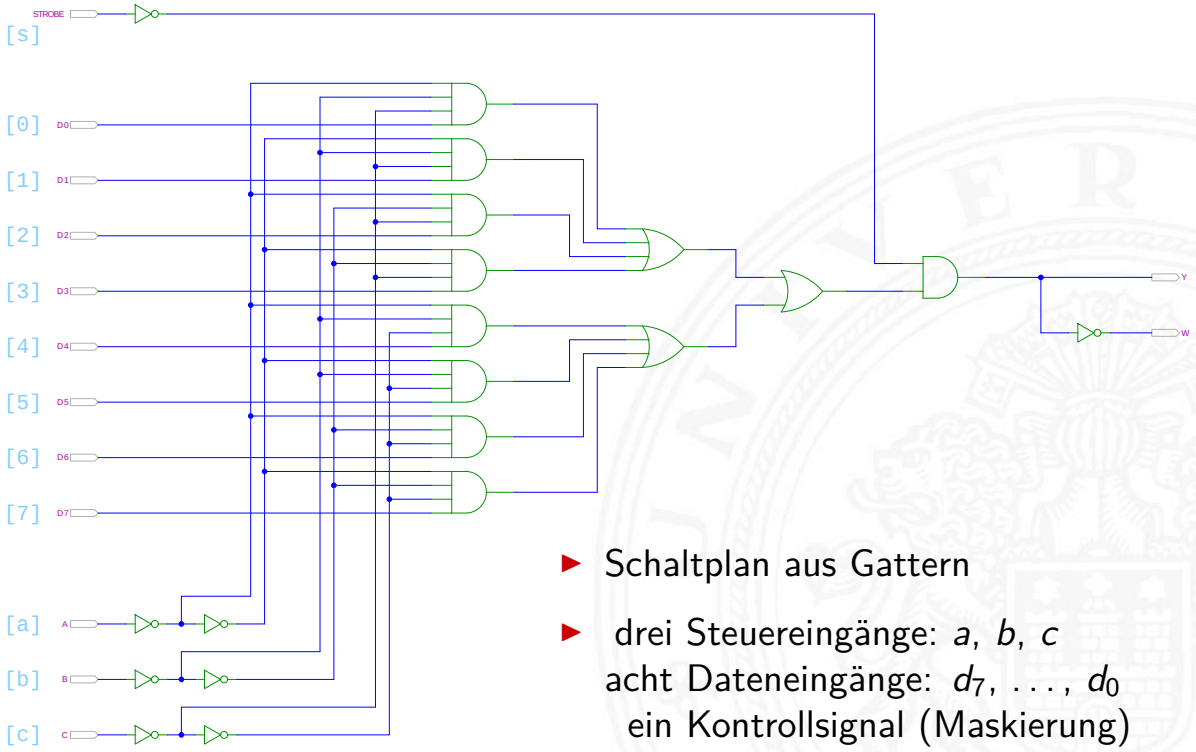


- keine einheitliche Anordnung der Dateneingänge in Schaltplänen: höchstwertiger Eingang manchmal oben, manchmal unten

Multiplexer und Demultiplexer

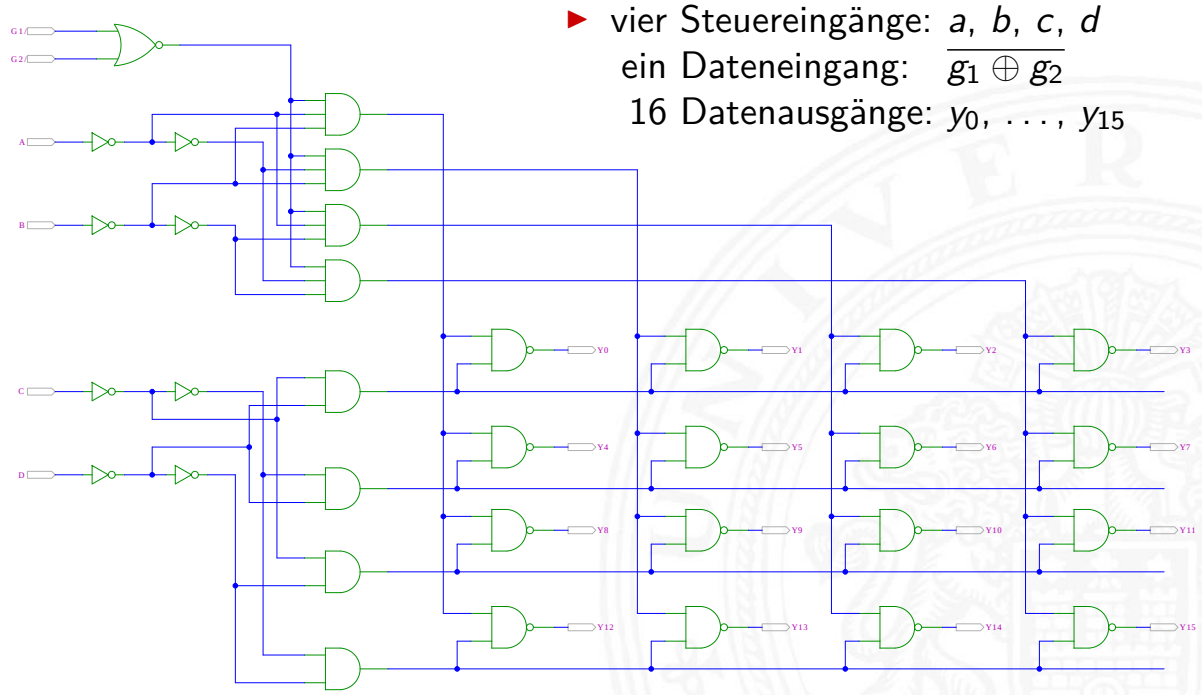


8-bit Multiplexer: Integrierte Schaltung 74151



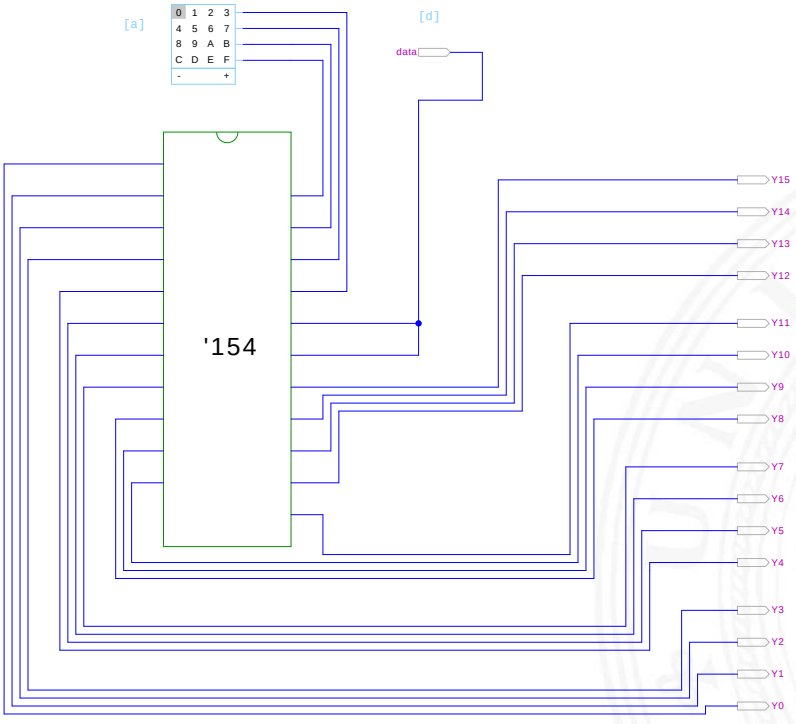
- Schaltplan aus Gattern
- drei Steuereingänge: a, b, c
  acht Dateneingänge: $d_7, \ldots, d_0$
  ein Kontrollsignal (Maskierung)

16-bit Demultiplexer: Integrierte Schaltung 74154



- vier Steuereingänge: a, b, c, d
  ein Dateneingang: $\overline{g_1 \oplus g_2}$
  16 Datenausgänge: $y_0, \ldots, y_{15}$

16-bit Demultiplexer: 74154 als Adressdecoder



Beispiele für Schaltnetze

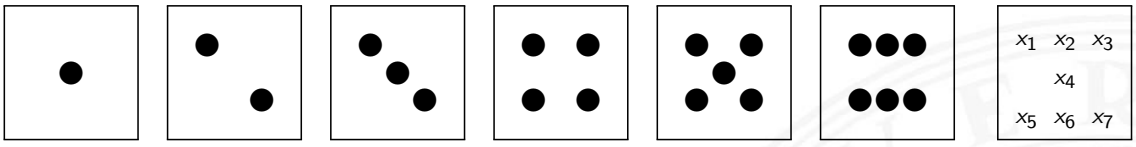
- Schaltungen mit mehreren Ausgängen
- Bündelminimierung der einzelnen Funktionen

ausgewählte typische Beispiele

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

Beispiel: „Würfel“-Decoder

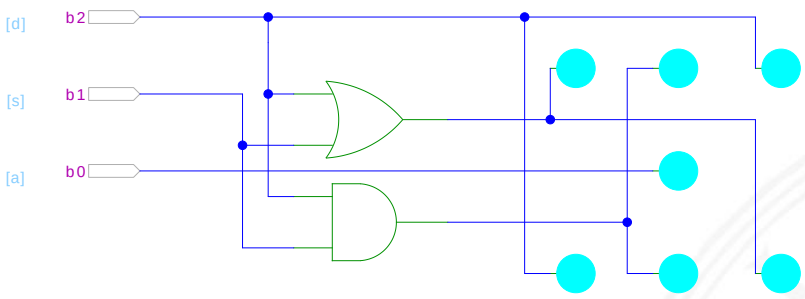
Visualisierung eines Würfels mit sieben LEDs



- Eingabewert von 0...6
- Anzeige als ein bis sechs Augen, bzw. ausgeschaltet

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.)



- Anzeige wie beim Würfel: ein bis sechs Augen
- Minimierung ergibt:

$x_1 = x_7 = b_2 \vee b_1$ links oben, rechts unten

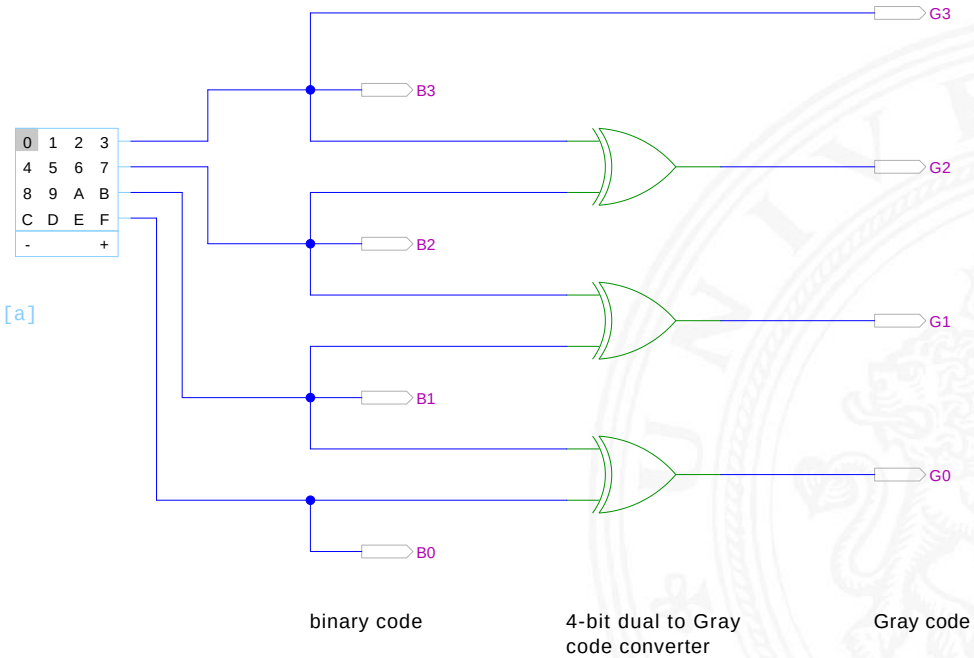
$x_2 = x_6 = b_0 \wedge b_1$ mitte oben, mitte unten

$x_3 = x_5 = b_2$ rechts oben, links unten

$x_4 = b_0$ Zentrum

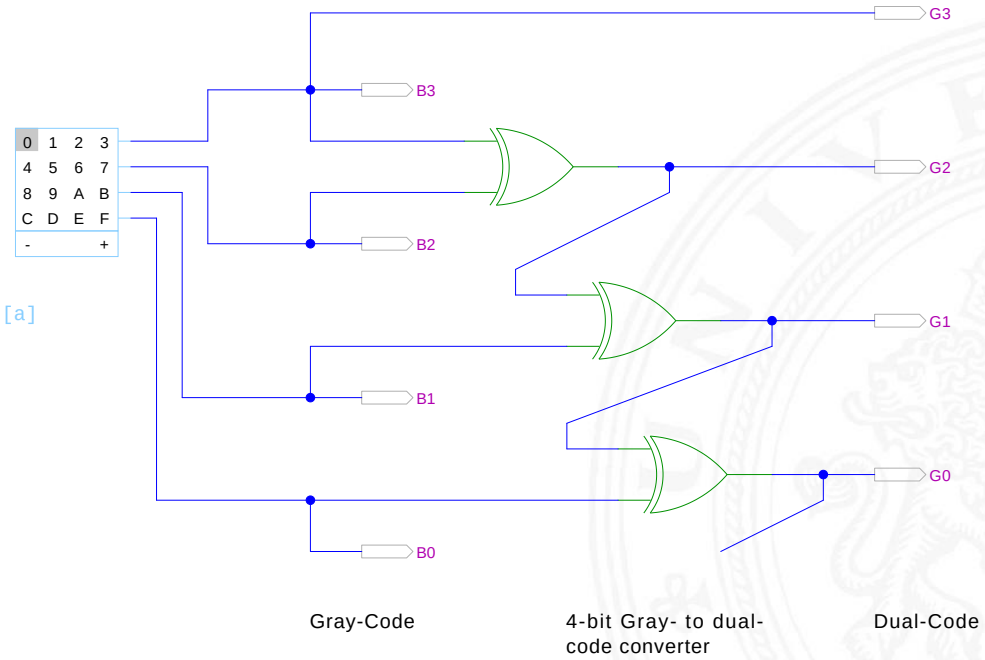
Beispiel: Umwandlung vom Dualcode in den Graycode

XOR benachbarter Bits



Beispiel: Umwandlung vom Graycode in den Dualcode

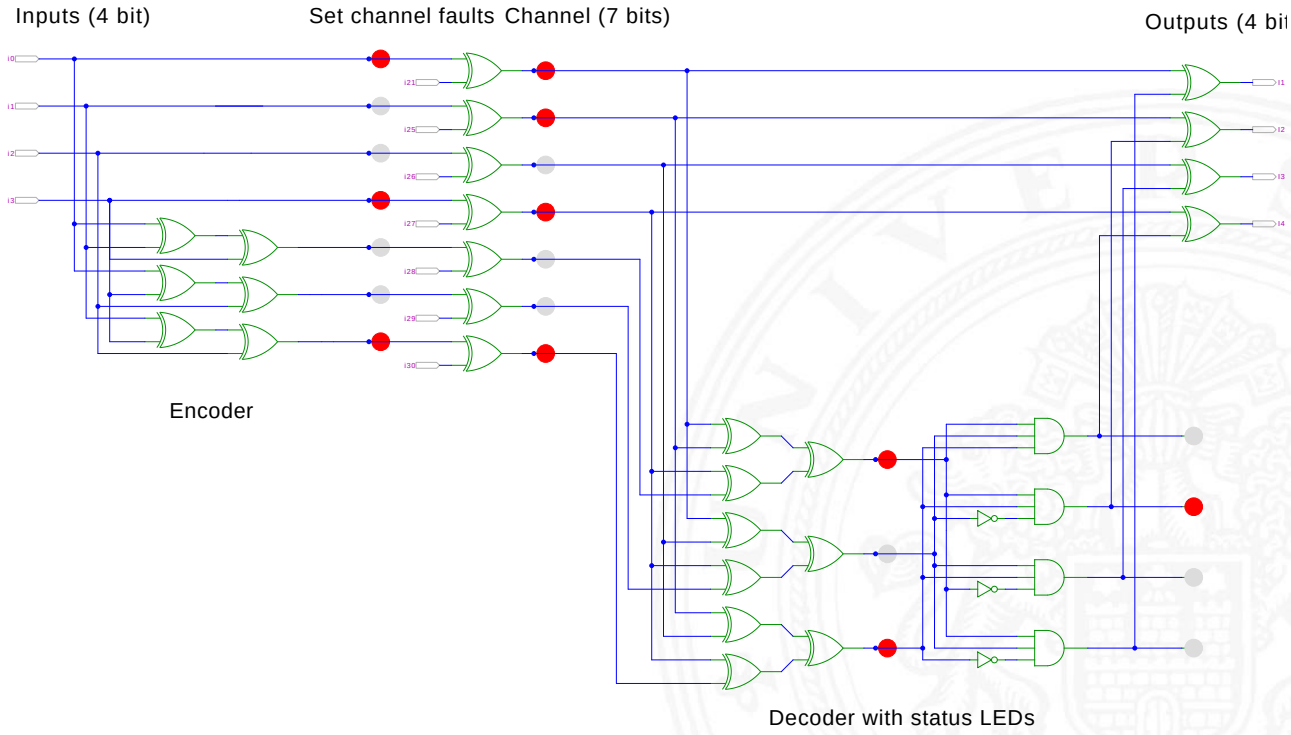
XOR-Kette



(7,4)-Hamming-Code: Encoder und Decoder

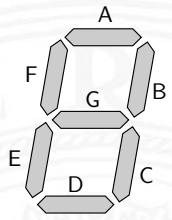
- Encoder — linke Seite
  - vier Eingabebits
  - Hamming-Encoder erzeugt drei Paritätsbits
- Übertragungskanal — Mitte
  - Übertragung von sieben Codebits
  - Einfügen von Übertragungsfehlern durch Invertieren von Codebits mit XOR-Gattern
- Dedoder und Fehlerkorrektur — rechte Seite
  - Decoder liest die empfangenen sieben Bits
  - Syndrom-Berechnung mit XOR-Gattern und Anzeige erkannter Fehler
  - Korrektur gekippter Bits — rechts oben

(7,4)-Hamming-Code: Encoder und Decoder (cont.)



Siebensegmentanzeige

- sieben einzelne Leuchtsegmente (z.B. Leuchtdioden)
- Anzeige stilisierter Ziffern von `0` bis 9
- auch für Hex-Ziffern: `A`, `b`, `C`, `d` `E`, `F`



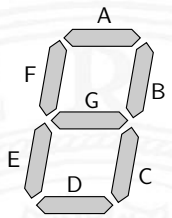
- sieben Schaltfunktionen, je eine pro Ausgang
- Umcodierung von 4-bit Dualwerten in geeignete Ausgangswerte
- Segmente im Uhrzeigersinn: A (oben) bis F, G innen

- eingeschränkt auch als alphanumerische Anzeige für Ziffern und (einige) Buchstaben
  - gemischt Groß- und Kleinbuchstaben
  - Probleme mit `M`, `N`, usw.

Siebensegmentanzeige: Funktionen

- Funktionen für Hex-Anzeige, 0...F

	0	1	2	3	4	5	6	8	9	A	b	C	d	E	F
A =	1	0	1	1	0	1	1	1	1	1	1	0	0	0	1	1
B =	1	1	1	1	1	0	0	1	1	1	1	0	0	1	0	0
C =	1	1	0	1	1	1	1	1	1	1	1	1	0	1	0	0
D =	1	0	1	1	0	1	1	0	1	1	0	1	1	1	1	0
E =	1	0	1	0	0	0	1	0	1	0	1	1	1	1	1	1
F =	1	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1
G =	0	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1



- für Ziffernanzeige mit *Don't Care*-Termen

  A = 1011011111 ∗ ∗ ∗ ∗ ∗ ∗
  B = usw.

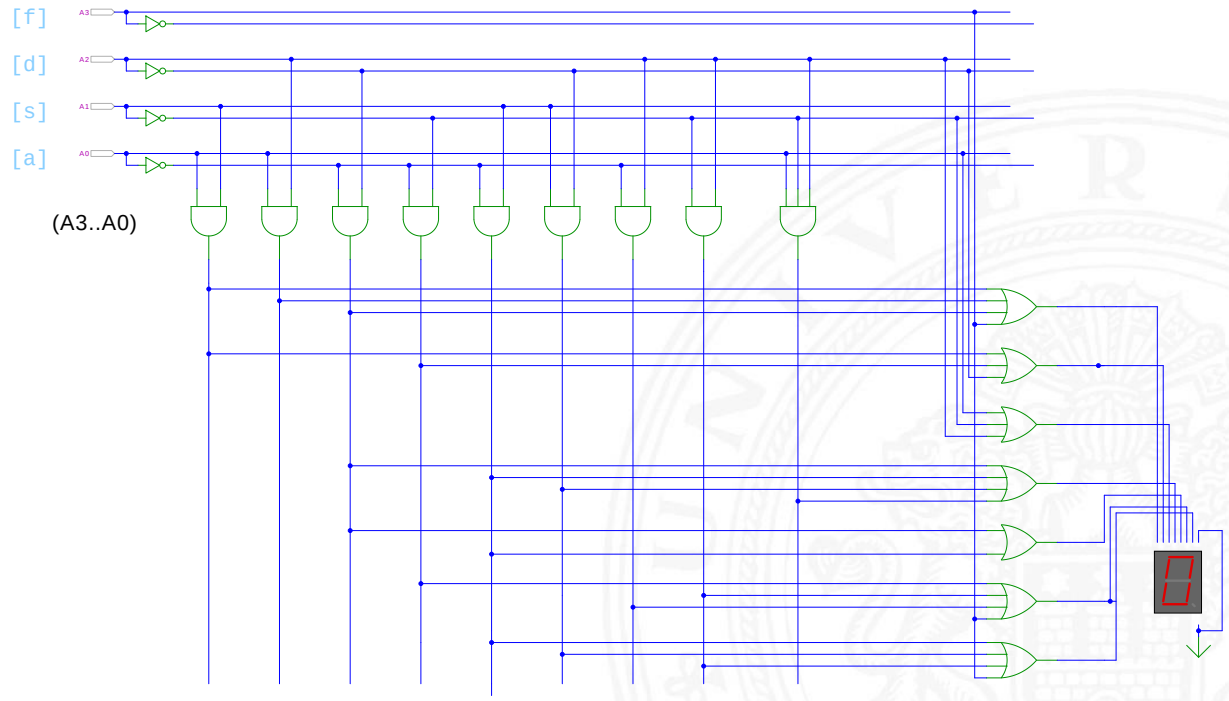
Siebensegmentanzeige: Bündelminimierung

- zum Beispiel mit sieben KV-Diagrammen...
- dabei versuchen, gemeinsame Terme zu finden und zu nutzen

  Minimierung als Übungsaufgabe?
- nächste Folie zeigt Lösung aus Schiffmann, Schmitz

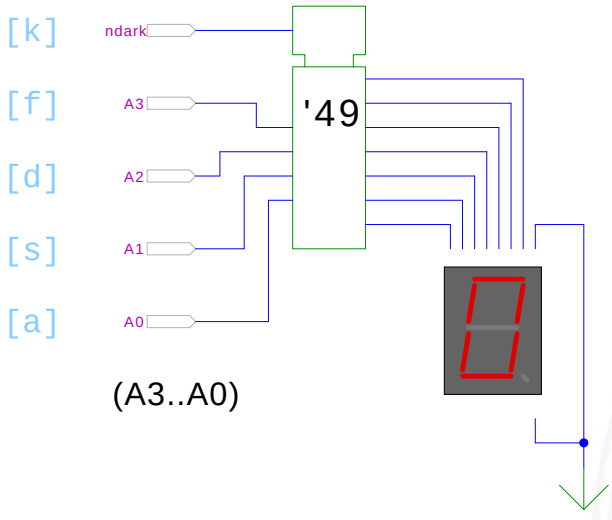
- als mehrstufige Schaltung ist günstigere Lösung möglich
  siehe Knuth: *AoCP, Volume 4, Fascicle 0*, 7.1.2 (Seite 112ff)

Siebensegmentdecoder: Ziffern 0..9



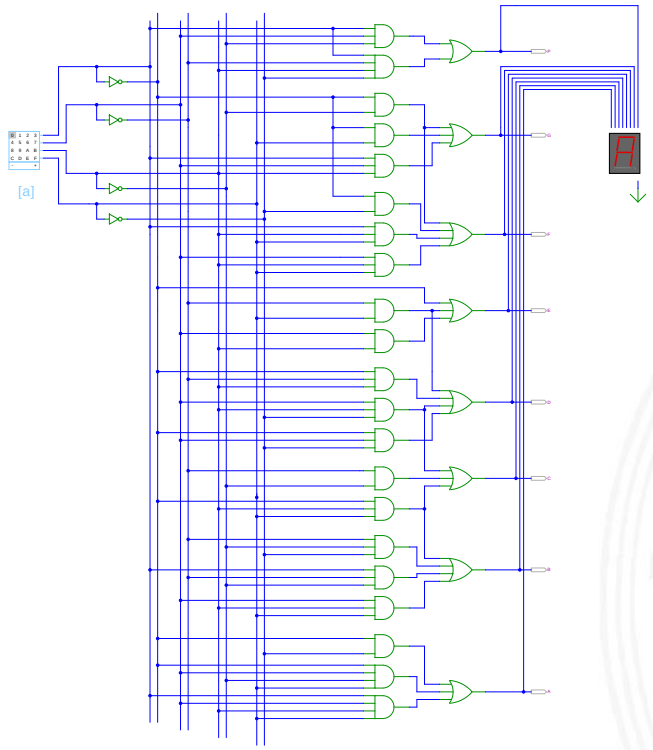
Schiffmann, Schmitz, *Technische Informatik I*

Siebensegmentdecoder: Integrierte Schaltung 7449



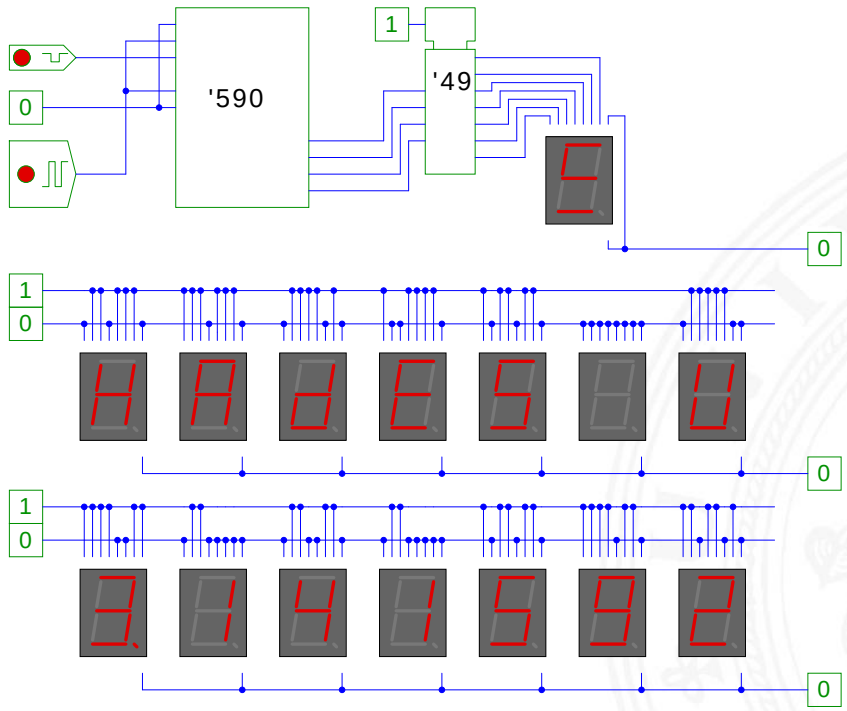
- Beispiel für eine integrierte Schaltung (IC)
- Anzeige von 0..9, Zufallsmuster für A..F, „Dunkeltastung“

Siebensegmentanzeige: Hades-Beispiele



- Buchstaben A. . . P

Siebensegmentanzeige: Hades-Beispiele (cont.)



Siebensegmentanzeige: mehrstufige Realisierung

Minimale Anzahl der Gatter für die Schaltung?

- Problem vermutlich nicht optimal lösbar (nicht *tractable*)
- Heuristik basierend auf „häufig“ verwendeten Teilfunktionen
- Eingänge x_1, x_2, x_3, x_4, Ausgänge $a, \ldots, g$

$$
\begin{array}{lll}
x_5 = x_2 \oplus x_3 & x_{13} = x_1 \oplus x_7 & \overline{a} = x_{20} = x_{14} \wedge \overline{x_{19}} \\
x_6 = \overline{x_1} \wedge x_4 & x_{14} = x_5 \oplus x_6 & \overline{b} = x_{21} = x_7 \oplus x_{12} \\
x_7 = x_3 \wedge \overline{x_6} & x_{15} = x_7 \vee x_{12} & \overline{c} = x_{22} = \overline{x_8} \wedge x_{15} \\
x_8 = x_1 \oplus x_2 & x_{16} = x_1 \vee x_5 & \overline{d} = x_{23} = x_9 \wedge \overline{x_{13}} \\
x_9 = x_4 \oplus x_5 & x_{17} = x_5 \vee x_6 & \overline{e} = x_{24} = x_6 \vee x_{18} \\
x_{10} = \overline{x_7} \wedge x_8 & x_{18} = x_9 \wedge x_{10} & \overline{f} = x_{25} = \overline{x_8} \wedge x_{17} \\
x_{11} = x_9 \oplus x_{10} & x_{19} = x_3 \wedge x_9 & g = x_{26} = x_7 \vee x_{16} \\
x_{12} = x_5 \wedge x_{11} & &
\end{array}
$$

Knuth, *AoCP, Volume 4, Fascicle 0*, Kap 7.1.2, Seite 113

Logische und arithmetische Operationen

- Halb- und Volladdierer
- Addierertypen
  - Ripple-Carry
  - Carry-Lookahead
- Multiplizierer
- Quadratwurzel
- Barrel-Shifter
- ALU

Halbaddierer

- **Halbaddierer**: berechnet 1-bit Summe s und Übertrag c_o (*carry-out*) von zwei Eingangsbits a und b

a	b	c_o	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$c_o = a \wedge b$

$s = a \oplus b$

Volladdierer

- **Volladdierer**: berechnet 1-bit Summe s und Übertrag c_o (*carry-out*) von zwei Eingangsbits a, b sowie Eingangsübertrag c_i (*carry-in*)

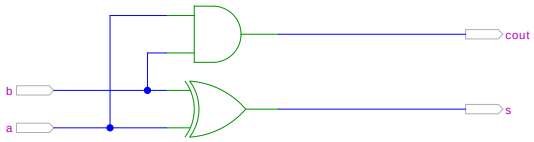
a	b	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$c_o = ab \vee ac_i \vee bc_i = (ab) \vee (a \vee b)c_i$

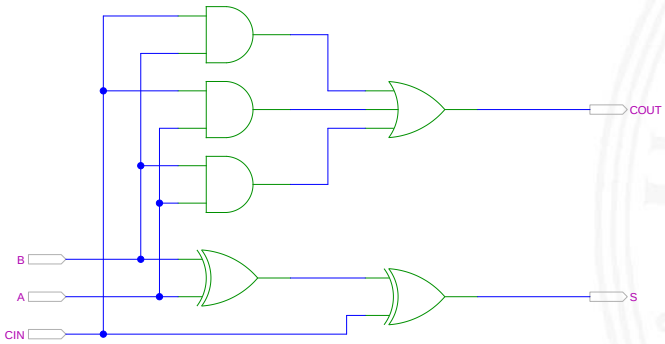
$s = a \oplus b \oplus c_i$

Schaltbilder für Halb- und Volladdierer

1-bit half-adder: (COUT,S) = (A+B)



1-bit full-adder: (COUT,S) = (A+B+Cin)



n-bit Addierer

- Summe: $s_n = a_n \oplus b_n \oplus c_n$

$$
\begin{aligned}
s_0 &= a_0 \oplus b_0 \\
s_1 &= a_1 \oplus b_1 \oplus c_1 \\
s_2 &= a_2 \oplus b_2 \oplus c_2 \\
&\ldots \\
s_n &= a_n \oplus b_n \oplus c_n
\end{aligned}
$$

- Übertrag: $c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$

$$
\begin{aligned}
c_1 &= (a_0 b_0) \\
c_2 &= (a_1 b_1) \vee (a_1 \vee b_1) c_1 \\
c_3 &= (a_2 b_2) \vee (a_2 \vee b_2) c_2 \\
&\ldots \\
c_{n+1} &= (a_n b_n) \vee (a_n \vee b_n) c_n
\end{aligned}
$$

n-bit Addierer (cont.)

- n-bit Addierer theoretisch als zweistufige Schaltung realisierbar
- direkte und negierte Eingänge, dann AND-OR Netzwerk
- Aufwand steigt exponentiell mit n an,
  für Ausgang n sind $2^{(2n-1)}$ Minterme erforderlich

$\Rightarrow$ nicht praktikabel

- Problem: Übertrag (*carry*)
  $c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$
  rekursiv definiert

n-bit Addierer (cont.)

Diverse gängige Alternativen für Addierer

- Ripple-Carry
  - lineare Struktur
  - \+ klein, einfach zu implementieren
  - – langsam, Laufzeit $O(n)$
- Carry-Lookahead (CLA)
  - Baumstruktur
  - \+ schnell
  - – teuer (Flächenbedarf der Hardware)
- Mischformen: Ripple-block CLA, Block CLA, Parallel Prefix
- andere Ideen: Carry Select, Conditional Sum, Carry Skip

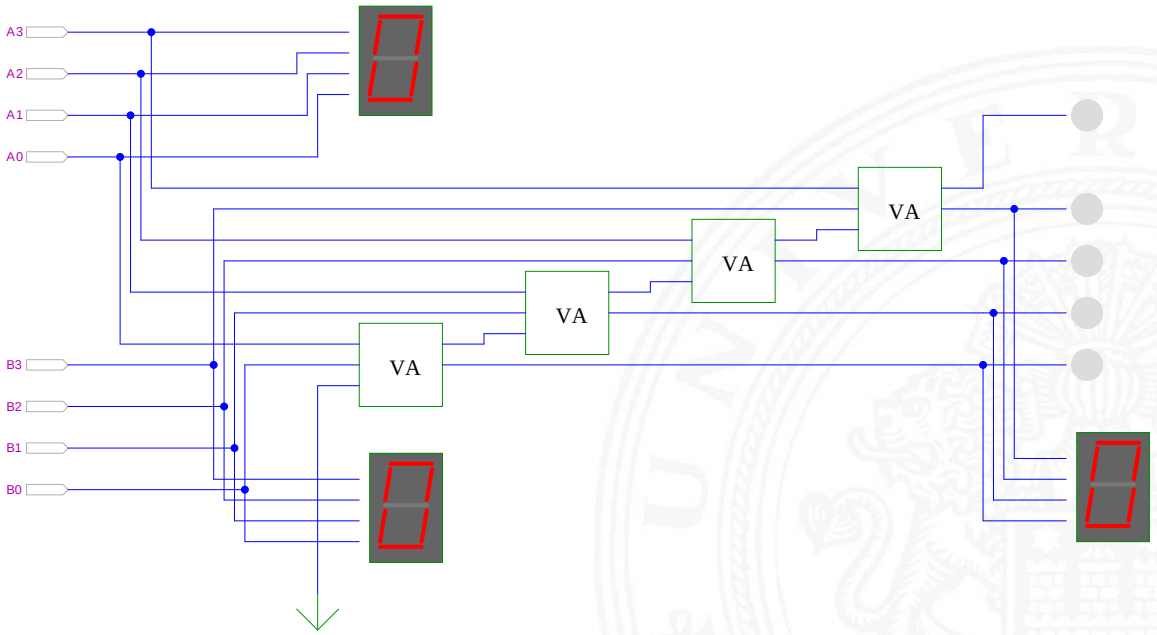
...

Ripple-Carry Adder

- Kaskade aus n einzelnen Volladdierern
- Carry-out von Stufe i treibt Carry-in von Stufe $i+1$
- Gesamtverzögerung wächst mit der Anzahl der Stufen als $O(n)$

- Addierer in Prozessoren häufig im *kritischen Pfad*
- möglichst hohe Performance ist essentiell
- ripple-carry in CMOS-Technologie bis ca. 10-bit geeignet
- bei größerer Wortbreite gibt es effizientere Schaltungen

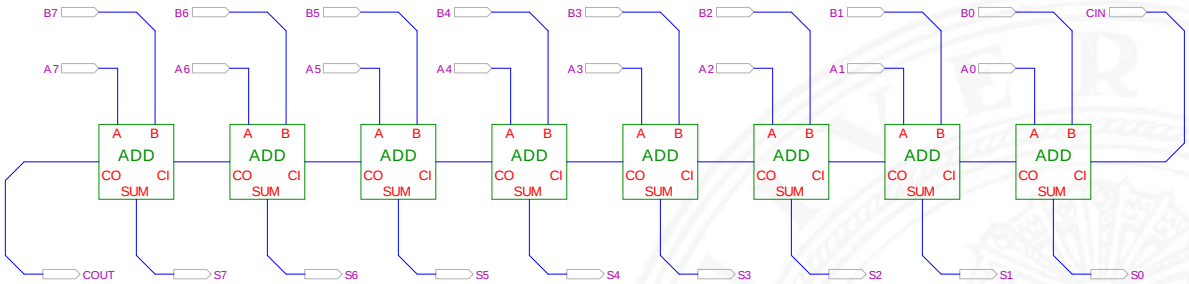
Ripple-Carry Adder: 4-bit



Schiffmann, Schmitz, *Technische Informatik I*

Ripple-Carry Adder: Hades-Beispiel mit Verzögerungen

- Kaskade aus acht einzelnen Volladdierern



- Gatterlaufzeiten in der Simulation bewusst groß gewählt
- Ablauf der Berechnung kann interaktiv beobachtet werden
- alle Addierer arbeiten parallel
- aber Summe erst fertig, wenn alle Stufen durchlaufen sind

Subtrahierer

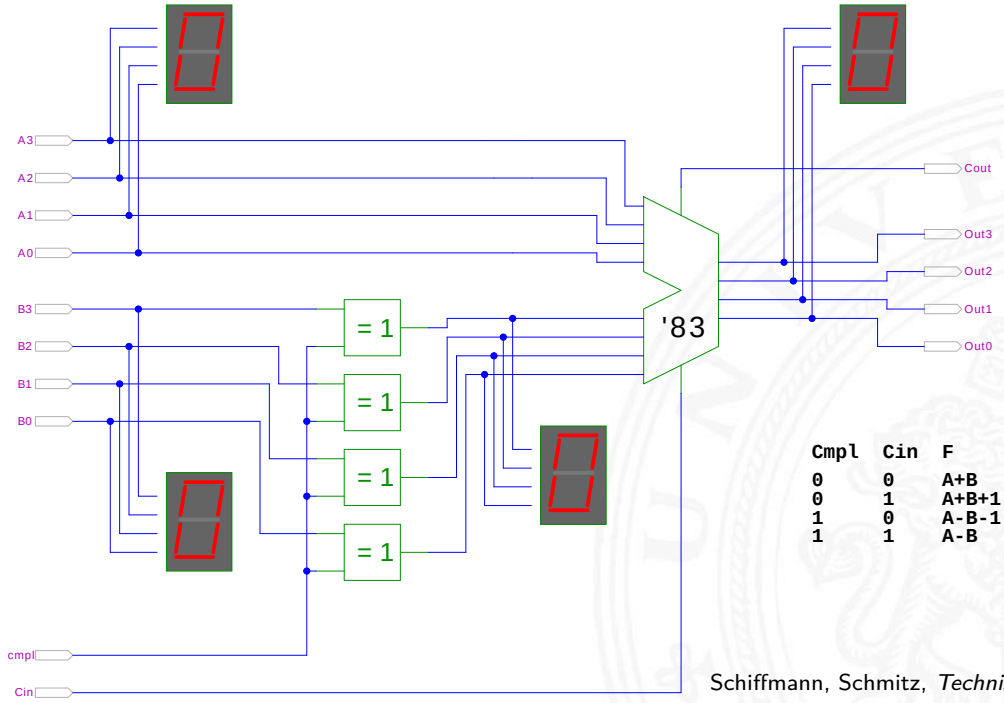
Zweierkomplement

- $(A - B)$ ersetzt durch Addition des 2-Komplements von B
- 2-Komplement: Invertieren aller Bits und Addition von Eins
- Carry-in Eingang des Addierers bisher nicht benutzt

Subtraktion quasi „gratis“ realisierbar

- normalen Addierer verwenden
- Invertieren der Bits von B (1-Komplement)
- Carry-in Eingang auf 1 setzen (Addition von 1)
- Resultat ist $A + (\neg B) + 1 = A - B$

Subtrahierer: Beispiel (7483 – 4-bit Addierer)



Cmpl	Cin	F	
0	0	A+B	
0	1	A+B+1	(A++)
1	0	A-B-1	(A--)
1	1	A-B	

Schiffmann, Schmitz, *Technische Informatik I*

Schnelle Addierer

- Addierer in Prozessoren häufig im *kritischen Pfad*
- möglichst hohe Performance ist essentiell

$\Rightarrow$ bestimmt Taktfrequenz

- Carry-Select Adder: Gruppen von Ripple-carry
- Carry-Lookahead Adder: Baumstruktur zur Carry-Berechnung
- ...

- über 10 Addierer „Typen“ (für 2 Operanden)
- Addition mehrerer Operanden
- Typen teilweise technologieabhängig

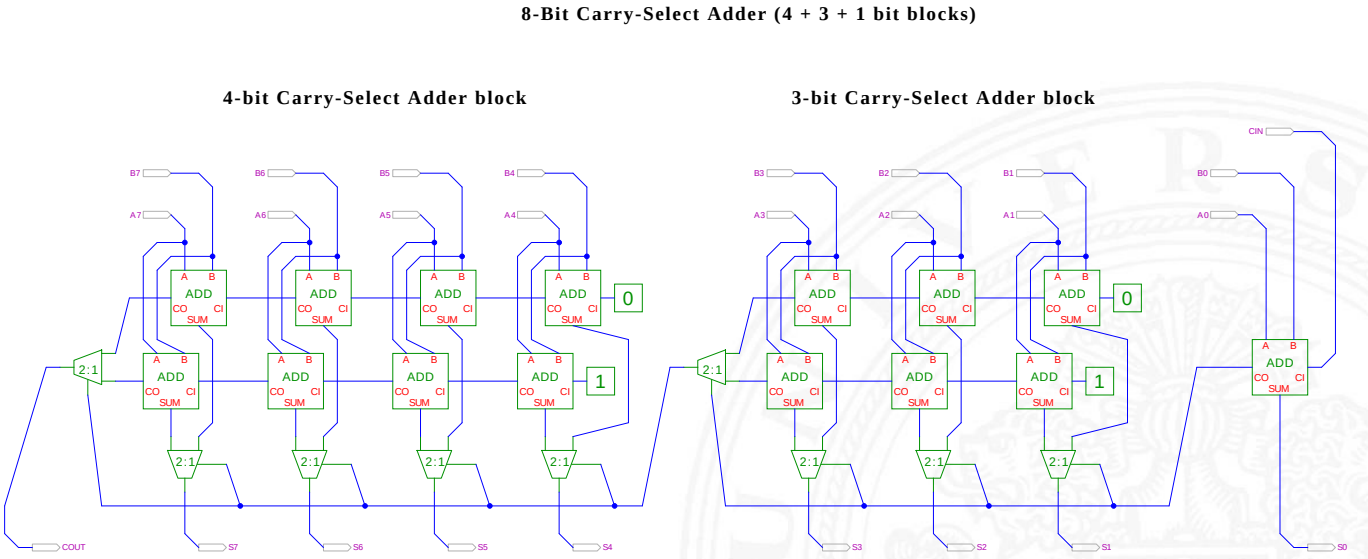
Carry-Select Adder: Prinzip

- Aufteilen des n-bit Addierers in mehrere Gruppen mit je m_i-bits
- für jede Gruppe
  - jeweils zwei m_i-bit Addierer
  - einer rechnet mit $c_i = 0$ $(a+b)$, der andere mit $c_i = 1$ $(a+b+1)$
  - 2:1-Multiplexer mit m_i-bit wählt die korrekte Summe aus

- Sobald der Wert von c_i bekannt ist (Ripple-Carry), wird über den Multiplexer die benötigte Zwischensumme ausgewählt
- Das berechnete Carry-out c_o der Gruppe ist das Carry-in c_i der folgenden Gruppe

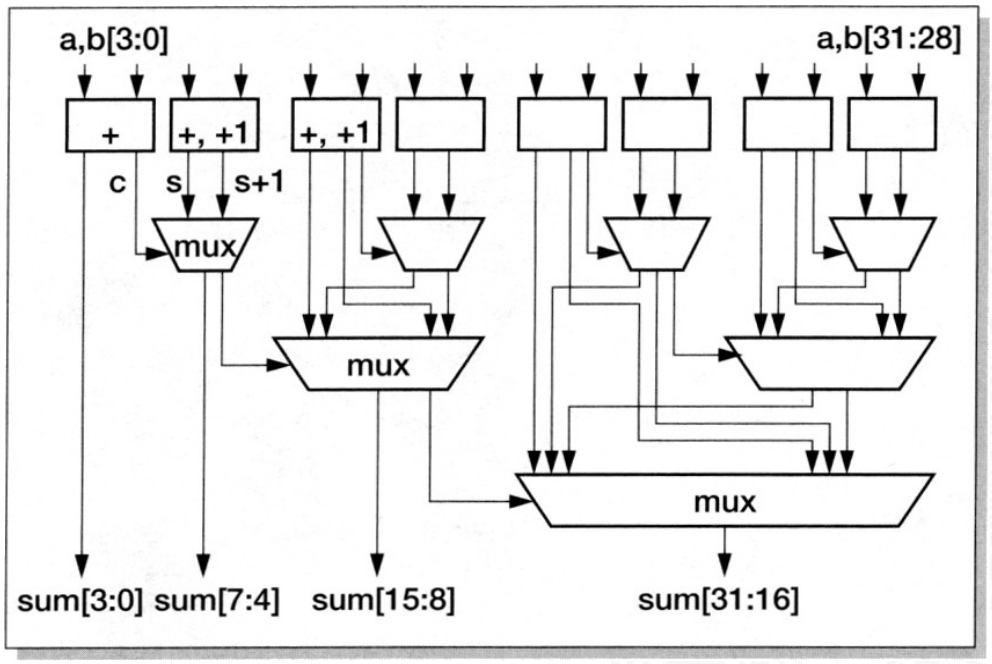
$\Rightarrow$ Verzögerung reduziert sich auf die Verzögerung eines m-bit Addierers plus die Verzögerungen der Multiplexer

Carry-Select Adder: Beispiel



- drei Gruppen: 1-bit, 3-bit, 4-bit
- Gruppengrößen so wählen, dass Gesamtverzögerung minimal

Carry-Select Adder: Beispiel ARM v6



S. Furber, *ARM System-on-Chip Architecture*, 2000

Carry-Lookahead Adder: Prinzip

- $c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$
- Einführung von Hilfsfunktionen

  $g_n = (a_n b_n)$ „generate carry“

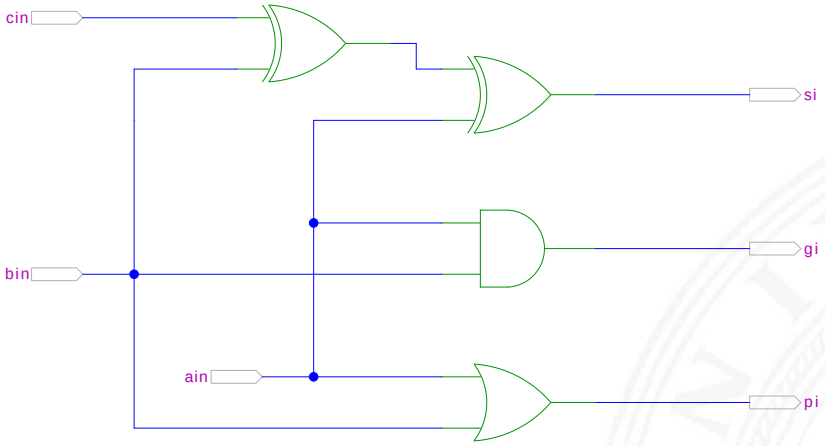
  $p_n = (a_n \vee b_n)$ „propagate carry“

  $c_{n+1} = g_n \vee p_n c_n$
- *generate*: Carry out erzeugen, unabhängig von Carry-in

  propagate: Carry out weiterleiten / Carry-in maskieren
- Berechnung der g_n und p_n in einer Baumstruktur

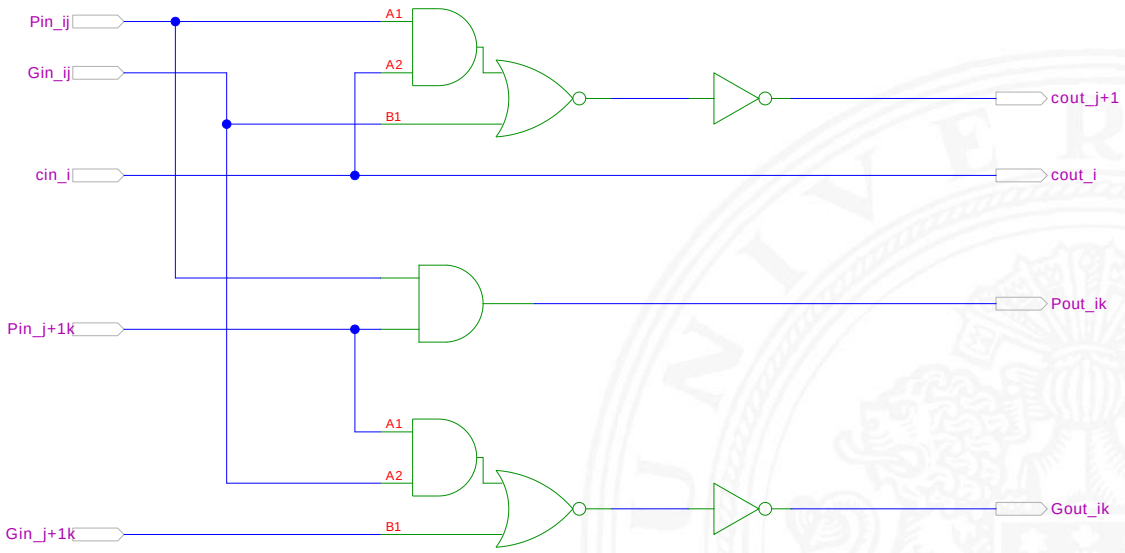
  Tiefe des Baums ist $\log_2 N \Rightarrow$ entsprechend schnell

Carry-Lookahead Adder: SUM-Funktionsblock

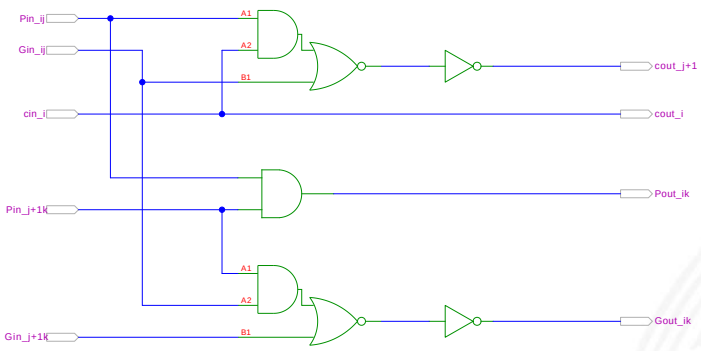


- 1-bit Addierer, $s = a_i \oplus b_i \oplus c_i$
- keine Berechnung des Carry-Out
- Ausgang $g_i = a_i \wedge b_i$ liefert *generate-carry* Signal
- Ausgang $p_i = a_i \vee b_i$ liefert *propagate-carry* Signal

Carry-Lookahead Adder: CLA-Funktionsblock

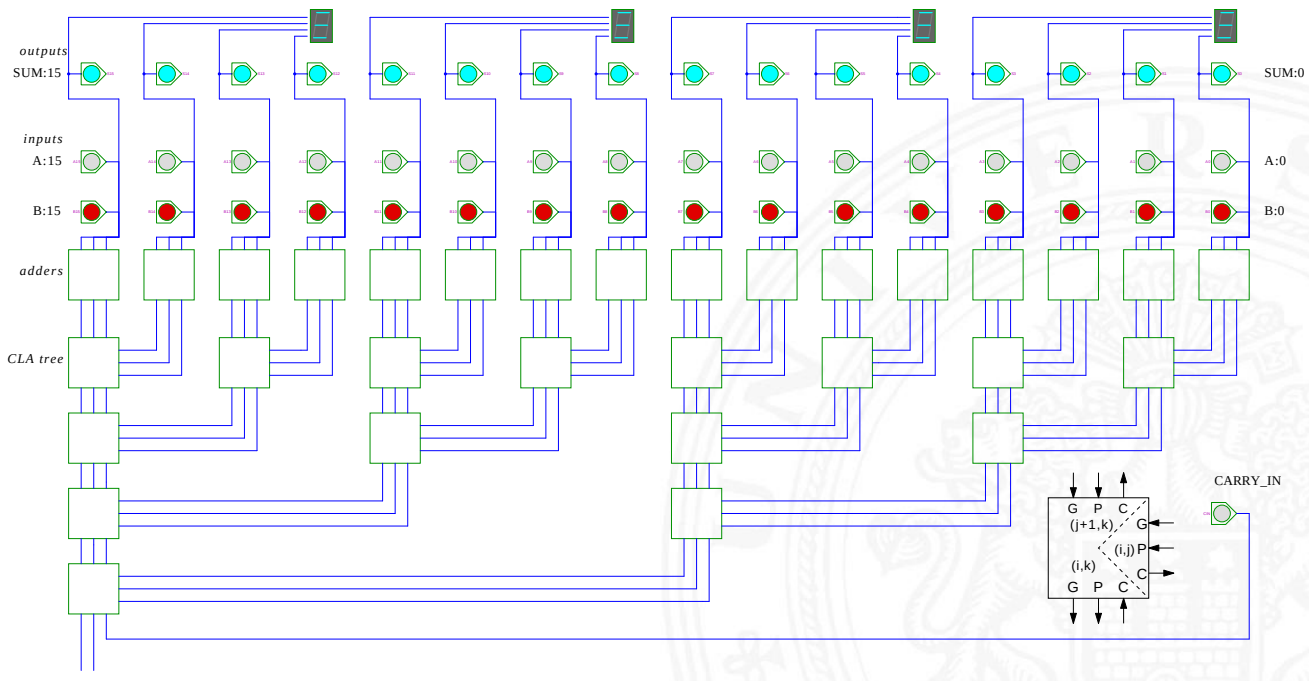


Carry-Lookahead Adder: CLA-Funktionsblock (cont.)

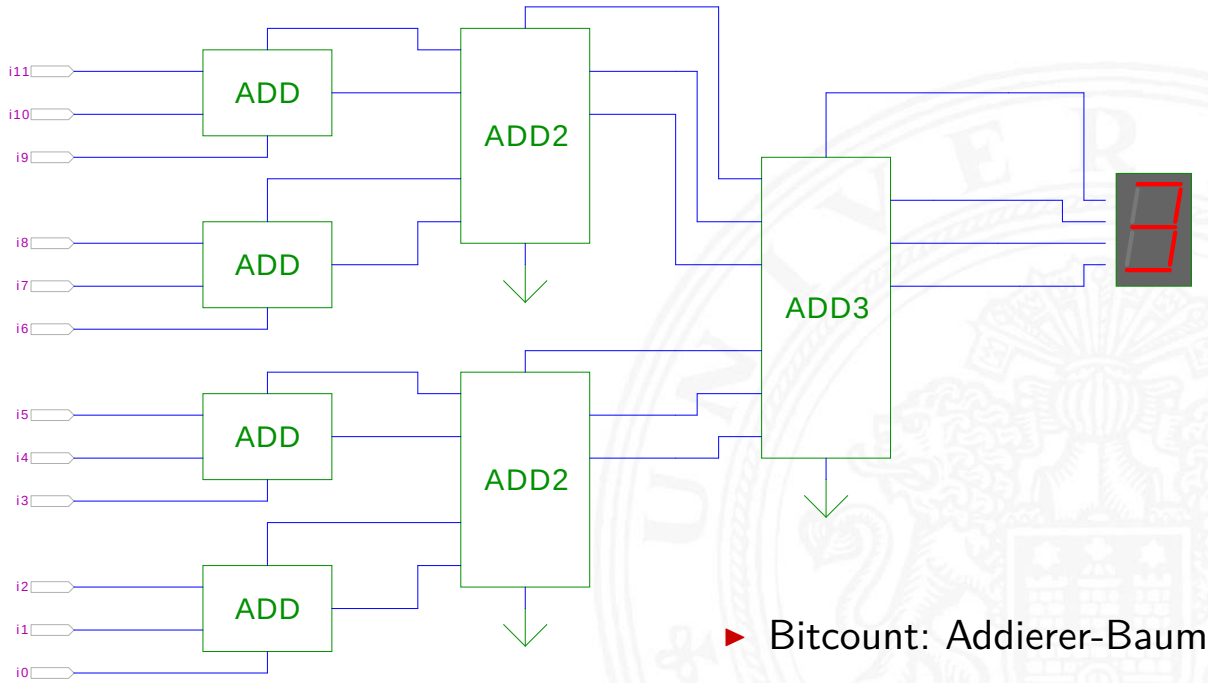


- Eingänge
  - propagate/generate Signale von zwei Stufen
  - carry-in Signal
- Ausgänge
  - propagate/generate Signale zur nächsthöheren Stufe
  - carry-out Signale: Durchleiten und zur nächsthöheren Stufe

Carry-Lookahead Adder: 16-bit Addierer



Addition mehrerer Operanden



- Bitcount: Addierer-Baum

Addierer: Zusammenfassung

- Halbaddierer $(a \oplus b)$
- Volladdierer $(a \oplus b \oplus c_i)$

- Ripple-carry
  - Kaskade aus Volladdierern, einfach und billig
  - aber manchmal zu langsam, Verzögerung: $O(n)$
- Carry-select Prinzip
  - Verzögerung $O(\sqrt{n})$
- Carry-lookahead Prinzip
  - Verzögerung $O(\ln n)$

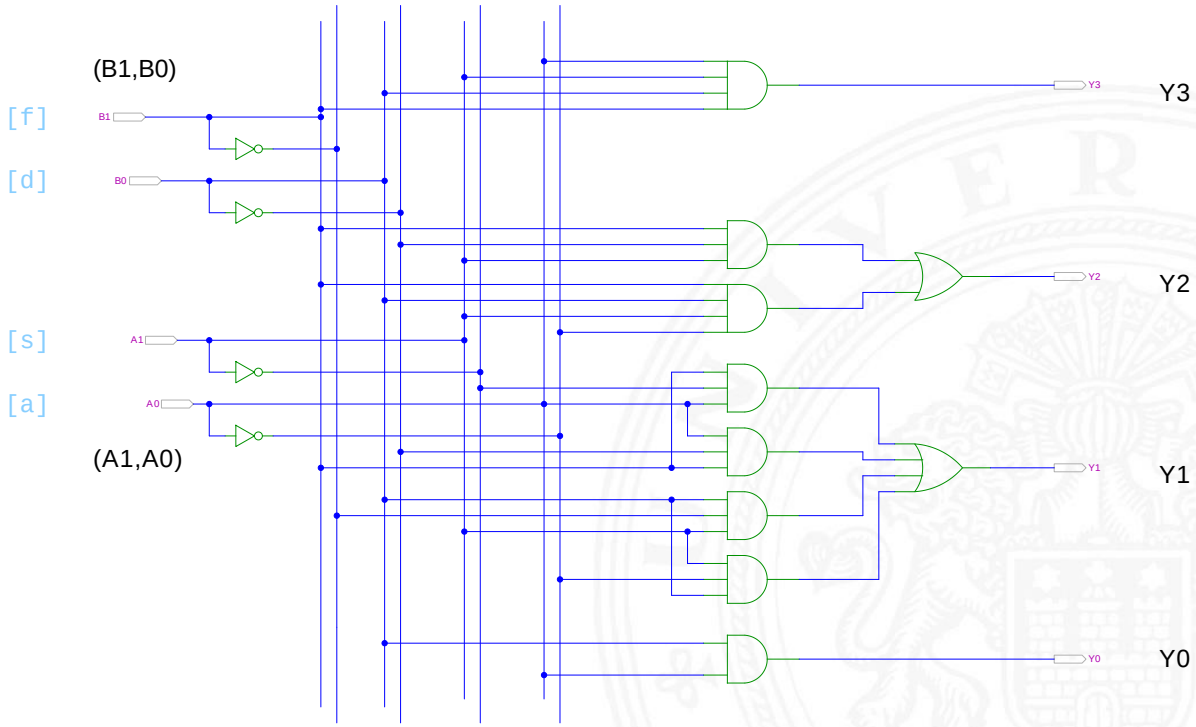
- Subtraktion durch Zweierkomplementbildung
  erlaubt auch Inkrement (`A++`) und Dekrement (`A--`)

Multiplizierer

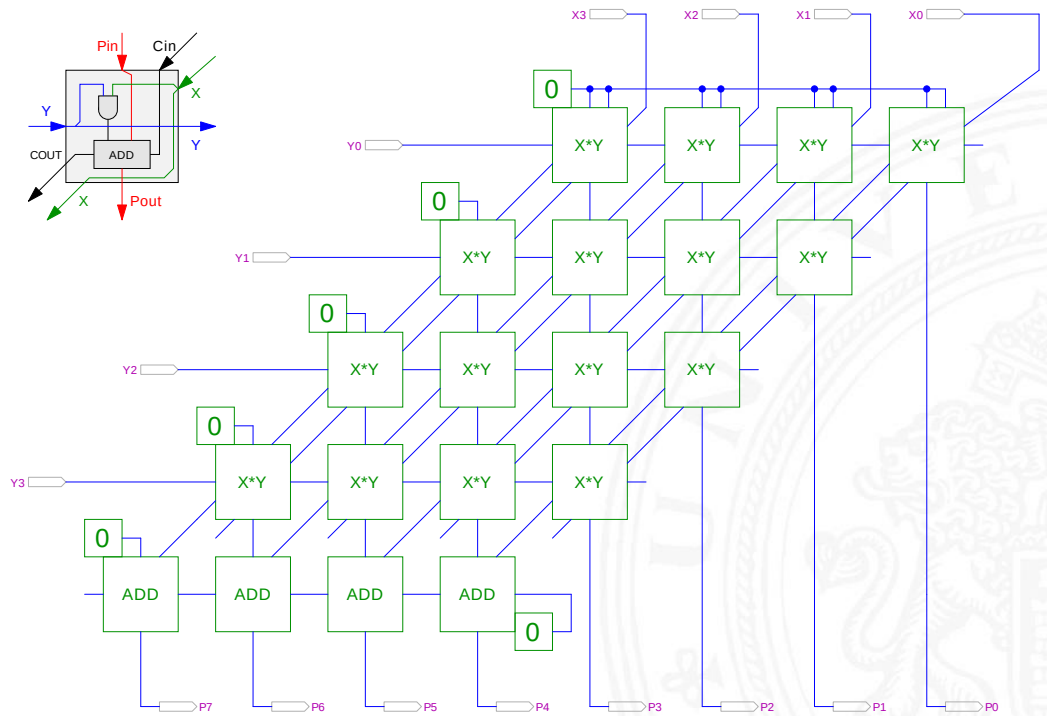
- Teilprodukte als UND-Verknüpfung des Multiplikators mit je einem Bit des Multiplikanden
- Aufaddieren der Teilprodukte mit Addierern
- Realisierung als Schaltnetz erfordert:
  n^2 UND-Gatter (bitweise eigentliche Multiplikation)
  n^2 Volladdierer (Aufaddieren der Teilprodukte)
- abschließend ein n-bit 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

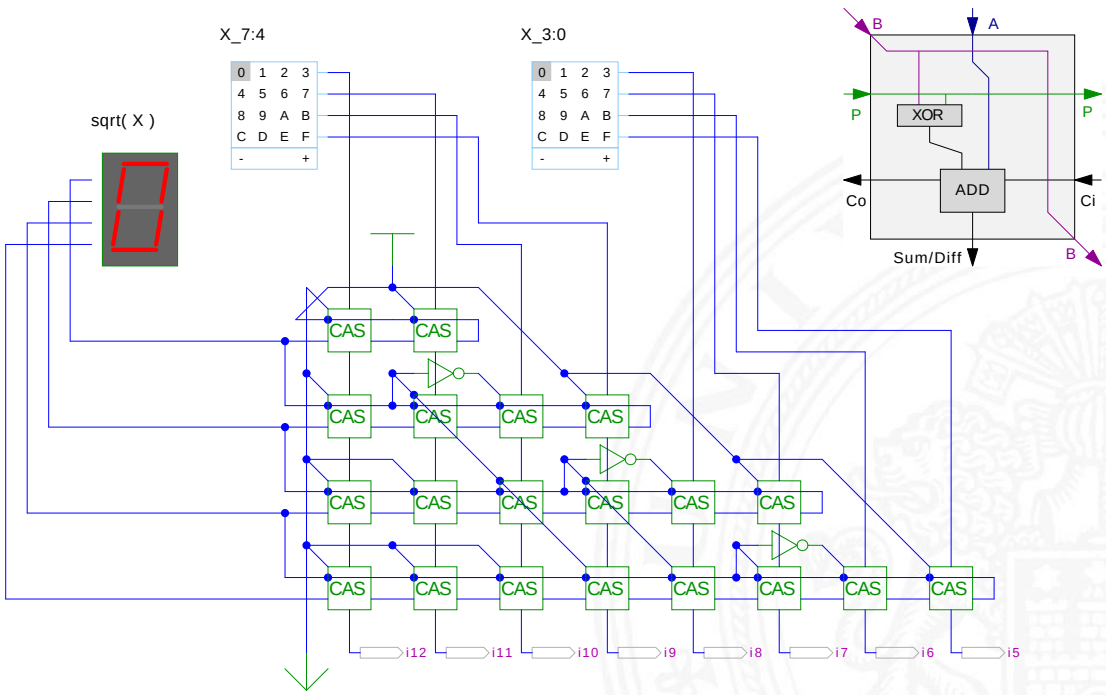
2x2-bit Multiplizierer – als zweistufiges Schaltnetz



4x4-bit Multiplizierer – Array



4x4-bit Quadratwurzel



Multiplizierer

weitere wichtige Themen aus Zeitgründen nicht behandelt

- *Booth*-Codierung
- *Carry-Save Adder* zur Summation der Teilprodukte
- Multiplikation von Zweierkomplementzahlen
- Multiplikation von Gleitkommazahlen

- CORDIC-Algorithmen
- bei Interesse: Literatur anschauen

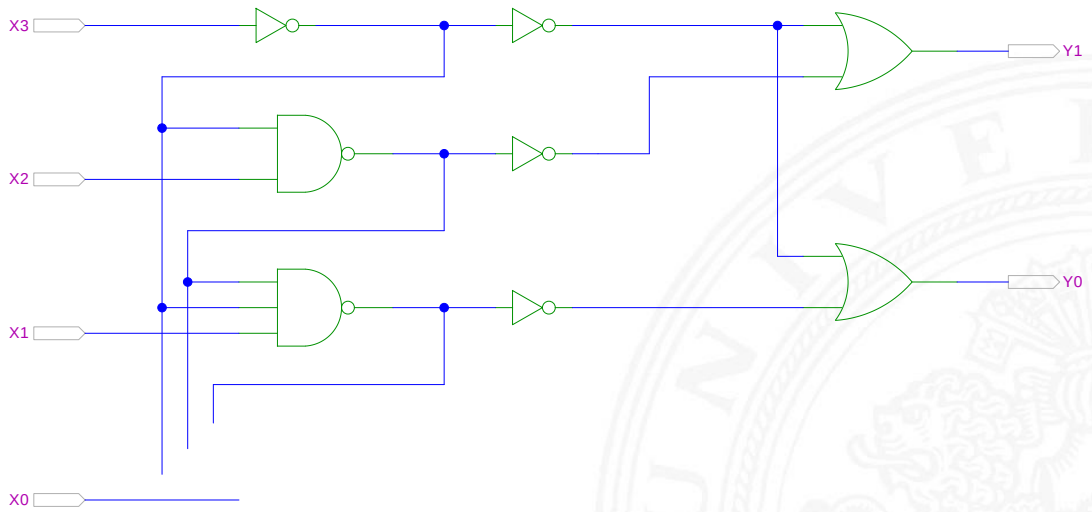
Priority Encoder

- Anwendung u.a. für Interrupt-Priorisierung
- Schaltung konvertiert n-bit Eingabe in eine Dualcodierung
- Wenn Bit n aktiv ist, werden alle niedrigeren Bits $(n-1), \ldots, 0$ ignoriert

x_3	x_2	x_1	x_0	y_1	y_0
1	*	*	*	1	1
0	1	*	*	1	0
0	0	1	*	0	1
0	0	0	*	0	0

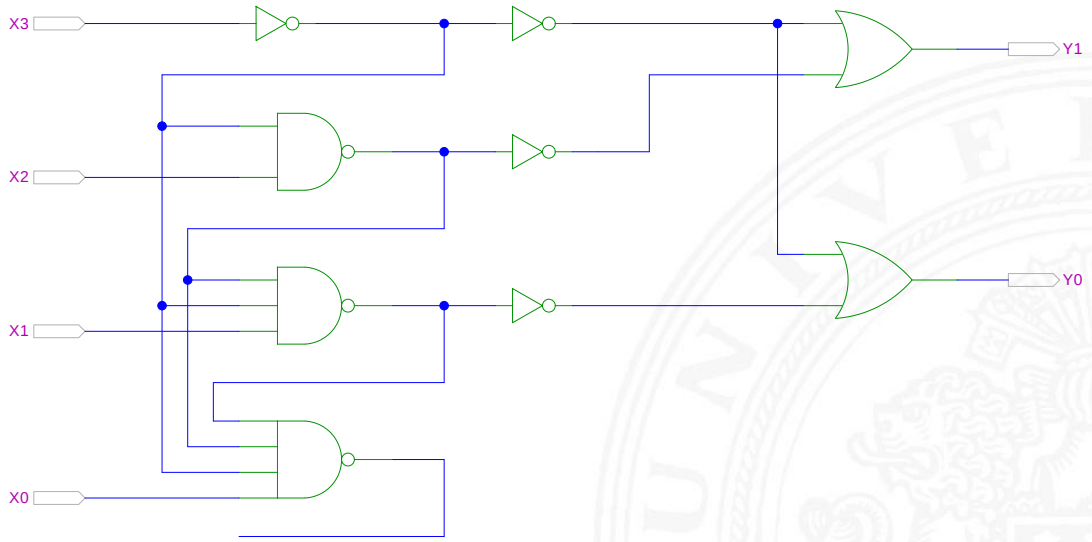
- unabhängig von niederwertigstem Bit, x_0 kann entfallen

4:2 Prioritätsencoder

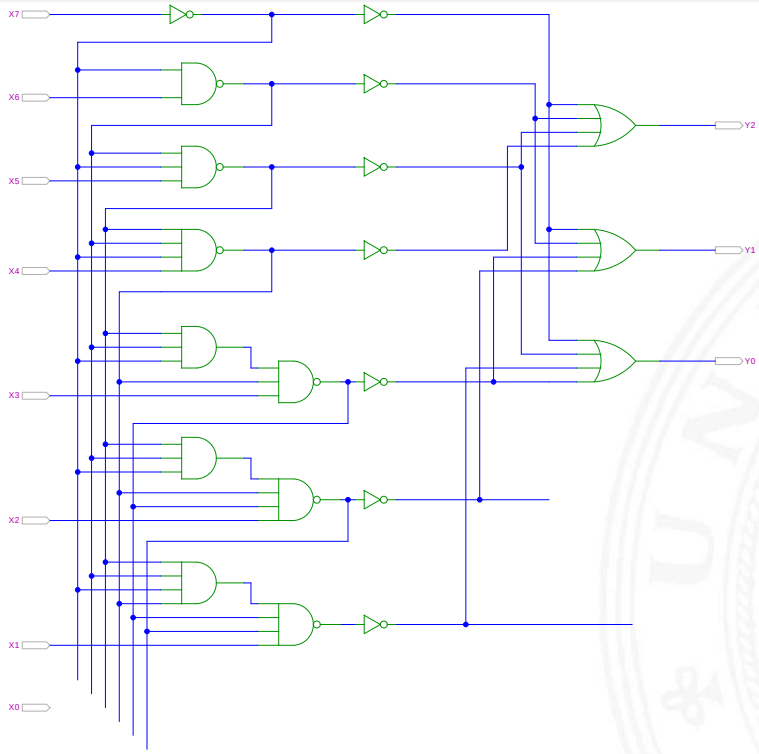


- zweistufige Realisierung
- aktive höhere Stufe blockiert alle niedrigeren Stufen

4:2 Prioritätsencoder: Kaskadierung

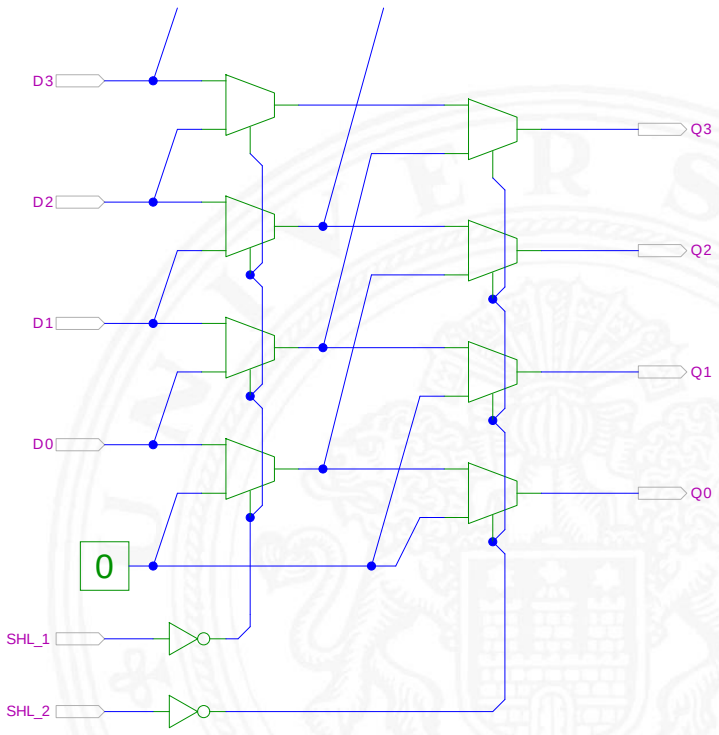


8:3 Prioritätsencoder

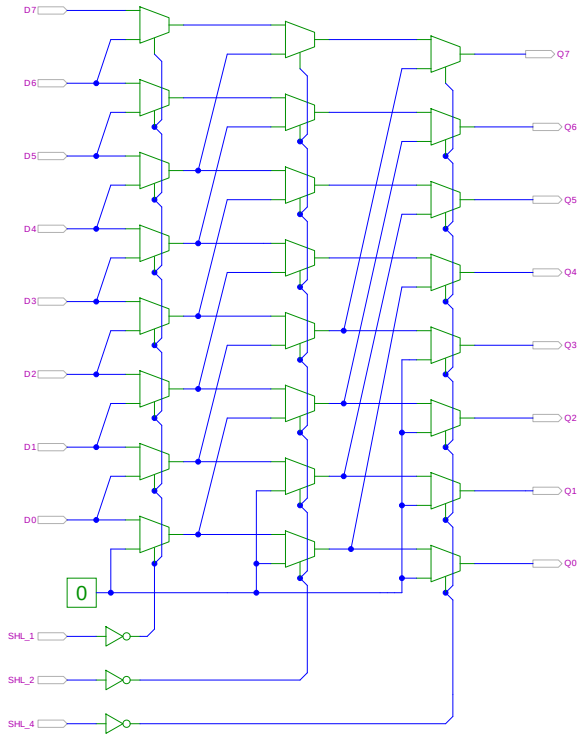


Shifter: zweistufig, shift-left um 0...3 Bits

- n-Dateneingänge D_i
  n-Datenausgänge Q_i
- 2:1 Multiplexer Kaskade
  - Stufe 0: benachbarte Bits
  - Stufe 1: übernächste Bits
  - usw.
- von rechts 0 nachschieben



8-bit Barrel-Shifter



Shift-Right, Rotate etc.

- Prinzip der oben vorgestellten Schaltungen gilt auch für alle übrigen Shift- und Rotate-Operationen
- Logic shift right: von links Nullen nachschieben
  Arithmetic shift right: oberstes Bit nachschieben
- Rotate left / right: außen herausgeschobene Bits auf der anderen Seite wieder hineinschieben

\+ alle Operationen typischerweise in einem Takt realisierbar
– Problem: Hardwareaufwand bei großen Wortbreiten und beliebigem Schiebe-/Rotate-Argument

Arithmetisch-Logische Einheit (ALU)

Arithmetisch-logische Einheit ALU (*Arithmetic Logic Unit*)

- kombiniertes Schaltnetz für arithmetische und logische Operationen
- das zentrale Rechenwerk in Prozessoren

Funktionsumfang variiert von Typ zu Typ

- Addition und Subtraktion — 2-Komplement
- bitweise logische Operationen — Negation, UND, ODER, XOR
- Schiebeoperationen — shift, rotate
- evtl. Multiplikation

- Integer-Division selten verfügbar (separates Rechenwerk)

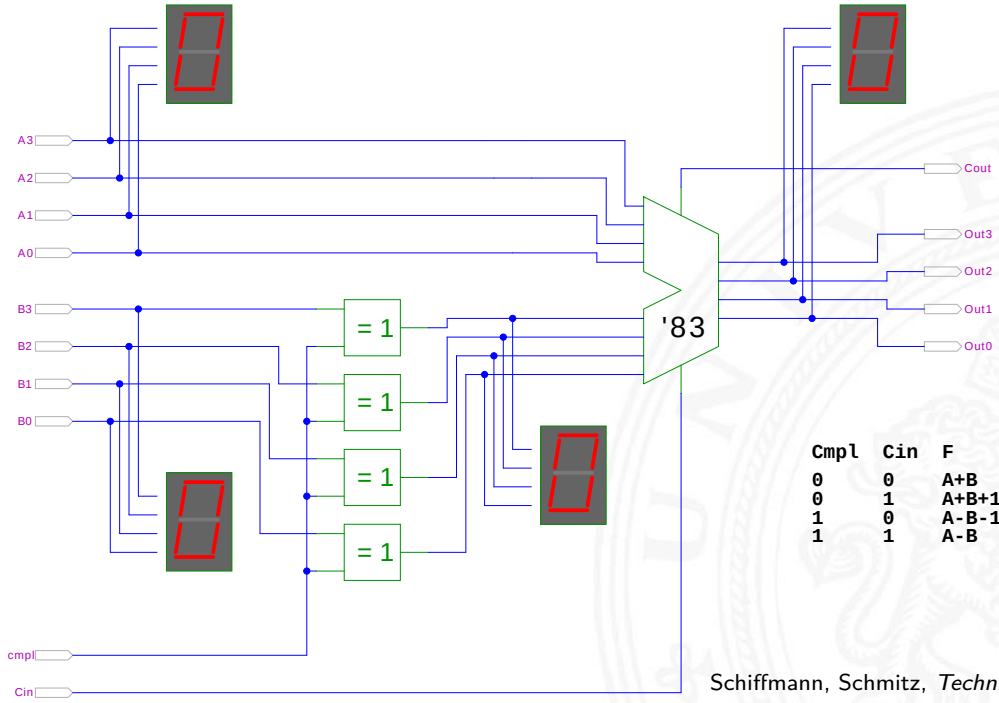
ALU: Addierer und Subtrahierer

- Addition $(A + B)$ mit normalem Addierer

- XOR-Gatter zum Invertieren von Operand B
- Steuerleitung *sub* aktiviert das Invertieren und den Carry-in c_i
- wenn aktiv, Subtraktion als $(A - B) = A + \neg B + 1$
- ggf. auch Inkrement $(A + 1)$ und Dekrement $(A - 1)$

- folgende Folien: 7483 ist IC mit 4-bit Addierer

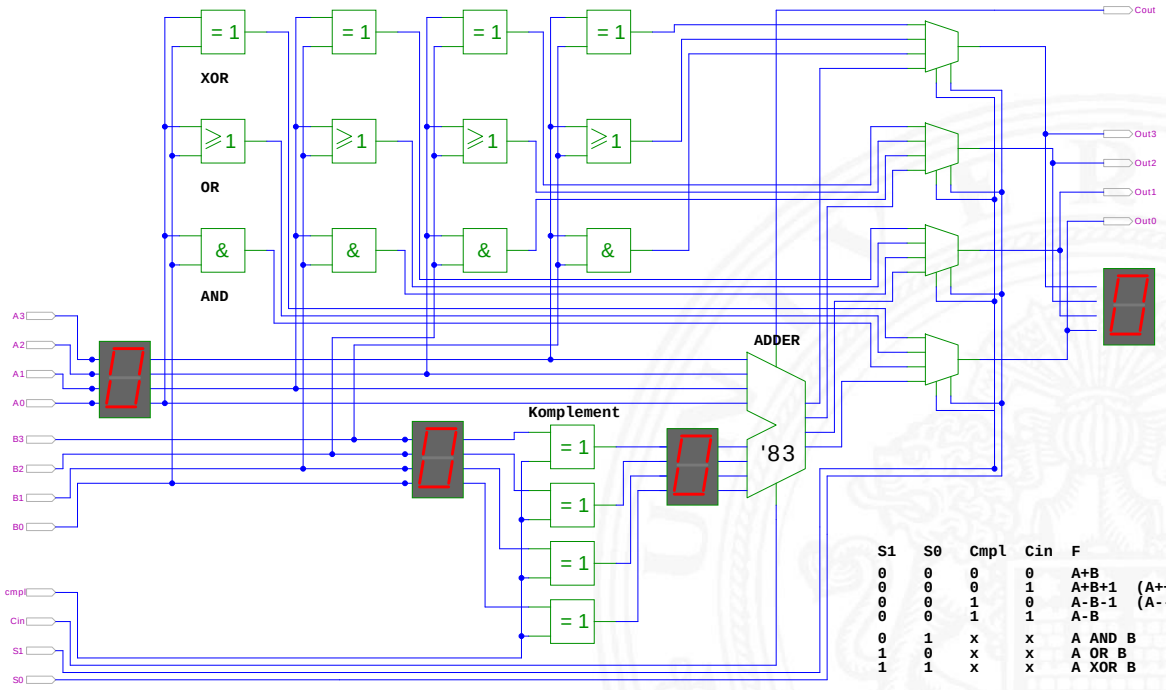
ALU: Addierer und Subtrahierer



Cmpl	Cin	F
0	0	A+B
0	1	A+B+1 (A++)
1	0	A-B-1 (A--)
1	1	A-B

Schiffmann, Schmitz, *Technische Informatik I*

ALU: Addierer und bitweise Operationen



S1	S0	Cmpl	Cin	F
0	0	0	0	A+B
0	0	0	1	A+B+1 (A++)
0	0	1	0	A-B-1 (A--)
0	0	1	1	A-B
0	1	x	x	A AND B
1	0	x	x	A OR B
1	1	x	x	A XOR B

Schiffmann, Schmitz, *Technische Informatik I*

ALU: Prinzip

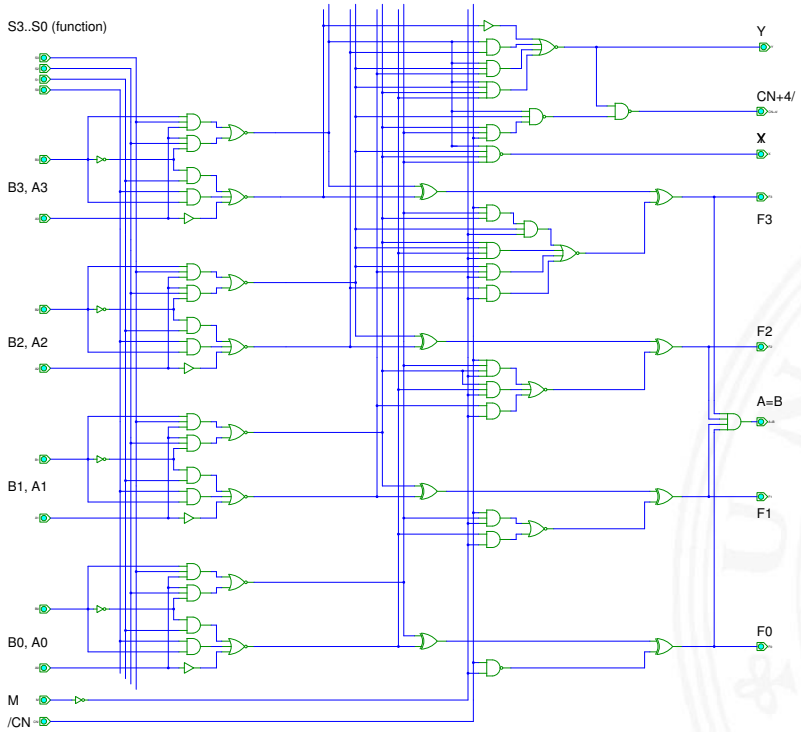
vorige Folie zeigt die „triviale“ Realisierung einer ALU

- 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)

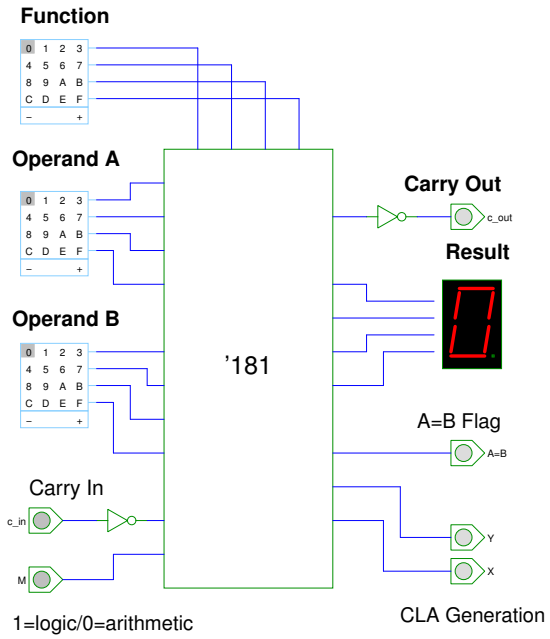
- erste Stufe für bitweise logische Operationen und Komplement
- zweite Stufe als Carry-lookahead Addierer
- weniger Gatter und schneller

ALU: 74181 – Aufbau



selection S3 S2 S1 S0	logic functions M = H	arithmetic functions M = L, Cn=H (no carry)
L L L L	F = !A	F = A
L L L H	F = !(A or B)	F = A or B
L L H L	F = !A * B	F = A or !B
L L H H	F = !A * B	F = MINUS 1
L H L L	F = 0	F = A PLUS (A*!B)
L H L H	F = !B	F = (A or B) PLUS (A * !B)
L H H L	F = A xor B	F = A MINUS B MINUS 1
L H H H	F = A * !B	F = (A * !B) MINUS 1
H L L L	F = !A or B	F = A PLUS (A*B)
H L L H	F = A xnor B	F = A PLUS B
H L H L	F = B	F = (A or !B) PLUS (A*B)
H L H H	F = A * B	F = (A*B) MINUS 1
H H L L	F = 1	F = A PLUS A
H H L H	F = A or !B	F = (A or B) PLUS A
H H H L	F = A or B	F = (A or !B) PLUS A
H H H H	F = A	F = A MINUS 1
		Cn=L: PLUS 1

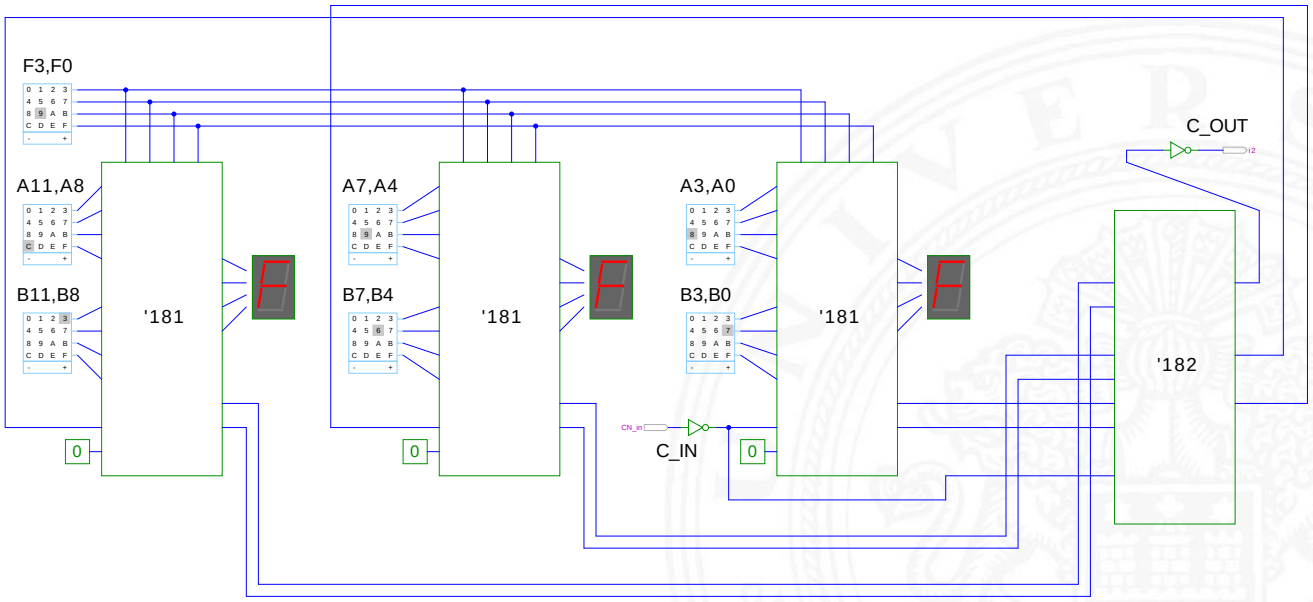
ALU: 74181 – Funktionstabelle



selection				logic functions	arithmetic functions
S3	S2	S1	S0	M = H	M = L, Cn=H (no carry)
L	L	L	L	F = !A	F = A
L	L	L	H	F = !(A or B)	F = A or B
L	L	H	L	F = !A * B	F = A or !B
L	L	H	H	F = !A * B	F = MINUS 1
L	H	L	L	F = 0	F = A PLUS (A*!B)
L	H	L	H	F = !B	F = (A or B) PLUS (A * !B)
L	H	H	L	F = A xor B	F = A MINUS B MINUS 1
L	H	H	H	F = A * !B	F = (A * !B) MINUS 1
H	L	L	L	F = !A or B	F = A PLUS (A*B)
H	L	L	H	F = A xnor B	F = A PLUS B
H	L	H	L	F = B	F = (A or !B) PLUS (A*B)
H	L	H	H	F = A * B	F = (A*B) MINUS 1
H	H	L	L	F = 1	F = A PLUS A
H	H	L	H	F = A or !B	F = (A or B) PLUS A
H	H	H	L	F = A or B	F = (A or !B) PLUS A
H	H	H	H	F = A	F = A MINUS 1
					Cn=L: PLUS 1

ALU: 74181 und 74182 CLA

12-bit ALU mit Carry-Lookahead Generator 74182



Literatur: Vertiefung

- Donald E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 0: Introduction to Combinatorial Algorithms and Boolean Functions*, Addison-Wesley, 2008
- Donald E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques, Binary Decision Diagrams*, Addison-Wesley, 2009

- Ingo Wegener, *The Complexity of Boolean Functions*, Wiley, 1987 `ls2-www.cs.uni-dortmund.de/monographs/bluebook`

- Bernd Becker, Rolf Drechsler, Paul Molitor, *Technische Informatik: Eine Einführung*, Pearson Studium, 2005
  Besonderheit: Einführung von BDDs/ROBDDs

Interaktives Lehrmaterial

- Klaus von der Heide,
  Vorlesung: Technische Informatik 1 — interaktives Skript
  `tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1`

- Norman Hendrich,
  HADES — HAmburg DEsign System
  `tams.informatik.uni-hamburg.de/applets/hades`
  KV-Diagram Simulation
  `tams.informatik.uni-hamburg.de/applets/kvd`

- John Lazarro,
  Chipmunk design tools (AnaLog, DigLog)
  `www.cs.berkeley.edu/~lazzaro/chipmunk`

Gliederung

Gliederung (cont.)

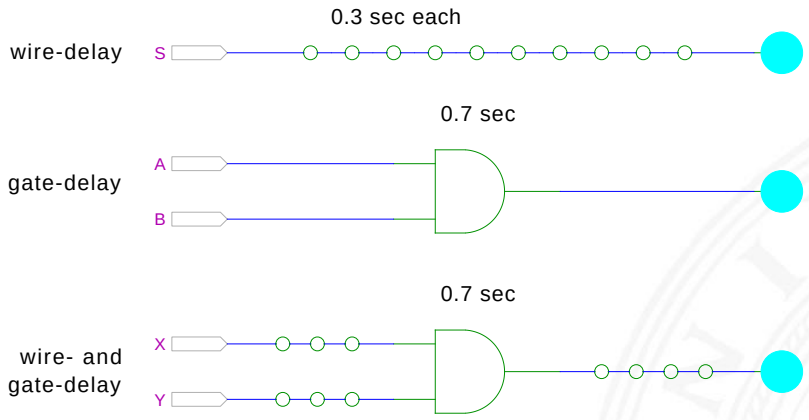
Zeitverhalten einer Schaltung: Modellierung

Wie wird das Zeitverhalten eines Schaltnetzes modelliert?

Gängige Abstraktionsebenen mit zunehmendem Detaillierungsgrad

1. algebraische Ausdrücke: keine zeitliche Abhängigkeit
2. „fundamentales Modell“: Einheitsverzögerung des algebraischen Ausdrucks um eine Zeit τ
3. individuelle Gatterverzögerungen
   - mehrere Modelle, unterschiedlich detailliert
   - Abstraktion elektrischer Eigenschaften
4. Gatterverzögerungen + Leitungslaufzeiten (geschätzt, berechnet)
5. Differentialgleichungen für Spannungen und Ströme (verschiedene „Ersatzmodelle“)

Gatterverzögerung vs. Leitungslaufzeiten



- früher: Gatterverzögerungen $\gg$ Leitungslaufzeiten
- Schaltungen modelliert durch Gatterlaufzeiten
- aktuelle „Submicron“-Halbleitertechnologie: Leitungslaufzeiten $\gg$ Gatterverzögerungen

Zeitverhalten

- alle folgenden Schaltungsbeispiele werden mit Gatterverzögerungen modelliert
- Gatterlaufzeiten als Vielfache einer Grundverzögerung (τ)
- aber Leitungslaufzeiten ignoriert

- mögliche Verfeinerungen
  - gatterabhängige Schaltzeiten für INV, NAND, NOR, XOR etc.
  - unterschiedliche Schaltzeiten für Wechsel: 0→1 und 1→0
  - unterschiedliche Schaltzeiten für 2-, 3-, 4-Input Gatter
  - Schaltzeiten sind abhängig von der Anzahl nachfolgender Eingänge (engl. *fanout*)

Exkurs: Lichtgeschwindigkeit und Taktraten

- Lichtgeschwindigkeit im Vakuum: $c \approx 300\,000\,\text{km/sec}$
  $\approx 30\,\text{cm/ns}$
- in Metallen und Halbleitern langsamer: $c \approx 20\,\text{cm/ns}$

$\Rightarrow$ bei 1 Gigahertz Takt: Ausbreitung um ca. 20 Zentimeter

Abschätzungen:

- Prozessor: ca. 2 cm Diagonale $\approx$ 10 GHz Taktrate
- Platine: ca. 20 cm Kantenlänge $\approx$ 1 GHz Takt
- UKW-Radio: 100 MHz, 2 Meter Wellenlänge

$\Rightarrow$ prinzipiell kann (schon heute) ein Signal innerhalb eines Takts nicht von einer Ecke des ICs zur Anderen gelangen

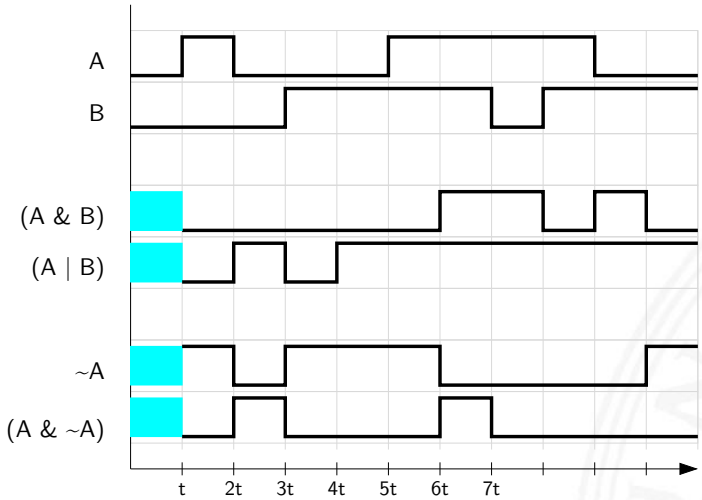
Impulsdiagramme

- **Impulsdiagramm** (engl. *waveform*): Darstellung der logischen Werte einer Schaltfunktion als Funktion der Zeit

- als Abstraktion des tatsächlichen Verlaufs
- Zeit läuft von links nach rechts
- Schaltfunktion(en): von oben nach unten aufgelistet

- Vergleichbar den Messwerten am Oszilloskop (analoge Werte) bzw. den Messwerten am Logic-State-Analyzer (digitale Werte)
- ggf. Darstellung mehrerer logischer Werte (z.B. 0,1,Z,U,X)

Impulsdiagramm: Beispiel



- im Beispiel jeweils eine „Zeiteinheit“ Verzögerung für jede einzelne logische Operation
- Ergebnis einer Operation nur, wenn die Eingaben definiert sind
- im ersten Zeitschritt noch undefinierte Werte

Hazards

- **Hazard**: die Eigenschaft einer Schaltfunktion, bei bestimmten Kombinationen der individuellen Verzögerungen ihrer Verknüpfungsglieder ein Fehlverhalten zu zeigen

- **Hazardfehler**: das aktuelle Fehlverhalten einer realisierten Schaltfunktion aufgrund eines Hazards

Hazards: Klassifikation

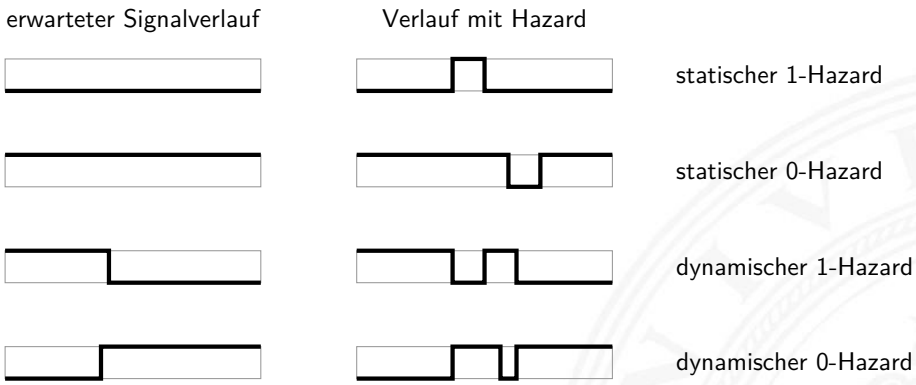
nach der Erscheinungsform am Ausgang

- **statisch**: der Ausgangswert soll stabil 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

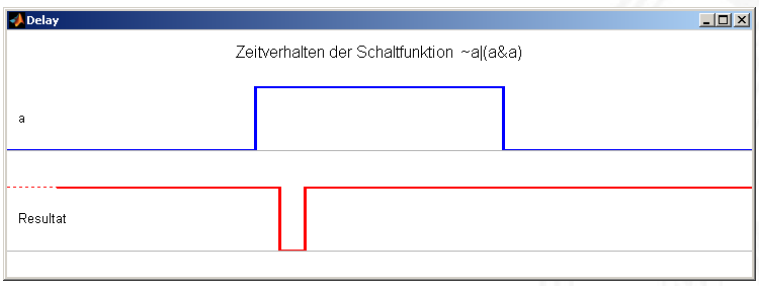
Hazards: statisch vs. dynamisch



- 1-Hazard wenn fehlerhaft der Wert 1 auftritt, und umgekehrt
- es können natürlich auch mehrfache Hazards auftreten
- Hinweis: Begriffsbildung in der Literatur nicht einheitlich

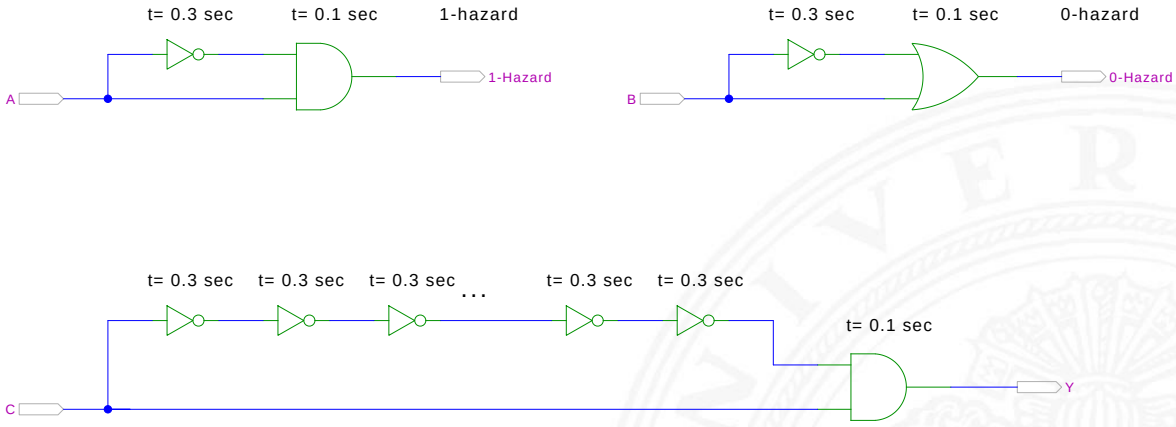
Hazards: Strukturhazard

- **Strukturhazard** wird durch die gewählte Struktur der Schaltung verursacht
- auch, wenn sich nur eine Variable ändert
- Beispiel: $f(a) = \neg a \vee (a \wedge a)$
  $\neg a$ schaltet schneller ab, als $(a \wedge a)$ einschaltet



- Hazard kann durch Modifikation der Schaltung beseitigt werden
  im Beispiel mit: $f(a) = 1$

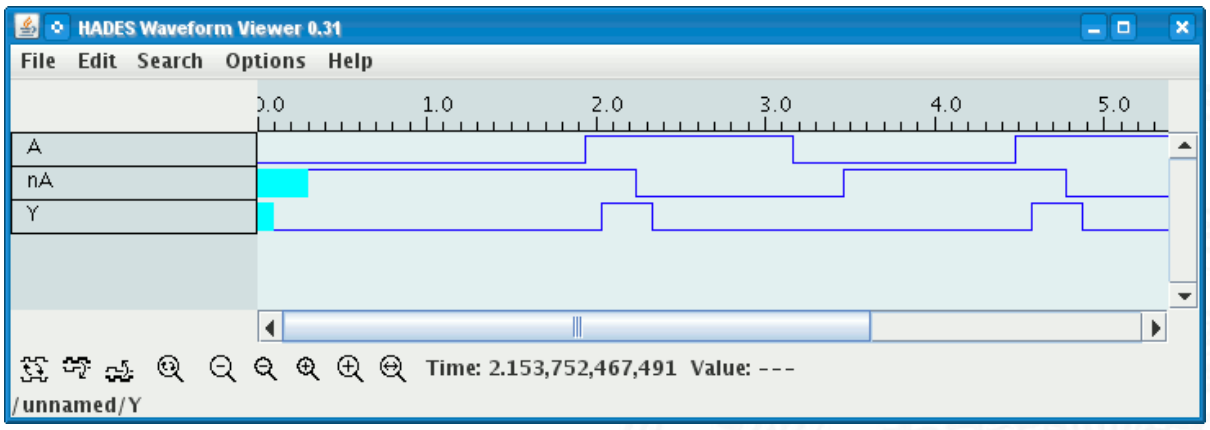
Strukturhazards: Beispiele



- logische Funktion ist $(a \wedge \overline{a}) = 0$ bzw. $(a \vee \overline{a}) = 1$
- aber ein Eingang jeweils durch Inverter verzögert

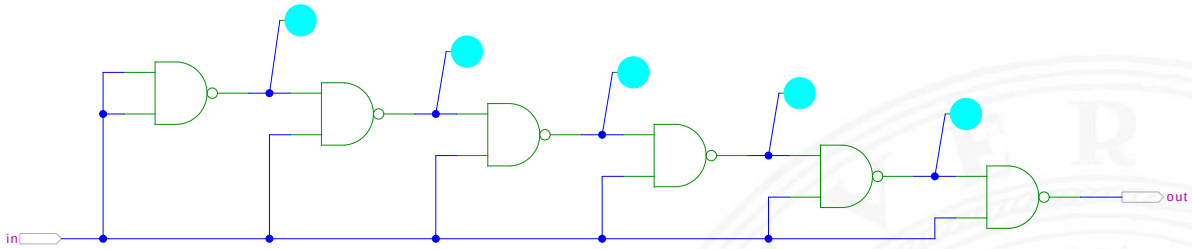
$\Rightarrow$ kurzer Impuls beim Umschalten von 0→1 bzw. 1→0

Strukturhazards: Beispiele (cont.)



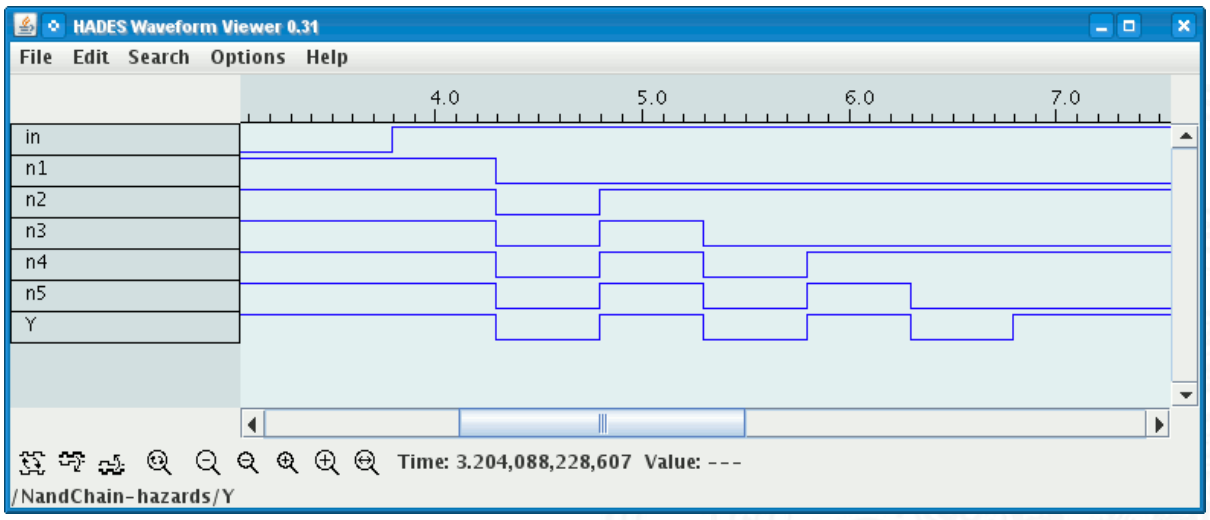
- Schaltung $(a \wedge \overline{a}) = 0$ erzeugt (statischen-1) Hazard
- Länge des Impulses abhängig von Verzögerung im Inverter
- Kette von Invertern erlaubt Einstellung der Pulslänge

Strukturhazards extrem: NAND-Kette



- 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 (dynamischen-0) Hazards
- Anzahl der Impulse abhängig von Anzahl der Gatter

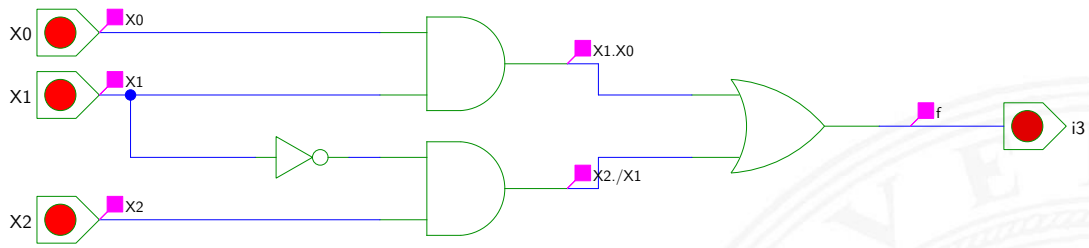
Strukturhazards im KV-Diagramm

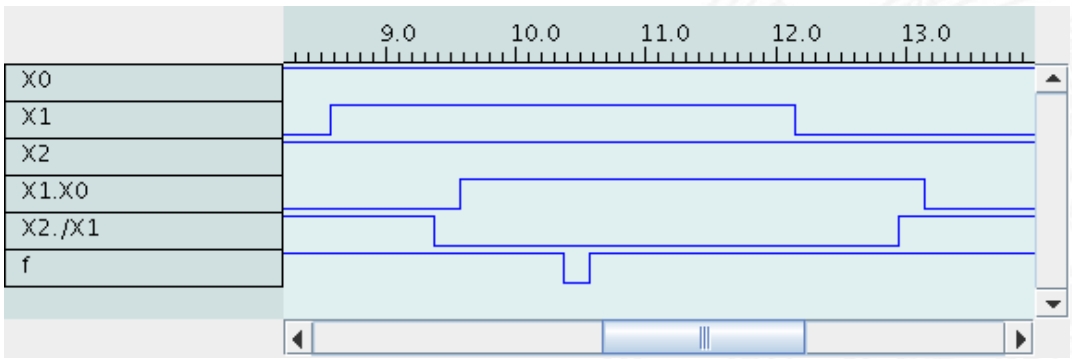
- Funktion $f = (x_2\overline{x_1}) \vee (x_1x_0)$
- realisiert in disjunktiver Form mit 2 Schleifen

Strukturhazard beim Übergang von $(x_2\overline{x_1}x_0)$ nach $(x_2x_1x_0)$

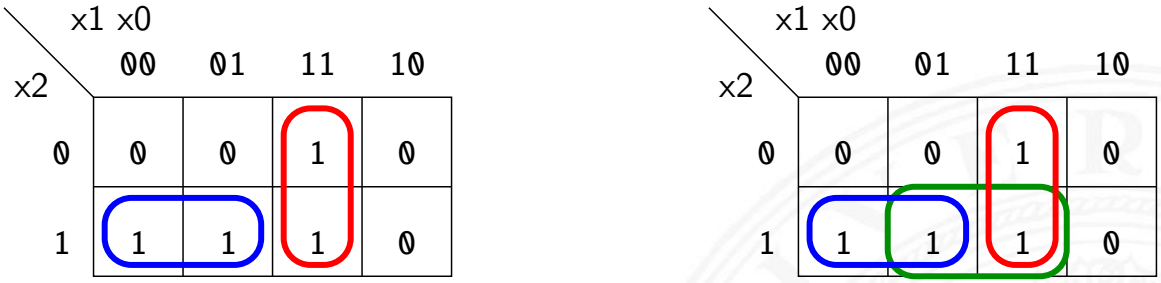
- Gatter $(x_2\overline{x_1})$ schaltet ab, Gatter (x_1x_0) schaltet ein
- Ausgang evtl. kurz 0, abhängig von Verzögerungen

Strukturhazards im KV-Diagramm (cont.)





Strukturhazards beseitigen

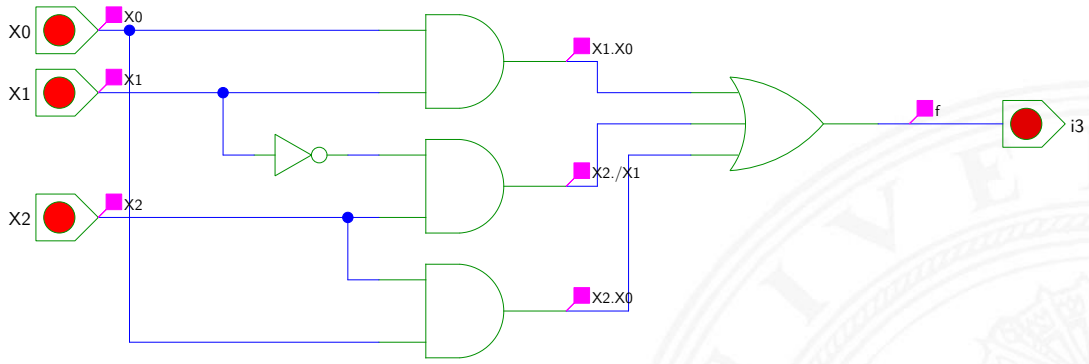


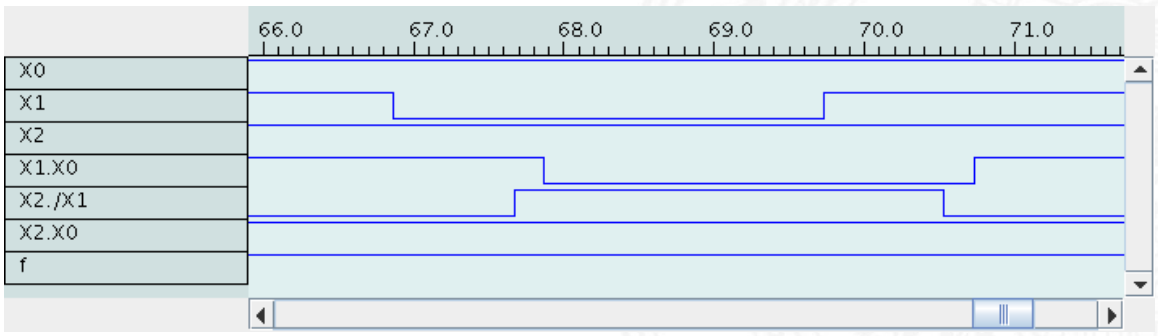
- Funktion $f = (x_2\overline{x_1}) \vee (x_1x_0)$
- realisiert in disjunktiver Form mit **3 Schleifen**
  $f = (x_2\overline{x_1}) \vee (x_1x_0) \vee (x_2x_0)$

\+ Strukturhazard durch zusätzliche Schleife beseitigt

– aber höhere Hardwarekosten als bei minimierter Realisierung

Strukturhazards beseitigen (cont.)



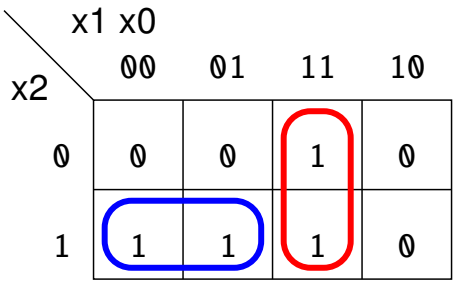


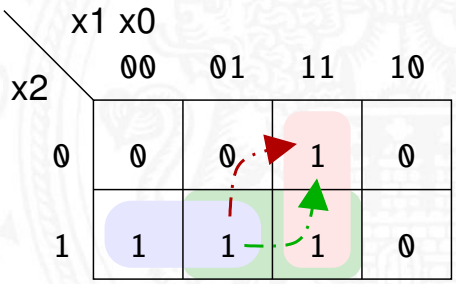
Hazards: Funktionshazard

- **Funktionshazard** kann bei gleichzeitigem Wechsel mehrerer Eingangswerte als Eigenschaft der Schaltfunktion entstehen
- Problem: Gleichzeitigkeit an Eingängen

$\Rightarrow$ Funktionshazard kann nicht durch strukturelle Maßnahmen verhindert werden

- Beispiel: Übergang von $(x_2\overline{x_1}x_0)$ nach $(\overline{x_2}x_1x_0)$





Gliederung

Gliederung (cont.)

Gliederung (cont.)

Schaltwerke

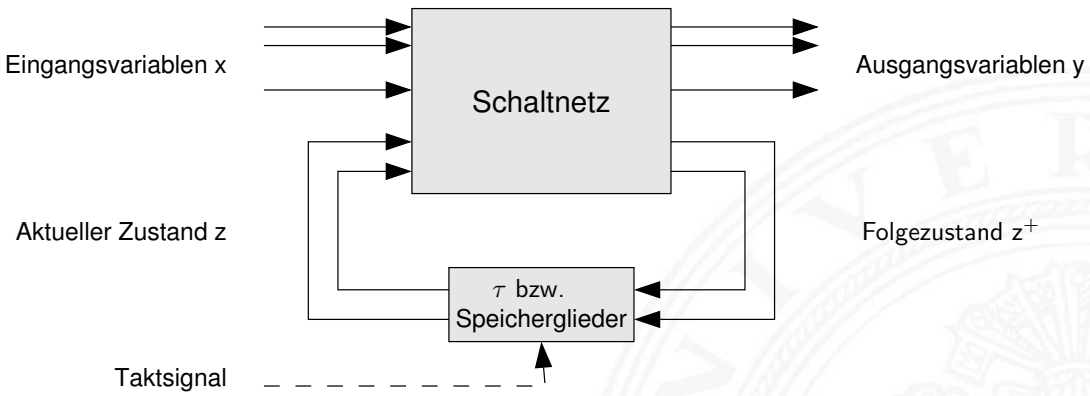
- **Schaltwerk**: Schaltung mit Rückkopplungen und Verzögerungen

- fundamental andere Eigenschaften als Schaltnetze
- Ausgangswerte nicht nur von Eingangswerten abhängig sondern auch von der Vorgeschichte

$\Rightarrow$ interner Zustand repräsentiert „Vorgeschichte“

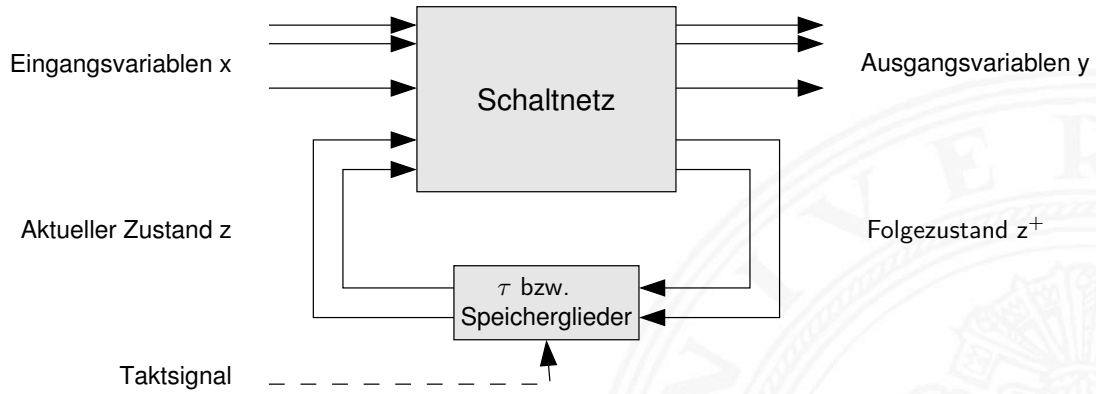
- ggf. stabile Zustände $\Rightarrow$ Speicherung von Information
- bei unvorsichtigem Entwurf: chaotisches Verhalten

Schaltwerke: Blockschaltbild



- Eingangsvariablen x und Ausgangsvariablen y
- Aktueller Zustand z
- Folgezustand z^+
- Rückkopplung läuft über Verzögerungen τ / Speicherglieder

Schaltwerke: Blockschaltbild (cont.)



zwei prinzipielle Varianten für die Zeitglieder

1. nur (Gatter-) Verzögerungen: **asynchrone** oder **nicht getaktete Schaltwerke**
2. getaktete Zeitglieder: **synchrone** oder **getaktete Schaltwerke**

Synchrone und Asynchrone Schaltwerke

- **synchrone Schaltwerke**: die Zeitpunkte, an denen das Schaltwerk von einem stabilen Zustand in einen stabilen Folgezustand übergeht, werden explizit durch ein Taktsignal (*clock*) vorgegeben

- **asynchrone Schaltwerke**: hier fehlt ein Taktgeber, Änderungen der Eingangssignale wirken sich unmittelbar aus (entsprechend der Gatterverzögerungen τ)
- potentiell höhere Arbeitsgeschwindigkeit
- aber sehr aufwendiger Entwurf
- fehleranfälliger (z.B. leicht veränderte Gatterverzögerungen durch Bauteil-Toleranzen, Spannungsschwankungen, usw.)

Theorie: Endliche Automaten

FSM – **F**inite **S**tate **M**achine

- Deterministischer Endlicher Automat mit Ausgabe
- 2 äquivalente Modelle
  - Mealy: Ausgabe hängt *von Zustand und Eingabe* ab
  - Moore: –"– *nur vom Zustand* ab
- 6-Tupel $(Z, \Sigma, \Delta, \delta, \lambda, z_0)$
  - Z Menge von Zuständen
  - Σ Eingabealphabet
  - Δ Ausgabealphabet
  - δ Übergangsfunktion $\delta : Z \times \Sigma \to Z$
  - λ Ausgabefunktion $\lambda : Z \times \Sigma \to \Delta$ Mealy-Modell
    $\lambda : Z \to \Delta$ Moore- –"–
  - z_0 Startzustand

Mealy-Modell und Moore-Modell

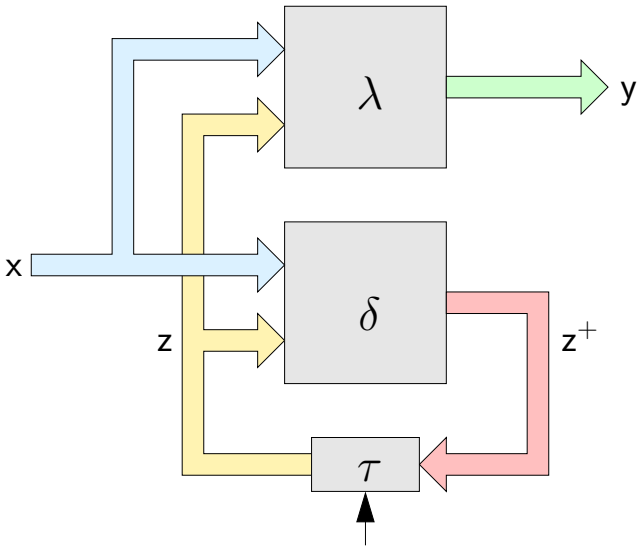
- **Mealy-Modell**: die Ausgabe hängt vom Zustand z und vom momentanen Input x ab
- **Moore-Modell**: die Ausgabe des Schaltwerks hängt nur vom aktuellen Zustand z ab

- **Ausgabefunktion**: $y = \lambda(z, x)$ Mealy
  $y = \lambda(z)$ Moore
- **Überführungsfunktion**: $z^+ = \delta(z, x)$ Moore und Mealy

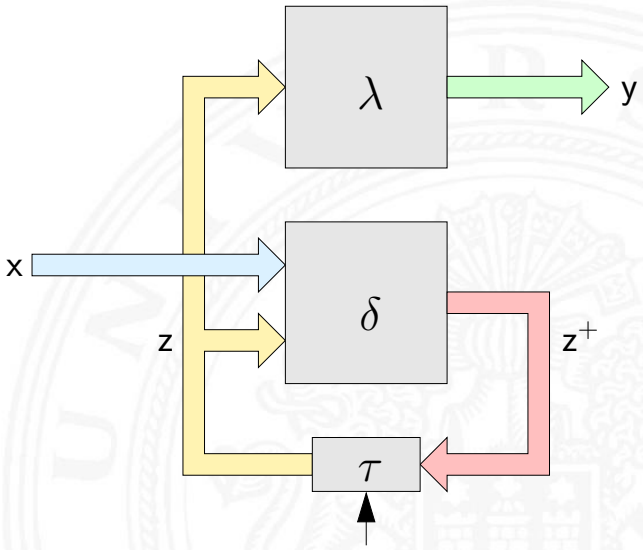
- **Speicherglieder** oder Verzögerung τ im Rückkopplungspfad

Mealy-Modell und Moore-Modell (cont.)

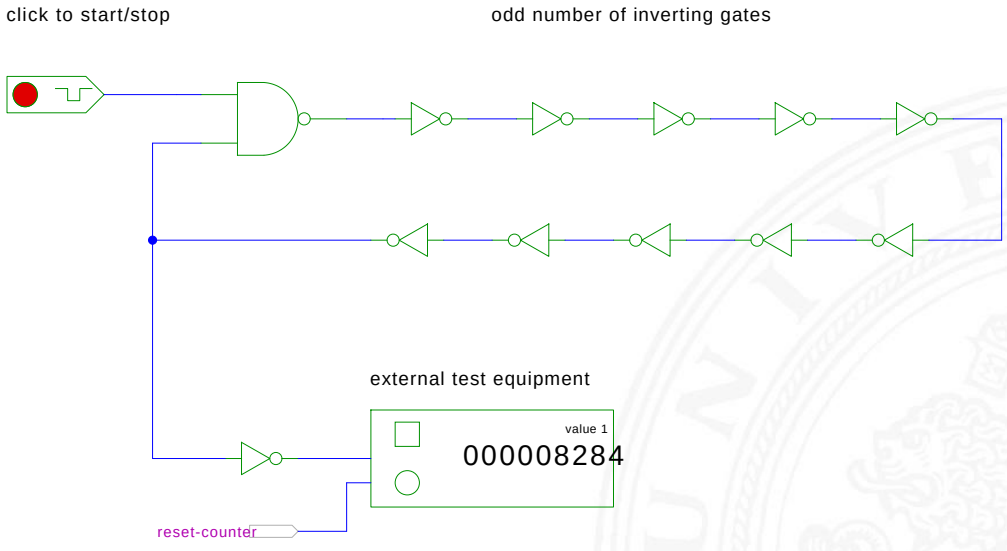
- Mealy-Automat



Moore-Automat

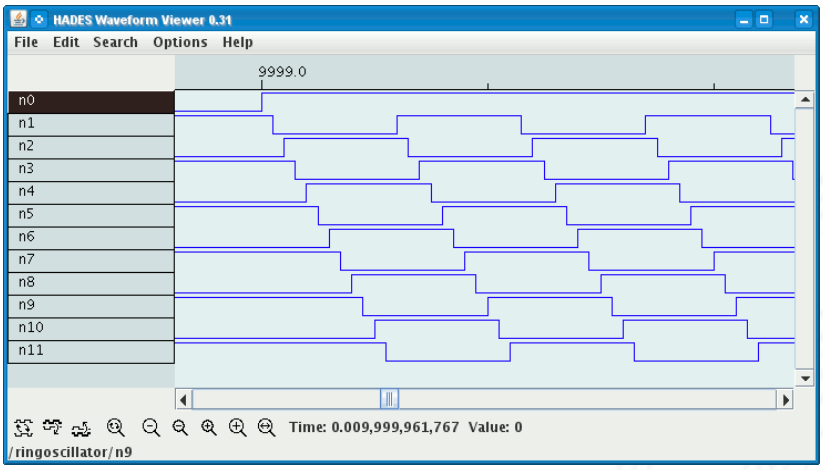


Asynchrone Schaltungen: Beispiel Ringoszillator



- stabiler Zustand, solange der Eingang auf 0 liegt
- instabil sobald der Eingang auf 1 wechselt (Oszillation)

Asynchrone Schaltungen: Beispiel Ringoszillator (cont.)



- ungerade Anzahl n invertierender Gatter ($n \geq 3$)
- Start/Stop über steuerndes NAND-Gatter
- Oszillation mit maximaler Schaltfrequenz
  z.B.: als Testschaltung für neue (Halbleiter-) Technologien

Asynchrone Schaltungen: Probleme

- das Schaltwerk kann stabile und nicht-stabile Zustände enthalten
- die Verzögerungen der Bauelemente sind nicht genau bekannt und können sich im Betrieb ändern
- Variation durch Umweltparameter, z.B. Temperatur, Versorgungsspannung, Alterung

- sehr schwierig, die korrekte Funktion zu garantieren
- z.B. mehrstufige Handshake-Protokolle

- in der Praxis überwiegen synchrone Schaltwerke
- Realisierung mit **Flipflops** als Zeitgliedern

Synchrone Schaltungen

- alle Rückkopplungen der Schaltung laufen über spezielle Zeitglieder: „Flipflops“
- diese definieren / speichern einen stabilen Zustand, unabhängig von den Eingabewerten und Vorgängen im δ-Schaltnetz

- Hinzufügen eines zusätzlichen Eingangssignals: „Takt“
- die Zeitglieder werden über das Taktsignal gesteuert verschiedene Möglichkeiten: Pegel- und Flankensteuerung, Mehrphasentakte (s.u.)

$\Rightarrow$ synchrone Schaltwerke sind wesentlich einfacher zu entwerfen und zu analysieren als asynchrone Schaltungen

Zeitglieder / Flipflops

- **Zeitglieder**: Bezeichnung für die Bauelemente, die den Zustand des Schaltwerks speichern können
- **bistabile Bauelemente** (Kippglieder) oder **Flipflops**

- zwei stabile Zustände $\Rightarrow$ speichert 1 Bit

  1 – Setzzustand

  0 – Rücksetzzustand

- Übergang zwischen Zuständen durch geeignete Ansteuerung

Flipflops

- Name für die **elementaren** Schaltwerke
- mit genau zwei Zuständen Z_0 und Z_1
- Zustandsdiagramm hat zwei Knoten und vier Übergänge (s.u.)

- Ausgang als Q bezeichnet und dem Zustand gleichgesetzt
- meistens auch invertierter Ausgang $\overline{Q}$ verfügbar

- Flipflops sind selbst nicht getaktet
- sondern „sauber entworfene“ asynchrone Schaltwerke
- Anwendung als Verzögerungs-/Speicherelemente in getakteten Schaltwerken

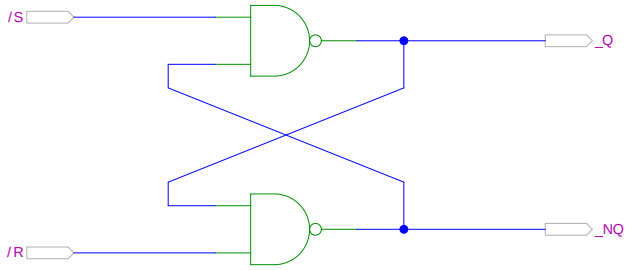
Flipflops: Typen

- Basis-Flipflop „Reset-Set-Flipflop“
- getaktetes RS-Flipflop

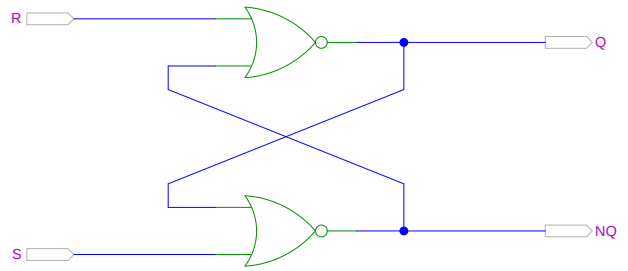
- pegelgesteuertes D-Flipflop „D-Latch“
- flankengesteuertes D-Flipflop „D-Flipflop“

- JK-Flipflop
- weitere...

RS-Flipflop: NAND- und NOR-Realisierung

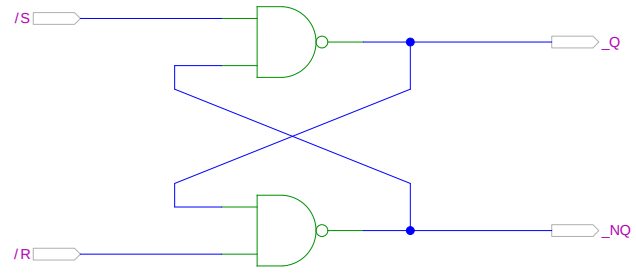


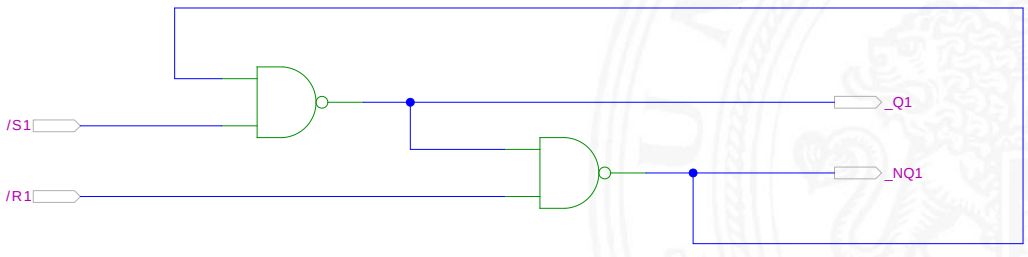
/S	/R	Q	NQ	NAND
0	0	1	1	(forbidden)
0	1	1	0	
1	0	0	1	
1	1	Q*	NQ*	(store)



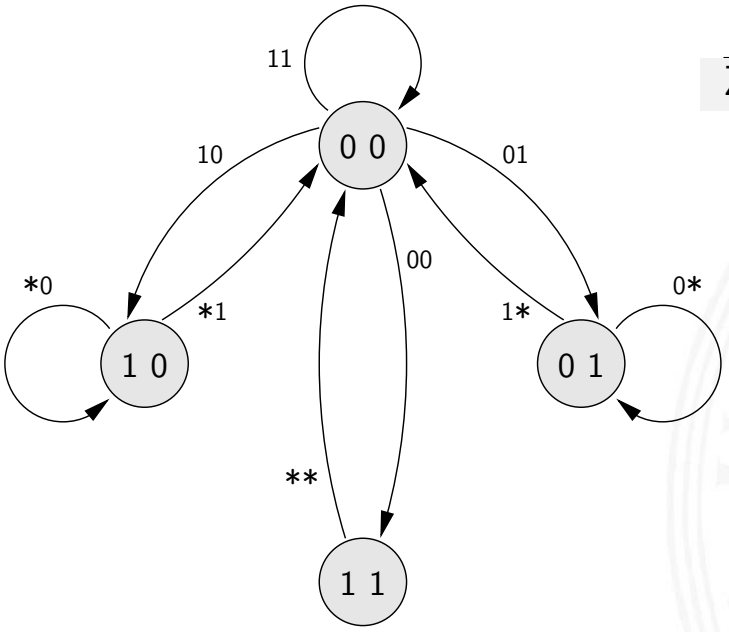
S	R	Q	NQ	NOR
0	0	Q*	NQ*	(store)
0	1	0	1	
1	0	1	0	
1	1	0	0	(forbidden)

RS-Flipflop: Varianten des Schaltbilds





NOR RS-Flipflop: Zustandsdiagramm und Flusstafel



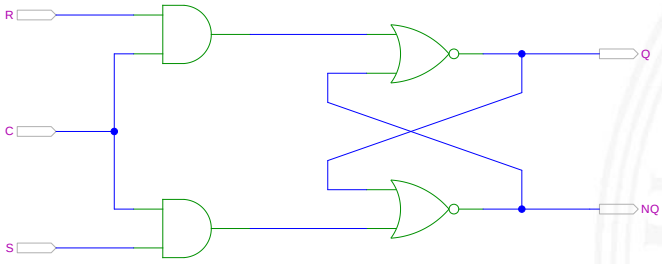
	Eingabe $[S\,R]$			
	00	01	11	10
Zustand	Folgezustand			$[Q\,\overline{Q}]$
00	11	01	00	10
01	01	01	00	00
11	00	00	00	00
10	10	00	00	10

stabiler Zustand

RS-Flipflop mit Takt

- RS-Basisflipflop mit zusätzlichem Takteingang C
- Änderungen nur wirksam, während C aktiv ist

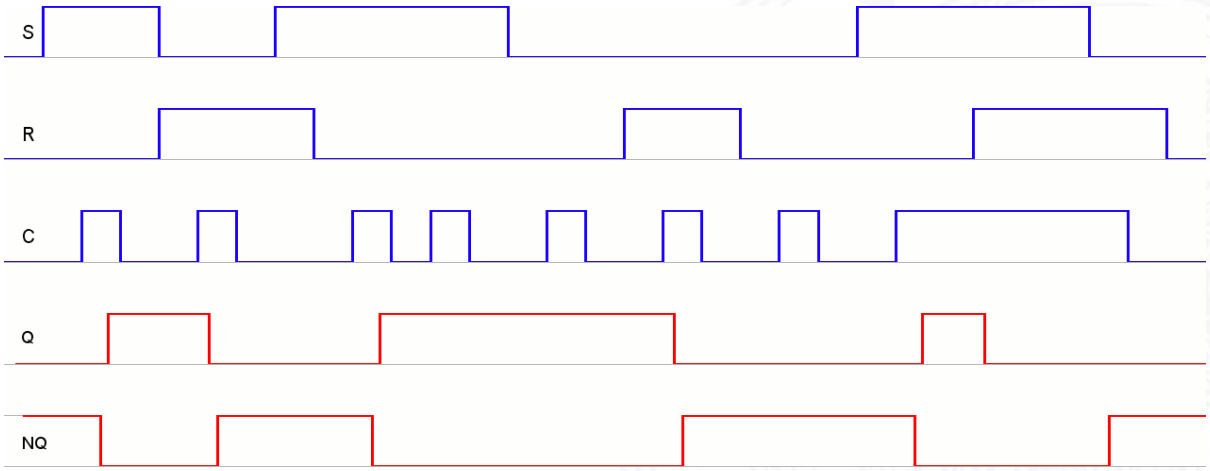
- Struktur



C	S	R	Q	NQ	NOR
0	X	X	Q*	NQ*	(store)
1	0	0	Q*	NQ*	(store)
1	0	1	0	1	
1	1	0	1	0	
1	1	1	0	0	(forbidden)

RS-Flipflop mit Takt (cont.)

- $Q = \overline{(NQ \vee (R \wedge C))}$
  $NQ = \overline{(Q \vee (S \wedge C))}$
- Impulsdiagramm



Pegelgesteuertes D-Flipflop (D-Latch)

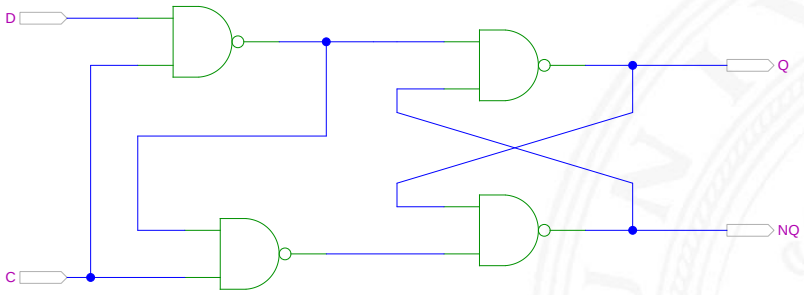
- Takteingang C
- Dateneingang D
- aktueller Zustand Q, Folgezustand Q^+

C	D	Q^+
0	0	Q
0	1	Q
1	0	0
1	1	1

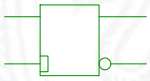
- Wert am Dateneingang wird durchgeleitet, wenn das Taktsignal
  1 ist $\Rightarrow$ *high*-aktiv
  0 ist $\Rightarrow$ *low*-aktiv

Pegelgesteuertes D-Flipflop (D-Latch) (cont.)

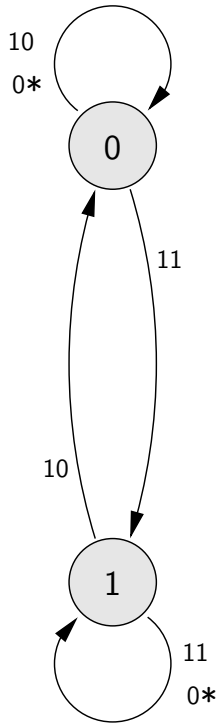
- Realisierung mit getaktetem RS-Flipflop und einem Inverter
  $S = D$, $R = \overline{D}$
- minimierte NAND-Struktur



- Symbol



D-Latch: Zustandsdiagramm und Flusstafel



	Eingabe $[C\ D]$			
	00	01	11	10
Zustand $[Q]$	Folgezustand $[Q^+]$			
0	0	0	1	0
1	1	1	1	0

stabiler Zustand

Flankengesteuertes D-Flipflop

- Takteingang C
- Dateneingang D
- aktueller Zustand Q, Folgezustand Q^+

C	D	Q^+
0	$*$	Q
1	$*$	Q
$\uparrow$	0	0
$\uparrow$	1	1

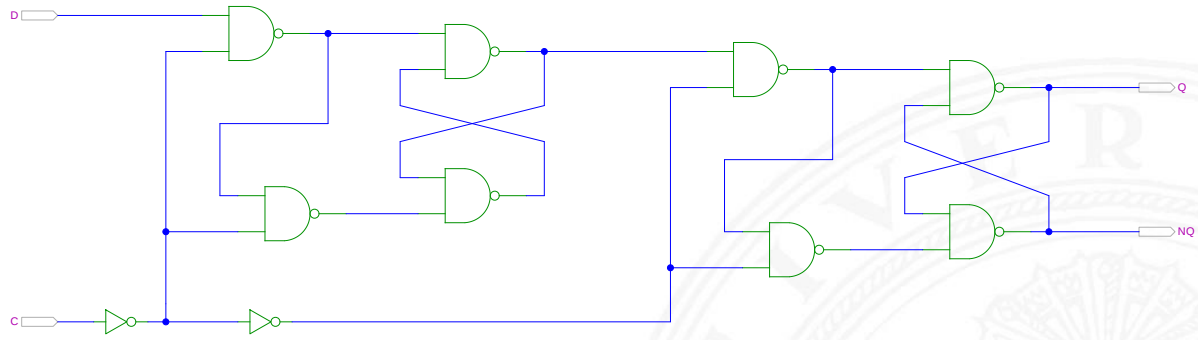
- Wert am Dateneingang wird gespeichert, wenn das Taktsignal sich von 0 auf 1 ändert $\Rightarrow$ Vorderflankensteuerung
  –"– 1 auf 0 ändert $\Rightarrow$ Rückflankensteuerung
- Realisierung als Master-Slave Flipflop oder direkt

Master-Slave D-Flipflop

- zwei kaskadierte D-Latches
- hinteres Latch erhält invertierten Takt
- vorderes „Master“-Latch: low-aktiv (transparent bei $C = 0$)
  hinteres „Slave“-Latch: high-aktiv (transparent bei $C = 1$)
- vorderes Latch speichert bei Wechsel auf $C = 1$
- wenig später (Gatterverzögerung im Inverter der Taktleitung) übernimmt das hintere „Slave“-Latch diesen Wert
- anschließend Input für das Slave-Latch stabil
- Slave-Latch speichert, sobald Takt auf $C = 0$ wechselt

$\Rightarrow$ dies entspricht effektiv einer **Flankensteuerung**:
Wert an D nur relevant, kurz bevor Takt auf $C = 1$ wechselt

Master-Slave D-Flipflop (cont.)



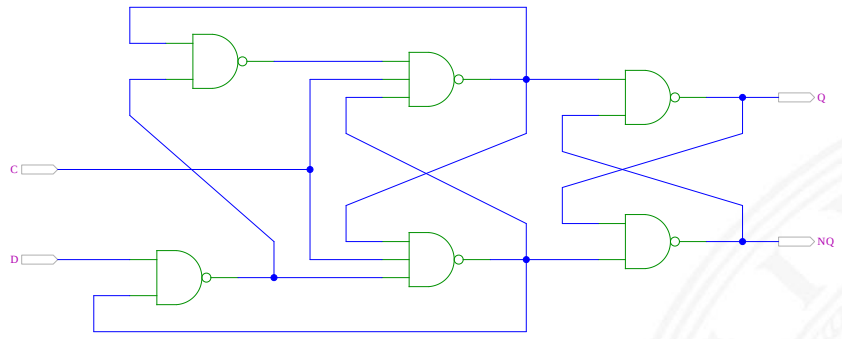
Hades Webdemos: 16-flipflops/20-dlatch/dff

- zwei kaskadierte pegel-gesteuerte D-Latches

C=0 Master aktiv (transparent)
Slave hat (vorherigen) Wert gespeichert

C=1 Master speichert Wert
Slave transparent, leitet Wert von Master weiter

Vorderflanken-gesteuertes D-Flipflop



- Dateneingang D wird nur durch Takt-Vorderflanke ausgewertet
- Gatterlaufzeiten für Funktion essentiell
- Einhalten der Vorlauf- und Haltezeiten vor/nach der Taktflanke (s.u. *Zeitbedingungen*)

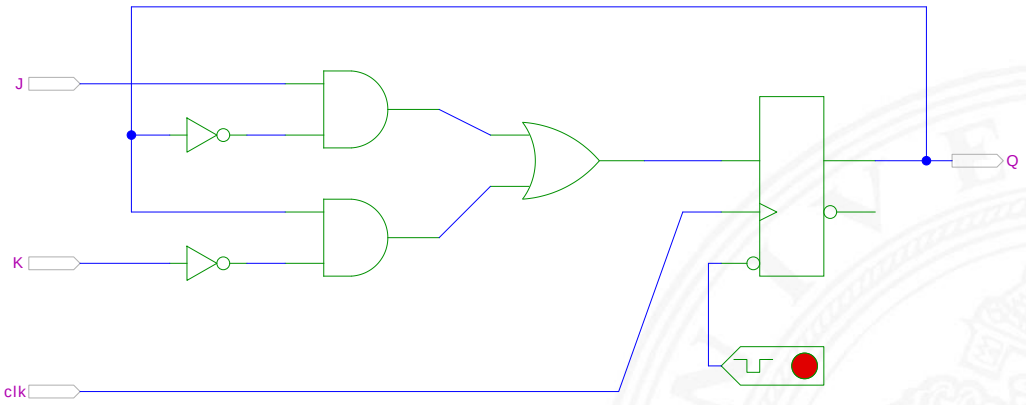
JK-Flipflop

- Takteingang C
- Steuereingänge J („jump“) und K („kill“)
- aktueller Zustand Q, Folgezustand Q^+

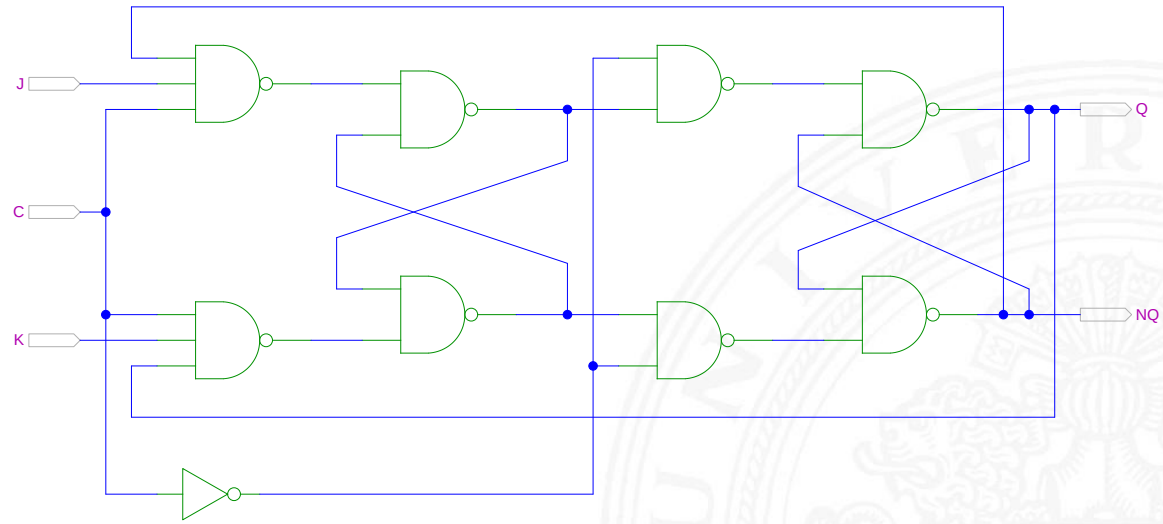
C	J	K	Q^+	Funktion
$*$	$*$	$*$	Q	Wert gespeichert
$\uparrow$	0	0	Q	Wert gespeichert
$\uparrow$	0	1	0	Rücksetzen
$\uparrow$	1	0	1	Setzen
$\uparrow$	1	1	$\overline{Q}$	Invertieren

- universelles Flipflop, sehr flexibel einsetzbar
- in integrierten Schaltungen nur noch selten verwendet

JK-Flipflop: Realisierung mit D-Flipflop



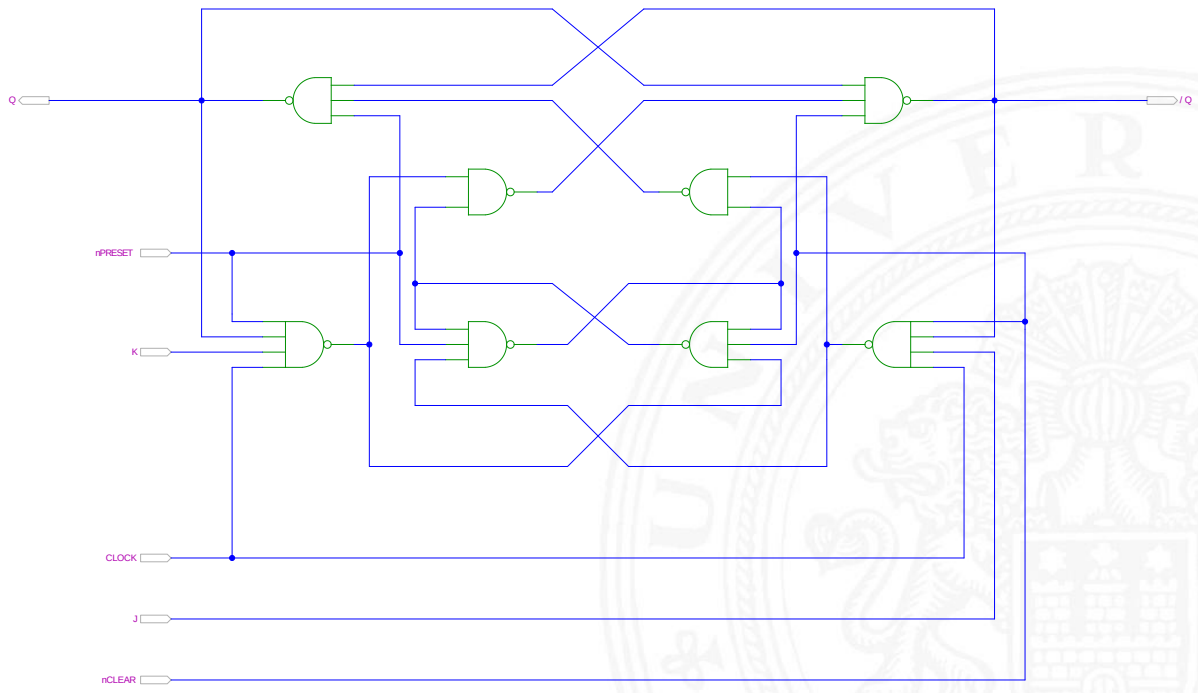
JK-Flipflop: Realisierung als Master-Slave Schaltung



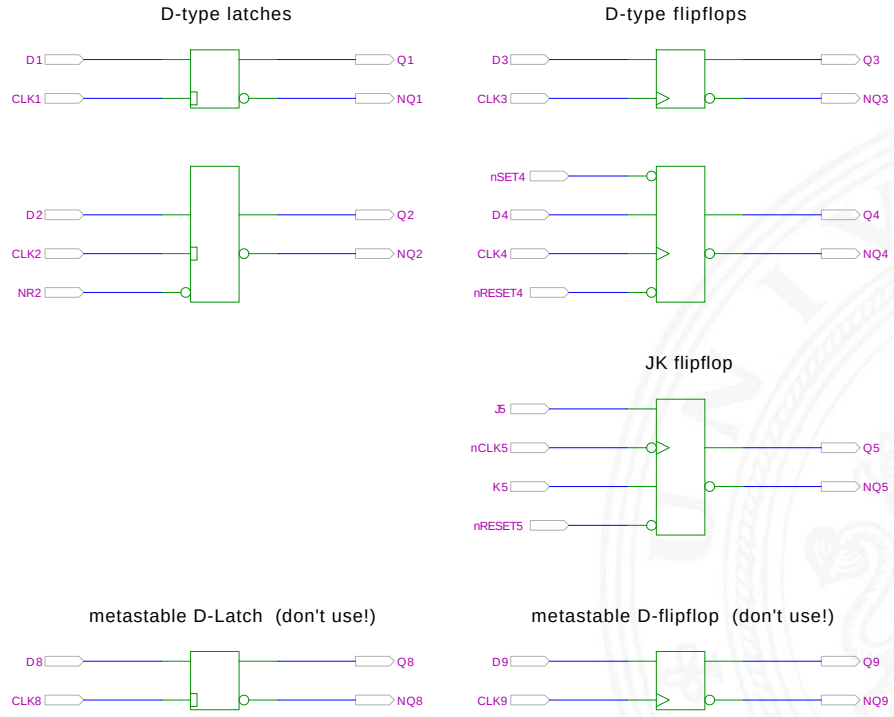
Hades Webdemos: 16-flipflops/40-jkff/jkff

Achtung: Schaltung wegen Rückkopplungen schwer zu initialisieren

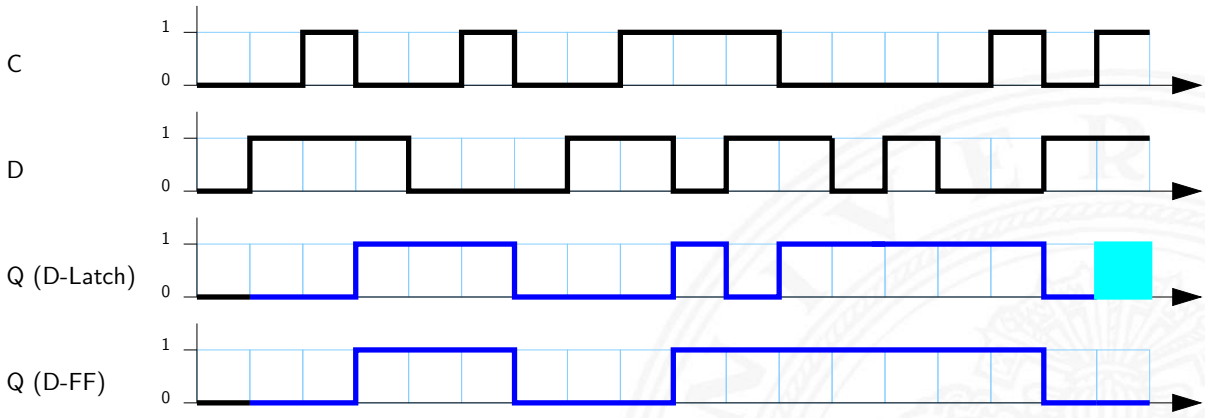
JK-Flipflop: tatsächliche Schaltung im IC 7476



Flipflop-Typen: Komponenten/Symbole in Hades



Flipflop-Typen: Impulsdiagramme



- pegel- und vorderflankengesteuertes Flipflop
- beide Flipflops hier mit jeweils einer Zeiteinheit Verzögerung
- am Ende undefinierte Werte wegen gleichzeitigem Wechsel von C und D (Verletzung der Zeitbedingungen)

Flipflops: Zeitbedingungen

- Flipflops werden entwickelt, um Schaltwerke einfacher entwerfen und betreiben zu können
- Umschalten des Zustandes durch das Taktsignal gesteuert

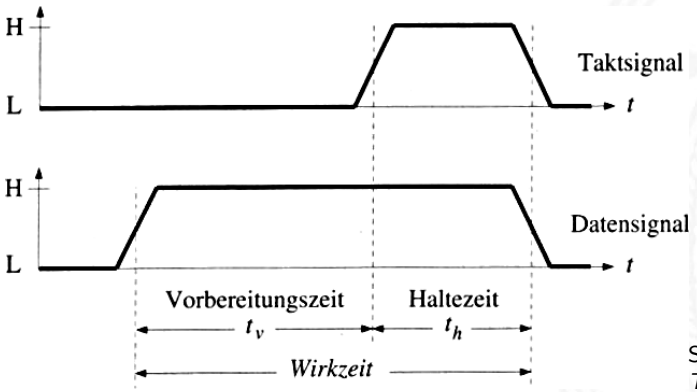
- aber: jedes Flipflop selbst ist ein asynchrones Schaltwerk mit kompliziertem internem Zeitverhalten
- Funktion kann nur garantiert werden, wenn (typ-spezifische) Zeitbedingungen eingehalten werden

$\Rightarrow$ „Vorlauf- und Haltezeiten“ (*setup- / hold-time*)

$\Rightarrow$ Daten- und Takteingänge dürfen sich nie gleichzeitig ändern

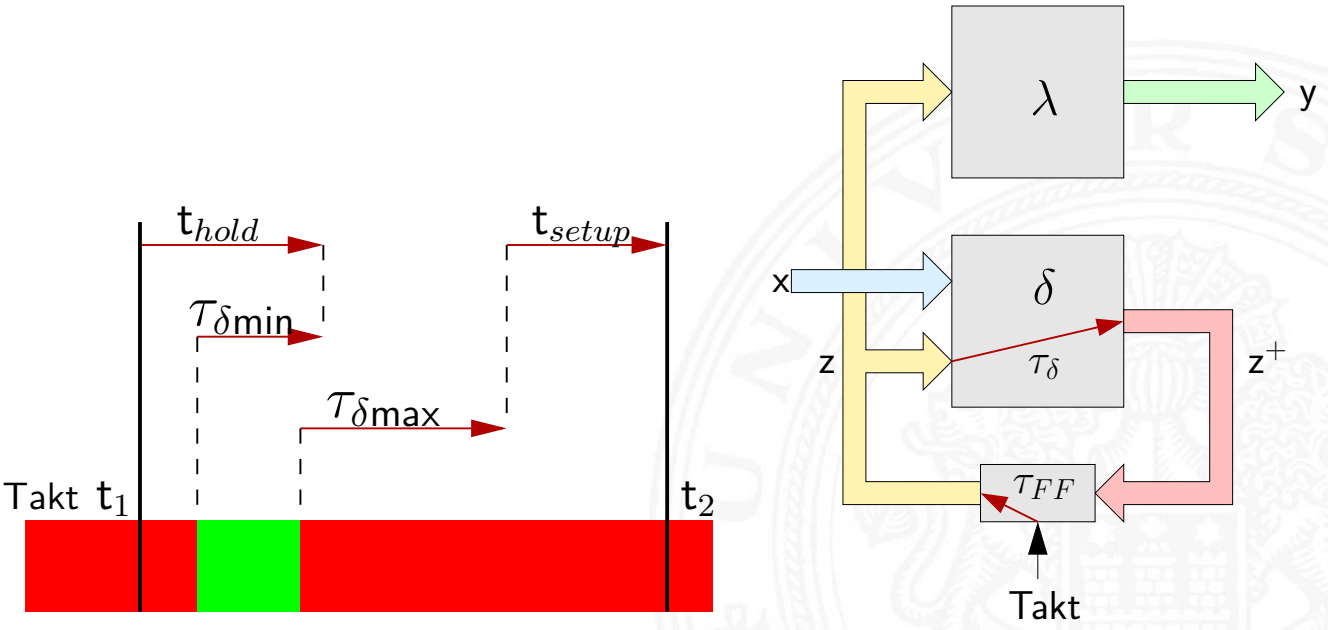
Flipflops: Vorlauf- und Haltezeit

- Vorlaufzeit (oder Vorbereitungszeit, engl. *setup-time*) t_S: Zeitintervall, innerhalb dessen das Datensignal vor dem nächsten Takt bereits stabil anliegen muss
- Haltezeit (*hold-time*) t_h: Zeitintervall, innerhalb dessen das Datensignal nach einem Takt noch stabil anliegen muss



Schiffmann, Schmitz,
Technische Informatik I, Kapitel 5

Zeitbedingungen: Eingangsvektor

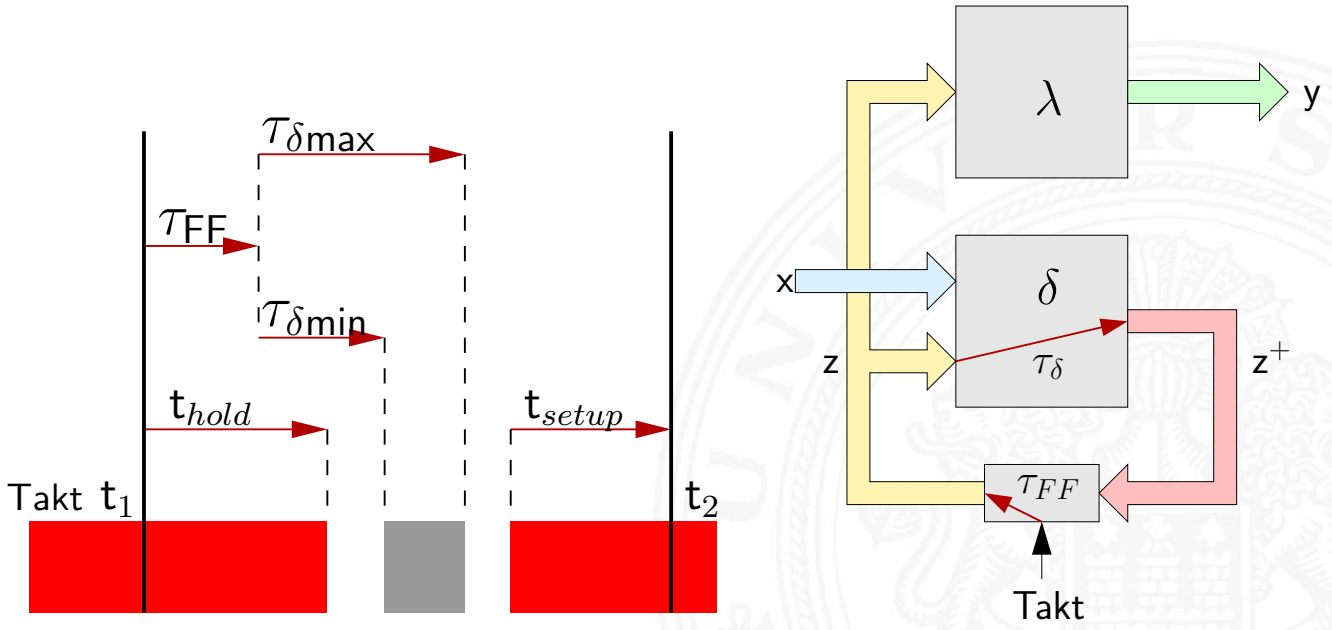


Zeitbedingungen: Eingangsvektor (cont.)

- Änderungen der Eingangswerte x werden beim Durchlaufen von δ mindestens um $\tau_{\delta_{\text{min}}}$, bzw. maximal um $\tau_{\delta_{\text{max}}}$ verzögert
- um die Haltezeit der Zeitglieder einzuhalten, darf x sich nach einem Taktimpuls frühestens zum Zeitpunkt $(t_1 + t_{hold} - \tau_{\delta_{\text{min}}})$ wieder ändern
- um die Vorlaufzeit vor dem nächsten Takt einzuhalten, muss x spätestens zum Zeitpunkt $(t_2 - t_{setup} - \tau_{\delta_{\text{max}}})$ wieder stabil sein

$\Rightarrow$ Änderungen dürfen nur im grün markierten Zeitintervall erfolgen

Zeitbedingungen: interner Zustand



Zeitbedingungen: interner Zustand (cont.)

- zum Zeitpunkt t_1 wird ein Taktimpuls ausgelöst
- nach dem Taktimpuls vergeht die Zeit τ_{FF}, bis die Zeitglieder (Flipflops) ihren aktuellen Eingangswert z^+ übernommen haben und als neuen Zustand z am Ausgang bereitstellen
- die neuen Werte von z laufen durch das δ-Schaltnetz, der schnellste Pfad ist dabei $\tau_{\delta_{min}}$ und der langsamste ist $\tau_{\delta_{max}}$

$\Rightarrow$ innerhalb der Zeitintervalls $\tau_{FF} + \tau_{\delta_{min}}$ bis $\tau_{ff} + \tau_{\delta_{max}}$ ändern sich die Werte des Folgezustands z^+ grauer Bereich

Zeitbedingungen: interner Zustand (cont.)

- die Änderungen dürfen frühestens zum Zeitpunkt $(t_1 + t_{hold})$ beginnen, ansonsten würde Haltezeit verletzt

  ggf. muss $\tau_{\delta_{min}}$ vergrößert werden, um diese Bedingung einhalten zu können (zusätzliche Gatterverzögerungen)
- die Änderungen müssen sich spätestens bis zum Zeitpunkt $(t_2 - t_{setup})$ stabilisiert haben (der Vorbereitungszeit der Flipflops vor dem nächsten Takt)

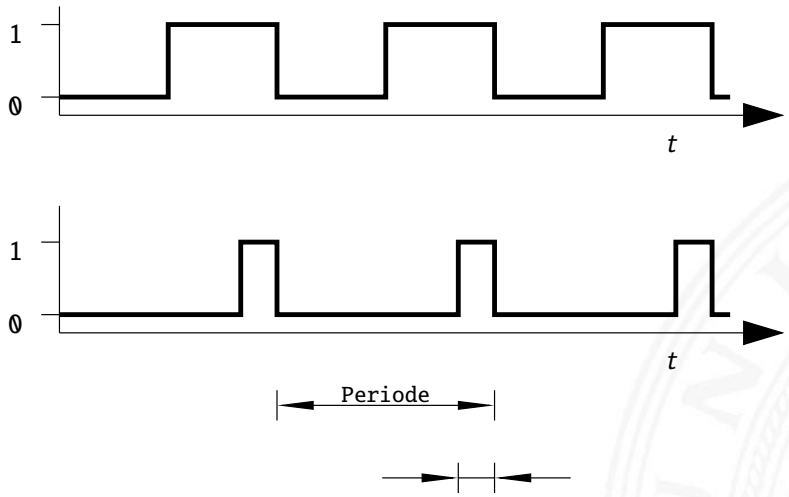
Maximale Taktfrequenz einer Schaltung

- aus obigen Bedingungen ergibt sich sofort die maximal zulässige Taktfrequenz einer Schaltung
- Umformen und Auflösen nach dem Zeitpunkt des nächsten Takts ergibt zwei Bedingungen

$$\Delta t \geq (\tau_{FF} + \tau_{\delta_{max}} + \tau_{setup}) \quad \text{und}$$
$$\Delta t \geq (\tau_{hold} + \tau_{setup})$$

- falls diese Bedingung verletzt wird („Übertakten“), kann es (datenabhängig) zu Fehlfunktionen kommen

Taktsignal: Prinzip

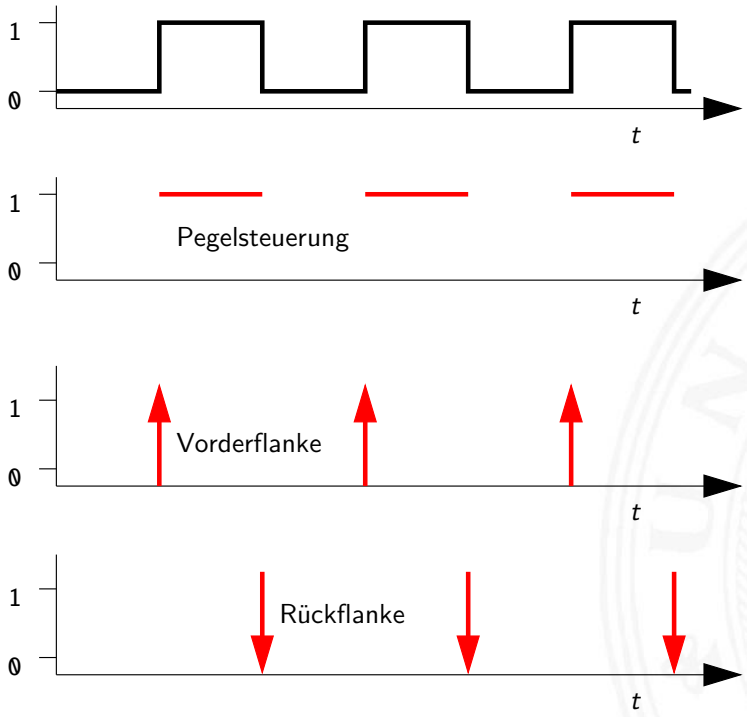


- periodisches digitales Signal, Frequenz f bzw. Periode τ
- oft symmetrisch
- asymmetrisch für Zweiphasentakt (s.u.)

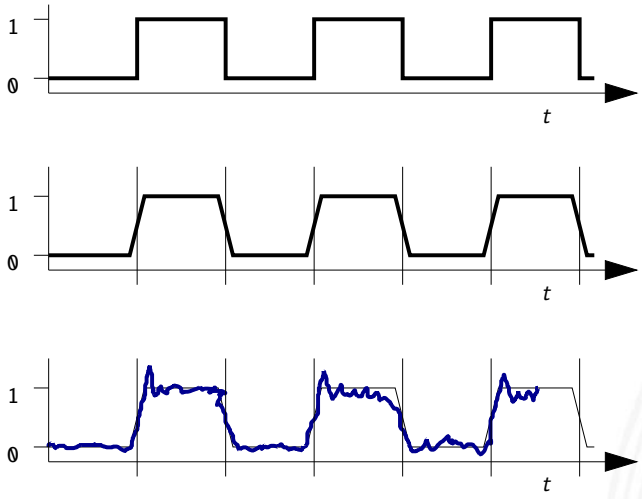
Taktsignal: Varianten

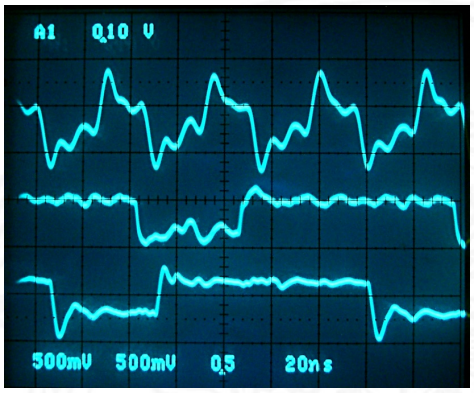
- **Pegelsteuerung**: Schaltung reagiert, während das Taktsignal den Wert 1 (bzw. 0) aufweist
- **Flankensteuerung**: Schaltung reagiert nur, während das Taktsignal seinen Wert wechselt
  - Vorderflankensteuerung: Wechsel von 0 nach 1
  - Rückflankensteuerung: –"– von 1 nach 0
- Zwei- und Mehrphasentakte

Taktsignal: Varianten (cont.)



Taktsignal: Prinzip und Realität

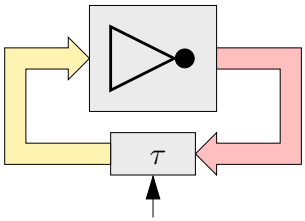




- Werteverläufe in realen Schaltungen stark gestört
- Überschwingen/Übersprechen benachbarter Signale
- Flankensteilheit nicht garantiert (bei starker Belastung)
  ggf. besondere Gatter („Schmitt-Trigger“)

Problem mit Pegelsteuerung

- während des aktiven Taktpegels werden Eingangswerte direkt übernommen
- falls invertierende Rückkopplungspfade in δ vorliegen, kommt es dann zu instabilen Zuständen (Oszillationen)



- einzelne pegelgesteuerte Zeitglieder (D-Latches) garantieren keine stabilen Zustände

$\Rightarrow$ Verwendung von je zwei pegelgesteuerten Zeitgliedern und Einsatz von Zweiphasentakt oder

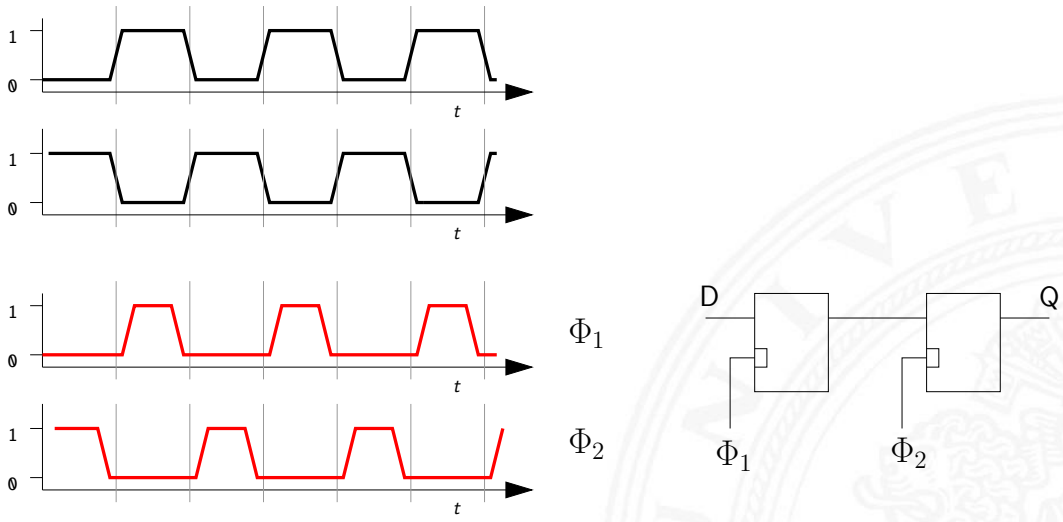
$\Rightarrow$ Verwendung flankengesteuerter D-Flipflops

Zweiphasentakt

- pegelgesteuertes D-Latch ist bei aktivem Takt *transparent*
- rück-gekoppelte Werte werden sofort wieder durchgelassen
- Oszillation bei invertierten Rückkopplungen

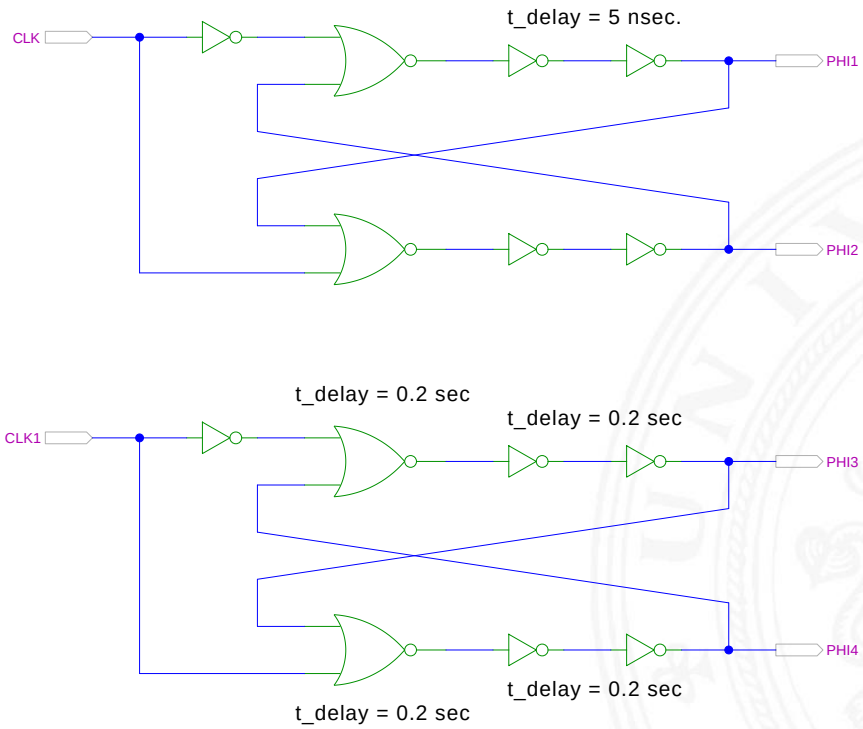
- Reihenschaltung aus jeweils zwei D-Latches
- zwei separate Takte Φ_1 und Φ_2
  - bei Takt Φ_1 übernimmt vorderes Flipflop den Wert erst bei Takt Φ_2 übernimmt hinteres Flipflop
  - vergleichbar Master-Slave Prinzip bei D-FF aus Latches

Zweiphasentakt (cont.)

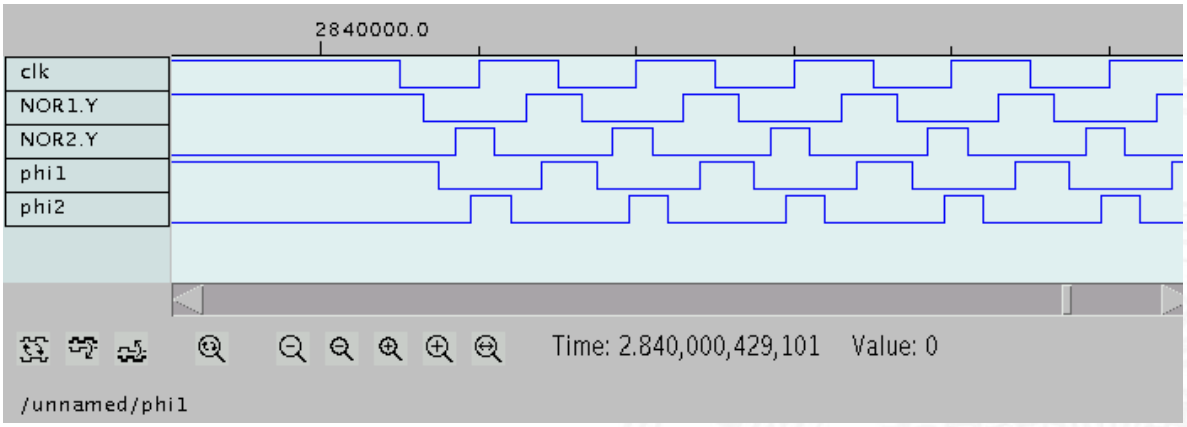


- nicht überlappender Takt mit Phasen Φ_1 und Φ_2
- vorderes D-Latch übernimmt Eingangswert D während Φ_1 bei Φ_2 übernimmt das hintere D-Latch und liefert Q

Zweiphasentakt: Erzeugung



Zweiphasentakt: Erzeugung (cont.)



- Verzögerungen geeignet wählen
- Eins-Phasen der beiden Takte c_1 und c_2 sauber getrennt

$\Rightarrow$ nicht-überlappende 2-Phasen-Taktimpulse zur Ansteuerung von Schaltungen mit 2-Phasen-Taktung

Beschreibung von Schaltwerken

- viele verschiedene Möglichkeiten
- graphisch oder textuell

- algebraische Formeln/Gleichungen
- Flusstafel und Ausgangstafel

- Zustandsdiagramm
- State-Charts (hierarchische Zustandsdiagramme)

- Programme (Hardwarebeschreibungssprachen)

Flusstafel und Ausgangstafel

- entspricht der Funktionstabelle von Schaltnetzen

- **Flusstafel**: Tabelle für die Folgezustände als Funktion des aktuellen Zustands und der Eingabewerte

= beschreibt das δ-Schaltnetz

- **Ausgangstafel**: Tabelle für die Ausgabewerte als Funktion des aktuellen Zustands (und der Eingabewerte [Mealy-Modell])

= beschreibt das λ-Schaltnetz

Beispiel: Ampel

- vier Zustände: {rot, rot-gelb, grün, gelb}
- Codierung beispielsweise als 2-bit Vektor (z_1, z_0)

- Flusstafel

Zustand	Codierung		Folgezustand	
	z_1	z_0	z_1^+	z_0^+
rot	0	0	0	1
rot-gelb	0	1	1	0
grün	1	0	1	1
gelb	1	1	0	0

Beispiel: Ampel (cont.)

- Ausgangstafel

Zustand	Codierung		Ausgänge		
	z_1	z_0	*rt*	*ge*	*gr*
rot	0	0	1	0	0
rot-gelb	0	1	1	1	0
grün	1	0	0	0	1
gelb	1	1	0	1	0

- Funktionstabelle für drei Schaltfunktionen
- Minimierung z.B. mit KV-Diagrammen

Zustandsdiagramm

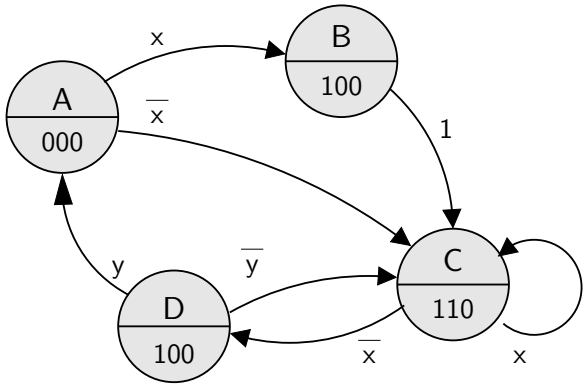
- **Zustandsdiagramm**: Grafische Darstellung eines Schaltwerks
- je ein Knoten für jeden Zustand
- je eine Kante für jeden möglichen Übergang

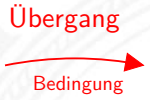
- Knoten werden passend benannt
- Kanten werden mit den Eingabemustern gekennzeichnet, bei denen der betreffende Übergang auftritt

- Moore-Schaltwerke: Ausgabe wird zusammen mit dem Namen im Knoten notiert
- Mealy-Schaltwerke: Ausgabe hängt vom Input ab und wird an den Kanten notiert

siehe auch en.wikipedia.org/wiki/State_diagram

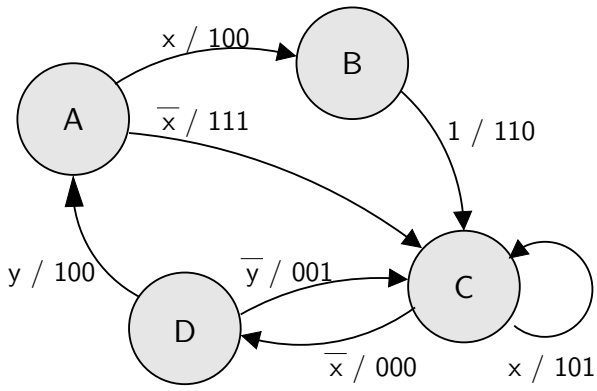
Zustandsdiagramm: Moore-Automat

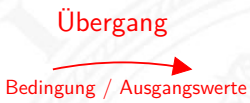




- Ausgangswerte hängen nur vom Zustand ab
- können also im jeweiligen Knoten notiert werden
- Übergänge werden als Pfeile mit der Eingangsbelegung notiert, die den Übergang aktiviert
- ggf. Startzustand markieren (z.B. Segment, doppelter Kreis)

Zustandsdiagramm: Mealy-Automat





- Ausgangswerte hängen nicht nur vom Zustand sondern auch von den Eingabewerten ab
- Ausgangswerte an den zugehörigen Kanten notieren
- übliche Notation: *Eingangsbelegung / Ausgangswerte*

„State-Charts“

- hierarchische Version von Zustandsdiagrammen
- Knoten repräsentieren entweder einen Zustand
- oder einen eigenen (Unter-) Automaten

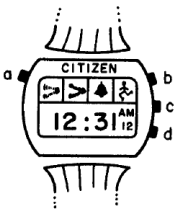
- beliebte Spezifikation für komplexe Automaten,
  Embedded Systems, Kommunikationssysteme, etc.

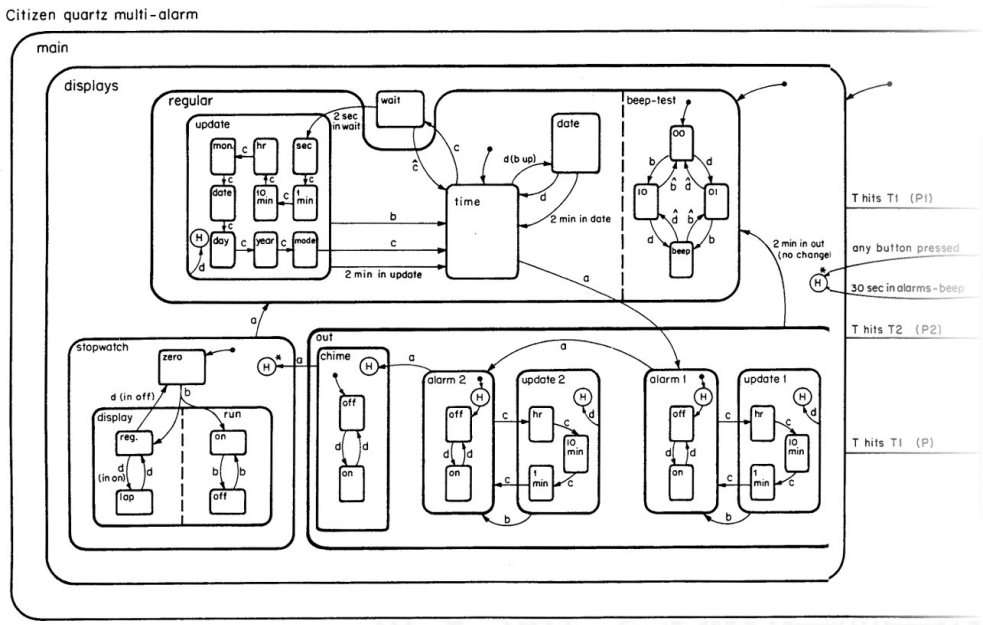
- David Harel, *Statecharts – A visual formalism for complex systems*, CS84-05, Department of Applied Mathematics, The Weizmann Institute of Science, 1984

  www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf

„State-Charts“ (cont.)

- Beispiel Digitaluhr





Hardwarebeschreibungssprachen

- Beschreibung eines Schaltwerks als Programm:

- normale Hochsprachen C, Java
- spezielle Bibliotheken für normale Sprachen SystemC, Hades
- spezielle Hardwarebeschreibungssprachen Verilog, VHDL

- Hardwarebeschreibungssprachen unterstützen Modellierung paralleler Abläufe und des Zeitverhaltens einer Schaltung
- wird hier nicht vertieft
- lediglich zwei Beispiele: D-Flipflop in Verilog und VHDL

D-Flipflop in Verilog

```
module dff (clock, reset, din, dout);
input clock, reset, din;
output dout;

reg dout;

  always @(posedge clock or reset)
  begin
    if (reset)
      dout = 1'b0;
    else
      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
port (  CLOCK    : in  std_logic;
        RESET    : in  std_logic;
        DIN      : in  std_logic;
        DOUT     : out std_logic);
end entity DFF;

architecture BEHAV of DFF is
begin
  DFF_P: process (RESET, CLOCK) is
  begin
    if RESET = '1' then
        DOUT <= '0';
    elsif rising_edge(CLOCK) then
        DOUT <= DIN;
    end if;
  end process DFF_P;
end architecture BEHAV;
```


Entwurf von Schaltwerken: sechs Schritte

1. Spezifikation (textuell oder graphisch, z.B. Zustandsdiagramm)
2. Aufstellen einer formalen Übergangstabelle
3. Reduktion der Zahl der Zustände
4. Wahl der Zustandscodierung und Aufstellen der Übergangstabelle
5. Minimierung der Schaltnetze
6. Überprüfung des realisierten Schaltwerks

ggf. mehrere Iterationen

Entwurf von Schaltwerken: Zustandscodierung

Vielfalt möglicher Codierungen

- binäre Codierung: minimale Anzahl der Zustände
- einschrittige Codes
- one-hot Codierung: ein aktives Flipflop pro Zustand
- applikationsspezifische Zwischenformen

- es gibt Entwurfsprogramme zur Automatisierung
- gemeinsame Minimierung des Realisierungsaufwands von Ausgangsfunktion, Übergangsfunktion und Speichergliedern

Entwurf von Schaltwerken: Probleme

Entwurf ausgehend von Funktionstabellen problemlos

- alle Eingangsbelegungen und Zustände werden berücksichtigt
- don't-care Terme können berücksichtigt werden

zwei typische Fehler bei Entwurf ausgehend vom Zustandsdiagramm

- mehrere aktive Übergänge bei bestimmten Eingangsbelegungen
  $\Rightarrow$ Widerspruch
- keine Übergänge bei bestimmten Eingangsbelegungen
  $\Rightarrow$ Vollständigkeit

Überprüfung der Vollständigkeit

p Zustände, Zustandsdiagramm mit Kanten $h_{ij}(x)$:
Übergang von Zustand i nach Zustand j unter Belegung x

- für jeden Zustand überprüfen:
  kommen alle (spezifizierten) Eingangsbelegungen auch tatsächlich in Kanten vor?

$$\forall i: \quad \bigvee_{j=0}^{2^p-1} h_{ij}(x) = 1$$

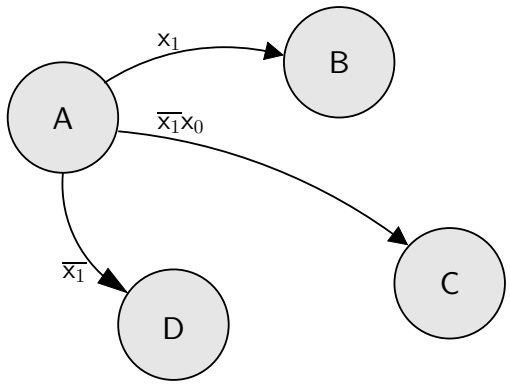
Überprüfung der Widerspruchsfreiheit

p Zustände, Zustandsdiagramm mit Kanten $h_{ij}(x)$:
Übergang von Zustand i nach Zustand j unter Belegung x

- für jeden Zustand überprüfen:
  kommen alle (spezifizierten) Eingangsbelegungen nur einmal vor?

$$\forall i: \quad \bigvee_{j,k=0,\ j\neq k}^{2^p-1} \left(h_{ij}(x) \wedge h_{ik}(x)\right) = 0$$

Vollständigkeit und Widerspruchsfreiheit: Beispiel



- Zustand A, Vollständigkeit: $x_1 \vee \overline{x_1}x_0 \vee \overline{x_1} = 1$ vollständig
- Zustand A, Widerspruchsfreiheit: alle Paare testen

  $x_1 \wedge \overline{x_1}x_0 = 0$ ok

  $x_1 \wedge \overline{x_1} = 0$ ok

  $\overline{x_1}x_0 \wedge \overline{x_1} \neq 0$ für $x_1 = 0$ und $x_0 = 1$ beide Übergänge aktiv

Entwurf von Schaltwerken: Beispiele

- Verkehrsampel
  - drei Varianten mit unterschiedlicher Zustandscodierung
- Zählschaltungen
  - einfacher Zähler, Zähler mit Enable (bzw. Stop),
  - Vorwärts-Rückwärts-Zähler, Realisierung mit JK-Flipflops und D-Flipflops
- Digitaluhr
  - BCD-Zähler
- ...

Entwurf von Schaltwerken: Ampel

Beispiel Verkehrsampel:

- drei Ausgänge: {rot, gelb, grün}
- vier Zustände: {rot, rot-gelb, grün, gelb}
- zunächst kein Eingang, feste Zustandsfolge wie oben

- Aufstellen des Zustandsdiagramms
- Wahl der Zustandscodierung
- Aufstellen der Tafeln für δ- und λ-Schaltnetz
- anschließend Minimierung der Schaltnetze
- Realisierung (je 1 D-Flipflop pro Zustandsbit) und Test

Entwurf von Schaltwerken: Ampel – Variante 1

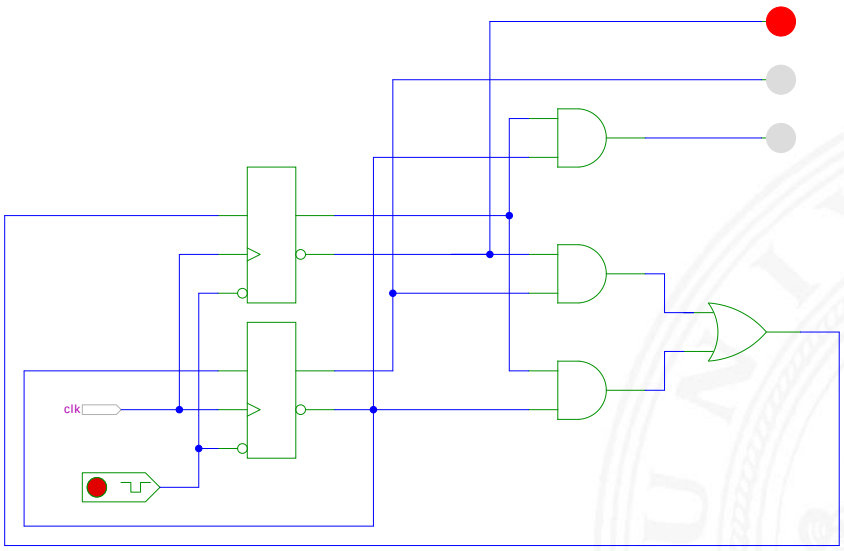
- vier Zustände, Codierung als 2-bit Vektor (z_1, z_0)
- Fluss- und Ausgangstafel für binäre Zustandscodierung

Zustand	Codierung		Folgezustand		Ausgänge		
	z_1	z_0	z_1^+	z_0^+	rt	ge	gr
rot	0	0	0	1	1	0	0
rot-gelb	0	1	1	0	1	1	0
grün	1	0	1	1	0	0	1
gelb	1	1	0	0	0	1	0

- resultierende Schaltnetze

$$z_1^+ = (z_1 \wedge \overline{z_0}) \vee (\overline{z_1} \wedge z_0) = z_1 \oplus z_0$$
$$z_0^+ = \overline{z_0}$$
$$rt = \overline{z_1}$$
$$ge = z_0$$
$$gr = (z_1 \wedge \overline{z_0})$$

Entwurf von Schaltwerken: Ampel – Variante 1 (cont.)



Hades Webdemos: 18-fsm/10-trafficLight/ampel_41

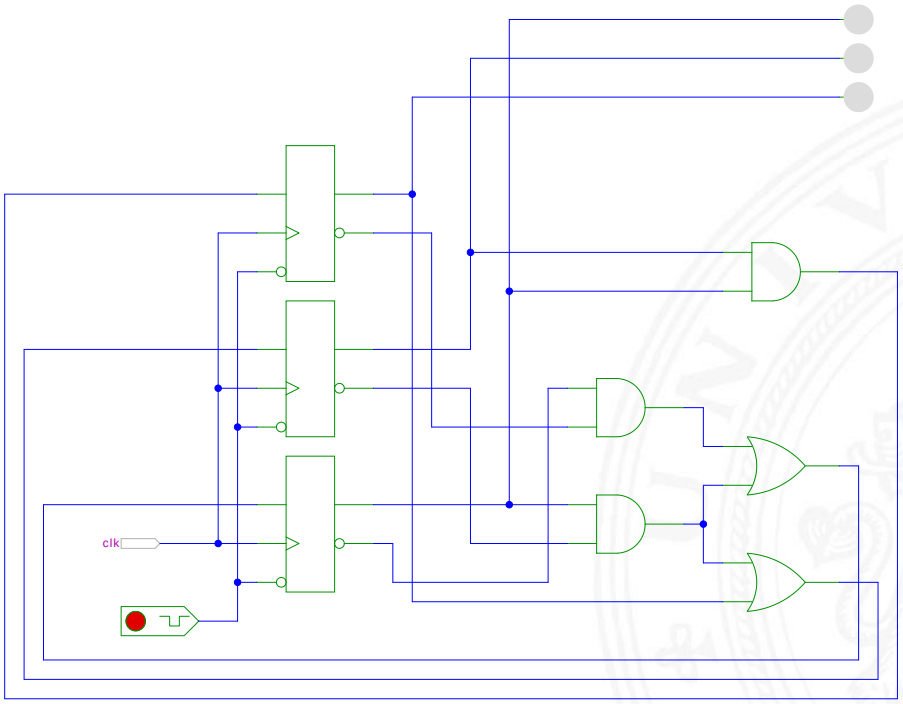
Entwurf von Schaltwerken: Ampel – Variante 2

- vier Zustände, Codierung als 3-bit Vektor (z_2, z_1, z_0)
- Zustandsbits korrespondieren mit den aktiven Lampen: $z_2^+ = gr$, $z_1^+ = ge$ und $z_0^+ = rt$

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
- dafür wird die Ausgangsfunktion λ minimal (leer)

Entwurf von Schaltwerken: Ampel – Variante 2 (cont.)



Hades Webdemos:
18-fsm/10-trafficLight/ampel_42

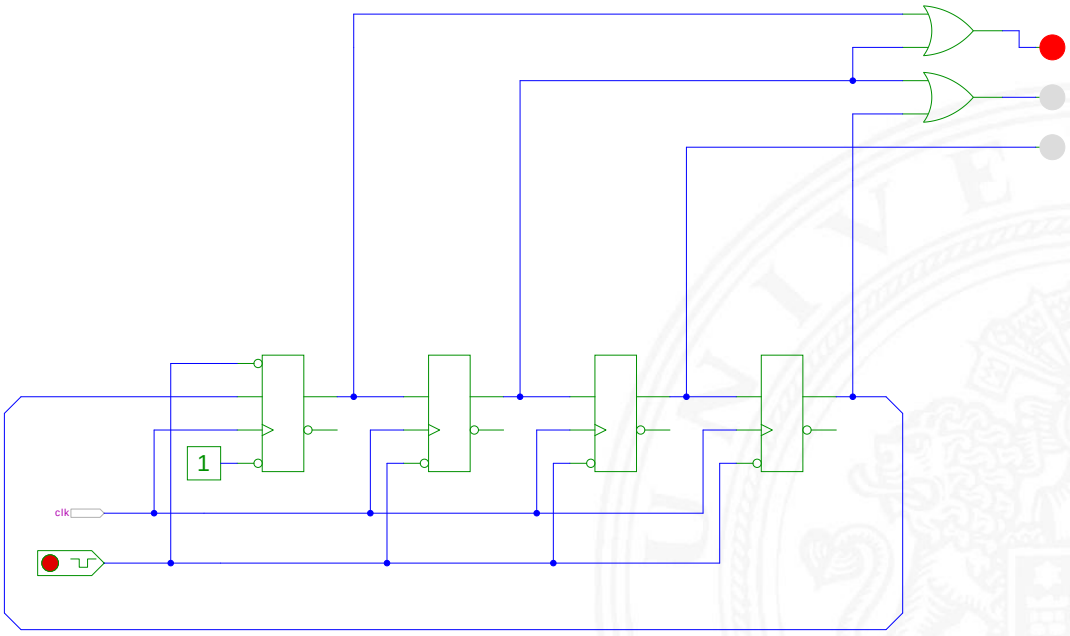
Entwurf von Schaltwerken: Ampel – Variante 3

- 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 (Automat sehr schnell)
- Ausgangsfunktion λ sehr einfach

Entwurf von Schaltwerken: Ampel – Variante 3 (cont.)



Hades Webdemos: 18-fsm/10-trafficLight/ampel_44

Entwurf von Schaltwerken: Ampel – Zusammenfassung

- viele Möglichkeiten der Zustandscodierung

- Dualcode: minimale Anzahl der Zustände
- applikations-spezifische Codierungen
- One-Hot Encoding: viele Zustände, einfache Schaltnetze
- ...

- Kosten/Performance des Schaltwerks abhängig von Codierung
- Heuristiken zur Suche nach (relativem) Optimum

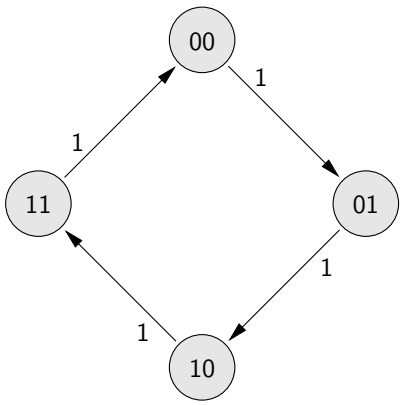
Zählschaltungen

- diverse Beispiele für Zählschaltungen

- Zustandsdiagramme und Flusstafeln
- Schaltbilder

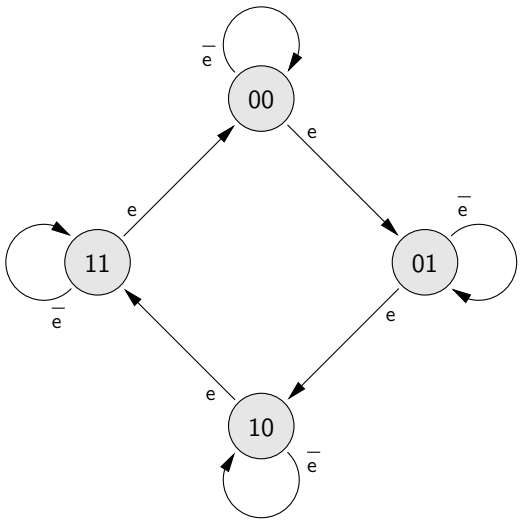
- *n*-bit Vorwärtszähler
- *n*-bit Zähler mit Stop und/oder Reset
- Vorwärts/Rückwärtszähler
- synchrone und asynchrone Zähler
- Beispiel: Digitaluhr (BCD-Zähler)

2-bit Zähler: Zustandsdiagramm



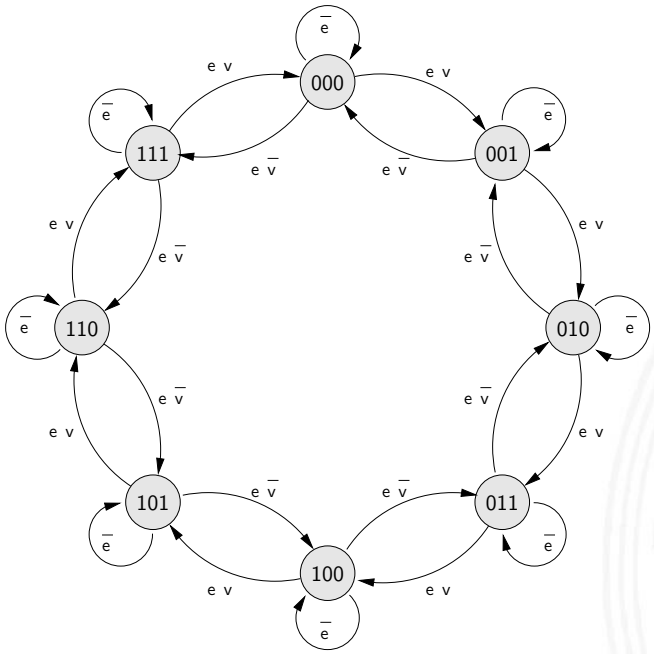
- Zähler als „trivialer“ endlicher Automat

2-bit Zähler mit Enable: Zustandsdiagramm und Flusstafel



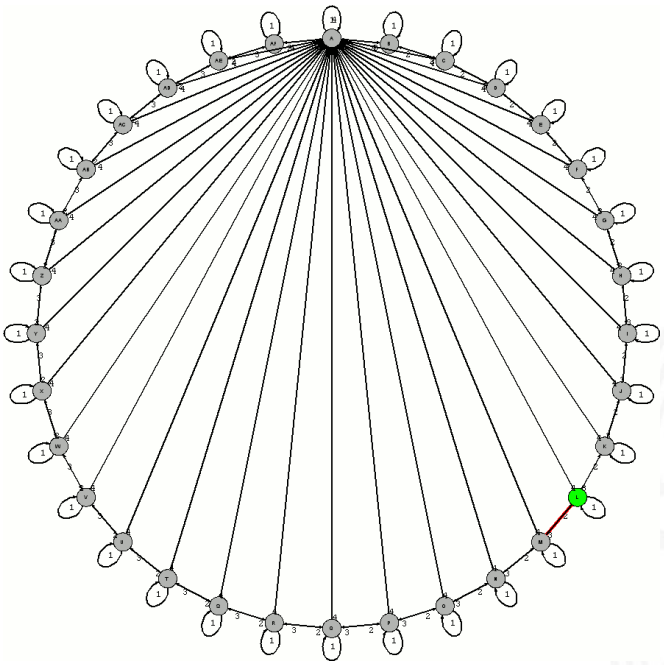
	e	$\overline{e}$
Zustand	Folgezustand	
00	01	00
01	10	01
10	11	10
11	00	11

3-bit Zähler mit Enable, Vor-/Rückwärts



	$e\,v$	$e\,\overline{v}$	$\overline{e}\,*$
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

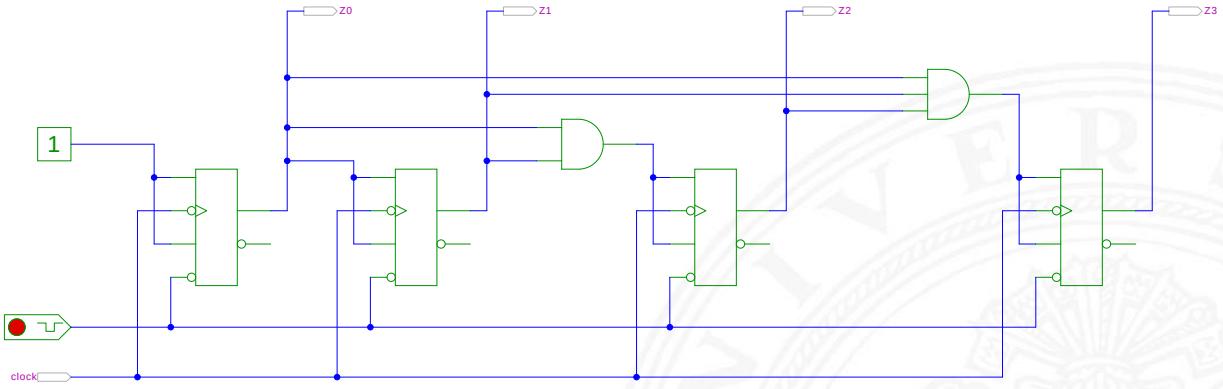
5-bit Zähler mit Reset: Zustandsdiagramm und Flusstafel



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

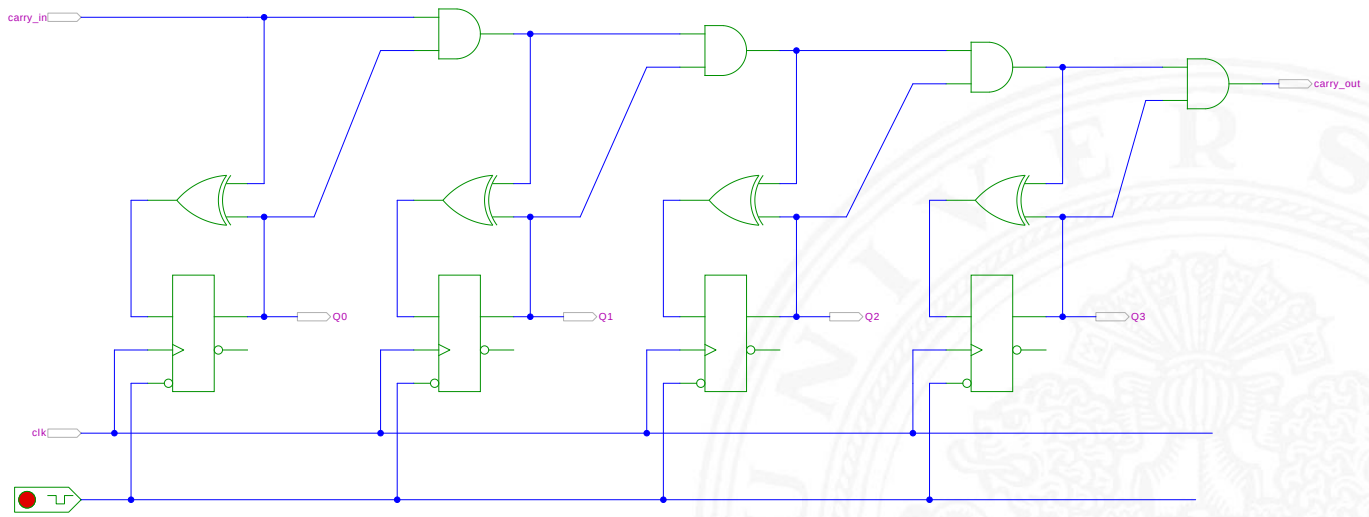
Eingabe 1: stop, 2: zählen, 3: rückwärts zählen, 4: Reset nach A

4-bit Binärzähler mit JK-Flipflops



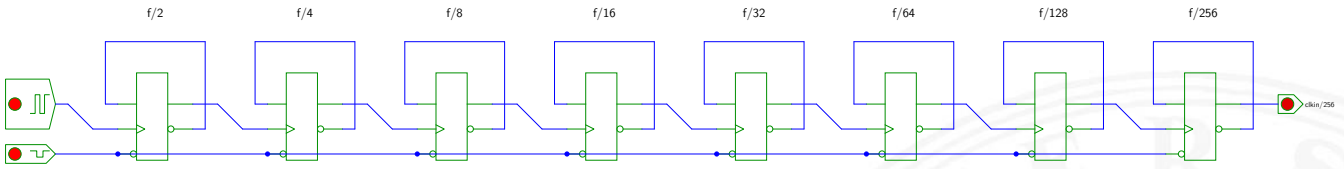
- $J_0 = K_0 = 1$: Ausgang z_0 wechselt bei jedem Takt
- $J_i = K_i = (z_0 z_1 \ldots z_{i-1})$: Ausgang z_i wechselt, wenn alle niedrigeren Stufen 1 sind

4-bit Binärzähler mit D-Flipflops (kaskadierbar)



- $D_0 = Q_0 \oplus c_{in}$ wechselt bei Takt, wenn c_{in} aktiv ist
- $D_i = Q_i \oplus (c_{in} Q_0 Q_1 \ldots Q_{i-1})$ wechselt, wenn alle niedrigeren Stufen und Carry-in c_{in} 1 sind

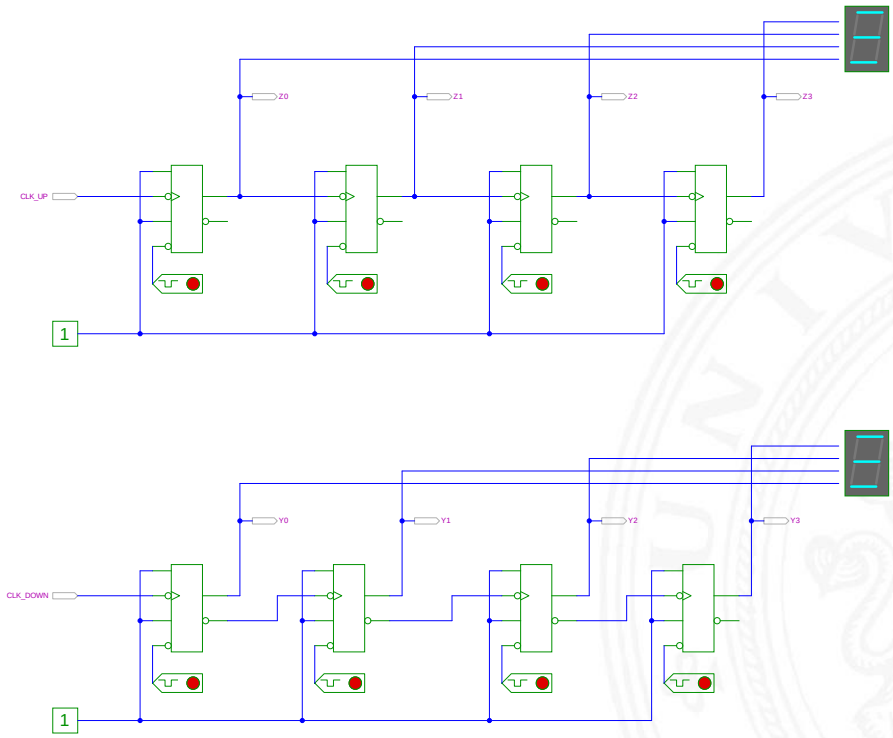
Asynchroner n-bit Zähler/Teiler mit D-Flipflops



- $D_i = \overline{Q}_i$: jedes Flipflop wechselt bei seinem Taktimpuls
- Takteingang C_0 treibt nur das vorderste Flipflop
- $C_i = Q_{i-1}$: Ausgang der Vorgängerstufe als Takt von Stufe i

- erstes Flipflop wechselt bei jedem Takt $\Rightarrow$ Zählrate $C_0/2$
  zweites Flipflop bei jedem zweiten Takt $\Rightarrow$ Zählrate $C_0/4$
  n-tes Flipflop bei jedem n-ten Takt $\Rightarrow$ Zählrate $C_0/2^n$
- sehr hohe maximale Taktrate
- – Achtung: Flipflops schalten nacheinander, nicht gleichzeitig

Asynchrone 4-bit Vorwärts- und Rückwärtszähler

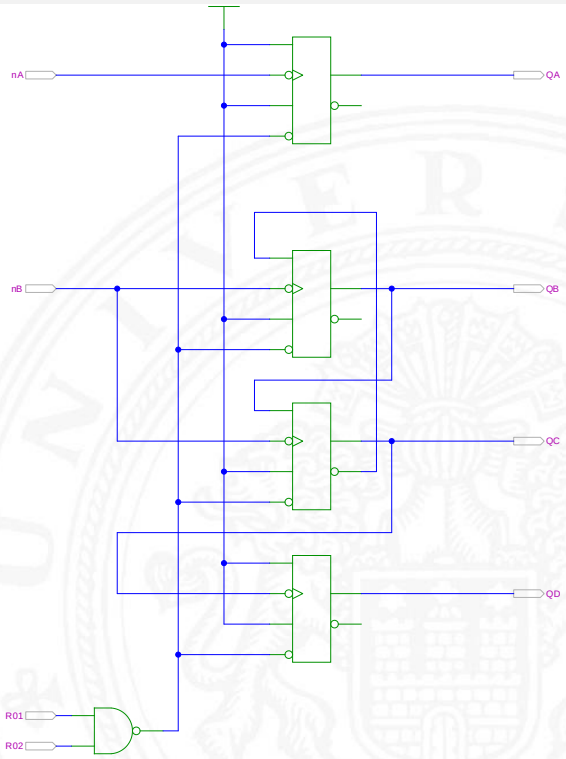


4-bit 1:2, 1:6, 1:12-Teiler mit JK-Flipflops: IC 7492

- vier JK-Flipflops
- zwei Reseteingänge
- zwei Takteingänge

- Stufe 0 separat (1:2)

- Stufen 1...3 kaskadiert (1:6)
- Zustandsfolge
  {000, 001, 010, 100, 101, 110}

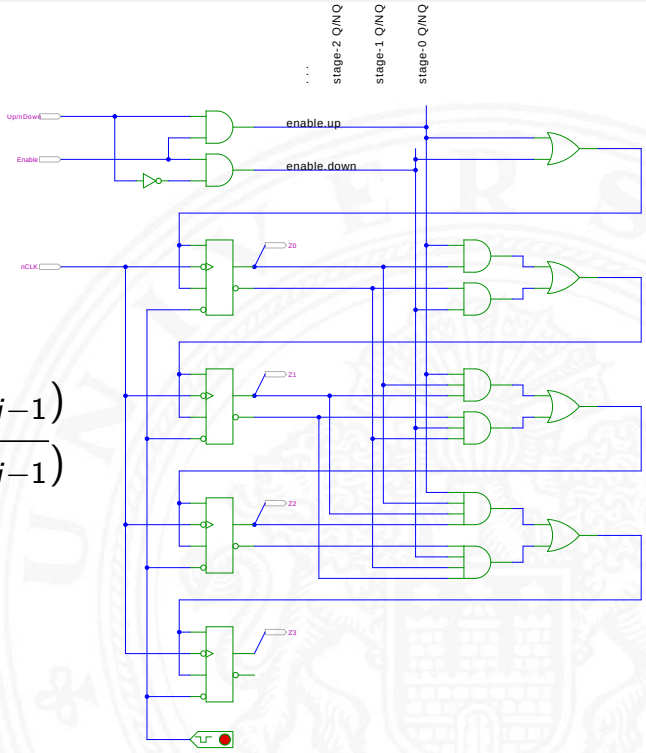


4-bit Vorwärts-Rückwärtszähler mit JK-Flipflops

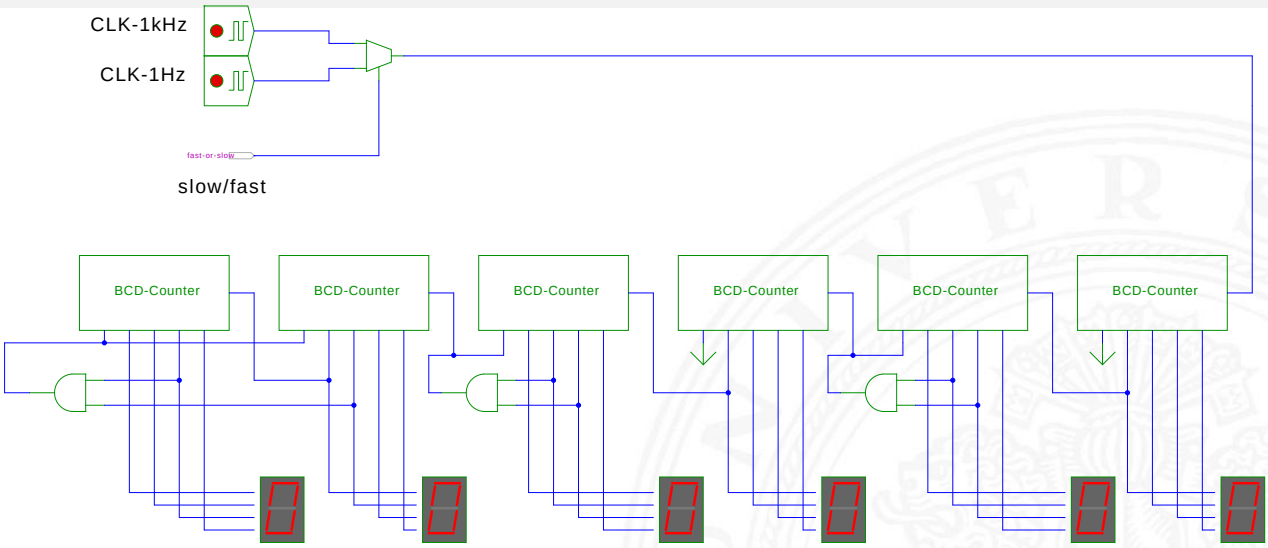
- Eingänge: nClk
  Enable
  Up/nDown
- Umschaltung der *Carry-Chain*

  up: $J_i = K_i = (E\ Q_0\ Q_1 \ldots Q_{i-1})$

  down: $J_i = K_i = (E\ \overline{Q_0}\ \overline{Q_1} \ldots \overline{Q_{i-1}})$



Digitaluhr mit BCD-Zählern



- Stunden Minuten Sekunden (hh:mm:ss)
- async. BCD-Zähler mit Takt (rechts) und Reset (links unten)
- Übertrag 1er- auf 10er-Stelle jeweils beim Übergang $9 \rightarrow 0$
- Übertrag und Reset der Zehner beim Auftreten des Wertes 6

Funkgesteuerte DCF 77 Uhr

- Beispiel für eine komplexe Schaltung aus mehreren einfachen Komponenten
- mehrere gekoppelte Automaten, bzw. Zähler

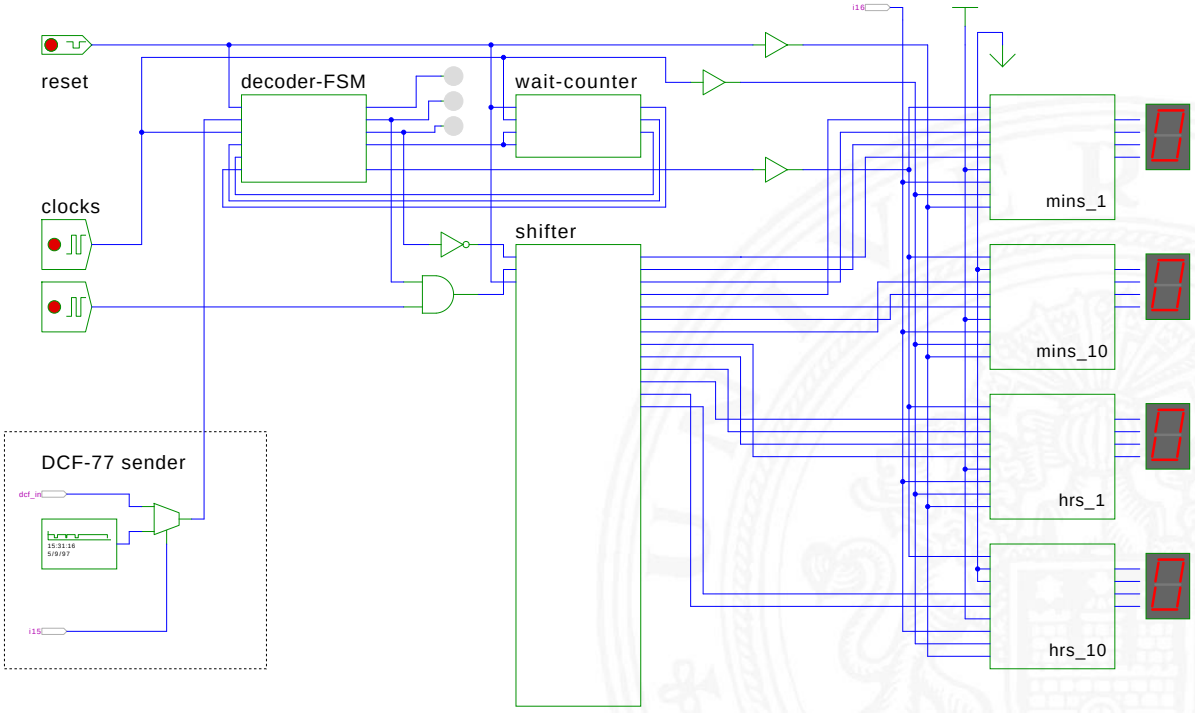
- DCF 77 Zeitsignal
  - Langwelle 77,5 KHz
  - Sender nahe Frankfurt
  - ganz Deutschland abgedeckt
- pro Sekunde wird ein Bit übertragen
  - Puls mit abgesenktem Signalpegel: „Amplitudenmodulation“
  - Pulslänge: 100 ms entspricht Null, 200 ms entspricht Eins
  - Pulsbeginn ist Sekundenbeginn

Funkgesteuerte DCF 77 Uhr (cont.)

- pro Minute werden 59 Bits übertragen
  - Uhrzeit hh:mm (implizit Sekunden), MEZ/MESZ
  - Datum dd:mm:yy, Wochentag
  - Parität
  - fehlender 60ster Puls markiert Ende einer Minute
- Decodierung der Bits nach DCF 77 Protokoll mit entsprechend entworfenem Schaltwerk

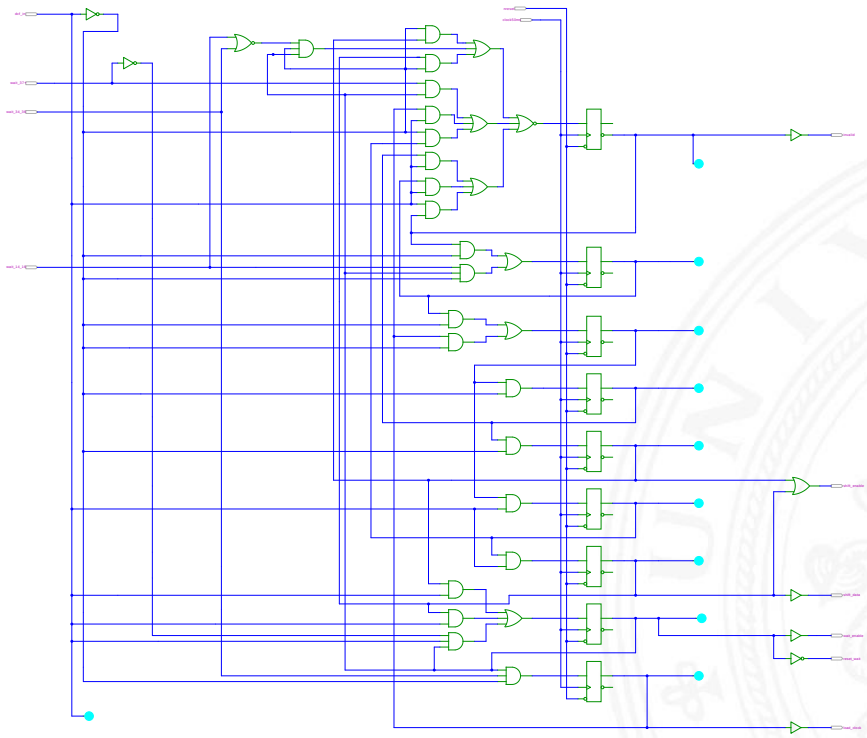
- Beschreibung z.B.: `de.wikipedia.org/wiki/DCF77`

Funkgesteuerte DCF 77 Uhr: Gesamtsystem



Hades Webdemos: 45-misc/80-dcf77/dcf77

Funkgesteuerte DCF 77 Uhr: Decoder-Schaltwerk



Hades Webdemos:
45-misc/80-dcf77/DecoderFSM

Multiplex-Siebensegment-Anzeige

Ansteuerung mehrstelliger Siebensegment-Anzeigen?

- direkte Ansteuerung erfordert $7 \cdot n$ Leitungen für n Ziffern
- und je einen Siebensegment-Decoder pro Ziffer

Zeit-Multiplex-Verfahren benötigt nur $7 + n$ Leitungen

- die Anzeigen werden nacheinander nur ganz kurz eingeschaltet
- ein gemeinsamer Siebensegment-Decoder
  Eingabe wird entsprechend der aktiven Ziffer umgeschaltet
- das Auge sieht die leuchtenden Segmente und „mittelt“
- ab ca. 100 Hz Frequenz erscheint die Anzeige ruhig

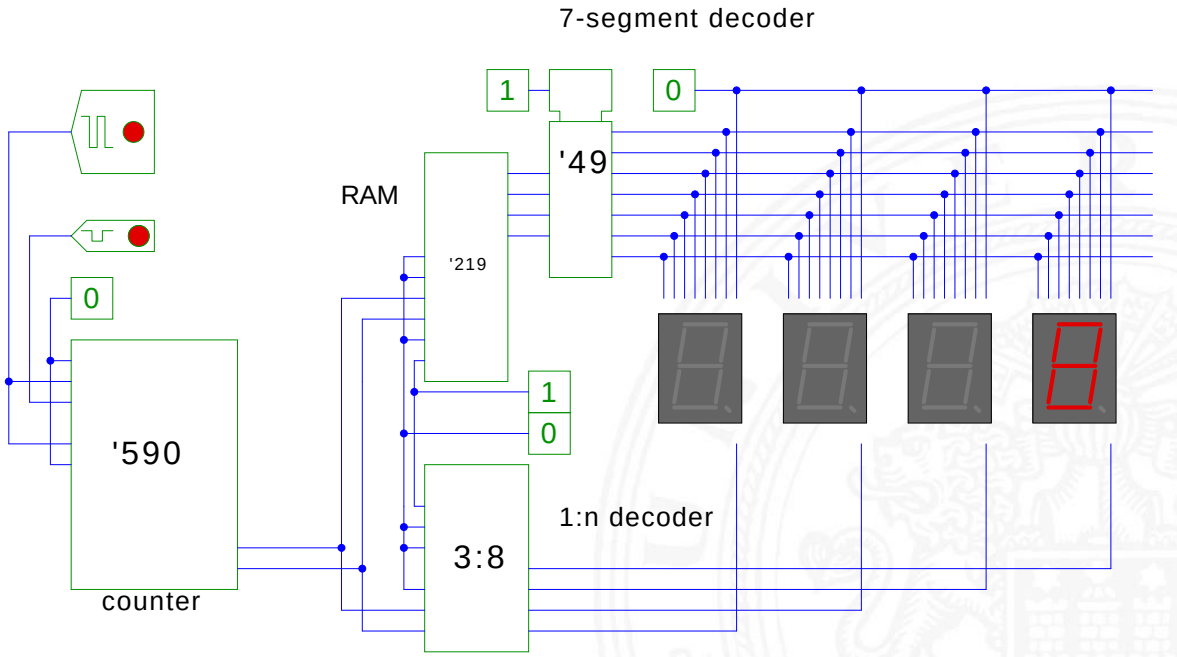
Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
14.6.3 Schaltwerke - Beispiele - verschiedene Beispiele
64-040 Rechnerstrukturen

Multiplex-Siebensegment-Anzeige (cont.)

Hades-Beispiel: Kombination mehrerer bekannter einzelner Schaltungen zu einem komplexen Gesamtsystem

- vierstellige Anzeige
- darzustellende Werte sind im RAM (74219) gespeichert
- Zähler-IC (74590) erzeugt 2-bit Folge {00, 01, 10, 11}
- 3:8-Decoder-IC (74138) erzeugt daraus die Folge {1110, 1101, 1011, 0111} um nacheinander je eine Anzeige zu aktivieren (low-active)
- Siebensegment-Decoder-IC (7449) treibt die sieben Segmentleitungen

Multiplex-Siebensegment-Anzeige (cont.)



Hades Webdemos: 45-misc/50-displays/multiplexed-display

Ausblick: Asynchrone Schaltungen

- Kosten und Verzögerung pro Gatter fallen
- zentraler Takt zunehmend problematisch: Performance, Energieverbrauch, usw.
- alle Rechenwerke warten auf langsamste Komponente

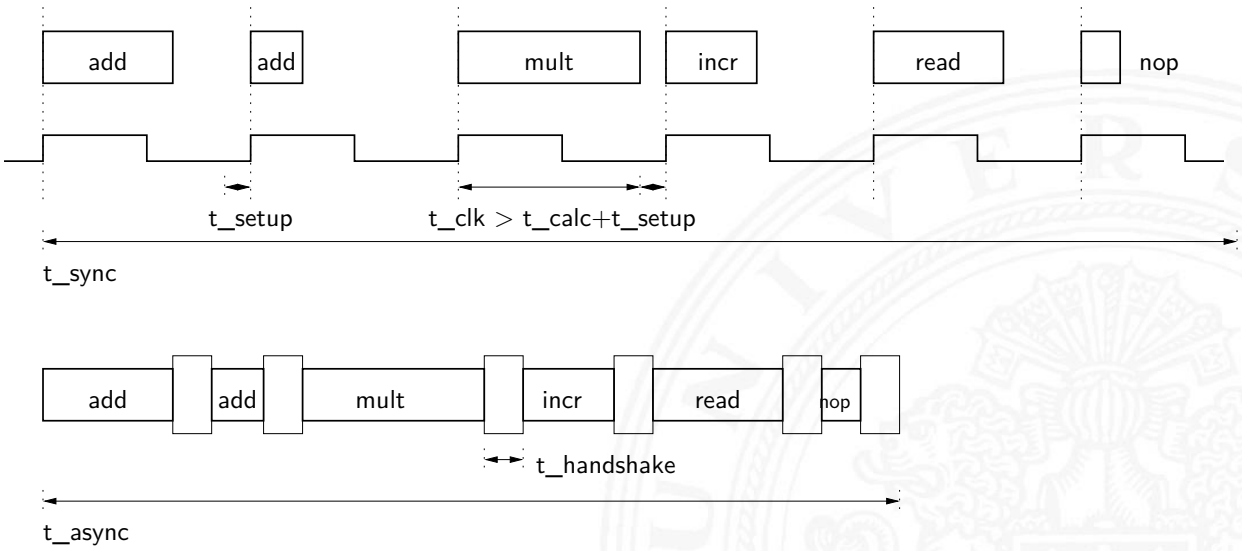
Umstieg auf nicht-getaktete Schaltwerke?!

- *Handshake*-Protokolle zwischen Teilschaltungen
  - Berechnung startet, sobald benötigte Operanden verfügbar
  - Rechenwerke signalisieren, dass Ergebnisse bereitstehen

\+ kein zentraler Takt notwendig $\Rightarrow$ so schnell wie möglich

– Probleme mit Deadlocks und Initialisierung

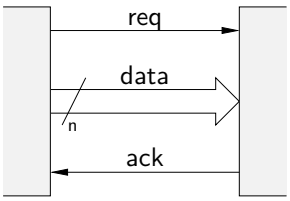
Asynchrone Schaltungen: Performance



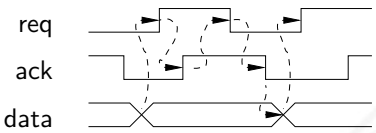
- synchron: Pipelining/Path-Balancing können Verschnitt verringern
- asynchron: Operationen langsamer wegen „completion detection“

Zwei-Phasen und Vier-Phasen Handshake

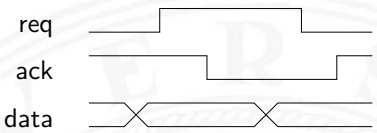
bundled data



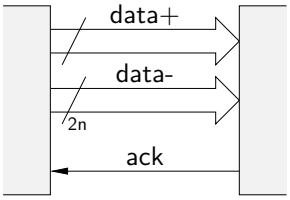
four-phase "level"



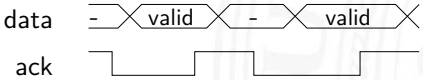
two-phase "edge"



dual rail



four-phase

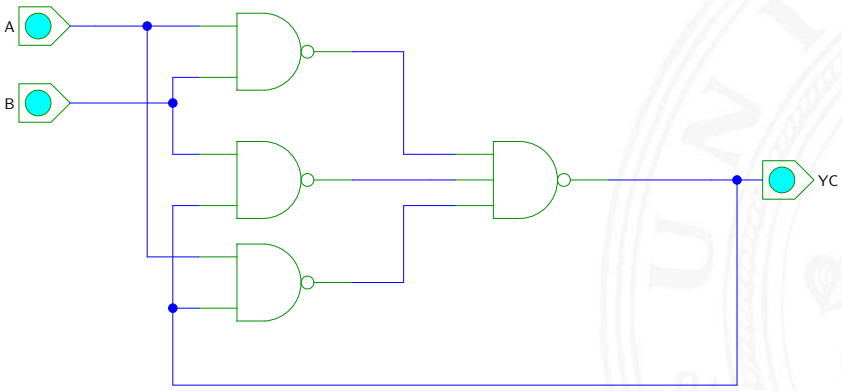


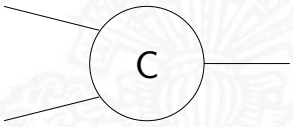
	d+	d-
empty	0	0
valid "0"	0	1
valid "1"	1	0
unused	1	1

Muller C-Gate

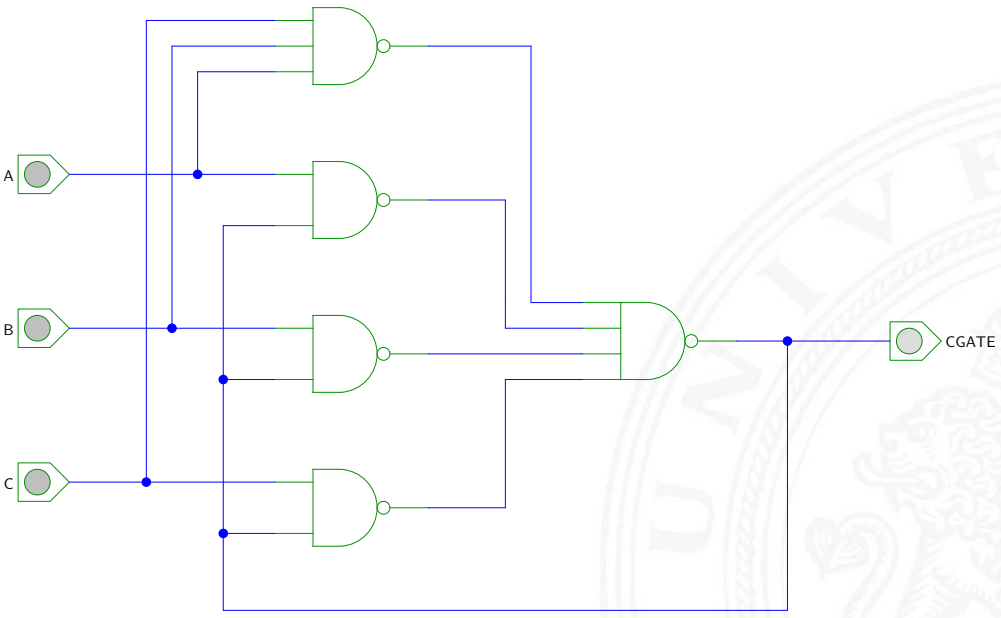
- asynchrones Schaltwerk
- alle Eingänge 0: Ausgang wird 0
  –"– 1: –"– 1
- wird oft in asynchronen Schaltungen benutzt

c \ ab	00	01	11	10
0	0	0	1	0
1	0	1	1	1





Muller C-Gate: 3-Eingänge



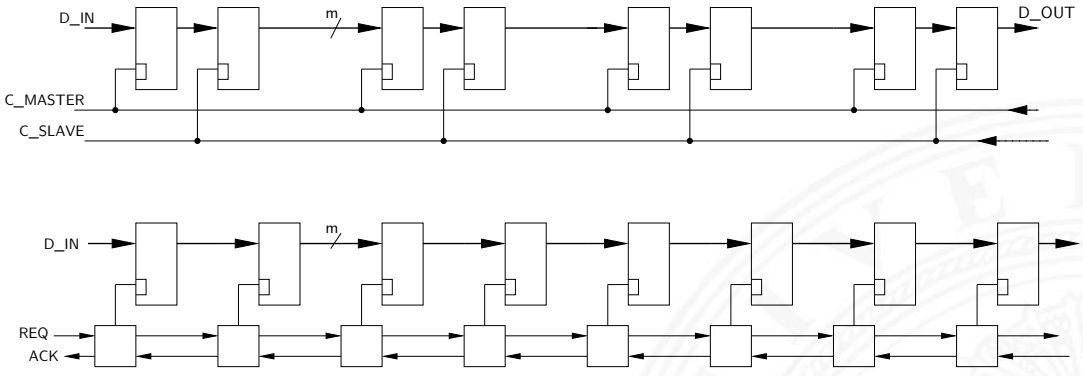
Asynchrone Schaltungen: Micropipeline

- einfaches Modell einer generischen nicht-getakteten Schaltung
- Beispiel zum Entwurf und zur Kaskadierung
- Muller C-Gate als Speicherglieder
- beliebige Anzahl Stufen

- neue Datenwerte von links in die Pipeline einfüllen
- Werte laufen soweit nach rechts wie möglich
- solange bis Pipeline gefüllt ist

- Datenwerte werden nach rechts entnommen
- Pipeline signalisiert automatisch, ob Daten eingefüllt oder entnommen werden können

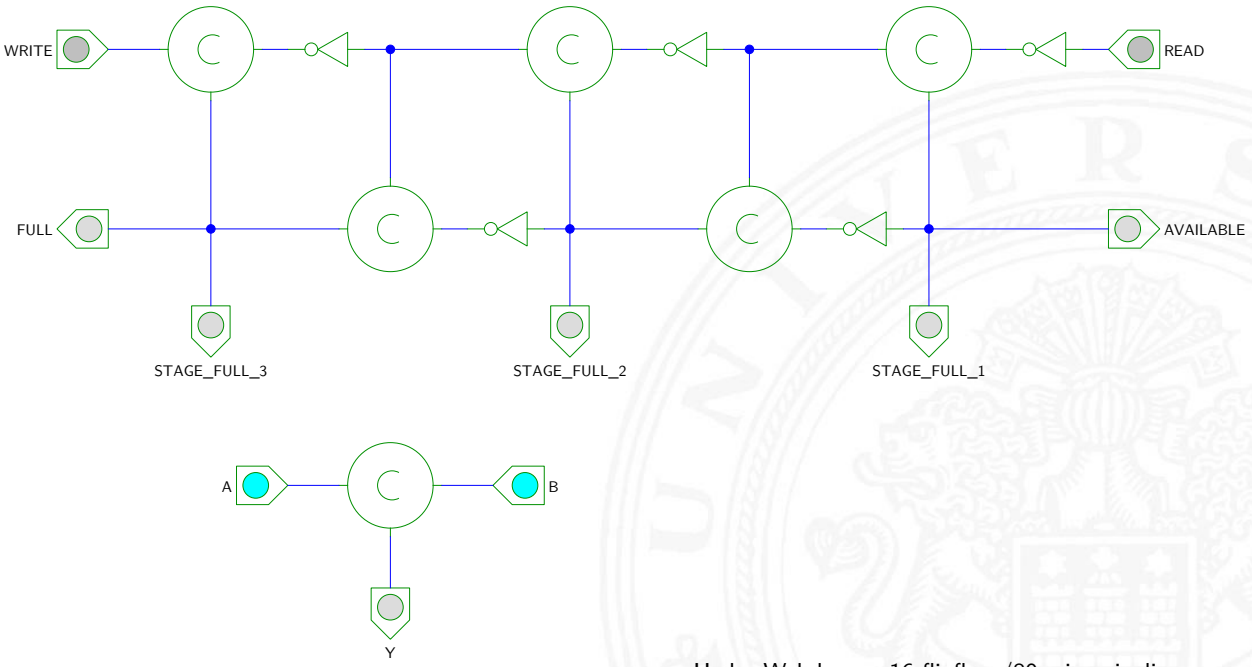
Micropipeline: Konzept



n-stufige Micropipeline vs. getaktetes Schieberegister

- lokales Handshake statt globalem Taktsignal
- Datenkapazität entspricht $2n$-stufigem Schieberegister
- leere Latches transparent: schnelles Einfüllen
- „elastisch“: enthält $0 \ldots 2n$ Datenworte

Micropipeline: Demo mit C-Gates



Hades Webdemos: 16-flipflops/80-micropipeline

Literatur: Vertiefung

- David Harel,
  Statecharts, A visual formalism for complex systems,
  CS84-05, Department of Applied Mathematics,
  The Weizmann Institute of Science, 1984
  `www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf`

- Neil H. E. Weste, Kamran Eshragian,
  Principles of CMOS VLSI Design — A Systems Perspective,
  Addison-Wesley Publishing, 1994

Interaktives Lehrmaterial

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

- Norman Hendrich,
  HADES — HAmburg DEsign System,
  Universität Hamburg, FB Informatik
  `tams.informatik.uni-hamburg.de/applets/hades`

Gliederung

Gliederung (cont.)

Gliederung (cont.)

Aufbau kompletter Rechensysteme

- bisher:
  - Gatter und Schaltnetze
  - Flipflops als einzelne Speicherglieder
  - Schaltwerke zur Ablaufsteuerung
- jetzt zusätzlich:
  - Speicher
  - Busse
  - Register-Transfer Komponenten eines Rechners
  - Ablaufsteuerung (Timing, Mikroprogrammierung)

Wiederholung: von-Neumann-Konzept

- J. Mauchly, J.P. Eckert, J. von-Neumann 1945
- System mit Prozessor, Speicher, Peripheriegeräten
- gemeinsamer Speicher für Programme und Daten
- 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.

Wiederholung: von-Neumann Rechner

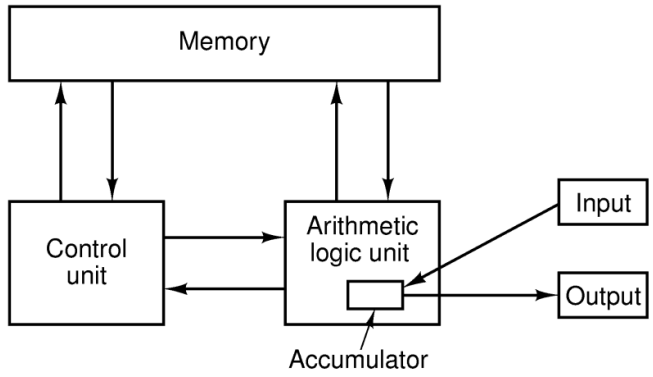


Figure 1-5. The original von Neumann machine.

Fünf zentrale Komponenten:

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

Wiederholung: von-Neumann Rechner (cont.)

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

Systemmodellierung

Modellierung eines digitalen Systems als Schaltung aus

- Speichergliedern
  - Registern — Flipflops, Register, Registerbank
  - Speichern — SRAM, DRAM, ROM, PLA
- Rechenwerken
  - Addierer, arithmetische Schaltungen
  - logische Operationen
  - „random-logic“ Schaltnetzen
- Verbindungsleitungen
  - Busse / Leitungsbündel
  - Multiplexer und Tri-state Treiber

Speicher

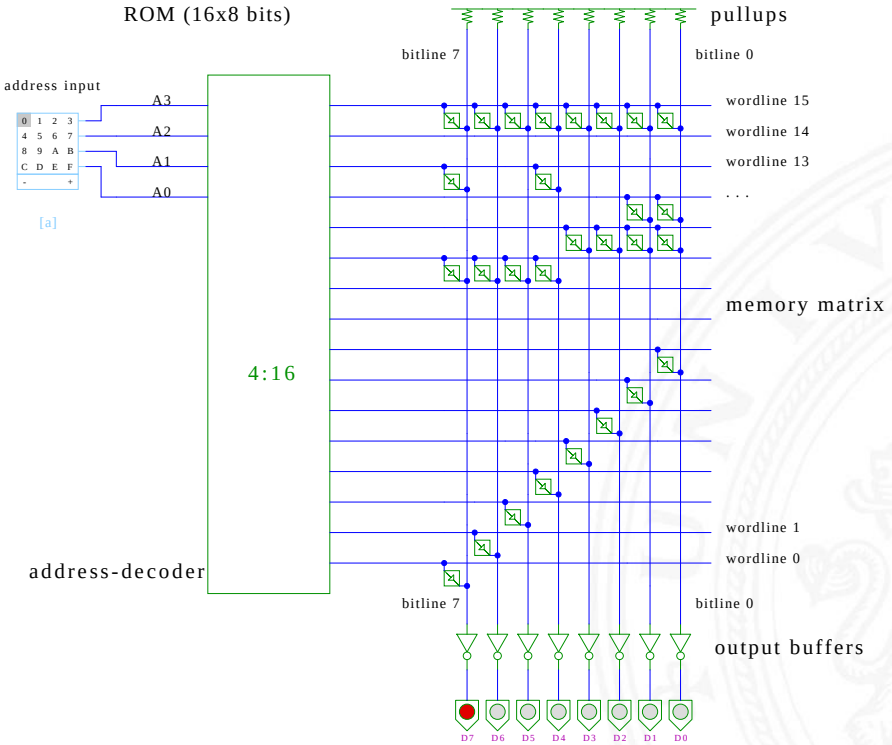
- System zur Speicherung von Information
- als Feld von N Adressen mit je m bit
- typischerweise mit n-bit Adressen und $N = 2^n$
- Kapazität also $2^n \times m$ bits

- 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

Type	Category	Erasure	Byte alterable	Volatile	Typical use
SRAM	Read/write	Electrical	Yes	Yes	Level 2 cache
DRAM	Read/write	Electrical	Yes	Yes	Main memory
ROM	Read-only	Not possible	No	No	Large volume appliances
PROM	Read-only	Not possible	No	No	Small volume equipment
EPROM	Read-mostly	UV light	No	No	Device prototyping
EEPROM	Read-mostly	Electrical	Yes	No	Device prototyping
Flash	Read/write	Electrical	No	No	Film for digital camera

ROM: Read-Only Memory



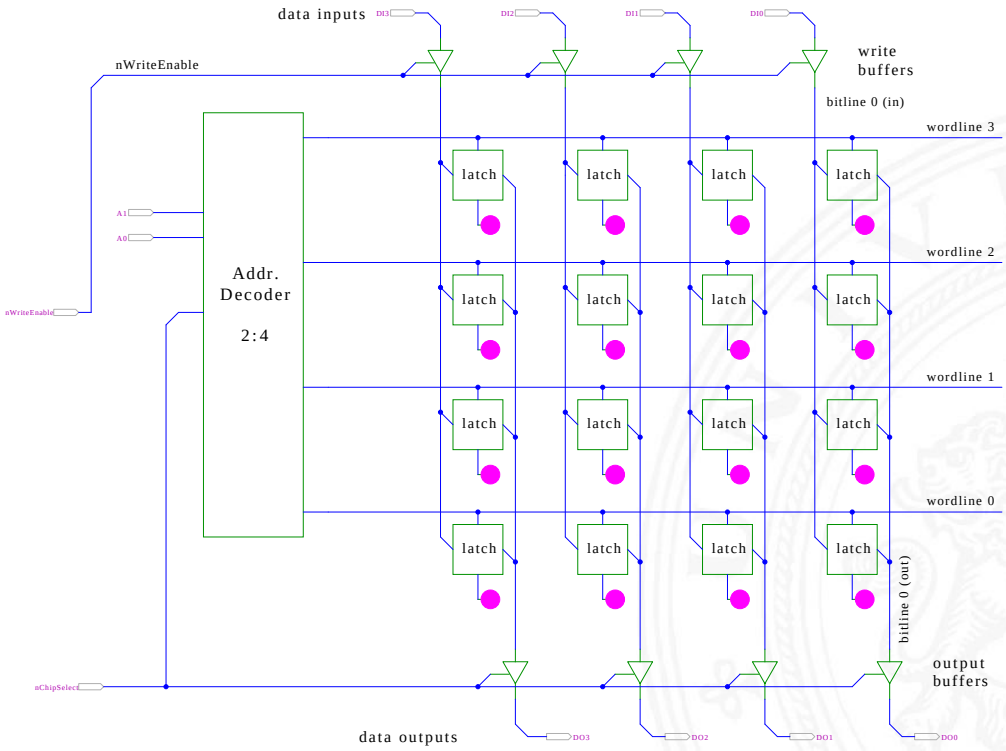
RAM: Random-Access Memory

Speicher, der im Betrieb gelesen und geschrieben werden kann

- Arbeitsspeicher des Rechners
- für Programme und Daten
- keine Abnutzungseffekte

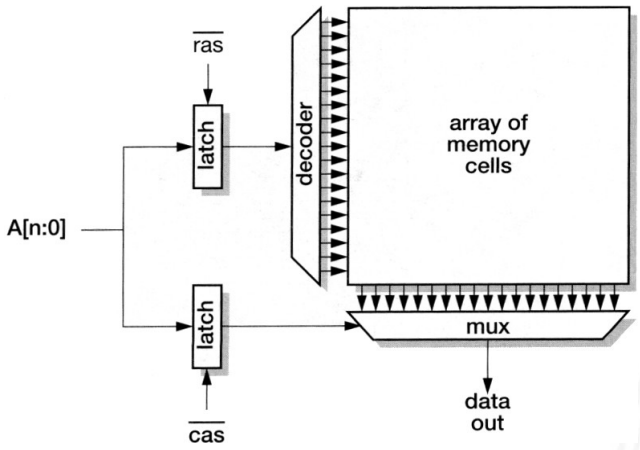
- Aufbau als Matrixstruktur
- n Adressbits, konzeptionell 2^n Wortleitungen
- m Bits pro Wort
- Realisierung der einzelnen Speicherstellen?
  - statisches RAM: 6-Transistor Zelle — SRAM
  - dynamisches RAM: 1-Transistor Zelle — DRAM

RAM: Blockschaltbild



4×4 bit
2-bit Adresse
4-bit Datenwort

RAM: RAS/CAS-Adressdecodierung

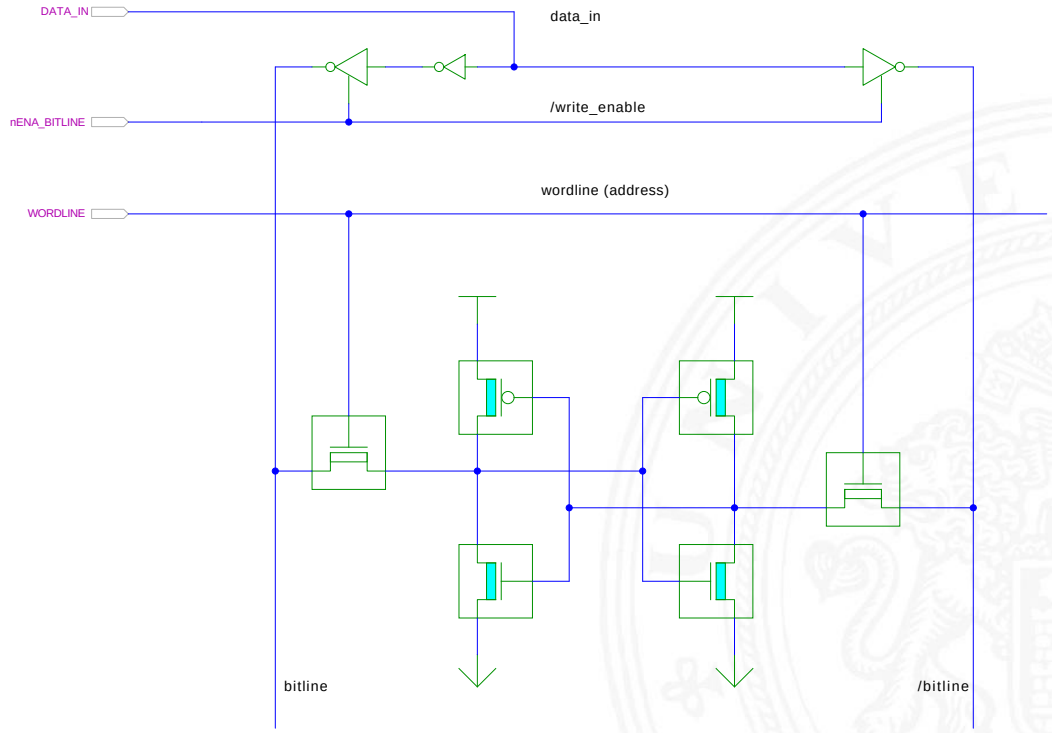


- Aufteilen der Adresse in zwei Hälften
- $\overline{ras}$ „row address strobe“ wählt „Wordline“
  $\overline{cas}$ „column address strobe“ –"– „Bitline“
- je ein $2^{(n/2)}$-bit Decoder/Mux statt ein 2^n-bit Decoder

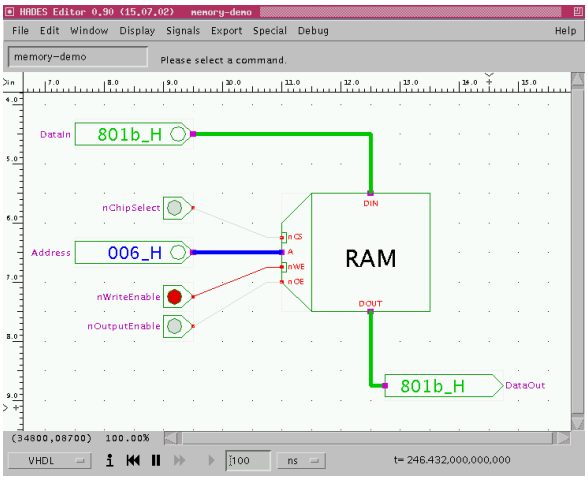
SRAM: statisches RAM

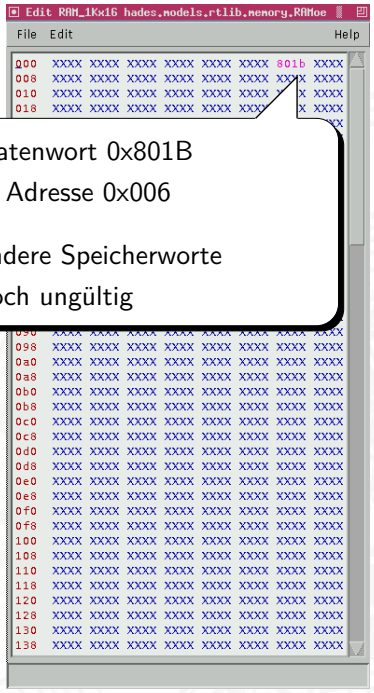
- Inhalt bleibt dauerhaft gespeichert solange Betriebsspannung anliegt
- *sechs-Transistor* Zelle zur Speicherung
  - weniger Platzverbrauch als Latches/Flipflops
  - kompakte Realisierung in CMOS-Technologie (s.u.)
  - zwei rückgekoppelte Inverter zur Speicherung
  - zwei n-Kanal Transistoren zur Anbindung an die Bitlines
- schneller Zugriff: Einsatz für Caches
- deutlich höherer Platzbedarf als DRAMs

SRAM: Sechs-Transistor Speicherstelle („6T“)



SRAM: Hades Demo





- nur aktiv wenn nCS=0 (chip select)
- Schreiben wenn nWE=0 (write enable)
- Ausgabe wenn nOE=0 (output enable)

`tams.informatik.uni-hamburg.de/applets/hades/webdemos/50-rtlib/40-memory/ram.html`

SRAM: Beispiel IC 6116

- integrierte Schaltung, 16 Kbit Kapazität
- Organisation als 2K Worte mit je 8-bit

- 11 Adresseingänge (A10 .. A0)
- 8 Anschlüsse für gemeinsamen Daten-Eingang/-Ausgang
- 3 Steuersignale
  - $\overline{CS}$ chip-select: Speicher nur aktiv wenn $\overline{CS} = 0$
  - $\overline{WE}$ write-enable: Daten an gewählte Adresse schreiben
  - $\overline{OE}$ output-enable: Inhalt des Speichers ausgeben

- interaktive Hades-Demo zum Ausprobieren

  `tams.informatik.uni-hamburg.de/applets/hades/webdemos/40-memories/40-ram`

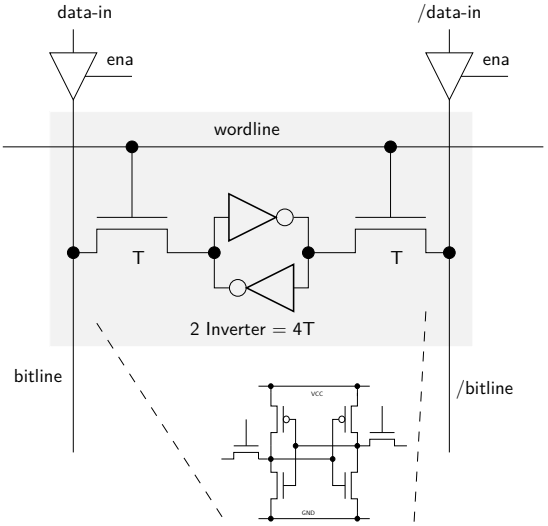
DRAM: dynamisches RAM

- Information wird in winzigen Kondensatoren gespeichert
- pro Bit je ein Transistor und Kondensator

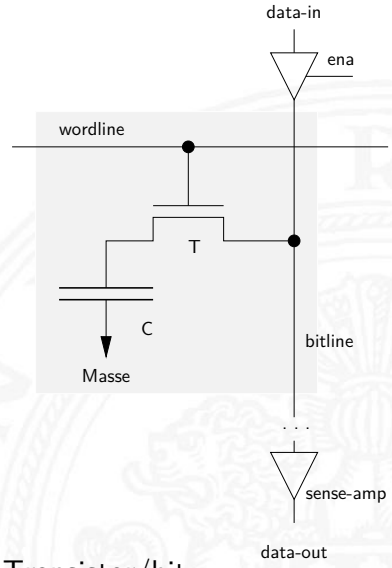
- jeder Lesezugriff entlädt den Kondensator
- *Leseverstärker* zur Messung der Spannung auf der Bitline
  Schwellwertvergleich zur Entscheidung logisch 0/1

– 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

DRAM vs. SRAM



- 6 Transistoren/bit
- statisch (kein refresh)
- schnell
- 10 .. 50X DRAM-Fläche



- 1 Transistor/bit
- C=10fF: ~200.000 Elektronen
- langsam (sense-amp)
- minimale Fläche

DRAM: Stacked- und Trench-Zelle

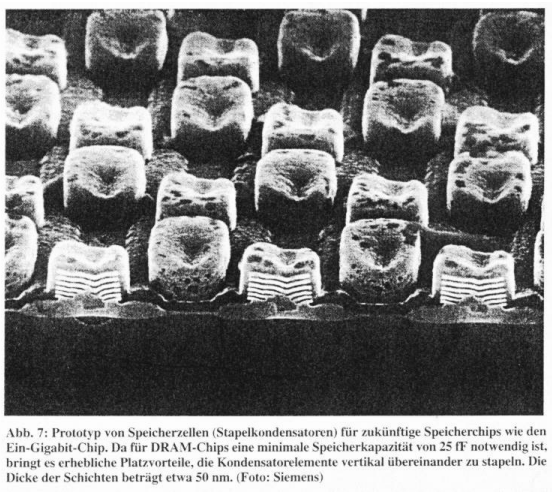
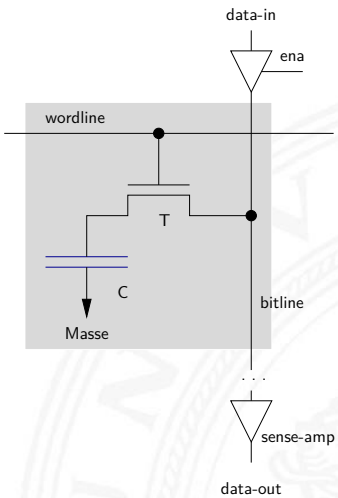


Abb. 7: Prototyp von Speicherzellen (Stapelkondensatoren) für zukünftige Speicherchips wie den Ein-Gigabit-Chip. Da für DRAM-Chips eine minimale Speicherkapazität von 25 fF notwendig ist, bringt es erhebliche Platzvorteile, die Kondensatorelemente vertikal übereinander zu stapeln. Die Dicke der Schichten beträgt etwa 50 nm. (Foto: Siemens)

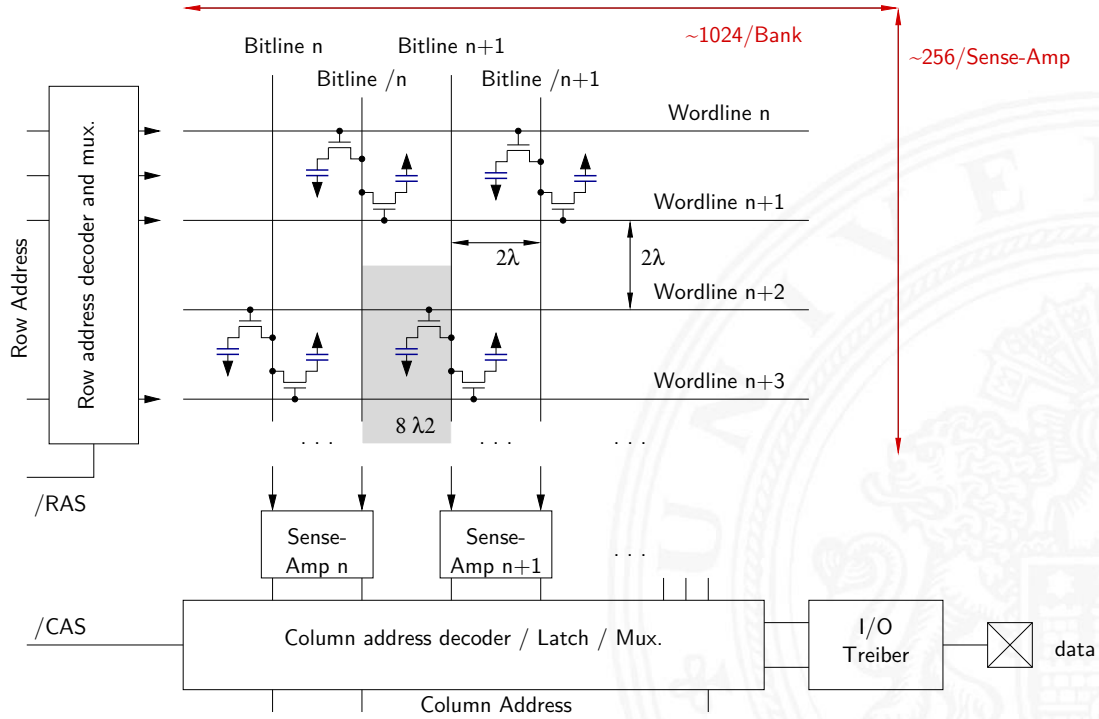
Siemens 1 Gbit DRAM



IBM CMOS-6X embedded DRAM

- zwei Bauformen: „stacked“ und „trench“
- Kondensatoren: möglichst kleine Fläche, Kapazität gerade ausreichend

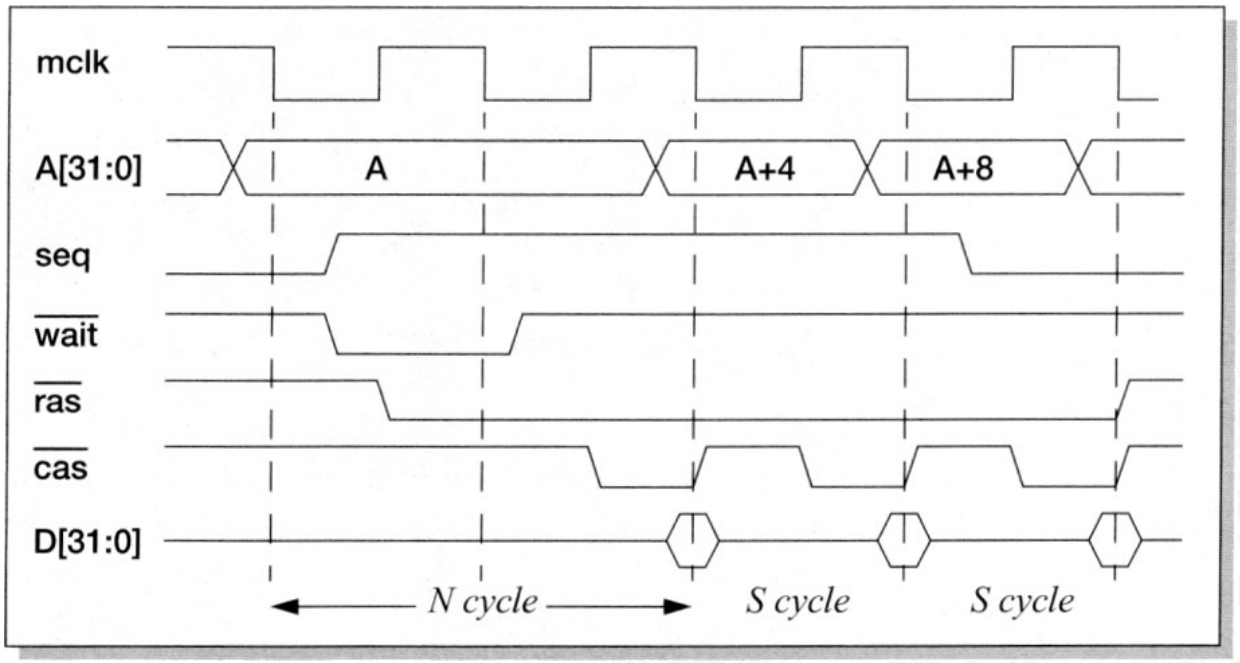
DRAM: Layout



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
  - DDR-2, DDR-3: Varianten mit höherer Taktrate
    aktuell Übertragungsraten bis 17 GByte/sec

SDRAM: Lesezugriff auf sequenzielle Adressen



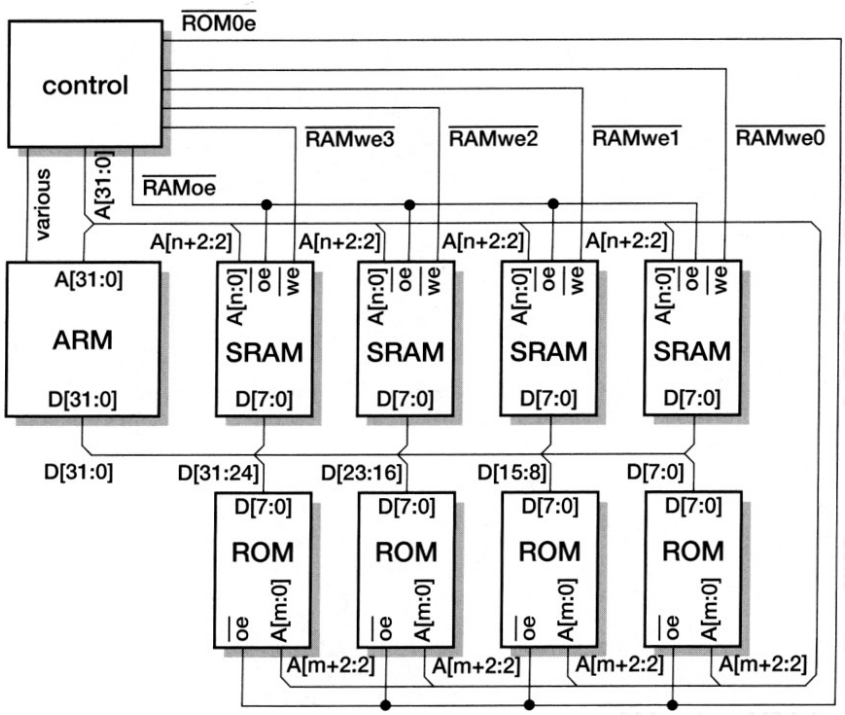
Flash

- 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

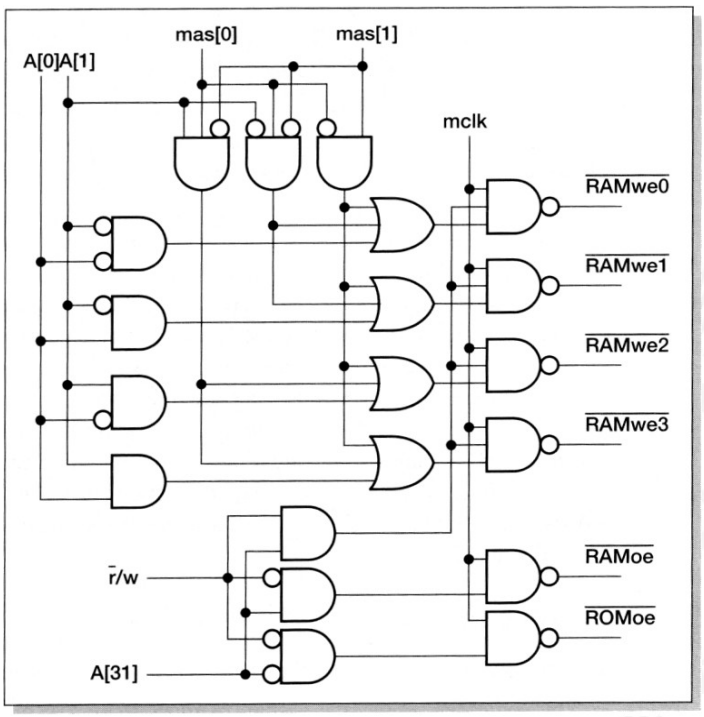
- Auslesen beliebig oft möglich, schnell
- Schreibzugriffe problematisch
  - intern hohe Spannung erforderlich (Gate-Isolierung überwinden)
  - Schreibzugriffe einer „0“ nur blockweise
  - pro Zelle nur einige 10 000. . . 100 000 Schreibzugriffe möglich

Typisches Speichersystem



32-bit Prozessor
4× 8-bit SRAMs
4× 8-bit ROMs

Typisches Speichersystem: Adressdecodierung

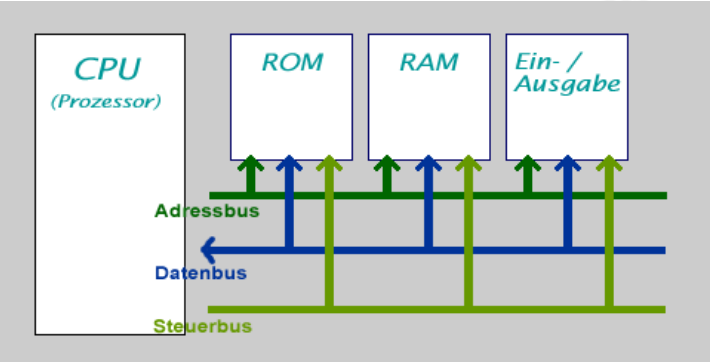


Bussysteme

- **Bus:** elektrische (und logische) Verbindung
  - mehrere Geräte
  - mehrere Blöcke innerhalb einer Schaltung
- Bündel aus Daten- und Steuersignalen
- mehrere Quellen (und mehrere Senken [lesende Zugriffe])
  - spezielle elektrische Realisierung:
    Tri-State-Treiber oder Open-Drain
- Bus-Arbitrierung: wer darf, wann, wie lange senden?
  - Master-Slave
  - gleichberechtigte Knoten, Arbitrierungsprotokolle
- synchron: mit globalem Taktsignal vom „Master“-Knoten
  asynchron: Wechsel von Steuersignalen löst Ereignisse aus

Bussysteme (cont.)

- typische Aufgaben
  - Kernkomponenten (CPU, Speicher...) miteinander verbinden
  - Verbindungen zu den Peripherie-Bausteinen
  - Verbindungen zu Systemmonitor-Komponenten
  - Verbindungen zwischen I/O-Controllern und -Geräten
  - ...



Bussysteme (cont.)

- viele unterschiedliche Typen, standardisiert mit sehr unterschiedlichen Anforderungen
  - High-Performance
  - einfaches Protokoll, billige Komponenten
  - Multi-Master-Fähigkeit, zentrale oder dezentrale Arbitrierung
  - Echtzeitfähigkeit, Daten-Streaming
  - wenig Leitungen bis zu Zweidraht-Bussen: I^2C, System-Management-Bus...
  - lange Leitungen: RS232, Ethernet...
  - Funkmedium: WLAN, Bluetooth (logische Verbindung)

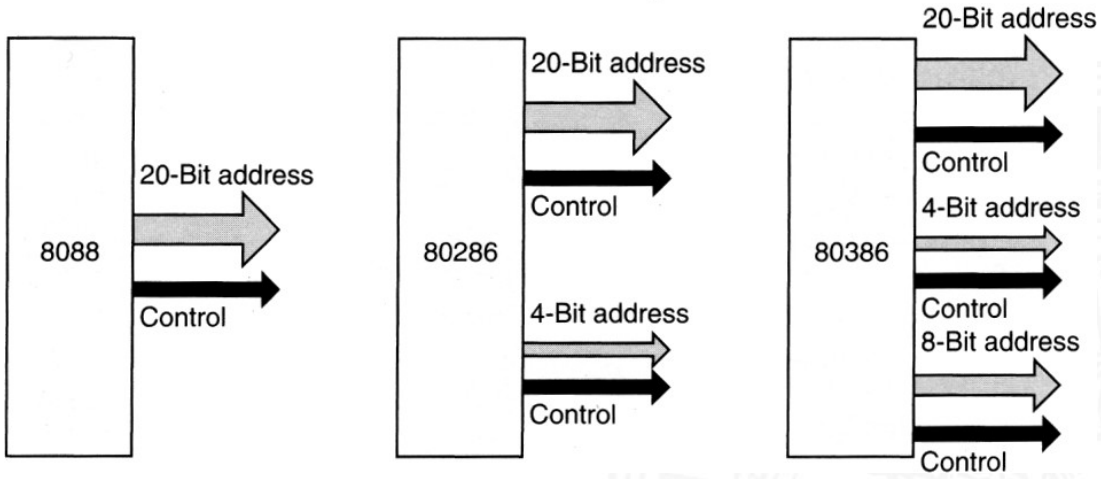
Bus: Mikroprozessorsysteme

typisches n-bit Mikroprozessor-System:

- n Adress-Leitungen, also Adressraum 2^n Bytes — Adressbus
- n Daten-Leitungen — Datenbus

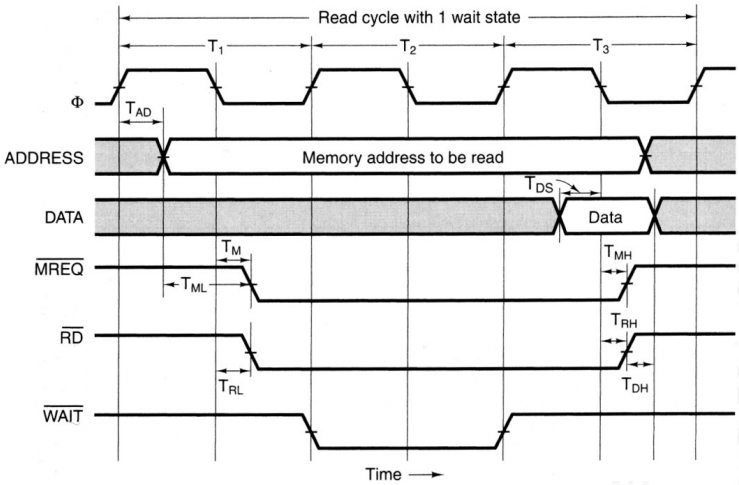
- Steuersignale — Control
  - clock: Taktsignal
  - read/write: Lese-/Schreibzugriff (aus Sicht des Prozessors)
  - wait: Wartezeit/-zyklen für langsame Geräte
  - ...
- um Leitungen zu sparen, teilweise gemeinsam genutzte Leitungen sowohl für Adressen als auch Daten. Zusätzliches Steuersignal zur Auswahl Adressen/Daten

Adressbus: Evolution beim Intel x86



- 20-bit: 1 MiByte Adressraum
  24-bit: 16 MiByte
  32-bit: 4 GiByte
- alle Erweiterungen abwärtskompatibel

Synchroner Bus: Timing



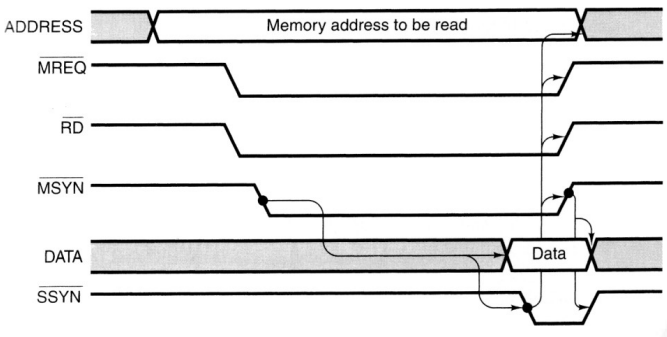
A.S. Tanenbaum, *Structured Computer Organization* 3.4.4

- 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: typische Zeit-Parameter

Symbol	Parameter	Min	Max	Unit
T_{AD}	Address output delay		4	nsec
T_{ML}	Address stable prior to $\overline{MREQ}$	2		nsec
T_M	$\overline{MREQ}$ delay from falling edge of Φ in T_1		3	nsec
T_{RL}	RD delay from falling edge of Φ in T_1		3	nsec
T_{DS}	Data setup time prior to falling edge of Φ	2		nsec
T_{MH}	$\overline{MREQ}$ delay from falling edge of Φ in T_3		3	nsec
T_{RH}	$\overline{RD}$ delay from falling edge of Φ in T_3		3	nsec
T_{DH}	Data hold time from negation of $\overline{RD}$	0		nsec

Asynchroner Bus: Lesezugriff



- Steuersignale $\overline{MSYN}$: Master fertig
  $\overline{SSYN}$: Slave fertig
- flexibler für Geräte mit stark unterschiedlichen Zugriffszeiten

Bus Arbitrierung

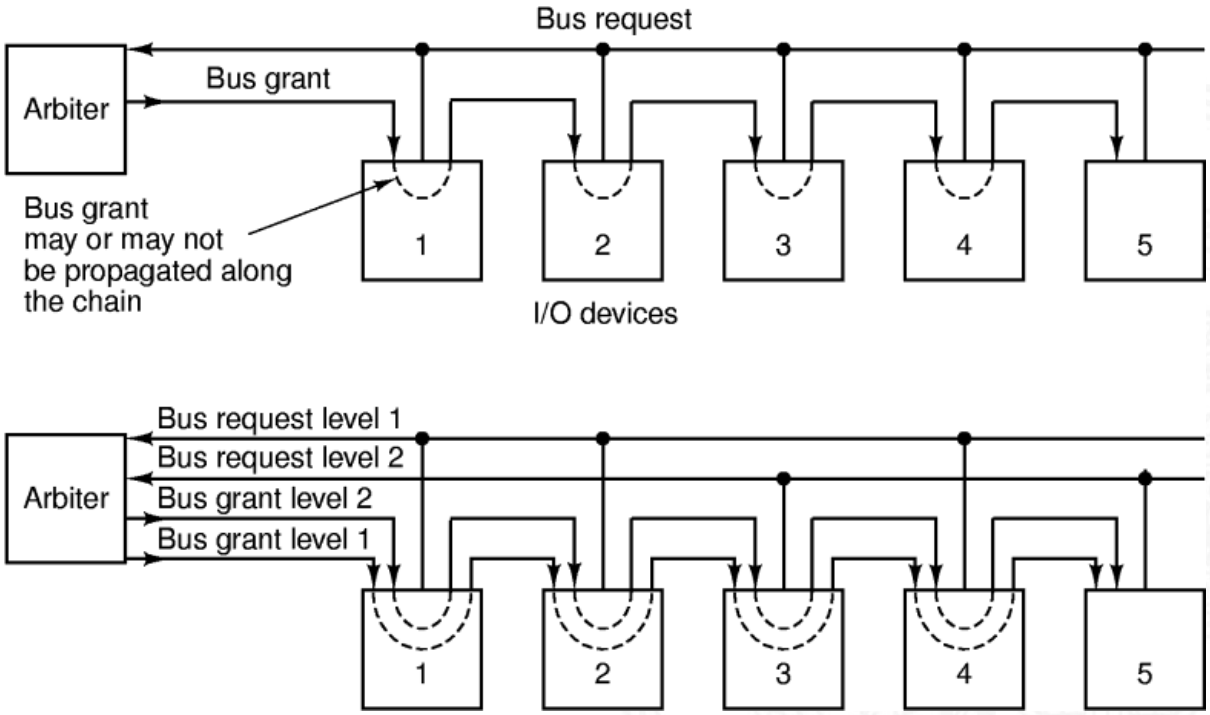
- mehrere Komponenten wollen Übertragung initiieren
  immer nur ein Transfer zur Zeit möglich
- der Zugriff muss serialisiert werden

1. zentrale Arbitrierung
   - Arbiter gewährt Bus-Requests
   - Strategien
     - Prioritäten für verschiedene Geräte
     - „round-robin“ Verfahren
     - „Token“-basierte Verfahren
     - usw.

Bus Arbitrierung (cont.)

2. dezentrale Arbitrierung
   - protokollbasiert
   - Beispiel
     - Komponenten sehen ob Bus frei ist
     - beginnen zu senden
     - Kollisionserkennung: gesendete Daten lesen
     - ggf. Übertragung abbrechen
     - „später“ erneut versuchen
- I/O-Geräte oft höher priorisiert als die CPU
  - I/O-Zugriffe müssen schnell/sofort behandelt werden
  - Benutzerprogramm kann warten

Bus Arbitrierung (cont.)



Bus Bandbreite

- Menge an (Nutz-) Daten, die pro Zeiteinheit übertragen werden kann
- zusätzlicher Protokolloverhead ⇒ Brutto- / Netto-Datenrate
- RS232 50 Bit/sec ... 460 KBit/sec
  I^2C 100 KBit/sec (Std.)... 3,4 MBit/sec (High Speed)
  USB 1,5 MBit/sec (1.x) ... 5 GBit/sec (3.0)
  ISA 128 MBit/sec
  PCI 1 GBit/sec (2.0) ... 4,3 GBit/sec (3.0)
  AGP 2,1 GBit/sec (1x) ... 16,6 GBit/sec (8x)
  PCIe 250 MByte/sec (1.x) ...1000 MByte/sec (3.0) x1...32
  HyperTransport 12,8 GByte/sec (1.0) ... 51,2 GByte/sec (3.1)

Beispiel: PCI-Bus

Peripheral Component Interconnect (Intel 1991)

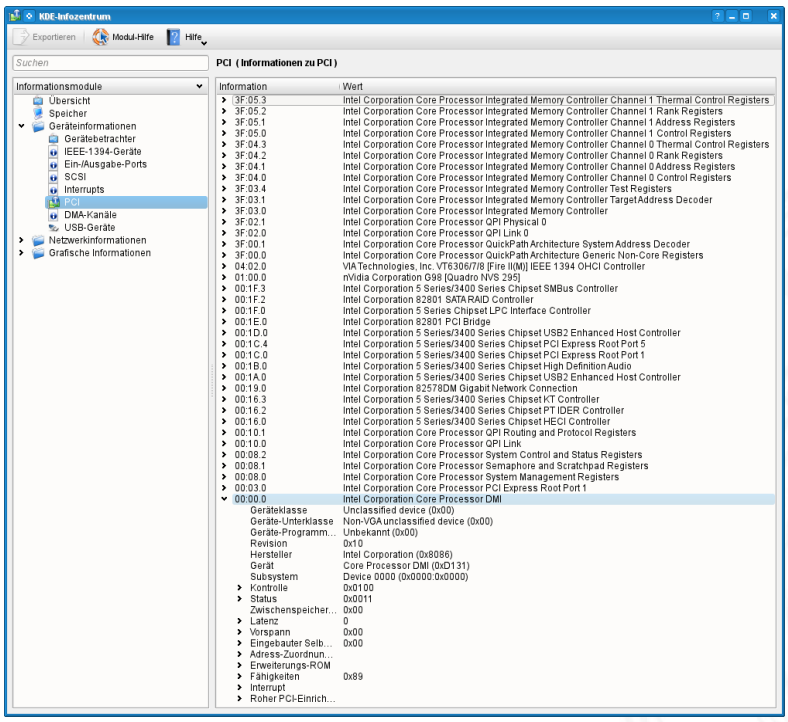
- 33 MHz Takt — optional 64 MHz Takt
- 32-bit Bus-System — optional auch 64-bit
- gemeinsame Adress-/Datenleitungen
- Arbitrierung durch Bus-Master — CPU

- Auto-Konfiguration
  - angeschlossene Geräte werden automatisch erkannt
  - eindeutige Hersteller- und Geräte-Nummern
  - Betriebssystem kann zugehörigen Treiber laden
  - automatische Zuweisung von Adressbereichen und IRQs

PCI-Bus: Peripheriegeräte

```
tams12> /sbin/lspci
00:00.0 Host bridge: Intel Corporation 82Q963/Q965 Memory Controller Hub (rev 02)
00:01.0 PCI bridge: Intel Corporation 82Q963/Q965 PCI Express Root Port (rev 02)
00:1a.0 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI #4 (rev 02)
00:1a.1 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI #5 (rev 02)
00:1a.7 USB Controller: Intel Corporation 82801H (ICH8 Family) USB2 EHCI #2 (rev 02)
00:1b.0 Audio device: Intel Corporation 82801H (ICH8 Family) HD Audio Controller (rev 02)
00:1c.0 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 1 (rev 02)
00:1c.4 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 5 (rev 02)
00:1d.0 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI #1 (rev 02)
00:1d.1 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI #2 (rev 02)
00:1d.2 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI #3 (rev 02)
00:1d.7 USB Controller: Intel Corporation 82801H (ICH8 Family) USB2 EHCI #1 (rev 02)
00:1e.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev f2)
00:1f.0 ISA bridge: Intel Corporation 82801HB/HR (ICH8/R) LPC Interface Controller (rev 02)
00:1f.2 IDE interface: Intel Corporation 82801H (ICH8 Family) 4 port SATA IDE Controller (rev 02)
00:1f.3 SMBus: Intel Corporation 82801H (ICH8 Family) SMBus Controller (rev 02)
00:1f.5 IDE interface: Intel Corporation 82801H (ICH8 Family) 2 port SATA IDE Controller (rev 02)
01:00.0 VGA compatible controller: ATI Technologies Inc Unknown device 7183
01:00.1 Display controller: ATI Technologies Inc Unknown device 71a3
03:00.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5754 Gigabit Ethernet PCI Express (rev 02)
```

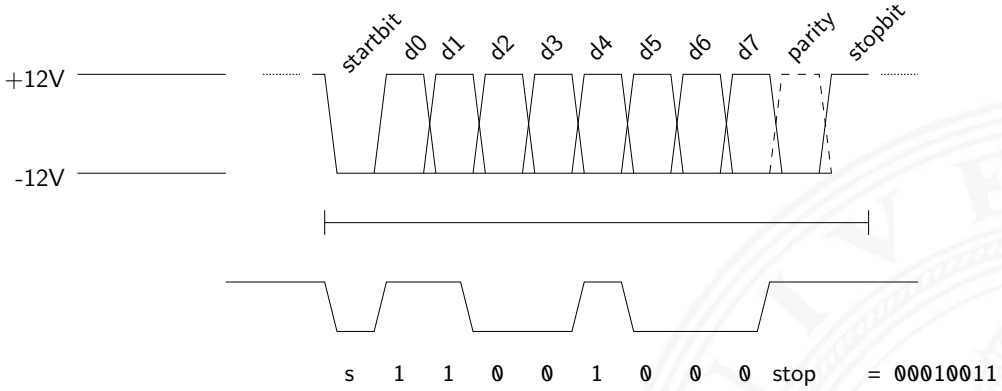
PCI-Bus: Peripheriegeräte (cont.)



PCI-Bus: Leitungen („mandatory“)

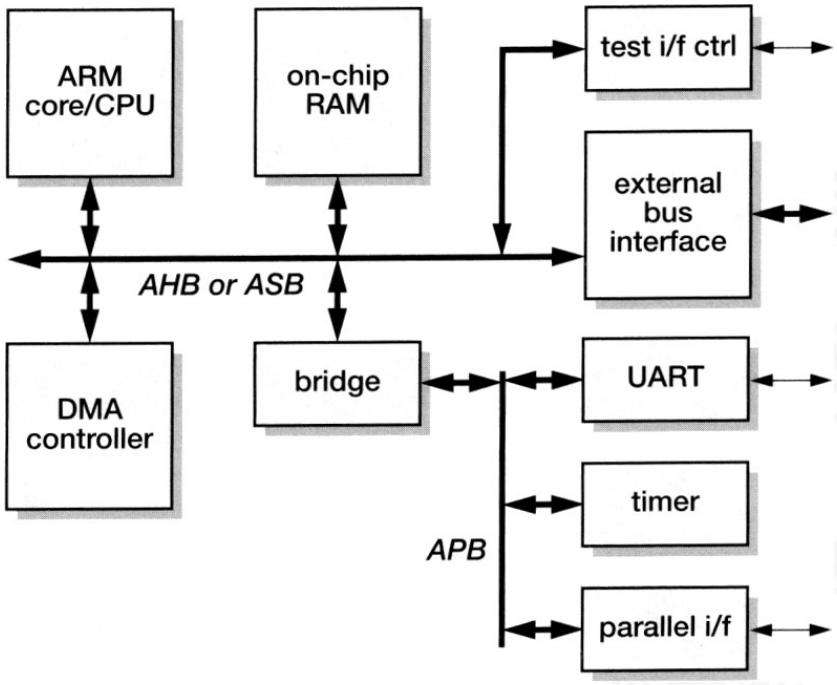
Signal	Lines	Master	Slave	Description
CLK	1			Clock (33 MHz or 66 MHz)
AD	32	×	×	Multiplexed address and data lines
PAR	1	×		Address or data parity bit
C/BE	4	×		Bus command/bit map for bytes enabled
FRAME#	1	×		Indicates that AD and C/BE are asserted
IRDY#	1	×		Read: master will accept; write: data present
IDSEL	1	×		Select configuration space instead of memory
DEVSEL#	1		×	Slave has decoded its address and is listening
TRDY#	1		×	Read: data present; write: slave will accept
STOP#	1		×	Slave wants to stop transaction immediately
PERR#	1			Data parity error detected by receiver
SERR#	1			Address parity error or system error detected
REQ#	1			Bus arbitration: request for bus ownership
GNT#	1			Bus arbitration: grant of bus ownership
RST#	1			Reset the system and all devices

RS-232: Serielle Schnittstelle



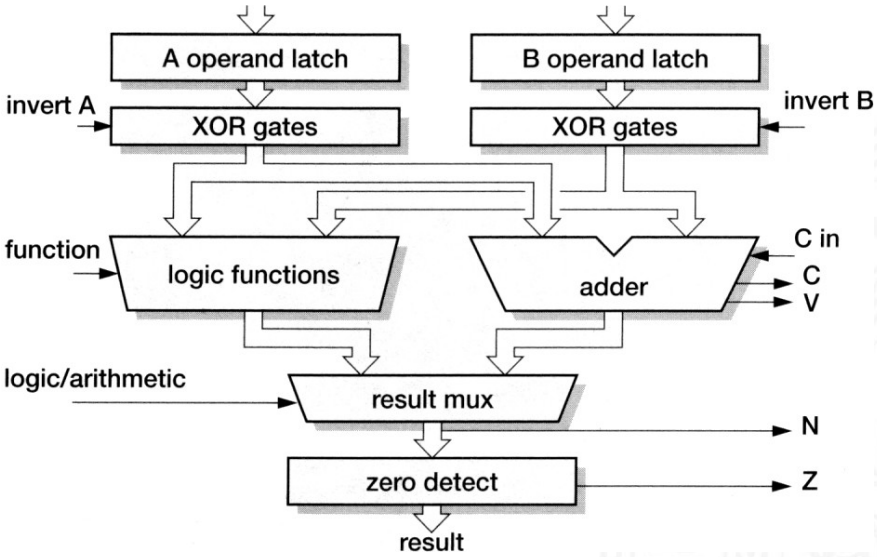
- Baudrate 300, 600, . . . , 19200, 38400, 115200 bits/sec
  Anzahl Datenbits 5, 6, 7, 8
  Anzahl Stopbits 1, 2
  Parität none, odd, even
- minimal drei Leitungen: GND, TX, RX (Masse, Transmit, Receive)
- oft weitere Leitungen für erweitertes Handshake

typisches ARM SoC System



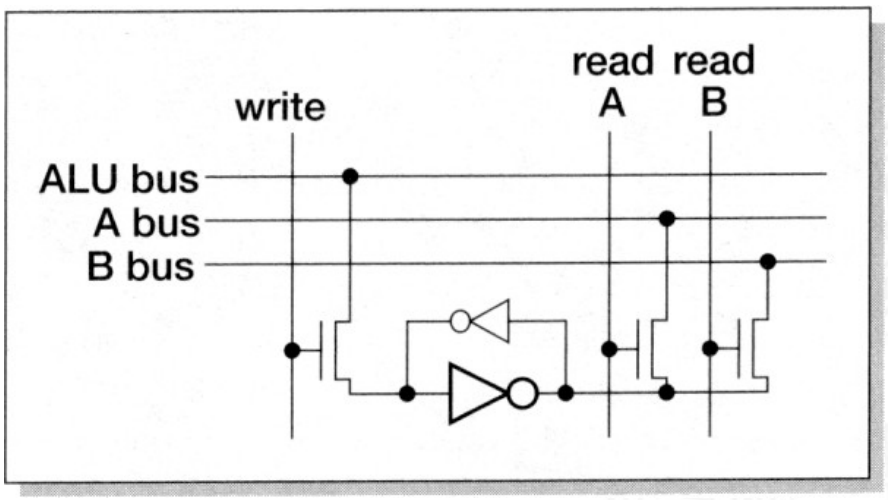
diese und viele folgende Abbildungen: S. Furber, *ARM System-on-Chip Architecture*

RT-Ebene: ALU des ARM-7 Prozessors



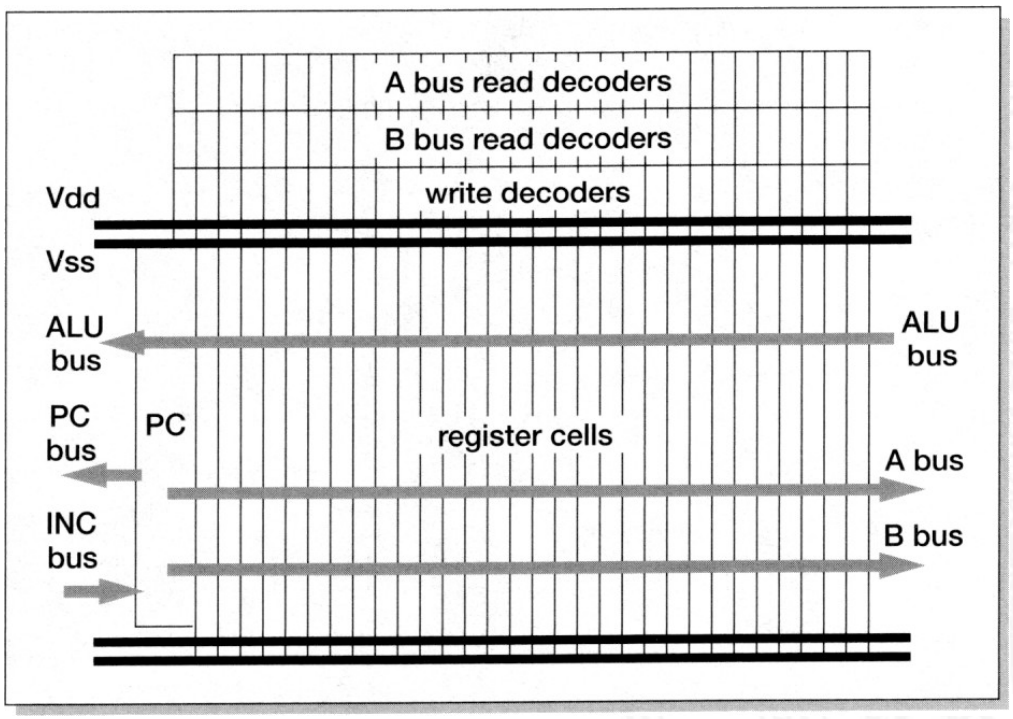
- Register für die Operanden A und B
- Addierer und separater Block für logische Operationen

Multi-Port-Registerbank: Zelle



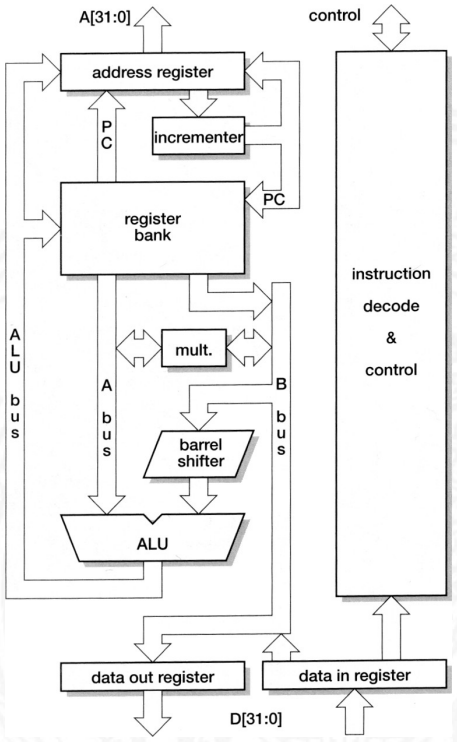
- Prinzip wie 6T-SRAM: rückgekoppelte Inverter
- mehrere (hier zwei) parallele Lese-Ports
- mehrere Schreib-Ports möglich, aber kompliziert

Multi-Port Registerbank: Floorplan/Chiplayout

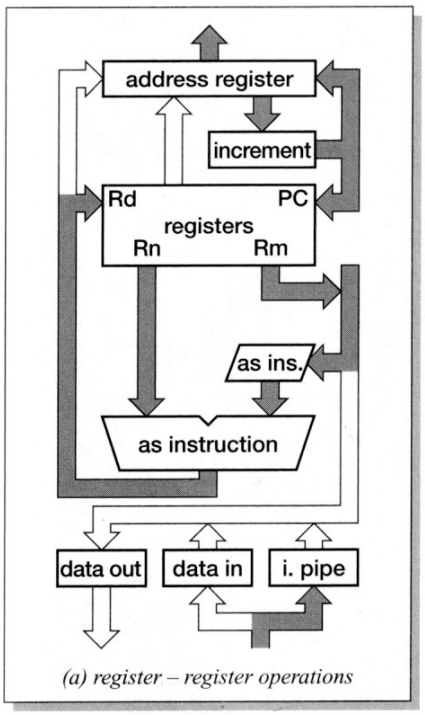


Kompletter Prozessor: ARM-3

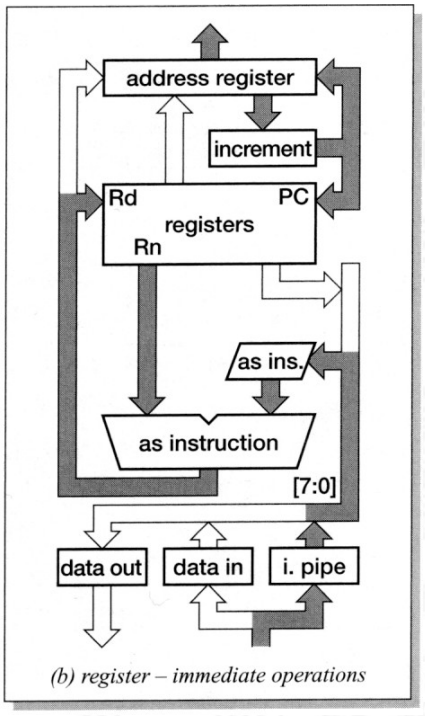
- Registerbank (inkl. Program Counter)
- Inkrementer
- Adress-Register
- ALU, Multiplizierer, Shifter
- Speicherinterface (Data-In / -Out)
- Steuerwerk



ARM-3 Datentransfer: Register-Operationen

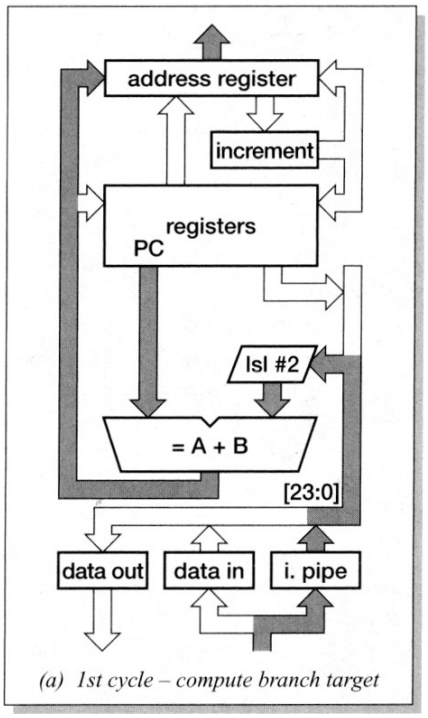


(a) register – register operations

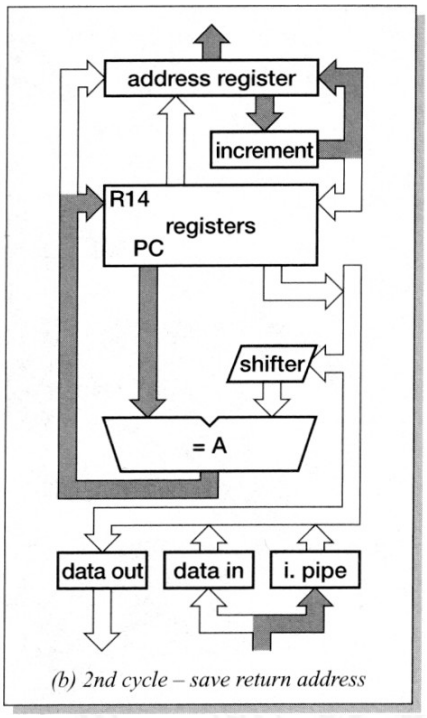


(b) register – immediate operations

ARM-3 Datentransfer: Funktionsaufruf/Sprungbefehl

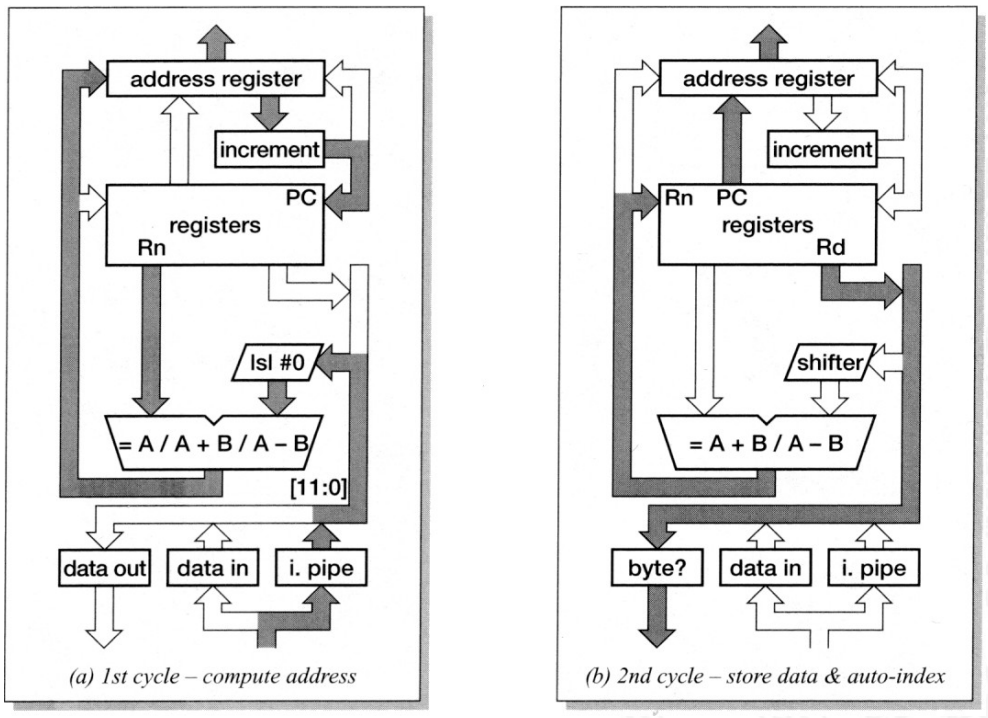


(a) 1st cycle – compute branch target



(b) 2nd cycle – save return address

ARM-3 Datentransfer: Store-Befehl



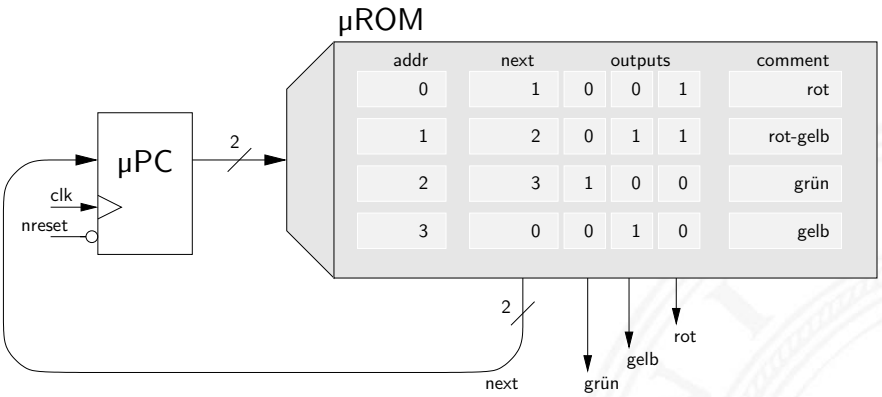
Ablaufsteuerung mit Mikroprogramm

- als Alternative zu direkt entworfenen Schaltwerken

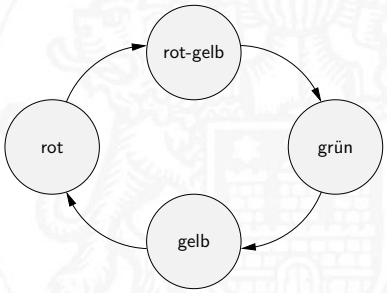
- *Mikroprogrammzähler* μPC: Register für aktuellen Zustand
- μPC adressiert den Mikroprogrammspeicher μROM
- μROM konzeptionell in mehrere Felder eingeteilt
  - die verschiedenen Steuerleitungen
  - ein oder mehrere Felder für Folgezustand
  - ggf. zusätzliche Logik und Multiplexer zur Auswahl unter mehreren Folgezuständen
  - ggf. Verschachtelung und Aufruf von Unterprogrammen: „nanoProgramm“

- siehe „Praktikum Rechnerstrukturen“

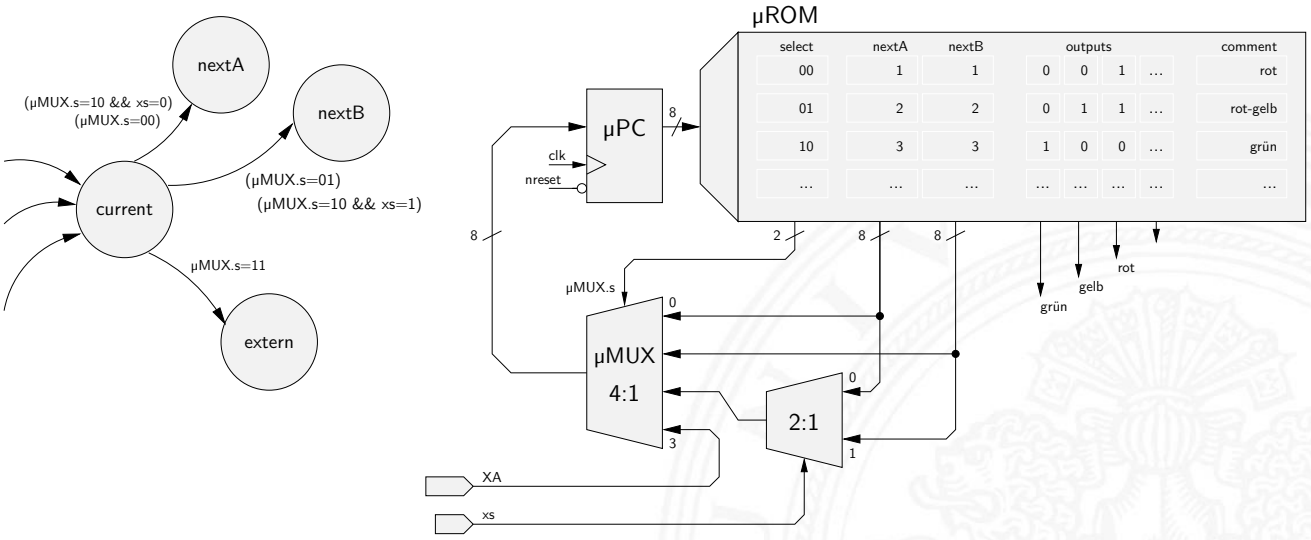
Mikroprogramm: Beispiel Ampel



- µPC adressiert das µROM
- "next"-Ausgang liefert den Folgezustand
  (Adresse 0: Wert 1, Adresse 1: Wert 2, usw)
- andere Ausgänge steuern die Schaltung
  (hier die Lampen der Ampel)

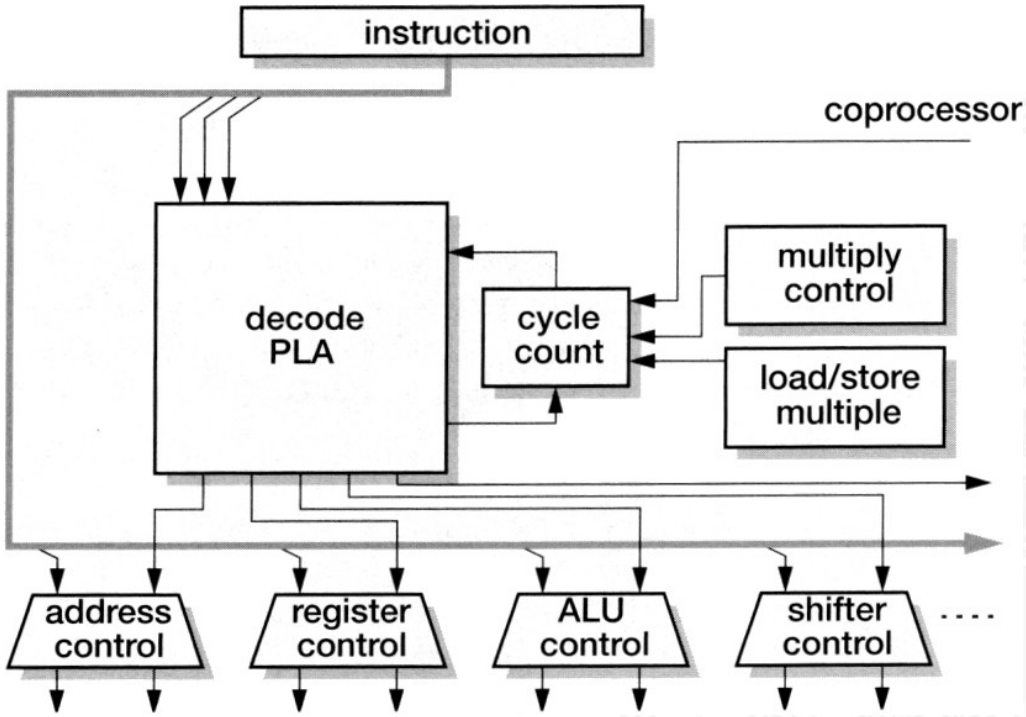


Mikroprogramm: Beispiel zur Auswahl des Folgezustands



- Mulitplexer erlaubt Auswahl des µPC Werts
- "nextA", "nextB" aus dem µROM, externer "XA" Wert
- "xs" Eingang erlaubt bedingte Sprünge

Mikroprogramm: Befehlsdecoder des ARM-7 Prozessors



Literatur: Quellen für die Abbildungen

- Andrew S. Tanenbaum,
  Computerarchitektur: Strukturen, Konzepte, Grundlagen,
  5. Auflage, Pearson Studium, 2006
- Steven Furber,
  ARM System-on-Chip Architecture,
  Addison-Wesley Professional, 2001
- Andreas Mäder,
  Vorlesung: Rechnerarchitektur und Mikrosystemtechnik,
  Universität Hamburg, FB Informatik, 2010
  `tams.informatik.uni-hamburg.de/lectures/2010ws/vorlesung/ram`

Gliederung

Gliederung (cont.)

Gliederung (cont.)

Erinnerung

Das **Konzept** des Digitalrechners (von-Neumann Prinzip) ist völlig unabhängig von der Technologie:

- mechanische Rechenmaschinen
- pneumatische oder hydraulische Maschinen
- Relais, Vakuumröhren, diskrete Transistoren
- molekulare Schaltungen
- usw.

Aber:

- nur hochintegrierte Halbleiterschaltungen („VLSI“) erlauben die billige Massenfertigung mit Milliarden von Komponenten
- **Halbleiter** und **Planarprozess** sind essentielle Basistechnologien

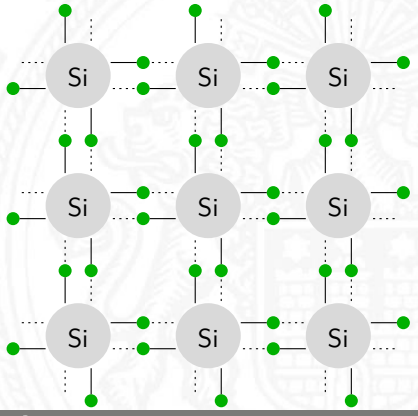
Halbleiter

Halbleiter stehen zwischen *Leitern* (z.B.: Metalle) und *Isolatoren*.

- bei Raumtemperatur quasi nicht-leitend
- Leitfähigkeit steigt mit der Temperatur ⇒ Heißleiter
- physikalische Erklärung über Bändermodell
  siehe `http://de.wikipedia.org/wiki/Halbleiter`

Kristallstruktur aus 4-wertigen Atomen

- elementare Halbleiter: Ge, Si
- Verbindungshalbleiter: GaAs, InSb

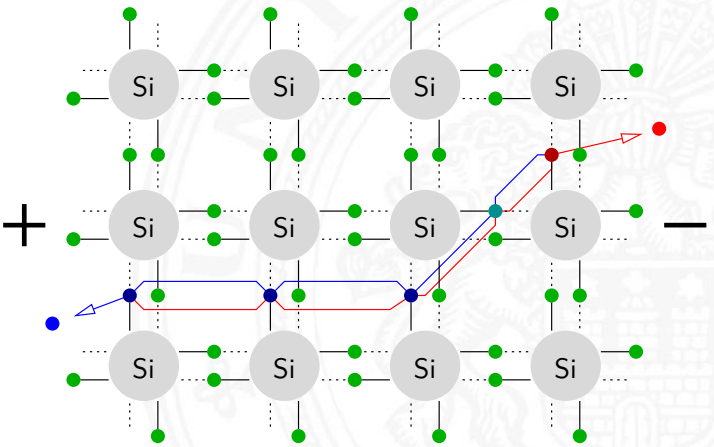


Leitung im undotierten Kristall

- Paarentstehung: Elektronen lösen sich aus Gitterverband
  Paar aus Elektron und „Loch“ entsteht.
- Rekombination: Elektronen und Löcher verbinden sich
  quasistatischer Prozess
- Eigenleitungsdichte n_i: temperatur- und materialabhängig

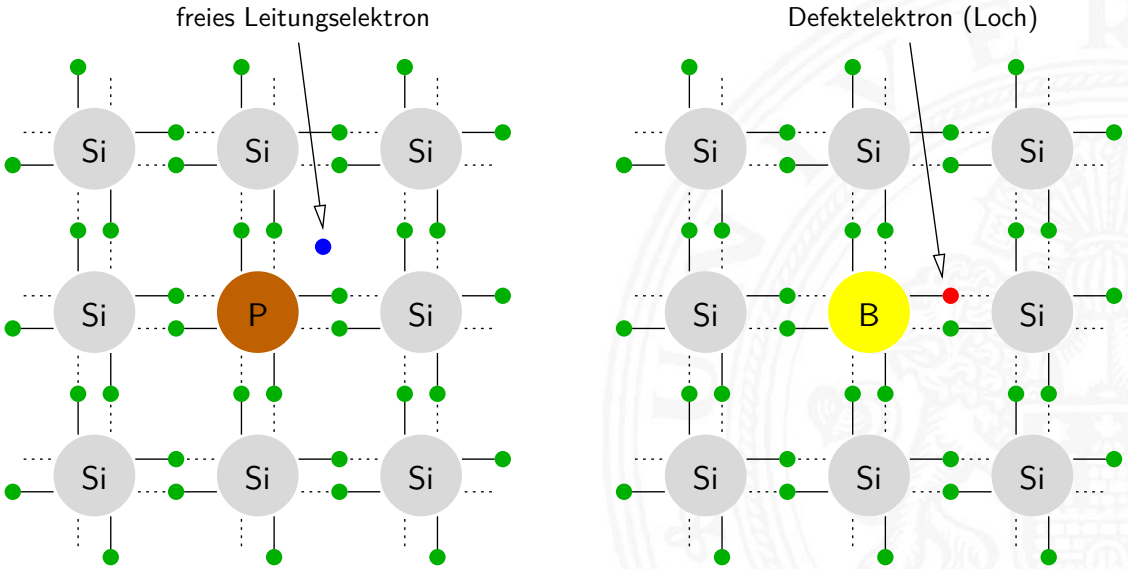
  Si : $1,2 \cdot 10^{10}\ cm^{-3}$
  Ge : $2,5 \cdot 10^{13}\ cm^{-3}$
  GaAs : $1,8 \cdot 10^{6}\ cm^{-3}$
  bei $300°K \approx 20°C$

  Atomdichte
  Si : $5 \cdot 10^{22}\ cm^{-3}$
- es gilt: $n_i^2 = n_n \cdot n_p$



Dotierung mit Fremdatomen

Ein kleiner Teil der vierwertigen Atome wird durch fünf- oder dreiwertige Atome ersetzt.



Dotierung mit Fremdatomen (cont.)

- Donatoren, Elektronenspender: Phosphor, Arsen, Antimon
- Akzeptoren: Bor, Aluminium, Gallium, Indium
- Dotierungsdichten

Stärke		Fremdatome $[cm^{-3}]$
schwach	n^-, p^-	$10^{15} \ldots 10^{16}$
mittel	n, p	$10^{16} \ldots 10^{19}$
stark	n^+, p^+	$10^{19} \ldots$

- Beweglichkeit μ: materialspezifische Größe

$T = 300°K$		Si	Ge	GaAs	$[cm^2/(Vs)]$
Elektronen	μ_n	1500	3900	8500	
Löcher	μ_p	450	1500	400	

- Leitfähigkeit: ergibt sich aus Material, Beweglichkeit und Ladungsträgerdichte(n)
  $K = e(n_n\mu_n + n_p\mu_p)$

Dotierung mit Fremdatomen (cont.)

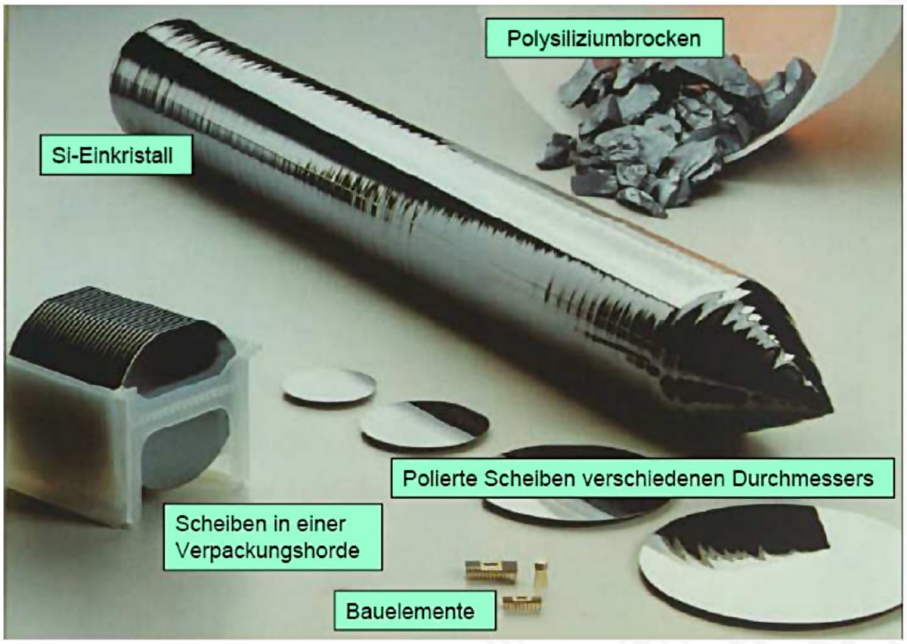
- selbst bei hoher Dotierung ist die Leitfähigkeit um Größenordnungen geringer als bei Metallen

  Si 1 freier Ladungsträger pro 500 Atome ($10^{19}/5 \cdot 10^{22}$)

  Met mindestens 1 Ladungsträger pro Atom
- Majoritätsträger: Ladungsträger in Überzahl (i.d.R. Dotierung)
  Minoritätsträger: Ladungsträger in Unterzahl
  $n_i^2 = n_n \cdot n_p$

Herstellung von Halbleitermaterial

Übersicht in: `http://de.wikipedia.org/wiki/Silicium`



Rohsilizium

- Siliziumoxid (SiO_2): Sand, Kies...
  ca. 20 % der Erdkruste
- Herstellung im Lichtbogenofen: Siliziumoxid + Koks
  $SiO_2 + 2C \rightarrow Si + 2CO$
- amorphe Struktur, polykristallin
- noch ca. 2 % Verunreinigungen (Fe, Al...)

Solarsilizium

Ziel: Fremdatome aus dem Silizium entfernen

1. Chemische Bindung des Siliziums
   $Si + 3HCl \rightarrow SiHCl_3 + H_2$
   Reaktion mit Salzsäure erzeugt

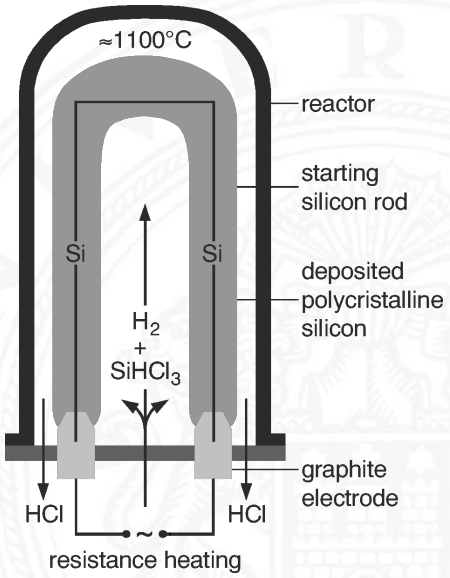
$SiHCl_3$	Trichlorsilan
$SiCl_4$	Siliziumchlorid (10%)
SiH_2Cl_4	div. andere Chlorsilane/Silane
$FeCl_2$, $AlCl_3$	div. Metallchloride

2. Verschiedene Kondensations- und Destillationsschritte trennen Fremdverbindungen ab, hochreines Trichlorsilan entsteht
   $<$ 1ppm Verunreinigungen

Solarsilizium (cont.)

3. CVD (Chemical Vapour Deposition) zur Abscheidung des Trichlorsilans zu elementarem Silizium
   $4SiHCl_3 \rightarrow Si + 3SiCl_4 + 2H_2$

$\Rightarrow$ polykristallines Silizium
$<$ 0,1ppm Verunreinigungen



Siliziumeinkristall

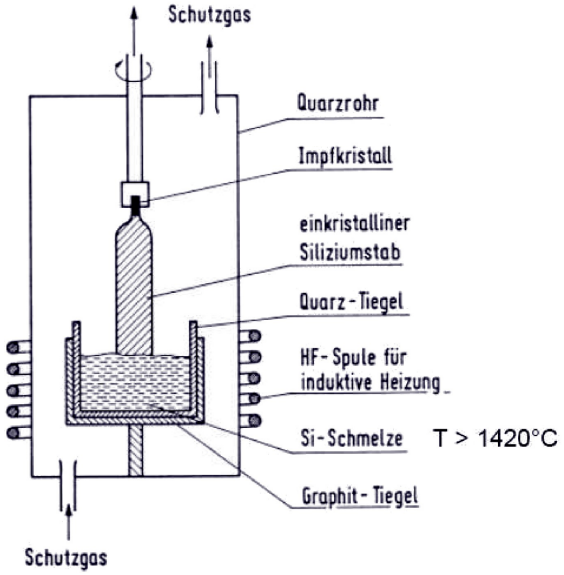
Weitere Ziele

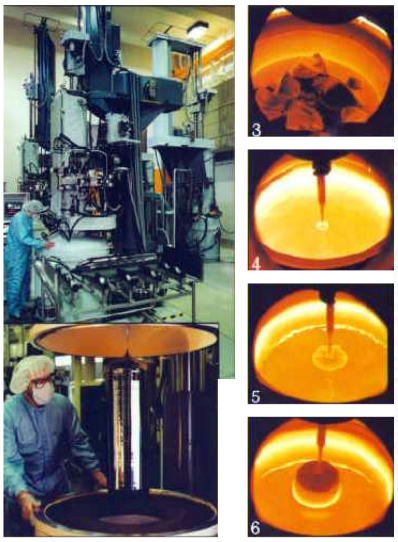
- Einkristalline Struktur erzeugen
- Reinheit für Halbleiterherstellung erhöhen
  $<, \ll$ 1ppb
- ggf. Dotierung durch Fremdatome einbringen

Es gibt dazu mehrere technische Verfahren, bei denen das polykristalline Silizium geschmolzen wird und sich monokristallin an einen Impfkristall anlagert.

Siliziumeinkristall (cont.)

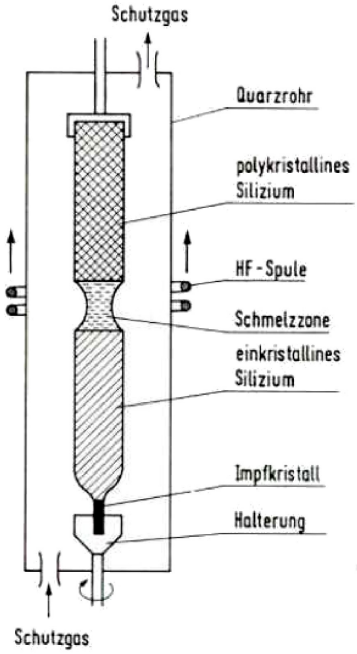
Czochralski-Verfahren (Tiegelziehverfahren)





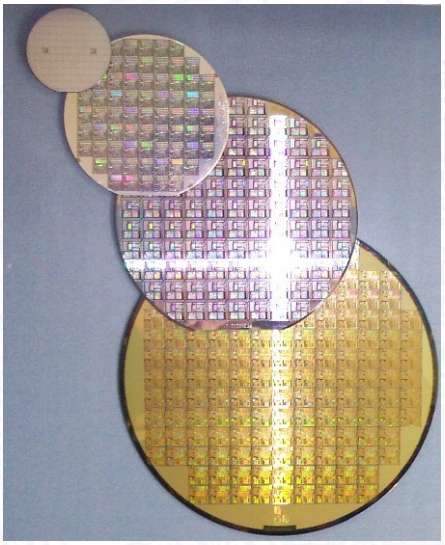
Siliziumeinkristall (cont.)

Zonenschmelz- / Zonenziehverfahren



Wafer

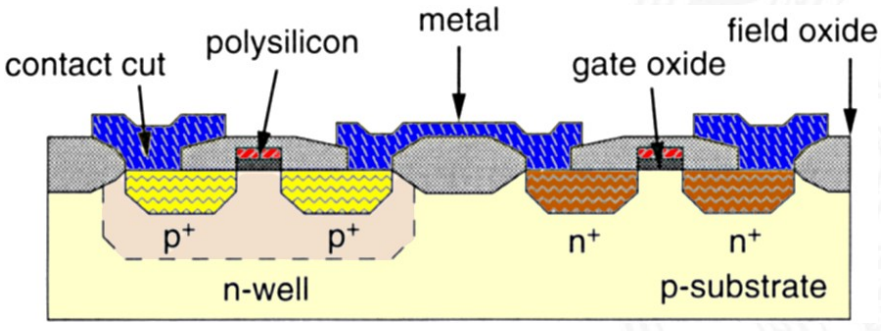
- weitere Bearbeitungsschritte:
  zersägen, schleifen, läppen, ätzen, polieren
- Durchmesser bis 30 cm
  2014: 45 cm
  Dicke < 1mm
  Rauhigkeit $\approx$ nm
- Markieren: Kerben, Lasercodes. . .
  früher „flats“



Technologien

Technologien zur Erstellung von Halbleiterstrukturen

- Epitaxie: Aufwachsen von Schichten
- Oxidation von Siliziumoberflächen: SiO_2 als Isolator
- Strukturerzeugung durch Lithografie
- Dotierung des Kristalls durch Ionenimplantation oder Diffusion
- Ätzprozesse: Abtragen von Schichten



Technologien (cont.)

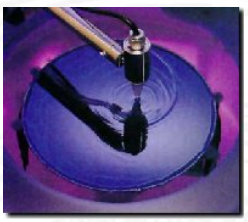
Links

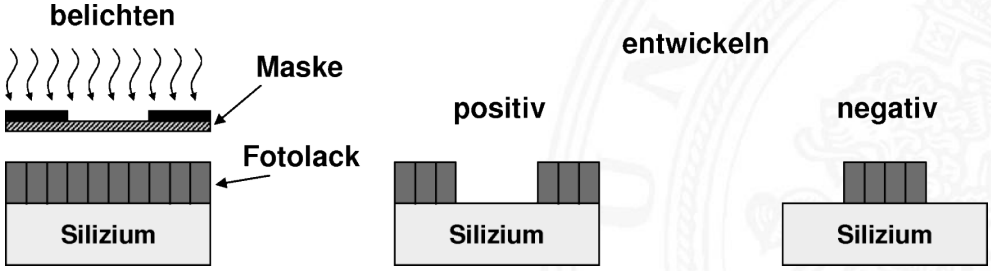
- http://www.halbleiter.org
- http://www.siliconfareast.com
- http://www2.renesas.com/fab/en
- http://en.wikipedia.org/wiki/Semiconductor_device_fabrication
- http://de.wikipedia.org/wiki/Halbleitertechnik

Lithografie

Übertragung von Strukturen durch einen Belichtungsprozess

1. Lack Auftragen (Aufschleudern)
   - Positivlacke: hohe Auflösung ⇒ MOS
   - Negativlacke: robust, thermisch stabil





Lithografie (cont.)

2. „Belichten“
   - Maskenverfahren: 1:1 Belichtung, Step-Verfahren
     UV-Lichtquelle
   - Struktur direkt schreiben: Elektronen- / Ionenstrahl
   - andere Verfahren: Röntgenstrahl- / EUV-Lithografie
3. Entwickeln, Härten, Lack entfernen
   - je nach Lack verschiedene chemische Reaktionsschritte
   - Härtung durch Temperatur

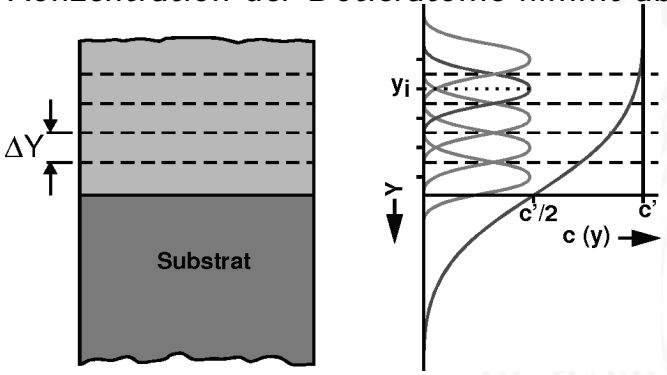
. . . weitere Schritte des Planarprozess

Dotierung

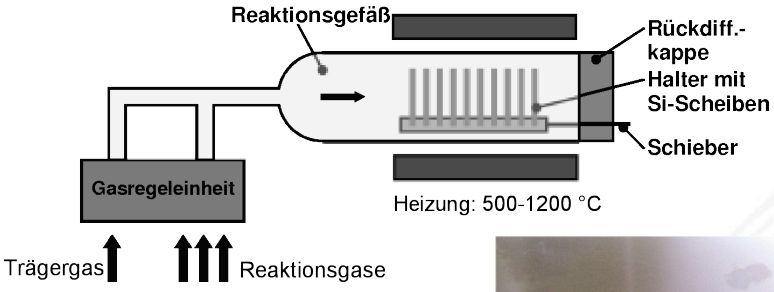
Fremdatome in den Siliziumkristall einbringen

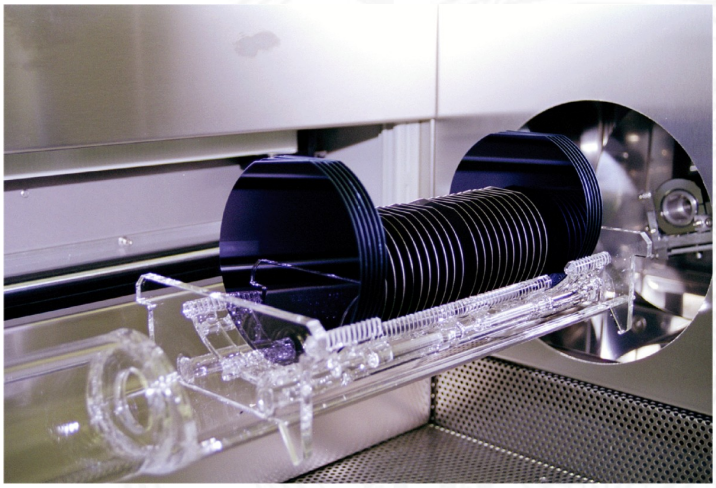
- Diffusion
  - Diffusionsofen
  - gaußförmiges Dotierungsprofil

  Konzentration der Dotieratome nimmt ab



Dotierung (cont.)



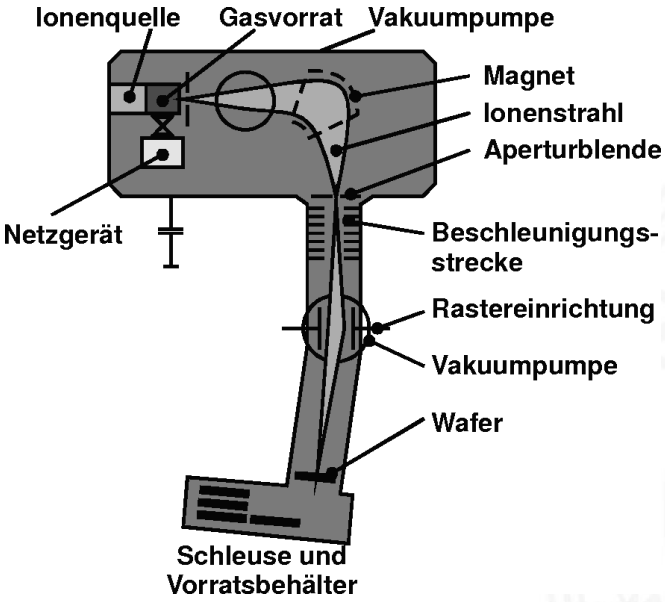


Dotierung (cont.)

- Ionenimplantation
  - „Beschuss“ mit Ionen
  - Beschleunigung der Ionen im elektrischen Feld
  - Über die Energie der Ionen kann die Eindringtiefe sehr genau eingestellt werden
  - „Temperung“ notwendig: Erhitzen des Einkristalls zur Neuorganisation des Kristallgitters

Dotierung (cont.)

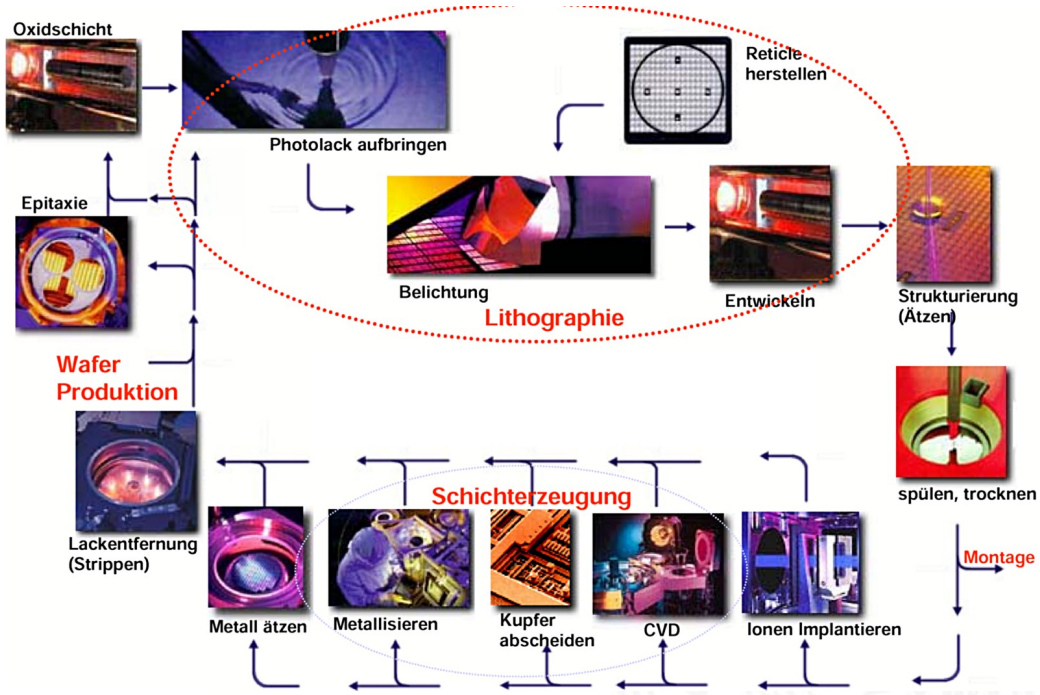
Ionenimplantation



Planarprozess

- Der zentrale Ablauf bei der Herstellung von Mikroelektronik
- Ermöglicht die gleichzeitige Fertigung aller Komponenten auf dem Wafer
- Schritte
  1. Vorbereiten / Beschichten des Wafers: Oxidation, CVD, Aufdampfen, Sputtern...
  2. Strukturieren durch Lithografie
  3. Übertragen der Strukturen durch Ätzprozesse
  4. Modifikation des Materials: Dotierung, Oxidation
  5. Vorbereitung für die nächsten Prozessschritte...

Planarprozess (cont.)



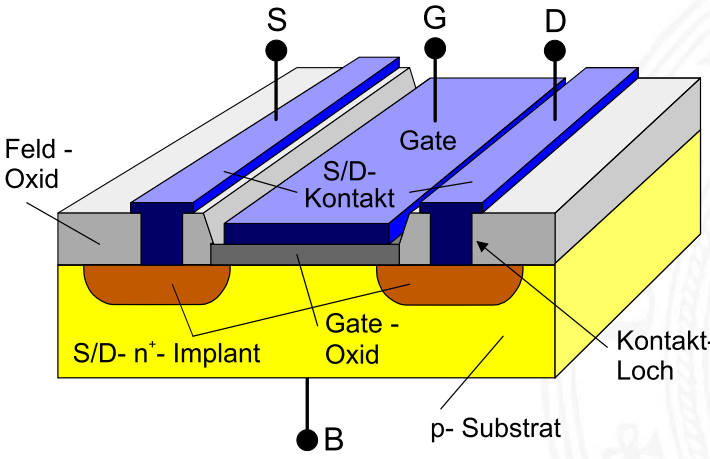
MOS-Transistor

- MOS: Metal Oxide Semiconductor
  FET : Feldeffekttransistor
  - `http://olli.informatik.uni-oldenburg.de/weTEiS/weteis/tutorium.htm`
  - `http://de.wikipedia.org/wiki/Feldeffekttransistor`
  - `http://de.wikipedia.org/wiki/MOSFET`
- unipolarer Transistor: nur eine Art von Ladungsträgern, die Majoritätsträger, ist am Stromfluss beteiligt

  im Gegensatz zu Bipolartransistoren
  siehe z.B.: U. Tietze, C. Schenk, *Halbleiter-Schaltungstechnik*

MOS-Transistor (cont.)

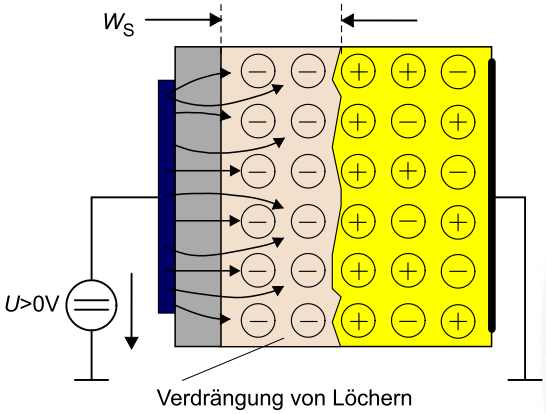
- Anschlüsse: **S**ource Quelle der Ladungsträger
  Gate steuert den Stromfluss
  Drain Senke der Ladungsträger
  Bulk siehe „Herstellungstechnik“



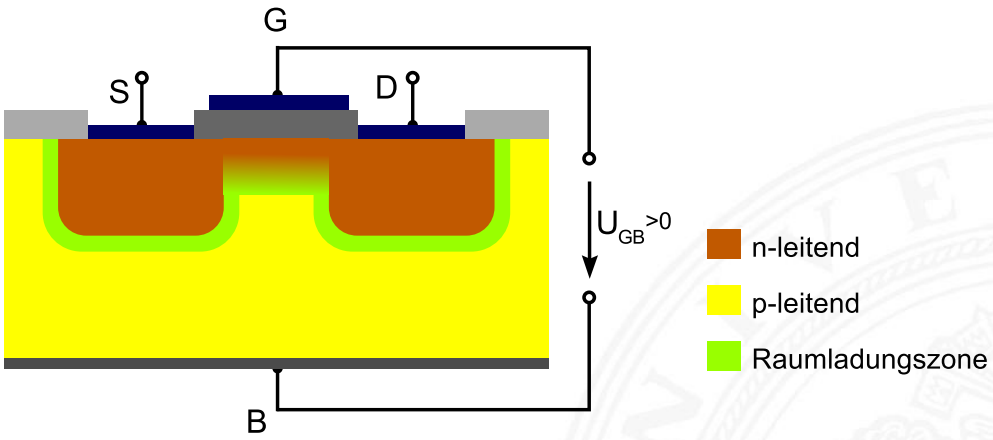
N. Reifschneider,
CAE-gestützte IC-Entwurfsmethoden

MOS-Transistor (cont.)

- Funktionsweise: die Ladung des Gates erzeugt ein elektrisches Feld. Durch Inversion werden Ladungsträger unterhalb des Gates verdrängt und ein leitender Kanal zwischen Source und Drain entsteht.



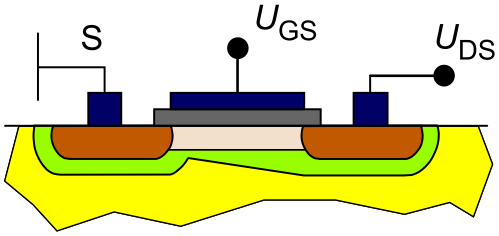
MOS-Transistor (cont.)



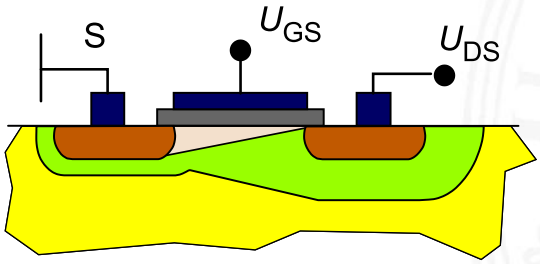
- Raumladungszone: neutral, keine freien Ladungsträger
- Schwellspannung U_P: abhängig von der Dotierungsdichte, den Parametern des MOS-Kondensators (Dicke und Material der Gate-Isolationsschicht)...
  U_P möglichst klein: 0,3...0,8 V früher: deutlich mehr

MOS-Transistor (cont.)

- $U_{DS} \ll U_{GS} - U_P$ normaler Betrieb (Triodenbereich)

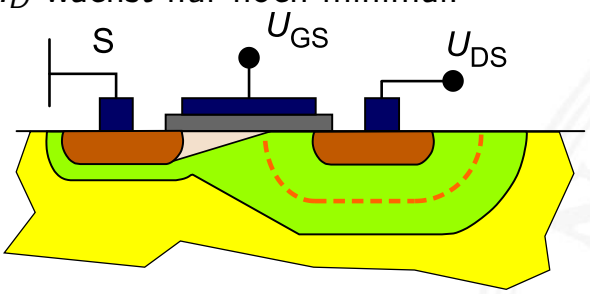


- $U_{DS} = U_{GS} - U_P$ Kanalabschnürung
  Spannungsabfall zwischen S und D durch den Kanalwiderstand



MOS-Transistor (cont.)

- $U_{DS} > U_{GS} - U_P$ Kanalverkürzung (Sättigungsbereich)
  Der Kanal wird weiter verkürzt, die Spannung U_{DS} bewirkt ein virtuell größeres Drain durch Inversion.
  I_D wächst nur noch minimal.

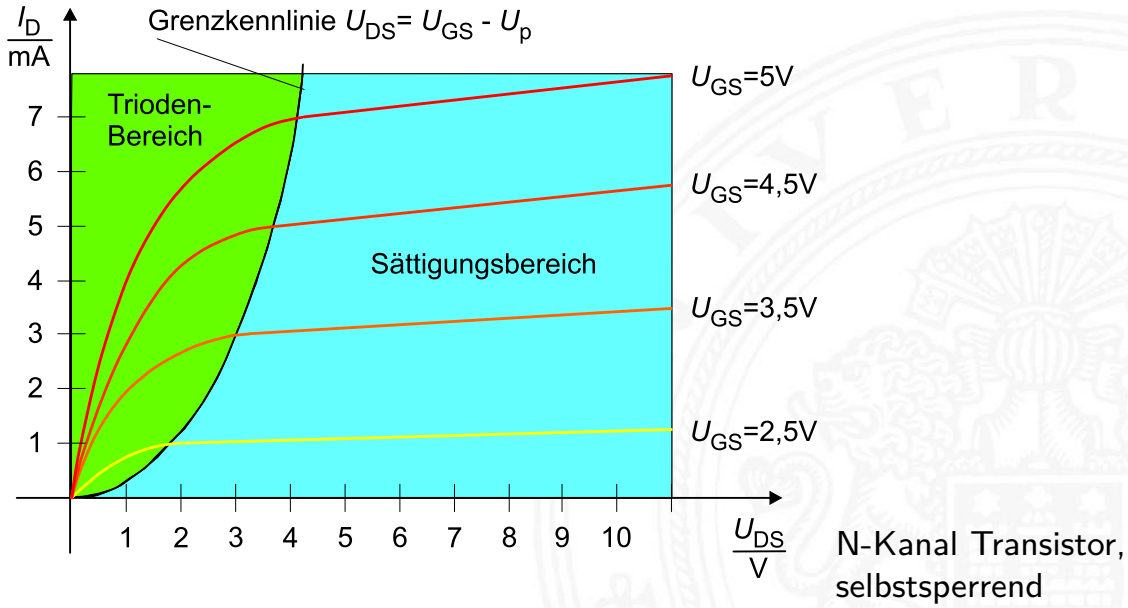


⇒ kurze Kanäle aktueller Submikronprozesse können allein durch hohe Spannungen U_{DS} leitend werden (Durchgreifbetrieb)

⇒ einer der Gründe für sinkende Versorgungsspannungen

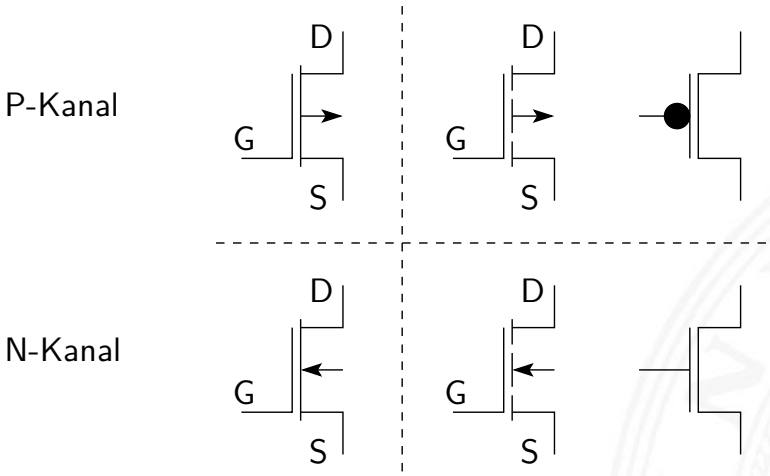
MOS-Transistor (cont.)

- Kennlinienfeld



N-Kanal Transistor, selbstsperrend

MOS-Transistor: Schaltsymbole



CMOS-Technologie

Complementary Metal-Oxide Semiconductor: die derzeit dominierende Techologie für alle hochintegrierten Schaltungen

- Schaltungsprinzip nutzt n-Kanal und p-Kanal Transistoren
- alle elementaren Gatter verfügbar
- effiziente Realisierung von *Komplexgattern*
- *Transmission-Gate* als elektrischer Schalter
- effiziente Realisierung von Flipflops und Speichern

\+ sehr hohe Integrationsdichte möglich, gut skalierbar
\+ sehr schnelle Schaltgeschwindigkeit der Gatter
\+ sehr geringer Stromverbrauch pro Gatter möglich
\+ Integration von digitalen und analogen Komponenten

CMOS: Überblick

- Schaltungsprinzip
- Inverter und nicht-invertierender Verstärker
- NAND, NAND3, NOR (und AND, OR)
- XOR

- Komplexgatter
- Transmission-Gate

- Beispiele für Flipflops
- SRAM

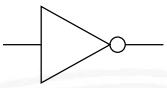
CMOS: Schaltungsprinzip von „static CMOS“

- Transistoren werden als Schalter betrachtet
- zwei zueinander **komplementäre** Zweige der Schaltung
  - n-Kanal Transistoren zwischen Masse und Ausgang y — *1 on*
  - p-Kanal –"– Vdd und Ausgang y — *0 on*

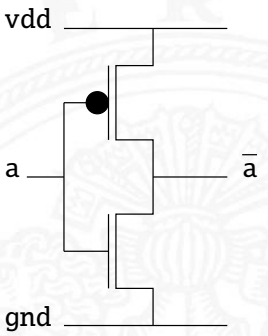
  - p-Kanal Zweig komplementär („dualer Graph“) zu n-Kanal Zweig: jede Reihenschaltung von Elementen wird durch eine Parallelschaltung ersetzt (und umgekehrt)

- immer ein direkt leitender Pfad von entweder Vdd („1“) oder Masse / Gnd („0“) zum Ausgang
- niemals ein direkt leitender Pfad von Vdd nach Masse
- kein statischer Stromverbrauch im Gatter

Inverter



Funktionsweise



- selbstsperrende p- und n-Kanal Transistoren
- komplementär beschaltet
- Ausgang: Pfad über p-Transistoren zu *Vdd*
  –"– n-Transistoren zu *Gnd*
- genau *einer* der Pfade leitet

- Eingang $\quad \text{Trans}_P \quad \text{Trans}_N \quad$ Ausgang

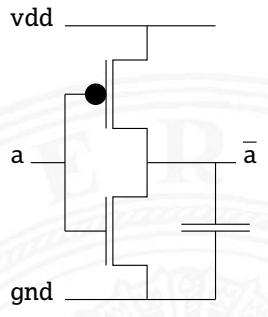
  $a = 0 \rightarrow$ leitet / sperrt $\rightarrow$ über T_P mit *Vdd* verbunden $= 1$

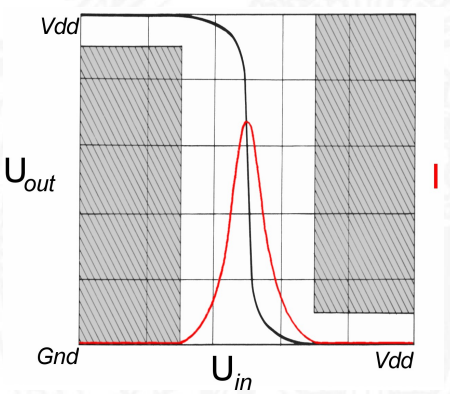
  $a = 1 \rightarrow$ sperrt / leitet $\rightarrow$ über T_N mit *Gnd* verbunden $= 0$

Inverter (cont.)

Leistungsaufnahme

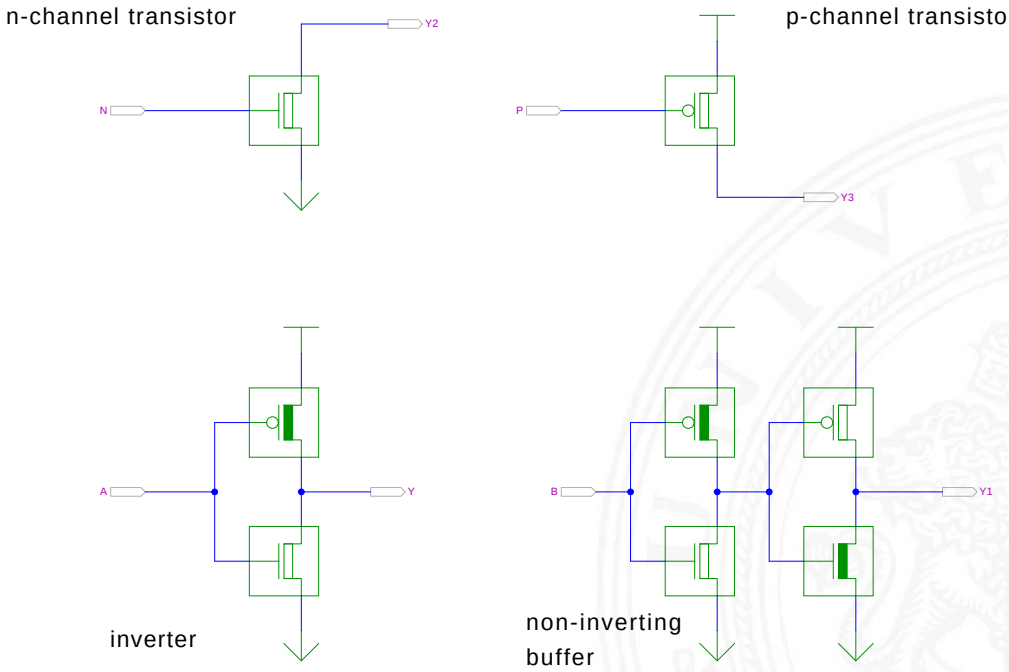
1. $U_{in} = 0$, bzw. *Vdd*: Sperrstrom, nur μA
   $\Rightarrow$ niedrige statische Leistungsaufnahme
2. Querstrom beim Umschalten:
   kurzfristig leiten beide Transistoren
   $\Rightarrow$ Forderung nach steilen Flanken
3. Kapazitive Last: Fanout-Gates
   Energie auf Gate(s): $W = \frac{1}{2} C_T \, Vdd^2$
   Verlustleistung$_{(0/1/0)}$: $P = C_T \, Vdd^2 \cdot f$





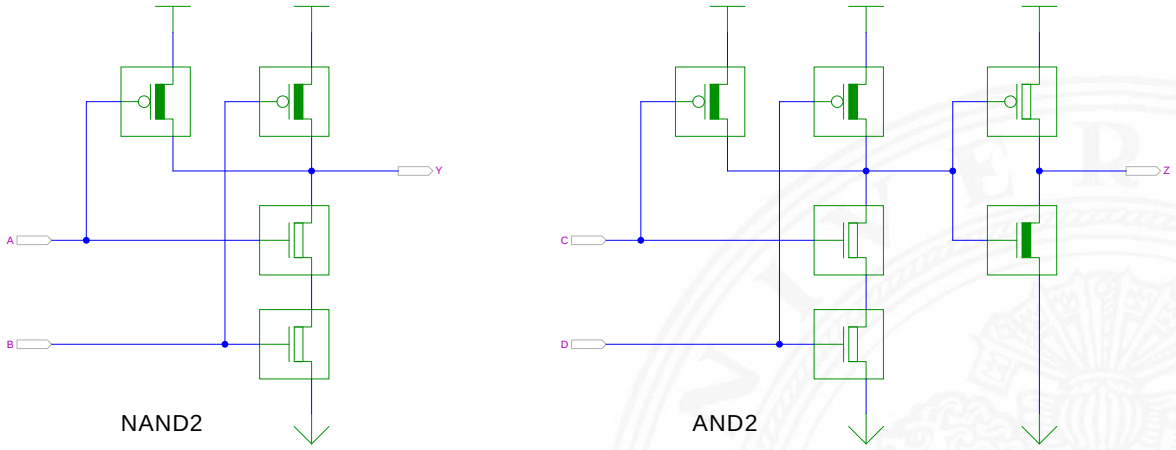
Transfercharakteristik

Hades: n- und p-Kanal Transistor, Inverter, Verstärker



$Y = \neg A$ $\quad Y = \neg\neg A = A$

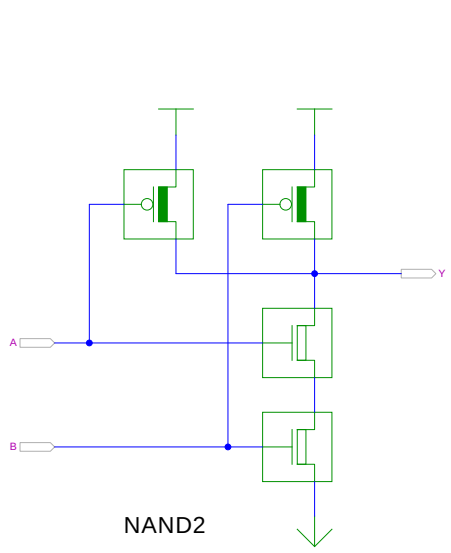
NAND- und AND-Gatter



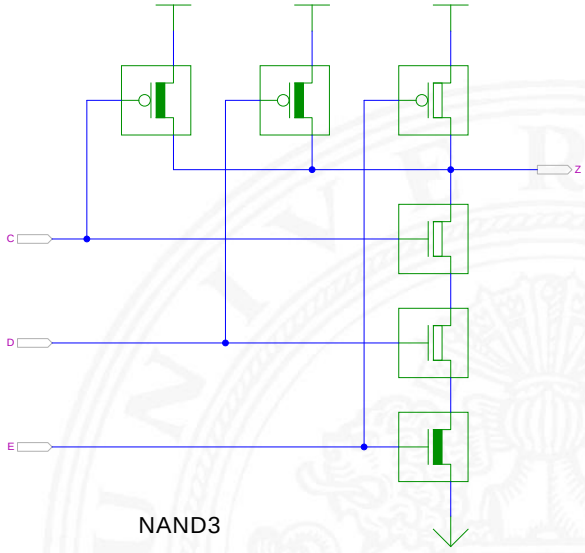
- NAND: n-Transistoren in Reihe, p-Transistoren parallel
- AND: Kaskade aus NAND und Inverter

tams.informatik.uni-hamburg.de/applets/hades/webdemos/05-switched/40-cmos

NAND- und AND-Gatter (cont.)



NAND2

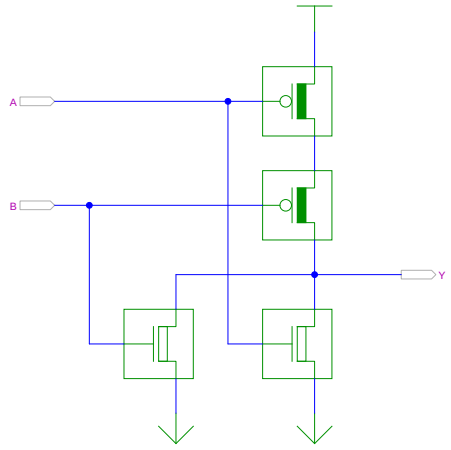


NAND3

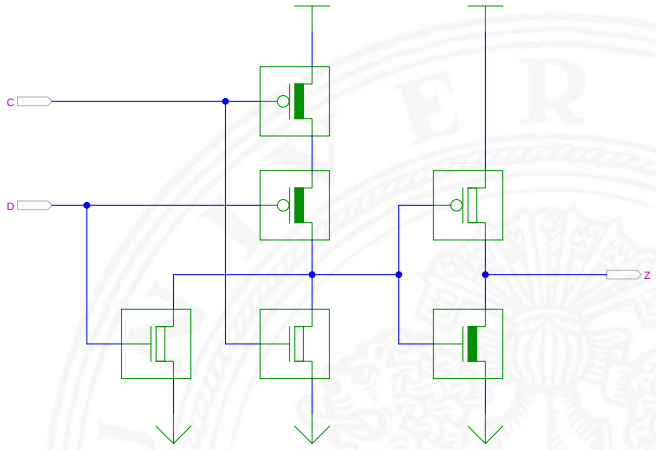
- n-Transistoren in Reihe, p-Transistoren parallel
- normalerweise max. 4 Transistoren in Reihe (Spannungsabfall)

NOR- und OR-Gatter

NOR2

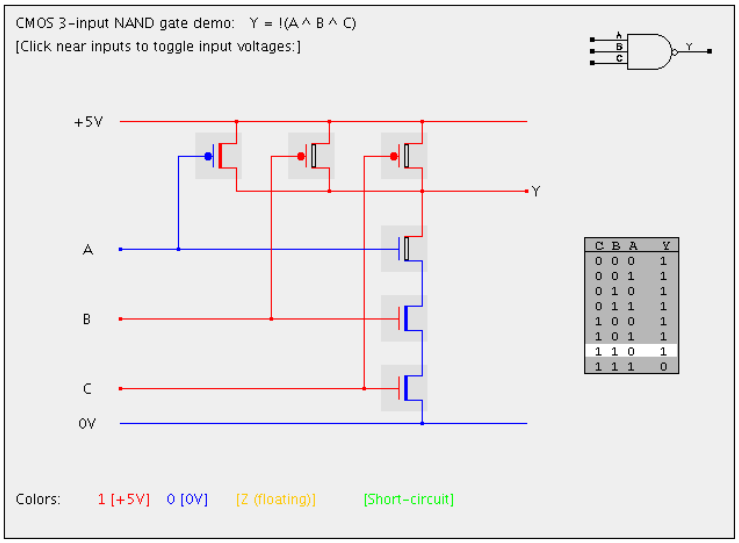


OR2



- Struktur komplementär zum NAND/AND
- n-Transistoren parallel, p-Transistoren in Reihe
- p-Transistoren schalten träge: etwas langsamer als NAND

CMOS-Technologie: Demos



- Interaktive Demonstration der CMOS-Grundgatter (Java)
  `http://tams.informatik.uni-hamburg.de/applets/cmos/`

CMOS: Komplexgatter

Gatterfunktionen

- Schaltungen: *negierte monotone boole'sche Funktionen*
- Beliebiger schaltalgebraischer Ausdruck *ohne Negation*: $\vee, \wedge$
- Negation des gesamten Ausdrucks: Ausgang *immer* negiert
- je Eingang: ein Paar p-/n-Kanal Transistoren
- Dualitätsprinzip: n- und p-Teil des Gatters

n-Teil	p-Teil	Logik, ohne Negation
seriell	$\Leftrightarrow$ parallel	$\equiv \wedge$ / und
parallel	$\Leftrightarrow$ seriell	$\equiv \vee$ / oder

CMOS: Komplexgatter (cont.)

- Konstruktion
  1. n-Teil aus Ausdruck ableiten
     beliebige Parallel- und Serienschaltung der n-Transistoren
  2. p-Teil dual dazu entwickeln
     komplementäre Seriell- und Parallelschaltung der p-Transistoren
     - typischerweise max. 4 Transistoren in Reihe
- viele invertierende logische Funktionen effizient realisierbar
- Schaltungslayout automatisch synthetisierbar
- zwei gängige Varianten
  - AOI-Gatter („AND-OR-invert“)
  - OAI-Gatter („OR-AND-invert“)

Komplexgatter

Beispiel: $\overline{(a \wedge b \wedge c) \vee d \vee (e \wedge f)}$

„AOI321-Gatter“, AND-OR-INVERT Struktur

- AND-Verknüpfung von (a,b,c)
- AND-Verknüpfung von (e,f)
- NOR-Verknüpfung der drei Terme

- direkte Realisierung hätte (6+2)+(0)+(4+2)+6 Transistoren
- Komplexgatter mit 12 Transistoren

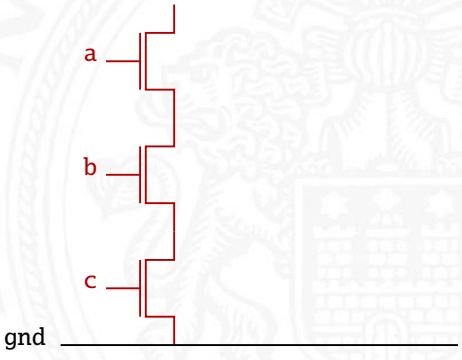
Komplexgatter (cont.)

Beispiel: $\overline{(a \wedge b \wedge c) \vee d \vee (e \wedge f)}$

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
16.2.2 VLSI-Entwurf und -Technologie - CMOS-Schaltungen - Komplexgatter
64-040 Rechnerstrukturen

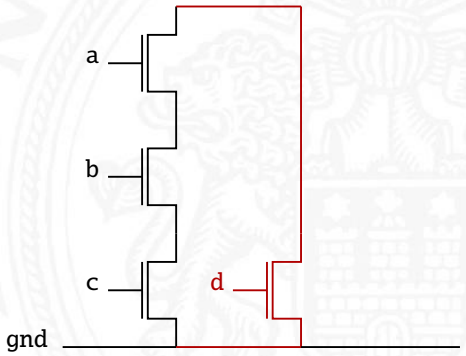
Komplexgatter (cont.)

Beispiel: $\overline{(a \wedge b \wedge c) \vee d \vee (e \wedge f)}$



Komplexgatter (cont.)

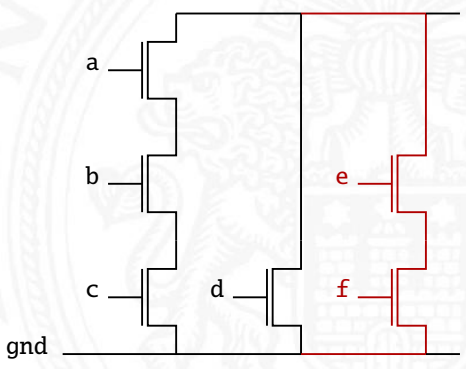
Beispiel: $\overline{(a \wedge b \wedge c) \vee d \vee (e \wedge f)}$



Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
16.2.2 VLSI-Entwurf und -Technologie - CMOS-Schaltungen - Komplexgatter
64-040 Rechnerstrukturen

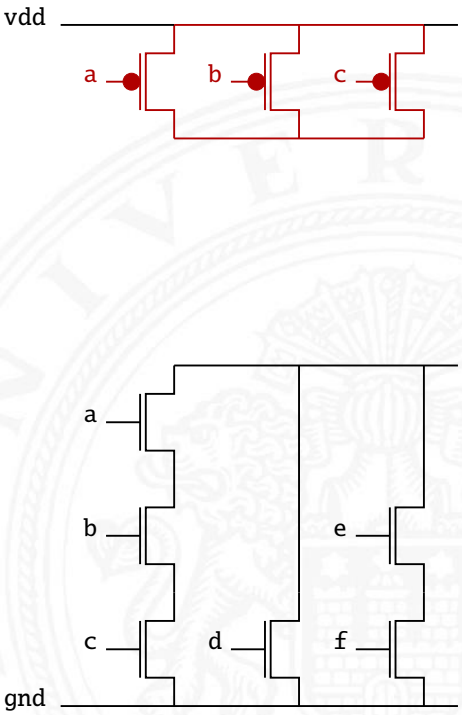
Komplexgatter (cont.)

Beispiel: $\overline{(a \wedge b \wedge c) \vee d \vee (e \wedge f)}$



Komplexgatter (cont.)

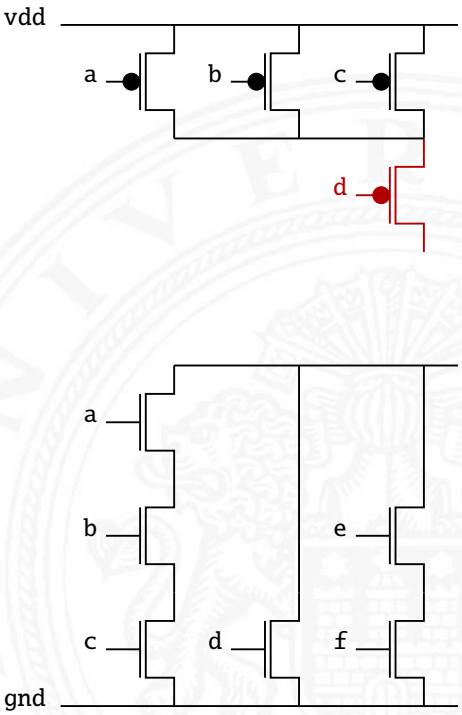
Beispiel: $\overline{(a \wedge b \wedge c) \vee d \vee (e \wedge f)}$



Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
16.2.2 VLSI-Entwurf und -Technologie - CMOS-Schaltungen - Komplexgatter
64-040 Rechnerstrukturen

Komplexgatter (cont.)

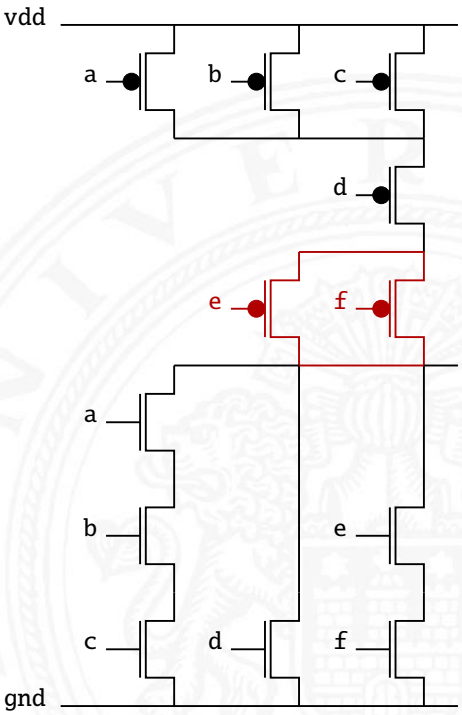
Beispiel: $\overline{(a \wedge b \wedge c) \vee d \vee (e \wedge f)}$



A. Mäder
846

Komplexgatter (cont.)

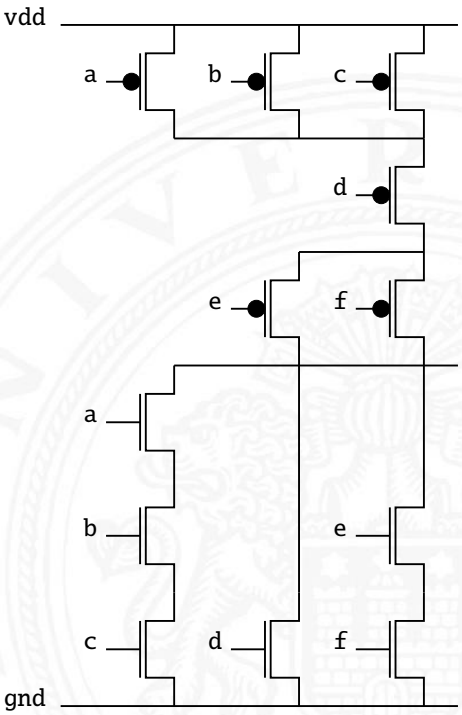
Beispiel: $\overline{(a \wedge b \wedge c) \vee d \vee (e \wedge f)}$



Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
16.2.2 VLSI-Entwurf und -Technologie - CMOS-Schaltungen - Komplexgatter
64-040 Rechnerstrukturen

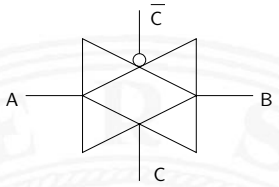
Komplexgatter (cont.)

Beispiel: $\overline{(a \wedge b \wedge c) \vee d \vee (e \wedge f)}$



Transmission-Gate

Transmissions-Gatter (*transmission gate*, *t-gate*)



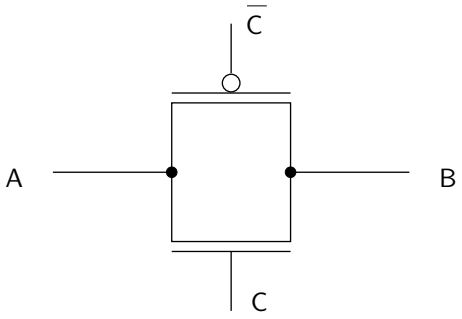
- Paar aus je einem n- und p-Kanal MOS-Transistor
- symmetrische Anordnung

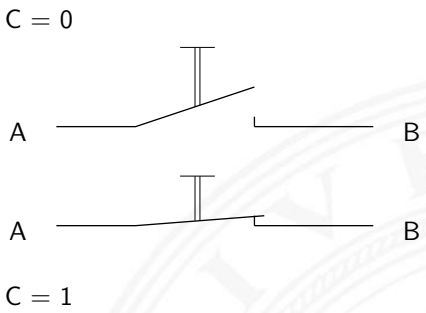
- Ansteuerung der Gate-Elektroden mit invertierter Polarität

$\Rightarrow$ entweder beide Transistoren leiten, oder beide sperren

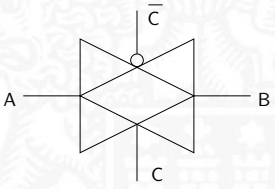
- Funktion entspricht **elektrisch gesteuertem Schalter**
- effiziente Realisierung vieler Schaltungen

Transmission-Gate (cont.)



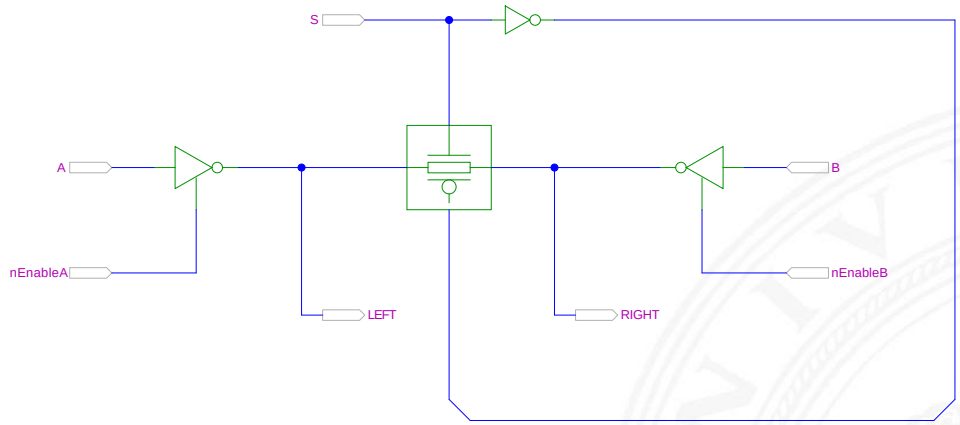


elektrisch gesteuerter Schalter:



- $C = 0$: keine Verbindung von A nach B
  $C = 1$: leitende Verbindung von A nach B
- symmetrisch in beide Richtungen

Transmission-Gate: Demo

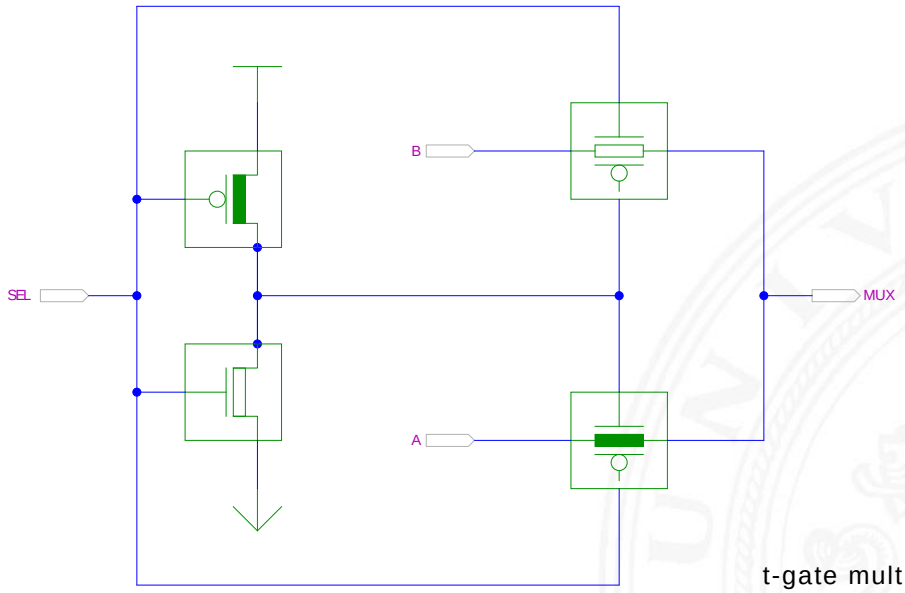


tgate demonstration

- Werte A und B anlegen, Treiber mit enable-Signalen aktivieren
- Gatter mit S ein- oder ausschalten

tams.informatik.uni-hamburg.de/applets/hades/webdemos/05-switched/40-cmos/tgate.html

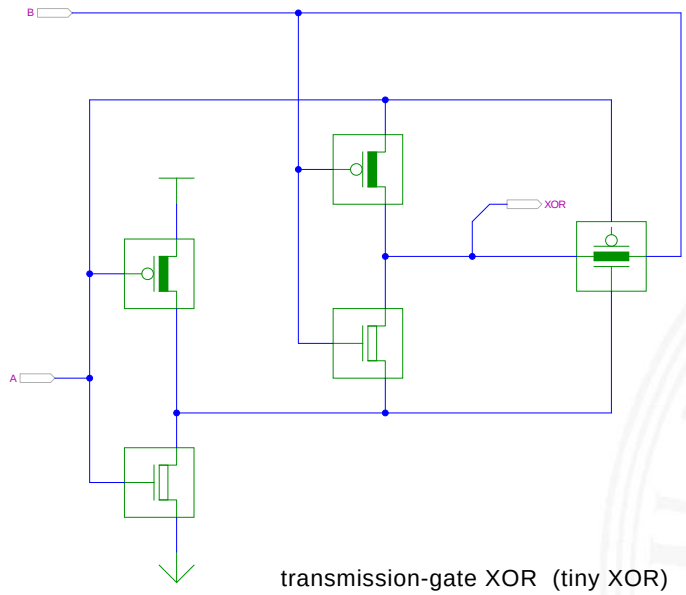
T-Gate Multiplexer



t-gate multiplexer

- kompakte Realisierung (4 bzw. 6 Transistoren)
- Eingänge a und b nicht verstärkt $\Rightarrow$ nur begrenzt kaskadierbar

T-Gate XOR-Gatter



transmission-gate XOR (tiny XOR)

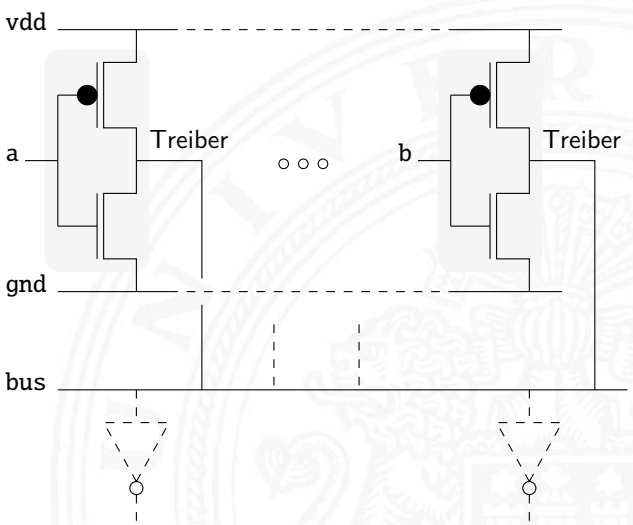
- kompakte Realisierung des XOR (nur 6 Transistoren)
- Eingang b nicht verstärkt $\Rightarrow$ nur begrenzt kaskadierbar

Tristate-Treiber

Bussysteme

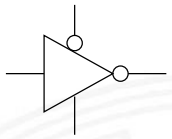
- Quellen: „Bustreiber“
- Senken : Gattereingänge
- Probleme
  - *Kurzschluss*
  - *offene Eingänge*

$\Rightarrow$ Tristate



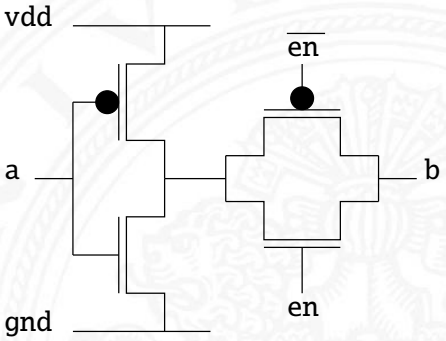
Tristate-Treiber (cont.)

Beispiel: Tristate-Inverter



Funktionsweise

- Ausgang elektrisch trennen
  z.B. mit Transmission-Gate
- 3-Pegel: 0, 1, Z *hochohmig*

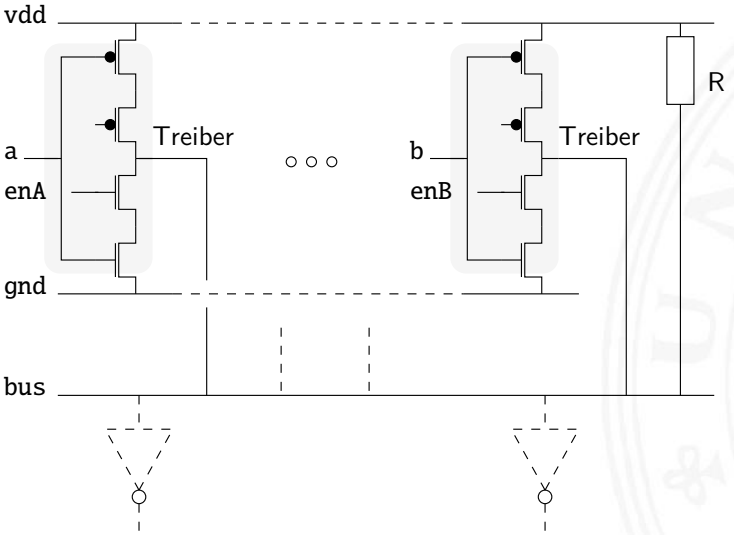


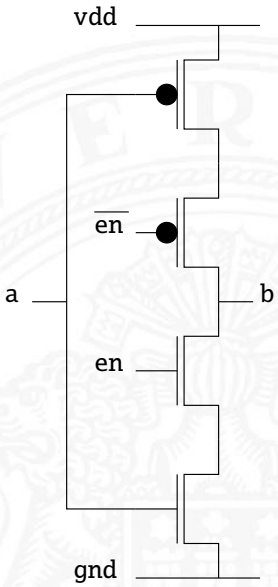
Enable	Verbindung	Ausgang	
$en = 0$	$\rightarrow$ getrennt	$\rightarrow bus = Z$	hochohmig
$en = 1$	$\rightarrow$ geschlossen	$\rightarrow bus = \neg a$	$f(a)$

Tristate-Treiber (cont.)

Tristate-Bussystem

- *pull-up/-down* Widerstand R (offene Eingänge)
- nur genau ein Treiber gleichzeitig aktiv





Latch / Flipflop: Speichertechnik

Methoden der Implementation

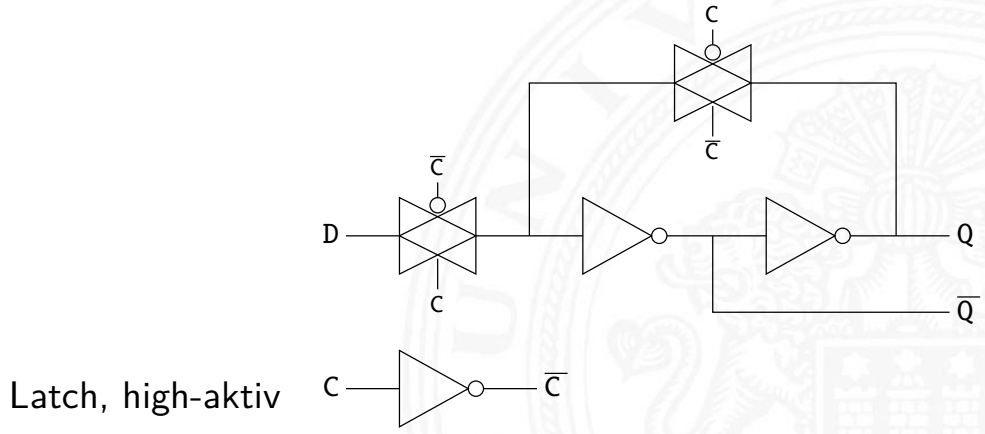
1. statisch
   - Speicherung: Rückkopplung von (statischen) Gattern
     siehe: „Schaltwerke – Flipflops“
   + taktunabhängig
   + sicher
2. quasi-statisch
   - Speicherung: Rückkopplung von Gattern
   - Transmission-Gates als Multiplexer
   + taktunabhängig
   + kleiner

Latch / Flipflop: Speichertechnik (cont.)

3. dynamisch
   - Speicherung: Gate-Kapazitäten
   - verschiedene Taktschemata/Schaltungsvarianten
   – muss getaktet werden
   – schwieriger zu Entwerfen (wegen Taktschema)
   + Integration in Datenpfade (arithmetische Pipelines)
   + sehr hohe Taktfrequenzen
   + sehr klein

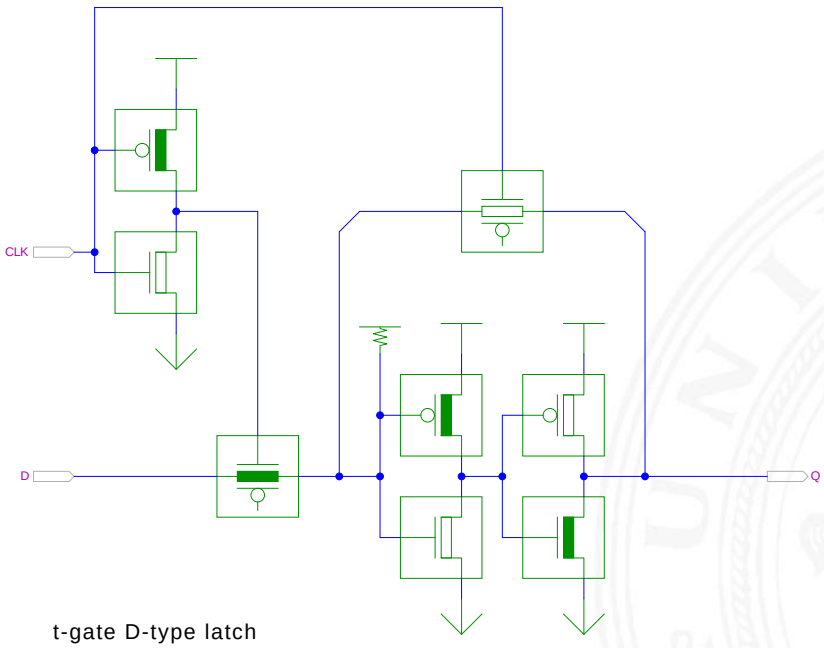
D-Latch: quasi-statisch

- Transmission-Gates als Schalter
  $C = 1$ Transparent: Eingang über die Inverter zum Ausgang
  $C = 0$ Speicherung: Rückkopplungspfad aktiv



Latch, high-aktiv

D-Latch: quasi-statisch (cont.)

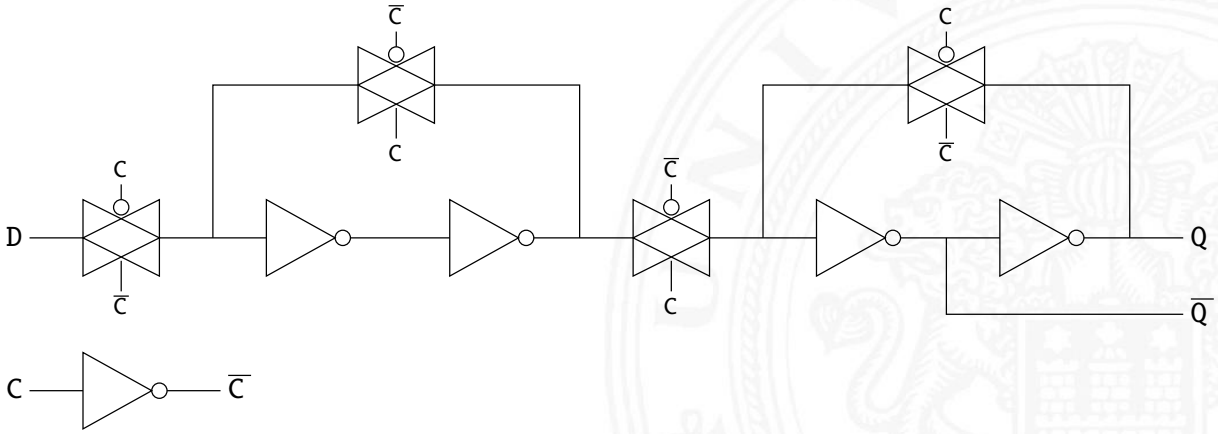


t-gate D-type latch

D-Flipflop: quasi-statisch

- Aufbau aus zwei Latches
- Vorderflanke: low-Transparent + high-Transparent

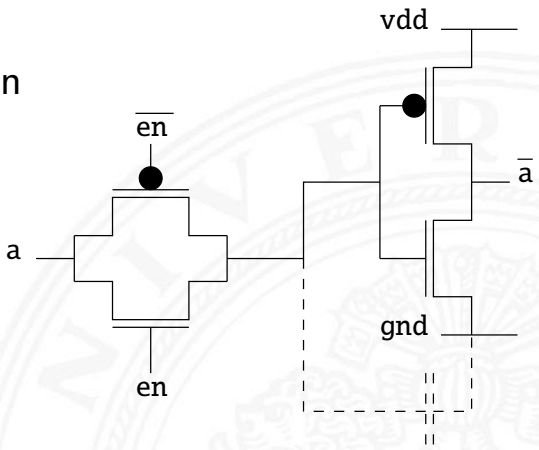
D-FF, vorderflankengest.



dynamische Speicherung

Schaltungsprinzip

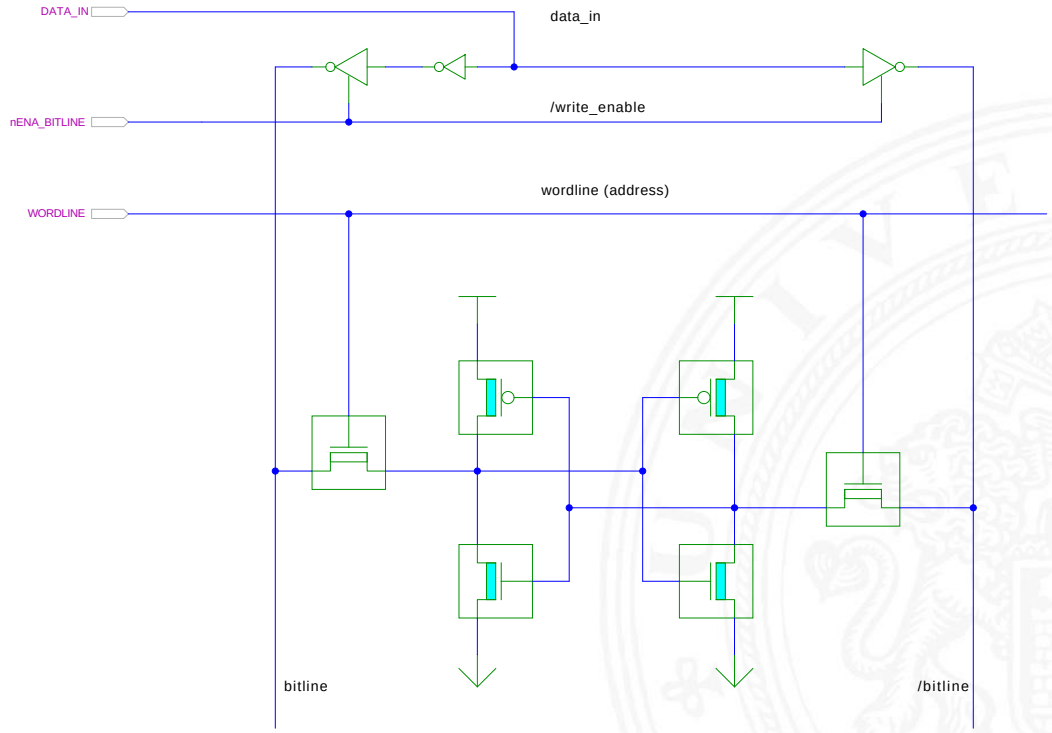
- Speicherung auf Gate-Kapazitäten



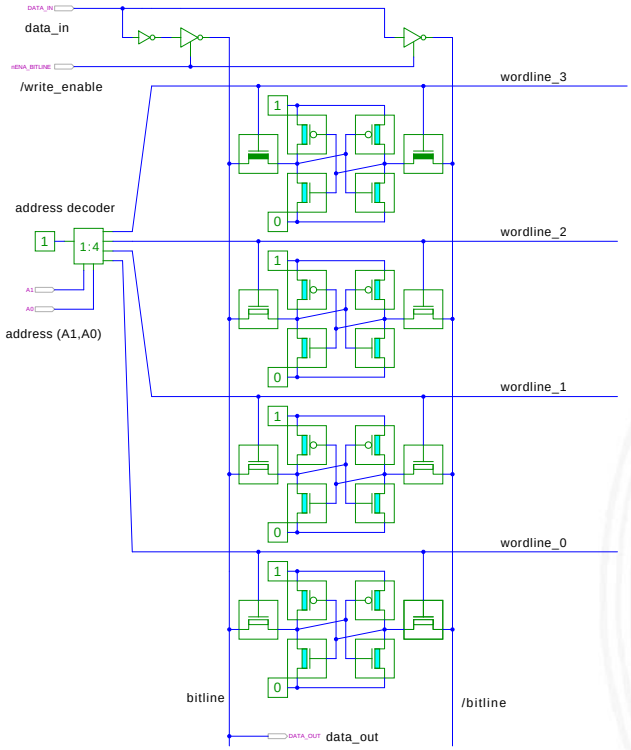
- viele unterschiedliche Taktschemata/Funktionsweisen
- Verbindung mit Logikgattern möglich
  ⇒ arithmetische Pipelines

. . . aus Zeitgründen nicht weiter vertieft

SRAM: Sechs-Transistor Speicherstelle („6T“)



Prinzip des SRAM



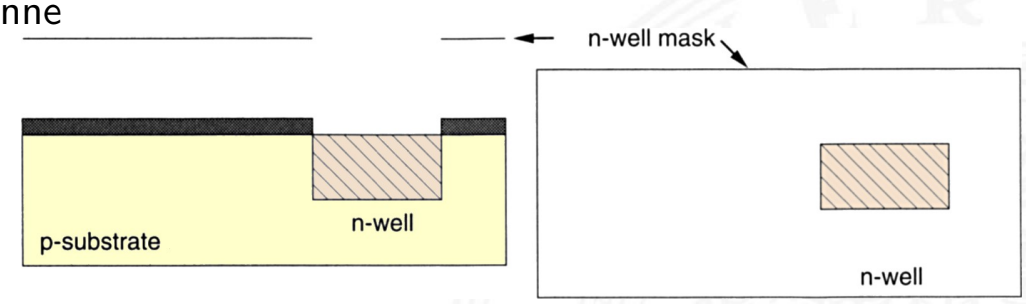
Hades Webdemos: 05-switched/40-cmos/sram4

CMOS Prozessschritte: Inverter

Ein n-Wannen Prozesses

Weste, Eshragian, *Principles of CMOS VLSI Design*

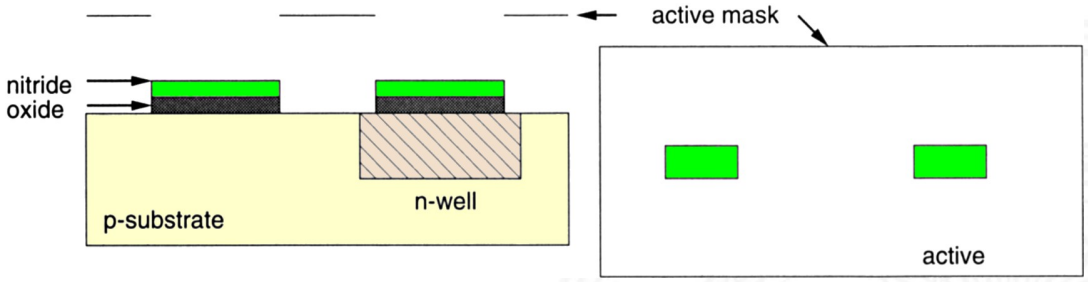
1. Ausgangsmaterial: p-dotiertes Substrat
2. n-Wanne



- Dotierung für p-Kanal Transistoren
- Herstellung: Ionenimplantation oder Diffusion

CMOS Prozessschritte: Inverter (cont.)

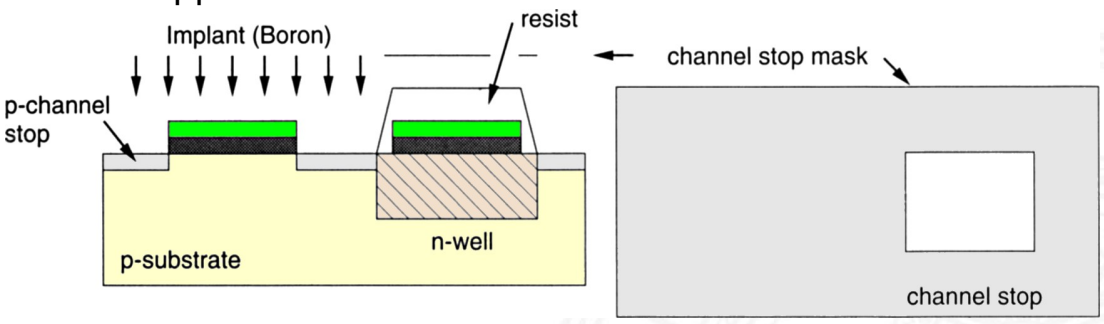
3. „aktive“ Fläche / Dünnoxid



- Spätere Gates und p^+-/n^+-Gebiete
- Herstellung: Epitaxie SiO_2 und Abdeckung mit Si_3N_4

CMOS Prozessschritte: Inverter (cont.)

4. p-Kanalstopp

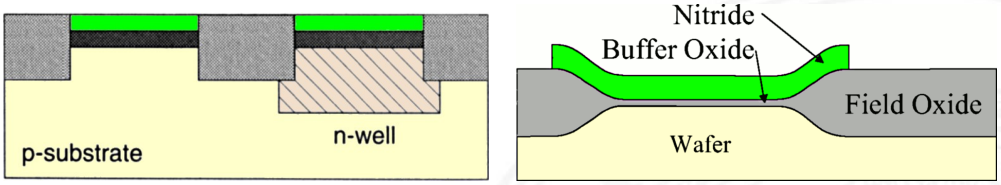


- Begrenzt n-Kanal Transistoren
- p-Wannen Maske, bzw. ¬ n-Wanne
- Maskiert durch Resist und Si_3N_4
- Substratbereiche in denen keine n-Transistoren sind
- Herstellung: p^+-Implant (Bor)

- n-Kanalstopp aktueller Prozesse: analog dazu

CMOS Prozessschritte: Inverter (cont.)

5. Resist entfernen
6. Feldoxid aufwachsen – SiO_2



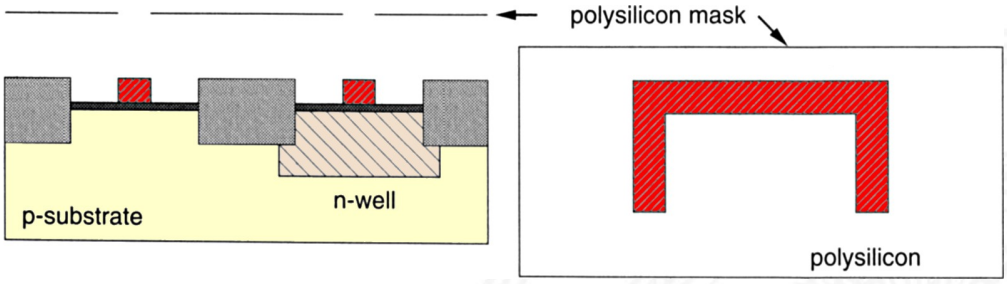
- LOCOS: **Loc**al **O**xidation of **S**ilicon
- Maskiert durch Si_3N_4
- Wächst auch lateral unter Si_3N_4/SiO_2 (aktive) Bereiche engl. *bird's beak*
- Der aktive Bereich wird kleiner als vorher maskiert
- Herstellung: Epitaxie und Oxidation
- Problem: nicht plane Oberfläche

CMOS Prozessschritte: Inverter (cont.)

7. Si_3N_4 entfernen, Gateoxid bleibt SiO_2
8. Transistor Schwellspannungen „justieren“
   - Meist wird das Polysilizium zusätzlich n^+ dotiert
     Grund: bessere Leitfähigkeit
   - Problem: $U_D(T_N) \approx \quad 0{,}5 \ldots \quad 0{,}7\,\mathrm{V}$
     $U_D(T_P) \approx -1{,}5 \ldots -2{,}0\,\mathrm{V}$
   - Maske: n-Wanne, bzw. p-Wanne
   - Herstellung: Epitaxie einer leicht negativ geladenen Schicht an der Substratoberfläche

CMOS Prozessschritte: Inverter (cont.)

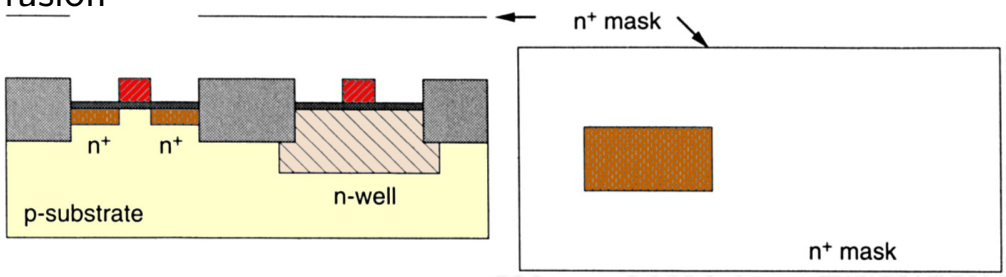
9. Polysilizium Gate



   - Herstellung: Epitaxie von Polysilizium, Ätzen nach Planarprozess

CMOS Prozessschritte: Inverter (cont.)

10. n^+-Diffusion

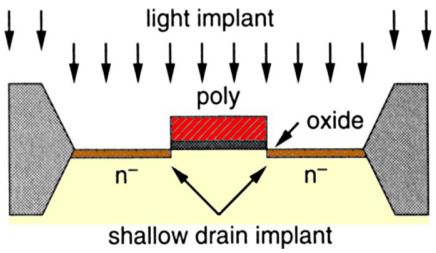


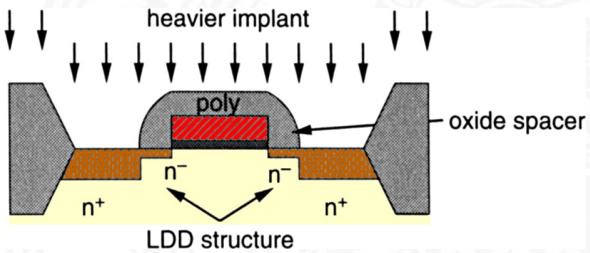
- Erzeugt Source und Drain der n-Kanal Transistoren
- Maskiert durch aktiven Bereich, n^+-Maske und Polysilizium
  $\Rightarrow$ Selbstjustierung
- Dotiert auch das Polysilizium Gate leicht (s.o.)
- Herstellung: Ionenimplantation, durchdringt Gateoxid

CMOS Prozessschritte: Inverter (cont.)

Zusätzliche Schritte bei der Source/Drain Herstellung

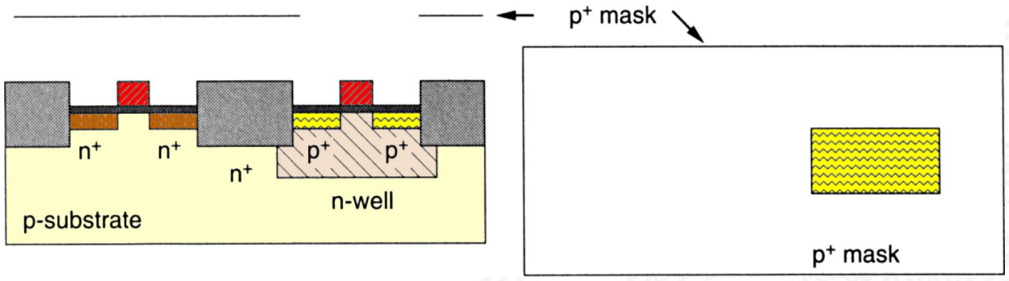
- Problem „Hot-Carrier“ Effekte (schnelle Ladungsträger):
  Stoßionisation, Gateoxid wird durchdrungen...
- Lösung: z.B. LDD (**L**ightly **D**oped **D**rain)
  a. „flaches“ n-LDD Implant
  b. zusätzliches SiO_2 über Gate aufbringen (*spacer*)
  c. „normales“ n^+-Implant
  d. Spacer SiO_2 entfernen





CMOS Prozessschritte: Inverter (cont.)

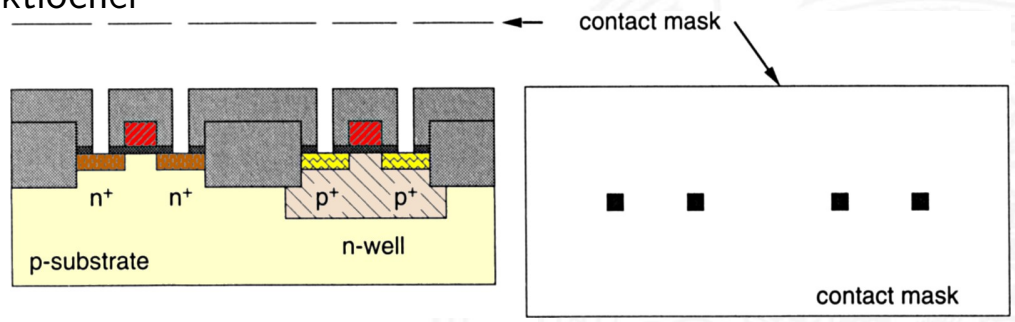
11. p^+-Diffusion



- Erzeugt Source und Drain der p-Kanal Transistoren
- Maskiert durch aktiven Bereich, p^+-Maske und Polysilizium
  $\Rightarrow$ Selbstjustierung
- teilweise implizite p^+-Maske $= \neg$ n^+-Maske
- wenig schnelle Ladungsträger (Löcher), meist keine LDD-Schritte
- Herstellung: Ionenimplantation, durchdringt Gateoxid

CMOS Prozessschritte: Inverter (cont.)

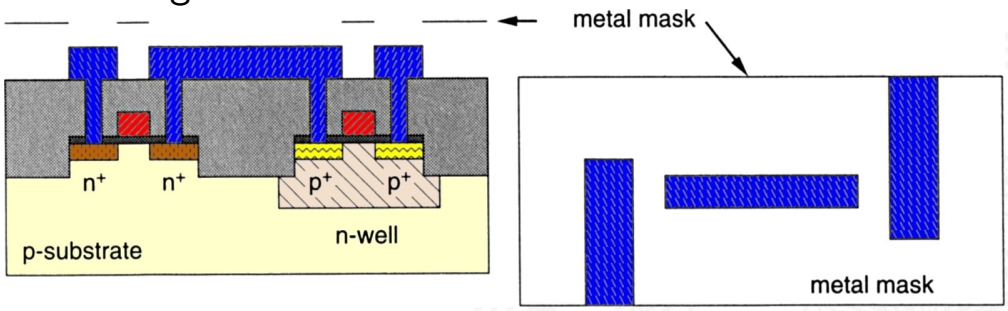
12. SiO_2 aufbringen, Feldoxid
    - Strukturen isolieren
    - Herstellung: Epitaxie
13. Kontaktlöcher



- Verbindet (spätere) Metallisierung mit Polysilizium oder Diffusion
- Anschlüsse der Transistoren: Gate, Source, Drain
- Herstellung: Ätzprozess

CMOS Prozessschritte: Inverter (cont.)

14. Metallverbindung



- Erzeugt Anschlüsse im Bereich der Kontaktlöcher
- Herstellung: Metall aufdampfen, Ätzen nach Planarprozess

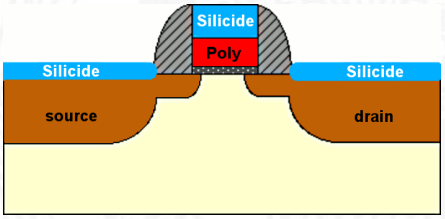
15. weitere Metalllagen
    - Weitere Metallisierungen, bis zu 7 $\times$ Metall
    - Schritte: 12. bis 14. wiederholen

CMOS Prozessschritte: Inverter (cont.)

16. Passivierung
    - Chipoberfläche abdecken, Plasmanitridschicht
17. Pad-Kontakte öffnen

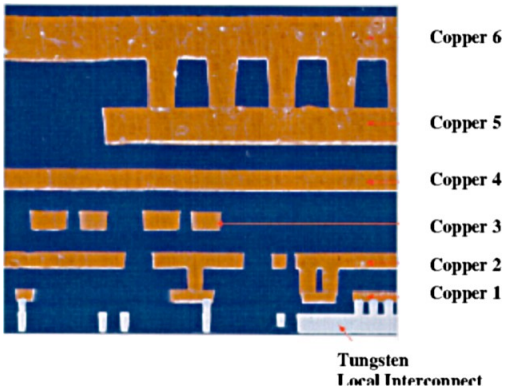
Zahlreiche Erweiterungen für Submikron CMOS-Prozesse

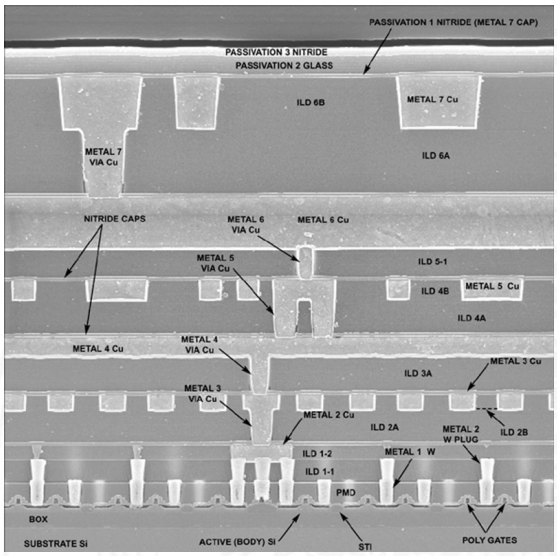
- „vergrabene“ Layer
    - verbessern elektrische Eigenschaften
    - Bipolar-Transistoren
    - Analog-Schaltungen
- Gate Spacer, seitlich SiO_2
- Silizidoberflächen: verringern Kontaktwiderstand zu Metallisierung



CMOS Prozessschritte: Inverter (cont.)

- Kupfer Metallisierung

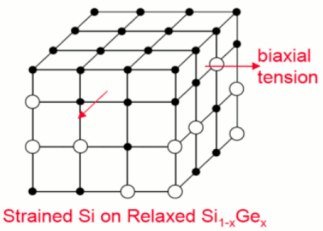


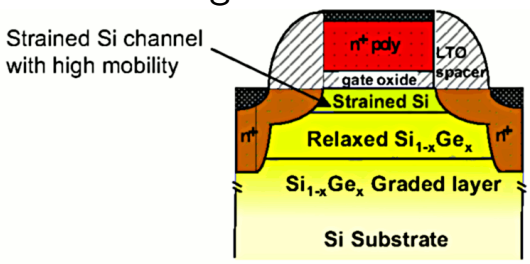


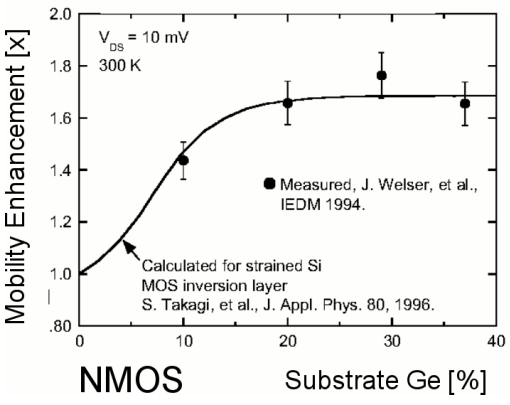
- high-k Dielektrika: Gate-Isolierung dicker, weniger Leckströme

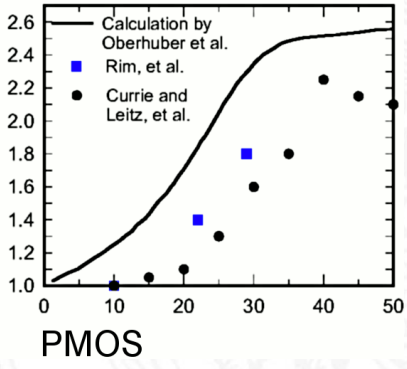
CMOS Prozessschritte: Inverter (cont.)

- „gestrecktes“ Silizium: höhere Beweglichkeit









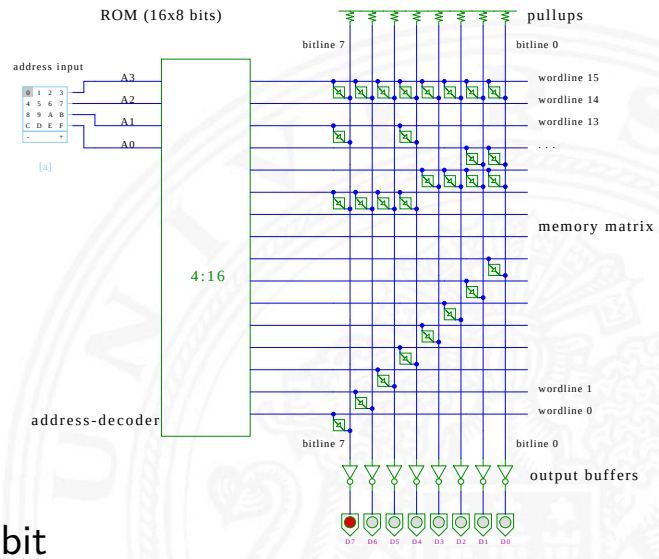
Programmierbare Logikbausteine

Kompromiss zwischen fest aufgebauter Hardware und Software-basierten Lösungen auf Computern

- Realisierung anwendungsspezifischer Funktionen und Systeme
  - gute bis sehr gute Performance
  - hoher Entwurfsaufwand
  - vom Anwender (evtl. mehrfach) programmierbar
- Klassifikation nach Struktur und Komplexität
  - PROM Programmable Read-Only Memory
  - PAL Programmable Array Logic
  - GAL Generic Array Logic
  - PLA Programmable Logic Array
  - CPLD Complex Programmable Logic Device
  - FPGA Field-Programmable Gate Array
  - ...

PROM: Programmable Read-Only Memory

- UND-ODER Struktur
- UND-Array
  - fest
  - voll auscodiert: 2^n Terme
- ODER-Terme
  - programmierbar
- auch: „LUT“ (look-up table)
- Hades Beispiel: $n = 4$, 16×8 bit



PAL: Programmable Array Logic

- disjunktive Form: UND-ODER Struktur
- UND-Ausgänge fest an die ODER-Eingänge angeschlossen
- Eingänge direkt und invertiert in die UND-Terme geführt
- Verknüpfungen der Eingänge zu den UND-Termen programmierbar

- heute durch GAL ersetzt (s.u.)

PLA: Programmable Logic Array

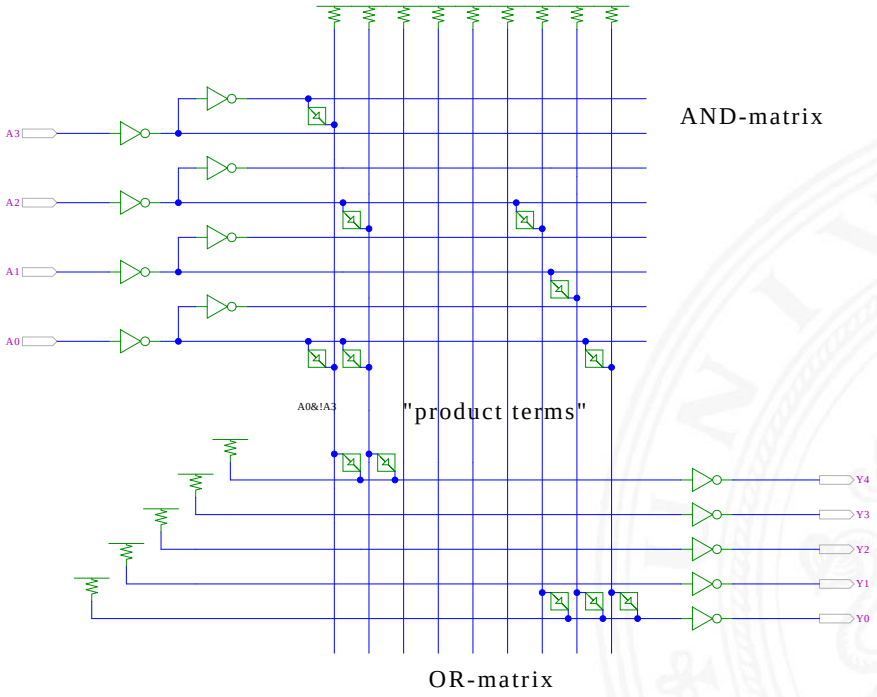
- disjunktive Form: logische UND-ODER Struktur
- Eingänge direkt und invertiert in die UND-Terme geführt
- Verknüpfungen Eingänge UND-Terme
- Verknüpfungen UND-Ausgänge zu ODER-Eingängen programmierbar

\+ in NMOS-Technologie sehr platzsparend realisierbar als NOR-NOR Matrix (de-Morgan Regel)

– statischer Stromverbrauch

– in CMOS-Technologie kaum noch verwendet

PLA: Programmable Logic Array (cont.)



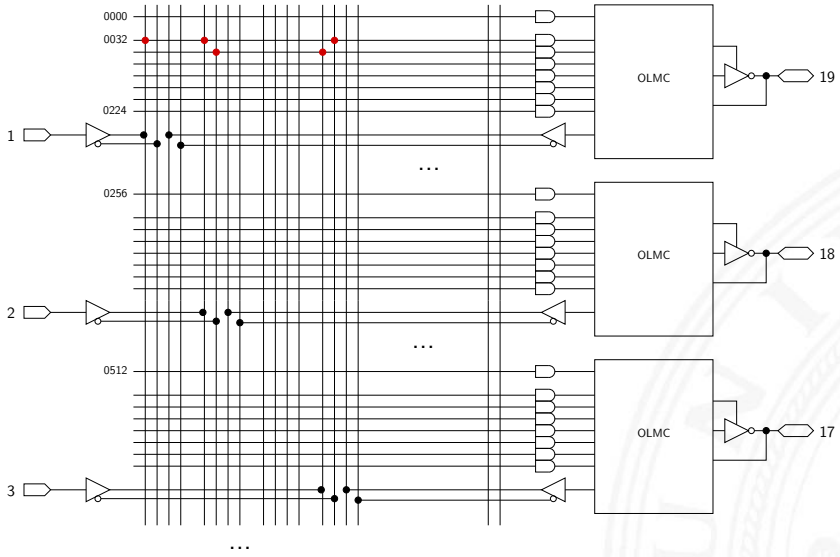
Hades Webdemos: 42-programmable/10-pla/pla.html

GAL: Generic Array Logic

- disjunktive UND-ODER Struktur
- externe Eingänge und Ausgangswerte direkt/invertiert
- „Fuses“ verbinden Eingangswerte mit den AND-Termen

- programmierbare Ausgabezellen (OLMC) mit je einem D-Flipflop
- Output-Enable über AND-OR Matrix steuerbar
- drei Optionen
  - synchron/kombinatorisch (Flipflop nutzen oder umgehen)
  - Polarität des Eingangs (D oder $\overline{D}$ speichern)
  - Polarität des Ausgangs (Q oder $\overline{Q}$ ausgeben)
- Beispiel: GAL16V8 mit 8 Ausgabezellen, je 7+1 OR-Terme pro Ausgabezelle, 32 Eingänge pro Term

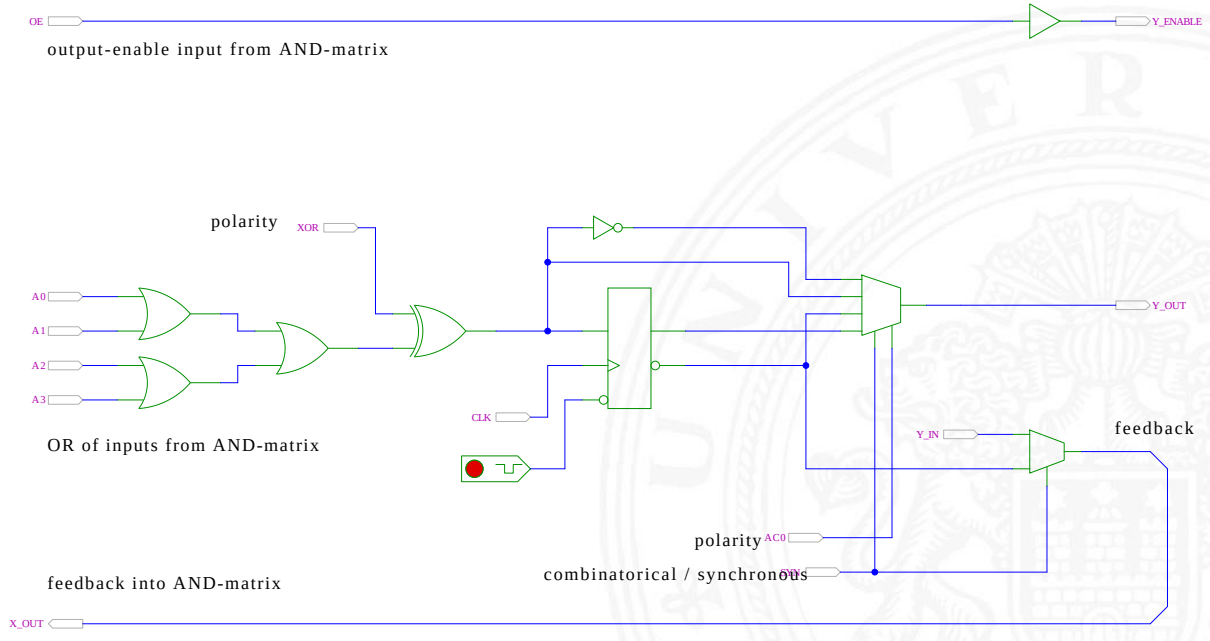
GAL: Blockschaltbild (Ausschnitt)



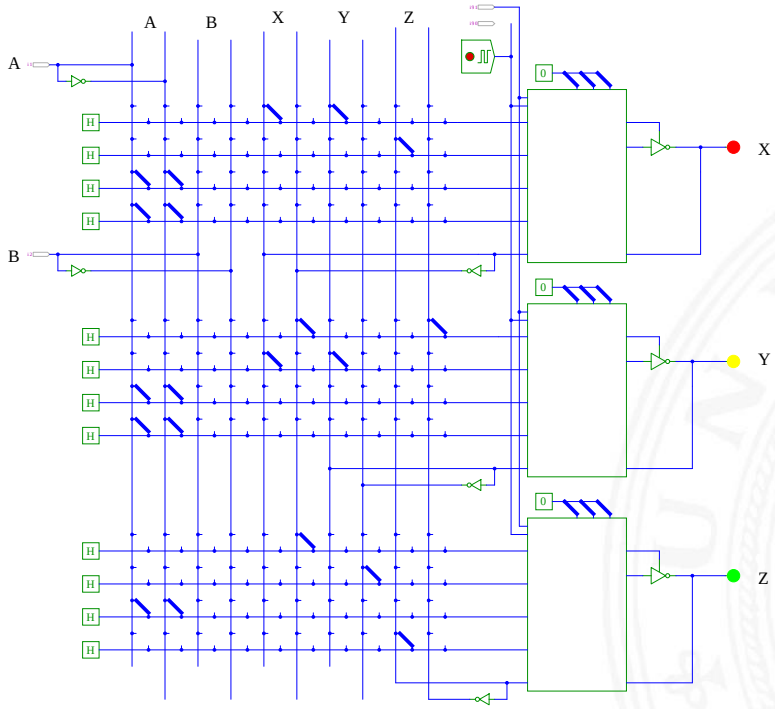
- programmierbare Sicherungen durchnummeriert
- kompakte Darstellung der UND-Terme: je eine Zeile
- Beispiel: zweiter Term (ab 0032) $y = 1 \vee 2 \vee \overline{3}$

GAL: Ausgabezelle mit Flipflop

OLMC: Output-Logic-Macrocell



GAL: Beispiel Ampel



FPGA: Field-Programmable Gate-Array

Sammelbegriff für „große“ anwenderprogrammierbare Schaltungen

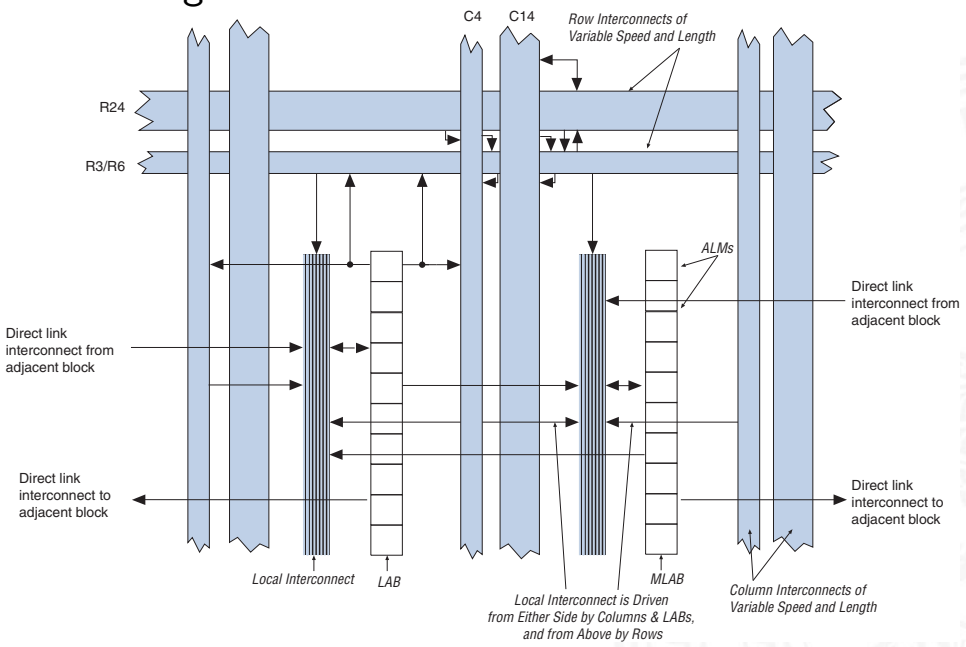
- Matrix von kleineren programmierbaren Zellen, beispielsweise
  - SRAM als Lookup für Funktionen
  - programmierbare Register
  - carry-lookahead Logik
- Multiplexer-Netzwerk als programmierbare Verbindung
- zusätzliche „Makrozellen“
  - Multiplizierer
  - eingebettete Prozessorkerne
- IO-Zellen
  - schnelle serielle Kommunikation
  - PLLs (programmierbare Taktgeneratoren)

FPGA: Field-Programmable Gate-Array (cont.)

- generierte Komponenten: ROM, RAM, FIFO...
- vorgefertigte IP-Blöcke („*Intellectual Property*“)
  - Netzwerkprotokolle
  - Speichercontroller
  - Bussysteme
  - ...
- Komplexität
  - $\approx$ 1 200 nutzbare I/O
  - $\approx$ 15 Mio. Gatteräquivalente (2 input NOR)
  - $\approx$ 1 GHz
- Xilinx, Altera, weitere Hersteller

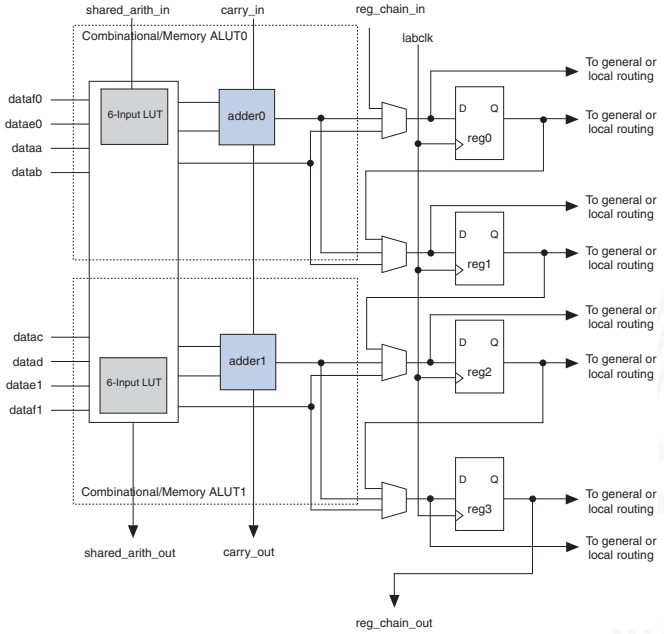
FPGA: Beispiel Altera

Verbindungsnetzwerk



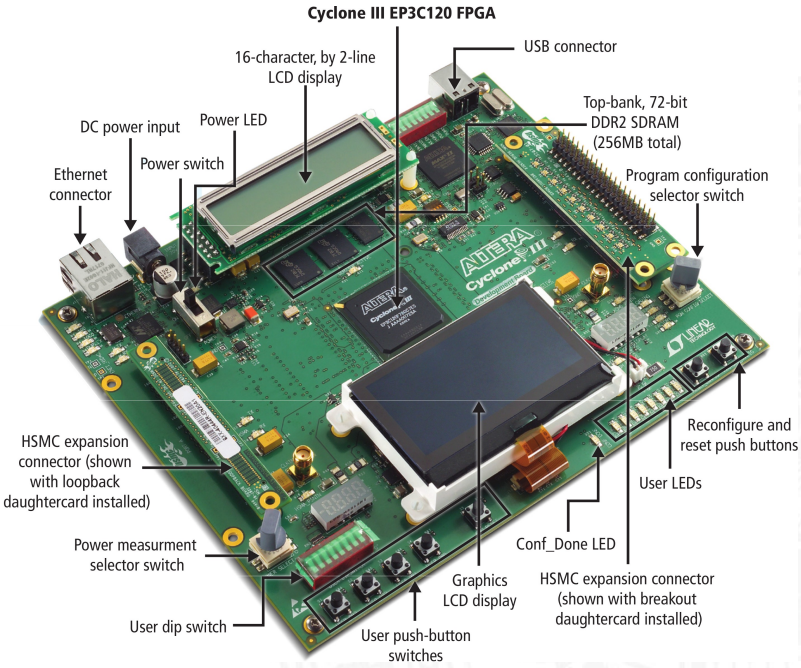
FPGA: Beispiel Altera (cont.)

programmierbarer Block



FPGA: Beispiel Altera (cont.)

Prototypenplatine



Entwurf Integrierter Schaltungen

besonders anspruchsvoller Bereich der Informatik

- Halbleiterfertigung benötigt vorab sämtliche Geometriedaten
- spätere Änderungen eines Chips nicht möglich
- Durchlauf aller Fertigungsschritte dauert Wochen bis Monate
- Entwürfe müssen komplett fehlerfrei sein

- spezielle Hardware-/System-Beschreibungssprachen
- Simulation des Gesamtsystems
- Analyse des Zeitverhaltens
- ggf. Emulation/Prototyping mit FPGAs

- Kombination von Hardware- oder Softwarerealisierung von Teilfunktionen, sog. **HW/SW-Codesign**

Entwurfsablauf

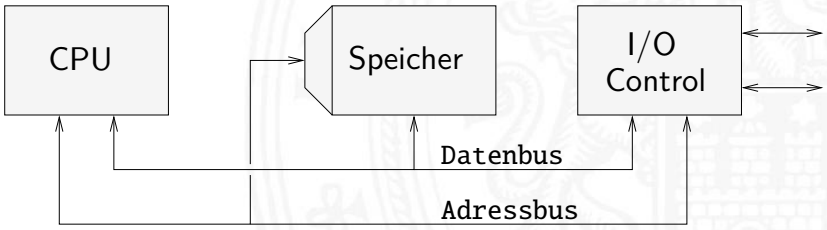
Wasserfallmodell

- Lastenheft
- Verhaltensmodell (Software)
- Aufteilung in HW- und SW-Komponenten
- funktionale Simulation/Emulation und Test
- Synthese oder manueller Entwurf der HW, Floorplan
- Generieren der „Netzliste“ (logische Struktur)
- Simulation mit Überprüfung der Gatter-/Leitungslaufzeiten
- Generieren und Optimierung des Layouts („Tapeout“)

Abstraktion im VLSI-Entwurf

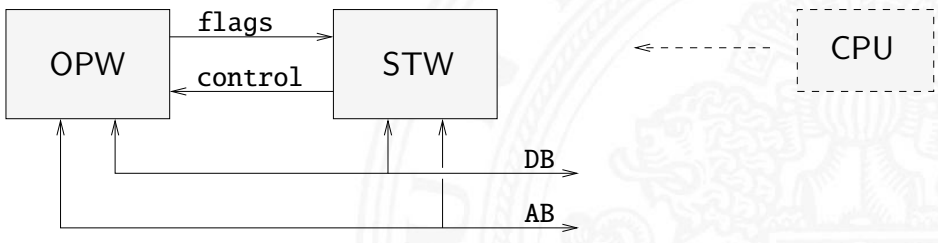
Abstraktionsebenen

- keine einheitliche Bezeichnung in der Literatur
- Architekturebene
  - Funktion/Verhalten Leistungsanforderungen
  - Struktur Netzwerk
    aus Prozessoren, Speicher, Busse, Controller...
  - Nachrichten Programme, Prokolle
  - Geometrie Systempartitionierung



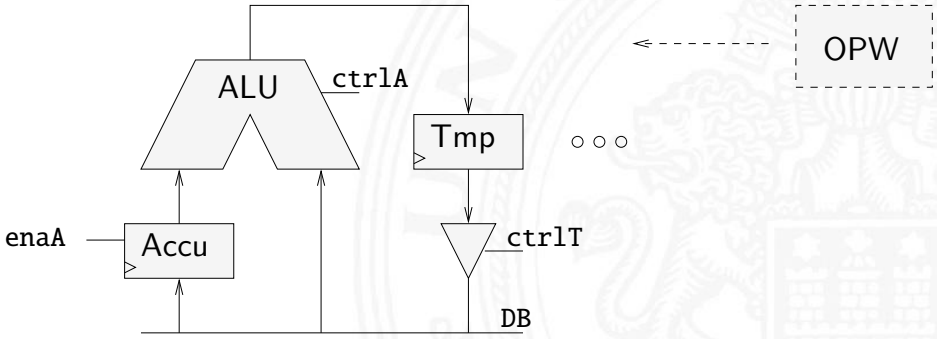
Abstraktion im VLSI-Entwurf (cont.)

- Hauptblockebene (Algorithmenebene, funktionale Ebene)
  - Funktion/Verhalten Algorithmen, formale Funktionsmodelle
  - Struktur Blockschaltbild
    aus Hardwaremodule, Busse...
  - Nachrichten Prokolle
  - Geometrie Cluster



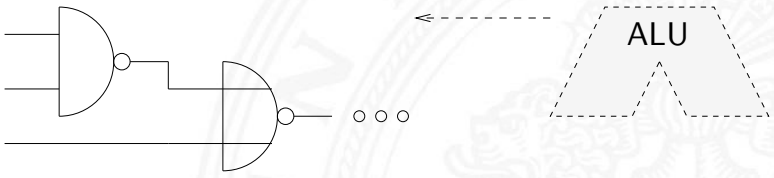
Abstraktion im VLSI-Entwurf (cont.)

- Register-Transfer Ebene
  - Funktion/Verhalten Daten- und Kontrollfluss, Automaten...
  - Struktur RT-Diagramm
    aus Register, Multiplexer, ALUs...
  - Nachrichten Zahlencodierungen, Binärworte...
  - Geometrie Floorplan



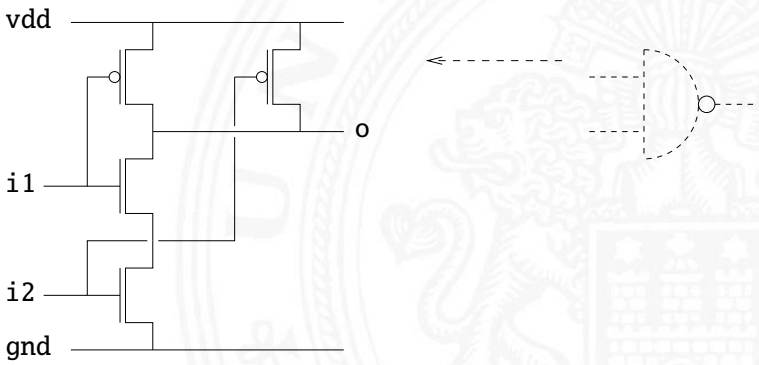
Abstraktion im VLSI-Entwurf (cont.)

- Logikebene (Schaltwerkebene)
  - Funktion/Verhalten Boole'sche Gleichungen
  - Struktur Gatternetzliste, Schematic
    aus Gatter, Flipflops, Latches...
  - Nachrichten Bit
  - Geometrie Moduln



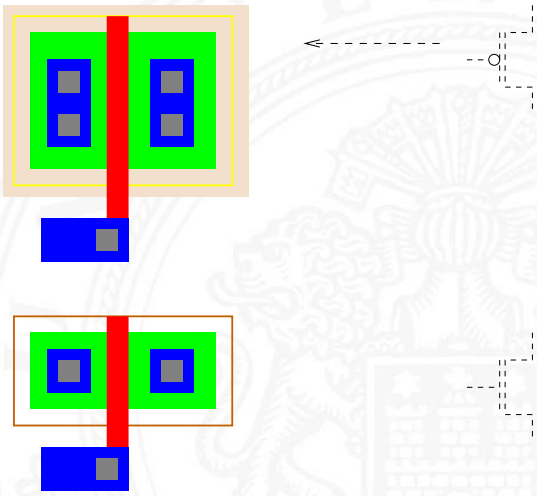
Abstraktion im VLSI-Entwurf (cont.)

- elektrische Ebene (Schaltkreisebene)
  - Funktion/Verhalten Differentialgleichungen
  - Struktur elektrisches Schaltbild
    aus Transistoren, Kondensatoren...
  - Nachrichten Ströme, Spannungen
  - Geometrie Polygone, Layout → physikalische Ebene

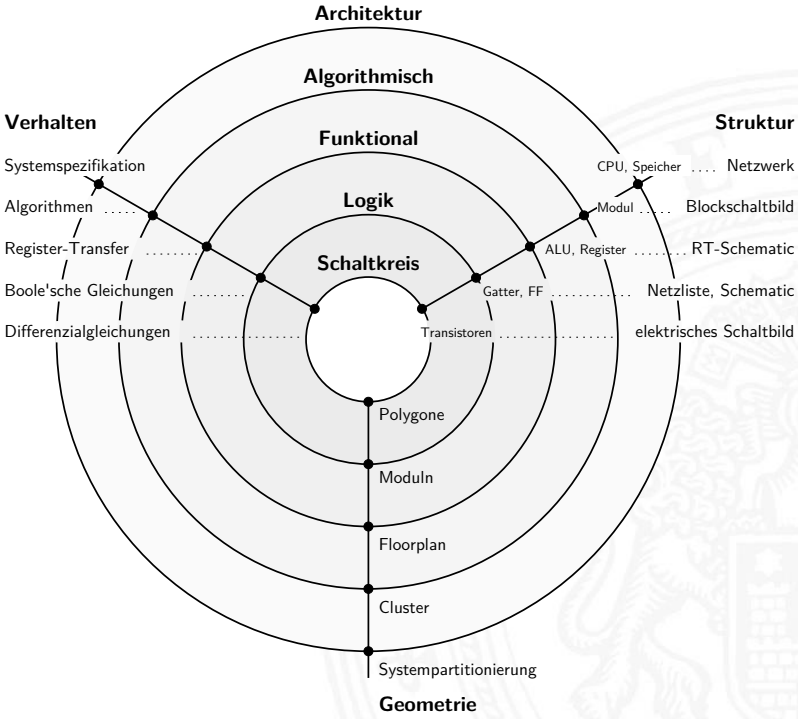


Abstraktion im VLSI-Entwurf (cont.)

- physikalische Ebene (geometrische Ebene)
  - Funktion/Verhalten partielle DGL
  - Struktur Dotierungsprofile



Y-Diagramm



D. Gajski, R. Kuhn 1983:
„New VLSI Tools“

Y-Diagramm (cont.)

drei unterschiedliche Aspekte/Dimensionen:

1. Verhalten
2. Struktur (logisch)
3. Geometrie (physikalisch)

- Start möglichst abstrakt, z.B. als Verhaltensbeschreibung
- Ende des Entwurfsprozesses ist vollständige IC Geometrie für die Halbleiterfertigung (Planarprozess)

- Entwurfsprogramme („EDA“, *Electronic Design Automation*) unterstützen den Entwerfer: setzen Verhalten in Struktur und Struktur in Geometrien um

Entwurfsstile

Was ist die „beste“ Realisierung einer gewünschten Funktionalität?

- mehrere konkurrierende Kriterien
  - Performance, Chipfläche, Stromverbrauch
  - Stückkosten vs. Entwurfsaufwand und Entwurfskosten
  - Zeitbedarf bis zur ersten Auslieferung und ggf. für Designänderungen
  - Schutz von Intellectual-Property
  - ...
- vier gängige Varianten
  - Full-custom Schaltungen
  - Semi-custom Bausteine: Standardzellen, Gate-Arrays
  - Anwenderprogrammierbare Bausteine: FPGA, PAL/GAL, ROM
  - Software auf von-Neumann Rechner: RAM, ROM

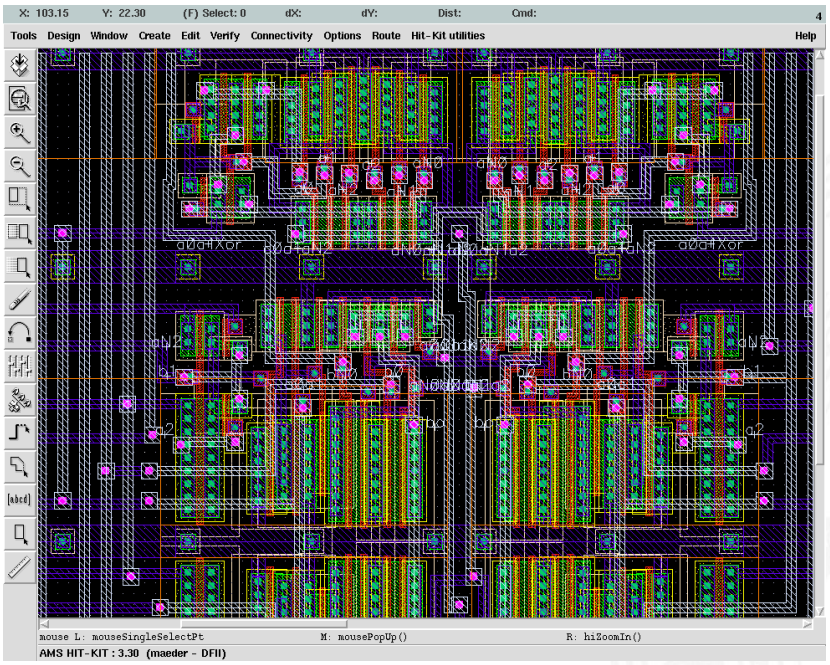
Full-custom / „Vollkunden-Entwurf“

- vollständiger Entwurf der gesamten Geometrie eines Chips
- jeder Transistor einzeln „maßgeschneidert“ und platziert
- vorgegeben sind lediglich die Entwurfsregeln (*design-rules*) des Herstellungsprozesses (Strukturbreite, Mindestabstände, usw.)
- oft Verwendung von Teilschaltungen/Makros des Herstellers

- minimale Fläche, beste Performance, kleinster Stromverbrauch
- geringste Stückkosten bei der Produktion
- aber höchste Entwurfs- und Maskenkosten
- erste Prototypen erst nach Durchlaufen aller Maskenschritte

- nur bei Massenprodukten wirtschaftlich $\gg 100\,000$ Stück
  z.B. Speicherbausteine (SRAM, DRAM), gängige Prozessoren

Full-custom / „Vollkunden-Entwurf“ (cont.)



Semi-custom: Standardzell-Entwurf

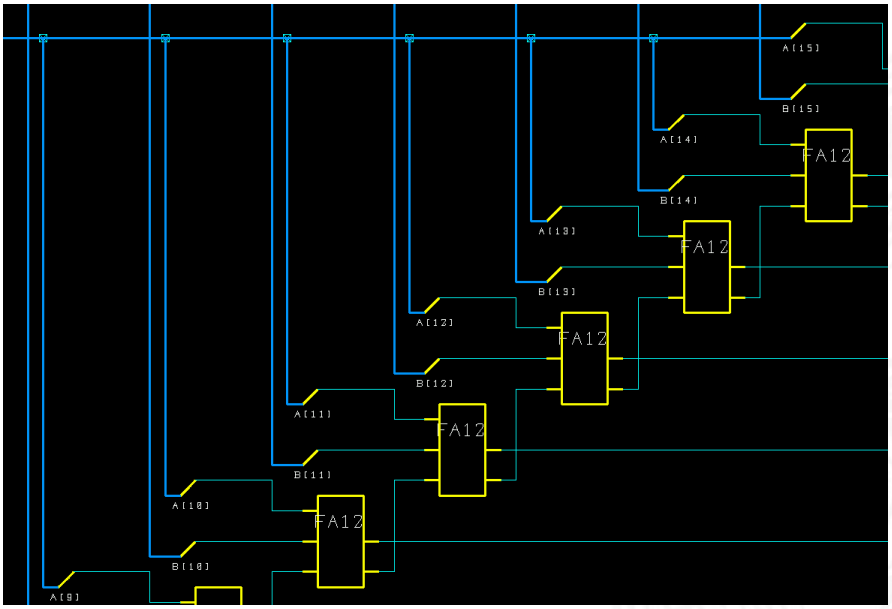
- Entwurf der Schaltung mit vorhandenen Grundkomponenten
  - Basisbibliothek mit Gattern und Flipflops
  - teilweise (konfigurierbare) ALUs, Multiplizierer
  - Generatoren für Speicher
- Entwurfsregeln sind der Bibliothek berücksichtigt
- Platzierung der Komponenten und Verdrahtung

- kleine Chipfläche, gute Performance, niedriger Stromverbrauch
- geringe Stückkosten
- hohe Maskenkosten (alle Masken erforderlich)
- erste Prototypen erst nach Durchlaufen aller Maskenschritte

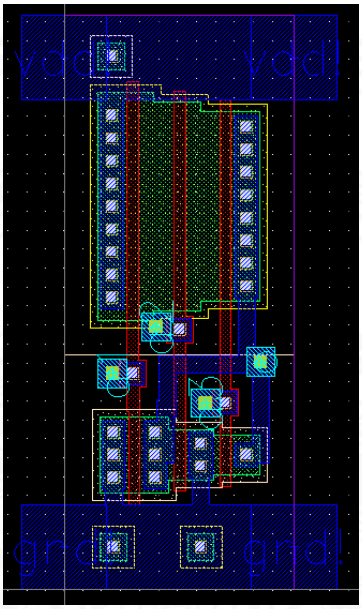
- nur bei größeren Stückzahlen wirtschaftlich $\gg 10\,000$ Stück

Semi-custom: Standardzell-Entwurf (cont.)

Schematic

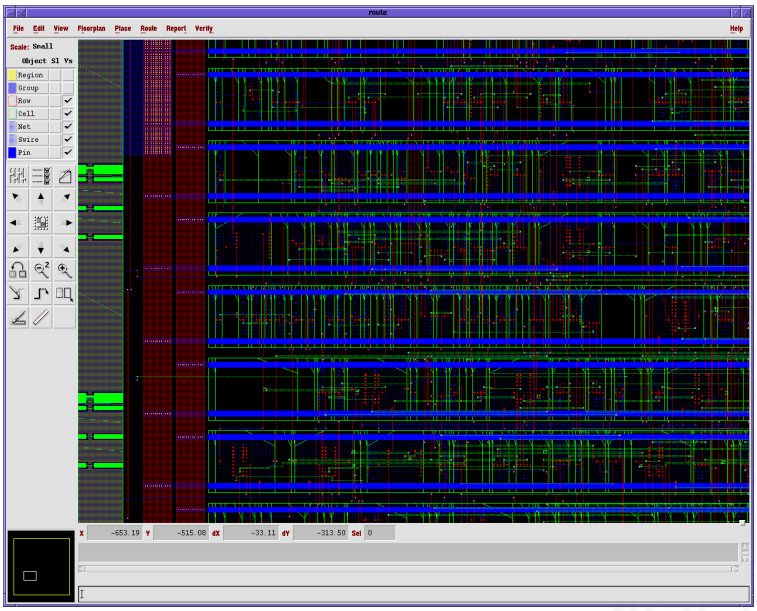


Zell-Layout



Semi-custom: Standardzell-Entwurf (cont.)

Standardzell Layout



Semi-custom: Gate-Arrays

- Schaltung mit Gattern/Transistoren an festen Positionen
- Entwurf durch Verdrahten der vorhandenen Transistoren
- überzählige Transistoren werden nicht angeschlossen

- mittlere Chipfläche, Performance und Stromverbrauch
- mittlere Stückkosten
- mittlere Maskenkosten (nur Verdrahtung kundenspezifisch)
- Prototypen schnell verfügbar (nur Verdrahtung)

- ab mittleren Stückzahlen wirtschaftlich $> 1\,000$ Stück
- werden von großen FPGAs verdrängt

FPGA: Field-Programmable Gate-Arrays

- Hunderte/Tausende von konfigurierbaren Funktionsblöcken
- Verschaltung dieser Blöcke vom Anwender programmierbar
- Entwurfsprogramme setzen Beschreibung des Anwenders auf die Hardware-Blöcke und deren Verschaltung um
- derzeit bis ca. 15 Mio. Gatter-Äquivalente möglich
- Taktfrequenzen bis max. GHz, typisch 100 MHz-Bereich
- zwei dominierende Hersteller: Xilinx, Altera

- nicht benutzte Blöcke liegen brach
- Schaltung kann in Minuten neu programmiert/verbessert werden

- optimal für geringe Stückzahlen, ca. $10 \ldots 1\,000$ Stück

FPGA selbstgemacht: Projekt 64-189

Ideen für einen Mikrochip? Zum Beispiel für Bildverarbeitung, 3D-Algorithmen, Parallelverarbeitung, usw.

- Hereinschnuppern: Projekt 64-189 *Entwurf eines Mikrorechners*
- eigenen Prozessor mit Befehlssatz etc. entwerfen und auf FPGA Prototypenplatine realisieren

- Demo-Boards von Altera und Xilinx und Entwurfssoftware sind bei uns am Fachbereich verfügbar

$\Rightarrow$ einfach bei TAMS oder TIS vorbeischauen

Literatur: Quellen für die Abbildungen

- Andreas Mäder,
Vorlesung: Rechnerarchitektur und Mikrosystemtechnik,
Universität Hamburg, FB Informatik, 2010
`tams.informatik.uni-hamburg.de/lectures/2010ws/vorlesung/ram`
- Norbert Reifschneider,
CAE-gestützte IC-Entwurfsmethoden,
Prentice Hall, 1998
- Neil H. E. Weste, Kamran Eshragian,
Principles of CMOS VLSI Design — A Systems Perspective,
Addison-Wesley Publishing, 1994

Literatur: Vertiefung

- Reiner Hartenstein,
  Standort Deutschland: Wozu noch Mikro-Chips,
  IT-Press Verlag, 1994 (vergriffen)
- Gabriela Nicolescu, Pieter J. Mosterman,
  Model-Based Design for Embedded Systems, CRC Press, 2010
- Carver Mead, Lynn Conway,
  Introduction to VLSI Systems, Addison-Wesley, 1980
- Giovanni de Micheli,
  Synthesis and Optimization of Digital Circuits,
  McGraw-Hill, 1994
- Ulrich Tietze, Christoph Schenk,
  Halbleiter-Schaltungstechnik, Springer-Verlag, 2009

Gliederung

Gliederung (cont.)

Was ist Rechnerarchitektur?

Definitionen

1. *The term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behaviour, as distinct from the organization and data flow and control, the logical and the physical implementation.* [Amdahl, Blaauw, Brooks]

2. *The study of computer architecture is the study of the organization and interconnection of components of computer systems. Computer architects construct computers from basic building blocks such as memories, arithmetic units and buses.*

Was ist Rechnerarchitektur? (cont.)

From these building blocks the computer architect can construct anyone of a number of different types of computers, ranging from the smallest hand-held pocket-calculator to the largest ultra-fast super computer. The functional behaviour of the components of one computer are similar to that of any other computer, whether it be ultra-small or ultra-fast.

By this we mean that a memory performs the storage function, an adder does addition, and an input/output interface passes data from a processor to the outside world, regardless of the nature of the computer in which they are embedded. The major differences between computers lie in the way of the modules are connected together, and the way the computer system is controlled by the programs. In short, computer architecture is the discipline devoted to the design of highly specific and individual computers from a collection of common building blocks. [Stone]

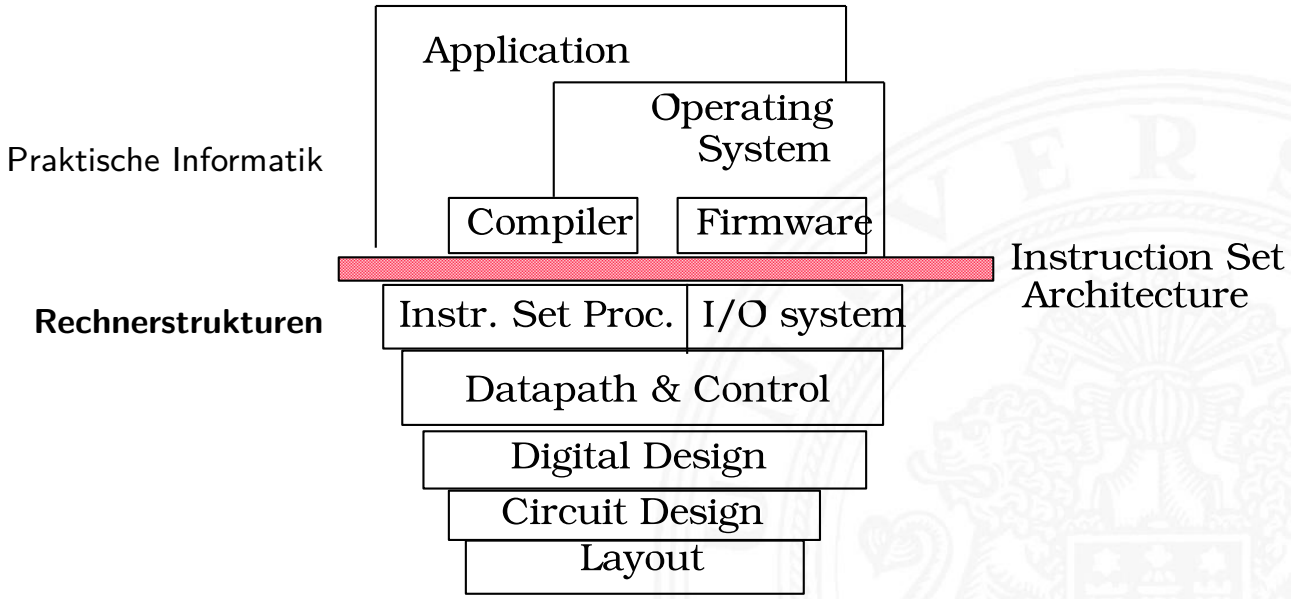
Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
17.1 Rechnerarchitektur - Motivation
64-040 Rechnerstrukturen

Was ist Rechnerarchitektur? (cont.)

Zwei Aspekte der Rechnerarchitektur

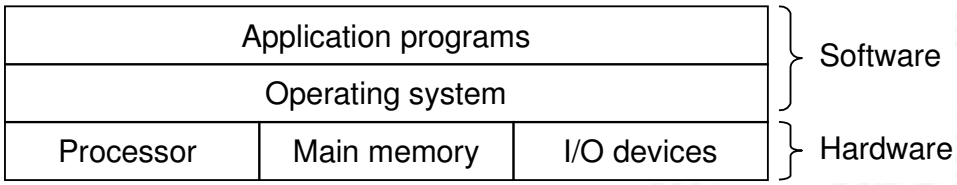
1. Operationsprinzip: das funktionelle Verhalten der Architektur
   - = Programmierschnittstelle
   - = ISA – **I**nstruction **S**et **A**rchitecture
     Befehlssatzarchitektur
   - = Maschinenorganisation
2. Hardwarestruktur: beschrieben durch Art und Anzahl der Hardware-Betriebsmittel sowie die sie verbindenden Kommunikationseinrichtungen
   - = Implementierung: welche Einheiten, wie verbunden...
   - = beispielsweise „von-Neumann“ Architektur

Schnittstelle zur praktischen Informatik

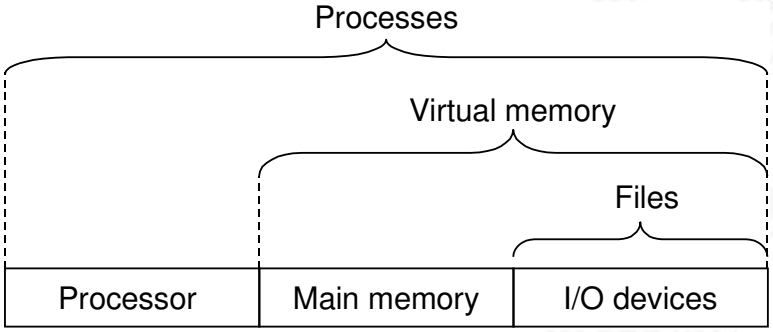


Beschreibungsebenen

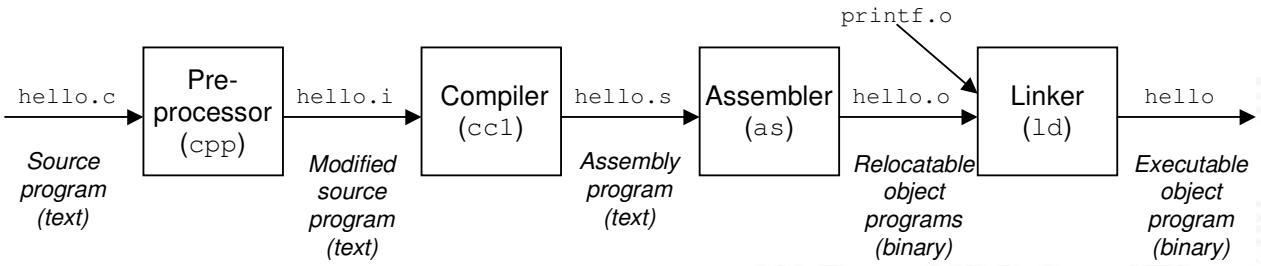
- Schichten-Ansicht: Software – Hardware



- Abstraktionen durch Betriebssystem



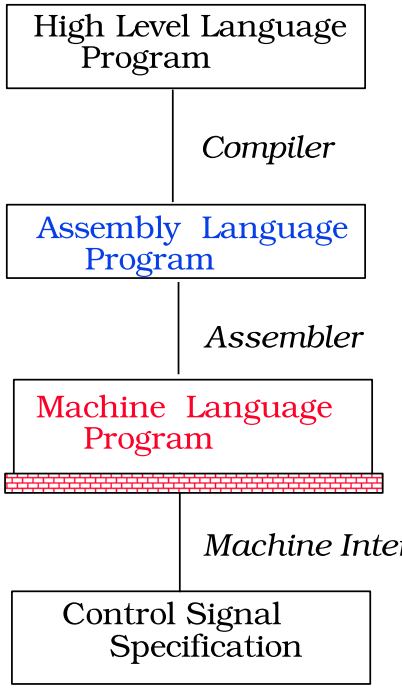
Das Kompilierungssystem



⇒ verschiedene Repräsentationen des Programms

- Hochsprache
- Assembler
- Maschinensprache

Das Kompilierungssystem (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
```

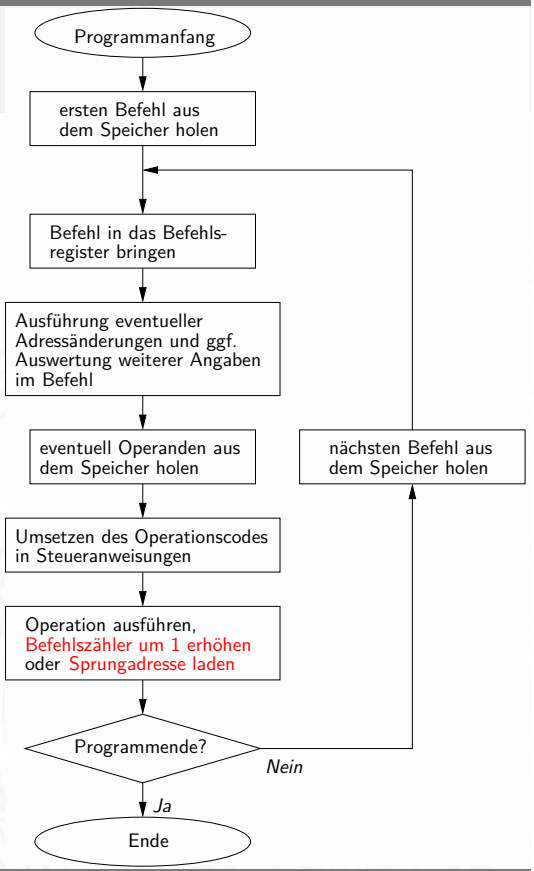
ALUOP[0:3] <= InstReg[9:11] & MASK

Wie rechnet ein Rechner?

- Architektur im Sinne von „Hardwarestruktur“ beispielsweise als von-Neumann Architektur
  - „Choreografie“ der Funktionseinheiten?
  - wie kommuniziert man mit Rechnern?
  - was passiert beim Einschalten des Rechners?
- Erweiterungen des von-Neumann Konzepts
  - *parallele*, statt sequentieller Befehlsabarbeitung
    Stichwort: superskalare Prozessoren
  - dynamisch veränderte Abarbeitungsreihenfolge
    Stichwort: *„out-of-order execution“*
  - getrennte Daten- und Instruktionsspeicher
    Stichwort: *Harvard-Architektur*
  - *Speicherhierarchie*, Caches etc.

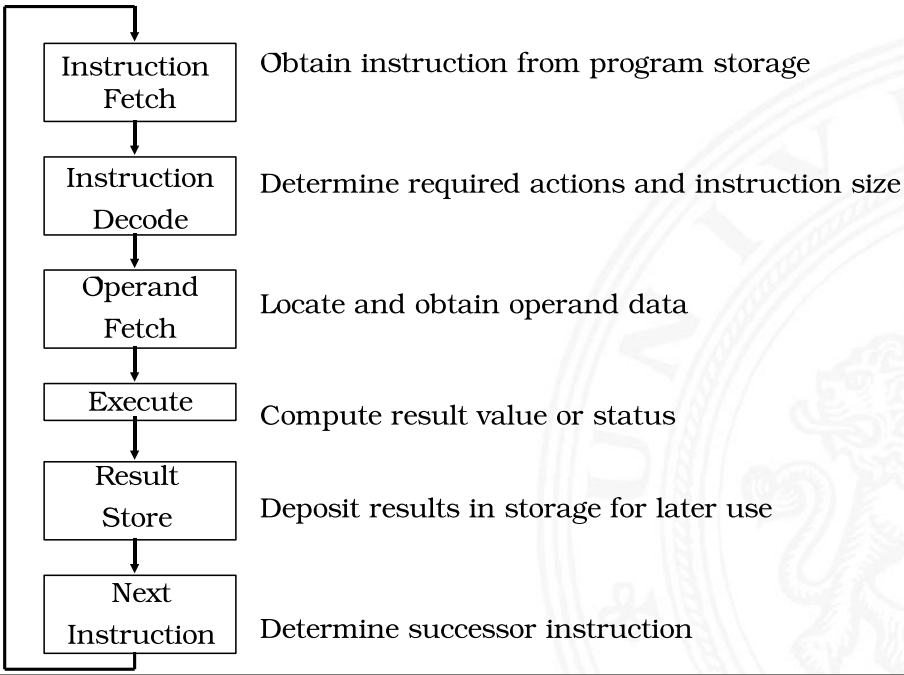
Programmverarbeitung

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

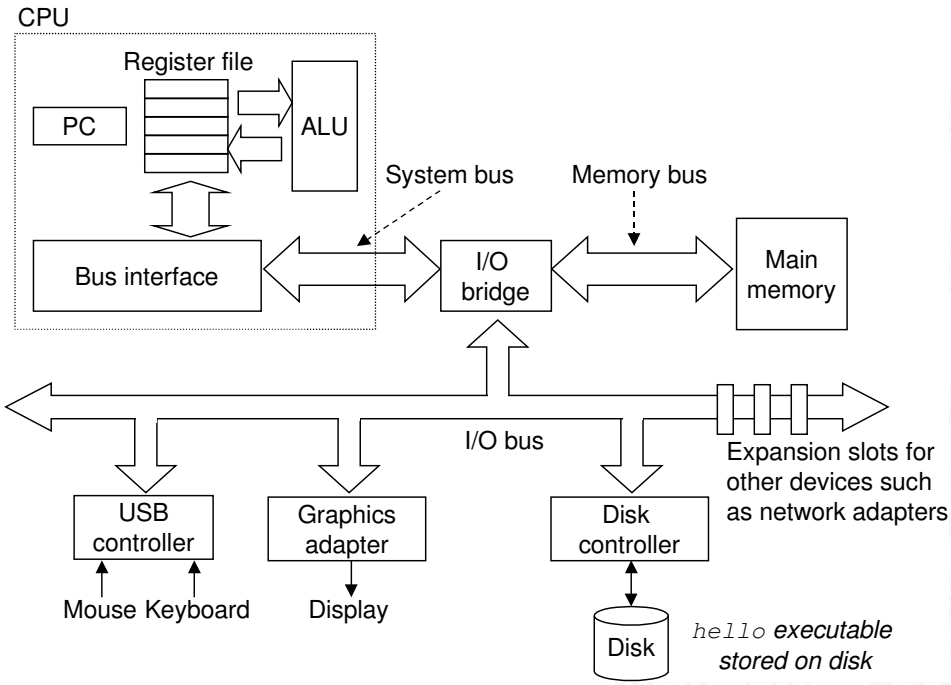


Programmverarbeitung (cont.)

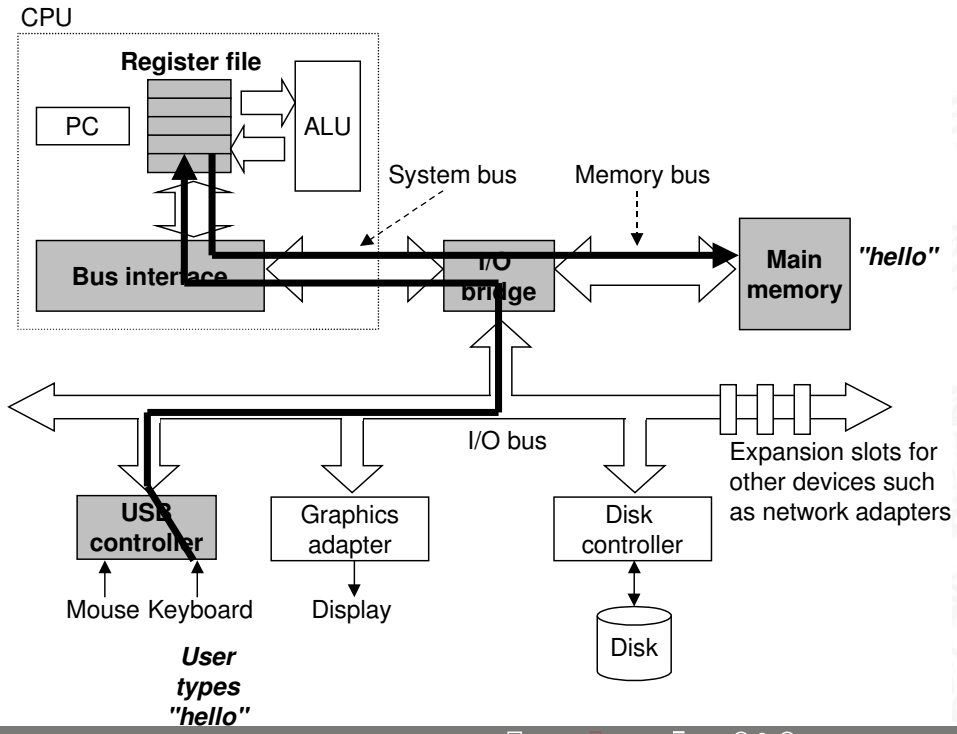
- Ausführungszyklus



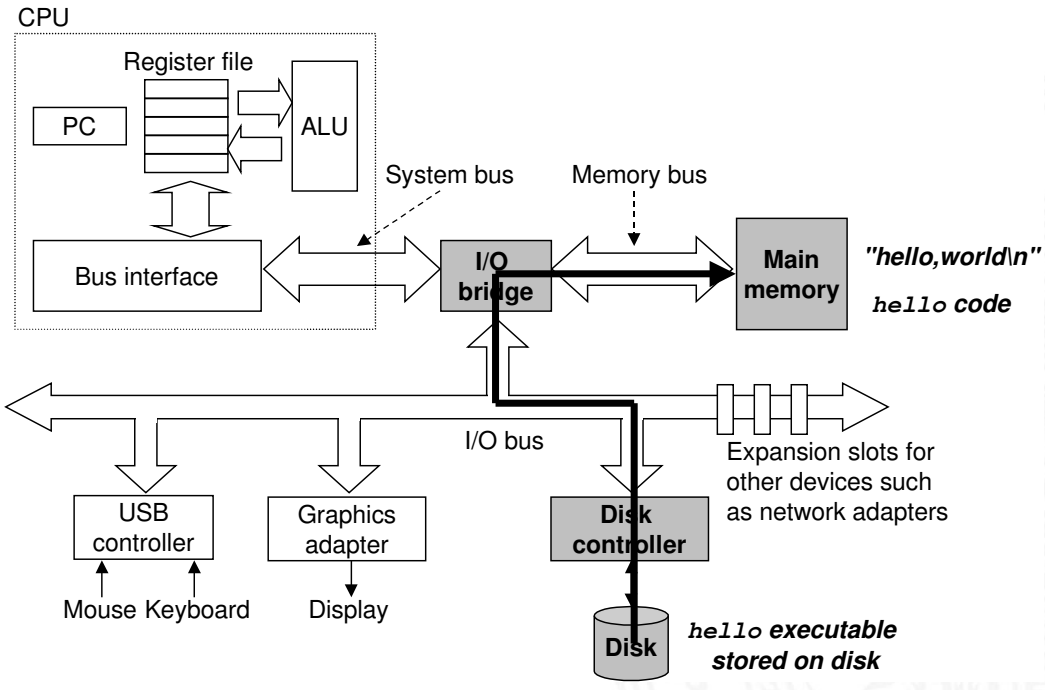
Hardwareorganisation eines typischen Systems



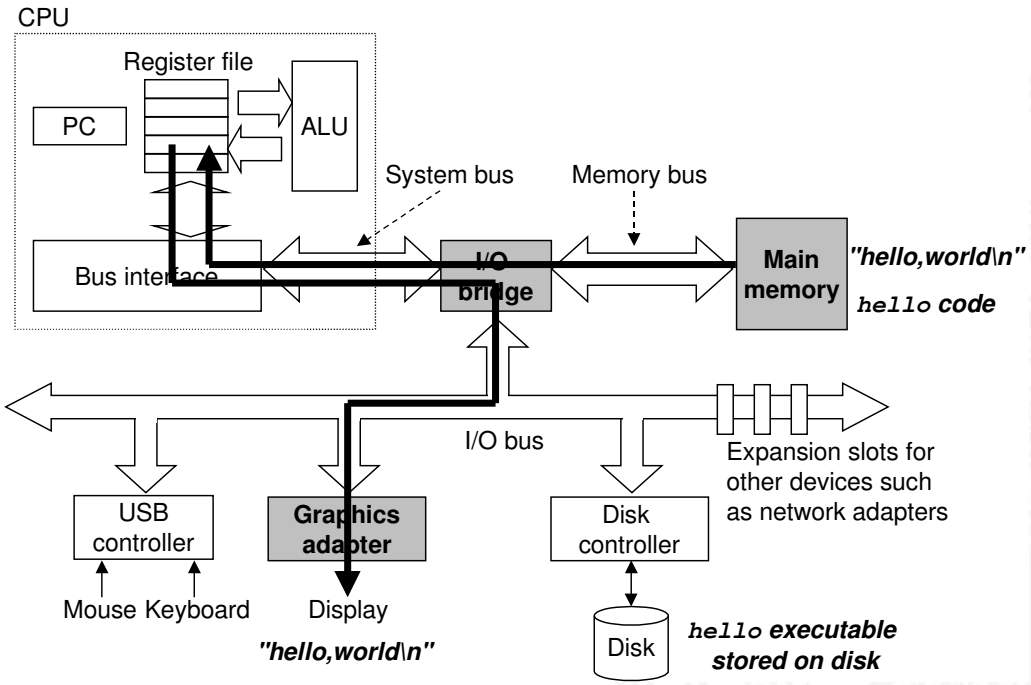
Programmausführung: 1. Benutzereingabe



Programmausführung: 2. Programm laden



Programmausführung: 3. Programmlauf



Boot-Prozess

Was passiert beim Einschalten des Rechners?

- Chipsatz erzeugt Reset-Signale für alle ICs
- Reset für die zentralen Prozessor-Register (PC, ...)
- PC wird auf Startwert initialisiert (z.B. 0xFFFF FFEF)
- Befehlszyklus wird gestartet
- Prozessor greift auf die Startadresse zu
  dort liegt ein ROM mit dem Boot-Programm
- Initialisierung und Selbsttest des Prozessors
- Löschen und Initialisieren der Caches
- Konfiguration des Chipsatzes
- Erkennung und Initialisierung von I/O-Komponenten
- Laden des Betriebssystems

Gliederung

Gliederung (cont.)

Befehlssatzarchitektur – ISA

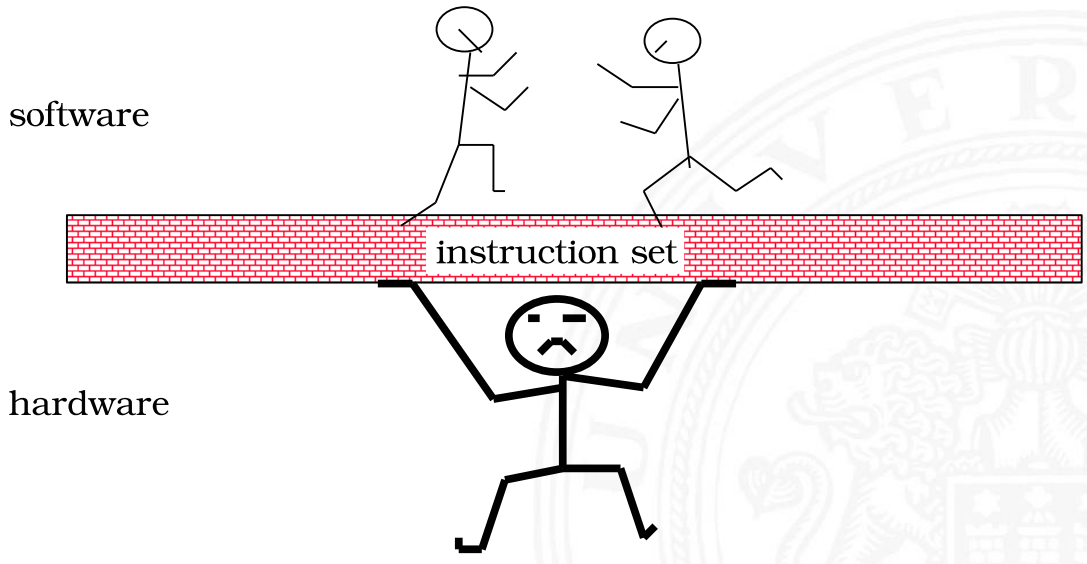
ISA – **I**nstruction **S**et **A**rchitecture

$\Rightarrow$ alle für den Programmierer sichtbaren Attribute eines Rechners

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

Befehlssatzarchitektur – ISA (cont.)

- Befehlssatz: die zentrale Schnittstelle



Merkmale der Instruction Set Architecture

- Speichermodell — Wortbreite, Adressierung, ...
- Rechnerklasse — Stack-/Akku-/Registermaschine
- Registersatz — Anzahl und Art der Rechenregister
- Befehlssatz — Definition aller Befehle
- Art, Zahl der Operanden — Anzahl/Wortbreite/Reg./Speicher
- Ausrichtung der Daten — Alignment/Endianness
- Ein- und Ausgabe, Unterbrechungsstruktur (Interrupts)
- Systemsoftware — Loader, Assembler, Compiler, Debugger

Beispiele für charakteristische ISA

in dieser Vorlesung bzw. im Praktikum angesprochen

- MIPS — klassischer 32-bit RISC
- D*CORE — „Demo Rechner“, 16-bit
- x86 — CISC, Verwendung in PCs

- Assemblerprogrammierung, Kontrollstrukturen und Datenstrukturen werden am Beispiel der x86-Architektur vorgestellt

- viele weitere Architekturen (z.B. Mikrokontroller) werden aus Zeitgründen nicht weiter behandelt

Artenvielfalt vom „Embedded Architekturen“









Prozessor	4..32 bit	8 bit	-	16..32 bit	32 bit	32 bit	32 bit	8..64 bit	..32 bit
Speicher	1K..1M	< 8K	< 1K	1..64M	1..64M	< 512M	8..64M	1K..10M	< 64M
ASICs	1 uC	1 uC	1 ASIC	1 uP ASIP	DSPs	1 uP, 3 DSP	1 uP, DSP	~ 100 uC, uP, DSP	uP, ASIP
Netzwerk	cardIO	-	RS232	diverse	GSM	MIDI	V.90	CAN,...	I2C,...
Echtzeit	nein	nein	soft	soft	hard	soft	hard	hard	hard
Safety	keine	mittel	keine	gering	gering	gering	gering	hoch	hoch

- riesiges Spektrum: 4..64 bit Prozessoren, DSPs, digitale/analoge ASICs, ...
- Sensoren/Aktoren: Tasten, Displays, Druck, Temperatur, Antennen, CCD, ...
- Echtzeit-, Sicherheits-, Zuverlässigkeitsanforderungen

Speicherorganisation

- Wortbreite, Größe / Speicherkapazität
- „Big Endian“ / „Little Endian“
- „Alignment“
- „Memory-Map“
- Beispiel: PC mit Windows

- spätere Themen
  - Cache-Organisation für schnelleren Zugriff
  - Virtueller Speicher für Multitasking
  - MESI-Protokoll für Multiprozessorsysteme
  - Synchronisation in Multiprozessorsystemen

Wortbreite

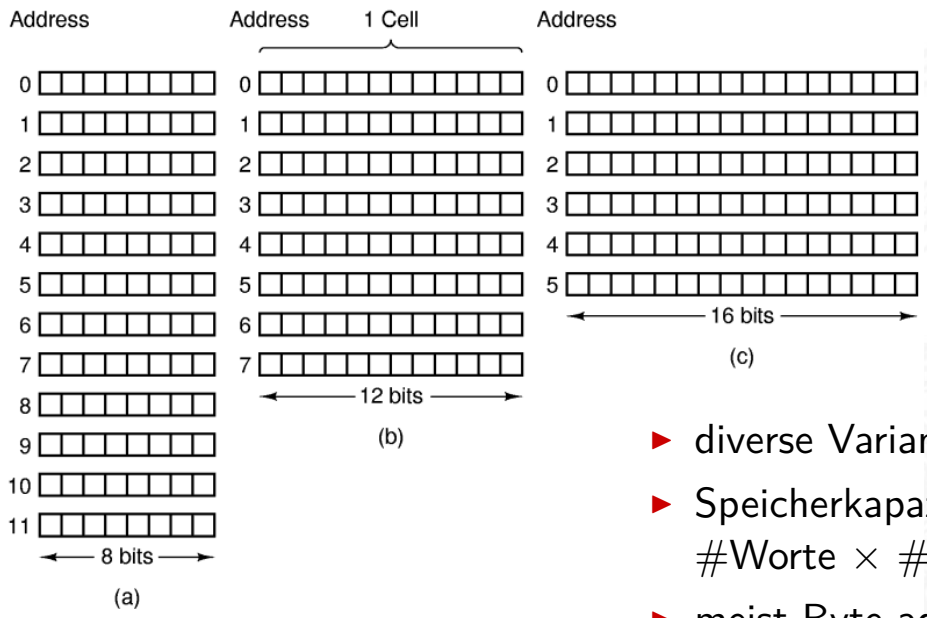
- Speicherwortbreiten historisch wichtiger Computer

Computer	Bits/cell
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

- heute dominieren 8/16/32/64-bit Systeme
- erlaubt 8-bit ASCII, 16-bit Unicode, 32-/64-bit Floating-Point
- Beispiel x86: „byte“, „word“, „double word“, „quad word“

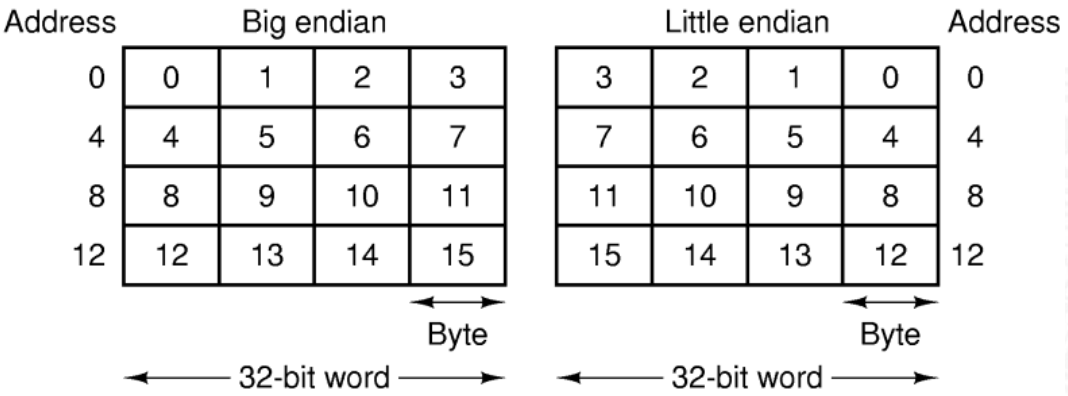
Hauptspeicherorganisation

Drei Organisationsformen eines 96-bit Speichers



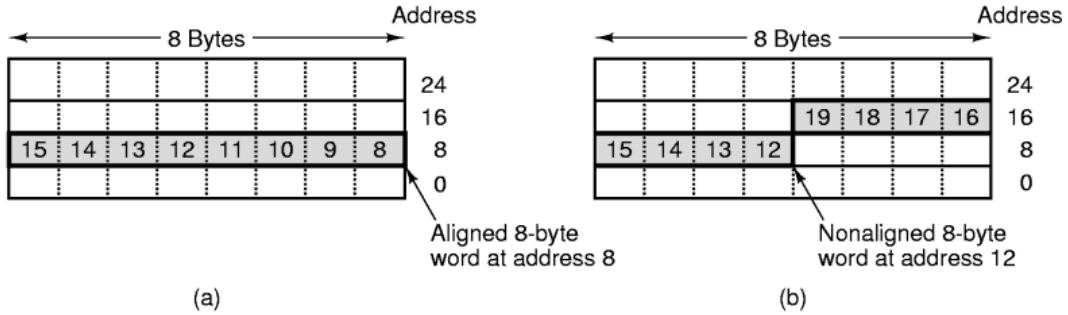
- diverse Varianten möglich
- Speicherkapazität:
  #Worte × #Bits/Wort
- meist Byte-adressiert

Big- vs. Little Endian



- Anordnung einzelner Bytes in einem Wort (hier 32 bit)
  - Big Endian: MSB kommt zuerst, gut für Strings
  - Little Endian: LSB kommt zuerst, gut für Zahlen
- beide Varianten haben Vor- und Nachteile
- ggf. Umrechnung zwischen beiden Systemen notwendig

„Misaligned“ Zugriff



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

⇒ was passiert bei „krummen“ (misaligned) Adressen?
- automatische Umsetzung auf mehrere Zugriffe (x86)
- Programmabbruch (MIPS)

Memory Map

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

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

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

Memory Map: typ. 16-bit System

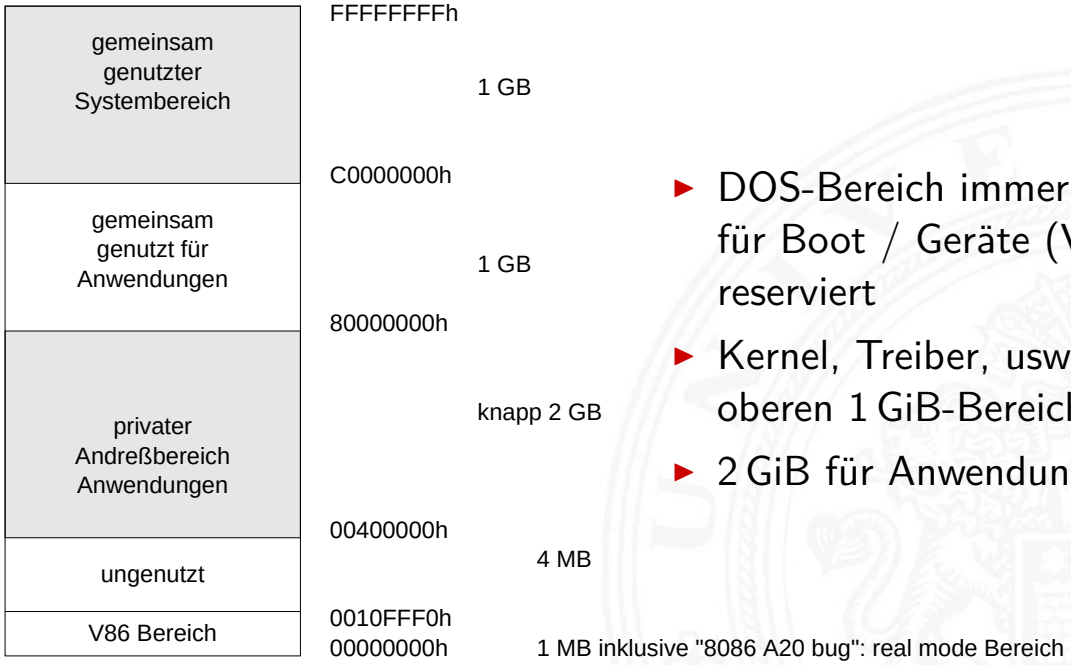
- 16-bit erlaubt 64K Adressen: 0x0000...0xFFFF

- ROM-Bereich für Boot / Betriebssystemkern
- RAM-Bereich für Hauptspeicher
- RAM-Bereich für Interrupt-Tabelle

- I/O-Bereiche für serielle / parallel Schnittstellen
- I/O-Bereiche für weitere Schnittstellen

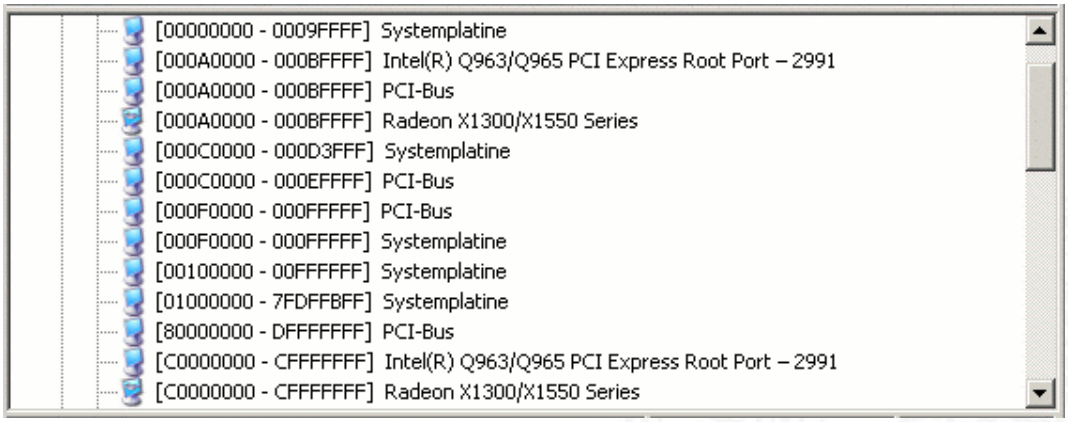
Demo und Beispiele: im Praktikum (64-042)

Memory Map: Windows 9x



- DOS-Bereich immer noch für Boot / Geräte (VGA) reserviert
- Kernel, Treiber, usw. im oberen 1 GiB-Bereich
- 2 GiB für Anwendungen

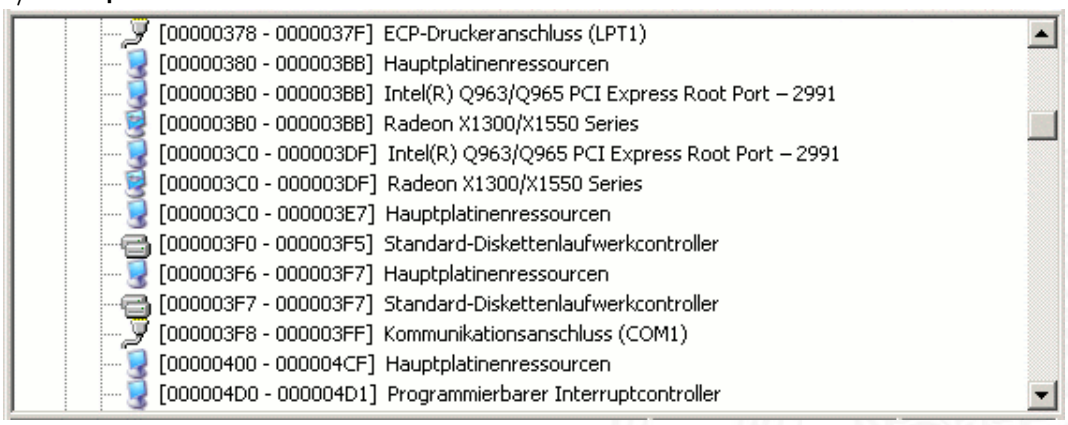
Memory Map: Windows 9x (cont.)



- 32-bit Adressen, 4 GiByte Adressraum
- Aufteilung 2 GiB für Programme, obere 1+1 GiB für Windows
- Beispiel der Zuordnung, diverse Bereiche für I/O reserviert

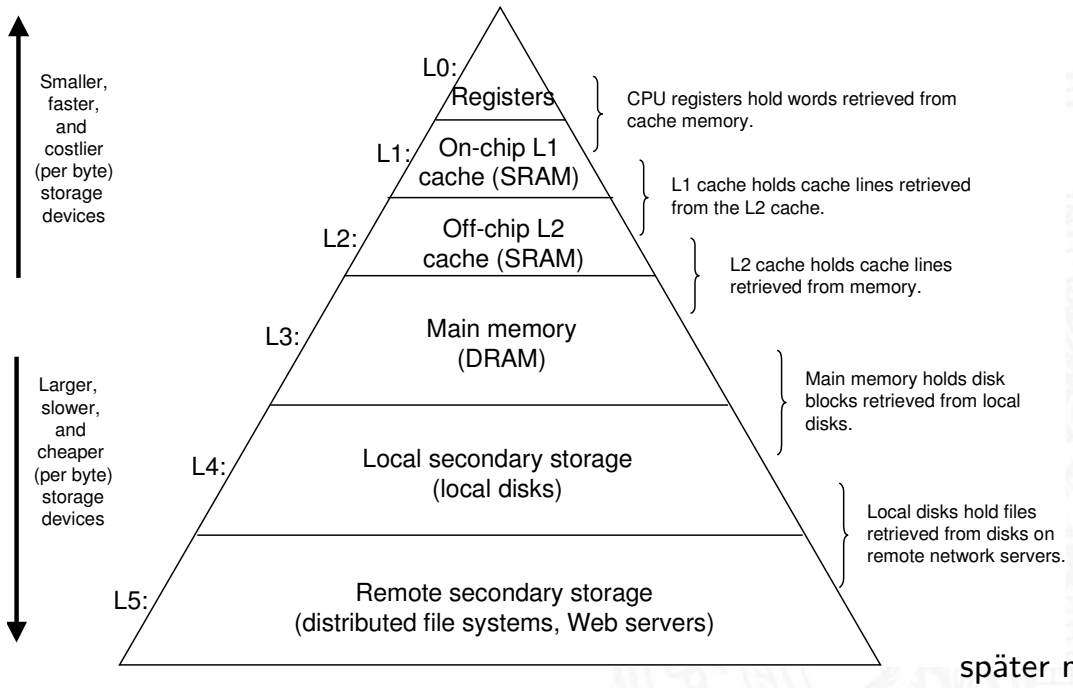
Memory Map: Windows 9x (cont.)

I/O-Speicherbereiche



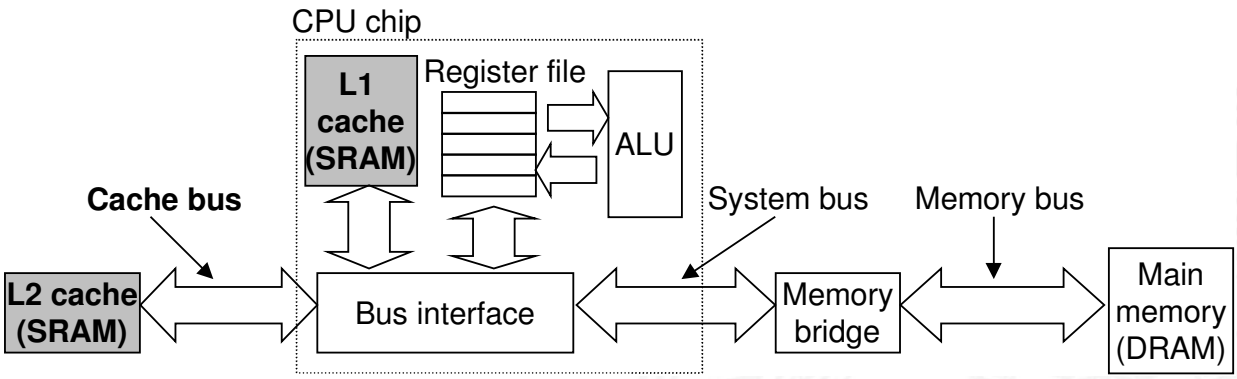
- x86 I/O-Adressraum gesamt nur 64 KiByte
- je nach Zahl der I/O-Geräte evtl. fast voll ausgenutzt
- Adressen vom BIOS zugeteilt

Speicherhierarchie



später mehr...

Cache-Speicher



- verschiedene Strategien
  - Welche Daten sollen in Cache?
  - Welche werden aus Cache entfernt?
- Abbildungsvorschriften (direct-mapped, n-fach assoziativ)
- Organisationsformen

Der Speicher ist wichtig

- Speicher ist nicht unbegrenzt
  - muss zugeteilt und verwaltet werden
  - viele Anwendungen werden vom Speicher dominiert
- Fehler, die auf Speicher verweisen, sind besonders gefährlich
  - Auswirkungen sind sowohl zeitlich als auch räumlich entfernt
- Speicherleistung ist nicht gleichbleibend
  Wechselwirkungen: Speichersystem $\Leftrightarrow$ Programme
  - „Cache“- und „Virtual“-Memory Auswirkungen können Performance/Programmleistung stark beeinflussen
  - Anpassung des Programms an das Speichersystem kann Geschwindigkeit bedeutend verbessern

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
- minimal benötigte Register

PC	Program Counter	Adresse des Befehls
IR	Instruction Register	aktueller Befehl
R0...R31	Registerbank	Rechenregister (Operanden)

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

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

Instruction Decode

„Befehl decodieren“ Phase im Befehlszyklus

▷ Befehl steht im Befehlsregister IR

1. Decoder entschlüsselt Opcode und Operanden
2. leitet Steuersignale an die Funktionseinheiten
3. Programmzähler wird inkrementiert

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

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

Welche Befehle braucht man?

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

CISC – Complex Instruction Set Computer

- Computer-Architekturen mit irregulärem, komplexem Befehlssatz
- typische Merkmale
  - sehr viele Befehle, viele Datentypen
  - komplexe Befehlscodierung, Befehle variabler Länge
  - viele Adressierungsarten
  - Mischung von Register- und Speicheroperanden

  $\Rightarrow$ komplexe Befehle mit langer Ausführungszeit

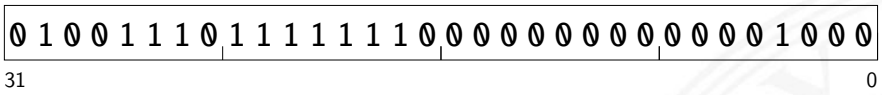
  – Problem: Compiler benutzen solche Befehle gar nicht
- Motivation
  - aus der Zeit der ersten Großrechner, 60er Jahre
  - Assemblerprogrammierung: Komplexität durch viele (mächtige) Befehle umgehen
- Beispiele: Intel 80x86, Motorola 68K, DEC Vax

RISC – Reduced Instruction Set Computer

- Oberbegriff für moderne Rechnerarchitekturen entwickelt ab ca. 1980 bei IBM, Stanford, Berkeley
- auch bekannt unter: „Regular Instruction Set Computer“
- typische Merkmale
  - reguläre Struktur, z.B. 32-bit Wortbreite, 32-bit Befehle
  - nur ein-Wort Befehle
  - alle Befehle in einem Taktschritt ausführbar
  - „Load-Store“ Architektur, keine Speicheroperanden
  - viele universelle Register, keine Spezialregister
  - optimierende Compiler statt Assemblerprogrammierung
- Beispiele: IBM 801, MIPS, SPARC, DEC Alpha, ARM
- Diskussion und Details CISC vs. RISC später

Befehls-Decodierung

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



Wie soll die Hardware diesen Wert interpretieren?

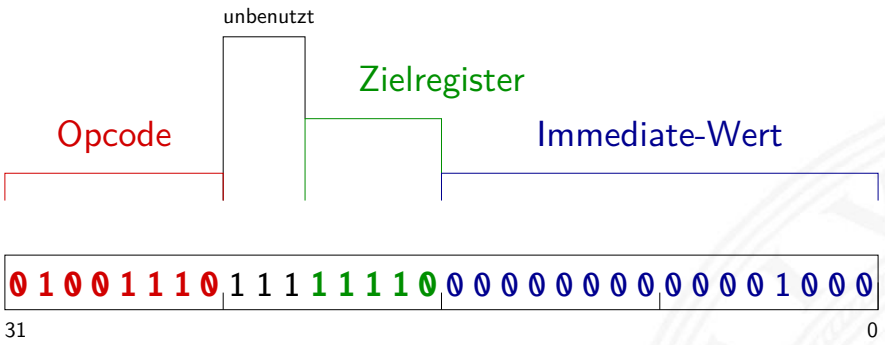
- direkt in einer Tabelle nachschauen (Mikrocode-ROM)
- Problem: Tabelle müsste 2^{32} Einträge haben

⇒ Aufteilung in Felder: Opcode und Operanden

⇒ Decodierung über mehrere, kleine Tabellen

⇒ unterschiedliche Aufteilung für unterschiedliche Befehle: Befehlsformate

Befehlsformate



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

Befehlsformat: drei Beispielarchitekturen

- MIPS: Beispiel für 32-bit RISC Architekturen
  - alle Befehle mit 32-bit codiert
  - nur 3 Befehlsformate (R, I, J)
- D*CORE: Beispiel für 16-bit Architektur
  - siehe RS-Praktikum (64-042) für Details
- Intel x86: Beispiel für CISC-Architekturen
  - irreguläre Struktur, viele Formate
  - mehrere Codierungen für einen Befehl
  - 1-Byte. . . 36-Bytes pro Befehl

Befehlsformat: Beispiel MIPS

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

wenige Befehlsformate

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

MIPS: Übersicht

„Microprocessor without Interlocked Pipeline Stages“

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

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

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

MIPS: Registermodell

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

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

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

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

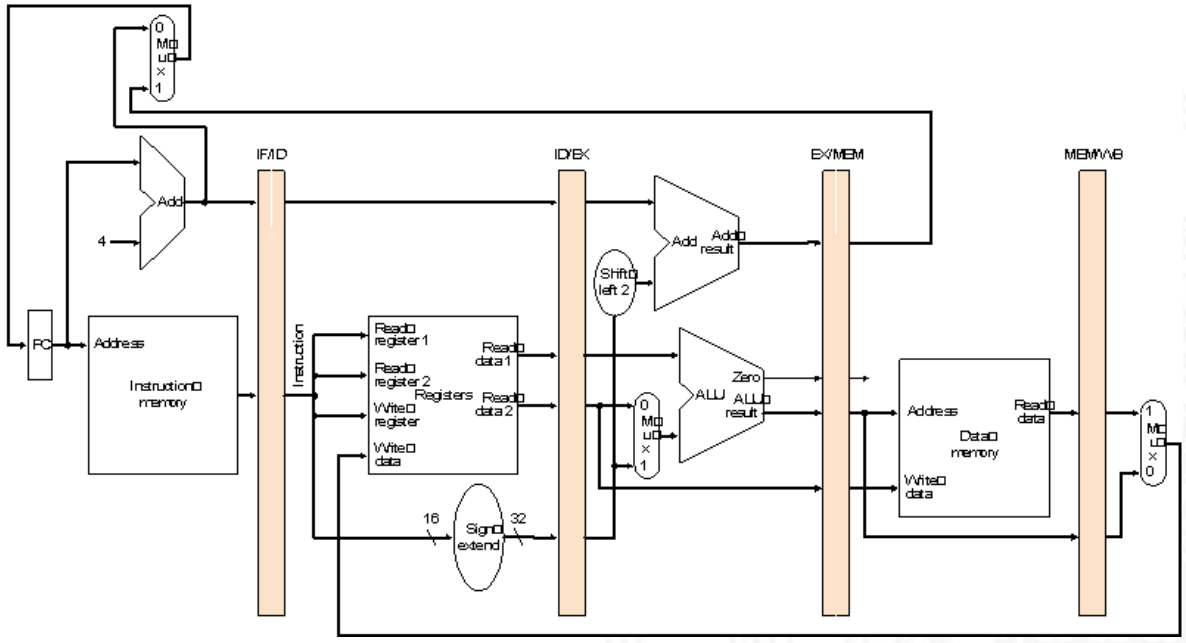
MIPS: Befehlssatz

- Übersicht und Details: David A. Patterson, John L. Hennessy, *Computer Organization and Design : the hardware/software interface*

- dort auch hervorragende Erläuterung der Hardwarestruktur
- klassische fünf-stufige Befehlspipeline

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

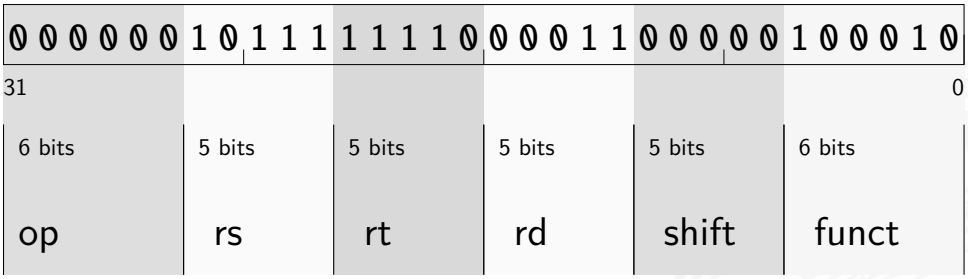
MIPS: Hardwarestruktur



PC	Register	ALUs	Speicher
I-Cache	(R0 .. R31)		D-Cache

Befehlsformat: Beispiel MIPS

Befehl im R-Format

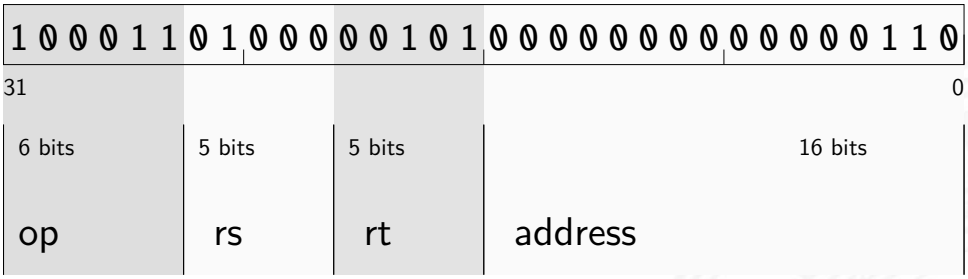


- op: Opcode — Typ des Befehls — 0 = „alu-op“
  rs: source register 1 — erster Operand — 23 = „r23“
  rt: source register 2 — zweiter Operand — 30 = „r30“
  rd: destination register — Zielregister — 3 = „r3“
  shift: shift amount — (optionales Shiften) — 0 = „0“
  funct: ALU function — Rechenoperation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

⇒ r3 = r23 - r30 `sub r3, r23, r30`

Befehlsformat: Beispiel MIPS

Befehl im I-Format



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

⇒ r8 = MEM[r5+addr] `lw r8, addr(r5)`

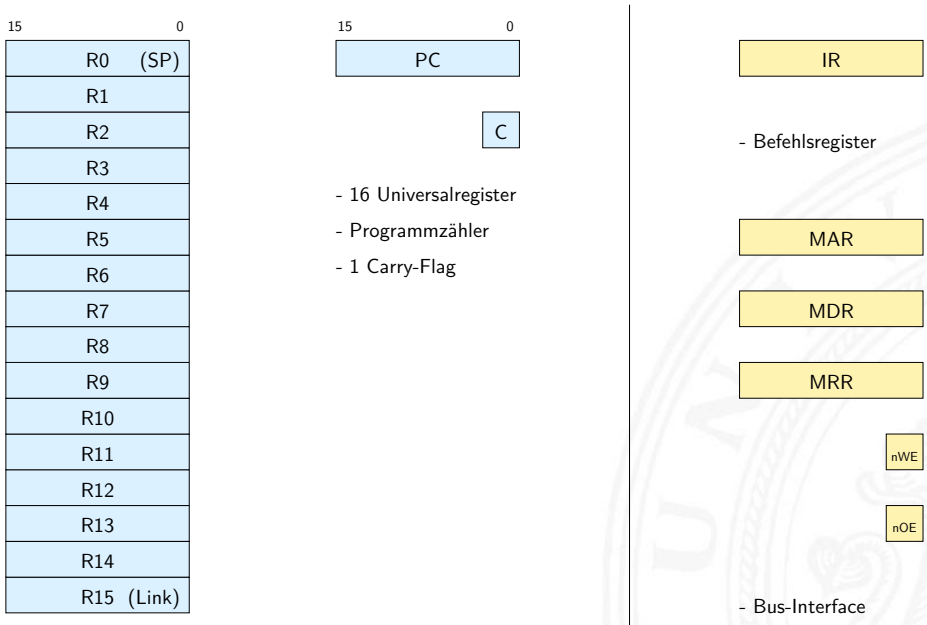
Befehlsformat: Beispiel M*CORE

- 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
  - häufig in Embedded-Systems
  - „smart cards“

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 oder Simulator mit Assembler (winT3asm.exe / t3asm.jar)

D*CORE: Registermodell

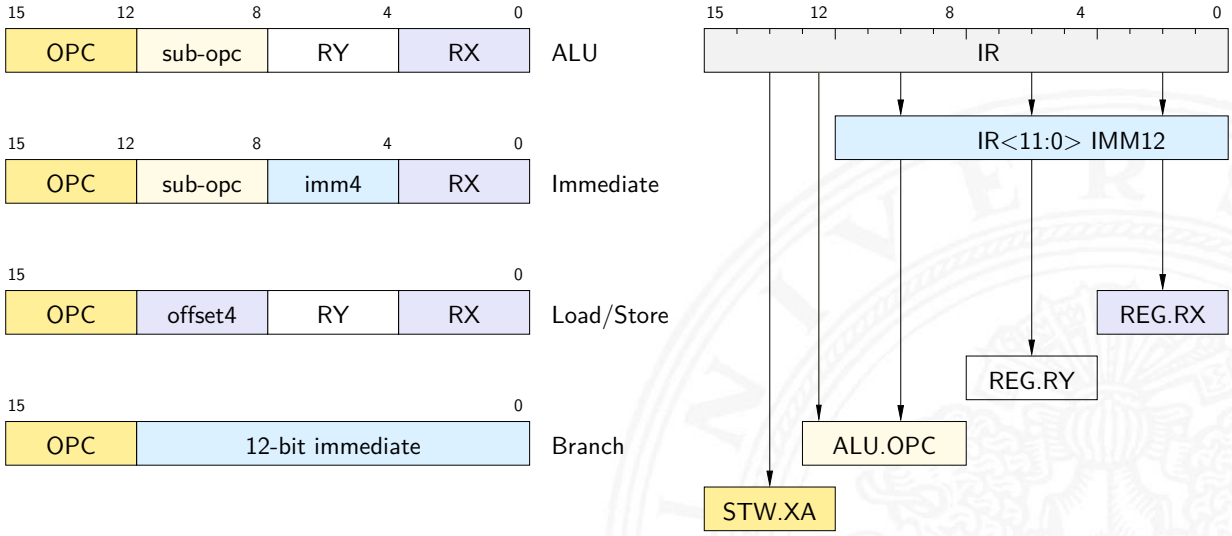


- sichtbar für Programmierer: R0...R15, PC und C

D*CORE: Befehlssatz

`mov`	move register
`addu, addc`	Addition (ohne, mit Carry)
`subu`	Subtraktion
`and, or xor`	logische Operationen
`lsl, lsr, asr`	logische, arithmetische Shifts
`cmpe, cmpne, ...`	Vergleichsoperationen
`movi, addi, ...`	Operationen mit Immediate-Operanden
`ldw, stw`	Speicherzugriffe, load/store
`br, jmp`	unbedingte Sprünge
`bt, bf`	bedingte Sprünge
`jsr`	Unterprogrammaufruf
`trap`	Software interrupt
`rfi`	return from interrupt

D*CORE: Befehlsformate



- 4-bit Opcode, 4-bit Registeradressen
- einfaches Zerlegen des Befehls in die einzelnen Felder

Adressierungsarten

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

$\Rightarrow$ wichtige Unterscheidungsmerkmale für Rechnerarchitekturen

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

Beispiel: Add-Befehl

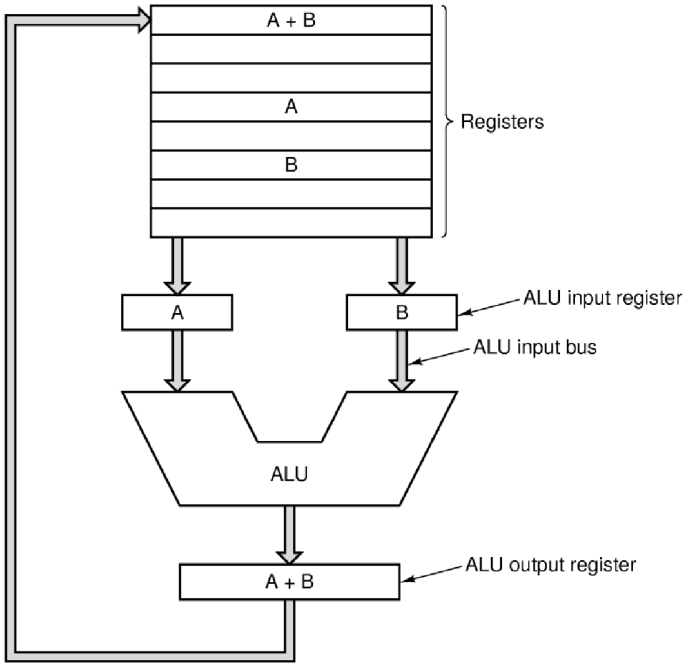
- Rechner soll „rechnen“ können
- typische arithmetische Operation nutzt 3 Variablen
  Resultat, zwei Operanden: $X = Y + Z$

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

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

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.)

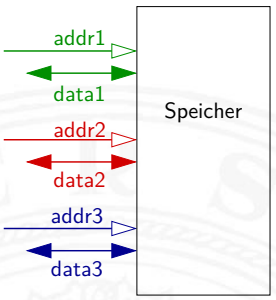


Woher kommen die Operanden?

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

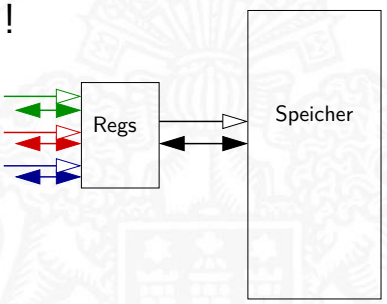
⇒ „Multiport-Speicher“: mit drei Ports ?

- 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 $n = \{3 \ldots 0\}$

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

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

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

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

Beispiel: n-Adress Maschine

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

3-Adress-Maschine

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

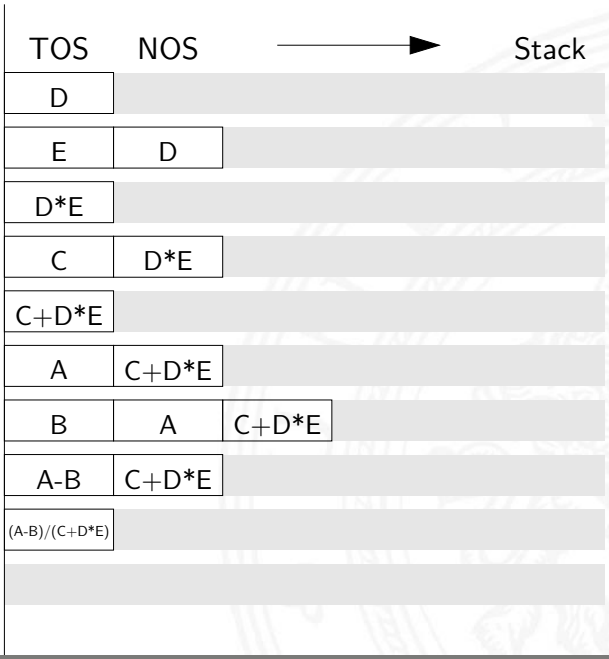
```
push  D
push  E
mul
push  C
add
push  A
push  B
sub
div
pop   Z
```


Beispiel: Stack-Maschine / 0-Adress Maschine

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

0-Adress-Maschine

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



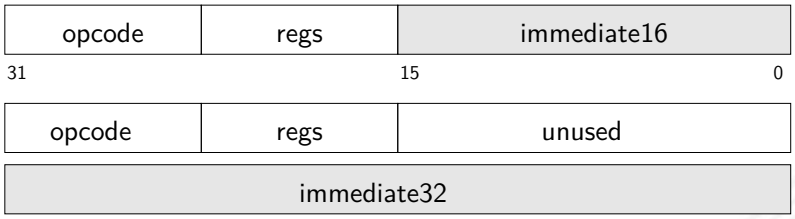
Adressierungsarten

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

Adressierungsarten (cont.)

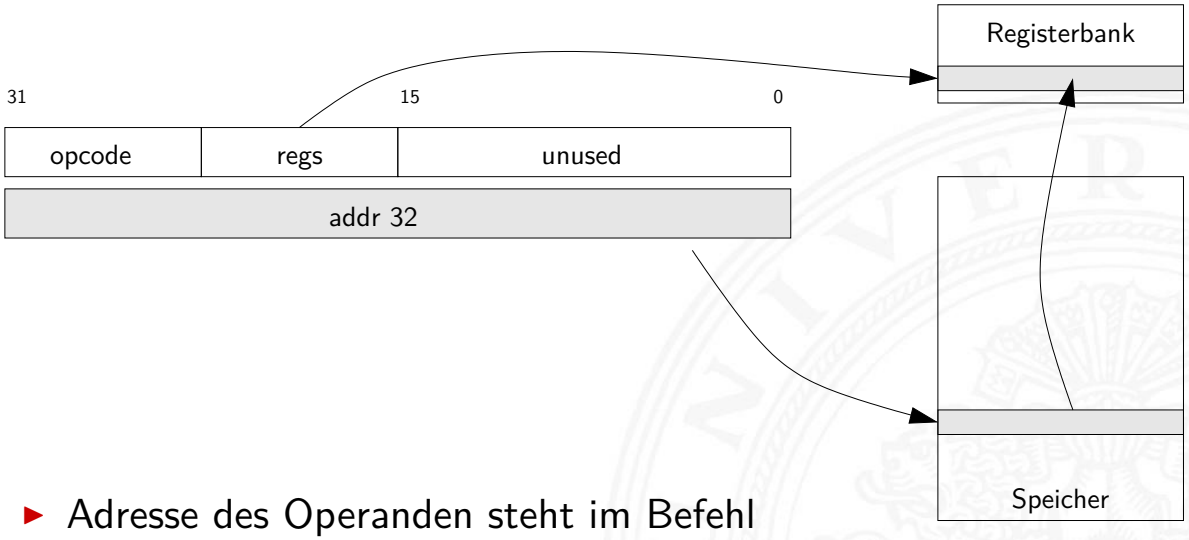
- „register“
  - wie Direktmodus, aber Register statt Speicher
  - 32 Register: benötigen 5 bit im Befehl
  - genug Platz für 2- oder 3-Adress Formate
- „register-indirekt“
  - Befehl spezifiziert ein Register
  - mit der Speicheradresse des Operanden
  - ein zusätzlicher Speicherzugriff
- „indiziert“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

Immediate-Adressierung



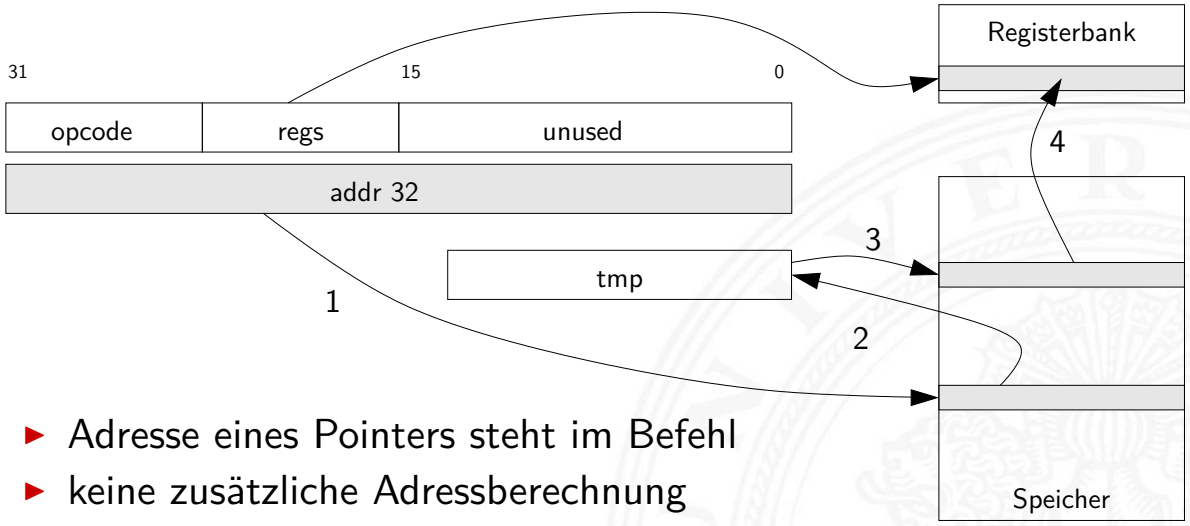
- Operand steht direkt im Befehl, kein zusätzlicher Speicherzugriff
- Länge des Operanden $<$ (Wortbreite - Opcodebreite)
- Darstellung größerer Zahlenwerte
  - 2-Wort Befehle (x86)
    zweites Wort für Immediate-Wert
  - mehrere Befehle (MIPS, SPARC)
    z.B. obere/untere Hälfte eines Wortes
  - Immediate-Werte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

Direkte Adressierung



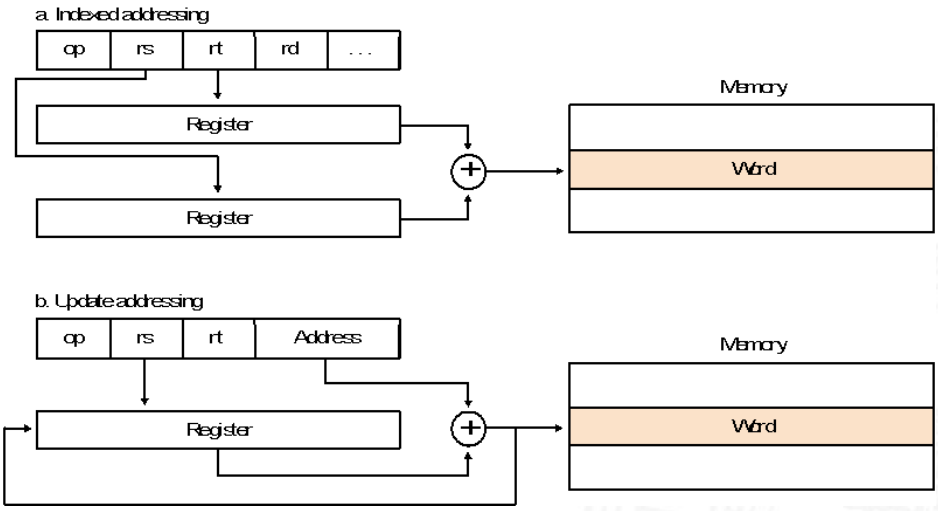
- Adresse des Operanden steht im Befehl
- keine zusätzliche Adressberechnung
- ein zusätzlicher Speicherzugriff: z.B. R3 = MEM[addr32]
- Adressbereich beschränkt, oder 2-Wort Befehl (wie Immediate)

Indirekte Adressierung



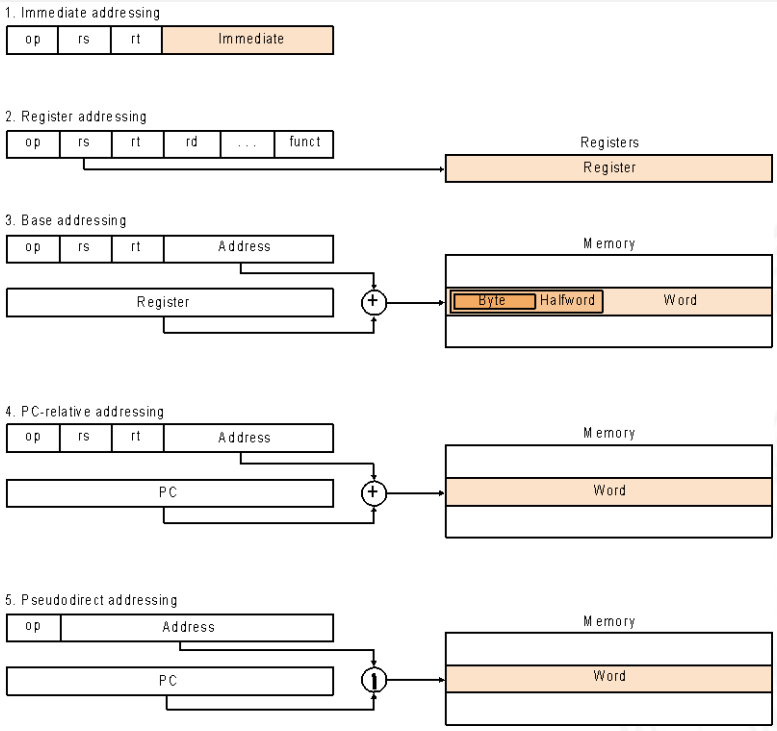
- Adresse eines Pointers steht im Befehl
- keine zusätzliche Adressberechnung
- zwei zusätzliche Speicherzugriffe:
  z.B. tmp = MEM[addr32 ; R3 = MEM[tmp]
- typische CISC-Adressierungsart, viele Taktzyklen
- kommt bei RISC-Rechnern nicht vor

Indizierte Adressierung



- indizierte Adressierung, z.B. für Arrayzugriffe
  - addr = (Basisregister) + (Sourceregister)
  - addr = (Sourceregister) + offset;
    Sourceregister = addr

weitere Adressierungsarten



typische Adressierungsarten

welche Adressierungsarten / Varianten sind üblich?

- 0-Adress (Stack-) Maschine — Java virtuelle Maschine
- 1-Adress (Akkumulator) Maschine — 8-bit Mikrokontroller, 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

Intel x86-Architektur

- übliche Bezeichnung für die Intel-Prozessorfamilie
- von 8086, 80286, 80386, 80486, Pentium. . . Pentium-IV, 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

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
18.5 Instruction Set Architecture - Intel x86-Architektur
64-040 Rechnerstrukturen

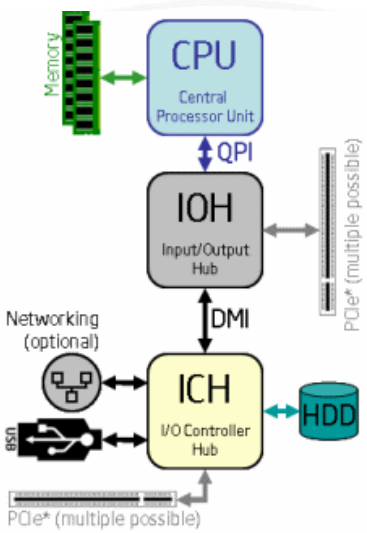
Intel x86: Evolution

Chip	Datum	MHz	Transistoren	Speicher	Anmerkungen
4004	4/1971	0,108	2 300	640	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 IV	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	Taktanpassung (Turbo Boost)
. . .					

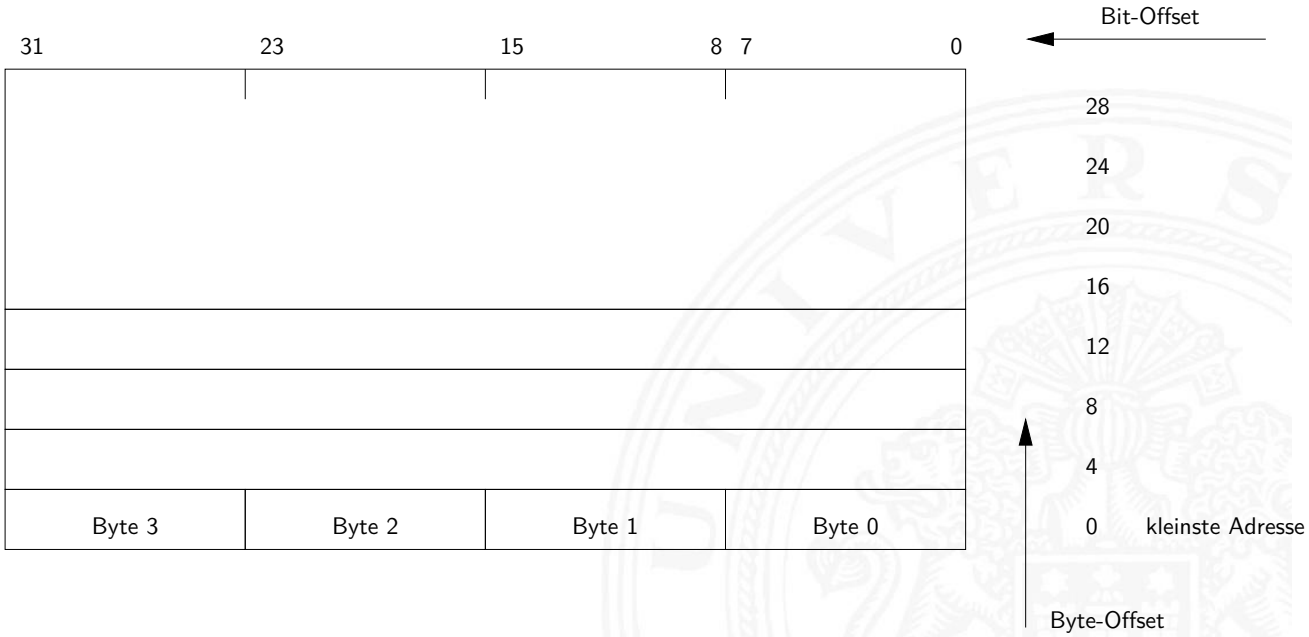
Beispiel: Core i7-960 Prozessor

Taktfrequenz	bis 3,46 GHz
Anzahl der Cores	4 ($\times$ 2 Hyperthreading)
QPI Durchsatz (quick path interconnect)	4,8 GT/s
Bus Interface	64 Bits
L1 Cache	4x (32 kB I + 32kB D)
L2 Cache	4x 256 kB (I+D)
L3 Cache	8192 kB (I+D)
Prozess	45 nm
Versorgungsspannung	0,8 - 1,375V
Wärmeabgabe	$\sim$ 130 W
Performance (SPECint 2006)	$\sim$ 38

Quellen: `ark.intel.com`, `www.spec.org`



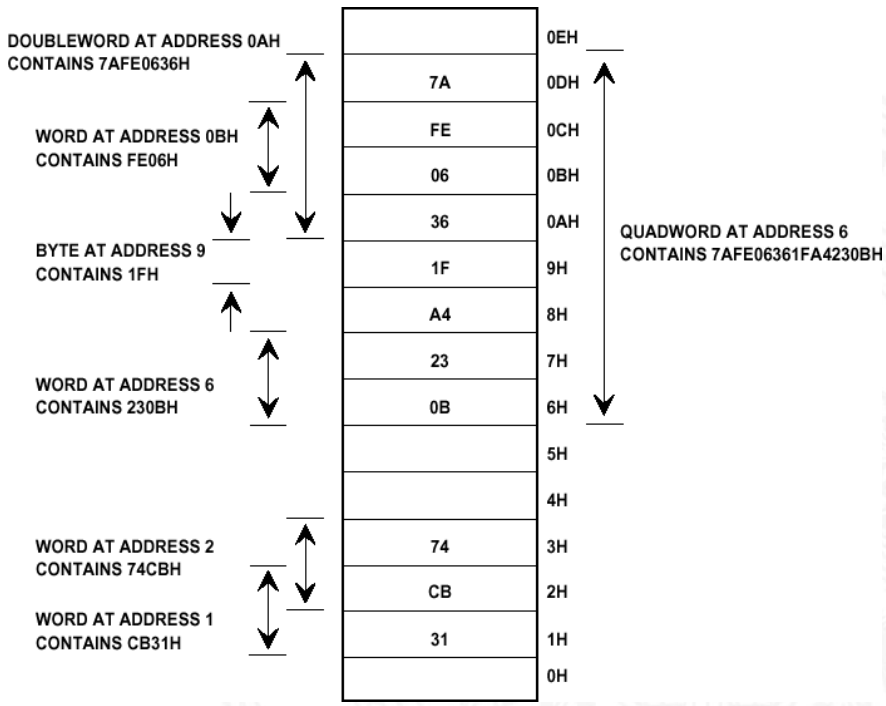
x86: Speichermodell



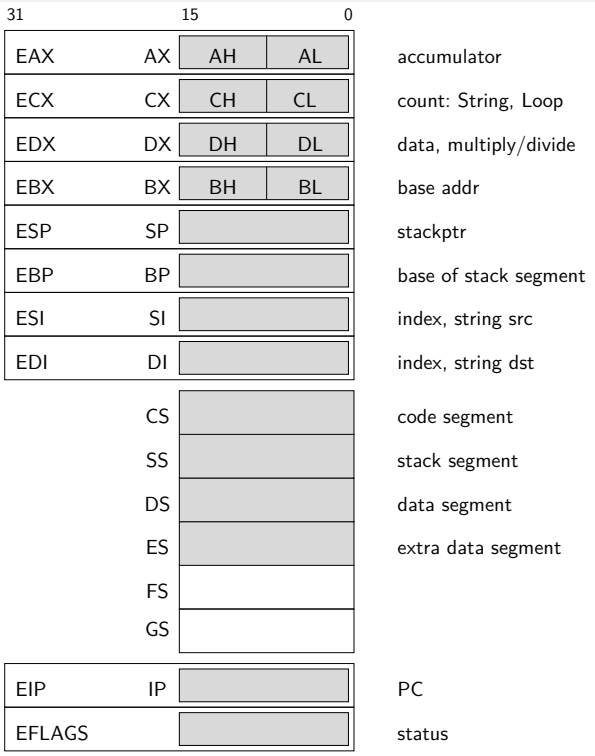
- „little endian“: LSB eines Wortes bei der kleinsten Adresse

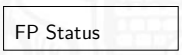
x86: Speichermodell (cont.)

- Speicher voll byte-adressierbar
- misaligned Zugriffe langsam
- Beispiel zeigt
  - Byte
  - Word
  - Doubleword
  - Quadword

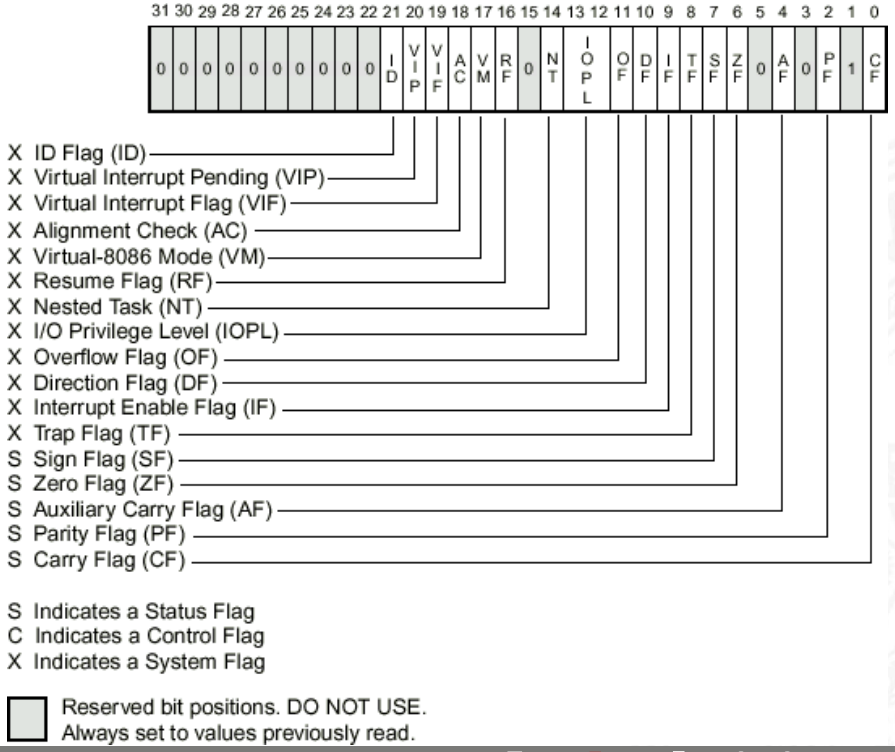


x86: Register



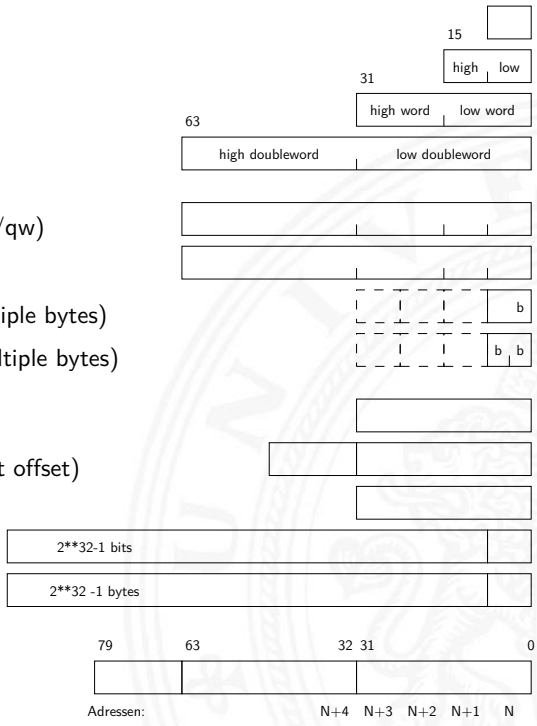


x86: EFLAGS Register



x86: Datentypen

bytes	
word	
doubleword	
quadword	
integer	(2-complement b/w/dw/qw)
ordinal	(unsigned b/w/dw/qw)
BCD	(one digit per byte, multiple bytes)
packed BCD	(two digits per byte, multiple bytes)
near pointer	(32 bit offset)
far pointer	(16 bit segment + 32 bit offset)
bit field	
bit string	
byte string	
float / double / extended	



x86: Befehlssatz

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

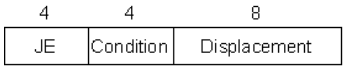
x86: Befehlsformat-Modifier („prefix“)

- alle Befehle können mit Modifiern ergänzt werden

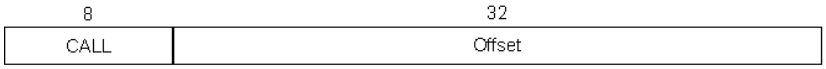
segment override	Adresse aus angewähltem Segmentregister
address size	Umschaltung 16/32-bit Adresse
operand size	Umschaltung 16/32-bit Operanden
repeat	Stringoperationen: für alle Elemente
lock	Speicherschutz bei Multiprozessorsystemen

x86 Befehlscodierung: Beispiele

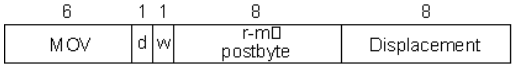
a. JE EIP + displacement



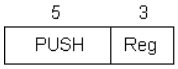
b. CALL



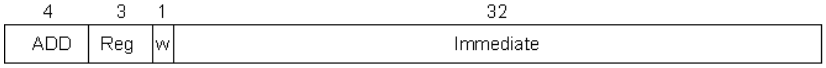
c. MOV EBX, [EDI + 45]



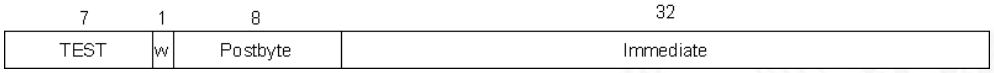
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- 1 Byte. . . 36 Bytes
- vollkommen irregulär
- w: Auswahl 16/32 bit

x86 Befehlscodierung: Beispiele (cont.)

Instruction	Function
JE name	If equal (CC) EIP = name};□ EIP – 128 ≤ name < EIP + 128
JMP name	{EIP = NAME};
CALL name	SP = SP – 4; M[SP] = EIP + 5; EIP = name;
MOVW EBX,[EDI + 45]	EBX = M [EDI + 45]
PUSH ESI	SP = SP – 4; M[SP] = ESI
POP EDI	EDI = M[SP]; SP = SP + 4
ADD EAX,#6765	EAX = EAX + 6765
TEST EDX,#42	Set condition codea (flags) with EDX & 42
MOVSL	M[EDI] = M[ESI];□ EDI = EDI + 4; ESI = ESI + 4

x86: Assembler-Beispiel print(...)

```
addr opcode    assembler                c quellcode
-------------------------------------------------------------------------------
               .file     "hello.c"
               .text
0000 48656C6C  .string  "Hello x86!\\n"
     6F207838
     36210A00
               .text
               print:
0000 55           pushl %ebp             | void print( char* s ) {
0001 89E5         movl %esp,%ebp
0003 53           pushl %ebx
0004 8B5D08       movl 8(%ebp),%ebx
0007 803B00       cmpb $0,(%ebx)         |   while( *s != 0 ) {
000a 7418         je .L18
                  .align 4
               .L19:
000c A100000000   movl stdout,%eax       |     putc( *s, stdout );
0011 50           pushl %eax
0012 0FBE03       movsbl (%ebx),%eax
0015 50           pushl %eax
0016 E8FCFFFF     call _IO_putc
     FF
001b 43           incl %ebx              |     s++;
001c 83C408       addl $8,%esp           |   }
001f 803B00       cmpb $0,(%ebx)
0022 75E8         jne .L19
               .L18:
0024 8B5DFC       movl -4(%ebp),%ebx     | }
0027 89EC         movl %ebp,%esp
0029 5D           popl %ebp
002a C3           ret
```

x86: Assembler-Beispiel main(...)

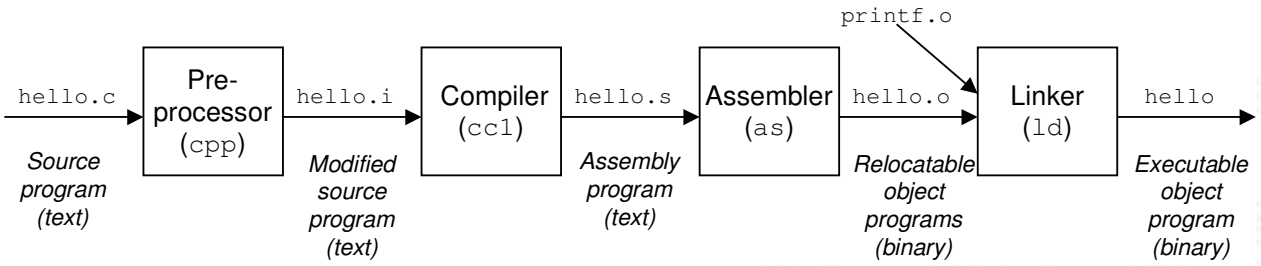
```
addr opcode     assembler                  c quellcode
-------------------------------------------------------------------------------
                .Lfe1:
                .Lscope0:
002b 908D7426      .align 16
     00
                main:
0030 55            pushl %ebp              | int main( int argc, char** argv ) {
0031 89E5          movl %esp,%ebp
0033 53            pushl %ebx

0034 BB00000000    movl $.LC0,%ebx         |   print( "Hello x86!\\n" );
0039 803D0000      cmpb $0,.LC0
     000000
0040 741A          je .L26
0042 89F6          .align 4
                .L24:
0044 A100000000    movl stdout,%eax
0049 50            pushl %eax
004a 0FBE03        movsbl (%ebx),%eax
004d 50            pushl %eax
004e E8FCFFFFFF    call _IO_putc
0053 43            incl %ebx
0054 83C408        addl $8,%esp
0057 803B00        cmpb $0,(%ebx)
005a 75E8          jne .L24
                .L26:
005c 31C0          xorl %eax,%eax          |   return 0;
005e 8B5DFC        movl -4(%ebp),%ebx      | }
0061 89EC          movl %ebp,%esp
0063 5D            popl %ebp
0064 C3            ret
```


Gliederung

Gliederung (cont.)

Gliederung (cont.)

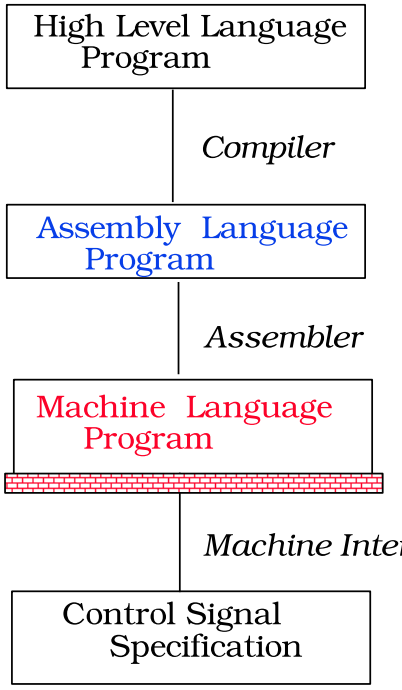
Das Kompilierungssystem



⇒ verschiedene Repräsentationen des Programms

- Hochsprache
- Assembler
- Maschinensprache

Das Kompilierungssystem (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

Warum Assembler?

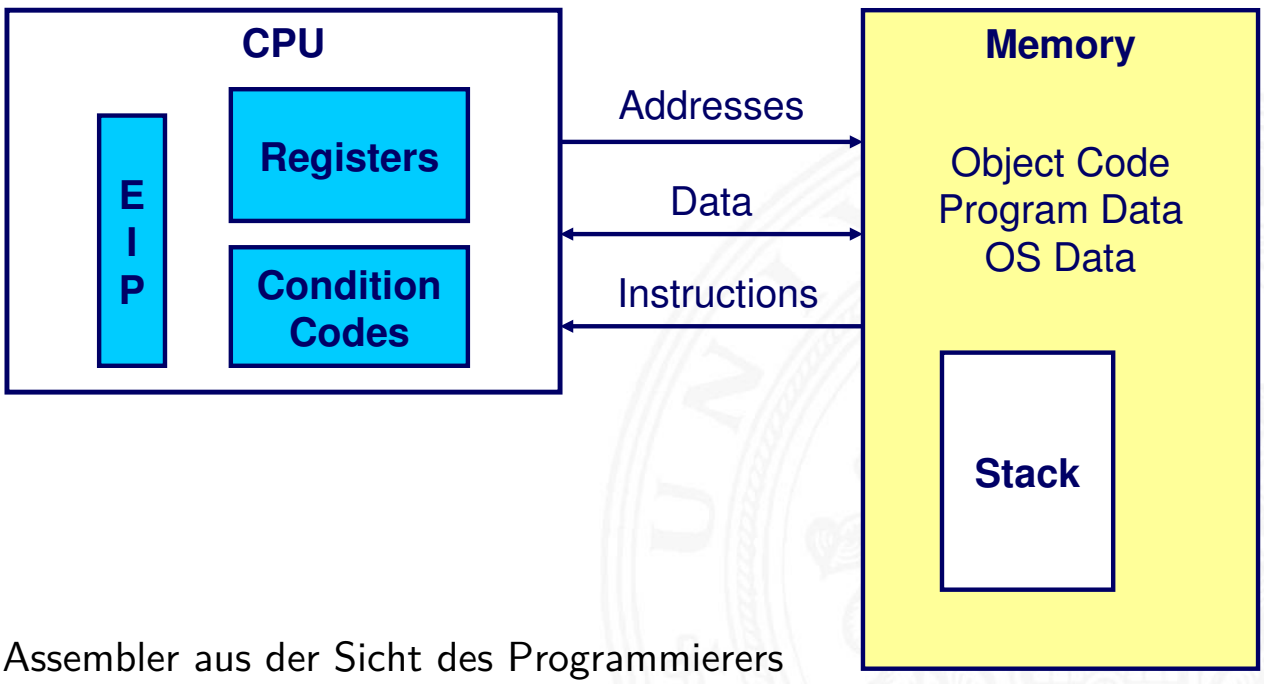
Programme werden nur noch selten in Assembler geschrieben

- 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
- Programmleistung verstärken
  - Ursachen für Programm-Ineffizienz verstehen
  - effiziente „maschinengerechte“ Datenstrukturen / Algorithmen
- Systemsoftware implementieren
  - Compilerbau: Maschinencode als Ziel
  - Betriebssysteme implementieren (Prozesszustände verwalten)
  - Gerätetreiber schreiben

Assembler-Programmierung

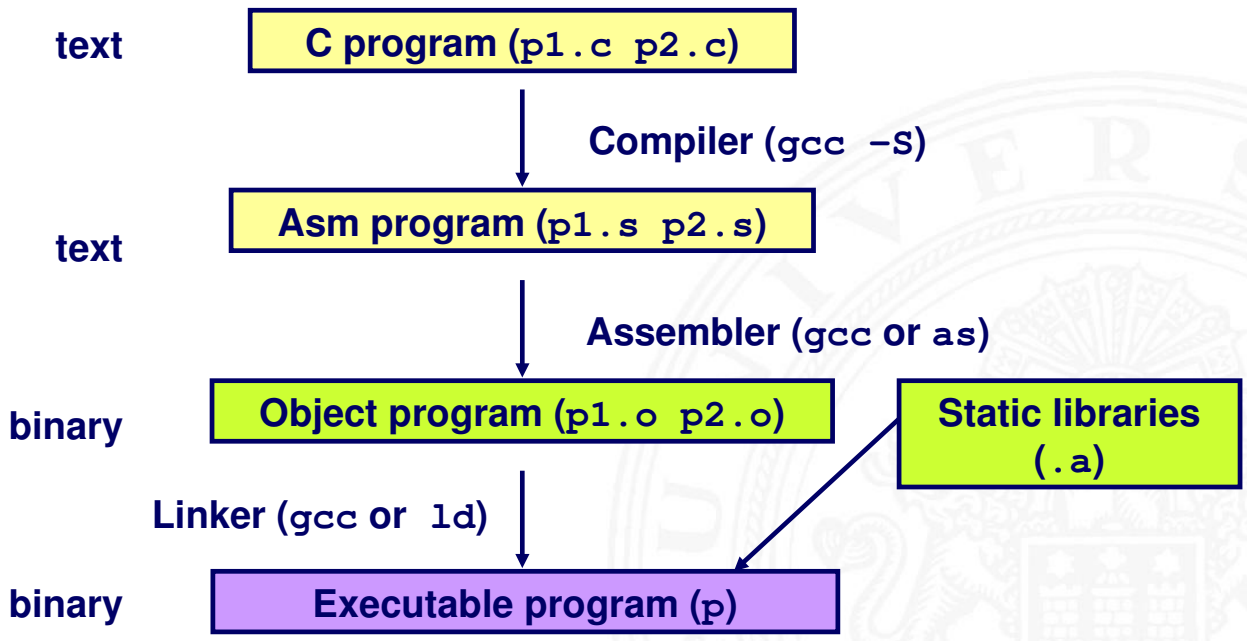


Assembler aus der Sicht des Programmierers

Beobachtbare Zustände (Assemblersicht)

- Programmzähler (*Instruction Pointer* EIP)
  - Adresse der nächsten Anweisung
- Registerbank
  - häufig benutzte Programmdaten
- Zustandscodes
  - gespeicherte Statusinformationen über die letzte arithmetische Operation
  - für bedingte Sprünge benötigt (*Conditional Branch*)
- Speicher
  - byteweise adressierbares Array
  - Code, Nutzerdaten, (einige) OS Daten
  - beinhaltet Kellerspeicher zur Unterstützung von Abläufen

Umwandlung von C in Objektcode



Kompilieren zu Assemblercode

code.c

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

code.s

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

- Befehl `gcc -O -S code.c`
- Erzeugt `code.s`

Assembler Charakteristika

Datentypen

- Ganzzahl- Daten mit 1, 2 oder 4 Bytes
  - Datenwerte
  - Adressen (*pointer*)
- Gleitkomma-Daten mit 4, 8 oder 10/12 Bytes
- keine Aggregattypen wie Arrays oder Strukturen
  - nur fortlaufend adressierbare Byte im Speicher

Assembler Charakteristika (cont.)

Primitive Operationen

- arithmetische/logische Funktionen auf Registern und Speicher
- Datentransfer zwischen Speicher und Registern
  - Daten aus Speicher in Register laden
  - Registerdaten im Speicher ablegen
- Kontrolltransfer
  - unbedingte / Bedingte Sprünge
  - Unterprogrammaufrufe: Sprünge zu/von Prozeduren

Objektcode

- 13 bytes
- Instruktionen: 1-, 2- oder 3 bytes
- Startadresse: `0x401040`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Assembler und Linker

Assembler

- übersetzt `.s` zu `.o`
- binäre Codierung jeder Anweisung
- (fast) vollständiges Bild des ausführbaren Codes
- Verknüpfungen zwischen Code in verschiedenen Dateien fehlen

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 zur Laufzeit erstellt

Beispiel: Maschinenbefehl

- C-Code
  - addiert zwei Ganzzahlen mit Vorzeichen

```
int t = x+y;
```

- Assembler
  - Addiere zwei 4-byte Integer
    - `long` Wörter (für gcc)
    - keine signed/unsigned Unterscheidung
  - Operanden

    `x:` Register `%eax`
    `y:` Speicher `M[%ebp+8]`
    `t:` Register `%eax`
    Ergebnis in `%eax`

```
addl 8(%ebp),%eax
```

Similar to expression
`x += y`

- Objektcode
  - 3-Byte Befehl
  - Speicheradresse `0x401046`

```
0x401046:    03 45 08
```

Objektcode Disassembler: objdump

```
00401040 <_sum>:
   0:      55                   push   %ebp
   1:      89 e5                mov    %esp,%ebp
   3:      8b 45 0c             mov    0xc(%ebp),%eax
   6:      03 45 08             add    0x8(%ebp),%eax
   9:      89 ec                mov    %ebp,%esp
   b:      5d                   pop    %ebp
   c:      c3                   ret
   d:      8d 76 00             lea    0x0(%esi),%esi
```

- `objdump -d ...`
  - Werkzeug zur Untersuchung des Objektcodes
  - rekonstruiert aus Binärcode den Assemblercode
  - kann auf vollständigem, ausführbaren Programm (`a.out`) oder einer `.o` Datei ausgeführt werden

Alternativer Disassembler: gdb

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Disassembled

```
0x401040 <sum>:      push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x401049 <sum+9>:    mov    %ebp,%esp
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea    0x0(%esi),%esi
```

gdb Debugger

`gdb p`

`disassemble sum`

- **Disassemble procedure**

`x/13b sum`

- **Examine the 13 bytes starting at `sum`**

Was kann „disassembliert“ werden?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55              push   %ebp
30001001:  8b ec           mov    %esp,%ebp
30001003:  6a ff           push   $0xffffffff
30001005:  68 90 10 00 30  push   $0x30001090
3000100a:  68 91 dc 4c 30  push   $0x304cdc91
```

- alles, was als ausführbarer Code interpretiert werden kann
- Disassembler untersucht Bytes und rekonstruiert Assemblerquelle

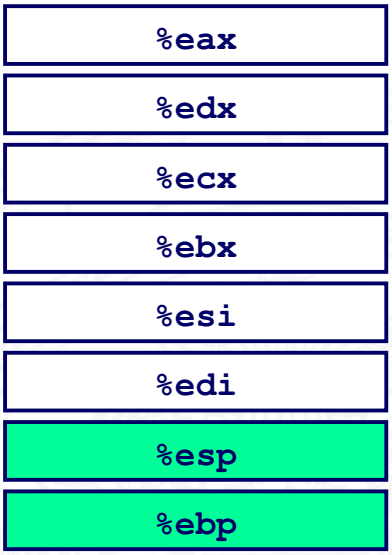
x86 Assemblerprogrammierung

- Adressierungsarten
- arithmetische Operationen
- Statusregister
- Umsetzung von Programmstrukturen

Datentransfer „move“

- Format: `movl` ⟨*src*⟩ , ⟨*dst*⟩
- transferiert ein 4-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: 8 Ganzzahl-Registern
    - %esp und %ebp für spezielle Aufgaben reserviert
    - z.T. andere Spezialregister für andere Anweisungen
  - Speicher: 4 konsekutive Speicherbytes
    - Zahlreiche Adressmodi



`movl` Operanden-Kombinationen

	Source	Destination		C Analogon
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

Elementare Befehle und Adressierungsarten

- Normal: (R) → Mem[Reg[R]]
  - Register R spezifiziert die Speicheradresse
  - Beispiel: `movl (%ecx), %eax`
- Displacement: D(R) → Mem[Reg[R]+D]
  - Register R
  - Konstantes „Displacement“ D spezifiziert den „offset“
  - Beispiel: `movl 8(%ebp), %edx`

Beispiel: einfache Adressierungsmodi

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish

indizierte Adressierung

- gebräuchlichste Form
  - `Imm(Rb,Ri,S)` → `Mem[Reg[Rb]+S*Reg[Ri]+Imm]`
    - ⟨*Imm*⟩ Offset
    - ⟨*Rb*⟩ Basisregister: eins der 8 Integer-Registern
    - ⟨*Ri*⟩ Indexregister: jedes außer %esp
      %ebp grundsätzlich möglich, jedoch unwahrscheinlich
    - ⟨*S*⟩ Skalierungsfaktor 1, 2, 4 oder 8
- spezielle Fälle
  - `(Rb,Ri)` → `Mem[Reg[Rb] + Reg[Ri]]`
  - `Imm(Rb,Ri)` → `Mem[Reg[Rb] + Reg[Ri] + Imm]`
  - `(Rb,Ri,S)` → `Mem[Reg[Rb] + S*Reg[Ri]]`

Beispiel: Adressberechnung

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

Arithmetische Operationen

- binäre Operatoren

Format	Computation	
`addl` *Src,Dest*	*Dest* = *Dest* + *Src*	
`subl` *Src,Dest*	*Dest* = *Dest* – *Src*	
`imull` *Src,Dest*	*Dest* = *Dest* * *Src*	
`sall` *Src,Dest*	*Dest* = *Dest* << *Src*	also called `shll`
`sarl` *Src,Dest*	*Dest* = *Dest* >> *Src*	Arithmetic
`shrl` *Src,Dest*	*Dest* = *Dest* >> *Src*	Logical
`xorl` *Src,Dest*	*Dest* = *Dest* ^ *Src*	
`andl` *Src,Dest*	*Dest* = *Dest* & *Src*	
`orl` *Src,Dest*	*Dest* = *Dest* | *Src*	

Arithmetische Operationen (cont.)

- unäre Operatoren

Format	Computation
`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* – 1
`negl` *Dest*	*Dest* = – *Dest*
`notl` *Dest*	*Dest* = ~ *Dest*

Beispiel: arithmetische Operationen

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```

```
arith:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

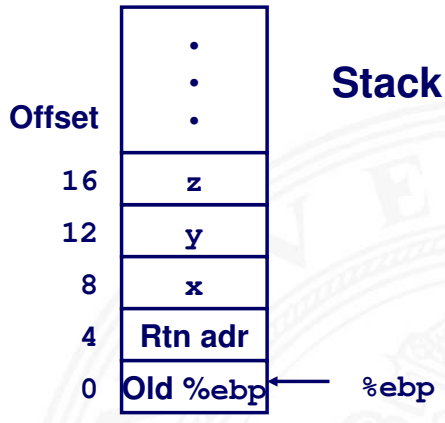
Set Up

Body

Finish

Beispiel: arithmetische Operationen (cont.)

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
movl 8(%ebp),%eax        # eax = x
movl 12(%ebp),%edx       # edx = y
leal (%edx,%eax),%ecx    # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx             # edx = 48*y (t4)
addl 16(%ebp),%ecx       # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax   # eax = 4+t4+x (t5)
imull %ecx,%eax          # eax = t5*t2 (rval)
```

Beispiel: logische Operationen

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192, 2^{13} - 7 = 8185$

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y     (t1)
sarl $17,%eax          eax = t1>>17  (t2)
andl $8185,%eax        eax = t2 & 8185
```


Kontrollfluss / Programmstrukturen

- Zustandscodes
  - Setzen
  - Testen
- Ablaufsteuerung
  - Verzweigungen: „If-then-else“
  - Schleifen: „Loop“-Varianten
  - Mehrfachverzweigungen: „Switch“

Zustandscodes

- vier relevante „Flags“ im Statusregister
  - CF Carry Flag
  - SF Sign Flag
  - ZF Zero Flag
  - OF Overflow Flag

1. implizite Aktualisierung durch arithmetische Operationen
   - Beispiel: addl $\langle src \rangle, \langle dst \rangle$ in C: t=a+b
   - CF höchstwertiges Bit generiert Übertrag: Unsigned-Überlauf
   - ZF wenn $t = 0$
   - SF wenn $t < 0$
   - OF wenn das Zweierkomplement überläuft
     $(a > 0 \;\&\&\; b > 0 \;\&\&\; t < 0) \;||\; (a < 0 \;\&\&\; b < 0 \;\&\&\; t \geq 0)$

Zustandscodes (cont.)

2. explizites Setzen durch Vergleichsoperation
   - Beispiel: cmpl $\langle src2 \rangle, \langle src1 \rangle$
     wie Berechnung von $\langle src1 \rangle - \langle src2 \rangle$ (subl $\langle src2 \rangle, \langle src1 \rangle$)
     jedoch ohne Abspeichern des Resultats
   - CF höchstwertiges Bit generiert Übertrag
   - ZF setzen wenn $src1 = src2$
   - SF setzen wenn $(src1 - src2) < 0$
   - OF setzen wenn das Zweierkomplement überläuft
     $(a > 0 \;\&\&\; b < 0 \;\&\&\; (a - b) < 0) \;||$
     $(a < 0 \;\&\&\; b > 0 \;\&\&\; (a - b) \geq 0)$

Zustandscodes (cont.)

3. explizites Setzen durch Testanweisung
   - Beispiel: `testl` $\langle src2 \rangle, \langle src1 \rangle$
     wie Berechnung von $\langle src1 \rangle \& \langle src2 \rangle$ (`andl` $\langle src2 \rangle, \langle src1 \rangle$)
     jedoch ohne Abspeichern des Resultats

   $\Rightarrow$ hilfreich, wenn einer der Operanden eine Bitmaske ist

   - ZF setzen wenn $src1 \& src2 = 0$
   - SF setzen wenn $src1 \& src2 < 0$

Zustandscodes lesen

`set`.. Anweisungen

- Kombinationen von Zustandscodes setzen einzelnes Byte

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Beispiel: Zustandscodes lesen

- ein-Byte Zieloperand (Register, Speicher)
- meist kombiniert mit `movzbl` (Löschen hochwertiger Bits)

%eax	%ah	%al
%edx	%dh	%dl
%ecx	%ch	%cl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

```
int gt (int x, int y)
{
  return x > y;
}
```

```
movl 12(%ebp),%eax   # eax = y
cmpl %eax,8(%ebp)    # Compare x : y
setg %al             # al = x > y
movzbl %al,%eax      # Zero rest of %eax
```


Sprungbefehle („Jump“)

j.. Anweisungen

- unbedingter- / bedingter Sprung (abhängig von Zustandscode)

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Beispiel: bedingter Sprung („Conditional Branch“)

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
        pushl %ebp
        movl %esp,%ebp

        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle L9
        movl %edx,%eax
L9:

        movl %ebp,%esp
        popl %ebp
        ret
```

Set Up

Body

Finish

Beispiel: bedingter Sprung („Conditional Branch“) (cont.)

```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)
    goto done;
  rval = x;
done:
  return rval;
}
```

- C-Code mit goto
- entspricht mehr dem Assemblerprogramm
- schlechter Programmierstil (!)

```
    movl 8(%ebp),%edx   # edx = x
    movl 12(%ebp),%eax  # eax = y
    cmpl %eax,%edx      # x : y
    jle L9              # if <= goto L9
    movl %edx,%eax      # eax = x
L9:                     # Done:
```

Skipped when x ≤ y

Beispiel: „Do-While“ Schleife

- C Code

```
int fact_do
  (int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Rückwärtssprung setzt Schleife fort
- wird nur ausgeführt, wenn „while“ Bedingung gilt

Beispiel: „Do-While“ Schleife (cont.)

```
int fact_goto
  (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

Register

```
%edx   x
%eax   result
```

```
_fact_goto:
  pushl %ebp          # Setup
  movl %esp,%ebp      # Setup
  movl $1,%eax        # eax = 1
  movl 8(%ebp),%edx   # edx = x

L11:
  imull %edx,%eax     # result *= x
  decl %edx           # x--
  cmpl $1,%edx        # Compare x : 1
  jg L11              # if > goto loop

  movl %ebp,%esp      # Finish
  popl %ebp           # Finish
  ret                 # Finish
```

„Do-While“ Übersetzung

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- beliebige Folge von C Anweisungen als Schleifenkörper
- Abbruchbedingung ist zurückgelieferter Integer Wert
  - $= 0$ entspricht Falsch
  - $\neq 0$ –"– Wahr

„While“ Übersetzung

C Code

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

„For“ Übersetzung

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update ;
}
```

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```


Mehrfachverzweigungen „Switch“

- Implementierungsoptionen
  1. Serie von Bedingungen
     - \+ gut bei wenigen Alternativen
     - – langsam bei vielen Fällen
  2. Sprungtabelle „Jump Table“
     - Vermeidet einzelne Abfragen
     - möglich falls Alternativen kleine ganzzahlige Konstanten sind
- Compiler (gcc) wählt eine der beiden Varianten entsprechend der Fallstruktur

```
typedef enum
 {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

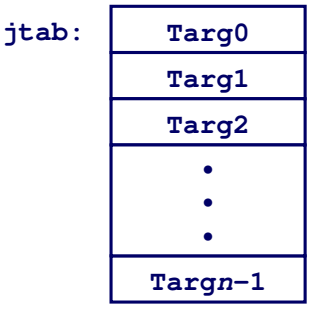
Anmerkung: im Beispielcode fehlt „Default“

Sprungtabelle

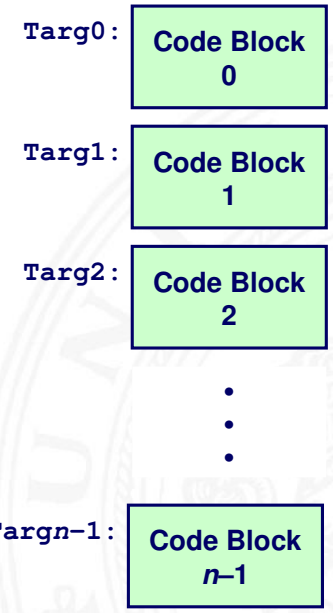
Switch Form

```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets



Approx. Translation

```
target = JTab[op];
goto *target;
```

- Vorteil: k-fach Verzweigung in $\mathcal{O}(1)$ Operationen

Beispiel: „Switch“

Branching Possibilities

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

Enumerated Values

```
ADD   0
MULT  1
MINUS 2
DIV   3
MOD   4
BAD   5
```

Setup:

```
unparse_symbol:
   pushl %ebp              # Setup
   movl %esp,%ebp          # Setup
   movl 8(%ebp),%eax       # eax = op
   cmpl $5,%eax            # Compare op : 5
   ja .L49                 # If > goto done
   jmp *.L57(,%eax,4) # goto Table[op]
```

Beispiel: „Switch“ (cont.)

Erklärung des Assemblers

- symbolische Label
  - Assembler übersetzt Label der Form `.L...` in Adressen
- Tabellenstruktur
  - jedes Ziel benötigt 4 Bytes
  - Basisadresse bei `.L57`
- Sprünge
  - `jmp .L49` als Sprungziel
  - `jmp *.L57(,%eax, 4)`
    - Sprungtabell ist mit Label `.L57` gekennzeichnet
    - Register %eax speichert *op*
    - Skalierungsfaktor 4 für Tabellenoffset
    - Sprungziel: effektive Adresse `.L57` + *op*×4

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
19.4.5 Assembler-Programmierung - x86 Assemblerprogrammierung - Mehrfachverzweigung (Switch) 64-040 Rechnerstrukturen

Beispiel: „Switch“ (cont.)

Table Contents

```
.section .rodata
   .align 4
.L57:
  .long .L51 #Op = 0
  .long .L52 #Op = 1
  .long .L53 #Op = 2
  .long .L54 #Op = 3
  .long .L55 #Op = 4
  .long .L56 #Op = 5
```

Enumerated Values

```
ADD    0
MULT   1
MINUS  2
DIV    3
MOD    4
BAD    5
```

Targets & Completion

```
.L51:
    movl $43,%eax # '+'
    jmp .L49
.L52:
    movl $42,%eax # '*'
    jmp .L49
.L53:
    movl $45,%eax # '-'
    jmp .L49
.L54:
    movl $47,%eax # '/'
    jmp .L49
.L55:
    movl $37,%eax # '%'
    jmp .L49
.L56:
    movl $63,%eax # '?'
    # Fall Through to .L49
```

Sprungtabelle aus Binärcode Extrahieren

```
Contents of section .rodata:
 8048bc0 30870408 37870408 40870408 47870408  0...7...@...G...
 8048bd0 50870408 57870408 46616374 28256429  P...W...Fact(%d)
 8048be0 203d2025 6c640a00 43686172 203d2025   = %ld..Char = %
 ...
```

- im read-only Datensegment gespeichert (`.rodata`)
  - dort liegen konstante Werte des Codes
- kann mit `objdump` untersucht werden
  `objdump code-examples -s --section=.rodata`
  - zeigt alles im angegebenen Segment
  - schwer zu lesen (!)
  - Einträge der Sprungtabelle in umgekehrter Byte-Anordnung
    z.B: `30870408` ist eigentlich `0x08048730`

Zusammenfassung – Assembler

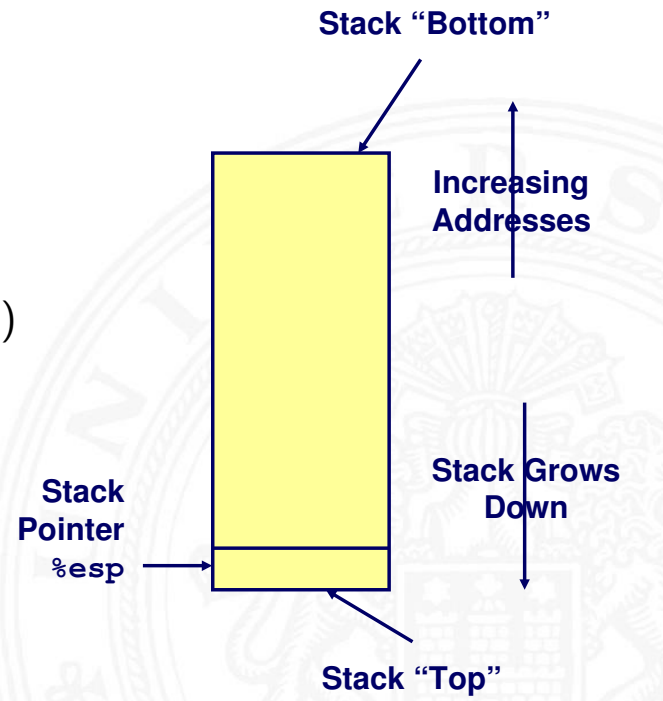
- C Kontrollstrukturen
  - „if-then-else“
  - „do-while“, „while“, „for“
  - „switch“
- Assembler Kontrollstrukturen
  - „Jump“
  - „Conditional Jump“
- Compiler
  - erzeugt Assembler Code für komplexere C Kontrollstrukturen
  - alle Schleifen in „do-while“ / „goto“ Form konvertieren
  - Sprungtabellen für Mehrfachverzweigungen „case“

Zusammenfassung – Assembler (cont.)

- Bedingungen CISC-Rechner
  - typisch Zustandscode-Register (wie die x86-Architektur)
- Bedingungen RISC-Rechner
  - keine speziellen Zustandscode-Register
  - stattdessen werden Universalregister benutzt um Zustandsinformationen zu speichern
  - spezielle Vergleichs-Anweisungen
    z.B. DEC-Alpha: `cmple $16, 1, $1`
    setzt Register `$1` auf 1 wenn Register `$16` ≤ 1

x86 Stack (Kellerspeicher)

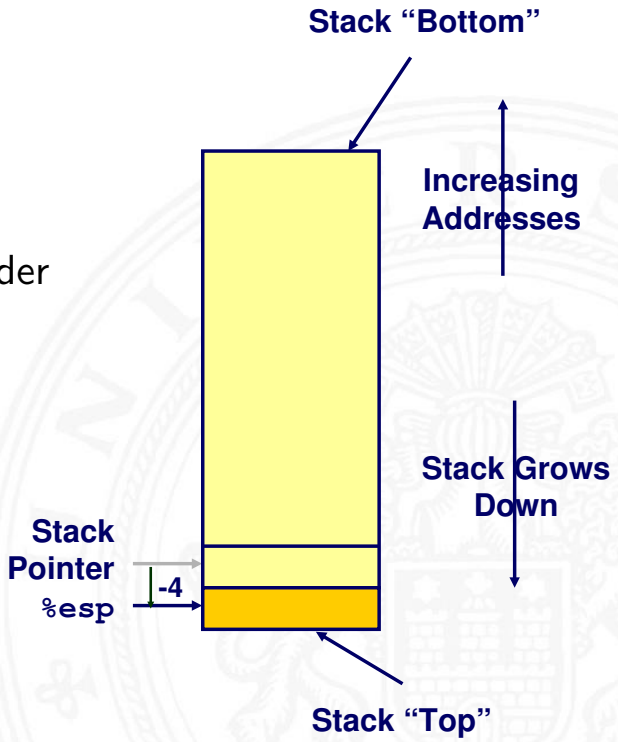
- Speicherregion
- Zugriff mit Stackoperationen
- wächst in Richtung niedrigerer Adressen
- Register `%esp` („Stack-Pointer“)
  - aktuelle Stack-Adresse
  - oberstes Element



Stack: Push

`pushl` ⟨*src*⟩

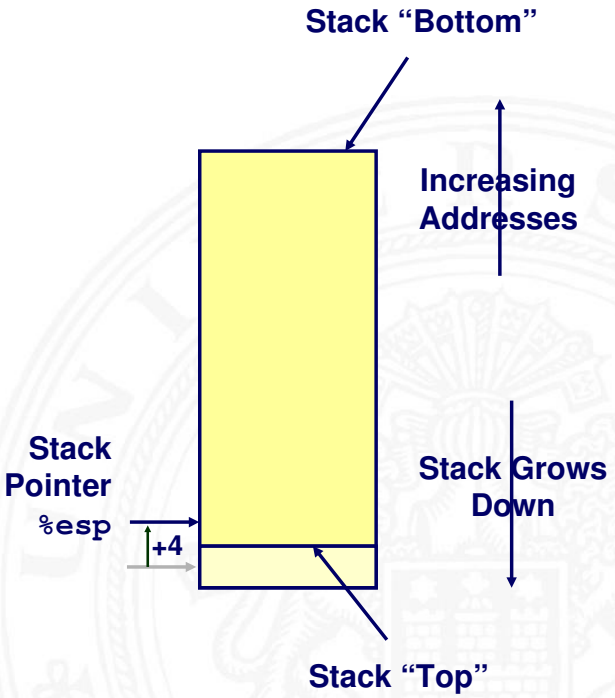
- holt Operanden aus ⟨*src*⟩
- dekrementiert %esp um 4
- speichert den Operanden unter der von %esp vorgegebenen Adresse



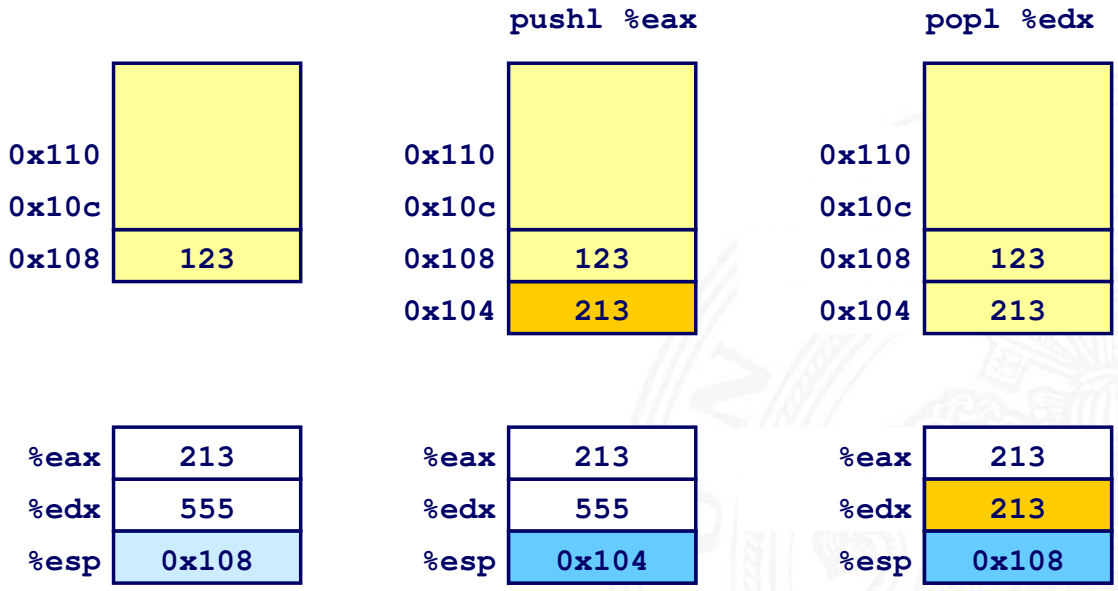
Stack: Pop

`popl` ⟨*dst*⟩

- liest den Operanden unter der von %esp vorgegebenen Adresse
- inkrementiert %esp um 4
- schreibt gelesenen Wert in ⟨*dst*⟩



Beispiele: Stack-Operationen



Prozeduraufruf

- Stack zur Unterstützung von `call` und `ret`
- Prozeduraufruf: `call` ⟨*label*⟩
  - Rücksprungadresse auf Stack („Push“)
  - Sprung zu ⟨*label*⟩
- Wert der Rücksprungadresse
  - Adresse der auf den `call` folgenden Anweisung
  - Beispiel:

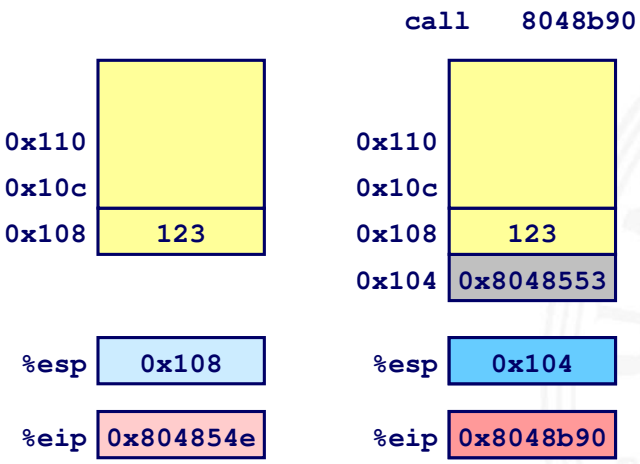
```
804854e:   e8 3d 06 00 00   ;call 8048b90
8048553:   50               ;pushl %eax
⟨main⟩     ...              ;...
8048b90:                    ;Prozedureinsprung
⟨proc⟩     ...              ;...
...        ret              ;Rücksprung
```

  - Rücksprungadresse `0x8048553`
- Rücksprung `ret`
  - Rücksprungadresse vom Stack („Pop“)
  - Sprung zu dieser Adresse

Beispiel: Prozeduraufruf

- Prozeduraufruf `call`

```
804854e:  e8 3d 06 00 00       call   8048b90 <main>
8048553:  50                   pushl  %eax
```

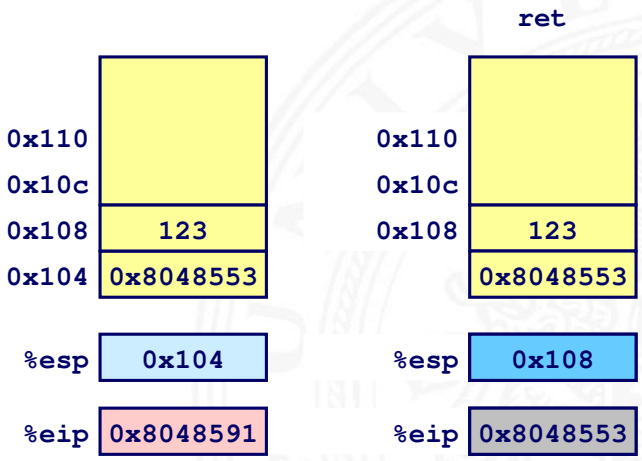


`%eip` **is program counter**

Beispiel: Prozeduraufruf (cont.)

- Prozedurrücksprung `ret`

```
8048591:  c3                   ret
```



`%eip` **is program counter**

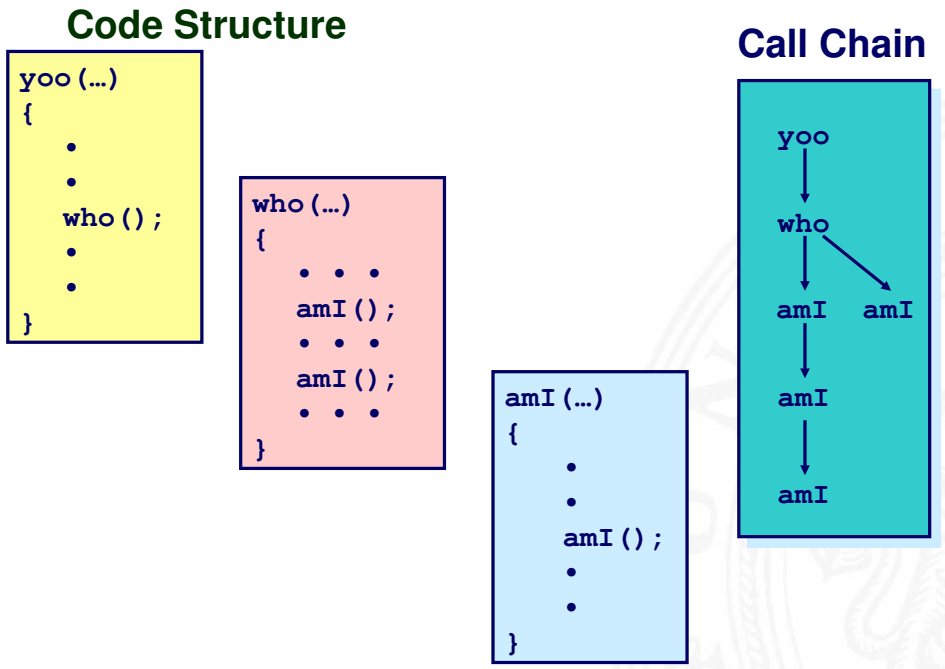
Stack-basierende Sprachen

- Sprachen, die Rekursion unterstützen
  - z.B.: C, Pascal, Java
  - Code muss „Reentrant“ sein
    - erlaubt mehrfache, simultane Instanziierungen einer Prozedur
  - Ort, um den Zustand jeder Instanziierung zu speichern
    - Argumente
    - lokale Variable
    - Rücksprungadresse
- Stack Verfahren
  - Zustandsspeicher für Aufrufe
    - zeitlich limitiert: von `call` bis `ret`
  - aufgerufenes Unterprogramm („Callee“) wird vor aufrufendem Programme („Caller“) beendet
- Stack „Frame“
  - Bereich/Zustand einer einzelnen Prozedur-Instanziierung

Stack-Frame

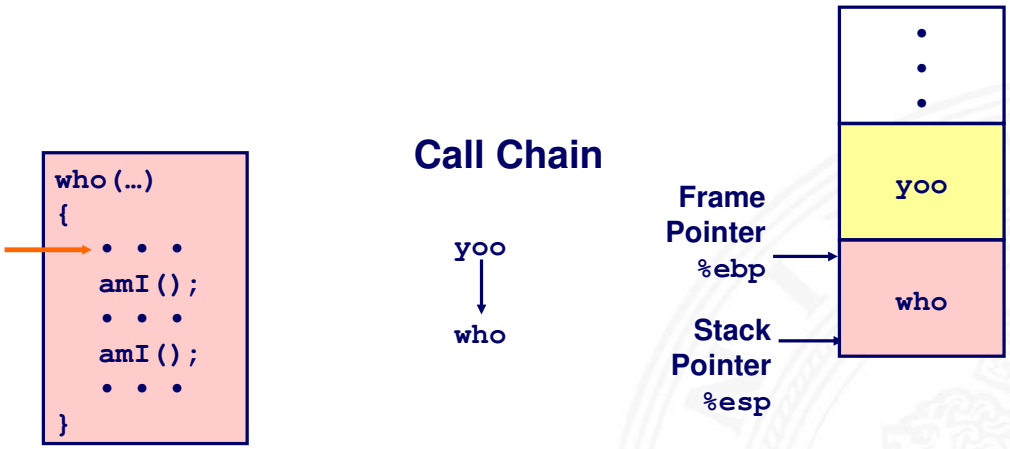
- Inhalt
  - Parameter
  - lokale Variablen
  - Rücksprungadresse
  - temporäre Daten
- Verwaltung
  - bei Aufruf wird Speicherbereich zugeteilt „Setup“ Code
  - bei Return –"– freigegeben „Finish“ Code
- Adressenverweise („Pointer“)
  - Stackpointer `%esp` gibt das obere Ende des Stacks an
  - Framepointer `%ebp` gibt den Anfang des aktuellen Frame an

Beispiel: Stack-Frame

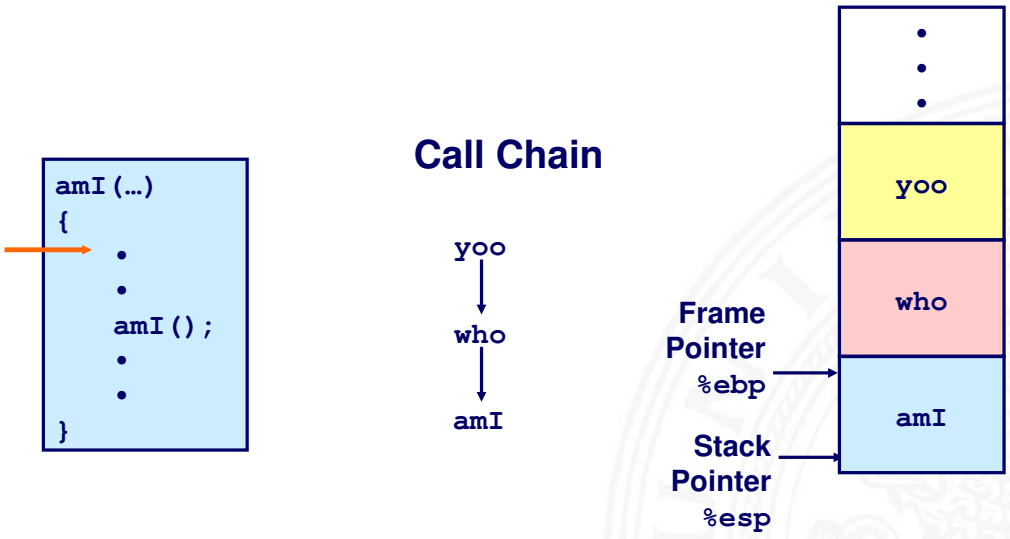


Beispiel: Stack-Frame (cont.)

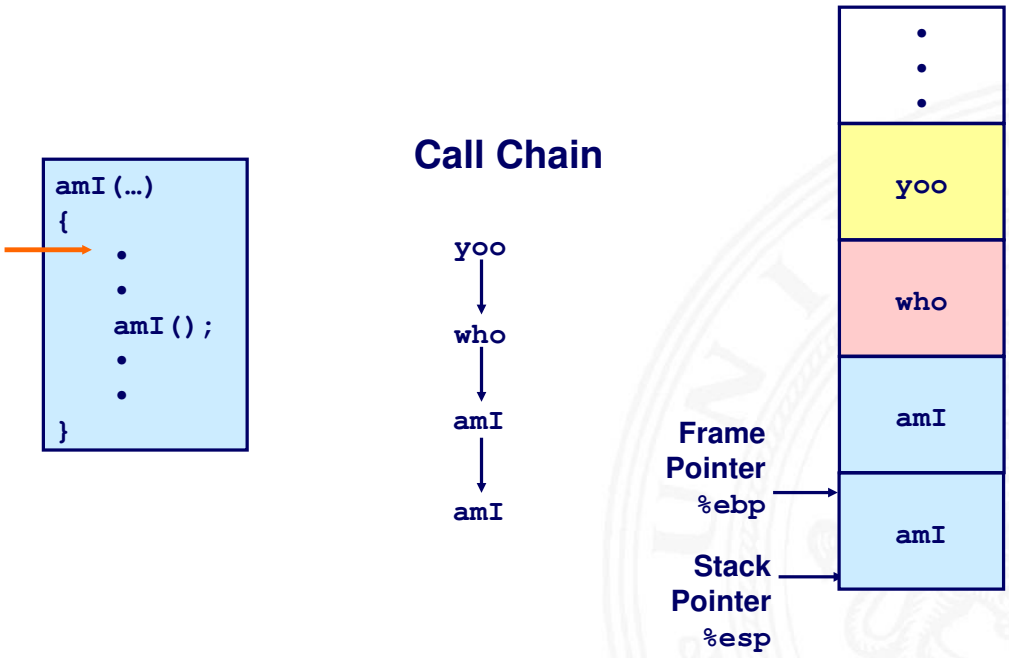
Beispiel: Stack-Frame (cont.)



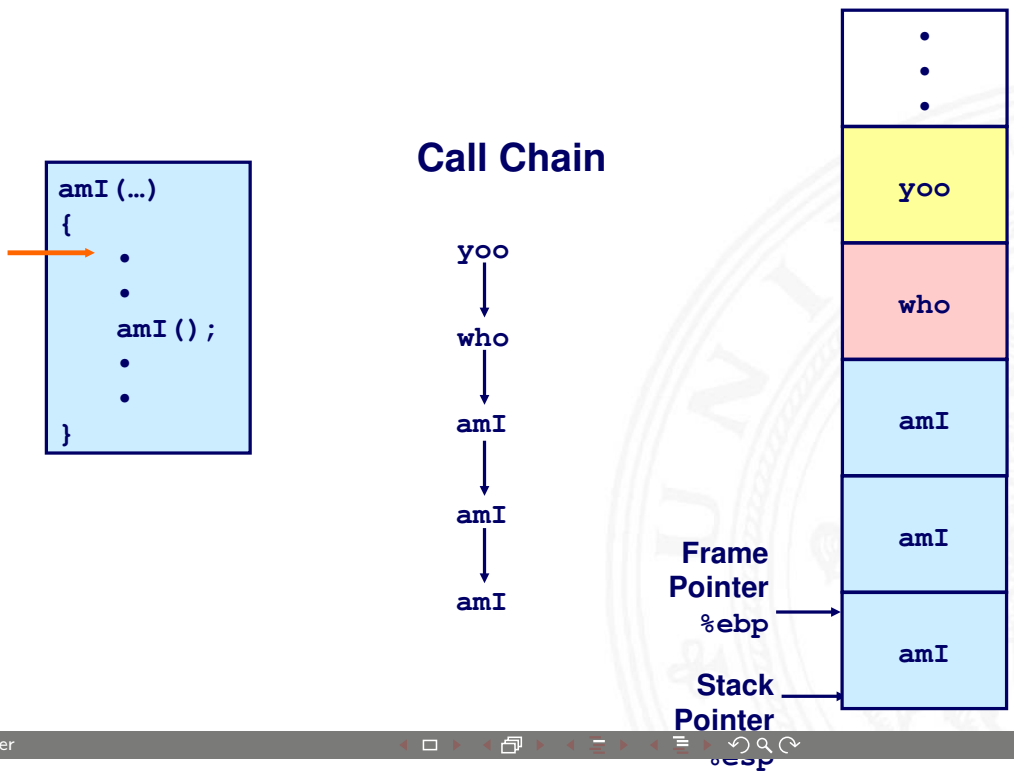
Beispiel: Stack-Frame (cont.)



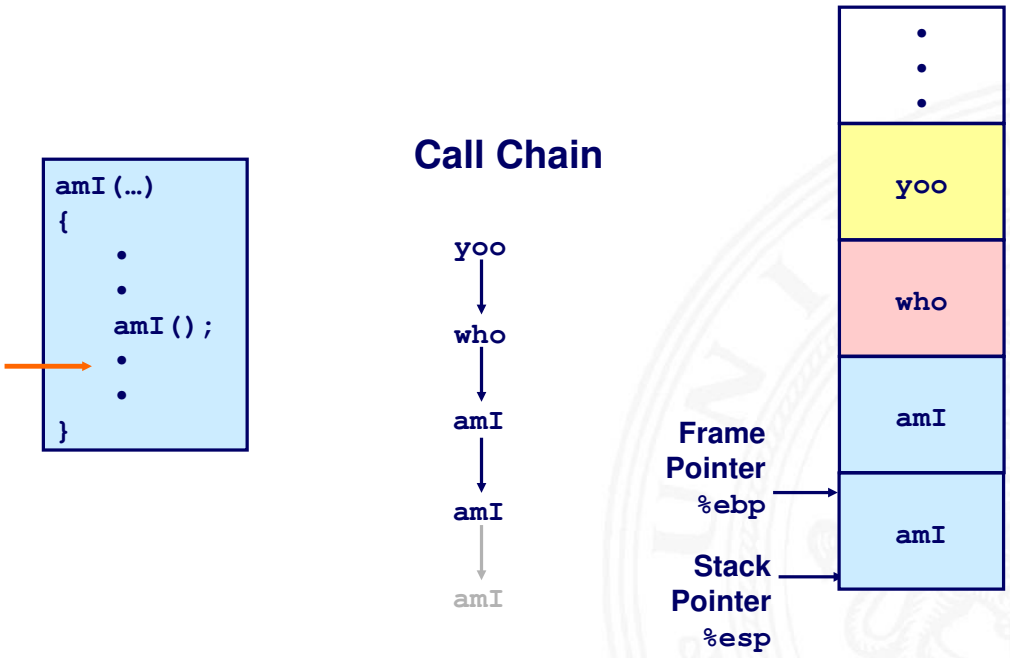
Beispiel: Stack-Frame (cont.)



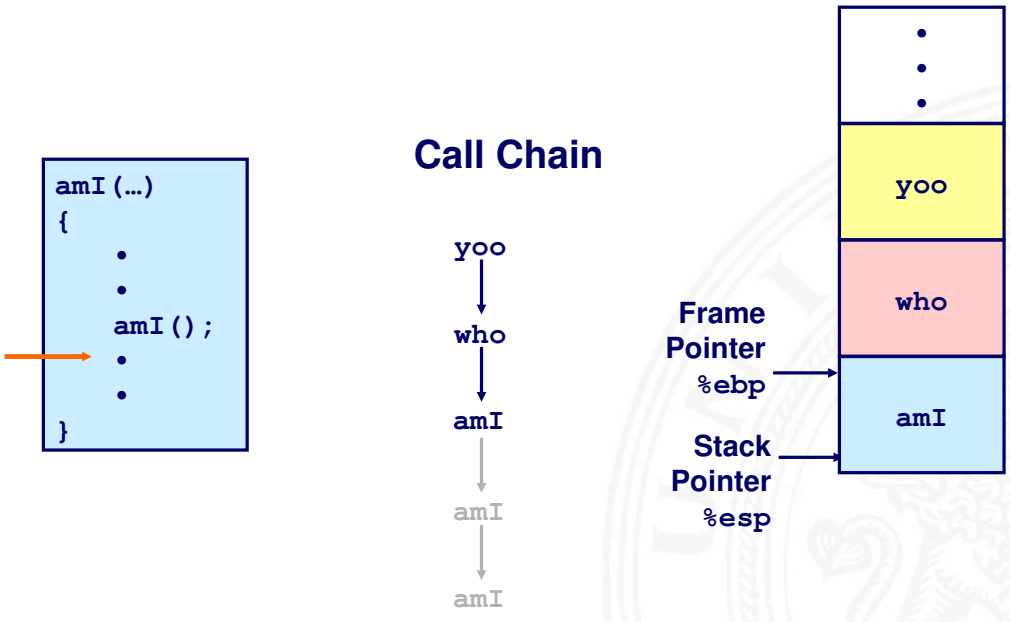
Beispiel: Stack-Frame (cont.)



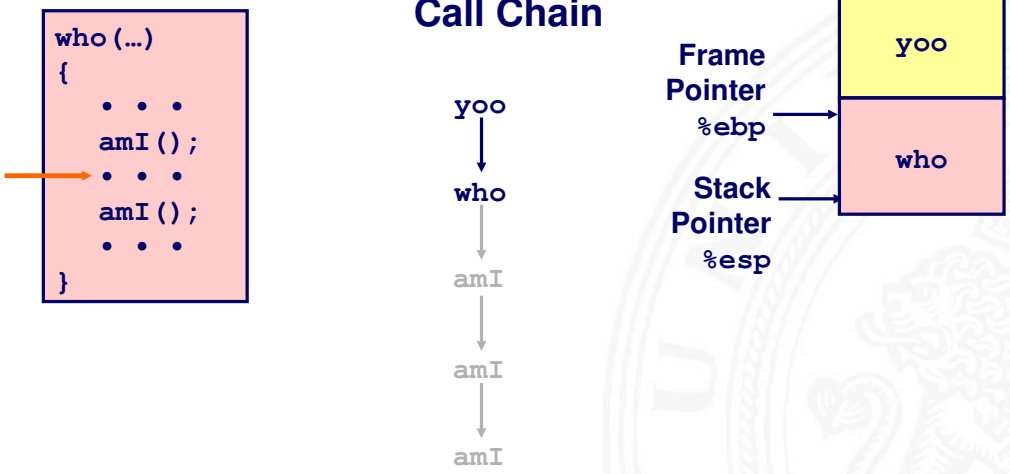
Beispiel: Stack-Frame (cont.)



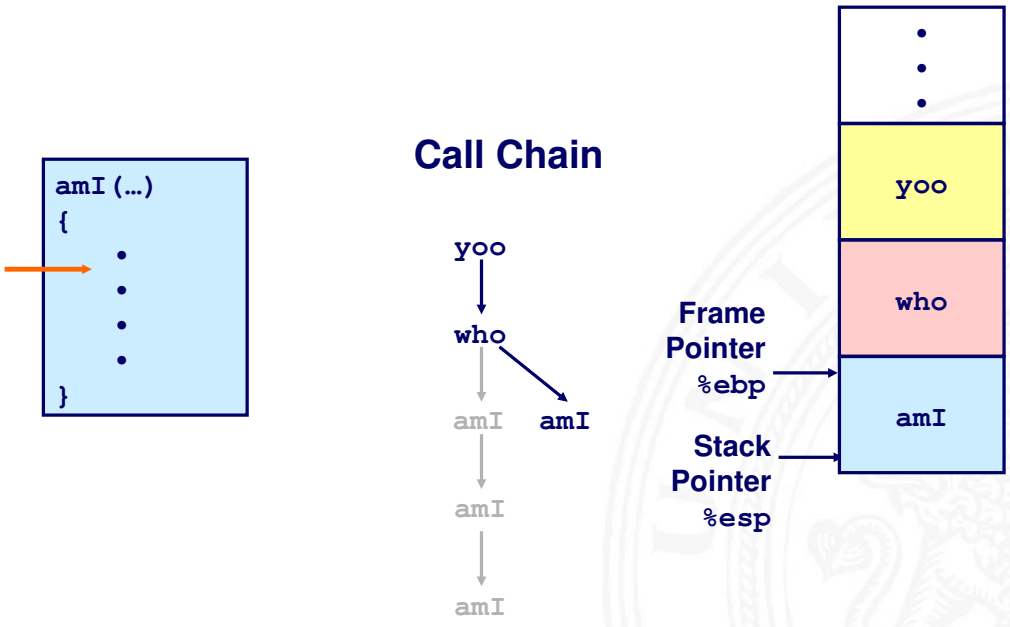
Beispiel: Stack-Frame (cont.)



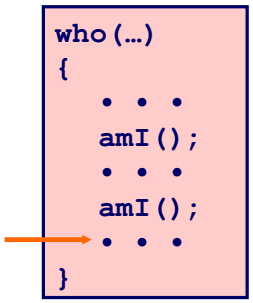
Beispiel: Stack-Frame (cont.)

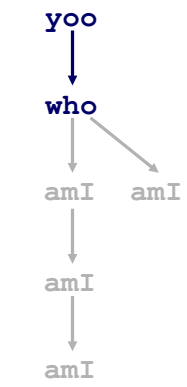


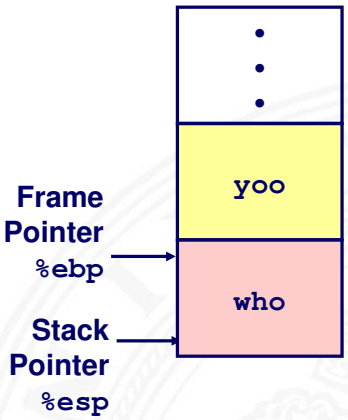
Beispiel: Stack-Frame (cont.)



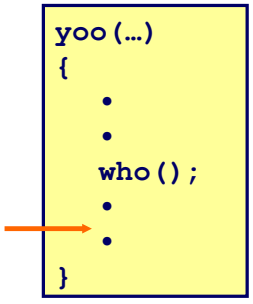
Beispiel: Stack-Frame (cont.)

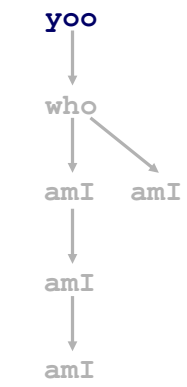


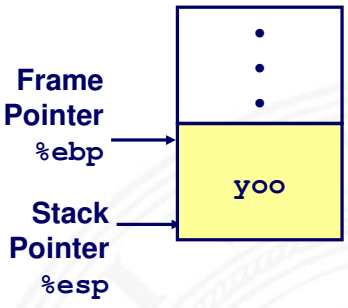




Beispiel: Stack-Frame (cont.)



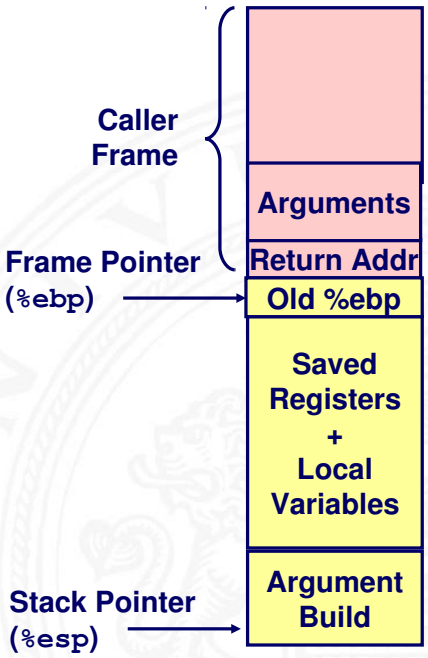




x86/Linux Stack-Frame

aktueller Stack-Frame

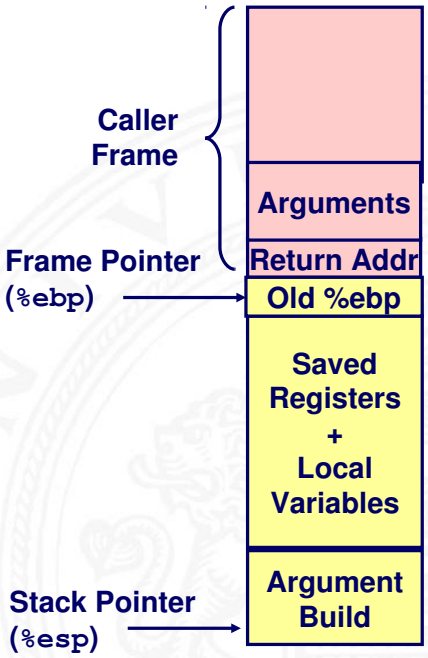
- von oben nach unten organisiert „Top“ . . . „Bottom“
- Parameter für weitere Funktion die aufgerufen wird `call`
- lokale Variablen
  - wenn sie nicht in Registern gehalten werden können
- gespeicherter Registerkontext
- Zeiger auf vorherigen Frame



x86/Linux Stack-Frame (cont.)

„Caller“ Stack-Frame

- Rücksprungadresse
  - von `call`-Anweisung erzeugt
- Argumente für aktuellen Aufruf



Register Sicherungskonventionen

- yoo („Caller“) ruft Prozedur `who` („Callee“) auf

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

- kann `who` Register für vorübergehende Speicherung benutzen?
  - Inhalt von `%edx` wird von `who` überschrieben

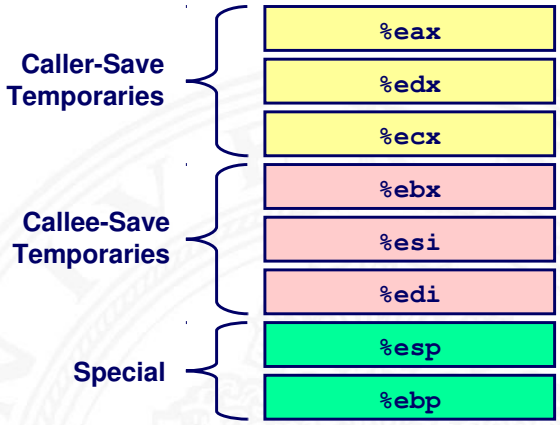
⇒ zwei mögliche Konventionen

- „Caller-Save“
  `yoo` speichert in seinen Frame vor Prozeduraufruf
- „Callee-Save“
  `who` speichert in seinen Frame vor Benutzung

x86/Linux Register Verwendung

Integer Register

- zwei werden speziell verwendet
  - `%ebp`, `%esp`
- „Callee-Save“ Register
  - `%ebx`, `%esi`, `%edi`
  - alte Werte werden vor Verwendung auf dem Stack gesichert
- „Caller-Save“ Register
  - `%eax`, `%edx`, `%ecx`
  - “Caller” sichert diese Register
- Register `%eax` speichert auch den zurückgelieferten Wert



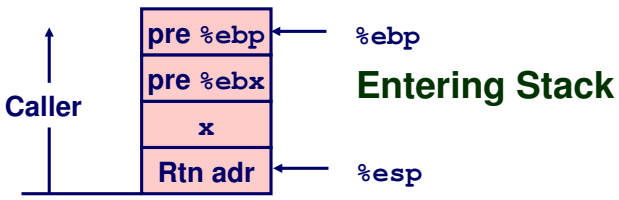
Beispiel: Rekursive Fakultät

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

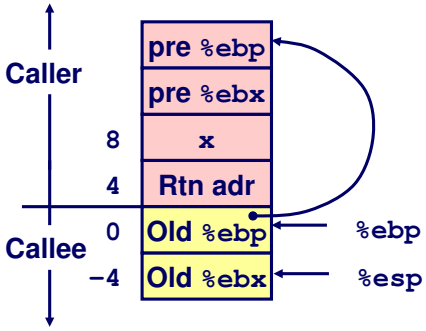
- %eax
  - benutzt ohne vorheriges Speichern
- %ebx
  - am Anfang speichern
  - am Ende zurückschreiben

```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```


Beispiel: `rfact` – Stack „Setup“




```
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```



Beispiel: `rfact` – Rekursiver Aufruf

Recursion

```
  movl 8(%ebp),%ebx   # ebx = x
  cmpl $1,%ebx        # Compare x : 1
  jle .L78            # If <= goto Term
  leal -1(%ebx),%eax  # eax = x-1
  pushl %eax          # Push x-1
  call rfact          # rfact(x-1)
  imull %ebx,%eax     # rval * x
  jmp .L79            # Goto done
.L78:               # Term:
  movl $1,%eax        # return val = 1
.L79:               # Done:
```

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1) ;
  return rval * x;
}
```

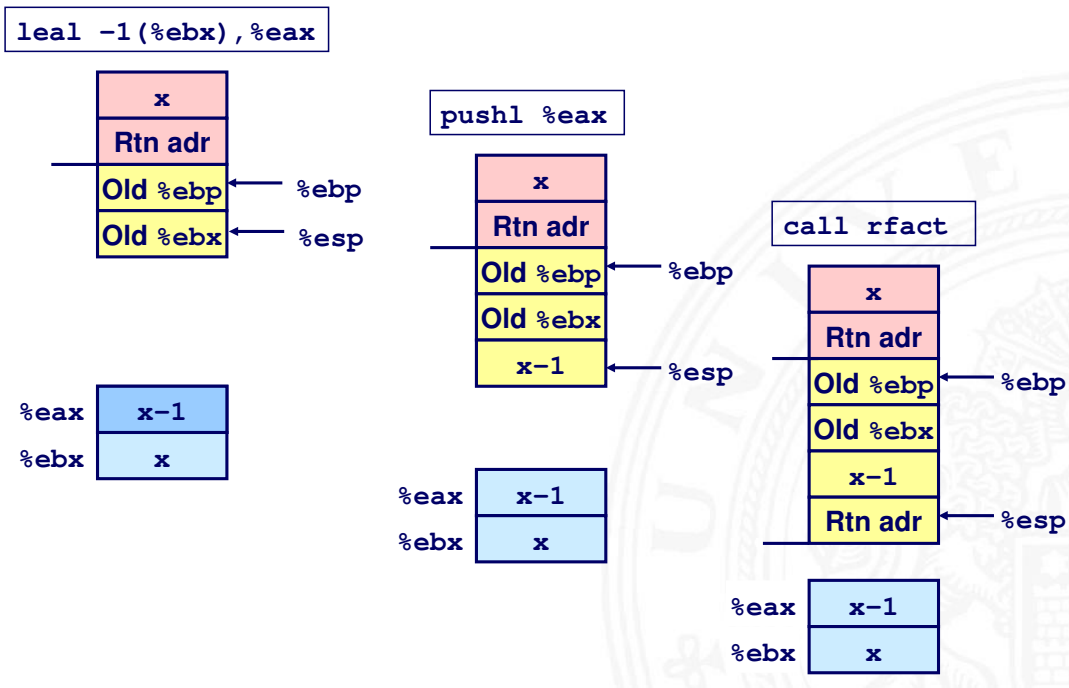
Registers

`%ebx` Stored value of x

`%eax`

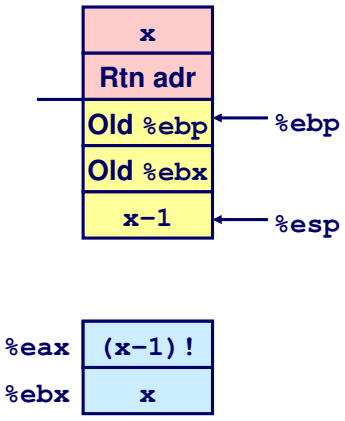
- Temporary value of `x-1`
- Returned value from `rfact(x-1)`
- Returned value from this call

Beispiel: `rfact` – Rekursion

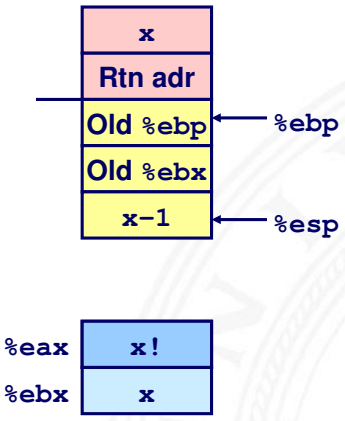


Beispiel: `rfact` – Ergebnisübergabe

Return from Call



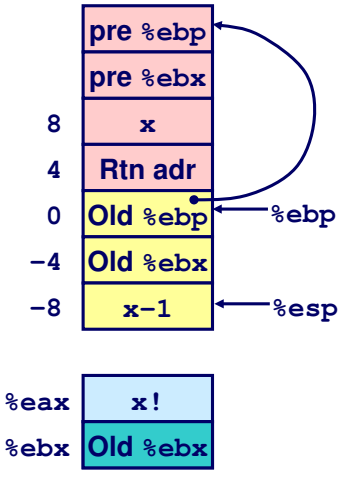

```
imull %ebx,%eax
```

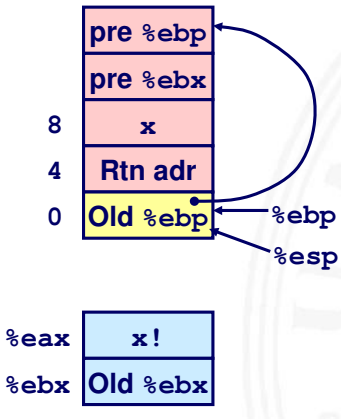


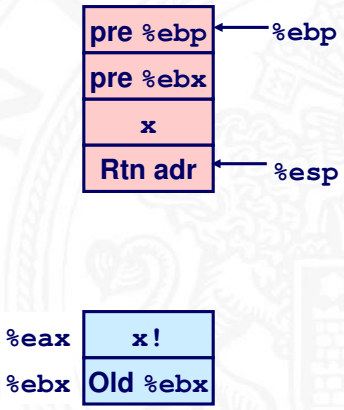
Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

Beispiel: `rfact` – Stack „Finish“

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```







Zeiger auf Adresse / *call by reference*

- Variable der aufrufenden Funktion soll modifiziert werden

⇒ Adressenverweis (*call by reference*)

- Beispiel: `sfact`

Recursive Procedure

```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

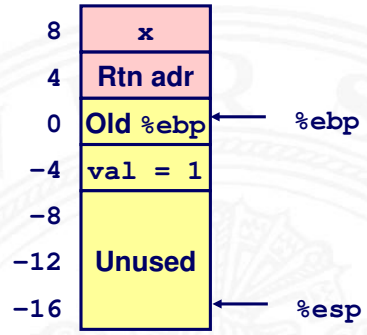
Top-Level Call

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```


Beispiel: `sfact`

Initial part of `sfact`

```
_sfact:
   pushl %ebp          # Save %ebp
   movl %esp,%ebp      # Set %ebp
   subl $16,%esp       # Add 16 bytes
   movl 8(%ebp),%edx   # edx = x
   movl $1,-4(%ebp)    # val = 1
```



- lokale Variable `val` auf Stack speichern
  - Pointer auf `val`
  - berechnen als `-4(%ebp)`
- Push `val` auf Stack
  - zweites Argument
  - `movl $1, -4(%ebp)`

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

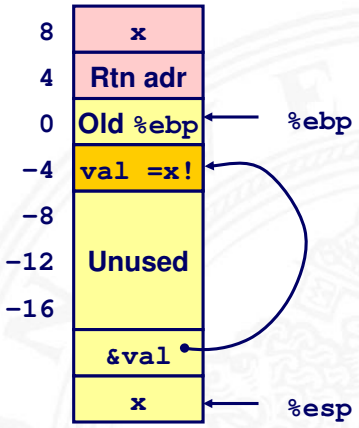
Beispiel: sfact – Pointerübergabe bei Aufruf

Calling s_helper from sfact

```
leal -4(%ebp),%eax # Compute &val
pushl %eax         # Push on stack
pushl %edx         # Push x
call s_helper      # call
movl -4(%ebp),%eax # Return val
• • •              # Finish
```

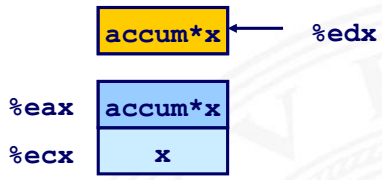
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Stack at time of call



Beispiel: sfact – Benutzung des Pointers

```
void s_helper
  (int x, int *accum)
{
  • • •
    int z = *accum * x;
    *accum = z;
  • • •
}
```




```
• • •
movl %ecx,%eax    # z = x
imull (%edx),%eax # z *= *accum
movl %eax,(%edx)  # *accum = z
• • •
```

- Register %ecx speichert x
- Register %edx mit Zeiger auf accum

Zusammenfassung: Stack

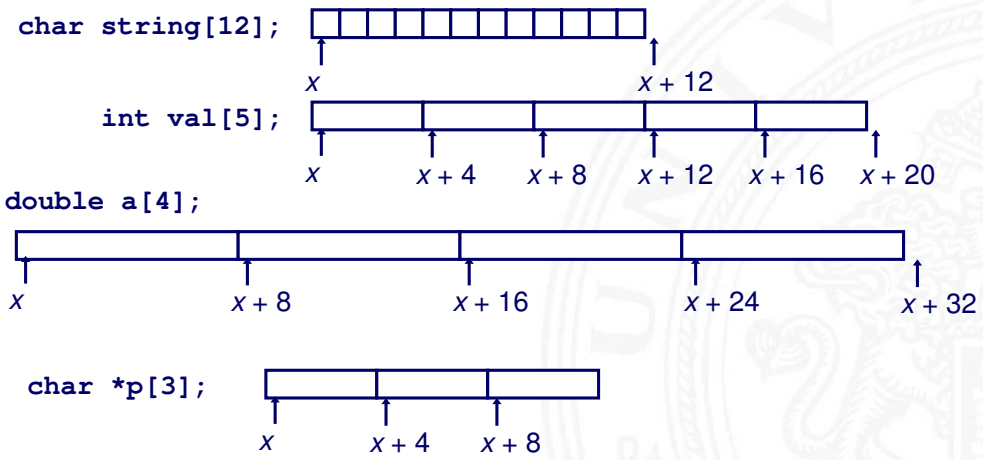
- Stack ermöglicht Rekursion
  - lokaler Speicher für jede Prozedur(aufruf) Instanz
    - Instanziierungen beeinflussen sich nicht
    - Adressierung lokaler Variablen und Argumente kann relativ zu Stackposition (Framepointer) sein
  - grundlegendes (Stack-) Verfahren
    - Prozeduren terminieren in umgekehrter Reihenfolge der Aufrufe
- x86 Prozeduren sind Kombination von Anweisungen + Konventionen
  - `call` / `ret` Anweisungen
  - Konventionen zur Registerverwendung
    - „Caller-Save“ / „Callee-Save“
    - `%ebp` und `%esp`
  - festgelegte Organisation des Stack-Frame

Grundlegende Datentypen

- Ganzzahl (Integer)
  - wird in allgemeinen Registern gespeichert
  - abhängig von den Anweisungen: *signed*/*unsigned*
Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
- Gleitkomma (Floating Point)
  - wird in Gleitkomma-Registern gespeichert
Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12	long double

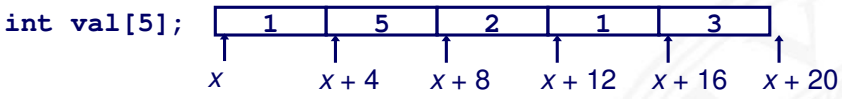
Array: Allokation / Speicherung

- `T A[N];`
  - Array `A` mit Daten von Typ `T` und `N` Elementen
  - fortlaufender Speicherbereich von `N`×sizeof(`T`) Bytes



Array: Zugriffskonvention

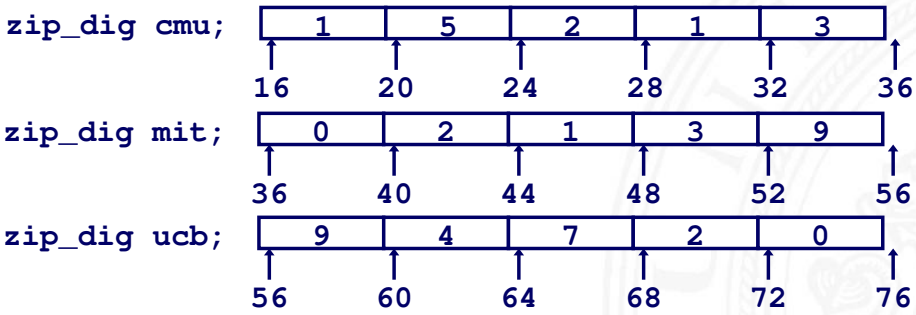
- `T A[N];`
  - Array `A` mit Daten von Typ `T` und `N` Elementen
  - Bezeichner `A` zeigt auf erstes Element des Arrays: Element 0



Reference	Type	Value
`val[4]`	`int`	`3`
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	`??`
`*(val+1)`	`int`	`5`
`val + i`	`int *`	*x* + 4 *i*

Beispiel: einfacher Arrayzugriff

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



Beispiel: einfacher Arrayzugriff (cont.)

- Register `%edx` : Array Startadresse
  `%eax` : Array Index
- Adressieren von 4×`%eax`+`%edx`

⇒ Speicheradresse `(%edx,%eax,4)`

```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

Memory Reference Code

```
# %edx = z
# %eax = dig
 movl (%edx,%eax,4),%eax # z[dig]
```

- keine Bereichsüberprüfung („*bounds checking*“)
- Verhalten außerhalb des Indexbereichs ist Implementierungsabhängig

Beispiel: Arrayzugriff mit Schleife

- Originalcode

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- transformierte Version: gcc
  - Laufvariable `i` eliminiert
  - aus Array-Code wird Pointer-Code
  - in „do-while“ Form
  - Test bei Schleifeneintritt unnötig

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```


Beispiel: Arrayzugriff mit Schleife (cont.)

- Register `%ecx` : z
  `%edx` : zi
  `%eax` : zend
- `*z + 2*(zi+4*zi)` ersetzt `10*zi + *z`
- z++ Inkrement: +4

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax          # zi = 0
  leal 16(%ecx),%ebx      # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx # 5*zi
  movl (%ecx),%eax        # *z
  addl $4,%ecx            # z++
  leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx          # z : zend
  jle .L59                # if <= goto loop
```

Strukturen

- Allokation eines zusammenhängenden Speicherbereichs
- Elemente der Struktur über Bezeichner referenziert
- verschiedene Typen der Elemente sind möglich

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout

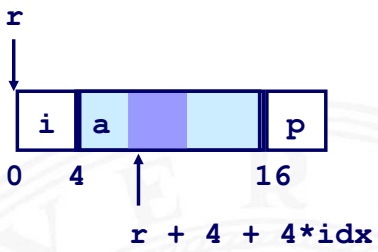

```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```


Strukturen: Zugriffskonventionen

- Zeiger auf Byte-Array für Zugriff auf Struktur(element) `r`
- Compiler bestimmt Offset für jedes Element



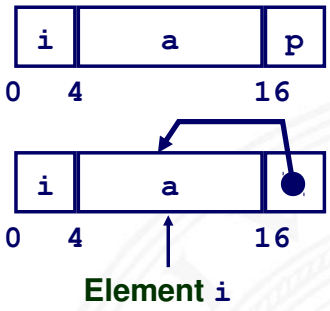

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax   # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

Beispiel: Strukturreferenzierung

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```




```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```

```
# %edx = r
movl (%edx),%ecx        # r->i
leal 0(,%ecx,4),%eax    # 4*(r->i)
leal 4(%edx,%eax),%eax  # r+4+4*(r->i)
movl %eax,16(%edx)      # Update r->p
```


Ausrichtung der Datenstrukturen (*Alignment*)

- Datenstrukturen an Wortgrenzen ausrichten
  double- / quad-word
- sonst Problem
  - ineffizienter Zugriff über Wortgrenzen hinweg
  - virtueller Speicher und Caching

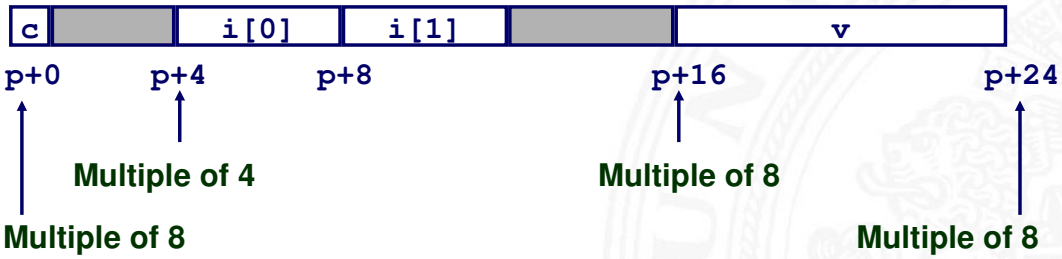
⇒ Compiler erzeugt „Lücken“ zur richtigen Ausrichtung

- typisches Alignment (IA32)

Länge	Typ		Windows	Linux
1 Byte	`char`	keine speziellen Verfahren		
2 Byte	`short`	Adressbits:	...0	...0
4 Byte	`int, float, char *`	–"–	...00	...00
8 Byte	`double`	–"–	...000	...00
12 Byte	`long double`	–"–	–	...00

Beispiel: Structure Alignment

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Zusammenfassung: Datentypen

- Arrays
  - fortlaufend zugeteilter Speicher
  - Adressverweis auf das erste Element
  - keine Bereichsüberprüfung (*Bounds Checking*)
- Compileroptimierungen
  - Compiler wandelt Array-Code in Pointer-Code um
  - verwendet Adressierungsmodi um Arrayindizes zu skalieren
  - viele Tricks, um die Array-Indizierung in Schleifen zu verbessern
- Strukturen
  - Bytes werden in der ausgewiesenen Reihenfolge zugeteilt
  - ggf. Leerbytes, um die richtige Ausrichtung zu erreichen

Gliederung

Gliederung (cont.)

Bewertung der ISA

Kriterien für einen *guten* Befehlssatz

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

Statistiken zeigen: Dominanz der einfachen Instruktionen

Bewertung der ISA (cont.)

- x86-Prozessor

	Anweisung	Ausführungshäufigkeit %
1.	load	22 %
2.	conditional branch	20 %
3.	compare	16 %
4.	store	12 %
5.	add	8 %
6.	and	6 %
7.	sub	5 %
8.	move reg-reg	4 %
9.	call	1 %
10.	return	1 %
Total		96 %

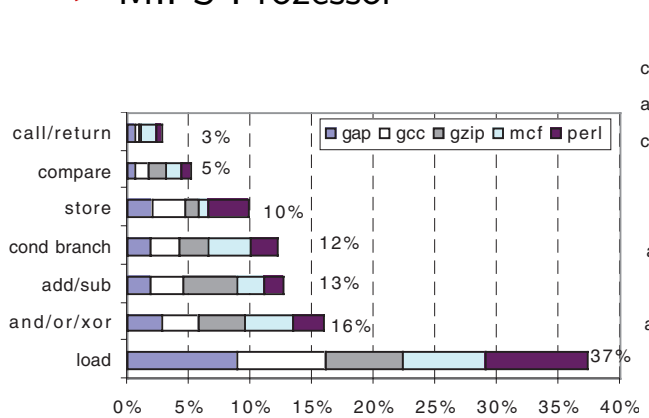
Bewertung der ISA (cont.)

Instruction	compress	eqntott	espresso	gcc (cc1)	li	Int. average
load	20.8%	18.5%	21.9%	24.9%	23.3%	22%
store	13.8%	3.2%	8.3%	16.6%	18.7%	12%
add	10.3%	8.8%	8.15%	7.6%	6.1%	8%
sub	7.0%	10.6%	3.5%	2.9%	3.6%	5%
mul				0.1%		0%
div						0%
compare	8.2%	27.7%	15.3%	13.5%	7.7%	16%
mov reg-reg	7.9%	0.6%	5.0%	4.2%	7.8%	4%
load imm	0.5%	0.2%	0.6%	0.4%		0%
cond. branch	15.5%	28.6%	18.9%	17.4%	15.4%	20%
uncond. branch	1.2%	0.2%	0.9%	2.2%	2.2%	1%
call	0.5%	0.4%	0.7%	1.5%	3.2%	1%
return, jmp indirect	0.5%	0.4%	0.7%	1.5%	3.2%	1%
shift	3.8%		2.5%	1.7%		1%
and	8.4%	1.0%	8.7%	4.5%	8.4%	6%
or	0.6%		2.7%	0.4%	0.4%	1%
other (xor, not, . . .)	0.9%		2.2%	0.1%		1%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
compare FP						0%
mov reg-reg FP						0%
other (abs, sqrt, . . .)						0%

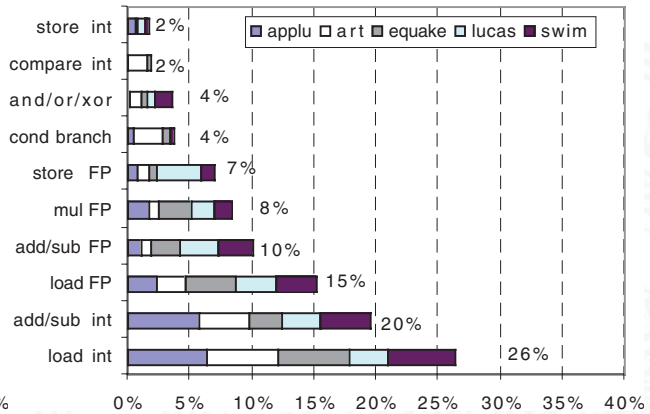
Figure D.15 80x86 instruction mix for five SPECint92 programs.

Bewertung der ISA (cont.)

- MIPS-Prozessor



SPECint2000 (96%)



SPECfp2000 (97%)

Bewertung der ISA (cont.)

- ca. 80 % der Berechnungen eines typischen Programms verwenden nur ca. 20 % der Instruktionen einer CPU
- am häufigsten gebrauchten Instruktionen sind einfache Instruktionen: load, store, add...

$\Rightarrow$ Motivation für RISC

CISC – Befehlssätze

Complex **I**nstruction **S**et **C**omputer

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

CISC Befehlssätze

- Instruktionssätze mit mehreren hundert Befehlen (> 300)
- sehr viele Adressierungsarten, -Kombinationen
- verschiedene, unterschiedlich lange Instruktionsformate
- fast alle Befehle können auf Speicher zugreifen
  - mehrere Schreib- und Lesezugriffe pro Befehl
  - komplexe Adressberechnung

CISC – Befehlssätze (cont.)

- Stack-orientierter Befehlssatz
  - Übergabe von Argumenten
  - Speichern des Programmzählers
  - explizite „Push“ und „Pop“ Anweisungen
- Zustandscodes („Flags“)
  - gesetzt durch arithmetische und logische Anweisungen

Konsequenzen

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

CISC – Mikroprogrammierung

- ein Befehl kann nicht in einem Takt abgearbeitet werden

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

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

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

CISC – Mikroprogrammierung (cont.)

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

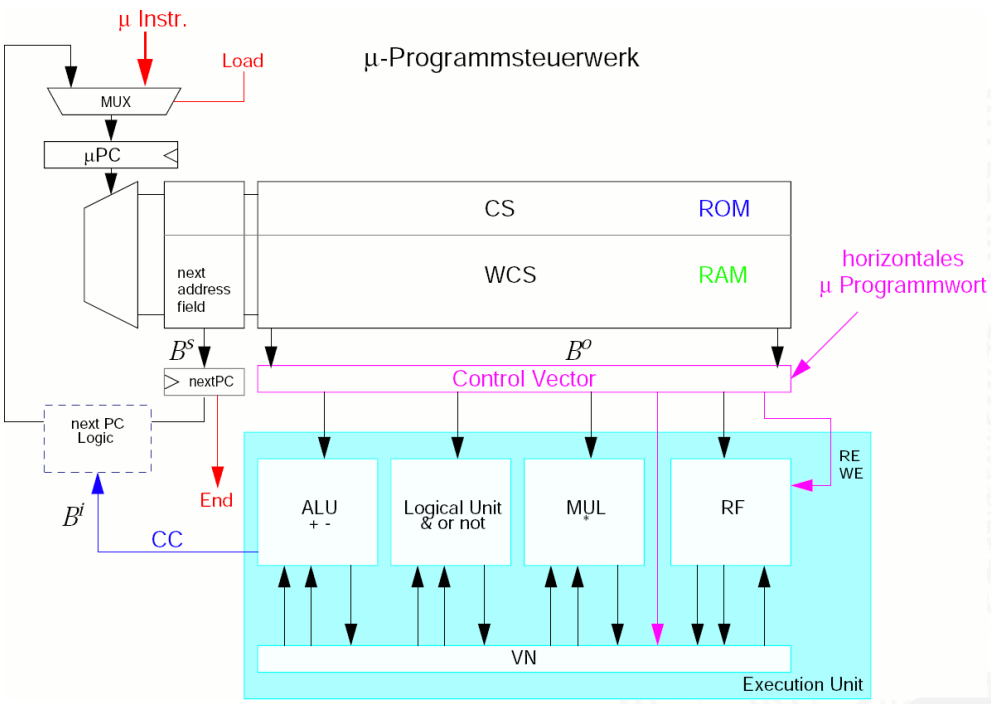
\+ CISC-Befehlssatz mit wenigen Mikrobefehlen realisieren

\+ bei RAM: Mikrobefehlssatz austauschbar

– (mehrstufige) ROM/RAM Zugriffe: zeitaufwändig

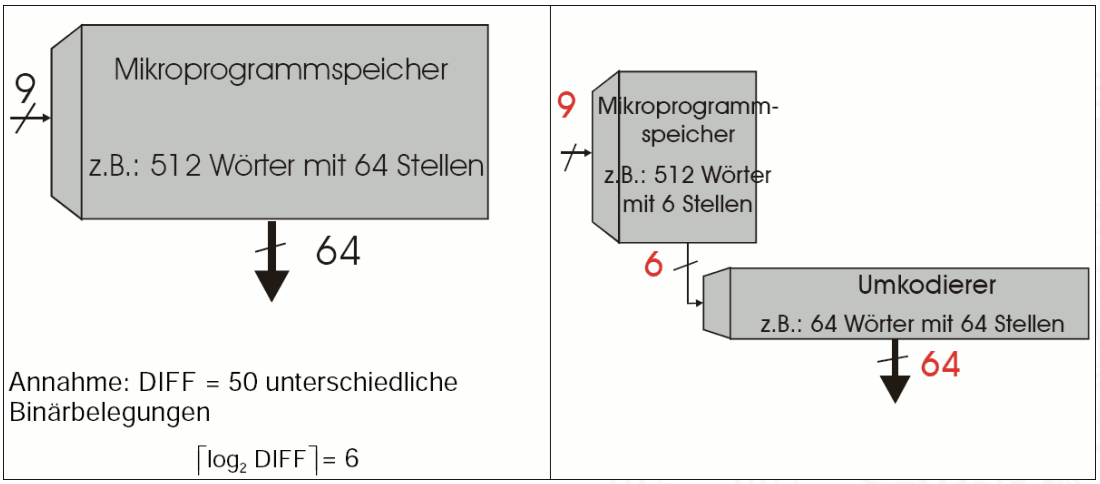
▶ ▸ horizontale Mikroprog. ▸ vertikale Mikroprog.

horizontale Mikroprogrammierung



◂ Mikroprogrammierung

vertikale Mikroprogrammierung



◂ Mikroprogrammierung

RISC – Befehlssätze

Reduced **I**nstruction **S**et **C**omputer

- Grundidee: Komplexitätsreduktion in der CPU
- internes Projekt bei IBM, seit den 80er Jahren: „RISC-Boom“
  - von Hennessy (Stanford) und Patterson (Berkeley) publiziert
- Hochsprachen und optimierende Compiler

⇒ kein Bedarf mehr für mächtige Assemblerbefehle

⇒ pro Assemblerbefehl muss nicht mehr „möglichst viel“ lokal in der CPU gerechnet werden (CISC Mikroprogramm)

RISC – Befehlssätze (cont.)

RISC Befehlssätze

- reduzierte Anzahl einfacher Instruktionen (z.B. 128)
  - benötigen in der Regel mehr Anweisungen für eine Aufgabe
  - werden aber mit kleiner, schneller Hardware ausgeführt
- Register-orientierter Befehlssatz
  - viele Register (üblicherweise ≥ 32)
  - Register für Argumente, „Return“-Adressen, Zwischenergebnisse
- Speicherzugriff *nur* durch „Load“ und „Store“ Anweisungen
- alle anderen Operationen arbeiten auf Registern
- keine Zustandscodes (Flag-Register)
  - Testanweisungen speichern Resultat direkt im Register

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
20.1 Computerarchitektur - Befehlssätze / ISA
64-040 Rechnerstrukturen

RISC – Befehlssätze (cont.)

Konsequenzen

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

- optimierende Compiler nötig / möglich
- High-performance Speicherhierarchie notwendig

CISC vs. RISC

ursprüngliche Debatte

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

aktueller Stand

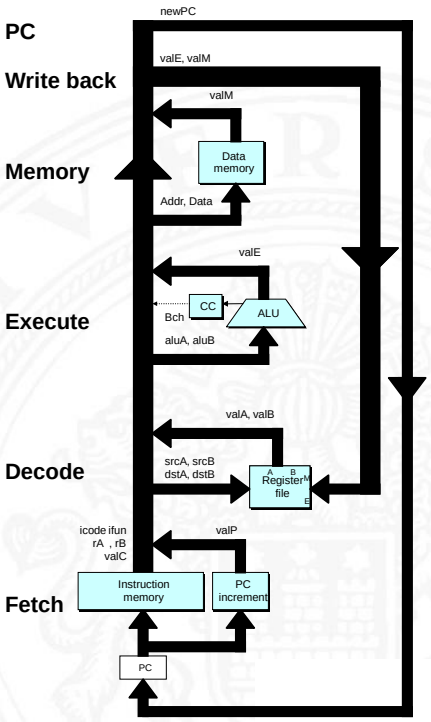
- Grenzen verwischen
  - RISC-Prozessoren werden komplexer
  - CISC-Prozessoren weisen RISC-Konzepte oder gar RISC-Kern auf
- für Desktop Prozessoren ist die Wahl der ISA kein Thema
  - Code-Kompatibilität ist sehr wichtig!
  - mit genügend Hardware wird alles schnell ausgeführt
- eingebettete Prozessoren: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

ISA Design heute

- Restriktionen durch Hardware abgeschwächt
- Code-Kompatibilität leichter zu erfüllen
  - Emulation in Firm- und Hardware
- Intel bewegt sich weg von IA-32
  - erlaubt nicht genug Parallelität
- hat IA-64 eingeführt („Intel Architecture 64-bit“)
  ⇒ neuer Befehlssatz mit expliziter Parallelität (EPIC)
  ⇒ 64-bit Wortgrößen (überwinden Adressraumlimits)
  ⇒ benötigt hoch entwickelte Compiler

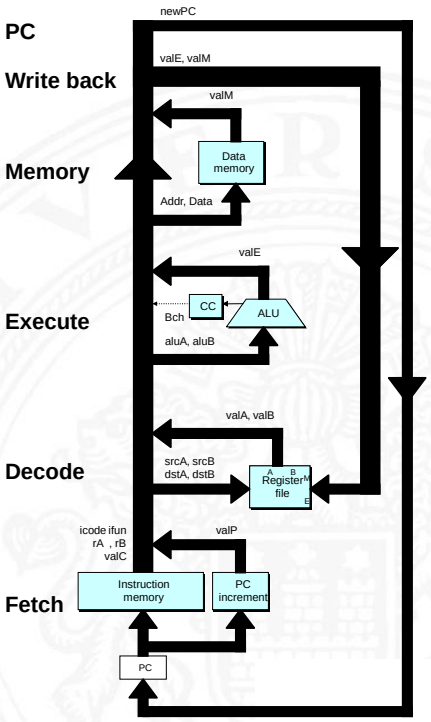
Sequenzielle Hardwarestruktur

- interner Zustand
  - Programmzähler Register PC
  - Zustandscode Register CC
  - Registerbank
  - Speicher
    - gemeinsamer Speicher für Daten und Anweisungen
- von-Neumann Abarbeitung
  - Befehl aus Speicher laden
    PC enthält Adresse
  - Verarbeitung durch die Stufen
  - Programmzähler aktualisieren

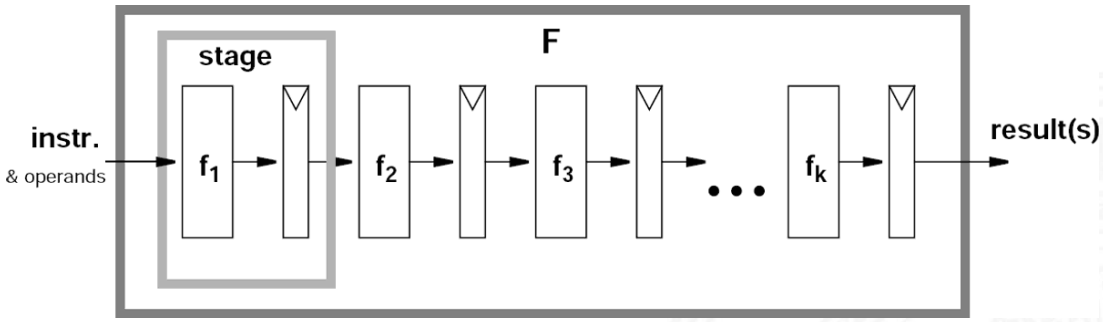


Sequenzielle Befehlsabarbeitung

- Befehl holen „Fetch“
  - Anweisung aus Speicher lesen
- Befehl decodieren „Decode“
  - Befehlsregister interpretieren
  - Operanden holen
- Befehl ausführen „Execute“
  - berechne Wert oder Adresse
- Speicherzugriff „Memory“
  - Daten lesen oder schreiben
- Registerzugriff „Write Back“
  - in Registerbank schreiben
- Programmzähler aktualisieren
  - inkrementieren –oder–
  - Speicher-/Registerinhalt bei Sprung



Pipelining / Fließbandverarbeitung



Grundidee

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

Pipelining / Fließbandverarbeitung (cont.)

Pipelining-Konzept

- Prozess in unabhängige Abschnitte aufteilen
- Objekt sequenziell durch diese Abschnitte laufen lassen
- zu jedem gegebenen Zeitpunkt werden zahlreiche Objekte bearbeitet

Konsequenz

- lässt Vorgänge gleichzeitig ablaufen
- „Real-World Pipelines“: Autowaschanlagen

Pipelining / Fließbandverarbeitung (cont.)

Arithmetische Pipelines

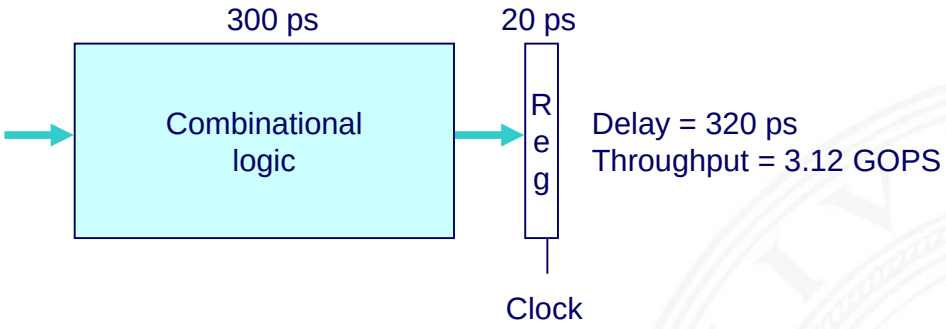
- Idee: lange Berechnung in Teilschritte zerlegen
  wichtig bei komplizierteren arithmetischen Operationen
  - die sonst sehr lange dauern (weil ein großes Schaltnetz)
  - die als Schaltnetz extrem viel Hardwareaufwand erfordern
  - Beispiele: Multiplikation, Division, Fließkommaoperationen...

\+ Erhöhung des Durchsatzes, wenn Berechnung mehrfach hintereinander ausgeführt wird

(RISC) Prozessorpipelines

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

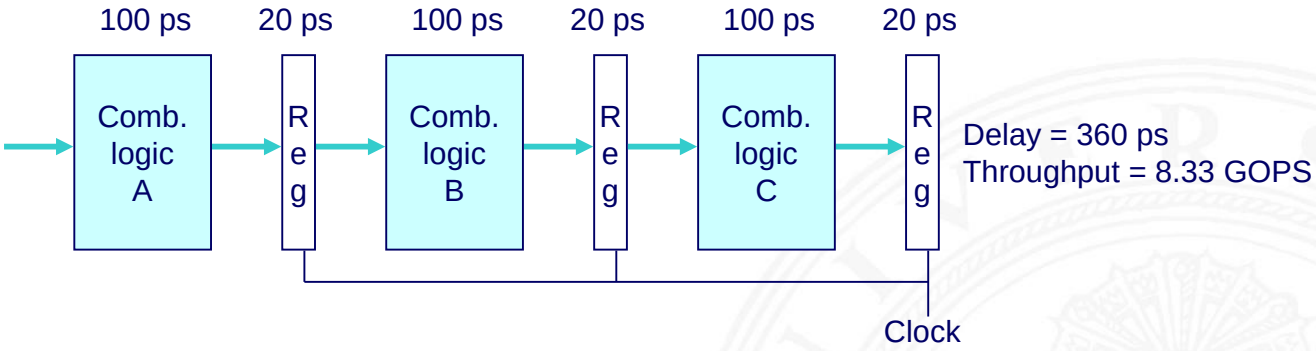
Berechnungsbeispiel: ohne Pipeline



System

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

Berechnungsbeispiel: Version mit 3-stufiger Pipeline

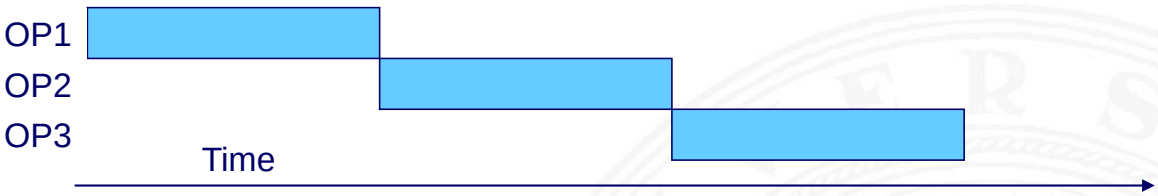


System

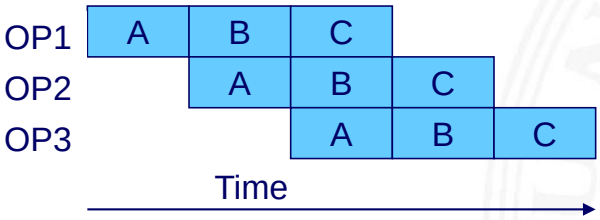
- Kombinatorische Logik in 3 Blöcke zu je 100 ps aufgeteilt
- neue Operation, sobald vorheriger Abschnitt durchlaufen wurde
  $\Rightarrow$ alle 120 ps neue Operation
- allgemeine Latenzzunahme
  $\Rightarrow$ 360 ps von Start bis Ende

Funktionsweise der Pipeline

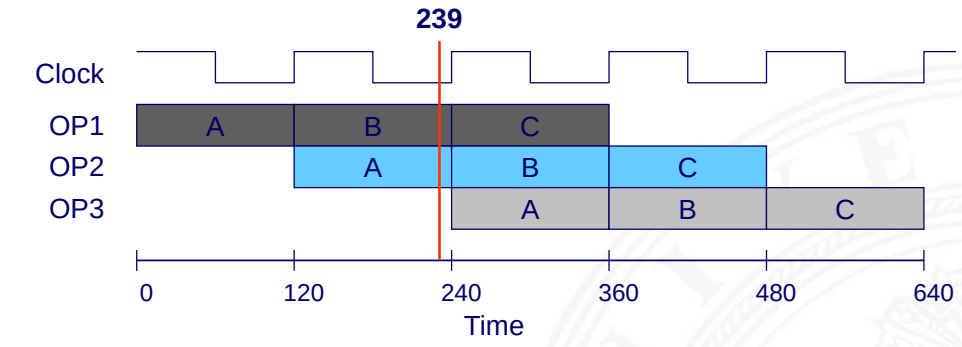
- ohne Pipeline

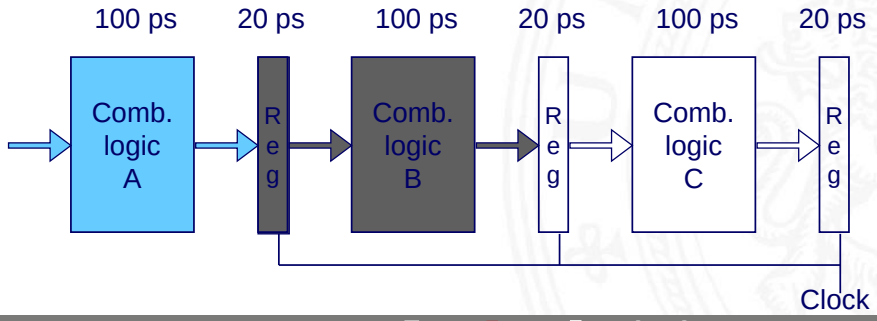


- 3-stufige Pipeline



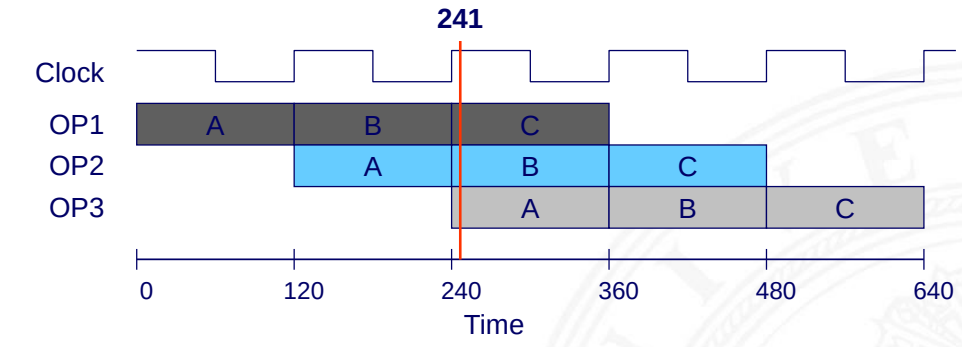
Beispiel: 3-stufige Pipeline

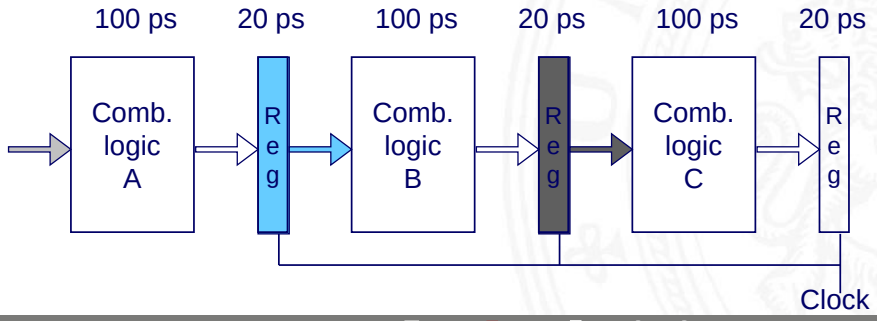




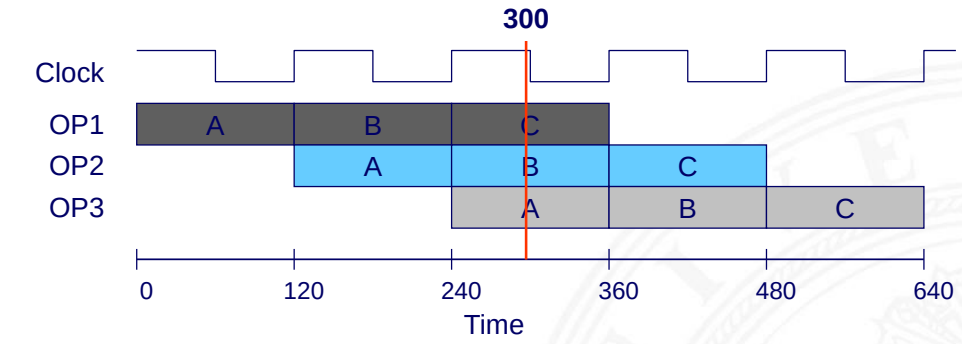
Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
20.3 Computerarchitektur - Pipelining
64-040 Rechnerstrukturen

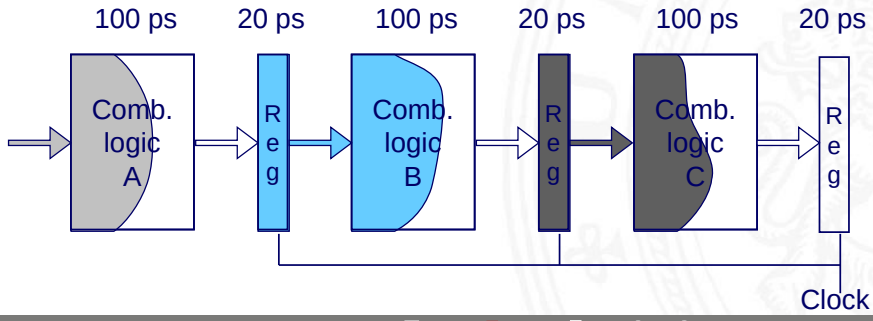
Beispiel: 3-stufige Pipeline



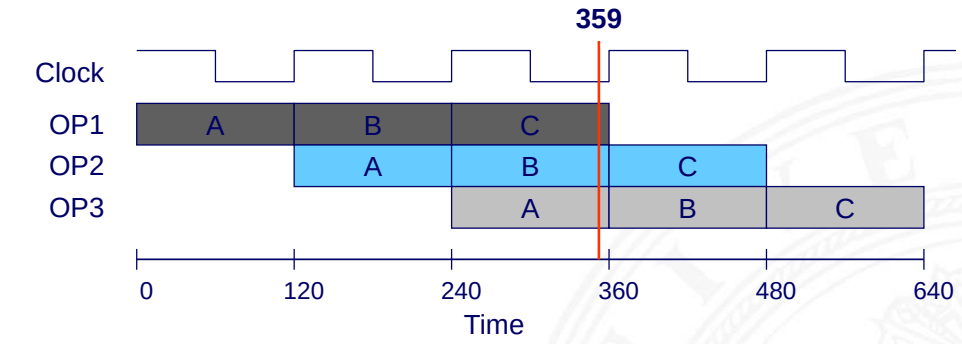


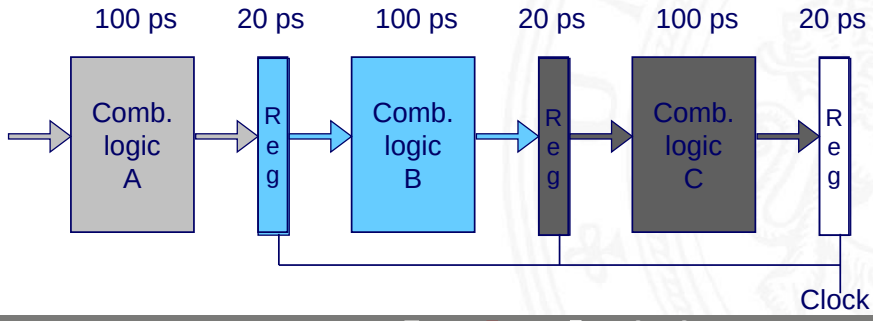
Beispiel: 3-stufige Pipeline



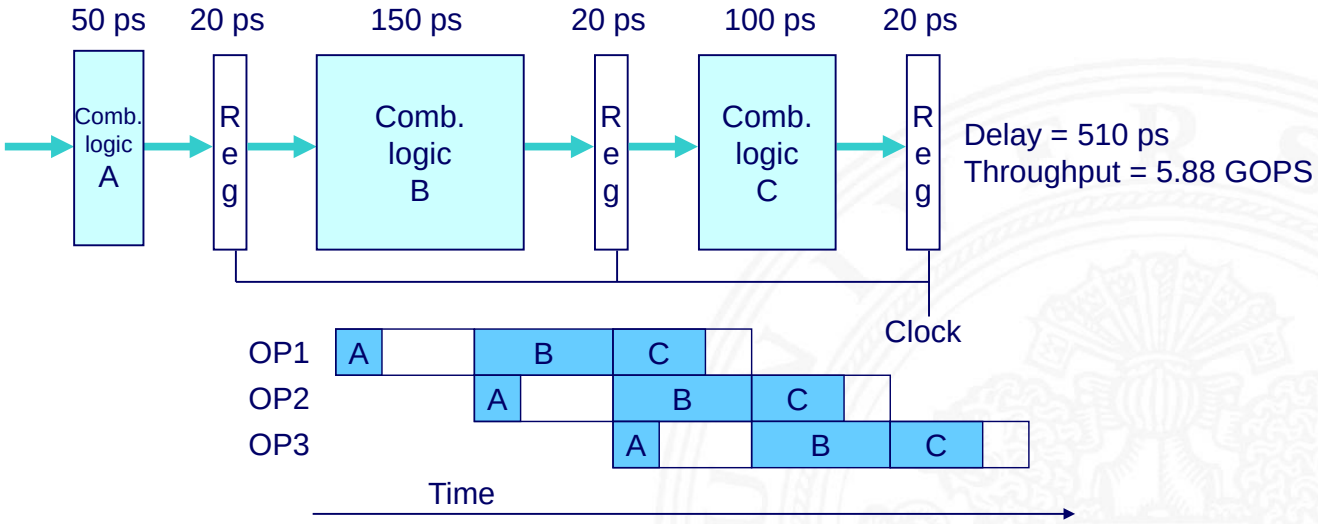


Beispiel: 3-stufige Pipeline



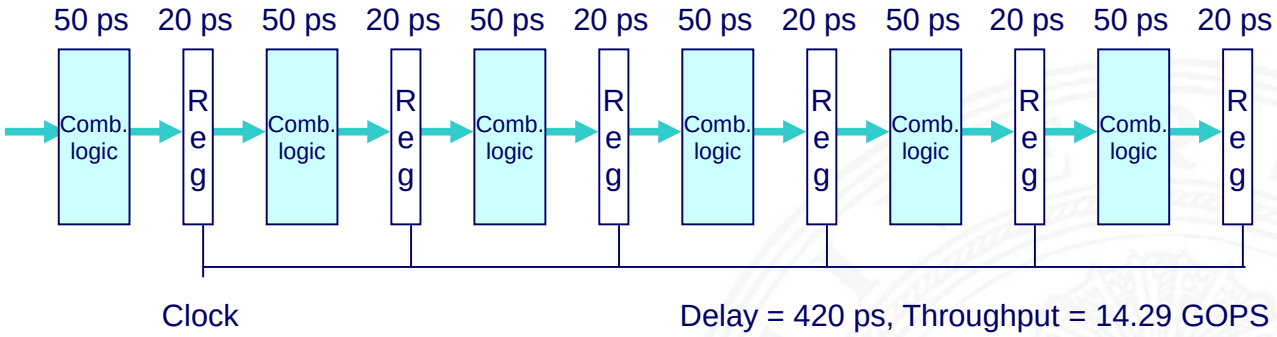


Probleme: nicht uniforme Verzögerungen



- größte Verzögerung bestimmt Taktfrequenz

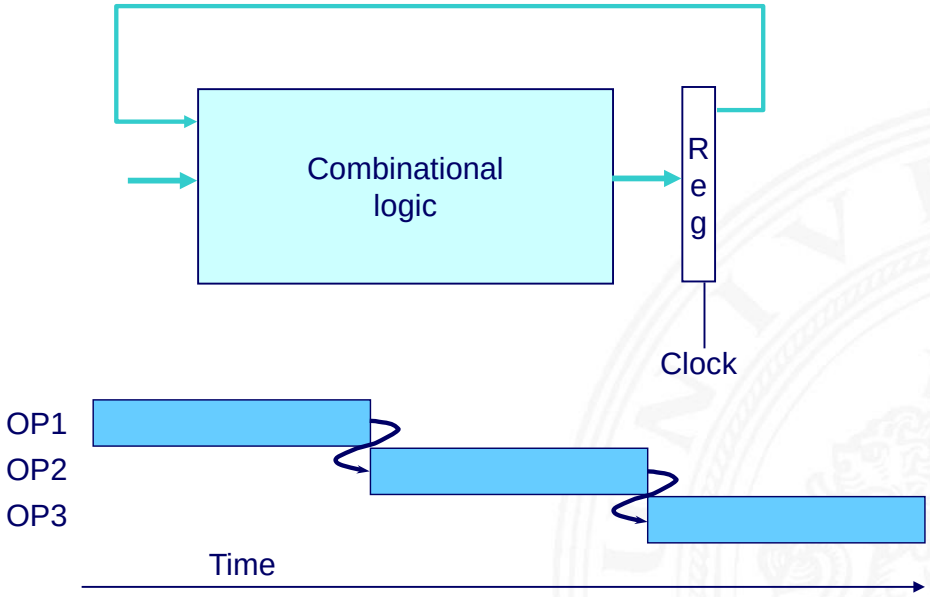
Probleme: Register „Overhead“



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

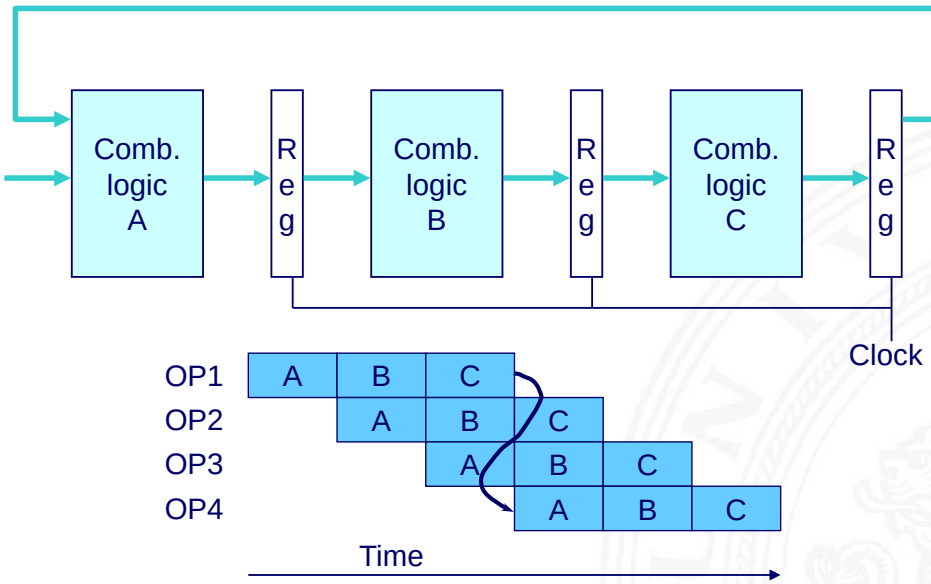
		Overhead	Taktperiode
1-Register:	6,25%	20 ps	320 ps
3-Register:	16,67%	20 ps	120 ps
6-Register:	28,57%	20 ps	70 ps

Probleme: Datenabhängigkeiten / „Daten Hazards“



- jede Operation hängt vom Ergebnis der Vorhergehenden ab

Probleme: Datenabhängigkeiten / „Daten Hazards“ (cont.)



⇒ Resultat-Feedback kommt zu spät für die nächste Operation

⇒ Pipelining ändert Verhalten des gesamten Systems

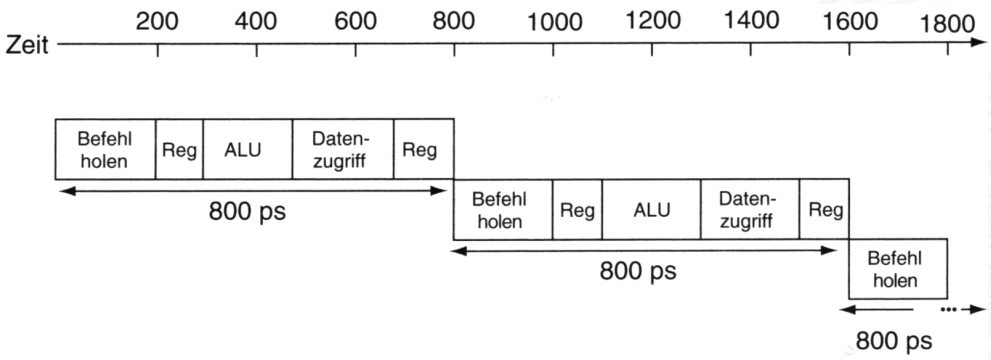
RISC Pipelining

Schritte der RISC Befehlsabarbeitung (von ISA abhängig)

▶ IF	**I**nstruction **F**etch Instruktion holen, in Befehlsregister laden
ID	**I**nstruction **D**ecode Instruktion decodieren
OF	**O**perand **F**etch Operanden aus Registern holen
EX	**Ex**ecute ALU führt Befehl aus
MEM	**Mem**ory access Speicherzugriff bei Load-/Store-Befehlen
WB	**W**rite **B**ack Ergebnisse in Register zurückschreiben

RISC Pipelining (cont.)

- je nach Instruktion sind 3-5 dieser Schritte notwendig
- Beispiel *ohne* Pipelining:

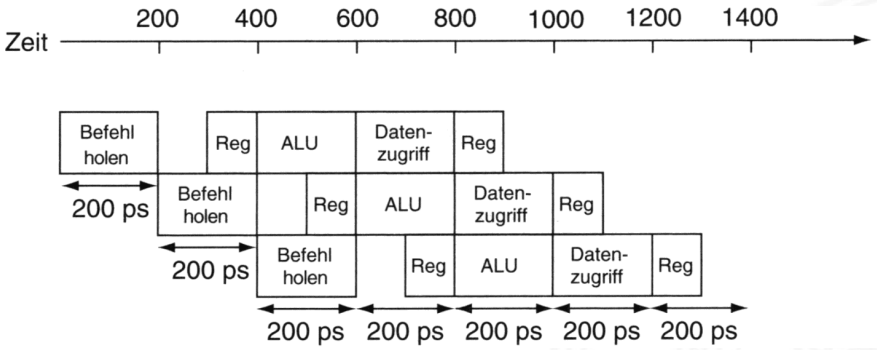


Patterson, Hennessy, *Computer Organization and Design*

RISC Pipelining (cont.)

Pipelining in Prozessoren

- Beispiel *mit* Pipelining:



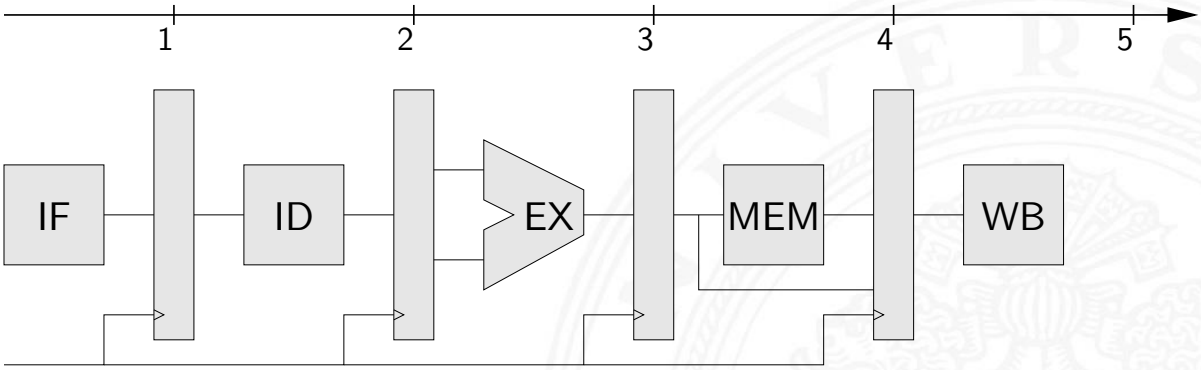
Patterson, Hennessy, *Computer Organization and Design*

- Befehle überlappend ausführen
- Register trennen Pipelinestufen

RISC Pipelining (cont.)

- RISC ISA: Pipelining wird direkt umgesetzt

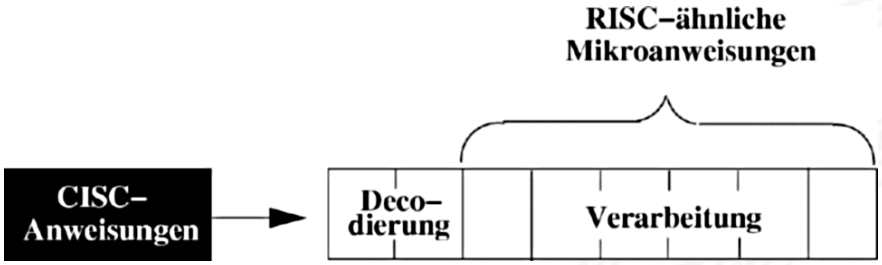
Pipelinestufen



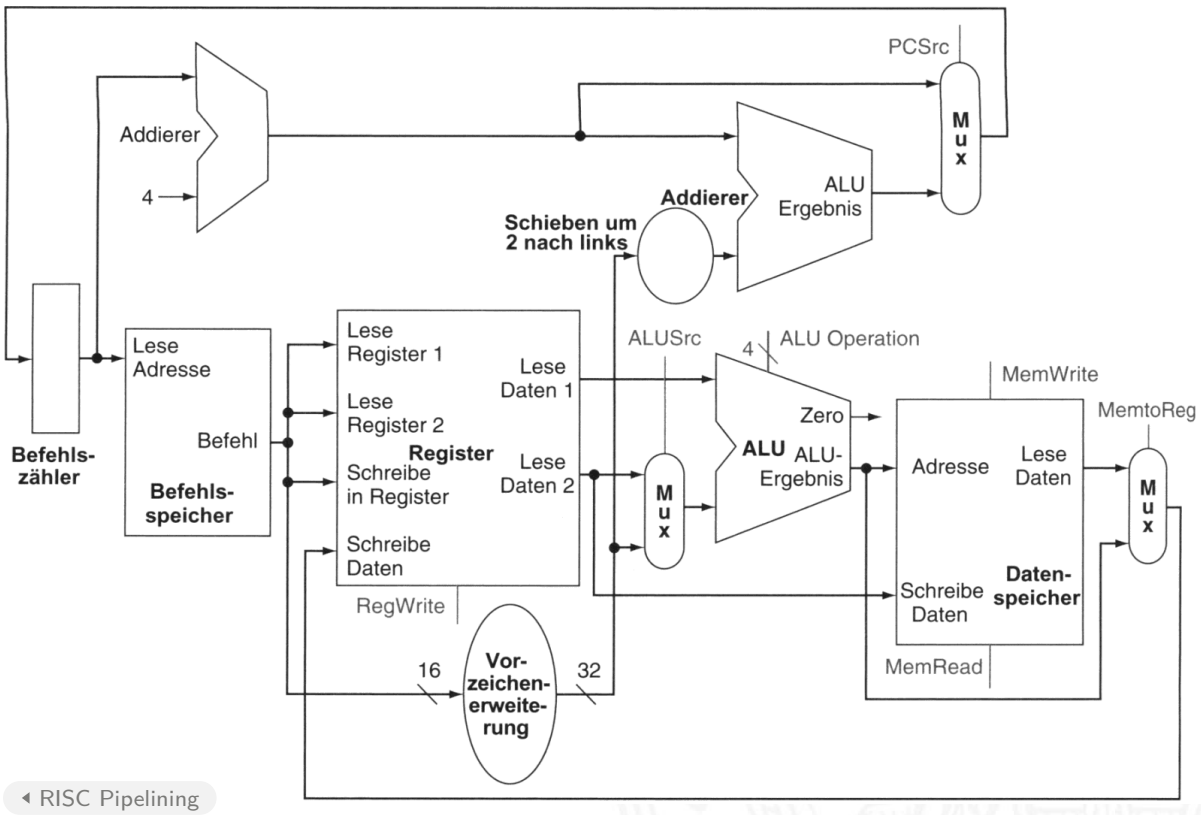
- MIPS-Architektur (aus Patterson, Hennessy)
  - MIPS ohne Pipeline
  - MIPS Pipeline
  - Pipeline Schema
- Bryant, O'Hallaron, *Computer systems*
  - Pipeline – Register
  - Pipeline – Architektur

RISC Pipelining (cont.)

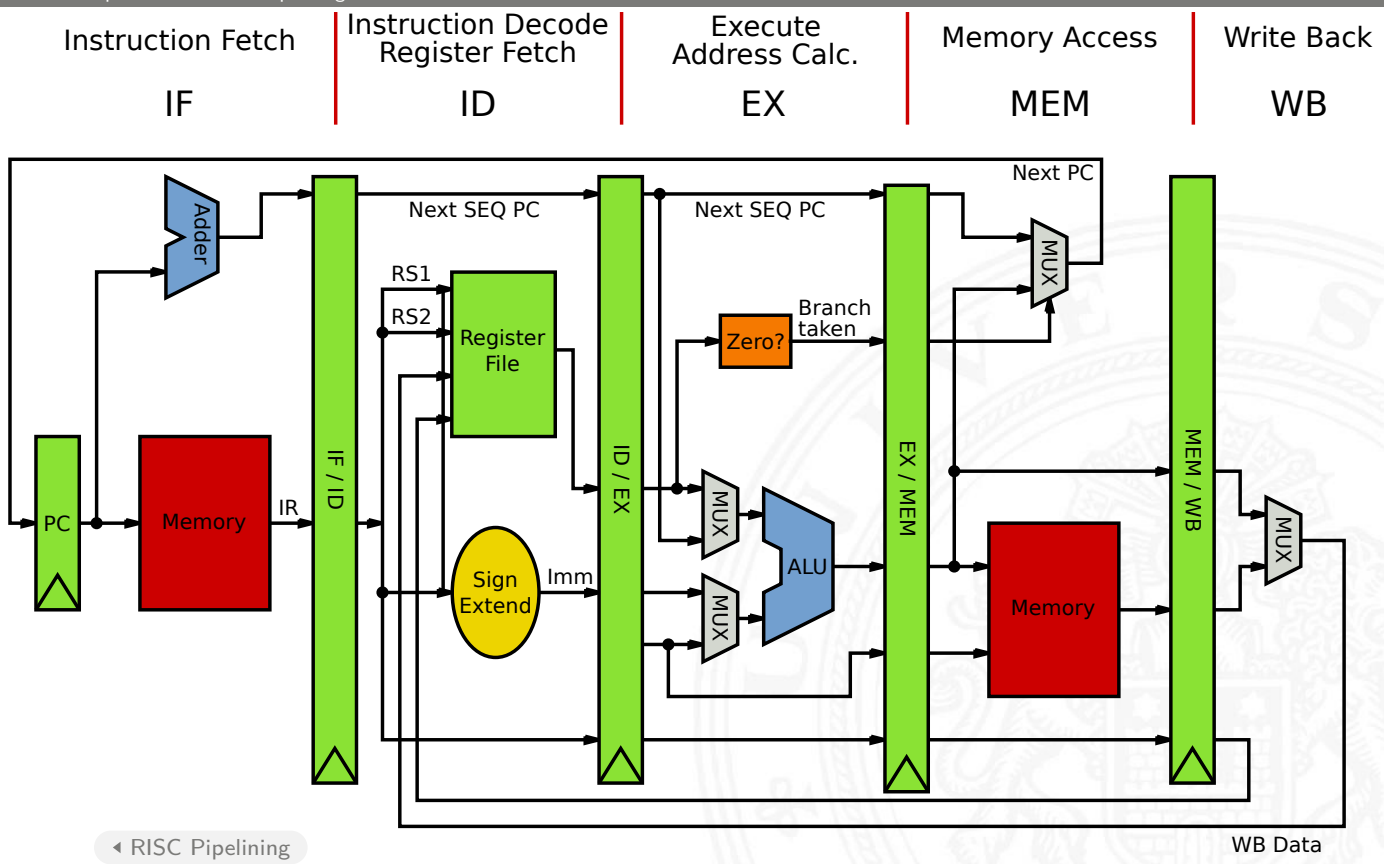
- CISC ISA (x86): Umsetzung der CISC Befehle in Folgen RISC-ähnlicher Anweisungen



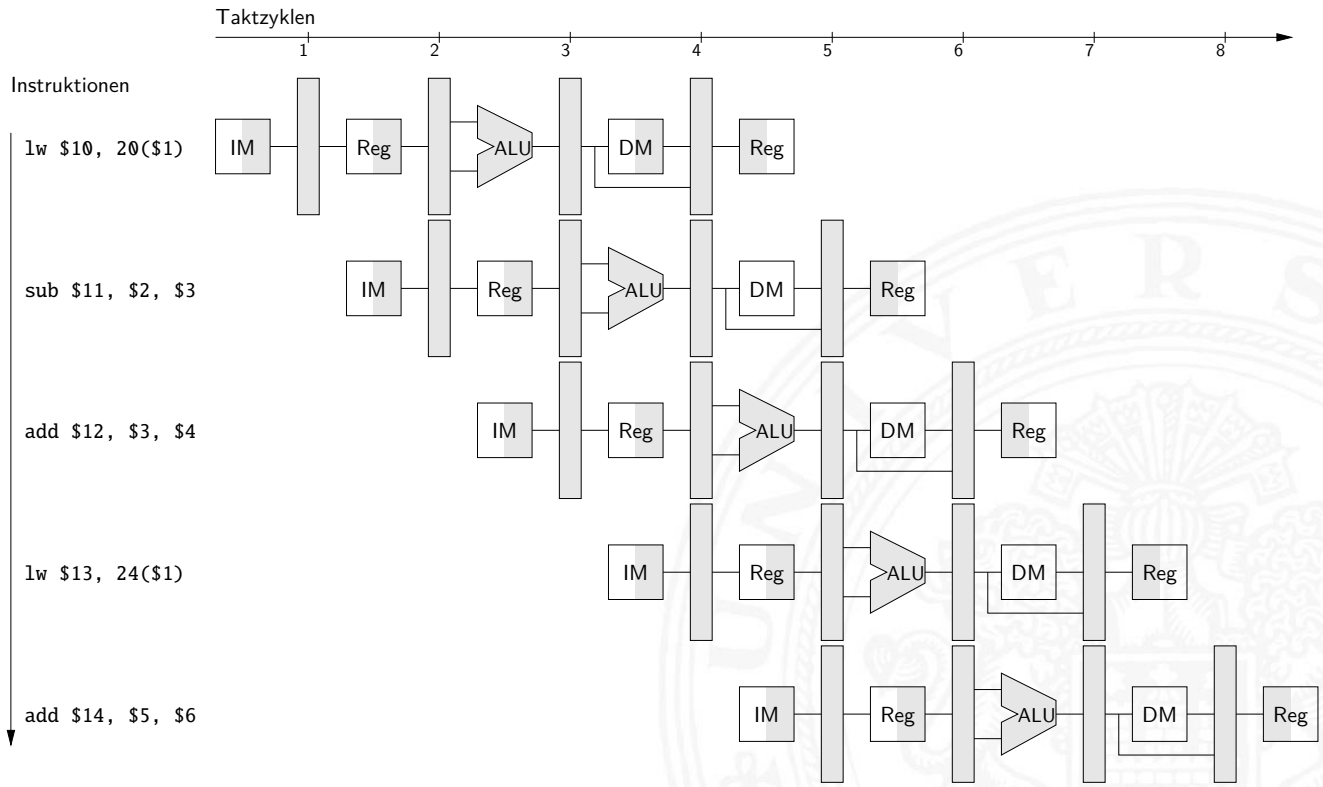
+ CISC-Software bleibt lauffähig
+ Befehlssatz wird um neue RISC Befehle erweitert



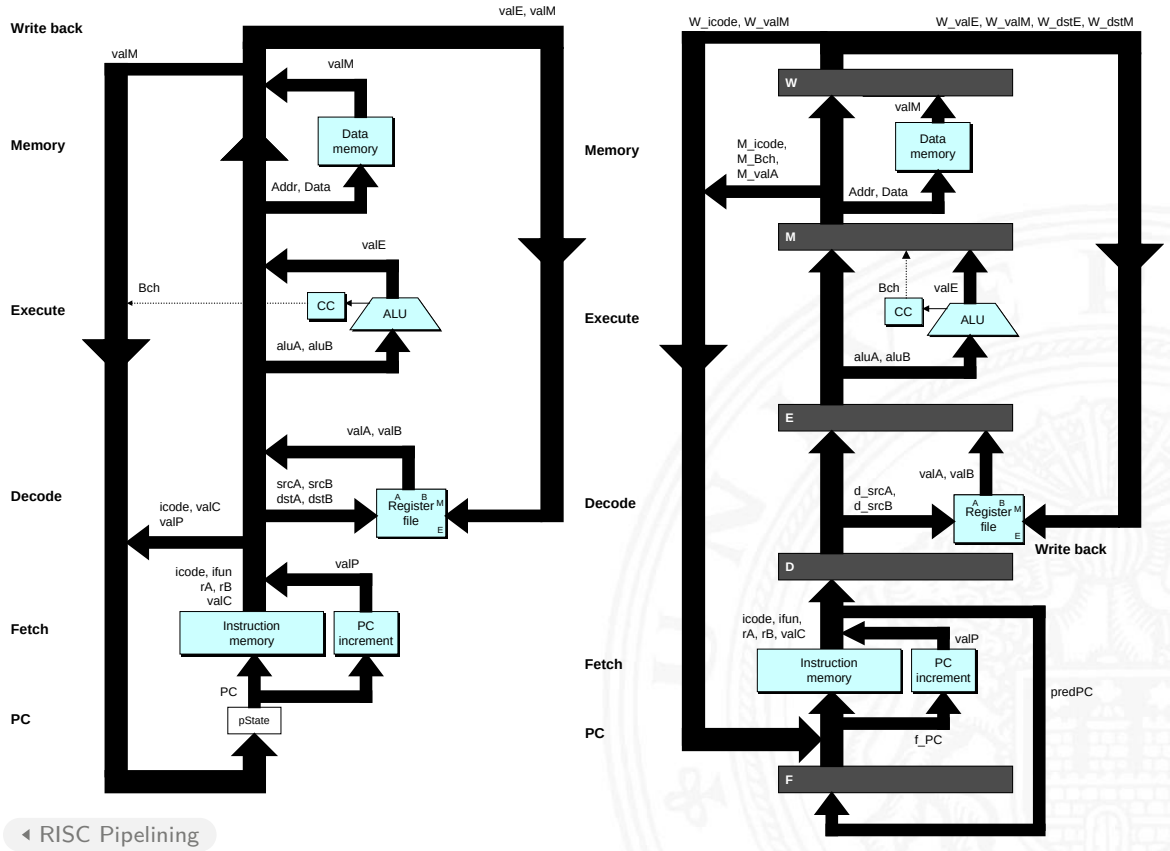
RISC Pipelining



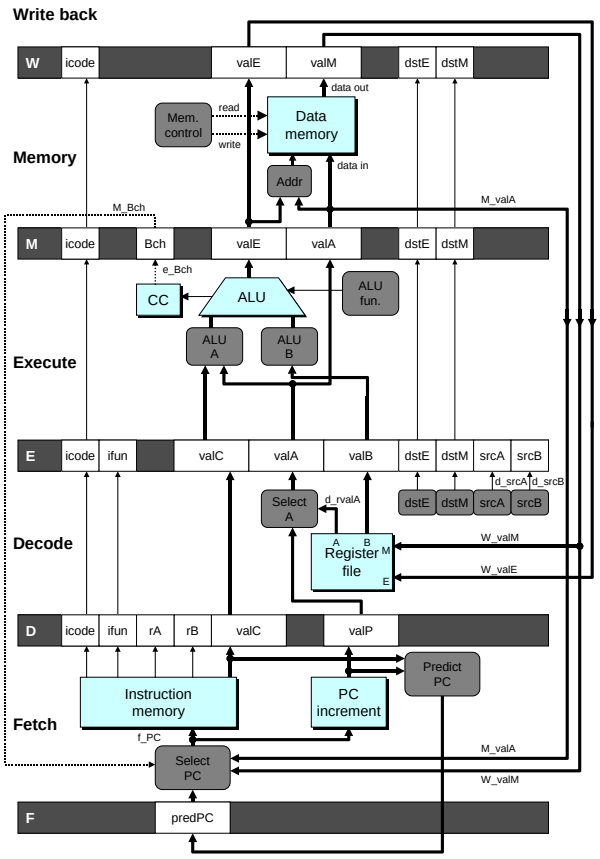
RISC Pipelining



RISC Pipelining



RISC Pipelining



RISC Pipelining

Prozessorpipeline – Begriffe

Begriffe

- **Pipeline-Stage**: einzelne Stufe der Pipeline
- **Pipeline Machine Cycle**:
  Instruktion kommt einen Schritt in Pipeline weiter
- **Durchsatz**: Anzahl der Instruktionen, die in jedem Takt abgeschlossen werden
- **Latenz**: Zeit, die eine Instruktion benötigt, um alle Pipelinestufen zu durchlaufen

Prozessorpipeline – Bewertung

Vor- und Nachteile

+ Pipelining ist für den Programmierer nicht sichtbar!
+ höherer Instruktionsdurchsatz $\Rightarrow$ bessere Performanz
– Latenz wird nicht verbessert, bleibt bestenfalls gleich
– Pipeline Takt limitiert durch langsamste Pipelinestufe
  unausgewogene Pipelinestufen reduzieren den Takt und damit die Performanz
– zusätzliche Zeiten, um Pipeline zu füllen bzw. zu leeren

Prozessorpipeline – Speed-Up

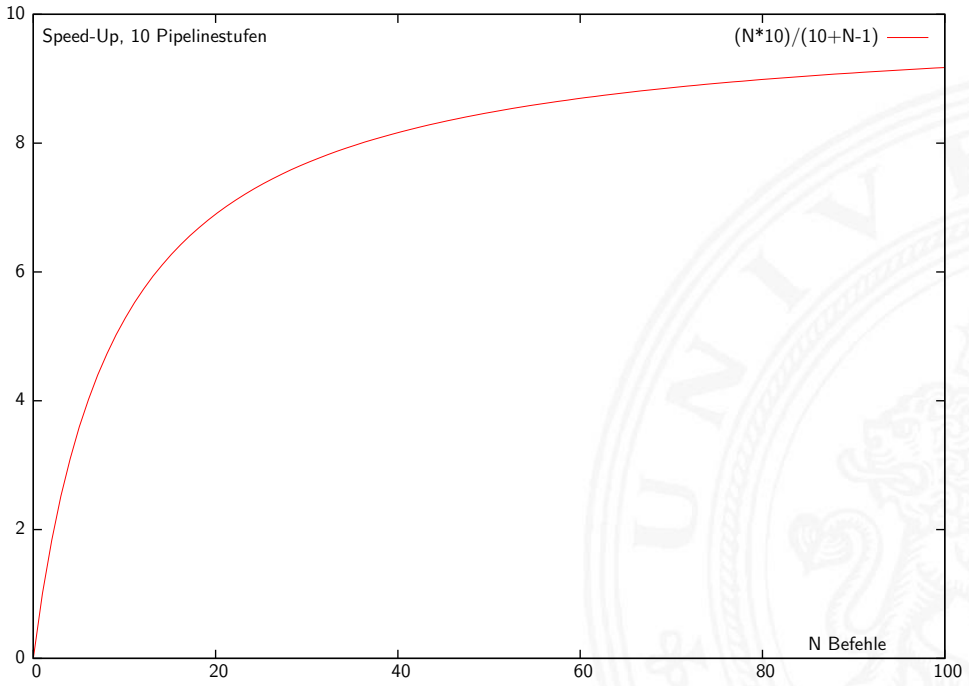
Pipeline Speed-Up

- N Instruktionen; K Pipelinestufen
- ohne Pipeline: $N \cdot K$ Taktzyklen
- mit Pipeline: $K + N - 1$ Taktzyklen
- Speed-Up $= \frac{N \cdot K}{K+N-1}$, $\lim_{N \to \infty} S = K$

$\Rightarrow$ ein großer Speed-Up wird erreicht durch

1. große Pipelinetiefe: K
2. lange Instruktionssequenzen: N

Prozessorpipeline – Speed-Up (cont.)



Prozessorpipeline – Dimensionierung

Dimensionierung der Pipeline

- Längere Pipelines
- Pipelinestufen in den Einheiten / den ALUs (*superskalar*)

$\Rightarrow$ größeres K wirkt sich direkt auf den Durchsatz aus

$\Rightarrow$ weniger Logik zwischen den Registern, höhere Taktfrequenzen

- Beispiele

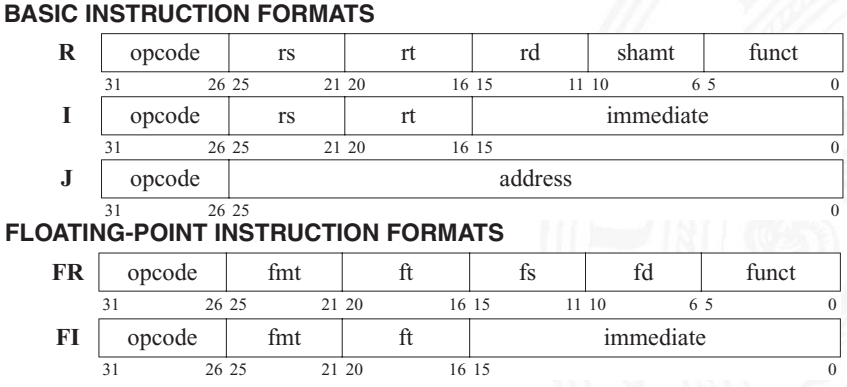
CPU	Pipelinestufen	Taktfrequenz [*MHz*]
Pentium	5	300
Motorola G4	4	500
Motorola G4e	7	1000
Pentium II/III	12	1400
Athlon XP	10/15	2500
Athlon 64, Opteron	12/17	$\leq$ 3000
Pentium 4	20	$\leq$ 5000

Prozessorpipeline – Auswirkungen

Architekturentscheidungen, die sich auf das Pipelining auswirken

gut für Pipelining

- gleiche Instruktionslänge
- wenige Instruktionsformate
- Load/Store Architektur



MIPS-Befehlsformate

Prozessorpipeline – Auswirkungen (cont.)

schlecht für Pipelining: *Pipelinekonflikte / -Hazards*

- Strukturkonflikt: gleichzeitiger Zugriff auf eine Ressource durch mehrere Pipelinestufen
- Datenkonflikt: Ergebnisse von Instruktionen werden innerhalb der Pipeline benötigt
- Steuerkonflikt: Sprungbefehle in der Pipelinesequenz

sehr schlecht für Pipelining

- Unterbrechung des Programmkontexts: Interrupt, System-Call, Exception...
- (Performanz-) Optimierungen mit „Out-of-Order Execution“ etc.

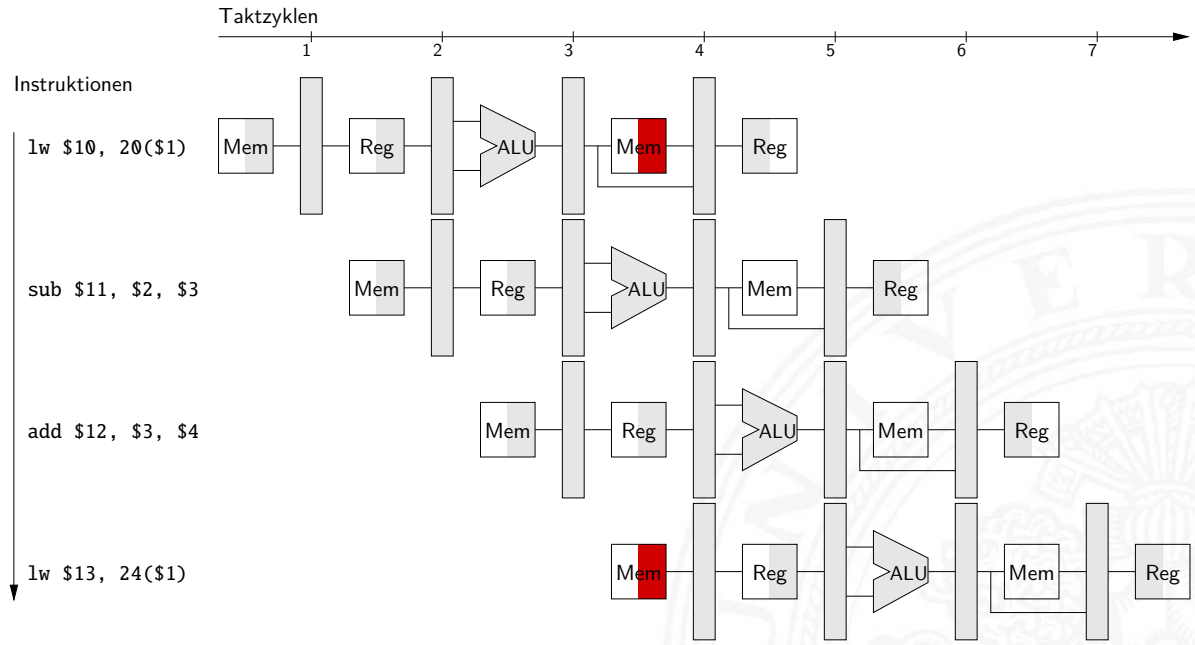
Pipeline Strukturkonflikte

Strukturkonflikt / Structural Hazard

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

⇒ Mehrfachauslegung der betreffenden Ressourcen

- Harvard-Architektur vermeidet Strukturkonflikt aus Beispiel
- Multi-Port Register
- mehrfach vorhandene Busse und Multiplexer...



◂ Strukturkonflikte

Pipeline Datenkonflikte

Datenkonflikt / Data Hazard

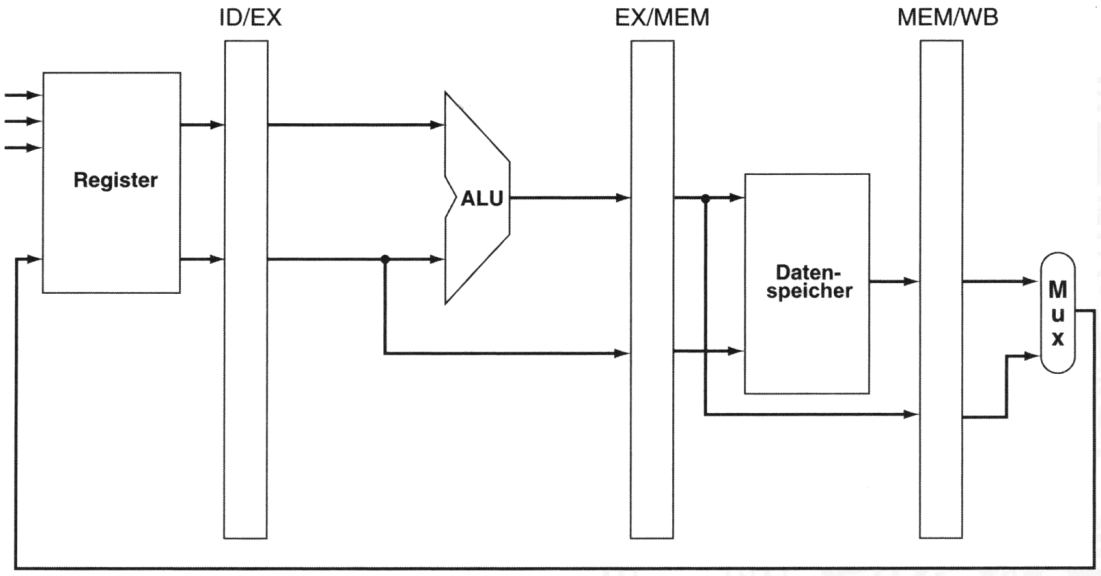
- eine Instruktion braucht die Ergebnisse einer vorhergehenden, diese wird aber noch in der Pipeline bearbeitet
- Datenabhängigkeiten der Stufe „Befehl ausführen“

▸ Beispiel

Forwarding

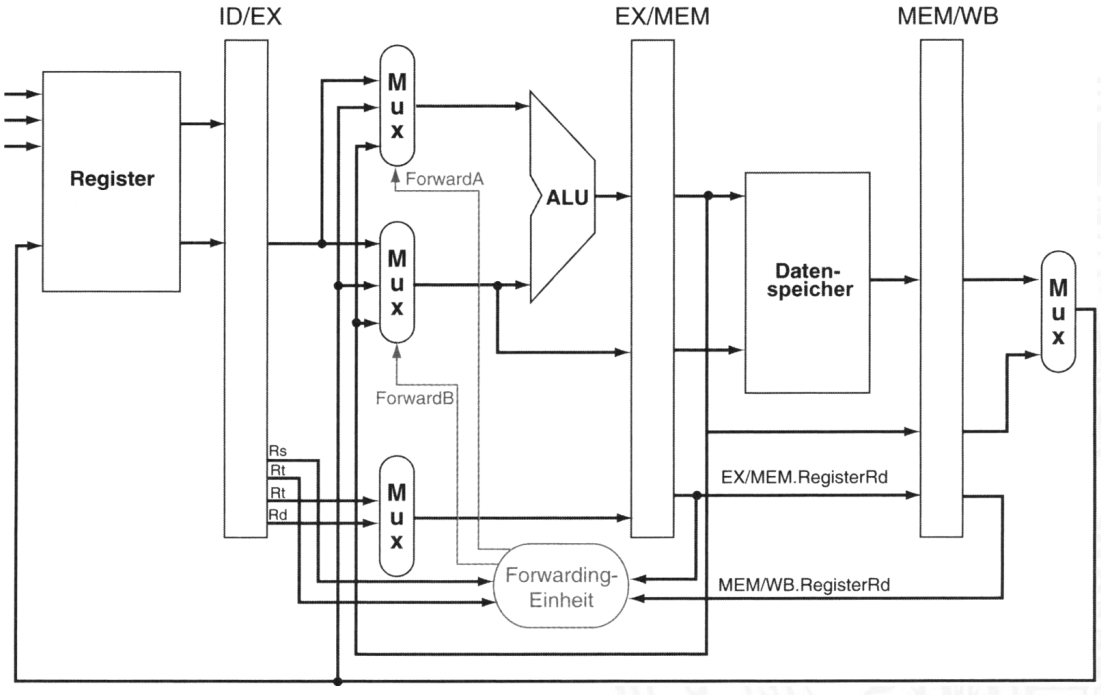
- kann Datenabhängigkeiten auflösen, s. Beispiel
- extra Hardware: *„Forwarding-Unit“*
- Änderungen in der Pipeline Steuerung
- neue Datenpfade und Multiplexer

Pipeline Datenkonflikte (cont.)



MIPS
Datenpfad

Pipeline Datenkonflikte (cont.)



MIPS
Forwarding

Pipeline Datenkonflikte (cont.)

Rückwärtsabhängigkeiten

- spezielle Datenabhängigkeit ▸ Beispiel
- Forwarding-Technik funktioniert nicht, da die Daten erst *später* zur Verfügung stehen
  - bei längeren Pipelines
  - bei Load-Instruktionen (s.u.)

Auflösen von Rückwärtsabhängigkeiten

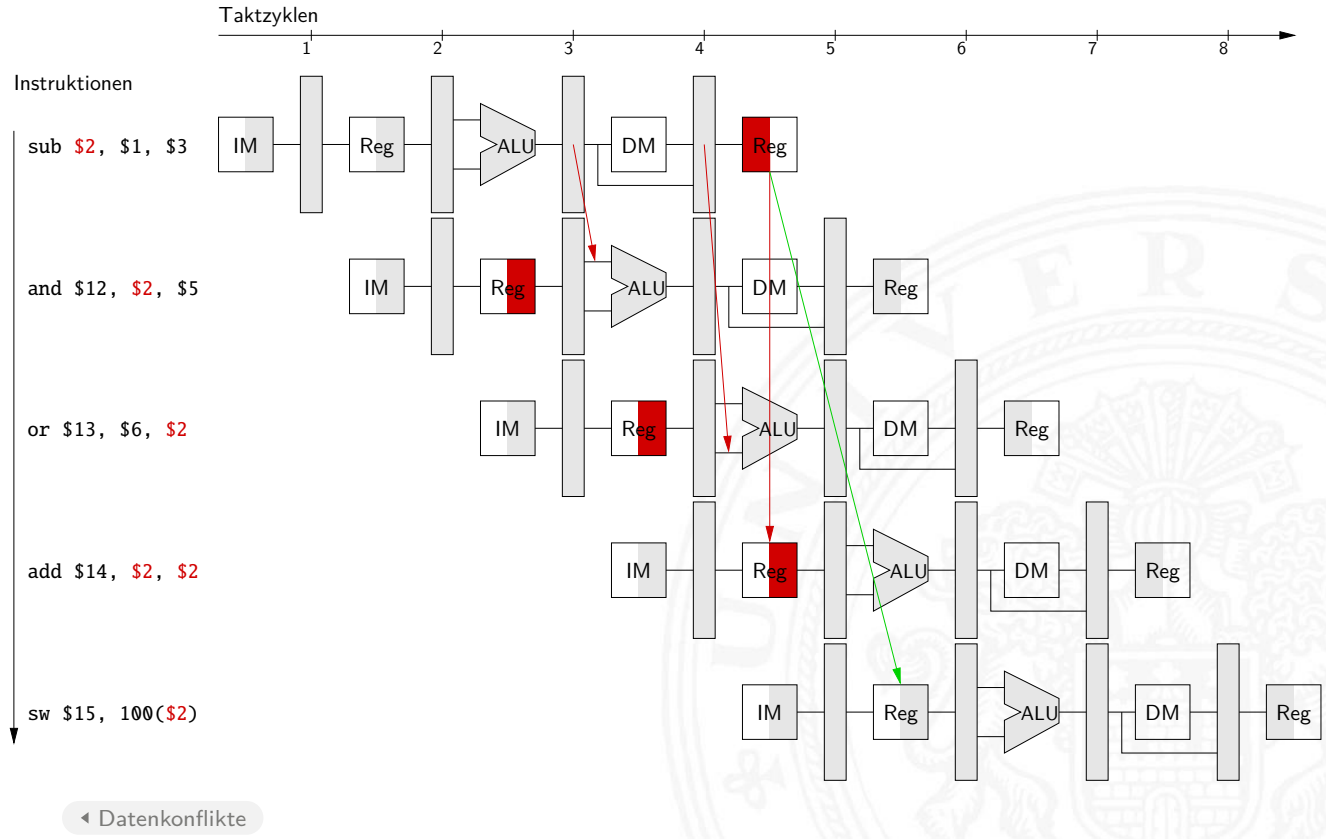
1. Softwarebasiert, durch den Compiler, Reihenfolge der Instruktionen verändern ▸ Beispiel
   - andere Operationen (ohne Datenabhängigkeiten) vorziehen
   - nop-Befehl(e) einfügen

Pipeline Datenkonflikte (cont.)

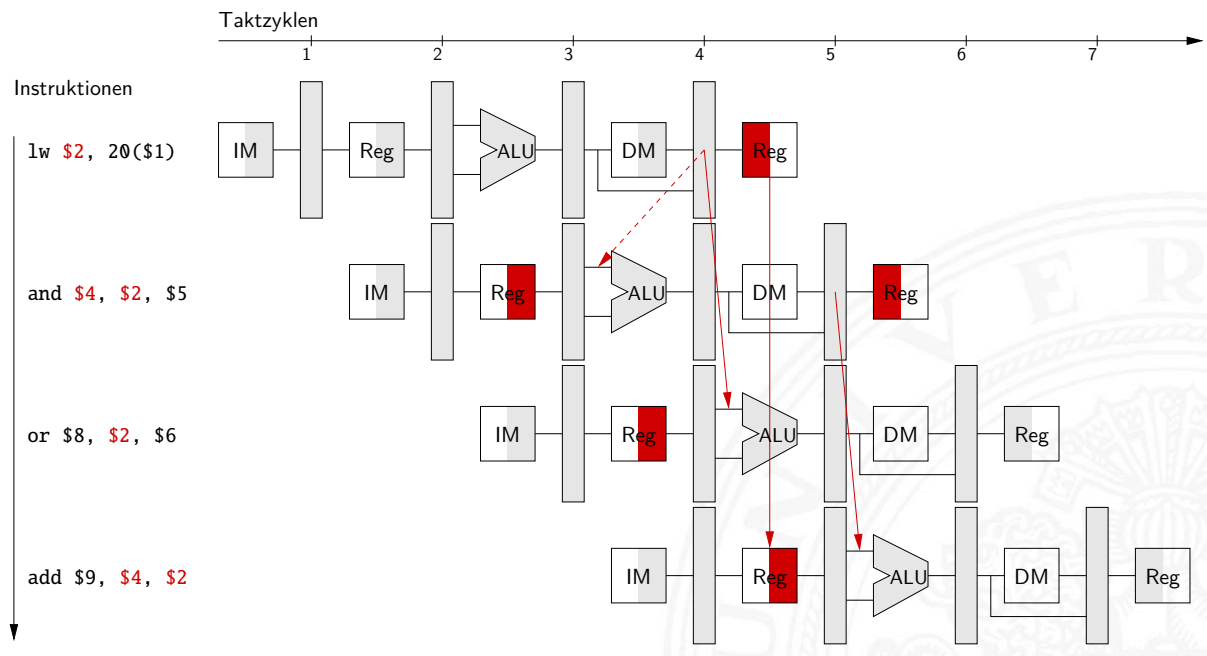
2. *„Interlocking“* ▸ Beispiel
   - zusätzliche (Hardware) Kontrolleinheit
   - verschiedene Strategien
   - in Pipeline werden keine neuen Instruktionen geladen
   - Hardware erzeugt: Pipelineleerlauf / *„pipeline stall“*

„Scoreboard“

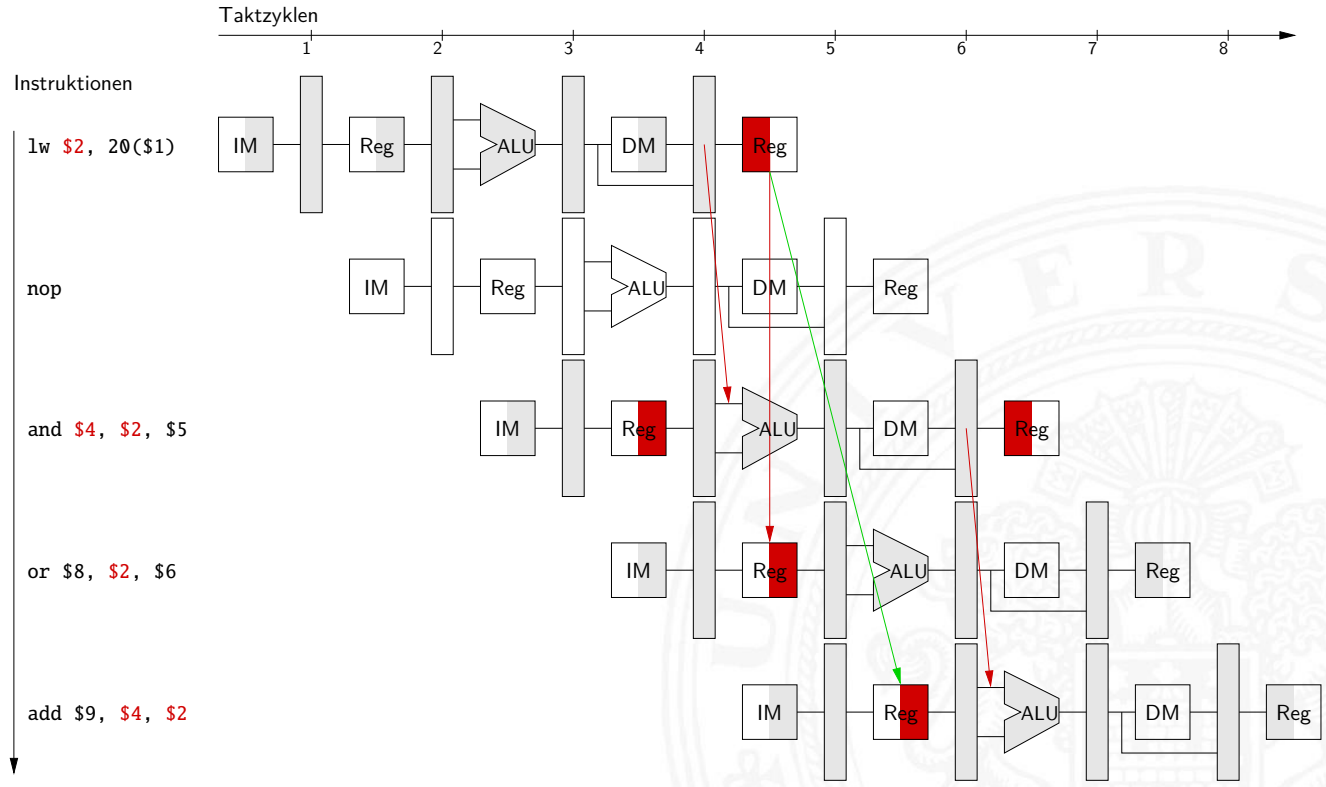
- Hardware Einheit zur zentralen Hazard-Erkennung und -Auflösung
- Verwaltet Instruktionen, benutzte Einheiten und Register der Pipeline



Datenkonflikte

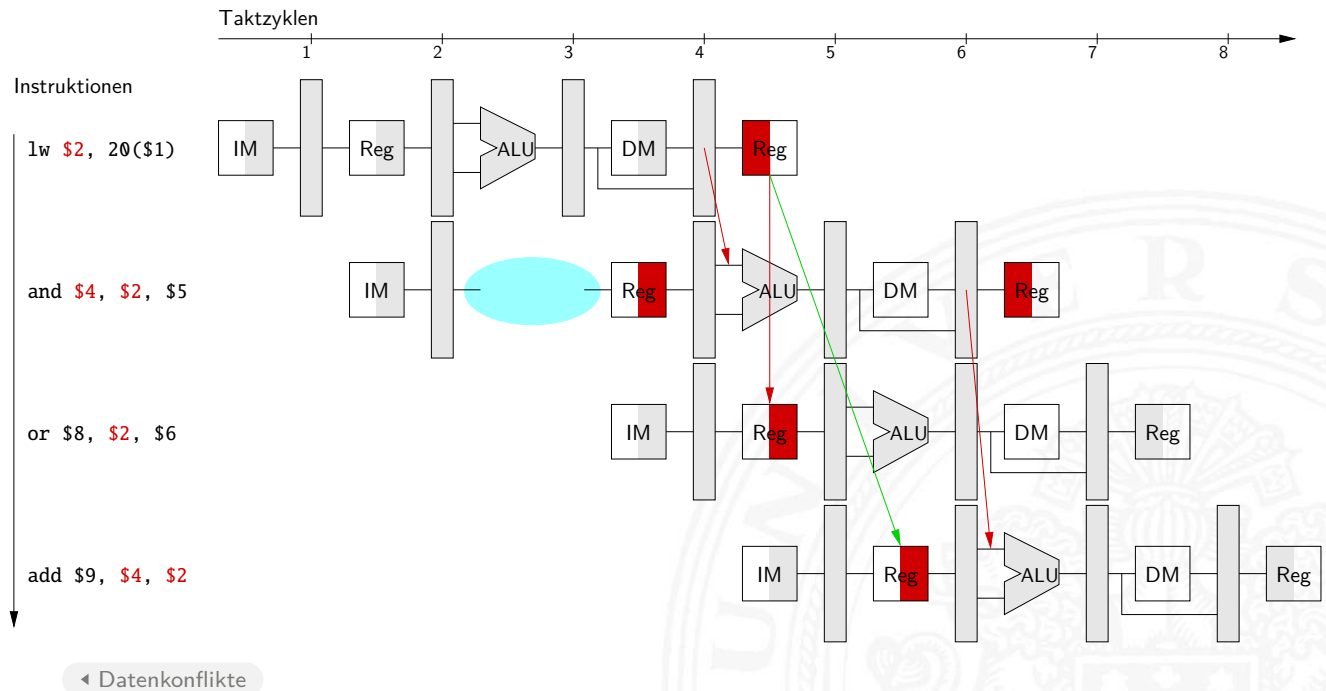


Datenkonflikte



Datenkonflikte

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
20.3 Computerarchitektur - Pipelining
64-040 Rechnerstrukturen



Datenkonflikte

Pipeline Steuerkonflikte

Steuerkonflikt / Control Hazard

- Sprungbefehle unterbrechen den sequenziellen Ablauf der Instruktionen
- Problem: Instruktionen die auf (bedingte) Sprünge folgen, werden in die Pipeline geschoben, bevor bekannt ist, ob verzweigt werden soll
- Beispiel: bedingter Sprung

▸ Beispiel

Pipeline Steuerkonflikte (cont.)

Lösungsmöglichkeiten für Steuerkonflikte

- ad-hoc Lösung: „Interlocking“ erzeugt Pipelineleerlauf
  - – ineffizient: ca. 19 % der Befehle sind Sprünge

1. Annahme: nicht ausgeführter Sprung / „untaken branch“
   - \+ kaum zusätzliche Hardware
   - – im Fehlerfall
     - Pipelineleerlauf
     - Pipeline muss geleert werden / „flush instructions“
2. Sprungentscheidung „vorverlegen“
   - Software: Compiler zieht andere Instruktionen vor Verzögerung nach Sprungbefehl / „delay slots“
   - Hardware: Sprungentscheidung durch Zusatz-ALU (nur Vergleiche) während Befehlsdecodierung (z.B. MIPS)

Pipeline Steuerkonflikte (cont.)

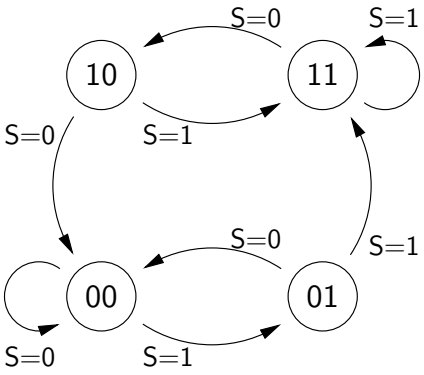
3. Sprungvorhersage / *„branch prediction“*
   - Beobachtung: ein Fall tritt häufiger auf:
     Schleifendurchlauf, Datenstrukturen durchsuchen etc.
   - mehrere Vorhersageverfahren; oft miteinander kombiniert
   + hohe Trefferquote: bis 90 %

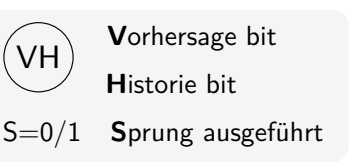
   Statische Sprungvorhersage (softwarebasiert)
   - Compiler erzeugt extra Bit in Opcode des Sprungbefehls
   - Methoden: Codeanalyse, Profiling...

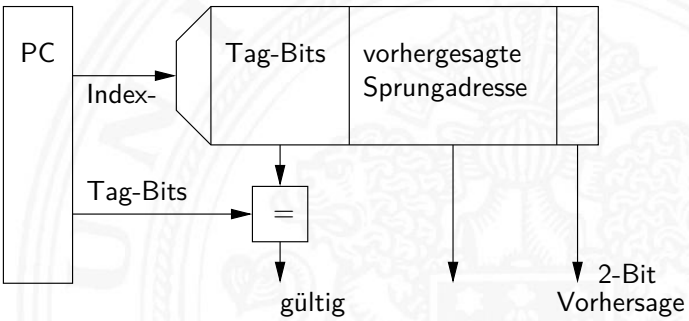
   Dynamische Sprungvorhersage (hardwarebasiert)
   - Sprünge durch Laufzeitinformation vorhersagen:
     Wie oft wurde der Sprung in letzter Zeit ausgeführt?
   - viele verschiedene Verfahren:
     History-Bit, 2-Bit Prädiktor, korrelationsbasierte Vorhersage,
     Branch History Table, Branch Target Cache...

Pipeline Steuerkonflikte (cont.)

- Beispiel: 2-Bit Sprungvorhersage + Branch Target Cache

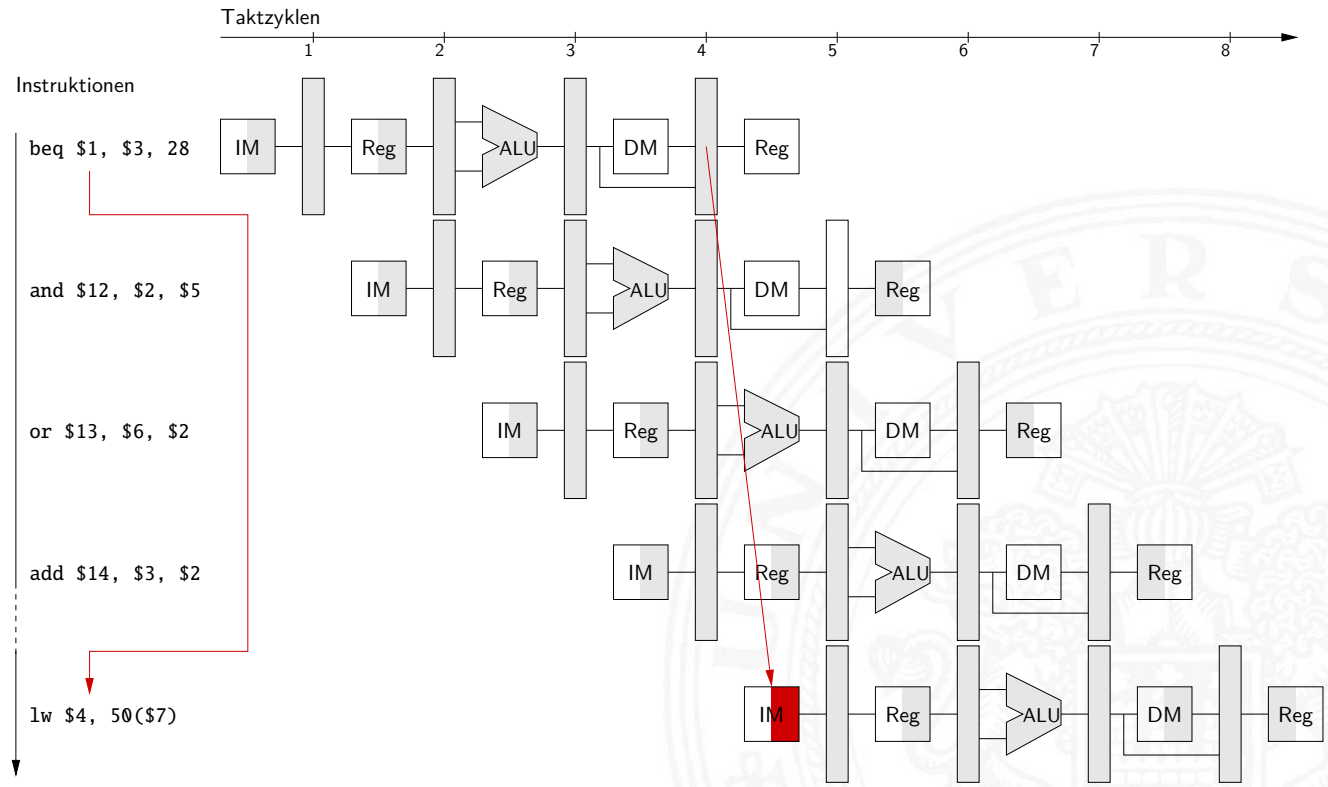






Pipeline Steuerkonflikte (cont.)

- Schleifen abrollen / „*Loop unrolling*“
  - zusätzliche Maßnahme zu allen zuvor skizzierten Verfahren
  - bei statische Schleifenbedingung möglich
  - Compiler iteriert Instruktionen in der Schleife (teilweise)
  - – längerer Code
  - + Sprünge und Abfragen entfallen
  - + erzeugt sehr lange Codesequenzen ohne Sprünge
    ⇒ Pipeline kann optimal ausgenutzt werden



Steuerkonflikte

Superskalare Prozessoren

- Superskalare CPUs besitzen mehrere Recheneinheiten: 4...10
- In jedem Takt werden (dynamisch) mehrere Instruktionen eines konventionell linearen Instruktionsstroms abgearbeitet: $CPI < 1$
  ILP (**I**nstruction **L**evel **P**arallelism) ausnutzen!
- Hardware verteilt initiierte Instruktionen auf Recheneinheiten
- Pro Takt kann *mehr als eine* Instruktion initiiert werden
  Die Anzahl wird dynamisch von der Hardware bestimmt:
  0...„*Instruction Issue Bandwidth*“

\+ sehr effizient, alle modernen CPUs sind superskalar

– Abhängigkeiten zwischen Instruktionen sind der Engpass, das Problem der Hazards wird verschärft

Superskalar – Datenabhängigkeiten

Datenabhängigkeiten

- RAW – **R**ead **A**fter **W**rite
  Instruktion I_x darf Datum erst lesen, wenn I_{x-n} geschrieben hat
- WAR – **W**rite **A**fter **R**ead
  Instruktion I_x darf Datum erst schreiben, wenn I_{x-n} gelesen hat
- WAW – **W**rite **A**fter **W**rite
  Instruktion I_x darf Datum erst überschreiben, wenn I_{x-n} geschrieben hat

Superskalar – Datenabhängigkeiten (cont.)

Datenabhängigkeiten superskalarer Prozessoren

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

„*Register Renaming*“

- Hardware löst Datenabhängigkeiten innerhalb der Pipeline auf
- Zwei Registersätze sind vorhanden
  1. Architektur-Register: „logische Register“ der ISA
  2. viele Hardware-Register: „Rename Register“
     - dynamische Abbildung von ISA- auf Hardware-Register

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
20.4 Computerarchitektur - Superskalare Prozessoren 64-040 Rechnerstrukturen

Superskalar – Datenabhängigkeiten (cont.)

- Beispiel

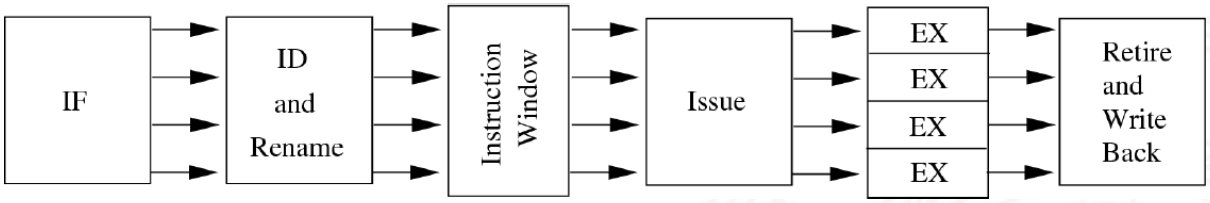
Originalcode	nach Renaming
`tmp  = a + b;`	`tmp1 = a + b;`
`res1 = c + tmp;`	`res1 = c + tmp1;`
`tmp  = d + e;`	`tmp2 = d + e;`
`res2 = tmp - f;`	`res2 = tmp2 - f;`
	`tmp  = tmp2;`

Parallelisierung des modifizierten Codes

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

Superskalar – Pipeline

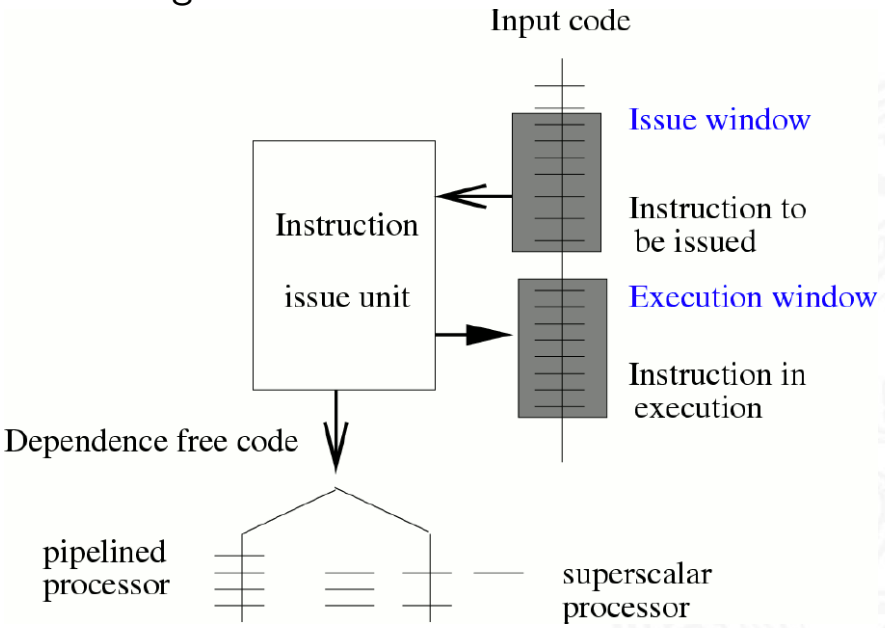
Aufbau der superskalaren Pipeline



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

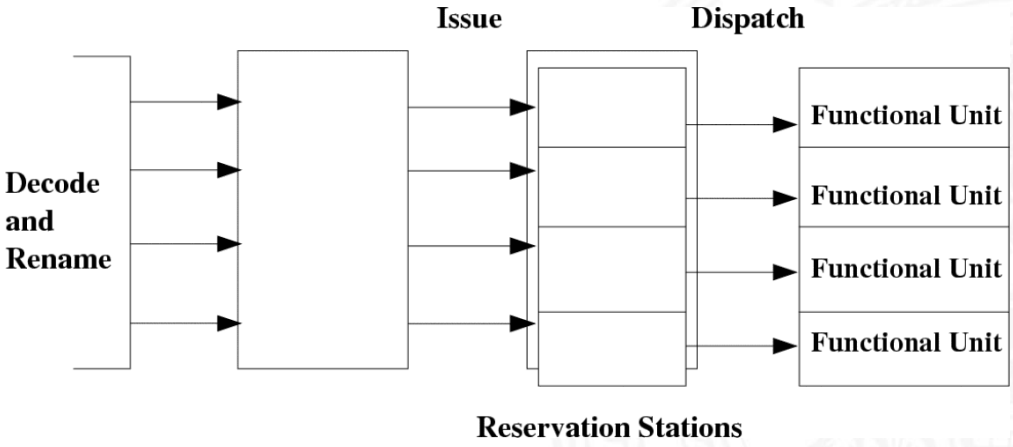
Superskalar – Pipeline (cont.)

Scheduling der Instruktionen



Superskalar – Pipeline (cont.)

- Dynamisches Scheduling erzeugt *out-of-order* Reihenfolge der Instruktionen
- Issue: globale Sicht
  Dispatch: getrennte Ausschnitte in „Reservation Stations“



Superskalar – Pipeline (cont.)

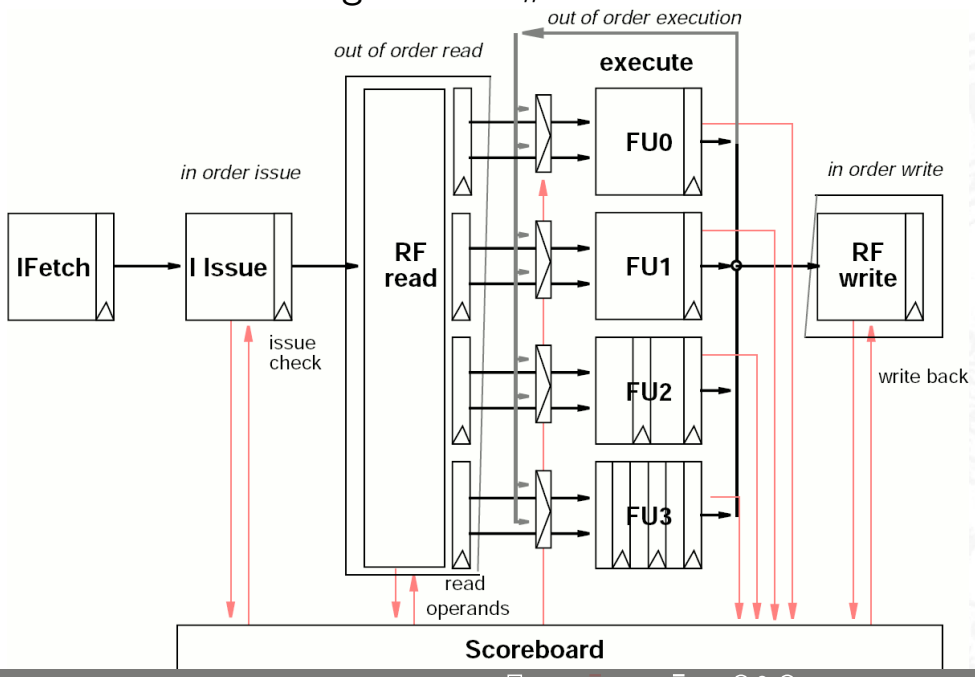
Reservation Station für jede Funktionseinheit

- speichert: initiierte Instruktionen die auf Recheneinheit warten
- –"– zugehörige Operanden
- –"– ggf. Zusatzinformation
- Instruktion bleibt blockiert, bis alle Parameter bekannt sind und wird dann an die zugehörige ALU weitergeleitet

- Dynamisches Scheduling: zuerst ’67 in IBM 360 (Robert Tomasulo)
  - Forwarding
  - Registerumbenennung und Reservation Stations

Superskalar – Scoreboard

Zentrale Verwaltungseinheit: „Scoreboard“



Superskalar – Scoreboard (cont.)

Scoreboard erlaubt das Management mehrerer Ausführungseinheiten

- out-of-order Ausführung von Mehrzyklusbefehlen
- Auflösung aller Struktur- und Datenkonflikte: RAW, WAW, WAR

Einschränkungen

- single issue (nicht superskalar)
- in-order issue
- keine Umbenennungen; also Leerzyklen bei WAR- und WAW-Konflikten
- kein Forwarding, daher Zeitverlust bei RAW-Konflikten

Superskalar – Retire-Stufe

„Retire“

- erzeugt wieder *in-order* Reihenfolge
- FIFO: Reorder-Buffer
- commit: „richtig ausgeführte“ Instruktionen gültig machen
- abort: Sprungvorhersage falsch
  Instruktionen verwerfen

Probleme superskalarer Pipelines

Spezielle Probleme superskalarer Pipelines

- weitere Hazard-Möglichkeiten
  - die verschiedenen ALUs haben unterschiedliche Latenzzeiten
  - Befehle „warten“ in den Reservation Stations
  - $\Rightarrow$ Datenabhängigkeiten können sich mit jedem Takt ändern
- Kontrollflussabhängigkeiten: Anzahl der Instruktionen zwischen bedingten Sprüngen limitiert Anzahl parallelisierbarer Instruktion

$\Rightarrow$ „Loop Unrolling“ wichtig
$+$ optimiertes (dynamisches) Scheduling: Faktor 3 möglich

Software Pipelining

Softwareunterstützung für Pipelining superskalarer Prozessoren

- Codeoptimierungen beim Compilieren: Ersatz für, bzw. Ergänzend zu der Pipelineunterstützung durch Hardware
- Compiler hat „globalen“ Überblick
  $\Rightarrow$ zusätzliche Optimierungsmöglichkeiten
- symbolisches Loop Unrolling
- Loop Fusion
- ...

Superskalar – Interrupts

Exceptions, Interrupts und System-Calls

- Interruptbehandlung ist wegen der Vielzahl paralleler Aktionen und den Abhängigkeiten innerhalb der Pipelines extrem aufwändig
  - da unter Umständen noch Pipelineaktionen beendet werden müssen, wird *zusätzliche Zeit* bis zur Interruptbehandlung benötigt
  - wegen des Register-Renaming muss sehr viel *mehr Information* gerettet werden als nur die ISA-Register
- Prinzip der Interruptbehandlung
  - keine neuen Instruktionen mehr initiieren
  - warten bis Instruktionen des Reorder-Buffers abgeschlossen sind

Superskalar – Interrupts (cont.)

- Verfahren ist von der „Art“ des Interrupt abhängig
  - Precise-Interrupt: Pipelineaktivitäten komplett Beenden
  - Imprecise-Interrupt: wird als verzögerter Sprung (Delayed-Branching) in Pipeline eingebracht
    Zusätzliche Register speichern Information über Instruktionen die in der Pipeline nicht abgearbeitet werden können (z.B. weil sie den Interrupt ausgelöst haben)
- Definition: Precise-Interrupt
  - Programmzähler (PC) zur Interrupt auslösenden Instruktion ist bekannt
  - Alle Instruktionen bis zur PC-Instruktion wurden vollständig ausgeführt
  - Keine Instruktion nach der PC-Instruktion wurde ausgeführt
  - Ausführungszustand der PC-Instruktion ist bekannt

Ausnahmebehandlung

Ausnahmebehandlung („Exception Handling“)

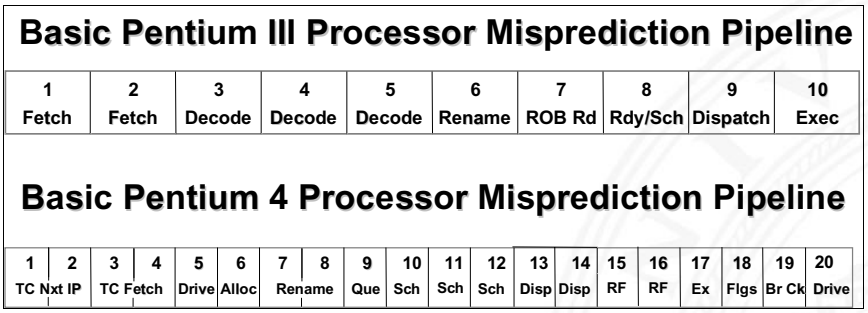
- Pipeline kann normalen Ablauf nicht fortsetzen
- Ursachen
  - „Halt“ Anweisung
  - ungültige Adresse für Anweisung oder Daten
  - ungültige Anweisung
  - Pipeline Kontrollfehler
- erforderliches Vorgehen
  - einige Anweisungen vollenden
    Entweder aktuelle oder vorherige (hängt von Ausnahmetyp ab)
  - andere verwerfen
  - „Exception Handler“ aufrufen: spez. Prozeduraufruf

Pentium 4 / NetBurst Architektur

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

Pentium 4 / NetBurst Architektur (cont.)

- „Double pumped“ ALUs (2 Operationen pro Taktzyklus)
- große Pipelinelänge $\Rightarrow$ sehr hohe Taktfrequenzen



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

Pentium 4 / NetBurst Architektur (cont.)

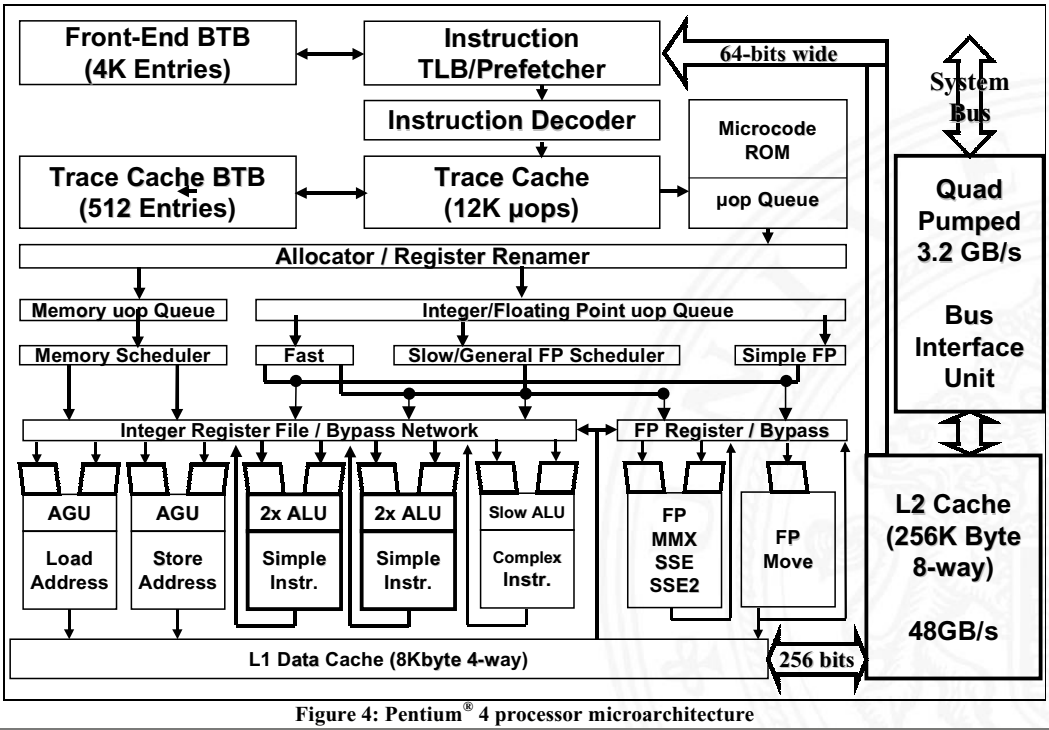
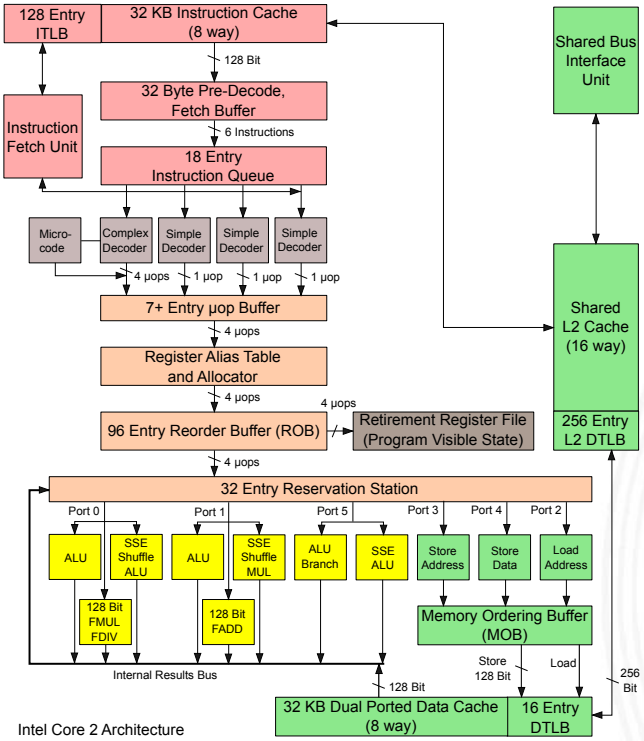


Figure 4: Pentium® 4 processor microarchitecture

Intel
Q1, 2001

Core 2 Architektur



Intel Core 2 Architecture

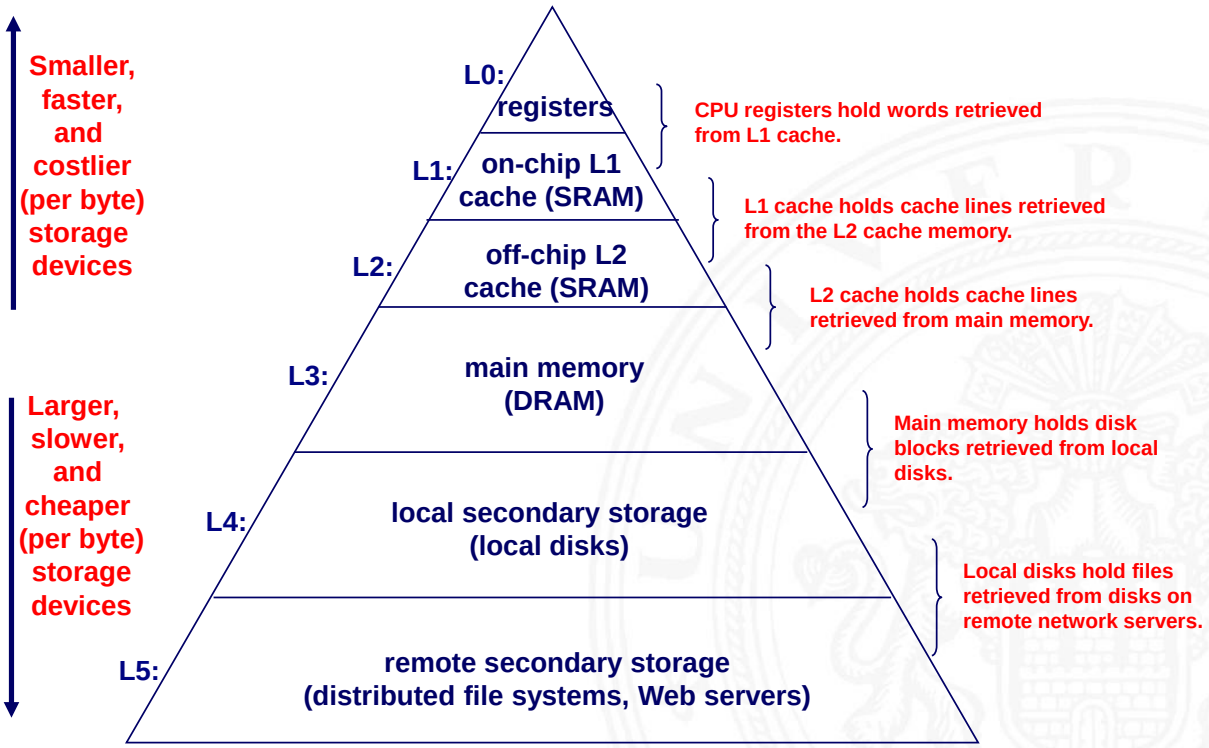
Gliederung

Gliederung (cont.)

Gliederung (cont.)

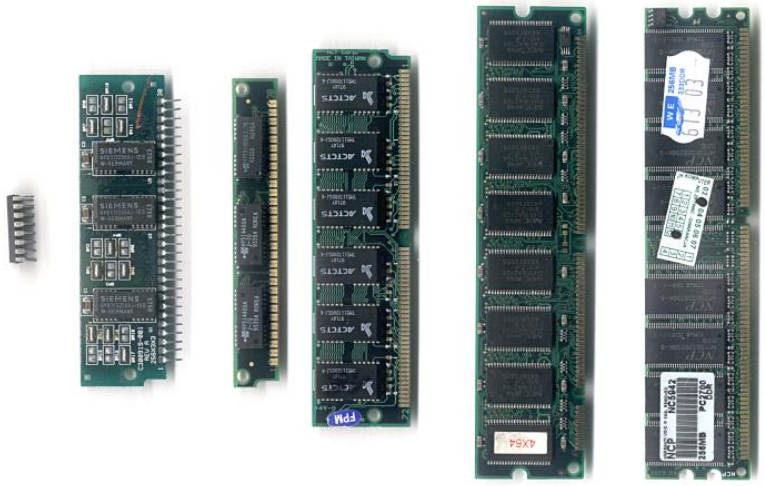
Cache Speicher
Virtueller Speicher
Beispiel: Pentium und Linux

Speicherhierarchie: Konzept



Random-Access Memory / RAM

- RAM ist als Chip gepackt
- Grundspeichereinheit ist eine Zelle (ein Bit pro Zelle)
- Viele RAM Chips bilden einen Speicher



Random-Access Memory / RAM (cont.)

Statischer RAM (SRAM)

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

Dynamischer RAM (DRAM)

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

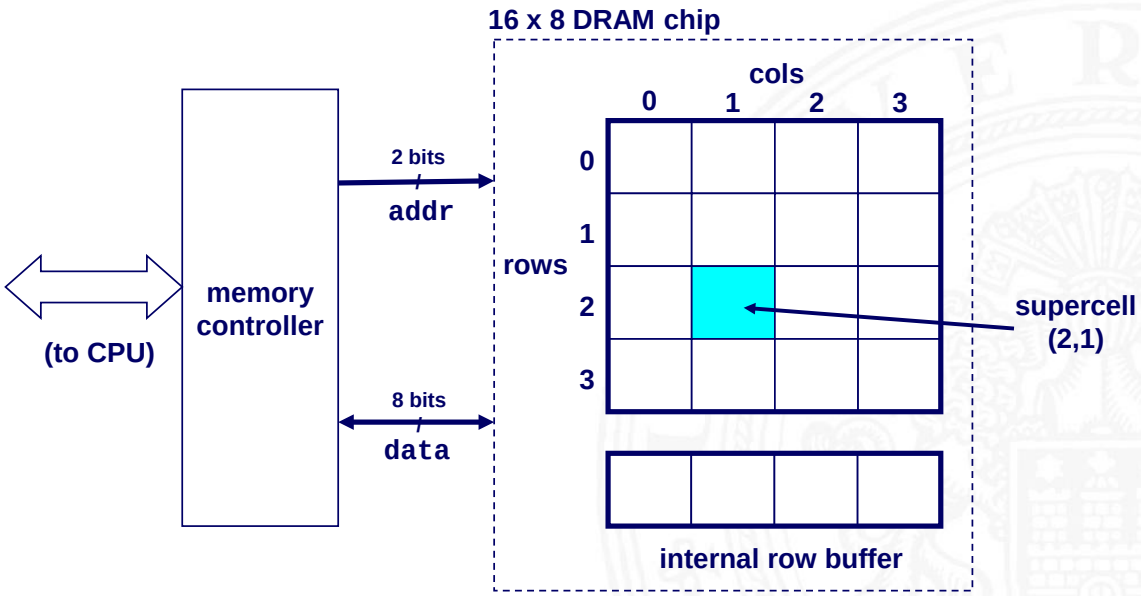
SRAM vs. DRAM

	SRAM	DRAM
Zugriffszeit	5...50 ns	60...100 ns t_{rac}
		20...300 ns t_{cac}
		110...180 ns t_{cyc}
Leistungsaufnahme	200...1300 mW	300...600 mW
Speicherkapazität	$<$ 72 Mbit	$<$ 4 Gbit
Preis	10 €/Mbit	0,2 Ct./Mbit

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

DRAM Organisation

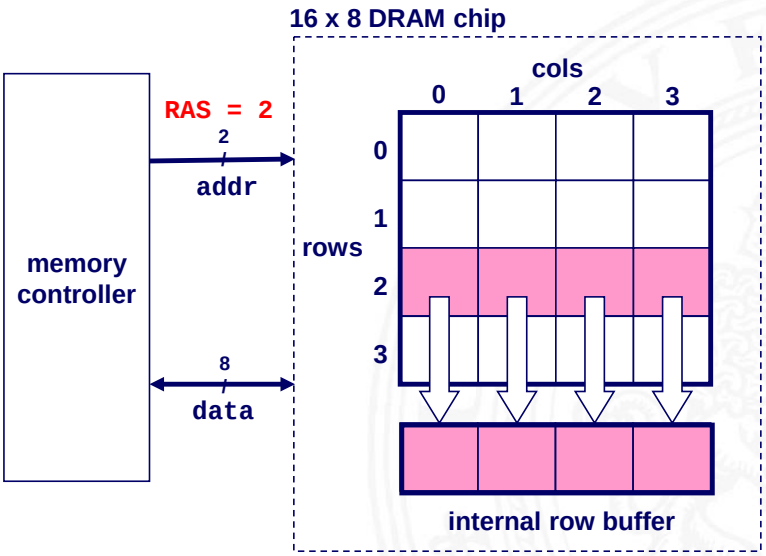
- (d × w) DRAM: organisiert als d-Superzellen mit w-bits



Lesen der DRAM Zelle (2,1)

1.a „Row Access Strobe“ (RAS) wählt Zeile 2

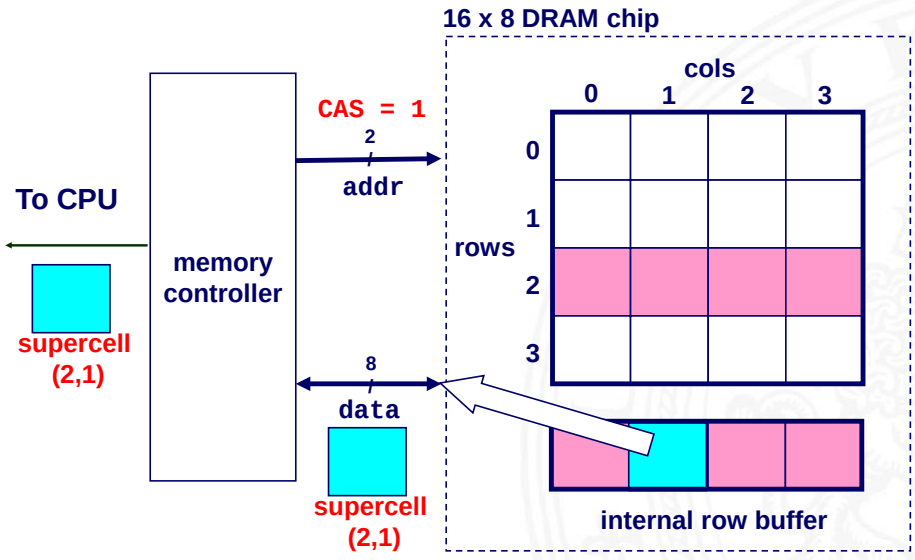
1.b Zeile aus DRAM Array in Zeilenpuffer („Row Buffer“) kopieren



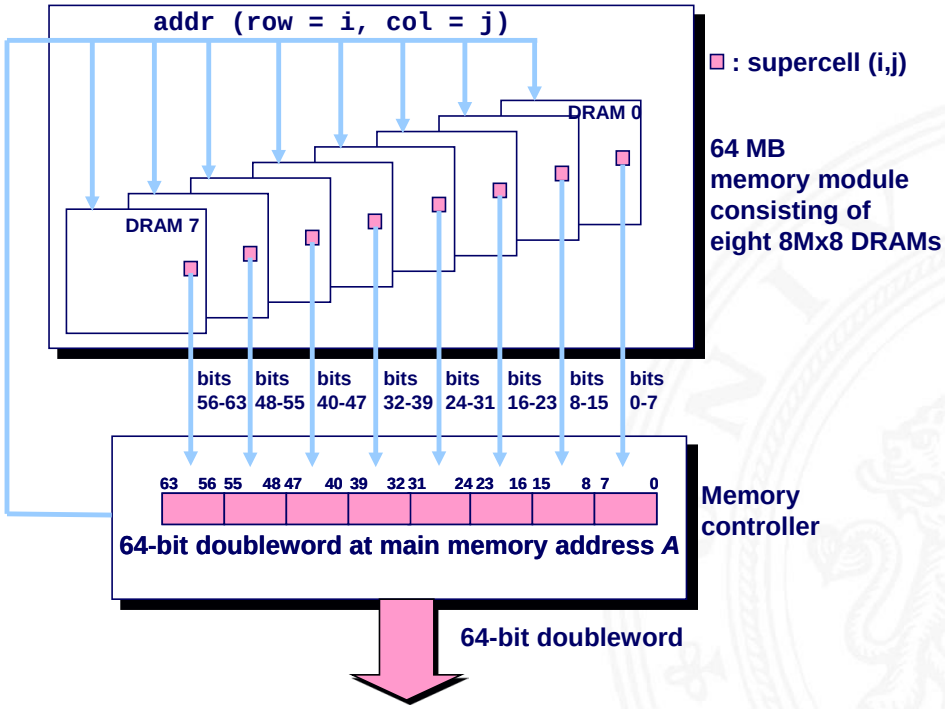
Lesen der DRAM Zelle (2,1) (cont.)

2.a „Column Access Strobe“ (CAS) wählt Spalte 1

2.b Superzelle (2,1) aus Buffer lesen und auf Datenleitungen legen



Speichermodule



Nichtflüchtige Speicher

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

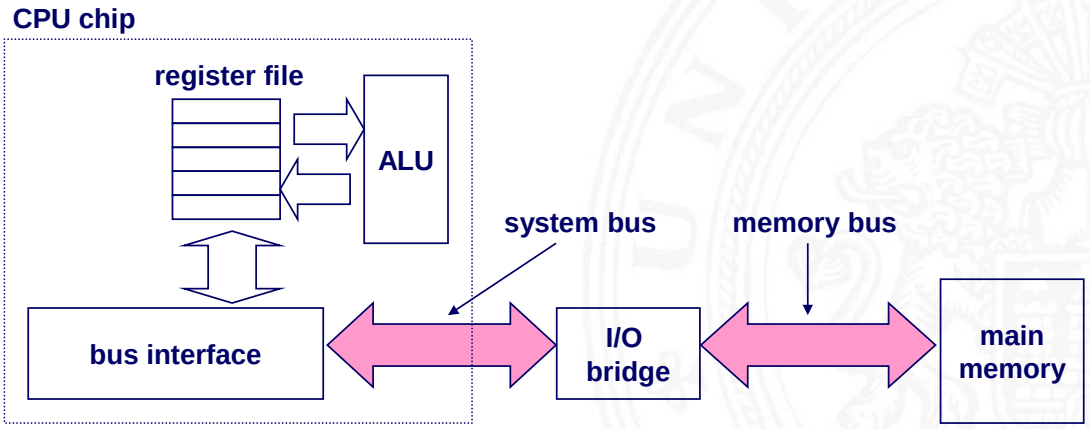
Nichtflüchtige Speicher (cont.)

Anwendungsbeispiel: nichtflüchtige Speicher

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

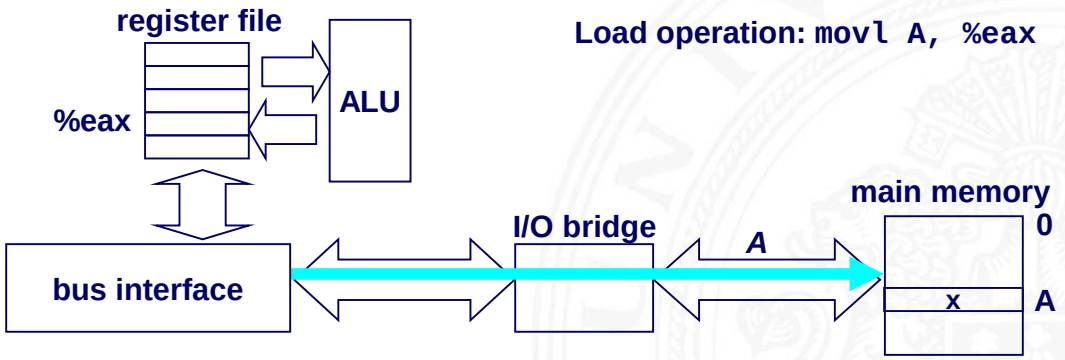
Bussysteme verbinden CPU und Speicher

- Busse
  - Bündel paralleler Leitungen
  - es gibt mehr als einen Treiber ⇒ Tristate-Treiber
- Busse im Rechner
  - zur Übertragung von Adressen, Daten und Kontrollsignalen
  - werden üblicherweise von mehreren Geräten genutzt



lesender Speicherzugriff

1. CPU legt Adresse A auf den Speicherbus

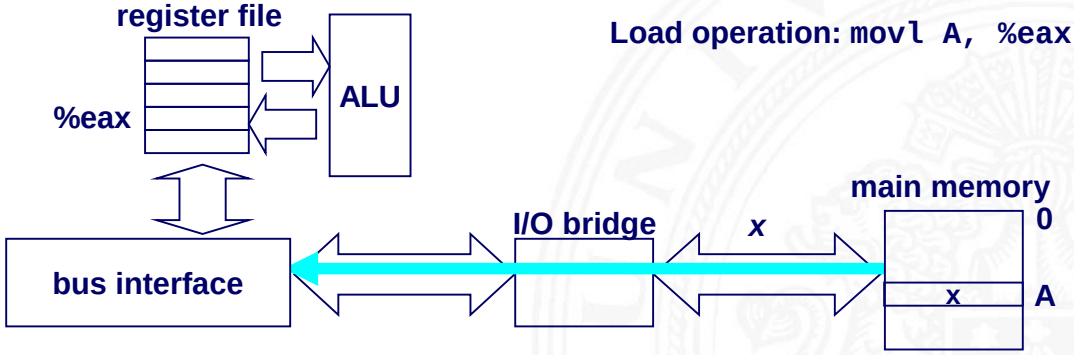


lesender Speicherzugriff (cont.)

2.a Hauptspeicher liest Adresse A vom Speicherbus

2.b –"– ruft das Wort x unter der Adresse A ab

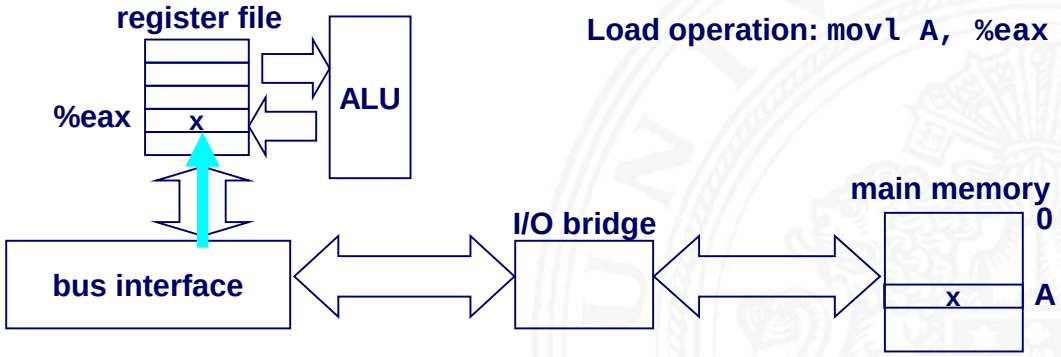
2.c –"– legt das Wort x auf den Bus



lesender Speicherzugriff (cont.)

3.a CPU liest Wort x vom Bus

3.b –"– kopiert Wert x in Register `%eax`

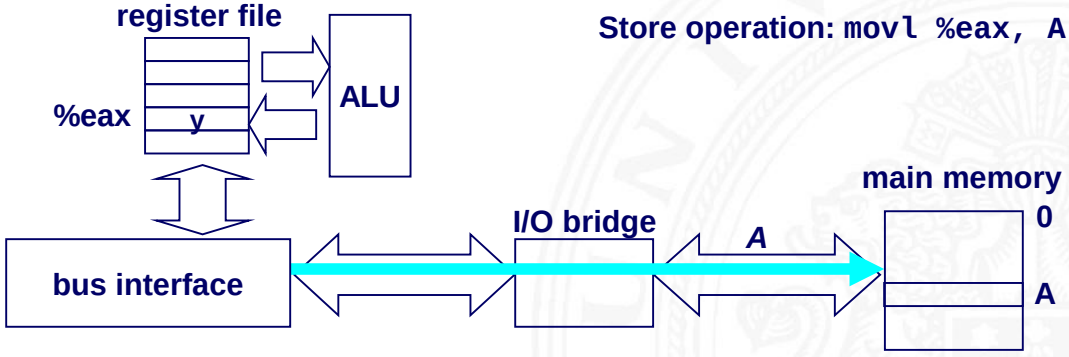


schreibender Speicherzugriff

1 CPU legt die Adresse A auf den Bus

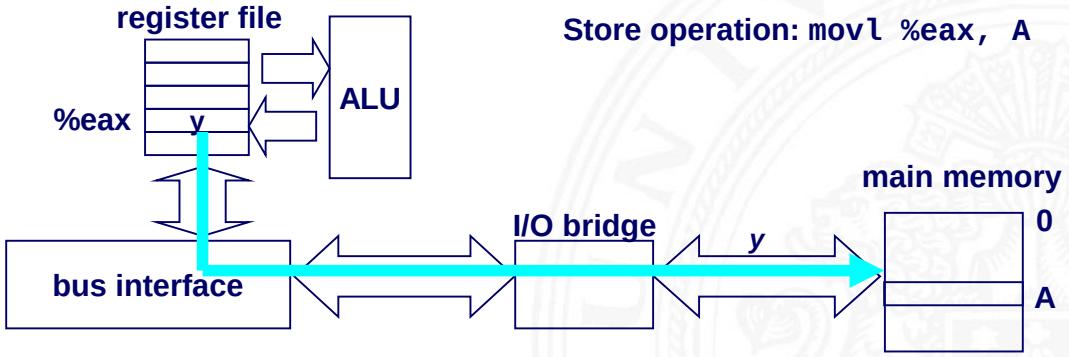
2.b Hauptspeicher liest Adresse

2.c –"– wartet auf Ankunft des Datenworts



schreibender Speicherzugriff (cont.)

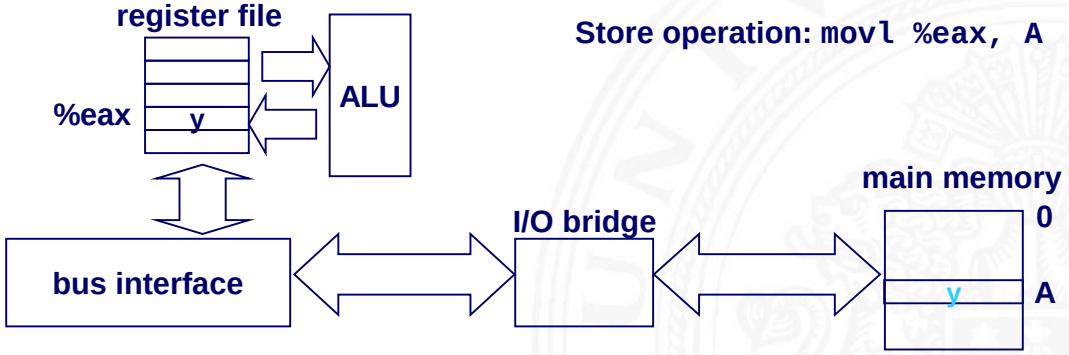
3. CPU legt Datenwort y auf den Bus



schreibender Speicherzugriff (cont.)

4.a Hauptspeicher liest Datenwort y vom Bus

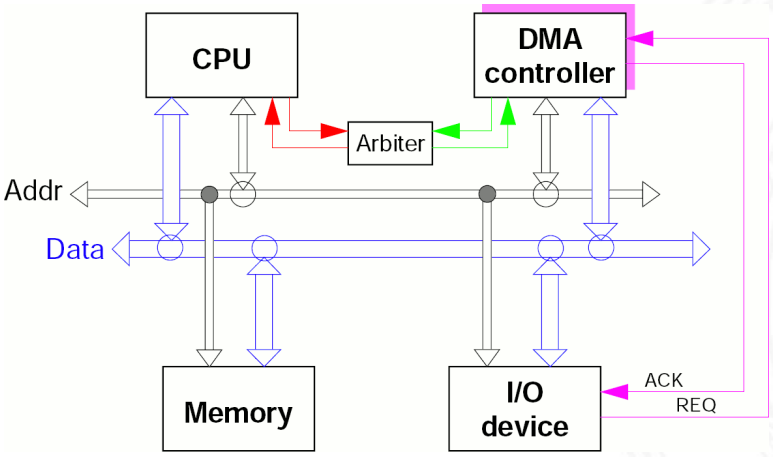
4.b –"– speichert Datenwort y unter Adresse A



Speicheranbindung – DMA

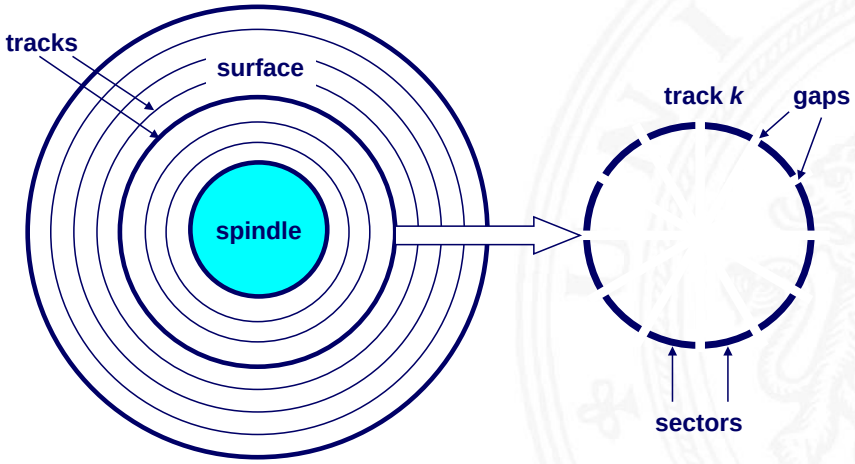
DMA – **D**irect **M**emory **A**ccess

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



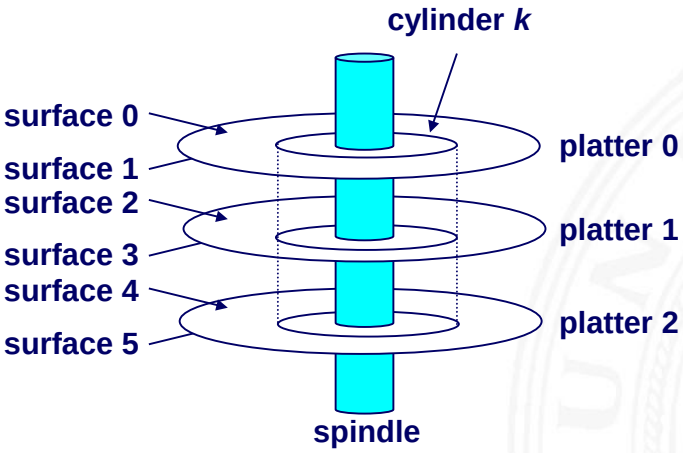
Festplattengeometrie

- Platten mit jeweils zwei Oberflächen („surfaces“)
- konzentrische Ringe der Oberfläche bilden Spuren („tracks“)
- Spur unterteilt in Sektoren („sectors“),
  durch Lücken (“gaps”) getrennt



Festplattengeometrie (cont.)

- untereinander liegende Spuren (mehrerer Platten) bilden einen Zylinder



Festplattenkapazität

- Kapazität: Höchstzahl speicherbarer Bits
- bestimmende technologische Faktoren
  - Aufnahmedichte [bits/in]: # Bits / 1-Inch Segment einer Spur
  - Spurdichte [tracks/in]: # Spuren / 1-Inch (radial)
  - Flächendichte [bits/in^2]: Aufnahme- $\times$ Spurdichte
- Spuren unterteilt in getrennte Zonen („Recording Zones“)
  - jede Spur einer Zone hat gleichviel Sektoren (festgelegt durch die Ausdehnung der innersten Spur)
  - jede Zone hat unterschiedlich viele Sektoren/Spuren

Festplattenkapazität (cont.)

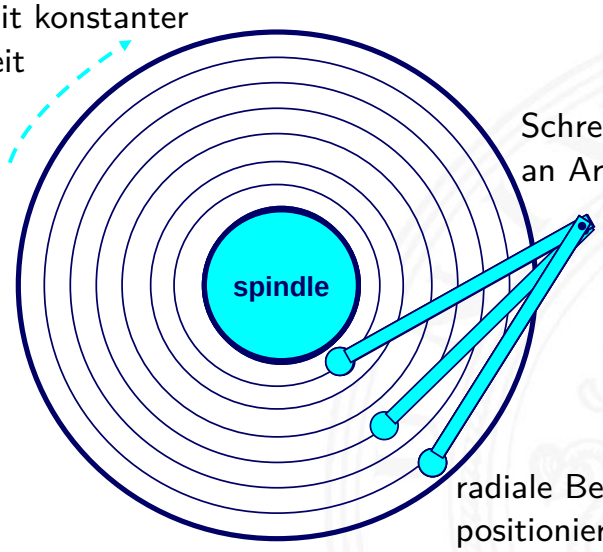
- Kapazität = Bytes/Sektor × ∅ Sektoren/Spur × Spuren/Oberfläche × Oberflächen/Platten × Platten/Festplatte
- Beispiel
  - 512 Bytes/Sektor
  - 300 Sektoren/Spuren (im Durchschnitt)
  - 20 000 Spuren/Oberfläche
  - 2 Oberflächen/Platten
  - 5 Platten/Festplatte

  ⇒ Kapazität = 512 × 300 × 20 000 × 2 × 5
  = 30 720 000 000 = 30,72 GB

Festplatten-Operation

- Ansicht einer Platte

Umdrehung mit konstanter Geschwindigkeit



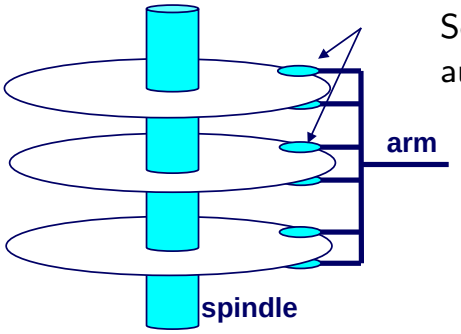
Schreib/Lese Kopf ist an Arm befestigt

„fliegt“ auf Luftpolster über Plattenoberfläche

radiale Bewegung des Arms positioniert Kopf auf Spuren

Festplatten-Operation (cont.)

- Ansicht mehrerer Platten



Schreib/Lese Köpfe werden gemeinsam auf Zylindern positioniert

Festplatten-Zugriffszeit

Durchschnittliche (avg) Zugriffszeit auf einen Zielsektor wird angenähert durch

- $T_{Zugriff} = T_{avgSuche} + T_{avgRotationslatenz} + T_{avgTransfer}$

Suchzeit ($T_{avgSuche}$)

- Zeit in der Schreib-Lese Köpfe („heads“) über den Zylinder mit dem Targetsektor positioniert werden
- üblicherweise $T_{avgSuche} = 8\,\text{ms}$

Festplatten-Zugriffszeit (cont.)

Rotationslatenzzeit ($T_{avgRotationslatenz}$)

- Wartezeit, bis das erste Bit des Targetsektors unter dem Schreib-Lese-Kopf durchrotiert
- $T_{avgRotationslatenz} = 1/2 \times 1/\text{RPMs} \times 60\,\text{Sek}/1\,\text{Min}$

Transferzeit ($T_{avgTransfer}$)

- Zeit, in der die Bits des Targetsektors gelesen werden
- $T_{avgTransfer} = 1/\text{RPM} \times 1/(\text{Durchschn. } \#\,\text{Sektoren/Spur}) \times 60\,\text{Sek}/1\,\text{Min}$

Festplatten-Zugriffszeit (cont.)

Beispiel für Festplatten-Zugriffszeit

- Umdrehungszahl = 7 200 RPM („Rotations per Minute“)
- Durchschnittliche Suchzeit = 8 ms
- Avg # Sektoren/Spur = 400

$\Rightarrow$ $T_{avgRotationslatenz}$
$= 1/2 \times (60\,\text{Sek}/7\,200\,\text{RPM}) \times 1\,000\,\text{ms/Sek} = 4\,\text{ms}$

$\Rightarrow$ $T_{avgTransfer}$
$= 60/7\,200\,\text{RPM} \times 1/400\,\text{Sek/Spur} \times 1\,000\,\text{ms/Sek} = 0{,}02\,\text{ms}$

$\Rightarrow$ $T_{avgZugriff}$
$= 8\,\text{ms} + 4\,\text{ms} + 0{,}02\,\text{ms}$

Festplatten-Zugriffszeit (cont.)

Fazit

- Zugriffszeit wird von Suchzeit und Rotationslatenzzeit dominiert
- erstes Bit eines Sektors ist das „teuerste“, der Rest ist quasi umsonst
- SRAM Zugriffszeit ist ca. 4 ns DRAM ca. 60 ns
- Kombination aus Zugriffszeit und Datentransfer
  - Festplatte ist ca. 40 000 mal langsamer als SRAM
  - 2 500 mal langsamer als DRAM

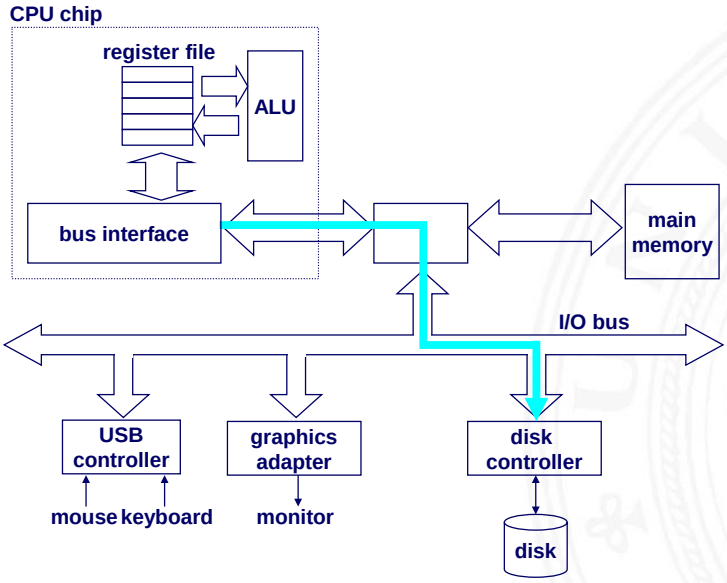
Logische Festplattenblöcke

- simple, abstrakte Ansicht der komplexen Sektorengeometrie
  - verfügbare Sektoren werden als Sequenz logischer Blöcke der Größe b modelliert (0,1,2,..,n)
- Abbildung der logischen Blöcke auf die tatsächlichen (physikalischen) Sektoren
  - durch Hard-/Firmware Einheit (Festplattencontroller)
  - konvertiert logische Blöcke zu Tripeln (Oberfläche, Spur, Sektor)
- Controller kann für jede Zone Ersatzzylinder bereitstellen

  $\Rightarrow$ Unterschied: „formatierte Kapazität“ und „maximale Kapazität“

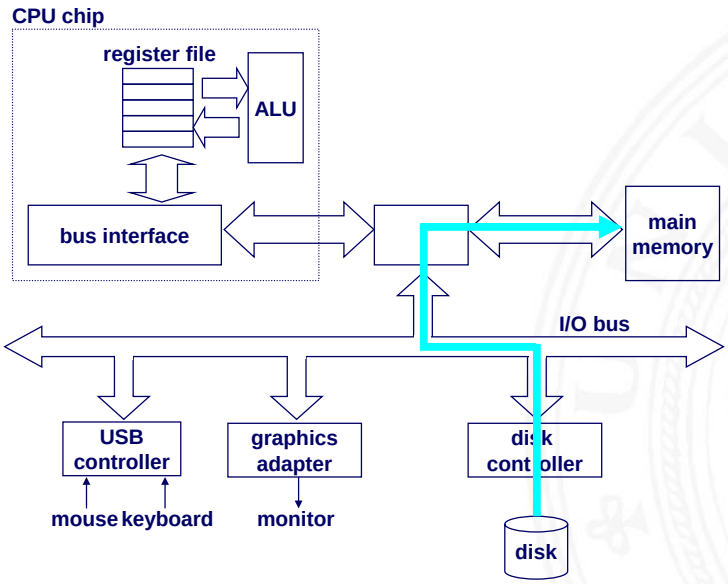
Lesen eines Festplattensektors

1. CPU initiiert Lesevorgang von Festplatte
   auf Port (Adresse) des Festplattencontrollers wird geschrieben
   - Befehl, logische Blocknummer, Zielspeicheradresse



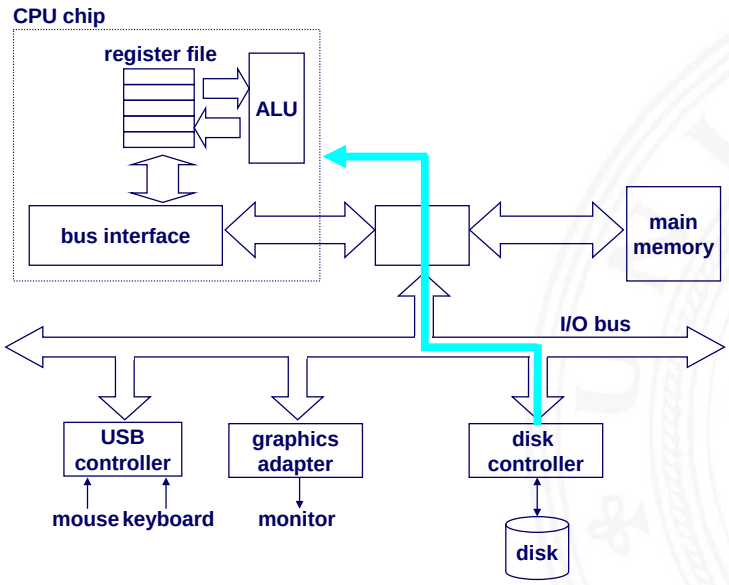
Lesen eines Festplattensektors (cont.)

2. Festplattencontroller liest den Sektor aus
3. –"– führt DMA-Zugriff auf Hauptspeicher aus



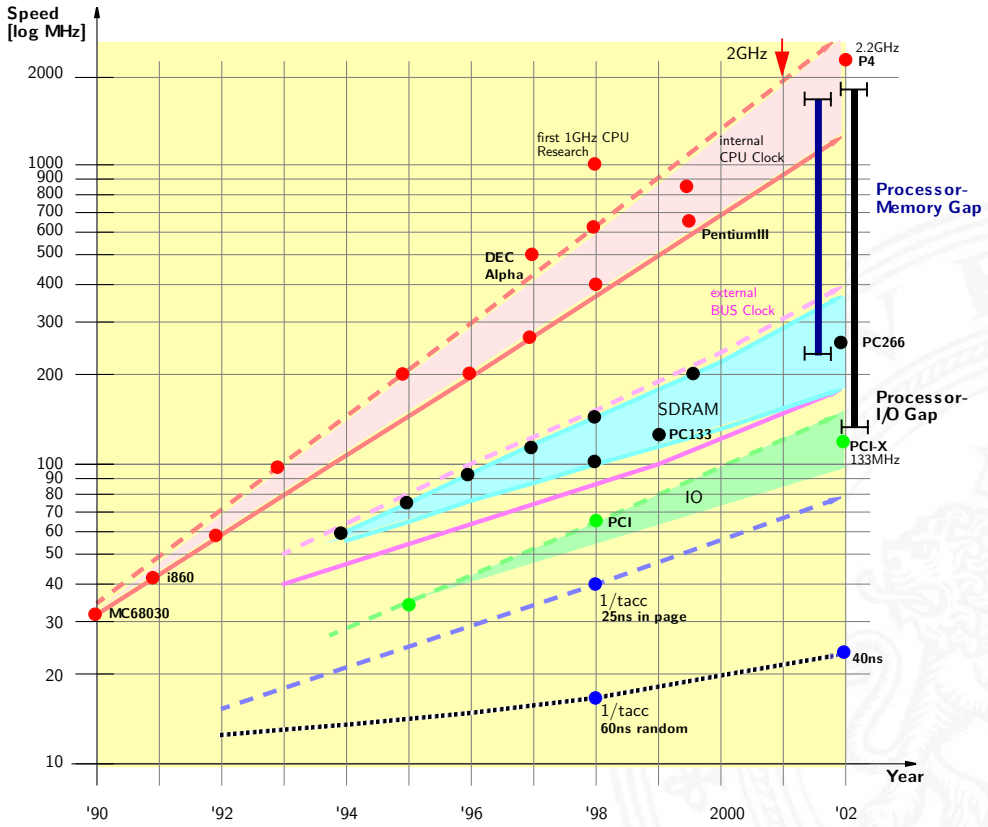
Lesen eines Festplattensektors (cont.)

4. Festplattencontroller löst Interrupt aus



Eigenschaften der Speichertypen

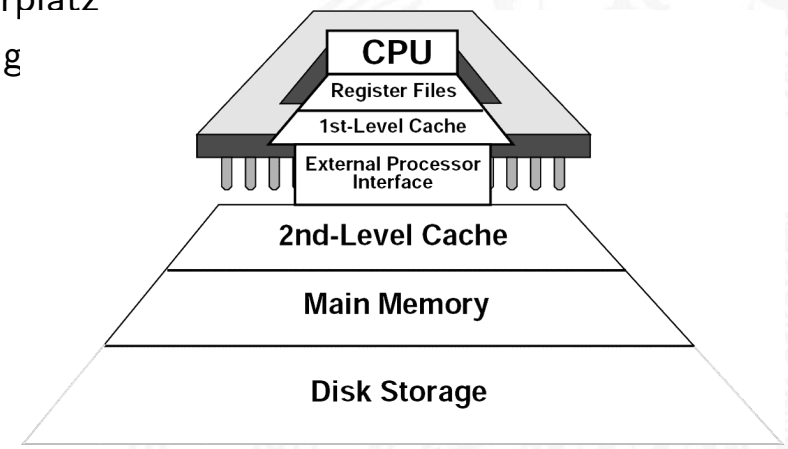
Speicher	Vorteile	Nachteile
Register	sehr schnell	sehr teuer
SRAM	schnell	teuer, große Chips
DRAM	hohe Integration	Refresh nötig, langsam
Platten	billig, Kapazität	sehr langsam, mechanisch
Beispiel	Hauptspeicher	Festplatte
Latenz	8 ns	4 ms
Bandbreite	$\approx$ 38,4 GByte/sec (triple Channel)	$\approx$ 750 MByte/sec (typisch $<$ 300)
Kosten	1 GByte, 5 €	1 TByte, 40 € (4 Ct./GByte)



Speicherhierarchie

Motivation

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



Speicherhierarchie (cont.)

Fundamentale Eigenschaften von Hard- und Software

- schnelle vs. langsame Speichertechnologie
  schnell : hohe Kosten/Byte geringe Kapazität
  langsam : geringe –"– hohe –"–
- Abstand zwischen CPU und Hauptspeichergeschwindigkeit vergrößert sich
- Lokalität der Programme wichtig
  - kleiner Adressraum im Programmkontext
  - gut geschriebene Programme haben meist eine gute Lokalität

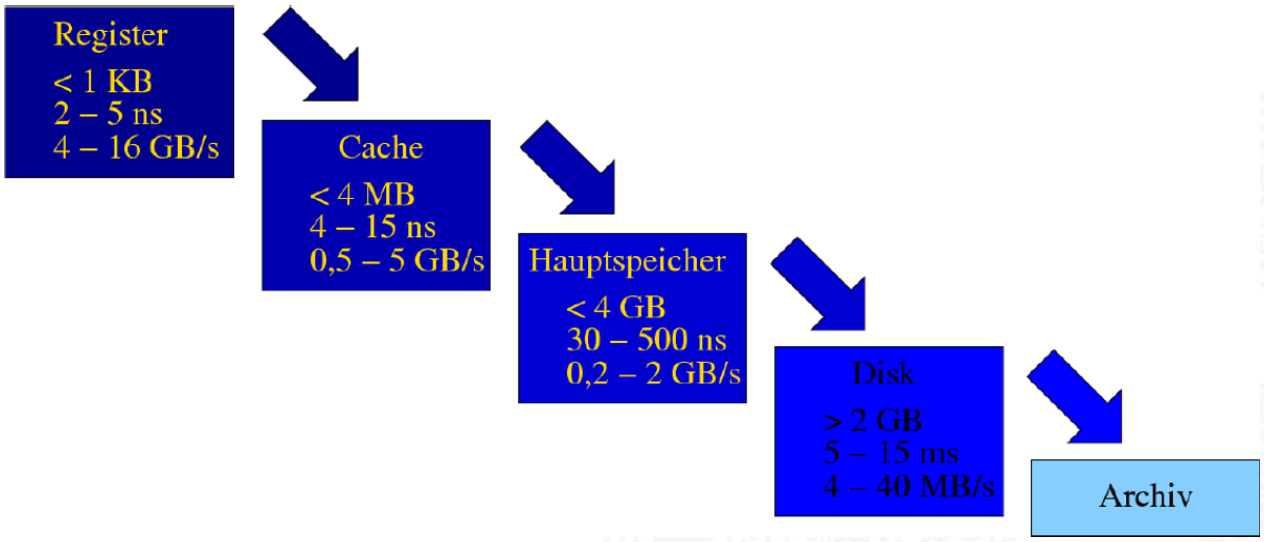
$\Rightarrow$ Motivation für spezielle Organisation von Speichersystemen **Speicherhierarchie**

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
21.2 Speicherhierarchie - Motivation
64-040 Rechnerstrukturen

Speicherhierarchie (cont.)

Cache Type	What Cached	Where Cached	Latency (cycles)	Managed By
Registers	4-byte word	CPU registers	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	32-byte block	On-Chip L1	1	Hardware
L2 cache	32-byte block	Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware+ OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Verwaltung der Speicherhierarchie



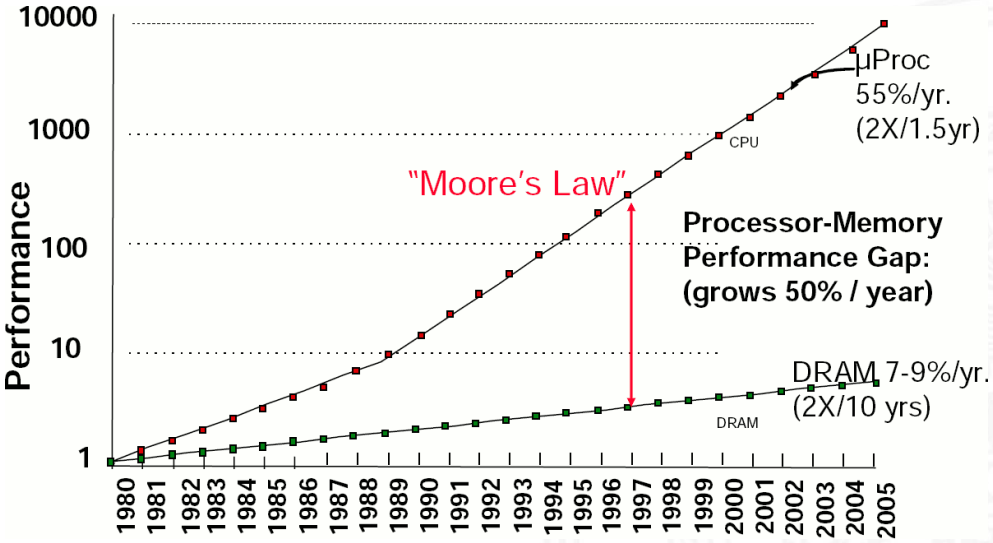
Verwaltung der Speicherhierarchie (cont.)

Verwaltung der Speicherhierarchie

- Register ↔ Memory
  - Compiler
  - Assembler-Programmierer
- Cache ↔ Memory
  - Hardware
- Memory ↔ Disk
  - Hardware und Betriebssystem (Paging)
  - Programmierer (Files)

Cache

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

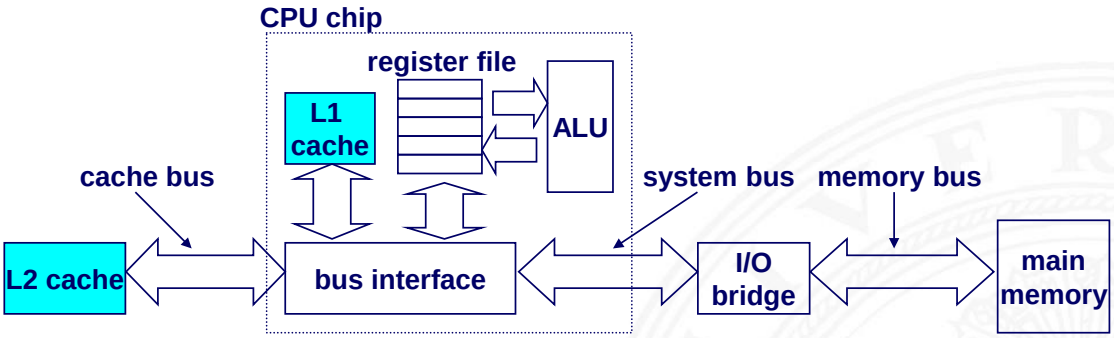


Cache (cont.)

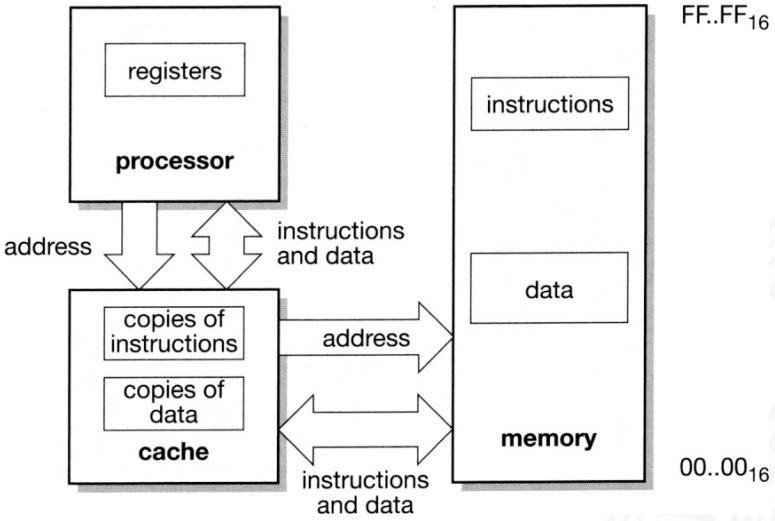
⇒ Cache als schneller Zwischenspeicher zum Hauptspeicher

- technische Realisierung: SRAM
- transparenter Speicher
  - Cache ist für den Programmierer nicht sichtbar!
  - wird durch Hardware verwaltet
- `http://de.wikipedia.org/wiki/Cache`
  `http://en.wikipedia.org/wiki/Cache`
- enthält Hauptspeicherblöcke mit erhöhter Zugriffswahrscheinlichkeit
- CPU referenziert Adresse
  - parallele Suche in L1 (level 1), L2... und Hauptspeicher
  - erfolgreiche Suche liefert Datum, Abbruch laufender Suchen

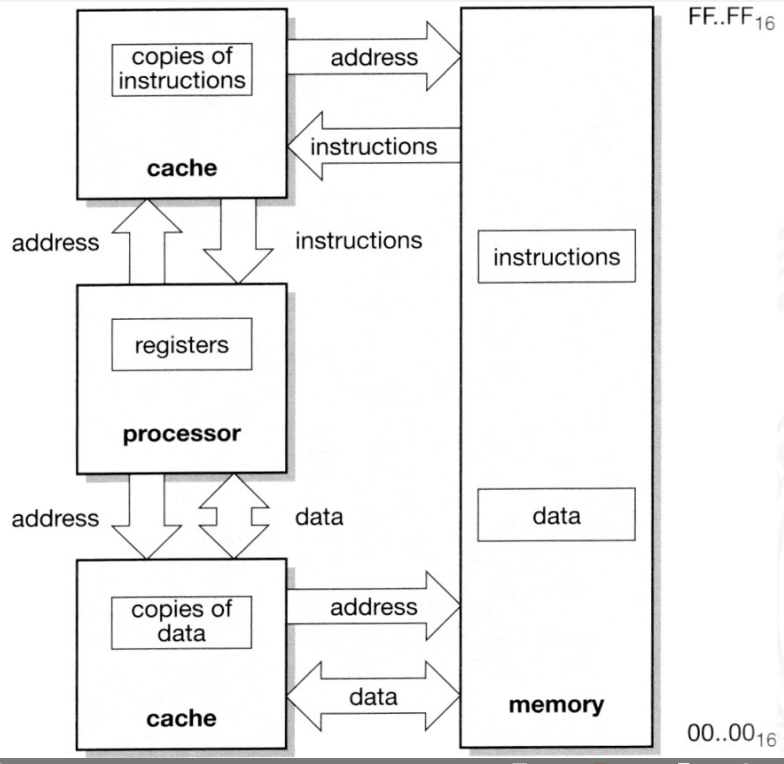
Cache (cont.)



gemeinsamer Cache / „unified Cache“

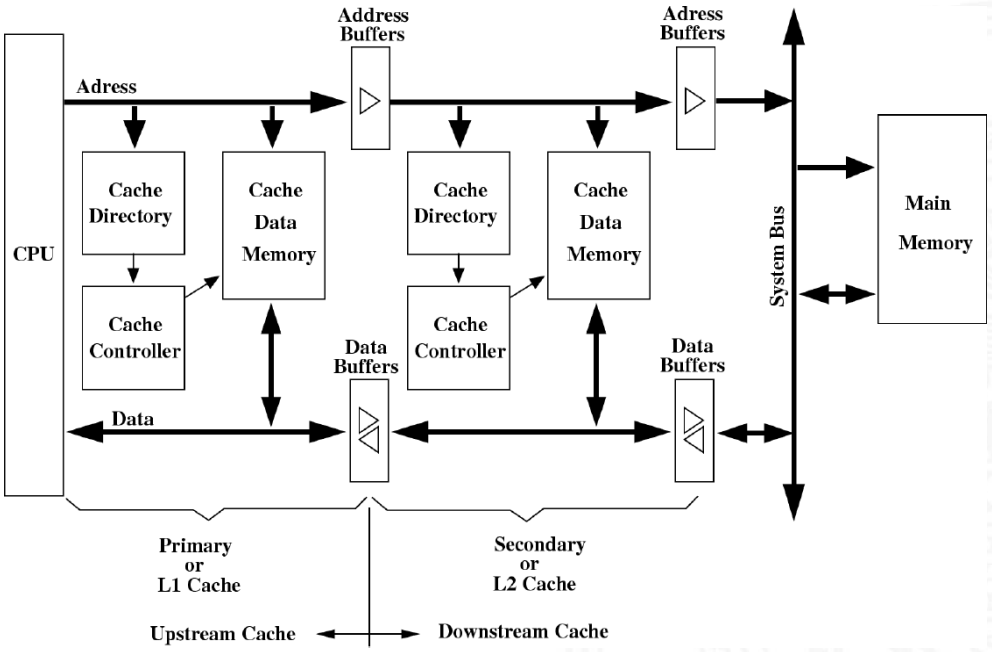


separate Instruction-/Data Caches



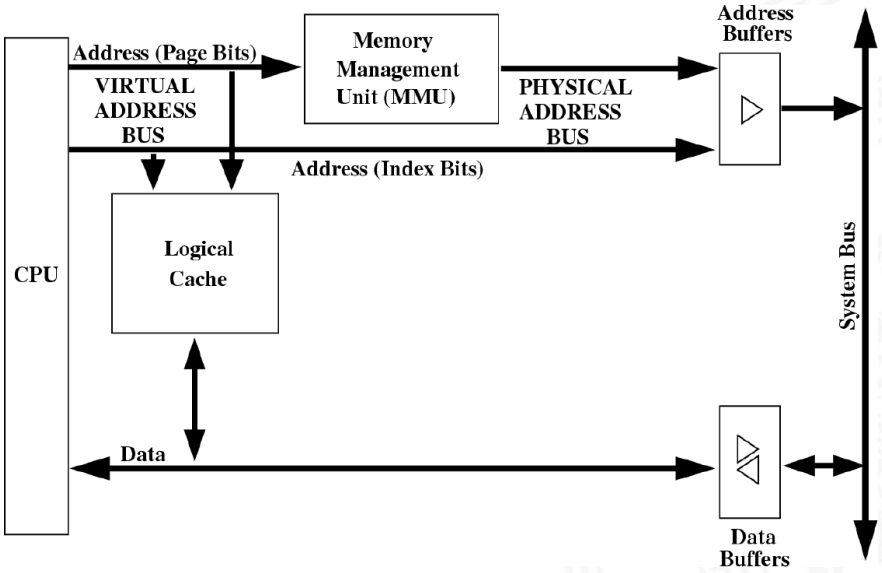
Cache – Position

- First- und Second-Level Cache



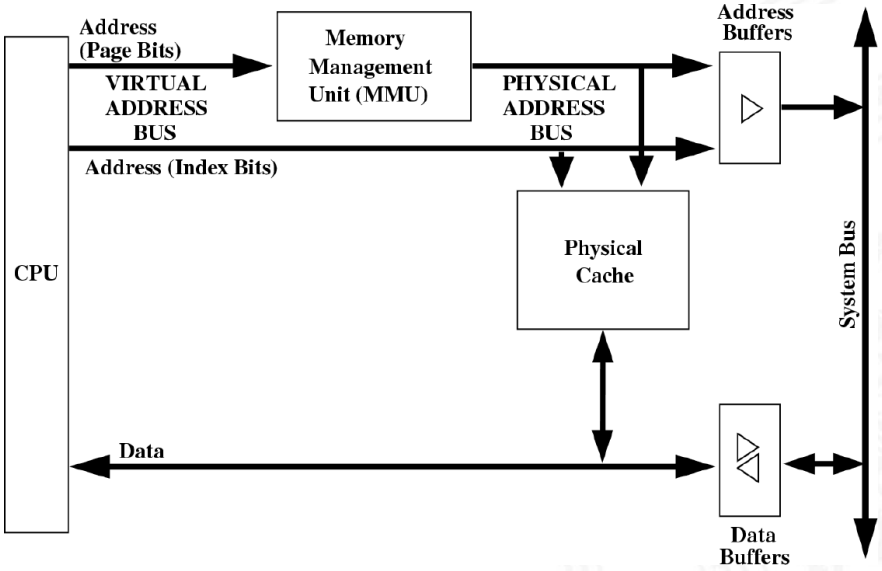
Cache – Position (cont.)

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



Cache – Position (cont.)

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



Cache – Position (cont.)

- typische Cache Organisation
  - First-Level Cache: getrennte Instruktions- und Daten-Caches
  - Second-Level Cache: gemeinsamer Cache je Prozessorkern
  - Third-Level Cache: gemeinsamer Cache für alle Prozessorkerne
- bei mehreren Prozessoren / Prozessorkernen ⇒ Cache-Kohärenz wichtig
  - gemeinsam genutzte Daten konsistent halten (s.u.)

Cache – Strategie

Cachestrategie: *Welche Daten sollen in den Cache?*
Diejenigen, die bald wieder benötigt werden!

- *temporale Lokalität*:
  die Daten, die zuletzt häufig gebraucht wurden
- *räumliche Lokalität*:
  die Daten, die nahe den zuletzt gebrauchten liegen
- verschiedene Platzierungs-, Ersetzungs- und Rückschreibestategien für den Cache

Cache – Performanz

Cacheperformanz

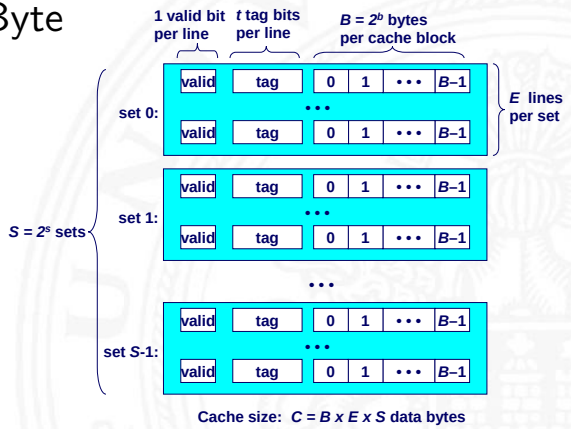
- Begriffe

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

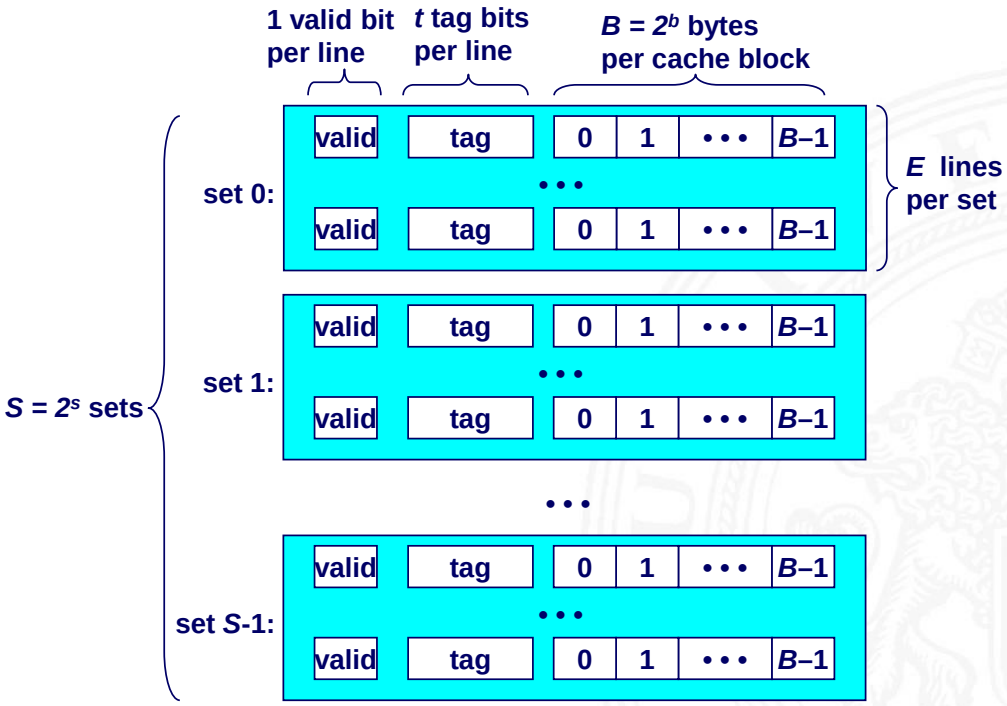
- Mittlere Speicherzugriffszeit $= T_{Hit} + R_{Miss} \cdot T_{Miss}$
- Beispiel
  $T_{Hit} = 1\ Takt,\ T_{Miss} = 20\ Takte,\ R_{Miss} = 5\,\%$
  Mittlere Speicherzugriffszeit $= 2\ Takte$

Cache Organisation

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

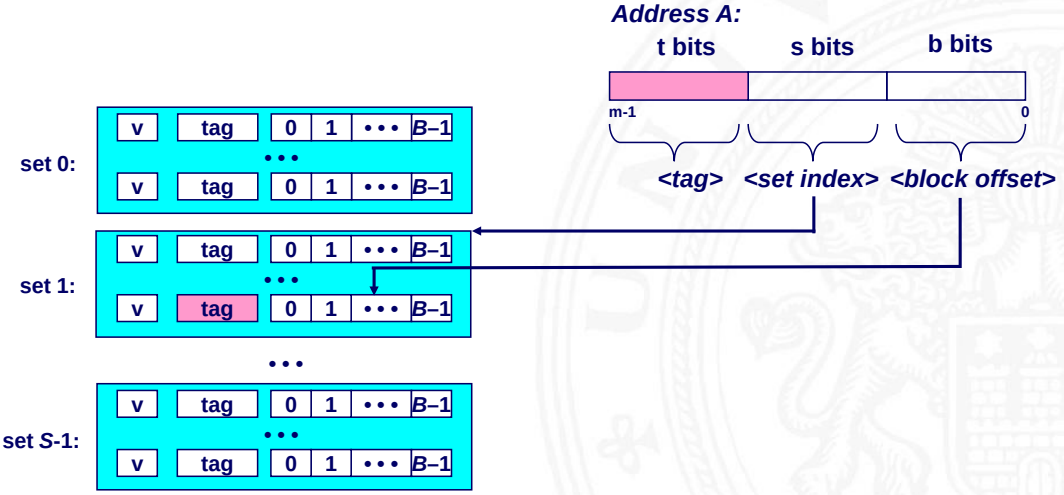


Cache Organisation (cont.)



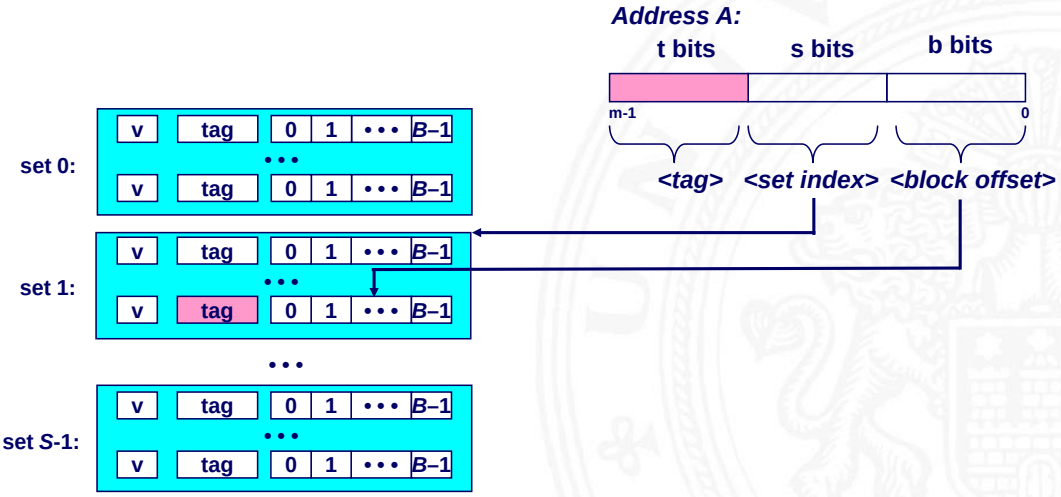
Adressierung von Caches

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



Adressierung von Caches (cont.)

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



Cache – Organisation

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

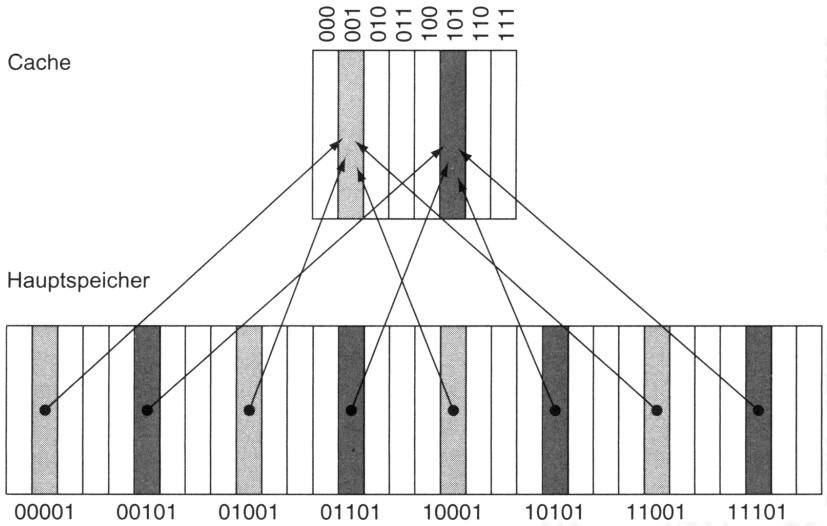
  direkt abgebildet / direct mapped jeder Speicheradresse ist genau eine Cache-Speicherzelle zugeordnet

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

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

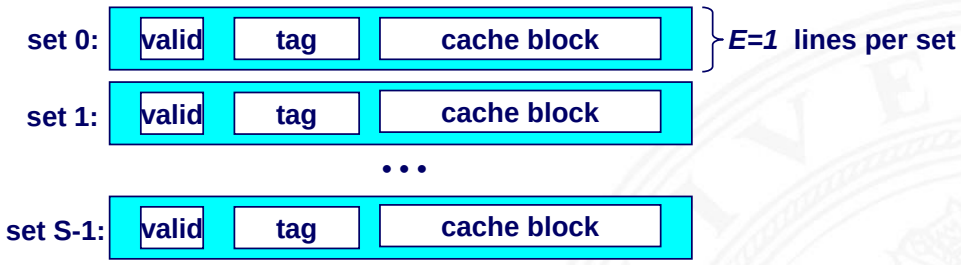
Cache: direkt abgebildet / „direct mapped“

- jeder Adresse ist genau eine Speicherzelle im Cache zugeordnet



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

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



\+ einfachste Cache-Art

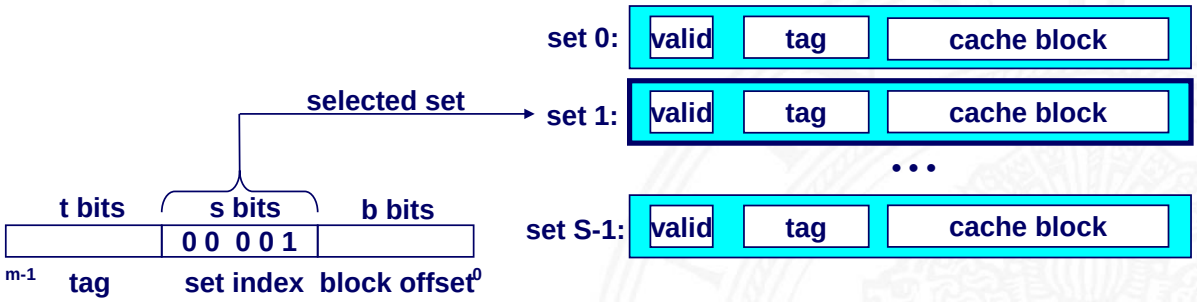
\+ große Caches möglich

– Effizienz, z.B. Zugriffe auf A, $A + n \cdot S \ldots$

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

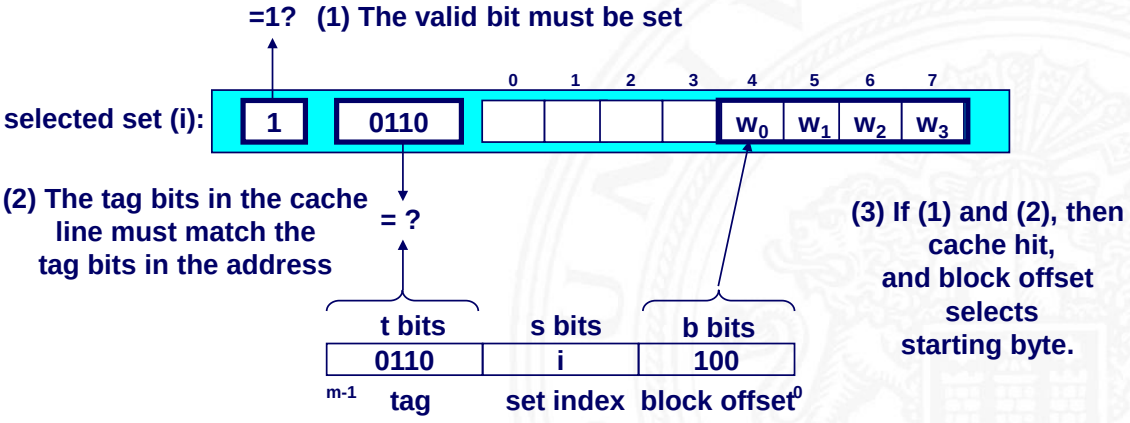
Zugriff auf direkt abgebildete Caches

1. Bereichsauswahl durch Bits ⟨*set index*⟩



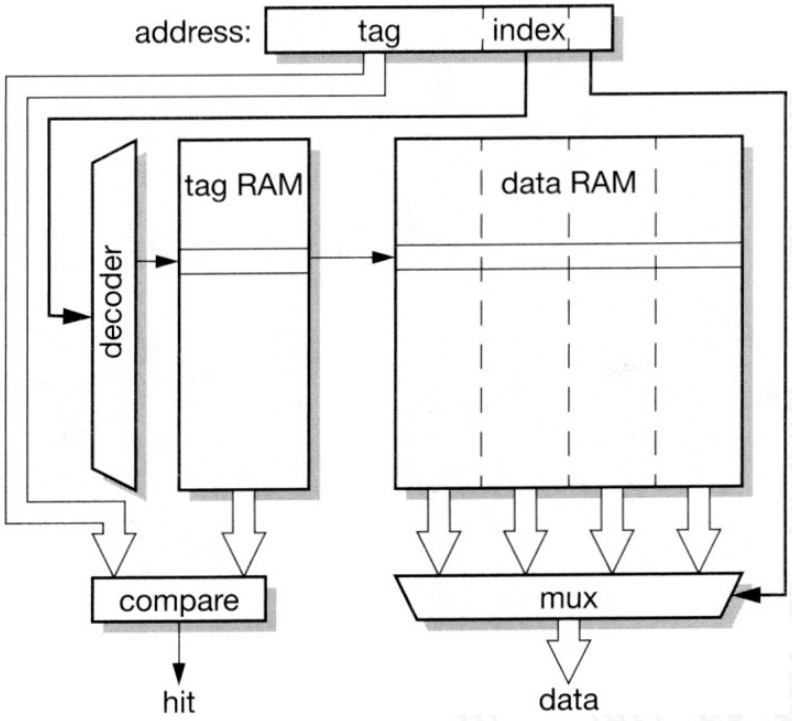
Cache: direkt abgebildet / „direct mapped“ (cont.)

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



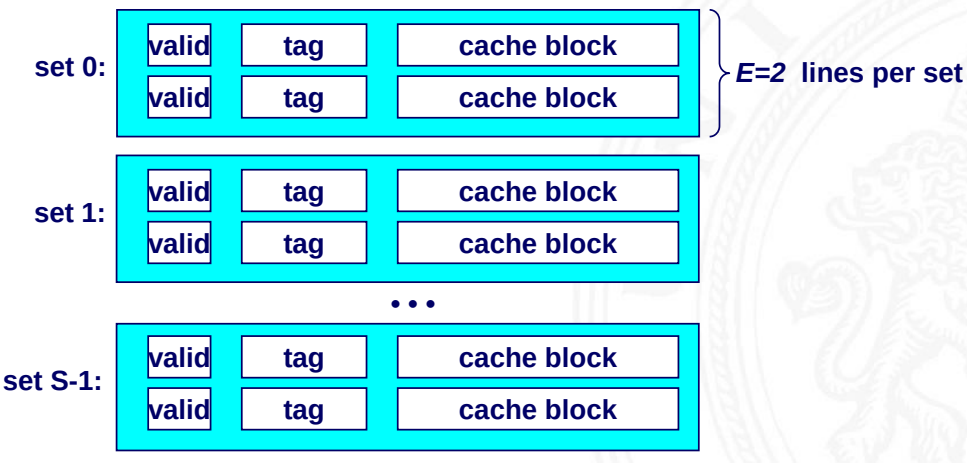
Cache: direkt abgebildet / „direct mapped“ (cont.)

Prinzip



Cache: bereichsassoziativ / „set associative“

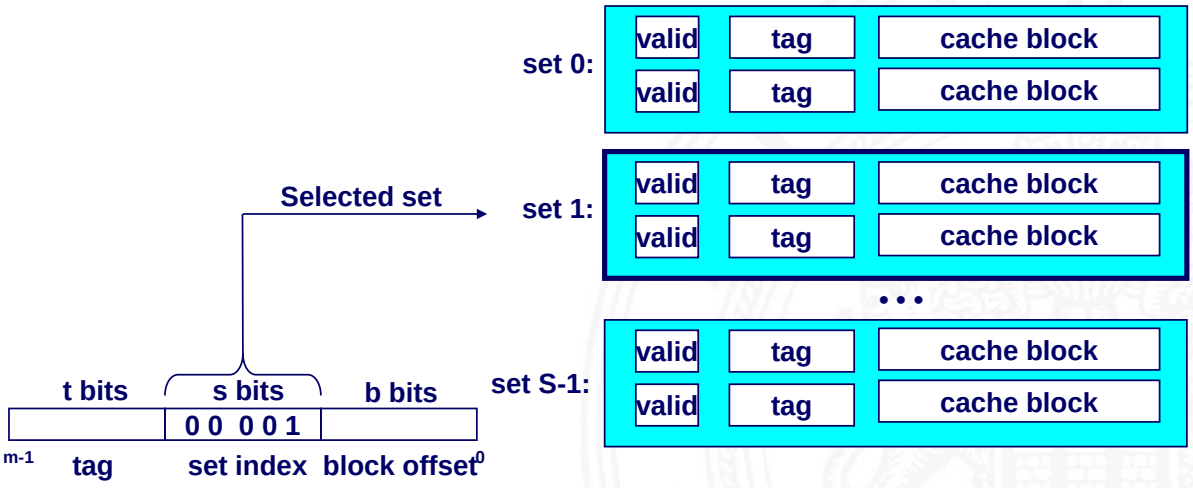
- jeder Speicheradresse ist ein Bereich S mit mehreren (E) Cachezeilen zugeordnet
- n-fach assoziative Caches: E=2, 4...
  „2-way set associative cache“, „4-way...“



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

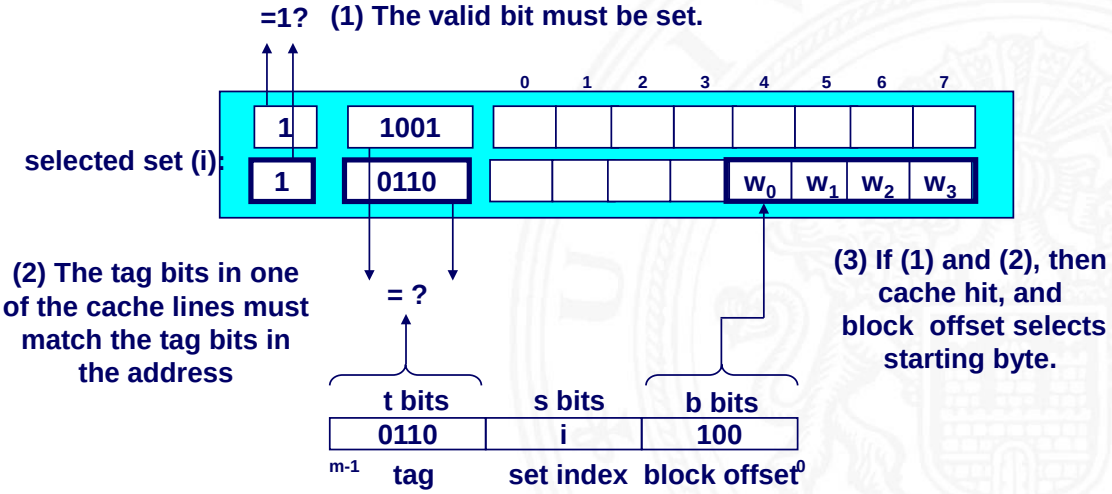
Zugriff auf n-fach assoziative Caches

1. Bereichsauswahl durch Bits ⟨*set index*⟩



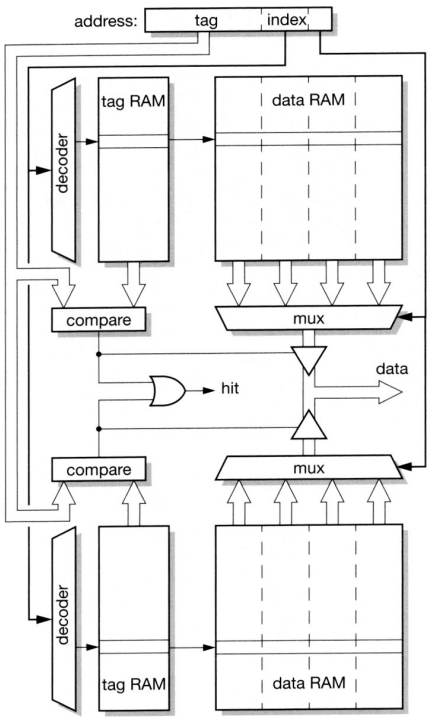
Cache: bereichsassoziativ / „set associative“ (cont.)

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



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

Prinzip

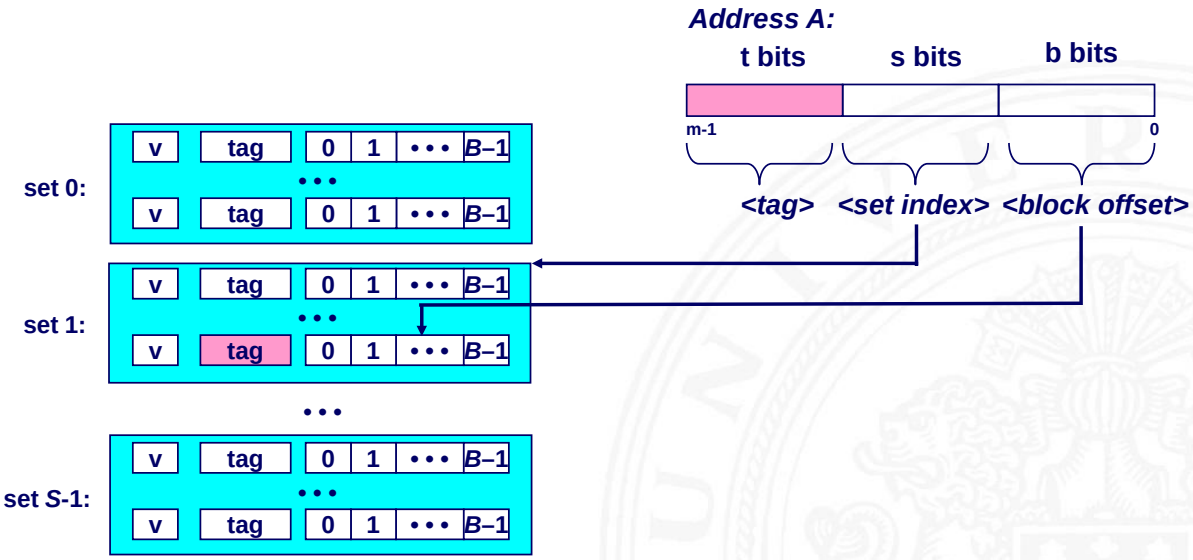


Cache: voll-assoziativ

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

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

Cache – Dimensionierung



- Parameter: S, B, E

Cache – Dimensionierung (cont.)

- Cache speichert immer größere Blöcke / „*Cache-Line*“
- Wortauswahl durch ⟨*block offset*⟩ in Adresse

\+ nutzt räumliche Lokalität aus

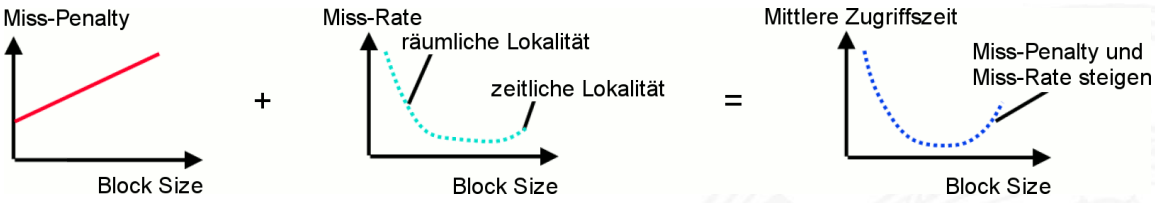
\+ Breite externe Datenbusse

\+ nutzt Burst-Adressierung des Speichers: Adresse nur für erstes Wort vorgeben, dann automatisches Inkrement

\+ kürzere interne Cache-Adressen

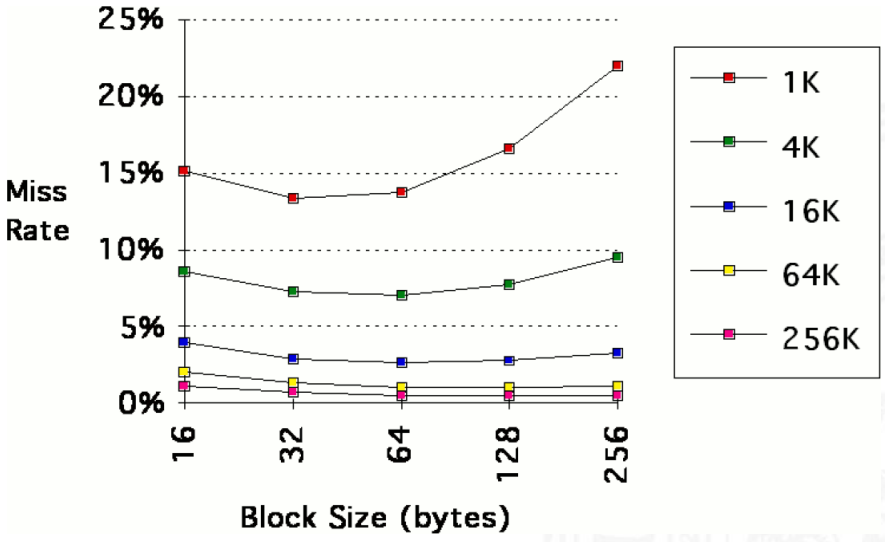
Cache – Dimensionierung (cont.)

Cache- und Block-Dimensionierung



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

Cache – Dimensionierung (cont.)



- Block-Size: 32...128 Byte
  L1-Cache: 4...256 KByte
  L2-Cache: 256...4096 KByte

Cache – Dimensionierung (cont.)

- zusätzliche Software-Optimierungen
  - Ziel: Cache Misses reduzieren
  - Schleifen umsortieren, verschmelzen...
  - Datenstrukturen zusammenfassen...

Cache Ersetzungsstrategie

Wenn der Cache gefüllt ist, welches Datum wird entfernt?

- zufällige Auswahl
- LRU (**L**east **R**ecently **U**sed):
  der „älteste“ nicht benutzte Cache Eintrag
  - echtes LRU als Warteschlange realisiert
  - Pseudo LRU mit baumartiger Verwaltungsstruktur:
    Zugriff wird paarweise mit einem Bit markiert,
    die Paare wieder zusammengefasst usw.
- LFU (**L**east **F**requently **U**sed):
  der am seltensten benutzte Cache Eintrag
  - durch Zugriffszähler implementiert

Cache Schreibstrategie

Wann werden modifizierte Daten des Cache zurückgeschrieben?

- Write-Through: beim Schreiben werden Daten sofort im Cache und im Hauptspeicher modifiziert
  - \+ andere Bus-Master sehen immer den „richtigen“ Speicherinhalt: *Cache-Kohärenz*
  - – Werte werden unnötig oft in Speicher zurückgeschrieben
- Write-Back: erst in den Speicher schreiben, wenn Datum des Cache ersetzt werden würde
  - \+ häufig genutzte Werte (z.B. lokale Variablen) werden nur im Cache modifiziert
  - – Cache-Kohärenz ist nicht gegeben
  - ⇒ spezielle Befehle für „Cache-Flush“
  - ⇒ „non-cacheable“ Speicherbereiche

Cache Kohärenz

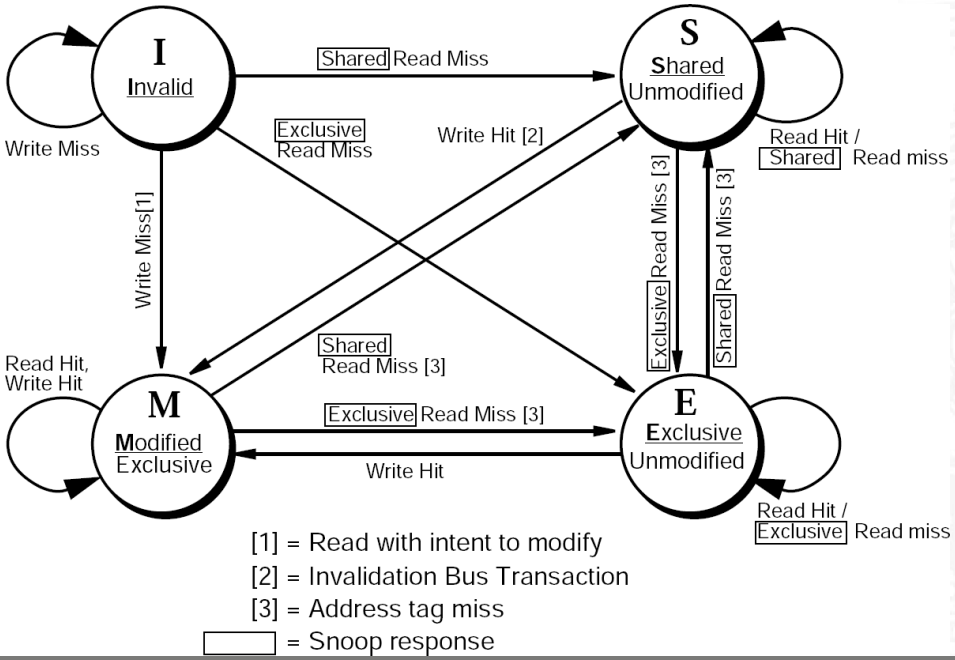
- Daten zwischen Cache und Speicher konsistent halten
- notwendig wenn auch andere Einheiten (Bus-Master) auf Speicher zugreifen können (oder Mehrprozessorsysteme!)
- Harvard-Architektur hat getrennte Daten- und Instruktions-Speicher: einfacherer Instruktions-Cache
  - Instruktionen sind read-only
  - kein Cache-Kohärenz Problem
- „Snooping“
  - Cache „lauscht“ am Speicherbus
  - externer Schreibzugriff ⇒ Cache aktualisieren / ungültig machen
  - externer Lesezugriff ⇒ Cache liefert Daten

Cache Kohärenz (cont.)

- MESI Protokoll
  - besitzt zwei Statusbits für die Protokollzustände
  - **M**odified: Inhalte der Cache-Line wurden modifiziert
  - **E**xclusive unmodified: Cache-Line „gehört“ dieser CPU, nicht modifiz.
  - **S**hared unmodified: Inhalte sind in mehreren Caches vorhanden
  - **I**nvalid: ungültiger Inhalt, Initialzustand

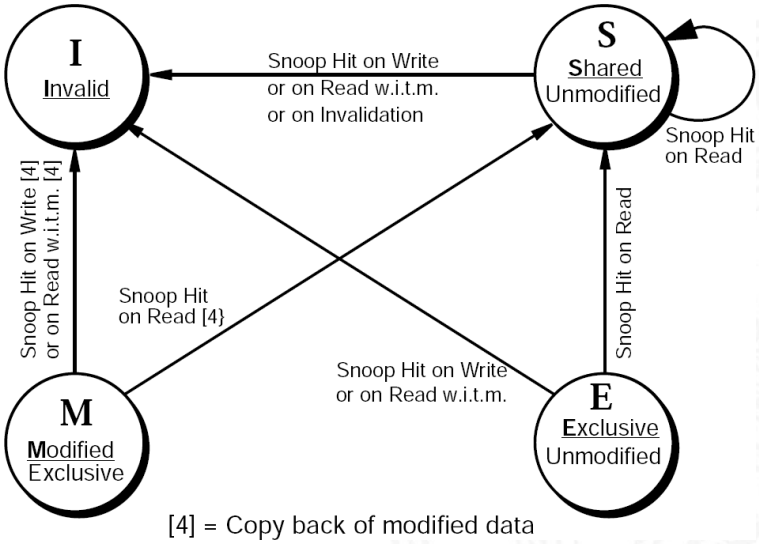
Cache Kohärenz (cont.)

- Zustandsübergänge für „Bus Master“ CPU



Cache Kohärenz (cont.)

- Zustandsübergänge für "Snooping" CPU



Optimierung der Cachezugriffe

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

$\Rightarrow$ Verbesserung der Cache Performanz durch kleinere T_{Miss} am einfachsten zu realisieren

- mehrere Cache Ebenen
- Critical Word First: bei großen Cache Blöcken (mehrere Worte) gefordertes Wort zuerst holen und gleich weiterleiten
- Read-Miss hat Priorität gegenüber Write-Miss
  $\Rightarrow$ Zwischenspeicher für Schreiboperationen (Write Buffer)
- Merging Write Buffer: aufeinanderfolgende Schreiboperationen zwischenspeichern und zusammenfassen
- Victim Cache: kleiner voll-assoziativer Cache zwischen direct-mapped Cache und nächster Ebene
  „sammelt" verdrängte Cache Einträge

Optimierung der Cachezugriffe (cont.)

⇒ Verbesserung der Cache Performanz durch kleinere R_{Miss}
- größere Caches (– mehr Hardware)
- höhere Assoziativität (– langsamer)

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

  Beispiel: NetBurst Architektur (Pentium 4)

Chiplayout

ARM7 / ARM10

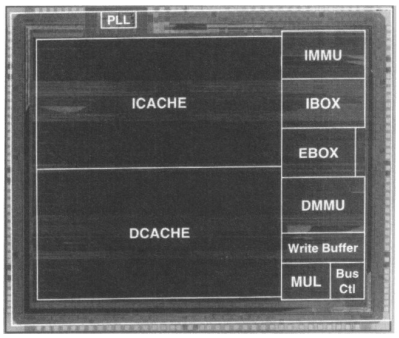
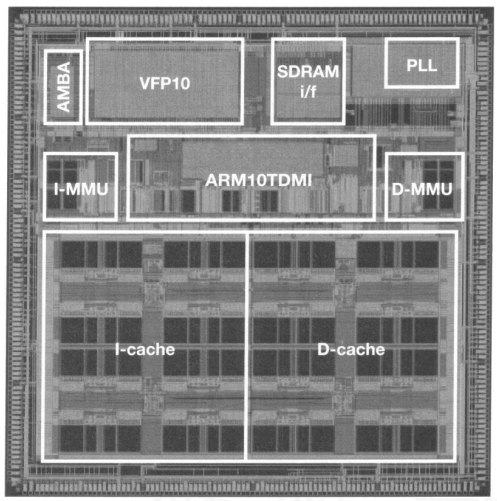


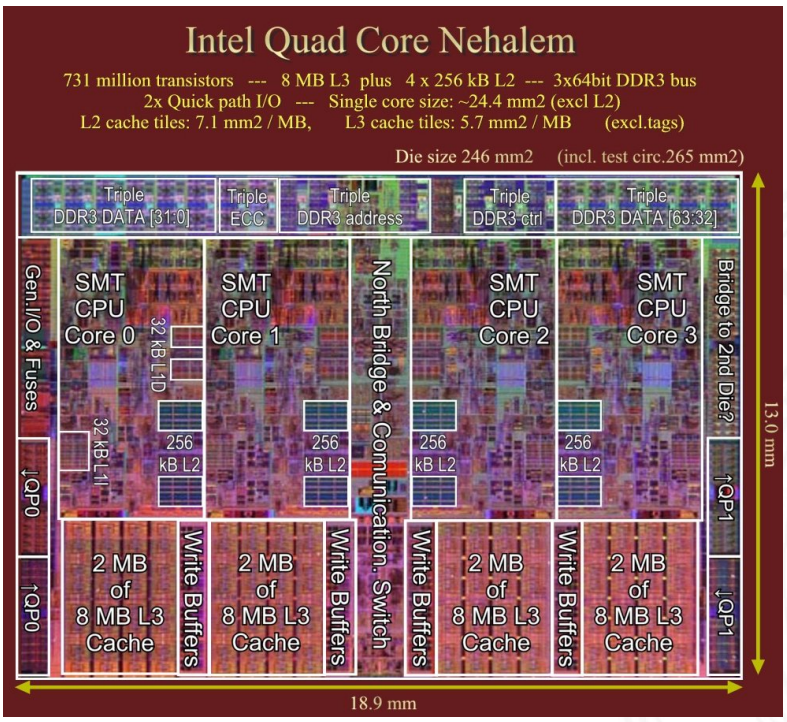
Photo courtesy of Intel Corp.



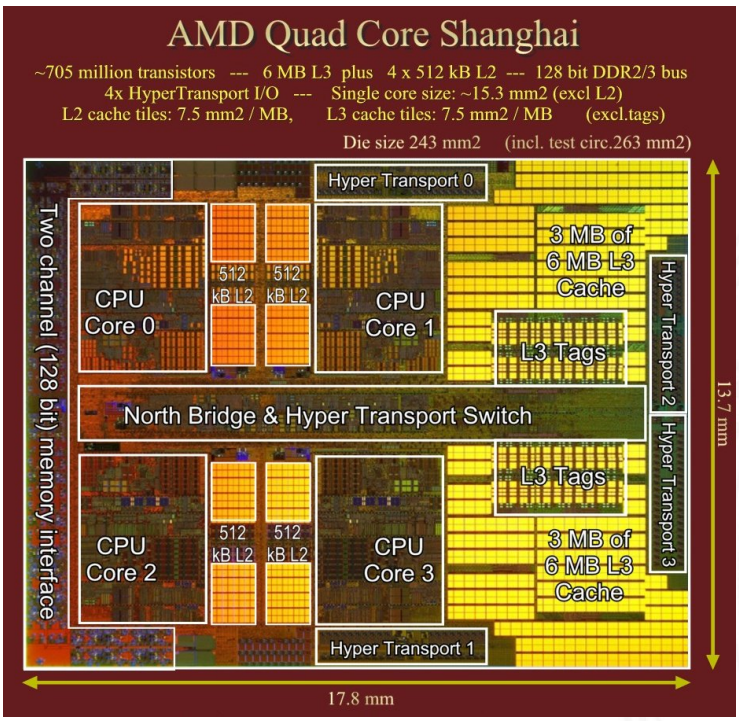
© ARM Limited

- IBOX: Steuerwerk (instruction fetch und decode)
  EBOX: Operationswerk, ALU, Register (execute)
  IMMU/DMMU: Virtueller Speicher (instruction/data TLBs)
  ICACHE: Instruction Cache
  DCACHE: Data Cache

Chiplayout (cont.)



Chiplayout (cont.)



Cache vs. Programmcode

Programmierer kann für maximale Cacheleistung optimieren

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

Systeme bevorzugen einen *Cache-freundlichen* Code

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

Virtueller Speicher – Motivation

Speicher-Paradigmen

- Programmierer
  - ein großer Adressraum
  - linear adressierbar
- Betriebssystem
  - eine Menge laufender Tasks / Prozesse
  - read-only Instruktionen
  - read-write Daten

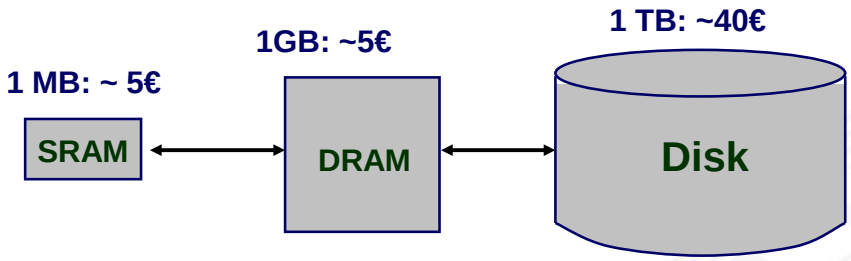
Konsequenz

virtueller Speicher umfasst drei Aspekte der (teilweise) widersprüchlichen Anforderungen an „Speicher“

Virtueller Speicher – Motivation (cont.)

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

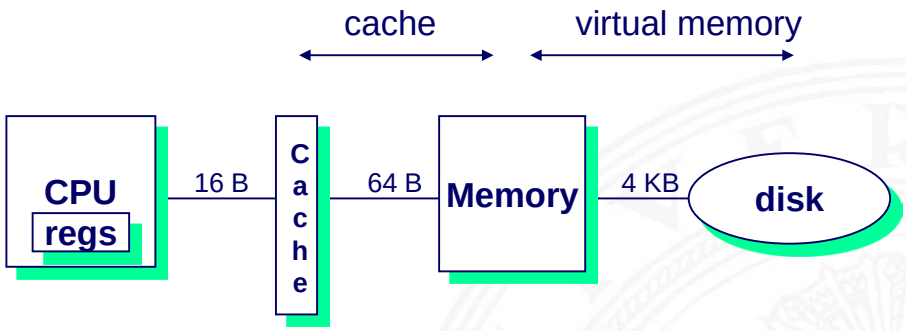
Festplatte „erweitert“ Hauptspeicher



- Vollständiger Adressraum zu groß ⇒ DRAM ist *Cache*
  - 32-bit Adressen: $\approx 4 \times 10^9$ Byte 4 Milliarden
  - 64-bit Adressen: $\approx 16 \times 10^{16}$ Byte 16 Quintillionen
- Speichern auf Festplatte ist $\approx 125\times$ billiger als im DRAM
  - 1 TiB DRAM: $\approx 5\,000\,€$
  - 1 TiB Festplatte: $\approx 40\,€$

⇒ kostengünstiger Zugriff auf große Datenmengen

Ebenen in der Speicherhierarchie

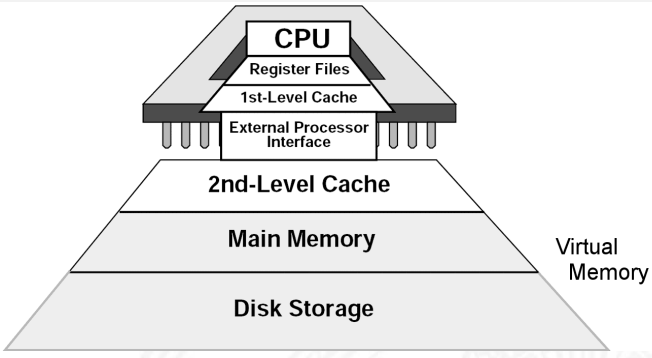


	Register	Cache	Memory	Disk Memory
size:	64 B	32 KB-12MB	8 GB	2 TB
speed:	300 ps	1 ns	8 ns	4 ms
$/Mbyte:		5€/MB	5€/GB	4 Ct./GB
line size:	16 B	64 B	4 KB	

larger, slower, cheaper

Ebenen in der Speicherhierarchie (cont.)

- Hauptspeicher als Cache für den Plattenspeicher

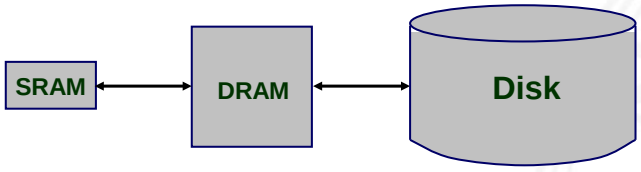


- Parameter der Speicherhierarchie

	1st-Level Cache	virtueller Speicher
Blockgröße	16-128 Byte	4-64 kByte
Hit-Dauer	1-2 Zyklen	40-100 Zyklen
Miss Penalty	8-100 Zyklen	70.000-6.000.000 Zyklen
Miss Rate	0,5-10 %	0,00001-0,001 %
Speichergröße	8-64 kByte	16-8192 MByte

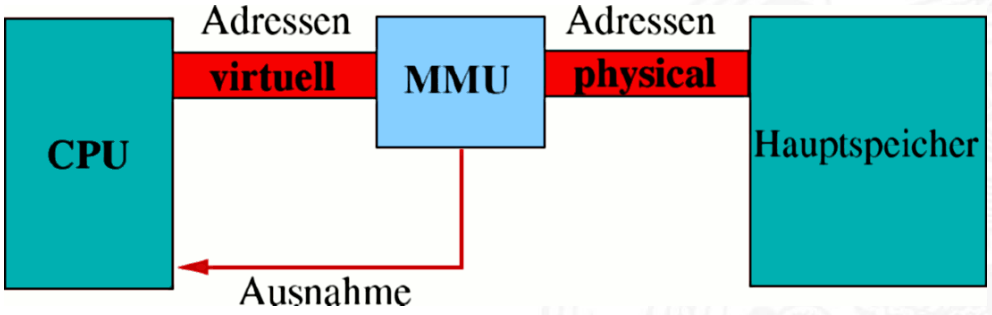
Ebenen in der Speicherhierarchie (cont.)

- DRAM vs. Festplatte ist extremer als SRAM vs. DRAM
  - Zugriffswartezeiten
    - DRAM $\approx 10\times$ langsamer als SRAM
    - Festplatte $\approx 500\,000\times$ langsamer als DRAM
  - $\Rightarrow$ Nutzung der räumlichen Lokalität wichtig
    - erstes Byte $\approx 500\,000\times$ langsamer als nachfolgende Bytes



Prinzip des virtuellen Speichers

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



  - Umsetzung von virtuellen zu physikalischen Adressen, Programm-Relokation

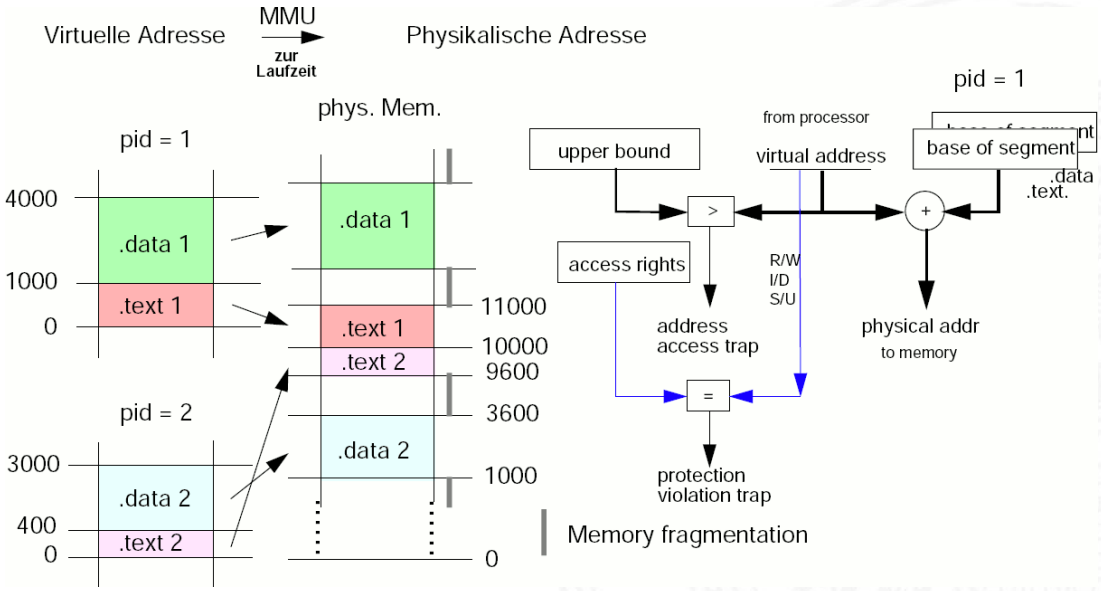
Prinzip des virtuellen Speichers (cont.)

- Umsetzungstabellen werden vom Betriebssystem verwaltet
- wegen des Speicherbedarfs der Tabellen beziehen sich diese auf größere Speicherblöcke (*Segmente* oder *Seiten*)

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

Virtueller Speicher: Segmentierung

- Unterteilung des Adressraums in kontinuierliche Bereiche *variabler* Größe



Virtueller Speicher: Segmentierung (cont.)

- Idee: Trennung von Instruktionen, Daten und Stack

⇒ Abbildung von *Programmen* in den *Hauptspeicher*

\+ Inhalt der Segmente: logisch zusammengehörige Daten

\+ getrennte Zugriffsrechte, Speicherschutz

\+ exakte Prüfung der Segmentgrenzen

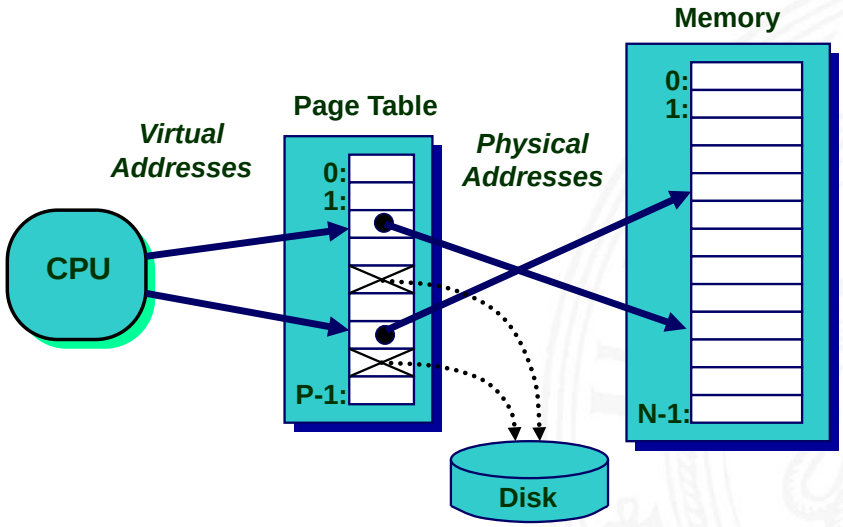
– Segmente könne sehr groß werden

– Ein- und Auslagern von Segmenten kann sehr lange dauern

– „Verschnitt“, **Memory Fragmentation**

Virtueller Speicher: Paging / Seitenadressierung

- Unterteilung des Adressraums in Blöcke *fester* Größe = Seiten
  Abbildung auf Hauptspeicherblöcke = Kacheln



Virtueller Speicher: Paging / Seitenadressierung (cont.)

⇒ Abbildung von *Adressen* in den *virtuellen Speicher*

\+ Programme können größer als der Hauptspeicher sein

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

\+ feste Seitengröße: einfache Verwaltung in Hardware

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

\+ gemeinsam genutzte Programmteile/-Bibliotheken können sehr einfach in das Konzept integriert werden

- Windows: `.dll`-Dateien
- Unix/Linux: `.so`-Dateien

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
21.4 Speicherhierarchie - Virtueller Speicher
64-040 Rechnerstrukturen

Virtueller Speicher: Paging / Seitenadressierung (cont.)

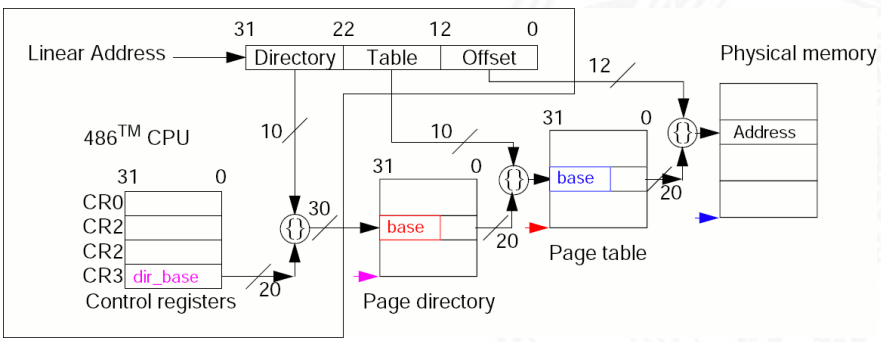
- große Miss-Penalty (Nachladen von der Platte)
  ⇒ Seiten sollten relativ groß sein: 4 oder 8 kByte

\− Speicherplatzbedarf der Seitentabelle
viel virtueller Speicher, 4 kByte Seitengröße

= sehr große Pagetable

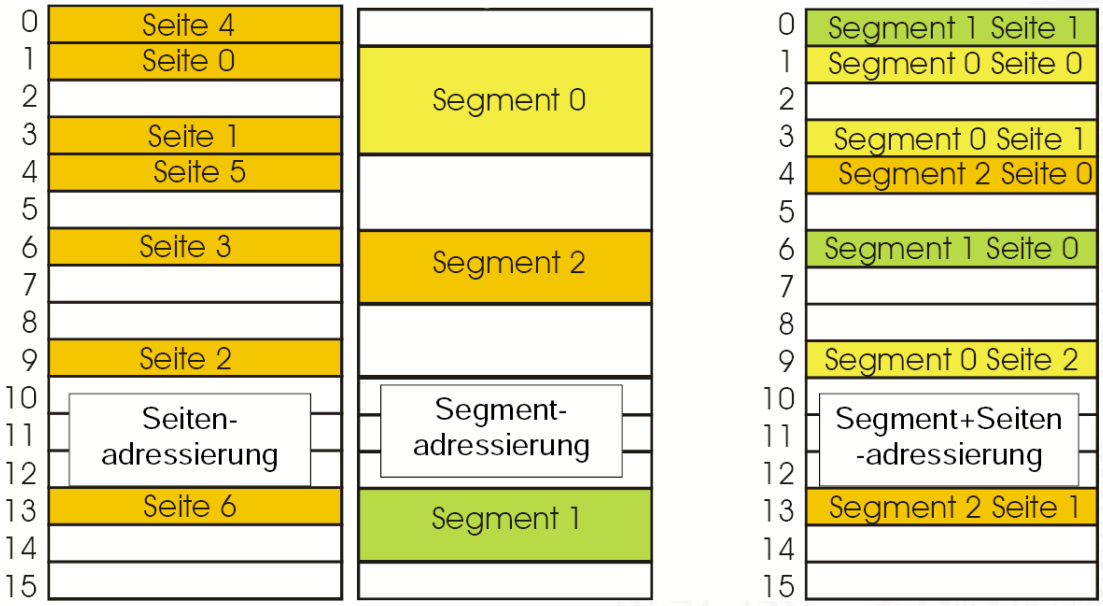
⇒ Hash-Verfahren (*inverted page tables*)

⇒ mehrstufige Verfahren



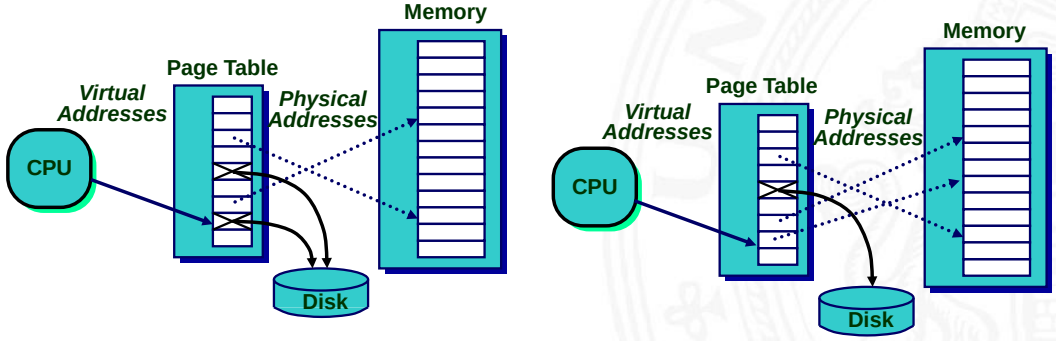
Virtueller Speicher: Segmentierung + Paging

aktuell = Mischung: Segmentierung und Paging (seit I386)



Seitenfehler

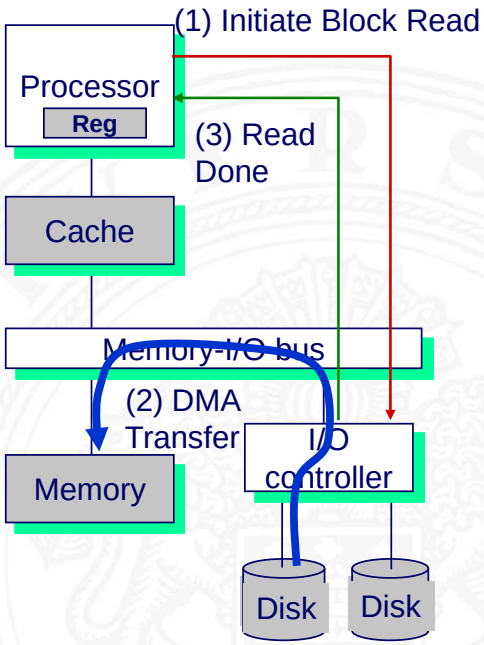
- Seiten-Tabelleneintrag: Startadresse der virt. Seite auf Platte
- Daten von Festplatte in Speicher laden:
  Aufruf des „Exception handler“ des Betriebssystems
  - laufender Prozess wird unterbrochen, andere können weiterlaufen
  - Betriebssystem kontrolliert die Platzierung der neuen Seite
    im Hauptspeicher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



Seitenfehler (cont.)

Behandlung des Seitenfehlers

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

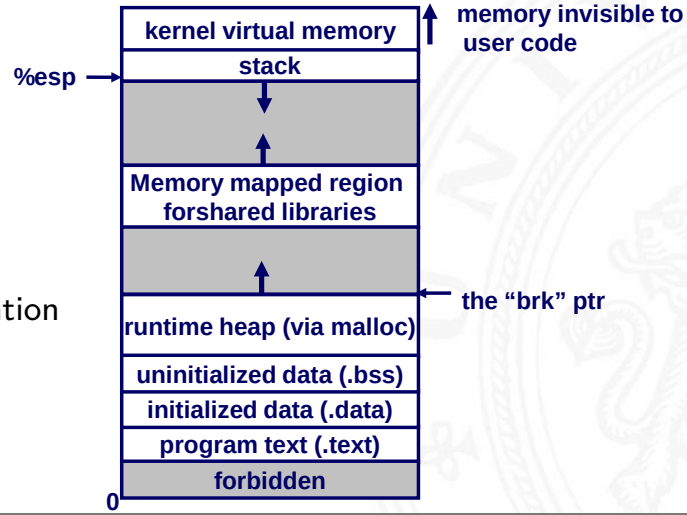


Separate virtuelle Adressräume

Mehrere Prozesse können im physikalischen Speicher liegen

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

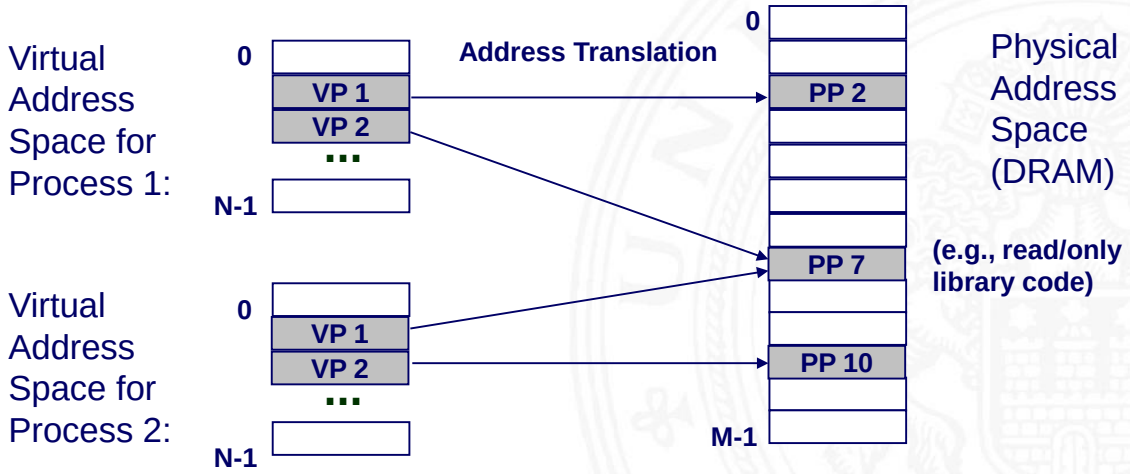
Linux x86
Speicherorganisation



Separate virtuelle Adressräume (cont.)

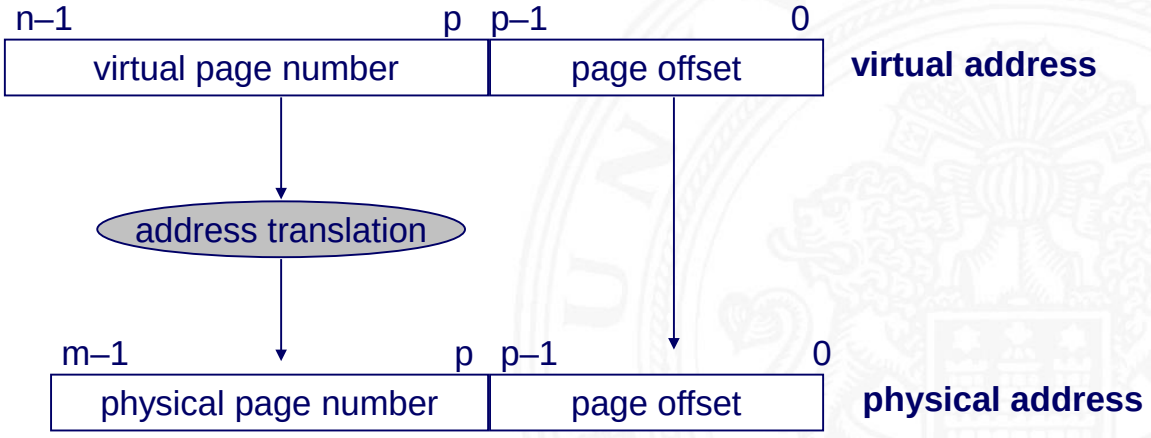
Auflösung der Adresskonflikte

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



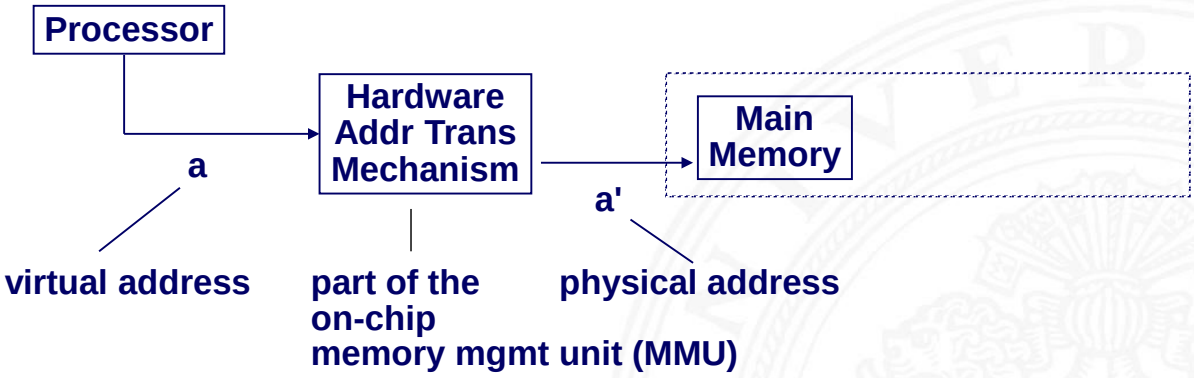
Virtueller Speicher – Adressumsetzung

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



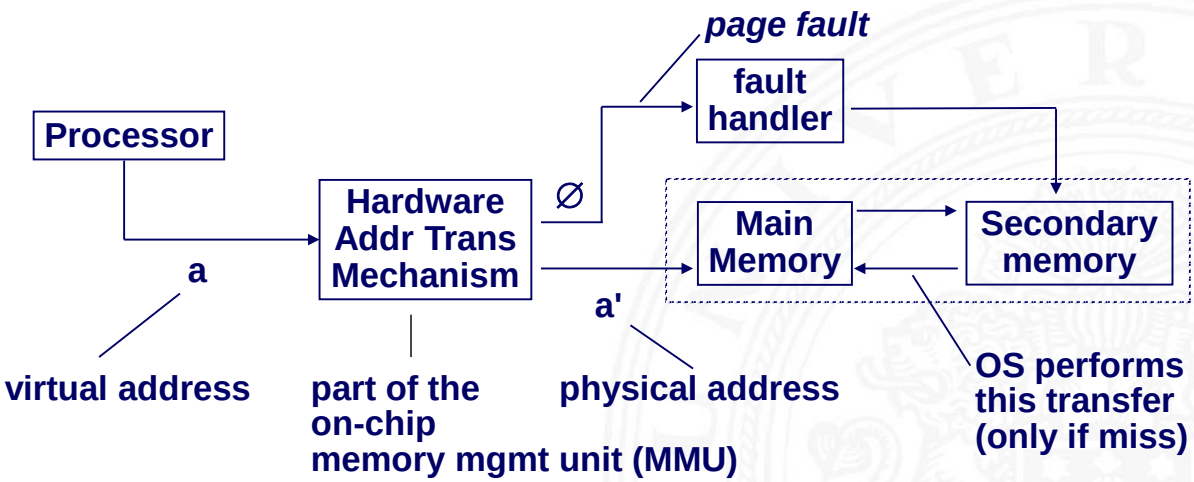
Virtueller Speicher – Adressumsetzung (cont.)

- virtuelle Adresse: Hit

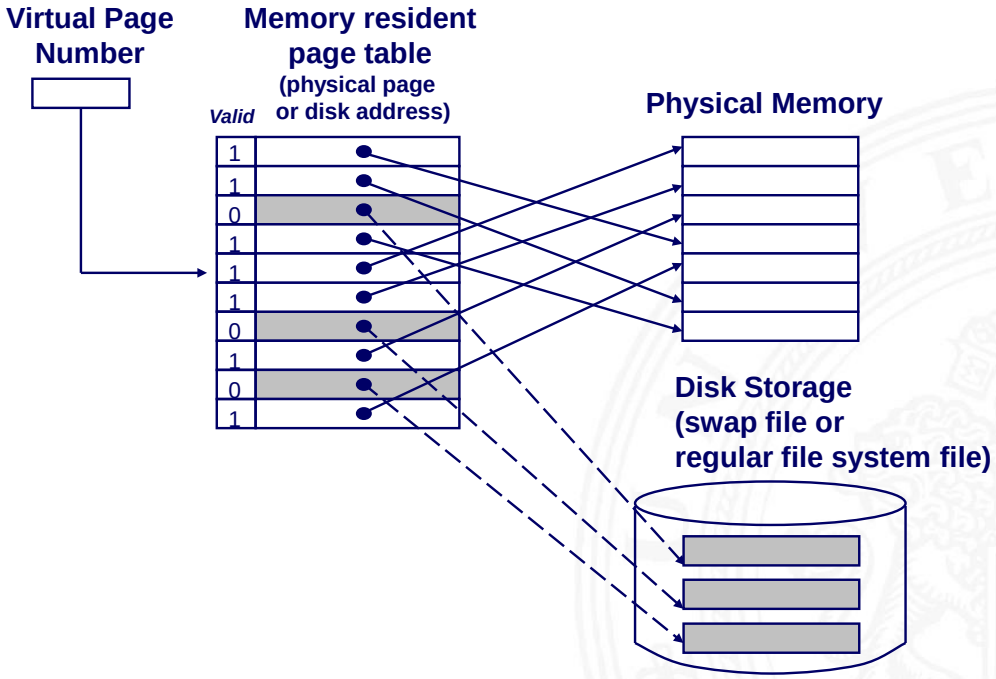


Virtueller Speicher – Adressumsetzung (cont.)

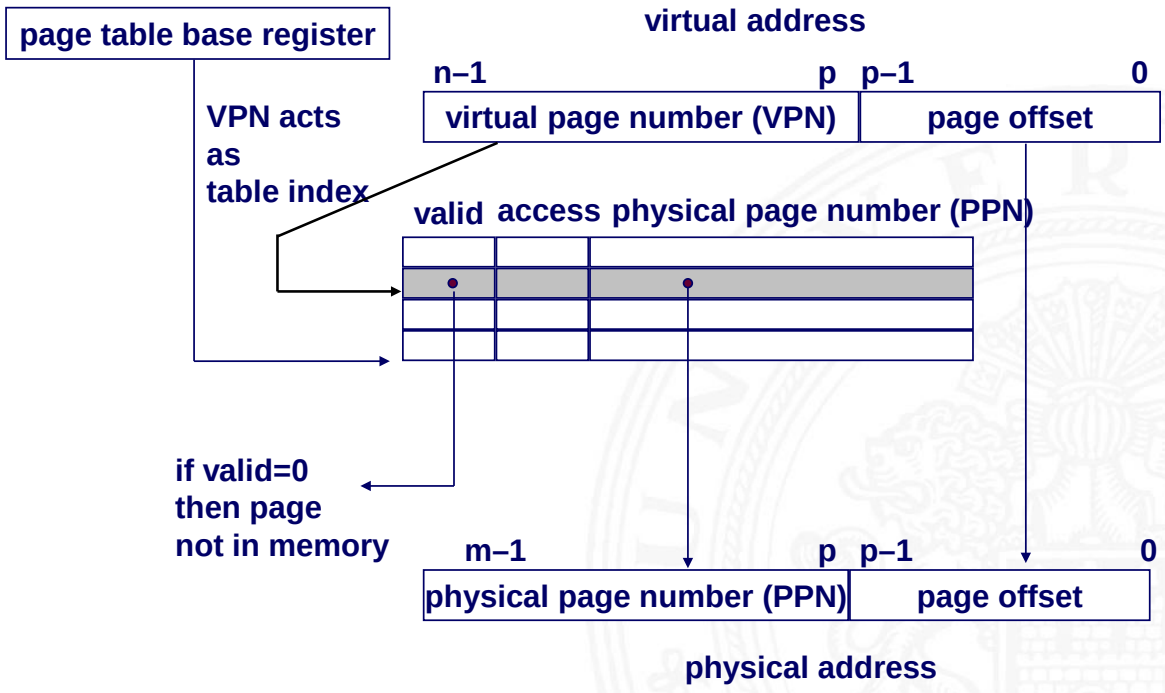
- virtuelle Adresse: Miss



Seiten-Tabelle

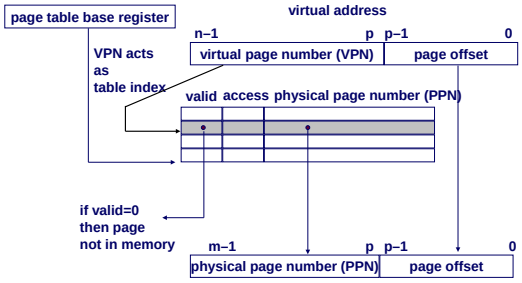


Seiten-Tabelle (cont.)



Seiten-Tabelle (cont.)

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

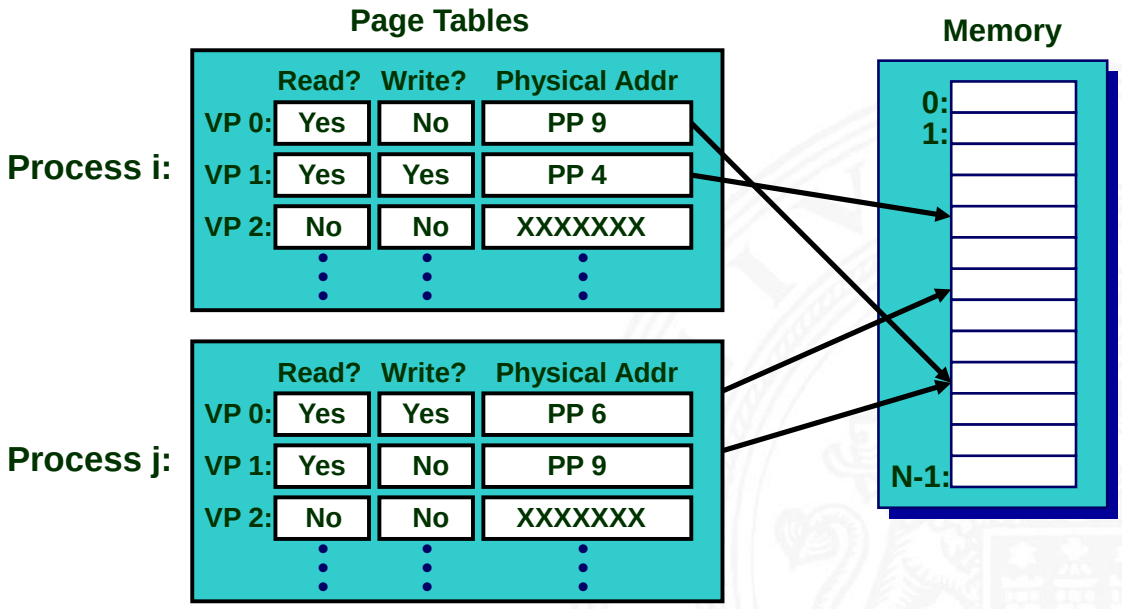


Zugriffsrechte

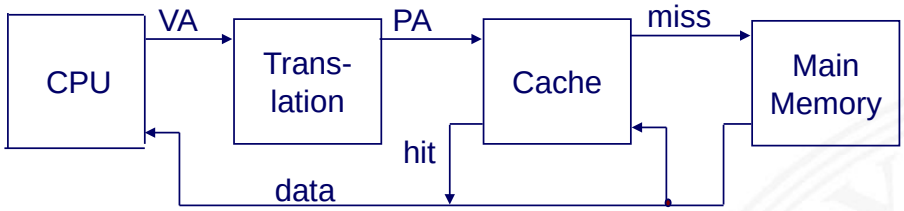
Schutzüberprüfung

- Zugriffsrechtefeld gibt Zugriffserlaubnis an
  - z.B. read-only, read-write, execute-only
  - typischerweise werden zahlreiche Schutzmodi unterstützt
- Schutzrechteverletzung wenn Prozess/Benutzer nicht die nötigen Rechte hat
- bei Verstoß erzwingt die Hardware den Schutz durch das Betriebssystem („Trap“ / „Exception“)

Zugriffsrechte (cont.)



Integration von virtuellem Speicher und Cache



Die meisten Caches werden *physikalisch adressiert*

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

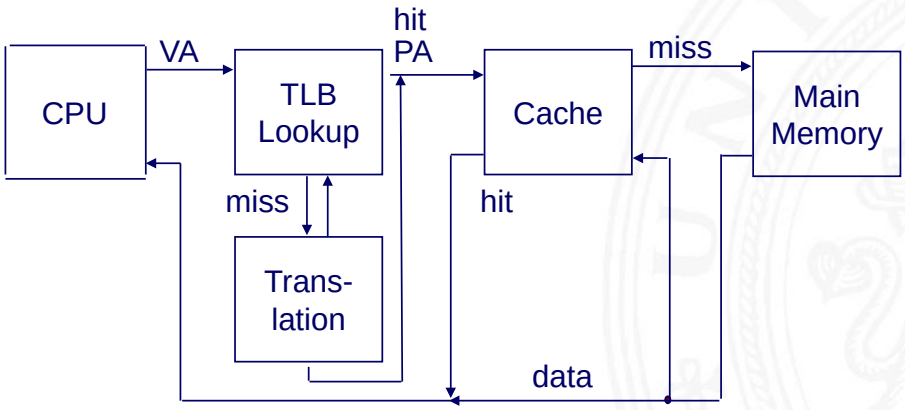
Die Adressumsetzung wird vor dem Cache „Lookup“ durchgeführt

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

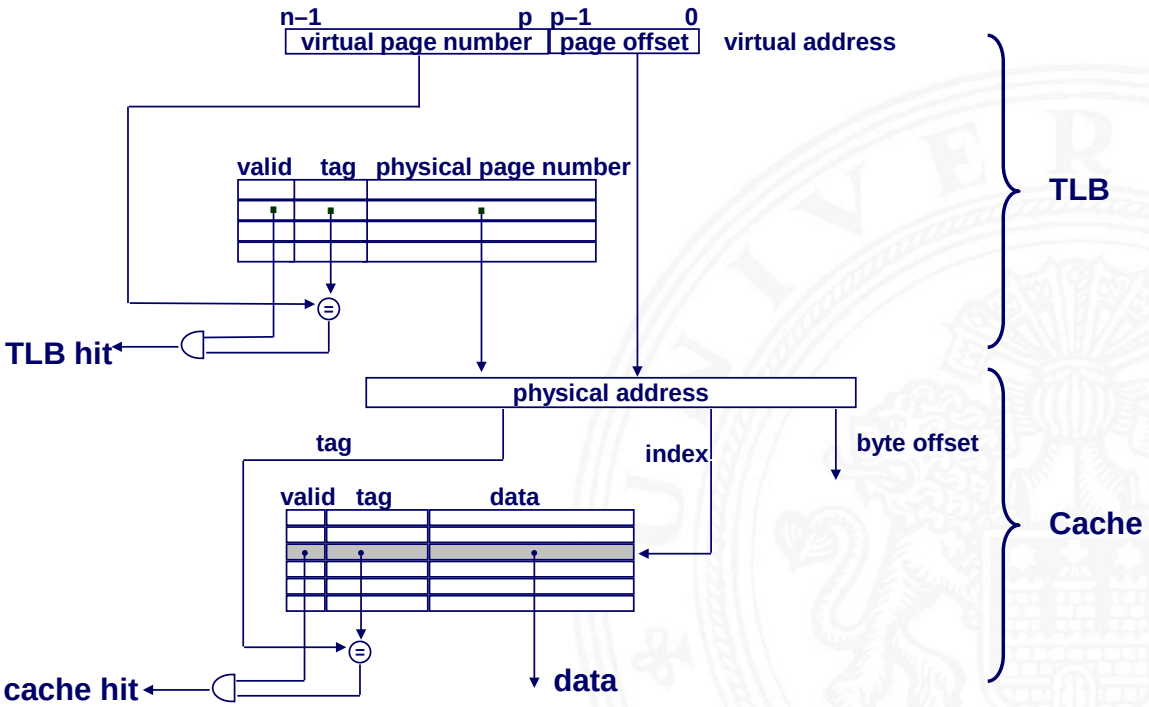
TLB / „Translation Lookaside Buffer“

Beschleunigung der Adressumsetzung für virtuellen Speicher

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



TLB / „Translation Lookaside Buffer“ (cont.)

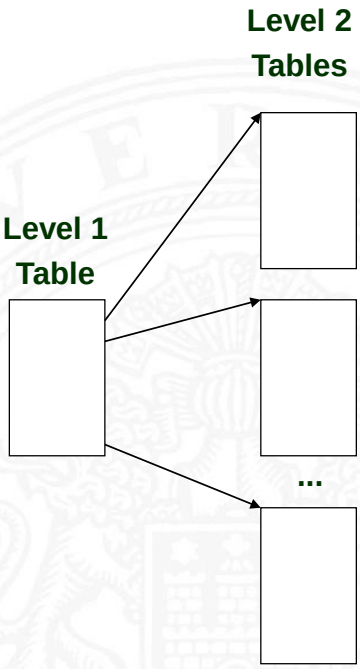


mehrstufige Seiten-Tabellen

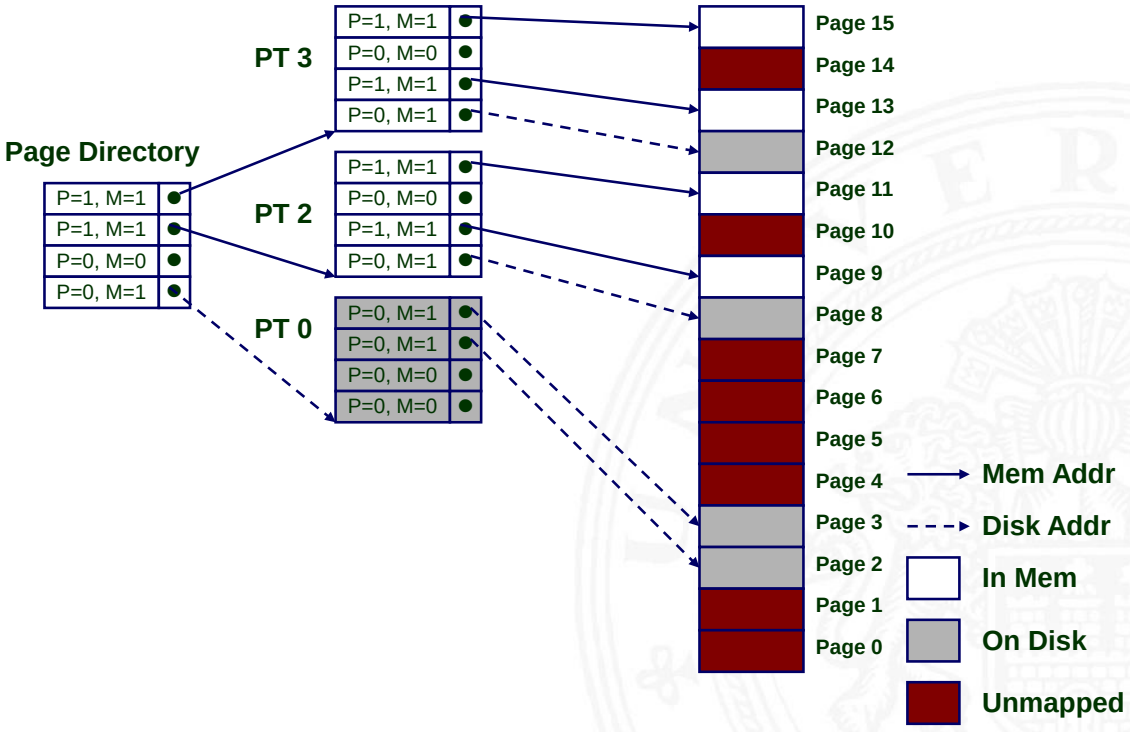
- Gegeben
  - 4 KiB (2^{12}) Seitengröße
  - 32-Bit Adressraum
  - 4-Byte PTE („Page Table Entry“) Seitentabelleneintrag
- Problem
  - erfordert 4 MiB Seiten-Tabelle
  - 2^{20} Bytes

$\Rightarrow$ übliche Lösung

- mehrstufige Seiten-Tabellen („multi-level“)
- z.B. zweistufige Tabelle (Pentium P6)
  - Ebene-1: 1024 Einträge $\rightarrow$ Ebene-2 Tabelle
  - Ebene-2: 1024 Einträge $\rightarrow$ Seiten

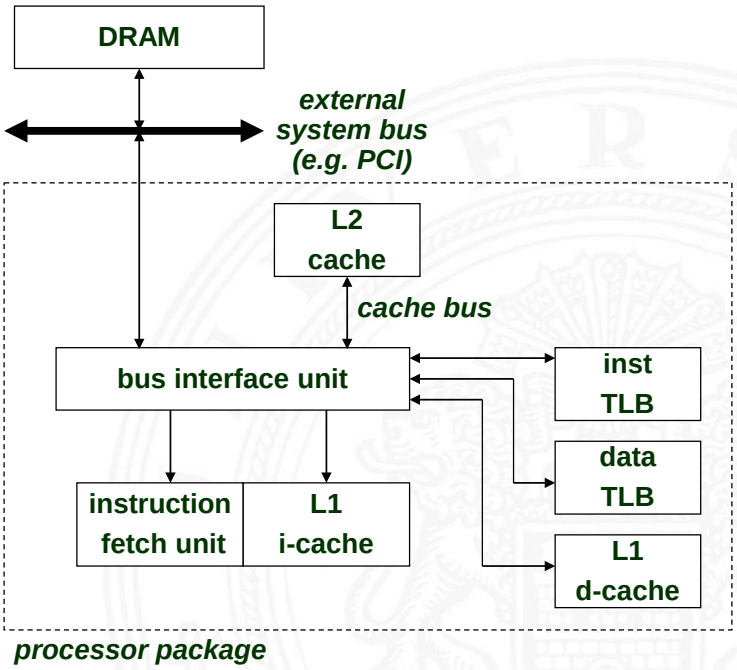


mehrstufige Seiten-Tabellen (cont.)

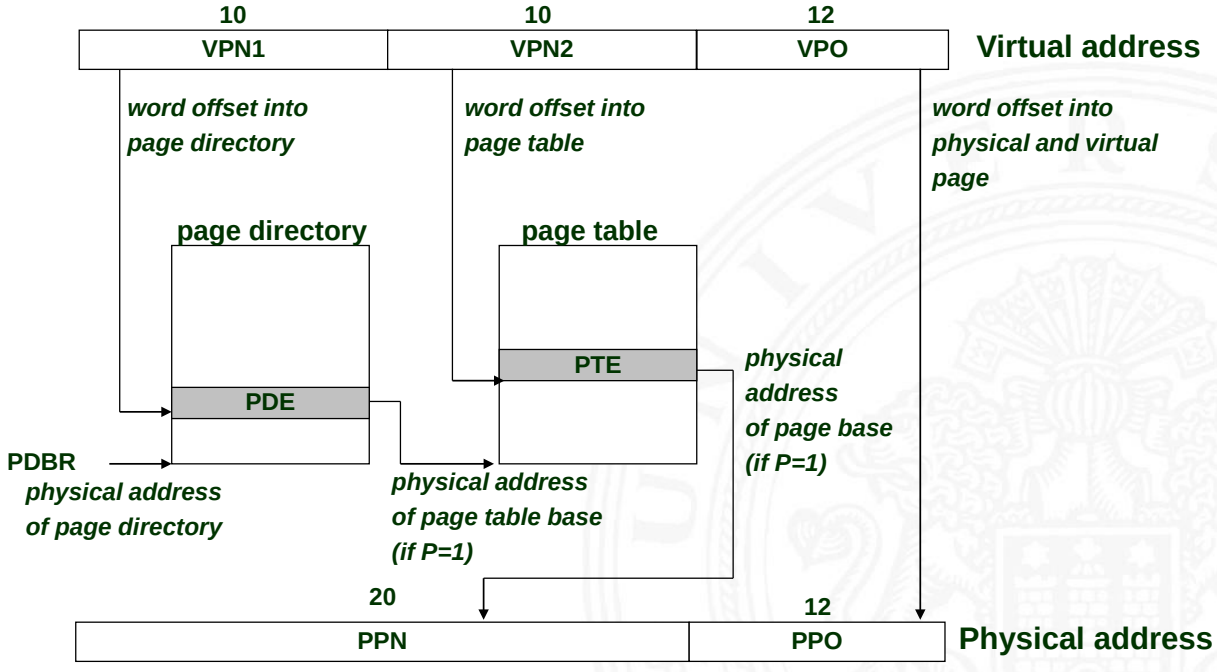


Beispiel: Pentium und Linux

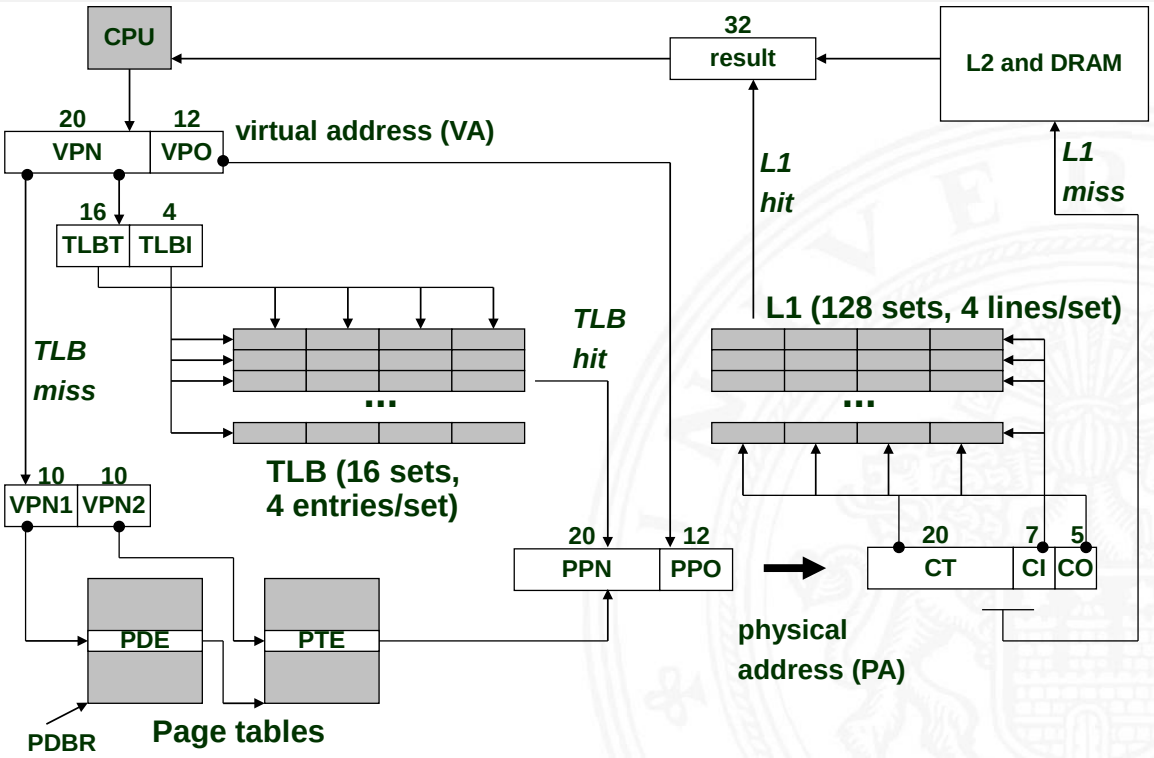
- 32-bit Adressraum
- 4 KB Seitengröße
- L1, L2 TLBs
  4fach assoziativ
- Instruktions TLB
  32 Einträge
  8 Sets
- Daten TLB
  64 Einträge
  16 Sets
- L1 I-Cache, D-Cache
  16 KB
  32 B Cacheline
  128 Sets
- L2 Cache
  Instr.+Daten zusammen
  128 KB . . . 2 MB



Beispiel: Pentium und Linux (cont.)



Beispiel: Pentium und Linux (cont.)



Zusammenfassung

Cache Speicher

- Dient nur zur Beschleunigung
- Verhalten unsichtbar für Anwendungsprogrammierer und OS
- Komplett in Hardware implementiert

Zusammenfassung (cont.)

Virtueller Speicher

- Ermöglicht viele Funktionen des Betriebssystems
  - Prozesse erzeugen („exec“ / „fork“)
  - Taskwechsel
  - Schutzmechanismen
- Implementierung mit Hardware und Software
  - Software verwaltet die Tabellen und Zuteilungen
  - Hardwarezugriff auf die Tabellen
  - Hardware-Caching der Einträge (TLB)

Universität Hamburg
MIN-Fakultät
Fachbereich Informatik
21.4.1 Speicherhierarchie - Virtueller Speicher - Beispiel: Pentium und Linux
64-040 Rechnerstrukturen

Zusammenfassung (cont.)

- Sicht des Programmierers
  - großer „flacher“ Adressraum
    - kann große Blöcke benachbarter Adressen zuteilen
  - Prozessor „besitzt“ die gesamte Maschine
    - hat privaten Adressraum
    - bleibt unberührt vom Verhalten anderer Prozesse
- Sicht des Systems
  - Virtueller Adressraum von Prozessen durch Abbildung auf Seiten
    - muss nicht fortlaufend sein
    - wird dynamisch zugeteilt
    - erzwingt Schutz bei Adressumsetzung
  - Betriebssystem verwaltet viele Prozesse gleichzeitig
    - ständiger Wechsel zwischen Prozessen
    - vor allem wenn auf Ressourcen gewartet werden muss