

Rechnerarchitektur

Einführung

Univ.-Prof. Dr. Rainer Böhme

Wintersemester 2021/22 · 6. Oktober 2021

Ziele dieser Lehrveranstaltung

- Vermittlung von Grundwissen über den **Entwurf** von **Digitalen Schaltungen**
- Verständnis des **Aufbaus** und der **Arbeitsweise** von Hardware
- Verständnis von **Architekturprinzipien** und **Organisationsformen** moderner Rechner
- Einsicht in das Zusammenspiel von Hardware und Software
- Erstellung von **maschinennahen Programmen** am Beispiel der ARM-Architektur
- Lernen von Ansätzen zur **Bewertung** und zum Vergleich von Rechnersystemen



Organisation dieser Lehrveranstaltung

Vorlesung

- Zeit: mittwochs, 09:15–11:00 Uhr
- Ablauf: Stream auf https://twitch.tv/uibkseclab_de mit Chat-Rückkanal
- Umfang: 3 ECTS-AP European Credit Transfer System-Anrechnungspunkte

Proseminar

- Zeit: donnerstags, 17:00–18:00 Uhr, ab 7.10.
- Ablauf: Stream auf https://twitch.tv/uibkseclab_de
- Interaktion: Chat-Rückkanal und Online-Tests im eCampus LMS („OLAT“)
- Umfang: 2 ECTS-AP – **Anwesenheitspflicht !**

Tutorium

- Zeit: donnerstags, 09:15–10:00 Uhr, ab 14.10., freiwillig
- Ort: HS E
- Tutor: Patrick Aschenbrenner Patrick.Aschenbrenner@student.uibk.ac.at

Bewertung

Vorlesung

- Schriftliche Klausur (Theorie), falls nötig online
- 1. Klausur 02.02.2022, 09:00–11:00 Uhr
- **Anmeldung** bis zum 18.01.2022 möglich.
- Zwei weitere Termine,
voraussichtlich Anfang April und Mitte September

Proseminar

- Bewertung der Einsendeaufgaben
- Wöchentliche, kurze Online-Tests
- Weitere Details im Proseminar

Lesen Sie auch die FAQ zu Prüfungen auf unserer Webseite.

Vorlesungsmaterialien

Folien

- Die Folien und Videoeinheiten stehen nach der Vorlesung im Internet zur Verfügung.
- Verwenden Sie die Seitenzahlen als Referenz für Ihre Mitschrift.
- Die kompletten Streams bleiben ca. 10 Tage online.

Lehrbücher

- D. A. Patterson and J. L. Hennessy, Computer Organization and Design, 5. Auflage, Morgan Kaufmann, 2014.
(4. Auflage auch auf Deutsch erschienen)
- D. W. Hoffmann, Grundlagen der Technischen Informatik, 3. Auflage, Hanser, 2013.
- W. Oberschelp und G. Vossen, Rechneraufbau und Rechnerstrukturen, 10. Auflage, Oldenbourg, 2006.

Danksagung

Die Folieninhalte basieren teilweise auf den Materialien meiner Vorgänger:
Univ.-Prof. Drs. Matthias Harders, Falko Dressler, Alfred Strey

Team

Professor



Univ.-Prof. Dr.-Ing.
Rainer Böhme

Sekretärin



Jenifer
Payr



Manuel
Knoflach-Schrott



Gloria
Dzida

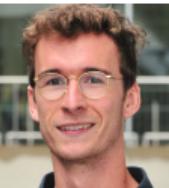
WissenschaftlerInnen



Dr. Svetlana
Abramova



Dr. Paulina
Pesch



Dr. Daniel
Woods



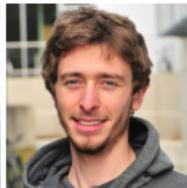
Michael
Fröwis



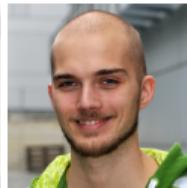
Maximilian
Hils



Nora
Hofer



Patrik
Keller



Alexander
Schlägl

Bemerkungen zur akademischen Ausbildung

Rolle der Universität

- Kombination von Forschung und Lehre
- Innovation = Idee + Ausführung
- **Lernziel:** Spaß an systematischem Denken
- „Rohstoff für Wachstum“

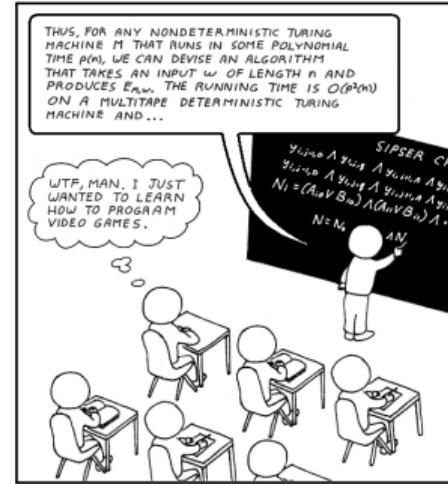
Bedeutung des „Stoffs“ in der Lehre

- Vermittlung von Prinzipien und Denkmethoden, die Innovation fördern
- Bekannte, in der Praxis verbreitete Lösungen sind Anschauungsmaterial
- Praxisrelevantes Faktenwissen als Nebeneffekt (veraltet schnell)



Bücherrad, Agostino Ramelli 1588, Bild: Wikipedia

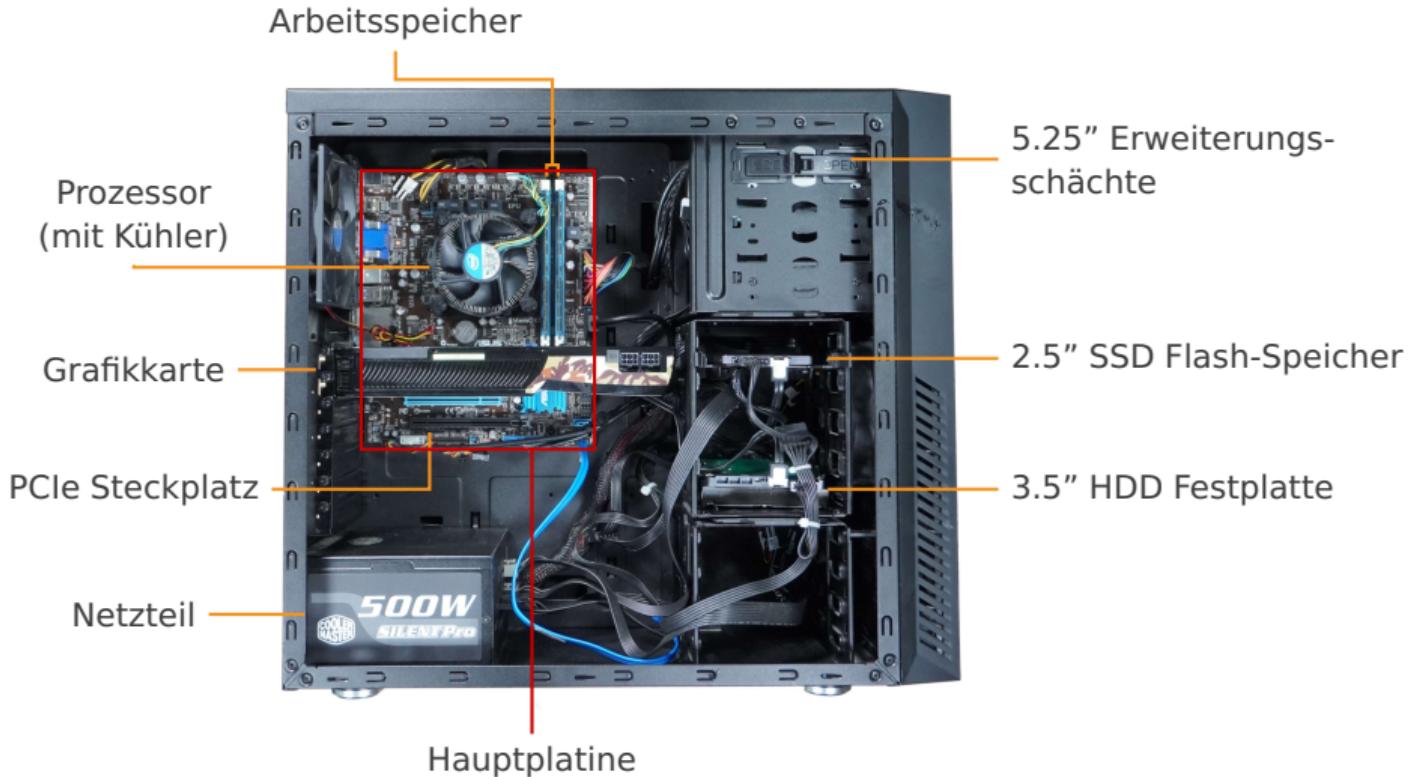
Konsequenzen für die Lehre im Bachelorstudium



Voraussetzungen

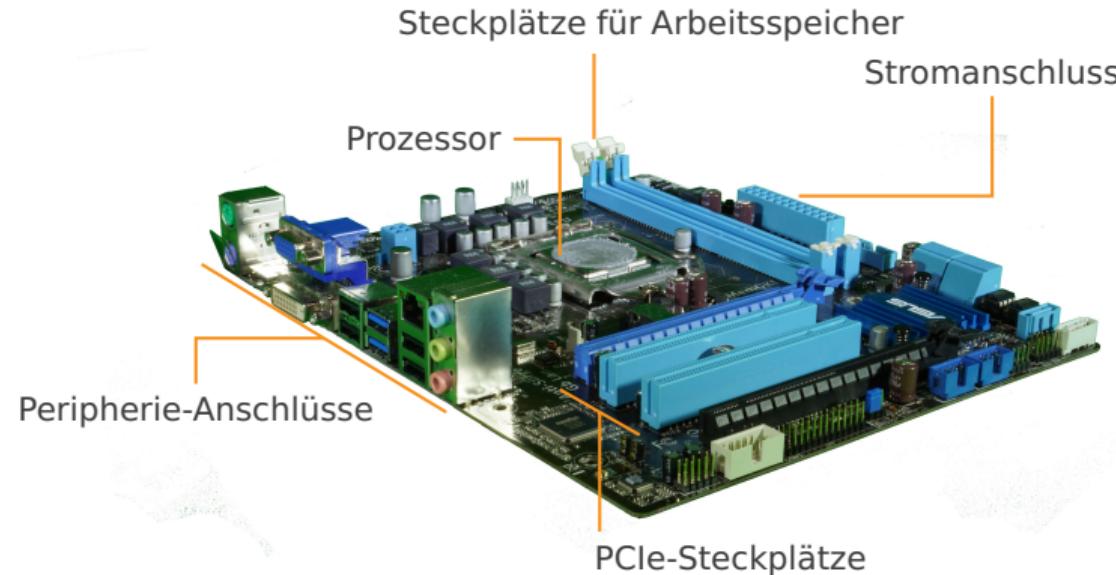
- **Grundlagen** des Fachs
- **Austausch:** Diskurs, Gastvorträge, Auslandssemester (ja, es geht wieder!), etc.
- Fachspezifische **Kommunikation**: Terminologie und Konventionen, Fremdsprachen, wissenschaftliche Literatur („Papers“)
- Fachspezifische **Fertigkeiten**, z. B. Programmieren
- **Ethik:** Standards guter wissenschaftlicher Praxis (z. B. Reproduzierbarkeit)
→ Fehler passieren trotzdem

Innenansicht eines Desktop-PCs



Hauptplatine

(engl. Motherboard)



Populäre Einplatinenrechner



Arduino

8-Bit-Atmel AVR CPU, 16 MHz

32 KB Flash-Speicher, USB

ca. 15 Euro



Raspberry Pi

32-Bit-ARM Cortex-A CPU, 1.2 GHz

1 GB Arbeitsspeicher, HDMI, Ethernet

ca. 35 Euro



2 €

Micro Bit

32-Bit-ARM Cortex-M0 CPU, 16 MHz

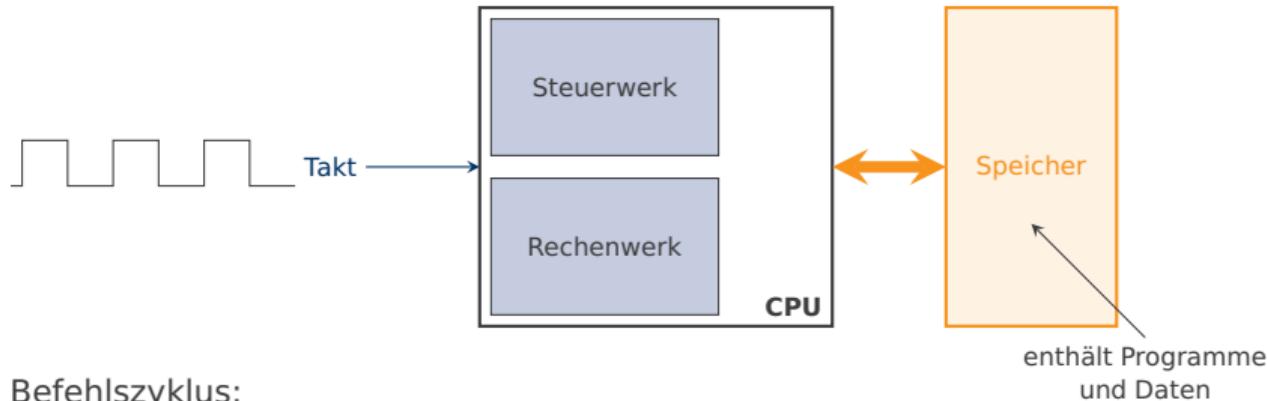
256 KB Flash-Speicher, USB, Bluetooth

Sensoren, LED-Matrix, Tasten

ca. 15 Euro

Prozessor

(engl. Central Processing Unit, CPU)



In jedem Befehlszyklus:

- Holen einer Instruktion aus dem Speicher
- Dekodieren im Steuerwerk
- Austausch benötigter Daten (= Operanden) mit dem Speicher
- Ausführen im Rechenwerk

Die CPU ist die Hauptkomponente jedes Rechners.

Beispiel

Programm in der Programmiersprache C

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Was ist Software?\n");
6
7     return 0;
8 }
```

Beispiel

Maschinennahes Assembler-Programm (x86-Befehlssatz, erzeugt vom Compiler)

```
text:00001F44          call    _exit
text:00001F49          hlt
text:00001F49 start     endp
text:00001F49
text:00001F4A
text:00001F4A ; ===== S U B R O U T I N E =====
text:00001F4A
text:00001F4A ; Attributes: bp-based frame
text:00001F4A
text:00001F4A         public   main
text:00001F4A _main      proc near             ; CODE XREF: start+30↑p
text:00001F4A         push    ebp
text:00001F4B         mov     ebp, esp
text:00001F4D         push    ebx
text:00001F4E         sub     esp, 14h
text:00001F51         call    $+5
text:00001F56         pop    ebx
text:00001F57         lea    eax, (aWasIstSoftware - 1F56h)[ebx] ; "Was ist Software?"
text:00001F5D         mov    [esp], eax      ; char *
text:00001F60         call    _puts
text:00001F65         mov    eax, 0
text:00001F6A         add    esp, 14h
text:00001F6D         pop    ebx
text:00001F6E         leave
text:00001F6F         retn
text:00001F6F _main     endp
text:00001F6F
text:00001F6F __text    ends
text:00001F6F
cstring:00001F70 ; =====
cstring:00001F70
cstring:00001F70 ; Segment type: Pure data
cstring:00001F70 __cstring    segment byte public 'DATA' use32
cstring:00001F70           assume cs:_cstring
cstring:00001F70           ;org 1F70h
cstring:00001F70 aWasIstSoftware db 'Was ist Software?',0 ; DATA XREF: _main+D↑o
cstring:00001F70 __cstring    ends
cstring:00001F70
symbol_stub:00001F82 ; =====
symbol_stub:00001F82
symbol_stub:00001F82 ; Segment type: Pure code
symbol_stub:00001F82 __symbol_stub    segment word public 'CODE' use32
symbol_stub:00001F82           assume cs:_symbol_stub
symbol_stub:00001F82           ;org 1F82h
```

Beispiel

Darstellung als Speicherauszug (Hexadezimaldarstellung der Byte-Werte)

```
00001FOA 00 00 6A 00 89 E5 83 E4 F0 83 EC 10 8B 5D 04 89 ..j.....].
00001F1A 1C 24 8D 4D 08 89 4C 24 04 83 C3 01 C1 E3 02 01 .$.M..L$.....
00001F2A CB 89 5C 24 08 8B 03 83 C3 04 85 C0 75 F7 89 5C ..\S.....u.\.
00001F3A 24 0C E8 09 00 00 00 89 04 24 E8 39 00 00 00 F4 $......$.9..
00001F4A 55 89 E5 53 83 EC 14 E8 00 00 00 00 5B 8D 83 1A U..S.....[...].
00001F5A 00 00 89 04 24 E8 23 00 00 00 B8 00 00 00 00 00 .....$.#.....
00001F6A 83 C4 14 5B C9 C3 57 61 73 20 69 73 74 20 53 6F ..{..Was ist So
00001F7A 66 74 77 61 72 65 3F 00 FF 25 1C 20 00 00 FF 25 ftware?%. ...%.
00001F8A 20 20 00 68 18 20 00 00 FF 25 14 20 00 00 90 ..h. ....%.
00001F9A 68 0C 00 00 00 E9 EA FF FF FF 68 00 00 00 E9 h.....h.....
00001FAA E0 FF FF FF 01 00 00 00 1C 00 00 00 00 00 ..... .
00001FB0 00 00 1C 00 00 00 00 00 00 00 00 00 00 02 00 ..... .
00001FCA 00 00 00 00 00 34 00 00 00 34 00 00 00 00 F9 0F .....4..4....
00001FDA 00 00 00 00 00 34 00 00 00 03 00 00 00 00 0C 00 .....4.....
00001FEA 01 00 10 00 01 00 00 00 00 00 00 00 00 00 00 ..... .
00002000 00 10 00 00 24 20 00 00 28 20 00 00 2C 20 00 00 .....$ ..( ... .
00002010 30 20 00 00 E8 31 00 00 00 00 00 00 E0 31 00 00 0..1....1..
00002020 E4 31 00 00 00 00 00 00 00 00 00 00 00 00 00 00 1.....
00002030 00 00 00 00 ..... .
00003000 11 40 64 79 6C 64 5F 73 74 75 62 5F 62 69 6E 64 .@dyld_stub_bind
00003010 65 72 00 51 72 14 90 00 72 1C 11 40 5F 65 78 69 er.Qr...r..@_exi
00003020 74 00 90 00 72 20 11 40 5F 70 75 74 73 00 90 00 t...r..@_puts...
00003030 02 02 5F 00 0C 73 74 61 72 74 00 4B 00 04 5F 00 .....start.K...
00003040 27 6D 61 69 6E 00 50 4E 58 41 72 67 00 55 65 6E 'main.PNXArg.Uen
00003050 76 69 72 6F 6E 00 67 00 02 6D 68 5F 65 78 65 63 viron.g..mh exec
00003060 75 74 65 5F 68 65 61 64 65 72 00 47 5F 70 72 6F ute_header.G_pro
00003070 67 6E 61 6D 65 00 6C 02 00 00 03 00 8C 1E 00 gname.l.....
00003080 03 00 CA 1E 00 00 02 63 00 5D 76 00 62 03 00 A4 .....c.]v.b...
00003090 20 00 03 00 A8 20 00 03 00 AC 20 00 03 00 B0 20 ..... .
000030A0 00 00 00 00 02 00 00 00 1E 04 00 00 8E 1F 00 00 ..... .
000030B0 10 00 00 00 0E 06 00 00 00 20 00 00 17 00 00 00 ..... .
000030C0 0F 09 00 00 24 20 00 00 1F 00 00 00 0F 09 00 00 .....$ .....
000030D0 28 20 00 00 27 00 00 00 0F 09 00 00 30 20 00 00 ( ..'.....0 ..
000030E0 33 00 00 00 03 00 10 00 00 10 00 00 47 00 00 00 3.....G...
000030F0 0F 09 00 00 2C 20 00 00 50 00 00 00 0F 01 00 00 .....P...
00003100 4A 1F 00 00 56 00 00 00 0F 01 00 00 0C 1F 00 00 J..V.....
00003110 5C 00 00 00 01 00 01 01 00 00 00 00 62 00 00 00 \.....b...
00003120 01 00 01 01 00 00 00 00 68 00 00 01 00 00 01 .....h.....
00003130 00 00 00 00 09 00 00 00 0A 00 00 00 00 00 00 00 00 40 .....@.....
00003140 00 00 00 40 09 00 00 00 0A 00 00 00 20 00 20 73 .....@..... s
00003150 74 75 62 20 68 65 6C 70 65 72 73 00 5F 70 76 61 tub helpers_pva
00003160 72 73 00 5F 4E 58 41 72 67 63 00 5F 4E 58 41 72 rs._NXArgc._NXAr
00003170 67 76 00 5F 5F 5F 70 72 6F 67 6E 61 6D 65 00 5F gv._ progame.
00003180 5F 6D 68 5F 65 78 65 63 75 74 65 5F 68 65 61 64 mh execute_head
```

Zahlendarstellung



Basis: Anzahl der verwendeten Ziffern in einem Stellenwertsystem

In Digitalschaltungen bietet sich die Basis 2 an: {0, 1}

$$(101101)_2 = (45)_{10} \leftarrow \text{Basis 10 i. d. R. weggelassen}$$
$$1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 4 \cdot 10^1 + 5 \cdot 10^0$$
$$1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 4 \cdot 10 + 5 \cdot 1$$

Die Zahlendarstellung zur Basis 2 nennt man **binär**.

Umrechnung von dezimal nach binär

Gesucht: Binärdarstellung der Dezimalzahl 37

$$(37)_{10} =$$

$$37 : 2 = 18 \text{ Rest } 1$$

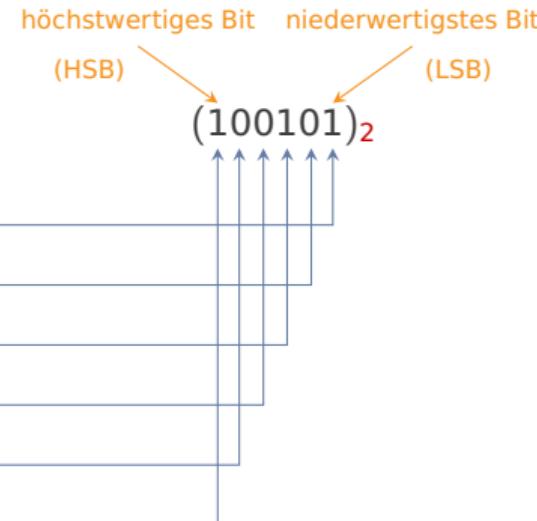
$$18 : 2 = 9 \text{ Rest } 0$$

$$9 : 2 = 4 \text{ Rest } 1$$

$$4 : 2 = 2 \text{ Rest } 0$$

$$2 : 2 = 1 \text{ Rest } 0$$

$$1 : 2 = 0 \text{ Rest } 1$$



Die niederwertigste Ziffer (Bit) gibt an, ob die Zahl gerade (0) oder ungerade (1) ist.

Hexadezimalzahlen

Stellenwertsystem zur Basis **16**: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

Grund: 8 Bit = 1 Byte, $2^4 = 16 \rightarrow$ Darstellung eines Bytes in zwei Ziffern

$$\underbrace{(1101}_{d} \underbrace{0110}_{6})_2$$

Umrechnung hexadezimal \rightarrow dezimal: $(d6)_{16} = 13 \cdot 16^1 + 6 \cdot 16^0 = (214)_{10}$

Umrechnung dezimal \rightarrow hexadezimal: Wiederholt durch **16** teilen und Reste notieren.

Konventionen

Verwechslungsgefahr, falls Hexadezimalzahlen keine Ziffern A–F enthalten.

Kennzeichnung durch Präfix **0x** oder Suffix **h**. Bsp.: $0x0400 = 400h = 1024$

Hörsaalfragen

Wir verwenden das ARSnova Audience Response System.

- Benutzen Sie Ihr Smartphone.
- Die URL lautet: <https://arsnova.uibk.ac.at>
- Der Zugangsschlüssel lautet: **24 82 94 16**
- Oder scannen Sie den QR-Kode.



Fragen

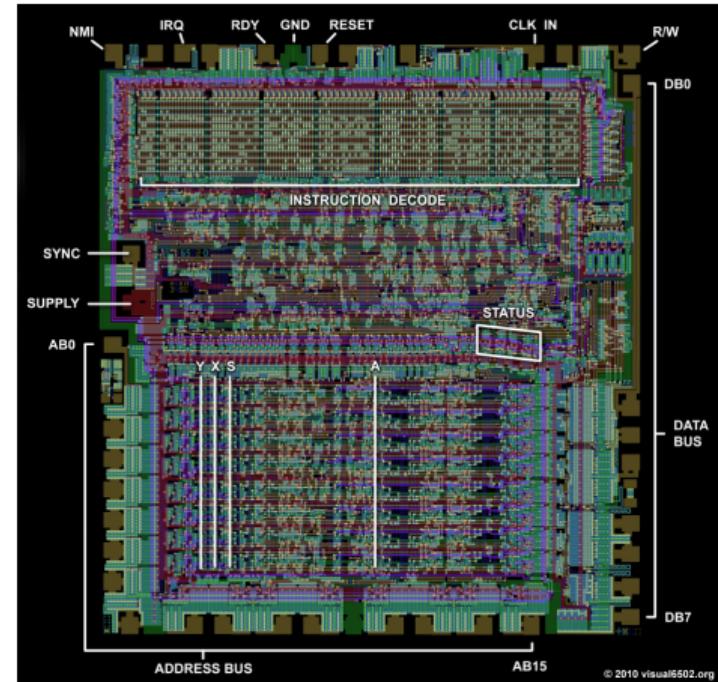
1. Welche Ziffernfolge entspricht der Dezimalzahl 13 in Binärdarstellung ?
2. Welche Zeichenfolge entspricht der Dezimalzahl 13 im Hexadezimalsystem ?

Beispiel einer historischen CPU

MOS 6502 (1975)

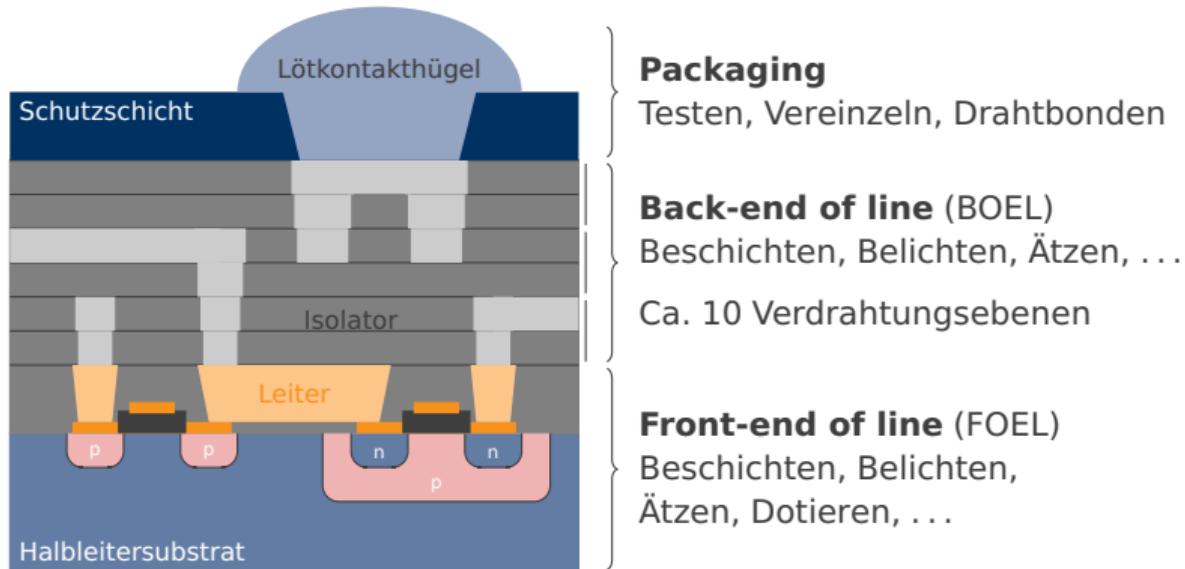


Bild: visual6502.org



Aufbau von Mikrochips

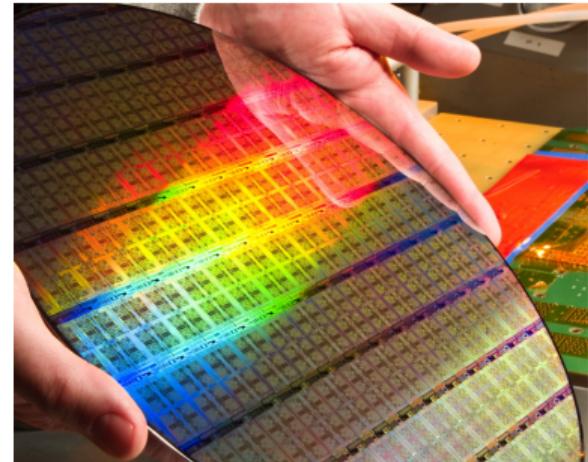
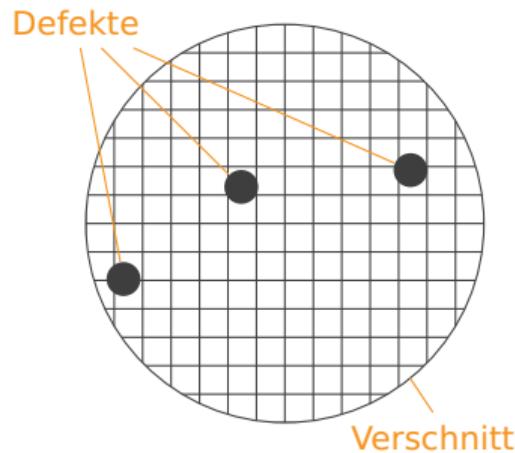
Schnitt durch einen “Die” (engl.), Herstellung im Planarprozess



Moderne Produktionsprozesse haben bis zu 500 Schritte.

Wafer

(engl. für „Waffel“ oder „Oblate“)



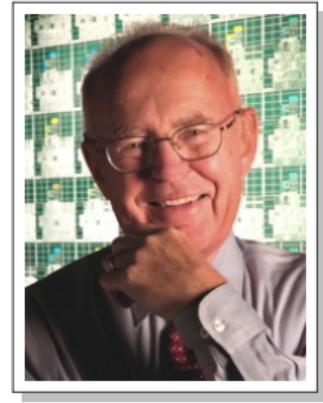
- Wafer-Durchmesser steigt: 50 mm (1970) – 300 mm (2002)
- Strukturgröße sinkt: 10 μm (1971) – 5 nm (2020)
- Die-Fläche steigt: 12 mm^2 (1971) – 815 mm^2 (2017)

Bildquelle: IBM (300 mm-Wafer)

Gordon Moore

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase."

Electronics Magazine, 1965

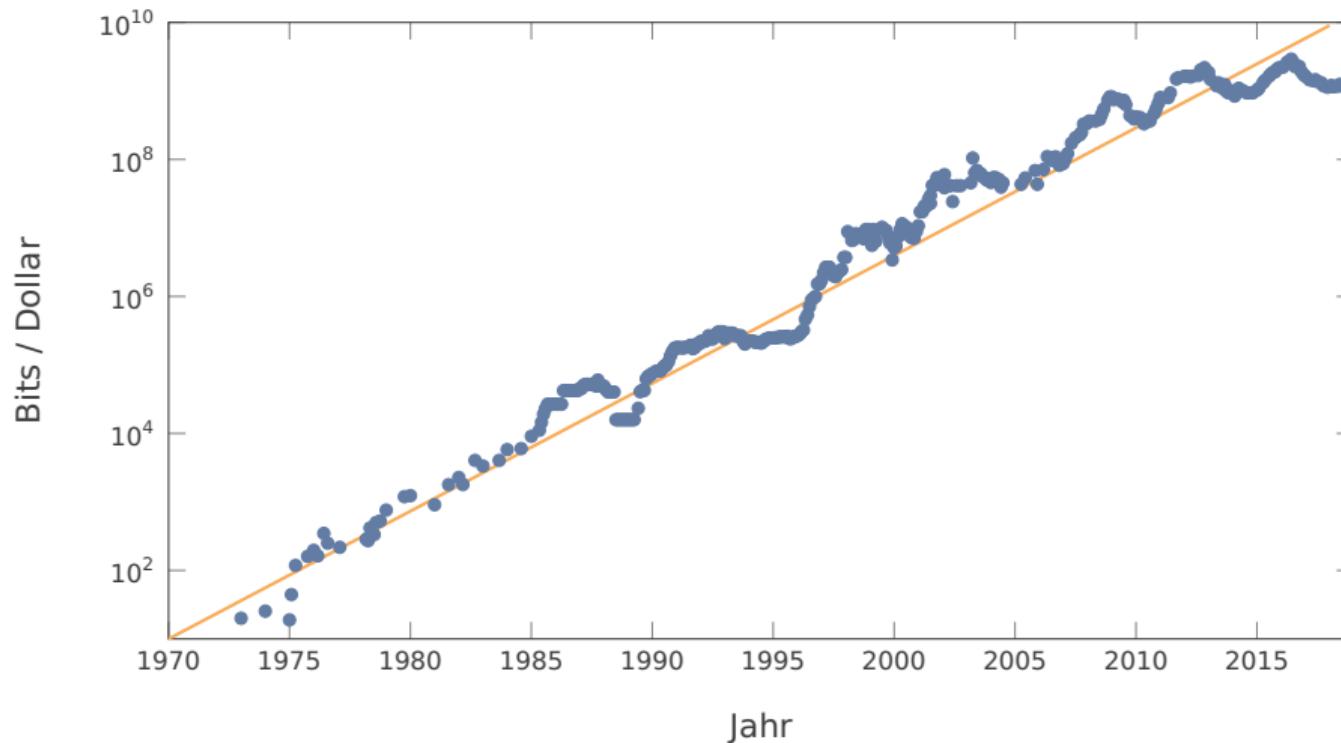


2005

Gordon E. Moore, geb. 1929, Mitgründer von Intel
Urheber des Moore'schen Gesetzes

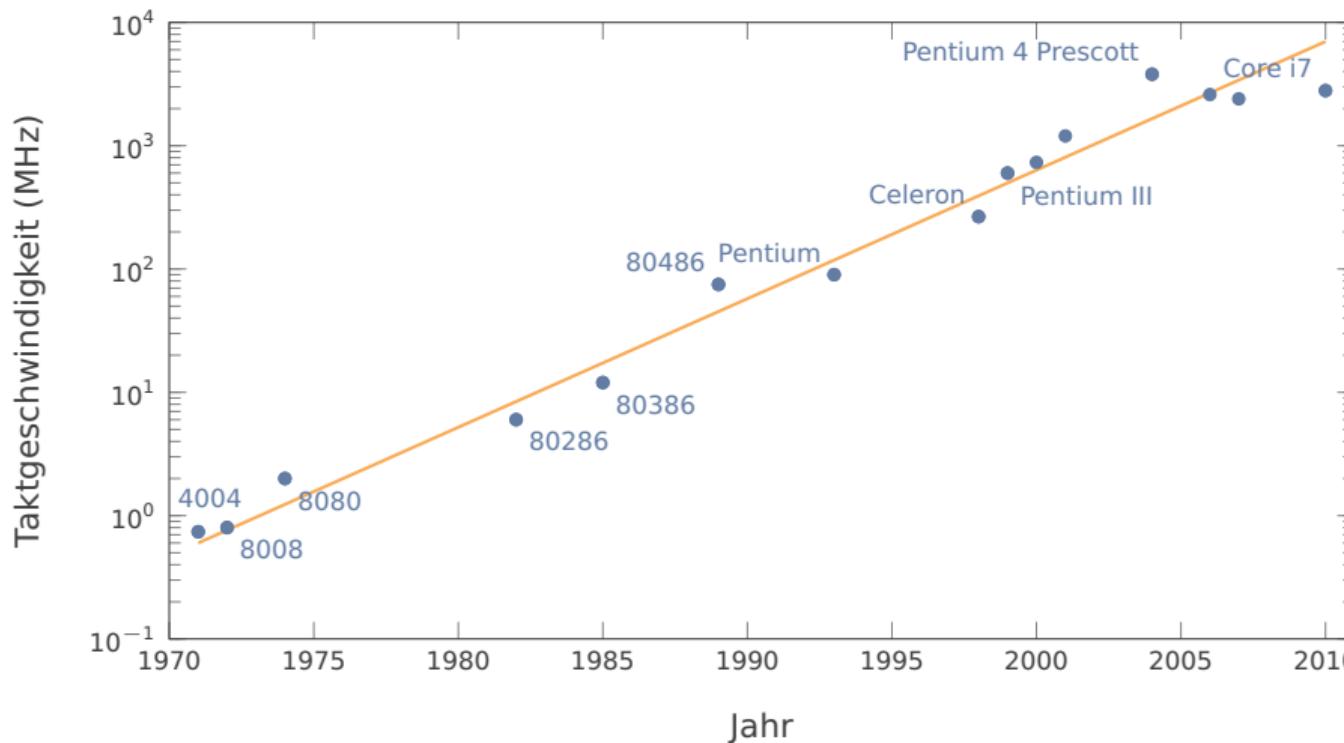
Bild: Intel

Entwicklung der Größe des Arbeitsspeichers



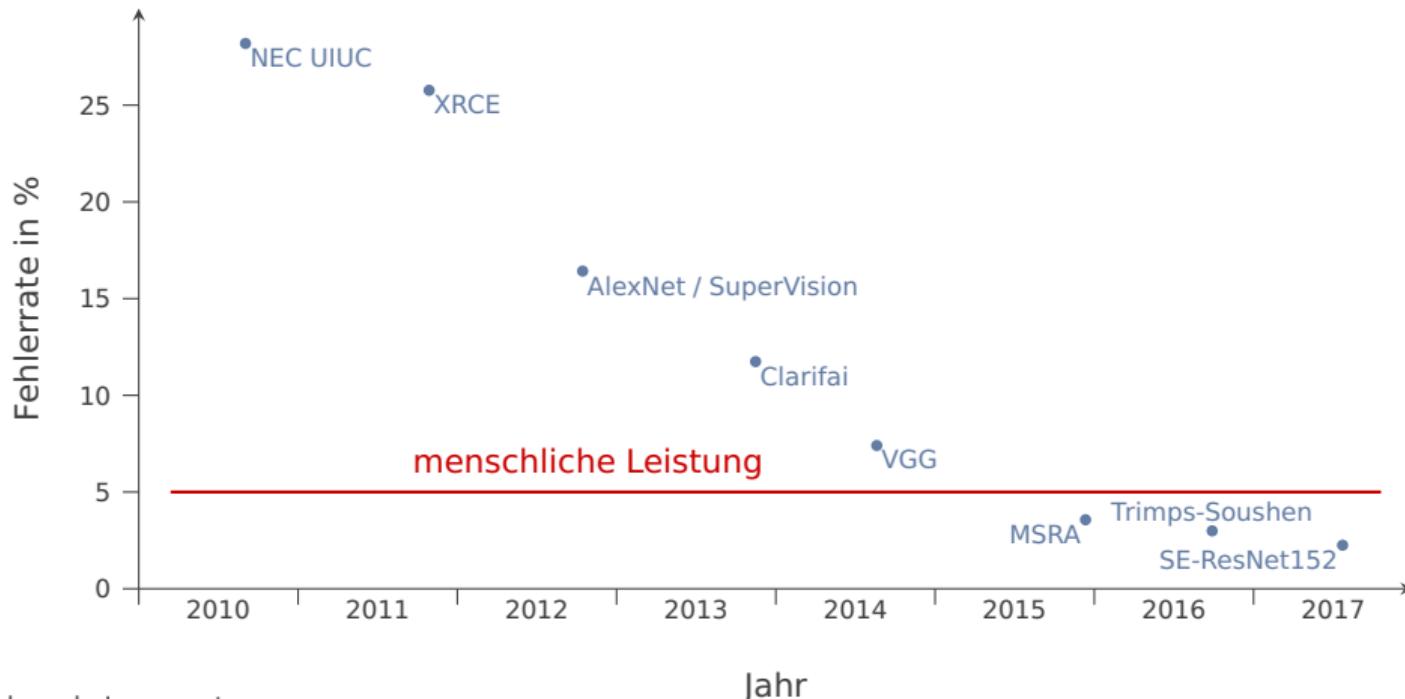
Datenquelle: John C. McCallum 2018, eigene Darstellung

Taktgeschwindigkeit von Intel-Prozessoren



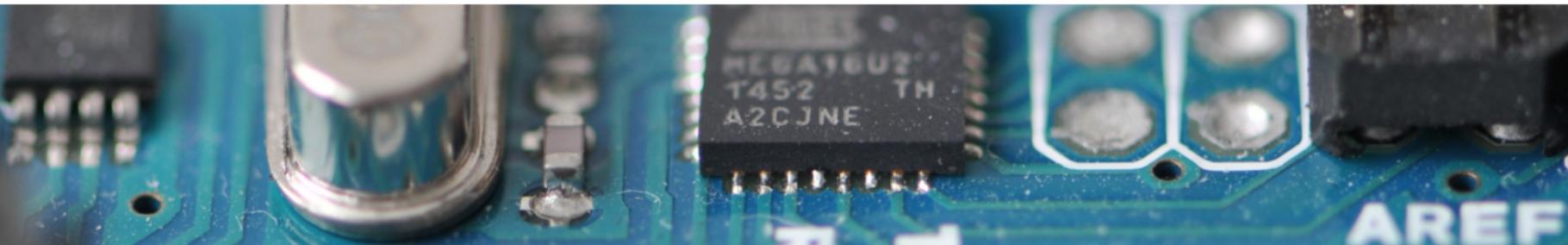
Datenquelle: Wikipedia 2016

Fortschritt in der Bilderkennung



Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Kombinatorische Logik I

Univ.-Prof. Dr.-Ing. Rainer Böhme

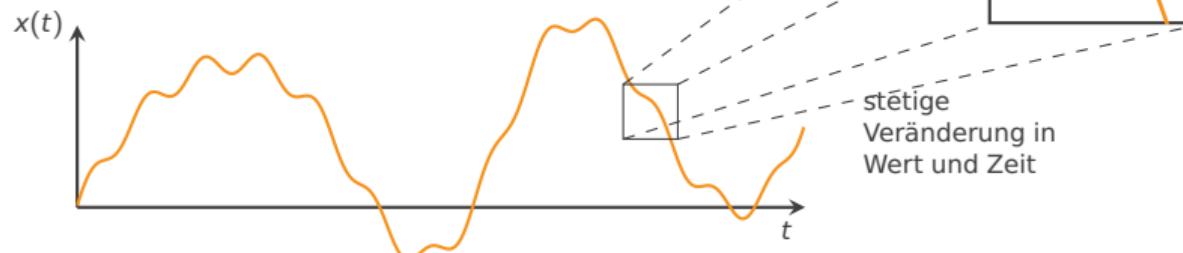
Wintersemester 2021/22 · 13. Oktober 2021

Gliederung heute

- 1. Grundlagen der Digitaltechnik**
2. Boolesche Algebra
3. Realisierung in Schaltungen

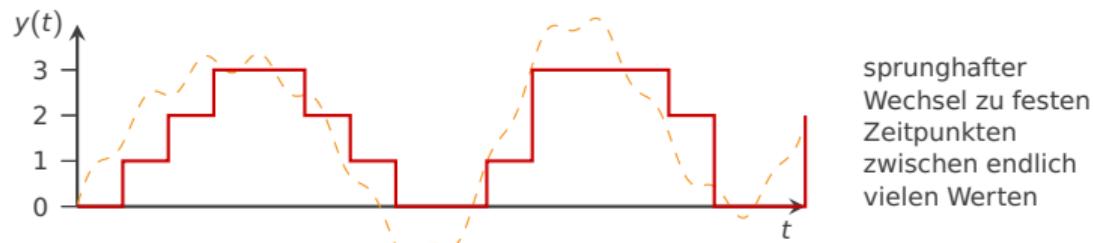
Analoge und digitale Signale

Analogtechnik: kontinuierliche Signale



stetige
Veränderung in
Wert und Zeit

Digitaltechnik: diskrete Signale (oft auch binär)

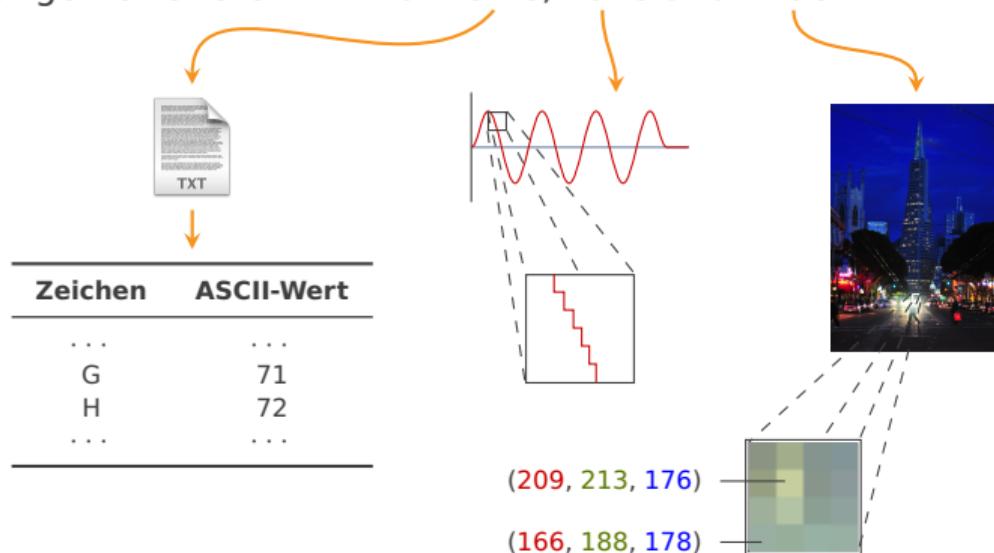


sprunghafter
Wechsel zu festen
Zeitpunkten
zwischen endlich
vielen Werten

Digitale Daten

Alle Arten von Daten werden digital als **diskrete Zahlen** gespeichert.

Zuordnungen existieren z. B. für Texte, Töne und Bilder.



Digitalrechner können nur digitale Daten verarbeiten.

Vergleich von Analog- und Digitaltechnik

Analogrechner

- + Multiplikation, Addition und Filter leicht realisierbar
- + geringer Flächenbedarf
- + sehr schnell
- nichtlineare Bauteile
- niedrige Genauigkeit
- temperaturabhängig
- Speicherung von Daten schwierig

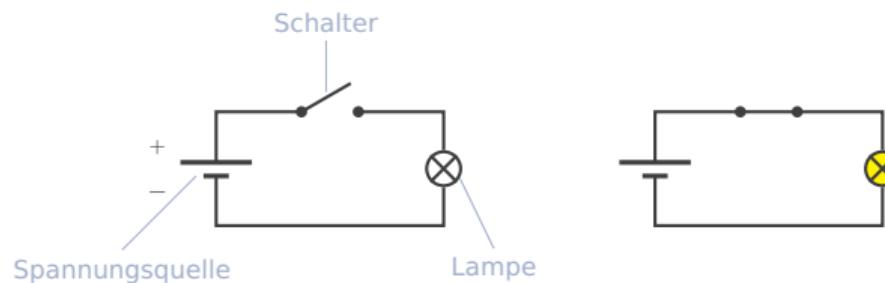
Digitalrechner

- + weniger störanfällig (z. B. Rauschen)
- + beliebig hohe Genauigkeit erreichbar
- + exakte Reproduktion und Übertragung von Daten
- + einfacher, modularer Entwurf
- oft hoher Flächenbedarf
- hoher Energieverbrauch

Kompromiss zwischen Analog- und Digitaltechnik: Hybridrechner

Digitaltechnik

Darstellung als elektrische Schaltung

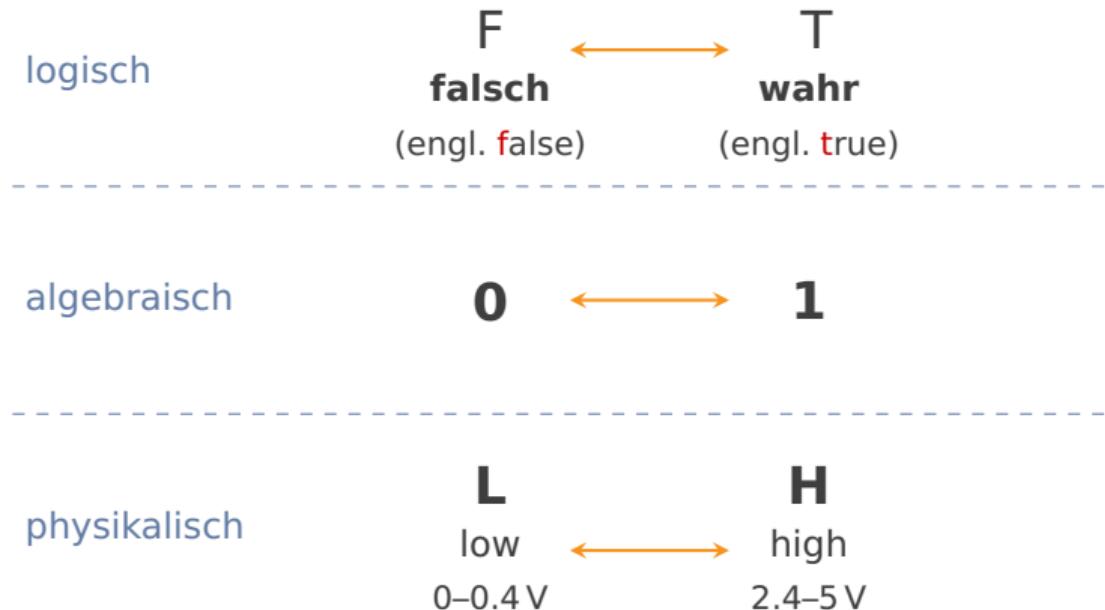


Zwei Zustände:

- 0. Strom fließt nicht** (Lampe leuchtet nicht)
- 1. Strom fließt** (Lampe leuchtet)

Varianten der Binärdarstellung

Interpretation der digitalen Zustände



Beispiel: positive Logik mit TTL-Ausgangspegeln

Konventionen in der kombinatorischen Logik

- Präferenz der digitalen Zustandsmenge $\{0, 1\}$
- Realisierung elementarer Operatoren durch **Gatter**
- Realisierung komplexer Funktionen durch Verschalten von Gattern
- Vektorschreibweise für mehrstellige digitale Zustände:

$$\mathbf{x} = (x_1, x_2, \dots, x_n) \in \{0, 1\}^n$$

- Die **Dimension** n ist dabei oft implizit.

Vorsicht!

InformatikerInnen zählen mitunter auch von 0 bis $n - 1$
(angelehnt z. B. an Felder in den Programmiersprachen C und Java).

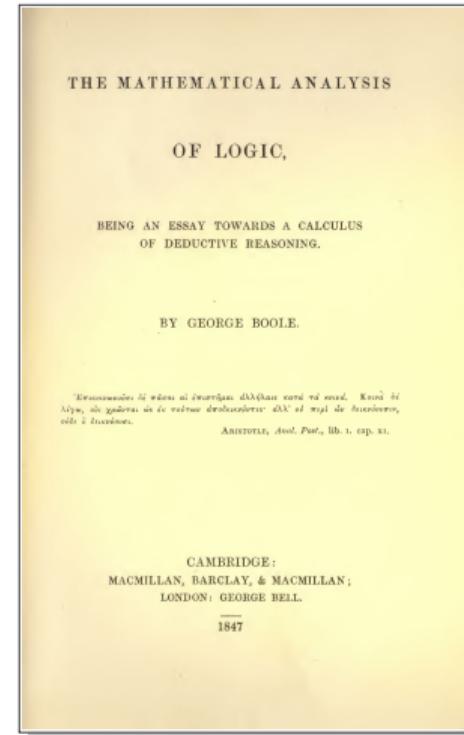
Gliederung heute

1. Grundlagen der Digitaltechnik
2. **Boolesche Algebra**
3. Realisierung in Schaltungen

Boolesche Algebra

nach **George Boole** (1815–1864)

- Verbindung von Philosophie (Logik) und Mathematik (Rechenregeln)
- Grundlage für heutige Rechner-Hardware
- Dient dem Entwurf, der Beschreibung, Berechnung und Vereinfachung von Schaltungen und Schaltwerken für die Verarbeitung binärer Größen
- gleichzeitig Grundlage der Theoretischen Informatik
→ *Einführung in die Theoretische Informatik*



Operatoren

- Gegeben sei die Boolesche Menge \mathbb{B} oft $\mathbb{B} = \{0, 1\}$
- Definition von Operatoren auf Variablen $x_1, x_2 \in \mathbb{B}$ z.B. $+, \cdot, -$
- Vollständige Bestimmung durch **Wahrheitstabelle**
- Die Schreibweise der Operatoren kann variieren.

Achten Sie jedoch auf Konsistenz bei eigener Verwendung !

Elementare Operatoren

OR-Operator

logische Summe (ODER)

$+$ \vee

x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1

Das Ergebnis ist 1, falls
mindestens ein Operand
den Wert 1 annimmt.

AND-Operator

logisches Produkt (UND)

\cdot $*$ \wedge

x_1	x_2	x_1 AND x_2
0	0	0
0	1	0
1	0	0
1	1	1

Das Ergebnis ist 1, genau
dann wenn **beide** Operanden
den Wert 1 annehmen.

NOT-Operator

Invertierung (NICHT)

\bar{x} $\neg x$ \neg

x	$\text{NOT } x$
0	1
1	0

Das Ergebnis ist 1, genau
dann wenn der Operand
den Wert 0 annimmt.

Boolesche Algebra

(im engeren Sinne)

Definition

Die Kombination der Booleschen Menge \mathbb{B} mit den Operatoren OR, AND und NOT wird als **boolesche Algebra** bezeichnet.

Schreibweisen

- $(\mathbb{B}, \text{AND}, \text{OR}, \text{NOT})$
- $(\mathbb{B}, \cdot, +, -) = (\{0, 1\}, \cdot, +, -)$
- $\mathbb{B}(\wedge, \vee, \neg)$ auch $B(\wedge, \vee, \neg)$

Ähnlich wie in der Schulalgebra kann der Punkt-Operator (AND) bei Ausdrücken auch weggelassen werden: $x_1 \cdot x_2 \Leftrightarrow x_1 x_2$

Es gilt Punkt vor Strich.

Axiome

der Booleschen Algebra zur Umformung logischer Gleichungen

Kommutativitat

$$x_1 + x_2 = x_2 + x_1 \quad (1)$$

$$x_1 \cdot x_2 = x_2 \cdot x_1 \quad (2)$$

Distributivitat

$$x_1 \cdot (x_2 + x_3) = (x_1 \cdot x_2) + (x_1 \cdot x_3) \quad (3)$$

$$x_1 + (x_2 \cdot x_3) = (x_1 + x_2) \cdot (x_1 + x_3) \quad (4)$$

Neutrale Elemente

$$0 + x = x \quad (5)$$

$$1 \cdot x = x \quad (6)$$

Komplementres Element

$$x + \bar{x} = 1 \quad (7)$$

$$x \cdot \bar{x} = 0 \quad (8)$$

Sätze

abgeleitet aus den Axiomen

Idempotenz

$$x + x = x \tag{9}$$

$$x \cdot x = x \tag{10}$$

Assoziativität

$$x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3 \tag{11}$$

$$x_1 \cdot (x_2 \cdot x_3) = (x_1 \cdot x_2) \cdot x_3 \tag{12}$$

Absorption

$$x_1 + (x_1 \cdot x_2) = x_1 \tag{13}$$

$$x_1 \cdot (x_1 + x_2) = x_1 \tag{14}$$

Substitution

$$x + 1 = 1 \tag{15}$$

$$x \cdot 0 = 0 \tag{16}$$

Doppelnegation

$$\overline{\overline{x}} = x \tag{17}$$

Weitere Gesetzmäßigkeiten

Komplementäre Werte $\bar{0} = 1$ und $\bar{1} = 0$

Abgeschlossenheit

Boolesche Operationen liefern nur boolesche Werte als Ergebnis.

Dualität

Für jede aus Axiomen ableitbare Aussage existiert eine duale Aussage.

Diese erhält man durch Tausch der Operatoren $+$ und \cdot sowie der Werte 0, 1.

De Morgansche Gesetze (folgende Folien)

Die De Morganschen Gesetze

Das 1. De Morgansche Gesetz lautet: $\overline{x_1 \cdot x_2} = \overline{x_1} + \overline{x_2}$

x_1	x_2	$x_1 \cdot x_2$	$\overline{x_1 \cdot x_2}$	$\overline{x_1}$	$\overline{x_2}$	$\overline{x_1} + \overline{x_2}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Die De Morganschen Gesetze (Forts.)

Das 2. De Morgansche Gesetz lautet: $\overline{x_1 + x_2} = \overline{x_1} \cdot \overline{x_2}$

x_1	x_2	$x_1 + x_2$	$\overline{x_1 + x_2}$	$\overline{x_1}$	$\overline{x_2}$	$\overline{x_1} \cdot \overline{x_2}$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Die De Morganschen Gesetze (Forts.)

Negation mithilfe der De Morganschen Gesetze

Negation von Termen erfolgt durch Tausch der Operatoren + und · sowie Komplementierung aller Variablen.

Beispiel:

$$\overline{(x_1 + x_2) \cdot \overline{x_3}}$$

=

$$(\overline{x_1} \cdot \overline{x_2}) + x_3$$

Schaltfunktionen

Definition

Eine Funktion $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ mit $n, m \geq 1$ heißt **Schaltfunktion**.

Spezialfall

Eine Schaltfunktion mit $m = 1$ heißt n -stellige **Boolesche Funktion**.

Beschreibung Boolescher Funktionen:

1. eindeutig mit (sortierter) **Wahrheitstabelle**
 2. kompakter, aber **nicht eindeutig** mit **Booleschem Ausdruck** (aus Booleschen Variablen und Operationen)
- Jede **Schaltfunktion** kann durch m **Boolesche Funktionen** zusammengesetzt werden.

Boolesche Funktionen

Wie viele n -stellige Boolesche Funktionen gibt es?

- Kombination aller 2^n n -Tupel aus $\{0, 1\}$ der Argumente mit den Werten $\{0, 1\}$: $2^{(2^n)}$
- Für $n = 1$: $f_0(x) = 0$ Kontradiktion (0-stellig)
 $f_1(x) = x$ Identität
 $f_2(x) = -x$ Negation
 $f_3(x) = 1$ Tautologie (0-stellig)
- Für $n = 2$: 16 zweistellige Boolesche Funktionen

Einige davon sind lediglich auf zwei Argumente erweiterte null- oder einstellige Boolesche Funktionen: $f_0, f_3, f_5, f_{10}, f_{12}, f_{15}$ (folgende Folien)

Zweistellige Boolesche Funktionen

$x_2 =$	0 1 0 1	Term	Bezeichnung	Sprechweise
$x_1 =$	0 0 1 1			
f_0	0 0 0 0	0	Nullfunktion	
f_1	0 0 0 1	$x_1 x_2$	Konjunktion	x_1 AND x_2
f_2	0 0 1 0	$x_1 \bar{x}_2$	1. Differenz	x_1 AND NOT x_2
f_3	0 0 1 1	x_1	1. Identität	
f_4	0 1 0 0	$\bar{x}_1 x_2$	2. Differenz	NOT x_1 AND x_2
f_5	0 1 0 1	x_2	2. Identität	
f_6	0 1 1 0	$\bar{x}_1 x_2 + x_1 \bar{x}_2$	Antivalenz	x_1 XOR x_2
f_7	0 1 1 1	$x_1 + x_2$	Disjunktion	x_1 OR x_2

Zweistellige Boolesche Funktionen (Forts.)

$x_2 =$	$0 \ 1 \ 0 \ 1$	Term	Bezeichnung	Sprechweise
$x_1 =$	$0 \ 0 \ 1 \ 1$			
f_8	1 0 0 0	$\overline{x_1} + x_2$	Negatdisjunktion	$x_1 \text{ NOR } x_2$
f_9	1 0 0 1	$(\overline{x_1} + x_2)(x_1 + \overline{x_2})$	Äquivalenz	$x_1 \Leftrightarrow x_2$
f_{10}	1 0 1 0	$\overline{x_2}$	2. Negation	$\text{NOT } x_2$
f_{11}	1 0 1 1	$x_1 + \overline{x_2}$	2. Implikation	$x_2 \Rightarrow x_1$
f_{12}	1 1 0 0	$\overline{x_1}$	1. Negation	$\text{NOT } x_1$
f_{13}	1 1 0 1	$\overline{x_1} + x_2$	1. Implikation	$x_1 \Rightarrow x_2$
f_{14}	1 1 1 0	$\overline{x_1x_2}$	Negatkonjunktion	$x_1 \text{ NAND } x_2$
f_{15}	1 1 1 1	1	Einsfunktion	

Vollständige Operatorenensysteme

1. Alle Booleschen Funktionen können mithilfe der

- **Disjunktion** (+, OR),
- **Konjunktion** (\cdot , AND) und
- **Negation** ($-$, NOT)

dargestellt werden.

→ Boolesche Basis

2. Alle Booleschen Funktionen können

- **entweder** mithilfe der Negation und der Konjunktion
- **oder** mithilfe der Negation und der Disjunktion

dargestellt werden.

→ De Morgan-Basis

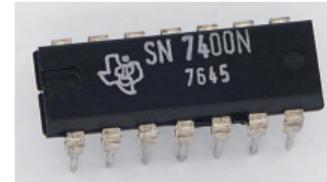
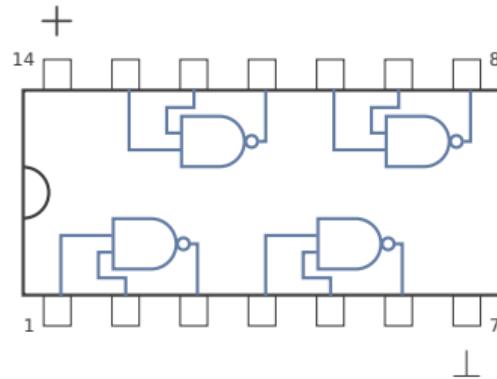
3. Alle Booleschen Funktionen können

- entweder mithilfe der **NAND-Verknüpfung**
- oder mithilfe der **NOR-Verknüpfung**

dargestellt werden.

Der 7400er-Chip

Chip der 7400er-Serie mit 4 NAND-Gattern:



Bildquelle: Wikimedia Commons

Gliederung heute

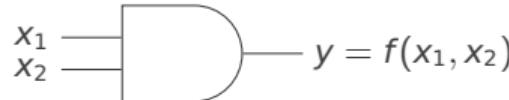
- 1. Grundlagen der Digitaltechnik**
- 2. Boolesche Algebra**
- 3. Realisierung in Schaltungen**

Technische Realisierung digitaler Systeme

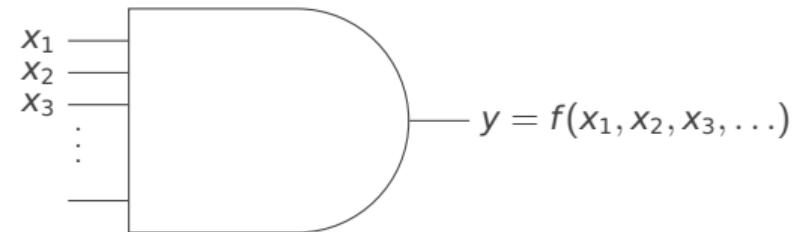
Gatter sind elektronische Schalter zur Verknüpfung binärer Argumente.

- Aufbau aus einfachen elektronischen Bauteilen: Widerständen, Dioden, Transistoren
- Verhalten realisiert Booleschen Funktionen mit $n \geq 1$ **Eingängen**, je einer pro Argument $(x_1, x_2, x_3, \dots) \in \{0, 1\}^n$, und einem **Ausgang** $y \in \{0, 1\}$

Schalsymbol mit zwei Eingängen
(hier: AND-Gatter)



Verallgemeinerung mit n Eingängen
z. B. $f(x_1, x_2, x_3) = f(x_1, f(x_2, x_3))$ usw.



Konvention: Die Form des Schalsymbols lässt auf dessen Funktion schließen.

Wichtige Boolesche Funktionen als Gatter

Eingabe		Ausgabe					
		elementar			kombiniert		
x_1	x_2	$x_1 + x_2$	$x_1 x_2$	\bar{x}_1	$\bar{x}_1 x_2 + x_1 \bar{x}_2$	$\bar{x}_1 + x_2$	$\bar{x}_1 x_2$
0	0	0	0	1	0	1	1
0	1	1	0	1	1	0	1
1	0	1	0	0	1	0	1
1	1	1	1	0	0	0	0

Schaltsymbol



Bezeichnung OR AND NOT XOR NOR NAND

Darstellungsvarianten

Realisierung der Booleschen Funktion $\overline{x_1} + x_2$ mit Gattern:



Beispiel einer logischen Schaltung

Gesucht Schaltung, die 1 ausgibt, wenn einer oder zwei von drei Eingängen x_1, x_2, x_3 den Wert 1 annehmen.

Wahrheitstabelle:

x_1	x_2	x_3	y
0	0	0	0
1	0	0	1
0	1	0	1
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	1
1	1	1	0

Beispiel einer logischen Schaltung

Gesucht Schaltung, die 1 ausgibt, wenn einer oder zwei von drei Eingängen x_1, x_2, x_3 den Wert 1 annehmen.

Wahrheitstabelle, Boolesche Funktion:

x_1	x_2	x_3	y
0	0	0	0
1	0	0	1 ✓
0	1	0	1 ✓
1	1	0	1 ✓
0	0	1	1 ✓
1	0	1	1 ✓
0	1	1	1 ✓
1	1	1	0

$$y = \overline{x_1}x_2 + \overline{x_1}x_3 + x_1\overline{x_2} + x_1\overline{x_3}$$

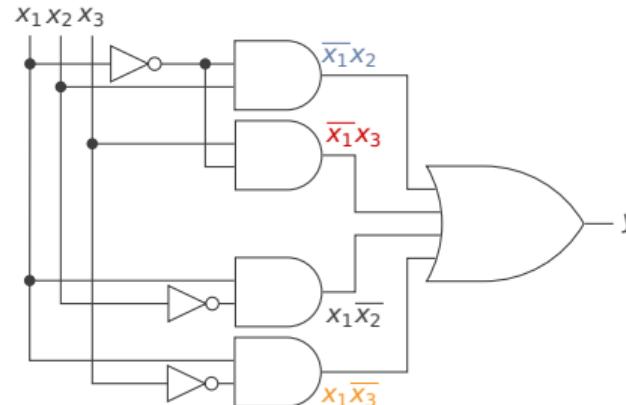
Beispiel einer logischen Schaltung

Gesucht Schaltung, die 1 ausgibt, wenn einer oder zwei von drei Eingängen x_1, x_2, x_3 den Wert 1 annehmen.

Wahrheitstabelle, Boolesche Funktion, Realisierung:

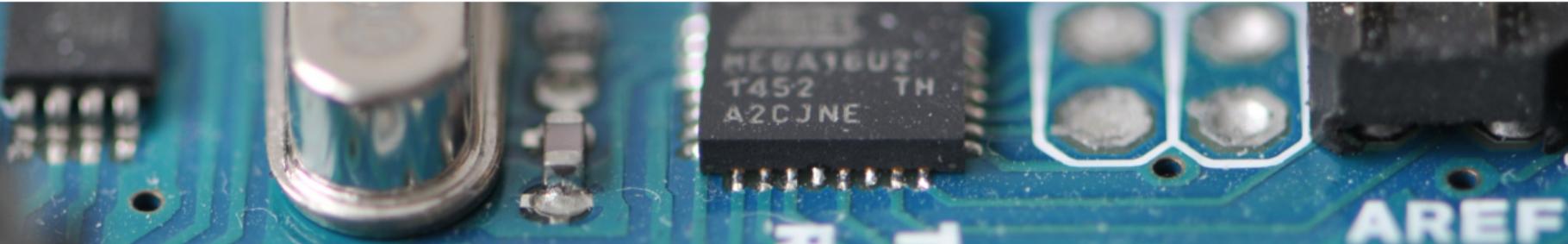
x_1	x_2	x_3	y
0	0	0	0
1	0	0	1 ✓
0	1	0	1 ✓
1	1	0	1 ✓
0	0	1	1 ✓
1	0	1	1 ✓
0	1	1	1 ✓
1	1	1	0

$$y = \overline{x_1}x_2 + \overline{x_1}x_3 + x_1\overline{x_2} + x_1\overline{x_3}$$



Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Kombinatorische Logik II

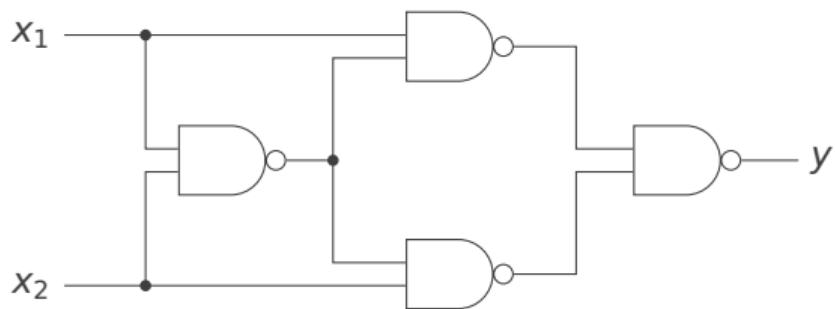
Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2021/22 · 20. Oktober 2021

Was tut diese Schaltung ?



24 82 94 16



x_1	x_2	y
0	0	
0	1	
1	0	
1	1	

Bitte wählen Sie die passende Spalte für y in ARSnova.

Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

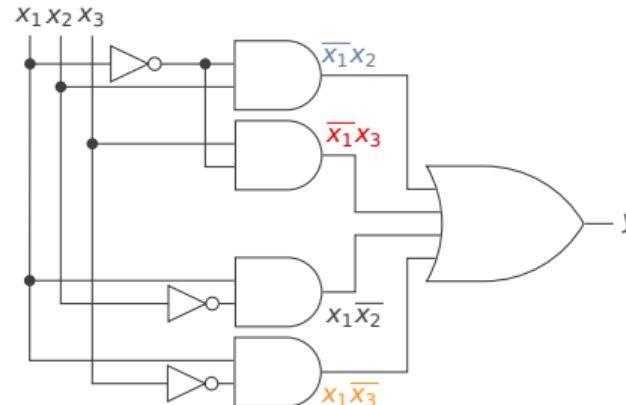
Beispiel einer logischen Schaltung (W)

Gesucht Schaltung, die 1 ausgibt, wenn einer oder zwei von drei Eingängen x_1, x_2, x_3 den Wert 1 annehmen.

Wahrheitstabelle, Boolesche Funktion, Realisierung:

x_1	x_2	x_3	y
0	0	0	0
1	0	0	1 ✓
0	1	0	1 ✓
1	1	0	1 ✓
0	0	1	1 ✓
1	0	1	1 ✓
0	1	1	1 ✓
1	1	1	0

$$y = \overline{x_1}x_2 + \overline{x_1}x_3 + x_1\overline{x_2} + x_1\overline{x_3}$$



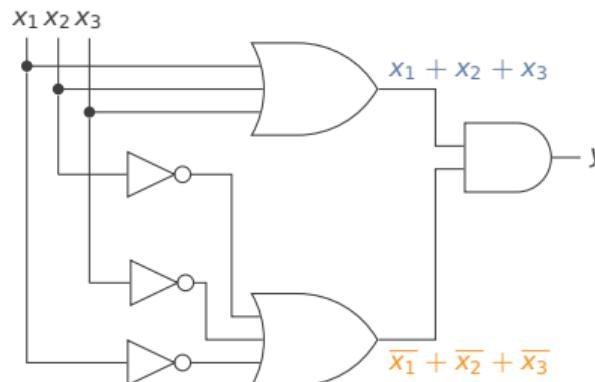
Beispiel einer logischen Schaltung (Forts.)

Gesucht Schaltung, die 1 ausgibt, wenn einer oder zwei von drei Eingängen x_1, x_2, x_3 den Wert 1 annehmen.

Alternative:

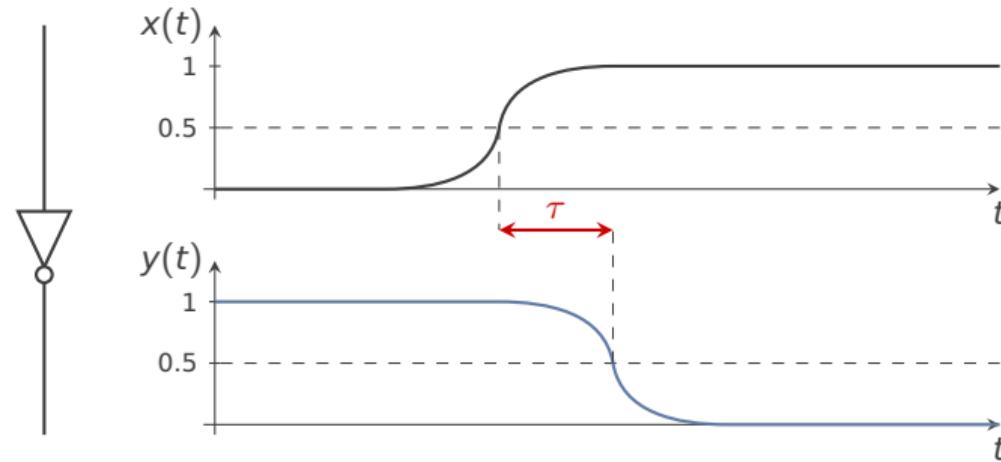
x_1	x_2	x_3	y
0	0	0	0
1	0	0	1
0	1	0	1
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	1
1	1	1	0

$$y = (x_1 + x_2 + x_3)(\overline{x_1} + \overline{x_2} + \overline{x_3})$$



Zeitverhalten eines Gatters

Gatter sind physische Bausteine. Sie verhalten sich nicht ideal.



Das Ausgangssignal reagiert verzögert

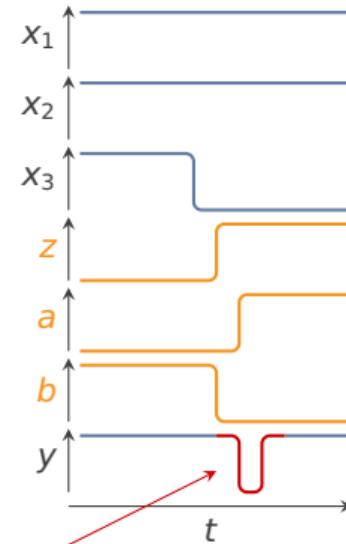
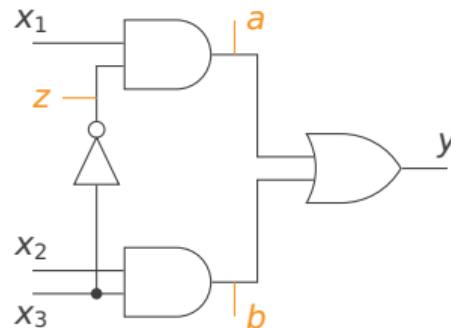
Die Verzögerung τ ist definiert als Dauer zwischen den Zeitpunkten der Überschreitung des 50 %-Pegels an Ein- bzw. Ausgang.

Störimpulse durch Laufzeiteffekte

Beispiel: Eingang x_3 steuert, ob x_1 oder x_2 am Ausgang y anliegt.

x_1	x_2	x_3	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

$$y = \bar{x}_3x_1 + x_3x_2$$

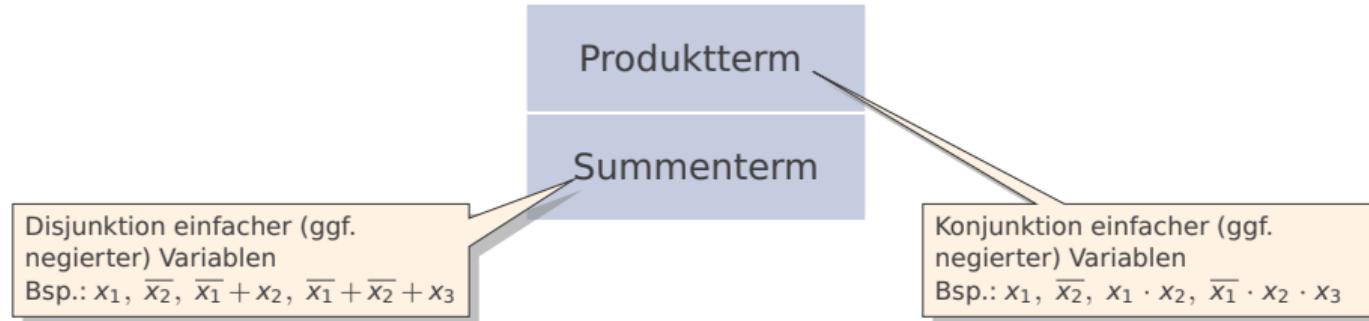


„statischer Eins-Hazard“

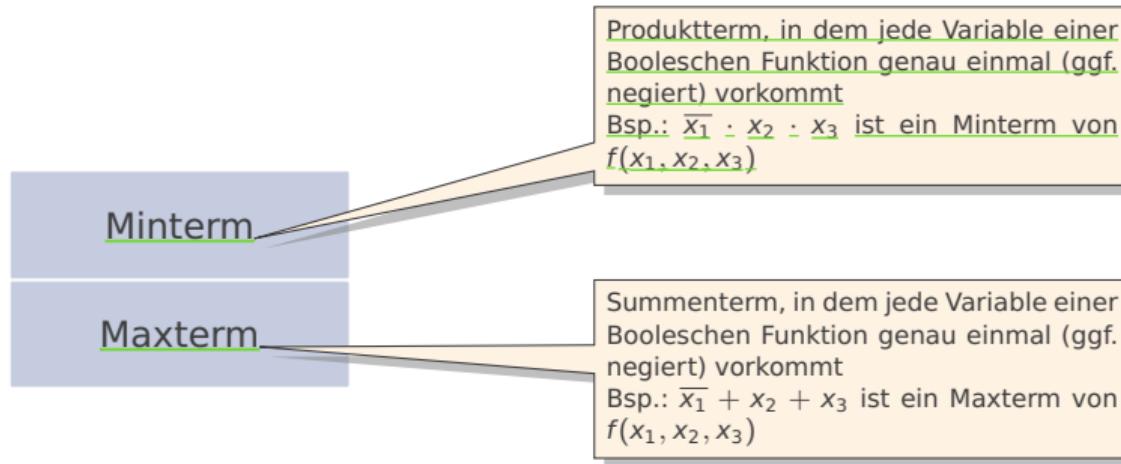
Gliederung heute

- 0. Konfrontation mit der Realität
- 1. Kanonische Darstellungen**
- 2. Minimierung
- 3. Typische Schaltnetze

Systematik der Darstellung Boolescher Funktionen



Systematik der Darstellung Boolescher Funktionen



Systematik der Darstellung Boolescher Funktionen

Disjunktion von Produkttermen
(Summe von Produkten, DNF)
Bsp.: $(x_1 \cdot x_2) + (\bar{x}_1 \cdot x_2 \cdot x_3)$

Disjunktive
Normalform

Konjunktion von Summentermen
(Produkt von Summen, KNF)
Bsp.: $(x_1 + x_2) \cdot (\bar{x}_1 + x_2 + x_3)$

Konjunktive
Normalform

Systematik der Darstellung Boolescher Funktionen

Eindeutige Darstellung einer Booleschen Funktion f als Disjunktion von Mintermen
Bsp.:

$$(\overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}) + (x_1 \cdot \overline{x_2} \cdot x_3) + (x_1 \cdot x_2 \cdot \overline{x_3})$$

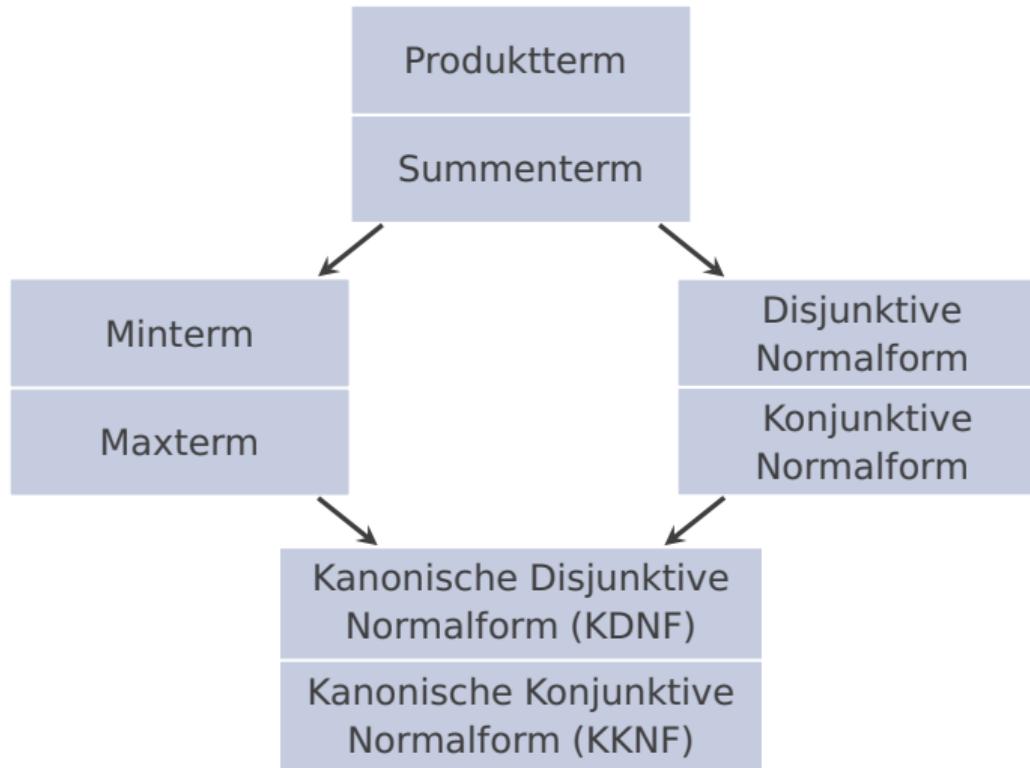
Eindeutige Darstellung einer Booleschen Funktion f als Konjunktion von Maxtermen
Bsp.:

$$(\overline{x_1} + \overline{x_2}) \cdot (\overline{x_1} + x_2) \cdot (x_1 + \overline{x_2})$$

Kanonische Disjunktive
Normalform (KDNF)

Kanonische Konjunktive
Normalform (KKNF)/

Systematik der Darstellung Boolescher Funktionen



Sätze zur Darstellung Boolescher Funktionen

1. **Jede** Boolesche Funktion lässt sich als **genau eine KDNF** (Disjunktion von Mintermen) darstellen.
2. **Jede** Boolesche Funktion lässt sich als **genau eine KKNF** (Konjunktion von Maxtermen) darstellen.
3. **Jede KDNF** kann in eine KKNF umgewandelt werden.
4. **Jede KKNF** kann in eine KDNF umgewandelt werden.
5. Aufgrund der **Dualität** gilt:

$$\text{KKNF}(f(x_1, x_2, \dots, x_n)) = \overline{\text{KDNF}(\overline{f(x_1, x_2, \dots, x_n)})}$$

und

$$\text{KDNF}(f(x_1, x_2, \dots, x_n)) = \overline{\text{KKNF}(\overline{f(x_1, x_2, \dots, x_n)})}$$

Bildung der KDNF (Disjunktion von Mintermen)

aus der Wahrheitstabelle einer n -stelligen Booleschen Funktion

- **Idee:** Summe nimmt den Wert **1** an, wenn mindestens ein Summand **1** ist.
- Für jede Zeile der Wahrheitstabelle mit $f(x_1, \dots, x_n) = \textcolor{red}{1}$ wird einer der Minterme ermittelt.
- Variable x_i wird negiert, wenn in der entsprechenden Zelle der Wert der Variable 0 ist.

Beispiel

x_1	x_2	x_3	$f(\mathbf{x})$
:	:	:	:
1	0	1	1
:	:	:	:

$\longrightarrow x_1 \cdot \overline{x_2} \cdot x_3$

Bildung der KKNF (Konjunktion von Maxtermen)

aus der Wahrheitstabelle einer n -stelligen Booleschen Funktion

- **Idee:** Produkt nimmt den Wert **0** an, wenn mindestens ein Faktor **0** ist.
- Für jede Zeile der Wahrheitstabelle mit $f(x_1, \dots, x_n) = 0$ wird einer der Maxterme ermittelt.
- Variable x_i wird negiert, wenn in der entsprechenden Zelle der Wert der Variable 1 ist.

Beispiel			
x_1	x_2	x_3	$f(\mathbf{x})$
:	:	:	:
0	0	1	0 → $(x_1 + x_2 + \overline{x_3})$
:	:	:	:

Äquivalenz von und über Normalformen

Eindeutigkeit

(Folgesätze)

- Die Darstellung einer Booleschen Funktion durch KDNF bzw. KKNF ist (abgesehen von der Reihenfolge) **eindeutig**.
- Zwei allgemeine Darstellungen Boolescher Funktionen sind **äquivalent**, wenn sie (durch Umformungen nach den Regeln der Booleschen Algebra) auf die gleiche KDNF bzw. KKNF zurückgeführt werden können.



Illustration: xkcd.com

Realisierung günstiger Schaltungen

Systematische Realisierung einer Booleschen Funktion f in drei Schritten:

1. Aufstellen der Wahrheitstabelle von f
2. Bilden der KDNF (oder KKNF) von f

$$\text{KDNF: } f(x_1, x_2) = \overline{x_1} \cdot \overline{x_2} + \overline{x_1} \cdot x_2 + x_1 \cdot x_2$$

$$\text{KKNF: } f(x_1, x_2) = \overline{x_1} + x_2$$

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1



3. Schaltungstechnische Realisierung mit Gattern (hier: KKNF)

Einfache Optimierungsregel

Eine KDNF ist günstiger als eine KKNF genau dann, wenn nur für wenige Kombinationen der Eingabewerte $f(x_1, x_2, \dots, x_n) = 1$ gilt.

Bemerkungen zur technischen Realisierung

Alle Booleschen Funktionen lassen sich mit ...

- maximal **zwei Gatterebenen** realisieren, wenn alle Eingangssignale x_i sowohl einfach als auch **negiert** vorliegen,
- sonst mit maximal **drei Gatterebenen**.

Realisierung einer KDNF

- Max. 2^n AND-Gatter mit je n Eingängen (eines pro Minterm)
- Ein OR-Gatter zur Disjunktion aller Minterme (mit max. 2^n Eingängen)

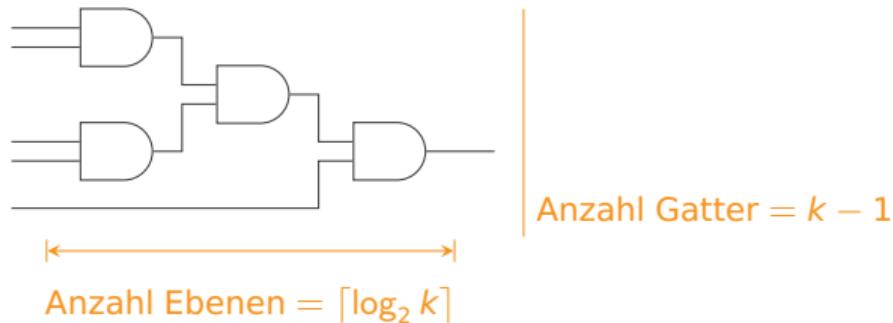
Realisierung einer KKNF

- Max. 2^n OR-Gatter mit je n Eingängen (eines pro Maxterm)
- Ein AND-Gatter zur Konjunktion aller Maxterme (mit max. 2^n Eingängen)

Bemerkungen zur technischen Realisierung (Forts.)

Viele Standardbauteile realisieren Gatter mit **zwei Eingängen**.

Beispiel für ein Gatter mit $k = 5$ Eingängen aus Standardbauteilen:

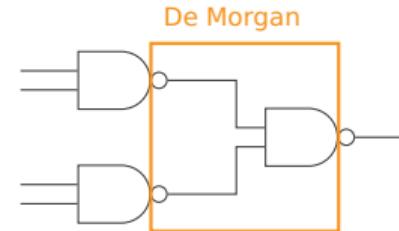
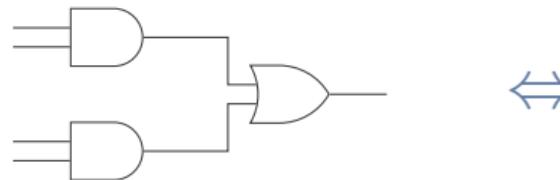


Realisierung einer **kanonischen Normalform**:

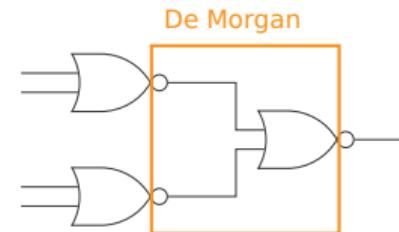
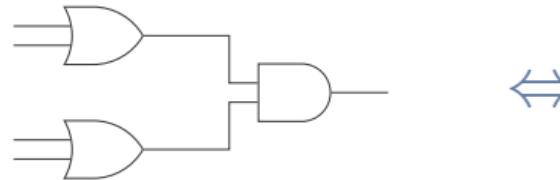
- Max. $2^n(n - 1) + (2^n - 1) = n \cdot 2^n - 1$ Gatter (mit 2 Eingängen)
- Max. $\log_2 n + \log_2 (2^n) = \log_2 n + n$ Ebenen (aus Gattern mit 2 Eingängen)

„Universelle“ Gatter

NAND-Gatter zur Realisierung von [K]**DNF**



NOR-Gatter zur Realisierung von [K]**KNF**



Gliederung heute

- 0. Konfrontation mit der Realität
- 1. Kanonische Darstellungen
- 2. **Minimierung**
- 3. Typische Schaltnetze

Minimierung

„Einfache“ Optimierungskriterien

- Anzahl Gatter → Anzahl **Boolescher Operationen**
- Anzahl Verbindungen
- Anzahl Produkt- bzw. Summenterme

Ansätze

- Händisches Umformen nach Regeln der Booleschen Algebra
- Graphische Verfahren (z. B. **Karnaugh-Veitch-Diagramme**)
- Algorithmen (z. B. **Quine & McCluskey**, auch bei vielen Variablen)

Resolutionsregeln

Für [K]DNF

Wenn sich zwei Summanden nur in **genau einer** komplementären Variable unterscheiden, dann können beide Terme durch ihren gemeinsamen Teil ersetzt werden.

Beispiel

$$x_1 \cdot \overline{x_2} \cdot x_3 \cdot \textcolor{red}{x_4} + x_1 \cdot \overline{x_2} \cdot x_3 \cdot \overline{\textcolor{red}{x_4}} \Leftrightarrow x_1 \cdot \overline{x_2} \cdot x_3$$

Beweis über

- Distributivität $x_1 \cdot \overline{x_2} \cdot x_3 \cdot (\textcolor{red}{x_4} + \overline{x_4})$ sowie (3)
- komplementäre und neutrale Elemente. (7) (6)

Siehe Folie **Axiome** aus der vergangenen Woche.

Resolutionsregeln (Forts.)

Für [K]KNF

Wenn sich zwei Faktoren nur in **genau einer** komplementären Variable unterscheiden, dann können beide Terme durch ihren gemeinsamen Teil ersetzt werden.

Beispiel

$$(x_1 + x_2 + \boxed{x_3} + \overline{x_4}) \cdot (x_1 + x_2 + \boxed{x_3} + \overline{x_4}) \Leftrightarrow (x_1 + x_2 + \overline{x_4})$$

Beweis über

- Kommutativität $(x_1 + x_2 + \overline{x_4} + \overline{x_3}) \cdot (x_1 + x_2 + \overline{x_4} + x_3)$ (1)
- Assoziativität $((x_1 + x_2 + \overline{x_4}) + \overline{x_3}) \cdot ((x_1 + x_2 + \overline{x_4}) + x_3)$ (11)
- Distributivität $(x_1 + x_2 + \overline{x_4}) + (x_3 \cdot \overline{x_3})$ sowie (4)
- komplementäre und neutrale Elemente. (8) (5)

Siehe Folie **Axiome** aus der vergangenen Woche.

Karnaugh–Veitch-Diagramme (KV)

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

A Karnaugh–Veitch diagram for two variables x_1 and x_2 . The horizontal axis is labeled x_1 and the vertical axis is labeled x_2 . The four cells are labeled $\overline{x}_1 \overline{x}_2$, $x_1 \overline{x}_2$, $\overline{x}_1 x_2$, and $x_1 x_2$. Arrows point from the table values to the corresponding cells.

- 2-dimensionale Darstellung der Funktionswerte aus der Wahrheitstabelle
- Jedes Element der Matrix repräsentiert einen Minterm.
- Anordnung der Elemente, sodass sich zwei (zyklisch) **benachbarte Elemente im Vorzeichen genau einer Variable unterscheiden**
- Ermöglicht Zusammenfassung benachbarter Minterme

Herkunft: Maurice Karnaugh's Weiterentwicklung (1953) der Diagramme von Edward Veitch ('52)

Minimierung einer KDNF mit KV-Diagrammen

1. Gegebene KDNF: z. B. $f(x_1, x_2) = \overline{x_1} \cdot \overline{x_2} + \overline{x_1} \cdot x_2 + x_1 \cdot x_2$

2. Erstellen des KV-Diagramms:

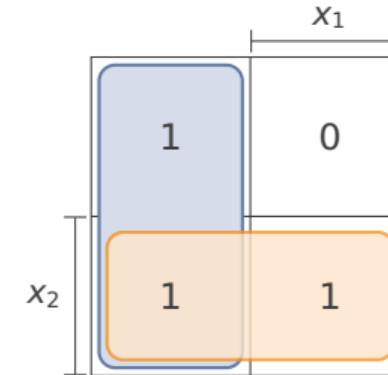
1 für jeden Minterm mit $f(\mathbf{x}) = 1$, sonst 0

3. Markierung möglichst weniger, größer, rechteckiger und ggf. überlappender Bereiche aus 2^k Einsen, sodass alle Einsen überdeckt sind

4. Bildung einer minimalen DNF durch

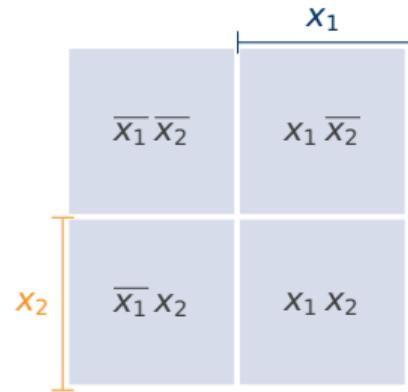
Summierung von genau einem

Produktterm pro markiertem Bereich:

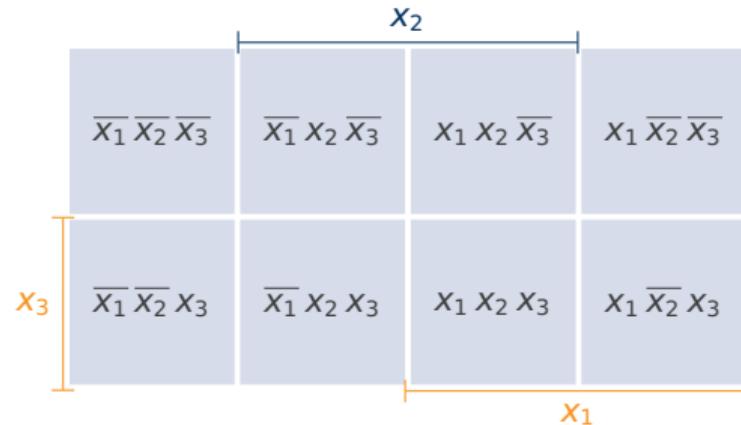


$$f(x_1, x_2) = \overline{x_1} + x_2$$

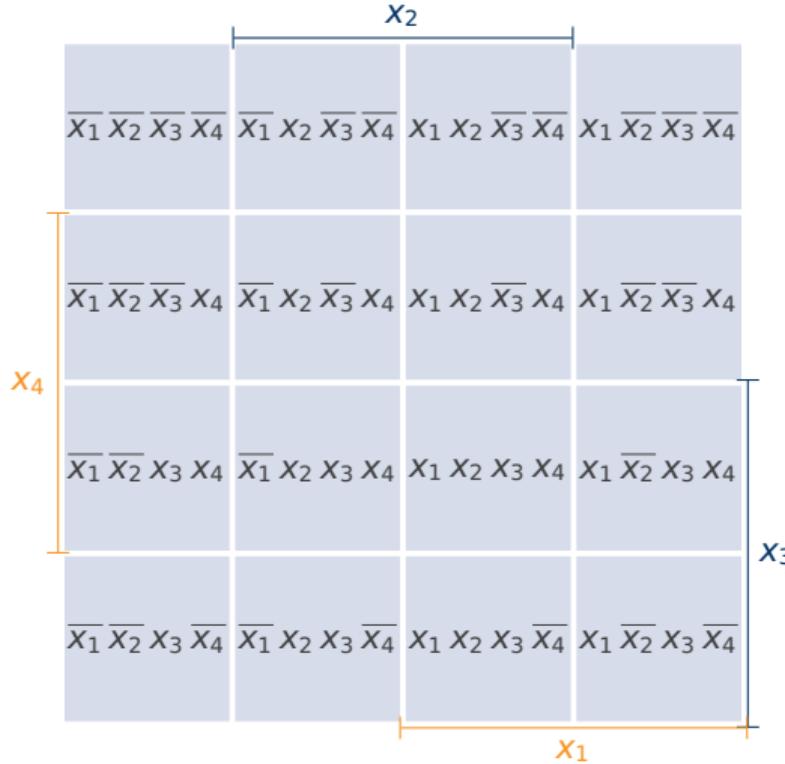
KV-Diagramme für mehrstellige Funktionen



KV-Diagramme für mehrstellige Funktionen

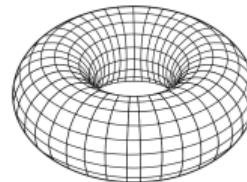
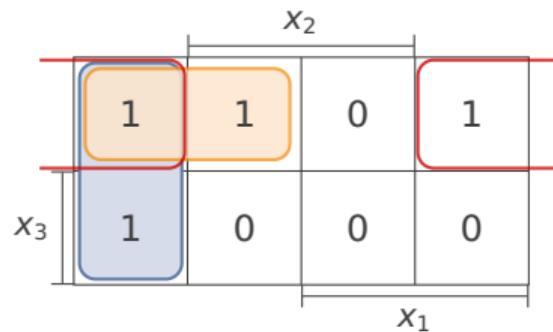


KV-Diagramme für mehrstellige Funktionen



Beispiel mit zyklischer Markierung

Minimiere $y = f(x_1, x_2, x_3)$



x_1	x_2	x_3	y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Minimale DNF: $y = \overline{x_1} \cdot \overline{x_2} + \overline{x_1} \cdot \overline{x_3} + \overline{x_2} \cdot \overline{x_3}$

Minimierung einer KKNF mit KV-Diagrammen

1. Markierung möglichst weniger, großer, rechteckiger und ggf. überlappender Bereiche aus 2^k Nullen, sodass alle Nullen überdeckt sind
2. Bildung einer DNF durch Summierung von genau einem Produktterm pro markiertem Bereich
3. Umwandlung in eine minimale KNF durch abschließende Negation

Grenzen der Karnaugh-Veitch-Diagramme

- Zyklische Markierungen können leicht übersehen werden.
- KV-Diagramme für Boolesche Funktionen mit fünf oder mehr Stellen sind ungebräuchlich.

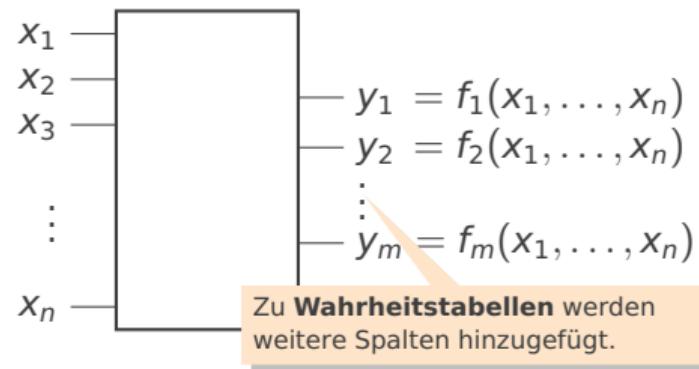
Gliederung heute

- 0.** Konfrontation mit der Realität
- 1.** Kanonische Darstellungen
- 2.** Minimierung
- 3. Typische Schaltnetze**

Synthese von Schaltnetzen

(Wiederholung)

Jede Schaltfunktion $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ mit $m, n \geq 1$ ist in m Boolesche Funktionen mit den gleichen n Eingangsvariablen zerlegbar:

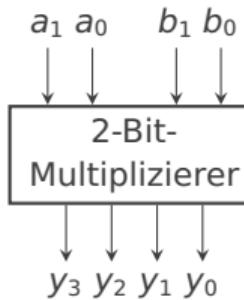


Definition

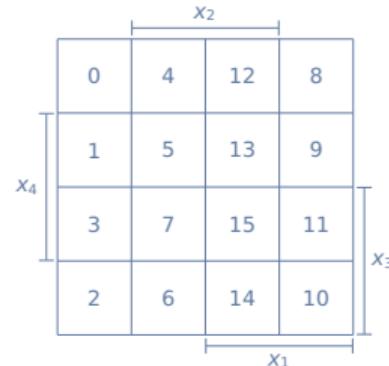
Ein **Schaltnetz** (auch synonym: *kombinatorische Logik*) ist eine schaltungstechnische Realisierung einer Schaltfunktion.

Beispiel mit Minimierung: 2-Bit-Multiplizierer

$a \times b = y$	1. Faktor		2. Faktor		Ergebnis			
	$a_1 = x_1$	$a_0 = x_2$	$b_1 = x_3$	$b_0 = x_4$	y_3	y_2	y_1	y_0
$0 \times 0 = 0$	0	0	0	0	0	0	0	0
$0 \times 1 = 0$	0	0	0	1	0	0	0	0
$0 \times 2 = 0$	0	0	1	0	0	0	0	0
$0 \times 3 = 0$	0	0	1	1	0	0	0	0
$1 \times 0 = 0$	0	1	0	0	0	0	0	0
$1 \times 1 = 1$	0	1	0	1	0	0	0	1
$1 \times 2 = 2$	0	1	1	0	0	0	1	0
$1 \times 3 = 3$	0	1	1	1	0	0	1	1
$2 \times 0 = 0$	1	0	0	0	0	0	0	0
$2 \times 1 = 2$	1	0	0	1	0	0	1	0
$2 \times 2 = 4$	1	0	1	0	0	1	0	0
$2 \times 3 = 6$	1	0	1	1	0	1	1	0
$3 \times 0 = 0$	1	1	0	0	0	0	0	0
$3 \times 1 = 3$	1	1	0	1	0	0	1	1
$3 \times 2 = 6$	1	1	1	0	0	1	1	0
$3 \times 3 = 9$	1	1	1	1	1	0	0	1



Reihenfolge für KV-Diagramm



KV-Diagramme für den 2-Bit-Multiplizierer

	x_2		
x_4	0	0	0
	0	1	1
	0	1	1
	0	0	0

$$\begin{aligned}y_0 &= x_2 \cdot x_4 \\&= a_0 \cdot b_0\end{aligned}$$

	x_2		
x_4	0	0	0
	0	0	1
	0	1	0
	0	1	0

$$\begin{aligned}y_1 &= \overline{x_1}x_2x_3 + x_2x_3\overline{x_4} + x_1\overline{x_3}x_4 + x_1\overline{x_2}x_4 \\&= \overline{a}_1a_0b_1 + a_0b_1\overline{b}_0 + a_1\overline{b}_1b_0 + a_1\overline{a}_0b_0\end{aligned}$$

	x_2		
x_4	0	0	0
	0	0	0
	0	0	0
	0	0	1

$$\begin{aligned}y_2 &= x_1 \cdot \overline{x_2} \cdot x_3 + x_1 \cdot x_3 \cdot \overline{x_4} \\&= a_1 \cdot \overline{a}_0 \cdot b_1 + a_1 \cdot b_1 \cdot \overline{b}_0\end{aligned}$$

	x_2		
x_4	0	0	0
	0	0	0
	0	0	0
	0	0	0

$$\begin{aligned}y_3 &= x_1 \cdot x_2 \cdot x_3 \cdot x_4 \\&= a_1 \cdot a_0 \cdot b_1 \cdot b_0\end{aligned}$$

Realisierung als minimiertes Schaltnetz

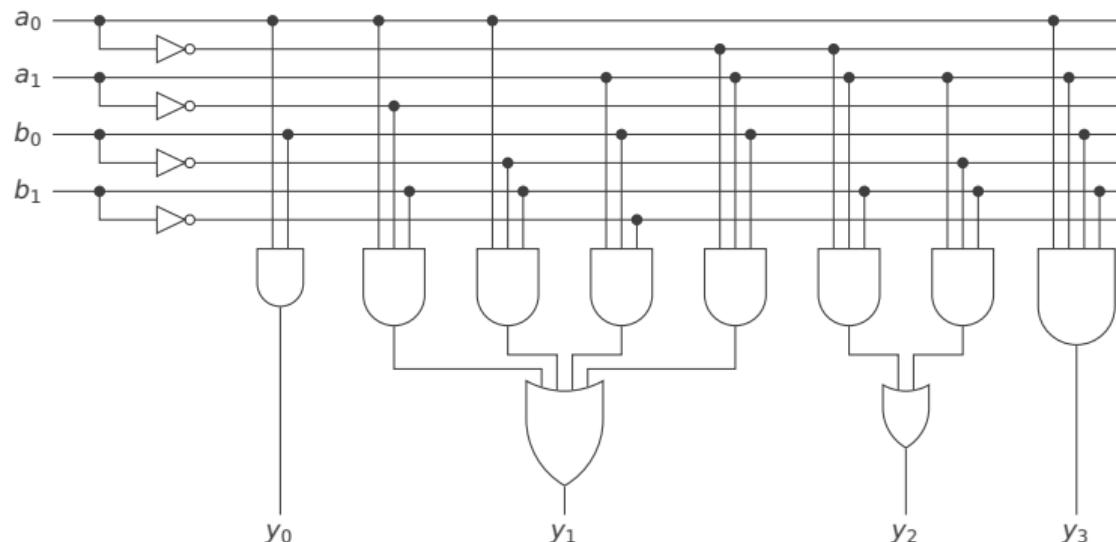
2-Bit-Multiplizierer

$$y_0 = a_0 \cdot b_0$$

$$y_1 = a_0 \cdot \overline{a_1} \cdot b_1 + a_0 \cdot \overline{b_0} \cdot b_1 + a_1 \cdot b_0 \cdot \overline{b_1} + \overline{a_0} \cdot a_1 \cdot b_0$$

$$y_2 = \overline{a_0} \cdot a_1 \cdot b_1 + a_1 \cdot \overline{b_0} \cdot b_1$$

$$y_3 = a_0 \cdot a_1 \cdot b_0 \cdot b_1$$

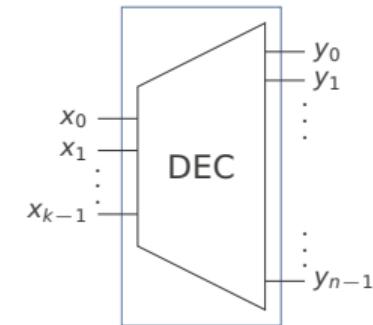


Dekodierer

k -zu- n -Dekodierer: **Auswahl eines** von n Ausgängen $y_i = 1$ durch Binärdarstellung an den Eingängen (x_0, \dots, x_{k-1}) .

Es gilt $0 \leq i < n$ und $1 \leq n \leq 2^k$.

Anwendung: Adressen oder Instruktionen



Beispiel: 2-zu-4-Dekodierer

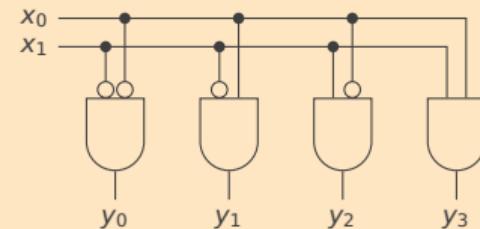
x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

$$y_0 = \overline{x_0} \cdot \overline{x_1}$$

$$y_1 = x_0 \cdot \overline{x_1}$$

$$y_2 = \overline{x_0} \cdot x_1$$

$$y_3 = x_0 \cdot x_1$$

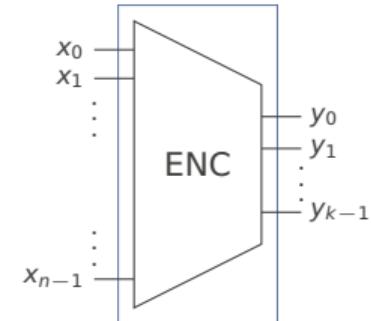


Kodierer

n -zu- k -Kodierer: Ausgabe (y_0, \dots, y_{k-1}) ist
Binärdarstellung für den Index
eines aktiven Eingangs $x_i = 1$.

Es gilt $0 \leq i < n$ und $k \geq \lceil \log_2 n \rceil$.

Anwendung: Kodierung gedrückter Taste

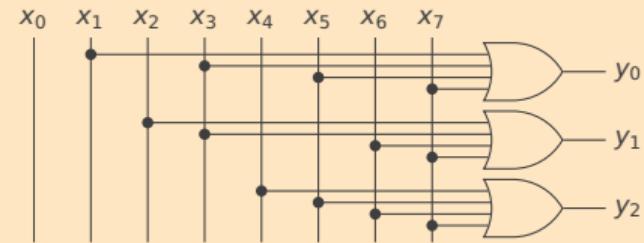


Beispiel: Naiver 8-zu-3-Kodierer

$$y_0 = x_1 + x_3 + x_5 + x_7$$

$$y_1 = x_2 + x_3 + x_6 + x_7$$

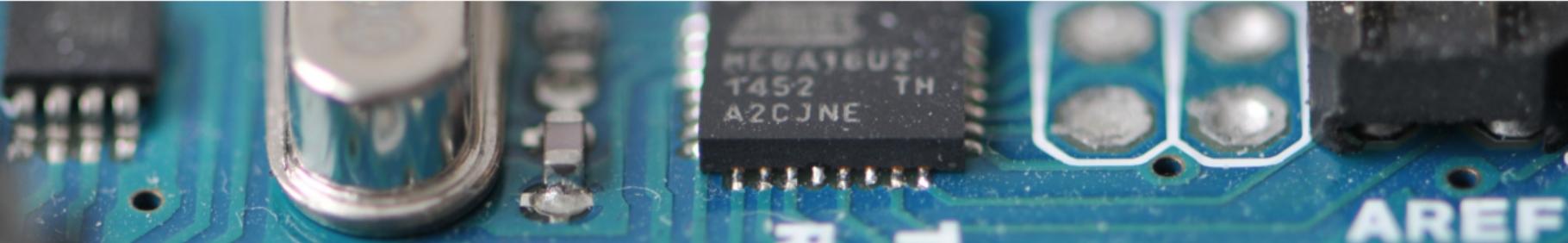
$$y_2 = x_4 + x_5 + x_6 + x_7$$



Problem: Undefinierte Ausgabe, falls mehrere Eingänge aktiv sind.

Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Sequenzielle Logik I

Univ.-Prof. Dr.-Ing. Rainer Böhme

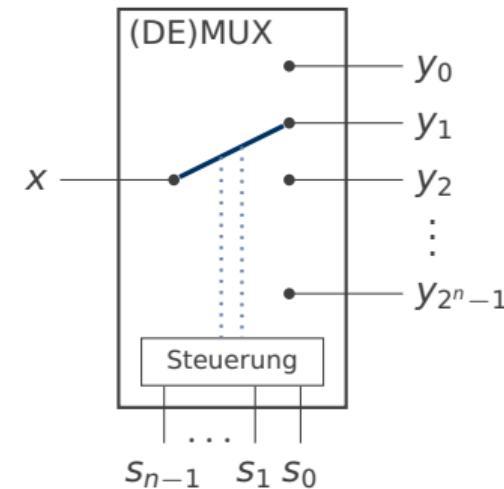
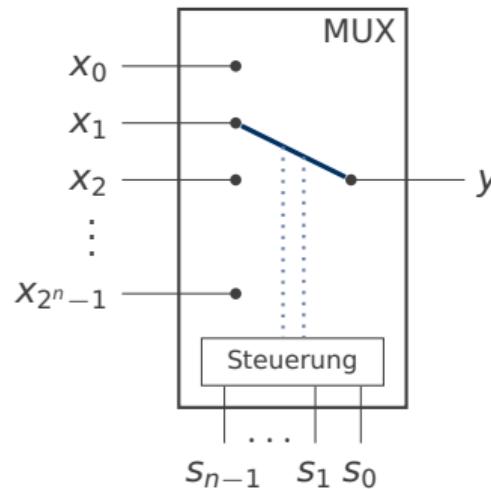
Wintersemester 2021/22 · 27. Oktober 2021

Gliederung heute

- 1. Abschluss der Kombinatorischen Logik**
2. Flipflops
3. Schaltwerke

Multiplexer

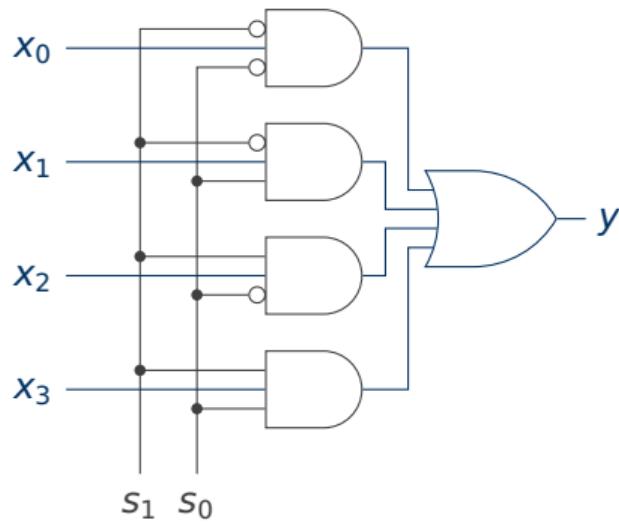
Auswahl einer Datenquelle x_i (bzw. Senke y_j für Demultiplexer) über n Steuerleitungen:



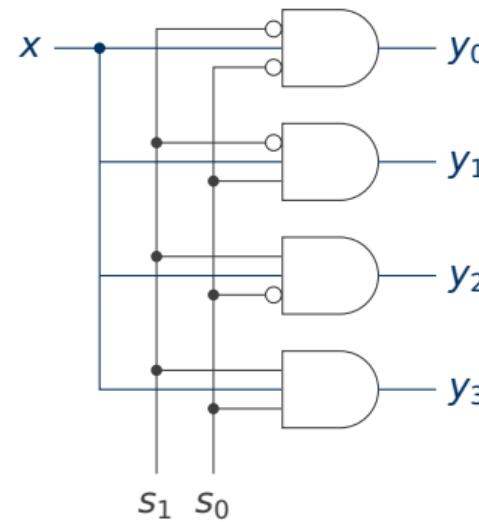
Schaltungstechnische Realisierung

für $n = 2$ Steuerleitungen

MUX



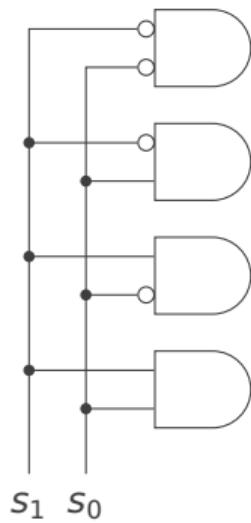
DEMUX



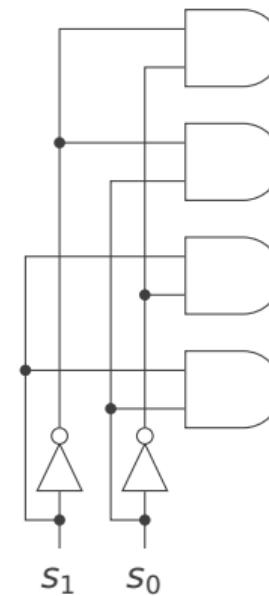
Schaltungstechnische Realisierung

für $n = 2$ Steuerleitungen

Steuerlogik



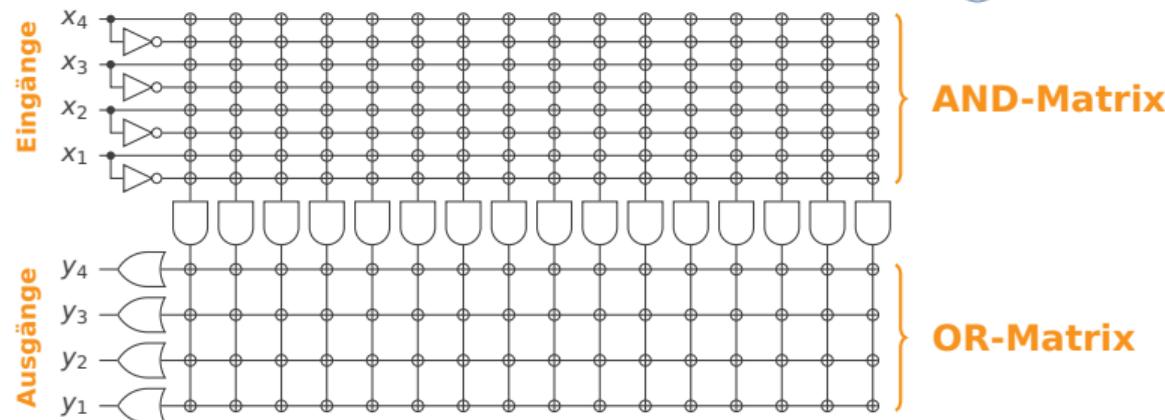
Alternative mit Vorinvertierung



OTP-Logikanordnungen

(engl. one time programmable)

Grundstruktur



Programmierung der Bauelemente

Hohe Schreibströme brennen Sicherungen (engl. fuses) auf dem Die durch oder neutralisieren Dioden.

Varianten

Programmable Read-Only Memory (PROM)

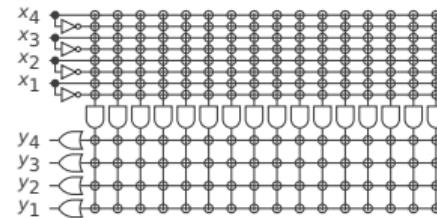
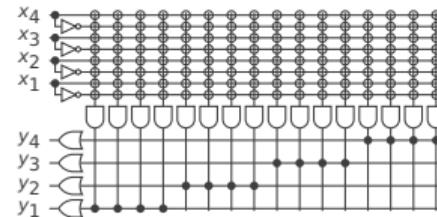
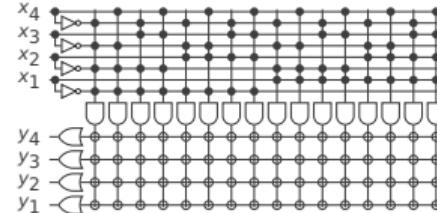
- AND-Matrix fest, OR-Matrix programmierbar
- Direkte Realisierung von Wahrheitstabellen
- PROM mit 2^n m -Bit-Worten implementiert jede Schaltfunktion $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$.

Programmable Array Logic (PAL)

- AND-Matrix programmierbar, OR-Matrix fest
- Realisiert DNFs bis zu einer Obergrenze von Produkttermen pro Summand

Programmable Logic Array (PLA)

- AND-Matrix **und** OR-Matrix programmierbar
- Realisiert DNFs bis zu einer Obergrenze an Produkttermen (gg. durch Anzahl der AND-Gatter)



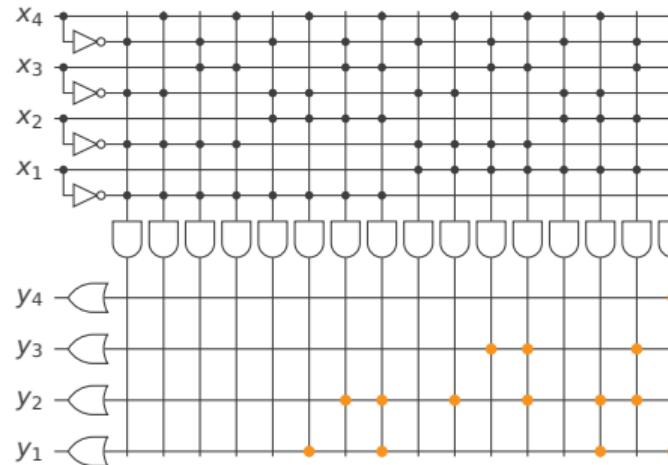
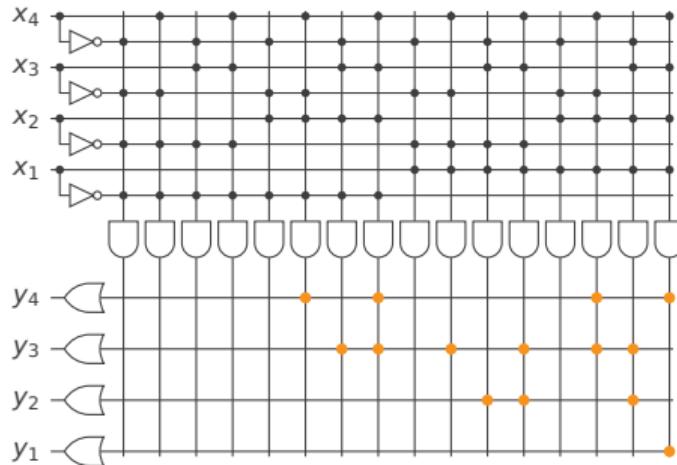
Hörsaalfrage



1. Welches PROM realisiert diesen 2-Bit-Multiplizierer?

24 82 94 16

$$8y_1 + 4y_2 + 2y_3 + y_4 = (2x_1 + x_2) \times (2x_3 + x_4)$$



Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Abgrenzung

Kombinatorische Logik

- Keine Rückkopplung
- Grundelemente sind **Gatter**
- Schaltnetze sind idealisiert **verzögerungsfrei**
- Beschreibung durch **Wahrheitstabellen** oder **Boolesche Ausdrücke**

Sequenzielle Logik

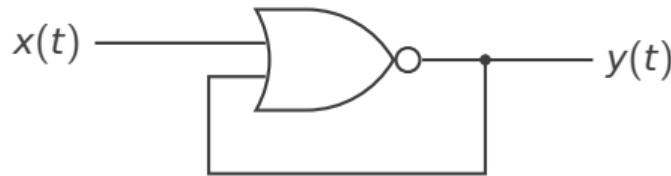
- **Kontrollierte** Rückkopplung
- Grundelemente sind **Flipflops** und Gatter
- Schaltwerke berücksichtigen **Zeitverhalten** unter Annahme konstanter **Gatterlaufzeit** τ
- Beschreibung durch **Zustandstabelle** und **Zustandsdiagramm** oder **Ansteuer-** und **Ausgabegleichungen**

Gliederung heute

1. Abschluss der Kombinatorischen Logik
2. **Flipflops**
3. Schaltwerke

Rückkopplung eines Gatterausgangs

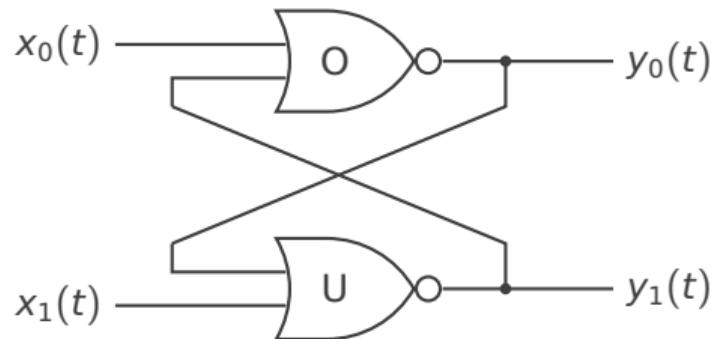
Beispiel: NOR-Gatter



$x(t)$	$y(t + \tau)$	$y(t + 2\tau)$	$y(t + 3\tau)$
0	$\overline{y(t)}$	$y(t)$	$\overline{y(t)}$
1	0	0	0

→ Ausgang oszilliert weitgehend unkontrolliert.

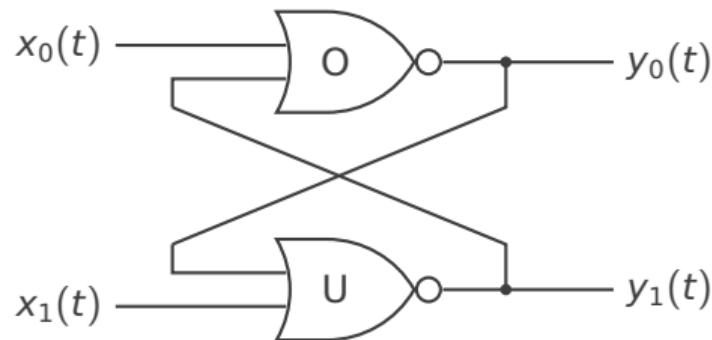
Kreuzweise Rückkopplung zweier Gatterausgänge



$x_0(t)$	$x_1(t)$	$y_0(t + \tau)$	$y_1(t + \tau)$	$y_0(t + 2\tau)$	$y_1(t + 2\tau)$
0	0	$\overline{y_1(t)}$	$\overline{y_0(t)}$?	?
0	1	$\overline{y_1(t)}$	0	1	0
1	0	0	$\overline{y_0(t)}$	0	1
1	1	0	0	0	0

Kreuzweise Rückkopplung zweier Gatterausgänge

Beispiel: NOR-Gatter



Bistabile Kippstufe

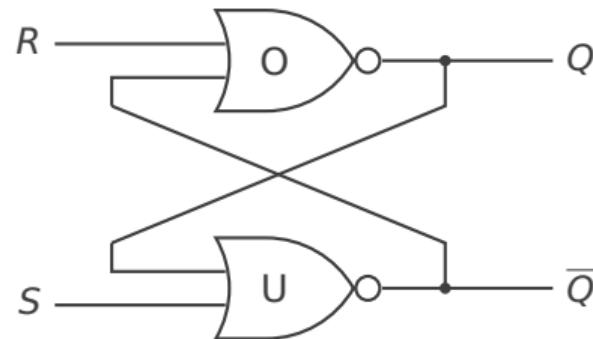
Diese Schaltung kann ein Bit in ihrem **Zustand** Q speichern.

RS-Flipflop

Bezeichnung der zwei Eingänge

- R = "reset" (löschen)
- S = "set" (setzen)

Realisierung mit zwei NOR-Gattern



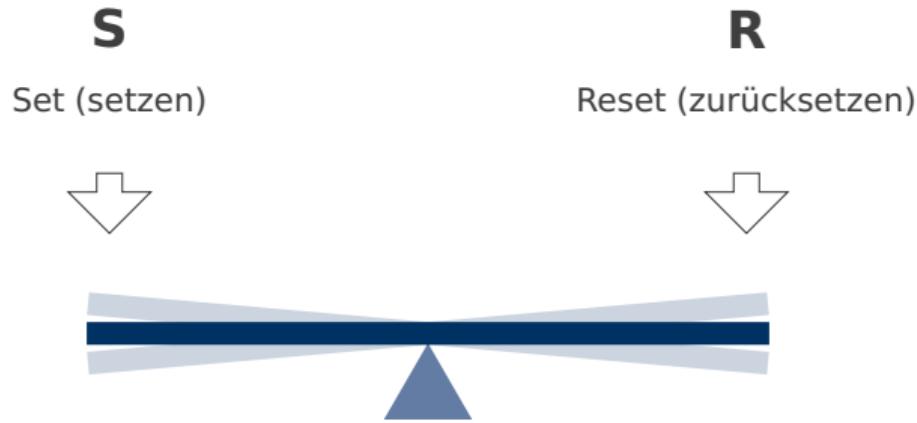
Folgezustand

charakteristische
Tabelle

R	S	Q'
0	0	Q
0	1	1
1	0	0
1	1	nicht erlaubt

Flipflop: bistabile Kippstufe (W)

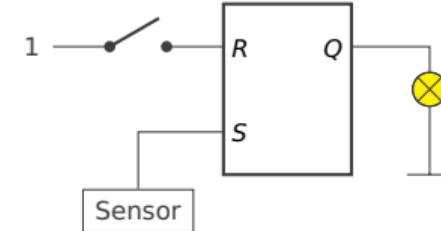
Mechanische Analogie: Wippe



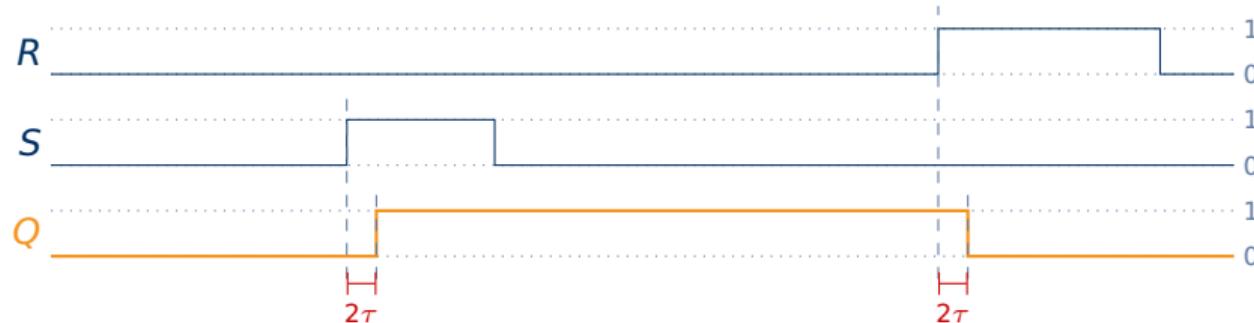
Beispiel für den Einsatz von RS-Flipflops

Einschalten der Warnlampe bei kurzfristiger Temperaturüberschreitung,
Taster zum Zurücksetzen

- Speicherung eines flüchtigen Werts
- Initialisierung erforderlich



Darstellung im Zeitverlauf



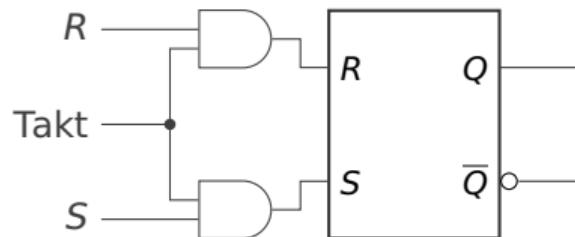
Getaktetes RS-Flipflop

Idee

Vermeidung unbeabsichtigter Zuständsübergänge durch Laufzeiteffekte:

Übernahme der Signale an R und S **nur wenn** Taktsignal 1 ist.

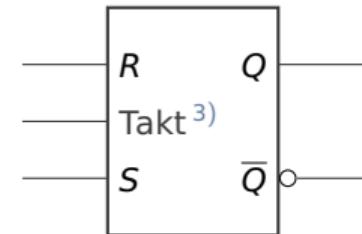
Realisierung



charakteristische Tabelle

R	S	Takt	Q'
d ¹⁾	d ¹⁾	0	Q
0	0	1	Q
0	1	1	1
1	0	1	0
1	1	1	_ ²⁾

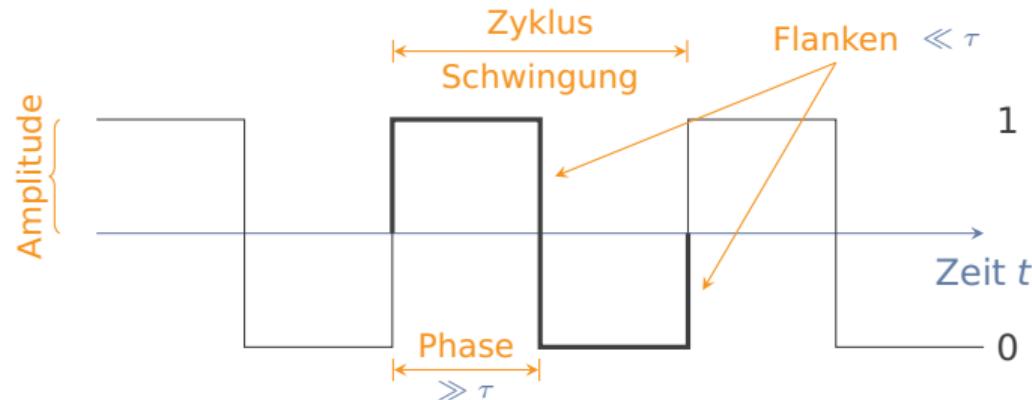
Symbol



¹⁾ don't care, ²⁾ i. d. R. nicht erlaubt, ³⁾ auch: Clk oder c für clock

Takt

Symmetrisches Rechtecksignal mit konstanter **Taktfrequenz** (Schwingungen pro Zeiteinheit) gemessen in **Hertz** [1 Hz = $1/s$]



Taktgenerierung

Industriell gefertigte **Quarze** schwingen sehr präzise (> 4 MHz).
Niedrigere Frequenzen erreicht man durch (mehrfache) Teilung.

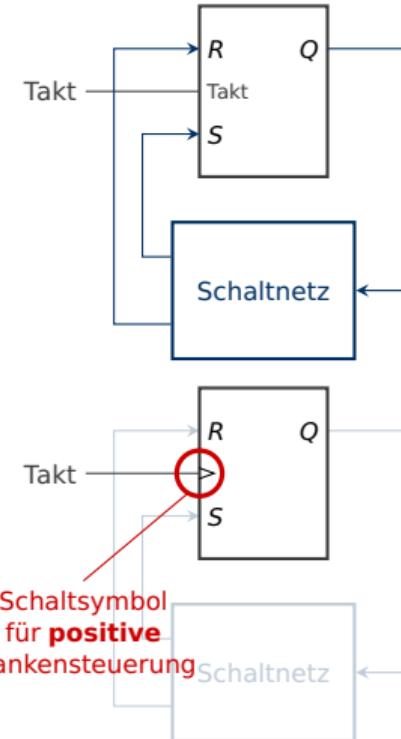
Flankengesteuertes RS-Flipflop

Nachteil getakteter RS-Flipflops

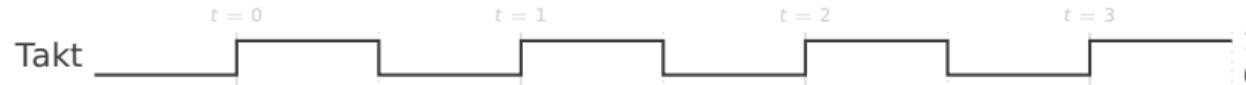
- mehrere Zustandsänderungen in einer Taktphase möglich
- erschwert kontrollierte Rückkopplung über Schaltnetz

Alternative: Flankensteuerung

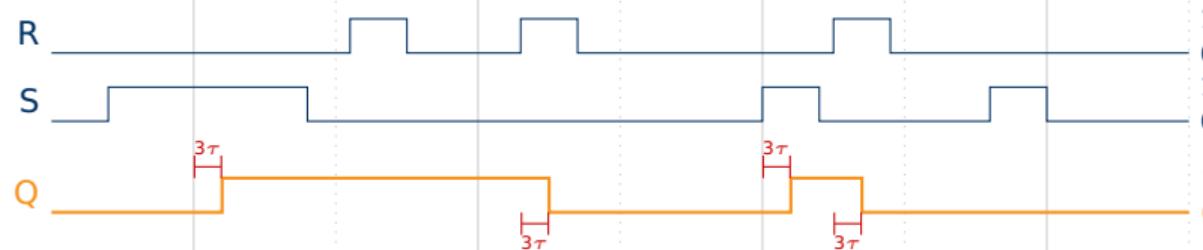
- Übernahme der Eingabewerte zu einem bestimmten Zeitpunkt
- Varianten für positive (steigende) oder negative (fallende) Flanke



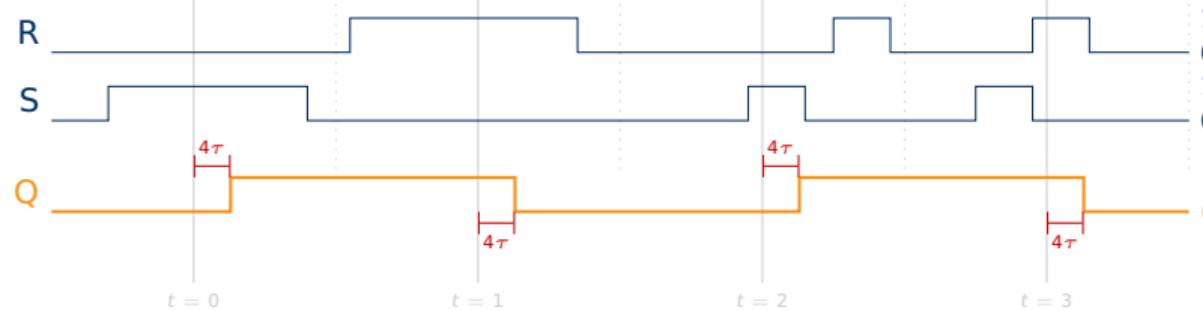
Zeitverhalten ausgewählter Flipflops



getaktetes RS-Flipflop



flankengesteuertes RS-Flipflop

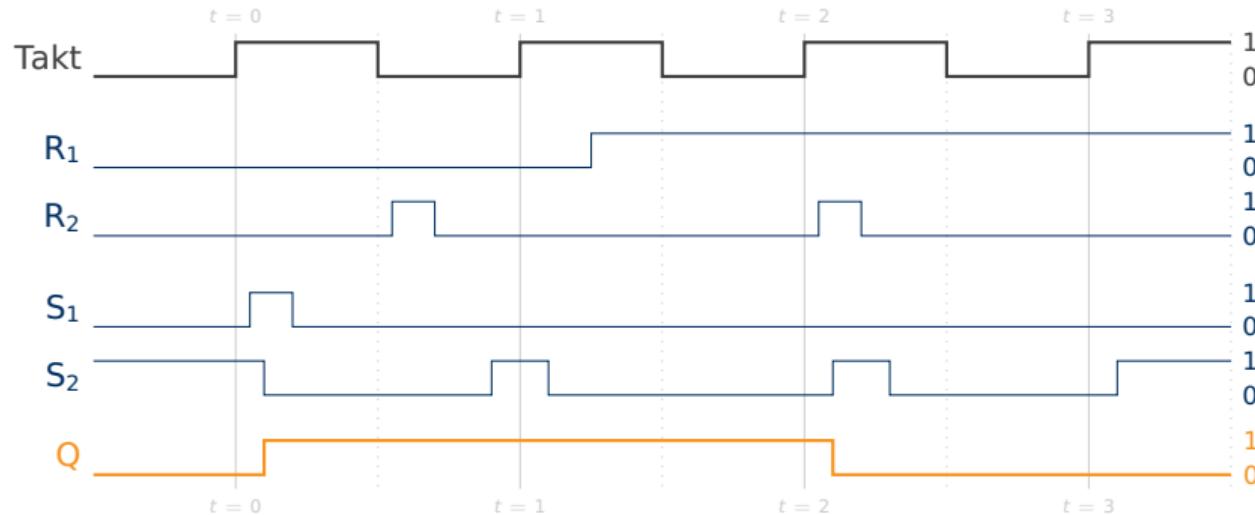


Hörsaalfrage



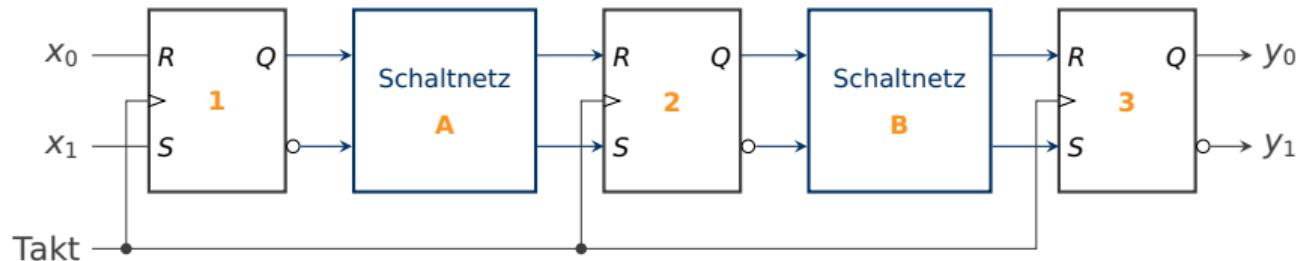
2. Welche R- und S-Signale passen zur (idealen) Ausgabe Q eines flankengesteuerten RS-Flipflops?

24 82 94 16



Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Realisierung einer „Pipeline“



- parallele Speicherung/Verarbeitung unterschiedlicher Daten
- schrittweise **Weitergabe** der Ergebnisse bei Taktflanke
- Ergebnis liegt (im Beispiel) nach drei Takten am Ausgang an.
- **Voraussetzung:** Die maximale Verzögerung aller Schaltnetze
– **der kritische Pfad –** ist kürzer als ein Taktzyklus.

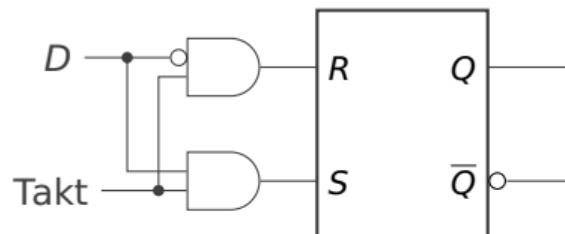
D-Flipflop

(Abk. für Daten oder Delay, engl. *Verzögerung*)

Eigenschaften

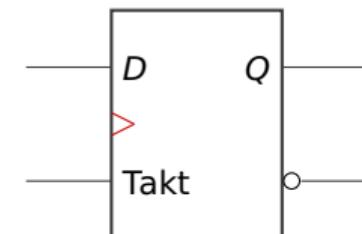
- Ein Datenbit $D = S = \bar{R}$ als Eingang (vermeidet $R = S = 1$).
- Bei Takt = 1 wird der Folgezustand $Q' = D$ gesetzt.

Realisierung



D	Takt	Q'
0	0	Q
0	1	0
1	0	Q
1	1	1

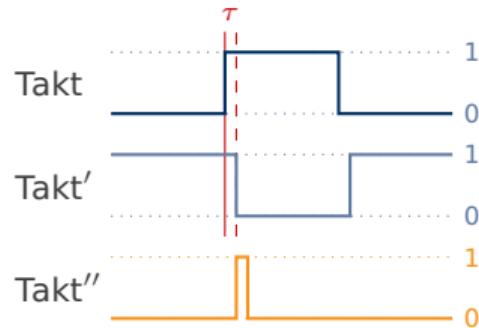
Symbol



→ D-Flipflops kommen häufig mit Flankensteuerung zum Einsatz.

Flankenerkennung

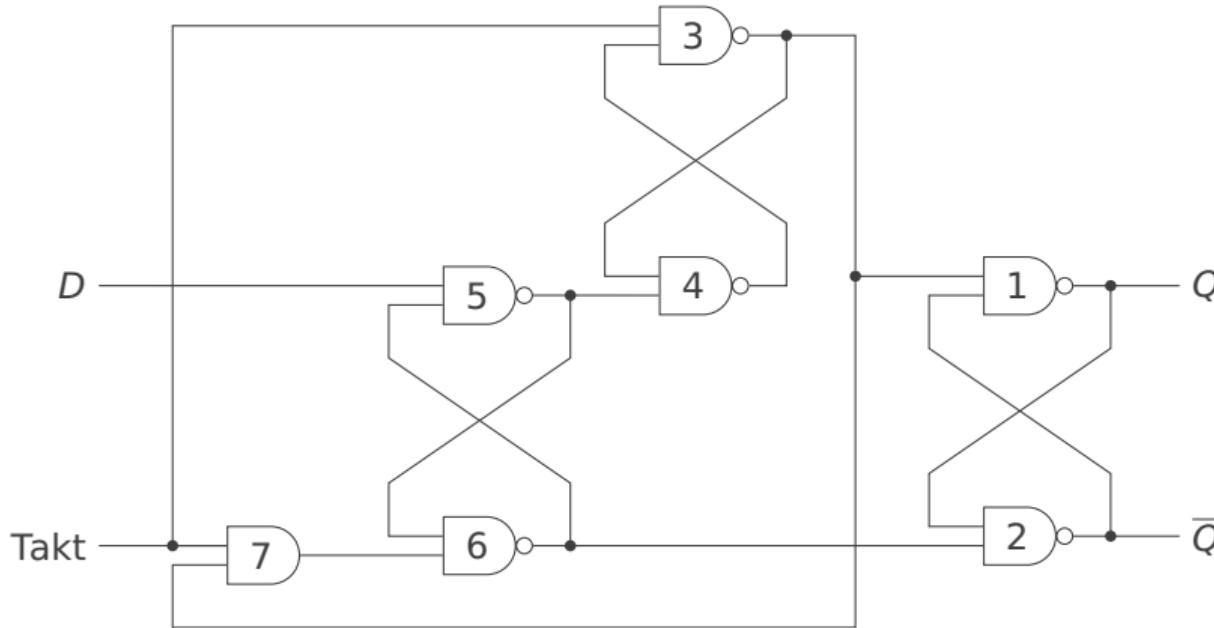
schaltungstechnische Realisierung für **positive** Flanke



→ Ausnutzung von Laufzeiteffekten

Realisierung mit NAND- und AND-Gattern

positiv flankengesteuertes D-Flipflop

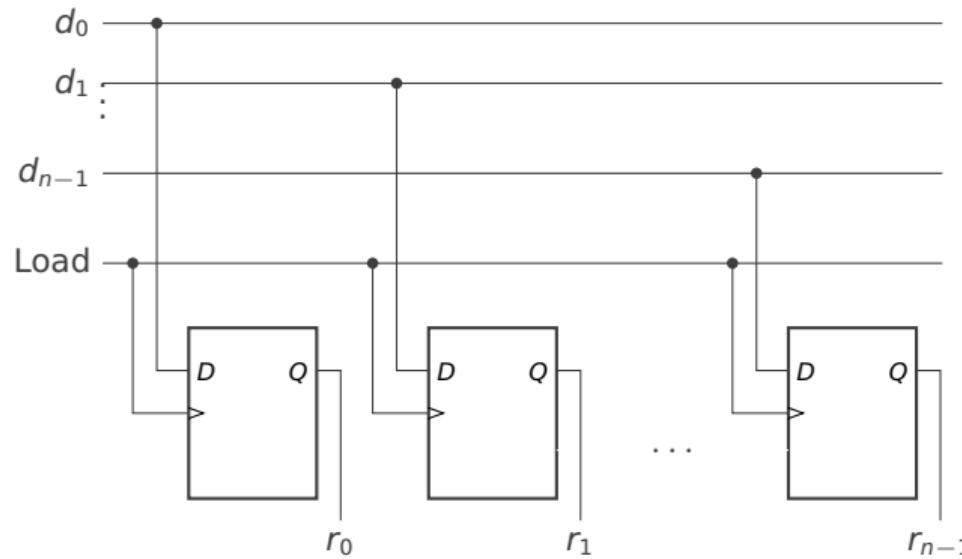


(Funktionsweise nicht prüfungsrelevant)

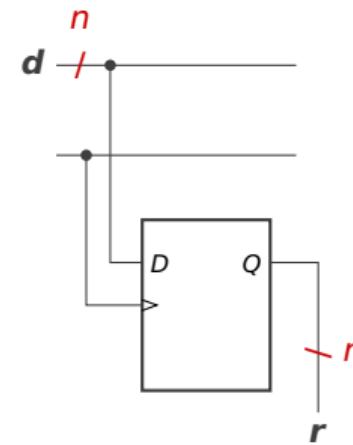
Gliederung heute

1. Abschluss der Kombinatorischen Logik
2. Flipflops
3. **Schaltwerke**

Beispiel: n -Bit-Register



vereinfachte
Darstellung:



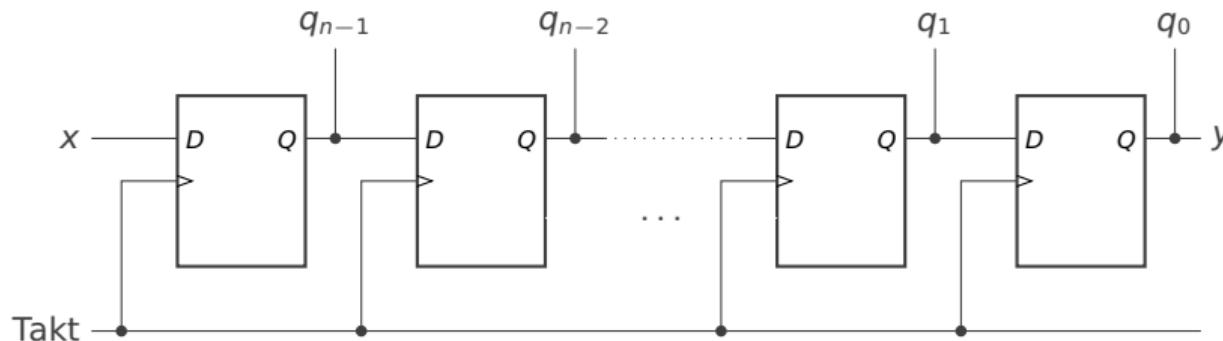
Konstruktion aus flankengesteuerten D-Flipflops

Übernahme der Daten von einem **Datenbus d** der Breite n Bit bei steigender Flanke des „Load“-Signals.

Beispiel: n -Bit-Schieberegister

Anwendungen

- seriell-zu-parallel-Wandlung
- arithmetische Operationen

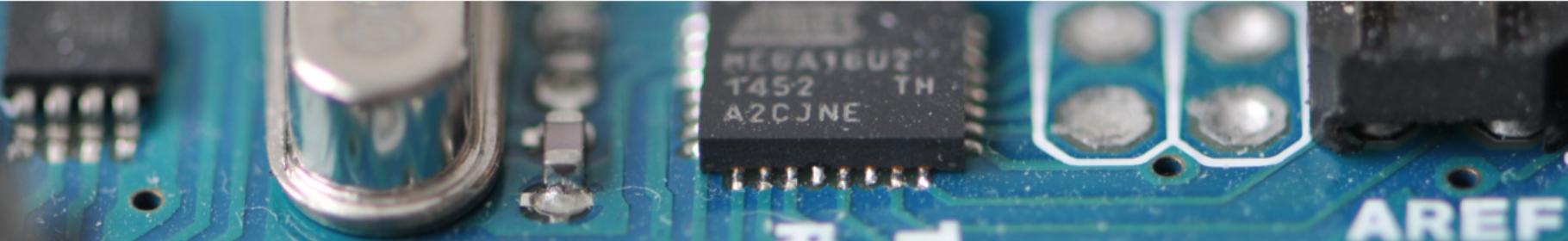


Konstruktion aus flankengesteuerten D-Flipflops

Verschiebung der Binärwerte pro Takt um eine Position nach rechts.

Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Sequenzielle Logik II

Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2021/22 · 3. November 2021

Gliederung heute

- 1. Schaltwerke** (Forts.)
2. Systematischer Entwurf synchroner Schaltwerke
3. Optional: Realisierung mit Transistoren

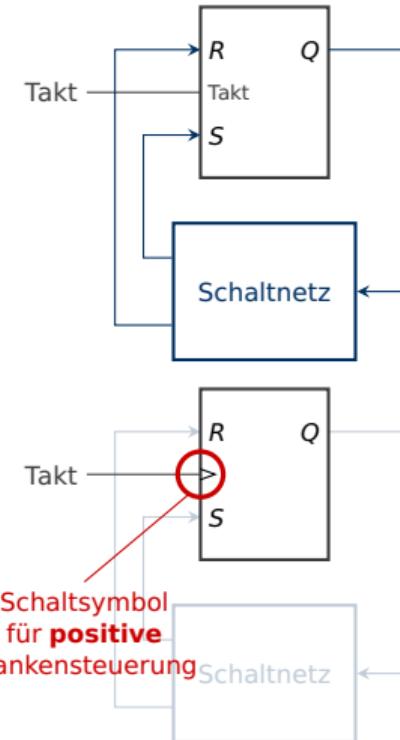
Flankengesteuertes RS-Flipflop (W)

Nachteil getakteter RS-Flipflops

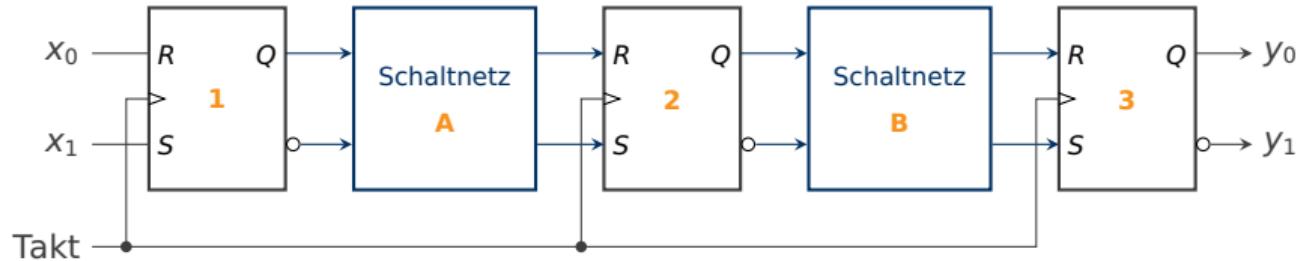
- mehrere Zustandsänderungen in einer Taktphase möglich
- erschwert kontrollierte Rückkopplung über Schaltnetz

Alternative: Flankensteuerung

- Übernahme der Eingabewerte zu einem bestimmten Zeitpunkt
- Varianten für positive (steigende) oder negative (fallende) Flanke



Realisierung einer „Pipeline“ (W)



- parallele Speicherung/Verarbeitung unterschiedlicher Daten
- schrittweise Weitergabe der Ergebnisse bei Taktflanke
- Ergebnis liegt (im Beispiel) nach drei Takten am Ausgang an.
- **Voraussetzung:** Die maximale Verzögerung aller Schaltnetze – der kritische Pfad – ist kürzer als ein Taktzyklus.

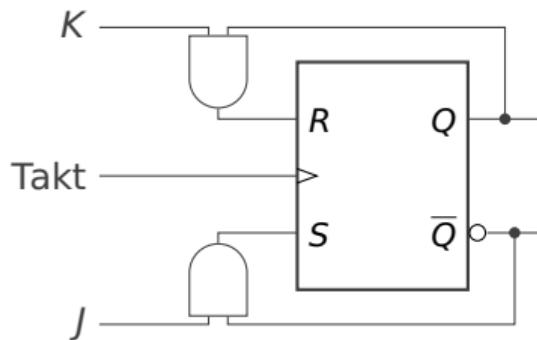
JK-Flipflop

(Abk. für Jump/Kill oder Initialen von Jack Kilby)

Idee

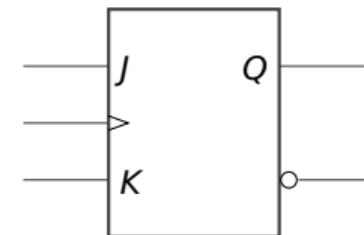
Nutzung der nicht benötigten Eingangskombination (1, 1) zur Invertierung von Q (engl. *toggle*)

Realisierung



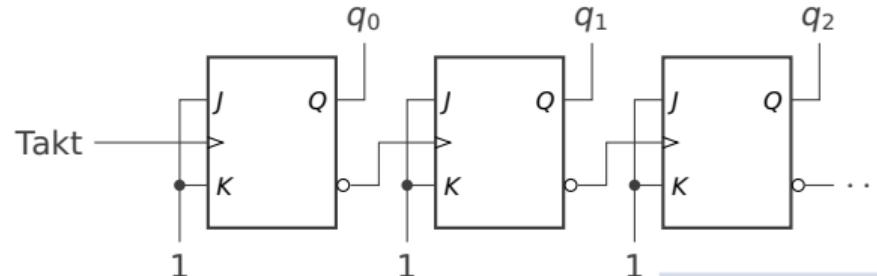
Symbol

J	K	Q'
0	0	Q
0	1	0
1	0	1
1	1	\bar{Q}

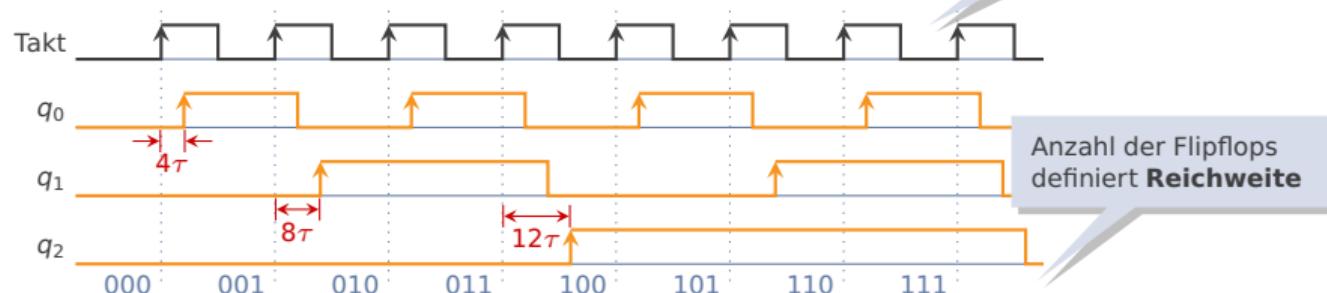


Asynchroner 3-Bit-Zähler

Zählt positive Taktflanken in Binärdarstellung (q_2, q_1, q_0)



Zeitverhalten



Die **maximale Taktfrequenz**
hängt von Verzögerung τ ab

Anzahl der Flipflops
definiert **Reichweite**

Wichtige Unterscheidung

Sequenzielle Logik



Asynchrone Schaltwerke

- Steuerung durch Änderung der Eingangssignale
- Zeitpunkt stabiler Ausgangssignale abhängig von Gatterlaufzeit
- Entwurf aufwändig
- Schaltungen sehr schnell

Synchrone Schaltwerke

- Steuerung durch zentralen Takt
- Ein- und Ausgangssignale zu festen Zeitpunkten
- systematischer Entwurf
- kritischer Pfad bestimmt maximale Taktfrequenz

Gliederung heute

1. Schaltwerke (Forts.)
2. **Systematischer Entwurf synchroner Schaltwerke**
3. Optional: Realisierung mit Transistoren

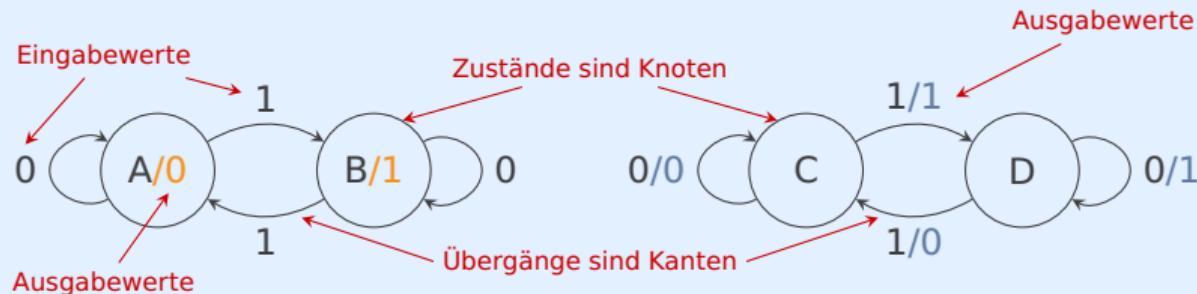
Zustandsautomat

Modell

- Zu jedem diskreten Zeitpunkt t befindet sich der Automat in genau einem Zustand S_t aus der endlichen Zustandsmenge \mathbb{S} .
- Zustandsübergänge sind eine Funktion von S und Eingabe x .
- Die **Ausgabe** y hängt ab von:
 - entweder nur dem Zustand
 - oder von Zustand **und** Eingabe

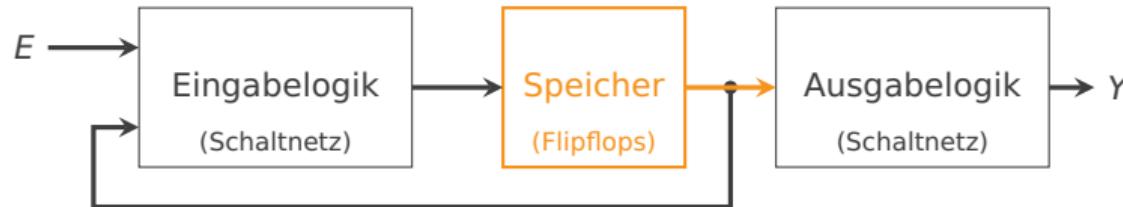
→ Moore-Automat
→ Mealy-Automat

Darstellung als gerichtete zyklische Graphen

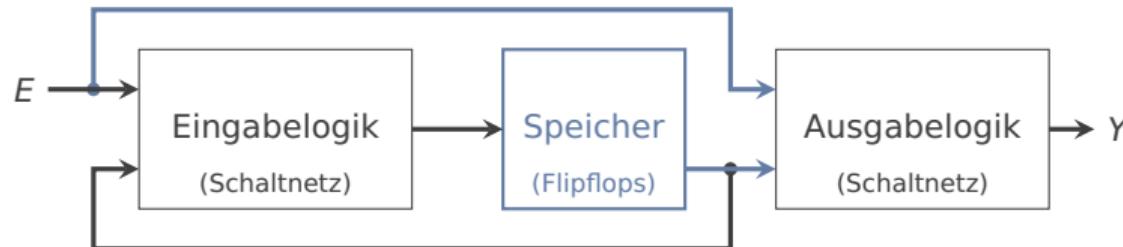


Blockschaltbilder

Moore-Automat



Mealy-Automat



Vorgehensweise beim Entwurf

1. Erstellung eines **Zustandsdiagramms**
2. Erstellung einer **Zustandstabelle**
3. Wahl einer **binären Zustandskodierung** und
Ableitung der **binären Zustandstabelle**
4. Auswahl der Flipflop-Typen und Ermittlung
der **Ansteuerlogik** für jeden Zustandsübergang
5. Ermittlung der **Ausgabegleichungen**
6. **Minimierung** der Ansteuer- und Ausgabegleichungen
7. **Realisierung** des Schaltwerks

Beispiel: Entwurf eines synchronen Schaltwerks

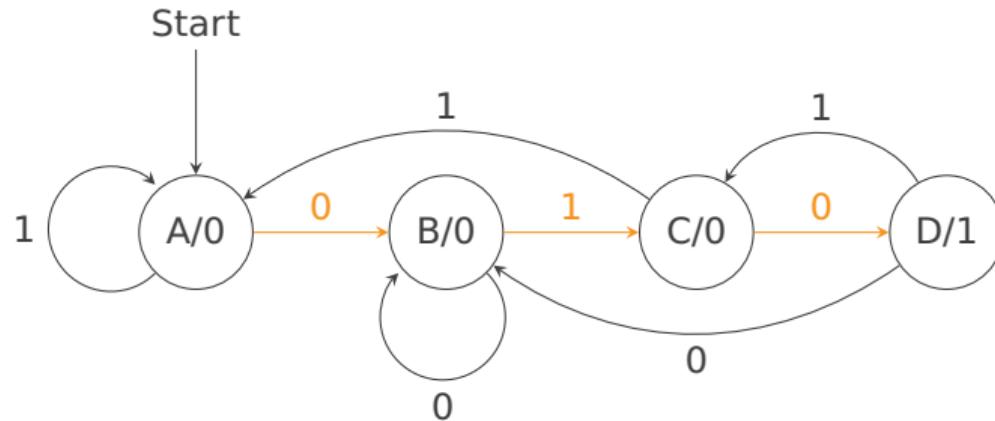
Aufgabenstellung

„Entwerfen Sie ein synchrones Schaltwerk zur Erkennung der Sequenz 010 im binären Eingabestrom $x(t)$. Die Ausgabe y nehme den Wert 1 an, sobald im Eingabestrom die Sequenz 010 erkannt wurde. Sonst sei $y = 0$.“

Vorgehensweise

- ausführliches Vorgehen für einen Moore-Automaten
- Betrachtung der Unterschiede im Falle eines Mealy-Automaten

Schritt 1: Zustandsdiagramm



Schritt 2: Zustandstabelle

Die Zustandstabelle enthält für jeden (symbolischen) Zustand $S \in \mathbb{S}$

- **Folgezustand S'** in Abhängigkeit von der Eingabe x (im Bsp.: x)
- **Ausgabe** (im Bsp.: y)

S	x	S'	y
A	0	B	0
A	1	A	0
B	0	B	0
B	1	C	0
C	0	D	0
C	1	A	0
D	0	B	1
D	1	C	1

Schritt 3: binär kodierte Zustandstabelle

Umkodierung von \mathbb{S} in **binäre Zustände** $Q \subseteq \{0, 1\}^k$
mit $k = \lceil \log_2 |\mathbb{S}| \rceil$.

S	x	S'	y		$q_1 q_0$	x	$q'_1 q'_0$	y
A	0	B	0	→	0 0	0	0 1	0
A	1	A	0		0 0	1	0 0	0
B	0	B	0		0 1	0	0 1	0
B	1	C	0		0 1	1	1 0	0
C	0	D	0		1 0	0	1 1	0
C	1	A	0		1 0	1	0 0	0
D	0	B	1		1 1	0	0 1	1
D	1	C	1		1 1	1	1 0	1

Schritt 4: Flipflop-Typen und Ansteuererlogik

Darstellung aller Zustandsübergänge $Q_i \rightarrow Q'_i$ über Flipflops

Ansteuerungstabelle für zwei flankengesteuerte **JK-Flipflops**

q_1	q_0	x	q'_1	q'_0	J_1	K_1	J_0	K_0
0	0	0	0	1	0	d	1	d
0	0	1	0	0	0	d	0	d
0	1	0	0	1	0	d	d	0
0	1	1	1	0	1	d	d	1
1	0	0	1	1	d	0	1	d
1	0	1	0	0	d	1	0	d
1	1	0	0	1	d	1	d	0
1	1	1	1	0	d	0	d	1



Der Schaltungsaufwand hängt von der Wahl der Flipflop-Typen ab.

Schritt 5: Ausgabegleichungen

Beim Moore-Automat kann die binäre Zustandstabelle auf alle eindeutigen Zeilen von Q (ohne Q') reduziert werden.

q_1	q_0	y
0	0	0
0	1	0
1	0	0
1	1	1

Daraus lassen sich Boolesche Funktionen als Ausgabegleichungen bestimmen:

$$y = q_0 \cdot q_1$$

(in diesem Beispiel bereits minimal)

Schritt 6: Minimierung der Ansteuergleichungen

- Flipflop des niederwertigen Bits der Zustandskodierung q_0

$$J_0 = \bar{x} \quad K_0 = x$$

(direkt aus der Ansteuerungstabelle, siehe Schritt 4)

- Flipflop des höherwertigen Bits der Zustandskodierung q_1

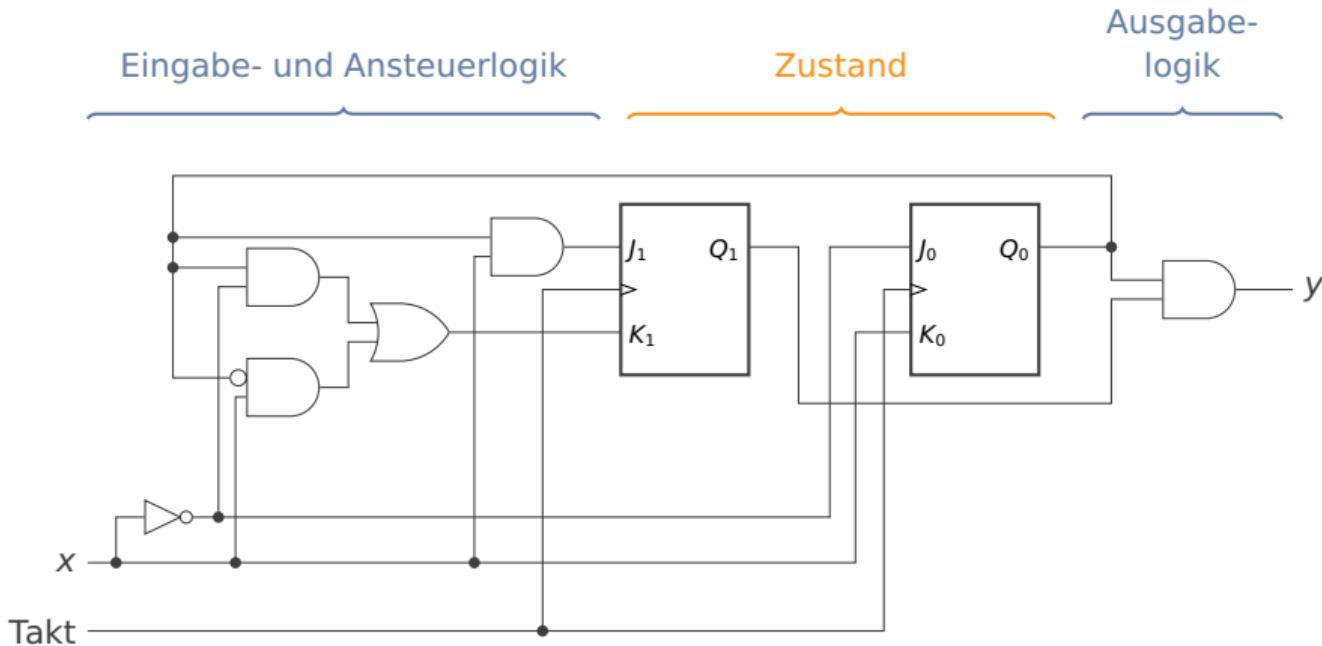
$$J_1 = q_0 \cdot x$$

$$K_1 = q_0 \cdot \bar{x} + \bar{q}_0 \cdot x$$

		q_1			
		0	d	d	0
x	0	d	d	1	
				q_0	

		q_1			
		0	1	d	
x	d	0	1	d	
				q_0	

Schritt 7: Realisierung des Schaltwerks



Beispiel: Entwurf eines synchronen Schaltwerks

Aufgabenstellung

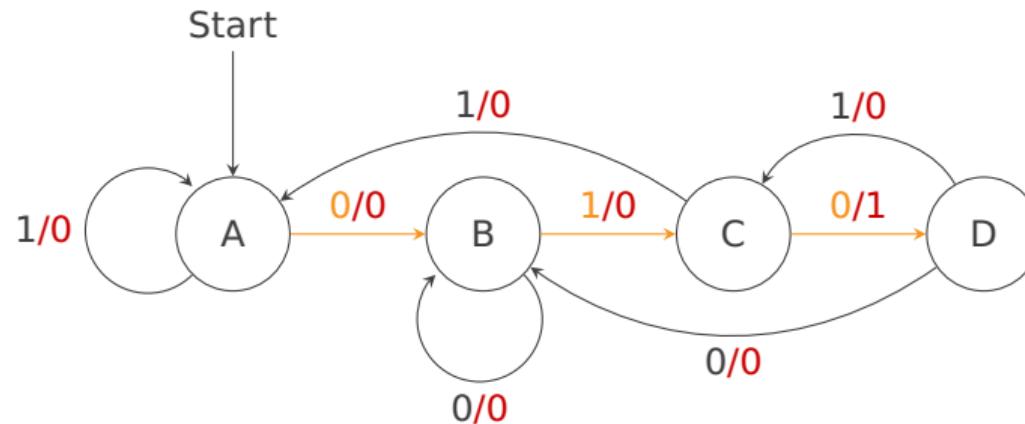
„Entwerfen Sie ein synchrones Schaltwerk zur Erkennung der Sequenz 010 im binären Eingabestrom $x(t)$. Die Ausgabe y nehme den Wert 1 an, sobald im Eingabestrom die Sequenz 010 erkannt wurde. Sonst sei $y = 0$.“

Vorgehensweise

- ausführliches Vorgehen für einen Moore-Automaten
- **Betrachtung der Unterschiede im Falle eines Mealy-Automaten**

Schritt 1: Zustandsdiagramm

bei Modellierung als **Mealy-Automat**



→ Markierung der Kanten (statt Knoten) mit Ausgabe y

Schritte 2 & 3: Zustandstabellen

Die prinzipielle Vorgehensweise ist analog zum Moore-Automat.

Beim **Mealy**-Automat ändert sich jedoch die Ausgabe y
im gleichen Takt, indem sich die Eingabe x ändert.

S	x	S'	y
A	0	B	0
A	1	A	0
B	0	B	0
B	1	C	0
C	0	D	1
C	1	A	0
D	0	B	0
D	1	C	0



q_1	q_0	x	q'_1	q'_0	y
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	0	1	0
1	1	1	1	0	0

Schritte 4 bis 6: Logik, Minimierung

- Ermittlung der **Ansteuerlogik**

(hier im Beispiel unverändert zum Moore-Automat)

- Ermittlung der **Ausgabegleichungen**

Abhängig von Zustand Q und Eingabe x

$$y = \overline{q_0} \cdot q_1 \cdot \bar{x}$$

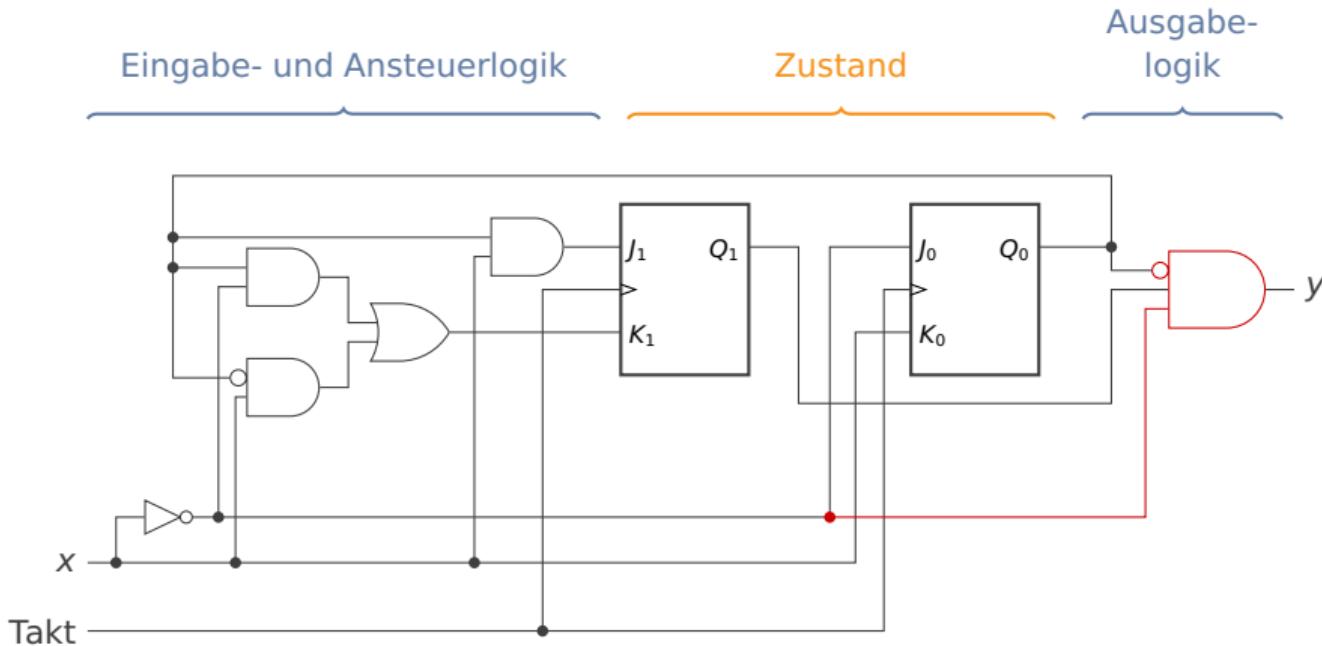
q_1	q_0	x	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

- Minimierung**

(hier im Beispiel unverändert bzw. bereits minimal)

Schritt 7: Realisierung des Schaltwerks

(jetzt als **Mealy**-Automat)



Hörsaalfragen



Welche Eigenschaften treffen eher auf Moore- oder Mealy-Automaten zu ?

24 82 94 16

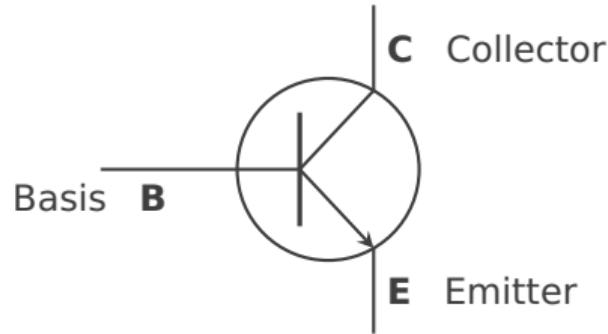
1. Schnelle Reaktion auf Veränderungen der Eingabesignale
2. Geringere Anzahl von Flipflops, wenn Zustände durch Übergänge mit verschiedenen Ausgaben erreicht werden können
3. Zum Entwurf beliebiger Schaltwerke geeignet
4. Geringer Schaltungsaufwand der Ausgabelogik
5. Asynchrone Störungen der Eingabesignale wirken sich niemals auf die Ausgabe aus.

Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Gliederung heute

1. Schaltwerke (Forts.)
2. Systematischer Entwurf synchroner Schaltwerke
3. **Optional: Realisierung mit Transistoren**

Transistor

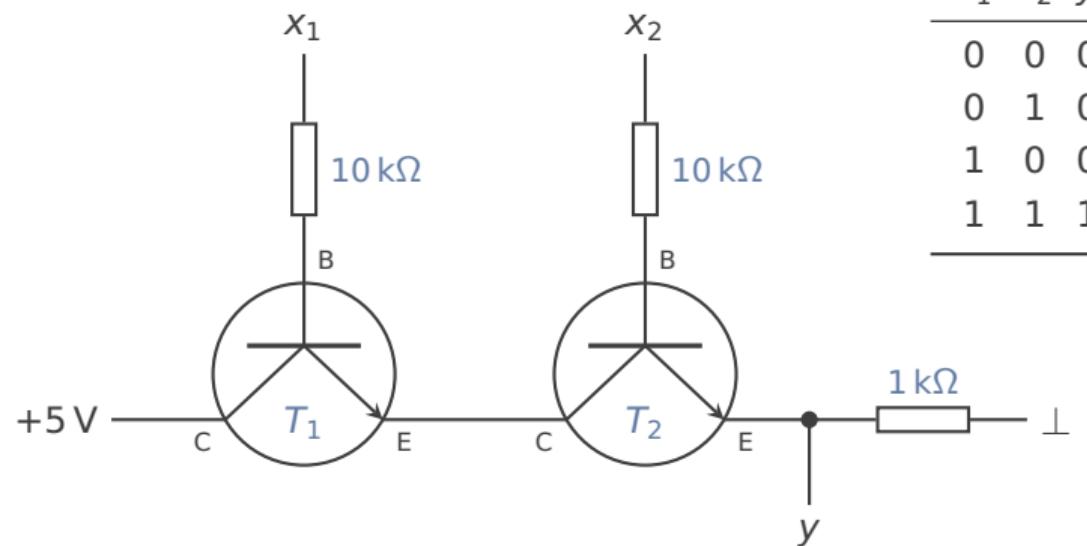


Transistor (engl. transfer resistor)

Strom fließt von C nach E genau dann wenn ein vernachlässigbar kleiner Steuerstrom von B nach E fließt.

Gatter-Schaltung

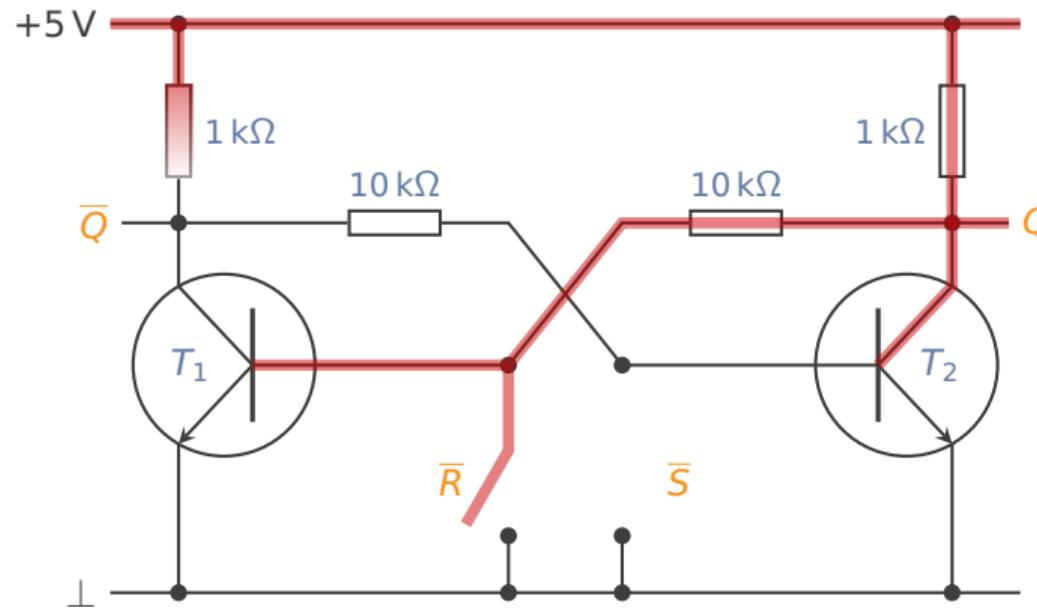
Realisierung eines AND-Gatters



Flipflop-Schaltung

(engl. auch "latch")

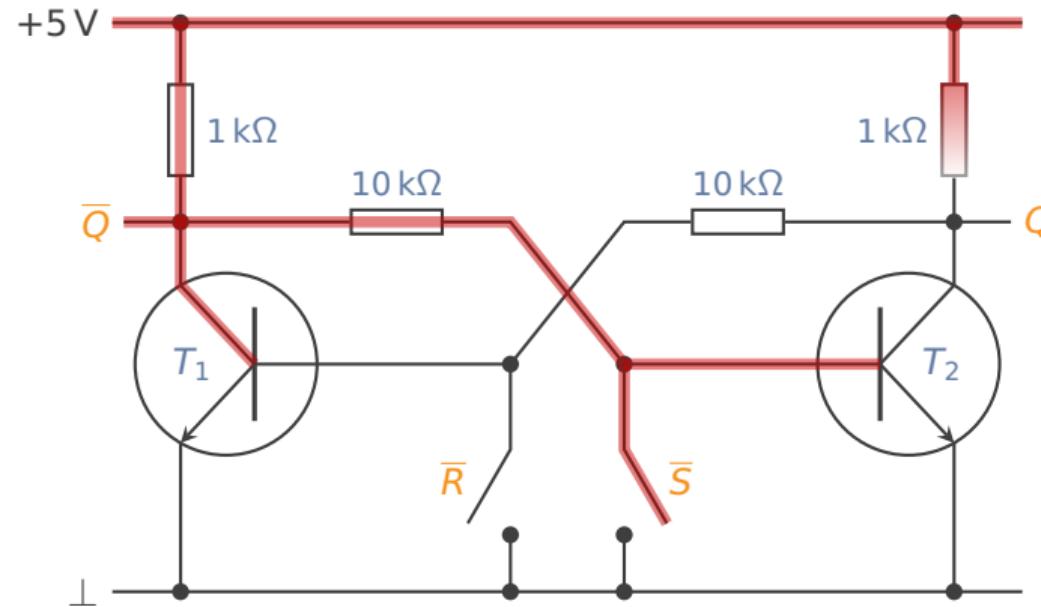
Realisierung mit zwei Bipolartransistoren



Flipflop-Schaltung

(engl. auch "latch")

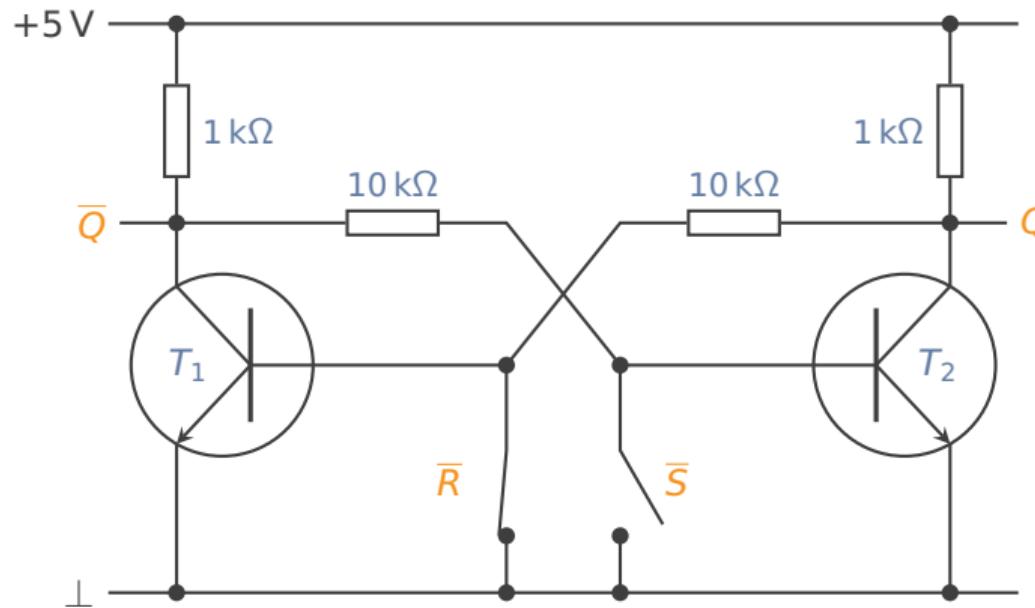
Realisierung mit zwei Bipolartransistoren



Flipflop-Schaltung

(engl. auch "latch")

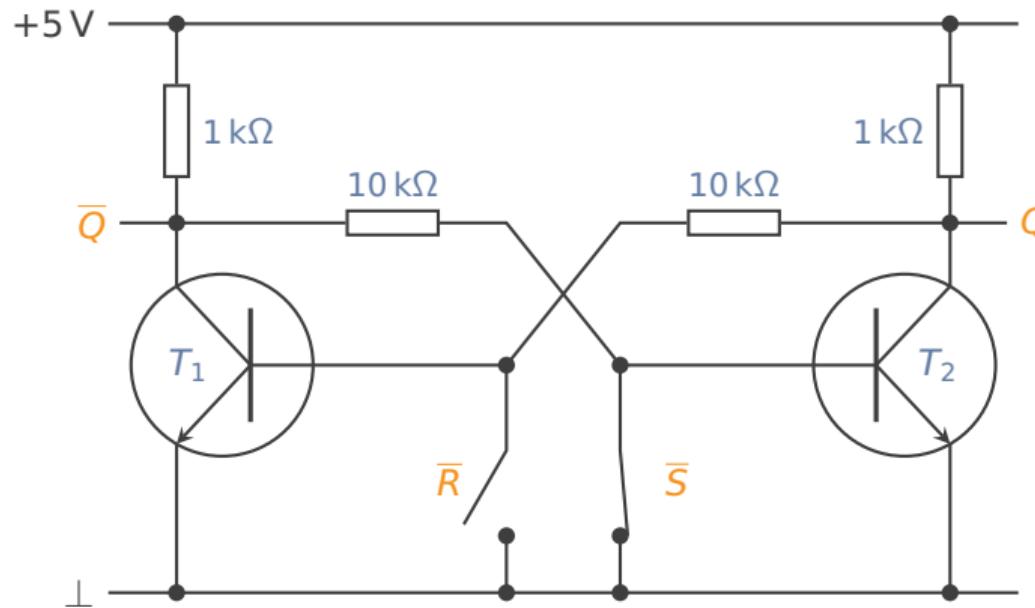
Realisierung mit zwei Bipolartransistoren



Flipflop-Schaltung

(engl. auch "latch")

Realisierung mit zwei Bipolartransistoren



Flipflops als integrierte Schaltungen

Realisierung in frühen integrierten Schaltkreisen

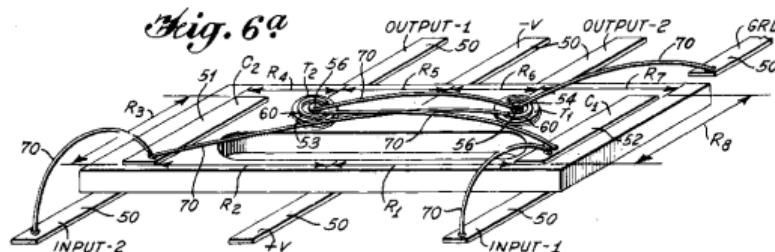
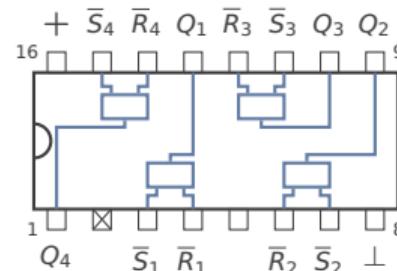


Fig. 4. Germanium flip-flop using mesa transistors, bulk resistors, diffused capacitors, and air isolation of the components. From US Patent 3,138,743.

Standardbauteil 4044 mit Vierfach-RS-Flipflop (engl. Quad RS Latches)



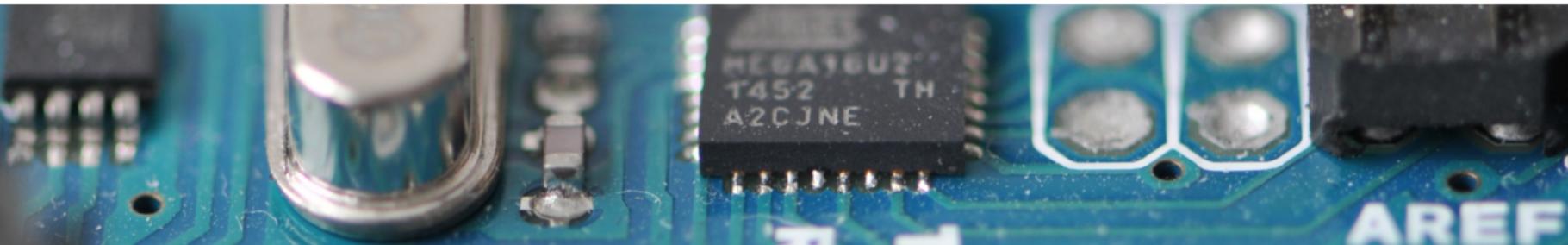
Bildquellen: J. Kilby, IEEE, West Florida Components

Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)

später
→

inday students



Rechnerarchitektur

Arithmetik I

Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2021/22 · 10. November 2021

Gliederung heute

- 1. Addition und Subtraktion**
2. Arithmetisch-logische Einheit

Addieren von Binärzahlen

Addition positiver n -stelliger Binärzahlen a und b stellenweise von rechts nach links:

- An jeder Stelle i kann ein **Übertrag** (engl. *carry*) $c_i = 1$ auftreten.
- Falls Summe $y = a + b \geq 2^n$, reichen n Bit nicht mehr für die Darstellung des Ergebnisses aus. Das $(n + 1)$ -te Summenbit wird als **Überlauf** (engl. *overflow*) bezeichnet.

Beispiele

(für $n = 8$)

Übertrag („1 gemerkt“) →

8-Bit-Summe

$$\begin{array}{r} 0001\ 0111 \\ + 0101\ 0110 \\ \hline \end{array}$$

$$\begin{array}{r} (23)_{10} \\ (86)_{10} \\ \hline \end{array}$$

1 11

$$0110\ 1101$$

$$(109)_{10}$$

$$\begin{array}{r} 0011\ 0111 \\ + 1101\ 0110 \\ \hline \end{array}$$

$$\begin{array}{r} (55)_{10} \\ (214)_{10} \\ \hline \end{array}$$

1 111 11

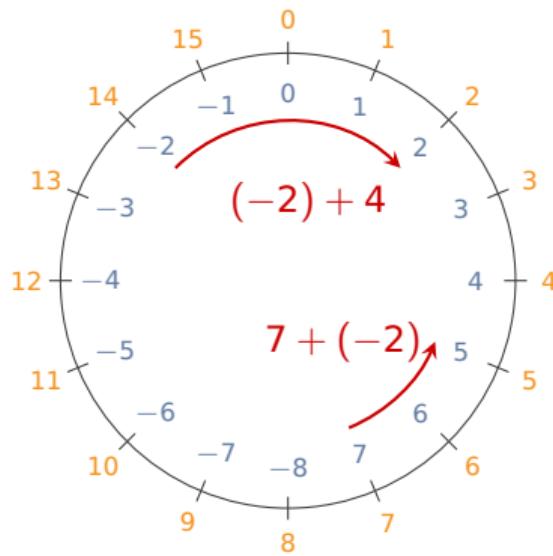
$$1\ 0000\ 1101$$

$$(269)_{10}$$

$$0000\ 1101$$

$$(13)_{10}$$

Zahlendarstellung



Beispiel $n = 4$ Bit für

- natürliche Zahlen $0 \leq k < 2^n$ (**eindeutig** in n -Bit-Register) und
- vorzeichenbehaftete** ganze Zahlen $-2^{(n-1)} \leq z < +2^{(n-1)}$ in der **Zweierkomplement**-Darstellung

Vergleich am Zahlenstrahl

Binärdarstellung vorzeichenbehafteter Ganzzahlen mit n Stellen

Wertebereiche

(Beispiele für $n = 8$ Bit)

positive Binärzahlen

$$0, 1, \dots, 2^{n-1} - 1, \quad 2^{n-1}, \dots, 2^n - 1$$

0 1 127 128 255

Zweierkomplement

$$-2^{n-1}, \dots, -1, 0, 1, \dots, 2^{n-1} - 1$$

-128 -1 0 1 127

Arithmetische Negation in Zweierkomplementdarstellung

1. Invertiere alle Bits. $(127)_{10} = (0111\ 1111)_2 \Rightarrow (1000\ 0000)_2 = (-128)_{10}$
2. Addiere 1. $(-128)_{10} + 1 = (1000\ 0000)_2 + 1 = (1000\ 0001)_2 = (-127)_{10}$

Fallunterscheidung

1. Zahlen a und b **positiv** (d. h. „Vorzeichenbits“ $a_{n-1} = b_{n-1} = 0$)

Arithmetischer Überlauf bei $y_{n-1} = 1$ ($\Leftrightarrow c_{n-1} = 0$ und $c_{n-2} = 1$)

2. Zahlen a und b **negativ** (d. h. „Vorzeichenbits“ $a_{n-1} = b_{n-1} = 1$)

$\Rightarrow a' = -a$ und $b' = -b$ positiv, also gilt:

$$y' = a + b = (2^n - a') + (2^n - b') = 2 \cdot 2^n - (a' + b')$$

Das korrekte Ergebnis $y = 2^n - (a' + b') = y' - 2^n$ wird durch Abschneiden und Ignorieren des Übertragsbits c_{n-1} erreicht.

Arithmetischer Überlauf bei $y_{n-1} = 0$ ($\Leftrightarrow c_{n-1} = 1$ und $c_{n-2} = 0$)

3. Vorzeichen von a und b **unterschiedlich** (z. B. sei b negativ)

$$y' = a + b = a + (2^n - b') = 2^n - (b' - a) \text{ ist korrekt für } |b| > |a|.$$

Das korrekte Ergebnis für $|b| \leq |a|$ ist $y = a - b' = y' - 2^n$.

Auch hier: Abschneiden und Ignorieren des Übertragsbits c_{n-1} .

Kein arithmetischer Überlauf möglich!

Rechenbeispiele

($n = 8$ Bit)

	0001 0111	$(23)_{10}$
	+ 1111 1111	$(-1)_{10}$
Übertrag	<u>1 1111 111</u>	
Register y'	<u>1 0001 0110</u>	
Ergebnis	0001 0110	$(22)_{10}$

	0011 0111	$(55)_{10}$
	+ 1101 0110	$(-42)_{10}$
	<u>1 111 11</u>	
	<u>1 0000 1101</u>	
	0000 1101	$(13)_{10}$

	1110 1111	$(-17)_{10}$
	+ 1111 1101	$(-3)_{10}$
Übertrag	<u>1 1111 111</u>	
Register y'	<u>1 1110 1100</u>	
Ergebnis	1110 1100	$(-20)_{10}$

	1110 1111	$(-17)_{10}$
	+ 0000 0011	$(3)_{10}$
	<u>1 111</u>	
	<u>0 1111 0010</u>	
	1111 0010	$(-14)_{10}$

Hörsaalfrage



24 82 94 16

Welche Bitfolge entspricht der Zweierkomplement-Darstellung der Zahl -4 bei $n = 8$ Bit?

- 0000 1011
- 0000 1100
- 1011 1111
- 1101 1111
- 1111 1011
- **1111 1100** denn: $0000\ 0100 \Rightarrow 1111\ 1011 + 1 = 1111\ 1100$

Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

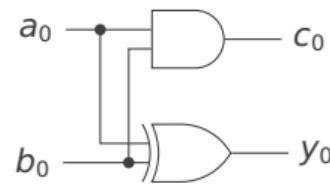
Halbaddierer (engl. Half Adder)

- Ermittelt aus a_0 und b_0 Summe y_0 und Übertrag c_0 .
- Einsatz für niederwertigste Bits.
- Verzögerung: τ für c_0 und 2τ für y_0 .

Wahrheitstabelle

a_0	b_0	y_0	c_0
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Realisierung

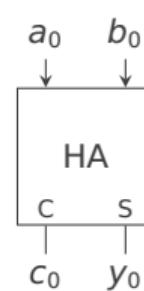


$$c_0 = a_0 \cdot b_0$$

$$y_0 = a_0 \oplus b_0$$

XOR

Symbol



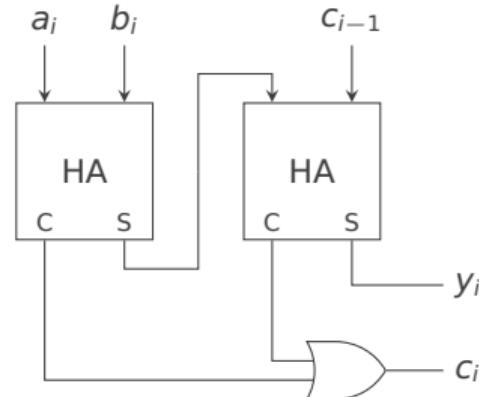
Volladdierer (engl. Full Adder)

- Addiert a_i , b_i und c_{i-1} an Bitpositionen $i = 1, \dots, n - 1$.
- Gibt Summe y_i und Übertrag c_i aus.

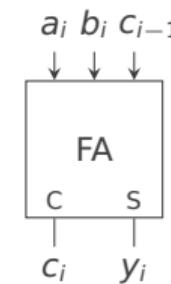
Wahrheitstabelle

a_i	b_i	c_{i-1}	y_i	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Realisierung



Symbol

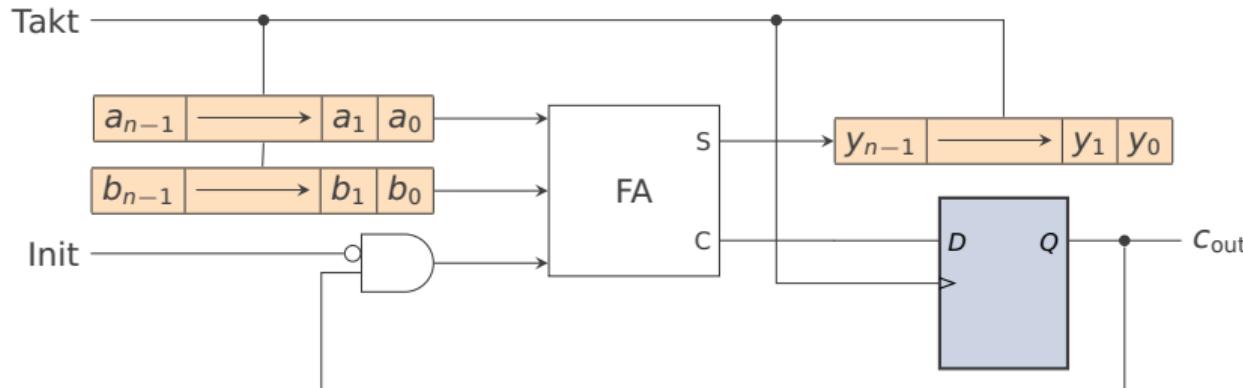


$$c_i = a_i \cdot b_i + a_i \cdot c_{i-1} + b_i \cdot c_{i-1}$$

$$y_i = a_i \oplus b_i \oplus c_{i-1}$$

Serielles Addierwerk

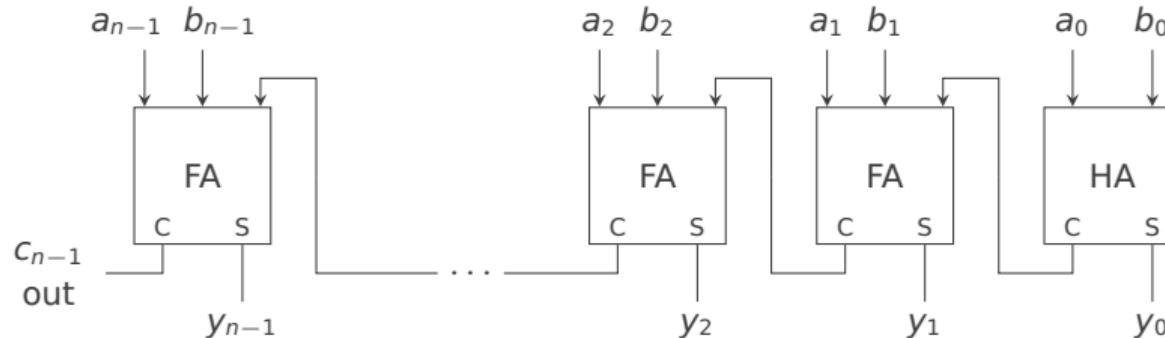
Konstruktion eines **synchronen Schaltwerks** aus einem Volladdierer, einem Flipflop und drei n -Bit-Schieberegistern:



- Der Init-Eingang dient zum Löschen des Übertrags, falls das Flipflop nicht initialisiert ist.
- In Takt t wird Ergebnisbit y_t aus a_t , b_t und c_{t-1} bestimmt.
- Die Addition von zwei n -Bit-Zahlen benötigt n Taktzyklen.

Paralleles Addierwerk

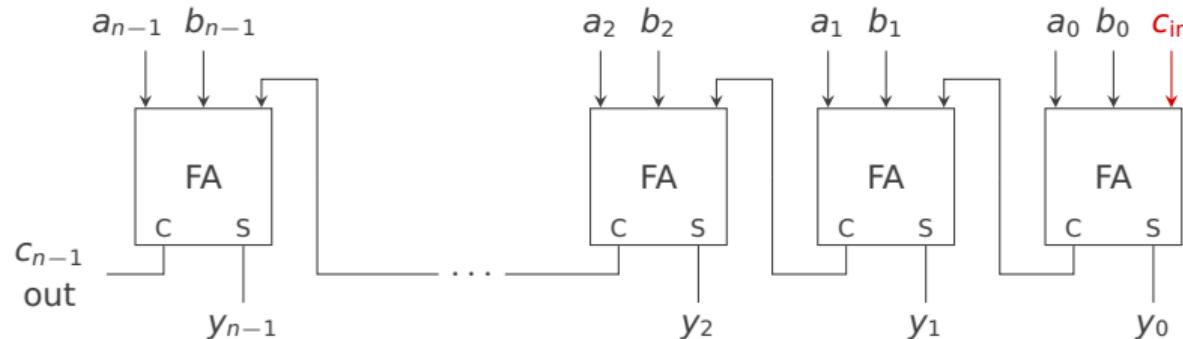
Konstruktion aus $n - 1$ Voll- und einem Halbaddierer:



- Übertrag an Position $i = 0$ kann alle Bitstellen 1 bis $n - 1$ durchlaufen, daher: “Ripple Carry”-Addierer (RCA)
- Maximale Verzögerung: $2n \cdot \tau$
- Verbesserung in der Praxis: “Carry Look-Ahead”-Addierer (CLA) addieren mit konstanter Verzögerung.

Paralleles Addierwerk

Konstruktion aus n Volladdierern **mit Übertrag-Eingang:**

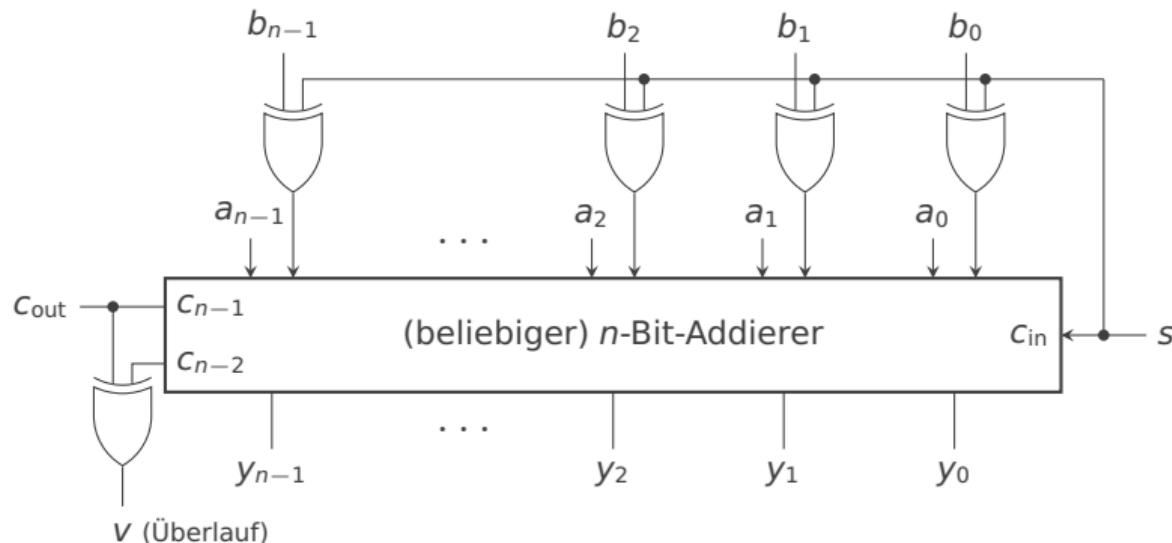


- Übertrag an Position $i = 0$ kann alle Bitstellen 1 bis $n - 1$ durchlaufen, daher: "Ripple Carry"-Addierer (RCA)
- Maximale Verzögerung: $2n \cdot \tau$
- Verbesserung in der Praxis: "Carry Look-Ahead"-Addierer (CLA) addieren mit konstanter Verzögerung.

Kombiniertes Addier-/Subtrahierwerk

Steuereingang s wählt zwischen **Addition** ($a + b$) für $s = 0$ und **Subtraktion** ($a - b$) für $s = 1$.

Idee: XOR-Gatter invertieren Bits b_i wenn $s = 1$.

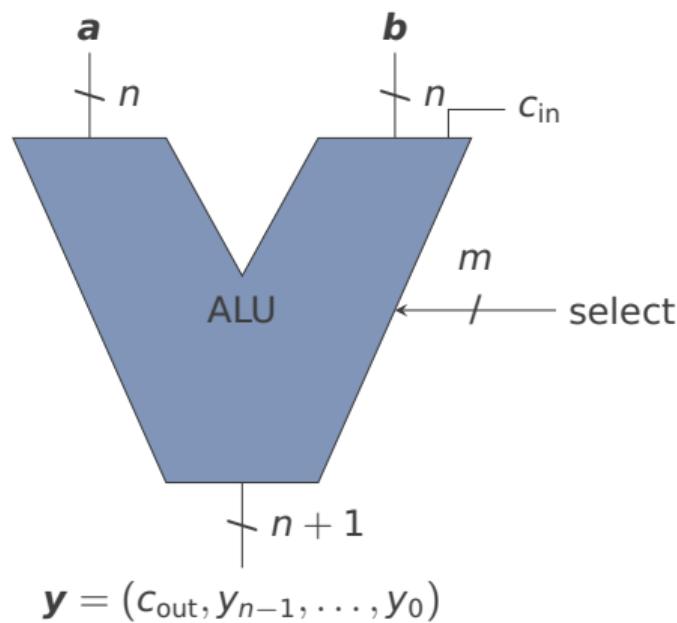


Gliederung heute

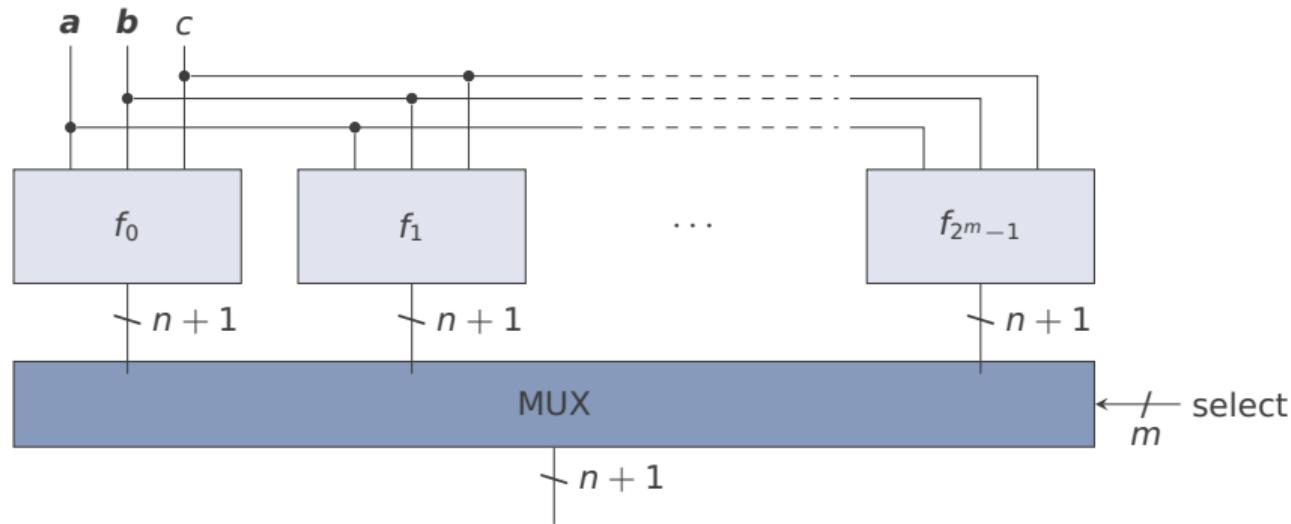
1. Addition und Subtraktion
2. **Arithmetisch-logische Einheit**

Arithmetisch-logische Einheit (ALU)

Multifunktionsmodul für Verknüpfungen zwischen n -Bit-Registern



Schematischer Schaltungsaufbau



Weitere Strukturierung der Select-Eingänge s_i sinnvoll, z. B.

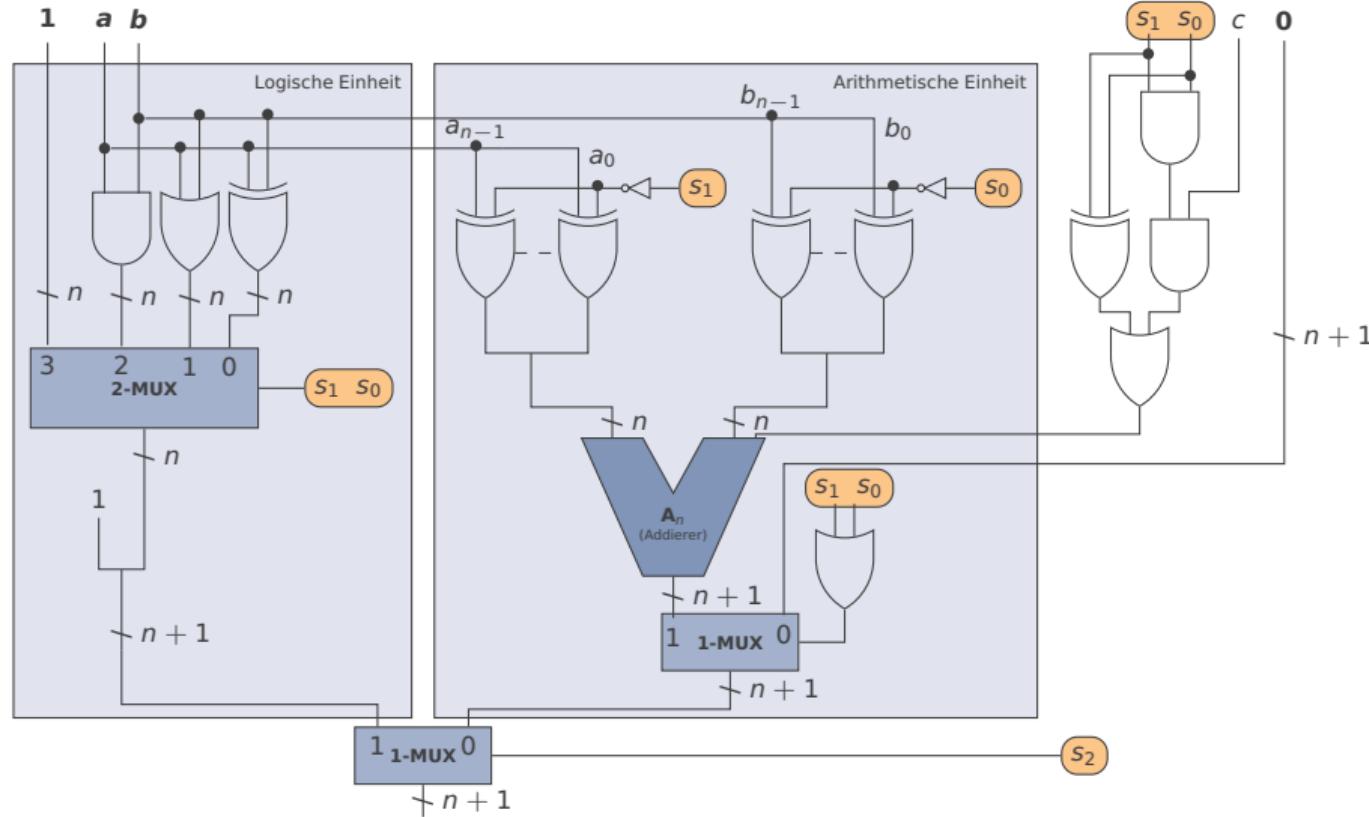
- s_2 entscheidet zwischen arithmetischen und logischen Operationen;
- s_1 und s_0 wählen konkrete (arithmetische oder logische) Operation.

Wahl der Operation

Beispiel für die Belegung des Select-Eingangs:

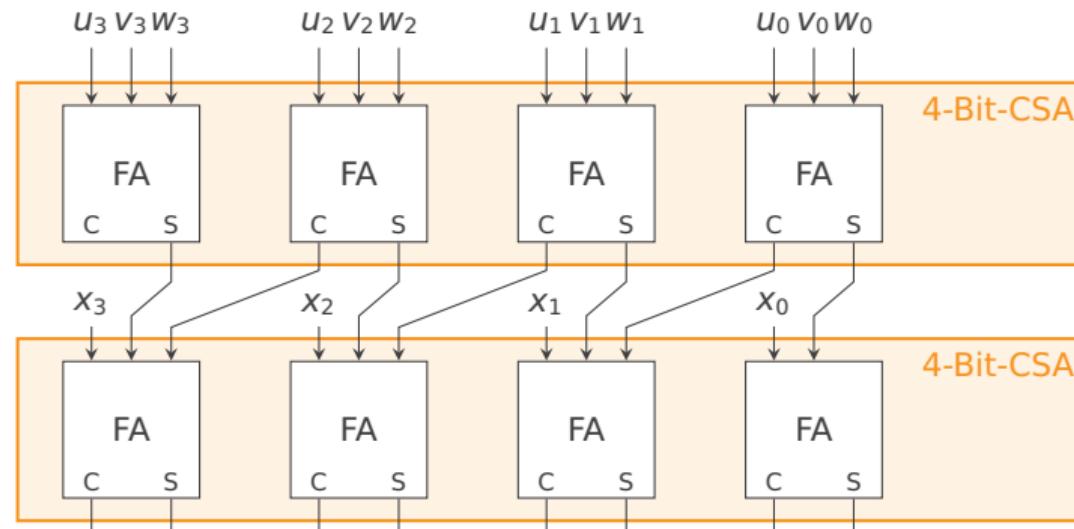
s_2	s_1	s_0	Funktion
0	0	0	0
0	0	1	b - a
0	1	0	a - b
0	1	1	a + b + c
<hr/>			
1	0	0	a \oplus b
1	0	1	a \vee b
1	1	0	a \wedge b
1	1	1	1
<hr/>			

Schaltungstechnische Realisierung einer ALU



Ausblick: Carry-Save-Addierer (CSA)

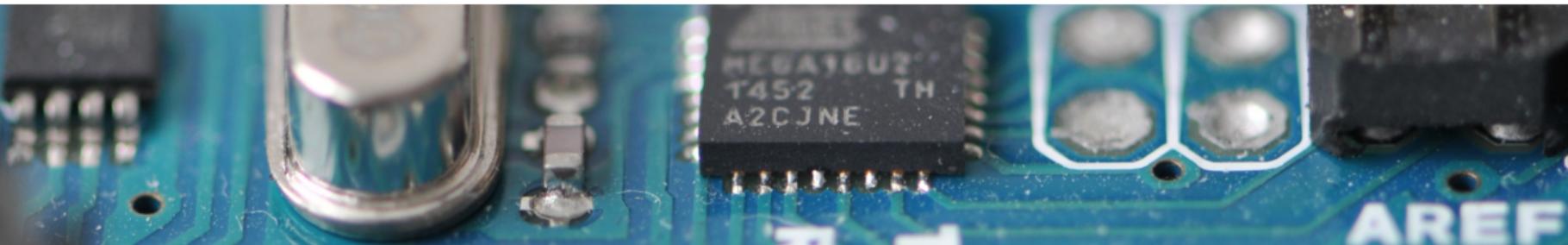
Anordnung zur **partiellen Addition**: Ein CSA-Baustein integriert n Volladdierer mit **separaten** Carry-Ausgängen



→ Kaskadierung zur schnellen Addition mehrerer Summanden

Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Arithmetik II

Univ.-Prof. Dr.-Ing. Rainer Böhme

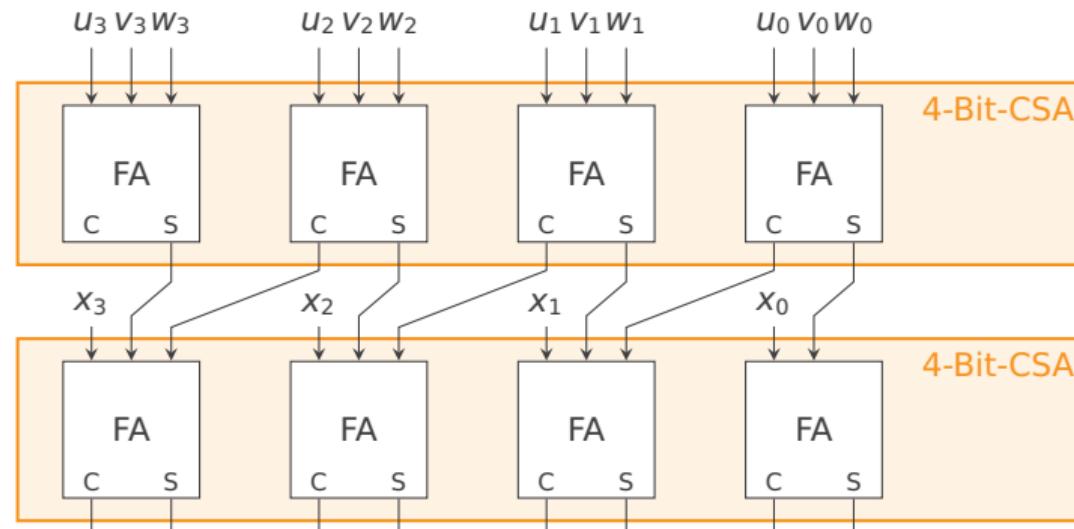
Wintersemester 2021/22 · 17. November 2021

Gliederung heute

- 1. Multiplikation**
- 2. Division**
- 3. Rechnen mit Nachkommastellen**

Carry-Save-Addierer (CSA)

Anordnung zur **partiellen Addition**: Ein CSA-Baustein integriert n Volladdierer mit **separaten Carry-Ausgängen**



→ Kaskadierung zur schnellen Addition mehrerer Summanden

Konstruktion von Multiplizierern

Ansätze (Auswahl)

- a. Zweistufiges Schaltnetz (vgl. VL Kombinatorische Logik II)
- b. Serielles Schaltwerk (vgl. serielles Addierwerk)
- c. Feldmultiplizierer (vgl. paralleles Addierwerk)
- d. Optimierungsmöglichkeiten

Multiplikation als Schaltnetz

$n \times n$ -Bit-Multiplizierer als Schaltnetz mit je $2n$ Ein- und Ausgängen

- Implementierung z. B. in PROM mit 2^{2n} Zeilen aus $2n$ -Bit-Worten
- **Sehr geringe Zeitverzögerung:** zwei Stufen \Rightarrow ca. 2τ
- **Sehr hoher Schaltungs-/Speicheraufwand**

n	Produkt: $2n$	Zeilen: 2^{2n}	(P)ROM Größe
2	4	16	64 Bit
4	8	256	256 Byte
8	16	65 536	128 Kilobyte
16	32	$4.3 \cdot 10^9$	16 Gigabyte
32	64	$1.8 \cdot 10^{19}$	148 Exabyte

→ „Skaliert nicht.“ Präziser: Speicheraufwand exponentiell in n

Multiplikation positiver Binärzahlen

Das Produkt $y = a \times b$ aus zwei n -Bit-Faktoren hat $2n$ Stellen.

Algorithmus wie bei schriftlicher Dezimal-Multiplikation:

Pseudocode

```
y ← 0
for i = 0 to n – 1 do
    if  $b_i = 1$  then
        x ← a um i Bit nach
            links verschoben
        y ← y + x
    end if
end for
```

Beispiel für $n = 5$

$$\begin{array}{r} a \times b: & 01010 & \times & 01101 \\ & 01010 & & \times 1 & b_0 \\ & 00000 & & \times 0 & b_1 \\ & 01010 & & \times 1 & b_2 \\ & 01010 & & \times 1 & b_3 \\ & 00000 & & \times 0 & b_4 \\ \hline y = & 001000010 \end{array}$$

→ Zurückführung auf bedingte Addition und Schiebeoperationen

Modifizierter Algorithmus

Ersetzen der Linksverschiebung von a durch Rechtsverschiebung von y .

Pseudocode

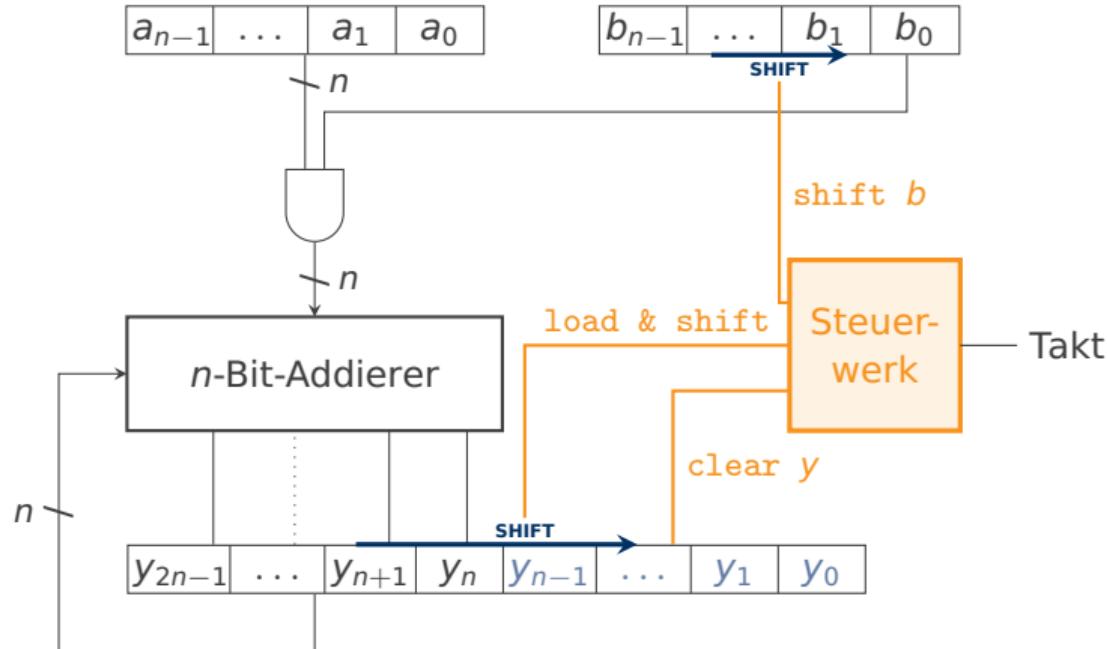
```
 $y \leftarrow 0$ 
for  $i = 0$  to  $n - 1$  do
    if  $b_i = 1$  then
         $(y_{2n-1}, \dots, y_n) \leftarrow (y_{2n-1}, \dots, y_n) + a$ 
    end if
    {Verschiebe  $y$  um ein Bit nach rechts.}
     $(y_{2n-1}, \dots, y_0) \leftarrow (0, y_{2n-1}, \dots, y_1)$ 
end for
```

Beispiel

$$\begin{array}{r} 01010 \quad \times \quad 01101 \\ \hline 0000000000 \\ + 01010 \quad \text{add } a \\ \hline 0101000000 \\ 0010100000 \text{ shift} \\ 0001010000 \text{ shift} \\ + 01010 \quad \text{add } a \\ \hline 0110010000 \\ 0011001000 \text{ shift} \\ + 01010 \quad \text{add } a \\ \hline 1000001000 \\ 0100000100 \text{ shift} \\ y = 001000010 \text{ shift} \end{array}$$

Vorteil: Nur ein Schieberegister mit $2n$ Bit

Realisierung als serielles Multiplizierwerk



Die Berechnung dauert n Taktzyklen.

Hörsaalfrage



Sei $n = 3$, $a = (101)_2$ und $b = (010)_2$.

Welche Folge aus Steuersignalen führt eine erfolgreiche Multiplikation $y = a \times b$ aus?

24 82 94 16

Antwort A	Antwort B	Antwort C
1. clear y	1. clear y	1. clear y
2. shift b	2. load & shift	2. load & shift
3. load & shift	3. shift b	3. shift b
4. shift b	4. load & shift	4. load & shift
5. load & shift	5. shift b	5. shift b
6. shift b	6. load & shift	6. load & shift
7. load & shift	7. shift b	7. clear y

Hinweis: Nehmen Sie an, dass pro Takt nur ein Steuersignal erzeugt wird.
(In der Praxis müssen mehrere Signale kombiniert werden, um die genannte Laufzeit von n Takten zu erreichen. Überlegen Sie als **Hausaufgabe**, welche.)

Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Code.

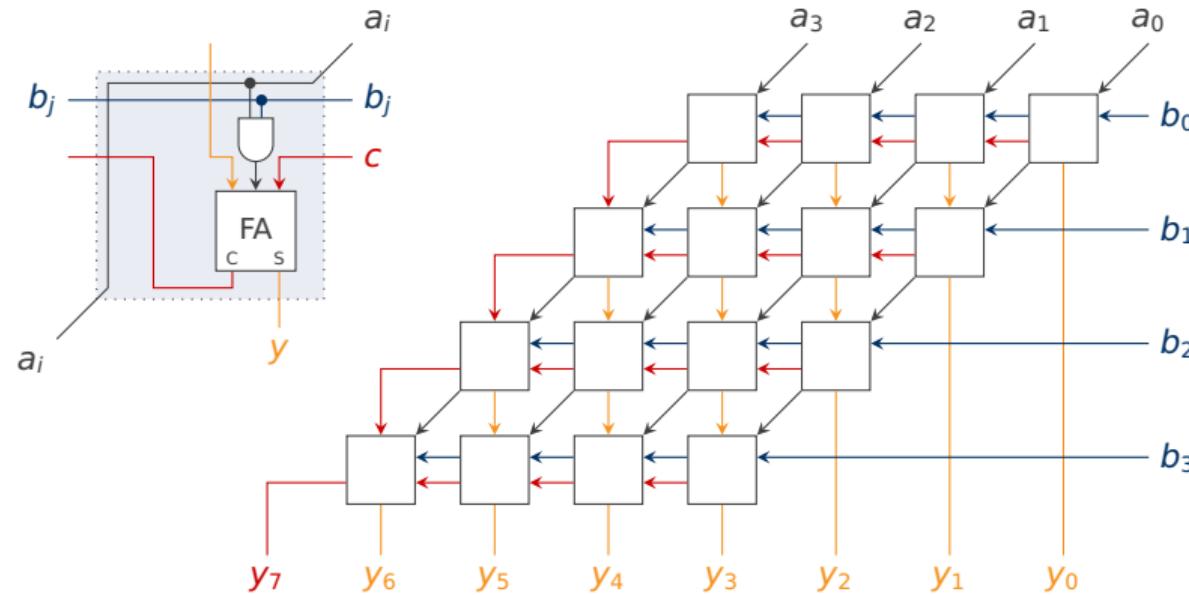
Direkte Schaltung einer schriftlichen Multiplikation

(a ₃ , a ₂ , a ₁ , a ₀)				×		(b ₃ , b ₂ , b ₁ , b ₀)			
						a ₃ b ₀	a ₂ b ₀	a ₁ b ₀	a ₀ b ₀
						a ₃ b ₁	a ₂ b ₁	a ₁ b ₁	a ₀ b ₁
						a ₃ b ₂	a ₂ b ₂	a ₁ b ₂	a ₀ b ₂
						a ₃ b ₃	a ₂ b ₃	a ₁ b ₃	a ₀ b ₃
<i>n² Bitprodukte</i>									

Feldmultiplizierer

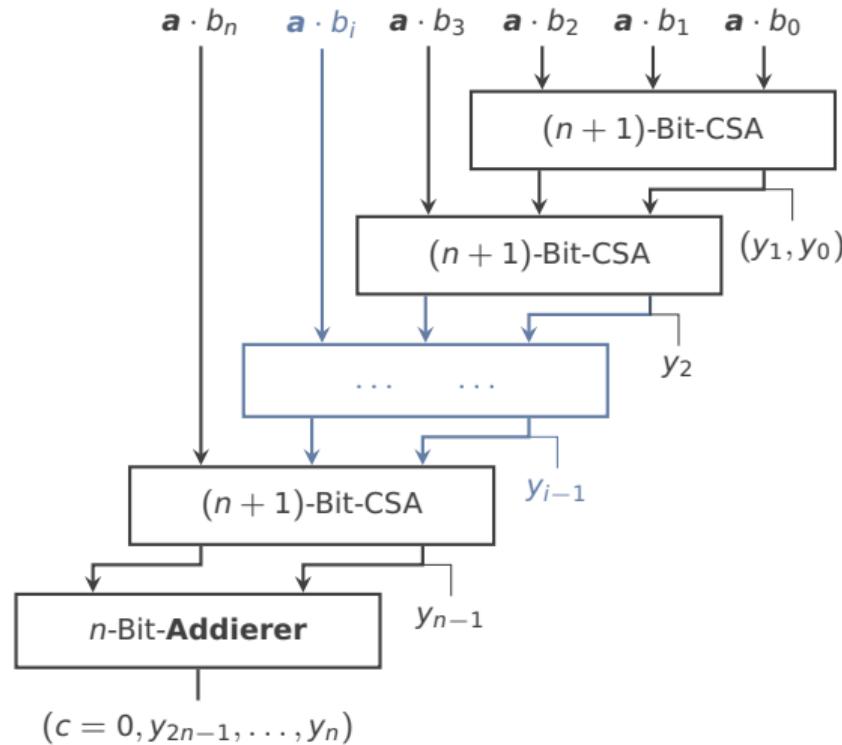
(engl. array multiplier)

Strukturierter Aufbau aus n^2 Multiplizierzellen



Variante mit CSA-Kaskade

(Skizze)



Multiplikation negativer Zahlen

Bislang Beschränkung auf **positive** Faktoren. Was passiert bei der Multiplikation von **negativen** Zahlen in **Zweierkomplement**-Darstellung?

$$a \cdot (-b) \equiv a \cdot (2^n - b) = a \cdot 2^n - a \cdot b \quad [\text{statt } 2^{2n} - a \cdot b]$$

$$(-a) \cdot b \equiv (2^n - a) \cdot b = b \cdot 2^n - a \cdot b \quad [\text{statt } 2^{2n} - a \cdot b]$$

$$(-a) \cdot (-b) \equiv (2^n - a) \cdot (2^n - b) = 2^{2n} - a \cdot 2^n - b \cdot 2^n + a \cdot b \quad [a \cdot b]$$

→ **Naives Multiplizieren liefert falsche Ergebnisse.**

Lösungsansätze

1. Trennung von Vorzeichen und Betrag
2. Addition von Korrekturtermen
3. Algorithmus von Booth (mit vorzeichenrichtiger Ergänzung)

Optimierungsmöglichkeiten

Beobachtung: Jede Eins in b kostet eine Addition.

$$a \times 111111 \quad \text{vs.} \quad a \times 100001$$

$$a \times 111111 = a \times 1000000 - a \times 0000001$$

Die Multiplikation mit einer 1-Folge kann immer durch
eine Addition und **eine** Subtraktion ersetzt werden.

Anwendung

Der Algorithmus von **Booth** analysiert zwei benachbarte Bits b_i und b_{i-1} :

11 nichts tun (wie nach wie vor bei 00)

10 Subtraktion von $a \times 2^i$

01 Addition von $a \times 2^i$

Wichtig um den Anfang (von rechts) einer Folge nicht zu verpassen:

Initiale Ergänzung $b_{-1} = 0$, falls $b_0 = 1$.

Booth 1951

Algorithmus von Booth

Beispiel für $n = 8$:

$$42 \times 92 = (0010\ 1010)_2 \times (0101\ 1100)_2$$

0010 1010	\times	0101 1100	
1111 11	11 0101 10	\leftarrow	0101 1100
			0101 1100
			0101 1100
0000 0101 010	\leftarrow	0101 1100	
11 11 0101 10	\leftarrow	0101 1100	
0001 0101 0	\leftarrow	0101 1100	
1 0000 1111 0001 1000	=	(3864) ₁₀	

Weitere Beispiele

mit negativen Faktoren und initialer Ergänzung ($n = 5$):

$$(10)_{10} = (01010)_2$$

$$(-10)_{10} = (10110)_2$$

$$(-13)_{10} = (10011)_2$$

$$(10)_{10} \times (-13)_{10}$$

$$\begin{array}{r} 01010 \\ \times 100110 \\ \hline \end{array} = b_{-1}$$

$$\begin{array}{r} 111110110 \leftarrow 100110 \\ 00001010 \leftarrow 100110 \\ 110110 \leftarrow 100110 \\ \hline 1110111110 = (-130)_{10} \end{array}$$

$$(-10)_{10} \times (-13)_{10}$$

$$\begin{array}{r} 10110 \\ \times 100110 \\ \hline \end{array} = b_{-1}$$

$$\begin{array}{r} 0000001010 \leftarrow 100110 \\ 11110110 \leftarrow 100110 \\ 001010 \leftarrow 100110 \\ \hline 10010000010 = (130)_{10} \end{array}$$

Gliederung heute

1. Multiplikation
2. **Division**
3. Rechnen mit Nachkommastellen

Schriftliche Division von Binärzahlen mit Rest

Beispiel $29 : 6 = 4 \text{ Rest } 5$

$n = 5$ Bits

$$\begin{array}{r} 000011101 : 00110 = 00100 \\ - 00110 \\ \hline 11011 \end{array}$$

Quotient

Korrektur

$$\begin{array}{r} + 00110 \\ \hline 00011 \\ - 00110 \\ \hline 11101 \end{array}$$

Korrektur

$$\begin{array}{r} + 00110 \\ \hline 000111 \\ - 00110 \\ \hline 000010 \\ - 00110 \\ \hline 11100 \end{array}$$

Korrektur

$$\begin{array}{r} + 00110 \\ \hline 000101 \\ - 00110 \\ \hline 11111 \end{array}$$

Korrektur

$$\begin{array}{r} + 00110 \\ \hline 00101 \end{array}$$

Rest

Divisionsalgorithmus mit „Restoring“

Verwendung von bedingter Addition, **Subtraktion** und Schiebeoperationen

Pseudocode

Require: Dividend a , Divisor b (jeweils n Bit)

$(y_{n-1}, \dots, y_0) \leftarrow a$ {Initialisiere (“load”) 2n-Bit-Register y .}

$(y_{2n-1}, \dots, y_n) \leftarrow 0$

for $i = 0$ to $n - 1$ **do**

$(y_{2n-1}, \dots, y_0) \leftarrow (y_{2n-2}, \dots, y_0, 0)$ {Schiebe y um ein Bit nach links.}

$(y_{2n-1}, \dots, y_n) \leftarrow (y_{2n-1}, \dots, y_n) - b$

if $y_{2n-1} = 0$ **then**

$y_0 \leftarrow 1$

else

$(y_{2n-1}, \dots, y_n) \leftarrow (y_{2n-1}, \dots, y_n) + b$ {Wiederherstellung des Rests}

end if

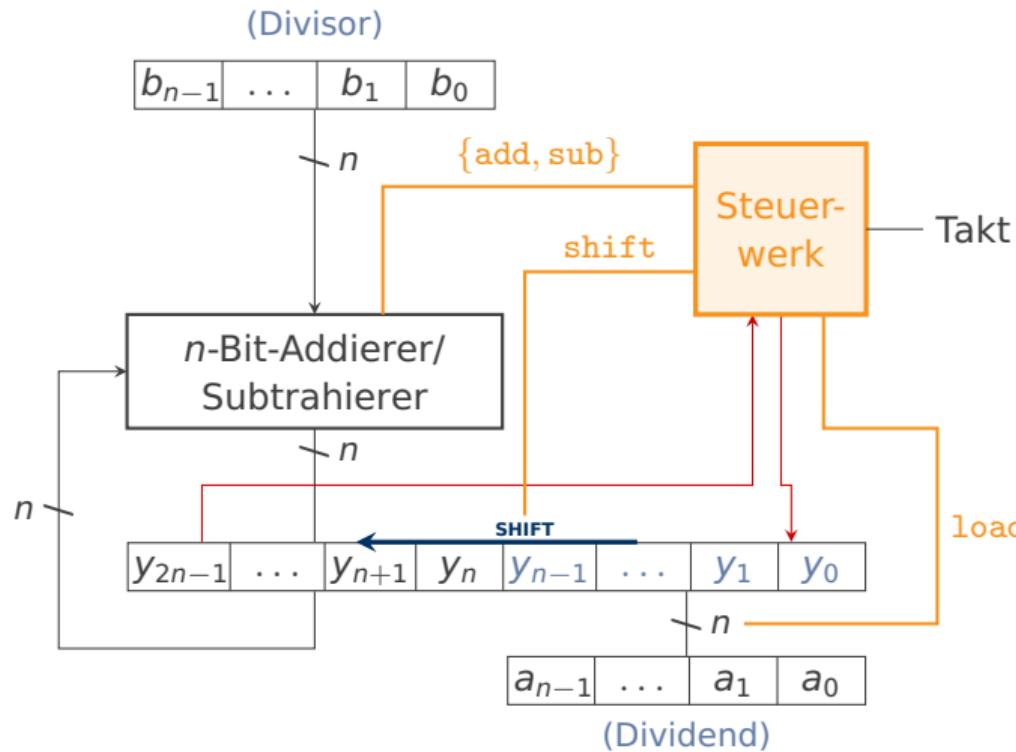
end for

$r \leftarrow (y_{2n-1}, \dots, y_n)$

$q \leftarrow (y_{n-1}, \dots, y_0)$

{Ergebnis. Es gilt: $a = b \times q + r$ }

Realisierung als serielles Dividierwerk



Steuersignale für Division mit „Restoring“

Beispiel $29 : 6 = 4 \text{ Rest } 5$

$n = 5 \text{ Bits}$	$000011101 : 00110 = 00100$	$00000\ 11101$	load
	$\begin{array}{r} - 00110 \\ \hline 11011 \end{array}$	$00001\ 11010$	shift
Korrektur	$\begin{array}{r} + 00110 \\ \hline 00011 \end{array}$	$11011\ 11010$	sub
	$\begin{array}{r} - 00110 \\ \hline 11101 \end{array}$	$00001\ 11010$	add
Korrektur	$\begin{array}{r} + 00110 \\ \hline 000111 \end{array}$	$00011\ 10100$	shift
	$\begin{array}{r} - 00110 \\ \hline 000010 \end{array}$	$11101\ 10100$	sub
Korrektur	$\begin{array}{r} + 00110 \\ \hline 000110 \end{array}$	$00011\ 10100$	add
	$\begin{array}{r} - 00110 \\ \hline 11100 \end{array}$	$00111\ 01000$	shift
Korrektur	$\begin{array}{r} + 00110 \\ \hline 000101 \end{array}$	$00001\ 01001$	sub
	$\begin{array}{r} - 00110 \\ \hline 11111 \end{array}$	$00010\ 10010$	shift
Korrektur	$\begin{array}{r} + 00110 \\ \hline 00101 \end{array}$	$11111\ 00100$	sub
Rest		$00101\ 00100$	add

Optimierungsmöglichkeiten bei der Division

Verbesserungen der „langsamem Verfahren“

(Berechnung von 1–2 Ergebnisstellen pro Iteration)

- CSA statt Addierer/Subtrahierer und Verrechnung der Überträge bei Korrektur bzw. im Folgeschritt (\rightarrow SRT-Division)
- Radix-4-Zahlendarstellung $\{-2, -1, 0, +1, +2\}$ oft kombiniert mit Quotienten-Tabelle in ROM (\rightarrow Intel Pentium-FDIV-Bug 1994)

“Is there a list of Pentium jokes? I need one! :-)"

“No, no. You meant to say you need .999856738903.”

„**Schnelle Verfahren**“ beginnen mit einer Schätzung und verdoppeln die Genauigkeit in jedem Schritt

- Das **Goldschmidt**-Verfahren multipliziert Dividend und Divisor mit Faktoren bis Divisor zum Wert 1 konvergiert.
- Die **Newton-Raphson**-Division sucht Kehrwert und multipliziert.

SRT: Sweeney, Robertson, Tochter (1958)

Gliederung heute

1. Multiplikation
2. Division
- 3. Rechnen mit Nachkommastellen**

Darstellung rationaler und reeller Zahlen

- 1. Festkomma:** Jede (darstellbare) Kommazahlen z wird durch lineare Skalierung auf eine ganze Zahl z' abgebildet.
Rechner arbeitet unverändert auf ganzzahliger Abbildung.
- 2. Gleitkomma:** Darstellung von Kommazahlen durch Argument (Mantisse) a und Charakteristik (Exponent) c zur Basis r :

$$z = a \times r^c \quad \text{Bsp. für } r = 10: 0.000035 \Rightarrow 3.5 \times 10^{-5}$$

Spezielle Rechenwerke erforderlich (oder Realisierung in Software)

- 3. Exakte Darstellung rationaler Zahlen** durch separate Speicherung und Verarbeitung von Zähler und Nenner als Ganzzahlen

Realisiert in Software für exaktes wissenschaftliches Rechnen
(hier nicht weiter behandelt)

Festkommazahlen

Zahl zur Basis b mit **fester Anzahl** $k < n$ Stellen **nach** dem Komma:

$$\begin{aligned} z &= \underbrace{(z_{n-1}, z_{n-2}, \dots, z_{k+1}, z_k)}_{\text{ganzzahliger Teil}} \cdot \underbrace{z_{k-1}, z_{k-2}, \dots, z_1, z_0}_b \\ &= z_{n-1} \cdot b^{n-k} + z_{n-2} \cdot b^{n-k-1} + \dots + z_{k+1} \cdot b^1 + z_k \cdot b^0 + \\ &\quad z_{k-1} \cdot b^{-1} + z_{k-2} \cdot b^{-2} + \dots + z_1 \cdot b^{-k+1} + z_0 \cdot b^{-k} \end{aligned}$$

- Die Konstante k kennt nur die Anwendung. Sie dient der **Interpretation** der Zahlen.
- Die Rechenwerke arbeiten **transparent** mit skalierten ganzen Binärzahlen $z' = z \cdot 2^k$.

Beispiel und Hörsaalfrage



Ein 8-Bit-Register enthält die Binärzahl $z' = (01101110)_2$.

24 82 94 16

Für $k = 3$ gilt:

$$z = (01101.110)_2 = 2^3 + 2^2 + 2^0 + 2^{-1} + 2^{-2} = 13.75$$

Hörsaalfrage

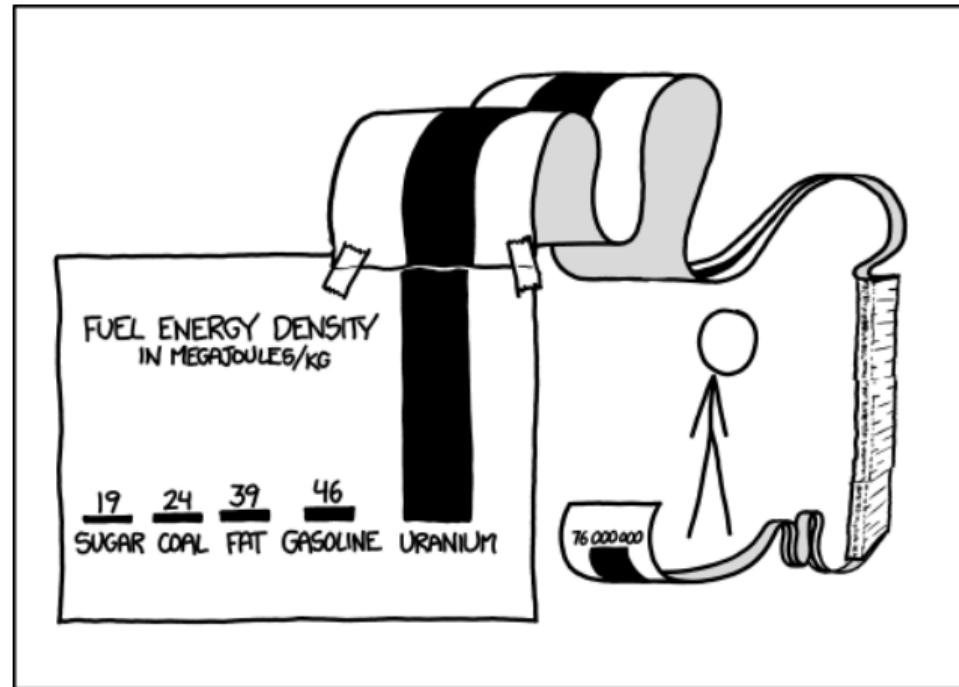
Welche Bitfolge ist die Festkommadarstellung von $z = 13.1875$ mit $k = 4$?

1. 00111101
2. 11010110
3. 011010010
4. 011010011
5. 110100011

Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Gleitkommazahlen

(auch Fließkomma, engl. *floating point*)



SCIENCE TIP: LOG SCALES ARE FOR QUITTERS WHO CAN'T
FIND ENOUGH PAPER TO MAKE THEIR POINT PROPERLY.

FLIPFLOPS

Gleitkommazahlen zur Basis $r = 2$

Allgemeine Darstellung nach IEEE 754

$$z = (-1)^s \times 1.f \times 2^{e-b}$$

Binärformat

$$\overbrace{(s, e_{p-1}, \dots, e_1, e_0, f_{m-1}, \dots, f_1, f_0)}^{1 + p + m = n \text{ Bit}}$$

- **Mantisse** aus Vorzeichen s und **normalisiertem Betrag** $a = 1.f$ im Bereich $1.00 \dots 00$ bis $1.11 \dots 11$ ohne die führende Eins
- **Exponent** e mit konstantem **Bias** $b = 2^{p-1} - 1 \geq 0$
- Darstellbarer Zahlenbereich: $\pm 2^{1-b}, \dots, (2 - 2^{-m}) \times 2^b$
- Zwischen 2^{e-b} und 2^{e-b+1} gibt es 2^m Gleitkommazahlen, deren Abstand von e abhängt.

IEEE 754

Standardisierte Formate für die Gleitkommazahlendarstellung

Genauigkeit		single	double	quad
Gesamtbreite	n [Bit]	32	64	128
davon:				
Mantisse	m	23	52	112
Exponent	p	8	11	15
Vorzeichen		1	1	1
Bias	b	127	1 023	16 383
Minimum (Betrag)	$ z_{\min} $	2^{-126} $\approx 10^{-38}$	2^{-1022} $\approx 10^{-308}$	2^{-16382} $\approx 10^{-4932}$
Maximum (Betrag)	$ z_{\max} $	$\approx 10^{38}$ $(2 - 2^{-23}) \times 2^{127}$	$\approx 10^{308}$ $(2 - 2^{-52}) \times 2^{1023}$	$\approx 10^{4932}$ $(2 - 2^{-112}) \times 2^{16383}$
gültige Dezimalstellen		7.22	15.95	34.02

Kodierung besonderer Zahlen

IEEE 754 definiert Spezialfälle, die mit $e = \mathbf{0}$ oder $e = \mathbf{1}$ kodiert werden:

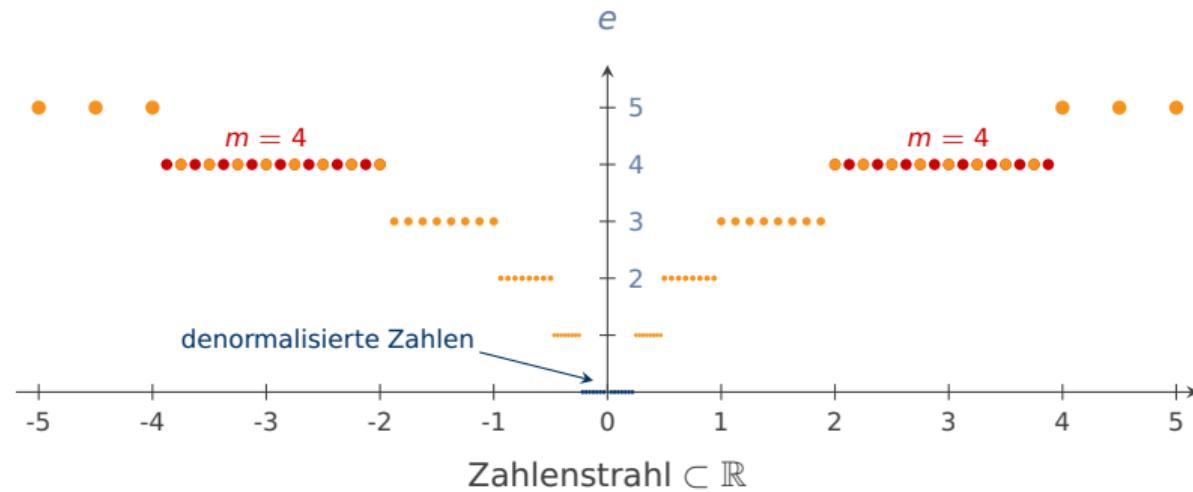
Zahl	Bezeichnung	Kodierung		
		e	f	s
$z = +0$	<i>positive zero</i>	0	0	0
$z = -0$	<i>negative zero</i>	0	0	1
$z = +\infty$	<i>positive infinity</i>	1	0	0
$z = -\infty$	<i>negative infinity</i>	1	0	1
$z = \text{NaN}$	<i>not a number</i>	1	$\neq 0$	d
$z = (-1)^s \times \mathbf{0}.f \times 2^{1-b}$	<i>denormalized number</i>	0	$\neq 0$	$\{0, 1\}$

Visualisierung

$$z = (-1)^s \times 1.\textcolor{orange}{f} \times 2^{e-b}$$

Beispiele für $p = 3, m = 3$

$$\Rightarrow b = 2^{3-1} - 1 = 3$$



Ausnahmesituationen

Überlauf, wenn nach Normalisierung für $z : e \geq e_{\max} = 1$

- Ausgabe von $+\infty$, falls $z > 0$; $-\infty$, falls $z < 0$
- Auch bei Division durch Null $\pm x : 0 = \pm\infty$ (falls $x \neq 0$)
- Rechenregeln für ∞ :
 $\infty \pm x = \infty$ (falls $x \neq \mp\infty$), $\infty \cdot x = \pm\infty$ (falls $x \neq 0$)

Unbestimmtes Ergebnis

- $\infty \cdot 0 = \text{NaN}$, $0 : 0 = \text{NaN}$, $\infty - \infty = \text{NaN}$
- Für alle Operationen mit NaN gilt: $F(x, \text{NaN}) = \text{NaN}$

Unterlauf, wenn nach Normalisierung für $z : e = 0$

- Ausgabe einer **denormalisierten** Darstellung von z
- Ausgabe von $z = 0$ ("flushing to zero")

Multiplikation von Gleitkommazahlen

Faktoren: $(-1)^s \times a \times 2^{\alpha-\text{bias}}$ und $(-1)^t \times b \times 2^{\beta-\text{bias}}$

1. Multipliziere Mantissen als Festkommazahlen: $y = a \times b$

$a = 1.f_a$ und $b = 1.f_b$ haben $m + 1$ Stellen $\Rightarrow y$ hat $2m + 2$ Stellen

2. Addiere Exponenten $\gamma = \alpha + \beta - \text{bias}$

3. Berechne Vorzeichen $u = s \oplus t$

4. Normalisiere Produkt $(-1)^u \times y \times 2^{\gamma-\text{bias}}$

i. Falls $y \geq 2$, schiebe y um ein Bit nach rechts und erhöhe γ um 1.

ii. Setze $y = 1.f_y = 1.(y_{2m-1}, y_{2m-2}, \dots, y_m)_2$ mit Rundung.

5. Behandle Ausnahmesituationen

i. Überlauf, falls $\gamma \geq e_{\max} = 2^p - 1 \Rightarrow$ Rückgabe $\pm\infty$ (abh. von u)

ii. Unterlauf, falls $\gamma \leq e_{\min} = 0 \Rightarrow$ Denormalisierung

iii. Zero, falls $y = 0 \Rightarrow$ Rückgabe ± 0 (abh. von u)

Addition von Gleitkommazahlen

Summanden: $(-1)^s \times a \times 2^{\alpha-\text{bias}}$ und $(-1)^t \times b \times 2^{\beta-\text{bias}}$

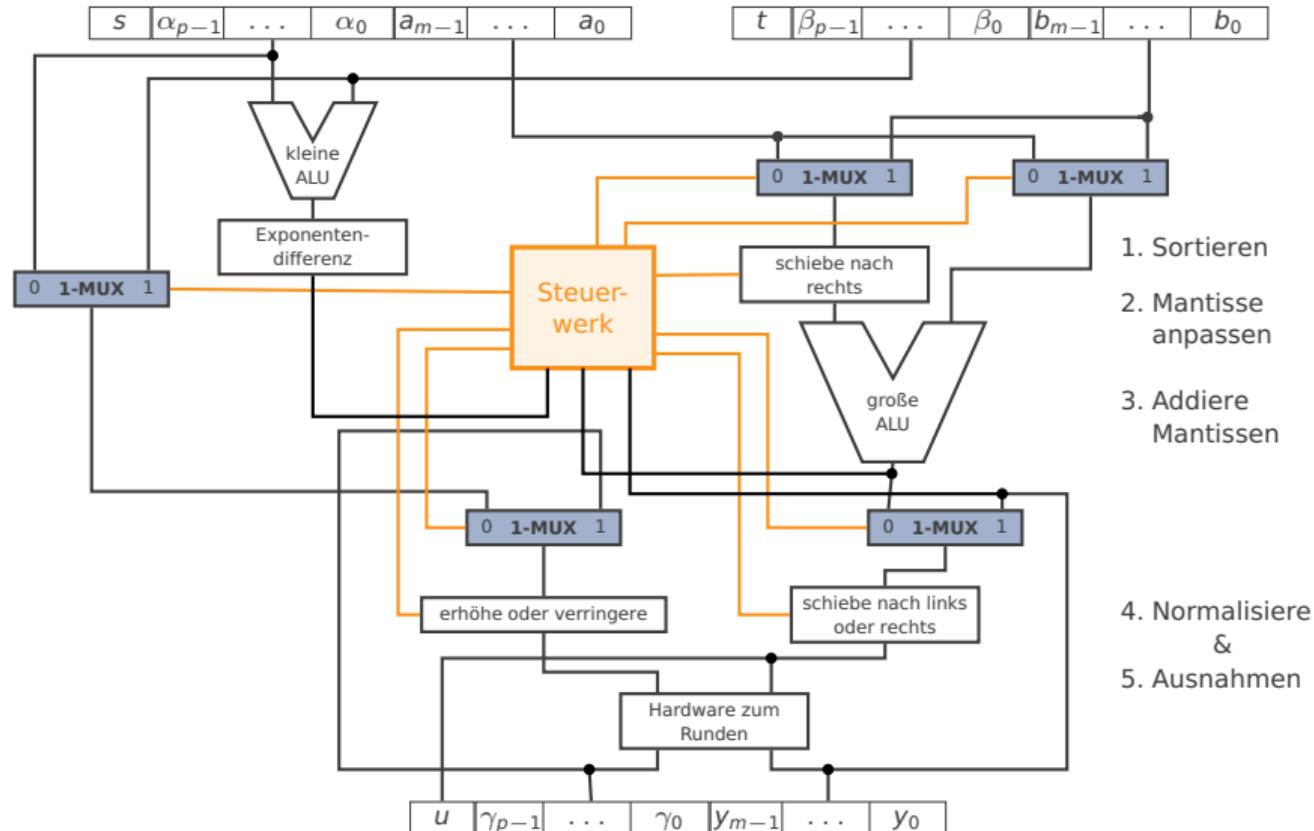
- 1. Sortiere** die Summanden, sodass $\alpha \leq \beta = \gamma$
- 2. Bitposition der Mantisse anpassen:** Bestimme a' , sodass

$$(-1)^s \times a \times 2^{\alpha-\text{bias}} = (-1)^s \times a' \times 2^{\gamma-\text{bias}}$$

durch Rechtsschieben von a um $\beta - \alpha$ Bits.

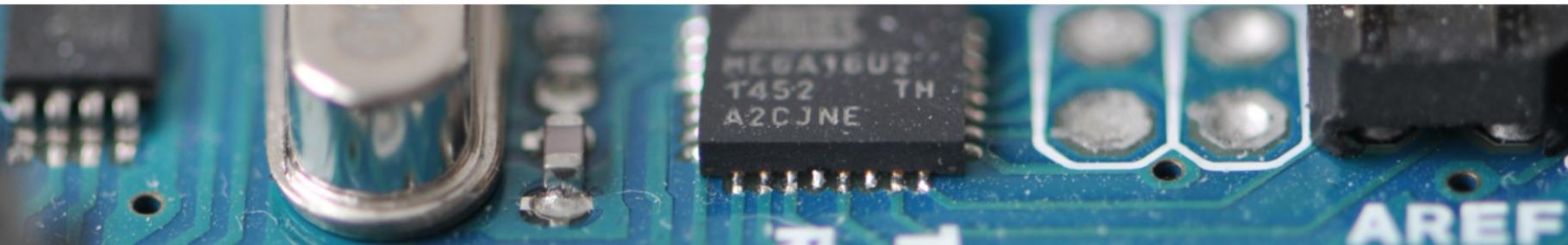
- 3. Addiere Mantissen**
 - Falls nötig, bilde Zweierkomplement von a' oder b (abh. von s und t)
 - Festkomma-Addition $y = a' + b$
 - Falls $y < 0$, setze $u = 1$ und bilde Zweierkomplement von y
- 4. Normalisiere Summe** $(-1)^u \times y \times 2^{\gamma-\text{bias}}$
 - Falls $y \geq 2$, schiebe y nach rechts und erhöhe γ um 1.
 - Solange $y < 1$, schiebe y nach links und verringere γ um 1.
- 5. Behandle Ausnahmesituationen:** Überlauf, Unterlauf, $y = 0$

Skizze eines Gleitkomma-Addierwerks



Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Befehlssatzarchitektur I

Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2021/22 · 24. November 2021

Gliederung heute

1. Von der sequenziellen Logik zum Mikroprozessor
2. **ARM-Mikroarchitektur**
3. ARM-Befehlssatz (ohne Speicherzugriff)
4. Unser erstes Assemblerprogramm

Hintergrund

Sophie Wilson und Steve Furber entwickeln die ARM-Architektur ab 1983 beim englischen Computerhersteller Acorn, heute ARM, in Cambridge.

- ARM stellt keine eigenen Chips her, sondern verkauft Lizenzen an Halbleiterhersteller, die den Prozessor an die Bedürfnisse ihrer Kunden anpassen und mit anderen Komponenten integrieren (z. B. System-on-a-Chip, SoC).
- Einige Lizenznehmer (Apple, Intel, Motorola, NXP etc.) dürfen auch den Kern weiterentwickeln.
- **Folge:** Es gibt eine Vielzahl an ARM-Varianten. → siehe z. B. Wikipedia-Artikel
- Wir behandeln ausgewählte Teile des ARMv6-Designs (32 Bit, 2002).
Es ist bei Mikrocontrollern noch weit verbreitet (z. B. Raspberry Pi).
- Aktuell ist ARMv9 (64 Bit), vorgestellt im März 2021.
- ARMv9 ist abwärtskompatibel bis ARMv5.

Namenskonventionen

Diese Folie dient allein der Orientierung und ist nicht prüfungsrelevant!

ARM unterscheidet Produktfamilien nach Einsatzbereichen:

Cortex-A für Anwendungen (Smartphones, Spielkonsolen)

Cortex-M für Mikrocontroller (Haushaltsgeräte, „Internet der Dinge“)

Cortex-R für Echtzeitanwendungen (Realtime: Automotive, Industriesteuerung)

SecurCore für Sicherheitsanwendungen (Geldautomaten)

In jeder Familie gibt es Produkte, die verschiedene Designs (ARMvX) umsetzen.

ARM-Chips lassen sich mit (bis zu 16) verschiedenen **Koprozessoren** konfigurieren, z. B. für digitale Signalverarbeitung (DSP), Gleitkommaarithmetik (VFP), Java-Hardwarebeschleunigung, Virtualisierung, Speicherverwaltung, ...

Registersatz

CPUs sind Zustandsautomaten. Ihr Zustand wird in wenigen, direkt mit der Logik verbundenen **Registern** gespeichert.

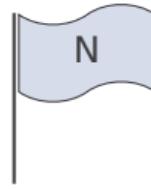
Bei ARM stehen im **User-Modus** 16 Register mit je 32 Bit zur Verfügung:

r0	zur freien Nutzung
r1	zur freien Nutzung
:	
r12	zur freien Nutzung
r13	reserviert für Stack-Pointer (SP)
r14	reserviert für Rücksprungadresse (Link Register, LR)
r15	reserviert für Programmzähler (PC)

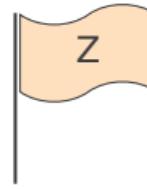
Die Verbindung mit dem (über den Systembus angebundenen)
Arbeitsspeicher erweitert den Zustandsraum erheblich.

Flags

Die ALU setzt Flags (Bits) in einem **Statusregister**.



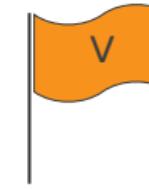
negative



zero



carry



overflow

Arithmetische Operationen

N = höchstwertiges Ergebnisbit

$Z = 1$: Ergebnis ist Null

$C = 1$: Übertrag; Ergebnis > 32 Bit

$V = 1$: arithmetischer Überlauf

Logische Operationen

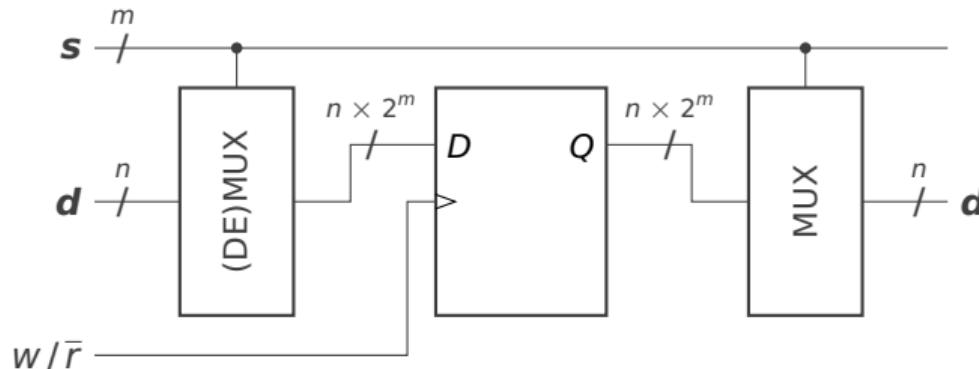
N = höchstwertiges Ergebnisbit

$Z = 1$: alle Bits im Ergebnis sind 0

C = Wert des hinaus geschobenen Bits einer Schiebeoperation

keine Bedeutung

Einfaches Speichermodell



Speicheranbindung des Prozessors über

- Datenbus d der Breite n Bits, oft gleich der Registerbreite
- Adressbus s der Breite m Bits

Beispiele

- Für $n = 8, m = 20: 2^{20} \times 8 \text{ Bit} = 1 \text{ MB}$ adressierbarer Speicher
- Unser Modell-ARM sei $n = 32, m = 26: 2^{26} \times 8 \text{ Bit} = 64 \text{ MB}$
Adressraum, mit **Byte**-genauer Adressierung von 32-Bit-Wörtern

“Endianness” und “Alignment”

Adresse	Little-Endian	Big-Endian	
⋮		aligned	nicht aligned
0x003F0013	X_{31}, \dots, X_4	X_7, \dots, X_0	X_7, \dots, X_0
0x003F0012	X_{23}, \dots, X_{16}	X_{15}, \dots, X_8	X_{15}, \dots, X_8
0x003F0011	X_{15}, \dots, X_8	X_{23}, \dots, X_{16}	
0x003F0010	X_7, \dots, X_0	X_{31}, \dots, X_{24}	
⋮	$n = 32$	$n = 32$	$n = 16$
			$n = 16$

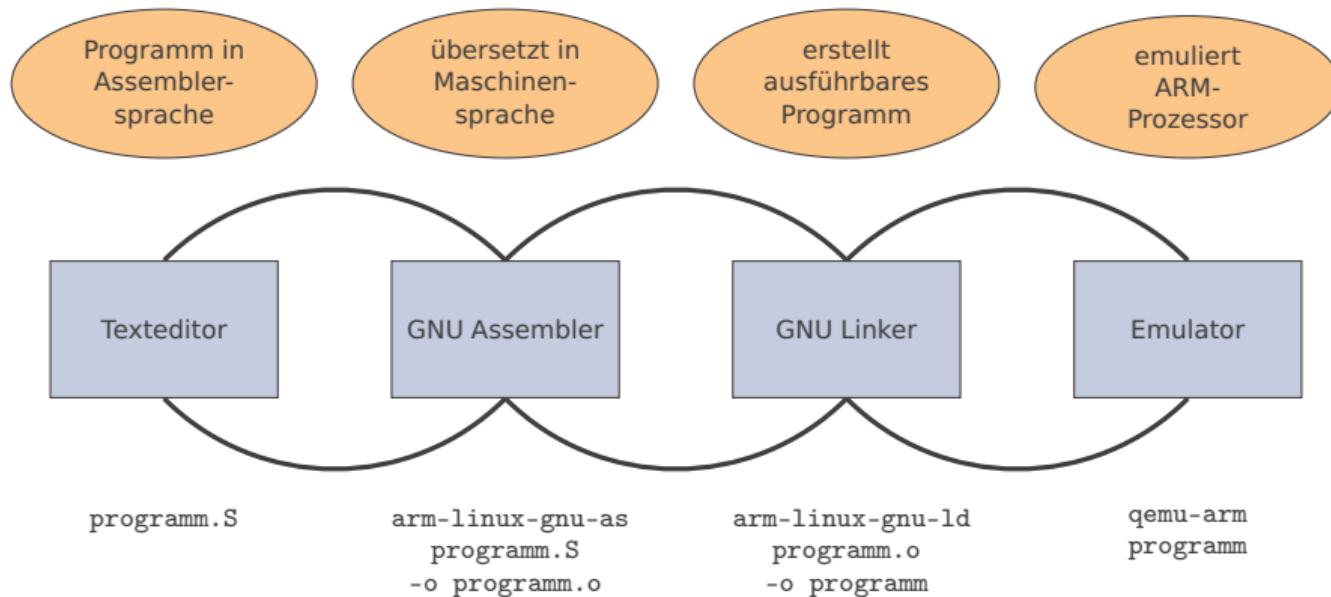
Das Kunstwort **Endianness** bezeichnet die Konvention zur **Reihenfolge** der Ablage von Bytes (8 Bit) eines **Wortes** ($n = k \times 8$ Bit) im Speicher:

- **Little-Endian:** niedwertigstes Byte zuerst, d. h. Wertigkeit nimmt mit zunehmender Adresse zu (z. B. MOS 6502, Intel x86)
- **Big-Endian:** höchstwertiges Byte zuerst, d. h. Wertigkeit nimmt mit zunehmender Adresse ab (z. B. PowerPC, Internet)

→ ARM unterstützt Big- und Little-Endian. Wir verwenden Little-Endian.

ARM-Entwicklungsumgebung

im Rechnerraum des Proseminars und auf dem ZID-GPL-Server



Dokumentation

Online verfügbar zum Nachschlagen und Selbststudium:

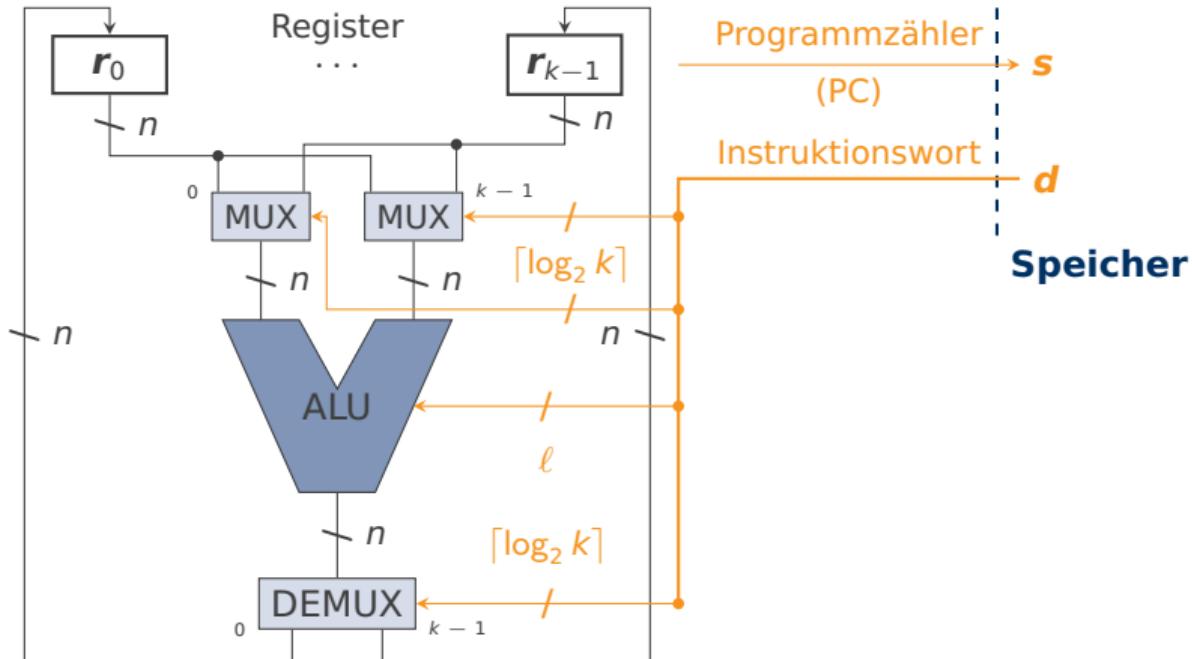
- GNU ARM Compiler Toolchain: Assembler Reference,
Version 5.03, ARM Ltd. 2013
- GNU ARM Assembler Quick Reference
<http://www.ic.unicamp.br/~celio/mc404-2014/docs/gnu-arm-directives.pdf>
- ARM and Thumb-2 Instruction Set: Quick Reference Card
<https://www.lri.fr/~de/ARM.pdf>
- Procedure Call Standard for the ARM Architecture
<https://developer.arm.com/documentation/ihi0042/e/>
(Aufrufkonventionen → nächste Woche)
- Pete Cockerell: ARM Assembly Language Programming
<http://www.peter-cockerell.net/aalp/html/frames.html>

(alle Links zuletzt abgerufen am 22. November 2021)

Gliederung heute

1. Von der sequenziellen Logik zum Mikroprozessor
2. ARM-Mikroarchitektur
- 3. ARM-Befehlssatz (ohne Speicherzugriff)**
4. Unser erstes Assemblerprogramm

Schaltskizze eines Mikroprozessors



Darstellung ohne Statusregister bzw. Flags, kein Speicherzugriff für Daten

Allgemeines Instruktionsformat

In menschenlesbarem **Assembler-Quelltext**

```
label:           ; Kommentar (mit // bei GNU)
    ADD [ggf. Bedingung] r0, r1, r2 [ggf. Optionen]
```

besteht eine Instruktion aus:

- **Mnemonic** (hier: ADD) für gewählte Instruktion
- **Zielregister** (hier: r0), Symbol **y**
- **Operanden** (hier: r1 und r2), Symbole **a** und **b**

Der ARM-Assembler übersetzt jede Zeile in **ein**

32-Bit-Instruktionswort.

Vom Programmierer wählbare **Labels** bezeichnen die Adresse des nachfolgenden Instruktionsworts und werden bei der Assemblierung aufgelöst (vgl. Binärkodierung beim Zustandsautomat).

Arithmetische Operationen

Mnemonic	Formel	Kommentar
ADD	$y = a + b$	Addition
ADC	$y = a + b + c$	Addition mit Übertrag
SUB	$y = a - b$	Subtraktion
SBC	$y = a - b + c - 1$	Subtraktion mit Übertrag
RSB	$y = b - a$	<i>reverse subtract</i>
RSC	$y = b - a + c - 1$	<i>reverse subtract</i> mit Übertrag
MUL	$y = a \cdot b$	Multiplikation
MLA	$y = (a \cdot b) + x$	<i>multiply accumulate</i>

Bemerkungen zur Multiplikation

- y erhält nur die niederwertigsten 32 Ergebnisbits.
- y und a können nicht das selbe Register sein. (Außerdem ist $r15$ nicht erlaubt.)
- Verwendet intern den Algorithmus von Booth mit Vorzeichen (bis 17 Taktzyklen)

Logische Operationen und Vergleiche

Mnemonic	Formel	Kommentar
AND	$y = a \wedge b$	bitweise AND-Verknüpfung
ORR	$y = a \vee b$	bitweise OR-Verknüpfung
EOR	$y = a \oplus b$	bitweise XOR-Verknüpfung
BIC	$y = a \wedge \bar{b}$	bitweise AND-NOT (<i>bit clear</i>)

Vergleichsoperation

verwerfen Ergebnis der ALU, aktualisieren Flags

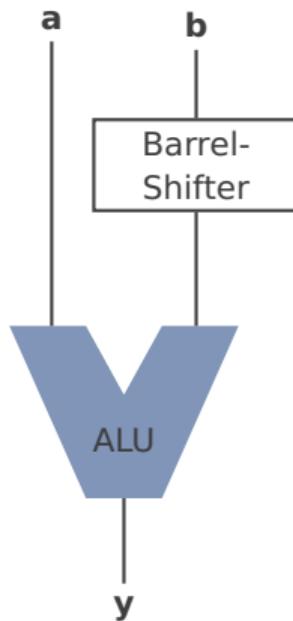
CMP	$a - b$	Vergleich
CMN	$a + b$	Vergleich mit Negation
TST	$a \wedge b$	Test
TEQ	$a \oplus b$	<i>test equivalence</i>

Registerinhalte kopieren

Mnemonic	Formel	Kommentar
MOV	$y = b$	Registerinhalt kopieren
MVN	$y = \bar{b}$	bitweise invertierte Kopie

→ MOV und MVN nutzen den ersten Operanden nicht.

Ansteuerung der ALU



***b* aus Register**

32 Bit

5-Bit-Zahl
(vorzeichenlos)
oder niedrigstes Byte
eines Registers

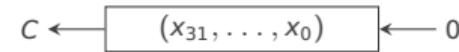
***b* aus Konstante**

8 Bit

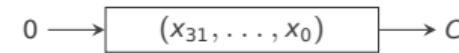
rotiert um 4-Bit Stellen:
 $\{0, 2, \dots, 30\}$
berechnet vom
Assembler

Barrel-Shifter

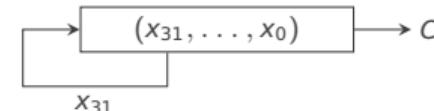
LSL – logische Linksverschiebung



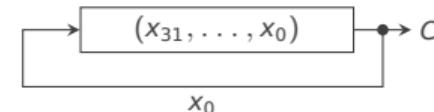
LSR – logische Rechtsverschiebung



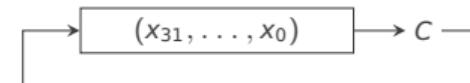
ASR – arithmetische Rechtsverschiebung



ROR – Rechtsrotation



RRX – erweiterte Rechtsrotation
(um genau 1 Bit)



ASL – arithmetische Linksverschiebung: Synonym für LSL

Effiziente Multiplikation mit Konstanten

Mit dem Barrel-Shifter können Multiplikationen mit $2^k \pm 1$ **in einem Taktzyklus** (statt 17 bei MUL) berechnet werden.

Beispiele

```
MOV r2, r0, LSL #2      ; r2 = r0 * 4
ADD r9, r5, r5, LSL #3  ; r9 = r5 * 9
RSB r9, r5, r5, LSL #3  ; r9 = r5 * 7
SUB r10, r9, r8, LSR #4 ; r10 = r9 - r8 : 16
MOV r12, r4, ROR r3     ; r12 = r4 um r3 Bits
                        nach rechts rotiert
```

Immediate-Werte

(engl. für „unmittelbar“; auch: direkte Werte, Programmkonstanten)

Assembler-Notation mit vorangestellter Raute #:
MOV r0, #13

Besonderheit bei ARM

Jedes Instruktionswort ist 32 Bit lang. Damit stehen nur 12 Bit für den zweiten Operanden **b** zur Verfügung.

- 8 Bit davon werden für Konstanten verwendet
- 4 Bit für ROR-Verschiebung in Vielfachen von 2: {0, 2, 4, ..., 30}

Wenn möglich, kümmert sich der Assembler um die Kodierung:

Beispiele

```
MOV r0, #4096  
MOV r1, #0xffffffff0
```

entsprechen

```
MOV r0, #0x40, ROR #26  
MVN r1, #15
```

Hörsaalfragen



24 82 94 16

Welche dieser Konstanten können über MOV oder MVN geladen werden?

- a. #508
- b. #510
- c. #1023
- d. #1024

Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Empfohlene Vorgehensweise

Verwendung der LDR-Ladelogik (ARM-spezifisch)

Bei Nutzung des LDR-Mnemonics sucht der Assembler den besten Weg zum Laden einer Konstante:

```
LDR r0, =0x42
```

; assembliert zu MOV r0, #0x42

```
LDR r0, =0xffffffff
```

; assembliert zu MVN r0, #0x00

```
LDR r0, =0x55555555
```

; assembliert zu LDR r0, [pc, Offset zu Konstantenpool]

:

; DCD 0x55555555 (Assembler-spezifische Pseudo-Instruktion)

LDR vertiefen wir nächste Woche beim Thema Speicherzugriff.

Einfache Sprünge

Bei ARM ist der Programmzähler r15 / pc ein Register wie jedes andere.

```
ADD  pc, pc, #8
MOV  r0, r1
MOV  r2, r3
; hier geht's weiter
```

loop:

```
    MOV  r0, r1
    MOV  r1, r2
    MOV  r3, r4
    MOV  r4, r0
    SUB  pc, pc, #20
```

„Weite“ Sprünge und Rücksprünge

Steuerung des Kontrollflusses

Wer sagt, GO TO sei böse ?

Mnemonic	Kommentar
B	Sprung an relative Zieladresse (<i>branch</i>) (Assembler berechnet 26-Bit-Offset zum Label)
BL	wie B, zusätzlich absolute Rücksprungadresse in r14 (lr) speichern (<i>with link</i>) (dient zum Aufruf von Unterprogrammen)

Sprünge an absolute Adressen können durch MOV in r15 realisiert werden,
z. B. Rücksprung aus Unterprogramm: MOV r15, r14 oder MOV pc, lr.

→ Alle Instruktionsworte müssen im Speicher “aligned” sein.

Bedingte Ausführung von Instruktionen

Besonderheit des ARM-Instruktionssatzes

Alle Instruktionen haben ein 4-Bit-Feld, das Bedingungen angibt, unter denen die Instruktion ausgeführt wird.

- Viele Architekturen erlauben dies nur für Sprünge (engl. *branches*).
- Bei ARM kommt diese Logik für jede Instruktion zum Einsatz.
- Nicht ausgeführte Instruktionen benötigen einen Taktzyklus.
- Deutliche Ersparnis gegenüber Verzweigungen, welche die Pipeline blockieren (3 Taktzyklen zum Füllen)
- **Assembler-Konvention:** Bedingung wird als Suffix an das Mnemonic angehängt

Einschränkung: Gilt nicht im Thumb-Modus (nicht Stoff dieser Lehrveranstaltung)

Bedingungen I

(engl. *conditions*)

Kodierung	Suffixe	Flags	Bedeutung
0000	EQ	Z	gleich (<i>equal</i>)
0001	NE	\bar{Z}	ungleich (<i>not equal</i>)
0010	HS CS	C	vorzeichenlos \geq (<i>higher or same</i>)
0011	LO CC	\bar{C}	vorzeichenlos $<$ (<i>lower</i>)
0100	MI	N	negativ (<i>minus</i>)
0101	PL	\bar{N}	positiv (<i>plus</i>)
0110	VS	V	Überlauf (<i>overflow set</i>)
0111	VC	\bar{V}	kein Überlauf (<i>overflow clear</i>)

Bedingungen II

(engl. *conditions*)

Kodierung	Suffix	Flags	Bedeutung
1000	HI	$C \cdot \bar{Z}$	vorzeichenlos $>$ (<i>higher</i>)
1001	LS	$\bar{C} + Z$	vorzeichenlos \leq (<i>lower or same</i>)
1010	GE	$NV + \bar{N}\bar{V}$	\geq mit Vorzeichen (<i>greater or equal</i>)
1011	LT	$\bar{N}\bar{V} + \bar{N}V$	$<$ mit Vorzeichen (<i>less than</i>)
1100	GT	$\bar{Z}NV + \bar{Z}\bar{N}\bar{V}$	$>$ mit Vorzeichen (<i>greater than</i>)
1101	LE	$N\bar{V} + Z + \bar{N}V$	\leq mit Vorzeichen (<i>less or equal</i>)
1110	AL	1	ohne Bedingung (<i>always</i>)
1111	NV	0	reserviert (<i>never</i>)

Anwendung bedingter Instruktionen

Konventionell

```
CMP r3, #7  
BEQ skip  
ADD r0, r1, r2
```

```
skip: ...
```

ARM-typisch

```
CMP r3, #7  
ADDNE r0, r1, r2  
...
```

Konsequent: Für jede Instruktion wird festgelegt, ob sie Flags setzt (Suffix: **S**).
Bedingungen bleiben bei Bedarf über mehrere Instruktionen erhalten.

Schleife

```
loop: ...  
SUBS r1, r1, #1  
BNE loop
```

Ausnahme: CMP braucht kein S.

Systemaufrufe

„Vorteil von Assembler: Man kann alles machen.“

„Nachteil von Assembler: Man muss alles machen.“

In vielen Fällen stellt das **Betriebssystem** grundlegende Funktionen bereit.

Die Schnittstelle ist abhängig von Architektur und Betriebssystem.

- **ARM** nutzt die Instruktion SWI (*software interrupt*) zum Aufruf von Funktionen im privilegierten Modus (SVC).
- **Linux** definiert, welche Funktion abhängig von den Werten in den Registern r0, ..., r7 ausgeführt wird.

Beispiel: r0=0, r7=1 zum geordneten Beenden des Programms.

- Diese Schnittstelle steht auch im ARM-Emulator zur Verfügung.

Kodierung der Instruktionswörter

Jeder ARM-Assemblerbefehl wird nach diesem Schema in genau ein 32-Bit-Instruktionswort kodiert:

Befehlstyp										
Bedingung	0 0 1	Opcode	S	Rn	Rd	2. Operand				Data Processing
Bedingung	0 0 0 0 0 0 A S		Rd	Rn	Rs	1 0 0 1	Rm			Multiply
Bedingung	0 1 I P U B w L		Rn	Rd	Offset					Single Data Transfer
Bedingung	1 0 0 P U B w L		Rn	Registerliste						Block Data Transfer
Bedingung	0 0 0 P U 1 w L		Rn	Rd	Offset 1	1 S H 1	Offset 2			Halfword Trans Imm
Bedingung	0 0 0 P U 0 w L		Rn	Rd	0 0 0 0 1 S H 1	Rm				Halfword Trans Reg
Bedingung	1 0 1 L	relative Zieladresse								Branch
Bedingung	0 0 0 1 0 0 1 0 1 1 1 1 1 1 1 1 0 0 0 1 S H 1			Rn						Branch Exchange
Bedingung	1 1 1 1	SWI-Nummer (vom Prozessor ignoriert)								Software Interrupt

Einige ARM-Prozessoren unterstützen zusätzlich eine kompaktere Kodierung, die 16- und 32-Bit-Worte mischt. Dieser **Thumb**- und **Thumb-2**-Kode ist meist kürzer, aber langsamer (und nicht Stoff dieser Lehrveranstaltung).

Gliederung heute

1. Von der sequenziellen Logik zum Mikroprozessor
2. ARM-Mikroarchitektur
3. ARM-Befehlssatz (ohne Speicherzugriff)
4. **Unser erstes Assemblerprogramm**

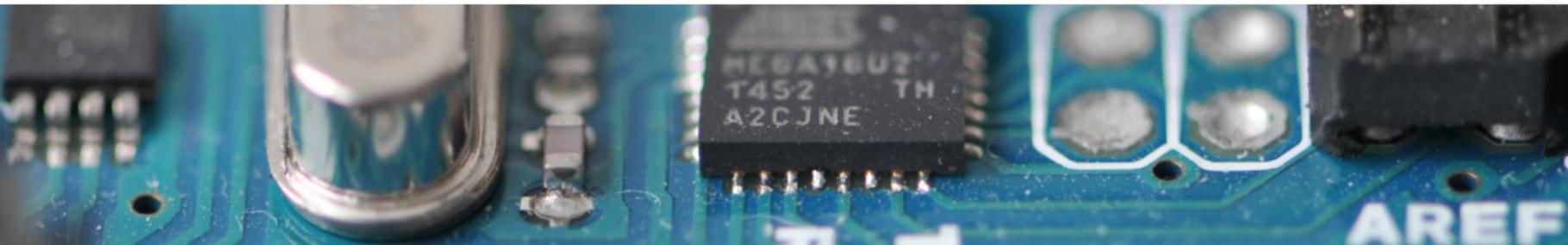
Unser erstes Assemblerprogramm

Hello Innsbruck!

```
.data
msg:
.ascii    "Hello Innsbruck!\n"
          len = . - msg
.text
.align
.global _start
_start:
/* write syscall */
    MOV      r0, #1
    LDR      r1, =msg
    LDR      r2, =len
    MOV      r7, #4
    SWI      #0
/* exit syscall */
    MOV      r0, #0
    MOV      r7, #1
    SWI      #0
          0000 48 65 6C 6C 6F 20 49 6E
          0008 6E 73 62 72 75 63 6B 21
          0010 0A
          0014 E3A00001
          0018 E59F1016
          001c E3A02012
          0020 E3A07004
          0024 EF000000
          0028 E3A00000
          002c E3A07001
          0030 EF000000
```

Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Befehlssatzarchitektur II

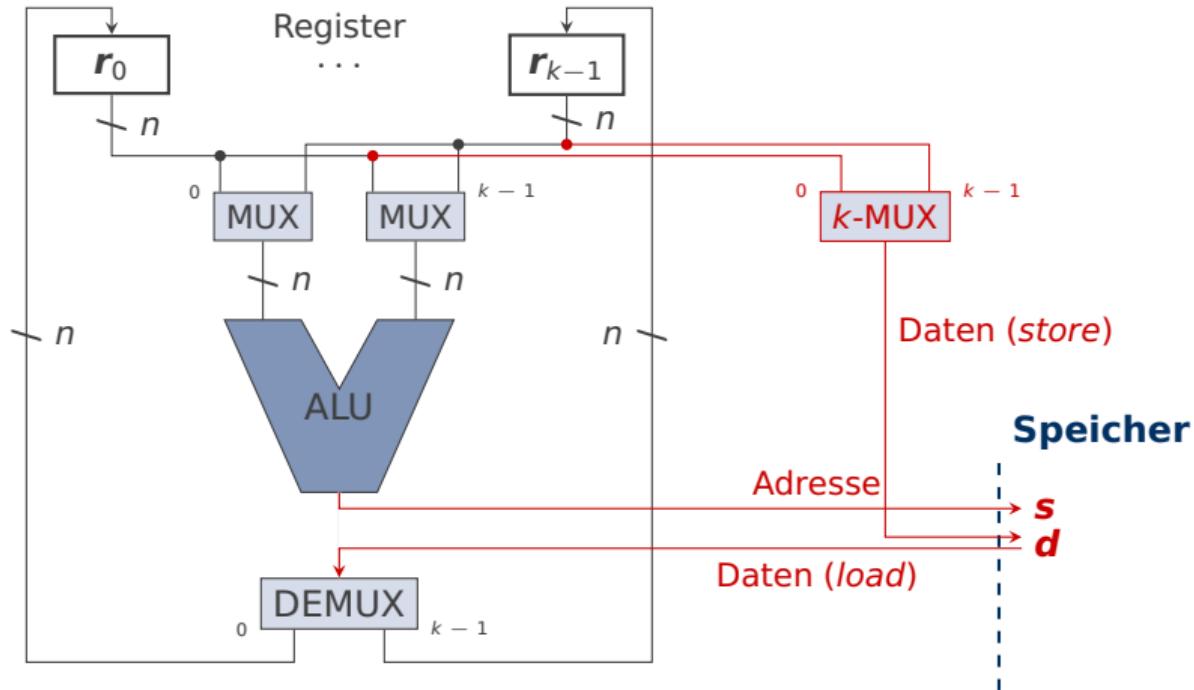
Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2021/22 · 1. Dezember 2021

Gliederung heute

- 1. Speicherzugriff**
2. Division und Zahlenausgabe in Assembler
3. Stapelorganisation und Funktionsaufrufe

Schaltskizze eines Mikroprozessors



Darstellung ohne Statusregister bzw. Flags, Load-Store-Architektur ohne Instruktionsdekodierung

Speicherzugriff

Mnemonics		Kommentar	
LDR	STR	SWP	Lese/schreibe/tausche 32-Bit-Wort
LDRB	STRB	SWPB	Lese/schreibe/tausche Byte
LDRH	STRH		Lese/schreibe Halbwort (16 Bit)
LDRSB			Lese Byte mit Vorzeichenerweiterung
LDRSH			Lese Halbwort mit Vorzeichenerweiterung

Die Adresse wird über ein **Basisregister** plus **Offset** angegeben:

STR r0, [r1] ; Inhalt von r0 an Adresse speichern,
; die in r1 steht.

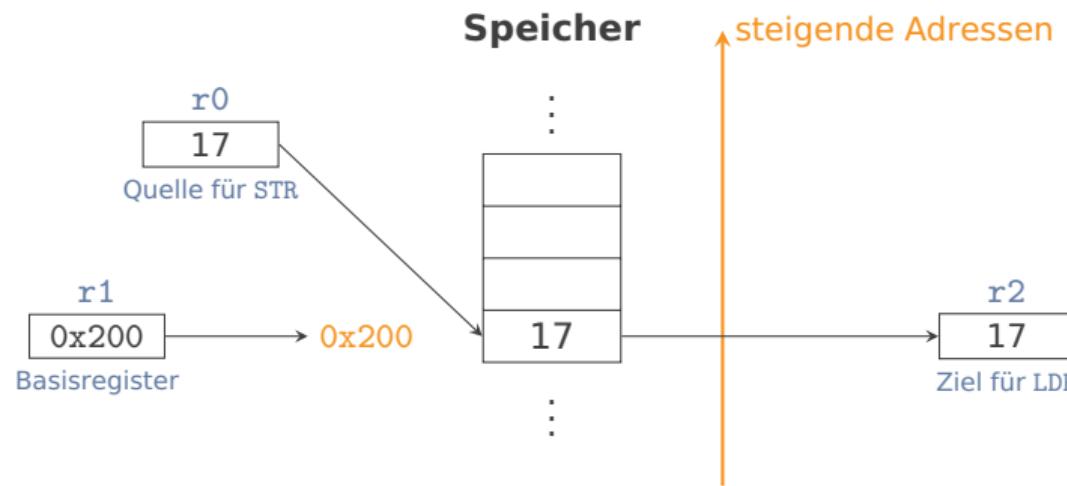
LDR r2, [r1,#-12] ; Speicherinhalt an der Adresse (r1-12)
; nach r2 laden.

Bedingungen sind möglich und werden zwischen Stamm-Mnemonic und Größensuffix eingeschoben, z. B. LDREQB.

Adressierungsarten

Angabe der Speicheradresse über **Basisregister**

STR r0, [r1] ; Inhalt von r0 an Adresse speichern, die in r1 steht.
LDR r2, [r1] ; Speicherinhalt an der Adresse r1 nach r2 laden.

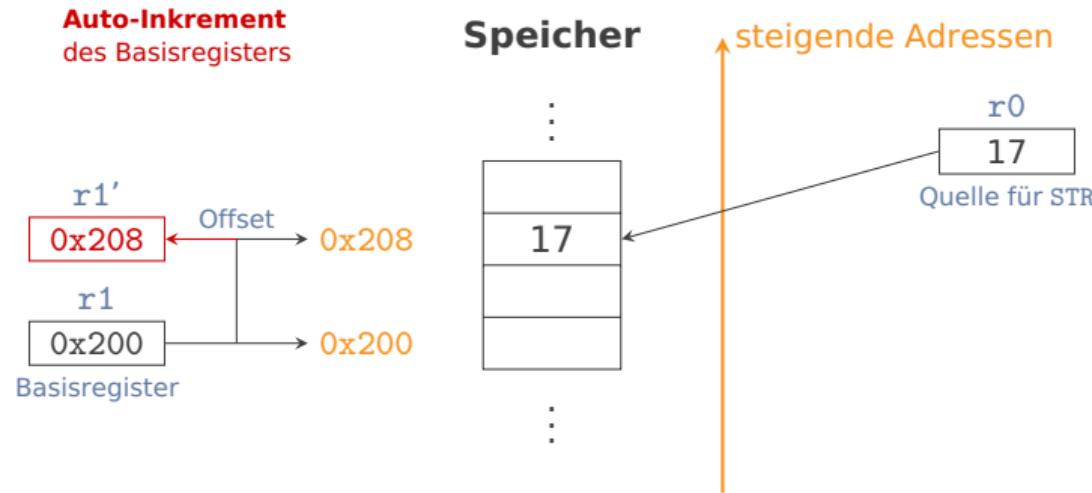


ARM unterstützt ausschließlich **indirekte** Adressierung.

Adressierungsarten (Forts.)

Angabe der Speicheradresse über **Basisregister** und **Offset**

STR r0, [r1, #8] ! ; Immediate (12 Bit plus Vorzeichen)

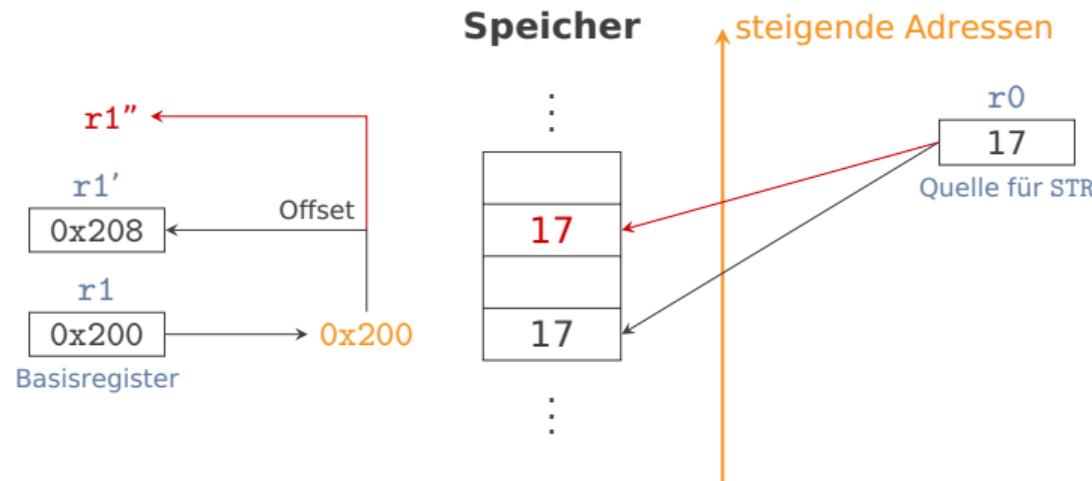


Adressierungsarten (Forts.)

Angabe der Speicheradresse über **Basisregister** und **Offset**

STR r0, [r1], #8 ; "Post-indexed"-Adressierung

STR r0, [r1], r2, LSL #3 ; mit Register (äquivalent falls r2 = 1)



Beispiele

Zugriff auf das k -te Element eines **Arrays**, das aus 16 Byte langen Datenstrukturen besteht

```
; erwarte k in r2  
LDR  r1, =beispiel+4  
; r1 zeigt auf feld[0].ziel  
LDR  r0, [r1, r2, LSL #4]  
; Lesezugriff, r1 unverändert
```

Beispiel-Struktur in C

```
1 struct beispiel_t {  
2     unsigned int quelle;  
3     unsigned int ziel;  
4     int         anzahl;  
5     int         pad;  
6 } feld[1024];
```

Beispiele (Forts.)

Kopieren von Speicherbereichen

; ggf. Rücksprungadresse in lr vorher sichern
; besser: Variante mit niedrigeren Registern schreiben

```
LDR    r12, =quelle ; erste zu kopierende Adresse
LDR    r13, =ziel   ; erste Zieladresse
LDR    r14, =len    ; Länge in Wörtern (> 0, sonst fatal)
```

copyloop:

```
LDR    r0, [r12], #4 ; Auto-Inkrement, post-indexed
STR    r0, [r13], #4 ; Auto-Inkrement, post-indexed
SUBS  r14, r14, #1
BNE   copyloop
```

; Sonderbehandlung nötig, wenn Daten nicht "aligned"

Geht es noch effizienter?

Block Data Transfer

Mnemonic	Kommentar
LDM	lese 1–16 Register (<i>load multiple</i>)
STM	schreibe 1–16 Register (<i>store multiple</i>)

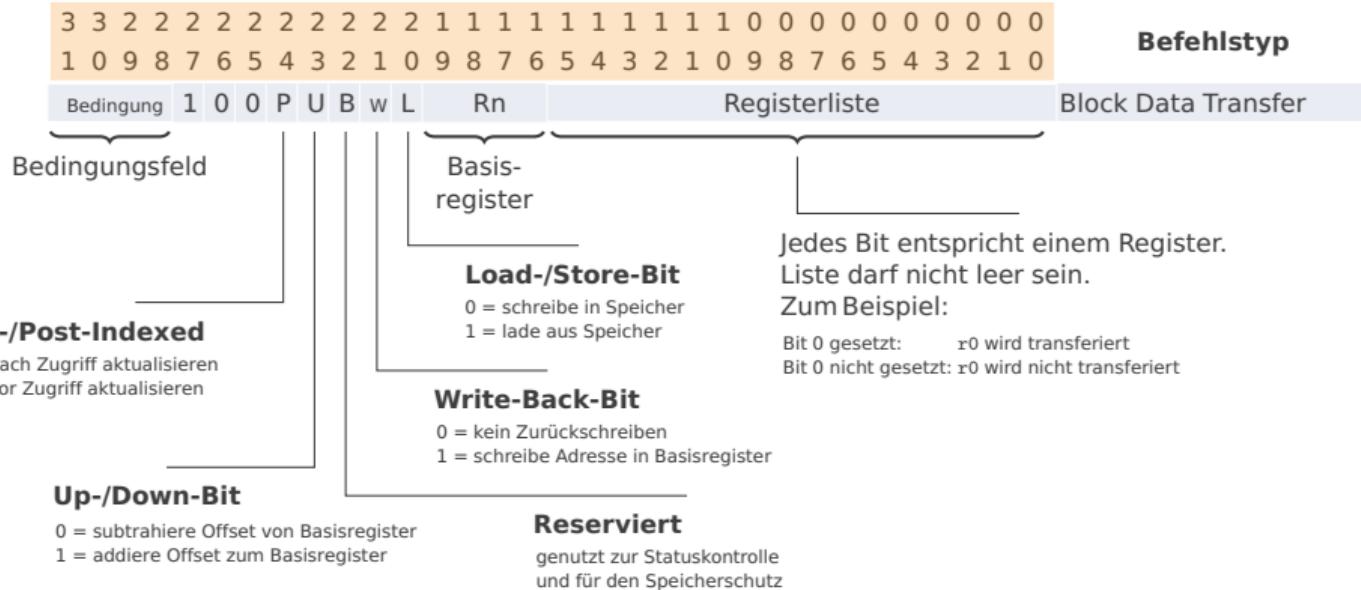
Adressierung erfolgt über Basisregister, jedoch ohne Offset:

```
STM    r0, {r1-r5}      ; r1 bis r5 an die Adressen  
                  ; [r0], ..., [r0 + 19] schreiben  
LDMIA  r0!, {r3,r6}    ; Register auch einzeln wählbar
```

- Die Reihenfolge ist festgelegt: Speicheradressen steigen mit Registernummer auf.
- Aktualisierung des Basisregisters (**Auto-Inkrement**) möglich
- Nützlich zum temporären Sichern der Registerinhalte

Beispiel für Kodierung im Instruktionswort

Die Dekodierung erfolgt in der Fetch-Stufe der Prozessor-Pipeline.



Gliederung heute

1. Speicherzugriff
2. **Division und Zahlenausgabe in Assembler**
3. Stapelorganisation und Funktionsaufrufe

Divisionsalgorithmus mit „Restoring“ (W)

Verwendung von bedingter Addition, **Subtraktion** und Schiebeoperationen

Pseudocode

Require: Dividend a , Divisor b (jeweils n Bit)

$(y_{n-1}, \dots, y_0) \leftarrow a$ {Initialisiere (“load”) 2n-Bit-Register y .}

$(y_{2n-1}, \dots, y_n) \leftarrow 0$

for $i = 0$ to $n - 1$ **do**

$(y_{2n-1}, \dots, y_0) \leftarrow (y_{2n-2}, \dots, y_0, 0)$ {Schiebe y um ein Bit nach links.}

$(y_{2n-1}, \dots, y_n) \leftarrow (y_{2n-1}, \dots, y_n) - b$

if $y_{2n-1} = 0$ **then**

$y_0 \leftarrow 1$

else

$(y_{2n-1}, \dots, y_n) \leftarrow (y_{2n-1}, \dots, y_n) + b$ {Wiederherstellung des Rests}

end if

end for

$r \leftarrow (y_{2n-1}, \dots, y_n)$

$q \leftarrow (y_{n-1}, \dots, y_0)$

{Ergebnis. Es gilt: $a = b \times q + r$ }

Realisierung in Assembler

```
; Dividend in r1 (16 Bit, vorzeichenlos)
; Divisor in r2 (16 Bit, vorzeichenlos)

div:
    MOV    r2, r2, LSL #16
    MOV    r3, #16          ; Schleifenzähler

divloop:

    RSBS   r1, r2, r1, LSL #1 ; schiebe und subtrahiere
    ORRPL  r1, r1, #1
    ADDMI  r1, r1, r2        ; Wiederherstellung des Rests
    SUBS   r3, r3, #1
    BNE    divloop

    ; Quotient in r115, ..., r10
    ; Rest in r131, ..., r116

    MOV    pc, lr            ; Rücksprung
```

Ausgabe von Hexadezimalzahlen

Nutzung des Systemaufrufs zur Ausgabe von ASCII-Zeichenketten:

```
; Ganzzahl in r4 (32 Bit, vorzeichenlos)

hex:
    MOV    r3, #8           ; 8 Hexadezimalstellen
    MOV    r7, #4           ; wähle Systemaufruf write
    MOV    r2, #1           ; Länge der Zeichenkette

hexloop:
    LDR    r1, =lut         ; Adresse der Zeichentabelle
    ADD    r1, r1, r4, LSR #28 ; addiere Bits 28–31 von r4
    SWI    #0
    MOV    r4, r4, LSL #4   ; nächste Hex-Ziffer in Bits 28–31
    SUBS   r3, r3, #1
    BNE    hexloop
    MOV    pc, lr           ; Rücksprung

lut:    .ascii "0123456789abcdef" ; Look-Up-Tabelle
```

Programmrumpf zum Test von div und hex

```
.arm          ; assembliere im Standard-ARM-Modus
.text         ; Start eines nicht beschreibbaren Programmreichs
.global _start           ; Linker soll Symbol _start kennen
_start:          ; Konvention für Einsprungpunkt
    LDR    r1, =169      ; Dividend
    LDR    r2, =12       ; Divisor
    BL     div          ; Division: Quotient und Rest in r1
    MOV    r4, r1
    BL     hex          ; Ausgabe
    MOV    r0, #0
    MOV    r7, #1       ; wähle Systemaufruf exit
    SWI    #0
div:   ...        ; von Folie 16
hex:  ...        ; von Folie 17
```

Hörsaalfrage



24 82 94 16

Welche Ausgabe erzeugt das Assemblerprogramm ?

- a. 14
- b. 0x000e
- c. e0001000
- d. 0001000e

Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Ausgabe von Dezimalzahlen

```
dec:          ; Ganzzahl in r1 (16 Bit, vorzeichenlos)
    MOV  r8, lr           ; Rücksprungadresse sichern
    LDR  r5, =buffer+5    ; Zeiger auf Ende des Puffers +1
    MOV  r6, #0x30         ; ASCII-Kode für 0 als Offset
    MOV  r7, #0             ; Stellenzähler

decloop:      ADD  r7, r7, #1        ; nächste Ziffer (mind. eine)
    MOV  r2, #10            ; Basis 10 (dezimal)
    BL   div                ; r1 : r2 von Folie 16
    ADD  r4, r6, r1, LSR #16 ; Rest als Ziffer in ASCII ...
    STRB r4, [r5,-r7]       ; ... rückwärts in Puffer schreiben
    BICS r1, r1, #0x000f0000 ; Rest löschen
    BNE   decloop          ; mehr Stellen wenn Quotient > 0
    SUB  r1, r5, r7          ; Start der Zeichenkette im Puffer
    MOV  r2, r7              ; Länge der Zeichenkette
    MOV  r7, #4              ; Systemaufruf write wählen
    SWI  #0
    MOV  pc, r8              ; Rücksprung

.data          ; für Linker: Start eines beschreibbaren Speicherbereichs
buffer:       .space 5           ; 5 Byte, denn  $\lceil \log_{10}(2^{16}) \rceil = 5$ 
```

Gliederung heute

1. Speicherzugriff
2. Division und Zahlenausgabe in Assembler
3. **Stapelorganisation und Funktionsaufrufe**

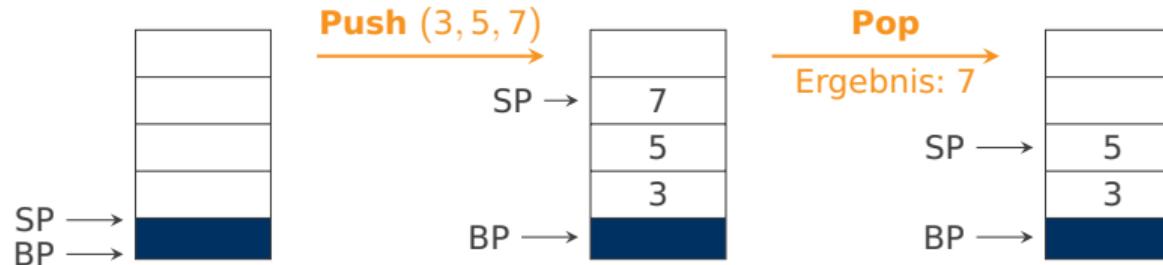
Stapel

Ein **Stapel** (engl. *stack*) ist eine Datenstruktur, die

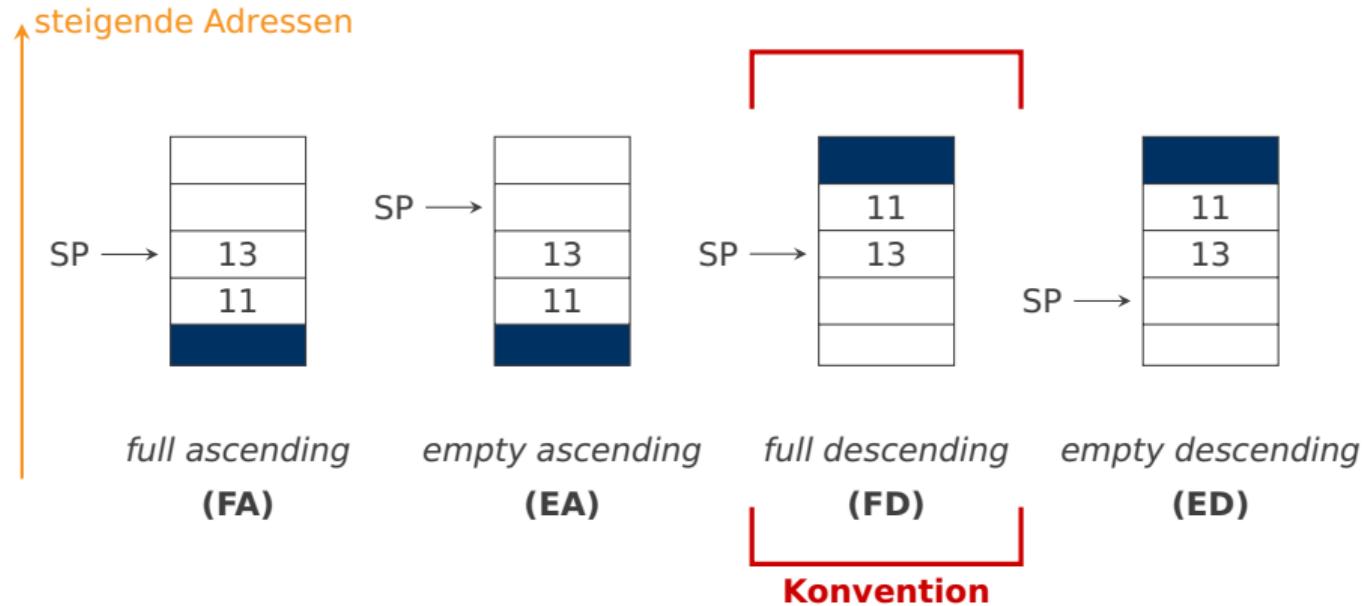
- **wächst**, wenn man neue Daten „darauf“ abgelegt ($\rightarrow push$) und
- **schrumpft**, wenn man Daten „von oben“ wegnimmt ($\rightarrow pop$).

Bei der Realisierung im **Speicher** definieren zwei **Zeiger** (engl. *pointer*) die aktuellen Grenzen des Stapels:

- **Base Pointer** (BP) zeigt auf den „Boden“.
- **Stack Pointer** (SP) zeigt auf die „Spitze“.



Varianten der Stapelorganisation



Realisierung mit dem Block Data Transfer

STM/LDM-Mnemonics können direkt um die Suffixe FA, EA, FD und ED ergänzt werden, um das gewünschte Verhalten zu erreichen.

Nützlich für verschachtelte und rekursive **Unterprogramme**:

proc:

```
STMFD sp!, {r0-r12, lr} ; alle Register  
; einschl. Rücksprungadresse  
: ; auf den Stapel legen  
LDMFD sp!, {r0-r12, pc} ; wiederherstellen  
; und Rücksprung
```

Beispiel für **Aufruf**:

```
BL proc
```

Alternative Suffixe für STM und LDM

Suffix	Bedeutung	verwendet bei
IA	<i>increment after</i>	STMEA LDMFD
IB	<i>increment before</i>	STMFA LDMED
DA	<i>decrement after</i>	STMED LDMFA
DB	<i>decrement before</i>	STMFD LDMEA

Anwendung: Skizze einer sehr effizienten Kopierschleife (vgl. Folie 9)

blockloop:

```
LDMIA    r12!, {r0-r11} ; 48 Bytes laden  
STMIA    r13!, {r0-r11} ; speichern  
SUBS    r14, r14, #1    ; Vielfache von 48  
BNE     blockloop  
; vor Rücksprung sp und lr wiederherstellen
```

Allgemeiner Ablauf eines Funktionsaufrufs

1. **Parameter** (Argumente) werden an vereinbarter Stelle (Speicher oder Register) abgelegt
2. Übergabe der Ablaufsteuerung an das Unterprogramm
3. Bereitstellung von Speicher für **lokale Variablen**
4. Vollständige Ausführung der Unterprogramms
5. **Ergebnis** (Wert) wird an Stelle abgelegt, auf welche das aufrufende Programm zugreifen kann
6. Rückgabe der Ablaufsteuerung an das aufrufende Programm; Fortführung an Position unmittelbar nach dem Aufruf

Aufrufkonventionen definieren diese Schnittstelle.

ARM-Aufrufkonventionen

(extrem vereinfacht; Annahme: alle Werte passen in 32 Bit)

Parameter

- Die ersten vier Argumente werden in den Registern r0, ..., r3 übergeben.
- Alle weiteren kommen auf einen *full descending* Stapel.

Lokale Variablen

- Liegen auf dem Stapel.

Ergebnis

- Rückgabe im Register r0.

Das Unterprogramm erhält die Werte aller Register ab r4.

Aufruf einer Funktion in der C-Standard-Library

```
.global _start
_start:
    LDR    r0, =msg1 ; 1. Argument (Zeiger auf Zeichenkette)
    BL     printf   ; Aufruf in Bibliothek (→ Linker)

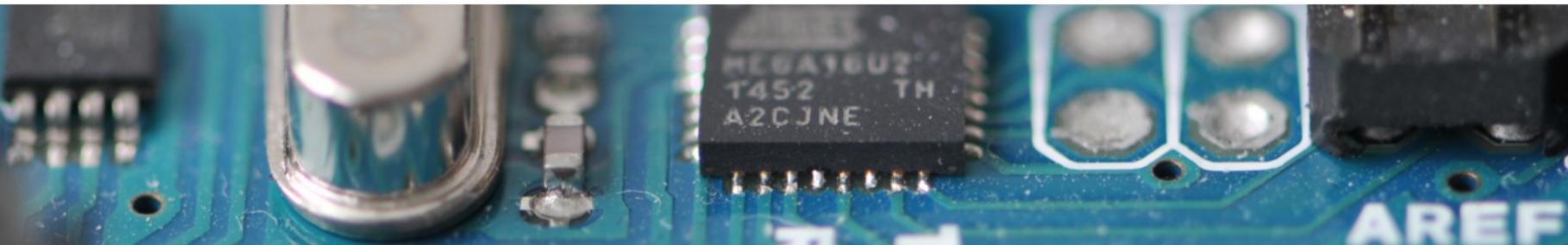
    MOV    r1, r0    ; Rückgabewert als 2. Argument
    LDR    r0, =msg2 ; Format-String als 1. Argument
    BL     printf   ; Ausgabe

    MOV    r0, #0    ; Programm beenden
    MOV    r7, #1
    SWI    #0
                                ; Null-terminierte Zeichenketten
msg1:   .asciz "I love assembler.\n"
msg2:   .asciz "Printed %i characters.\n"
```

Zum Debuggen der C-Schnittstelle: Compiler mit der Option -S aufrufen.

Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Ein-/Ausgabe

Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2021/22 · 15. Dezember 2021

Gliederung heute

- 1. Touch-Eingabetechnologien**
2. Ansteuerung von E/A-Bausteinen
3. Unterbrechungsanforderungen
4. Praxisbeispiel zur Ausgabe

Technologien zur Berührungsmeßung

Berührung wird immer indirekt gemessen.

- **Lichtintensität**

Infrarotgitter, Kamera-basiert

- **Spannung**

Resistive Verfahren

- **Strom**

Oberflächen-kapazitive Verfahren

- **Kapazitätsänderung**

Projiziert-kapazitive Verfahren

- **Zeitverzögerung**

Akustische Oberflächenwelle

- **Kraft**

Vergleichskriterien

- Haltbarkeit
- Durchsichtigkeit
- Flexibilität der Eingabe (Stift, Finger)
- Multitouch-Fähigkeit
- Kalibrierungsstabilität
- Energiebedarf
- Bauform
- Kosten

Viele verschiedene Technologien mit spezifischen Vor- und Nachteilen

Optische Berührungsmeßung

Fotodioden

$x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9$

y_9

y_8

y_7

y_6

y_5

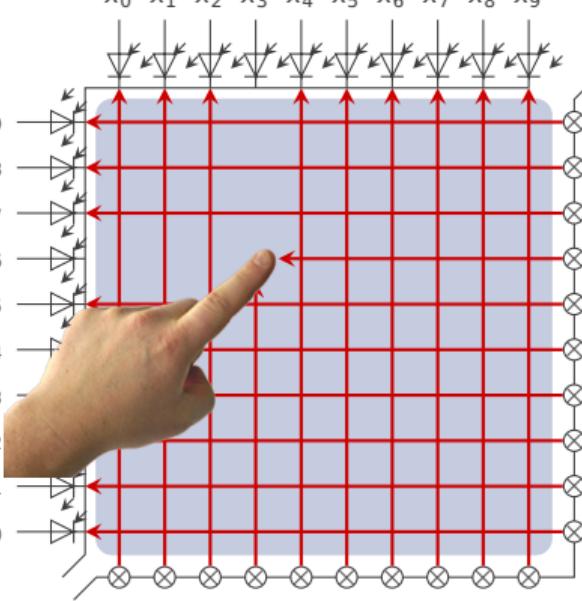
y_4

y_3

y_2

y_1

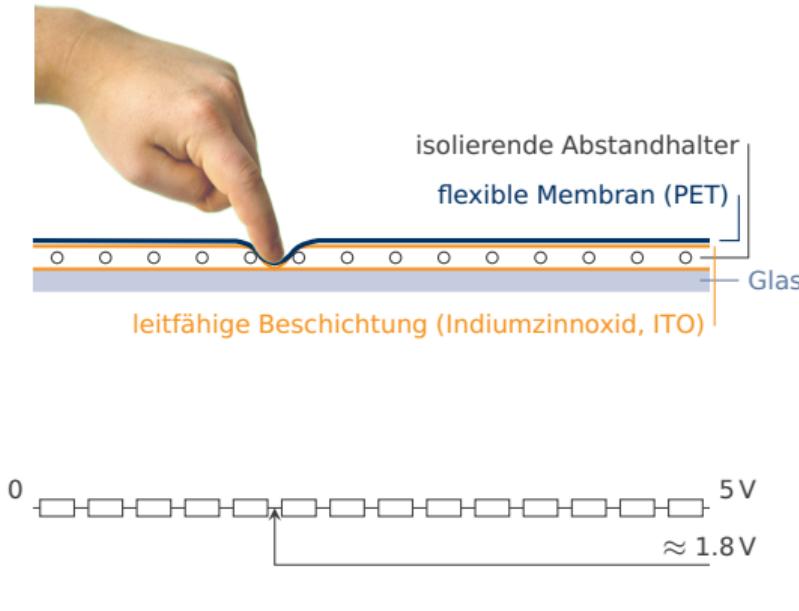
y_0



Infrarot-LEDs

Resistive Berührungsmeßung

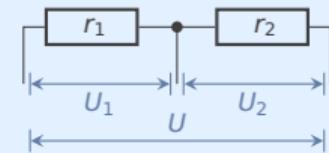
1D-Modell



Schaltung von Widerständen



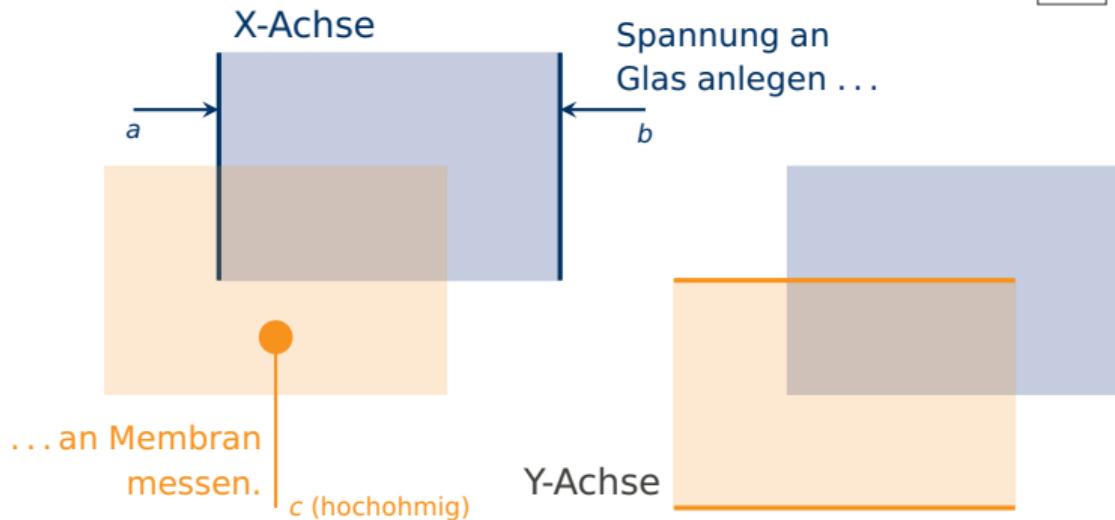
$$r = r_1 + r_2$$



$$U_i = \frac{U \cdot r_i}{r_1 + r_2}$$

Resistive Berührungsmeßung

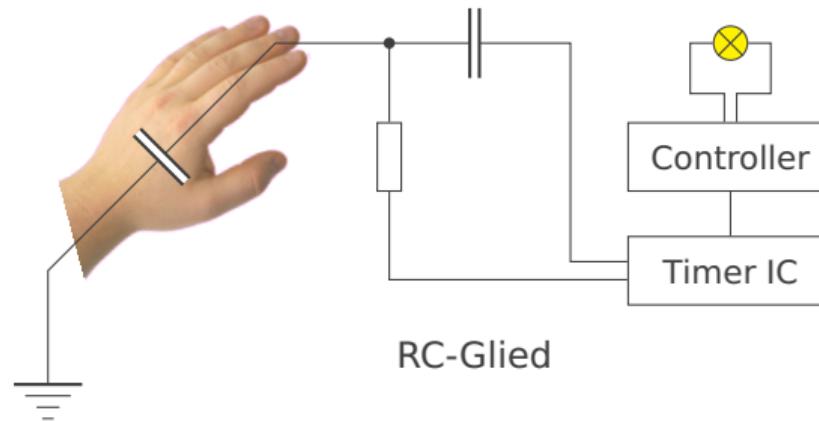
2D-Realisierung



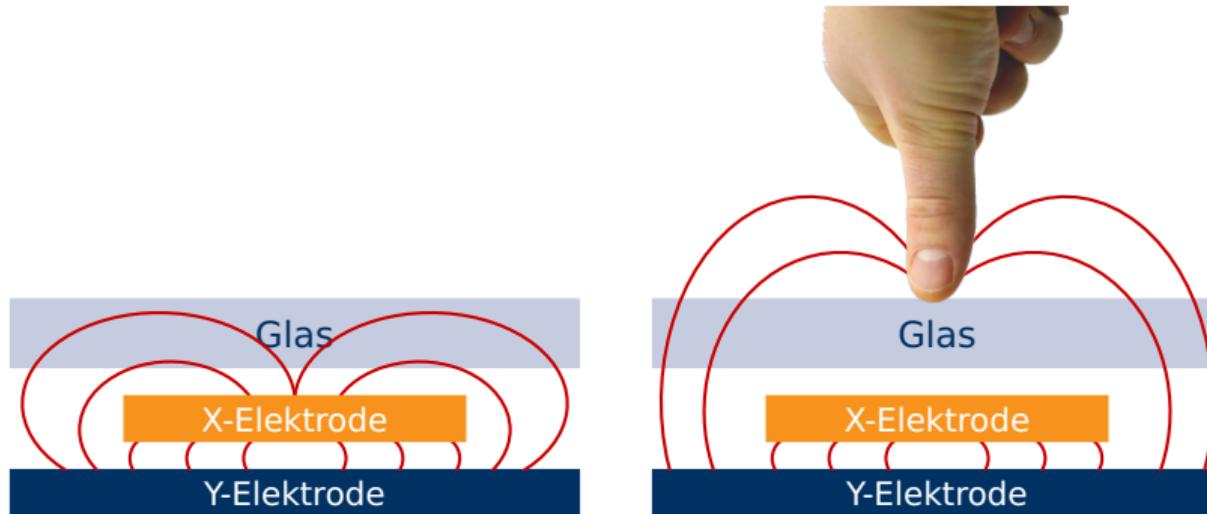
Alternierende Messung von X- und Y-Koordinate

Kapazitive Berührungsmessung

Kapazität beeinflusst Frequenz einer Wechselstromschaltung



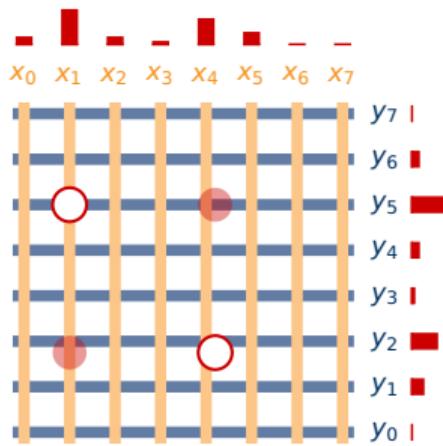
Projiziert-kapazitive Berührungsmeßung



- Finger „stehlen“ Ladung von der X-Elektrode. Dadurch ändert sich die Kapazität zwischen den Elektroden.
- Bei Berührung werden die Feldlinien über die berührungs-empfindliche Oberfläche **projiziert**.

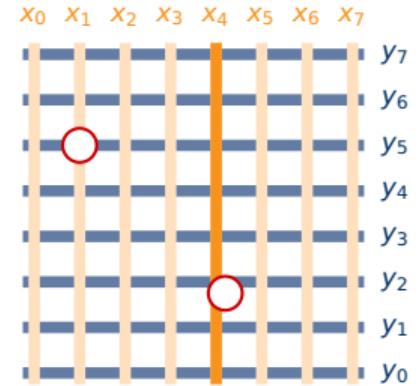
Ausleseverfahren bei „Pro-Cap“-Touchscreens

Variante 1: **Perimeter Scan**



Variante 2: **Imaging**

Steuerleitungen: Spaltenauswahl



Signalleitungen

- Messwerte $(x_0, \dots, x_7, y_0, \dots, y_7)$ sequenziell an A/D-Wandler
- „Geisterpunkte“ bei zwei Fingern

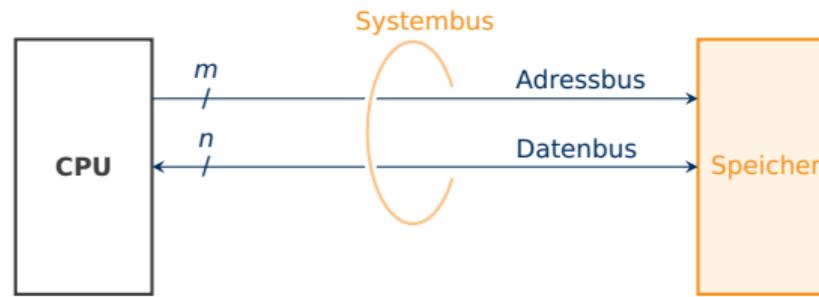
- Smartphones z. B. 9×16 , 20–200 Hz
- Interpolierte Auflösung: 1024×1024 , ca. 16 Punkte

Gliederung heute

1. Touch-Eingabetechnologien
2. **Ansteuerung von E/A-Bausteinen**
3. Unterbrechungsanforderungen
4. Praxisbeispiel zur Ausgabe

Systembus

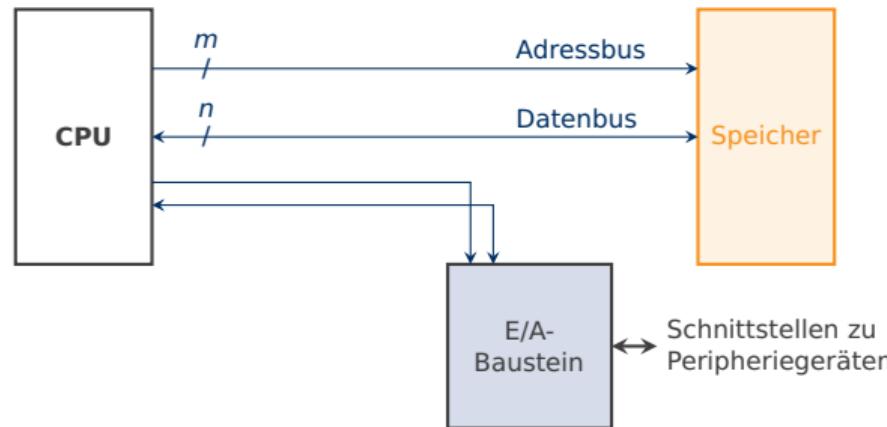
Kommunikationskanal der CPU mit anderen Systemkomponenten



Nicht dargestellt: Takt- und Steuerleitungen (w/r),
Speicherverwaltungslogik (MMU)

Ansteuerung von E/A-Bausteinen

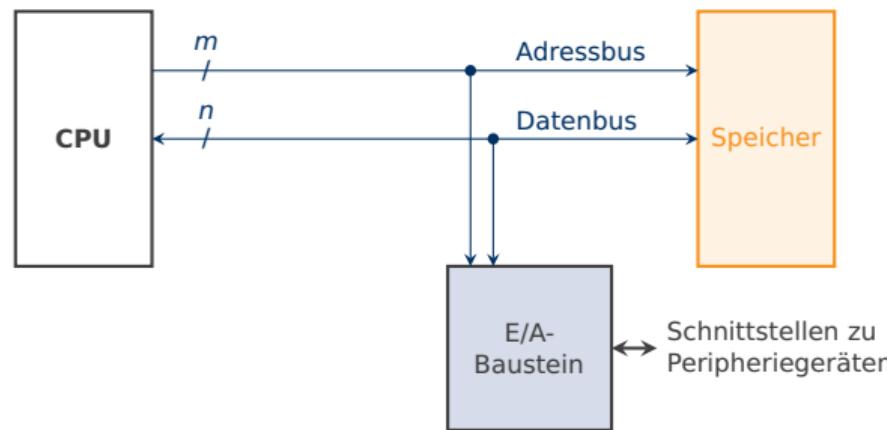
E/A-Bausteine (engl. *I/O controller*) steuern physikalische Schnittstellen.



Variante 1 **Ports** an einem separatem E/A-Bus werden mitspeziellen Instruktionen (z. B. `in`, `out` bei x86) angesprochen.

Ansteuerung von E/A-Bausteinen

E/A-Bausteine (engl. *I/O controller*) steuern physikalische Schnittstellen.



Variante 2 Einblenden der **Register** von E/A-Bausteinen in den (Daten-)Adressraum der CPU. Zugriff wie auf Speicher.

Typen von E/A-Registern

1. Kontrollregister zur Initialisierung und Funktionswahl

- Auswahl ob Leitung als Eingang oder Ausgang arbeitet
- Aktivierung von Unterbrechungsanforderungen

2. Datenregister zum Puffern von ein- oder ausgehenden Daten

- E/A-Geräte verarbeiten Daten langsamer und asynchron zur CPU.
- Oft als FIFO-Puffer (*first in, first out*) realisiert

3. Statusregister zum Austausch von Zustandsinformationen

- Verfügbarkeit neuer Eingabewerte an Datenregister
- Abschluss des Sendevorgangs aus Datenregister
- Timer

Statusregister werden bei **programmierter Ein-/Ausgabe** regelmäßig vom Hauptprogramm abgefragt (*polling*).

Viele E/A-Register sind nur lesbar oder nur schreibbar.

Beispiel: Zugriff auf System-Timer

Dieser Timer ist wie ein E/A-Baustein im BCM 2835 des Raspberry Pi Zero integriert.

```
wait:          ; warte r0 Mikrosekunden ( $r0 \cdot 10^{-6}$  Sekunden)
    STMFD sp!, {r0-r12,lr} ; Register auf Stapel sichern (Push)

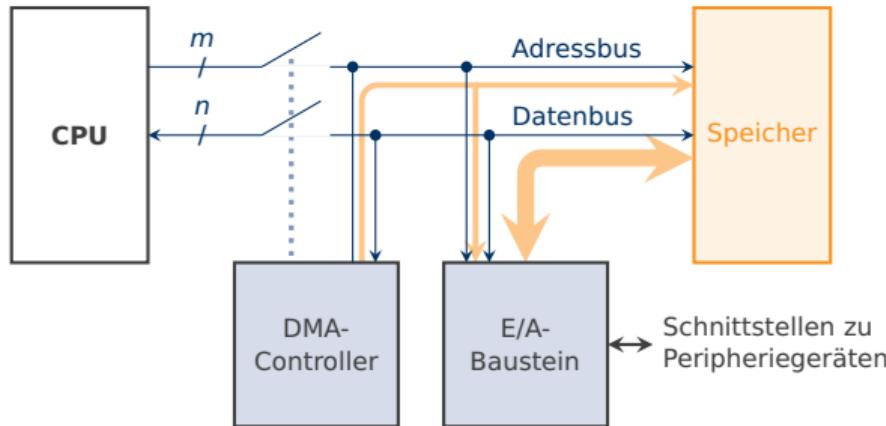
    LDR r3, =0x20003000 ; Basisadresse des System-Timers
    LDR r2, [r3, #4]      ; 32-Bit-Mikrosekundenzähler holen

sleep:
    LDR r1, [r3, #4]      ; Aktuellen Zählerstand
    SUB r1, r1, r2        ; um Startwert verringern
    CMP r1, r0            ; und mit Wartezeit vergleichen.
    BLS sleep             ; Wiederhole, wenn nicht abgelaufen
    LDMFD sp!, {r0-r12,pc} ; Register wiederherstellen (Pop)
                           ; und Rücksprung
```

Vorsicht, dieses Beispiel funktioniert nur fast immer. **Warum?**

Speicherdirektzugriff

(engl. *direct memory access*, DMA)



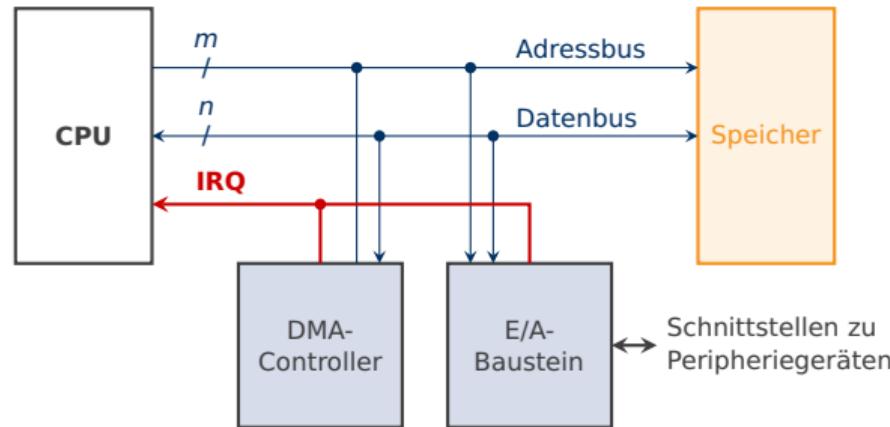
- Effizientere Übertragung großer Datenmengen
- Gleichzeitige Nutzung der CPU für „anspruchsvollere“ Aufgaben

Gliederung heute

1. Touch-Eingabetechnologien
2. Ansteuerung von E/A-Bausteinen
- 3. Unterbrechungsanforderungen**
4. Praxisbeispiel zur Ausgabe

Unterbrechungsanforderungen

(engl. *interrupt request, IRQ*)



Alternative zum Polling

- E/A-Bausteine melden Statusänderung über spezielle Leitung.
- CPU **verändert Kontrollfluss** i. d. R. beim nächsten Fetch-Zyklus.

ARM Exception Vector Table (EVT)

Adresse	Ausnahme (exception)	Kommentar
0xffff0000	Reset	(und Systemstart)
0xffff0004	Undefined instruction	nützlich für Softwareemulation
0xffff0008	Software interrupt	SWI-Instruktion
0xffff000c	Prefetch abort	pc enthält unzulässige Adresse
0xffff0010	Data abort	unzulässiger Speicherzugriff
0xffff0018	Interrupt request (IRQ)	
0xffff001c	Fast IRQ (FIQ)	höhere Priorität

- EVT-Einträge enthalten in der Regel eine Sprunganweisung (B) an die Speicheradresse des **Handlers**. Zum Beispiel steht an Adresse 0xffff0008 ein Sprung zum Linux-Syscall-Handler.
- Die **höchstwertigen 16 Bit** der Speicheradresse des EVT werden durch ein Statusbit festgelegt (z.B. 1 bei Linux, 0 beim CPULator).

Bei x86: Interrupt Descriptor Table (IDT), abhängig vom Betriebsmodus

Aufbau eines Interrupt-Handlers

Minimalbeispiel für eine Quelle

irq:

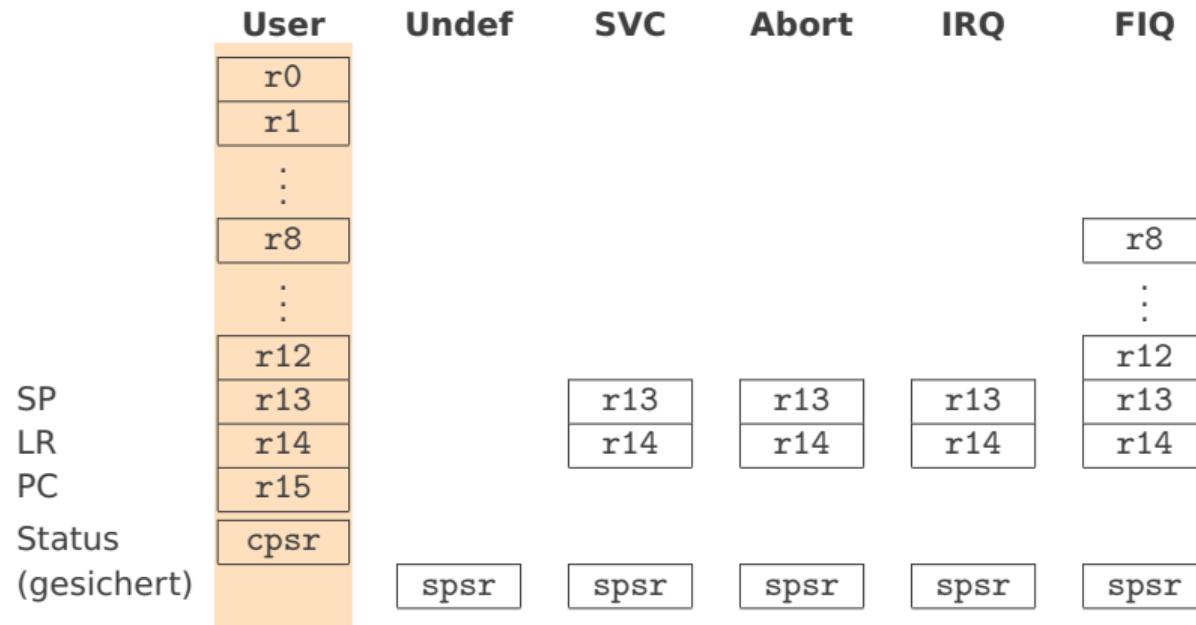
```
STMFD  sp!, {r0-r12,lr} ; Register auf Stapel sichern (Push)
BL      do_something     ; Interrupt-Logik
LDR    r0, =timerbase   ; Interrupt-Quelle zurücksetzen
STR    r0, [r0+0x0c]     ; hier z. B. ARM SP804 Timer
LDMFD  sp!, {r0-r12,lr} ; Register wiederherstellen (Pop)
SUBS  pc, lr, #4        ; Sprung an korrigierte Rücksprungadresse
                           ; und Statusregister wiederherstellen
```

Interrupt-Multiplexing

Wenn mehrere Quellen IRQs auslösen, findet der Interrupt-Handler die Quelle heraus (durch Auslesen aller infrage kommenden *interrupt pending*-Bits) und verzweigt in das dazugehörige Unterprogramm.

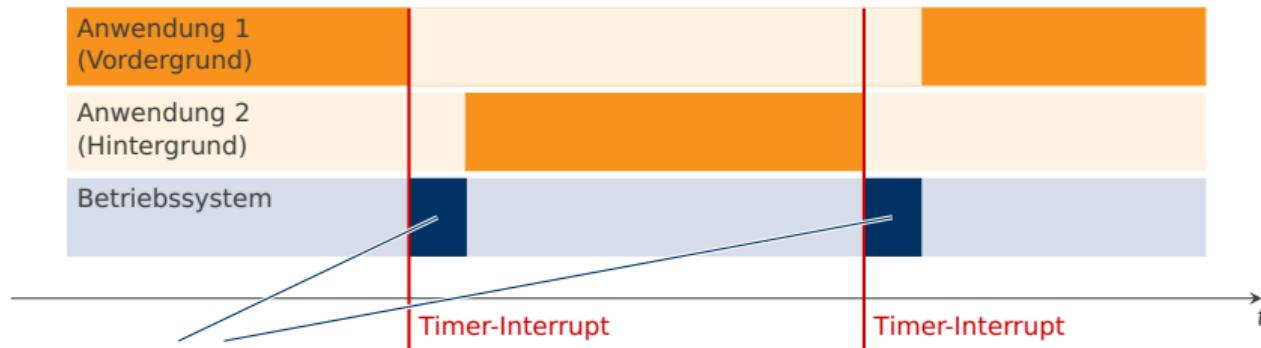
Registersatz mit Bänken

Abhängig vom **Modus** trennt ARM einige Registerinhalte nach Bänken:



Einsatz zum Mehrprozessbetrieb auf einem Kern

(engl.: *multitasking, time sharing*)



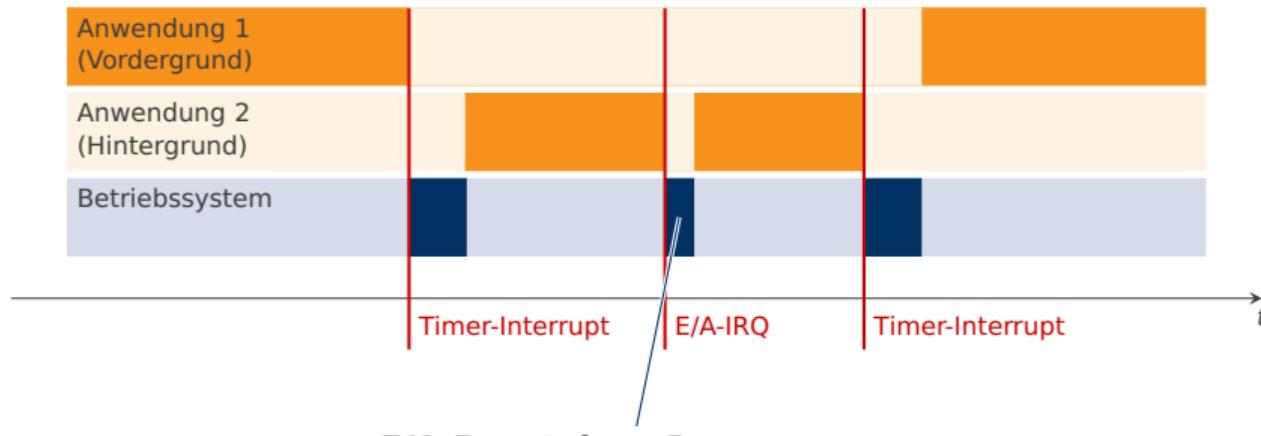
Taskwechsel

- Register sichern
- Stack-Rahmen anpassen (pro Prozess ein Stapel)
- Zugriffsrechte (Kontext) anpassen
- Register des nächsten Prozesses wiederherstellen
- „Rücksprung“

Vertiefung in **Betriebssysteme**, Pflichtmodul, 2. Semester

Einsatz zum Mehrprozessbetrieb auf einem Kern

(engl.: *multitasking, time sharing*)



E/A-Baustein, z. B.

- Mausbewegung
- Datenpaket vom Netz

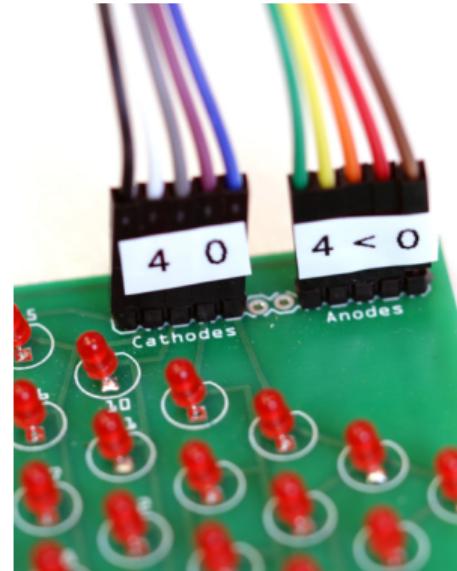
Vertiefung in **Betriebssysteme**, Pflichtmodul, 2. Semester

Gliederung heute

- 1. Touch-Eingabetechnologien**
- 2. Ansteuerung von E/A-Bausteinen**
- 3. Unterbrechungsanforderungen**
- 4. Praxisbeispiel zur Ausgabe**

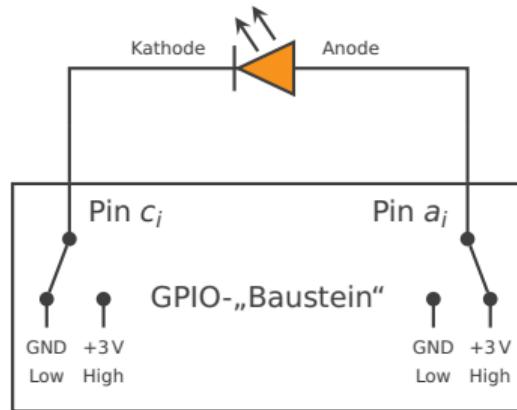
Hardware

Der Raspberry Pi Zero V1.3 basiert auf einem System-on-a-Chip (SoC) mit 1 GHz ARM-CPU und *general purpose I/O* (GPIO)-Pins.



Ansteuerung einer Leuchtdiode

Leuchtdiode (LED)



Pin		
c_i	a_i	LED
L	L	aus
L	H	an
H	L	aus
H	H	aus

Software

Drei Schritte zur Initialisierung (LEDs aus)

1. Pins im Kontrollregister als Ausgänge schalten

```
LDR  r9, =0x20200000 ; Basisadresse der GPIO-Komponente  
MOV  r0, #1           ; Bit-Kodierung 001 für „Ausgang“ nach r0  
STR  r0, [r9, #0]      ; setze Funktion in GPFSEL-Register  
...                  ; wiederholen für weitere Pins
```

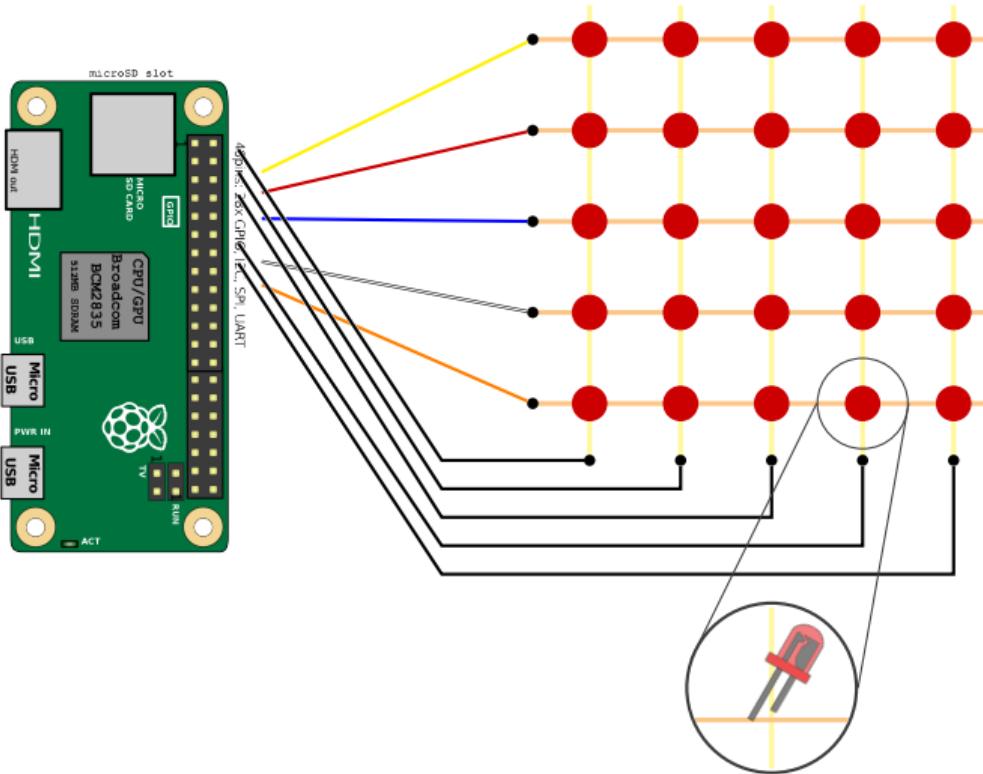
2. Kathoden auf H schalten (über Set-Datenregister)

```
                   ; r1 enthält gesetztes Bits pro Kathoden-Pin  
STR  r1, [r9, #28]   ; Bits in GPSET0-Register (Basis+28) setzen
```

3. Anoden auf L schalten (über Clear-Datenregister)

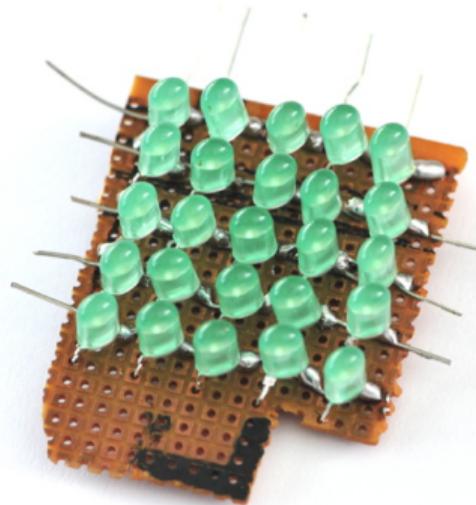
```
                   ; r2 enthält gesetztes Bit pro Anoden-Pin  
STR  r2, [r9, #40]   ; Bits in GPCLR0-Register setzen
```

Schaltung für 25 LEDs



Erste Umsetzung

für eine Ansteuerung mit 10 Pins



Bits im GPIO-Datenregister

Kathoden 2, 3, 4, 17, 27

Anoden 14, 15, 18, 23, 24

→ Ansteuerung unter Nutzung menschlicher Wahrnehmungsschwächen

Registerbelegung

Register Funktion

r0	Hilfsregister, temporäre Belegung
r1	Kathoden-Maske: GPIO_0 <i>clear</i> für LED an; in Spalten
r2	Anoden-Maske: GPIO_0 <i>set</i> für LED an; in Zeilen
r3	Laufvariable für Kathoden-Pins (stets nur ein Bit gesetzt)
r4	Laufvariable für Anoden-Pins (stets nur ein Bit gesetzt)
r5	Laufvariable für LED 0, ..., 24
r6	LED-Bild in Bits 0–24 (Bit gesetzt \Rightarrow LED an)
r7	Spaltenbild: Einsen für alle Zeilen-Pins, wo LEDs leuchten
r8	Schleifenzähler der Bildwiederholung
r9	Basisadresse der GPIO-Register

Steuerung der Laufvariablen für Pins

Idee: Ein Bit „läuft“ durch Rotation alle Pins ab.
Masken-Register zeigen gültige Bit-Positionen an.

```
LDR    r1, =0x0802001c ; Kathoden-Maske
LDR    r2, =0x0184c000 ; Anoden-Maske
LDR    r3, =0x00000004 ; Laufvariable für Kathoden-Pins
LDR    r4, =0x00004000 ; Laufvariable für Anoden-Pins
...
shiftAnod:
    TST    r2, r4          ; testen, ob die Eins richtig steht
    MOVEQ  r4, r4, ROR #31 ; falls nicht, eins nach links rotieren
    BEQ    shiftAnod
shiftCath:
    TST    r1,r3           ; analog für Kathoden
    ...
...
```

Ablaufskizze

r5	r3	r4	
0	0x00000004	0x00004000	Bitweise OR-Verknüpfung aller Belegungen von r4, bei denen LED an $\Leftrightarrow r6_{(r5)} = 1$.
1	0x00000004	0x00008000	
2	0x00000004	0x00040000	
3	0x00000004	0x00800000	
4	0x00000004	0x01000000	
5	0x00000008	0x00004000	
:	:	:	
9	0x00000008	0x01000000	
:	:	:	
24	0x08000000	0x01000000	
nächste Spalte			
usw.			

Masken	r1	r2
	0x0802001c	0x0184c000

Spaltensteuerung

LED-Bild befindet sich in Bits 0–24 von **r6**.

```
        MOV      r5, #0          ; LED-Zähler initialisieren
        ...
rowloop:
        MOV      r7, #0          ; Spaltenbild leeren
        ...
shiftAnod: ...
        MOV      r0, r6, LSR r5 ; Bitmaske nach rechts schieben
        TST      r0, #1          ; testen, ob diese LED leuchten soll
        ORRNE   r7, r7, r4       ; falls ja, OR-verknüpfen

        ADD      r5, r5, #1       ; 25 LED Zähler inkrementieren
        MOV      r4, r4, ROR #31 ; Anoden-Maske verschieben
        CMP      r4, r2          ; Spalte fertig ?
        BLO      shiftAnod      ; wiederhole, falls nicht
```

Ansteuerung des GPIO-Bausteins

Programmierte Ein-/Ausgabe über Datenregister

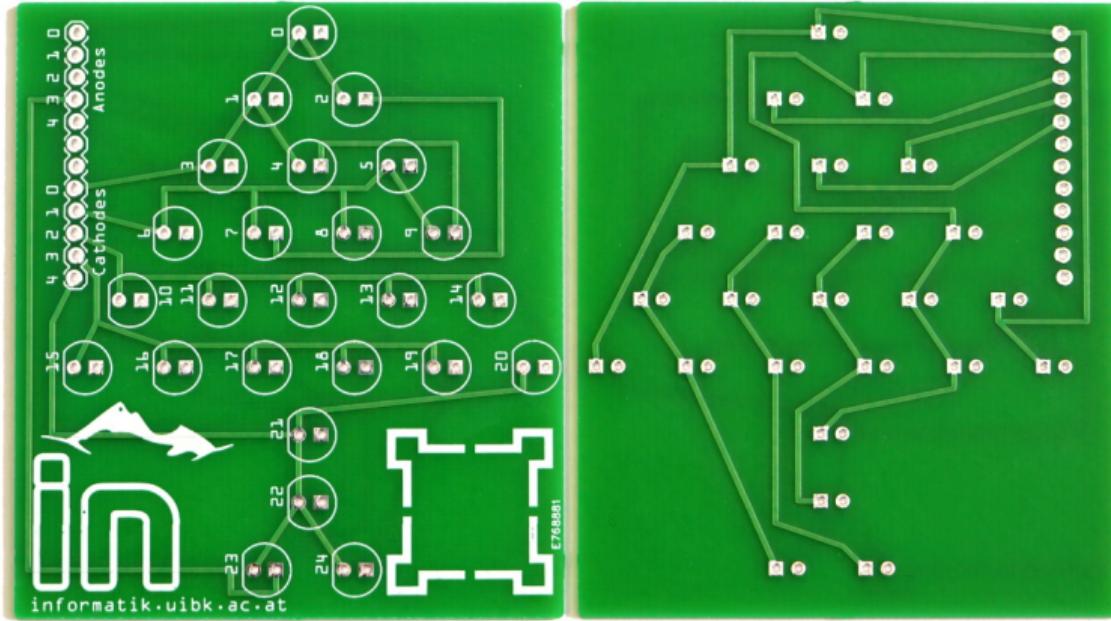
(LED-Spalte ein, warten, aus)

```
STR  r7, [r9, #28] ; gesammelte Anoden auf H  
STR  r3, [r9, #40] ; entsprechende Kathoden auf L  
  
LDR  r0, =50        ; Wartezeit 50 µs  
BL   wait          ; aktives Warten über Timer (Folie 16)  
  
STR  r3, [r9, #28] ; Kathoden wieder H  
STR  r7, [r9, #40] ; Anoden wieder L  
  
MOV  r7, #0         ; Spaltenbild zurücksetzen
```

Wir ersparen uns (und Ihnen) den Rest der Schleifenlogik.

Leiterplatte

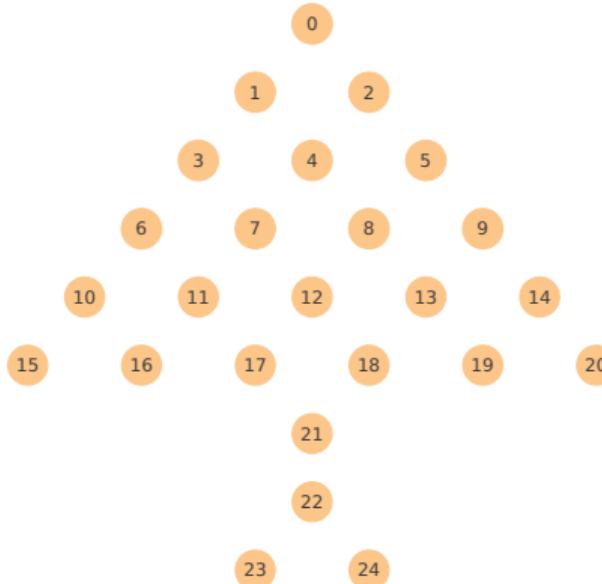
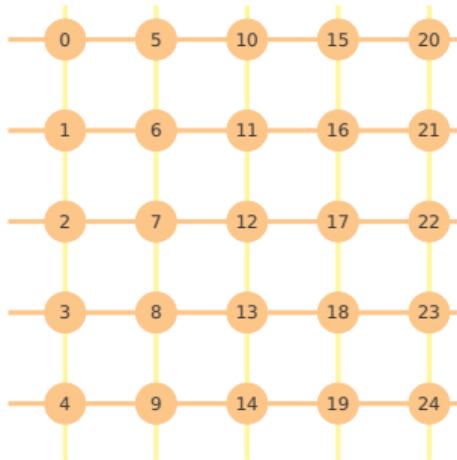
(engl. *printed circuit board, PCB*)



Vorderseite

Rückseite

Von der Matrix zum Baum



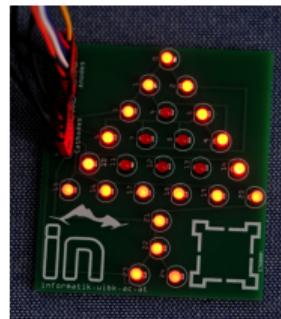
Beispielanimation: Rotation eines Bits in $r6_{(24)}, \dots, r6_{(0)}$.

Hoh, Hoh, Hörsaalfrage

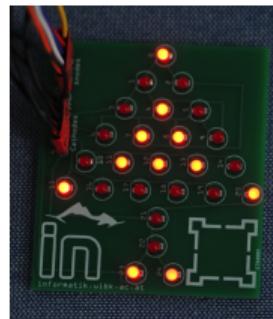


24 82 94 16

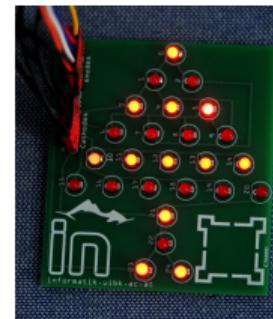
Welche Ausgabe erhalten Sie, wenn Sie das gezeigte Programm mit dem Wert 0x00262a6f in Register r6 aufrufen ?



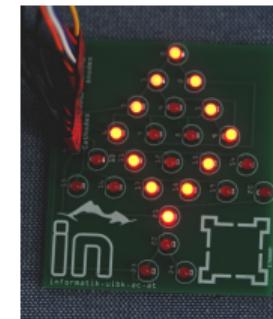
Antwort A



Antwort B



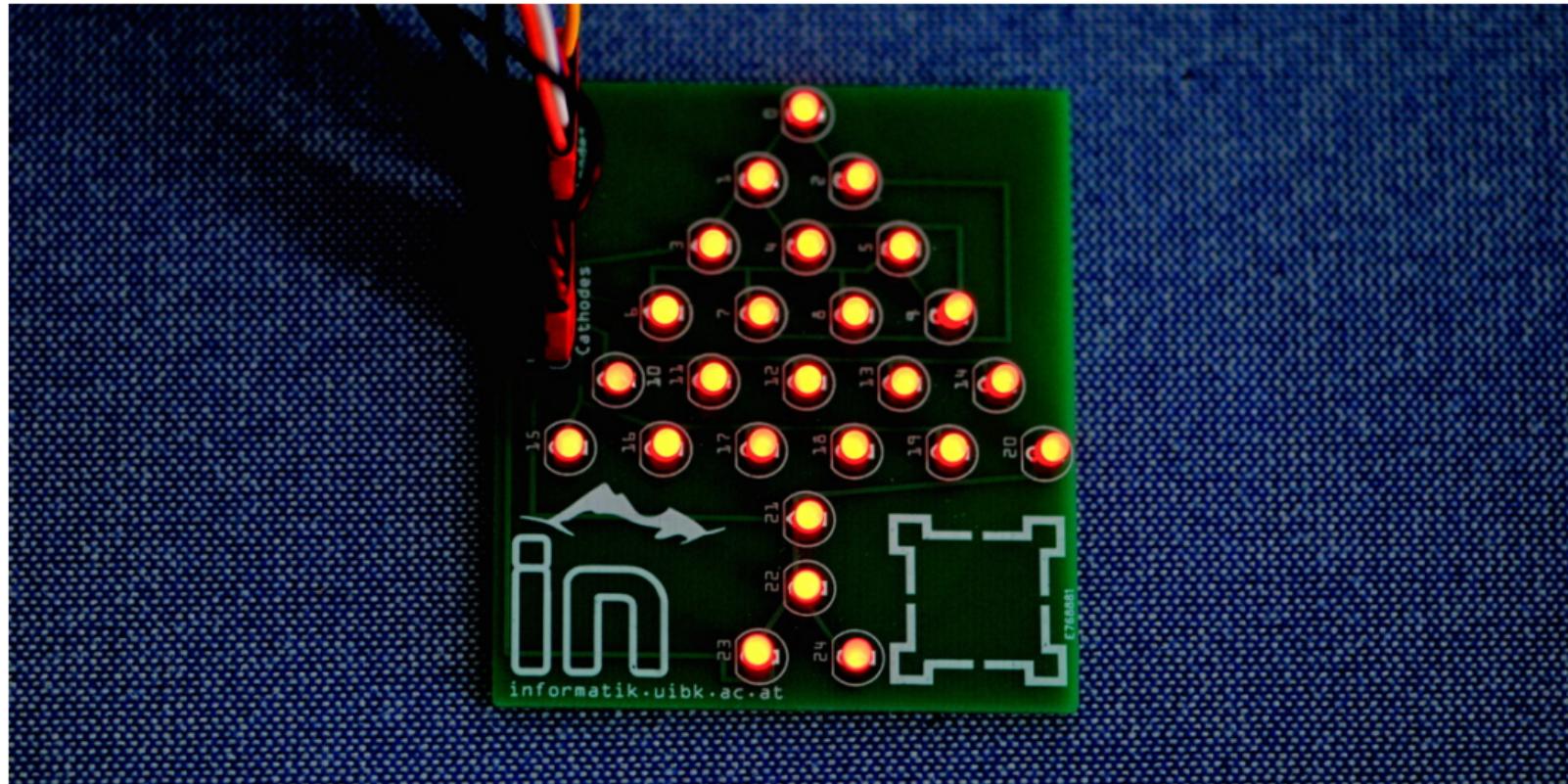
Antwort C



Antwort D

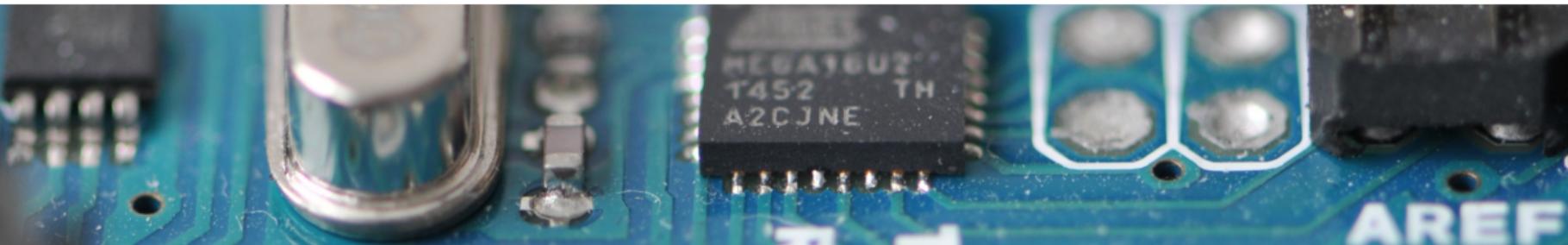
Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Frohes Fest



Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Prozessorarchitekturen

Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2021/22 · 12. Jänner 2022

Erinnerung

Nicht vergessen !

Melden Sie sich online bis **spätestens 19.01.2022** zum ersten Klausurtermin an.
Nachmeldungen per E-Mail können wir **nicht** berücksichtigen.

Vorbemerkung

Das heutige Thema ist eine Erweiterung Ihres Horizonts.

Behalten Sie den Überblick!

Kein Beispiel lässt sich direkt mit Ihren Kenntnissen der ARM-Architektur kombinieren oder umsetzen.

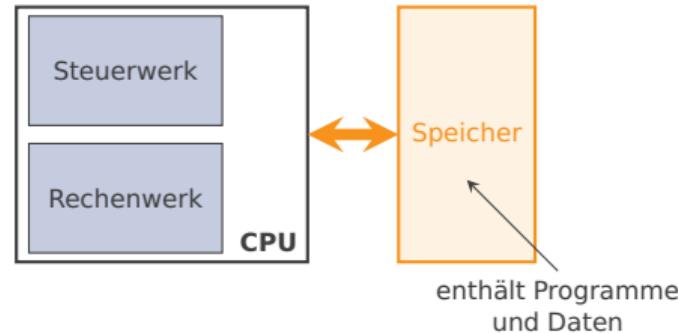
Im Proseminar und in der kommenden Vorlesung zur Ein- und Ausgabe nutzen und vertiefen wir weiterhin die ARM-Assemblerprogrammierung.

Gliederung heute

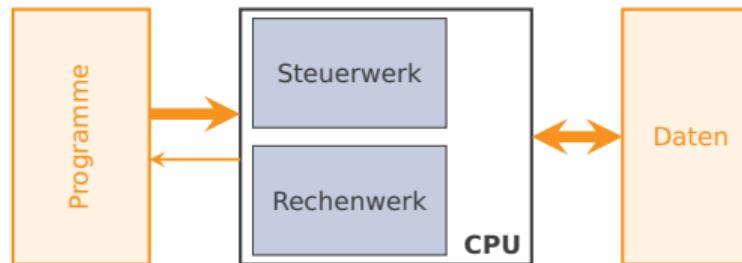
- 1. Klassifikation von Prozessorarchitekturen**
- 2. Intel x86-Architektur**
- 3. Datenparallele Architekturen**

Klassifikation nach Anbindung des Speichers

Von-Neumann-Architektur



Harvard-Architektur



nach dem Relaisrechner Mark I von Aiken (Harvard Universität) und IBM (1943/44)

Klassifikation nach Anbindung der ALU

Einteilung von Mikroarchitekturen nach möglichen Quellen und Zielen von arithmetisch-logischen Operationen:

- Register/Register-basiert
- Register/Speicher-basiert
- Akkumulator-basiert
- Stapel-basiert

Register

r1
r2
r3

Speicher



Register/Register-basierte Architekturen

Werte aus zwei Registern werden miteinander verknüpft.

Speicherzugriffe erfolgen separat (*Load-Store-Architekturen*).

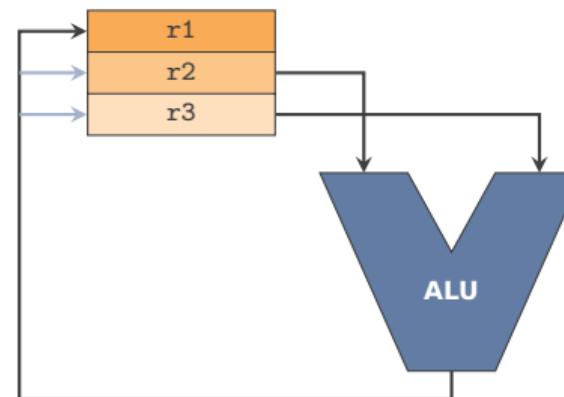
Register

Op. 1: Register

Op. 2: Register

Ergebnis: Register

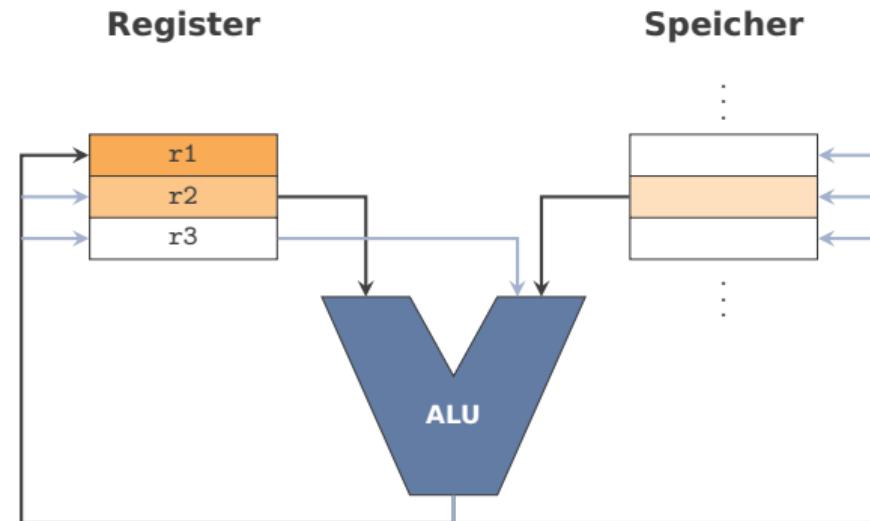
Beispiele: ARM, MIPS,
DLX, RISC-V,
SPARC, AVR



Register/Speicher-basierte Architekturen

Ein Wert aus einem der Register wird mit einem Wert aus einem Register oder dem Speicher verknüpft.

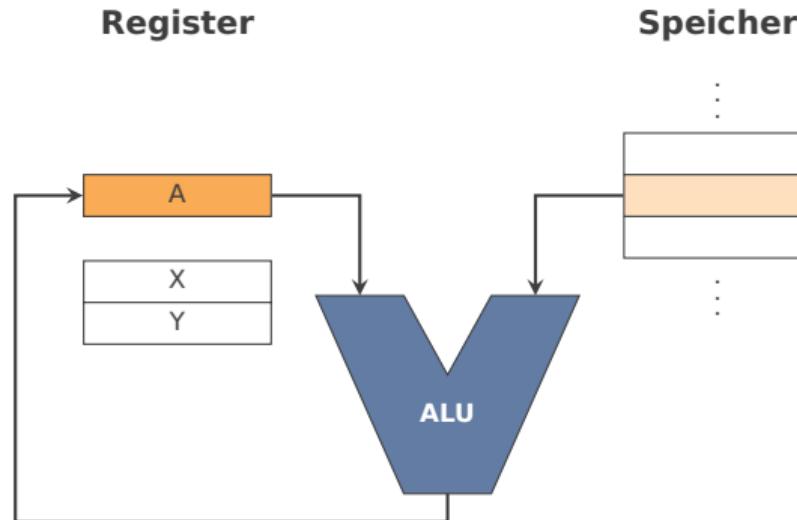
- Op. 1: Register
- Op. 2: Register o. Speicher
- Ergebnis: Register o. Speicher
- Beispiele: Intel x86,
Motorola 68K,
PowerPC



Akkumulator-basierte Architekturen

Der Wert aus einem Spezialregister (**Akkumulator**) wird mit einem Wert aus dem Speicher verknüpft. Das Ergebnis kommt immer in den Akkumulator.

Op. 1: Akkumulator
Op. 2: Speicher
Ergebnis: Akkumulator
Beispiele: Intel 4040,
MOS 6502,
Zilog Z80

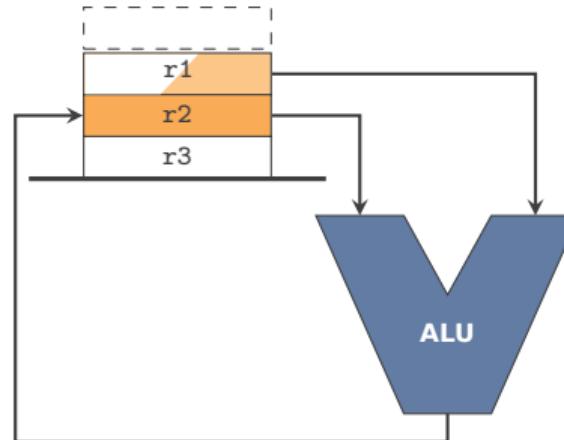


Stapel-basierte Architekturen

Alle Register werden als Stapel organisiert. Die **obersten beiden** Werte auf dem Stapel werden miteinander verknüpft und das Ergebnis wieder oben auf dem Stapel abgelegt.

- Op. 1: Wert unter Stapelspitze
- Op. 2: Stapelspitze
- Ergebnis: (neue) Stapelspitze
- Beispiele: x87 FPU,
Java VM, WASM,
Ethereum VM

Register



Klassifikation nach Komplexität des Befehlssatzes

RISC: Reduced Instruction Set Computer

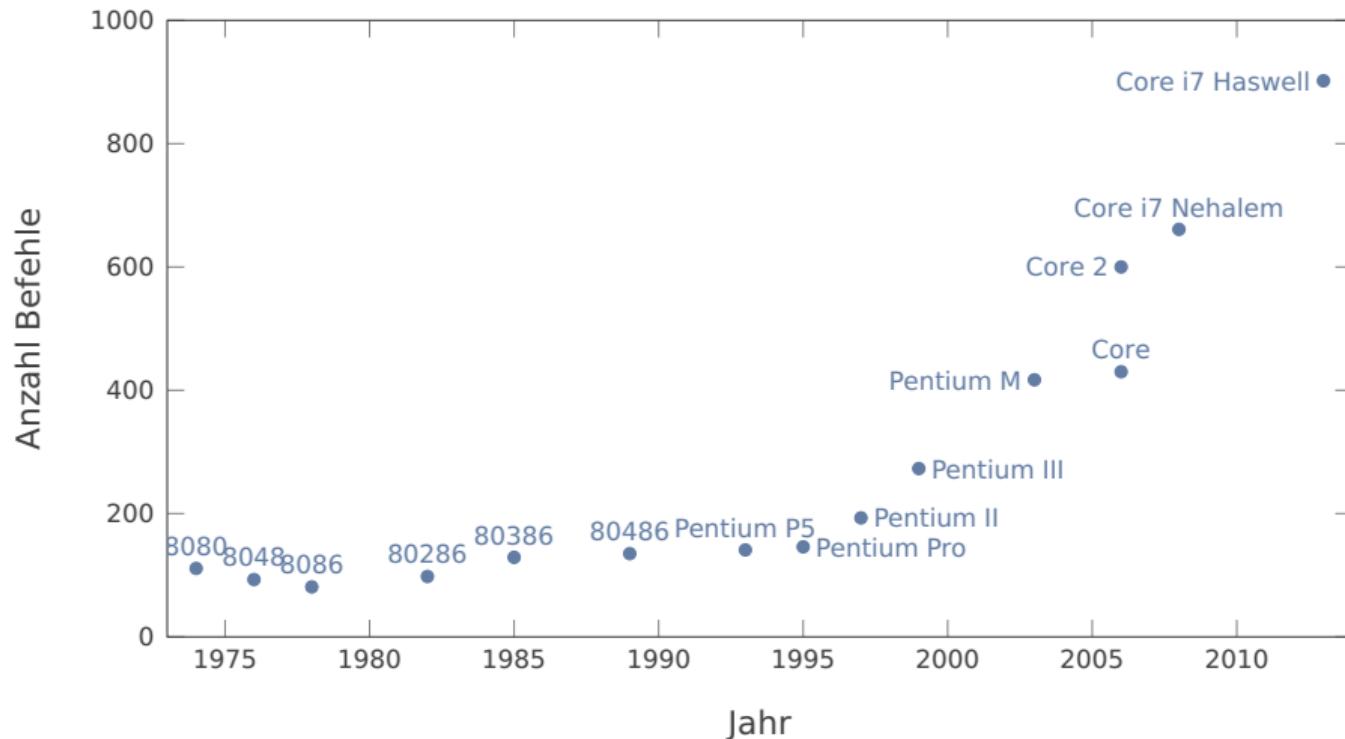
- wenige elementare Maschinenbefehle
- ermöglichen schlanken Pipelines: Richtwert ist ein Taktzyklus pro Stufe
- kompakte Instruktionskodierung, oft **orthogonal**
(→ Einschränkungen bei Immediate-Werten und Addressierungsarten)
- Load-Store-Architekturen

CISC: Complex Instruction Set Computer

- viele mächtige Spezialbefehle, z. T. in **Mikroprogrammen** realisiert
- Optimierung schwierig
- organisches Wachstum der Befehlssätze und Instruktionskodierung
- Compiler nutzen oft nur eine Untermenge der verfügbaren Befehle.

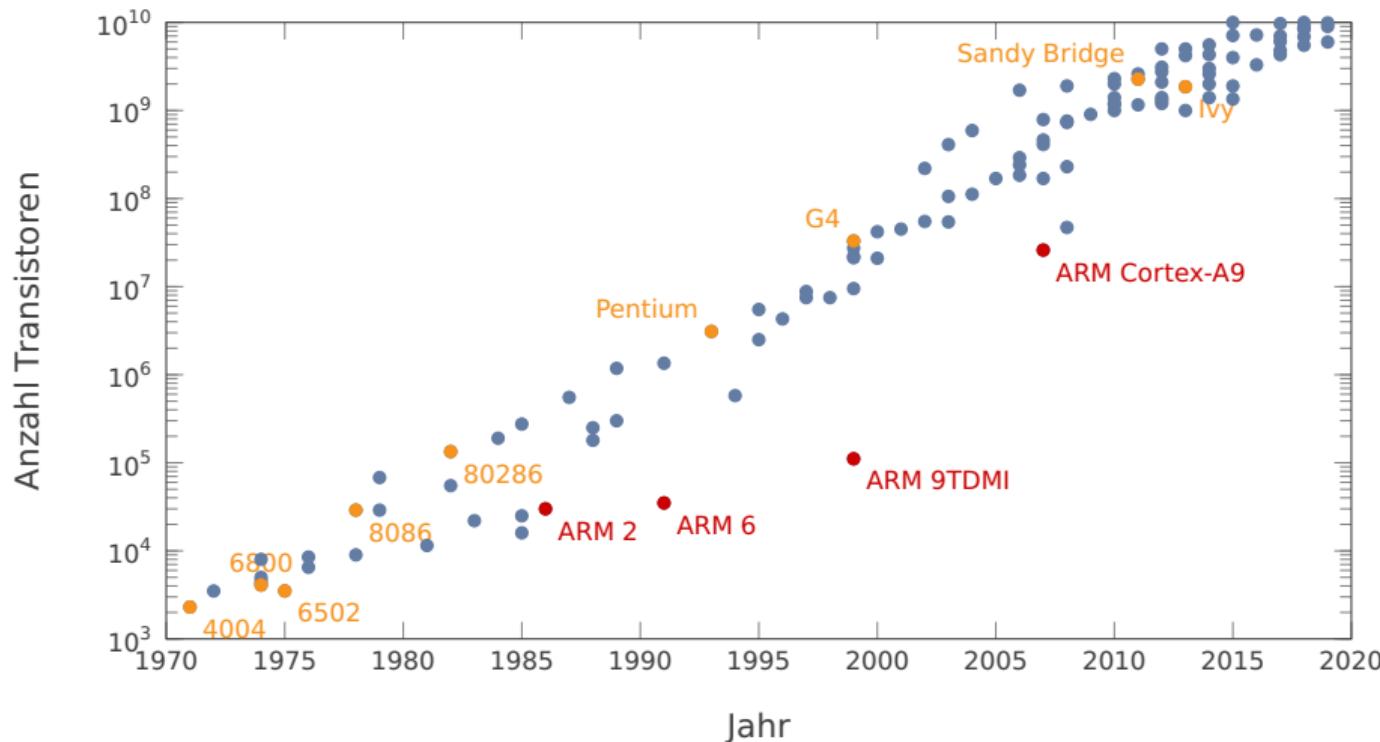
Kompromisse: RISC und CISC existieren heute selten in Reinform.

Anzahl der Instruktionen von CISC-Prozessoren



Datenquelle: Wikipedia 2016

Anzahl der Transistoren in Mikroprozessoren



Datenquelle: Wikipedia 2020

Klassifikation nach Art der Parallelverarbeitung

		Anzahl der Befehlsströme	
		1	> 1
Anzahl der Datenströme		<i>single instruction</i>	<i>multiple instruction</i>
1	<i>single data</i>	SISD	MISD
	<i>multiple data</i>	SIMD	MIMD

Flynn 1966

Gliederung heute

1. Klassifikation von Prozessorarchitekturen
2. **Intel x86-Architektur**
3. Datenparallele Architekturen

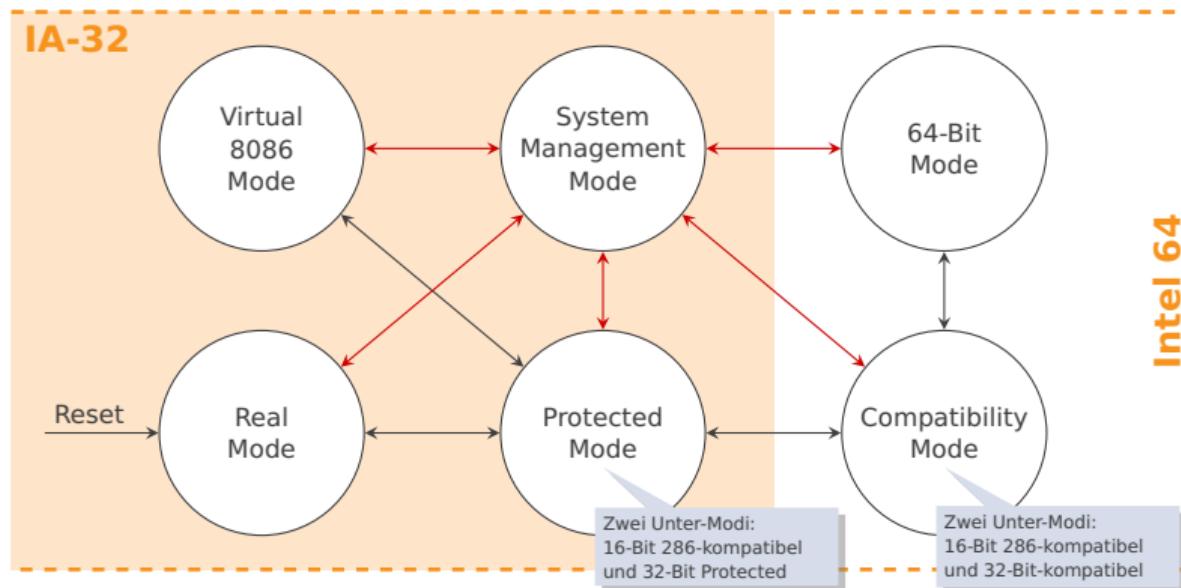
Entwicklung der x86-CISC-Architektur

- 1978 Intel 8086** erscheint (**16-Bit**-Architektur)
- 1980 Intel 8087 FPU (Koprozessor für Gleitkommazahlen)**
- 1982** Intel 80286, 24 Bit Adressraum, neue Instruktionen
- 1985 Intel 80386 (32-Bit**-Architektur), neue Adressierung
- 1989** Intel 80486 (weniger Mikrokode, **Integration der FPU** ab 486 DX)
- 1993** Intel **Pentium**: Integration von **RISC**-Prinzipien (1995: Pentium Pro)
- 1997** Pentium II mit 57 neuen **MMX-Instruktionen** (Ganzzahl-**SIMD** für Multimedia)
- 1999** Pentium III mit 70 neuen **SSE-Instruktionen** (Gleitkomma-**SIMD**)
- 2001** Pentium 4 mit 144 neuen SSE2-Instruktionen (später: SSE3, SSE4, ..., AVX-512)
- 2003** **64-Bit**-Architektur von **AMD**, seit 2004 von Intel unterstützt
- 2006** Hardwareunterstützte **Virtualisierung** (AMD-V bzw. Intel VT-x)
- 2015** Vertrauenswürdige Laufzeitumgebung (TEE) – „Enklave“

Fett gedruckte Begriffe sollte jede InformatikerIn kennen und sind prüfungsrelevant.

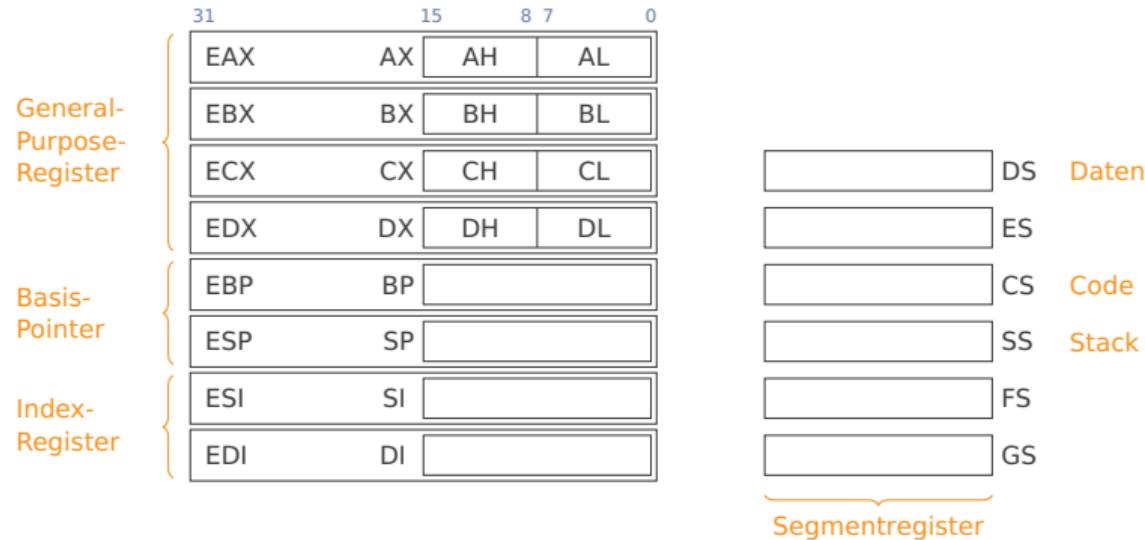
Problemfeld Abwärtskompatibilität

am Beispiel der Betriebsmodi aller modernen x86-Prozessoren



Alternative: Kontinuität; sonst neu kompilieren, nach einiger Zeit emulieren

Registersatz



Im 16-Bit-Modus steuern **Segmente** die höchstwertigen Adressbits.

(vereinfacht: 32-Bit-Modus; ohne Koprozessor, Kontrollregister, 128-Bit-Media-Register)

ALU-Anbindung

ALU-Befehle haben in der Regel **zwei Operanden**:

- Der erste Operand ist gleichzeitig Quelle und Ziel.

ADD AX, BX ; $a = a + b$ (Gleichheitszeichen ist Zuweisung)
ADD DX, 13 ; $d = d + 13$

- Maximal ein Operand darf ein Speicherwort sein.

ADD AX, mem16 ; $a = a + m_{16}$
ADD mem16, AX ; $m_{16} = m_{16} + a$
ADD mem16, 42 ; $m_{16} = m_{16} + 42$

- Kürzere Spezialbefehle:

INC ESI ; $i = i + 1$
DEC mem8 ; $m_8 = m_8 - 1$

Achtung: Einige Befehle sind fest mit definierten Registern verknüpft
(z. B. MUL und DIV für Ganzzahl-Punktoperationen mit AX und DX)

Adressierungsarten

Absolut

MOV EAX, adr ; Register mit Inhalt an Speicheradresse adr laden

Indirekt

MOV EAX, [EBX] ; Register EBX zeigt auf Speicheradresse

Basis mit **Index** (nur {EBP | EBX} plus {ESI | EDI})

MOV EAX, [EBX + ESI] ; Adresse aus EBX und ESI berechnen

... sowie **Displacement** (8, 16, 32 Bit) und **Skalierung** (Faktor 2, 4, 8)

MOV EAX, [EBX + ESI*4 + 2] ; nützlich für Zugriff auf Felder

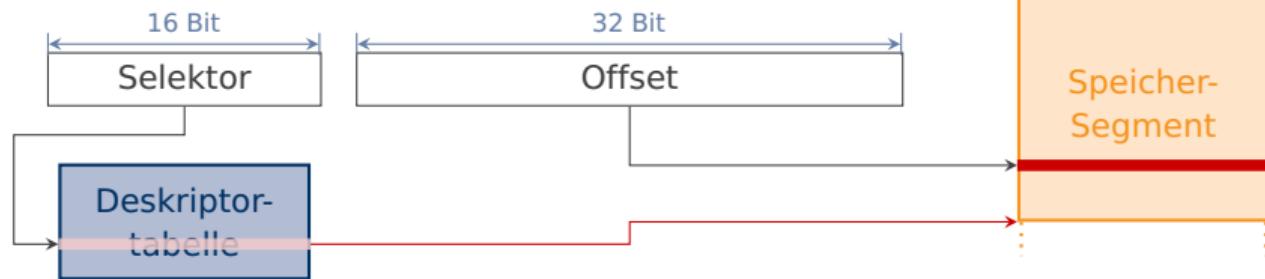
Segmentüberschreibung (kombinierbar mit allen Adressierungsarten)

MOV EAX, ES:[EBX] ; abweichend vom Datensegment DS

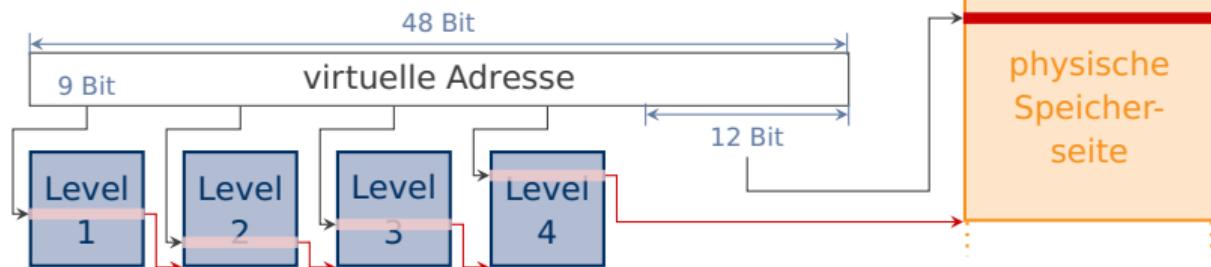
Die Art der Speichersegmentierung ist abhängig vom Betriebsmodus.

Berechnung der physischen Adresse

Segmentierung über Deskriptoren im 32-Bit *Protected Mode*



Mehrstufiges Paging im 64-Bit-Modus



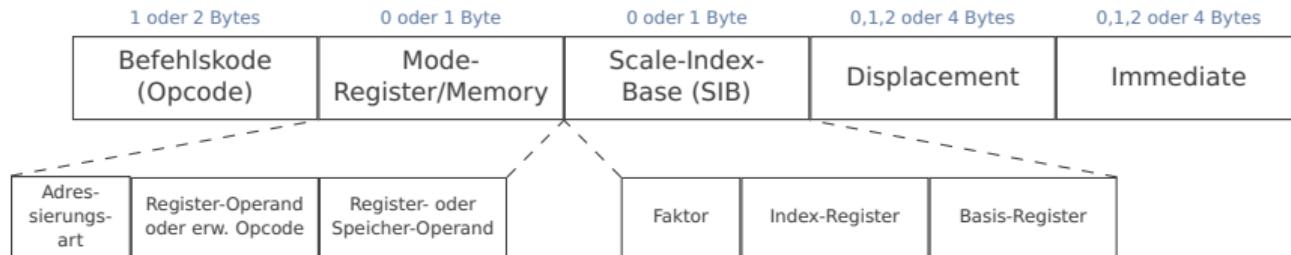
Instruktionskodierung

CISC-Instruktionen setzen sich aus mehreren Komponenten zusammen:

1. Optionale Präfixe

0 oder 1 Byte	0 oder 1 Byte	0 oder 1 Byte	0 oder 1 Byte
Instruktions-Präfix	Adressgrößen-Präfix	Operandgrößen-Präfix	Segmentüberschreibung

2. Allgemeines Instruktionsformat



Die Länge von x86-Instruktionen variiert zwischen 1 und 16 Bytes.

Stapelorganisation

Ein *full descending* Stapel wird vom Prozessor über das Registerpaar SS:ESP (*stack segment : extended stack pointer*) organisiert.

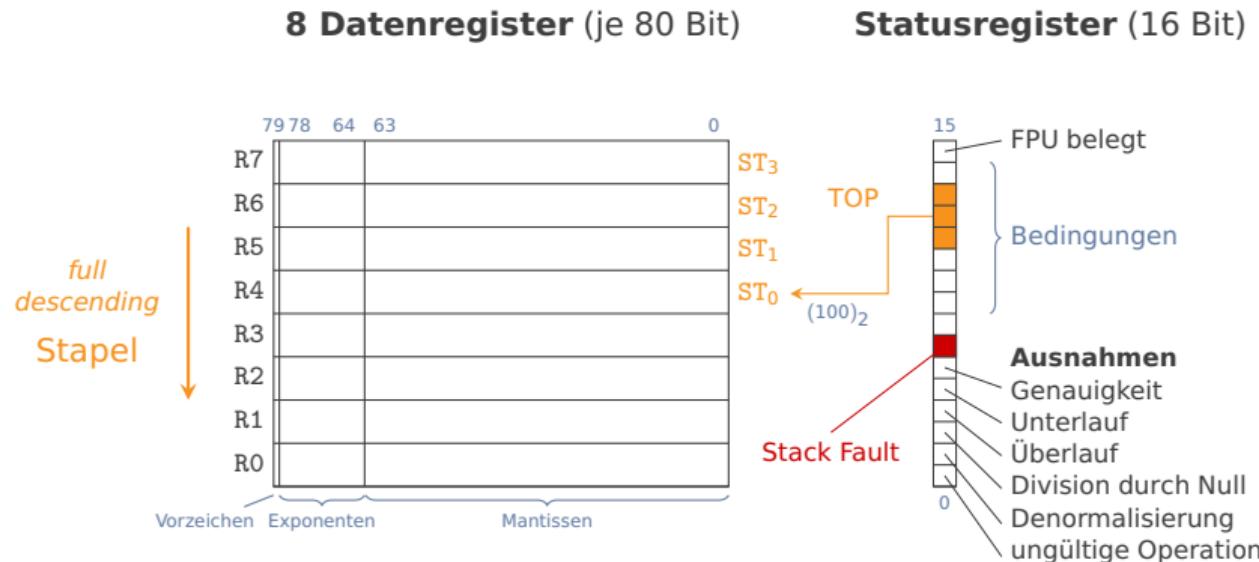
Spezialbefehle (trifft man oft beim Disassemblieren an)

Mnemonic	Kommentar
PUSH	Legt Register, Speicherinhalt oder Konstante auf Stapel.
POP	Holt Wert vom Stapel in Register oder Speicherinhalt.
CALL	Aufruf eines Unterprogramms
RET ^N	Rücksprung von Unterprogramm
ENTER	Platz aus dem Stapel für lokale Variablen reservieren
LEAVE	Platz für lokale Variablen freigeben

Gleitkommaeinheit

(engl. *floating point unit*, FPU)

Realisierung einer Stapel-basierten Befehlssatzarchitektur



Ausgewählte FPU-Befehle zum Datentransfer

Mnemonic	Kommentar		
Gleitkomma-, Ganzzahl, BCD*-Zahl			
FLD	FILD	FBLD	aus dem Speicher nach ST ₀ laden
FST	FIST		aus ST ₀ in den Speicher schreiben
FSTP	FISTP	FBSTP	– “ – und vom Stapel löschen (<i>pop</i>)

- Die FPU konvertiert automatisch zwischen Zahlendarstellungen.
- Die FPU unterstützt keine Immediate-Werte.
- Wichtige mathematische Konstanten liegen im ROM vor (z. B. FLDPI lädt die Kreiszahl π nach ST₀).

* : Die BCD-Kodierung speichert zwei Dezimalstellen pro Byte.
Werte, deren Hex-Darstellung Buchstaben erfordert, sind unzulässig.

Ausgewählte Arithmetik-Befehle der FPU

Mnemonics			Kommentar
FADD	FADDP	FIADD	Gleitkomma-Addition
FSUB	FSUBP	FISUB	Gleitkomma-Subtraktion
FMUL	FMULP	FIMUL	Gleitkomma-Multiplikation
FDIV	FDIVP	FIDIV	Gleitkomma-Division
FSQRT			Gleitkomma-Quadratwurzel
FABS			Absolutbetrag
FRNDINT			auf Ganzzahl runden

- Alle Gleitkomma-Operationen erfolgen nach IEEE 754.
- Varianten mit **I** erlauben Ganzzahl (Integer) als ersten Operand.
- Für Subtraktion und Division existieren “reverse”-Varianten mit vertauschten Operanden, z. B. FSUBP → FSUBRP, FIDIV → FIDIVR.

Beispiel für FPU-Programmierung

Berechnung eines Skalarprodukts $y = a_1 \cdot b_1 + a_2 \cdot b_2$

Assembler-Befehlsfolge

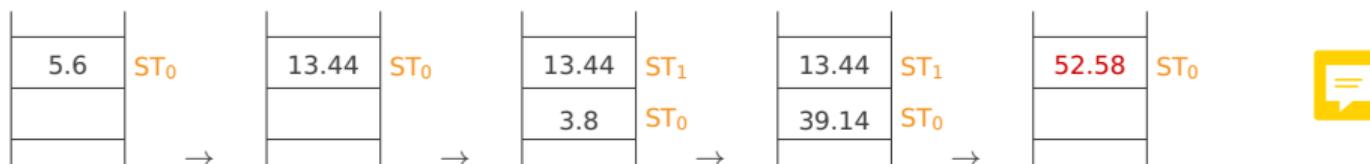
sprod:

```
FLD    adr1
FMUL   adr3
FLD    adr2
FMUL   adr4
FADDP
```

Belegungsbeispiel

Variable	Wert	Label (Speicheradresse)
a_1	5.6	adr1
a_2	3.8	adr2
b_1	2.4	adr3
b_2	10.3	adr4

Ablauf



Hörsaalfrage



24 82 94 16

Wie viele freie Speicherplätze benötigen Sie auf dem Stapel, um folgenden Ausdruck mit der FPU effizient zu berechnen (ohne dabei Zwischenergebnisse in den Speicher zu schreiben) ?

$$\mathbf{y} = \begin{pmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

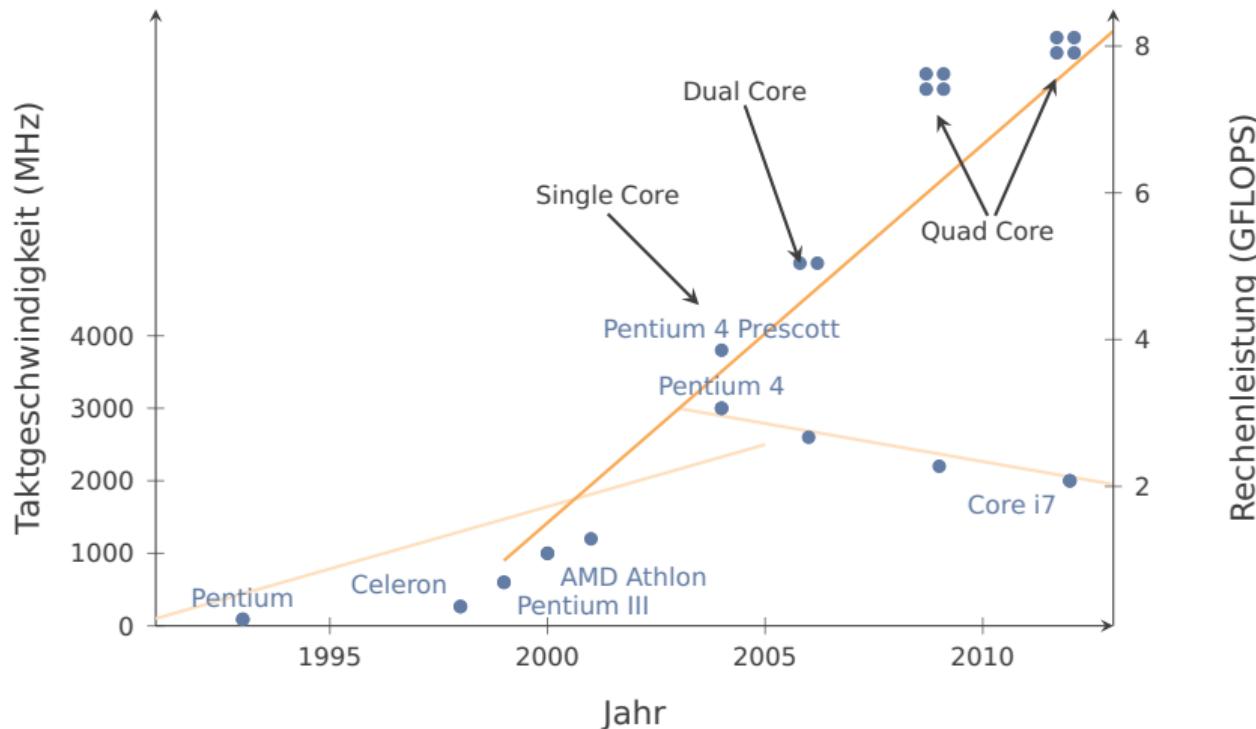
- a. 1
- b. 2
- c. 3
- d. 4
- e. 8
- f. 12
- g. 13
- h. 16

Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Gliederung heute

1. Klassifikation von Prozessorarchitekturen
2. Intel x86-Architektur
3. **Datenparallele Architekturen**

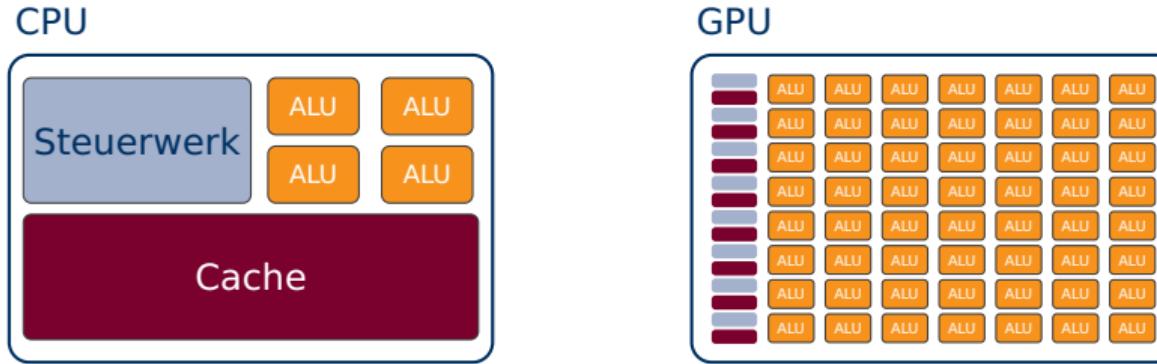
Entwicklung der Leistung von x86-Prozessoren



Quelle: https://en.wikipedia.org/wiki/Comparison_of_Intel_processors

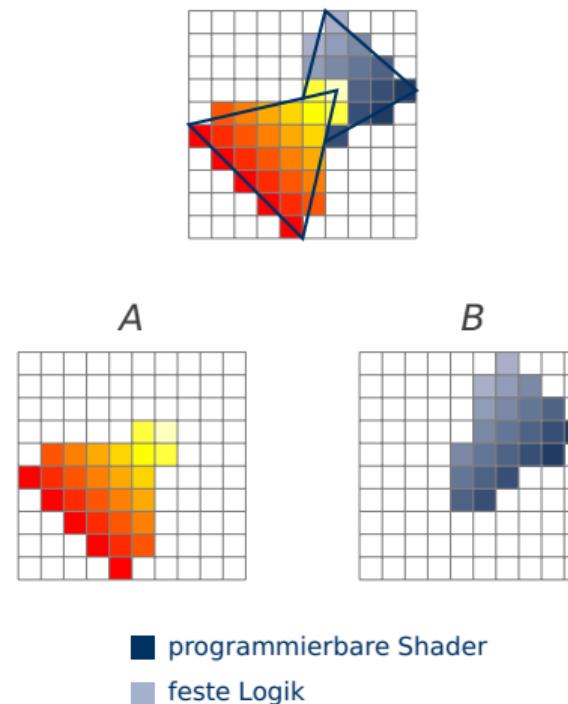
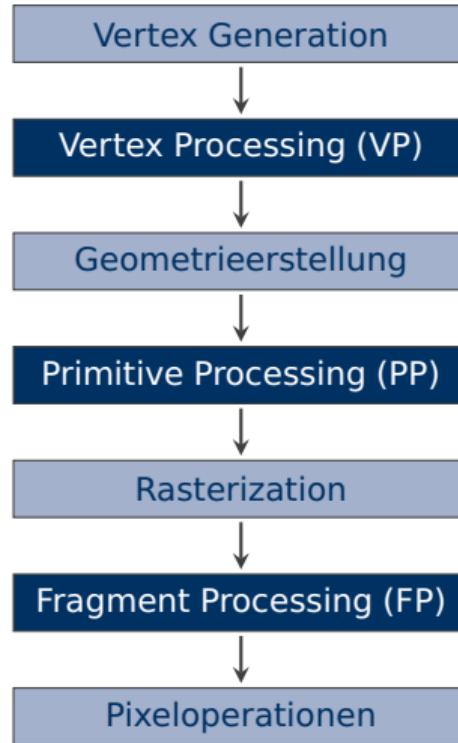
Datenparallele Architekturen am Beispiel von GPUs

Nutzung der Chipfläche von CPUs und Grafikprozessoren (GPUs)

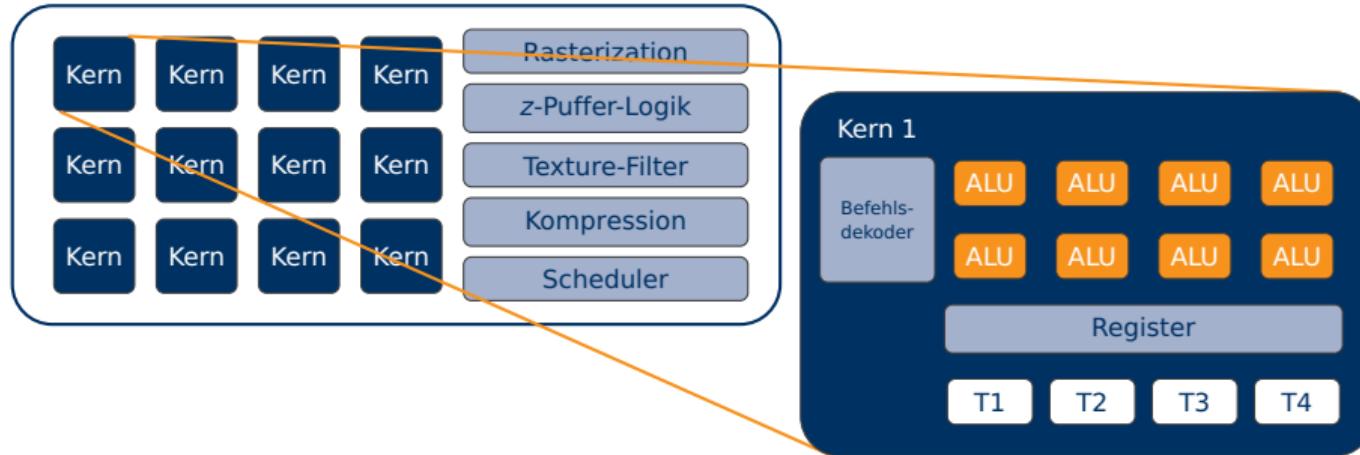


- **Datenparallelität:** Sehr effiziente, da gleichzeitige Bearbeitung vieler gleichartiger Daten auf die gleiche Weise.
- Voraussetzung: **geringe Datenabhängigkeit;** der Kontrollfluss ist weitgehend unabhängig von Zwischenergebnissen.
- Früher: **Vektorprozessoren** in Supercomputern, z. B. Cray

Grafik-Pipeline mit Hardwareunterstützung



Aufbau moderner GPUs



- Ca. 10^2 Kerne sind in einer Matrix-Struktur organisiert.
- **SIMD-Prinzip:** In jedem Kern teilen sich 10^1 ALUs ein Steuerwerk.
- **Hardware-Threads:** Die ALUs erledigen andere anstehende Aufgaben um die Speicherlatenz (10^2 – 10^3 Taktzyklen!) zu überbrücken.

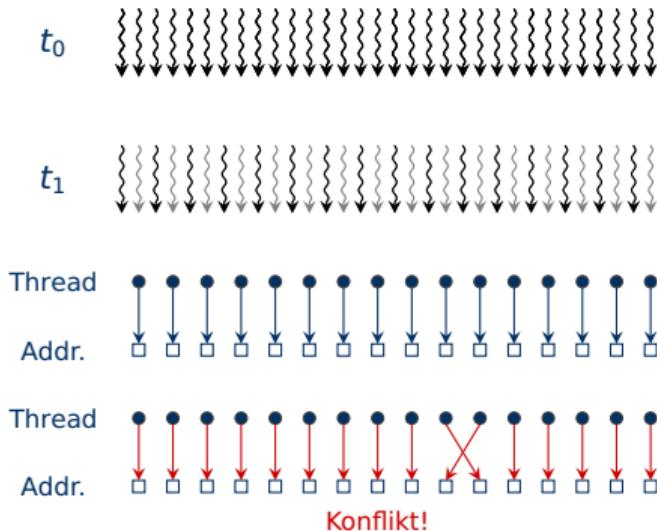
General Purpose GPU: Schnittstelle für beliebige Rechenaufgaben

Vertiefung in **Parallele Programmierung**, Pflichtmodul, 4. Semester BSc Informatik

Optimierung für Datenparallelität

(am Beispiel der Nvidia-CUDA-Architektur)

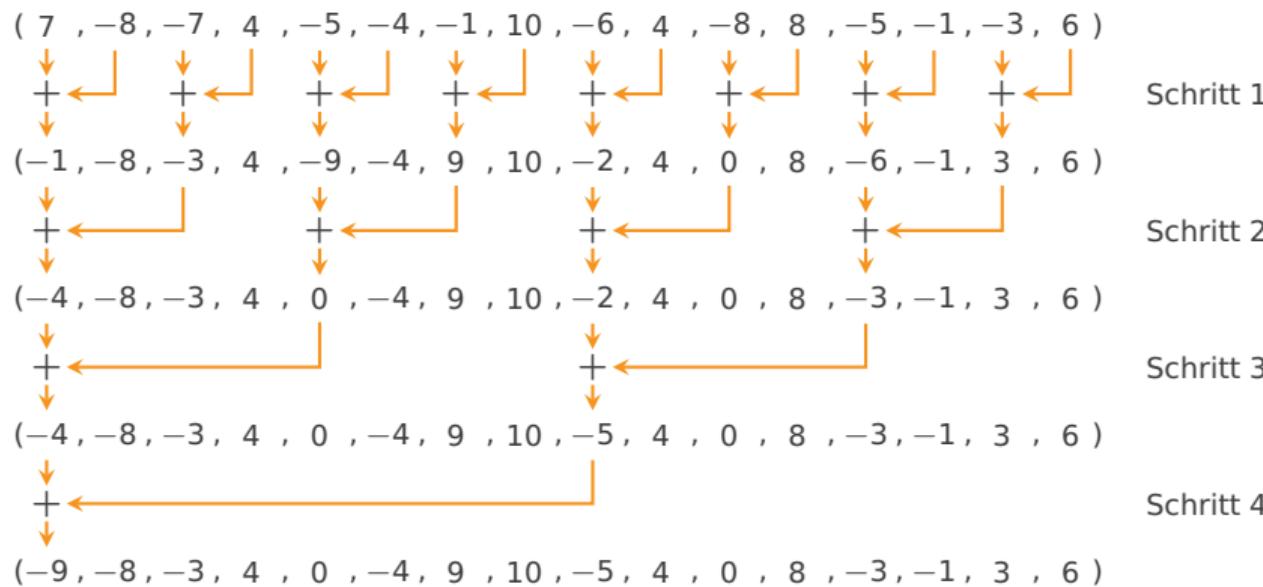
- **SIMD-Breite 32:** mind. 32 Threads müssen das Gleiche tun – bis auf Bedingungen
- **Globaler Speicher** ohne Cache: Alignment nötig, um 16-fache (!) Verzögerung zu vermeiden



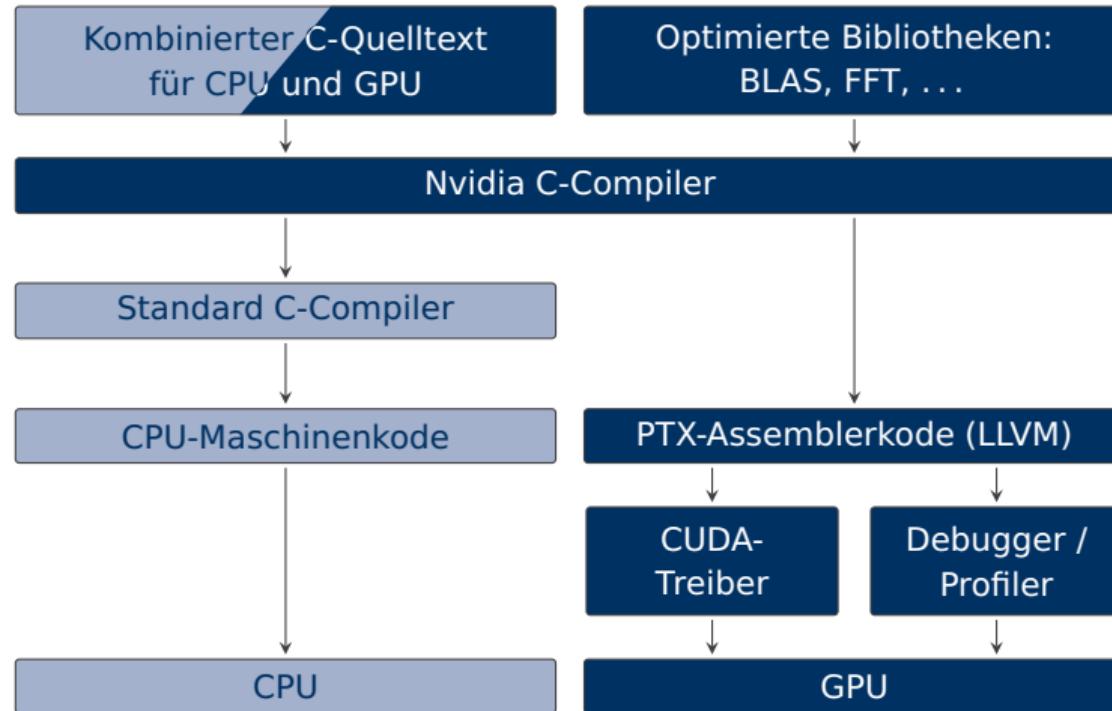
Grundsatz: Besser in Datenlayouts statt in Algorithmen denken !

Beispiel: Parallelreduktion

Summe der Elemente eines Vektors $x_1 \leftarrow \sum_{i=1}^{|x|} x_i$



Entwicklungswerkzeuge am Beispiel Nvidia CUDA



Spezial- versus Universalhardware

On the Design of Display Processors

T. H. MYER

Bolt Beranek and Newman Inc, Cambridge, Mass.

AND

I. E. SUTHERLAND*

Harvard University, Cambridge, Mass.

"We have built up the display channel until it is itself a general purpose processor with a display.

[...]

In short, we have come exactly once around the wheel of reincarnation."

Fortschritt passiert oft in Kreisläufen mit Wiederentdeckungen.

Myer & Sutherland, *Communications of the ACM*, **11** (6), 1968, S. 412

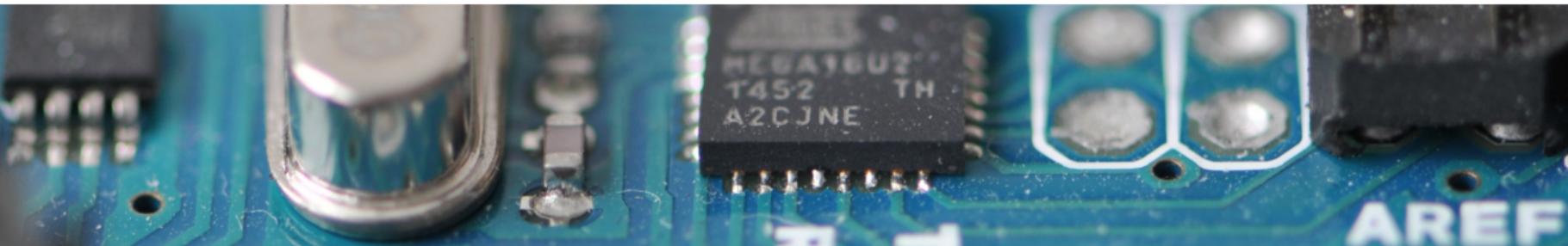
Erinnerung

Nicht vergessen !

Melden Sie sich online bis **spätestens 19.01.2022** zum ersten Klausurtermin an.
Nachmeldungen per E-Mail können wir **nicht** berücksichtigen.

Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Speicher

Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2021/22 · 19. Jänner 2022

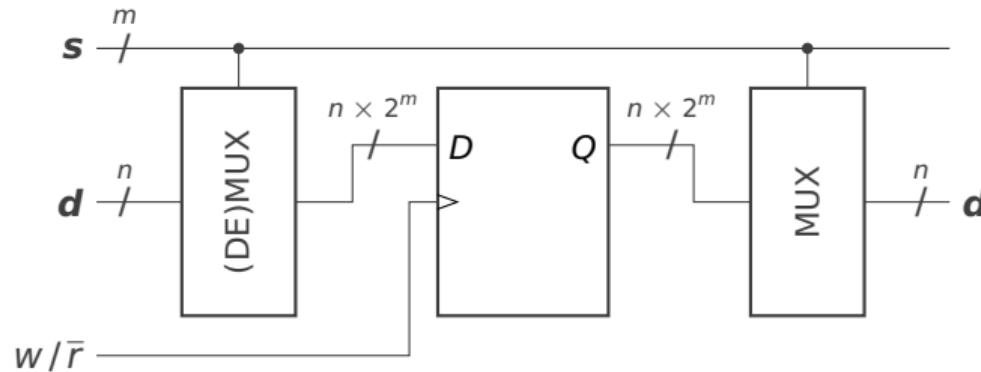
Erinnerung

Nicht vergessen !

Melden Sie sich online bis **spätestens 19.01.2022** zum ersten Klausurtermin an.
Nachmeldungen per E-Mail können wir **nicht** berücksichtigen.

Ausgangspunkt

Wiederholung der Skizze unseres bisherigen Speichermodells:



Problemfelder

- Kosten-Nutzen-Verhältnis
- Persistenz

Technologien zum Speichern von Information

- **Modifikation von Strukturen**

Lochkarte, Schallplatte

- **Magnetismus**

Magnetkernspeicher,
Magnetband, Diskette,
Festplatte

- **Elektrische Ladung**

Kondensator, isoliertes Gatter

- **Rückkopplung**

Flipflop, Schwingkreis

- **Optik**

Barkode, CD-ROM, DVD

Vergleichskriterien

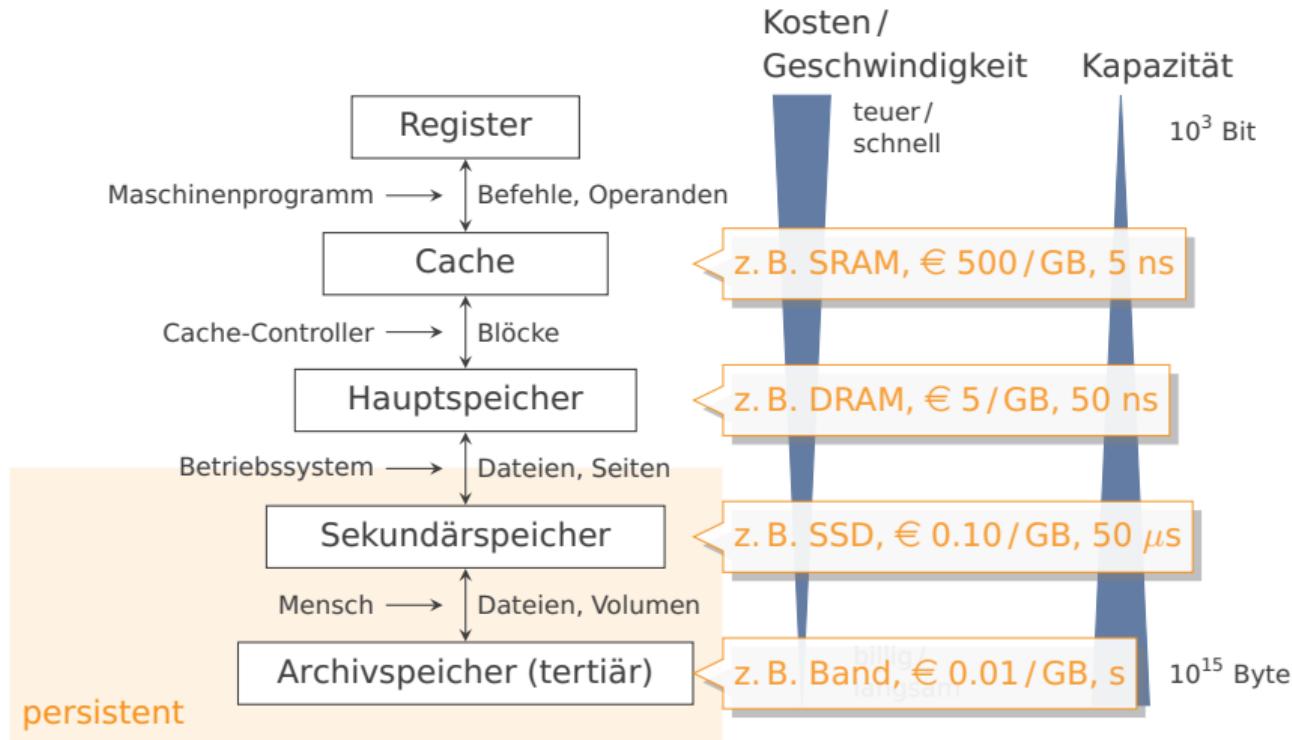
- Persistenz
- Geschwindigkeit
(Zugriff, Übertragung)
- Kapazität
- Dichte
- Energiebedarf
- Robustheit
- Kosten

Es gibt auch **Kombinationen**, z. B. magneto-optisch.

Gliederung heute

- 1. Speicherhierarchie**
2. Aufbau und Ansteuerung von dynamischem RAM
3. Aufbau und Ansteuerung von Flash-Speicher
4. Ausblick

Speicherhierarchie



Quelle für Preise: J. McCallum, <https://jcmit.net>, eigene Rundung und Aktualisierung 2020

Vorsätze für Maßeinheiten

Präfix	Aussprache	Menge dezimal	Menge binär
E	Exa	$(10^3)^6 = 10^{18}$	$(2^{10})^6 = 2^{60}$
P	Peta	$(10^3)^5 = 10^{15}$	$(2^{10})^5 = 2^{50}$
T	Tera	$(10^3)^4 = 10^{12}$	$(2^{10})^4 = 2^{40}$
G	Giga	$(10^3)^3 = 10^9$	$(2^{10})^3 = 2^{30}$
M	Mega	$(10^3)^2 = 10^6$	$(2^{10})^2 = 2^{20}$
k	Kilo	$(10^3)^1 = 10^3 = \textcolor{red}{1\,000}$ $(10^3)^0 = 1$	$(2^{10})^1 = 2^{10} = \textcolor{red}{1\,024}$
m	Milli	$(10^3)^{-1} = 10^{-3}$	
μ	Mikro	$(10^3)^{-2} = 10^{-6}$	
n	Nano	$(10^3)^{-3} = 10^{-9}$	
p	Piko	$(10^3)^{-4} = 10^{-12}$	

Aufgepasst! Kapazitäten von Speicherbausteinen werden immer noch **binär** angegeben.
Bei Speichermedien und in der Datenübertragung ist die **dezimale** Interpretation verbreitet.

Akronyme für flüchtigen Speicher

- **RAM** (*random access memory*): Speicher mit wahlfreiem Zugriff auf beliebige Adressen (früher im Gegensatz zu Bandspeicher)
- **SRAM** (*static . . .*): statisches RAM, am besten vergleichbar mit D-Flipflops, 4–8 Transistoren pro Bit
- **DRAM** (*dynamic . . .*): dynamisches RAM, das Inhalte nach Auslesen und durch Entladung eines Kondensators im Zeitverlauf vergisst, 1 Transistor pro Bit (→ viel dichter und damit billiger als SRAM)
- **SDRAM** (*synchronous DRAM*): Steuerleitungen wirken bei steigender Flanke des **Speichertakts** (i. d. R. niedriger als CPU-Takt)
- **DDR-SDRAM** (*double data rate . . .*): Leistungssteigerung durch Auslesen mehrerer benachbarter Bits pro Zugriff

Alle **flüchtigen Speicher** verlieren ihren Inhalt ohne Stromversorgung.

Akronyme für persistenten Speicher

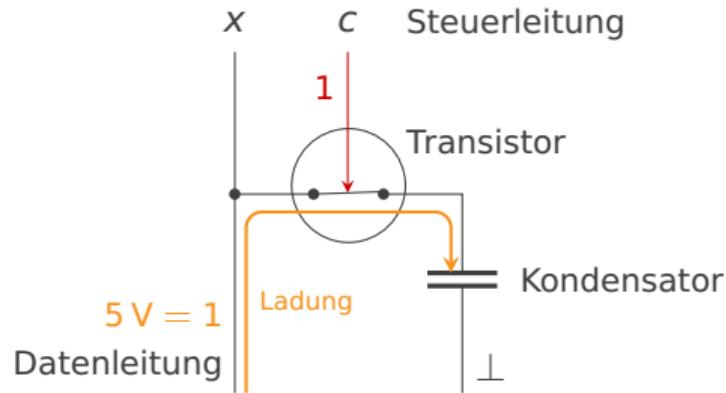
(persistent = nicht flüchtig: Inhalt bleibt ohne Stromversorgung erhalten)

- **ROM** (*read only memory*): nur Lesezugriff
- **PROM** (*programmable ROM*): einmaliger Schreibzugriff
(z. B. durch Durchbrennen von „Sicherungen“ (*fuses*)
bei der Herstellung)
- **EPROM** (*erasable PROM*): elektrisch programmierbares
„brennen“) und durch UV-Licht löschbares PROM
- **EEPROM** (*electrically erasable . . .*): Löschen und
Wiederbeschreiben geschieht elektrisch
(→ heute bekannt als Flash-Speicher in SSDs)
- **HDD** (*hard disk drive*): magnetische Festplatte mit
einem Stapel rotierender Scheiben

Gliederung heute

1. Speicherhierarchie
2. **Aufbau und Ansteuerung von dynamischem RAM**
3. Aufbau und Ansteuerung von Flash-Speicher
4. Ausblick

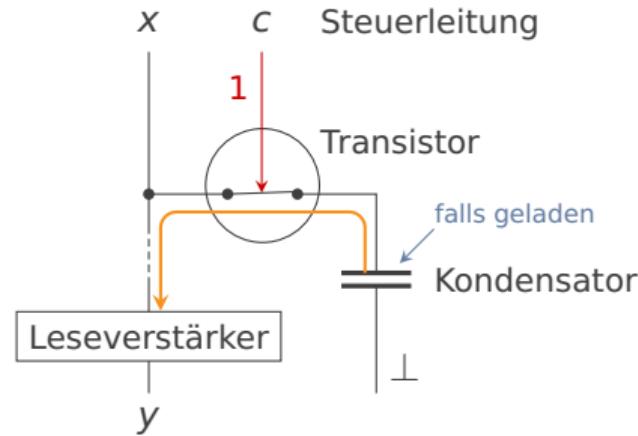
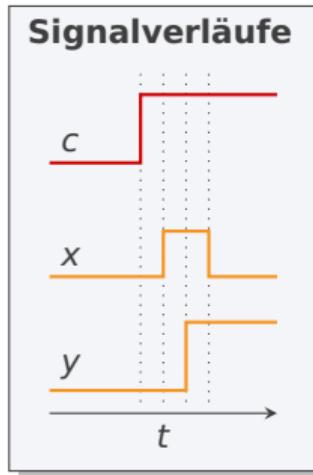
DRAM-Speicherzelle



Schreiben einer DRAM-Zelle

1. Setze die Datenleitung x auf das abzuspeichernde Potenzial.
2. Setze die Steuerleitung $c = 1$, sodass der Transistor leitet.
3. Nach Abschluss der (Ent-)Ladung kann $c = 0$ gesetzt werden.

DRAM-Speicherzelle



Auslesen einer DRAM-Zelle

1. Schalte die Datenleitung an den Eingang eines Leseverstärkers.
2. Setze die Steuerleitung $c = 1$, sodass der Transistor leitet.
3. Schalte ein stabiles Flipflop, falls ein Impuls gemessen wird.

DRAM-Auffrischung

Der Kondensator jeder DRAM-Zelle entlädt sich beim Auslesen und durch Leckströme von selbst.

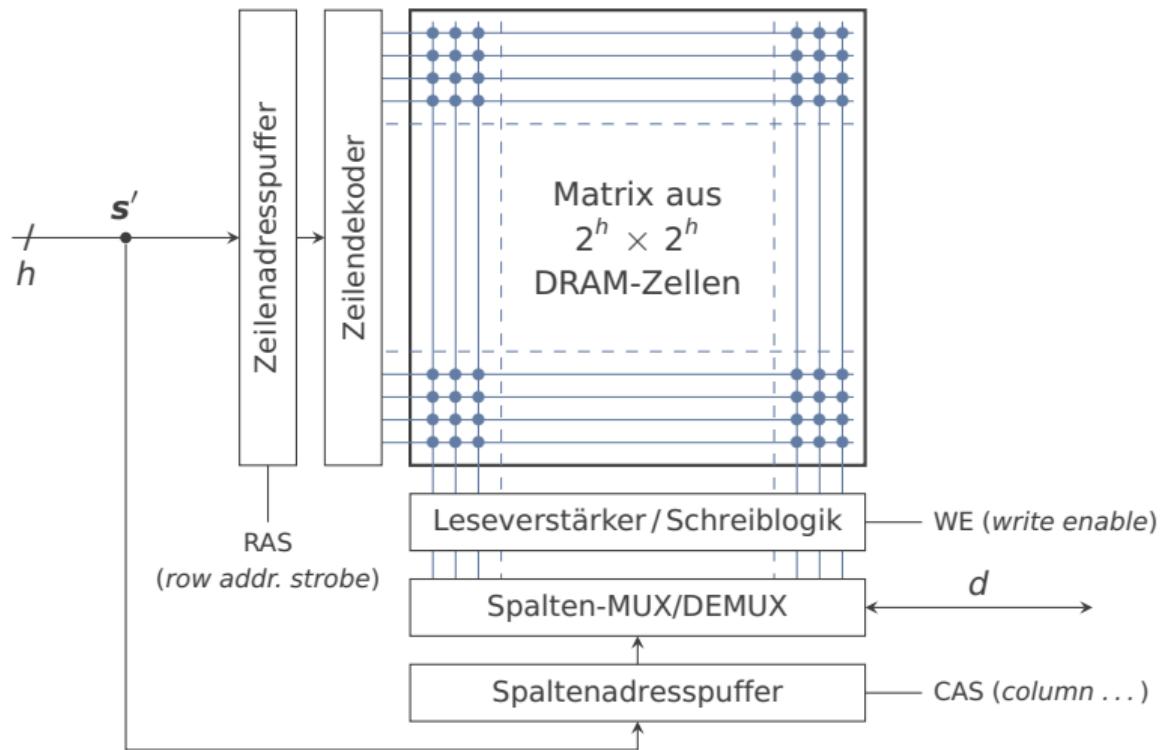
Abhilfe: Nach dem Auslesen und zyklisch ca. alle 30–60 Millisekunden wird der aktuelle Wert erneut geschrieben. Diese Aufgabe übernimmt i. d. R. die Ansteuerungslogik oder ein Speicher-Controller.

DRAM verbraucht deshalb im Betrieb mehr Energie
(und erzeugt mehr Wärme) als SRAM.

Trend: Leckströme und Energieverbrauch mit Spannung senken:

- 2.5 V bei DDR-SDRAM (2000)
- 1.8 V bei DDR2-SDRAM (2004)
- 1.5 V bei DDR3-SDRAM (2007)
- 1.2 V bei DDR4-SDRAM (2012)
- 1.1 V bei DDR5-SDRAM (2020)

Aufbau von DRAM-Bausteinen



Ansteuerung beim Lesezugriff

1. WE deaktivieren.
2. Anlegen der **Zeilenadresse** und Übernahmen in den Zeilenadresspuffer durch Aktivierung des Steuersignals RAS.
→ Bitvektor der aktivierte Zeile liegt am Leseverstärker an.
3. Anlegen der **Spaltenadresse** und Übernahmen in den Spaltenadresspuffer durch Aktivierung des Steuersignals CAS.
→ Der Spaltenmultiplexer gibt das gewählte Bit aus.

Beschleunigung bei Mehrfachzugriff

- Neue Spaltenadresse anlegen (Fast Page Mode, FPM), ca. 2×
- Spaltenadresse automatisch hochzählen (Burst-Modus), ca. 16×

4. WE aktivieren, um Bitvektor zurückzuschreiben.

Ansteuerung beim Schreibzugriff

1. Anlegen der **Zeilenadresse** und Übernahmen in den Zeilenadresspuffer durch Aktivierung des Steuersignals RAS.
→ Bitvektor der aktivierte Zeile liegt am Leseverstärker an.
2. Anlegen der **Spaltenadresse** und Übernahmen in den Spaltenadresspuffer durch Aktivierung des Steuersignals CAS und WE aktivieren.
→ Der Spaltendemultiplexer fügt Bit *d* in die Zeile ein.
→ Die Zeile wird zurückgeschrieben.

Erweiterungen zur Erhöhung der Zuverlässigkeit

Behandlung von Fertigungs- und Umgebungseinflüssen

- **Kodierung** zur Fehlererkennung und Fehlerkorrektur
- z. B. zusätzliche Paritätsbits (binäre Quersumme), Kreuzsicherung, Schreiben unter Berücksichtigung von defekten Zellen, ...

Vertiefung in **Rechnernetze und Internettechnik**, Pflichtmodul, 3. Semester

Behandlung von Fehlern der Ansteuerung (insb. Programmierung)

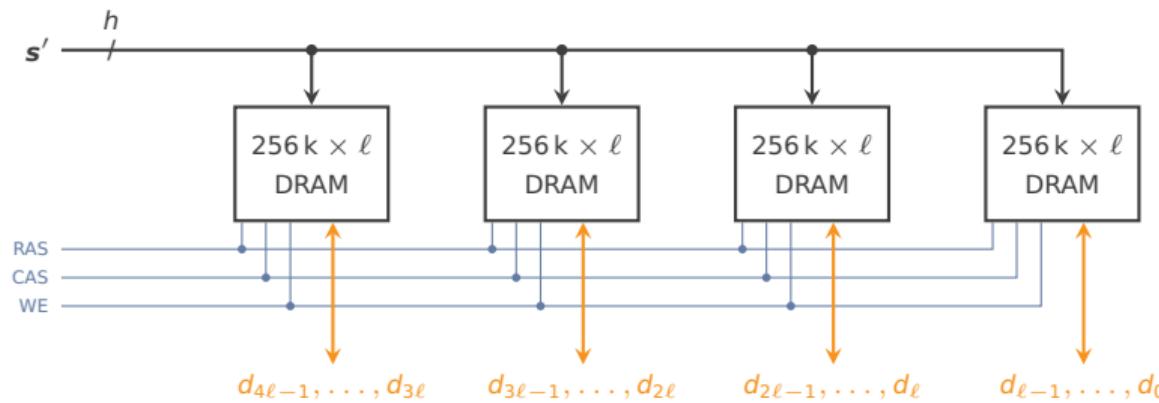
- **Speicherschutz**: CPU prüft Adressbus vor Zugriff.
- Ausnahmebehandlung bei Verletzung
- Berechtigungen werden im privilegierten Modus (d. h. vom Betriebssystem) vergeben.

Vertiefung in **Betriebssysteme**, Pflichtmodul, 2. Semester

Hauptspeicher aus mehreren DRAM-Bausteinen

Beispiel: $h = 9 \Rightarrow 2^{18} = 256 \text{ k} \times \ell$ Bit pro Baustein

Variante 1: Erweiterung der Wortbreite

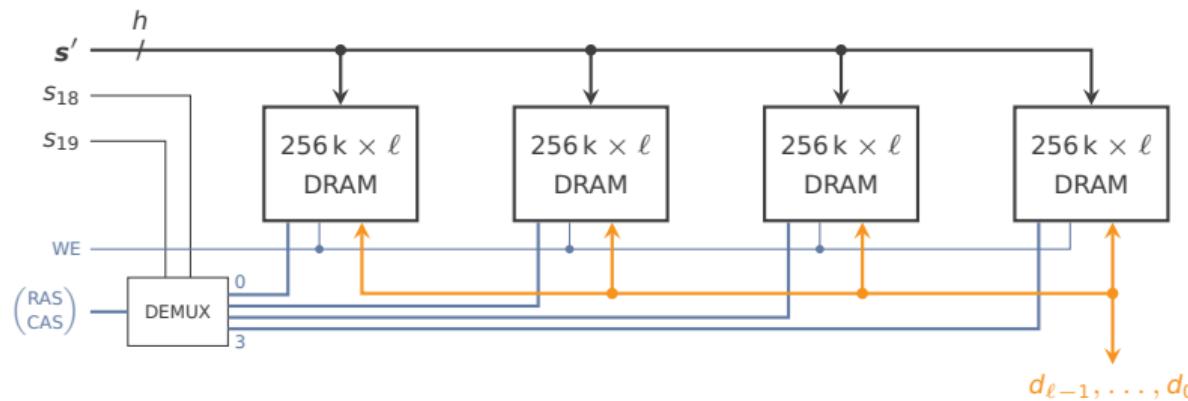


Gemeinsame Adress- und Steuerleitungen; getrennte Datenleitungen

Hauptspeicher aus mehreren DRAM-Bausteinen

Beispiel: $h = 9 \Rightarrow 2^{18} = 256 \text{ k} \times \ell$ Bit pro Baustein

Variante 2: Erweiterung des Adressraums



Gemeinsame Datenleitungen; Auswahl des Bausteins durch Schaltung der Steuersignale über die höchstwertigen Adressleitungen

Hörsaalfragen



24 82 94 16

Es sei $\ell = 8$ Bit.

1. In welchem DRAM-Baustein wird das vierte Bit der Adresse 0x020000 bei
Variante 1 (Erweiterung der Wortbreite) gespeichert ?
2. In welchem DRAM-Baustein wird das vierte Bit der Adresse 0x020000 bei
Variante 2 (Erweiterung des Adressraums) gespeichert ?
 - a. erster Baustein (von links)
 - b. zweiter Baustein
 - c. dritter Baustein
 - d. vierter Baustein (ganz rechts)

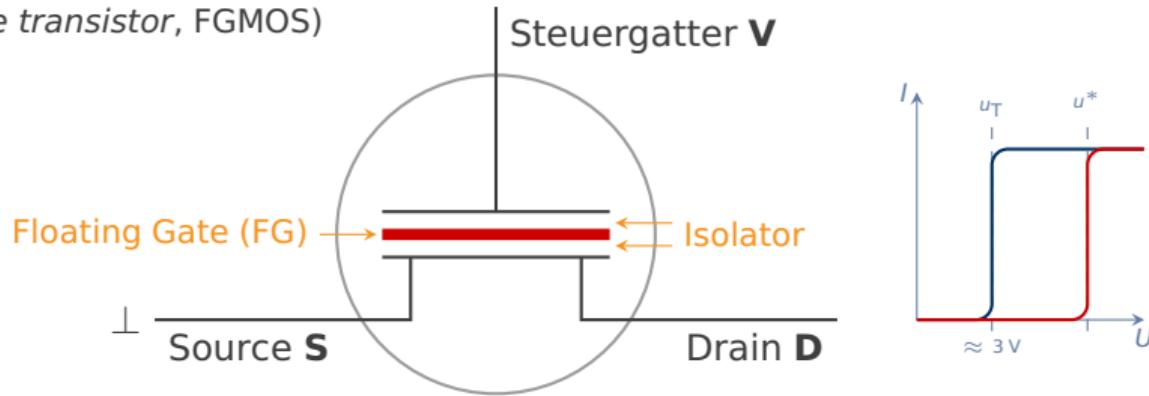
Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Gliederung heute

1. Speicherhierarchie
2. Aufbau und Ansteuerung von dynamischem RAM
- 3. Aufbau und Ansteuerung von Flash-Speicher**
4. Ausblick

Feldeffekttransistor mit isoliertem Gatter

(engl. *floating gate transistor*, FGMOS)



- Im Floating Gate können Ladungsträger durch quantenmechanische Tunneleffekte „eingesperrt“ werden. Dazu sind Spannungen $\approx 10\text{ V}$ nötig.
- Wenn das FG nicht geladen ist, reagiert der FGMOS wie ein Transistor: Er leitet, wenn die Spannung am Steuergatter größer als u_T ist.
- Ein geladenes FG schirmt das Steuergatter ab und verschiebt den Arbeitspunkt des Transistors auf höhere Spannungen u^* .
- Im Zustand des FG lässt sich (mindestens) ein Bit speichern.

Ansteuerung eines FGMOS

Zustands(übergangs)tabelle

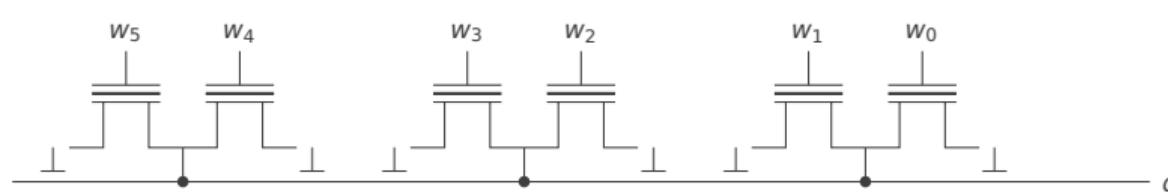
Steuergatter V	Drain D	FG-Ladung	Transistor	Zustand
Lesezugriff				
$u_T < 3.3\text{ V}$	$> \perp$	ungeladen	leitet	logisch 1
$u_T < 3.3\text{ V} < u^*$	$> \perp$	negativ geladen	sperrt	logisch 0
	$> u^*$	$> \perp$	negativ geladen	leitet
Schreib- und Löschzugriff				
$> 10\text{ V}$	\perp	steigt		$\rightarrow 0$
\perp	$> 10\text{ V}$	fällt		$\rightarrow 1$

Aufbau von Flash-Bausteinen

NOR-Flash

word lines

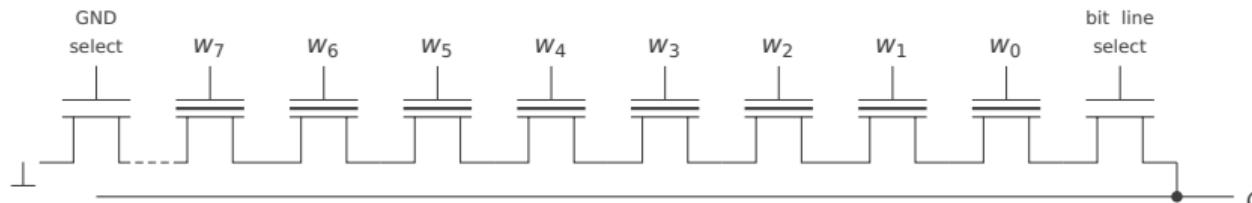
Intel seit 1988



Die Datenleitung d geht auf 0, wenn an mindestens einer word line w_i eine Spannung über dem Arbeitspunkt des Feldeffekttransistoren anliegt.

NAND-Flash

Toshiba seit 1989



Die bit line geht auf 0, wenn an allen word lines w_i eine Spannung über den jeweiligen Arbeitspunkten der Feldeffekttransistoren anliegt.

Vergleich von Flash-Technologien

NOR-Flash

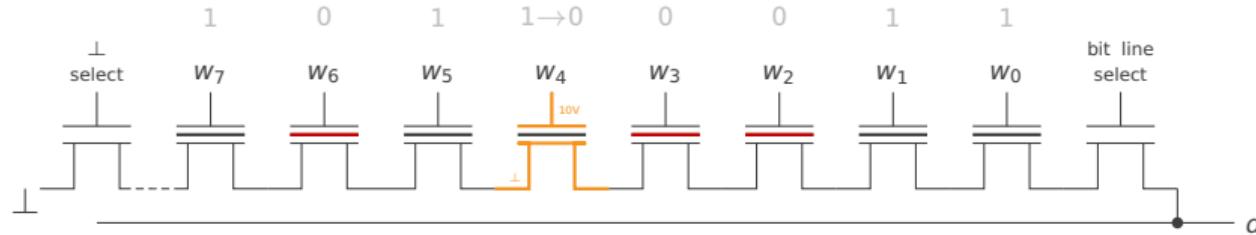
- + adressierbar wie RAM
- + wahlfreies Lesen und Schreiben
- + i. d. R. fehlerfrei
- Schreiben nur langsam
- hoher Energieverbrauch
- geringe Speicherdichte
- teuer (pro Bit)
- geeignet für Programmkode

NAND-Flash

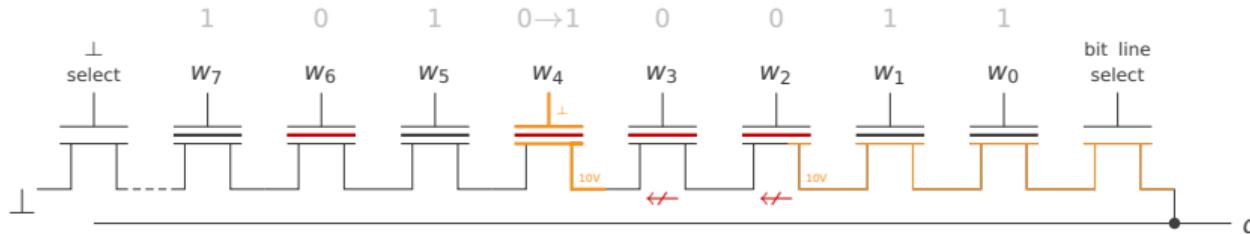
- + billig (pro Bit)
- + hohe Speicherdichte
- + schnell
- + wahlfreies Lesen
- Löschen nur blockweise
- komplizierte Ansteuerung
- Produktionsfehler nicht vernachlässigbar
- Abnutzung
- geeignet für Dateisysteme

Schreiben und Löschen bei NAND-Flash

NAND-Flash schreiben



NAND-Flash löschen

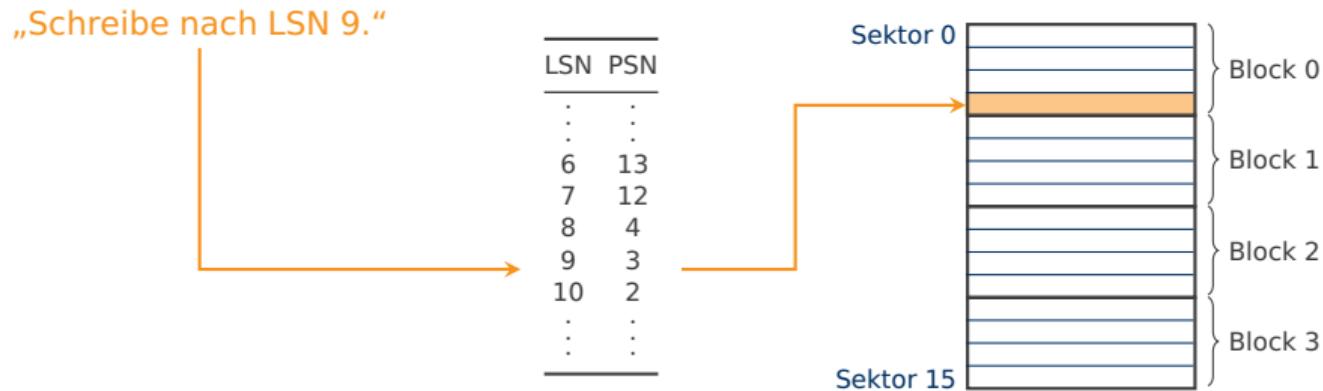


Flash Translation Layer (FTL)

Verbergen der **Nachteile von NAND-Flash** in einer Steuerschicht:

1. Übersetzt logische (L^*N) in physische (P^*N) Adressen
2. Gleichmäßige Verteilung der Löschoperationen (*wear leveling*)
3. Ausschluss fehlerhafter Blöcke (ggf. Fehlerkorrektur)

Variante 1: Sektor-Mapping



Neue Anforderung: Konsistenz der Verwaltungsdaten bei Stromausfall

Flash Translation Layer (FTL)

Verbergen der **Nachteile von NAND-Flash** in einer Steuerschicht:

1. Übersetzt logische (L^*N) in physische (P^*N) Adressen
2. Gleichmäßige Verteilung der Löschoperationen (*wear leveling*)
3. Ausschluss fehlerhafter Blöcke (ggf. Fehlerkorrektur)

Variante 2: Block-Mapping

„Schreibe nach LSN 9.“

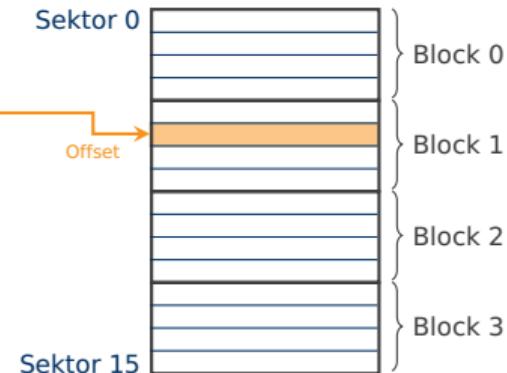


$$LSN = 4 \times LBN + \text{Offset}$$

$$\Rightarrow LBN = 2, \text{Offset} = 1$$



LBN	PBN
0	2
1	3
2	1
3	0



Neue Anforderung: Konsistenz der Verwaltungsdaten bei Stromausfall

Flash Translation Layer (FTL)

Verbergen der **Nachteile von NAND-Flash** in einer Steuerschicht:

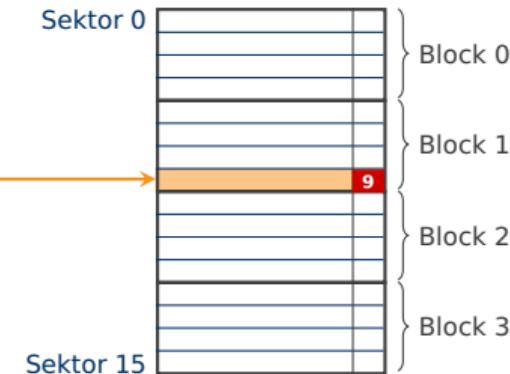
1. Übersetzt logische (L^*N) in physische (P^*N) Adressen
2. Gleichmäßige Verteilung der Löschoperationen (*wear leveling*)
3. Ausschluss fehlerhafter Blöcke (ggf. Fehlerkorrektur)

Variante 3: Hybrides Mapping

„Schreibe nach LSN 9.“

$$LBN = \lfloor LSN : 4 \rfloor = 2$$

LBN	PBN
0	2
1	3
2	1
3	0



Neue Anforderung: Konsistenz der Verwaltungsdaten bei Stromausfall

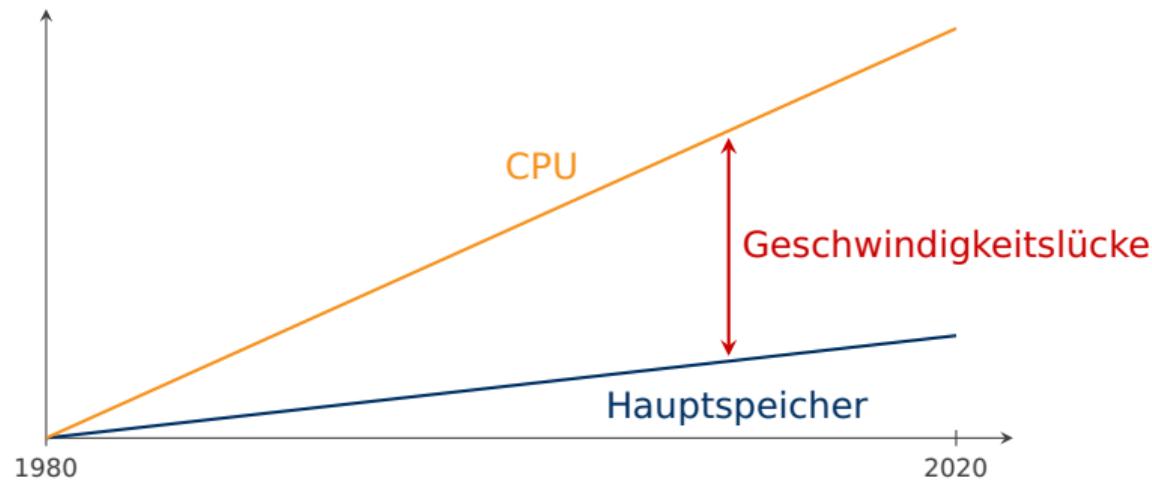
Gliederung heute

- 1. Speicherhierarchie**
- 2. Aufbau und Ansteuerung von dynamischem RAM**
- 3. Aufbau und Ansteuerung von Flash-Speicher**
- 4. Ausblick**

Betrachtung der Systemleistung

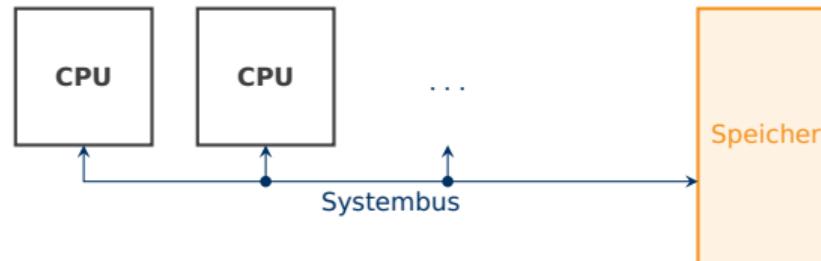
Seit 1980 wächst die Geschwindigkeit des ...

- Hauptspeichers um 7 % pro Jahr
- Prozessors um 50 % pro Jahr

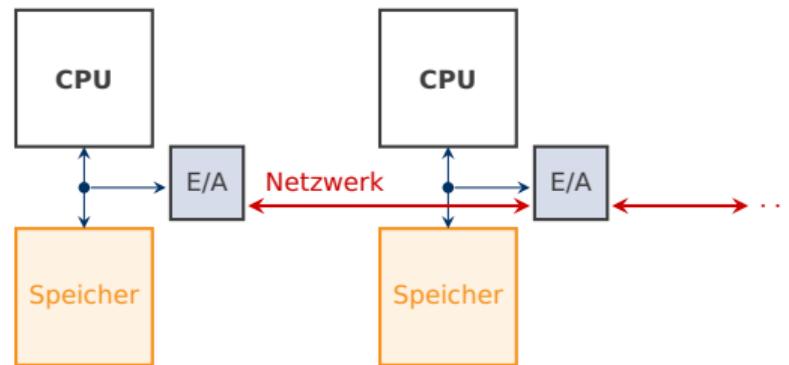


Speicherarchitekturen für Mehrprozessorsysteme

Shared Memory („gemeinsam genutzt“)

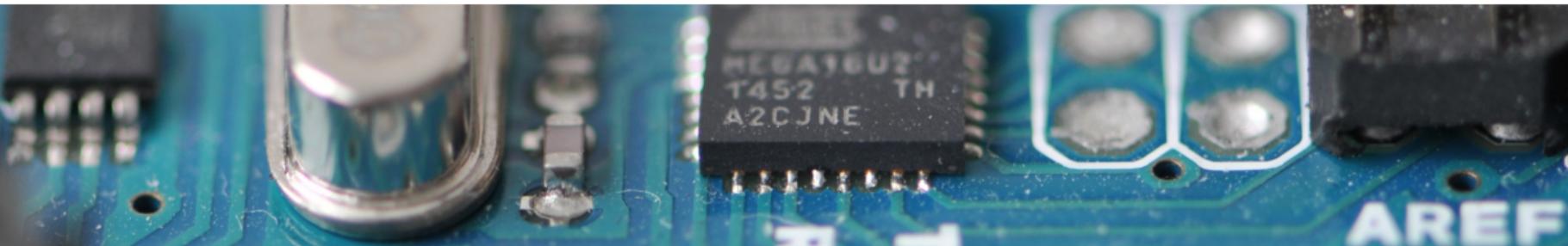


Distributed Memory („verteilt“)



Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)



Rechnerarchitektur

Leistung

Univ.-Prof. Dr.-Ing. Rainer Böhme

Wintersemester 2021/22 · 26. Jänner 2022

Gliederung

- 1. Leistungsmessung**
2. Caches: Optimierung des Speicherzugriffs
3. Pipelines: Optimierung der CPU-Nutzung

Leistung

Allgemeine Definition aus der **Physik**:

$$\text{Leistung} = \frac{\text{Arbeit}}{\text{Zeit}}$$

Arbeit in der **Informatik**: i. d. R. Rechenoperationen

- Instruktionen allgemein → MIPS (*million instructions per second*)
- Integer-Operationen → MOPS (*million operations per second*)
- Gleitkommaoperationen → FLOPS (*floating point operations ...*)
- ...

Die Einheiten lassen sich nur sehr begrenzt ineinander umrechnen.

Leistungsverhältnis

Bei **konstanter Arbeit:**

$$\text{Beschleunigung } S = \frac{\text{Ausführungszeit unter Referenzbedingungen}}{\text{Ausführungszeit unter Testbedingungen}}$$

Beispiele

1. **Referenzbedingung:** Smartphone aus dem Jahr 2018
Testbedingung: Smartphone von heute
 $S = 2$ bedeutet, das neuere Smartphone ist doppelt so leistungsfähig.

2. **Referenzbedingung:** Bubble-Sort-Algorithmus auf Intel 8086
Testbedingung: Quicksort-Algorithmus auf Intel 8086
 $S = 4$ bedeutet, Quicksort ist (für die gewählten Daten, insb. Datenmenge) viermal schneller.
Grund: weniger Instruktionen

Amdahlsches Gesetz

Annahme Anteil α (z. B. in %) eines Programms lässt sich um Faktor c beschleunigen

Es gilt:

$$S = \frac{t}{(1 - \alpha) \cdot t + \alpha \cdot \frac{t}{c}} = \frac{1}{(1 - \alpha) + \frac{\alpha}{c}}$$

Beispiel

- 20 % der Ausführungszeit wird für Divisionen benötigt.
- Spezielle Hardware (z. B. Koprozessor) beschleunigt die Division um den Faktor $c = 10$.
- Rechnerische Beschleunigung $S = 1.219512$

Amdahl 1967 (für Parallelisierung)

Messung der Ausführungszeit

Verwendung von **Hardwarezählern** zur Messung der CPU-Zyklen

- Zugriff bei ARM über Koprozessor 15, Mnemonics MRC und MCR*
- Berechnung der Ausführungszeit über Zykluszeit Δt bzw. Taktfrequenz $1/\Delta t$.
- Unterscheidung **Gesamtzeit** (inkl. Interrupts, Betriebssystem, Festplattenzugriffe) und **CPU**-Zeit (ohne Wartezeit auf E/A-Geräte)

Einflussfaktoren auf die Leistung eines Programms

Implementierung

- Algorithmus
- Programmiersprache
- Compiler
- Optimierungsstufe

Ausführungsumgebung

- Befehlssatzarchitektur
- Mikroarchitektur
- Betriebssystem
- sonstige Hardware

* in dieser Vorlesung nicht behandelt

Leistung und Leistungsabruf

- **Maximale Leistung** (*peak performance*)
Theoretisches Maximum, unter Idealbedingungen
kurzfristig erreichbar
- **Durchschnittliche Leistung** (*average performance*)
Erwartungswert bei typischen Aufgaben
- **„Nachhaltige“ Leistung** (*sustained performance*)
Unterschranke für die über den gesamten Lebenszyklus
versprochene Leistung eines Gesamtsystems
oder von Komponenten mit Verschleißeffekten (z. B. Flash-Speicher)

Methodische Schwierigkeiten bei der Leistungsmessung

Beim **Vergleich von Implementierungen** auf definierter Architektur:
(d. h. Hardware und Systemsoftware)

- Granulare Messung einzelner Programmteile
→ Messfehler durch künstliche Aufrufbedingungen
(Rückwirkungsabweichung)
- Grobe Messung ganzer Programmläufe
→ Messfehler durch Umgebungseinflüsse
(z. B. Unterbrechungsanforderungen)

Beim **Vergleich von Architekturen**:

- Wahl vergleichbarer Aufgaben und Implementierungen
- Bei Datenabhängigkeit: Wahl der Testdaten

Weitere Unschärfe durch die Verbreitung von **Virtualisierungstechniken**

Benchmarks

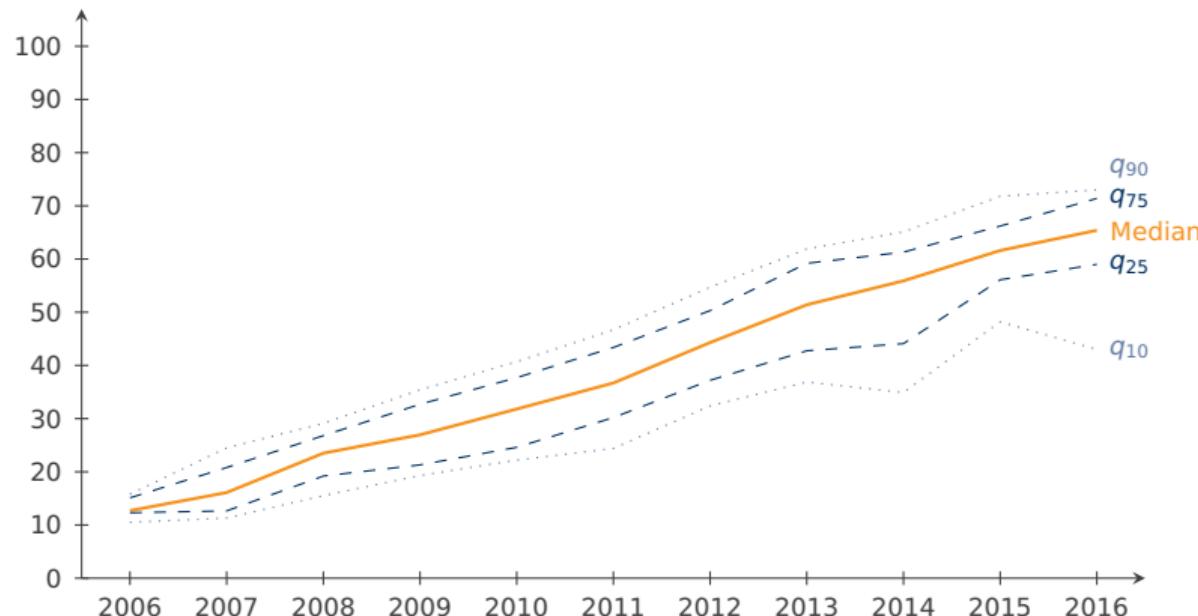
Spezialsoftware zur Leistungsmessung, simuliert typische Arbeitslast

- **Synthetische Benchmarks** messen Instruktionen pro Sekunde: z. B. **Whetstone** für Gleitkomma-, **Dhrystone** für Ganzzahloperationen
- **Kernels** sind ausgewählte Algorithmen, die sich als Benchmark etabliert haben: z. B. **LINPACK**
- **Mikrobenchmarks** sind auf die Messung einzelner Komponenten (z. B. Speicheranbindung, Grafik) spezialisiert
- **Benchmark-Suiten** enthalten verschiedene, gut portierbare Programme aus unterschiedlichen Anwendungsgebieten: z. B. **Standard Performance Evaluation Corporation (SPEC)** misst Leistungsverhältnis (aktuell in der Version 2017)

Leistungsentwicklung

SPEC 2006 Integer-Benchmarks

(8131 dokumentierte Tests zwischen 2006 und 2016)

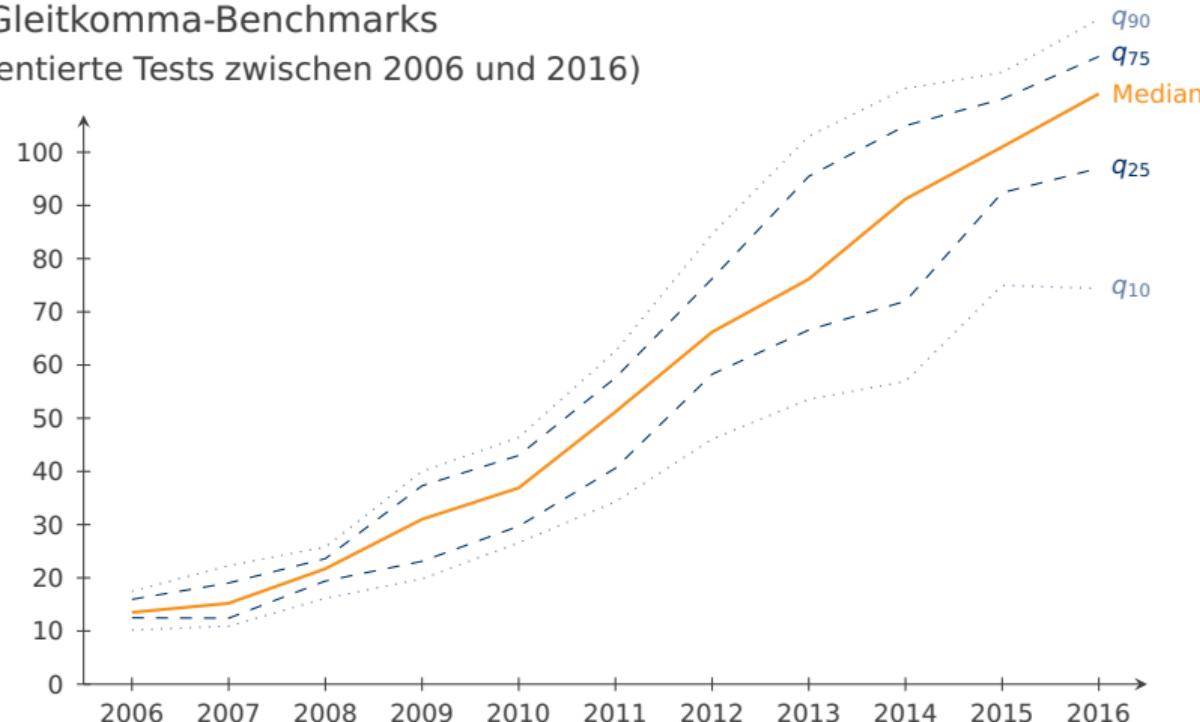


Daten von www.spec.org/cpu2006/results

Leistungsentwicklung

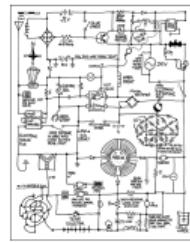
SPEC 2006 Gleitkomma-Benchmarks

(7972 dokumentierte Tests zwischen 2006 und 2016)



Daten von www.spec.org/cpu2006/results

Grundsätzliche Schwierigkeiten bei der Leistungsmessung komplexer Systeme mit einfachen Kennzahlen



Projektion

42

Reduzierung von Komplexität

Realität

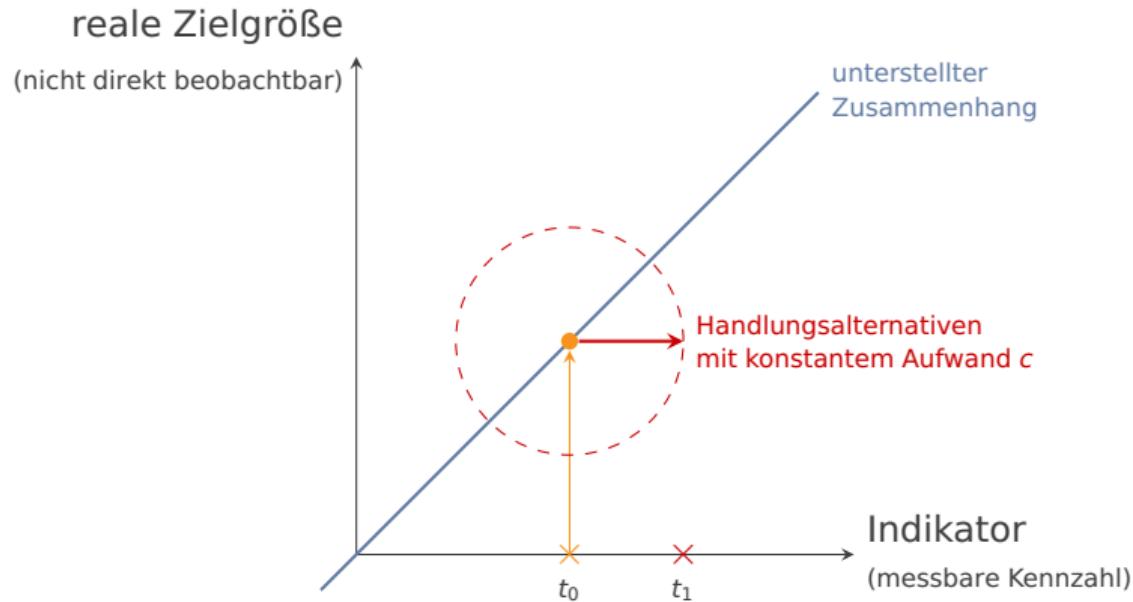
Kennzahl

Reaktion

rationales Verhalten mit Kenntnis des Messverfahrens

vgl. z. B. Campbell 1976, Goodhart 1981; Illustration: xkcd.com

Rationale Reaktion: Überanpassung



Folge: Kennzahlen verlieren Aussagekraft, Verhaltensanreize verfehlten Ziel

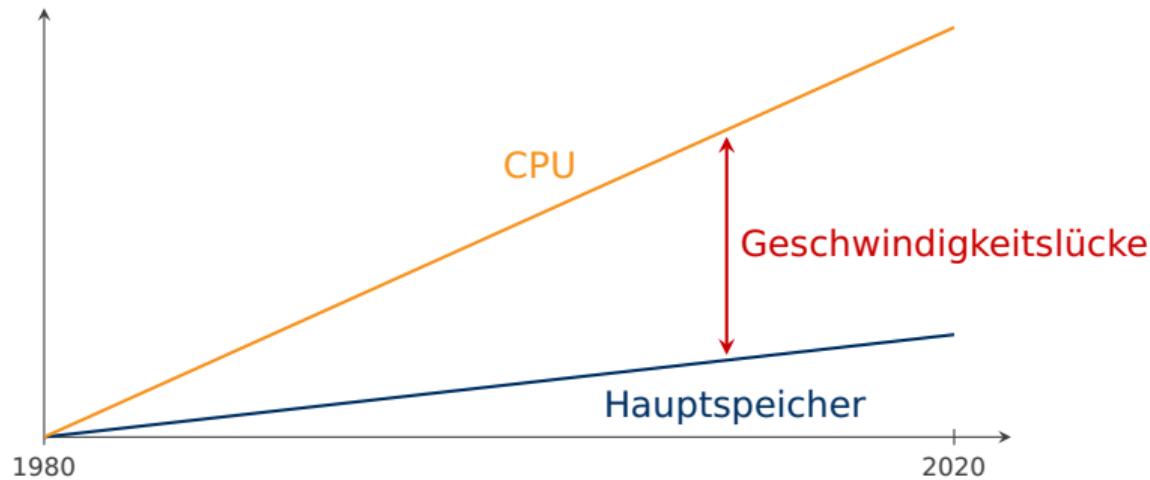
Gliederung

1. Leistungsmessung
2. **Caches: Optimierung des Speicherzugriffs**
3. Pipelines: Optimierung der CPU-Nutzung

Geschwindigkeitslücke (W)

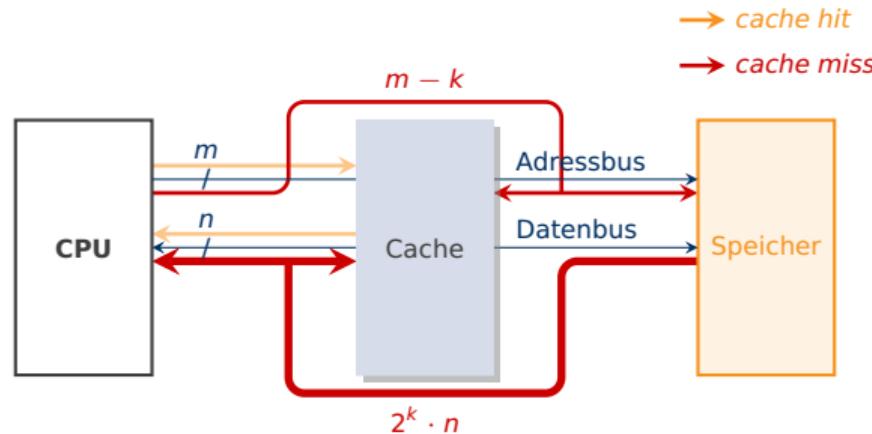
Seit 1980 wächst die Geschwindigkeit des ...

- Hauptspeichers um 7 % pro Jahr
- Prozessors um 50 % pro Jahr



Anbindung des Cache an CPU und Speicher

Der Cache ist ein kleiner, schneller, mit SRAM realisierter Pufferspeicher.

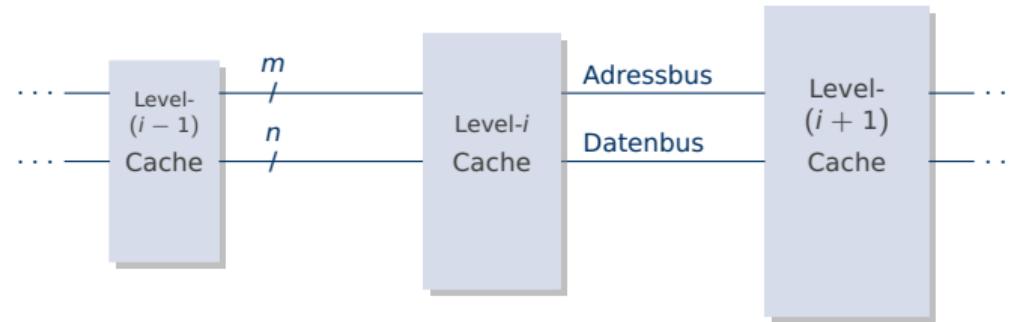


Vision

- Vorhersage, auf welche Speicherzelle die CPU als nächstes zugreift
- Daten schon im Vorfeld im Cache ablegen (*read ahead*)

Cache-Hierarchie

Ein oder mehrere Cache-Ebenen bleiben für die CPU **transparent**.



Lokalitätsprinzip

Ziel: Vorhersage der Adressen von wahrscheinlichen Lesezugriffen

- **Zeitliche Lokalität:** Hohe Wahrscheinlichkeit, dass eine Speicheradresse wiederholt verwendet wird (z. B. bei Schleifen)
→ Verwendete Adressen und Inhalte puffern.
- **Räumliche Lokalität:** Hohe Wahrscheinlichkeit, dass auf benachbarte Elemente zugegriffen wird (z. B. bei Arrays)
→ Alle benachbarten Speicherworte lesen und puffern.
(Übertragung von Blöcken aus 2^k Worten im DRAM-Burst-Modus.)

Leistungssteigerung durch Cache-Einsatz

Trefferrate

$$h = \frac{\text{Anzahl } \textcolor{orange}{\text{cache hits}}}{\text{Anzahl } \textcolor{orange}{\text{cache hits}} + \text{Anzahl } \textcolor{red}{\text{cache misses}}} \quad (1)$$

Sei

- c die Zugriffszeit auf den Cache z. B. 5 ns
- r die Zugriffszeit auf den Hauptspeicher z. B. 50 ns

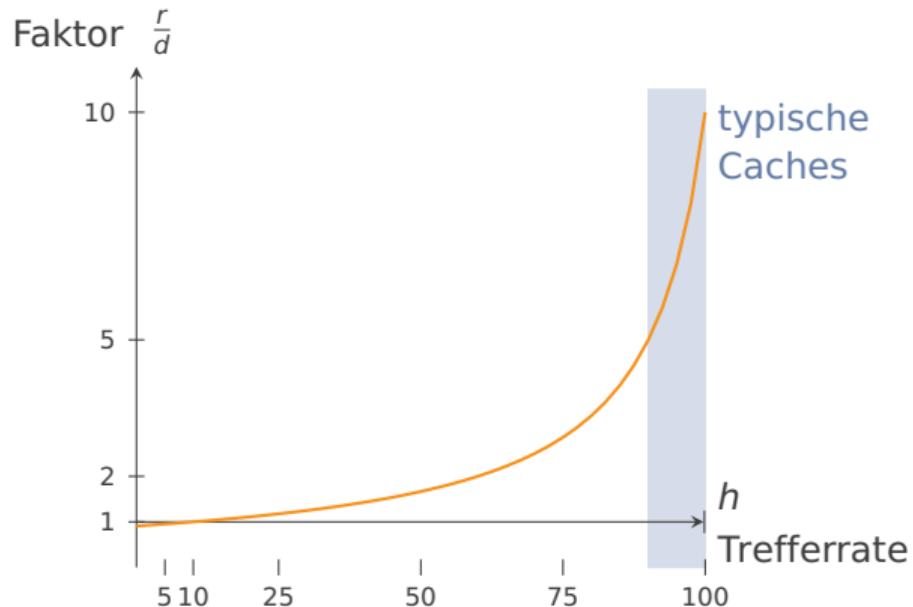
dann ist die **mittlere Speicherzugriffszeit**

$$d = c + (1 - h) \cdot r . \quad (2)$$

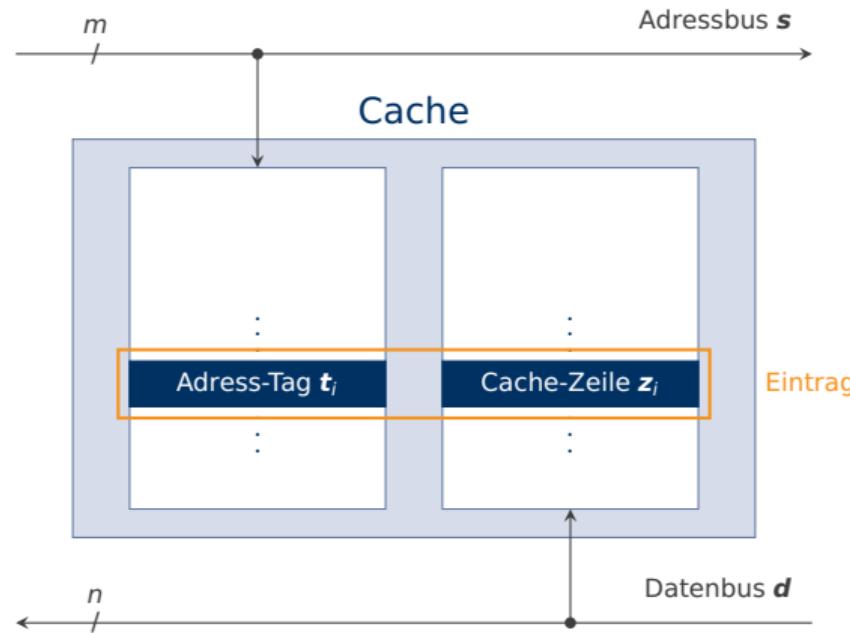
Nachteil: Schlechtere Vorhersagbarkeit des exakten Zeitverhaltens

Grafische Darstellung

der Leistungssteigerung durch Cache-Einsatz



Aufbau eines Caches



Cache-Verwaltung

Reinform 1: vollassoziativ

- Jeder Block des Hauptspeichers kann in jeder beliebigen Cache-Zeile abgespeichert werden.

$$\mathbf{z} \in \{0,1\}^{n \cdot 2^k}, k \in \mathbb{N} \Rightarrow \mathbf{t} \in \{0,1\}^{(m-k)}$$

- Verschiedene Verdrängungsstrategien beim Schreibzugriff
- Lesezugriff erfordert Vergleich mit allen Tags → **aufwändig**

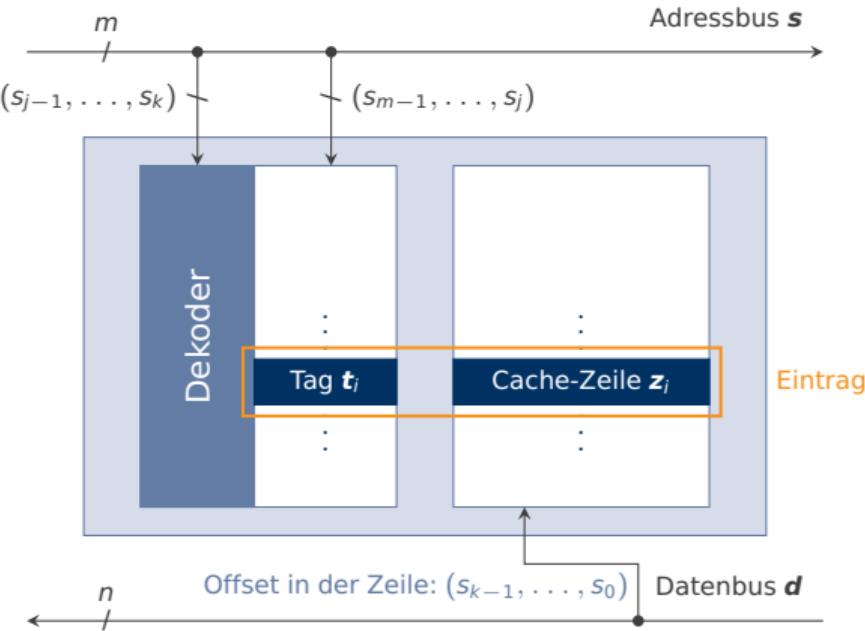
Reinform 2: direkt abgebildet (engl. *direct mapped*, DM)

- Die j niederwertigsten Adressbits definieren eine Cache-Zeile für jeden Block des Hauptspeichers. $j > k \Rightarrow \mathbf{t} \in \{0,1\}^{(m-j)}$
- Beim Lesezugriff Dekodierung und ein Vergleich

Mischformen z. B. zweifach assoziativer Cache

- Jeder Block des Hauptspeichers kann in einer von zwei definierten Cache-Zeilen abgespeichert werden.

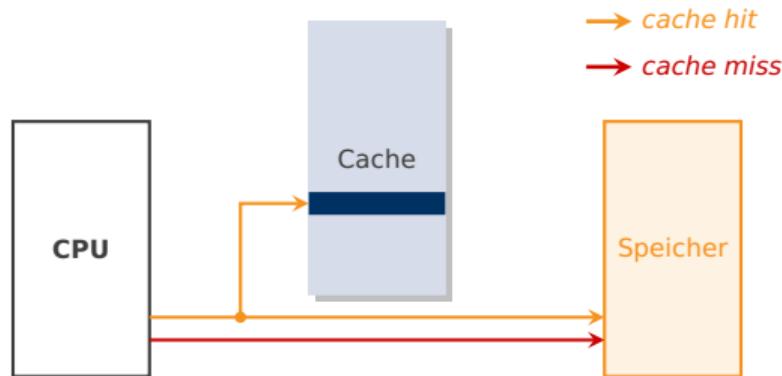
Aufbau eines Caches



Cache-Kohärenz

Sicherung der Konsistenz von Speicher und Cache nach **Schreibzugriff**

Write through-Methode

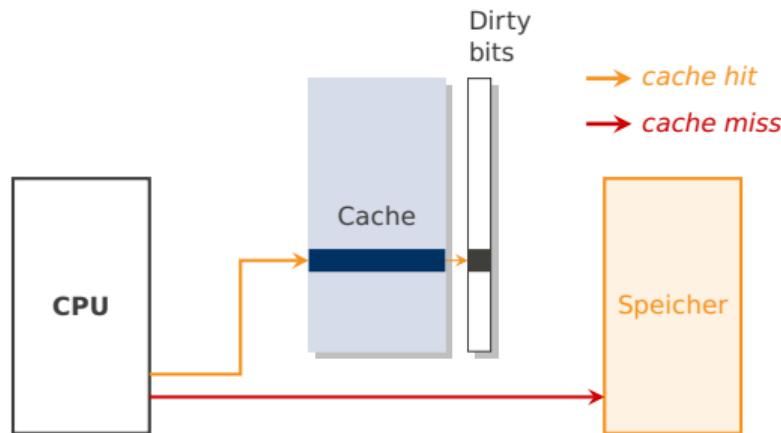


Sofortige Aktualisierung → keine Inkonsistenz

Cache-Kohärenz

Sicherung der Konsistenz von Speicher und Cache nach **Schreibzugriff**

Write back-Methode

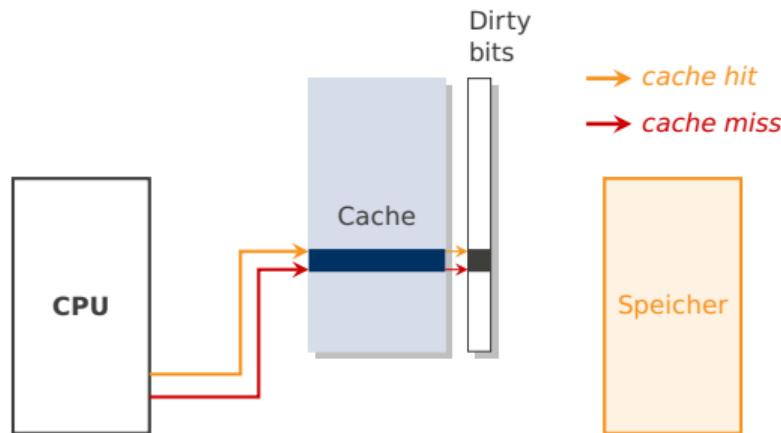


Bei einem Treffer wird nur der Cache aktualisiert. *Dirty bit* zeigt an, dass die Zeile bei Verdrängung zurückgeschrieben werden muss.

Cache-Kohärenz

Sicherung der Konsistenz von Speicher und Cache nach **Schreibzugriff**

Write allocation-Methode



Jeder Schreibzugriff wird zunächst im Cache gepuffert. Es kann zur Verdrängung beim Schreiben kommen.

Hörsaalfrage



24 82 94 16

Welche Cache-Architektur verspricht mehr Leistung ?

- a. **Cache 1:** 64 KB vollassoziativer, in die CPU integrierter SRAM-Cache mit Trefferrate $h = 92\%$ und Zugriffszeit 3 ns.
Länge der Cache-Zeile: 16 Byte.
- b. **Cache 2:** 4 MB direkt abgebildeter SRAM-Cache zwischen CPU und Hauptspeicher mit Trefferrate $h = 95\%$ und Zugriffszeit 12 ns. Länge der Cache-Zeile: 512 Byte.

In beiden Fällen besteht der Hauptspeicher aus schnellem DRAM mit Zugriffszeit 40 ns.

Zugang: <https://arsnova.uibk.ac.at> mit Zugangsschlüssel **24 82 94 16**. Oder scannen Sie den QR-Kode.

Ansätze zur Cache-„optimierten“ Programmierung

Ursachen für Cache-Fehlzugriffe (*cache misses*)

- Erstbelegung nach Programmstart → *compulsory*
- Verdrängung benötigter Zeilen mangels Kapazität → *capacity*
- Verdrängung benötigter Zeilen durch Konflikte → *conflict*

Maßnahmen zur Erhöhung der Trefferrate

1. Rechtzeitiges Holen durch explizite **Prefetch-Anweisungen** (durch Compiler oder manuell im Assemblerprogramm)
2. **Vermeidung von Konflikten** durch Füllworte (*padding*, oft durch Compiler) und Wahl der Dimensionen von Feldern in Zweierpotenzen
3. **Erhöhung der Lokalität** beim Zugriff auf Daten

Beispiele in C

Datenlayout

Vorher

```
1 | char *name[1000];  
2 | int matrikelnr[1000];
```

Nachher

```
3 | struct student {  
4 |     char *name;  
5 |     int matrikelnr;  
6 | } hoerer[1024];
```

Schleifenaustausch

Vorher

```
7 | for (j=0;j<100;j=j+1)  
8 |     for (i=0;i<5000;i=i+1)  
9 |         x[i][j] = 2*x[i][j];
```

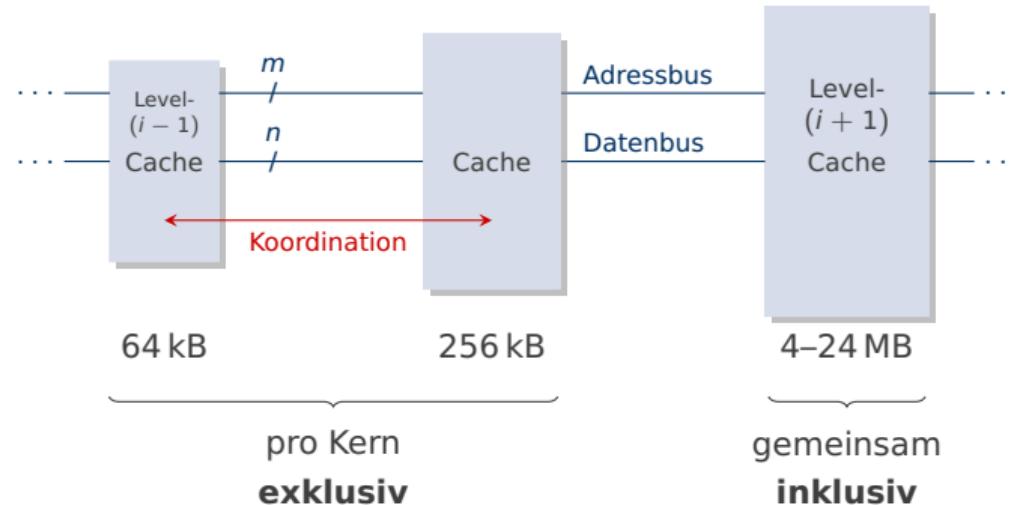
Nachher

```
10 | for (i=0;i<5000;i=i+1)  
11 |     for (j=0;j<100;j=j+1)  
12 |         x[i][j] = 2*x[i][j];
```

Cache-Hierarchie

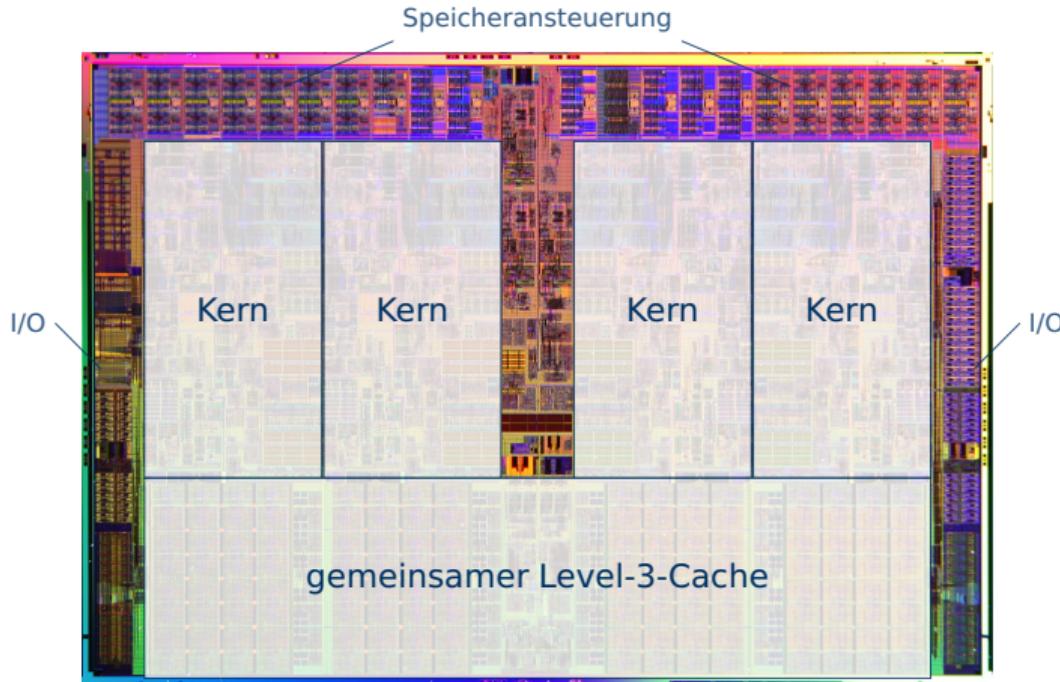
Ein oder mehrere Cache-Ebenen bleiben für die CPU **transparent**.

Beispiel: Intel Nehalem-Mikroarchitektur (2008)



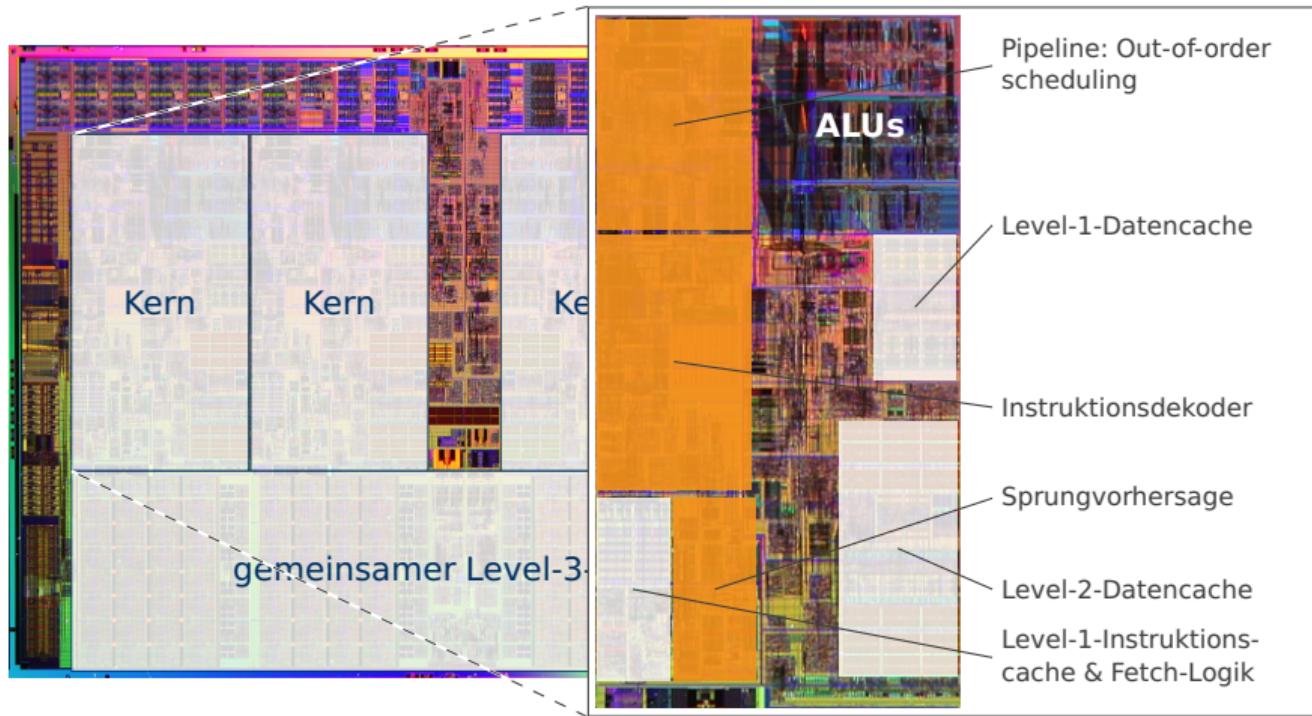
- keine Duplikate
- „Absinken“ bei Verdrängung
- unabhängig

Nutzung der Die-Fläche bei einem Mehrkernprozessor



nach Hennessy & Patterson 2012, S. 29; Bild: Intel (Nehalem-Architektur von 2010)

Nutzung der Die-Fläche bei einem Mehrkernprozessor



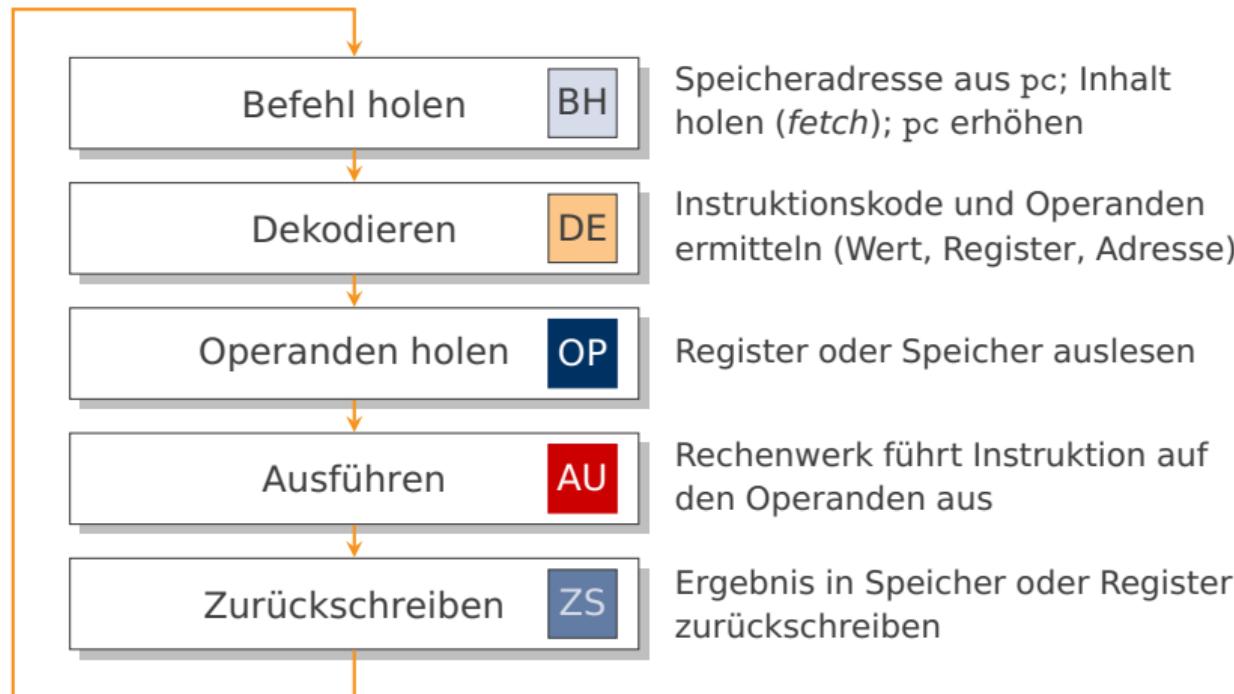
nach Hennessy & Patterson 2012, S. 29; Bild: Intel (Nehalem-Architektur von 2010)

Gliederung

1. Leistungsmessung
2. Caches: Optimierung des Speicherzugriffs
3. **Pipelines: Optimierung der CPU-Nutzung**

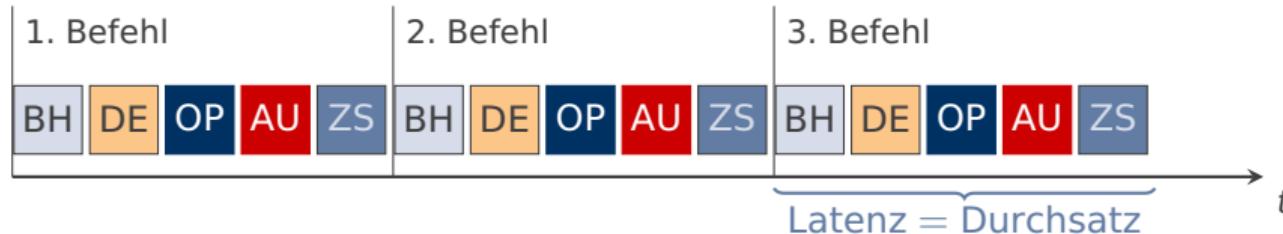
Maschinenbefehlszyklus

Modellablauf in Steuer- und Rechenwerk pro Befehl (hier: RISC)



Prinzip des Pipelinings

Sequenzielle Abarbeitung der Teilschritte



Überlappendes Abarbeiten



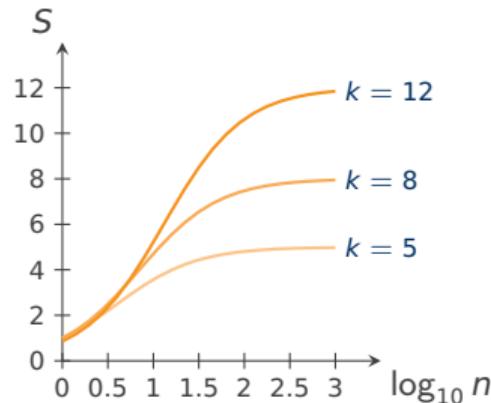
Pro Taktzyklus wird ein Maschinenbefehl abgeschlossen.

Theoretische Leistungssteigerung

Annahmen ideale k -stufige Pipeline, Programm mit n Befehlen,
Zykluszeit Δt , d. h. Taktfrequenz = $1/\Delta t$

Berechnung der **Beschleunigung**

$$S(k) = \frac{\text{Ausführungszeit ohne Pipeline}}{\text{Ausführungszeit mit Pipeline}} = \frac{n \cdot k \cdot \Delta t}{[n + (k - 1)] \cdot \Delta t} \quad (3)$$



$$\lim_{n \rightarrow \infty} S(k) = \frac{n \cdot k}{n + (k - 1)} = k \quad (4)$$

Gründe für Abweichungen in der Praxis

Strukturkonflikte

structural hazard

- Hardware unterstützt Befehlskombination nicht
- **Lösung:** Umordnen (durch Compiler), sonst Leertakte (NOPs)

Datenkonflikte

data hazard

- Operanden noch nicht verfügbar
- **Lösung:** dezidierte *Forwarding*-Logik, sonst wie oben

Steuerkonflikte

control hazard

- Geholter Befehl ist nicht der benötigte (nach Verzweigungen)
- **Lösung:** Sprungvorhersage (*branch prediction*), Anhalten

Operationen mit mehreren Zyklen

- Multiplikation, Division, Gleitkommaoperationen, Kontextwechsel
- **Lösung:** separate Gleitkomma-Pipeline, (Geduld)

Beispiel zur Vermeidung von Datenkonflikten

Aufgabe

$$a = b + c$$

$$d = e - f$$

Langsamer Kode	Schnellerer Kode
LDR r2, b	LDR r2, b
LDR r3, c	LDR r3, c
ADD r1, r2, r3	LDR r5, e
STR r1, a	ADD r1, r2, r3
LDR r5, e	LDR r6, f
LDR r6, f	STR r1, a
SUB r4, r5, r6	SUB r4, r5, r6
STR r4, d	STR r4, d

Annahme: Variablen a-f befinden sich im Speicher relativ zu sp oder pc adressierbar.

Nächste Woche

1. Klausurtermin **hilfsweise als Online-Prüfung:** 2. Februar 2022, 12:15–12:45 Uhr

Organisatorisches

- Die Prüfung besteht aus 15 Fragen in der Form der Proseminar-Tests.
- Gesamtbearbeitungszeit: 30 Minuten
- Zum Bestehen benötigen Sie 50 % der erreichbaren Punkte. Raten lohnt sich nicht.
- Die Prüfung findet im OLAT-Kurs der **2021W703063 VO Rechnerarchitektur** statt.
- Dort wird am Tag vor der Prüfung ein Probetest erscheinen, den alle korrekt angemeldeten KandidatInnen sehen. Bitte probieren Sie das aus.
- Probleme/Fragen ausschließlich an **rechnerarchitektur-informatik@uibk.ac.at**.
- Lesen Sie die Mitteilungen im OLAT-Kurs der Vorlesung.

Zulässige Hilfsmittel

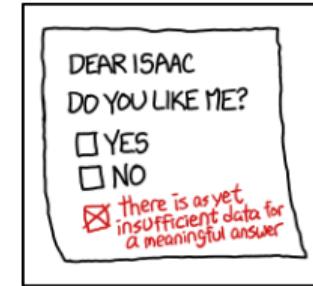
- Internet-Browser zum Zugriff auf OLAT
- ARM-Befehlsreferenz (verfügbar im OLAT)

Hinweise zur Präsenzklausur

Gilt nicht für die Online-Prüfung am ersten Termin !

Meine Prüfungsphilosophie

- Verständnis von Konzepten prüfen, nicht die Kapazität Ihres Kurzzeitgedächtnisses
- Ihre Antworten werden von Menschen korrigiert:
Falls Sie Annahmen zur Lösung brauchen,
schreiben Sie diese deutlich dazu.
- Raten oder Mehrdeutigkeit zählen sich nicht aus.
- Allein mit Auswendiglernen erreichen Sie ca. 25 % der Punkte.
- Ohne ARM-Kenntnisse verschenken Sie ca. 25 % der Punkte.



Weitere Informationen finden Sie auf unserer **Webseite** unter <http://informationsecurity.uibk.ac.at/teaching/>.

Hinweise zur Präsenzklausur (Forts.)

Gilt nicht für die Online-Prüfung am ersten Termin !

Organisatorisches

- Warten Sie vor dem Hörsaal bis Sie aufgerufen werden.
- Gemeinsamer Beginn, Bearbeitungszeit 75 Minuten
- Identitätsprüfung mit gültigem **Studentenausweis und weiterem amtlichen Lichtbildausweis**
- Alle zu bewertenden Antworten ausschließlich auf Klausurpapier
- Ergebnisbekanntgabe im OLAT anhand von Klausur-ID

Zulässige Hilfsmittel

- Stift, Geodreieck, nicht-programmierbarer Taschenrechner
- ARM-Referenz wird bereitgestellt (identisch wie im OLAT)

Sonst keine elektronischen Geräte am Platz !

(Ausschalten reicht nicht.)

Syllabus – Wintersemester 2021/22

06.10.21	1. Einführung
13.10.21	2. Kombinatorische Logik I
20.10.21	3. Kombinatorische Logik II
27.10.21	4. Sequenzielle Logik I
03.11.21	5. Sequenzielle Logik II
10.11.21	6. Arithmetik I
17.11.21	7. Arithmetik II
24.11.21	8. Befehlssatzarchitektur (ARM) I
01.12.21	9. Befehlssatzarchitektur (ARM) II
15.12.21	10. Ein-/Ausgabe
12.01.22	11. Prozessorarchitekturen
19.01.22	12. Speicher
26.01.22	13. Leistung
02.02.22	Klausur (1. Termin)